PACO Lab5 - Informe

Ixent Cornella, Arnau Roca (PACO1201)

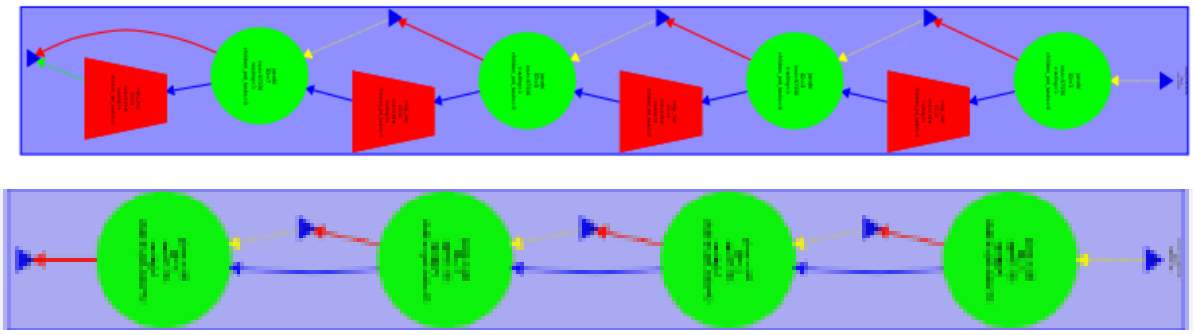
SESSIÓ 1 - Sequential heat diffusion program and analysis with Tareador

Executem primer multisort de forma seqüencial amb sbatch:

```
********************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
********************************************************************************
Initialization time in seconds: 0.681094
Multisort execution time: 5.191687
Check sorted data execution time: 0.011787
Multisort program finished
********************************************************************************
```

Figura 1: Temps d'execució de multisort seqüencial

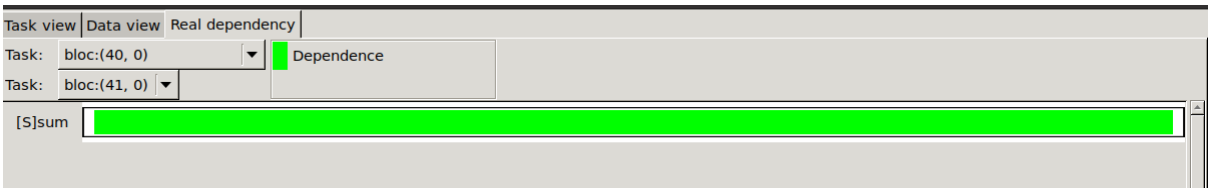
Un cop fetes les dues execucions del jacobi i el gauss-seidel ens adonem que les imatges son diferents, el color blau representa fred i roig calent.
Aquí tenim les sortides de les execucions per jacobi (figura 2) i gauss-seidel (figura 3):



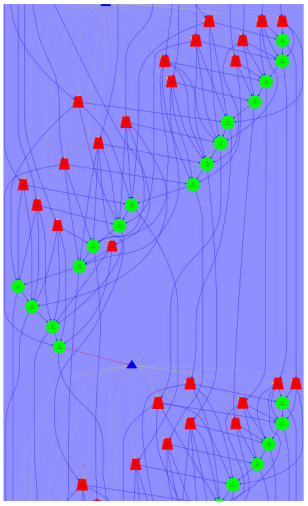
Figures 2 i 3: Grafs de dependència de tasques

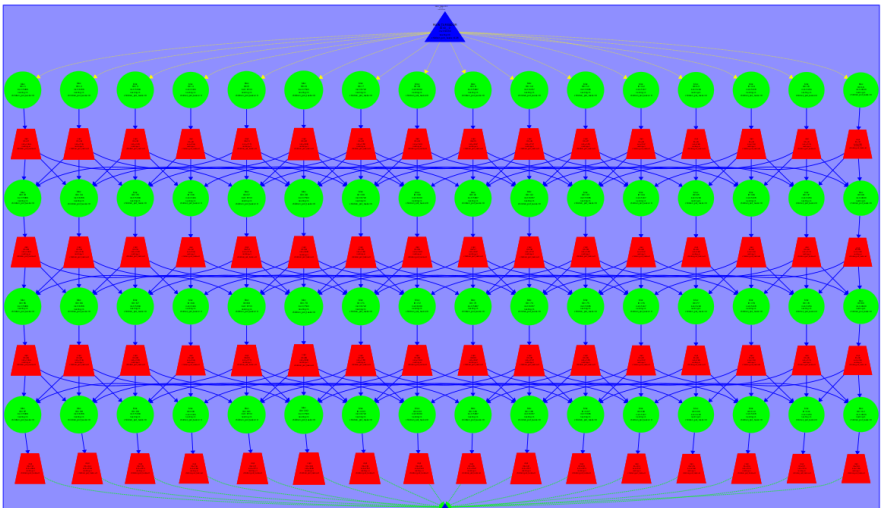
Com podem veure en les imatges i en el codi, hi ha varies parts que podem paral·lelitzar, per tant si volem augmentar la paralelització hauriem de canviar la granularització a una més fina.

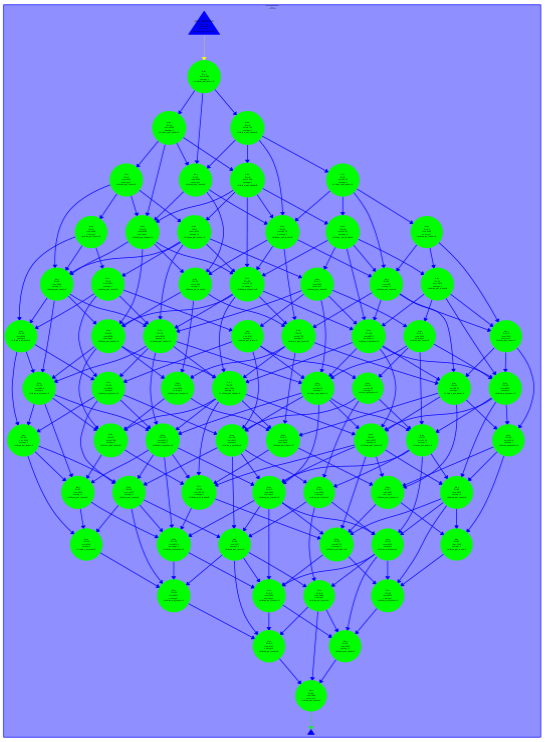
Podem veure la dependència real amb la pestanya del tareador, que ens indica que és sum. Per tant ens quedaria el codi de la següent manera.



Figures 4: Pestanya que ens mostra on està la dependència



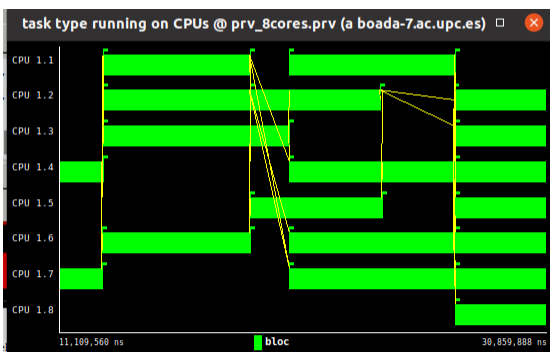


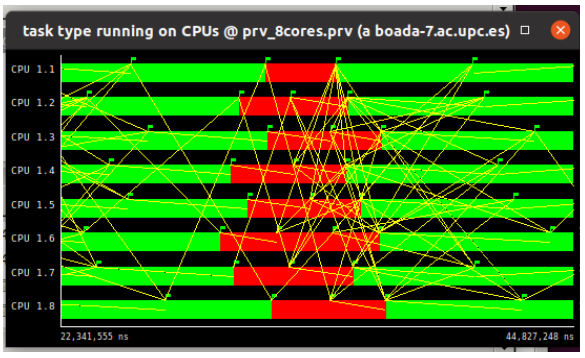


Figures 5, 6 i 7: Pas intermig del paraver i pas final del jacobi i gauss

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {

    int nblocksi=4;
    int nblocksj=4;

    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("copi");
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++)
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++)
            v[i*sizey+j] = u[i*sizey+j];
        tareador_end_task("copi");
        }
    }
}

// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=4;
    int nblocksj=4;

    tareador_disable_object(&sum);
    for (int blocki=0; blocki<nblocksi; ++blocki) {
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        tareador_start_task("bloc");
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
                tmp = 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 = tmp - u[i*sizey+ j];
                sum += diff * diff;
                unew[i*sizey+j] = tmp;
```

Figura 8: codi final del solver





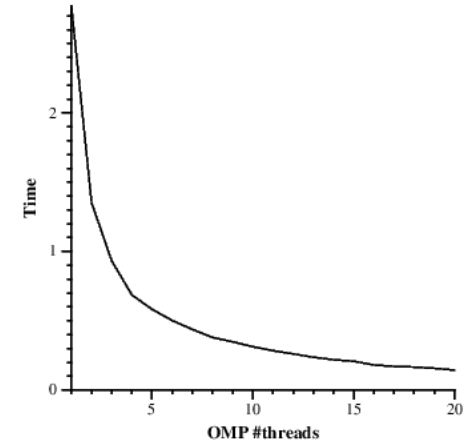
SESSIÓ 2 - Parallelisation of the heat equation solvers (Jacobi)

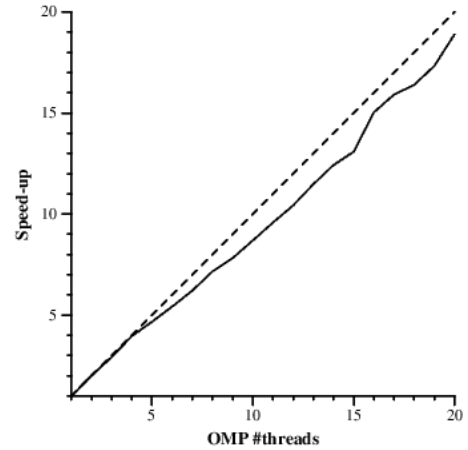
Ara aplicarem el que hem pogut analitzar en la sessió anterior amb el Tareador. Començant pel Jacobi, aquest és el codi que ens quedaria, parallelitzant les dos funcions (copy_mat i solve). En la primera dividim les iteracions en blocs de la mateixa mida i en la segona amb un pragma omp parallel ja aconseguim que el codi pugui ser executat en parallel per diferents threads. Ens quedaria un codi així:

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
  int nblocksi=omp_get_max_threads();
  int nblocksj=1;
  #pragma omp parallel
  {
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++)
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++)
            v[i*sizey+j] = u[i*sizey+j];
      }
  }
}
```

```
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
  double tmp, diff, sum=0.0;
  int nblocksi=omp_get_max_threads();
  int nblocksj=1;

  #pragma omp parallel private(diff) reduction(+:sum)
  {
    int blocki = omp_get_thread_num();
    int i_start = lowerb(blocki, nblocksi, sizex);
    int i_end = upperb(blocki, nblocksi, sizex);
    for (int blockj=0; blockj<nblocksj; ++blockj) {
      int j_start = lowerb(blockj, nblocksj, sizey);
      int j_end = upperb(blockj, nblocksj, sizey);
      for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
        for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
        tmp = 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 = tmp - u[i*sizey+ j];
        sum += diff * diff;
        unew[i*sizey+j] = tmp;
        }
      }
    }
  }
  return sum;
}
```

i el resultat de l'execució millora significativament com podem observar:






```
paco1201@boada-6:~/lab5$ diff heat-jacobi.ppm heat-jacobi-1.ppm
```

Amb el diff comprovem que tot estigui correcte. I amb les taules podem veure que tenim una escalabilitat correcta.

SESSIÓ 3 - Parallelisation of the heat equation solvers (Gauss)

Ens demanen de paralelitzar el codi del solver per fer-ho amb Gauss. D'aquesta manera, el que hem de fer és afegir un pragma de paral·lelisme amb variable privada "diff" i un reduction per a evitar data races a la suma. També cal afegir alguns atomics i reads.

```
// 2D-blocked solver: one iteration step
double solve (double *u, double *unew, unsigned sizex, unsigned sizey) {
    double tmp, diff, sum=0.0;

    int nblocksi=omp_get_max_threads();
    int nblocksj=1;
    int blocks[nblocksi];
    for(int i = 0; i < nblocksi; i++) blocks[i] = 0;

    #pragma omp parallel private(diff) reduction(+:sum)
    {
      int contador;
      int blocki = omp_get_thread_num();
      int i_start = lowerb(blocki, nblocksi, sizex);
      int i_end = upperb(blocki, nblocksi, sizex);
      for (int blockj=0; blockj<nblocksj; ++blockj) {
        int j_start = lowerb(blockj, nblocksj, sizey);
        int j_end = upperb(blockj, nblocksj, sizey);
        if(u ==unew && blocki != 0){
            do{
                #pragma omp atomic read
                contador = blocks[blocki-1];
            }while(contador <= blockj);
        }
        for (int i=max(1, i_start); i<=min(sizex-2, i_end); i++) {
          for (int j=max(1, j_start); j<=min(sizey-2, j_end); j++) {
                tmp = 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 = tmp - u[i*sizey+ j];
                sum += diff * diff;
                unew[i*sizey+j] = tmp;
          }
        }
        if (u == unew){
            #pragma omp atomic write
            blocks[blocki] = blocks[blocki] + 1;
        }
      }
    }

    return sum;
}
```

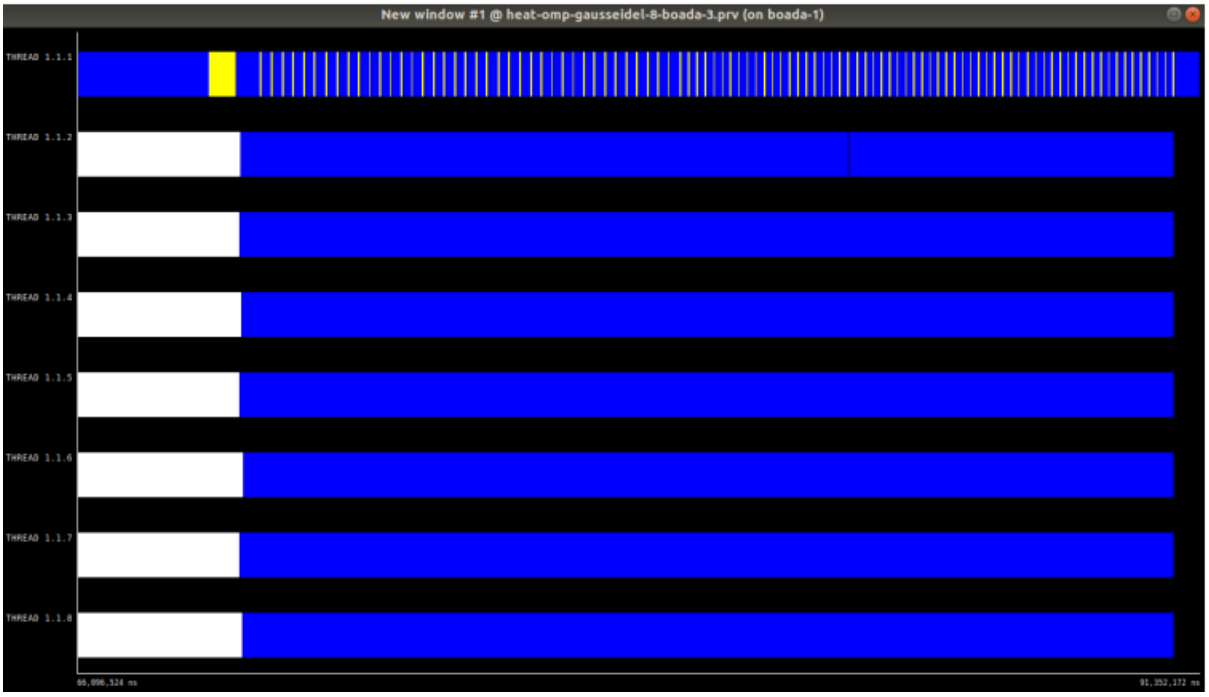
Figura: Així queda el codi ja paralelitzat.

Podem observar que funciona fent el diff amb el resultat que hem obtingut anteriorment i l'execució d'aquest codi:

```
paco1201@boada-6:~/lab5$ diff heat-gausseidel.ppm heat-gauss-1.ppm
paco1201@boada-6:~/lab5$
```

Figura: Resultat del diff buit, que indica que son iguals.

Include the Modelfactor tables, the plot of scalability, and the window timelines or paraver Hints that you consider necessary. Is the scalability observed appropriate? Is there any metric reported by modelfactors.py that you should further investigate? Do you think we can increase the parallelism?



Overview of the Efficiency metrics in parallel fraction, ϕ=99.58%				
Number of processors	1	4	8	16
Global efficiency	99.96%	98.29%	92.59%	89.84%
Parallelization strategy efficiency	99.96%	62.61%	99.93%	99.93%
Load balancing	100.00%	62.65%	100.00%	99.99%
In execution efficiency	99.96%	99.94%	99.94%	99.93%
Scalability for computation tasks	100.00%	156.99%	92.65%	89.91%
IPC scalability	100.00%	53.04%	54.33%	48.95%
Instruction scalability	100.00%	299.44%	185.99%	205.94%
Frequency scalability	100.00%	98.85%	91.70%	89.18%

Table 2: Analysis done on Wed Dec 21 02:03:43 PM CET 2022, paco1201

Statistics about explicit tasks in parallel fraction				
Number of processors	1	4	8	16
Number of implicit tasks per thread (average us)	1000.0	1000.0	1000.0	1000.0
Useful duration for implicit tasks (average us)	6859.6	4369.45	7403.45	7629.76
Load balancing for implicit tasks	1.0	0.63	1.0	1.0
Time in synchronization implicit tasks (average us)	0	0	0	0
Time in fork/join implicit tasks (average us)	2.69	5218.63	5.11	5.63

Table 3: Analysis done on Wed Dec 21 02:03:43 PM CET 2022, paco1201

Overview of whole program execution metrics				
Number of processors	1	4	8	16
Elapsed time (sec)	6.89	7.01	7.44	7.67
Speedup	1.00	0.98	0.93	0.90
Efficiency	1.00	0.98	0.93	0.90

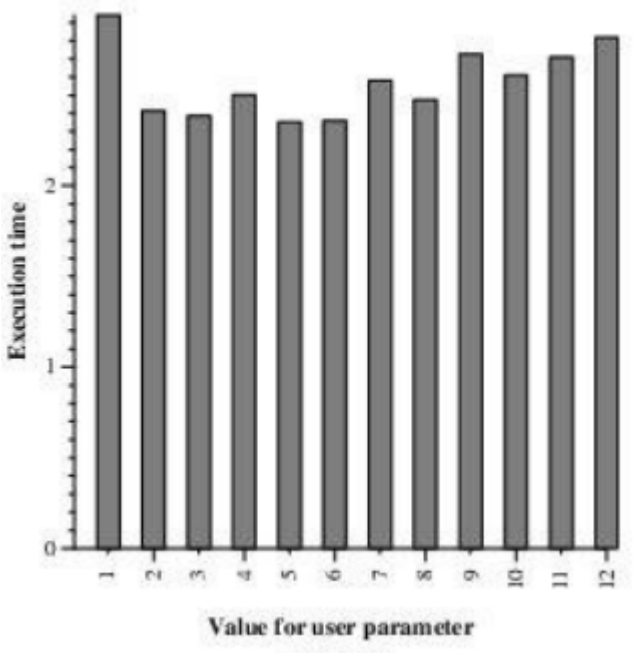
Table 1: Analysis done on Wed Dec 21 02:03:43 PM CET 2022, paco1201

En estudiar l'escalabilitat de la solució de Gauss-Seidel, vam observar una escalabilitat més baixa que en la versió de Jacobi. En examinar els temps

d'execució i els speed-ups d'aquesta versió i tenint en compte els temps d'execució de les versions de Jacobi amb *copy_mat* paral·lelitzat i no paral·lelitzat, vam concloure que, encara que el solucionador de Gauss-Seidel no respon tan bé a la paral·lelització com Jacobi, això es deu a la funció *copy_mat*.

Per millorar l'eficiència del programa, vam provar diferents valors per a la dimensió j. Per a testejar diferents valors sense haver de recompilar i poder fer els gràfics per observar el seu comportament, hem de modificar el codi i fer la següent assignació: *nblocksj = parametre * nblocksi*.

Amb el codi modificat procedim a executar el submit-userparam-omp.sh amb diferents valors pel nombre de fils i obtenim els següents resultats (8 threads):




```
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: 5.826
Flops and Flops per second: (8.806 GFlop => 1511.51 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```