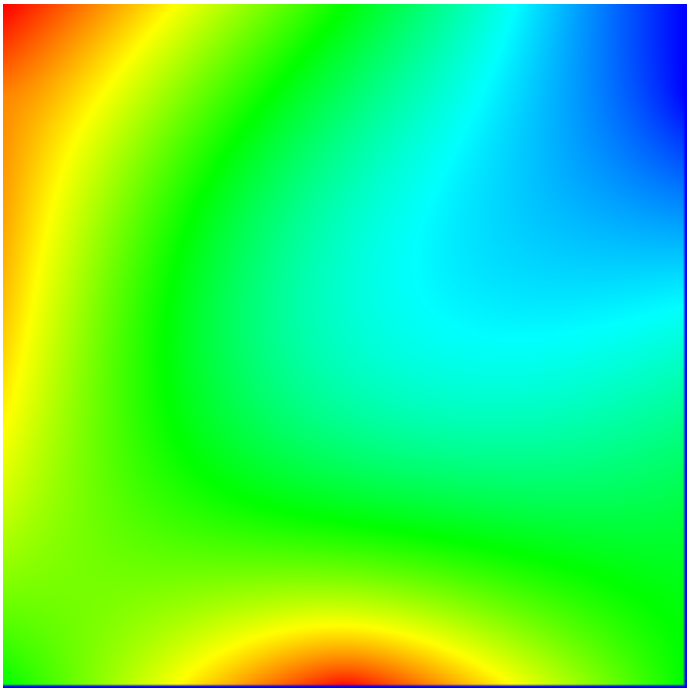
PAR Laboratory Assignment

Lab 5: Geometric (data) decomposition:
heat diffusion equation



Aleix Boné
Alex Herrero
par2109

2020-06-05

Contents

1 Introduction

In this session we are going to work on the parallelization of a sequential code that simulates heat diffusion in a solid body using two different solvers for the heat equation. The program reads a configuration file that specifies the maximum number of steps, the size of the body and the solver to use which can be either *Jacobi* or *Gauss-Seidel*.

The two solvers have different numerical properties and behave differently when parallelized. With the *Jacobi* method, the values of the ith iteration remain unchanged until the next iteration has been calculated, with Gauss-Seidel the results are used immediately. Moreover, Gauss-Seidel converges faster than Jacobi. [1].

When we tried the sequential version of the program with the input in `test.dat` we obtained different results as shown in figure 1 and *Gauss-Seidel* was more than twice as fast than *Jacobi*:

```
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: 2.394
Flops and Flops per second: (8.806 GFlop => 3679.06 MFlop/s)
Convergence to residual=0.000050: 12409 iterations

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: 4.819
Flops and Flops per second: (11.182 GFlop => 2320.50 MFlop/s)
Convergence to residual=0.000050: 15756 iterations
```

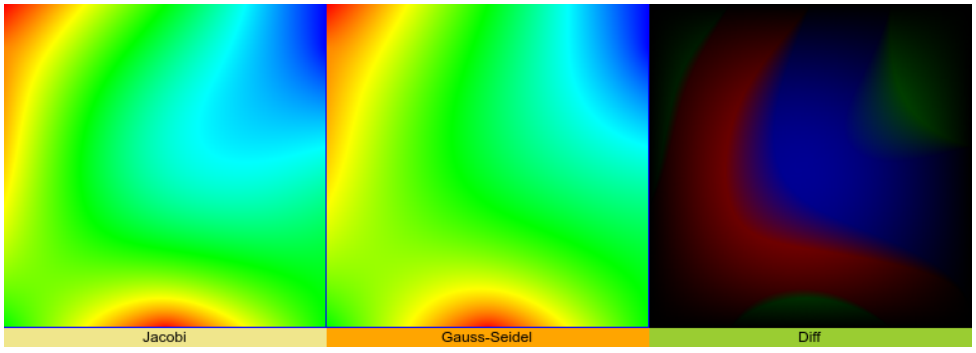


Figure 1: Comparison between heat map generated by Jacobi and Gauss-Seidel

[1] https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf

2 Analysis of task granularities and dependences with *Tareador*

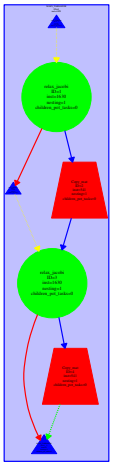


Figure 2: Coarse dependency graph of Jacobi

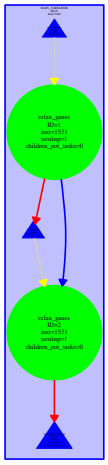


Figure 3: Coarse dependency graph of Gauss-Seidel

Figures 2 and 3 show the dependency graph obtained with *tareador* for Jacobi and Gauss-Seidel respectively (with very coarse granularity).

The first thing we notice is that there are two executions of `relax_jacobi` and `relax_gauss`. Taking a look at the code shown in listing 1 we can see that the program applies the relax function on the data until the residual obtained is small enough or the maximum number of iterations is reached. Each iteration depends on the results of the previous one, so they cannot be parallelized. In the case ran by *tareador* this is barely noticeable since the data used is very small and the maximum number of iterations is set 2 with a resolution of 4. However for `test.dat` the maximum number of iterations is set to 2500 with a resolution of 254.

In the case of Jacobi there is also the function `copy_mat` that is used as an auxiliary matrix, since Jacobi does not modify `u` in place and Gauss-Seidel does. In fact, this is the main difference between the two algorithms, Jacobi uses the matrix from the previous iteration of the method while Gauss-Seidel uses modifies it as it traverses the matrix.

Listing 1: heat.c

```
83      while (1) {
84          switch (param.algorithm) {
85              case 0:                 // JACOBI
86                  residual = relax_jacobi(param.u, param.uhelp, np, np);
87                  // Copy uhelp into u
88                  copy_mat(param.uhelp, param.u, np, np);
89                  break;
90              case 1:                 // GAUSS
91                  residual = relax_gauss(param.u, np, np);
92                  break;
93          }
94
95          iter++;
96
97          // solution good enough ?
98          if (residual < 0.00005)
99              break;
100
101         // max. iteration reached ? (no limit with maxiter=0)
102         if (param.maxiter > 0 && iter >= param.maxiter)
103             break;
104     }
```

In order to further analysis the potential parallelization of both both methods, we have to look into the inner workings of each step of the relax functions. To achieve this, we added finer *tareador* task definitions, defining tasks for each iteration of the loops as shown in Listing 2:

Listing 2: solver-tareador.c

```
    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("i_jacobi");
                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("i_jacobi");
            }
        }
    }
```

```
    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("i_gauss");
                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("i_gauss");
            }
        }
    }
```

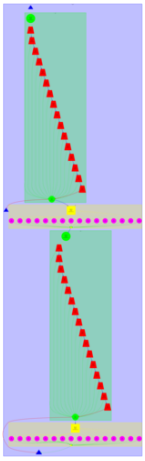


Figure 4: Fine dependency graph Jacobi

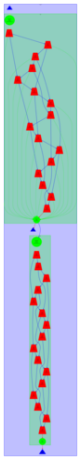


Figure 5: Fine dependency graph Gauss-Seidel

Figure 4 and 5 show the task dependency graph obtained with *tareador*. The results are disappointing, since there are dependencies making all the code sequential, nevertheless, if we look at the variables causing those dependencies (shown in figures 6 and 7) we see that the dependence in both cases is caused by the `sum` variable which in this case is only used once and can be reduced or protected by an atomic clause. This means that we can still parallelize most of the work.

However in the case of Gauss-Seidel there is another dependency corresponding to some memory positions in the heap, which by analyzing the code we can determine that this data on the heap is the matrix `u`. Since Gauss-Seidel modifies the data in `u`, this creates a dependency between iterations of the method. More specifically they have a dependency on the left cell and the top cell of the matrix.

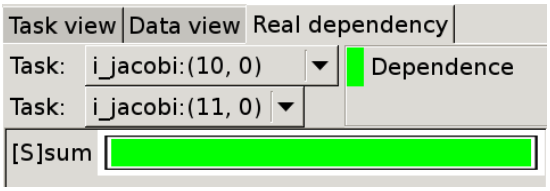


Figure 6: Task dependence Jacobi

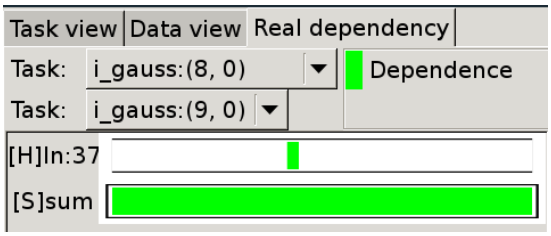


Figure 7: Task dependence Gauss-Seidel

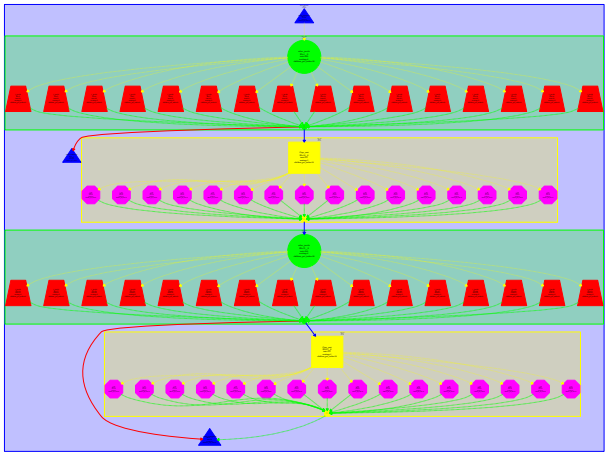


Figure 8: Fine dependency graph Jacobi (no sum)

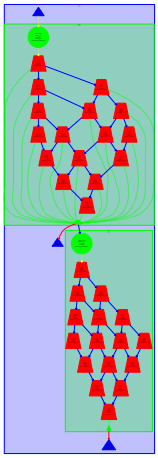


Figure 9: Fine dependency graph Gauss-Seidel (no sum)

In figure 8 and 9 we show the data decomposition obtained with *tareador* ignoring the `sum` variable dependency. We can see that if we eliminate the sum dependency in Jacobi the iterations of the loop inside `relax_jacobi` are embarrassingly parallel. With Gauss-Seidel, we can clearly see the top-left dependency of the iterations caused by the modification of the matrix `u`.

3 OpenMP parallelization and execution analysis: *Jacobi*

Listing 3: solver.c

```
17 double relax_jacobi(double *u, double *utmp, unsigned sizex,
18         unsigned sizey)
19 {
20     double diff, sum = 0.0;
21 
22     int howmany = 4;
23     for (int blockid = 0; blockid < howmany; ++blockid) {
24         int i_start = lowerb(blockid, howmany, sizex);
25         int i_end = upperb(blockid, howmany, sizex);
26         for (int i = max(1, i_start); i <= min(sizex - 2, i_end); i++) {
27             for (int j = 1; j <= sizey - 2; j++) {
28                 utmp[i * sizey + j] = 0.25 * (u[i * sizey + (j - 1)] +        // left
29                         u[i * sizey + (j + 1)] +        // right
30                         u[(i - 1) * sizey + j] +        // top
31                         u[(i + 1) * sizey + j]);        // bottom
32                 diff = utmp[i * sizey + j] - u[i * sizey + j];
33                 sum += diff * diff;
34             }
35         }
36     }
37 
38     return sum;
39 }
```

If we examine `solver.c` (listing 3 we see that there are some differences from the *tareador* version, mainly that there is an additional outer for loop that is repeated `howmany` times and determines the range of the `i` variable of the first `for`. If we look at what `lowerb` and `upperb` (listing 4) we can see that they compute the index bounds of the block number `id` if there are `n` indices divided in `p` blocks.

The code splits the matrix in `howmany` (in this case 4) blocks of rows and computes them. We decided to use this division as a basis for our OpenMP implementation.

Listing 4: heat.h `#define`

```
59 #define lowerb(id, p, n)  ( id * (n/p) + (id < (n%p) ? id : n%p) )
60 #define numElem(id, p, n) ( (n/p) + (id < (n%p)) )
61 #define upperb(id, p, n)  ( lowerb(id, p, n) + numElem(id, p, n) - 1 )
```

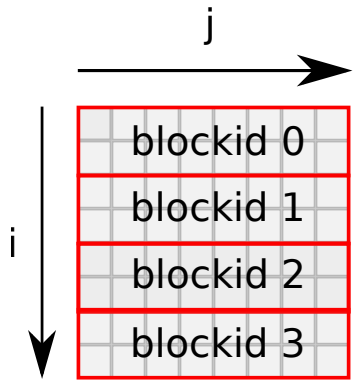


Figure 10: Jacobi row block division with 4 threads on a 8x8 matrix

Figure 10 shows an example of the block division by rows performed on a 8x8 matrix with 4 threads. In cases where the number of threads does not divide the dimension of the matrix, the final block would be smaller to make up the difference.

As we discussed on the *tareador* analysis, the only dependency between tasks is the variable `sum`, so to guarantee that there are no data races we must apply a reduction. Given that we cannot use the `pragma omp for` directives, we decided to create a variable `sum_tmp` private on each task to store the sum of each task and then perform an atomic addition to `sum` at the end of the tasks. We also added `private(diff)` and `firstprivate(blockid)` since they where declared before the task (blockid has to be firstprivate since the value is needed). Listing 5 shows the first version of our code

Listing 5: solver-omp.c Initial OpenMP version of jacobi method

```
26 double relax_jacobi(double *u, double *utmp, unsigned sizex,
27         unsigned sizey)
28 {
29     double diff, sum = 0.0;
30 
31     int howmany = 4;
32     #pragma omp parallel
33     #pragma omp single
34     for (int blockid = 0; blockid < howmany; ++blockid)
35     #pragma omp task firstprivate(blockid) private(diff)
36     {
37         int i_start = lowerb(blockid, howmany, sizex);
38         int i_end = upperb(blockid, howmany, sizex);
39         double sum_tmp = 0.0;
40         for (int i = max(1, i_start); i <= min(sizex - 2, i_end); i++) {
41             for (int j = 1; j <= sizey - 2; j++) {
42                 utmp[i * sizey + j] = 0.25 * (u[i * sizey + (j - 1)] +        // left
43                         u[i * sizey + (j + 1)] +        // right
44                         u[(i - 1) * sizey + j] +        // top
45                         u[(i + 1) * sizey + j]);        // bottom
46                 diff = utmp[i * sizey + j] - u[i * sizey + j];
47                 sum_tmp += diff * diff;
48             }
49         }
50         #pragma omp atomic
51         sum += sum_tmp;
52     }
53 
54     return sum;
55 }
```

Our initial version of the code did not modify the variable `howmany` which meant that we always divided the matrix in 4 blocks which is not optimal since if we have more than 4 threads there are threads that are not doing any work. This can be seen in the *paraver* trace we generated and that we show in figure 11 where we can see that there are always 4 concurrent tasks and 4 threads are unused.

We also noticed that there where huge sequential sections, which were caused by the matrix copy operation. In our analysis with *tareador* we saw that there were no dependencies between loop iterations so we decided to parallelize it to remove this bottleneck. Our first approach, which was quite naive, was to assign a task to every iteration of the loop, however the code run much slower than the sequential version due to the massive overhead of the task creation, we decided to use the same geometric data decomposition used in the *Jacobi* function.

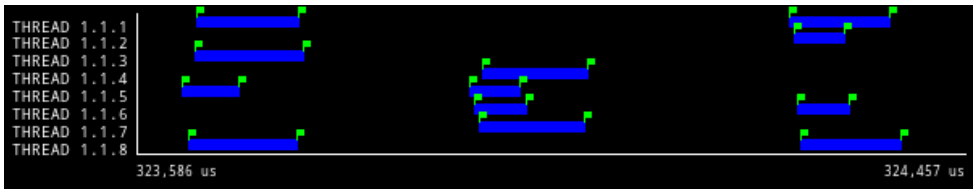


Figure 11: Parallel tasks Jacobi

Listing 6 shows the improved version of *Jacobi* with improvements on `copy_mat` and `relax_jacobi`. We removed the outer `for` since is not needed as we use the parallel region with all its threads to divide the matrix. We also applied the same division and parallelization approach to the `copy_mat` function. In figure 12 we can see the improvement over the previous version and how this time there are much more tasks and all threads are used in parallel, although some have more work than others.

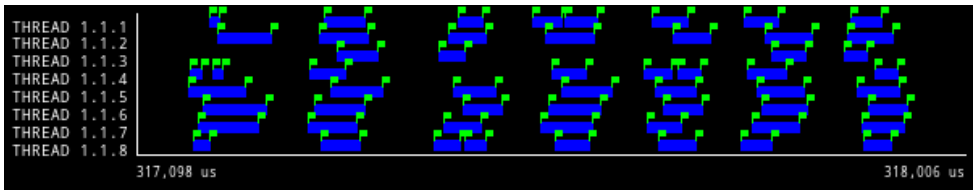


Figure 12: Parallel tasks jacobi improved

Listing 6: solver-omp.c Improved jacobi

```
8  void copy_mat(double *u, double *v, unsigned sizex, unsigned sizey)
9  {
10 #pragma omp parallel
11     {
12         int howmany = omp_get_num_threads();
13         int blockid = omp_get_thread_num();
14         int i_start = lowerb(blockid, howmany, sizex);
15         int i_end = upperb(blockid, howmany, sizex);
16         for (int i = max(1, i_start); i <= min(sizex - 2, i_end); i++)
17             for (int j = 1; j <= sizey - 2; j++)
18                 v[i * sizey + j] = u[i * sizey + j];
19     }
```

```
25 double relax_jacobi(double *u, double *utmp, unsigned sizex,
26         unsigned sizey)
27 {
28     double diff, sum = 0.0;
29
30     #pragma omp parallel private(diff)
31     {
32         int howmany = omp_get_num_threads();
33         int blockid = omp_get_thread_num();
34         int i_start = lowerb(blockid, howmany, sizex);
35         int i_end = upperb(blockid, howmany, sizex);
36         double sum_tmp = 0.0;
37         for (int i = max(1, i_start); i <= min(sizex - 2, i_end); i++) {
38             for (int j = 1; j <= sizey - 2; j++) {
39                 utmp[i * sizey + j] = 0.25 * (u[i * sizey + (j - 1)] +        // left
40                         u[i * sizey + (j + 1)] +        // right
41                         u[(i - 1) * sizey + j] +        // top
42                         u[(i + 1) * sizey + j]);        // bottom
43                 diff = utmp[i * sizey + j] - u[i * sizey + j];
44                 sum_tmp += diff * diff;
45             }
46         }
47         #pragma omp atomic
48         sum += sum_tmp;
49     }
50
51     return sum;
52 }
```

In figure 13 we can see the time and speed-up plots for different number of OMP threads of the improved version. We can see that the execution time goes from 5 seconds to less than 1 second achieving a very good speed-up till around 9 threads, this is probably caused by the task creation overhead.

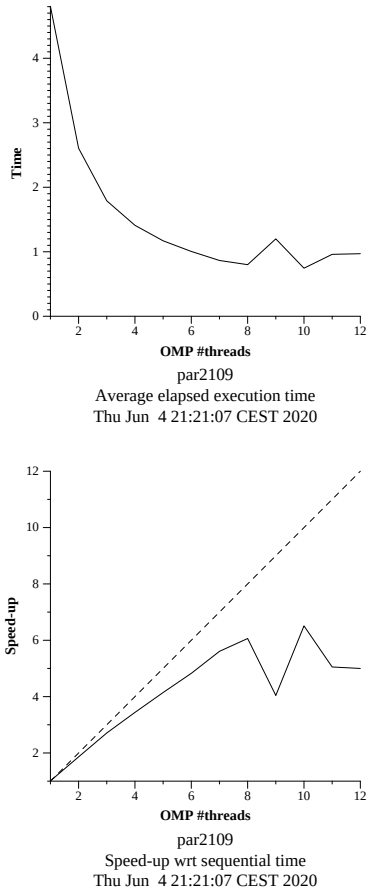


Figure 13: Strong scalability graph
Jacobi

4 OpenMP parallelization and execution analysis: *Gauss-Seidel*

A row or column based decomposition would not work in this case, since as we have seen in the *tareador* analysis, each iteration depends on the value of the previous row and column, which means that if we implemented it like that the resulting code would be sequential.

The best approach is to divide the matrix in blocks and define the dependencies between the iterations using. Figure 14 shows an example of the block decomposition and the dependencies in blue. For the task corresponding to the block i, j to begin executing, it must wait for both $i-1, j$ and $i, j-1$ to finish.

Listing 7 shows the OpenMP implementation of the Gauss-Seidel method. Notice that we create 2 outer for loops to divide the matrix both by rows and by columns. We define a reduction on the `sum` variable that created one of the data dependencies and we define 2 sinks depending on the block one row before and the other on the block on the column before.

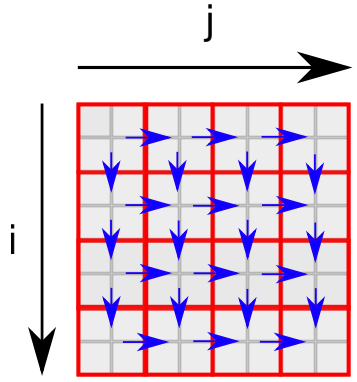


Figure 14: Gauss block decomposition with 4 threads on a 8x8 matrix

Listing 7: solver-omp.c Gauss-Seidel

```
57 double relax_gauss(double *u, unsigned sizex, unsigned sizey)
58 {
59     double unew, diff, sum = 0.0;
60 
61     #pragma omp parallel
62     {
63         int howmany = omp_get_num_threads();
64         #pragma omp for ordered(2) private(unew, diff) reduction(+:sum)
65         for (int row = 0; row < howmany; ++row) {
66             for (int col = 0; col < howmany; ++col) {
67                 int row_start = lowerb(row, howmany, sizex);
68                 int row_end = upperb(row, howmany, sizex);
69                 int col_start = lowerb(col, howmany, sizey);
70                 int col_end = upperb(col, howmany, sizey);
71                 #pragma omp ordered depend(sink: row-1, col) depend(sink: row, col-1)
72                 for (int i = max(1, row_start); i <= min(sizex - 2, row_end); i++) {
73                     for (int j = max(1, col_start); j <= min(sizey - 2, col_end);
                        ↪ j++) {
74                         unew = 0.25 * (u[i * sizey + (j - 1)] +  // left
75                                 u[i * sizey + (j + 1)] +  // right
76                                 u[(i - 1) * sizey + j] +  // top
77                                 u[(i + 1) * sizey + j]);  // bottom
78                         diff = unew - u[i * sizey + j];
79                         sum += diff * diff;
80                         u[i * sizey + j] = unew;
81                     }
82                 }
83                 #pragma omp ordered depend(source)
84             }
85         }
86     }
87 
88     return sum;
89 }
```

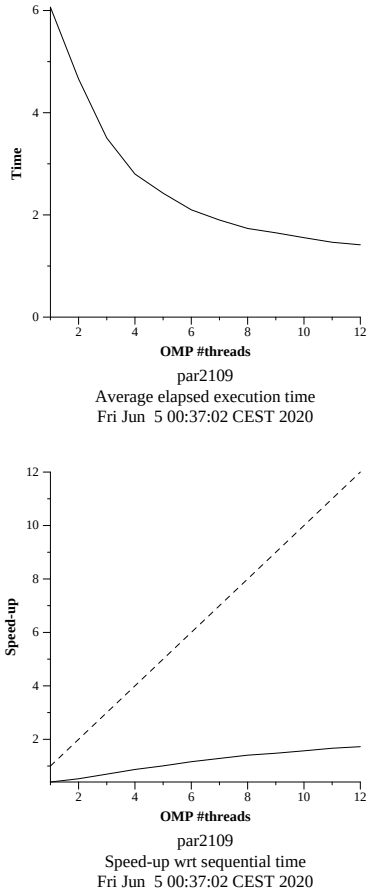


Figure 15: Strong scalability graph
Gauss

Figure 16: Thread State Gauss

As in the previous section, in figure 15 we can see the time and speed-up plots for different number of OMP threads, but in this case, of the execution of the Gauss-Seidel solver. We can observe how the execution time improves with the number of threads, and the speed-up gets better. But, as we can see, the speed-up is not even close to the ideal speed-up line and performs far worse than the *Jacobi* version. This is caused due to the synchronization overhead introduced by the data dependencies, if we have more threads and therefore we split the matrix in more blocks, we have to keep track and synchronize more tasks which generates an overhead. Therefore, we should determine the block size that gives the best ratio between synchronization and computation by choosing the proper block size.

The size of the block in our code depends on the value of `howmany`, that we decided to set to the number of threads in the parallel region: if we have 8 threads we divide the matrix in 8×8 blocks. However by setting a different value we can change the number and therefor the size of the blocks used. We tried changing the value of `howmany` form 2 to 32 [2] and executing the program with input `"test.dat"` and 8 threads. The best results where with `howmany = 16`. In figure 17 we can see a plot of the execution times based on the value of `howmany` where we can see that the inflection point happens around `howmany = 16`.

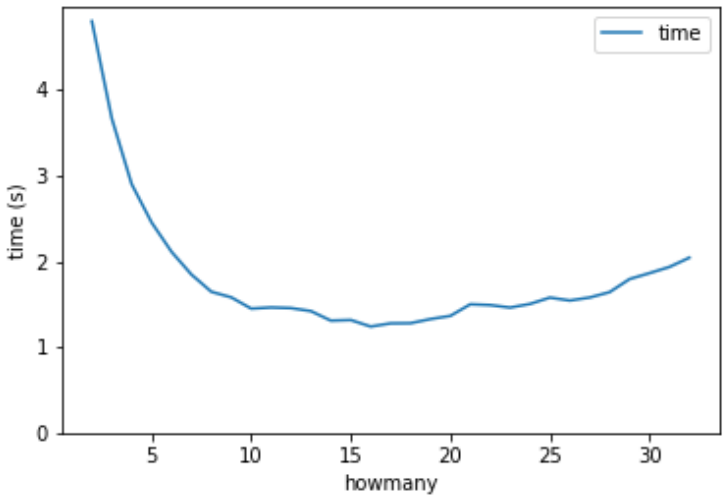


Figure 17: Execution time vs. howmany (Gauss)

5 Conclusions

Although the sequential version of *Gauss-Seidel* greatly outperformed *Jacobi* when parallelizing the code as we have seen *Jacobi* allows for a much greater parallelization due to the lack dependencies between loop iterations. This shows that although an algorithm may seem slow when compared to another in its sequential version, it may be more parallelizable than the other.

[2] We modified the program to accept howmany as a parameter