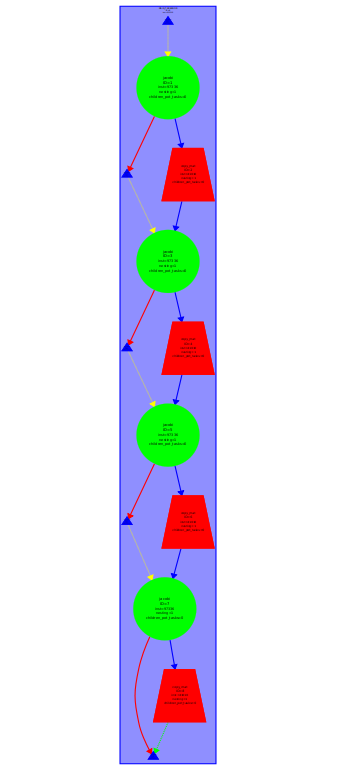
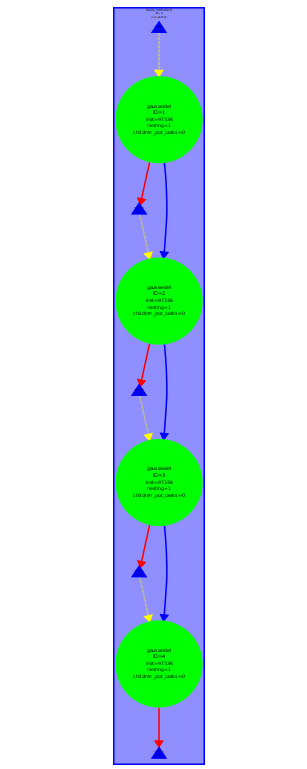
**2. Sequential heat diffusion program and analysis with Tareador**

**For the deliverable: Include the task dependency graph shown by Tareador. Is there any parallelism that can be exploited at this granularity level?**

Jacobi:

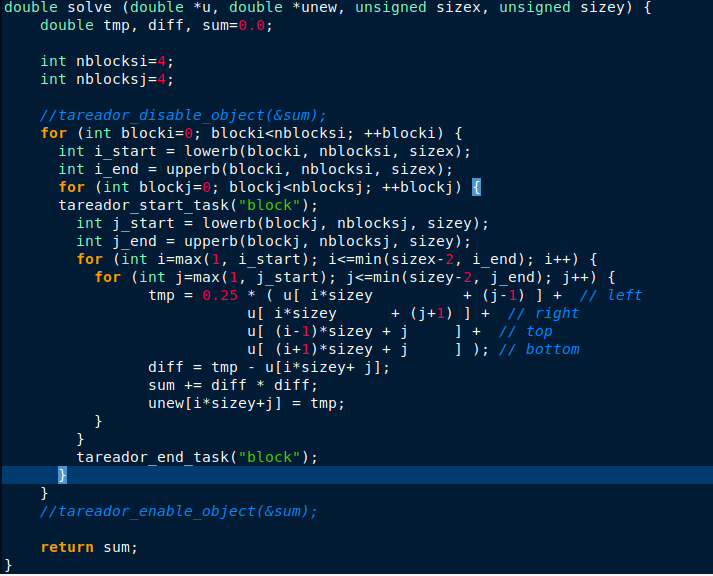
****

Gauss:

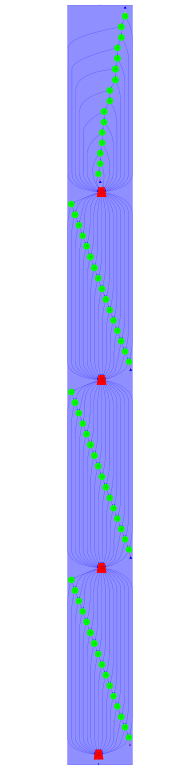


No se puede paralelizar con esta granularidad, para que se pueda paralelizar tendriamos que hacer una granularidad más fina .

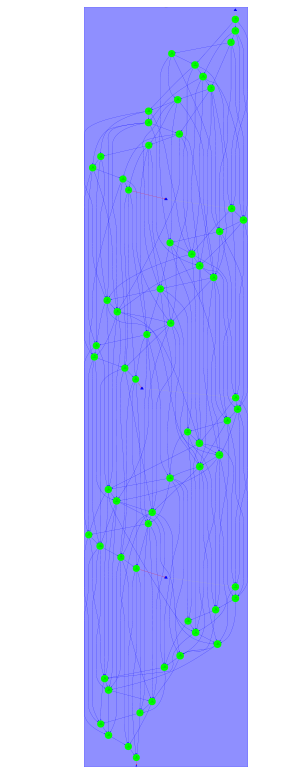
**For the deliverable: Include the excerpt of the code that you have modified in order to specify one task per block.**

****

Jacobi:

****

Gauss

****

**a) Which variable is causing the serialisation of all the tasks? Use the Dataview option in Tareador to identify it.**

Jacobi:

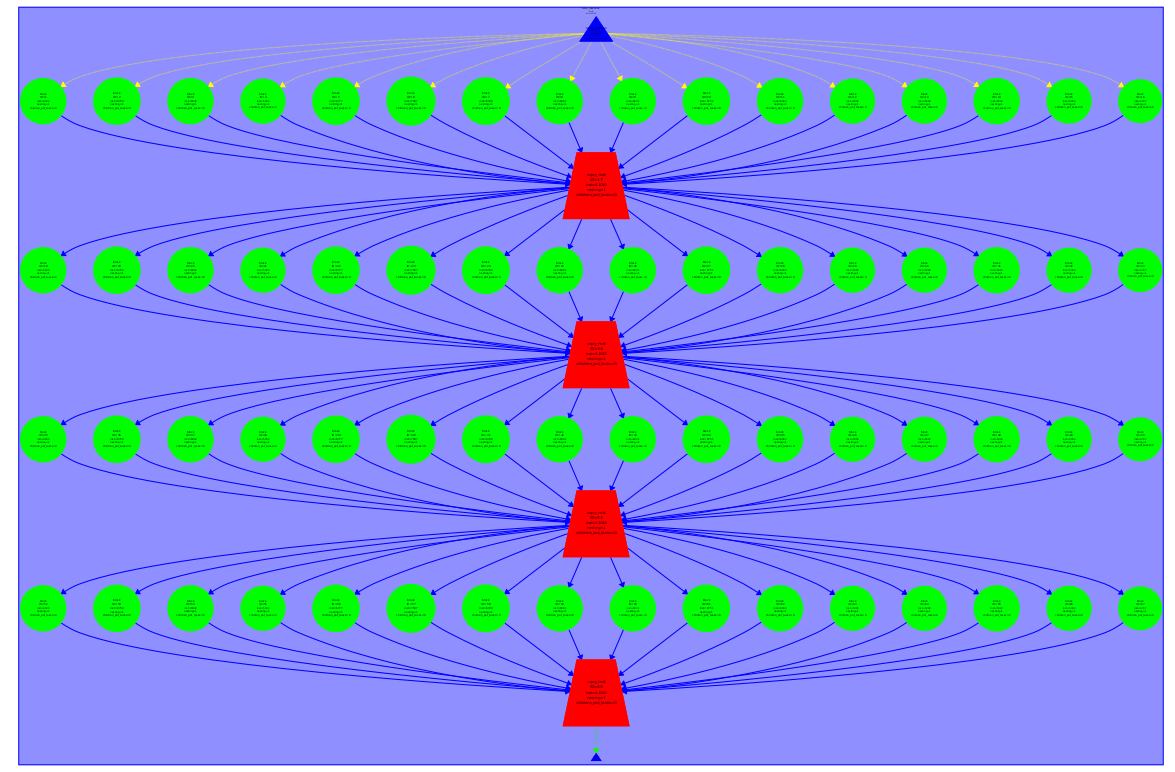
La variable que causa la dependencia es la variable sum

Gauss:

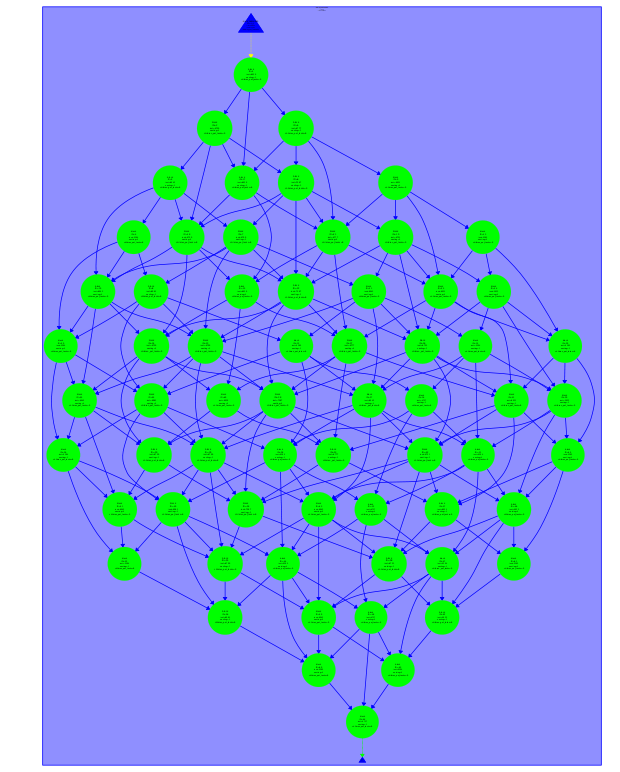
Las variables que causan la dependencia son la variable sum i la matriz

**(b) In order to emulate the effect of protecting the dependences caused by this variable, you can use the tareador disable object and tareador enable object calls, already introduced in the code as comments. With these calls you are telling to Tareador to filter the dependences caused by the variable indicated as object. Uncomment them, recompile and execute.**

Jacobi



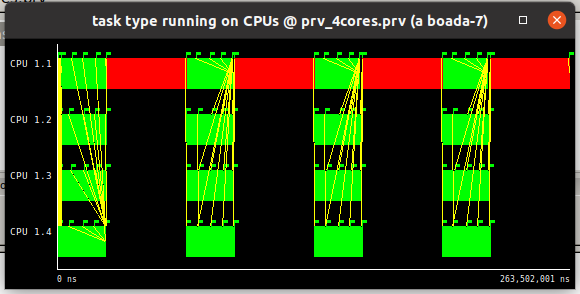
Gauss



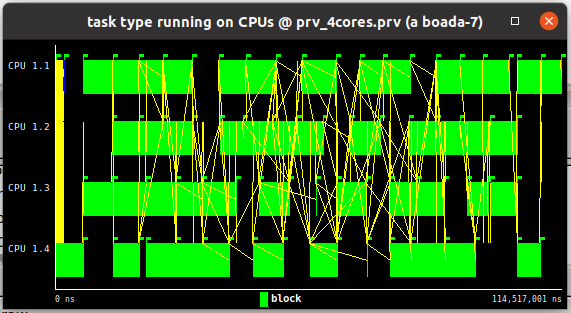
**(c) Simulate the execution of both solvers when using 4 threads. Is there any other part of the code that can also be parallelised (take into account this question for the parallel version!)?. If so, modify again the instrumentation to parallelise it.**

Aquí el disable y el enable esta descomentado:

Jacobi



Gauss

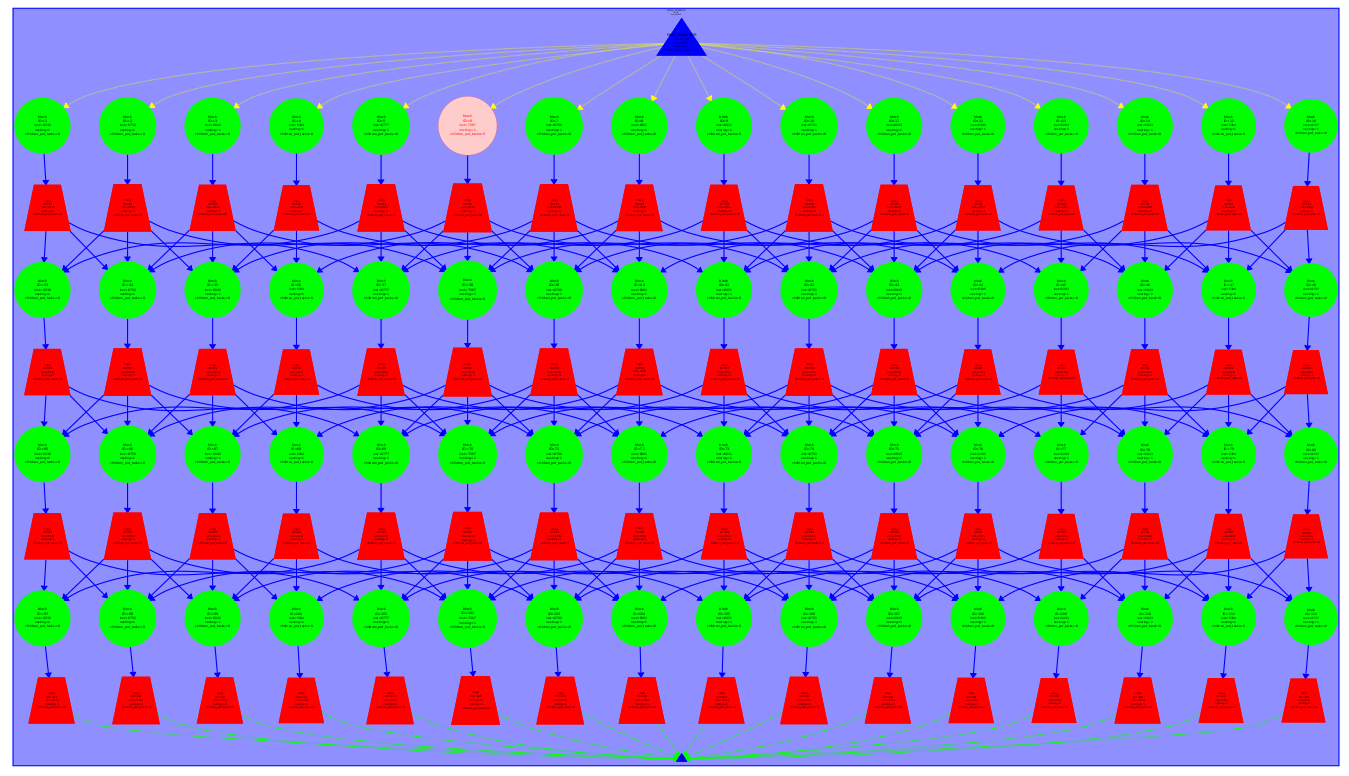
****

Podemos paralelizar la función copy para paralelizar mas el código como se ve en la simulación del jacobi.

**For the deliverable: Include the task dependency graph shown by Tareador after adding the dependences filter and the new tasks per block in other parts of the code to increase parallelism. Which variable was causing the serialisation of all the tasks? How will you protect the access to this variable in your OpenMP implementation? Are you obtaining more parallelism than in the previous version? Is the parallelism achieved the same for Jacobi and Gauss-Seidel ?**

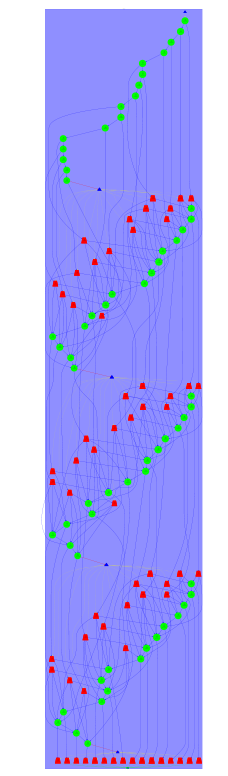
Con el disable descomentado:

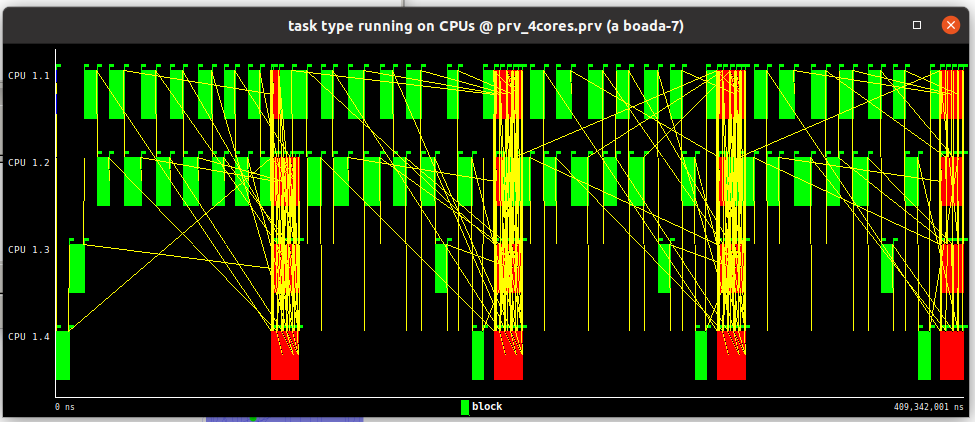
Jacobi

****

Con el disable comentado

Jacobi





La variable que estaba causando dependencias era la variable sum. Aunque la matriz también crea dependencia.

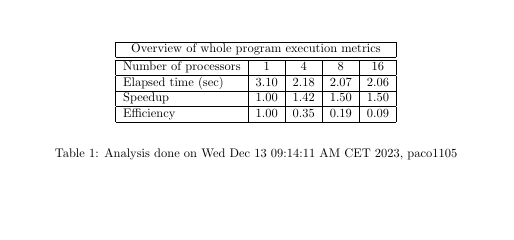
Para proteger la variable en OMP podemos utilizar ATOMIC.

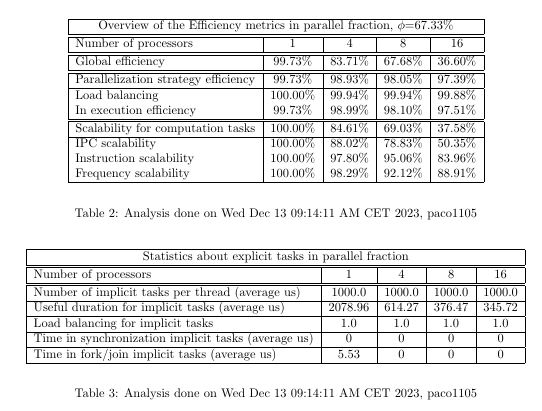
En Jacobi si que se llama a copy por tanto, al paralelizar la función copy, se ha obtenido mayor paralelización en Jacobi.

**3.1 Jacobi solver**

**3.1.2 Overall Analysis**

**For the deliverable: Include the modelfactors tables. Is the scalability that is obtained with this initial parallelisation appropriate? Which is the metric reported by mfLite.py that you should address first?**

****

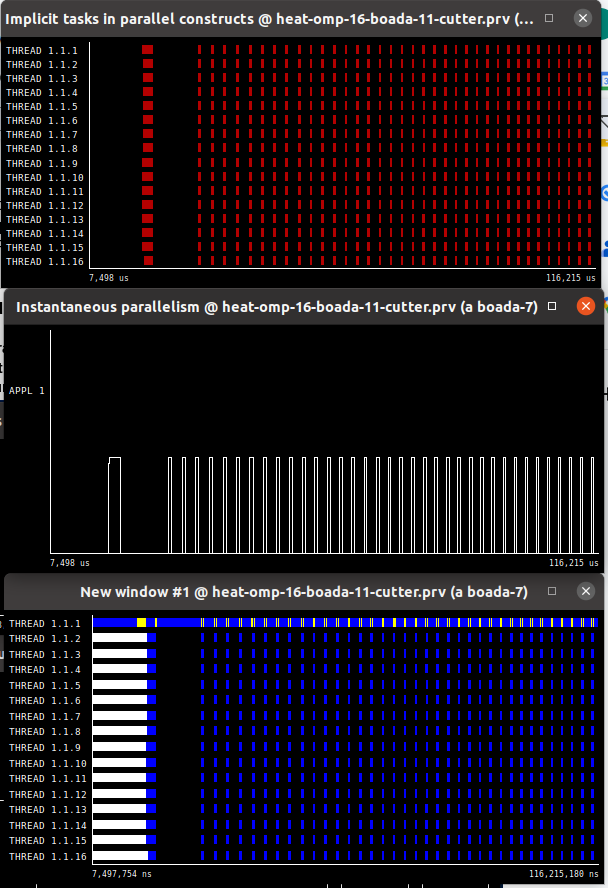
****

La scalibility obtenida no es nada buena ya que cuando aumentan los numeros de threads descende muchissimo, esto principalment se puede dar por el IPC scalability no es nada buena y con más threads baja mucho

Las métricas que deberíamos abordar principalmente tendria que ser la global efficiency ya que con muchos threads es muy baja. Despues tendriamos que tener en cuenta la IPC scalability que es muy baja con muchos procesadores.

**3.1.3 Detailed Analysis**

**For the deliverable: Include some captures of the window timelines to show the problem. What is the region of the code that is provoking the low value for the parallel fraction in your parallelisation? Hint: Remember the Tareador analysis you did.**

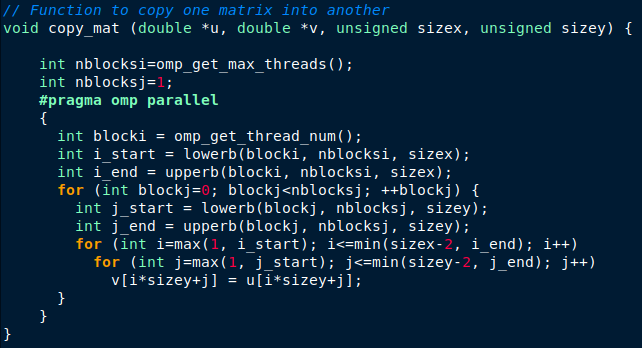
****

La parte del código que provoca la poca paralelización es la función copy. ja que esta no se paraliza.

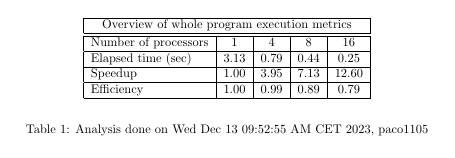
Como vemos en la última foto del tareador vemos que el thread1 es el encargado de hacer los copys y la parte que está en paralelo sería el solve. Podemos concluir que la parte que hace paralelizable(los solves) los hace muy bien como vemos en la primera foto, el problema es que no todo el código esta paralelizado.

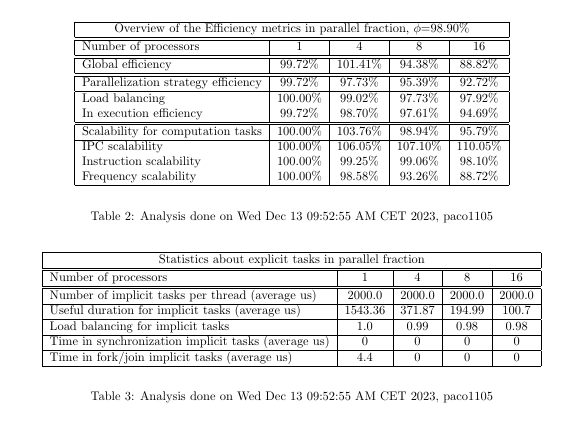
**3.1.4 Optimization**

**For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code.**

****

**For the deliverable: Include the modelfactors tables and the plot of scalability. Is the execution time reduced?. Have you increased the scalability? Is the parallel fraction larger than before? What is the speedup that you have achieved compared to your first implementation for 16 threads?**

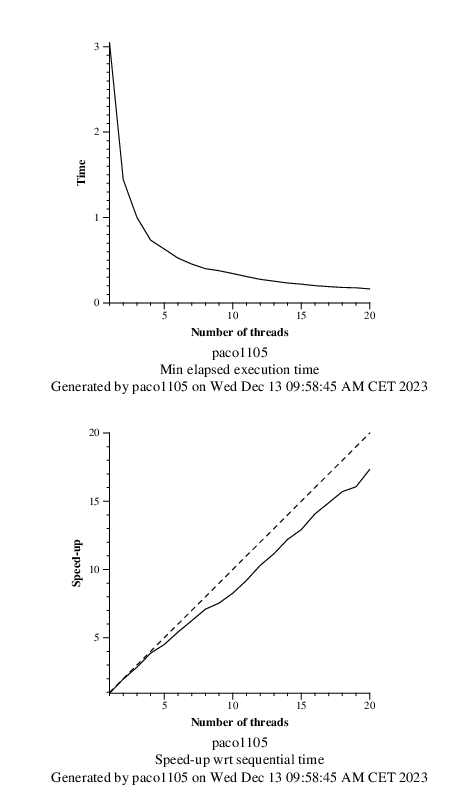
****

****

El execution time es mucho más bajo que la anterior versión sobre todo con más threads.

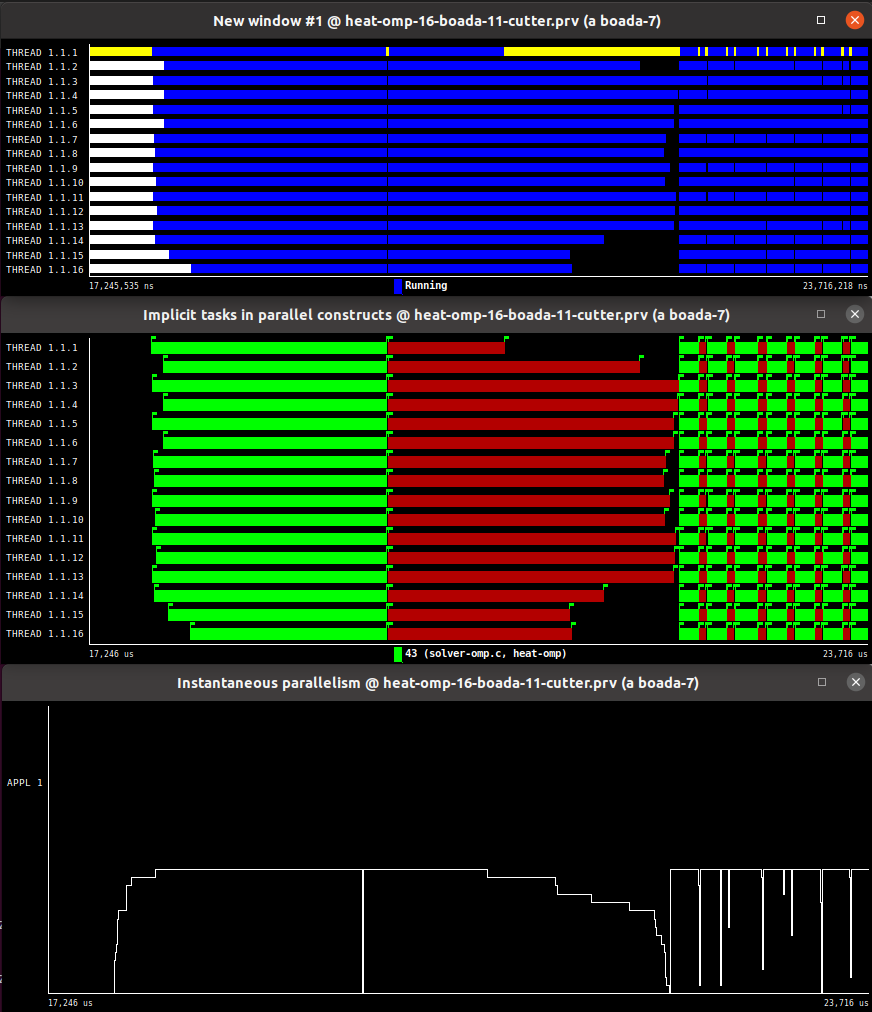
La fraccions paralela pasa a ser de 98%, es decir, prácticamente todo el programa esta paralelizado, en la anterior versión es mucho más bajo.

El speed up es de 12,60 lo que es muy alto y bueno, en la versión anterior es solo de 1,5. El speed up no es perfecto ja que estamos forzados a hacer el reduction del sum.

****

Como vemos el speed up es casi ideal, no es ideal por culpa mayoritariamente del reduction del sum.

**For the deliverable: Include some captures of the window timelines.**

****

**Podemos ver que prácticamente todo el codigo es paralelizable. i de vez en cuando hay bajadas en la instantaneous parallelism que serà cuando se hace el reduction del sum.**

**3.2 Gauss–Seidel solver**

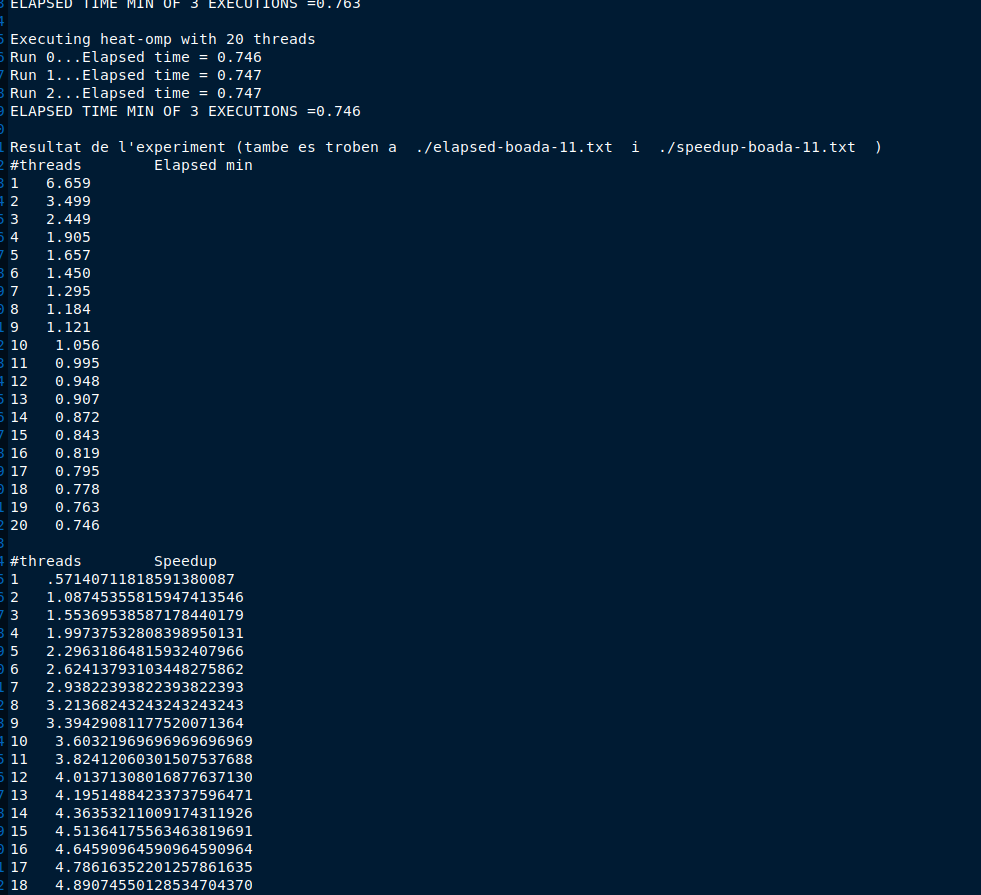
**3.2.1 First Implementation**

**For the deliverable: Include an excerpt of the code to show the modifications done.**

****

**3.2.2 Overall Analysis**

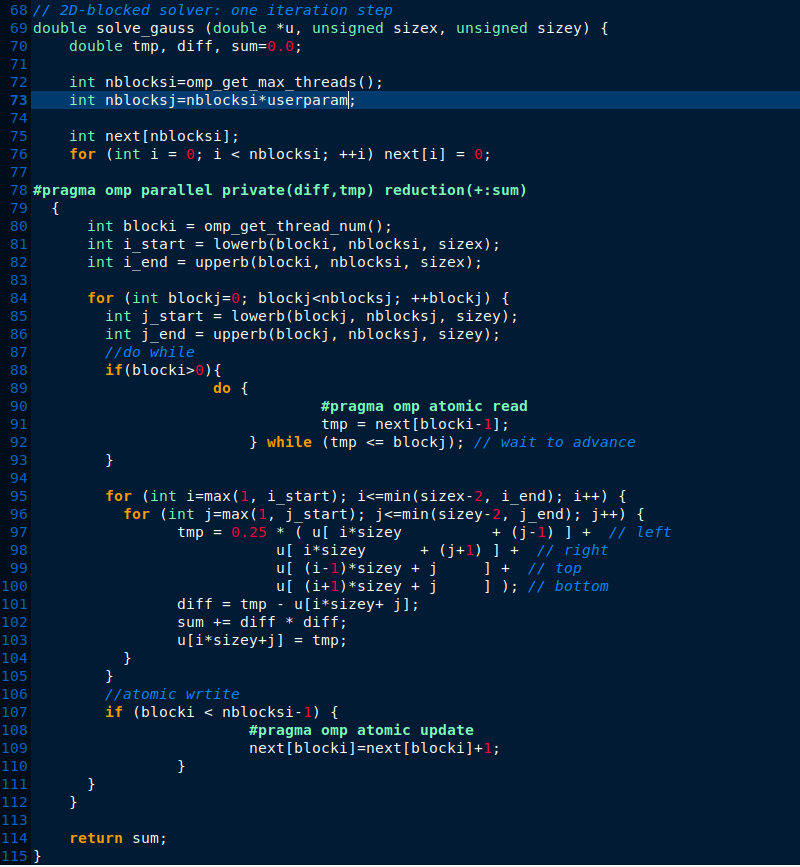
**For the deliverable: Include the plot obtained with submit-strong-omp.sh. Do you observe a linear speedup? Reason your answer. Note: Remember the analysis done with tareador with a simulation with 4 threads.**

****

En gauss seidel podemos observar una scalibilidad peor que la de Jacobi. Esto podria ser por culpa de la funcion copy\_mat.

**3.2.3 Detailed Analysis and Optimization**

**For the deliverable: Include an excerpt of the code to show the modifications done**

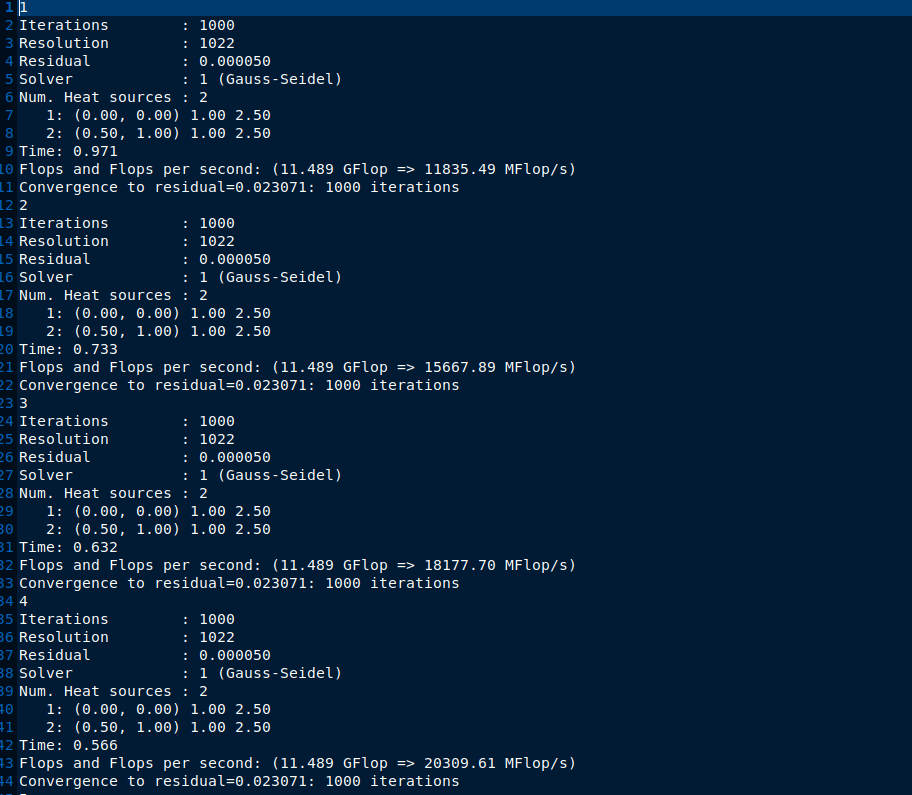
****

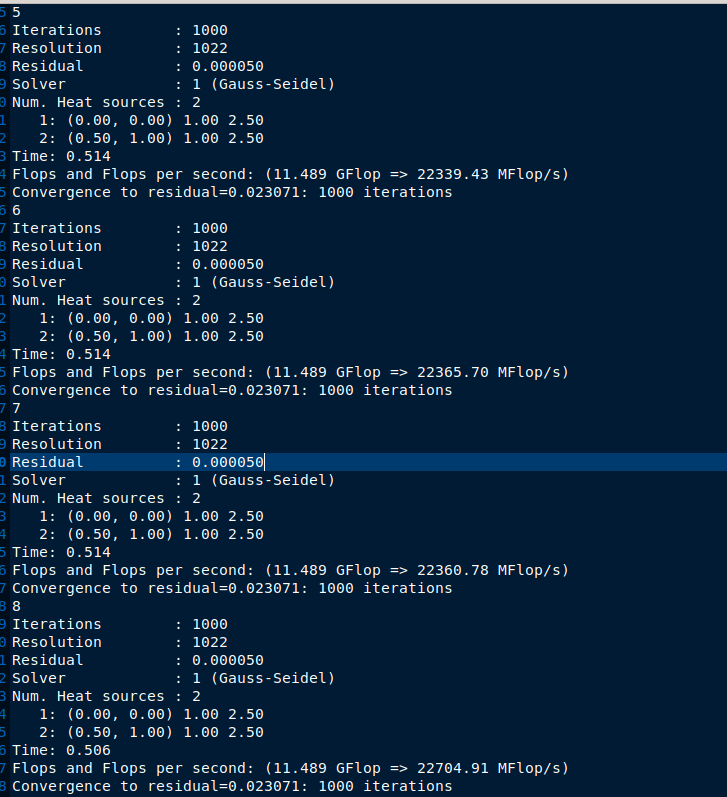
**3.2.4 Finding the appropriate value for the number of blocks**

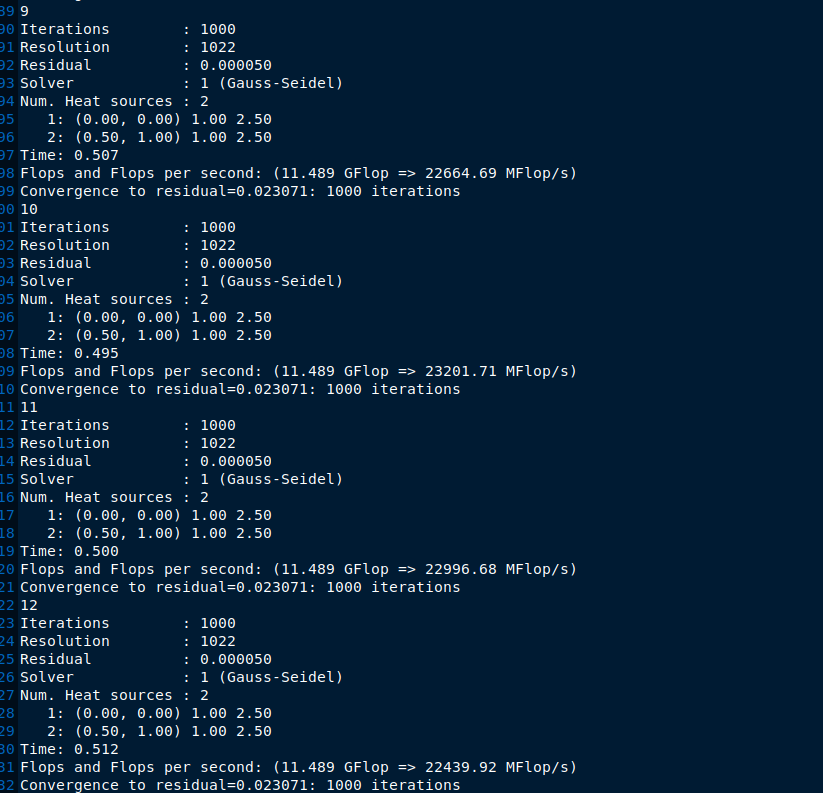
**For the deliverable: Include the plots obtained with submit-userparam-omp.sh and submit-strong-omp.sh using the best userparam and 16 threads. Reason why changing the number of blocks in the j dimension changes the ratio between computation and synchronisation. Compare the strong scalability for both**

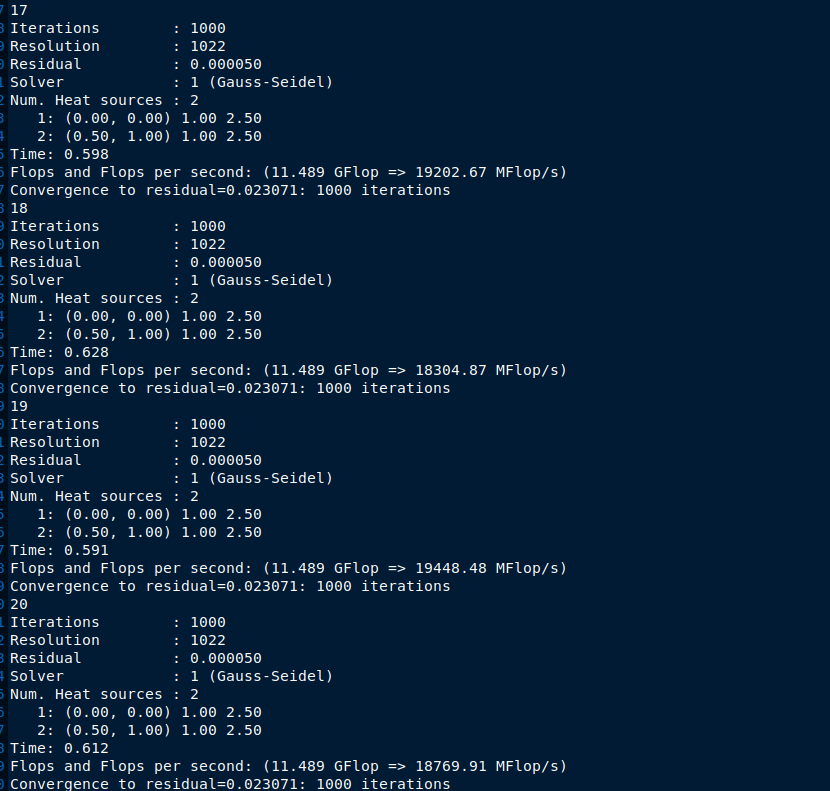
**cases nblocksj=20 and nblocksj=best userparam\*nblocksi (Finding the appropriate value for the number of blocks).**

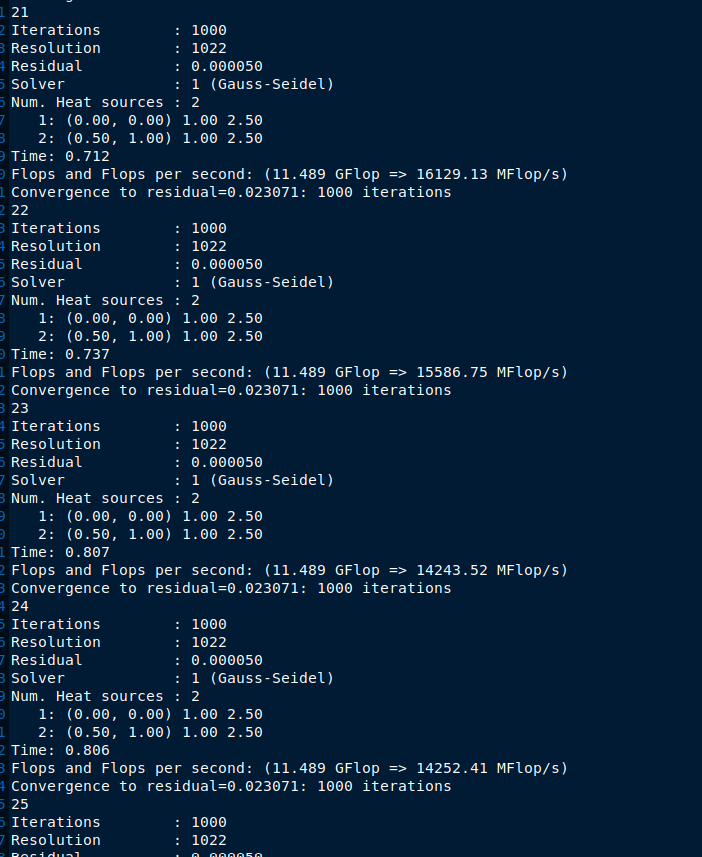
Ejecutando en h submit-userparam-omp.sh podemos ver que el mejor valor para el userparam es 10, ja que es cuando tarda menos.



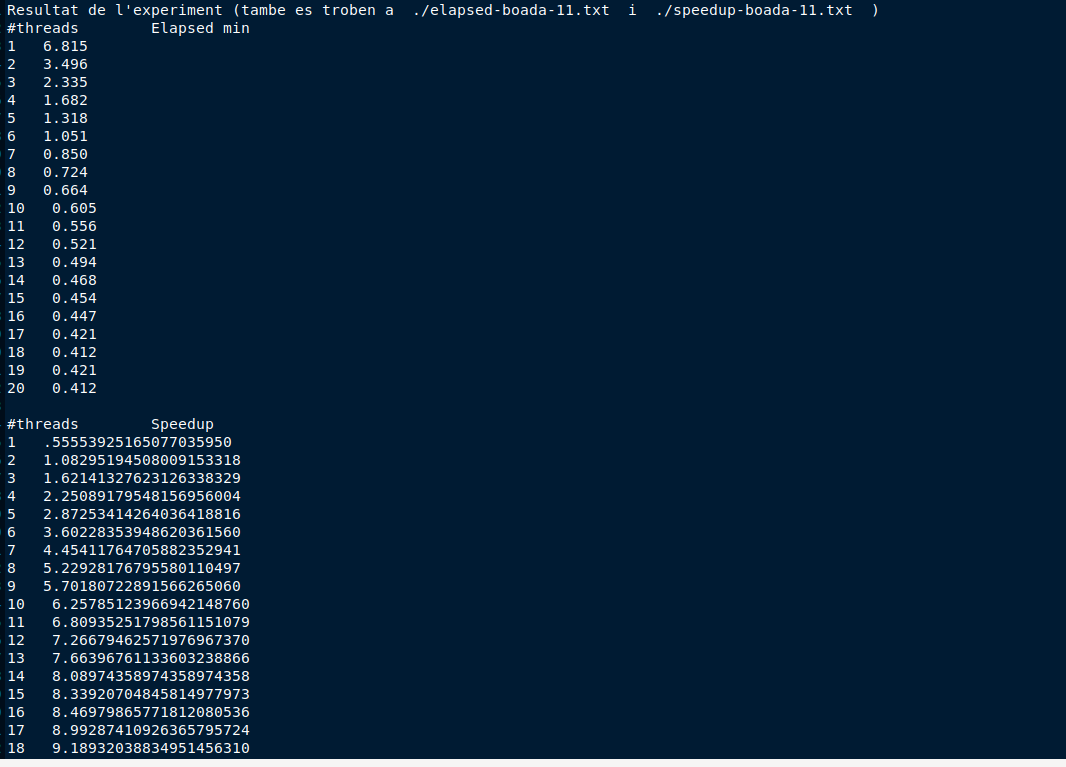








Ejecutando el submit-strong-omp.sh.

  
Podemos ver que el speed up mejora bastante que en la versión con nblocksj=20. Esto podría ser porque al ajustar el número de bloques en la dimensión j afecta la relación entre cómputo y sincronización porque modifica la distribución de la carga de trabajo y la cantidad de datos compartidos entre los hilos, lo que a su vez impacta en la eficiencia de la paralelización.