PAR2110

4th Deliverable

Èric Casanovas

22/05/2018

Analysis with Tareador

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.

A la funció main abans de la crida al multisort (la funció que volem analitzar) activem el tareador amb *tareador_ON()* i després, al finalitzar el deactivem amb *tareador_OFF()*.

```
tareador_ON();
multisort(N, data, tmp);
tareador_OFF();
```

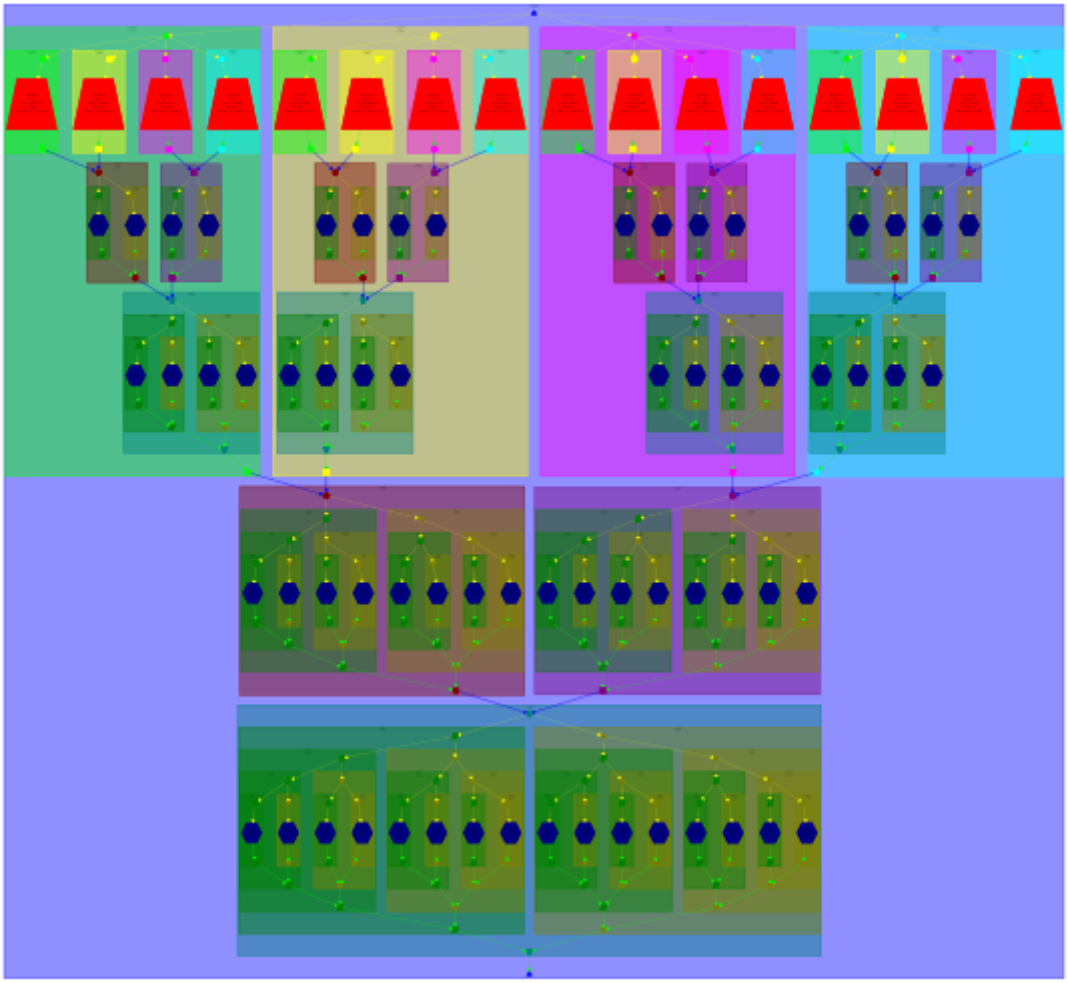
Dins la funció multisort, a cada crida a una funció (multisort, merge o basic sort) volem saber les dependències que tenen per tant colocarem abans de cada funció *tareador_start_task("nom tarea");* i després de la funció *tareador_end_task("nom tarea");*.

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("multisortmultisort1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("multisortmultisort1");
                tareador_start_task("multisortmultisort2");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("multisortmultisort2");
                tareador_start_task("multisortmultisort3");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("multisortmultisort3");
                tareador_start_task("multisortmultisort4");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("multisortmultisort4");

                tareador_start_task("multisortmerge1");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("multisortmerge1");
                tareador_start_task("multisortmerge2");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("multisortmerge2");

                tareador_start_task("multisortmerge3");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("multisortmerge3");
    } else {
        // Base case
        tareador_start_task("basicsort");
        basicsort(n, data);
        tareador_end_task("basicsort");
    }
}
```

També fem el mateix que a la funció multisort amb la funció merge, així podrem obserbar les dependencies d'aquesta funció també.

```
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
                tareador_start_task("mergemerge1");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("mergemerge1");
                tareador_start_task("mergemerge2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("mergemerge2");
        }
}
```

Després de fer tot això ja tenim totes les tasques genereades amb tareador, executarem a la línia de



comandes $> ./run_tareador.sh multisort-tareador i ens sortirà una finestra amb el TDG (Task Dependency Graph) en el que observem com al principi es un bloc (abans del tareador) però quan entrem a la funció multisort es va dividint en 4 les dades que tenim cada vegada que es crida a dita funció. Pel que fa la funció multisort observem que han d'acabar els multisorts per a començar a executar els merge, també observem que per executar l'últim merge caldrà executar els 2 anteriors i per últim que per acabar el multisort caldria esperar que acabi l'últim merge. Pel que la funció merge, observem un altra dependència que fa que s'hauran d'executar els 2 merge recursius abans de finalitzar la funció merge. Tot això es degut a que primer s'ha d'ordenar abans de ser "fusionat".

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.

Processadors	Temps execució	Speedup
1	20.334.421.001 ns	-
2	10.173.712.001 ns	1,9987
4	5.086.801.001 ns	2
8	2.550.377.001 ns	1,9945
16	1.289.899.001 ns	1,9772
32	1.289.850.001 ns	1
64	1.289.850.001 ns	1

Com podem observar a partir de 16 processadors no tenim un guany de rendiment. Això s'explica observant el TDG de l'apartat anterior, observem que màxim en un mateix nivell tenim 16 processos, això vol dir que amb 16 processadors tindrem suficient i que si n'afegim més aquests no faràn res ja que no tenen cap procés que puguin fer.

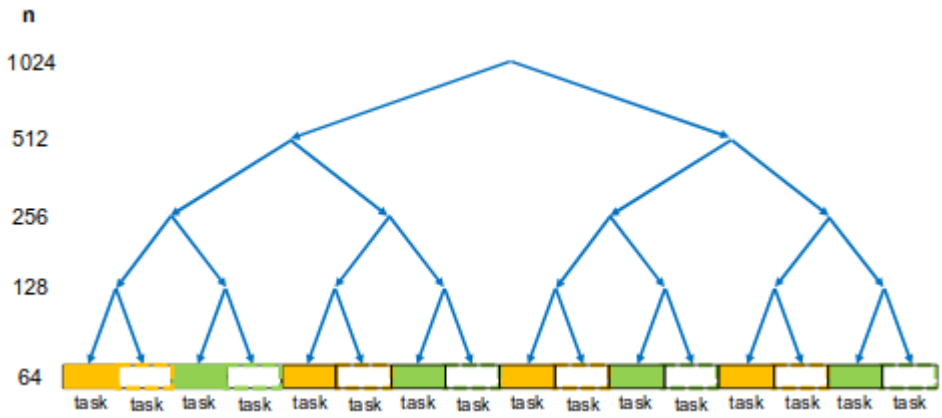
Parallelization and performance analysis with tasks

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

Aquest fragment de codi el tindran tant el leaf com el tree, es crea la regió paral·lela i fem que el multisort l'executi un unic thread.

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

Pel que fa el Leaf la estrategia consisteix en crear els processos a les fulles del arbre de dependencies, o sigui, quan la recursiva del divide-and-conquer acabi.

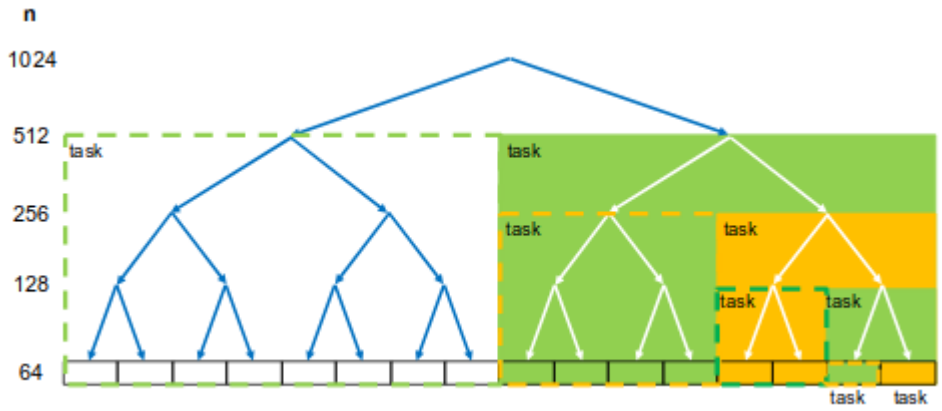


```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                multisort(n/4L, &data[0], &tmp[0]);
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                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
        #pragma omp task
        basicsort(n, data);
    }
}
```

```
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);
                #pragma omp taskwait
        }
}
```

Als 2 codis observem com creem tasques al acabar les recursives (dins els ifs que contenen les condicions de parada) i coloquem taskwaits per fer que es respectin les dependències que hem observat a la imatge del TDG.

D'altra banda l'estrategia tree consisteix en anar creant processos conforme anem cridant a les funcions recursives.

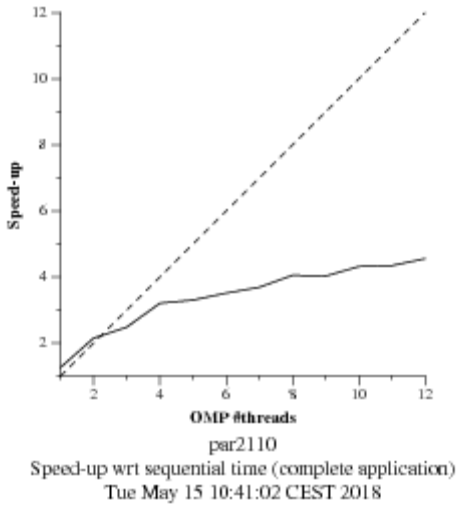


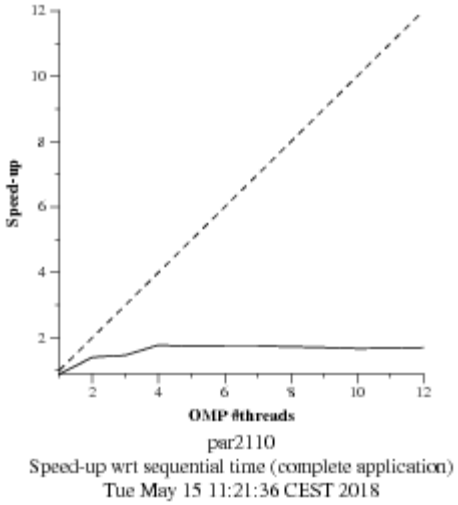
```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task
                multisort(n/4L, &data[0], &tmp[0]);
                #pragma omp task
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                #pragma omp task
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                #pragma omp task
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                #pragma omp taskwait

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

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

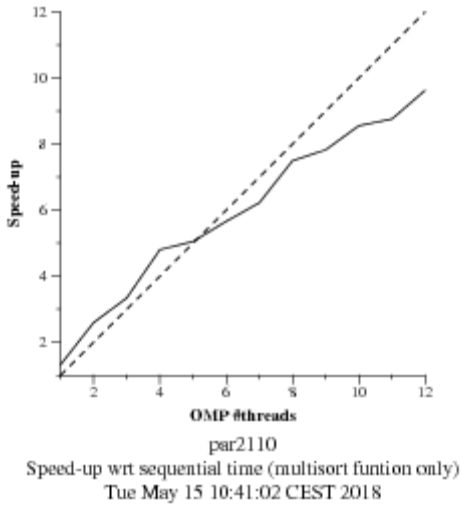
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
              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

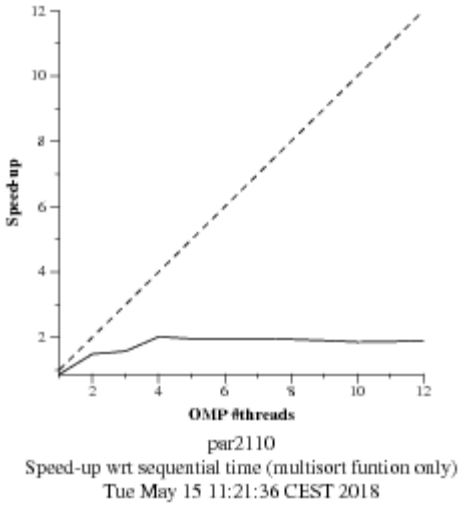
Observem als 2 codis com tenim els taskwait al mateix lloc que al leaf (per respectar les dependencies) i que anem creant tasques just abans de fer les crides recursives.

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.









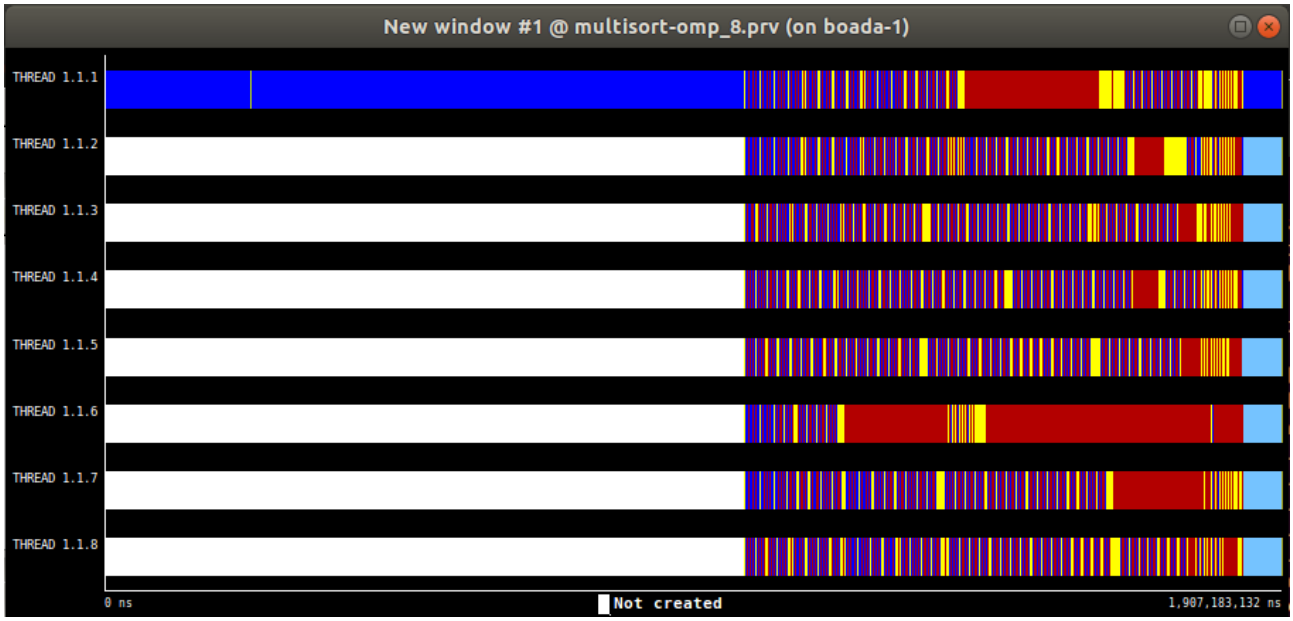
Speedup Tree

Speedup Leaf

Observant els grafics podem treure la conclusió de que l'estrategia de tree escala millor que la leaf en aquest cas, mirant el graf de sota del tree veiem que més o menys tenim un speedup òptim respecte els processadors (esta una mica per sota però es el normal), en canvi el de leaf veiem que quan arriba a 4 processadors, si n'afegim més no té cap efecte en el rendiment. Això s'explica perquè com a la funció multisort dividim les dades en 4 en el moment de arribar a les fulles tenim 4 processos a executar que que fins que no finalitzin no en podrem crear els següents 4. Es a dir, observant la figura del apartat 1 de la funció multisort del leaf strategy veiem que es faran els 4 multisorts i cada un creara una tasca al arribar al basicsort, fins un màxim de 4 i fins que no finalitzin no passaran el taskwait. En canvi pel que fa el tree trategy no tindrem aquest problema ja

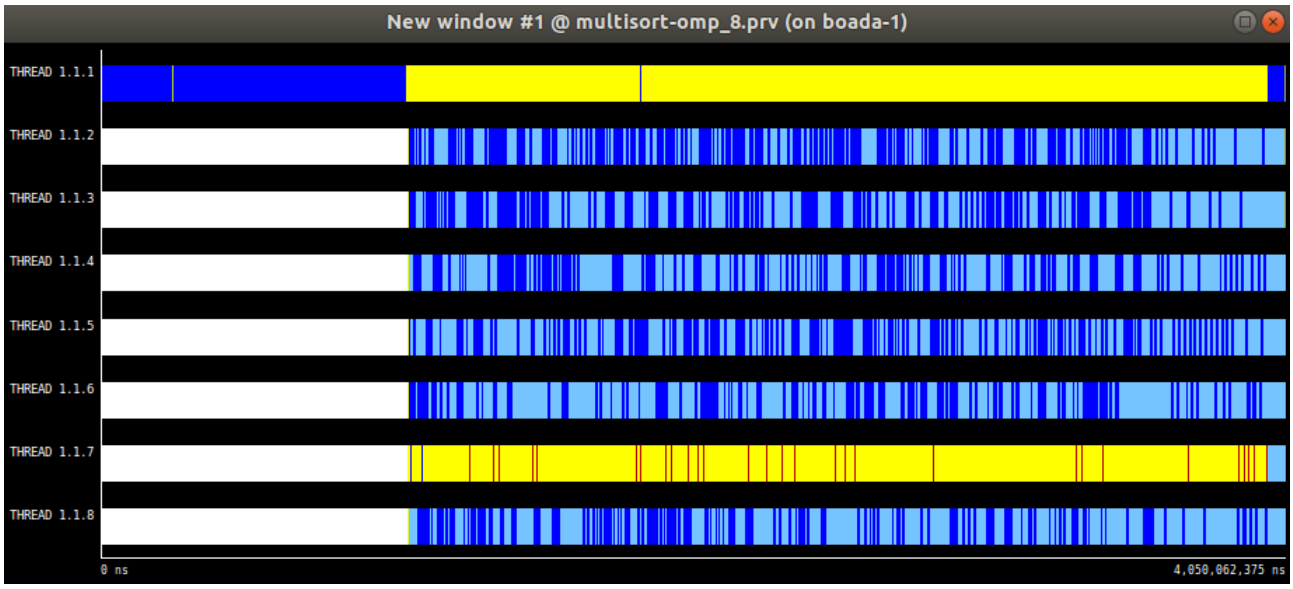
que anirem creant tasques que es quedaran en cua i com podem observar al TDG tret amb el tareador no tindrem una disminució de l'speedup fins als 16 processadors.

Pel que fa als grafics de sobre, cal destacar que no escala tant be el tree degut a que tota la part del porgrama que no es el multisort no esta paralelitzada i només l'executa un thread això fa que tinguem una part que es sempre constant en temps per molts processadors que tinguem. Al leaf no es nota tant degut a que si la part paralelitzable es bastant més gran però podem afegir pocs processadors no ho notarem però en el cas del tree al afegir molts processadors reduim molt la part paralelitzable i fa que es noti més la que no s'ha paralelitzat.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,679,368,375 ns	-	223,450,124 ns	4,136,561 ns	225,365 ns	2,707 ns
THREAD 1.1.2	741,714,945 ns	1,035,281,219 ns	61,428,860 ns	870,153 ns	198,893 ns	-
THREAD 1.1.3	749,344,915 ns	1,035,280,372 ns	53,203,156 ns	941,000 ns	208,927 ns	-
THREAD 1.1.4	754,891,691 ns	1,035,280,447 ns	47,506,043 ns	975,646 ns	200,686 ns	-
THREAD 1.1.5	746,757,395 ns	1,035,277,626 ns	54,880,832 ns	926,473 ns	196,707 ns	-
THREAD 1.1.6	227,995,955 ns	1,035,221,519 ns	577,855,297 ns	476,761 ns	77,223 ns	-
THREAD 1.1.7	650,014,910 ns	1,035,220,509 ns	154,718,735 ns	883,224 ns	164,495 ns	-
THREAD 1.1.8	767,336,358 ns	1,035,278,491 ns	33,032,276 ns	885,619 ns	179,878 ns	-
Total	6,317,424,544 ns	7,246,840,183 ns	1,206,075,323 ns	10,095,437 ns	1,452,174 ns	2,707 ns
Average	789,678,068 ns	1,035,262,883.29 ns	150,759,415.38 ns	1,261,929.62 ns	181,521.75 ns	2,707 ns
Maximum	1,679,368,375 ns	1,035,281,219 ns	577,855,297 ns	4,136,561 ns	225,365 ns	2,707 ns
Minimum	227,995,955 ns	1,035,220,509 ns	33,032,276 ns	476,761 ns	77,223 ns	2,707 ns
StDev	376,500,346.74 ns	26,506.25 ns	172,815,407.41 ns	1,096,357.44 ns	42,929.80 ns	0 ns
Avg/Max	0.47	1.00	0.26	0.31	0.81	1

Aquestes 2 imatges pertanyen a el codi amb tree strategy (amb 8 processadors) i ens permet observar la repartició de feina a cada thread. Observem que no hi ha cap thread que no faci feina. Si augmentessim els threads veuriem com es repartiria mes la feina entre els threads.



OpenMP Statistics @ multisort-omp_8.prv (on boada-1)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,908,999,376 ns	-	-	2,140,362,862 ns	697,292 ns	2,845 ns
THREAD 1.1.2	795,467,069 ns	1,043,857,010 ns	-	940 ns	681,544 ns	-
THREAD 1.1.3	768,427,960 ns	1,043,857,852 ns	-	903 ns	662,542 ns	-
THREAD 1.1.4	618,900,699 ns	1,043,859,305 ns	-	855 ns	578,682 ns	-
THREAD 1.1.5	652,527,419 ns	1,043,861,580 ns	-	897 ns	574,684 ns	-
THREAD 1.1.6	575,429,646 ns	1,043,780,311 ns	-	4,327 ns	615,353 ns	-
THREAD 1.1.7	2,563,375,970 ns	1,043,780,306 ns	362,693,475 ns	9,947,271 ns	1,898,252 ns	-
THREAD 1.1.8	515,001,990 ns	1,043,896,421 ns	-	753 ns	647,151 ns	-
Total	8,398,130,129 ns	7,306,892,785 ns	362,693,475 ns	2,150,318,808 ns	6,355,500 ns	2,845 ns
Average	1,049,766,266.12 ns	1,043,841,826.43 ns	362,693,475 ns	268,789,851 ns	794,437.50 ns	2,845 ns
Maximum	2,563,375,970 ns	1,043,896,421 ns	362,693,475 ns	2,140,362,862 ns	1,898,252 ns	2,845 ns
Minimum	515,001,990 ns	1,043,780,306 ns	362,693,475 ns	753 ns	574,684 ns	2,845 ns
StDev	709,535,684.07 ns	40,939.92 ns	0 ns	707,395,598.06 ns	419,313.68 ns	0 ns
Avg/Max	0.41	1.00	1	0.13	0.42	1

Aquestes 2 imatges pertanyen a el codi amb leaf strategy (amb 8 processadors) i ens permet observar la repartició de feina a cada thread. Observem que hi ha threads que no fan feina en certs moments (blau cel de la 1ª imatge) i també observem que a part del thread 1 hi ha un altre que fa molta mes feina que els demes (no te perquè ser el 7 es aleatori) això vol dir que aquest thread executa els processos creats al merge i això fa coll d'ampolla, i fa que els demes estiguin en idle mentre l'1 i el 7 treballen.

Parallelization and performance analysis with dependent tasks

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

Aquesta part per crear la regió paral·lela i fer que el multisort l'executi unicament un thread.

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

Pel que fa al multisort hem tret els *#pragma omp taskwait* degut a que ara ho hem de fer amb dependencies i hem colocat seguit del *#pragma omp task* la seva dependencia. Pels 4 multisort, la dependencia es que no es podra executar una tasca que faci servir data[0], data[n/4L], data[n/2L] i data[3L*n/4L] respectivament, això vol dir que com el 1r merge necesita la primera meitat de data, haura d'esperar a que acabin els 2 primers multisort, i pel segon els 2 últims multisort. Per aquesta rao, els multisort tenen un "out" i els merge un "in".

Pel que fa el darrer merge necessitara que s'hagin executat els 2 merge anteriors perquè necessitara tmp, per això els 2 merge anteriors tindran un "out" amb la part que facin de tmp i el darrer merge tindra un "in" amb demanant tmp.

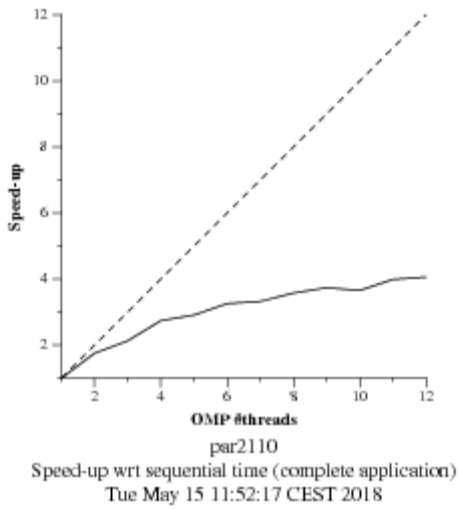
En resum quan una tasca amb una dependencia "out" acaba pot començar la tasca amb un "in".

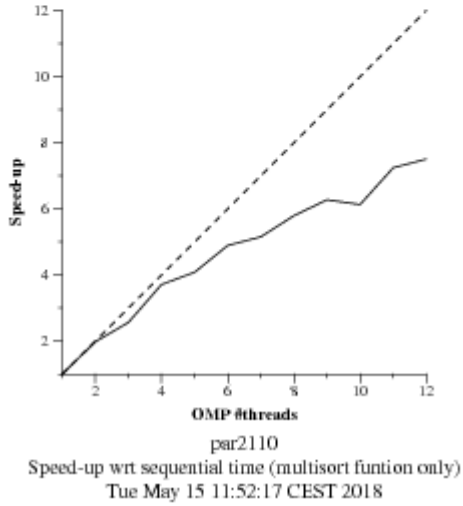
```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task depend (out: data[0])
                multisort(n/4L, &data[0], &tmp[0]);
                #pragma omp task depend (out: data[n/4L])
     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

```
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
        }
}
```

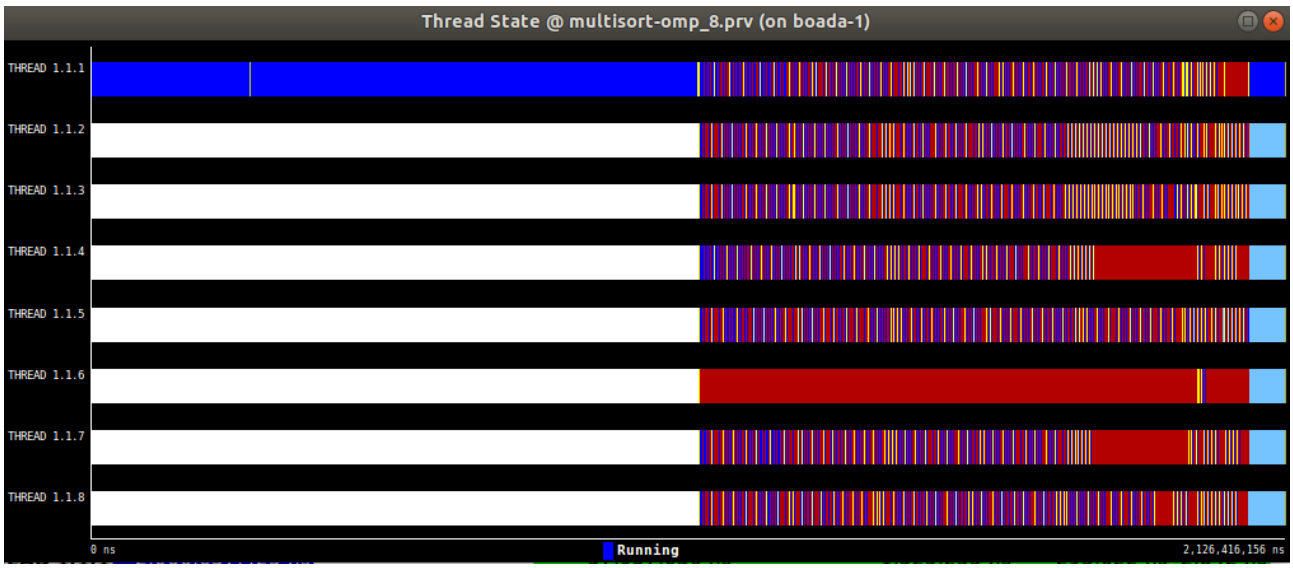
Pel que fa a la funcio merge no podem fer-la amb dependencies ja que no en té, l'únic que ha de fer es esperar a que acabin les 2 funcions recursives abans de continuar.

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.





Els grafs són força semblants als del tree strategy tot i que una mica per sota. S'esperava potser que fos una mica millor que amb el taskwait ja que permet que quan acabin els 2 primers (o últims) multisorts s'executi el 1r merge (o el 2^n), però al ser un programa força curt i la recursivitat no gaire llarga poden donar-se aquests casos de que estigui una mica per sota.

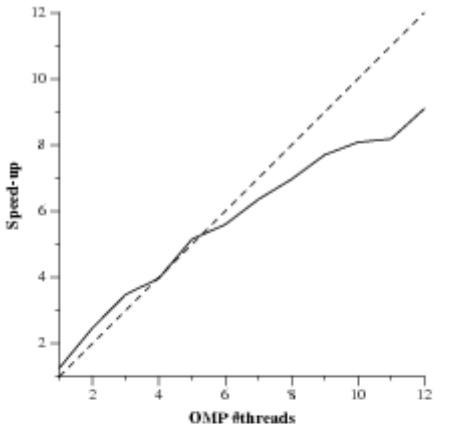


OpenMP Statistics @ multisort-omp_8.prv (on boada-1)

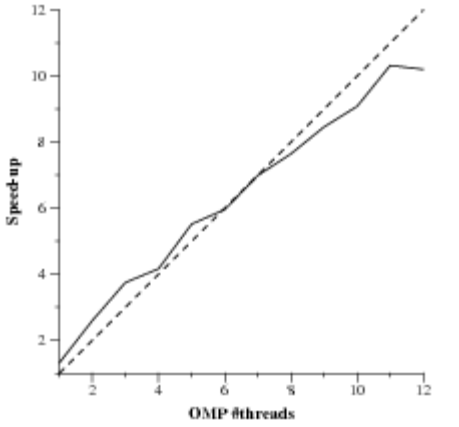
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	2,066,637,423 ns	-	57,957,036 ns	1,525,233 ns	293,622 ns	2,842 ns
THREAD 1.1.2	952,984,125 ns	1,080,463,404 ns	20,271,150 ns	1,039,362 ns	236,583 ns	-
THREAD 1.1.3	960,797,074 ns	1,080,465,521 ns	16,449,845 ns	1,018,297 ns	214,375 ns	-
THREAD 1.1.4	760,457,667 ns	1,080,466,446 ns	217,561,517 ns	793,807 ns	163,117 ns	-
THREAD 1.1.5	946,243,351 ns	1,080,425,645 ns	28,352,395 ns	831,761 ns	207,489 ns	-
THREAD 1.1.6	13,673,060 ns	1,080,381,085 ns	965,675,316 ns	188,583 ns	13,678 ns	-
THREAD 1.1.7	771,615,170 ns	1,080,381,080 ns	206,538,023 ns	831,440 ns	170,505 ns	-
THREAD 1.1.8	911,351,588 ns	1,080,409,139 ns	62,962,751 ns	808,181 ns	193,968 ns	-
Total	7,383,759,458 ns	7,562,992,320 ns	1,575,768,033 ns	7,036,664 ns	1,493,337 ns	2,842 ns
Average	922,969,932.25 ns	1,080,427,474.29 ns	196,971,004.12 ns	879,583 ns	186,667.12 ns	2,842 ns
Maximum	2,066,637,423 ns	1,080,466,446 ns	965,675,316 ns	1,525,233 ns	293,622 ns	2,842 ns
Minimum	13,673,060 ns	1,080,381,080 ns	16,449,845 ns	188,583 ns	13,678 ns	2,842 ns
StDev	522,867,456.20 ns	35,659.53 ns	300,197,408.82 ns	345,719.59 ns	75,768.22 ns	0 ns
Avg/Max	0.45	1.00	0.20	0.58	0.64	1

Optional

Optional 1: Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors. Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.

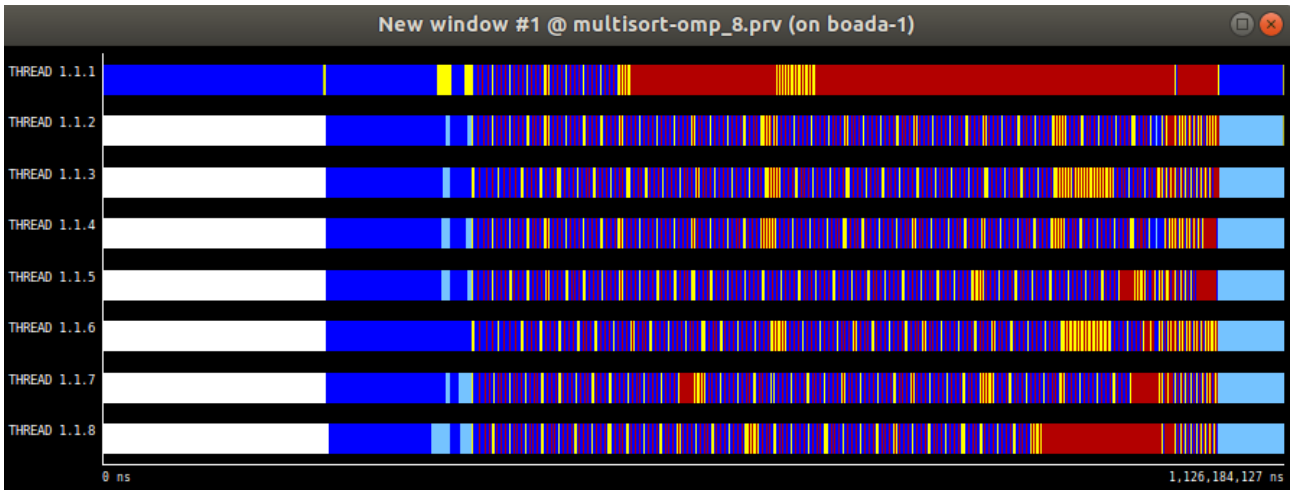


par2110
Speed-up wrt sequential time (complete application)
Mon May 21 18:34:12 CEST 2018



par2110
Speed-up wrt sequential time (multisort funtion only)
Mon May 21 18:34:12 CEST 2018

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

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



Observem com havent paralelitzat les 2 funcions initialize i clear observem una millora en el rendiment molt millor que unicment paralelitzant la funció multisort. Aixo es degut a que fent això hem reduït la part del codi no paralela (part blanca de la imatge del paraver) i hem augmentat la part paralelitzable (l'altra part de la imatge del paraver), aquest es el causant de la millora.

Conclusions

Com hem vist al llarg d'aquestes sessions hem observat la diferencia de speedup amb les diferents estrategies que hem fet servir (la leaf, tree i tree with dependences) en algoritmes de divide-and-conquer i observar les mancances i punts forts de cada estrategia. Tambe amb l'opcional hem observat com millora el programa al paralelitzar la part no paralelitzada del programa.