DEPARTMENT OF APPLIED MATHEMATICS
AND COMPUTER SCIENCE

02614 - High Performance Computing

2. The Poisson Problem

Authors:
Maria Garofalaki - s202378
Jorge Sintes Fernández - s202581
Klara Maria Ute Wesselkamp - s202382

Course responsible:
Bernd Dammann

January 15, 2021

Individual responsibilities

All the parts in this assignment were solved equally between the three group members.

Part I. Implementation of the Jacobi-Algorithm

Before beginning the analysis of the algorithm, we want to discuss an important subroutine as well as the algorithm itself, and what was done in order to optimize performance of the sequential algorithm.

Assign values

In the Jacobi iteration we'll need three matrices, f, u and u_old, the last of which will be used to save the values of the last iteration of the algorithm. Here we chose to assign values to both the matrix f and the matrix u as well as u_old in a single function, thus making optimal use of the three nested for-loops used to initialize the matrices with zeros (loop 1).
In order to assign the values of the walls of the cube we found that this could be done in a double-loop, by simply fixing the boundary values. They need to be assigned in both loops, as the algorithm is not going to touch the walls, they thus need to be set beforehand. (loop 2)
Finally in (loop 3) we assign the values to the matrix f. Instead of using the expensive if-function in a nested for-loop, we calculated the indices of the boundary conditions reduced the loop boundaries to those indices.

```c
void assign_ufu_old(double ***u, double ***f, double ***u_old, int N, double start_T) {
    int i, j, k;

    //loop 1
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            for (k = 0; k < N; k++){
                u[i][j][k] = start_T;
                u_old[i][j][k] = start_T;
                f[i][j][k] = 0;
            }
        }
    }

    //loop 2
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){

            u[i][0][j] = 0.;
            u[i][N-1][j] = 20.;
            u_old[i][0][j] = 0.;
            u_old[i][N-1][j] = 20.;

            u[i][j][0] = 20.;
            u[i][j][N-1] = 20.;
            u_old[i][j][0] = 20.;
            u_old[i][j][N-1] = 20.;

            u[0][i][j] = 20.;
            u[N-1][i][j] = 20.;
            u_old[0][i][j] = 20.;
            u_old[N-1][i][j] = 20.;
        }
    }

    int radxi = 0,
        radxf = (5*N)/16, // (-3/8 + 1) * N/2
        radyi = 0,
        radyf = N/4, // (1/2 + 1) * N/2
        radzi = N/6 + (N%6 > 0), // (-2/3 + 1) * N/2 truncating upwards if there's some
   remainder.
        radzf = N/2; // (0 + 1) * N/2

    printf("X: %d - %d. Y: %d - %d. Z: %d - %d\n",
            radxi, radxf, radyi, radyf, radzi, radzf);

    //loop 3
    for (i = radxi; i <= radxf; i++) {
        for (j = radyi; j <= radyf; j++) {
            for (k = radzi; k <= radzf; k++) {
                f[i][j][k] = 200;
            }
        }
    }
}
```

Listing 1: assign f and u

Comment: This function is re-used for the Gauß-Seidel Algorithm. However for this algorithm we don't need the matrix u_old. We thus made another function leaving that matrix out. Which of the two function if called is determined by the environment variable _JACOBI or _GAUSS_SEIDEL respectively.

Implementation of the Jacobi-Algorithm

The implementation of a single iteration is a pretty straightforward triple loop. The results are saved to the matrix u. However, after this we need to overwrite u_old in order to re-use it in the next iteration. This is the code-snippet that we can see here:

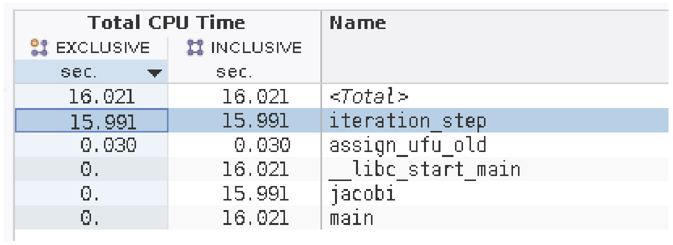
```
for (i = 0; i < N; i++){
    for(j = 0; j < N; j++){
        for(k = 0; k < N; k++){
            temp2 = u[i][j][k] - prev_u[i][j][k];
            squarednorm += temp2 * temp2;
            prev_u[i][j][k] = u[i][j][k];
        }
    }
}
```

Listing 2: Calculation of the norm inside Jacobi

It should be remarked, that the calculation of the Frobenius-Norm aka. the stopping condition is done in this loop.

Checking the function

We did a first quick check for bottlenecks and unexpectedly expensive operations using the Oracle analyzer tool. The input parameters were $N = 150$ and a tolerance of 5.

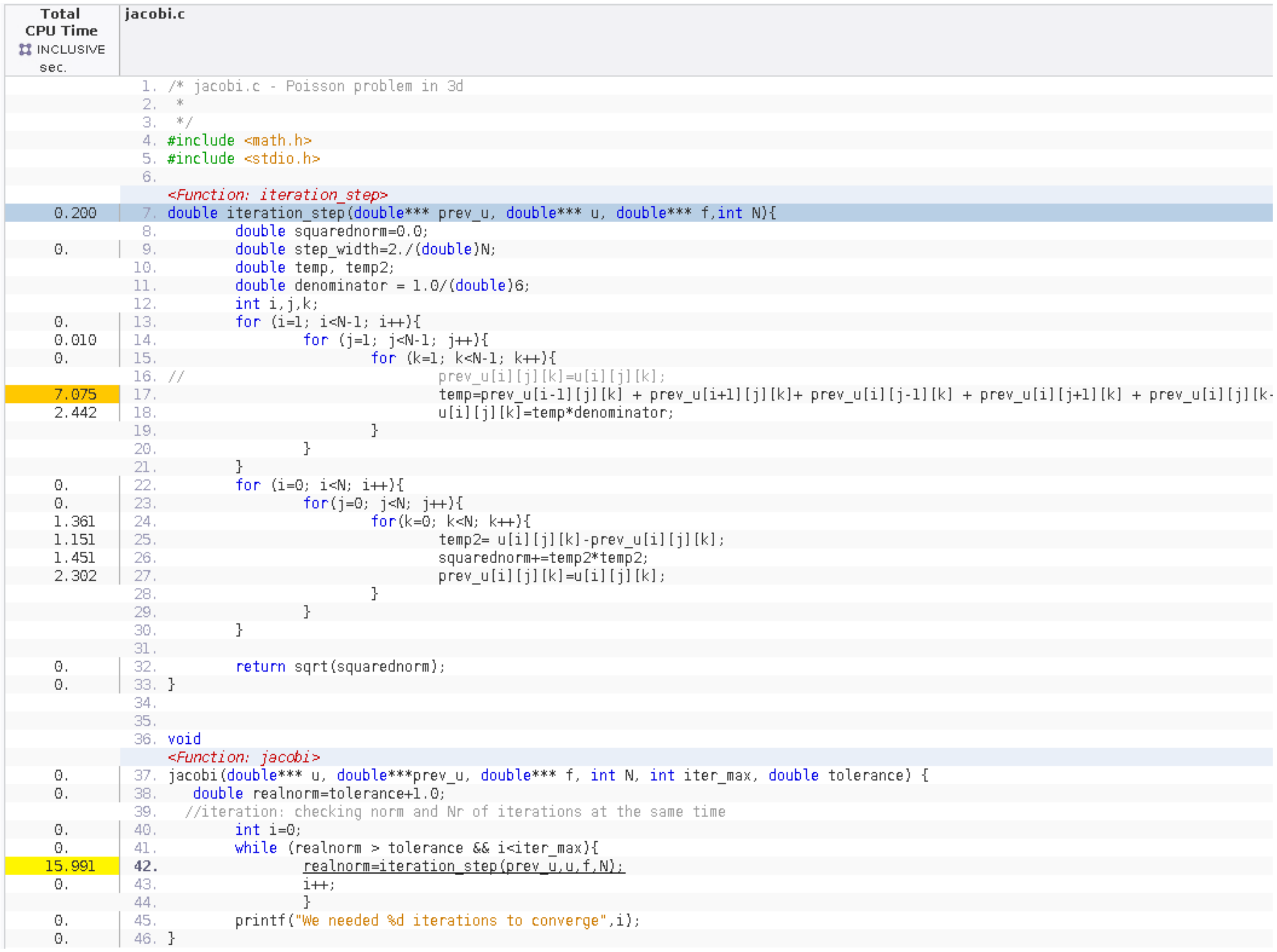


Total CPU Time EXCLUSIVE sec.	Total CPU Time INCLUSIVE sec.	Name
16.021	16.021	<Total>
15.991	15.991	iteration_step
0.030	0.030	assign_ufu_old
0.	16.021	__libc_start_main
0.	15.991	jacobi
0.	16.021	main

Figure 1: Breakdown of Time - Jacobi

The function was compiled using the -O3-compiler option. It was testes with and without this option and O3 was found to reduce the time by a factor of 8, while keeping the cache hit - cache miss- ratio at a reasonable rate of around 2.4. As was to be expected, the most expensive function is the iteration step with the triple-loop. The function that assigns values to our matrices is also somewhat relevant, although in no way compares to the weight of the iteration step. (See 2)

Taking a closer look at the iteration step, we can see that the calculation of the new matrix u takes up around 70% of the time, and the calculation of the stopping condition makes for around 30%. A possible way to improve the



```
Total
CPU Time
INCLUSIVE
sec.     jacobi.c

         /* jacobi.c - Poisson problem in 3d
          *
          */
         #include <math.h>
         #include <stdio.h>

         <Function: iteration_step>
0.200    double iteration_step(double*** prev_u, double*** u, double*** f,int N){
                 double squarednorm=0.0;
0.               double step_width=2./(double)N;
                 double temp, temp2;
                 double denominator = 1.0/(double)6;
                 int i,j,k;
0.               for (i=1; i<N-1; i++){
0.010                    for (j=1; j<N-1; j++){
0.                               for (k=1; k<N-1; k++){
         //                              prev_u[i][j][k]=u[i][j][k];
7.075                                    temp=prev_u[i-1][j][k] + prev_u[i+1][j][k]+ prev_u[i][j-1][k] + prev_u[i][j+1][k] + prev_u[i][j][k-
2.442                                    u[i][j][k]=temp*denominator;
                                 }
                         }
                 }
0.               for (i=0; i<N; i++){
0.                       for(j=0; j<N; j++){
1.361                            for(k=0; k<N; k++){
1.151                                    temp2= u[i][j][k]-prev_u[i][j][k];
1.451                                    squarednorm+=temp2*temp2;
2.302                                    prev_u[i][j][k]=u[i][j][k];
                                 }
                         }
                 }

0.               return sqrt(squarednorm);
0.       }


         void
         <Function: jacobi>
0.       jacobi(double*** u, double***prev_u, double*** f, int N, int iter_max, double tolerance) {
0.          double realnorm=tolerance+1.0;
           //iteration: checking norm and Nr of iterations at the same time
0.               int i=0;
0.               while (realnorm > tolerance && i<iter_max){
15.991                   realnorm=iteration_step(prev_u,u,f,N);
0.                       i++;
                         }
0.               printf("We needed %d iterations to converge",i);
0.       }
```

Figure 2: Detailed Running Times - Jacobi

performance here and quicken the process would be to only start calculating the condition after a certain number of iterations. This would, considering the scope of the assignment, be a question for another time.

Visualizing

We plotted the Jacobi-Function for 150 points on each axis, and visualized it with ParaView. Since the interesting part is going on inside the cube, the behaviour is best demonstrated by slicing the cube. We once slice from the cold wall to it's opposite side (that is in the y-direction), then we slice in x-direction.

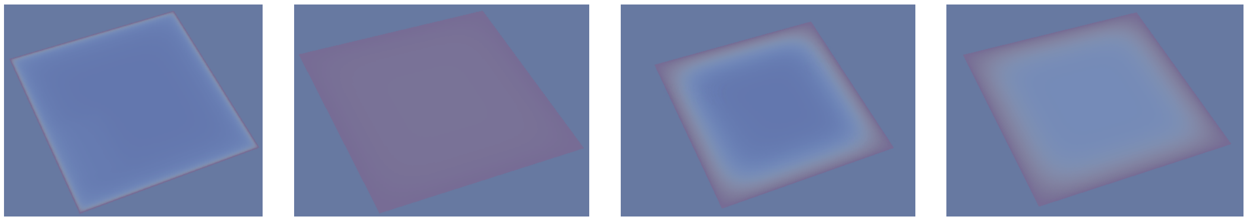


Figure 3: Y-Axis - Slicing

We can see from the pictures that this seems like a reasonable way of heat distribution in a room, the heat is disseminating from the walls, when it hit's the cold from the cold wall, there's a mingling effect. We can thus say, that it seems like a plausible solution to the problem.

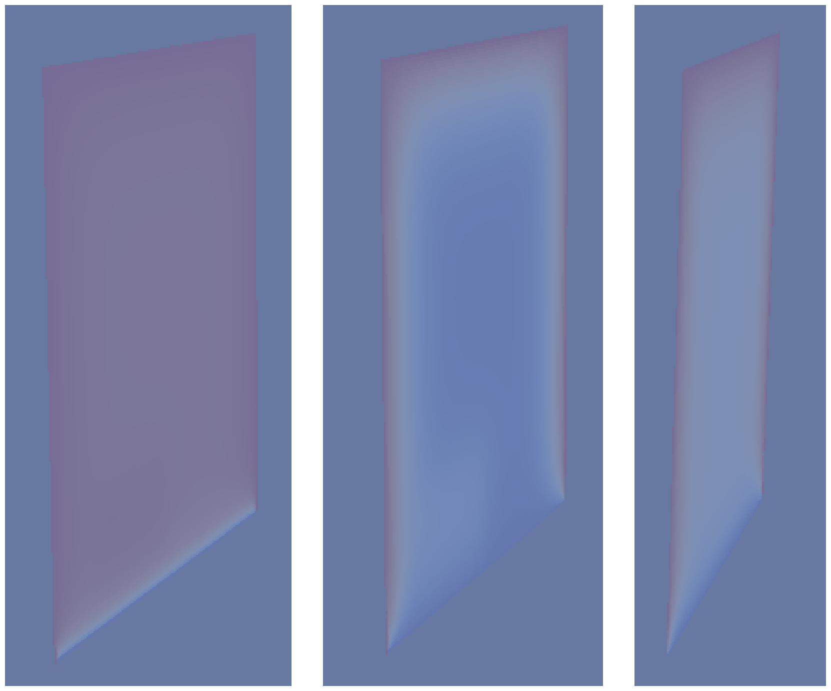


Figure 4: X-Axis - Slicing

I.4.1 Testing

All the following codes were tested in the DTU LSF Cluster. Hardware specifications can be found at the end of the document. We tested the performance and behaviour of the function using differently sized matrices. Here the exact parameters are sizes from 50 to 200 in 25-step increments. The threshold was at 1 and the number of iterations was set to 100'000 to make sure that this point would not be reached to quickly, thus making the actually needed iterations comparable.

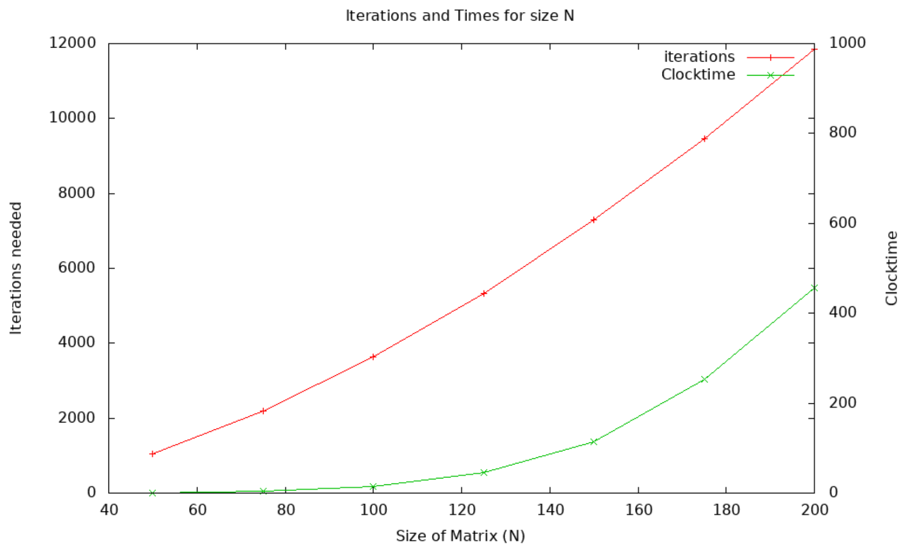


Figure 5: Sequential Jacobi - Performance

As expected, we see the exponential behaviour of the time here. Since the matrix is growing at a cubic rate, so is the time needed. At this size it it hard to say which behaviour the number of iterations demonstrate, it could be imagined linearly as well as exponentially.

Part II. Implementation of the Gauß-Seidel Algorithm:

A lot of things have already been said, that apply to Gauß-Seidel as well as Jacobi. An important difference in the way we update the matrix-values is demonstrated in the following code-snippet:

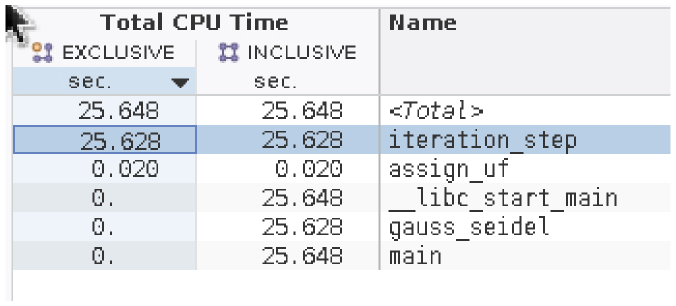
```
for (i=1; i<N-1; i++){
  for (j=1; j<N-1; j++){
    for (k=1; k<N-1; k++){
      placeholder=u[i][j][k];
      temp=u[i-1][j][k] + u[i+1][j][k]+ u[i][j-1][k] + u[i][j+1][k] + u[i][j][k-1] + u
  [i][j][k+1] + step_width*step_width*f[i][j][k];
      u[i][j][k]=temp*denominator;
      placeholder-=u[i][j][k];
      squarednorm+=placeholder*placeholder;
    }
  }
}
return sqrt(squarednorm);
```

Listing 3: iteration for gauss

We see that here u is directly overwritten, thus making possible the direct re-use of the new iteration. It is also possible to calculate the norm directly in this for loops. It is left to say whether this yield any results in terms of time.

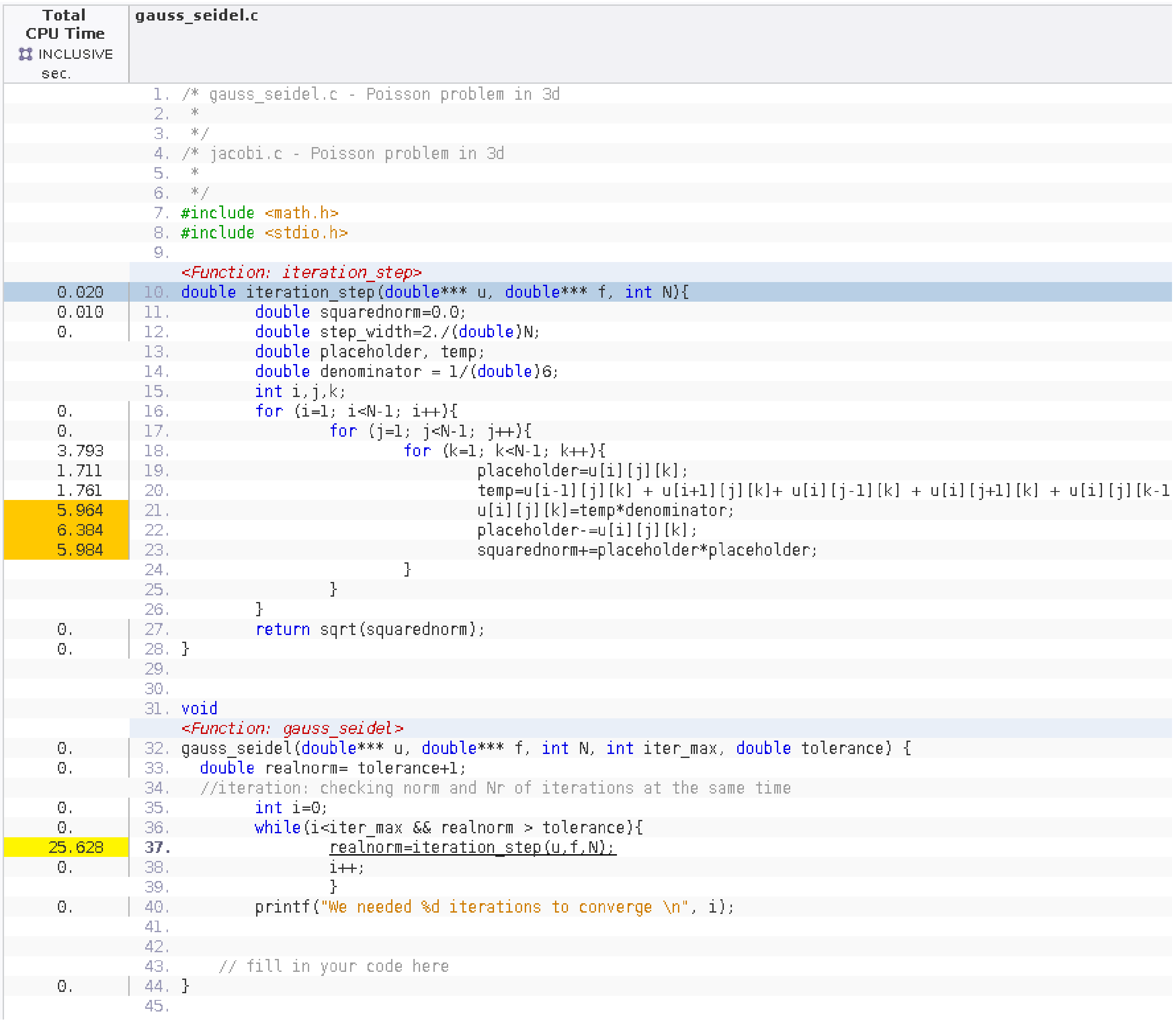
Checking the function

When running the analyzer tool for the same values as above, we get a total time of 25.65s compared to 16.02, a quite significant difference.



Total CPU Time		Name
EXCLUSIVE	INCLUSIVE	
sec.	sec.	
25.648	25.648	<Total>
25.628	25.628	iteration_step
0.020	0.020	assign_uf
0.	25.648	__libc_start_main
0.	25.628	gauss_seidel
0.	25.648	main

Figure 6: Analyzer of Gauss-Seidel

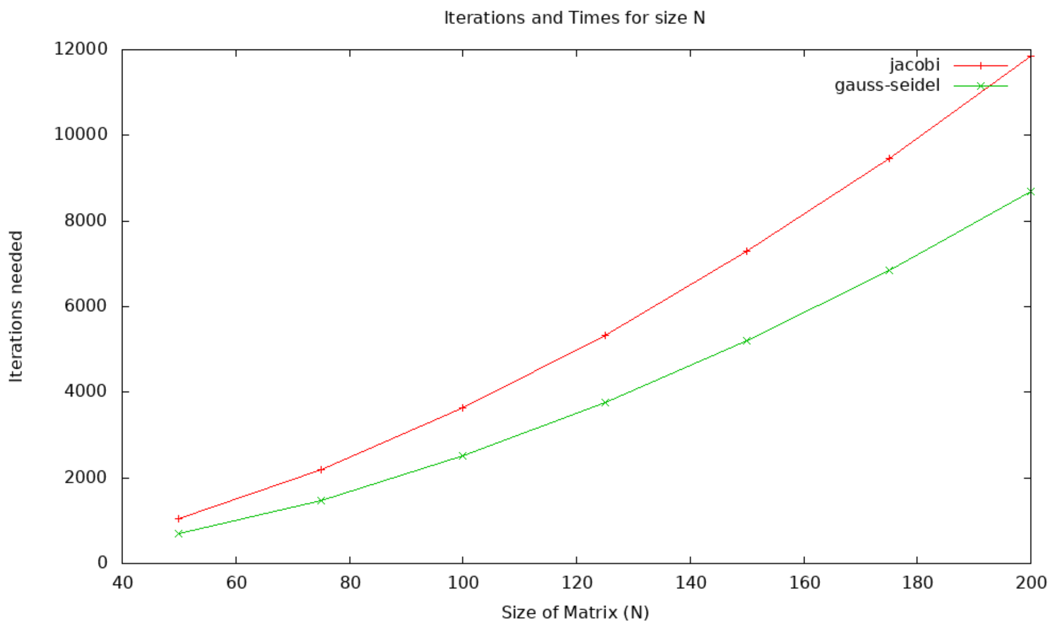


```
Total
CPU Time
INCLUSIVE     gauss_seidel.c
sec.

                /* gauss_seidel.c - Poisson problem in 3d
                 *
                 */
                /* jacobi.c - Poisson problem in 3d
                 *
                 */
                #include <math.h>
                #include <stdio.h>

                <Function: iteration_step>
      0.020     double iteration_step(double*** u, double*** f, int N){
      0.010             double squarednorm=0.0;
      0.                double step_width=2./(double)N;
                        double placeholder, temp;
                        double denominator = 1/(double)6;
                        int i,j,k;
      0.                for (i=1; i<N-1; i++){
      0.                        for (j=1; j<N-1; j++){
      3.793                             for (k=1; k<N-1; k++){
      1.711                                     placeholder=u[i][j][k];
      1.761                                     temp=u[i-1][j][k] + u[i+1][j][k]+ u[i][j-1][k] + u[i][j+1][k] + u[i][j][k-1
      5.964                                     u[i][j][k]=temp*denominator;
      6.384                                     placeholder-=u[i][j][k];
      5.984                                     squarednorm+=placeholder*placeholder;
                                        }
                                }
                        }
      0.                return sqrt(squarednorm);
      0.        }


                void
                <Function: gauss_seidel>
      0.        gauss_seidel(double*** u, double*** f, int N, int iter_max, double tolerance) {
      0.          double realnorm= tolerance+1;
                  //iteration: checking norm and Nr of iterations at the same time
      0.                int i=0;
      0.                while(i<iter_max && realnorm > tolerance){
     25.628                     realnorm=iteration_step(u,f,N);
      0.                        i++;
                                }
      0.                printf("We needed %d iterations to converge \n", i);


                   // fill in your code here
      0.        }

```

Figure 7: Analyzer of Gauss-Seidel

As we can see here the bulk of the runtime is again spent in the iteration. Surprisingly, the stopping condition is not only using a higher fraction of the total time, it is also using a higher absolute amount of time. This could prove even more problematic for increasing N, as this will also grow at a cubic rate.

Comparison of both algorithms



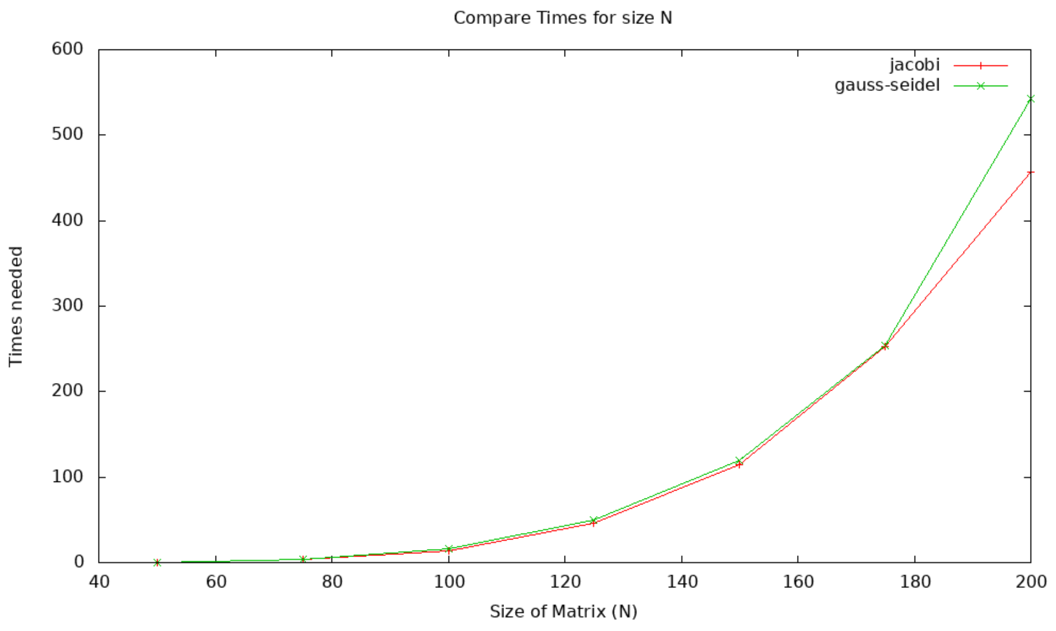


Figure 8: Comparison of Runtime and Iterations

The first graphic is a good sign for us, indicating that Jacobi does, indeed, need more iterations than Gauss. However, the Gauß-Seidel Algorithm takes more time for bigger matrices. It seems that there is possibly a memory distribution issue, which, by hazard, the Jacobi-Algorithm was able to adress better than the Gauß-Seidel Algorithm. This is underlined by the following figure 9, which takes another angle at the comparison between both algorithms.

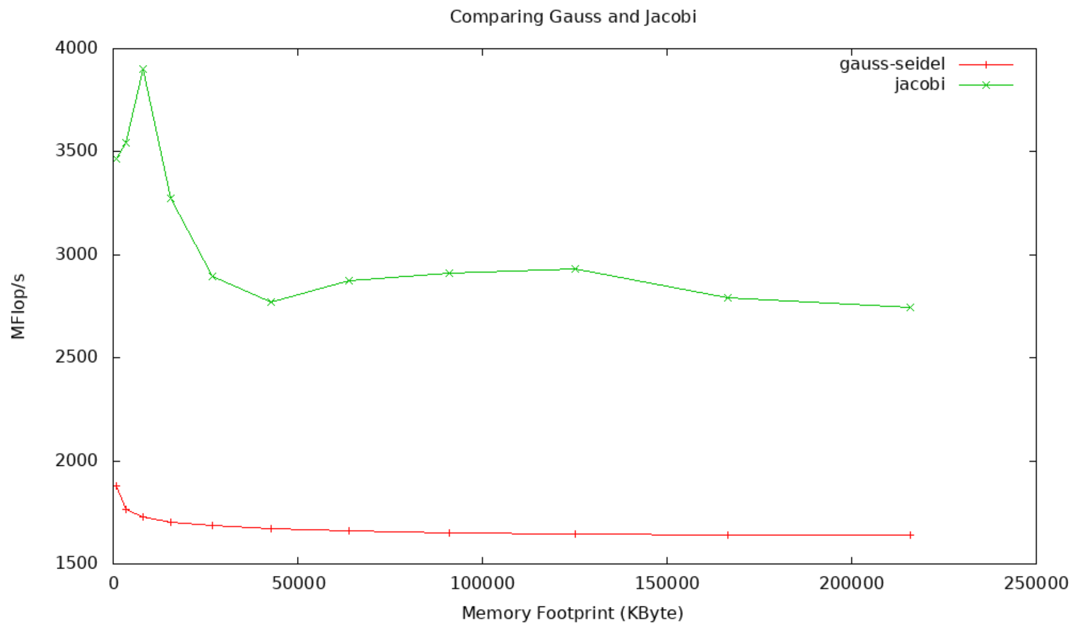


Figure 9: Mflop/s vs Memory Footprint

Part III. Parallelizing the Jacobi-Algorithm

In the following part, we will implement a parallelized version of the jacobi algorithm presented in part I. We'll start from a simpler implementation and using that one as a baseline, try to tweak it to get some scaling improvement.

However, before starting with the paralellization, we noticed there is a better way of doing the exchange between `u` and `u_old` than hard-copying it. We can profit of the pointer declaration of the matrices to do it without iterating. We changed this for the parallelization bit, so the code will be a little different from the serial version. Also, we erased the different function calls and implemented everything inside the while loop, to make the parallelization task a little bit easier.

First Version

Our first trial will be the simplest one. Running the serial version on a profiler tool we observe that, as expected, most of the execution time is spent inside the iterations, calculating the values of u and the norm for every step. For that exact reason, our first version will try to parallelyze this calculation in the simplest way possible: we'll just declare a parallel region inside the `while` loop to parallelize the `for` loops.

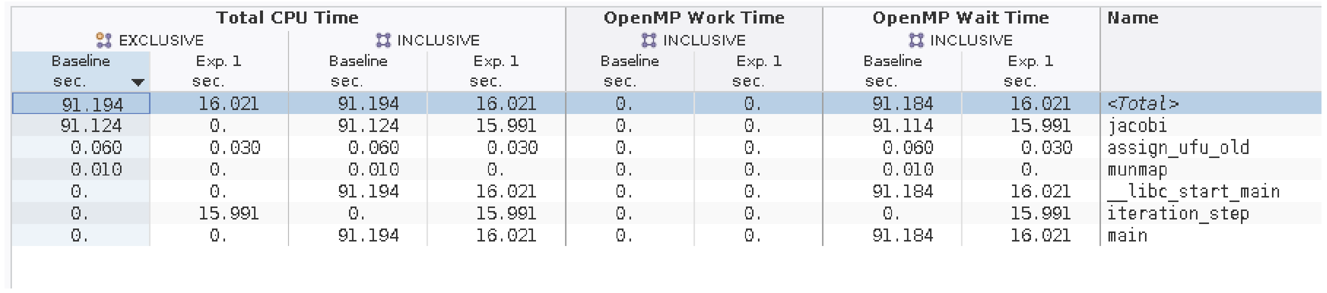
We need to be careful though: the variable `squarednorm` needs to be shared and has the potential of creating a data race problem. To avoid this, we'll use an atomic clause.

```
void jacobi(double ***u, double ***prev_u, double ***f, int N, int iter_max,
            double tolerance) {
    double step_width = 2. / (double) N;
    double temp, ***swap, realnorm = 123456789;
    double denominator = 1.0 / (double) 6;
    double temp2;
    int i, j, k, t_id = 0;
    int iter = 0;

    tolerance = tolerance * tolerance;

    while (realnorm > tolerance && iter < iter_max) {

    swap = u;
    u = prev_u;
```

```c
    prev_u = swap;
    realnorm = 0.0;
    iter++;

    # pragma omp parallel for shared(u, prev_u, f, N, iter_max, step_width,
                                    denominator, realnorm) private(i, j, k, temp, temp2)
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < N - 1; j++) {
            for (k = 1; k < N - 1; k++) {

                temp = prev_u[i-1][j][k] + prev_u[i+1][j][k] + prev_u[i][j-1][k]
                    + prev_u[i][j+1][k] + prev_u[i][j][k-1]
                    + prev_u[i][j][k+1] + step_width * step_width * f[i][j][k];
                u[i][j][k] = temp * denominator;

                temp2 = u[i][j][k] - prev_u[i][j][k];
                # pragma omp atomic
                realnorm += temp2 * temp2;
            }
        }
    }
    printf("Tolerance: %.2f/%.2f. Iterations: %d/%d\n", sqrt(realnorm), sqrt(tolerance),
     iter, iter_max);
}
```

Listing 4: Parallel Jacobi V1

As was to be expected, the succesive calling of parallel region inside the `while` produces an overhead which, together with the waiting time the atomic clause introduces, makes us don't expect this implementation to yield any significant improvement. We'll compare this first parallel implementation (baseline) and the sequential version with the analyzer tool, running them for a matrix of size 150, tolerance 5 and 1000 iterations. The results are shown in figure 10.



Total CPU Time				OpenMP Work Time		OpenMP Wait Time		Name
EXCLUSIVE		INCLUSIVE		INCLUSIVE		INCLUSIVE		
Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	Baseline sec.	Exp. 1 sec.	
91.194	16.021	91.194	16.021	0.	0.	91.184	16.021	<Total>
91.124	0.	91.124	15.991	0.	0.	91.114	15.991	jacobi
0.060	0.030	0.060	0.030	0.	0.	0.060	0.030	assign_ufu_old
0.010	0.	0.010	0.	0.	0.	0.010	0.	munmap
0.	0.	91.194	16.021	0.	0.	91.184	16.021	__libc_start_main
0.	15.991	0.	15.991	0.	0.	0.	15.991	iteration_step
0.	0.	91.194	16.021	0.	0.	91.184	16.021	main

Figure 10: Comparison between Parallel Jacobi V1 (baseline) and the serial implementation

As we can see here, our first attempt it's way worse than the sequential version, so we need immediate improvement.

Version 2

For the second version, we'll try to avoid the overhead caused by the declaration of parallel inside the `while` loop. We were having a lot of trouble doing this as some threads were entering the while loop and modifying the value of `realnorm` before other threads entered the loop so the program couldn't run properly. Also, we needed a way to stop all threads to modify the iteration variable over the while loop. We finally managed to solve these problems introducing a barrier and a single block in our code.

```c
# pragma omp parallel shared(u, prev_u, swap, f, N, iter_max, tolerance, step_width,
                              denominator, iter, realnorm)
                              private(i, j, k, temp, temp2, t_id)
{
while (realnorm > tolerance && iter < iter_max) {
    # pragma omp barrier
    # pragma omp single
    {
        swap = u;
        u = prev_u;
        prev_u = swap;
        realnorm = 0.0;
        iter++;
    }
    # pragma omp for
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < N - 1; j++) {
            for (k = 1; k < N - 1; k++) {
                temp = prev_u[i-1][j][k] + prev_u[i+1][j][k] + prev_u[i][j-1][k]
                     + prev_u[i][j+1][k] + prev_u[i][j][k-1] + prev_u[i][j][k+1]
                     + step_width * step_width * f[i][j][k];
                u[i][j][k] = temp * denominator;

                temp2 = u[i][j][k] - prev_u[i][j][k];
                # pragma omp atomic
                realnorm += temp2 * temp2;
            }
        }
    }
    // printf("2. %f, %d. Thread no.: %d\n", realnorm, iter, t_id);
}
} // end of parallel region
```

Listing 5: Parallel Jacobi V2

The function behaves now a little better than the first version. Unfortunately, as shown in figure 11, this implementation still isn't good enough to justify its use over the serial version, the algorithm is actually slower!

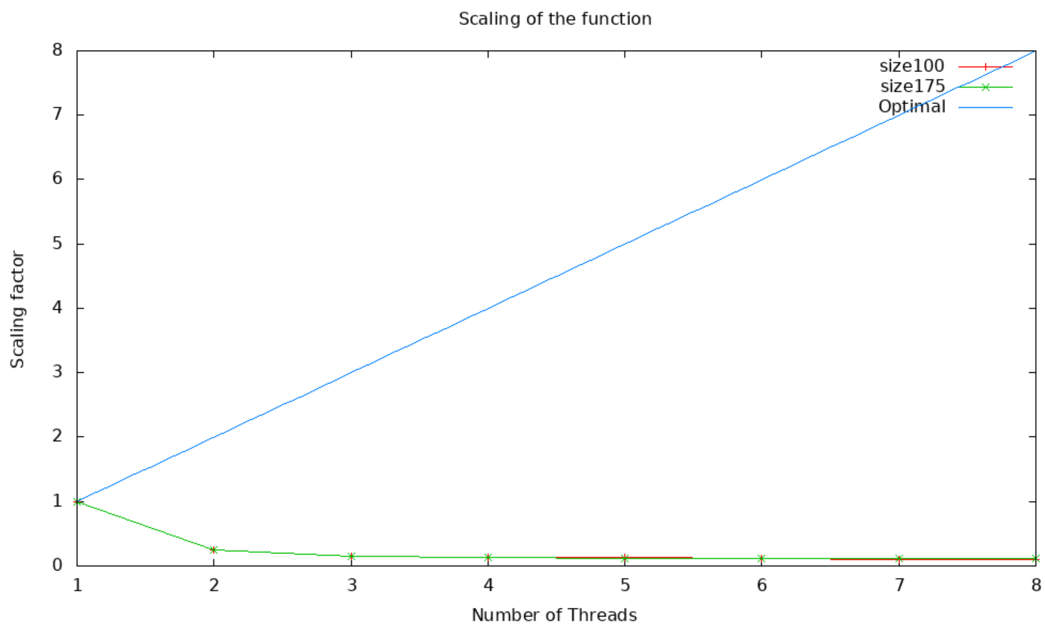


Figure 11: Scaling of Parallel Jacobi V2

Version 3

To improve last algorithm, we'll get rid of the atomic clause. Instead, we can use OpenMP reduction over the value of `realnorm` to prevent the data race. The only change will be when calling the nested `for` loop.

```
# pragma omp for reduction(+:realnorm)
for (i = 1; i < N - 1; i++) {
    for (j = 1; j < N - 1; j++) {
        for (k = 1; k < N - 1; k++) {
            temp = prev_u[i-1][j][k] + prev_u[i+1][j][k] + prev_u[i][j-1][k]
                 + prev_u[i][j+1][k] + prev_u[i][j][k-1] + prev_u[i][j][k+1]
                 + step_width * step_width * f[i][j][k];
            u[i][j][k] = temp * denominator;

            temp2 = u[i][j][k] - prev_u[i][j][k];
            realnorm += temp2 * temp2;
        }
    }
}
```

Listing 6: Parallel Jacobi V3

This subtle change makes a huge difference in the scaling of the Jacobi algorithm, as shown in figure 12. However, we can still try to push it a little bit higher.

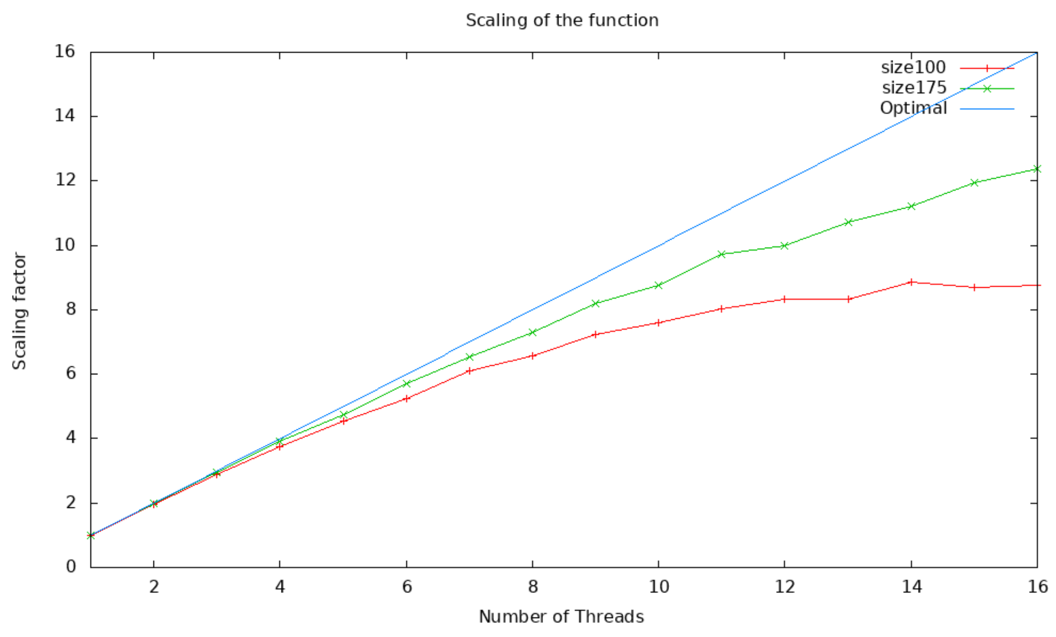


Figure 12: Scaling of Parallel Jacobi V3

Version 4

For this last trial, we will parallelize the initialization of the matrices `u`, `u_old` and `f` on the function `assign_ufu_old` with a simple parallel section with `# pragma omp for` on every outer `for` loop. We also tried to implement the `collapse` clause on every `for` loop of our functions, but it only made it worse. We examine the scaling of this function for Matrices of two different size. The result can be seen in figure 13.

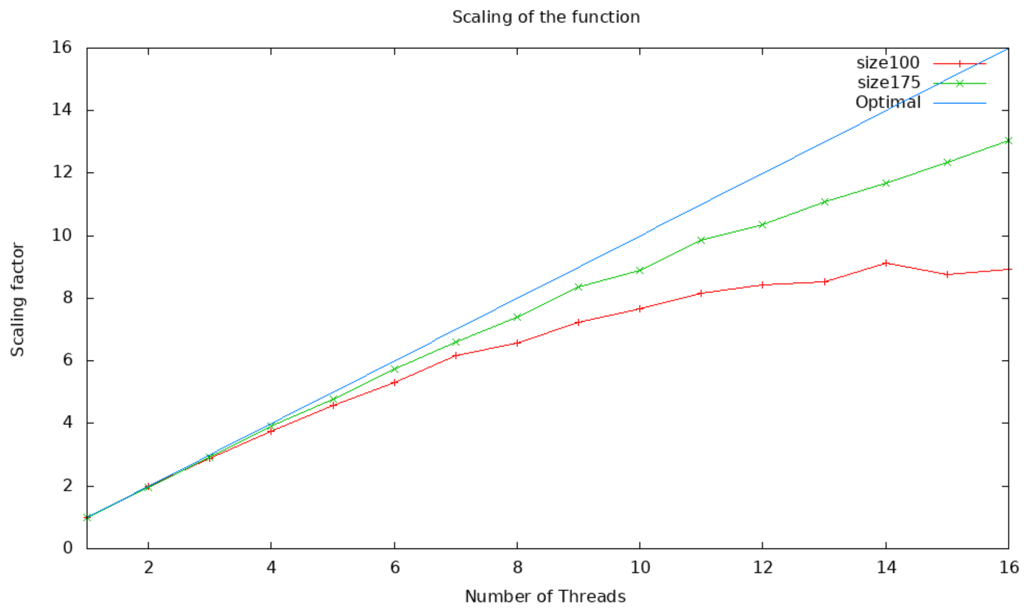


Figure 13: Scaling of Parallel Jacobi V4

We can, from the hereby obtained values calculate the percentage of overhead and the parallel percentage by using the following formula:

$$f = \frac{1 - \frac{T(P)}{T(1)}}{1 - \frac{1}{P}}.$$

We can obtain a theoretical parallel percentage from the results of a matrix of size 100. We get an approximate value of around 97%. In figure 14, we can see the experimental results for 100 along with the curve of 97% according to Amdahl's law. They look pretty similar for the initial number of threads and it's not until 12 threads when they start to differ.

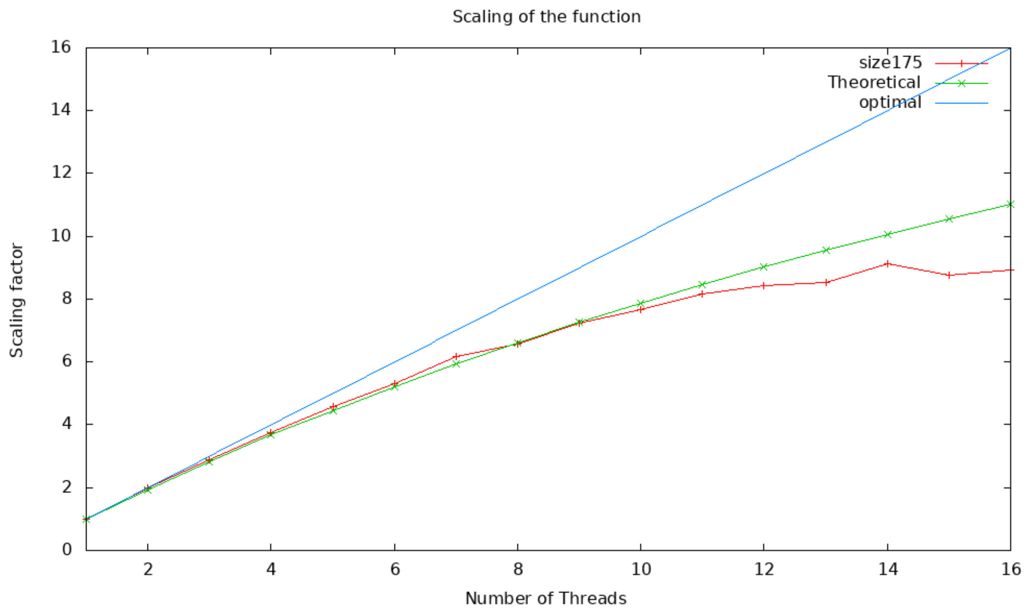


Figure 14: Calculated Parallel Fraction

Comparison

When comparing the data for the third and the final version we find that the differences are actually very small (hence no plot was used for visualization), but definitely present:

Size	50	100	150	200	250	300	350
Version 3	0.047s	0.400s	2.175s	7.361s	19.034s	42.797s	1m23.131s
Version 4	0.043s	0m0.396s	2.143s	7.269s	18.876s	42.507s	1m22.644s

Table 1: Comparison of Parallel Jacobi V3 and V4

Overall, our two final implementations behave really well. Our scaling analysis results correspond to a parallelization percentage of more than 95% according to Amdahl's law. The difference between these two may not seem significant. However, when implementing the parallelization over the initialization of the matrices, we open the door to play with different NUMA configurations. This way, we might get even better results but unfortunately we lack the time needed to implement these changes over our algorithm.

Part IV. OpenMP Gauss-Seidel

The paralellisation of Gauss-Seidel algorithm is trickier than Jacobi algorithm. That is due to the way that we update the matrix-values. As we mentioned already in Part II the values at each iteration are dependent on the order of the original equations and that make the paralisation of this algorithm very hard to achieve.

However we overpass that problem by using `#pragma omp ordered` as we can see in the code below. That code is our first working version which is the simplest one and the less efficient.

Version 1

```
double iteration_step(double ***u, double ***f, int N) {
    double squarednorm = 0.0;
    double step_width = 2. / (double) N;
    double placeholder, temp;
    double denominator = 1 / (double) 6;
    int i, j, k;

    # pragma omp parallel for ordered(2) private(i, j, k, placeholder)
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < N - 1; j++) {

            # pragma omp ordered depend(sink: i - 1, j) depend(sink: i, j - 1)
            for (k = 1; k < N - 1; k++) {
                placeholder = u[i][j][k];

                temp = u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k]
                     + u[i][j+1][k] + u[i][j][k-1] + u[i][j][k+1]
                     + step_width * step_width * f[i][j][k];

                u[i][j][k] = temp * denominator;
                placeholder -= u[i][j][k];
                # pragma omp atomic
                squarednorm += placeholder * placeholder;
            }
            # pragma omp ordered depend(source)
        }
    }
    return sqrt(squarednorm);
}
```

Listing 7: Parallel Gauss-Siedel V1

More specifically the `ordered` clause force that certain events within the loop happen in a predicted order. That way, `ordered` sequentializes and orders the execution of ordered regions while allowing code outside the region to run in parallel. In that way with the depend clauses `i,j-1` and `i-1,j` we specify the order in which the threads in the team implement the update of the matrix-values.

Moreover we used the `# pragma omp atomic` in order to avoid data races as the value of the variable `squarednorm` is changed in each loop and it is possible to different threads to access it's value at the same time. However when we execute that code and tested with different sizes of the matrices and different thread numbers and compare it with the serial version of the algorithm we realise that in some cases the run time of that parallel version was even slower than the serial one.

After some searching we realized that the `atomic` variable comes with a price, and this price is synchronization. In order to ensure that there is no race conditions, threads must synchronize which effectively means that you lose parallelism, so threads are serialized. In our second version we'll address this problem.

Version 2

```c
double iteration_step(double ***u, double ***f, int N) {
    double squarednorm = 0.0;
    double step_width = 2. / (double) N;
    double placeholder, temp;
    double denominator = 1 / (double) 6;
    int i, j, k;

    # pragma omp parallel
    for ordered(2) private(i, j, k, placeholder) reduction(+: squarednorm)
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < N - 1; j++) {

            # pragma omp ordered depend(sink: i - 1, j) depend(sink: i, j - 1)
            for (k = 1; k < N - 1; k++) {
                placeholder = u[i][j][k];

                temp = u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k]
                     + u[i][j+1][k] + u[i][j][k-1] + u[i][j][k+1]
                     + step_width * step_width * f[i][j][k];

                u[i][j][k] = temp * denominator;
                placeholder -= u[i][j][k];
                // # pragma omp atomic
                squarednorm += placeholder * placeholder;
            }
            # pragma omp ordered depend(source)
        }
    }
    return sqrt(squarednorm);
}
```

Listing 8: Parallel Gauss-Siedel V2

In the code above(our second implementation) the only change we have made from our original parallel version is that we replaced the `atomic` variable with the `reduction(+:squarednorm)` clause. Reduction is a general operation that can be carried out in parallel in comparison with the `atomic` clause. After testing that improved version of our code we see a big difference compared to the first own so the replacement of the atomic clause with the reduction actually worked. However we still weren't satisfied with the performance of our code. For that reason we made our third and final version which is presented in the code below:

Version 3

```c
double iteration_step(double ***u, double ***f, int N, double *squarednorm) {
    static double loopnorm;
    double step_width = 2. / (double) N;
    double placeholder, temp;
    double denominator = 1 / (double) 6;
    int i, j, k;

    loopnorm = 0.0;

    # pragma omp for ordered(2) reduction(+: loopnorm)
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < N - 1; j++) {

            # pragma omp ordered depend(sink: i - 1, j) depend(sink: i, j - 1)
            for (k = 1; k < N - 1; k++) {
                placeholder = u[i][j][k];

                temp = u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k]
                     + u[i][j+1][k] + u[i][j][k-1] + u[i][j][k+1]
                     + step_width * step_width * f[i][j][k];

                u[i][j][k] = temp * denominator;
                placeholder -= u[i][j][k];
                loopnorm += placeholder * placeholder;
            }
            # pragma omp ordered depend(source)
        }
    }

    *squarednorm = sqrt(loopnorm);

    void gauss_seidel(double ***u, double ***f, int N, int iter_max, double tolerance) {
    double squarednorm = tolerance + 1;

    int i = 0;
    # pragma omp parallel shared(squarednorm)
    {
        while (i < iter_max && squarednorm > tolerance) {
            # pragma omp barrier
            # pragma omp single {
                squarednorm = 0.0;
                i++;
            }

            iteration_step(u, f, N, & squarednorm);
        }
    } // end of parallel region
    printf("Tolerance: %.2f/%.2f. Iterations: %d/%d\n", (squarednorm), (tolerance), i,
   iter_max);
}
```

Listing 9: Parallel Gauss-Seidel V3

As we can see the big change we have made in this version which improved the efficiency of our code it that we put the `# pragma omp parallel` outside of the while loop (before the calling of the function `gauss_seidel`). In that way, as we have already mentioned in the part III, we reduce the overhead of making new threads all the time. However by doing that change we had to take care of the correct calculation of the variable `squarednorm`. That calculation is happening inside the function `gauss_seidel` and we pass it as an argument when we call the function. However because that variable is shared by all thread and and we want to get the correct value of calculated norm after the execution of the function we pass as an argument to the function not the variable (because that would it private) but a pointer to that variable so as all threads can share that variable.

Finally we put the value of the calculated norm in the address which is shown by the double pointer squarednorm which we pass as an argument to our function and we declare the variable loopnorm in the beginning of the function as static so that it's value can be shared by all the threads.

Moreover, in order to check for better performance of our parallelized Gauss-Seidel algorithm we used the environment variable `OMP_SCHEDULE` to control the schedule kind and chunk size our loops. However, when we used `OMP_SCHEDULE` =dynamic we experienced an odd behavior for a large number of threads. More specifically for threads number less than eight our algorithm had the same output as the serial one. When we set the number of thread to a number bigger than eight the output of our algorithm changed and also the performance remained stable so to prevent this unstable behaviour we changed the schedule to 'static,1'.

Comparison

For having a better overview and understanding of how our different versions of our algorithm work with different numbers of threads we have made 3 different plots (one for each version) which present how they scale with respect to the number of threads.

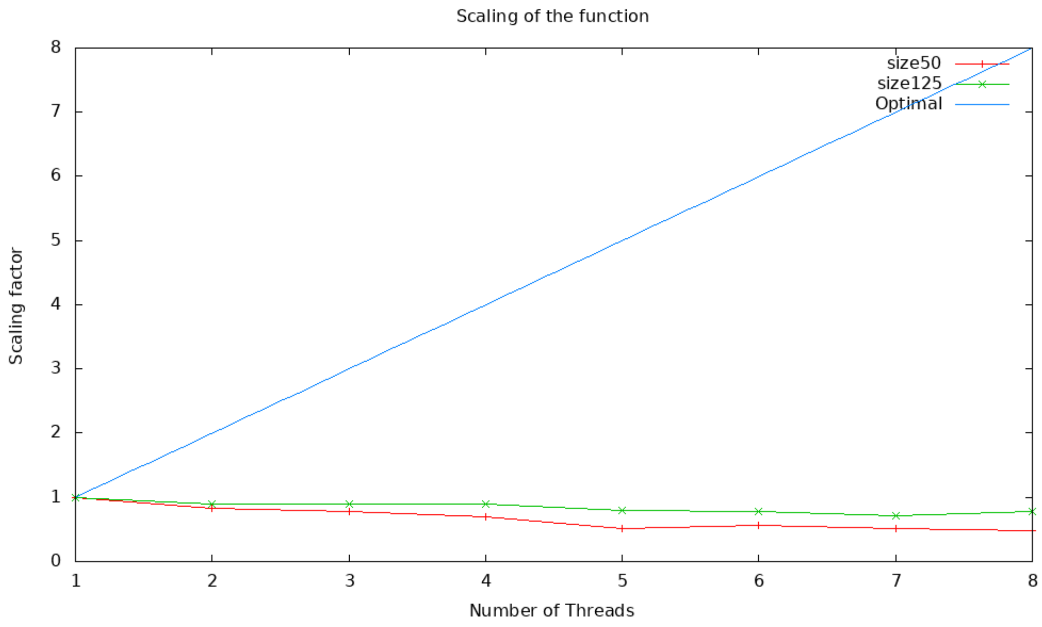


Figure 15: Scaling of Parallel Gauss-Seidel V1

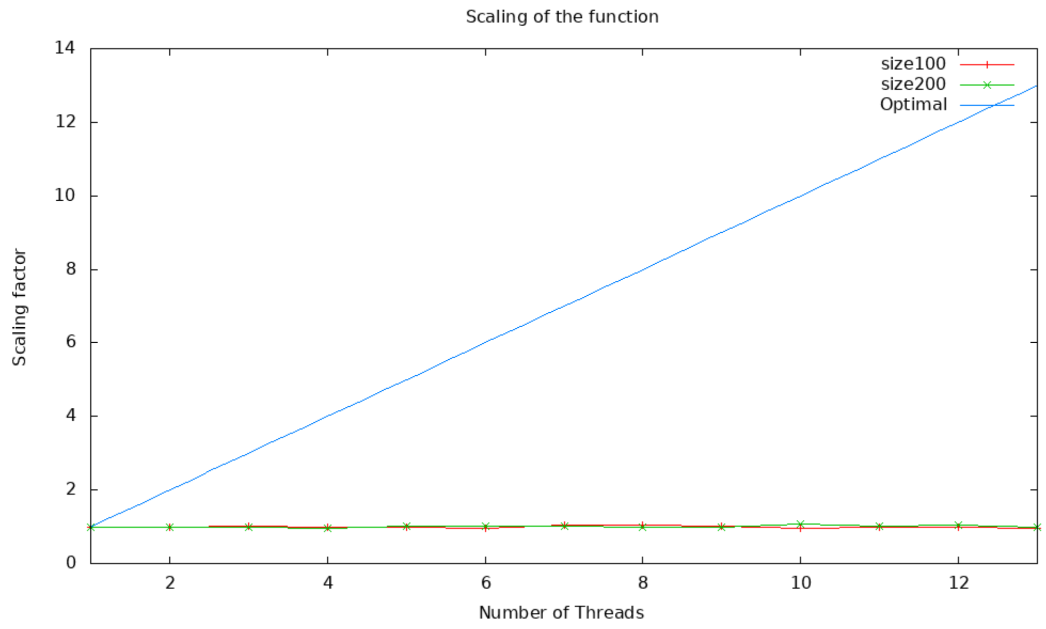


Figure 16: Scaling of Parallel Gauss-Seidel V2

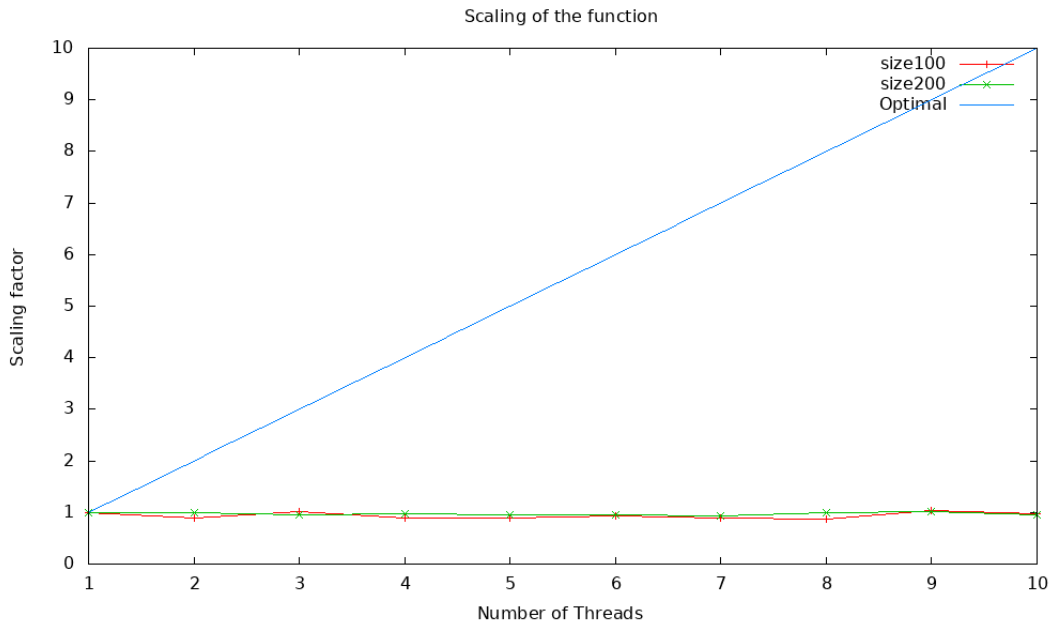
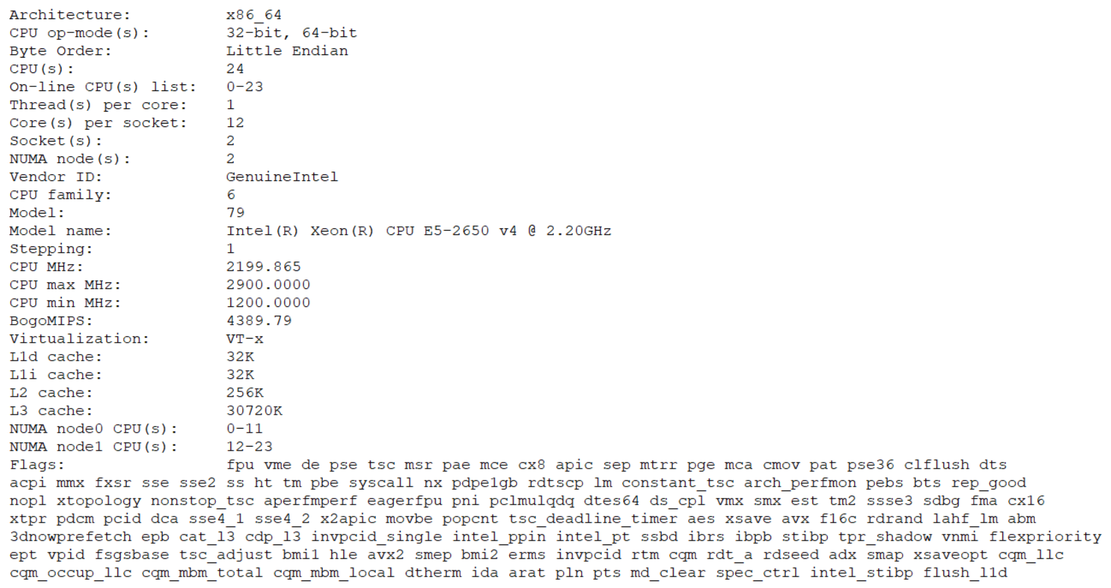


Figure 17: Scaling of Parallel Gauss-Seidel V3

Unfortunately the results weren't as we expected because, as we can see in the above figures, our parallel version of Gauss-Seidel does not seem to scale with respect to the number of threads. Of course the to last versions are better from the first one but still the parallel version is really inefficient. That is maybe due to the fact that we used the `ordered` clause for achieving the correct update of the matrix(u)-values. That is happening because the `ordered` clause sequentializes the execution of ordered regions and that slows down a lot our algorithm because that exact region is executed repeatedly in our algorithm.

CPU architecture

For the efficient palatalization of our code is really important to have a good knowledge of the system you are working on, for that reason we executed the command `lscpu` which gave us the following output: 18



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
Stepping:              1
CPU MHz:               2199.865
CPU max MHz:           2900.0000
CPU min MHz:           1200.0000
BogoMIPS:              4389.79
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-11
NUMA node1 CPU(s):     12-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good
nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16
xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm
3dnowprefetch epb cat_l3 cdp_l3 invpcid_single intel_ppin intel_pt ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdt_a rdseed adx smap xsaveopt cqm_llc
cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts md_clear spec_ctrl intel_stibp flush_l1d
```

Figure 18: CPU architecture

The command lscpu gives us the CPU architecture from which we can extract really useful information. For example the logical socket number (a socket can contain several cores) also we get the number of CPUs, threads, cores, sockets, and Non-Uniform Memory Access (NUMA) nodes. There is also information about the CPU caches and cache sharing, family, model, bogoMIPS, byte order, and stepping.

So I we actually had more time we could use all that information to achieve better efficiency in our code by taking advantage of Non-Uniform Memory Access. To implement that we will parallelize the initialization of our matrices in order to divide move the memory close to different threads and later we could define to access each specific memory part with the same threads which is closer to that part of memory. In that way we could have archived better performance by keeping threads and their data close.