Danmarks
Tekniske
Universitet

OpenMP: The Poisson Problem

Authors

Chandykunju Alex - s200113
Santiago Gutiérrez Orta - s200140
Carlos Marcos Torrejón - s202464
Raúl Ortega Ochoa - s202489

02614 High-Performance Computing Jan 21

January 18, 2021

Contents

Part	**studyno.**	**Name**
I	s202489	Raúl Ortega Ochoa
	s202464	Carlos Marcos Torrejón
	s200140	Santiago Gutierrez Orta
II	s202489	Raúl Ortega Ochoa
III	s202464	Carlos Marcos Torrejón
IV	s200140	Santiago Gutierrez Orta

I Introduction

Partial differential equations appear often in fields such us Physics or Engineering. Often, this systems do not accept analytical solutions but on specific conditions, thus the majority of them need to be solve iteratively by a computer. In this report we implement two iterative methods: Jacobi and Gauss-Seidel, and we compare their performance in the sequential version (with a single thread) and then compare how they scale when parallelizing with a number of threads with OpenMP.

The differential equation that we work with in this report is Poisson's equation in 3 dimensions. This equation will describe the heat distribution in a small cubic room with a radiator placed near the cold wall.

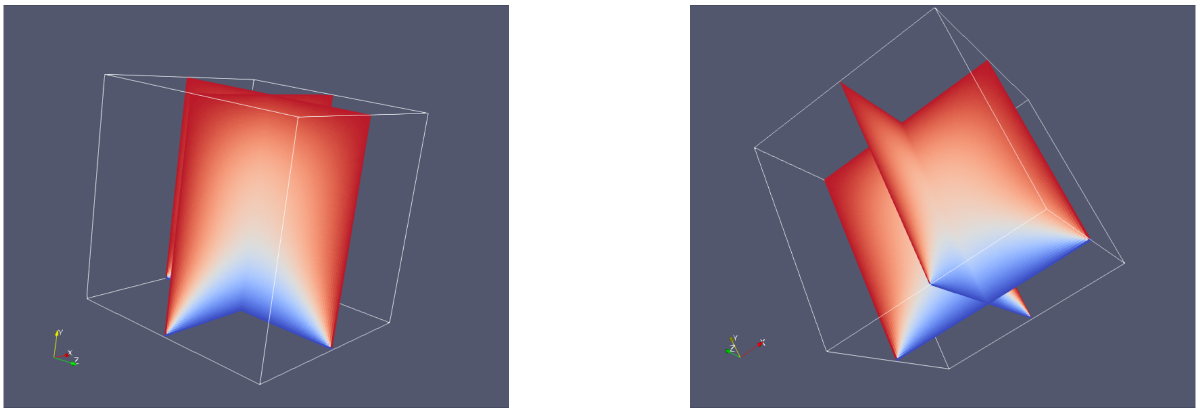

Figure 1: Visual representation of the Poisson problem for a 100x100x100 grid size. The temperature of the region is represented by its color, going from 20°C (red) to 0 (blue). The color fade in between is a consequence of the gradient of temperatures present in the evolution of the system.

The Poisson equation is a generalization of Laplace's equation for $f \neq 0$, and in the case of three dimensions in Cartesian coordinates it is written:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) U(x, y, z) = f(x, y, z) \quad (x, y, z) \ \in \ \Omega$$

Poisson's equation appears frequently in physics, for example in Newtonian gravity $\Delta U = 4\pi G\rho$ or electrostatics $\Delta U = -\rho/\varepsilon$. In this case we will use it to describe a physical system on which a radiator heats up a room. Our space is then $\Omega = \{x, y, z : |x|, |y|, |z| \leq 1\}$. In this case out function $f(x, y, y)$ is defined as:

$$f(x, y, z) = \begin{cases} 200, & if x \in [-1, -\frac{3}{8}], y \in [-1, -\frac{1}{2}], z \in [-\frac{2}{3}, 0] \\ 0, & \text{elsewhere} \end{cases}$$

And the Dirichlet boundary conditions are:

$$U(x,1,z) = 20 \quad U(x,-1,z) = 0 \quad \forall x \in \Omega$$

$$U(1,y,z) = U(-1,y,z) = 20 \quad \forall y \in \Omega$$

$$U(x,y,1) = U(x,y,-1) = 20 \quad \forall z \in \Omega$$

I.I Experimental Hardware Specification

```
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:               2200.000
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
```

II Sequential code - Jacobi method

For the first part the results are computed using Jacobi iterative method. This method will consist of three for nested loops inside a while loop conditional that will limit the number of iterations and fix the tolerance of the updates to check for convergence of theh seven-point stenncil formula. Three different matrixes will be used to hold the memory of the old version to the new one: *uOld*, *uNew*, *uSwap*. In the Jacobi method, once the three nested for loops finish updating *uNew* it is copied to *uOld* after the next iteration of the while loop begins and it is used to update *uNew* once again. The tolerance for the iterative process is iteratively set to be slower than the difference between updates of u, computed as the squared norm 2. The tolerance will be set to 1 for both the Jacobi and the Gauss-Seidel methods. The code is shown down below.

```c
void
jacobi(double*** uNew, double*** uOld, double*** uSwap, double*** f, int N, int iter_max,
    double gridSpace, double tolerance) {
    double invCube = 1/6.;
    double d=100000.0;
    int i, j, k, iter;
   for (iter = 0; (iter < iter_max && d > tolerance); iter++) {
        d = 0.0;
        uSwap = uNew;
        uNew = uOld;
        uOld = uSwap;

        for (i = 1; i < N-1; i++) {
                for (j = 1; j < N-1; j++) {
                        for (k = 1; k < N-1; k++) {

                                /* Compute update of uNew */
                                uNew[i][j][k] = invCube*(uOld[i-1][j][k] + uOld[i+1][j][k]
                                        + uOld[i][j-1][k] + uOld[i][j+1][k] + uOld[i][j][k
                                            -1] + uOld[i][j][k+1] + gridSpace*f[i][j][k]);
                                d += (uOld[i][j][k]-uNew[i][j][k])*(uOld[i][j][k]-uNew[i][j
                                    ][k]);
                        }
                }
        }
        }
}
```

size(N)	iter	time(total)	iter/s	memory(kBytes)
50	639	0.214	2988.962	3000
75	1248	1.563	798.437	10125
100	1977	5.444	363.141	24000
125	2812	16.573	169.674	46875
150	3717	38.510	96.522	81000
175	4705	86.989	54.087	128625
200	5758	145.758	39.504	192000

Table 1: Jacobi method results for different function sizes.

III Sequential code - Gauss-Seidel method

The Gauss-Seidel method follows the same steps than the Jacobi method, but with one key difference. In Jacobi method *uNew* was updated using *uOld* values from theh previous while loop iteration, so it generates the updates in "blocks" of the three nested for loops. In Gauss-Seidel method not only *uOld* values will be used but also the available *uNew* values iteratively generated in the three nested for loops, this will lead to more parallelization difficulties when using OpenMP. The code is shown down below.

```
void
gauss_seidel(double*** uNew, double*** uOld, double*** uSwap, double*** f, int N, int
    iter_max, double gridSpace, double tolerance) {
    double invCube = 1/6.;
    double d=100000.0;
    int i, j, k, iter;

    for (iter = 0; (iter < iter_max && d > tolerance); iter++) {
        d = 0.0;

        uSwap = uNew;
        uNew = uOld;
        uOld = uSwap;

        for (i = 1; i < N-1; i++) {
                for (j = 1; j < N-1; j++) {
                        for (k = 1; k < N-1; k++) {

                                /* Compute update of uNew */
                                uNew[i][j][k] = invCube*(uNew[i-1][j][k] + uOld[i+1][j][k]
                            + uNew[i][j-1][k] + uOld[i][j+1][k] + uNew[i][j][k-1]
                            + uOld[i][j][k+1] + gridSpace*f[i][j][k]);

                                d += (uOld[i][j][k]-uNew[i][j][k])*(uOld[i][j][k]-uNew[i][j
                                    ][k]);
                        }
                }
        }
        }
}
```

size(N)	iter	time(total)	iter/s	memory(kBytes)
50	487	0.437	1113.958	3000
75	995	3.301	301.447	10125
100	1637	13.345	122.670	24000
125	2401	39.355	61.008	46875
150	3259	93.827	34.734	81000
175	4217	195.754	21.542	128625
200	5258	366.375	14.351	192000

Table 2: Gauss-Seidel method results for different function sizes.

The performance of both iterative algorithms can be compared attending to the number of iterations per second for different function sizes. Sice three matrices are needed to be allocated for the iterative proccess, the total memory footprint in is calculated as $3 \times N^3 \times 8\ bytes(double)$. This operation can also be performed updating $uOld$ with another triple for loop before instead of using a $uSwap$ matrix, but we observed a decrease in performance using such method. For that reason, the results shown in this report will all be computing using a $uSwap$ matrix for in Jacobi and Gauss-Seidel methods.

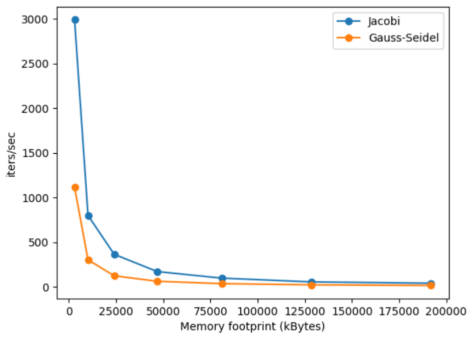

Figure 2: Comparison v1. Iters/sec vs Memory footprint.

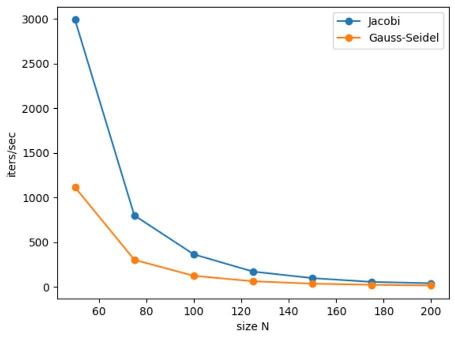

Figure 3: Comparison v1. Iters/sec vs matrix size N.

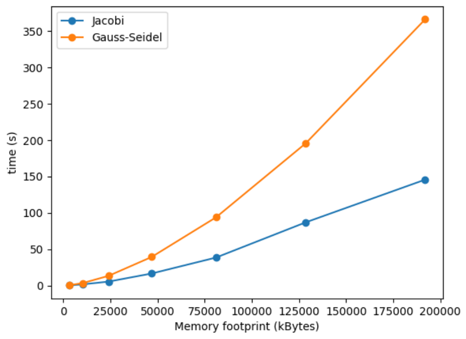

Figure 4: Comparison v1. Total time (s) vs Memory footprint.

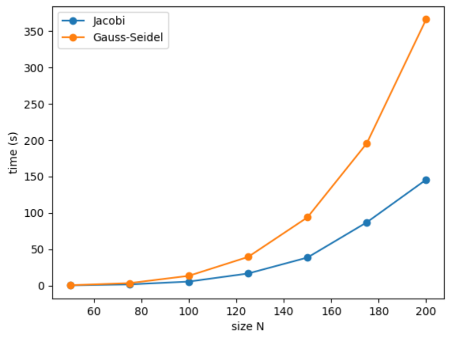

Figure 5: Comparison v1. Total time (s) vs Memory footprint.

As we can see from Fig. 1, the Jacobi method seems to provide better performance than Gauss-Seidel method for all the function sizes explored. The gap in runtimes is specially noticeable for higher gri sizes and it narrows as the total memory usage increases. In this case we have only used 1 thread for computing the results. In following sections we will explore how parallelization affects the performance of these two algorithms and how the behaviour seen in Figures 2-5 evolves with it.

IV OpenMP Jacobi

Parallelization of Jacobi's iterative method had less restrictions that Gauss', due to Gauss not only using *uOld* for the updates but also *uNew*. In this section we only parallelize the function *jacobi()* although we can also do the same for the initialization functions *init_f(), init_3d()* because this functions contain loops that only run once and so the speedup will not be significantly affected by their parallelization.

Jacobi v1

There are different ways to parallelize Jacobi with *OpenMP* syntax, here we show two different approaches and compare the results. The first one, *Jacobi v1* is the simpler approach where before the three nested for loops a call to `#pragma omp parallel for` is added, as can be seen in the code below. Also notice how the three nested for loop is splitted in two so that the update of `d` doesn't create any data race problems.

```
int
jacobi(double*** uNew, double*** uOld, double*** uSwap, double*** f, int N, int iter_max,
    double gridSpace, double tolerance) {
    double invCube = 1/6.;
    double d=100000.0;
    int i, j, k, iter;

        for (iter = 0; (iter < iter_max ||  d > tolerance); iter++) {
            d = 0.0;
            uSwap = uNew;
            uNew = uOld;
            uOld = uSwap;
        #pragma omp parallel for
            for (i = 1; i < N-1; i++) {
                for (j = 1; j < N-1; j++) {
                    for (k = 1; k < N-1; k++) {
                        /* Compute update of uNew */
                        uNew[i][j][k] = invCube*(uOld[i-1][j][k] + uOld[i+1][j][k]
                            + uOld[i][j-1][k] + uOld[i][j+1][k] + uOld[i][j][k-1]
                            + uOld[i][j][k+1] + gridSpace*f[i][j][k]);
                    }
                }
            }

            for (i = 1; i < N-1; i++) {
                for (j = 1; j < N-1; j++) {
                    for (k = 1; k < N-1; k++) {
                        /* Compute new d */
                        d += abs(uNew[i][j][k] - uOld[i][j][k]);
                    }
                }
            }
        }
    return iter;
}
```

This version of Jacobi's method is then run for different sizes and on 1, 2, 4, 8 and 16 threads to compare the performance. For every run we collect the total number of iterations that it ran, the total time, the number of iterations per second, the memory footprint and the

MLUPS (Million Lattice Updates per Second). The memory footprint is computed using:

$$\text{Memory footprint (kBytes)} = 3 \cdot \frac{N^3 \cdot 8 \text{ (Bytes)}}{10^3}$$

Where N^3 is the number of elements in the matrices, 8 is the size of a *double* type then the fraction represents the total memory footprint of updating a matrix a number *iters* of times, and since we have three matrices *uOld, uNew, uSwap* then it is multiplied by 3. The MLUPS is computed using:

$$MLUPS = 3 \cdot \frac{iters \cdot N^3}{10^6 \cdot time}$$

Where *time* is the total runtime of the execution. In *Table 3, in Appendix* this results are shown. This results are then represented in the plots *figures 6-12* shown below.

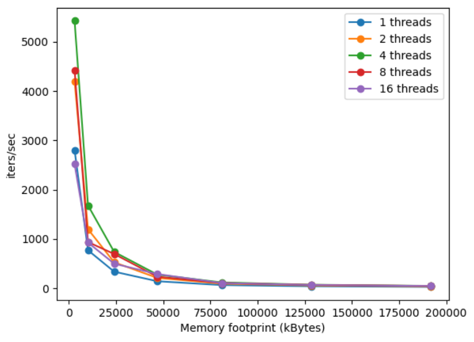

Figure 6: Jacobi v1. Iters/sec vs Memory footprint.

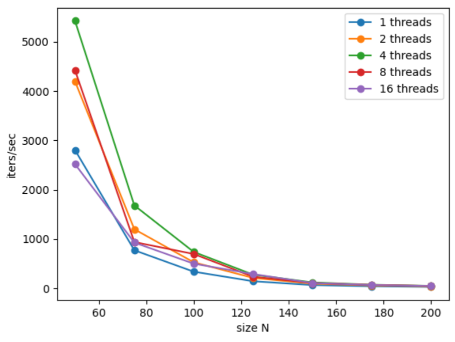

Figure 7: Jacobi v1. Iters/sec vs matrix size N.

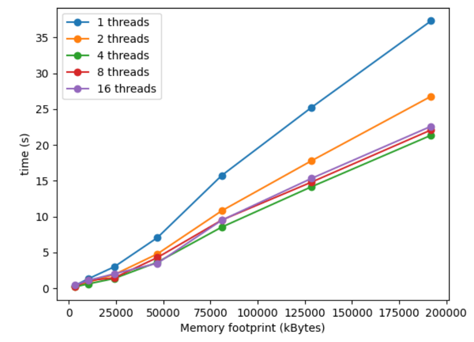

Figure 8: Jacobi v1. Total time (s) vs Memory footprint.

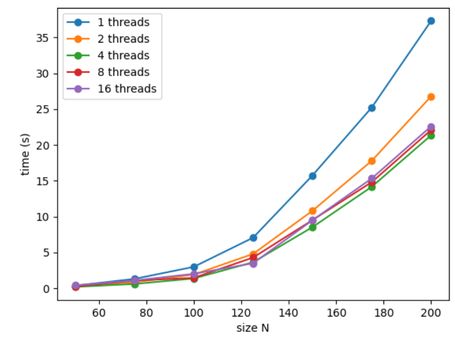

Figure 9: Jacobi v1. Total time (s) vs matrix size N.

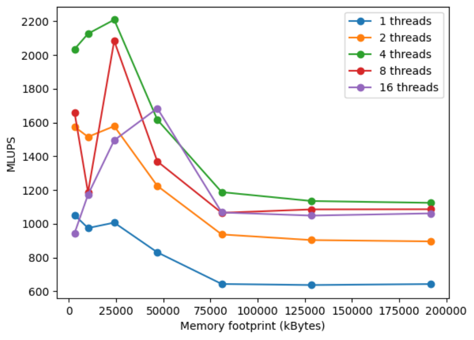

Figure 10: Jacobi v1. MLUPS vs Memory footprint.

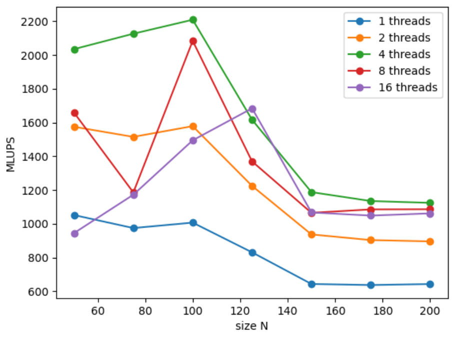

Figure 11: Jacobi v1. MLUPS vs matrix size N.

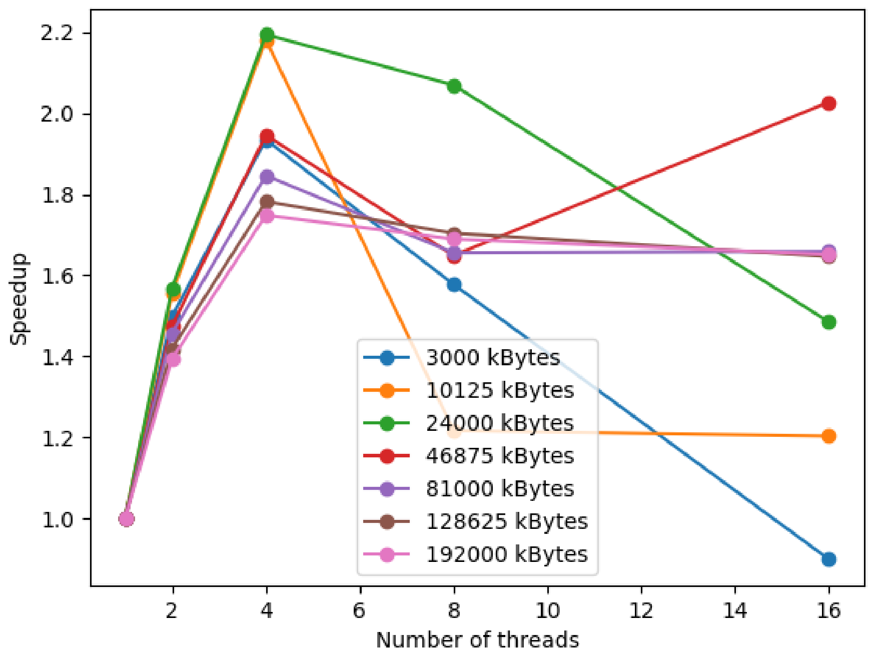

Figure 12: Jacobi v1. Speedup vs Number of threads.

As can be seen from *figures 6-12*, and specially in *figures 10, 11, 12* for this version of Jacobi's method parallellized we achieve the optimal performance with 4 threads. From *figure 12* we can see how the speedup is approximately equal for the different memory footprints (related with the size of the matrix) for less than 4 threads, but already with 4 threads the difference in speedup is noticeable and after this the evolution of the different lines is really sparse. For the case of lower memory footprint, speedup goes down as we increase the number of threads, while for medium to the higher memory footprints tend to stabilize after 8 threads.

Jacobi v2

Some upgrades can be made from the previous version. the omp parallel condition can be specified outside the while loop, stating its conditional *iter_max* as the firstprivate variable. As the matrixes and *iter* are shared variables we decided to nest them in a single thread after each while loop. Finally, the *d* variable is calculated iteratively by the three nested for loops after the the update of *uNew* matrix. This lead to performance problems on multi-thread runs. A way to fix this was to add a reduction option to the variable *d*. This way, the reduction takes care of making a private copy of *d* for each thread. Once the parallel region finishes, *d* is reduced in one atomic operation, aggregating all the private areas of each thread.

```c
int
jacobi(double*** uNew, double*** uOld, double*** uSwap, double*** f, int N, int iter_max,
    double gridSpace, double tolerance) {
    double invCube = 1./6.;
    double d=100000.0;
    int i, j, k, iter=0;
    #pragma omp parallel shared(uNew, uOld, uSwap, f, N, gridSpace, tolerance, iter)
        firstprivate(iter_max)
    {
    while (iter < iter_max || d > tolerance){
          #pragma omp single
          {
        d = 0.0;
            uSwap=uOld;
            uOld=uNew;
            uNew=uSwap;
          }
       #pragma omp for private(i,j,k) reduction(+:d)
           for (i = 1; i < N-1; i++) {
               for (j = 1; j < N-1; j++) {
                    for (k = 1; k < N-1; k++) {

                          /* Compute update of uNew */
                          uNew[i][j][k] = invCube*(uOld[i-1][j][k] + uOld[i+1][j][k]
                              + uOld[i][j-1][k] + uOld[i][j+1][k] + uOld[i][j][k-1]
                              + uOld[i][j][k+1] + gridSpace*f[i][j][k]);
                    d += abs(uNew[i][j][k] - uOld[i][j][k]);
                    }
               }
           }
      #pragma omp single
      {
        iter++;
      }
        }

        }
    return iter;
}
```

The results of this implementation are shown in *Table 4, at the Appendix*. This results are then plotted in *figures 13-19*, shown below.

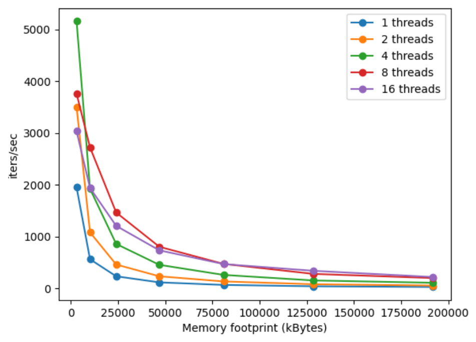

Figure 13: Jacobi v2. Iters/sec vs Memory footprint.

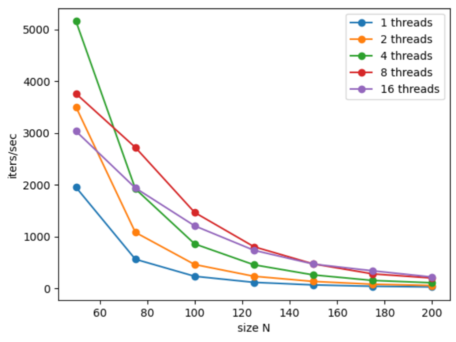

Figure 14: Jacobi v2. Iters/sec vs matrix size N.

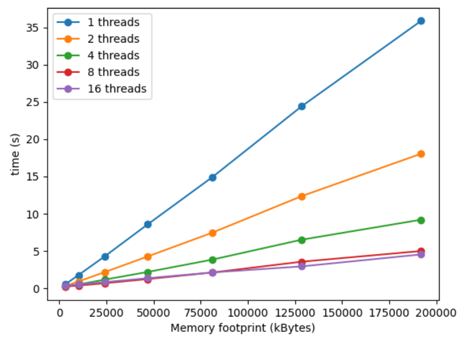

Figure 15: Jacobi v2. Total time (s) vs Memory footprint.

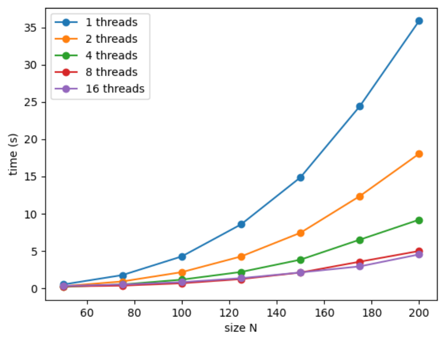

Figure 16: Jacobi v2. Total time (s) vs Memory footprint.

For this version of the Jacobi method parallelization there is an improvement in the performance of higher threads runs, which translates into lower runtimes specially for the 4, 8 and 16 threads cases.

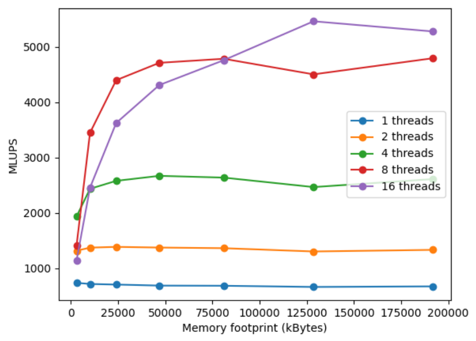

Figure 17: Jacobi v2. MLUPS vs Memory footprint.

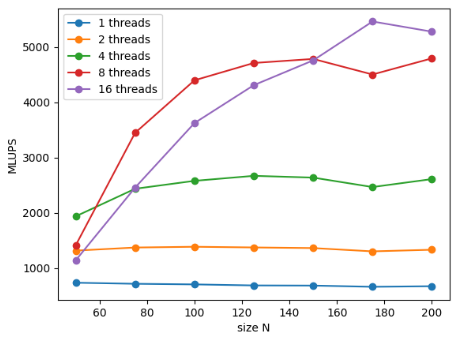

Figure 18: Jacobi v2. MLUPS vs matrix size N.

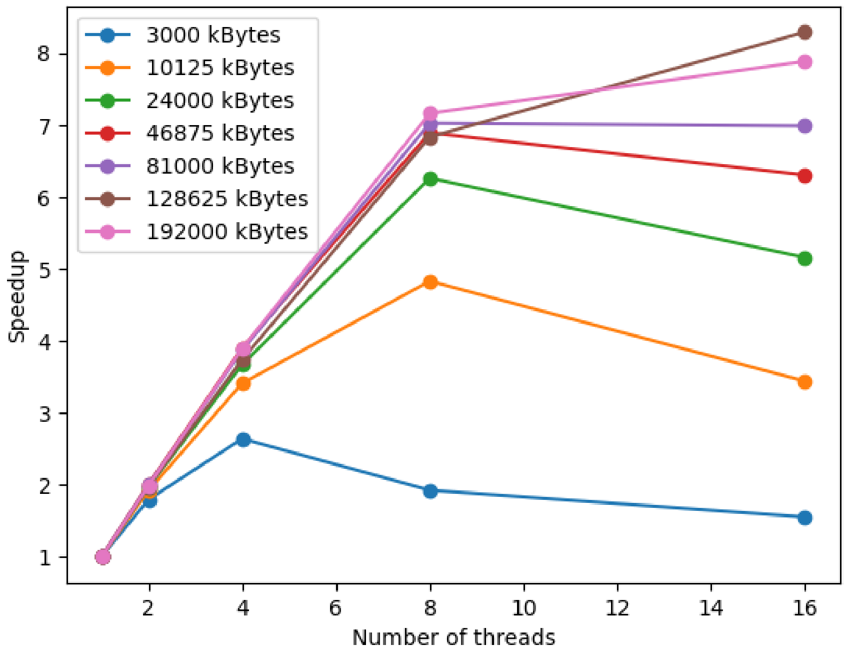

Figure 19: Jacobi v2. Speedup vs Number of threads.

In the first Jacobi parallelization method we observed a drastic fall in speedup using more than 4 threads. Nevertheless, for this second version we get a more consistent speedup as we increase the number of threads for the largest memory footprints, and it flattens for more than 8 threads. A better usage of the multi-threads can also be seen in figures 34-35 of the Appendix with respect to the previous version.

Jacobi v3

This third version contain elements that allows the function to parallelize the Jacobi method applying a threshold tolerance. This problem in performance could be caused by data races in the calculation of d and its reinitialization at the beginning of the while loop. We experimented that adding a pragma barrier and consequently performing the reinitialization of d and the matrices inside the master thread greatly fixed this problem. The computation was performed using a tolerance of 1 as well as in sections 2 and 3 for the sequential method.

```
int
jacobi(double*** uNew, double*** uOld, double*** uSwap, double*** f, int N, int iter_max,
    double gridSpace, double tolerance) {
    double invCube = 1./6.;
    double d=100000.0;
    int i, j, k, iter=0;
        #pragma omp parallel shared(uNew, uOld, uSwap, f, N, gridSpace, tolerance, iter, d)
            firstprivate(iter_max)
        {
        while (iter < iter_max && d>tolerance){
        #pragma omp barrier
        #pragma omp master
          {
            d = 0.0;
            uSwap=uOld;
            uOld=uNew;
            uNew=uSwap;
          }
           #pragma omp for private(i,j,k) reduction(+:d)
           for (i = 1; i < N-1; i++) {
                   for (j = 1; j < N-1; j++) {
                             for (k = 1; k < N-1; k++) {

                                       /* Compute update of uNew */
                                       uNew[i][j][k] = invCube*(uOld[i-1][j][k]
                                       + uOld[i+1][j][k] + uOld[i][j-1][k] + uOld[i][j+1][k]
                                              + uOld[i][j][k-1] + uOld[i][j][k+1]
                                              + gridSpace*f[i][j][k]);
                                       d += (uOld[i][j][k]-uNew[i][j][k])*(uOld[i][j][k]
                                       -uNew[i][j][k]);
                          }
                      }
                  }
          #pragma omp master
          {
            iter++;
            //printf("%i\n",d);
          }
        }

        }
    return iter;
}
```

The results of this implementation are shown in *Table 5, at the Appendix*. Using this results the plots shown below in *figures 20 - 26* are generated.

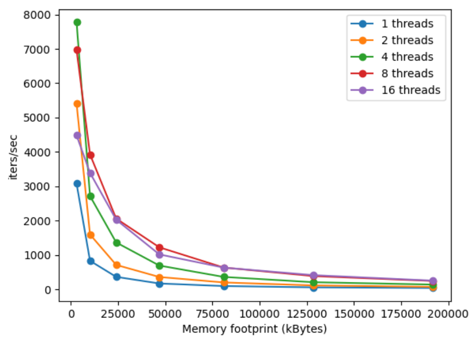

Figure 20: Jacobi v3. Iters/sec vs Memory footprint.

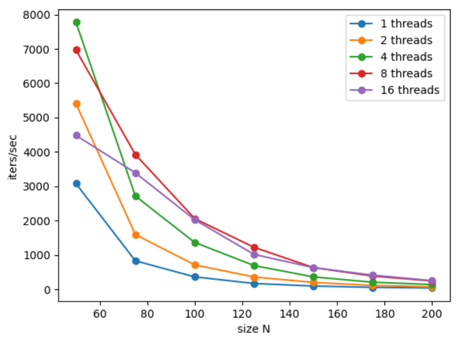

Figure 21: Jacobi v3. Iters/sec vs matrix size N.

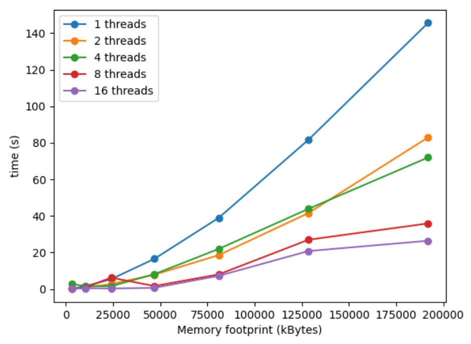

Figure 22: Jacobi v3. Total time (s) vs Memory footprint.

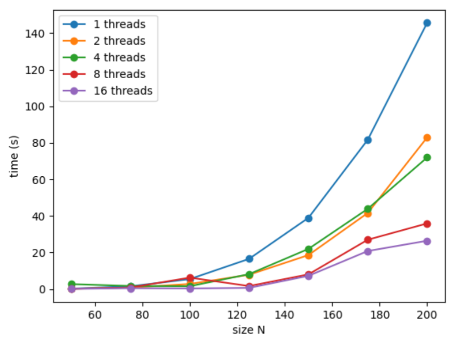

Figure 23: Jacobi v3. Total time (s) vs matrix size N.

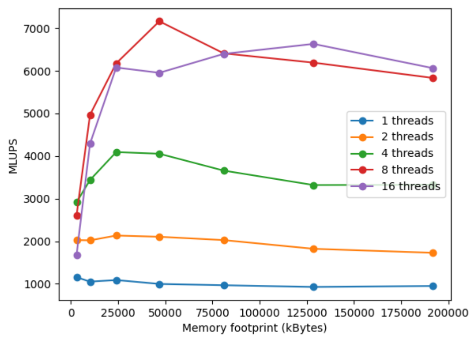

Figure 24: Jacobi v3. MLUPS vs Memory footprint.

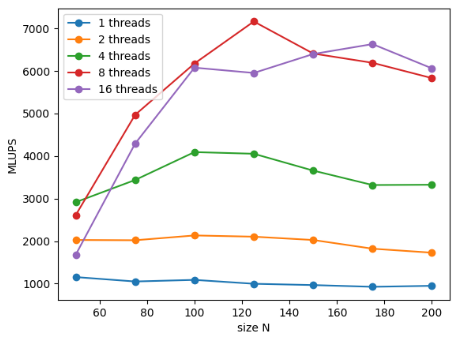

Figure 25: Jacobi v3. MLUPS vs matrix size N.

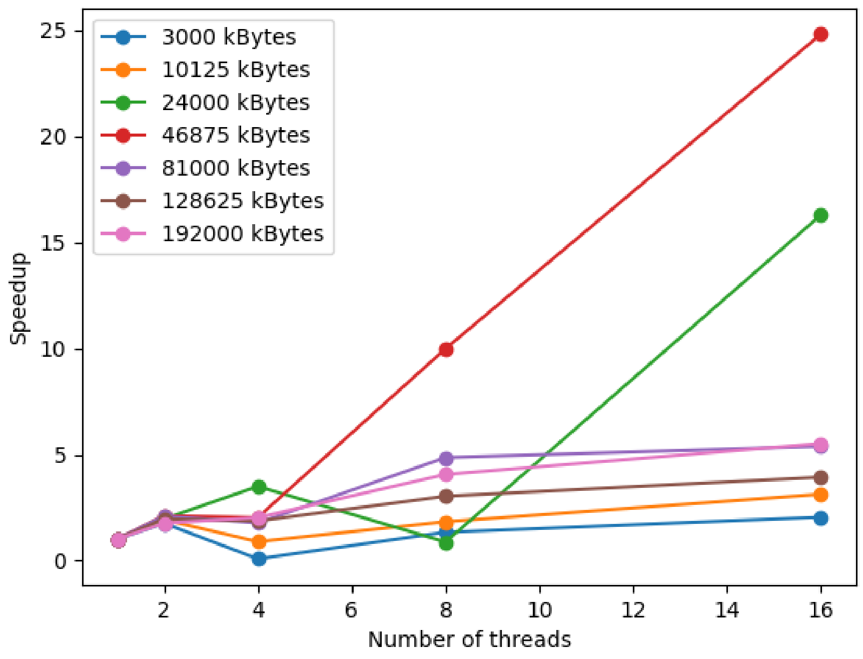

Figure 26: Jacobi v3. Speedup vs Number of threads.

For this last version of the Jacobi paralelization the runtimes decreases for higher number of threads used as it is expected, but the speedup is affected by some weird behaviour for certain number of threads used in some runs. This effect can be explained by the fact that the master thread carries most of the workload during the proccess as can be seen in Figure 36 of the Appendix. The parts concerning the three nested for loop are carried in parallel among the different threads, which leads to speedup but for the rest of the while loop the master thread takes most of the work.

V OpenMP Gauss-Seidel

The method Gauss-Seidel, in contrast with Jacobi, have limitations with respect to the parallelization of the code. The Jacobi method can be parallelized roughly without any additional clauses, while Gauss-Seidel need a different approach to deal with static dependencies. For implementing our parallel version of Gaus-Seidel, we applied doacross-loops which provide logic inside the iterations defining the loop-carried dependency and its distance.

In order to define the dependencies in the doacross-loop it was used the OMP ordered(d) construct which allows to execute the block in sequential order, this is important since for compute an new element, a previous element is needed. Furthermore, the schedule used for this task is static with chunk-size 1. With this schedule, the iterations are divided into chunks of size 1 and distributed to threads in a circular order so that each thread have to wait the previous one to finish so we make sure that the sequence flow benefit the fulfilling of the dependencies of the new iteration element.

```c
int
gauss_seidel(double*** uNew, double*** p, double*** uSwap, double*** f, int N, int iter_max,
     double gridSpace, double tolerance) {
    double invCube = 1/6.;
    double d=100000.0;
    int i, j, k, iter;

    for (iter = 0; (iter < iter_max); iter++) {
        d = 0.0;
        uSwap = uNew;
        uNew = p;
        p = uSwap;
        #pragma omp parallel
        {
        #pragma omp for schedule(static,1)  ordered(2) private(j,k)
        for (i = 1; i < N-1; ++i) {
            for (j = 1; j < N-1; ++j) {
        #pragma omp ordered depend(sink: i-1,j-1) depend(sink: i-1,j) \
                        depend(sink: i-1,j+1) depend(sink: i,j-1)
                for (k = 1; k < N-1; ++k) {
                    double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                                + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                                + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
                    double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                                + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                                + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
                    double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                                + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                                + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);

                    uNew[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
                    //printf("%3.4f  ", uNew[i][j][k]);
                }
                //printf("\n");
        #pragma omp ordered depend(source)
            }
        }
        #pragma omp for schedule(dynamic)
        for (i = 1; i < N-1; i++) {
            for (j = 1; j < N-1; j++) {
                for (k = 1; k < N-1; k++) {

                    /* Compute new d */
                    d += abs(uNew[i][j][k] - p[i][j][k]);
                }
            }
        }
        //printf("distance : %8.8f", d);printf("\n");
        }
    }
return iter;
}
```

The collected data can be found in the appendix in Table 6, at the Appendix. The following graphs presented in the figures 27 - 33 show these results.

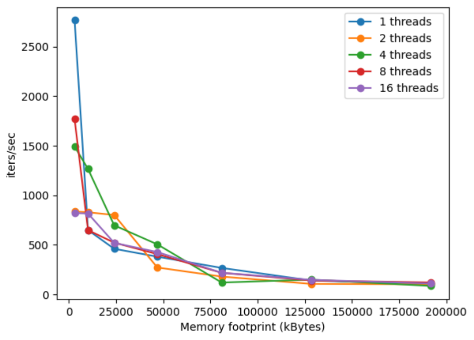

Figure 27: Gauss-Seidel. Iters/sec vs Memory footprint.

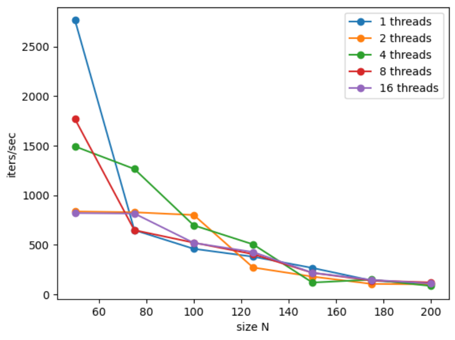

Figure 28: Gauss-Seidel. Iters/sec vs matrix size N.

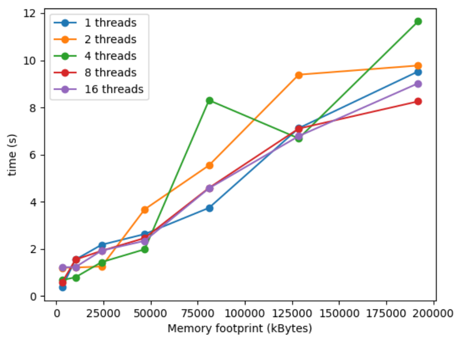

Figure 29: Gauss-Seidel. Total time (s) vs Memory footprint.

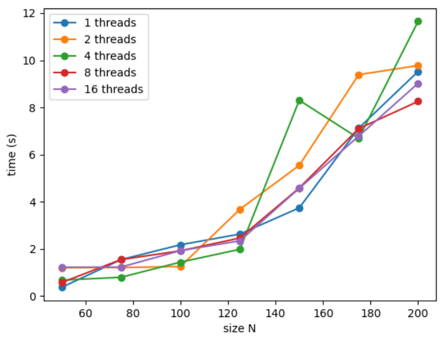

Figure 30: Gauss-Seidel. Total time (s) vs Memory footprint.

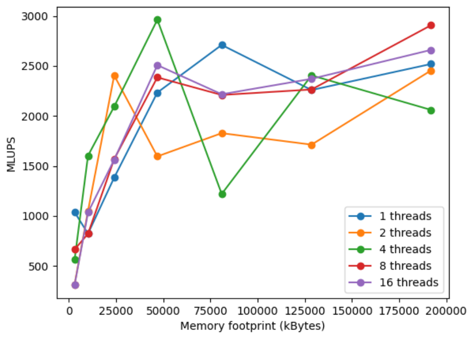

Figure 31: Gauss-Seidel. MLUPS vs Memory footprint.

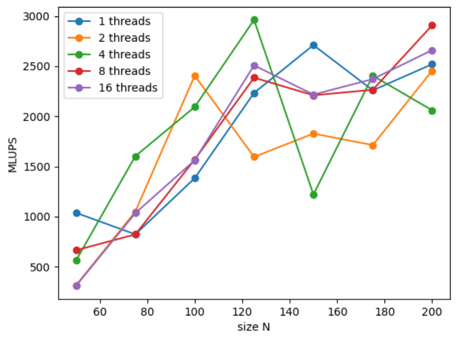

Figure 32: Gauss-Seidel. MLUPS vs size N.

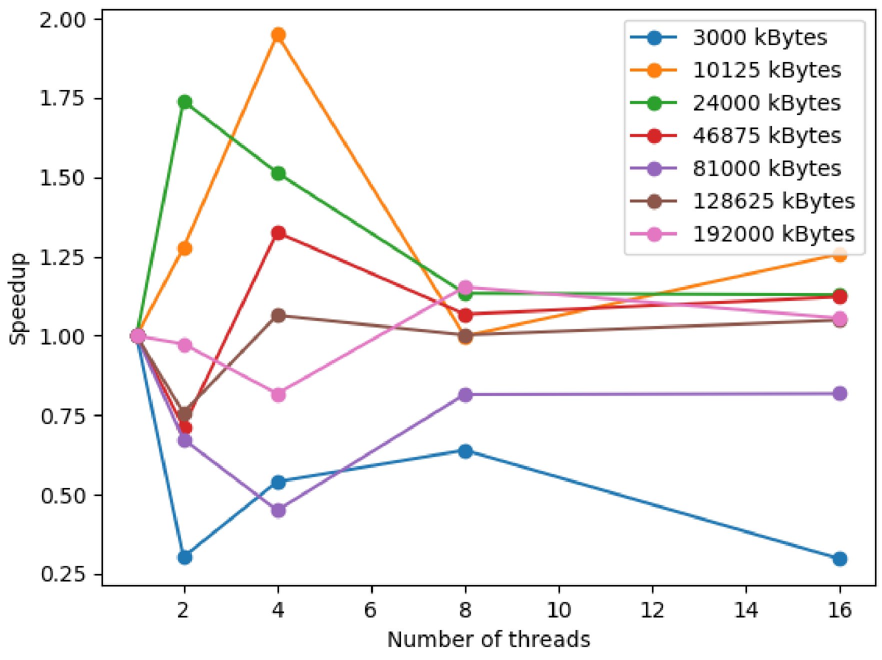

Figure 33: Gauss-Seidel. Speedup vs Number of threads.

Since the Gauss-Seidel parallezition requires a large synchronization computation, we expect a worse performance than the sequential execution so that a longer convergence time. However, the convergence condition was not the same in the different implementations .Additionally, as we can infer from the results, the increase on the number of threads is independent of the performance .That probably is caused by the fact that OMP is assigning sequentially each of the iterations to each thread and each thread have to wait the previous one to finish. Therefore, the speed up does not increase with the number of threads.

VI Conclusion

In this assignment we evaluate the performance of two different iterative methods: Jacobi and Gauss-Seiddel applyied to Poisson's problem. We initially compared the performance using an only thread. From there we built a tree of strategies to parallelize the operations of Jacobi and Gauss-Seidel methods using multiple threads.

Among the versions of parallelized Jacobi method, the second one provides the best multi-thread performance, as it improves over the first one the share of workload among multiple threads. The third version however, overloads the master thread, which translates into long sleep intervals in the rest of the threads.

The parallelization of the Gauss-Seidel method does not gives a visible improvement on the sequential method, probably, due to the fact that the parallel method spend a big amount of time in synchronization. Because this synchronization, there is not relation between the increase in number of threads and the algorithm speed, since, the iterations are split one by one on each thread with the static schedule and with the ordered construct, the thread have to wait a previous one to finish.

VII Appendix

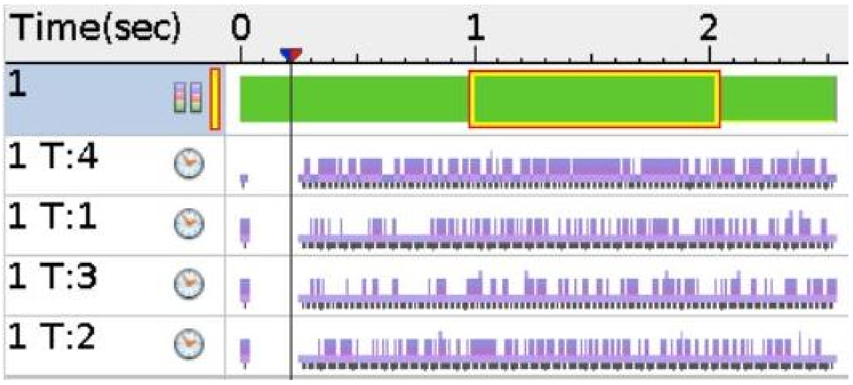

Figure 34: Threads work analyzer for Jacobi parallelization v1.

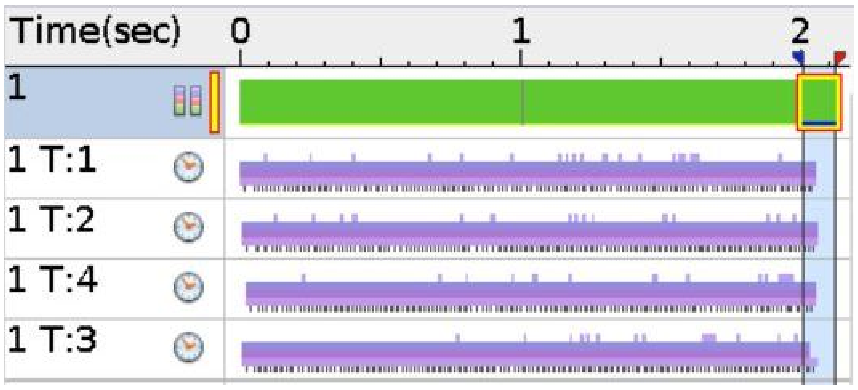

Figure 35: Threads work analyzer for Jacobi parallelization v2.

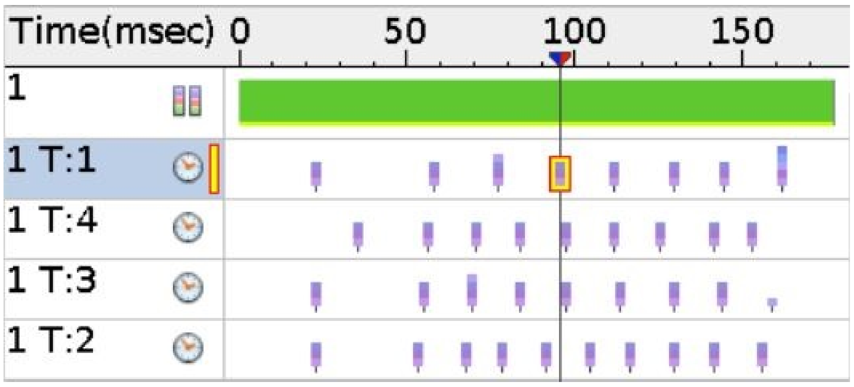

Figure 36: Threads work analyzer for Jacobi parallelization v3.

size(N)	iter	time(s)	iter/s	Mem(kBytes)	Threads	SpeedUp	LUPS ($\cdot 10^9$)
50	1000	0.356598	2804.27899	3000	1	1.0	1.05
50	1000	0.238148	4199.069944	3000	2	1.49	1.57
50	1000	0.184303	5425.84632	3000	4	1.93	2.03
50	1000	0.226071	4423.391443	3000	8	1.57	1.65
50	1000	0.39699	2518.956264	3000	16	0.89	0.94
75	1000	1.298668	770.019798	10125	1	1.0	0.97
75	1000	0.835487	1196.906532	10125	2	1.55	1.51
75	1000	0.595288	1679.860398	10125	4	2.18	2.12
75	1000	1.067455	936.807385	10125	8	1.21	1.18
75	1000	1.079785	926.110606	10125	16	1.20	1.17
100	1000	2.979981	335.572589	24000	1	1.0	1.00
100	1000	1.9007	526.122062	24000	2	1.56	1.57
100	1000	1.358214	736.261184	24000	4	2.19	2.20
100	1000	1.43989	694.497541	24000	8	2.06	2.08
100	1000	2.007628	498.100255	24000	16	1.48	1.49
125	1000	7.053597	141.771638	46875	1	1.0	0.83
125	1000	4.785722	208.954886	46875	2	1.47	1.22
125	1000	3.624995	275.862432	46875	4	1.94	1.61
125	1000	4.278594	233.721638	46875	8	1.64	1.36
125	1000	3.479328	287.411809	46875	16	2.02	1.68
150	1000	15.739728	63.5335	81000	1	1.0	0.64
150	1000	10.808495	92.519819	81000	2	1.45	0.93
150	1000	8.52724	117.271242	81000	4	1.84	1.18
150	1000	9.508747	105.166324	81000	8	1.65	1.06
150	1000	9.488722	105.388275	81000	16	1.65	1.06
175	1000	25.244688	39.612293	128625	1	1.0	0.63
175	1000	17.798661	56.184003	128625	2	1.41	0.90
175	1000	14.168852	70.577349	128625	4	1.78	1.13
175	1000	14.817523	67.487663	128625	8	1.70	1.08
175	1000	15.335684	65.207393	128625	16	1.64	1.04
200	1000	37.330521	26.787732	192000	1	1.0	0.64
200	1000	26.797019	37.317584	192000	2	1.39	0.89
200	1000	21.357054	46.822937	192000	4	1.74	1.12
200	1000	22.098884	45.251154	192000	8	1.68	1.08
200	1000	22.609962	44.228292	192000	16	1.65	1.06

Table 3: Jacobi method (v1. Parallel). Results for different function sizes and different number of threads.

size(N)	iter	time(s)	iter/s	Mem(kBytes)	Threads	SpeedUp	LUPS ($\cdot 10^9$)
50	1000	0.512192	1952.391902	3000	1	1.0	0.73
50	1000	0.285439	3503.369473	3000	2	1.79	1.31
50	1000	0.193833	5159.070893	3000	4	2.64	1.93
50	1000	0.265882	3761.070097	3000	8	1.92	1.41
50	1000	0.329373	3036.072646	3000	16	1.55	1.13
75	1000	1.774967	563.390846	10125	1	1.0	0.71
75	1000	0.924083	1082.154441	10125	2	1.92	1.36
75	1000	0.520105	1922.689113	10125	4	3.41	2.43
75	1000	0.36725	2722.942798	10125	8	4.83	3.44
75	1000	0.515598	1939.494381	10125	16	3.44	2.45
100	1000	4.273684	233.990145	24000	1	1.0	0.70
100	1000	2.1688	461.084497	24000	2	1.97	1.38
100	1000	1.164153	858.993899	24000	4	3.67	2.57
100	1000	0.682249	1465.74139	24000	8	6.26	4.39
100	1000	0.828025	1207.693203	24000	16	5.16	3.62
125	1000	8.575938	116.605324	46875	1	1.0	0.68
125	1000	4.273996	233.973098	46875	2	2.01	1.37
125	1000	2.19648	455.273836	46875	4	3.90	2.66
125	1000	1.243746	804.022967	46875	8	6.89	4.71
125	1000	1.359661	735.47752	46875	16	6.31	4.30
150	1000	14.874967	67.227039	81000	1	1.0	0.68
150	1000	7.446946	134.283228	81000	2	1.99	1.35
150	1000	3.841603	260.307999	81000	4	3.87	2.63
150	1000	2.116245	472.535126	81000	8	7.02	4.78
150	1000	2.127877	469.952029	81000	16	6.99	4.75
175	1000	24.414881	40.958626	128625	1	1.0	0.65
175	1000	12.366683	80.862427	128625	2	1.97	1.30
175	1000	6.523435	153.29348	128625	4	3.74	2.46
175	1000	3.57092	280.039841	128625	8	6.83	4.50
175	1000	2.943551	339.725718	128625	16	8.29	5.46
200	1000	35.865004	27.882333	192000	1	1.0	0.66
200	1000	18.051631	55.396655	192000	2	1.98	1.32
200	1000	9.200098	108.6945	192000	4	3.89	2.60
200	1000	5.005617	199.775576	192000	8	7.16	4.79
200	1000	4.54707	219.921839	192000	16	7.88	5.27

Table 4: Jacobi method (v2. Parallel). Results for different function sizes and different number of threads.

size(N)	iter	time(s)	iter/s	Mem(kBytes)	Threads	SpeedUp	LUPS ($\cdot 10^9$)
50	1000	0.361462	2766.540873	3000	1	1.0	1.03
50	1000	1.194239	837.353655	3000	2	0.30	0.31
50	1000	0.669329	1494.034226	3000	4	0.54	0.56
50	1000	0.56512	1769.534866	3000	8	0.64	0.66
50	1000	1.217779	821.166808	3000	16	0.30	0.30
75	1000	1.53945	649.582642	10125	1	1.0	0.82
75	1000	1.205889	829.264055	10125	2	1.28	1.04
75	1000	0.789711	1266.285511	10125	4	1.95	1.60
75	1000	1.540876	648.981432	10125	8	0.99	0.82
75	1000	1.224002	816.992115	10125	16	1.26	1.03
100	1000	2.169472	460.941695	24000	1	1.0	1.38
100	1000	1.247082	801.871867	24000	2	1.74	2.40
100	1000	1.433051	697.811962	24000	4	1.51	2.09
100	1000	1.913102	522.711406	24000	8	1.13	1.56
100	1000	1.921043	520.550647	24000	16	1.13	1.56
125	1000	2.623011	381.241229	46875	1	1.0	2.23
125	1000	3.673205	272.241829	46875	2	0.71	1.59
125	1000	1.977898	505.587314	46875	4	1.32	2.96
125	1000	2.4559	407.182662	46875	8	1.07	2.38
125	1000	2.336033	428.076178	46875	16	1.12	2.50
150	1000	3.735622	267.693018	81000	1	1.0	2.71
150	1000	5.540191	180.499188	81000	2	0.67	1.82
150	1000	8.302553	120.44488	81000	4	0.45	1.21
150	1000	4.581801	218.254809	81000	8	0.82	2.20
150	1000	4.567057	218.959372	81000	16	0.82	2.21
175	1000	7.116563	140.517273	128625	1	1.0	2.25
175	1000	9.387658	106.522837	128625	2	0.76	1.71
175	1000	6.683342	149.625749	128625	4	1.06	2.40
175	1000	7.099784	140.849355	128625	8	1.00	2.26
175	1000	6.780504	147.481652	128625	16	1.05	2.37
200	1000	9.524809	104.988978	192000	1	1.0	2.51
200	1000	9.774868	102.303167	192000	2	0.97	2.45
200	1000	11.649586	85.839958	192000	4	0.82	2.06
200	1000	8.253957	121.154011	192000	8	1.15	2.90
200	1000	9.020174	110.862607	192000	16	1.06	2.66

Table 5: Jacobi method (v3. Parallel). Results for different function sizes and different number of threads.

size(N)	iter	time(s)	iter/s	Mem(kBytes)	Threads	SpeedUp	LUPS ($\cdot 10^9$)
50	639	0.207642	3077.410922	3000	1	1.0	1.80
50	640	0.118447	5403.279301	3000	2	1.75	3.16
50	21076	2.71088	7774.597061	3000	4	0.076	0.13
50	1089	0.156315	6966.687292	3000	8	1.33	2.39
50	456	0.101765	4480.915543	3000	16	2.04	3.68
75	1248	1.50364	829.985936	10125	1	1.0	0.84
75	1249	0.782542	1596.079513	10125	2	1.92	1.61
75	4621	1.701396	2716.005374	10125	4	0.88	0.74
75	3230	0.822659	3926.294869	10125	8	1.83	1.53
75	1635	0.482139	3391.137463	10125	16	3.12	2.62
100	1977	5.450125	362.743943	24000	1	1.0	0.55
100	1979	2.783371	711.008227	24000	2	1.96	1.07
100	2131	1.561698	1364.540426	24000	4	3.49	1.92
100	12917	6.273473	2058.987252	24000	8	0.87	0.47
100	678	0.334586	2026.385421	24000	16	16.29	8.96
125	2812	16.529265	170.12251	46875	1	1.0	0.35
125	2818	7.847868	359.078416	46875	2	2.11	0.74
125	5643	8.155597	691.917447	46875	4	2.03	0.71
125	2025	1.656025	1222.8075	46875	8	9.98	3.53
125	677	0.666144	1016.296574	46875	16	24.80	8.79
150	3717	38.962655	95.399042	81000	1	1.0	0.25
150	3721	18.594941	200.108192	81000	2	2.10	0.54
150	7938	21.969381	361.321061	81000	4	1.77	0.46
150	5087	8.033325	633.237175	81000	8	4.85	1.26
150	4578	7.242796	632.076305	81000	16	5.38	1.39
175	4705	81.737026	57.562652	128625	1	1.0	0.19
175	4718	41.64083	113.302257	128625	2	1.96	0.38
175	9065	43.914575	206.423494	128625	4	1.86	0.36
175	10402	27.003671	385.20689	128625	8	3.03	0.59
175	8590	20.820517	412.573812	128625	16	3.92	0.77
200	5758	145.657074	39.531207	192000	1	1.0	0.16
200	5973	82.980468	71.980794	192000	2	1.75	0.28
200	9977	72.001645	138.566278	192000	4	2.02	0.33
200	8730	35.915012	243.073843	192000	8	4.05	0.66
200	6680	26.440216	252.645444	192000	16	5.50	0.90

Table 6: Gauss method (v1. Parallel). Results for different function sizes and different number of threads.

List of Figures

List of Tables