D. Parallel Simulation of the Time Independent Diffusion Equation

D.1. Introduction

In many cases one is only interested in the final steady state concentration field and not so much in the transient behavior, i.e. the route towards the steady state is not relevant. This may be so because the diffusion is a very fast process in comparison with other processes in the system, and then one may neglect the transient behavior (an example is the diffusion limited aggregation that is presented as a case study at the end of this chapter).

Setting all time derivatives to zero in the diffusion equation (equation [4]) we find the time independent diffusion equation:

$$\nabla^2 c = 0 \,. \qquad [14]$$

This is the famous *Laplace equation* that is encountered in many places in science and engineering, and is therefore very relevant to study in some more detail.

Again consider the two-dimensional domain and the discretisation of the previous section. The concentration c no longer depends on the time variable, so we denote the concentration on the grid point here as $c_{l,m}$. Substituting the finite difference formulation for the spatial derivatives (Equation [10]) into Equation [14] results in

$$c_{l,m} = \frac{1}{4}\left[c_{l+1,m} + c_{l-1,m} + c_{l,m+1} + c_{l,m-1}\right] \,,\ \forall (l,m) \quad . \qquad [14]$$

Equation [14] contains a set of $(L+1)\times(L+1)$ linear equations with $(L+1)\times(L+1)$ unknowns. The unknowns are the value of the concentration of all grid points. It is our task to solve this set of equations. These equation can be reformulated as

$$\mathbf{A}\mathbf{x} = \mathbf{b} \,, \qquad [15]$$

where $\mathbf{A}$ is a $(N\times N)$ matrix and $\mathbf{x}$ and $\mathbf{b}$ are $(N\times 1)$ vectors. As an example, let us derive an explicit formulation for $\mathbf{A}$ in the one-dimensional case. Consider a one-dimensional domain $0 \le x \le 1$ and a discretisation with $\delta x = 1/L$, such that $c_l = c(l\delta x)$ with $0 \le l \le L$. Furthermore, assume that the concentration on the boundaries of the domain are fixed, i.e. $c_0 = C0$ and $c_L = CL$. The one-dimensional Laplace equation reads $\partial^2 c/\partial x^2 = 0$, and therefore the finite difference equations become

$$c_{l-1} - 2c_l + c_{l+1} = 0, \quad 0 \le l \le L \quad .$$

In other words

$$\begin{aligned} C0 - 2c_1 + c_2 &= 0 \\ c_1 - 2c_2 + c_3 &= 0 \\ &\vdots \\ c_{L-3} - 2c_{L-2} + c_{L-1} &= 0 \\ c_{L-2} - 2c_{L-1} + CL &= 0 \end{aligned} \quad \text{or} \quad \begin{pmatrix} 2 & -1 & 0 & & \cdots & 0 \\ -1 & 2 & -1 & & & \\ 0 & -1 & 2 & -1 & & \\ \vdots & & & \ddots & & \vdots \\ & & & -1 & 2 & -1 \\ 0 & & \cdots & & -1 & 2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \\ \vdots \\ c_{L-2} \\ c_{L-1} \end{pmatrix} = \begin{pmatrix} C0 \\ 0 \\ \\ \vdots \\ 0 \\ CL \end{pmatrix} .$$

In this example we therefore find that $\mathbf{A}$ is a tri-diagonal matrix with $N = L - 1$. In higher dimensions $\mathbf{A}$ is still banded but has a somewhat more complicated structure. You may try yourself to find $\mathbf{A}$ for the two dimensional domain and boundary conditions used in the previous section.

Numerically solving sets of linear equations is a huge business and we can only give some important results and discuss some of the implications with respect to parallelism. A more in depth discussion can be found in the appendix, section E, and references therein.

One may solve the set of equation **Ax** = **b** using *direct* or *iterative* methods. In direct methods one solves the equations directly within some numerical error. In iterative methods one seeks an approximate solution via some iterative procedure. Table 1 shows the computational complexity for direct and iterative methods, both for banded and dense matrices. For iterative methods it is assumed that the number of iterations needed to find a solution within certain accuracy is much smaller than N. Direct methods allow finding the solution within numerical accuracy at the expense of a higher computational complexity. Especially for very large matrix dimensions, as are routinely encountered in today's large scale simulations this large computational complexity becomes prohibitive, even when executed on high-end massively parallel computers. That is the most important reason to consider iterative methods. They are cheaper, typically easier to parallelise, however at the expense only finding approximate solution. Sometimes convergence can be very slow, and iterative methods then require very specialized (preconditioning) techniques to recover convergence. These issues will not be discussed here. We will first give a short survey of some of the direct methods to solve a set of equations, and then turn our attention to iterative methods, applied to the case of the discretised two-dimensional Laplace equation. Much more in-depth discussion can be found in the appendix, section E and reference therein.

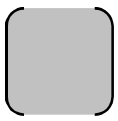
	dense matrix	banded matrix
		
direct method	$O(N^3)$	$O(N^2)$
iterative method	$O(N^2)$	$O(N)$

Table 1: Computational complexity for direct and iterative methods.

D.2. Direct Methods

Gaussian Eliminiation

The standard way to solve a system of equations is by *Gaussian Elimination*. You may know this method also under the name *matrix sweeping*. The idea is to bring the system of equations in *upper triangular form*. This means that all elements of the matrix below the diagonal are zero. So, the original system of equations

$$\begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}$$

is transformed to the system

$$\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1N} \\ 0 & a'_{22} & & \vdots \\ \vdots & & \ddots & a'_{N-1,N} \\ 0 & \cdots & 0 & a'_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b'_1 \\ \vdots \\ b'_N \end{pmatrix}.$$

This upper triangular form is easily solved through *back substitution*. First calculate

$$x_N = \frac{b'_N}{a'_{NN}},$$

followed by

$$x_{N-1} = \frac{b'_{N-1} - a'_{N-1,N} x_N}{a'_{N-1,N-1}},$$

etc. In a single expression we find

$$x_i = \frac{b'_i - \Sigma_{j=i+1}^{N} a'_{ij} x_j}{a'_{ii}},\ i = N, \cdots, 1\ .$$

Our main concern is to transform the original system **Ax** = **b** into the upper triangular form **A'x** = **b'**. This is achieved using Gaussian Eliminiation. Consider the composed matrix

$$(\mathbf{A}, \mathbf{b}) = \begin{pmatrix} a_{11} & \cdots & a_{1N} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{N1} & \cdots & a_{NN} & b_N \end{pmatrix}.$$

The recipe for Gaussian Elimination is

a) Determine an element $a_{r1} \neq 0$, proceed with (b). If such element does not exist, **A** is singular, stop.

b) Interchange row *r* and row 1 of (**A**, **b**), the result is $(\overline{\mathbf{A}}, \overline{\mathbf{b}})$.

c) For $i = 2, 3, \ldots, N$, subtract multiple $l_{i1} = \overline{a}_{i1} / \overline{a}_{11}$ of row 1 from row *i*. This results is matrix

$$(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1N}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(1)} & \vdots & \vdots & \vdots \\ \vdots & & \ddots & & \\ 0 & a_{N1}^{(1)} & & a_{NN}^{(1)} & b_N^{(1)} \end{pmatrix} = \left[\begin{array}{c|c|c} a_{11}^{(1)} & (\mathbf{a}^{(1)})^T & b_1^{(1)} \\ \hline 0 & \tilde{\mathbf{A}} & \tilde{\mathbf{b}} \end{array}\right]$$

Next, the same procedure is applied to $(\tilde{\mathbf{A}}, \tilde{\mathbf{b}})$ and repeated, i.e

$(\mathbf{A}, \mathbf{b}) := (\mathbf{A}^{(0)}, \mathbf{b}^{(0)}) \to (\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) \to \cdots \to (\mathbf{A}^{(N-1)}, \mathbf{b}^{(N-1)}) =: (\mathbf{U}, \mathbf{c})$.

The system (**U**,**c**) has the desired upper triangular form and can be solved easily by back substitution.

The element a_{r1} is called the *pivot* element, and the choice of the pivot is very important. In general it is not a good idea to take e.g. $a_{r1} = a_{11}$. This may lead to loss of accuracy, or even instability of the Gaussian Elimination. This happens is a_{11} is relatively small compared to the other elements. In that way the multipliers l_{i1} may become very large and in the subtraction of rows we may end up subtracting large numbers from each other, leading to enormous loss of precision. To avoid such problems we must always apply *partial pivoting*, i.e. choose $|a_{r1}| = \max_i |a_{i1}|$, i.e. pick the maximum element (in absolute measure) as the pivot. In general this procedure leads to stable and accurate solutions. For more information on Gaussian Elimination, read section **Error! Reference source not found.** of the appendix.

LU Factorization

Let us take a closer look at the sweeping of the matrix. Note that step (b), the interchanging of rows, can be formulated as

$$(\overline{\mathbf{A}}, \overline{\mathbf{b}}) = \mathbf{P}_1(\mathbf{A}, \mathbf{b})$$

where $\mathbf{P}_1$ is a permutation matrix. Consider for example the case where N =4 and we want to exchange row 1 and 2. In that case we find

$$\mathbf{P}_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Note that $\mathbf{P}_1\mathbf{P}_1 = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. From this we can conclude, and this is generally true for the permutation matrix, that $\mathbf{P}_1^{-1} = \mathbf{P}_1$, i.e. $\mathbf{P}_1$ is its own inverse. Also the actual sweeping in step (c) can be formulated as

$$(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \mathbf{G}_1(\overline{\mathbf{A}}, \overline{\mathbf{b}})$$

where

$$\mathbf{G}_1 = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ -l_{21} & 1 & & \\ \vdots & & \ddots & \\ -l_{N1} & & 0 & 1 \end{pmatrix}.$$

Note that the inverse of $\mathbf{G}_1$ is immediately found to be

$$\mathbf{G}_1^{-1} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & & \\ \vdots & & \ddots & \\ l_{N1} & & 0 & 1 \end{pmatrix}.$$

Combining this results in $(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \mathbf{G}_1\mathbf{P}_1(\mathbf{A}, \mathbf{b})$, which is easily generalized to $(\mathbf{A}^{(j)}, \mathbf{b}^{(j)}) = \mathbf{G}_j\mathbf{P}_j(\mathbf{A}^{(j-1)}, \mathbf{b}^{(j-1)})$, where $\mathbf{P}_j$ is a permutation matrix and $\mathbf{G}_j$ is

$$\mathbf{G}_j = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{j+1,j} & & \\ & & \vdots & \ddots & \\ 0 & & -l_{N,j} & & 1 \end{pmatrix}.$$

With all these definitions we can express Gaussian Elimination formally as

$$(\mathbf{U}, \mathbf{c}) = \mathbf{G}_{N-1}\mathbf{P}_{N-1}\mathbf{G}_{N-2}\mathbf{P}_{N-2}\cdots\mathbf{G}_1\mathbf{P}_1(\mathbf{A}, \mathbf{b}). \qquad [16]$$

With these definitions we can touch upon another very important class of direct methods, that of LU decompostion. The idea is to decompose the original matrix $\mathbf{A}$ into a product of an upper triangular matrix $\mathbf{U}$ and a lower triangular matrix $\mathbf{L}$, i.e. $\mathbf{A} = \mathbf{LU}$. Such decomposition is important because, once we have it, we can solve the system of equations for may right hand sides. This is easy seen. The original system was $\mathbf{Ax} = \mathbf{b}$. With the LU decomposition this becomes $\mathbf{LUx} = \mathbf{b}$. First solve $\mathbf{Ly} = \mathbf{b}$ using forward substitution (the same trick as with backward substitution, but now starting with y_1 and working in the forward direction). Next, solve $\mathbf{Ux} = \mathbf{y}$ through backward substitution.

In the special case of Gaussian Elimination *without* the need for row exchanges, i.e. when $\mathbf{P}_j = \mathbf{I}$, we can derive a **LU** decomposition using the multipliers l_{ij}. In this special case, from equation [16] we can immediately derive that

$\mathbf{U} = \mathbf{G}_{N-1}\mathbf{G}_{N-2}\cdots\mathbf{G}_1\mathbf{A} \Rightarrow \mathbf{A} = \mathbf{LU}$, where $\mathbf{L} = \mathbf{G}_1^{-1}\mathbf{G}_2^{-1}\cdots\mathbf{G}_{N-1}^{-1}$. Furthermore, it is easily verified that

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & \ddots & & \vdots \\ \vdots & & 1 & 0 \\ l_{N1} & \cdots & l_{N,N-1} & 1 \end{pmatrix}.$$

As an example, consider $\begin{pmatrix} 3 & 1 & 6 \\ 1 & 1 & 3 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 23 \\ 12 \\ 6 \end{pmatrix}$. With Gaussian Eliminiation one can easily verify that $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, $\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ 1/3 & 1 & 1 \end{pmatrix}$, and $\mathbf{U} = \begin{pmatrix} 3 & 1 & 6 \\ 0 & 2/3 & 1 \\ 0 & 0 & -2 \end{pmatrix}$. As an exercise perform the calculation yourself and check the results.

In general **LU** = **PA** with $\mathbf{P} = \mathbf{P}_{N-1}\,\mathbf{P}_{N-2}\,\ldots\,\mathbf{P}_1$ and **L** some permutation of elements from the $\mathbf{G}_j$ matrices. Many algorithms exist to find LU factorizations (see e.g. [2]). Finally note that also many optimized algorithms exist for special matrices, e.g for banded matrices. For more information on LU decomposition read section **Error! Reference source not found.** and **Error! Reference source not found.** of the appendix.

Parallel Libraries for Direct Methods

The scientific computing community has invested a lot of time in creating very powerful and highly efficient (parallel) public domain libraries for numerical algorithms. Many of them can be found on the famous *netlib* web site,.[3] which you are advised to visit and use the available software whenever needed.

In most linear algebra algorithms one typically encounters a small set of basic routines. These basic routines are formalized into the BLAS set (Basis Linear Algebra Subroutines, for more in-depth information, see the appendix, **Error! Reference source not found.**. The BLAS set is divided into three levels,

- BLAS 1: O(N) operations on O(N) data items, typically a vector update, e.g. : $\mathbf{y} = a\,\mathbf{x} + \mathbf{y}$;
- BLAS 2: O(N^2) operations on O(N^2) data items, typically matrix vector products, e.g. : $\mathbf{y} = a\mathbf{A}\,\mathbf{x} + b\,\mathbf{y}$;
- BLAS 3: O(N^3) operations on O(N^2) data items, typically matrix-matrix products, e.g. : $\mathbf{C} = a\mathbf{A}\,\mathbf{B} + b\,\mathbf{C}$.

Highly optimized BLAS libraries are available on e.g. netlib. Performance increases in going from BLAS-1 to BLAS-3 due to better ratio between amount of calculations and needed memory references. BLAS3 was specifically developed for use on parallel computers. Gaussian elimination or LU factorization can be implemented using BLAS-1, BLAS-2, or BLAS-3. Modern linear algebra libraries use BLAS-3, where the algorithms are formulated as block algorithms. An example of such a block LU factorization can be found in the appendix, section **Error! Reference source not found.**.

Using BLAS routines a number of well-known linear algebra libraries have been developed and put into the public domain. As examples we mention:

---

[3] www.netlib.org

EISPACK, Fortran routines for calculation of eigenvectors and eigenvalues for dense and banded matrices, based on BLAS-1;

LINPACK, Fortran routines to solve linear equations for dense and banded matrices, column oriented BLAS-1 approach;

LAPACK, successor of EISPACK and LINPACK, Fortran routines for eigenvalues and solving linear equation, for dense and banded matrices, exploit BLAS-3 (to improve speed), runs efficiently on vector machines and distributed memory parallel computers;

ScaLAPACK, extend LAPACK to run efficiently on distributed memory parallel computers, fundamental building blocks are distributed memory versions of BLAS-2 and BLAS-3 routines and a set of Basic Linear Algebra Communication Subprograms (BLACS).

We end this section with noting that a detailed discussion of parallel algorithms for e.g. LU factorization is beyond the scope of this lecture. The main features of such parallel algorithms are a block oriented BLAS-3 approach in combination with special (scattered) decompositions of the matrices. For more information we refer to e.g. Reference [4], chapter 9.5 or Reference [5].

D.3. Iterative Methods

The General Idea

As was already explained before, in iterative methods one seeks an approximate solution via some iterative procedure. The idea is to consider an iteration of the form $x^{(n+1)} = \Phi(x^{(n)})$, $n = 0, 1, 2, \ldots$ The superscript n is the iteration number. Given a first guess solution $x^{(0)}$ and the iteration function Φ one iterates the solution forward until some convergence criterion is met. Typically one demands that $\|x^{(n+1)} - x^{(n)}\| < \varepsilon$, where ε is some small number.

For our linear system $\mathbf{Ax} = \mathbf{b}$ an iterative procedure can be constructed as follows. First note that

$$\mathbf{Ax} = \mathbf{b} \rightarrow \mathbf{Bx} + (\mathbf{A} - \mathbf{B})\mathbf{x} = \mathbf{b}$$

then put

$$\mathbf{Bx}^{(n+1)} + (\mathbf{A} - \mathbf{B})\mathbf{x}^{(n)} = \mathbf{b}$$

or

$$\mathbf{x}^{(n+1)} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{x}^{(n)} - \mathbf{B}^{-1}\mathbf{b}$$

Many possibilities exist for the choice of the matrix **B**, thus leading to many different types of iterative methods (see e.g. [2], chapter 8). In this section we will only consider three more or less related iterative methods, the *Jacobi* iteration, the *Gauss-Seidel* iteration, and *Successive Over Relaxation*. Reference [2] and the appendix, sections **Error! Reference source not found. - Error! Reference source not found.**, provide many more examples of iterative methods. Here we especially mention the family of iterative methods called *Krylov space* methods, also knows as *Conjugate Gradient* methods. They are very powerful, in the sense that they require not so many iterations and that they can handle bad conditioned systems as well. In many cases they are *the* method of choice as a solver. Like the iterative methods that will be introduced here they are relatively easy to parallelize. Finally, many efficient parallel implementations of iterative methods are available in the public domain and a highly readable book that provides a cookbook style of using iterative methods (Reference [6]) can be obtained in electronic from netlib.

The Jacobi Iterative Method

Let us now return our attention to the specific case of the finite difference scheme for the two-dimensional Laplace equation (Eq. [14]). This equation immediately suggests an iterative scheme:

$$c_{l,m}^{(n+1)} = \frac{1}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n)}\right] . \qquad [17]$$

As before, n is the iteration index. This iterative scheme is known as the Jacobi iteration. Note the relation of Equation [17] with the finite difference scheme for the time dependent diffusion equation, Eq. [11]. If we take the by the CFL condition allowed maximum time step, i.e. $D\delta x^2/\delta t = ¼$ and use that in Eq. [11] we reproduce the Jacobi iteration of Eq. [17]. In other words, we expect that the Jacobi iteration will also suffer from a relative large amount of iterations needed before convergence.

Algorithm 2 provides the code for the inner loops of a Jacobi iteration. We again assume the square domain with periodic boundaries in the x-direction and fixed boundaries in the y-direction. Furthermore, a specific stopping criterion is implemented. We demand that

$$\max_{ij}\left|c_{ij}^{(n+1)} - c_{ij}^{(n)}\right| < \varepsilon .$$

The difference in concentration between two iterations on all grid points should be smaller that some small number ε. This is a rather severe stopping condition. Others can also be used, e.g. calculating the mean difference and demanding that this should be smaller that some small number. Here we will not pay any further attention to the stopping criterion, but you should realize this is an important issue to be considered in any new application of an iterative method.

```
/* Jacobi update, square domain, periodic in x, fixed */
/* upper and lower boundaries                         */
do {
   δ = 0
   for i=0 to max {
      for j=0 to max {
         if(c_ij is a source) c_ij^(n+1) = 1.0
         else if(c_ij is a sink) c_ij^(n+1) = 0.0
         else {
            /* periodic boundaries */
            west = (i==0) ? c_max-1,j^(n) : c_i-1,j^(n)
            east = (i==max) ? c_1,j^(n) : c_i+1,j^(n)
            /* fixed boundaries */
            south = (j==0) ? c0 : c_i,j-1^(n)
            north = (j==max) ? cL : c_i,j+1^(n)
             c_ij^(n+1) = 0.25 * (west + east + south + north)
         }
         /* stopping criterion */
         if(|c_ij^(n+1) - c_ij^(n)| > tolerance) δ = |c_ij^(n+1) - c_ij^(n)|
      }
   }
while (δ > tolerance)
```

Algorithm 2: The sequential Jacobi iteration

As an example again we take the two-dimensional square domain, $0 \le x, y \le 1$ with periodic boundary conditions in x-direction, $c(x,y) = c(x+1,y)$, and fixed values for the upper and lower boundaries, $c(x,y=0) = 0$ and $c(x,y=1) = 1$. Because of symmetry the

solution will not depend on the x-coordinate, and that the exact solution is a simple linear concentration profile: $c(x,y) = y$ (derive this result yourself !). The availability of the exact solution allows us to measure the error in an iterative method as a function of the applied stopping criterion. We will show some results after having introduced the other iterative methods, allowing for a comparison. Here we first concentrate on the number of iterations needed for convergence.

Set the small number ε in the stop condition to $\varepsilon = 10^{-p}$, where p is a positive integer. Next we measure the number of iterations that is needed for convergence, and we do this for two grid sizes, $N = 40$ and $N = 80$ (where $N = L + 1$). As first guess for the solution we just take all the concentration equal to zero. The results, shown in Figure 12, seem to suggest a linear dependence between the number iteration and p, i.e. a linear dependence on $-\log(\varepsilon)$. Furthermore, the results suggest an N^2 dependence. These results can be obtained from mathematical analysis of the algorithm, see e.g. [2, 3]. This N^2 dependence was also found for the time dependent case (see discussion in section C.1). Finally note that the number of iterations can easily become much larger than the number of grid points (N^2) and in that case direct methods to solve the equations are preferred. In conclusion, the Jacobi iteration works, but only as an example and certainly not to be used in real applications. However, the Jacobi iteration is a stepping stone towards a very efficient iterative procedure.

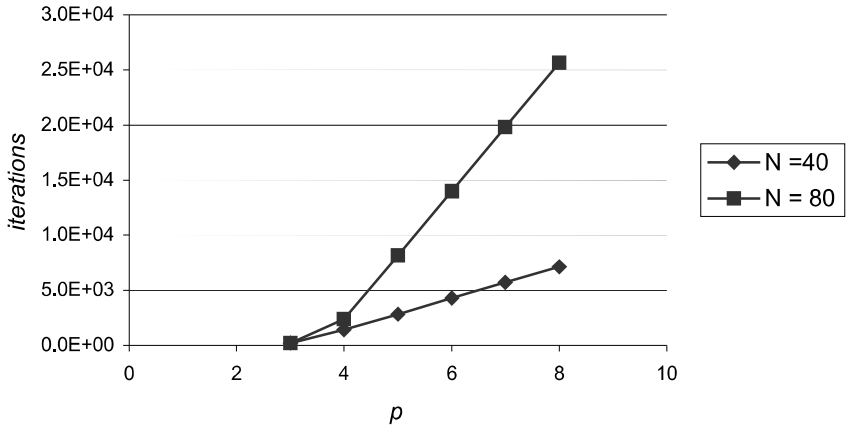


Figure 12: The number of iterations for the Jacobi iteration as a function of the stopping condition and as a function of the number of grid points along each dimension.

At first sight a parallel Jacobi iteration seems very straightforward. The computation is again based on a local five-point stencil, as in the time dependent case. Therefore, we can apply a domain decomposition, and resolve all dependencies by first exchanging all boundary points, followed by the update using the Jacobi iteration. However, the main new issue is the stopping condition. In the time dependent case we know before run time how many time steps need to be taken. Therefore each processor knows in advance how many iterations are needed. In the case of the Jacobi iteration (and all other iterative methods) we can only decide when to stop during the iterations. We must therefore implement a parallel stopping criterion. This posses a problem, because the stopping condition is based on some global measure (e.g. a maximum or mean error over the complete grid). That means that a global communication (using e.g. `MPI-Reduce`) is needed that may induce a large communication overhead. In practical implementations one must seriously think about this. Maybe it pays off to calculate the stopping condition once every q iterations (with q some small positive number) instead of after each iteration. This depends on many details, and you are challenged to think about this yourself (and apply your ideas in the lab course that is associated to this lecture). With all this in mind the pseudo code for the parallel Jacobi iteration is given in Algorithm 3. It is clear that it closely resembles the time-dependent pseudo code.

```
main () /* pseudo code for parallel Jacobi iteration */
{
   decompose lattice;
   initialize lattice sites;
   set boundary conditions;
   do {
      exchange boundary strips with neighboring processors;
      for all grid points in this processor {
         update according to Jacobi iteration;
         calculate local δ_l parameter; /* stopping criterion */
      }
      obtain the global maximum δ of all local δ_l values
   }
   while (δ > tolerance)
   print results to file;
}
```

Algorithm 3: Pseudo code for the parallel Jacobi iteration.

The Gauss-Seidel Iterative Method

The Jacobi iteration is not very efficient, and here we will introduce a first step to improve the method. The *Gauss-Seidel* iteration is obtained by applying a simple idea. In the Jacobi iteration we always use results from the previous iteration to update a point, *even* when we already have new results available. The idea of Gauss-Seidel iteration is to apply new results as soon as they become available. In order to write down a formula for the Gauss-Seidel iteration we must specify the order in which we update the grid points. Assuming a row-wise update procedure (i.e. we increment l while keeping m fixed) we find for the Gauss-Seidel iteration

$$c_{l,m}^{(n+1)} = \frac{1}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n+1)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n+1)}\right] . \qquad [18]$$

One immediate advantage of the Gauss-Seidel iteration lies in the memory usage. In the Jacobi iteration you would need two arrays, one to store the old results, and another to store the new results. In Gauss-Seidel you immediately use the new results as soon as they are available. So, we only need one array to store the results. Especially for large grids this can amount to enormous savings in memory! We say that the Gauss-Seidel iteration can be computed *in place*.

Is Gauss-Seidel iteration also faster than Jacobi iteration. According to theory it turns out that a Gauss-Seidel iteration requires a factor of two iterations less then Jacobi (see [3], section 17.5). This is also suggested by a numerical experiment. We have taken the same case as in the previous section, and for $N = 40$ we have measured the number of iterations needed for Gauss-Seidel and compared to Jacobi. The results are shown in Figure 13. The reduction of the number of iterations with a constant number is indeed observed, and this constant number is very close to the factor of two as predicted by the theory. This means that Gauss-Seidel iteration is still not a very efficient iterative procedure (as compared to direct methods). However, Gauss-Seidel is also only a stepping stone towards the Succesive Over Relaxation method (see next section) which *is* a very efficient iterative method.

The Gauss-Seidel iteration posses a next challenge to parallel computation. At first sight we must conclude that the parallelism available in Jacobi iteration is now completely destroyed by the Gauss-Seidel iteration. Gauss-Seidel iteration seems inherently sequential. Well, it is in the way we introduced it, with the row-wise ordering of the computations. However, this row-wise ordering was just a convenient choice. It turns out that if we take another ordering of the computations we can restore parallelism in the

Gauss-Seidel iteration. This is an interesting case where reordering of computations provides parallelism. Keep this in mind, as it may help you in the future in finding parallelism in algorithms!

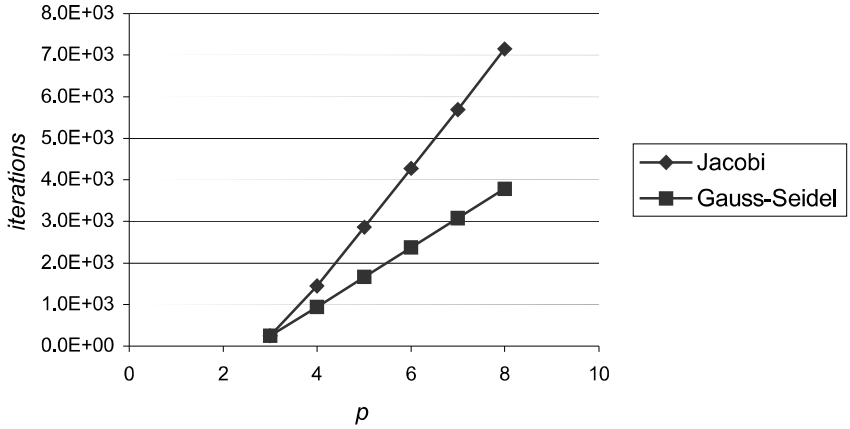


Figure 13: The number of iterations for the Jacobi and Gauss-Seidel iteration as a function of the stopping condition for N = *40.*

The idea of the reordering of the computations is as follows. First, color the computational grid as a checkerboard, with red and black grid points. Next, given the fact that the stencil in the update procedure only extends to the nearest neighbors, it turns out that all red points are independent from each other (they only depend on black points) and vice-versa. So, instead of the row-wise ordering we could do a red-black ordering, were we first update all red points, and next the black points (see Figure 14). We also call this Gauss-Seidel iteration, because although the order in which grid points are updated is now different, we also do the computation in place, and use new results as soon as they become available.

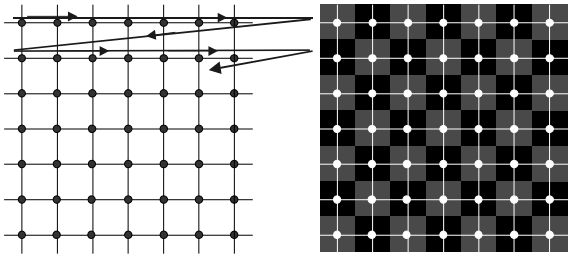


Figure 14: Row wise ordering (left) versus red-black ordering (right).

This new red-black ordering restores parallelism. We can now first update all red points in parallel, followed by a parallel update of all black point. The pseudo code for parallel Gauss-Seidel with red-black ordering is show in Algorithm 4. The computation is now split into two parts, and before each part a communication with neighboring processors is needed. On first sight this would suggest that the parallel Gauss-Seidel iteration requires twice as many communication time in the exchange part. This however is not true, because it is not necessary to exchange the complete set of boundary points, but only half. This is because for updating red points we only need the black point, so also only the black boundary points need to be exchanged. This means that, in comparison with parallel Jacobi, we only double the setup times required for the exchange operations, but keep the sending times constant. Finally note that the same parallel stop condition as before can be applied.

```
/* only the inner loop of the parallel Gauss-Seidel method with */
/* Red Black ordering */
   do {
      exchange boundary strips with neighboring processors;
      for all red grid points in this processor {
         update according to Gauss-Seidel iteration;
      }
      exchange boundary strips with neighboring processors;
      for all black grid points in this processor {
         update according to Gauss-Seidel iteration;
      }
      obtain the global maximum δ of all local δ_l values
   }
   while (δ > tolerance)
```

Algorithm 4: The pseudo code for parallel Gauss-Seidel iteration with red-black ordering.

Another final remark is that instead of applying red-black ordering to each grid point in the domain, we can also create a coarse-grained red-black ordering, as drawn in Figure 15. Here the red-black ordering is done on the level of the decomposed grid. The procedure could now be to first update the red domain followed by an update of the black doamin. Within each domain updating can now be done with the row-wise ordering scheme (why?).

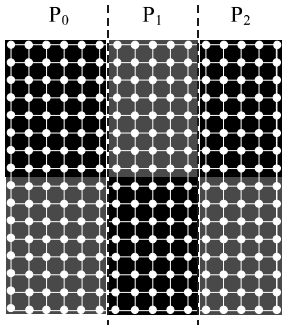


Figure 15: A coarse-grained red-black ordering.

Successive Over Relaxation

The ultimate step in the series from Jacobi and Gauss-Seidel is to apply a final and as will become clear very efficient idea. In the Gauss-Seidel iteration the new iteration results is completely determined by its four neighbors. In the final method we apply a correction to that, by mixing the Gauss-Seidel result with the current value, i.e.

$$c_{l,m}^{(n+1)} = \frac{\omega}{4}\left[c_{l+1,m}^{(n)} + c_{l-1,m}^{(n+1)} + c_{l,m+1}^{(n)} + c_{l,m-1}^{(n+1)}\right] + (1-\omega)\, c_{l,m}^{(n)}. \qquad [19]$$

The parameter ω determines the strength of the mixing. One can prove (see e.g. [2]) that for $0 < \omega < 2$ the method is convergent. For $\omega = 1$ we recover the Gauss-Seidel iteration, for $0 < \omega < 1$ the method is called Successive Under Relaxation. For $1 < \omega < 2$ we speak of *Successive Over Relaxation*, or SOR.

It turns out that for finite difference schemes SOR is very advantageous and gives much faster convergence then Gauss-Seidel. The number of needed iterations will however strongly depend on the value of ω. One needs to do some experiments in order to determine its optimum value. In our examples we have take ω always close to 1.9. The enormous improvement of SOR as compared to Gauss-Seidel and Jacobi is illustrated by

again measuring the number of iterations for the example of the square domain with N =40. The results are shown in Figure 16, and show the dramatic improvement of SOR. Another important result is the final accuracy that is reached. Remember that we know the exact solution in our example. So, we can compare the simulated results with the exact solution and from that calculate the error. We have done that and the results are shown in Figure 17. Note the logarithmic scale on error-axis in Figure 17. Because we observe a linear relationships between the logarithm of the error and p we can conclude that we find a linear relationship between the stopping condition ε and the mean error in the simulations. Furthermore we observe that the error in the SOR is much smaller than in Jacobi and Gauss-Seidel. This further improves the efficiency of SOR. Suppose you would like to get mean errors of 0.1%. In that case in SOR we only need to take $p = 4$, whereas Jacobi or Gauss-Seidel require $p = 6$. SOR is a practically useful iterative method and as such is applied regularly. Finally note that parallel SOR is identical to parallel Gauss-Seidel. Only the update rules have been changed.

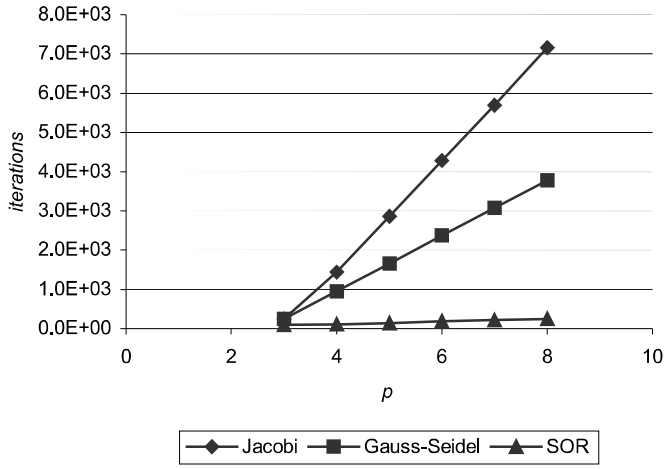


Figure 16: The number of iterations for the Jacobi, Gauss-Seidel, and SOR iteration as a function of the stopping condition for N = *40.*

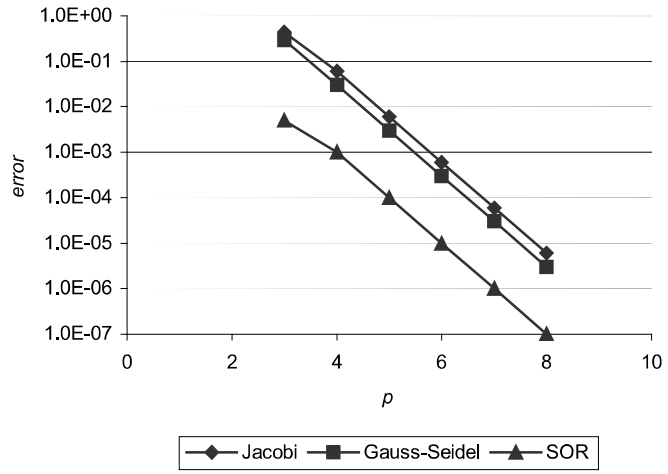


Figure 17: The mean error for the Jacobi, Gauss-Seidel, and SOR iteration as a function of the stopping condition for N = *40.*

Iterative Methods in Matrix Notation

So far we only considered the iterative methods in the special case of the finite difference discretization of the two-dimensional Laplace equation. Now we return to the general idea

of constructing iterative methods. Remember that in general an iterative method can be constructed from $\mathbf{x}^{(n+1)} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{x}^{(n)} - \mathbf{B}^{-1}\mathbf{b}$. Now with

$$\mathbf{A} = \mathbf{D} + \mathbf{E} + \mathbf{F}, \qquad \mathbf{D} = \begin{pmatrix} a_{11} & & 0 \\ & \ddots & \\ 0 & & a_{NN} \end{pmatrix}$$

$$\mathbf{E} = \begin{pmatrix} 0 & & & 0 \\ a_{21} & \ddots & & \\ \vdots & & \ddots & \\ a_{N1} & \cdots & a_{N,N-1} & 0 \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1N} \\ & \ddots & & \vdots \\ & & \ddots & a_{N-1,N} \\ 0 & & & 0 \end{pmatrix}$$

we can define in a general way the three iterative methods introduced so far.

- Jacobi iteration

$$\mathbf{B} = \mathbf{D} \;\Rightarrow\; a_{ii} x_i^{(n+1)} = -\sum_{i \neq j} a_{ij} x_j^{(n)} + b_i$$

- Gauss-Seidel iteration

$$\mathbf{B} = \mathbf{D} + \mathbf{E} \;\Rightarrow\; a_{ii} x_i^{(n+1)} = -\sum_{i<j} a_{ij} x_j^{(n+1)} - \sum_{i>j} a_{ij} x_j^{(n)} + b_i$$

- SOR

$$\mathbf{B} = \frac{1}{\omega}\mathbf{D} + \mathbf{E} \;\Rightarrow a_{ii} x_i^{(n+1)} = \omega\left(-\sum_{i<j} a_{ij} x_j^{(n+1)} - \sum_{i>j} a_{ij} x_j^{(n)} + b_i\right) + (1-\omega) a_{ii} x_i^{(n)}$$

D.4. An Application: Diffusion Limited Aggregation

Diffusion Limited Aggregation (DLA) is a model for non-equilibrium growth, where growth is determined by diffusing particles. It can model e.g. a Bacillus subtilis bacteria colony in a petri dish. The idea is that the colony feeds on nutrients in the immediate environment, that the probability of growth is determined by the concentration of nutrients and finally that the concentration of nutrients in its turn is determined by diffusion. The basic algorithm is shown in Algorithm 5.

```
1. Solve Laplace equation to get distribution of nutrients,
assume that the object is a sink (i.e. c = 0 on the object)
2. Let the object grow
3. Go back to (1)
```

Algorithm 5: The DLA algorithm.

The first step in Algorithm 5 is done by a parallel SOR iteration. Step 2, growing of the object, requires three steps;

1. determine growth candidates;
2. determine growth probabilities;
3. grow.

A growth candidate is basically a lattice site that is not part of the object, but whose north -, or east -, or south -, or west neighbor is part of the object. In Figure 18 a possible configuration of the object is shown; the black circles form the current object, while the white circles are the growth candidates.

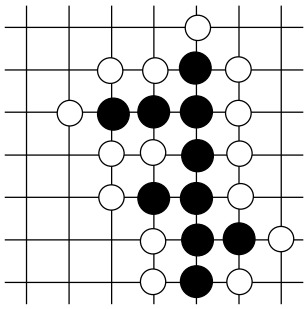


Figure 18: The object and possible growth sites.

The probability for growth at each of the growth candidates is calculated by

$$p_g\left((i,j)\in\circ \rightarrow (i,j)\in\bullet\right)=\frac{\left(c_{i,j}\right)^{\eta}}{\sum_{(i,j)\in\circ}\left(c_{i,j}\right)^{\eta}}. \qquad [20]$$

The parameter η determines the shape of the object. For $\eta = 1$ we get the normal DLA cluster, i.e. a fractal object. For $\eta < 1$ the object becomes more compact (with $\eta = 0$ resulting in the Eden cluster), and for $\eta > 1$ the cluster becomes more open (and finally resembles say a lightning flash).

Modeling the growth is now a simple procedure. For each growth candidate a random number between zero and one is drawn and if the random number is smaller than the growth probability, this specific site is successful and is added to the object. In this way, on average just one single site is added to the object.

The results of a simulation are shown in Figure 19. The simulations were performed on a 256^2 lattice. SOR was used to solve the Laplace equation. As a starting point for the SOR iteration we used the previously calculated concentration field. This reduced the number of iterations by a large factor. For standard DLA growth ($\eta = 1.0$) we obtain the typical fractal pattern. For Eden growth a very compact growth form is obtained, and for $\eta = 2$, a sharp lighting type of pattern is obtained.

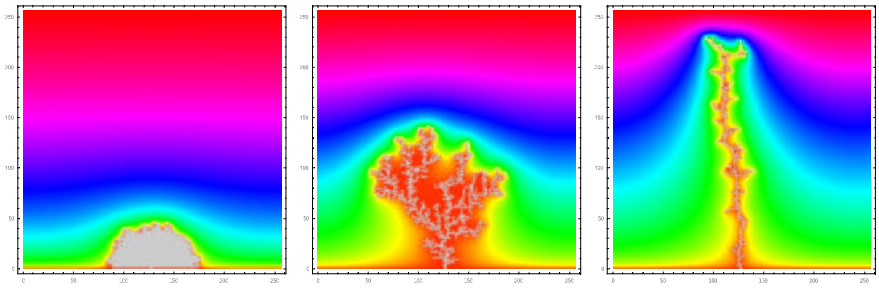


Figure 19: Results of DLA growth on a 256^2 lattice. The left figure is for $\eta = 0$, the middle for $\eta = 1.0$ and the right for $\eta = 2.0$.

Introduction of parallelism in DLA posses an interesting new problem (that we will not solve here, but in Chapter 4). It is easiest is to keep using the same decomposition as for the iterative solvers (e.g. the strip-wise decomposition of the grid). The growth step is local, and therefore can be computed completely in parallel. Calculation of the growth probabilities requires a global communication (for the normalization, i.e. the denominator of the equation for the probability). However, due to the growing object the load balancing gets worse and worse during the simulation. Our computational domain is *dynamically* changing during the simulation. This calls for a completely different view on decomposition, and that will be discussed in detail in Chapter 4.