Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

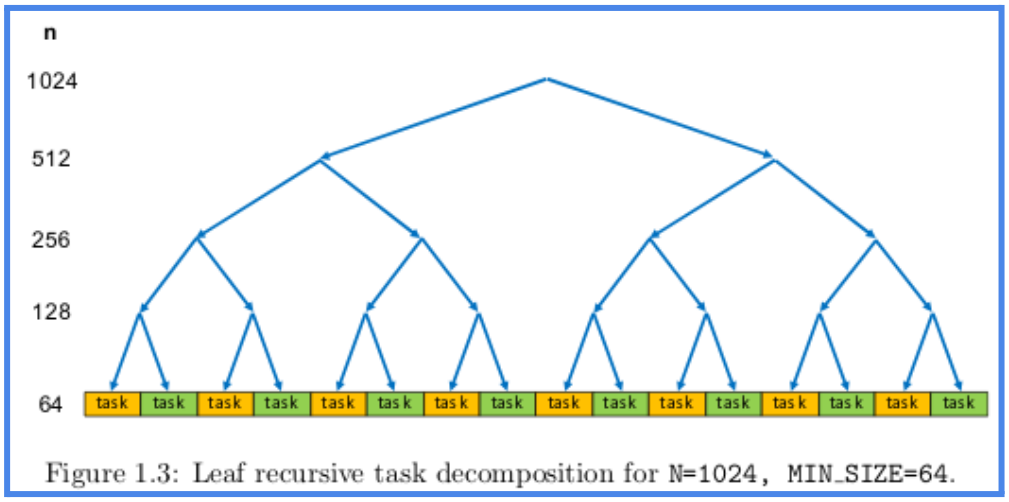


Figure 1.3: Leaf recursive task decomposition for `N=1024, MIN_SIZE=64`.

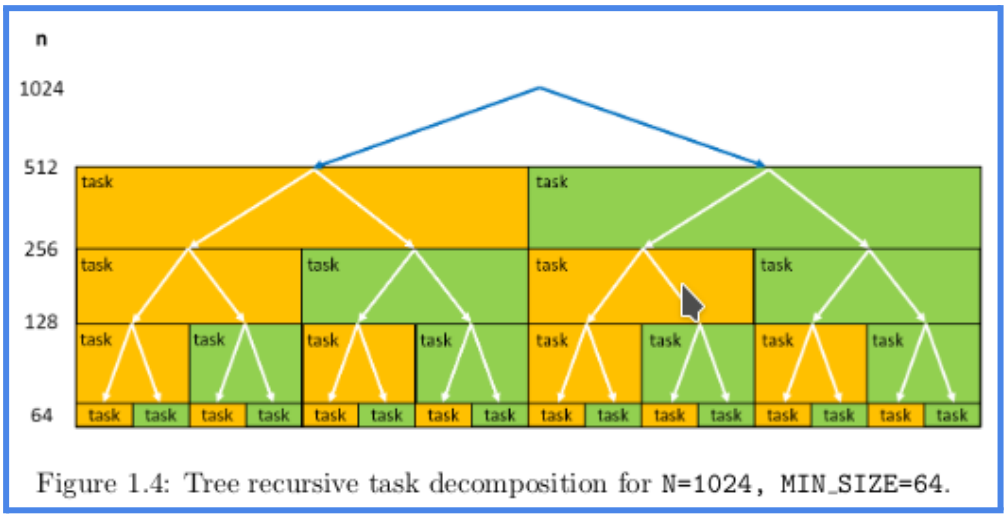


Figure 1.4: Tree recursive task decomposition for `N=1024, MIN_SIZE=64`.

Adrià Redondo
Miguel Gutiérrez Jariod
Group 23
24/03/2021 - QP 2021-2022

Index:

4.1 Analysis with tareador

Leaf strategy:

```
par2311@boada-1:~/lab4$ ./multisort-seq -n 32768 -s 1024 -m 1024
********************************************************************************
********************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN
_MERGE_SIZE=1024
********************************************************************************
********************
Initialization time in seconds: 0.886967
Multisort execution time: 6.396762
Check sorted data execution time: 0.017775
Multisort program finished
********************************************************************************
********************
par2311@boada-1:~/lab4$
```

Image 1.1: ./multisort-seq -n 32768 -s 1024 -m 1024

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, l
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        tareador_start_task("basicmerge1");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("basicmerge1");

    } 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]);

        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
        tareador_start_task("basicsort1");
        basicsort(n, data);
        tareador_end_task("basicsort1");
    }
```

Image 1.2: code of leaf strategy

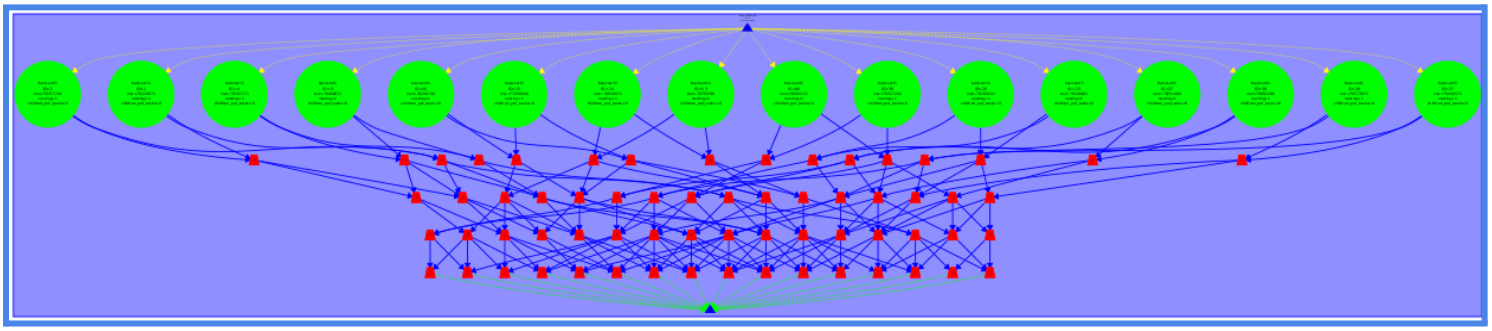


Image 1.3: graph task dependence of leaf strategy

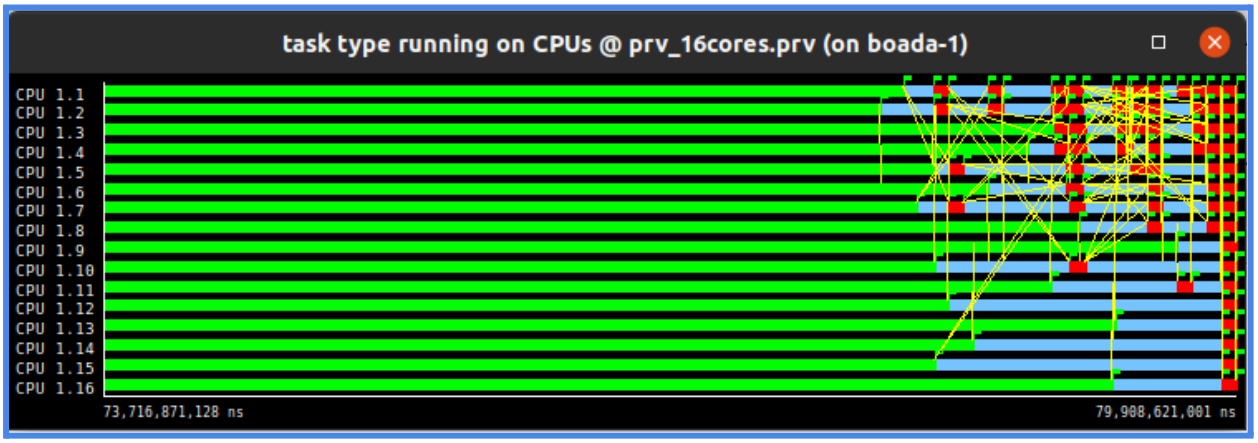


Image 1.4: paraver simulation of leaf strategy 16 processors

In the first part of the practice we have developed different strategies with the tareador. The image 1.2 shows the code of the leaf strategy, so the tareador tasks are situated on the base case.

Once we have the code, we execute the simulation and the graph that creates this strategy is shown in the image 1.3. On it we can appreciate that the tasks are only created at the deepest level (the leafs ones). Later we ran the paraver with 16 processors, as shows the image 1.4. On it we can see that the red points are the ones where we can look at the strategy accomplishment.

Tree strategy:

```
void merge(long n, T left[n], T right[n], T result[n*2], long s
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        tareador_start_task("merge1");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("merge1");
        tareador_start_task("merge2");
        merge(n, left, right, result, start + length/2, length/
        tareador_end_task("merge2");
    }
}

void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("multisort1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("multisort1");
        tareador_start_task("multisort2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("multisort2");
        tareador_start_task("multisort3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("multisort3");
        tareador_start_task("multisort4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("multisort4");

        tareador_start_task("merge3");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("merge3");
        tareador_start_task("merge4");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0,
        tareador_end_task("merge4");

        tareador_start_task("merge5");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge5");

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

Image 1.5: code of tree strategy

In this part we have developed the tree strategy. Now, compared with the previous code, the tareador tasks are situated on the recursive part, so not only the leafs create tasks, these are in each part of the tree.

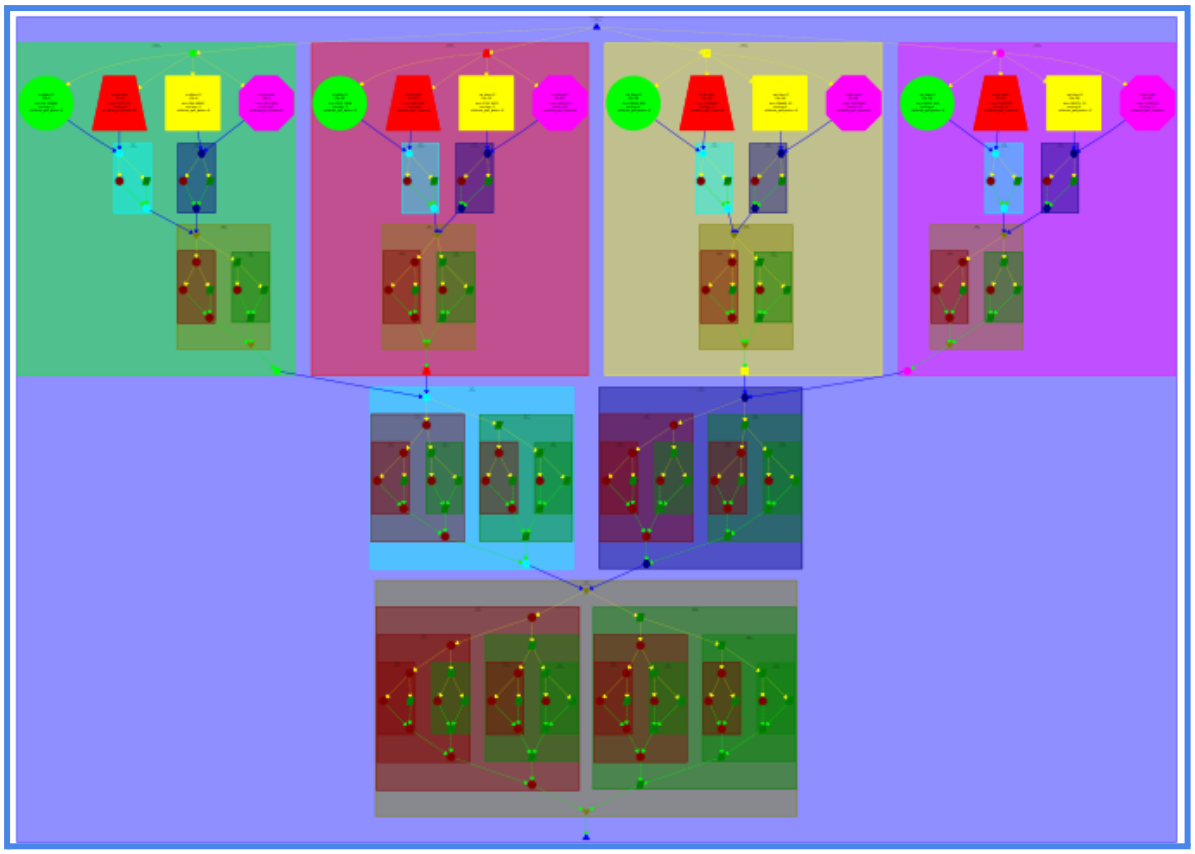


Image 1.6: graph task dependence tree strategy

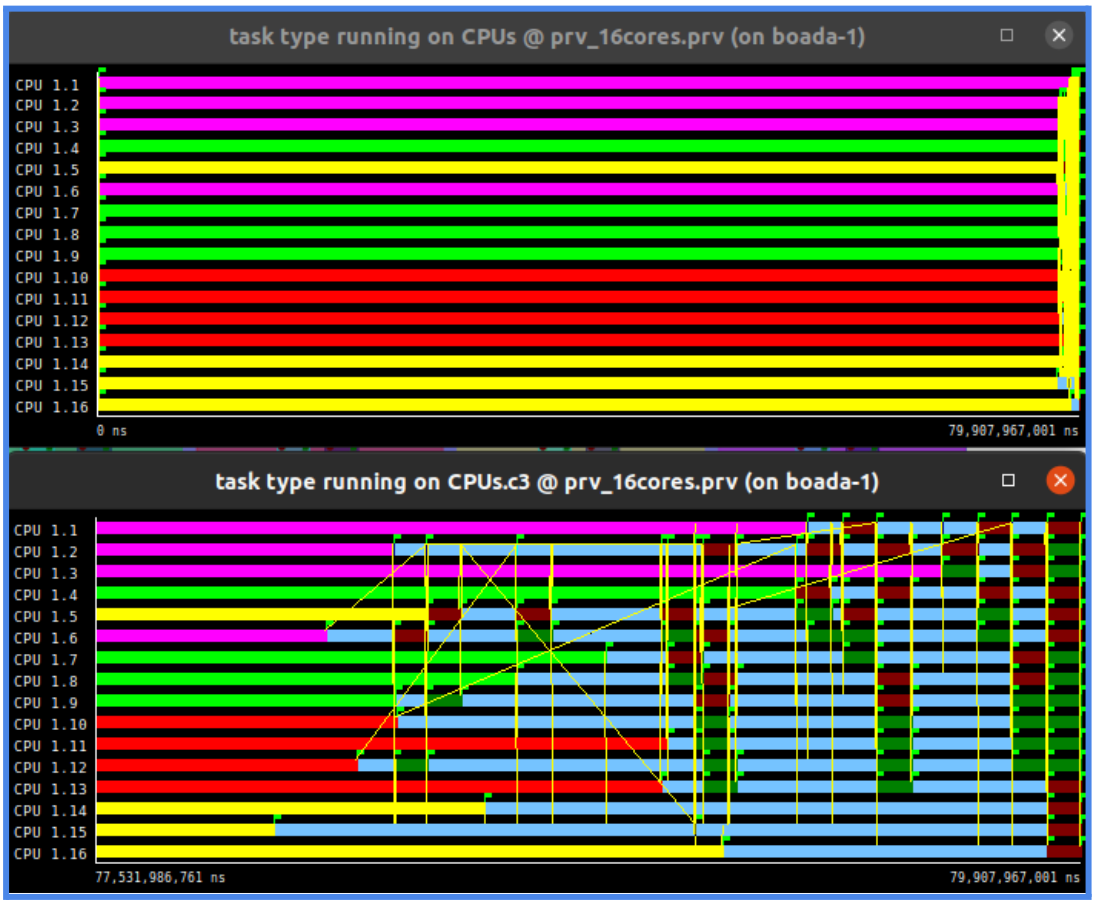


Image 1.7: paraver simulation of tree strategy 16 processors

Once we execute the code, we obtain the graph result reflected on the image 1.6. We can appreciate that now on each level, for each part, a task has been developed, that's the reason why we can see the shape of all the tree. Later we execute the paraver with 16 processors and we obtain the simulation result of the image 1.7

4.2 Shared memory parallelization with OpenMp

Leaf strategy:

```
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);
        }
```

Image 3.0.1 code snippet of leaf strategy in merge function

```
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);
        }
    }
}
```

Image 3.0.2 code snippet of leaf strategy in multisort function

```
************************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************************************
Initialization time in seconds: 0.859158
Multisort execution time: 6.352677
Check sorted data execution time: 0.016631
Multisort program finished
************************************************************************************************
```

Image 3.0.3 execució de ./multisort-omp amb leaf strategy

Image 3.0.4 paraver for leaf strategy with 2 threads

In this strategy we generate tasks only when the base case is invoked as shown in images 3.0.1 and 3.0.2.

Due to the recursive nature of the multisort function, the number of tasks corresponds to the number of leafs in the recursion tree generated by the execution. That is n / (MIN_SORT_SIZE * 4L) tasks.

From image 3.0.3 and 3.0.4 we can see that, although we are executing the program with 2 threads, the execution is heavily sequential.

This phenomenon occurs because of the nature of the execution. In theory, the leafs of the recursive program do not have a specific execution order and there should be no problem in parallelizing them. Nevertheless, the execution follows a depth-first search type of route and the execution of the next leaf never begins before the previous one's execution has not finished.

Tree strategy:

```
************************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************************************
Initialization time in seconds: 0.854392
Multisort execution time: 4.905185
Check sorted data execution time: 0.018834
Multisort program finished
************************************************************************************************
```

Image 3.0.5 execution ./multisort-omp with cut-off

```
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);
        #pragma omp taskwait
    }
}

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

        // Recursive decomposition
        #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

        #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

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

```
#pragma omp parallel
{
    #pragma omp single
    multisort(N, data, tmp);
}
```

Image 3.0.6 and 3.0.7, code of the tree strategy and parallelization in the main part.

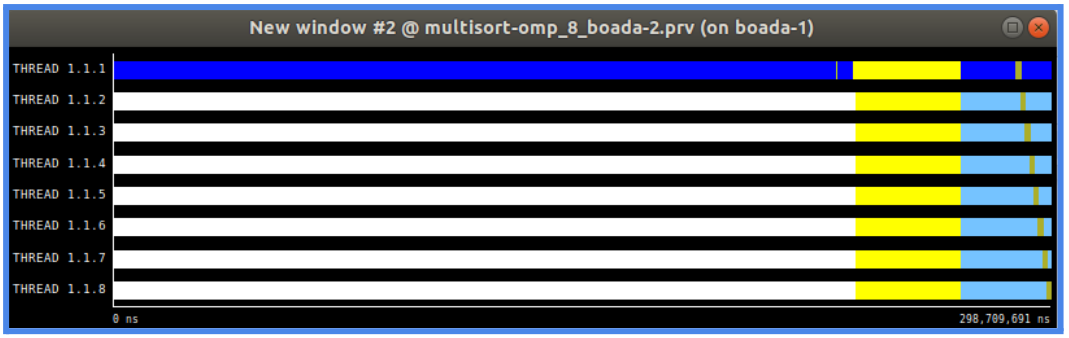


Image 3.0.8, paraver simulation with 8 processors.

Comparing plots leaf vs tree:

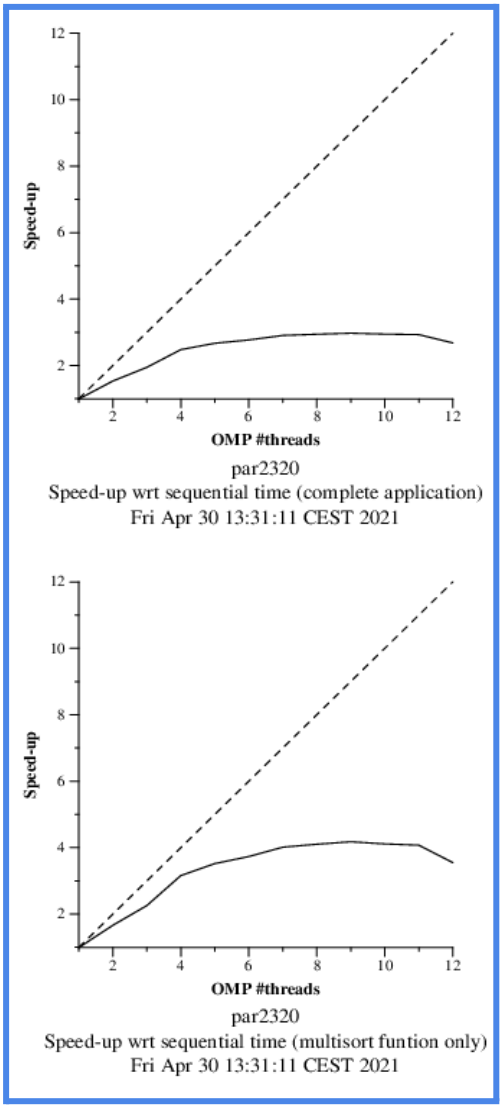


Image 3.0.9 scalability plots for leaf strategy

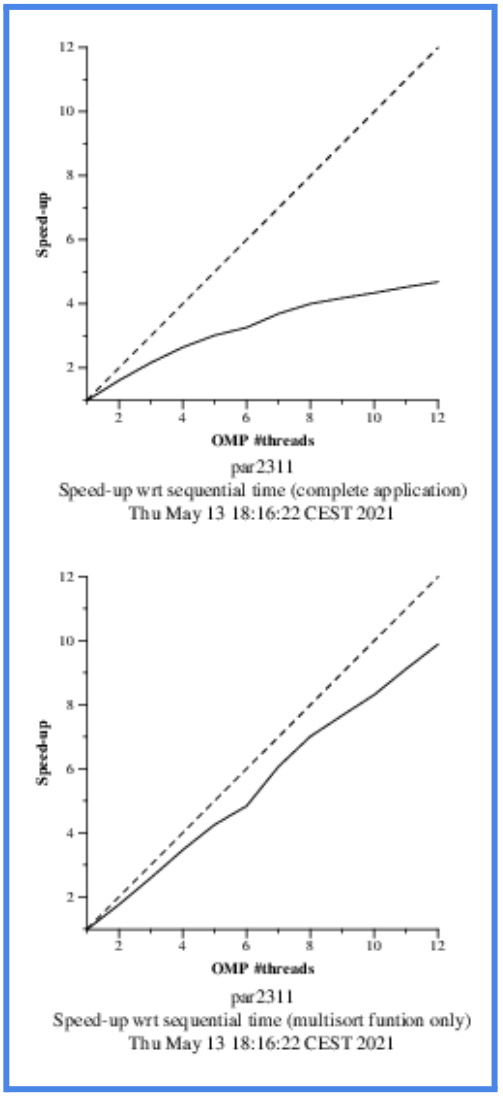


Image 3.0.10 scalability plots for tree strategy

In this part of the practice we have developed the tree strategy, the one that consists of making tasks all along the “tree”, in which every part takes place. As we can see on image 3.0.5 there are all the parameters which informs about the execution and later each of the time that takes every part of the strategy.

Later on we have modified the code to make possible the strategy tree as are shown in images 3.0.6 and 3.0.7. In the first one, for each call, we execute a task and when the calls are done we need to introduce taskwaits to make sure that all the task waits. It's important to take this into account because the different levels need to be finished at time so the next calls can be executed properly. In the second one we added the parallel region and a single to maintain correctly the execution of threads.

Moreover we have executed a simulation on paraver with 8 processors, as shown in image 3.0.8, and on it we can appreciate that the first thread initializes the structure and later the rest develop the parallel part so each one runs their tasks. Finally they end up with a synchronization.

Once we have developed the different plots for the leaf and tree strategies, we can see that the results are quite different as pictures 3.0.9 and 3.0.10 shows. Inthe first plot of the image 3.0.9 we can appreciate that as increasing the number of threads, there is a point where the speed-up doesn’t increase, so keeps the main result for a higher number. Later on for the multisort plot we can see a similarity between the plot commented before but we can appreciate that there is a significant pronunciation on the curve, but as said before there’s also a point where the increasing of the speed-up stops and decreases. This is because there is a lower number of tasks created.

In the other hand, in the first plot of the tree strategy we can see that as increasing the number of threads, the speed-up value also keeps increasing. That is because by having a higher number of threads and also developing a high number of tasks we can develop a better optimization of our resources. Also in the second plot, for the multisort part we can appreciate that it tries to keep its development quite the same as the lineral case, so shows us again that developing a higher number of tasks, if we increase the number of threads we can approach our resources.

4.3 Task granularity control: the cut-off mechanism

```
//CUTT OF
//------------
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int coff) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                if (!omp_in_final()) {
                        #pragma omp task final (coff >= CUTOFF)
                        merge(n, left, right, result, start, length/2, coff+1);
                        #pragma omp task final (coff >= CUTOFF)
                        merge(n, left, right, result, start + length/2, length/2, coff+1);
                        #pragma omp taskwait
                }
                else {
                        merge(n, left, right, result, start, length/2, coff+1);
                        merge(n, left, right, result, start + length/2, length/2, coff+1);
                }
        }
}
```

Image 3.1.1 code snippet of cut-off in merge function

```
void multisort(long n, T data[n], T tmp[n], int coff) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                if (!omp_in_final()) {
                        #pragma omp task final (coff >= CUTOFF)
                        multisort(n/4L, &data[0], &tmp[0], coff+1);
                        #pragma omp task final (coff >= CUTOFF)
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], coff+1);
                        #pragma omp task final (coff >= CUTOFF)
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], coff+1);
                        #pragma omp task final (coff >= CUTOFF)
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], coff+1);

                        #pragma omp taskwait

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

                        #pragma omp taskwait

                        #pragma omp task final (coff >= CUTOFF)
                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, coff+1);
                        #pragma omp taskwait
                }
                else {
                        multisort(n/4L, &data[0], &tmp[0], coff+1);
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], coff+1);
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], coff+1);
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], coff+1);
                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, coff+1);
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, coff+1);
                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, coff+1);
                }
        } else {
                // Base case
                basicsort(n, data);
        }
}
//------------
```

Image 3.1.2 code snippet of cut-off in multisort function

Image 3.1.3, paraver execution with 8 processors, task generation level (i.e. -c 0)

Image 3.1.4, paraver execution with 8 processors, task generation level (i.e. -c 1)

Now we have implemented the cut-off mechanism in order to increase the task granularity, so it allows us to control the maximum recursion level for the task generation.

As we can see on images 3.1.1 and 3.1.2 we have included in the tree code strategy to cut-off mechanism. To implement that we need to know that there would be a point where the final part we wouldn't generate more tasks, so to cover it we have reached the omp_in_final to mark the parts that we want to establish as a final or not. Also for the ones that are not final we need to include a variable which is going to be compared with the parameter of the cut-off to assign up to a level to reach.

Once we have compiled the code we have executed some paraver simulations with a number of 8 processors to see the difference between giving the cut-off mechanism or not in the strategy. In the first simulation, picture 3.1.3, we can see that there's not a cut-off implementation because we continue producing tasks on each recursion level, so we aren't controlling the maximum recursion task generation.

However in the second simulation, picture 3.1.4, we can appreciate that there is a cut-off mechanism because we have developed less tasks, they have established a better optimization of the task generation and finally can have the control of the maximum level for the task generation.

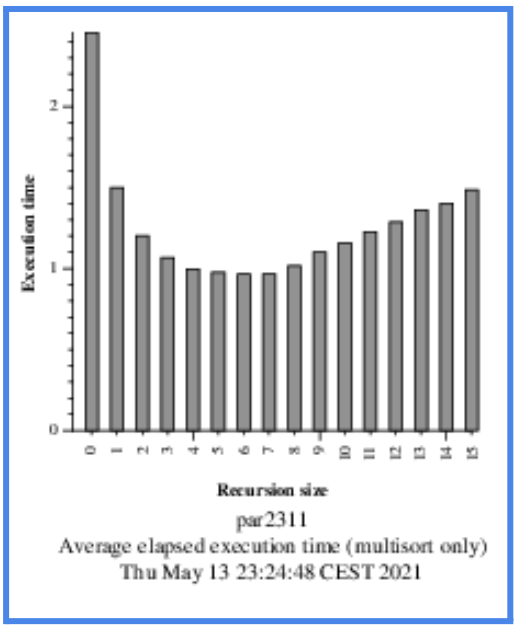


Image 3.1.5, multisort-omp-8-cutoff.ps

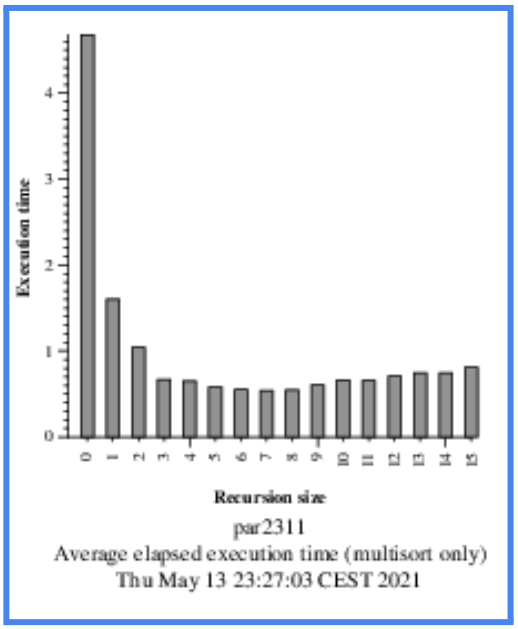


Image 3.1.6, multisort-omp-32-cutoff.ps

In this part we have to reach which of the values for cut-off are the better ones to consider, so we have executed the submit-cutoff-omp script with 8 and 32 processors, as we can see on images 3.1.5 and 3.1.6, and the value that we are looking for would be 7, that is the one in the graphics where there is a point of decreasing from the left and increasing to the right.

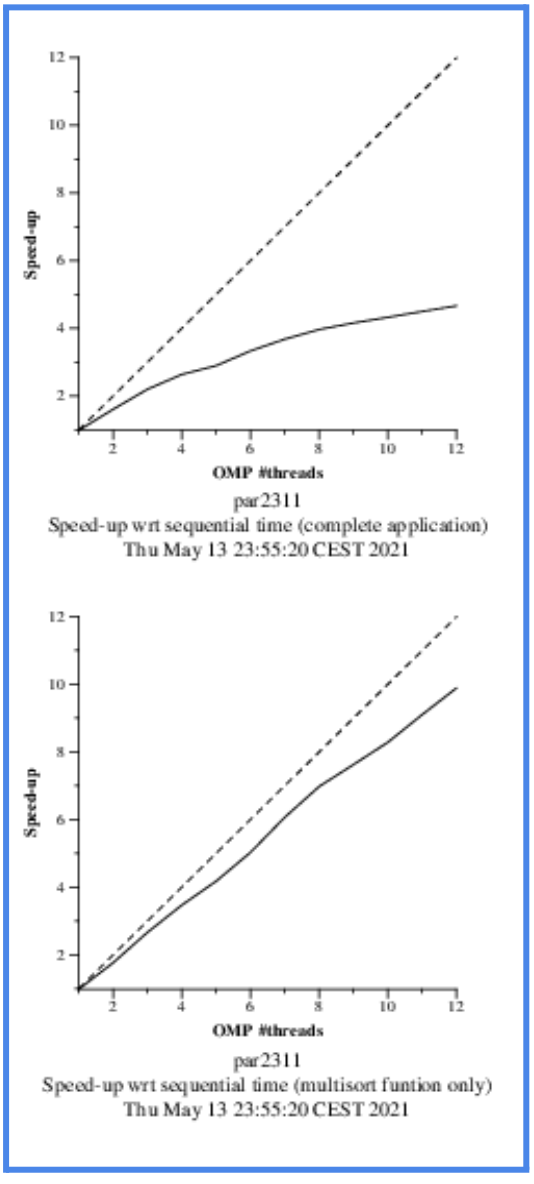


Image 3.1.7, plot of cut-off mechanism without modifications

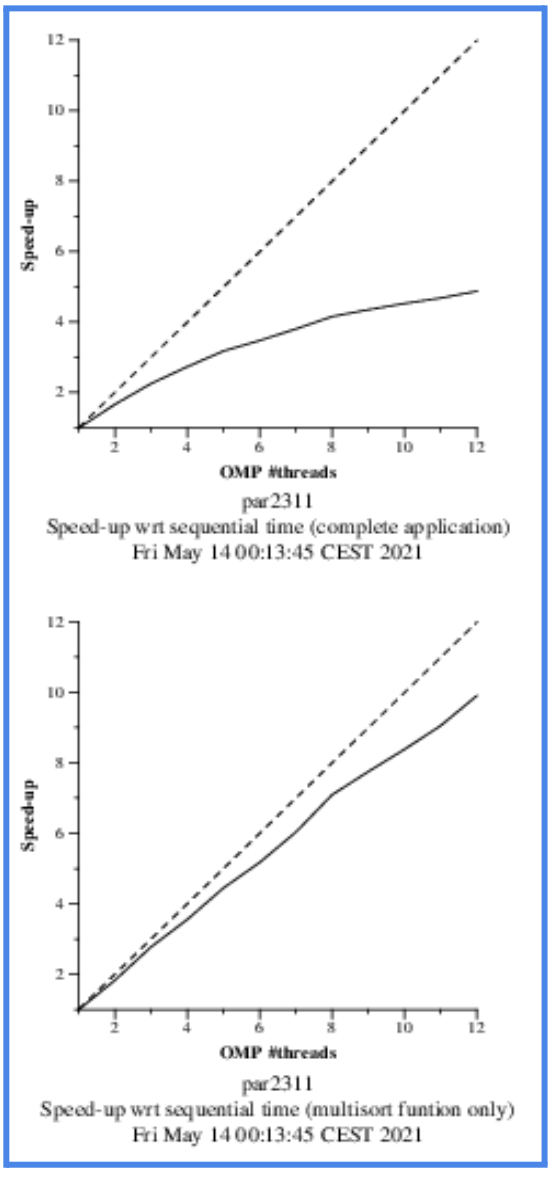


Image 3.1.8, plot of cut-off mechanism modifying the corresponding parametres

In this section we have upgraded some of the parameters for the execution, as said in the practice, the sort_size and merge_size values to 128 and cut-off to 7. On image 3.1.7 there is the original execution with the same values as seen before. But on image 3.1.8 there are the changes commented before and we can appreciate in the first plot a slightly more pronounced curvature compared to the other image, and also for the second graphic there is a closer approximation to the linear line compared to the other image. Moreover we can say that shapes are quite similar, so there's not a high upgrade.

Opcional 1:

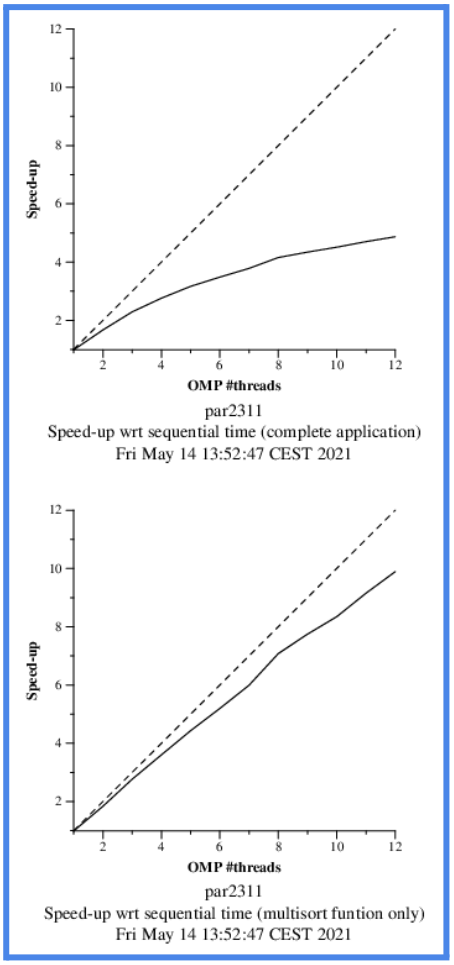


Image 3.2.1, plot of the cut-off mechanism with 24 processors

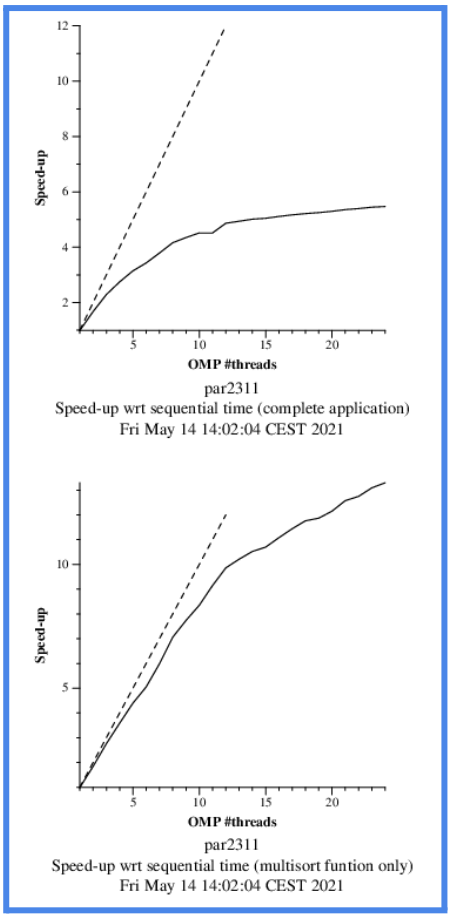
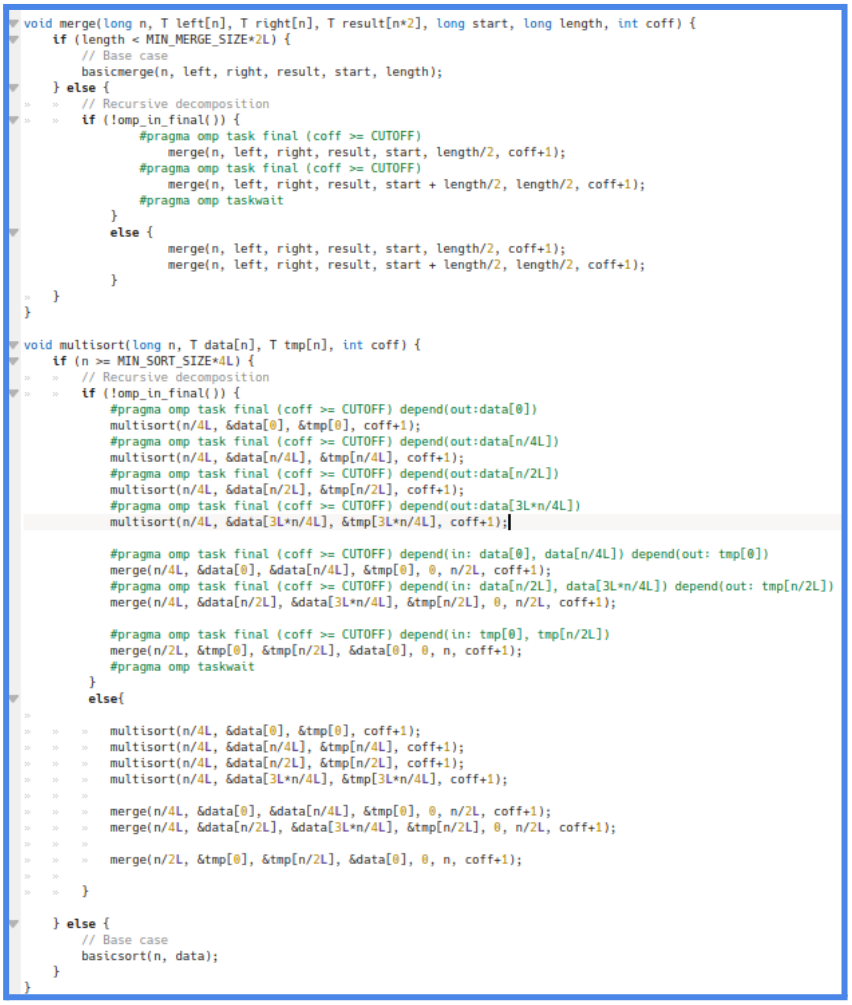


Image 3.2.2, plot of the cut-off mechanism with 24 processors and np_NMax = 24

Now, in this optional part we have developed a plot scalability of the tree implementation using cut-off and executed with 24 threads, as it's shown on image 3.2.1. This one is quite similar to the image 3.1.8 because we can appreciate such an identical shape.

However if we change the np_NMax, which refers to the maximum number of cores to be used, the plot changes into the result of image 3.2.2. As having increased the respective number from 12 to 24 we can see significant changes. In the first plot it seems to grow up in a faster way and later it maintains increasing little by little respect for the other one. Furthermore in the second graphic we can appreciate that with a less number of threads, respect the other image, as increasing them, the speed-up grows up more faster. It keeps increasing, we believe that the responsibility of such change its about the number of cores that are involved.

4 Using OpenMp task dependencies



```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int coff) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        if (!omp_in_final()) {
                #pragma omp task final (coff >= CUTOFF)
                    merge(n, left, right, result, start, length/2, coff+1);
                #pragma omp task final (coff >= CUTOFF)
                    merge(n, left, right, result, start + length/2, length/2, coff+1);
                #pragma omp taskwait
            }
            else {
                    merge(n, left, right, result, start, length/2, coff+1);
                    merge(n, left, right, result, start + length/2, length/2, coff+1);
            }
    }
}

void multisort(long n, T data[n], T tmp[n], int coff) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if (!omp_in_final()) {
            #pragma omp task final (coff >= CUTOFF) depend(out:data[0])
            multisort(n/4L, &data[0], &tmp[0], coff+1);
            #pragma omp task final (coff >= CUTOFF) depend(out:data[n/4L])
            multisort(n/4L, &data[n/4L], &tmp[n/4L], coff+1);
            #pragma omp task final (coff >= CUTOFF) depend(out:data[n/2L])
            multisort(n/4L, &data[n/2L], &tmp[n/2L], coff+1);
            #pragma omp task final (coff >= CUTOFF) depend(out:data[3L*n/4L])
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], coff+1);

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

            #pragma omp task final (coff >= CUTOFF) depend(in: tmp[0], tmp[n/2L])
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, coff+1);
            #pragma omp taskwait
         }
         else{

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

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

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

        }

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

Image 4.0.1, code for the tree strategy with tasks dependencies

In this part we were asked to implement and add into the code of the tree strategy and cut-off mechanism their tasks dependencies. Here in image 4.0.1 we can see the corresponding implementation that fusion the strategy asked before. To develop the dependencies we have helped form the image 1.6, from tareador, to see the role of the tasks being of types: in, out, inout.

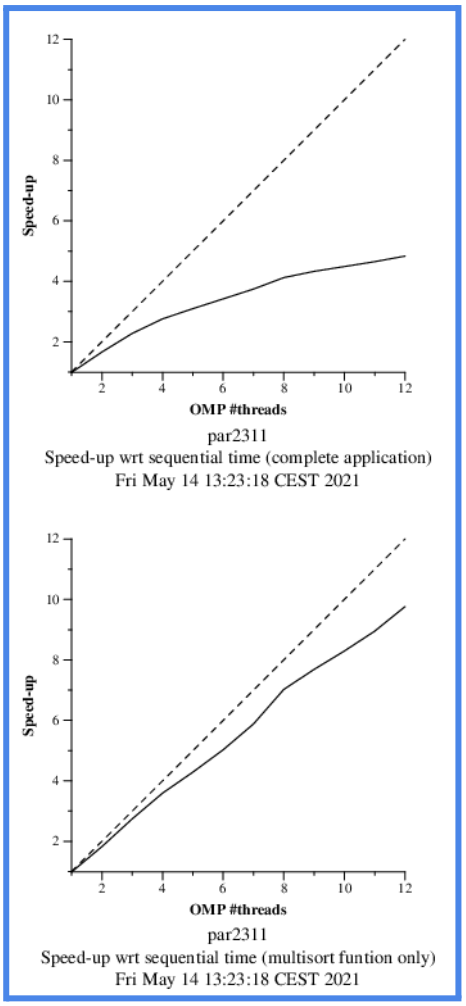


Image 4.0.2, plot of the tree strategy dependencies with 8 processors

Once we have executed our code we have simulated in paraver the strategy commented before with 8 processors as we can appreciate in image 4.0.2. Now we can see in the first plot that the curve keeps having the pronunciation that the others had.

Moreover it's the same case for the second plot, in which the shape tries to keep this similarity to the linear line. Here the speed-up increases with a higher number of threads thanks to the strategy applied that takes the dependences and the cut-off. We have believed that the results would take a better upgrade.

Image 4.0.3, paraver execution of the tree strategy dependencies with 8 processors without cutoff mechanism

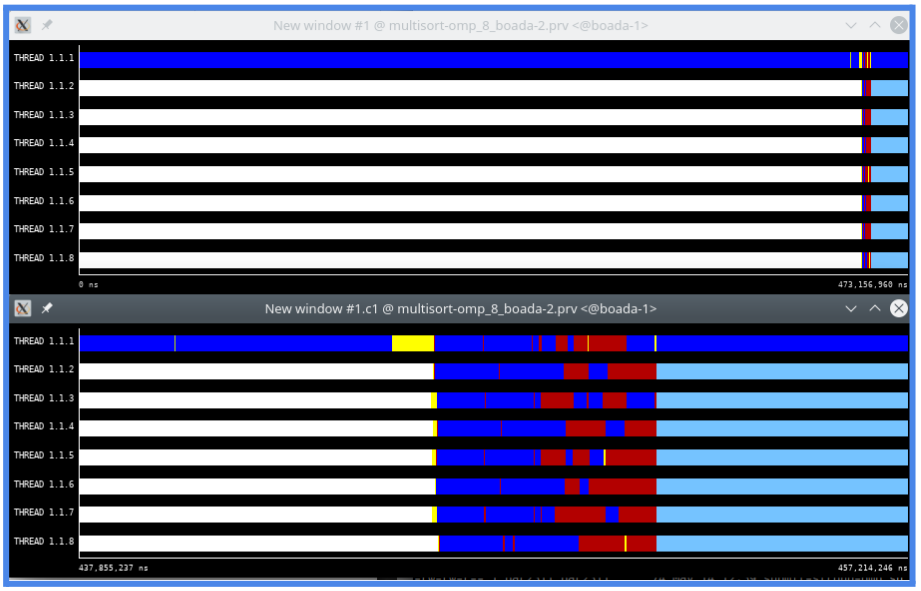


Image 4.0.4, paraver execution of the tree strategy dependencies with 8 processors with cutoff mechanism

At this point, to make sure that the dependencies take effect, we have executed some simulations in paraver to make visible the cases. In the first image (picture 4.0.3) we don't have added the cut-off mechanism while in the second one (picture 4.0.4) we added it. We can see a clear difference between both plots, in the first one we have developed more tasks, taking more time, while in the second, all the strategy has significantly reduced the generation of tasks. It's amazing how the change is significant.

Optional 2:

```
static void initialize(long length, T data[length]) {
    long i;
    long hw_many = (long) omp_get_num_threads();
    long chunk_size = length / hw_many;
    #pragma omp taskloop grainsize(chunk_size)
    for (i = 0; i < length; i++) {
        if (!(i%chunk_size)) {
            data[i] = rand();
        } else {
            data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }
}
```

Image 4.1.1, modified code of the initialization

In this part of the practice we have developed a new initialization of the vector, as the exercicie saids (image 4.1.1). The original code initializes only the first element of the vector and later for each step that takes, it's needed to see the box before. In our modified version we have divided the vector in parts of the chunk_size, that it's the length divided by the number of threads. Once they are divided in chunks, the first one randomly takes the value and the rest depend on the value of the box before. Now it can be faster because they don't have to wait sequentially, so each thread can focus on a chunk, as we can appreciate on image 4.1.2.

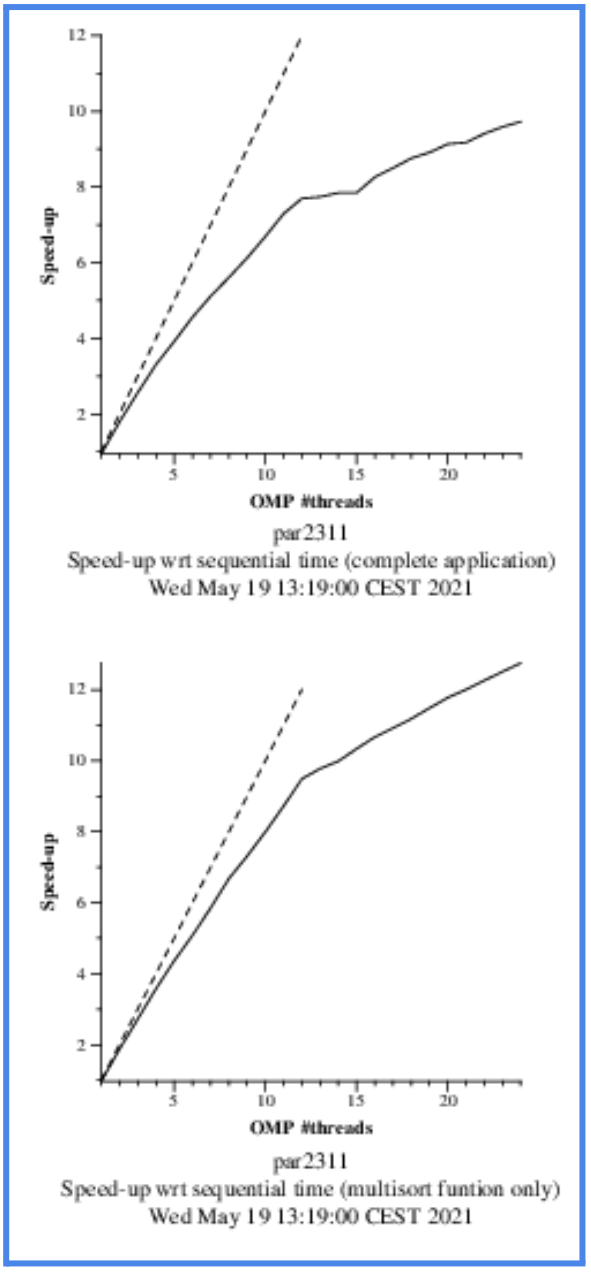


Image 4.1.2, plot of the optional part with 8 processors

As we expected we have a better increase of the speed-up while the number of threads increases, so as we divide the vector by the number of threads each can take a chunk and generate it more faster. On the image 4.1.2 we can see the results for the strategy implemented before with 8 processors. In the second plot we can clearly see the comments before but with a big pronunciation respect the others graphics. But in the first we can see clearly that as parallelizing the initialization, the upgrade is considerable because the complete application changes the way of processing to a parallel one.

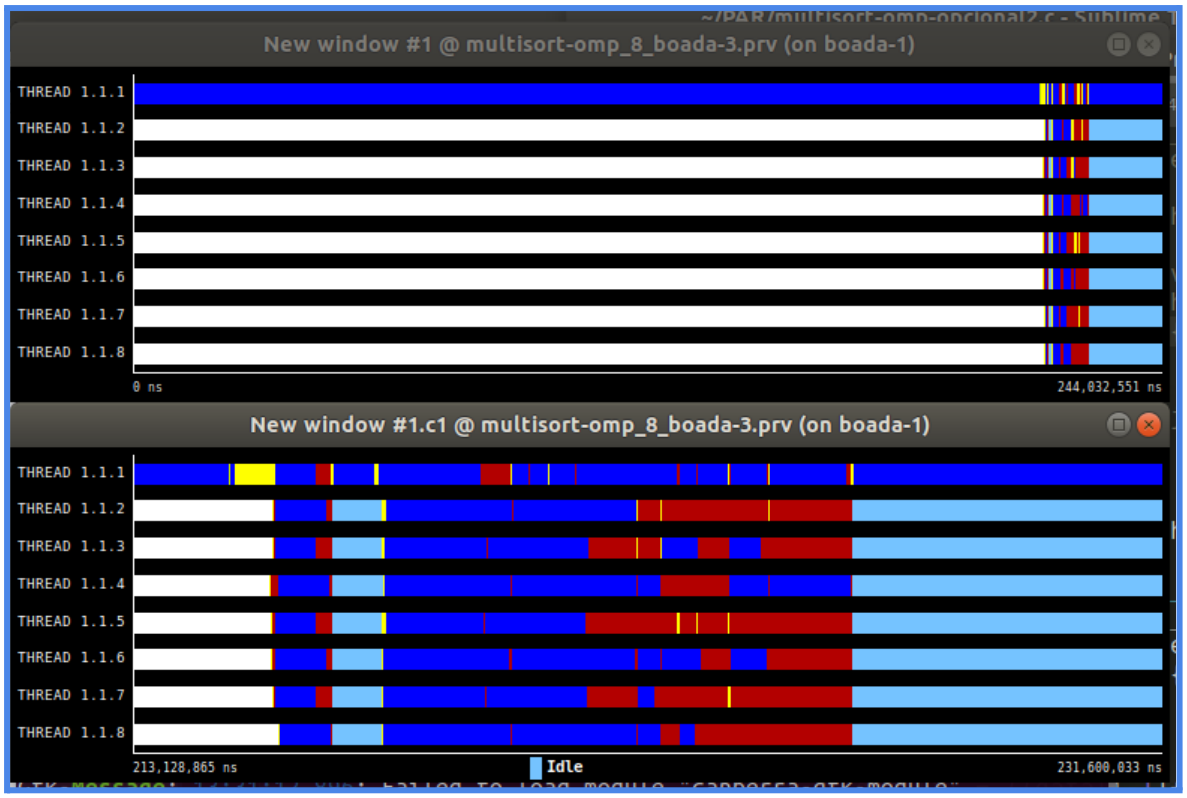


Image 4.1.3, paraver execution of the optional with 8 processors

Finally we have added some simulation that we have realized in paraver, as shows image 4.1.3. Now we can see that the yellow part appears less than the others graphics, it is the initialization, because it's parallelized. Again we can see a faster generation thanks to the strategy as shows the first plot.