PAR Laboratory Assignment Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Ismael Douha Prieto, Eloi Cruz Harillo - Group 1305

Q1 2019/20

1 Introduction

In this paper we will analyse the implementation of a sorting algorithm called Mergesort. This algorithm bases its implementation in a "divide and conquer" strategy. This strategy is based on the fact that in some cases the best way to solve a big and complex problem is to divide it to create smaller problems that will be easier to solve.

So given a disordered vector Mergesort is going tho divide it recursively until all the created subvectors reach a size that is reasonably small. When arrived at this point all of them are going to be ordered individually using the quicksort algorithm. The next step is to merge all the ordered subvector to get the final result. Int he Figure 1 we can see the scheme of how the algorithm works.

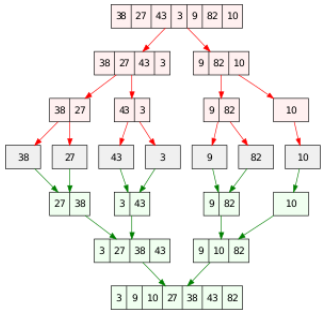


Figure 1: Graphic representation of the divide and conquer strategy used extracted from Wikipedia.

2 Parallelization strategies

As we have studied this course there are two strategies to parallelize recursive programs. They are tree and leaf strategies. In the next section we will analyse the performance of mergesort using both of them. But first we will use tareador to visualise the graph of tasks generated and and to look for dependencies between them.

So first of all we have created a parallel version of the mergesort program. Figure 2 shows where the calls to the Tareador API have been placed. As we can see every instance of the multisort function and every instance of the mere function are taks. This is shown in the Figure 3. Multisort tasks are represented in colour green and are the ones that divide the vectors of size n into vectors of size n/2. In the other hand merge tasks are represented by colour red. This ones are responsible to merge subvectors previously created once they had been ordered by quicksort. Arrived at this point we need to make a remark. For simplicity of the code the task is created once every instance of the function is created and called, so wee don't englobe the time of function creation, parameter passing and returning value of every function call. So if we wanted to get a more accurate representation of the program execution we would have to create the task before the function was called.

Now we will proceed to make an analysis of the Figure 3. The first thing that we will highlight is that tasks are represented inside other tasks because we are in front of a recursive graph. The parent task (the first of all created tasks) is represented by a blue rectangle. Te four green squares are the first four calls to multisort. The next calls to this functions are representated by the four green circles inside each of the previus squares. This are the three levels of problem division. So we can see that the calls to basic sort are ubicated in the same level wich means that doesn't exist dependences between them. This is because the divide and conquer strategy divides a big problem into independent problems of smaller size. In the other hand we can see that dependences exist between levels of the recursion because they are show in vertical. If wee focus our attention in the merge tasks they are executed iterativelly while subvectors are smaller than 2k after that they are executed recursivelly.

In conclusion if we are able to parallelize the program respecting the dependences mentioned above wee will achieve an usable and efficient parallel code.

```
void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long length);

void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        tareador_start_task("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");
}

void multisort(long n, T data[n], T tmp[n]) {
        tareador_start_task("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");
}
```

Figure 2: Code used for the task decomposition in Tareador.

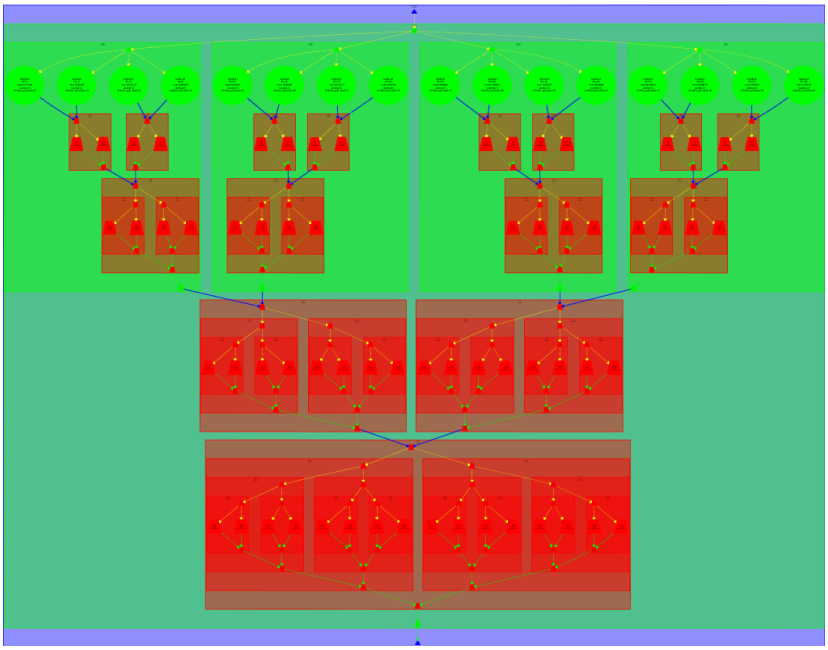


Figure 3: Graphic representation of the executed tasks by tareador.

Table 1 illustrates the times of execution of the parallel code using different number of threads. This times are approximations because the program is instrumented to create files that tareador will analyse to create the graphs and this includes an overhead. Answering to the question in the report, the results are ideal to the idle case. This is because as we increment the number of threads the speedup is increased until we arrive to 16 processors. All extra processors that we assign to the execution of the process will not help to increase the speedup. In the figure 4 the timelines for every execution are represented.

Number of threads	Execution time	Speed-up
64	1,28 s	15,86
32	1,28 s	15,86
16	1,28 s	15,86
8	2,55 s	8
4	5,08 s	4
2	10,15 s	2
1	20,3 s	1

Table 1: Execution times and Speed-ups in function of the number of threads.

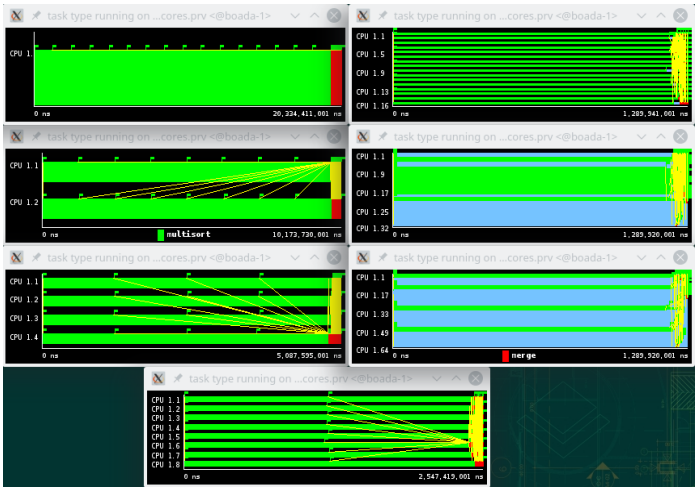


Figure 4: Timelines of the execution in different processors.

3 Performance evaluation

3.1 Leaf strategy

In this second section we will modify the given code to create two different versions of the parallel code. First we will begin creating a code that follows the leaf strategy. Figure 5 shows pragma clauses that we have added to the program. We can see that the tasks are created in the base case of both recursive functions. For this reason the calls to basicmerge and basicsort are the only ones that generate tasks. To make this code correct we need to add to extra synchronisation clauses. This is because the algorithm needs to wait until the vectors are ordered before proceeding to merge them. The taskwait clause ensures that.

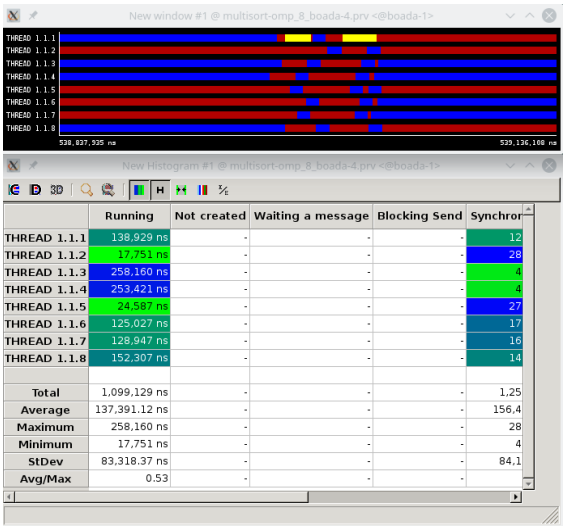
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long
length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                #pragma omp task
                basicmerge(n, left, right, result, start, length);

        } 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);

        }
}
```

Figure 5: Code used for the leaf strategy decomposition.

The next two images show us the results obtained after the execution of the program. Figure 6 shows the time of execution in the interactive node of the boada. Figure 7 is a more interesting one. From this image we can conclude that the leaf strategy is not a good one. We can identify two different types of tasks. The large blue lines this ones represent the execution of basic sort tasks. The little blue lines represent the basicmerge tasks. The first characteristic to describe is that the basicsort tasks take a lot bigger than the basicmerge tasks. Another important thing to keep in mind is that only four basicsort instances that can be executed simultaneously. So this the main inconvenient for this parallelization strategy. Before sorting a vector we have to wait until both subvectors are ordered and merged. This will cause a limitation in the speedup as we will see.

```
par1305@boada-1:~/lab4$ ./multisort-omp
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.819533
Multisort execution time: 1.670383
Check sorted data execution time: 0.027753
Multisort program finished
par1305@boada-1:~/lab4$
```

Figure 6: Execution time.



	Running	Not created	Waiting a message	Blocking Send	Synchron
THREAD 1.1.1	138,929 ns	-	-	-	12
THREAD 1.1.2	17,751 ns	-	-	-	28
THREAD 1.1.3	258,160 ns	-	-	-	4
THREAD 1.1.4	253,421 ns	-	-	-	4
THREAD 1.1.5	24,587 ns	-	-	-	27
THREAD 1.1.6	125,027 ns	-	-	-	17
THREAD 1.1.7	128,947 ns	-	-	-	16
THREAD 1.1.8	152,307 ns	-	-	-	14
Total	1,099,129 ns	-	-	-	1,25
Average	137,391.12 ns	-	-	-	156,4
Maximum	258,160 ns	-	-	-	28
Minimum	17,751 ns	-	-	-	4
StDev	83,318.37 ns	-	-	-	84,1
Avg/Max	0.53	-	-	-	

Figure 7: Execution data extracted from paraver.

Finally we will talk about the strong scalability plot shown in the figure 8 taking in mind explications that we have done above. As we can see this program doesn't has strong scalability. The image shows us to plots, the first one shows the strong scalability for all the code. The second plot shows the strong scalabilty of the multisort function. As we can see both of them can achieve a maximum speedup of 4. This happens because at the end of the function multisort we must add a taskwait clause that will let the program be executed using a maximum of four threads.

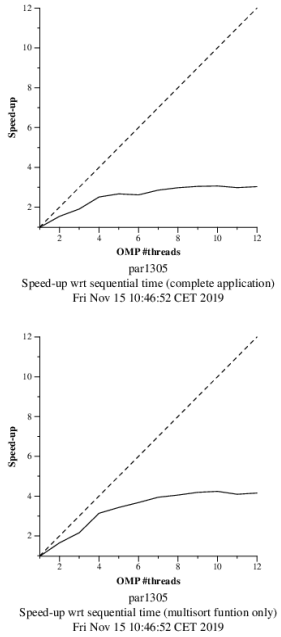


Figure 8: Strong scalability plot leaf strategy.

3.2 Tree strategy without cutoff

Now we have finished the analysis of the leaf strategy so we will proceed to analyse the code paralleled using the tree strategy. As we have done above we will start explaining the modifications that we have done to achieve an usable and efficient code. Figure 9 shows our implementation of the tree strategy decomposition. We can see that we create a task for every recursive call to multisort and merge instead of creating a task only for the base case (basicmerge and basicsort). Doing this we achieve a task graph as the one shown in the Figure 3 obtained from tareador. We have to remark that two synchronisation clauses had been added to the code. After doing the four calls to multisort because wee need to have all subvecors ordered before merging them and after mergering two halfs because given four ordered vectors we can only get one after they had been ordered two by two. The last task created in multisort is unnecessary because a taskwait is called just before. Finally we have to add that in the main function the first call to multisort is created as a task too.

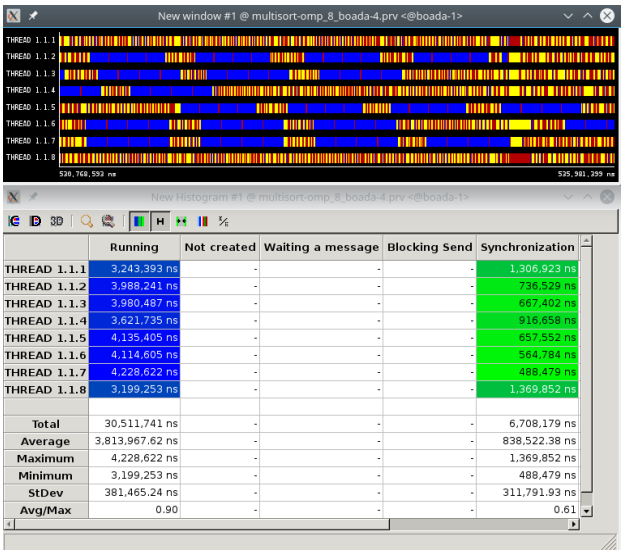
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case

                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);

        }
}
```

Figure 9: Code used for the tree strategy decomposition.

The results of the analysis are show in figures 10 and 11. The first one shows that multisort has been executed in approximately 0.4 seconds. This is a good result considering that in the previous section we achieved an execution time of 1.6 seconds.

```
par1305@boada-1:~/lab4$ ./multisort-omp
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.819922
Multisort execution time: 0.428735
Check sorted data execution time: 0.021335
Multisort program finished
```

Figure 10: Execution time.



	Running	Not created	Waiting a message	Blocking Send	Synchronization
THREAD 1.1.1	3,243,393 ns	-	-	-	1,306,923 ns
THREAD 1.1.2	3,988,241 ns	-	-	-	736,529 ns
THREAD 1.1.3	3,980,487 ns	-	-	-	667,402 ns
THREAD 1.1.4	3,621,735 ns	-	-	-	916,658 ns
THREAD 1.1.5	4,135,405 ns	-	-	-	657,552 ns
THREAD 1.1.6	4,114,605 ns	-	-	-	564,784 ns
THREAD 1.1.7	4,228,622 ns	-	-	-	488,479 ns
THREAD 1.1.8	3,199,253 ns	-	-	-	1,369,852 ns
Total	30,511,741 ns	-	-	-	6,708,179 ns
Average	3,813,967.62 ns	-	-	-	838,522.38 ns
Maximum	4,228,622 ns	-	-	-	1,369,852 ns
Minimum	3,199,253 ns	-	-	-	488,479 ns
StDev	381,465.24 ns	-	-	-	311,791.93 ns
Avg/Max	0.90	-	-	-	0.61

Figure 11: Execution data extracted from paraver.

In the Paraver trace we can observe that now the code uses more cores at the same time than leaf strategy (blue part). This happens because with this strategy we do not need to wait until the basicsort to create the task, every invocation to the function multisort is a task and this means that in the taskpool we have more than 4 tasks, as happened in leaf strategy

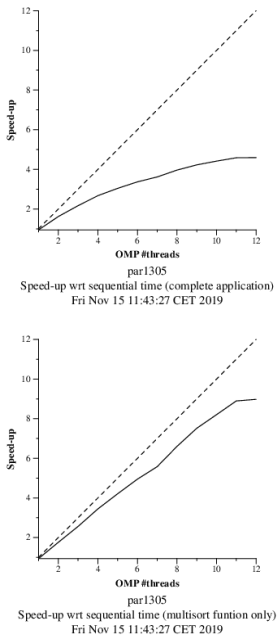


Figure 12: Strong scalability plot tree strategy.

With this plot we can see that now the code is nearly to a good scalability for the multisort function. However, for the whole code still performing bad. This happens because the initialisation of the vector is made sequentially. This problem we will solve it in the optional part.

3.3 Tree strategy with cutoff

The next part of this section will consists in the analysis of the same algorithm but adding a cut-off mechanism that will reduce the number of tasks created. The Figure 14 shows the code that we have used to implement it. The implementation consists in adding a new parameter in both multisort and merge functions. This parameter called depth indicates in which level of recursion we are in each moment and it is incremented for every recursive call. In the creation of every task a final clause is added that checks if the actual depth is greater than a given depth that will mark the end of the recursion. When the result of the call `om_in_final` is true no more tasks will be created. As we has explained in the previous section three synchronisation clauses are necessary. They consist in adding taskwait clauses after the two merge calls and the four multisort calls.

```
File  Edit  View  Search  Tools  Documents  Help
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 {
                if(omp_in_final()){
                        merge(n, left, right, result, start, length/2, depth+1);
                        merge(n, left, right, result, start + length/2, length/2, depth+1);
                } else {
                        // Recursive decomposition
                        #pragma omp task final (depth >= CUTOFF) mergeable
                        merge(n, left, right, result, start, length/2, depth+1);
                        #pragma omp task final (depth >= CUTOFF) mergeable
                        merge(n, left, right, result, start + length/2, length/2, depth+1);
                        #pragma omp taskwait
                }
        }
}

void multisort(long n, T data[n], T tmp[n], int depth) {

        if (n >= MIN_SORT_SIZE*4L) {
                if(omp_in_final()) {
                        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 {
                        // Recursive decomposition
                        #pragma omp task final (depth >= CUTOFF) mergeable
                        multisort(n/4L, &data[0], &tmp[0], depth+1);
                        #pragma omp task final (depth >= CUTOFF) mergeable
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
                        #pragma omp task final (depth >= CUTOFF) mergeable
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
                        #pragma omp task final (depth >= CUTOFF) mergeable
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
                        #pragma omp taskwait

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

Figure 13: Code used for the tree strategy decomposition with cut-off.

To begin with we will execute the program using a maximum recursion level of 0. This means that only 4 tasks of multisort will be created. Figure 14 shows that the time of execution of the multisort is of 2.76 seconds. In comparison with the time obtained in the previous section (0.4 seconds) we notice that this is not a good value for the cut-off algorithm.

```
par1305@boada-1:~/lab4$ ./multisort-omp -c 0
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=0
Initialization time in seconds: 0.820289
Multisort execution time: 2.768724
Check sorted data execution time: 0.025594
Multisort program finished
```

Figure 14: Execution time.

The paraver trace shows us why the execution is two slow. We can see the 7 tasks created. The first 4 executed by threads 0 to 3 correspond to the 4

multisort tasks. The next three tasks executed by threads 2,3 and 5 correspond to the merge tasks. We are not exploiting the maximum grade of parallelism so we can arrive to the conclusion that we need to create more tasks in every execution to achieve better execution times.

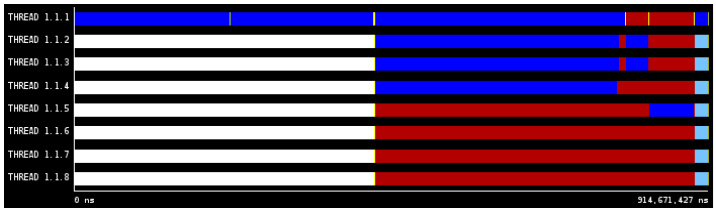


Figure 15: Execution data extracted from paraver.

As the results found previously are not good we have to look for the optimal depth. To do so we will submit the `submit-cutoff-omp.sh`script. Graphic shown in the Figure 16 shows the execution time using different cut-off values. In our case we can see that the level 5 is the one that gives us a better performance.

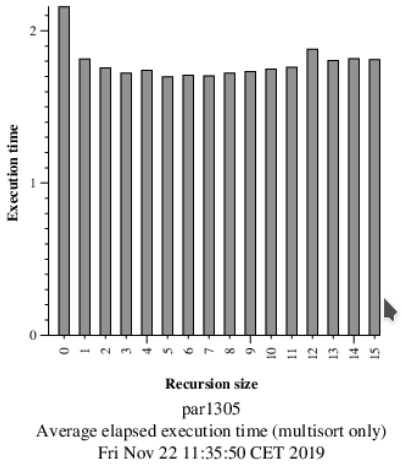


Figure 16: Execution data extracted from paraver.

Fially Figure 17 shows the strong scalability plot creating taks until we arrive to the level 5 of recursion. As we has explained in previous sections the first image corresponds to the scalabitity plot for the first program and it is not very good because the initialisation of the vector is made in a sequential way and it reduces a lot the scalability. The second plot is the one that is interesting for our analysis. It shows that the strong scalability four our multisort function is very good and it is a little better that the strong scalability of the recursive algorithm without any depth limit.

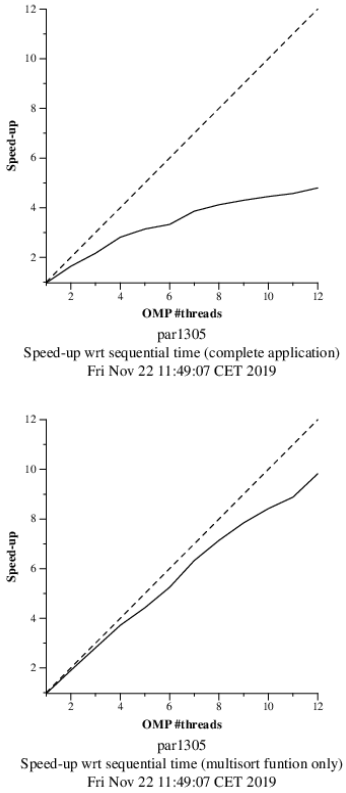


Figure 17: Strong scalability plot tree strategy with cutt-off.

3.4 Tree strategy with depend clauses

The last part of this section will consist in control the synchronisation using depend clauses instead of using taskwait clauses and analyse the performance given by this strategy. Figure 18 shows our implementation. For every multisort task we create an out dependence that at the same is an in dependence for merge task. Doing this for every multisort call we can delete the taskwait clause because merge statement only will be executed when the multisort call from which depend it finishes. The first step after finishing the implementation has been executing it with 8 processors to check that there are no problems in the sorting algorithm. Figure 19 sows that the program finishes with no errors. The main characteristic of this execution is that tasks are very little and that exists a lot of time of synchronisation between them as it is shown in Figure 20.

```
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]);

            #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);
            #pragma omp task depend(in: tmp[0], tmp[n/2L])
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
            #pragma omp taskwait //wait until merged

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

    }
}
```

Figure 18: Code used for the tree strategy decomposition using depend clauses.

```
par1305@boada-1:~/lab4.depend$ OMP_NUM_THREADS=8 ./multisort-omp
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.824667
Multisort execution time: 0.842477
Check sorted data execution time: 0.017485
Multisort program finished
```

Figure 19: Execution time and verification of the execution.

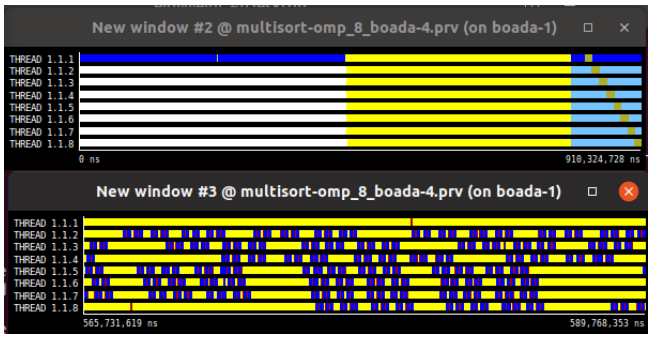


Figure 20: Execution data extracted from paraver.

Figure 21 shows the strong scalability plot of the program described above. This image is surprisingly similar to the ones in the previous sections. It shows a very good strong scalability. We achieve a speed-up of 10 wen we use 12 processors in the multisort function.

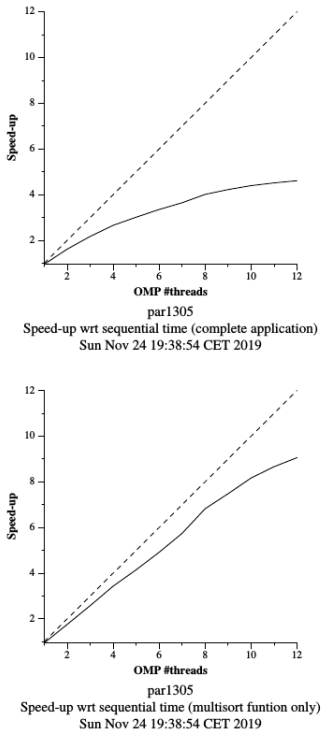
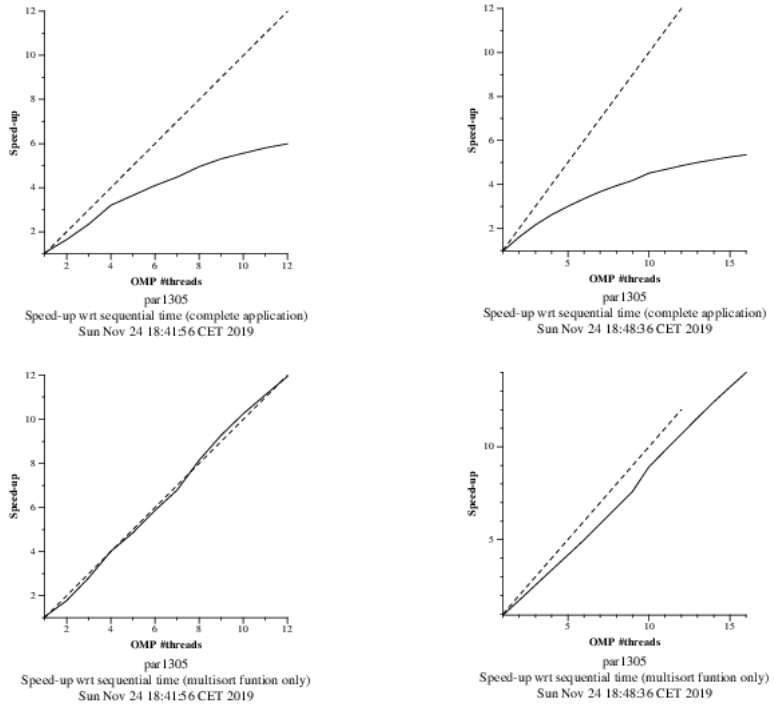


Figure 21: Strong scalability plot tree strategy with depend clauses.

3.5 Optional 1

As is required in the assignment, we did the analysis for the tree stretegy code submitting the code to all the nodes of the boada. We include only the plot for boada 5 and boada 6-8 because for the boada 1-4 we did the analysis in the section 3.2.



(a) Boada 5

(b) Boada 6-8

Figure 22: Strong scalability for the boada nodes

For the submission to boada 6-8 node, we needed to edit the *submit-strong-omp.sh* file, because this node has 16 cores instead of the 12 cores of the rest of the nodes.

```
#!/bin/bash
### Directives pel gestor de cues
# following option makes sure the job will run in the current directory
#$ -cwd
# following option makes sure the job has the same environment variables as the submission shell
#$ -V
# Per canviar de shell
#$ -S /bin/bash

SEQ=multisort
PROG=multisort-omp

size=32768
sort_size=1
merge_size=1
cutoff=4

np_NMIN=1
np_NMAX=16
N=3

# Make sure that all binaries exist
make $SEQ
make $PROG

HOST=$(echo $HOSTNAME | cut -f 1 -d'.')

out=$PROG-strong_${HOST}.txt    # File where you save the execution results
aux=./tmp.txt           # archivo auxiliar

outputpath1=./speedup1.txt
outputpath2=./speedup2.txt
outputpath3=./elapsed.txt
```

Figure 23: submit-strong-omp.sh file edited in order to use all the available cores

3.6 Optional 2

For this optional part, we parallelized the initialisation of the vectors *data* and *tmp*. For the *data* vector we can not do the same initialization as the sequential version because that code is unparallelizable. In this case, we did the same as the sequential code but in a thread level.

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

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

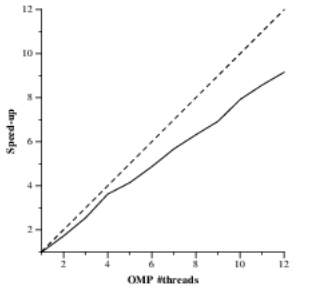
Figure 24: Code of the parallelization of the vector initialisation

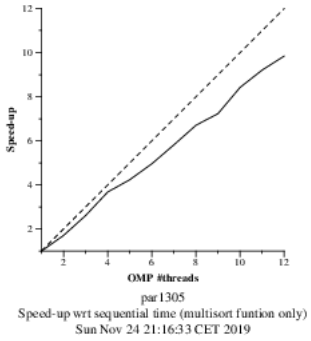
Once the parallelization is done, we checked that the execution time improved.

```
par1305@boada-1:~/lab4$ ./multisort-omp
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.076879
Multisort execution time: 0.419655
Check sorted data execution time: 0.025697
Multisort program finished
```

Figure 25: Execution time with the initialization of the vector parallelized

Finally, we analysed the code with using strong scalability and paraver tools.





(a) Strong sclalability plot

(b) Paraver

Figure 26: Analysis of the initial vector parallelization

We deduce that the parallelization happens because the speed-up for the complete code is near to the perfect scalability. However, when was not parallelized was far from there. Also with the paraver we see that the initialisation function has been shared between the eight threads instead of being executed only with the first thread.

4 Conclusions

In conclusion, between the two main strategies that we can apply to a sequential code, tree is which obtain a better performance. Also for tree strategy, the variant without cutoff is the best, getting a huge difference in comparison to the rest of options.