UNIVERSITAT POLITÈCNICA DE CATALUNYA

Paralelismo

Laboratorio 5: Geometric (data) decomposition using implicit tasks: heat diffusion equation

01/06/2022

Haopeng Lin Ye

Joan Sales de Marcos

ÍNDICE

1 - introducción

En este deliverable trabajaremos algunas formas de descomponer los datos para mejor paralelización. Para ello se nos han facilitado varios códigos que hacen simulaciones de calor en cuerpos sólidos usando una matriz.

Por ejemplo, podemos aplicar la descomposición de datos por bloques. Básicamente dividimos una matriz en bloques por filas y columnas (i,j) y cada uno es una tarea para paralelizar:

bloque 0,0	bloque 0,1	…	bloque 0,j
bloque 1,0	bloque 1,1	…	bloque 1,j
…	…	…	…
bloque i,0	bloque i,1	…	bloque i,j

En este caso habrán i*j tareas que se repartirán entre el número de threads que ejecutarán el programa. Esta es una forma de “Descomponer” la Matriz, otras maneras pueden ser por filas, por columnas, por bloques cíclicamente…

2 - Programa de difusión de calor (secuencial) y análisis con Tareador

2.1 - Introducción a los códigos

En este caso en concreto, tenemos dos algoritmos para calcular la difusión de calor: el algoritmo de Jacobi y el de Gauss-Seidel.

Los códigos que analizaremos en este caso son heat.c y solver.c.

El primero es el código principal, que en resumidas cuentas inicializa las variables y llama a la función “solve” (de solver.c) para resolver en caso de que el algoritmo a usar sea el de Gauss-Seidel y también llama a la función “copy_mat” (también está en solver.c) si se usa el algoritmo de Jacobi.

Al ejecutar el programa sin modificaciones, estas son las salidas:

Usando Jacobi:

```
par2319@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
```

```
Solver            : 0 (Jacobi)
Num. Heat sources : 2
  1: (0.00, 0.00) 1.00 2.50
  2: (0.50, 1.00) 1.00 2.50
Time: 4.596
Flops and Flops per second: (11.182 GFlop => 2432.94 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Usando Gauss-Seidel:

```
par2319@boada-1:~/lab5$ ./heat test.dat -a 1 -o heat-gauss.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 1 (Gauss-Seidel)
Num. Heat sources : 2
  1: (0.00, 0.00) 1.00 2.50
  2: (0.50, 1.00) 1.00 2.50
Time: 5.997
Flops and Flops per second: (8.806 GFlop => 1468.57 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

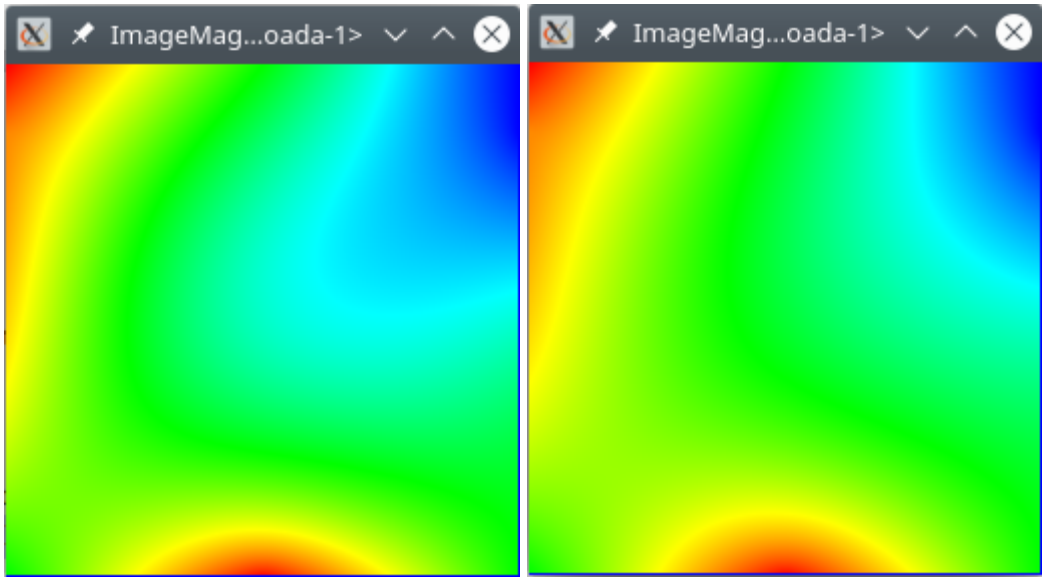


Figura 2.1: Imágenes obtenidas al hacer display heat-jacobi.ppm y display heat-gauss.ppm respectivamente

Se observa que el algoritmo de Jacobi es ligeramente más rápido que el de Gauss-Seidel (5.997 / 4,596 = ~1,3 veces más rápido) y tiene más GigaFlops. De todas formas, en la figura 2.1 se pueden ver unas diferencias en el resultado entre las dos versiones. Estos resultados e imágenes los usaremos para comparaciones más adelante.

2.2 - Análisis con Tareador

En esta sección exploraremos la paralelización de los dos algoritmos con Tareador. Primero ejecutaremos la instrumentación básica que se nos ha otorgado, definiendo las funciones principales como tareas:

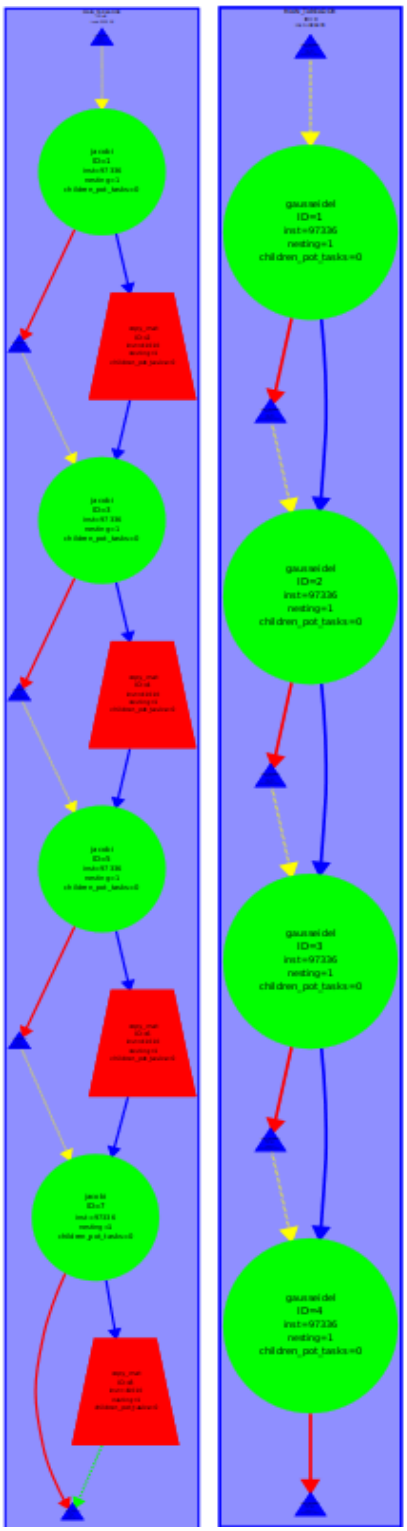


Figura 2.2: Grafos de dependencias del algoritmo de Jacobi (Izquierda) y el de Gauss-Seidel (Derecha)

En estos grafos (claramente secuenciales) se puede ver que son muy mejorables, lamentablemente en este nivel de granularidad no se puede hacer mucho, ya que están las tareas definidas como las funciones enteras. Sin embargo, si nos adentramos en éstas podremos conseguir mejoría.

Más concretamente, dentro del código de la función solver hay una descomposición de datos en bloques. Entonces, ¿qué pasaría si definimos esos bloques como tareas (Explicado anteriormente en la introducción)? Los resultados se pueden observar en la siguiente figura:

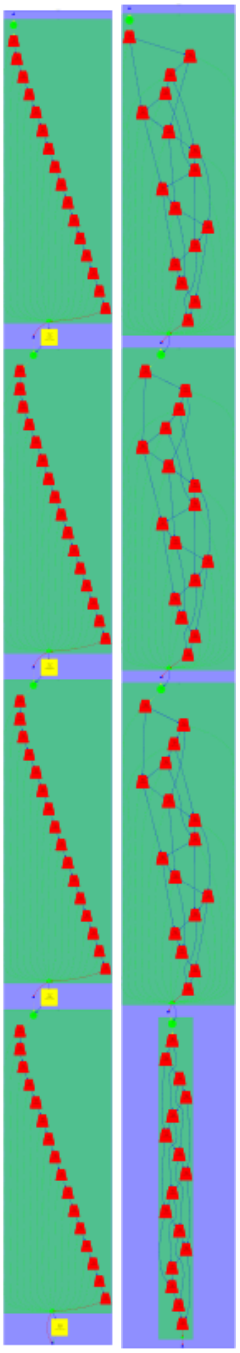


Figura 2.3: TDGs del algoritmo de Jacobi (Izquierda) y el de Gauss-Seidel (Derecha), aplicando Block Decomposition.

Se ve a simple vista que aunque hayamos aplicado la descomposición por bloques los grafos aún son secuenciales. Para saber qué provoca esto analizaremos las dependencias.

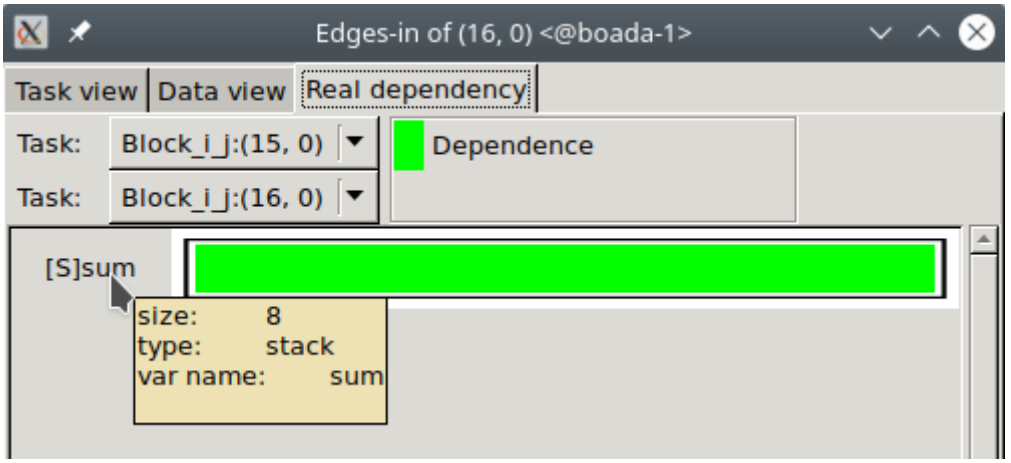


Figura 2.4: Dependencia del TDG de Jacobi

En el caso del algoritmo de Jacobi, la variable “sum” está causando la serialización.

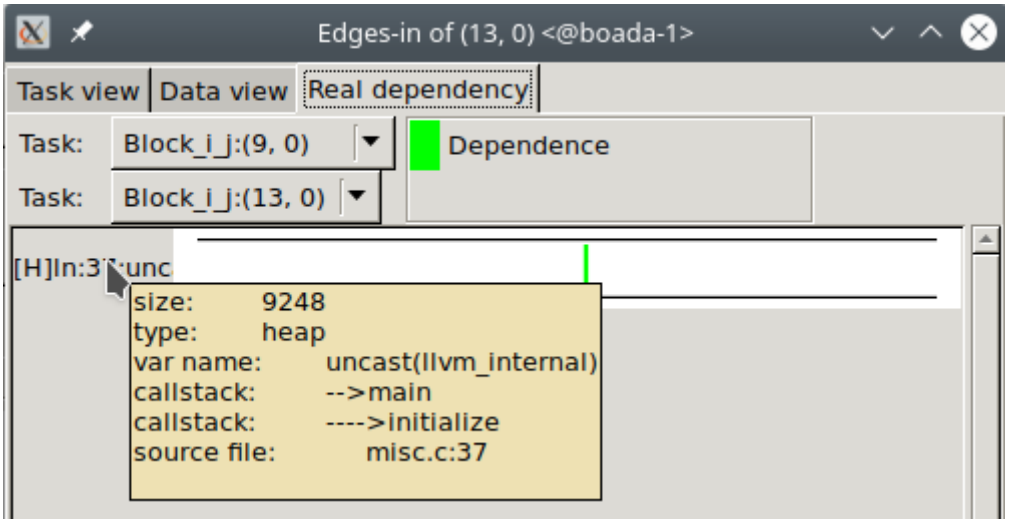


Figura 2.5: Dependencia del TDG de Gauss-Seidel

En la línea 37 de misc.c hay:

```
(param->u) = (double*)calloc( sizeof(double),np*np );
```

En el caso del algoritmo de Gauss, la variable “u” está causando la serialización.

2.2.1 - Explorando el algoritmo de Jacobi

Para arreglar la secuencialización que provoca la dependencia de la variable “sum” usaremos unas funciones que incluye Tareador (tareador_enable_object y tareador_disable_object) para hacer que esta variable no sea tomada como importante entre los threads y así poder paralelizar el programa más aún. Los resultados se ven a continuación:

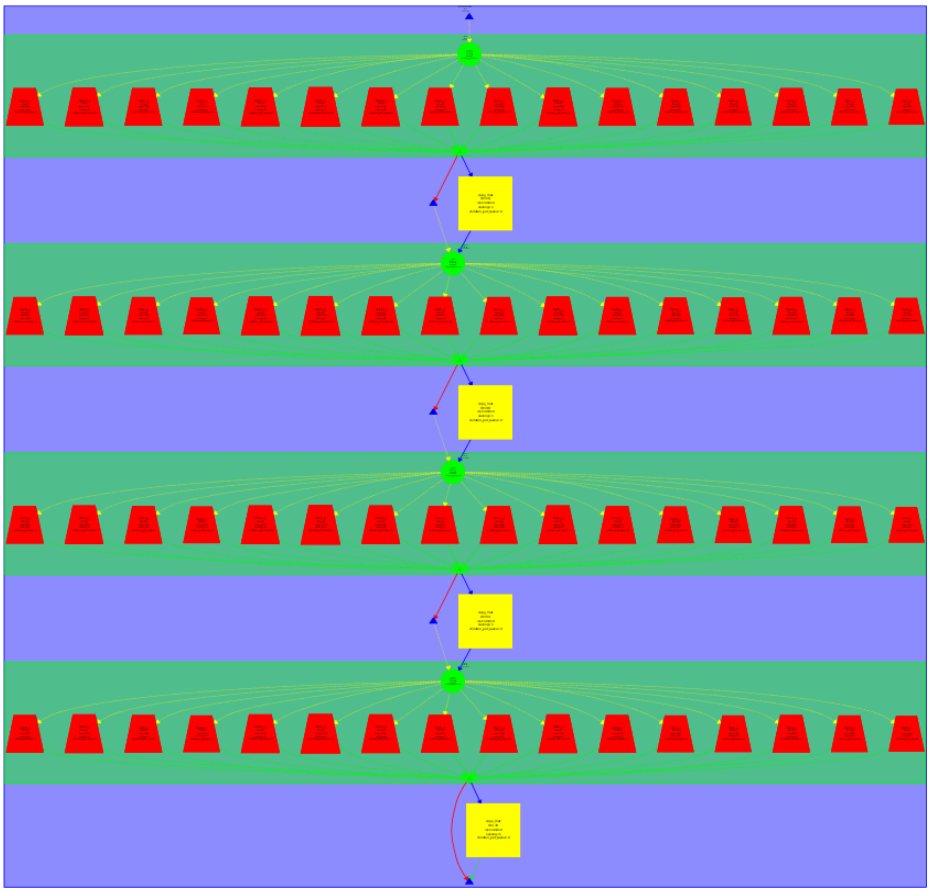


Figura 2.6: TDG del algoritmo de Jacobi usando tareador_enable/disable_object

Para el algoritmo de Jacobi, el grafo de dependencias tiene mucha mejor pinta, ya que los pisos son más horizontales que anteriormente. Para proteger el acceso a la variable “sum” al hacer el código en OpenMP real se podría usar `#pragma omp atomic` o bien `#pragma omp critical`.

Aún así, hay una parte del grafo que todavía es secuencial: Los cuadrados amarillos pertenecientes a la llamada a la función “copy_mat”. En la figura 2.7 se puede ver la dependencia exacta:

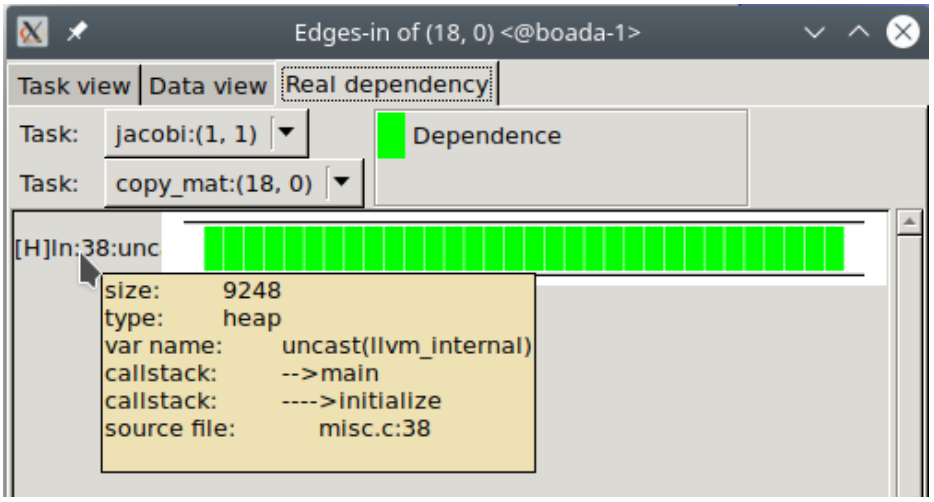


Figura 2.7: Dependencia de los cuadrados amarillos del TDG visto en la figura 2.6

En la línea 38 de misc.c hay:

```
(param->uhelp) = (double*)calloc( sizeof(double),np*np );
```

La variable uhelp está causando la serialización en este caso.

Haciendo una simulación con 4 procesadores obtenemos la línea temporal de trabajo siguiente:

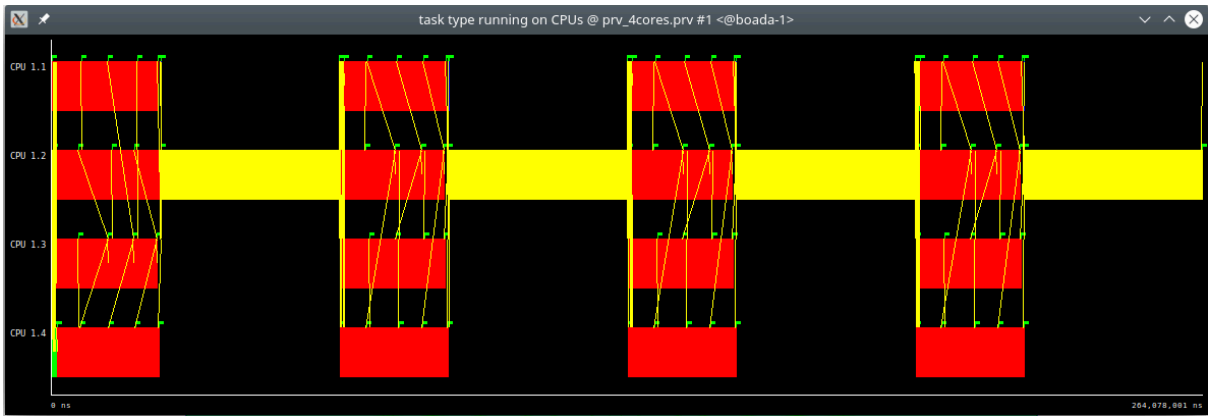


Figura 2.8: Timeline de las tareas en el algoritmo de Jacobi

Para este algoritmo, aún nos queda esta parte que se puede paralelizar, los sectores amarillos que corresponden a copy_mat. Así que volveremos a aplicar la descomposición de datos en bloques dentro de esa función, el resultado es el siguiente:

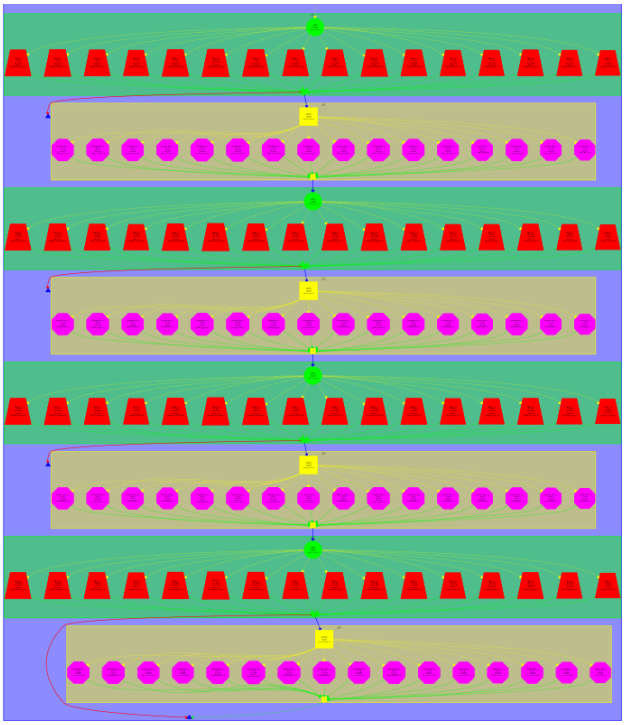


Figura 2.9: TDG definitivo de Jacobi, descomponiendo todo por bloques.

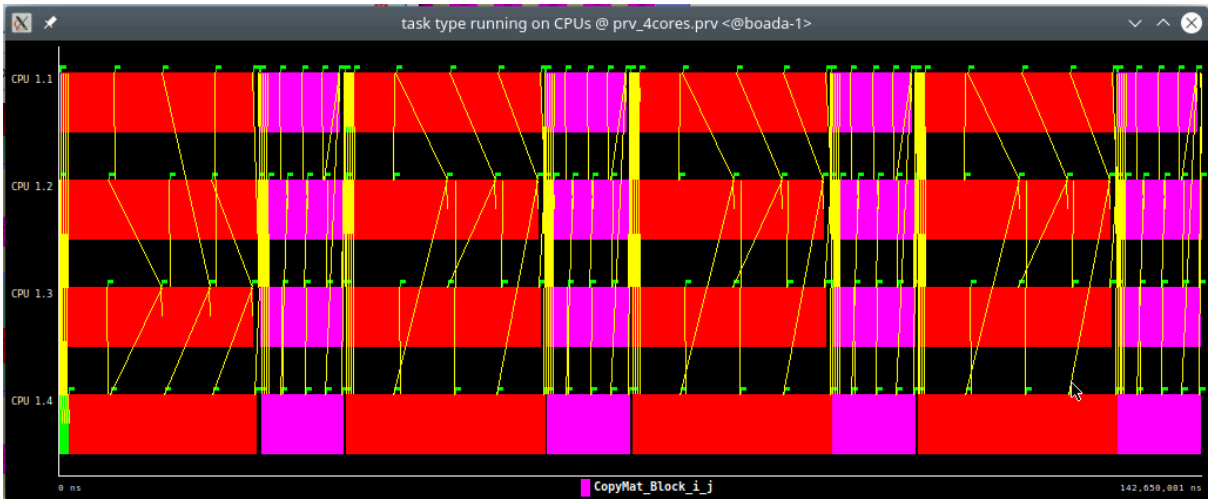


Figura 2.10: Timeline de trabajo de Jacobi definitivo.

Como se observa en las figuras 2.9 y 2.10, el programa ahora prácticamente no pierde el tiempo ya que ningún thread se queda haciendo nada esperando a que acabe otro. Esta forma es mucho más paralela que la básica.

Ahora que tenemos la versión definitiva de Jacobi, vamos a ver qué podemos hacer con el algoritmo de Gauss-Seidel.

2.2.2 - Explorando el algoritmo de Gauss-Seidel

Primero haremos la paralelización principal, que es usando tareador_enable / disable_object. Se pueden ver pequeños cambios en el TDG:

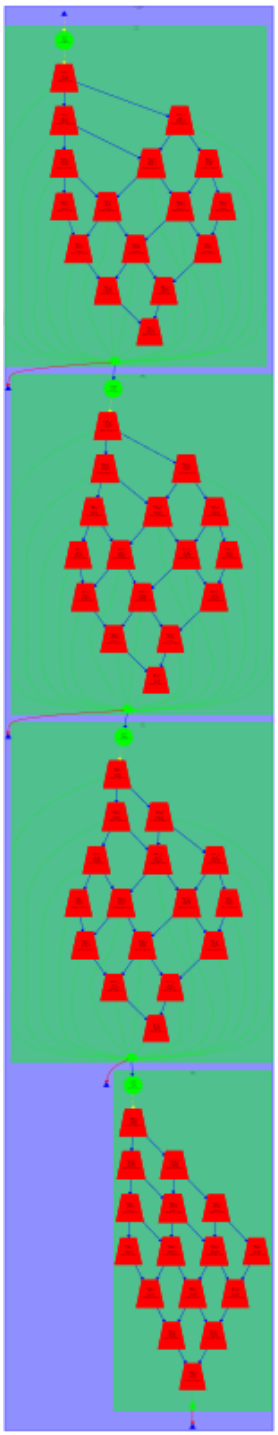


Figura 2.11: TDG del algoritmo de Gauss usando tareador_enable/disable_object

Aunque no cambie mucho respecto al original de este algoritmo, está mucho mejor estructurado y las tareas están más organizadas en pisos, a diferencia del caos absoluto que se vislumbraba en el grafo anterior (sin usar las funciones).

Las dependencias de las tareas de los trapecios rojos son las siguientes:

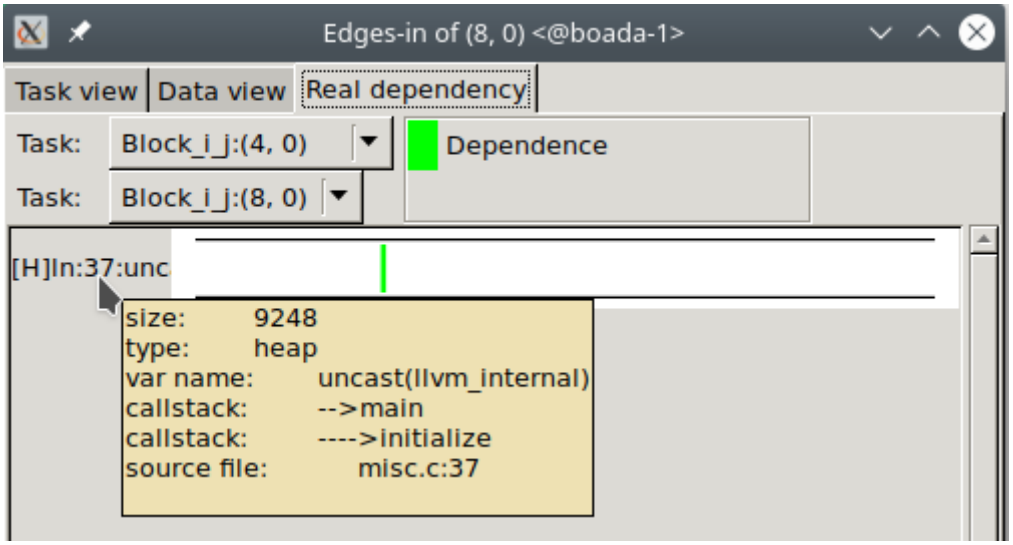


Figura 2.12: Dependencia de los trapecios rojos del nuevo TDG de Gauss-Seidel

En la línea 37 de misc.c hay:

```
(param->u) = (double*)calloc( sizeof(double),np*np );
```

“u” está causando la serialización.

Se aprecia que la dependencia no ha cambiado en absoluto respecto al original. Y aplicando la descomposición de datos por bloques en la función “copy_mat” en este caso no sirve porque al usar este algoritmo no se llama a esa función.

Para terminar con este apartado, en la figura 2.13 podemos ver una simulación de este algoritmo con 4 procesadores:

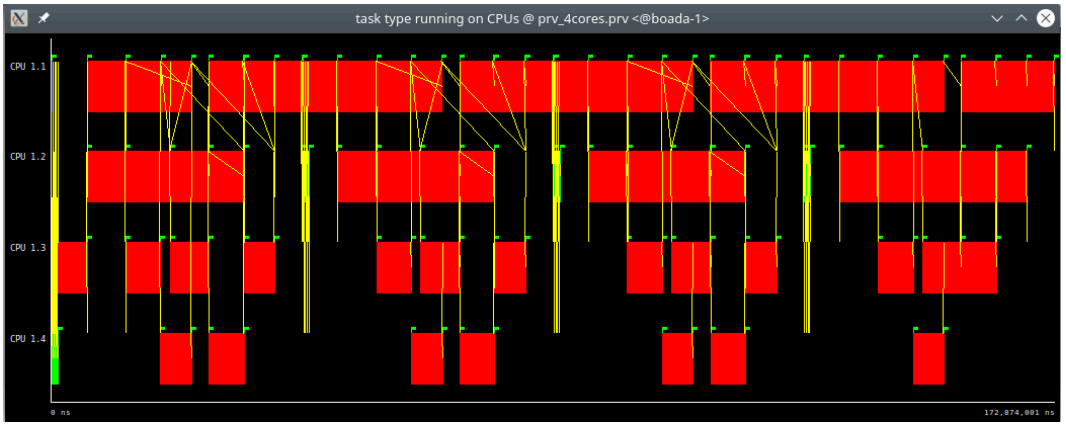


Figura 2.13: Timeline de las tareas del algoritmo de Gauss-Seidel

Como se puede ver a simple vista, no hay un lugar que claramente sea secuencial y se tenga que paralelizar, así que damos este tema por zanjado.

3 - Paralelización de los solvers de la ecuación de calor.

En esta sección aplicaremos el paralelismo al programa de difusión de calor según lo que hemos aprendido y visto usando Tareador. Utilizaremos los pragmas y funciones que nos ofrece OpenMP tanto en la versión de Jacobi como en la de Gauss-Seidel.

3.1 - Solver de Jacobi

Según hemos visto en Tareador, se puede paralelizar tanto la función “solve” como la de “copy_mat”. Empezaremos por la primera, los cambios hechos en el código están en el archivo solver-omp-jacobi.c.

En resumen: hemos obtenido el número de threads y de bloques para poder repartirlos entre ellos de forma equitativa dentro de los bucles, cada uno con su ID de thread, así nos aseguramos que todos hagan el trabajo que les corresponde.

Además se ha usado el #pragma omp parallel para declarar la región paralela, seguido de private(diff) y reduction(+:sum) porque en Tareador hemos visto que la variable sum crea dependencias, y “diff” se usa para obtener el resultado de la variable “sum”, usando estas cláusulas evitaremos el posible datarace.

Después de hacer estos cambios hemos ejecutado el programa con diversos números de threads. Los resultados se pueden ver a continuación:

secuencial:

```
par2311@boada-1:~/lab5$ ./heat test.dat -a 0 -o heat-jacobi.ppm
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 7.105
Flops and Flops per second: (11.182 GFlop => 1573.85 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

con 4 threads:

```
par2311@boada-1:~/lab5$ cat heat-omp-jacobi-4-boada-2.txt
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
```

```
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 4.471
Flops and Flops per second: (11.182 GFlop => 2500.95 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

con 8 threads:

```
par2311@boada-1:~/lab5$ cat *txt
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 5.133
Flops and Flops per second: (11.182 GFlop => 2178.28 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Los tiempos de ejecución obtenidos son:

Secuencial	**4 threads**	**8 threads**
7.105s	4.471s	5.133s

Se observa que el tiempo de ejecución con 4 threads es mucho más rápido que el secuencial e incluso un poco más que el de 8 threads (teorizamos que puede ser porque la función “copy_mat” aún no se ha paralelizado y está provocando serialización). Después de aplicar el comando de la terminal “diff” para comparar resultados hemos visto que no hay diferencias con las imágenes originales (Figura 2.1).

Para ver mejor la escalabilidad usaremos un script que se nos ha proporcionado que simula varias ejecuciones hasta 12 threads. Los gráficos obtenidos se observan en la figura 3.1:

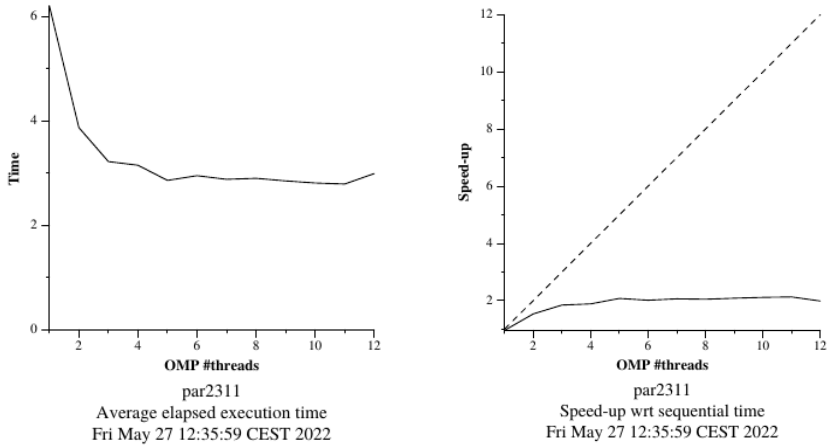


Figura 3.1: Gráficas que muestran el tiempo de ejecución y el Speedup según el número de threads (Jacobi, copy_mat no paralelizado)

La clara parte plana que hay a partir de 3-4 Threads demuestra que la escalabilidad de esta versión del código no es muy buena. Lo más probable es que esto ocurra porque aún falta parte del programa por paralelizar.

Para averiguar lo que está ocurriendo usaremos Paraver para ver más en profundidad y exactitud el trabajo de los threads, concretamente usaremos 8 threads, que es el valor por defecto.

Las timelines y gráficos más importantes y de los que podemos sacar más conclusiones se encuentran en las siguientes figuras:

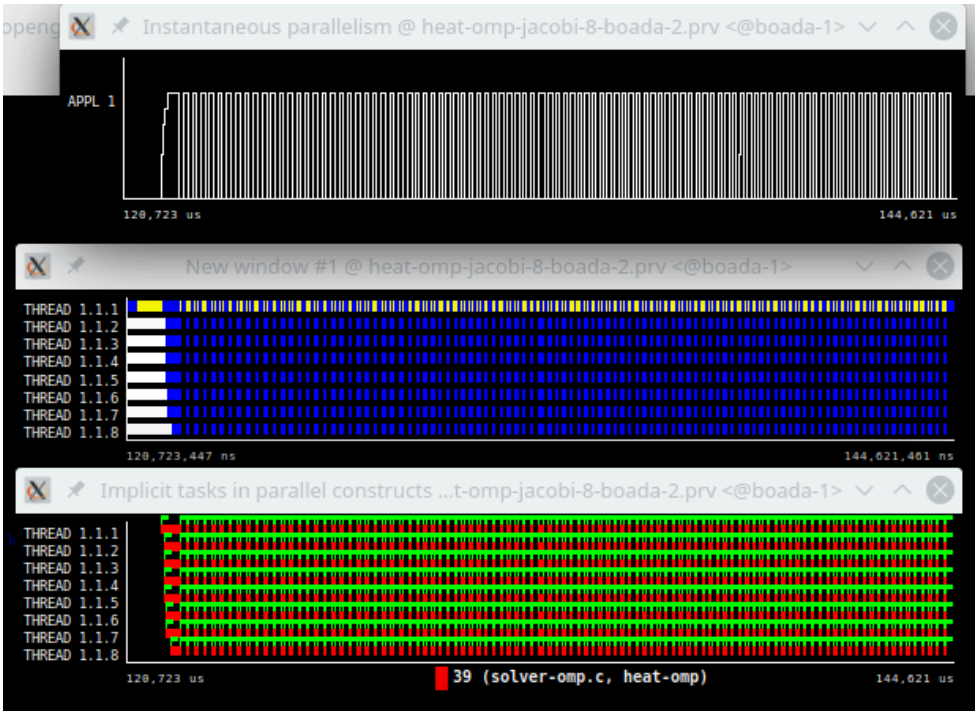
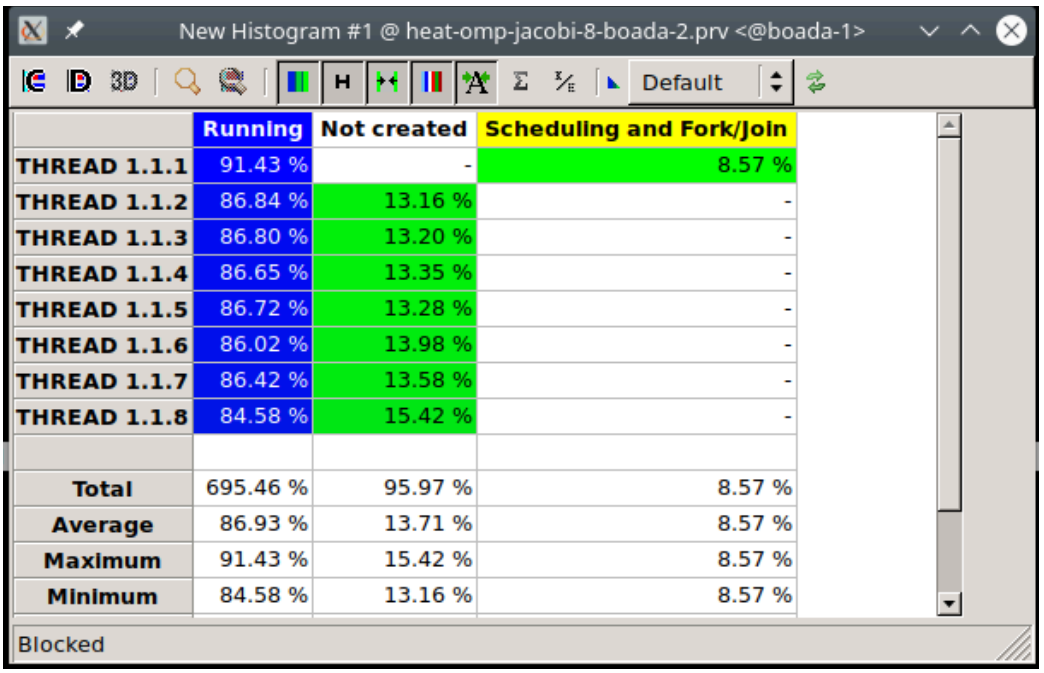


Figura 3.2: Timelines del paralelismo y trabajo de los threads según el tiempo



	Running	Not created	Scheduling and Fork/Join
THREAD 1.1.1	91.43 %	-	8.57 %
THREAD 1.1.2	86.84 %	13.16 %	-
THREAD 1.1.3	86.80 %	13.20 %	-
THREAD 1.1.4	86.65 %	13.35 %	-
THREAD 1.1.5	86.72 %	13.28 %	-
THREAD 1.1.6	86.02 %	13.98 %	-
THREAD 1.1.7	86.42 %	13.58 %	-
THREAD 1.1.8	84.58 %	15.42 %	-
Total	695.46 %	95.97 %	8.57 %
Average	86.93 %	13.71 %	8.57 %
Maximum	91.43 %	15.42 %	8.57 %
Minimum	84.58 %	13.16 %	8.57 %

Figura 3.3: Tabla con los porcentajes de tiempo de cada thread

En la Figura 3.2 de puede ver que los threads están trabajando en perfecto paralelismo pero con una particularidad, hay espacios en negro entre cada tarea donde no están haciendo nada y el tiempo es desaprovechado. Como muestra la figura 3.3, el porcentaje de tiempo en que los threads están trabajando ronda el ~86%, y el cúmulo de huecos “vacíos” en las timelines conforman un ~14% (que es bastante mejorable).

Como ya se ha dicho, aún falta por paralelizar la función copy_mat, que parece que son las barras blancas del principio en la figura 3.2 (en medio). Los cambios efectuados para solucionar esto están en el archivo solver-omp-jacobi-paralize.c.

La salida del programa con esta nueva versión y 8 threads es la siguiente:

```
par2311@boada-1:~/lab5$ cat *txt
Iterations        : 25000
Resolution        : 254
Residual          : 0.000050
Solver            : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 0.749
Flops and Flops per second: (11.182 GFlop => 14933.38 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Se observa que la salida es correcta (se ha comparado con diff) y el tiempo de ejecución es muchísimo menor al de la salida original con 8 threads (e incluso 4 threads, que era más rápida).

La pregunta es: ¿Por qué este nimio cambio ha mejorado tanto el programa? Utilizaremos una vez más Paraver para descubrirlo. En las siguientes figuras se observan los mismos gráficos que la versión de antes pero actualizados:

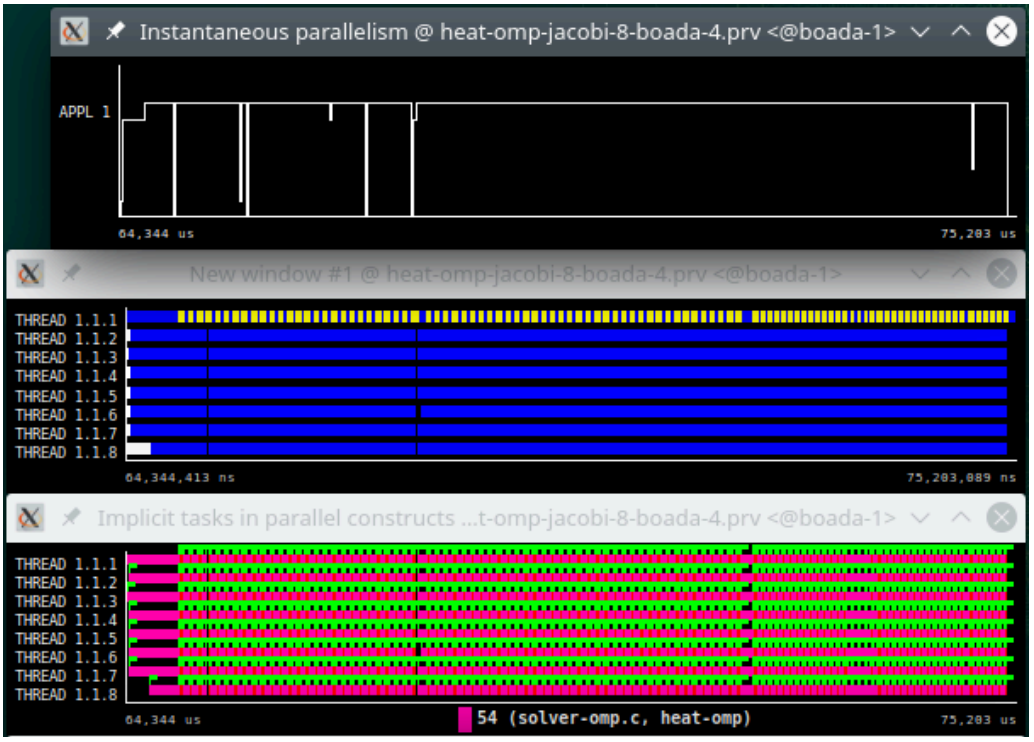
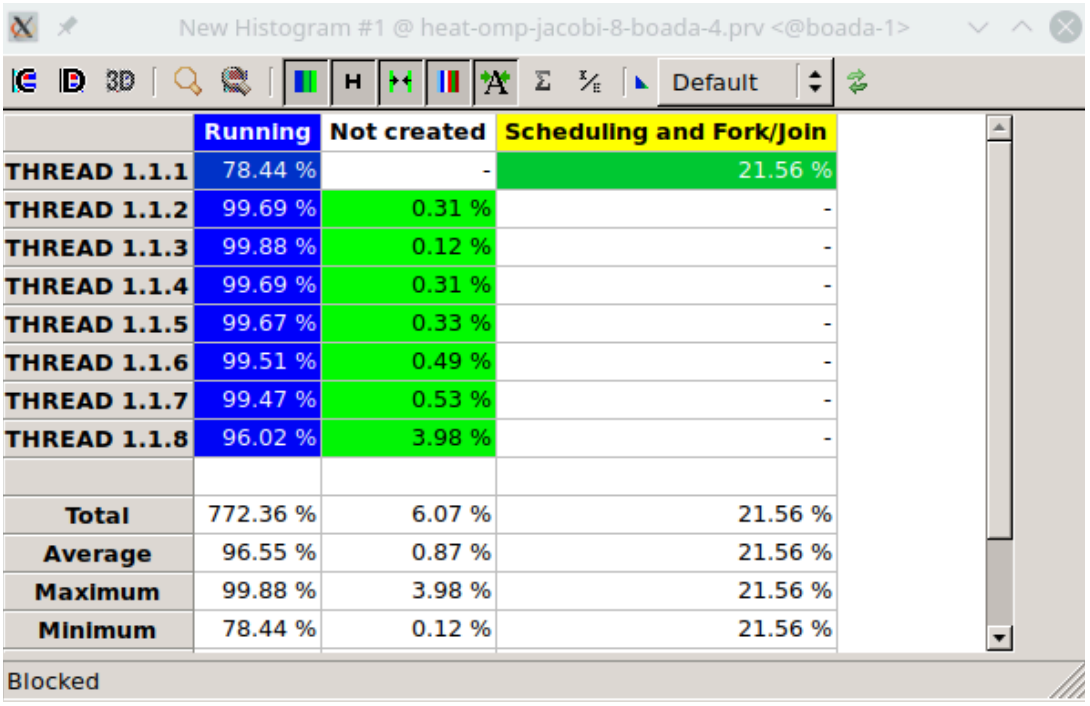


Figura 3.4: Timelines del paralelismo y trabajo de los threads según el tiempo (versión mejorada)



	Running	Not created	Scheduling and Fork/Join
THREAD 1.1.1	78.44 %	-	21.56 %
THREAD 1.1.2	99.69 %	0.31 %	-
THREAD 1.1.3	99.88 %	0.12 %	-
THREAD 1.1.4	99.69 %	0.31 %	-
THREAD 1.1.5	99.67 %	0.33 %	-
THREAD 1.1.6	99.51 %	0.49 %	-
THREAD 1.1.7	99.47 %	0.53 %	-
THREAD 1.1.8	96.02 %	3.98 %	-
Total	772.36 %	6.07 %	21.56 %
Average	96.55 %	0.87 %	21.56 %
Maximum	99.88 %	3.98 %	21.56 %
Minimum	78.44 %	0.12 %	21.56 %

Figura 3.5: Tabla con los porcentajes de tiempo de cada thread (versión mejorada)

Ahora que la función copy_mat ha sido paralelizada ningún thread tiene que estar esperando a nada, en la figura 2.4 se ve reflejado este cambio, ahora casi no hay espacios en negro de tiempo desperdiciado. Además en tabla de la figura 2.5 se aprecia la diferencia de porcentaje de tiempo en el que el thread está trabajando, que llega a ser casi el 100%.

Por último, para acabar de analizar este algoritmo simularemos la ejecución del programa con diferentes threads para discutir las diferencias de su escalabilidad respecto a la versión anterior. Los resultados están en la figura 2.6:

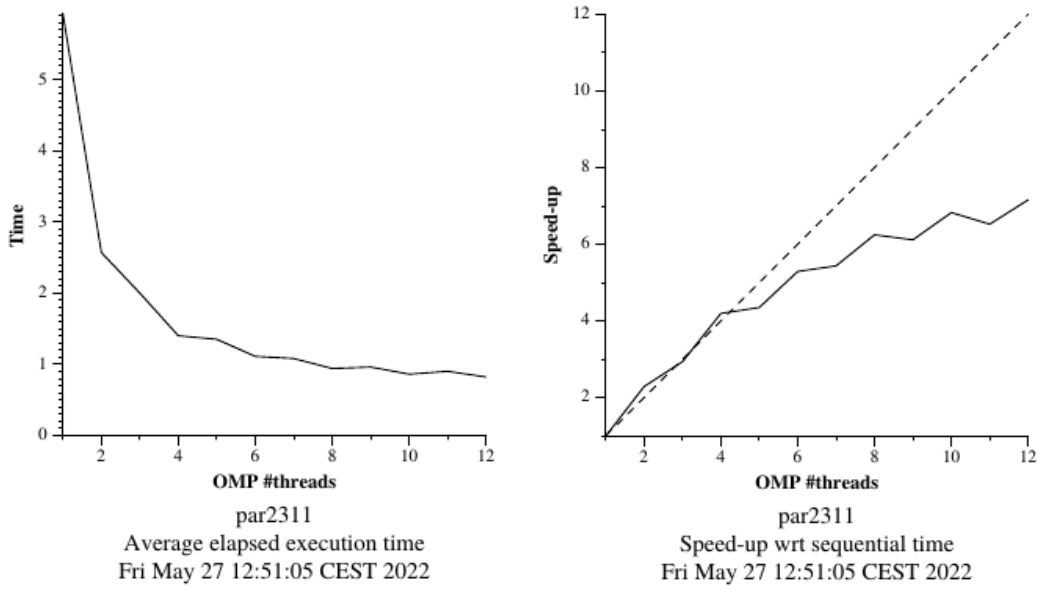


Figura 3.6: Gráficas que muestran el tiempo de ejecución y el Speedup según el número de threads (Jacobi, copy_mat paralelizado)

En este caso el tiempo de ejecución va disminuyendo muchísimo más que en la figura 3.1. Ahora que todo lo “serial” del programa ya está resuelto, por la ley de Amdahl se nota mucho más la mejoría. La gráfica del speedup también refleja este cambio ya que nunca llega a ser del todo plana, sinó que sigue una curva bastante bien definida hasta los 12 threads.

3.2 - Solver de Gauss-Seidel

Ahora que ya hemos analizado y paralelizado el algoritmo de Jacobi por completo nos enfocaremos en si podemos obtener el mismo rendimiento con el de Gauss-Seidel. El método será similar a la sección anterior.

Los cambios hechos en el código están en el archivo solve-omp-gauss.c.

Las modificaciones principales son:

- La igualación de nblocksj con nblocksi (para hacer una división de bloques cuadrada).
- La utilización de un vector auxiliar que nos dirá si cierta parte del programa está ya calculada o no.
- El uso de atomic read y atomic write para mantener la coherencia de resultados
- La comparación de si u == unew para saber si estamos usando el algoritmo de Gauss o no.

La salida del programa con estos cambios es la siguiente:

```
par2311@boada-1:~/lab5$ cat *txt
Iterations      : 25000
Resolution      : 254
Residual        : 0.000050
Solver          : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 1.894
Flops and Flops per second: (8.806 GFlop => 4649.37 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

El tiempo de ejecución con 8 threads es de 1,89 segundos, un poco más lento que la versión totalmente paralelizada del algoritmo de Jacobi pero aproximadamente la mitad de rápido que la versión sin paralelizar copy_mat. Tras usar diff en la terminal entre la imagen obtenida y la original se ha comprobado que el resultado es correcto.

Para saber de dónde proviene esta directa mejora del algoritmo de Jacobi (sin copy_mat paralelizado) utilizaremos la herramienta Paraver como hemos hecho las demás veces, viendo los datos más importantes. En las siguientes figuras se pueden ver estos mismos:

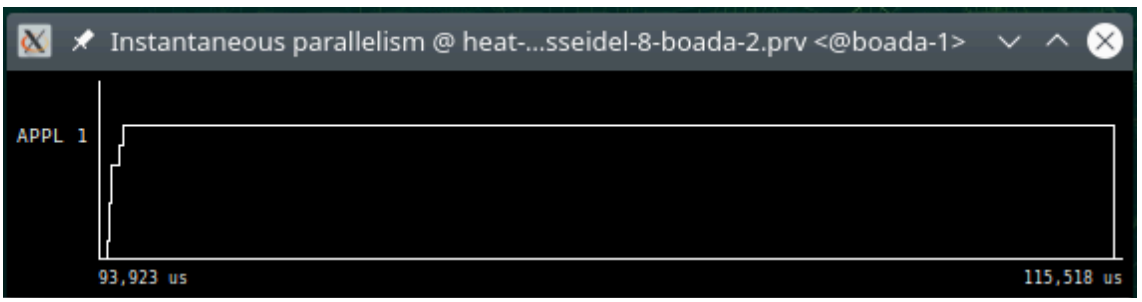


Figura 3.7: Gráfica del paralelismo según el tiempo

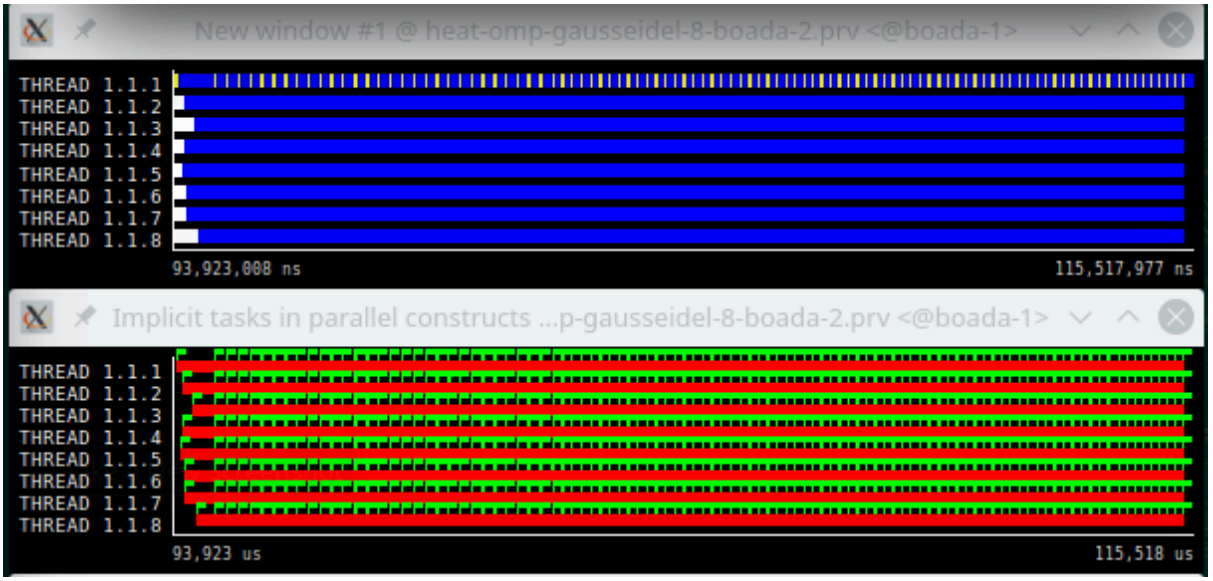
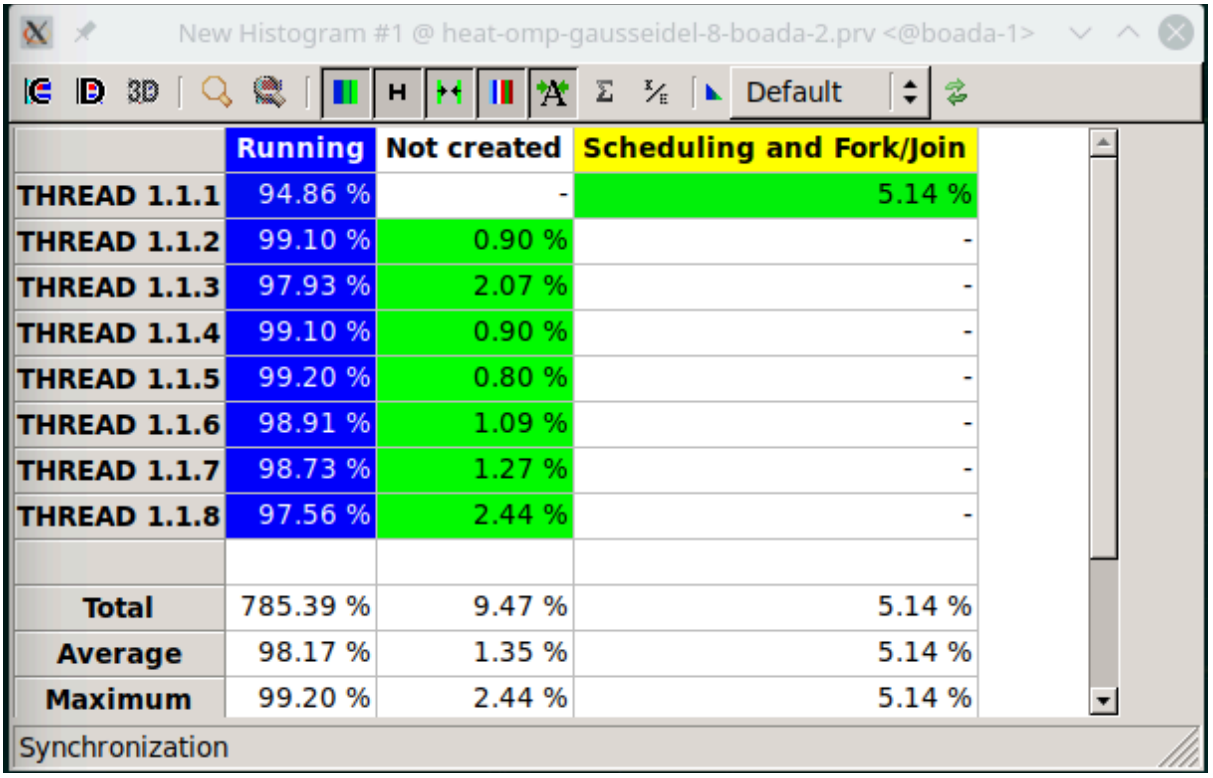


Figura 3.8: Timelines del trabajo de los threads según el tiempo



	Running	Not created	Scheduling and Fork/Join
THREAD 1.1.1	94.86 %	-	5.14 %
THREAD 1.1.2	99.10 %	0.90 %	-
THREAD 1.1.3	97.93 %	2.07 %	-
THREAD 1.1.4	99.10 %	0.90 %	-
THREAD 1.1.5	99.20 %	0.80 %	-
THREAD 1.1.6	98.91 %	1.09 %	-
THREAD 1.1.7	98.73 %	1.27 %	-
THREAD 1.1.8	97.56 %	2.44 %	-
Total	785.39 %	9.47 %	5.14 %
Average	98.17 %	1.35 %	5.14 %
Maximum	99.20 %	2.44 %	5.14 %

Figura 3.9: Tabla con los porcentajes de tiempo de los threads

Como se ve en las figuras 3.7 y 3.8, el tiempo en que los threads están esperando prácticamente no existe. Esto se ve reflejado numéricamente en la tabla de la figura 3,9 donde se muestra que casi el 100% del tiempo los threads están trabajando (un resultado similar al algoritmo de Jacobi totalmente paralelizado).

Esto rompe un poco las expectativas que teníamos con la figura 2.11, que mostraba un TDG más o menos vertical (insinuando secuencialismo). De todas formas, se ha comprobado que eso no es real y que el programa ya está muy bien paralelizado.

A continuación haremos un análisis de la escalabilidad del programa. después de usar el script submit-strong-omp.sh obtenemos los siguientes gráficos:

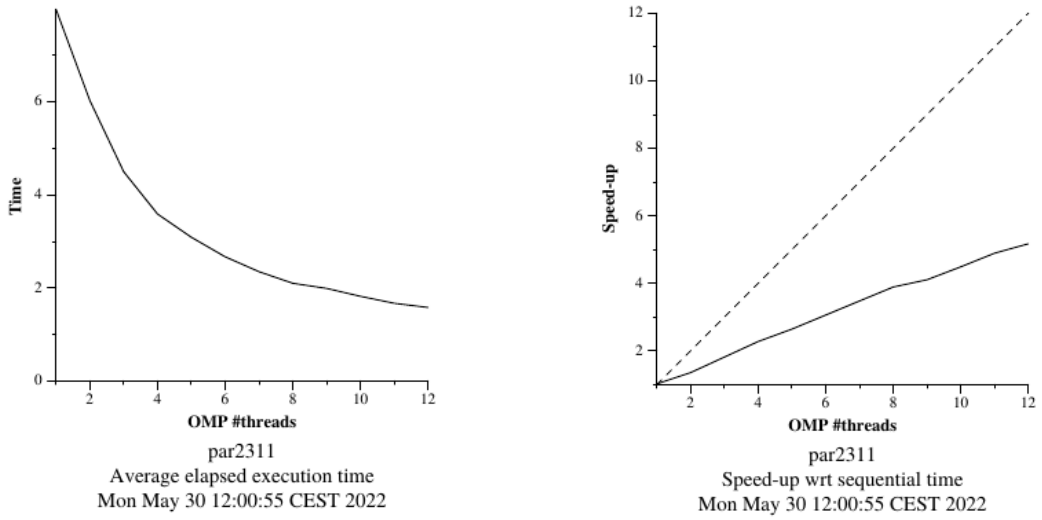


Figura 3.10: Gráficas que muestran el tiempo de ejecución y el Speedup según el número de threads (Gauss-Seidel)

Comparados con los gráficos de Jacobi, éstos no son tan buenos. El tiempo de ejecución parece que se queda estancado en los ~1,7 segundos a diferencia de los ~0,8 segundos del otro algoritmo, y el speedup parece crecer ligeramente más despacio a consecuencia de lo primero mencionado.

Para intentar mejorar el paralelismo de este algoritmo una opción válida sería jugar con los valores de la descomposición de datos por bloques, más específicamente con el número de bloques en la “j” ya que es el responsable de las dependencias por columnas y podría llevar a un false-sharing (dos threads escriben al mismo tiempo en la misma línea de memoria).

Para ello hemos hecho unas modificaciones en el código que se encuentran en el archivo solver-omp-gauss-user.c.

Los cambios efectuados han sido utilizar una variable externa “userparam” que nos limita los bloques en la j para así poder darle diferentes valores y encontrar el más óptimo.

Tras usar el script submit-userparam-omp.sh hemos obtenido diversos gráficos con el rendimiento del programa según “userparam”, esto ha sido hecho con varios números de threads para tener una más amplia cantidad de información al sacar conclusiones. Los resultados están en las siguientes figuras:

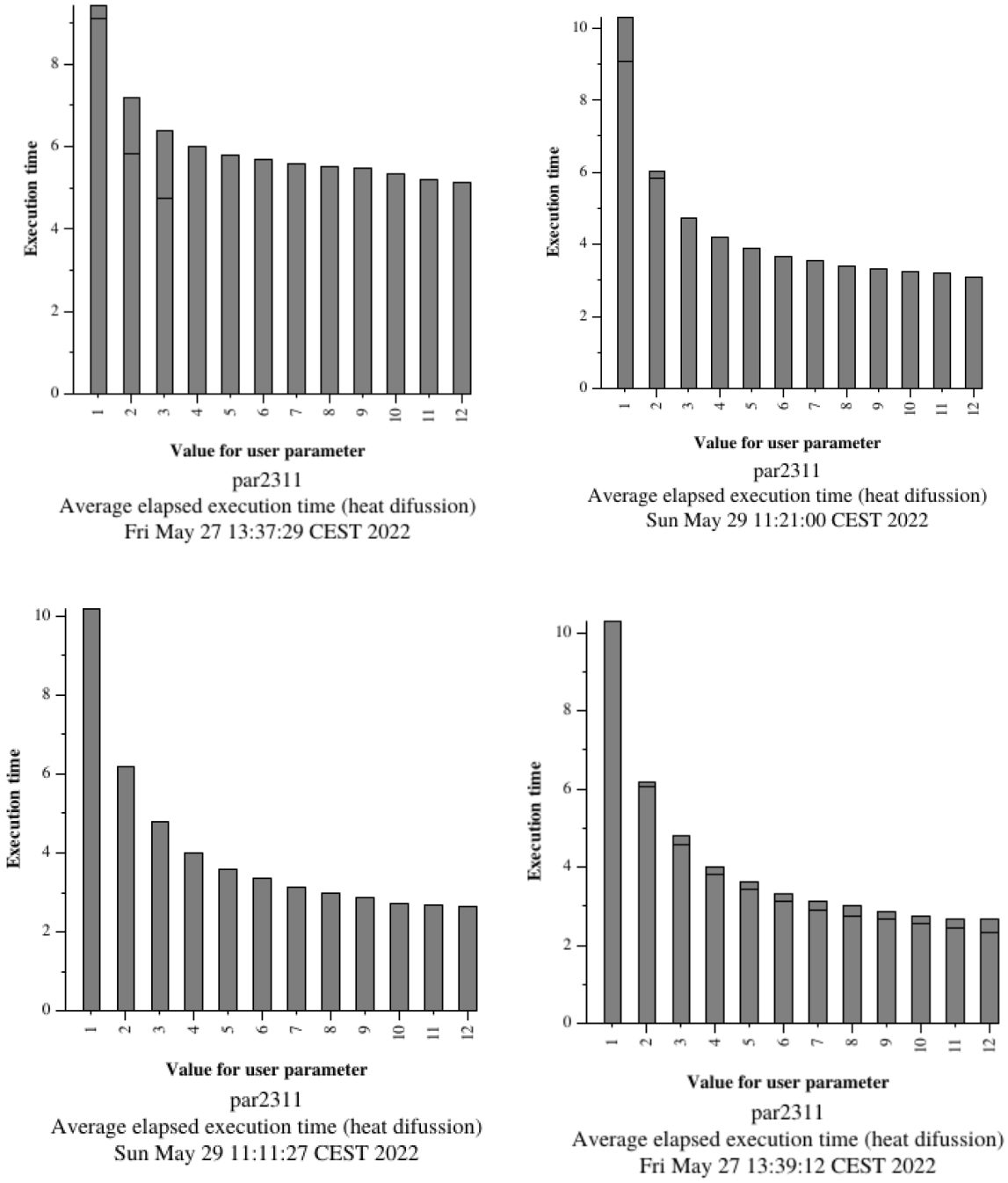


Figura 3.11: Gráficas del tiempo de ejecución según “userparam” (2 threads, 4 threads, 8 threads y 12 threads)

Se observa que a partir de que se supera el número de threads en cada caso el rendimiento deja de aumentar, además de que los 4 gráficos tienen una forma muy similar que no da indicios de cambio (y por tanto ninguna información que nos sirva a parte de lo ya mencionado).

Ahora probaremos a hacer que los bloques en la j sean proporcionales a los de la i, para ello modificaremos el código haciendo `nblocksj = userparam * nblocksi`, de esta manera podemos explorar lo que ya hemos visto pero ver además si el número de bloques en la “i” influye de alguna manera.

Los gráficos obtenidos son los siguientes:

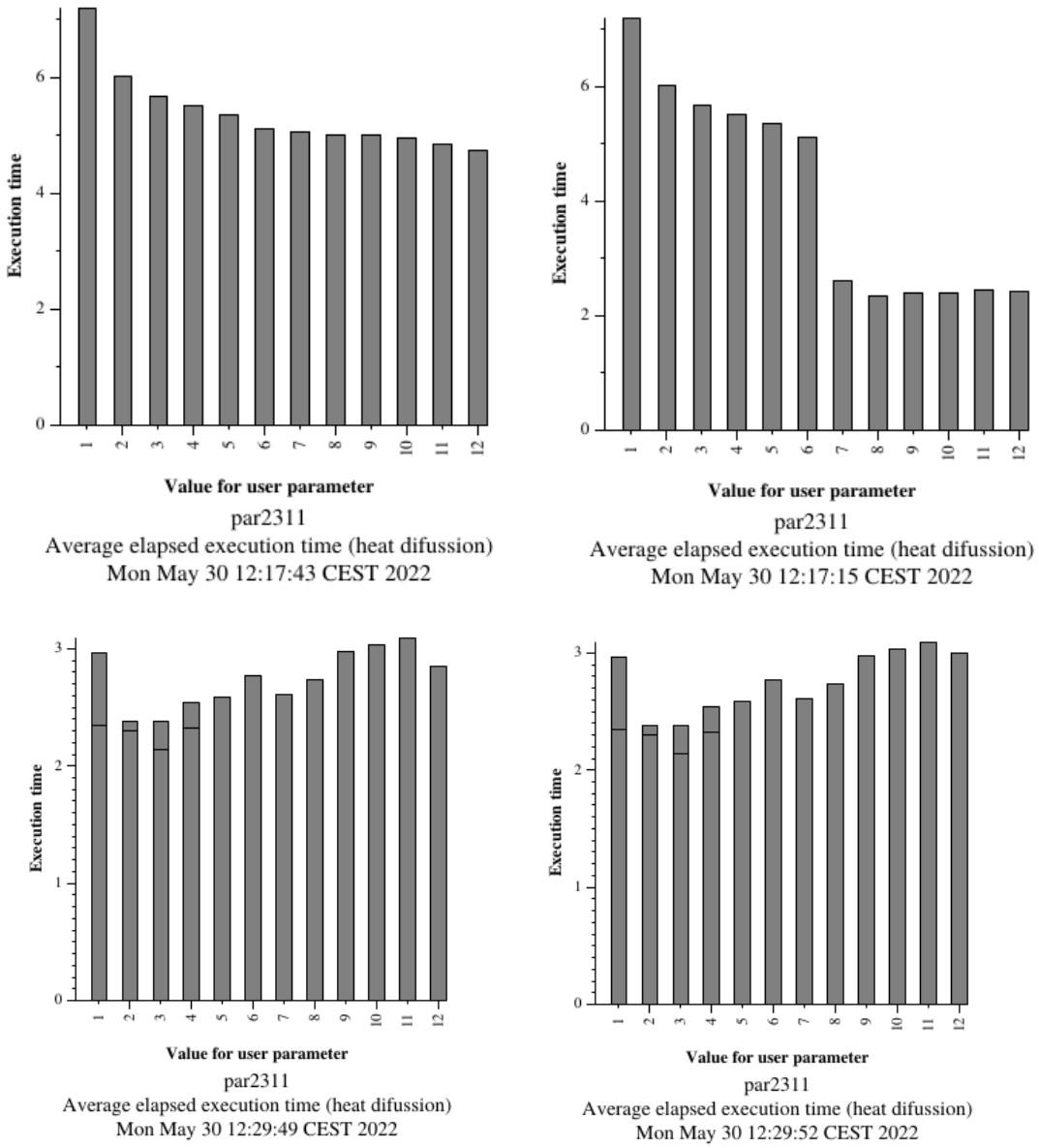


Figura 3.12: Gráficas del tiempo de ejecución según “userparam”, nueva versión (2 threads, 4 threads, 8 threads y 12 threads)

Se ve a simple vista que de esta forma el tiempo de ejecución va empeorando (cuando hay muchos threads) a medida que aumenta “userparam”, el motivo es porque la ganancia que proporciona aumentar esta variable se ve eclipsada por el alto número de threads que se tienen que repartir los bloques, que al hacer tareas requieren más sincronización (por ejemplo, en las gráficas de 8 y 12 threads se ve al inicio indicios de mejoría en el tiempo de ejecución con “userparams” pequeños que pronto dejan de ser mejoras y se convierten en overheads que dañan el resultado del tiempo de ejecución resultante).

4 - Conclusiones

En este deliverable hemos comparado dos algoritmos que hacen lo mismo y visto sus puntos fuertes y débiles en cuanto a formas de paralelizar. Además hemos experimentado y jugado con la descomposición de datos por bloques para asignar tareas, que ha dado en general buenos resultados.

Con esto hemos aprendido que no siempre hay una mejor paralelización, sinó que hay muchas posibles soluciones a cómo se plantean los problemas de paralelismo y hay que analizarlas todas. También hemos descubierto que un método que puede ser usado para paralelizar un programa es posible que no sirva para otro distinto, el paralelismo no es universal.