PARAL·LELISME

Entrega lab2

Autors:
Sergi Soriano
Mingjian Chen

Grup: 21

4.1.1 Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

```
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("merge2");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("merge2");
                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("multi");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("multi");

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

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

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

                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("merge2");

                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);
        }
}
```

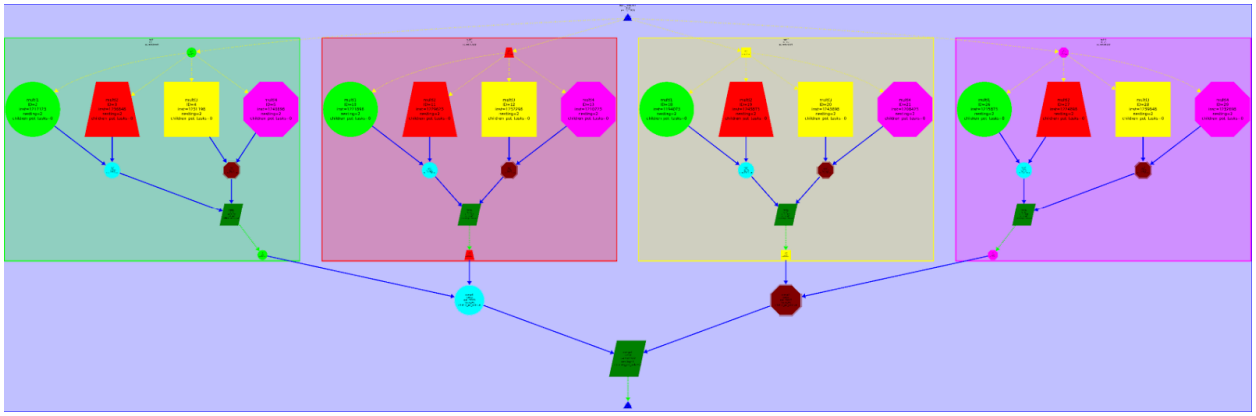


Figure 1: Graf de dependències sense tenir en compte la funció merge

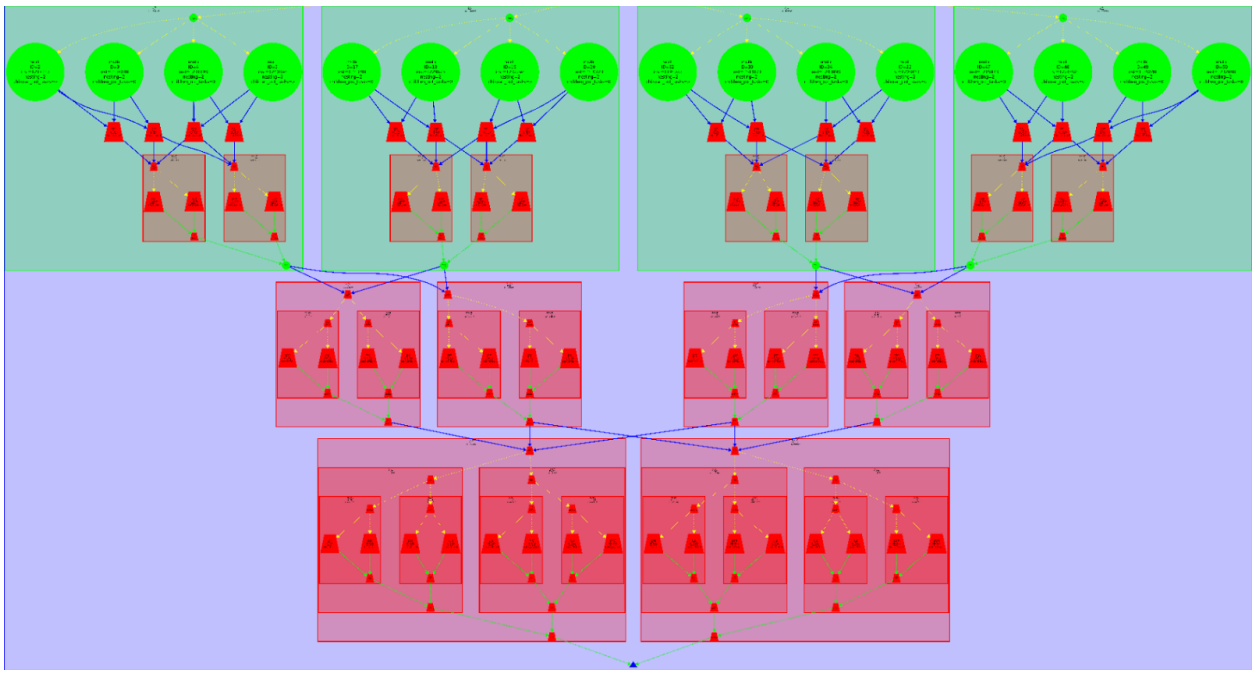


Figure 2: Graf de dependències tenint en compte la funció merge

Per poder observar bé les dependències en una funció recursiva cal veure que passa a cada crida que no compleix el cas base.

Podem veure que les crides recursives a *multisort* no tenen cap dependència entre elles, ara bé, si que generen dependència amb les crides a *merge*. És a dir, un cop hem dividit el vector i ordenat per trossos, després cal reconstruir aquest, per tant cal que els trossos a unir ja hagin acabat. Aquesta dependència es crea a les dues primeres crides a *merge.*

La tercera crida a *merge* depèn de les altres dues anteriors.

Pel que fa a les crides de la funció *merge*, la dependència que tenen és que han d'haver acabat abans de poder-se tornar a executar.

4.1.2 Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

# Processors	Execution time	Speed-up
1	20,339,974,001 ns	1
2	10,301,204,001 ns	1.97
4	5,279,337,001 ns	3.85
8	2,816,632,001 ns	7.22
16	1,584,057,001 ns	12.84
32	1,584,057,001 ns	12.84
64	1,584,057,001 ns	12.84

El cas ideal és que el speed up sigui igual al número de threats que s'executa en aquell moment. Per tant el resultat no es tanca amb el cas ideal ja que quan s'executa amb 16,32 i 64 threats el speed up es de 12.84.
Veiem que amb més número de processadors que hi hagin, el temps d'execució seria menys, així que arriba un moment en què les tasque no es poden descomposar més, i el speed-up es manté.

4.2.1. Include the relevant portion of the codes that implement the two versions (Leaf and Tree), commenting whatever necessary.

```
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
        {
#if _EXTRAE_
                Extrae_event(PROGRAM, MERGE);
#endif
                basicmerge(n, left, right, result, start, length);
#if _EXTRAE_
                Extrae_event(PROGRAM, END);
#endif
        }
        } 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);
                #pragma omp taskwait

        } else {

        // Base case
#if _EXTRAE_

                Extrae_event(PROGRAM, SORT);
#endif
                basicsort(n, data);
#if _EXTRAE_
                Extrae_event(PROGRAM, END);
#endif
        }
}
```

Figure 3: Codi de multisort-omp en versió leaf.

```
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
#if _EXTRAE_
                Extrae_event(PROGRAM, MERGE);
#endif
                basicmerge(n, left, right, result, start, length);
#if _EXTRAE_
                Extrae_event(PROGRAM, END);
#endif

        } 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
#if _EXTRAE_
                Extrae_event(PROGRAM, SORT);
#endif
                basicsort(n, data);
#if _EXTRAE_
                Extrae_event(PROGRAM, END);
#endif

        }
}
```

Figure 4: Codi de multisort-omp en versió tree.

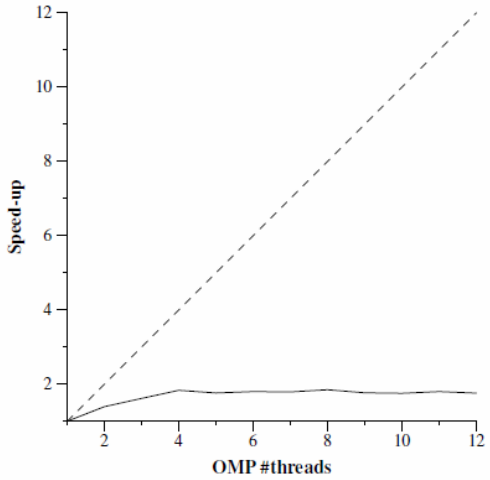
En els codis mostrats, tant per la versió leaf com per la versió tree, en el main, abans de fer la crida a multisort, s'ha de posar les següents linies de codi:

#pragma omp parallel
#pragma omp single

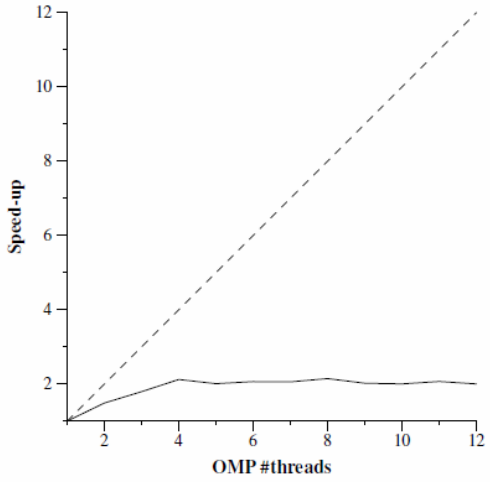
En la versió *leaf* es crea la tasca quan arribi al cas base, o sigui un cop s'hagi acabat la crida recursiva *multisort*.
Com que hi ha dependència entre la primera i segona crida a *merge* i les quatres funcions *multisort*, hem definit el primer *#pragma omp taskwait* just després de que acabin les funcions *multisort*, i com que l'ultim *merge* necessita que acabin els dos *merge* anteriors, hem posat un altre *#pragma omp taskwait* després del segon *merge*.
I per últim un *#pragma omp taskwait* espera que tots hagin acabat.

En la versió *tree* es crea les tasques a totes les crides a funcions ja siguin crides recursives o no, quan no es compleix el cas base.
El fet de crear les tasques ens obliga a posar o bé *taskwait* o bé *taskgroup*. Nosaltres ens hem decantat per la segona opció, tot i que ambdues opcions serien correctes i no és significatiu la diferencia de temps que es pugui produir entre un cas i l'altre.
Com deiem, hem hagut de posar taskgroup. El primer *taskgroup* el tenim perquè les dues primeres crides a *merge* te una dependència amb les crides a *multisort*. Per tant, hem d'assegurar que quan el programa arribi a les crides *merge* les dades que necesita ja estiguin calculades.
El mateix ens passa amb la última crida a *merge*. Aquesta depèn de les dues primeres crides a *merge*, per tant haurem d'englobar les dues primeres crides a *merge* en un *taskgroup*, per tal d'assegurar que quan s'executi la última crida a *merge* tingui les dades que necessita calculades.

4.2.2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

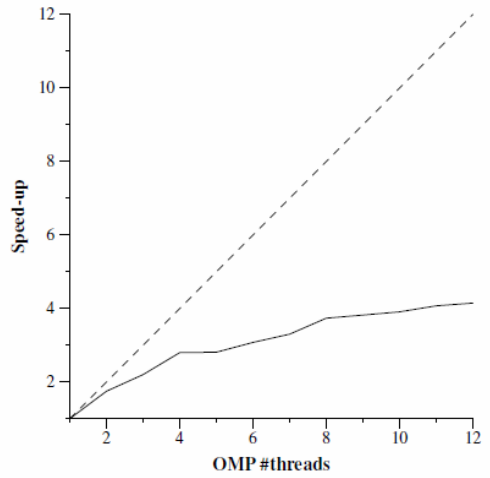


Speed-up wrt sequential time (complete application)

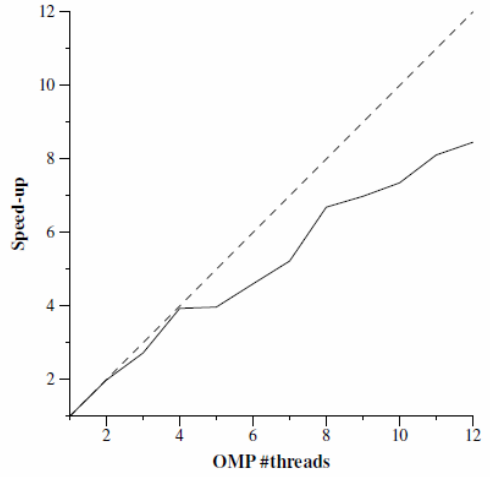


Speed-up wrt sequential time (multisort funtion only)

Figure 5: Plot de la versió leaf



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Figure 6: Plot de la versió tree

Veiem que la versió *tree* té un *speed-up* bastant millor que la versió *leaf*. La versió *leaf* quasi es manté el *speed-up*. En canvi, la versió *tree* té un increment substancial del *speed-up*.
Això és degut que la versió *leaf* només es crea la tasca quan arribi al cas base independenment de números de threads que hi ha, al contrari la versió *tree* es crea la tasca a mesura que es va fent les crides recursivament, la diferència es nota molt quan tenim més números de threads.
Com podem veure amb les imatges de les traces creades, en la versió *leaf* es confirma que només 4 threads tenen feina a fer en paral·lel . En canvi quan treballem en la versió *tree* tots els threads tenen feina a fer en paral·lel , fet que, com es natural, fa que es vegi repercutit en el *speed-up*.
Podem veure clarament en la figura 8 com s'executen en paral·lel com a màxim 4 threads

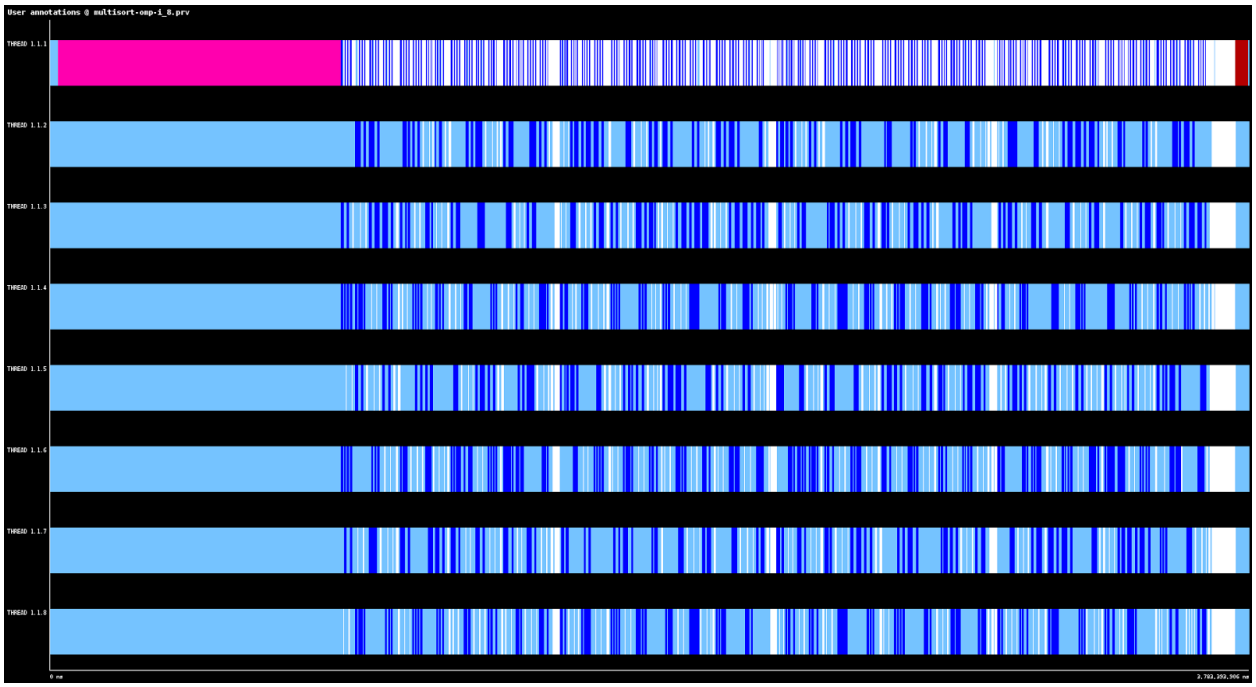


Figure 7: Traça de la versió *leaf*

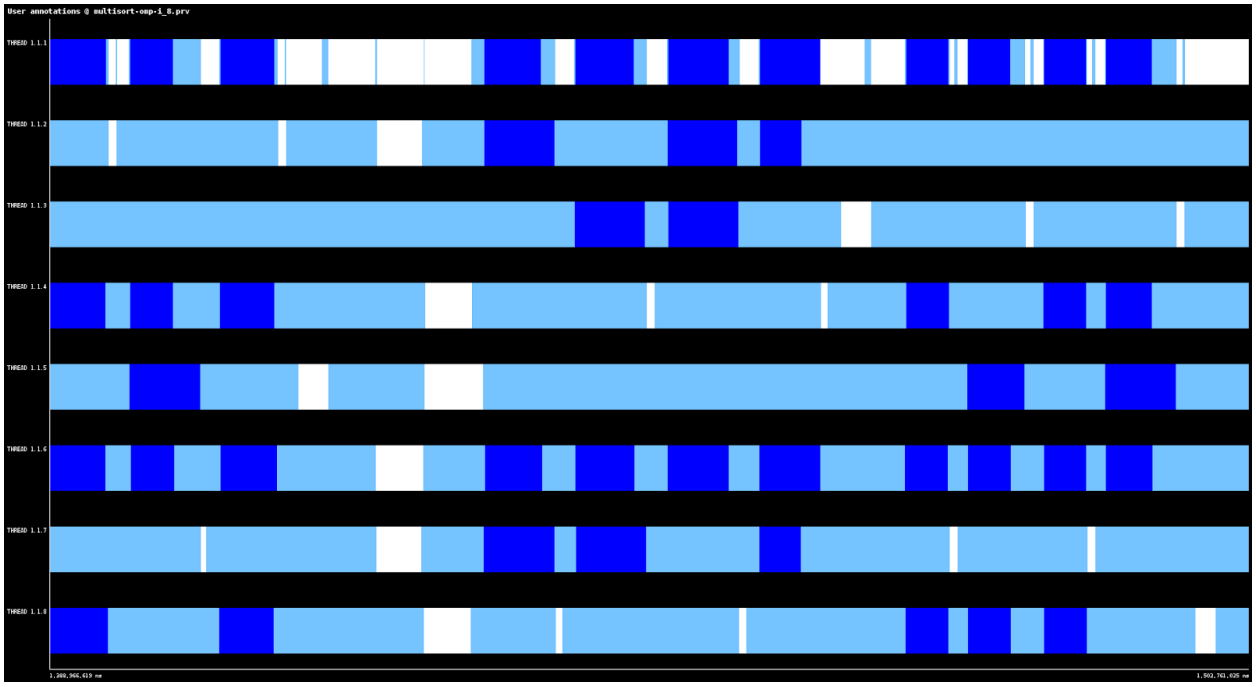


Figure 8: Traça de la versió *leaf* ampliada.

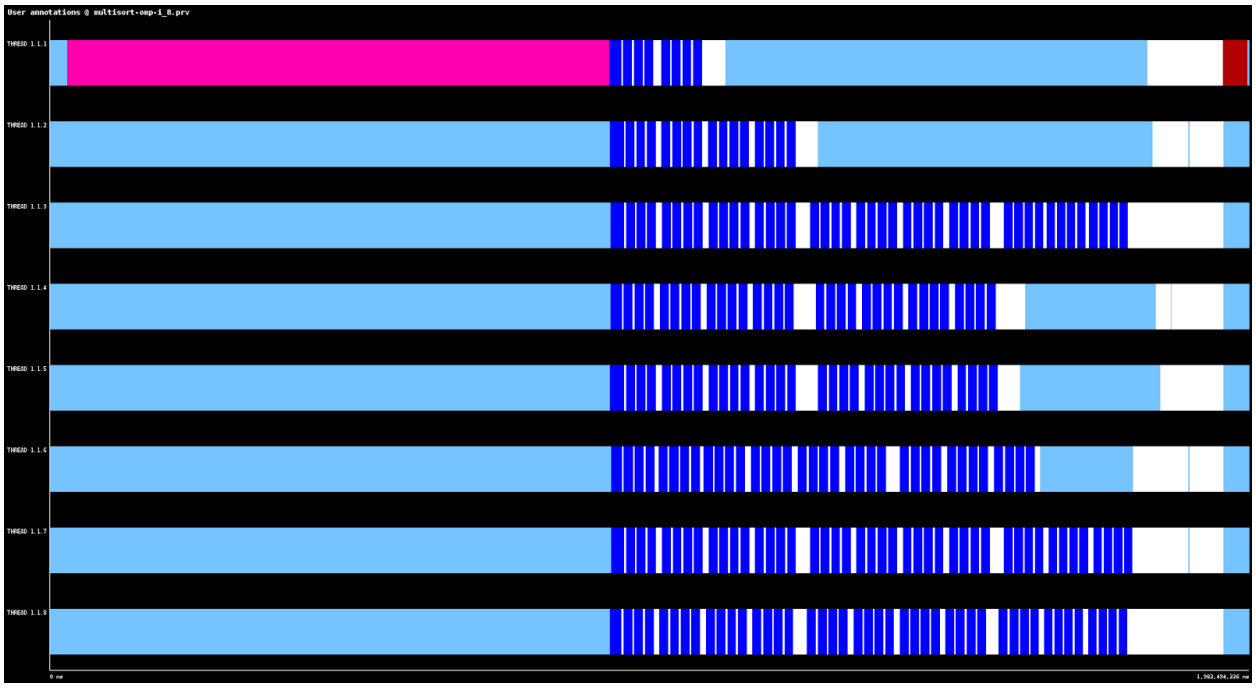


Figure 9: Traça de la versió *tree*

4.2.3. Analyze the influence of the recursivity depth in the Tree version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?

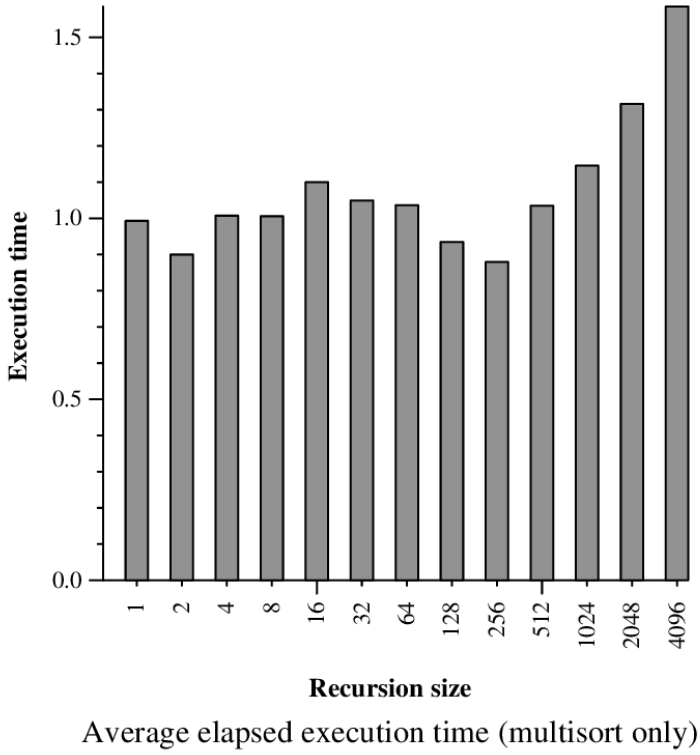


Figure 10: ***Plot de temps d'execucions en vers la mida***

El cas óptim és amb mida 2 i 256. Diem que aquests dos són óptims ja que el temps d'execució són els més baixos.

4.3.1. Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.

```
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
#if _EXTRAE_
                Extrae_event(PROGRAM, MERGE);
#endif
                basicmerge(n, left, right, result, start, length);
#if _EXTRAE_
                Extrae_event(PROGRAM, END);
#endif

        } 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 taskwait
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);


        } else {
        // Base case
#if _EXTRAE_
                Extrae_event(PROGRAM, SORT);
#endif
                basicsort(n, data);
#if _EXTRAE_
                Extrae_event(PROGRAM, END);
#endif

        }
}
```

Figure 11: Codi de la versió *tree* de *multisort-omp* usant depend

Gràcies al tareador i al seu anàlisi hem pogut veure quines dependències en generaven en cada moment. A traves de veure les variables que feien *load* i store hem vist que les crides a *multisort* generen una dependència a les dues primeres crides de *merge* sobre la variable *data*. Després podem veure que les dues primeres crides a *merge* necessiten una dada d'entrada (com ja hem dit es *data*) i generen unes dependències sobre la variables *tmp* que ho necessitará la tercera crida a *merge*.

4.3.2. Reason about the performance that is observed, including the speed–up plots that have been obtained different numbers of processors and with captures of Paraver windows to justify your reasoning.

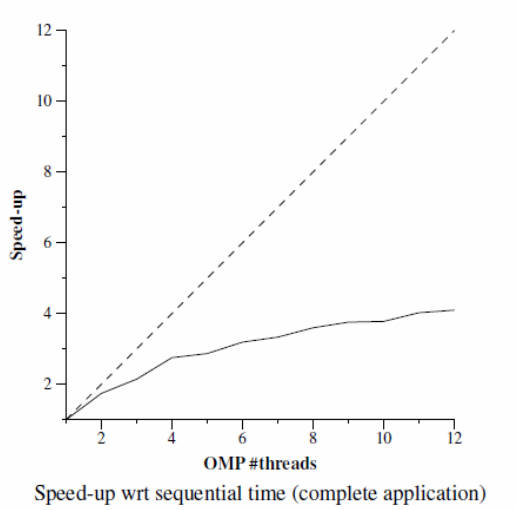
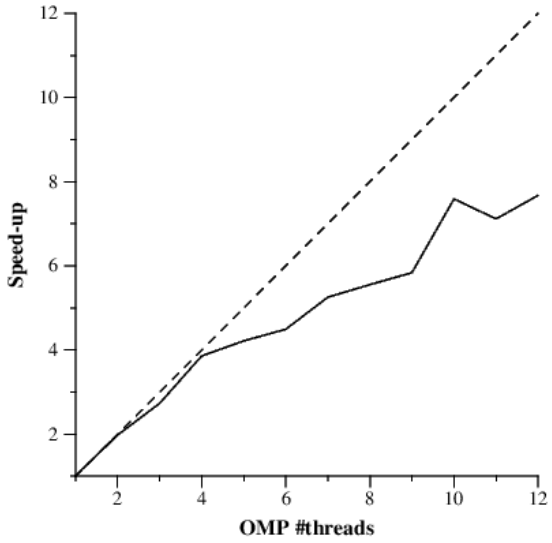


Figure 12: Plot de la versió tree usant depend

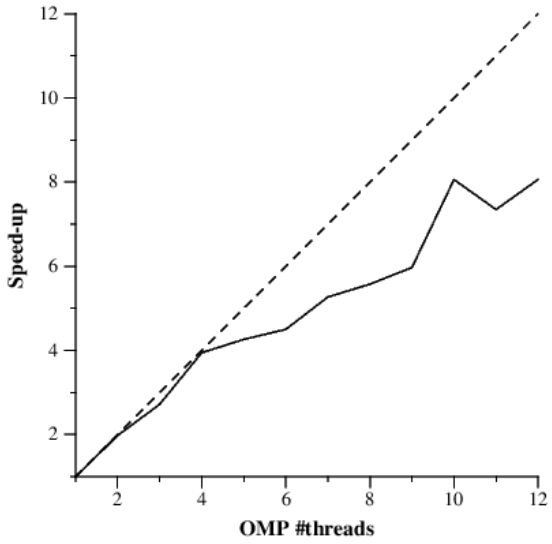
Veiem que el plot de *speed-up* generat pel codi amb el que hem utilitzat *depend* no millora gaire respecte el plot generat pel codi que hem fet servir els *taskgroup*. No millora gaire ja que realment en els moments importants ens hem d'esperar a que el contingut estigui preparat per poder tractar-lo.

OPTIONAL

1. If you have done any of the optional parts in this laboratory assignment, please include and comment in your report the relevant portions of the code and performance plots that have been obtained.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Figure 13: Plot de la versió tree paral·lelitzant les inicialitzacions

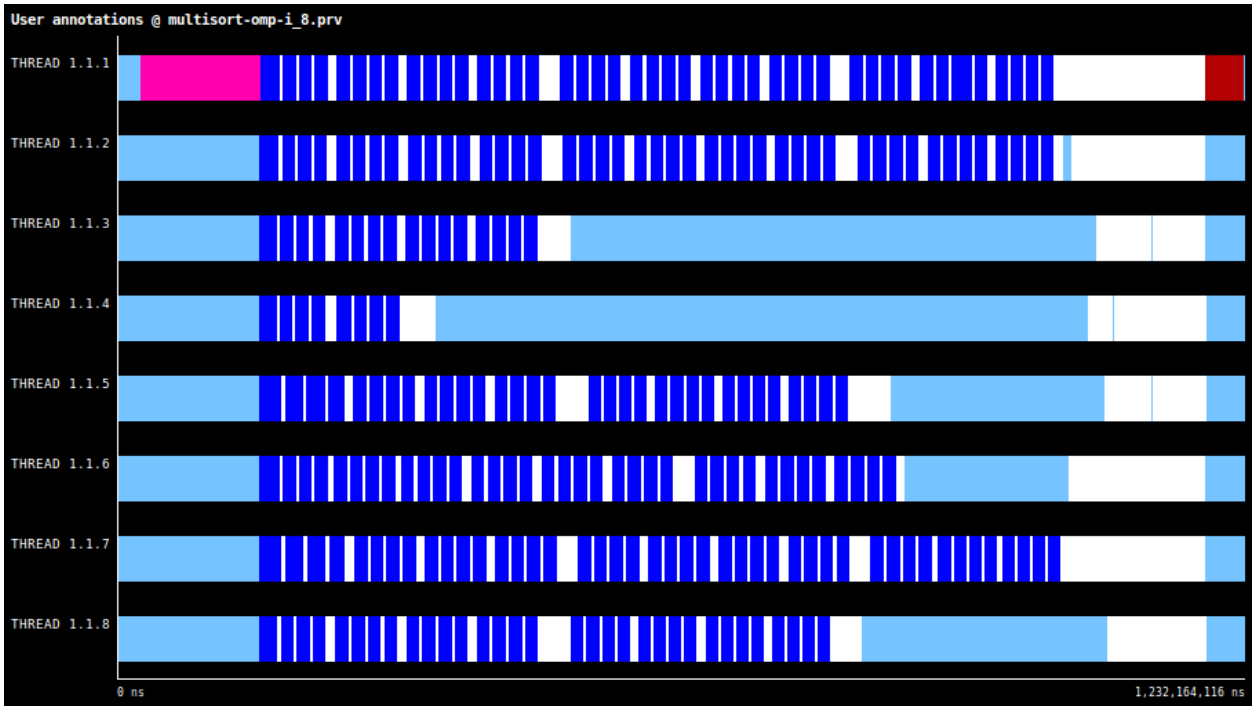


Figure 14: Traça de la nova versio de multisort omp

Les imatges de les dues traces, tant la versió sense optimitzar les inicialitzacions dels vectors tmp i data, com la de la nova versió, no es veuen bé els temps, però eren:

Versió sense optimitzar inicialitzacions: 1.983.494.336 ns

Versió optimitzant les inicialitzacions: 1.232.164.116

Com podem apreciar tant al plot, com a la traça com al temps que ens propociona paraver, veiem que aplicant una simple linia de omp *#pragma omp parallel for* es redueix el temps en 1983494336 - 1232164116 = 751.300.220 ns. És a dir, un 0,75s, molt quan parlem en temps així de baixos.