PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

15/12/2021

Fall 2021-22

Index

1. Introduction 1

2. Task decomposition analysis for Mergesort 2

2.1. Leaf strategy 2

2.2. Tree strategy 3

2.3. Conclusions 4

3. Shared-memory parallelisation with OpenMP tasks 5

3.1. Leaf strategy in *OpenMP* 5

3.2. Tree strategy in *OpenMP* 7

3.3. Task granularity control: the cut-off mechanism 8

4. Using OpenMP task dependencies 12

5. Optionals 14

5.1. Optional 1 14

5.2. Optional 2 16

6. Conclusions 19

1. Introduction

In this laboratory assignment we explore the use of parallelism in recursive programs. Recursive task decompositions are parallelisation strategies that try to exploit this parallelism. Let's say that we have use a "divide and conquer" algorithm and by decomposing this problem we obtain the following approach.

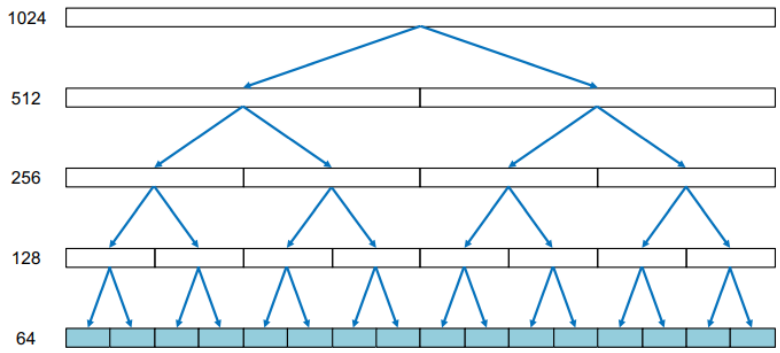
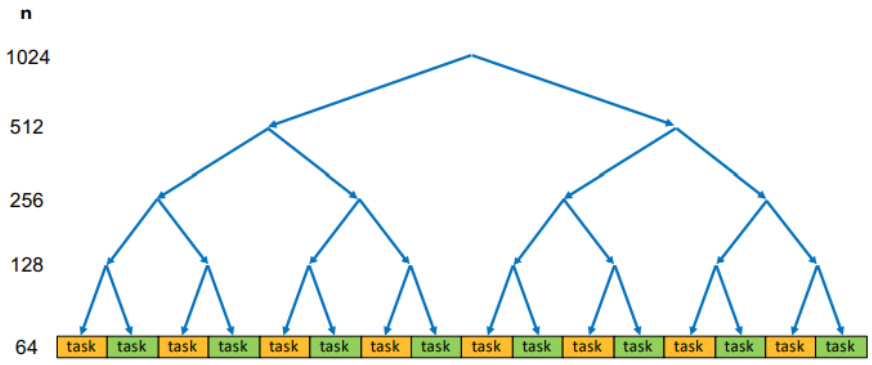


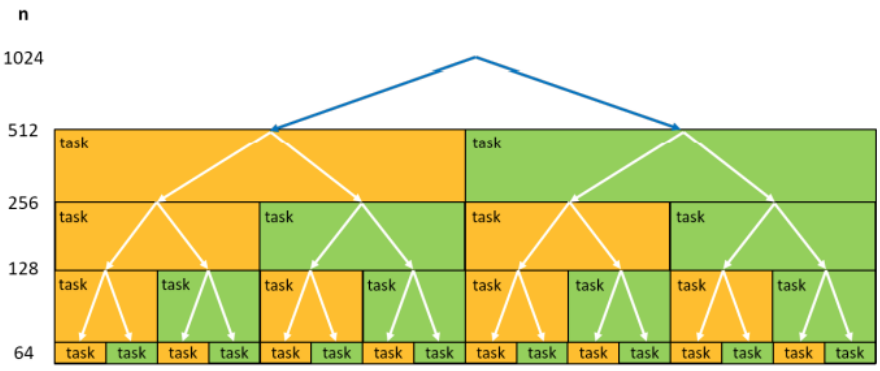
Figure 1 – Divide-and-conquer approach

The problem can follow the next to following recursive task decomposition strategies:

- A *leaf recursive task decomposition* a new task is generated every time the recursion base case is reached (i.e. a leaf in the recursion tree is reached).



- A *tree recursive task decomposition* a new task is generated every time a recursive call is performed (i.e. at every internal branch in the recursion tree).



2. Task decomposition analysis for Mergesort

2.1. Leaf strategy

Here we have the part of the code that has been modified.

```
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
        tareador_start_task("basicmerge");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("basicmerge");
    } else {
        …
    }
}

void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        …
    } else {
        // Base case
        tareador_start_task("basicsort");
        basicsort(n, data);
        tareador_end_task("basicsort");
    }
}
```

The results of the modifications with Tareador and Paraver.

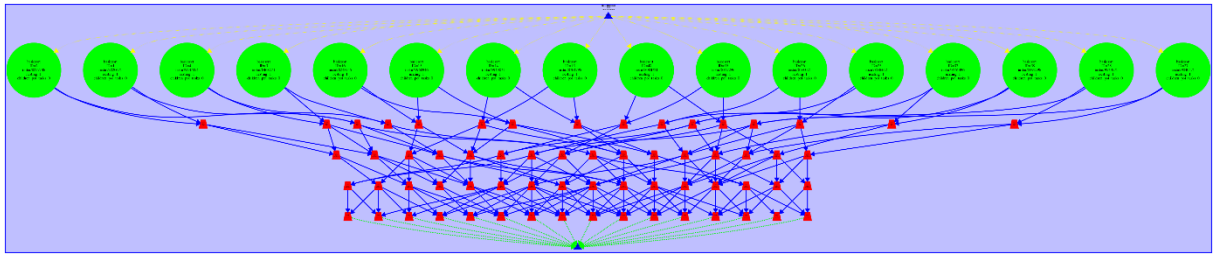


Figure 2.1.1 – Leaf strategy dependency graph

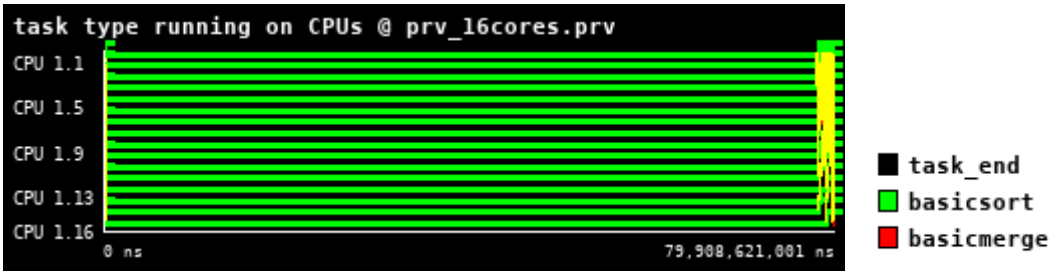


Figure 2.1.2 – Leaf strategy with *Paraver* simulation with 16 processors

2.2. Tree strategy

Here we have the part of the code that has been modified

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        …
    } 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/2);
        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("first_merge");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("first_merge");
        tareador_start_task("second_merge");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("second_merge");

        tareador_start_task("third_merge");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("third_merge");
    } else {
        …
    }
}
```

The results of the modifications with Tareador and Paraver.

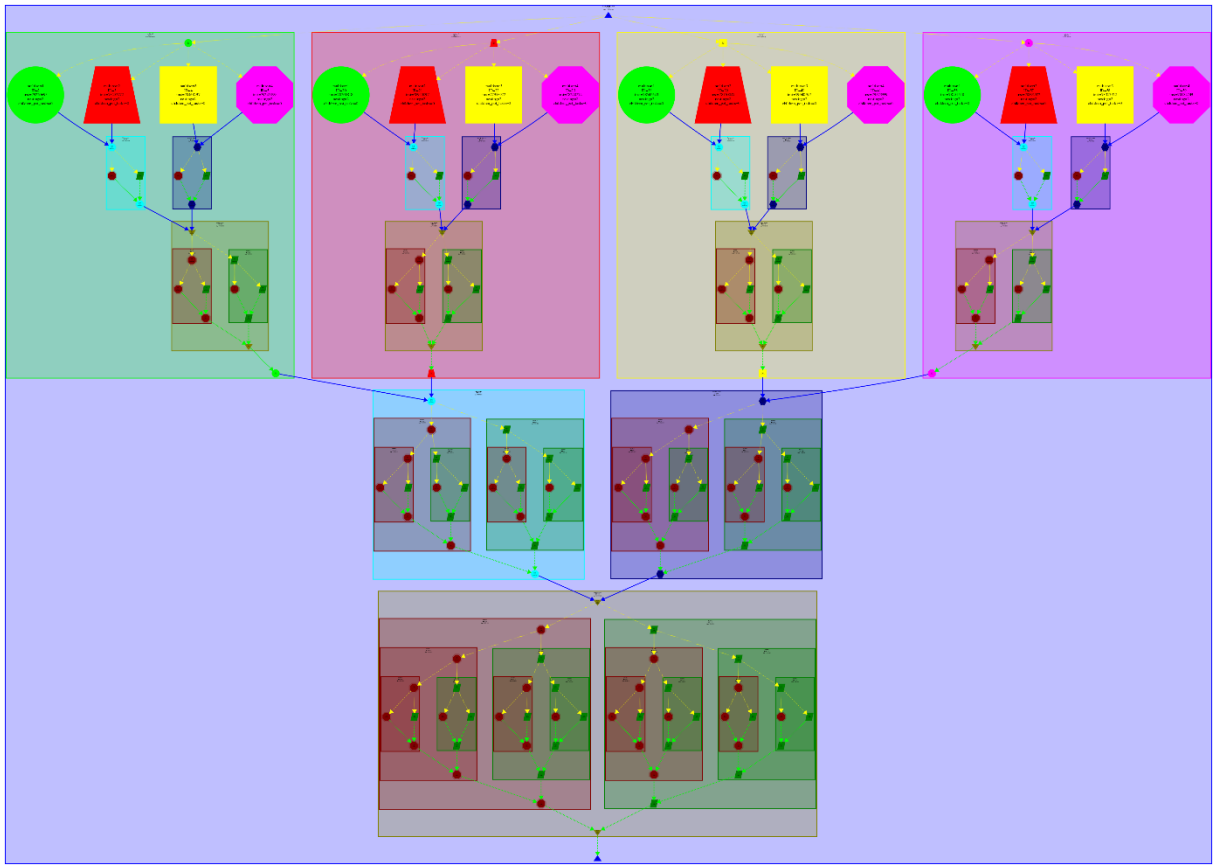


Figure 2.2.1 – Tree strategy dependency graph

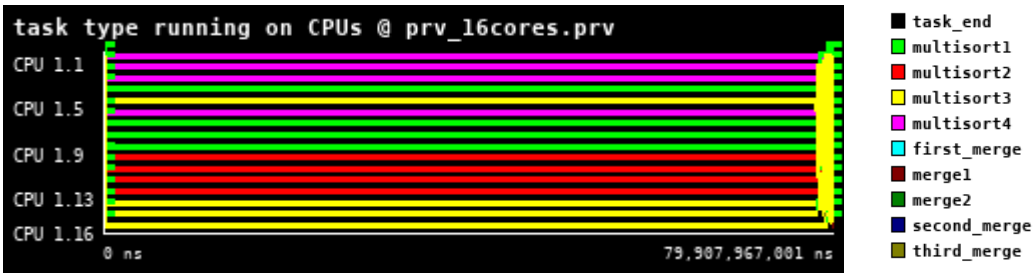


Figure 2.2.2 – Tree strategy with *Paraver* simulation with 16 processors

2.3. Conclusions

If we look at the *Tareador* dependency graph of both strategies we notice instantly that the structure is totally different, because of the different strategies that we used, the leaf one is generating a new tasks every time that the base case is executed, generating tasks sequentially, and the tree strategy every time that the recursive case is executed, generating parallel tasks. The first strategy creates dependencies between tasks, the second strategy no.

Notice that in the first simulation we spend most of the time in *basicsort* function, and in the second one in *multisort* functions. The main difference is that the merge function is executed at the final of multisort functions for the tree strategy and for the leaf strategy the basicmerge no.

3. Shared-memory parallelisation with OpenMP tasks

In this second section of the laboratory assignment, we will parallelise the original sequential code in multisort.c using OpenMP, having in mind the conclusions we gathered from our analysis with *Tareador*. As In the previous section, two different parallel versions will be explored: leaf and tree.

3.1. Leaf strategy in *OpenMP*

The code modifications that we made.

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

Notice that for this strategy we decided to parallelize the base cases, in *merge* function the *basicmerge* function and in *multisortfunction basicsort* function, but when we tried to execute the code, we found that it gave us some errors, these one's corresponds to data races, to avoid this problem we decided to add some *taskwaits* in *multisort* function in the recursive descomposition.

After that, we executed the code with `submit-strong-omp.sh` script, that tries our code with a number of processors in the range 1 to 12 by default.

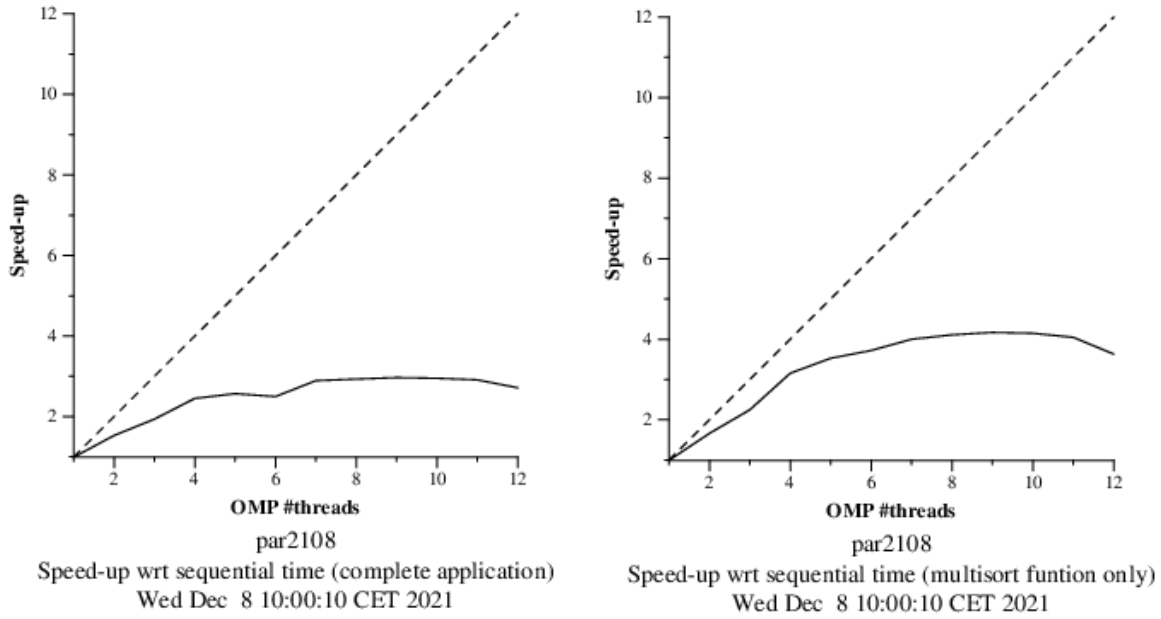


Figure 3.1.1 – Speed-up for leaf strategy

The result that we obtained with our implementation is not what we expected, to figured out what was happening we submitted the execution the binary using `submit-extrae.sh` script for 8 processors and open the generated trace with *Paraver* to understand what was going on.

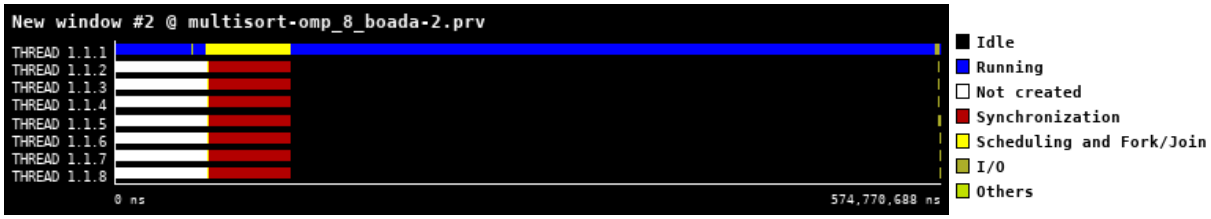


Figure 3.1.2 – *Paraver* simulation with 8 threads for leaf strategy

As we can see at the image above, the program does not generate enough tasks to simultaneously feed all processors, the first processor is executing a big part of the program that is not parallelized and distributed with the other processors.

3.2. Tree strategy in *OpenMP*

The code modifications that we made.

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

Observe that for the Tree strategy we decide to parallelize the recursive cases, in function *merge* the recursive calls to *merge* functions and in function *multisort* the calls to itself and also the *merge* functions. To avoid data races, we maintain the *taskwaits* from the leaf strategy version and add a new one's at the end of recursive decompositions of each function.

After that, we executed the code with `submit-strong-omp.sh` script, and we obtain the following results.

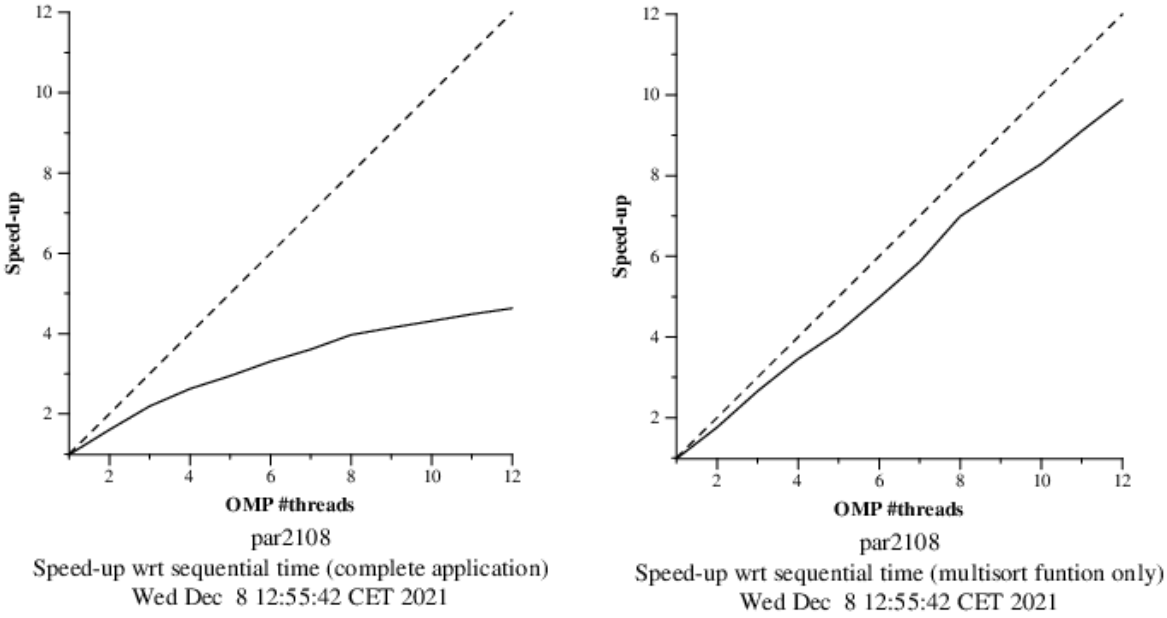


Figure 3.2 – Speed-up for tree strategy

This implementation obtained a better result as we can see at the figure above, the speed-up improves as the number of threads increases. That's the result of a fine-grained number of tasks that is distributed along all the threads and split the work in a more equal way.

3.3. Task granularity control: the cut-off mechanism

The code modifications that we made.

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

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

```
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
            #pragma omp task
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

            #pragma omp taskwait
            #pragma omp task
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
            #pragma omp taskwait
        } else {
            multisort(n/4L, &data[0], &tmp[0], depth+1);
            multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
...
int main(int argc, char **argv) {
    ...
    START_COUNT_TIME;
    #pragma omp parallel
    #pragma omp single
    multisort(N, data, tmp);
    ...
}
```

The changes that we made code are just add another parameter to the functions *merge* and *multisort* and check if this parameter is smaller than the cut-off, if is smaller then implement the tree strategy that we made is previous section otherwise execute the original code.

We generated a trace to see the changes when we set the cut-off to 0 and to 1.

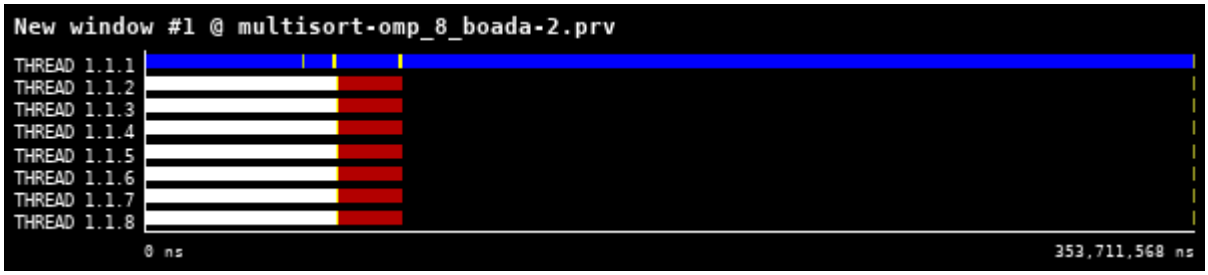


Figure 3.3.1 – Multisort with the cutoff = 0

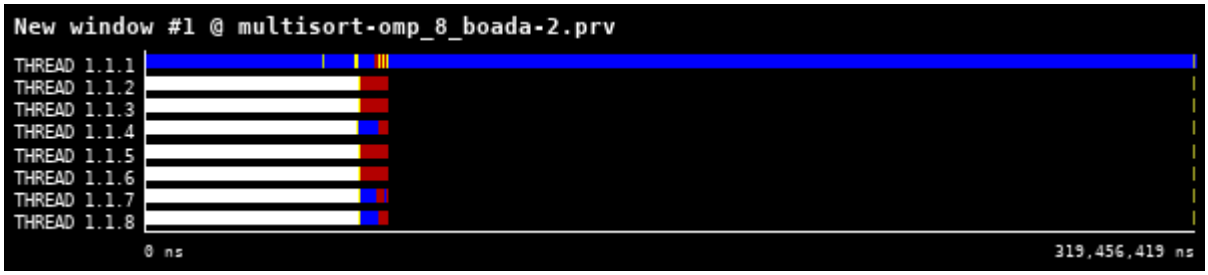


Figure 3.3.2 – Multisort with the cutoff = 1

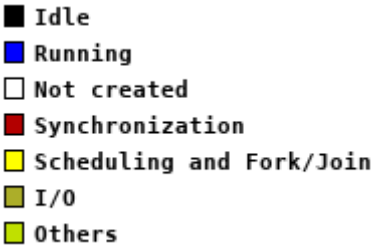


Figure 3.3.3 – Mutlisort legend for both executions (cut-off = 0 and cut-off = 1)

As we can see, if we set the cut-off to 0, the code is running sequentially, but if we set the cut off to 1 the other threads are running part of the code.

After understanding the cut-off mechanism, we explored the cut-off level depending on the number of processors used, we tried different number of processors. Finally for the next step, we decided to use 8 processors.

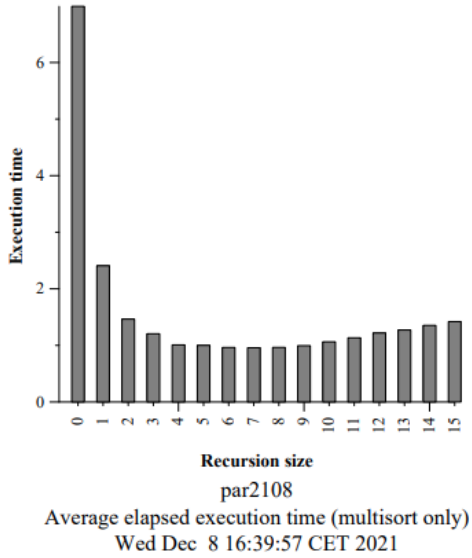


Figure 3.3.4 – submit-cutoff-omp.sh with 8 processors

From the previous figure we can conclude that the optimal value for *the* cut-off is 7. Now we edit the script `submit-strong-omp.sh` values *sort_size* and *merge_size* to 128 and compared the plots generated when we set the cut-off at 16 and at 7.

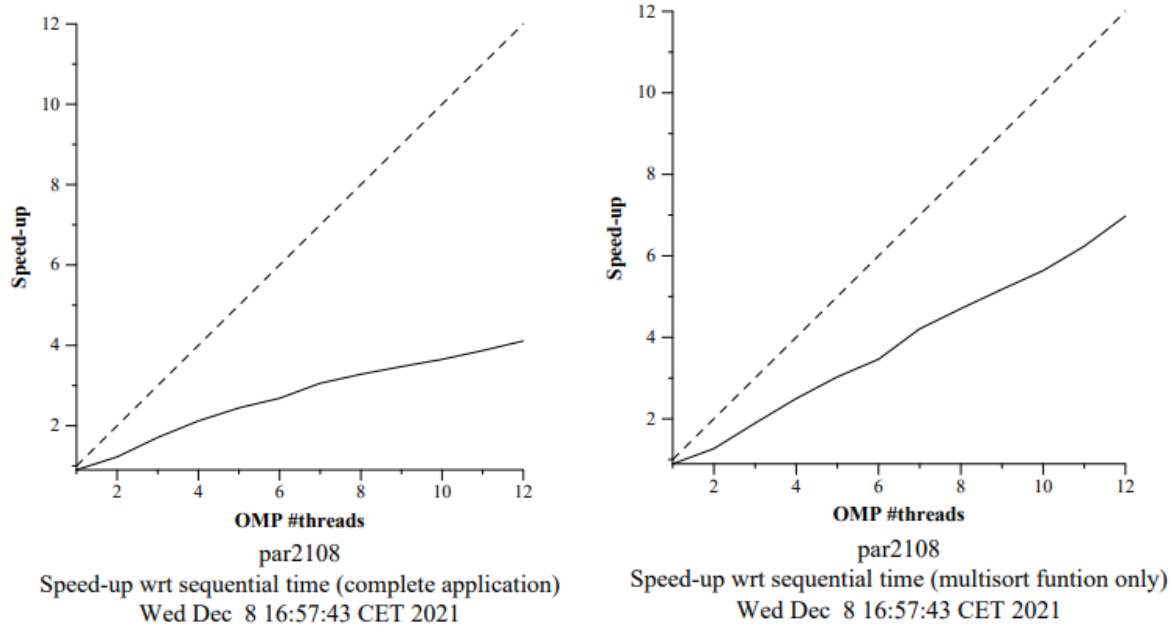


Figure 3.3.4 – Speed-up with the cut-off set at 16 (original)

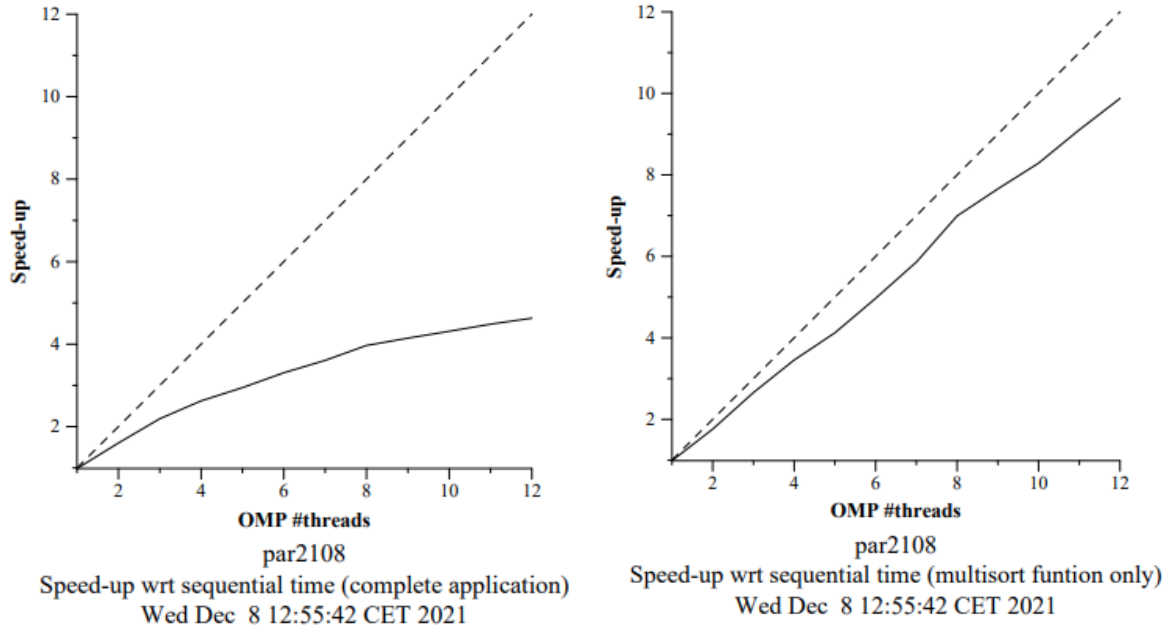


Figure 3.3.5 – Speed-up with the cut-off set at 7 (optimum)

Notice the improve of speed-up. When we change the value of the cut-off, we can control the number of tasks that our code generates when we use recursion, that allows us to distribute even better the work of the execution that is running between the threads.

4. Using OpenMP task dependencies

In this last section of the laboratory assignment, we will change the tree parallelisation in the previous chapter in order to express dependencies among tasks and avoid some of the taskwait/taskgroup synchronisations that we introduced in order to enforce task dependences.

The code modifications that we made.

…

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

            #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, depth+1);
            #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, depth+1);

            #pragma omp task depend(in: tmp[0], tmp[n/2L])
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);

            #pragma omp taskwait

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

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

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

…

The multisorts depend on the piece of vector that we send to order, while the merges depend on the output of the corresponding multisorts. We add output dependencies in the recursive calls of the multisort and merge functions (except the last merge) and input dependencies in the merge functions.

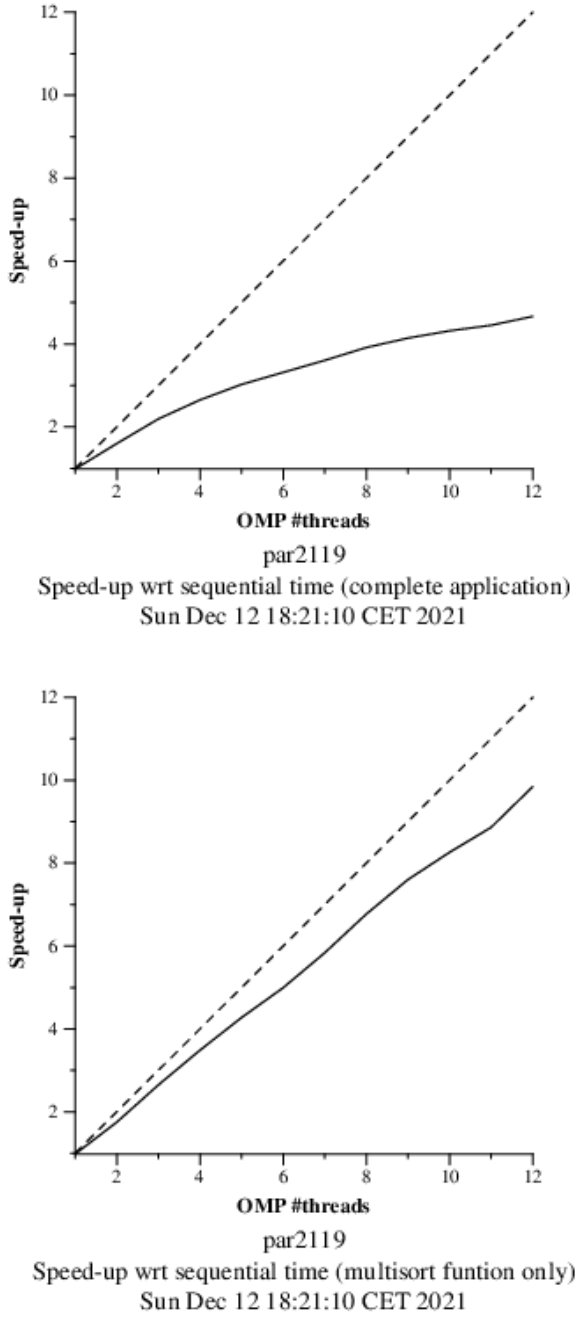


Figure 4.1 – Speed-up for tree strategy with task dependencies

If we compare the speedup graphs of the tree strategy with task dependencies with the tree strategy with optimum cut-off (figure 3.3.5) we can see that they are quite similar. In terms of programmability, the previous version seems simpler to code, although the version with dependencies is more understandable.

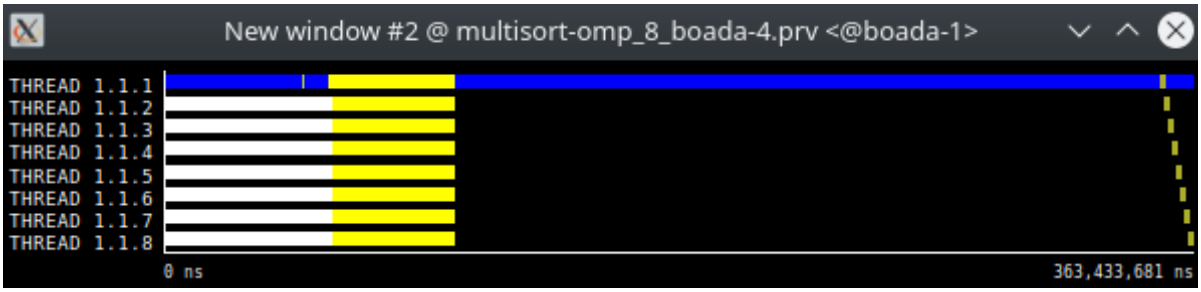


Figure 4.2 – *Paraver* simulation for tree strategy with task dependencies (8 threads)

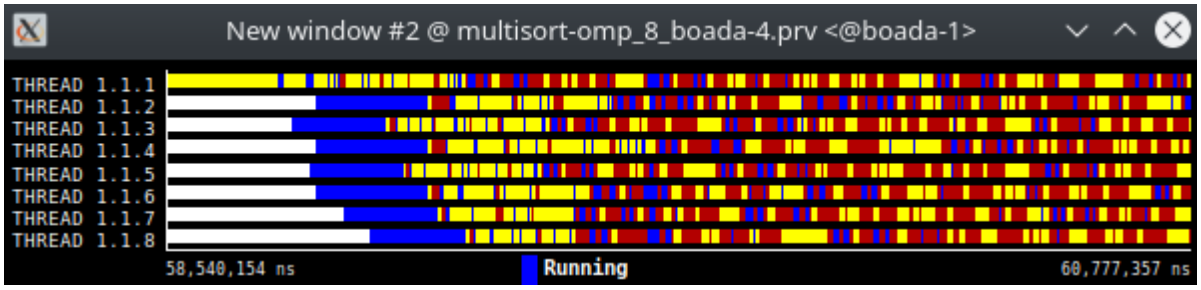


Figure 4.3 – Enlarged *Paraver* simulation of figure 4.2

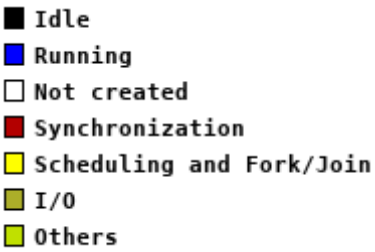


Figure 4.4 – Legend of the *Paraver* simulation of the figures 4.2 and 4.3

If we compare the *Paraver* simulation of the program execution of this version (Figure 4.2) and the previous one (Figure 3.3.2) we can see that in this execution the threads spend more time waiting for the dependencies than in the previous version, where it spends more time running.

5. Optionals

5.1. Optional 1

To explore the scalability of our tree implementation with cut-off when using 24 threads we need to edit the file submit-strong-omp.sh to set the variable np NMAX to 24.

After submitting the strong scalability script, we obtain the following plots (Figure 5.1.1).

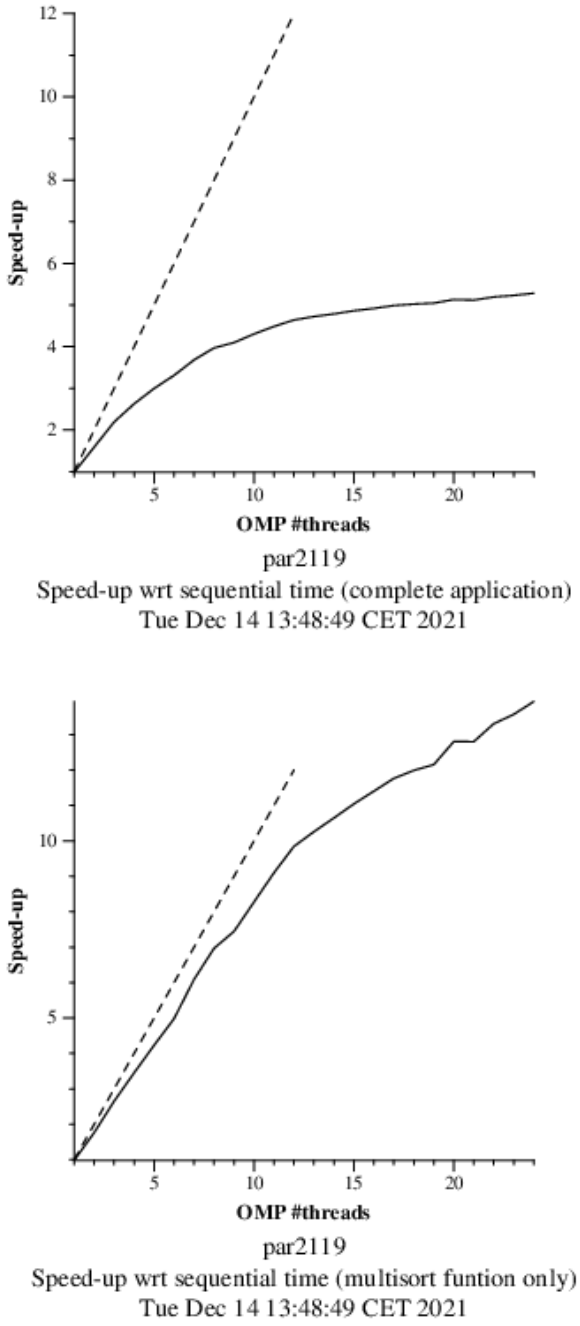


Figure 5.1.1 – Strong scalability plot with cutoff and max_threads=24

We can see that the speedup still growing even though Boada only offers 12 physical threads. This happens because, if we remember the architecture of boada that we explored in the first laboratory (Figure 5.1.2 and Figure 5.1.3), in each node of boada there are 2 sockets and each socket has 6 cores with 2 threads, so the total number of processors is 24.

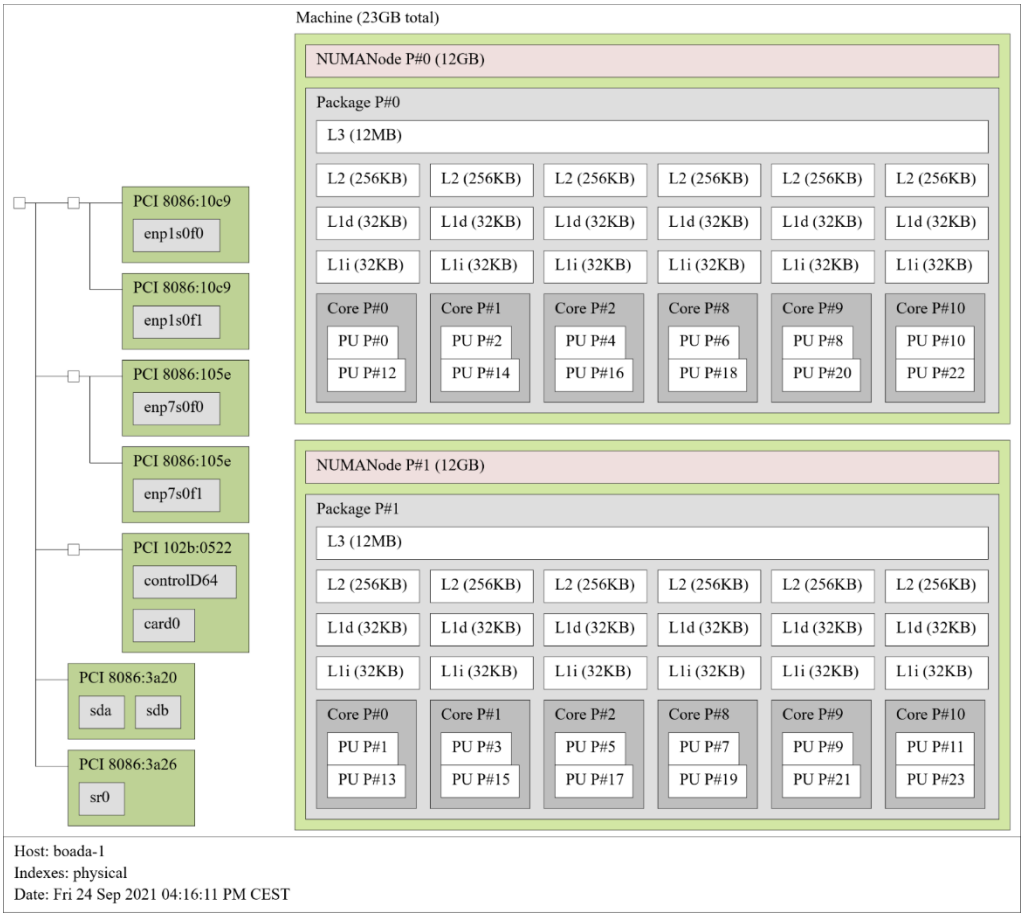


Figure 5.1.2 – Architectural diagram

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

Figure 5.1.3 – Information about the hardware of Boada

5.2. Optional 2

In this section we have to parallelize the two functions that initialise the data and tmp vectors.

The code modifications that we made.

…

```
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;
    }
}
```

...

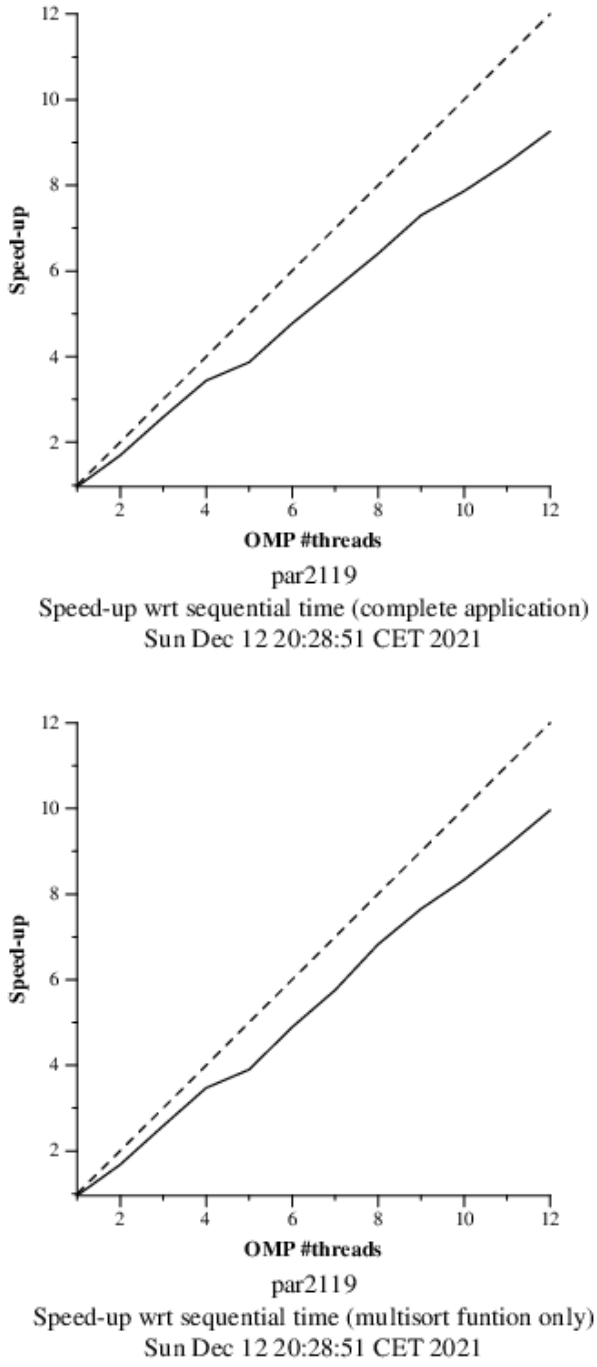


Figure 5.2.1 – Strong scalability plot with parallelized initialization

We can see that the graphs of complete application and multisort function only are very similar. This is because if we remove the multisort function, the biggest part of the program is the initialization of the vectors.

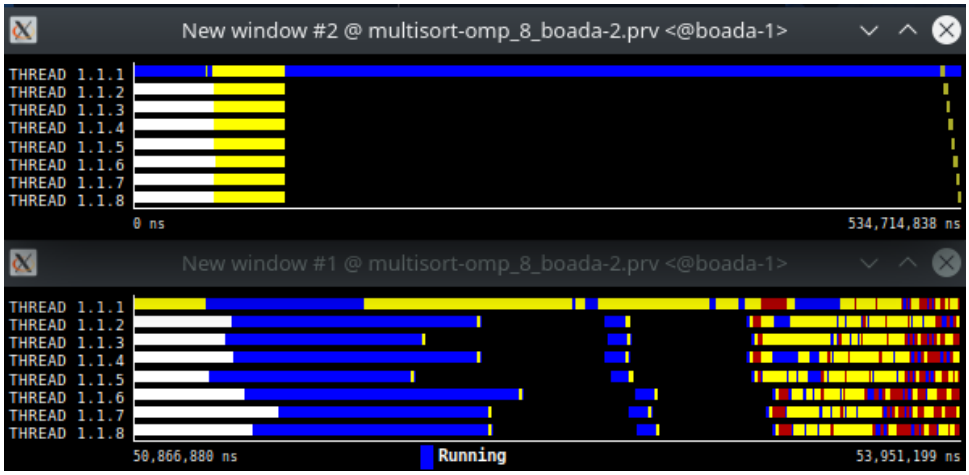


Figure 5.2.2 – Full execution (above) and extended (below)

In the *Paraver* simulation we can see how the threads are in charge of initializing the data vector first and then the tmp vector, and then start with the parallelization of the multisort function.

6. Conclusions

To summarize, we have seen how the leaf and tree strategy works, how to implement different OpenMP commands to achieve the recursive strategies and how these strategies combined with different OpenMP structures influence the performance of our programs. Also, we get a look at the cut-off mechanism that allowed us take control of the task granularity.

The conclusions that we extract of this session is that for the divide and conquer algorithm the best strategy to follow is the tree one, we obtain a better speed-up. We also have experimented different cut-off values to obtain the best speed-up for a specific number of threads, in our case, for 8 threads, we obtained that the best cut-off value was 7. Finally, for the tree strategy, that for the first part we had implemented with OpenMP task (taskwaits), for this part we implemented with task dependencies, and we analysed the performance change.