

PACO Lab4 - Informe

Ixent Cornella, Arnau Roca (PACO1201)

SESSIÓ 1

Executem primer multisort de forma seqüencial amb sbatch:

```
*****
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
*****
Initialization time in seconds: 0.681094
Multisort execution time: 5.191687
Check sorted data execution time: 0.011787
Multisort program finished
*****
```

Figura 1: Temps d'execució de multisort seqüencial

Ara, pel mergesort, abordarem dues estratègies:

- Dividir en tasques les “leafs”
- Dividir en tasques el “tree”

Per tant, primer dividirem les tasques quan ja no hi ha recursivitat, és a dir, leafs, i després quan s'està fent recursitivitat (tree). Aquest és el codi canviat:

```
#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
        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");
    }
}
```

Figura 2: Extracte del codi modificat de multisort-tareador.c

I aquest és el graf de dependències resultant amb tareador:

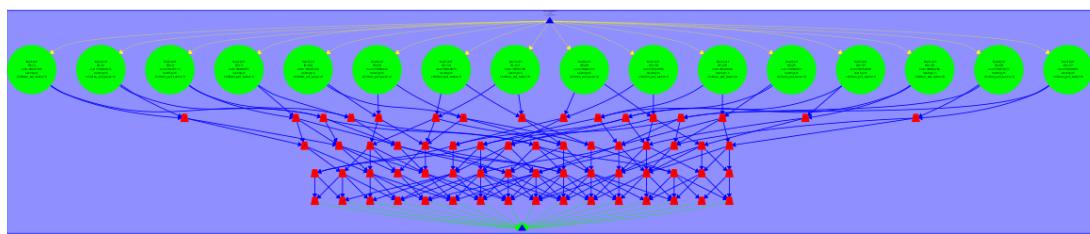


Figura 3: Graf de dependències del codi modificat de multisort-tareador.c

Es pot observar que inicialment hi ha 16 tasques, cosa que té sentit tenint en compte els paràmetres inicials (es fan 4 divisions recursivament, i 4 més per cada), ens deixen amb 16 tasques que son ja feta la recursivitat.

Ara farem el mateix però canviant l'estratègia, en aquest cas cal fer que les tasques es divideixin fent la recursivitat. Aquest és el codi canviat:

```

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

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

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

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

```

Figura 4: Extracte del codi modificat de multisort-tareador.c

I aquest és el graf de dependències resultant amb tareador:

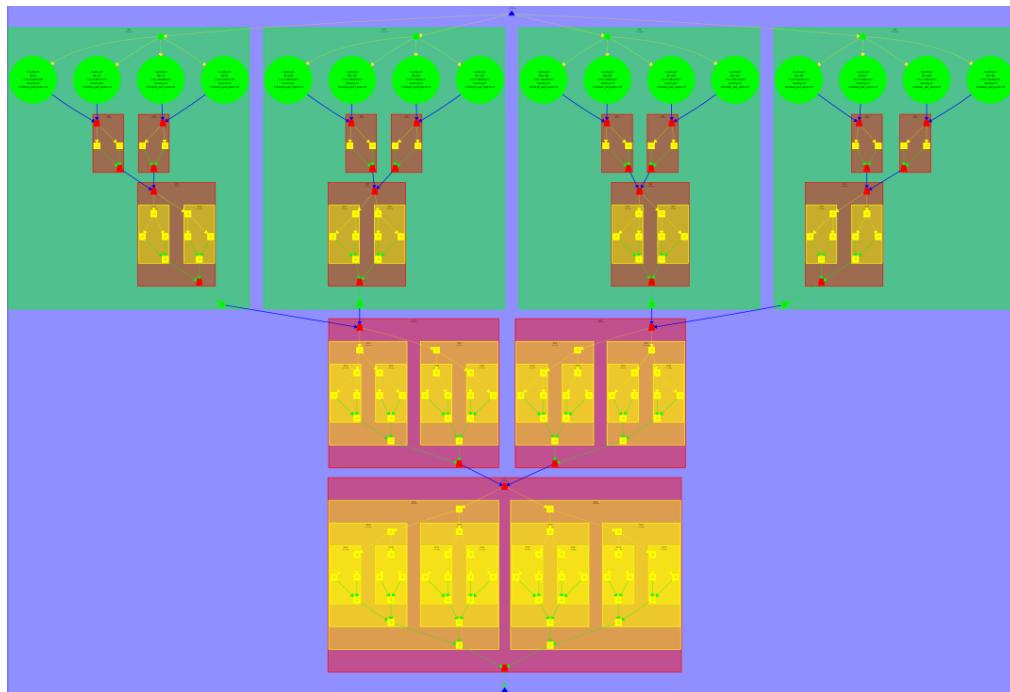


Figura 5: Graf de dependències de tasques del codi modificat de multisort-tareador.c

Ara amb el codi fet per a que faci tasques quan fa recursivitat, es veu la divisió de 1 tasca a 16 abans esmentada. Després hi ha les tasques no recursives, produint com a resultat aquest graf de tasques.

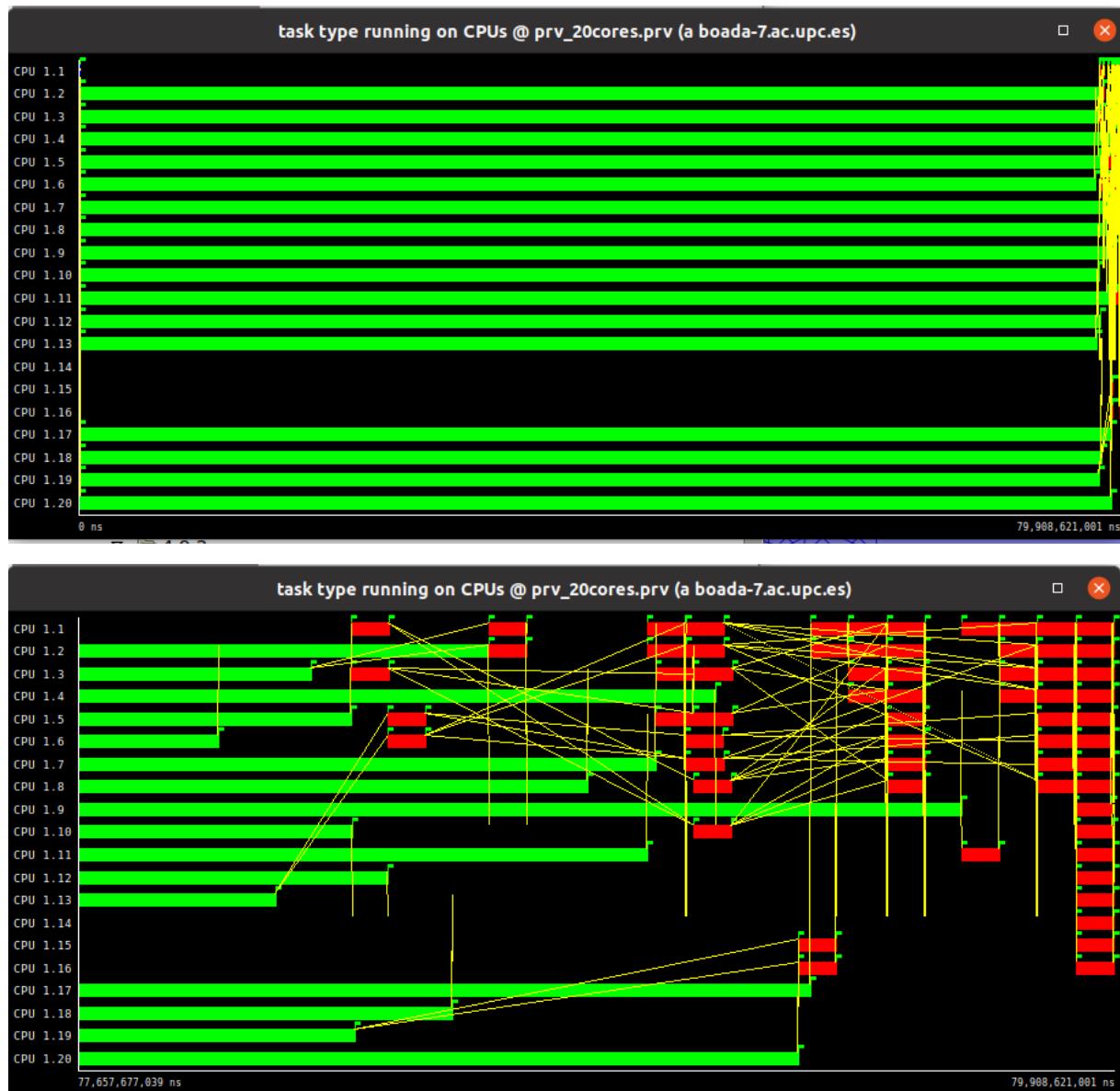
La principal diferència és on es creen les tasques, en un és al principi quan s'està fent la recursivitat (tree) i en l'altra quan ja s'ha acabat la recursivitat (leaf). Això es pot veure clarament als grafs de dependències (Figura 3 i Figura 5).

Els punts del codi on hauríem de incluir les sincronitzacions es poden veure clarament a la figura 5 (son just abans de cada rectangle). Per tant nosaltres en aquest cas optarem per posar un taskgroup (el qual ja té un taskwait implicit) després de cada crida a merge, que és on s'acaben els nivells de l'arbre en la imatge.

Com podem veure en les figures 6 i 7 que son les que corresponen a l'execució de la versió leaf,



Figures 6 i 7: Simulació de paraver des de tareador, (tree)



Figures 8 i 9: Simulació de paraver des de tareador, (leaf)

SESSIÓ 2

ESTRATEGIA LEAF

```

GNU nano 6.2                               a0033021@BI101-14: ~/EscriptorI/home-ubewan-a0033021
multisort-omp.c *

#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
        #pragma omp taskgroup
        {
            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 taskgroup
        {
            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
        #pragma omp task
        basicsort(n, data);
    }
}

```

Figura 10: Codi de l'estrategia leaf

```

GNU nano 6.2                               submit-omp.sh.o40227
icc -g -std=c99 -Wall -O2 -fp-model precise -fopenmp multisort-omp.c kernels.o -lm -o multisort-omp
: Terminal:::::
: Terminal::omp_4_boada-11.times.txt
::::::::::::::::::
*****
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=4
*****
Initialization time in seconds: 0.687899
Multisort execution time: 1.740554
Check sorted data execution time: 0.013507
Multisort program finished
*****

```

Figura 11: Execució del codi anterior

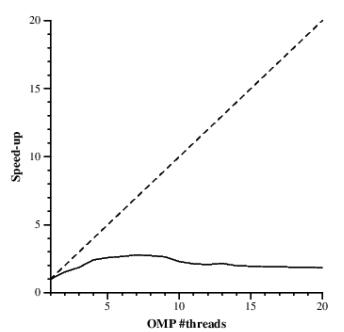
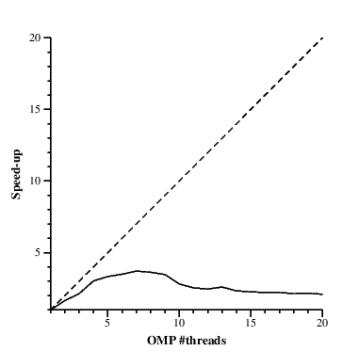


Figura 12: gràfiques de escalabilitat del leaf

Overview of the Efficiency metrics in parallel fraction, $\phi=89.06\%$										
Number of processors	1	2	4	6	8	10	12	14	16	
Global efficiency	91.94%	40.34%	23.33%	15.14%	11.74%	8.98%	7.07%	5.71%	4.64%	
Parallelization strategy efficiency	91.94%	53.44%	35.60%	26.32%	20.48%	16.21%	13.51%	11.40%	9.58%	
Load balancing	100.00%	98.48%	91.54%	61.99%	39.45%	30.55%	22.63%	18.02%	15.19%	
In execution efficiency	91.94%	54.26%	38.89%	42.46%	51.91%	53.07%	59.70%	63.29%	63.06%	
Scalability for computation tasks	100.00%	75.48%	65.54%	58.64%	57.35%	55.39%	52.34%	50.05%	48.44%	
IPC scalability	100.00%	67.95%	58.43%	55.02%	54.03%	53.98%	51.11%	49.10%	47.57%	
Instruction scalability	100.00%	111.96%	113.45%	113.03%	112.26%	112.02%	111.23%	110.79%	111.00%	
Frequency scalability	100.00%	99.23%	98.87%	94.30%	94.55%	91.61%	92.08%	92.02%	91.75%	

Table 2: Analysis done on Wed Nov 16 01:46:18 PM CET 2022, paco1201



Statistics about explicit tasks in parallel fraction										
Number of processors	1	2	4	6	8	10	12	14	16	
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	
LB (number of explicit tasks executed)	1.0	0.71	0.75	0.77	0.79	0.77	0.81	0.83	0.8	
LB (time executing explicit tasks)	1.0	0.8	0.81	0.78	0.8	0.78	0.81	0.83	0.81	
Time per explicit task (average us)	2.79	3.59	4.02	4.24	4.06	4.14	4.11	4.11	4.13	
Overhead per explicit task (synch %)	1.38	69.19	180.42	337.49	524.11	718.72	962.62	1229.64	1539.81	
Overhead per explicit task (sched %)	9.03	37.59	47.41	36.24	29.28	29.24	25.41	24.13	26.35	
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	

Table 3: Analysis done on Wed Nov 16 01:46:18 PM CET 2022, paco1201

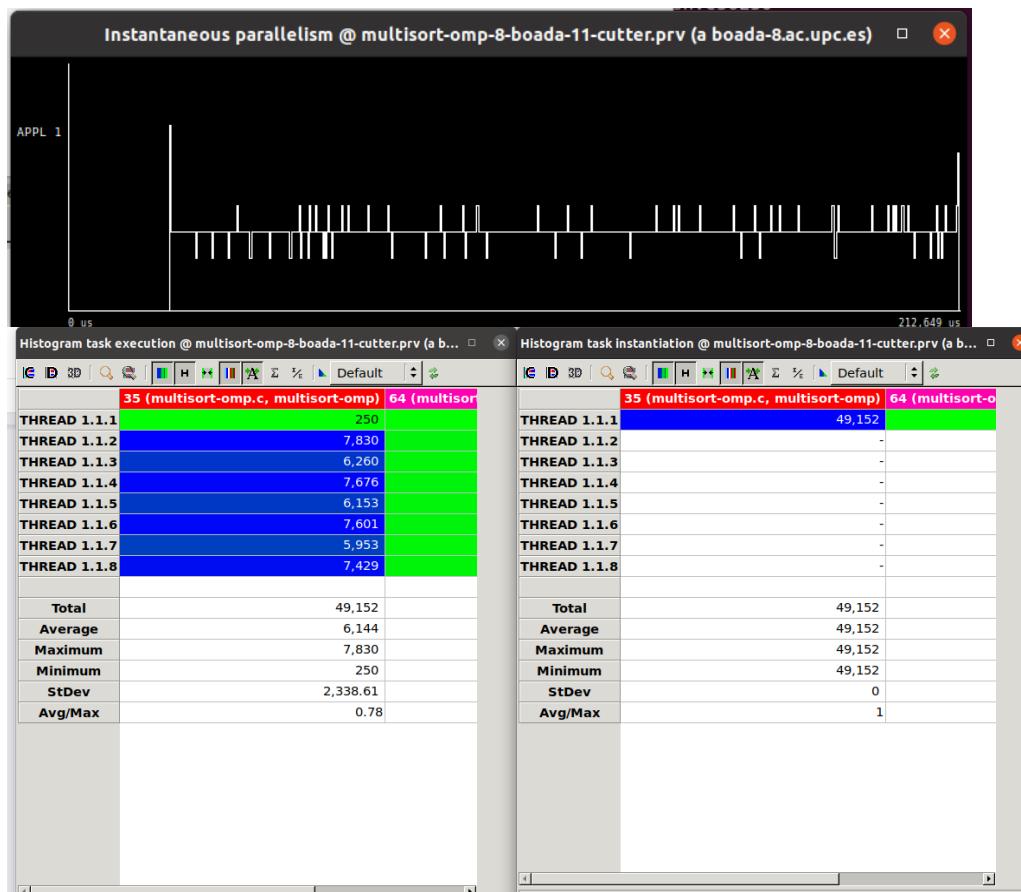
Overview of whole program execution metrics										
Number of processors	1	2	4	6	8	10	12	14	16	
Elapsed time (sec)	0.22	0.24	0.21	0.21	0.21	0.22	0.23	0.25	0.26	
Speedup	1.00	0.89	1.01	1.00	1.01	0.97	0.93	0.88	0.82	
Efficiency	1.00	0.45	0.25	0.17	0.13	0.10	0.08	0.06	0.05	

Table 1: Analysis done on Wed Nov 16 01:46:18 PM CET 2022, paco1201

Speed-up wrt sequential time (multisort function only)
Generated by paco1201 on Wed Nov 16 01:26:13 PM CET 2022



Figura 15: execució normal



Com es pot observar a la figura 15, les tasques no es realitzen en paral·lel fins que s'arriba a les “leaves”, és seqüencial, per tant fins que no s'acaba la recursivitat (bona part de l'execució) realment no hi ha paral·lelisme.

S'executa de 4 en 4 ja que tenim el group, encara que abans ens digues que podien ser les 16, per tant estem posant mes sincronització de la necessària (el rosa és el merge (a més, tarda més que el sort) i veiem que només poden ser 4 alhora, a la figura 15)

TREE

Ara modificarem el codi per a que generi una tasca cada cop que hi ha recursivitat. L'únic que cal fer és treure els pragmas anteriors i afegir tasques a crides recursives, a més de dos taskgroups pel 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
        #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);
    }
}
```

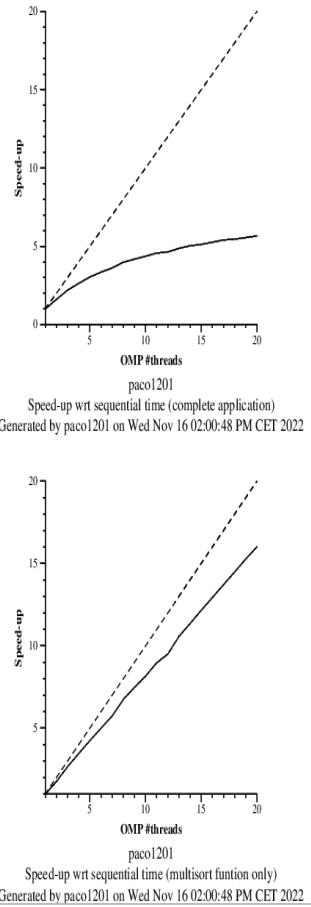
Figura 18: Codi del tree

```
*****
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=4
*****
Initialization time in seconds: 0.683609
Multisort execution time: 1.500964
Check sorted data execution time: 0.014713
Multisort program finished
*****
paco1201@boada-7:~/lab4$ 
```

Figura 19: Execució del tree

Overview of whole program execution metrics								
Number of processors	1	2	4	6	8	10	12	14
Elapsed time (sec)	0.22	0.24	0.21	0.21	0.21	0.22	0.23	0.25
Speedup	1.00	0.89	1.01	1.00	1.01	0.97	0.93	0.88
Efficiency	1.00	0.45	0.25	0.17	0.13	0.10	0.08	0.06

Table 1: Analysis done on Wed Nov 16 01:46:18 PM CET 2022, paco1201



Overview of the Efficiency metrics in parallel fraction, $\phi=89.06\%$									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	91.94%	40.34%	23.33%	15.44%	11.74%	8.98%	7.07%	5.71%	4.64%
Parallelization strategy efficiency	91.94%	53.44%	35.60%	26.32%	20.48%	16.21%	13.51%	11.40%	9.58%
Load balancing	100.00%	98.48%	91.54%	61.99%	39.45%	30.55%	22.63%	18.02%	15.19%
In execution efficiency	91.94%	54.26%	38.89%	42.46%	51.91%	53.07%	59.70%	63.29%	63.06%
Scalability for computation tasks	100.00%	75.48%	65.54%	58.64%	57.35%	55.39%	52.34%	50.05%	48.44%
IPC scalability	100.00%	67.95%	58.43%	55.02%	54.03%	53.98%	51.11%	49.10%	47.57%
Instruction scalability	100.00%	111.96%	113.45%	113.03%	112.26%	112.02%	111.23%	110.79%	111.00%
Frequency scalability	100.00%	99.23%	98.87%	94.30%	94.55%	91.61%	92.08%	92.02%	91.75%

Table 2: Analysis done on Wed Nov 16 01:46:18 PM CET 2022, paco1201

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.71	0.75	0.77	0.79	0.77	0.81	0.83	0.8
LB (time executing explicit tasks)	1.0	0.8	0.81	0.78	0.8	0.78	0.81	0.83	0.81
Time per explicit task (average us)	2.79	3.59	4.02	4.24	4.06	4.14	4.11	4.11	4.13
Overhead per explicit task (sync %)	1.38	69.19	180.42	337.49	524.11	718.72	962.62	1229.64	1539.81
Overhead per explicit task (sched %)	9.03	37.59	47.41	36.24	29.28	29.24	25.41	24.13	26.35
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Wed Nov 16 01:46:18 PM CET 2022, paco1201

Figura 20, 21 i 22: Taules del model factors i gràfiques del tree

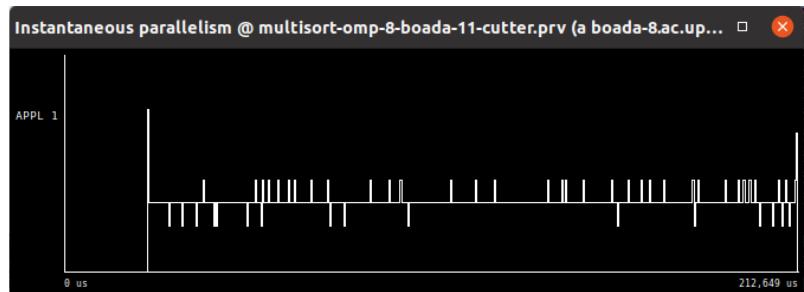


Figura 23: execució instantaneus parallelism

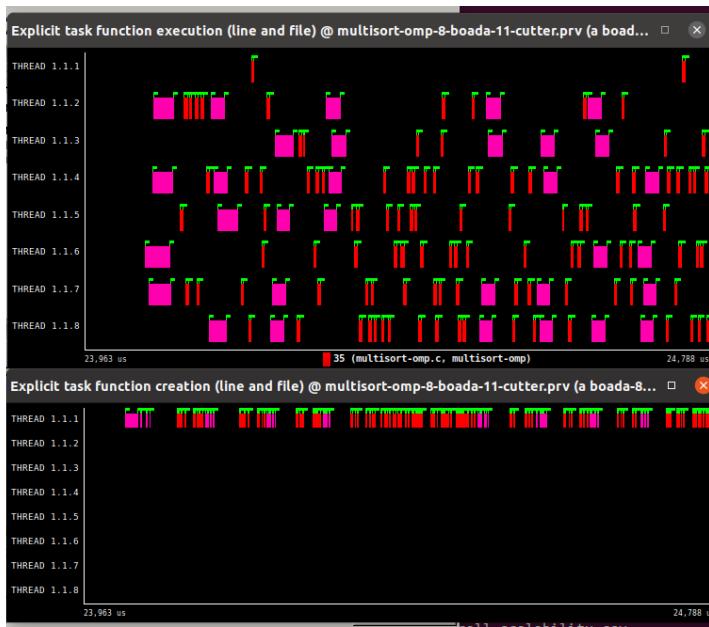


Figura 24: execució normal

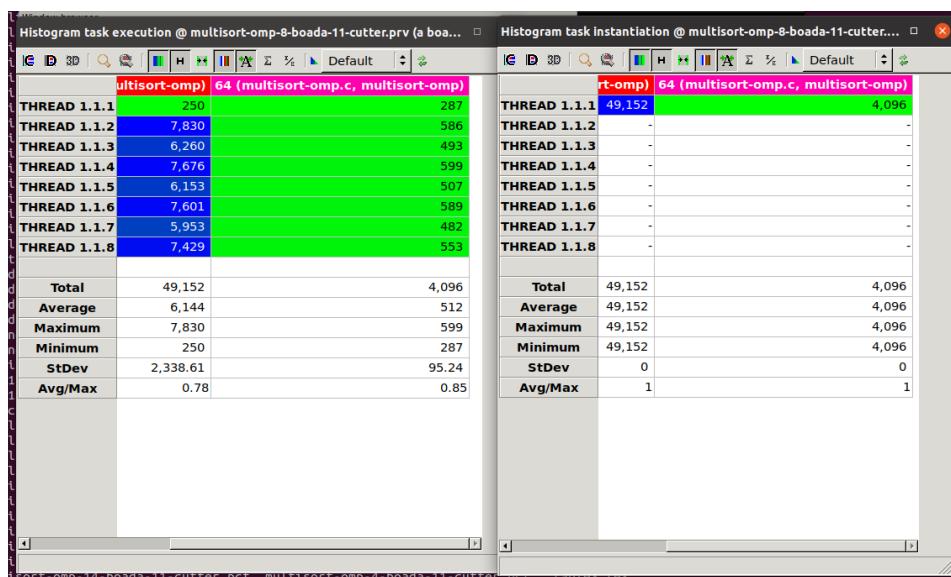


Figura 25: Número de tasques

Ara és pot observar com cada cop que es crea una crida recursiva també es crea una tasca, i ja que no hi ha la sequencialitat a la part més extensa en temps del programa (la recursivitat), tindrem threads treballant desde bon principi, però més overhead. Això a grans efectes el que ens causa és un millor speed-up que amb leaf, però no acaba de ser una solució òptima.

TREE AMB CUTOFF

Ara se'ns demana de fer un mecanisme de cutoff amb el qual podem decidir quan deixar de crear tasques un cop ja s'ha arribat a la profunditat (recursivament) desitjada. Per fer-ho, afegirem al codi un paràmetre i variable que serà “depth”, representant així la profunditat màxima a la que podem arribar. També usarem finals com a pragmas per a poder decidir si cal fer tasques o no.

```

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
        #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);
    }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #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 {
        // Base case
        basicsort(n, data);
    }
}

```

Figura 26: Codi del tree modificat amb cutoff (nota: falta distingir si ja s'ha arribat al final o no amb un if-else a cada funció).

Aquest és el resultat de l'execució (CUTOFF 50, per tant no hi haura massa diferència):

```

*****
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=1
*****
Initialization time in seconds: 0.684943
Multisort execution time: 5.274082
Check sorted data execution time: 0.011182
Multisort program finished
*****
paco1201@boada-7:~/lab4$ sbatch submit-omp.sh multisort-omp 1

```

Figura 27: Resultat de l'execució amb cutoff = 50.

Si fem proves amb diferents cutoffs, veiem que el millor valor per al cutoff ronda el 6, aquí es veu una comparació de cutoff = 1 i cutoff = 6, i finalment una comparació amb molts valors de cutoff:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.20	0.14	0.10	0.09	0.08	0.08	0.08	0.08	0.08
Speedup	1.00	1.40	1.99	2.13	2.32	2.35	2.37	2.41	2.40
Efficiency	1.00	0.70	0.50	0.36	0.29	0.23	0.20	0.17	0.15

Table 1: Analysis done on Wed Nov 23 02:09:20 PM CET 2022, paco1201

Figura 28: Resultat de l'execució amb cutoff = 1.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.15	0.09	0.06	0.06	0.05	0.05	0.05	0.05	0.05
Speedup	1.00	1.72	2.62	2.69	3.06	3.21	3.21	3.19	3.23
Efficiency	1.00	0.86	0.66	0.45	0.38	0.32	0.27	0.23	0.20

Table 1: Analysis done on Wed Nov 23 02:13:40 PM CET 2022, paco1201

Figura 29: Resultat de l'execució amb cutoff = 6.

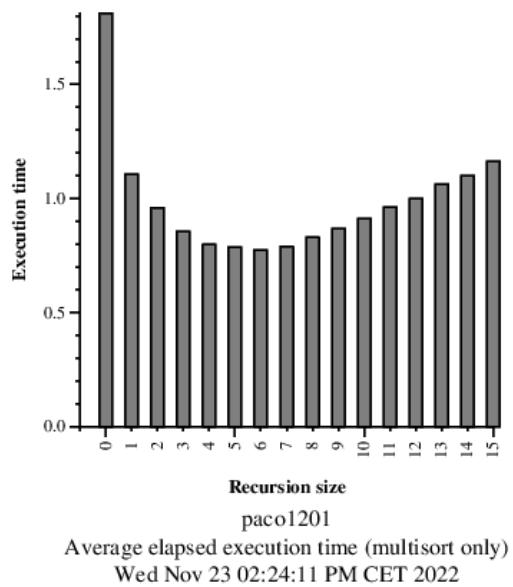


Figura 30: Comparació de temps d'execució de diferents cutoffs.

Vista la comparació, sembla que un cutoff de 6 és més òptim per al temps d'execució, ja que combina paralelisme (la creació de tasques per a fer la recursivitat) amb el control d'overhead, donant-nos així el millor resultat.

SESSIÓ 3

DEPEND

Ara se'ns demana de modificar el codi del tree amb cutoff per a fer-ho amb clàusules “depend”. Aquestes haurien de millorar l'eficiència del programa.

```
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()){
            // Recursive decomposition
            #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);
            #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);
        }
    }
}
```

Figura 31: Funció merge del codi modificada amb depends.

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

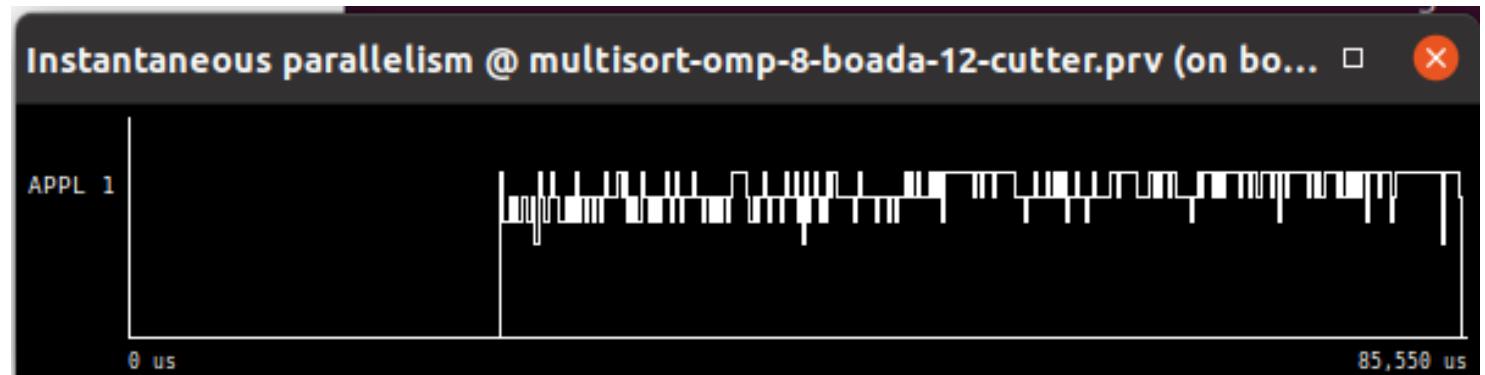
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);

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

Figura 32: Funció multisort del codi modificada amb depends.

Com veiem en les figures 31 i 32, el codi ens ha quedat de la següent manera.

Afegim els depends amb les variables que toquen. També cal destacar la inclusió d'aquesta comprovació de si estem en el final, per tal d'evitar fer els depends en les últimes clàusules.



Overview of the Efficiency metrics in parallel fraction, $\phi=88.07\%$									
number of processors	1	2	4	6	8	10	12	14	16
total efficiency	93.51%	68.48%	52.13%	38.63%	32.41%	25.69%	21.38%	18.55%	15.50%
parallelization strategy efficiency	93.51%	73.36%	61.89%	50.09%	42.17%	34.31%	28.75%	24.64%	20.38%
load balancing	100.00%	98.52%	99.14%	97.26%	95.50%	88.80%	91.61%	91.72%	95.13%
execution efficiency	93.51%	74.47%	62.43%	51.50%	44.16%	38.64%	31.38%	26.87%	21.43%
availability for computation tasks	100.00%	93.35%	84.24%	77.12%	76.86%	74.88%	74.38%	75.30%	76.03%
C scalability	100.00%	87.66%	80.57%	77.11%	77.10%	77.18%	77.27%	78.66%	79.50%
structure scalability	100.00%	106.81%	106.87%	106.82%	106.77%	106.77%	106.79%	106.74%	106.76%
frequency scalability	100.00%	99.70%	97.84%	93.63%	93.36%	90.88%	90.14%	89.67%	89.58%

Table 2: Analysis done on Sat Nov 26 08:59:59 PM CET 2022, paco1201

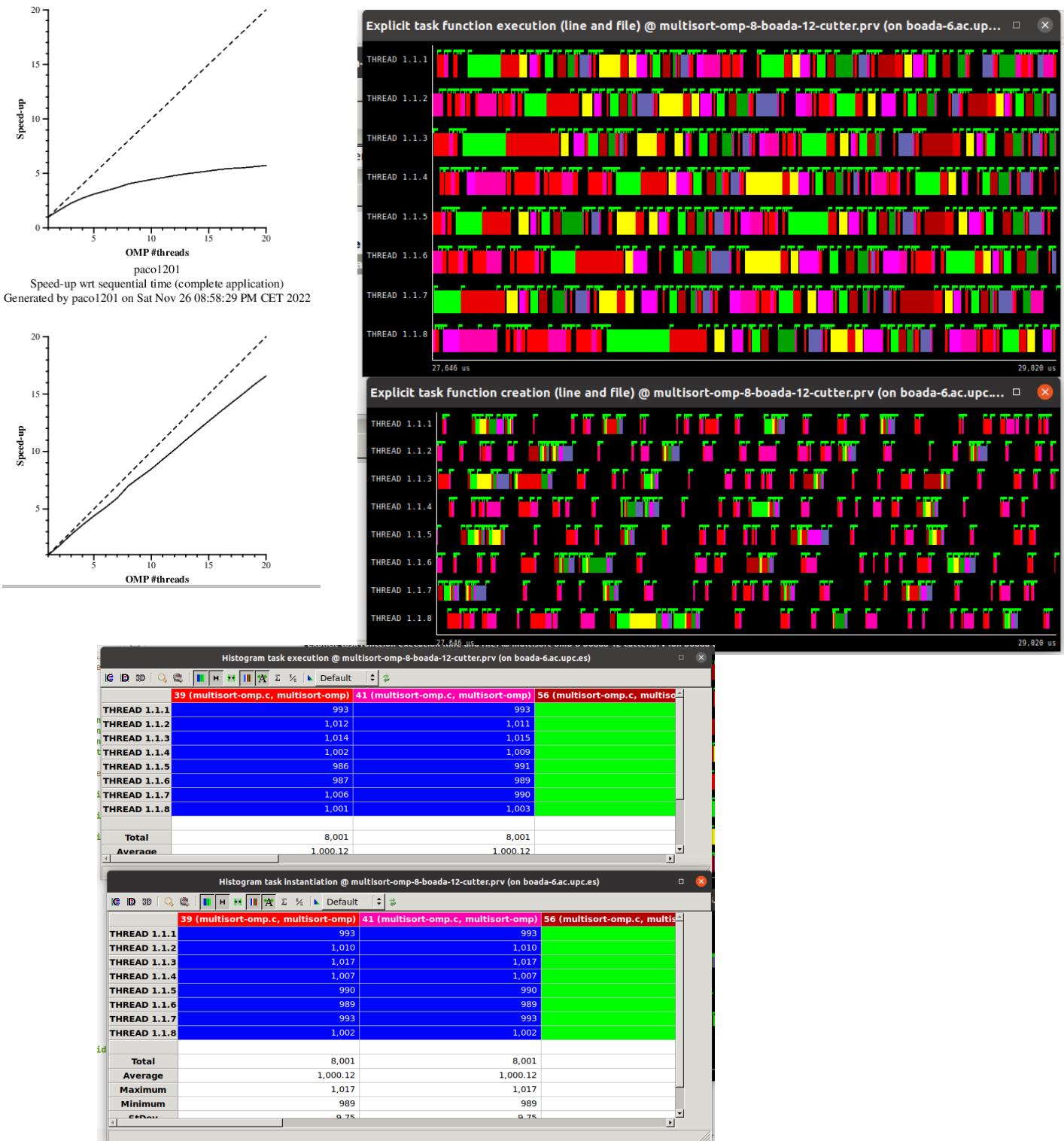
Statistics about explicit tasks in parallel fraction									
number of processors	1	2	4	6	8	10	12	14	16
number of explicit tasks executed (total)	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0
(number of explicit tasks executed)	1.0	1.0	0.99	0.95	0.97	0.97	0.96	0.97	0.99
(time executing explicit tasks)	1.0	0.99	0.99	0.98	0.99	0.98	0.99	0.98	0.99
time per explicit task (average us)	5.69	7.64	9.49	12.38	14.39	17.95	21.27	24.34	28.83
overhead per explicit task (synch %)	2.87	20.47	29.22	37.4	43.49	48.37	52.33	54.74	57.47
overhead per explicit task (sched %)	4.73	11.2	18.7	27.51	33.66	40.31	45.18	49.09	53.42
number of taskwait/taskgroup (total)	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0

Table 3: Analysis done on Sat Nov 26 08:59:59 PM CET 2022, paco1201

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.19	0.14	0.10	0.09	0.09	0.09	0.09	0.09	0.09
Speedup	1.00	1.39	1.94	2.08	2.26	2.23	2.23	2.25	2.18
Efficiency	1.00	0.69	0.48	0.35	0.28	0.22	0.19	0.16	0.14

Table 1: Analysis done on Sat Nov 26 08:59:59 PM CET 2022, paco1201

Figures 33, 34, 35: Instantaneous parallelism i taules del model factors



Figures 36, 37, 38: Gràfiques del model factors, execució “normal del codi” i nombre de tasques creades i executades

Com veiem ens les simulacions anteriors (Figures 36,37 i 38) podem observar una gran diferència en el nombre de tasques de la figura 38, ja que tenim dividides les crides i ens deixa amb unes 80000 tasques per cada crida. També es pot veure el nombre total en la figura 34 on veiem el total de tasques (25557), la qual ve molt afectada degut al cutoff de 6 que fem. Si comparem la figura 24 amb la 37 podem veure que tenim una molt millor distribució i paralelisme, fet que resulta amb un temps d'execució molt menor vist en el instantaneous parallelism de la figura 33.

CONCLUSIONS

Un cop vistes totes aquestes possibilitats de paralelització i optimització del programa, podem extreure algunes conclusions.

Primerament hem vist que la versió tree, és millor respecte la leaf, i per tant hem seguit optimitzant encara més i optimitzant-la (introduint cutoff, amb depends, etc). Per tant pas a pas hem anat descobrint com d'important és anar gestionant les dependències que ens sorgeixen en execucions. També veiem que s'ha de paralelitzar tot el codi i no només parts per poder aconseguir la millor eficiència i temps d'execució.

Hem vist eines com el cutoff que ens poden ser molt útils, i hem après a combinar diferents estratègies de paralelització (tant explícites com implícites), com ara el task group, el taskwait i depend. Finalment cal dir que paralelitzar un codi recursiu es pot observar que és més difícil de plantejar però si està ben fet pot ser una molt bona opció.