High Performance Computing

Assignment 2: The Poisson Problem

13th January 2017

Miguel Suau de Castro (161333)
Alberto Chiusole (162501)
Anders Jakobsen (122467)

Technical University of Denmark
02614 High-Performance Computing

Contents

1 Introduction

In this assignment we are going to apply the Poisson equation to study the heat distribution in a square room. The room is modeled using a square grid of size NxN. Each point in the grid represents a point in the room.

The interior walls of the room are set to a temperature of 20 °C while the fourth wall is fixed at 0 °C. Moreover, a radiator is placed close to the cold wall.

The solution of the set of differential equations is obtained for each point in the grid using the finite difference method, in which all the points are updated in each iteration until the difference between the updated matrix and the old version is smaller than a predefined tolerance.

The update has been carried out with two different approaches called Jacobi and Gauss-Seidel method. We shall study the performances of those methods and try to apply OpenMP parallelization tools to the Jacobi version to speed up the execution.

2 | Sequential Jacobi method

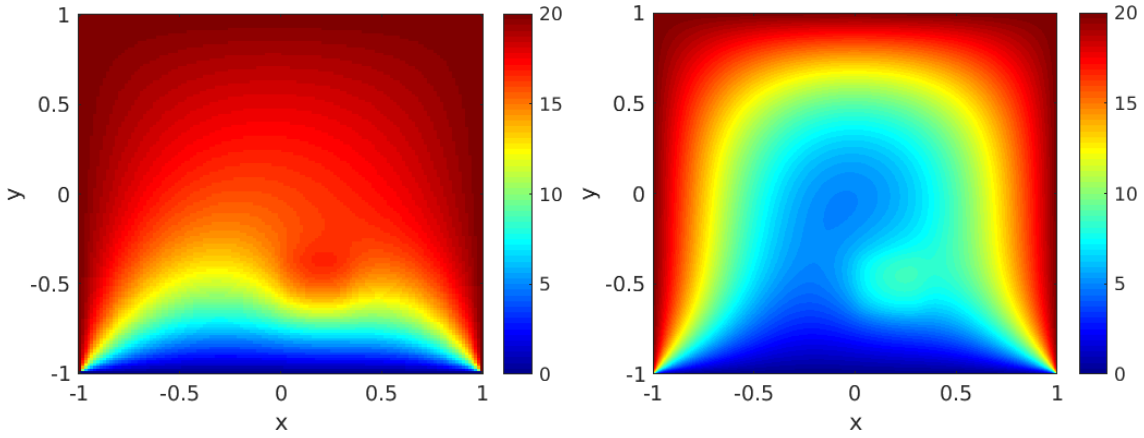
The Jacobi routine accepts two grid matrices as arguments and computes the solution of the Poisson differential equation by applying the finite difference method. It contains three nested loops. The two inner loops run through all the elements of the matrices and update their value, whereas the outer loop takes care of the iterations. It is built in order to stop either when the number of iterations is larger than a prefixed value or when the difference between the new grid matrix and the old version is smaller than the threshold.

Once the matrix has been updated and the difference has been computed, the pointers of the two matrices are swapped, so the following iteration overwrites the values of the old matrix.

We found out in the previous assignment that when using double pointers (e.g. to store matrices) some compiler optimization tools such as loop interchange cannot be applied or do not have any effect. Therefore the routine expects single pointers to the matrices, to encourage and help the compiler optimize the code.

```
1  #include <stdio.h>
2
3  void jacobi(double * unew, double * uold, double * f,
4              double lambda, int N, int kmax, double treshold, int * k){
5
6    double d = treshold+1;
7    double lambda2 = lambda*lambda;
8    int M = N+2;
9    double* swapper;
10
11   for (*k = 0; (*k < kmax && d > treshold); (*k)++){
12     d = 0;
13     for (int i = 1; i < N+1; i++){
14       for (int j = 1; j < N+1; j++){
15         unew[i*M+j] = ( 0.25*(uold[(i-1)*M+j]+uold[(i+1)*M+j]+
16                         uold[i*M+j-1]+uold[i*M+j+1]+lambda2*f[i*M+j]) );
17
18         d += (unew[i*M+j]-uold[i*M+j])*(unew[i*M+j]-uold[i*M+j]);
19       }
20     }
21     swapper = uold;
22     uold = unew;
23     unew = swapper;
24   }
25
26   swapper = unew;
27   unew = uold;
28   unew = swapper;
29 }
```

The code has been tested for different grid sizes. The results of these tests are shown in table 3.1. Figure 2.1 is a contour plot of the solution obtained when using a grid of size N = 128 and N = 1024.



(a) N=128 **(b)** N=1024

Figure 2.1: Visualization of the solution of the problem with different resolution. As we can see, the more points are used in the the plot, the more precision is obtained.

3 | Sequential Gauss-Seidel method

The Gauss-Seidel algorithm represents a variation of the Jacobi method, where the memory usage is reduced by using a single grid point matrix. In this implementation the update is made using values of the grid previously calculated in the same iteration. This feature, makes the code difficult to parallelize, as we are going to discuss in section 6 of this report, where we present a different approach to the problem using the Gauss-Siedel method.

```
1  #include <stdio.h>
2
3  void gauss (double * u, double * f, double lambda,
4              int N, int kmax, double treshold, int * k) {
5
6    double d = treshold + 1;
7    int M = N + 2;
8    double lambda2 = lambda * lambda;
9    double T;
10
11   for (*k = 0; (*k < kmax && d > treshold); (*k)++){
12     d = 0;
13     for (int i = 1; i < N+1; i++) {
14       for (int j = 1; j < N+1; j++) {
15         T = 0.25 * ( u[(i-1)*M + j] + u[(i+1)*M + j] + u[i*M + (j-1)] +
16                      u[i*M + (j+1)] + lambda2 * f[i*M + j] );
17
18         d += (T - u[i*M + j]) * (T - u[i*M + j]);
19         u[i*M+j] = T;
20       }
21     }
22   }
23 }
```

We tested the code varying the resolution of the grid, as we can see in table 3.1.

	Jacobi			Gauss-Seidel		
N	iterations	time (s)	$\frac{\text{iterations}}{\text{s}}$	iterations	time	$\frac{\text{iterations}}{\text{s}}$
16	285	0.000099	2887383.2	163	0.000206	790371.7
32	926	0.000951	974147.4	539	0.00351	153665.7
64	3001	0.0113	266145.9	1795	0.0515	34869.2
128	9484	0.154	61412.6	5905	0.709	8325.9
256	28370	1.833	15475.9	18810	9.24	2035.8
512	76141	25.23	3017.6	56496	112.25	503.3
1024	159532	297.84	535.6	151947	1214.7	125.1

Table 3.1: Comparison between the Jacobi and Gauss-Siedel implementations, with optimization options enabled in the compiler.

The comparison plot between Jacobi and Gauss-Seidel can be seen in figure 3.1.
The plot shows that, even though the memory usage is optimized in the Gauss-Seidel implementation, the performance seems to be worse. We believe that the compiler is able to apply more efficient optimization techniques for the Jacobi method, since we read from and write to different matrices (and therefore memory locations in the cache), as opposed to the Gauss-Siedel method, that reads and writes to the same memory location.

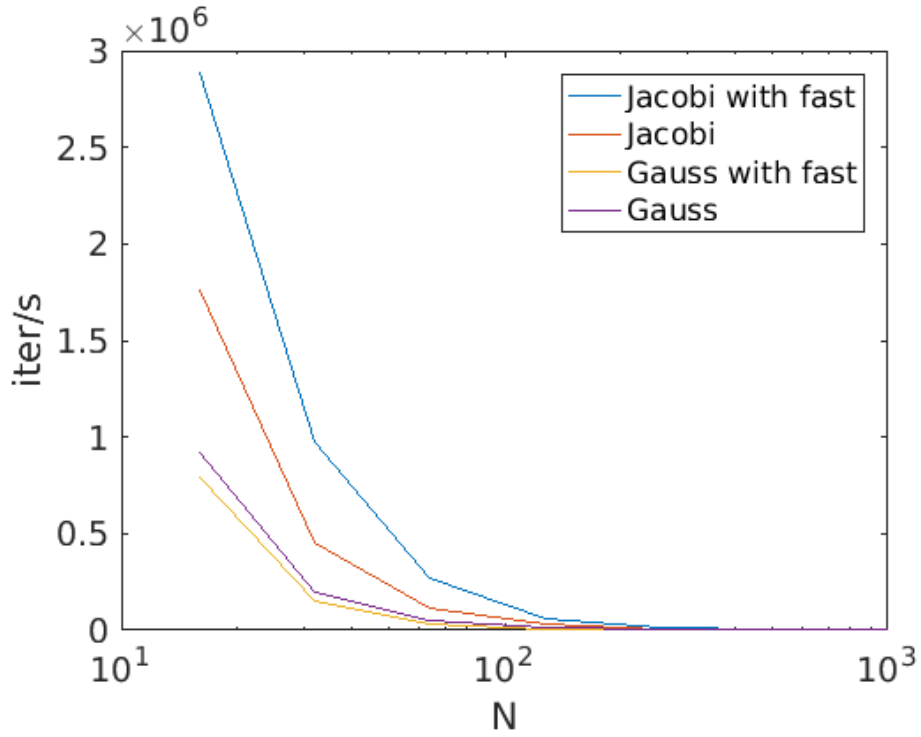


Figure 3.1: Comparison between Jacobi and Gauss, with different options enabled.

Consequently we decided to compile and run again both methods disabling the optimization flags. The results are shown in table 3.1.

We can see that the Jacobi version is still faster, even after disabling the optimization flag in the compiler. However Gauss-Seidel algorithm seems to converge with a lower number

N	Jacobi			Gauss-Seidel		
	iterations	time (s)	$\frac{\text{iterations}}{\text{s}}$	iterations	time	$\frac{\text{iterations}}{\text{s}}$
16	285	0.000162	1763092.4	163	0.000178	915223.0
32	926	0.00204	453835.7	539	0.00269	200632.8
64	3001	0.0260	115612.7	1795	0.0368	48754.6
128	9484	0.329	28865.4	5905	0.496	11899.4
256	28370	4.053	6999.0	18810	6.371	2952.3
512	76141	46.52	1636.7	56496	77.76	726.5
1024	159532	378.67	421.3	151947	836.43	181.7

Table 3.2: Comparison between the Jacobi and Gauss-Siedel implementations, without optimization options enabled.

of iterations. Its better usage of memory makes this algorithm more appropriate for really large problem sizes.

4 | OpenMP Jacobi method

In these chapter we are going to study three different parallel versions of the Jacobi function.

4.1 First parallel version

We parallelized the code by placing a OpenMP directive inside the outer for loop that manages the number of iterations:

```
1  ...
2  void jacobi(double * unew, double * uold, double * f,
3              double lambda, int N, int kmax, double treshold, int * k){
4
5      double lambda2 = lambda*lambda;
6      int M = N+2;
7      int i,j;
8      double d = treshold+1;
9      double * swapper;
10
11     for (*k = 0; (*k < kmax && d > treshold); (*k)++){
12       d = 0;
13       #pragma omp parallel for reduction(+: d)
14       for (i = 1; i < N+1; i++) {
15         for (j = 1; j < N+1; j++) {
16           unew[i*M+j] = ( 0.25*(uold[(i-1)*M+j]+uold[(i+1)*M+j]+
17                         uold[i*M+j-1]+uold[i*M+j+1]+lambda2*f[i*M+j]) );
18           d += (unew[i*M+j]-uold[i*M+j])*(unew[i*M+j]-uold[i*M+j]);
19         }
20       }
21       swapper = uold;
22       uold = unew;
23       unew = swapper;
24     }
25
```

```
26      swapper = uold;
27      uold = unew;
28      unew = swapper;
29  }
```

When testing this implementation by varying the number of threads and the size of the grid, we found out that with small values of `N` the parallel version does not scale. In fact, it seems that the sequential version is faster. However, as soon as the size of the grid increases, the wall time becomes slightly lower when adding more threads (figure 4.1).

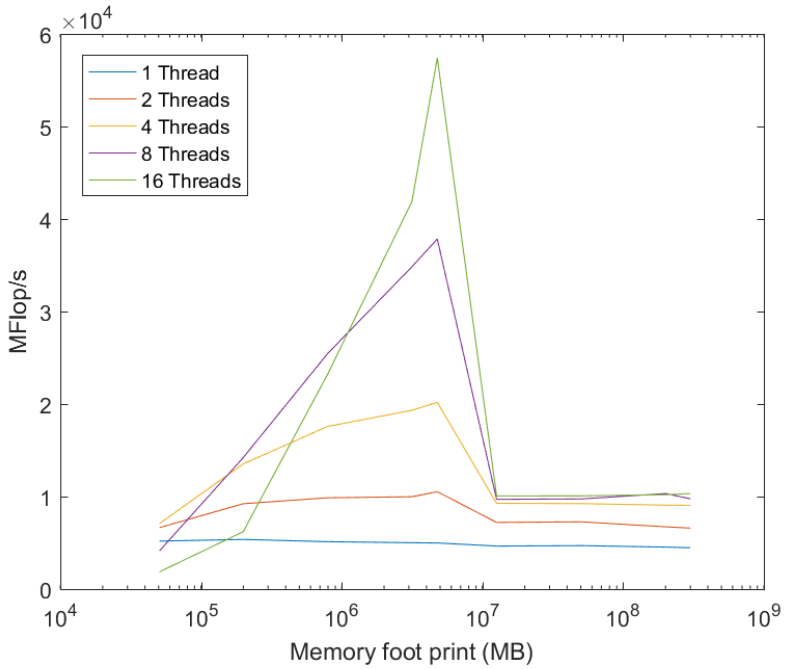


Figure 4.1: Memory against speed for different number of threads.

The program has been tested using 10000 points of resolution (N), **100** iterations maximum and a threshold of **0.001** as stopping criteria.

Num. cores allocated	Elapsed time	$\frac{\text{iterations}}{\text{s}}$
1	22.422842	4.459738
2	15.179183	6.587970
4	11.050148	9.049653
8	10.260929	9.745707
16	9.690500	10.319384

Table 4.1: Benchmarks on the first parallel version: this version slightly scale, but it is still quite inefficient.

In this implementation the outer loop contains the parallel regions and thus, the workers are created and destroyed in every iteration.

4.2 Second parallel version

In this second version we made use of the collapse clause. This function allows to specify how many loops can be turned into a large iteration space.

```
1   ...
2     for (*k = 0; (*k < kmax && d > treshold); (*k)++){
3       d = 0;
4       #pragma omp parallel for collapse(2) reduction(+: d)
5       for (i = 1; i < N+1; i++) {
6         for (j = 1; j < N+1; j++) {
7           unew[i*M+j] = ( 0.25*(uold[(i-1)*M+j]+uold[(i+1)*M+j]+
8                         uold[i*M+j-1]+uold[i*M+j+1]+lambda2*f[i*M+j]) );
9           d += (unew[i*M+j]-uold[i*M+j])*(unew[i*M+j]-uold[i*M+j]);
10        }
11      }
12      swapper = uold;
13      uold = unew;
14      unew = swapper;
15    }
16  ...
```

As shown in figure 4.2 this functionality has a negative effect in the performance of the code compared to the first version in figure 4.1 (makes the execution 3 times slower) and thus we decided to discard it from our final implementation.

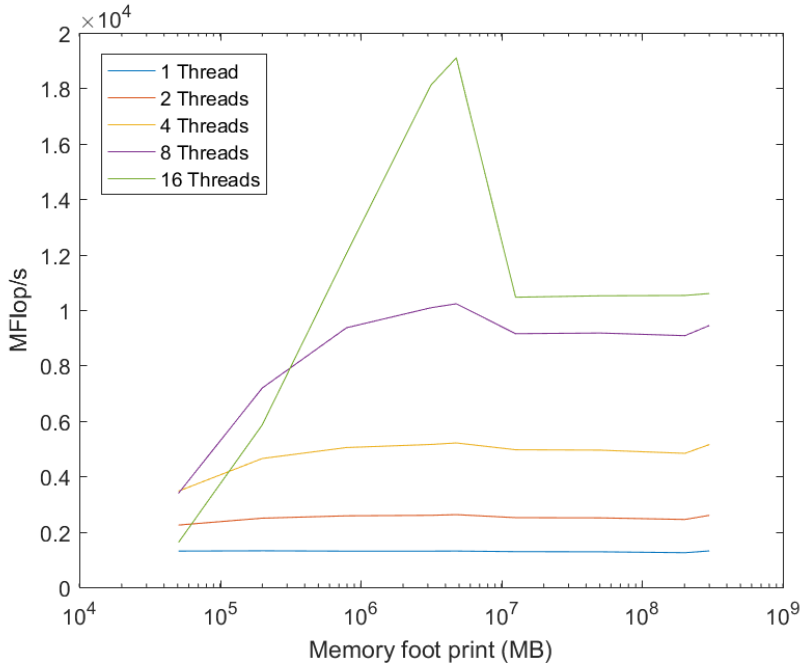


Figure 4.2: Memory against speed for different number of threads

4.3 Third parallel version

It is known that creating and destroying workers is time consuming and that is why our third improvement is focused on trying to create threads just once before entering the iterative loop.

In this version we included a single thread region where just one thread updates the values of the number of iterations and the grid point difference, while the rest are put on hold or are spinning.

For this reason, we had to substitute the `for` outer loop with a `while` loop, in order to have a finer control on the cycle of operations.

As shown in the code snippet below, in a initial implementation we included a barrier before the variables `k` and `d` were set for the next iteration, so as two prevent the two variables to be modified before all the threads enter the first `for` loop:

```
1  ...
2
3      #pragma omp parallel shared(k, unew, uold, f, lambda2, M)
4      {
5          while (*k < kmax && diff >= treshold) {
6
7            #pragma omp barrier
8
9            #pragma omp single
10           {
11             diff = 0;
12             (*k)++;
13           }
14
15           #pragma omp for reduction(+: diff)
16             for (int i = 1; i < N+1; i++) {
17               for (int j = 1; j < N+1; j++) {
18                 unew[i*M+j] = ( 0.25*(uold[(i-1)*M+j]+uold[(i+1)*M+j]+
19                                 uold[i*M+j-1]+uold[i*M+j+1]+lambda2*f[i*M+j]) );
20                 diff += (unew[i*M+j]-uold[i*M+j])*(unew[i*M+j]-uold[i*M+j]);
21               }
22             } // Implicit barrier
23
24           #pragma omp single
25           {
26             swapper = uold;
27             uold = unew;
28             unew = swapper;
29           } // Implicit barrier
30       }
31   }
32  ...
```

However, this slowed down the execution and we realized that we could create a third variable to store the value of the squared difference.

The version without explicit barriers is the following:

```
1      ...
2      double diff = 0, d = treshold+1;
3      *k = 0;
```

```
4
5       #pragma omp parallel shared(k, unew, uold, f, lambda2, M)
6       {
7           while (*k < kmax && d >= treshold) {
8
9             #pragma omp for collapse(2) reduction(+: diff)
10              for (int i = 1; i < N+1; i++) {
11                for (int j = 1; j < N+1; j++) {
12                  unew[i*M+j] = ( 0.25*(uold[(i-1)*M+j]+uold[(i+1)*M+j]+
13                                        uold[i*M+j-1]+uold[i*M+j+1]+lambda2*f[i*M+j]) );
14                  diff += (unew[i*M+j]-uold[i*M+j])*(unew[i*M+j]-uold[i*M+j]);
15                }
16              } // implicit barrier here
17
18              #pragma omp single
19              {
20                d = diff;
21                (*k)++;
22
23                swapper = uold;
24                uold = unew;
25                unew = swapper;
26              } // implicit barrier here
27        }
28      }
29  ...
```

Unfortunately, we cannot use a `nowait` clause in the for loop, since we need all threads to finish to read from the `uold` matrix and write on the `unew` matrix, before swapping the matrices pointers in the `single` block.

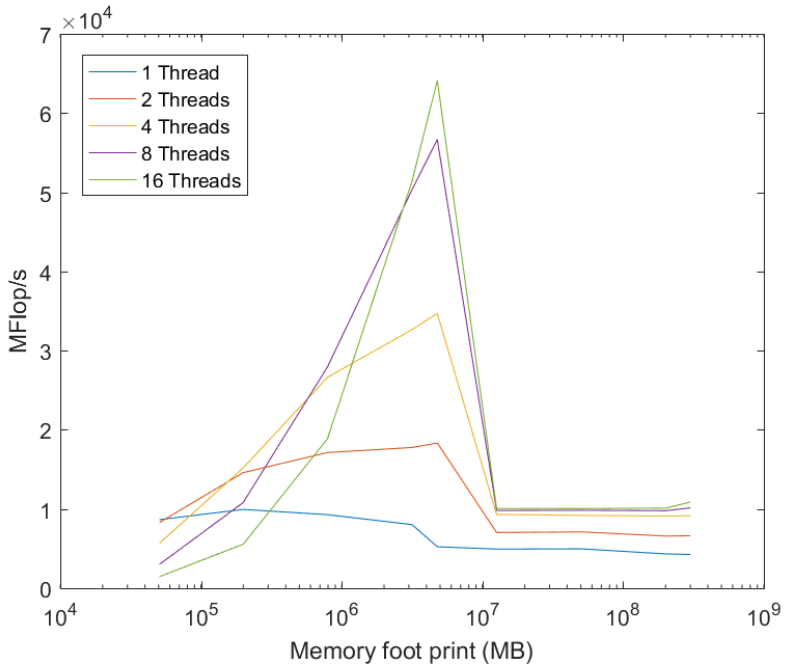


Figure 4.3: Memory against speed using different number of threads, with the version introduced in this section.

The results using different problem sizes and different number of threads are shown in figure 4.3. If we compare these results with the ones obtained in the previous versions

using 8 threads (figure 4.4) we see that there is an obvious increase in performance for the middle range values of memory, but as we approximate to larger problem sizes the results are practically the same.

This means that for large arrays the time spent creating and destroying the threads in each iteration is insignificant compared to the time used to compute the values of the array.

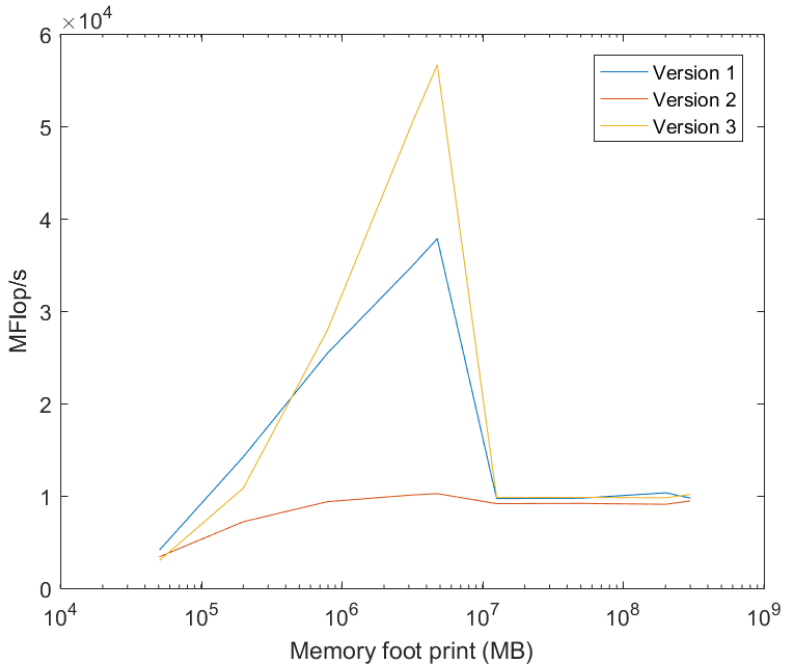


Figure 4.4: Memory against speed using **8** threads, and the 3 different versions presented in this report.

We also compared the performance switching on and off the compiler optimization tools. Notice that the compiler automathically enables the -xO3 flag to support parallelization. The results are shown in figure 4.5 were it is possible to see a slight decrease in performance for the -xO3.

Finally we compared the three versions to Amdahl's law (figure 4.6). The black dashed lines represent the theoretical bounds when 70% and 60% of the code is parallelizable. We see that the code scales well for a small number of processor but it fails to follow the upper bound when the number increases. In this case, either 4 our 8 threads are optimal options for both version 1 and version 3.

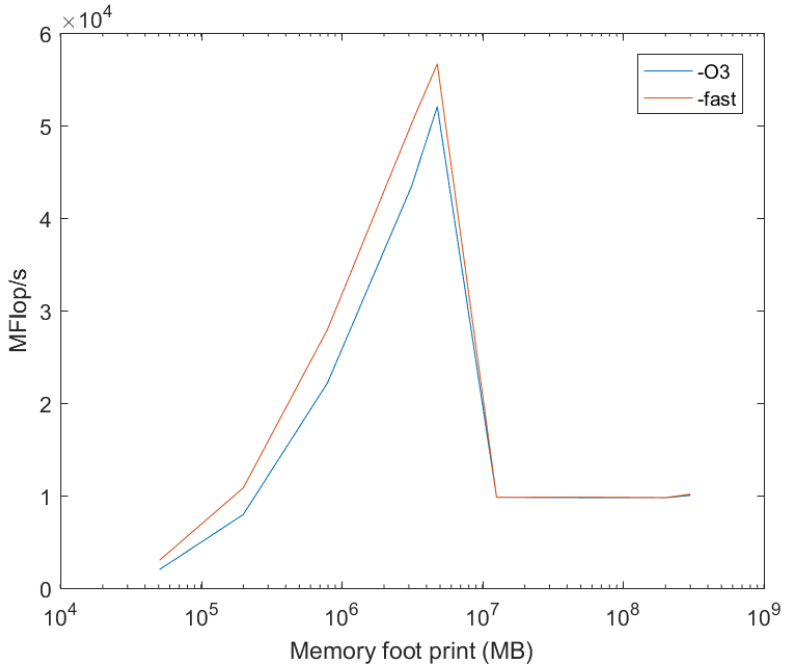


Figure 4.5: Memory against speed using **8** threads, and optimization tools on and off

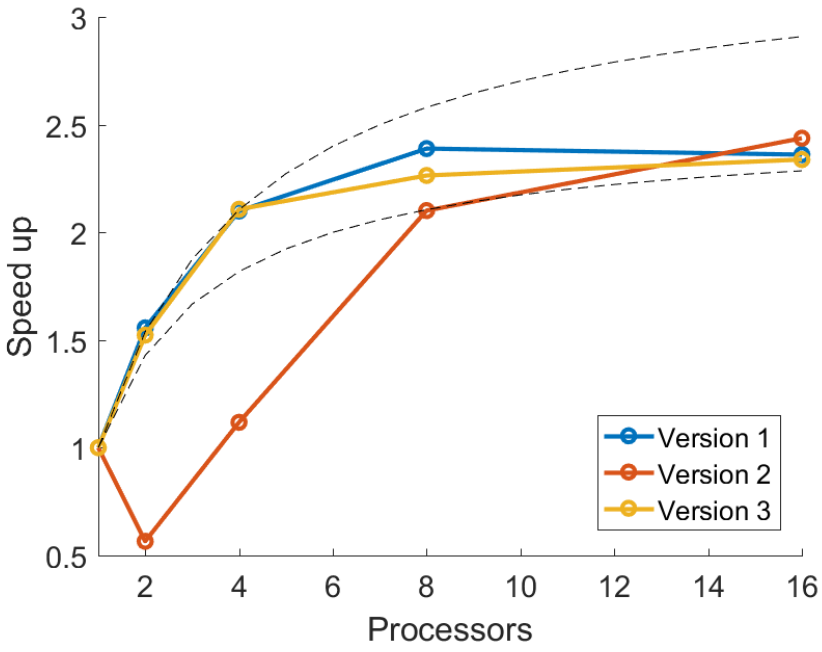


Figure 4.6: Number of processors vs speed up. The speed-up is calculated as $\frac{T(P)}{T(1)}$.

4.4 Analysis with the Sun Studio performance Analyzer

Using the Sun Studio Performance Analyzer we studied the performance of some of the solution proposed, especially the one presented in section 4.1, and two 4.3 versions, the first with an explicit barrier and the second without.

To be able to record and monitor the time spent in an idle phase or at a barrier, we had to set the OMP flag `OMP_WAIT_POLICY=active` at the submission of the job.

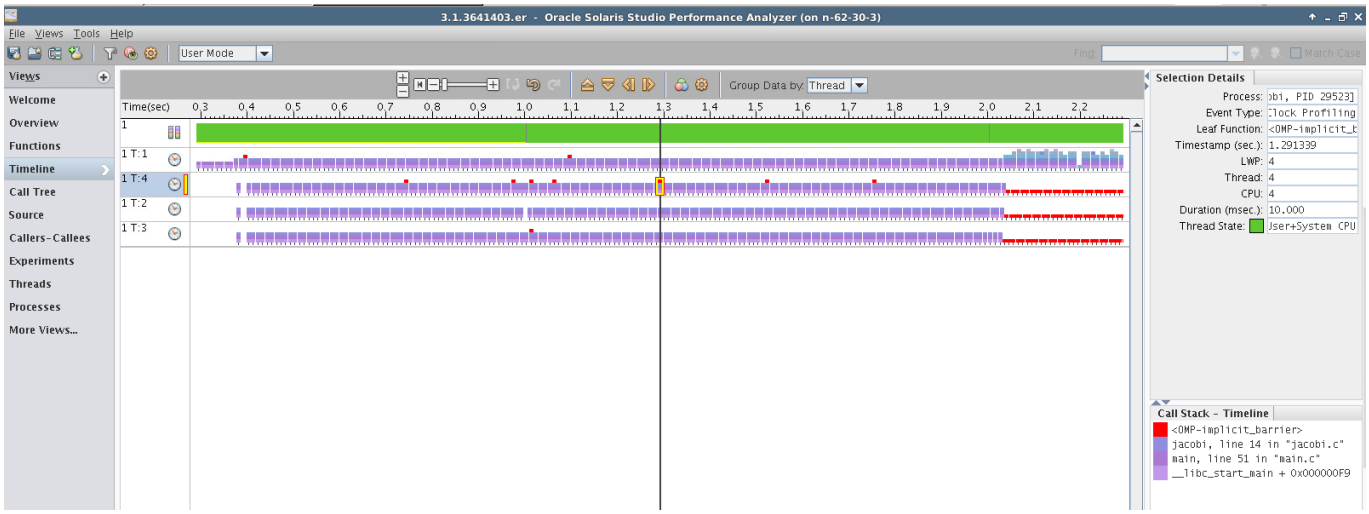


Figure 4.7: Timeline analysis on the first parallel version.

The red dots in figure 4.7 represent the time the threads are waiting for the others to perform an operation. The red dots at the right represent time spent writing to disk the resulting matrices inside the main function (we used it as a debug), so it is not correlated with the Jacobi function.

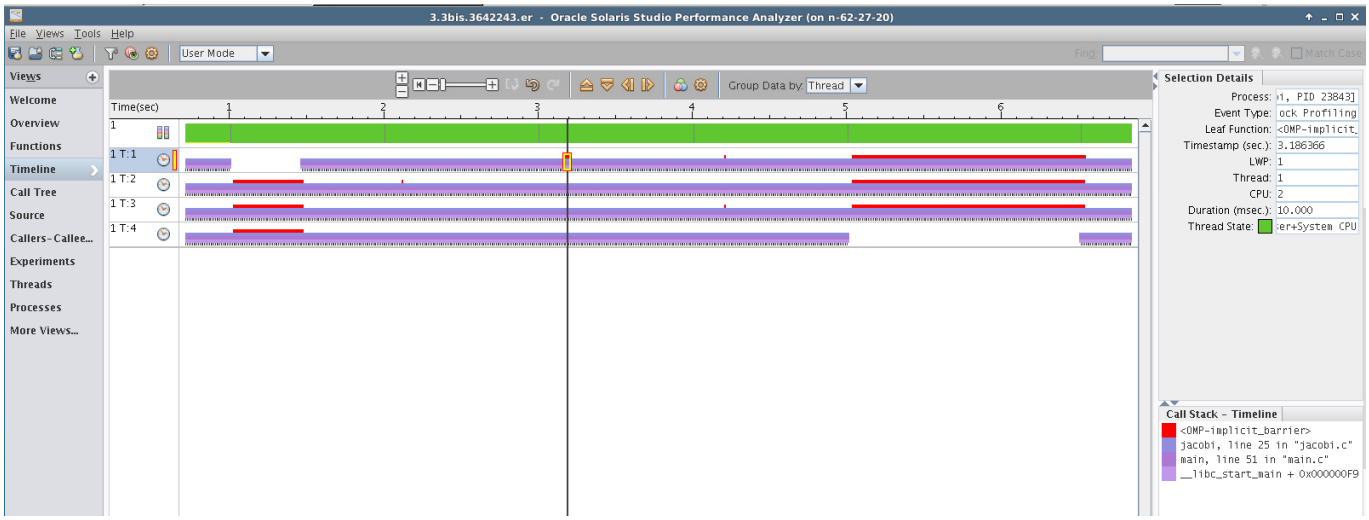


Figure 4.8: Timeline analysis on the third parallel version, using an explicit barrier.

We can see in figure 4.8 that the use of explicit barriers puts three of the four threads on

hold for long periods while only one is working.

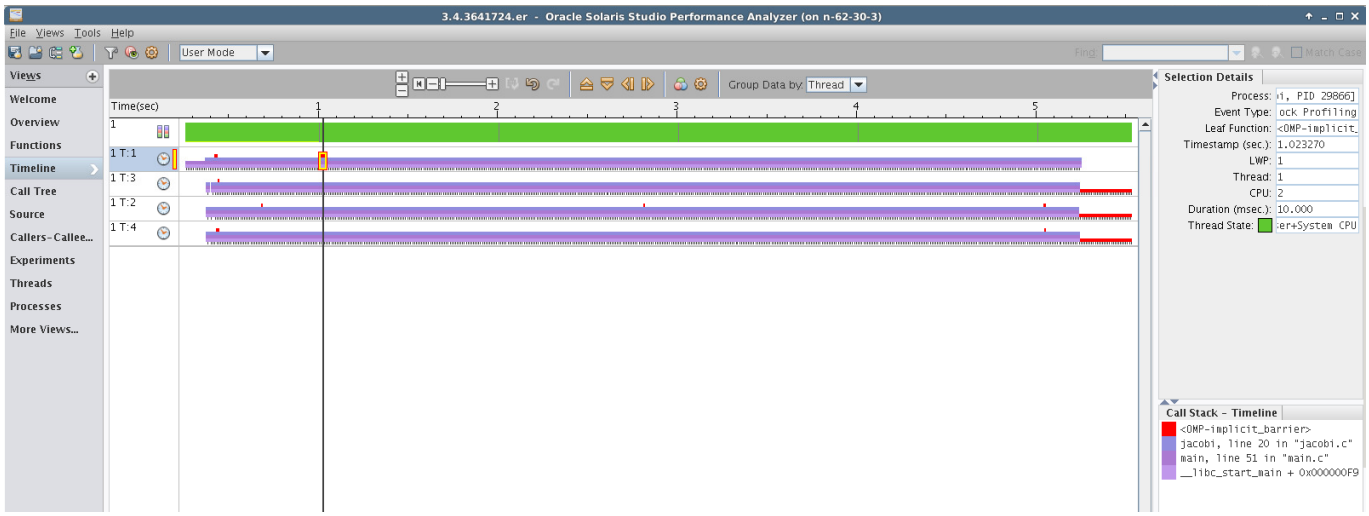


Figure 4.9: Timeline analysis on the third parallel version, without explicit barriers.

Figure 4.9 reveals the improvement, mentioned in section 4.3, of adding a new variable to store the squared difference. In this final implementation there is no need of explicit barriers and thus all workers can work simultaneously almost all the time.

5 | Comparison with Mandelbrot

To compare the scalability of the OpenMP Jacobi method with the Mandelbrot function, we used $N = 4096$ in the Jacobi method. As the Jacobi uses three matrices (u_{new}, u_{old} and f) made of doubles, with a size of $(N+2)^2$ each, this produces a memory footprint of[1]:

$$3 \cdot (4096 + 2)^2 \cdot \text{sizeof}(double) = 403 \text{ MB} \tag{5.1}$$

The Mandelbrot allocates a single int matrix of size N^2; therefore to get a similar memory footprint:

$$N^2 \cdot \text{sizeof}(int) = 403 \text{ MB} \Leftrightarrow \tag{5.2}$$

$$N = \sqrt{\frac{403 \text{ MB}}{\text{sizeof}(int)}} \approx 10038 \tag{5.3}$$

The memory footprint of both is really close. The Mandelbrot almost follows the line of $f = 1$ for the tested domain, while the Poisson implementation stops scaling well at around 8 processors.

[1] using 8 bytes as the size of a double and 4 as the size of a int.

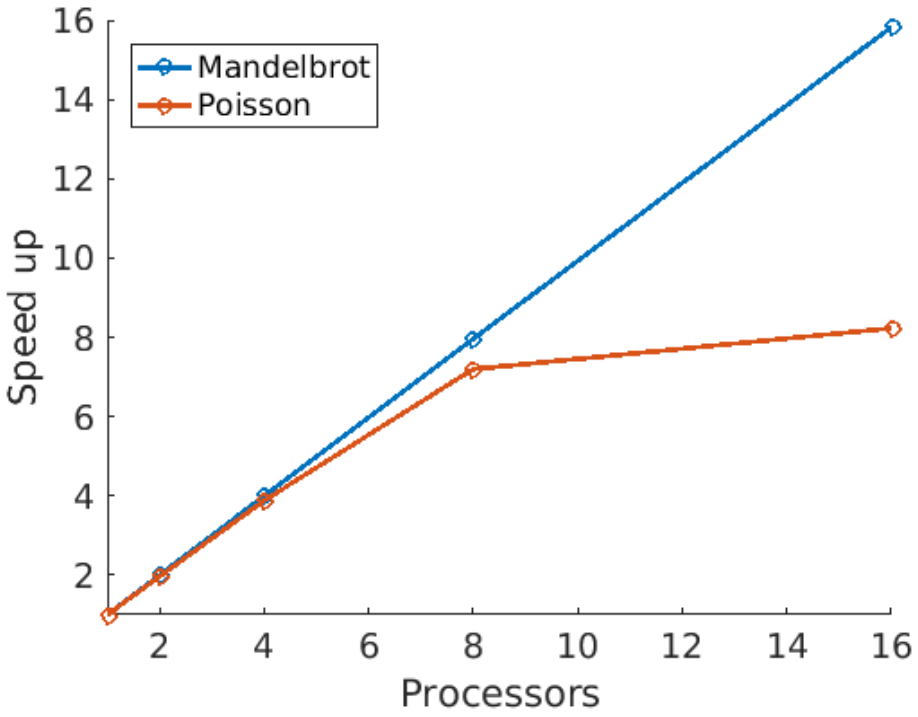


Figure 5.1: Number of processors vs speed up, using the final version of the Jacobi OpenMP implementation and the OpenMP implementation of Mandelbrot.

Therefore, and according to the results shown in figure 5.1, we can conclude that the best number of threads to use in the Poisson problem is around 8, whereas for the Mandelbrot problem this number can be as big as possible since the problem scales linearly.

6 | Parallel Gauss-Seidel method

Each iteration of the two inner loop of the regular Gauss-Seidel method is dependent on the four closest cells. The code can be be parallelized by splitting it into a sequence of two double loops, such that the iterations becomes independent.

Consider figure 6.1 as a small example of the problem with $N = 6$ (the gray area is the room wall area, which is never updated). By splitting the cells into a checkered grid, we can see that all the white cells are updated only using the black and the *wall* cells, while the black cells are only updated using the white and *wall* cells.

The algorithm can then be implemented by having and outer for loop testing the number of iterations and the convergence, as we already did for the preceding algorithms. Inside it, a parallel double loop runs and updates the white cells. After the white cells are updated and the threads are synchronized, another parallel double loop updates the black cells.[1]

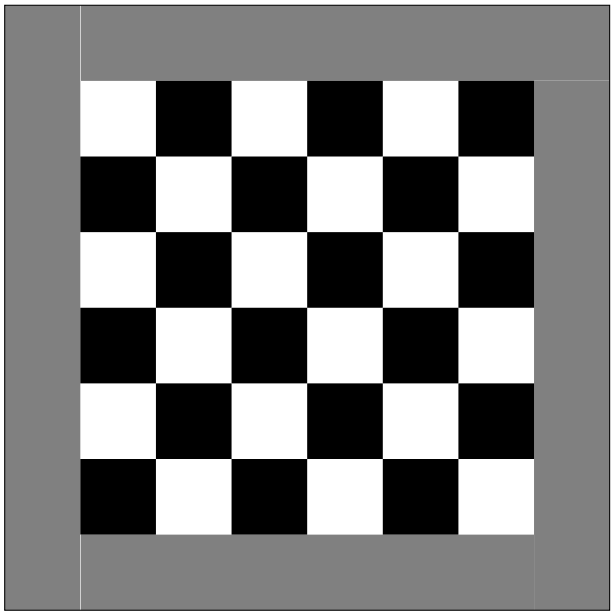


Figure 6.1: Sketch of a parallel Gauss-Seidel method implementation.
The method is parallelized by alternating the update of the black and white cells until reaching the convergence, as the updating of the white cells solely depends on the black cells and vice versa.
The grey border represents the wall cells which are never updated, but are read from when updating the inner cells.

[1] David Bindel, Cornell University, `http://www.cs.cornell.edu/~bindel/class/cs5220-s10/slides/lec14.pdf`