1 Numerical Solvers

A. Introduction

In this chapter we will discuss Continuous Time and Discrete Time models (see Figure 1), and ask the question how we can obtain (parallel) simulations for such models.

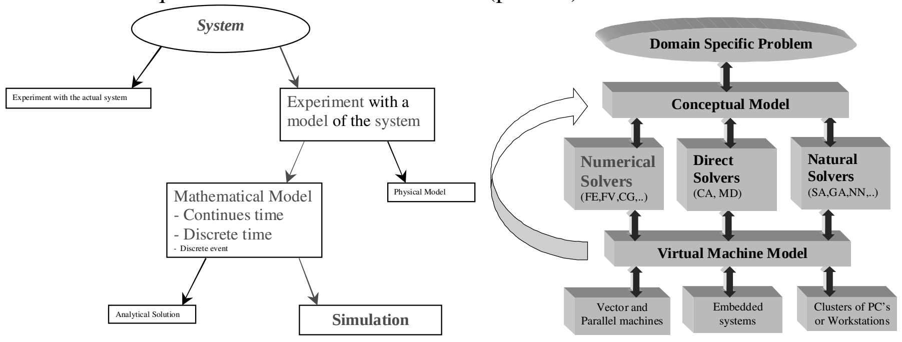


Figure 1: Focussing in on continuous time and discrete time models for the system under study (left) and assuming numerical solvers to map the models to the virtual machine (right).

We introduce an important, but not to complicated mathematical model, the *Diffusion Equation*, and discuss many ways in which numerical solvers may be obtained for this specific partial differential equation and how these solvers may be mapped to a parallel virtual machine model. This chapter will then end with a more involved case study, being the numerical solver for the equations that model hydrodynamic flow.

B. The Diffusion Equation

We consider the process of diffusion of e.g. heat in a solid material or solutes in a solvent. Diffusion can be modeled by a second order, linear partial differential equation, and we will investigate several ways to solve this equation numerically and simulate it on parallel computers. Despite the fact that the model itself is not too complicated, solving it numerically and simulating it on a parallel computers forces us to touch upon many important themes of parallel scientific computing that are also encountered in more complicated mathematical models. In that sense the diffusion equation provides us with a "graceful" introduction in the field of parallel numerical solvers for mathematical models.

Let us consider some chemical compound that is solved in a fluid. This compound will have a certain concentration *c* (in units of number of molecules/m^3). We assume that this compound is only transported by free diffusion in the solvent[1]. We will now derive an equation for the concentration *c*, by considering mass balance in a small volume and invoking Fick's law that relates a diffusion flux in a linear way with the concentration gradient.

[1] So we exclude the possibility of advection, that is transport of the solutes by the flow of the solvent.

Consider the situation, as in Figure 2, where a concentration gradient ∇c exists and that due to this gradient a net flux of solutes exists. A flux is an amount of material that passes through a certain area per time unit. This flux, **J**, is therefore measured in units of number (of molecules)/m^{2}s. Fick's law states that the flux and the concentration gradient depend on each other in a linear way,

$$\mathbf{J} = -D\nabla c\,, \qquad [1]$$

where D is the diffusion coefficient (in unit m^2/s). Fick's law is valid to a high degree of accuracy.

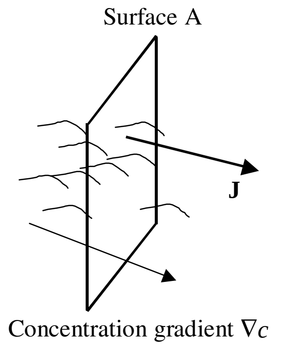


Figure 2: Due to a concentration ∇c gradient a flux **J** *exists.*

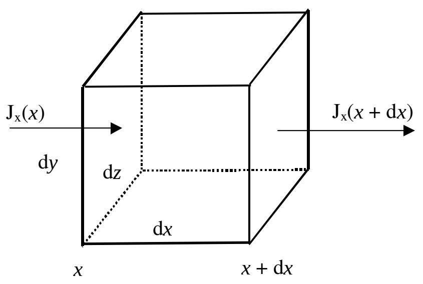


Figure 3: A small volume dxdydz

Next, consider an infinitesimally small volume dV = dxd$y$dz (see Figure 3) and calculate the amount of material that diffuses into and out of the cube. We do this here by treating the three Cartesian directions independently and next adding all contribution to get the total flux. For the amount of material flowing into the cube per unit time in the x-direction we may write $(\mathrm{J}_x(x) - \mathrm{J}_x(x + \mathrm{d}x))\mathrm{d}y\mathrm{d}z$, and likewise in the other two directions. Next, the total increase in amount of material per unit time must be equal to the total amount of material diffusing into the cube, i.e.

$$\begin{aligned}\frac{\partial c}{\partial t}\mathrm{d}V &= (\mathrm{J}_x(x) - \mathrm{J}_x(x+\mathrm{d}x))\mathrm{d}y\mathrm{d}z \\ &\quad + (\mathrm{J}_y(y) - \mathrm{J}_y(y+\mathrm{d}y))\mathrm{d}x\mathrm{d}z + (\mathrm{J}_z(z) - \mathrm{J}_z(z+\mathrm{d}z))\mathrm{d}x\mathrm{d}y\end{aligned}\,, \qquad [2]$$

where t denotes time. Dividing Equation [2] by dV and taking the limit dx, dy, dz $\to$ 0 we find

$$\frac{\partial c}{\partial t} = -\frac{\partial \mathrm{J}_x}{\partial x} - \frac{\partial \mathrm{J}_y}{\partial y} - \frac{\partial \mathrm{J}_z}{\partial z} = -\nabla\cdot\mathbf{J}\,. \qquad [3]$$

Finally, by combining Equation [3] with Fick's law (Equation [1]) we obtain the diffusion equation

$$\frac{\partial c}{\partial t} = D\nabla^2 c\,. \qquad [4]$$

One obvious solution to Equation [4] is c = constant. Not a very interesting result, which only applies in special situations (that is, under special initial – and boundary conditions). Let us investigate some other, more interesting exact solutions of the diffusion equation. These solutions are not only physically and mathematically interesting, they also provide us with solutions that allow us to *test* the correctness and accuracy of our simulations. Here we will just list some exact solutions of the diffusion equation, without deriving them. For this you need to invoke Laplace transformations. For those of you interested in the derivations, please consult your teacher.

First consider an unbounded one-dimensional domain, $-\infty < x < \infty$. The diffusion equation now reduces to $\partial c/\partial t = D\,\partial^2 c/\partial x^2$. Next we assume that on $x = 0$ and on $t = 0$ a pulse is applied, i.e. exactly at that time a certain amount of material is injected into the system, and let to diffuse. This initial condition can be mathematically expressed with the Dirac delta function, $\delta(x)$, defined as

$$\delta(x) = 0\ ,\ x \neq 0,$$

$$\int_{-\infty}^{\infty} \delta(x)dx = 1 \quad .$$

The initial condition now becomes $c(x,t) = 0$ for $t < 0$ and $c(x, t{=}0) = \delta(x)$. The solution to the diffusion equation becomes

$$c(x,t) = \frac{1}{\sqrt{4\pi Dt}} \exp\left(\frac{-x^2}{4Dt}\right). \qquad [5]$$

In the limit of $t \to 0$ Equation [5] again becomes $\delta(x)$ as it should. For $t > 0$ we observe a very smooth (Gaussian) curve that becomes broader and with smaller amplitude, see Figure 4. The total area under the curves remains constant, expressing the fact that the total mass is conserved. In the limit of $t \to \infty$ the concentration $c \to 0$ everywhere.

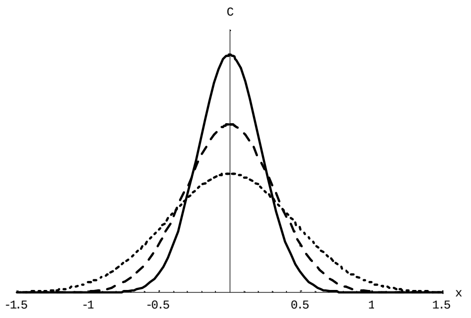


Figure 4: The evolution of the diffusing concentration (Equation [5]) as a function of Dt *(where the solid line is for* Dt = *0.1, the dashed line is for* Dt = *0.2 and the dotted line is for* Dt = *0.4).*

The smooth solutions of Equation [5] are typical for diffusion, and this simplifies the task of numerical solutions drastically. In other models one may encounter wildly varying solutions (think e.g. of complicated waves in acoustics or shocks in gas flow) with very steep gradients. These solutions require special attention and rather specialized numerical treatments, which are not part of this lecture.

Instead of applying a single pulse at $t = 0$, we may set the concentration to a constant value, starting at $t = 0$. This boundary condition simulates an infinite reservoir of solutes at a certain constant concentration. We now assume a half-infinite medium, i.e. $0 \leq x < \infty$. The boundary condition can be expressed mathematically with the Heavyside step function H(t), defined as

$$\mathrm{H}(t) = \begin{cases} 0\ ,\ t < 0 \\ 1\ ,\ t \geq 0 \end{cases} .$$

With H(t) we can express the initial and boundary conditions as $c(x,t) = 0$ for $t < 0$ and $c(x{=}0, t) = \mathrm{H}(t)$. The solution to the diffusion equation now becomes

$$c(x;t) = \operatorname{erfc}\left(\frac{x}{2\sqrt{Dt}}\right), \qquad [6]$$

where $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = 1 - \int_0^x e^{-t^2}\,dt$. These solutions are drawn in Figure 5. Again we observe nice smooth curves. The domain slowly fills due to the influx of the diffusing compounds. In the limit $t \longrightarrow \infty$ the concentration $c \longrightarrow 1$ everywhere.

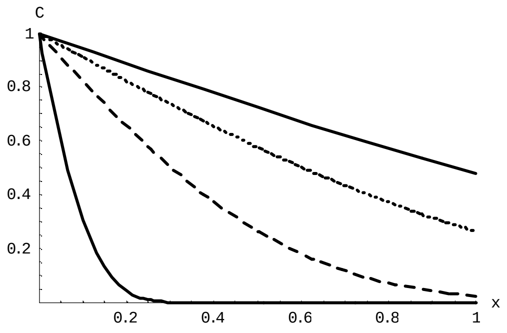


Figure 5: The evolution of the diffusing concentration (Equation [6]) as a function of Dt *(where the lower, solid line is for* Dt = *0.005, the dashed line is for* Dt = *0.1, the dotted line is for* Dt = *0.4, and the upper solid line is for* Dt = *1.0).*

A variant to the previous example is a bounded domain $0 \le x \le 1$ with a *source* of material on one side and a *sink* on the other side. A sink is a region where the concentration is constantly held at $c = 0$. A sink could e.g. model a surface that constantly adsorbs the chemical compounds in the solution. So, the boundary conditions become in this case $c(x, t) = 0$ for $t < 0$, $c(x=0, t) = 0$, and $c(x=1, t) = \mathrm{H}(t)$. The exact solution in this case becomes

$$c(x;t) = \sum_{i=0}^{\infty}\left[\operatorname{erfc}\left(\frac{1-x+2i}{2\sqrt{Dt}}\right) + \operatorname{erfc}\left(\frac{1+x-2i}{2\sqrt{Dt}}\right)\right]. \qquad [7]$$

The solutions as a function of time are drawn in Figure 6. As time increases the solution slowly diffuses to a steady state, i.e. a linear concentration profile. We can easily proof this. In the steady state situation there is no longer a dependence on time. All terms containing derivatives to time can be removed from the equations. So, we only need to look at the time-independent diffusion equation, $\partial^2 c/\partial x^2 = 0$. With the boundary conditions $c(x=0) = 0$ and $c(x=1) = 1$ it is easy to prove (do it yourself!) that the solution becomes $c(x) = x$. Note however that this long time behavior is quite different from the previous examples. In the previous examples the concentration was constant for long times, i.e. no concentration gradients existed and therefore no net mass flux exists anymore in the system. In this case however the steady state is a linear profile, i.e. a constant flux $\mathrm{J} = -\mathrm{D}\,\nabla c = -\mathrm{D}$ exists in the system. Clearly this is due to the constant influx of material at $x = 1$ and removal at $x = 0$. Such a steady state is usually called a Non-Equilibrium Steady State (NESS), meaning that we have a system in a steady state, but not in thermodynamic equilibrium.

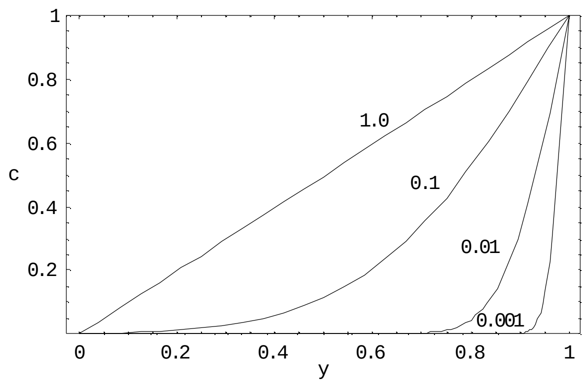


Figure 6: The evolution of the diffusing concentration (Equation [7]) as a function of Dt. *The numbers denote the value for* Dt *for the corresponding line.*

So far we only considered diffusion in one dimension. Pure one-dimensional diffusion is a very special case, only encountered in rare situations. It is more natural to consider diffusion in higher dimensions. In the rest of this chapter we consider diffusion in two dimensions, although all issues are easily translated to the case of three dimensions.

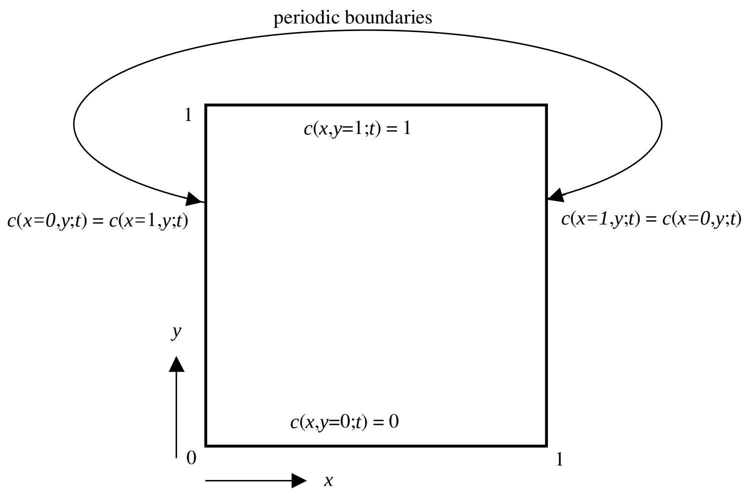


Figure 7: The 2 dimensional square domain with periodic boundary conditions in the x-direction.

Now consider a square domain, and without loss of generality we assume that $0 \le x \le 1$ and $0 \le y \le 1$. Furthermore, we always assume the following boundary conditions: $c(x, y = 1; t) = \mathrm{H}(t)$ and $c(x, y = 0; t) = 0$. So, on the top of the domain the concentration is always equal to one, and on the bottom of the domain the concentration is always equal to zero. Furthermore, in the *x*-direction we will always assume period boundary conditions: $c(x = 0, y; t) = c(x = 1, y; t)$, or in general $c(x = \xi, y; t) = c(x = \xi + 1, y; t)$. These boundary conditions are once more drawn in Figure 7. Note that because of the periodic boundary conditions the diffusion essentially becomes one-dimensional and the exact solution to the diffusion equation is again Equation 7 (with '*x*' replaced by '*y*'). This exact solution is very useful as it gives us a mean to study the accuracy of the numerical solution of the two-dimensional diffusion equation. Finally, to give a flavor of why we need numerical solutions to the diffusion equation, consider the example of Figure 8, where a square object is situated in the domain. The object is assumed to be absorbing, i.e. the boundary condition on the object simply states $c_{\text{on-object}} = 0$. This situation no longer allows for an analytical solution, and the diffusion equation was solved numerically using techniques that will be introduced in the next section. The presence of the object drastically changes the concentration field. Note however that the steady state solution, as shown in Figure 8 remains a well-behaved smooth function, as is expected from diffusion.

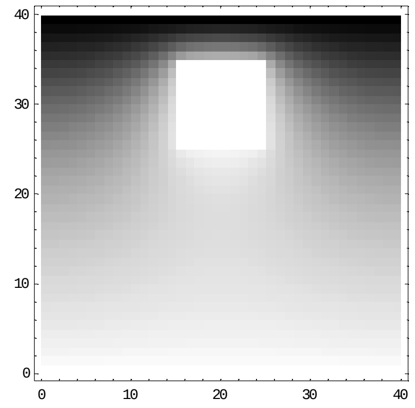


Figure 8: Diffusion in the domain of Figure 7, in which an absorbing object (the white square) has been placed. The concentration for large times is shown, with black for high concentration and white denoting low concentration.

C. Parallel Simulation of the Time Dependent Diffusion Equation

C.1. Discretisation

Let us now concentrate on the two-dimensional time dependent diffusion equation, i.e.

$$\frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right). \qquad [8]$$

We must solve this equation numerically, and for this we will apply the *explicit finite differencing*. Consider again a two-dimensional square domain with $0 \le x, y \le 1$. Furthermore, introduce a small time step δt and a small spatial step δx, and define a discretised spatial and temporal domain

$$\begin{array}{ll} t = n\,\delta t & n = 0, 1, 2, \ldots, \\ x = l\,\delta x & l = 0, 1, 2, \ldots, L, \text{ with } \delta x = 1/L, \\ y = m\,\delta x & m = 0, 1, 2, \ldots, L\,. \end{array}$$

Furthermore, define the notation

$$c(l\delta x, m\delta x, n\delta t) \equiv c_{l,m}^{n}\,.$$

Consider a first order Taylor expansion of $c(x,y,t+\delta t)$,

$$c(x,y,t+\delta t) = c(x,y,t) + \delta t\,\frac{\partial c(x,y,t)}{\partial t} + \mathrm{O}(\delta t^2)\,.$$

For small δt we can, to a good degree of accuracy, approximate the time derivative of c as

$$\frac{\partial c(x,y,t)}{\partial t} = \frac{c(x,y,t+\delta t) - c(x,y,t)}{\delta t},$$

or, in terms of the discretised points,

$$\frac{\partial c_{l,m}^{n}}{\partial t} = \frac{c_{l,m}^{n+1} - c_{l,m}^{n}}{\delta t}. \qquad [9]$$

For the spatial derivatives we perform a second order Taylor expansion,

$$c(x+\delta x, y, t) = c(x,y,t) + \delta x \frac{\partial c(x,y,t)}{\partial x} + \frac{1}{2}\delta x^2 \frac{\partial^2 c(x,y,t)}{\partial x^2} + \mathrm{O}(\delta x^3),$$
$$c(x-\delta x, y, t) = c(x,y,t) - \delta x \frac{\partial c(x,y,t)}{\partial x} + \frac{1}{2}\delta x^2 \frac{\partial^2 c(x,y,t)}{\partial x^2} + \mathrm{O}(\delta x^3).$$

By adding both equations and neglecting the small errors we find, again to a good approximation, an expression for the spatial derivatives (in terms of the grid points),

$$\frac{\partial^2 c_{l,m}^{n}}{\partial x^2} = \frac{c_{l-1,m}^{n} - 2c_{l,m}^{n} + c_{l+1,m}^{n}}{\delta x^2}. \qquad [10]$$

Likewise it is easy to derive an approximate expression for the derivatives in *y*:

$$\frac{\partial^2 c_{l,m}^{n}}{\partial y^2} = \frac{c_{l,m-1}^{n} - 2c_{l,m}^{n} + c_{l,m+1}^{n}}{\delta x^2}.$$

Substitution of Eqs [9-10] into Eq. [8] and rearranging terms results in the explicit finite difference equation that approximates the two-dimensional diffusion equation:

$$c_{l,m}^{n+1} = c_{l,m}^{n} + \frac{\delta t D}{\delta x^2}\left[c_{l-1,m}^{n} + c_{l+1,m}^{n} + c_{l,m-1}^{n} + c_{l,m+1}^{n} - 4c_{l,m}^{n}\right]. \qquad [11]$$

Equation [11] gives us a scheme to calculate the concentration on the next time step as a function of (known) concentrations on the previous time step. This allows a straightforward, *explicit* integration into time. Once a suitable initial condition is defined Equation [11] therefore allows a numerical solution of the time dependent diffusion equation. Extension of Equation 11 to three dimensions is straightforward and left as an exercise to the reader.

Note that Equation [11] is a *stencil* operation, comparable to the one you already encountered for the numerical solution of the one-dimensional wave equation. The main feature is that for updating the concentration on the next time step on a certain grid point (*l*,*m*), we only need information about the concentration, on the previous time step, on that grid point and on the nearest neighbors (*l*±1,*m*±1). Explicit schemes with stencil operations are very suitable for parallel computing (because of the inherent data locality in the stencil operation). We will come back to this in a moment. However, first we should consider some other important features of Equation [11].

We assume that Equation [11] results in a solution of the original diffusion equation [8]. However, in deriving Equation [11] we have introduced small errors. We therefore need to know what the accuracy of the solution of Equation [11] exactly is, and if Equation [11] always produces correct result or if other, non-physical solutions, are also possible. These questions to *accuracy*, *consistency*, and *stability* of the numerical approximation require a detailed mathematical analysis, that we will not provide here. For this we you are referred to text books, e.g. [2, 3].

Consistency is the requirement for any algebraic approximation to a differential equation to reproduce the differential equation in the limit of an infinitesimal time step. Without proof we state that Equation [11] is consistent with the two-dimensional diffusion equation. Accuracy and stability are related to the requirement that the deviation of the computed values of concentration from values given by the solution of the differential equation is small. It is possible to derive expressions for the accuracy of the finite difference scheme of Equation [11]. However, here we limit ourselves with the general remark that the time step δt and spatial step δx should be 'small enough' to capture all details of the solution. Given the fact that the concentration fields are well behaved, smooth functions, as suggested in the previous section, we may hope that this requirement of "smallness" does not result in prohibitive small δt and δx.

Finally consider the issue of stability. Stability is all about propagation of errors. Even if truncation and round off errors are very small, a scheme will be of little use if the effects of the small errors grow rapidly with time. Instability arises from the nonphysical solutions of the discretised equations. If the discrete equations have solutions which grow much more rapidly that the correct solution of the differential equations, then even a very small round off error will eventually explode, resulting in meaningless numerical results. Stability is a very important property of a numerical scheme. One can find schemes that are unconditionally stable, meaning that for any value of the parameters the iteration will keep errors under control. Other, rather useless algorithms are unconditionally unstable. Many schemes are conditionally stable, meaning that stability analysis results in bounds on the parameters in the scheme (e.g. a maximum allowed time step).

Our explicit finite difference scheme belongs to the class of conditionally stable algorithms. This can be shown easily by a so-called *von Neuman stability analysis*. We will demonstrate this for our specific numerical scheme, as it leads to the so-called *Courant-Friedrichs-Lewy stability criterion*, or in short the *CFL condition* or *Courant condition*. The CFL condition is frequently encountered in explicit schemes (you already encountered a CFL condition for the explicit scheme for the wave equation, remember it?) and therefore quite relevant.

The von Neuman stability analysis is based on the idea of expressing the solution to the finite difference equation as a summation of Fourier modes, and analyzing the amplitude of each Fourier mode. More concrete, consider a Fourier mode of the form

$$U_{l,m}^{n} = \xi^{n} e^{ik_x l\delta x} e^{ik_y m\delta x} . \qquad [12]$$

Substitution of Equation 12 into Equation 11 leads to

$$\xi^{n+1} e^{ik_x l\delta x} e^{ik_y m\delta x} = \xi^{n} e^{ik_x l\delta x} e^{ik_y m\delta x} + \frac{D\delta t}{\delta x^2} \xi^{n} [e^{ik_x (l+1)\delta x} e^{ik_y m\delta x} + e^{ik_x (l-1)\delta x} e^{ik_y m\delta x} + \\ e^{ik_x l\delta x} e^{ik_y (m+1)\delta x} + e^{ik_x l\delta x} e^{ik_y (m-1)\delta x} - 4 e^{ik_x l\delta x} e^{ik_y m\delta x}]$$

.

Dividing $\xi^{n} e^{ik_x l\delta x} e^{ik_y m\delta x}$ leads to

$$\begin{aligned} \xi &= 1 + \frac{D\delta t}{\delta x^2}[e^{ik_x\delta x} + e^{-ik_x\delta x} + e^{ik_y\delta x} + e^{-ik_y\delta x} - 4] \\ &= 1 + \frac{D\delta t}{\delta x^2}[2\cos(k_x\delta x) - 2 + 2\cos(k_y\delta x) - 2] \\ &= 1 - \frac{4D\delta t}{\delta x^2}[\sin^2(\frac{1}{2}k_x\delta x) + \sin^2(\frac{1}{2}k_y\delta x)] . \end{aligned}$$

The solution can only be stable if $|\xi| \le 1$, or $-1 \le \xi \le 1$. The condition $\xi \le 1$ is always true, however $\xi \ge -1$ leads to the CFL condition

$$\frac{D\delta t}{\delta x^2} \le \frac{1}{4} . \qquad [13]$$

In practice, this CFL condition means that given a step size δx and a diffusion coefficient D that we are limited in the time step δt.

Physically this means that the maximum allowed time step is limited to a typical "diffusion time" over a lattice site, $t_D \sim (\delta x)^2/D$. Assume we simulate in a domain with spatial scales $\lambda = L\delta x$. The diffusion time through the domain is $\tau \sim \lambda^2/D$. This implies that in a simulation the number of time steps therefore will be in the order of $\tau/t_D = \lambda^2/\delta x^2 = L^2$. Therefore, if a small step size is needed to capture small scale details of the computational domain, or if the computational domain is very large, the O(L^2) squared scaling of the number of time steps is very bad and may result in large execution times.

A possible way to circumvent this problem of relative small time steps lies in the application of implicit methods. We are not going to discuss these methods in detail, but just mention a few facts that show that this line of thinking poses an interesting dilemma to the computational scientist. In the derivation of the explicit scheme of Equation [11] we evaluated the spatial derivatives on the 'old time' t. We might as well evaluate them on the 'new time' $t + \delta t$. This would result in

$$c_{l,m}^{n+1} = c_{l,m}^{n} + \frac{\delta t D}{\delta x^2}\left[c_{l-1,m}^{n+1} + c_{l+1,m}^{n+1} + c_{l,m-1}^{n+1} + c_{l,m+1}^{n+1} - 4c_{l,m}^{n+1}\right].$$

It is easy to prove, using the von Neumann stability analysis, that this scheme is unconditionally stable (try this yourself!). This means that we can take much larger time steps, far beyond those allowed by the explicit scheme. However, the downside is that the implicit scheme requires the solution of a system of L^2 equations with L^2 unknowns, which implies many more computations then the simple forward integration of the explicit scheme. So, in the implicit scheme the number of required iterations goes down, but the cost per iteration goes up. To complicate things even further, explicit schemes, as we will see below, are easy to parallelize and the parallel execution will typically have efficiencies very close to 1. The implicit scheme however is much more difficult to parallelize, and the efficiency will be not as good. So, here is the dilemma. If you want to simulate the time dependent diffusion equation on a massively parallel computer, what numerical scheme would you choose? Think about this for a while, how would you solve this problem?

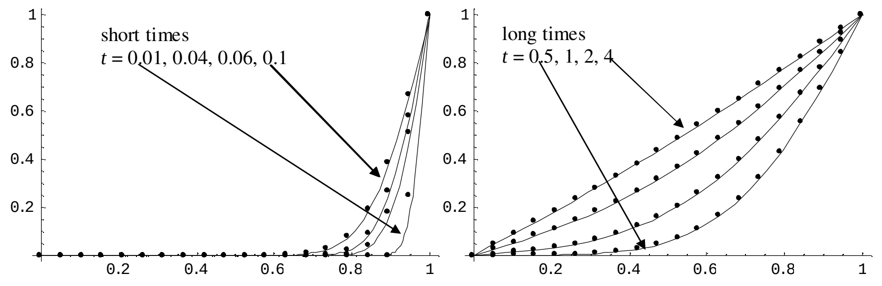


Figure 9: The results of the simulation of the diffusion equation in the 2D domain, with a step in the concentration at y = 1. The dots are the results of the numerical simulation, the solid lines are the exact solution from Equation [7].

To show that the explicit scheme actually works, consider the domain from Figure 7. As stated before, the boundary conditions are: $c(x, y = 1; t) = \mathrm{H}(t)$, $c(x, y = 0; t) = 0$ and $c(x = \xi, y; t) = c(x = \xi + 1, y; t)$. The spatial domain was discretised with a grid space δx = 1/20. We take the maximum allowed time step in the explicit scheme: $D\delta x^2/\delta t = 1/4$; assume D = 1; dt = 0.01. We simulated the diffusion equation using the explicit finite difference scheme from Equation [11]. The results are shown in Figure 9. The simulation results agree very good with the theoretical expression in Equation [7], both for short and long times. Also note that for $t = 4$ the solution has already converged to the steady state, i.e. the linear profile. Finally note that for the shortest time, where the steepest gradients in the concentration exist the numerical errors are largest. This is off course expected.

C.2. Parallelization

Parallelization is straightforward, since the stencil operation in the finite difference scheme is strictly local. We can therefore apply the same technique as with the parallelization of the one-dimensional string [1]. We apply a domain decomposition of the computational domain. The only difference is that the domain in this case is two-dimensional.

Suppose the domain is decomposed into strips (a *strip-wise decomposition*), as drawn in Figure 10. Each strip will be assigned to a processor P_i. To get good load balancing each strip should have an equal amount of columns. In general this is not possible (e.g. in Figure 10 P_0 gets 6 columns, the other 2 processors get 5 columns) and a fractional load imbalance overhead will be induced. The effect of this will be analyzed below.

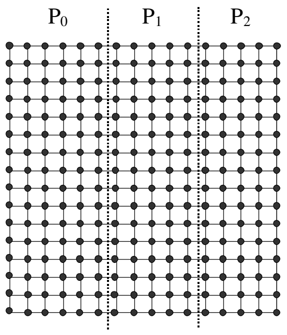


Figure 10: A 16×16 domain decomposed into 3 vertical strips, each assigned to a processor P_i.

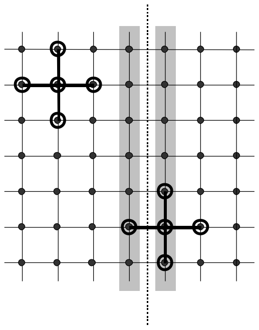


Figure 11: Stencils in the domain. The left stencil is internal, i.e. not overlapping with neighboring processor (dotted line is the border between two processors). The right stencil has an overlap with the neighboring processor and therefore induces a communication. All grid points in the gray boundary columns induce communication.

To update a point (l,m) we need information from its nearest neighbors, i.e. from (l-1,m), (l+1,m), (l,m-1), and (l,m+1). This results in a stencil. In Figure 11 two of these stencils are schematically drawn. This figure also shows the border between two processors (the dotted line). We can now define internal points as points of the domain where the stencil connected to that point completely lies in the sub domain associated to a processor. Boundary points are those points with stencils that partly lie in the sub domain of another processor. For our specific stencil and for the strip wise decomposition, the boundary points are just columns immediately left and right to processor borders. In Figure 11 these are the gray zones.

In a parallel computation, assuming the SPMD model and distributed memory computers, the boundary points will induce communication. This is because the values of the concentration on points in the stencil that lie in a different processor must first be known (i.e. communicated) before the concentration of a boundary point can be updated to the next time step. After a communication step in which are boundary points are exchanged, the iteration can proceed as in the sequential case, because each processor has all needed data in memory. The pseudo code for this parallel algorithm is shown in Algorithm 1. This pseudo code is very straightforward and applies to any stencil based explicit update scheme. Just compare Algorithm 1 with the pseudo code for the vibrating string example in the lecture “Introduction Parallel Computing” and you will immediately notice the similarity. The main difference lies in the fact that here we have to deal with a two-dimensional domain. This means that in the exchange operation not single points are communicated between processors, as in the case of the one dimensional string example, but a complete column of boundary points is exchanged.[2] In general, if we have a D-dimensional domain and stencil based operations the communication will be based on D-1 dimensional arrays.

```
main () /* pseudo code for parallel simulation          */
        /* of the time dependent diffusion equation     */
        /* using the explicit finite difference scheme  */
   {
     decompose lattice;      /* create stip-wise decomposition */
     initialize lattice sites;
     set boundary conditions;
     for N time steps {
        exchange boundary strips with neighboring processors;
        for all grid points in this processor {
           update according to finite difference scheme
        }
        print results to file; /* e.g. after every n time steps */
     }
   }
```

Algorithm 1: Parallel pseudo code for the simulation of the time dependent diffusion equation.

As always it is good practice to analyze the expected parallel execution time.. The resulting expressions are of course approximations, but they do give a good insight in the expected relative efficiency. Here we will just analyze the efficiency of a single iteration of the explicit finite difference scheme. Assuming that the iterations will take most of the execution time, it is reasonable to just consider a single iteration. However, when scaling the calculation to many processors, we off course must realize that other effects kick in, such as small sequential overheads outside the main loop and the like.

Using the same notation as in reference [1] and noting the N is the number of grid points along a single dimension (i.e. $N = L + 1$), and assuming the standard linear relationship for point-to-point communication time, we find

$$\begin{aligned}
T_1 &= N^2 \tau_{calc} \,, \\
T_p &= \frac{N^2}{p} \tau_{calc} + T_{comm} \\
&= \frac{N^2}{p} \tau_{calc} + 2(\tau_{setup} + N \tau_{exchange}) \,, \\
\varepsilon_p &= \frac{S_p}{p} = \frac{T_1}{pT_p} = \frac{1}{1 + \frac{2p}{N^2} \frac{\tau_{setup}}{\tau_{calc}} + \frac{2p}{N} \frac{\tau_{exchange}}{\tau_{calc}}} \,.
\end{aligned}$$

The relative efficiency ε_p has the well-known format, and displays two sources of overhead, both due to communication. These overhead also take the typical form, in the sense that we find a quotient of hardware parameters (τ_{setup}/τ_{calc} and $\tau_{exchange}/\tau_{calc}$) and a quotient that is related to the grain size (N^2/p and N/p). For reasonably well balanced parallel computers (i.e. with the hardware quotients of the O(1)) the efficiency will be very good if N/p, i.e. the number of column in the grid assigned to each processor, is much larger then one. However, modern processor technology has resulted in an

[2] Note that in the exchange operation you are not going to exchange the boundary points one-by-one, but collect all points to be exchanged in a single vector (e.g. using the special MPI data types for strided arrays) and then exchange the complete vector in one communication. What is the reason for this ?

enormous speed increase in processors that has not been matched by a comparable gain in network speeds. This means that the hardware coefficients are typically much larger then 1, and therefore high efficiencies require larger grain size. Especially the τ_{setup}/τ_{calc} can become very large. So, depending on the value of N and p the efficiency will deteriorate when increasing p either due to large setup times for communication or due to the transfer of the data itself.

The analysis so far assumed that N/p is an integer. This of course is not true in general. We need to take this into account, giving rise to the issue of load imbalance overhead. Below we show how this can be included in the expressions (where $\lceil x \rceil$ is the ceiling function).

$$T_p = N\left\lceil \frac{N}{p} \right\rceil \tau_{calc} + T_{comm} ,$$

$$\begin{aligned}\varepsilon_p &= \frac{\frac{N^2}{p}\tau_{calc}}{N\left\lceil \frac{N}{p} \right\rceil \tau_{calc} + T_{comm}} = \frac{1}{\frac{p}{N}\left\lceil \frac{N}{p} \right\rceil + \frac{pT_{comm}}{N^2\tau_{calc}}} , \\ &= \frac{1}{1 + \frac{pN\left\lceil \frac{N}{p} \right\rceil - N^2}{N^2} + \frac{pT_{comm}}{N^2\tau_{calc}}} .\end{aligned}$$

The term $\frac{pN\left\lceil \frac{N}{p} \right\rceil - N^2}{N^2}$ is the fractional load imbalance overhead, which has again the same form as for the one dimensional string example in [1]. The nominator is the difference between the total amount of calculations that could have been done in parallel and the total amount of calculations that are actually done. The fractional load imbalance overhead therefore gives the relative amount of lost, or unused cycles.

Finally some remarks about the decomposition. We have chosen for the strip-wise decomposition. However, even in the simple square domain it is possible to define many other types of decomposition, e.g. a block wise decomposition where the domain is decomposed into square blocks. Such decomposition leads to other expressions for the relative efficiency. It turns out that the efficiency becomes better (we leave it as an exercise to you to actually try to derive the expressions, so, do it !). However, if the decomposition is not done in a clever way, the fractional load imbalance overhead may drastically increase. Depending on the value of p, a decomposition in square blocks may not be possible, and one must consider rectangular blocks. For instance, if p = 12, one could decompose the x-direction in 4 parts and the y-direction in 3 parts, or decompose the x-direction in 2 parts and the y-direction in 6 parts. Clearly, many choices are possible. Another issue is that block-wise decomposition also results in more complicated codes, because a grid point that lies in a corner of a processor domain will have its stencil pointing into two other processors. Decomposition becomes even more difficult if the computational domain is no longer square, but takes any complicated form. In that case we need to invoke special algorithms for decomposition. This is the topic of chapter 4 of this syllabus.

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

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

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

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

where $\mathbf{A}$ is a $(N\times N)$ matrix and $\mathbf{x}$ and $\mathbf{b}$ are $(N\times1)$ 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

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

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 $\mathbf{Ax} = \mathbf{b}$ 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.

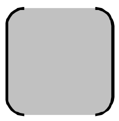
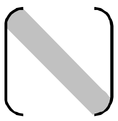
	dense matrix 	banded matrix 
direct method	O(N^3)	O(N^2)
iterative method	O(N^2)	O(N)

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

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

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, …, N, subtract multiple $l_{i1} = \overline{a}_{i1} / \overline{a}_{11}$ of row 1 from row i. This results is matrix

$$(\mathbf{A}^{(1)}, \mathbf{b}^{(1)}) = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1N}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(1)} & \vdots & \vdots & \vdots \\ \vdots & & \ddots & & \\ 0 & a_{N2}^{(1)} & & a_{NN}^{(1)} & b_N^{(1)} \end{pmatrix} = \left[\begin{array}{c:c:c} a_{11}^{(1)} & (\mathbf{a}^{(1)})^T & b_1^{(1)} \\ \hdashline 0 & \tilde{\mathbf{A}} & \tilde{\mathbf{b}} \end{array} \right]$$

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

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

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 if 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 an 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 decomposition. 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 many right hand sides. This is easly 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 $\mathbf{LU}$ 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 **P** = $\mathbf{P}_{N\text{-}1}\,\mathbf{P}_{N\text{-}2}\,\ldots\,\mathbf{P}_1$ 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:

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;

[3] www.netlib.org

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 than 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 {
   $\delta$ = 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 ) $\delta$ = |$c_{ij}^{(n+1)}$ - $c_{ij}^{(n)}$|
      }
   }
while ($\delta$ > 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, $c(x{=}0,y) = c(x{=}1,y)$, 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(ε). 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.

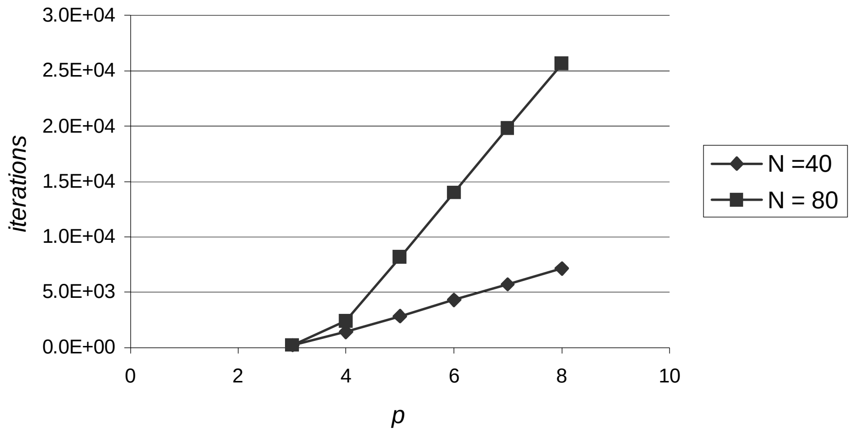


Figure 12: The number of iterations for the Jacobi iteration as a function of the stopping condition and as a function of the number of grid points along each dimension.

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 $\delta_l$ parameter; /* stopping criterion */
      }
      obtain the global maximum $\delta$ of all local $\delta_l$ values
   }
   while ($\delta$ > 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!

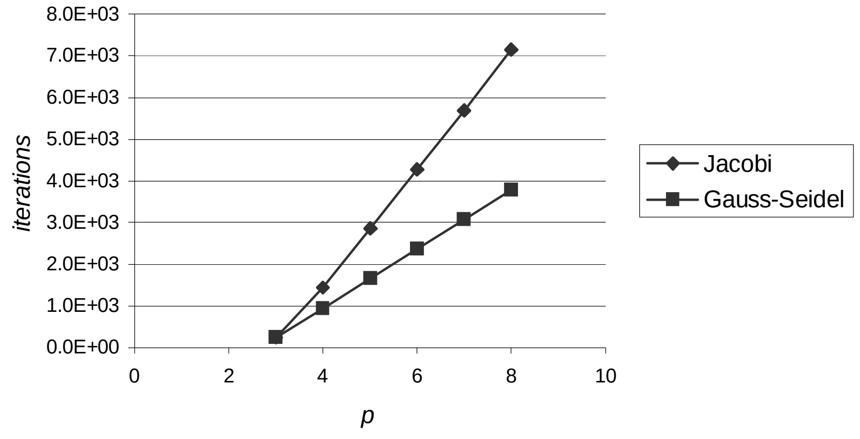


Figure 13: The number of iterations for the Jacobi and Gauss-Seidel iteration as a function of the stopping condition for N = *40.*

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.

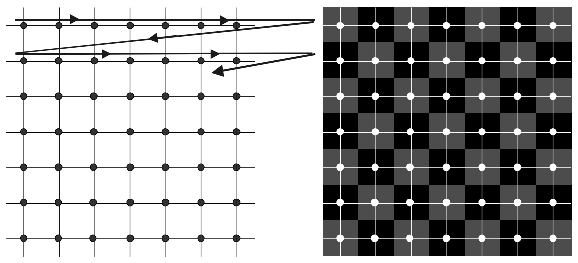


Figure 14: Row wise ordering (left) versus red-black ordering (right).

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 $\delta$ of all local $\delta_l$ values
   }
   while ($\delta$ > 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?).

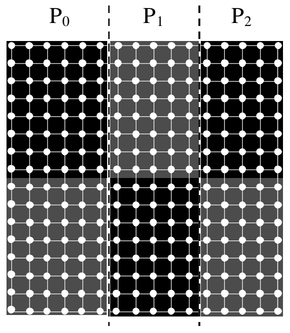


Figure 15: A coarse-grained red-black ordering.

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.

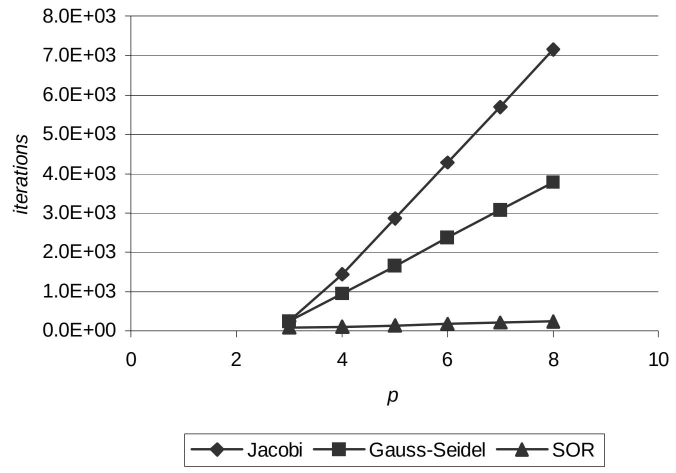


Figure 16: The number of iterations for the Jacobi, Gauss-Seidel, and SOR iteration as a function of the stopping condition for N = *40.*

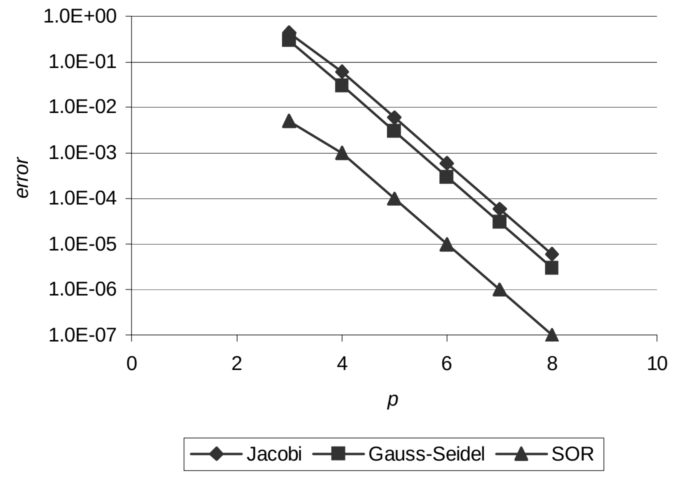


Figure 17: The mean error for the Jacobi, Gauss-Seidel, and SOR iteration as a function of the stopping condition for N = *40.*

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

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

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.

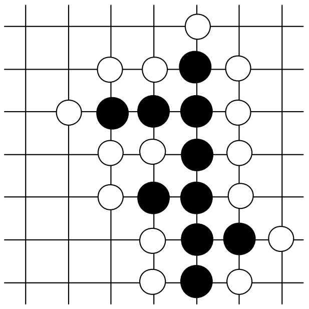


Figure 18: The object and possible growth sites.

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

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

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.

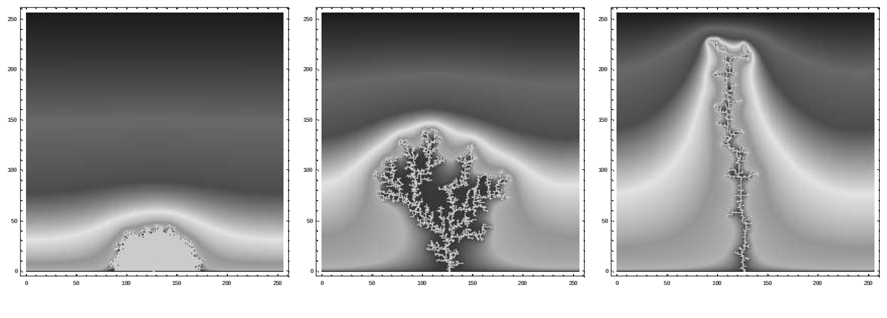


Figure 19: Results of DLA growth on a 256^2 lattice. The left figure is for $\eta = 0$, the middle for $\eta = 1.0$ and the right for $\eta = 2.0$.

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.

E. A Numerical Solver for Hydrodynamic Flow

E.1. Introduction

This section presents a more involved example of parallel simulation of incompressible hydrodynamic flow. Although the model itself is much more complicated than that for diffusion we can use many of the techniques introduced above to discretize the model and parallelize the resulting algorithms.

The flow of a time dependent incompressible fluid, a fluid with constant density as for example water under most conditions, can be described by the two basic equations from hydrodynamics:

$$\nabla \cdot V = 0\,, \qquad [21]$$

$$\frac{\partial V}{\partial t} = -(V \cdot \nabla)V - \nabla P + \nu \nabla^2 V\,. \qquad [22]$$

where V is the velocity, t time, P the pressure, and ν the kinematic viscosity which will be assumed to be constant. In the first equation the conservation of mass is expressed; it states that the density at a point in space can only change by in- or out- flow of matter. The second equation is the standard Navier-Stokes equation which expresses the conservation of momentum. This equation describes the velocity changes in time, due to convection $(V{\cdot}\nabla)V$, spatial variations in pressure ∇P, and viscous forces $\nu\nabla^2 V$. Both equations can be solved analytically only in a very few, simple, cases. In most cases these equations can only be solved by simulation. There is a wide variety of numerical methods available for solving these equations, which already indicates that there is still no perfect method within reach. A good overview of the numerical methods for solving the hydrodynamic equations can be found in [7].

E.2. The numerical solver: finite differencing

In this section it will be demonstrated how the hydrodynamic equations can be solved using the Marker-and-Cell technique [8]. This method is based on a finite difference approximation of the hydrodynamic equations. This method is chosen as an example for several reasons. It is conceptually a simple method, the original equations [21,22] are converted straightforward into finite difference equations, parallellization of the algorithm is relatively easy, extension to three dimensions is trivial, and most types of boundary conditions can relatively easy be specified. Eq. [22] can be written as a set of partial differential equations:

$$\frac{\partial u}{\partial t} = -\frac{\partial u^2}{\partial x} - \frac{\partial uv}{\partial y} - \frac{\partial P}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \qquad [23]$$

$$\frac{\partial v}{\partial t} = -\frac{\partial uv}{\partial x} - \frac{\partial v^2}{\partial y} - \frac{\partial P}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right). \qquad [24]$$

Eq. [21] can be written as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0\,, \qquad [25]$$

where V of Eqs. [21] and [22] is represented in two dimensions by the vector (u, v). The extension to 3D, where V is represented by a vector (u, v, w), is done analogous to the two-dimensional equations by adding a z-coordinate and an equation for $\partial w/\partial t$ in Eqs [23-25]. An additional equation for the pressure P, a Poisson equation, can be obtained by differentiation and addition of Eqs. [23,24]:

$$\nabla^2 P = -\frac{\partial^2 u^2}{\partial x^2} - 2\frac{\partial^2 uv}{\partial x \partial y} - \frac{\partial^2 v^2}{\partial y^2} - \frac{\partial D}{\partial t} + \nu\left(\frac{\partial^2 D}{\partial x^2} + \frac{\partial^2 D}{\partial y^2}\right), \qquad [26]$$

where D is the divergence:

$$D = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}. \qquad [27]$$

A feature of this equation is that in the mass conservation Eq. [25] it is stated that D has the value zero.

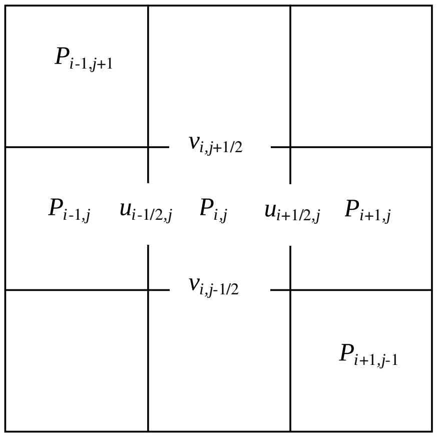


Figure 20: Diagram of the Marker-and-Cell structure, showing cells in the neighborhood of lattice position (i,j) *(after[7]).*

In the Marker-and-Cell technique the finite difference approximation is applied to solve the Eqs. [23-25]. In this method space is subdivided into cells with length Δx, Δy. In Figure 20 a diagram is shown of this structure. The velocities are located at the cell faces, while the pressure is located at the cell center. The cells are labeled with an index (i, j), $P_{i,j}$ is the pressure at the cell center, while $u_{i+1/2}$ denotes the velocity in the x-direction between the cells (i, j) and $(i+1, j)$ etc. The finite difference expressions of Eqs. [23, 24] are:

$$\begin{aligned} u^{n+1}_{i+1/2,j} = u_{i+1/2,j} + \Delta t\Bigg[&-\frac{1}{\Delta x}\left(u^2_{i+1,j} - u^2_{i,j}\right) \\ &-\frac{1}{\Delta y}\left((uv)_{i+1/2,j+1/2} - (uv)_{i+1/2,j-1/2}\right) \\ &-\frac{1}{\Delta x}\left(P_{i+1,j} - P_{i,j}\right) - \nu\Bigg(\frac{1}{\Delta x^2}(-u_{i+3/2,j} + 2u_{i+1/2,j} - u_{i-1/2,j}) \\ &-\frac{1}{\Delta y^2}(u_{i+1/2,j+1} - 2u_{i+1/2,j} + u_{i+1/2,j-1})\Bigg)\Bigg] \end{aligned} \qquad [28]$$

$$\begin{aligned} v_{i,j+1/2}^{n+1} = v_{i,j+1/2} + \Delta t \Bigg[& -\frac{1}{\Delta y}\left(v_{i,j+1}^2 - v_{i,j}^2\right) \\ & -\frac{1}{\Delta x}\left((uv)_{i+1/2,j+1/2} - (uv)_{i-1/2,j+1/2}\right) \\ & -\frac{1}{\Delta y}\left(P_{i,j} - P_{i,j+1}\right) + \nu\left(\frac{1}{\Delta x^2}(v_{i+1,j+1/2} + 2v_{i,j+1/2} + v_{i-1,j+1/2}) \right. \\ & \left. + \frac{1}{\Delta y^2}(v_{i+1,j+3/2} - 2v_{i,j+1/2} + v_{i,j-1/2})\right)\Bigg] \end{aligned} \qquad [29]$$

where $u_{i+1/2,j}^{n+1}$ and $v_{i,j+1/2}^{n+1}$ are the velocities obtained by advancing the previous velocities $u_{i+1/2,j}$ and $v_{i,j+1/2}$ one time step. Values like $u_{i+1,j}$ are obtained by using the average: $u_{i+1,j} = ½(u_{i+3/2,j} + u_{i+1/2,j})$ and product terms are evaluated as the product of averages:

$$(uv)_{i+1/2,j+1/2} = \frac{1}{2}(u_{i+1/2,j} + u_{i+1/2,j+1}) \cdot \frac{1}{2}(v_{i+1,j+1/2} + v_{i,j+1/2}) . \qquad [30]$$

The results $u_{i+1/2,j}^{n+1}$ and $v_{i,j+1/2}^{n+1}$ do not necessarily satisfy the mass conservation Eq. [25]. In the Marker and Cell method an iterative process is used in which the cell pressures are modified to obtain a value $D = 0$. For this purpose in each cell (i, j) the value of $D_{i,j}$, the finite difference form of Eq. [27] is determined:

$$D_{i,j} = \frac{1}{\Delta x}\left(u_{i+1/2,j} - u_{i-1/2,j}\right) + \frac{1}{\Delta y}\left(v_{i,j+1/2} - v_{i,j-1/2}\right). \qquad [31]$$

If the value $D_{i,j}$ is below a certain level *tolerance*, the flow is locally incompressible and it is not necessary to change the velocities at the cell face. If $D_{i,j}$ is above the level *tolerance*, the pressure is changed with a small value:

$$\Delta P_{i,j} = -\beta D_{i,j} \qquad [32]$$

where β is related to a relaxation factor β_0:

$$\beta = \beta_0 / 2\Delta t(1/\Delta x^2 + 1/\Delta y^2) . \qquad [33]$$

It is necessary to use a value $\beta_0 < 2$, otherwise the iteration process is not stable (compare this to the *Courant stability* derived in the previous sections). More details about the finite difference iterations using relaxation factors and its stability can be found in [3] and [9]. A plausible value for β_0 is 1.7. Once $\Delta P_{i,j}$ is determined for each cell (i,j) it is necessary to add it to $P_{i,j}$ and to adjust the velocity components at the faces of the cell (i, j):

$$\begin{aligned} u_{i+1/2,j} &\rightarrow u_{i+1/2,j} + (\Delta t / \Delta x)\Delta P_{i,j} \\ u_{i-1/2,j} &\rightarrow u_{i-1/2,j} - (\Delta t / \Delta x)\Delta P_{i,j} \\ v_{i,j+1/2} &\rightarrow v_{i,j+1/2} + (\Delta t / \Delta y)\Delta P_{i,j} \\ v_{i,j-1/2} &\rightarrow v_{i,j-1/2} - (\Delta t / \Delta y)\Delta P_{i,j} \end{aligned} . \qquad [34]$$

The process is repeated successively in all cells until no cell has a divergence D greater than *tolerance*. The complete iteration process is summarized in Algorithm 6.

When the iteration process converges and the mass conservation equation is satisfied, a next time step can be done. It can be demonstrated [10] that adjusting P and V in this iterative process is equivalent with solving the Poisson Equation [26].

```
Advance the finite difference Eqs. 28 and 29 one time step $\Delta t$
do
   determine $D_{i,j}$ (Eq. 21) for each cell
   if ($D_{i,j}$ > tolerance) then
      determine $\Delta P_{i,j}$ (Eq. 22)
      add $\Delta P_{i,j}$ to $P_{i,j}$
   adjust velocity components (Eq. 24)
   end if
while (there are cells present with $D$ > tolerance)
goto next time step
```

Algorithm 6: Pseudo code of the Marker and Cell method.

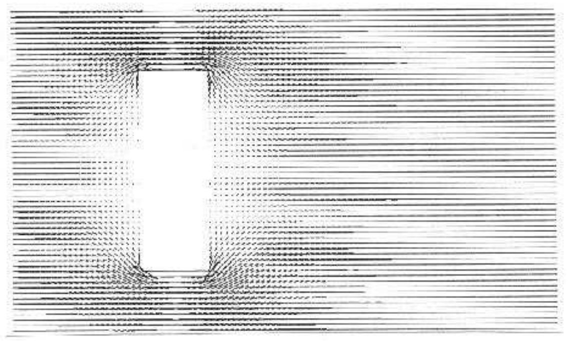


Figure 21: A 2D slice of a flow pattern about an obstacle for a large value of ν = 1.0 in a 3D lattice consisting of 100^3 cells.

Before Eqs. [23-25] can be solved it is necessary to specify the boundary conditions, otherwise the problem is ill posed. Five common types of boundary conditions are: rigid free-slip walls, rigid no-slip walls, inflow, outflow boundaries, periodic boundaries, moving boundaries, free boundaries (for example a moving water surface). The specification of these boundary conditions, in general, is a difficult problem especially in the case of complex geometry (see [7]). If the value of ν of the kinematic viscosity is chosen large enough the equations are stable and it is possible to compute the flow pattern about an obstacle. In Figure 21 the flow pattern about a rectangular obstacle is determined. The flow, in this example, is directed in the positive x-direction. In this example four types of boundary conditions are used. At the cells situated at the obstacle the "rigid no-slip wall" condition is used by setting the u and v component to the value zero. There is an inflow boundary where the velocity is set a certain input value. Finally, there is an outflow boundary where the velocity V^{n+1} in cell (i, j) is set to the V^{n+1} velocity of the neighboring upstream cell $(i\text{-}1, j)$, while for the two other borders of the lattice periodic boundary conditions are used. If ν is chosen too low, the convective term $(V \cdot \nabla)V$ in Eq. 22 starts to dominate. For low values of ν the iteration process becomes unstable, furthermore truncation errors, which are unavoidable in finite difference approximations, may lead to instabilities. More details about the stability analysis, in which the lower limit of ν is determined, of the iteration process shown can be found in [8]. An example of a flow pattern generated with a lower value of ν is shown in Figure 22. In this example the same type of boundary conditions are used as in the previous example.

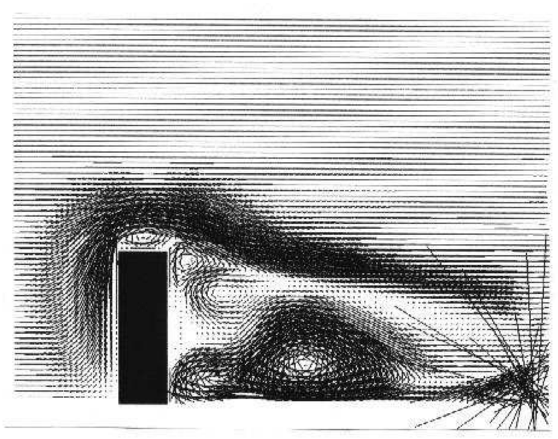


Figure 22: Example of a flow pattern about an obstacle for a low value of ν *= 0.20 in a lattice consisting of* 100^3 *cells.*

In this example it can be seen that eddies start to develop behind the obstacle and furthermore an instability develops (which can be recognized by the strange size of the velocity vectors), due to truncation errors, at the very right of the picture.

In many papers on hydrodynamics the range of ν that can be simulated with the model, before instabilities occur, is expressed in the range of possible Reynolds numbers. The relation between ν and the Reynolds number Re is given by the equation:

$$\mathrm{Re} = \frac{V_0 L}{\nu}, \qquad [35]$$

where V_0 is the typical velocity of the system (for example the input velocity) and L the typical dimension resolved by the system. In the experiments shown in Figure 21 and Figure 22 a plausible choice of L is the height of the obstacle (number of cells necessary to represent the height times Δy. The choice of L depends on the geometry of the obstacle, more details can be found in the references [7]. Especially for an irregular geometry a plausible choice of L can become problematic.

E.3. The parallel implementation

From the finite difference Eqs.[28, 29, 31] used in the algorithm of the Marker and Cell method it can be seen that for the computation of the values of u^{n+1}, v^{n+1}, and D^{n+1} only the u, v, P values of the 8 nearest neighbors are required (see Figure 20). This locality can be used in a parallel implementation of the Marker and Cell algorithm. In this algorithm a successive updating scheme is used, where the P-values (and also the u and v values) of the cells neighboring cell (i, j) are in the states of Eq. [36],

$$\begin{matrix} P^{n}_{i-1,j+1} & P^{n}_{i,j+1} & P^{n}_{i+1,j+1} \\ P^{n+1}_{i-1,j} & P^{n}_{i,j} & P^{n}_{i+1,j} \\ P^{n+1}_{i-1,j-1} & P^{n+1}_{i,j-1} & P^{n+1}_{i+1,j-1} \end{matrix} . \qquad [36]$$

In a parallel implementation this locality can be used and the lattice of cells can be subdivided into a, square grid where each grid site is located on a different processor. In Figure 23 the situation is depicted for a lattice which is subdivided into four parts and where the lattice is mapped onto a grid of 2×2 processors. Each processor is bordered by boundary strips and corners which contain copies of the values of the adjacent cells. In the parallel version of Algorithm 6 a four-colored checkerboard domain partitioning [11] is used which accounts for the correct state (n or $n + 1$) of the adjacent cells as shown in Eq.

36, when cell (i, j) is updated. In the parallel version of Algorithm 6 the lattice of cells is updated in four successive phases. This is shown in Algorithm 7.

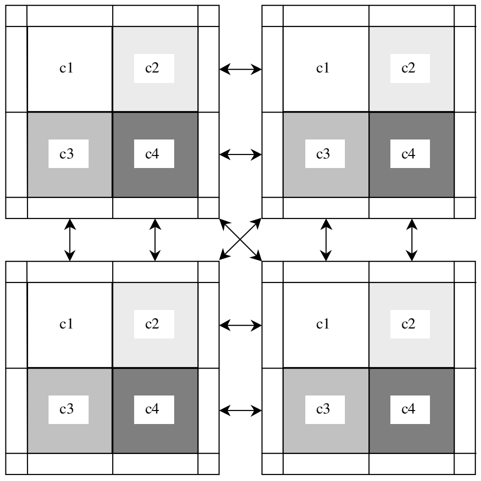


Figure 23: Mapping of the lattice of cells onto a grid of 2×2 processors using a four-colored checkerboard domain partitioning. The arrows indicate the exchange of boundary strips and boundary corners.

```
for each time step do
   exchange boundary strips between regions $c1$ and $c2$
   exchange boundary strips between regions $c1$ and $c3$
   exchange boundary corners between regions $c1$ and $c4$
   update of region $c1$ using algorithm 6
   exchange boundary strips between regions $c2$ and $c1$
   exchange boundary strips between regions $c2$ and $c4$
   exchange boundary corners between regions $c2$ and $c3$
   update of region $c2$ using algorithm 6
   exchange boundary strips between regions $c3$ and $c1$
   exchange boundary strips between regions $c3$ and $c4$
   exchange boundary corners between regions $c3$ and $c2$
   update of region $c3$ using algorithm 6
   exchange boundary strips between regions $c4$ and $c2$
   exchange boundary strips between regions $c4$ and $c3$
   exchange boundary corners between regions $c4$ and $c1$
   update of region $c4$ using algorithm 6
endfor
```

Algorithm 7: Pseudo code for a parallel numerical solver of the hydrodynamical equations

References

1. Hoekstra, A.G.: Syllabus APR Part 1 : Introduction to Parallel Computing. University of Amsterdam, (1999)
2. Stoer, J.,Bulirsch, R.: Introduction to Numerical Analysis.
3. Press, W.H., Flannery, B.P., Teukolsky, S.A.,Verrerling, W.T.: Numerical Recipes in C, the art of scientific computing.
4. Fox, G.C., Williams, R.D.,Messina, P.: Parallel Computing Works! (1994)

5. Dongarra, J.J.,Walker, D.W.: Constructing Numerical Software Libraries for High Performance Computer Environments. In: A. Zomaya (eds.): Parallel & Distributed Computing Handbook. (1998)
6. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C.,Van der Vorst, H.v.d.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. Philadelphia, PA (1994)
7. Roache, P.J.: Computational Fluid Dynamics. Hermosa Publishers, Albequerque (1976)
8. Hirt, C.W.,Cook, J.L.: Calculating three-dimensional flow around structures and over rough terrain. J. Comput. Phys. **10** (1972) 324-340
9. Ames, W.F.: Numerical Methods for Partial Differential Equations. Academic Press, New York (1977)
10. Viecelli, J.A.: A computing method for incompressible flows bounded by moving walls. J. Comput. Phys. **8** (1971) 119-143
11. Fox, G.C., Johnson, M., Lyzenga, G., Otto, S., Salmon, J.,Walker, D.: Solving Problems on Concurrent Processors. Pretince Hall, (1988)
12. Wilkinson, J.H.: Error analysis of direct methods of matrix inversion. J. Assoc. Comp. Mach. **8** (1961) 281 - 330
13. Lawson, C., Hanson, R., Kincaid, D.,Krogh, F.: Basic Linear Algebra Subprograms for FORTRAN usage. ACM TOMS (Transactions On Math. Software) **5** (1979) 308-325
14. Golub, G.H.,Loan, C.F.v.: Matrix Computations. John Hopkins Univ. Press, (1989)
15. Freeman, T.L.,Philips, C.: Parallel Numerical Algorithms. Prentice Hall, (1992)