PAR — Deliverable for Laboratory Session 4

Víctor Jiménez Arador, Miguel Moreno Gómez

Group 12_{02} — Fall semester 2017/2018

1 Introduction

The purpose of this laboratory is to evaluate and implement a parallel strategy for a sorting algorithm, in this case multisort: multisort is a divide-and-conquer algorithm that operates like a mergesort, except that instead of splitting a list up to a certain point (generally just one element), the merging cuts once reached a minimum subvector size. Then, it sorts these sublists using quicksort; finally it will merge these (already sorted) sublists into a greater sublist, until the final (and sorted) list is generated.

Unlike the previous laboratory session (Mandelbrot set), which was embarrassingly parallel, there are dependencies among parts of the programs, and we have to make sure these dependencies are correctly executed, without interferences on any data structures.

2 Parallelization Strategies

There are two parallelization strategies for this algorithm, we'll describe them briefly:

- **Leaf strategy:** Once recursion stops, tasks are created per each function call. In this case, for each time a subvector has to make a call to basicmerge or basicsort (which are, in fact, the base cases of the recursion).
- **Tree strategy:** new tasks are made for each recursion call of the program; for us, it's each call to multisort() and merge() functions.

3 Performance evaluation

There are three ways of evaluating the performance of our strategies:

- **Strong scalability:** submitting a program to be run between 1 and 12 cores to evaluate the speed-up compared to the sequential time and the average time elapsed per core execution.
- **Recursion size:** for a few problems we can see interesting the size of recursion (or depth in our case) of all recursive calls in our program. These run from 2^0 up to 2^{12}.
- **Traces:** we can see the traces of the executions of our programs, using 8 threads, to get an idea of what each thread is doing. In this deliverable we just used the default *Paraver* setup, which draws a timeline for each thread.

In layman's terms, we are interested in seeing if cores as much dark blue as possible (executing code), a minimum of yellow (scheduling and fork/join) and as little red (synchronization) and white (not started yet) as possible; all of this has to happen to as many cores as possible at the same time.

4 Analysis with Tareador

4.1 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.

All *Tareador* calls surround each and every recursive call to the functions multisort, merge and basicmerge.

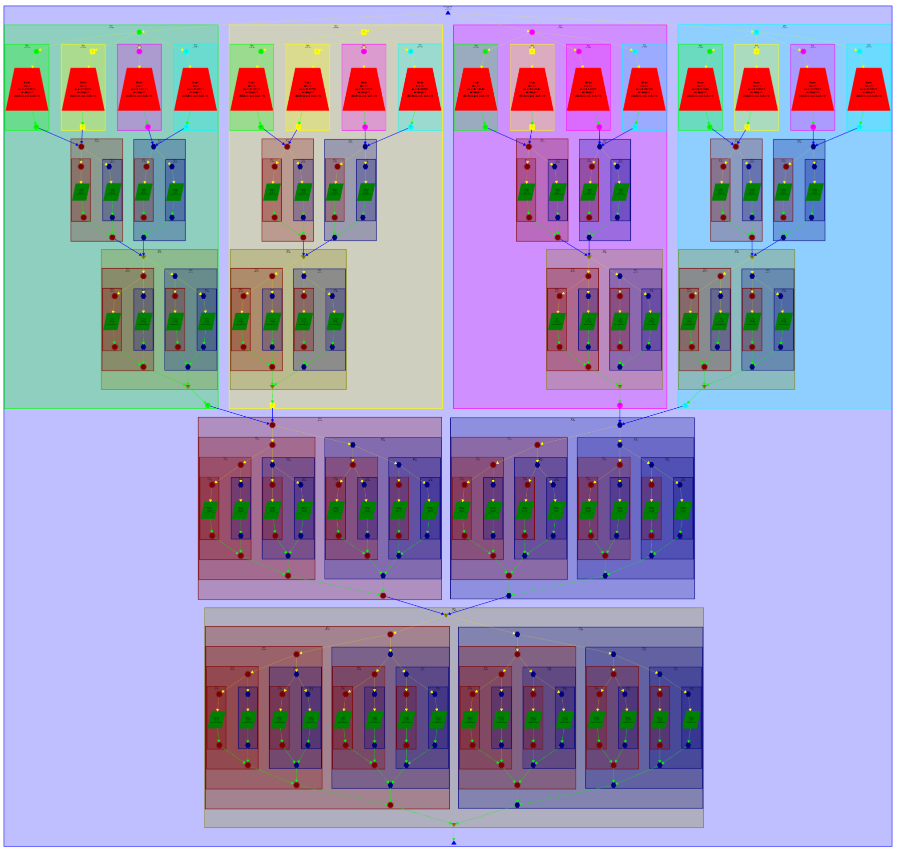


Figure 1: Task Dependency Graph (TDG) generated with Tareador for the multisort algorithm.

The are at most 16 tasks in parallel in the graph, that is, the maximum tasks displayed in horizontal in the TDG, or (in other words) the parallelism of the program.

4.2 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.

Num. of cores	1	2	4	8	16	32	64
Speed-up	1	1,998	3,997	7,97	15,775	15,775	15,775
Execution time	20,334	10,174	5,087	2,550	1,289	1,289	1,289

(All times displayed in seconds. Speed-up compared to $\#_{cores} = 1$*)*

We consider results are close to the ideal case: when the number of processors is equal or lower than 16, the speed-up is quite close to the number of cores, but when the number of cores is greater than 16, the speed-up is stuck at 15.775, because that's the potential parallelism of the program.

5 Parallelization and performance analysis with tasks

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

(Note that there are two pragma directives on the 3 merges of multisort(), as the first two merge from four subvectors to two, and the second one merges these two into one.)

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
        #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);
        }
    }
}

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 taskwait

        // Merges 4 vectors -> returns 2 vectors
        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);

        // Merges 1 vectors -> returns 1 vector
        #pragma omp taskwait
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Leaf version:

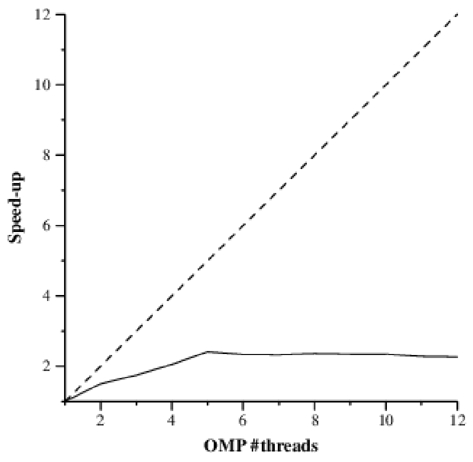
```
void merge(long n, T left[n], T right[n], T result[n*2],
           long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        #pragma omp task
        basicmerge(n, left, right, result, start, length);

    } 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]);

        // Merges 4 vectors -> returns 2 vectors
        #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);

        // Merges 2 vectors -> returns 1 vector
        #pragma omp taskwait
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
     // Base case
     #pragma omp task
     basicsort(n, data);
    }
}
```

Common fragment of code for both versions:

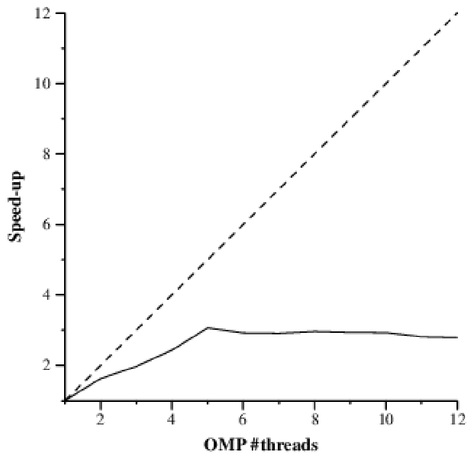
```
int main(int argc, char **argv) {
    ...
    #pragma omp parallel
    #pragma omp single
    multisort(N, data, tmp);
    ...
}
```

5.2 For 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.

For the leaf plot, the speed-up of the program seems to increase, but once more than 5 processors are used, it gets stuck with a speed-up close to 3 once we execute with more processors. The reason is that while most of the threads are constantly working or idle (this is a potential issue!), others are simply managing the scheduling and forks/joins, doing little to no computing for the algorithm. We suspect that, the more threads are used, these "new" threads are handling this scheduling and *not* into computing the multisort algorithm, which is not what we want!



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Figure 2: Leaf version's strong scalability plot.

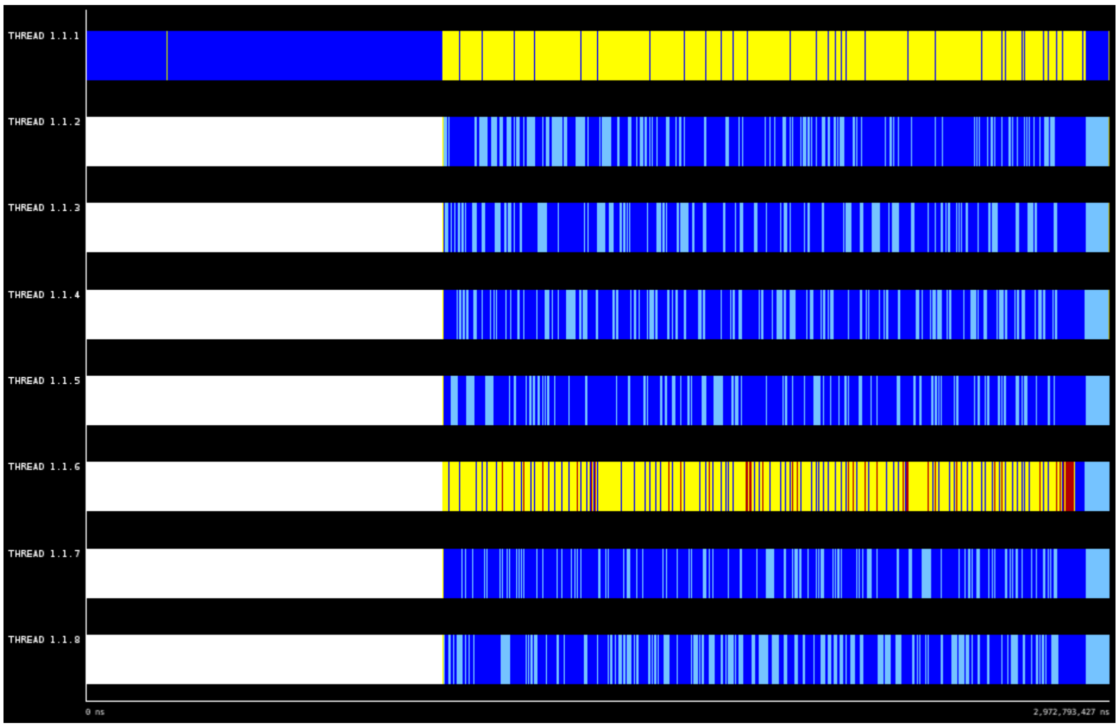
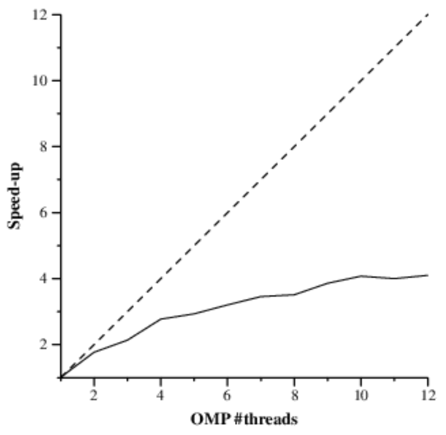
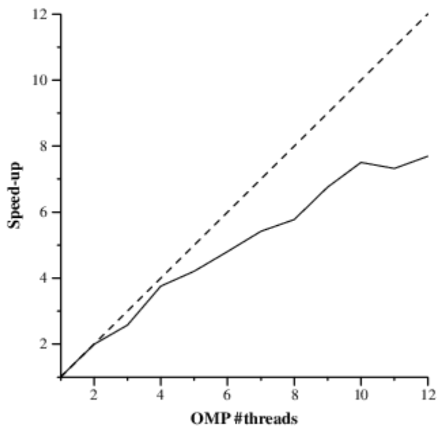


Figure 3: Paraver window for MultiSort, Leaf version.

For the tree plot, instead, speed-ups are more irregular yet they keep increasing as more processors are added. When the number of threads is 11, however, there seems to be a slight drop, perhaps because of task management among threads. Also, the *Tree* version of the program runs at 1.4 seconds, which is more than twice as fast as *Leaf* strategy.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Figure 4: Tree version's strong scalability plot.

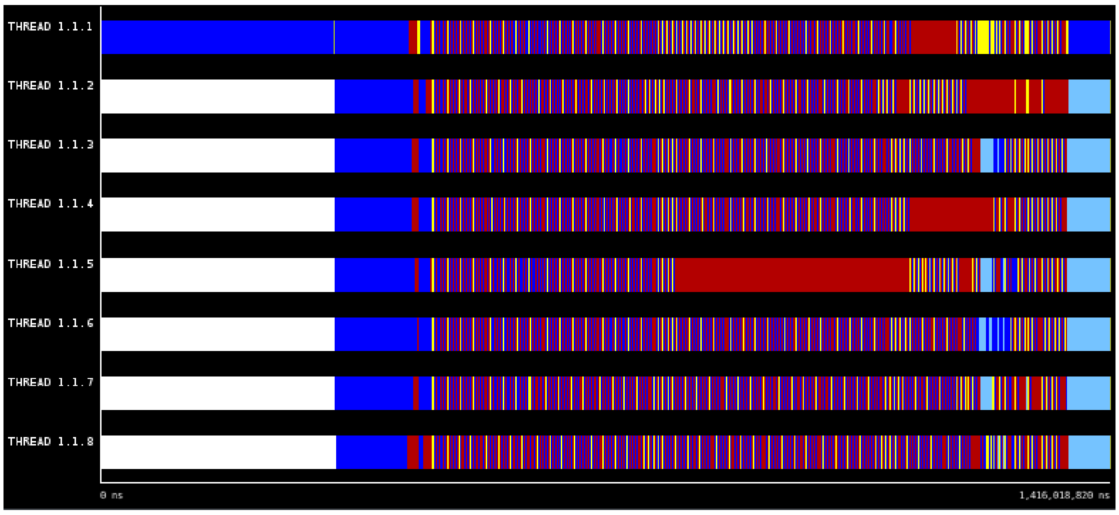
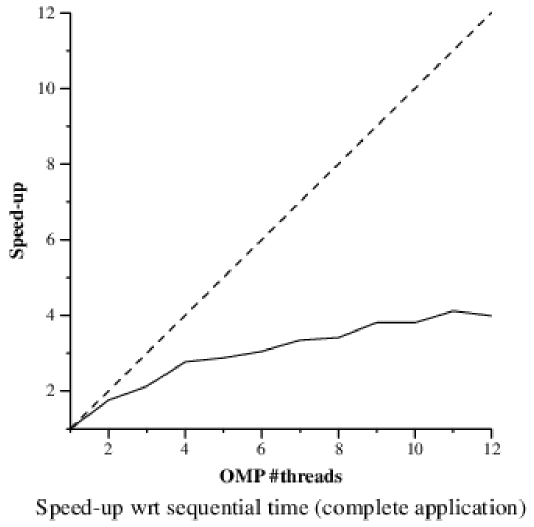


Figure 5: Paraver window for MultiSort, Tree version.

5.3 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?



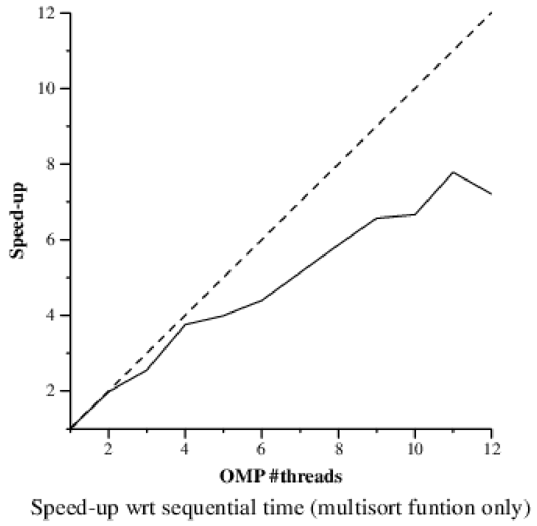


Figure 6: Tree version's strong scalability plot, but with $MAX\_DEPTH = 2$.

Compared to the original plot, it seems quite similar; the most noticeable difference is that the drop in speed-up happens with 12 threads, and not 11.

An theoretical optimal value is 2: when the vector has to be split, it's done by calling `multisort()` recursively four times (one call per task).

Thus, $4^2 = 16$. Remember that the maximum potential parallelism of the algorithm is 16; this means that going deeper won't improve performance because no more (generated) threads can run at the same time.

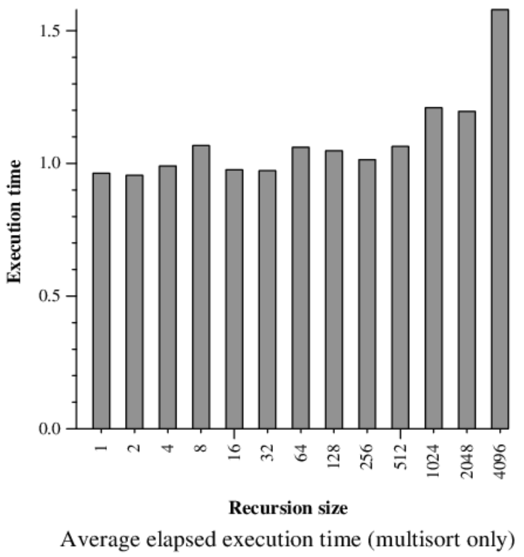


Figure 7: Tree version's recursion plot.

The recursion plot shows, instead that execution time is more or less constant for any depth... except depth 4096, where the execution time increases dramatically!

5.4 Optional 1: Complete the parallelization of the *Tree* version by parallelizing the two functions that initialize the `data` and `tmp` vectors. 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.

(Note: this was made using the original *Tree* file, not the one using the one with the recursion depth optimization)

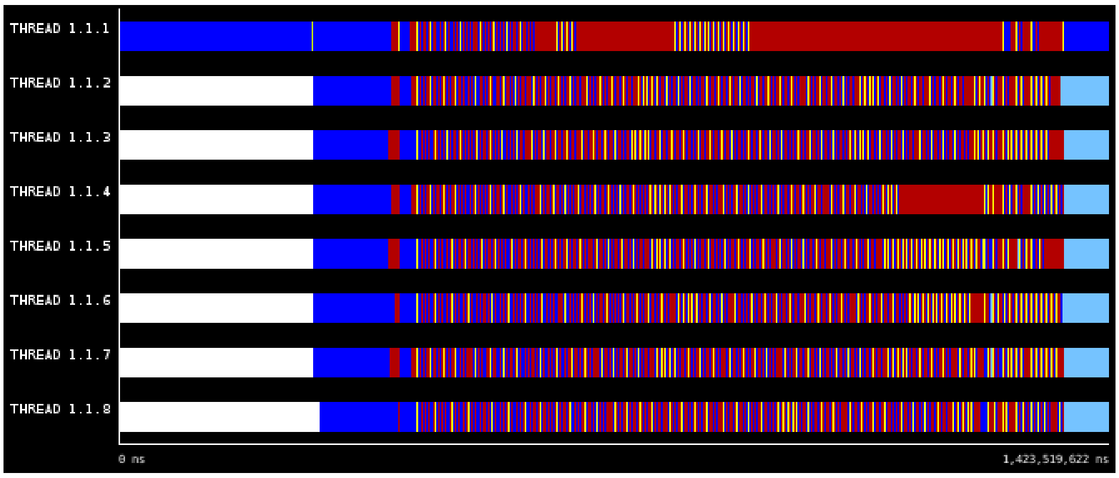
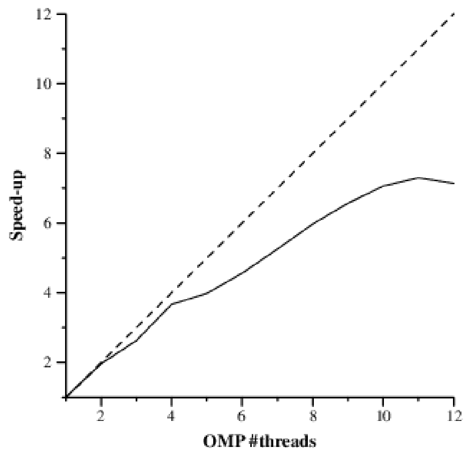


Figure 8: Paraver timeline of the optimized Tree code.

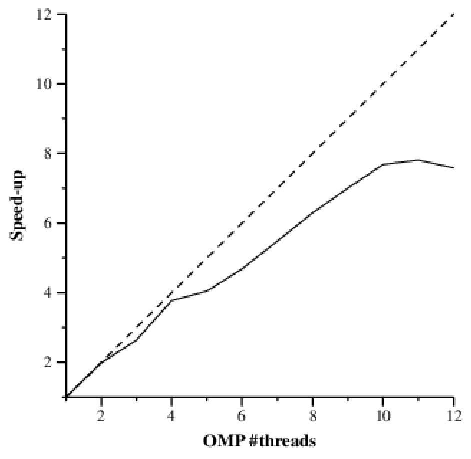
The code has an overall better speedup, because the initialization, if sequential, can be slower than the parallelized multisort! With this optimization, now that part's speed improvements are proportional to the number of processors used. Moreover, these speed-up plots show that the difference among processors is smoother than without this optimization.

The plot in Paraver, however, doesn't show this potential optimization as well, as it seems non-created time of the 7 other threads start to execute code is the same, but it's just a bit lower than in the main code.

In spite of that, it's good to have this optimization in the code, as it doesn't hurt the execution of the multisort algorithm.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Figure 9: Tree version's strong scalability plot, but even more parallel.

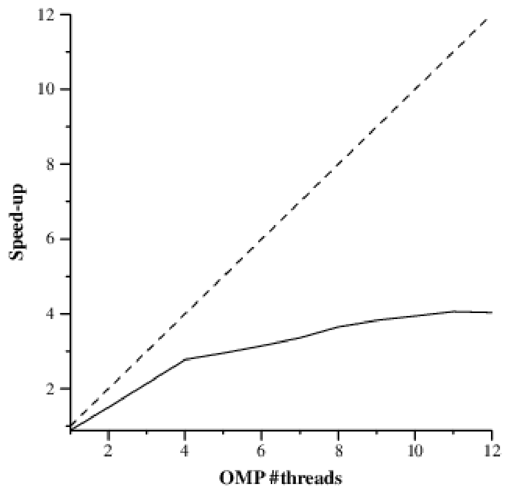
6 Parallelization and performance analysis with dependent tasks

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

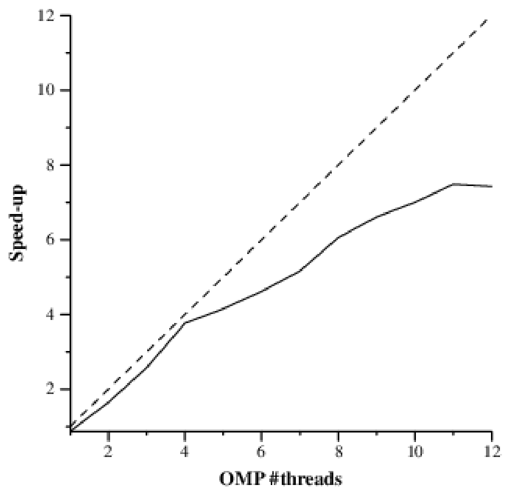
The only relevant part of the code with task dependencies is the multisort() function. The rest is the same compared to the original *Tree* decomposition:

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task depend(out: data[0])
        multisort(n/4L, &data[0], &tmp[0]);
        #pragma omp task depend(out: data[n/4L])
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        #pragma omp task depend(out: data[n/2L])
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        #pragma omp task depend(out: data[3L*n/4L])
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

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

        // Merges 1 vectors -> returns 1 vector
        #pragma omp task depend(in: tmp[0], tmp[n/2L]) depend(out: data[0])
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        #pragma omp taskwait
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

6.2 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)

Figure 10: Tree version's strong scalability plot, with dependencies.

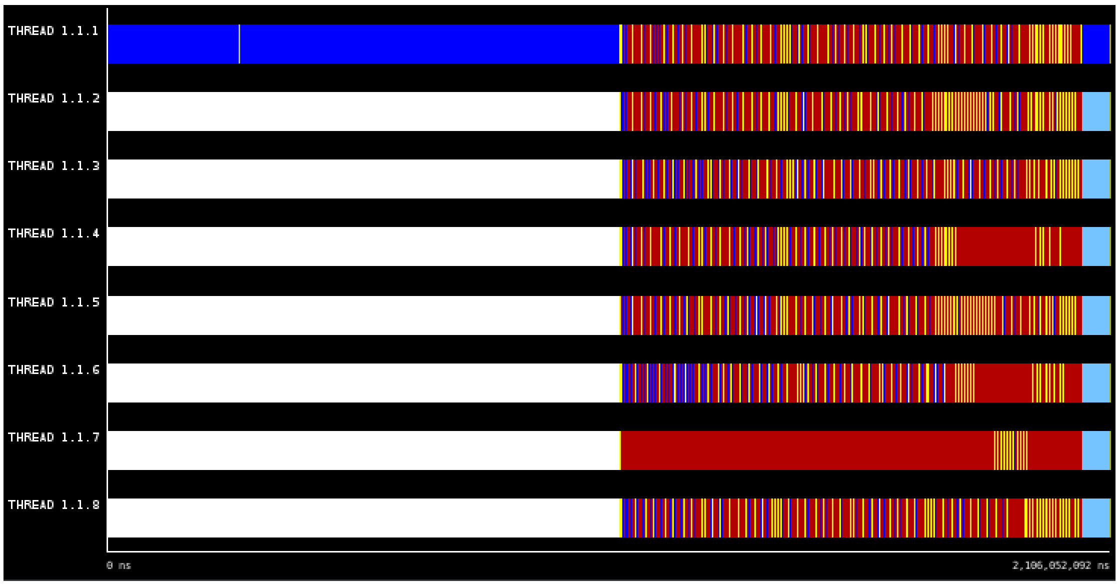


Figure 11: Paraver window for Tree version with dependencies.

There are not too many differences between these plots and the ones we discussed earlier (sans the first optional one), except for the fact that the speed-up curve is a bit smoother than the previous one. The *Paraver* tracem, however, shows another story: while it takes roughly half of the time to initialize the recursion steps, the other half of the time is done executing the code. This also causes the program takes 2.1 seconds to execute, that is, about 0.7 seconds slower than the other strategy, so the results aren't as satisfactory as we expected

6.3 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.

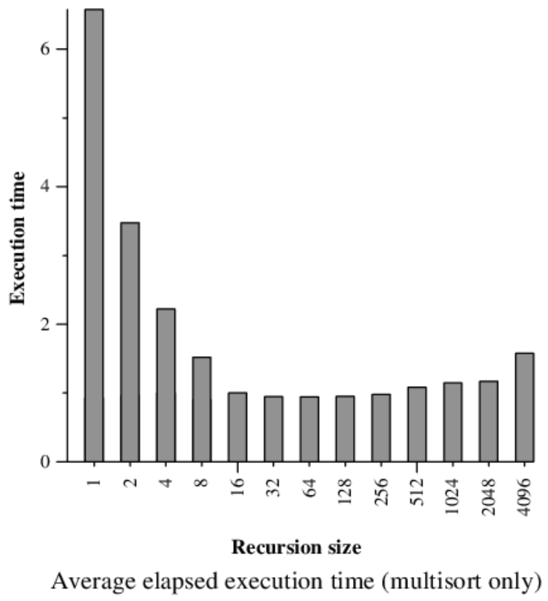
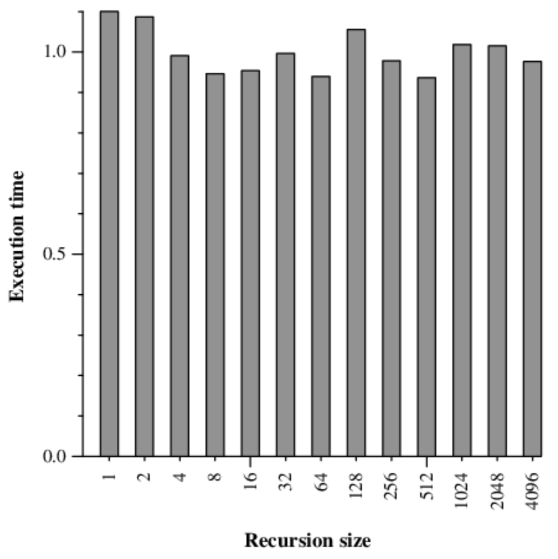


Figure 12: Recursion plot for the depth size.



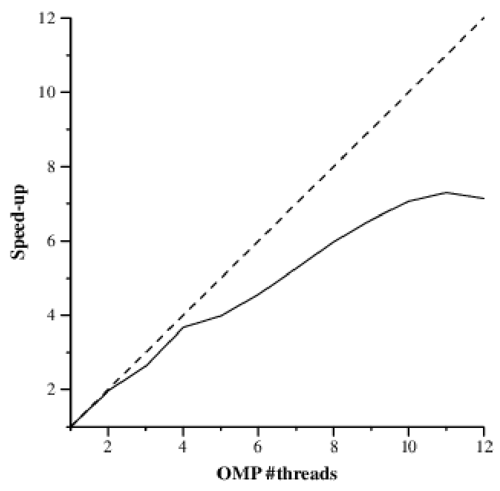
Average elapsed execution time (multisort only)

Figure 13: Recursion plot for the merge size.

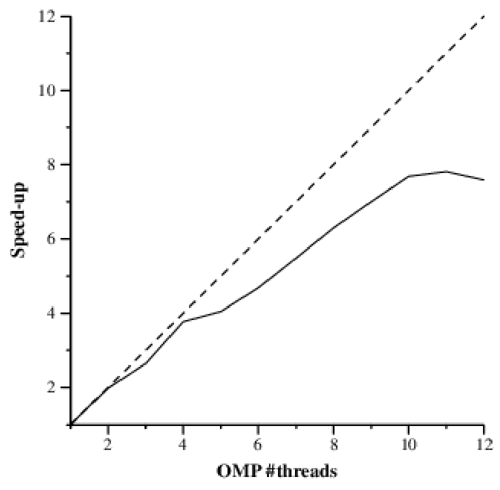
Some of the best values are (bolded are the values chosen for the plot):

- **For depth size:** 16, 32, **64**, 128
- **For merge size:** 8, 16, **64**, 256

The plots don't show that much of a difference compared to the ones we obtained in the first optional exercise. That said, the performance is greater than in subsection 6.2, perhaps due to the fact that depth size can be quite punishing if its value is too low (1, 2, 4).



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Figure 14: Tree plot for the merge size = 64 and depth size = 64.

7 Conclusions

Finally, we can draw several conclusions from the development of this laboratory session:

- First of all, the *Leaf* has proven to be an ineffective parallelization strategy, because it stagnates with just a few cores (exactly, 5). Compared to the potential parallelism of Tareador's Task Dependency Graph, it's disappointing to find out that our code's parallelism is *3 times lower* than the theoretical one!

- Fortunately, the *Tree* strategy performs much better, but the original version leaves some room for improvement:
  - The original version of *Tree* is the fastest, and there's not that much difference between that and the one using recursion depth limits, so any choice could be considered valid; in spite of that, the first version runs only a bit faster than the depth one, so it has a slight advantage.
  - The version with dependencies, while it has a more predictable speed-up plot, is slower than the original version. Considering that one of the main goals (if not THE main goal) of parallelism is to execute code as fast as possible, we'd have to discard it, compared to the original one. Nonetheless, it's a good way to learn that code can be parallelized by forcing the input and output of data structures, not just by using task/taskgroup/taskloop directives.
- We also learned that we have to be cautious with the type of OpenMP directives used, because they can destabilize the code (or straight-up break it) during execution; for example, we faced incorrectly sorted vectors and segmentation faults as we completed this deliverable.
- As a conclusion, the best options for this program so it can run as fast as possible are:
  - Using the improved initialization scheme from the first optional part.
  - Use the standard Tree version for the parallelization of the multisort algorithm.