PAR2110
5th Deliverable

Èric Casanovas

22/05/2018

INDEX

Introducció

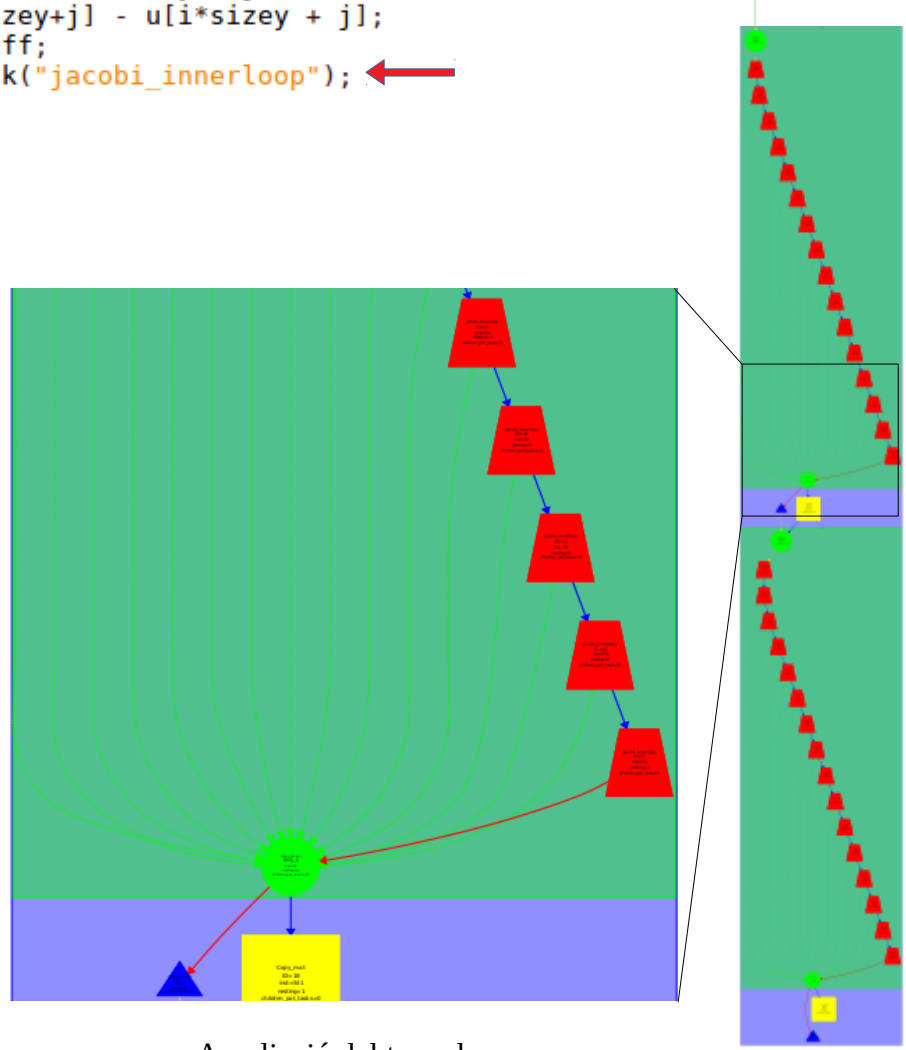
En aquest darrer bloc del laboratori practicarem i observarem diferents tecniques de paralelitzacio per a les funcions de calor Jacobi i Gauss-Seidel.

Analysis with Tareador

Jacobi

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
         tareador_start_task("jacobi_innerloop");
         utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
                                   u[ i*sizey     + (j+1) ]+  // right
                         u[ (i-1)*sizey + j     ]+  // top
                         u[ (i+1)*sizey + j     ]); // bottom
         diff = utmp[i*sizey+j] - u[i*sizey + j];
         sum += diff * diff;
         tareador_end_task("jacobi_innerloop");
       }
        }
    }

    return sum;
}
```

Podem observar com hem afegit les 2 linies per observar les dependencies dins el bucle mes intern de la funció relax_jacobi, i podem veure com totes les iteracions depenen de l'anterior per continuar i a més a més (tot i no observar-se gaire bé al tareador degut als colors) han de finalitzar totes les iteracions per poder continuar. Doncs podem afirmar observant el codi que la variable «sum» es qui dona problemes.

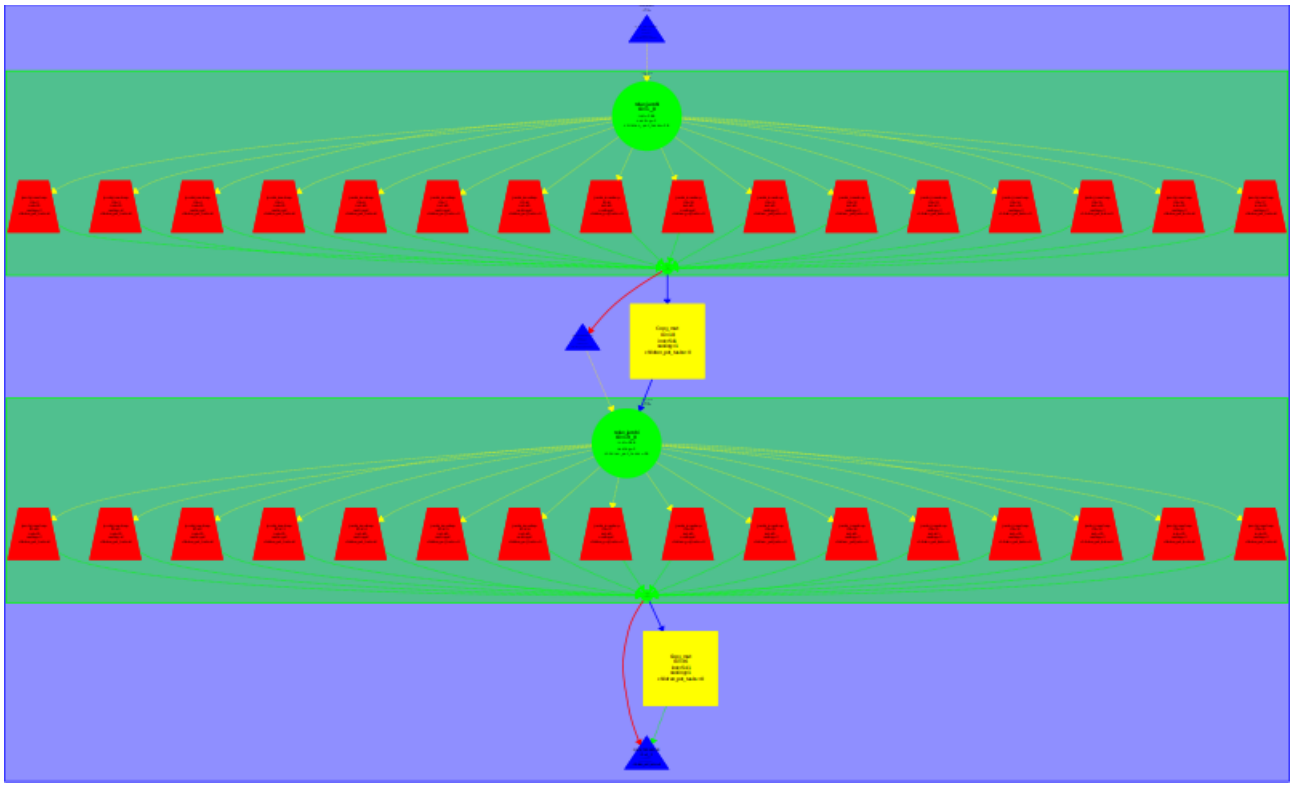


Ampliació del tareador

Resultat Tareador

Per demostrar que era la variable sum la que creava dependencia hem afegit aquestes dues linies al codi i hem executat el tareador.

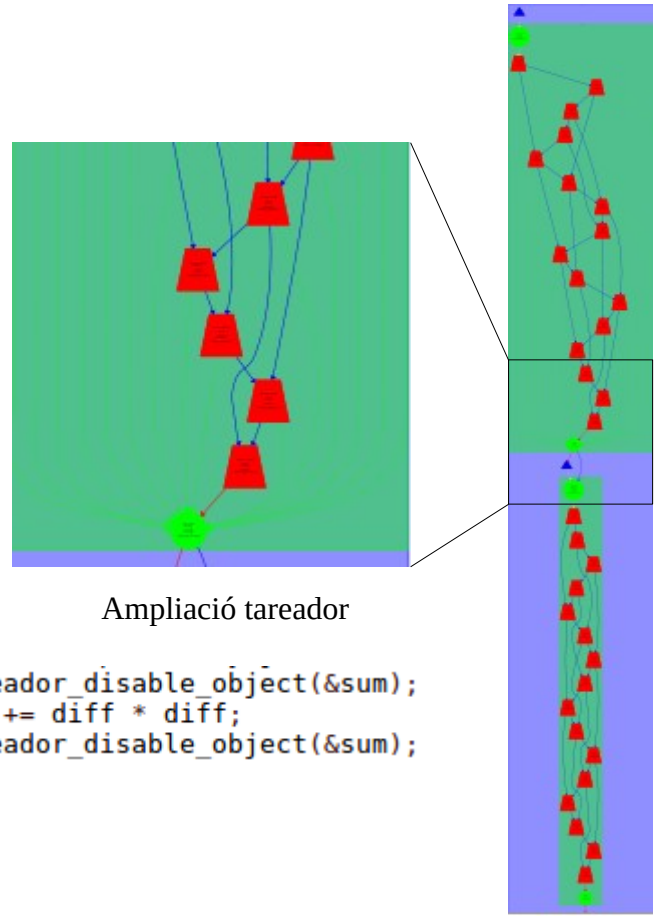
```
tareador_disable_object(&sum);
sum += diff * diff;
tareador_disable_object(&sum);
```



Observem ara com ja no tenen dependencies entre iteracions del bucle, doncs hem vist que la variable sum es la causant de les dependencies.

Gauss-Seidel (codi següent pàgina)

En aquest cas observem com sembla que les dependencies siguin practicament aleatories, però segueixen depenent de la variable sum. Per tant seguirem la mateixa estrategia que hem fet servir al Jacobi que serà descativar la variable sum pel tareador i observar com es comporta el programa.

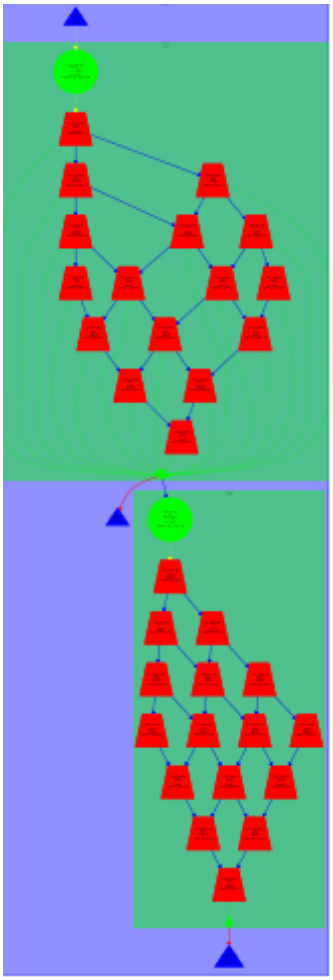


Ampliació tareador

Un cop afegides aquestes tres linies tornem a executar el tareador i observem, les dependencies.

```
tareador_disable_object(&sum);
sum += diff * diff;
tareador_disable_object(&sum);
```

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
        tareador_start_task("gauss_innerloop");
        unew= 0.25 * ( u[ i*sizey   + (j-1) ]+  // left
               u[ i*sizey   + (j+1) ]+  // right
               u[ (i-1)*sizey   + j     ]+  // top
               u[ (i+1)*sizey   + j     ]); // bottom
        diff = unew - u[i*sizey+ j];
        sum += diff * diff;
        u[i*sizey+j]=unew;
        tareador_end_task("gauss_innerloop");
        }
      }
    }

    return sum;
}
```



Observem una millora respecte a les que feiem servir la variable sum dins el tareador, i ens adonem que aquestes dependencies són degut a que per fer el comput de la iteració necessites l'anterior bloc de la matriu i el bloc de sobre teu.

OpenMP parallelization and execution analysis: Jacobi

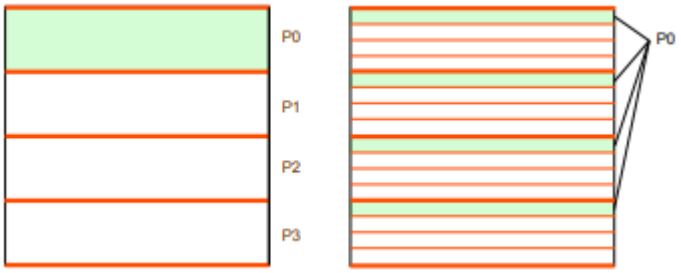
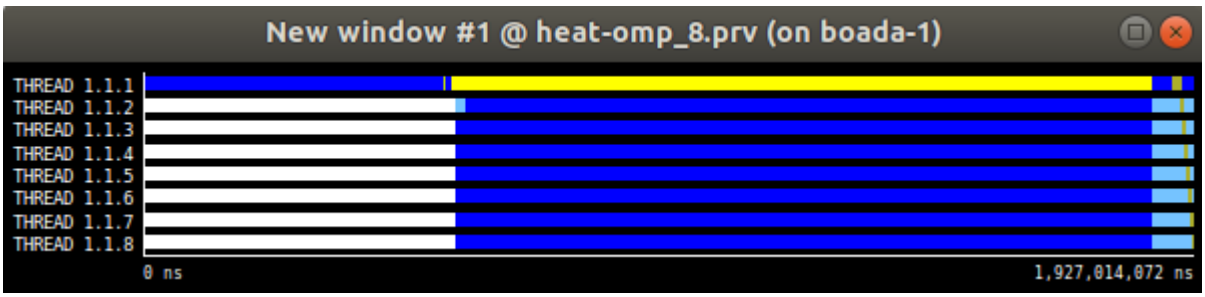


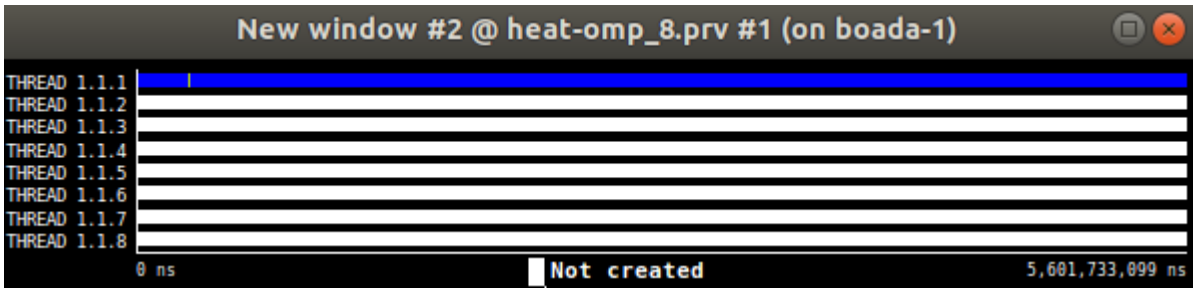
Foto obtinguda de la teoria (esquerra per blocs, dreta ciclic)

La estrategia que nosaltres utilitzarem serà la de l'esquerra, la de per blocs, ho aconseguirem fent uns petits calculs amb el numero total de threads i el numero de thread que l'executara.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;
    #pragma omp parallel private(diff) reduction(+:sum)
    {

        int howmany = omp_get_num_threads();
        int myid = omp_get_thread_num();
        int rem = sizex % howmany;
        int i_start = (myid * sizex/howmany) + (myid<rem ? myid : rem);
        int i_end = i_start + (sizex/howmany) + (myid < rem);

        for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {

            for (int j=1; j<= sizey-2; j++) {

                utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
                                          u[ i*sizey     + (j+1) ]+  // right
                                          u[ (i-1)*sizey + j     ]+  // top
                                          u[ (i+1)*sizey + j     ]); // bottom
                diff = utmp[i*sizey+j] - u[i*sizey + j];
                sum += diff * diff;

            }

        }

    }

    return sum;
}
```

Aqui tenim el codi amb la paralelitzacio per blocs observarem una millora de temps d'execució respecte la versió seqüencial com observarem a les següents fotos del paraver, ja que el que fara en comptes de executar tot el bucle for en ordre seqüencialment, dividir el for en bloc de total de iteracions entre numero total de threads i assignara a cada thread una part proporcional del del bucle for. Això podria tenir el problema de que hem de respectar la dependencia que té la variable sum respecte la seva anterior iteració, però, per això tenim la directiva reduction(+:sum) que fara que cada thread tingui una copia de sum i executi la seva part del bucle calculant el valor de sum, i al final al acabar el for unir totes les variables sum de cada thread en una sola sumant-les, fent que tingues el valor esperat. Per tant, imaginem el cas que tinguem 4 threads, el for es dividiria entre 4 parts iguals i se li assignaria a cada thread una de les parts, doncs, aproximadament el programa trigarà un quart del que tardaria el seqüencial (hem de contar que no seria exactament un quart perquè hem de contar amb l'overhead que serà bastant gran).

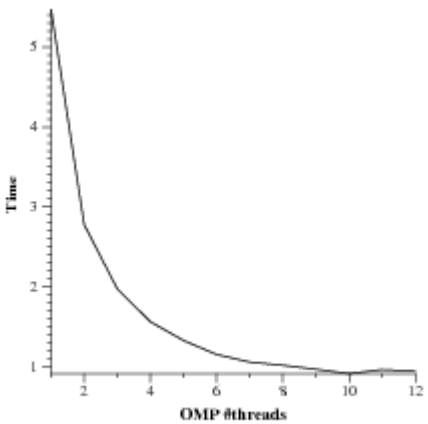




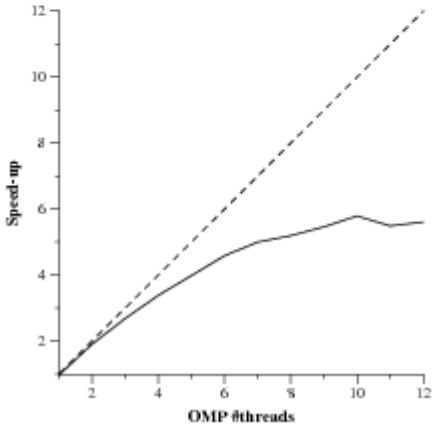
Veiem que mes o menys es el que hem predit el temps del segon entre el del primer ens dona 2,9.

A més a més també hem paralelitzat la funcio copy_mat:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```



Average elapsed execution time



Speed-up wrt sequential time

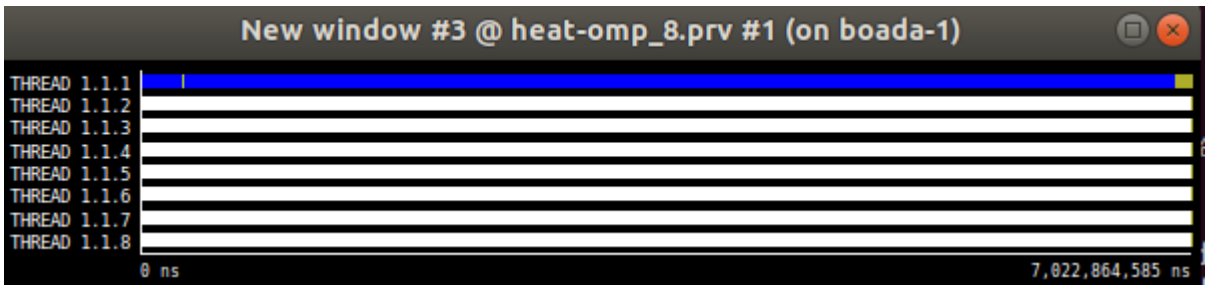
Pel que fa a l'strong scalability, observem que escala força bé, però observem que al ficar molts threads fem que l'overhead sigui ja bastant més gran que el que tarda en executar-se la part paralela, per tant a partir de uns 10 threads no seria convenient colocar-ne més perquè el guany seria minim i fins i tot podria perjudicar al temps d'execució del programa.

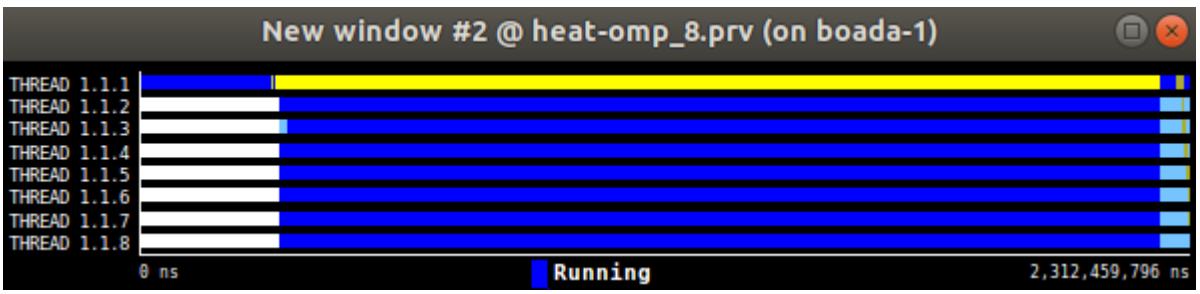
OpenMP parallelization and execution analysis: Gauss-Seidel

En aquest cas haurem de fer la data decomposition amb un pragma omp for.

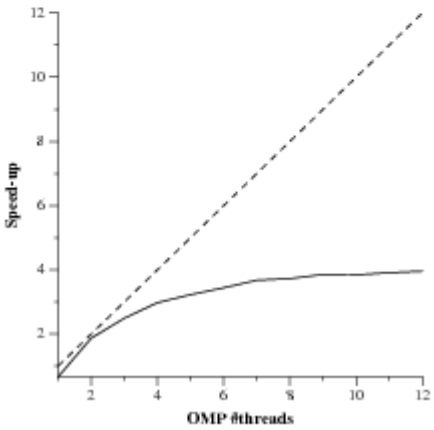
```
double relax_gauss (double *u, unsigned sizex, unsigned sizey){
    double unew, diff, sum=0.0;
    int howmany = omp_get_max_threads();
    int processed[howmany];

    #pragma omp parallel for
    for (int i = 0; i < howmany; ++i) {
        processed[i] = 0;
    }
    int nb = 8;

    #pragma omp parallel for private(diff, unew) reduction(+ : sum)
    for (int i = 0; i < howmany; ++i) {
        int ii_start = lowerb(i, howmany, sizex);
        int ii_end = upperb(i, howmany, sizex);
        for (int j = 0; j < nb; j++){
            if (i > 0){
                while (processed[i-1] <= j){
                #pragma omp flush
                }
            }
            int jj_start = lowerb(j,nb,sizey);
            int jj_end = upperb(j,nb,sizey);
            for (int ii = max(1, ii_start); ii <= min(sizex-2, ii_end); ii++) {
                for (int jj= max(1, jj_start); jj <= min(sizey-2, jj_end); jj++){
                    unew = 0.25* (u[ii * sizey + (jj-1)] +  // left
                                  u[ii * sizey + (jj+1)] +  // right
                                  u[(ii-1) * sizey + jj] +  // top
                                  u[(ii+1) * sizey + jj]); // bottom
                    diff = unew - u[ii * sizey + jj];
                    sum += diff*diff;
                    u[ii*sizey+jj] = unew;
                }
            }
            ++processed[i];
            #pragma omp flush
        }
    }
    return sum;
}
```

Hem fet un omp for amb les variables diff i wnew privades per a que no comparteixin valor entre threads i pugui haver problemes. També hem afegit un reduction per al final sumar totes les variables sum tal i com hem fet al Jacobi. Per a que la paralelització funciones haviem de crear també un vector amb mida nombre de vectors que ens contara quantes files fara cada thread i anira actualitzant els blocas anteriors pels quals passem amb el omp flush per a que així tinguem una correcta sortida.

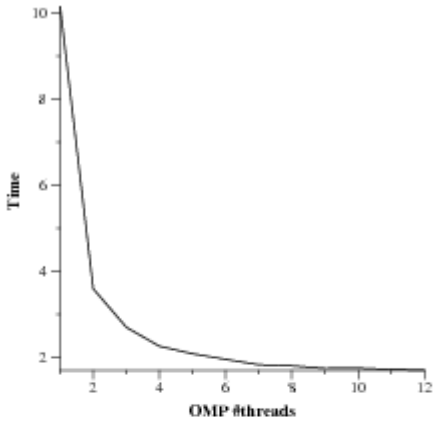




Observem una millora considerable despres d'haver fet la paralelitzacio.



Speed-up wrt sequential time



Average elapsed execution time

Per acabar pel que fa l'strong scalability podem observar com augmenta be, però al arribar a tenir 4 threads fem que l'overhead sigui ja bastant més gran que el que tarda en executar-se la part paralela, per tant a partir d'aquest punt no creix tant l'speedup i podriem considerar que ja no surt a compte ficar mes threads.

Conclusió

En aquest laboratori hem après a paralelitzar fent data decomposition i ho hem posat en practica amb els algoritmes de Jacobi i Gauss-Seidel. Hem observat també l'escalabilitat que tenien aquests 2 algoritmes i els seus espectius speedups. Per concluir podriem destacar que la data decomposition te força semblances amb el task decomposition.