LAB 5: DELIVERABLE

Alfonso Toro Roll
Albert Figuera Pérez
PAR1313
Q1 2017/2018

INTRODUCTION

Carl Gustav Jacob Jacobi was a German mathematician, who made fundamental contributions to elliptic functions, dynamics, differential equations, and number theory. His name is occasionally written as Carolus Gustavus Iacobus Iacobi in his Latin books, and his first name is sometimes given as Karl.

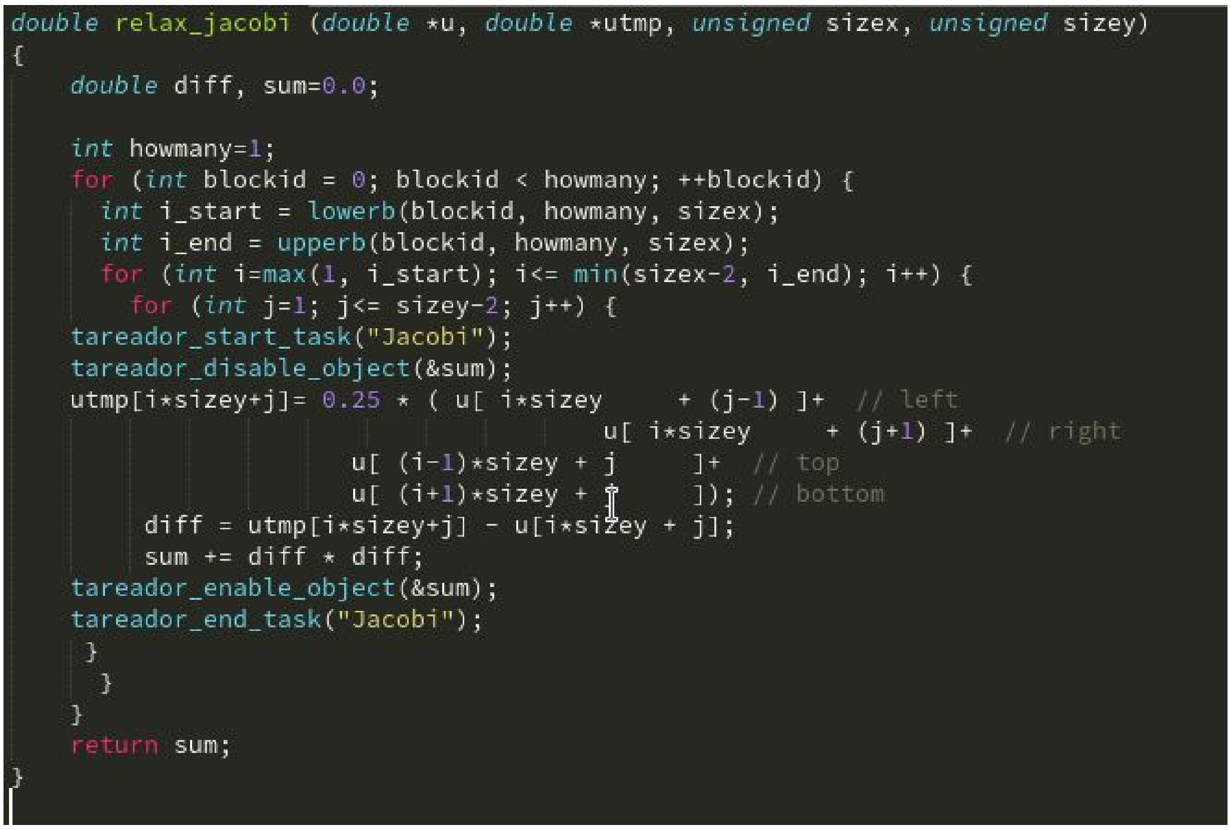
Johann Carl Friedrich Gauss was a German mathematician who made significant contributions to many fields, including number theory, algebra, statistics, analysis, differential geometry, geodesy, geophysics, mechanics, electrostatics, magnetic fields, astronomy, matrix theory, and optics.

In this session we will work on the parallelization of a sequential code (heat.c) 1 that simulates heat diffusion in a solid body using two different solvers for the heat equation Jacobi and Gauss-Seidel.

We analyzed the potential parallelism and the dependence of the code with both models with Tareador, after this we parallelized the code using the OpenMP commands and analyzed the scalability of the code.

Analysis with Tareador

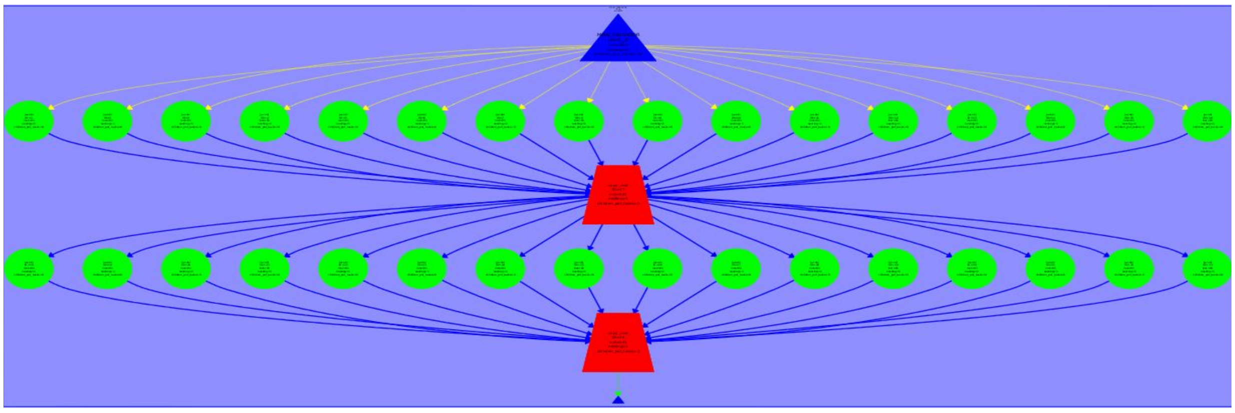
1.



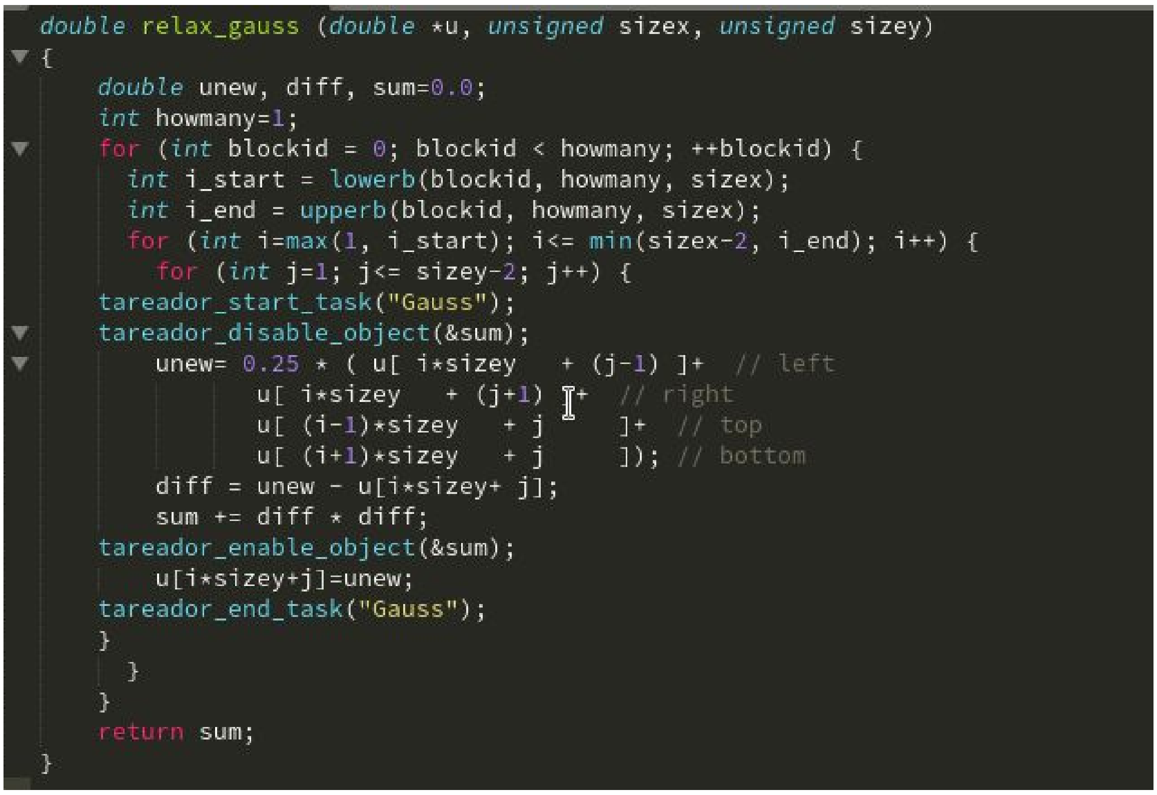
```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    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");
    tareador_disable_object(&sum);
    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_enable_object(&sum);
    tareador_end_task("Jacobi");
     }
      }
    }
    return sum;
}
```

Creating task in the second for of the function and disabling the sum variable, because it was causing dependencies that wouldn't let us parallelize our code, we get the next Tareador dependency graph:

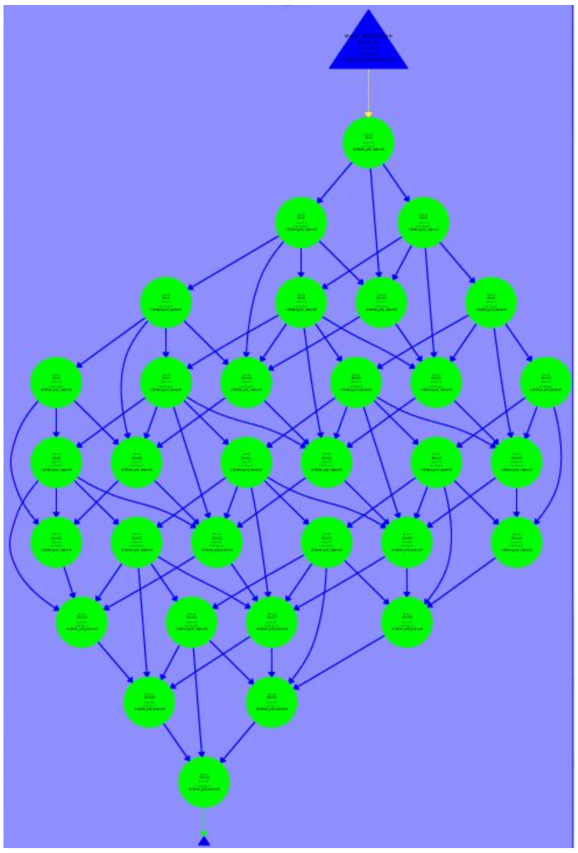


Then, we do the same with the Gauss-Seidel algorithm:



```
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");
    tareador_disable_object(&sum);
        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;
    tareador_enable_object(&sum);
        u[i*sizey+j]=unew;
    tareador_end_task("Gauss");
    }
      }
    }
    return sum;
}
```

And generate the tareador dependency graph.

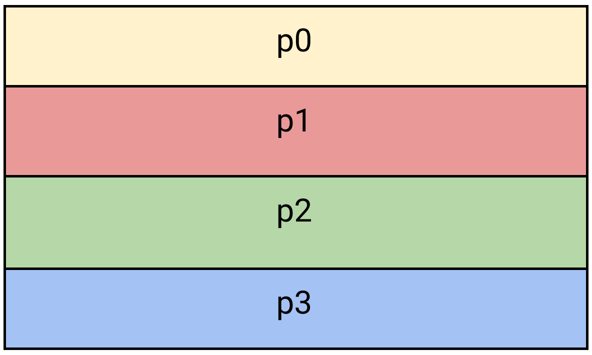


As we can see the jacobi and Gauss-Seidel approaches have different levels of parallelization and thus we can use different techniques when working with OpenMP.

For the Gauss-Seibel approach we would use depend clauses.

OpenMP parallelization and execution analysis: Jacobi

1.



The data decomposition technique used is Geometric Data Block Decomposition, the data is divided in 4 smaller block and each block is executed by one thread. In this case the number of processors is 4 so each block has a size of N/4 rows (being N the total number of rows to compute).

2.

Using #pragma omp for private(diff) reduction(+:sum)

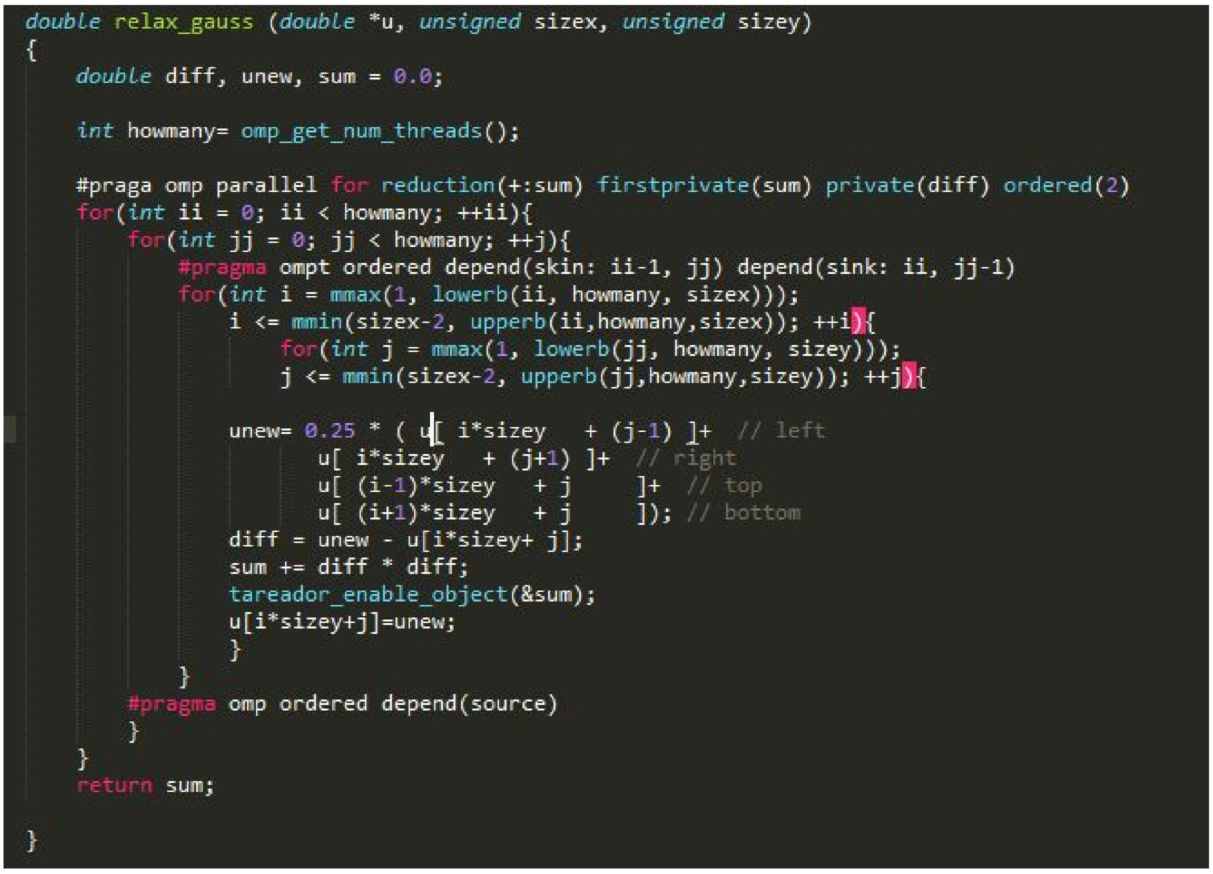
This way the compiler creates a private copy of sum that is properly initialized to the identity value, and at the end of the region the compiler updates sum with the partial values of each thread, using the + operator. We need to make the diff variable private so that each thread has a copy of it.

3.

The execution time and speed-up is worse than we first imagined, but trying to parallelize other parts of the code made it even worse, at the end we haven't figured out which part of the code should be parallelized and there's obviously a better implementation to solve the problem.

OpenMP parallelization and execution analysis: GaussSeidel 1

We want to parallelized relax_gauss dividing the matrix in rows of blocks. Each row will be assigned to a single thread.



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

    int howmany= omp_get_num_threads();

    #praga omp parallel for reduction(+:sum) firstprivate(sum) private(diff) ordered(2)
    for(int ii = 0; ii < howmany; ++ii){
        for(int jj = 0; jj < howmany; ++j){
            #pragma ompt ordered depend(skin: ii-1, jj) depend(sink: ii, jj-1)
            for(int i = mmax(1, lowerb(ii, howmany, sizex)));
                i <= mmin(sizex-2, upperb(ii,howmany,sizex)); ++i){
                    for(int j = mmax(1, lowerb(jj, howmany, sizey)));
                    j <= mmin(sizex-2, upperb(jj,howmany,sizey)); ++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;
                tareador_enable_object(&sum);
                u[i*sizey+j]=unew;
                }
            }
        #pragma omp ordered depend(source)
        }
    }
    return sum;

}
```

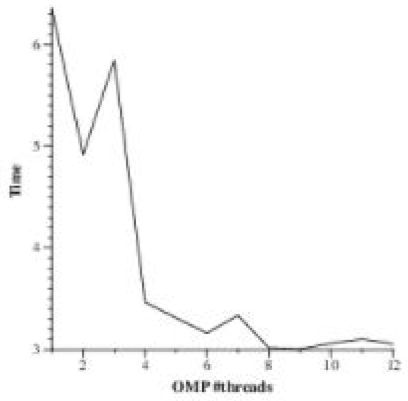
In Gauss one of the importants things is to do is to find the optimum ratio between computation and synchronization.

Using the ordered(n) clause with different block size mesures. Each block is computed with a single thread, and synchronization is achieved using ordered depend pragmas

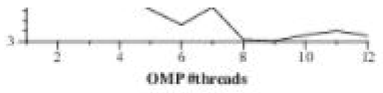
After the use of ordered(2), we obtained that the best option is 32.

This is because it has the smaller execution time. If there are more threads the overhead for synchronization is to high.

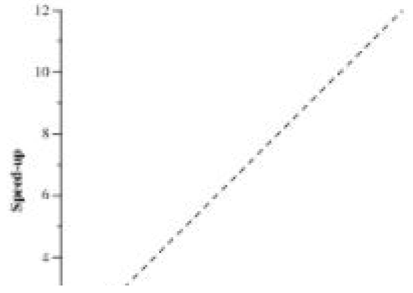
2



Average elapsed execution time



Average elapsed execution time



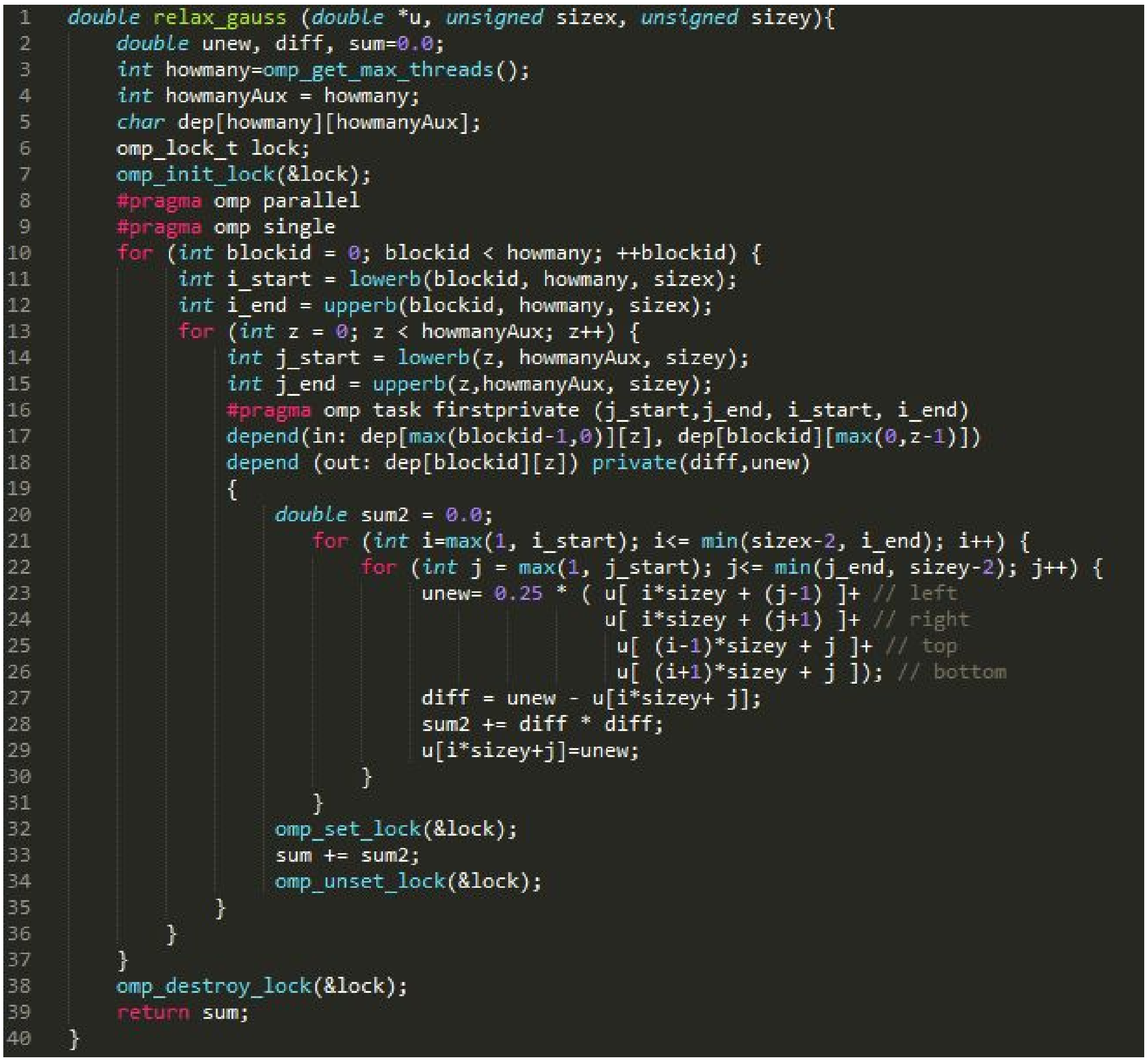
We observed a poor scalability, when we have 8 threads we appreciate an inexistent speedup. This is because a single thread will be inactive approximately half of the total computation time of the matrix, so it's easier to reach the point where the synchronization times off all the blocks exceed the computation gain of adding another thread.

3

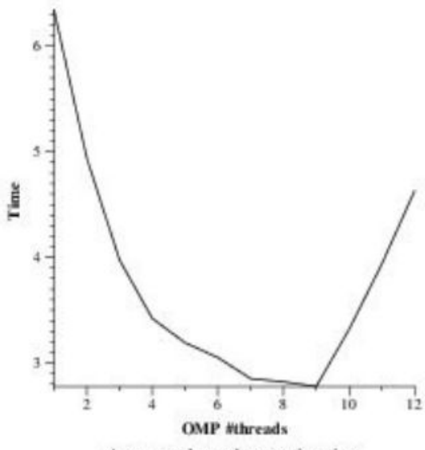
To reach the optimum ratio comp/sync we were changing the number of blocks per row to be the number of threads multiplied by a certain number B_MULT. After do this with 1 to 16 values of B_MULT we obtained that the optimum value of B_MULT was 7, for this strategy.

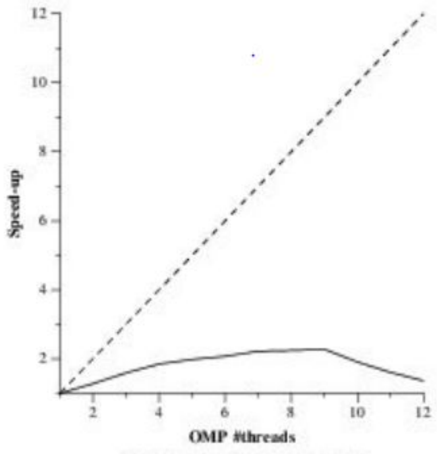
Optional

Optional 1: Implement an alternative parallel version for Gauss-Seidel using #pragma omp task and task dependences to ensure their correct execution. Compare the performance against the #pragma omp for version and reason about the better or worse scalability observed.



```
double relax_gauss (double *u, unsigned sizex, unsigned sizey){
    double unew, diff, sum=0.0;
    int howmany=omp_get_max_threads();
    int howmanyAux = howmany;
    char dep[howmany][howmanyAux];
    omp_lock_t lock;
    omp_init_lock(&lock);
    #pragma omp parallel
    #pragma omp single
    for (int blockid = 0; blockid < howmany; ++blockid) {
         int i_start = lowerb(blockid, howmany, sizex);
         int i_end = upperb(blockid, howmany, sizex);
         for (int z = 0; z < howmanyAux; z++) {
             int j_start = lowerb(z, howmanyAux, sizey);
             int j_end = upperb(z,howmanyAux, sizey);
             #pragma omp task firstprivate (j_start,j_end, i_start, i_end)
             depend(in: dep[max(blockid-1,0)][z], dep[blockid][max(0,z-1)])
             depend (out: dep[blockid][z]) private(diff,unew)
             {
                 double sum2 = 0.0;
                    for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                        for (int j = max(1, j_start); j<= min(j_end, sizey-2); 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];
                             sum2 += diff * diff;
                             u[i*sizey+j]=unew;
                        }
                    }
                 omp_set_lock(&lock);
                 sum += sum2;
                 omp_unset_lock(&lock);
             }
         }
    }
    omp_destroy_lock(&lock);
    return sum;
}
```


Average elapsed execution time


Speed-up wrt sequential time

Comparing this plots with the GaussSeidel we apreciated that using a task strategy is worse than using the omp for strategy. This is basically due to the additional time that is spent managing tasks that makes our performance slower at any number of threads.

Conclusions

After do this session the main conclusion we can get is that block data decomposition has a lot of problems. When we wave a program with a lot of dependences block data it’s difficult to implement. There are better strategies to use when a program has a lot of dependences.