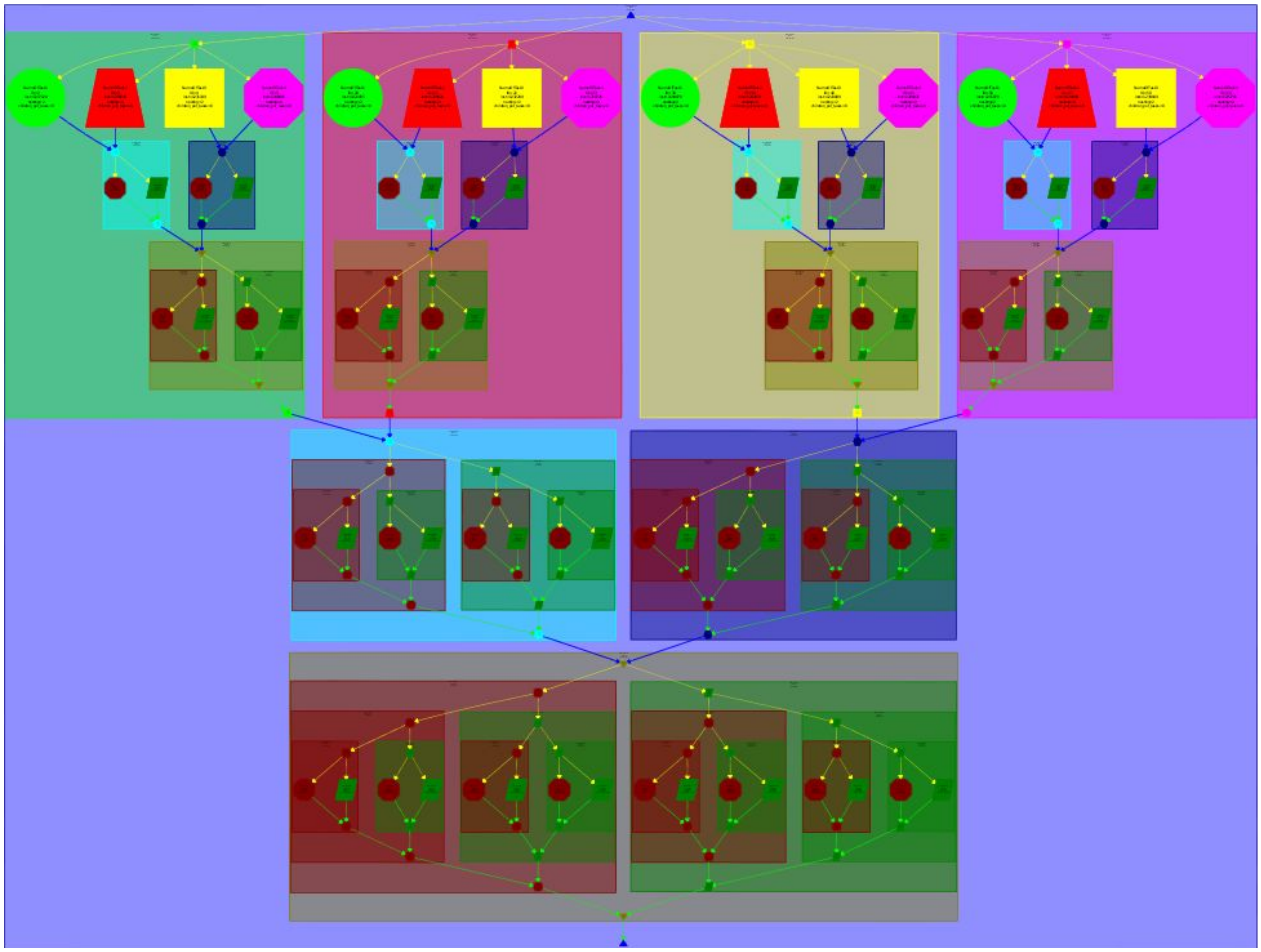
PAR Laboratory Assignment
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

PAR2102

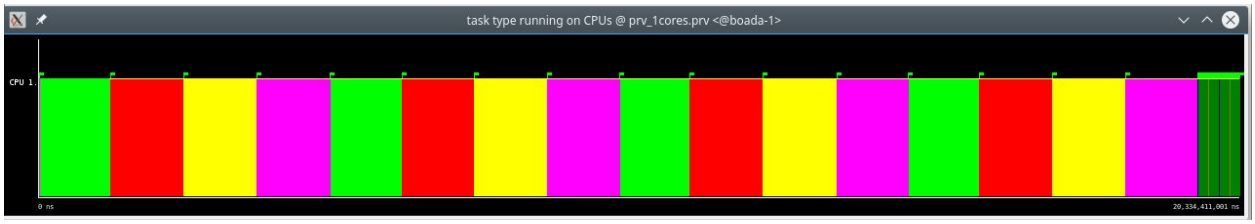
Alejandro Alarcón Torres
Guillem Reig Gaset

Session 1

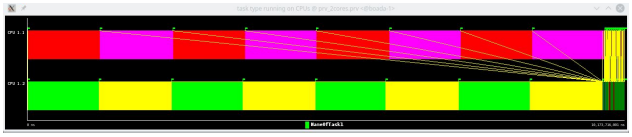


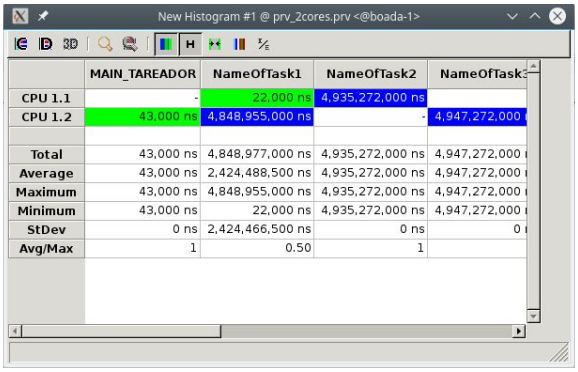
Dependency graph.

There are two main functions that allow us to run this program in parallel mode: “multisort” and “merge”. They are both recursive functions, so in our case, we decided to create a task for the execution of every recursive call.
Regarding the dependency graph, we can observe that the children tasks do not have any dependencies between them. Therefore, we have implemented a tree strategy as we think it is better optimised for this kind of algorithm. A leaf strategy could also be implemented. We also need to synchronize the “multisort” functions before executing the “merge” ones.



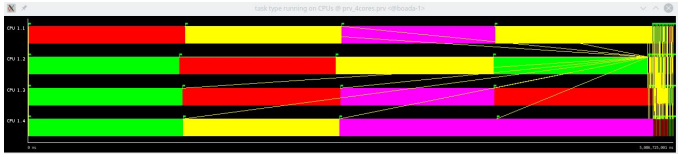
1 thread simulation.

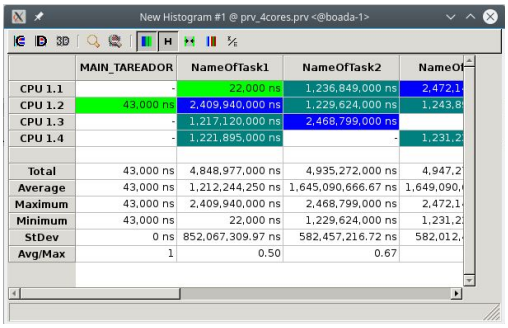




	MAIN_TAREADOR	NameOfTask1	NameOfTask2	NameOfTask3
CPU 1.1	-	22,000 ns	4,935,272,000 ns	
CPU 1.2	43,000 ns	4,848,955,000 ns	-	4,947,272,000
Total	43,000 ns	4,848,977,000 ns	4,935,272,000 ns	4,947,272,000
Average	43,000 ns	2,424,488,500 ns	4,935,272,000 ns	4,947,272,000
Maximum	43,000 ns	4,848,955,000 ns	4,935,272,000 ns	4,947,272,000
Minimum	43,000 ns	22,000 ns	4,935,272,000 ns	4,947,272,000
StDev	0 ns	2,424,466,500 ns	0 ns	0
Avg/Max	1	0.50	1	

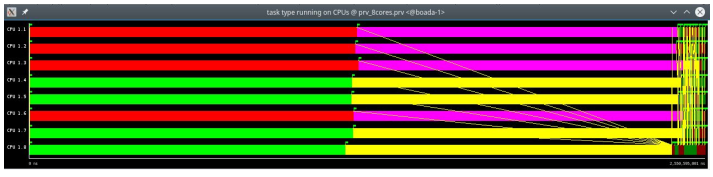
2 threads simulation.

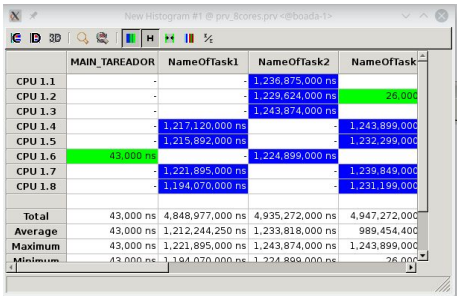




	MAIN_TAREADOR	NameOfTask1	NameOfTask2	NameOf
CPU 1.1	-	22,000 ns	1,236,849,000 ns	2,472,1
CPU 1.2	43,000 ns	2,409,940,000 ns	1,229,624,000 ns	1,243,8
CPU 1.3	-	1,217,120,000 ns	2,468,799,000 ns	
CPU 1.4	-	1,221,895,000 ns	-	1,231,2
Total	43,000 ns	4,848,977,000 ns	4,935,272,000 ns	4,947,2
Average	43,000 ns	1,212,244,250 ns	1,645,090,666.67 ns	1,649,090,(
Maximum	43,000 ns	2,409,940,000 ns	2,468,799,000 ns	2,472,1
Minimum	43,000 ns	22,000 ns	1,229,624,000 ns	1,231,2
StDev	0 ns	852,067,309.97 ns	582,457,216.72 ns	582,012,(
Avg/Max	1	0.50	0.67	

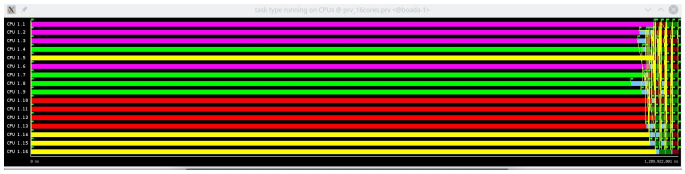
4 threads simulation.

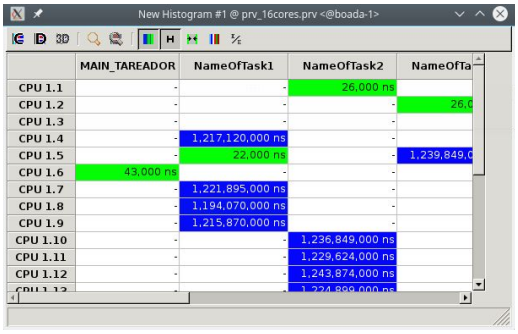




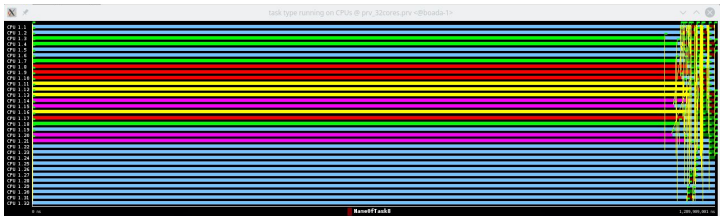
	MAIN_TAREADOR	NameOfTask1	NameOfTask2	NameOfTask
CPU 1.1	-	-	1,236,875,000 ns	
CPU 1.2	-	-	1,229,624,000 ns	26,000
CPU 1.3	-	-	1,243,874,000 ns	
CPU 1.4	-	1,217,120,000 ns	-	1,243,899,000
CPU 1.5	-	1,215,892,000 ns	-	1,232,299,000
CPU 1.6	43,000 ns	-	1,224,899,000 ns	
CPU 1.7	-	1,221,895,000 ns	-	1,239,849,000
CPU 1.8	-	1,194,070,000 ns	-	1,231,199,000
Total	43,000 ns	4,848,977,000 ns	4,935,272,000 ns	4,947,272,000
Average	43,000 ns	1,212,244,250 ns	1,233,818,000 ns	989,454,400
Maximum	43,000 ns	1,221,895,000 ns	1,243,874,000 ns	1,243,899,000
Minimum	43,000 ns	1,194,070,000 ns	1,224,899,000 ns	26,000

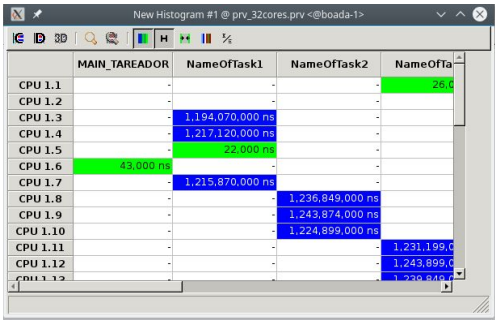
8 threads simulation.



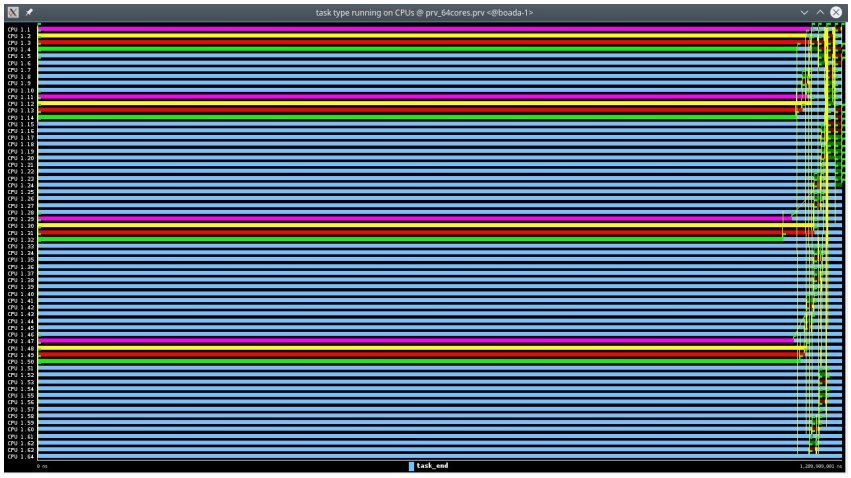


16 threads simulation.



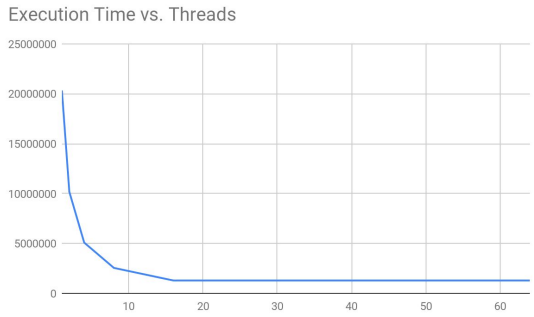


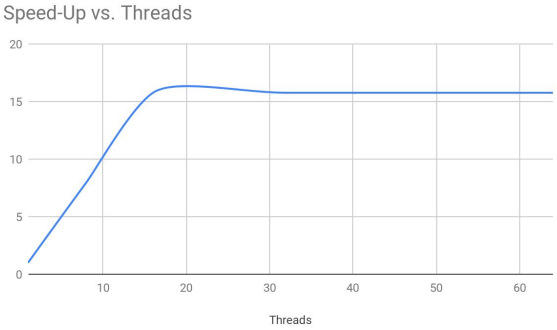
32 threads simulation.



64 threads simulation.

Threads	Execution Time	Speed-up
1	20334421	1
2	10173720	1.999
4	5086725	3.997
8	2550595	7.97
16	1289909	15.76
32	1289909	15.76
64	1289909	15.76





If we take a look at these plots, we can observe an improvement in the parallel execution time until we reach the 16 threads. After that, the execution time remains constant up until the 64 threads, as far as we have tested (we know that is the maximum level of parallelism given our modifications to the code, as the value of Pmin is 16).

```
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
                tareador_start_task("NameOfTask8");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("NameOfTask8");

                tareador_start_task("NameOfTask9");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("NameOfTask9");
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("NameOfTask1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("NameOfTask1");

                tareador_start_task("NameOfTask2");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("NameOfTask2");

                tareador_start_task("NameOfTask3");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("NameOfTask3");

                tareador_start_task("NameOfTask4");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("NameOfTask4");

                tareador_start_task("NameOfTask5");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("NameOfTask5");

                tareador_start_task("NameOfTask6");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("NameOfTask6");

                tareador_start_task("NameOfTask7");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("NameOfTask7");
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Code for multisort and merge.

Session 2

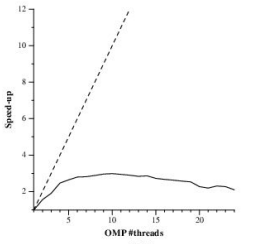
Leaf decomposition:

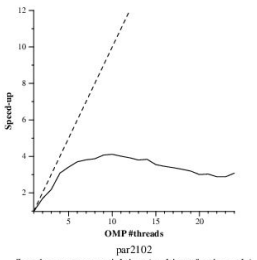
In this decomposition we only need to create tasks on the leaves of the recursion tree, so we need to create the tasks right before the calls of “basicmerge” and “basicsort”. As mentioned before, the children tasks do not have any dependencies between them, so we can create the tasks without using any synchronization mechanism or data sharing clause.
In this strategy we have one thread per leaf, which means that we are bound by the number of leaves, decreasing the parallelism.

```
// N and MIN must be powers of 2
long N;
long MIN_SORT_SIZE;
long MIN_MERGE_SIZE;
int CUTOFF;

#define BLOCK_SIZE 1024L

#define T int

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) {
        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);
        }
}
```

Code for merge and multisort.

Paraver Leaf.





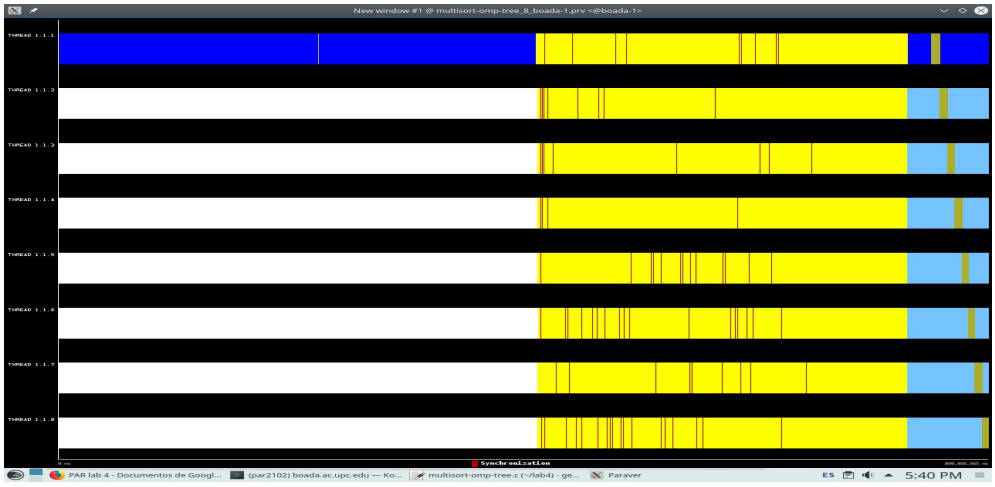
Leaf strong.

Tree decomposition:

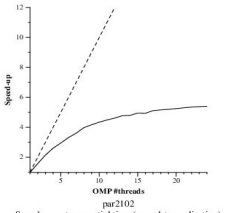
In this decomposition we need to create tasks at each call except the ones that are on the leaves of the tree. In this case we need to create a task before each “multisort” call and each “merge” call. Additionally, we need to satisfy the dependencies between the tasks, which are defined by the data accesses, as seen in the Tareador analysis.
Using this strategy we have one thread for each function, which means that we have better parallelism than with the leaf decomposition, as we are not bound by the number of leaves.

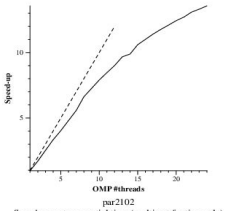
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition

                #pragma omp task
                merge(n, left, right, result, start, length/2);

                #pragma omp task
                merge(n, left, right, result, start + length/2, length/2);
        }
}

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

                #pragma omp task
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);

                #pragma omp task
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);

                #pragma omp task
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                }

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

Tree decomposition code.



Paraver tree decomposition.





Tree strong.

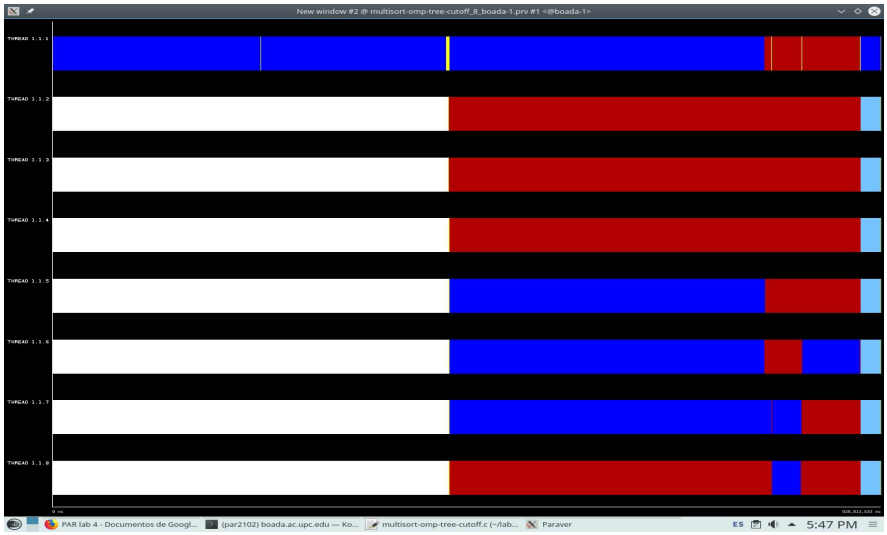
Cut-off mechanism:

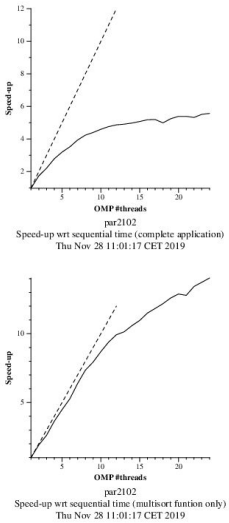
To implement the cut-off mechanism for the tree decomposition we need to use the final clause, the “omp_in_final()” function and the global variable “CUTOFF”.
We have the same case as in the tree decomposition strategy, but this time we limit the number of threads to the CUTOFF that we define, meaning that we can have a good parallelism while limiting the overheads generated. To check that the function is correct we can use a CUTOFF = 0, as we can see the execution time is larger than in the tree decomposition.

```
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 (!omp_in_final()) {
                        #pragma omp task final (depth >= CUTOFF)
                        merge(n, left, right, result, start, length/2,depth+1);

                        #pragma omp task final (depth >= CUTOFF)
                        merge(n, left, right, result, start + length/2, length/2,depth+1);
                }
                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 (!omp_in_final()) {
                        #pragma omp taskgroup
                        {
                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[0], &tmp[0],depth+1);

                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[n/4L], &tmp[n/4L],depth+1);

                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[n/2L], &tmp[n/2L],depth+1);

                        #pragma omp task final (depth >= CUTOFF)
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],depth+1);
                        }

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

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

Tree cut-off code.





Paraver tree cut-off.

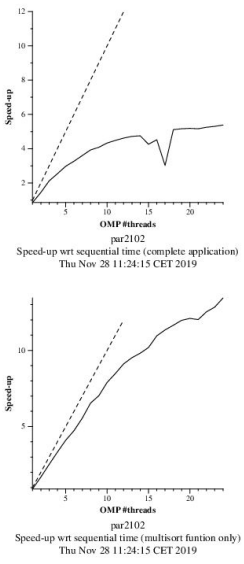
Tree-cutoff strong.

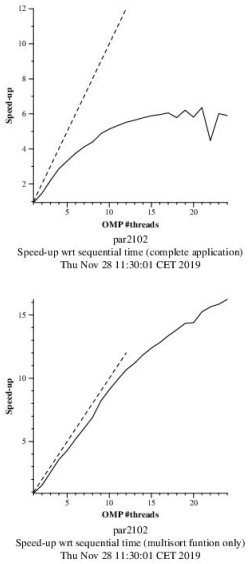
Optional 1:

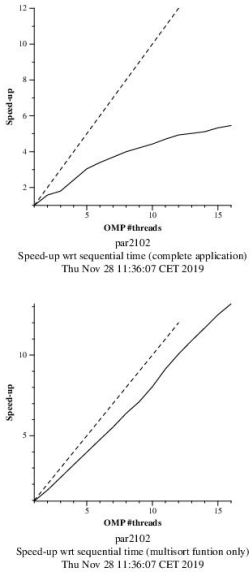
Boada 1 - 5 uses np_NMAX = 24, as it has 2 sockets with 6 cores per socket and 2 threads per core, which adds up to 24 logical cores.

Boada 6 - 8 uses np_NMAX = 16, as it has 2 sockets with 8 cores per socket but just one thread per core, which means that it has a total of 16 logical cores.

As we can see in the plots, the boada 7 has a smoother speed-up than boada-3 and boada-5, but at the beginning both boada-3 and boada-5 have a slightly better speed-up.







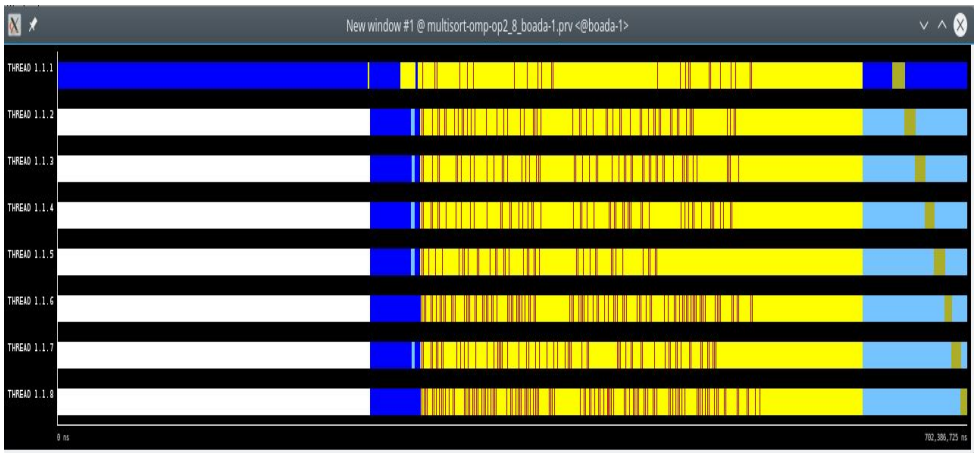
Boada 3 (opcional 1).

Boada 5 (opcional 1).

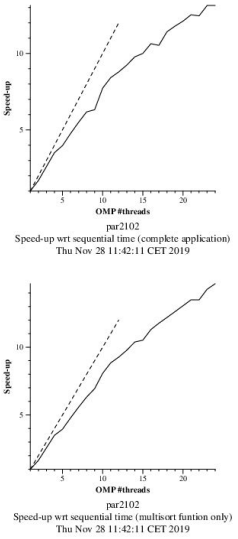
Boada 7 (opcional 1).

Optional 2:

To parallelize the “initialize” and “clear” functions we can use the “for” clause with “static” scheduling, because every iteration has the same workload (the exception is the iteration 0 of the “initialize” loop). As we can see, we have a better speed-up, because the execution of the “initialize” and “clear” functions is faster than before.



Paraver opcional 2 submit-strong.



Strong opcional 2.

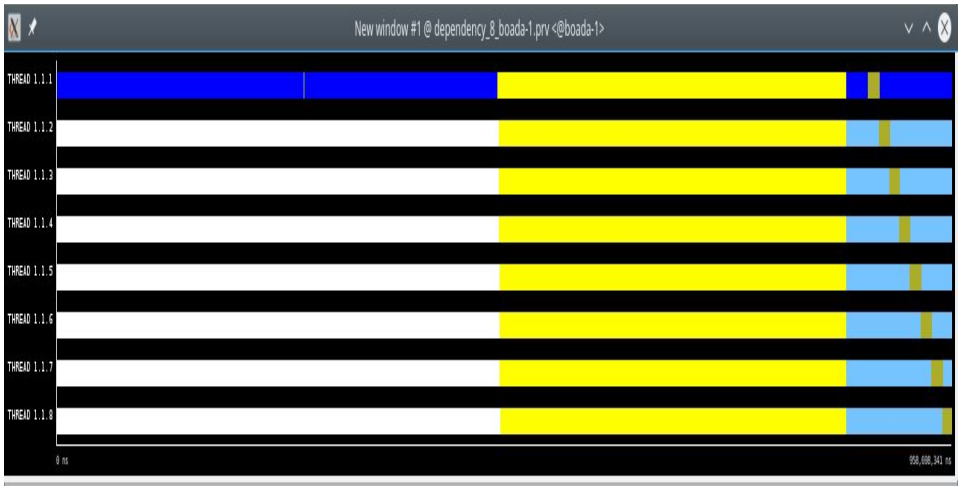
```
static void initialize(long length, T data[length]) {
   long i;
   #pragma omp parallel for shared(data)
   for (i = 0; i < length; i++) {
      if (i==0) {
         data[i] = rand();
      } else {
         data[i] = ((data[i-1]+1) * i * 104723L) % N;
      }
   }
}

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

Code for initialize and clear functions.

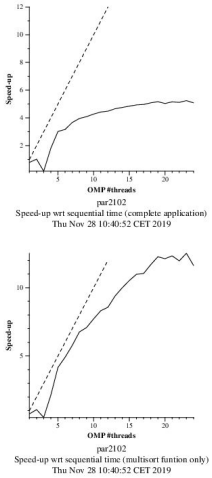
Session 3

Depend tasks:

In the previous tree decomposition we have a “taskgroup” for all the “multisort” functions and another one for the two first “merge” functions, which means that we need to execute all the “multisort” before the execution of “merge”, but as we can see in the Tareador analysis we can start the execution of the first “merge” once we’ve finished executing the first two “multisorts”. It can be achieved by using the “task depend” clause.



Paraver dependencies.



Strong dependencies.

```
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 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[3L*n/4L], data[n/2L]) 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
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Code for dependencies.