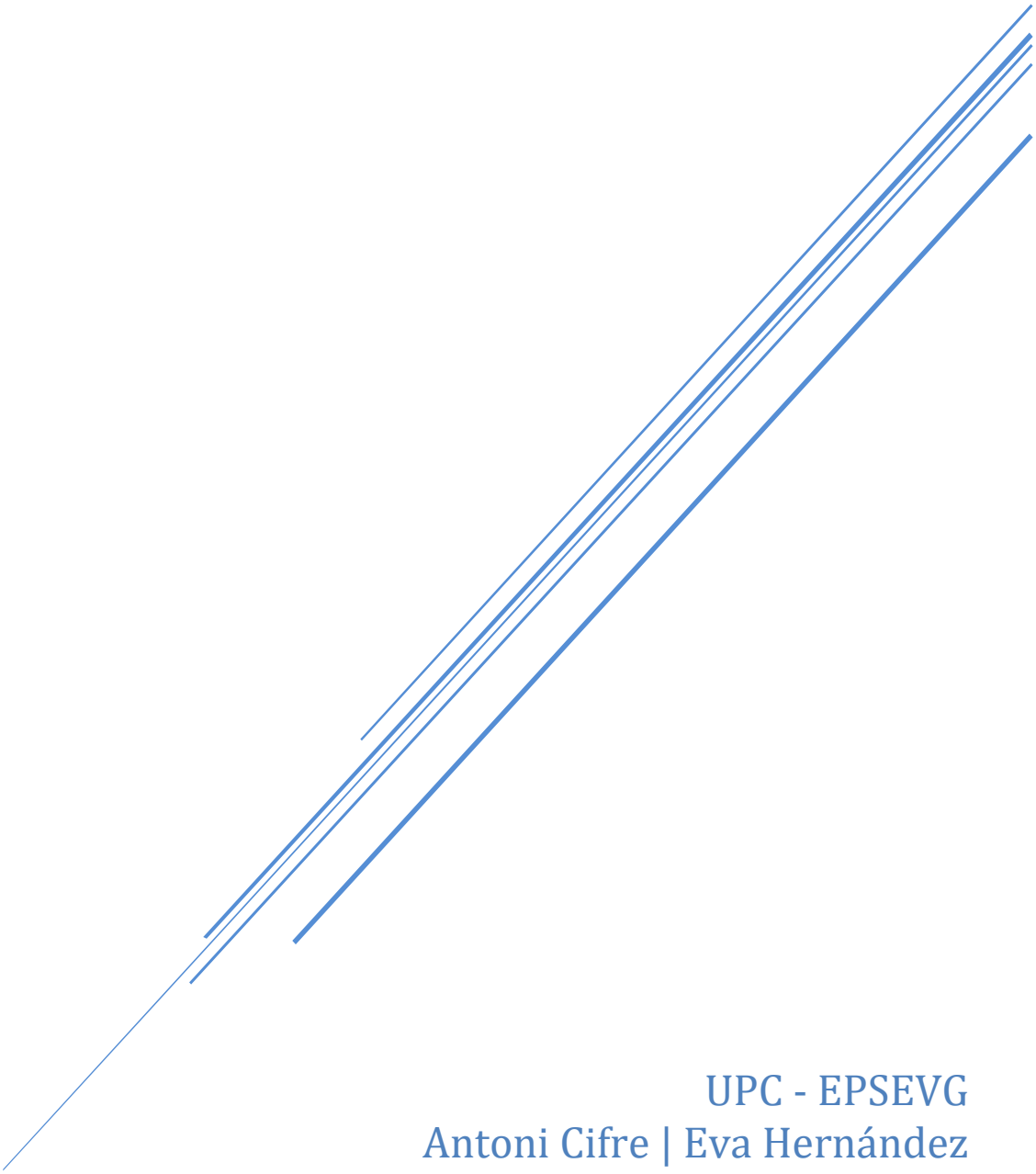
DELIVERABLE

LAB 5



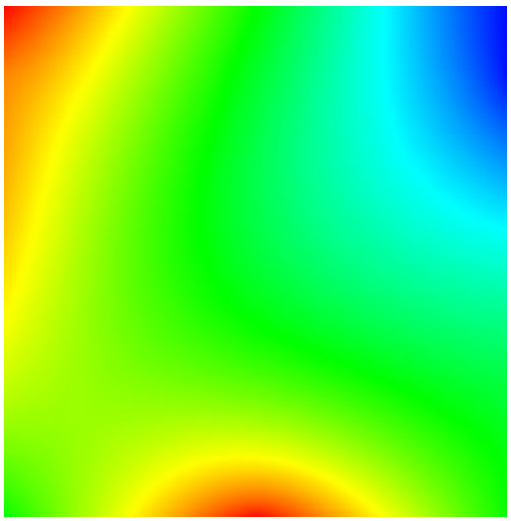
UPC - EPSEVG
Antoni Cifre | Eva Hernández
Fecha: 30/12/2019
3r año, 1r cuatrimestre

Contenido

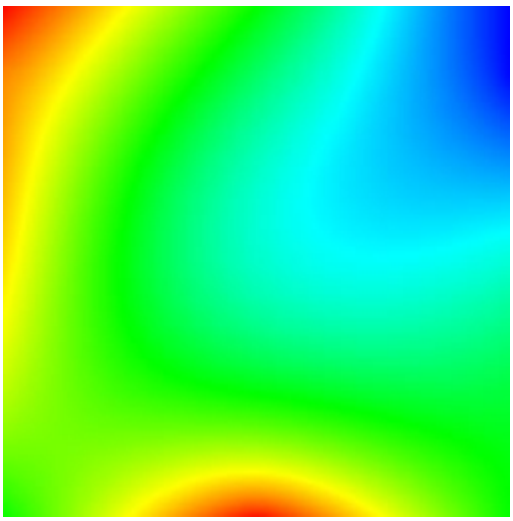
1 Sequential heat diffusion program

2. Change the solver from Jacobi to Gauss-Seidel by editing the configuration file provided (test.dat), execute again the sequential program and observe the differences with the previous execution. Note:the images generated when using the two solvers are slightly different (you can check this by applying diff to the two image files generated). Again, save the .ppm files generated with a different name, we will need them later to check the correctness of the parallel versions you will program.

Gauss

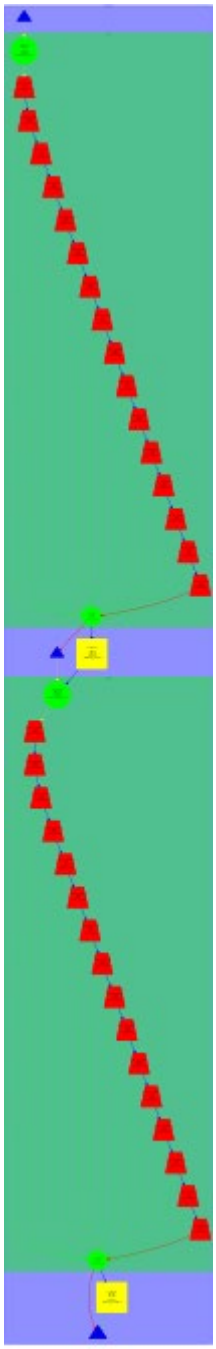


Jacobi



2 Analysis with Tareador

2. Next explore the dependences that happen when a much finer-grain task decomposition is used: one task for each iteration of the body of the innermost loop. Change the original Tareador instrumentation for Jacobi to reflect the new proposed task granularity. Compile again, execute and analyze the task graph generated.



(a) Which accesses to variables are causing the serialization of all the tasks? Use the Dataview option in Tareador to identify them.

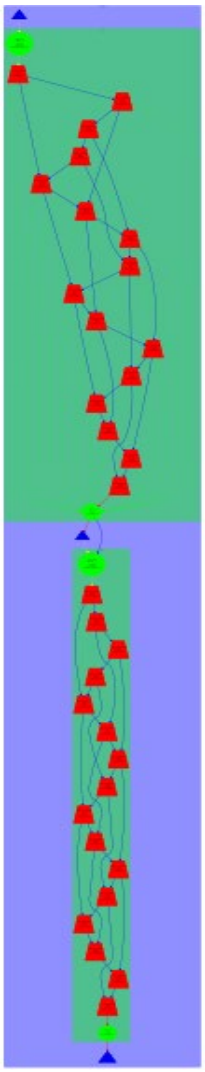
La variable sum.

(b) Use the appropriate calls to temporarily filter the analysis for the variables you suspect are causing the serialization and obtain a new task graph. Are you increasing the parallelism? How would you guarantee these dependences in your OpenMP parallelization if using a parallelization strategy based on #pragma omp for?

Para resolver estas dependencias utilizando pragma omp for, utilizaríamos el reduction sum.

3. Repeat the process with the Gauss-Seidel solver, identifying the causes for the dependences that appear between the tasks. How would you guarantee these dependences in your OpenMP parallelization if using a parallelization strategy based on #pragma omp for? What if based on #pragma omp task?

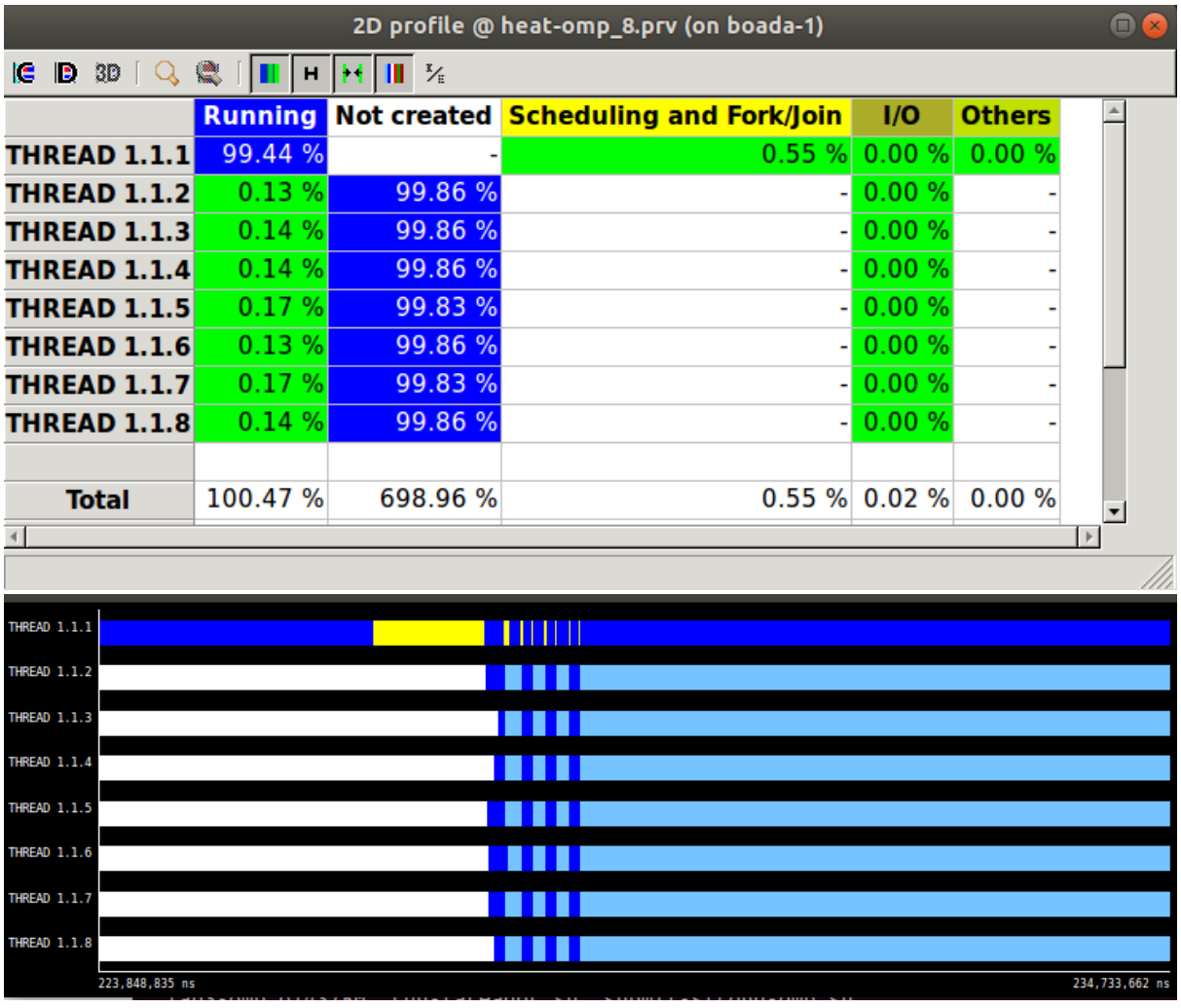
En este caso tendríamos dos variables, las cuales nos generan una serialización. La variable sum i la matriz u.



3 Parallelization of Jacobi with OpenMP parallel

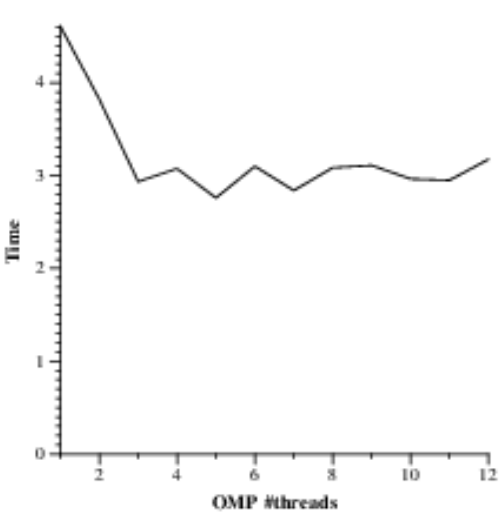
3. Instrument the execution of the binary with Extrae by submitting the submit-omp-i.sh script. Notice we are using instrum.dat as the configuration file for the Extrae instrumented executions (which just performs four iterations on a problem with the same resolution as the one defined in test.dat). Reason about the behavior observed, trying to answer the following questions: is the parallel execution appropriate for 8 threads? Why the Jacobi solver does not benefit from using 8 threads? Modify the source code to avoid this bottleneck and execute again without and with instrumentation. Is the parallel efficiency improving?. Is there a load balancing problem?

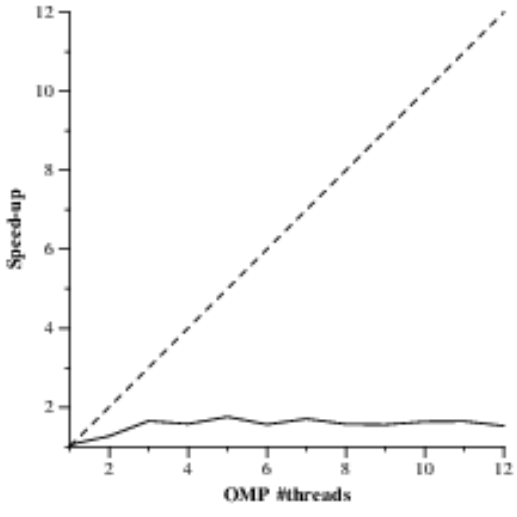
Paraven



El instrum data, al solo hacer 4 iteraciones, si comparamos con todo el trabajo que hacen las otras funciones no paralelizadas, es insignificante el trozo paralelizado, por lo tanto, sí que es mejor usando los 8 threads pero solo se usan en un breve lapso de tiempo.

Scalability

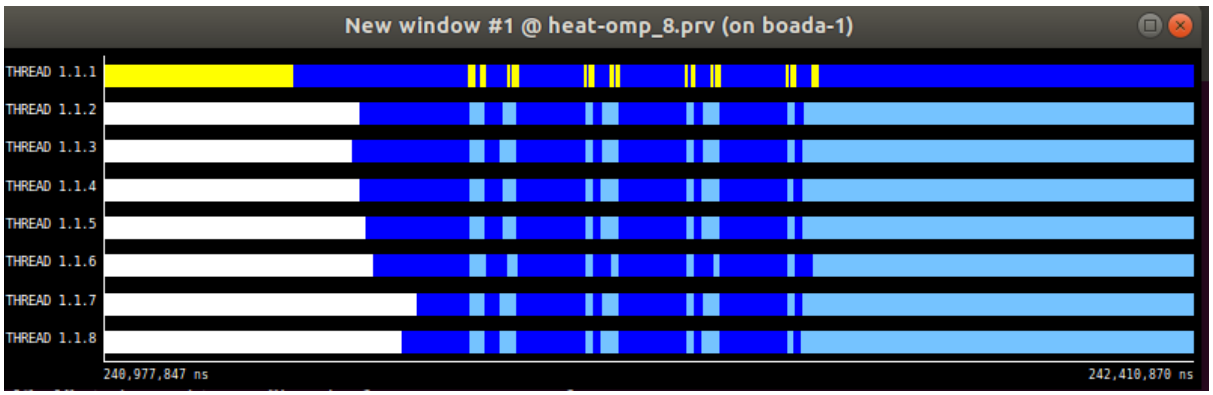


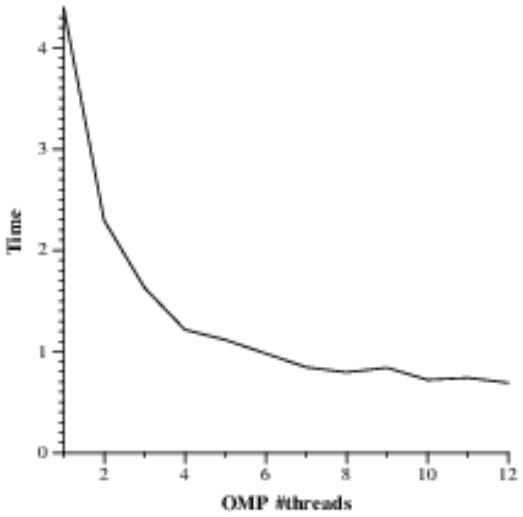


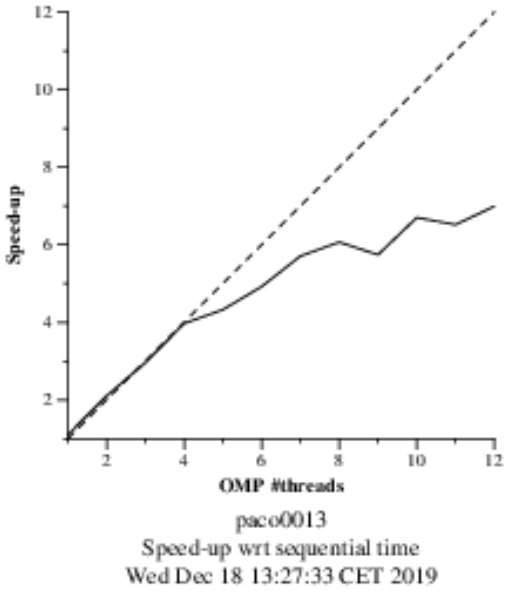
El actual problema es que al solo hacer 4 iteraciones en la función de Jacobin, obtendremos un cuello de botella bastante importante en las otras funciones, con lo cual pasaremos a paralelizar dicha función para evitar el cuello de botella y poder observar mejor la escalabilidad de la función de Jacobin.

4. Is there any serious serialization in your parallel execution? Parallelize other parts of the code, or simply rewrite them in a different way, in order to improve even more the efficiency of your parallel code. Execute again without and with instrumentation in order to see the new behavior. Do the load balancing and/or code serialization problems persist?

Al analizar el código hemos observado que el cuello de botella reside en la función copy_mat, con lo cual la hemos procedido a paralelizado y hemos obtenido el siguientes resultados.







Como ahora sí podemos observar, el tiempo de paralelización es mucho mejor, además, el speed up es perfecto hasta los 4 threads debido a que solo se hacen 4 iteraciones en la función de Jacobin.

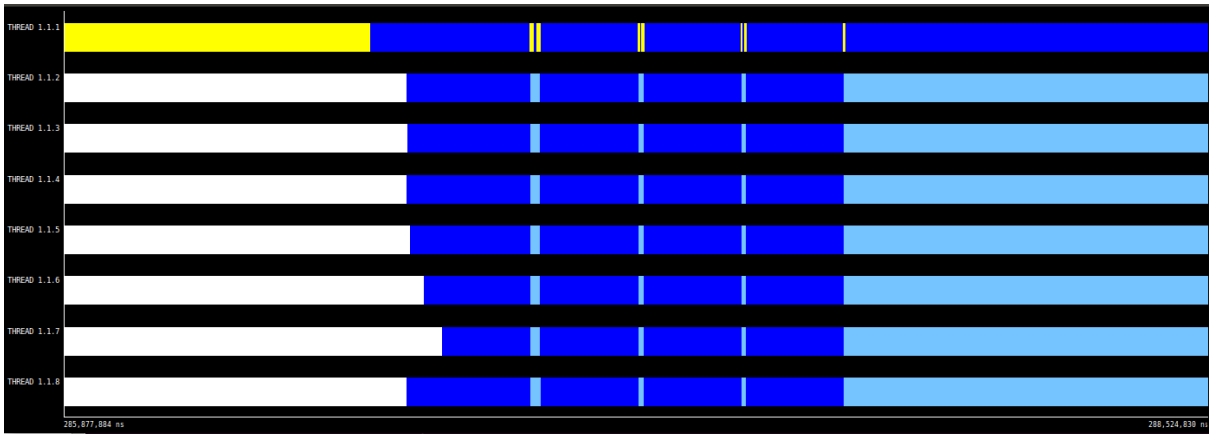
5. Once you are satisfied with the parallel behaviour observed, use the submit-strong-omp.sh script to queue the execution of heat-omp and analyze the scalability of the parallelization for different number of processors (1 to 12). Reason about the scalability that is observed.

Al paralelizar la función copy_mat podemos observar que la escalabilidad mejora muchísimo y el speed up es perfecto hasta los 4 threads, respecto a cuando no estaba paralelizado.

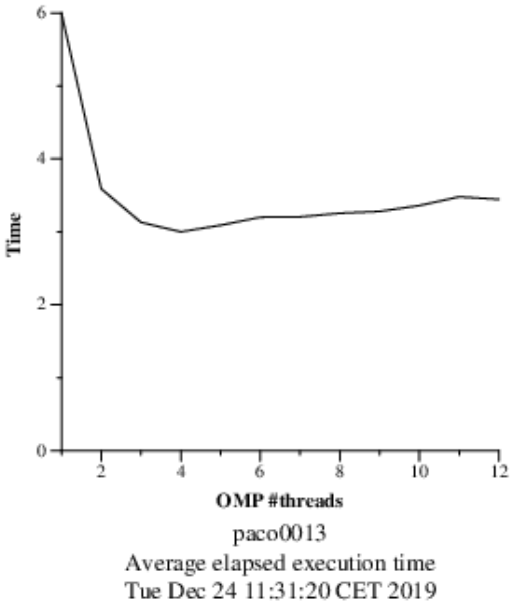
4 Parallelization of Gauss-Seidel with OpenMP ordered

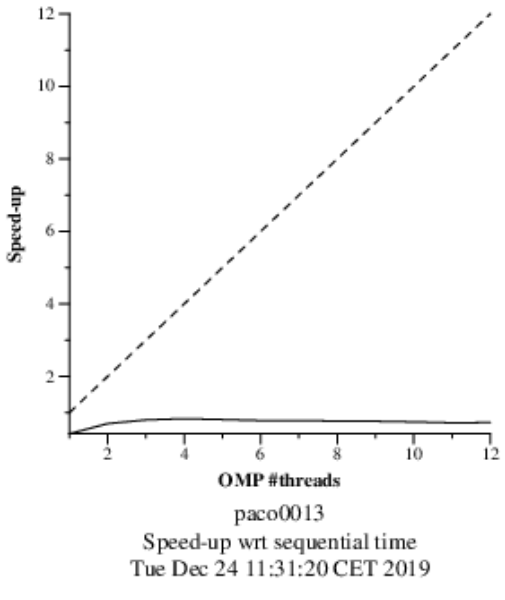
2. Instrument with Extrae by submitting the submit-omp-i.sh script and visualize the traces generated for the parallel execution. Does the parallel behaviour match your expectations?

Coincide con nuestras expectativas ya que en esta primera prueba, hemos echo el blocking de 4 columnas por proceso.



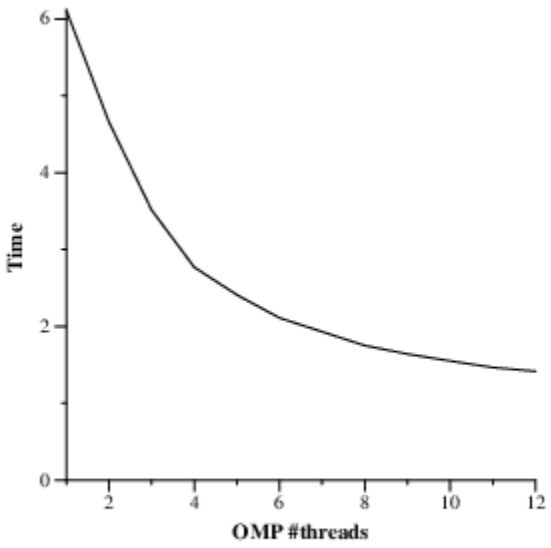
3. Once the parallelization is correct, use the submit-strong-omp.sh script to queue the execution and analyze the scalability of the parallelization.

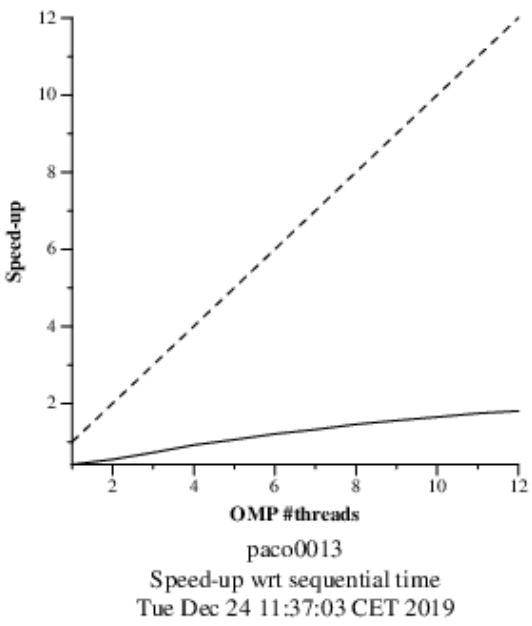




Podemos observar que la escalabilidad es muy mala, al realizar un par de pruebas nos hemos dado cuenta de que la mejor manera sería variando el nombre de bloques

dependiendo del número de threads de los que disponemos, de esta forma esperamos tener una escalabilidad más uniforme.





Una vez modificado el nombre de bloques generados en función del número de hilos de los que disponemos, podemos observar que la escalabilidad aun es mala, pero mejora respecto al anterior, y además aumenta de forma uniforme al aumentar el número de procesadores..

4. How can you control in your code the trade-off between computation and synchronization? Is there an optimum value for the ratio between computation and synchronization? For the execution with 8 threads, explore possible ratios and plot how the execution time varies.

Para controlar el tiempo de computación y sincronización debemos controlar en nombre de generación y repartición de tareas entre los distintos procesadores dependiendo de la dependencia de datos que existe en la función a paralelizar, y así encontrar el equilibrio entre las dos.

Ejecución de la función de Gauss con 8 threads

n Col p or thread*	**Tiempo**
1	6.557
2	4.926
4	2.825
6	2.019
8	1.632
10	1.467
12	1.324
15	1.359
20	1.303
25	1.404
30	1.680
35	1.924
40	2.259