Technical University of Denmark

High-Performance Computing

Course 02614

Assignment 2

Authors:
Oskar Hint, s161559
Mikkel Jensen, s123184
Philip Rasmussen, s103124

January 13, 2017

Contents

1 Introduction

The assignment seeks to solve the Poisson equation in two dimensions and the specific stated steady state heat problem of the heat distribution in a square room with a known boundary conditions. Taking this information, the goal of this assignment is to implement sequential versions of two iterative PDE solvers, namely the Jacobi method and the Gauss-Seidel method, which are both multi grid smoothers, that can be used to find the solution to the Poisson problem. The Jacobi method is parallelized using openMP and the scalability is compared to the parallelization of the calculation of the Mandelbrot set. All of the mentioned implimentations are then analyzed for different parameters and we seek to determine the prospects and limitations of the different approaches. Finally a method for parallelizing the Gauss-Seidel method is discussed but not implemented.

1.1 Computing Specifications

Unless otherwise stated a lot of the specifications are fixed throughout the report. The code is written in C and compiled with the Sun Studio compiler. All the sequential experiments are run on the hpcintro queue on a single node with one processors per node (ppn), which is specified in the submit.sh script. The details of the parallel experiments will be specified on the go. All the nodes on the cluster are identical, and the full specifications are given in table 1.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	20
On-line CPU(s) list:	0-19
Thread(s) per core:	1
Core(s) per socket:	10
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Stepping:	2
CPU MHz:	2601.000
BogoMIPS:	5187.68
Virtualization:	VT-x
L1d cache:	32kB
L1i cache:	32kB
L2 cache:	256kB
L3 cache:	25600kB
NUMA node0 CPU(s):	0-9
NUMA node1 CPU(s):	10-19

Table 1: Table of the full machine specifications.

2 Assignment

The two sequential methods are implemented as separate functions that are called from within a while loop in the main file. The while loop runs as long as the checksum is larger than the threshold, d, AND while the number of iterations, k is smaller than a user-specified $k_{\max}$:

```
while(checksum > d && k < kmax)
```

The checksum is defined as the Frobenius norm of the difference between the updated matrix and the previous version of the matrix. The norm is

$$||u - u_O||_F =\equiv \sqrt{\sum_i \sum_j (u_{i,j} - u_{O,i,j})^2}. \tag{1}$$

The functions are implemented in a straight-forward way with a double for loop. The input variables are the new matrix, u, the old matrix, uo, the source matrix, f, the size of the matrices, N, and the grid spacing squared Δ^2. The loops run from 1 to $N-2$ such that the edge of the matrices are not updated.

```
void jacobi_seq(double ** u, double ** uo, double ** f, int N,
    double delta2){

  int i,j;
  for(i = 1; i < N-1; i++){
    for(j = 1; j < N-1; j++){
      u[i][j] = 0.25*(uo[i-1][j] + uo[i+1][j] + uo[i][j+1] +
          uo[i][j-1] + delta2*f[i][j]);
    }
  }
}
```

Listing 1: Implementation of the sequential Jacobi iterative process

```
void gauss_seidel(double ** u, double ** f, int N, double delta2){

  int i,j;
  for(i = 1; i < N-1; i++){
    for(j = 1; j < N-1; j++){
      u[i][j] = 0.25*(u[i-1][j] + u[i+1][j] + u[i][j-1] +
          u[i][j+1] + delta2*f[i][j]);
    }
  }
}
```

Listing 2: Implementation of the sequential Gauss-Seidel iterative process

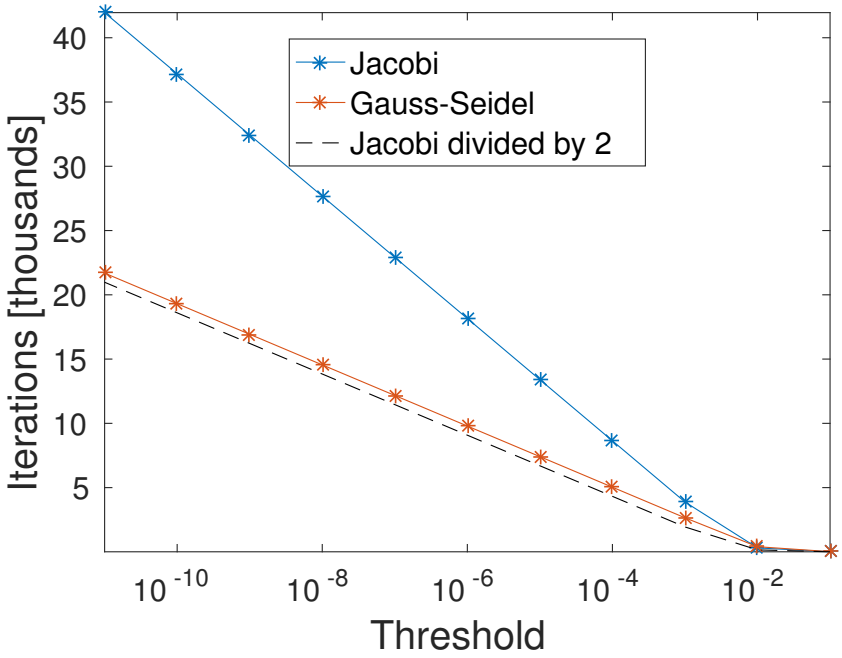

Figure 1: The convergence of the Jacobi method and the Gauss-Seidel method with changing threshold.

2.1 Comparing convergence

The two functions are run with a fixed size of $N = 100$, and $k_{\max}$ large enough that the clause that stops the while loop is always the convergence criterion. The threshold is varied from 10^{-1} to 10^{-11}, and the resulting number of iterations until convergence fro the two functions is seen in figure 1. The x-axis is logarithmic, so the behavior is exponential, and the Gauss-Seidel is approximately twice as fast as the Jacobi method. The black dashed line is Jacobi line divided by two, and it follows the Gauss-Seidel line with a slight off-set, but the slopes are identical.

The convergence is now investigated as a function of matrix size, N. For a constant threshold, the matrix size is varied, and the number of iterations required to converge is reported. The result can be seen in figure 2. We expected the initial increase to continue steadily for all matrix sizes, but as seen, the curves have a maximum depending on the threshold d. This can be explained by the fact that the Frobenius norm is the average norm of all points in the matrix. The first order central difference scheme only uses the next neighbors, and so, the change of u happens like a wave coming from the edges with the boundary conditions and the points where $f \neq 0$. This means that only the points where the wave is has notable differences between u and u_O. For example, the points in the center of the domain is initialized to zero, and they will remain that for approximately $N/2$ iterations before the change gets to the center. This implies that the norm of the center of the domain is zero, significantly lowering the total norm. After many iterations, the points near the sides of the domain have all converged, and the norm of these points will lower the total norm as well. This means that when the matrix size is increased, so is the number of points in the center which does not change in the beginning, and the norm is lowered to a point where the program thinks the

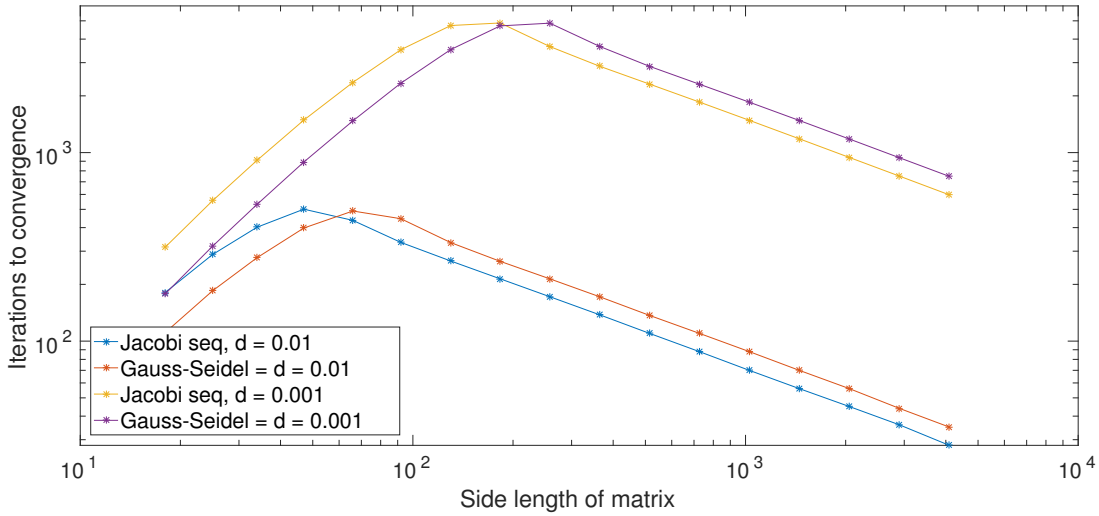

Figure 2: The convergence of the Jacobi method and the Gauss-Seidel method with changing matrix size.

problem has converged, but in reality it has not. For smaller thresholds, the size where the decrease in iterations happen will increase, but the general behavior is identical. This implies that when using the Frobenius norm, one must make sure that the threshold and the size are such that it is on the increasing part of figure 2 in order to be sure that the result is indeed the correct result. Alternatively, one could initialize the u and u_O arrays to random numbers, but that would require a lot of extra computations to be done. Another solution is to compare the change in each element with the threshold, but that requires an if-clause for each element, which also reduced performance significantly. Finally, it should be noted that the performance of the subsequent sections are still valid because it only compares timings and scaling behavior and not the result of the computation.

2.2 Comparing performance

The performance of the two sequential functions are analyzed by comparing the number of point updates per second. The program is forced to run for at least three seconds to get a reliable timing. The resulting graphs is shown in figure 3. For small matrices the Jacobi method is $\sim$ twice as fast as the Gauss-Seidel method. This is due to the fact that the Gauss-Seidel method read and write in the same array, which cause some memory "clogging" in the caches. For matrices exceeding the L3 cache, the Jacobi method slows down significantly, due to the smaller bandwidth to/from memory compared to the caches. The Gauss-Seidel method is reduced as well for the same reasons, but not nearly as significantly due to the memory clogging already taking place in the caches. However, as seen on figure 1, the Gauss-Seidel method requires half as many iterations to converge to the same precision, making it faster in wall clock time.

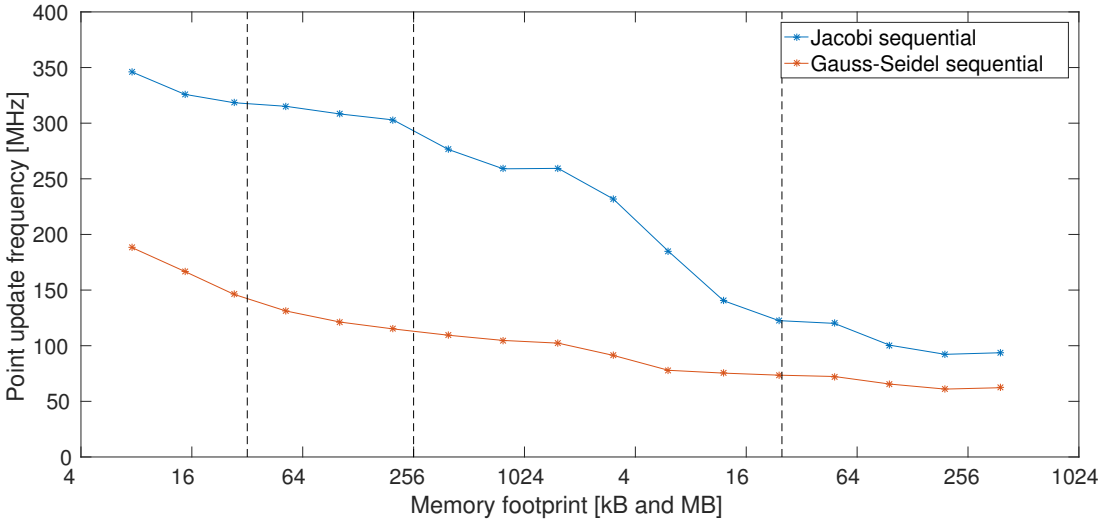

Figure 3: The performance of the two sequential functions with changing domain size. The black dashed lines represent the cache sizes of the CPU.

2.3 OpenMP of the Jacobi Method

The Jacobi method updates a new matrix from the values of an old one, and it is thus easily parallelizable. The program is paralellized using orphaning in openMP. The `#pragma omp parallel` is inside the while loop described above. That has the obvious disadvantage that the worker team will be created and destroyed with each iteration, creating a lot of overhead that will reduce performance. To measure and analyze the performance and scaling behavior of the paralized implementations, we map the speedup as a function of the number of processors/threads used. To investigate the speedup, the program is sent through the Sun Studio Performance Analyzer tool in sequential mode to see the fraction of time f spent in the parallelized region, to facilitate a comparison with Amdahl's law. All experiments in this subsection are run with $N = 10000$, $d = 0.01$ and $k_{max} = 10000$. In all cases and with 1 core, the wall time is ~ 16 s, verifying that our parallelization does not have any immediate negative effects on our program.

$$S(P) = \frac{T(1)}{T(P)} = \frac{1}{(f/P + 1 - f)} \tag{2}$$

S(P) is the speed up. T(1) is the sequential execution time, P is the number of processors, T(P) is the execution time on P processors and f is the parallized fraction of the code. The first implementation only parallelizes the Jacobi update loop. So we have a parallel region before the call to jacobi function and a pragma for in the function itself.

```
while(checksum > d && k < kmax){
    #pragma omp parallel default(none) shared(u,uo,f,N,delta2)
        private(i,j)
    {
      jacobi_seq(u,uo,f,N,delta2);
    } // end parallel
    checksum = fnorm_squared(u,uo,N);
```

```
    for(i = 0; i<N; i++){
      for(j = 0; j<N; j++){
        uo[i][j] = u[i][j];
      }
    }

    k++;
  }
```

Listing 3: While loop of our initial omp program

The speedup of the initial simple implementation is seen in figure 4. It is seen, that we experience some speed up on more processors, but it does stabilize quickly around 5 threads with a speed up factor of 1.15. The f value of 0.22 is found with the analyzer tool, and it is rather low and clearly much of computation time is not spent in parallel, and when we compare it to Amdahl's law, it is clearly not performing well, most likely due to parallel overhead.

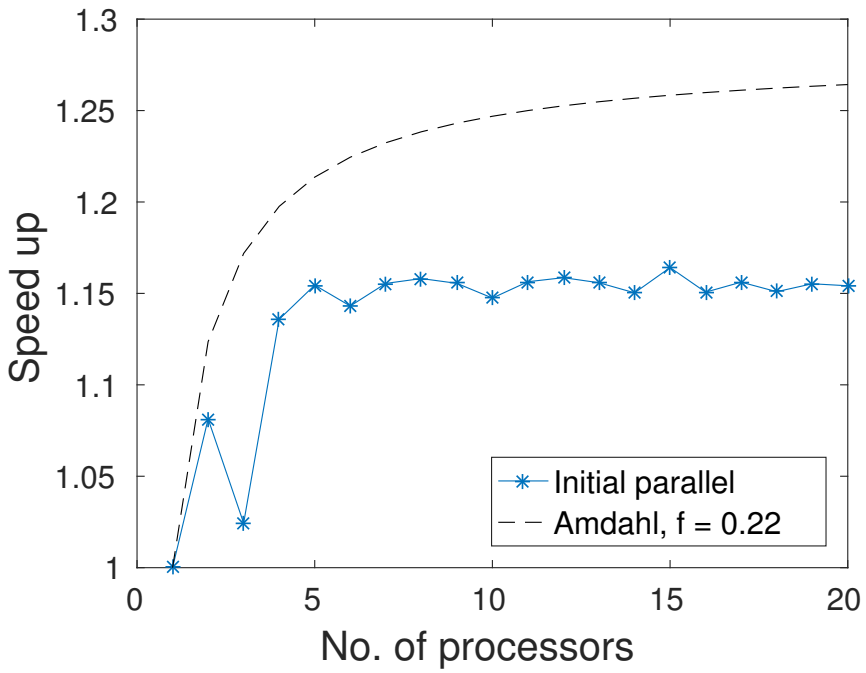

Figure 4: Scaling of the first parallelized version.

We now seek to improve the performance of our OpenMP program. First we realize, that the initial version is only running the Jacobi calculation in parallel, and that the program spent a lot of time on calculating the Frobenius norm and updating the old matrix, about 70 %. Furthermore, initializing the parallel region in each iteration is not optimal, so we seek to move the parallel region declaration outside the while loop. We created an additional parallel implementation called `omp2` that deals with these shortcomings. Our while loop for omp2 is as follows.

```
#pragma omp parallel default(none) shared (u,uo,f,N,delta2,
    checksum, k, d, kmax) private(i,j)
```

```
{
while(checksum > d && k < kmax){
    jacobi_seq(u,uo,f,N,delta2);
    #pragma omp for private(i,j)  reduction(+:checksum)
    for(i = 1; i <N-1; i++){
      for(j = 1; j<N-1; j++){
        checksum += (u[i][j]-uo[i][j])*(u[i][j]-uo[i][j]);
      }
    }
    #pragma omp for private(i,j)
    for(i = 0; i<N; i++){
      for(j = 0; j<N; j++){
        uo[i][j] = u[i][j];
      }
    }
    #pragma omp master
    {
    k++;
    checksum=checksum/(N*N);
    }
    #pragma omp barrier
} // end while

} // end parallel
```

Listing 4: While loop of our optimized (omp2) program

The specific changes are described in the following: The parallel initialization is moved outside the while loop with the shared variables u,uo,f,N,delta2, checksum, k, d and kmax. The Jacobi calculation is parallel using pragma omp for as in the simple version, the checksum is also parallel by using a omp for loop with reduction on checksum, so we do not experience any simultaneous update errors. The update to iteration count k and checksum normalization is inside a pragma omp master directive such that they are only updated once each iteration to ensure the same behavior as the sequential version. In the end we set a barrier to make sure, that the while loop is finished and the loop variables checksum and k updated before starting a new iteration.

We now end up with a much higher parallel fraction f of 0.93, again from the analyzer tool. In figure 5 the scaling behavior is seen for the more optimized version. The speed up is clearly much bigger stabilizing at 7 threads with a bit less than a factor 4. It does however not keep up with Amdahl's law, again due to parallel overhead, and the fact that memory initialization is done by the master thread, and used by all threads, which can reduce speedup when the number of cores exceed the number of cores per socket, which in this case is 10.

As a final step, we seek to optimize the program even more by implementing a parallelization of the memory allocation and initialization of the matrices to avoid remote memory access when using larger number of threads. This is only relevant when the number of processors exceeds 10 because then processors 10 and above will access data from the other socket. We added an if-clause that checks the number of threads and if it is above 10, runs the memory allocation in parallel. We also moved the initialization of the matrices to a parallel region and added pragma fors for the initialization loops.

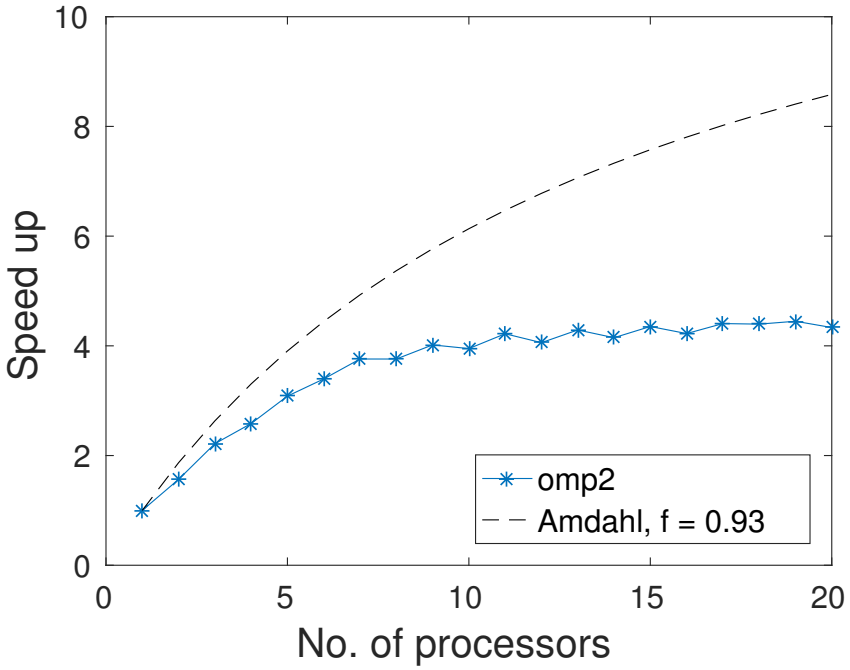

Figure 5: Scaling of the second parallelized version of the whole computational while loop.

The way we allocated memory to matrices initially is as follows:

```
double ** dmalloc_2d(int m, int n) {
  if (m <= 0 || n <= 0) return NULL;
  double **A = (double **)malloc(m * sizeof(double *));
  if (A == NULL) return NULL;
  A[0] = (double *)malloc(m*n*sizeof(double));
  if (A[0] == NULL) {
    free(A);
    return NULL;
  }
  int i;
  for (i = 1; i < m; i++)
    A[i] = A[0] + i * n;
  return A;
}
```

Listing 5: Initial memory allocation

With this approach the entire m*n array of matrix values is contiguous meaning everything is stored in a single location.

Since we want different threads to be able to store relevant rows of the matrices to memory locations close to them we need to make sure the calls to malloc are within a parallel region and workshared. The simplest way to do this is with a pragma omp for as follows:

```
double ** dmalloc_2d_opt(int m, int n) {
  if (m <= 0 || n <= 0) return NULL;
  double **A = (double **)malloc(m * sizeof(double *));
  if (A == NULL) return NULL;
  //A[0] = (double *)malloc(m*n*sizeof(double));
  #pragma omp parallel default(none) shared (A, m, n)
  {
    int i;
    #pragma omp for private(i)
    for (i = 0; i < m; i++)
      A[i] = (double *)malloc(n*sizeof(double));
  }
  return A;
}
```

Listing 6: Parallelized memory allocation

We call this version `omp3`. The speedup plot can be seen in figure 6. It should be noted that for this plot we fit a line for our speedup curve and computed the f value from the fitted line. Although there is a small dropoff at 19 and 20 threads, the observed f of 0.955 effectively represents the proportion of the problem that we managed to 'perfectly' parallelize.

As seen from figure 6 the increase in speedup continues above 10 processors, which was not the case with `omp2`.

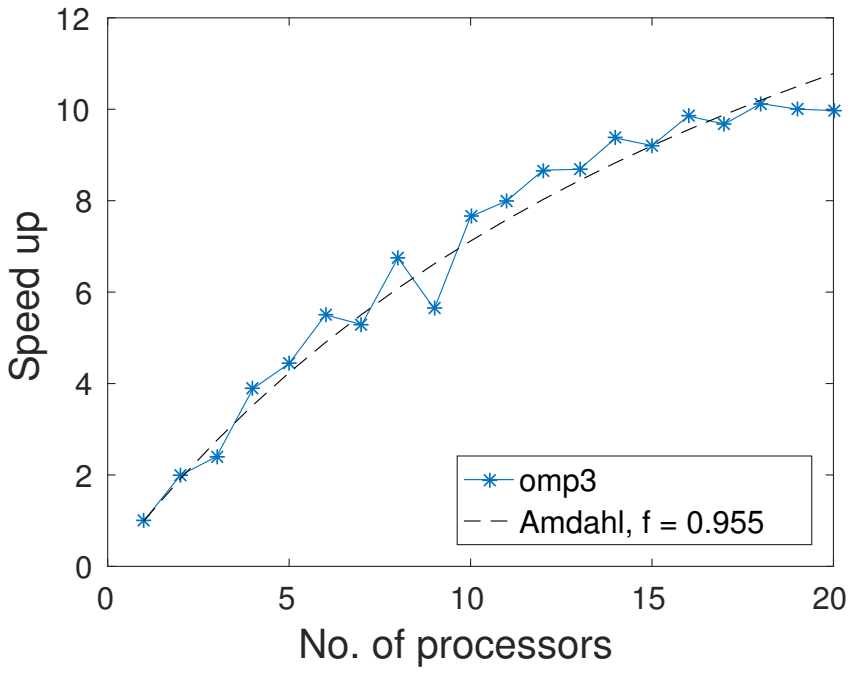

Figure 6: Speed-up scaling of the third parallelized version, which includes parallel memory allocation.

Lastly, we compare the efficiency of our three different parallel implementations. This is done by computing the efficiency, which is the speed up increase per added thread. Figure 7

clearly shows how different the efficiency curves are for the three different implementations. The simple implemenation clearly falls of right at the beginning, where the efficiency drops to a little higher than 50% already at thread nr. 2, which is obvious since the speed up factor is only slightly above 1. For the case of the more optimized implementation, it is clear that the omp3 version reigns supreme, and it does not reach 50% effiency before the 19th to 20th added thread, which makes sense, since we saw in the speed-up scaling that it started to stagnate at that point. Meanwhile the omp2 parallelization already reaches 50% around 7-8 threads.

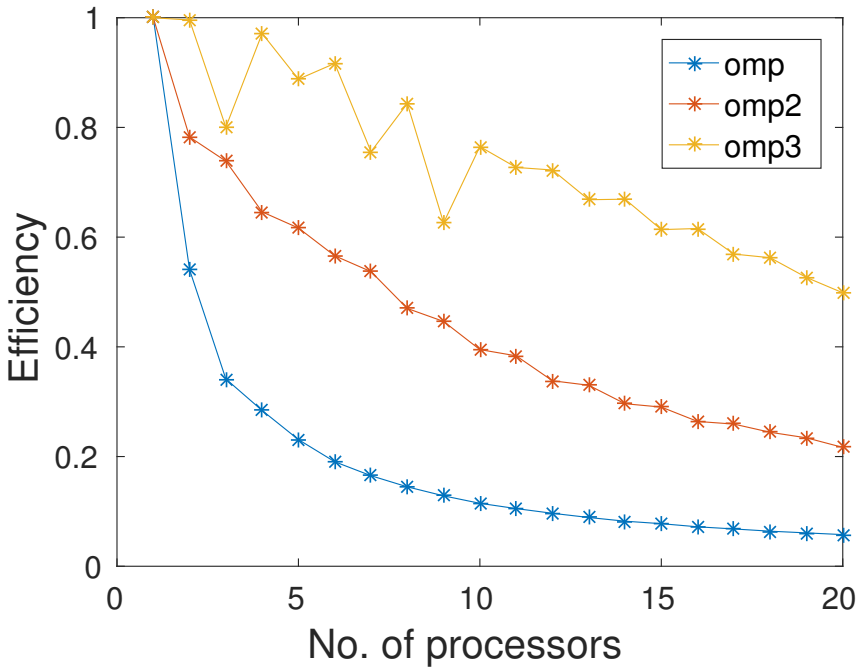

Figure 7: Effiency plot of the three parallel versions.

2.4 Speedup with varying matrix size

In this subsection, the speedup of the `omp3` parallelization is investigated with changing matrix sizes. The threshold level is chosen individually for each size such that the program runs for at least a few seconds. The resulting curves can be seen in figure 8. Despite the differing thresholds, the curves are still comparable due to the fact that the parallel region is outside the loop in `omp3`, which means that each iteration is identical, and so a larger number of iterations does not influence the speedup because it is relative to the case with one processor. The number of iterations for a single curve is constant for all number of processors. For a small number of processors, the speedup is similar, but as the number of processors increase, so does the parallel overhead required to keep track of the threads. This limits the speedup, and for a large number of processors, it even decreases for matrix sizes of $N = 500$ and $N = 1000$. For $N = 100$ the quickest is to use one core, simply because the overhead of even two cores takes more time than is gained by the parallelization. For $N = 10000$ the speedup is not seen to decrease because the matrix is so large that the speed gained from the parallelization is larger than the loss from the parallel overhead. If a larger

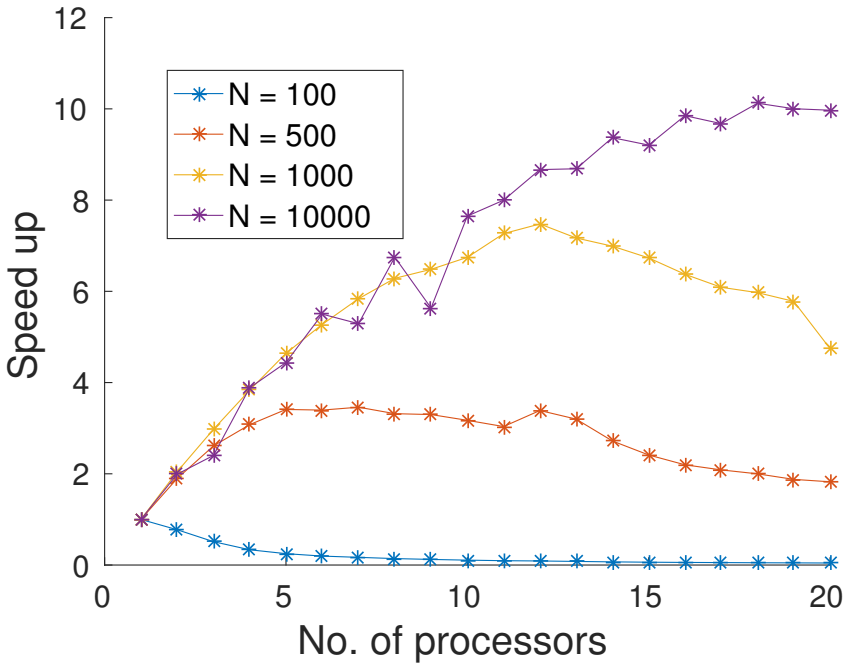

Figure 8: The speedup of the Jacobi iteration with the best parallellization method, `omp3`, with different matrix sizes.

number of cores was available, a decrease in speedup is expected. For $N = 100$, the analyzer tool was used to confirm our hypothesis: For 1 core, the program took ~ 2.5 s to run, but for 10 cores, the program spent 39 s on the implicit barriers after the for loops alone, i.e. a lot of work is done to keep track of the synchronization of the threads while the fraction of time where they are actually working is very small. And that is even when the parallel region is outside the main while loop, so the worker team is not destroyed on each iteration, but it is put to sleep and woken up many times, which also affect performance.

2.5 Speedup with different compiler option

We now investigate how different compiler options affect the parallelization. The options investigated are `-fast`, `-xO3`, `-xO4`, and `-xO5`. Optimization levels below 3 is not possible with openMP. The result is seen in figure 9. As can be seen, the wall times vary a bit, but generally speaking, they follow the same trend, irregardless of the compiler optimizations. This is most likely due to the fact that the openMP is processes before the optimizer compiler options. In order to be able to parallelize properly, the openMP outlines loops into functions, such that they can be passed to the worker team, but in turn that also means that the compiler will have a difficult time applying the most advanced optimization schemes as we saw in the exercise calculating pi.

2.6 Comparison with Mandelbrot program

In this subsection, the scaling of our best parallelized version of the Jacobi method, `omp3` is compared with the scaling of the computation of the Mandelbrot set. The scaling of the

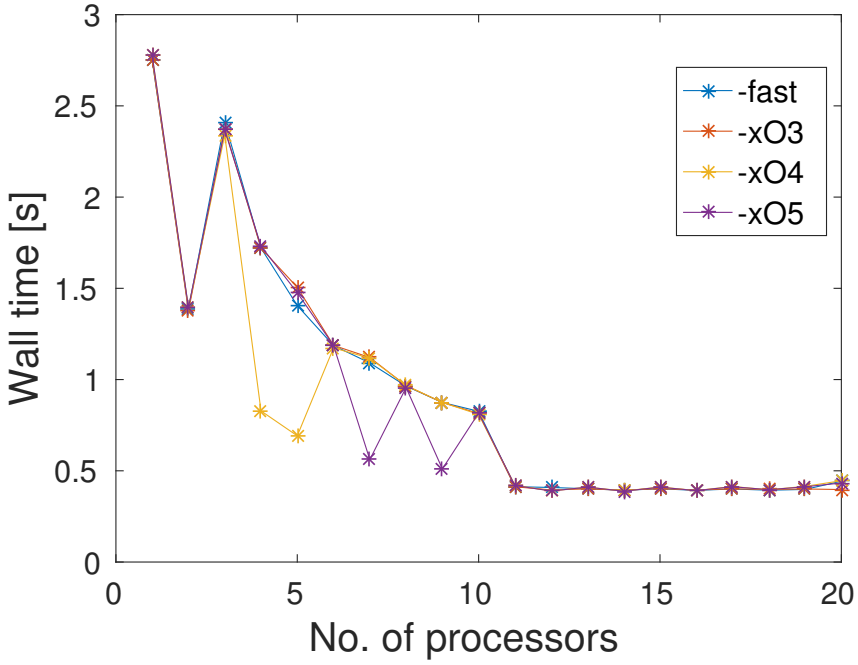

Figure 9: The wall times for different number of threads with different compiler optimizations activated.

Mandelbrot calculations with a direct implementation and a scheduled version with `static` with chunk size of 10. The results are shown in figure 10 with matrix sizes of 2600 such that in all cases the matrices will only fit in the memory and not the caches. The naive implementation of the Mandelbrot scales badly, especially for a few processors, and better for a larger number of processors. This is due to the fact that when the worker team hits the out for loop in the Mandelbrot program, the columns are divided evenly between the workers, but the work needed to be done is not spread out evenly across the columns. This causes a very imbalanced work distribution, and the resulting speedup is low, only about 50 % extra for 4 cores compared to 1. For a number of processors larger than 4, the slice of the image each worker gets is so small that more than one worker will work on the heavy section, which causes a speedup of about a factor 2 when going from 4 to 8 cores. However, this comes off a low starting point so it is not too impressive. To mitigate this problem, the worksharing was done with a static schedule with chunk size 10, such that each worker now does columns in chunks of 10, and this causes the work in the computationally heavy region to be spread on all involved processors, giving a near-perfect speedup, as is seen on figure 10. The finally optimized version of the Jacobi method follows the chunked Mandelbrot line when the number of processors are below 5, and then the speedup falls off, to reach a factor of 10 at 20 processors. The difference in speedup can be explained by the fact that the Mandelbrot parallelization is very simple, it only requires a single loop to be parallelized, while the Jacobi method needs quite some work and tricks in order to reach proper scaling. These many loops and opening and closing of parallel regions means that the overhead is much greater in the Jacobi exercise compared to the overhead in the Mandelbrot exercise, which explains why the speedups are similar for a few cores, and not for a large number of cores. The times for the sequential versions of the Mandelbrot and the parallel with 1 cores, was in both cases ~ 1.4 s without writing the image to the disc, verifying that the parallization is implemented

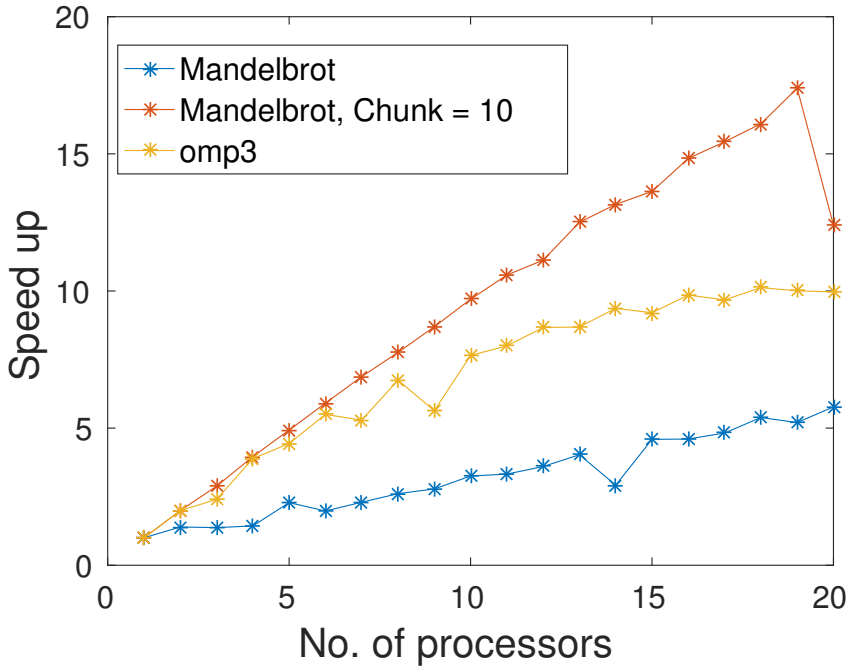

Figure 10: The speedup of the two the parallelization of the mandelbrot computation with the default scheduling, with a chunk size of 10 compared to the speedup of the Jacobi iteration.

correctly.

3 OpenMP Gauss-Seidel

The problem with the Gauss-Seidel smoother is, that it is built around internal data dependencies. It is therefore difficult to implement simple parallelized techniques, since the communication cost between the different threads would be really high. The Jacobi method on the other hand uses another matrix, the u old, which takes up memory space, but it is much easier to feed the data to the threads and finish the computation step. Since implementing a parallelization of the Gauss-Seidel smoothening is more difficult, the Jacobi method or alternatives have been more widely used for multigrid problems.

There does however exist modifications to the natural/lexographical version of the Gauss-Seidel smoother, which then seeks parallize using different methods, i.e. blocking. The article by D. Wallin, H. Löf, E. Hagersten and S. Holmgren[1] give insight to the complications of the subject and describe both the standard red-black method and their new temporally blocked natural Geiss-Seidel smoother. The completely natural Gauss-Seidel is a single sweep for each step/iteration, and therefore the data is interchanged. The red-black ordering is based on splitting each iteration step up in sweeps, which are fully data parallel. The grid is split into sets of even and odd gridpoints, even points have data dependencies to odd points and

[1]Dan Wallin, Henrik Löf, Erik Hagersten and Sverker Holmgren, *Multigrid and GaussSeidel Smoothers Revisited: Parallelization on Chip Multiprocessors*, ICS '06 Page 145-155, ACM NY USA, 2006-06-28

vice versa. However, the points do not have dependence to points of the same type. This means a loop over points of the same type can be done in parallel. In practice our initial approach would thus likely be to have a loop over one type of points, then a barrier and then a loop over points of the other type.

Wallin et al succeeded in improving the efficiency up to 40% compared to some other Gauss-Seidel parallelized smoothers, but it does however depend much on the system and problem at hand.

There exist several possible implementations of the Gauss-Seidel smoother, but it is an evolving field and there exist different approaches. The article by H. Courtecuisse and J. Allard[2] comes with different approaches to the same issue. They present a new parallel dense Gauss-Seidel algorithm, which is based on atomic update counter, which stores an integer counter in shared memory describing the processed blocks. This depends on the system used, but if it is possible on the system, then after an initial delay, the algorithm will get up to speed without facing the huge communication costs. The article further analyzes this algorithm in depth and its performance across different problem parameters and hardware systems.

4 Conclusion

In this report, a sequential implementation of the Jacobi iterative method and the Gauss-Seidel iterative method was implemented to solve Poisson's differential equation. The Gauss-Seidel method was found to converge twice as few iterations as the Jacobi method, but in turn the grid point updates per second was slower, but in total, the Gauss-Seidel method was found to be quicker in wall clock time. The Jacobi function can be easily parallelized, so three different versions with increasing amount of parallelization was created. The first only parallelized the loop updating each point, which gave a speedup of maximum 15 %. Secondly, the Frobenius norm checking and the updating of old matrix values was parallelized, as well as ensuring the worker threads are not created and killed needlessly. This increased the speedup to a factor of ~ 4. Finally, the memory allocation and initialization was parallellized, bringing the top speedup to a factor of 10, which corresponds nicely to $f = 0.955$ in Amdahl's law, suggesting that our best implementation has parallelized over 95 % of the problem. Using this last implementation, it was found that the optimal number of cores used depends on the matrix size due to increasing parallel overhead from an increasing number of cores. Finally, a method for parallelizing the Gauss-Seidel method is discussed, but not implemented.

[2]Hadrien Courtecuisse, Jérémie Allard, *Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors*, High Performance Computing and Communications, 1th IEEE Conference 2009, IEEE, 2009-06-25