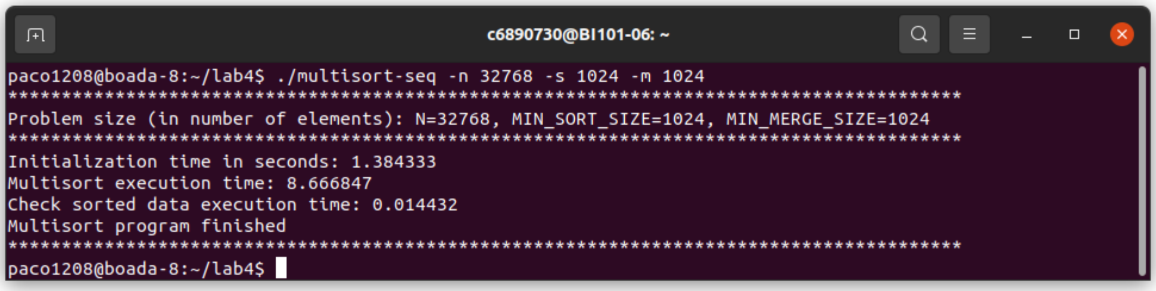
David Martos
Mariona Farré
Grupo: paco1208
Curso: 2022-23

PACO:
Laboratorio 4

Índice:

1. Before starting lab assignment
2. Task decomposition analysis for Mergesort
2.1. Divide and conquer

For the deliverable: Take note of the times reported for the sequential execution and use them as reference times to check the scalability of the parallel versions in OpenMP you will develop.



```
paco1208@boada-8:~/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: 1.384333
Multisort execution time: 8.666847
Check sorted data execution time: 0.014432
Multisort program finished
*****************************************************************************************
paco1208@boada-8:~/lab4$
```

2.2. Task decomposition analysis with Tareador

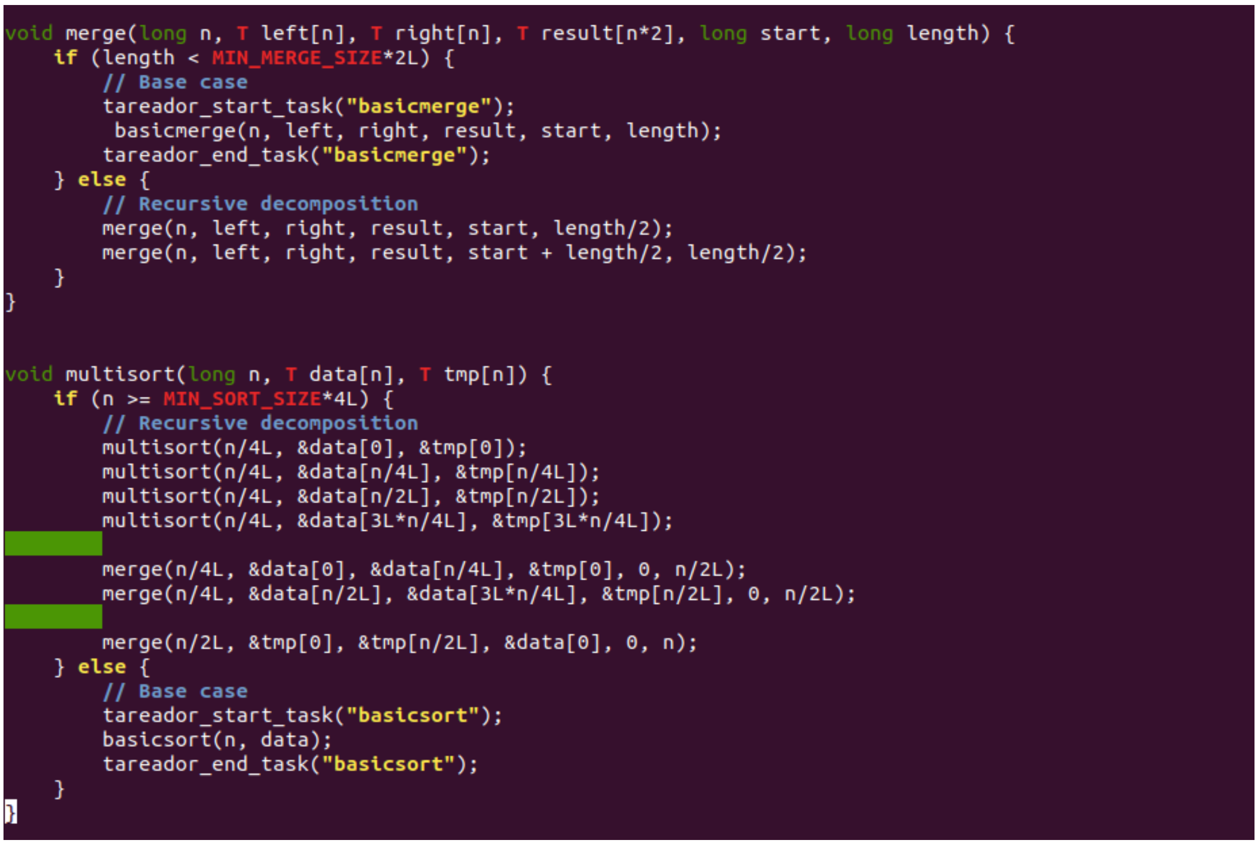
multisort-tareador.c is already prepared to insert Tareador instrumentation. Complete the instrumentation to understand the potential parallelism that each one of the two recursive task decomposition strategies (leaf and tree) provide when applied to the sort and merge phases:

- In the leaf strategy you should define a task for the invocations of basicsort and basicmerge once the recursive divide–and–conquer decomposition stops.
- In the tree strategy you should define tasks during the recursive decomposition, i.e. when invoking multisort and merge.

For the deliverable: Include both tareador codes (in the .zip file) and show code excepts in the report.

Leaf strategy:

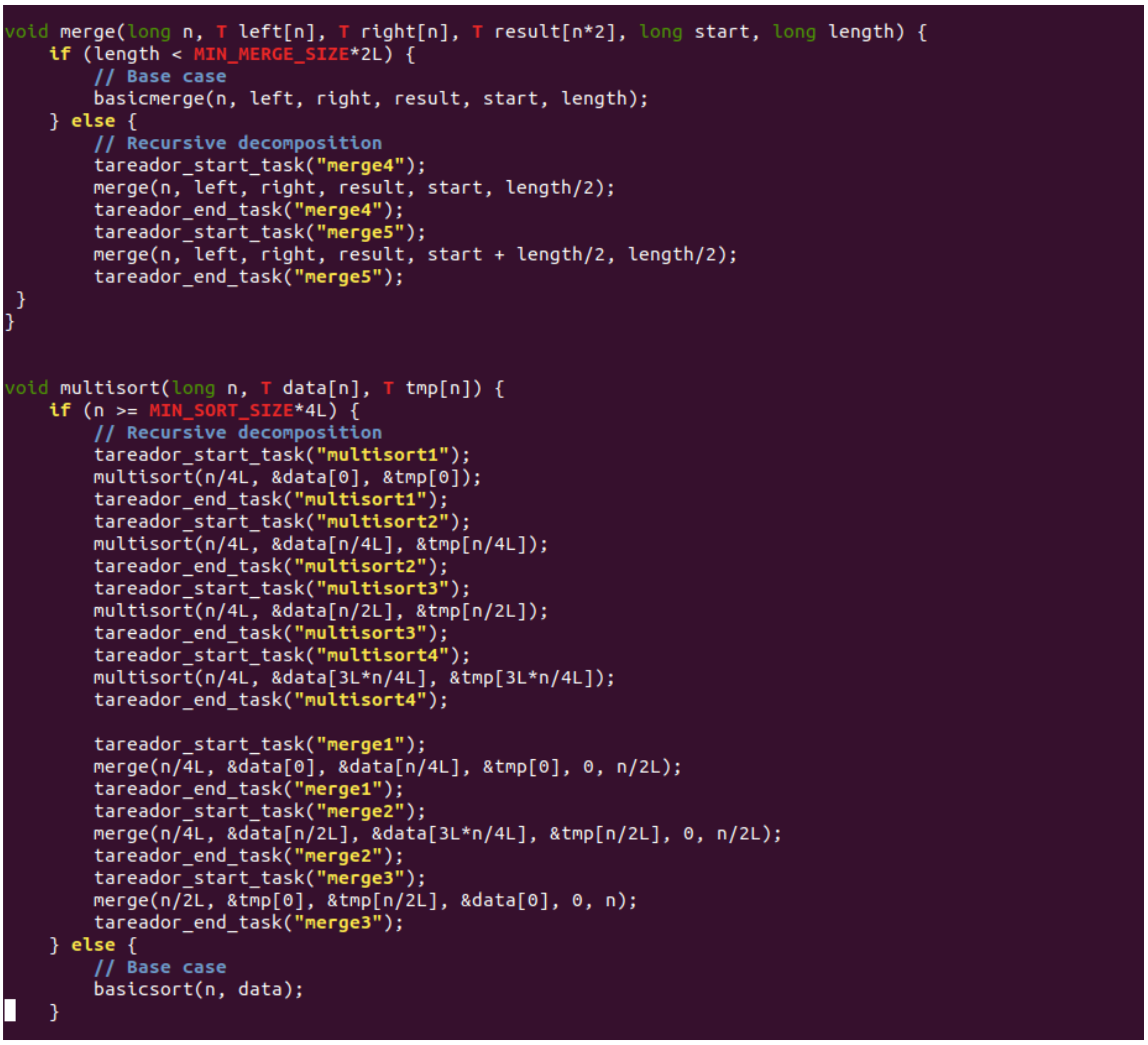
Definir las tareas del tareador en el basicmerge y el basicsort.



```
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 {
        // 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("basicsort");
        basicsort(n, data);
        tareador_end_task("basicsort");
    }
}
```

Tree strategy:

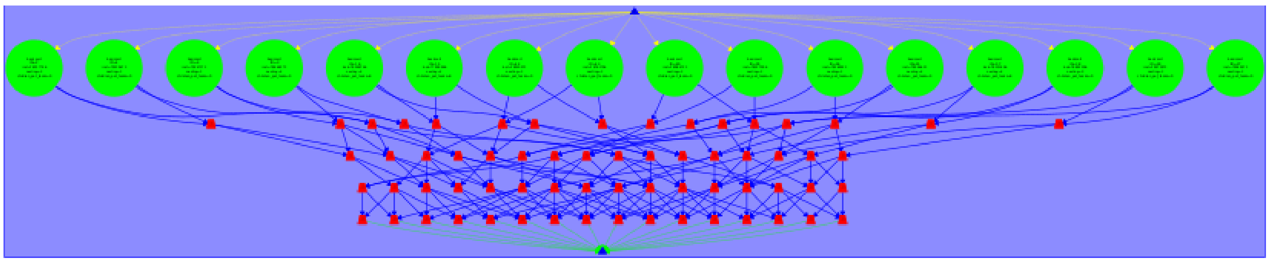
Definir las tareas del tareador en el merge y el multisort.



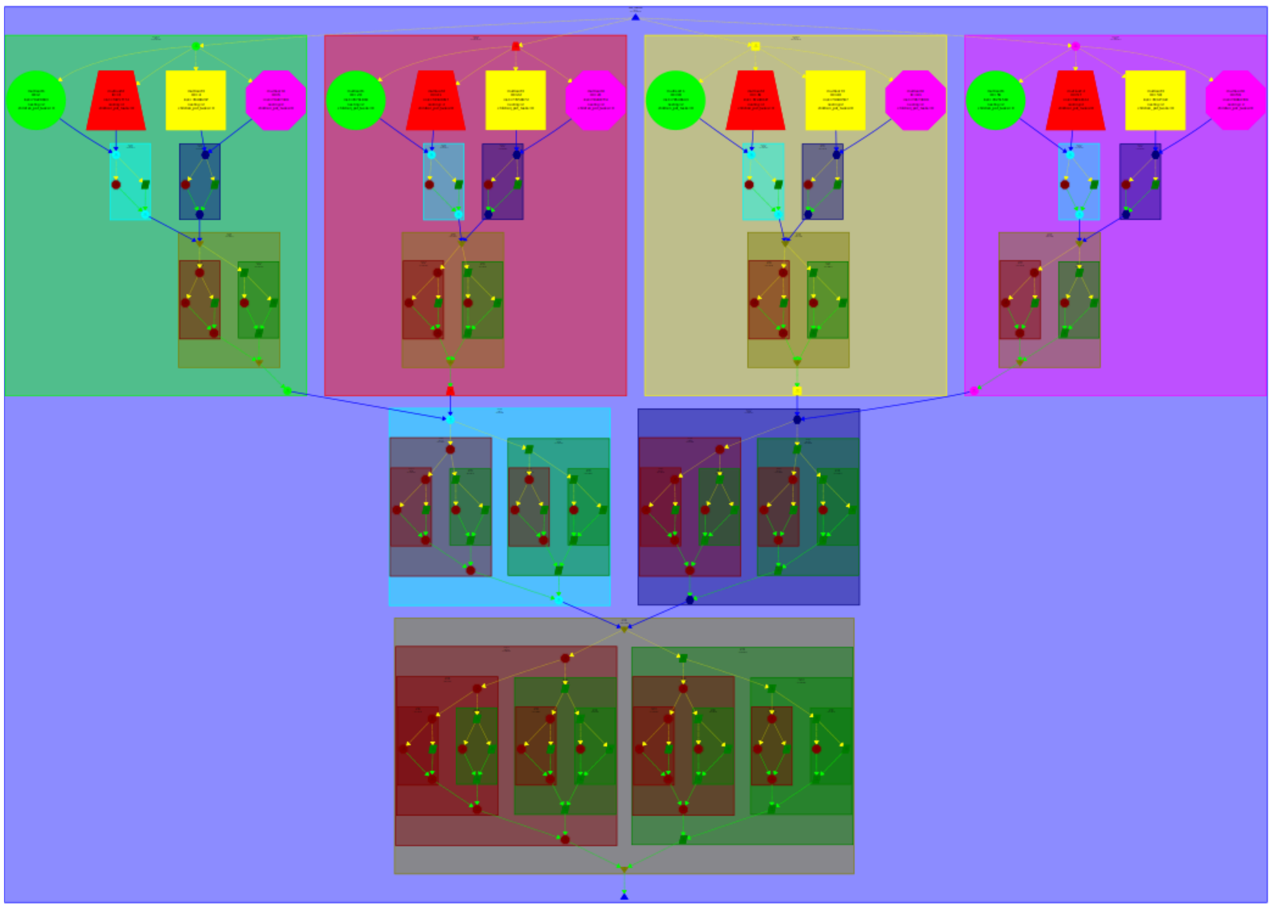
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        tareador_start_task("merge4");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("merge4");
        tareador_start_task("merge5");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("merge5");
 }
}


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("merge1");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("merge1");
        tareador_start_task("merge2");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("merge2");
        tareador_start_task("merge3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge3");
    } else {
        // Base case
        basicsort(n, data);
    }
```

For the deliverable: From the task dependence graphs that are generated for leaf and tree, do you observe any major differences in terms of structure, types, number and granularity of tasks, ...? Save the image of the two graphs in order to include them in the report.

Leaf strategy:



Tree strategy:



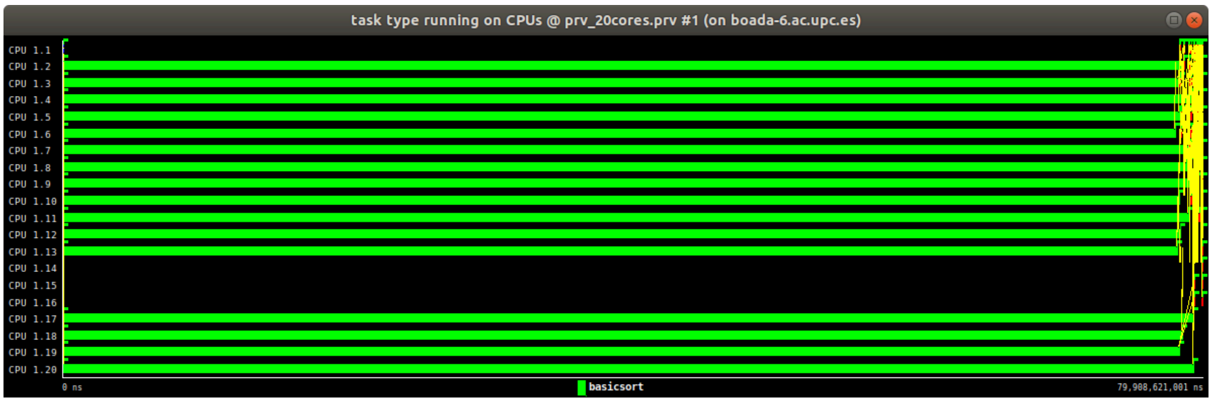
Como podemos observar los dos gráficos son muy diferentes para cada estrategia.
La estrategia del árbol tiene muchas mas tareas generadas porque hay muchas mas invocaciones para el basicsort y el mergesort, por lo contrario la estrategia de hoja genera solo tareas en el caso base de las funciones de merge y el multisort, no como el caso recursivo de la estrategia del árbol.

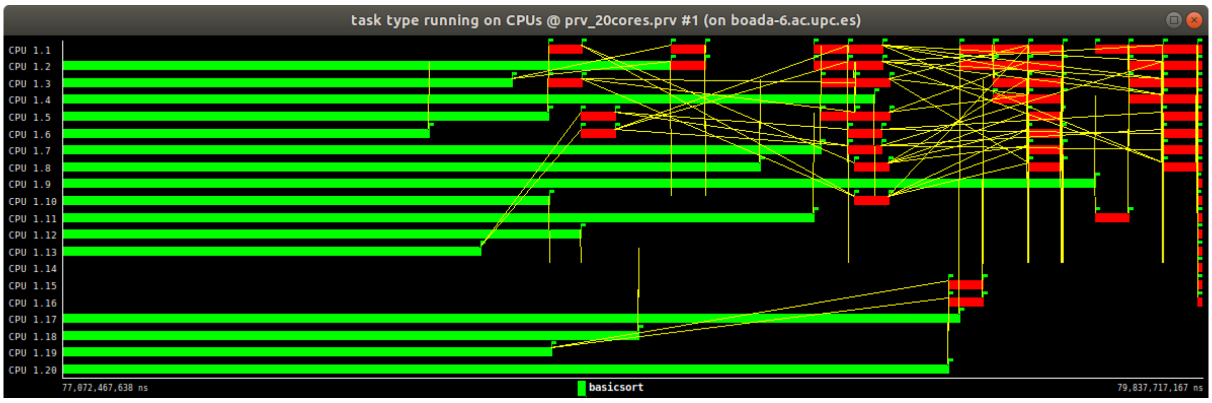
For the deliverable: Identify the points of the code where you should include synchronizations to guarantee the dependences

En las dos estrategias, la función merge crea dependencias porque es la función que ordena las dos partes del vector. original, mientras que en la función multisort como no depende de estas partes del vector no crea estas dependencias. Y podemos ver que la función multisort necesita mas tiempo porque hay muchas más tareas creadas.

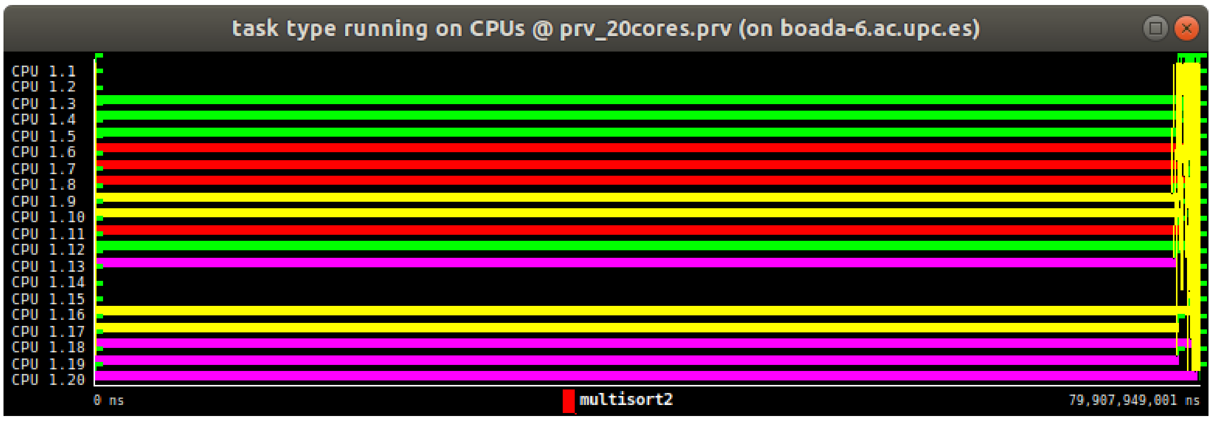
For the deliverable: Capture the necessary Paraver windows in order to support your explanations in the deliverable.

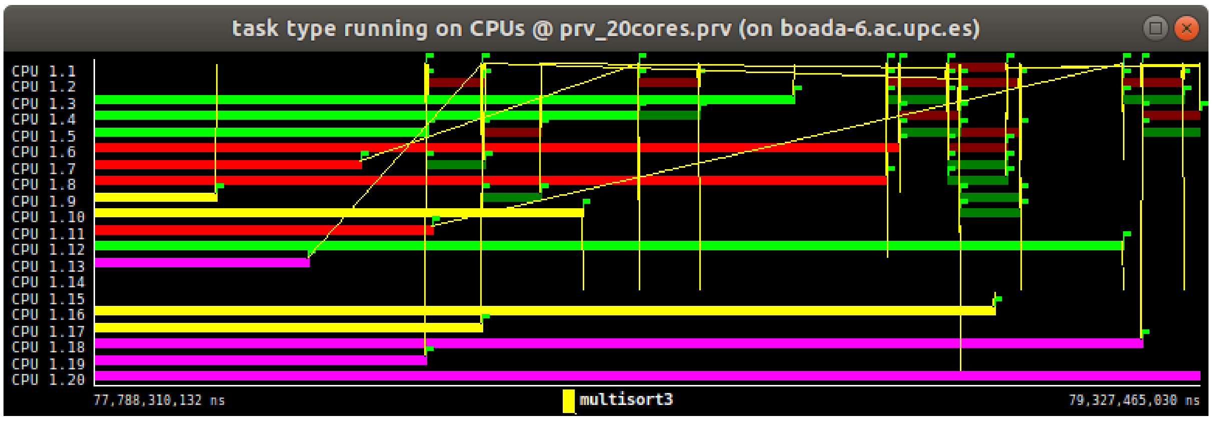
Leaf strategy:





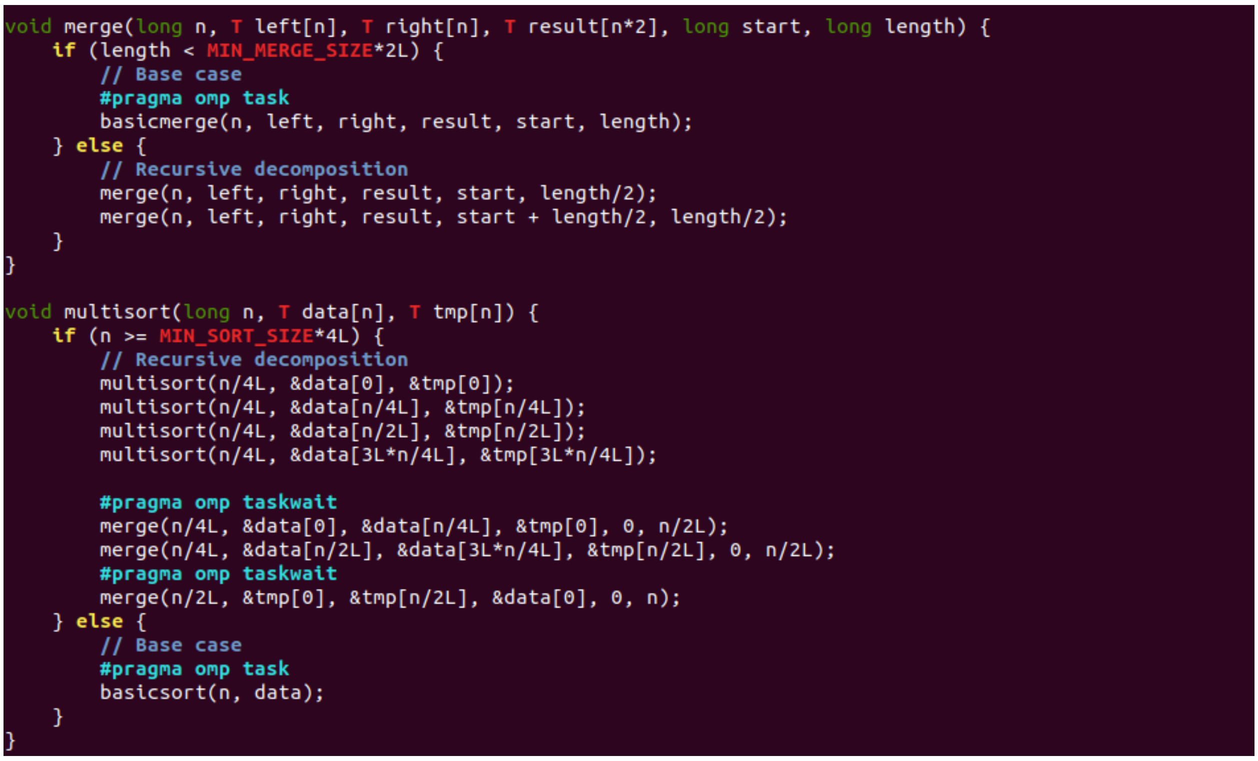
Tree strategy:



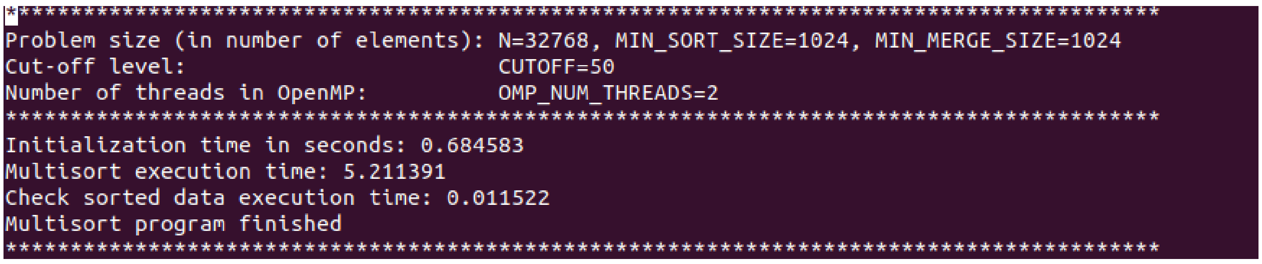


3. Shared-memory parallelisation with OpenMP tasks

3.1. Leaf strategy in OpenMP



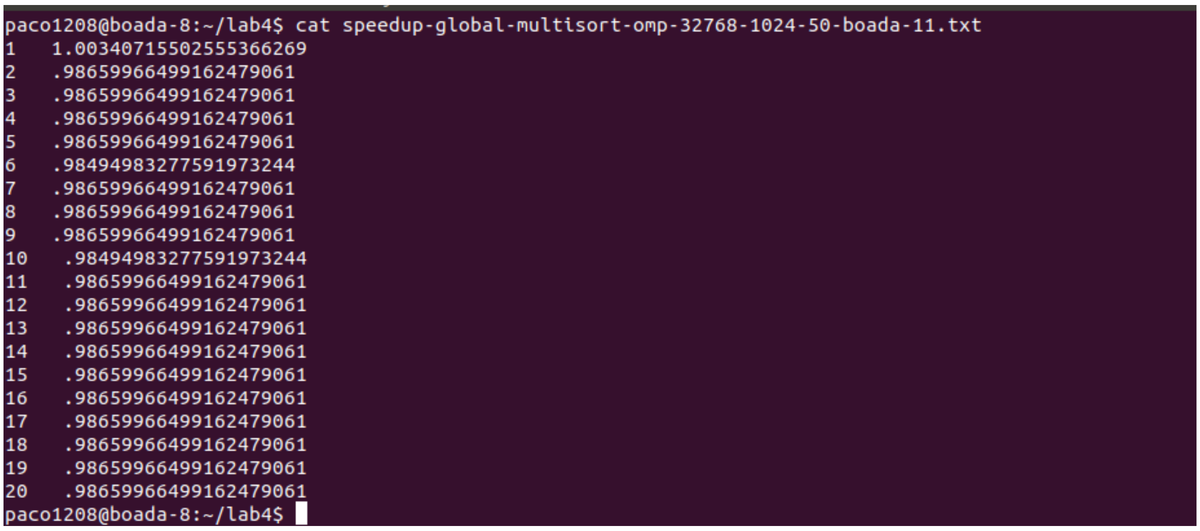
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        #pragma omp task
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        merge(n, left,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



```
*******************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
*******************************************************************************
Initialization time in seconds: 0.684583
Multisort execution time: 5.211391
Check sorted data execution time: 0.011522
Multisort program finished
*******************************************************************************
```

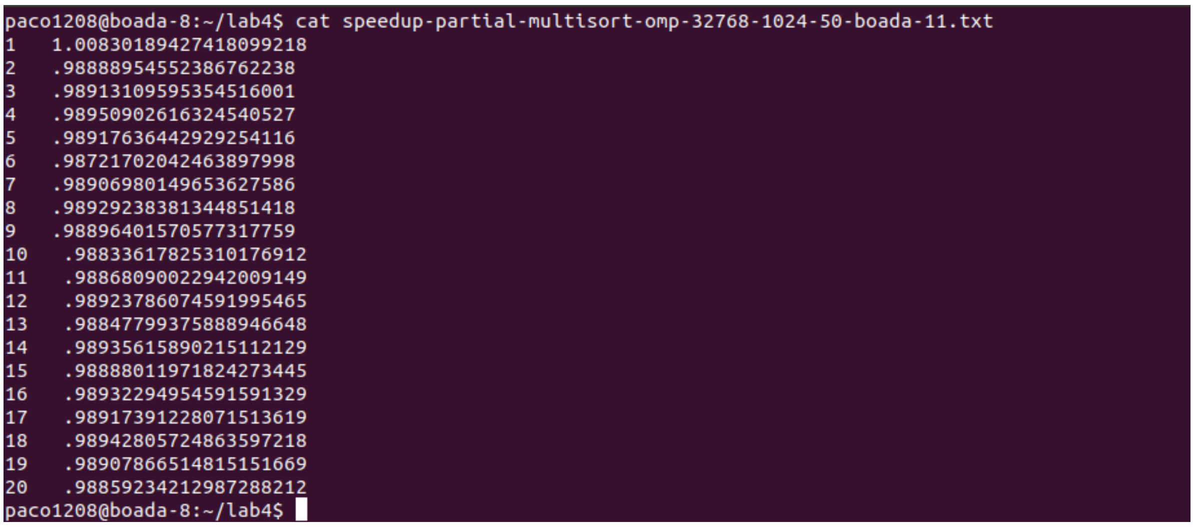
Text files:

<u>speedup-global:</u>

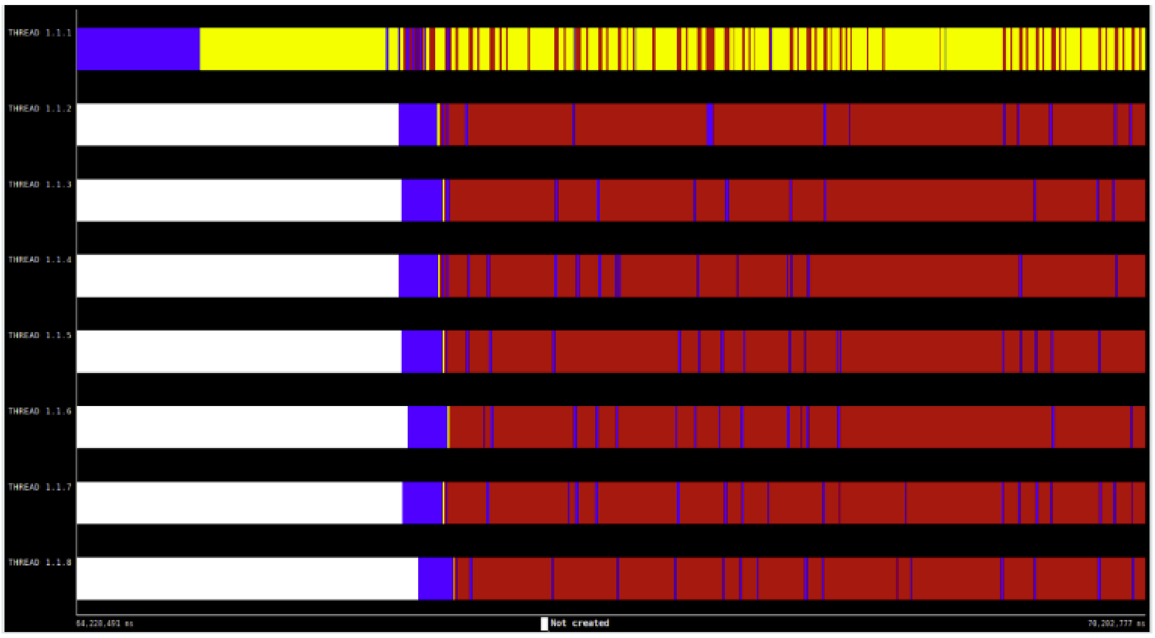


```
paco1208@boada-8:~/lab4$ cat speedup-global-multisort-omp-32768-1024-50-boada-11.txt
1   1.00340715502555366269
2   .98659966499162479061
3   .98659966499162479061
4   .98659966499162479061
5   .98659966499162479061
6   .98494983277591973244
7   .98659966499162479061
8   .98659966499162479061
9   .98659966499162479061
10   .98494983277591973244
11   .98659966499162479061
12   .98659966499162479061
13   .98659966499162479061
14   .98659966499162479061
15   .98659966499162479061
16   .98659966499162479061
17   .98659966499162479061
18   .98659966499162479061
19   .98659966499162479061
20   .98659966499162479061
paco1208@boada-8:~/lab4$
```

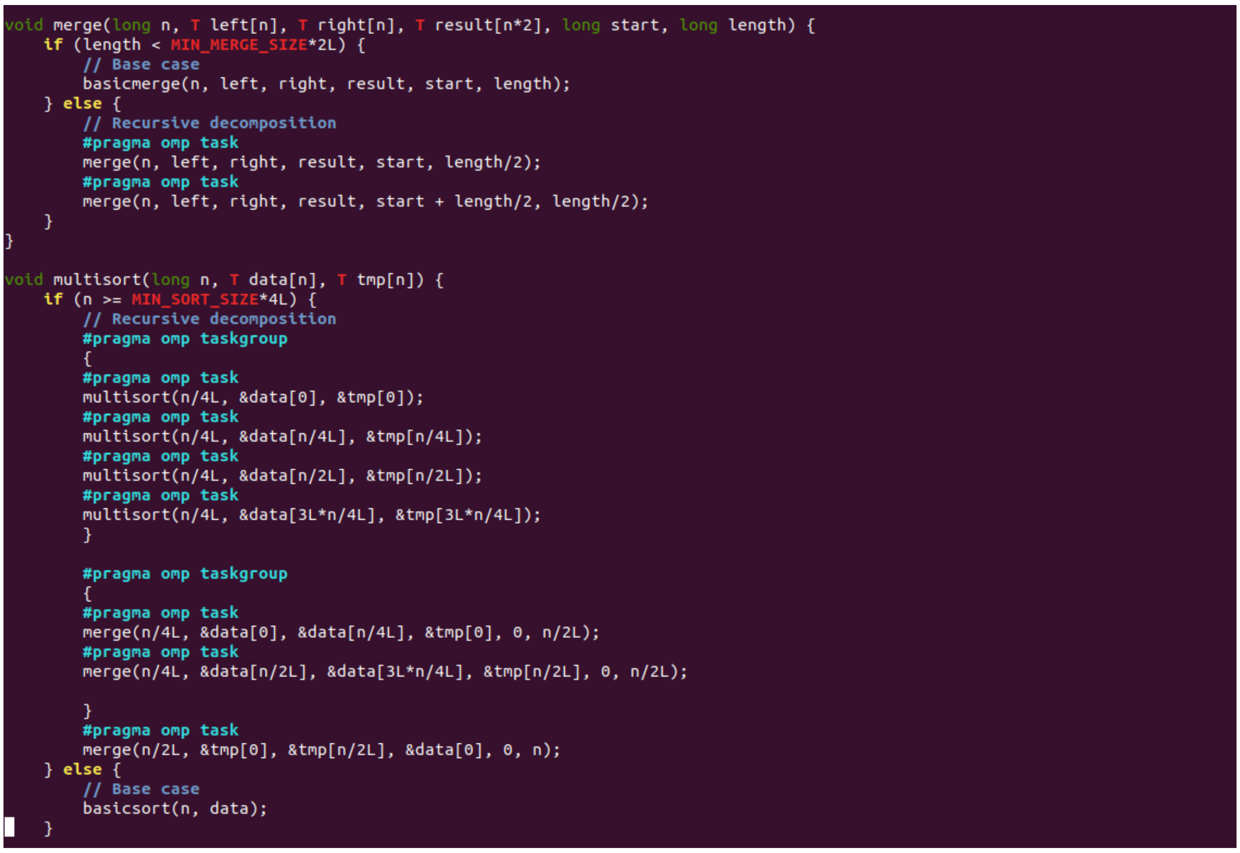
<u>speedup partial:</u>



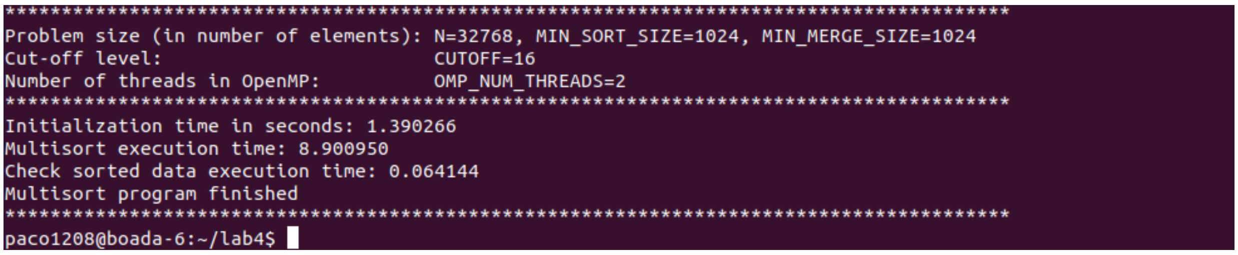
```
paco1208@boada-8:~/lab4$ cat speedup-partial-multisort-omp-32768-1024-50-boada-11.txt
1   1.00830189427418099218
2   .98888954552386762238
3   .98913109595354516001
4   .98950902616324540527
5   .98917636442929254116
6   .98721702042463897998
7   .98906980149653627586
8   .98929238381344851418
9   .98896401570577317759
10   .98833617825310176912
11   .98868090022942009149
12   .98923786074591995465
13   .98847799375888946648
14   .98935615890215112129
15   .98888011971824273445
16   .98932294954591591329
17   .98917391228071513619
18   .98942805724863597218
19   .98907866514815151669
20   .98859234212987288212
paco1208@boada-8:~/lab4$
```



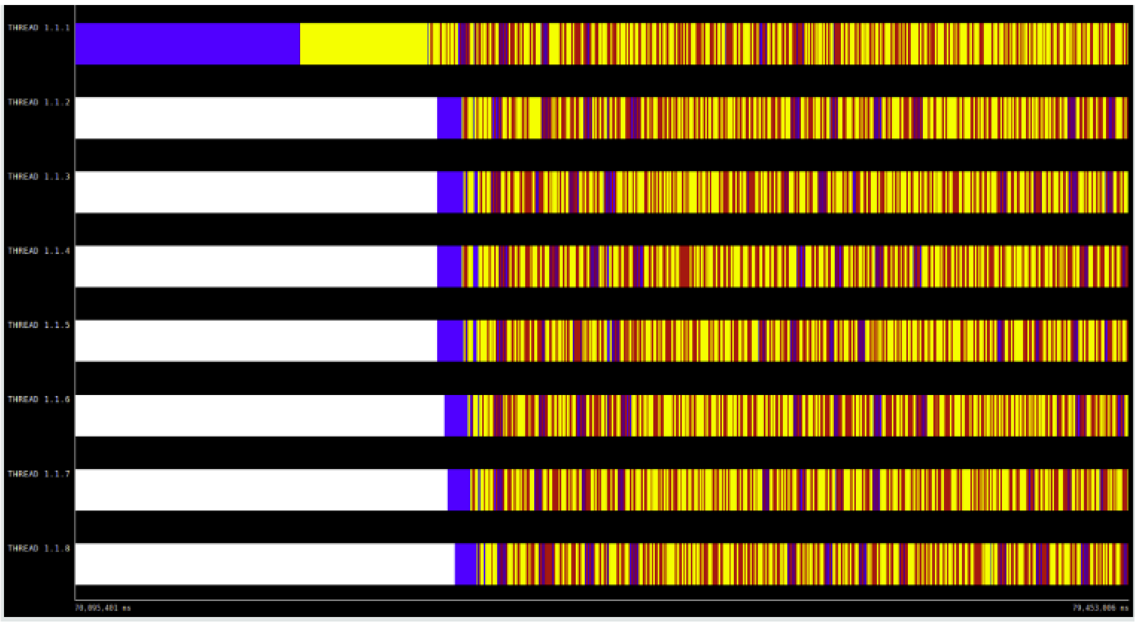
3.2. Tree strategy in OpenMP



```
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);
    }
```

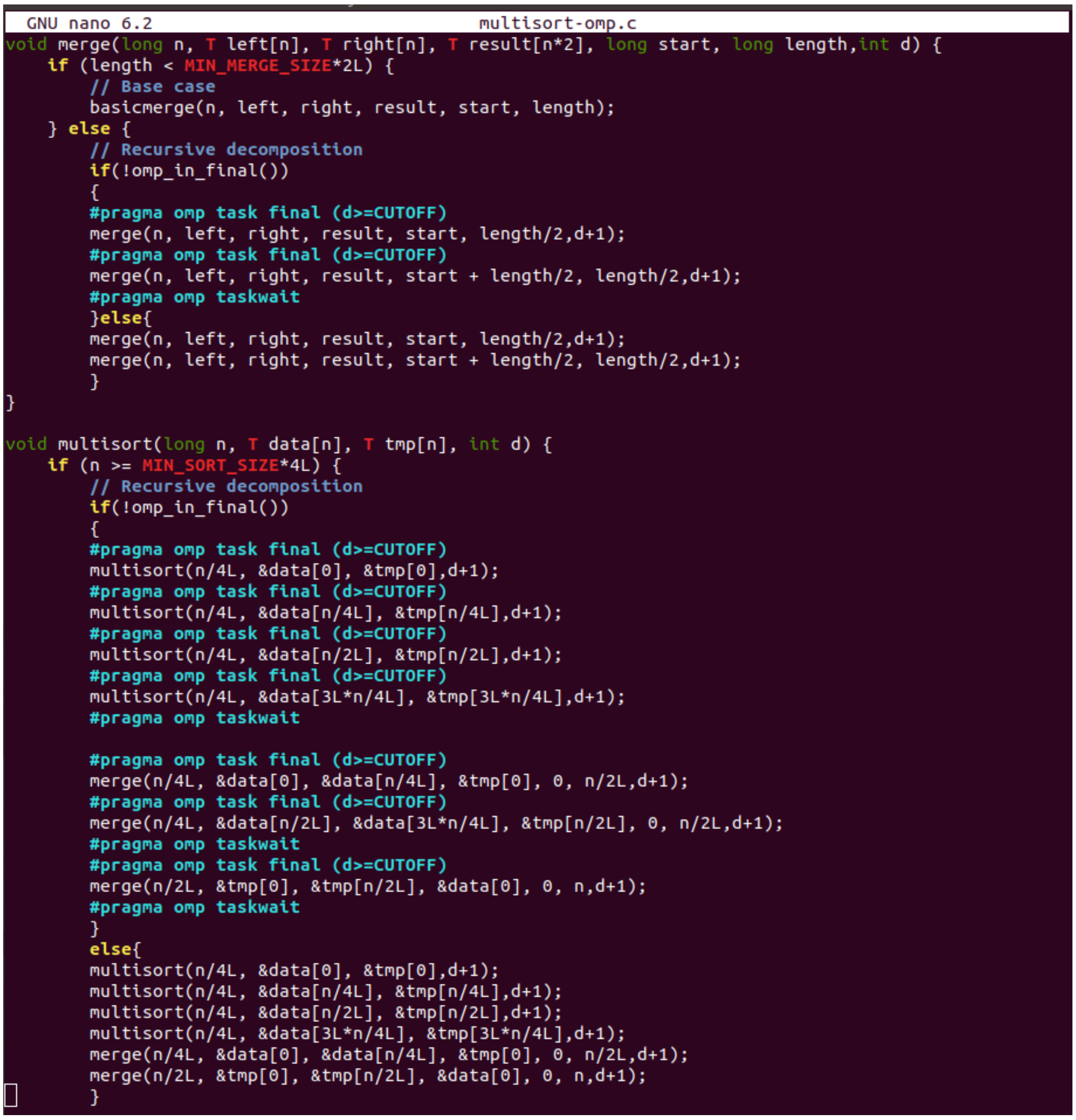


```
*******************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
*******************************************************************************
Initialization time in seconds: 1.390266
Multisort execution time: 8.900950
Check sorted data execution time: 0.064144
Multisort program finished
*******************************************************************************
paco1208@boada-6:~/lab4$
```



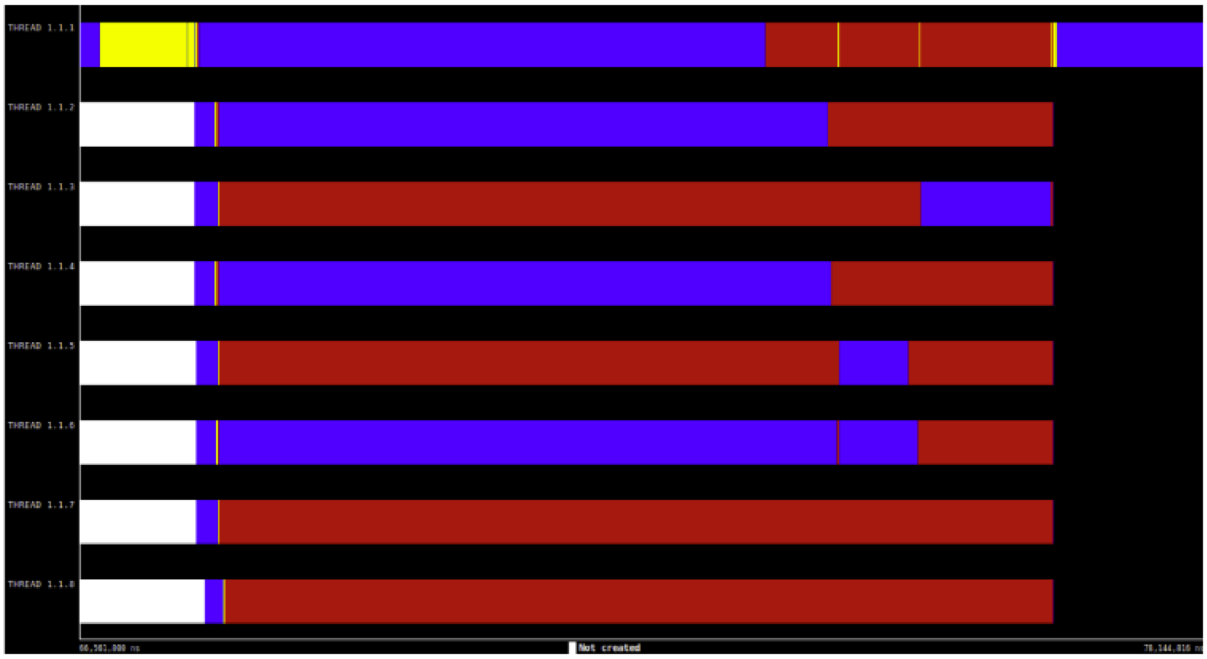
Mientras la versión leaf es implementada creando tascas en los casos base, y la versión del árbol solo las crea antes de las llamadas del multisort y merge functions.
Mientras que en el tiempo de ejecución no hay mucha diferencia, pero la versión árbol es más eficiente.
Podemos observar que las dos versiones tienen problemas de escalabilidad, però la versión árbol mejora considerablemente bien el speedup con la función multisort que puede seguir más las escalabilidad.

3.3. Optimization: Cut-off mechanism

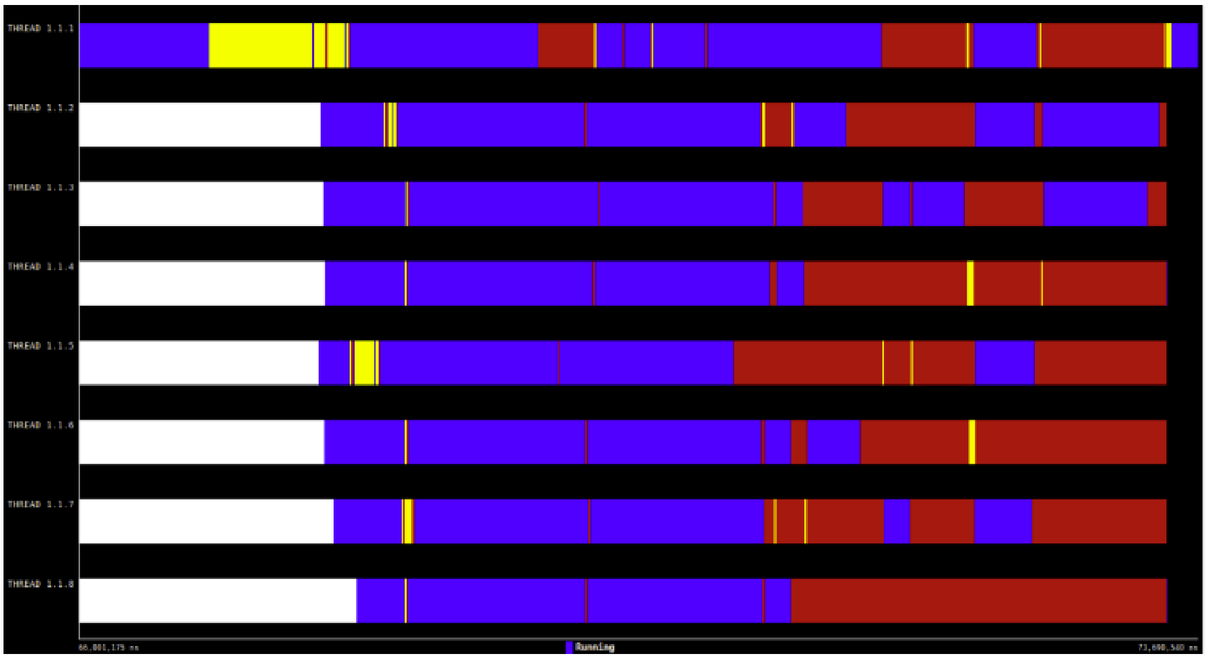


```
  GNU nano 6.2                                   multisort-omp.c
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length,int d) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        if(!omp_in_final())
        {
        #pragma omp task final (d>=CUTOFF)
        merge(n, left, right, result, start, length/2,d+1);
        #pragma omp task final (d>=CUTOFF)
        merge(n, left, right, result, start + length/2, length/2,d+1);
        #pragma omp taskwait
        }else{
        merge(n, left, right, result, start, length/2,d+1);
        merge(n, left, right, result, start + length/2, length/2,d+1);
        }
}

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

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

sbatch -p exectuion submit-extrae.sh multisort-omp 8 0

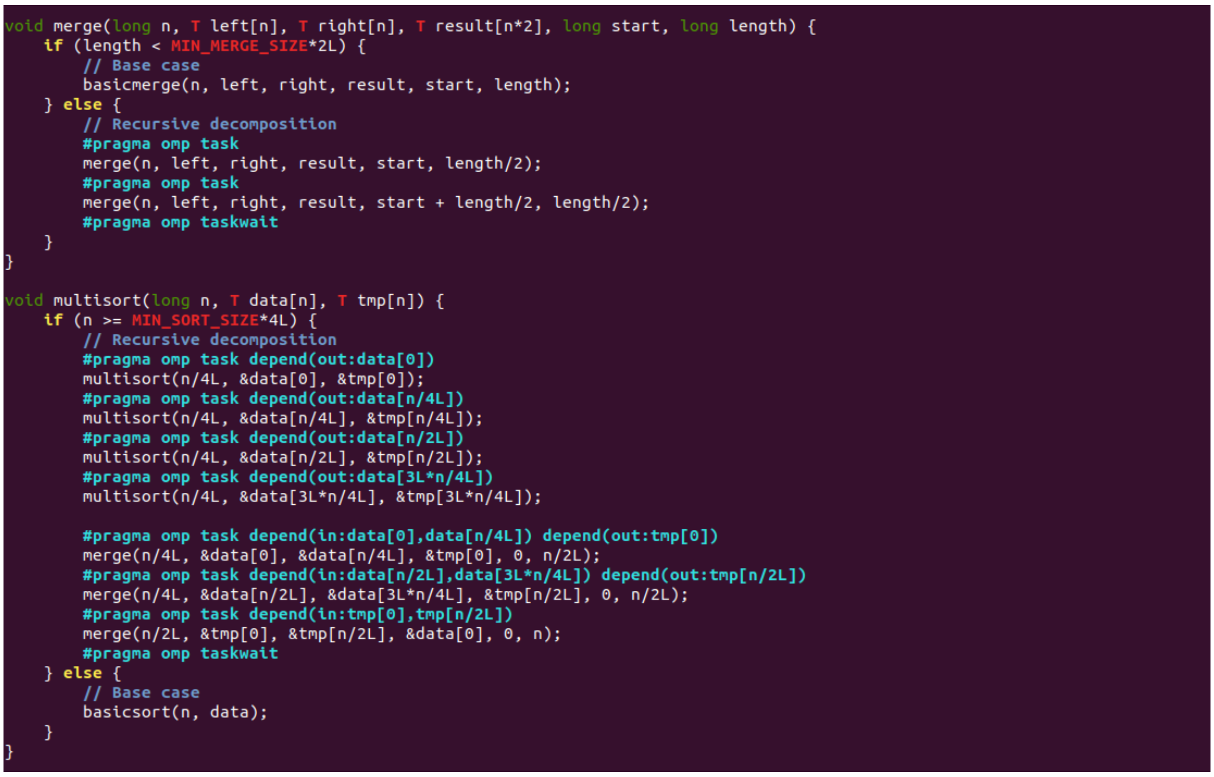


sbatch -p exectuion submit-extrae.sh multisort-omp 8 1



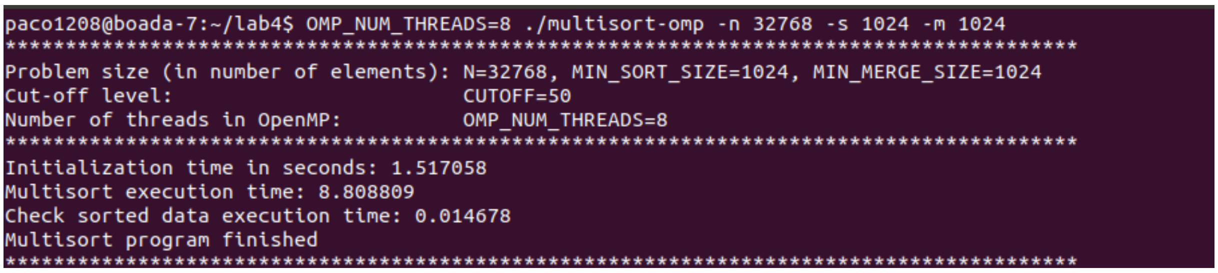
4. Shared-memory parallelisation with OpenMP task using dependencies

For the deliverable: Include the tables of modelfactors, scalability plots and Paraver windows to support the comparison.

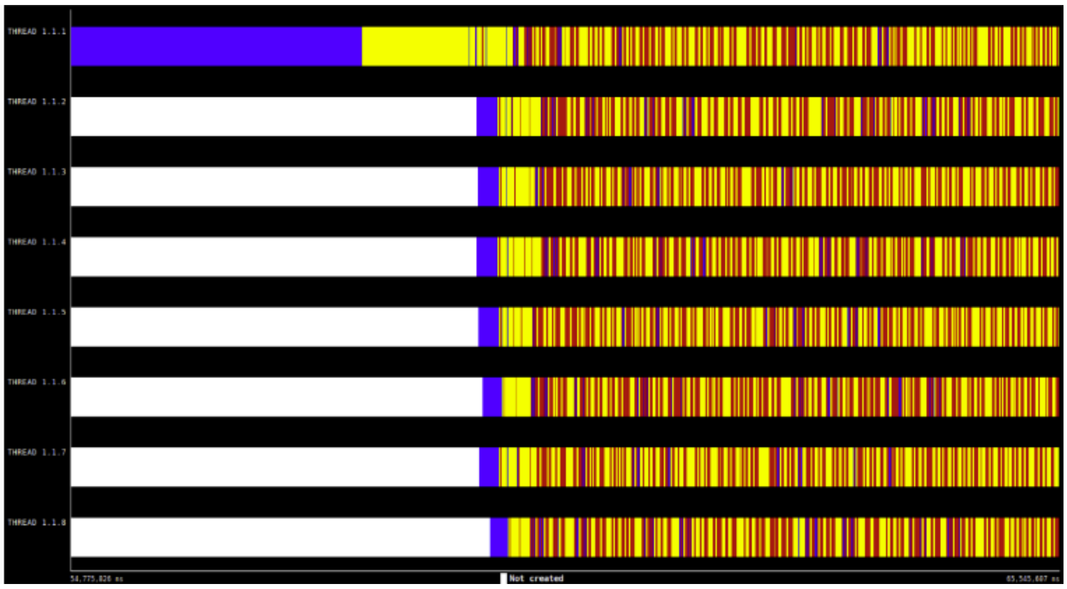


```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left,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

OMP_NUM_THREADS=8 CUTOFF=16 ./multisort-omp -n 32768 -s 1024 -m 1024



```
paco1208@boada-7:~/lab4$ OMP_NUM_THREADS=8 ./multisort-omp -n 32768 -s 1024 -m 1024
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=8
*****************************************************************************************
Initialization time in seconds: 1.517058
Multisort execution time: 8.808809
Check sorted data execution time: 0.014678
Multisort program finished
*****************************************************************************************
```


THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
54,775,826 ns
Not created
65,545,687 ns