LAB 4 DELIVERABLE

PAR 2108

Raül Montoya Pérez
Miguel Ángel Álvarez Vázquez

Index

Introduction ……. 1

Analysis with Tareador ……. 2

Parallelization and performance analysis with tasks …….. 7

Parallelization and performance analysis with dependent tasks ……. 12

Optional …… 14

Conclusions ……. 17

Introduction

In this deliverable, we will be showing the effects of parallelization in a tree-like algorithm, how can it be parallelized and how it affects to the different execution times.

For these concrete laboratory sessions, we will be working with the well known Multisort algorithm, a tree-algorithm. How is it a tree-algorithm? This is due to it using a “Divide and conquer” strategy, the first call to the multisort, divides its problem in at least, two parts. With this strategy, we can work with a lot of small and easiest problems. When each part is executed, the solutions are combined to give a solution to the original problem, in our case, when all the parts are sorted then can merge in order to combine both parts. This recursivity generates a tree with its different calls, being the root the initial call, each branch a recursive decomposition of the problem until we arrive at the leafs, the base case, where we can execute and cannot make it smaller or easier.

Analysis with Tareador

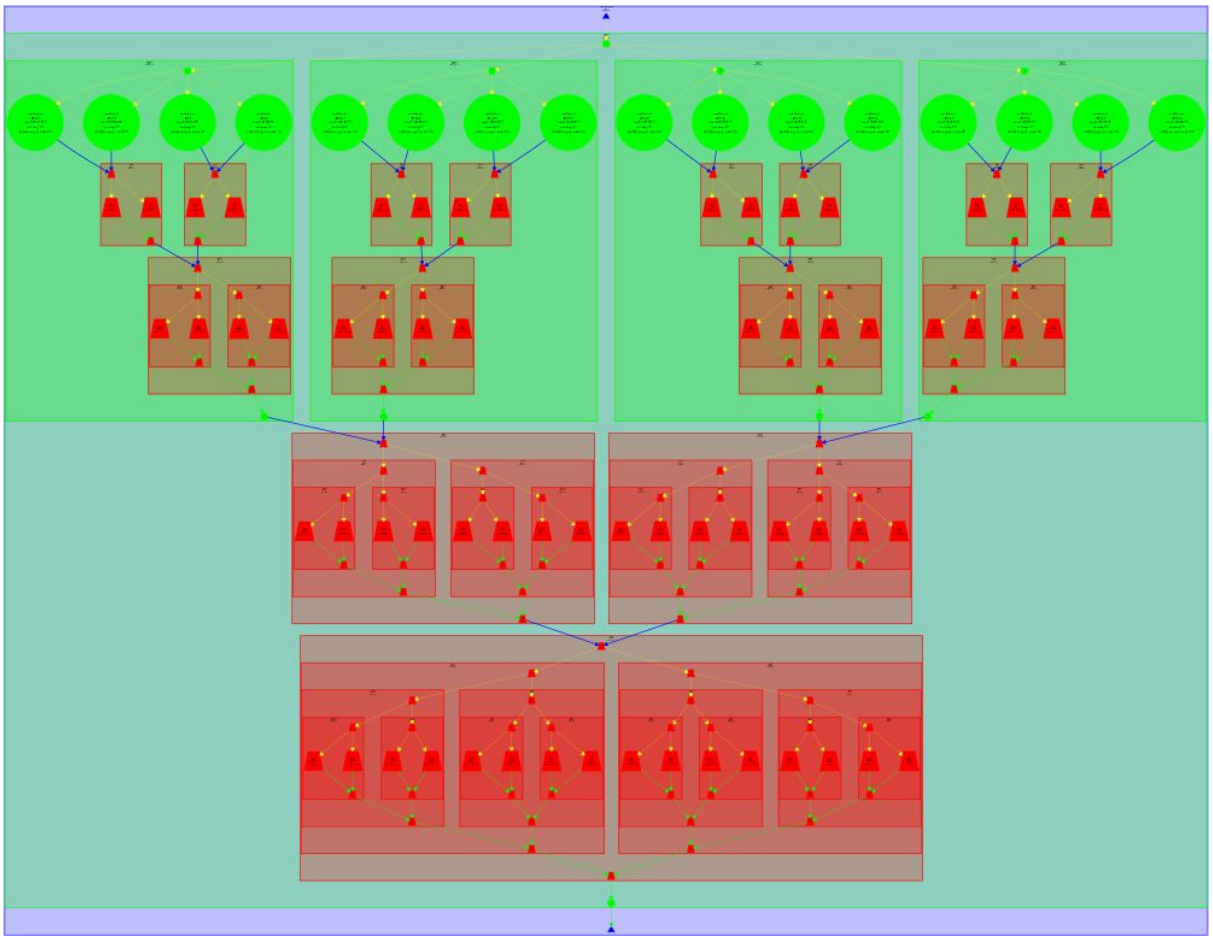
1. Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

Code

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        tareador_start_task("merge"); //code added to start a task for each merge
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                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);
        }
        tareador_end_task("merge"); //code added to end a task for each merge
}

void multisort(long n, T data[n], T tmp[n]) {
        tareador_start_task("multisort"); //code added to start a task for each multisort
        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]);
                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);
   } else {
         // Base case
         basicsort(n, data);
   }
   tareador_end_task("multisort"); //code added to end a task for each multisort
}
```

Multisort-tareador dependences task graph



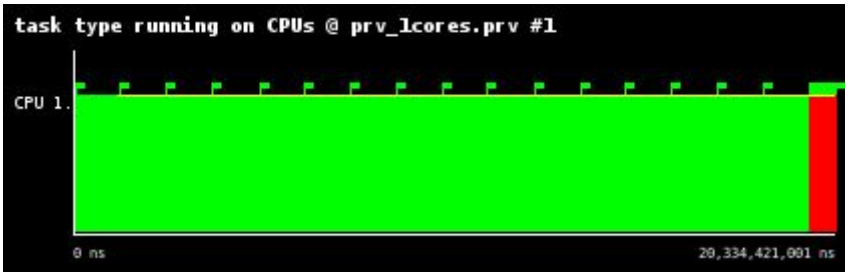
In the multisort function, there is a recursive decomposition which is done by dividing the data of the vector into four parts and each of these parts into other four parts until the leafs (base case) and then the basic function is called. During the recursive decomposition, the sorted parts are merged two by two until all the parts are merged into a single one.

In order to see the parallelizable parts of the code, we have created a task for each region of the code that involves the recursive decomposition or the base case. We can see that the dependences obtained in the previous graph are caused by the data of the vector, because each part has to be sorted before merge it with another part.

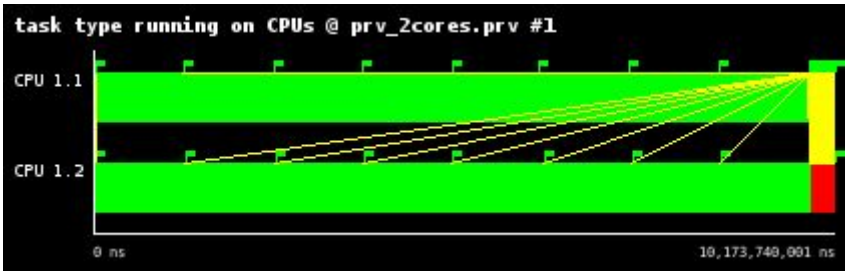
2. Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

Nthreads	Time Execution (ns)	Speed-up
1	20.334.421.001	-
2	10.173.740.001	1’9987
4	5.087.595.001	3’9969
8	2.547.419.001	7’9823
16	1.289.941.001	15’7638
32	1.289.920.001	15’7640
64	1.289.920.001	15’7640

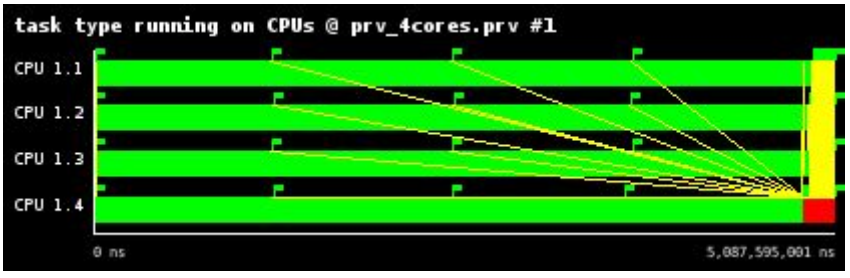
Paraver trace 1core



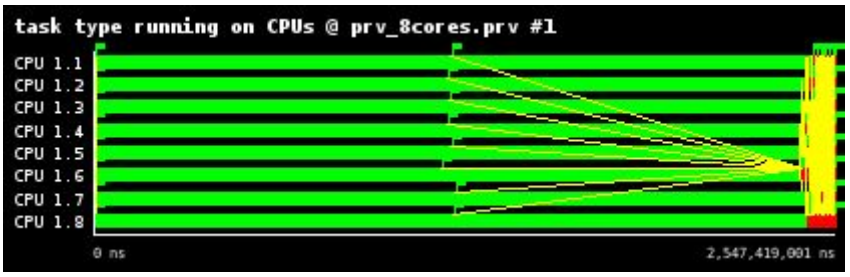
Paraver trace 2cores



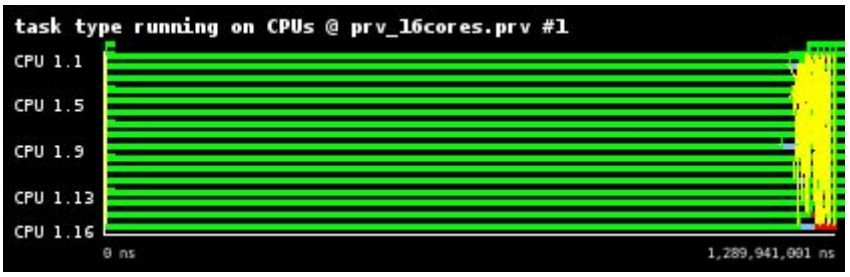
Paraver trace 4cores



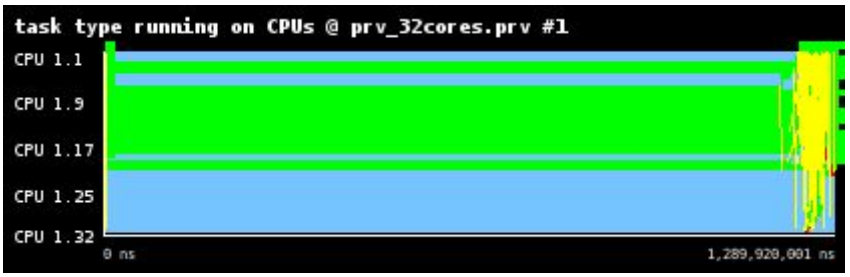
Paraver trace 8cores



Paraver trace 16cores



Paraver trace 32cores



Paraver trace 64cores

We can see that if we use more processors we obtain less execution time and better speed-up unless we use a number of processors bigger than 16. In that case, the maximum parallelization is reached and as consequence the execution time and the speed up are the same when we use more than 16 processors.

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.

For both versions we have to put **#pragma omp parallel** and **#pragma omp single** in the main before calling the multisort function in order to create the parallel region, the threads and a task for each thread. Only one thread will execute the multisort function.

Leaf version

```
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
                //we create a task for each basicmerge
        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
                //before merge all the parts, we have to wait until all of them are sorted
        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
                //both parts have to be sorted before calling the next merge so we need
                 to wait until we get both parts
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
   } else {
         // Base case
        #pragma omp task
                 //we create a task for each basicsort
         basicsort(n, data);
   }
}
```

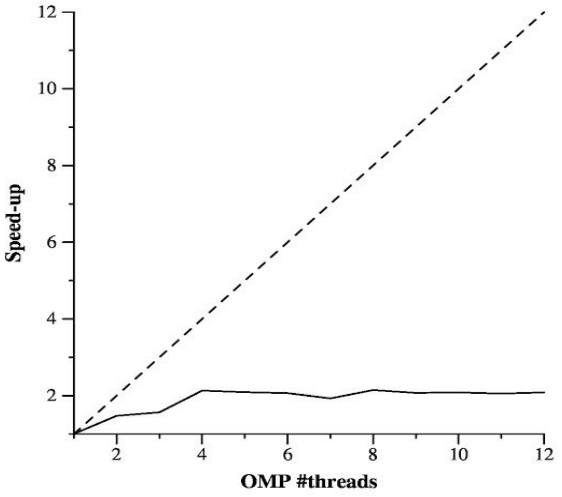
Tree version

```
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
                //we create a task for each merge
        #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
                //we use taskwait to ensure that the previous calls have been finished
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
                //we create a task for each multisort
        #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
                //we have to wait all the multisort calls have been finished to call the
                merge function
                //we create a task for each merge
        #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
                //we have to wait until both merge calls have been finished to call the
                merge function
                //we create a task for the merge
        #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        #pragma omp taskwait
                //we use taskwait to ensure that the previous calls have been finished
   } else {
         // Base case
         basicsort(n, data);
   }
}
```

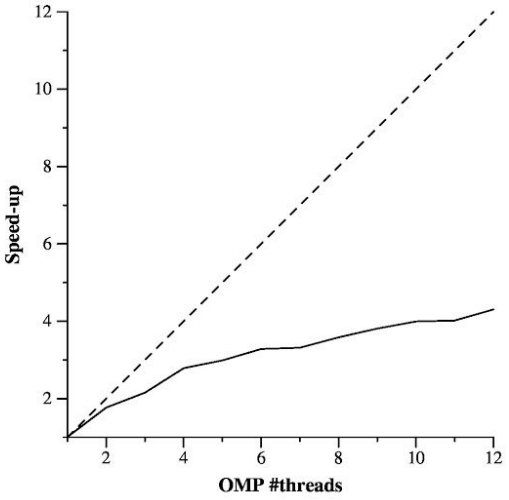
2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

Speed up plot Leaf version

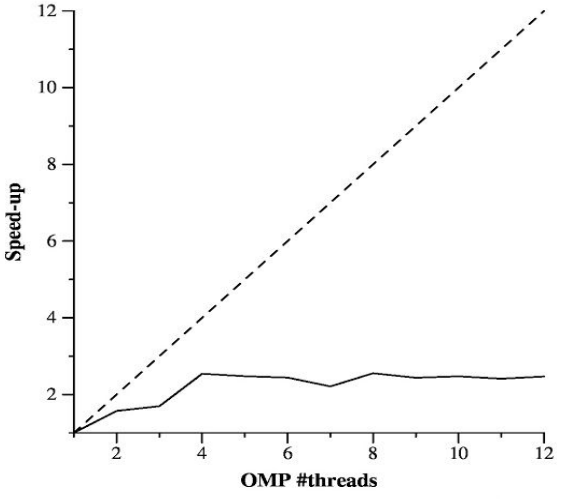


Speed-up wrt sequential time (complete application)

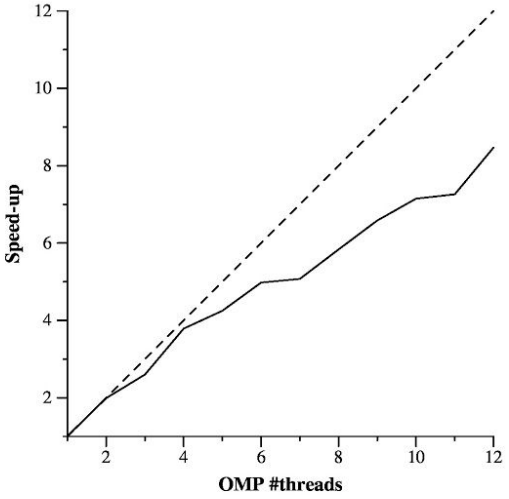
Speed up plot Tree version



Speed-up wrt sequential time (complete application)

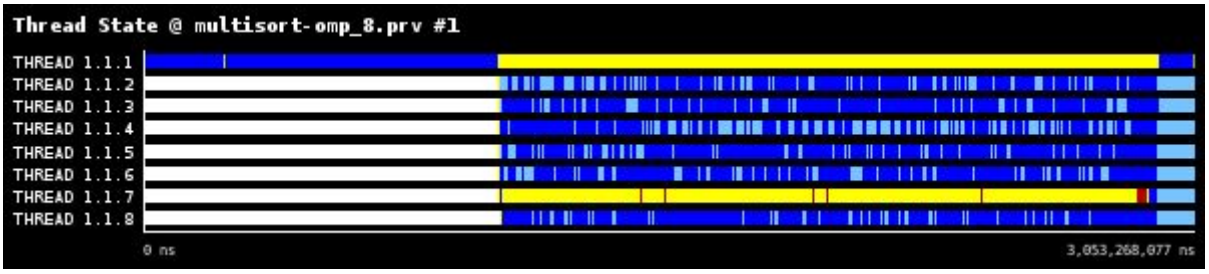


Speed-up wrt sequential time (multisort funtion only)

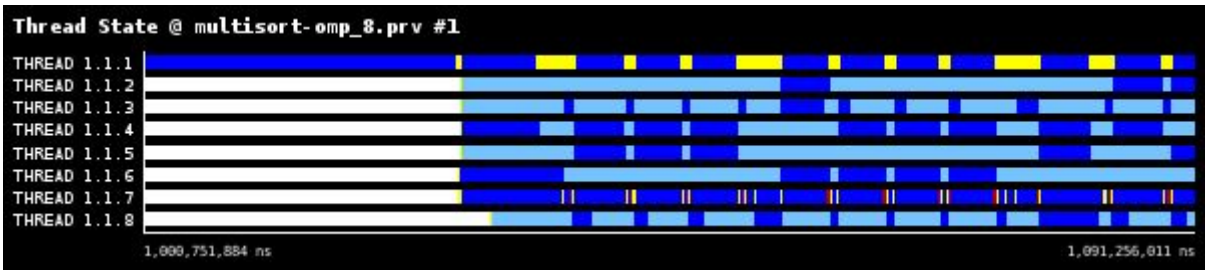


Speed-up wrt sequential time (multisort funtion only)

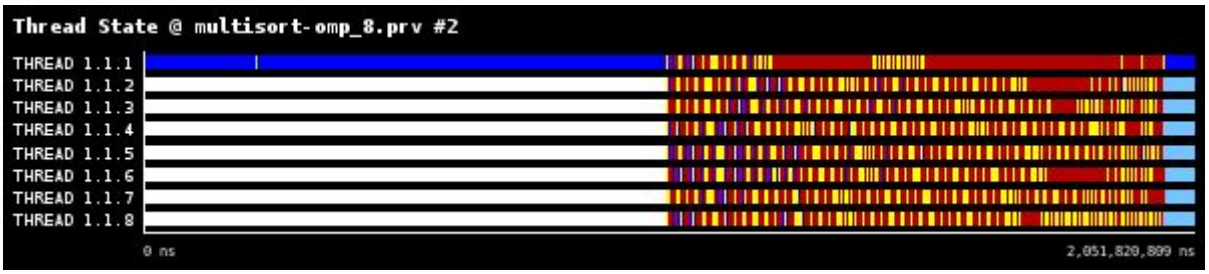
Paraver trace for Leaf version



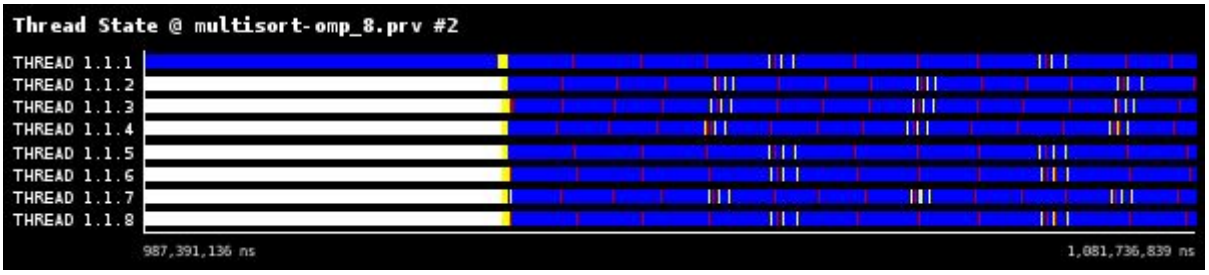
Paraver trace for Leaf version (zoom)



Paraver trace for Tree version



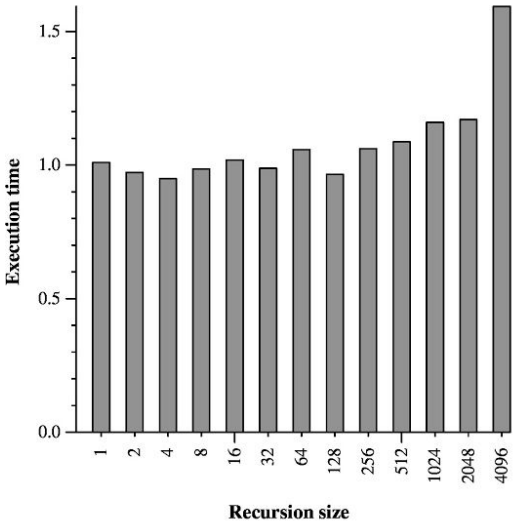
Paraver trace for Tree version (zoom)



In the Leaf version, we create a task in the leaf, and as a consequence of the recursive calls, the tasks wait until their child tasks finish, so the further away we are from the leafs, the time spend waiting increases and this causes that the speed up is worse than in the Tree version, where a task is created for each recursive call and each task creates other tasks until the leaf is reached. As consequence, the speed up obtained in the Tree version is better because the threads create tasks and only have to wait the tasks that each one has generated.

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?

Execution time plot of Tree version



Average elapsed execution time (multisort only)

In the Tree version we create a task for each recursive call so the more recursive calls we have, the more tasks are created and each task has to wait until the tasks created by itself finish the execution. Therefore, the more recursive calls we have, the more execution time we obtain. The difference of the execution time is greater when the recursion size is big. On the plot, we can see that the optimal value of recursion size is 4.

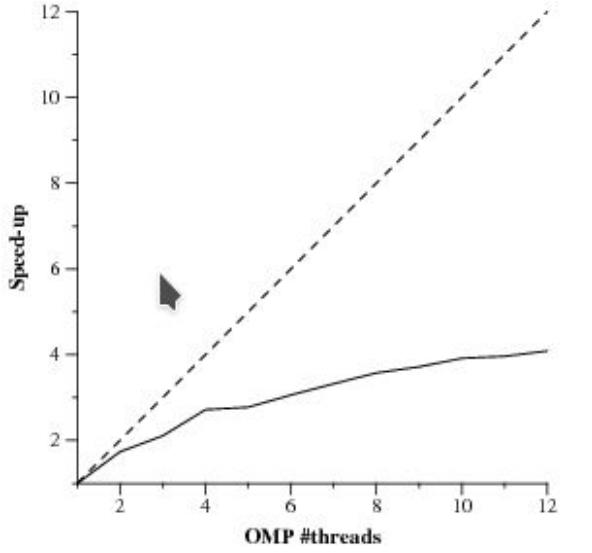
Parallelization and performance analysis with dependent tasks

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

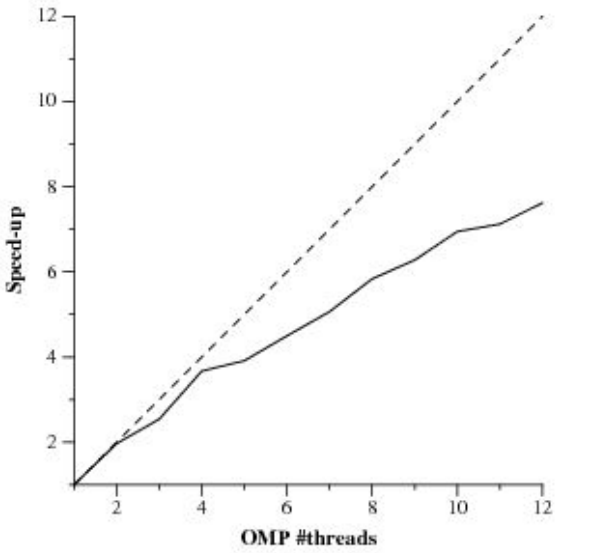
With the line ***#pragma omp task depend (in: …) depend (out: ...)***, we set a waiter-like with the variables in *"in:"*. The following code to that line, the new task, will be waiting to other tasks who are using the variables in *":in"*, once they have finished with that variable, they notify to the waiting tasks which variables on its *":out"* headers are now free.

The advantage using this line as a task creator, is that when it's done, it notifies to those task that are waiting for the variables on the *"out:"* header so they can start to execute.

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.

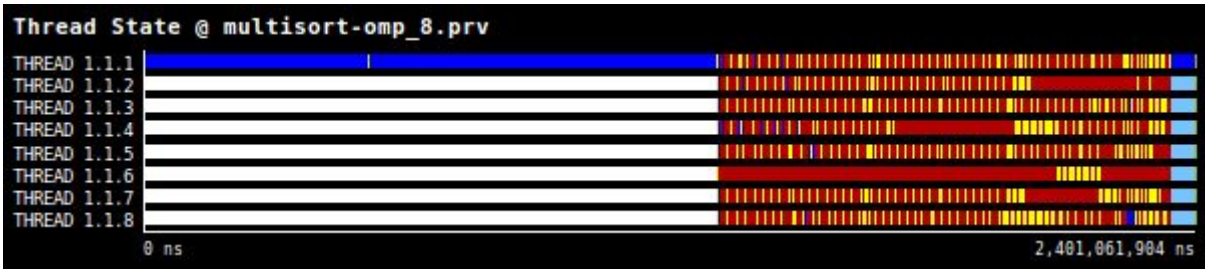


Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

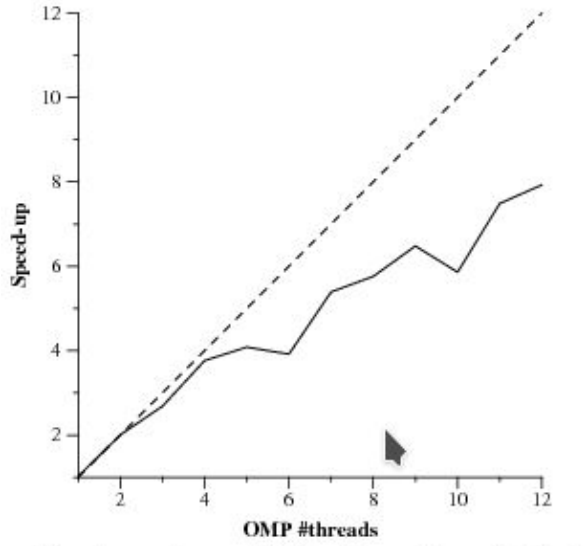
Paraver trace for Tree version with task dependencies



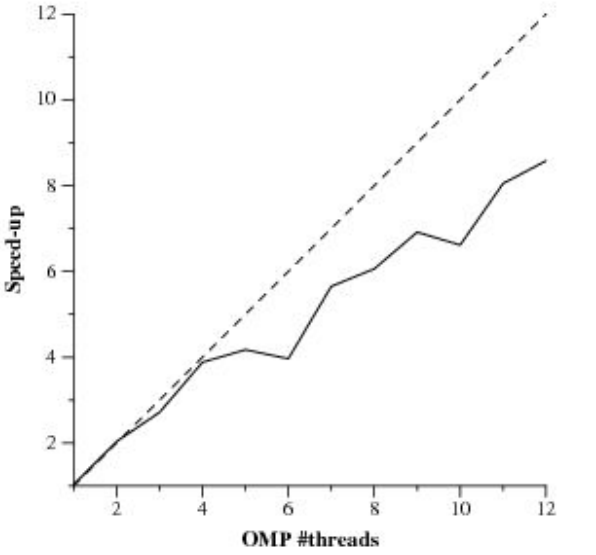
As shown in previous graphics and the ones in this exercise, we obtain similar values for the speed-up, but we think better results can be obtained thanks to the explicit definition of task dependencies, because the tasks just wait for the tasks that generate their values in *“in:”*, while in Tree version without explicit definition of dependencies, the *taskwait* just makes the task wait until its child tasks end.

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.

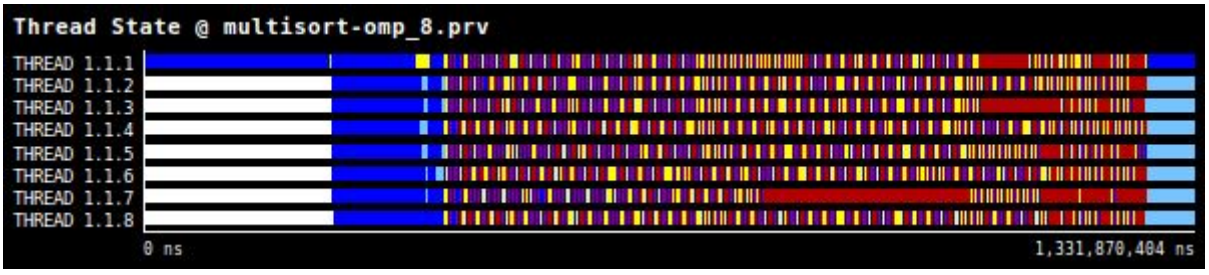


Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

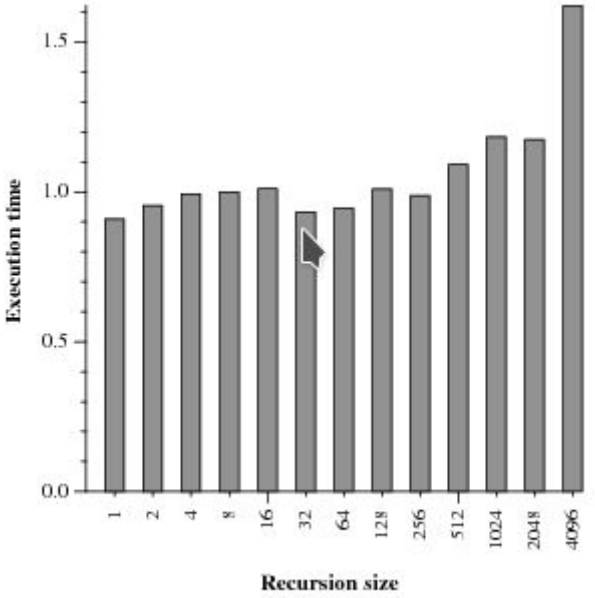
Paraver trace for Tree version parallel initialize



We can see that the speed up of the Tree version by parallelizing the two functions that initialize the data and tmp vectors1 is greater that the original Tree version. The reason is that, as we can see in the paraver trace, the parallelization of the program starts earlier so the program is executed faster and the speed up as consequence increases.

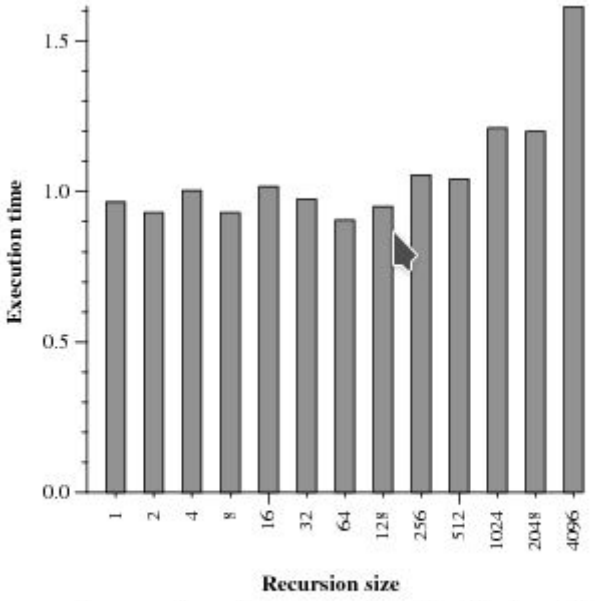
Optional 2: Explore the best possible values for the sort size and merge size arguments used in the execution of the program. For that you can use the submit-depth-omp.sh script, modified to first explore the influence of one of the two arguments, select the best value for it, and then explore the other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

Mergesize 256



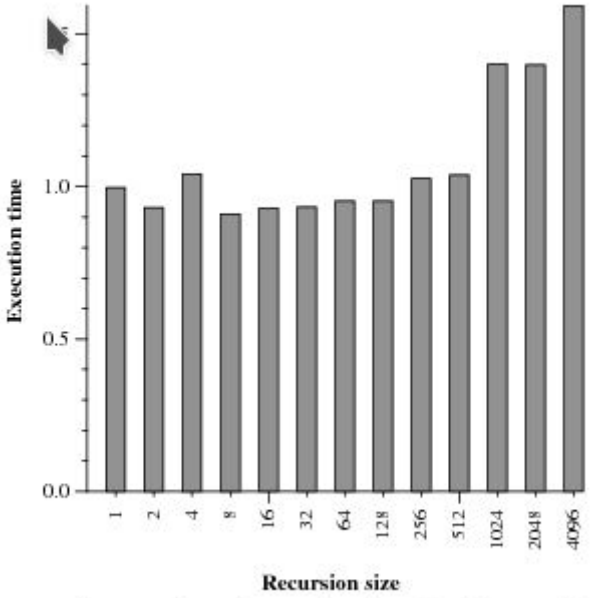
Average elapsed execution time (multisort only)

Mergesize 512



Average elapsed execution time (multisort only)

Mergesize 1024



Average elapsed execution time (multisort only)

Conclusions

Now, with all the job done, we can assure one of the most important things with this kind of tree-algorithm, is precisely, the Tree-Strategy. Just looking at the numbers provided before, we can see that using a Tree-Strategy to parallelize the algorithm we obtain much desirable times.

Several things to take into account for this kind of problem related with a similar code to the multisort one are:

- Using Tree-Strategy
- Dependencies
- Using *omp parallel for* in the initial loops of the app
- Better sort size is 32
- Better merge size is 256