UNIVERSITAT POLITÈCNICA DE CATALUNYA

Paralelismo

Laboratorio 4: Divide and Conquer parallelism with OpenMP: Sorting

19/05/2022

Haopeng Lin Ye

Joan Sales de Marcos

ÍNDICE

1 - introducción

En este deliverable trabajaremos el análisis de programas recursivos con tareas. Hay dos formas principales para este análisis: *Leaf strategy* y *Tree Strategy*.

La primera consiste en identificar las tareas como las hojas del final del grafo de la recursión mientras que la segunda consiste en considerar como tarea cada llamada recursiva (provocando así un número mayor de tareas, ya que esta forma incluye también las hojas).

Leaf strategy: Tree strategy:

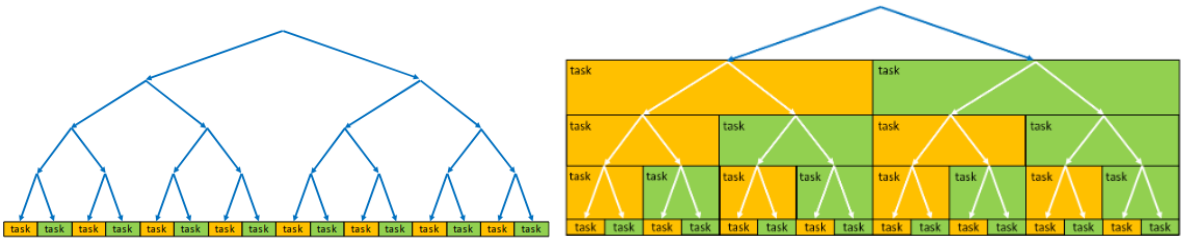


Figura 1.1: Esquemas de las diferentes estrategias

2 - Análisis de descomposición de tareas para Mergesort

Para esta sección se nos ha facilitado un código de Mergesort que ordena un array de elementos haciendo llamadas recursivas a unas funciones: 4 multisorts que dividen el trabajo en 4 partes de igual tamaño y 3 merges que juntan las divisiones de dos en dos, es decir, dos merges fusionan 4 divisiones para dar 2 y otro merge fusiona esas dos divisiones en una única siendo ese el resultado.

Dos ejecuciones del código básico (sin modificaciones) con 32768 elementos y 1024 de tamaño mínimo de sort y merge (`./multisort-seq -n 32768 -s 1024 -m 1024`) han dado las siguientes salidas:

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
************************************************************************
**********************
Initialization time in seconds: 0.857624
Multisort execution time: 6.322096
Check sorted data execution time: 0.015596
Multisort program finished
```

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
************************************************************************
************************
Initialization time in seconds: 0.859451
Multisort execution time: 6.322566
Check sorted data execution time: 0.015704
Multisort program finished
************************************************************************
************************
```

Como se puede ver el código tarda aproximadamente 6.32 segundos en ejecutarse. Esta medida la usaremos para comparaciones en futuras versiones.

2.1 - Análisis con Tareador

Como se ha explicado en la introducción, hay dos maneras principales de analizar algoritmos recursivos. Para ver las referencias de los cambios hechos en el programa para cada caso ver las funciones merge y multisort en los códigos "`multisort-tareador-leaf.c`" y "`multisort-tareador-tree.c`".

Primero analizaremos la estrategia de hojas considerando como tarea la llamada recursiva del caso base, el grafo de dependencias queda así:

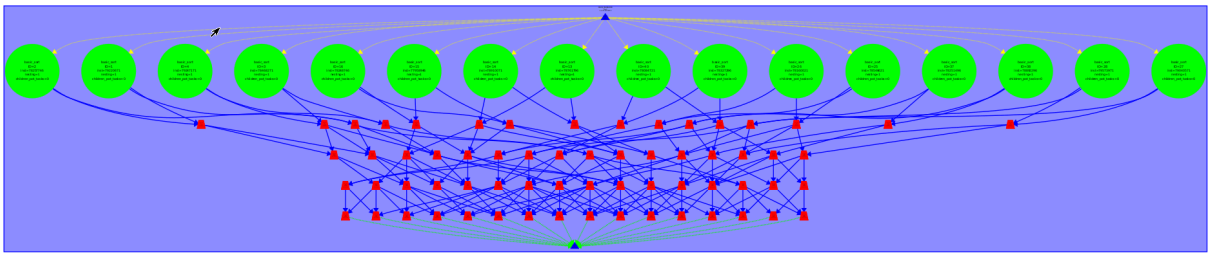


Figura 2.1: TDG de multisort-tareador-leaf.c

Se ve a simple vista que este TDG es bastante caótico. En total hay 16 tareas de multisort que hacen la ordenación y 16*4 = 64 tareas de merge que juntan los resultados. En esta estructura se observan 5 niveles de paralelismo divididos en estas secciones. Las tareas de los merges son más pequeñas en cuanto a peso respecto a las del sort. En el primer nivel se observa que los basicsorts no tienen dependencias de nada, se pueden ejecutar perfectamente en paralelo. Sin embargo, para después fusionar los resultados se necesitan los susodichos, por tanto en ese caso sí que hay dependencias en los merge, como se puede apreciar en el grafo.

En la figura 2.2 se muestra el grafo de dependencias con la estrategia tree:

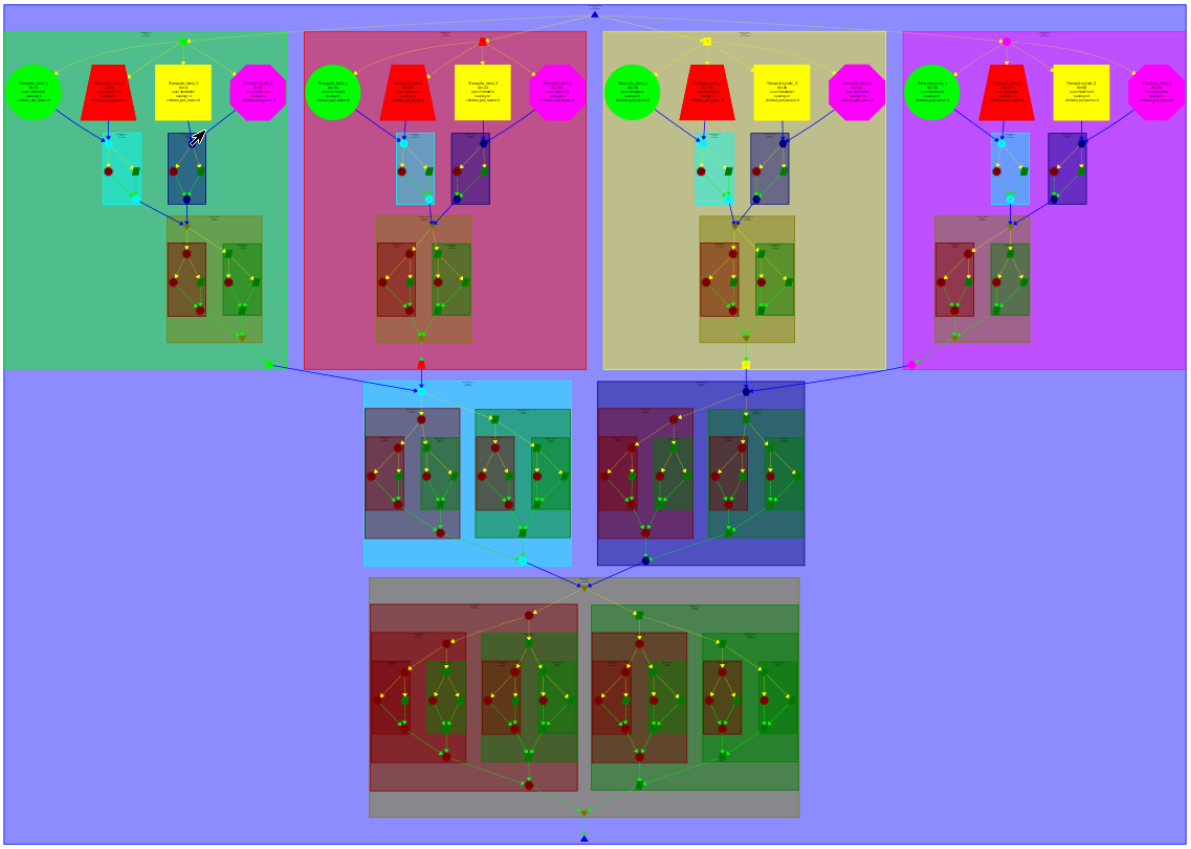


Figura 2.2: TDG de multisort-tareador-tree.c

En este TDG se ve más claramente la estructura de la explicación del código: las 4 llamadas recursivas a multisort que dividen el trabajo (por ejemplo las secciones rectangulares grandes de la mitad de arriba: verde, rojo, amarillo y rosa) y las 3 llamadas a merge que fusionan los resultados (por ejemplo las secciones rectangulares grandes de la mitad de abajo: azul claro, azul oscuro y marrón).

La diferencia principal con la estrategia leaf es que hay muchos más pisos en el grafo, indicando más dependencias y probablemente más tiempo de sincronización. Otra diferencia notable es que la estructura es mucho más clara y ordenada, una persona entendería mejor este TDG que el de leaf.

En cuanto a tamaño de tarea, el de la estrategia tree tiene menor peso en las tareas de merge, pero con el contrapunto de que hay una cantidad bastante mayor. Esto, como ya se ha dicho, puede influir negativamente en el tiempo de Overhead.

Para sincronizar las tareas se podría hacer un taskgroup en cada piso de la recursión o en cada llamada recursiva diferente (en este segundo caso, sería mejor hacer taskwait). Otra opción sería usar la cláusula “final” o contar la profundidad en la estrategia tree para hacer cutoff, pero esto será explorado más adelante.

En las figuras 2.2, 2.3, 2.4 y 2.5 se pueden ver las simulaciones de tareas trabajando con diferente números de procesadores en la estrategia tree:

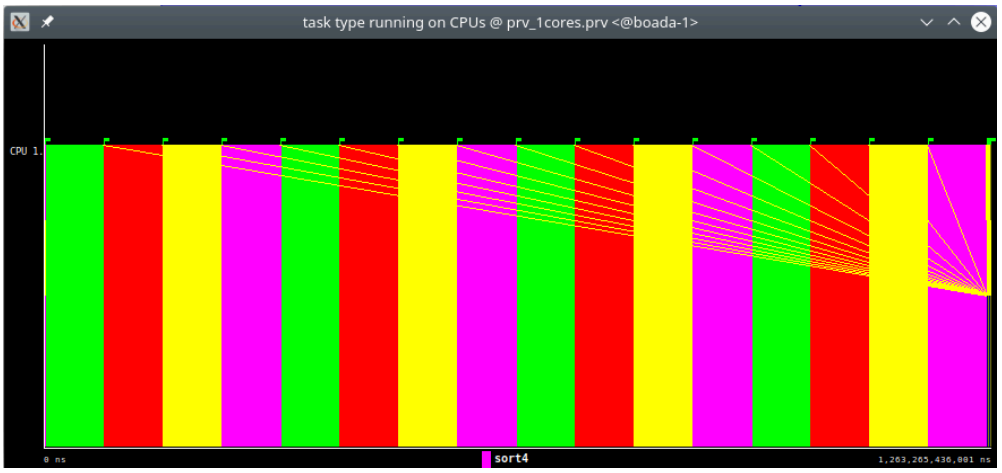


Figura 2.2: Simulación con 1 procesador (Tree strategy)

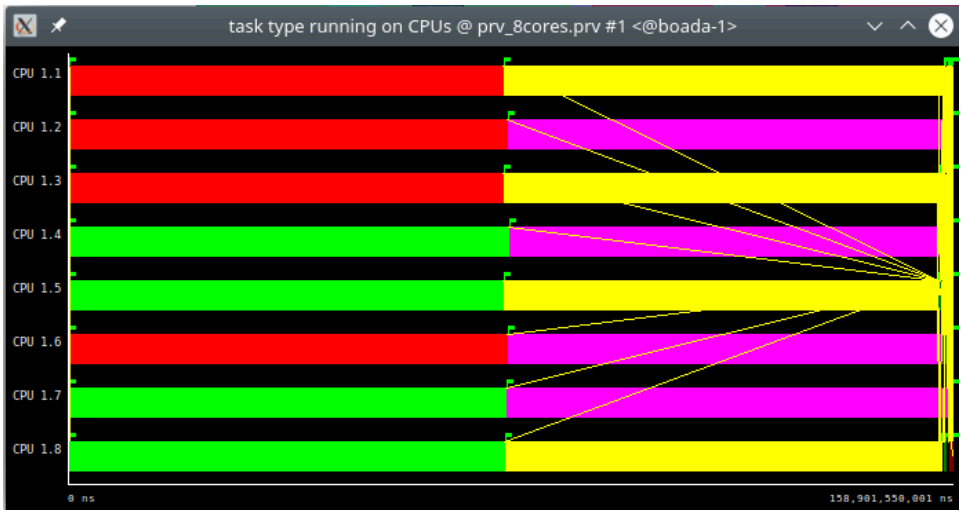


Figura 2.3: Simulación con 8 procesadores (Tree strategy)

Figura 2.4: Simulación con 16 procesadores (Tree strategy)

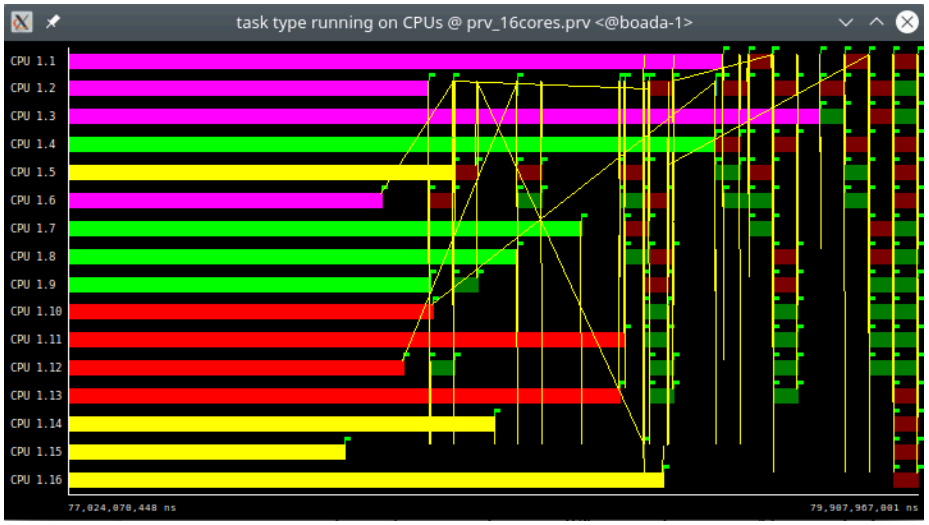


Figura 2.5: Simulación con 16 procesadores (Tree strategy) (Zoom en la parte del final)

De estas tres simulaciones podemos observar que la mejor en cuanto a tiempo de ejecución es la de dieciséis procesadores, por eso la hemos tomado como candidata para estudiarla. La compararemos con la misma gráfica de la estrategia leaf:

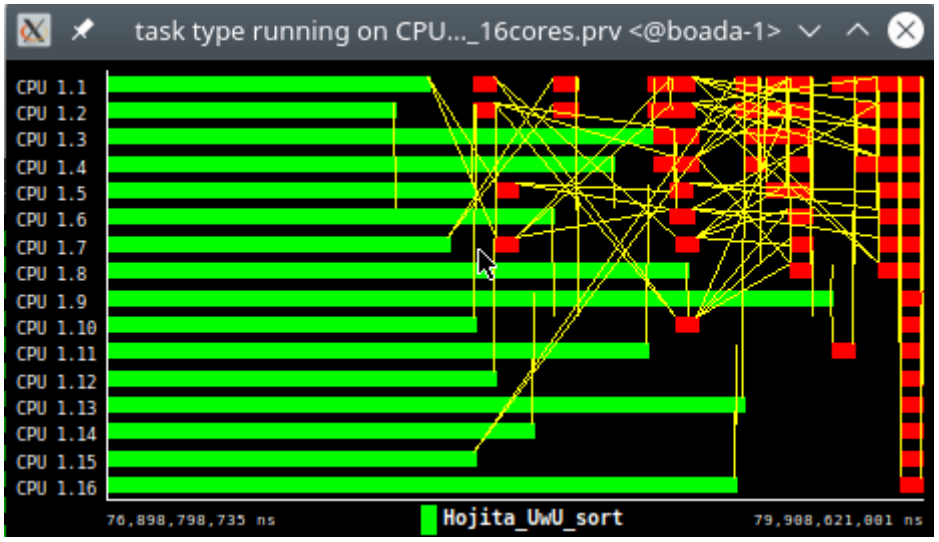


Figura 2.6: Simulación con 16 procesadores (Leaf strategy) (Zoom en la parte del final)

La diferencia más básica (a parte del orden de las tareas) son las líneas amarillas que marcan dependencia. En tree strategy, éstas son la mayoría casi verticales, mientras que en leaf strategy hay muchas más diagonales, indicando que hay mucho tiempo perdido por la cpu esperando la información que precede su tarea. En la estrategia tree esto no ocurre, así que en cuanto este aspecto se podría decir que es mejor.

3 - Paralelización con tareas OpenMP

En esta sección aplicaremos todo lo aprendido anteriormente con los análisis de tareador para encontrar el mejor modo de paralelizar el código de multisort.

Primero, al ejecutar el código base sin nada modificado y 2 threads (`sbatch ./submit-omp.sh multisort-omp 2`), se obtiene la siguiente salida:

```
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: 0.855307
Multisort execution time: 6.264999
Check sorted data execution time: 0.015247
Multisort program finished
```

Se puede ver que el tiempo de ejecución no cambia respecto a la primera ejecución de todas con multisort-seq. Vamos a arreglar eso.

3.1 - Aplicando leaf strategy a multisort-omp

Primero aplicaremos la estrategia de separación de tareas en hojas, como ya hemos visto anteriormente.

Los cambios que hemos hecho en el código están en el fichero multisort-omp-leaf.c.

En resumen, hemos definido como tareas las funciones “basicX” y usado taskwait para sincronizar los resultados de las funciones recursivas.

Una vez aplicados estos cambios, vamos a comprobar que la salida sea correcta. Ejecutaremos el programa con 2 y 4 threads:

```
************************************************************************
**********************
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: 0.855570
Multisort execution time: 3.763493
Check sorted data execution time: 0.016482
```

```
Multisort program finished
************************************************************************
************************
```

```
par2311@boada-1:~/lab4-sesion2$ cat
multisort-omp_4_boada-2.times.txt
************************************************************************
************************
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=4
************************************************************************
************************
Initialization time in seconds: 0.854988
Multisort execution time: 1.988477
Check sorted data execution time: 0.016577
Multisort program finished
************************************************************************
************************
```

Efectivamente, la salida es correcta.

El tiempo de ejecución con 2 threads es aproximadamente 3,76 segundos (casi la mitad que el código base) y con 4 threads el resultado es 1,98 segundos (más o menos un tercio respecto al código base).

Para un análisis más profundo observaremos las gráficas de la figura 3.1:

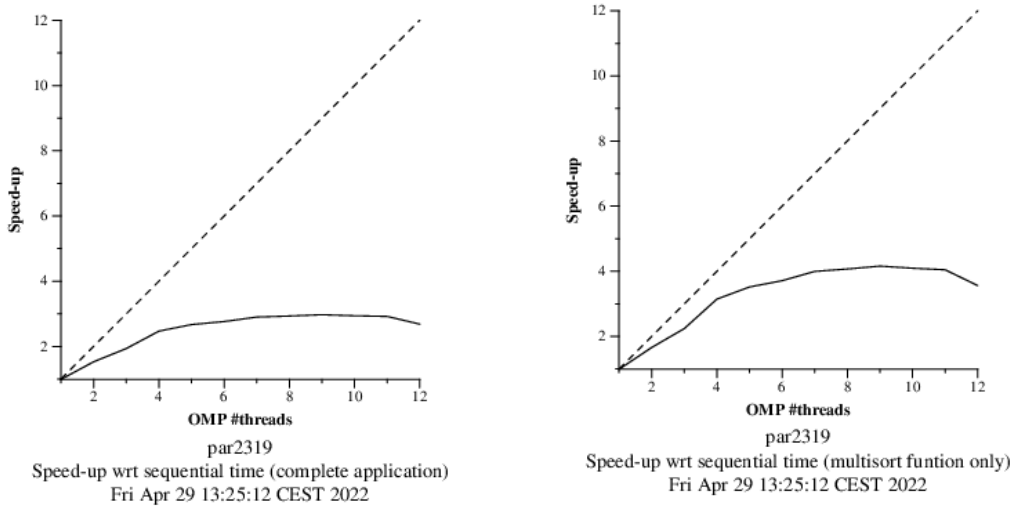


Figura 3.1: Gráficas del speedup de todo el programa y de solamente la parte recursiva según el número de threads (leaf strategy aplicada).

Esa curva que hacen los gráficos dice mucho de lo mala que es la escalabilidad de esta estrategia. No es muy deseable que cuantos más threads tenga el programa menos mejora se consiga.

Aún así, se puede concluir que con esta estrategia el número de threads óptimo está entre 6 y 8. A partir de aquí el programa no mejora del todo o simplemente empeora a causa del overhead.

Para saber por qué el programa no escala tan bien haremos un análisis con Paraver ejecutando en la terminal el script de submit-extrae.sh con ocho procesadores. En las siguientes figuras se muestran diversos datos de las cosas que hacen los threads:

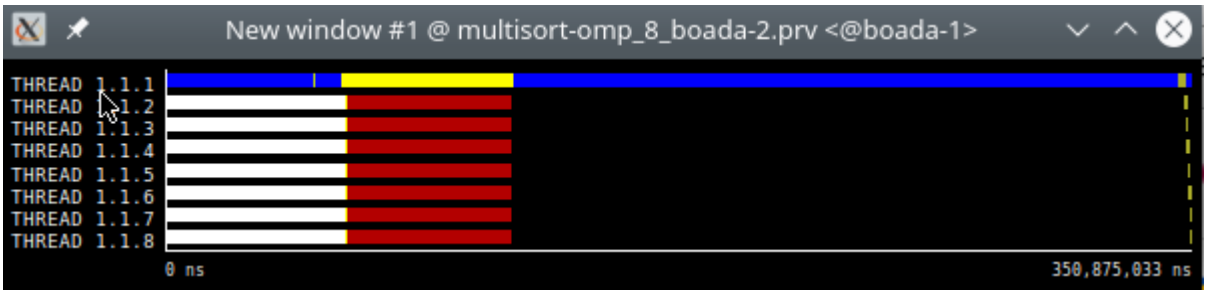


Figura 3.2: Timeline general de threads con leaf strategy

En la figura 3.2 se observa que hay una gran parte secuencial en el programa que no está paralelizada. Esta parte principalmente es la comprobación de resultados de la recursión. Esta parte no nos importa demasiado así que haremos zoom a la parte del multisort en las siguientes figuras.

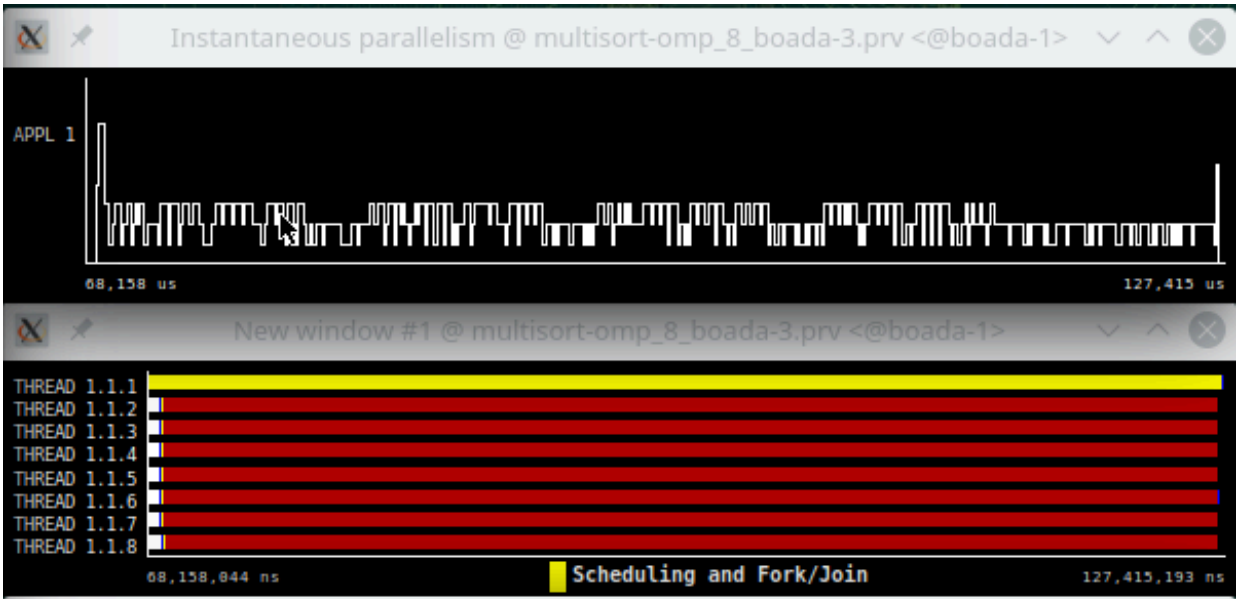


Figura 3.3: Timelines de asignación y sincronización de tareas, leaf strategy.

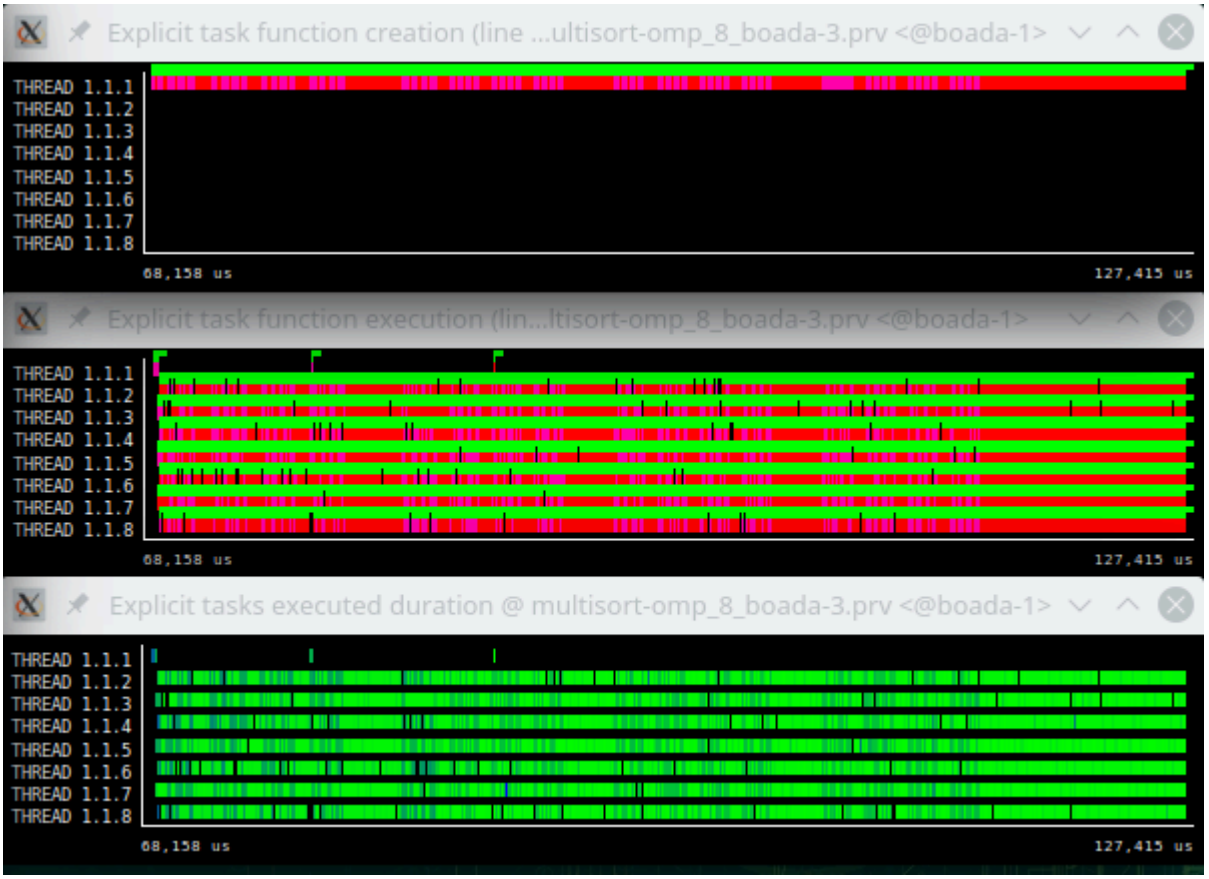
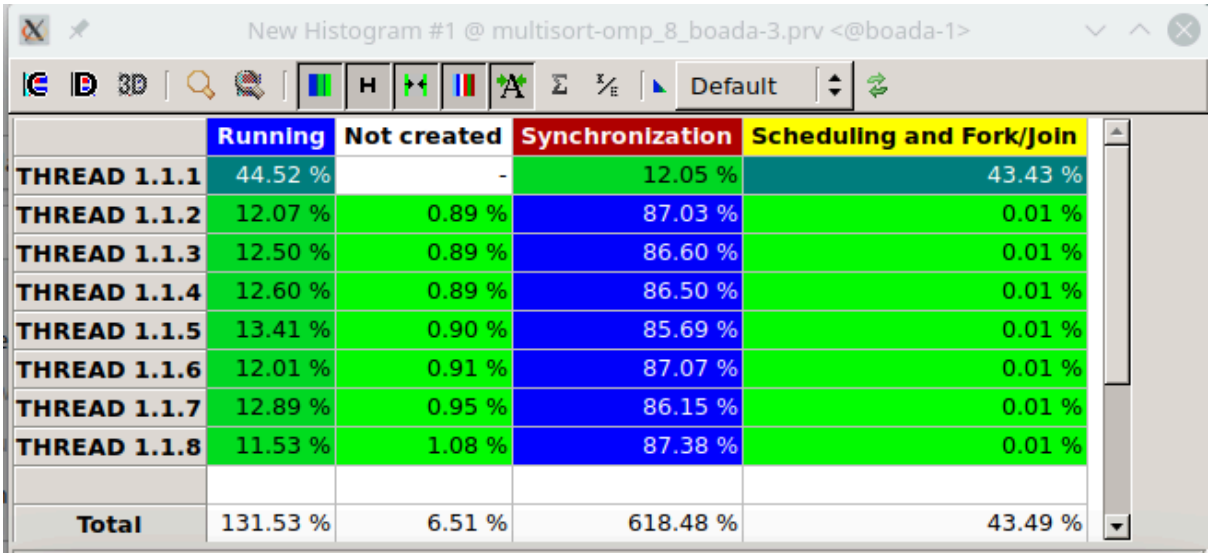


Figura 3.4: Timelines de creación y ejecución de tareas, leaf strategy.

En estas figuras se aprecia claramente que el thread número 1 es el encargado de recorrer todo el código para asignar las tareas a los demás. Para los datos más exactos mostraremos las tablas de las siguientes figuras:



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	44.52 %	-	12.05 %	43.43 %
THREAD 1.1.2	12.07 %	0.89 %	87.03 %	0.01 %
THREAD 1.1.3	12.50 %	0.89 %	86.60 %	0.01 %
THREAD 1.1.4	12.60 %	0.89 %	86.50 %	0.01 %
THREAD 1.1.5	13.41 %	0.90 %	85.69 %	0.01 %
THREAD 1.1.6	12.01 %	0.91 %	87.07 %	0.01 %
THREAD 1.1.7	12.89 %	0.95 %	86.15 %	0.01 %
THREAD 1.1.8	11.53 %	1.08 %	87.38 %	0.01 %
Total	131.53 %	6.51 %	618.48 %	43.49 %

Figura 3.5: Tabla con el porcentaje de tiempo empleado por cada thread en las diferentes funciones de trabajo, leaf strategy.

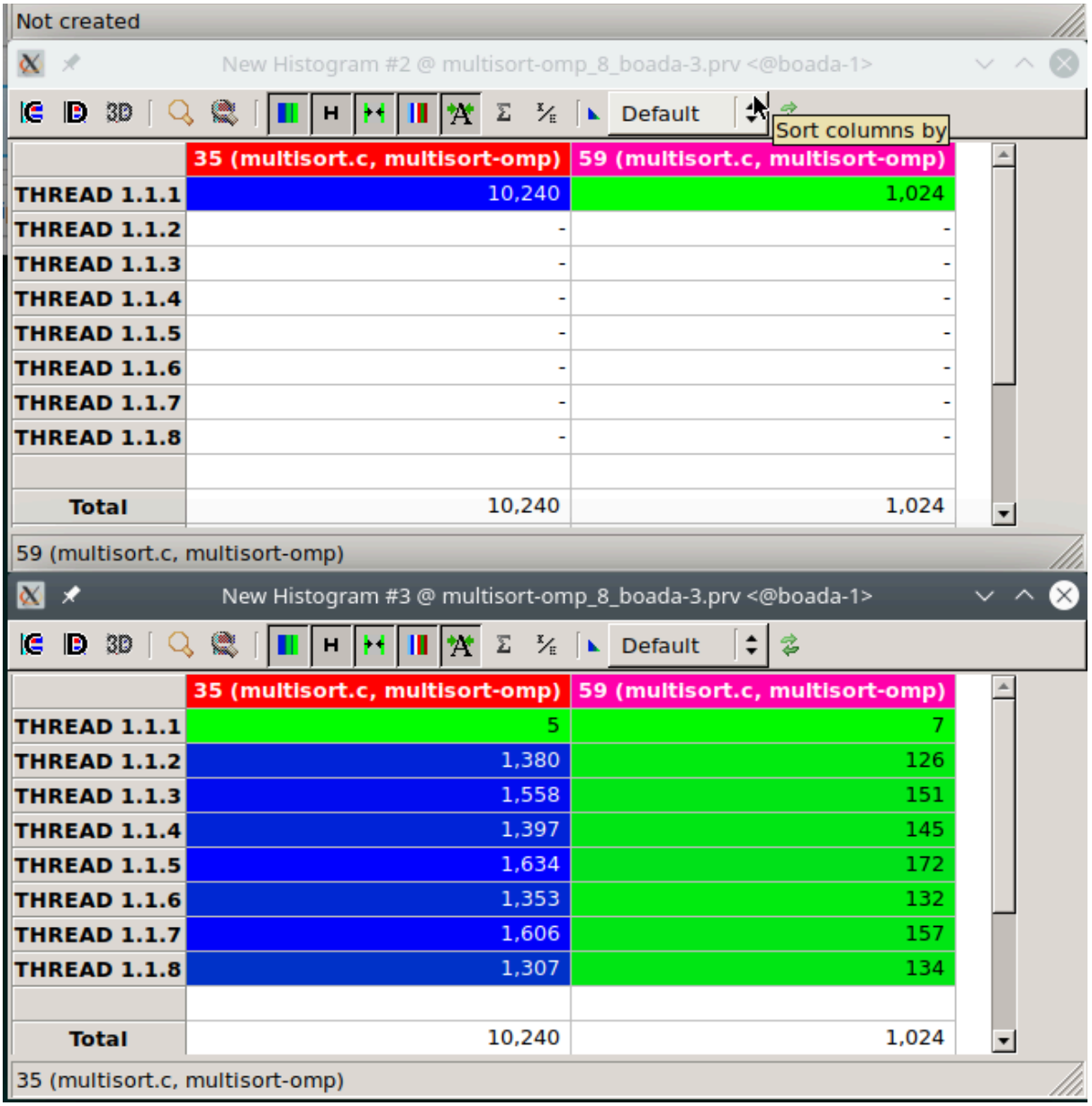


Figura 3.6: Tablas con el número de tareas asociadas a cada thread, leaf strategy (la primera de tareas de creación y la segunda de tareas de ejecución)

En la figura 3.5 se ve que el tiempo empleado por los threads para sincronizarse es muy elevado, probablemente por este motivo la escalabilidad del programa es tan mala. Además de que el thread 1 está haciendo mucha parte de la ejecución él solo (~40%).

En resumen: la estrategia de definición de tareas en hojas de la recursión provoca un tiempo de sincronización absurdo además de que un thread necesita recorrer el código entero y asignar tareas, impidiendo que durante ese tiempo pueda hacer trabajo propio del programa.

3.2 - Aplicando tree strategy a multisort-omp

En este apartado haremos lo mismo que el anterior pero aplicando la estrategia de asociación de tareas en cada llamada recursiva en lugar de en las “hojas”.

Los cambios hechos en el código están en el archivo multisort-omp-tree.c. Básicamente definimos las tareas en las llamadas a las funciones “merge” y “multisort”, añadimos los pragmas parallel y single antes de empezar la recursión y hacemos taskwait para sincronizar los resultados de las llamadas recursivas.

Tras ejecutar el código con las mismas características que la estrategia leaf la salida ha sido la siguiente:

```
par2311@boada-1:~/lab4-sesion2.3$ cat *txt
************************************************************
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: 0.857440
Multisort execution time: 3.569313
Check sorted data execution time: 0.017709
Multisort program finished
************************************************************
************************************************************
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=4
************************************************************
Initialization time in seconds: 0.856585
Multisort execution time: 1.823686
Check sorted data execution time: 0.017998
Multisort program finished
************************************************************
```

La salida es correcta.

El tiempo de ejecución con 2 threads es de aproximadamente 3,57 segundos, más o menos la mitad que el código sin modificaciones, y el resultado con 4 threads es de 1.82 segundos, menos que un tercio del código base. Una observación que se puede hacer es que estos tiempos son bastante similares a los de la estrategia leaf.

En la figura 3.7 se muestran las gráficas de speedup de esta versión:

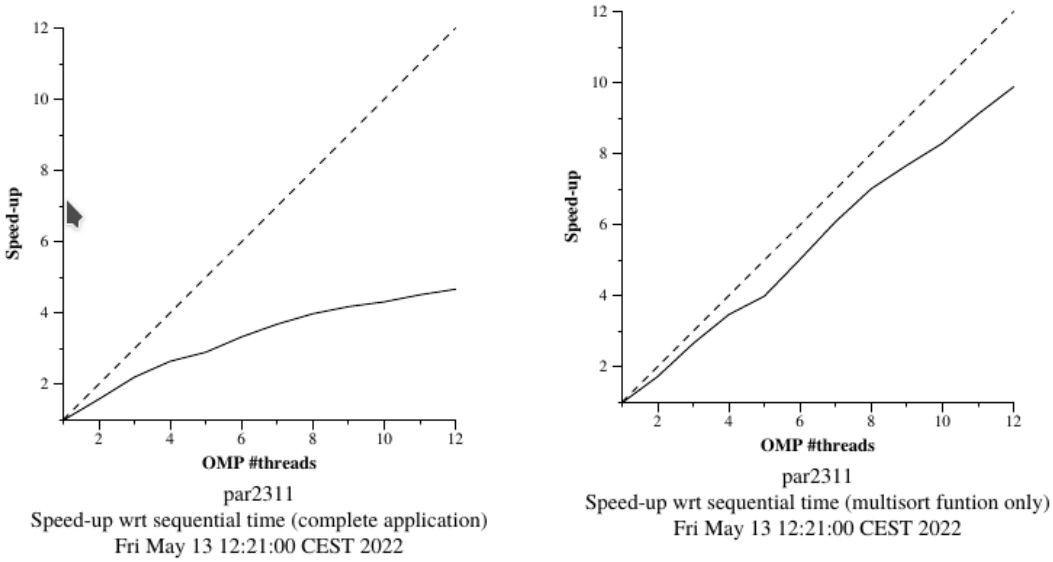


Figura 3.7: Gráficas del speedup de todo el programa y de solamente la parte recursiva según el número de threads (tree strategy aplicada).

Se puede ver que el speedup del programa entero no es del todo deseable mientras que el de solamente la parte de la función multisort casi sigue la diagonal. La no tan buena mejoría es debida a que no se ha podido paralelizar la parte del programa donde se inicializan las variables y se comprueban los resultados, así que por la ley de Amdahl esta mejoría no repercute tan enormemente como desearíamos en el cómputo global del programa.

Una vez más, haremos un análisis con Paraver para saber qué está ocurriendo exactamente:

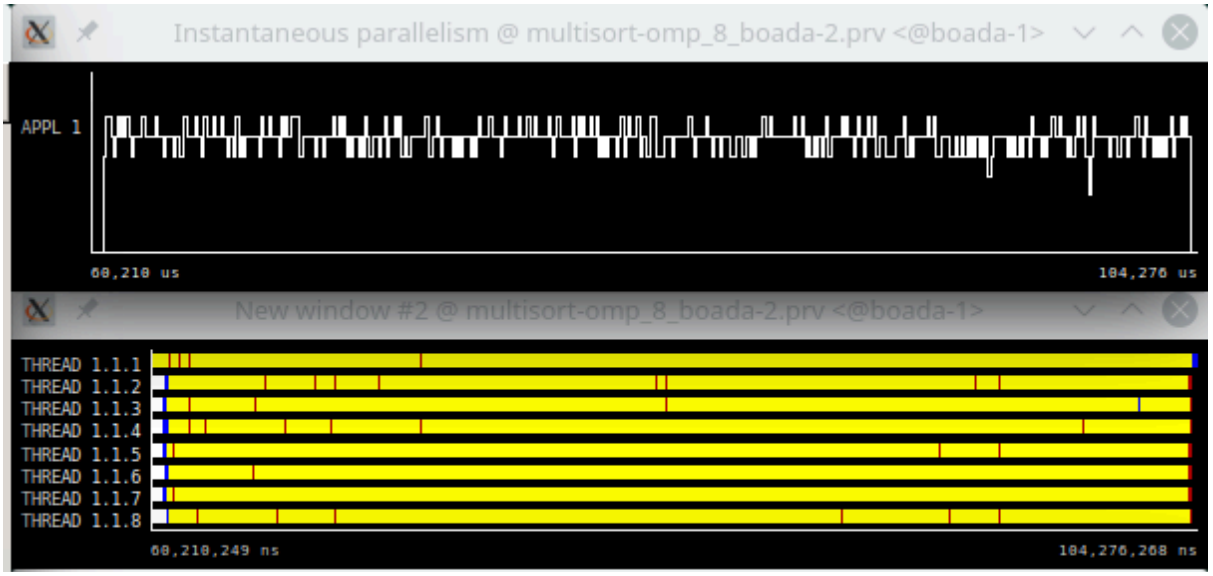


Figura 3.8: Timelines de asignación y sincronización de tareas, tree strategy.

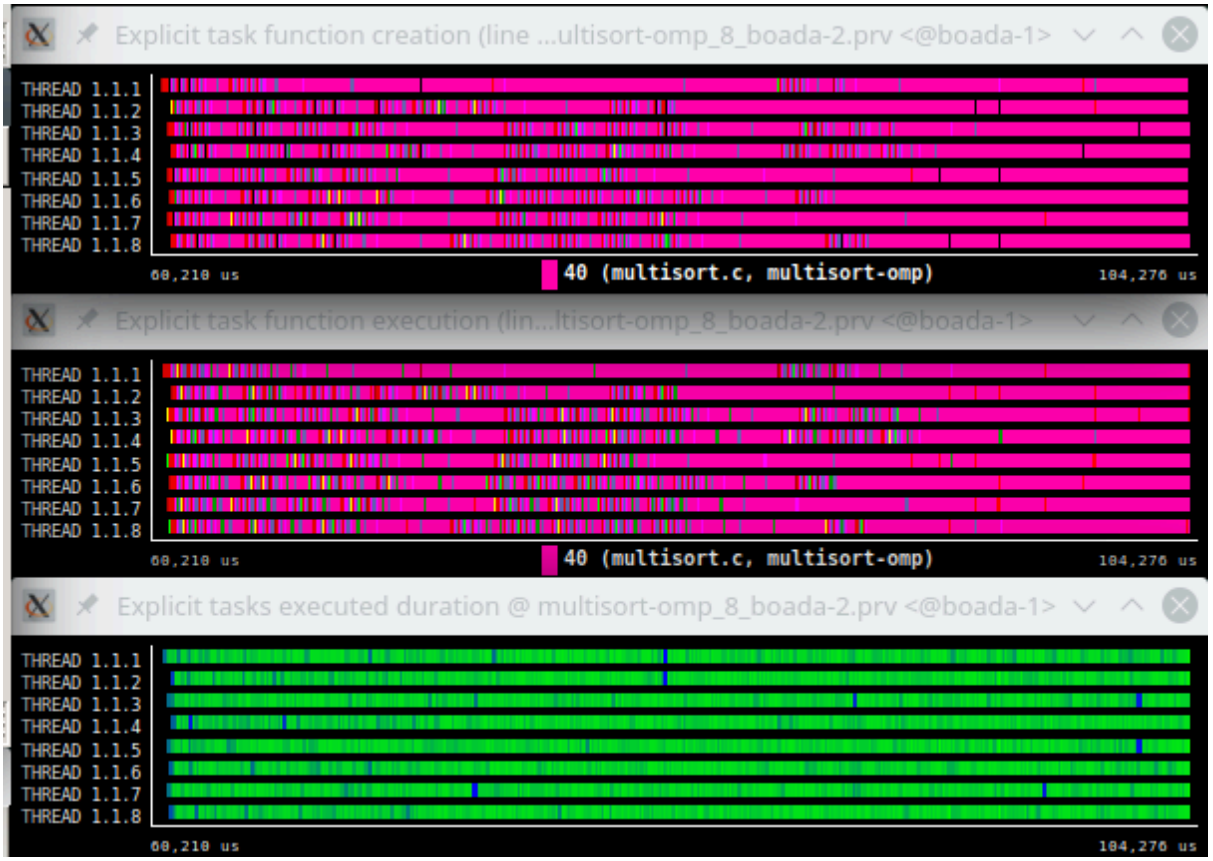
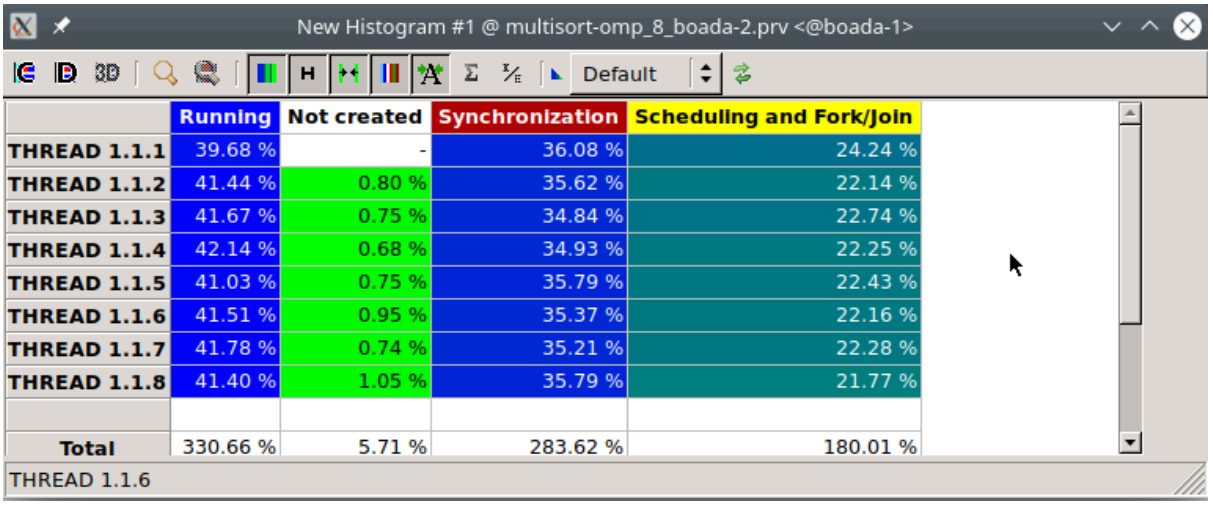


Figura 3.9: Timelines de creación y ejecución de tareas, tree strategy.

En este caso todos los threads hacen trabajo tanto de asignación de tareas como de ejecución, aprovechando mucho más el tiempo. Para datos más exactos tenemos las figuras 3.10 y 3.11:



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	39.68 %	-	36.08 %	24.24 %
THREAD 1.1.2	41.44 %	0.80 %	35.62 %	22.14 %
THREAD 1.1.3	41.67 %	0.75 %	34.84 %	22.74 %
THREAD 1.1.4	42.14 %	0.68 %	34.93 %	22.25 %
THREAD 1.1.5	41.03 %	0.75 %	35.79 %	22.43 %
THREAD 1.1.6	41.51 %	0.95 %	35.37 %	22.16 %
THREAD 1.1.7	41.78 %	0.74 %	35.21 %	22.28 %
THREAD 1.1.8	41.40 %	1.05 %	35.79 %	21.77 %
Total	330.66 %	5.71 %	283.62 %	180.01 %

Figura 3.10: Tabla con el porcentaje de tiempo empleado por cada thread en las diferentes funciones de trabajo, tree strategy.

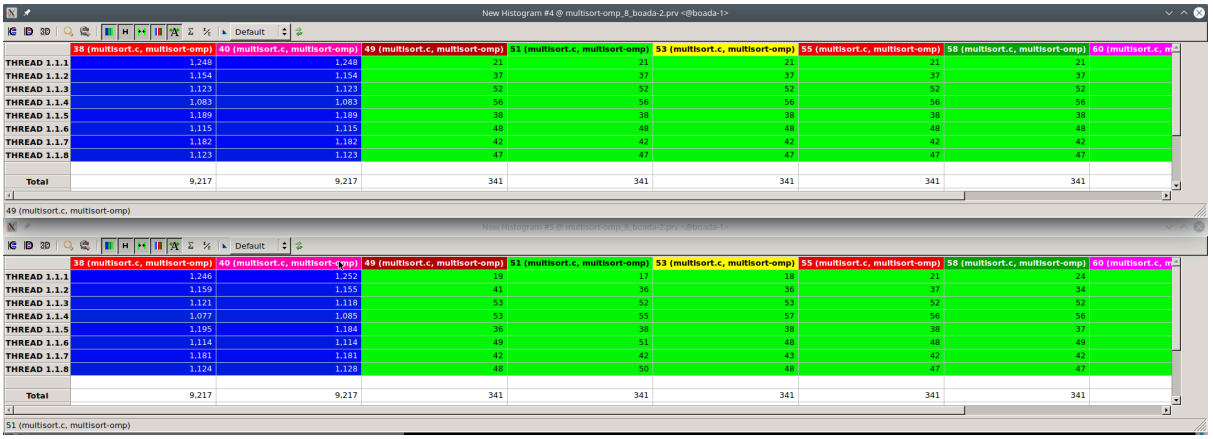


Figura 3.11: Tablas con el número de tareas asociadas a cada thread, leaf strategy (la primera de tareas de creación y la segunda de tareas de ejecución)

Se puede ver más específicamente en la figura 3.10 que todos los threads están un ~40% del tiempo haciendo trabajo, mucho más balanceado que la estrategia leaf.

Hay un total de 20.821 tareas. Este número es mucho mayor que en la otra estrategia ya que esta forma de crearlas incluye todas las llamadas recursivas. Aunque haya mejor balance de trabajo entre threads, el hecho de que existan más tareas podría hacer que exista también un mayor tiempo de sincronización, sin embargo, se puede ver claramente que los threads están un 35% del tiempo sincronizándose, a diferencia de la estrategia leaf, que estaban alrededor del 85%.

Para concluir: la estrategia de definición de tareas por ramas es mucho más balanceada que la de su subconjunto (hojas), haciendo que sea mejor paralelizable.

3.3 - Aplicando el mecanismo de control cut-off a multisort-omp

Para este apartado aplicaremos un mecanismo de cut-off a la estrategia tree para encontrar aún más formas de hacer el programa más eficiente y paralelizable.

Los cambios hechos en el código están en el archivo multisort-omp-tree-cutoff.c.

La salida con 2 y 4 threads, con cutoff a 0 y a 1 respectivamente es la siguiente:

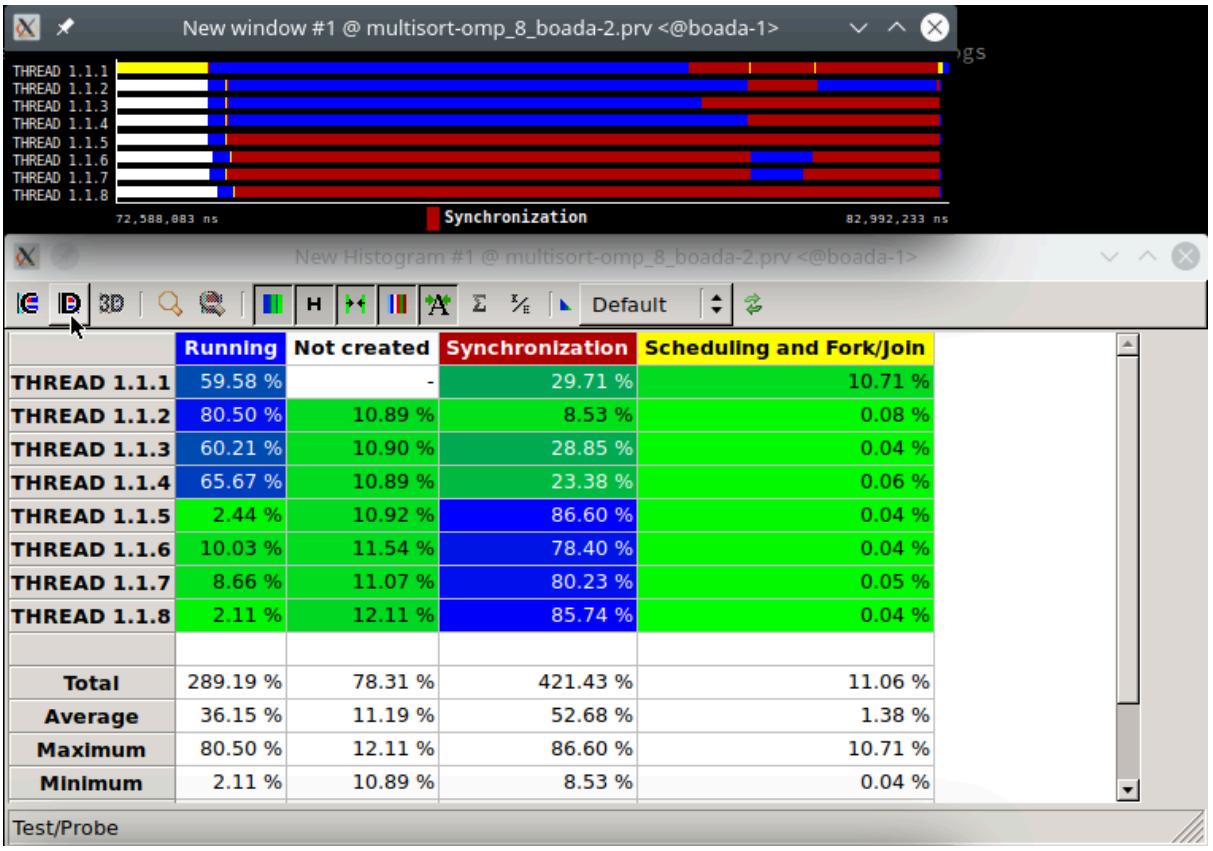
```
************************************************************************
**
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=0
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************
**
Initialization time in seconds: 0.856549
```

```
Multisort execution time: 3.531877
Check sorted data execution time: 0.017952
Multisort program finished
************************************************************************
**
************************************************************************
**
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                          CUTOFF=1
Number of threads in OpenMP:            OMP_NUM_THREADS=2
************************************************************************
**
Initialization time in seconds: 0.859314
Multisort execution time: 3.387583
Check sorted data execution time: 0.017999
Multisort program finished
************************************************************************
**
************************************************************************
**
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                          CUTOFF=0
Number of threads in OpenMP:            OMP_NUM_THREADS=4
************************************************************************
**
Initialization time in seconds: 0.858369
Multisort execution time: 1.998767
Check sorted data execution time: 0.017903
Multisort program finished
************************************************************************
**
************************************************************************
**
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                          CUTOFF=1
Number of threads in OpenMP:            OMP_NUM_THREADS=4
************************************************************************
**
Initialization time in seconds: 0.857141
Multisort execution time: 1.816958
Check sorted data execution time: 0.018009
Multisort program finished
************************************************************************
**
```

Las salidas son nuevamente correctas.

El tiempo de ejecución respecto a la estrategia tree base es el mismo cuando el cutoff está a cero, mientras que al usar un cut-off de 1 al parecer el programa tarda alrededor de una décima de segundo menos.

Una vez más, analizaremos con Paraver lo que está ocurriendo al usar 8 threads tanto con 0 como con 1 de cut-off:



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	59.58 %	-	29.71 %	10.71 %
THREAD 1.1.2	80.50 %	10.89 %	8.53 %	0.08 %
THREAD 1.1.3	60.21 %	10.90 %	28.85 %	0.04 %
THREAD 1.1.4	65.67 %	10.89 %	23.38 %	0.06 %
THREAD 1.1.5	2.44 %	10.92 %	86.60 %	0.04 %
THREAD 1.1.6	10.03 %	11.54 %	78.40 %	0.04 %
THREAD 1.1.7	8.66 %	11.07 %	80.23 %	0.05 %
THREAD 1.1.8	2.11 %	12.11 %	85.74 %	0.04 %
Total	289.19 %	78.31 %	421.43 %	11.06 %
Average	36.15 %	11.19 %	52.68 %	1.38 %
Maximum	80.50 %	12.11 %	86.60 %	10.71 %
Minimum	2.11 %	10.89 %	8.53 %	0.04 %

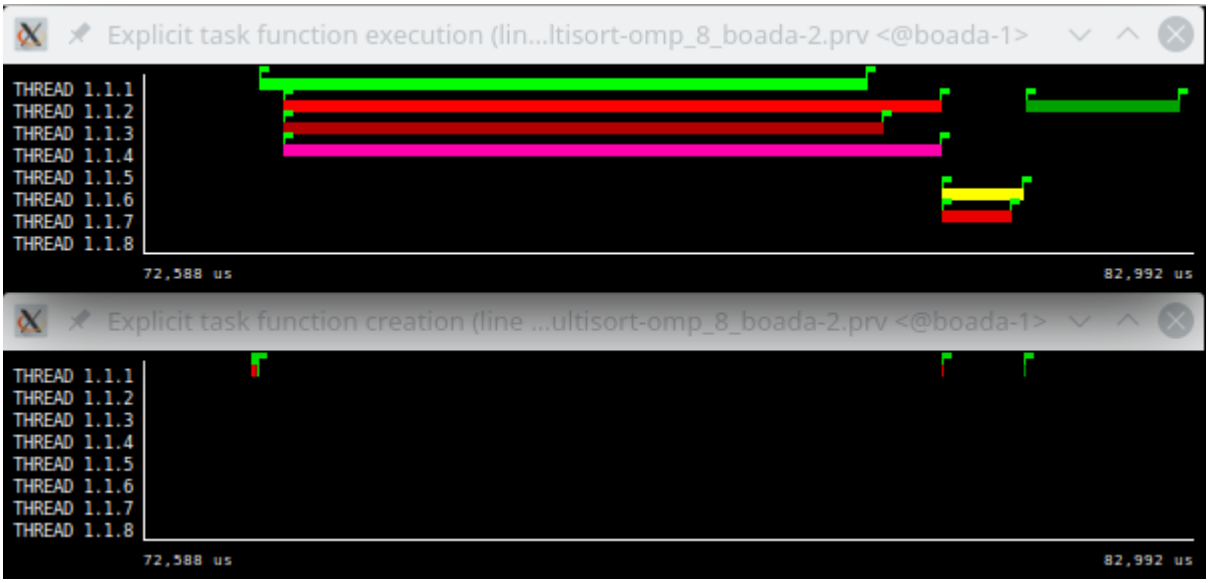
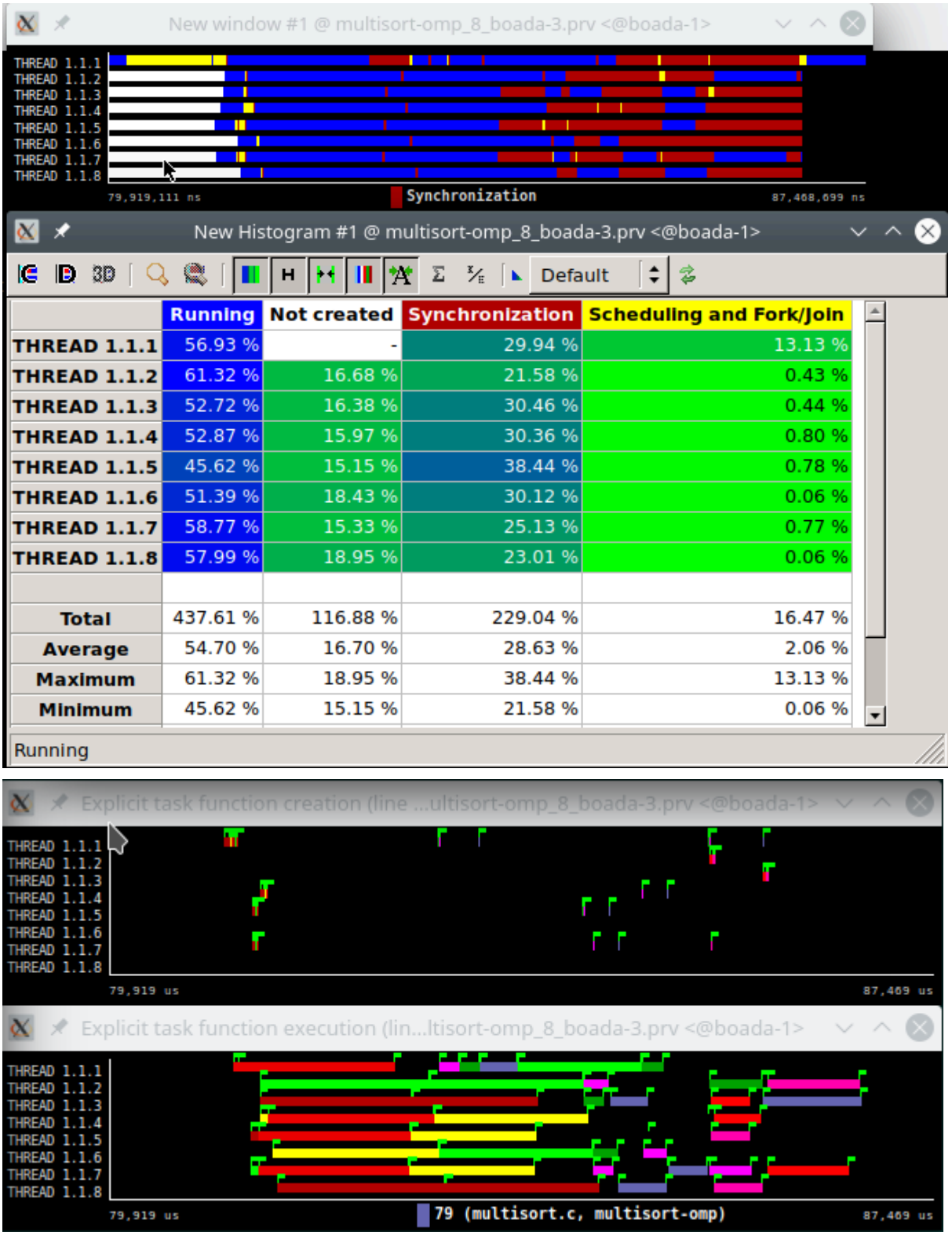


Figura 3.12: Gráficas y tablas con datos del trabajo de threads de la estrategia tree con cut-off a 0.



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	56.93 %	-	29.94 %	13.13 %
THREAD 1.1.2	61.32 %	16.68 %	21.58 %	0.43 %
THREAD 1.1.3	52.72 %	16.38 %	30.46 %	0.44 %
THREAD 1.1.4	52.87 %	15.97 %	30.36 %	0.80 %
THREAD 1.1.5	45.62 %	15.15 %	38.44 %	0.78 %
THREAD 1.1.6	51.39 %	18.43 %	30.12 %	0.06 %
THREAD 1.1.7	58.77 %	15.33 %	25.13 %	0.77 %
THREAD 1.1.8	57.99 %	18.95 %	23.01 %	0.06 %
Total	437.61 %	116.88 %	229.04 %	16.47 %
Average	54.70 %	16.70 %	28.63 %	2.06 %
Maximum	61.32 %	18.95 %	38.44 %	13.13 %
Minimum	45.62 %	15.15 %	21.58 %	0.06 %

Figura 3.13: Gráficas y tablas con datos del trabajo de threads de la estrategia tree con cut-off a 1.

Primero hablaremos de las gráficas del cut-off a 0.

En la timeline de ejecución de tareas se puede observar el esquema del código básico visto anteriormente con tareador: las 4 llamadas a multisort y las 3 llamadas a merge. Solo hay estas tareas porque el cutoff a 0 hace que únicamente se aplique la definición de tarea en un piso del árbol de la recursión.

En cuanto a tiempos, se puede ver que hay 4 threads que están la mayoría de tiempo haciendo trabajo útil mientras que los otros 4 están esperando. Esto se debe a la poca cantidad de tareas que hay.

Por otra parte, las gráficas de la figura 3.13 muestran un mejor balance de tiempo en general de todos los threads. Por lo que parece, un mayor cutoff podría implicar un mejor posible paralelismo.

Eso mismo exploraremos ahora. Usando el script submit-cutoff-omp.sh obtenemos el gráfico de la figura 3.14:

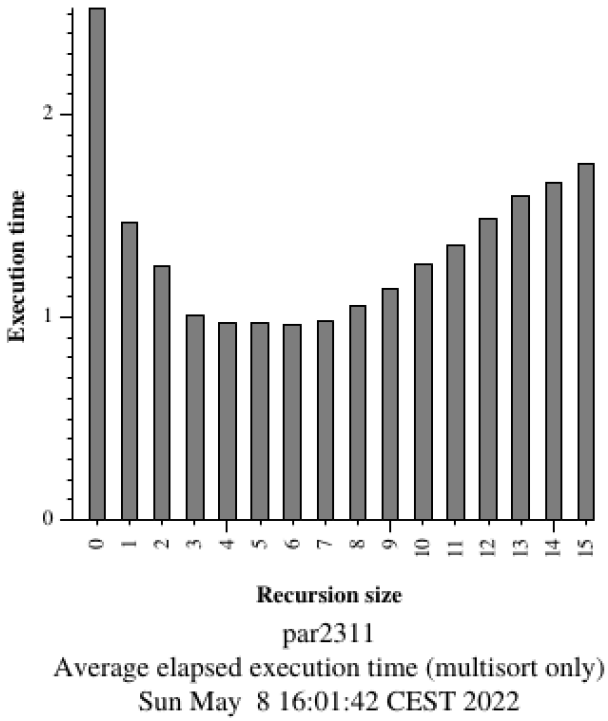


Figura 3.14: Gráfico del tiempo de ejecución de multisort-omp-tree-cutoff.c según el cut-off

Aparentemente el mejor nivel de recursión para detener la asignación de tareas y paralelizar está entre el 4 y el 7 (probablemente el 6), después de éstos valores el tiempo de ejecución empeora considerablemente. Así que con este gráfico se demuestra que más cut-off no siempre significa mejor rendimiento.

Para comparar más visualmente se puede ver la enorme diferencia en el speedup del cut-off por defecto vs. el cut-off a 6 en la siguiente figura:

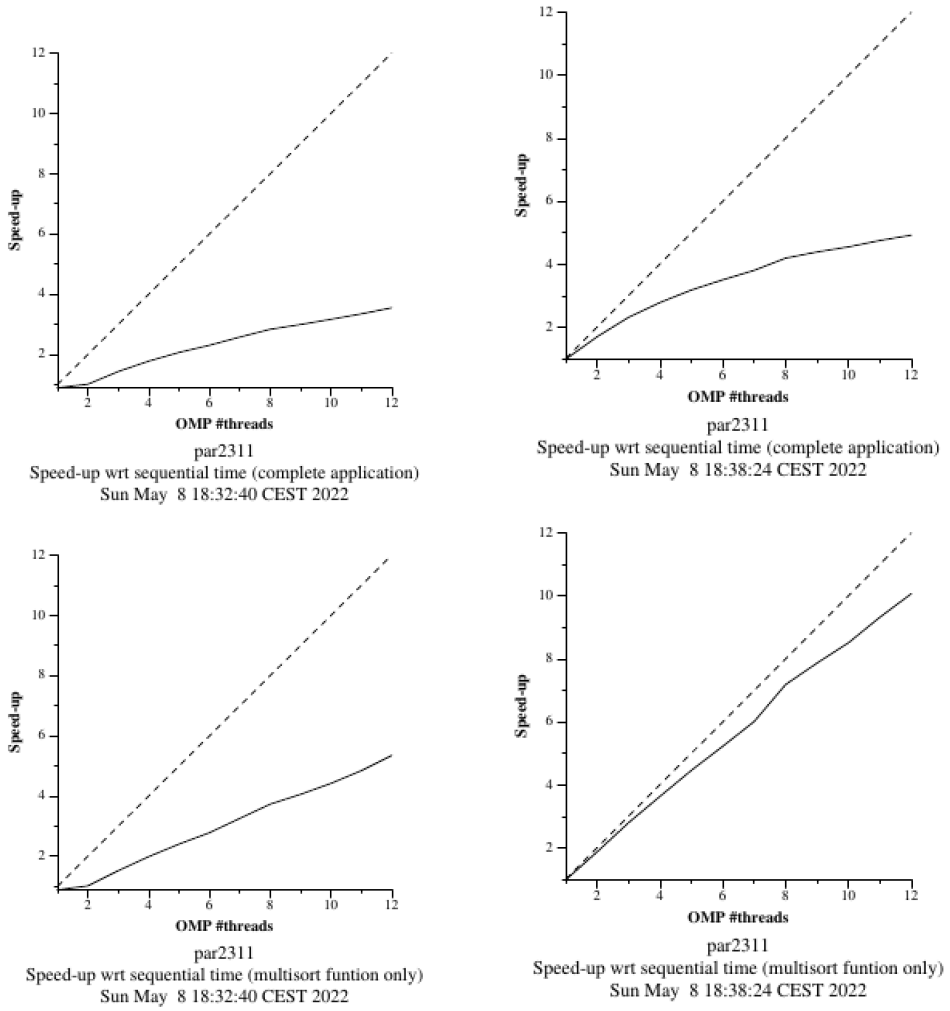


Figura 3.15: Plots del speedup del programa entero (arriba) y solamente la recursión (abajo) de cut-off por defecto (izquierda) y del cut-off a 6 (derecha).

Como se ha visto en la figura 3.14 el mejor cut-off posible para paralelizar es el 6, así que tiene mucho sentido que el speedup vaya aumentando a un ritmo superior que el básico según el número de threads usados.

3.4 - Opcional: Escalabilidad del cut-off hasta el infinito y más allá.

En este pequeño experimento seguiremos explorando cómo cambiaría el tiempo de ejecución con el mecanismo de cut-off aumentando el número de threads hasta 24.

Evidentemente escogeremos el mejor nivel que se ha visto posible de cutoff, que es el 6 (aunque también haremos una prueba con un cutoff más grande para ver si hay

mejoría al haber más threads). Las gráficas se pueden observar en las siguientes figuras:

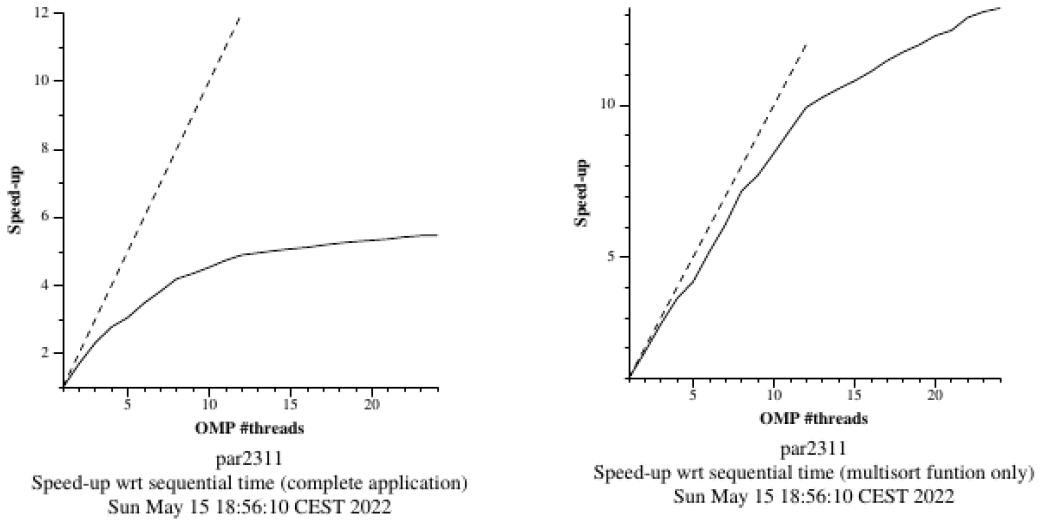


Figura 3.16: Speedup de hasta 24 threads con cutoff a 6 y merge size y sort size a 128

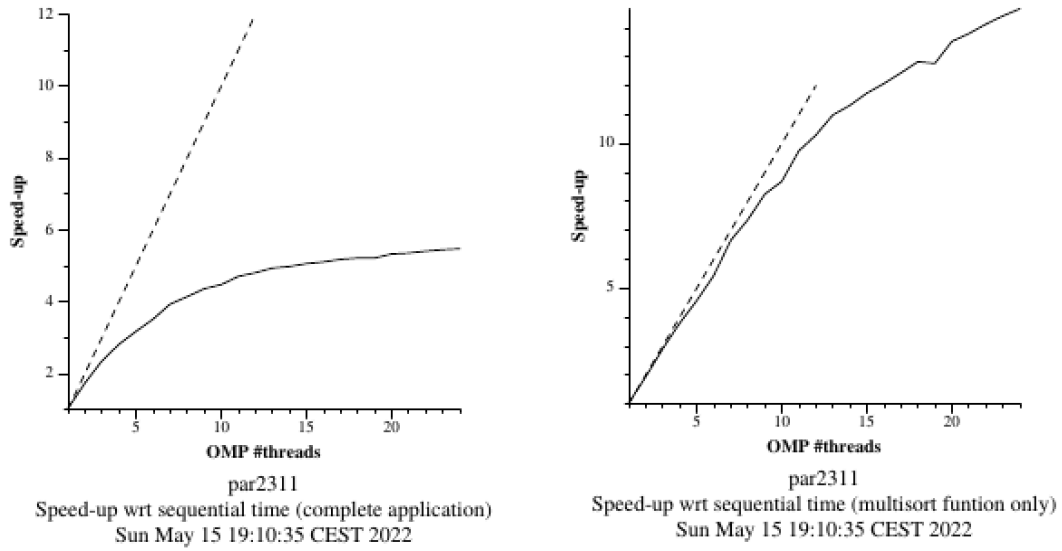


Figura 3.17: Speedup de hasta 24 threads con cutoff a 6 y merge size y sort size a 1024

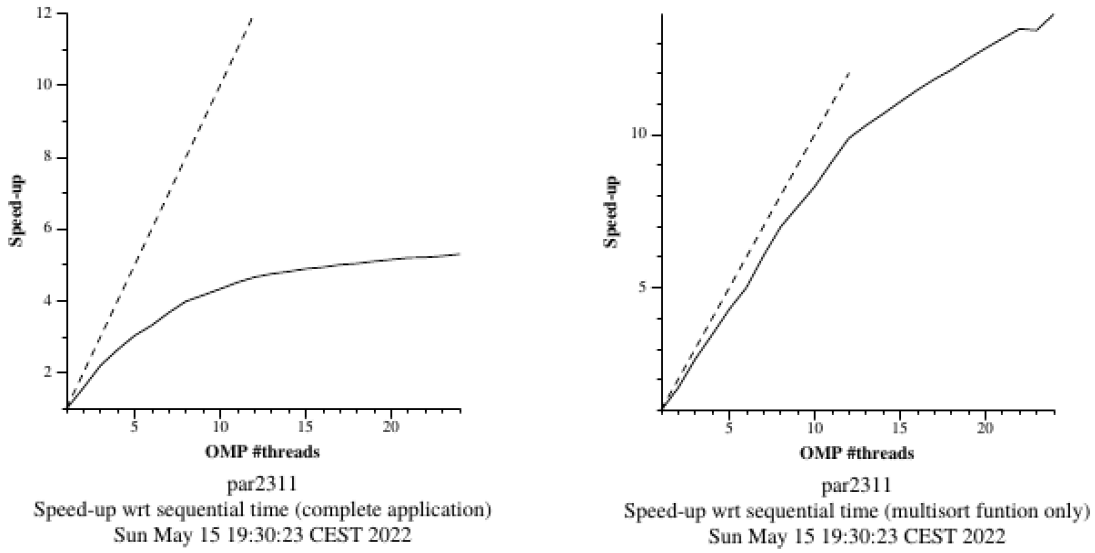


Figura 3.18: Speedup de hasta 24 threads con cutoff a 16 y merge size y sort size a 1024

El merge size y el sort size se han modificado en varias ejecuciones para cambiar la granularidad en la recursión.

Se aprecia que el rendimiento del programa en las tres ejecuciones diferentes sigue aumentando aunque ya no de una forma tan lineal como lo hacía hasta 12 threads, esto seguramente sea provocado por el gran tiempo de overhead que se necesita para la estrategia Tree. Aún así la escalabilidad sigue siendo bastante buena ya que el cut-off permite poner un “tope” a la creación de tareas disminuyendo el vasto tiempo de sincronización que se requeriría si no se usase.

4 - Usando dependencias de tareas en OpenMP

Finalmente haremos un análisis como ya hemos hecho anteriormente pero esta vez añadiendo dependencias en las tareas en lugar de taskwaits o taskgroups. Este método muy probablemente nos ahorra el tiempo perdido de sincronización que nos provocan los pragmas ya mencionados.

Los cambios hechos para satisfacer estos criterios están en el archivo “multisort-omp-tree-cutoff-dependency.c”.

Básicamente los cambios hechos han sido quitar los taskwaits y añadir en las definiciones de las tareas la cláusula “depend” para marcar la dependencia de las variables.

La salida del programa usando 8 threads y cut-off a 6 (el más eficiente según se ha comprobado antes) es la siguiente:

```
************************************************************************
**
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                          CUTOFF=6
Number of threads in OpenMP:            OMP_NUM_THREADS=8
************************************************************************
**
Initialization time in seconds: 0.857651
Multisort execution time: 0.859873
Check sorted data execution time: 0.015826
Multisort program finished
************************************************************************
**
```

Que como se puede ver es correcta y tiene un tiempo de ejecución de 0,85 segundos, mucho más rápida que el código del principio del todo del deliverable, aproximadamente 6.322566 / 0.859873 ≈ 7.35 veces más rápido. Otra característica a destacar es que el tiempo de ejecución se ha reducido tanto que ha alcanzado a ser prácticamente igual al tiempo de inicialización.

Como ya es costumbre, miraremos el speedup obtenido. Las gráficas se encuentran en la siguiente figura:

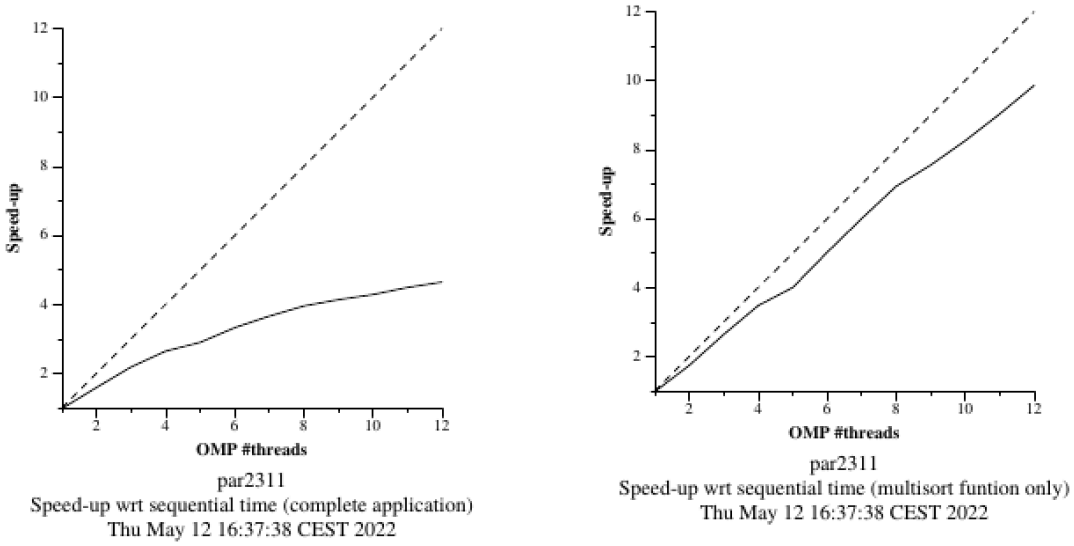


Figura 4.1: Gráficas del nuevo speedup según el número de threads. A la izquierda el programa entero, a la derecha solamente la parte recursiva.

Son speedups muy buenos pero no del todo diferentes a los ya vistos en la sección 3.2. Esto hace pensar que aunque esta versión puede llegar a ser más eficiente es muchísimo más compleja de programar así que no merece del todo la pena el esfuerzo.

Para terminar esta sección haremos el ritual que ya hemos repetido varias veces: mirar los gráficos, timelines y tablas de Paraver para saber qué está ocurriendo exactamente.
Las figuras 4.2, 4.3 y 4.4 muestran los siguientes datos haciendo zoom a la parte recursiva:

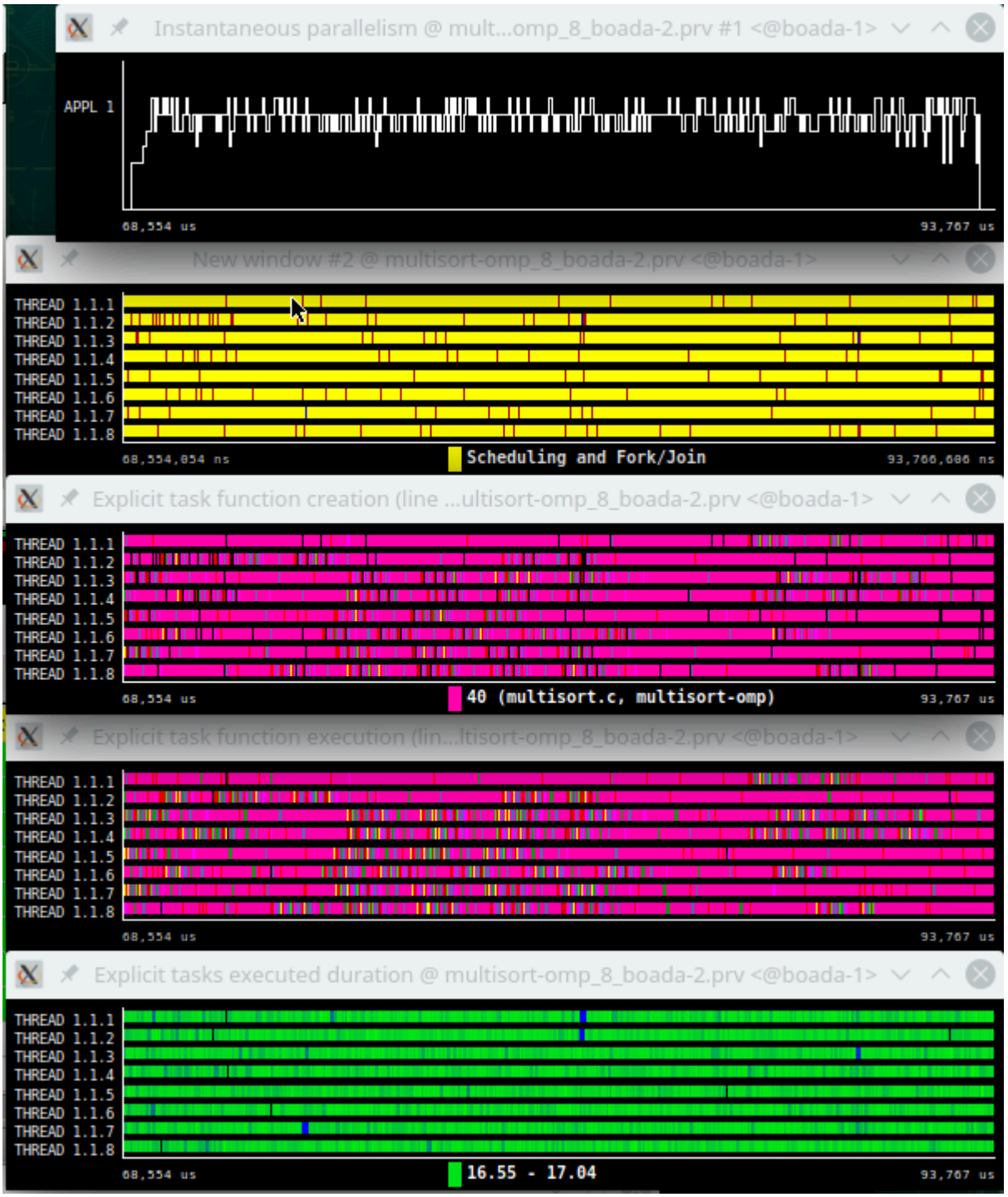
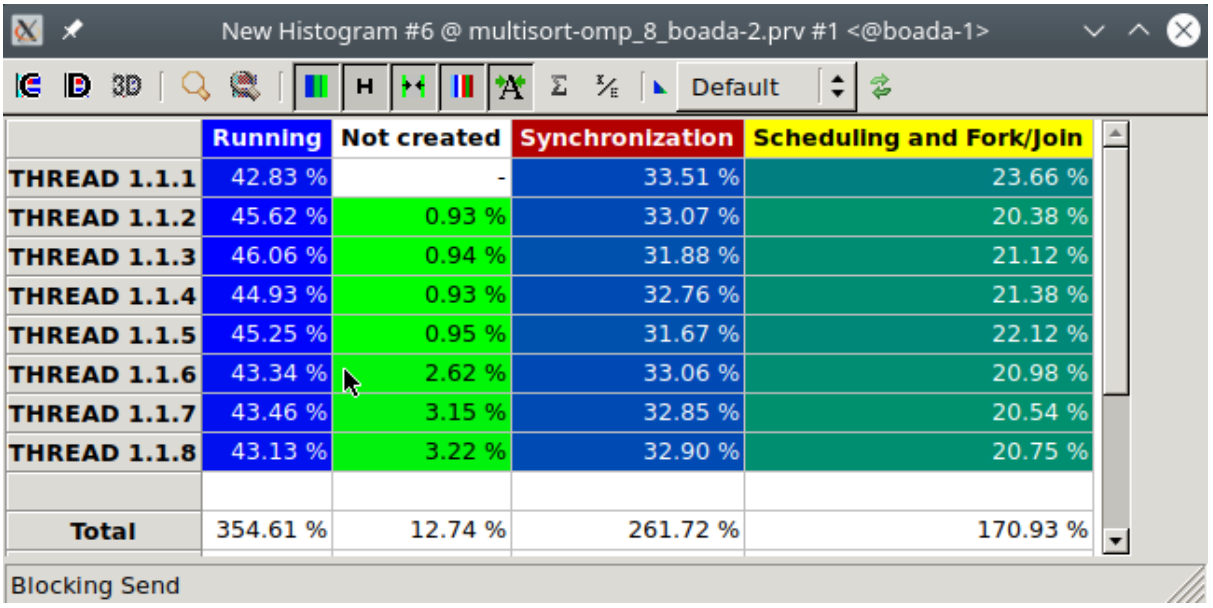
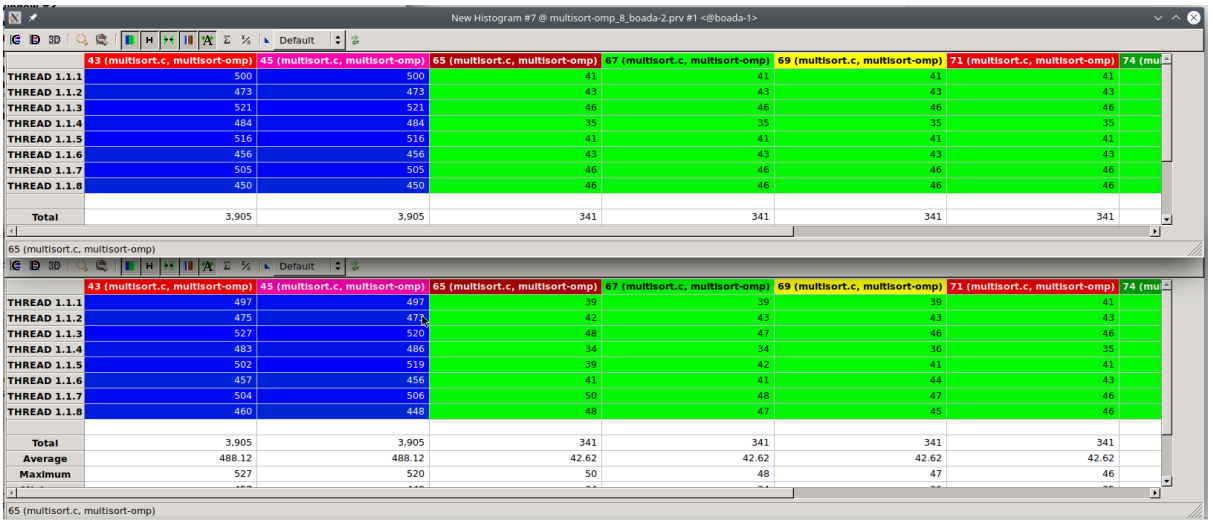


Figura 4.2: Timelines de la sección paralela de la creación y ejecución de tareas.



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	42.83 %	-	33.51 %	23.66 %
THREAD 1.1.2	45.62 %	0.93 %	33.07 %	20.38 %
THREAD 1.1.3	46.06 %	0.94 %	31.88 %	21.12 %
THREAD 1.1.4	44.93 %	0.93 %	32.76 %	21.38 %
THREAD 1.1.5	45.25 %	0.95 %	31.67 %	22.12 %
THREAD 1.1.6	43.34 %	2.62 %	33.06 %	20.98 %
THREAD 1.1.7	43.46 %	3.15 %	32.85 %	20.54 %
THREAD 1.1.8	43.13 %	3.22 %	32.90 %	20.75 %
Total	354.61 %	12.74 %	261.72 %	170.93 %

Figura 4.3: Tabla con los porcentajes de tiempo de los diferentes threads.



	43 (multisort.c, multisort-omp)	45 (multisort.c, multisort-omp)	65 (multisort.c, multisort-omp)	67 (multisort.c, multisort-omp)	69 (multisort.c, multisort-omp)	71 (multisort.c, multisort-omp)	74 (mul
THREAD 1.1.1	500	500	41	41	41	41	
THREAD 1.1.2	473	473	43	43	43	43	
THREAD 1.1.3	521	521	46	46	46	46	
THREAD 1.1.4	484	484	35	35	35	35	
THREAD 1.1.5	516	516	41	41	41	41	
THREAD 1.1.6	456	456	43	43	43	43	
THREAD 1.1.7	505	505	46	46	46	46	
THREAD 1.1.8	450	450	46	46	46	46	
Total	3,905	3,905	341	341	341	341	

	43 (multisort.c, multisort-omp)	45 (multisort.c, multisort-omp)	65 (multisort.c, multisort-omp)	67 (multisort.c, multisort-omp)	69 (multisort.c, multisort-omp)	71 (multisort.c, multisort-omp)	74 (mul
THREAD 1.1.1	497	497	39	39	39	41	
THREAD 1.1.2	475	475	42	43	43	43	
THREAD 1.1.3	527	520	48	47	46	46	
THREAD 1.1.4	483	486	34	34	36	35	
THREAD 1.1.5	502	519	39	42	41	41	
THREAD 1.1.6	457	456	41	41	44	43	
THREAD 1.1.7	504	506	50	48	47	46	
THREAD 1.1.8	460	448	48	47	45	46	
Total	3,905	3,905	341	341	341	341	
Average	488.12	488.12	42.62	42.62	42.62	42.62	
Maximum	527	520	50	48	47	46	

Figura 4.4: Tabla con el número de tareas creadas y ejecutadas.

En las timelines se aprecia un enorme aprovechamiento del tiempo por parte de los threads ya que casi no se distingue una parte donde no estén haciendo trabajo. Esto se ve más exactamente en la tabla con los porcentajes de tiempo, que muestran que aunque el tiempo funcionando (Running) no cambia mucho respecto a la versión anterior y el tiempo sincronizándose aumenta bastante, el porcentaje de tiempo donde no están haciendo nada porque no existen disminuye considerablemente (de ~16% pasa a ~2%).

Con estos datos concluimos que aunque sea un método que provoque más overhead el tiempo se aprovecha mucho más y podría incluso ser mejor escalable que la versión anterior si se usan más threads.

4.1 - Opcional: Paralelizando la inicialización de variables.

Para terminar vamos a ver si podemos mejorar la paralelización del programa de alguna otra manera.

Para ello paralelizaremos la inicialización de variables del programa, que es una parte de momento secuencial que se come una gran parte del tiempo de ejecución del código.

Los cambios hechos en el código para conseguir esto están en el fichero “multisort-omp-tree-cutoff-dependency-opt2.c”.

En las siguientes figuras están las gráficas de speedup de las que ya deberíamos estar familiarizados, pero esta vez hay una diferencia:

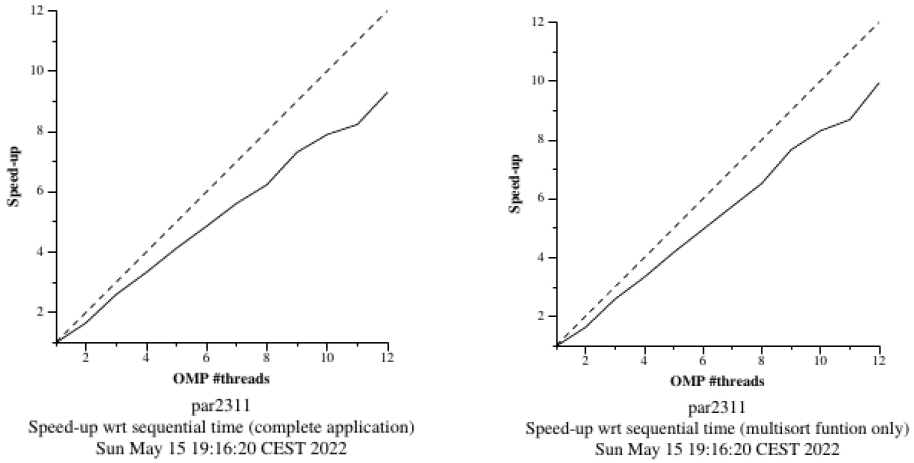


Figura 4.5: Gráficas del speedup de todo el programa y de solamente la parte recursiva según el número de threads.

Esta vez la gráfica de todo el programa escala de manera similar a la de solo la parte recursiva. Este logro se ha obtenido ya que como hemos “mejorado” la fracción del tiempo que se comía la parte de inicialización de variables, por la ley de Amdahl ahora el código entero mejora más.

Las siguientes figuras muestran las timelines de los threads en Paraver:

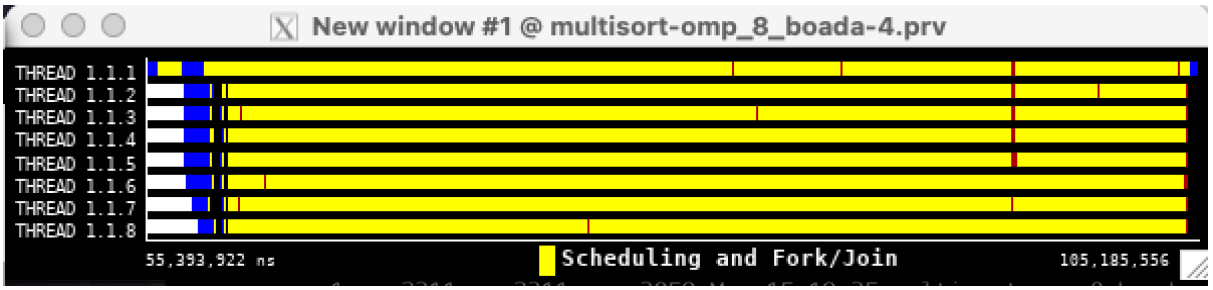


Figura 4.6: Timeline de creación y ejecución de tareas.

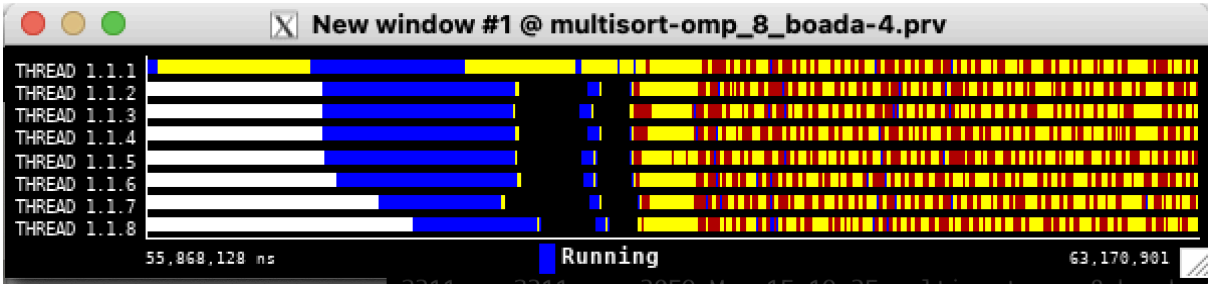


Figura 4.7: Timeline de creación y ejecución de tareas (Zoom en la zona del principio, donde se inicializan las variables).

Se observa que el tiempo de ejecución de la parte inicial ha disminuido bastante al repartir el trabajo entre los threads.

Con esto hemos conseguido el extra de paralelismo que necesitábamos en el código para alcanzar una escalabilidad más deseable.

5 - Conclusiones

En este deliverable hemos visto diferentes formas de paralelizar la recursión y el análisis de su eficiencia. La mejor estrategia es la definición de tareas en las divisiones de la recursión que hemos visto en la práctica es la *tree*, ya que de este modo no hay un thread responsable de crear las tareas y se puede aprovechar más el tiempo. Aún así, para árboles de recursión pequeños podría llegar a ser más útil la estrategia de *leaf* ya que tiene muchas menos tareas y por tanto el tiempo de sincronización es más pequeño y no llegaría a importar el desbalance de trabajo.