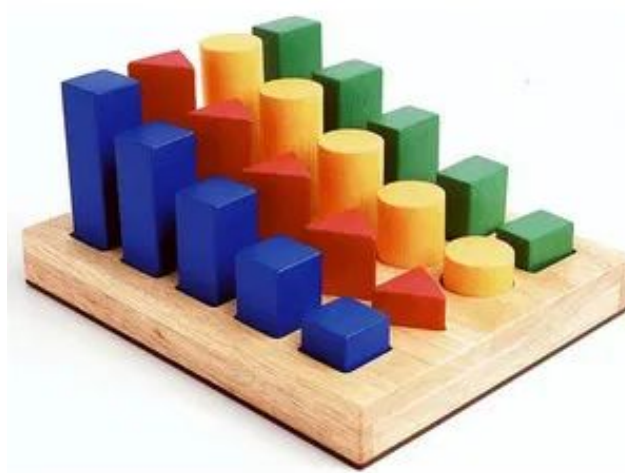


PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting



Edgar Perez Blanco
Bartomeu Perelló Comas
Group: **PAR2301**

PAR - GEI FIB
Spring 2019-2020

Contents

Introduction	4
1. Task decomposition analysis for Mergesort	5
1.1. Leaf task decomposition analysis with Tareador	5
Dependencies	7
Task Creation	8
Scalability	9
1.2. Tree task decomposition analysis with Tareador	11
Dependencies	12
Task Creation	13
Scalability	13
1.3. Comparison Leaf vs. Tree	16
Task Creation	16
Scalability	16
2. Shared-memory parallelisation with tasks	17
2.1. Leaf Strategy OpenMP Implementation	17
Code	17
Tracing	18
Scalability	20
2.2. Tree Strategy OpenMP Implementation	21
Code	21
Tracing	22
Scalability	24
2.3. Task cut-off mechanism	25
Code	25
Tracing	26
Using 24 threads (Optional 1)	28
3. Using OpenMP task dependencies	30
Using Dependencies	30
Code	30
Tracing	31
Scalability	32
Parallel Initialization (Optional 2)	33
Code	33
Tracing	34
Scalability	35
Conclusion	36

Introduction

This laboratory assignment is about divide and conquer parallelism, in this case we will be using mergesort as the algorithm to parallelize. Because of the divide and conquer, the strategies that will be used are tree and leaf. In addition we will also experiment with the cut-off mechanism and we will be using a different methods to deal with task dependencies.

1. Task decomposition analysis for Mergesort

There are tons of ways to parallelize a code. One of them consists in decomposing the code into tasks, and depending on the number of them and the order in which we execute them, we are dealing with a decomposition or another.

In order to inspect the existing dependencies in a sequential code divided into tasks we make use of `tareador`.

To help ourselves understanding which recursion level had been reached by each task, we have past the “depth” parameter to generate different types of tasks.

1.1. Leaf task decomposition analysis with Tareador

When dealing with recursive code, there are different ways to parallelize it. The one we are going to apply in this section is the Leaf one. This strategy consists in creating a task for each base case execution, which means that a single thread goes through the full recursion tree until reaches the leaves, where it creates tasks for each one.

The following code portion shows where we put the tareador start/end task directives to generate the task dependency graph:

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        sprintf(msg, "Basic_Merge_lvl_%d", depth);
        tareador_start_task(msg);
        basicmerge(n, left, right, result, start, length);
        tareador_end_task(msg);
    } else {
        // Recursive decomposition
        merge(n, left, right, result, start, length/2, depth+1);
        merge(n, left, right, result, start + length/2, length/2, depth+1);
    }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        multisort(n/4L, &data[0], &tmp[0], depth+1);
        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
    } else {
        // Base case
        sprintf(msg, "Basic_Sort_lvl_%d", depth);
        tareador_start_task(msg);
        basicsort(n, data);
        tareador_end_task(msg);
    }
}
```

As explained before, we added the depth parameter in order to classify the different created task by recursion level. In the task dependence below, the same node color means the same function and recursion depth.

Dependencies

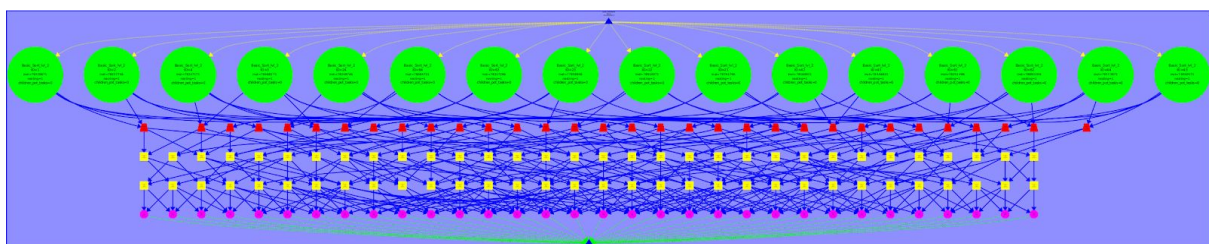
There exists some ordering constraints in terms of execution. If we try to parallelize with a leaf strategy, we just have to worry about the dependencies between the different base cases. This mergesort version is divided into three phases:

1. Sort 4 subsets of elements
2. Merge the first with the second and the third with the fourth subsets respectively, generating 2 new sorted subsets.
3. Merge the generated subsets in the previous step.

So, when parallelizing this code is compulsory to respect the following statements:

- Two portions of the vector must be “basicsorted” before they have been “basicmerged”.
- Two portions of the vector, must be “basicmerged” by the two first calls before they have been “basicmerged” by the third.

This described statements are the ones which have been drawn by taredor in the following referenced Task Dependence Graph:



Leaf Decomposition TDG

Task Creation

	MAIN_TAREADOR	Basic_Sort_lvl_2	Basic_Merge_lvl_4	Basic_Merge_lvl_5	Basic_Merge_lvl_6
CPU 1.1	1	16	32	64	32
Total	1	16	32	64	32
Average	1	16	32	64	32
Maximum	1	16	32	64	32
Minimum	1	16	32	64	32
StDev	0	0	0	0	0
Avg/Max	1	1	1	1	1

#tasks created at each recursion level

As we can see on the table above, the program just create tasks for computation nodes. The recursion level shown at the different tasks corresponds to the deepest level achieved by each initial call.

There are 16 executions of “basic sort” and the maximum recursion level achieved is 2 (starting from 0). This result is correct because we have 4 recursive calls to multisort inside of the multisort function (branching factor = 4) and $\log_4(16) = 2$, which corresponds to the maximum reached depth.

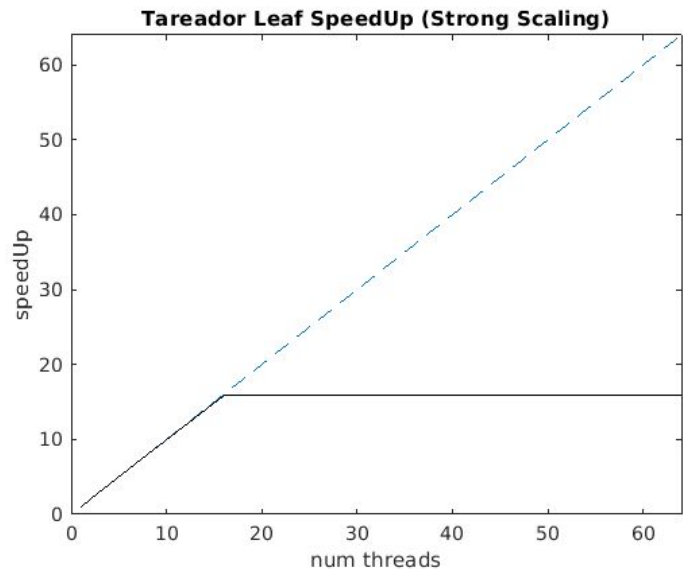
Then, as we call the merge function 3 times: 2 times with half* of the elements + 1 time with all* the elements; we have 32 + 32 “basicmerge” executions for the 2 first calls, and 64 executions for the last call. The last call to merge, reaches a deeper level due to treat double the elements of the previous ones at the first call.

** Of the treated portion at the corresponding call.*

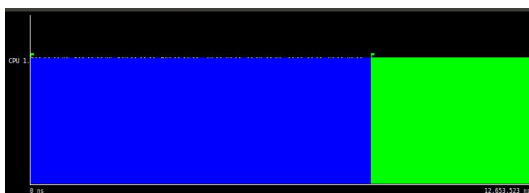
Scalability

In terms of scalability we have seen that the potential strong scaling is ideal until we reach the use of 16 threads, where the speedup becomes constant. We want to remark that as this is a simulation, the parallelization overheads are not considered, therefore, the obtained results are just achievable with an ideal machine.

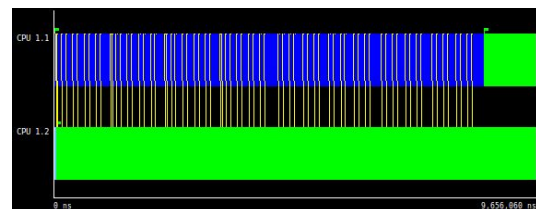
The limit is on the 16 threads due to the critical path. In this case, as there are 32768 elements, and the base case gets executed with 1024 elements. Therefore, there are $32768/1024 = 32$ sorted portions to merge. As each merge treat with 2 portions, the biggest number of merges than can be done in parallel are 16 tasks. So even if we use more than 16 threads, the execution time would not be reduced.



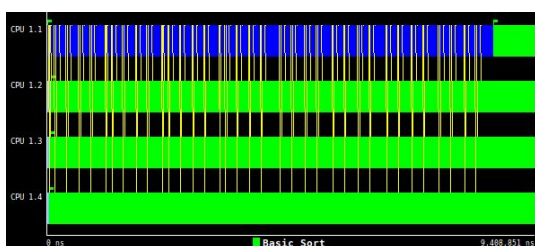
The following timeline pictures show the task creation part for simulations for diverse number of threads. We can appreciate how the main thread, the one who traversed the single construct instantiates all the tasks, giving work to the different threads:



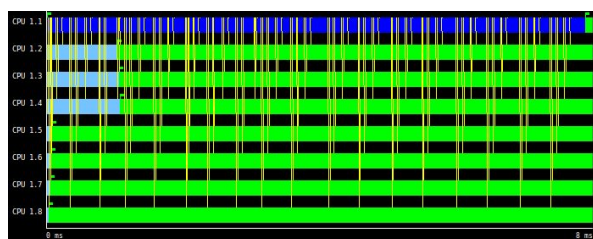
1 threads



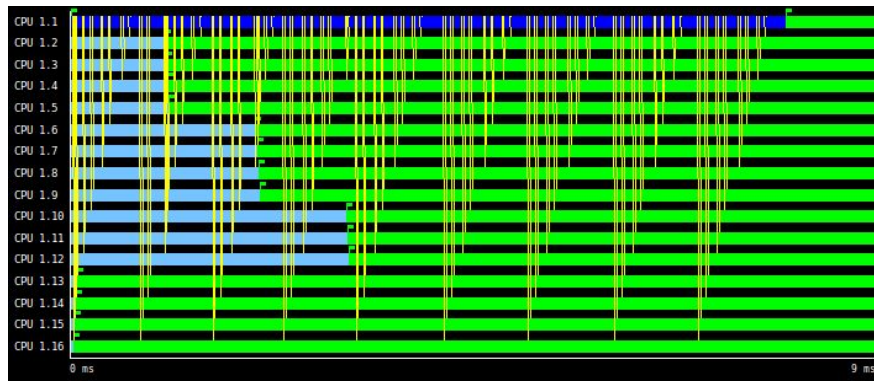
2 threads



4 threads



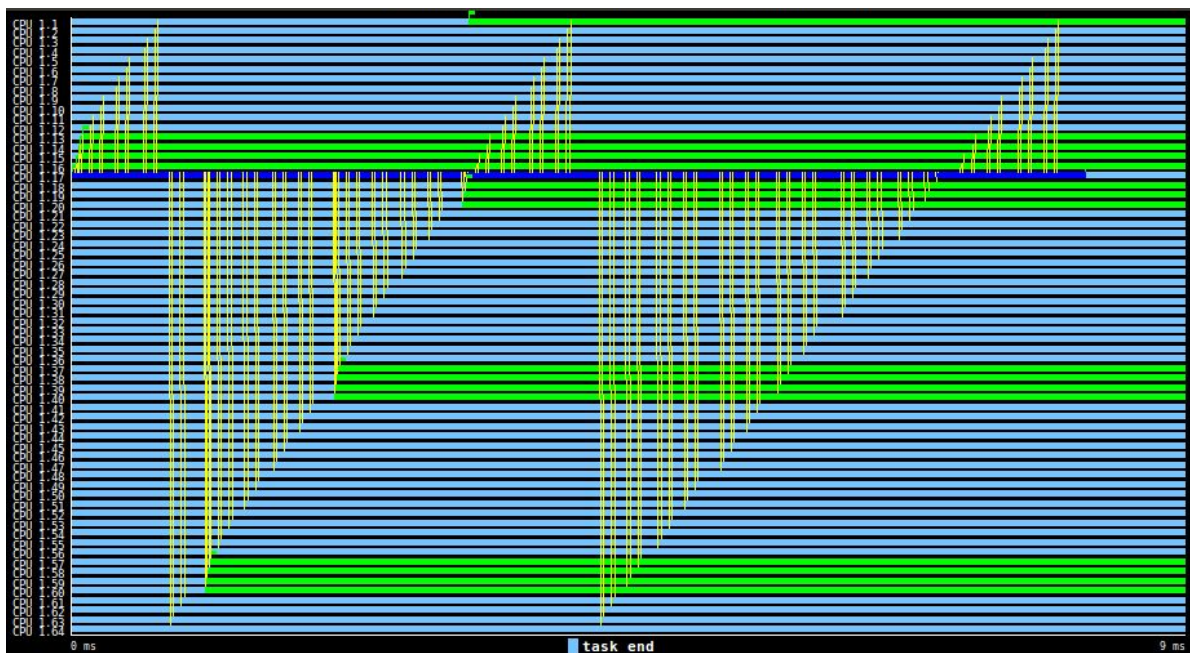
8 threads



16 threads



32 thread



64 threads

1.2. Tree task decomposition analysis with Tareador

Tree task decomposition is used when we have a huge amount of processors and want to exploit the parallelism due to its behaviour. A tree decomposition relies on creating a task each time we divide our problem in smaller ones producing way more tasks compared to the leaf one.

In order to make use of this decomposition we have created a task for each of the four multisort calls and one for each merge, in this case three and finally the merge function as well. The following code portion is the one used to do the analysis:

```
        sprintf(msg, "Merge_lvl_%d", depth);
        tareador_start_task(msg);
        merge(n, left, right, result, start, length/2, depth+1);
        tareador_end_task(msg);

        sprintf(msg, "Merge_lvl_%d", depth);
        tareador_start_task(msg);
        merge(n, left, right, result, start + length/2, length/2, depth+1);
        tareador_end_task(msg);
    }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {

        // Recursive decomposition
        sprintf(msg, "MultiSort_lvl_%d", depth);
        tareador_start_task(msg);
        multisort(n/4L, &data[0], &tmp[0], depth+1);
        tareador_end_task(msg);

        sprintf(msg, "MultiSort_lvl_%d", depth);
        tareador_start_task(msg);
        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
        tareador_end_task(msg);

        sprintf(msg, "MultiSort_lvl_%d", depth);
        tareador_start_task(msg);
        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
        tareador_end_task(msg);

        sprintf(msg, "MultiSort_lvl_%d", depth);
        tareador_start_task(msg);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
        tareador_end_task(msg);

        sprintf(msg, "Merge_lvl_%d", depth);
        tareador_start_task(msg);
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
        tareador_end_task(msg);

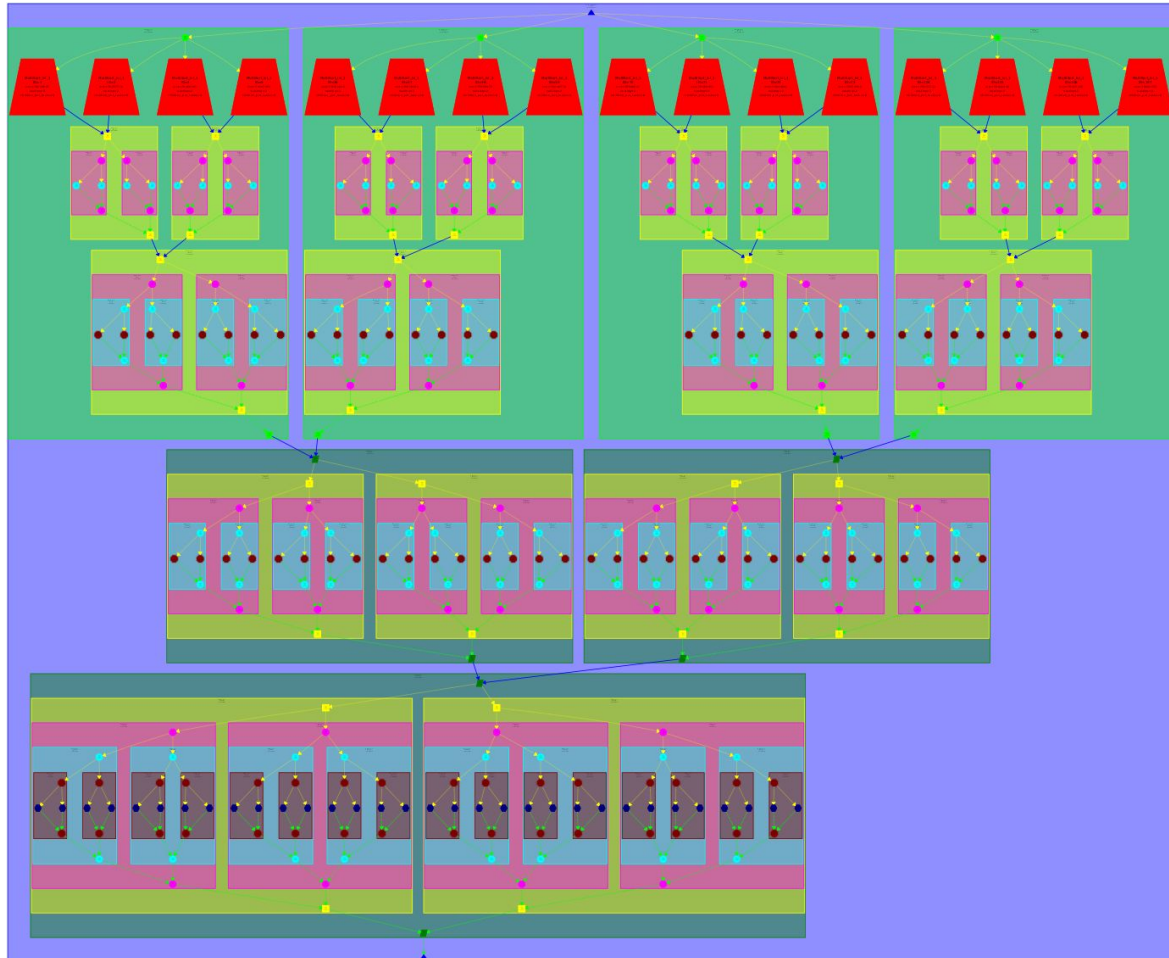
        sprintf(msg, "Merge_lvl_%d", depth);
        tareador_start_task(msg);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);
        tareador_end_task(msg);

        sprintf(msg, "Merge_lvl_%d", depth);
        tareador_start_task(msg);
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        tareador_end_task(msg);

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

Dependencies

Dependencies on this strategy is that all tasks created in a higher recursion level must be finished before the merge tasks are created, if not, we surely would be merging sections of the vector that are not yet sorted. Below we can see the task dependence graph as we take into account the recursion depth level.



Tree Decomposition TGD

Task Creation

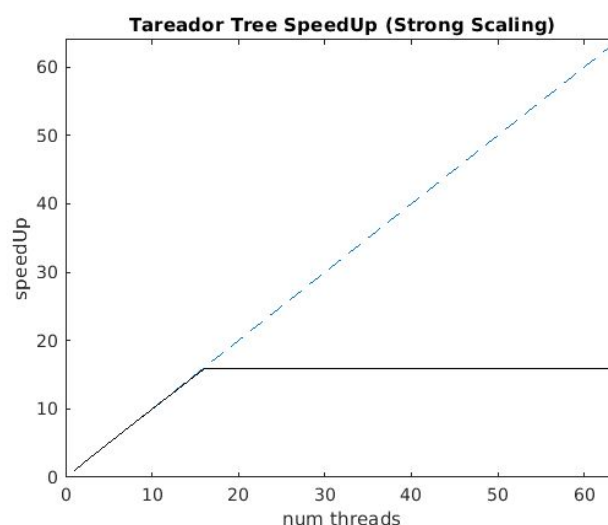
As we can appreciate on the chart below and the task dependency graph above we have 16 multisort tasks that end up computing which come from the initial call at the multisort, once those tasks are done, the program starts creating merge tasks until a fourth level of depth is reached, afterwards 64 merges start computing and 16 of them go one level deeper and create 32 more merge tasks that end up computing, so we end up with 96 merge tasks working.

	MAIN_TAREADOR	MultiSort_lvl_0	MultiSort_lvl_1	Merge_lvl_1	Merge_lvl_2	Merge_lvl_3	Merge_lvl_4	Merge_lvl_0	Merge_lvl_5
CPU 1.1	1	4	16	18	36	72	80	3	32
Total	1	4	16	18	36	72	80	3	32
Average	1	4	16	18	36	72	80	3	32
Maximum	1	4	16	18	36	72	80	3	32
Minimum	1	4	16	18	36	72	80	3	32
StDev	0	0	0	0	0	0	0	0	0
Avg/Max	1	1	1	1	1	1	1	1	1

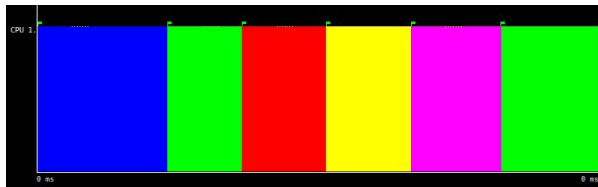
#tasks created at each recursion level

Scalability

If we check scalability with tareador means that we are able to see the maximum potential speedup we could achieve with an ideal system that can execute every task with no overheads involved, in our case the maximum speedup is 15.8 when 16 thread are reached so, even if we had that ideal system we could not get further this 15.8 speedup due to the size of our problem.



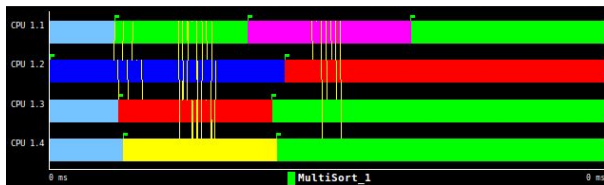
The following timeline pictures show the task creation part for simulations for diverse number of threads:



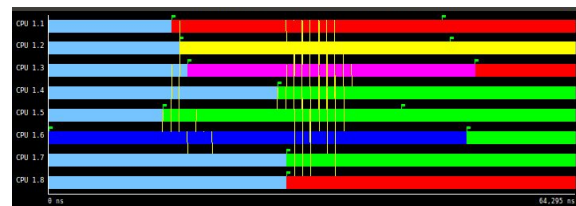
1 thread



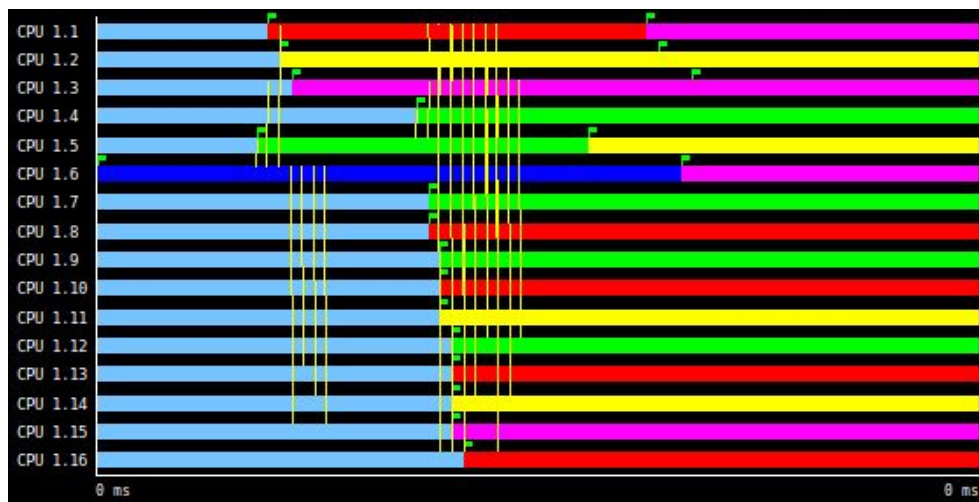
2 threads



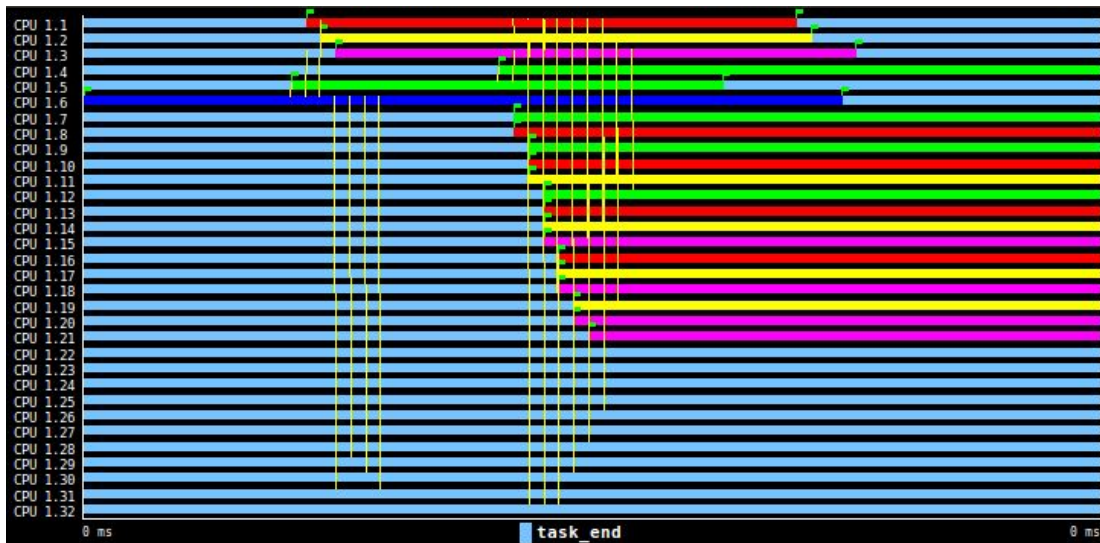
4 threads



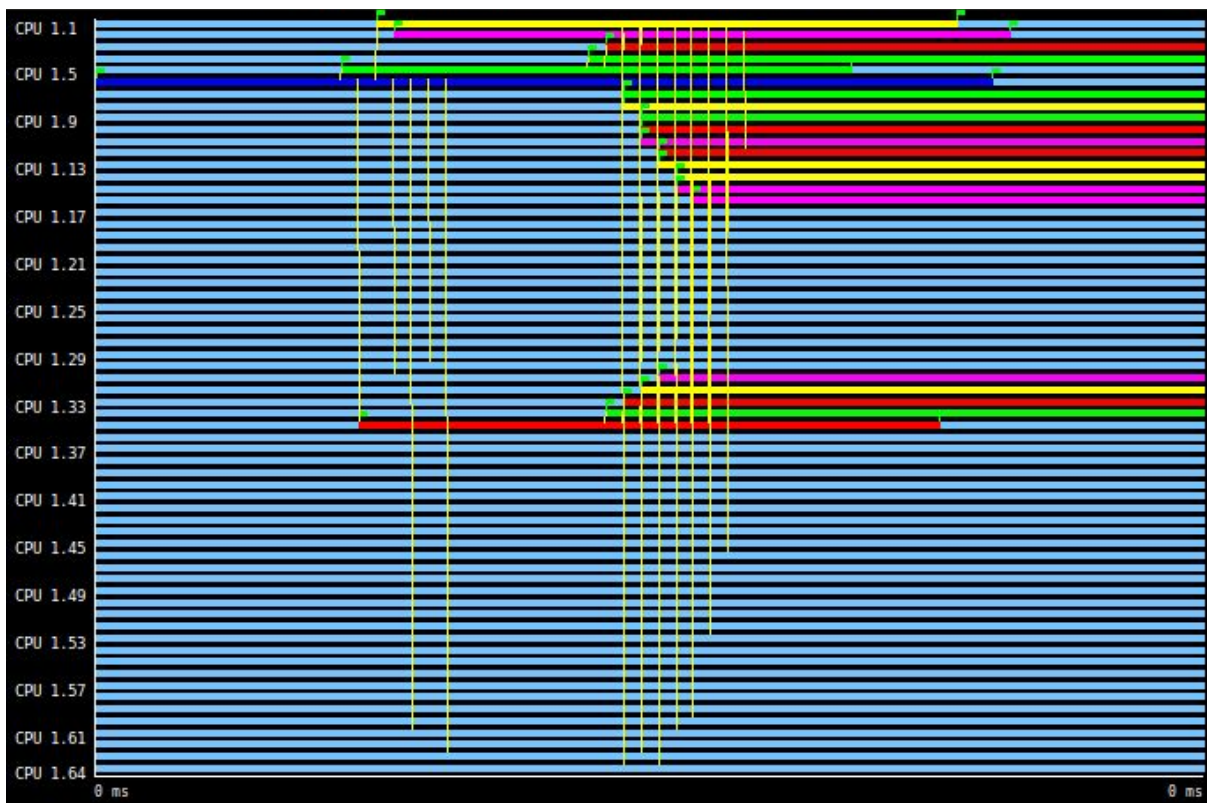
8 threads



16 threads



32 threads



64 threads

We can see how the main thread instantiates the first task but once reached that point, all the created tasks, assigned to different threads, instantiate new ones and wait them (synchronizations).

1.3. Comparison Leaf vs. Tree

Task Creation

Recursion Level	Leaf Decomposition		Tree Decomposition	
	Computing	Internal	Computing	Internal
0	0	1 (main)	0	7 + 1 (main)
1	0	0	16	18
2	16	0	0	36
3	32	0	32	40
4	64	0	64	16
5	32	0	32	0
TOTAL	16 + 128	1	16 + 128	118

Having the same number of computing tasks on both cases its normal because the recursion tree is the same-

Scalability

num_threads	leaf_time	tree_time	leaf_SpeedUp	tree_SpeedUp
1	1.2634e+06	1.2634e+06	1	1
2	6.3169e+05	6.3169e+05	2	1.9999
4	3.1585e+05	3.1585e+05	3.9998	3.9998
8	1.5838e+05	1.5882e+05	7.9766	7.9544
16	79789	79787	15.834	15.834
32	79746	79745	15.842	15.842
64	79746	79745	15.842	15.842

As we can see in the table above, taredor suggests that there is no difference between a leaf and a tree task decomposition. This is happening because taredors does not consider any task creation overheads nor synchronization, so, it is useful for studying dependencies but not for simulating timings.

2. Shared-memory parallelisation with tasks

2.1. Leaf Strategy OpenMP Implementation

Code

The omp implementation of the leaf strategy is pretty similar to the treader code structure. Where there was an “Start Task” now there is a `#pragma omp task`. The major difference is that now, we must insert some synchronization clauses (`taskwait`) in order to respect the task ordering constraints*.

* Previously described at *1.1.Dependencies*.

All the tasks are dealing with shared data because the way we call to the recursive functions, we are delimiting the memory zone each function has to compute, so between same function calls, there exists no dependencies. Moreover, with the use of `taskwait` we are ensuring that tasks which have to access to the same memory zone will not overlap.

The following code portion shows exactly where the pragmas are located at:

```
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);
    }
}

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

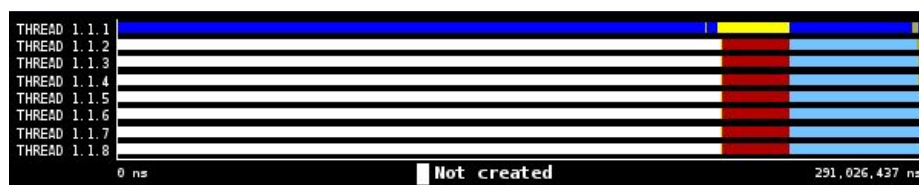
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        #pragma omp taskwait

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        #pragma omp task
        basicsort(n, data);
    }
}
```

Tracing

When we look for how much work has been done by each thread, we noticed that all the threads (except the main one) had been not created almost 90% of the time. This is due to the initial fully sequential part of the program plus the single threaded walking through the recursion tree. All the threads must wait to the main one to reach the last recursion level and generate tasks. Until that moment, they could not do any work.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	90.58 %	-	1.64 %	7.15 %	0.62 %	0.00 %
THREAD 1.1.2	1.69 %	89.74 %	8.47 %	0.00 %	0.10 %	-
THREAD 1.1.3	1.70 %	89.74 %	8.45 %	0.00 %	0.10 %	-
THREAD 1.1.4	1.29 %	89.77 %	8.86 %	0.00 %	0.07 %	-
THREAD 1.1.5	1.74 %	89.74 %	8.41 %	0.00 %	0.10 %	-
THREAD 1.1.6	1.85 %	89.76 %	8.27 %	0.00 %	0.11 %	-
THREAD 1.1.7	1.79 %	89.76 %	8.36 %	0.00 %	0.09 %	-
THREAD 1.1.8	1.72 %	89.80 %	8.37 %	0.00 %	0.11 %	-
Total	102.38 %	628.32 %	60.84 %	7.16 %	1.30 %	0.00 %
Average	12.80 %	89.76 %	7.60 %	0.89 %	0.16 %	0.00 %
Maximum	90.58 %	89.80 %	8.86 %	7.15 %	0.62 %	0.00 %
Minimum	1.29 %	89.74 %	1.64 %	0.00 %	0.07 %	0.00 %
StDev	29.40 %	0.02 %	2.26 %	2.37 %	0.17 %	0 %
Avg/Max	0.14	1.00	0.86	0.13	0.26	1

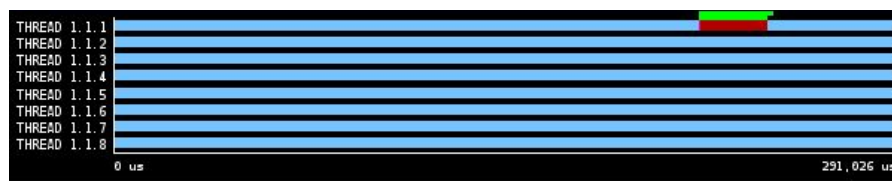


Parallel execution timeline

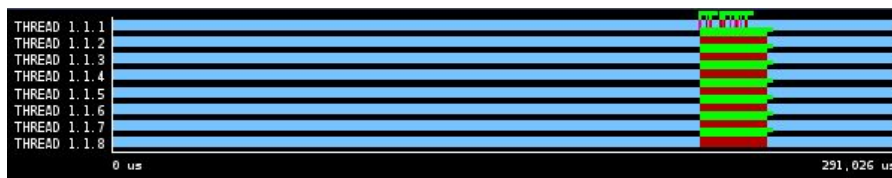
The table below shows how many tasks are executed by each thread. We consider that even though there exists a good load balancing once the tasks are created, the big initial sequential part ruins the global parallel performance.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	18	11,264
THREAD 1.1.2	1,721	-
THREAD 1.1.3	1,735	-
THREAD 1.1.4	957	-
THREAD 1.1.5	1,747	-
THREAD 1.1.6	1,839	-
THREAD 1.1.7	1,485	-
THREAD 1.1.8	1,762	-
Total	11,264	11,264
Average	1,408	11,264
Maximum	1,839	11,264
Minimum	18	11,264
StDev	588.68	0
Avg/Max	0.77	1

The following timelines show when the tasks were created and executed:

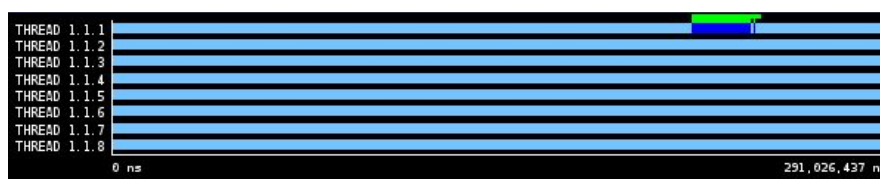


task instantiation



task execution

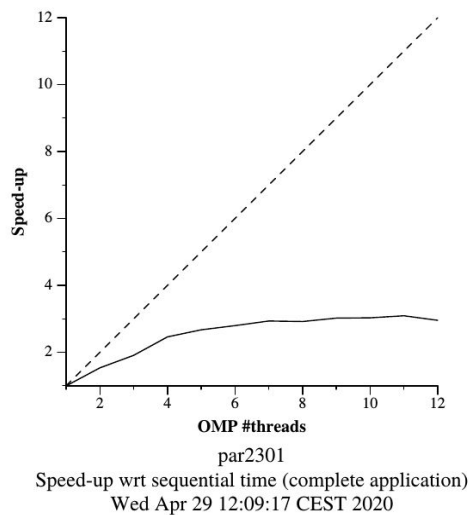
The taskwait execution works correctly. It only affects into the parallel region, and as there were much more tasks than taskwaits (just three) it does not affect to the parallel performance. We want to remark that the taskwait is not an optional part of the code, if we do not use it, we get wrong results.



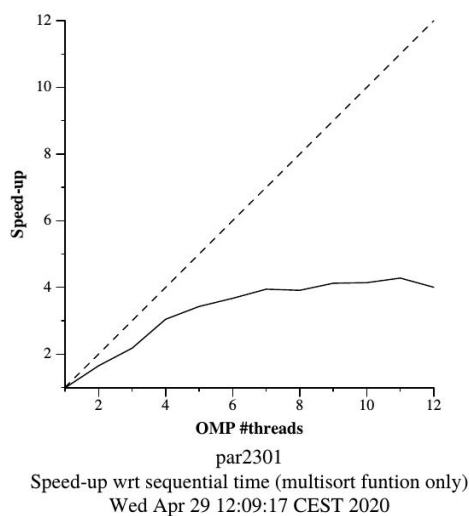
taskwait execution

Scalability

In terms of global strong scalability, we have not good news. Due to the little parallel fraction of the code, the application speed up would be mediocre no matter the parallelized portion speed up (this is the effect of Amdahl's law). The following plot shows how the application speed up almost can not reach a x3 improvement:



If we isolate the parallelized code (i.e. : the multisort code) while measuring, we can assess the improvements made in the parallelization. The plot below shows the multisort function speed up, which has an acceptable behaviour until the use of 4 threads, where the number of used threads vs. the extra obtained performance becomes not worth at all. We think it could be better, but due to the leaf strategy, the program cannot speedup more than what the results shown.



2.2. Tree Strategy OpenMP Implementation

Code

The key point when implementing the tree strategy in this case is to make sure each multisort recursive call has finished when we move to the merges so we don't start merging sections that are not sorted or even modify sections which are already sorted, in this case we have chosen to make two different taskgroup groups.

```
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);
    }
}

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);
        }

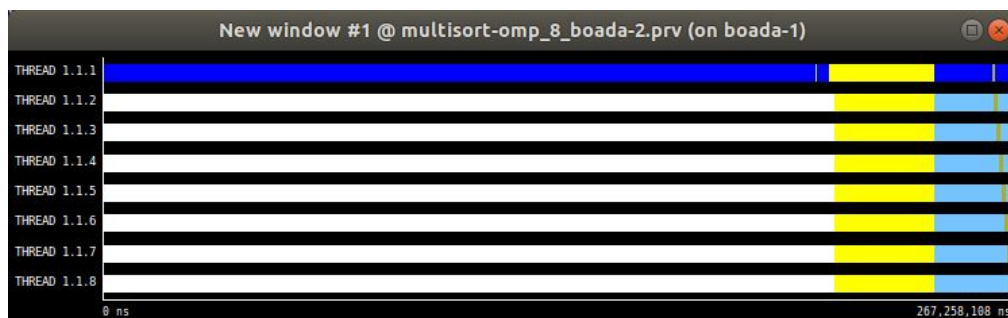
        #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Code of the tree implementation

Tracing

Once we finished the testing we found out similar results on running time compared to the leaf strategy, mostly all threads stay most of the time waiting for the main thread to run the sequential part of the program but once it hits the tasks every thread starts creating and executing tasks, unfortunately as the sequential code represents almost the whole execution we cannot appreciate the advantages of the tree strategy when using a high number of threads.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	90.90 %	-	4.59 %	4.30 %	0.20 %	0.00 %
THREAD 1.1.2	2.67 %	87.79 %	5.05 %	4.18 %	0.31 %	-
THREAD 1.1.3	2.38 %	87.79 %	5.38 %	4.15 %	0.30 %	-
THREAD 1.1.4	2.68 %	87.76 %	5.15 %	4.06 %	0.34 %	-
THREAD 1.1.5	2.61 %	87.76 %	5.09 %	4.20 %	0.35 %	-
THREAD 1.1.6	2.51 %	87.82 %	5.28 %	4.10 %	0.29 %	-
THREAD 1.1.7	2.62 %	87.79 %	5.10 %	4.14 %	0.35 %	-
THREAD 1.1.8	2.61 %	87.85 %	5.05 %	4.24 %	0.25 %	-
Total	108.99 %	614.56 %	40.69 %	33.37 %	2.38 %	0.00 %
Average	13.62 %	87.79 %	5.09 %	4.17 %	0.30 %	0.00 %
Maximum	90.90 %	87.85 %	5.38 %	4.30 %	0.35 %	0.00 %
Minimum	2.38 %	87.76 %	4.59 %	4.06 %	0.20 %	0.00 %
StDev	29.21 %	0.03 %	0.22 %	0.07 %	0.05 %	0 %
Avg/Max	0.15	1.00	0.95	0.97	0.85	1

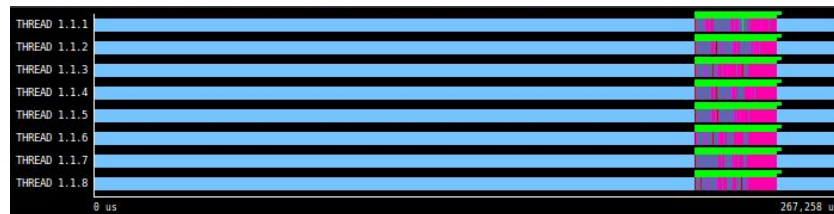


Parallel execution timeline

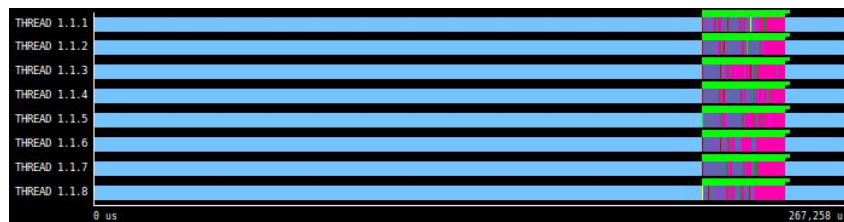
At least the task execution and creation are balanced in the execution while in the leaf we appreciated that only one thread created the tasks and almost executed any task.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	2,110	2,135
THREAD 1.1.2	2,922	2,911
THREAD 1.1.3	2,430	2,434
THREAD 1.1.4	3,002	3,005
THREAD 1.1.5	2,970	2,961
THREAD 1.1.6	2,308	2,300
THREAD 1.1.7	2,991	2,984
THREAD 1.1.8	2,088	2,091
Total	20,821	20,821
Average	2,602.62	2,602.62
Maximum	3,002	3,005
Minimum	2,088	2,091
StDev	382.66	376.05
Avg/Max	0.87	0.87

To complement this explanation we can also see the timeline referring to the instantiation and execution.

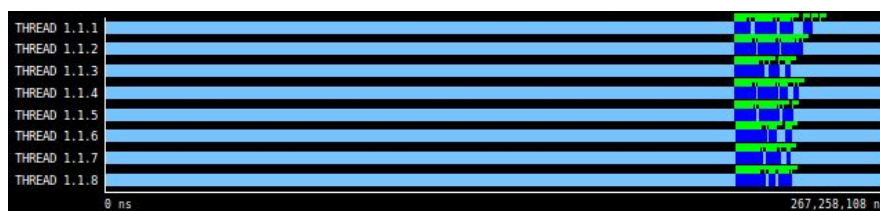


task instatiation



task execution

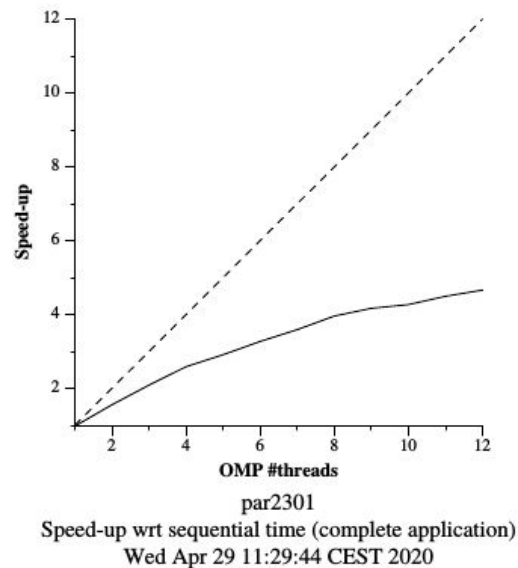
In this case we must use the taskgroup construct because in the tree strategy we are creating tasks which try to sort smaller parts of the vector so we cannot execute one merge until its children have finished because we could be merging parts which are not already sorted. As the tasks are distributed between all threads the taskgroups will also be spreaded.



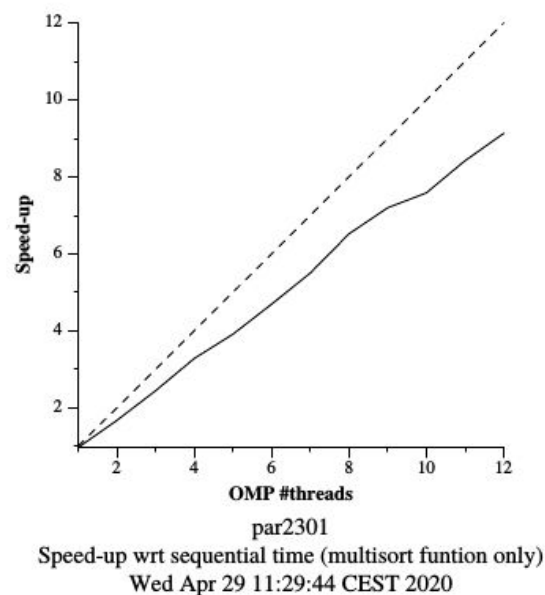
taskgroup constructs

Scalability

If we take a look at the whole program scalability we can see an improvement compared to the previous execution but the results are not good either, for the same reasons as before we cannot achieve a very good level of speedup, the only reason we get higher number is due to the tree decomposition that benefits from having high number of threads working.



At least if we isolate the parallel code from the sequential code we can see the the advantage of the tree parallelization over the leaf, the speedup improvement is hughe. That's the reason behind the whole execution having higher speedups.



2.3. Task cut-off mechanism

Code

In order to implement the cut-off mechanism we have had to modify a bit the code to make sure tasks are only generated on a recursion level previous than the specified by the user.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition

        #pragma omp task final (depth >= CUTOFF)
        merge(n, left, right, result, start, length/2, depth+1);
        #pragma omp task final (depth >= CUTOFF)
        merge(n, left, right, result, start + length/2, length/2, depth+1);
    }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if(!omp_in_final()){

            #pragma omp taskgroup
            {
                #pragma omp task final (depth >= CUTOFF)
                multisort(n/4L, &data[0], &tmp[0], depth+1);
                #pragma omp task final (depth >= CUTOFF)
                multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
                #pragma omp task final (depth >= CUTOFF)
                multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
                #pragma omp task final (depth >= CUTOFF)
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
            }

            #pragma omp taskgroup
            {
                #pragma omp task final (depth >= CUTOFF)
                #pragma omp task depend(in: )
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
                #pragma omp task final (depth >= CUTOFF)
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);
            }

            #pragma omp task final (depth >= CUTOFF)
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }

    } else {

        multisort(n/4L, &data[0], &tmp[0], depth+1);
        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

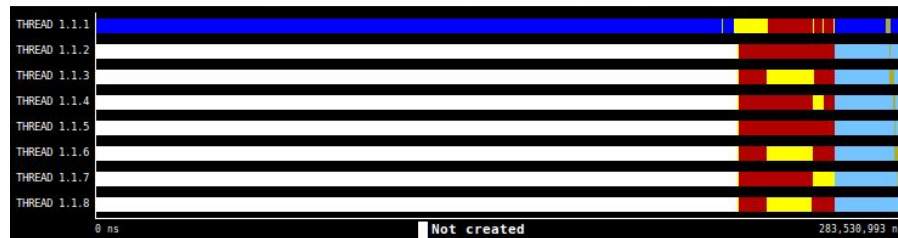
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
    }

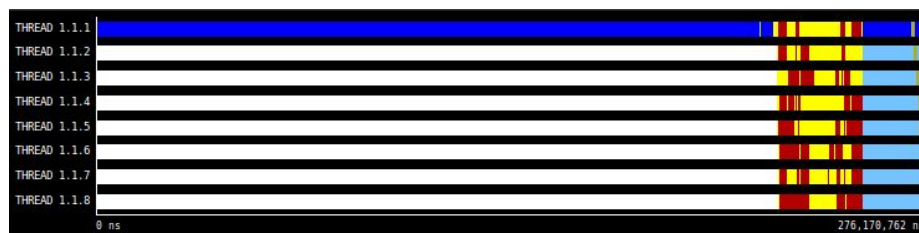
} else {
    // Base case
    basicsort(n, data);
}
```

Tracing

If we take a look at both time execution charts for two recursion levels 0 and 1 respectively, the level 0 has a worse execution time because when it hits the parallel region it doesn't create a high amount of tasks so most of time threads are doing nothing or syncing but when we increase the recursion level, threads start working more time so the execution time is reduced.



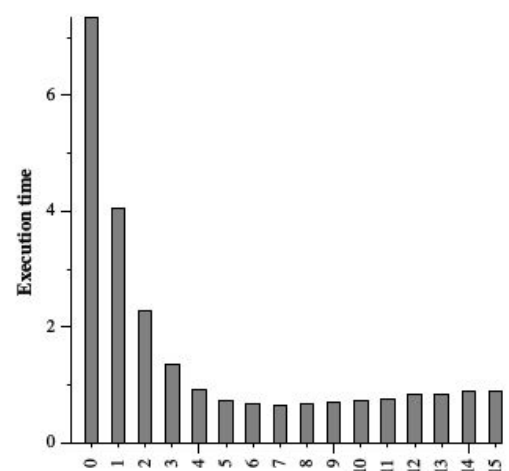
Recursion level 0 time chart



Recursion level 1 time chart

Finally as we keep increasing the recursion level, thread stay more time working so we get better results until we hit a bottleneck produced by the overheads of creating too much tasks for the amount of threads we have. We can see this behavior on the next plot.

Our best case is with a recursion level of seven, and it's almost the same than the eight, here we have the results of both just to compare the differences.

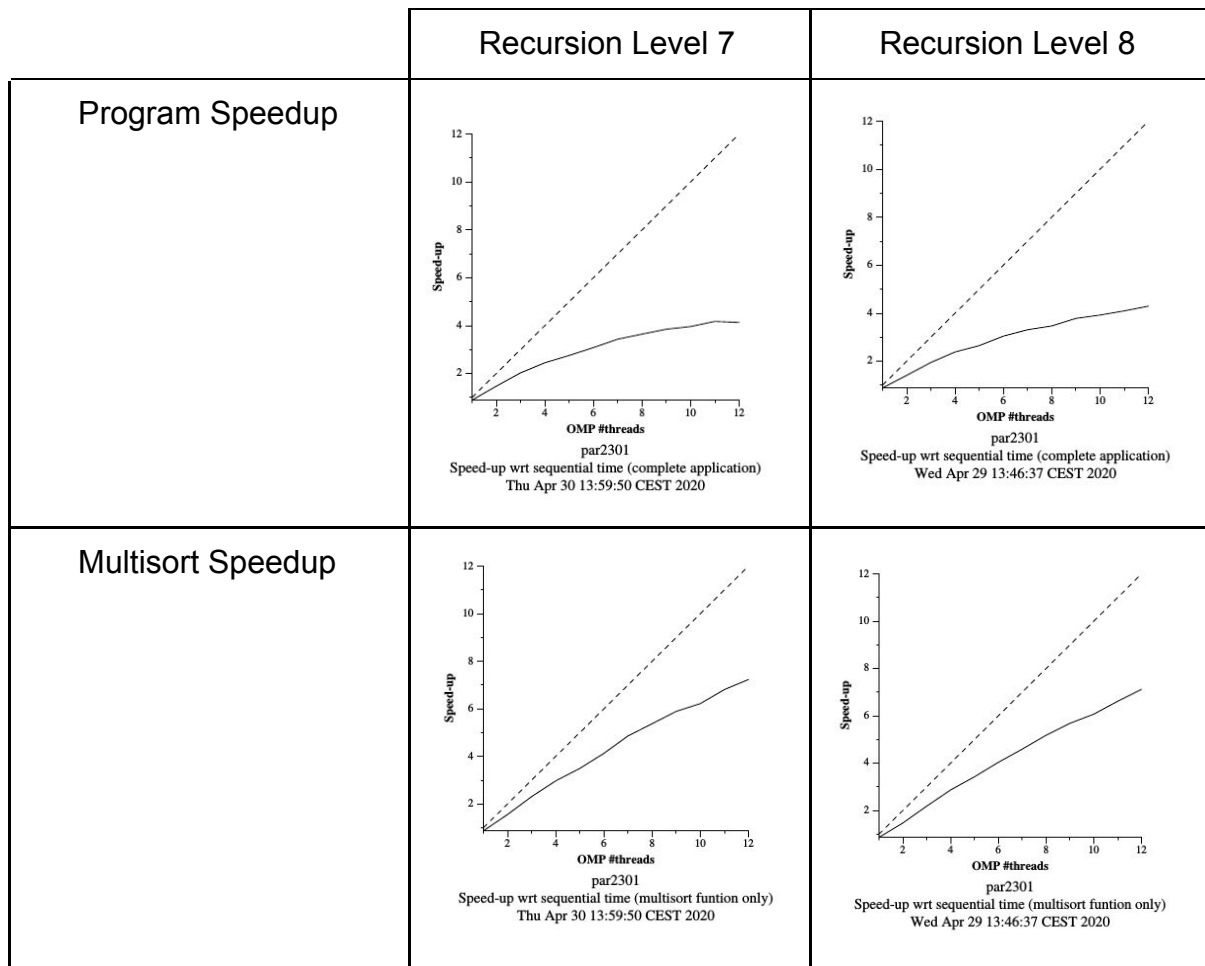


Recursion size

par2301

Average elapsed execution time (multisort only)

Wed Apr 29 13:40:09 CEST 2020



As we can see it's pretty hard to see a difference between both levels, one strange thing is that the level 8 has a bit more sequential advantage on the speedup, probably due to external factors of the execution because when we take a look at the actual numbers on the multisort speedup the seventh level always has a small lead.

```

1 .86934073265749502324
2 1.55343847801800717243
3 2.30608151996580396152
4 2.96851348052921763984
5 3.49333736066790188242
6 4.12696996998265499121
7 4.86013549686130522328
8 5.37248881356784340673
9 5.89190651433198985138
10 6.21933824282779120643
11 6.81006201296686058651
12 7.23331340244275627296

```

Level 7 Speedups

```

1 .86451444791697113162
2 1.47249225389783545287
3 2.18434457130507060716
4 2.86800338008994012468
5 3.42366318060615522100
6 4.03527656839846813192
7 4.59199051665145092415
8 5.17884767441332404214
9 5.68520282879602235873
10 6.07135619088182576443
11 6.6144533558467186474
12 7.11990197812717869741

```

Level 8 Speedups

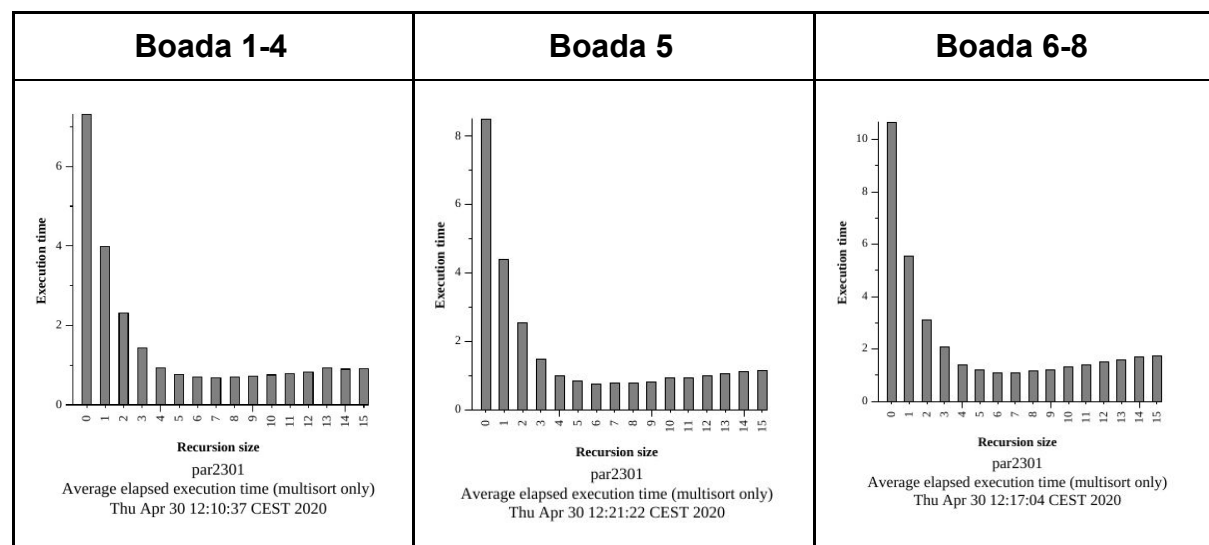
Using 24 threads (Optional 1)

In spite of having just 12 physical cores (in boada 1-4), due to the hyperthreading, each core can manage up to 2 threads. In practise, we can say that we can use 24 logical cores.

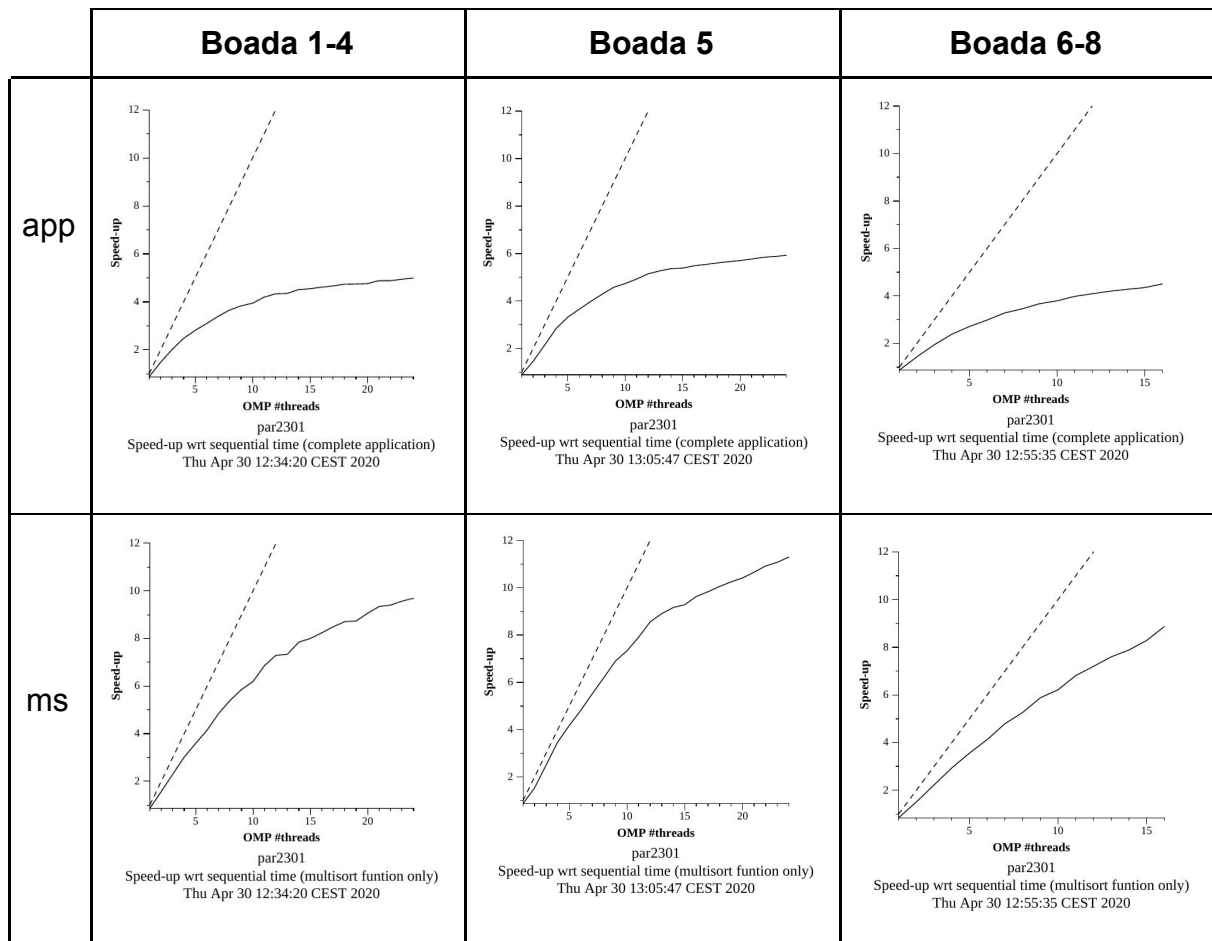
We have been testing in different boada node types: boada 1-4, boada 5 and boada 6-8. In boada 1 to 4 and 5, we can use up to 24 threads, meanwhile in boada 6 to 8, due to the hyperthreading absence, we can use up to 16 threads. The chart below from the first laboratory report shows the different associated architecture:

	boada-1 to 4	boada-5	boada-6 to 8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1

We had executed the optimal cutoff level exploration on all node types, and we have obtained the following results:



The minimum values are located between 6 and 7 recursion depth levels. As the performance differences were very little, we decided to use the level 6 in all nodes for the scalability study.



As we can see, even though we are using all the available threads, the scalability grows, but not as fastest as from 1 to 12 threads. As more threads we use, bigger the number of synchronizations is and, therefore, bigger the overheads are. Another thing important to mention is that the best results were obtained in a node from Boada 5, which we think is due to having faster cores than the others.

3. Using OpenMP task dependencies

Using Dependencies

Code

The changes we have made to the code are removing the taskgroup and put at every multisort a depend of their own data[] and afterwards on the next two merges put two depends in, the first one makes sure the sorting on the section it merges are done so does the second, finally after those two we put a taskwait to make sure we do not advance on the next merge while any or both of the previous one haven't finished.

It's easy to write this new addition because the elements that go on the clauses are the variables that are required at each call of the function merge which is pretty simple to understand.

```
void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if(!omp_in_final()){

            #pragma omp task final (depth >= CUTOFF) depend(out: data[0])
            multisort(n/4L, &data[0], &tmp[0],depth+1);
            #pragma omp task final (depth >= CUTOFF) depend(out: data[n/4L])
            multisort(n/4L, &data[n/4L], &tmp[n/4L],depth+1);
            #pragma omp task final (depth >= CUTOFF) depend(out: data[n/2L])
            multisort(n/4L, &data[n/2L], &tmp[n/2L],depth+1);
            #pragma omp task final (depth >= CUTOFF) depend(out: data[3L*n/4L])
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],depth+1);

            #pragma omp task final (depth >= CUTOFF) depend(in: data[0], data[n/4L])
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,depth+1);
            #pragma omp task final (depth >= CUTOFF) depend(in: data[n/2L], data[3L*n/4L])
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,depth+1);

            #pragma omp taskwait

            #pragma omp task final (depth >= CUTOFF)
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,depth+1);

            #pragma omp taskwait

        }

        else {
            multisort(n/4L, &data[0], &tmp[0],depth+1);
            multisort(n/4L, &data[n/4L], &tmp[n/4L],depth+1);
            multisort(n/4L, &data[n/2L], &tmp[n/2L],depth+1);
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],depth+1);

            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,depth+1);
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,depth+1);

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,depth+1);

        }

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


Tracing

To see if we got an improvement we will compare it to the best performer with the taskgroup construct which was the one with a cutoff level of 7, also this execution has been done with a recursion level of 7:

If we take a look at the state profiles we see a clear difference between both, the new one has its other seven threads running more time probably giving it overall a better execution time and speed up, while have the same number of total tasks.

2D profile (
	Running	Not created
THREAD 1.1.1	91.21 %	-
THREAD 1.1.2	2.67 %	88.12 %
THREAD 1.1.3	2.56 %	88.02 %
THREAD 1.1.4	2.48 %	88.03 %
THREAD 1.1.5	2.69 %	87.97 %
THREAD 1.1.6	2.62 %	87.99 %
THREAD 1.1.7	2.43 %	88.09 %
THREAD 1.1.8	2.47 %	88.15 %
Total	109.12 %	616.38 %
Average	13.64 %	88.05 %
Maximum	91.21 %	88.15 %
Minimum	2.43 %	87.97 %
StDev	29.32 %	0.06 %
Avg/Max	0.15	1.00

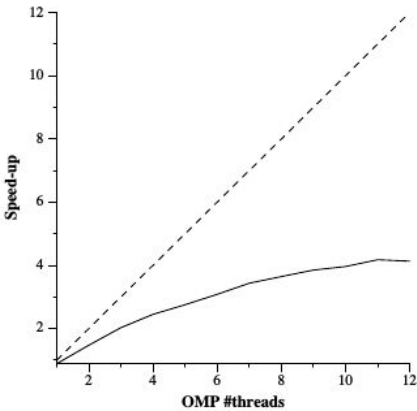
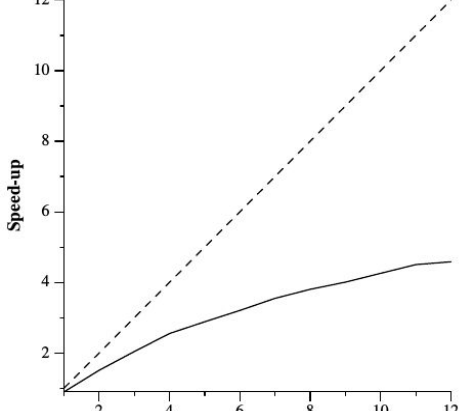
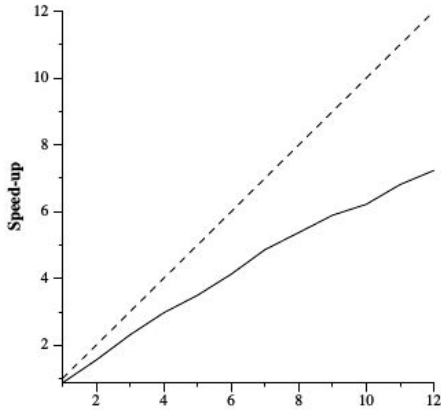
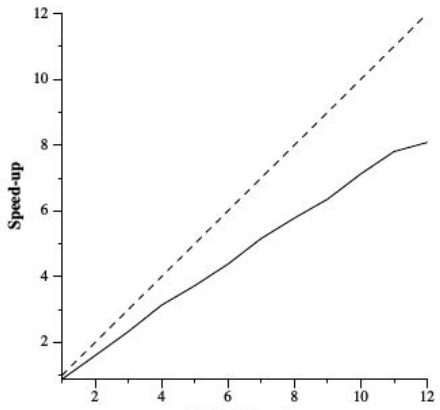
Taskgroup Based Profile

2D profile (
	Running	Not created
THREAD 1.1.1	89.14 %	-
THREAD 1.1.2	3.57 %	84.74 %
THREAD 1.1.3	3.60 %	84.70 %
THREAD 1.1.4	3.53 %	84.69 %
THREAD 1.1.5	3.70 %	84.68 %
THREAD 1.1.6	3.57 %	84.74 %
THREAD 1.1.7	3.48 %	84.66 %
THREAD 1.1.8	3.49 %	84.95 %
Total	114.06 %	593.16 %
Average	14.26 %	84.74 %
Maximum	89.14 %	84.95 %
Minimum	3.48 %	84.66 %
StDev	28.30 %	0.09 %
Avg/Max	0.16	1.00

Depend Based Profile

Scalability

By the same reason as before, the new version with depends has a better speedup when we take a look at the multisort plots, the whole execution one look almost the same due to the improvement be not that high.

	without dependencies	with dependencies
app	 <p>par2301 Speed-up wrt sequential time (complete application) Thu Apr 30 13:59:50 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time (complete application) Tue May 5 12:04:52 CEST 2020</p>
ms	 <p>par2301 Speed-up wrt sequential time (multisort funtion only) Thu Apr 30 13:59:50 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time (multisort funtion only) Tue May 5 12:04:52 CEST 2020</p>

Parallel Initialization (Optional 2)

Code

The initialization part is divided into two parts: `initialize` and `clear`.

- The `clear` function was the easiest function to parallelize because it just consists in set to 0 the whole vector. To do this, we just included a `#pragma omp parallel for` with an static scheduling, because the problem is embarrassingly parallel and all the iterations have the same cost.

```
static void initialize(long length, T data[length]) {  
    long R[10] = {rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand(), rand()};  
    long i;  
    #pragma omp parallel for schedule(static)  
    for (i = 0; i < length; i++)  
        data[i] = (R[i%10] * i * 104723L) % N;  
}
```

- The `initialize` function have been re-written in order to generate random numbers in a fastest way, and avoiding dependencies between different iterations. We decided to generate ten random numbers and saving them into a vector. We also adapted the previous number generator expression to access to this new read-only vector instead to previous positions of the written-vector (avoiding dependencies). Once we converted our full sequential loop into an embarrassingly parallel one with the same cost for all the iterations, we parallelized it as the `clear` function, with a `#pragma omp parallel for` with again an static scheduling.

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

Tracing

As we can see in the following captures, the number of task has increased as well as the parallel fraction. The parallel threads' running time has increased 1% in front of the previous version.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	90.49 %	-	4.84 %	4.26 %	0.42 %	0.00 %
THREAD 1.1.2	3.12 %	86.89 %	5.31 %	4.18 %	0.50 %	-
THREAD 1.1.3	3.12 %	86.92 %	5.30 %	4.18 %	0.48 %	-
THREAD 1.1.4	3.11 %	86.92 %	5.31 %	4.18 %	0.48 %	-
THREAD 1.1.5	3.17 %	86.87 %	5.31 %	4.14 %	0.51 %	-
THREAD 1.1.6	3.09 %	86.93 %	5.30 %	4.22 %	0.46 %	-
THREAD 1.1.7	3.22 %	86.99 %	5.32 %	3.99 %	0.48 %	-
THREAD 1.1.8	3.09 %	86.00 %	5.32 %	4.10 %	0.49 %	-
Total	112.41 %	608.50 %	42.02 %	33.24 %	3.82 %	0.00 %
Average	14.05 %	86.93 %	5.25 %	4.15 %	0.48 %	0.00 %
Maximum	90.49 %	86.00 %	5.32 %	4.26 %	0.51 %	0.00 %
Minimum	3.09 %	86.87 %	4.84 %	3.99 %	0.42 %	0.00 %
StDev	28.89 %	0.04 %	0.16 %	0.08 %	0.03 %	0 %
Avg/Max	0.16	1.00	0.99	0.98	0.94	1

Parallel execution timing

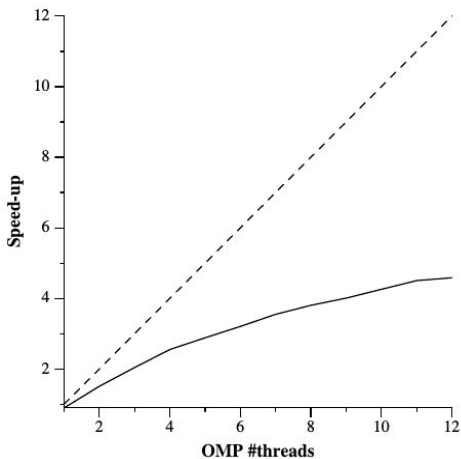
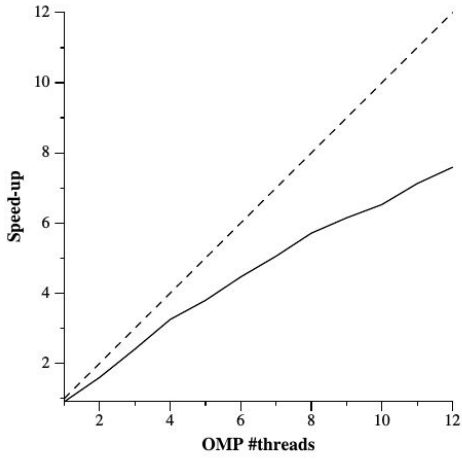
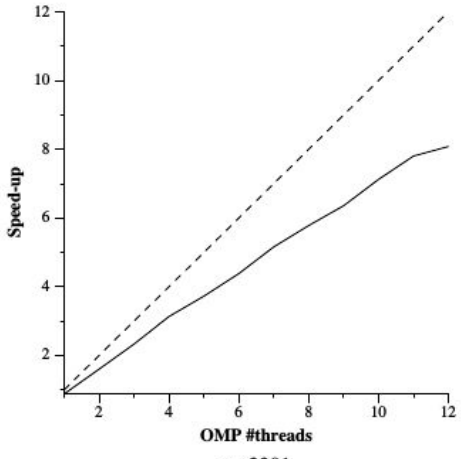
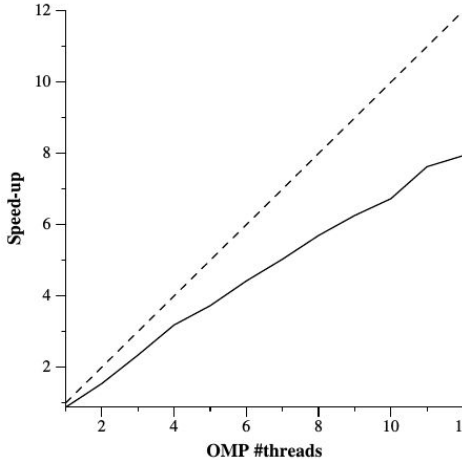
	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	2,802	2,810
THREAD 1.1.2	2,477	2,478
THREAD 1.1.3	2,669	2,667
THREAD 1.1.4	2,699	2,695
THREAD 1.1.5	2,562	2,579
THREAD 1.1.6	2,403	2,387
THREAD 1.1.7	2,676	2,679
THREAD 1.1.8	2,533	2,526
Total	20,821	20,821
Average	2,602.62	2,602.62
Maximum	2,802	2,810
Minimum	2,403	2,387
StDev	122.96	127.46
Avg/Max	0.93	0.93

Executed/Instantiated task counters

Scalability

The strong scaling study was which changed our mind about the improvements. The application speedup plot is now way similar to the multisort one, which means that the initialization overhead of the application has been almost reduced.

The following table compares the scalability plots of application and multisort for each version, the ones with sequential initialization and the ones with parallelized initialization:

	sequential Initialization	parallelized Initialization
app	 <p>par2301 Speed-up wrt sequential time (complete application) Tue May 5 12:04:52 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time (complete application) Tue May 5 13:17:10 CEST 2020</p>
ms	 <p>par2301 Speed-up wrt sequential time (multisort funtion only) Tue May 5 12:04:52 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time (multisort funtion only) Tue May 5 13:17:10 CEST 2020</p>

Conclusion

With this report we learnt a lot about the parallelization of recursive programs. We have seen how the tree parallelization strategy is a good choice for big recursive executions and also how the Amdahl's law is always present. Not only is important to parallelize the algorithm part, but also the initialization if possible.

As always, the overheads have played an important role, provoking big differences between the simulations and the obtained results.

Finally, we would like to remark the importance of the hardware when executing and evaluating a program. Differences between CPUs, or memory system can make the difference.