LAB 5

Geometric (data) decomposition: heat diffusion equation

2018-2019 Q1

Par2013

Daniel Palomo Cabrera i David Soldevila Puigbi

Introduction

In the last session of Par, we are going to study the potential parallel performance of two heat diffusion algorithms, Jacobi and Gauss-Seidel. Then we are going to parallelize the code using OpenMP.

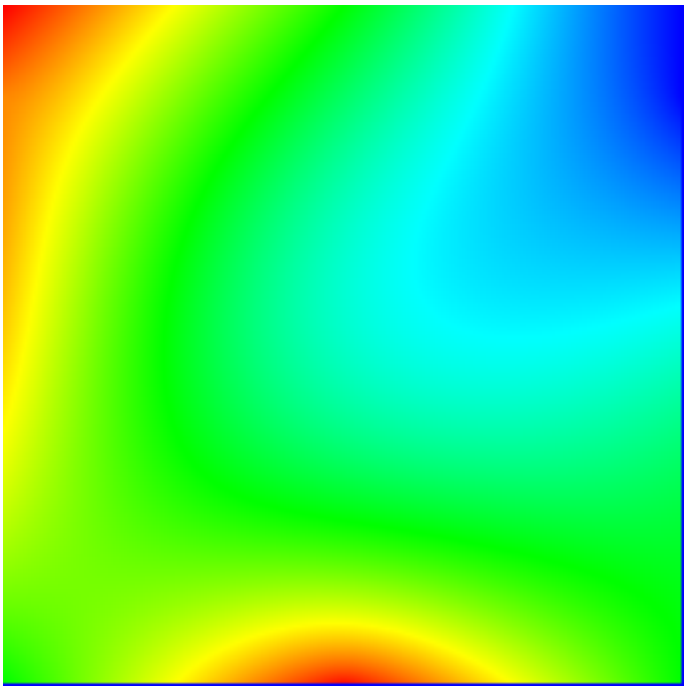
Sequential heat diffusion program

First of all, lets execute the sequential version of heat twice, one using Jacobi algorithm and the other using Gauss-Seidel algorithm.

Jacobi solver:

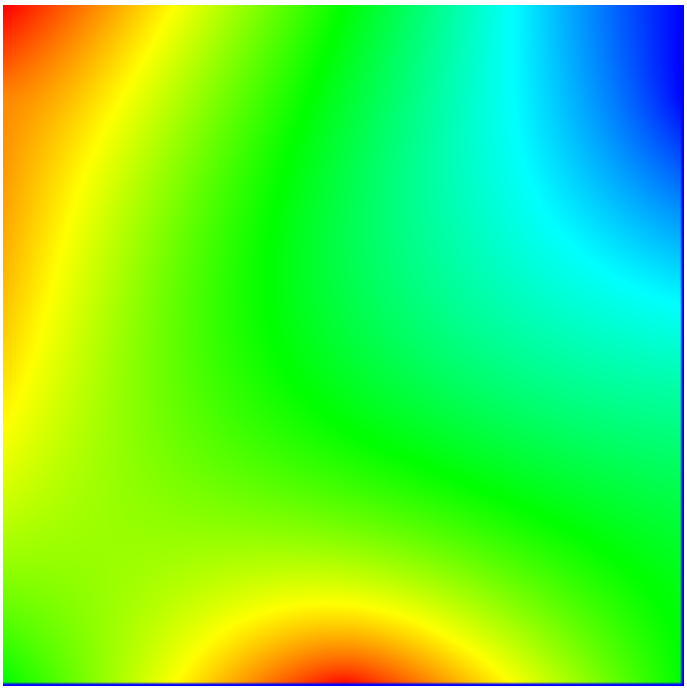
```
Iterations        : 25000
Resolution        : 254
Algorithm         : 0 (Jacobi)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 5.365
Flops and Flops per second: (11.182 GFlop => 2084.06 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

Result heat map:



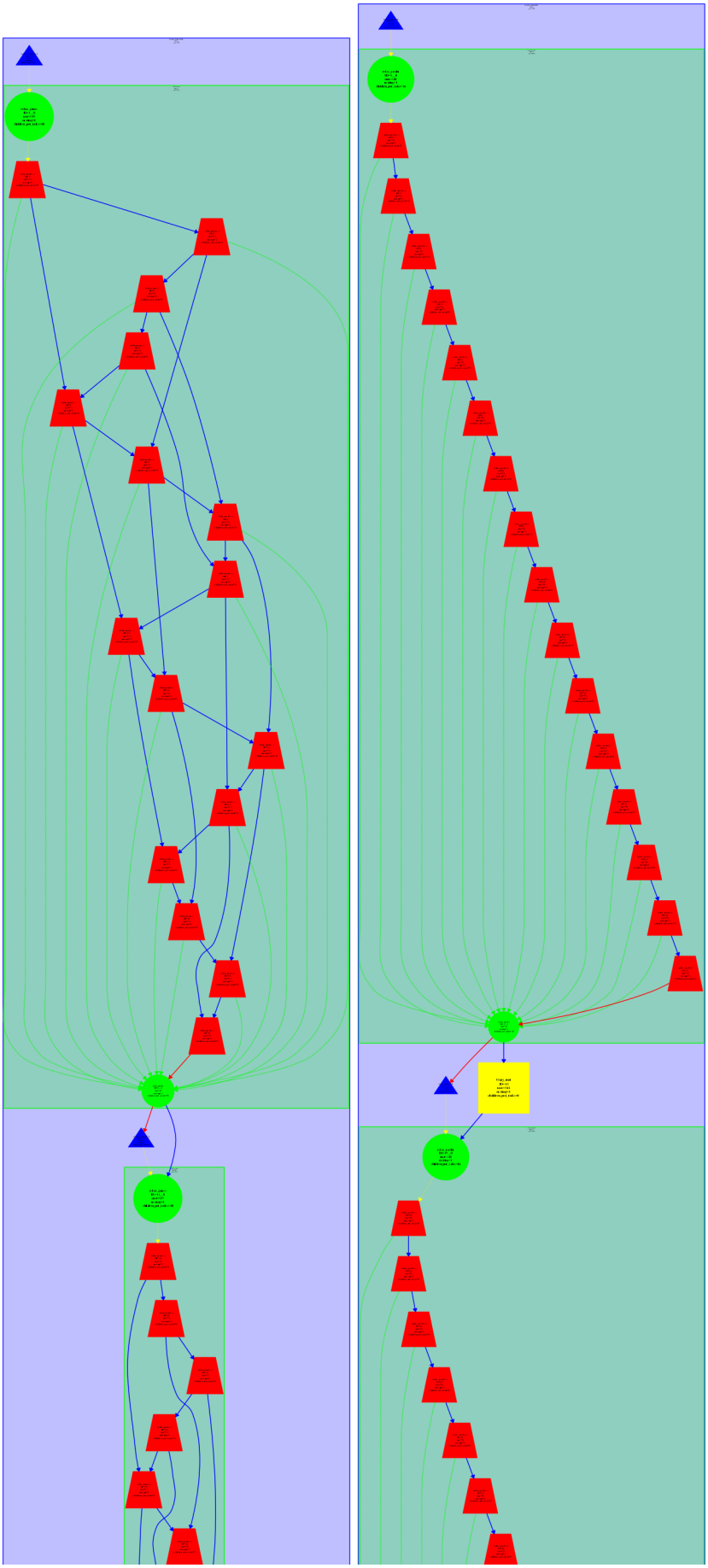
```
Iterations        : 25000
Resolution        : 254
Algorithm         : 1 (Gauss-Seidel)
Num. Heat sources : 2
   1: (0.00, 0.00) 1.00 2.50
   2: (0.50, 1.00) 1.00 2.50
Time: 6.305
Flops and Flops per second: (8.806 GFlop => 1396.78 MFlop/s)
Convergence to residual=0.000050: 12409 iterations
```

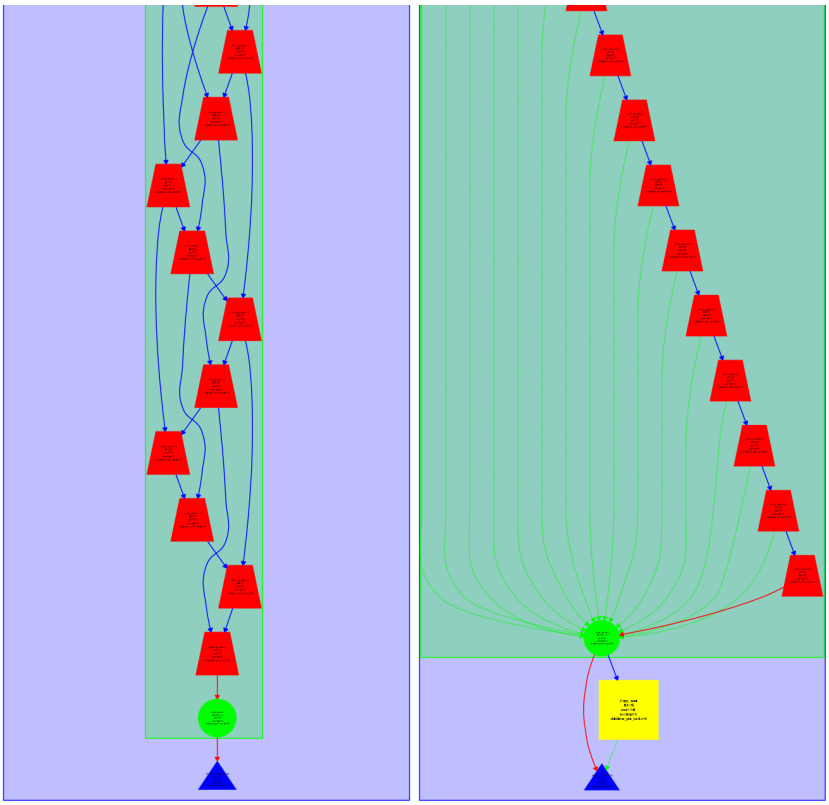
Result heat map:



Analysis with Tareador

Now we are going to study Tareador dependences graphs. We got two differents graphs, one with Jacobi solver algorithm and another with Gauss-Seidel algorithm.





Dependnece graph of the program using Gauss-Seidel and Jacovi algorthms.

Observing the first dependence graph we can conclude that there are data dependences from sum. Dependences come from two different iterations in some cases.

As we can observe at the Jacobi graph also exists a data dependence between iterations of jacobi_rexlax loop. The variable that cause that dependence is sum form the previous iteration. It could be parallelized using a reduction.

Parallelization of Jacobi With OpenMP parallel

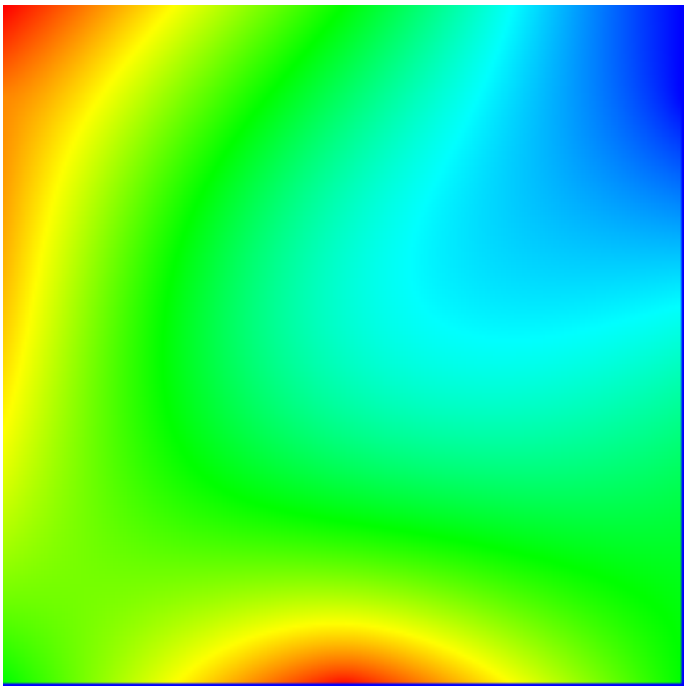
In order to parallelize the Jacobi solver function we used a parallel region with a reduction sum and declaring diff as a private variable. We tried to replace the reduction clause with atomic or critical on sum, but we didn't manage to get a satisfactory performance, or any at all. This may because the optimizations within the reduction.

The resulting code is:

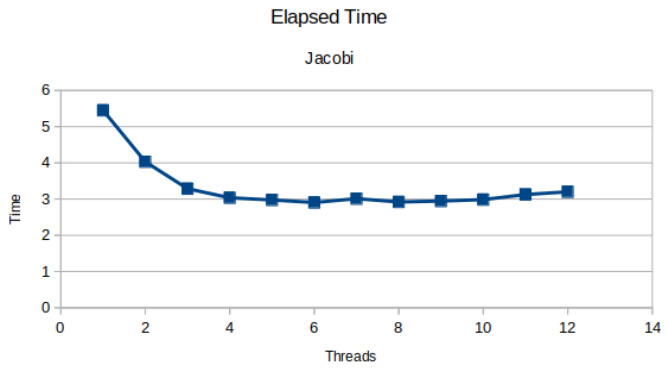
```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned
sizey)
{
  double diff, sum=0.0;
  #pragma omp parallel private(diff) reduction (+:sum)
  {
    int funci= omp_get_thread_num();
    int thre = omp_get_num_threads();
    int i_start = lowerb(funci, thre, sizex);
    int i_end = upperb(funci, thre, sizex);
    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;
}
```

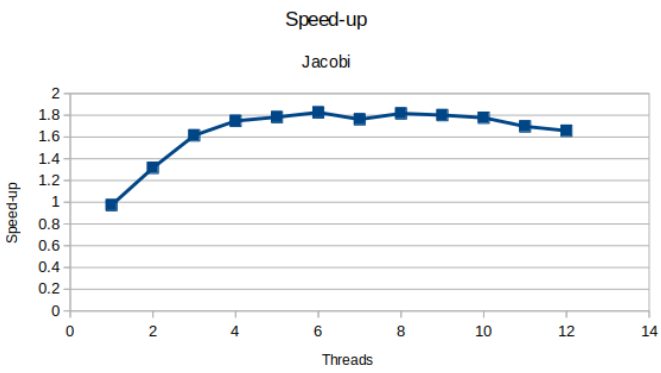
solver-omp.c

With that code we got the following heat map.

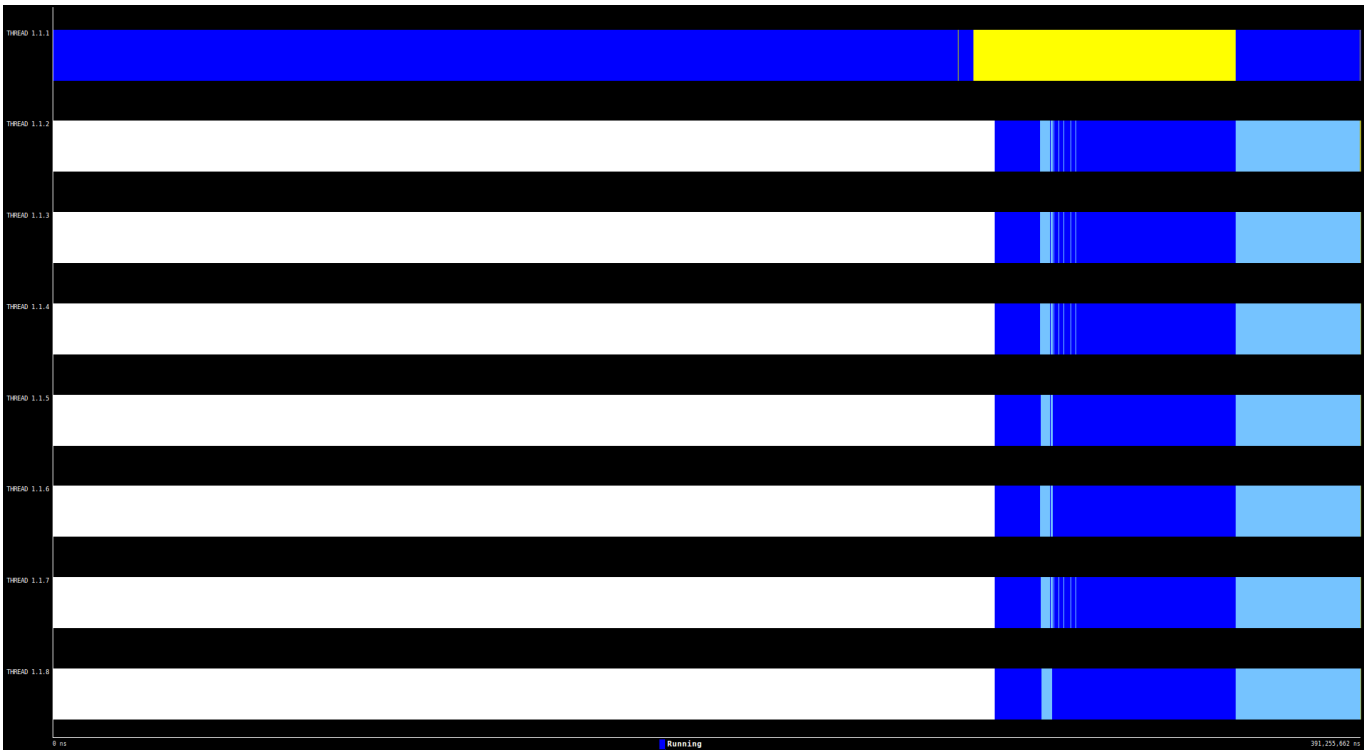


With that parallelization strategy we got the following plots:





Here we can see a clearly stop in performance gain at 4 cores, we aren't sure why it stops so abruptly, maybe is because the nature of the code, or an undersight by our part. We also see a small decrease in speedup, (and its consequentially rise in elapsed time) at eleven threads, probably due to overheads.



Time line of the threads

	Running
THREAD 1.1.1	503
THREAD 1.1.2	250
THREAD 1.1.3	250
THREAD 1.1.4	250
THREAD 1.1.5	250
THREAD 1.1.6	250
THREAD 1.1.7	250
THREAD 1.1.8	250

OpenMP task instantation

As we can see at the timelines and at the table of instances every thread of the processor gets the same amount of work, excepting the thread 1. That one is the manager of tasks and makes all the previous work of the program.

Parallelization of Gauss-Seidel with OpenMP ordered

To parallelize the Gauss algorithm we must use ordered clause. It indicates a block of code that has to be executed sequentially, additionally you can specify a number of variables as a dependency of that execution.

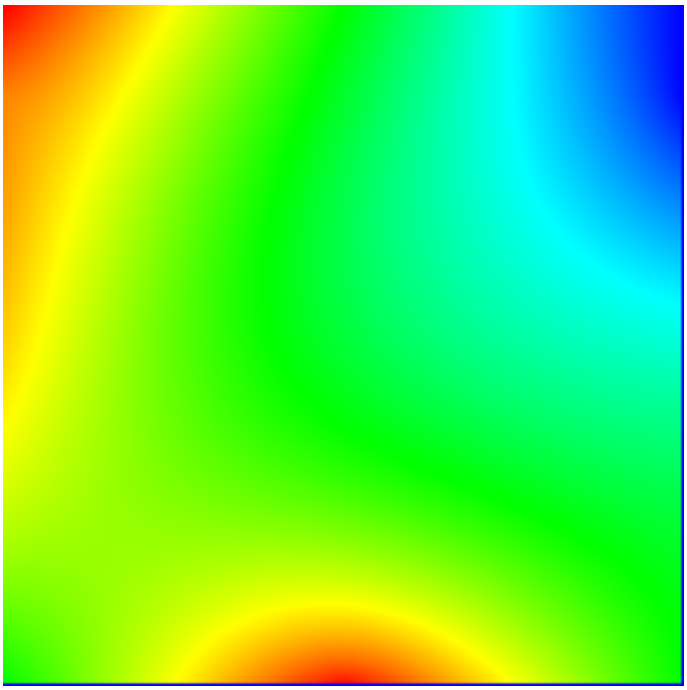
We added to the ordered clause at the pragma omp parallel and then we specified the ordered region with a source dependence. That region includes the peace of code that process one chunk.

Using that principle the resulting code is:

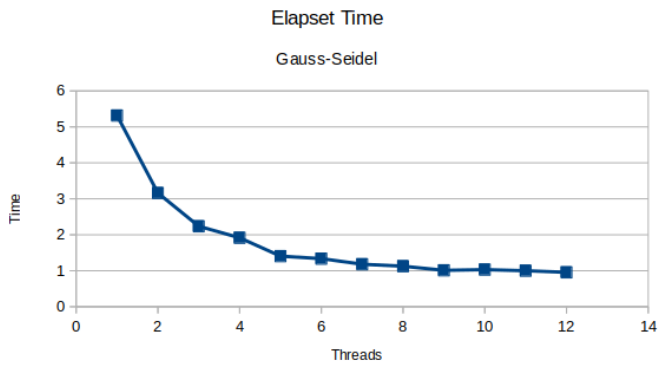
```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    int howmany = 24;
    int chunkx = sizex/howmany;
    int chunky = sizey/howmany;
    #pragma omp parallel for reduction(+:sum) ordered(2)
    for(int bx = 0; bx < sizex/chunkx; bx++){
        for(int by = 0; by < sizey/chunky; by++){
            int i_start = lowerb(bx, howmany, sizex);
            int i_end = upperb(bx, howmany, sizex);
            int j_start = lowerb(by, howmany, sizey);
            int j_end = upperb(by, howmany, sizey);
            #pragma omp ordered depend(source)
            {
                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++) {
                        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;
                    }
                }
            }
        }
    }
  return sum;
}
```

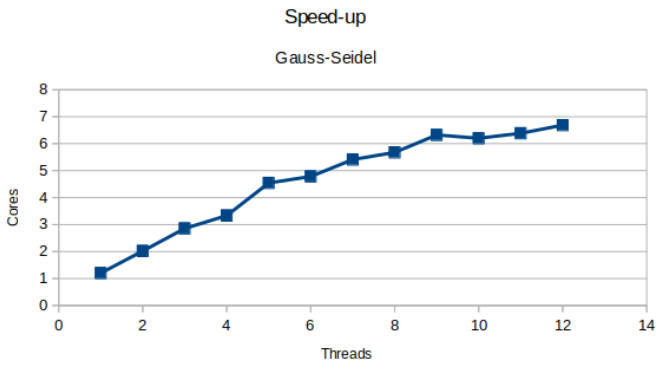
solver-omp.c

With that code we got the following heat map.



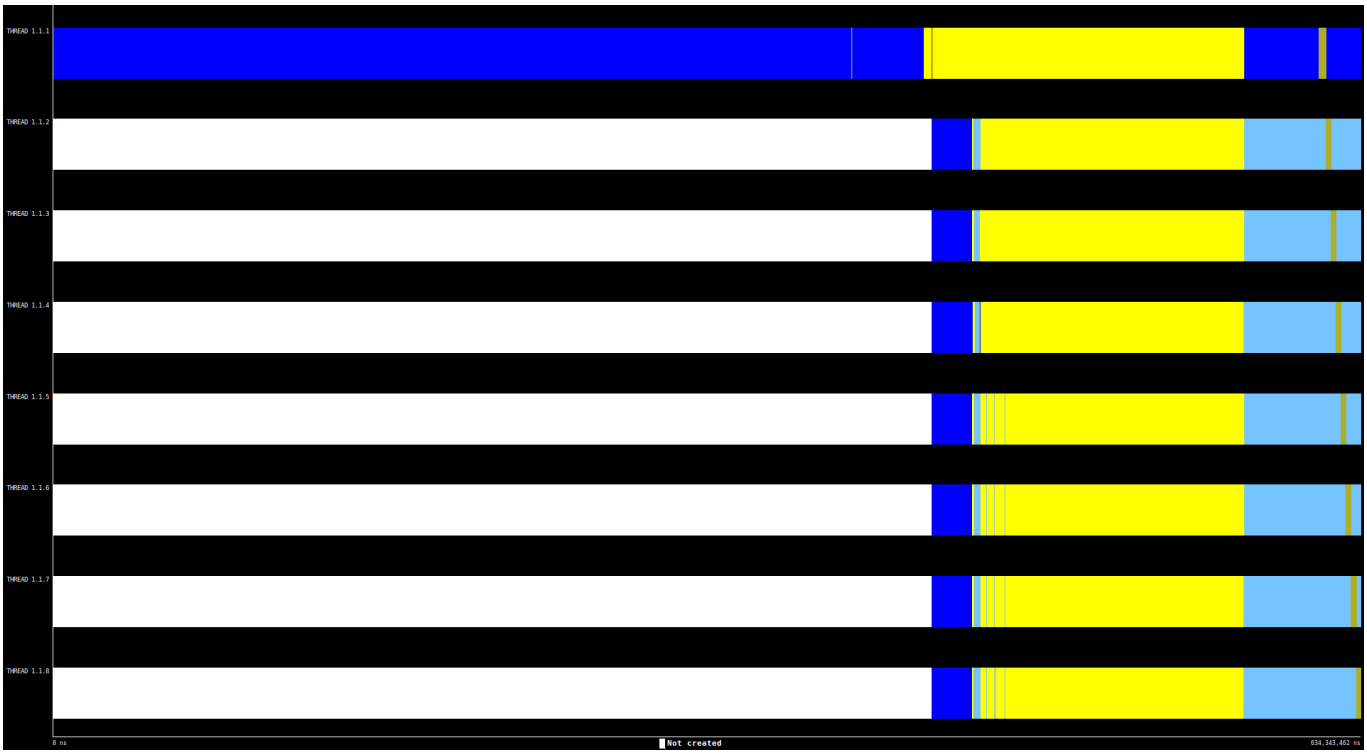
The resulting plots of the strong scalability of that code are:





Now let's study the traces with Paraver. The execution with Extrae has only done 250 iterations in order to generate a reasonable trace.

We can observe that it reaches the limit of performance at 6 threads. In comparation with Jacobi solver, that method is more paralellizable and perform better. At 12 cores Gauss solver is about 3 times faster than the other one. And also the paralellization limit is higher. Now let's study the traces with Paraver. The execution with extra has only done 250 iterations in order to generate a reasonable trace.



Time line of the threads

	Running
THREAD 1.1.1	503
THREAD 1.1.2	250
THREAD 1.1.3	250
THREAD 1.1.4	250
THREAD 1.1.5	250
THREAD 1.1.6	250
THREAD 1.1.7	250
THREAD 1.1.8	250

OpenMP task instantation

Equal than the previous traces, thread 1 has more work than the others. However the amount of tasks per thread is the same.

Conclusions

There is more than task decomposition strategies to parallelize a code. Data decomposition is useful when the parallelisation constrain is the access to a matrix or a vector.