PACO Laboratory Assignment

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

#### Mario Acosta, Eduard Ayguad´e, Rosa M. Badia (Q2), Jesus Labarta,

#### Josep Ramon Herrero (Q1), Daniel Jim´enez-Gonz´alez, Pedro Mart´ınez-Ferrer, Adrian Munera, Jordi Tubella and Gladys Utrera

#### Fall 2023-24



# Index

[**Index 1**](#_gjdgxs)

1. **Before starting this laboratory assignment ... 3**
   1. [Some data decomposition strategies 3](#_30j0zll)
2. **Sequential heat diffusion program and analysis with Tareador 5**
3. **Parallelisation of the heat equation solvers 7**
   1. [Jacobi solver 7](#_1fob9te)
      1. [First Implementation 7](#_3znysh7)
      2. [Overall Analysis 8](#_2et92p0)
      3. [Detailed Analysis 8](#_tyjcwt)
      4. [Optimization 8](#_3dy6vkm)
   2. [Gauss–Seidel solver 8](#_1t3h5sf)
      1. [First Implementation 9](#_4d34og8)
      2. [Overall Analysis 9](#_2s8eyo1)
      3. [Detailed Analysis and Optimization 9](#_17dp8vu)
      4. [Finding the appropriate value for the number of blocks 9](#_3rdcrjn)
4. **Annex 1: Creating your own synchronisation objects 10**
5. **Laboratory 5 notebook 13**
   1. [Sequential heat diffusion program and analysis with Tareador 13](#_26in1rg)
   2. [OpenMP parallelization and execution analysis: *Jacobi* 13](#_lnxbz9)
   3. OpenMP parallelization and execution analysis: *Gauss-Seidel* 14

##### Note:

All files necessary to do this laboratory assignment are available in a compressed tar file available from the following location: /scratch/nas/1/par0/sessions/lab5.tar.gz. Copy it to your home directory in boada.ac.upc.edu and uncompress it with this command line: "tar -zxvf lab5.tar.gz".

**1**

**Before starting this laboratory assignment ...**

Before going to the labroom to start this laboratory assignment, we strongly recommend that you take a look at this section and try to solve the simple questions we propose to you. This will help to better face your last programming assignment in OpenMP: data decomposition for solving the heat diffusion equation.

## Some data decomposition strategies

The tasking model in OpenMP making use of explicit tasks that we have used in the two previous laboratory assignments is very versatile, allowing to express a wide range of dynamic task decompositions, both iterative (Lab 3) and recursive (Lab 4); however the programmer has no control on data locality. In this laboratory assignment you will explore the last decomposition strategy we study in this course: *data decomposition* making use of **implicit tasks**. With this strategy the computation that each implicit task has to perform is determined by the data it has to access, either read or write. This may have clear benefits in terms of data locality exploitation because each thread will always execute implicit tasks that access to the same data, whenever possible. *Data decompositions* can be *Geometric* or *Recursive*. In this lab we will focus our attention on the Geometric ones that are applied to n-dimensional matrices (including vectors).

Assume the code shown in Figure 1.1 for which we want to write a parallel code ensuring that each thread will compute the elements of matrix C that are stored in its own memory.

void vectoradd(int \*A, int \*B, int \*C, int N) { int i, j;

for (i=0; i< N; i++)

for (j=0; j< N; j++)

C[i\*N+j] = A[i\*N+j] + B[i\*N+j];

}

Figure 1.1: Simple example performing matrix sum.

Assuming that *P* is the number of processors and that *threadk* is executed on processor *Pk*, the left part of Figure 1.2 shows a possible distribution for the matrices by rows, so that the memory associated to each processor has a block of consecutive *N P* rows of matrices A, B and C. On the right we show a parallel implementation using implicit tasks for the loop that follows the so called *Owner-computes Rule*: each processor is responsible for the computations on the elements that are allocated in its main memory. Observe that, based on the value returned by intrinsic omp get thread num(), each thread can determine the subset of iterations of the i loop to execute (range between i start and i end); since

columns are not distributed, each thread executes all iterations of the j loop. For simplicity, the code in Figure 1.2 assumes that the number of elements *N* is a multiple of the number of threads executing the parallel region; think how it would change if this condition does not hold (think about it, trying to maximise load balancing).

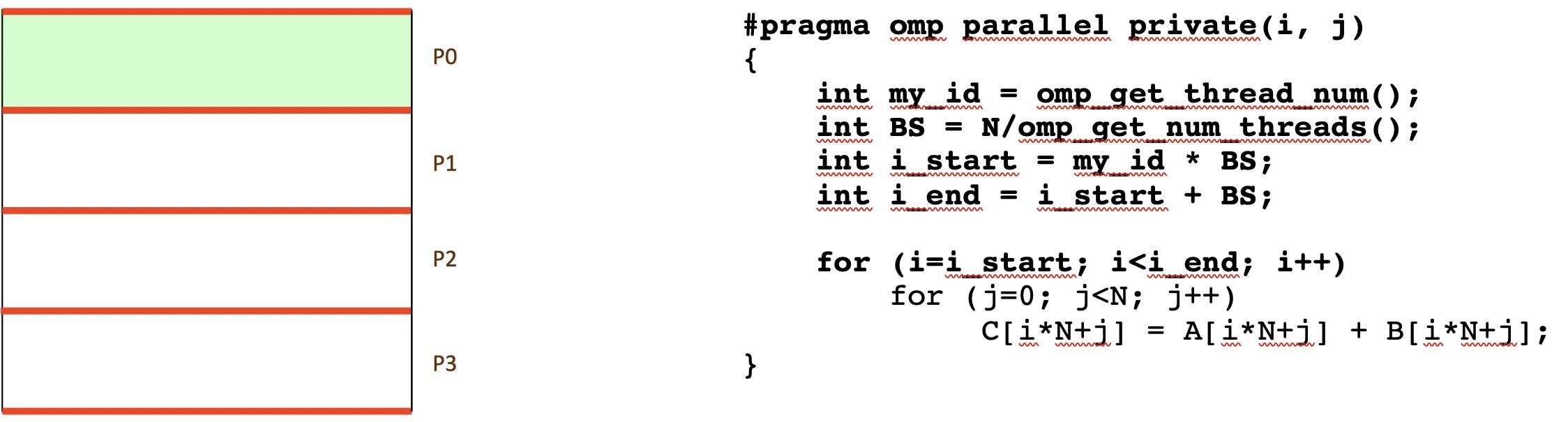


Figure 1.2: Left: geometric block data decomposition by rows. Right: parallelisation using implicit tasks.

Similarly, Figures 1.3 and 1.4 show two different data decomposition strategies and the code associ- ated, always using implicit tasks, that follows the owner computes rule. In Figure 1.3 each processor also has *N P* rows but now distributed in a cyclic way, starting with row 0 assigned to thread 0. Observe that the loop i now traverses the iteration space starting from the identifier of the thread executing the implicit task, jumping as many iterations as threads in the parallel region until reaching *N* .

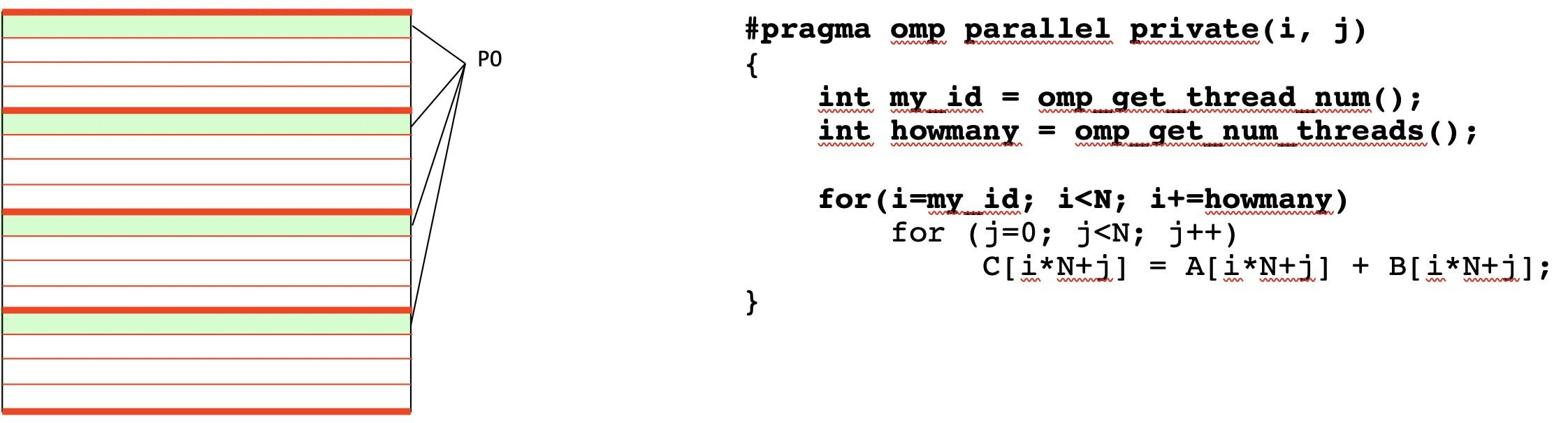


Figure 1.3: Left: geometric cyclic data decomposition by rows. Right: parallelisation using implicit tasks.

In Figure 1.4 each processor has blocks of *BS* = 2 consecutive columns assigned in a cyclic way, starting with the first block assigned to thread 0. Observe that now the loop that is decomposed is the j loop since it is the one used to access the columns. Do you understand how the loop is transformed in order to follow the owner-computes rule?

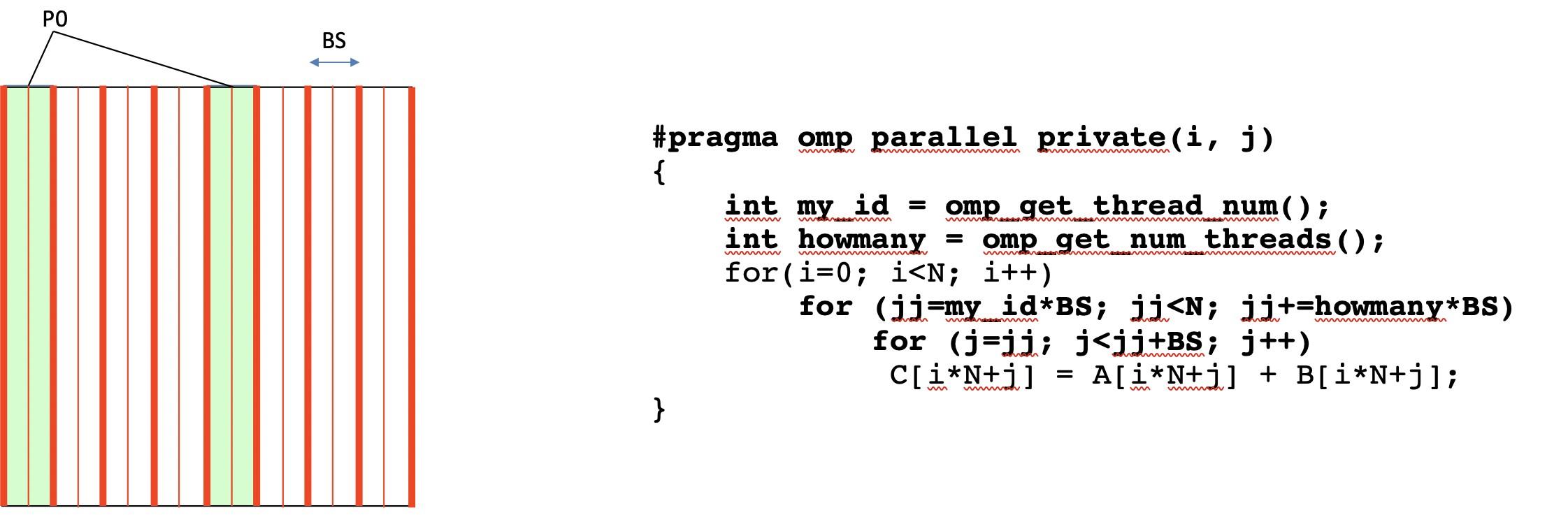


Figure 1.4: Left: geometric block-cyclic data decomposition by columns. Right: parallelisation using implicit tasks, assuming *N* %(*BS × howmany*) == 0.

Considering that two–dimensional matrices are stored in memory by rows, which should be the value (or values) for *BS* to avoid false sharing when writing elements C[i\*N+j]?

**2**

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

In this laboratory assignment you will work on the parallelisation of a sequential code that simulates the diffusion of heat in a solid body using two different solvers for the heat equation (*Jacobi* and *Gauss- Seidel* ). Each solver has different numerical properties which are not relevant for the purposes of this laboratory assignment; we use them because they show different parallel behaviours. In any case, you should be familiar with the two solvers since we have been using them quite extensively in the course.

Take a look at the the source code of heat-seq.c (where the solver is invoked) and solver-seq.c (where the solvers are coded). You will soon realise that both solvers use the same function solve. The difference is that *Jacobi* uses a temporary matrix to store the new computed matrix (param.uhelp) while *Gauss-Seidel* directly updates the same matrix (param.u). Notice that function solve is iteratively invoked inside a while loop that iterates while two different conditions are met: 1) the maximum number of iterations param.maxiter is not reached; and 2) the value returned by the solver is larger than param.residual. And also that at each iteration of the while loop *Jacobi* needs to copy the newly computed matrix into the original one in order to repeat the process, by simply invoking funtion copy mat also defined inside solver-seq.c.

The picture in Figure 2.1 shows the resulting heat diffusion when two heat sources are placed in the borders of the 2D solid (one in the upper left corner and the other in the middle of the lower border). The program is executed with a configuration file (test.dat) that specifies the number of heat sources, their position, size and temperature. The program also accepts several execution arguments: -n: maximum number of simulation steps or iterations; -s: the size of the body (resolution); -r: the residual value that stops the algorithm; -a: the solver to be used; and -o the output file (an image similar to the one shown in Figure 2.1 in portable pixmap file format, showing a gradient from red (hot) to dark blue (cold)). The execution of the program reports the execution time and performance measurements.

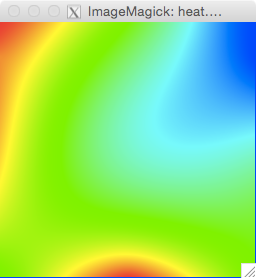
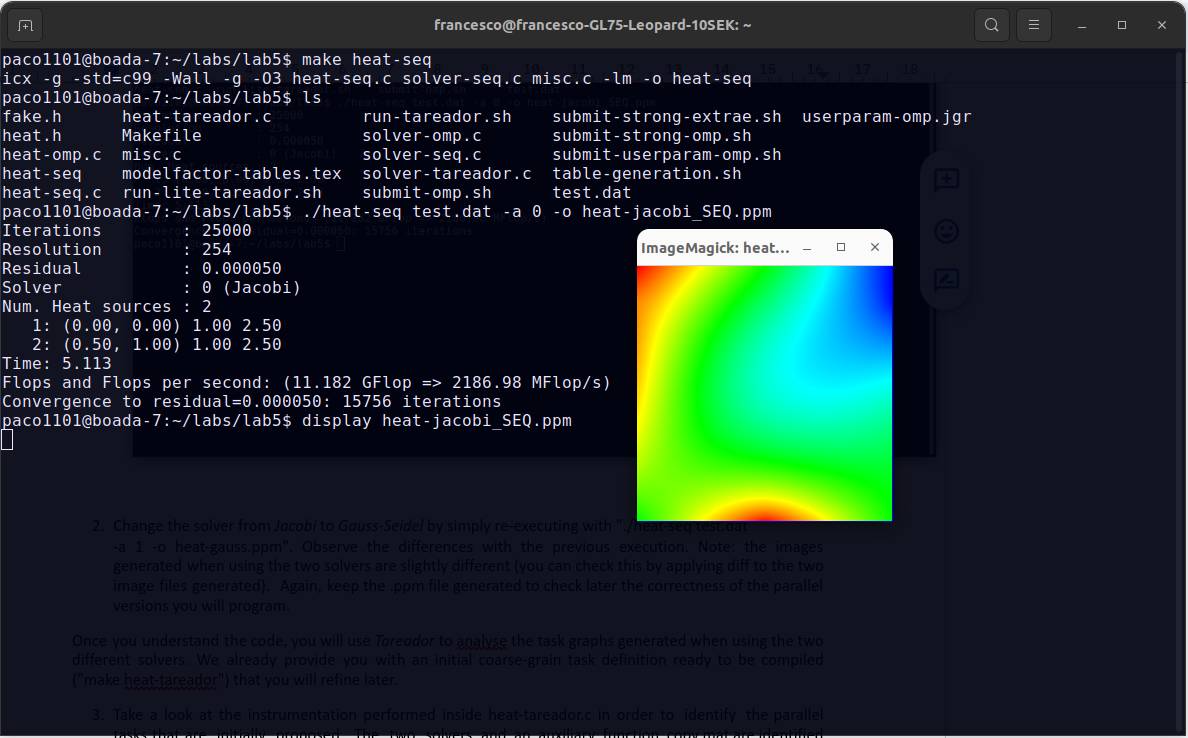


Figure 2.1: Image representing the temperature in each point of the 2D solid body

* + 1. Compile the sequential version of the program using "make heat-seq" and execute the binary generated using the *Jacobi* solver: "./heat-seq test.dat -a 0 -o heat-jacobi.ppm". The execution reports the execution time (in seconds), the number of floating point operations (Flop) performed, the average number of floating point operations performed per second (Flop/s), the residual and the number of simulation steps performed to reach that residual. Visualise the image file generated with an image viewer (e.g. "display heat-jacobi.ppm"); keep this file in your directory to check later the correctness of the parallel versions you will program.

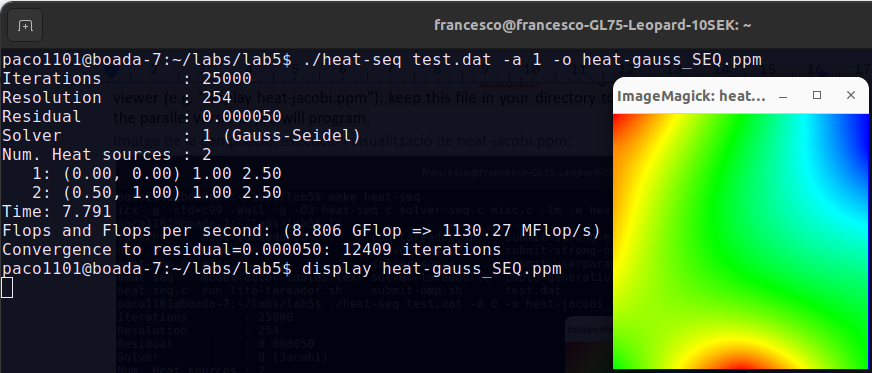
Imatge de la compilació, execució i visualització de heat-jacobi.ppm:



* + 1. Change the solver from *Jacobi* to *Gauss-Seidel* by simply re-executing with "./heat-seq test.dat

-a 1 -o heat-gauss.ppm". 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, keep the .ppm file generated to check later the correctness of the parallel versions you will program.

Imatge de l’execució i visualització de heat-gauss.ppm:



Podem observar que les 2 imatges són lleugerament diferents, sobretot si mirem el blau clar de la imatge obtinguda amb Jacobi, on s’exten una mica més cap al centre que amb el de Gauss.

Once you understand the code, you will use *Tareador* to analyse the task graphs generated when using the two different solvers. We already provide you with an initial coarse-grain task definition ready to be compiled ("make heat-tareador") that you will refine later.

* + 1. Take a look at the instrumentation performed inside heat-tareador.c in order to identify the parallel tasks that are initially proposed. The two solvers and an auxiliary function copy mat are identified as tasks in *Tareador*. Compile with the appropriate make target and execute with ./run-tareador.sh; the script has to receive the name of the executable and the solver to be used (0 for *Jacobi* or 1 for *Gauss-Seidel* ). The script internally specifies a very small test case which performs a few iterations on a very small body.

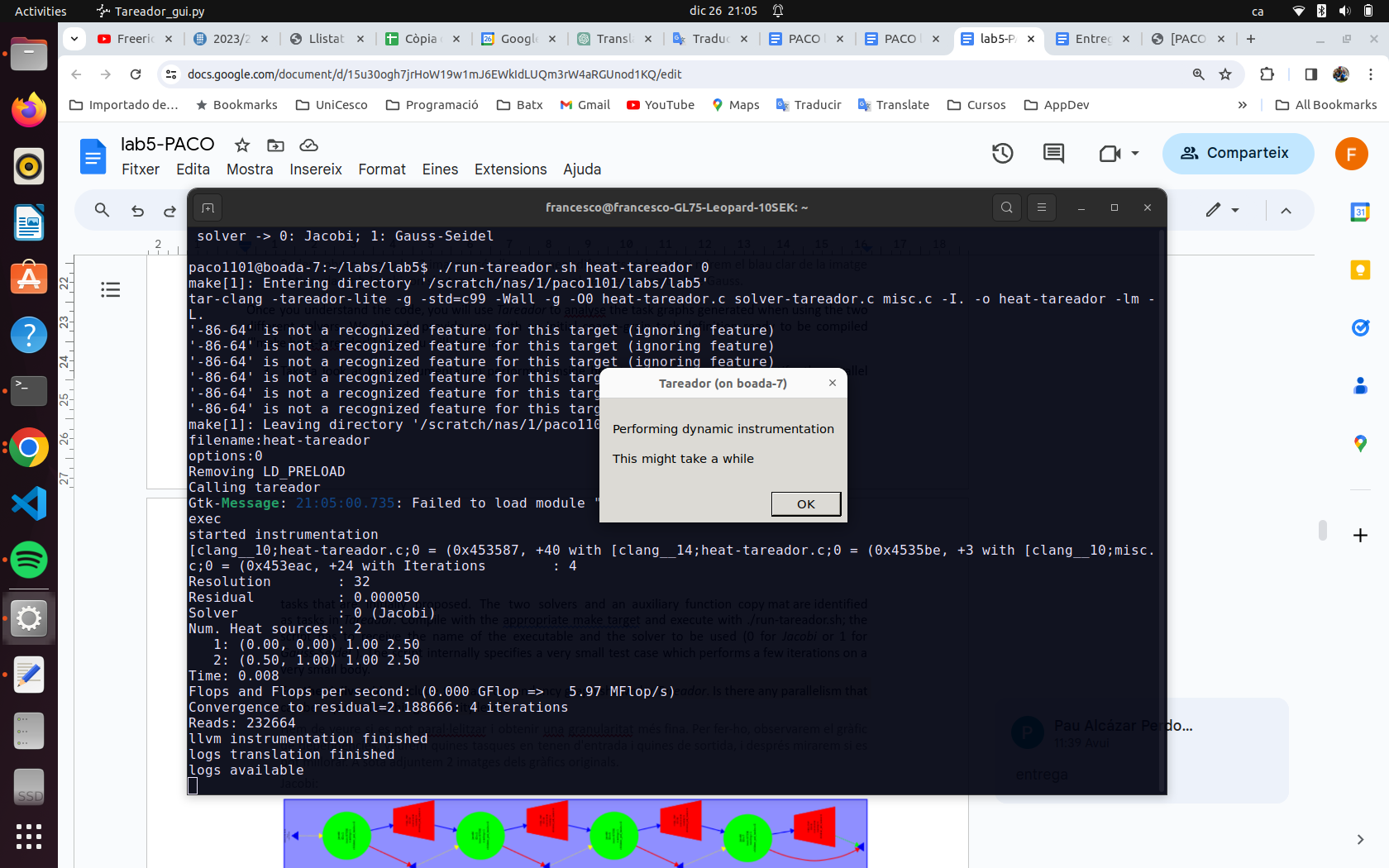
**For the deliverable:** Include the task dependency graph shown by *Tareador*. Is there any parallelism that can be exploited at this granularity level?

Hem de veure si es pot paral·lelitzar i obtenir una granularitat més fina. Per fer-ho, observarem el gràfic de dependències, veurem quines tasques en tenen d'entrada i quines de sortida, i després mirarem si es pot millorar. A sota adjuntem 2 imatges dels gràfics originals.

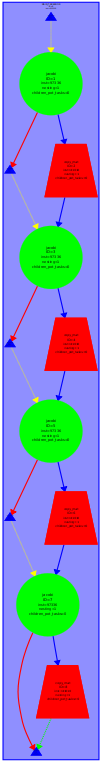
Per obtenir-les hem hagut de compilar heat-tareador i després escriure la comanda :

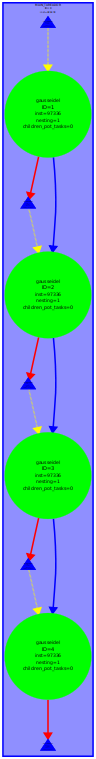
./run-tareador.sh heat-tareador 0

Escrivint 0 hem seleccionat Jacobi. Després hem tornat a executar el tareador però cambiant el 0 per l’1, i d’aquesta manera hem obtigut el graf de Gauss.



Jacobi:



Gauss:  


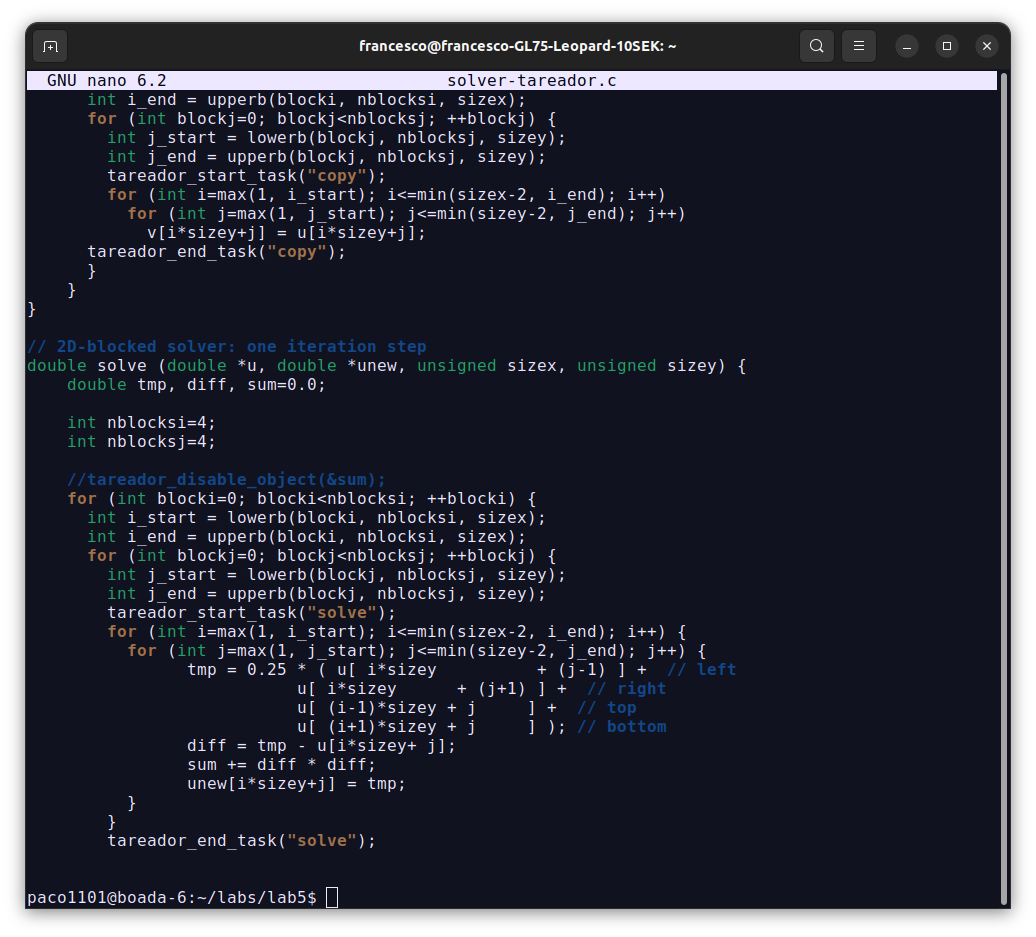
A les imatges podem veure que si canviem la granularitat a una més fina hi haurà certes parts que es paral·lelitzaran. Si obrim el tareador veurem que la dependència és de sum.

* + 1. We assume that the answer to the previous question was not affirmative. Let’s explore a finer granularity for both solvers. Open the solver-tareador.c file and take a closer look at the implementation of function solve. Notice that the function divides the computation of the 2- dimensional matrix unew using u in blocks, each block computing a subset of rows and columns. the lower and upper bounds in each dimension are computed (i start, i end, j start and j end) based on the size of the matrices that are used. **Important:** This is the granularity level we want you to explore for the tasks: **one task per block**. Make sure you understand the macros that are defined in the same file and how they are used to implement the blocking transformation.
    2. Change the original *Tareador* instrumentation to reflect the new proposed task granularity. Compile again, execute and analyze the task graphs that are generated when using both *Jacobi* and *Gauss- Seidel*.

**For the deliverable:** Include the excerpt of the code that you have modified in order to specify

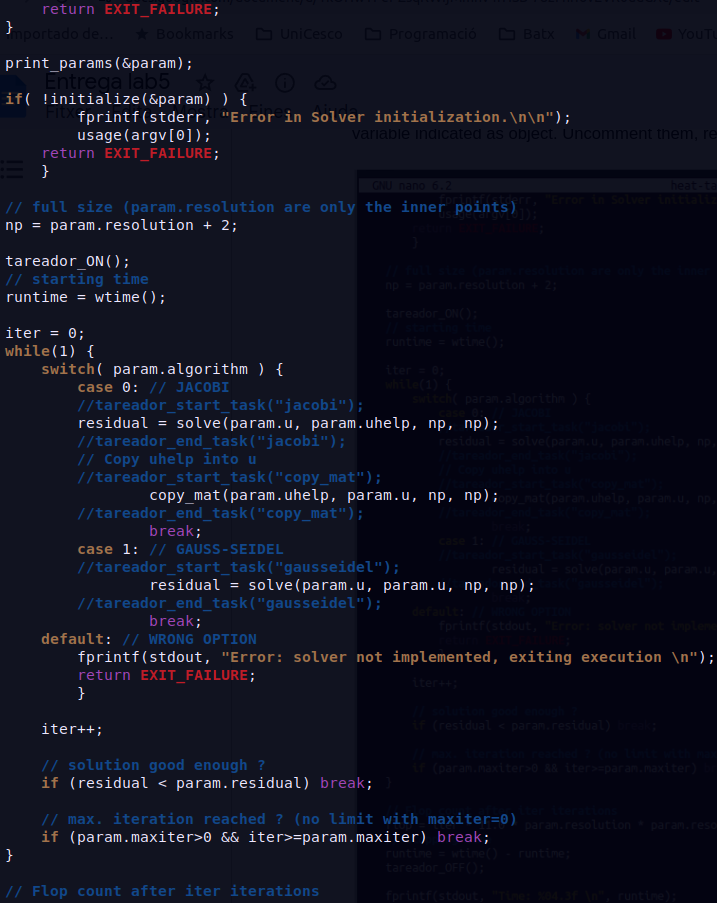
##### one task per block.

Aquest és el codi que ens permet assignar una tasca per bloc.



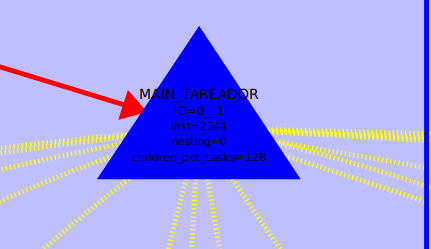
* + - 1. Which variable is causing the serialisation of all the tasks? Use the *Dataview* option in

*Tareador* to identify it.

Hem canviat el codi de heat-tareador.c comentant els diferents tareador\_start/end\_tasks():  


A les imatges podem veure que si canviem la granularitat a una més fina hi haurà certes parts que es paral·lelitzaran. Si obrim el Dataview de tareador veurem que la dependència és de sum.

Per fer el Dataview hem seleccionat tant per Jacobi com Gauss la tasca blava



Visualització del Dataview en l'eina de perfilat Tareador, mostrant la dependència crítica causada per la variable sum en l'execució del solucionador Jacobi. Els arcs vermells il·lustren com la serialització afecta la paral·lelització, amb una tasca esperant que la tasca anterior acabi degut a l'actualització de sum.

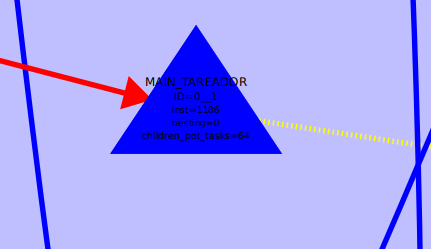
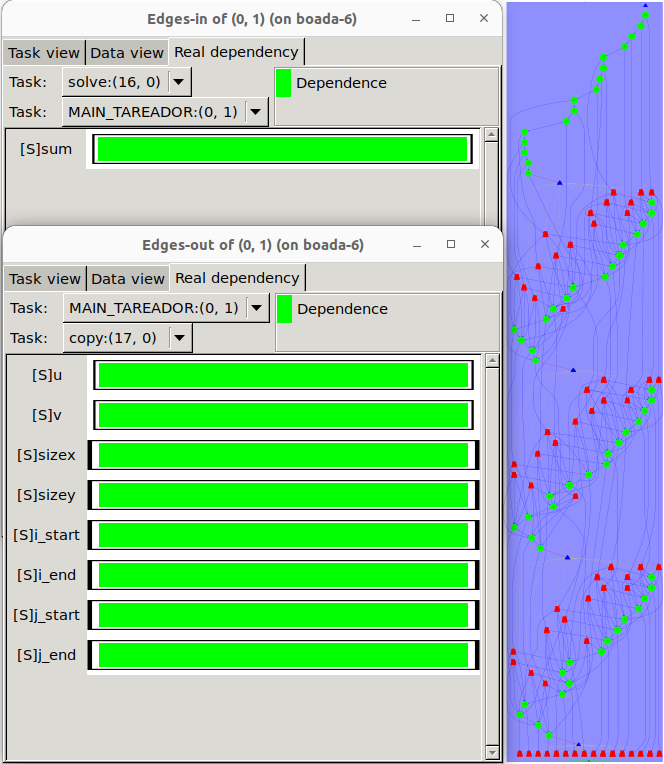
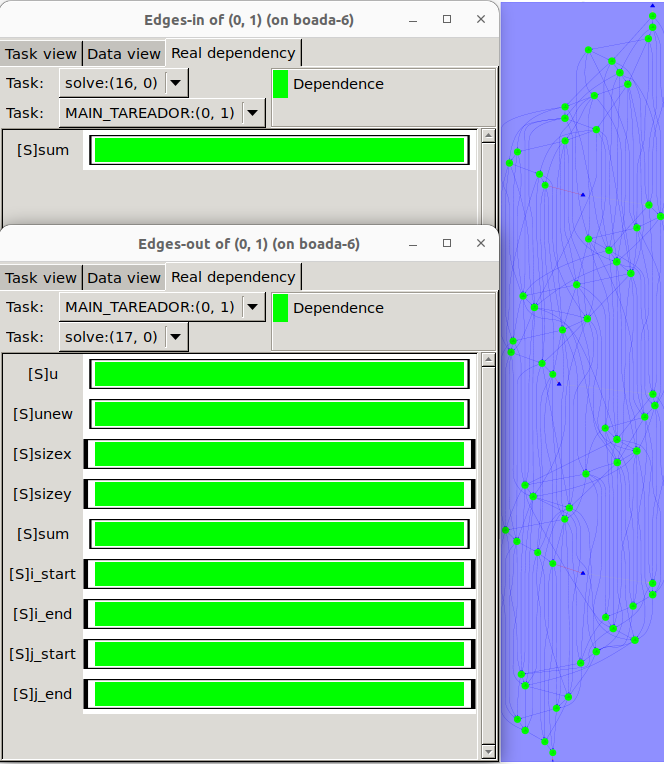


Diagrama de Dataview del solucionador Gauss-Seidel, on es pot observar una disminució de la dependència crítica en comparació amb el solucionador Jacobi. Les fletxes blaves indiquen la selecció de la tasca blava, permetent identificar les interdependències i potencials optimitzacions en la paral·lelització de les tasques.

Jacobi: Gauss:

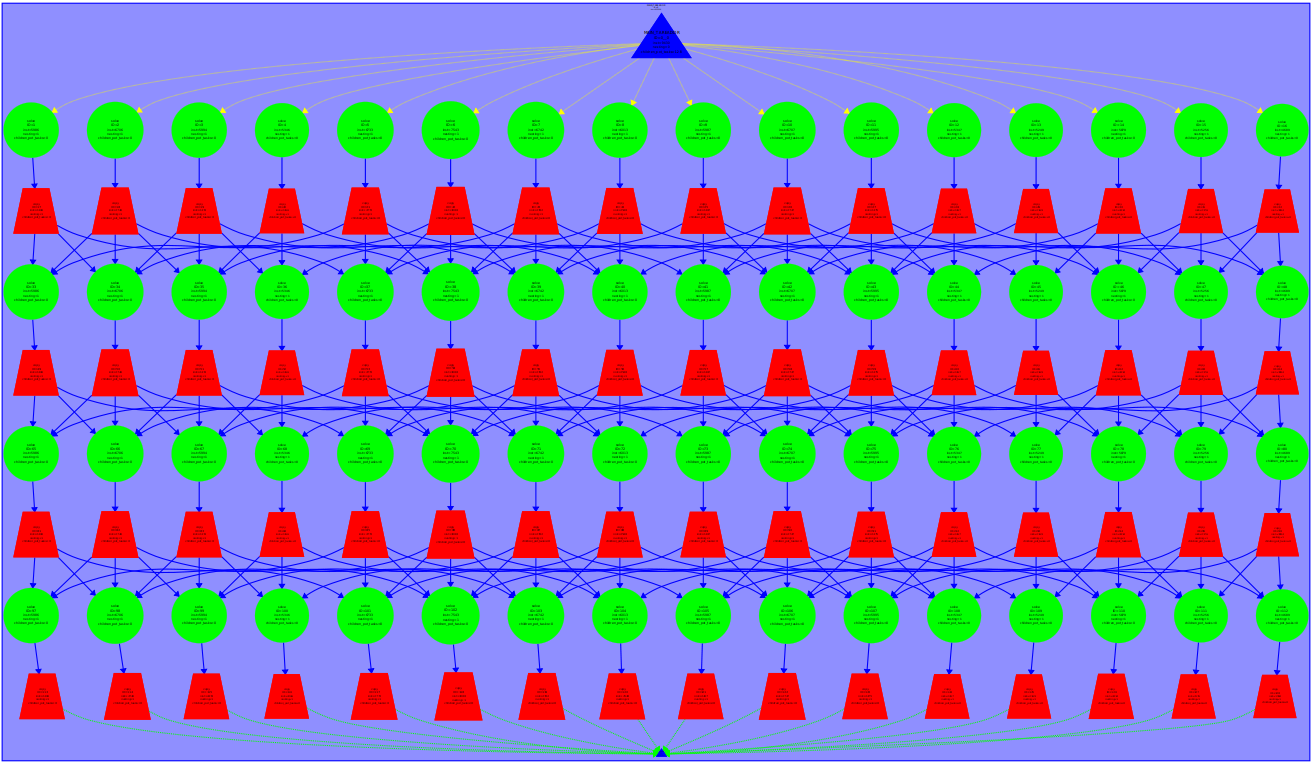
 

Comparativa dels perfils d'execució dels solucionadors Jacobi i Gauss-Seidel visualitzats amb l'eina Tareador. A l'esquerra, el solucionador Jacobi mostra una seqüència de dependències (línies vermelles) que suggereixen una alta serialització entre les tasques, limitant el paral·lelisme. A la dreta, el solucionador Gauss-Seidel exhibeix un patró d'execució amb paral·lelisme millorat, com es pot veure per la manca de dependències serialitzades i la presència predominant de línies verdes que indiquen dependències paral·leles entre les tasques.

* + - 1. 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.

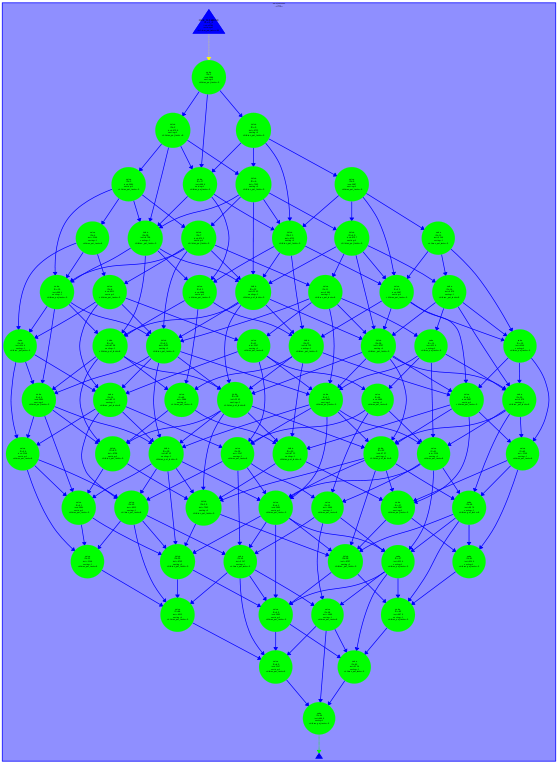
Per fer aquest segon apartat, hem tornat a modificar el codi de solver-tareador.c descomentant les linies tareador\_disable\_object(&sum); i tareador\_enable\_object(&sum);

Jacobi:



Representació de les dependències de tasques del solucionador Jacobi després de desactivar l'objecte $sum amb les funcions de Tareador, mostrant una alta densitat de dependències (triangles vermells) que apunten a una forta serialització en l'execució de les tasques, reflectint l'impacte de la variable $sum en el paral·lelisme de l'algorisme.

Gauss:



Gràfic de dependències per al solucionador Gauss-Seidel després d'habilitar l'objecte $sum per a la simulació amb Tareador, on es pot observar un patró de dependències menys concentrat i més distribuït (cercles verds), suggerint un millor aprofitament del paral·lelisme comparat amb el solucionador Jacobi.

* + - 1. 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.

**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* ?

Jacobi:

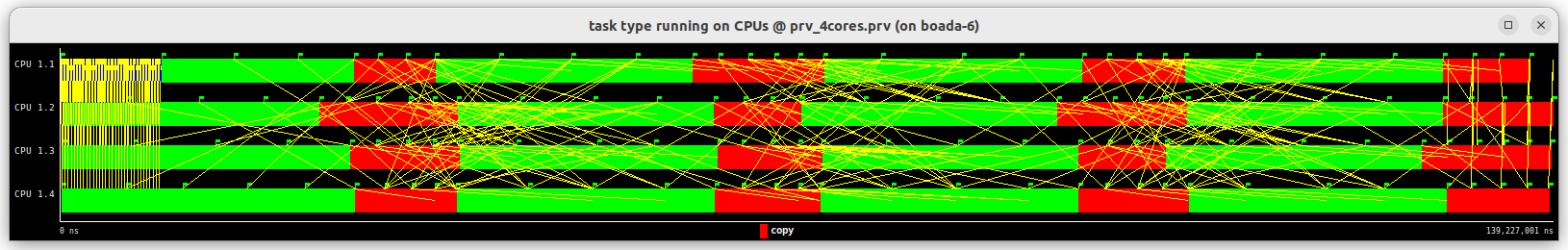
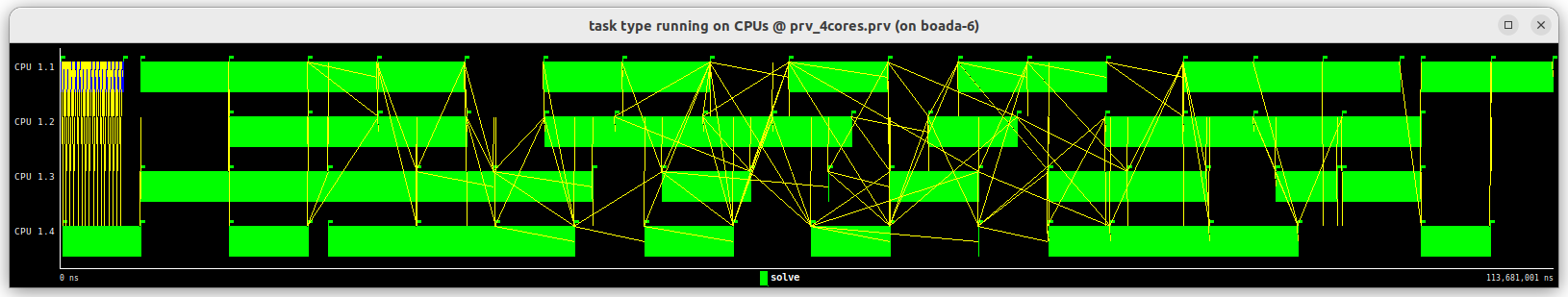


Diagrama de dependència de tasques per al solucionador Jacobi utilitzant quatre fils. Les barres horitzontals representen les tasques executades i les línies vermelles indiquen dependències entre tasques que suggereixen una serialització significativa. Aquest patró de dependències mostra com cada fil ha d'esperar la finalització d'una tasca abans de començar la següent, indicant un possible coll d'ampolla en l'execució paral·lela

Gauss:



Visualització de l'execució paral·lela per al solucionador Gauss-Seidel amb quatre fils. A diferència del solucionador Jacobi, aquest diagrama mostra un major grau de paral·lelisme amb menys dependències serialitzades, com es pot observar pels blocs verds que representen tasques executant-se simultàniament sense esperar per altres tasques.

**3**

**Parallelisation of the heat equation solvers**

## Jacobi solver

In this section you will first parallelise the sequential code for the heat equation code considering the use of the *Jacobi* solver, using the **implicit tasks** generated in #pragma omp parallel1, following a *geometric block data decomposition by rows*, as shown in Figure 3.1 for 4 threads running on 4 processors.

### First Implementation

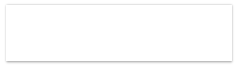
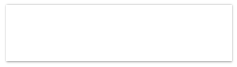
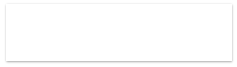
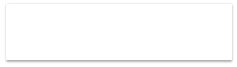
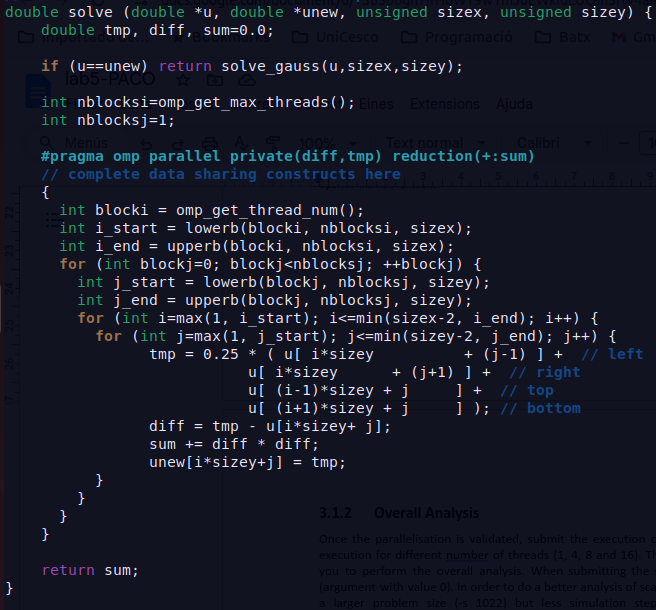
u and utmp

Figure 3.1: Geometric (data) decomposition for matrix u (and utmp) by rows, for 4 threads.

With this strategy the computation that the implicit task executed each thread has to perform is deter- mined by the data it has to access, either read or write. This may have clear benefits in terms of data locality exploitation because each thread will always execute implicit tasks that access to the same data, whenever possible. How could we parallelise the sequential code for function solve to ensure that each processor computes its assigned elements?

To start with, analyse the initial parallelisation proposed in solver-omp.c for function solve. Try to understand how the proposed parallelisation follows the data decomposition strategy mentioned above. **Complete the parallel code so that it honors the dependences that you have discovered when applying the *Jacobi* solver**.

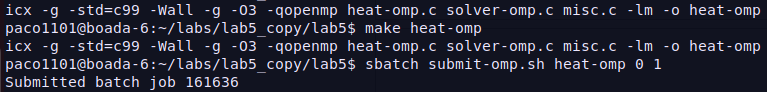
Modifiquem el codi de solver-omp.c perquè generi paral·lelització quedant de la següent manera:



Compile using make heat-omp and submit its execution to the queue using the submit-omp.sh script, specifying the binary file, the use of the Jacobi solver and 1 (sbatch submit-omp.sh heat-omp 0 1) and 8 threads (sbatch submit-omp.sh heat-omp 0 8). Validate the parallelisation by visually inspecting the image generated and making a diff with the file generated with the original sequential version.

1**Important:** You can not make use of other pragmas to create explicit tasks or distribute the work among the implicit tasks. You can use OpenMP intrinsic functions.

Imatge de la compilació i execució a la cua amb 8 threads:





Si fem un diff del binari generat al principi de tot amb l’execució seqüencial, amb el nou binari, veiem que no hi ha cap diferència. Això vol dir que el nou programa, quan l’executem amb 8 threads, obtenim el mateix resultat i per tant el que nosaltres hem modificat és correcte.



### Overall Analysis

Once the parallelisation is validated, submit the execution of the submit-strong-extrae.sh script to trace the execution for different number of threads (1, 4, 8 and 16). This script will also execute mfLite.py that will help you to perform the overall analysis. When submitting the script you should specify the solver to be used (argument with value 0). In order to do a better analysis of scalability, this script is already modified to run with a larger problem size (-s 1022) but less simulation steps (-n 1000). **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.

### Detailed Analysis

We recommend you to open with paraver the trace that has been generated for 16 threads and observe what is causing the low value for that metric. Use paraver Hints Instantaneous Parallelism and Implicit Tasks in Parallel Constructions, synchronize them and zoom in one of the trace to see the detail of the execution.

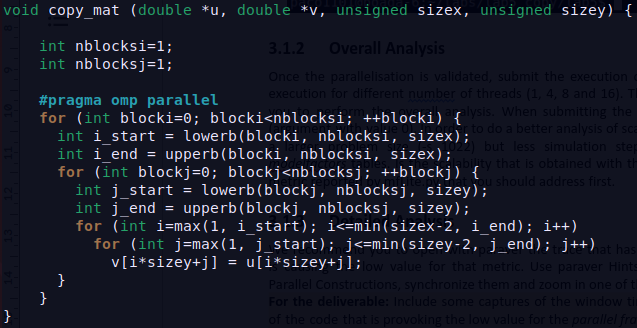
**For the deliverable:** Include some captures of the window timelines to show the problem. What is the region of the code that is provoking the low value for the *parallel fraction* in your parallelisation? Hint: Remember the *Tareador* analysis you did.

### Optimization

Parallelise other parts of the code in order to improve the efficiency of your parallel code. Compile the new version and submit its execution to the queue using the submit-omp.sh script, specifying the binary file, the use of the Jacobi solver and 16 threads. Make sure the new parallel version still generates correct results.

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

Per optimitzar l’eficiència, afegim #pragma omp parallel al principi de la funció copy\_mat de solver-omp.c, i així aconseguim paral·lelitzar la següent zona:



##### Overall Analysis of the Optimized Code

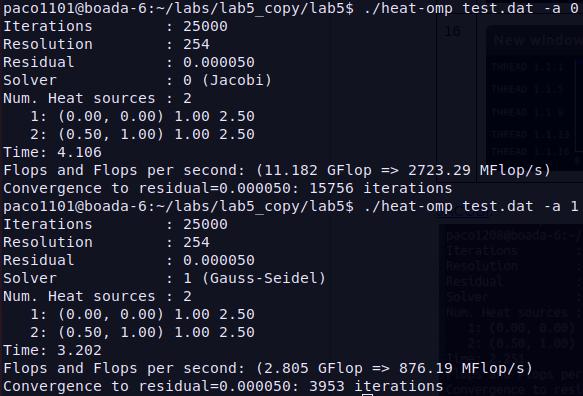
Once validated, submit again the submit-strong-extrae.sh script and analyse the scalability of the new parallelisation for different number of threads (1, 4, 8 and 16); do not forget to specify with an argument the solver to be used (0). For completeness, use the submit-strong-omp.sh script to queue the execution of heat-omp and obtain the scalability plot to be included in the deliverable; do not forget to specify with an argument the solver to be used (0).

**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?

##### Detailed Analysis of the Optimized Code

We recommend you to open with paraver the trace that has been generated for 16 threads and observe the behaviour for your parallel code. Use paraver Hints Instantaneous Parallelism and Implicit Tasks in Parallel Constructions, synchronize them and zoom in one of the trace to see the detail of the execution. **For the deliverable:** Include some captures of the window timelines.

Imatge de l’execució per a Jacobi:



## Gauss–Seidel solver

Once the parallelization of the solver for *Jacobi* is appropriate, you should continue the parallelization process considering the dependences that appear when the *Gauss-Seidel* is used (that you discovered using *Tareador* in the previous chapter).

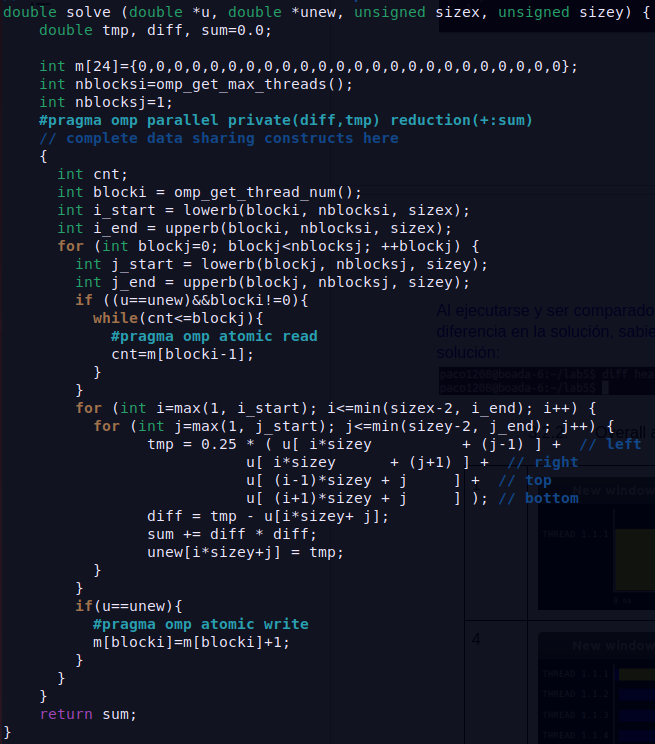
### First Implementation

The parallelisation should follow the same *geometric block by rows data decomposition* that is shown in Figure 3.1, and again ONLY making use of the implicit tasks in parallel regions.

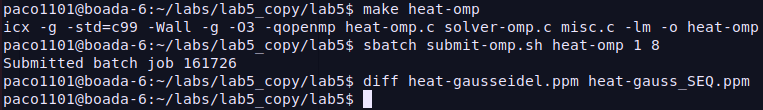
##### Note: before continuing, we suggest you take a look at the explanation in *Annex 1* to make sure you understand how to express ordering constraints among (implicit) tasks using shared variables and the *memory consistency* problem that may occur.

* + - 1. Parallelise the *Gauss-Seidel* solver (*solve gauss* function), introducing the necessary synchronisa- tion among implicit tasks and data sharing constraints. The number of blocks in the i dimension should be determined by the geometric data decomposition (i.e. the number of threads or implicit tasks in the parallel region); for your first implementation we suggest to use 20 blocks in the j dimension, being 20 the maximum number of threads of the strong scaling analysis. Later you will explore different numbers of blocks in the j dimension.
      2. Compile using make heat-omp and submit the execution of the binary using the submit-omp.sh script to validate the parallelisation (by making a diff with the file generated with the original sequential version). Do not forget to specify the *Gauss-Seidel* solver when submitting the script
         1. as well as the number of threads to use.

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

Imatge de com queda el codi després d’aplicar els canvas especificats:  
La primera funció, la de copy\_mat, queda igual que a la última modificació, amb la paral·lelització utilitzant la comanda #pragma omp parallel. I la funció solve quedarà de la següent manera:  


Per comprovar que efectivament el codi és correcte hem de fer el mateix que amb Jacobi, i fer un diff amb el resultat obtingut al principi de tot (el seqüencial):



### Overall Analysis

Once validated, submit the execution of the submit-strong-omp.sh script to queue the execution of heat-omp and obtain the scalability plot; do not forget to specify with an argument the solver to be used (1).

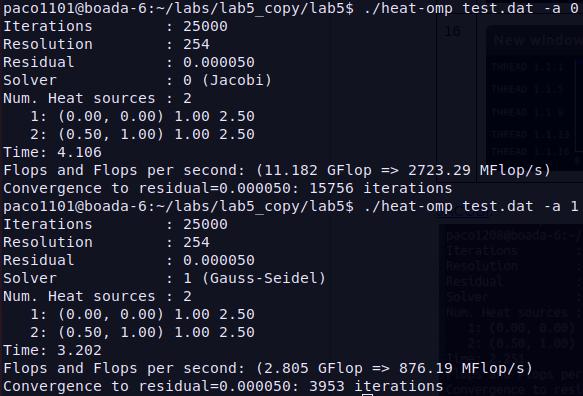
**For the deliverable:** Include the plot obtained with submit-strong-omp.sh. Do you observe a linear speedup? Reason your answer. Note: Remember the analysis done with *tareador* with a simulation with 4 threads.

### Detailed Analysis and Optimization

In order to exploit more parallelism in the execution of the solver, you should change the number of blocks in the j dimension. Why? Changing the number of blocks in the j dimension changes the ratio between computation and synchronisation, and may increase the parallelism, why? In order to do this change in your code, main is already prepared to receive an argument in the command-line execution (-u value) that is stored in global variable userparam. Modify your code accordingly to make use of this value and change the number of blocks in the j dimension without recompiling the code for each new value to test. Use the value in userparam as a factor of the number of blocks nblocksi to set nblocksj (i.e. nblocksj=nblocksi\*userparam). Compile and execute with some different values and check that it returns the correct results and has the expected effect.

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

Imatge de l’execució per a Gauss:



### Finding the appropriate value for the number of blocks

Use the provided submit-userparam-omp.sh script to explore a range of values of the number of blocks. The only argument for this script is the number of threads to be used for the exploration. Submit this script for 16 threads. Select a number of blocks, edit submit-strong-omp.sh to change the value of userparam variable and submit submit-strong-omp.sh script to queue the execution of heat-omp and obtain the scalability plot; do not forget to specify with an argument the solver to be used (1).

**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).

**4**

**Annex 1: Creating your own synchronisation objects**

The implicit tasks used to express parallelisation strategies based on data decomposition can not syn- chronise themselves using task dependences (i.e. depend clauses can not be applied to implicit tasks). For this reason in this laboratory assignment you will also implement your own synchronisation objects to implement some sort of task ordering constraints. These synchronisation objects will be implemented using shared variables for which one has to ensure that all accesses (reads and writes) to them always access to memory. This is what is called the *memory consistency problem*: usually the compiler tries to optimise memory accesses by placing variables in registers, and then only read/write from/to memory at certain points in the parallel program, usually at synchronisation points.

For example consider the simple producer-consumer code shown in Figure 4.1, using implicit tasks. Notice that all implicit tasks can do their *computation A* part in parallel; however, the instance of the implicit task executed by myid cannot execute *computation B* until the previous thread has executed it (in other words, the execution of *computation B* has to be done in an ordered way). Only the instance of the implicit task executed by thread 0 can do the execution of *computation B* initially. This is controlled by vector next with as many positions as threads (i.e. instances of the implicit task): if position myid is 0, then the implicit task will wait in the while loop; once it is set to 1 by the implicit task myid-1, the implicit task will be allowed to continue.

int next[P];

...

next[0] = 1;

for (int i = 1; i < P; i++) next[i] = 0;

...

#pragma omp parallel num\_threads(P)

{

int myid = omp\_get\_thread\_num();

// computation A

while (next[myid] == 0); // wait to advance

// computation B

if (myid < P-1) next[myid+1]++;

}

Figure 4.1: Simple producer-consumer code.

Although the code seems to be correct, it has memory consistency problems. To ensure that each task always accesses to the last value of the shared variable next, the programmer has to introduce some sort of data sharing/synchronisation construct. The preferred one is atomic, which can have three different clauses: read, write and update. Clauses read and write are used to express that memory accesses are always served by main memory. The resulting code is shown in Figure 4.2. Inside the do

while construct we are ensuring that the element of vector next is read from memory by adding the atomic read; in order to ensure that the element of vector next is updated in memory we need to add atomic write.

int next[P];

...

next[0] = 1;

for (int i = 1; i < P; i++) next[i] = 0;

...

#pragma omp parallel num\_threads(P)

{

int myid = omp\_get\_thread\_num();

// computation A

do {

#pragma omp atomic read tmp = next[myid];

} while (tmp == 0); // wait to advance

// computation B if (myid < P-1) {

#pragma omp atomic write next[myid+1] = 1;

}

}

Figure 4.2: Simple producer-consumer code making use of atomic pragmas to guarantee memory consis- tency.

How would you change the code above if we introduce an iterative loop that repeats the execution of *computation A* and *computation B* several times, always ensuring the same execution order constraints? Figure 4.3 shows the code to be completed in order to ensure the appropriate execution ordering.

int next[P];

...

next[0] = ...;

for (int i = 1; i < P; i++) next[i] = ...;

...

#pragma omp parallel num\_threads(P)

{

int myid = omp\_get\_thread\_num();

for (int iters = 0; iters < num\_iters; iters++) {

// computation A

do { ... } while ( ... ); // wait to advance

// computation B

if (myid < P-1) next[myid+1] = ...;

}

}

Figure 4.3: Second version of simple producer-consumer code.

**Question:** Can the access to vector next cause false sharing? If you answered yes, how would you solve the problem?

Sí, l'accés al vector next pot causar false sharing. Això succeeix perquè diverses entrades del vector poden estar en la mateixa línia de cau i, per tant, l'actualització d'una entrada per part d'un fil pot provocar la invalidació de la línia de cau que un altre fil està llegint repetidament.

Per solucionar aquest problema, podem emprar el que es coneix com a "padding", que consisteix a expandir la mida de cada element del vector next perquè cada element ocupi una línia de cau completa. Això es pot fer emmagatzemant cada enter en una estructura que té suficient espai addicional per omplir una línia de cau. D'aquesta manera, quan un fil actualitza una posició del vector, no afectarà la línia de cau d'una altra posició que podria estar sent accedida per un altre fil.

Codi que es podria implementar:

#define CACHE\_LINE\_SIZE 64 // Suposem que la mida de la línia de cau és 64 bytes

#define PADDING (CACHE\_LINE\_SIZE - sizeof(int)) // Mida del padding necessari per a cada element

struct padded\_int {

int value;

char pad[PADDING]; // El padding per omplir la resta de la línia de cau

};

struct padded\_int next[P]; // El vector amb padding

i dins del codi es modificaria d’aquesta manera:

next[0].value = 1;

for (int i = 1; i < P; i++) {

next[i].value = 0;

}

#pragma omp parallel num\_threads(P)

{

int myid = omp\_get\_thread\_num();

for (int iters = 0; iters < num\_iters; iters++) {

// computation A

// Espera activa amb padding per evitar el false sharing

while (next[myid].value == 0);

// computation B

if (myid < P-1) {

next[myid+1].value = 1;

}

}

}

**How to:**

**Document contents**

##### The length of the document is expected to be maximum 8 pages (including figures and tables) with a font size of at least 11pt.

You have to add comments and observations following the questions that you have found along the document indicating **For the deliverable:** in each section (in this laboratory we have included those **For the deliverable** questions in the notebook with setences in gray under each **Comments/Observations**). DON’T include code in the document except for the fragment modified with respect to the original one.

Finally, as you know, this course contributes to the **transversal competence ”Tercera llengua”**. Deliver your material in English if you want this competence to be evaluated. Please refer to the ”Rubrics for the third language competence evaluation” document to know the *R*ubric that will be used.

**Submission**

Only PDF format for this document will be accepted. Deliverable assignment will be opened in *Atenea* and set the appropriate dates for the delivery. You also have to deliver the complete C source codes for Tareador instrumentation and all the OpenMP parallelisation strategies that you have done. Your lab professor should be able to re-execute the parallel codes based on the files you deliver. You will have to **deliver TWO files**, one with the document in PDF format and one compressed file (tgz, .gz or .zip) with the requested C source codes.

**5**

**Laboratory 5 notebook**

## Sequential heat diffusion program and analysis with Tareador

This part refers to section 2 of the practice document.

You must include:

The task dependency graph shown by *Tareador* for both solvers *Jacobi* and *Gauss-Seidel* for the default codes (naive version).

The task dependency graph shown by *Tareador* for both solvers for the proposed task granularity, adding the dependences filter and new tasks per block in other parts of the code to increase parallelism.

The excerpt of the last version of the *Tareador* code that you have modified in order to specify

**one task per block** and exploit other parts of the code in addition to solver function.

##### Comments/Observations

For the naive version: Is there any parallelism that can be exploited at the naive version granularity? For the proposed task granularity: 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 in the proposed task granularity than in the default version? Is the parallelism achieved the same for *Jacobi* and *Gauss-Seidel* solvers?

Per la Versió Naïve:

En la versió naïve dels solucionadors, el paral·lelisme que pot ser explotat pot ser limitat degut a la manera com les tasques i les dades estan organitzades. Aquesta versió pot no tenir una granularitat fina o una correcta distribució de les tasques, la qual cosa pot conduir a dependències i serialització no desitjades. No obstant això, si la càrrega de treball per a cada iteració és significativa i les dades estan suficientment desacoblades, podríem trobar oportunitats per explotar el paral·lelisme en algunes operacions independents, com ara l'actualització de diferents regions de l'estructura de dades si aquestes no interfereixen entre elles.

Per la Granularitat de Tasca Proposada:

La variable que estava causant la serialització de totes les tasques era sum. Aquesta variable s'actualitza de manera acumulativa durant l'execució dels solucionadors i, per tant, qualsevol accés concurrent a aquesta variable pot conduir a una condició de carrera, on els resultats poden ser incorrectes si diverses tasques intenten llegir o escriure en sum simultàniament.

Protecció de l'Accés a la Variable sum en la Implementació OpenMP:

Per protegir l'accés a sum en la implementació OpenMP, podem utilitzar la clàusula reduction en un bucle paral·lelitzat amb #pragma omp for. Això assignarà a cada fil una còpia privada de sum, que es pot actualitzar de manera independent durant l'execució del bucle. Al final del bucle, totes les còpies locals es combinaran en una única variable sum global. Aquest enfocament evita la serialització innecessària i permet que els fils treballin de manera més paral·lela.

Comparació del Paral·lelisme entre la Versió Proposada i la Naïve:

S'esperaria que la granularitat de tasca proposada oferís més paral·lelisme que la versió naïve perquè la nova estratègia redueix o elimina les dependències innecessàries i permet una major distribució del treball entre els fils. Això es pot confirmar mitjançant la revisió dels gràfics de dependència de tasques generats per eines com Tareador, així com amb proves de rendiment per mesurar el temps d'execució i l'ús de la CPU.

Consistència del Paral·lelisme entre els Solucionadors Jacobi i Gauss-Seidel:

Quant a si el paral·lelisme aconseguit és el mateix per als solucionadors Jacobi i Gauss-Seidel, això depèn de les característiques intrínseques de cada algorisme. El solucionador Jacobi és inherentment més paral·lel perquè cada actualització és independent de les altres (ja que només utilitza valors de l'iteració anterior), mentre que Gauss-Seidel té una naturalesa més serial ja que utilitza les actualitzacions a mesura que es calculen. Per tant, fins i tot amb una granularitat de tasca optimitzada, Jacobi podria ser capaç de aconseguir un major grau de paral·lelisme en comparació amb Gauss-Seidel. Tanmateix, això hauria de ser confirmat amb una anàlisi detallada de l'execució paral·lela i mesuraments de rendiment.

## OpenMP parallelization and execution analysis: *Jacobi*

This part refers to section 3.1 of the practice document. In addition, some indication details of the specific subsections each question refers to are specified below.

You must include:

An excerpt of the code to show the OpenMP annotations you have added to the code after the optimizations (Optimization section - Jacobi solver).

The *modelfactors* tables and the plot of scalability for the first implementation (Overall Analysis section - Jacobi solver) and your last optimized implementation (Overall Analysis of the Optimized Code - Jacobi solver).

Captures of the window timelines for both first (Detailed Analysis section - Jacobi solver) and last implementation (Detailed Analysis of the Optimized Code).

##### Comments/Observations

What was the region of the code that was provoking the low value for the *parallel fraction* in your first parallelisation? (Detailed Analysis section) Compare the *parallel fraction* of your first and last versions (Overall Analysis of the Optimized Code section). Is the execution time reduced from your first to the last version?. Have you increased the scalability? (Overall Analysis of the Optimized Code section) Compare the timelines of both executions under the point of view of instantaneous parallelism (Detailed Analysis of the Optimized Code section).

Regió del Codi que Provocava la Baixa Fracció Paral·lela en la Primera Paral·lelització: La regió del codi que va provocar una baixa fracció paral·lela en la primera paral·lelització va ser probablement aquella on les tasques tenien dependències fortes, com ara actualitzacions repetitives a la variable compartida sum. Aquesta variable acumulativa, essent central per a l'actualització dels resultats dels solucionadors, necessita ser accedida i modificada de manera coherent entre els diferents fils d'execució, cosa que pot haver provocat un coll d'ampolla en l'execució paral·lela.

Comparació de la Fracció Paral·lela entre la Primera i l'Última Versió: La fracció paral·lela, que indica la proporció del programa que pot executar-se en paral·lel, hauria d'haver millorat en l'última versió del codi després d'optimitzar les regions crítiques i reduir les dependències. Utilitzant eines com Tareador i Paraver, podem comparar la fracció paral·lela de la primera versió amb l'última. L'expectativa és que l'última versió tingui una fracció paral·lela més alta, indicant una millor paral·lelització del codi.

Reducció del Temps d'Execució de la Primera a l'Última Versió: Una optimització exitosa resultaria en un temps d'execució reduït des de la primera fins a l'última versió del programa. Això es pot verificar mitjançant la comparació dels temps d'execució recollits durant les proves de rendiment en ambdues versions. Una reducció en el temps d'execució és un indicador que les millores implementades han estat efectives.

Increment de l'Escala: L'escalabilitat d'un programa paral·lel es refereix a la seva capacitat per augmentar el rendiment amb l'addició de més recursos de processament. Si l'última versió del codi ha millorat l'escalabilitat, això significa que aprofita millor els fils addicionals per a una execució més eficient. Això es pot mesurar amb proves de scaling fort, on es compara el rendiment amb un nombre creixent de fils.

Comparació de les Línies de Temps de les Execucions: Comparant les línies de temps de les execucions de la primera i l'última versió des del punt de vista del paral·lelisme instantani, podem veure visualment com les optimitzacions han afectat l'execució paral·lela. L'optimització efectiva es reflectiria en un augment dels períodes de temps on més tasques s'executen en paral·lel i una disminució dels períodes on les tasques estan bloquejades esperant dependències.

* 1. **OpenMP parallelization and execution analysis: *Gauss- Seidel***

This part refers to section 3.2 of the practice document. In addition, some indication details of the specific subsections each question refers to are specified below.

You must include:

The plot of scalability when using 20 blocks in the j dimension, nblocksj=20 (Overall Analysis section).

An excerpt of the OpenMP code to show the modifications done: OpenMP annotations and syn- chronization mechanisms (once you make nblocksj=userparam \* nblocksi) (Detailed Analysis and Optimization section).

The plots obtained with submit-userparam-omp.sh using 16 threads (Finding the appropriate value for the number of blocks section) and and strong scalability for the best value of userparam (nblocksj=userparam \* nblocksi).

##### Comments/Observations

Do you observe a linear speedup for 20 blocks in the j dimension? (Overall Analysis section). Reason *why* changing the number of blocks in the j dimension changes the ratio between computation and synchronisation (Detailed Analysis and Optimization, Number of blocks tune section) and explain the plots that you have included, comparing the strong scalability for both cases nblocksj=20 and nblocksj=best userparam \* nblocksi (Finding the appropriate value for the number of blocks).

Per a respondre aquestes preguntes especificades en la secció "Comments/Observations" del document:

Acceleració Lineal per a 20 Blocs en la Dimensió j: L'acceleració lineal es refereix al cas ideal en què duplicar el nombre de fils de processament resulta en una reducció a la meitat del temps d'execució. No obstant això, l'acceleració real sovint és sublineal a causa de l'overhead de la gestió de fils i la sincronització. Si observem una acceleració lineal per a 20 blocs en la dimensió j, això indicaria que els blocs estan prou desacoblats per a que la computació dins de cada bloc es pugui fer de manera independent, maximitzant així l'ús eficient dels fils disponibles.

Rao per Canviar el Nombre de Blocs en la Dimensió j: Canviar el nombre de blocs en la dimensió j pot afectar la relació entre la computació i la sincronització per diverses raons. Un nombre major de blocs pot reduir la mida de cada bloc, resultant en temps de computació més curts per bloc i, potencialment, una reducció de l'overhead de sincronització si la gestió dels blocs és eficient. D'altra banda, un nombre excessiu de blocs podria portar a un increment de l'overhead de sincronització, contrarestant els guanys de la reducció del temps de computació per bloc. La clau està en trobar un equilibri entre la granularitat de la tasca i l'overhead de gestió associat.

Explicació de les Gràfiques Inclòses: Les gràfiques inclòses haurien de mostrar la escalabilitat forta per als casos de nblocksj=20 i nblocksj=best userparam \* nblocksi. Aquestes gràfiques proporcionen una representació visual de com el canvi en el nombre de blocs afecta el rendiment i l'escalabilitat. En el cas de nblocksj=20, es podria esperar veure un patró d'escalabilitat que mostra l'efecte de tenir una quantitat fixa de blocs a mesura que augmentem el nombre de fils. Amb nblocksj=best userparam \* nblocksi, la gràfica hauria de reflectir l'impacte de l'ajust del nombre de blocs basat en el paràmetre d'usuari òptim, que idealment milloraria tant el rendiment com l'escalabilitat.

**Final Laboratory Survey**

We would like to get some feedback from you so that we can continue improving the practical part of the course. In particular, we are interested in your opinion about the usage of *modelfactors*. Can you please tell us briefly your opinion about it? Was *modelfactors* useful for you in order to understand the performance of your parallel application? Would you advice its use in the laboratory assignments of this course in future editions?

From 0 to 10, how would you rate:

* *modelfactors*;
* *Tareador* ;
* *Extrae + Paraver*.

On the other hand, we have made an effort to reduce the volume of things to deliver in the lab documents, trying to guide more the information we require. From 0 to 10, how would you rate (0 - too

much, 10 - very well) the volume of documentation to be delivered.

Feel free to include any other comment that you want to add about the practical sessions.