Fast Solution Methods and Parallelization for Computational Science Applications

Lecture Notes for course (wi4145TU)
"Computational Science and Engineering"

by: C.W. Oosterlee, H. Bijl, H.X. Lin, S.W. de Leeuw,
J.B. Perot, C. Vuik, and P. Wesseling

Contents

1 Introduction

Computational Science and Engineering (CSE) is a rapidly growing multidisciplinary area with connections to the sciences, engineering, mathematics and computer science. CSE focuses on the development of problem-solving methodologies and robust tools for the solution of scientific and engineering problems. We believe that CSE will play an important if not dominating role for the future of the scientific discovery process and engineering design. Below we give a more detailed description of Computational Science and Engineering. For more information on CSE we refer to the Society for Industrial and Applied Mathematics (SIAM).

In CSE we deal with the simulation of processes, as they occur amongst others in the engineering, physical and economic sciences. Examples of application areas are fluid dynamics (the aerodynamics of cars and aircrafts, combustion processes, pollution spreading), semi-conductor technology (breeding of cristals, oxidation processes), weather and climate prediction (the growing and tracking of tornados, global warming), applied physics (many particle simulations, protein folding, drug design) but also financial mathematics (prediction of stock and option prices). Simulation is nowadays an equal and indispensable partner in the advance of scientific knowledge next to the theoretical and experimental research.

It is characteristic for CSE that practical relevant results are typically achieved by combining methods and research results from different scientific areas.

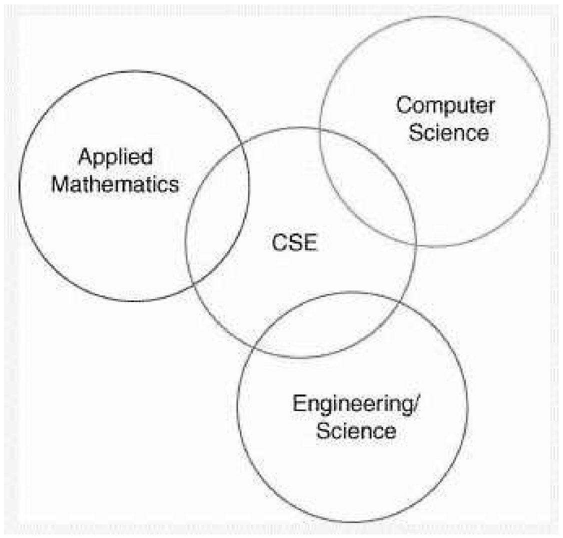


Figure 1.0.1: Embedding of computational science and engineering.

The application area brings the typical problem-dependent know-how, the knowledge to formulate the problem and model and to verify the computed results by means of real experiments. Applied mathematics deals with the definition of a mathematical model, existence and uniqueness results and develops efficient methods to solve the mathematical model with a computer. Computer science typically takes care for the usability, and programming of modern computers and designs powerful software packages. Especially the interplay of the disciplines is necessary for success. For instance: it is possible to parallelize a poor mathematical method and implement it on a supercomputer, but this does not help us much! We need much more a continuing development of mathematical models and numerical algorithms, the transfer of the algorithms into powerful and user-friendly software, running this on state-of-the-art computers that steadily increase their performance.

Basically, we can distinguish a number of important steps in simulation:

- Setting up a model.
  At the start of each simulation we have the development of a mathematical model of the process of interest. This must be a simplified image of the reality that contains many relevant phenomena. It must be formulated such that the model has a (unique) solution. Often we obtain a system of (not analytically solvable) differential equations.

- The analytical treatment.
  Analytical tools can be used to obtain properties of the solution: existence, uniqueness, maximum principle etc. Furtermore for simple problems an analytical solution can be found. In a number of cases approximate solutions can be derived using an asymptotical approach.

- The numerical treatment.
  Since a computer can only handle discrete numbers, we have to discretize the model (the equations and the solution domain), so that the computer can deal with them. For the solution of the resulting matrix from the discrete equations, mathematicians provide efficient methods.

- The implementation.
  Next to the choice of computer language and data structures, especially the distributed computation is of major importance.

- The embedding.
  The numerical simulation is just one step in the product development in industrial applications. Therefore, we need interfaces, so that we can link the simulation programme with CAD tools. Only in this way it is, for example, possible to use aerodynamics car simulation results on drag coefficients at an early stage into the design process.

- The visualisation.
  Typically, after a simulation we have huge data sets (not only one drag coefficient as in the example above). Often we need the velocity in the complete flow domain, or one is interested for the optimal control of a

robot in the path of the robot arm. We need to present such results with the help of computer graphics, so visualization is very important.

- The validation.
  After long computer times with many many computations, we have the result of a simulation. It is of a primary importance to verify the results obtained.

2 Iterative Solution Methods for Partial Differential Equations

2.1 Notation and Initial Examples

We start with the classical formulation of differential equations with differential (and boundary) operators. For discretization, we use the *discrete differential operators*. The grid–oriented notation emphasizes clearly the correspondence between the discrete and continuous problems.

2.1.1 Continuous and Discrete Boundary Value Problems

We denote scalar linear boundary value problems by

$$\begin{aligned} \mathcal{L}^{\Omega} u(\boldsymbol{x}) &= b^{\Omega}(\boldsymbol{x}) \qquad && (\boldsymbol{x} \in \Omega) \\ \mathcal{L}^{\Gamma} u(\boldsymbol{x}) &= b^{\Gamma}(\boldsymbol{x}) \qquad && (\boldsymbol{x} \in \Gamma := \partial\Omega). \end{aligned} \tag{2.1.1}$$

Here $\boldsymbol{x} = (x_1, \ldots, x_d)^T$ and $\Omega \subset \mathbb{R}^d$ is a given open domain with boundary Γ. $\mathcal{L}^{\Omega}$ is a linear differential operator on Ω and $\mathcal{L}^{\Gamma}$ represents one or several linear boundary operators. b^{Ω} denotes a given function on Ω and b^{Γ} one or several functions on Γ. We always denote solutions of (2.1.1) by $u = u(\boldsymbol{x})$. For brevity, we also simply write $\mathcal{L}u = b$ instead of (2.1.1). In the case $d = 2$ or $d = 3$, we will usually write (x, y) and (x, y, z).

The discrete analog of (2.1.1) is denoted by

$$\begin{aligned} L_h^{\Omega} u_h(x,y) &= b_h^{\Omega}(x,y) \qquad && ((x,y) \in \Omega_h) \\ L_h^{\Gamma} u_h(x,y) &= b_h^{\Gamma}((x,y)) \qquad && ((x,y) \in \Gamma_h). \end{aligned} \tag{2.1.2}$$

Here h is a (formal) discretization parameter. Using the infinite grid

$$\boxed{\mathbf{G}_h := \{(x,y) : x = ih_x, y = jh_y;\ i, j, \in \mathbb{Z}\}}\ , \tag{2.1.3}$$

where $\boldsymbol{h} = (h_x, h_y)$ is a vector of fixed mesh sizes, we define $\Omega_h = \Omega \cap \mathbf{G}_h$ and Γ_h the set of discrete intersection points of the "grid lines" with Γ. In the special case of square grids, we identify $h = h_x = h_y$.

The discrete solution u_h is a function defined on the grid $\Omega_h \cup \Gamma_h$, i.e., a grid function, and b_h^{Ω} and b_h^{Γ} are discrete analogs of b^{Ω} and b^{Γ}, that is, b^{Ω}, respectively b^{Γ}, restricted to the grid. Instead of $u_h(x,y) = u_h(ih_x, jh_y)$, we sometimes write $u_{i,j}$.

L_h^{Ω} and L_h^{Γ} are grid operators, i.e., mappings between spaces of grid functions. (L_h^{Ω} is also called a *discrete* (differential) or *difference operator*, L_h^{Γ} a *discrete boundary operator*.)

Clearly the concrete definitions of Ω_h, Γ_h etc. also depend on

– the given PDE,
– the domain Ω,
– the boundary conditions,
– the grid approach and
– the discretization.

For ease of presentation, we will first assume that discrete boundary equations are eliminated from (2.1.2). We then simply write

$$L_h u_h = b_h \quad (\Omega_h) \ . \tag{2.1.4}$$

Here u_h and b_h are grid functions on Ω_h and L_h is a linear operator

$$L_h : \mathcal{G}(\Omega_h) \to \mathcal{G}(\Omega_h) \ , \tag{2.1.5}$$

where $\mathcal{G}(\Omega_h)$ denotes the *linear space of grid functions on* Ω_h. Clearly, (2.1.4) can be represented as a system of linear algebraic equations.

2.1.2 Stencil Notation

For the concrete definition of discrete operators L_h^Ω (on a Cartesian grid) the stencil terminology is convenient. We restrict ourselves to the case $d = 2$. On the infinite grid $\mathbf{G}_h$, we consider grid functions

$$\begin{aligned} w_h : \mathbf{G}_h &\longrightarrow \mathbb{R} \quad (\text{or } \mathbb{C}) \\ (x,y) &\longmapsto w_h(x,y). \end{aligned}$$

A general stencil $[s_{\kappa_1\kappa_2}]_h$

$$[s_{\kappa_1\kappa_2}]_h = \begin{bmatrix} & \vdots & \vdots & \vdots & \\ \cdots & s_{-1,1} & s_{0,1} & s_{1,1} & \cdots \\ \cdots & s_{-1,0} & s_{0,0} & s_{1,0} & \cdots \\ \cdots & s_{-1,-1} & s_{0,-1} & s_{1,-1} & \cdots \\ & \vdots & \vdots & \vdots & \end{bmatrix}_h \qquad (s_{\kappa_1\kappa_2} \in \mathbb{R})$$

defines an operator on the set of grid functions by

$$\boxed{[s_{\kappa_1\kappa_2}]_h w_h(x,y) = \sum_{(\kappa_1,\kappa_2)} s_{\kappa_1\kappa_2} w_h(x + \kappa_1 h_x,\ y + \kappa_2 h_y)} \quad , \tag{2.1.6}$$

where w_h is a grid function. We assume that a *finite number of coefficients* $s_{\kappa_1\kappa_2}$ *are non–zero.* The coefficients $s_{\kappa_1\kappa_2}$ usually depend on h.

Many of the stencils will be 5–*point* or *compact* 9–*point* stencils

$$\begin{bmatrix} & s_{0,1} & \\ s_{-1,0} & s_{0,0} & s_{1,0} \\ & s_{0,-1} & \end{bmatrix}_h , \quad \begin{bmatrix} s_{-1,1} & s_{0,1} & s_{1,1} \\ s_{-1,0} & s_{0,0} & s_{1,0} \\ s_{-1,-1} & s_{0,-1} & s_{1,-1} \end{bmatrix}_h . \tag{2.1.7}$$

The discrete operators L_h^Ω usually are given on a finite grid Ω_h. For the identification of a discrete operator L_h^Ω with "its" stencil $[s_{\kappa_1\kappa_2}]_h$, we in general have to restrict the stencil to Ω_h (instead of $\mathbf{G}_h$). Near boundary points the stencils may have to be modified.

2.1.3 Types of PDEs

We will give the usual classification of second order scalar 2D PDEs here. Generalizations of this classification to 3D, higher order equations or systems of PDEs can be found, for example, in [53]. We consider equations $\mathcal{L}u = b$ in some domain $\Omega \in \mathbb{R}^2$ where

$$\mathcal{L}u = a_{11}u_{xx} + a_{12}u_{xy} + a_{22}u_{yy} + a_1u_x + a_2u_y + a_0u \quad (\Omega) \ , \tag{2.1.8}$$

with coefficients a_{ij}, a_i, a_0 and a right-hand side f which, in general, may depend on x, y, u, u_x, u_y (the quasilinear case). $\mathcal{L}$ is called

- elliptic if $4a_{11}a_{22} > a_{12}^2$
- hyperbolic if $4a_{11}a_{22} < a_{12}^2$
- parabolic if $4a_{11}a_{22} = a_{12}^2$

This classification, in general, depends on (x, y) and, in the nonlinear case, also on the solution u. Prototypes of the above equation are

- the Poisson equation $-\Delta u = -u_{xx} - u_{yy} = b$,
- the wave equation $u_{xx} - u_{yy} = 0$ and
- the heat conduction equation $u_{xx} - u_y = 0$.

Most of our basic presentation is oriented to Poisson's equation. Other important elliptic model equations are

- the anisotropic model equation $-\varepsilon u_{xx} - u_{yy} = b$,
- the convection-diffusion equation $-\varepsilon\Delta u + a_1u_x + a_2u_y = b$,
- and the equation with mixed derivatives $-\Delta u + \tau u_{xy} = b$.

All these equations serve as model equations for special features and complications and are thus representative for a larger class of problems with similar features. These model equations depend on a parameter ε or τ in a crucial way. For certain parameter values we have a singular perturbation: The type of the equation changes and the solution behaves qualitatively different (if it exists at all). For instance, the anisotropic equation becomes parabolic for $\varepsilon \to 0$, the equation with mixed derivatives is elliptic for $|\tau| < 2$, parabolic for $|\tau| = 2$ and hyperbolic for $|\tau| > 2$. All the model equations represent classes of problems which are of practical relevance.

2.1.4 Grids and Discretization Approaches

The differential equations to be solved need to be *discretized on a suitable grid* (or, synonymously, mesh). Here, we give a rough survey (with some examples) of those types of grids which are used in practice.

These are

- *Cartesian grids*,
- *boundary-fitted, logically rectangular grids*,
- *block-structured boundary-fitted grids*.

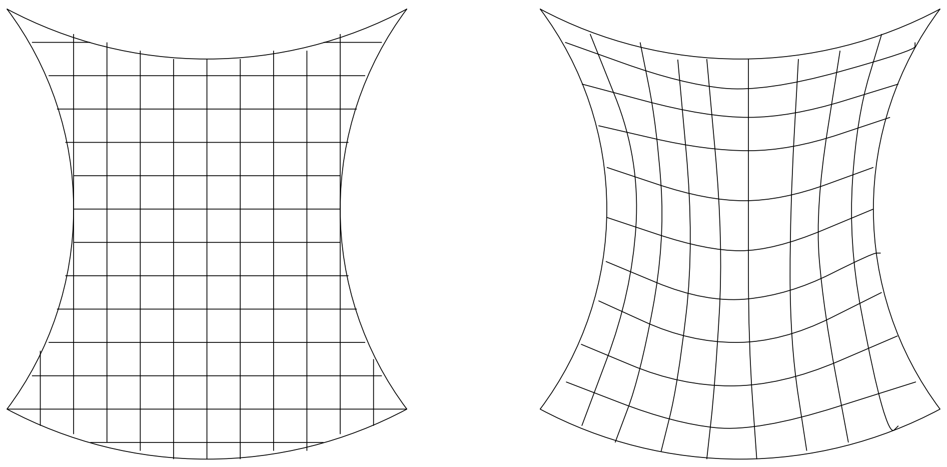


Figure 2.1.1: A square Cartesian grid (left picture) in a domain Ω and a boundary-fitted (logically rectangular) grid (right picture)

Figure 2.1.1 is an example of a Cartesian grid. For simple domains Ω and simple boundaries $\partial\Omega$, Cartesian grids are numerically convenient.

Figure 2.1.1 gives also an example of a boundary-fitted grid. A systematic introduction to boundary-fitted grids is given in [136].

In the context of boundary-fitted grids, there are two different common approaches: In the first, coordinate transformations are used to obtain simple, rectangular domains, and correspondingly simple grids. Here the differential (and/or the discrete) equations are transformed to the new curvilinear coordinates. In the second approach, the computations are performed in the physical domain with the original (non–transformed) equations.

Block-structured boundary-fitted grids are used if the given domain cannot (or cannot reasonably) be mapped to a rectangular domain, but can be decomposed into a finite number of subdomains each of which can be covered with a boundary-fitted grid. Quite complicated domains may be treated with this approach, as can be seen in Figure 2.1.2.

In many software packages, *unstructured, irregular grids* are used today. These grids have become quite popular because unstructured automatic mesh generation

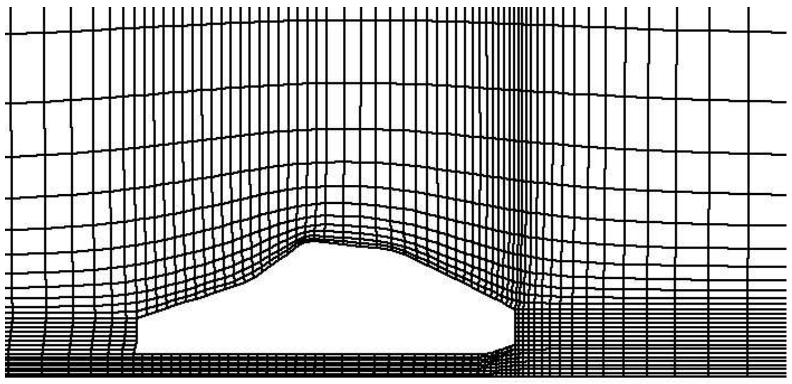


Figure 2.1.2: Boundary-fitted block-structured grid around a car.

is much easier than the generation of block-structured grids for very complicated 2D and 3D domains.

An example of an unstructured grid is the grid around a car during a crash simulation. A part of this grid is presented in Figure 2.1.3.

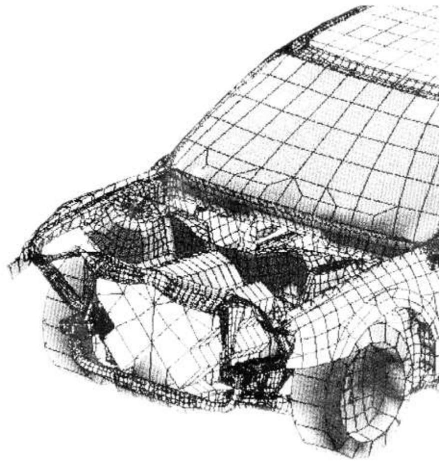


Figure 2.1.3: Unstructured grid around a car in a crash simulation application

Although unstructured grids are widely used today, even complicated domains allow other than *purely* unstructured grid approaches. Often a *hybrid* approach, an unstructured grid close to the boundary and a structured grid in the interior part of the domain (or vice versa) is suited for the treatment of such complicated domains.

More generally, all types of grids mentioned above can be used in the context of *composite grids*.

A typical situation for the use of composite grids is that an overall Cartesian grid is combined with a local boundary-fitted grid. An example of this approach is shown in Figure 2.1.4. Such composite grids differ from block-structured grids in that they usually consist of overlapping subgrids (Chimera technique).

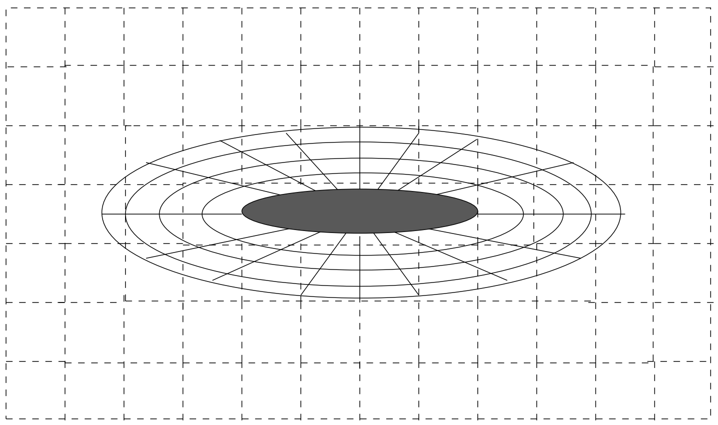


Figure 2.1.4: A simple composite grid

Finally, we mention the *self-adaptive grids*, here. Self-adaptive grids are constructed automatically during the solution process according to an error estimator that takes the behavior of the solution into account. For an example see Figure 2.1.5.

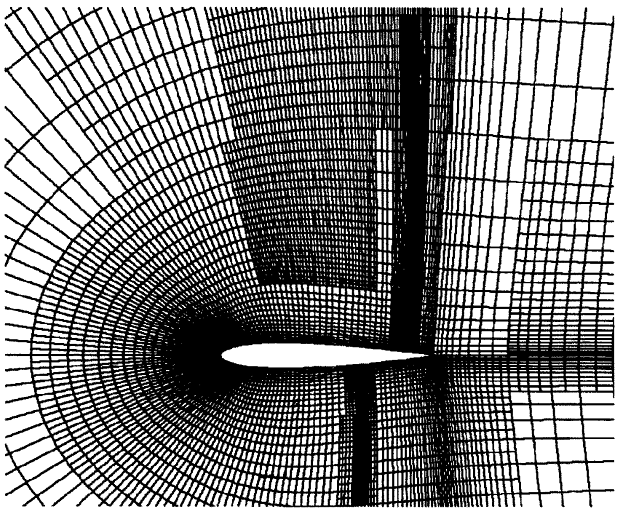


Figure 2.1.5: Adaptively refined grid around an airfoil

In principle, any type of grid can be used with any type of *discretization approach*. In practice, however, finite difference and finite volume methods are traditionally used in the context of Cartesian, logically rectangular and block-structured boundary-fitted grids, whereas finite elements and also finite volumes are widely used in the context of unstructured grids.

Another important choice to be made in discretization is the *arrangement of the unknowns within a grid*. The unknowns can be defined at the vertices of a grid (*vertex-centered location of unknowns*). Another option is the choice of the unknowns at cell centers (*cell-centered location of unknowns*).

For systems of PDEs it is possible to choose different locations for different types of unknowns. A well-known example is the *staggered grid* for the system of the incompressible Navier-Stokes equations, in which pressure unknowns are placed at the cell centers and velocity components at cell boundaries. Examples of a vertex-centered, a cell-centered and a staggered Cartesian grid are sketched in Figure 2.1.6. Often, the discrete equations are defined at the same locations as the unknowns. It is hard to say which location of unknowns and which location

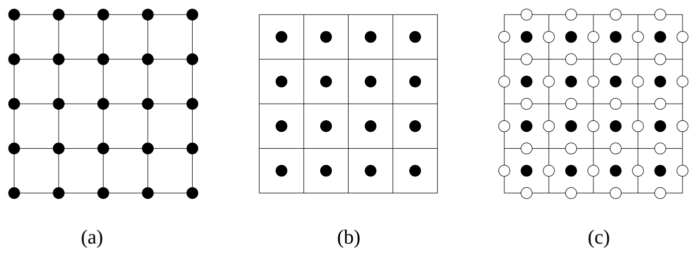


Figure 2.1.6: Three arrangements of unknowns in a Cartesian grid: a) a vertex-centered grid, b) a cell-centered grid, c) a staggered grid.

of the discretization is best in general. Often these choices depend on the type of boundary conditions and on the application. In the following chapters we mainly present results for vertex-centered locations of unknowns.

2.1.5 Poisson's Equation, Matrix Terminology

We introduce Model Problem 1, Poisson's equation in 2D.

The discrete Model Problem 1 is the classical model for a discrete elliptic boundary value problem. Every numerical solver has been applied to this problem for comparison.

Model Problem 1 *We will study in detail the discrete Poisson equation with Dirichlet boundary conditions*

$$\begin{aligned} -\Delta_h u_h(x,y) &= b_h^{\Omega}(x,y) \quad \Big((x,y) \in \Omega_h\Big) \\ u_h(x,y) &= b_h^{\Gamma}(x,y) \quad \Big((x,y) \in \Gamma_h = \partial\Omega_h\Big) \end{aligned} \tag{2.1.9}$$

in the unit square $\Omega = (0,1)^2 \subset \mathbb{R}^2$ *with* $h = 1/n,\ n \in \mathbb{N}$. *Here,* $_h = -\Delta_h$ *is the standard 5–point* $O(h^2)$*–approximation of the partial differential operator* $\mathcal{L}$,

$$\mathcal{L}u = -\Delta u = -u_{xx} - u_{yy} \tag{2.1.10}$$

on the square grid $\mathbf{G}_h$.

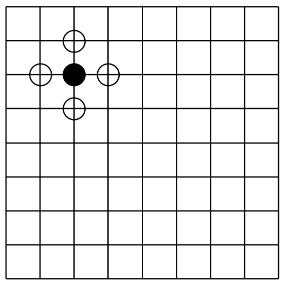
$O(h^2)$ here means that one can derive consistency relations of the form

$$\mathcal{L}u - L_h u = O(h^2) \qquad \text{for } h \to 0$$

for sufficiently smooth functions u ($u \in C_4(\overline{\Omega})$, for example), i.e. the leading terms in an asymptotic expansion with respect to h are proportional to h^2.

To illustrate what the elimination of boundary conditions means, we consider the following example.

Example 2.1.1 The *discrete* Poisson equation with eliminated Dirichlet boundary conditions *can formally be written in the form (2.1.4). For $(x, y) \in \Omega_h$ not adjacent to a boundary this means:*



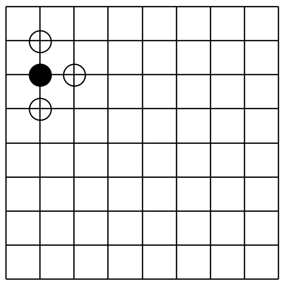
$$\begin{aligned}
b_h(x,y) &= b^\Omega(x,y)\\
L_h u_h(x,y) &= -\Delta_h u_h(x,y)\\
&= \frac{1}{h^2}[4u_h(x,y) - u_h(x-h,y) - u_h(x+h,y)\\
&\qquad\qquad -u_h(x,y-h) - u_h(x,y+h)]\\
&= \frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h u_h(x,y)
\end{aligned}$$

The notation

$$-\Delta_h = \frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h$$

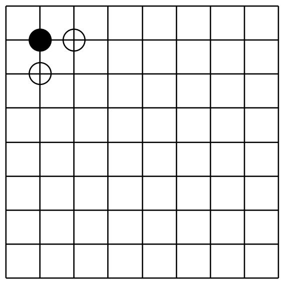
is a first example for the stencil notation.

For $(x, y) \in \Omega_h$ adjacent to a (here: west) boundary, L_h (2.1.4) reads



$$\begin{aligned}
b_h(x,y) &= b^\Omega(x,y) + \frac{1}{h^2} b^\Gamma(x-h,y)\\
L_h u_h(x,y) &= \frac{1}{h^2}\Big[4u_h(x,y) - u_h(x+h,y)\\
&\quad -u_h(x,y-h) - u_h(x,y+h)\Big]\\
&= \frac{1}{h^2}\begin{bmatrix} & -1 & \\ 0 & 4 & -1 \\ & -1 & \end{bmatrix}_h u_h(x,y)\ .
\end{aligned}$$

For $(x, y) \in \Omega_h$ in a (here: the northwest) corner we have



$$
\begin{aligned}
b_h(x,y) &= b^{\Omega}(x,y) + \frac{1}{h^2}\Big[b^{\Gamma}(x-h,y) + b^{\Gamma}(x,y+h)\Big]\\
L_h u_h(x,y) &= \frac{1}{h^2}\Big[4u_h(x,y) - u_h(x+h,y) - u_h(x,y-h)\Big]\\
&= \frac{1}{h^2}\begin{bmatrix} & 0 & \\ 0 & 4 & -1 \\ & -1 & \end{bmatrix}_h u_h(x,y) \ .
\end{aligned}
$$

In this example, elimination of the boundary conditions is simple. Often, elimination of boundary conditions may be complicated and not preferable.

Discrete operators L_h are often represented by *matrices* A. Each matrix row then represents connections of one unknown in the discretization of a PDE to its neighboring unknowns. Which of the matrix entries is different from 0, depends on the ordering of grid points, i.e., the ordering of the components of the vector of unknowns.

As an example we consider Model Problem 1. For a column- or row-wise ordering of grid points (also called *lexicographical* ordering, see Figure 2.1.7a, starting with points at the left lower corner) and eliminated Dirichlet boundary conditions, the resulting matrix is a block tri-diagonal matrix with a regular sparsity pattern:

$$
A = \frac{1}{h^2}\begin{pmatrix} T & -I & & \\ -I & T & -I & \\ & -I & T & -I \\ & & -I & T \end{pmatrix} \ , \tag{2.1.11}
$$

where

$$
I = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & -1 & 4 & -1 \\ & & -1 & 4 \end{pmatrix} \ .
$$

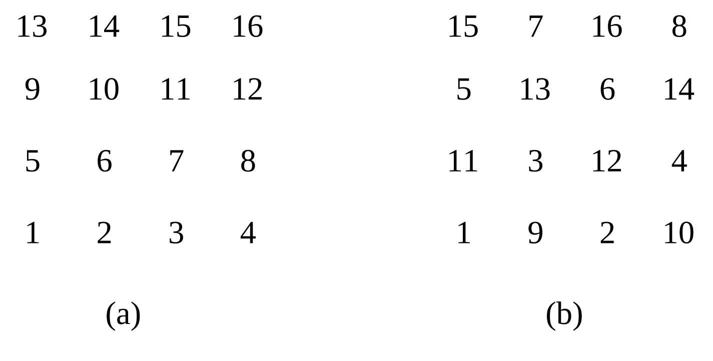


Figure 2.1.7: An example of a lexicographic (a) and a red-black (b) ordering of grid points.

Due to the elimination of Dirichlet boundary points, every matrix row corresponding to a grid point near a boundary has only three or two entries connecting them to neighboring grid points. This can be seen, for example, in the first rows of the matrix, where only two or three -1 entries can be found instead of four for interior points.

The dependence of the matrix structure on the ordering of the grid points can be observed when we write the matrix down for a *red-black ordering* of the grid points (see Figure 2.1.7b). First all unknowns at odd (red) grid points are considered, then the unknowns at the even (black) points. The corresponding matrix A is now a block matrix with blocks A_{rr}, representing the connections of the red grid points to red grid points, A_{rb} the connections of the red points to the black points, A_{br} the connections of black points to red points, and A_{bb} the connections of black points to black points. So:

$$A = \begin{bmatrix} A_{rr} & A_{rb} \\ A_{br} & A_{bb} \end{bmatrix} \tag{2.1.12}$$

For the five point discretization considered above, the blocks A_{rr} and A_{bb} are diagonal matrices with $4/h^2$ as the diagonal elements. The resulting block matrix $A_{rb}(= A_{br}^T)$ of Example 2.1.1 is

$$A_{rb} = \frac{1}{h^2}\begin{pmatrix} -1 & 0 & -1 & & & & & \\ -1 & -1 & 0 & -1 & & & & \\ -1 & 0 & -1 & -1 & -1 & & & \\ & -1 & 0 & -1 & 0 & -1 & & \\ & & -1 & 0 & -1 & 0 & -1 & \\ & & & -1 & -1 & -1 & 0 & -1 \\ & & & & -1 & 0 & -1 & -1 \\ & & & & & -1 & 0 & -1 \end{pmatrix} . \tag{2.1.13}$$

In order to put the iterative methods for the solution of linear equations in a readable form and to analyze the convergence criteria, we can use the matrix notation.

Let u, v, f, r, s etc. be N-dimensional vectors and $A = (a_{mn})$, $\widehat{A} = (\widehat{a}_{mn})$, Q (N,N)-matrices, I is the (N,N)-Identity matrix, 0 the (N,N)-Zero matrix.

We write

$$\begin{array}{rll} u \le v, & \text{if} \quad u_m \le v_m & (m = 1, \ldots, N) \ , \\ u > v, & \text{if} \quad u_m > v_m & (m = 1, \ldots, N) \ , \\ A \le \widehat{A}, & \text{if} \quad a_{mn} \le \widehat{a}_{mn} & (m, n = 1, \ldots, N); \end{array}$$

$$|u| = \begin{pmatrix} |u_1| \\ \vdots \\ |u_N| \end{pmatrix} , \qquad |A| = (|a_{mn}|) \ .$$

2.1.6 Exercises

Exercise 2.1.1 Show that the operator $\mathcal{L}$ with mixed derivatives in the equation $-\Delta u + \tau u_{xy} = b$ is elliptic for $|\tau| < 2$, parabolic for $|\tau| = 2$ and hyperbolic for $|\tau| > 2$.

Exercise 2.1.2 Determine the type of each of the following PDEs for $u = u(x, y)$.

a) $u_{xx} + u_{yy} = b$
b) $-u_{xx} - u_{yy} + 3u_y = b$
c) $u_{xy} - u_x - u_y = b$
d) $3u_{xx} - u_x - u_y = b$
e) $-u_{xx} + 7u_{xy} + 2u_x + 3u = b$

Exercise 2.1.3 Determine the type of each of the following PDEs in dependence of the parameter ε for $u = u(x, y)$.

a) $-\varepsilon u_{xx} - u_{yy} = b$
b) $-\varepsilon \Delta u + a_1 u_x + a_2 u_y = b$

Exercise 2.1.4 In the case of nonconstant coefficients $a_{11} = a_{11}(x, y)$, $a_{12} = a_{12}(x, y)$, $a_{22} = a_{22}(x, y)$, it is possible to generalize the determination of the type of an equation (elliptic, hyperbolic, parabolic) in a straightforward way. In that case, the determinant

$$a_{12}^2(x, y) - 4a_{11}(x, y)a_{22}(x, y)$$

depends on x, y. The equation type then also depends on x, y. With coefficients a_{ij} depending of solution u and/or derivatives u_x, u_y the differential equation is called quasilinear. It is also in that case possible to determine the equation type in a similar way.

An equation describing the flow of a stationary, rotation-free, inviscid ideal fluid is the so-called full potential equation. In 2D, it reads,

$$u_{xx} + u_{yy} - \frac{1}{C^2}(u_x^2 u_{xx} + 2u_x u_y u_{xy} + u_y^2 u_{yy}) = 0$$

where $C = C(u_x, u_y)$ is the local speed of sound. (C also depends on other physical quantities, such as density ρ.) Vector (u_x, u_y) represents the velocity, with components in x- and y-direction, respectively.

Determine the type of the equation in dependence on C.

Can you interpret your solution from a physical point of view ?

Exercise 2.1.5 Show that, for sufficiently smooth functions u, the differential equation

$$\tilde{u}_{\xi\eta} = 0$$

can be transformed into the well-known wave equation

$$u_{xx} - u_{yy} = 0$$

by the coordinate transformation

$$x = (\xi + \eta)/2, \quad y = (\xi - \eta)/2.$$

Exercise 2.1.6 Show that the Black-Scholes equation

$$\frac{\partial u}{\partial t} + \frac{\sigma^2}{2} s^2 \frac{\partial^2 u}{\partial s^2} + rs\frac{\partial u}{\partial s} - ru = 0$$

for $u(s,t)$ is equivalent to the equation

$$\frac{\partial y}{\partial \tau} = \frac{\partial^2 y}{\partial x^2}$$

for $y(x,\tau)$. To show this, proceed as follows:

a) For the transformation $s = Ee^x$ and an appropriate transformation $t \leftrightarrow \tau$, the Black-Scholes equation is equivalent to

$$-\dot{u} + u'' + \alpha u' + \beta u = 0$$

with $\dot{u} = \frac{\partial u}{\partial \tau}$, $u' = \frac{\partial u}{\partial x}$, and α, β depending on r and σ.

b) The remaining part follows after a transformation of the type

$$u = Ee^{\gamma x + \delta \tau} y(x,\tau)$$

with appropriate γ, δ.

Exercise 2.1.7 Let $[s_{\kappa_1\kappa_2}]_h$ be the five-point stencil

$$[s_{\kappa_1\kappa_2}]_h = \frac{1}{h^2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}_h$$

and $\Omega_h = \{(ih, jh) \mid 0 < i, j < 5\}$.

a) Let w_h be the grid function

$$w_h(ih, jh) = 1 \qquad (ih, jh) \in \Omega_h\,.$$

Calculate $[s_{\kappa_1\kappa_2}]_h w_h(3h, 2h)$ and $[s_{\kappa_1\kappa_2}]_h w_h(3h, 3h)$.

b) Let w_h be the grid function

$$w_h(ih, jh) = jh \qquad (ih, jh) \in \Omega_h \,.$$

Calculate $[s_{\kappa_1\kappa_2}]_h w_h(3h, 2h)$ and $[s_{\kappa_1\kappa_2}]_h w_h(3h, 3h)$.

c) Let w_h be the grid function

$$w_h(ih, jh) = (jh)^2 \qquad (ih, jh) \in \Omega_h \,.$$

Calculate $[s_{\kappa_1\kappa_2}]_h w_h(3h, 2h)$ and $[s_{\kappa_1\kappa_2}]_h w_h(3h, 3h)$.

d) In Section 1.4, we will see that the five-point stencil is a discretization of the Laplacian $-\Delta = -\partial^2/\partial x^2 - \partial^2/\partial y^2$. In view of this fact, interpret the results of a) - c) in terms of their continuous analogs.

Exercise 2.1.8 We derive finite difference methods systematically for second order derivatives of functions of one variable. Therefore, we consider the central difference for a second derivative, which we write in the form

$$(u_i'')_h = \frac{1}{h^2}(a_1 u_{i-1} + a_2 u_i + a_3 u_{i+1})$$

where a_1, a_2 und a_3 represent parameters, that must be calculated so that the scheme has a high accuracy. By Taylor's expansion of u_{i-1} and u_{i+1} around u_i we find

$$\begin{aligned}(u_i'')_h =& \frac{1}{h^2}\Big(u_i(a_1 + a_2 + a_3) + hu_i'(-a_1 + a_3) + \frac{1}{2!}h^2 u_i''(a_1 + a_3) \\ &+ \frac{1}{3!}h^3 u_i'''(-a_1 + a_3) + \frac{1}{4!}h^4 u_i''''(a_1 + a_3) + \cdots\Big)\end{aligned}$$

Use this equation to compute the best possible approximation for the second derivative of u under the assumption that h is very small.

Exercise 2.1.9 One can obtain a higher order central difference scheme for the approximation of a first derivative u', if one uses, for example, u-values at four grid points, $x = -2h,\ x = -h,\ x = h$ and $x = 2h$ (see Fig. 2.1.1).

The approximation for the first derivative at $x = 0$ is written in the form

$$(u_0')_h = a_1 u_1 + a_2 u_2 + a_3 u_3 + a_4 u_4,$$

where a_1, a_2, a_3 and a_4 are parameters that must be calculated so that the scheme has the required accuracy.

Calculate the parameters a_1, a_2, a_3 and a_4 so, that the approximation of u' is exact for the functions $u(x) = 1, u(x) = x, u(x) = x^2$ and $u(x) = x^3$.

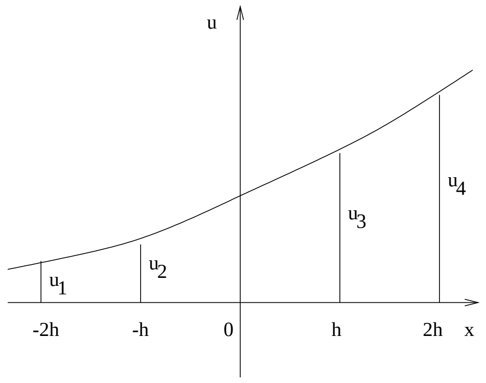


Figure 2.1.1: Grid points for a higher order central difference scheme.

Exercise 2.1.10 Let the following Sturm-Liouville boundary value problem be given:

$$-u'' + p(x)u' + q(x)u = r(x), \quad u(a) = \alpha, \quad u(b) = \beta$$

with $q(x) \geq q_0 > 0$ for $x \in [a, b]$.
We search for approximations $\tilde{u}_i$ of the exact values $u(x_i)$, $x_i = a + ih$, $i = 1, .., n$ and $h = \frac{b-a}{n+1}$. If one replaces $u'(x_i)$ by $\frac{\tilde{u}_{i+1} - \tilde{u}_{i-1}}{2h}$ and $u''(x_i)$ by $\frac{\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1}}{h^2}$ for $i = 1, ..., n$ and further sets $\tilde{u}_0 = \alpha$ and $\tilde{u}_{n+1} = \beta$, a system of equations is obtained for the vector $\tilde{u} := (\tilde{u}_1, ..., \tilde{u}_n)^T$

$$A\tilde{u} = b \quad \text{with} \quad A \in \mathbb{R}^{n \times n},\ c \in \mathbb{R}^n.$$

a) Determine A and b.
b) For which $h > 0$ does A satisfy the requirement $a_{i,j} \leq 0$ for $i \neq j$?

Exercise 2.1.11

a) Set up three second-order accurate discretizations for the mixed derivative of the form u_{xy}, that can be represented in stencil notation by,

$$A_h = \frac{1}{h^2}\begin{bmatrix} a_2 & a_1 & 0 \\ a_1 & a_3 & a_1 \\ 0 & a_1 & a_2 \end{bmatrix}_h, \quad B_h = \frac{1}{h^2}\begin{bmatrix} 0 & b_1 & b_2 \\ b_1 & b_3 & b_1 \\ b_2 & b_1 & 0 \end{bmatrix}_h, \quad C_h = \frac{1}{h^2}\begin{bmatrix} c_1 & 0 & c_2 \\ 0 & 0 & 0 \\ c_3 & 0 & c_4 \end{bmatrix}_h;$$

b) Consider the equation

$$\begin{aligned} -\Delta u - \tau u_{xy} &= 0 \quad (\Omega = (0,1)^2) && (2.1.1) \\ u &= g \quad (\partial\Omega). && (2.1.2) \end{aligned}$$

Write down a discretization in stencil notation for an interior grid point.

c) A Matrix A is called Z-matrix, if $a_{ij} \le 0$ for $i \ne j$. The Z-matrix property is (for example, as a part of the definition of M-matrices) a basis for convergence proofs of certain iterative solution methods. For which τ-values and which discretization approaches discussed under (a), does the boundary value problem (2.1.2) result in a Z-matrix ? Which discretization approach for (2.1.2) would give the Z-matrix property for general $-2 < \tau < 2$?

Exercise 2.1.12 Prove the relation

$$\mathcal{L}u - L_h u = O(h^2) \quad \text{for } h \to 0$$

for sufficiently smooth functions u for the standard five-point discretization

$$-\Delta_h = \frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h$$

of the operator $\mathcal{L} = -\Delta$ using Taylor expansion.

Exercise 2.1.13 Consider the 1D problem $\mathcal{L}u(x) = -u''(x) = b(x)$ on the interval $\Omega = (0,1)$ with boundary conditions $u(0) = u_0$, $u(1) = u_1$. Set up the discrete problem (matrix and right-hand side) for the discretization

$$L_h = \frac{1}{h^2} \begin{bmatrix} -1 & 2 & -1 \end{bmatrix}$$

and $h = 1/4$

a) without elimination of boundary conditions,
b) with elimination of boundary conditions.

Exercise 2.1.14 Consider $\mathcal{L}u(x) = -\Delta u(x,y) = b(x,y)$ on the domain $\Omega = (0,1)^2$ with boundary conditions $u(x,y) = g(x,y)$. Set up the discrete problem (matrix and right-hand side) for the standard five-point discretization of $\mathcal{L}$ (see Exercise 2.1.12) and $h = 1/3$

a) without elimination of boundary conditions,
b) with elimination of boundary conditions,

using a lexicographic ordering of grid points in both.

Exercise 2.1.15 Consider $\mathcal{L}u(x) = -\Delta u(x,y) = b(x,y)$ on the domain $\Omega = (0,1)^2$ with boundary conditions $u(x,y) = g(x,y)$. Set-up the discrete problem (matrix and right-hand side) for the standard five-point discretization of $\mathcal{L}$ (see Exercise 2.1.12) and $h = 1/3$

a) without elimination of boundary conditions,

b) with elimination of boundary conditions, using a red–black ordering of grid points.

Exercise 2.1.16 Determine the sparsity structure of the five-point stencil if the grid points are ordered in alternating lines

13	14	15	16
5	6	7	8
9	10	11	12
1	2	3	4

2.2 Some Basic Facts on Iterative Methods

2.2.1 Inner Products and Norms

The solution of matrix $Au_h = b$ by iterative methods will bring us an approximate solution u_h^i for the exact solution u_h. A question how to determine the accuracy of u_h^i is answered with the help of a vector norm (in $\mathbb{C}^N$):

$$|| \cdot || : \mathbb{C}^N \to \mathbb{R}$$

It assigns to each vector $u_h \in \mathbb{C}^N$ a real number $||u_h||$, that serves as a measure for the "size" of u_h. The map $||.|| : \mathbb{C}^N \to \mathbb{R}$ is called a norm, if

- $||u_h|| > 0 \quad$ for all $u_h \in \mathbb{C}^N, \quad u_h \neq 0$ (definiteness)
- $||\alpha u_h|| = |\alpha| ||u_h|| \quad$ for all $\alpha \in \mathbb{C},\ u_h \in \mathbb{C}^N$ (homogeneity)
- $||u_h + v_h|| \leq ||u_h|| + ||v_h|| \quad$ for all $u_h, v_h \in \mathbb{C}^N$ (triangle inequality).

Most considerations will be based on the *Euclidean inner product*

$$< u_h, v_h >_2 := \frac{1}{\#\Omega_h} \sum_{\boldsymbol{x} \in \Omega_h} u_h(\boldsymbol{x}) v_h(\boldsymbol{x}) \ , \qquad (2.2.1)$$

where $\#\Omega_h$ is the number of grid points of Ω_h. The scaling factor $(\#\Omega_h)^{-1}$ allows one to compare grid functions on different grids and also the corresponding continuous functions on Ω. The induced norm is

$$||u_h||_2 = \sqrt{< u_h, u_h >_2}.$$

For practical purposes the *infinity norm*

$$||u_h||_\infty := \max\{|u_h(\boldsymbol{x})| \ : \boldsymbol{x} \in \Omega_h\} \qquad (2.2.2)$$

is often used.

It is a well-known theory that all norms in $\mathbb{R}^N$ ($\mathbb{C}^N$) are equivalent in the following sense: For each pair of norms $p_1(u_h),\ p_2(u_h)$ there are constants $m_1 > 0$ and $m_2 > 0$, so that for all u_h it follows that

$$m_1\, p_2(u_h) \leq p_1(u_h) \leq m_2\, p_2(u_h).$$

For matrices $A \in \mathbb{R}^{n_1 \times n_2}$, i.e. for $n_1 \times n_2$-matrices, norms $||A||$ have been introduced:

$$\begin{aligned} ||A|| &> 0 \quad \text{for all} \quad A \neq 0,\ A \in \mathbb{R}^{n_1 \times n_2} \\ ||\alpha A|| &= |\alpha| ||A|| \\ ||A + B|| &\leq ||A|| + ||B|| \end{aligned}$$

The matrix norm is called compatible with vector norm $||.||_a$ on $\mathbb{C}^N$ and $||.||_b$ on $\mathbb{C}^N$, if

$$||Au_h||_b \leq ||A||\;||u_h||_a \quad \text{for all } \; u_h \in \mathbb{C}^N, \;\; A \in \mathbb{R}^{m\times n}.$$

A matrix norm $||.||$ for quadratic matrices $A \in \mathbb{R}^{m\times n}$ is called sub-multiplicative if

$$||AB|| \leq ||A||\;||B|| \quad \text{for all} \quad A, B \in \mathbb{R}^{m\times n}.$$

Norms that are often used are

$$\begin{aligned} a)\;\; ||A|| &= \max_i \sum_{k=1}^{N} |a_{ik}| \quad \text{(Row sum norm)} \\ b)\;\; ||A|| &= \left(\sum_{i,k=1}^{N} |a_{ik}|^2 \right)^{\frac{1}{2}} \quad \text{(Schur norm)} \\ c)\;\; ||A|| &= \max_{i,k} |a_{ik}| \end{aligned}$$

a) and b) are sub-multiplicative, c) is not, b) is compatible with the Euclidean vector norm. Closely related are the operator norms $||L_h||$ for operators L_h.

The operator norm corresponding to the 2-norm for discrete operators L_h on $\mathcal{G}(\Omega_h)$ is the *spectral* norm denoted by $||\cdot||_S$.

For L_h, symmetric and positive definite, we also consider the *energy inner product*

$$< u_h, v_h >_E := < L_h u_h, v_h >_2 \tag{2.2.3}$$

and the corresponding operator norm $||\cdot||_E$, which is given by

$$||B_h||_E = ||{L_h}^{1/2} B_h {L_h}^{-1/2}||_S = \sqrt{\rho(L_h B_h {L_h}^{-1} B_h^*)}\;, \tag{2.2.4}$$

where B_h denotes any linear operator $B_h : \mathcal{G}(\Omega_h) \to \mathcal{G}(\Omega_h)$.

Sobolev spaces and their extensions are natural for the theory of PDEs (and their discrete approximations) and make these theories transparent.

2.2.2 Iterative Solvers, Splittings, Preconditioners

Here, we list some basic facts. Since these facts are valid for general matrices, we use linear algebra notation in this section, i.e. matrices A instead of discrete operators L_h. Let

$$Au = b \tag{2.2.5}$$

be a linear system with an invertible matrix A.

Iterative methods for the solution of linear systems of equations are useful, if the number of iterations necessary is not too big, if A is sparse, or A has a special structure, or if a good guess for u is available, as in time-stepping methods, or

A is, for example, not known explicitly. Otherwise, direct methods may be more advantageous. We use the following *general iteration*,

$$u^{i+1} = Qu^i + s, \ (i = 0, 1, 2, \ldots), \tag{2.2.6}$$

so that the system $u = Qu + s$ is equivalent to the original problem. Here, Q is the *iteration matrix*. It is, however, usually not computed explicitly.

From the matrix equation (2.2.5), we construct a splitting $A = B + (A - B)$, so that B is "easily invertible".

With the help of a general nonsingular $N \times N$ matrix B we obtain such iteration procedures from the equation

$$Bu + (A - B)u = b.$$

If we put

$$Bu^{i+1} + (A - B)u^i = b \ ,$$

or, rewritten for u^{i+1};

$$u^{i+1} = u^i - B^{-1}(Au^i - b) = (I - B^{-1}A)u^i + B^{-1}b \ .$$

It is important to have the eigenvalues of $Q = I - B^{-1}A$ as small as possible. This is more likely, the better B resembles A.

An iterative process $u^{i+1} = Qu^i + s$ is called *consistent*, if matrix $I - Q$ is not singular and if $(I - Q)^{-1}s = A^{-1}b$. If the iteration is consistent, the equation $(I - Q)u = s$ has exactly one solution $u^{i\to\infty} \equiv u$.

An iteration is called *convergent*, if the sequence $u^0, u^1, u^2, \ldots$ converges to a limit, independent of u^0.

Before we proceed we consider how perturbations in A and b affect the solution u. The condition number $\kappa_p(A)$, for a nonsingular matrix A, is defined by

$$\kappa_p(A) := \|A\|_p \|A^{-1}\|_p.$$

Theorem 2.1 *Suppose $Au = b$, $A \in I\!R^{n\times n}$ and A is nonsingular, $0 \neq b \in I\!R^n$, $(A + \Delta A)y = b + \Delta b$, $\Delta A \in I\!R^{n\times n}$, $\Delta b \in I\!R^n$ with $\|\Delta A\|_p \leq \delta\|A\|_p$ and $\|\Delta b\|_p \leq \delta\|b\|_p$. If $\kappa_p(A)\delta = \overline{\rho} < 1$ then $A + \Delta A$ is nonsingular and*

$$\frac{\|u - y\|_p}{\|u\|_p} \leq \frac{2\delta}{1 - \overline{\rho}}\kappa_p(A).$$

<u>*Proof*</u>: see [57], p.83.

The simplest iterative scheme is the *Richardson iteration*

$$u^{i+1} = u^i + \tau(b - Au^i) = (I - \tau A)u^i + \tau b \quad (i = 0, 1, ...) \tag{2.2.7}$$

with some acceleration parameter $\tau \neq 0$.

For Richardson's iteration, we have $Q = I - \tau A,\ s = \tau b$ in (2.2.6).

The iterative method $u^{i+1} = Qu^i + s \quad i = 0, 1, 2, \dots$ yields a convergent sequence $\{u^i\}$ for a general u^0, s ("the method converges"), if

$$\rho(Q) < 1 \ .$$

The spectral radius $\rho(Q)$ of a (N, N) matrix Q is

$$\rho(Q) = \max \Big\{ |\lambda| : \lambda \text{ eigenvalue of } Q \Big\} \ . \tag{2.2.8}$$

It is the infimum over all (vector norm related) matrix norms,

$$\rho(Q) = \inf \Big\{ \parallel Q \parallel \Big\}.$$

For each matrix norm $||Q||$, we have

$$\lim_{i \to \infty} \sqrt[i]{\parallel Q^i \parallel} = \rho(Q) \ .$$

The spectral radius is the asymptotic convergence rate of the iteration. Asymptotically (i.e., for $i \to \infty$) we have $||u - u^{i+1}|| \leq \rho(Q)||u - u^i||$.

In the case $\rho(Q) < 1$, matrix $I - Q$ is regular, there is for each s exactly one fix point of the iterative scheme. The convergence (and the asymptotic convergence speed) of Richardson's iteration is characterized by the spectral radius

$$\rho(I - \tau A).$$

The spectral radius can also be used for error reduction purposes. Starting from the general splitting of A,

$$Bu^{i+1} + (A - B)u^i = b \equiv Au = Bu + (A - B)u$$

or,

$$B(u^{i+1} - u) = (B - A)(u^i - u).$$

It follows that,

$$(u^{i+1} - u) = Q(u^i - u).$$

By induction, we can obtain:

$$(u^i - u) = Q^i(u^0 - u).$$

Let us assume $\rho(Q) < 1$, and suppose we desire a reduction of the error of 10^{-d} $(d = 3, 4)$. It can be found that

$$\frac{||e^i||}{||e^0||} \leq ||Q^i|| \text{ where } e^i = u^i - u.$$

Now, one approximates $||Q^i||$ by $\rho(Q)^i$, i.e., the spectral radius is an indication for the size of the error reduction. Iteration number i must be chosen such that

$$\begin{aligned} \rho(Q)^i &\leq 10^{-d}, \text{or} && (2.2.9) \\ i &\geq \frac{d}{-\log \rho(Q)}. && (2.2.10) \end{aligned}$$

There are many ways to specify Q leading to different iterative solvers. Here, we present three different but equivalent ways to formulate or to introduce the iteration (2.2.6). These three approaches differ only in their motivation, not mathematically.

Approximate solution of the defect equation:
If u^i is any approximation of u and

$$r^i = b - Au^i \tag{2.2.11}$$

is its *defect* ((or residual), then the defect equation $Ae^i = r^i$ is equivalent to the original equation: By solving for the *correction* e^i, we obtain the solution $u = u^i + e^i$. If we use, however, an *approximation* $\widehat{A}$ *of* A, such that

$$\widehat{A}\widehat{e}^i = r^i \tag{2.2.12}$$

can be solved more easily, we obtain an iterative process of the form

$$r^i = b - Au^i, \quad \widehat{A}\widehat{e}^i = r^i, \quad u^{i+1} = u^i + \widehat{e}^i \quad (i = 0, 1, 2...) \tag{2.2.13}$$

This process is obviously equivalent to (2.2.6) where

$$Q = I - (\widehat{A})^{-1}A \ .$$

Vice versa, if Q is given, this yields an approximation $\widehat{A}$ of A according to

$$\widehat{A} = A(I - Q)^{-1} \ .$$

Splitting:
An equivalent way of constructing Q is to start with a *splitting*

$$A = \widehat{A} - R$$

and to use the iteration

$$\widehat{A}u^{i+1} = Ru^i + b \ . \tag{2.2.14}$$

Here

$$Q = (\widehat{A})^{-1}R = I - (\widehat{A})^{-1}A \ .$$

Preconditioning:

A third, also equivalent, approach is based on the idea of *preconditioning*. Here the original equation $Au = b$ is replaced by an equivalent equation

$$M^{-1}Au = M^{-1}b. \tag{2.2.15}$$

M^{-1} is called a *(left) preconditioner* of A. The identification with the above terminology is by

$$(\widehat{A})^{-1} = M^{-1}.$$

In other words, the inverse $(\widehat{A})^{-1}$ of any (invertible) approximation $\widehat{A}$ is a left preconditioner and vice versa.

Furthermore, we see that Richardson's iteration for the preconditioned system (2.2.15) (with $\tau = 1$)

$$u^{i+1} = u^i + M^{-1}(b - Au^i) = (I - M^{-1}A)u^i + M^{-1}b \tag{2.2.16}$$

is equivalent to the general iteration (2.2.6) with $Q = I - M^{-1}A$. This also means that any iteration of the general form (2.2.6) is a Richardson iteration (with $\tau = 1$) for the system that is obtained by preconditioning the original system (2.2.5).

Remark 2.2.1 One speaks of a *right preconditioner* M^{-1}, if the original equation is replaced by

$$AM^{-1}z = b, \quad u = M^{-1}z \ . \tag{2.2.17}$$

Richardson's method based on right preconditioning (with $\tau = 1$) would result in

$$z^{i+1} = (I - AM^{-1})z^i + b \ . \tag{2.2.18}$$

Since

$$(I - AM^{-1}) = A(I - M^{-1}A)A^{-1},$$

we have

$$\rho(I - AM^{-1}) = \rho(I - M^{-1}A)$$

$\gg$

We add some remarks, which further motivate the term *preconditioning*. The idea behind preconditioning is that the condition of the system (2.2.5) – measured by the condition number

$$\kappa(A) = ||A|| \ ||A^{-1}|| \tag{2.2.19}$$

(in some appropriate norm) – is to be improved by multiplying A by M^{-1}, in the form (2.2.15) or (2.2.17). The condition number, on the other hand, is relevant for the convergence speed of certain iterative approaches, for instance Richardson's iteration and conjugate gradient type methods. We summarize some well-known facts here. For that purpose, we assume that the matrix A is *symmetric and positive definite* (s.p.d.) with maximum and minimum eigenvalues $\lambda_{\max}, \ \lambda_{\min} > 0$ respectively. We consider the Euclidean norm in $\mathbb{R}^N$ and the corresponding spectral matrix norm $||A||_S = \lambda_{\max}$.

Remark 2.2.2 If A is s.p.d., the Richardson iteration converges for

$$0 < \tau < 2||A||_S{}^{-1} \ .$$

Then the optimal τ (for which the spectral radius $\rho(I - \tau A)$ becomes minimal) is $\tau_{\text{opt}} = 2/(\lambda_{\max} + \lambda_{\min})$, and one can prove

$$\rho(I - \tau_{\text{opt}} A) = ||I - \tau_{\text{opt}} A||_S = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} = \frac{\kappa_S(A) - 1}{\kappa_S(A) + 1} \ , \tag{2.2.20}$$

where $\kappa_S(A)$ is the spectral condition number of A (for the proof, see for example [72]).

If we use a left preconditioner M^{-1} which is also s.p.d., we obtain

$$\frac{\kappa_S(M^{-1}A) - 1}{\kappa_S(M^{-1}A) + 1} \tag{2.2.21}$$

instead of (2.2.20). $\gg$

2.2.3 Exercises

Exercise 2.2.1 Calculate the Euclidean inner products of the following vectors

a) $(1, -7, 12, 2)^T$, $(3, 2, 0, 5)^T$
b) $(\sin x, \cos x)^T$, $(\cos x, -\sin x)^T$.

Exercise 2.2.2 Calculate the Euclidean norm of the following vectors

a) $(-1, 5, 2)^T$
b) $(\sin x, \cos x)^T$.

Exercise 2.2.3 Let the matrix A be given by

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} .$$

$\|v\|_E = (\langle Av, v\rangle_2)^{1/2}$ is called the energy norm of v with respect to A.

a) Calculate the energy norm (with respect to A) of the vector $(v_1, v_2, v_3, v_4)^T$.
b) Show that the energy norm calculated in a) is positive, unless v is the zero vector.

Exercise 2.2.4 The matrix norm induced by a vector norm $\|\cdot\|$ is defined by

$$\|A\| = \sup_{v\neq 0} \frac{\|Av\|}{\|v\|} = \sup_{\|v\|=1} \|Av\| .$$

The spectral norm $\|A\|_S$ of a symmetric positive definite matrix A is given by

$$\|A\|_S = \max_{\lambda \text{ eigenvalue of } A} \lambda .$$

One can show that the matrix norm induced by the Euclidean norm is the spectral norm (see Exercise 2.2.5).

This exercise shows that the above statement is true for the matrix A given by

$$A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} .$$

a) Show that A is symmetric, positive definite.
b) Compute the spectral norm of A by determining its eigenvalues.
c) Determine the norm of A induced by the Euclidean norm by using the method of Lagrange multipliers to calculate $\sup_{\|v\|=1} \|Av\|$. Compare your answer with the largest eigenvalue from b).

Exercise 2.2.5

a) Let $A \in \mathbb{R}^{n\times n}$ be a positive definite symmetric $(n \times n)$-matrix. Show, that

$$||x|| := \sqrt{x^T A x}$$

defines a vector norm in $\mathbb{R}^n$.

b) Show that the spectral norm for quadratic matrices is the matrix norm induced by the Euclidean norm (see Exercise 2.2.4 for the definition of induced matrix norms).
Hint: The spectral norm is defined as follows:

$$\| A \|_2 := \sqrt{\rho(\overline{A}^T A)}$$

where ρ is the spectral radius.

c) Let $D \in \mathbb{R}^{n\times n}$ be a diagonal $(n \times n)$-matrix with diagonal elements $d_i > 0$ and
let the $(n \times n)$-matrix $C \in \mathbb{R}^{n\times n}$ be symmetric.
Show that, the following inequality is valid for the spectral norm:

$$\| D^{-1}C \|_2 \leq (\min\{d_i : 1 \leq i \leq n\})^{-1} \rho(C)$$

d) Determine the unit circle $E := \{x \in \mathbb{R}^2 | x^T A x \leq 1\}$ for

$$A = \begin{pmatrix} 9/5 & -8/5 \\ -8/5 & 21/5 \end{pmatrix}$$

and sketch E.

Exercise 2.2.6 Calculate the uniform (infinity) norm of the vectors in Exercises 2.2.1, 2.2.2. In 2.2.2 b), determine the points x, $0 \leq x < 2\pi$, for which the vector $(\sin x, \cos x)^T$ has the largest infinity norm.

Exercise 2.2.7 Two vectors u_h and v_h are said to be orthogonal if $\langle u_h, v_h \rangle = 0$. In the following, take $\langle \cdot, \cdot \rangle$ to be the Euclidean inner product.

a) Find a vector which is orthogonal to $(a, b)^T$.
b) Use a) to find a vector which is orthogonal to $(\sin x, \cos x)^T$. Sketch the two vectors for $x = \pi/4$.

Exercise 2.2.8 Show that the iteration given by

$$r^i = b - Au^i, \quad \hat{A}\hat{e}^i = r^i, \quad u^{i+1} = u^i + \hat{e}^i \quad (i = 0, 1, 2 \ldots)$$

can be written in the form $u^{i+1} = Qu^i + s$, with $Q = I - (\hat{A})^{-1}A$.

Exercise 2.2.9 Show that the iteration

$$\hat{A}u^{i+1} = Ru^i + b \ .$$

(which is based on a splitting $A = \hat{A} - R$ leads to an iteration of the form $u^{i+1} = Qu^i + s$, with $Q = I - (\hat{A})^{-1}A$.

Exercise 2.2.10 What is the condition number of the identity matrix I in any norm?

Exercise 2.2.11 Let A be a symmetric, positive definite matrix. Show that the Richardson iteration applied to A converges in the Euclidean norm for

$$0 < \tau < 2\|A\|_S^{-1} \ .$$

Exercise 2.2.12 Let A be a symmetric, positive definite matrix with largest and smallest eigenvalues $\lambda_{\max}$ and $\lambda_{\min}$ respectively. Show that the value of τ for which $\rho(I - \tau A)$ becomes minimum, and thus the optimal value for the Richardson iteration, is

$$\tau = \frac{2}{\lambda_{\max} + \lambda_{\min}} \ .$$

Exercise 2.2.13 For the solution of the system $Au = b$, an iterative method

$$u^{i+1} = Qu^i + s$$

with $u^0 = 0$ is chosen. Prove that for u^i

$$u^i = (I - Q^i)A^{-1}f$$

is valid. Indicate the assumptions and determine s.

2.3 Jacobi and Gauss-Seidel Iterative Solution Methods

Two well-known examples for a splitting are the Jacobi and the Gauss-Seidel methods.

Let $A \in \mathbb{R}^{N,N}$ be a regular matrix, $A = (a_{mn}),\ m, n = 1, \ldots, N$ with $a_{mm} \neq 0$ $(m = 1, \ldots, N)$, $b \in \mathbb{R}^N$.
An equation with $N = 3$ reads

$$\begin{aligned}
a_{11}u_1 + a_{12}u_2 + a_{13}u_3 &= b_1 \\
a_{21}u_1 + a_{22}u_2 + a_{23}u_3 &= b_2 \\
a_{31}u_1 + a_{32}u_2 + a_{33}u_3 &= b_3.
\end{aligned}$$

These equations can be solved for by “updating” $u_1,\ u_2,\ u_3$ as follows:

$$\begin{aligned}
u_1 &= (b_1 - \phantom{a_{21}u_1 -} a_{12}u_2 - a_{13}u_3)/a_{11} \quad \text{and} \\
u_2 &= (b_2 - a_{21}u_1 - \phantom{a_{12}u_2 -} a_{23}u_3)/a_{22} \\
u_3 &= (b_3 - a_{31}u_1 - a_{32}u_2 \phantom{- a_{13}u_3})/a_{33}.
\end{aligned}$$

One finds similar equations for general $N \in \mathbb{N}$.
This procedure can be used to solve a linear system of equations iteratively.

In the Jacobi method, we compute, starting with initial guess u_m^0, $m = 1, 2, 3$, recursively the vectors $u^1, \ldots, u^p$:

$$u_m^{i+1} = \frac{1}{a_{mm}}(b_m - \sum_{k=1 (k \neq m)}^{N} a_{mn}u_k^i), \quad m = 1, \ldots, N.$$

In Gauss-Seidel's method one uses, different from Jacobi's method, during the computation of u_m^{i+1} the components computed previously in this iteration $u_1^{i+1} \ldots, u_{m-1}^{i+1}$ instead of $u_1^i \ldots, u_{m-1}^i$. Herewith, the approximations u^i should reach an exact solution u faster, if the process converges at all,

$$u_m^{i+1} = \frac{1}{a_{mm}}(b_m - \sum_{k=1}^{m-1} a_{mk}u_k^{i+1} - \sum_{k=m+1}^{N} a_{mk}u_k^i), \quad m = 1, \ldots, N.$$

The difference in computation between both iterative methods is clearly seen during the coding.

```
JAC(u, N)    /* one iteration of Jacobi's method */
             {   for m = 1, ..., N
                 $q_m = (b_m - \sum_{n \neq m} a_{nm} u_n)/a_{mm};$
                 $u = q;$
             }
GS(u, N)     /* one iteration of Gauss-Seidel's method */
             {   for m = 1, ..., N
                 $u_m = (b_m - \sum_{n \neq m} a_{mn} u_n)/a_{mm};$
             }
```

The Gauss-Seidel iteration is applied by successively overwriting the pointwise approximations. Instead of two vectors u and q storage, one only needs vector u.

There is difference in *parallel complexity* between the Jacobi and Gauss-Seidel methods: In the Jacobi iteration each point can be computed independently of the current computations in the other points. They can all be computed at the same time. The degree of parallelism is $\#\Omega_h$.

On a serial machine Jacobi can be implemented with half the computations of Gauss-Seidel (for PDE derived matrices). In Gauss-Seidel there is a sequential behavior, some unknowns can only be computed after the computation of other unknowns: Diagonal points can be computed in parallel: The degree of parallelism is at most $\sqrt{\#\Omega_h}$.

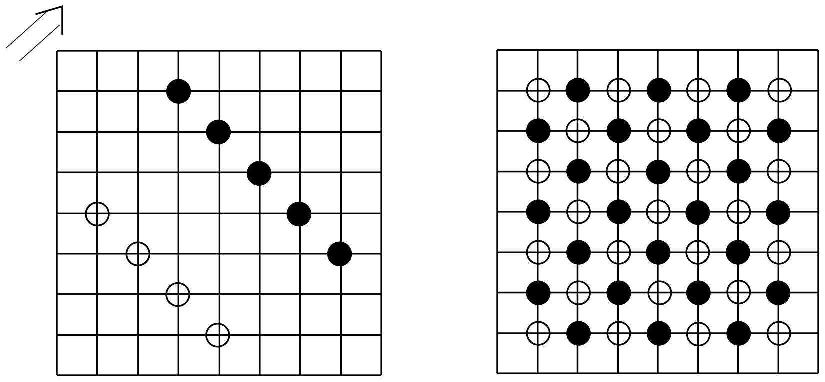


Figure 2.3.2: a) Unknowns along diagonals can be updated simultaneously during lexicographical Gauss-Seidel, b) Gauss-Seidel with red-black ordering

The red-black Gauss-Seidel method has better parallel characteristics. In a first step, all odd (red) unknowns are treated. With a 5-point discretization stencil, these are independent of each other and can be updated independently.

Then, in the second step, the even (black) unknowns are updated independently of each other, using the latest values at the red points. As the convergence of Gauss-Seidel is not influenced by the ordering of unknowns, it is the same as for lexicographical Gauss-Seidel. The degree of parallelism is, however, much better $0.5\#\Omega_h$.

The matrix view of the Jacobi method is found by splitting A up in the following form $A = D - C$, with a diagonal matrix D and a matrix C, that only contains zeros in the main diagonal. Then, the Jacobi method reads in *matrix notation*

$$Du^{i+1} = Cu^i + b \ ,$$

with *vectors* u^i and u^{i+1}. It can be written in the form $u^{i+1} = Qu^i + s$ with

$$Q = Q_{JAC} = D^{-1}C, \ s = D^{-1}b \ .$$

In the Gauss-Seidel method, C is split up into: $C = C_1 + C_2$, with a lower triangular matrix C_1 and an upper triangular matrix C_2.

Then, the Gauss-Seidel method reads in matrix notation

$$(D - C_1)u^{i+1} = C_2u^i + b \ .$$

After a multiplication with $(D - C_1)^{-1}$ we find the form $u^{i+1} = Qu^i + s$ with

$$Q = Q_{GS} = (D - C_1)^{-1}C_2, \ s = (D - C_1)^{-1}b \ .$$

2.3.1 Starting Vectors and Termination Criteria

Starting Vectors All the given iterative solution methods used to solve $Au = b$ start with a given vector u^0. In this subsection we shall give some ideas how to choose a good starting vector u^0. These choices depend on the problem to be solved. If no further information is available one typically starts with $u^0 = 0$. The solution of a nonlinear problem is in general approximated by the solution of a number of linear systems. In such a problem the final solution of the iterative method at a given outer iteration can be used as a starting solution for the iterative method used to solve the next linear system.

Suppose we have a time dependent problem. In each time step the solution of the continuous problem is approximated by a discrete solution $u_h^{(n)}$ satisfying the following linear system

$$A^{(n)}u_h^{(n)} = b^{(n)}.$$

These systems are approximately solved by an iterative method where the iterates are denoted by $u^{(n,k)}$. An obvious choice for the starting vector is $u^{(n+1,0)} = u^{(n,k_n)}$ where k_n denotes the number of iterations in the n^{th} time step. A better initial estimate can be obtained by the following extrapolation:

$$u^{(n+1)} \cong u^{(n)} + \triangle t \frac{du^{(n)}}{dt},$$

where $\frac{du^{(n)}}{dt}$ is approximated by $\frac{u^{(n,k_n)}-u^{(n-1,k_{n-1})}}{\triangle t}$. This leads to the following starting vector

$$u^{(n+1,0)} = 2u^{(n,k_n)} - u^{(n-1,k_{n-1})} \ .$$

Finally starting vectors can sometimes be obtained by solution of related problems, e.g., analytic solution of a simplified problem, a solution computed by a coarser grid, a numerical solution obtained by a small change in one of the parameters etc.

Stopping Criteria In the previous section we have specified basic iterative methods to solve $Au = b$. However, no criteria to stop the iterative process have been given. In general, the iterative method should be stopped if the approximate solution is accurate enough. A good termination criterion is very important for an iterative method, because if the criterion is too weak the approximate solution is useless, whereas if the criterion is too severe the iterative solution method never stops or costs too much work.

We start by giving a termination criterion for a *linear convergent* process. An iterative method is *linearly convergent* if the iterates satisfy the following relation:

$$\|u^k - u^{k-1}\|_2 \approx \overline{\rho}\|u^{k-1} - u^{k-2}\|_2, \ \ \overline{\rho} < 1 \tag{2.3.1}$$

and $u^k \to A^{-1}b$ for $k \to \infty$. Relation (2.3.1) is easily checked during the computation. In general initially (2.3.1) is not satisfied but after some iterations the quantity $\frac{\|u^k - u^{k-1}\|_2}{\|u^{k-1} - u^{k-2}\|_2}$ will converge to $\overline{\rho}$. The Jacobi and Gauss Seidel methods are linear convergent. We have the following theorem:

Theorem 2.2 *For a linear convergent process we have the following inequality*

$$\|u - u^i\|_2 \leq \frac{\overline{\rho}}{1-\overline{\rho}}\|u^i - u^{i-1}\|_2.$$

Proof

Using (2.3.1) we obtain the following inequality for $k \geq i + 1$.

$$\|u^k - u^i\|_2 \leq \sum_{j=i}^{k-1} \|u^{j+1} - u^j\|_2 \leq \sum_{j=1}^{k-i} \overline{\rho}^j \|u^i - u^{i-1}\|_2$$

$$= \overline{\rho}\frac{1-\overline{\rho}^{k-i-1}}{1-\overline{\rho}r}\|u^i - u^{i-1}\|_2 \ .$$

Since $\lim\limits_{k\to\infty} u^k = u$, this implies that

$$\|u - u^i\|_2 \leq \frac{\overline{\rho}}{1-\overline{\rho}}\|u^i - u^{i-1}\|_2. \qquad \square$$

The result of this theorem can be used to give a stopping criterion for linear convergent methods. Sometimes the iterations are terminated if $\|u^i - u^{i-1}\|_2$ is small enough. If $\overline{\rho}r$ is close to one this may lead to inaccurate results since $\frac{\overline{\rho}}{1-\overline{\rho}}\|u^i - u^{i-1}\|_2$ and thus $\|u - u^i\|_2$ may be large. A safe stopping criterion is:

$$\text{Stop if: } \frac{\overline{\rho}}{1-\overline{\rho}} \frac{\|u^i - u^{i-1}\|_2}{\|u^i\|_2} \leq \epsilon.$$

If this condition holds then the relative error is less than ε:

$$\frac{\|u - u^i\|_2}{\|u\|_2} \cong \frac{\|u - u^i\|_2}{\|u^i\|_2} \leq \frac{\overline{\rho}}{1-\overline{\rho}} \frac{\|u^i - u^{i-1}\|_2}{\|u^i\|_2} \leq \epsilon.$$

Furthermore, Theorem 2.2 yields the following result:

$$\|u - u^i\|_2 \leq \frac{\overline{\rho}^i}{1-\overline{\rho}}\|u^1 - u^0\|_2 \ . \qquad (2.3.2)$$

So assuming that expression (2.3.1) can be replaced by an equality

$$\log \ \|u - u^i\|_2 \ = \ i \log\,(\overline{\rho}) + \log\left(\frac{\|u^1 - u^0\|_2}{1-\overline{\rho}}\right) \ . \qquad (2.3.3)$$

This implies that the curve $\log \ \|u - u^i\|_2$ is a straight line as a function of i, and was the motivation for the term "linear convergent process". Given the quantity $\overline{\rho}$, which is also known as the *rate of convergence*, or *reduction factor*, the required accuracy and $\|u^1 - u^0\|_2$ it is possible to estimate the number of iterations to achieve the required accuracy. In general $\overline{\rho}$ may be close to one and hence a small increase in $\overline{\rho}$ may lead to a large increase in the required number of iterations.

For iterative methods, that have a different convergence behavior most stopping criteria are based on the norm of the residual. Below we shall give some of these criteria and give comments on their properties.

- *Criterion 1:* $\|b - Au^i\|_2 \leq \epsilon$.

  The main disadvantage of this criterion is that it is not scaling invariant. This implies that if $\|b - Au^i\|_2 < \epsilon$ this does not hold for $\|100(b - Au^i)\|_2$. Although the accuracy of u^i remains the same. So a correct choice of ϵ depends on properties of the matrix A.

  The next criteria are all scaling invariant.

- *Criterion 2:* $\frac{\|b - Au^i\|_2}{\|b - Au^0\|_2} \leq \epsilon$

  The number of iterations is independent of the initial estimate u^0. This may be a drawback since a better initial estimate will not lead to a decrease in the number of iterations. If the initial estimate is very good this criteria may never be satisfied, due to round off.

- *Criterion 3:* $\dfrac{\|b - Au^i\|_2}{\|b\|_2} \leq \epsilon$

  This is a good termination criterion. The norm of the residual is small with respect to the norm of the right-hand side. Replacing ϵ by $\epsilon/\kappa_2(A)$ we can show that the relative error in u is less than ϵ. Here, $\kappa_2(A) = \|A\|_2\|A^{-1}\|_2$ is the condition number. It follows that:

$$\frac{\|u - u^i\|_2}{\|u\|_2} \leq \kappa_2(A)\frac{\|b - Au^i\|_2}{\|b\|_2} \leq \epsilon.$$

  In general $\|A\|_2$ and $\|A^{-1}\|_2$ are not known. Some iterative methods give approximations of these quantities.

- *Criterion 4:* $\dfrac{\|b - Au^i\|_2}{\|u^i\|_2} \leq \epsilon/\|A^{-1}\|_2$

  This criterion is closely related to Criterion 3. In many cases this criterion also implies that the relative error is less than ϵ:

$$\frac{\|u - u^i\|_2}{\|u\|_2} \cong \frac{\|u - u^i\|_2}{\|u^i\|_2} = \frac{\|A^{-1}(b - Au^i)\|_2}{\|u^i\|_2} \leq \frac{\|A^{-1}\|_2\|b - Au^i\|_2}{\|u^i\|_2} \leq \epsilon$$

Sometimes physical relations lead to other termination criteria. This is the case if the residual has a physical meaning, for instance the residual is equal to some energy or the deviation from a divergence free vector field etc.

It appears that, due to rounding errors, the solution u represented on a computer has a residual which may be of the magnitude of $\mu\|b\|_2$, where μ is the machine precision. So, we cannot expect that a solution computed by an iterative solution method has a smaller norm of the residual. In a proper implementation of an iterative method, a warning is given if the required accuracy is too high. If, for instance, the termination criterion is $\|b - Au^i\|_2 \leq \epsilon$ and ϵ is chosen less then $1000\mu\|b\|_2$ a warning should be given and ε should be replaced by $\epsilon = 1000\mu\|b\|_2$. The arbitrary constant 1000 is used for safety reasons.

2.3.2 Convergence of Jacobi and Gauss-Seidel

The following theorems exist for the convergence of Jacobi and Gauss-Seidel.

1. The Jacobi method for the iterative solution of $Au = b$ converges for general right-hand side b towards the unique solution u, if one of the following criteria is fulfilled:

Row sum criterion (strong):

$$\text{for} \quad m = 1, \ldots, N \quad \text{holds} \qquad \sum_{n=1 (n \neq m)}^{N} \frac{|a_{mn}|}{|a_{mm}|} < 1 \ .$$

Column sum criterion:

$$\text{for} \quad n = 1, \ldots, N \quad \text{holds} \qquad \sum_{m=1 (m \neq n)}^{N} \frac{|a_{mn}|}{|a_{mm}|} < 1 \ .$$

Square sum criterion:

$$\sum_{m,n=1 (n \neq m)}^{N} |\frac{a_{mn}}{a_{mm}}|^2 < 1$$

Here, the error bounds of Theorem 2.2 can be applied, with:

1. $||.|| = ||.||_\infty, \quad \overline{\rho} = \max_m \frac{\sum_{n=1(n\neq m)}^{N} |a_{mn}|}{|a_{mm}|}.$
2. $||.|| = ||.||_1, \quad \overline{\rho} = \max_n \frac{\sum_{m=1(m\neq n)}^{N} |a_{mn}|}{|a_{mm}|}.$
3. $||.|| = ||.||_2, \quad \overline{\rho} = \sum_{m,n=1(n\neq m)}^{N} \left(\frac{|a_{mn}|}{|a_{mm}|}\right)^2.$

2. Let A be an $N \times N$ matrix with $a_{mm} \neq 0 \ \ (m = 1, \ldots, N)$. If for $m = 1, \ldots, N$:

$$\sum_{n=1 (n \neq m)}^{N} |a_{mn}| \leq |a_{mm}|, \sum_{n=m+1}^{N} |a_{mn}| < |a_{mm}|,$$

is valid, then the Gauss-Seidel method converges (with known $\overline{\rho} < 1$ wrt $||.||_\infty$). This is called the "weak row sum criterion".

3. If A is diagonally dominant, i.e.,

$$|a_{mm}| > \sum_{n \neq m, 1}^{N} |a_{mn}|, \ m = 1, \ldots, N$$

then $||Q_{GS}||_\infty \leq ||Q_{JAC}||_\infty < 1$.

4. If matrix A is a so-called M-matrix, i.e., if

   - $a_{mn} \leq 0$ for $m \neq n$, $a_{mm} \geq \sum_{n=1(n\neq m)}^{n} a_{mn}$ (>-sign for at least one n value),
   - $a_{mm} > 0$,

  - A is irreducible (the matrix cannot be split into independent submatrices),

  then the Gauss-Seidel and Jacobi methods converge and

$$\rho(Q_{GS}) < \rho(Q_{JAC}), \text{ unless } \rho(Q_{GS}) = \rho(Q_{JAC}) = 0.$$

5. Moreover, for M-matrices resulting from the $O(h^2)$-discretization of an equation, such a the Poisson equation, we have

$$\rho(Q_{GS}) = (\rho(Q_{JAC}))^2.$$

Let's consider the convergence of Gauss-Seidel and Jacobi's method for the Poisson equation on a Cartesian grid, with mesh sizes $h_x = h_y = h = 1/\sqrt{N}$.

The eigenfunctions of the Laplace operator $-\Delta_h$ are:

$$\varphi_h^{k,\ell}(mh, nh) = 2h \sin mk\pi h \sin n\ell\pi h \quad k, \ell = 1, 2, \dots \sqrt{N} - 1.$$

The corresponding eigenvalues are

$$\frac{4}{h^2}(\sin^2 \frac{k\pi h}{2} + \sin^2 \frac{\ell\pi h}{2}).$$

Herewith, we obtain as the eigenvalues of the Jacobi iteration:

$$\begin{aligned}
\lambda_{k,\ell} &= 1 - \sin^2 \frac{k\pi h}{2} - \sin^2 \frac{\ell\pi h}{2}, \quad k, \ell : 1, \dots, \sqrt{N} - 1, \text{ or:} \\
\lambda_{k,\ell} &= 1 - \frac{1}{2}(1 - \cos k\pi h) - \frac{1}{2}(1 - \cos \ell\pi h), \quad k, \ell : 1, \dots, \sqrt{N} - 1.
\end{aligned}$$

So, we find the eigenvalues of the Jacobi iteration matrix as:

$$\lambda_{k,\ell} = \frac{1}{2}(\cos k\pi h + \cos \ell\pi h), \quad k, \ell : 1, \dots, \sqrt{N} - 1.$$

For the spectral radius:

$$\begin{aligned}
\rho(Q_{JAC}) &= \cos \pi h = 1 - \frac{1}{2}(\pi h)^2 + O(h^4) \\
\rho(Q_{GS}) &= \cos^2 \pi h = 1 - (\pi h)^2 + O(h^4).
\end{aligned}$$

With a mesh size $h = 1/64$ this gives for the asymptotic convergence factor:

$$\rho_{JAC} = 0.998, \ \rho_{GS} = 0.9976.$$

The convergence of Gauss-Seidel can be improved by *inclusion of a relaxation parameter* ω as follows,

$$\begin{cases} u_m^* = \frac{1}{a_{mm}}(b_m - \sum_{k=1}^{m-1} a_{mk} u_k^{i+1} - \sum_{k=m+1}^{N} a_{mk} u_k^i), \quad m = 1, \dots, N, \\ u_m^{i+1} = u_m^i + \omega(u_m^* - u_m^i) \end{cases}$$

This method is called “Succesive Overrelaxation” (SOR), because $\omega > 1$. With $\omega = 1$, we have the Gauss-Seidel method. The iteration matrix Q_{SOR} reads for SOR:

$$Q_{SOR} = (D - \omega C_1)^{-1}((1-\omega)D + \omega C_2), \quad s = \omega(D - \omega C_1)^{-1} b \ .$$

The optimal relaxation parameter ω for which $\rho(Q_{SOR})$ is minimal for Poisson's equation is

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho(Q_{JAC})^2}}.$$

In that case, we get

$$\rho(Q_{SOR})|_{\omega_{opt}} = \omega_{opt} - 1.$$

For Poisson's equation, we find:

$$\omega_{opt} = \frac{2}{1 + \sin \pi h}, \quad \rho(Q_{SOR})|_{\omega_{opt}} = \omega_{opt} - 1 = 1 - 2\pi h + O(h^2).$$

For $h = 1/64$, we find for the Poisson equation: $\rho(Q_{SOR}) \approx 0.90$.

We have

$$\rho(Q_{GS}) = 1 - (\pi h)^2 + O(h^4), \rho(Q_{SOR}) = 1 - 2\pi h + O(h^2).$$

Suppose we require an error reduction by 10^{-d} as in (2.2.10) when solving the discrete Poisson equation, then we obtain for the number of iterations i_{GS} and i_{SOR}:

$$\begin{aligned} i_{GS} \geq \ln 10 \cdot d/(\pi h)^2 [1 + O(h^2)] &\approx \ln 10 d N / \pi^2 \\ i_{SOR} \geq \ln 10 \cdot d/(2\pi h)[1 + O(h)] &\approx \ln 10 d \sqrt{N}/(2\pi) \end{aligned}$$

The work for each grid point is for Gauss-Seidel 8 and for SOR 11 flops per grid point per iteration for Poisson's equation. For the total amount of work we find:

$$\begin{aligned} W_{GS} &\approx ln 10 \frac{8d}{\pi^2} N^2 \\ W_{SOR} &\approx ln 10 \frac{11d}{2\pi} N^{3/2}. \end{aligned}$$

The total amount of work required by these single grid basic iterative solvers is of order $O(N^\alpha)$, with $\alpha > 1$. This will lead, especially for large values of N, to extremely slowly converging iterative methods. A speed-up is obtained by including hierarchy into them or by an acceleration via Krylov subspace iterative solvers.

2.3.3 Exercises

Exercise 2.3.1 Consider the problem

$$\begin{aligned} 10u_1 + u_2 &= 1 \\ u_1 + 10u_2 &= 10 \end{aligned}$$

with the solution $\boldsymbol{u} = (u_1, u_2)^T = (0, 1)^T$. For a general system of equations

$$A\boldsymbol{u} = \boldsymbol{b}$$

with an $n \times n$ matrix A, which consists of a lower triangular submatrix L, the diagonal D and the upper triangular submatrix U ($A = L + D + U$), Gauss–Seidel iteration is defined by

$$\boldsymbol{u}^{i+1} = D^{-1}\left(b - L\boldsymbol{u}^{i+1} - U\boldsymbol{u}^i\right)$$

and Jacobi iteration is given by

$$\boldsymbol{u}^{i+1} = D^{-1}\left(b - L\boldsymbol{u}^{i} - U\boldsymbol{u}^i\right)$$

Perform

a) four Gauss–Seidel iterations,
b) four Jacobi iterations,

starting with the initial approximation $(u_1, u_2) = (0, 0)$.

Exercise 2.3.2 Perform

a) four Gauss–Seidel iterations,
b) four Jacobi iterations,

starting with the initial approximation $(u_1, u_2) = (0, 0)$, for the problem

$$\begin{aligned} u_1 + u_2 &= 2 \\ 4u_1 + 5u_2 &= 9 \end{aligned}$$

The exact solution is $(u_1, u_2) = (1, 1)$. Compare the convergence speed of both iterations with that in Exercise 2.3.1.

Exercise 2.3.3 Let the matrix A be as in Exercise 2.2.3. This matrix is a discretization of $\mathcal{L} = -d^2/dx^2$ with homogeneous boundary conditions.

a) Compute the eigenvalues and eigenvectors of L_h. What is the spectral norm of L_h?

b) Sketch the eigenvectors belonging to the largest and smallest eigenvalues as grid functions on the grid $[0, 1, 2, 3, 4, 5]$, appending zeros before and after them.

Exercise 2.3.4 Apply the lexicographic Gauss-Seidel method for the five-point stencil on a grid of mesh size h one time at the point (x_5, y_6) given that

a) $b_h(x_m, y_n) = 0$ for all m, n, $u_h^{i+1}(x_5 - h, y_6) = 1$, $u_h^{i+1}(x_5, y_6 - h) = 1$, $u_h^i(x_5 + h, y_6) = 1$, $u_h^i(x_5, y_6 + h) = 1$.

b) $b_h(x_m, y_n) = 0$ for all m, n, $u_h^{i+1}(x_5 - h, y_6) = -1$, $u_h^{i+1}(x_5, y_6 - h) = -1$, $u_h^i(x_5 + h, y_6) = 1$, $u_h^i(x_5, y_6 + h) = 1$.

Exercise 2.3.5

a) Show that the Jacobi iterative method, but not the Gauss-Seidel method, converges for

$$A = \begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & -1 \\ -2 & -2 & 1 \end{pmatrix}.$$

b) Show that the Gauss-Seidel method, but not the Jacobi method, converges for

$$A = \frac{1}{2}\begin{pmatrix} 2 & 1 & 1 \\ -2 & 2 & -2 \\ -1 & 1 & 2 \end{pmatrix}.$$

3 Krylov Subspace Methods

3.1 Method for Systems with a Symmetric Positive Definite Matrix

3.1.1 Introduction

In the basic iterative solution methods we compute the iterates by the following recursion:

$$u^{i+1} = u^i + M^{-1}(b - Au^i) = u^i + M^{-1}r^i$$

Writing out the first steps of such a process we obtain:

$$\begin{array}{lll} u^0, & & \\ u^1 & = & u^0 + (M^{-1}r^0), \\ u^2 & = & u^1 + (M^{-1}r^1) = u^0 + M^{-1}r^0 + M^{-1}(b - Au^0 - AM^{-1}r^0), \\ & = & u^0 + 2M^{-1}r^0 - M^{-1}AM^{-1}r^0, \\ & \vdots & \end{array}$$

This implies that

$$u^i \in u^0 + \text{ span } \left\{ M^{-1}r^0, M^{-1}A(M^{-1}r^0), \ldots, (M^{-1}A)^{i-1}(M^{-1}r^0) \right\}.$$

The subspace $K^i(A; r^0) := \text{span}\{r^0, Ar^0, \ldots, A^{i-1}r^0\}$ is called the Krylov-space of dimension i corresponding to matrix A and initial residual r^0. A u^i calculated by a basic iterative method is an element of $u^0 + K^i(M^{-1}A; M^{-1}r^0)$.

In this chapter we shall describe the Conjugate Gradient method. This method minimizes the error $u - u^i$ in an adapted norm, without any information about the eigenvalues. In Section 3.1.3 we give theoretical results concerning the convergence behavior of the CG method.

3.1.2 The Conjugate Gradient (CG) method

In this section we assume that $M = I$, and $u^0 = 0$ so $r^0 = b$. These assumptions are only needed to facilitate the formula's. They are not necessary for the CG method itself. Furthermore we assume that A satisfies the following condition.

Condition 2.1.2:
The matrix A is symmetric ($A = A^T$) and positive definite ($x^TAx > 0$ for $x \neq 0$).

This condition is crucial for the derivation and success of the CG method. Later on we shall derive extensions to non-symmetric matrices.

The first idea would be to construct a vector $u^i \in K^i(A, r^0)$ such that $\|u - u^i\|_2$ is minimal. The first iterate u^1 can be written as $u^1 = \alpha_0 r^0$ where α_0 is a constant which has to be chosen such that $\|u - u^1\|_2$ is minimal. This leads to

$$\|u - u^1\|_2^2 = (u - \alpha_0 r^0)^T (u - \alpha_0 r^0) = u^T u - 2\alpha_0 (r^0)^T u + \alpha_0^2 (r^0)^T r_0 \; . \qquad (3.1.1)$$

The norm given in (3.1.1) is minimized if $\alpha_0 = \dfrac{(r^0)^T u}{(r^0)^T r^0}$. Since u is unknown this choice cannot be determined, so this idea does not lead to a useful method. Note that $Au = b$ is known so using an adapted inner product implying A could lead to an α_0 which is easy to calculate. To follow this idea we define the following inner product and related norm.

Definition 2.1.3
The A-inner product is defined by $(y, z)_A = y^T Az$, and the A-norm by

$$\|y\|_A = \sqrt{(y, y)_A} = \sqrt{y^T Ay}.$$

It is easy to show that if A satisfies Condition 2.1.2 $(.,.)_A$ and $\|.\|_A$ satisfy the rules for inner product and norm, respectively. In order to obtain u^1 such that $\|u - u^1\|_A$ is minimal we note that

$$\|u - u^1\|_A^2 = u^T Au - 2\alpha_0 (r^0)^T u + \alpha_0^2 \, (r^0)^T Ar^0,$$

so $\alpha_0 = \dfrac{(r^0)^T Au}{(r^0)^T Ar^0} = \dfrac{(r^0)^T b}{(r^0)^T Ar^0}$. We see that this new inner product leads to a minimization problem, which can easily be solved. In the following iterations, we compute u^i such that

$$\|u - u^i\|_A = \min_{y \in K^i(A; r^0)} \|u - y\|_A \qquad (3.1.2)$$

The solution of this minimization problem leads to the Conjugate Gradient method. First we specify the CG method, thereafter we summarize some of its properties.

Conjugate Gradient method

```
$k = 0$ ;   $u^0 = 0$ ; $r^0 = b$                              initialization
while       $r^k \neq 0$ do                                    termination criterion
            $k := k + 1$                                       $k$ is the iteration number
            if $k = 1$ do
                $p^1 = r^0$
            else
                $\beta_k = \frac{(r^{k-1})^T r^{k-1}}{(r^{k-2})^T r^{k-2}}$      $p^k$ is the search direction vector
                $p^k = r^{k-1} + \beta_k p^{k-1}$               to update $u^{k-1}$ to $u^k$
            end if
            $\alpha_k = \frac{(r^{k-1})^T r^{k-1}}{(p^k)^T A p^k}$
            $u^k = u^{k-1} + \alpha_k p^k$                     update iterate
            $r^k = r^{k-1} - \alpha_k A p^k$                   update residual
end while
```

The first description of this algorithm is given in [76]. Besides the two vectors u^k, r^k and matrix A, only one extra vector p^k should be stored in memory. Note that the vectors from a previous iteration can be overwritten. One iteration of CG costs one matrix vector product and 10 N flops for vector calculations. If the CG algorithm is used in a practical application the termination criterion should be replaced by one of the criteria given in Section 2.3.1. In this algorithm r^k is computed from r^{k-1} by the equation $r^k = r^{k-1} - \alpha_k A p^k$. This is done in order to save one matrix vector product for the original calculation $r^k = b - Au^k$. In some applications the updated residual obtained from the CG algorithm can deviate significantly from the exact residual $b - Au^k$ due to rounding errors. It is therefore strongly recommended to recompute $b - Au^k$ after the termination criterion is satisfied for the updated residual and compare the norm of the exact and updated residual. If the exact residual does no satisfy the termination criterion the CG method should be restarted with u^k as its starting vector.

The vectors defined in the CG method have the following properties:

Theorem 3.1

1. $$\textit{span}\ \{p^1, \ldots, p^k\} = \textit{span}\{r^0, \ldots, r^{k-1}\} = K^k(A; r^0), \quad (3.1.3)$$
2. $$(r^j)^T r^i = 0 \quad i = 0, \ldots, j-1 \quad ; \quad j = 1, \ldots, k \quad , \quad (3.1.4)$$
3. $$(r^j)^T p^i = 0 \quad i = 1, \ldots, j \quad ; \quad j = 1, \ldots, k \quad , \quad (3.1.5)$$
4. $$(p^j)^T A p^i = 0 \quad i = 1, \ldots, j-1 \quad ; \quad j = 2, \ldots, k \quad (3.1.6)$$
5. $$\|u - u^k\|_A = \min_{y \in K^k(A; r^0)} \|u - y\|_A. \quad (3.1.7)$$

Proof: see [57], Section 10.2.

Some remarks on the properties given in Theorem 3.1 are:

- It follows from (3.1.3) and (3.1.4) that the vectors $r^0, \ldots, r^{k-1}$ form an *orthogonal basis* of $K^k(A; r^0)$.

- In theory the CG method is a finite method. After N iterations the Krylov subspace is identical to $I\!R^N$. Since $\|u - y\|_A$ is minimized over $K^N(A; r^0) = I\!R^N$ the norm should be equal to zero and $u_N = u$. However, in practice this property is never utilized for two reasons: firstly in many applications N is very large so that it is not feasible to perform N iterations; secondly even if N is small, rounding errors can spoil the results such that the properties given in Theorem 3.1 do not hold for the computed vectors.

- The sequence $\|u - u^k\|_A$ is theoretically monotonically decreasing, so
$$\|u - u^{k+1}\|_A \leq \|u - u^k\|_A \, .$$
This follows from (3.1.7) and the fact that $K^k(A; r^0) \subset K^{k+1}(A; r^0)$. In practice, $\|u - u^k\|_A$ is not easily computed since u is unknown. The norm of the residual is given by $\|r^k\|_2 = \|u - u^k\|_{A^TA}$. This sequence is not necessarily monotonically decreasing. In applications it may occur that $\|r^{k+1}\|_2$ is larger than $\|r^k\|_2$. This does not mean that the CG process becomes divergent. The inequality
$$\|r^k\|_2 = \|Au^k - b\|_2 \leq \sqrt{\|A\|_2} \|u - u^k\|_A$$
shows that $\|r^k\|_2$ is less than the monotonically decreasing sequence $\sqrt{\|A\|_2}\|u - u^k\|_A$, so after some iterations the norm of the residual decreases again.

- The direction vector p^j is *A-orthogonal or A-conjugate* to all p^i with index i less than j. This is the motivation for the name of the method: the directions or gradients of the updates are mutually conjugate.

- In the algorithm we see two ratios, one in calculating β_k and the other one for α_k. If the denominator is equal to zero, the CG method breaks down. With respect to β_k this implies that $(r^{k-2})^T r^{k-2} = 0$, which implies $r^{k-2} = 0$ and thus $u^{k-2} = u$. The linear system is solved. The denominator of α_k is zero if $(p^k)^T A p^k = 0$, so if $p^k = 0$. Using property (3.1.3) this implies that $r^{k-1} = 0$ so, again, the problem is already solved.
The conclusion is that if the matrix A satisfies Condition 2.1.2 then the CG method is robust.

In the following chapter we shall give CG type methods for more general matrices A, but first we shall extend Condition 2.1.2 in such a way that also singular matrices are permitted. If the matrix A is symmetric and positive *semi* definite ($x^TAx \geq 0$) the CG method can be used to solve the linear system $Au = b$, provided b is an element of the column space of A (range(A)). This is a natural condition because if it does not hold there would be no vector u such that $Au = b$. For further details and references see [85].

3.1.3 The Convergence Behavior of the CG Method

An important topic is the rate of convergence of the CG method. The optimality property enables one to obtain easy to calculate upper bounds of the distance between the k^{th} iterate and the exact solution.

Theorem 3.2 *The iterates u^k obtained from the CG algorithm satisfy the following inequality:*

$$\|u - u^k\|_A \leq 2\left(\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1}\right)^k \|u - u^0\|_A. \qquad (3.1.8)$$

<u>*Proof*</u>
We shall only give a sketch of the proof. It is easily seen that $u - u^k$ can be written as a polynomial, say $p_k(A)$ with $p_k(0) = 1$, times the initial residual

$$\|u - u^k\|_A = \|p_k(A)(u - u^0)\|_A.$$

Due to the minimization property every other polynomial $q_k(A)$ with $q_k(0) = 1$ does not decrease the error measured in the A-norm:

$$\|u - u^k\|_A \leq \|q_k(A)(u - u^0)\|_A.$$

The right-hand side can be written as

$$\|q_k(A)(u-u^0)\|_A = \|q_k(A)\sqrt{A}(u-u^0)\|_2 \leq \|q_k(A)\|_2\|\sqrt{A}(u-u^0)\|_2 = \|q_k(A)\|_2\|u-u^0\|_A$$

Taking $q_k(A)$ equal to the Chebyshev polynomial (see, for example, [57]) gives the desired result. $\square$

Note that the rate of convergence of CG is comparable to that of the Chebyshev method, however it is not necessary to estimate or calculate eigenvalues of the matrix A. Furthermore increasing diagonal dominance leads to a better rate of convergence.

Initially, the CG method was not popular. The reason for this is that the convergence can be slow for systems where the condition number $\kappa_2(A)$ is very large.

On the other hand the fact that the solution is found in N iterations is also not useful in practice, as N may be very large, and the property does not hold in the presence of rounding errors. To illustrate this we consider the following classical example:

Example 3.1.1 *The linear system $Au = b$ is to be solved where $N = 40$ and $b = (1, 0, \ldots, 0)^T$. The matrix A is given by*

$$A = \begin{bmatrix} 5 & -4 & 1 & & & & & & \\ -4 & 6 & -4 & 1 & & & & \oslash & \\ 1 & -4 & 6 & -4 & 1 & & & & \\ & \ddots & \ddots & \ddots & \ddots & & & & \\ & & \ddots & \ddots & \ddots & \ddots & & & \\ & & & & 1 & -4 & 6 & -4 & 1 \\ & \oslash & & & & 1 & -4 & 6 & -4 \\ & & & & & & 1 & -4 & 5 \end{bmatrix}.$$

This can be interpreted as a finite difference discretization of the bending beam equation:
$u'''' = f$. The eigenvalues of this matrix are given by:

$$\lambda_k = 16 sin^4 \frac{k\pi}{82} \quad k = 1, \ldots, 40.$$

The matrix A is symmetric positive definite, so the CG method can be used to solve the linear system. The condition number of A is approximately equal to $(82/\pi)^4$. The resulting rate of convergence given by

$$\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1} \cong 0.997$$

is close to one. This explains a slow convergence of the CG method for the first iterations. However after 40 iterations the solution should be obtained. In Figure 3.1.1 the convergence behavior is given where the rounding error is equal to 10^{-16}, [56].

This example suggests that CG has only a restricted range of applicability. This idea changed, however, completely after the publication of [113]. Herein it has been shown that the CG method can be very useful for a class of linear systems, not as a direct method, but as an iterative method. The problem class originates from discretized partial differential equations. It appears that not the size of the matrix is important for convergence but the extreme eigenvalues of A.

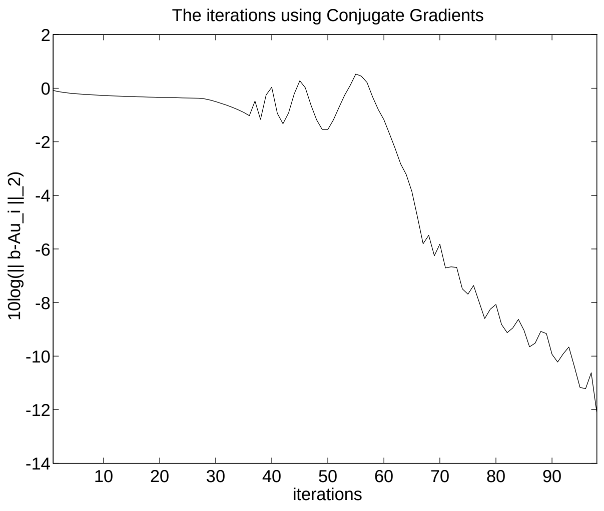


Figure 3.1.1: The convergence behavior of CG.

One of the results which is based on the extreme eigenvalues is given in Theorem 3.2. Inequality (3.1.8) is an upper bound for the error of the CG iterates, and suggests that the CG method is a linearly convergent process (see Figure 3.1.2). However, in practice the convergence behavior looks like the one given in Figure 3.1.3. This is called *superlinear convergence behavior*. So, the upper bound is only sharp for the initial iterates. It seems that after some iterations the condition number in Theorem 3.2 is replaced by a smaller condition number. To illustrate this we give the following example:

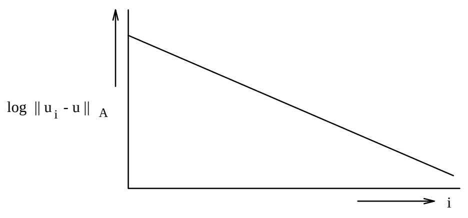


Figure 3.1.2: A linear convergent behavior

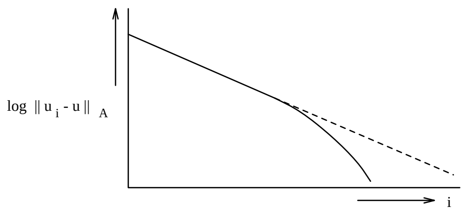


Figure 3.1.3: A super linear convergent behavior

Example 3.1.2 *The matrix A is from the discretized Poisson operator. The physical domain is a two-dimensional unit square. The grid used consists of an equidistant distribution of* 30×30 *grid points. The dimension of A is equal to* 900 *and the eigenvalues are given by*

$$\lambda_{k,\ell} = 4 - 2cos\frac{\pi k}{31} - 2cos\frac{\pi \ell}{31} \quad , \quad 1 \leq k, \ell \leq 30.$$

Using Theorem 3.2 it appears that 280 *iterations are necessary to ensure that*

$$\frac{\|u - u^i\|_A}{\|u - u^0\|_A} \leq 10^{-12}.$$

Computing the solution it appears that the CG iterates satisfy the given termination criterion after 120 *iterations. So, for this example the estimate given in Theorem 3.2 is not sharp.*

To obtain a better insight in the convergence behavior we have a closer look into the CG method. We have seen that CG minimizes $\|u - u^i\|_A$ in the Krylov subspace. This can also be seen as the construction of a polynomial q_i of degree i and $q_i(0) = 1$ such that

$$\|u - u^i\|_A = \|q_i(A)(u - u^0)\|_A = \min_{\substack{\tilde{q_i}, \\ \tilde{q}_i(0) = 1}} \|\tilde{q}_i(A)(u - u^0)\|_A \ .$$

Suppose that the orthonormal eigensystem of A is given by: $\{\lambda_j, y^j\}_{j=1,\ldots,n}$ where $Ay^j = \lambda_j y^j$, $\|y^j\|_2 = 1$, $(y^j)^T y^i = 0, j \neq i$, and $0 < \lambda_1 \leq \lambda_2 \ldots \leq \lambda_N$. The initial errors can be written as $u - u^0 = \sum\limits_{j=1}^{N} \gamma_j y^j$, which implies that

$$u - u^i = \sum_{j=1}^{N} \gamma_j q_i(\lambda_j) y^j \ . \qquad (3.1.9)$$

If for instance $\lambda_1 = \lambda_2$ and $\gamma_1 \neq 0$ and $\gamma_2 \neq 0$ it is always possible to change y^1 and y^2 to $\tilde{y}_1$ and $\tilde{y}_2$ such that $\tilde{\gamma}_1 \neq 0$ but $\tilde{\gamma}_2 = 0$. This, in combination with equation (3.1.9), implies that if $q_i(\lambda_j) = 0$ for all different λ_j then $u^i = u$. So if there are only $m < N$ different eigenvalues the CG method stops at least after m iterations. Furthermore, the upper bound given in Theorem 3.2 can be sharpened.

Remark 3.1.1 For a given linear system $Au = b$ and a given u^0 (note that $u - u^0 = \sum\limits_{j=1}^{N} \gamma_j y^j$) the quantities α and β are defined by:

$$\alpha = \min \{\lambda_j | \gamma_j \neq 0\} ,$$
$$\beta = \max \{\lambda_j | \gamma_j \neq 0\} .$$

It is easy to show that the following inequality holds:

$$\|u - u^i\|_A \leq 2 \left(\frac{\sqrt{\frac{\beta}{\alpha}} - 1}{\sqrt{\frac{\beta}{\alpha}} + 1} \right)^i \|u - u^0\|_A. \tag{3.1.10}$$

The ratio $\frac{\beta}{\alpha}$ is called the *effective condition number* of A. $\gg$

It follows from Theorem 3.1 that $r^0, \ldots, r^{k-1}$ form an orthogonal basis for $K^k(A; r^0)$. Then, the vectors $\tilde{r^i} = r^i/\|r^i\|_2$ form an orthonormal basis for $K^k(A; r^0)$. We define the following matrices

$$\begin{array}{l} R_k \in I\!R^{N \times k} \text{ and the } j^{\text{t}h} \text{ column of } R_k \text{ is } \tilde{r^j}, \\ T_k = R_k^T A R_k \text{ where } T_k \in I\!R^{k \times k}. \end{array}$$

Matrix T_k can be seen as the projection of A on $K^k(A; r^0)$. It follows from Theorem 3.1 that T_k is a tridiagonal symmetric matrix. The coefficients of T_k can be calculated from the α_i's and β_i's of the CG process. The eigenvalues θ_i of the matrix T_k are called *Ritz values* of A with respect to $K^k(A; r^0)$. If z^i is an eigenvector of T_k so that $T_k z^i = \theta_i z^i$ and $\|z^i\|_2 = 1$ then $R_k\ z^i$ is called a *Ritzvector* of A. Ritzvalues and Ritzvectors are approximations of the eigenvalues and eigenvectors and play an important role in a better understanding of the convergence behavior of CG. Some important properties are:

- the rate of convergence of a Ritzvalue to its limit eigenvalue depends on the distance of this eigenvalue to the rest of the spectrum.

- in general the extreme Ritzvalues converge fastest and their limits are α and β.

In practical experiments we see that, if Ritzvalues approximate the extreme eigenvalues of A, then the rate of convergence appears to be based on a smaller effective condition number (the extreme eigenvalues seem to be absent). We first give an heuristic explanation. Thereafter an exact result from the literature is cited.

From Theorem 3.1 it follows that $r^k = A(u - u^k)$ is perpendicular to the Krylov subspace $K^k(A; r^0)$. If a Ritzvector is a good approximation of an eigenvector y^j of A this eigenvector is "nearly" contained in the subspace $K^k(A; r^0)$. These observations combined yield that $(A(u - u^k))^T y^j \cong 0$. The exact solution and the approximation can be written as

$$u = \sum_{i=1}^{N} (u^T y^i) y^i \quad \text{and} \quad u^k = \sum_{i=1}^{N} ((u^k)^T y^i) y^i.$$

From $(A(u - u^k))^T y^j = (u - u^k)^T \lambda_j y^j \cong 0$ it follows that $x^T y^j \cong (u^k)^T y^j$. So the error $u - u^k$ has a negligible component in the eigenvector y^j. This suggests that

λ_j does no longer influence the convergence of the CG process.

For a more precise result we define a *comparison process*. The iterates of this process are comparable to that of the original process, but its condition number is less than that of the original process.

Definition
Let u^i be the i-th iterate of the CG process for $Au = b$. For a given integer i let $\overline{u}^j$ denote the j-th iterate of the comparison CG process for this equation, starting with $\overline{u}^0$ such that $u - \overline{u}^0$ is the projection of $u - u^i$ on $\text{span}\{y^2, \ldots, y^N\}$.

Note that for the comparison process the initial error has no component in the y^1 eigenvector.

Theorem 3.3 *[128]*
Let u^i be the i-th iterate of CG, and $\overline{u}^j$ the j-th iterate of the comparison process. Then for any j there holds:

$$\|u - u^{i+j}\|_A \leq F_i \|u - \overline{u}^j\|_A \leq F_i \frac{\|u - \overline{u}^j\|_A}{\|u - \overline{u}^0\|_A} \|u - u^i\|_A$$

with $\quad F_i = \dfrac{\theta_1^i}{\lambda_1} \max\limits_{k \geq 2} \dfrac{|\lambda_k - \lambda_1|}{|\lambda_k - \theta_1^i|}$, *where θ_1^i is the smallest Ritz value in the i-th step of the CG process.*

Proof: see [128], Theorem 3.1.

The theorem shows that from any stage i for which θ_1^i does not coincide with an eigenvalue λ_k, the error reduction in the next j steps is at most a fixed factor F_i worse than the error reduction in the first j steps of the comparison process in which the error vector has no y_1-component. As an example we consider the case that $\lambda_1 < \theta_1^i < \lambda_2$ we then have

$$F_i = \frac{\theta_1^i}{\lambda_1} \frac{\lambda_2 - \lambda_1}{\lambda_2 - \theta_1^i},$$

which is a kind of *relative convergence measure* for θ_1^i relative to λ_1 and $\lambda_2 - \lambda_1$. If $\frac{\theta_1^i - \lambda_1}{\lambda_1} < 0.1$ and $\frac{\theta_1^i - \lambda_1}{\lambda_2 - \lambda_1} < 0.1$ then we have $F_i < 1.25$. Hence, already for this modest degree of convergence of θ_1^i the process virtually converges as well as the comparison process (as if the y_1-component was not present). For more general results and experiments we refer to [128].

3.1.4 Exercises

a. Show that $(y, z)_A = \sqrt{y^T A z}$ is an inner product if A is symmetric and positive definite.

b. Give the proof of inequality (3.1.10).

c. (a) Show that an A-orthogonal set of nonzero vectors associated with a symmetric and positive definite matrix is linearly independent.

   (b) Show that if $\{v^1, v^2, \ldots, v^N\}$ is a set of A-orthogonal vectors in $I\!R^N$ and $z^T v^i = 0$ for $i = 1, \ldots, N$ then $z = 0$.

d. Define
$$t_k = \frac{(v^k, b - Au^{k-1})}{(v^k, Av^k)}$$
and $u^k = u^{k-1} + t_k v^k$, then $(r^k, v^j) = 0$ for $j = 1, \ldots, k$, if the vectors v^j form an A-orthogonal set. To prove this, use the following steps using mathematical induction:

   (a) Show that $(r^1, v^1) = 0$.

   (b) Assume that $(r^k, v^j) = 0$ for each $k \leq l$ and $j = 1, \ldots, k$ and show that this implies that
$$(r^{l+1}, v^j) = 0 \text{ for each } j = 1, \ldots, l.$$

   (c) Show that $(r^{l+1}, v^{l+1}) = 0$.

e. Take $A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$ and $b = \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}$. We solve $Au = b$.

   (a) Show that Conjugate Gradients applied to this system should convergence in 1 or 2 iterations (using the convergence theory).

   (b) Choose $u^0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ and do 2 iterations with the Conjugate Gradient method.

f. Suppose that A is symmetric and indefinite. Give an example that shows that the Conjugate Gradient method can break down.

3.2 Preconditioning of Krylov Subspace Methods

We have seen that the convergence behavior of Krylov subspace methods strongly depends on the eigenvalue distribution of the coefficient matrix. A preconditioner is a matrix that transforms the linear system such that the transformed system has the same solution but the transformed coefficient matrix has a *more favorable spectrum*. As an example we consider a matrix M which resembles the matrix A. The transformed system is given by

$$M^{-1}Au = M^{-1}b \ ,$$

and has the same solution as the original system $Au = b$. The requirements on the matrix M are the following:

- the eigenvalues of $M^{-1}A$ should be clustered around 1,
- it should be possible to obtain $M^{-1}y$ at low cost.

Most of this chapter contains preconditioners for symmetric positive definite systems (Section 3.2.1). For non-symmetric systems the ideas are analogous. So, in Section 3.3.7 we give some details, which can be used only for non-symmetric systems.

3.2.1 The Preconditioned Conjugate Gradient (PCG) method

In Section 3.1.3 we observed that the rate of convergence of CG depends on the eigenvalues of A. Initially the condition number $\frac{\lambda_N}{\lambda_1}$ determines the decrease of the error. After a number of iterations $\frac{\lambda_N}{\lambda_1}$ is replaced by the effective condition number $\frac{\lambda_N}{\lambda_2}$ etc. So the question arises, whether it is possible to change the linear system $Au = b$ in such a way that the eigenvalue distribution becomes more favorable with respect to the CG convergence? This is indeed possible and the approach is known as: *the preconditioning of a linear system.* Consider the $N \times N$ symmetric positive definite linear system $Au = b$. The idea behind Preconditioned Conjugate Gradients is to apply the "original" Conjugate Gradient method to the transformed system

$$\tilde{A}\tilde{u} = \tilde{b} \ ,$$

where $\tilde{A} = P^{-1}AP^{-T}$, $u = P^{-T}\tilde{u}$ and $\tilde{b} = P^{-1}b$, and P is a nonsingular matrix. The matrix M defined by $M = PP^T$ is called the preconditioner. The resulting algorithm can be rewritten in such a way that only quantities without a ˜ tilde appear.

Preconditioned Conjugate Gradient method

```
k = 0 ;    $u^0 = 0$ ;    $r^0 = b$ ;    initialization
while      ($r^k \neq 0$) do             termination criterion
           $z^k = M^{-1}r^k$             preconditioning
           k := k + 1
           if $k = 1$ do
              $p^1 = z^0$
           else
              $\beta_k = \frac{(r^{k-1})^T z^{k-1}}{(r^{k-2})^T z^{k-2}}$    update of $p^k$
              $p^k = z^{k-1} + \beta_k p^{k-1}$
           end if
           $\alpha_k = \frac{(r^{k-1})^T z^{k-1}}{(p^k)^T A p^k}$
           $u^k = u^{k-1} + \alpha_k p^k$        update iterate
           $r^k = r^{k-1} - \alpha_k A p^k$      update residual
end while
```

Observations and properties for this algorithm are:

- it can be shown that the residuals and search directions satisfy:

$$\begin{aligned}(r^j)^T M^{-1} r^i &= 0 \quad , \quad i \neq j \; , \\ (p^j)^T (P^{-1}AP^{-T}) p^i &= 0 \quad , \quad i \neq j \; .\end{aligned}$$

- The denominators $(r^{k-2})^T z^{k-2} = (z^{k-2})^T M z^{k-2}$ never vanish for $r^{k-2} \neq 0$ because M is a positive definite matrix.

With respect to the matrix P we have the following requirements:

- the multiplication of $P^{-T}P^{-1}$ by a vector should be cheap (comparable with a matrix vector product using A). Otherwise one iteration of PCG is much more expensive than one iteration of CG and hence preconditioning leads to a more expensive algorithm.

- The matrix $P^{-1}AP^{-T}$ should have a favorable distribution of the eigenvalues. It is easy to show that the eigenvalues of $P^{-1}AP^{-T}$ are the same as for $P^{-T}P^{-1}A$ and $AP^{-T}P^{-1}$. So we can choose one of these matrices to study the spectrum.

In order to give more details on the last requirement we note that the iterate u^k obtained by PCG satisfies

$$u^k \in u^0 + K^k(P^{-T}P^{-1}A \; ; \; P^{-T}P^{-1}r^0), \text{ and} \tag{3.2.1}$$

$$\|u - u^k\|_{P^{-1}AP^{-T}} \leq 2 \left(\frac{\sqrt{\kappa_2(P^{-1}AP^{-T})} - 1}{\sqrt{\kappa_2(P^{-1}AP^{-T})} + 1} \right)^k \|u - u^0\|_{P^{-1}AP^{-T}} \ . \qquad (3.2.2)$$

So a small condition number of $P^{-1}AP^{-T}$ leads to fast convergence. Two extreme choices of P show the possibilities of PCG. Choosing $P = I$ we have the original CG method, whereas if $P^T P = A$ the iterate u^1 is equal to u so PCG converges in one iteration. For a classical paper on the success of PCG we refer to [101]. In the following pages some typical preconditioners are discussed.

Diagonal scaling A simple choice for P is a diagonal matrix with diagonal elements $p_{ii} = \sqrt{a_{ii}}$. In [127] it has been shown that this choice minimizes the condition number of $P^{-1}AP^{-T}$ if P is a diagonal matrix. For this preconditioner it is advantageous to apply CG to $\tilde{A}\tilde{u} = \tilde{b}$. The reason is that $P^{-1}AP^{-T}$ is easily calculated. Furthermore $diag\ (\tilde{A}) = 1$ which saves n multiplications in the matrix vector product.

Basic iterative method The basic iterative methods use a splitting of the matrix $A = M - R$. In the beginning of Section 3.1.2 we showed that the i-th iterate y^i from a basic method is an element of $u^0 + K^i(M^{-1}A, M^{-1}r^0)$. Using this matrix M in the PCG method we see that the iterate u^i obtained by PCG satisfies the following inequality:

$$\|u - u^i\|_{P^{-1}AP^{-T}} = \min_{z \in K^i(M^{-1}A; M^{-1}r^0)} \|u - z\|_{P^{-1}AP^{-T}} \ .$$

This implies that $\|u - u^i\|_{P^{-1}AP^{-T}} \leq \|u - y^i\|_{P^{-1}AP^{-T}}$, so measured in the $\| \ . \ \|_{P^{-1}AP^{-T}}$ norm the error of a PCG iterate is less than the error of a corresponding result of a basic iterative method. The extra costs to compute a PCG iterate with respect to the basic iterate are in general negligible. This leads to the notion that any basic iterative method based on the splitting $A = M - R$ can be accelerated by the Conjugate Gradient method so long as M (the preconditioner) is symmetric and positive definite.

Incomplete decomposition This type of preconditioner is a combination of an iterative method and an approximate direct method. As illustration we use the model problem. The coefficient matrix of this problem $A \in I\!R^{N \times N}$ is a matrix with at most 5 nonzero elements per row. Furthermore the matrix is symmetric and positive definite. The nonzero diagonals are numbered as follows:

m is number of grid points in the x-direction.

$$A = \begin{bmatrix} a_1 & b_1 & & c_1 & & & & & \\ b_1 & a_2 & b_2 & & c_2 & & & & \\ \vdots & \ddots & \ddots & & & \ddots & & & \emptyset \\ c_1 & & b_m & a_{m+1} & b_{m+1} & & c_{m+1} & & \\ & \ddots & \emptyset & \ddots & \ddots & \ddots & \emptyset & \ddots & \\ & \emptyset & & & & & & & \end{bmatrix} \qquad (3.2.3)$$

An optimal choice with respect to convergence is to take a lower triangular matrix L such that $A = L^T L$ and $P = L$ (L is the Cholesky factor). However it is well known that the zero elements in the band of A become non zero elements in the band of L. So the amount of work to construct L is large. With respect to memory we note that A can be stored in $3N$ memory positions, whereas L needs m . N memory positions. For large problems the memory requirements are not easily fulfilled.

If the Cholesky factor L is calculated one observes that the absolute value of the elements in the band of L decreases considerably if the "distance" to the non zero elements of A increases. The non zero elements of L on positions where the elements of A are zero are called fill-in (elements). The observation of the decrease of fill-in motivates to discard fill-in elements entirely, which leads to an incomplete Cholesky decomposition of A. Since the Cholesky decomposition is very stable this is possible without break down for a large class of matrices. The matrix of our model problem is an M-matrix. Furthermore, we give a notation for the elements of L that should be kept to zero. The set of all pairs of indices of off-diagonal matrix entries is denoted by

$$Q_N = \{(i,j) |\, i \neq j \ , \ 1 \leq i \leq N \ , \ 1 \leq j \leq N \ \} \ .$$

The subset Q of Q_N are the places (i,j) where L should be zero. Now the following theorem can be proved:

Theorem 3.4 *If A is a symmetric M-matrix, there exists for each $Q \subset Q_N$ (with the property that $(i,j) \in Q$ implies $(j,i) \in Q$), a uniquely defined lower triangular matrix L and a symmetric nonnegative matrix R with $l_{ij} = 0$ if $(i,j) \in Q$ and $r_{ij} = 0$ if $(i,j) \notin Q$, such that the splitting $A = LL^T - R$ leads to a convergent iterative process*

$$LL^T u^{i+1} = Ru^i + b \quad \textit{for each choice } u^0 \ ,$$

where $u^i \rightarrow u = A^{-1}b$.

<u>*Proof*</u> (see [101]; p.151.)

After the matrix L is constructed it is used in the PCG algorithm. Note that in this algorithm multiplications by L^{-1} and L^{-T} are necessary. This is never done by forming L^{-1} or L^{-T}. It is easy to see that L^{-1} is a full matrix. If for instance one wants to calculate $z = L^{-1}r$ we compute z by solving the linear system $Lz = r$. This is cheap since L is a lower triangular matrix so the forward substitution algorithm can be used.

Example 3.2.1 *We consider the model problem and compute a slightly adapted incomplete Cholesky decomposition: $A = LD^{-1}L^T - R$ where the elements of the lower triangular matrix L and diagonal matrix D satisfy the following rules:*

a) $l_{ij} = 0$ for all (i,j) where $a_{ij} = 0 \quad i > j$,

b) $l_{ii} = d_{ii}$,

c) $(LD^{-1}L^T)_{ij} = a_{ij}$ for all (i,j) where $a_{ij} \neq 0 \quad i \geq j$.

In this example $Q_0 = \{(i,j) | \ |i-j| \neq 0, 1, m\}$. If the elements of L are given as follows:

$$L = \begin{bmatrix} \tilde{d}_1 & & & & & \\ \tilde{b}_1 & \tilde{d}_2 & & & & \\ & \ddots & \ddots & \emptyset & & \\ \tilde{c}_1 & & \tilde{b}_m & \tilde{d}_{m+1} & & \\ & \ddots & \emptyset & \ddots & \ddots & \\ \emptyset & & & & & \end{bmatrix} \tag{3.2.4}$$

it is easy to see that (using the notation as given in (3.2.3))

$$\left.\begin{array}{rcl} \tilde{d}_i & = & a_i - \dfrac{b_{i-1}^2}{\tilde{d}_{i-1}} - \dfrac{c_{i-m}^2}{\tilde{d}_{i-m}} \\ \tilde{b}_i & = & b_i \\ \tilde{c}_i & = & c_i \end{array}\right\} \quad i = 1, ..., N \ . \tag{3.2.5}$$

where elements that are not defined are replaced by zeros. For this example the amount of work for $P^{-T}P^{-1}$ times a vector is comparable to the work to compute A times a vector. The combination of this incomplete Cholesky decomposition process with Conjugate Gradients is called the ICCG(0) method ([101]; p. 156). The 0 means that no extra diagonals are used for fill in. Note that this variant is very cheap with respect to memory: only one extra vector to store D is needed.

Another successfull variant is obtained by a smaller set Q. In this variant the matrix L has three more diagonals than the lower triangular part of the original matrix A. This preconditioning is obtained for the choice

$$Q^3 = \{(i,j) | \ |i-j| \neq 0, 1, 2, m-2, m-1, m\}$$

For the formula's to obtain the decomposition we refer to ([101]; p. 156). This preconditioner combined with PCG is known as the ICCG(3) method. A drawback is that all the elements of L are different from the corresponding elements of A so 6 extra vectors are needed to store L in memory.

To give an idea of the power of the ICCG methods we have copied some results from [101].

Example 3.2.2 *As a first example we consider the model problem, where the boundary conditions are somewhat different:*

$$\begin{aligned} &\frac{\partial u}{\partial x}(x,y) = 0 \quad \textit{for} \quad \left\{ \begin{array}{l} x = 0\,, \quad y \in [0,1] \\ x = 1\,, \quad y \in [0,1] \end{array} \right. \,, \\ &\frac{\partial u}{\partial y}(x,y) = 0 \quad \textit{for} \quad y = 1\,,\ x \in [0,1]\,, \\ &u(x,y) = 1 \qquad \textit{for} \quad y = 0\,,\ x \in [0,1]\,. \end{aligned}$$

The distribution of the grid points is equidistant with $h = \frac{1}{31}$. The results for CG, ICCG(0) and ICCG(3) are plotted in Figure 3.2.4.

From inequality (3.2.2) it follows that the rate of convergence can be bounded by

$$\overline{\rho} = \frac{\sqrt{\kappa_2(P^{-1}AP^{-T})} - 1}{\sqrt{\kappa_2(P^{-1}AP^{-T})} + 1}. \qquad (3.2.6)$$

To obtain a better insight in the fast convergence of ICCG(0) and ICCG(3) the eigenvalues of $A, (L_0L_0^T)^{-1}A$ and $(L_3L_3^T)^{-1}A$ are computed and given in Figure 3.2.5. For this result given in [101] a small matrix of order $n = 36$ is used, so all eigenvalues can be calculated.
The eigenvalues as given in Figure 3.2.5 can be substituted in formula (3.2.6). We then obtain

$$\begin{aligned} &\overline{\rho} = 0.84 \quad \textit{for} \quad \textit{CG}\,, \\ &\overline{\rho} = 0.53 \quad \textit{for} \quad \textit{ICCG(0)}\,, \\ &\overline{\rho} = 0.23 \quad \textit{for} \quad \textit{ICCG(3)}\,, \end{aligned} \qquad (3.2.7)$$

which explains the fast convergence of the PCG variants.

In our explanation of the convergence behavior we have also used the notion of Ritz values. Applying these ideas to the given methods we note the following:

- For CG the eigenvalues of A are more or less equidistantly distributed. So if a Ritzvalue has converged we only expect a small decrease in the rate of convergence. This agrees with the results given in Figure 3.2.4, the CG method has a linear convergent behavior.

- For the PCG method the eigenvalue distribution is completely different. Looking to the spectrum of $(L_3L_3^T)^{-1}A$ we see that $\lambda_{36} = 0.446$ is the

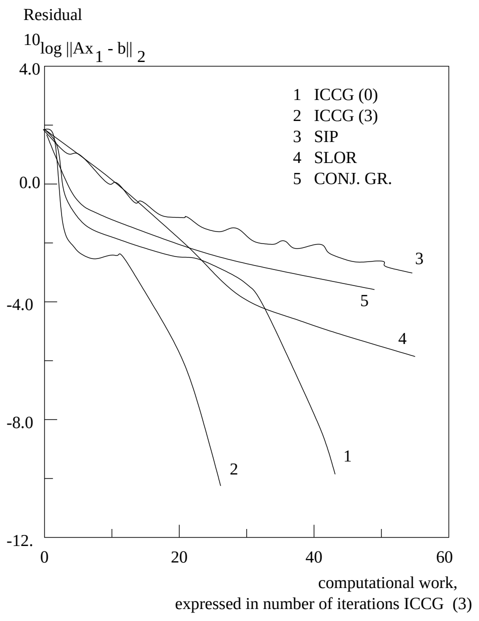


Figure 3.2.4: The results for the CG, ICCG(0) and ICCG(3) methods, compared with SIP (Strongly Implicit Procedure) and SLOR (Successive Line Over Relaxation method)

smallest eigenvalue. The distance between λ_{36} and the other eigenvalues is relatively large which implies that there is a fast convergence of the smallest Ritz-value to λ_{36}. Furthermore, if the smallest Ritzvalue is a reasonable approximation of λ_{36} the effective condition number is much less than the original condition number. Thus super linear convergence is expected. This again agrees very well with the results given in Figure 3.2.4.

So, the faster convergence of ICCG(3) comes from a smaller condition number and a more favorable distribution of the internal eigenvalues.

Finally, the influence of the order of the matrix on the number of iterations required to reach a certain precision was checked for both ICCG(0) and ICCG(3). Therefore several uniform rectangular meshes have been chosen, with mesh spacings varying from $\sim 1/10$ up to $\sim 1/50$. This resulted in linear systems with matrices of order 100 up to about 2500. In each case it was determined how many iterations were necessary, to reduce the ∞ norm of the residual vector below some fixed small number ε. In Figure 3.2.6 the number of iterations are plotted against the order of the matrices for $\varepsilon = 10^{-2}$, $\varepsilon = 10^{-6}$ and $\varepsilon = 10^{-10}$. It can be seen that the number of iterations, necessary to get the residual vector sufficiently small, increases only slowly for increasing order of the matrix. The dependence of $\kappa_2(A)$ for this problem is $O(\frac{1}{h^2})$. For ICCG preconditioning it can be shown that there is a cluster of large eigenvalues of $(L_0L_0^T)^{-1}A$ in the vicinity of 1, whereas the small eigenvalues are of order $O(h^2)$ and the gaps between them

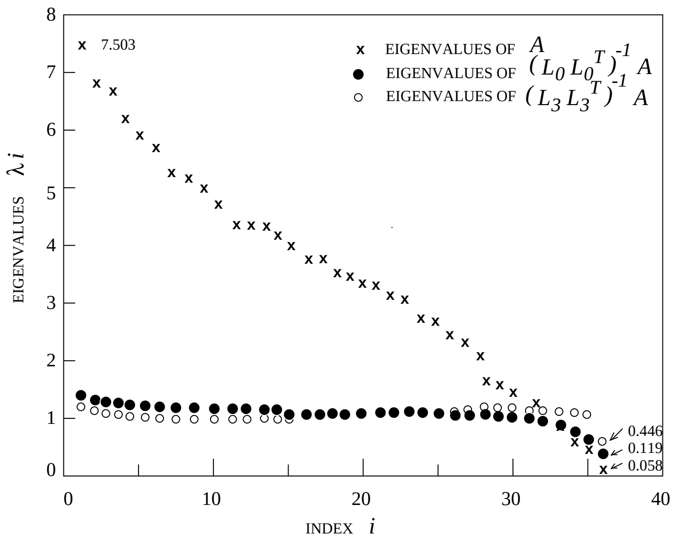


Figure 3.2.5: The eigenvalues of A, $(L_0L_0^T)^{-1}A$ and $(L_3L_3^T)^{-1}A$.

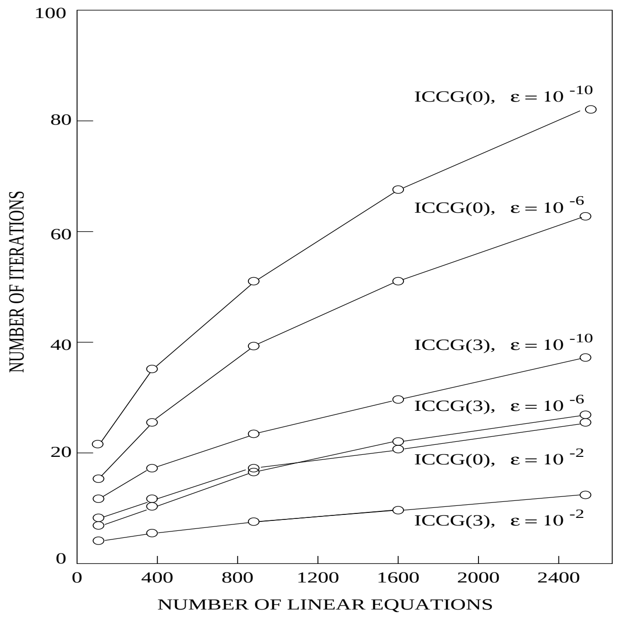


Figure 3.2.6: Effect of number of equations on the rate of convergence

are relatively large. So also for ICCG(0) the condition number is $O(\frac{1}{h^2})$. Faster convergence can be explained by the fact that the constant in front of $\frac{1}{h^2}$ is less for the ICCG(0) preconditioned system than for A and the distribution of the internal eigenvalues is much better. Therefore, super linear convergence sets in after a small number of iterations.

The success of the ICCG method has led to many variants. In the following we describe two of them MICCG(0) given in [64] (MICCG means Modified ICCG) and RICCG(0) given in [8] (RICCG means Relaxed ICCG).

MICCG In the MICCG method the MIC preconditioner is constructed by slightly adapted rules. Again A is split as follows $A = LD^{-1}L^T - R$, where L and D satisfy the following rules:

a) $l_{ij} = 0$ for all (i, j) where $a_{ij} = 0 \;\; i > j$,

b) $l_{ii} = d_{ii}$,

c) rowsum $(LD^{-1}L^T)$=rowsum(A) for all rows and $(LD^{-1}L^T)_{ij} = a_{ij}$ for all (i, j) where $a_{ij} \neq 0 \;\; i > j$.

Consequence of c):$LD^{-1}L^T \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = A \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$ so $(LD^{-1}L^T)^{-1}A \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$.

This means that if $Ax = b$ and x and/or b are slowly varying vectors this incomplete Cholesky decomposition is a very good approximation for the inverse of A with respect to x and/or b.

Using the same notation of L as given in (3.2.4) we obtain

$$\left.\begin{array}{rcl} \tilde{d}_i & = & a_i - (b_{i-1} + c_{i-1})\dfrac{b_{i-1}}{\tilde{d}_{i-1}} - (b_{i-m} + c_{i-m})\dfrac{c_{i-m}}{\tilde{d}_{i-m}} \\ \tilde{b}_i & = & b_i \\ \tilde{c}_i & = & c_i \end{array}\right\} \quad i = 1, .., N \qquad (3.2.8)$$

It can be proved that for this preconditioning there is a cluster of small eigenvalues in the vicinity of 1 and the largest eigenvalues are of order $\frac{1}{h}$ and have large gap ratio's. So the condition number is $O(1/h)$.

In many problems the initial iterations of MICCG(0) converge faster than ICCG(0). Thereafter for both methods super linear convergence sets in. Using MICCG the largest Ritz values are good approximations of the largest eigenvalues of the preconditioned matrix. A drawback of MICCG is that due to rounding errors components in eigenvectors related to large eigenvalues return after some iterations. This deteriorates the rate of convergence. So if many iterations are needed ICCG can be better than MICCG.

In order to combine the advantages of both methods the RIC preconditioner is proposed in [8], which is an average of the IC and MIC preconditioner. For the details we refer to [8]. Only the algorithm is given: choose the average parameter $\alpha \in [0, 1]$ then $\tilde{d}_i, \tilde{b}_i$ and $\tilde{c}_i$ are given by:

$$\left.\begin{array}{rcl} \tilde{d}_i & = & a_i - (b_{i-1} + \alpha c_{i-1})\frac{b_{i-1}}{\tilde{d}_{i-1}} - (\alpha b_{i-m} + c_{i-m})\frac{c_{i-m}}{\tilde{d}_{i-m}} \\ \tilde{b}_i & = & b_i \\ \tilde{c}_i & = & c_i \end{array}\right\} \quad i = 1, ..., N \qquad (3.2.9)$$

However, the question how to choose α remains? In Figure 3.2.7 which is copied from [138] a typical convergence behavior as function of α is given. This motivates the choice $\alpha = 0.95$, which leads to a very good rate of convergence on a wide range of problems.

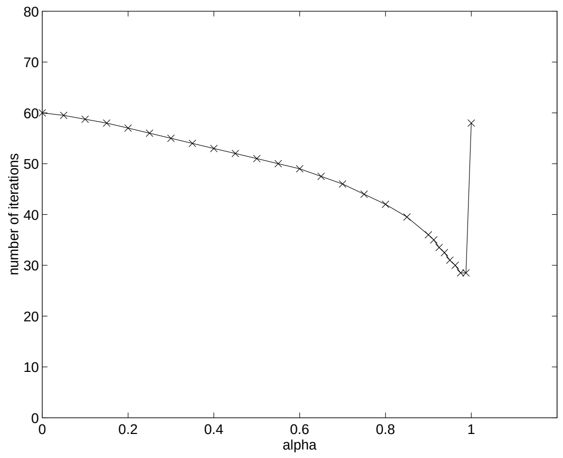


Figure 3.2.7: Convergence in relation to α

Diagonal scaling The above given preconditioners IC, MIC and RIC can be optimized with respect to work. One way is to look at the explicitly preconditioned system:

$$D^{1/2}L^{-1}AL^{-T}D^{1/2}y = D^{1/2}L^{-1}b \tag{3.2.10}$$

Applying CG to this system one has to solve lower triangular systems of equations with matrix $LD^{-1/2}$. The main diagonal of this matrix is equal to $D^{1/2}$. It saves time if we can change the system in such a way that the main diagonal is equal to the identity matrix. One idea is to replace (3.2.10) by

$$D^{1/2}L^{-1}D^{1/2}D^{-1/2}AD^{-1/2}D^{1/2}L^{-T}D^{1/2}y = D^{1/2}L^{-1}D^{1/2}D^{-1/2}b \, .$$

with $\tilde{A} = D^{-1/2}AD^{-1/2}$, $\tilde{L} = D^{-1/2}LD^{-1/2}$ and $\tilde{b} = D^{-1/2}b$ we obtain

$$\tilde{L}^{-1}\tilde{A}\tilde{L}^{-T}y = \tilde{L}\tilde{b} \, . \tag{3.2.11}$$

Note that $\tilde{L}_{ii} = 1$ for $i = 1, ..., N$. PCG now is the application of CG to this preconditioned system.

Eisenstat implementation In this section we restrict ourselves to the IC(0), MIC(0) and RIC(0) preconditioner. We have already noted that the amount of work of one PCG iteration is approximately 2 times as large as for a CG iteration. In [40] it is shown that much of the extra work can be avoided. If CG is applied to (3.2.11) products of the following form are calculated: $v^{j+1} = \tilde{L}^{-1}\tilde{A}\tilde{L}^{-T}v^j$. For the preconditioners used, the off-diagonal part of $\tilde{L}$ is equal to the off-diagonal part of $\tilde{A}$. Using this we obtain:

$$\begin{aligned} v^{j+1} &= \tilde{L}^{-1}\tilde{A}\tilde{L}^{-T}v^j = \tilde{L}^{-1}(\tilde{L} + \tilde{A} - \tilde{L} - \tilde{L}^T + \tilde{L}^T)\tilde{L}^{-T}v^j \\ &= \tilde{L}^{-T}v^j + \tilde{L}^{-1}(v^j + (diag\ (\tilde{A}) - 2I)\tilde{L}^{-T}v^j) \end{aligned} \tag{3.2.12}$$

So v^{j+1} can be calculated by a multiplication by $\tilde{L}^{-T}$ and $\tilde{L}^{-1}$ and some vector operations. The saving in CPU time is the time to calculate the product of A times a vector. Now one iteration of PCG costs approximately the same as one iteration of CG.

General stencils In practical problems the stencils in finite element methods may be larger or more irregularly distributed than in finite difference methods. The same type of preconditioners can be used. However, there are some differences. We restrict ourselves to the IC(0) preconditioner. For the five point stencil we see that the off-diagonal part of L is equal to the strictly lower triangular part of A. For general stencils this property does not hold. Drawbacks are: All the elements of L should be stored, so the memory requirements of PCG are two times as large as for CG. Furthermore, the Eisenstat implementation can no longer be used. This motivates another preconditioner constructed by the following rules:

ICD (Incomplete Cholesky restricted to the Diagonal).
A is again splitted as $A = LD^{-1}L^T - R$ and L and D satisfy the following rules:

a) $l_{ij} = 0$ for all (i, j) where $a_{ij} = 0 \quad i > j$

b) $l_{ii} = d_{ii}, \quad i = 1, ..., N$

c) $l_{ij} = a_{ij}$ for all (i, j) where $a_{ij} \neq 0 \ \ i > j$
$(LD^{-1}L^T)_{ii} = a_{ii} \quad i = 1, ..., N.$

This enables us to save memory (only the diagonal D should be stored) and CPU time (since now Eisenstat implementation can be used) per iteration. For large problems the rate of convergence of ICD is worse than for IC. Also MICD and RICD preconditioners can be given.

3.2.2 Exercises

a. Derive the preconditioned CG method using the CG method applied to $\tilde{A}\tilde{u} = \tilde{b}$.

b. (a) Show that the formula's given in (3.2.5) are correct.

   (b) Show that the formula's given in (3.2.8) are correct.

c. (a) Suppose that $a_i = 4$ and $b_i = -1$. Show that $\lim\limits_{i\to\infty} \tilde{d}_i = 2 + \sqrt{3}$, where $\tilde{d}_i$ is as defined in (3.2.5).

   (b) Do the same for $a_i = 4$, $b_i = -1$ and $c_i = -1$ with $m = 10$, and show that $\lim\limits_{i\to\infty} \tilde{d}_i = 2 + \sqrt{2}$.

   (c) Prove that the $LD^{-1}L^T$ decomposition (3.2.5) exists if $a_i = a, b_i = b, c_i = c$ and A is diagonally dominant.

d. **A practical exercise**
   Use as test matrices:

$$[a, f] = poisson(5, 5, 0, 0,' central')$$

   (a) Adapt the matlab cg algorithm such that preconditioning is included. Use a diagonal preconditioner and compare the number of iterations with cg without preconditioner.

   (b) Use the formula's given in (3.2.5) to obtain an incomplete $LD^{-1}L^T$ decomposition of A. Make a plot of the diagonal elements of D. Can you understand this plot?

   (c) Use the $LD^{-1}L^T$ preconditioner in the method given in (a) and compare the convergence behavior with that of the diagonal preconditioner.

3.3 Krylov Subspace Methods for General Matrices

In Chapter 3.1 we have discussed the Conjugate Gradient method. This Krylov subspace method can only be used if the coefficient matrix is symmetric and positive definite. In this chapter we discuss Krylov subspace methods for increasing classes of matrices. For these we give different iterative methods, and presently there is no method which is the best for all cases. This is in contrast with the symmetric positive definite case. In Subsection 3.3.6 we give some guidelines for choosing an appropriate method for a given system of equations. In Section 3.3.1 we consider symmetric indefinite systems. General real matrices are the subject of Section 3.3.2. We end this chapter with a section containing iterative methods for complex linear systems.

3.3.1 Indefinite Symmetric Matrices

In this section we relax the condition that A should be positive definite (Chapter 3.1), and only assume that A is symmetric. This means that $x^TAx > 0$ for some x and possibly $y^TAy < 0$ for other y. For the real eigenvalues this implies that A has positive and negative eigenvalues. For these types of matrices $\|.\|_A$ no longer define a norm. Furthermore CG may have a serious break down since $(p^k)^TAp^k$ may be zero whereas $\|p^k\|_2$ is not zero. In this section we give two different (but related) methods to overcome these difficulties. These methods are defined in [104].

SYMMLQ In the CG method we have defined the orthogonal matrix $R_k \in \mathbb{R}^{N\times k}$ where the j^{th} column of R_k is equal to the normalized residual $r^j/\|r^j\|_2$. It appears that the following relation holds

$$AR_k = R_{k+1}\bar{T}_k, \tag{3.3.1}$$

where $\bar{T}_k \in \mathbb{R}^{k+1\times k}$ is a tridiagonal matrix. This decomposition is also known as the *Lanczos algorithm* for the tridiagonalisation of A ([57]; p.477). This decomposition is always possible for symmetric matrices, also for indefinite ones. The CG iterates are computed as follows:

$$\begin{aligned} u^k &= R_k y^k, \quad \text{where} & (3.3.2)\\ R_k^TAR_ky^k &= T_ky^k = R_k^Tb. & (3.3.3)\end{aligned}$$

Note that T_k consists of the first k rows of $\bar{T}_k$. If A is positive definite then T_k is positive definite. It appears further that in the CG process an LDL^T factorization of T_k is used to solve (3.3.3). This can lead to break down because T_k may be indefinite in this section (compare [57]; Section 9.3.1, 9.3.2, 10.2.6). In the SYMMLQ method [104] problem (3.3.3) is solved in a stable way by using an LQ decomposition. So T_k is factorized in the following way:

$$T_k = \bar{L}_kQ_k \quad \text{where} \quad Q_k^TQ_k = I \tag{3.3.4}$$

with $\bar{L}_k$ lower triangular. For more details to obtain an efficient code we refer to [104] section 5.

MINRES In SYMMLQ we have solved (3.3.3) in a stable way and obtained the "CG" iteration. However, since $\|.\|_A$ is no longer a correct norm, the optimality properties of CG are lost. To get them back one can use the decomposition

$$AR_k = R_{k+1}\bar{T}_k$$

and minimize the error in the $\|.\|_{A^TA}$ norm. This is a proper norm if A is nonsingular, and leads to the following approximation:

$$u^k = R_k y^k, \tag{3.3.5}$$

where y_k is such that

$$\|Au^k - b\|_2 = \min_{y\in\mathbb{R}^k} \|AR_k y - b\|_2. \tag{3.3.6}$$

Note that $\|AR_k y - b\|_2 = \|R_{k+1}\bar{T}_k y - b\|_2$ using (3.3.1).

Starting with $u^0 = 0$ implies that $r^0 = b$. Since R_{k+1} is an orthogonal matrix and $R^T_{k+1}b = \|r^0\|_2 e_1$, where $e_1 = (1, 0, ldots, 0)^T$, we have to solve the following least squares problem

$$\min_{y\in\mathbb{R}^k} \|\bar{T}_k y - \|r^0\|_2 e_1\|_2 \ . \tag{3.3.7}$$

Again for more details see [104]; Section 6.7. A comparison of both methods is given in [104]. In general the rate of convergence of SYMMLQ or MINRES for indefinite symmetric systems of equations is much worse than of CG for definite systems. Preconditioning techniques for these methods are specified in [120].

3.3.2 Iterative Methods for General Matrices

In this section we consider iterative methods to solve $Au = b$ where the only requirement is that $A \in \mathbb{R}^{N\times N}$ is nonsingular. In the symmetric case we have seen that CG has the following two nice properties:

- optimality, the error is minimal measured in a certain norm,
- short recurrences, only the results of one foregoing step are necessary; work and memory do not increase for an increasing number of iterations.

It has been shown in [42] that it is impossible to obtain a Krylov method based on $K^i(A; r_0)$, which satisfies both properties for general matrices. So either the method has an optimality property but long recurrences, or no optimality and short recurrences. Recently, some surveys on general iteration methods have been published: [27], [57] Section 10.4, [48], [122], [60], [13].

It appears that there are essentially three different ways to solve non-symmetric linear systems, while maintaining some kind of orthogonality between the residuals:

a. Solve the normal equations $A^T Au = A^T b$ with Conjugate Gradients.

b. Construct a basis for the Krylov subspace by a 3-term bi-orthogonality relation.

c. Make all the residuals explicitly orthogonal in order to have an orthogonal basis for the Krylov subspace.

These classes form the subject of the following subsections. An introduction and comparison of these classes is given in [102].

3.3.3 CG Applied to the Normal Equations

The first idea to apply CG to the normal equations

$$A^T Au = A^T b, \tag{3.3.8}$$

or

$$AA^T y = b \quad \text{with} \quad u = A^T y \tag{3.3.9}$$

is obvious. When A is nonsingular $A^T A$ is symmetric and positive definite. So all the properties and theoretical results for CG can be used. There are, however, some drawbacks. First of all, the rate of convergence now depends on $\kappa_2(A^T A) = \kappa_2(A)^2$. In many applications $\kappa_2(A)^2$ is extremely large. Hence the convergence of CG applied to (3.3.8) is very slow. Another difference is that the convergence of CG applied to $Au = b$ depends on the eigenvalues of A whereas the convergence of CG applied to (3.3.8) depends on the eigenvalues of $A^T A$, which are equal to the singular values of A squared.

Per iteration a multiplication with A and A^T are necessary, so the amount of work is approximately two times as much as for the CG method. Furthermore, in several (FEM) applications Av is easily obtained but $A^T v$ is not because of an unstructured grid and the corresponding data structure. Finally, not only the convergence depends on $\kappa_2(A)^2$ but also the error due to rounding errors. To improve the numerical stability it is suggested in [15] to replace inner products like $p^T A^T Ap$ by $(Ap)^T Ap$. Another improvement is the method LSQR proposed by [105]. This method is based on the application of the Lanczos method to the auxiliary system

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} r \\ u \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} .$$

This is a very reliable algorithm. It uses reliable stopping criteria and estimates of standard errors for x and the condition number of A.

3.3.4 BiCG Type Methods

In this type of method we have short recurrences but we do not have the optimality property. We have seen that CG is based on the Lanczos algorithm. The Lanczos algorithm for non-symmetric matrices is called the Bi-Lanczos algorithm. BiCG type methods are based on Bi-Lanczos. In the Lanczos method we look for a matrix Q such that $Q^TQ = I$ and

$$Q^TAQ = T \quad \text{tridiagonal} .$$

In the Bi-Lanczos algorithm we construct a similarity transformation X such that

$$X^{-1}AX = T \quad \text{tridiagonal} .$$

To obtain this matrix we construct a basis $r^0, ..., r^{i-1}$, which are the residuals, for $K^i(A; r^0)$ such that $r^j \perp K^j(A^T; s^0)$ and $s^0, ..., s^{i-1}$ form a basis for $K^i(A^T; s^0)$ such that $s_j \perp K^j(A; r^0)$, so the sequences $\{r_i\}$ and $\{s_i\}$ are bi-orthogonal. Using these properties and the definitions $R_k = [r^0 ... r^{k-1}]$, $S_k = [s^0 ... s^{k-1}]$ the following relations can be proved [139]:

$$AR_k = R_kT_k + \alpha_k r^k e_k^T \ , \tag{3.3.10}$$

and

$$S_k^T(Au^k - b) = 0 \ .$$

Using (3.3.10), $r^0 = b$ and $u^k = R_k y$ we obtain

$$S_k^T R_k T_k y = s_0 r_0^T e_1. \tag{3.3.11}$$

Since $S_k^T R_k$ is a diagonal matrix with diagonal elements $(r^j)^T s^j$, we find if all these diagonal elements are nonzero,

$$T_k y = e_1 \ , \quad u^k = R_k y \ .$$

This algorithm fails when a diagonal element of $S_k^T R_k$ becomes (nearly) zero, because these elements are used to normalize the vectors s_j (compare [57] §9.3.6). This is called a serious (near) break down. The way to get around this difficulty is the so-called look-ahead strategy. For details on look-ahead we refer to [106], and [49]. Another way to avoid break down is to restart as soon as a diagonal element gets small. This strategy is very simple, but one should realize that at a restart the Krylov subspace that has been built up, is thrown away, which destroys possibilities for faster (superlinear) convergence.

(The description of the methods given below is based on [139].)

BiCG As has been shown for Conjugate Gradients, the LU decomposition of the tridiagonal system T_k can be updated from iteration to iteration. It leads to a recursive update of the solution vector, which avoids saving all intermediate r and s vectors. This variant of Bi-Lanczos is usually called Bi-Conjugate Gradients, or shortly Bi-CG [45]. Of course, one can in general not be sure that an LU decomposition (without pivoting) of the tridiagonal matrix T_k exists, and if it does not exist a serious break-down of the Bi-CG algorithm will occur. This break-down can be avoided in the Bi-Lanczos formulation of this iterative solution scheme. For the algorithm we refer to [13] page 22.

Note that for symmetric matrices Bi-Lanczos generates the same solution as Lanczos, provided that $s^0 = r^0$, and under the same condition, Bi-CG delivers the same iterates as CG, for positive definite matrices. However, the Bi-orthogonal variants do so at the cost of two matrix vector operations per iteration step. The method has been superseded by CGS and Bi-CGSTAB.

QMR The QMR method [50] is related to Bi-CG in a similar way as MINRES is to CG. For stability reasons the basis vectors r^j and s^j are normalized (as is usual in the underlying Bi-Lanczos algorithm).

If we group the residual vectors r^j, for $j = 0, ..., i-1$ in a matrix R_i, we can write the recurrence relations as

$$AR_i = R_{i+1}\bar{T}_i \ ,$$

with

$$\bar{T}_i = \begin{array}{cc} \longleftarrow \quad i \quad \longrightarrow & \\ \left(\begin{array}{cccc} \ddots & \ddots & & \\ \ddots & \ddots & \ddots & \\ & \ddots & \ddots & \ddots \\ & & \ddots & \ddots \\ & & & \ddots \end{array}\right) & \begin{array}{c}\uparrow\\ \\ i+1\\ \\ \downarrow\end{array} \end{array} .$$

Similar as for MINRES we would like to construct the u^i, with

$$u^i \in span\ \{r^0, Ar^0, ..., A^{i-1}r^0\}\ , \quad u^i = R_i\bar{y},$$

for which

$$\begin{aligned} \|Au^i - b\|_2 &= \|AR_i\bar{y} - b\|_2 \\ &= \|R_{i+1}\bar{T}_i y - b\|_2 \\ &= \|R_{i+1}D_{i+1}^{-1}\{D_{i+1}\bar{T}_i y - \|r^0\|_2 e_1\}\|_2 \end{aligned}$$

is minimal. However, this would be quite an amount of work since the columns of R_{i+1} are not necessarily orthogonal. In [50] it is suggested to solve the minimum norm least squares problem

$$\min_{y \in I\!\!R^i} \|D_{i+1}\bar{T}_i y - \|r^0\|_2 e_1\|_2 \text{ with } e_1 = (1, 0, \ldots, 0)^T \ . \tag{3.3.12}$$

It leads to the simplest version of the QMR method. A more general form arises if the least squares problem (3.3.12) is replaced by a weighted least squares problem. No strategies are yet known for optimal weights, however. In [50] the QMR method is carried out on top of a look-ahead variant of the bi-orthogonal Lanczos method, which makes the method more robust. Experiments suggest that QMR has a much smoother convergence behavior than Bi-CG, but it is not essentially faster than Bi-CG. For the algorithm we refer to [13] page 24.

CGS The bi-conjugate gradient residual vectors can be written as $r^j = P_j(A)r^0$ and $\hat{r}^j = P_j(A^T)\hat{r}^0$. Because of the bi-orthogonality relation we have that

$$(r^j, \hat{r}^i) = (P_j(A)r^0, P_i(A^T)\hat{r}^0) = (P_i(A)P_j(A)r^0, \hat{r}^0) = 0 \ , \text{ for } i < j.$$

The iteration parameters for bi-conjugate gradients are computed from innerproducts like above. Sonneveld observed in [131] that we can also construct the vectors $r^j = P_j^2(A)r^0$, using only the latter form of the innerproduct for recovering the bi-conjugate gradients parameters (that implicitly define the polynomial P_j). By doing so, neither the vectors $\hat{r}^j$ have to be formed, nor is it necessary to multiply by the matrix A^T. The resulting CGS [131] method works well in general for many non-symmetric linear problems. It often converges much faster than Bi-CG (about twice as fast in some cases). However, CGS usually shows a very irregular convergence behavior. This behavior can even lead to cancellation and a spoiled solution [138].

The following scheme carries out the CGS process for the solution of $Au = b$, with a given preconditioner M (to be discussed next):

```
Conjugate Gradient Squared method
    u$^0$ is an initial guess; $r^0 = b - Au^0$;
    $\tilde{r}^0$ is an arbitrary vector, such that
    $(r^0, \tilde{r}^0) \neq 0$ ,
    e.g., $\tilde{r}^0 = r^0$ ; $\rho_0 = (r^0, \tilde{r}^0)$ ;
    $\beta_{-1} = \rho_0$ ; $p_{-1} = q_0 = 0$ ;
    for $i = 0, 1, 2, ...$ do
        $w^i = r^i + \beta_{i-1} q^i$ ;
        $p^i = w^i + \beta_{i-1}(q^i + \beta_{i-1} p^{i-1})$ ;
        $\hat{p} = M^{-1} p^i$ ;
        $\hat{v} = A\hat{p}$ ;
        $\alpha_i = \frac{\rho_i}{(\tilde{r}^0, \hat{v})}$ ;
        $q^{i+1} = w^i - \alpha_i \hat{v}$ ;
        $\hat{w} = M^{-1}(w^i + q^{i+1})$
        $u^{i+1} = u^i + \alpha_i \hat{w}$ ;
        if $u^{i+1}$ is accurate enough then quit;
        $r^{i+1} = r^i - \alpha_i A\hat{w}$ ;
        $\rho_{i+1} = (\tilde{r}^0, r^{i+1})$ ;
        if $\rho_{i+1} = 0$ then method fails to converge!;
        $\beta_i = \frac{\rho_{i+1}}{\rho_i}$ ;
    end for
```

In exact arithmetic, the α_j and β_j are the same as those generated by BiCG. Therefore, they can be used to compute the Petrov-Galerkin approximations for eigenvalues of A.

Bi-CGSTAB Bi-CGSTAB [139] is based on the following observation. Instead of squaring the Bi-CG polynomial, we can construct another iteration method, by which iterates u^i are generated so that $r^i = \tilde{P}_i(A)P_i(A)r^0$ with another i^{th} degree polynomial $\tilde{P}_i$. An obvious possibility is to take for $\tilde{P}_j$ a polynomial of the form

$$Q_i(x) = (1 - \omega_1 x)(1 - \omega_2 x)...(1 - \omega_i x) ,$$

and to select suitable constants $\omega_j \in I\!R$. This expression leads to an almost trivial recurrence relation for the Q_i. In Bi-CGSTAB, ω_j in the j^{th} iteration step is chosen as to minimize r^j, with respect to ω_j, for residuals that can be written as $r^j = Q_j(A)P_j(A)r^0$.
The preconditioned Bi-CGSTAB algorithm for solving the linear system $Au = b$, with preconditioning M reads as follows:

```
Bi-CGSTAB method
    $u^0$ is an initial guess; $r^0 = b - Au^0$;
    $\bar{r}^0$ is an arbitrary vector, such that $(\bar{r}^0, r^0) \neq 0$, e.g., $\bar{r}^0 = r^0$ ;
    $\rho_{-1} = \alpha_{-1} = \omega_{-1} = 1$ ;
    $v^{-1} = p^{-1} = 0$ ;
    for $i = 0, 1, 2, ...$ do
        $\rho_i = (\bar{r}^0, r^i)$ ; $\beta_{i-1} = (\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$ ;
        $p^i = r^i + \beta_{i-1}(p^{i-1} - \omega_{i-1}v^{i-1})$ ;
        $\hat{p} = M^{-1}p^i$ ;
        $v^i = A\hat{p}$ ;
        $\alpha_i = \rho_i/(\bar{r}^0, v^i)$ ;
        $s = r^i - \alpha_i v^i$ ;
        if $\|s\|$ small enough then
            $u^{i+1} = u^i + \alpha_i \hat{p}$ ; quit;
        $z = M^{-1}s$ ;
        $t = Az$ ;
        $\omega_i = (t, s)/(t, t)$ ;
        $u^{i+1} = u^i + \alpha_i \hat{p} + \omega_i z$ ;
        if $u^{i+1}$ is accurate enough then quit;
        $r^{i+1} = s - \omega_i t$ ;
    end for
```

The matrix M in this scheme represents the preconditioning matrix and the way of preconditioning [139]. The above scheme in fact carries out the Bi-CGSTAB procedure for the explicitly postconditioned linear system

$$AM^{-1}y = b ,$$

but the vectors y_i and the residual have been transformed back to the vectors u^i and r^i corresponding to the original system $Au = b$. Compared to CGS two extra innerproducts need to be calculated.

In exact arithmetic, the α_j and β_j have the same values as those generated by Bi-CG and CGS. Hence, they can be used to extract eigenvalue approximations for the eigenvalues of A (see Bi-CG).

An advantage of these methods is that they use *short recurrences*. A disadvantage is that there is only a *semi-optimality property*. As a result of this, more matrix vector products are needed and no convergence properties have been proved. In experiments we see that the convergence behavior looks like CG for a large class of problems. However, the influence of rounding errors is much more significant than for CG. Small changes in the algorithm can lead to instabilities. Finally, it is always necessary to compare the norm of the updated residual to the exact residual $\|b - Au^k\|_2$. If "near" break down had occurred in the algorithm these quantities may differ by several orders of magnitude. In such a case the method should be restarted.

3.3.5 GMRES Type Methods

These methods are based on *long recurrences*, but have certain *optimality properties*. The long recurrences imply that the amount of work per iteration and the required memory grow for an increasing number of iterations. Consequently, in practice one cannot afford to run the full algorithm, and it becomes necessary to use *restarts* or to *truncate* vector recursions. In this section we describe GMRES, GCR and a combination of both GMRESR.

GMRES In this method, Arnoldi's method is used for computing an orthonormal basis $\{v^1, ..., v^k\}$ of the Krylov subspace $K^k(A; r^0)$. The modified Gram-Schmidt version of Arnoldi's method can be described as follows [123]:

```
a. Start: choose $u^0$ and compute $r^0 = b - Au^0$ and $v^1 = r^0/\|r^0\|_2$,

b. Iterate: for $j = 1, ..., k$ do:
                $v^{j+1} = Av^j$
                for $i = 1, .., j$ do:
                    $h_{ij} := (v^{j+1})^T v^i$ , $v^{j+1} := v^{j+1} - h_{ij} v^i$ ,
                end for
                $h_{j+1,j} := \|v^{j+1}\|_2$ , $v^{j+1} := v_{j+1}/h_{j+1,j}$
            end for
   The entries of upper $k + 1 \times k$ Hessenberg matrix $\bar{H}_k$ are the scalars $h_{ij}$.
```

In GMRES (General Minimal RESidual method) the approximate solution $u^k = u^0 + z^k$ with $z^k \in K^k(A; r^0)$ is such that

$$\|r^k\|_2 = \|b - Au^k\|_2 = \min_{z \in K^k(A; r^0)} \|r^0 - Az\|_2. \tag{3.3.13}$$

As a consequence of (3.3.13) it appears that r^k is orthogonal to $AK^k(A; r^0)$, so $r^k \perp K^k(A; Ar^0)$. If A is symmetric the GMRES method is equivalent to the MINRES method (described in [104]). For the matrix $\bar{H}_k$ it follows that $AV_k = V_{k+1}\bar{H}_k$ where the $N \times k$ matrix V_k is defined by $V_k = [v^1, ..., v^k]$. With this equation it is shown in [123] that $u^k = u^0 + V_k y^k$ where y^k is the solution of the following least squares problem:

$$\|\beta e_1 - \bar{H}_k y^k\|_2 = \min_{y \in \mathbb{R}^k} \|\beta e_1 - \bar{H}_k y\|_2, \tag{3.3.14}$$

with $\beta = \|r^0\|_2$ and e_1 is the first unit vector in $\mathbb{R}^{k+1}$. GMRES is a stable method and break down does not occur. If $h_{j+1,j} = 0$ then $u^j = u$ so this is a "lucky" break down (see [123]; Section 3.4).

Due to the optimality (see inequality (3.3.13)) convergence proofs exist [123]. If the eigenvalues of A are real the same bounds on the norm of the residual can be proved as for the CG method. For a more general eigenvalue distribution we shall give one result in the following theorem. Let P_m be the space of all polynomials of degree less than m and let σ represent the spectrum of A.

Theorem 3.5 *Suppose that A is diagonalizable so that $A = XDX^{-1}$ and let*

$$\varepsilon^{(m)} = \min_{\substack{p \in P_m \\ p(0)=1}} \max_{\lambda_i \in \sigma} |p(\lambda_i)|$$

Then the residual norm of the m-th iterate satisfies:

$$\|r^m\|_2 \leq K(X)\varepsilon^{(m)}\|r^0\|_2 \tag{3.3.15}$$

where $K(X) = \|X\|_2\|X^{-1}\|_2$. If furthermore all eigenvalues are enclosed in a circle centered at $C \in I\!R$ with $C > 0$ and having radius R with $C > R$, then

$$\varepsilon^{(m)} \leq \left(\frac{R}{C}\right)^m . \tag{3.3.16}$$

Proof: see [123]; p. 866.

For GMRES we see in many cases a superlinear convergence behavior comparable to CG. Recently, the same type of results have been proved for GMRES [140]. As we have already noted in the beginning, work per iteration and memory requirements increase for an increasing number of iterations. In this algorithm the Arnoldi process requires k vectors in memory in the k-th iteration. Furthermore $2k^2 \cdot N$ flops are needed for the total Gram Schmidt process. To restrict work and memory requirements one stops GMRES after m iterations, forms the approximate solution and uses this as a starting vector for a following application of GMRES. This is denoted by the GMRES(m) procedure (not restarted GMRES is denoted by *full* GMRES). However, restarting destroys many of the nice properties of full GMRES, for instance the optimality property is only valid inside a GMRES(m) step and the superlinear convergence behavior is lost. This is a severe drawback of the GMRES(m) method.

GCR Slightly earlier than GMRES, the GCR method was proposed in [41] (Generalized Conjugate Residual method). The algorithm is given as follows:

```
GCR algorithm
Choose $u^0$, compute $r^0 = b - Au^0$

    for $i = 1, 2, ...$ do
        $s^i = r^{i-1}$ ,
        $v^i = As^i$ ,
        for $j = 1, ..., i-1$ do
            $\alpha = (v^i, v^j)$ ,
            $v^i := v^i - \alpha v^j$ ,  $s^i := s^i - \alpha s^j$ ,
        end for
        $v^i := v^i / \|v^i\|_2$ ,  $s^i := s^i / \|v^i\|_2$
        $u^i := u^{i-1} + (r^{i-1}, v^i) s^i$ ;
        $r^i := r^{i-1} - (r^{i-1}, v^i) v^i$ ;
    end for
```

The storage of s^i and v^i costs two times as much memory as for GMRES. The rate of convergence of GCR and GMRES is comparable. However, there are examples where GCR may break down. So, when comparing full GMRES and full GCR the first one is to be preferred in many applications.
When the required memory is not available GCR can be restarted. However, another strategy is possible which is known as *truncation*. An example of this is to replace the j-loop by

```
    for $j = i - m, ..., i$ do
```

Now $2m$ vectors are needed in memory. Other truncation variants to discard search directions are possible. In general, we see that truncated methods have a better convergence behavior, especially if super linear convergence plays an important role. If restarting or truncation is necessary truncated GCR is in general better than restarted GMRES. For convergence results and other properties we refer to [41].

GMRESR Recently, methods have been proposed to diminish the disadvantages of restarting and truncation. One of these methods is GMRESR proposed in [141] and further investigated in [143]. This method consists of an outer- and an inner loop. In the inner loop we approximate the solution of a linear system by GMRES to find a good search direction. Thereafter in the outer loop the minimal residual approximation using these search directions is calculated by a GCR approach.

```
GMRESR algorithm
Choose $u^0$ and $m$, compute $r^0 = b - Au^0$

    for $i = 1, 2, ...$ do
        $s^i = P_{m,i-1}(A)r^{i-1}$ ,
        $v^i = As^i$ ,
        for $j = 1, ..., i-1$ do
            $\alpha = (v^i, v^j)$ ,
            $v^i := v^i - \alpha v^j$ ,  $s^i := s^i - \alpha s^j$ ,
        end for
        $v^i := v^i/\|v^i\|_2$ ,  $s^i := s^i/\|v^i\|_2$
        $u^i := u^{i-1} + (r^{i-1}, v^i)s^i$ ;
        $r^i := r^{i-1} - (r^{i-1}, v^i)v^i$ ;
    end for
```

The notation $s^i = P_{m,i-1}(A)r^{i-1}$ indicates that one applies one iteration of GMRES(m) to the system $As = r^{i-1}$. The result of this operation is s^i. For $m = 0$ we have just GCR, whereas for $m \to \infty$ one outer iteration is sufficient and we find GMRES. For the amount of work we refer to [141], where optimal choices of m are also given. In many problems the rate of convergence of GMRESR is comparable to full GMRES, whereas the amount of work and memory is much less. In the following picture we visualize the strong point of GMRESR in comparison with GMRES(m) and truncated GCR. A search direction is indicated by v^i. We

$v^1v^2v^3$ restart $v^1v^2v^3$ restart   GMRES(3)

$v^1v^2v^3 \lfloor \underset{\longrightarrow}{\underline{v^4v^5v^6}} \rfloor$ ...   GCR truncated with 3 vectors

$\hat{v}^1\hat{v}^2\hat{v}^3$ $\downarrow$ condense v^1 $\hat{v}^1\hat{v}^2\hat{v}^3$ $\downarrow$ condense v^2 ...   GMRESR with GMRES(3) as innerloop.

Figure 3.3.8: The use of search directions for restarted GMRES, truncated GCR and full GMRESR.

see for GMRES(3) that after 3 iterations all information is thrown away. For GCR(3) a window of the last 3 vectors moves from left to right. For GMRESR the information after 3 inner iterations is condensed in to one search direction so information does not get lost.
Also for GMRESR restart and truncation is possible [143]. In the inner loop other iterative methods can be used. Several of these choices lead to good iterative methods. In theory, we can call the same loop again, which motivates

the name GMRES Recursive. A comparable approach is the FGMRES method given in [121]. Herein the outer loop consists of a slightly adapted GMRES algorithm. FGMRES and GMRESR are comparable in work and memory. However, FGMRES can not be truncated. Therefore, we prefer the GMRESR method.

3.3.6 Choice of Iterative Method

For non-symmetric matrices it is difficult to decide which iterative method should be used. All the methods treated here have their own type of problems for which they are winners. Furthermore the choice depends on the computer used and the availability of memory. In general CGS and Bi-CGSTAB are easy to implement and reasonably fast for a large class of problems. If break down or bad convergence occurs, GMRES like methods may be preferred. Finally LSQR always converges but can take a large number of iterations.

In [143] we specified some easy to obtain parameters to facilitate a choice. Firstly, one should have an (crude) idea of the total number of iterations (mg) using full GMRES. Secondly, one should measure the ratio f

$$f = \frac{\text{the CPU time used for one preconditioned matrix vector product}}{\text{the CPU time used for a vector update}}$$

Note that f also depends on the hardware used. Under certain assumptions, given in [143], we obtain Figure 3.3.9. This figure gives only qualitative information. It illustrates the dependence of the choice on f and mg. If mg is large and f is small, Bi-CGSTAB is the best method. For large values of f and small values of mg the GMRES method is optimal and for intermediate values GMRESR is the best method. In [13] a flowchart is given with suggestions for the selection of a suitable iterative method.

3.3.7 Preconditioning for General Matrices

The preconditioning for non-symmetric matrices goes along the same lines as for symmetric matrices. There is a large amount of literature for generalization of the incomplete Cholesky decompositions. In general, it is more difficult to prove that the decomposition does not break down or that the resulting preconditioned system has a spectrum which leads to fast convergence. Since symmetry is no longer an issue the number of possible preconditioners is larger. Furthermore, if we have an incomplete LU decomposition of A, we can apply the iterative methods from 3.3.5 to the following three equivalent systems of equations:

$$U^{-1}L^{-1}Au = U^{-1}L^{-1}b \; , \tag{3.3.17}$$

$$L^{-1}AU^{-1}y = L^{-1}b \; , \quad u = U^{-1}y \; , \tag{3.3.18}$$

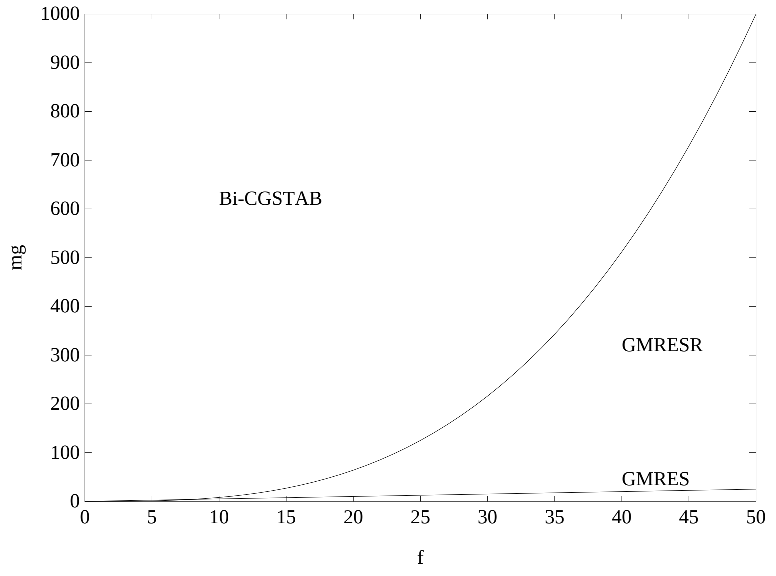


Figure 3.3.9: Regions of feasibility of Bi-CGSTAB, GMRES, and GMRESR.

or

$$AU^{-1}L^{-1}y = b\ , \quad u = U^{-1}L^{-1}y\ . \tag{3.3.19}$$

The rate of convergence is approximately the same for all variants. When the Eisenstat implementation is applied one should use (3.3.18). Otherwise, we prefer (3.3.19) because in that case the stopping criterion is based on $\|r\|_2 = \|b - Au^k\|_2$ whereas for (3.3.17) it is based on $\|U^{-1}L^{-1}r^k\|_2$, and for (3.3.18) it is based on $\|L^{-1}r^k\|_2$.

3.3.8 Exercises

a. Show that the solution $\begin{pmatrix} y \\ u \end{pmatrix}$ of the augmented system

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} y \\ u \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}$$

is such that u satisfies $A^T Au = A^T b$.

b. Take the following matrix

$$\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}.$$

(a) Suppose that GCR is applied to the system $Au = b$. Show that GCR converges in 1 iteration if $u - u^0 = cr^0$, where $c \neq 0$ is a scalar and $r^0 = b - Au^0$.

(b) Apply GCR for the choices $b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $u^0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

(c) Do the same for $u^0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

c. In the GCR algorithm the vector r^i is obtained from vector updates. Show that the relation $r^i = b - Au^i$ is valid.

d. Prove the following properties for the GMRES method:

- $AV_k = V_{k+1}\hat{H}_k$,
- $u^k = u^0 + V_k y^k$, where y^k is obtained from (3.3.14).

e. Figure 3.3.9 can give an indication which solution method should be used. Give advice in the following situations:

- Without preconditioning Bi-CGSTAB is the best method. What happens if preconditioning is added?
- We use GMRESR for a stationary problem. Switching to an instationary problem, what are good methods?
- We use GMRES. After optimizing the matrix vector product, which method is optimal?

f. **A practical exercise**
For the methods mentioned below we use as test matrices:

$$[a, f] = poisson(5, 5, 100, 0, 'central')$$

and

$$[a, f] = poisson(5, 5, 100, 0,' upwind')$$

(a) Adapt the matlab cg algorithm such that it solves the normal equations. Apply the algorithm to both matrices.

(b) Implement Bi-CGSTAB from the lecture notes. Take $K = I$ (identity matrix). Apply the algorithm to both matrices.

(c) Implement the GCR algorithm from the lecture notes. Apply the algorithm to both matrices.

(d) Compare the convergence behavior of the three methods.

4 The Multigrid Method

4.1 A First Glance at Multigrid

4.1.1 The Two Ingredients of Multigrid

In this section, we introduce the multigrid idea heuristically for Poisson's equation.

Therefore, we reconsider the *lexicographical* Gauss–Seidel method (GS-LEX) for Poisson's equation:

$$\begin{aligned} u_h^{m+1}(x_i, y_j) &= \frac{1}{4}\Big[h^2 b_h(x_i, y_j) + u_h^{m+1}(x_i - h, y_j) + u_h^m(x_i + h, y_j) \\ &\quad + u_h^{m+1}(x_i, y_j - h) + u_h^m(x_i, y_j + h)\Big] \ , \end{aligned} \tag{4.1.1}$$

with $(x_i, y_j) \in \Omega_h$ (u_h^m and u_h^{m+1} are the approximations of $u_h(x_i, y_j)$ before and after an iteration).

If we apply (4.1.1) to Poisson's equation, we recognize the following phenomenon: After a few iterations, the *error* of the approximation becomes *smooth* – it doesn't necessarily become small, but smooth: See Figure 4.1.1 for an illustration of this error smoothing effect. Looking at the error

$$e_h^m(x_i, y_j) = u_h(x_i, y_j) - u_h^m(x_i, y_j) \ ,$$

Formula (4.1.1) means

$$\begin{aligned} e_h^{m+1}(x_i, y_j) &= \frac{1}{4}\Big[e_h^{m+1}(x_i - h, y_j) + e_h^m(x_i + h, y_j) \\ &\quad + e_h^{m+1}(x_i, y_j - h) + e_h^m(x_i, y_j + h)\Big] \ . \end{aligned} \tag{4.1.2}$$

Obviously, the iteration formula can be interpreted as an error averaging process.

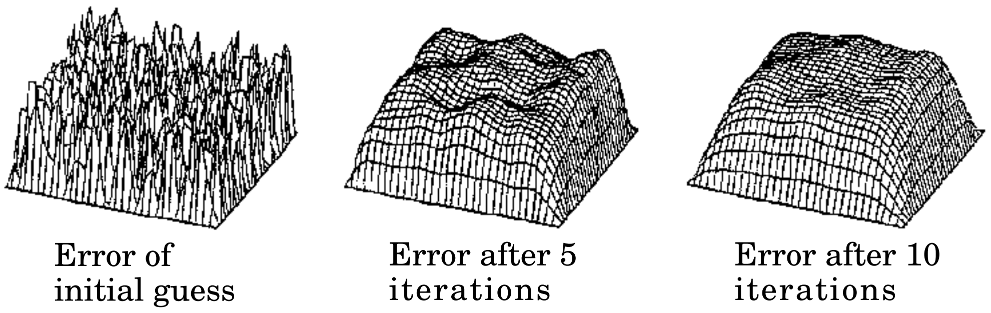


Figure 4.1.1: Influence of Gauss–Seidel iteration (4.1.1) on the error (Model Problem 1)

Error-smoothing is one of the two basic principles of the multigrid approach. The other basic principle is the following: A quantity that is smooth on a certain grid can – without an essential loss of information – also be approximated on a coarser grid, say a grid with double the mesh size. In other words: If we are sure that the error of our approximation has become smooth after some relaxation sweeps, we may approximate this error by a suitable procedure on a (much) coarser grid.

Let us illustrate these considerations by looking at the Fourier expansion of the error. In our model problem the error $e_h = e_h(x, y)$ considered as a function of the discrete variables x and y can be written as

$$e_h(x, y) = \sum_{k,\ell=1}^{n-1} \alpha_{k,\ell} \sin k\pi x \ \sin \ell\pi y \ . \tag{4.1.3}$$

For $(x, y) \in \Omega_h$, the functions

$$\varphi_h^{k,\ell}(x, y) = \sin k\pi x \sin \ell\pi y \quad (k, \ell = 1, \ldots, n-1) \tag{4.1.4}$$

are the discrete eigenfunctions of the discrete operator Δ_h. The fact that this error becomes smooth after some iteration steps means that the *high frequency components* in (4.1.3), i.e. the

$$\alpha_{k,\ell} \sin k\pi x \sin \ell\pi y \quad \text{with} \quad k \ \textit{or} \ \ l \ \textit{large} \tag{4.1.5}$$

become small after a few iterations whereas the *low frequency components*, i.e. the

$$\alpha_{k,\ell} \sin k\pi x \sin \ell\pi y \quad \text{with} \quad k \ \textit{and} \ \ l \ \textit{small} \tag{4.1.6}$$

hardly change.

4.1.2 High and Low Frequencies – and Coarse Meshes

We again consider Model Problem 1 on a grid Ω_h with mesh size $h = 1/n$. Additionally, we consider Model Problem 1 also on a coarser grid Ω_H with mesh size $H > h$. Assuming that n is an even number, we may choose

$$H = 2h$$

which is a very natural choice in the multigrid context. This choice of the coarse grid is called *standard coarsening*.

For the definition of the high and low frequencies, we return to the eigenfunctions $\varphi^{k,\ell}$ in (4.1.4). For given (k, ℓ), we consider the (four) eigenfunctions

$$\varphi^{k,\ell}, \ \varphi^{n-k,n-\ell}, \ \varphi^{n-k,\ell}, \ \varphi^{k,n-\ell}$$

and observe that they coincide on Ω_{2h} in the following sense:

$$\varphi^{k,\ell}(x, y) = -\varphi^{n-k,\ell}(x, y) = -\varphi^{k,n-\ell}(x, y) = \varphi^{n-k,n-\ell}(x, y) \quad \text{for} \quad (x, y) \in \Omega_{2h} \ .$$

This means that these four eigenfunctions cannot reasonably be distinguished on Ω_{2h}. (For k or $l = n/2$, the $\varphi^{k,\ell}$ vanish on Ω_{2h}.) This gives rise to the following definition of low and high frequencies:

Definition (in the context of Model Problem 1):
For $k, \ell \in \{1, \cdots, n-1\}$, we denote $\varphi^{k,\ell}$ to be an eigenfunction (or a *component*) of

low frequency	if	$\max(k,\ell) < n/2$	and
high frequency	if	$n/2 \leq \max(k,\ell) < n$	.

Obviously, only the low frequencies *are visible* on Ω_{2h} since all high frequencies coincide with a low frequency on Ω_{2h} (or vanish on Ω_{2h}). For the 1D case with $n = 8$, we illustrate the above definition in Figure 4.1.2.

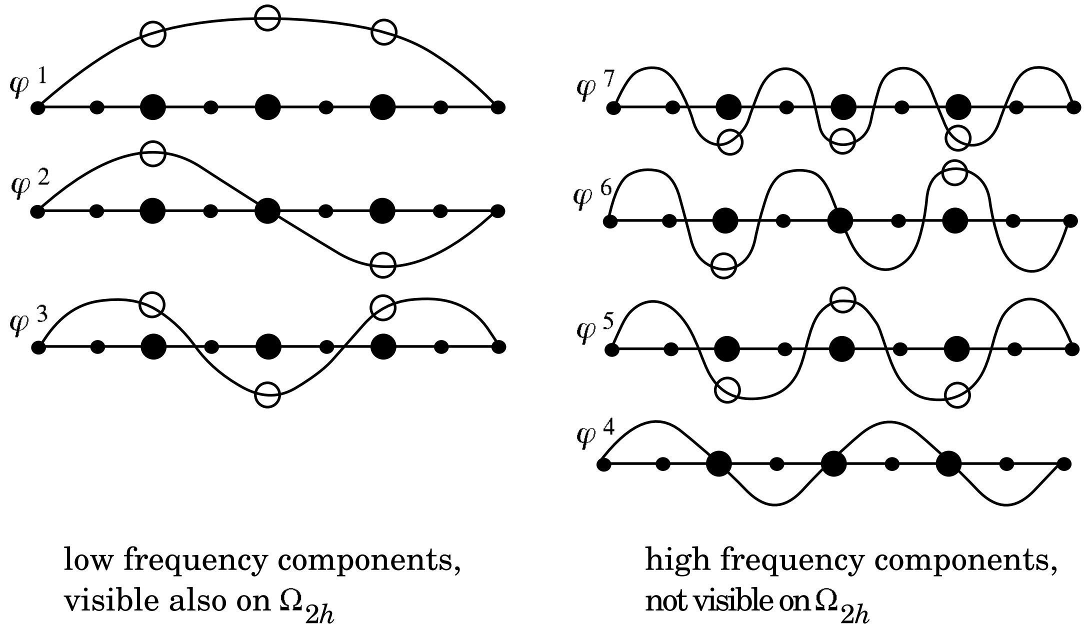

φ^1
φ^2
φ^3
φ^7
φ^6
φ^5
φ^4
low frequency components, visible also on Ω_{2h}
high frequency components, not visible on Ω_{2h}

Figure 4.1.2: Low and high frequency components for a 1D example ($n = 8$).

In the following, we will continue our heuristic considerations a little bit further extending them from two grid levels Ω_h, Ω_{2h} to a sequence of levels. In this setting, we will also be able to explain the *multigrid* (not only the two grid) idea, of course, still very heuristically. Figure 4.1.3 shows such a sequence of grids.

We assume additionally that n is a power of 2 meaning that $h = 2^{-p}$. Then we can form the grid sequence

$$\Omega_h, \; \Omega_{2h}, \; \Omega_{4h}, \cdots, \Omega_{h_0} \tag{4.1.7}$$

just by doubling the mesh size successively. This sequence ends with a coarsest grid Ω_{h_0}.

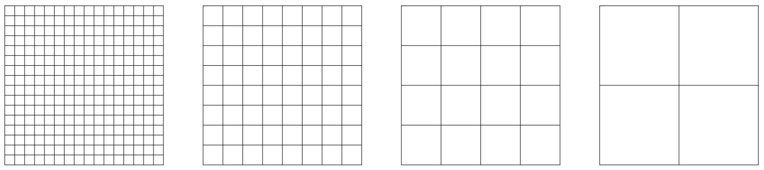


Figure 4.1.3: A sequence of coarse grids starting with $h = 1/16$.

In the same way as we have distinguished low and high frequencies with respect to the pair of grids Ω_h and Ω_{2h} in the previous section, we make an additional distinction between high and low frequencies with respect to the pair Ω_{2h} and Ω_{4h}. (This distinction, of course, only applies to those frequencies which are low with respect to the pair Ω_h and Ω_{2h}). We continue with the pair Ω_{4h} and Ω_{8h} and so on.

4.1.3 Exercises

Exercise 4.1.1 Consider the functions

$$\varphi^{k,\ell}(x,y) = \sin k\pi x \sin \ell\pi x \quad (k,\ell = 1,\dots,n-1)$$

on the grid Ω_h, $h = 1/n$, n even, $\Omega = (0,1)^2$. Show that

$$\varphi^{k,\ell}(x,y) = -\varphi^{n-k,\ell}(x,y) = -\varphi^{k,n-\ell}(x,y) = \varphi^{n-k,n-\ell}(x,y) \quad \text{for} \quad (x,y) \in \Omega_{2h} \ .$$

Exercise 4.1.2 The iteration formula for the classical Jacobi method is

$$\begin{aligned} u_h^{m+1}(x_i,y_j) &= \frac{1}{4}\big(h^2 b_h(x_i,y_j) + u_h^m(x_i-h,y_j) + u_h^m(x_i+h,y_j) + \\ &\quad u_h^m(x_i,y_j-h) + u_h^m(x_i-h,y_j+h)\big) \ . \end{aligned}$$

Derive an error formula for the Jacobi method analogous to the error formula

$$\begin{aligned} e_h^{m+1}(x_i,y_j) &= \frac{1}{4}\big[e_h^{m+1}(x_i-h,y_j) + e_h^m(x_i+h,y_j) \\ &\quad + e_h^{m+1}(x_i,y_j-h) + e_h^m(x_i,y_j+h)\big] \ . \end{aligned}$$

for the Gauss-Seidel method.

4.2 Error Smoothing Procedures

The classical iteration methods, Gauss–Seidel and Jacobi are often called *relaxation* methods (or smoothing methods or smoothers) if they are used for the purpose of error smoothing.

4.2.1 Jacobi-Type Iteration (Relaxation)

For Model Problem 1, the iteration formula of the Jacobi iteration reads

$$\begin{aligned} z_h^{m+1}(x_i,y_j) &= \frac{1}{4}\Big[h^2 b_h(x_i,y_j) + u_h^m(x_i-h,y_j) + u_h^m(x_i+h,y_j) \\ &\quad + u_h^m(x_i,y_j-h) + u_h^m(x_i,y_j+h)\Big] \\ u_h^{m+1} &= z_h^{m+1} \ , \end{aligned} \tag{4.2.1}$$

(with $(x_i,y_j) \in \Omega_h$) where u_h^m denotes the old and u_h^{m+1} the new approximation of the iteration. The corresponding Jacobi iteration operator is

$$S_h = I_h - \frac{h^2}{4}L_h$$

(where I_h is the identity operator). We can generalize this iteration by introducing a relaxation parameter ω:

$$u_h^{m+1} = u_h^m + \omega(z_h^{m+1} - u_h^m) \ , \tag{4.2.2}$$

which is called the ω-(damped) Jacobi relaxation (ω-JAC). Obviously, ω-JAC and Jacobi iteration coincide for $\omega = 1$. The operator for ω-JAC reads

$$S_h(\omega) = I_h - \frac{\omega h^2}{4}L_h = \frac{\omega}{4}\begin{bmatrix} & 1 & \\ 1 & 4(\frac{1}{\omega}-1) & 1 \\ & 1 & \end{bmatrix}_h \ . \tag{4.2.3}$$

The *convergence properties* of ω-JAC can be analyzed easily by considering the eigenfunctions of S_h, which are the same as those of L_h, namely

$$\varphi_h^{k,\ell}(x) = \sin k\pi x \sin \ell\pi y \quad ((x,y) \in \Omega_h; (k,\ell = 1,\ldots,n-1)) \ . \tag{4.2.4}$$

The corresponding eigenvalues of S_h are

$$\lambda_{k,\ell} = \lambda_{k,\ell}(\omega) = 1 - \frac{\omega}{2}(2 - \cos k\pi h - \cos \ell\pi h) \ . \tag{4.2.5}$$

For the spectral radius $\rho(S_h) = \max\{|\lambda_{k,\ell}| : (k,\ell = 1,\ldots,n-1)\}$, we obtain

$$\begin{aligned} \text{for } 0 < \omega \le 1: &\quad \rho(S_h) = |\lambda_{1,1}| = |1 - \omega(1-\cos\pi h)| = 1 - 0(h^2) \\ \text{else}: &\quad \rho(S_h) \ge 1 \quad \text{(for } h \text{ small enough)} \ . \end{aligned} \tag{4.2.6}$$

In particular, when regarding the (unsatisfactory) asymptotic convergence, there is no use in introducing the relaxation parameter: $\omega = 1$ is the best choice.

4.2.2 Smoothing Properties of ω-Jacobi Relaxation

The situation is different with respect to the *smoothing properties* of ω-Jacobi relaxation: In order to achieve reasonable smoothing, we have to introduce a parameter $\omega \neq 1$ here.

For $0 < \omega \leq 1$, we first observe from (4.2.6) that the smoothest eigenfunction $\varphi_h^{1,1}$ is responsible for the slow convergence of Jacobi's method. Highly oscillating eigenfunctions are reduced much faster *if ω is chosen properly.* To see this, we consider the approximations before (w_h) and after ($\overline{w}_h$) one relaxation step and expand the errors before (e_h) and after ($\overline{e_h}$) the relaxation step, namely

$$e_h := u_h - w_h \quad \text{and} \quad \overline{e}_h := u_h - \overline{w}_h \ ,$$

into discrete eigenfunction series:

$$e_h = \sum_{k,\ell=1}^{n-1} \alpha_{k,\ell}\varphi_h^{k,\ell} \ , \quad \overline{e}_h = \sum_{k,\ell=1}^{n-1} \lambda_{k,\ell}\alpha_{k,\ell}\varphi_h^{k,\ell} \ . \tag{4.2.7}$$

In order to measure the smoothing properties of $S_h(\omega)$ quantitatively, we introduce the *smoothing factor* of $S_h(\omega)$. The *smoothing factor* $\mu(h;\omega)$ of $S_h(\omega)$, representing the worst factor by which high frequency error components are reduced per relaxation step, and its supremum μ^* over h, are defined as

$$\begin{aligned} \mu(h;\omega) &:= \max\{|\lambda_{k,\ell}(\omega)| : n/2 \leq \max(k,\ell) \leq n-1\} \ , \\ \mu^*(\omega) &:= \sup_{h\in\mathcal{H}} \mu(h;\omega) \ . \end{aligned} \tag{4.2.8}$$

$\mathcal{H}$ denotes the set of admissible (or reasonable) mesh sizes. Below we will see, for example, that for $\Omega = (0,1)^2$ the coarsest grid on which smoothing is applied is characterized by $h = 1/4$. In this case we would then define $\mathcal{H} = \{h = 1/n : h \leq 1/4\}$.

Inserting (4.2.5), we get from (4.2.8)

$$\begin{aligned} \mu(h;\omega) &= \max\{|1 - \frac{\omega}{2}(2 - \cos k\pi h - \cos \ell\pi h)| : n/2 \leq \max(k,\ell) \leq n-1\} \\ \mu^*(\omega) &= \max\{|1-\omega/2|, |1-2\omega|\} \ . \end{aligned} \tag{4.2.9}$$

This shows that Jacobi's relaxation has no smoothing properties for $\omega \leq 0$ or $\omega > 1$:

$$\mu(h;\omega) \geq 1 \text{ if } \omega \leq 0 \text{ or } \omega > 1 \quad \text{(and } h \text{ sufficiently small)} \ .$$

For $0 < \omega < 1$, however, the smoothing factor is smaller than 1 and bounded away from 1, independently of h. For $\omega = 1$, we have a smoothing factor of $1 - O(h^2)$ only. In particular, we find from (4.2.9):

$$\boxed{\mu(h;\omega) = \begin{cases} \cos \pi h & \text{if } \omega = 1 \\ (2+\cos \pi h)/4 & \text{if } \omega = 1/2 \\ (1+2\cos \pi h)/5 & \text{if } \omega = 4/5 \end{cases} \qquad \mu^*(\omega) = \begin{cases} 1 & \text{if } \omega = 1 \\ 3/4 & \text{if } \omega = 1/2 \\ 3/5 & \text{if } \omega = 4/5 \end{cases} . \quad (4.2.10)}$$

The choice $\omega = 4/5$ is optimal in the following sense:

$$\inf \ \{\mu^*(\omega) : 0 \le \omega \le 1\} = \mu^*(4/5) = 3/5 \ .$$

With respect to $\mu(h,\omega)$, one obtains

$$\inf \ \{\mu(h;\omega) : 0 \le \omega \le 1\} = \mu\left(h; \frac{4}{4+\cos \pi h}\right) = \frac{3\cos \pi h}{4+\cos \pi h} = \frac{3}{5} - |0(h^2)| \ .$$

This means that one step of ω-JAC with $\omega = 4/5$ reduces all high frequency error components at least by a factor of 3/5 (independent of the grid size h).

4.2.3 Gauss-Seidel-Type Iteration (Relaxation)

The smoothing properties of Gauss-Seidel iterations depend, in general, on relaxation parameters *and* on the ordering of grid points (numbering of unknowns). With respect to the ordering of grid points, this behavior is substantially different from that of Jacobi-type methods, where the ordering is totally irrelevant. In Section 4.1.1 we have introduced Gauss-Seidel iteration with a lexicographic ordering of the grid points. A different ordering is the red-black ordering. If we use this red-black ordering for Gauss-Seidel iteration, we obtain the GS-RB method.

The significance of using a relaxation parameter ω in Gauss-Seidel iteration is well known in the classical Gauss-Seidel *convergence* theory. For Model Problem 1, lexicographic Gauss-Seidel with a relaxation parameter is described by

$$\begin{aligned} z_h^{m+1}(x_i,y_j) &= \frac{1}{4}\Big[h^2 b_h(x_i,y_j) + u_h^{m+1}(x_i-h,y_j) + u_h^m(x_i+h,y_j) \\ &\quad + u_h^{m+1}(x_i,y_j-h) + u_h^m(x_i,y_j+h)\Big] \\ u_h^{m+1} &= u_h^m + \omega(z_h^{m+1} - u_h^m) \qquad (4.2.11) \end{aligned}$$

The parameter ω does not only enter explicitly in (4.2.11), but also implicitly in the “new values” $u_h^{m+1}(x_i-h,y_j)$ and $u_h^{m+1}(x_i,y_j-h)$.

As mentioned before, for Model Problem 1, the convergence of Gauss–Seidel iteration can be substantially improved by an overrelaxation parameter ω: With

$$\omega = \frac{2}{1+\sqrt{1-\rho(\text{JAC})^2}} = \frac{2}{1+\sin \pi h}$$

we obtain $\rho(\omega\text{-GS}) = 1 - O(h)$ instead of $\rho(\text{GS}) = 1 - O(h^2) \quad (\text{for} \quad \omega = 1)$. This is the classical result on *successive overrelaxation* (SOR) [149].

In the multigrid context the *smoothing properties* of Gauss-Seidel are much more important than the convergence properties. Here, we only summarize the results of the smoothing analysis for Gauss-Seidel-type relaxations. For Model Problem 1, we obtain the smoothing factors

$$\begin{aligned} \mu(\text{GS-LEX}) &= 0.50 \qquad (\text{for } \omega = 1) \ , \\ \mu(\text{GS-RB}) &= 0.25 \qquad (\text{for } \omega = 1) \ . \end{aligned}$$

(The factor of 0.25 for GS-RB is valid if only one or two smoothing steps are performed.) This result shows that the ordering of the grid points has an essential influence on the smoothing properties in the case of Gauss-Seidel relaxations. Furthermore, for Model Problem 1, the introduction of a relaxation parameter does *not* improve the smoothing properties of GS-LEX relaxation essentially.

4.2.4 Exercises

Exercise 4.2.1 For $0 \le \omega \le 1$, the smoothing factor $\mu(h;\omega)$ of ω-JAC for Model Problem 1 is

$$\mu(h;\omega) = \max\{|1 - \frac{\omega}{2}(2 - \cos k\pi h - \cos \ell\pi h)| : n/2 \le \max(k,\ell) \le n-1\} \ .$$

Show that

$$\mu^*(\omega) := \sup_{h \in \mathcal{H}} \mu(h;\omega) = \max\{|1-\omega/2|, |1-2\omega|\} \ .$$

with $\mathcal{H}$ the set of admissible mesh sizes ($h > 1/2$, for example).

Exercise 4.2.2 Let

$$\mu^*(\omega) = \max\{|1-\omega/2|, |1-2\omega|\} \ .$$

Show that

$$\inf \ \{\mu^*(\omega) : 0 \le \omega \le 1\} = \mu^*(4/5) = 3/5 \ .$$

Exercise 4.2.3 Let

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

be an $(n-1) \times (n-1)$ matrix, that originates from a discretization of u'' on $\Omega = [0,1]$ with $h = 1/n$.

(a) Confirm by computation that the

$$\lambda_k \;=\; 2 - 2\cos k\pi h$$

are the eigenvalues and that

$$\varphi^k \;=\; (\sin k\pi i h)_{i=1,\ldots,n-1}$$

are the eigenvectors of A for all $k = 1, \ldots, n-1$.

(b) Show that $\lambda = \cos(k\pi h)$ for $k = 1, .., n-1$ are the eigenvalues of the Jacobi iteration matrix Q_{JAC} for the iterative solution of the linear problem $Au = b$, with A from (a). Determine the spectral radius $\rho(Q_{\mathrm{JAC}}) := \max_{k=1,\ldots,n-1} |\lambda_k|$.

(c) Show that $\lambda = \cos^2(k\pi h)$ for $k = 1, .., n-1$ are the eigenvalues of Q_{GS} for the Gauss–Seidel iteration matrix for the iterative solution of the linear problem $Au = b$, with A from (a). Determine $\rho(Q_{\mathrm{GS}})$.

(d) Let A be the $(n-1)^2 \times (n-1)^2$ matrix from a discretization of the Laplace operator on $\Omega = [0,1]^2$ with $h = \frac{1}{n}$. With a lexicographic ordering (from left to right and from down to up) of grid points, the matrix A is

$$A \;=\; \begin{pmatrix} D & -I & & & \\ -I & D & -I & & \\ & -I & D & -I & \\ & & \ddots & \ddots & \ddots \\ & & & -I & D \end{pmatrix}$$

with

$$D \;=\; \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & 4 & -1 & \\ & & \ddots & \ddots & \ddots \\ & & & -1 & 4 \end{pmatrix}.$$

and I identity matrix blocks. Confirm by computation that

$$\lambda_{k,\ell} \;=\; 4 - 2\cos k\pi h - 2\cos \ell\pi h$$

are the eigenvalues and

$$\varphi^{k,\ell} \;=\; (\sin k\pi i h \sin \ell\pi j h)_{i,j=1,\ldots n-1}$$

are the eigenvectors of A for all $k, \ell = 1, \ldots, n-1$.

Exercise 4.2.4 Consider the problem

$$\begin{aligned} \mathcal{L}u(x) = -u''(x) = b(x) \quad &\text{in } (0,1) \\ u(0) = u(1) = 0, \end{aligned}$$

discretized by the operator L_h, as follows

$$\frac{1}{h^2}[-1 \quad 2 \quad -1]_h u_h = b_h, \quad h = \frac{1}{n},\ n \text{ even}.$$

The corresponding *normalized* eigenfunctions φ_h^k are

$$\varphi_h^k(x_i) = \sqrt{2h}\sin(k\pi x_i) \text{ for } k = 1, ..., n-1, \text{ with } x_i = ih, i = 0, ..., n$$

and eigenvalues $4/h^2 \sin^2(k\pi h/2)$.

(a) Determine the eigenfunctions and eigenvalues $\lambda_k(\omega\text{-JAC})$ of the damped Jacobi iteration (ω-JAC)

$$u_h^{j+1} = u_h^j - \omega D_h^{-1}(L_h u_h^j - b_h)$$

with $\omega \in (0,1)$ and D_h the diagonal part of the discretization L_h.

(b) Choose ω-JAC with $\omega = 0.5$. Show that $\rho(Q_{\text{JAC}}) = 1 - ah^2 + O(h^4)$ and determine a.

(c) The smoothing properties of iterative schemes can be analyzed if one divides the spectrum into low ($1 \le k < n/2$) and high ($n/2 \le k < n$) frequencies. In the 1D case, the smoothing factor is defined as

$$\mu(\omega_{\text{JAC}})_h = \max\{|\lambda_k(\omega_{\text{JAC}})| : n/2 + 1 \le k < n\}.$$

Determine the smoothing factor for ω-JAC with $\omega = 0.5$.

Exercise 4.2.5 Let the linear system $Au = b$ be given with

$$A := \begin{pmatrix} 3 & -2 & 0 & 0 \\ -1 & 3 & -2 & 0 \\ 0 & -1 & 3 & -2 \\ 0 & 0 & -1 & 3 \end{pmatrix}, \quad b := \begin{pmatrix} -1 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

An SOR matrix iteration reads:

$$B(\omega)u^{m+1} = (B(\omega) - A)u^m + b,$$

with $B(\omega) = 1/\omega D(I - \omega L)$. Here, D is the diagonal part of matrix A, L is the strickly lower diagonal part of A.

Matrix A has such properties that the optimal relaxation parameter ω can be computed as $\omega = \frac{2}{1+\sqrt{1-\rho(Q_{\text{JAC}})}}$ Compute ω. Set up the SOR iteration and perform one iteration with ω towards the solution of the linear system $Au = b$ using the initial approximation $u = 0$.

Exercise 4.2.6 Let a system $Au = b$ be given with

$$A := \begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} 24 \\ 30 \\ -24 \end{pmatrix}.$$

Show that the given matrix is symmetric positive definite. Compute the optimal relaxation parameter ω (see Exercise 4.2.5) for the SOR iterative scheme and compare the spectral radii of the Jacobi-, Gauss–Seidel- and the SOR iteration.

Exercise 4.2.7 Use the approximation

$$\rho(Q_{JAC}) \approx 1 - \frac{1}{2}\pi^2 h^2 \approx 1 - 5h^2$$

to determine how many iterations of Jacobi relaxation with relaxation paramater $\omega = 1$, are needed to reduce the error by a factor of $\varepsilon = 10^{-3}$ for

a) $h = 2^{-4}$
b) $h = 2^{-10}$.

Exercise 4.2.8 Use the formula

$$\rho(Q_{\omega\text{-GS}}) = \frac{1 - \sin \pi h}{1 + \sin \pi h}$$

for the spectral radius of Gauss–Seidel relaxation with the optimal parameter

$$\omega = \frac{2}{1 + \sqrt{1 - \rho(Q_{\text{JAC})^2}}} = \frac{2}{1 + \sin \pi h}$$

in the form

$$\rho(Q_{\omega\text{-GS}}) \approx 1 - 2\pi h \approx 1 - 6h$$

to determine how many iterations are needed to reduce the error by a factor of $\varepsilon = 10^{-3}$ for

a) $h = 2^{-4}$
b) $h = 2^{-10}$.

Compare the result with that of Exercise 4.2.7.

Exercise 4.2.9 Compare the convergence of ordinary Gauss–Seidel relaxation ($\omega = 1$) and optimally overrelaxed Gauss–Seidel relaxation. To do this, use the estimates

$$\begin{aligned} \rho(Q_{\text{GS}}) &\approx 1 - \pi^2 h^2 \approx 1 - 10 \cdot h^2 \\ \rho(Q_{\omega-\text{GS}}) &\approx 1 - 2\pi h \approx 1 - 6h\,. \end{aligned}$$

Compare the number of iterations needed by each of the methods to reduce the error by a factor of $\varepsilon = 10^{-3}$ for

a) $h = 2^{-4}$
b) $h = 2^{-10}$.

4.3 Introducing the Two–Grid Cycle

Basic multigrid consists of two ingredients: smoothing and coarse grid correction. We start with the two–grid cycle, the natural basis for any multigrid algorithm. For this purpose, we consider a discrete linear elliptic boundary value problem of the form

$$L_h u_h = b_h \quad (\Omega_h) \ . \tag{4.3.1}$$

and assume ${L_h}^{-1}$ to exist.

4.3.1 Iteration by Approximate Solution of the Defect Equation

For any *approximation* u_h^m of the solution u_h of (4.3.1), the *error* is given by $e_h^m := u_h - u_h^m$ and the *defect* (or *residual*) by $r_h^m := b_h - L_h u_h^m$. Trivially, the *defect equation* $L_h e_h^m = r_h^m$ is equivalent to the original equation (4.3.1) since $u_h = u_h^m + e_h^m$. If L_h is approximated here by any "simpler" operator $\widehat{L}_h$ such that $\widehat{L}_h^{-1}$ exists, the solution $\widehat{e}_h^m$ of

$$\widehat{L}_h \widehat{e}_h^m = r_h^m \tag{4.3.2}$$

gives a new approximation

$$u_h^{m+1} := u_h^m + \widehat{e}_h^m. \tag{4.3.3}$$

The procedural formulation then looks like

$$\boxed{u_h^m \longrightarrow r_h^m = b_h - L_h u_h^m \longrightarrow \widehat{L}_h \widehat{e}_h^m = r_h^m \longrightarrow u_h^{m+1} = u_h^m + \widehat{e}_h^m} \ . \tag{4.3.4}$$

Starting with some u_h^0, the successive application of this process defines an *iterative procedure.* The *iteration operator* of this method is given by

$$Q_h = I_h - M_h^{-1} L_h : \quad \mathcal{G}(\Omega_h) \to \mathcal{G}(\Omega_h), \tag{4.3.5}$$

where $M_h^{-1} := (\widehat{L}_h)^{-1}$ and I_h denotes the identity on $\mathcal{G}(\Omega_h)$. We have

$$u_h^{m+1} = Q_h u_h^m + s_h \ \text{ with } \ s_h = (\widehat{L})_h^{-1} b_h \quad (m = 0, 1, 2, \ldots) \ . \tag{4.3.6}$$

For the errors, it follows that

$$e_h^{m+1} = Q_h e_h^m = \Big(I_h - M_h^{-1} L_h \Big) e_h^m \quad (m = 0, 1, 2, \ldots) \tag{4.3.7}$$

and for the defects that

$$r_h^{m+1} = L_h Q_h {L_h}^{-1} r_h^m = \Big(I_h - L_h M_h^{-1} \Big) r_h^m \quad (m = 0, 1, 2, \ldots) \ . \tag{4.3.8}$$

If we start the iteration with $u^0 = 0$, then we can represent u_h^m $(m = 1, 2, \ldots)$ as

$$\begin{aligned} u_h^m &= \Big(I_h + Q_h + {Q_h}^2 + \cdots + {Q_h}^{m-1} \Big)(\widehat{L}_h)^{-1} b_h \\ &= (I_h - {Q_h}^m)(I_h - Q_h)^{-1}(\widehat{L}_h)^{-1} b_h \\ &= \Big(I_h - {Q_h}^m \Big) {L_h}^{-1} b_h \; . \end{aligned} \tag{4.3.9}$$

The asymptotic convergence properties of the above iterative process are characterized by the *spectral radius (asymptotic convergence factor)* of the iteration operator, i.e.

$$\rho(Q_h) = \rho(I_h - M_h^{-1} L_h) = \rho(I_h - L_h M_h^{-1}) \; . \tag{4.3.10}$$

If some norm $\| \cdot \|$ on $\mathcal{G}(\Omega_h)$ is introduced,

$$\|I_h - M_h^{-1} L_h\|, \; \|I_h - L_h M_h^{-1}\| \tag{4.3.11}$$

give the *error reduction factor* and the *defect reduction factor*, respectively, for one iteration.

Remark 4.3.1 Classical iterative linear solvers such as Jacobi or Gauss-Seidel iteration if applied to (4.3.1) can be interpreted as (iterated) approximate solvers for the defect equation. For ω–JAC we have, for example,

$$\widehat{L}_h = \frac{1}{\omega} D_h$$

where D_h is the "diagonal" part of the matrix corresponding to L_h. Similarly, GS–LEX is obtained by setting $\widehat{L}_h$ to be the "lower triangular" part of the matrix corresponding to L_h including its diagonal part. $\gg$

4.3.2 Coarse Grid Correction

The idea to approximately solve the defect equation is to use an appropriate approximation L_H of L_h on a coarser grid Ω_H, for instance the grid with mesh size $H = 2h$. This means that the defect equation is replaced by

$$L_H \widehat{e}_H^m = r_H^m. \tag{4.3.12}$$

Here we assume

$$L_H : \mathcal{G}(\Omega_H) \to \mathcal{G}(\Omega_H), \quad \dim \mathcal{G}(\Omega_H) < \; \dim \mathcal{G}(\Omega_h) \tag{4.3.13}$$

and ${L_H}^{-1}$ to exist. As r_H^m and $\widehat{e}_H^m$ are grid functions on the coarser grid Ω_H, we assume two (linear) transfer operators

$$I_h^H : \mathcal{G}(\Omega_h) \to \mathcal{G}(\Omega_H), \quad I_H^h : \mathcal{G}(\Omega_H) \to \mathcal{G}(\Omega_h) \tag{4.3.14}$$

to be given. I_h^H is used to *restrict* r_h^m to Ω_H:

$$\boxed{r_H^m := I_h^H r_h^m} \quad , \tag{4.3.15}$$

and I_H^h is used to *interpolate* (or *prolongate*) the correction $\widehat{e}_H^m$ to Ω_h:

$$\boxed{\widehat{e}_h^m := I_H^h \widehat{e}_H^m} \quad . \tag{4.3.16}$$

The simplest example for a restriction operator is the "injection" operator

$$r_H(P) = I_h^H r_h(P) := r_h(P) \qquad \text{for} \quad P \in \Omega_H \subset \Omega_h \ , \tag{4.3.17}$$

which identifies approximations at coarse grid points with the corresponding approximations at fine grid points. A fine and a coarse grid with the injection operator are presented in Figure 4.3.4. Altogether, one coarse grid correction

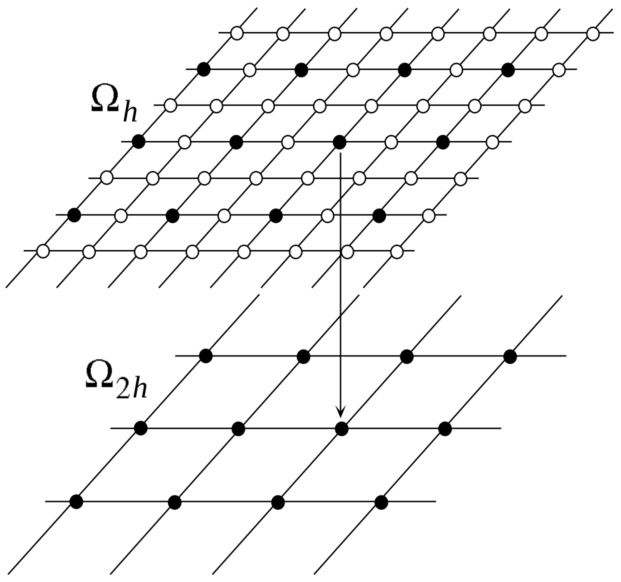


Figure 4.3.4: A fine and a coarse grid with the injection operator.

step (calculating u_h^{m+1} from u_h^m) proceeds as follows:

<u>Coarse grid correction</u> $u_h^m \to u_h^{m+1}$

–	Compute the defect	$r_h^m = b_h - L_h u_h^m$
–	Restrict the defect (fine–to–coarse transfer)	$r_H^m = I_h^H r_h^m$
–	Solve exactly on Ω_H	$L_H \widehat{e}_H^m = r_H^m$
–	Interpolate the correction (coarse–to–fine transfer)	$\widehat{e}_h^m = I_H^h \widehat{e}_H^m$
–	Compute a new approximation	$u_h^{m+1} = u_h^m + \widehat{e}_h^m$

The associated iteration operator is given by

$$I_h - M_h^{-1} L_h \qquad \text{with} \qquad M_h^{-1} = I_H^h {L_H}^{-1} I_h^H . \tag{4.3.18}$$

However: Taken on its own, the coarse grid correction process is of no use: It is not convergent! We have

$$\rho\Big(I_h - I_H^h \, {L_H}^{-1} I_h^H L_h \Big) \geq 1 \ . \tag{4.3.19}$$

This follows directly from the fact that I_h^H maps $\mathcal{G}(\Omega_h)$ into the lower dimensional space $\mathcal{G}(\Omega_H)$ and therefore $M_h^{-1} = I_H^h {L_H}^{-1} I_h^H$ is not invertible. This implies that

$$M_h^{-1} L_h e_h = 0 \quad \text{for certain} \quad e_h \neq 0 \ .$$

It may be illustrative to see what $M_h^{-1} L_h e_h = 0$ means in practice. For the simple injection operator (4.3.17), for example, any error function $e_h \in \mathcal{G}(\Omega_h)$ with

$$L_h e_h(P) = \begin{cases} 0 & \text{for } P \in \Omega_H \\ \text{arbitrary} & \text{for } P \notin \Omega_H \end{cases}$$

is annihilated by I_h^H and therefore by M_h^{-1}. Such an error function e_h will thus not be changed by a pure coarse grid correction.

Two-grid cycle The above considerations imply that it is necessary to combine the two processes of smoothing and of coarse grid correction. In general, the interpolation of coarse grid corrections reintroduces high frequency error components on the fine grid. One natural approach to reduce them is to introduce one or a few additional smoothing sweeps after the coarse grid correction. These sweeps are commonly denoted as *post-smoothing*.

Each iteration step (cycle) of a two-grid method consists of a *pre-smoothing*, a *coarse grid correction* and a *post-smoothing* part. One step of such an iterative two-grid (h, H)–method (calculating u_h^{m+1} from u_h^m) proceeds as follows:

Two–grid cycle $\mathsf{u}_{\mathsf{h}}^{\mathsf{m}+1} = \mathsf{TGCYC}\Big(u_h^m, L_h, b_h, \nu_1, \nu_2 \Big)$

(1) Pre–smoothing

- Compute $\overline{u}_h^m$ by applying $\nu_1 (\geq 0)$ steps of a given smoothing procedure to u_h^m:

$$\overline{u}_{\mathsf{h}}^{\mathsf{m}} = \mathsf{SMOOTH}^{\nu_1}\Big(u_h^m, L_h, b_h \Big) \quad . \tag{4.3.20}$$

(2) Coarse grid correction (CGC)

- Compute the defect $\qquad \overline{r}_{\mathsf{h}}^{\mathsf{m}} = \mathsf{b}_{\mathsf{h}} - \mathsf{L}_{\mathsf{h}}\overline{u}_{\mathsf{h}}^{\mathsf{m}}$.
- Restrict the defect (fine–to–coarse transfer) $\qquad \overline{r}_{\mathsf{H}}^{\mathsf{m}} = \mathsf{I}_{\mathsf{h}}^{\mathsf{H}}\overline{r}_{\mathsf{h}}^{\mathsf{m}}$.
- Solve on Ω_H

$$\mathsf{L}_{\mathsf{H}}\widehat{\mathsf{e}}_{\mathsf{H}}^{\mathsf{m}} = \overline{r}_{\mathsf{H}}^{\mathsf{m}} \quad . \tag{4.3.21}$$

- Interpolate the correction (coarse–to–fine transfer) $\qquad \widehat{\mathsf{e}}_{\mathsf{h}}^{\mathsf{m}} = \mathsf{I}_{\mathsf{H}}^{\mathsf{h}}\widehat{\mathsf{e}}_{\mathsf{H}}^{\mathsf{m}}$.
- Compute the corrected approximation $\qquad \mathsf{u}_{\mathsf{h}}^{\mathsf{m,after\ CGC}} = \overline{u}_{\mathsf{h}}^{\mathsf{m}} + \widehat{\mathsf{e}}_{\mathsf{h}}^{\mathsf{m}}$.

(3) Post–smoothing

- Compute u_h^{m+1} by applying $\nu_2 (\geq 0)$ steps of the given smoothing procedure to $u_h^{m,\mathsf{after\ CGC}}$:

$$\mathsf{u}_{\mathsf{h}}^{\mathsf{m}+1} = \mathsf{SMOOTH}^{\nu_2}\Big(u_h^{m,\mathsf{after\ CGC}}, L_h, b_h \Big) \quad . \tag{4.3.22}$$

From the above description, one immediately obtains the iteration operator Q_h^H of the (h,H) two–grid cycle:

$$\boxed{Q_h^H = {S_h}^{\nu_2} K_h^H {S_h}^{\nu_1} \qquad \text{with} \qquad K_h^H := I_h - I_H^h {L_H}^{-1} I_h^H L_h} \quad . \tag{4.3.23}$$

The following individual *components of the* (h,H)*–cycle* have to be specified:

- the smoothing procedure SMOOTH (u_h^m, L_h, b_h)
- the numbers ν_1, ν_2 of smoothing steps,
- the coarse grid Ω_H,
- the fine–to–coarse restriction operator I_h^H,
- the coarse grid operator L_H,
- the coarse–to–fine interpolation operator I_H^h.

4.3.3 Multigrid Components

We introduce and list some important examples of how some of the *multigrid components* can be specified, i.e.

- coarse grid specification Ω_H,
- transfer operator: restriction I_h^H,
- transfer operator: interpolation I_H^h and
- coarse grid operator L_H.

We will mention some possible and common choices for the grid Ω_H. The simplest and most frequently used choice is *standard coarsening*, doubling the mesh size h in every direction. Most of the results and considerations in this book refer to this choice. In d dimensions, the relation between the number of grid points (neglecting boundary effects) is

$$\#\Omega_H \approx \frac{1}{2^d}\#\Omega_h \quad .$$

We speak of *semi–coarsening* in 2D if the mesh size h is doubled in one direction only, i.e. $H = (2h_x, h_y)$ (x-semi-coarsening, see Figure 4.3.5) or $H = (h_x, 2h_y)$ (y-semi-coarsening). This is especially of interest for anisotropic operators. Note that in this case

$$\#\Omega_H \approx \frac{1}{2}\#\Omega_h \quad . \tag{4.3.24}$$

We speak of red-black coarsening if the coarse grid points are distributed in a checkerboard manner. Another coarsening is $4h-$coarsening.

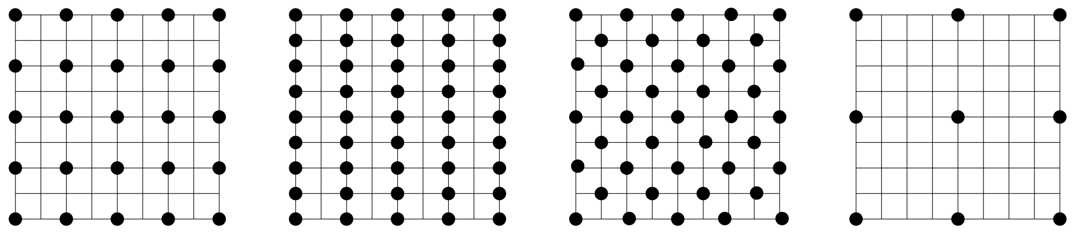


Figure 4.3.5: Examples of standard, x-semi–, red–black and $h-4h$ coarsening in a square computational domain Ω. The grid points of Ω_H are marked by dots.

We finally mention that in the context of the Algebraic Multigrid approach (AMG), the coarse grid Ω_H is not formed according to such a fixed simple strategy. Using the algebraic relations in the corresponding matrix, Ω_H is determined by the AMG process itself in the course of calculation.

Up to now, we have not yet described precisely how the coarse grid operator L_H can be chosen. A natural choice is to use the direct analog of L_h on the grid Ω_H. For Model Problem 1, this means

$$L_H u_H = \frac{1}{H^2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}_H \quad .$$

The direct coarse grid analog of the fine grid operator L_h will be used in most parts of this book, in particular in all chapters on basic multigrid.

There are, however, applications and multigrid algorithms, which make use of a different approach. *The so-called Galerkin coarse grid operator is defined by*

$$L_H := I_h^H L_h I_H^h \ , \tag{4.3.25}$$

where I_h^H and I_H^h are appropriate transfer operators. The Galerkin approach has its benefits in the context of problems with discontinuous coefficients and in algebraic multigrid methods, where algebraic systems of equations without a grid-oriented background will be treated. In the latter situation, the natural approach is not available.

The choice of restriction and interpolation operators I_h^H and I_H^h, for the intergrid transfer of grid functions, is closely related to the choice of the coarse grid. Here, we introduce transfer operators in the case of standard coarsening (see Figure 4.3.5), i.e. the grid transfers between the grid Ω_h and the $2h$–grid Ω_{2h}.

A *restriction operator* I_h^{2h} maps h-grid functions to $2h$-grid functions. One restriction operator already discussed is the *injection* operator (4.3.17). Another frequently used restriction operator is the *Full Weighting* operator (FW), which in stencil notation reads

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_h^{2h} \ . \tag{4.3.26}$$

Applying this operator to a grid function $r_h(x,y)$ at a coarse grid point $(x,y) \in \Omega_{2h}$ means

$$\begin{aligned} r_{2h}(x,y) &= I_h^{2h} r_h(x,y) \\ &= \frac{1}{16}\Big[4r_h(x,y) \quad + \quad 2r_h(x+h,y) + 2r_h(x-h,y) \\ &\qquad\qquad\qquad + \quad 2r_h(x,y+h) + 2r_h(x,y-h) \\ &\qquad\qquad\qquad + \quad r_h(x+h,y+h) + r_h(x+h,y-h) \\ &\qquad\qquad\qquad + \quad r_h(x-h,y+h) + r_h(x-h,y-h)\Big] \ . \end{aligned} \tag{4.3.27}$$

Obviously, a 9–point weighted average of r_h is obtained.

The *interpolation (prolongation) operators* map $2h$–grid functions into h–grid functions. A very frequently used interpolation method is *bilinear interpolation* from G_{2h} to G_h, which is given by:

$$I_{2h}^h \widehat{e}_{2h}(x,y) = \begin{cases} \widehat{e}_{2h}(x,y) & \text{for } \bullet \\ \frac{1}{2}\Big[\widehat{e}_{2h}(x,y+h) + \widehat{e}_{2h}(x,y-h)\Big] & \text{for } \square \\ \frac{1}{2}\Big[\widehat{e}_{2h}(x+h,y) + \widehat{e}_{2h}(x-h,y)\Big] & \text{for } \diamond \\ \frac{1}{4}\Big[\widehat{e}_{2h}(x+h,y+h) + \widehat{e}_{2h}(x+h,y-h) & \\ \quad + \widehat{e}_{2h}(x-h,y+h) + \widehat{e}_{2h}(x-h,y-h)\Big] & \text{for } \bigcirc \end{cases} \quad (4.3.28)$$

Figure 4.3.6 presents (part of) a fine grid with the symbols for the fine and coarse grid points referred to by formula (4.3.28).

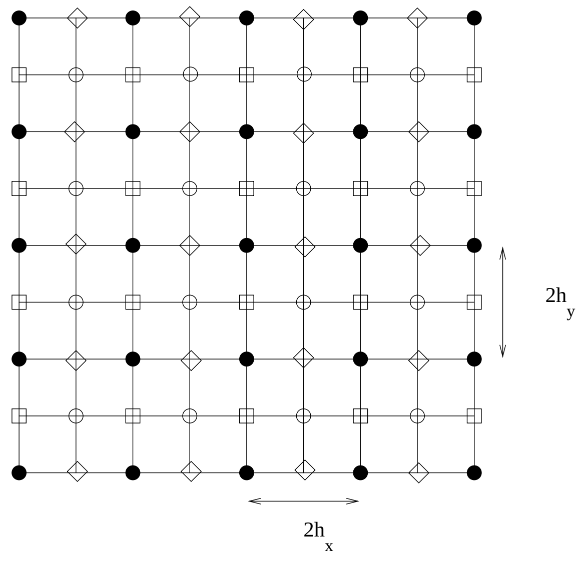


Figure 4.3.6: A fine grid with symbols indicating the bilinear interpolation (4.3.28) used for the transfer from the coarse grid (•).

The formulae for linear interpolation can be applied near Dirichlet boundaries even if the boundary conditions have been eliminated. The corrections from a boundary point are assumed to be 0 in this case.

Multigrid cycle It is neither useful nor necessary to solve the coarse grid defect equation (4.3.21) exactly. Instead, without essential loss of convergence speed, one may replace $\widehat{e}_H^m$ by a suitable approximation. A natural way to obtain such an approximation is to apply the two–grid idea to the Equation (4.3.21) again, now employing an even coarser grid than Ω_H.

This is possible, as obviously the coarse grid equation (4.3.21) is of the same form as the original equation (4.3.1). If the convergence factor of the two–grid method is small enough, it is sufficient to perform only a few, say γ (see Figure 4.3.7), two–grid iteration steps to obtain a good enough approximation to the solution of (4.3.21). This idea can, in a straightforward manner, be applied recursively, using coarser and coarser grids, down to some coarsest grid.

On this coarsest grid any solution method may be used (e.g. a direct method or some relaxation type method if it has sufficiently good convergence properties on that coarsest grid). In ideal cases, the coarsest grid consists of just one grid point. On a very coarse grid, classical iterative solvers are usually satisfactory solvers; their bad convergence properties just appear for $h \to 0$.

Figure 4.3.7 illustrates the structure of *one iteration step* (*cycle*) of a multigrid method with a few pictures. Usually, the cases $\gamma = 1$ and $\gamma = 2$ are particularly interesting.

For obvious reasons, we refer to the case $\gamma = 1$ as *V–cycles* and to $\gamma = 2$ as *W–cycles*. The number γ is also called *cycle index*.

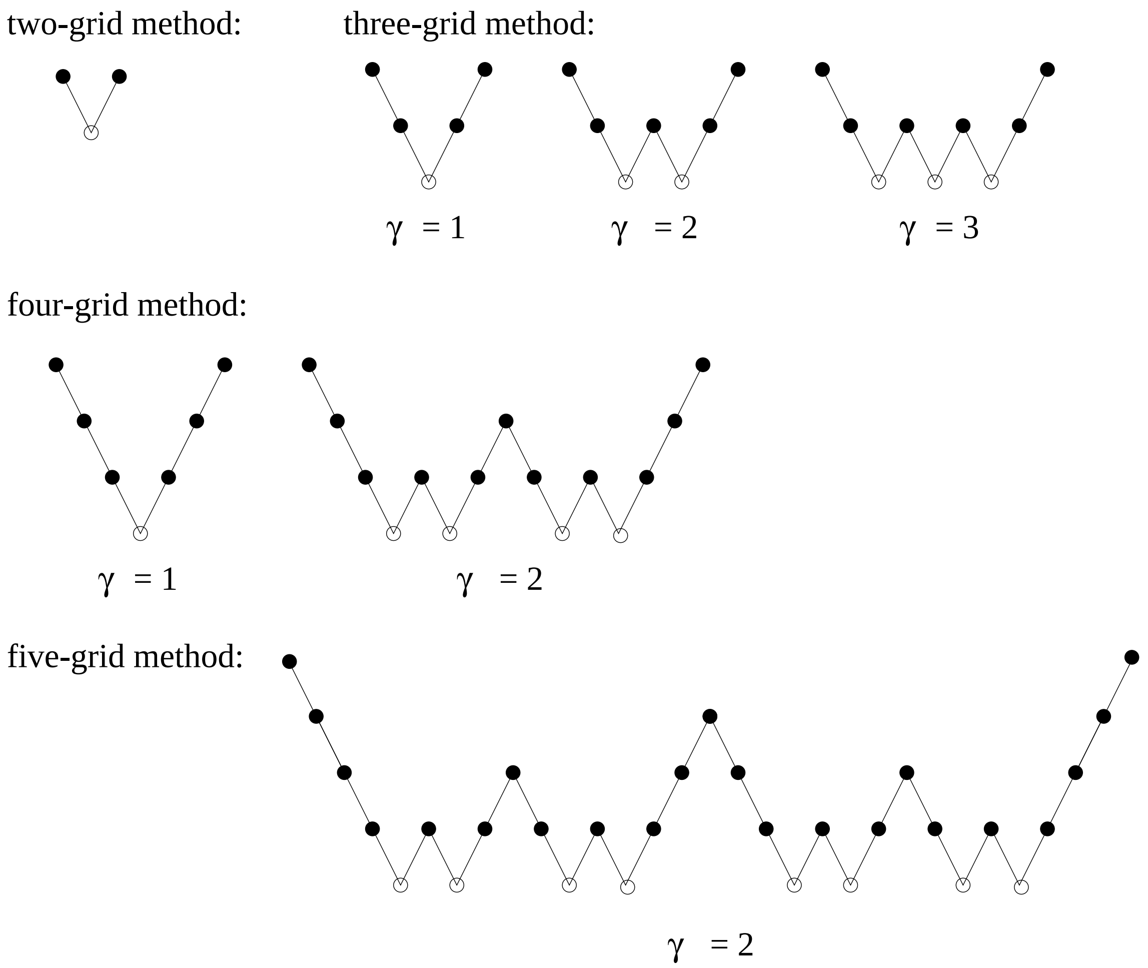


Figure 4.3.7: Structure of one multigrid cycle for different numbers of grids and different values of the cycle index γ ($\bullet$ stands for smoothing, $\circ$ for exact solution, $\backslash$ for fine-to-coarse and / for coarse-to-fine transfer).

For a formal description of multigrid methods we now use a sequence of coarser and coarser grids Ω_{h_k}, characterized by a sequence of mesh sizes h_k:

$$\Omega_{h_\ell}, \Omega_{h_{\ell-1}}, \cdots, \Omega_{h_0} \quad .$$

The coarsest grid is characterized by the mesh size h_0 (index 0), whereas the index ℓ corresponds to the given finest grid Ω_h: $h = h_\ell$. For simplicity, we replace the index h_k by k (for grids, grid functions and grid operators) in the following. For each Ω_k, we assume that linear operators

$$\begin{aligned} L_k &: \mathcal{G}(\Omega_k) \to \mathcal{G}(\Omega_k), & S_k &: \mathcal{G}(\Omega_k) \to \mathcal{G}(\Omega_k), \\ I_k^{k-1} &: \mathcal{G}(\Omega_k) \to \mathcal{G}(\Omega_{k-1}), & I_{k-1}^{k} &: \mathcal{G}(\Omega_{k-1}) \to \mathcal{G}(\Omega_k) \end{aligned} \tag{4.3.29}$$

are given, where

$$L_k u_k = b_k \quad (\Omega_k) \tag{4.3.30}$$

are discretizations of $\mathcal{L}u = b$ on Ω_k for $k = \ell, \ell-1, \ldots, 0$. The operators S_k denote the linear iteration operators corresponding to given smoothing methods on Ω_k. The calculation of a new iterate u_k^{m+1} from a given approximation u_k^m to the solution u_k proceeds as follows:

<u>**Multigrid cycle**</u> $\mathsf{u}_\mathsf{k}^{\mathsf{m}+1} = \mathrm{MGCYC}(\mathsf{k}, \gamma, \mathsf{u}_\mathsf{k}^\mathsf{m}, \mathsf{L}_\mathsf{k}, \mathsf{b}_\mathsf{k}, \nu_1, \nu_2)$

(1) Pre–smoothing

– Compute $\overline{u}_k^m$ by applying $\nu_1 (\geq 0)$ smoothing steps to u_k^m

$$\overline{u}_\mathsf{k}^\mathsf{m} = \mathrm{SMOOTH}^{\nu_1}(\mathsf{u}_\mathsf{k}^\mathsf{m}, \mathsf{L}_\mathsf{k}, \mathsf{b}_\mathsf{k}) \ .$$

(2) Coarse grid correction

– Compute the defect $\qquad \overline{r}_\mathsf{k}^\mathsf{m} = \mathsf{b}_\mathsf{k} - \mathsf{L}_\mathsf{k}\overline{u}_\mathsf{k}^\mathsf{m}$.

– Restrict the defect $\qquad \overline{r}_{\mathsf{k}-1}^\mathsf{m} = \mathsf{I}_\mathsf{k}^{\mathsf{k}-1}\overline{r}_\mathsf{k}^\mathsf{m}$.

– Compute an approximate solution $\widehat{e}_{k-1}^m$ of the defect equation on Ω_{k-1}

$$\mathsf{L}_{\mathsf{k}-1}\widehat{\mathsf{e}}_{\mathsf{k}-1}^\mathsf{m} = \overline{r}_{\mathsf{k}-1}^\mathsf{m} \tag{4.3.31}$$

by

If $k = 1$, use a direct or fast iterative solver for (4.3.31).
If $k > 1$, solve (4.3.31) approximately by performing $\gamma (\geq 1)$ k-grid cycles using the zero grid function as a first approximation

$$\widehat{\mathsf{e}}_{\mathsf{k}-1}^\mathsf{m} = \mathrm{MGCYC}^{\gamma}(\mathsf{k}-1, \gamma, 0, \mathsf{L}_{\mathsf{k}-1}, \overline{r}_{\mathsf{k}-1}^\mathsf{m}, \nu_1, \nu_2) \ . \tag{4.3.32}$$

– Interpolate the correction $\qquad \widehat{\mathsf{e}}_\mathsf{k}^\mathsf{m} = \mathsf{I}_{\mathsf{k}-1}^\mathsf{k}\widehat{\mathsf{e}}_{\mathsf{k}-1}^\mathsf{m}$.

– Compute the corrected approximation on Ω_k $\qquad \mathsf{u}_\mathsf{k}^{\mathsf{m},\text{after CGC}} = \overline{u}_\mathsf{k}^\mathsf{m} + \widehat{\mathsf{e}}_\mathsf{k}^\mathsf{m}$.

(3) Post–smoothing

– Compute u_k^{m+1} by applying ν_2 (≥ 0) smoothing steps to $u_k^{m,\text{after CGC}}$:

$$\mathsf{u}_\mathsf{k}^{\mathsf{m}+1} = \mathrm{SMOOTH}^{\nu_2}(\mathsf{u}_\mathsf{k}^{\mathsf{m},\text{after CGC}}, \mathsf{L}_\mathsf{k}, \mathsf{b}_\mathsf{k}) \ .$$

In (4.3.32) the parameter γ appears twice: once (as an argument of MGCYC) for the indication of the cycle type to be employed on coarser levels and once (as a power) to specify the number of cycles to be carried out on the current coarse grid level.

Since on coarse grids we deal with *corrections* to the fine grid approximation, this multigrid scheme is also called the *correction scheme* (CS).

4.3.4 Exercises

Exercise 4.3.1 Consider the system of equations $Au = b$ with a symmetric positive definite matrix A. Compare the asymptotic error and defect reduction factors for the ω-JAC iteration

$$u^{m+1} = Qu^m + s \quad (m = 0, 1, 2, \ldots)$$

with

$$Q = I - \omega D^{-1} A \quad \text{and} \quad s = \omega (D)^{-1} b\,,$$

where D is the diagonal of A (cf. (4.3.5)-(4.3.6).

Exercise 4.3.2 Let Ω_ℓ be the grid contained in the d-dimensional cube with grid size $h = 2^{-\ell}$. Suppose that we apply standard coarsening to Ω_ℓ, obtaining Ω_ℓ, $\Omega_{\ell-1}$, $\ldots \Omega_1$. Give an estimate for the sum of the numbers of grid points over all of the grids (for large ℓ).

Exercise 4.3.3

a) Let 1_h and 1_{2h} be grid functions which are equal to 1 everywhere on Ω_h, respectively on Ω_{2h}. Show that for the full weighting, we have

$$I_h^{2h} 1_h = 1_{2h}\,.$$

b) Let $w_h(ih, jh) = j$ respectively $w_{2h}(i2h, j2h) = j$ be functions which increase linearly with j on Ω_h, respectively on Ω_{2h}. Let I_h^{2h} be the full weighting weighting restriction operator. Is

$$I_h^{2h} w_h = w_{2h}\,?$$

Explain your answer.

c) What condition must a general stencil for a restriction operator I_h^{2h}

$$I_h^{2h} = \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ t_{3,1} & t_{3,2} & t_{3,3} \end{bmatrix}_h^{2h}$$

satisfy in order that $I_h^{2h} 1_h = 1_{2h}$?

Exercise 4.3.4 Let 1_h and 1_{2h} be grid functions which are equal to 1 everywhere on Ω_h, respectively on Ω_{2h}. Show that $I_{2h}^h 1_{2h} = 1_h$ if I_{2h}^h is the bilinear interpolation.

Exercise 4.3.5 Let Ω_ℓ be the grid contained in the unit square with grid size $h = 2^{-\ell}$. Suppose that we recursively apply red–black coarsening starting on the $2^\ell \times 2^\ell$ grid Ω_ℓ. Give an estimate for the sum of the numbers of grid points over all of the grids.
Hint: The result of two red–black coarsenings is the same as one standard coarsening.

Exercise 4.3.6 Show that the 1D (and thus also the 2D) FW operator is $O(h^2)$-accurate if applied to a sufficiently smooth function w:

$$I_h^{2h} w = w + O(h^2) \ .$$

Why does this property not impair the *accuracy* of the discrete solution in a multigrid algorithm?

Exercise 4.3.7 Show that the 1D FW operator can be derived from a discrete version of the condition

$$\int_{x-h}^{x+h} w(\xi)d\xi = \int_{x-h}^{x+h} (I_h^{2h} w)(\xi)d\xi \tag{4.3.33}$$

where the midpoint rule is used to approximate the integral on the right-hand side of the equation and the trapezoidal rule is used to approximate the integral on the left-hand side.

Exercise 4.3.8 Show that the 2D FW operator can be derived from a discrete version of the condition

$$\int_{\Omega_{x,y}} w(\tilde{x}, \tilde{y})d\Omega = \int_{\Omega_{x,y}} (I_h^{2h} w)(\tilde{x}, \tilde{y})d\Omega \tag{4.3.34}$$

for $\Omega_{x,y} = [x-h, x+h] \times [y-h, y+h]$ where the midpoint rule is used to approximate the integral on the right-hand side of the equation and the trapezoidal rule is used to approximate the integral on the left-hand side.

Exercise 4.3.9 Let the 2D Poisson equation with Dirichlet boundary conditions be given

$$\begin{aligned} \mathcal{L}u = -\Delta u = b \quad & \text{on } \Omega \\ u = g \quad & \text{on } \Gamma = \partial\Omega. \end{aligned} \tag{4.3.35}$$

with the discretization

$$L_h u_h = \frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h u_h = b_h, \quad h = \frac{1}{n}, \quad n \text{ even}, \quad x_i = ih, \ y_j = jh.$$

(a) Determine the high- and low-frequency components of the eigenfunctions on an Ω_{2h}-grid.

(b) Determine the high- and low-frequency components of the eigenfunctions on an x-semicoarsened grid, i.e. a grid with mesh size $2h$ in x-direction, but h in y-direction (see Fig. 4.3.8).

(c) Determine a corresponding direct coarse grid discretization L_H of the fine grid operator L_h on an x-semicoarsened grid.

(d) Determine the high- and low-frequency components of the eigenfunctions on a red–black coarsened grid.

(e) Determine a corresponding direct coarse grid discretization L_H of the fine grid operator L_h on a red–black coarsened grid.

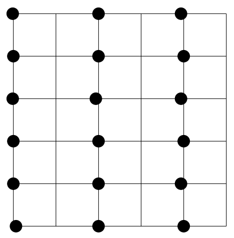


Figure 4.3.8: x-semicoarsening

Exercise 4.3.10 Sketch the structure of one multigrid cycle on four grids for $\gamma = 3$ and $\gamma = 4$.

Exercise 4.3.11 Determine the three-grid operator in the case of standard coarsening.

4.4 Multigrid Convergence and Efficiency

4.4.1 Computational Work

From the recursive definition of a multigrid cycle, it immediately follows that the *computational work* W_ℓ per multigrid cycle Ω_ℓ is recursively given by

$$W_1 = W_1^0 + W_0 \ , \quad W_{k+1} = W_{k+1}^k + \gamma_k W_k \quad (k = 1, \dots, \ell - 1) \ . \tag{4.4.1}$$

Here W_{k+1}^k denotes the computational work of one (h_{k+1}, h_k) two–grid cycle *excluding* the work needed to solve the defect equations on Ω_k, and W_0 denotes the work needed to compute the exact solution on the coarsest grid Ω_0. By "computational work", we denote the number of arithmetic operations needed. If γ is *independent of* k, we obtain from (4.4.1)

$$W_\ell = \sum_{k=1}^{\ell} \gamma^{\ell-k} W_k^{k-1} + \gamma^{\ell-1} W_0 \quad (\ell \geq 1) \ . \tag{4.4.2}$$

Let us discuss *standard coarsening* in 2D with γ independent of k. Obviously, we have in this case

$$N_k \doteq 4 N_{k-1} \quad (k = 1, 2, \dots, \ell) \tag{4.4.3}$$

where $N_k = |\Omega_k|$ (number of grid points on Ω_k) and "$\doteq$" means equality up to lower order terms (boundary effects). Furthermore, we assume that the multigrid components (relaxation, computation of defects, fine–to–coarse and coarse–to–fine transfer) require a number of arithmetic operations per point of the respective grids which is bounded by a constant C, independent of k:

$$W_k^{k-1} \dot{\leq} C N_k \quad (k = 1, 2, \dots, \ell) \ . \tag{4.4.4}$$

("$\dot{\leq}$" means "$\leq$" up to lower order terms.)

Under these assumptions, one obtains the following estimate for the *total computational work* W_ℓ *of one complete multigrid cycle in 2D* from (4.4.2):

$$\boxed{W_\ell \dot{\leq} \begin{cases} \frac{4}{3} C N_\ell & (\text{for } \gamma = 1) \\ 2 C N_\ell & (\text{for } \gamma = 2) \\ 4 C N_\ell & (\text{for } \gamma = 3) \\ O(N_\ell \log N_\ell) & (\text{for } \gamma = 4) \end{cases}} \ . \tag{4.4.5}$$

For $\gamma = 4$ the total computational work on each level is essentially constant (up to lower order terms) and the number of grid levels is $O(\log N_\ell)$.

Summary: This estimate of W_ℓ shows that the number of arithmetic operations needed for one 2D multigrid cycle is proportional to the number of grid points of the finest grid for $\gamma \leq 3$ and standard coarsening (under quite natural assumptions which are satisfied for reasonable multigrid methods). Together with the h–independent convergence, this means that multigrid methods achieve a fixed reduction (independent of h) of the defect in $O(N)$ operations. The constant of proportionality depends on the type of the cycle, i.e. on γ, the type of coarsening and the other multigrid components. For reasonable choices of these components, the constants of proportionality are small.

4.4.2 An Efficient 2D Multigrid Poisson Solver

Here, we introduce the first specific multigrid algorithm. For that purpose we return to Model Problem 1, the discrete Poisson equation with Dirichlet boundary conditions in the 2D unit square. The algorithm presented here is a highly efficient Poisson solver.

The algorithm is characterized by the following multigrid components. In this characterization certain parameters and components still have to be specified:

– $L_k = L_{h_k} = -\Delta_{h_k} = \dfrac{1}{{h_k}^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_{h_k}$,
– smoother: Red- Black Gauss-Seidel relaxation as presented in Section 4.2,
– restriction I_k^{k-1}: Full Weighting (4.3.27) of defects,
– prolongation I_{k-1}^{k}: bilinear interpolation (4.3.28),
– standard coarsening: $h_{k+1} = h_k/2$,
– size of coarsest grid: $h_0 = 1/2$.

In the following we present the influence which the number of relaxations $\nu = \nu_1 + \nu_2$ and the cycle type (V, W) have on the convergence speed of the algorithm and consequently for the *numerical efficiency* of the algorithm.

We use the notation V(ν_1, ν_2), W(ν_1, ν_2) to indicate the cycle type and the number of pre- and post-smoothing iteration steps employed. The finest grid is $h = 1/256$. On this fine grid, we can observe the excellent convergence of multigrid. In Figure 4.4.9 the multigrid convergence of the V(0,1)-cycle, of the V(1,1)-, the W(0,1)- and the W(1,1)-cycle is presented, using FW as the restriction operator. The l_2 norm of the defect is plotted in a log scale along the y-axis. The x-axis shows the number of multigrid iterations.

From Figure 4.4.9, we observe rapid convergence of multigrid, especially for the V(1,1)- and W(1,1)-cycles: they reduce the defect by a factor of 10^{-12} within 12 multigrid iterations. Also the benefits of processing the coarse grid levels

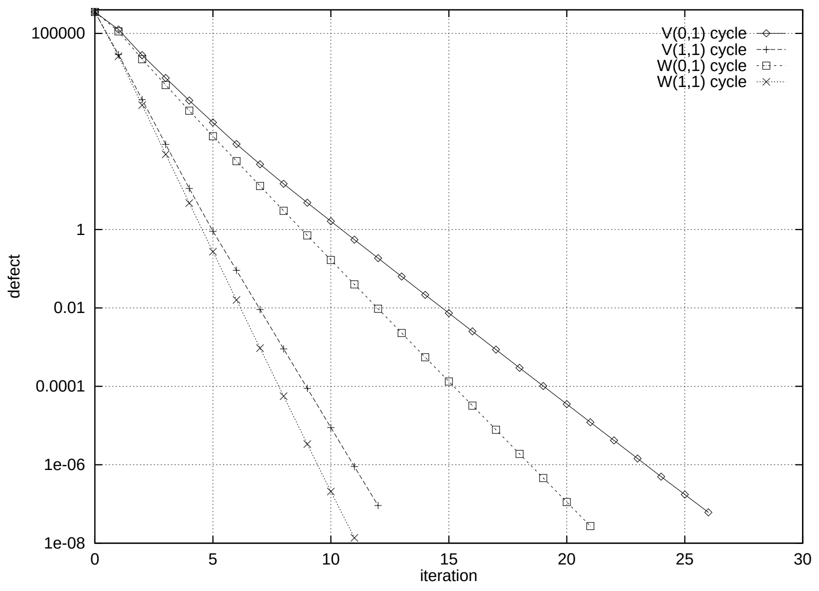


Figure 4.4.9: The convergence of different multigrid cycles for Model Problem 1.

more frequently can be seen: The W-cycle shows a better performance (versus the number of iterations) than the V-cycle.

Remark 4.4.1 In practice, it is usually *not* necessary to reduce the defect by a factor of 10^{-12}. Convergence to discretization accuracy $O(h^2)$ is sufficient in most cases, and of course obtained much faster. Here, we reduce the defect further in order to estimate the *asymptotic* convergence of the multigrid cycle. $\gg$

4.4.3 How to Measure the Multigrid Convergence Factor in Practice

In order to construct, evaluate and analyze a multigrid iteration one often wants to determine its convergence factor ρ empirically (by measurement). In general, the only quantities that are available for the determination of ρ are the defects r_h^m $(m = 1, 2, \ldots)$. We can measure, for example,

$$\tilde{\rho}^{(m)} = \frac{\| r_h^m \|}{\| r_h^{m-1} \|} \tag{4.4.6}$$

or

$$\widehat{\rho}^{(m)} = \sqrt[m]{\tilde{\rho}^{(m)} \tilde{\rho}^{(m-1)} \ldots \tilde{\rho}^{(1)}} = \sqrt[m]{\frac{\| r_h^m \|}{\| r_h^0 \|}} \tag{4.4.7}$$

in some appropriate norm, say the discrete $\| \cdot \|_2$ norm (see Section 2.2.1). The quantity $\widehat{\rho}^{(m)}$ represents an average defect reduction factor over m iterations. For "sufficiently general" $r_h^0 \neq 0$ we have

$$\widehat{\rho}^{(m)} \longrightarrow \rho \ .$$

$\widehat{\rho}^{(m)}$ is a good estimate for ρ if m is sufficiently large. In many cases the *convergence history* is also of interest.

The numerically measured convergence of the Red-Black-Multigrid Poisson Solver is essentially independent of the size of the finest grid in the multigrid cycle. This behavior is demonstrated by the results in Table 4.4.1. This behavior is also predicted by multigrid theory.

Cycle	$h = 1/512$	$h = 1/256$	$h = 1/128$	$h = 1/64$	$h = 1/32$	$h = 1/16$
V(1,1):	0.10	0.10	0.10	0.10	0.10	0.10
F(1,1):	0.063	0.063	0.063	0.063	0.063	0.063
W(1,1):	0.063	0.063	0.063	0.063	0.063	0.063

Table 4.4.1: Measured convergence factors of the Red-Black Multigrid Poisson Solver (with FW) for Model Problem 1 on grids of different mesh size; in each case, the coarsest grid has a mesh size of $h_0 = 1/2$.

The practical relevant measure for the efficiency of a solution method is, of course, the total CPU time it takes to solve the problem of interest. If, we consider the CPU time to solve the problem here, however, the V-cycle is to be preferred !

4.4.4 Exercises

Exercise 4.4.1 Show that the total computational work W_ℓ of one complete multigrid cycle in 2D, with a coarsening such that the number of grid points behaves like

$$N_k \doteq \tau N_{k-1} \quad (k = 1, 2, \ldots, \ell) \text{ with } \tau > 1 \ ,$$

is

$$W_\ell \doteq \begin{cases} \dfrac{\tau}{\tau - \gamma} C N_\ell & (\text{for } \gamma < \tau) \\ O(N_\ell \log N_\ell) & (\text{for } \gamma = \tau) \end{cases} .$$

What happens if $\gamma > \tau$?

Exercise 4.4.2 Show that the total computational work W_ℓ of one complete multigrid cycle in 2D (with constant γ) with standard coarsening is

$$W_\ell \dot{\leq} \begin{cases} \frac{4}{3} C N_\ell & (\text{for } \gamma = 1) \\ 2 C N_\ell & (\text{for } \gamma = 2) \\ 4 C N_\ell & (\text{for } \gamma = 3) \end{cases} ,$$

where "$\dot{\leq}$" means "$\leq$" up to lower order terms.

Exercise 4.4.3 Assume two iterative methods with asymptotic convergence factors $\rho_1 = 0.1$ and $\rho_2 = 0.9$. How many iteration steps of the slower iteration have asymptotically to be performed to reach ar least the same error reduction as one iteration step of the fast iteration?

Exercise 4.4.4 Assume two iterative methods with asymptotic convergence factors $\rho_1 = 0.9$ and $\rho_2 = 0.999$. How many iteration steps have asymptotically to be performed with the slower iteration to reach the same error reduction as one iteration step of the fast iteration?

Exercise 4.4.5 Assume two iterative methods with asymptotic convergence factors $\rho_1 = 0.3$ and $\rho_2 = 0.6$ and that one iteration step with the fast iteration requires 1 sec computing time. How much time may one iteration step of the slow iteration require at most to be competitive with the fast iteration?

4.5 First Generalizations and Multigrid in 3D

4.5.1 Time-Dependent Problems

One way to apply multigrid to time-dependent problems is to use an *implicit (or semi-implicit) time discretization* and to apply multigrid to each of the (discrete) problems that have to be solved in each time step. We explain this approach only for two simple, but representative examples, a parabolic and a hyperbolic one. For both cases, we will discuss the features of the arising discrete problems and the consequences for their multigrid treatment.

Example 4.5.1 (Heat equation:) We consider the parabolic initial boundary value problem

$$\begin{array}{lcll} u_t & = & \Delta u & (\Omega,\ \ t>0) \\ u(x,y,0) & = & u_0(x,y) & (\Omega,\ \ t=0) \\ u(x,y,t) & = & b^{\partial\Omega}(x,y,t) & (\partial\Omega,\ \ t>0) \end{array} \tag{4.5.8}$$

for the function $u = u(x,y,t)$, $(x,y) \in \Omega = (0,1)^2$, $t \geq 0$. The simplest *explicit* discretization is the *forward Euler scheme*

$$\frac{u_{h,\tau}(x,y,t+\tau) - u_{h,\tau}(x,y,t)}{\tau} = \Delta_h u_{h,\tau}(x,y,t) \ \ ,$$

where τ is the step size in t-direction and h the grid size of the space discretization. Δ_h is again the usual 5-point discretization of the Laplace operator.

Obviously, the values of $u_{h,\tau}$ at a new time step $t+\tau$ can be calculated immediately (explicitly) from the values of the previous time step. Such explicit time discretization schemes typically lead to a restrictive "*CFL stability condition*" [30] of the form

$$\tau \leq \text{const}\ h^2 \ \ ,$$

where the constant is 1/2 in our example. Implicit time discretization schemes, on the other hand, are *unconditionally stable* if arranged appropriately. In particular, there is no time step restriction.

We consider three implicit schemes, the *backward Euler scheme*

$$\frac{u_{h,\tau}(x,y,t+\tau) - u_{h,\tau}(x,y,t)}{\tau} = \Delta_h u_{h,\tau}(x,y,t+\tau) \ \ , \tag{4.5.9}$$

the Crank-Nicolson scheme

$$\frac{u_{h,\tau}(x,y,t+\tau) - u_{h,\tau}(x,y,t)}{\tau} = \frac{1}{2}\left(\Delta_h u_{h,\tau}(x,y,t+\tau) + \Delta_h u_{h,\tau}(x,y,t)\right) \tag{4.5.10}$$

and the so-called *backward difference formula BDF(2)* [54]

$$\frac{3u_{h,\tau}(x,y,t+\tau) - 4u_{h,\tau}(x,y,t) + u_{h,\tau}(x,y,t-\tau)}{2\tau} = \Delta_h u_{h,\tau}(x,y,t+\tau) \ \ . \tag{4.5.11}$$

The time discretization accuracy is $O(\tau)$ for the explicit and implicit Euler scheme, $O(\tau^2)$ for the Crank-Nicolson and the BDF(2) scheme. In order to calculate the grid function $u_{h,\tau}(x,y,t+\tau)$ from $u_{h,\tau}(x,y,t)$ (and from $u_{h,\tau}(x,y,t-\tau)$ in case of the BDF(2) scheme), one has to solve problems of the form

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4+\alpha\dfrac{h^2}{\tau} & -1 \\ & -1 & \end{bmatrix} u_{h,\tau}(x,y,t+\tau) = F_{h,\tau}\left(u_{h,\tau}(x,y,t), u_{h,\tau}(x,y,t-\tau)\right) \tag{4.5.12}$$

in each time step, where $F_{h,\tau}\left(u_{h,\tau}(x,y,t), u_{h,\tau}(x,y,t-\tau)\right)$ contains only known values from previous time steps $t, t-\tau$. Here, we have $\alpha = 1$ for backward Euler, $\alpha = 2$ for Crank-Nicolson and $\alpha = 3/2$ for the BDF(2) scheme. In each case, this is obviously a discrete Helmholtz equation corresponding to a Helmholtz constant $c = \alpha/\tau > 0$. The positive contributions on the main diagonal of the corresponding matrix A amplify the diagonal dominance so that smoothing and convergence factors are improved (compared to the Poisson case). Obviously, the smaller the time steps, the stronger is the diagonal dominance and the better are the smoothing and convergence factors. $\triangle$

Hyperbolic time like problems have somewhat different features with respect to the multigrid treatment. Again, we consider only a simple example.

Example 4.5.2 (Wave equation:) We consider the hyperbolic initial boundary value problem

$$\begin{array}{lcll} u_{tt} & = & \Delta u & (\Omega,\ t) \\ u(x,y,0) & = & u_0(x,y) & (\Omega,\ t=0) \\ u_t(x,y,0) & = & u_1(x,y) & (\Omega,\ t=0) \\ u(x,y,t) & = & b^{\partial\Omega}(x,y,t) & (\partial\Omega,\ t) \end{array} \tag{4.5.13}$$

for the function $u = u(x,y,t)$, $t \in \mathbb{R}$, $(x,y) \in \Omega = (0,1)^2$. In such hyperbolic situations, the CFL stability condition for explicit schems is much more moderate. It typically reads

$$\tau \le \text{const}\ h\ .$$

Again, appropriate implicit time discretization schemes are unconditionally stable.

For the wave equation we treat only one common implicit discretization. Similar considerations apply to other implicit schemes. With respect to t, we use the approximation

$$u_{tt} = \frac{1}{\tau^2}\left[\begin{array}{ccc} 1 & -2 & 1 \end{array}\right]_\tau u(\cdot,t) + O(\tau^2)\ ,$$

whereas the following combination of three time levels is used for the space discretization

$$\Delta u = \Delta_h \left(\frac{1}{4} u(x,y,t+\tau) + \frac{1}{2} u(x,y,t) + \frac{1}{4} u(x,y,t-\tau) \right) + O(h^2) + O(\tau^2)) \ . \tag{4.5.14}$$

In this case, the resulting discrete problem that has to be solved in each time step is characterized by a discrete Helmholtz equation of the form

$$\frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 + \dfrac{h^2}{\tau^2} & -1 \\ & -1 & \end{bmatrix} u_{h,\tau}(x,y,t+\tau) = F_{h,\tau}(x,y,t) \ , \tag{4.5.15}$$

which corresponds to the Helmholtz constant $c = h^2/\tau^2 > 0$. Note, that the diagonal dominance reflected by this Helmholtz constant is qualitatively stronger than in the parabolic case: If $\tau = O(h)$ (which is natural in the hyperbolic case) the constant is $c = \text{const}/h^2$ (const > 0) leading to a strong diagonal dominance constant, independent of h. $\triangle$

The considerations in the above examples are typical for more general time-dependent parabolic and hyperbolic problems. If a discretization L_h of an operator $\mathcal{L}$ can be treated efficiently by multigrid, this holds also for implicit discretizations of the problems $u_t = -\mathcal{L}u$ and $u_{tt} = -\mathcal{L}u$. The multigrid method for L_h can also be used for the arising discrete problems per time step. If the time steps become very small, even simple relaxation methods may have good convergence properties and may be competitive to (or even more efficient than) multigrid. Multigrid can be applied without difficulty to the 3D Poisson equation. It has essentially the same properties (complexity) for 3D as for 2D problems. The 2D multigrid ideas and components introduced in Sections 4.2 4.3 and 4.3.3 can be generalized directly to the 3D Poisson equation, the 3D analog of Model Problem 1 (2.1.9):

Model Problem 2

$$\begin{aligned} -\Delta_h u_h(x,y,z) &= b_h^\Omega(x,y,z) \quad \Big((x,y,z) \in \Omega_h \Big) \\ u_h(x,y,z) &= b_h^\Gamma(x,y,z) \quad \Big((x,y,z) \in \Gamma_h = \partial\Omega_h \Big) \end{aligned} \tag{4.5.16}$$

in the unit cube $\Omega = (0,1)^3 \subset \mathbb{R}^3$ *with* $h = 1/n,\ n \in \mathbb{N}$.

In 3D the infinite grid $\mathbf{G}_h$ is defined by

$$\boxed{\mathbf{G}_h := \{(x,y,z) : x = ih_x, y = jh_y, z = kh_z;\ i,j,k \in \mathbb{Z}\}} \ , \tag{4.5.17}$$

where $\boldsymbol{h} = (h_x, h_y, h_z)$ is a vector of fixed mesh sizes. In the case of cubic Cartesian grids, we simply identify $h = h_x = h_y = h_z$. Δ_h denotes the standard 7–point $O(h^2)$–approximation of the 3D Laplace operator Δ

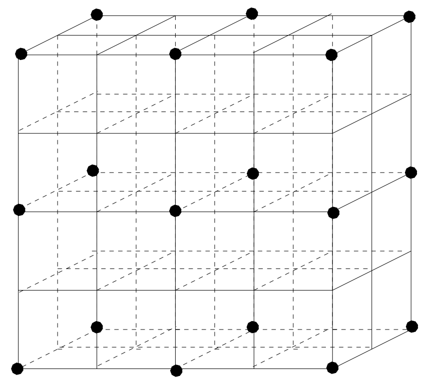
$$\begin{aligned} L_h u_h(x,y,z) &= -\Delta_h u_h(x,y,z) \\ &= \frac{1}{h^2}\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_h u_h(x,y,z-h) \\ &+ \frac{1}{h^2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 6 & -1 \\ 0 & -1 & 0 \end{bmatrix}_h u_h(x,y,z) \\ &+ \frac{1}{h^2}\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_h u_h(x,y,z+h)\ , \end{aligned} \qquad (4.5.18)$$

where the 2D stencils are applied to the x- and y-coordinates as in 2D. Introducing the *3D stencil notation*, we write in short

$$-\Delta_h u_h = \frac{1}{h^2}\left[\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_h \begin{bmatrix} 0 & -1 & 0 \\ -1 & 6 & -1 \\ 0 & -1 & 0 \end{bmatrix}_h \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_h\right] u_h\ .$$

4.5.2 3D Multigrid Components

Standard coarsening is as in 2D ($H = 2h$). Figure 4.5.10 shows a part of a fine grid with the coarse grid points.



● : coarse grid point

Figure 4.5.10: A (vertex-centered) 3D fine grid with a coarse grid coming from standard coarsening.

For the transfer operators, we obtain the following generalizations of the 2D operators. The 3D *Full Weighting operator* (FW) is given by

$$\frac{1}{64}\left[\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}_h^{2h}\begin{bmatrix}2&4&2\\4&8&4\\2&4&2\end{bmatrix}_h^{2h}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}_h^{2h}\right]\ . \tag{4.5.19}$$

The generalization of the 2D *bilinear interpolation operator* introduced in (4.3.28) to 3D is *trilinear interpolation.* Along planes that contain coarse grid points the formulae from (4.3.28) are still valid (with an additional index for the respective third dimension). One extra formula is necessary for the trilinear interpolation to the fine grid points not belonging to any plane of the coarse grid (those with eight coarse grid neighbors).

$$\begin{aligned}\widehat{e}_h(x,y,z) &= I_{2h}^h\widehat{e}_{2h}(x,y,z)\\ &= \frac{1}{8}\Big[\widehat{e}_{2h}(x+h,y+h,z+h)+\widehat{e}_{2h}(x+h,y+h,z-h)\\ &\quad+\widehat{e}_{2h}(x+h,y-h,z+h)+\widehat{e}_{2h}(x+h,y-h,z-h)\\ &\quad+\widehat{e}_{2h}(x-h,y+h,z+h)+\widehat{e}_{2h}(x-h,y+h,z-h)\\ &\quad+\widehat{e}_{2h}(x-h,y-h,z+h)+\widehat{e}_{2h}(x-h,y-h,z-h)\Big]\ .\end{aligned} \tag{4.5.20}$$

The *smoothers* ω–JAC, GS–LEX and GS–RB as described for Model Problem 1 can be generalized immediately to 3D, too. It is also straightforward to determine the *smoothing factor* (4.2.8) of ω–JAC for Model Problem 2. First of all, the discrete eigenfunctions of the discrete 3D operator Δ_h are

$$\begin{gathered}\varphi_h^{k,\ell,m}(x,y)=\sin k\pi x\sin l\pi y\sin m\pi z\\ \Big((x,y,z)\in\Omega_h,\ \ (k,\ell,m=1,\ldots,n-1)\Big)\ .\end{gathered} \tag{4.5.21}$$

The corresponding eigenvalues of the ω–JAC relaxation operator are

$$\lambda_{k,\ell}=\lambda_{k,\ell}(\omega)\ =\ 1-\frac{\omega}{3}(3-\cos k\pi h-\cos\ell\pi h-\cos m\pi h)\ . \tag{4.5.22}$$

For standard coarsening, the low frequency components are the $\varphi_h^{k,\ell,m}$ with $\max(k,\ell,m)<n/2$, high frequency components those with $n/2\le\max(k,\ell,m)<n$. Thus, the smoothing factor $\mu(h;\omega)$ of S_h and its supremum μ^* over h can be defined by

$$\begin{aligned}\mu(h;\omega)\ &:=\ \max\{|\lambda_{k,\ell,m}(\omega)|:n/2\le\max(k,\ell,m)\le n-1\}\ ,\\ \mu^*(\omega)\ &:=\ \sup_{h\in\mathcal{H}}\mu(h;\omega)\ .\end{aligned} \tag{4.5.23}$$

It is straightforward to find the smoothing factor of ω–JAC for Model Problem 2 as

$$\mu^*(\omega)\ =\ \max\{|1-\omega/3|,|1-2\omega|\}\ .$$

Optimal smoothing is obtained for $\omega = 6/7$ (≈ 0.857) for which we find the smoothing factor $\mu^* = 5/7$ (≈ 0.714).

It is a general observation, that the smoothing factors, also for *GS–LEX* and *GS–RB*, in 3D are not identical to the 2D case, but somewhat worse. For example, the smoothing factors of GS-RB and GS-LEX increase from 0.25 and 0.5 in 2D to 0.445 and 0.567 in 3D.

Computational work The *computational work* W_ℓ per 3D multigrid cycle Ω_ℓ is recursively given by (4.4.1) as in 2D. In case of *3D standard coarsening* with fixed cycle index γ we have

$$N_k \doteq 8N_{k-1} \quad (k = 1, 2, \ldots, \ell) \tag{4.5.24}$$

where $N_k = |\Omega_k|$ (number of grid points on Ω_k) and "$\doteq$" again means equality up to lower order terms (boundary effects). Under the same assumptions needed to estimate the work for the 2D multigrid cycle in Section 4.4.1, one immediately obtains the following estimate for the *total computational work* W_ℓ *of one complete multigrid cycle in 3D*,

$$\boxed{W_\ell \dot{\leq} \begin{cases} \frac{8}{7}CN_\ell & (\text{for } \gamma = 1) \\ \frac{4}{3}CN_\ell & (\text{for } \gamma = 2) \\ \frac{8}{5}CN_\ell & (\text{for } \gamma = 3) \\ \end{cases}} \;. \tag{4.5.25}$$

This estimate of W_ℓ together with the h–independent convergence shows the asymptotic optimality of iterative 3D multigrid methods. For standard coarsening, W(γ)-cycles remain asymptotically optimal up to $\gamma \leq 7$.

4.6 Multigrid as a Preconditioner

We discuss the question whether multigrid should be used as a solver or as a preconditioner. In particular, the question to be answered is which approach should be used when.

First of all, it is not useful to accelerate a highly efficient multigrid algorithm (with an ideal convergence factor) by Krylov subspace acceleration: The extra effort does not pay off.

Multigrid as a preconditioner is particularly interesting with respect to *robustness*. An argument for combining multigrid with an acceleration technique is that problems become more and more complex if we treat real life applications. For such complicated applications, it is far from trivial to choose optimal multigrid components uniformly for a large class of problems. Often different complications such as convection dominance, anisotropies, nonlinearities or strong positive off-diagonal stencil elements occur simultaneously.

Therefore, the fundamental idea of multigrid, to reduce the high frequency components of the error by smoothing procedures and to take care of the low frequency error components by coarse grid corrections, does not work optimally in all cases if straightforward multigrid approaches are used. Certain error components may remain large since they cannot be reduced by *standard* smoothing procedures combined with standard coarse grid approximations. These specific error components (and the corresponding eigenvectors/eigenvalues) are then responsible for the poor multigrid convergence. In such situations, the combination with Krylov subspace methods may have the potential to give a substantial acceleration.

Often, more sophisticated multigrid components may lead to very satisfactory convergence factors. But they are more involved to realize and implement. In other cases, the efficiency of the multigrid components may strongly depend on problem parameters. Then, the combination of multigrid with Krylov subspace may have advantages.

Multigrid as a preconditioner is also interesting for several other problems, for example, on unstructured grids, for problems with small geometric details, which are not visible on coarse grids [25], and for the problems with geometric singularities.

Remark 4.6.1 (Multigrid as a preconditioner) In the same way as the classical single grid iterative methods can be used as preconditioners, it is also possible to use multigrid as a preconditioner: We choose $M^{-1} = (I - Q)A^{-1}$ where Q is the multigrid iteration operator. $\gg$

For s.p.d. problems, the robustness of multigrid as a preconditioner for CG has been presented in [87]. In that situation, one has to take care of the symmetrization of the multigrid preconditioner. Typically, the transfer operators, restriction and prolongation, are the transpose of each other (full weighting is the transpose of bilinear interpolation), which is the basis for a symmetric multigrid iteration. Important is further that also the smoother is symmetric. For example, forward lexicographical Gauss-Seidel is used as the pre-smoother, whereas backward lexicographical Gauss-Seidel is the post-smoother.

By using multigrid as a preconditioner for GMRES or BiCGSTAB [139], also nonsymmetric problems like the convection-diffusion equation can be handled. This has been demonstrated, for example, in [103].

Remark 4.6.2 With a *varying* preconditioner, like multigrid with a different cycle in each iteration, a Krylov subspace method in which the preconditioner can change from iteration to iteration is needed. The flexible GMRES method (FGMRES) [121], or the GMRESR method [141] allows such a varying preconditioner.

FMGRES stores defects and approximations of the solution. $\gg$

4.6.1 Exercises

Exercise 4.6.1 Determine for the initial value problem $u' = b(u) = qu, \quad u(0) = 1, q \in \mathbb{R}$ the approximations to the solution, with the forward (explicit) Euler scheme

$$u_h^{j+1} = u_h^j + hb(x_j, u_h^j), \quad x_{j+1} = x_j + h,$$

and with the implicit Euler scheme

$$u_h^{j+1} = u_h^j + hb(x_j, u_h^{j+1}), \quad x_{j+1} = x_j + h.$$

Under what conditions on the mesh size h do the resulting discrete approximations qualitatively resemble the exact solution $u(x) = e^{qx}, 0 \leq x \leq 1$?

Hint: Derive relations of the form $u^{j+1} = \lambda u^j$, that lead to solutions of the form $u_h^{j+1} = c\lambda^j$.

Exercise 4.6.2 Consider the PDE,

$$\mathcal{L}u(x,y,t) = u_t(x,y,t) - \Delta u(x,y,t) = 0.$$

Show that the time discretization accuracy is $O(\tau)$ for the forward and backward (implicit) Euler scheme, $O(\tau^2)$ for the Crank-Nicolson and the BDF(2) scheme.

Exercise 4.6.3 Let $\mathcal{L}u(x,y) = x^2 u_{xx}(x,y) - (1+y^2)u_{yy}$.

a) What is the discrete version, in stencil form, of the PDE $\mathcal{L}u = b$ at the point (x_i, y_j)? Ignore boundary conditions and use a mesh size of h.

b) What is the stencil at the point $(x_i, y_j) = (0,0)$? Is the operator elliptic there?

Exercise 4.6.4 Consider the PDE $u_t = \Delta u$ with suitable initial and boundary conditions. The equation

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4+\alpha(h^2/\tau) & -1 \\ & -1 & \end{bmatrix} u_{h,\tau}(x,y,t+\tau) = F_{h,\tau}\left(u_{h,\tau}(x,y,t), u_{h,\tau}(x,y,t-\tau)\right)$$

gives a general form of the discrete spatial equation to be solved at each time step. Determine the function $F_{h,\tau}(x,y,t)$, $u_{h,\tau}(x,y,t-\tau)$ for each of the following discretizations

a) backward Euler scheme

b) Crank-Nicolson scheme

c) backward difference formula

Exercise 4.6.5 Show that the computational work W_ℓ of one complete 3D multigrid cycle with point relaxation and standard coarsending is

$$W_\ell \dot{\leq} \begin{cases} \frac{8}{7} C N_\ell & \text{for } \gamma = 1 \\ \frac{4}{3} C N_\ell & \text{for } \gamma = 2 \\ \frac{8}{5} C N_\ell & \text{for } \gamma = 3 \end{cases} .$$

Exercise 4.6.6 (a) Discretize the discrete Poisson equation in 3D and determine the resulting matrix for an $n \times n \times n$ grid. Use a numbering, so that grid points (x_i, y_j, z_k) are processed as follows. First, the index i runs, then j und finally k.
Eliminate the boundary conditions.

(b) How many nonzero elements does this matrix contain in each row ?

(c) What is the bandwidth on the regular $n \times n \times n$ grid ?

5 Application: Molecular Dynamics Simulations

5.1 Introduction

Molecular dynamics simulation is a technique to study the behavior of a classical many body system. In this context "classical" refers to the fact that the constituent particles move according to the laws of classical mechanics. This is an excellent approximation for most materials. Consequently the technique is widely used to study the structural, thermodynamic and transport properties of solids and liquids.
Molecular dynamics simulations are in many ways similar to a real experiment. As in a real experiment we first prepare a sample, i.e. we select a model system consisting of N particles and we solve Newton's equations of motion for this system until its properties no longer change with time (the system is equilibrated). Subsequently we perform the actual measurements of the quantities of interest.
To measure an observable quantity in a Molecular Dynamics simulation we require an expression of the observable in terms of the coordinates and velocities of the particles in the system. As an example, the temperature in a classical many-body system can be obtained from the equipartition theorem, which states that in a classical system the average kinetic energy per degree of freedom is given by:

$$\left\langle \frac{1}{2} m v_\alpha^2 \right\rangle = \frac{1}{2} k_B T. \tag{5.1.1}$$

Here k_B is Boltzmann's constant, $k_B = 1.38x10^{-23}$. In a Molecular Dynamics simulation we can use this to define an instantaneous temperature $T(t)$ by dividing the total kinetic energy by the number of degrees of freedom f (=$3N-3$) for a 3 dimensional system of N particles with fixed total momentum $\vec{P}$:

$$T(t) = \sum_{i=1}^{N} \frac{m_i v_i^2}{k_B f}. \tag{5.1.2}$$

As the total kinetic energy of the system fluctuates, so does the instantaneous temperature. The relative fluctuations will be of order $1/\sqrt{f}$. Typically $f \sim 1000$, so that statistical fluctuation will be $5-10\%$. To get an accurate estimate of the temperature an average over many fluctuations is required.

5.2 Boundary Conditions

Molecular Dynamics (and, for that matter Monte Carlo) simulations of atomic and molecular systems aim to provide information about the properties of a macroscopic sample. Most simulations are carried out for a few hundred to a few thousand particles, although simulations of 10^9 atoms have been reported. In such small systems one cannot assume that the choice of boundary conditions has no

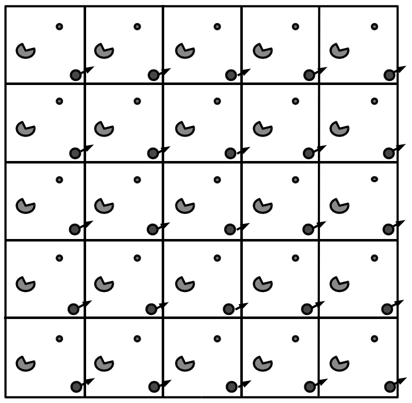


Figure 5.2.1: Periodic boundary conditions in 2 dimensions. When a particle leaves the central computational box its image enters the box from the other side.

effect on the systems behavior, as is the case for macroscopic systems. With free boundary conditions the fraction of atoms at the surface is proportional to $N^{-\frac{1}{3}}$. For example, a simple cubic crystal containing as much as one million atoms still has 6% of its atoms on the surface, for many purposes a non-negligible fraction. To simulate the bulk behavior one has to choose boundary conditions that mimic the presence of an infinite bulk surrounding our N particle system. This is usually achieved by employing periodic boundary conditions. Thus the Molecular Dynamics cell containing N particles is surrounded on all sides by replicas of itself, so that in fact we simulate an infinite periodic lattice of identical cells. A particle, say particle i, now interacts with all other particles in the infinite periodic system, i.e. all other particles in the cell and all particles (including its own image) in the periodic replicas. Suppose we have pairwise additive interactions $u(r)$ between the molecules, then the total potential energy of the N particles in any one periodic box is:

$$U = \frac{1}{2} \sum_{i,j,\mathbf{n}}{}' u\left(|\mathbf{r}_{ij} + \mathbf{n}L|\right). \tag{5.2.1}$$

Here, L is the side of the periodic Molecular Dynamics box (assumed cubic for convenience) and $\mathbf{n} = (\nu_\mathbf{1}, \nu_\mathbf{2}, \nu_\mathbf{3})$, the ν's being arbitrary integers. The factor $\frac{1}{2}$ avoids double counting of the potential energy and the prime on the summation sign indicates that for $\mathbf{n} = \mathbf{0}$ the term $i = j$ has to be omitted. Fortunately, we generally deal with so-called "short range" interactions, which decay very rapidly with increasing inter-atomic distance. In that case, it is permitted to truncate all intermolecular interactions beyond a certain cut-off distance r_c, i.e. $u(r) = 0$ for $r > r_c$. If we choose $r_c < L/2$, at most one term in the sum over $\mathbf{n}$ survives

for each pair i, j, the nearest image.
Notable exceptions to this are the "long range" interactions, which include the important cases of the Coulomb and dipolar interactions, falling of as, respectively, r^{-1} and r^{-3} in 3 dimensions. The sum in (5.2.1) is then generally semi-convergent and special techniques are required to deal with these situations. These cases will be treated later on.

5.3 Molecular Dynamics

The principle of the molecular dynamics technique is straightforward. Given a potential function $U(\mathbf{r}_1, \cdots, \mathbf{r}_N)$ for a set of N interacting particles with positions $(\mathbf{r}_1, \cdots, \mathbf{r}_N)$ the classical equations of motion are solved numerically on a computer. Typically $N \approx 10^2 - 10^6$. The solution of these equations yields the trajectories $(\mathbf{p}_1(t), \cdots, \mathbf{p}_N(t), \mathbf{r}_1(t), \cdots, \mathbf{r}_N(t))$ of the particles in phase space.
We shall first write down the classical equations of motion. We start by considering the simplest possible case, namely N classical structureless particles with positions $\mathbf{r}_1, \cdots, \mathbf{r}_N$ in a volume V interacting through a potential function, which is a sum of pair-potentials:

$$U(\mathbf{r}_1, \cdots, \mathbf{r}_N) = \sum_{i>j} u(r_{ij}), \tag{5.3.1}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the separation between particles i and j. There is no external field. The force $\mathbf{F}_i$ acting on particle i is then:

$$\mathbf{F}_i = -\frac{\partial U}{\partial \mathbf{r}_i} = \sum_{j \neq i} \mathbf{f}_{ij}. \tag{5.3.2}$$

Here $\mathbf{f}_{ij}$ is the force on particle i due to its interaction with particle j:

$$\mathbf{f}_{ij} = -\frac{\partial u(r_{ij})}{\partial \mathbf{r}_{ij}} = -\frac{du(r_{ij})}{dr_{ij}} \frac{\mathbf{r}_{ij}}{r_{ij}}. \tag{5.3.3}$$

The Newtonian equations of motion are then:

$$m_i \ddot{\mathbf{r}}_i = \mathbf{F}_i. \tag{5.3.4}$$

Given initial positions $\mathbf{r}_i(0)$ and velocities $\dot{\mathbf{r}}_i(0)$ for $i = 1, \cdots, N$ the set of coupled ordinary second order differential equations (5.3.4) can be solved to yield the positions $\mathbf{r}_i(t)$ and velocities $\dot{\mathbf{r}}_i(t)$ at later times t. In a molecular dynamics simulation this is done numerically on a computer for a set of typically $N \approx 1000$ particles. To mimic the bulk material we introduce periodic boundary conditions for the reasons discussed above. Thus, the particles are put in a cubic box, which on all sides is surrounded by exact replicas of itself. If a particle leaves the box on the right hand side its image enters the box on the left hand side and so on.

For the sums in (5.3.1) and (5.3.2) the minimum image convention is assumed.

For definiteness we shall illustrate the *MD* procedure with the Lennard-Jones system, for which the interaction potential reads:

$$\phi(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]. \tag{5.3.5}$$

This potential gives a reasonable representation of intermolecular interactions in noble gases, such as Ar and Kr, and systems composed of almost spherical molecules, such as CH_4 and $C(CH_3)_4$. To get an idea of magnitudes, for Ar values are $\epsilon/k_B = 120K$ and $\sigma = 0.34nm$
The potential energy of the system is then:

$$U(\mathbf{r}_1, \cdots, \mathbf{r}_N) = 4\epsilon \sum_{i=1}^{N-1} \sum_{j>i}^{N} \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} \right]. \tag{5.3.6}$$

The equations of motions read:

$$m\ddot{\mathbf{r}}_i = 48\epsilon \sum_{i\neq j}^{N} \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{6} \right] \frac{\mathbf{r}_{ij}}{r_{ij}^2}.$$

At this stage it is convenient to introduce *reduced units*. We measure length in units of σ, energy in units of ϵ, mass in units of m and time in units of τ, with $\tau^2 = m\sigma^2/\epsilon$. Typically, τ is of the order of 1 psec. Thus:

$$\mathbf{r} \;\rightarrow\; \frac{\mathbf{r}}{\sigma},\; E \rightarrow \frac{E}{\epsilon},\; t \rightarrow \frac{t}{\tau}.$$

Dimensionless units of temperature T and density $\rho = N/V$ are:

$$T^* \;=\; \frac{k_B T}{\epsilon},\; \rho^* = \rho\sigma^3. \tag{5.3.7}$$

In these units the energy of the system becomes:

$$u(\mathbf{r}_1, \cdots, \mathbf{r}_N) = 4 \sum_{i=1}^{N-1} \sum_{j>i}^{N} \left({r_{ij}}^{-12} - {r_{ij}}^{-6} \right), \tag{5.3.8}$$

and the equations of motion reduce to:

$$m\ddot{\mathbf{r}}_i = 48 \sum_{i\neq j}^{N} \left({r_{ij}}^{-14} - \frac{1}{2}{r_{ij}}^{-8} \right) \mathbf{r}_{ij}. \tag{5.3.9}$$

step 1	initialize: $\{\mathbf{r}_i, \dot{\mathbf{r}}_i\}$
	do i = 1, no. of time-steps **MD-loop**
step 2	call force (compute forces)
step 3	call advance (advance one time-step δt)
step 4	collect data
	enddo **MD-loop**
step 5	analysis results

Table 5.3.1: Schematic structure of an *MD* program

In this form the equations of motion have a simple structure. In addition, a theorem of corresponding states holds. Finally, most quantities appearing in these equations are of the order of unity, which is advantageous from a computational point of view, since then computations can be done with a high accuracy. Hereafter we shall always use reduced units for the Lennard-Jones system.

Schematically the structure of an *MD* program is as shown in Table 5.3.1 .

We shall discuss each of the steps in the table, using a Lennard-Jones system as an example.

5.3.1 Initialization

We start the algorithm by reading in the parameters determining the state of the system, such as the density ρ and the temperature T. Then, we assign positions and velocities to the particles. This can be done in several ways.

- *Start from lattice positions*
  This is the simplest way. For example, the Lennard-Jonesium crystallizes in a face-centered cubic lattice. As there are 4 atoms per unit cell the MD sample has $N = 4n^3$ atoms. Hence the numbers 108, 256, 864 etc., often occur in *MD* simulations of Lennard-Jones systems.
  The particles are then assigned random velocities, either uniformly distributed or drawn from a Gaussian distribution. The linear momentum $\mathbf{P} = \sum \mathbf{v}_i$ is set to zero. The velocities are scaled to the required temperature. Thus, we determine:

$$T_i = \frac{\sum_{i=1}^{N} {v_i}^2}{dN - d}, \tag{5.3.10}$$

  and multiply all velocities by a scale factor $\sqrt{T_r/T_i}$, where T_r is the required temperature.

- *Start from a previously stored configuration*
  If a number *MD* simulations already have been carried out we can use the

final configuration of a previous run, stored on disc or tape. If necessary we can scale the velocities to the required temperature, using the same technique as described above.

An *MD* program usually has a set of options for different starting configurations.

A Fortran code for the second part of the algorithm, the *force calculation*, is presented in Appendix A.0.1.

5.3.2 Advance Particle Positions

A suitable algorithm is now used to integrate the equations of motion and advance the particle coordinates and momenta to their values at time-step $t + \delta t$. Typically $\delta t \approx 0.005$. Several algorithms exist. The choice of algorithm is influenced by considerations of efficiency and stability. For example, the force calculation is often the most time-consuming step in an *MD* simulation. It is then unlikely that Runge-Kutta algorithms, which require several force evaluations per time-step, are efficient.

One of the most used algorithms is the so-called *velocity Verlet* algorithm [170]. This algorithm is one of a larger class of *Verlet* algorithms [172], that are widely used in molecular dynamics simulations because of their relative simplicity and stability. The velocity Verlet algorithm is known to be reversible, i.e., if we reverse the velocities at a certain time the particles will retrace their trajectories. This automatically implies that the algorithm is absolutely stable. The positions and velocities are advanced according to:

$$\mathbf{r}_i(t+\delta t) = \mathbf{r}_i(t) + (\delta t)\mathbf{v}_i(t) + \frac{\delta t^2}{2m_i}\mathbf{F}_i(t), \tag{5.3.11}$$

$$\mathbf{v}_i(t+\delta t) = \mathbf{v}_i(t) + 0.5(\delta t)\frac{\mathbf{F}_i(t) + \mathbf{F}_i(t+\delta t)}{m_i}. \tag{5.3.12}$$

In order to advance the velocities we require the force at time $t + \delta t$. Therefore, one first advances the particle positions to time $t + \delta t$ and the velocities only '*half a*' time step. After this, one computes the forces at $t + \delta t$ and finally the advancement of the velocities is completed. A simple piece of code, implementing the velocity Verlet scheme for a Lennard-Jones system, is given below.

```
vkin = 0.0
dtsq = 0.5*deltat*deltat
dth = 0.5*deltat


do i = 1,n
advance positions one full time step
   x(i) =x(i)+deltat*vx(i)+dtsq*fx(i)
   y(i) =y(i)+deltat*vy(i)+dtsq*fy(i)
   z(i) =z(i)+deltat*vz(i)+dtsq*fz(i)
advance velocities one half time step
   vx(i) = vx(i)+dth*fx(i)
   vy(i) = vy(i)+dth*fy(i)
   vz(i) = vz(i)+dth*fz(i)
enddo
calculate force at new positions
call force
vkin=0.0
advance velocities to full time step
do i=1,n
   vx(i) = vx(i)+dth*fx(i)
   vy(i) = vy(i)+dth*fy(i)
   vz(i) = vz(i)+dth*fz(i)

accumulate kinetic energy
   vkin = vkin + vx(i)**2 + vy(i)**2 + vz(i)**2
enddo


vkin = 0.5*vkin
vtot = vpot + vkin
```

Note that the force $\mathbf{F}$ at times t and $t + \delta t$ are both denoted by fx, fy, fz to show that only one array is needed for the forces. In the force calculation at the new positions the old forces can be overwritten.

In an elegant paper Tuckerman et. al. [171] have shown how to derive time-reversible integrators of the equations of motion for classical systems, such as the velocity Verlet algorithm, in a systematic manner.

Another class of algorithms based on *predictor-corrector* schemes has been proposed by Gear [54]. These algorithms were used extensively in the early days of computer simulation and offer some advantages in special cases. However these algorithms are not time reversible and not absolutely stable. Nowadays these algorithms are rarely used.

5.3.3 Equilibration

Averages of thermodynamic quantities obtained in an *MD* simulation do not depend on the initial conditions if the simulation is carried out for a sufficiently long time. Nevertheless the starting conditions are usually far from representative for the typical equilibrium conditions of the system. Therefore one usually divides the simulation into two parts.

- *Equilibration*
  The initial part of a simulation is usually referred to as "equilibration". A suitable starting configuration is chosen and initial velocities are assigned. The *MD*-loop is carried out. If the system is far from equilibrium, potential energy will be converted into kinetic energy or vice-versa. Therefore, the temperature changes. If required, the velocities are rescaled to the correct temperature. This is carried on until the system is stationary. Although various quantities fluctuate, no systematic drift should be observed. The final configuration is stored. The various accumulators for structural and thermodynamic quantities are discarded.

- *Production*
  The final positions and velocities (and, when the *predictor-corrector* technique is used, other time-derivatives as well) are used to start the simulation. The *MD*-loop is repeated the required number of times. In an *NVE*-simulation (where the energy E is conserved and the number of particles and volume V are fixed in the simulation) the velocities are no longer rescaled. As the simulation proceeds values of thermodynamic, structural and dynamical quantities are accumulated for averaging afterwards. It should be noted that the production run often consists of a number of separate simulations, each starting where the previous simulation finished.

5.4 Thermodynamic and Structural Quantities

The calculation of equation of state data from an *MD* simulation is straightforward. Here we illustrate this for the Lennard-Jones system. We denote time-averages by $< \cdots >$. The following data are routinely calculated, as follows.

- *Total energy E:*

$$E = 4 \sum_{i=1}^{N-1} \sum_{j>i}^{N} \left({r_{ij}}^{-12} - {r_{ij}}^{-6} \right) + \frac{1}{2} \sum_{i=1}^{N} \mathbf{v}_i^2. \qquad (5.4.13)$$

Obviously the total energy is constant (apart from small errors due to integration algorithm and round-off errors), so that averaging is not required.

- *Temperature T:*

$$< T^* > = \frac{1}{3N-3} < \sum_{i=1}^{N} \mathbf{v}_i^2 > . \tag{5.4.14}$$

- *Pressure P:* The pressure P is calculated from the viral theorem:

$$P\Omega = Nk_BT - \frac{1}{3} < \sum_{i=1}^{N-1} \sum_{j>i}^{N} \mathbf{r}_{ij} \cdot \frac{\partial\phi(\mathbf{r}_{ij})}{\partial\mathbf{r}_{ij}} > . \tag{5.4.15}$$

  For a Lennard-Jones potential this reduces to:

$$\frac{P\Omega}{Nk_BT} = 1 + \frac{8}{N < T^* >} < \sum_{i=1}^{N-1} \sum_{j>i}^{N} \left(2{r_{ij}}^{-12} - {r_{ij}}^{-6}\right) > . \tag{5.4.16}$$

  Note that in reduced units the unit of pressure is ϵ/σ^3.

- *Specific heat;* The specific heat is calculated from the fluctuations in the kinetic energy K. The well-known fluctuation [164] formula reads:

$$\frac{< \delta K^2 >}{< K >^2} = \frac{2}{3N}\left(1 - \frac{3N}{2C_V}\right) , \tag{5.4.17}$$

  where $< \delta K^2 > = < K^2 > - < K >^2$. Note that, since the total energy is conserved, the fluctuations in potential energy V should equal the fluctuations in kinetic energy. Thus $< \delta V^2 > = < \delta K^2 >$.

Structural properties of a liquid are usually discussed in terms of the radial distribution function $g(r)$, defined by:

$$\begin{aligned} g(r) &= \frac{1}{\rho^2} < \sum_{i=1}^{N} \sum_{j\neq i}^{N} \delta(\mathbf{r}_i)\delta(\mathbf{r} - \mathbf{r}_j) > \\ &= \frac{\Omega}{N^2} < \sum_{i=1}^{N} \sum_{j\neq i}^{N} \delta(\mathbf{r} - \mathbf{r}_{ij}) > . \end{aligned} \tag{5.4.18}$$

Essentially $g(r)$ gives the ratio of the average number of particles at a distance r from a particle at the origin, compared to the average number in an ideal gas at the same density.
The radial distribution function is an important physical quantity for several reasons. Firstly, it provides insight in the local structure of a liquid. Furthermore,

ensemble-averages of pair-functions can be expressed in terms of integrals over the radial distribution function. For example, for a system interacting through a pairwise additive potential $\phi(r)$ the energy can be written as:

$$E = \frac{3}{2}Nk_BT + 2\pi N\rho \int_0^\infty r^2 g(r)\phi(r)dr, \tag{5.4.19}$$

and the pressure as:

$$P\Omega = Nk_BT + \frac{2\pi N\rho}{3} \int_0^\infty r^3 g(r)\frac{d\phi(r)}{dr}dr, \tag{5.4.20}$$

so that knowledge of g(r) is sufficient to compute the equation of state data. Finally, the radial distribution function can be measured experimentally through X-ray or elastic neutron scattering experiments. The observed angular dependence of the intensity $I(\theta)$ is a measure of the *static structure factor* $S(k)$.

$$\begin{aligned} S(k) &= 1 + \rho \int (g(r) - 1) \exp(i\mathbf{k}\cdot\mathbf{r})d\mathbf{r} \\ &= 1 + 4\pi\rho \int_0^\infty r^2 \left(g(r) - 1\right) \frac{\sin(kr)}{kr}dr. \end{aligned} \tag{5.4.21}$$

By inverting the Fourier transform, $g(r)$ can be obtained.
To calculate $g(r)$ a histogram is made of pair-separations in a simulation. This is implemented in the following code:

```
do i = 1, N-1
   do j = i+1, N
compute minimum image distance
      rsq = dx*dx+dy*dy+dz*dz
      if (rsq. lt. rsqmax) then
      rij = sqrt(rsq)
      bin = int(rij/delr)
      hist(bin) = hist(bin) + 2
      endif
   enddo
enddo
```

If this procedure has been performed N_h times, the average number at separation $r = (b + 0.5) * \delta r$ is:

$$n(b) = \frac{hist(b)}{N \times N_h}$$

The average number of atoms in the same interval in an ideal gas is:

$$n^{id}(b) = \frac{4\pi\rho}{3}\left((r + \delta r)^3 - r^3\right)$$

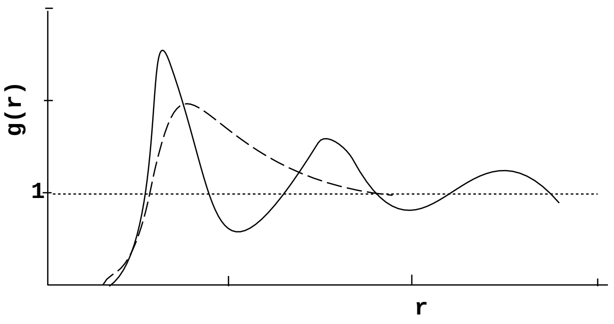


Figure 5.4.2: Radial distribution function $g(r)$ in a dense liquid (solid line) and a dilute gas (dashed line) of Lennard-Jones molecules.

Hence,

$$g(r) = \frac{n(b)}{n^{id}(b)}.$$

In Figure 5.4.2 we show the typical behavior of the radial distribution function $g(r)$ for a dense fluid and a dilute gas. For small distances r the strong repulsive interactions between molecules lead to an excluded volume of radius $\approx \sigma$ around a central molecule, so that $g(r) = 0$ for $r \leq \sigma$. Packing effects of spheres will lead to the typical oscillatory structure of $g(r)$ observed in a dense liquid. For large distances there is no correlation between the positions of the molecules; $g(r) \to 1$ asymptotically. In a dilute gas only pairs of molecules will occupy the same region in space at any given time, so that the behavior of $g(r)$ is entirely determined by the pair-potential $\varphi(r)$:

$$g(r) \approx \exp\left(-\frac{\varphi(r)}{k_B T}\right) \qquad \rho \to 0.$$

5.5 Dynamical Properties

Besides equation of state and structural data an *MD* simulation can be used to obtain information about time-dependent quantities and transport coefficients. It has been shown in most standard texts on non-equilibrium phenomena that transport coefficients can be expressed in terms of the so-called time correlation functions, that describe the decay of fluctuations of the system in equilibrium. We illustrate this for the case of self-diffusion of particles in a fluid.
Consider a solute that is present in very low concentration in a fluid solvent. The solute can, in fact, be a few tagged solvent molecules. The concentration of the solute molecules is sufficiently low, so that their mutual interaction can be neglected. Let $n(\mathbf{r}, t)$ the macroscopic density of solute molecules at position $\mathbf{r}$ and time t. Thus $n(\mathbf{r}, t) = \bar{\rho}(\mathbf{r}, t)$, where the microscopic density $\rho(\mathbf{r}, t)$ is defined as

$$\rho(\mathbf{r},t) = \sum_{j=1}^{N} \delta\left(\mathbf{r} - \mathbf{r}_j(t)\right). \tag{5.5.1}$$

($\bar{\rho}(\mathbf{r},t)$ refers to the average density of tagged particles at position $\mathbf{r}$ and time t.) Here $\mathbf{r}_j(t)$ is the position of the j^{th} solute molecule at time t. The macroscopic density and the corresponding particle flux $\mathbf{j}(\mathbf{r},t)$ of the solute molecules satisfy an equation of continuity of the form:

$$\frac{\partial}{\partial t} n(\mathbf{r},t) + \boldsymbol{\nabla} \cdot \mathbf{j}(\mathbf{r},t) = 0. \tag{5.5.2}$$

and a constitutive relation is provided by *Fick's law*:

$$\mathbf{j}(\mathbf{r},t) = -D\boldsymbol{\nabla} n(\mathbf{r},t), \tag{5.5.3}$$

where D is the *coefficient of self-diffusion*. Fick's law states that the mass flow is proportional to the concentration gradient. Since the macroscopic thermodynamic mechanism for mass flow is a gradient in a chemical potential, or, equivalently for dilute solutions, a concentration gradient, this seems a reasonable phenomenological relationship. Combining (5.5.2) with Fick's law yields:

$$\frac{\partial}{\partial t} n(\mathbf{r},t) = D\boldsymbol{\nabla}^2 n(\mathbf{r},t), \tag{5.5.4}$$

which is also referred to as Fick's law. The solution of Fick's law is readily obtained by Fourier-transformation:

$$n_{\mathbf{k}}(t) = \int d\mathbf{r}\, n(\mathbf{r},t) \exp(-\imath \mathbf{k} \cdot \mathbf{r}), \tag{5.5.5}$$

with the result that

$$n_{\mathbf{k}}(t) = n_{\mathbf{k}}(0) \exp(-Dk^2 t). \tag{5.5.6}$$

Here $n_{\mathbf{k}}(0)$ is the spatial Fourier transform of the solute density at time $t = 0$. Let us now consider how the transport coefficient D is related to the microscopic dynamics. Consider the time correlation function of the microscopic solute density $G(\mathbf{r},t)$, defined by:

$$G(\mathbf{r},t) = < \rho(\mathbf{r},t)\rho(\mathbf{0},0) > . \tag{5.5.7}$$

This function describes the decay of density fluctuations in our system. However, since we are considering a very dilute solution, $G(\mathbf{r},t)$ can also be interpreted as the probability $P(\mathbf{r},t)$ of finding a particle at position $\mathbf{r}$ at time t, given there was a particle at the origin $\mathbf{0}$ at time $t = 0$. The crucial next step is to assume that fluctuations in microscopic quantities decay, in the mean, according to the laws that govern their macroscopic behavior. This is in essence the content of

Onsager's *regression* hypothesis [165, 166], which brought him the Nobel prize in Chemistry. It is an intuitively appealing hypothesis, that can be justified rigorously on the basis of the *fluctuation-dissipation* theorem [163]. Hence the probability $P(\mathbf{r},t)$ obeys

$$\frac{\partial}{\partial t}P(\mathbf{r},t) = D\boldsymbol{\nabla}^2 P(\mathbf{r},t), \tag{5.5.8}$$

with solution

$$P_{\mathbf{k}}(t) = P_{\mathbf{k}}(0)\exp(-Dk^2t). \tag{5.5.9}$$

Since the particle is known to be at the origin at $t=0$ we have that $P(\mathbf{r},0)=\delta(\mathbf{r})$, whence

$$P_{\mathbf{k}}(0) = 1.$$

The Fourier transform can be inverted to yield:

$$P(\mathbf{r},t) = P(r,t) = \frac{1}{(4\pi Dt)^{\frac{3}{2}}}\exp\left(-\frac{r^2}{4Dt}\right). \tag{5.5.10}$$

Equation (5.5.10) describes the probability for a particle to travel a distance r in time t. Hence the mean square of the distance traveled by the particle in time t: $<\Delta r^2(t)>=<|\mathbf{r}(t)-\mathbf{r}(0)|^2>$, usually referred to as the *mean square displacement*, is obtained as:

$$<\Delta r^2(t)>= 4\pi\int_0^\infty P(r,t)r^4dr = 6Dt. \tag{5.5.11}$$

This result, which was first derived by Einstein in 1902, clarifies the physical meaning of diffusion coefficient D. Note that, since Fick's law is a macroscopic description, valid only for long times, the Einstein relation (5.5.11) is only valid after an initial transient time. In Figure 5.5.3 the behavior of the mean square displacement is shown.
To conclude the analysis, note that

$$\mathbf{r}(t)-\mathbf{r}(0) = \int_0^t \mathbf{v}(t')dt',$$

where $\mathbf{v}$ is the velocity of the tagged particle. Hence

$$<\Delta r^2(t)>= \int_0^t dt'\int_0^t dt'' <\mathbf{v}(t')\cdot\mathbf{v}(t'')> .$$

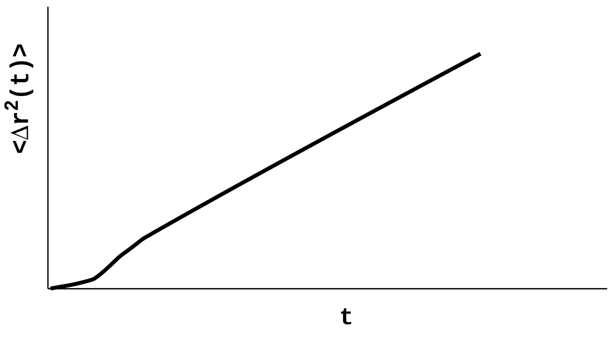


Figure 5.5.3: Mean square displacement of a tagged molecule as a function of time.

Differentiating both sides gives:

$$\begin{aligned}\frac{d}{dt} < \Delta r^2(t) > &= 2 < \mathbf{v}(t) \cdot (\mathbf{r}(t) - \mathbf{r}(0)) >= 2 < \mathbf{v}(0) \cdot (\mathbf{r}(0) - \mathbf{r}(-t)) > \\ &= 2 \int_{-t}^{0} dt' < \mathbf{v}(0) \cdot \mathbf{v}(t') >= 2 \int_{0}^{t} dt' < \mathbf{v}(0) \cdot \mathbf{v}(t') > .\end{aligned}$$

In deriving this result we have made use of two important properties of time correlation functions. Since the time correlation function describes the average decay of fluctuations in equilibrium it must be *stationary*, i.e. independent of the origin of time. Therefore:

$$< A(t')B(t'') >=< A(t+t')B(t+t'') > .$$

Furthermore time autocorrelation functions must be even functions of time, so that:

$$< A(t')A(t'') >= C(t''-t') = C(t'-t'').$$

Now, clearly,

$$\lim_{t\to\infty} \frac{d}{dt} < \Delta r^2(t) >= 6D.$$

Therefore:

$$D = \frac{1}{3} \int_{0}^{\infty} < \mathbf{v(0)} \cdot \mathbf{v(t)} > dt. \qquad (5.5.12)$$

Equation (5.5.12) is an example of a Green-Kubo relation, relating a transport coefficient to an integral over a time correlation function. For all transport coefficients such a relation exists.
Finally, consider the velocity relaxation time τ_{rel} defined by

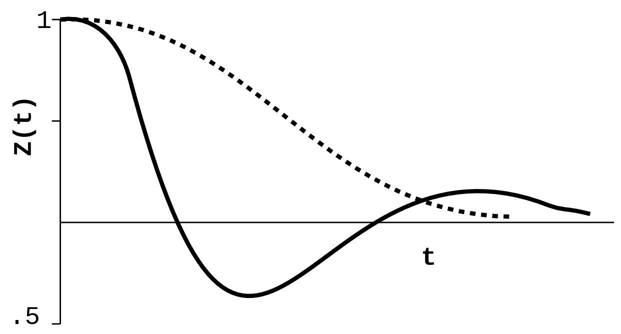


Figure 5.5.4: Velocity autocorrelation function in dense liquid (solid line) and dilute gas (dashed line) of Lennard-Jones molecules

$$\tau_{rel} = \frac{1}{3}\int_0^\infty \frac{<\mathbf{v(0)}\cdot\mathbf{v(t)}>}{<\mathbf{v}^2(0)>}dt = \frac{mD}{k_BT}.$$

This is the time it takes for the diffusional regime to dominate the motion and the time it takes for non-equilibrium velocity distributions to relax to the Gaussian equilibrium distribution. For time $t << \tau_{rel}$ the motion is inertial and we have that:

$$<\Delta r^2(t)>=<\mathbf{v}^2> t^2 = k_BTt^2.$$

In the case the solute is a tagged solvent molecule the diffusion coefficient describes how a solvent molecule diffuses through the system in time. The velocity autocorrelation function $Z(t)$ can then be averaged over all solvent molecules. In a dilute gas binary collisions are the only mechanisms changing the velocity. This can be regarded as a stochastic Poisson process, leading to an exponential decay of $Z(t)$:

$$Z(t) = \frac{k_BT}{m}\exp\left(-\frac{t}{\tau_{rel}}\right).$$

In a dense liquid $Z(t)$ shows a more complex behavior. A molecule moves in a cage of its neighbors and collides after a short time. As a result the velocity autocorrelation function exhibits a negative region. For long times $Z(t)$ exhibits the so-called hydrodynamic tail [150, 154]:

$$Z(t) \propto t^{-\frac{d}{2}};$$

d being the dimensionality of the system. Note that this behavior implies, through (5.5.12) that in 2 dimensional systems the diffusion coefficient does not exist. In Figure 5.5.4 we show the typical behavior of Z(t) for a system of Lennard-Jones molecules in a dilute gas or in a dense liquid.

In an *MD* simulation $Z(t)$ is calculated as a time-average:

$$Z(t) = \frac{1}{T}\int_0^T \frac{1}{N}\sum_{i=1}^{N} \mathbf{v}(t+\tau)\mathbf{v}(\tau)d\tau, \tag{5.5.13}$$

which, in discretized form, reads:

$$Z(t_m) = \frac{1}{NN_t}\sum_{i=1}^{N}\sum_{n_t=1}^{N_t} \mathbf{v}_i(t_{n_t})\cdot\mathbf{v}_i(t_{n_t}+t_m).$$

Here the sum over n_t is over different time origins in the system and $t_m = m\delta t$.

For completeness we give the Green-Kubo relations for some other transport coefficients. The Green-Kubo relation for the shear viscosity is:

$$\eta = \frac{1}{\Omega k_B T}\int_0^\infty < \sigma_{xy}(t)\sigma_{xy}(0) > dt. \tag{5.5.14}$$

Here $\sigma_{\alpha\beta}$ is an element of the stress-tensor:

$$\sigma_{\alpha\beta} = \sum_{i=1}^{N}\left(m v_{i\alpha} v_{i\beta} + \frac{1}{2}\sum_{j\neq i}^{N} \frac{r_{ij\alpha} r_{ij\beta}}{r_{ij}^2}\frac{\partial u(r_{ij})}{\partial r_{ij}}\right), \tag{5.5.15}$$

where $\alpha, \beta = x, y, z$ and we have assumed, for simplicity, that the particles interact through a pair-potential $u(r)$. Note that the virial theorem gives immediately, that

$$< \sigma_{\alpha\beta} >= P\Omega\delta_{\alpha\beta},$$

where $\delta_{\alpha\beta}$ is the Kronecker δ.

The Green-Kubo relation for the thermal conductivity λ reads

$$\lambda = \frac{1}{3\Omega k_B T}\int_0^\infty dt < \mathbf{j}_e(t)\cdot\mathbf{j}_e(0) > dt, \tag{5.5.16}$$

where the heat current $\mathbf{j}_e$ is defined as:

$$\mathbf{j}_e = \sum_{i=1}^{N}(\epsilon_i - < \epsilon_i >)\mathbf{v}_i + \sum_{i=1}^{N-1}\sum_{j>i}^{N}(\mathbf{v}_i\cdot\mathbf{f}_{ij})\mathbf{r}_{ij}, \tag{5.5.17}$$

and ϵ_i is the energy of particle i:

$$\epsilon_i = \frac{1}{2}\sum_{j\neq i}^{N}\varphi(r_{ij}).$$

Note the factor $\frac{1}{2}$, ensuring that the total energy $E = \sum_{i=1}^{n}\epsilon_i$ is indeed the sum of single particle energies ϵ_i.

5.6 Statistics

In both, *Monte Carlo* and *molecular dynamics* simulations, the quantities of interest are usually obtained as averages over many configurations of the system. Since we are averaging, a statistical error is invariably associated with these quantities and we have to determine the accuracy of our results. If the configurations generated in a simulation were independent, standard statistical techniques could be applied to arrive at an estimate of the statistical error.
Suppose our simulation covers a total of $\mathcal{M}$ time intervals (in the case of *Monte Carlo* simulations read *Monte Carlo* "moves"). We seek an estimate of the mean value $< A >$ of some quantity $A(\{\mathbf{r}^N\}, \{\mathbf{p}^N\})$ and of the standard deviation. Let us denote by $\bar{A}_{\mathcal{M}}$ the estimate of $< A >$. Then, clearly,

$$\bar{A}_{\mathcal{M}} = \frac{1}{\mathcal{M}} \sum_{m=1}^{\mathcal{M}} A_m. \tag{5.6.1}$$

If our simulation is done correctly and the ergodic hypothesis is valid, then

$$\bar{A}_{\mathcal{M}} = \lim_{\mathcal{M}\to\infty} < A > . \tag{5.6.2}$$

So, an estimate for the variance of $\bar{A}_{\mathcal{M}}$ is:

$$\begin{aligned} var(A_{\mathcal{M}}) &= < \bar{A}^2{}_{\mathcal{M}} > - < \bar{A}_{\mathcal{M}} >^2 \\ &= \frac{1}{\mathcal{M}^2} < \sum_{m=1}^{\mathcal{M}} A_m \sum_{n=1}^{\mathcal{M}} A_n > - \frac{1}{\mathcal{M}^2} < \sum_{m=1}^{\mathcal{M}} A_m >< \sum_{n=1}^{\mathcal{M}} A_n > \\ &= \frac{1}{\mathcal{M}^2} < \sum_{m=1}^{\mathcal{M}} \sum_{n=1}^{\mathcal{M}} (A_m - < A >)(A_n - < A >) > \\ &\equiv \frac{1}{\mathcal{M}^2} < \sum_{m=1}^{\mathcal{M}} \sum_{n=1}^{\mathcal{M}} \delta A_m \delta A_n > . \end{aligned} \tag{5.6.3}$$

If successive configurations were uncorrelated $< \delta A_m \delta A_n >= \delta_{mn}$, so that

$$var(A_{\mathcal{M}}) = \frac{1}{\mathcal{M}} < \delta A^2 >, \tag{5.6.4}$$

which is the well-known formula for the error estimate for a set of independent observations. In a Monte Carlo or *MD* simulation successive configurations are, however, strongly correlated. We rewrite (5.6.3) in the form:

$$
\begin{aligned}
var(A_{\mathcal{M}}) &= \frac{1}{\mathcal{M}^2} < \sum_{m=1}^{\mathcal{M}} \delta A_m^2 > +2 < \sum_{m=1}^{\mathcal{M}-1} \sum_{n=m+1}^{\mathcal{M}} \delta A_m \delta A_n > \\
&= \frac{1}{\mathcal{M}^2} < \sum_{m=1}^{\mathcal{M}} \delta A_m^2 > +2 < \sum_{m=1}^{\mathcal{M}-1} \sum_{n=1}^{\mathcal{M}-m} \delta A_m \delta A_{m+n} > .
\end{aligned}
$$

In the last term we recognize the *time correlation function*, $C_{AA}(t = n\delta t)$, defined as $\langle A(0)A(t)\rangle$, and the sum

$$
\sum_{n=1}^{\infty} C_{AA}(n\delta t) + \frac{1}{2} C_{AA}(0) =< \delta A^2 > \frac{\tau_A^c}{\delta t}, \tag{5.6.5}
$$

defines a correlation time τ_A^c for the quantity A. Clearly, τ_A^c is a measure of the length of time over which the fluctuations of A are correlated. Hence if the duration of the simulation $T_{\mathcal{M}} \equiv \mathcal{M}\delta t >> \tau_A^c$ we may write:

$$
var(A_{\mathcal{M}}) \approx 2\frac{\tau_A^c}{T_{\mathcal{M}}} < \delta A^2 > . \tag{5.6.6}
$$

Compared to (5.6.5) we see that the root-mean-square (rms) error is proportional to $\sqrt{\tau_A^c / T_{\mathcal{M}}}$. This is hardly surprising, since the number of independent configurations is proportional to $T_{\mathcal{M}}/\tau_A^c$ and the rms error is inversely proportional to the square root of the number of independent configurations.
The results show that for accurate results one has to simulate for times much larger than the typical correlation times to obtain the quantities of interest reliably. In a number of cases, for example when studying hydrodynamic fluctuations or fluctuations, that drive a symmetry-breaking transition, the corresponding modes have a large life-time, proportional to the square of their wave-lengths. The system size must be large compared to relevant wave-lengths. Hence to study such fluctuations very long simulations (simulation time $\propto L^2$) are required.
The error estimate requires therefore the calculation of the correlation function $C_{AA}(t)$. This is often a costly calculation. A particularly elegant way of calculating the error is through a blocking transformation, which systematically removes correlations, suggested by Petersen and Flyvbjerg in 1989 [168]. Suppose we wish to estimate the error in the quantity A from a simulation. We have a set of observations $A_m, m = 1, \cdots, M$ of the quantity A, with estimates $\bar{A}_M$ for the mean and $var(A_M)$ for its variance. We now transform this set of observations into a new set by a blocking transformation

$$
A'_m = \frac{1}{2}(A_{2m} + A_{2m-1}). \tag{5.6.7}
$$

From this new set of observations we can again estimate the mean and the variance. Clearly the transformation leaves the estimate of the mean unchanged, i.e.,

$\bar{A}'_{\frac{M}{2}} = \bar{A}_M$. If successive configurations were uncorrelated we would find for the variance:

$$var(A'_{\frac{M}{2}}) = 2\ var(A_M). \tag{5.6.8}$$

By repeatedly applying this transformation we gradually remove correlations between successive observations. We can check this by plotting the quantity

$$\sigma^2(A^{(n)}) = \frac{var(A^{(n)})}{2^n},$$

versus the number of applications n of the blocking transformation, starting with $n = 0$ (i.e. $M = \mathcal{M}$). Initially, for small values of n, correlations will still be present and we generally obtain an underestimate of the standard deviation. After a few transformations, during which generally an increase in $\sigma^2(A^{(n)})$ is observed, the successive observations must become independent, so that the plot of $\sigma^2(A^{(n)})$ will show a horizontal line. For large values of n we only have a few observations, so that the statistics are not reliable. If there is no horizontal region the simulation did not last long enough and it is not possible to obtain a reliable error estimate.

Finally, we mention that a similar transformation can be used to obtain error estimates in time correlation functions. For details we refer to the literature [168].

5.7 Long Range Interactions

When particles interact through a short-range potential, such as the Lennard-Jones potential, we can cut-off the interaction at distances $r \geq r_c$ with $r_c \leq \frac{1}{2}L$, L being the side of the simulation cell. Many interesting cases exist, however, where this is no longer possible. The important cases of electrostatic and gravitational interactions fall in this category. Thus if we wish to simulate systems, containing ions and/or polar molecules, it is no longer permissible to cut off the potential and only take minimum image interactions. There are two possible strategies to deal with such situations.

Firstly, rather than considering a periodic system one can simulate a large finite system. This is often done in astrophysical applications, e.g. simulations of stellar dynamics under the influence of gravity. Note that if the system contains N particles one has to evaluate $N(N-1)/2$ interactions and forces, so that the algorithm is $\mathcal{O}(N^2)$. Hierarchical algorithms can be used to reduce this to an $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$ problem. Such an algorithm was originally devised by Barnes and Hut [152] for the gravitational problem. The best known algorithm is due to Greengard and Rokhlin [158], who devised an $\mathcal{O}(N)$ algorithm for electrostatic interactions. We will illustrate the algorithm for the case of a two-dimensional Coulomb system.

An alternative strategy is the use of periodic boundary conditions and explicitly evaluate the lattice sum equation (5.2.1). Also this procedure will be illustrated for a 2-dimensional Coulomb system.

5.7.1 The Barnes-Hut Algorithm for N-body Problems

First we briefly review an example of particle simulation. The basic algorithm of a particle simulation is

```
t = 0
while $t < t\_final$
   for $i = 1$ to $n$ /* n = number_of_particles */
      compute $F(i)$ = force on particle $i$
      move particle $i$ under force $F(i)$ for time $dt$
   end for
   compute interesting properties of particles
   t = t + dt /* dt = time increment */
end while
```

The force on a particle is typically composed of three more basic forces:

$$force = external\_force + nearest\_neighbor\_force + far\_field\_force \quad (5.7.1)$$

The *external forces* (e.g. current in the sharks and fish examples) can be computed for each particle independently, in an embarrassingly parallel way. The *nearest neighbor forces* (e.g. collision forces in the bouncing balls assignment) only require interactions with nearest neighbors to compute, and are still relatively easy to parallelize. The *far field* forces like gravity, on the other hand, are more expensive to compute because the force on each particle depends on all the other particles. The simplest expression for a far field force $F(i)$ on particle i is

```
for $i = 1$ to $n$
   $F(i) = \sum_{j=1,j\neq i}^{n} f(i,j)$
end for
```

where $f(i,j)$ is the force on particle i due to particle j. Thus, the cost seems to rise as $O(n^2)$, whereas the external and nearest neighbor forces appear to grow like $O(n)$. Even with parallelism, the far field forces will be much more expensive.

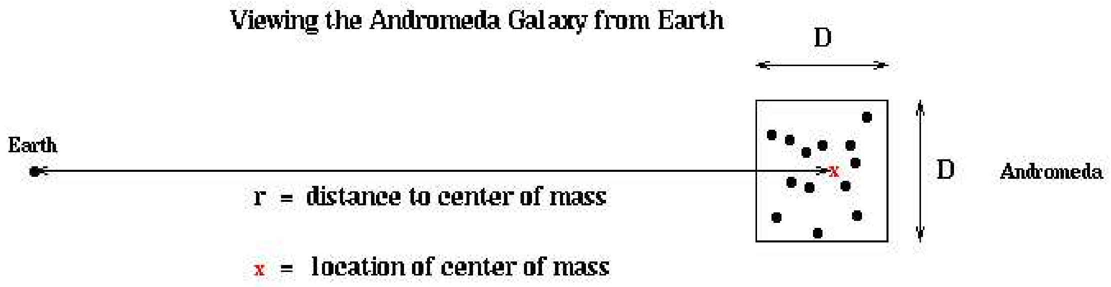
Particle simulations arise in astrophysics and celestial mechanics (gravity is the force), plasma simulation and molecular dynamics (electrostatic attraction or repulsion is the force), and computational fluid dynamics (using the vortex method). The same N-body solutions can also be used in solving elliptic partial differential equations (not just the Poisson equation, which governs gravity and electrostatics) and radiosity calculations in computer graphics (with n mutually illuminating bodies).

All these examples strongly motivate us to find a better N-body algorithm, one that even costs less than $O(n^2)$ on a serial machine. Fortunately, it turns out that there are clever divide-and-conquer algorithms which only take $O(nlog(n))$ or even just $O(n)$ time for this problem.

The success of this fast algorithm is illustrated by the following example in astrophysics application, which explored the creation and statistical distribution of galaxies, simulated over 17 million particles (stars) for over 600 time steps. This run took 24 hours on a 512 processor Intel Delta [173], where each processor sustained 10 Mflops of computational speed, for an aggregate sustained speed of over 5 Gflops. The $O(n^2)$ algorithm would have run for well over a year on the same problem.

How to reduce the number of particles in the force sum? There are two kinds of divide-and-conquer algorithms (will be described later), both depend on the following simple physical intuition. Suppose we wanted to compute the gravitational force on the earth from the known stars and planets. A glance skyward on a clear night reveals a tauntingly large number of stars that must be included in the calculation, each one contributing a term to the force sum.

One of those dots of light we might want to include in our sum is, however, not a single star (particle) at all, but rather the Andromeda galaxy, which itself consist of billions of stars. But these appear so close together at this distance


Viewing the Andromeda Galaxy from Earth
D
Earth
D
Andromeda
r = distance to center of mass
x = location of center of mass

that they show up as a single dot to the naked eye. It is tempting – and correct – to suspect that it is good enough to treat the Andromeda galaxy as a single point anyway, located at the center of mass of the Andromeda galaxy, and with a mass equal to the total mass of the Andromeda galaxy. This is indicated below, with a x marking the center of mass. More mathematically, since the ratio

$$\frac{D}{r} = \frac{size\ of\ box\ containing\ Andromeda}{distance\ of\ center\ of\ mass\ from\ Earth} \tag{5.7.2}$$

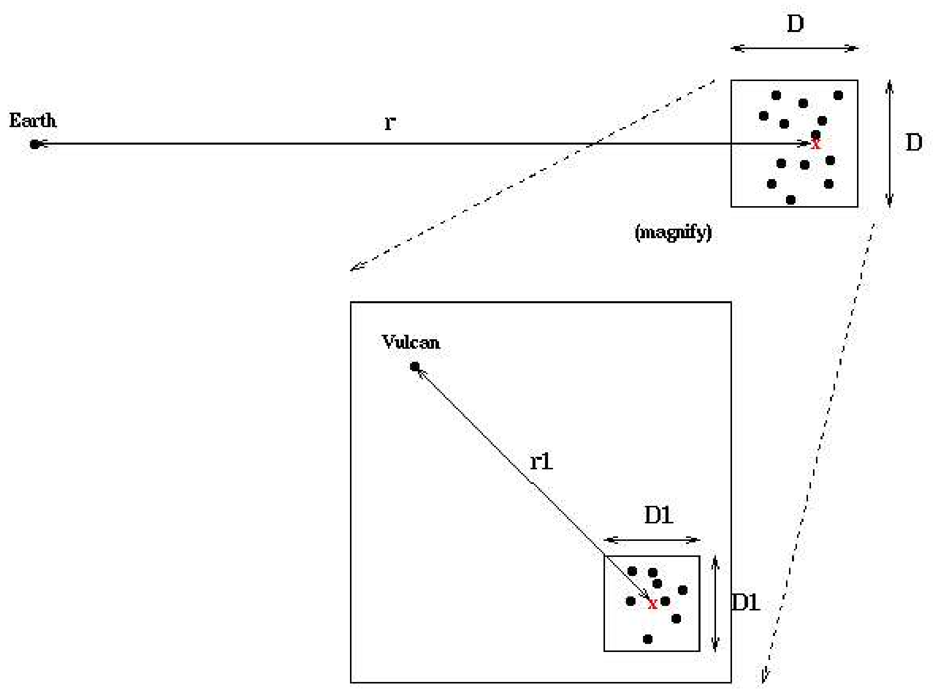
is so small, we can safely and accurately replace the sum over all stars in Andromeda with one term at their center of mass.

This idea is hardly new. Indeed, Newton modeled the earth as a single point mass located at its center of mass in order to calculate the attracting force on the falling apple, rather than treating each tiny particle making up the earth separately.

What is new is applying this idea recursively. First, it is clear that from the point of view of an observer in the Andromeda galaxy, our own Milky Way galaxy can also be well approximated by a point mass at our center of mass. But more importantly, within the Andromeda (or Milky Way) galaxy itself, this geometric picture repeats itself as shown below: As long as the ratio $D1/r1$ is also small, the stars inside the smaller box can be replaced by their center of mass in order to compute the gravitational force on, say, the planet Vulcan. This nesting of boxes within boxes can be repeated recursively.

Quadtrees and Octtrees What we need is a data structure to subdivide space that makes this recursion easy. The answer in 3D is the *octtree*, and in 2D the answer is the *quadtree*. We begin by describing the quadtree, because it is easier to draw in 2D; the octtree will be analogous. The quadtree begins with a square in the plane; this is the root of the quadtree. This large square can be broken into four smaller squares of half the perimeter and a quarter the area each; these are the four children of the root. Each child can in turn be broken into 4 subsquares to get its children, and so on. This is shown below. Each colored dot in the tree corresponds to a square in the picture on the left, with edges of that color (and

Replacing Clusters by their Centers of Mass Recursively



of colors from higher tree levels). An octtree is similar, but with 8 children per node, corresponding to the 8 subcubes of the larger cube.

The algorithms begin by constructing a quadtree (or octtree) to store the particles. Thus, the leaves of the tree will contain (or have pointers to) the positions and masses of the particles in the corresponding box.

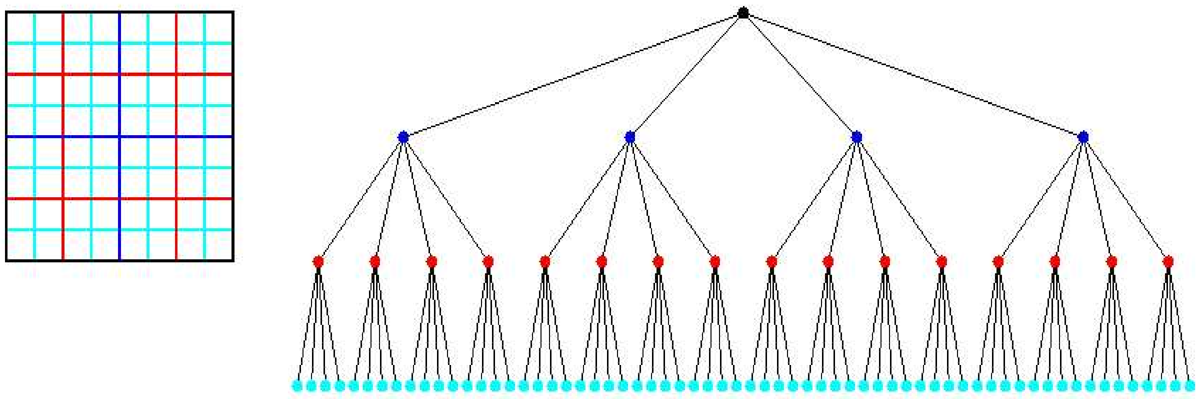
The most interesting problems occur when the particles are not uniformly distributed in their bounding box, so that many of the leaves of a complete quadtree would be empty. In this case, it makes no sense to store these empty parts of the quadtree. Instead, we continue to subdivide squares only when they contain more than 1 particle (or some small number of particles). This leads to the adaptive quadtree shown in the figure below. Note that there are exactly as many leaves as particles. Children are ordered counterclockwise starting at the lower left.

Given a list of particle positions, a quadtree can be constructed recursively as follows:

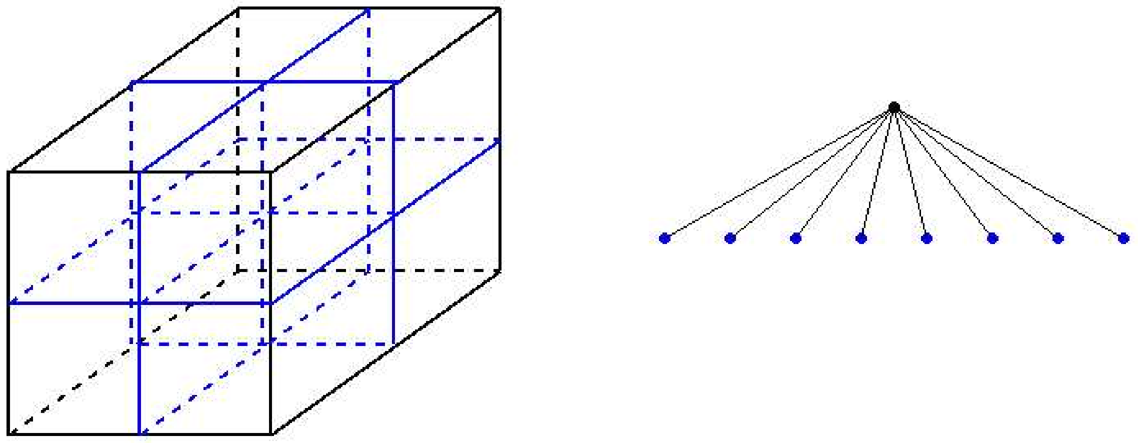
```
procedure QuadtreeBuild
  Quadtree = {empty}
    For i = 1 to n          ... loop over all particles
      QuadInsert(i, root)   ... insert particle i in quadtree
    end for
    ... at this point, the quadtree may have some empty
    ... leaves, whose siblings are not empty
    Traverse the tree (via, say, breadth first search),
      eliminating empty leaves

procedure QuadInsert(i,n)
  ... Try to insert particle i at node n in quadtree
  ... By construction, each leaf will contain either
  ... 1 or 0 particles
  if the subtree rooted at n contains more than 1 particle
     determine which child c of node n particle i lies in
       QuadInsert(i,c)
  else if the subtree rooted at n contains one particle
     ... n is a leaf
     add n's four children to the Quadtree
     move the particle already in n into the child
        in which it lies
     let c be child in which particle i lies
     QuadInsert(i,c)
  else if the subtree rooted at n is empty
     ... n is a leaf
     store particle i in node n
```

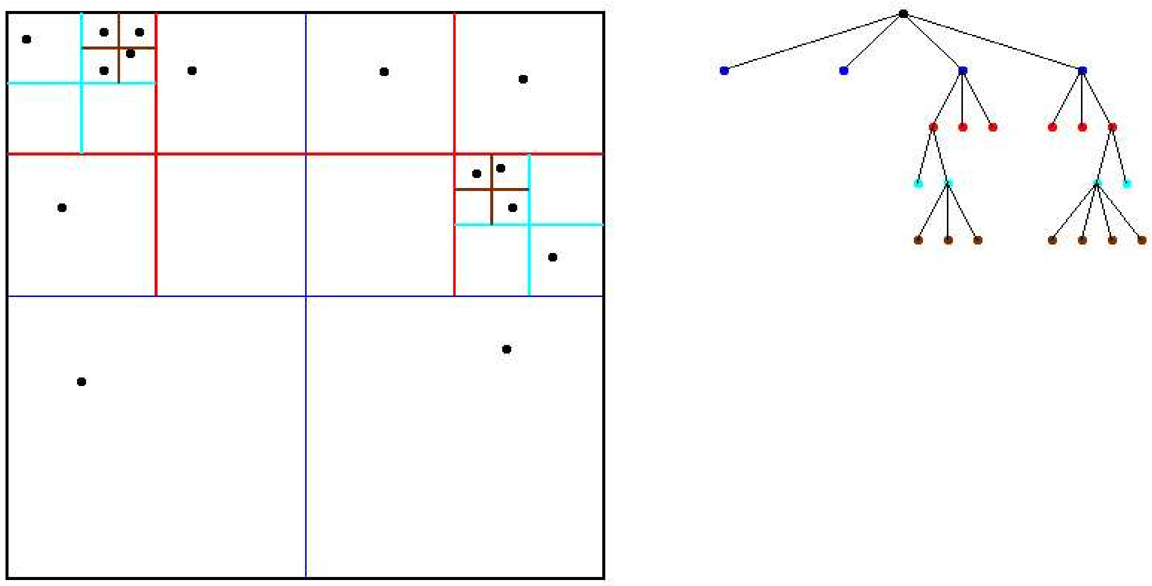
A Complete Quadtree with 4 Levels



2 Levels of an Octree



Adaptive quadtree where no square contains more than 1 particle



```
    endif
```

The complexity of QuadtreeBuild depends on the distribution of the particles inside the bounding box. The cost of inserting a particle is proportional to the distance from the root to the leaf in which the particle resides (measured by the level of the leaf, with the root at level 0). If the particles are all clustered closely together, so that they all lie in a single tiny subsquare of the bounding box, then the complexity can be large because the leaf can be far from the root. But it cannot be farther than the number of bits b used to represent the positions of the particles, which we may consider a constant. If the particles are uniformly distributed, so the leaves of the quadtree are all on the same level, the complexity of inserting all the particles will be $O(n*log(n)) = O(n*b)$. (Note that $log(n) \leq b$, since we can have at most 2^b distinct particles.) Whether to call this $O(n*log(n))$ or $O(n)$ depends on your point of view.

We will return to data structure representations of quadtrees and octtrees when we discuss parallelization. But for now we know enough to discuss the simplest hierarchical N-body algorithm, which is due to Barnes and Hut.

The Barnes-Hut Algorithm The Barnes-Hut algorithm was presented in [152] and is based on an earlier algorithm of A. Appel in 1985. It is widely used in astrophysics, and has been most thoroughly parallelized. It is not as accurate, however, as the slower Fast Multipole Method (FMM). [158] Barnes-Hut is often used when 1% accuracy is desired, which is often enough for astrophysical calculations.

At a high level, the Barnes-Hut algorithm comprises the following steps:

a. Build the Quadtree, using QuadtreeBuild as described above.

b. For each subsquare in the quadtree, compute the center of mass and total mass for all the particles it contains.

c. For each particle, traverse the tree to compute the force on it.

The first step of the building the quadtree has already been described in the previous subsection. The second step of the algorithm is accomplished by a simple "post order traversal" of the quadtree (post order traversal means the children of a node are processed before the node). Note that the mass and center of mass of each tree node n are stored at n; this is important for the next step of the algorithm.

```
... Compute the center of mass and total mass
... for particles in each subsquare
( mass, cm ) = Compute_Mass(root)   ... cm = center of mass

function ( mass, cm ) = Compute_Mass(n)
     ... Compute the mass and center of mass (cm) of
     ... all the particles in the subtree rooted at n
     if n contains 1 particle
          ... the mass and cm of n are identical to
          ... the particle's mass and position
          store ( mass, cm ) at n
          return ( mass, cm )
     else
          for all four children c(i) of n (i=1,2,3,4)
              ( mass(i), cm(i) ) = Compute_Mass(c(i))
          end for
          mass = mass(1) + mass(2) + mass(3) + mass(4)
                ... the mass of a node is the sum of
                ... the masses of the children
          cm = (  mass(1)*cm(1) + mass(2)*cm(2)
                + mass(3)*cm(3) + mass(4)*cm(4)) / mass
                ... the cm of a node is a weighted sum of
                ... the cm's of the children
          store ( mass, cm ) at n
          return ( mass, cm )
     end
```

The cost of this algorithm is proportional to the number of nodes in the tree, which is no more than $O(n)$, modulo the distribution of particles (in other words, it could be $O(n*log(n))$ or $O(n*b)$ where b is the number of bits used to represent position).

Finally, we come to the core of the algorithm, computing the force on each particle. We use the idea in the first figure above, namely that if the ratio

$$\frac{D}{r} = \frac{size\ of\ box}{distance\ from\ particle\ to\ center\ of\ mass\ of\ box}$$

is small enough, then we can compute the force due to all the particles in the box just by using the mass and center of mass of the particles in the box. We will compare $\frac{D}{r}$ to a user-supplied threshold theta (usually a little less than 1) to make this decision.

Thus, if $\frac{D}{r} < \theta$, we compute the gravitational force on the particle as follows. Let

- (x, y, z) be the position of the particle (in 3D),
- m be the mass of the particle,
- (x_{cm}, y_{cm}, z_{cm}) be the position of the center of mass of the particles in the box,
- m_{cm} be the total mass of the particles in the box, and
- G be the gravitational constant.

Then the gravitational force attracting the particle is approximated by

$$force = G * m * m_{cm} * \frac{x_{cm} - x}{r^3}, \frac{y_{cm} - y}{r^3}, \frac{z_{cm} - z}{r^3}) \tag{5.7.3}$$

where $r = \sqrt{x_{cm} - x)^2 + (y_{cm} - y)^2 + (z_{cm} - z)^2}$ is the distance from the particle to the center of mass of the particles in the box.

Here is the algorithm for step 3 of Barnes-Hut:

```
... For each particle, traverse the tree
... to compute the force on it.
For i = 1 to n
    f(i) = TreeForce(i,root)
end for

function f = TreeForce(i,n)
    ... Compute gravitational force on particle i
    ... due to all particles in the box at n
```

```
f = 0
if n contains one particle
    f = force computed using formula (3) above
else
    r = distance from particle i to
          center of mass of particles in n
    D = size of box n
    if D/r < theta
        compute f using formula (3) above
    else
        for all children c of n
            f = f + TreeForce(i,c)
        end for
    end if
end if
```

The correctness of the algorithm follows from the fact that the force from each part of the tree is accumulated in f recursively. To understand the complexity, we will show that each call to $TreeForce(i, root)$ has a cost proportional to the depth at which the leaf holding particle i occurs in the tree. To illustrate, consider the figure below, which shows the result of calling $TreeForce(i, root)$ with $\theta > 1$, for the particle i shown in the lower right corner. For each of the unsubdivided squares shown, except the one containing particle i, we have $D/r \leq 1 < \theta$. (There are 3 such large blue unsubdivided squares, 3 smaller red squares, 3 yet smaller cyan squares, and 3 smallest brown squares.) Since the test $D/r < \theta$ is satisfied for each unsubdivided square, a constant amount of work is done for each unsubdivided square. Since there are at most 3 undivided squares at each level of the tree, the total amount of work for particle i is proportional to the depth at which particle i is stored.

Thus, the total cost of step 3 of the Barnes-Hut algorithm is the sum of the depths in the quadtree at which each particle is stored; as before, this may be bounded either by $O(n * log(n))$ or $O(n * b)$.

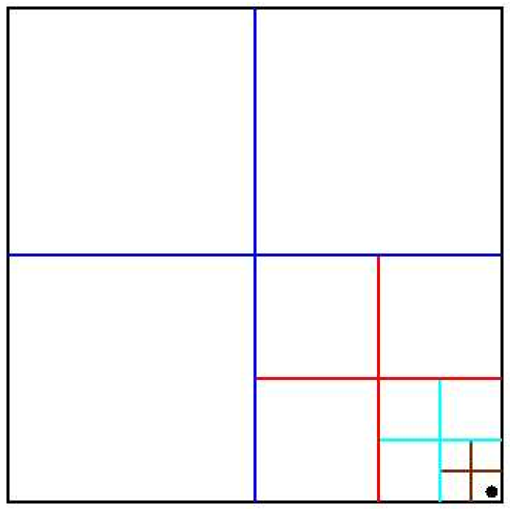
When $\theta < 1$ (as it usually is), it is more complicated to describe which boxes satisfy $D/r < \theta$, but the same $O(.)$ style analysis yields.

5.7.2 Greengard-Rokhlin Algorithm

We consider a system in two dimensions, consisting of N charges interacting through a pairwise additive Coulomb potential. In two dimensions the Coulomb potential $\phi(\mathbf{x})$ at a point $\mathbf{x} = (x, y)$, due to a unit point charge at a point $\mathbf{x}_0 = (x_0, y_0)$, is given by:

$$\phi(\mathbf{x}) = -\log|\mathbf{x} - \mathbf{x}_0|, \tag{5.7.4}$$

Sample Barnes–Hut Force calculation
For particle in lower right corner
Assuming theta > 1



and the corresponding electrostatic field is

$$\mathbf{E}(\mathbf{x}) = \frac{\mathbf{x} - \mathbf{x}_0}{|\mathbf{x} - \mathbf{x}_0|^2}. \tag{5.7.5}$$

Physically this corresponds to an infinite charged line with a charge $\lambda = 1$ per unit length. The potential $\phi(\mathbf{x})$ is a harmonic function, i.e. a solution of the Laplace equation:

$$\nabla^2\phi(\mathbf{x}) = 0, \tag{5.7.6}$$

in any region not containing $\mathbf{x}_0$. It is well-known that in this case an analytic function w exists in the complex domain, $w : \mathbb{C} \to \mathbb{C}$, such that $\phi(x, y) = \Re w(x, y)$ and the force field is given by:

$$\nabla\phi(x, y) = (\phi_x, \phi_y) = (\Re w'. - \Im w'). \tag{5.7.7}$$

The prime denotes differentiation with respect to the complex variable $z = x + \imath y$. This equation follows immediately from the Cauchy-Riemann relations for the derivative of an analytic function. In the following we follow the general custom to refer to the analytic function $\log(z)$ as the potential to a unit charge (note the sign change!). We will now derive a multipole expansion for a set of charges $q_1, q_2, \cdots, q_m$ located at points $z_1, z_2, \cdots, z_m$. Let $|z_i| \leq r$, Then, for any $z \in \mathbb{C}$ with $|z| > r$ the potential is given by

$$\begin{aligned}\phi(z) &= \sum_{i=1}^{m} q_i \log(z - z_i) \\ &= \sum_{i=1}^{m} q_i \log z + \sum_{i=1}^{m} q_i \log\left(1 - \frac{z_i}{z}\right) \\ &= Q\log(z) + \sum_{k=1}^{\infty}\sum_{i=1}^{m} \frac{-q_i}{k}\left(\frac{z_i}{z}\right)^k \\ &= Q\log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k},\end{aligned}$$

so that:

$$\phi(z) = Q\log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k}, \tag{5.7.8}$$

with

$$Q = \sum_{i=1}^{m} q_i, \quad a_k = \sum_{i=1}^{m} \frac{-q_i z_i^k}{k}. \tag{5.7.9}$$

Equation (5.7.8) is the multipole expansion for two-dimensional system containing a set of charges q_i located at z_i. For computational purposes it is important to have an error bound when the multipole expansion is *truncated.* In other words, we want an upper bound when the series (5.7.8) is truncated at $k = p$. We have:

$$\begin{aligned}\left|\phi(z) - Q\log(z) - \sum_{k=1}^{p} \frac{a_k}{z^k}\right| &= \left|\sum_{k=p+1}^{\infty} \frac{a_k}{z^k}\right| = \\ \left|\sum_{k=p+1}^{\infty} \frac{-\sum_{i=1}^{m} q_i z_i^k}{kz^k}\right| &\leq \sum_{i=1}^{m} |q_i| \sum_{k=p+1}^{\infty} \frac{r^k}{k|z^k|} \leq \\ A\sum_{k=p+1}^{\infty} \left|\frac{r}{z}\right|^k &= \frac{A}{1 - |r/z|}\left|\frac{r}{z}\right|^{p+1} \\ &= \frac{A}{|z/r| - 1}\left|\frac{r}{z}\right|^{p}.\end{aligned}$$

Hence we have that:

$$\left|\phi(z) - Q\log(z) - \sum_{k=1}^{p} \frac{a_k}{z^k}\right| \leq \frac{A}{c-1}\left(\frac{1}{c}\right)^p, \tag{5.7.10}$$

with

$$A \;=\; \sum_{i=1}^{m} |q_i|\,,\; c = \left|\frac{z}{r}\right|. \tag{5.7.11}$$

In particular if $c = 2$ then

$$\left|\phi(z) - Q\log(z) - \sum_{k=1}^{p} \frac{a_k}{z^k}\right| \leq A\left(\frac{1}{2}\right)^p. \tag{5.7.12}$$

We have derived the multipole expansion for a 2D system of charges and provided an upper bound for the error when the series is truncated.

The essence of the Greengard-Rokhlin algorithm is the clustering of charges far away from a point where the potential and force have to be calculated. To see that such an approach is feasible, consider two sets of charges, which we distinguish by a prime and a double prime. The primed set of m' charges are all located in a circle D' around z_0' with radius R, whereas the m'' doubly-primed charges are in a circle D'' around z''_0 also with radius R. The electrostatic energy of the system is

$$U = \sum_i \sum_j log(z_i' - z_j''), \tag{5.7.13}$$

which obviously requires $m'm''$ evaluations of the logarithm. When the charges are sufficiently far apart, i.e. $|z_0' - z_0''| > 2R$ the multipole expansion can be used. We shall actually impose a more stringent condition, namely $|z_0' - z_0''| \geq 3R$, so that $c \geq 2$ and the bound (5.7.12) applies. This is shown in Figure 5.7.5. When two sets of charges satisfy this condition we call these sets *well separated.* Truncating the expansion at $k = p$ we require $\sim p$ operations to evaluate the multipoles and hence $\mathcal{O}(p(m' + m''))$ operations to evaluate the electrostatic energy to an accuracy provided by the truncation. For large numbers of charges the second option obviously reduces the computational cost considerably. Moreover, the bound (5.7.12) shows that to attain a relative accuracy ϵ, p should be of the order $-\log_2(\epsilon)$.

The application of the multipole expansion for the evaluation of electrostatic forces requires the translation of multipole centers. In the following we derive the necessary translation formulae with error bounds. Suppose that

$$\phi(z) = a_0\log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k}, \tag{5.7.14}$$

is a multipole expansion of the potential due to a set of charges $q_1, q_2, \cdots, q_m$, all of which are located inside a circle $\mathcal{D}$ of radius R and center z_0. Then for z outside a circle $\mathcal{D}_1$ of radius $R + |z_0|$ centered at the origin, as shown in Figure 5.7.6, we have that:

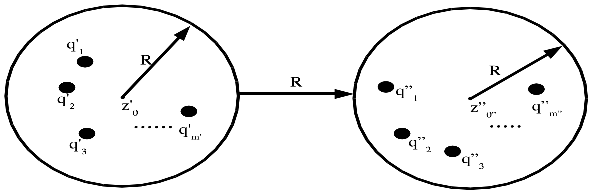


Figure 5.7.5: Well separated sets of charges in the plane.

$$\phi(z) = a_0 \log(z) + \sum_{l=0}^{\infty} \frac{b_l}{z^l}. \tag{5.7.15}$$

The proof follows from the expansion of the analytic functions in (5.7.14).

$$\begin{aligned}
\phi(z) &= a_0 \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k} \\
&= a_0 \log(z) + a_0 \log\left(1 - \frac{z_0}{z}\right) + \sum_{k=1}^{\infty} \frac{a_k}{z^k \left(1 - \frac{z_0}{z}\right)^k} \\
&= a_0 \log(z) + a_0 \sum_{l=1}^{\infty} \frac{(-1)}{l} \left(\frac{z_0}{z}\right)^l + \sum_{k=1}^{\infty} \frac{a_k}{z^k} \sum_{l=0}^{\infty} \frac{k(k+1)\cdots(k+l-1)}{l!} \left(\frac{z_0}{z}\right)^k.
\end{aligned}$$

In the third term we substitute $k + l = p$ and $k = q$ to obtain:

$$\sum_{k=1}^{\infty} \sum_{l=0}^{\infty} \frac{a_k}{z^k} \begin{pmatrix} k+l-1 \\ k-1 \end{pmatrix} \left(\frac{z_0}{z}\right)^l = \sum_{p=1}^{\infty} \sum_{q=1}^{p} \frac{a_q}{z^p} \begin{pmatrix} p-1 \\ q-1 \end{pmatrix} z_0^{p-q}, \tag{5.7.16}$$

and (5.7.14) and (5.7.15) follow immediately. Here $\begin{pmatrix} k \\ l \end{pmatrix}$ denotes the binomial coefficient.

To obtain an error bound for the truncation of the series we note that the coefficients b_l are the coefficients of a unique multipole expansion about the origin of

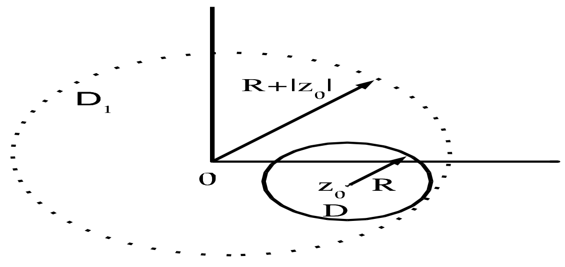


Figure 5.7.6: The domains D and D_1 in the complex plane.

the charges contained in $\mathcal{D}$. The expansion converges outside $\mathcal{D}_1$. Hence we can immediately apply the error bound for the truncation of the multipole expansion equation (5.7.10) to obtain:

$$\left| \phi(z) - a_0 \log(z) - \sum_{l=1}^{p} \frac{b_l}{z^l} \right| \leq \frac{A}{1-\omega} \omega^{p+1}, \tag{5.7.17}$$

with:

$$\omega = \left| \frac{|z_0| + R}{z} \right|. \tag{5.7.18}$$

Once the set of values $\{a_0, a_1, \cdots, a_p\}$ has been determined we can calculate the set of coefficients $\{b_0, b_1, \cdots, b_p\}$ exactly. Thus, we can shift the origin without any loss in accuracy.

Now consider a set of charges $q_1, q_2, \cdots, q_m$ located inside a circle $\mathcal{D}_1$ with radius R and center z_0, with $|z_0| > (c+1)R$ and $c > 1$, as shown in Figure 5.7.7. Then the corresponding multipole expansion (5.7.14) around z_0 converges inside a circle $\mathcal{D}_2$ of radius R centered around the origin. To obtain an expression for the potential inside $\mathcal{D}_2$ we expand the different terms in (5.7.14) in powers of z/z_0:

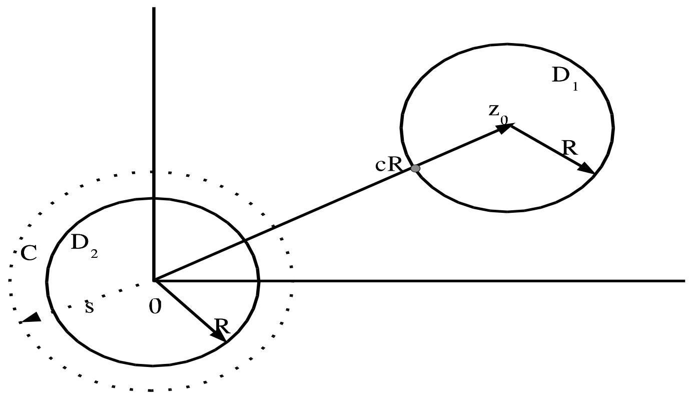


Figure 5.7.7: The domains D_1 and D_2 and the circle C in the complex plane.

$$\begin{aligned}
\phi(z) &= a_0\log(-z_0) + a_0\log(1-z/z_0) + \sum_{k=1}^{\infty}\frac{(-1)^k a_k}{z_0^k}\frac{1}{\left(1-\frac{z}{z_0}\right)^k} \\
&= a_0\log(-z_0) - a_0\sum_{l=1}^{\infty}\left(\frac{z}{z_o}\right)^l + \sum_{k=1}^{\infty}\frac{(-1)^k a_k}{z_0^k}\sum_{l=0}^{\infty}\binom{k+l-1}{k-1}\left(\frac{z}{z_0}\right)^l.
\end{aligned}$$

Hence the potential inside $\mathcal{D}_2$ has a power series expansion, as follows,

$$\phi(z) = \sum_{l=0}^{\infty} b_l z^l, \tag{5.7.19}$$

with coefficients

$$\begin{aligned}
b_0 &= \sum_{k=1}^{\infty}\frac{(-1)^k a_k}{z_0^k} + a_0\log(-z_0) \\
b_l &= \frac{1}{z_0^l}\sum_{k=1}^{\infty}\frac{(-1)^k a_k}{z_0^k}\binom{l+k-1}{k-1} - \frac{a_0}{l z_0^l} \quad (l \geq 1).
\end{aligned}$$

To find an error bound upon truncation of the series consider:

$$\left|\phi(z) - \sum_{l=0}^{p} b_l z^l\right| = \left|\sum_{l=p+1}^{\infty} b_l z^l\right|. \tag{5.7.20}$$

Now let $\gamma_0 \equiv a_0\log(-z_0)$ and $\gamma_l = -a_l/l z_0^l$ for $l \geq 1$. Further, let $\beta_l = b_l - \gamma_l$. Then, by the Schwartz inequality:

$$\left|\phi(z) - \sum_{l=0}^{p} b_l z^l\right| \leq \left|\sum_{l=p+1}^{\infty} \gamma_l z^l\right| + \left|\sum_{l=p+1}^{\infty} \beta_l z^l\right| = S_1 + S_2. \tag{5.7.21}$$

An upper bound for S_1 is easily found since,

$$\begin{aligned}\left|\sum_{l=p+1}^{\infty} \gamma_l z^l\right| &\leq |a_0| \sum_{l=p+1}^{\infty} \frac{1}{l}\left|\frac{z}{z_0}\right| < |a_0| \sum_{l=p+1}^{\infty} \left|\frac{z}{z_0}\right| \\ &= \frac{A}{\left|\frac{z}{z_0} - 1\right|}\left|\frac{z}{z_0}\right|^p. \end{aligned} \tag{5.7.22}$$

To find a bound for S_2 we consider a circle C around the origin with radius s such that $s = cR(p-1)/p$, also shown in Figure 5.7.7. Note that s increases continuously with $p > 0$, so that

$$R < \frac{1}{2}(c+1)r < S < Cr.$$

Let $\phi_1 : C\backslash\mathcal{D}_1 \to C$ be defined as:

$$\phi_1(z) = \phi(z) - a_0 \log(z - z_0),$$

so that $\phi_1(z)$ has the expansion:

$$\phi_1(z) = \sum_{l=0}^{\infty} \beta_l z^l,$$

and therefore:

$$S_2 = \left|\phi_1(z) - \sum_{l=0}^{p} \beta_l z^l\right| \quad = \quad \left|\sum_{l=p+1}^{\infty} \beta_l z^l\right| \leq \frac{M}{1 - \frac{|z|}{s}}\left(\frac{|z|}{s}\right)^{p+1},$$

with $M = \overset{max}{C}\ \phi_1(t)$, where we have used a well-known theorem for the truncation of the Taylor expansion [162]. Now, for any t lying on the circle C:

$$\phi_1(t) \geq \sum_{k=1}^{\infty} \left|\frac{a_k}{(t - z_0)^k}\right|.$$

Obviously,

$$|a_k| \leq AR^k \text{ and } |t - z_0| \geq R + cR - s = R(c+1) - cR(p-1)/pR + cR/p,$$

so that:

$$\begin{aligned}\phi_1(t) &\leq \sum_{k=1}^{\infty} \frac{AR^k}{(R(c+p)/p)^k} = A\sum_{k=1}^{\infty}\left(\frac{p}{c+p}\right)^k \\ &= A\frac{p}{c+p}\frac{1}{1-\dfrac{p}{p+c}} = \frac{Ap}{c}.\end{aligned}$$

Furthermore,

$$1-|z|/s = (s-|z|)/s \geq (s-R)s \geq (cR-R)/(cR+R) = (c-1)/(c+1).$$

Collecting all terms, we find:

$$\begin{aligned}S_2 &\leq A\left(\frac{p}{c}\right)\left(\frac{c+1}{c-1}\right)\left(\frac{|z|}{cR}\right)^{p+1}\left(\frac{p}{p-1}\right)^{p+1} \\ &\leq A\left(\frac{p}{c}\right)\left(\frac{c+1}{c-1}\right)\left(\frac{1}{c}\right)^{p+1}\left(1+\frac{1}{p-1}\right)^{p+1} \\ &= A\left(\frac{p}{c}\right)\left(\frac{c+1}{c-1}\right)\left(\frac{1}{c}\right)^{p+1}\left(1+\frac{1}{p-1}\right)^{p-1}\left(1+\frac{1}{p-1}\right)^{2} \\ &< 4eA\left(\frac{p}{c}\right)\left(\frac{c+1}{c-1}\right)\left(\frac{1}{c}\right)^{p+1},\end{aligned}$$

where we have used that $(1+1/n)^n < e$ and $(1+1/(p-1))^2 \leq 4$ for integers $p > 1$. Therefore,

$$\begin{aligned}S_1+S_2 &< \frac{A}{c-1}\left(\frac{1}{c}\right)^{p} + 4eA\left(\frac{p}{c}\right)\left(\frac{c+1}{c-1}\right)\left(\frac{1}{c}\right)^{p+1} \\ &= \frac{A}{c(c-1)}\left(\frac{1}{c}\right)^{p+1}\left(c^2+4p(c+1)e\right).\end{aligned}$$

Hence an upper bound for the error upon truncating equation (5.7.19) is given by:

$$\left|\phi(z)-\sum_{l=1}^{p} b_l z^l\right| < \frac{A}{c(c-1)}\left(\frac{1}{c}\right)^{p+1}\left(c^2+4p(c+1)e\right). \tag{5.7.23}$$

Finally, we include the binomial expansion, which reads:

$$\sum_{k=0}^{n} a_k(z-z_0)^k = \sum_{l=0}^{n}\left[\sum_{k=l}^{n} a_k \left(\begin{array}{c} k \\ l \end{array}\right)(-z_0)^{k-l}\right] z^l. \tag{5.7.24}$$

We are now in a position to derive an algorithm to evaluate the electrostatic energy of a system of N charges in $\mathcal{O}(N)$ operations.

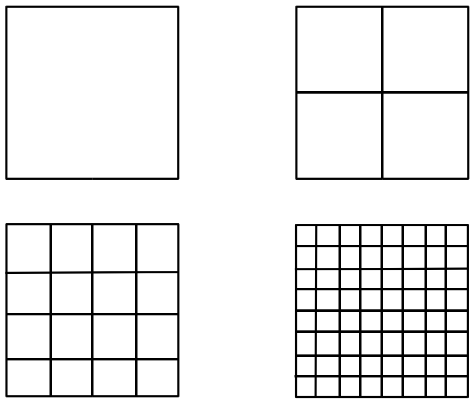


Figure 5.7.8: The computational box and three levels of refinement

5.7.3 The Fast Multipole Algorithm

Consider a set of charges in a computational cell, which we take as a square of side one, and consider "free space boundary conditions", i.e., the boundary can be ignored. We will apply the expansions and shifting formulae derived in the previous section to devise an algorithm for the rapid evaluation of the electrostatic interactions and forces (in 2D).

The strategy is to *cluster* particles at various length scales. The interactions between clusters that are sufficiently far away are evaluated by means of the multipole expansion. Interactions between particles nearby are handled directly. To be precise we evaluate the interactions between sets that are well separated by the multipole expansion. Hence, the bound (5.7.12) with $c = 2$ applies and we can set $p = -\log_2 \epsilon$ to obtain a desired precision ϵ. To implement this strategy *a hierarchy of meshes* is introduced, that refines the computational box into smaller and smaller regions, as shown in Figure 5.7.8. Mesh level $l = 0$ composes the entire computational cell and mesh level $l + 1$ is obtained from mesh level l by subdividing each cell into four equivalent subcells. The number of distinct cells at mesh level l is therefore 4^l. For a given cell, say i, at level l the four cells obtained from i at level $l + 1$ are referred to as its *children*, while i is the *parent* of the four cells at level $l + 1$.

Other notation referred to in the description of the algorithm is:

$\Phi_{l,i}$ is the *$p-$term multipole expansion* (about the cell center) of the potential field created by particles *inside* cell i at level l.

$\Psi_{l,i}$ is the $p-$term expansion about the center of the cell of cell i at level l, describing the potential field due to all particles outside the cell and its nearest

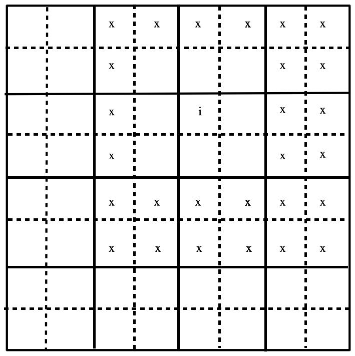


Figure 5.7.9: Interaction list of cell i. The solid lines correspond to mesh level 2 and the broken lines to mesh level 3. The interaction list of cell i is formed by the cells marked x, which are well separated from cell i, but contained within the nearest neighbors of i's parent.

neighbors.

$\tilde{\Psi}_{l,i}$ is the $p-$term expansion about the center of cell i at level l, describing the potential field due to all particles outside i's *parent* cell and the *parent* cell's nearest neighbors.

Interaction list for cell i at level l is the set of cells that are children of the nearest neighbors of i's parent and that are well separated from box i, as shown in Figure 5.7.9.

Suppose that at level $l-1$ we have a local expansion $\Psi_{l-1,i}$ around the center of cell i. Thus, we have the coefficients $b_{n,i}^{l-1}$ in the expansion:

$$\Psi_{l-1,i}(z) = \sum_{n=0}^{p} b_{n,i}^{l-1} \left(z - z_{0,i}^{l-1}\right)^n . \tag{5.7.25}$$

Then, the centers of the children of i at level l are located at $z_{0,i}^{l-1} + \omega_\alpha^l$, with $\alpha = 1, \cdots, 4$ and $\omega_\alpha^l = \pm 1/(2l) \pm \imath/(2l)$. Let $v = z - z_{0,i}^{l-1} - \omega_\alpha^l$. By the binomial theorem 5.7.24, it follows that

$$\begin{aligned} \Psi_{l-1,i} &= \sum_{n=0}^{p} b_{n,i}^{l-1} \left(v + \omega_\alpha^l\right)^n \\ &= \sum_{m=0}^{p} \left(\sum_{n=m}^{p} b_{n,i}^{l-1} \begin{pmatrix} n \\ m \end{pmatrix} \left(\omega_\alpha^l\right)^{n-m} \right) v^m, \end{aligned} \tag{5.7.26}$$

which is a local series expansion about the center of the α-th child of cell i at level $l-1$ due to the charges outside its parent and the neighbors of its parent. Let this cell be labeled j at level l, then we have the potential $\tilde{\Psi}_{l,j}$. Note that to obtain $\Psi_{l,j}$ we have to add the contribution of the charges in the cells contained in the *interaction list* of j at level l. This can be done by application of (5.7.19) to transform the multipole expansions of the particles in the cells contained in the interaction list of cell j into a local expansion about the center of j and by adding these to $\tilde{\Psi}_{l,j}$. Note also that with free space boundary conditions $\Psi_{0,i}$ and $\Psi_{1,i}$ are zero, as there are no well separated cells at these levels. Thus, we begin by forming local expansions at level 2. We are now in a position to describe the complete algorithm. Comments are in italics.

FM Algorithm:

Initialization

```
  Choose a level of refinement $n \approx \log_4(N)$ and a precision $\epsilon$.
  Set $p \approx \log_2(\epsilon)$.
```

Upward pass:

```
Step 1
Form multipole expansions about the centers of the potential field in each
cell due to the particles in the cell about the cell center at the finest level
do $icell = 1, \cdots, 4^n$
    Form a p-term multipole expansion $\Phi_{n,icell}$ about the cell center
    using (5.7.8)
enddo
```

```
Step 2
Form multipole expansions about the centers of all cells at all coarser levels
do $l = n-1, \cdots, 0$
    do $icell = 1, \cdots, 4^l$
        Form a p-term multipole expansion $\Phi_{l,icell}$ by using the shifting
        formula (5.7.15) to shift the center of each child cell's expansion
        to the current cell center and adding these together.
    enddo
enddo
```

Downward pass:

In the downward pass interactions are consistently computed at the coarsest possible level. For a given cell this is accomplished by including interactions with cells that are well separated and whose interactions have not yet been accounted for.

```
Step 3
Form a local expansion about the center of each cell at each mesh level
$l \le n-1$. This local expansion describes the field due to all particles in the
system that are not contained in the current box or its nearest neighbors. In
the second loop this expansion is shifted to the centers of the cell's children,
forming the initial expansion for the box at the next level.
Set $\tilde{\Psi}_{1,\alpha} = 0; \alpha = 1, \cdots, 4$.
do $l = 1, \cdots, n-1$
    do $icell = 1, \cdots, 4^l$
      Form $\Psi_{l,icell}$ using (5.7.19) to convert the multipole expansion $\Phi_{l,j}$ of
      cell $j$ in the interaction list of cell $icell$ to a local expansion
      about the center of cell $icell$ and add these to $\tilde{\Psi}_{l,icell}$.
    enddo
    do $icell = 1, \cdots, 4^l$
      Form the expansion $\tilde{\Psi}_{l+1,j}$ for $icell$'s children using the binomial
      expansion (5.7.24) to expand $\Psi_{l,icell}$ about the centers of the children $j$
      of $icell$.
    enddo
enddo
```

```
Step 4
Compute interactions at the finest mesh level.
do $icell = 1, \cdots, 4^n$
        Repeat the first loop of step 3.
enddo
```

We now have local expansions at the finest mesh level.

```
**Step 5**
Evaluate local expansions and their derivative (for the force) at the particle positions.
do $icell = 1, \cdots, 4^n$
    For each particle in the cell $icell$ at position $z_k$ evaluate $\Phi_{icell,n}(z)$
    and $d\Phi_{icell,n}(z)/dz$ for $z = z_k$.
enddo
```

```
**Step 6**
Compute direct interactions and forces.
do $icell = 1, \cdots, 4^n$
    For each particle in cell $icell$ compute the interactions and forces with
    all other particles in the cell and its nearest neighbors.
    Add direct interactions and forces to the far-field terms.
enddo
```

If we assume that the particles are reasonably homogeneously distributed over the simulation cell it is seen that the computational cost of each step in the algorithm scales linearly with the number of particles N in the system. Careful analysis shows that the running time T_r for evaluation of the interactions obeys a linear scaling:

$$T_r = N\left(-2a\log_2(\epsilon) + 56b\left(\log_2(\epsilon)\right)^2 + dn_{max} + e\right). \qquad (5.7.27)$$

Here a, b, c, d are constants, whose value depends on the compiler, machine and so on, and n_{max} is an upper bound for the number of particles that can reside in a cell at the finest mesh level. Similarly the storage requirements M are linear in the number of particles:

$$M = (\alpha - \beta\log_2(\epsilon))\, N, \qquad (5.7.28)$$

where, again, α, β are determined by computer system, compiler and so on. Even if the distribution is highly nonhomogeneous an adaptive version of the algorithm, although more complex, retains the linear scaling behavior.

5.7.4 Lattice Sums

In computer simulations of electrostatic systems periodic boundary conditions are used extensively. Until the 1990's the standard approach was the use of lattice sums, i.e. one evaluates (5.2.1) for the Coulomb potential. A standard approach uses Jacobi's imaginary transformation for θ-functions [176] and the

result is known as the Ewald transformation [156]. The slowly (*semi-*)convergent series is transformed into a sum of rapidly convergent series suitable for numerical evaluation on a computer. In this section we apply this procedure to the two-dimensional Coulomb system, interacting through the logarithmic potential (5.7.4).
We consider a square cell of unit length containing N charges $q_1, \cdots, q_N$ at positions $\mathbf{r}_1, \cdots, \mathbf{r}_N$. The square cell is periodically repeated as discussed in Subsection 5.2. Charge neutrality is imposed on the system to ensure that the lattice sum (5.2.1) has a finite answer, i.e. we have that

$$\sum_{\alpha=1}^{N} q_\alpha = 0. \tag{5.7.29}$$

The electrostatic energy of the cell then reads,

$$U(\mathbf{r}_1, \cdots, \mathbf{r}_N) = -\frac{1}{2}{\sum_{\mathbf{n}}}' \sum_{\alpha=1}^{N} \sum_{\beta=1}^{N} q_\alpha q_\beta \log\left(|\mathbf{r}_\alpha - \mathbf{r}_\beta + \mathbf{n}|\right). \tag{5.7.30}$$

The sum over $\mathbf{n}$ is over all pairs of integers (n_1, n_2). Note that this series is conditionally convergent when the charge neutrality condition is obeyed and diverges otherwise. The prime in the summation over $\mathbf{n}$ denotes that when $\mathbf{n} = (0,0)$ the term $\alpha = \beta$ has to be omitted. We now use the identity:

$$\int_0^\infty \sum_{i=1}^{N} q_i e^{-r_i^2 t} \frac{dt}{t} = -\sum_{i=1}^{N} q_i \log r_i^2, \tag{5.7.31}$$

valid for a charge-neutral system. Applying this to the lattice sum (5.7.30) we have that

$$U(\mathbf{r}_1, \cdots, \mathbf{r}_N) = \frac{1}{4}{\sum_{\mathbf{n}}}' \int_0^\infty \sum_{\alpha=1}^{N} \sum_{\beta=1}^{N} q_\alpha q_\beta exp\left(-t\left(\mathbf{r}_\alpha - \mathbf{r}_\beta + \mathbf{n}\right)^2\right). \tag{5.7.32}$$

The integral is split up into two regions:

$$\int_0^\infty (\cdots)dt = \int_0^{\kappa^2} (\cdots)dt + \int_{\kappa^2}^\infty (\cdots)dt,$$

to obtain

$$U(\mathbf{r}_1, \cdots, \mathbf{r}_N) = U_I + U_{II}.$$

The parameter κ can be chosen to give optimal convergence properties to the final result. The sum U_{II}, involving $\int_{\kappa^2}^\infty dt$, is absolutely convergent. We have

$$\int_{\kappa^2}^{\infty} exp(-r^2 t)\frac{dt}{t} = E_1(\kappa^2 r^2). \qquad (5.7.33)$$

Here $E_1(z)$ is the exponential integral function

$$E_1(z) = \int_z^{\infty} \frac{e^{-t}}{t} dt.$$

We remove the prime on the sums over $\mathbf{n}$ by adding the term $\alpha = \beta$ in (5.7.30) and by subtracting it in U_{II}. Using

$$\lim_{r\to 0} \frac{1}{2} E_1(\kappa^2 r^2) + log(r) = -\frac{1}{2}(\gamma + \log(\kappa^2)), \qquad (5.7.34)$$

where $\gamma = 0.5772\cdots$ is Euler's constant, we readily find:

$$U_{II} = \frac{1}{4}{\sum_{\mathbf{n}}}' \sum_{\alpha=1}^{N} \sum_{\beta=1}^{N} q_\alpha q_\beta E_1\left(\kappa^2 \left|\mathbf{r}_\alpha - \mathbf{r}_\beta + \mathbf{n}\right|^2\right) - \frac{1}{4}\left(\gamma + \log\left(\kappa^2\right)\right) \sum_{\alpha=1}^{N} q_\alpha^2, \qquad (5.7.35)$$

where the term proportional to $\sum q_\alpha^2$ arises from a partial cancellation of the divergences of $E_1(\kappa^2 r^2)$ and $log(r)$ as $r \to 0$.
To obtain a rapidly convergent expansion for U_I we consider the sum

$$S(\mathbf{r}) = \sum_{\mathbf{n}} exp(-|\mathbf{r}+\mathbf{n}|^2 t).$$

Obviously, $S(\mathbf{r})$ is a periodic function in $\mathbf{r}$ with unit period in $x-$ and $y-$directions, so that we may expand it in a Fourier series:

$$S(\mathbf{r}) = \sum_{\mathbf{k}} A(\mathbf{k}) e^{2\pi\imath \mathbf{k}\cdot\mathbf{r}},$$

with $\mathbf{k}$ a vector in the reciprocal lattice, which in the present case coincides with the real space lattice, and

$$\begin{aligned}
A(\mathbf{k}) &= \int_{\mathcal{A}} d\mathbf{r} \sum_{\mathbf{n}} exp\left(-\left(\mathbf{r}+\mathbf{n}\right)^2 t\right) e^{-2\pi\imath\mathbf{k}\cdot\mathbf{r}} \\
&= \int_{\mathcal{R}} d\mathbf{r}\, exp\left(-r^2 t - 2\pi\imath \mathbf{k}\cdot\mathbf{r}\right) \\
&= \int_{\mathcal{R}} d\mathbf{r}\, exp\left(-\left(\mathbf{r} + \frac{\pi\imath\mathbf{k}}{t}\right)^2 t - \frac{\pi^2 k^2}{t}\right) \\
&= exp\left(-\frac{\pi^2 k^2}{t}\right) \int_{\mathcal{R}} d\mathbf{r}\, exp(-r^2 t) \\
&= \left(\frac{\pi}{t}\right) exp\left(-\frac{\pi^2 k^2}{t}\right). \qquad (5.7.36)
\end{aligned}$$

The first integral is over the unit square $\mathcal{A}$. The sum over $\mathbf{n}$ tessellates the plane $\mathcal{R}$ with unit squares and, since the complex exponential is periodic with the same unit cell, we may remove the sum and integrate over the whole plane $\mathcal{R}$ instead. The Gaussian integral is a standard integral.
Hence we have that:

$$\sum_{\mathbf{n}} exp\left(-\left|\mathbf{r}+\mathbf{n}\right|^2 t\right) = \left(\frac{\pi}{t}\right) \sum_{\mathbf{k}} exp\left(-\frac{\pi^2 k^2}{t}\right) exp\left(2\pi\imath \mathbf{k}\cdot\mathbf{r}\right). \tag{5.7.37}$$

Equation (5.7.37) is in essence Jacobi's imaginary transformation for θ functions applied twice to the sum $S(\mathbf{r})$, once for each direction in space.
Consequently we may write[1]

$$\begin{aligned} U_I &= \frac{1}{4}\sum_{\alpha=1}^{N}\sum_{\beta=1}^{N} q_\alpha q_\beta \int_0^{\kappa^2} dt \frac{\pi}{t^2} \sum_{\mathbf{k}} exp\left(-\frac{\pi^2 k^2}{t}\right) e^{2\pi\imath \mathbf{k}\cdot\mathbf{r}_{\alpha\beta}} \\ &= \frac{1}{4\pi}\sum_{\alpha=1}^{N}\sum_{\beta=1}^{N} q_\alpha q_\beta \sum_{\mathbf{k}} \frac{exp\left(-\frac{\pi^2 k^2}{\kappa^2}\right)}{k^2} e^{2\pi\imath \mathbf{k}\cdot\mathbf{r}_{\alpha\beta}} \\ &= \frac{1}{4\pi}\sum_{\mathbf{k}} \frac{exp\left(-\frac{\pi^2 k^2}{\kappa^2}\right)}{k^2} \sum_{\alpha=1}^{N}\sum_{\beta=1}^{N} q_\alpha q_\beta e^{2\pi\imath \mathbf{k}\cdot\mathbf{r}_{\alpha\beta}} \\ &= \frac{1}{4\pi}\sum_{\mathbf{k}} \frac{exp\left(-\frac{\pi^2 k^2}{\kappa^2}\right)}{k^2} \left|\sum_{\alpha=1}^{N} q_\alpha e^{2\pi\imath \mathbf{k}\cdot\mathbf{r}_\alpha}\right|^2. \end{aligned} \tag{5.7.38}$$

The first integral is easily solved by the substitution $u = 1/t$.
Combining (5.7.35) and (5.7.38), we find for the energy:

$$\begin{aligned} U(\mathbf{r}_1,\cdots,\mathbf{r}_N) &= \frac{1}{4\pi}\sum_{\mathbf{k}} \frac{exp\left(-\frac{\pi^2 k^2}{\kappa^2}\right)}{k^2} \left|\sum_{\alpha=1}^{N} q_\alpha e^{2\pi\imath \mathbf{k}\cdot\mathbf{r}_\alpha}\right|^2 - \frac{1}{4}\left(\gamma + \log\left(\kappa^2\right)\right)\sum_{\alpha=1}^{N} q_\alpha^2 + \\ & \frac{1}{4}{\sum_{\mathbf{n}}}' \sum_{\alpha=1}^{N}\sum_{\beta=1}^{N} q_\alpha q_\beta E_1\left(\kappa^2 \left|\mathbf{r}_\alpha - \mathbf{r}_\beta + \mathbf{n}\right|^2\right). \end{aligned} \tag{5.7.39}$$

The energy is now expressed as a sum of two rapidly convergent series. The value of the parameter κ has no effect on the result but opposite effects on the

[1]Here we interchange the order of summation and integration. Since the series is only conditionally convergent this requires a rigorous justification. For full details, see [167]. The answer we arrive at is the one customarily used in computer simulations and corresponds to so-called *tinfoil* boundary conditions.

convergence of each series. Hence an optimum value should be chosen. Note also that U_{II} contains a double summation over all the particles whereas U_I only requires a single sum over particles. Hence the evaluation of the reciprocal space part of the energy is generally much faster than the real space part. To find a optimum value for κ, let us suppose we desire an accuracy δ and we wish to cut-off the real space interaction at $r_c < \frac{1}{2}$. We can then use a linked-cell technique or neighbour tables to evaluate this part in $\mathcal{O}(N)$ operations. In that case κ is chosen such that

$$E_1(\kappa^2 r_c^2) \ll \delta.$$

For large values of its argument the exponential function behaves like

$$E_1(z) \approx \frac{e^{-z}}{z}.$$

Hence, when neglecting some smaller terms, κ should be chosen somewhat larger than $\sqrt{-4\log(\delta)}/r_c$. Now consider the reciprocal space part. For consistency we have to truncate the series at a value k_{max}, such that

$$\frac{exp\left(-\frac{\pi^2 k_{max}^2}{\kappa^2}\right)}{k_{max}^2} \ll \delta,$$

which requires k_{max} to be somewhat larger than $\kappa\sqrt{-\log(\delta)}/pi$. The total number of operations to evaluate the real space part is $\propto r_c^2 N^2$, since the number of interactions to be evaluated for each particle is proportional to the number of particles within an area of the size r_c^2. Similarly, the number of operations to evaluate the reciprocal space sum is $\propto N\pi k_{max}^2$, since the number of reciprocal lattice vectors is proportional to the area of a circle with radius k_{max}. Hence the total time T_r for the evaluation of the electrostatic energy is given by an expression of the form:

$$T_r = \left(\frac{AN^2}{\kappa^2} + BN\kappa^2\right)(-log\delta)\,. \tag{5.7.40}$$

In (5.7.40) we have also made explicit the (logarithmic) dependence of the total time on the accuracy. A and B are again constants depending on the implementation, machine, compiler and so on. Minimizing equation (5.7.40) with respect to κ shows that the optimal value of κ scales with the number of particles as $N^{\frac{1}{4}}$ and the running time for the algorithm scales as $N^{\frac{3}{2}}$. The Greengard-Rokhlin algorithm scales linearly with the number of particles and is therefore more efficient for large numbers of particles. In practice, the Ewald sum (5.7.39) is often used for smaller numbers of particles, up to several thousands. For larger numbers of particles a linear scaling algorithm is to be preferred. The Greengard-Rokhlin algorithm is not the only linear scaling algorithm. Alternatives, such as the particle-particle particle mesh (PPPM) method, utilize Fourier transform techniques.

6 Parallel Computing for Scientific Applications

6.1 Introduction

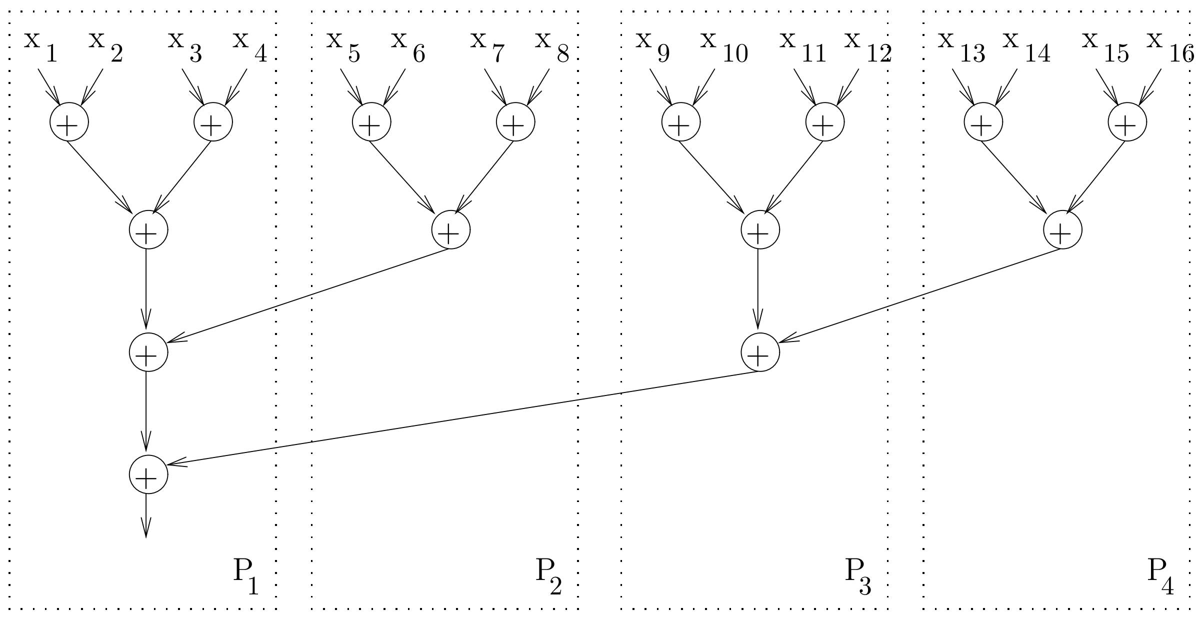
Since the introduction of the first electronic computer in the nineteen forties, the computing speed of a single processor has increased by more than nine orders of magnitude in five decades. However, the demand for faster computers has not decreased. Instead, the increased available computing capacity often stimulates the development of larger and more accurate computing models for the simulation and optimization of increasingly more complex problems. It is widely argued that the physical laws set fundamental limitations on the computational speed of a single processor computer:

- The finite speed of light constraints the propagation speed of signals,
- The Heisenberg's uncertainty principle constraints the minimal width on an integrated circuit and therefore the minimum distance a signal must propagate,

These two lead to the conclusion that the theoretical upper limit of a single processor is a few giga floating point operations per second (Gflops) (e.g. [31]). It is argued that the clock speed of the fastest processors is approximately O(10) GHz. The discussion on how fast a single processor can be is not yet concluded, some argue we might eventually arrive at a clock speed much higher than O(10) GHz through new technology. However, the demand for the ever greater computing power requires an alternative rather than "wait for the processor to get fast enough".

Parallel processing, by putting the computing power of a number of processors together, provides a way to by-pass the physical limits of the sequential computer. Another argument supporting the trend away from sequential towards (massively) parallel computation is that parallel processing is a more economic way to build computers with high computing speed. As opposed to many (vector) supercomputers which rely on the most advanced and therefore expensive technologies, parallel computers can be built by using off-the-shelf technology so that high-performance computers with a better price/performance ratio can be obtained. This trend is not only visible in today's large parallel computer market, it also happens in the area of servers and workstations (e.g. SGI's Power Challenge and SUN's Ultra Enterprise Servers), and PCs with multiple processors are emerging. Of course, there is yet another huge potential of parallel processing power. Workstations and PC's interconnected by high-speed networks form a large parallel system. An example is the Beowulf system built with a cluster of PCs.

The effective and efficient use of parallel systems requires a non-trivial transformation from sequential computing techniques to parallel computing techniques.



x_1 x_2 x_3 x_4
x_5 x_6 x_7 x_8
x_9 x_{10} x_{11} x_{12}
x_{13} x_{14} x_{15} x_{16}
P_1
P_2
P_3
P_4

Figure 6.1.1: Illustration of the summation of n=16 numbers by p=4 processors.

This chapter aims at systematically introducing all the aspects of parallel computer architectures, designing parallel algorithms and parallel programming techniques in a fundamental and yet practical manner.

6.1.1 What Is Parallel Computing ?

Consider the problem of summing n numbers x_i, i.e. calculating $S = \sum_{i=1}^{n} x_i$. Suppose that we have p processors available for the summation. The n numbers are divided into p groups each of n/p numbers (assuming n is divisible by p), and each processor sums up a group of n/p numbers. This takes a time of $n/p - 1$ additions. Finally, the p partial sums are summed recursively in a pair-wise manner (Fig. 6.1.1). This will take a time of $log_2(p)$ additions. In total the summation takes $n/p + log_2(p) - 1$ additions as compared to $n - 1$ additions using a single processor. Therefore, in terms of the number of additions, the calculation time is reduced by a factor of nearly p for $n \gg p$. However, there are extra overhead introduced due to the requirement of communicating the partial sums between the processors.

An analogy can be drawn to a real life situation. For example, consider a team working on a large engineering design project. The design work is divided into a number of sub-designs, and then each team member makes a sub-design. Finally all these sub-designs are verified and put together into the final design. In this design process, we can identify a number of phases. First, the design project

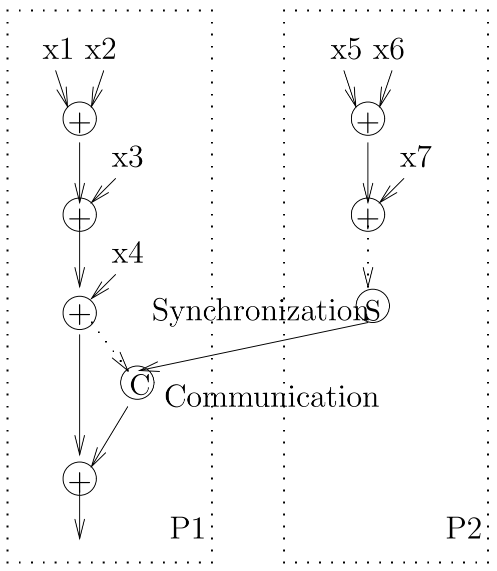


Figure 6.1.2: Illustration of synchronization and communication.

has to be divided into a number of sub-designs (*partitioning*). Then, the work of one or more sub-designs will be assigned to each individual team member (*task assignment*, or *mapping*).

Generally, a good design team can make a complex design much faster than a single designer. Even a super designer (expensive) cannot beat an organized team of average (cheaper) designers. That is the power of parallel processing! However, we should recognize that besides all those advantages of concurrent engineering, it also bears additional overhead inherently. First, there must be *communication* between the team members in order to fit the sub-designs into the final design. Second, the sub-designs may depend on each others completion. Therefore it can happen that during the design process, some team member has to wait for the completion of the sub-design of another team member before he/she can continue (*synchronization*). This is the extra overhead introduced by 'parallel processing'.

Fig. 6.1.2 illustrates the synchronization and communication in a parallel summation of 7 numbers by 2 identical processors. Processor 2 finishes the addition of 3 numbers (x_5,x_6,x_7) before processor 1 finishes. Hence, processor 2 must wait for processor 1 to do its fourth addition (synchronization). Then, processor 2 sends the sum of 3 numbers to processor 1 (communication). Processor 1 receives the result from processor 2 and adds it to its own result to produce the final result.

6.1.2 The Need for Parallel Processing

The advent of computer technology and simulation methods has a great impact on the direction of scientific research. It is becoming more and more evident that computer simulation is now a third approach[2] besides the theoretical and experimental approach. It is being increasingly recognized by industry that computer simulation is an attractive alternative to conventional experimental tests. For example, in airplane design the computer can be viewed as an "electronic wind tunnel". In many areas of scientific research, a powerful computer is an instrument like a powerful telescope is for astrophysics discoveries.

Although the computing speed of computers has increased significantly since the birth of the first electronic computer, the demand for more computing power has not diminished. For some scientific and industrial computer applications it is estimated that they require computers with terabytes of total memory and with more than teraflops performance. For example, in a report [59] prepared by the Committee on Physical, Mathematical, and Engineering Sciences (U.S.A.), a large number of so-called "Grand Challenge" problems (Fig 6.1.3) have been identified. The solution of these problems requires a computing power beyond the teraflops. Furthermore, many computer aided modeling, design, and optimization problems demand an increasing computing power in order to provide interactive facilities.

6.1.3 Parallelism in Nature

What is parallelism? *Parallelism* is the measure for the number of tasks that are or can be performed concurrently. So, we say the parallelism of a program (execution) is high if the number of tasks that are being executed is large, and a problem with a high degree of parallelism means that the problem has a large number of tasks that can be performed in parallel (thus the *potential* of being performed in parallel).

Parallelism exists everywhere in the real world. Natural systems, complex technical systems, and even social changes are all highly parallel processes. The growth of a plant is simultaneously influenced by a large number of factors. The starting of a motor requires the coordination of many components. On the stock market, share prices depend on the positions of thousands of purchasers and sellers at the same market as well as other stock markets around the world.

The human brain works in parallel as well. Figure 6.1.4 shows a comparison between a sequential von Neumann computer and a brain which consists of a (natural) network of *neurons*. A neuron is a device which can generate outputs to many other neurons using inputs from many neurons. Some neurons have up to 100,000 such inputs via an antenna-like structure referred to as a *dendrite*. The

[2]Today, computational physics, computational chemistry, and computational mathematics are important studies in the department of physics, chemistry and mathematics, respectively.

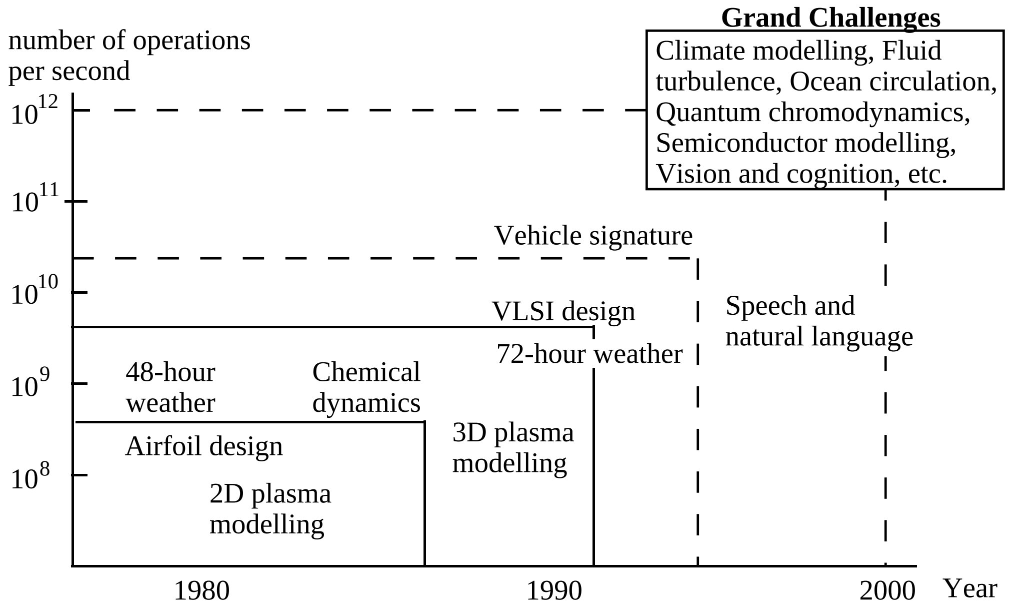


Figure 6.1.3: Illustration of the estimated demand for computing power by a number of "Grand Challenge" problems [2].

brain can be viewed as a parallel computer where many neurons simultaneously perform a variety of memory-functions, sensory, pattern recognition, control and reasoning. It is a flexible, powerful parallel computer which is in many cases the most sophisticated implementation of parallel processing yet discovered. The specific algorithms used by the brain are the subject of intense theoretical and experimental study and are outside the scope of this text. One thing to be mentioned here is that the power of the brain lies more in its remarkable multiplicity of connections, rather than in the speed of individual neurons. Electronic circuits (IC chips) run at a cycle time of from 10^{-8} to 10^{-10} seconds. Neurons operate at a maximum frequency of about 10^3 hertz, or a cycle time of 10^{-3} seconds. However, neurons may have as many as 100,000 connections, in contrast to today's parallel computers which typically have less than 10 connections per processor.

The idea of parallel computing dates back even before the birth of the first electronic computer. In 1922, L.F. Richardson wrote in his book *Weather Prediction by Numerical Process* [114] the idea of using a number of individuals to perform the calculation in the weather prediction, see Fig. 6.1.5. In order to make the calculation fast enough (i.e. ahead of the real weather development), he proposed to divide the globe into a number of subspaces and to assign the task of calculations corresponding to each subspace to an individual. He estimates that 64,000 individuals are required for simultaneously performing the calculations. This is a form of parallel computation. Techniques of dividing a domain into a number of smaller parts and for solving them simultaneously are known today

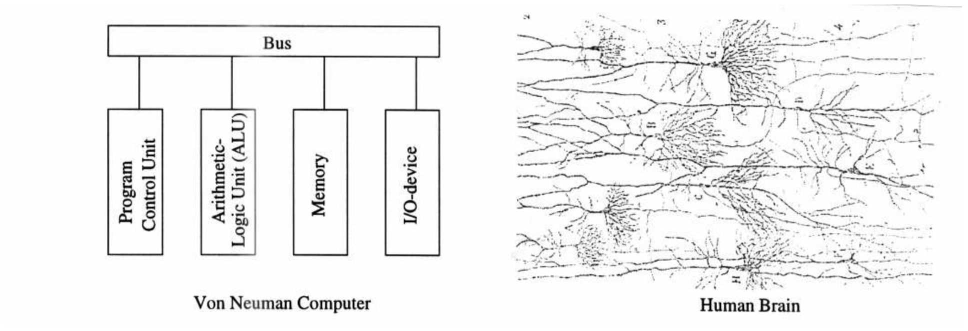


Figure 6.1.4: Von Neumann computer versus brain (neurons).

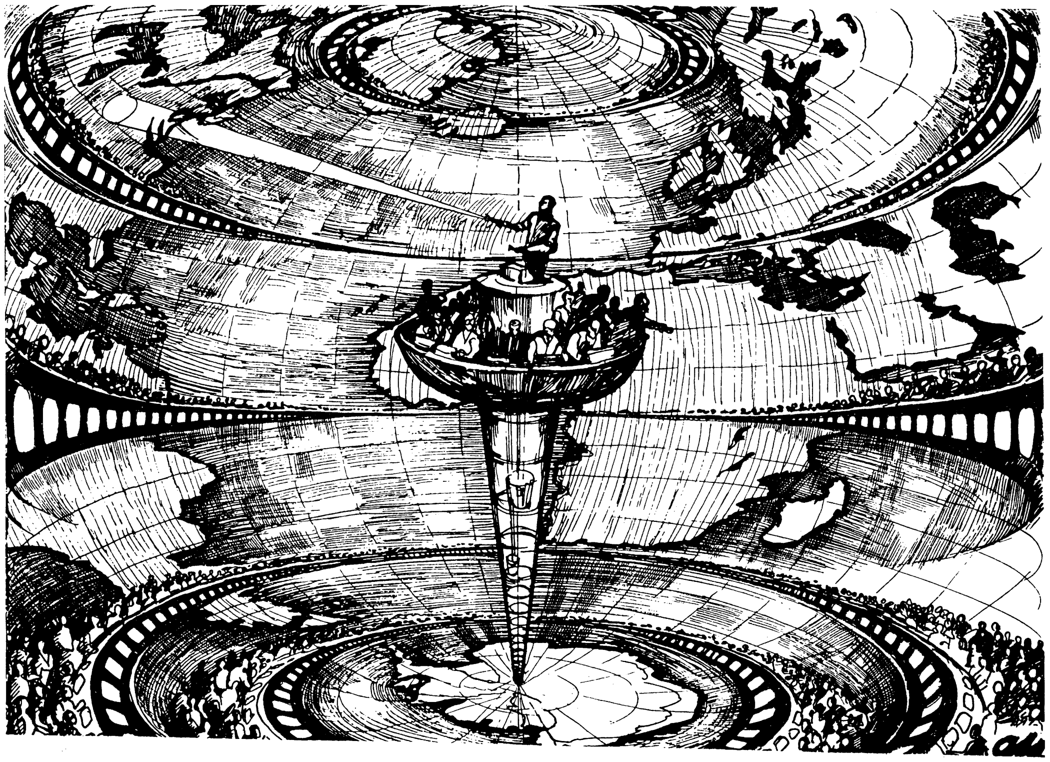


Figure 6.1.5: Richardson's dream of parallel weather prediction.

as *domain decomposition* or in another form *grid partitioning*, which are often applied in parallelizing large scale computer applications.

From the considerations above, we have seen that there is a great potential of inherent parallelism in nature. This promises that many problems to be modeled and solved by computers can be efficiently solved by means of parallel computing. The sequential nature of today's programming practices is simply a historical outcome of the von Neumann computer model which has been very successful. However, its sequential nature is an artificial restriction, a way to organize operations to fit the sequential computer.

The efficient use of parallel computers is often not a trivial transition from the sequential programs and experiences. First of all, many existing programs require a complicated transformation (e.g. loop-transformation or variable renaming or copying) in order to become parallelizable. Secondly, many existing algorithms, which are often optimized for sequential computing (e.g. requiring a minimal number of operations), have a low degree of parallelism or are even inherently sequential. In these cases, new algorithms with a higher degree of parallelism must be developed. Thirdly, many problems are modeled with the 'sequential' computing model in mind so that often inherent parallelism is lost during the modeling steps. Therefore, it is sometimes preferable to remodel the problem and keep as much parallelism in the original problem as possible (*parallel modeling*).

6.1.4 Issues in Parallel Computing

The use of parallel computers introduces a number of new problems which are not present or not important in the traditional use of serial computers. These problems can be distinguished into three categories:

- *parallel hardware* — How to organize and construct computers with up to thousands or more processing elements? The increased number of degrees of freedom in a parallel architecture makes the design choices more complicated. The uncertainty of which architecture is the best one (which is dependent on the state of technology) has led to a diversity of existing parallel computers.

- *parallel algorithms* — A parallel computer is of little use unless efficient parallel algorithms are available. The issues in designing parallel algorithms are very different from those in designing their sequential counterparts. As we will see in Section 6.3, the notion of *optimality* of a parallel algorithm is different from that of a sequential algorithm. In addition to the operation-count and memory-size requirement in the complexity analysis of sequential algorithms, the *communication complexity* must also be considered in the analysis of parallel algorithms. Furthermore, the performance of a parallel algorithm depends on the architecture of the parallel computer being used.

- *parallel programming* — System software and application software are dramatically affected by the architectural choices. Parallel programming paradigms and languages which are flexible for adapting solution methods and algorithms, allow efficient implementation and are easy to program must be developed. It is also necessary to develop compilers that can automatically extract implicit parallelism from (sequential) programs. To facilitate the programming of parallel computers, comprehensive programming environments and tools are required. In order to develop efficient and portable parallel programs, software development methods tailored to the construction of parallel software must be developed and applied.

These three topics, parallel hardware, parallel algorithms and parallel programming will be discussed in the following sections of this chapter. Section 6.2 gives an overview of different parallel architectures and classification schemes of parallel computers from different view points are described. Section 6.3 discusses the properties of parallel algorithms. The DAG (Direct Acyclic Graph) is introduced for representing the dependence structure of an algorithm. Section 7.1 gives an introduction to the design and analysis of parallel algorithms with emphasis on numerical algorithms. We also discuss the relation between algorithms and architectures with respect to performance and optimality. Section 6.4 gives an introduction to parallel programming. Some basic issues like parallel languages, communication and synchronization are discussed. The well known message-passing library MPI (Message Passing Interface) and the OpenMP programming system are described.

6.2 Characteristics of Parallel Computers

Since the mid-seventies of last century a large variety of parallel computers has emerged. To name a few: commercial systems, like the Cray-1,2,3, the Cray Y-MP and T3E, the IBM SP2, the NEC SX-4/5/6, the Intel Paragon, the Convex C3 series, the Thinking Machine CM-2 and CM-5, the nCUBE2 of nCUBE Corp., and research prototypes like the Delft Parallel Processor (DPP) [32], the NYU's Ultracomputer [58], the Cedar at University of Illinois [51], the Manchester Data Flow Machine [63] and MIT's Alewife [1]. Some of these vendors (projects) are building new parallel computers with new architectures and some others have ceased their activities (e.g. the CDC Cyber, the Alliant FX and the Thinking Machine). All these machines differ not only in performance and technology, but they also have very different architectural characteristics such as the way in which the interconnection between the various processors and memory units are organized. The most important components in a parallel architecture are: (i) processors, (ii) the interconnection network, and (iii) the memory organization. These three components will be discussed in the following sections.

6.2.1 Vector Processors

A distinction often used for processors is the CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) type of processors (e.g. for workstations). Another important difference between processors is the word lengths where upon they operate (e.g. the processor of the CM-1 is a simple single bit processor, whereas a processor of the Cray C90 consists of many functional units operating in 64-bit word length). However, the most important difference in processors used in parallel computers is between the type of *scalar* (e.g. the Intel 80486 microprocessor) and *vector* processors (e.g. the vector unit on the Cray 1). In the following we will discuss the vector processors.

Two types of parallelism can be exploited in constructing parallel architectures to speed up a group of operations: *spatial parallelism* and *temporal parallelism*. Spatial parallelism corresponds to operations on multiple data items which can be performed simultaneously by a number of replicated processing units. An example of spatial parallelism is the addition of two matrices where all elements of the resulting matrix can be computed in parallel. Temporal parallelism corresponds to a succession of operations on multiple data items. Vector processors exploit temporal parallelism. A *vector processor* or a *pipeline processor* consists of a cascade of “special-purpose” processing units corresponding to the operations in a single succession of operations.

Consider the *addition of two floating-point numbers* that is performed in a number of stages:

a. compare the exponents of the two numbers;

b. equalize the exponents by shifting the mantissa (alignment);

c. add the two mantissas together (fixed-point number addition);

d. make the sum a number in a normalized floating-point number representation (post normalization);

A vector add pipeline consists of four *segments*, each performing the corresponding operation in a stage is shown in Figure 6.2.6. Pipelining techniques enhance the utilization of the hardware components and therefore increase the processing throughput. This can be explained as follows. Assume that the scalar adder has a latency time of α μs, which is the time required for input signals to propagate through the hardware circuit implementing the add-operation. Now if we divide the addition into p stages each can be implemented with a hardware component of latency time α/p. If the result of each stage is stored in a latch, then the clock cycle of this pipeline adder can be reduced by a factor p as compared to the scalar adder (if we ignore the relative small delay induced by the latch). That means both the utilization of each of the hardware components and the throughput of the adder is increased by a factor p.

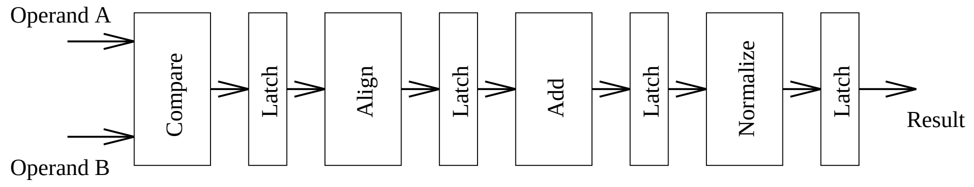


Figure 6.2.6: Illustration of a floating-point addition pipeline.

Figure 6.2.7 illustrates the process of vector processing. The problem is to compute $C(i) = A(i) + B(i)$ for $i = 1, ..., 100$. At the beginning of this vector addition, not all segments are doing useful work (the *start-up* phase), but from the 4-th clock cycle all the segments are busy. At the end (from the 101-th cycle), the segments in the pipeline are gradually becoming empty again (the *drain* phase). Thus, the segments in the pipeline are fully utilized during the vector addition operation, except in the start-up and the drain phases. It is not difficult to see that the efficiency of the vector operation increases with the length of the vector (i.e. the number of data items to be added in this case).

Given a pipeline consisting of p segments and the clock cycle time τ μs, the time required for performing a vector operation with a vector length n is,

$$T_{pipe}(n, p) = ((p - 1) + n) \cdot \tau \tag{6.2.1}$$

For $n \gg p$, $n/T_{pipe} \approx 1/\tau$. Thus, the asymptotic speed (peak performance) of a pipeline processor is $1/\tau$ Mflops. The so-called *half vector length* $n_{\frac{1}{2}}$ is introduced by Hockney and Jesshope [77] as an indicator for vector performance. $n_{\frac{1}{2}}$ is defined as the length of a vector such that the performance of a pipeline processor equals half its peak performance. It can be derived that

$$n_{\frac{1}{2}} = p - 1 \tag{6.2.2}$$

In general a vector operation includes loading (reading) the data items into the so-called vector-registers, performing the arithmetic operation, and writing the results back back to memory. This sequence as a whole can be performed by pipelining. So, the half vector length of a vector computer is usually larger than the number of segments in the arithmetic pipeline. In general, $p \cdot \tau$ in Eq. (6.2.1) should be replaced by the total start up time of a vector operation t_s (including loading the input vector(s) and writing the resultant vector to memory). A detailed analysis of vector processing can be found in [78].

6.2.2 Networks

The interconnection network of a parallel computer is a very important part which connects the processors and one or more memory modules with each other. The

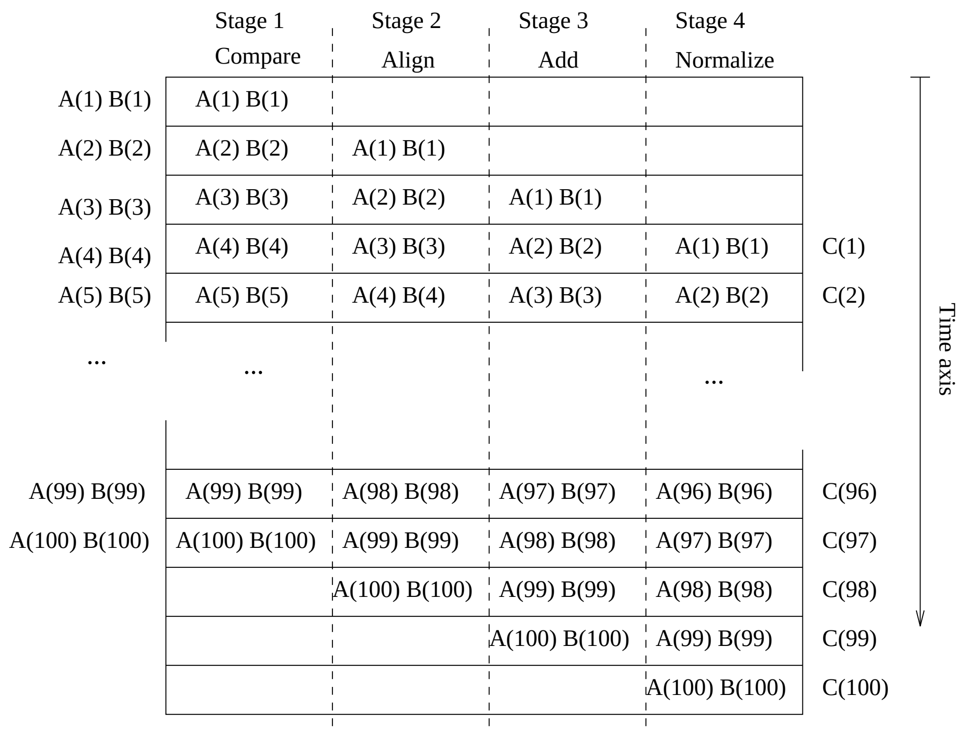


Figure 6.2.7: Illustration of a vector addition operation.

network is used for inter-processor communication, synchronization, etc. The quality of the interconnection network is crucial for the performance of a parallel system. In [6] a measure of the quality of an interconnection network has been put in the following terms: *how quick can it deliver how much of what is needed to the right place reliably and at good cost and value.*

A simplified view on the communication cost of sending a message between two different processors is to split it in two parameters: the message *setup* time t_s, which is the time required to initiate the communication, and the transfer time per word or byte t_w, which is determined by the physical bandwidth of the communication channel linking the source and destination processors. In this way sending a message of length L requires a time $T_{msg}(L) = t_s + t_w \cdot L$. However, in general a message may have to travel through several intermediate processors (hops)between the source and destination. A more general formula for the communication time of sending a message is thus

$$T_{msg}(L) = \text{Setup} + \text{Routing-Delay} + \text{Contention-Delay} + f(t_w, L) \qquad (6.2.3)$$

The *routing delay* is the time to move a given symbol, say the first bit of the message, from the source to the destination, in a network with no other data traffic (which may cause contention). The routing delay is a function of the number of links/channels on the route (called the *routing distance*), the delay incurred at each intermediate processor/switch, and the transmission bandwidth of the links. The *contention delay* occurs in a congested network when more than one message wants to use the same link/channel simultaneously. The function f represents the time that the message occupies the links.

From the above discussion we identify the following important terms for an interconnection network:

a. *Latency or Diameter* — the signal propagation time or the maximum distance between any pair of processors;

b. *Bandwidth* — how much data traffic per time unit (e.g. bytes/s) the network can handle;

c. *Connectivity* — how many immediate neighbors each processor has and how many different paths exist between a pair of processors;

d. *Extensibility* — how well is the network scalable to a larger network while retaining the same performance characteristics.

Network structures and implementations can be divided into *static* and *dynamic* networks. In a static network, processors are connected directly by wires. A static network is often called a *point to point* network. Connections between processors in a dynamic network are made through switches. It is similar to the telephone network where a direct connection from the caller to the callee is set up by means of the switches in telephone centers. A dynamic network is therefore often called a *switching* network. In the following subsections we will discuss these two classes and consider some example implementations.

Point to point networks In a point to point network, a processor can only exchange information with neighboring processors which are *directly connected by physical wires.* Communication between two non-neighbor processors must be realized by routing the message through one or more intermediate neighbor processors. The main characteristic of point to point networks is the topology. In a point to point network, the latency is directly related to the *diameter* of the network, which is the maximum distance between any pair of processors. The bandwidth is related to the *connectivity*, which is a measure of the number of 'disjoint' paths [3] connecting a pair of processors. In Figure 6.2.8 a number of static network topologies is shown. In the following discussion all connections are assumed to be bi-directional unless stated otherwise.

[3]Two paths are disjoint if they have only the begin (source) and end (destination) in common.

Ring Structure The ring structure has only two connections per processor. The ring can be easily enlarged by adding more processors to it while the number of connections per processors remains a constant 2. That means it is scalable in size. However, for a ring with p processors the diameter is $p/2$, so the latency increases proportionally with the size of the ring. Thus it is not scalable in performance.

Mesh and Torus The most simple mesh is the 1-dimensional mesh or linear array. Like a ring, each processor has two connections except the processors at the two end-points. The diameter of a linear array is $p - 1$. So, with respect to extensibility the same conclusion for the ring structure holds for the linear array.

A popular interconnection topology is the *2-dimensional mesh*. In a 2D mesh, each processor has 4 connections except those at the boundaries of the mesh which have fewer connections. The diameter is $2 \cdot p^{\frac{1}{2}} - 2$ (for a square mesh). A variant of the 2D mesh is the so-called 8 way nearest neighbor mesh, where each interior processor has 8 neighbors. The diameter is cut in half through the doubling of the number of connections. In contrast to this, a *3D mesh* has 6 connections per interior processor, however, the diameter is only $3 \cdot p^{\frac{1}{3}} - 3$.

A k-dimensional torus is obtained by connecting the boundary processors by 'folding' each pair of boundaries at the same dimension. So, the 1D torus is the ring structure. The diameter of a k-dimensional torus is half of that of the k-dimensional mesh.

Although the k-dimensional ($k > 1$) meshes and tori have a smaller diameter than the ring and linear array. The diameter can still be very large for a large number of processors. The hypercube is a network topology which further reduces the diameter.

Hypercubes A processor in a hypercube of dimension k has k neighbor processors. The zero dimensional hypercube is a single processor. A $(k+1)$-dimensional hypercube can be constructed from two k-dimension hypercubes by connecting corresponding processors. For example, a 3-dimensional hypercube is constructed by 2 squares (2-dimensional hypercube) by connecting each processor of the 'left' square with the appropriate processor of the 'right' square. A 4-dimensional hypercube is made of two 3-dimensional hypercubes as shown in Figure 6.2.8.

Each processor has $log_2(p)$ connections to neighboring processors, which is not constant as is the case for the ring, mesh and torus. However, the diameter is reduced to $log_2(p)$. The hypercube is therefore a network topology whose diameter remains small even for a quite large number of processors.

Trees Trees are the special kind of topology which has a constant number of connections per processor and a logarithmic diameter. In a binary tree each processor has 3 connections except the root and the 'leaf' processors. The diameter

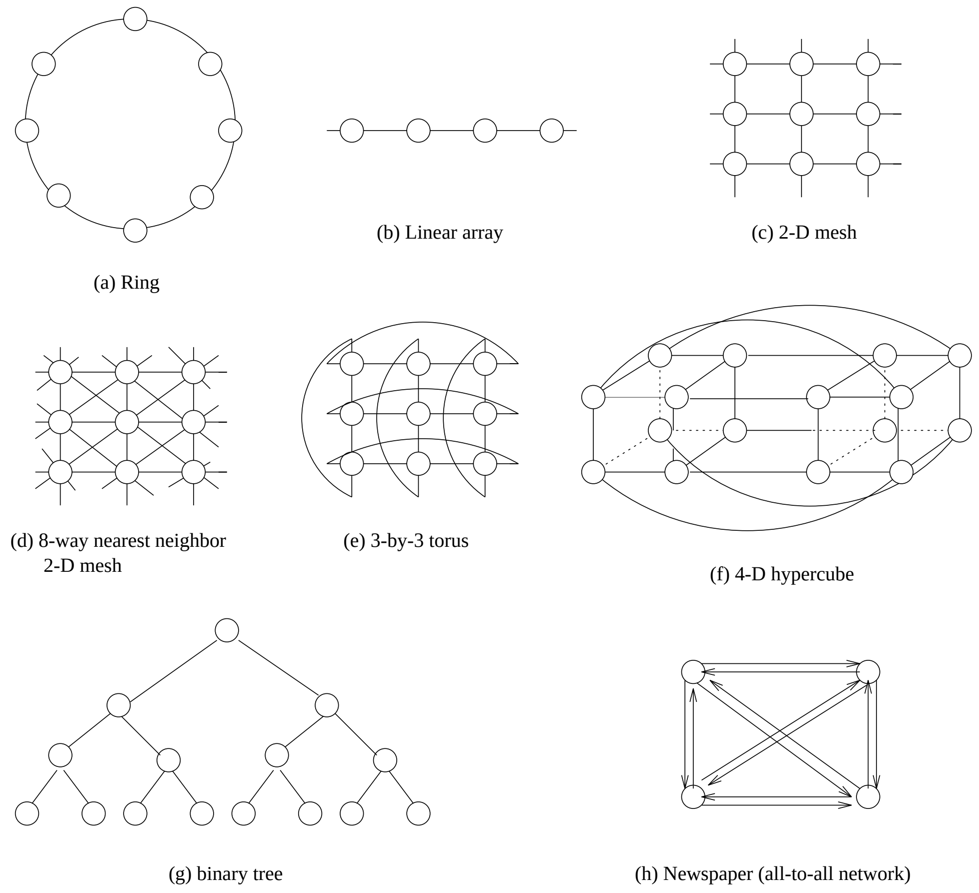


Figure 6.2.8: Point to point network topologies.

is $2 \cdot log_2(p) - 2$. In a quadtree each processor has 4 children and the diameter between two processors is $2 \cdot log_4(p) - 2$. In general, a k-ary tree has k children per processor and the diameter is $2 \cdot log_k(p) - 2$.

The disadvantage of the tree topology is the 'root bottleneck', which allows only 1 information exchange at a time between the two sub-trees. To overcome this bottleneck, the so-called tree-mesh topology has been designed [108]. A tree-mesh combines the advantages of the mesh and the tree. It has a number of disjoint paths between any two processors and a logarithmic diameter $log_2(p)$. Another possible remedy is to construct a 'fat tree', as discussed in Section 6.2.2.

Newpaper The newspaper network is an all-to-all (fully interconnected) network in which each processor is directly connected to all other processors. The diameter is thus 1. Each processor has $2 \cdot (p-1)$ uni-directional connections, $p-1$ connections for the output (sending) and $(p-1)$ for the input (receiving). Unlike the crossbar switching network (see Section 6.2.2), the newspaper network is able to connect all processors to all processors simultaneously. Such a network has been employed in the Delft Parallel Processor (DPP) [32]. The advantage of the newspaper network is that the communications among the processors are uniformly fast and therefore enabling efficient fine-grain parallel computation. The disadvantage is that the number of wires increases with p^2. Perhaps the coming age of opto-electronic technology can provide a high performance and yet affordable newspaper network which scales to hundreds or more processors.

Switching networks Switching networks are built with active switches. A connection between two processors or between a processor and a memory module is realized through control signals on the switches. As is the case with static networks, there is a large variety of switching networks which differ in complexity and performance. In the following we will discuss the bus-based networks, crossbar networks and multistage switching networks.

Bus-based networks The bus is a common network which has also been used in von Neumann computers (e.g. a bus is used to connect several functional units with the main memory in a sequential computer). For a bus-based parallel computer, the processors are connected to a global memory by means of a bus. Figure 6.2.9 illustrates a typical bus architecture. Whenever a processor accesses the global memory, that processor generates a request over the bus. The data is then fetched from (written to) memory over the bus. Notice that only 1 processor may use the bus at a time, so an arbitration takes place before a processor acquires the right to use the bus.[4]

[4]Parallel reading from the same memory address is possible over the bus, but neither parallel writing nor parallel reading from different addresses is possible.

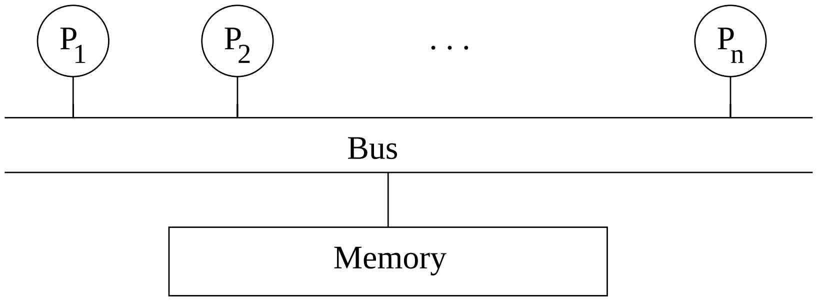


Figure 6.2.9: Illustration of a bus network.

If the number of processors increases, the network becomes the bottleneck because the bandwidth of the bus is fixed. One way to increase the bandwidth is to use multiple buses. An alternative is to reduce the data traffic through the bus by providing each processor with a *local cache memory.* Frequently accessed data are stored in local cache and therefore alleviate the bus bottleneck. For parallel computers with a number of processors more than ten, other types of networks are usually employed.

Crossbar networks A crossbar network connecting p processors to m memory modules consists of a grid of switching elements as shown in Figure 6.2.10. Each processor is connected to m switches at the crosspoint in a row. A connection between processor i and memory module j is made by switching on the switch element at the crosspoint (i,j) and leaving the other switches at row p_i in the off-state. The total number of switching elements required to implement such a network is $O(p \cdot m)$. It is reasonable to assume the number of memory modules m is at least p; otherwise, at any given time, some processors will be unable to access any memory module. Therefore, the complexity of the crossbar network grows at least with $O(p^2)$.

A crossbar switching network is capable in making $min\{p, m\}$ connections at the same time. It has also the flexibility to connect any processor to an arbitrary memory module. The disadvantage of a crossbar switching network is its cost of construction. With the current technology the crossbar network is not very scalable in terms of cost. As the number of processors becomes large, the cost of constructing the crossbar network becomes the dominating factor in the cost of the entire parallel computer.

Examples of parallel computers with a crossbar switching network are the Cray Y-MP and Fujitsu VPP 500. The crossbar network used in the VPP 500 is a remarkable one, it is a 224 $\times$ 224 crossbar network connecting 222 processors and 2 control processors.

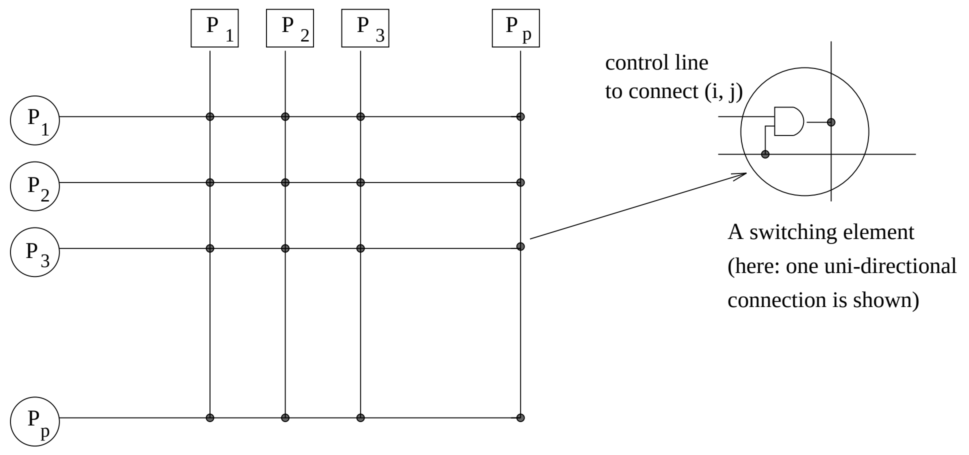


Figure 6.2.10: A crossbar switching network.

Multistage networks The crossbar network is scalable in terms of performance but unscalable in terms of cost. Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of performance. The multistage networks is a class of switching networks with characteristics in between those of a crossbar and a bus network. As the name already suggests, a multistage network consists of a number of *switching stages* (see Figure 6.2.11). Each stage is capable of implementing a number of permutations of the inputs.

In the following, we consider the commonly used *omega network* as an example to discuss the multistage networks. An omega network for connecting p processors to p memory modules consists of $log_2(p)$ stages. Each stage has $p/2$ switching elements. Each switch can perform one of the four switch functions as shown in Figure 6.2.12.[5] The p outputs of a stage are connected to the inputs of the next stage according to the so-called *perfect shuffle* pattern (Figure 6.2.13).

Figure 6.2.11 shows an omega network connecting eight processors (inputs) to eight memory modules (outputs). An interconnection is implemented through the control on the switches. Let s and d be the binary representation of the source and destination of a connection. Each binary number has $log_2(p)$ bits. Number the stages in an omega network from the outputs to the inputs starting at 1 and number the bits in s and d from the least significant bit to the most significant bit. If the i-th bit of s and d are the same, then the corresponding switch at the i-th stage is set to perform the switch function *pass-through*. Otherwise, it is set to perform the switch function *cross-over*. Figure 6.2.11 illustrates two connections in an omega network: from processor 1 (001) to memory module 5 (101) and from processor 5 (101) to memory module 7 (111). It can be seen that both connections use the link AB. So, only one of them can get through, the

[5]Some omega networks use a simpler switch element with only the switch functions pass-through and cross-over

other one is blocked. The blocking caused by contention in this situation can be dissolved in a (full) crossbar network. Although contentions can also occur in a (full) crossbar network (e.g. when two processors want to access different memory addresses in the same memory module). The cost of an omega network with $p/2 \times log_2(p)$ switching elements is $O(p \cdot log_2(p))$, which is less than the $O(p^2)$ cost of a crossbar network.

Examples of parallel computers based on the omega network are the BBN Butterfly, IBM RP-3 and NYU Ultracomputer.

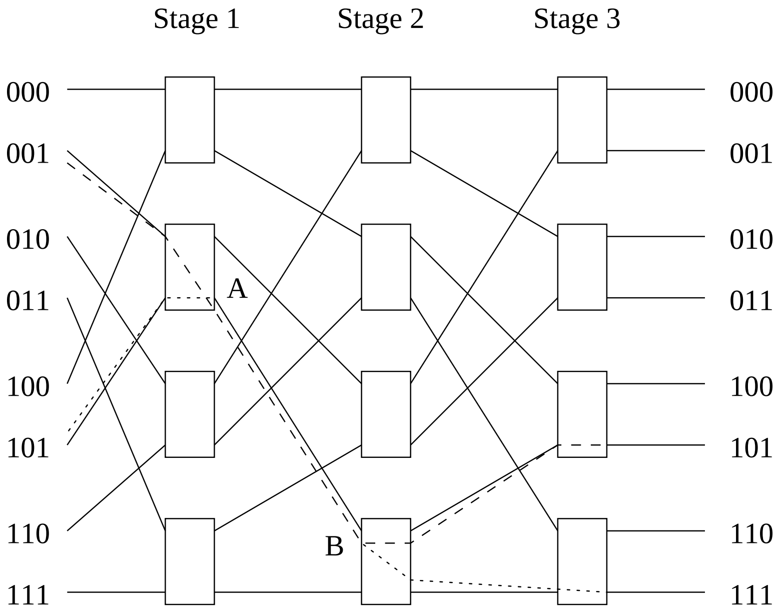


Figure 6.2.11: An omega network connecting eight inputs and eight outputs. A blocking example is shown. The dotted lines illustrate the connections from 001 to 101 and from 101 to 111.

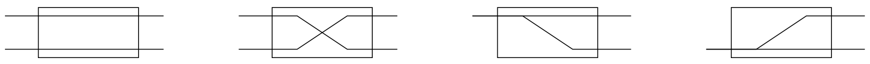


Figure 6.2.12: The four switch functions of an omega network switch element: pass-through, cross-over, upper-broadcast, lower-broadcast.

Fat tree As the last example of networks in this section we consider the so-called fat tree [93]. As mentioned in the discussion about the binary tree network, binary tree networks suffer from a communication bottleneck at higher levels of the tree. For example, when many processors in the left subtree of a node want

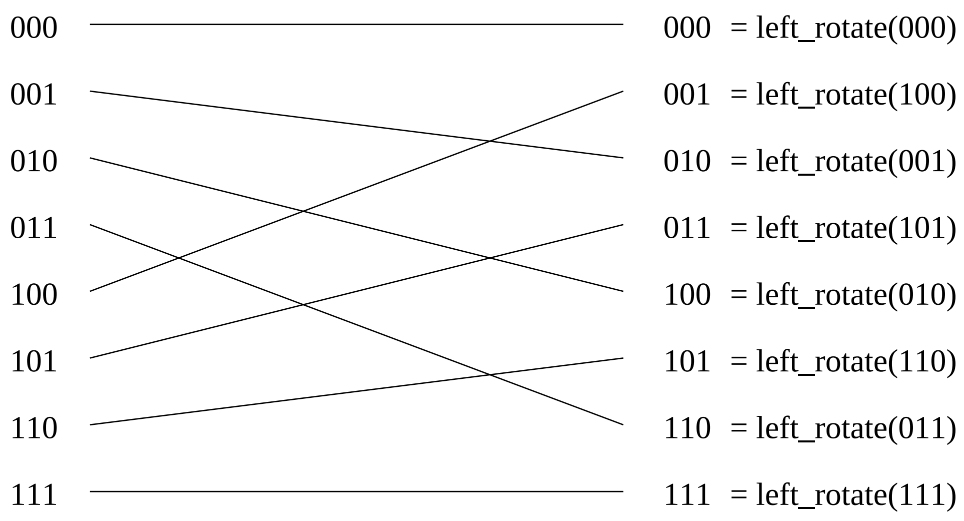


Figure 6.2.13: Illustration of a perfect shuffle of eight inputs.

to communicate with processors in the right subtree, all these communications must pass through the root node (bottleneck). The bottleneck problem can be alleviated by increasing the number of communication links between nodes that are closer to the root. Figure 6.2.14 is a sketch of a binary fat tree. The fat tree is more like the real (biological) tree in that the branches are thicker near the root. The non-leaf nodes in the fat tree are usually switches.

The idea of a fat tree has been applied in the Connection Machine CM-5 from Thinking Machines Co.

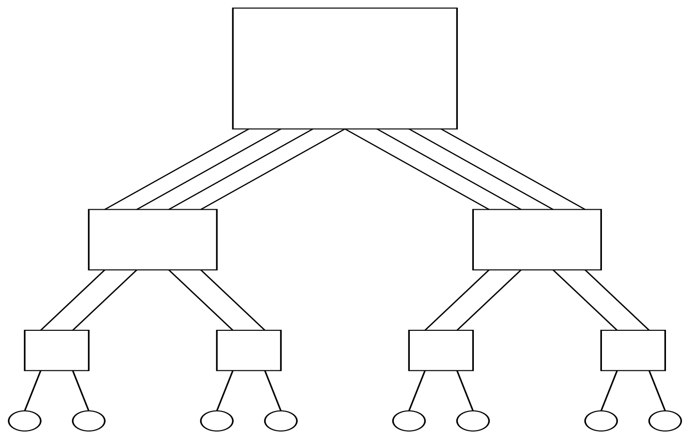


Figure 6.2.14: A fat tree network of eight processors.

Communication and routing When the source and destination are not directly connected by a physical link, the message must travel through a number of intermediate links. There are two basic approaches *store-and-forward* and *cut-through* routing (*cut-through* routing is also called *wormhole routing*). Suppose that processor A wants to send a message to processor C, and processor B lies between A and C. With *store-and-forward* routing processor B waits until the entire message has arrived and then send it to C. In *cut-through* routing, processor B immediately forwards each identifiable piece, or *packet*, of the message. Figure 6.2.15a and Figure 6.2.15b illustrate the case of sending a message of four packets $x1, \ , x2, \ , x3, \ , x4$ using the *store-and-forward* and the *cut-through* routing respectively. As we can see, the store-and-forward routing scheme needs longer time (in this case twice as long as the time it takes to send a message between two adjacent processors), while the cut-through routing only adds the time it takes to send a packet. Furthermore, store-and-forward routing uses considerably more memory on the intermediate processors. Thus, most systems use some variant of cut-through routing.

Another issue is which route or path should the message travel? In a static routing the shortest path from the source to the destination is often chosen. But the shortest path is not always the best route, for instance if (part of) the route is heavily congested by communication traffic then another route might be faster. This leads to *dynamic routing* which makes decisions based on the real-time or recent network traffic.

Time	Data on		
	processor A	processor B	processor C
0	x4 x3 x2 x1		
1	x4 x3 x2	x1	
3	x4 x3	x2 x1	
2	x4	x3 x2 x1	
4		x4 x3 x2 x1	
5		x4 x3 x2	x1
6		x4 x3	x2 x1
7		x4	x3 x2 x1
8			x4 x3 x2 x1

(a)

Time	Data on		
	processor A	processor B	processor C
0	x4 x3 x2 x1		
1	x4 x3 x2	x1	
3	x4 x3	x2	x1
2	x4	x3	x2 x1
4		x4	x3 x2 x1
5			x4 x3 x2 x1

(b)

Figure 6.2.15: Illustration of (a) Store-and-forward routing; (b) Cut-through routing.

6.2.3 Memory Organization

Besides the processor-type and interconnection network, the memory organization is another important feature of the parallel computer architecture. Memory organization can be distinguished in *shared-* or *distributed-memory* systems. In a shared-memory (or global memory) computer architecture, a number of processors are connected to a global memory system through an interconnection network (Figure 6.2.16a). Processors can directly access each memory location of the entire memory system. In a distributed-memory (or local memory) computer, each processor has its own local memory as illustrated in Figure 6.2.16b. A processor can access its own local memory, however, it cannot directly access the data in local memories of other processors. A processor can only obtain data from another processor through communication via the interconnection network.

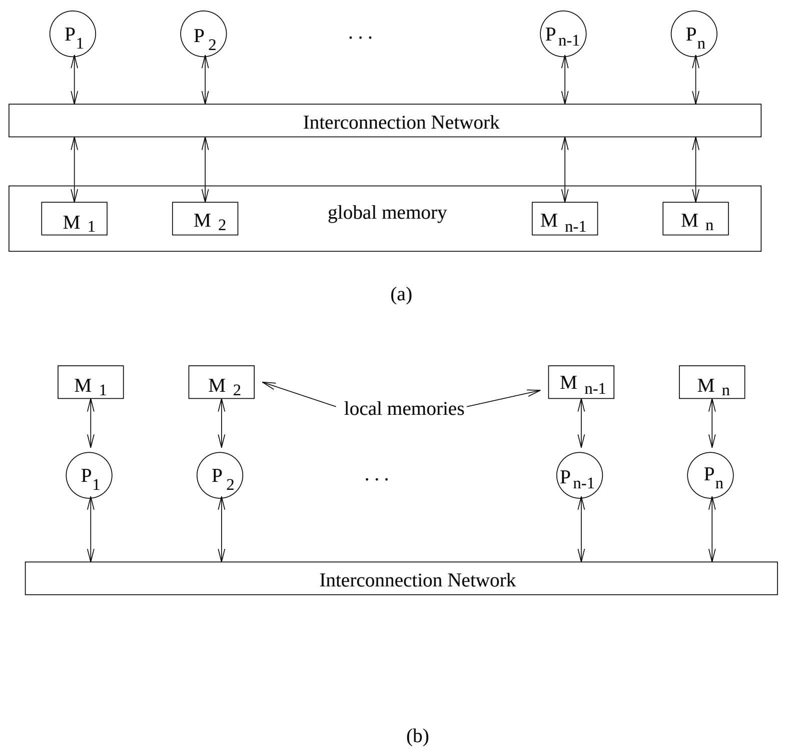


Figure 6.2.16: (a) A shared-memory parallel computer; (b) A distributed-memory parallel computer.

6.2.4 Classification of Parallel Computers

Traditional sequential computers are based on the von Neumann model with a central processing unit (CPU) and memory. However, there is not (yet) a single universal model for parallel computers. There are too many parallel computers to name all of them, and parallel computer architecture are still a fast evolving subject. So, instead of trying to give a detailed description of the hardware architectures of these machines, we will describe the important characteristic architectural features of different parallel computers. There exist many classification schemes for computer architectures, each taking a different point of view.

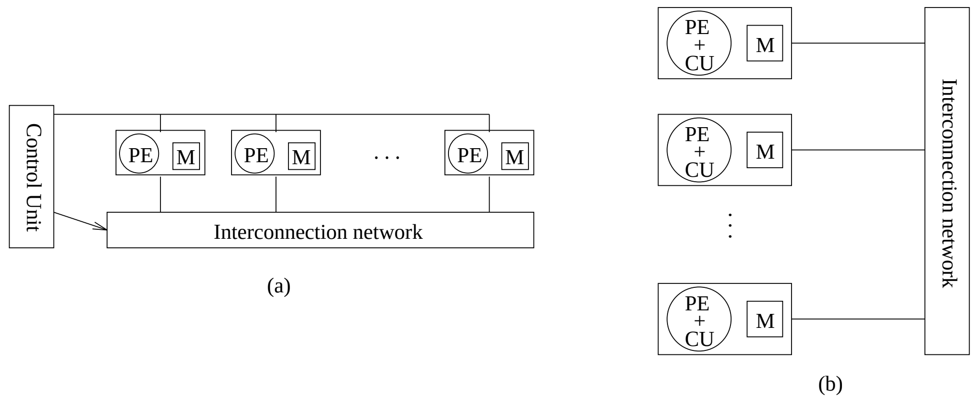


Figure 6.2.17: (a) A typical SIMD architecture and (b) A typical MIMD architecture. (CU = Control Unit)

In the following we will only briefly discuss several (simple) classification schemes. For a more comprehensive study the reader is referred to [126].

Classification based on instruction and data streams An initial and much-referenced taxonomy of computer architectures has been given by Flynn [46]. It is based on the type of control instructions and data streams the processor(s) operate upon. He divides machines into four categories:

- SISD — Single Instruction stream, Single Data stream;
- SIMD — Single Instruction stream, Multiple Data stream;
- MISD — Multiple Instruction stream, Single Data stream;
- MIMD — Multiple Instruction stream, Multiple Data stream.

An example of the SISD category is the Intel 80486 microprocessor. An array of processors with a central controller (instruction unit) is an SIMD computer. A network of processors, each being able execute instructions autonomously, is of the MIMD category. Figure 6.2.17 illustrates typical SIMD and MIMD architectures. MISD machines are seldomly built, although some people view a (special-purpose) linear array of processors executing different instruction streams on the same data stream (e.g. a systolic array performing a linear filter function) as an MISD machine [81].

Classification based on memory access time The SIMD/MIMD classification distinguishes architectures based on control and data streams. However, an important architectural feature — the data access mechanism has not been included in the SIMD/MIMD classification scheme. A classification based on the memory organization structure divides parallel computer architectures into shared- and distributed-memory architectures (Section 6.2.3). Based on memory access time characteristics the shared-memory computers are classified into two categories: *uniform memory access* (UMA) computer and *nonuniform memory access* (NUMA) computer. In a UMA computer, all processors have equal access time to all memory locations. In a NUMA computer, the access time for a processor varies with the memory locations. For example, the time to access a remote memory location is longer than the time to access a local one.

Note that almost every modern computer is a NUMA computer. For example, the processing unit or CPU of a (sequential) workstation is usually surrounded by a hierarchy of fast to slower memory subsystems: registers, caches, main-memory, disks, etc. Also, the most shared-memory parallel computers employ (local) cache or vector-registers to improve performance.

For historical reasons, the NUMA architecture is often referred to a shared-memory architecture. In principle, distributed-memory and most of shared-memory computers are of the NUMA architecture, and the local caches in a shared-memory computer can be viewed as local memories like that in the distributed-memory architecture. Therefore, we use the term NUMA also to refer to distributed-memory architectures. The major difference between the NUMA shared-memory and distributed-memory architecture is that a NUMA shared-memory architecture provides hardware support for read and write access to the memories of other processors, whereas in a message-passing distributed-memory architecture, remote access must be emulated by explicit message-passing.

Many other classification schemes based on different attributes have been proposed in literature [52, 78, 137]. Besides the instruction stream, data stream, and memory system in a parallel computer, there are classifications based on the type of the functional unit of a processor. For instance, the names scalar and vector (pipeline) computers are based on the type of the functional units of a processor. In [126], a two-level hierarchical classification scheme has been proposed. In that scheme the upper level classifies architectures based on the interconnection network(s) between the processors and/or data memories. The lower level is based on a state machine view of processors for a more precise classification. Although it is attempted to give an accurate classification, still many aspects need to be improved. However, it is beyond the scope of this text to discuss such extensions.

Classification from the programmer's point of view The previous classifications for computer architectures mainly represent the computer architect's (hardware designer) view of parallel systems. A programmer may have a different classification toward the computer he/she uses. From the programmer's point of view, the model (architecture) of computation depends as much on the functional behavior presented by software tools as on the underlying hardware architecture. In [86] a simple classification of architectures from the programmer's point of view is suggested, which consists of only three classes: shared-memory, distributed-memory (message-passing), and hybrid architectures (i.e. a mixture between shared- memory and distributed-memory). It states that this simple classification does reflect the architectural aspects that affect the programming style nowadays. The programmer's view of computer architectures necessarily mixes the actual hardware architecture with the picture of the machine presented by the software available. For example, a group of processors sharing a global memory would be classified as a message-passing system if the built-in system functions only consist of message-passing functions using the shared-memory as message buffers. On the other hand, using the programming concept of virtual shared-memory system [94], a distributed-memory (hardware) system can look like a shared-memory system to the programmer. The IBM RP3 can even be configured through programming as a shared-memory, a local memory message-passing system, or a mixture of both at run-time. These examples show how the programmer's view can be influenced by the functional behavior of software tools.

6.3 Characteristics of Parallel Algorithms

The performance of a sequential algorithm is often measured in terms of the time complexity (operation count) and space complexity (memory requirement). In the literature and in courses treating the foundations of computer science (e.g. [2]) the Random Access Machine (RAM) is used as the computer model for analyzing sequential algorithms. This RAM is a successful abstraction of the von Neumann type computers. New models are required to describe parallel computers. Unfortunately, there is a large architectural variety among existing parallel computers as discussed in Section 6.2, and there is currently not a universal parallel computer model. This complicates the analysis of parallel algorithms. A theoretical model often used for parallel computers is the Parallel Random Access Machine (PRAM). In this model the communication cost is assumed to be zero. However, the realization of the interaction structure between operations in an algorithm requires communication between processors. Therefore, in addition to the operation count and memory requirements, the communication complexity of a parallel algorithm must be studied. Furthermore, a parallel algorithm which performs well on one parallel computer may not do so on another parallel computer. For example, a parallel algorithm which is efficient for a parallel computer with about ten processors may be inefficient for a parallel computer with 1000 processors. In this section, we discuss a number of properties and characteristics of parallel algorithms that have important consequences for the performance of these algorithms.

6.3.1 An Idealized Parallel Computer

A frequently used theoretical model for parallel computers in the literature is the *parallel random access machine* (PRAM). It is an idealized parallel computer with a shared memory of unbounded size that is uniformly accessible to all processors. An arbitrary but finite number of processors can access any value in the shared memory. Each processor can perform a basic operation (like add, multiply, or logic compare) in one time unit (one clock cycle). Processors share a common clock but each has its own local program and may execute different instructions in each cycle. PRAM models are therefore synchronous shared memory MIMD computers. The PRAM model is idealized because (*i*) each processor is linked directly to all memory locations and (*ii*) the communication delay between processors is assumed to be zero (or uniform). These two assumptions are not physically realizable or at least impractical with present-day technology.

In a shared memory parallel computer, several processors can try to read from or write to the same memory location simultaneously. To model this simultaneous memory access, four different PRAM models have been developed:

a. *Exclusive-read, exclusive-write (EREW) PRAM.* Access to a memory location is exclusive: simultaneous read or write to the same memory location by more than one processor is not allowed. This is the most restricted form of the PRAM model.

b. *Concurrent-read, exclusive-write (CREW) PRAM.* Multiple read accesses to a memory location are performed simultaneously. However, multiple write accesses to a memory location are performed serially.

c. *Exclusive-read, concurrent-write (ERCW) PRAM.* Multiple write accesses to a memory location are performed simultaneously, but multiple read accesses are performed serially. This is a unrealistic form of the PRAM model.

d. *Concurrent-read, concurrent-write (CRCW) PRAM.* Multiple read and write accesses to a memory location are performed simultaneously. This is the most powerful (and unrealistic) form of the PRAM model.

To illustrate the use of the PRAM model, we analyze a simple algorithm. Consider the problem of computing the sum S of $n = 2^k$ numbers stored in an array A using $n/2$ processors $P_1, P_2, ..., P_{n/2}$. The summation can be performed by pair-wise addition of the intermediate results. The algorithm is shown in Figure 6.3.18. The statement "*B(i) := B(2i-1) +B(2i)*" in the loop do parallel means: "*read B(2i-1) and B(2i); add them together; write the result to B(i);*" in one cycle, for all $1 \leq i \leq n/2^h$. For each iteration value of h, the execution of the loop do parallel is globally synchronized. The structure of this parallel algorithm is the same as that in Figure 6.3.19b by substituting the operator '+' for the 's'.

```
for all $i$, $1 \leq i \leq n$ do parallel
    B(i) := A(i)
enddo;
for h = 1 to $\log_2(n)$ do
    for all $i$, $1 \leq i \leq n/2^h$ do parallel
        B(i) := B(2i − 1) + B(2i)
    enddo;
enddo;
S := B(1)
```

Figure 6.3.18: An algorithm for summing n numbers on a PRAM computer.

The first for-loop can be performed in 1 time unit. The inner for-loop of the nested for-loops can also be performed in 1 time unit. The last statement takes 1 time unit. So, the time required for computing the summation of n numbers is equal to $log_2(n)+2$ time units. We say that the *time complexity* of the summation is $O(log_2(n))$.

6.3.2 The DAG as Representation of Parallel Algorithms

In the complexity analysis of sequential algorithms, the algorithms are usually stated in pseudo code. Each group of operations is expressed as a control statement like a loop- or repeat-statement. All operations in an algorithm are put in a sequence. In general, such a description of algorithms provides sufficient information for the purpose of complexity analysis of sequential algorithms (e.g. [2, 3]). For parallel algorithms, additional constructs like 'for all $k \in L$ do parallel' have been added in pseudo-codes to describe groups of operations that can be performed in parallel, as illustrated in Figure 6.3.18 (see also [111, 55, 4]). Such additional constructs enable the analysis of operation count of an algorithm. However, they do not express the spatial and temporal interaction structure among the operations in a parallel algorithm. For that purpose, the *Directed Acyclic Graph* (DAG) is introduced.

A directed acyclic graph (DAG) is a directed graph that has no cycles (i.e. there is no path with the starting node equal the end node). A DAG can be used to represent a parallel algorithm (e.g., [2, 14]).

Each node in a DAG represents a *task*: a group of operations that are executed sequentially on a single processor. For example, a task can be one floating point operation, or a sequence of more complex operations. During the execution of a task, no inter-processor communication is required.

A *dependence graph* $DG(V, E)$ is a DAG, where the set of nodes $V = 1, ..., n$ represents the tasks, and the set of directed edges E represents direct data dependencies between the tasks. An edge $(i,j) \in E$ indicates that task j is directly dependent on task i. Generally speaking, this dependence can either be,

- data availability, i.e. $(i,j) \in E$ means that task j requires data (results) from task i before j can start with execution, or
- mutual exclusion, i.e. the execution of task i and j are mutually excluded from each other (e.g. tasks i and j can not be executed at the same time because of common resource contention), or
- other restrictions imposed on the execution of i and j.

In our discussion we will restrict the meaning of dependence to data availability unless specified otherwise.[6] Figure 6.3.19 shows two dependence graphs of the problem of finding the maximum of n numbers. Task i in $DG(V,E)$ is a *predecessor* of task j if $(i,j) \in E$. The *in-degree* of task i is the number of predecessors of that task. The *out-degree* of task i is the number of tasks for which i is a predecessor. For $(i,j) \in E$, the edge going from i to j is an *incoming edge* of task j and an *outgoing edge* of task i. A task which only depends on inputs is called a *start task* (*Inputs* are initial data which have in-degree zero). Tasks with out-degree zero are called *end* or *termination tasks*.

Let x_i be the result of the operation corresponding to task i in the dependence graph (x_i can be a single variable or a set of variables). Then the dependence can be viewed as a representation of functional dependencies of the form:

$$x_j = f_j(x_i \mid i \text{ is a predecessor of } j) \tag{6.3.1}$$

Here f_j is a function (or procedure) describing the operation corresponding to task j.

A dependence graph is a representation of a virtual (parallel) algorithm but it is only a partial representation of a parallel execution of an algorithm. It specifies what operations are to be performed, on what operands, and imposes certain precedence constraints on the temporal order in which these operations are to be performed (in this sense a dependence graph is sometimes also called as a precedence graph). In order to completely specify (or determine) a parallel execution of an algorithm, we have to specify which processor performs what operations (tasks) and at what time. Together with the dependence graph, this is a total specification of a parallel execution.

In the following, we discuss the parallel execution of a dependence graph in more detail. Let $p(i)$ denote the processor assigned to the execution of task i, and let $t_b(i)$ and $t_e(i)$ be positive integer variables specifying the time at which

[6]for algorithms with conditional statement like 'if B then perform S else perform T', additional constructs to the DG are required. We assume here that such a conditional statement is considered as a single task.

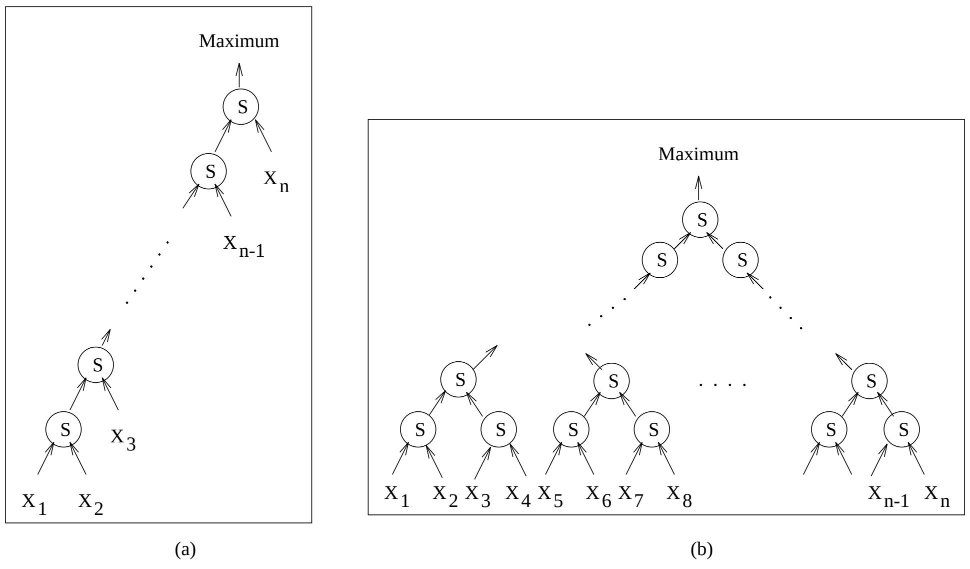


Figure 6.3.19: Dependence graphs (DAGs) corresponding to two algorithms for finding the maximum of n numbers. The operator S of each task selects the largest of two numbers.

the execution of task i is started and completed, respectively. The following two constraints hold:

- A processor can perform at most one task at a time. Thus, if $i \in V$, $j \in V$, $i \neq j$, and $p(i) = p(j)$, then $(t_b(i), t_e(i)) \cap (t_b(j), t_e(j)) = \emptyset$.
- If $(i, j) \in E$, then $t_b(j) \geq t_e(i)$. This reflects the fact that task j can only start after i has been completed.

Consider two tasks i and j in $DG(V, E)$. We say that task j is *dependent* of task i, if there is a path[7] from i to j. Two tasks are *independent* of each other if no path between them exists. A group of tasks can be executed in parallel if they are all independent of each other.

Figure 6.3.19 shows that there can be several DAGs corresponding to different algorithms for the same computational problem. Figure 6.3.19a and Figure 6.3.19b correspond to a sequential algorithm and a parallel algorithm respectively. Obviously, the DAG of Figure 6.3.19b contains more parallelism than the DAG in Figure 6.3.19a. Given some optimality measure, it is therefore of interest to find a DAG which maximizes this measure. For several interesting classes of problems, there exist methods for constructing DAGs that come within a constant factor of the optimal DAG. However, it is in general not trivial to find the

[7]A path from i to j exists in $DG(V, E)$ if $(i, j) \in E$ or there exists a nonempty set of tasks in V $\{v_1, ..., v_k\}$ with the property (i, v_1), (v_1, v_2), ..., (v_k, j) are all edges in E.

optimal DAG for a computational problem (which corresponds to finding the optimal parallel algorithm for a parallel computer model). In this sense, designing parallel algorithms corresponds to constructing DAGs that are good instead of optimal with respect to an optimality measure.

6.3.3 Analysis of Parallel Algorithms

Algorithms are usually analyzed in terms of complexity measures, that are intended to quantify the amount of computational resources required by an algorithm. Some important complexity measures for parallel algorithms are:

a. The required/effective number of processors

b. The total time required by the algorithm (time complexity)

c. The time spent on communication during the execution (communication time complexity)

In order to analyze the time complexity, the computing time of each task in a dependence graph must be known. To model this, each task i in $DG(V, E)$ is assigned a *weight*[8] W_i that corresponds to the number of time units required for the execution of that task. The amount of data to be transferred from task i to task j is denoted by $C_{i \to j}$. Notice that corresponding to each edge in E there is a $C_{i \to j}$.

The network topology of a parallel computer can be modeled as a processor graph $PG(P, L)$, where $P = \{P_1, P_2, \ ..., \ P_p\}$ is the set of processors and L is the set of links among the processors. The communication bandwidth from P_k to P_l is modeled by $L_{k \to l}$.

As discussed in the previous section, the parallel execution of a dependence graph is only completely specified if a *mapping* is known, which specifies which task is to be performed by which processor (*task assignment*) and at what time (*scheduling*). The problem of determining which task is to be executed on which processor and when a task is to be executed is called a *mapping problem*. In the literature, the *mapping problem* is also often called the *scheduling problem*. We use the term mapping problem to refer to both task assignment and scheduling (i.e. determining the execution sequence of the tasks).

Given a dependence graph $DG(V, E)$ and a processor graph $PG(P, L)$. A mapping for the dependence graph DG is *optimal* with respect to the parallel computer PG if its execution time is minimal. Finding an optimal mapping is a very hard problem: It is NP-complete even for the simple case of mapping for two processors with all tasks requiring one or two time units (without even

[8]For simplicity, we assume here that all processors are identical. A more general model should use the workload of the tasks as the weight.

considering communication!). This means that in practice heuristics must be used to find suboptimal solutions for the mapping problem.

The optimal mapping of a dependence graph $DG(V, E)$ can be very different for different processor graphs. Our purpose in doing complexity analysis is to reveal some important characteristics of a parallel algorithm.

Before discussing some complexity measures, we will define the symbols O, Ω and Θ as follows. Let A be a subset of $\mathbb{R}$ and let $f : A \mapsto \mathbb{R}^+$ and $g : A \mapsto \mathbb{R}^+$ be some functions. The notation $f(x) = O(g(x))$ means that there exists some positive constant c and some x_0 such that for every $x \in \mathbb{R}$ satisfying $x \geq x_0$, we have $f(x) \leq c \cdot g(x)$. The notation $f(x) = \Omega(g(x))$ means that there exists some positive constant d and x_1 such that for every $x \in A$ satisfying $x \geq x_1$, we have $f(x) \geq d \cdot g(x)$. The notation $f(x) = \Theta(g(x))$ means that both $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$ are true. Informally, one may read O as 'at most', Ω as 'at least', and Θ as 'equals'.

6.3.4 Time Complexity of Parallel Algorithms

The *time complexity* of a parallel algorithm is defined as the minimum execution time of all possible mappings. In the analysis of time complexity of parallel algorithms, we will ignore the communication cost. This assumption enables us to concentrate on the degree of parallelism of an algorithm represented by a DAG. Furthermore, we assume that the number of processors available is always large enough. Under these two assumptions the same time complexity as for the PRAM model results.

First, we study the parallelism in a dependence graph by means of constructing a number of sets of tasks (called *levels* such that the tasks in each level are independent from each other. Consider a dependence graph $DG(V, E)$ where all tasks have a unit execution time, and there is a single termination task in $DG(V, E)$. Notice that the restriction on the task execution time can be easily generalized to arbitrary values. The requirement of a single termination task is not a restriction. For a dependence graph with more than one termination tasks, we can simply introduce a dummy task with zero execution time as the termination task and let all the original termination tasks have this dummy task as the successor.

The leveling algorithm for constructing the levels containing independent tasks works as follows. Label the termination task as level 1. Next, label all predecessors of the termination task with the level number 2. In this manner, all predecessors of the tasks in level l are labeled with the level number $l + 1$. This process is repeated until a empty level has been obtained (which means each task has been assigned a level number). Note that a task that has already been labeled, will get a new label $l + 1$ if it (also) has a predecessor in level l. Figure 6.3.20 shows an algorithm for constructing the levels.

Algorithm Leveling

```
Label the termination task with level number 1;
current_level := { the termination task };
$l := 1$;
while current_level $\neq \emptyset$ do
    $l := l + 1$;
    next_level := $\emptyset$;
    for each task $i$ in current_level do
        Label all predecessors of task $i$ with level number $l$;
        next_level := next_level $\cup$ {all predecessors of task $i$};
    enddo;
    current_level := next_level;
enddo;
for $i := 1$ to $l$ do
    level(i) := $\emptyset$;
enddo;
for $i = 1$ to $|N|$ do
    if task $i$ has label $k$ then level(k) := level(k) $\cup$ {i};
enddo;
```

Figure 6.3.20: An algorithm for constructing the level sets of a $DG(V, E)$. N=the number of tasks in DG.

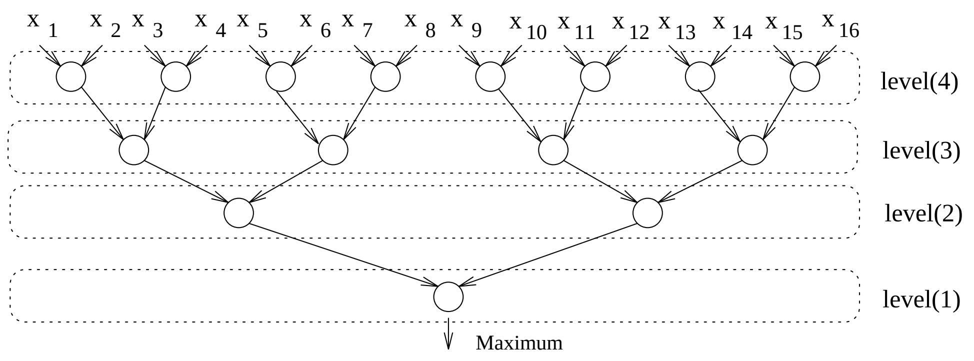


Figure 6.3.21: Levels in the dependence graph of an algorithm for finding the maximum of 16 numbers.

The levels, $level(i)$, $i = 1, \ldots, , max\_level$, determined by the Algorithm Leveling in Figure 6.3.20 have the property that all tasks in the same level are mutually independent. This can be proved as follows. Assume there are two dependent tasks a and b in $level(i)$, say task a is a predecessor of task b. Then during the labeling process from $level(i)$ to $level(i+1)$, task a will be labeled with $(i+1)$. This contradicts the preassumption that a and b are both in $level(i)$.

Because all tasks in a level are mutually independent, they can be executed in parallel in one time unit. Starting at $level(max\_level)$, the tasks in $level(i)$ can be executed after the tasks in $level(i+1)$ have been executed. Thus, the dependence graph can be executed in max_level time units. Furthermore, there exists at least a path from the termination task to any task in $level(max\_level)$, with along this path a task in each $level(i)$, $i = 2, \ldots, (max\_level - 1)$. The parallel execution of the dependence graph takes at least max_level time units. We conclude that the time complexity of the dependence graph under consideration equals max_level time units.

Figure 6.3.21 shows the levels of a dependence graph corresponding to the problem of finding the maximum of 16 numbers. Note that Algorithm Leveling can be used for mapping and scheduling (e.g. Hu's algorithm [80]). Figure 6.3.22 depicts the so-called Higher-Level-First (HLF) algorithm, which assigns the tasks in the levels obtained with the Algorithm Leveling to processors. It is a simple and fast assignment algorithm which assigns the tasks in a level to the available processors in a wrap-around manner, starting from the highest level to the lowest level. This HLF algorithm does not consider the optimization of communication (which can be easily included). If the execution time is arbitrary, then each task will be labeled with the time equal to that of the longest path from that task to the termination task (the so-called list scheduling algorithms use this information).

The time complexity of a parallel algorithm is equal to the longest path of

```
The HLF Assignment Algorithm

Let $p$ be the number of processors;
k := 1;
for  i = l to 1 do
    while level(i) $\neq \emptyset$ do
        take a task $t_j$ from level(i) and assign $t_j$ to processor $k$;
        level(i) = level(i) \ {t_j} ;
        k := (k + 1)mod p ; /* (wrap-around assignment) */
    enddo;
enddo;
```

Figure 6.3.22: A simple HLF algorithm for task assignment to processors.

$DG(V, E)$. That is, let $LP = \{q_1, q_2, \ ..., \ q_k\}$ be the set of tasks along the longest path of $DG(V, E)$, then the time complexity T_∞ of algorithm $DG(V, E)$ is

$$T_\infty = \sum_{i=1}^{k} W_{q_i} \tag{6.3.2}$$

The longest path can be found with the methods known from graph theory. Algorithm Leveling can also be modified to compute the longest path and the corresponding time complexity of a dependence graph with arbitrary task execution times. Instead of assigning a level number, the highest execution time of the successors found so far plus its own execution time will be assigned to a task in the inner-loop of the 'while-for' nested loops of Figure 6.3.20.

In the following, we will discuss a number of properties of the dependence graph. We define T_1 as the time needed for a sequential execution of $DG(V, E)$, T_p as the minimum execution time of all possible mappings that use p processors. T_∞ is the time complexity of the algorithm specified by $DG(V, E)$ when a sufficiently large number of processors is available. It holds thus $T_\infty = \min_{p \geq 1} T_p$. Obviously, T_1 is equal to the sum of the weights of all tasks in $DG(V, E)$. For an arbitrary value of p, we have $T_1 \geq T_p \geq T_\infty$. In general, the exact value of T_p is not easy to determine, however, some useful bounds for T_p can be derived. These are stated in the following propositions [14].

In the following 4 propositions, we assume that each task in the dependence graph has a unit execution time. Proposition 4.4.1 expresses the fact that if the number of processors is reduced by a certain factor, then the execution time is increased by at most that factor.

Proposition 4.4.1 If c is a positive integer and $q = c \cdot p$ then $T_p \leq c \cdot T_q$.

Proof. Consider a mapping which takes time T_q using q processors. At each time unit, at most q independent tasks are performed. They can be carried out in at most $q/p = c$ time units using p processors. Thus, by splitting each level into c levels, we have obtained a mapping using p processors which takes at most $c \cdot T_q$ time units. □

Another useful result is the following. This proposition shows that the extra time needed above the optimal T_∞ is inversely proportional to the number of processors p.

Proposition 4.4.2. $T_p < T_\infty + T_1/p$ for all positive integers $p \in \aleph$.

Proof. Consider a mapping from which the execution time is T_∞. This mapping can be viewed as consisting of T_∞ levels. A new mapping S' using only p processors can be constructed as follows. Let n_i be the number of tasks in *level*(i), $i = 1,\ 2,\ ...,\ T_\infty$. Using p processors, the tasks in *level*(i) can be performed in $\lceil n_i/p \rceil$[9] time units. The time required by mapping S' is an upper bound of T_p, thus,

$$T_p \leq \sum_{i=1}^{T_\infty} \lceil \frac{n_i}{p} \rceil < \sum_{i=1}^{T_\infty} \left(\frac{n_i}{p} + 1 \right) = \frac{T_1}{p} + T_\infty \tag{6.3.3}$$

where the fact $\sum_{i=1}^{T_\infty} n_i = T_1$ is used. □

Proposition 4.4.3.

- (a) If $p \geq T_1/T_\infty$, then $T_p < 2T_\infty$. More generally, if $p = \Omega(T_1/T_\infty)$, then $T_p = O(T_\infty)$.
- (b) If $p \leq T_1/T_\infty$, then $T_1/p \leq T_p < 2 \cdot T_1/p$. More generally, if $p = O(T_1/T_\infty)$, then $T_p = \Theta(T_1/p)$.

Proof.

- (a) If $p \geq T_1/T_\infty$ [respectively, $p = \Omega(T_1/T_\infty)$], then $T_1/p \leq T_\infty$ [respectively, $T_\infty/p = O(T_\infty)$], and the result follows from Proposition 4.4.2.
- (b) If $p \leq T_1/T_\infty$ [respectively, $p = O(T_1/T_\infty)$], then $T_\infty \leq T_1/p$ [respectively, $T_p = O(T_1/p)$]. So, from Proposition 4.4.2 it holds $T_p < 2 \cdot T_1/p$ [respectively, $T_p = O(T_1/p)$]. Furthermore, Proposition 4.4.1 yields $T_1 \leq p \cdot T_p$. Thus, $T_p \geq T_1/p = \Omega(T_1/p)$. □

Proposition 4.4.3 states an important result: although T_∞ is defined under the assumption of an unlimited number of processors, $\Omega(T_1/T_\infty)$ processors are actually sufficient to obtain an execution time within a constant factor of T_∞ (Proposition 4.4.3(a)). Proposition 4.4.3(b) shows that as long as $p = O(T_1/T_\infty)$, the

[9] $\lceil x \rceil$ is the ceiling function which returns the smallest integer $\geq x$

availability of p processors allows us to speed up the computation by a factor proportional to p.

The following proposition states a result about the fundamental limitation on the speed of a parallel algorithm [14].

Proposition 3.4. Suppose that there exists a path from every input to the termination task and that the in-degree of each task is at most 2. Then,

$$T_\infty \geq log_2(n) \tag{6.3.4}$$

where n is the number of inputs in the dependence graph.

Proof. We say that a task a in the dependence graph depends on k inputs if there exist a path from each of the k inputs to task a. We want to prove that $t_a \geq log_2(k)$ for each task a depending on k inputs and for every mapping. The claim is clearly true for $k = 1$. Assume that the claim is true for every $k \leq k_0$ and consider a task a that depends on $k_0 + 1$ inputs. Since task a can have at most two predecessors, it must have a predecessor b that depends on at least $\lceil (k_0 + 1)/2 \rceil$ inputs. Then, using the induction hypothesis, it follows,

$$t_a \geq t_b + 1 \geq log_2 \lceil (k_0 + 1)/2 \rceil + 1 \geq log_2(k_0 + 1) \tag{6.3.5}$$

The proposition is proved by this complete induction. □

6.3.5 Communication Complexity of Parallel Algorithms

The analysis of communication complexity is a more complicated problem. The actual communication between two tasks only takes place if these two tasks are executed on different processors. This means that the communication time is dependent on the mapping of tasks onto processors.

The worst-case bound on the communication is the case when every task needs to actually communicate to all its adjacent neighbors, i.e. no adjacent tasks are assigned to the same processor. This worst-case communication bound can be easily calculated for any given $DG(V, E)$ and $PG(P, L)$. Let T^w_{comm} be the worst-case communication bound, then T^w_{comm} is equal to the sum of the weights of all edges in $DG(V, E)$, i.e.,

$$T^w_{comm} = \sum_{(i,j)\in E} C_{i \to j} \tag{6.3.6}$$

An optimistic estimate for the communication complexity is obtained under the assumption that all communications among the tasks in each level are performed simultaneously. This leads to a lower bound, T^b_{comm}, for the communication complexity of $DG(V, E)$. Let l be the number of levels in $DG(V, E)$, then

$$T^{b}_{comm} = \sum_{k=1}^{l} \max_{i \in level(k+1) \text{ and } j \in level(k)} C_{i \to j} \tag{6.3.7}$$

The worst-case communication complexity says something about the interaction intensity among the tasks in an algorithm and the best-case (lower bound) says something about the minimum communication cost. However, the actual communication cost often can only be calculated when the mapping and the network topology are known. In the following we consider an example to discuss this problem.

Example 6.3.1 *Matrix-vector product on a ring We consider the problem of computing the matrix vector product $\underline{y} = A\underline{x}$ on a ring of p processors, where A is a full $n \times n$ matrix, $\underline{x}$ is a vector of length n, and $p \leq n$. Assume that n is divisible by p and let $r = n/p$. Let A be partitioned into p submatrices: $A = (A_1, A_2, \ ..., \ A_p)$, where each A_i is of size $n \times r$ (a set of r columns). Furthermore, let $\underline{x}$ be partitioned into p subvectors $\underline{x} = (\underline{x}_1, \underline{x}_2, \ ..., \ \underline{x}_p)$, where each $\underline{x}_i$ consists of r elements. The parallel calculation proceeds as follows. First compute $\underline{z}_i = A_i\underline{x}_i$ on processor P_i, for $i = 1, 2, \ ..., \ p$, and then accumulate the sum $\underline{y} = \underline{z}_1 + \underline{z}_2 + ... + \underline{z}_p$. The accumulation of the sum of the result vectors $\underline{z}_i$ can be performed, for instance, by circulating and summing the partial sums clockwise through the ring. The output $\underline{y}$ will be stored in P_p. This is illustrated in Figure 6.3.23.*

The computation performed by each processor consists of the multiplication of an $n \times r$ matrix with a vector of length r, which takes $n(2r-1)$ time units. During the accumulation, processor P_i sends its result $\sum_{j=1}^{j=i} \underline{z}_j$ to P_{i+1}, which waits for the data from P_i before performing the addition $\sum_{j=1}^{i} \underline{z}_j + \underline{z}_{i+1}$ and then passes the result to P_{i+2}. The total communication is therefore $(p-1) \cdot n$. Assume that one multiplication or one addition takes one time unit and the transfer of each number takes τ time units. Then the total execution time is $n \cdot (2n-p)/p + n \cdot (p-1) \cdot \tau$.

6.3.6 Performance Measures and Scalability

The main reason for parallel computing is to achieve high performance. In this section, we discuss some important performance measures for parallel algorithms. One of them is the *speedup*, which indicates how much faster the same problem can be computed on p processors, as compared to just a single processor. Another important property is the *scalability* of a parallel algorithm on an architecture, which is a measure of its ability to achieve performance proportional to the number of processors.

Speedup *Speedup* is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel

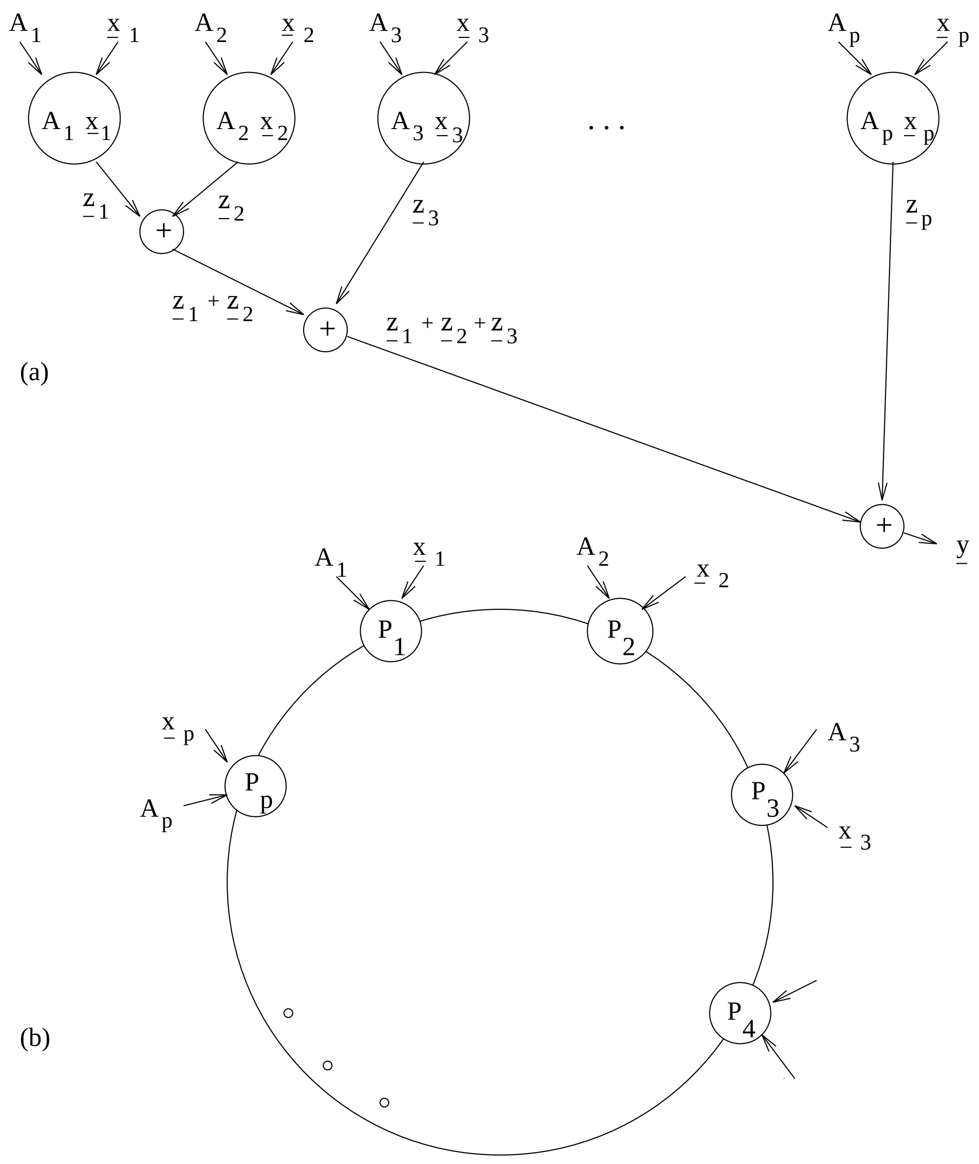


Figure 6.3.23: (a) The dependence graph of the matrix vector product algorithm; (b) Mapping on the ring of processors.

computer with p processors. Let T_1 and T_p be the time required to solve a problem with a single processor and p processors respectively. Then the speedup $S(p)$ can be denoted by

$$S(p) = \frac{T_1}{T_p} \tag{6.3.8}$$

A 'true' speedup factor should be calculated by using the known best sequential solution time as T_1 and the solution time of the parallel algorithm under consideration as T_p. The best sequential algorithm is usually not the best parallel algorithm. However, in practice it is often convenient to use the solution time by a single processor using the same parallel algorithm as T_1. In this case, $S(p)$ is sometimes called a relative speedup.

Efficiency The *efficiency* of a parallel algorithm is defined as the ratio of speedup and the number of processors, i.e.,

$$E(p) = \frac{S(p)}{p} = \frac{T_1}{p \cdot T_p} \tag{6.3.9}$$

$E(p)$ is a measure of the (average) fraction of time for which a processor is usefully employed. Due to load imbalance, communication and synchronization, the efficiency of a parallel algorithm is usually less than 100%. Consider the example of finding the maximum of n numbers (see Figure 6.3.21). If the number of processors $p = n/2$, then the computation takes $T_p = log_2(n)$ time units (ignoring the communication cost). Thus the efficiency of this parallel algorithm is $E(p) = 2(n-1)/log_2(n)$. The loss of efficiency is caused by the load imbalance.

Figure 6.3.24 shows a so-called *parallelism profile* of the algorithm in Figure 6.3.21. The number of tasks that can be executed in parallel is shown as a function of the time. After each time unit the parallelism of the algorithm drops to the half. Therefore, increasingly more and more processors are becoming idle. The efficiency is equal to the ratio between the shaded area and the rectangle $(T_\infty, 8)$.

Scalability In the previous subsection, the notion of efficiency has been introduced. A question which maybe raise is what efficiency value a good parallel algorithm must have? This question cannot be answered in a simple way. For example, an algorithm using 2 processors obtaining an efficiency figure of 0.5 is a very bad one (no speed up at all), however, an algorithm using 1000 processors obtaining the same efficiency figure can be considered as very good (a speedup of 500 times). In general, the communication overhead and load imbalance increases as the number of processors involved increases. So, for a problem with fixed size the efficiency decreases as the number of processors used to solve it increases. Therefore, an important property of a parallel algorithm is how well

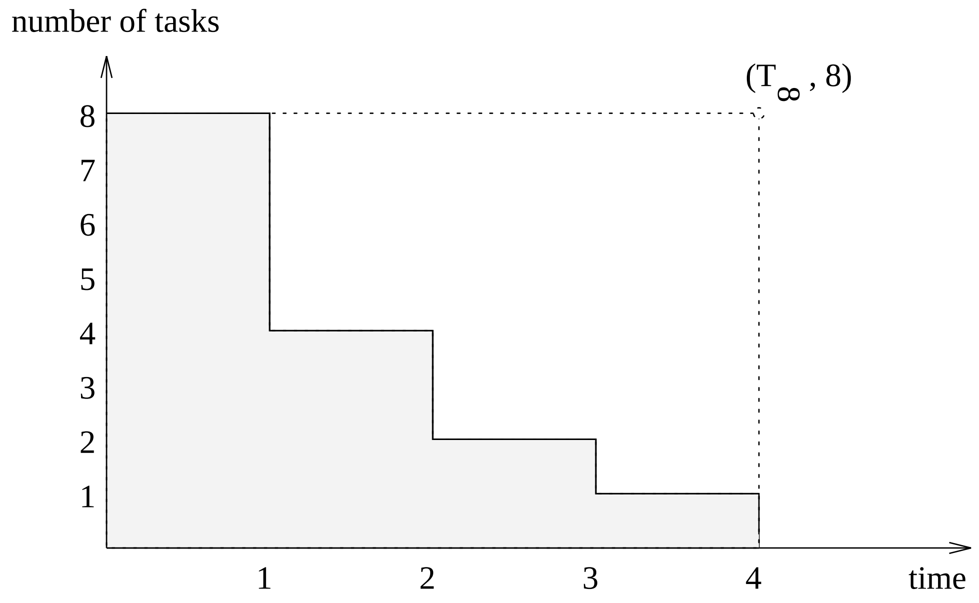


Figure 6.3.24: Illustration of the parallelism profile of finding the maximum of 16 numbers (Figure 6.3.21).

the performance of the algorithm scales up with the increase in the number of processors and in the size of a problem. This property is called the *scalability* property of a parallel algorithm.

In many cases, it is possible to keep the efficiency for a parallel algorithm constant by simultaneously increasing the number of processors and the size of the problem. Such parallel algorithms are called *scalable parallel algorithms*. The *iso-efficiency* concept, relating the workload to the machine size p needed to maintain a fixed efficiency E when implementing a parallel algorithm on a parallel computer, is introduced in [91].

The *workload* W is defined as the number of basic computation steps in the best sequential algorithm to solve the problem on a single processor, i.e. the workload is equal to the sequential time complexity of the best sequential algorithm (known so far). W is a function of the size of the input. For example, the workload W of the summation of n numbers is $(n-1)$, and that for the multiplication of an $n \times n$ matrix and a vector of size n is $n(2n-1)$.

In practice, the execution of parallel algorithms does not achieve an efficiency of one due to, among other issues, interprocessor communication, synchronization and possibly algorithmic overhead[10]. The *total overhead* or the *overhead function*

[10]The total number of basic operations in a parallel algorithm can be more than the best (known) sequential algorithm (for instance, a parallel algorithm may be chosen because it exhibits more parallelism). This extra amount of work in a parallel algorithm is the algorithmic overhead.

of an execution of a parallel algorithm can be expressed as the part of its cost (i.e. $p \cdot (execution\ time)$) that is not incurred by the fastest known sequential algorithm on a sequential computer. It is the total time collectively spent by all the processors in addition to that required by the fastest known sequential algorithm for solving the same problem on a single processor. The overhead function H is a function of $W(n)$ and p, and we write it as $H(W(n), p)$. It yields $H = p \cdot T_p - W$. For example, on a PRAM computer the overhead function of the addition of $2n$ numbers using $p = n$ processors is

$$H(W(2n), n) = n \cdot log_2(n) - (2n - 1).$$

The overhead functions for the matrix-vector multiplication of an $n \times n$ matrix and an n-vector are $H(W(n), n) = 0$ and $H(W(n), n^2) = n^2 \cdot (log_2(n) + 1) - n(2n - 1) = n^2 \cdot log_2(n) - n^2 + n$ using $p = n$ and $p = n^2$ processors respectively.

The efficiency of a parallel algorithm on a given parallel computer can thus be written as

$$E = \frac{W(n)}{W(n) + H(W(n), p)} = \frac{1}{1 + H(W(n), p)/W(n)} \tag{6.3.10}$$

In general, the overhead increases with p. With a fixed workload, the efficiency decreases as p increases. On the other hand, when the overhead H grows slower than the workload W, one can expect to maintain a constant efficiency if the workload W increases properly with the number of processors p. From Eq.(6.3.10), it can be seen that the efficiency can be maintained at a fixed value if the ratio $H(W(n), p)/W(n)$ is kept at a constant value. In order to maintain a desired value α of efficiency, the workload $W(n)$ should grow in proportion to the overhead $H(W(n), p)$. The condition of this growth rate can be derived as follows.

$$\alpha = \frac{1}{1 + H(W(n), p)/W(n)} \tag{6.3.11}$$

$$W(n) = \frac{\alpha}{1 - \alpha} H(W(n), p) \tag{6.3.12}$$

The factor $C = \alpha/(1 - \alpha)$ is a constant for a fixed efficiency α. Thus we can define the *iso-efficiency function* as follows,

$$f_{iso}(n, p) = C \cdot H(W(n), p) \tag{6.3.13}$$

The iso-efficiency function determines the case with which a parallel algorithm can maintain a constant efficiency and hence achieve speedups proportional to the number of processors. A small iso-efficiency function means that small increments in the workload are sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel algorithm is highly scalable. However, a large iso-efficiency function indicates a poorly scalable parallel algorithm. The iso-efficiency function does not exist for unscalable parallel algorithms, because in such algorithms (on a given parallel architecture) the efficiency cannot be kept at any constant value as p increases, no matter how fast the workload is increased.

In the following, two examples of scalability analysis are discussed.

Example 6.3.2 *Adding n numbers on a PRAM The parallel addition time of n numbers on a PRAM takes $n/p + log_2(p) - 1$ time steps using p processors. The workload is $W(n) = n - 1$, so the overhead function is*

$$H(W(n), p) = p \cdot (\frac{n}{p} + log_2(p) - 1) - (n - 1) = p \cdot log_2(p) - p + 1 \qquad (6.3.14)$$

The iso-efficiency function for a fixed efficiency α is

$$f_{iso}(n, p) = \frac{\alpha}{1 - \alpha} \cdot (p \cdot log_2(p) - p + 1) \qquad (6.3.15)$$

Thus, if the number of processors involved is increased, the efficiency of the parallel addition remains at an efficiency α if n is increased as $[\alpha/(1 - \alpha)] \cdot [(p \cdot log_2(p)) - p + 1)]$.

Example 6.3.3 *Fast Fourier transform on mesh and hypercube computers This example demonstrates the effect of machine architecture on the scalability of the FFT [62]. We consider the Cooley-Turkey algorithm for one-dimensional n-point fast Fourier transform. It holds that $W(n) = \Theta(n \cdot log_2(n))$.*

In [62] the overhead functions for FFT have been calculated: $H_1(n, p) = \Theta(p \cdot log_2(p) + n \cdot log_2(p))$ on a hypercube with p processors, and $H_2(n, p) = \Theta(p \cdot log_2(p) + n \cdot \sqrt{p})$ on an $\sqrt{p} \times \sqrt{p}$ mesh.

For an n-point FFT, the total workload involved is $W(n) = \Theta(n \cdot log_2(n))$. Equating the workload with overheads, we must satisfy $\Theta(n \cdot log_2(n)) = \Theta(p \cdot log_2(p))$ and $\Theta(n \cdot log_2(n)) = \Theta(n \cdot log_2(p)$, resulting in the iso-efficiency function $f_{iso,1}(p) = \Theta(p \cdot log_2(p))$ for the hypercube computer.

Similarly, on the mesh computer $W(n) = H_2(n, p)$ requires that $\Theta(n{\cdot}log_2(n)) = \Theta(p \cdot log_2(p))$ and $\Theta(n \cdot log_2(n)) = \Theta(n \cdot \sqrt{p})$. Thus the iso-efficiency function is $f_{iso,2}(p) = \Theta(\sqrt{p} \cdot k^{\sqrt{p}})$ for some constant $k \geq 2$. (Here the order is determined by the highest among the two).

The above analysis shows that the FFT on a hypercube is better scalable than on a mesh computer. The main reason is that the longest distance between two processors is $log_2(p)$ on the hypercube as compared to $2\sqrt{p} - 2$ on the mesh.

A parallel algorithm on a parallel architecture is very scalable if it has an iso-efficiency function which is a linear function of p. A parallel algorithm on a parallel architecture scales poorly if it has an exponential iso-efficiency function. Note that the memory requirement generally increases with the workload, an exponential increase of the workload may require an exponential increase in each processor's memory. For algorithms with exponential iso-efficiency functions, the increase of workload (problem size) demands more than just increasing the number of processors.

6.4 Introduction to Parallel Programming

As mentioned in section 6.1, the three major steps in building a parallel program are: *partitioning, allocation*, and *scheduling* (the last two steps are often referred to as the *mapping* problem (see Section 6.3)). The partitioning step divides a computational problem into a number of subcomputations (tasks) which can be computed in parallel. Tasks can be defined either by directly partitioning the (basic) operation or by first partitioning the result data and then defining the operations on each partition of the result data as a task. The allocation step determines the distribution of tasks and data to the processors, i.e. what data is to be initially stored on which processor, and which processor is responsible for the execution of a task. The scheduling step determines the execution sequence of the tasks. Naturally, the execution sequence of the tasks must obey the data dependency (in a DAG) between the tasks. However, sometimes two independent tasks cannot be executed in parallel due to contention, e.g., if these two tasks are assigned to the same processor then only one of them can be executed at a time.

In this section, we discuss the basics of building parallel programs. We consider the fundamental parallel constructs in parallel programming languages. There are many different parallel programming languages, they vary from existing sequential languages extended with some parallel constructs like communication and/or synchronization constructs, to new parallel programming languages like Occam and High Performance Fortran. It is not our intention to discuss all these languages, we only briefly discuss the different fundamental concepts in building parallel programs.

Although in the literature there are many different classification schemes for parallel programming languages and programming models, in this chapter we use a simple classification of parallel programming models: *shared-variable* and *message-passing*. This is a kind of analogy of the classification of parallel computers into shared- and distributed-memory architectures[11].

6.4.1 Shared-Variable Programming Model

In the previous chapters we consider a task as the basic computational unit in a parallel program. A generalization to the concept of task is the name *process*. A task is only active during a part of the total execution of a parallel program, e.g.

[11]Another often used simple classification is SIMD data parallel languages (like C* and CM Fortran) and MIMD languages. Because we are focusing on MIMD general purpose parallel processing, we omit this classification in the discussion.

once all required inputs are available, the task transforms the inputs into output data and terminates itself. A process does not necessary terminate itself after one input-to-output transformation, it may repeat doing this on other inputs a number of times or even indefinitely. A general parallel program can be viewed as a set of cooperating processes that require synchronization and communication among each other.

In the shared-variable programming model, all processes can directly access (read or write) the shared-variables. Interprocess communication (IPC) is based on the use of shared-variables (in a (virtual) shared-memory) (Fig. 6.4.25a).

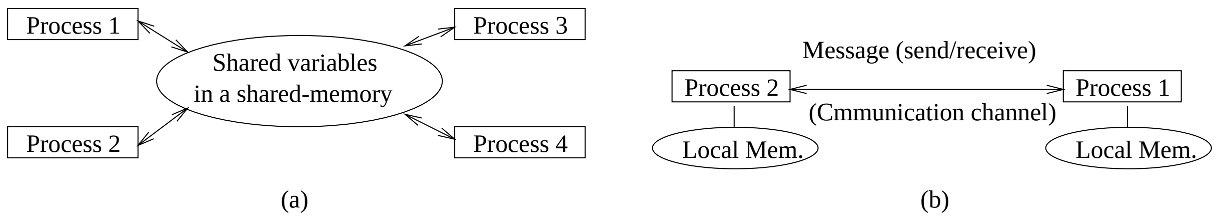


Figure 6.4.25: (a) Shared-variable model; (b) Message-passing.

6.4.2 Synchronization

The performance and correctness of a parallel program execution rely heavily on efficient synchronization among concurrent computations in multiple processors. In the following we first identify the causes requiring synchronization. Then we describe the mechanism *semaphore* for implementing the synchronization.

Process-synchronization constraints There are two kinds of synchronization constraints. A *precedence constraint* between two tasks is a requirement that one of the tasks is executed before the other. We denote such a constraint that task T_a must precede task T_b by $T_a \rightarrow T_b$. Precedence constraints generally stem from data dependencies: A constraint $T_a \rightarrow T_b$ occurs when T_b requires data produced by T_a.

The second kind of synchronization constraint is the *mutual-exclusion constraint*. A mutual-exclusion constraint between two tasks is a requirement that the execution of them do not overlap in time. It does not matter which task begins first, but whichever does begin first must finish before the other starts. If tasks T_a and T_b are mutually exclusive, the mutual exclusion constraint says that either $T_a \rightarrow T_b$ or $T_a \leftarrow T_b$. Therefore we use the notation $T_a \rightleftharpoons T_b$ to signify a mutual-exclusion constraint between T_a and T_b.

Mutual-exclusion constraints typically arise from sharing a resource, such as a data object (e.g. pointer or data base) or a link in a communication network. For example, consider the following computation: $S = S + A + B$. If we define two tasks for doing this job as follows: T_a adds matrix A to matrix S, and T_b adds matrix B to matrix S. Since both tasks use the matrix S for storing the (intermediate) result of the addition, T_a and T_b must be executed mutual-exclusively.

Semaphores Generally, modifying a variable residing in memory consists of the sequence of actions: *read, modify* and *write*. This *read-modify-write* operation is called an *atomic operation* if the sequence of actions is *indivisible*. A *semaphore* S is a data object that includes an integer counter and supports the two atomic operations $wait(S)$ and $signal(S)$. In general, a semaphore is initialized to an integer value and subsequently subjected to a sequence of *wait* and *signal* operations performed by various processes. $wait(S)$ causes a delay of the calling process performing it until the count in S is greater than 0, at which point it decrements the count by 1 and returns. Thus *wait* can delay the execution of a process for an arbitrarily time period. $signal(S)$ increments the count in S by 1 and returns. Thus *signal* does not delay the calling process and can have the side effect of allowing other processes that have been delayed by $wait(S)$ to proceed.

A *semaphore* S initialized to the value n enforces the constraint

$$signal_i(S)^S \rightarrow wait_{i+n}(S)^F \tag{6.4.1}$$

where $signal_i(S)^S$ denotes the start of the i-th *signal* operation on S and $wait_j(S)^F$ denotes the completion of the j-th *wait* operation to complete on S.

The following is an example which uses a counting semaphore to enforce precedence constraints. Consider the parallelization of the following loop:

```
for I = 1, N do
   A(I) = B(I)*C(I)
od;
here follows the code which uses the results A(I), for I=1, .., N.
..
```

On a shared-memory system, the compiler can parallelize the execution of this loop by splitting the loop into P sub-loops. Each of the P sub-loops can then be performed by a processor or process [12]. Because the part of code directly following this loop depends on the results $A(I)$, synchronization is required to force all processes (processors) waiting for completion of all sub-loops before starting the subsequent operations (such a synchronization is called a *barrier*). The parallelization of the above loop with a semaphore guarding the correctness is illustrated as in Figure 6.4.26. The loop $\{$ for $J = 1, P$ do $wait(S)$ $\}$ forces all processes (processors) to wait for the completion of all P sub-loops before proceeding further.

[12] A light weighted process, called a thread, is usually spawned to execute a sub-loop

```
reset S to 0;
for J = 1, P do parallel
   for I = 1+(J-1)*(N/P), J*(N/P) do
      A(I) = B(I)*C(I)
   od;
   signal(S);
od;
for J = 1, P do
   wait(S);
od;

here follows the code depending on A(I), for I=1,..,N.
```

Figure 6.4.26: Parallelization of a loop using a semaphore for synchronization.

A special case of the semaphore, in which the count is constrained to take on values of only 0 or 1, is called the *binary semaphore*. A binary semaphore is used in many parallel programming environments. The functions $lock(S)$ and $unlock(S)$ are often used for a binary semaphore S (another pair often used is $set(S)$ and $free(S)$). The semantic constraint imposed by a binary semaphore is as follows:

- if S is 0, $lock(S)$ delays the execution until the value of S is set to 1 and then set S to 0 and returns, otherwise if S is 1, S is set to 0 and returns immediately.
- $unlock(S)$ set the value of S to 1 unconditionally and returns.

Binary semaphores can be used easily to implement mutual-exclusion constraints. A *critical section* is a code segment which requires that only a single process at a time is executing this part of code. For example, the tasks T_a and T_b for computing $S = S + A + B$ must be executed one at a time. A possible implementation using a binary semaphore for implementing the mutual-exclusivity is illustrated as follows,

```
task Ta:                   task Tb:

  lock(S);                   lock(S);
  Ta;                        Tb;
  unlock(S);                 unlock(S);
  .                          .
  .                          .
```

The part of a program where a shared memory address space (e.g. a shared variable or a shared data structure) is accessed (modified) is called a *critical section.* For a correct parallel execution, processes must not execute a corresponding critical section in parallel. In general, two mutually exclusive critical sections may be executed in either order; however, any overlap of their execution is forbidden.

The granularity (i.e. the size) of a critical section affects the performance of a parallel program. A fine grain critical section allows more parallelism at the cost of larger synchronization overhead, whereas a coarse grain critical section reduces parallelism but has a lower synchronization overhead.

Deadlock A *deadlock* in a parallel program execution is a state where the progress of the execution of all processes (processors) is blocked indefinitely. Deadlock occurs when parallel processes are holding resources (e.g., via locks) and preventing each other from completing their execution.[13] In the following, we use the classic *five dining philosophers problem*[14] to discuss the deadlock problem.

Consider five philosophers, P_0 through P_4, seated at a round table such that philosopher $P_{((i+1) \ mod \ 5)}$ is just to the right of philosopher P_i (see Figure 6.4.27). To the left of each philosopher P_i (and therefore to the right of philosopher $P_{((i-1) \ mod \ 5)}$ is a chop-stick C_i. In the center of the table is a large bowl of spaghetti. It is assumed that one can only eat with two chop-sticks.

Each philosopher periodically becomes hungry and at that time executes the following algorithm (in a strict sequence as numbered):

a. Pick up a chop-stick to the left.

b. Pick up a chop-stick to the right.

c. Eat spaghetti.

d. Put both chop-sticks back where they were found.

[13]Another situation causing a deadlock is when a number of processes are circularly data dependent from each other, i.e. the data dependence relations among these processes form a circle in the data dependence graph. Such a situation indicates that an error is made in the design of the algorithm.

[14]This problem is introduced by Dijkstra in 1965 to discuss a synchronization problem

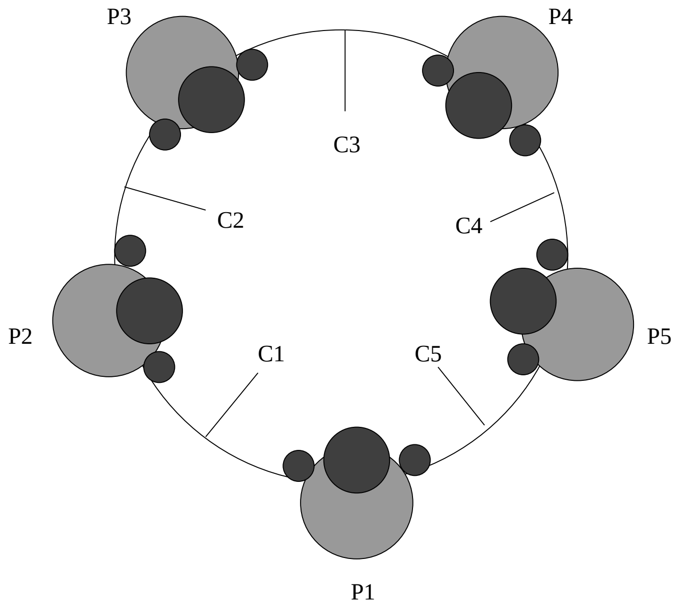


Figure 6.4.27: Illustration of the five dining philosophers problem.

In Figure 6.4.28, an implementation of the above actions of each philosopher is described.

The function *take_chopstick* waits until a specified chopstick is available and then take this chopstick. This seemingly logical implementation suffers from a potential deadlock. If hunger pangs attack all five philosophers at approximately the same time, all five may take their left chop-sticks at the same time. The result is that each philosopher P_i is stubbornly holding the chopstick C_i and waiting for chopstick $C_{((i+1)\ mod\ 5)}$ to become available. Thus, the parallel execution runs into a deadlock, and no progress can be made further.

Deadlock is a computational equivalent of the gridlock that plagues traffic-filled city streets. The key feature of both situations is a directed *cycle* in the graph of interprocess dependencies. A careless usage of semaphores can cause deadlocks. In the implementation of the five dining philosophers problem in Figure 6.4.28, the chopstick can be used by only one philosopher at a time (mutual-exclusion). The realization of this mutual-exclusivity and the waiting for a chopstick to become available in the function *take_chopstick* can be thought of as equivalent to using a semaphore lock for each chopstick.

Techniques for preventing parts of a parallel execution from "locking up" forever as a result of deadlock can be distinguished in two general categories: *deadlock avoidance* and *deadlock recovery*. Deadlock avoidance involves imposing some discipline on the way in which resources are accumulated by a process, so that deadlock becomes impossible. In contrast to deadlock avoidance, the strategy of deadlock recovery allows deadlocks to happen but provides a mechanism for

```
philosopher(i)
int i;                          /* number of the philosopher 0 - 4 */
{
   while(TRUE) {
      think();                          /* the philosopher thinks */
      take_chopstick(i);                /* take chopstick to the left */
      take_chopstick((i+1) mod 5);  /* take chopstick to the right */
      eat();                            /* delicious spaghetti */
      put_chopstick(i);                 /* put the left chopstick back */
      put_chopstick((i+1) mod 5);   /* put the right chopstick back */
   }
}
```

Figure 6.4.28: A deadlock-prone solution of the five dining philosopher problem.

detecting when a deadlock has occurred and then "backing out" (undoing) some operation, releasing resources so that other processes needing those resources can proceed. The backed-out process then resumes execution at a later time. There exists many techniques for both deadlock avoidance and deadlock recovery. It is beyond the scope of this chapter to discuss them extensively, we shall only give some glimpses of typical methods in the following.

A typical deadlock avoidance technique is to put all resources (for example, semaphores) into some numerical order and then require that processes always acquire resources in increasing numerical order. This means that a process should never attempt to acquire a resource numbered i if it is already holding another resource whose number is greater than i. The five dining philosophers problem can be made free from deadlock by letting each philosopher always take the *lower-numbered* adjacent chopstick first, and take the *higher-numbered* adjacent chopstick thereafter. The modified solution is shown in Figure 6.4.29. It can be verified that the modified solution is free from deadlocks. The directed cycles that cause deadlock are now impossible because if any philosopher P is waiting for some resource (chopstick) C, any resources P is already holding must have lower numbers than C. There can be no philosopher $\tilde{P}$ who holds C but is waiting (directly or indirectly) for some resource held by P, since all of those resources have lower numbers than C. Therefore the situation in which any philosophers are indirectly blocking themselves from making further progress cannot occur.

The above deadlock avoidance scheme of acquiring resources in a certain predefined order is limited to situations in which the resources that will be required by a transaction are always known before any of the resources are acquired. If some of the needed resources cannot be known until other resources have already been acquired, there is a chance that a transaction will need to acquire a resource with a lower number than the resources it has already acquired, thus resulting in a potential deadlock. In such a case, one may take a very rigid action by putting

```
philosopher(i)
int i;                  /* number of the philosopher 0 - 4 */
{
   while(TRUE) {
      think();
      if (i < 4 ) {       /* take lower-numbered chopstick first */
         take_chopstick(i);
         take_chopstick((i+1) mod 5);
      }
      else {
         take_chopstick((i+1) mod 5);
         take_chopstick(i);
      }
      eat();
      put_chopstick(i);
      put_chopstick((i+1) mod 5);
   }
}
```

Figure 6.4.29: A correct solution of the five dining philosopher problem.

all the resources under the guard of a single semaphore. In this way, a process, that has locked the semaphore, is guaranteed of successful claim for all the resources it needs and consequently of progress. The semaphore will be unlocked by the process at completion. This rigid scheme is free from deadlock, but it has a serious problem: only one of the processes requiring the shared resources guarded by that semaphore can be active, the others have to wait. So, the execution is strictly sequential. This can become the bottleneck in the parallel performance.

An alternative is to use the deadlock recovery technique. Deadlock recovery requires detecting a deadlock state and backing up some process involved in a deadlock and releasing the resources so that other processes can acquire those resources and proceed. Deadlock recovery can be used when a simple deadlock avoidance scheme cannot be applied (such as the situations in which some needed resources cannot be known until other resources have already been acquired). Backing up a process P during a deadlock recovery requires that any visible side effects performed by P after a certain point must be undone (so that P can be transparently restarted from that point). If many side effects must be undone, it can be very time consuming and very difficult. So, a process P should avoid introducing side effects (other than the inevitable side effects inherent in a *wait* operation) until all resources have been acquired. In doing so, a backup can be performed simply by backing up the program counter to the first *wait* operation and *signaling* all semaphores that have been acquired.

6.4.3 Message-Passing Programming Model

Communication In a distributed memory computing environment, processes running on different processors have to communicate to each other through e.g. message-passing. This also holds for processes running on the same processor if they do not share the same memory address space. The synchronization problem and the techniques discussed in Section 6.4.2 are very similar to that of an operating system. In this section, we discuss the type of communication mechanisms and show some semantical analogy between the semaphore lock and the blocking type of communication. We will also illustrate how deadlock can be avoided by scheduling the communication among the processes (processors).

Synchronous and asynchronous communication The basic operations needed for message-passing are *send* and *receive*. In the following discussion, the communication between one sending process and one receiving process is assumed, so broadcasting (one process send to many at the same time) is not considered. Also we view each process as a virtual processor which operates autonomously, therefore we use the term process for both process and processor when appropriate.

The message-passing communication protocols can be divided into three categories:

a. **Synchronous communication**: Blocking send and receive, i.e. the process that first arrives at a send (or receive) statement will have to wait until the receiving (or sending) process also arrives at the corresponding statement;

b. **Asynchronous communication**: Non-blocking send and non-blocking receive, i.e. the communicating processes are not blocked by an execution of a send or receive statement;

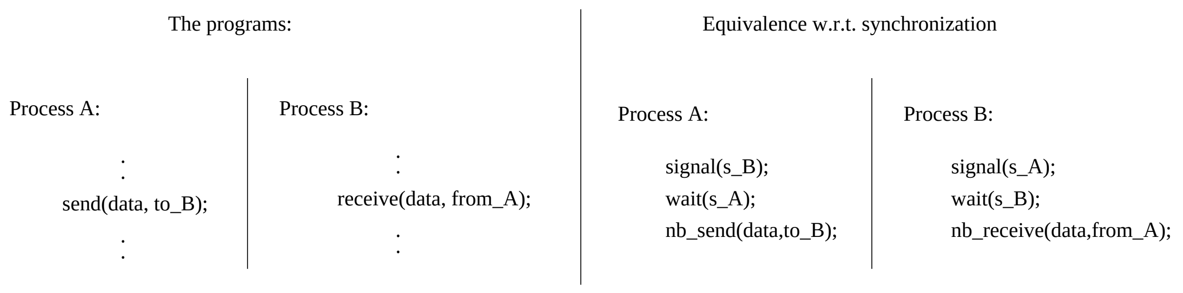
c. **Hybrid**: Non-blocking send and blocking receive.[15]

Synchronous communication forces two processes to wait for each other at the "communication point". This has the effect of synchronization similar to the use of a semaphore. This is illustrated in Figure 6.4.30.

Non-blocking communication primitives do not provide automatic synchronization. Synchronization in that case has to be done explicitly by the programmer. However the hybrid communication protocol with its blocking receive and non-blocking send can easily be used to implement data dependencies (precedence constraints) in execution of task graphs.

Deadlock and communication scheduling

[15]In the literature, this protocol is often referred as synchronous.



```
Process A:
    :
send(data, to_B);
    :
```

```
Process B:
    :
receive(data, from_A);
    :
```

```
Process A:
signal(s_B);
wait(s_A);
nb_send(data,to_B);
```

```
Process B:
signal(s_A);
wait(s_B);
nb_receive(data,from_A);
```

Figure 6.4.30: Synchronization effect of blocking send and blocking receive. Both semaphores s_A and s_B are initialized to 0. nb_send() and nb_receive() are non-blocking.

Implicit parallel programming languages In the previous two sections we discussed some basic parallel constructs for building parallel programs. So far, we have assumed that these constructs are used by the programmer *explicitly* telling the parallel program how and when to synchronize or communicate. New parallel programming languages have been designed to ease the implementation of parallel programs which need less programmer's control on the parallel execution. For instance, in a fully implicit parallel programming language the programmer can write a program in a similar way like a sequential Fortran program without the explicit usage of parallel constructs. An advanced parallelizing compiler is used to analyze the dependency, partition the computation into several parts and generating an executable for a target parallel computer. Between fully explicit and fully implicit languages, there are different languages supporting different degree of implicit parallelism.

HPF HPF (High Performance Fortran) is a parallel programming language with the emphasis on the decomposition of parallel data to be processed. There are no explicit communication constructs. The parallelization of operations on a set (array) of data is realized in three steps (Fig. 6.4.31).

- Decomposition — Decomposition declares a logic structure. The array structure, name, number of dimensions, and size of the dimensions is specified without reserving memory. For example, the construct

  ```
  DECOMPOSITION A(10,10)
  ```

  declares a two dimensional structure A with 10 elements in each dimension.

- Align — Align defines a logic mapping. Arrays defined earlier can be assigned a decomposition with the ALIGN construct without a concrete association with a physical processor being involved. In the following, the array T is directly mapped onto the structure S:

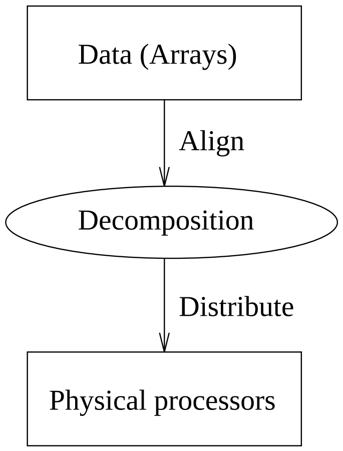


Figure 6.4.31: Data alignment and distribution.

```
REAL T(100,100)
DECOMPOSITION S(100,100)
ALIGN T(I,J) WITH S(I,J)
```

The mapping does not have to be one-to-one. For instance, entire columns or rows can likewise be collapsed to a single element of the structure. In the following example, each row in array T is mapped onto one element of structure A.

```
REAL A(100)
ALIGN T(I,J) WITH A(I)
```

- Distribute — Distribute defines a physical mapping. The virtual elements of a structure (with the array elements mapped to them) are distributed among physical processors available. One of the following three distribution procedures can be selected for each of the dimensions of the structure: 1. BLOCK - division of the decomposition into sequential blocks of equal size for each processor; 2. CYCLIC - Every decomposition element is mapped circularly to the next processor; 3. BLOCK_CYCLIC(x) - like CYCLIC, but with blocks of size x.

  The following example shows that the mapping of the structure S, with 100 rows each having 100 columns, onto 50 physical processors (a '*' means that all of the decomposition elements in this dimension are mapped to the same processor):

```
DISTRIBUTE S(BLOCK,*)       /* Every processor gets two
                               neighboring rows */

DISTRIBUTE S(CYCLIC,*)      /* Every processor gets two rows
                               that have 50 rows in between them */

DISTRIBUTE S(BLOCK,CYCLIC)  /* The rows are mapped in block mode
                            while the columns are mapped circularly */
```

There is a FORALL construct for the parallel execution of instructions. With this command, all loop iteration is executed in parallel on different processors, according to the mapping for the array specified earlier. A direct mapping of the loop passes onto physical processors can be achieved using ON clause, which, however, is not used in the following example.

```
FORALL I = 1, 100
   FORALL J = 1, 100
      T(I,J) = 3*T(I,J) - 10
   ENDDO
ENDDO
```

7 Parallel Iterative Methods

In this chapter we discuss the parallelization of the iterative methods that we have seen in the previous chapters.

Generally speaking, there are two different types of approaches for the parallel treatment of PDEs. Both of them are based on a geometric decomposition of the domain, on which the PDE is to be solved. The first approach is to use a fast sequential solver for the given problem and to parallelize this solver as efficiently as possible. In practice, this typically means that a fast iterative method is used for the solution of the global problem and *grid partitioning* is used for parallelization. Typically, the parallel versions of the iterative methods *are equivalent to their sequential counterparts*. In Section 7.2 the concept of *grid partitioning* will be introduced. We will introduce terms such as *speed-up*, *parallel efficiency*, *scalability* and discuss the *boundary-volume effect*.

The second approach is to start with *a decomposition of the given problem into a number of subproblems* on subdomains with or without an overlap. This is the basic principle of some of the "*domain decomposition*" methods (DD).

In Section 7.4.3, we will discuss parallel multigrid and, in particular, the extension of *grid partitioning* to multigrid. Whereas (point) relaxation methods are *local* methods (i.e. all operations at a grid point involve only values at points in a local neighborhood), multigrid has non-local features. On coarse grids the relation of computation and communication becomes worse than on fine grids. Thus, the definition of the coarsest grid and the corresponding solution process may have to be reconsidered. We will also discuss the boundary-volume effect and scalability in the context of multigrid.

We will start with some general remarks on parallel architectures and discuss two basic rules for an efficient parallelization. Here, the question of communication comes into play. For our presentation, we will use the terminology of parallel systems with *distributed memory* at some places in this chapter. For example, data is assumed to be *communicated* between processors by *sending* and *receiving messages*. On other parallel architectures, for example shared memory computers, other data transfer (and/or synchronization) techniques are common.

7.1 Parallel Algorithms for Dense Matrix Problems

Designing algorithms usually requires the consideration of a number of tradeoffs. For instance, traditionally in designing sequential algorithms we are accustomed to the tradeoff of time versus space: an algorithm that is constrained to use less memory space may be slower than one not so constrained (e.g., discarding (some) intermediate results and compute them again when needed versus storing intermediate results for later usage). A number of additional tradeoffs arise in designing parallel algorithms. For example, the tradeoff between maximal load balance and minimal communication overhead. Another important factor influencing the choice among algorithms is the *degree of parallelism* in an algorithm. Some best sequential algorithms (in terms of operation count) posses little parallelism or are even inherently sequential (such as the solution of the tri-diagonal system of equations by means of Gaussian elimination. Parallel algorithms with a high degree of parallelism and a lower communication requirement are an important subject of research.

The general principle of making an existing sequential algorithm parallel is to consider the operations to be performed and partition them into a number of tasks. These tasks should be defined such that they exhibit a high degree of parallelism (i.e., there are few dependence among the tasks). Because communication and synchronization introduce overhead, the tasks should also have a large *data locality* (i.e., a high ratio between the amount of computation within the task and the amount of the input and output data of the task).

Tasks can be defined directly considering the partitioning of the (basic) operations, or by first partitioning the result data and then defining the operations on each partition of (result) data as a task.

As mentioned in Section 6.3.3, the parallel execution of an algorithm is only completely specified when the mapping of the tasks is also determined. The mapping specifies which task is to be performed on which processor and the relative execution sequence of the tasks on a processor[16]. So, in analyzing the performance of an implementation of a parallel algorithm on a parallel computer, we must also consider the mapping problem.

Solving systems of linear equations is at the core of many problems in engineering and scientific computing. A system of n linear equations can be represented in matrix form by $A \cdot u = b$, where A is the $n \times n$ matrix of coefficients, b is an $n \times 1$ vector, and u is the $n \times 1$ solution vector. In this section, we assume that the matrix A is a dense matrix.

Iterative methods are techniques to solve the systems of equations of the form $A \cdot u = b$ through generating a sequence of approximations to the solution vector u. Let u^k denote the k-th approximation to u. Many iterative methods can be described as a Picard (or fixed-point) iteration: $u^k = Q \cdot u^{k-1} + s$, where $Q = (I - M^{-1} \cdot A)$ is a matrix and M is some non-singular matrix (also called preconditioner), and the vector $s = M^{-1} \cdot b$. The matrix Q depends on the type of iterative method. From the above consideration, we observe that a matrix-vector multiplication is performed in each iteration. The number of iterations required to solve a system of equations with a desired precision is usually data dependent; hence the number of iterations is not known prior to executing the algorithm. In this section, we limit our analysis of performance and scalability to a single iteration of an iterative method, *although the eventual choice of a parallel iterative method also depends on the convergence rate of that method* (which together with

[16]The data flow execution scheme does not require the determination of the relative execution sequence prior to the execution.

the single iteration parallel efficiency determines the overall performance of an iterative method on a parallel architecture).

In the following, we first consider the Jacobi and the Gauss-Seidel (GS) methods for problems with dense matrices.

7.1.1 The Jacobi Method for Dense Matrix Problems

The Jacobi iterative method is one of the simplest iterative techniques. Let D be a diagonal matrix whose principal diagonal elements are equal to the corresponding principal diagonal elements of matrix A. The k-th approximation with the Jacobi method can be expressed as

$$u^k \;=\; D^{-1} \cdot (b - (A - D) \cdot u^{k-1}) \;=\; (I - D^{-1} \cdot A) \cdot u^{k-1} + D^{-1} \cdot b \qquad (7.1.1)$$

At the beginning, an initial guess u^0 for the solution vector u is used to start the iteration process. We assume that all the diagonal elements of A are nonzero (or are made nonzero by permuting the rows and columns of A). The expression for the evaluation of the i-th element of u^k is

$$u^k[i] \;=\; \left(b[i] - \sum_{j \neq i} A[i,j] \cdot u^{k-1}[j] \right) / A[i,i] \qquad (7.1.2)$$

The vector $r^k = (b - A \cdot u^k)$ is the *residual* after k iterations. The iteration process has converged when the magnitude of the residual r^k, $\|r^k\|_2$, becomes very small. The iteration process terminates when $\|r^k\|_2$ falls below a predetermined threshold, which is usually $\epsilon\|r^0\|_2$ with a very small ϵ $(0 < \epsilon \ll 1)$.

Equation (7.1.1) can be expressed in terms of the residual r^k as follows

$$u^k \;=\; D^{-1} \cdot r^{k-1} + u^{k-1} \qquad (7.1.3)$$

The Jacobi iteration process is described in Figure 7.1.1. There are three main computations in a Jacobi iteration:

a. the inner product for computing the norm of r^k,

b. computing the new approximation u^k,

c. the matrix-vector multiplication for computing the new residual r^k.

In the following we consider the parallel implementation of the Jacobi iteration process. The second main computation, $u^k := D^{-1} \cdot r^{k-1} + u^{k-1}$ corresponds to the computations $u^k[i] := r^{k-1}[i]/A[i,i] + u^{k-1}[i]$, for $i = 1, 2, \ ..., \ n$, is the most simple one among the three main computations. When $A[i,i]$, $r^{k-1}[i]$ and $u^{k-1}[i]$ are assigned to the same processor for $i = 1, 2, \ ..., \ n$, then these computations can

```
1    begin
2        $k := 0$;
3        select an initial solution vector $u^0$;
4        $r^0 := b - A \cdot u^0$;
5        while $(\|r^k\| > \epsilon\|r^0\|)$ do
6            $k := k + 1$;
7            $u^k := D^{-1} \cdot r^{k-1} + u^{k-1}$;
8            $r^k := b - A \cdot u^k$;
9        od;
10       $u := u^k$;
11   end
```

Figure 7.1.1: An algorithm for the Jacobi iterative method.

be performed without requiring any communication. However, the computation of the matrix-vector product and the norm of the residual are more complicated.

Consider the operation in line 8, once the vector u^k has been computed, all the components $r^k[i]$ of the residual vector r^k can be computed in parallel independent from each other. However, to compute the value of $r^k[i]$, all elements $u^k[j]$ for $1 \leq j \leq n$ are required ($r^k[i] := b[i]$-(i-th row of A) $\cdot\, u^k$); some communication is necessary unless the computation of all elements of u^k and r^k is performed sequentially on a single processor.

In parallelizing an existing algorithm, we usually start with analyzing the part of the computation requiring the most communications (i.e., the part with the most intensive data dependence relations). Therefore, we first consider the matrix-vector multiplication. Because $b[i]$ is a constant, so we can simply assign $b[i]$ to the processor which is responsible for computing $r^k[i]$.

The two major techniques to partition the computation into tasks and then assign them onto processors are: 1. Partition/cluster directly the basic operations involved in a computation and define each of the partitions/clusters as a task to be performed by one processor; 2. First partition/cluster the results data (results of the operations), then define the operations necessary for computing the results (data) of each partition as a task to be performed by one processor. In fact, every parallel computation requires the partition of the operations and the distribution of the data. The partition and the distribution are inter-dependent, so usually after one of the two has been determined, the other must be matched. For regular computations, such as most dense matrix and vector computations, the second technique (i.e., the owner-compute rule) is often very convenient and consequently has been often applied. In the following, we consider several data-partitioning schemes for the matrix-vector multiplication $s = A \cdot u$.

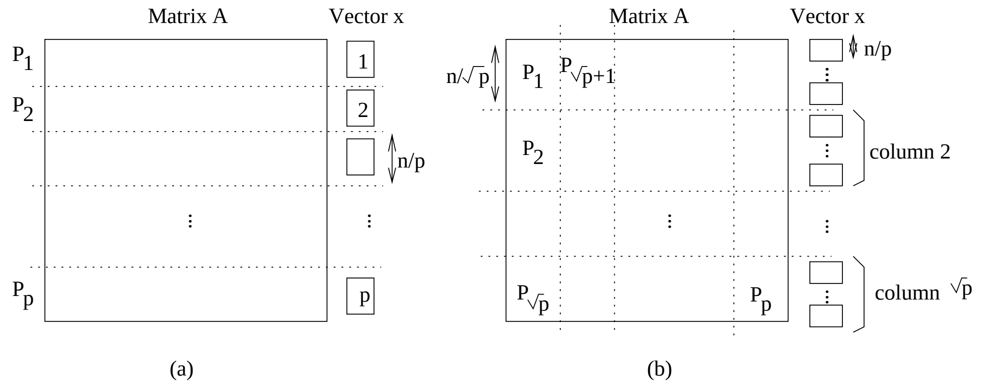


Figure 7.1.2: (a) Striped partitioning; (b) Block partitioning.

Striped partitioning In the *striped partitioning* of a matrix, the matrix is divided into groups of complete rows or columns, and each processor is assigned one such group. Usually, all groups have an equal number of rows or columns. Each group comprises contiguous rows or columns. Figure 7.1.2a shows a row-wise striped partitioning of a $n \times n$ matrix on p processors. Note that using the striped partitioning, the maximum number of partitions for an $n \times n$ matrix is n, therefore a maximum of n processors can be used. In the following, we consider the parallel computer is a ring of p processors.

Another partitioning concerns the vector u. To compute an element $s[i]$ of the result vector s, an inner product of each row of A and the vector u must be calculated. As is already mentioned, the update of u in each iteration must be distributed among the processors (otherwise the Jacobi iteration process is sequential). We partition u into p groups each consisting of n/p elements according to the partitioning of the matrix (Figure 7.1.2a).

Now tasks in the matrix-vector multiplication can be defined according to the partitions of the matrix A. Each task computes n/p inner products of the rows in a partition of A and the vector u. Consider now the case that matrix-vector multiplication is to be performed on a ring of p processors. Each processor is assigned with the computation of one task. Since the n elements of u are distributed across the p processors and each task requires the complete vector, each partition of u (n/p elements) must be first broadcast to all the processors. This can be done by letting all processors simultaneously transfer the data clockwise around the ring. Suppose that each processor can perform a send and a receive operation simultaneously and that the data transfer time of n/p elements is $t_{comm}(n/p)$, then the total communication time for broadcasting is equal to

$$T_{str,1}(n,p) \ = \ (p-1) \cdot t_{comm}(\frac{n}{p}) \qquad (7.1.4)$$

After the step of broadcasting, all the processors can perform a computing task (i.e., calculating n/p inner products) simultaneously. The time required by this step is

$$T_{str,2}(n,p) \ = \ (\frac{n}{p}) \cdot (2n-1) \tag{7.1.5}$$

Therefore, with the striped partitioning the total parallel execution time of a matrix-vector multiplication on a ring is

$$T_{str,mult} = T_{str,1} + T_{str,2} \ = \ (\frac{n}{p}) \cdot (2n-1) + (p-1) \cdot t_{comm}(\frac{n}{p}) \tag{7.1.6}$$

After the matrix-vector multiplication, a subtraction of two vectors followed to finish the operations in line 8 (Figure 7.1.1). This subtraction can be done in parallel and takes only n/p time units. After line 8, the norm of the residual $\|\underline{r}^k\|$ must be evaluated to determine the stopping condition. Each processor starts to calculate an inner product of a vector of (n/p) elements of $\underline{r}^k$ which takes $(2n/p - 1)$ time units. Next, the p products must be summed up. Since all processors needs to know the information about whether the stopping criterion has been met, so we choose to let each processor sum the p products locally. It proceeds as follows: each processor broadcast its local inner product to all others by first sending its own local inner product followed by circulating the received local inner product $(p-1)$ times clockwise. The broadcast takes $(p-1) \cdot t_{comm}(1)$ time units. After this broadcast, each processor has all p local inner products and the global inner product can be summed in $(p-1)$ time units. Thus the computation of (global) inner product takes a time of

$$T_{str,ip} \ = \ \frac{2n}{p} - 1 + (p-1) \cdot t_{comm}(1) + p - 1 \tag{7.1.7}$$

Since the operations in line 7 can be performed in parallel with a time of $2n/p$ (assuming the corresponding diagonal elements of D are allocated locally in each processor), the total parallel execution time per Jacobi-iteration on a ring of p processors is

$$\begin{aligned} T_{str,iter} &= T_{str,mult} + T_{str,ip} + \frac{2n}{p} \\ &= \frac{n(2n+3)}{p} + (p-1) \cdot (t_{comm}(\frac{n}{p}) + t_{comm}(1)) + (p-2) \end{aligned} \tag{7.1.8}$$

Typically, the communication time of m numbers takes a time of $t_s + m \cdot \tau$, where t_s is the start time for a communication and τ is the transmission time for one number.

A column-wise striped partitioning for the matrix A can also be defined. The parallel algorithm for this partitioning is considered in 6.3.5.

Block partitioning In the *block partitioning* of a matrix, the $n \times n$ elements of the matrix are partitioned into blocks (squares) each of $n/\sqrt{p} \times n/\sqrt{p}$ elements (Figure 7.1.2b). Consider a 2D $\sqrt{p} \times \sqrt{p}$ torus. Each processor is assigned with the computation associated with a partition of the matrix. The first $\sqrt{p} \cdot (n/p) = n/\sqrt{p}$ elements of u are involved in the multiplication with the first column of blocks in the partitioned matrix. So we assign n/p elements of $\underline{x}$ to each of the processors in the first column. A total of $n/\sqrt{p}$ elements of u is assigned to the i-th column of processors.

Before the multiplication takes place, each partition of n/p elements of u must be transferred to all processors in the same column. In a torus, each column of processors is connected as a ring. So this data transfer can be done in the same way as for striped partitioning. The communication time for the broadcasting in each column of $\sqrt{p}$ processors is

$$T_{blc,1}(n,p) \;=\; (\sqrt{p}-1) \cdot t_{comm}(\frac{n}{p}) \tag{7.1.9}$$

Next, the multiplication of an $n/\sqrt{p} \times n/\sqrt{p}$ block-matrix with a vector of $n/\sqrt{p}$ elements can be performed locally in all processors simultaneously. This takes a time of

$$T_{blc,2}(n,p) \;=\; \left(\frac{2n}{\sqrt{p}} - 1\right) \cdot \frac{n}{\sqrt{p}} = \frac{2n^2}{p} - \frac{n}{\sqrt{p}} \tag{7.1.10}$$

Now, each processor has $n/\sqrt{p}$ partial sums that must be accumulated along each row to obtain the final result of the matrix-vector multiplication. Since in the Jacobi iteration the result $r^k = b - A \cdot u^{k-1}$ must be distributed again for the computation in line 7 of Figure 7.1.1, we let all processors on a row accumulate the $n/\sqrt{p}$ elements of u. This takes another communication time of $(\sqrt{p}-1) \cdot t_{comm}(n/\sqrt{p})$ and a time of $(\sqrt{p}-1) \cdot (n/\sqrt{p})$ for the additions. So the time required for the accumulation is

$$T_{blc,3}(n,p) \;=\; (\sqrt{p}-1) \cdot \left(\frac{n}{\sqrt{p}} + t_{comm}(\frac{n}{\sqrt{p}})\right) \tag{7.1.11}$$

Thus, the total execution time for the matrix-vector multiplication is

$$\begin{aligned} T_{blc,mult}(n,p) \quad &= \quad T_{p,1} + T_{p,2} + T_{p,3} \;=\; \frac{2n^2}{p} - \frac{2n}{\sqrt{p}} + n \\ &\qquad + (\sqrt{p}-1) \cdot \left(t_{comm}(\frac{n}{p}) + t_{comm}(\frac{n}{\sqrt{p}})\right) \end{aligned} \tag{7.1.12}$$

In the following, we use the above parallel matrix-vector multiplication algorithm to implement of the Jacobi method on a 2D $\sqrt{p} \times \sqrt{p}$ torus.

Let the vectors u^{k-1} and r^{k-1} be distributed according to the block partitioning scheme as described in the previous consideration of the matrix-vector multiplication. Each processor holds n/p elements of u^{k-1} and r^{k-1}. The first group of $(n/\sqrt{p})$ elements of a vector is distributed over the first column of $\sqrt{p}$ processors, the second group is distributed over the second column of processors, and so on. With this allocation scheme, the computation in line 7 of Figure 7.1.1 can be performed in $2n/p$ time units and no communication is required. In line 8, after matrix-vector multiplication, another n/p time units are required to complete the subtraction. From Eq.(7.1.12), it follows that the computation in lines 7 and 8 can be performed in a time of

$$T_{blc,7+8}(n,p) \;=\; \frac{2n^2+3n}{p} - \frac{2n}{\sqrt{p}} + n + (\sqrt{p}-1)\cdot\left(t_{comm}(\frac{n}{p}) + t_{comm}(\frac{n}{\sqrt{p}})\right) \quad (7.1.13)$$

Now each column of processors contains the whole vector r^k, therefore the inner product in line 5 can be computed by the processors in one of the $\sqrt{p}$ columns. However, the information about the test on the condition $\|r\| < \epsilon\|r_0\|$ is required by all the processors, so we let all $\sqrt{p}$ columns of processors compute the inner product simultaneously (they are otherwise idle anyway). The computation of the inner product proceeds as follows. First, each processor computes a local inner product of a vector with $(n/\sqrt{p})$ elements. This takes $(2n/\sqrt{p}-1)$ time units. Then, the local inner products are accumulated to the final result. By circulating the local inner products along the column-ring, each processor can accumulate the final result (i.e., global inner product). The accumulation takes $(\sqrt{p}-1)$ additions and $(\sqrt{p}-1)$ communications each of 1 number. Thus, the accumulation process takes a time of $(\sqrt{p}-1)\cdot(1+t_{comm}(1))$. After this, all the processors can evaluate the condition simultaneously.

The total parallel execution time of one iteration in the Jacobi method is

$$\begin{aligned} T_{blc,iter}(n,p) \quad &= \quad \frac{2n^2+3n}{p} + n + \sqrt{p} - 2 \\ &\quad +(\sqrt{p}-1)\cdot(t_{comm}(\frac{n}{\sqrt{p}}) + t_{comm}(\frac{n}{p}) + t_{comm}(1)) \quad (7.1.14) \end{aligned}$$

Scalability analysis In the following, we analyze the scalability of the parallel implementation of the Jacobi iteration on a 2D torus. The sequential run time (work load) of one Jacobi iteration is

$$W(n) = 2n^2 + 4n \approx 2n^2 \quad (7.1.15)$$

Applying the relation $H(p,n) = p \cdot T_p(n) - W(n)$, we get the following expression for the overhead function for this parallel Jacobi iteration:

$$H(p,n) = p \cdot n \cdot (2+\tau) + p^{3/2} \cdot (1+\tau) \qquad (7.1.16)$$

where for $t_{comm}()$ the communication time function $t_{comm}(m) = m \cdot \tau$ is substituted and the terms of low order are ignored.

Therefore the iso-efficiency function for the parallel Jacobi iteration on a 2D torus is (Eq.(6.3.10)),

$$f_{iso}(W,p) = p \cdot \sqrt{W}(\sqrt{2} + \tau/\sqrt{2}) + p^{3/2}(1+\tau) \qquad (7.1.17)$$

The iso-efficiency function Eq.(7.1.17) is a function of W and p. Recall that the condition for constant efficiency is that the ratio H/W remains fixed (see Section 6.3.6). As p and W increase, the efficiency is nondecreasing as long as none of the terms of H grows faster than W. So, we may balance W against each term of H (or Eq.(7.1.17)) and compute the respective iso-efficiency functions for individual terms. The component of H that requires the highest growth rate in problem size with respect to p determines the overall asymptotic iso-efficiency function of the parallel implementation.

The first term requires that $W = K_1 \cdot p \cdot \sqrt{W}$, where K_1 is some constant. This results in $W = K_1^2 \cdot p^2$. The second term requires that $W = K_2 \cdot p^{3/2}$ (K_2 is some constant). Therefore, the asymptotic iso-efficiency function of the parallel Jacobi iteration is $f_{iso} = \Theta(p^2)$. Thus in order to maintain a fixed parallel efficiency the workload must grow with p^2 as the number of processors p increases. This means the size of the matrix system n must grow linearly with p.

7.1.2 The Gauss-Seidel Method for Dense Matrix Problems

In the k-th iteration of the Jacobi method, the approximation u^k is computed using u^{k-1} of the $(k-1)$-th iteration. The Gauss-Seidel algorithm differs from the Jacobi algorithm in that the most recent values of each element in u^k are used to compute new approximations. Thus, assume that the values of $u^k[i]$ are computed in increasing order of i, and the values of $u^k[1], u^k[2], \ldots, u^k[i-1]$ have already been computed. Then the value of $u^k[i]$ is computed using these newly computed values of the k-th iteration. Although both the Jacobi method and the Gauss-Seidel method are not always guaranteed to converge, the Gauss-Seidel method often achieves faster convergence than the Jacobi method.

Let the matrix A be subdivided into a lower triangular matrix L, a diagonal matrix D and a upper triangular matrix U, i.e. $A = L + D + U$. The relation between the approximation u^k and u^{k-1} in a Gauss-Seidel iteration can be expressed as

$$(L+D)u^k = -Uu^{k-1} + b \qquad (7.1.18)$$

Eq.(7.1.18) can be rewritten in the form of a Picard process as,

$$u^k = (L+D)^{-1} \cdot (-Uu^{k-1} + b) \tag{7.1.19}$$

However, the computation of the inverse of $(L+D)^{-1}$ is very expensive, in general its computational cost is of the same order as solving the system of equations directly. So, in the Gauss-Seidel iteration $u^k[i]$ is usually computed by substituting the newly computed values $u^k[1], u^k[2], \ldots, u^k[i-1]$. This iteration process can be written as

$$u^k = -D^{-1}(Lu^k + Uu^{k-1}) + D^{-1}b \tag{7.1.20}$$

The expression for the evaluation of the i-th element of u^k is

$$u^k[i] = \left(b[i] - \sum_{j=1}^{i-1} A[i,j] \cdot u^k[j] - \sum_{j=i+1}^{n} A[i,j] \cdot u^{k-1}[j] \right) / A[i,i] \tag{7.1.21}$$

From Eq.(7.1.21) it can be observed that the computation of $u^k[1], u^k[2], \ldots, u^k[n]$ in the k-th iteration is completely sequential because $u^k[i]$ cannot be computed until $u^k[i-1]$ has been computed for $1 \le i \le n$.

Now, the parallelism must be sought in the evaluations of $u^k[1], u^k[2], \ldots, u^k[n]$ in the k-th iteration. After the value of $u^k[j]$ is computed, this value is needed for the evaluation of $u^k[i]$ for all i with $j+1 \le i \le n$. Furthermore, $u^k[j]$ is also used for the evaluation of the values $u^{k+1}[i]$ for $1 \le i \le j$ in the next iteration. So, a proper arrangement of the evaluation sequence will result in a maximum of n parallel pairs of multiplications and additions (Eq.(7.1.21)). In the following, we consider parallel Gauss-Seidel algorithms on different computer architectures and show that different algorithms are preferred on different architectures. Throughout the discussion, we assume that computation and communication on the same processor cannot overlap.

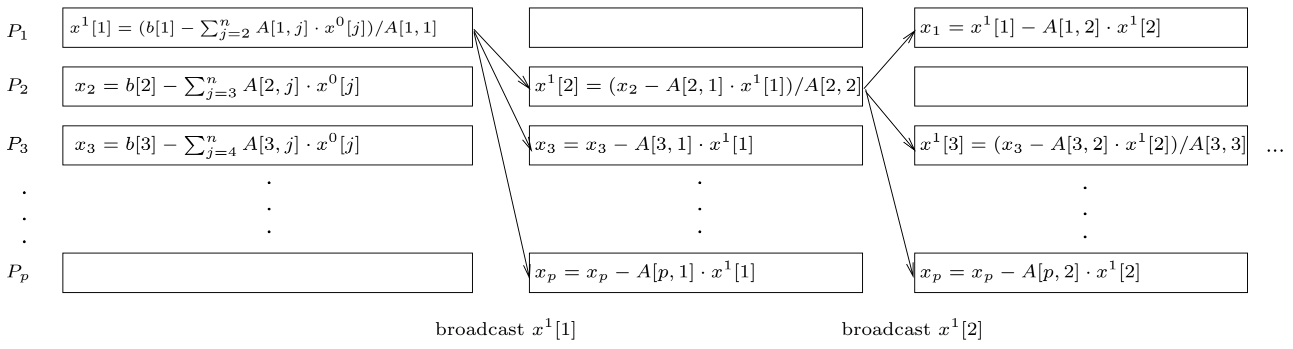
A parallel computer with a broadcast interconnect Assume p processors are interconnected to a broadcast network (i.e., any one of the processors can send the same message to all others at once). Assume further that a floating point operation (+, -, * and /) takes one time unit and broadcasting one number to all processors takes τ time units. Figure 7.1.3 shows a simple parallel algorithm which can be naturally implemented on this architecture. For simplicity, we assume $n = p$ and processor P_i is assigned the computation of $u^k[i]$. At the beginning, only $u^1[1]$ can be calculated from the initial values $u^0[k]$, for $k = 1, \ldots, n$. We let all the initial values be broadcast to all processors and let processor P_i calculate $\hat{u}_i = (b[i] - \sum_{j=i+1}^{n} A[i,j] \cdot u^0[j])$ (we use $\hat{u}_i$ to denote the incompletely updated result of $u^k[i]$). After this initial phase, processor P_1 sends the newly computed $u^1[1]$ to all other processors. Upon receiving $u^1[1]$, processors P_i, $i = 2, \ldots, p$, computes $\hat{u}_i = \hat{u}_i - A[i,1] \cdot u^1[1]$. Processor P_2 can now compute $u^1[1]$

through $u^1[2] = u_2/A[2,2]$. Next P_2 sends the value of $u^1[2]$ to all other processors, and so on. The parallel computation continues alternately with a computation phase and a communication (broadcast) phase. This type of parallel computation is often called a *lock step* mode execution, meaning that after completion of a computation phase, a communication phase follows and a new computation phase can only begin after the communication phase is finished. Each computation phase takes 3 time units (the longest time counts which equals one subtraction, one multiplication and one division). The broadcasting of one number takes τ time units. There are n computation and communication phases per iteration, the time required by each iteration for this simple Gauss-Seidel algorithm is

$$T_{cast,iter}(n = p, p) \;=\; p \cdot (3 + \tau), \qquad (p > 1) \tag{7.1.22}$$

The sequential Gauss-Seidel requires $2 \cdot p^2 - p$ time units ($p^2 - p$ subtractions, $p^2 - p$ multiplications and p divisions) per iteration, therefore the efficiency of this simple parallel algorithm is

$$E(p) \;=\; \frac{2 \cdot p - 1}{p \cdot (3 + \tau)} \tag{7.1.23}$$



P_1
P_2
P_3
P_p
$x^1[1] = (b[1] - \sum_{j=2}^{n} A[1,j] \cdot x^0[j])/A[1,1]$
$x_2 = b[2] - \sum_{j=3}^{n} A[2,j] \cdot x^0[j]$
$x_3 = b[3] - \sum_{j=4}^{n} A[3,j] \cdot x^0[j]$
$x^1[2] = (x_2 - A[2,1] \cdot x^1[1])/A[2,2]$
$x_3 = x_3 - A[3,1] \cdot x^1[1]$
$x_p = x_p - A[p,1] \cdot x^1[1]$
$x_1 = x^1[1] - A[1,2] \cdot x^1[2]$
$x^1[3] = (x_3 - A[3,2] \cdot x^1[2])/A[3,3]$
$x_p = x_p - A[p,2] \cdot x^1[2]$
...
broadcast $x^1[1]$
broadcast $x^1[2]$

Figure 7.1.3: A parallel Gauss-Seidel algorithm on a broadcast interconnect.

On a parallel architecture with a fast one-to-all broadcast interconnect (i.e., a small τ) a good efficiency is achieved with this algorithm. When the communication cost τ is high, the efficiency can be very poor. A higher efficiency can be achieved if more than one, say m, values $u^k[i]$ are assigned to each processor. The work to be performed by a processor during a computation phase becomes $2 \cdot m + 1$, subsequently the efficiency becomes $E = (2 \cdot p \cdot m - 1)/(p \cdot (2 \cdot m + 1 + \tau))$. For a large value of m, a high efficiency is achieved.

On a ring network Suppose that the Gauss-Seidel iteration is to be implemented on a ring of p processors. First we consider that the algorithm in Figure 7.1.3 is implemented on the ring network and the broadcast is emulated in this point-to-point network. The time of broadcast one number is equal to $(p-1) \cdot t_{comm}(1)$ time units (through circulating the message clockwise). The time required by one communication phase increases with p. The efficiency of the parallel implementation on the ring network is

$$E = \frac{2 \cdot p \cdot m - 1}{p \cdot (2 \cdot m + 1 + (p-1) \cdot t_{comm}(1))} \tag{7.1.24}$$

$$\approx \frac{2 \cdot m}{2 \cdot m + p \cdot t_{comm}(1)} \tag{7.1.25}$$

The efficiency is poor for large p and especially when $p \gg m$.

In the following we consider a better alternative of the Gauss-Seidel iteration on a ring of processors. Again we assume that the number of equations is $n = m \cdot p$ and let each processor evaluate m values $u^k[i]$. However, instead of assigning m consecutive $u^k[i]$ and the corresponding m consecutive rows of the matrix A to a processor, a different partitioning and assignment is applied. Processor P_q is assigned $u^k[i]$ with $i = \{q, p+q, ..., (m-1) \cdot p + q\}$ and the corresponding rows of A. At the beginning (the first iteration $k = 1$), all processors start to evaluate the values as much as possible (i.e., all operations corresponding to the initial vector $u^0[i]$, $i = 1, ..., n$ are computed). Some processors have less work than the others. P_1 has the most work. All processors synchronize and wait for P_1. P_1 computes $u^1[1]$ and sends it to P_2. Upon receiving $u^1[1]$ P_2 can now finish the evaluation of $u^1[2]$ followed by sending $u^1[1]$ and $u^1[2]$ to P_3. In general, P_q receives $u^1[1]$, $u^1[2]$, ..., $u^1[q-1]$ from P_{q-1} and computes $u^1[q]$ and sending $u^1[1]$, $u^1[2]$, ..., $u^1[q]$ to P_{q+1}, and so on.

A Gauss-Seidel iteration comprises m cycles, each processor completes the computation of 1 variable $u^k[i]$ in a cycle; $n = m \cdot p$ variables are computed in each iteration. Figure 7.1.4 illustrates the procedure of this parallel Gauss-Seidel iteration. For processor P_q, the j-th cycle $(0 \leq j \leq (m-1))$ in iteration k starts with computing $u^k[(j+1)p+q]$, followed by sending the message of $(p-1)$ numbers $u^k[(j+1)p+1]$, ..., $u^k[(j+1)p+q], u^k[j \cdot p+q+2]$, ..., $u^k[(j+1) \cdot p]$ to P_{q+1}. A cycle ends when a message $u^k[(j+2)p+1]$, ..., $u^k[(j+2)p+q-1], u^k[(j+2)p+q+1]$, ..., $u^k[(j+2) \cdot p]$ from P_{q-1} arrives at P_q. Between the two messages, P_q uses the received $(p-1)$ numbers and $u^k[j \cdot p+q]$ to update the $(m-1)$ values u_q, u_{p+q}, ..., $u_{(m-1)p+q}$. The update computation in a cycle requires $t_{upd}(m,p) = 2(m-1) \cdot p$ time units. The time elapsed after P_q sent the message and the arrival of the next message is the time a message travels from P_{q+1} around the ring and finally arrives at P_q, plus the time to calculate $(p-1)$ values $u^k[i]$, which is equal to $t_{com+cal}(m,p) = (p-1)(2p-1)+(p-2) \cdot t_{comm}(p-1))$.

A Gauss-Seidel iteration consists of m cycles, therefore on each processor the time required for one iteration is

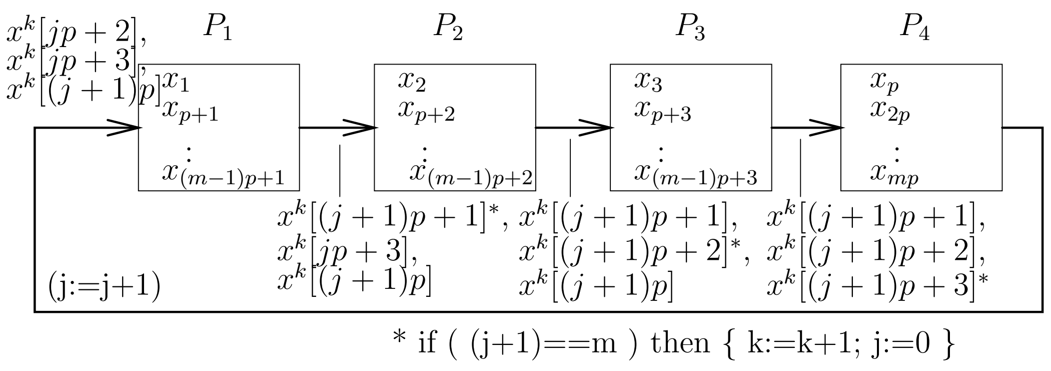

P_1
P_2
P_3
P_4
$x^k[jp+2]$, $x^k[jp+3]$, $x^k[(j+1)p]$
x_1, x_{p+1}, $\vdots$, $x_{(m-1)p+1}$
x_2, x_{p+2}, $\vdots$, $x_{(m-1)p+2}$
x_3, x_{p+3}, $\vdots$, $x_{(m-1)p+3}$
x_p, x_{2p}, $\vdots$, x_{mp}
$x^k[(j+1)p+1]^*$, $x^k[jp+3]$, $x^k[(j+1)p]$
$x^k[(j+1)p+1]$, $x^k[(j+1)p+2]^*$, $x^k[(j+1)p]$
$x^k[(j+1)p+1]$, $x^k[(j+1)p+2]$, $x^k[(j+1)p+3]^*$
(j:=j+1)
* if ((j+1)==m) then { k:=k+1; j:=0 }

Figure 7.1.4: Illustration of the alternative parallel Gauss-Seidel algorithm. P=4.

$$
\begin{aligned}
T_{alt}(m,p) &= m \cdot max\{t_{upd}(m,p) + 2p - 1 + 2t_{comm}(p-1),\ t_{com+cal}(m,p)\} \\
&= m \cdot max\{2m \cdot p - 1 + 2t_{comm}(p-1), \\
&\quad 2p^2 - 3p + 1 + (p-2)t_{comm}(p-1))\} \qquad (7.1.26)
\end{aligned}
$$

Notice that P_1 first starts a cycle, followed by P_2, and so on. A cycle does not start and end on all processors at the same time, but each cycle has the same duration on all processors. Since the computation and the communication must synchronize, the maximum of the two times is taken as the time required by one cycle.

The efficiency of this alternative parallel Gauss-Seidel algorithm is ($n = m \cdot p$)

$$
\begin{aligned}
E &= \frac{2 \cdot n^2 - n}{m \cdot max\{2m \cdot p - 1 + 2t_{comm}(p-1), 2p^2 - 3p + 1 + (p-2)t_{comm}(p-1))\}} \frac{1}{p} \\
= & \quad \frac{2m \cdot p - 1}{max\{2m \cdot p - 1 + 2t_{comm}(p-1), 2p^2 - 3p + 1 + (p-2)t_{comm}(p-1))\}} \qquad (7.1.27)
\end{aligned}
$$

Figure 7.1.5 shows the efficiency of the two parallel Gauss-Seidel algorithms on a ring as a function of m and p. It can be observed that for $m \geq p$ the alternative parallel Gauss-Seidel algorithm (Eq.7.1.27) performs much better than the first one (Eq.7.1.24). $t_{comm}(p-1) = (p-1)$ is substituted in plotting the figures.

Discussion So far, we have considered parallel algorithms dealing with *dense matrices*. In practice, the majority of large linear systems are *sparse*, i.e., only a few elements in the coefficient matrix A are non-zero. In such a case, the computation of $u^k[i]$ needs only those values of $u[j]$ when $A[i,j] \neq 0$. Because there are only few non-zeros in a row of A, the above two parallel Gauss-Seidel algorithms are not efficient due to the fact that in each cycle there is only few

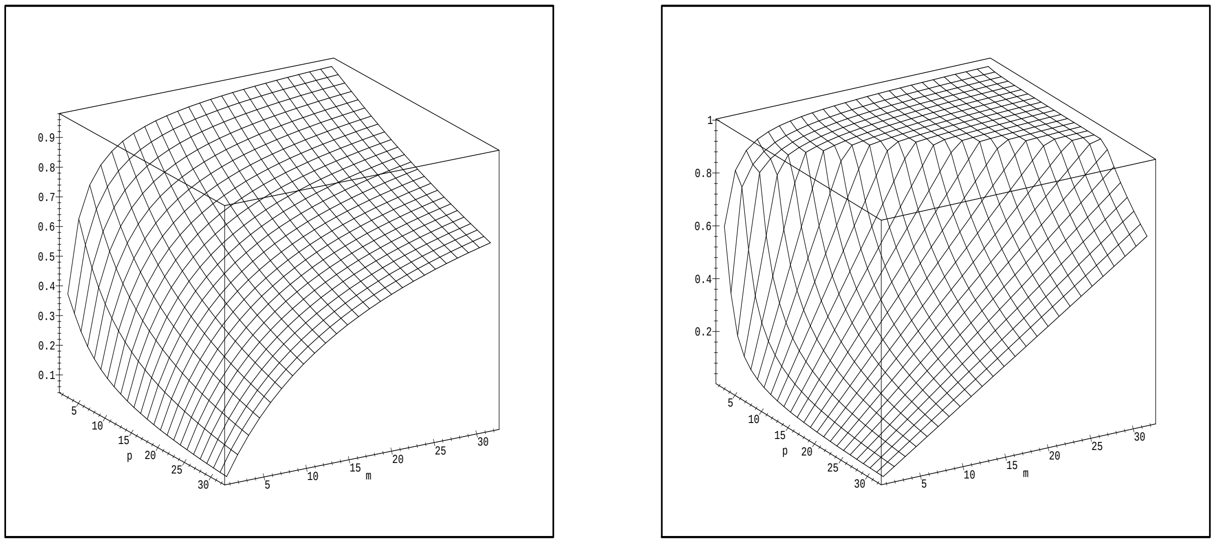


Figure 7.1.5: Comparison of the efficiency between two Gauss-Seidel algorithms. (a) emulating a broadcasting on a ring; (b) the alternative method.

computations required on each processor. Hence for sparse matrices, different parallelization approaches must be applied. Since the computation $u^k[i]$ depends only on the non-zero elements $A[i, j]$, different $u^k[i]$ (of the same iteration k) can be computed in parallel. We refer to section 7.2.2 for the discussion of this case.

7.2 Grid Partitioning

Here, we are interested in the practical questions, such as the objective of minimizing the corresponding parallelization overhead.

Such practical questions are often related to the architecture of the parallel computer at hand and to the programming model employed.

7.2.1 Parallel Systems and Basic Rules for Parallelization

Parallel computer architectures have been developing and changing very rapidly. The answer to the question of how to design an ideal parallel iterative solver clearly depends on the concrete parallel architecture to be employed: whether we use a parallel (multi-processor) system with shared, distributed or some hierarchical memory, whether it consists of vector, cache or scalar processors and which type of interconnection network is used (a static or a dynamic one, the type of topology etc.). Principally, we also regard workstation and PC clusters as parallel systems here.

Figure 7.2.6 shows an example of such a parallel system, a cluster consisting of 16 workstations. We assume that the workstations are connected by some network in order to exchange data and to communicate with each other.

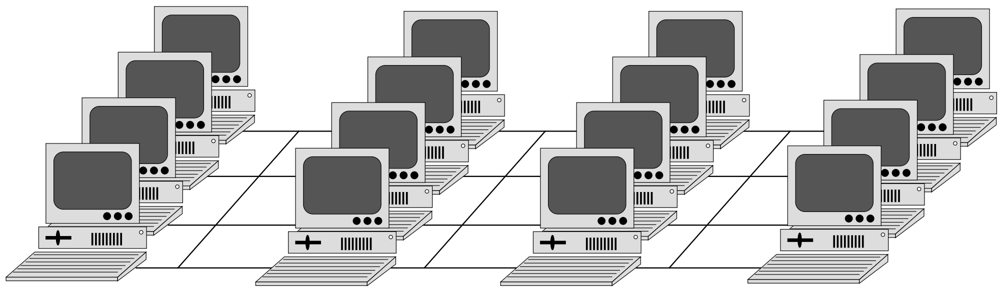


Figure 7.2.6: 16 workstations as an example of 16 connected processors

For such (and other) architectures, the memory/cache organization may have an essential impact on the overall (parallel) efficiency of an algorithm (with the phenomenon of "superlinear speed-ups" etc.).

The real performance of a parallel algorithm on a concrete parallel system is also influenced by other details like whether the hardware and the software allow an overlap of computation and communication etc. and, of course, the operating system and the compiler may have an essential influence, too.

There are two obvious reasons why an algorithm and/or a parallel system may show an unsatisfactory performance:

– load imbalance and
– communication overhead.

Load imbalance means that some processors have to do much more work than most of the others. In this case, most of the processors have to wait for others to finish their computation before a data exchange can be carried out. A purely sequential algorithm or a sequential phase in a parallel algorithm produces extreme load imbalance as only one processor is busy in that case.

Remark 7.2.1 (Basic rule for load balance) In general, it does not matter if one (or a few) processors have much less computational work than the average, but it is harmful for the performance of the parallel application if one (or a few) processors have much more computational work than the average. Then, most of the processors will be idle and have to wait for the overloaded ones to finish their parts of the computations. $\gg$

Communication overhead means that the communication and data transfer between the processors takes too much time compared to the effective computing time. This overhead may even lead to slow-down instead of speed-up when more and more processors are used. Summarizing:

> Avoiding load imbalance and limiting the communication overhead are the two most important principles in the parallelization of a given sequential algorithm.

7.2.2 Grid Partitioning for Jacobi and Red-Black Relaxation

If grid applications are to be implemented on parallel computers, *grid partitioning* is a natural approach. In this approach, the original grid Ω_h is split into P parts (subdomains, subgrids), such that P available processors can jointly solve the underlying discrete problem.

Each subgrid (and the corresponding "subproblem", i.e. the equations located in the subgrid) is assigned to a different process such that each process is responsible for the computations in its part of the domain.

The grid partitioning idea is widely independent of the particular boundary value problem to be solved and of the particular parallel architecture to be used. It is applicable to general d-dimensional domains, structured and unstructured grids, linear and nonlinear equations and systems of partial differential equations. Here and in the following sections, we focus on this approach.

Example 7.2.1 If we consider a parallel system consisting of 16 processors as in Figure 7.2.6, we are interested in a grid partitioning into *16 subgrids*. Obviously, a grid can be partitioned in various ways. Two examples, a 4×4 (2D) and a 1×16 (1D) partitioning, are shown in Figure 7.2.7. The partitionings generate certain artificial boundaries within the original domain.

When applying a 4×4 partitioning in the case of 16 processors, each process is responsible for the computations in about $n/4 \times n/4$ of the computational grid (n being the number of grid points in each direction). $\triangle$

In order to illustrate the basic ideas of grid partitioning as simply as possible, we start our discussion with the parallel treatment of ω-JAC as an iterative *solver* for Model Problem 1 on the square Cartesian grid.

Remember that ω-JAC is fully Ω_h parallel. When distributing the work performed during ω-JAC iterations to the available processors (all of the same performance), it is crucial that each processor obtains roughly the same amount of work at any stage of the solution procedure. Since the work of ω-JAC is the same at each interior grid point of Ω_h, a good load balance can easily be obtained: Just split Ω_h into as many (rectangular) subgrids as processors are available so that each subgrid contains *approximately the same number of grid points.* (If the processors have different performances, the distribution of grid points has to be adjusted accordingly.)

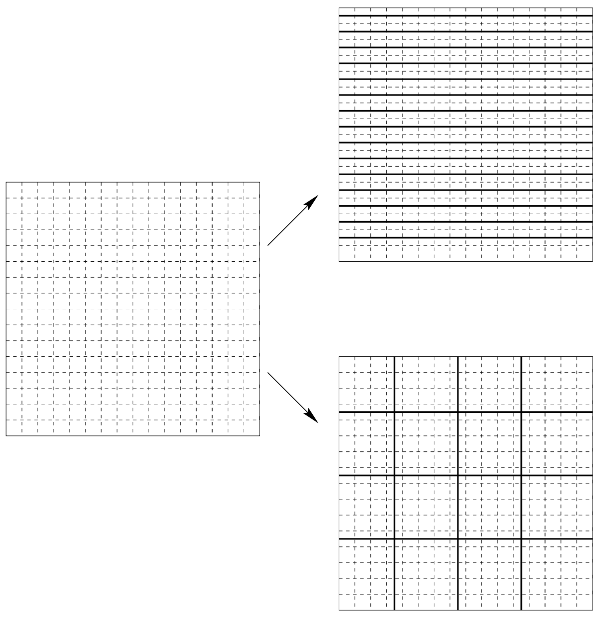


Figure 7.2.7: Partitioning of a 16×16 grid into 4×4 or 1×16 (rectangular) subgrids

Starting with some approximation u_h^m on the partitioned grid, each process can compute a new ω-JAC iterate u_h^{m+1} for each point in the interior of its subgrid. Near the subgrid boundaries, each process needs the old approximations u_h^m located at those points which are direct *neighbors* of its subgrid (see upper picture in Figure 7.2.8). In principle, each process can get this data from the neighbor process by sending and receiving it in a pointwise fashion, whenever such data is needed for the computations. But this approach would require a very large number of messages to be sent and received during the computations which would result in a large communication overhead due to the corresponding *large start-up time of sending many small messages.*

An efficient and elegant approach is obtained if each process does not only store the data belonging to its subgrid but also a copy of data located in neighbor subgrids in a *thin overlap area of a certain width w*, for example an overlap of one grid point ($w = 1$) (see lower picture in Figure 7.2.8).

Then, each process can perform a full ω-JAC iteration without any communication in between. Only after an iteration, the copies in the overlap areas have to be updated by communication so that the next ω-JAC iteration can be carried out. In our example, this communication can be realized easily: Each process sends all its data belonging to one side of the overlap area of a neighbor subgrid collectively (i.e. in one long message) to the corresponding "neighbor" process and receives the data corresponding to its own overlap area from that neighbor. This communication via the interconnection network of the processors is a typical example of *message passing* in a parallel system.

The situation is very similar if we replace ω-JAC by GS-RB. Starting with an approximation u_h^m, we can perform the first half step of GS-RB fully in parallel. Before the relaxation of the black points, we have to update the approximations u_h^{m+1} at the *red points of the overlap regions.* After the second half step of GS-RB, we have to update the approximation u_h^{m+1} in the *black* points of the overlap regions. We thus need two communication steps per GS-RB iteration instead of one for ω-JAC, but with only half of the points being exchanged in each step.

In this respect, computing phases and communication phases alternate during the execution of the parallel program for both ω-JAC and GS-RB. This alternation of computing and communication phases is natural for parallel programs based on the grid partitioning idea.

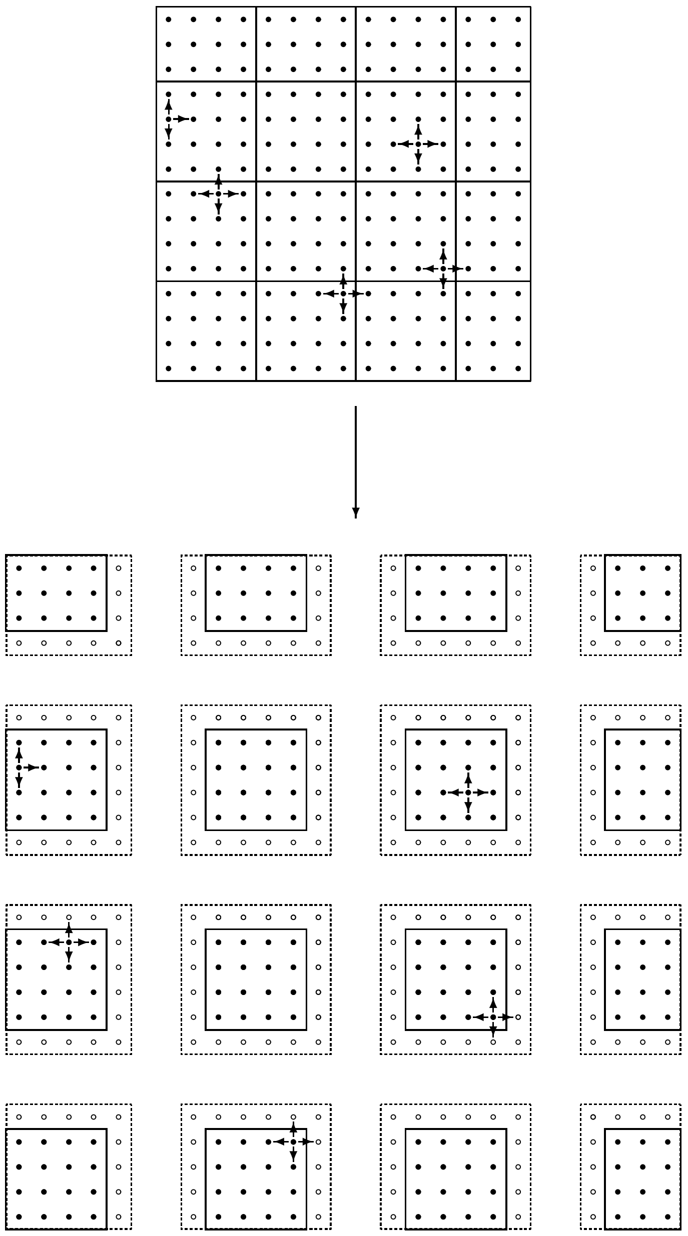


Figure 7.2.8: Introduction of an overlap area around the subgrids

The parallel algorithms ω-JAC and GS-RB as described above are *algorithmically equivalent* to their sequential versions: The results of both algorithms are the same. This algorithmical equivalence is not naturally achieved in all cases, for example not, if sequential methods such as GS-LEX are modified to improve their parallel properties.

Remark 7.2.2 (Parallel modification of GS-LEX) The degree of parallelism of GS-LEX is not satisfactory. An update of unknowns in GS-LEX depends on previously calculated values.

A modification of GS-LEX, which better suits the grid partitioning concept is to apply the GS-LEX smoother *only within a subgrid of a grid-partitioned application* (block). As a consequence, the resulting relaxation procedure is no longer a classical GS relaxation, but a combination of Jacobi-type relaxation and GS-LEX: All the blocks are treated simultaneously ("block"-Jacobi) and within each block GS-LEX is used (see Figure 7.2.9). $\gg$

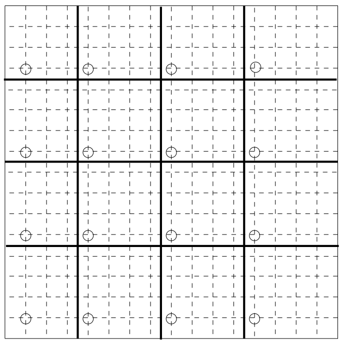


Figure 7.2.9: Modified GS-LEX: The subgrids are relaxed in parallel. A GS-LEX iteration is used in each block starting, for example, at the grid points marked by $\circ$.

7.2.3 Remarks on Speed-Up and Parallel Efficiency

In the following, we assume that we have a homogeneous parallel system with at least P processors (all of the same performance). We consider an algorithm for the solution of a given problem which runs on $P \geq 1$ processors in a computing time $T(P)$.

We are interested in the behavior of the speed-up S and the parallel efficiency E as a function of P. Ideally, one would like to achieve

$$S(P) \approx P, \text{ or equivalently } E(P) \approx 1.$$

This would mean that we are able to accelerate the computation by a factor close to P if we use P processors instead of one. In many cases, however, an efficiency

E close to 1 cannot be achieved. Usually, *the cost of inter-processor communication is not negligible*, in particular not, if for a given problem the number of processors P is increased. In other words, the efficiency E will decrease if a large parallel system (P large) is used for a small problem.

Increasing the number of processors is usually reasonable if the size of the problem (characterized, for example, by the number of grid points or unknowns N) is also "*scaled up*". Therefore, we are interested in $S = S(P,N),\ E = E(P,N)$ as a function of P *and* N.

The above definitions and assumptions are somewhat problematic in certain situations. *In fact, S and E as defined above usually do not merely represent the parallel properties of a given algorithm but include also other effects due to special processor characteristics.* For example, it may be unrealistic to assume that the entire application can also be executed on $P = 1$ processor of the system considered. The memory of only *one* processor may not be large enough for the whole application. Similar phenomena and complications occur when considering cache-based processors or vector processors. Here, the computing time may strongly depend on the arrangement of the unknowns in the memory and on the order and range of the loops in the computations. In such cases, the above definition would not be applicable, or – if used – one would observe a *superlinear* speed-up ($E(P) > 1$).

With respect to these complications, one always has to interpret the meaning of S and E according to the situation considered: One may use the above definitions, but should not interpret them naively.

Most importantly, the above quantities S, E and $\widetilde{E}$ have some principal limitations: The fact that an algorithm has a "good" efficiency on a certain parallel machine, does not at all mean that it is a *numerically* efficient algorithm.

Often numerically inefficient algorithms give (much) better parallel efficiencies E than more sophisticated and numerically efficient ones since they are much easier to parallelize. A typical example is a single grid iterative method like classical Jacobi-type iteration compared to multigrid. As the Jacobi-type iteration is fully parallelizable and *local*, one will find a very good parallel efficiency. Multigrid, however, is more involved so that its parallel efficiency will be worse than that of Jacobi. Nevertheless, the overall efficiency (i.e., the total computing time to solve a given problem) will typically be *better by far* for the multigrid approach.

Typically, concrete results on the parallel efficiency of an algorithm are obtained by measurements on a particular parallel computer. Often not only an algorithm is evaluated, but implicitly also different parallel computer architectures (and computer products from different vendors) are included in a comparison.

Some quantitative and qualitative results can also be derived theoretically using computer and communication models (performance prediction). A substantial number of performance models have been developed, some of which are very sophisticated. Some of them have been used for the evaluation of parallel multigrid methods, for example [79, 99].

One of the simplest communication models is already useful in practice. Here, the time needed for sending a message of length L is modeled by the formula

$$t_{comm} = \alpha + \beta L$$

with parameters α and β: α is the so-called *start-up time* which has to be spent whenever a message is to be sent, and $1/\beta$ is the *bandwidth* of the respective communication channel. For a realistic evaluation of the performance of a solution method on a particular parallel system, t_{comm} has to be compared with the computing time t_{comp} needed, e.g. for an arithmetic operation.

For a concrete parallel computer, it may be useful or even necessary to take the size of α and β into account when a specific algorithm is parallelized.

If α is large, the *number* of messages should be minimized, and if β is large, the communication *volume* is the issue.

In any concrete case, it depends on the hardware and on the application which of these parameters is crucial. On distributed memory systems, often α is the important parameter.

As an example, reconsider two partitioning options in Figure 7.2.7: In the square (2D) partitioning the communication volume is smaller (less than half) than in the strip (1D) partitioning, but the number of messages is larger (4 versus 2 per subgrid). The amount of data to be communicated is proportional to the total length of the interior boundaries, the number of messages equals the total number of edges of the subgrids.

7.2.4 Scalability and the Boundary-Volume Effect

In this section, we will show that the boundary-volume effect is the reason for the fact that grid partitioning leads, in general, to satisfactory efficiencies for sufficiently large problems. We first study the behavior of an algorithm if the number of processors P is fixed, but the size of the problem N is increased. We make the following assumptions:

- the given problem is characterized by local dependencies, as in finite difference and finite volume discretizations,
- the solution method has a sufficiently high degree of parallelism (for example, proportional to the number of grid points) and is sufficiently local and

– the number of grid points and the number of arithmetic operations per grid point are (asymptotically, for $N \to \infty$) equal for all subgrids.

Under these assumptions, we obtain

$$\boxed{E(P,N) \to 1 \text{ for } P \text{ fixed, } N \to \infty\ .} \tag{7.2.1}$$

for a large class of applications and algorithms. The result is known as the *boundary-volume effect*. The reason for this is that the ratio T_{comm}/T_{comp} (i.e. the overall time for communication versus the overall time for computation) behaves like the number of *boundary* grid points of (any of) the subgrids versus the number of *interior* grid points of the subgrids. For $N \to \infty$ and P fixed this means that $T_{comm}/T_{comp} \to 0$ and this – together with the other assumptions – implies $E(P,N) \to 1$.

For local methods, like ω-JAC and GS-RB, the boundary-volume effect holds trivially. But as we will see in Section 7.4.5, standard multigrid methods with sufficiently parallel smoothers also exhibit the boundary-volume effect.

The situation is less trivial if both the grid size N and the number of processors P are increased. In the grid partitioning applications considered here, it is reasonable to assume that the number of grid points per processor is constant if P is increased:

$$N/P = \text{ const} \quad \text{for } P \to \infty$$

The term "*scalability*" refers to this situation and assumption . We call a parallel algorithm and a corresponding application "*E-scalable*", if

$$E(P,N) \geq \text{ const } > 0 \text{ for } P \to \infty,\ N/P = \text{ const}.$$

Typically, local parallel algorithms like ω-JAC and GS-RB relaxation methods turn out to be E-scalable, not, however, *numerically* highly efficient algorithms such as multigrid methods (see Section 7.4.5).

As mentioned before, the terms of parallel efficiency and E-scalability are questionable because they do not take the numerical efficiency into account.

Different levels of parallelism are often distinguished, indicated by *coarse-grain, medium-grain and fine-grain* parallelism. With coarse grain parallelism macro-data flow computations are meant, in which each task is the computation of an entire group of columns; loop-level parallelism is known as medium-grain, whereas statement-level parallelism is typically known as "fine-grain", in which each task consists of only one or two floating point operations, such as a multiply-add pair.

7.3 Parallel Preconditioned Krylov Methods

In this chapter a number of parallel iterative methods are considered. We start with a coarse grain parallel method, domain decomposition with accurate solution of the subdomain problems, in Sections 7.3.1, 7.3.2. Thereafter some remarks are given on approximate solution of the subdomain problems (medium grain parallel) in Section 7.3.3. Section 7.3.4 contains a description of a number of fine grain parallel methods. Finally, a block Jacobi preconditioner combined with Deflation is given in Section 7.3.5. For further reading we refer to [138], [92], [37], [34] and [13]; Section 4.4.

7.3.1 An Overview of Domain Decomposition Methods

The second approach to parallelization of iterative methods lies in the domain decomposition (DD) framework. There are two books [129, 110] on DD methods available. In this brief overview many variants of domain decomposition will be mentioned. Specific implementations of efficient versions are detailed from Section 7.3.2 onwards.

One root of the domain decomposition development is the classical alternating Schwarz method. For simplicity, we consider the problem

$$\begin{aligned} -\Delta u &= b^{\Omega}(x,y) \quad (\Omega) \\ u &= b^{\Gamma}(x,y) \quad (\Gamma = \partial\Omega) \end{aligned} \tag{7.3.1}$$

in the rectangular domain $\Omega = (0,2)\times(0,1)$. In order to illustrate the DD idea, we use a decomposition of Ω into two overlapping domains

$$\Omega_1 = (0, 1+\delta)\times(0,1)$$
$$\Omega_2 = (1-\delta, 2)\times(0,1)$$

(see Figure 7.3.10). The parameter δ controls the overlap $\Omega_1 \cap \Omega_2$. By Γ_1 and

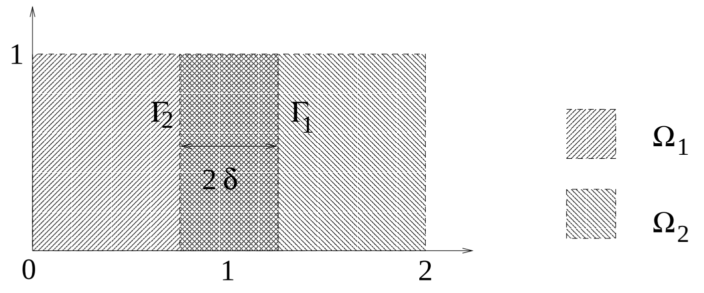


Figure 7.3.10: A domain Ω divided into two overlapping parts Ω_1 and Ω_2.

Γ_2, we denote the interior boundary lines

$$\Gamma_1 = \{(1+\delta, y) : 0 \le y \le 1\}, \;\; \Gamma_2 = \{(1-\delta, y) : 0 \le y \le 1\}.$$

In the classical alternating Schwarz method, the subproblems in Ω_1 and in Ω_2 are solved alternatingly, according to the iteration

$$\left.\begin{aligned} -\Delta u_1^{i+1/2} &= b^{\Omega} && (\Omega_1) \\ u_1^{i+1/2} &= b^{\Gamma} && (\partial\Omega_1 \setminus \Gamma_1) \\ u_1^{i+1/2} &= u_2^{i} && (\Gamma_1) \end{aligned}\right| \quad \begin{aligned} -\Delta u_2^{i+1} &= b^{\Omega} && (\Omega_2) \\ u_2^{i+1} &= b^{\Gamma} && (\partial\Omega_2 \setminus \Gamma_2) \\ u_2^{i+1} &= u_1^{i+1/2} && (\Gamma_2) \end{aligned} \tag{7.3.2}$$

with, for example, $u^0 = 0$. For $\delta > 0$, the above iteration is convergent under quite general conditions, its convergence speed depends, however, on the overlap parameter δ. For the above example, the convergence factor ρ behaves like

$$\rho \approx 1 - \alpha\delta + O(\delta^2)$$

with some constant α: *the smaller δ, the slower the convergence; for $\delta \to 0$ the convergence factor tends to* 1.

From a practical point of view, a solver for each of the subproblems has to be applied in each iteration step. For that purpose, we consider discrete analogs of the above subproblems and obtain by that a discrete version of the alternating Schwarz method on $\Omega_{1,h}$ and $\Omega_{2,h}$.

Many extensions and modifications of the classical method have been proposed: extension to many subdomains, so-called additive versions, DD with one or several coarse levels etc. For a rough survey on DD methods, we will use matrix terminology to avoid formalizing the discrete versions of different DD approaches and specifying the corresponding grids and spaces, which is not needed for our purposes. In this formulation, A is the matrix corresponding to the discrete version of the original problem (7.3.1) and $u^i, u^{i+1/2}, r^i, r^{i+1/2}$ $(m = 0, 1, ...)$ are "full" vectors corresponding to Ω_h.

We start with the alternating Schwarz method above. We denote by A_1 and A_2 the matrices belonging to the discrete analogs of the problems on Ω_1 and Ω_2 respectively. Using a defect formulation, as generally introduced in Section 4.3, for both half steps, one complete step of the Schwarz iteration reads

$$\begin{aligned} u^{i+1/2} &= u^i + P_1 A_1^{-1} R_1 r^i, \quad \text{where } r^i = b - Au^i \\ u^{i+1} &= u^{i+1/2} + P_2 A_2^{-1} R_2 r^{i+1/2}, \quad \text{where } r^{i+1/2} = b - Au^{i+1/2} \end{aligned}$$

Here R_1, R_2 denote matrices which restrict the full Ω_h vectors to $\Omega_{h,1}$ and $\Omega_{h,2}$, respectively, whereas P_1, P_2 extend the vectors defined on $\Omega_{h,1}, \Omega_{h,2}$ to full Ω_h vectors (extension by 0). That only the $\Omega_{1,h}$ and the $\Omega_{2,h}$ parts of $u^{i+1/2}$ and u^{i+1} are updated at each half step of the iteration is reflected by the terms $P_1 A_1^{-1} R_1$ and $P_2 A_2^{-1} R_2$. From the above representation we find that the iteration matrix for a complete iteration of the Schwarz method is given by

$$M = I - P_1 A_1^{-1} R_1 A - P_2 A_2^{-1} R_2 A + P_2 A_2^{-1} R_2 A P_1 A_1^{-1} R_1 A \ . \tag{7.3.3}$$

Here the last term is characteristic for the *alternating* character of the Schwarz method, in which the Ω_1 problem is solved in the first half step and the result is used (on Γ_2) to solve the Ω_2 problem. If we neglect this term, we obtain the *additive* variant of Schwarz's method characterized by

$$M_{\text{add}} = I - P_1 A_1^{-1} R_1 A - P_2 A_2^{-1} R_2 A \ . \tag{7.3.4}$$

In this setting, the Ω_1 and the Ω_2 problem can be solved simultaneously. In that respect, the additive variant is the natural parallel version of the Schwarz method.

For distinction, the original alternating approach (7.3.3) is called *multiplicative* ($M = M_{\text{mult}}$). The additive variant corresponds to a block Jacobi type method and the multiplicative variant to a block Gauss-Seidel type method where the Ω_1 and the Ω_2 problems characterize the blocks. If we generalize the additive and the multiplicative version of Schwarz's method to p domains, the corresponding iteration matrices become

$$M_{\text{add}} = I - \sum_{j=1}^{p} P_j A_j^{-1} R_j A \tag{7.3.5}$$

and

$$M_{\text{mult}} = \prod_{j=1}^{p} (I - P_j A_j^{-1} R_j A) \ . \tag{7.3.6}$$

The fact that the pure Schwarz methods, whether multiplicative or additive, are only slowly convergent for small overlap is worse for the many domains case. Therefore *acceleration techniques* have been introduced. We want to mention three of them.

Remark 7.3.1 (DD as a preconditioner) In the first approach, the Schwarz methods are not used as iterative *solvers*, but as *preconditioners*. The use of Schwarz methods as preconditioners is included in the above formalism by writing M in the form $M = I - CA$ where

$$C_{\text{add}} = \sum_{j=1}^{p} P_j A_j^{-1} R_j$$

in the additive case and

$$C_{\text{mult}} = [I - \prod_{j=1}^{p} (I - P_j A_j^{-1} R_j A)] A^{-1}$$

in the multiplicative case. (The original additive and multiplicative Schwarz methods are preconditioned Richardson iterations, with $\tau = 1$, (2.2.7), in this interpretation.) In practice, the above preconditioners are used with Krylov subspace methods. Under certain assumptions, the additive preconditioner C_{add} can be arranged in a symmetric way (for example, by choosing $P_j = R_j^T$) so that the conjugate gradient method can be used for acceleration. In the multiplicative case, usually GMRES is chosen for acceleration. $\gg$

Remark 7.3.2 (DD with a coarse grid) The second acceleration possibility which is usually combined with the first one, is characterized by a *coarse grid* Ω_H (in addition to Ω_h). This turns out to be necessary for satisfactory convergence in particular if p is large.

Formally, this "two-level approach" can be included in the above formalism by replacing

$$\sum_{j=1}^{p} \quad \text{by} \quad \sum_{j=0}^{p} \quad \text{and} \quad \prod_{j=1}^{p} \quad \text{by} \quad \prod_{j=0}^{p} \ .$$

Here, the term corresponding to index $j = 0$ characterizes the coarse grid part; it has the same form as the other summands (or factors).

Let us denote the typical diameter of a subdomain by H. In [38], it has been shown that in this case the condition number $\kappa(C_{\text{add}}A)$ fulfills

$$\kappa(C_{\text{add}}A) \leq \text{const } (1 + \frac{H}{\delta}) \ .$$

So, the condition number of the preconditioned system for the two level overlapping Schwarz method (with exact solution of the subdomain problems) is bounded independently of h and H, if the overlap is uniformly of width $O(H)$. Similar bounds also hold for the convergence factors of the multiplicative version.

Finally, we mention that a lot of theoretical and practical work is devoted to *non-overlapping DD approaches* (also known as *iterative substructuring* methods). In the non–overlapping approaches, the idea of *parallel* treatment of the subproblems is more strongly addressed than in the overlapping case. For a general description and survey articles, see, e.g., [129, 110, 147] and the references therein.

Among the best–known and most tested non-overlapping domain decomposition methods for elliptic PDEs are the FETI and Neumann–Neumann families of iterative substructuring algorithms. The former is a domain decomposition method with Lagrange multipliers which are introduced to enforce the intersubdomain continuity. The algorithm iterates on the Lagrange multipliers (the dual variables) and is often called a dual domain decomposition method (see [116, 88] and the references therein). The Neumann–Neumann method is a primal domain decomposition algorithm which iterates on the original set of unknowns (the primal variables). A unified analysis for both, FETI and Neumann–Neumann algorithms, has been given in [88].

7.3.2 Domain Decomposition (Accurate Subdomain Solution)

In this subsection a domain decomposition algorithm is used to solve a system of equations in parallel. For more details we refer to [18].

Consider the linear system

$$Au = b. \tag{7.3.7}$$

We decompose A into blocks such that each block corresponds to all unknowns in a single subdomain. For two subdomains one obtains

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{7.3.8}$$

where A_{11} and A_{22} represent the subdomain discretization matrices and A_{12} and A_{21} represent the coupling between subdomains. A domain decomposition iteration for (7.3.7) has the following form:

$$u^{i+1} = (I - M^{-1}A)u^i + M^{-1}b, \tag{7.3.9}$$

where M denotes a block Gauss-Seidel, or a block Gauss-Jacobi matrix:

$$M = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \text{(Gauss-Seidel)}, \quad M = \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix} \text{(Gauss-Jacobi)}.$$

Block Gauss-Seidel and block Gauss-Jacobi iterations are algebraic generalizations of the Schwarz domain decomposition algorithm. Similar to Schwarz domain decomposition, in each iteration, subdomains are solved using values from the neighboring block. For instance formula (7.3.9) for domain 1 becomes

$$u_1^{i+1} = u_1^i + A_{11}^{-1}(b_1 - A_{11}u_1^i - A_{12}u_2^i),$$

where u_2^i are the values from the neighboring block. The iterate update $\delta u_1^{i+1} = u_1^{i+1} - u_1^i$ is computed from

$$A_{11}\delta u_1^{i+1} = b_1 - A_{11}u_1^i - A_{12}u_2^i$$

by a direct method or an iterative solution method where the method is stopped after an accurate solution has been obtained.

When the subdomain problems are solved accurately the system

$$M^{-1}Au = M^{-1}b \tag{7.3.10}$$

can be reduced to a system only involving unknowns near the interfaces. Suppose that the unknowns in the vicinity of the interfaces are denoted by u_r and the remaining ones by u_{nr}. When the components are ordered as $u = \begin{pmatrix} u_{nr} \\ u_r \end{pmatrix}$, system (7.3.10) has the form

$$M^{-1}Au = \begin{pmatrix} I & R \\ 0 & D \end{pmatrix} \begin{pmatrix} u_{nr} \\ u_r \end{pmatrix} = \begin{pmatrix} g_{nr} \\ g_r \end{pmatrix},$$

which can be reduced to $Du_r = g_r$ and $u_{nr} = g_{nr} - Ru_r$. System $Du_r = g_r$ is much smaller than (7.3.10) and is known as the *interface equations.* The interface equations are solved by a Krylov subspace method (for instance GMRES). This implies that matrix vector products $y_r = Du_r$ have to be calculated. To obtain this product a solution of the subdomain problems is needed (see [18], Section 3).

We consider a *parallel implementation* of the GMRES accelerated domain decomposition method. This means that the block Jacobi (additive Schwarz) algorithm is used. The parallel implementation is based on message passing using MPI. In the parallel computer (or cluster of workstations) each node is assigned certain subdomains to solve. The host program controls the acceleration of the domain decomposition algorithm using GMRES. Because of the reduction of (7.3.10) to a system concerning only the interface unknowns it is not necessary to parallelize GMRES. So the GMRES acceleration procedure is only executed on the host. Computation of the matrix vector product is performed in parallel.

Each node receives initial interface data (parts of the vector u_r^i), solves the assigned subdomain problems and sends the results (parts of the vector u_r^{i+1}) back to the host. Note that this programming model is not completely SPMD (single processor multiple data), because the host program has additional functions like acceleration and writing output.

The amount of computation to solve a subdomain problem accurately is much larger than the amount of communication. Therefore this is a coarse grain parallel method. For results on a cluster of workstations we refer to [18].

7.3.3 Domain Decomposition (Approximate Subdomain Solution)

In the previous subsection the subdomain problems are solved accurately. Since these problems have a similar nonzero structure as the original matrix, and since they may still be quite large, it is reasonable to solve them using a second Krylov subspace iteration. A question which arises naturally, addresses the tolerance to which these inner iterations should converge. It seems senseless, for example, to solve the subdomain problems with a much smaller tolerance than is desired for the global solution. The influence of the accuracy of the subdomain solution on the convergence has been investigated in [19]. A large gain in CPU time has been obtained on a sequential computer, when the subdomain accuracy is relatively low.

Note that if the subdomains are solved using a Krylov subspace method such as GMRES, then the approximate solution is a function of the right-hand side, which is the residual of the outer iteration. Furthermore, if the subdomain problems are solved to a tolerance, the number of inner iterations may vary from one subdomain to another, and in each outer iteration. The effective preconditioner is therefore a nonlinear operator and varies in each outer iteration. A variable preconditioner presents a problem for the GMRES method: namely the recurrence relation no longer holds. To allow the use of a variable preconditioner the GCR method is used as acceleration method.

Note that the GCR method is applied to the global matrix. So the GCR method should also be parallelized. For the details of this we refer to the following subsection. Depending on the required accuracy the method can be seen as a coarse grain (high accuracy), medium grain (medium accuracy) or fine grain (low accuracy) parallel method.

7.3.4 Parallel Iterative Methods

Many iterative methods (basic iterative methods, or preconditioned Krylov subspace methods) consist of 4 building blocks: vector updates, inner products, matrix vector products, and the solution of lower or upper triangular systems. First we give a description of the data distribution for our model problem. Thereafter the parallel implementation of the building blocks for the preconditioned GMRES method is considered. Parallel implementations of Krylov subspace methods are given in [125] and [92]. The large number of inner products used in the GMRES method can be a drawback on distributed memory computers, because an inner product needs global communication. In general, the preconditioner is the most difficult part to parallelize.

Data Distribution On distributed memory machines it is important to keep information in local storage as much as possible. For this reason we assign storage space and update tasks as follows. The domain is subdivided into a regular grid of rectangular blocks. Each processor is responsible for all updates of variables associated with its block. An extra row of auxiliary points is added to the boundaries of a block to provide storage space for variables used in matrix-vector products and preconditioner construction.

Vector update, inner product The *vector update* ($y = y + \alpha x$) is easy to parallelize. Suppose the vectors x and y are distributed. Once α is sent to all processors, each processor can update its own segment independently.

The work to compute an *inner product* is distributed as follows. First the inner product of the vector elements that reside on a processor is calculated for each processor. Thereafter these partial inner products are summed to form the full inner product. To obtain the full inner product global communication is required.

Communication time is the sum of the start-up time (latency) and the sending time. On many parallel computers the sending time is an order of magnitude less than the latency. For this reason it is attractive to combine communications as much as possible. Using the Classical Gram-Schmidt (CGS) method in the GMRES algorithm all inner products can be computed independently. So the communication steps can be clustered, which saves much start-up time. A drawback of CGS is that the resulting vectors may be not orthogonal due to rounding errors. Therefore, the Modified Gram-Schmidt (MGS) method is preferred, which is stable with respect to rounding errors. However, the inner products are calculated sequentially when MGS is used in the original GMRES method. So clustering of the inner product communications is impossible. Since for the Cray T3D, the latency is relatively small, we use in our T3D specific code [145] the Modified Gram-Schmidt method for stability reasons. On a computer with a relative large latency, it is better to use an adapted GMRES method ([134], [10]) where a parallel (clustered) variant of the MGS method can be used.

Table 7.3.1 contains the Megaflop rates for the inner product on the Cray T3D. In theory the maximum Megaflop rate of 1 processor is 150 Mflop/s. The observed flop rates are much lower: 33.5 for the inner product (Table 7.3.1). It appears that memory access is the most time consuming part. Note that on 1 processor the Megaflop rate increases for an increasing vector length (pipelining). For the

	measured		
p	1	16	128
#elem per proc			
32	10	34	194
1024	28	331	2346
8192	33	496	3875
65536	33	529	4212

Table 7.3.1: Inner product performance in Megaflops per second on the Cray T3D

inner product this leads to an expected rate of 4288 Mflop/s on 128 processors. For long vectors the observed rate (see Table 7.3.1) is very close to this value.

Matrix vector product To calculate a *matrix vector product*, we divide the matrix in blocks. As an example we consider the matrix given in Figure 7.3.11. The parts A_{11}, A_{12} are on processor 1, A_{21}, A_{22}, A_{23} on processor 2 etc. For the vector u which is divided on the processors we need the following communication: u_1 to processor 2, u_2 to processor 1 and 3 etc, note that a 1D torus network is well suited for this approach. Then $y_1 = A_{11}u_1 + A_{12}u_2$ is calculated on processor 1 and $y_2 = A_{21}u_1 + A_{22}u_2 + A_{23}u_3$ is calculated on processor 2 etc. Other schemes are possible but this illustrates the ideas to parallelize a matrix vector product.

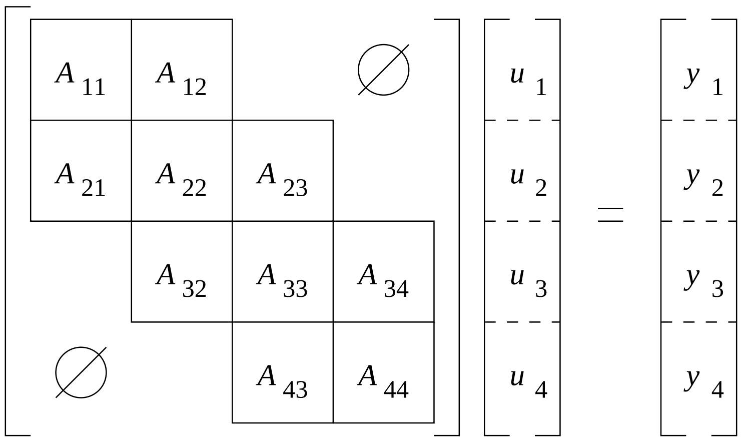


Figure 7.3.11: The calculation of $A * u$

Preconditioning We restrict ourselves to an ILU preconditioning. This is in general the most difficult part to parallelize, because lower and upper triangular systems have to be solved. A lower triangular system is sketched in Figure 7.3.12. Note that recurrences prohibit parallelization of the computation of $L^{-1}b$. One

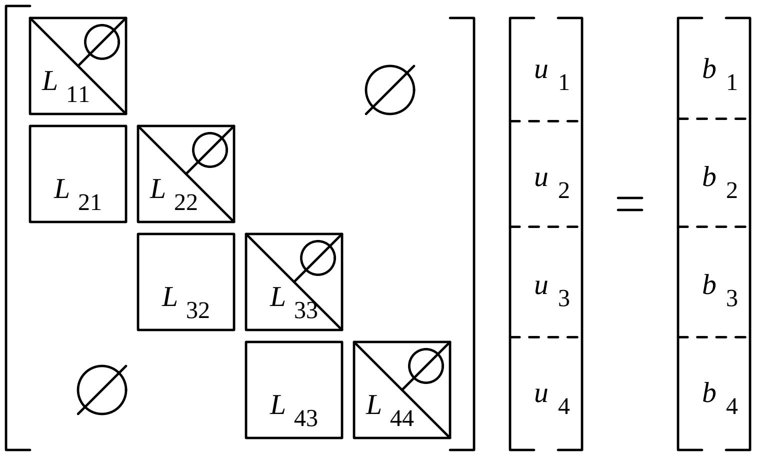


Figure 7.3.12: The calculating of $L^{-1}b$

can only start the computation of u_2 if u_1 is known. In many references a slightly adapted ILU decomposition is considered, which has better parallel properties. To parallelize the ILU preconditioner the couplings between the blocks are deleted. For most of these preconditioners the rate of convergence deteriorates when the number of blocks (processors) increases. When overlapping blocks are used the convergence behavior depends only slightly on the number of blocks. However, overlapping costs extra work. Another possibility for structured grids is to use the original ILU preconditioner and try to parallelize this. Both approaches are considered in more detail.

Parallel adapted ILU
In order to delete the couplings between the blocks one takes $L_{21} = 0, L_{32} = 0$ and $L_{43} = 0$ (looks like block Jacobi) [125]. This leads to very good parallel code. However, the number of iterations may increase because the parallel preconditioner is worse than the original preconditioner. In order to reduce this effect, it is suggested in [112] to construct incomplete decompositions on slightly overlapping domains. This requires communication similar to that of matrix vector products. In [112] good speedup is observed for this idea.

Parallel original ILU
Parallelization of the construction of L, U, and the solution of the triangular systems is comparable. So we only consider the parallel implementation of the solution of $Lu = b$. The algorithm is explained for a matrix originating from a 5-point stencil. This approach is denoted as staircase parallelization of the ILU preconditioner [145]. A related approach is to calculate the unknowns on a diagonal of the grid, which is used for vectorization.

A rectangular computational domain is first decomposed into p strips parallel to the x_2-axis. We assume that the number of strips is equal to the number of processors. The number of grid points in the x_d-direction is denoted by n_d. For ease of notation we assume that n_1 can be divided by p and set $n_x = n_1/p$ and $n_y = n_2$. The index i refers to the index in x_1-direction and j to the index in x_2-direction. The k^{th} strip is described by the following set $S_k = \{(i,j)|i \in [(k-1)\cdot n_x+1, k\cdot n_x]$, $j \in [1, n_y]\}$.

The vector of unknowns is denoted by $u(i,j)$. For a 5-point stencil it appears that in the solution of $Lu = b$, unknown $u(i,j)$ only depends on $u(i-1,j)$ and $u(i,j-1)$. The parallel algorithm now runs as follows: first all elements $u(i,1)$ for $(i,1) \in S_1$ are calculated on processor 1. Thereafter communication takes place between processor 1 and 2. Now $u(i,2)$ for $(i,2) \in S_1$ and $u(i,1)$ for $(i,1) \in S_2$ can be calculated in parallel etc. After some start-up time all processors are busy (Figure 7.3.13).

Table 7.3.2 is copied from [145]. Note that when the grid size is large enough the total time per unknown is independent of the number of processors, which means that the method is scalable. In figures 7.3.14 and 7.3.15 we present the percentage of the total time for the various parts of GMRES. Figure 7.3.14 contains the results for $p = 8$ and an increasing grid size. It appears that the preconditioner vector product is the most time consuming part, it takes 65 % of the time for a small grid size and 45 % for a large grid size. In Figure 7.3.15 the results are shown for the Gustafsson model, the grid size increases linearly with the number of processors. There is only a small increase in the percentage used for the preconditioner vector product. This model suggests, as expected, that the preconditioner can be a bottle-neck especially if the number of grid cells in

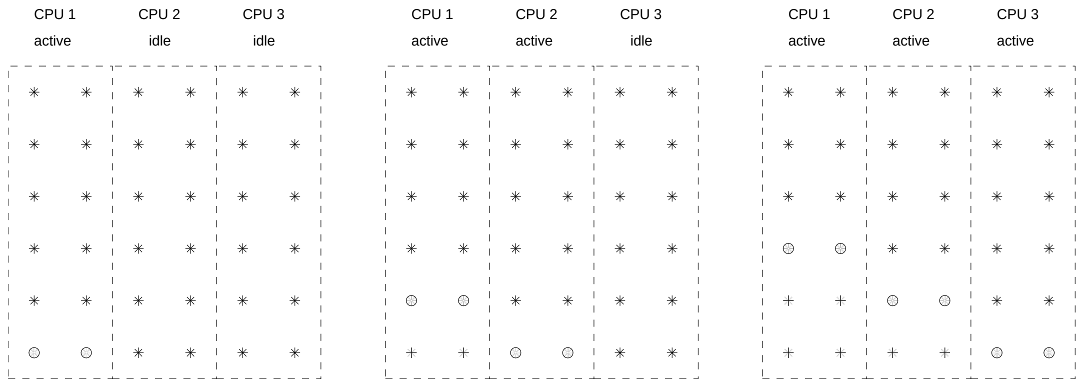


Figure 7.3.13: The first stages of the staircase parallel solution of the lower triangular system $Lu = b$. The symbols denote the following: $*$ nodes to be calculated, o nodes being calculated, and + nodes that have been calculated.

p	1	2	4	8	16	32	64	128	256
32×8	259	645	1204	2510					
64×16	185	402	597	965	1969				
128×32	179	340	395	518	830	1694			
256×64	163	319	331	380	487	833	1521		
512×128		306	311	338	373	478	740	1443	
1024×256				317	335	375	469	731	1444
2048×512						354	374	492	722

Table 7.3.2: Measured total time per unknown in μ seconds for the solution of a Poisson equation on the Cray T3D (p = number of processors)

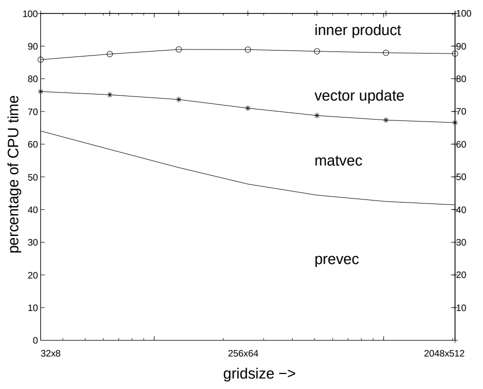
the x_1-direction per processor is small.

7.3.5 The Block Jacobi Preconditioner combined with Deflation

We consider an elliptic partial differential equation discretized using a cell-centered finite difference method on a computational domain Ω. Let the domain be the union of $\hat{M}$ nonoverlapping subdomains Ω_m, $m = 1, \ldots, \hat{M}$. Discretization results in a sparse linear system $Au = b$, with u, $b \in \mathrm{R}^N$. When the unknowns in a subdomain are grouped together one gets the block system:

$$\begin{bmatrix} A_{11} & \ldots & A_{1\hat{M}} \\ \vdots & \ddots & \vdots \\ A_{\hat{M}1} & \ldots & A_{\hat{M}\hat{M}} \end{bmatrix} \begin{pmatrix} u_1 \\ \vdots \\ u_{\hat{M}} \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_{\hat{M}} \end{pmatrix}. \tag{7.3.11}$$

In this system, the diagonal blocks A_{mm} express coupling among the unknowns defined on Ω_m, whereas the off-diagonal blocks A_{mn}, $m \neq n$ represent coupling



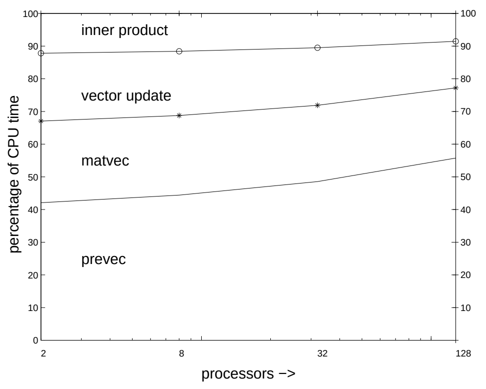


Figure 7.3.14: The percentage of time used by the various parts (8 processors)

Figure 7.3.15: The percentage of time used by the various parts (grid size $256 \cdot \sqrt{p/2} \times 64 \cdot \sqrt{p/2}$)

across subdomain boundaries. The only nonzero off-diagonal blocks are those corresponding to neighboring subdomains.

In order to solve system (7.3.11) with a Krylov subspace method we use the block Jacobi preconditioner:

$$M = \begin{bmatrix} A_{11} & & \\ & \ddots & \\ & & A_{\hat{M}\hat{M}} \end{bmatrix}.$$

When this preconditioner is used, systems of the form $Mv = r$ have to be solved. Since there is no overlap the diagonal blocks $A_{mm}v_m = r_m$, $m = 1, \ldots, \hat{M}$ can be solved in parallel. In our method a blockwise application of the RILU preconditioner is used.

Example 7.3.1 *As a test example, we consider a Poisson problem, discretized with the finite volume method on a square domain. We do not exploit the symmetry of the Poisson matrix in these experiments. The domain is composed of a* $\sqrt{p} \times \sqrt{p}$ *array of subdomains, each with an* $n \times n$ *grid. With* $h = \Delta x = \Delta y = 1.0/(n\sqrt{p})$ *the discretization is*

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{i,j}.$$

The right hand side function is $f_{i,j} = f(ih, jh)$*, where* $f(x, y) = -32(x(1-x) + y(1-y))$*. Homogeneous Dirichlet boundary conditions* $u = 0$ *are defined on* $\partial\Omega$*, implemented by adding a row of ghost cells around the domain, and enforcing the condition, for example,* $u_{0,j} = -u_{1,j}$ *on boundaries. This ghost cell scheme allows natural implementation of the block preconditioner as well.*

For the tests of this section, GCR is restarted after 30 iterations, and modified

Gram-Schmidt was used as the orthogonalization method for all computations. The solution was computed to a fixed tolerance of 10^{-6}. *We compare results for a fixed problem size on the* 300×300 *grid using 4, 9, 16 and 25 blocks. In Table 7.3.3 the iteration counts are given. Note that the number of iterations increases when the number of blocks grows. This implies that the parallel efficiency decreases when one uses more processors. In the next sections we present two different*

	$p=4$	$p=9$	$p=16$	$p=25$
RILU	341	291	439	437

Table 7.3.3: Number of iterations for various number of blocks

approaches to diminish this drawback.

It is well known that the convergence of an *overlapping block preconditioner* is nearly independent of the subdomain grid size when the physical overlap region is constant. To describe the overlapping block preconditioner we define the subdomains $\Omega^*_m \subset \Omega$. The domain Ω^*_m consists of Ω_m and n_{over} neighboring grid points (see Figure 7.3.16). The matrix corresponding to this subdomain is denoted by

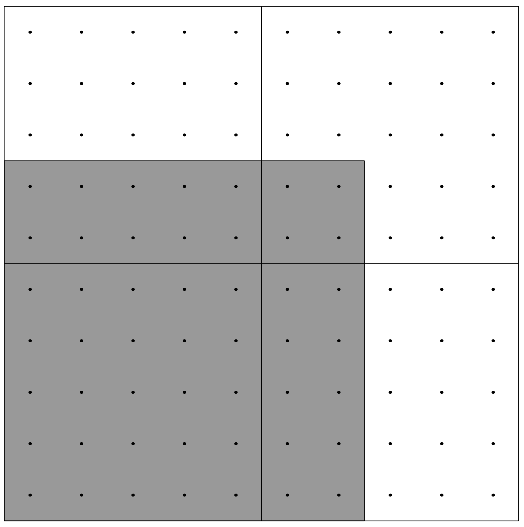


Figure 7.3.16: The shaded region is subdomain Ω^*_1 for $n_{over} = 2$

A^*_{mm}. Application of the preconditioner goes as follows: given r compute v using the steps

a. r^*_m is the restriction of r to Ω^*_m,

b. solve $A^*_{mm} v^*_m = r^*_m$ in parallel,

c. form v_m, which is the restriction of v^*_m to Ω_m.

A related method is presented by Cai, Farhat and Sarkis [29]. A drawback of overlapping subdomains is that the amount of work increases proportional to n_{over}. Furthermore it is not so easy to implement this approach on top of an existing software package.

We present the *deflation acceleration* only for the symmetric case. In our implementation we use the Deflated ICCG method as defined in [144]. To define the Deflated ICCG method we need a set of projection vectors $v_1, ..., v_{\hat{M}}$ that form an independent set. The projection on the space A-perpendicular to span$\{v_1, ..., v_{\hat{M}}\}$ is defined as

$$P = I - VE^{-1}(AV)^T \text{ with } E = (AV)^T V \text{ and } V = [v_1 ... v_{\hat{M}}] \, .$$

The solution vector u can be split into two parts $u = (I - P)u + Pu$. The first part can be calculated as follows $(I - P)u = VE^{-1}VAu = VE^{-1}V^T b$. For the second part we project the solution u^j obtained from DICCG to Pu^j. DICCG consists of applying CG to $L^{-T}L^{-1}P^T Au = L^{-T}L^{-1}P^T b$.

The Deflated ICCG algorithm reads (see Reference [144]):

```
**DICCG**
$j = 0,\ \hat{r}^0 = P^T r^0,\ p^1 = z^1 = L^{-T}L^{-1}\hat{r}^0;$
while $\|\hat{r}^j\|_2 >$ accuracy do
    $j = j + 1;\ \alpha_j = \frac{(\hat{r}^{j-1}, z^{j-1})}{(p^j, P^T A p^j)};$
    $u^j = u^{j-1} + \alpha_j p^j;$
    $\hat{r}^j = \hat{r}^{j-1} - \alpha_j P^T A p^j;$
    $z^j = L^{-T}L^{-1}\hat{r}^j;\ \beta_j = \frac{(\hat{r}^j, z^j)}{(\hat{r}^{j-1}, z^{j-1})};$
    $p^{j+1} = z^j + \beta_j p^j;$
end while
```

For the Deflation acceleration we choose the vectors v_m as follows:

$$v_m(i) = 1, \;\; i \in \Omega_m, \text{ and } v_m(i) = 0, \;\; i \notin \Omega_m. \qquad (7.3.12)$$

We are able to give a sharp upperbound for the effective condition number of the deflated matrix, used with and without classical preconditioning [47]. This bound provides direction in choosing a proper decomposition into subdomains and a proper choice of classical preconditioner. If grid refinement is done keeping the subdomain resolutions fixed, the condition number can be shown to be independent of the number of subdomains.

In parallel, we first compute and store $((AV)^T V)^{-1}$ in factored form on each processor. Then to compute $P^T Ap$ we first perform the matrix-vector multiplication

$w = Ap$, requiring nearest neighbor communications. Then we compute the local contribution to the restriction $\tilde{w} = V^T w$ and distribute this to all processors. With this done, we can solve $\tilde{e} = ((AV)^T V)^{-1}\tilde{w}$ and compute $(AV)^T\tilde{e}$ locally. The total communications involved in the matrix-vector multiplication and deflation are a nearest neighbor communication of the length of the interface and a global gather-broadcast of dimension $\hat{M}$.

Example 7.3.2 *Block preconditioner and overlap*
We consider a Poisson problem on a square domain with Dirichlet boundary conditions and a constant right-hand-side function. The problem is discretized by cell-centered finite differences. We consider overlap of 0, 1 and 2 grid points and use A_{mm}^{-1} in the block preconditioner. Table 7.3.4 gives the number of iterations necessary to reduce the initial residual by a factor 10^6 using a decomposition into 3×3 blocks with subgrid dimensions given in the table. Note that the number of iterations is constant along the diagonals. This agrees with domain decomposition theory that the number of iterations is independent of the subdomain grid size when the physical overlap remains the same.

	overlap		
grid size	0	1	2
5×5	10	8	7
10×10	14	9	8
20×20	19	13	10
40×40	26	18	14

Table 7.3.4: Iterations for various grid sizes

In the second experiment we take a 5×5 grid per subdomain. The results for various number of blocks are given in Table 7.3.5. Note that without overlap the number of iterations increases considerably, whereas the increase is much smaller when 2 grid points are overlapped. The large overlap (2 grid points on a 5×5 grid) that has been used in this test, is not affordable for real problems.

Example 7.3.3 *Deflation acceleration*
We perform the same experiments using the Deflation acceleration (see Table 7.3.5). Initially we see some increase of the number of iterations, however, for more than 16 blocks the increase levels off. This phenomenon is independent of the amount of overlap. The same conclusion holds when block RILU is used instead of fully solving the subdomain problems.

Finally we present some timing results on the Cray T3E for a problem on a

	overlap			overlap+Deflation		
decomposition	0	1	2	0	1	2
2×2	6	5	4	6	4	4
3×3	10	8	7	11	6	6
4×4	15	9	7	14	9	6
5×5	18	12	9	16	10	8
6×6	23	13	10	17	11	9
7×7	25	16	12	17	12	10
8×8	29	17	12	18	13	10
9×9	33	19	14	18	14	11

Table 7.3.5: Iterations for various block decompositions with and without Deflation acceleration (subdomain grid size 5×5)

480×480 *grid. The results are given in Table 7.3.6. In this experiment we use GCR with the block RILU preconditioner combined with Deflation acceleration. Note that the number of iterations decreases when the number of blocks increases. This leads to an efficiency larger than 1. The decrease in iterations is partly due to the improved approximation of the RILU preconditioner for smaller subdomains. On the other hand when the number of blocks increases, more small eigenvalues are projected to zero which also accelerates the convergence (see [47]). We expect that there is some optimal value for the number of subdomains, because at the extreme limit there is only one point per subdomain and the coarse grid problem is identical to the original problem so there is no speedup at all.*

p	iterations	time	speedup	efficiency
1	485	710	-	-
4	322	120	5	1.2
9	352	59	12	1.3
16	379	36	20	1.2
25	317	20	36	1.4
36	410	18	39	1.1
64	318	8	89	1.4

Table 7.3.6: Speedup of the iterative method using a 480×480 grid

7.4 Parallelism of Multigrid Components

Here, we will discuss the parallel aspects of multigrid. Although suitable multigrid *components* may be highly parallel, the *overall structure* of standard multigrid is intrinsically not fully parallel for two reasons. The first reason is that the grid levels are run through sequentially in standard multigrid. The second reason is that the *degree of parallelism* of multigrid is different on different grid levels (i.e., small on coarse grids). A corresponding theoretical complexity discussion is presented in Section 7.4. The problem of *communication*, which is an important aspect in practical parallel computing, is not addressed there.

The *degree of parallelism* reflects the number of processors that can contribute to the computation of an algorithm or of algorithmic components. It can be interpreted as the number of operations that can be carried out in parallel. A parallel system that would exploit the degree of parallelism of a fully parallel relaxation method would need to have at least as many processors P as grid points N. (In practice, typically only a limited number of processors is available. In this case, of course, the asymptotic considerations do not apply.

7.4.1 Parallel Components for Poisson's Equation

We reconsider Model Problem 1 and a corresponding multigrid method with standard coarsening. The crucial multigrid component with respect to parallelism usually is the *smoothing procedure.* We first recall the parallel properties of ω-JAC, GS-LEX and GS-RB on Ω_h from Section 4.2: ω-Jacobi relaxation is *fully Ω_h parallel.* We say that its degree of parallelism is $\#\Omega_h$. Correspondingly, for GS-LEX the degree of parallelism is less than or equal to $(\#\Omega_h)^{\frac{1}{2}}$ and for GS-RB the degree of parallelism is $\frac{1}{2}\#\Omega_h$. For Model Problem 1, GS-RB has the best smoothing properties of these relaxations *and* it is highly parallel. For discretizations with larger stencils, *multi-color Gauss-Seidel* relaxation or JAC-RB have good parallelization properties.

The three other multigrid components can be applied in parallel for all (interior) grid points:

- *Calculation of defects:* The defect computations at different grid points are independent of each other and can be performed in parallel for all points of Ω_h.

- *Fine-to-coarse transfer:* The computations and transfer operations at different grid points are again independent of each other and can be performed in parallel for all points of the coarse grid Ω_{2h}. This applies to all of the restriction operators (Full Weighting, Half Weighting and – trivially – injection) discussed so far.

- *Coarse-to-fine transfer:* Interpolation from coarse to fine refers to the Ω_h grid points. Typically, the operations to be performed are different for different types of grid points (see, for example, (4.3.28) and Figure 4.3.6, for the case of bilinear interpolation), but the operations can be performed in parallel.

So far, the discussion of the parallelism in multigrid components has been oriented to the levels Ω_h and Ω_{2h}.

Clearly, all considerations carry over to any other (coarser) pair of levels h_k, h_{k-1}, with the corresponding (smaller) degree of parallelism.

7.4.2 Parallel Complexity

On the coarse grids, the degree of parallelism decreases substantially since

$$\#\Omega_k << \#\Omega_\ell \quad \text{for } k < \ell \ ,$$

until finally $\#\Omega_0 = 1$ if the coarsest grid consists of only one grid point. The problem of the very coarse grids leads to multigrid specific parallel complications which do not occur in classical single grid algorithms.

This crucial impact of the coarse grids increases, the more often the coarse grids are processed in each cycle. A parallel W-cycle, for example, has a *substantially* different parallel complexity than a parallel V-cycle.

This can be seen by the following results of a simple analysis of parallel multigrid complexity. Here, the parallel complexity is the number of parallel steps/operations that a processor has to carry out assuming that the degree of parallelism is fully exploited.

Result 7.4.1 *Let N denote the overall number of fine grid points (unknowns) in case of Model Problem 1:*

$$N = \#\Omega_h = (n-1)^2$$

and let us consider the Red-Black Multigrid Poisson Solver. Then we obtain the sequential and parallel complexities (the sequential complexity being simply the number of required floating point operations) listed in Table 7.4.7. For MGI (i.e. multigrid iteration) we assume an accuracy of ε (error reduction) as stopping criterion.

Proof: The sequential complexities (total computational work) in Table 7.4.7 have already been derived in Section 4.4.1. The parallel complexities are obtained

	Cycle type	sequential	parallel
MGI	V	$O(N \log \varepsilon)$	$O(\log N \log \varepsilon)$
	F	$O(N \log \varepsilon)$	$O(\log^2 N \log \varepsilon)$
	W	$O(N \log \varepsilon)$	$O(\sqrt{N} \log \varepsilon)$
FMG	V	$O(N)$	$O(\log^2 N)$
	F	$O(N)$	$O(\log^3 N)$
	W	$O(N)$	$O(\sqrt{N} \log N)$
Lower bound (for any solver)		$O(N)$	$O(\log N)$

Table 7.4.7: Sequential and parallel complexities of 2D-multigrid

by summing up the number of times the multigrid levels are processed in the corresponding algorithms. In the case of the W-cycle, for example, the number of visits of the level $\Omega_{\ell-k}$ per cycle is $O(2^k)$ which sums up to $O(2^\ell) = O(\sqrt{N})$. The other results can be obtained similarly.

That at least $O(\log N)$ parallel steps are required for any solver follows from the fact that the solution of a discrete elliptic problem in general depends on all discrete entries. Therefore, full (weighted) sums of all entries have to be evaluated which requires at least $O(\log N)$ parallel steps. $\Box$

The two main observations from Table 7.4.7 are:

- The step from V- to W-cycles gives a substantial increase of parallel complexity, from
$$O(\log N) \quad \text{to} \quad O(\sqrt{N})$$
(whereas in the sequential case the number of operations is only multiplied by a factor of $\frac{3}{2}$).

- The parallel FMG approach (based on a V-cycle) needs
$$O(\log^2 N)$$
parallel steps instead of the theoretically lower bound of $O(\log N)$ (whereas in the sequential case FMG is optimal in the sense that it requires $O(N)$ operations).

The increase of complexity for W-cycles means that V-cycles are preferable from a massively parallel point of view. This is a relevant aspect if we want to work with a highly parallel system which consists of nearly as many processors as grid points.

7.4.3 Grid Partitioning and Multigrid

So far, we have described the idea of grid partitioning and discussed some details of its realization only for "local" iterative methods such as ω-JAC and GS-RB relaxation. Grid partitioning is also the natural parallelization approach for multigrid. *Its extension of the single grid case to parallel multigrid is straightforward.*

We consider two grids in a specific multigrid algorithm (correction scheme) in order to discuss the impact of the parallelization on the multigrid components. Let us assume that we perform the parallel computations during the smoothing iteration on the fine grid as described in Section 7.2.2, with an overlap width of $w = 1$. The computations on the coarse grid can be assumed to be of the same type as those on the fine grid. In particular, the arithmetic relations are *local* with respect to the grid level on both grids. Since the coarse grid problem is a direct analog of the fine grid problem, we can perform grid partitioning on the coarse grid accordingly. In general, there is no reason to change the partitioning of the subdomains and the mapping to the processors. On the contrary, if the information on the same *geometric* points on different grids Ω_h and Ω_H were allocated to different processors, additional communication among these processors would be required during the intergrid transfers. Therefore, the grid partitioning of the coarse grid is carried out as for the fine grid.

If extended to multigrid, this means that one processor is responsible for the computations on a sequence of grid levels on the same subdomain.

Of course, the overlap idea has to be adapted according to the different grid levels. On each grid, we need an overlap region of at least $w = 1$ in order to be able to perform the parallel smoothing steps and defect calculations. Since the distance between two adjacent grid points increases on coarse grids, the "geometric size of overlap regions" will be different on different grid levels.

As on the fine grid, communication on the coarser grids is "local" with respect to the corresponding grid.

Let us now assume that we have performed one (or several) GS-RB smoothing steps on the fine grid and that the approximations in the overlap region have been updated afterwards.

For the prolongation back to the fine grid, we have a similar situation. After performing one or several smoothing steps on the coarse grid and updating the overlap regions afterwards, we can immediately perform bilinear interpolation because all data of coarse grid points required for the interpolation at a fine grid point is available in the same process.

It is thus sufficient in this parallel multigrid algorithm to perform communication only after each smoothing half step.

If we employ Full Weighting in the above parallel multigrid algorithm the computation of the defects in the overlap areas (which are required) can be performed by the computation of the corresponding defects in the adjacent neighbor process followed by an additional communication step for these defects.

7.4.4 Multigrid and the Very Coarse Grids

If we process coarser and coarser grids during a multigrid cycle, first smaller and smaller numbers of grid points are mapped to each process. Then more and more processes do not have grid points on these very coarse grids any more and finally only one or a few processes have one (or a few) grid points. At the same time, the relative communication overhead on the grids (as compared to the time for arithmetic computations) increases and may gradually dominate the arithmetic work on the coarsest grids. This can result in a significant loss of performance for the overall parallel application. In particular in W-cycles these coarse grids are processed often.

Whereas idling processes on very coarse grids seem to be the main problem at first sight, experience and theoretical considerations show that *the large communication overhead on the very coarse grids is usually more annoying than the idling processes.* Idling processes means that the communication relations among the processes change. Algorithmically, the computations continue as usually in the processes which still have grid points. The other ones are idle until the multigrid algorithm returns to a level, on which these processes have grid points again. Actually, the following aspects are important:

- On coarse grids, the ratio between communication and computation becomes worse than on fine grids, up to a (possibly) large communication overhead on very coarse grids.
- The time spent on very coarse grids in W-cycles may become unacceptable.
- On very coarse grids we may have (many) idle processes.
- On coarse grids, the communication is no longer local (in the sense of finest grid locality).

Remark 7.4.1 In practice, it depends on the particular application under consideration to which extent the coarse grids reduce the parallel efficiency. For Poisson's equation on a Cartesian grid (a scalar equation with few arithmetic operations per grid point), for example, the communication overhead on the coarse grids may have a strong impact on the parallel efficiency. In that respect Poisson's equation is a hard test case for multigrid on a parallel system. For more complex applications such as nonlinear systems of equations on general curvilinear grids (applications with many arithmetic operations per grid point), the effect is much less severe. Really large scale parallel applications (which are the ones that indeed need parallel processing) are dominated by computational work on fine grids, at least in the case of V-or F-cycles. $\gg$

Of course, the arithmetic work performed by active processes on the coarse grids is very small in comparison to the load balanced computations on the fine grid. There are various approaches to overcome the *problem of the communication overhead.*

a. Firstly, if the overall performance on a particular parallel system is not strongly affected by the communication on coarse grids, we can well stay with this approach. This variant may often be the best one.

b. A possibility to reduce frequent communication on very coarse grids is known as the *agglomeration technique* [75]. The idea of this approach is to "agglomerate" grid points to new "process units" and to redistribute these new units to a subset of the active processes. For example, instead of using 64 processors for 64 coarse grid points and performing communication between them, it can be more efficient to group these 64 grid points to sets of 4 or 16 points (corresponding to 16 or 4 process units, see Figure 7.4.17) or even to one group of 64 points (to one process unit). This means that only a part of the processors is responsible for the computations on these very coarse grids, the majority of processors being idle. *Communication is avoided by this grouping or agglomeration* – and the fact that most of the processors become idle turns out to be acceptable for many computers, although the agglomeration itself requires, of course, communication.

   Agglomeration can also be applied within FMG: On the very coarse grids only a subset of the available processes is responsible for the computations, whereas the full number of processes is employed on finer grids.

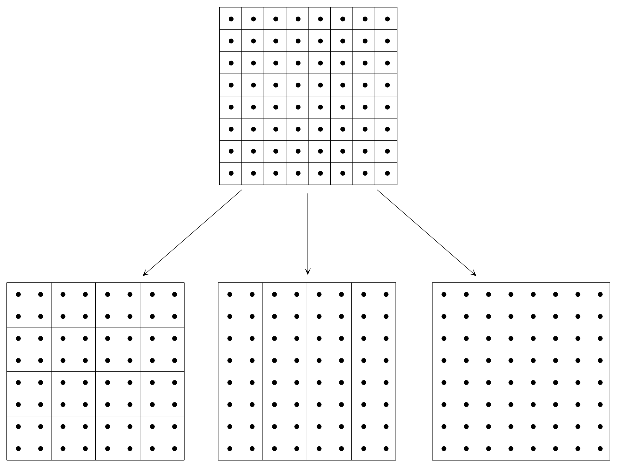


Figure 7.4.17: Agglomeration of 64 grid points to 16, 4 or 1 processes

c. A third approach to treating the coarse grid problem is to redefine what the coarsest grid is, i.e. reduce the number of levels. One way is to define the coarsest grid such that *each process has at least one grid point*. Since the coarsest grid then consists of $O(P)$ points, the parallel algorithm is in general different from the sequential one: The parallel algorithm does not process the *coarsest possible grid*. The efficiency of this strategy then depends particularly on the solution procedure on the coarsest grid. One will usually employ an iterative method on the coarsest grid. Of course, it is important to use parallel iteration schemes with good convergence properties. Remember that, for example, ω-GS-RB (GS-RB with an optimal overrelaxation parameter ω) has much better *convergence* properties than GS-RB ($1 - O(h)$ instead of $1 - O(h^2)$) for Model Problem 1.

   The effect of this truncation of a multigrid W-cycle is illustrated in Figure 7.4.18.

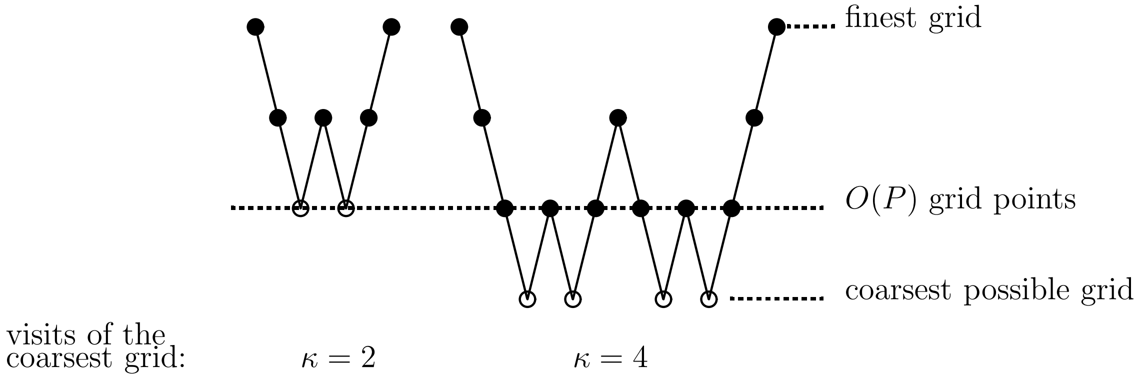


Figure 7.4.18: W-cycle without and with the coarsest possible grid

7.4.5 Boundary-Volume Effect and Scalability in the Multigrid Context

As discussed in Section 7.2.4, simple local iterative methods like ω-JAC and GS-RB relaxations are E-scalable, and the boundary-volume effect is trivially fulfilled for them. The situation is somewhat different for multigrid: The boundary-volume effect is valid for multigrid methods, too, whereas, instead of the E-scalability, only a somewhat weaker property can be achieved.

Result 7.4.2 *For Model Problem 1 and the Red Black Multigrid Poisson solver, we have*

$$E(P, N) \longrightarrow 1 \quad \textit{for } N \longrightarrow \infty,\ P \textit{ fixed} \ .$$

With respect to E-scalability, we obtain the results in Table 7.4.8 for $N \to \infty$, $N/P = const.$

Cycle	$E(P,N)$
V	$O(1/\log P)$
F	$O(1/\log^2 P)$
W	$O(1/\sqrt{P})$

Table 7.4.8: Asymptotic parallel efficiencies for $N \to \infty$, $\frac{N}{P} = \text{const}$

Proof: On the fine grids, the ratio between communication and computation tends to 0 if $N \to \infty$ and P is fixed. For $N \to \infty$, the relative influence of the fine grids will thus finally dominate the influence of the coarse grids, so that the boundary-volume effect is maintained.

From

$$E(P) = \frac{S(P)}{P} = \frac{T(1)}{T(P) \cdot P} \ , \quad \frac{N}{P} = \text{const} \ \Rightarrow \ P \sim N \ ,$$

we obtain

V-cycle:	$T(P) = O(\log P \log \varepsilon)$	$\Rightarrow$	$E(P) = O(1/\log P)$
F-cycle:	$T(P) = O(\log^2 P \log \varepsilon)$	$\Rightarrow$	$E(P) = O(1/\log^2 P)$
W-cycle:	$T(P) = O(\sqrt{P} \log \varepsilon)$	$\Rightarrow$	$E(P) = O(1/\sqrt{P})$
FMG (based on V-cycle):	$T(P) = O(\log^2 P)$	$\Rightarrow$	$E(P) = O(1/\log^2 P)$

□

Compared to E-scalability, we loose a logarithmic term for V-cycles and a square root term for W-cycles. The reason for loosing the logarithmic term is the increasing number of grid levels: If more and more processors are engaged and the problem gets larger and larger ($N/P = \text{const.}$), also the number of levels increases ($\sim \log N$) (if the coarsest level is fixed).

Remark 7.4.2 (Vector computers) Many of our parallel considerations carry over to the situation of vector computing: The parallelism of multigrid components can also be exploited on vector computers, and the complications on coarse grids (lower degree of parallelism) are also seen there (shorter vectors on coarse grids). $\gg$

7.5 Parallel Molecular Dynamics

The challenge in parallel computing is to devise an algorithm which makes optimal use of the parallel processors. Since the introduction of parallel computers considerable effort has been put in efficient parallel codes for molecular dynamics simulations. Both, algorithms for SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) architectures, have been devised with considerable success. In this section we discuss parallelization strategies for molecular dynamics on a Linux Cluster. This is a MIMD architecture, but the molecular dynamics is done as an SPMD (Single Program Multiple Data) calculation. In fact, when we are dealing with a system with short range forces, molecular dynamics simulations lend themselves particularly well for parallelization.

It is important to keep all processors as much as possible at work, with as little as possible idle time, e.g., waiting for information from other processors. Good load balancing is essential for attaining a high parallel efficiency.

7.5.1 Parallelization

The distributed memory architecture has proven to be the most popular architecture for many MD applications. Here we will confine our discussion of parallel molecular dynamics to this architecture only. Two different strategies have been used for efficient particle simulations on parallel architectures: *particle decomposition* and *domain decomposition*. We consider particle decomposition first.

- *Particle Decomposition.*
  Suppose we wish to carry out a molecular dynamics simulation of N particles on p processors. In particle decomposition methods we assign N/p particles to each processor. Thus particles $1 \cdots N/p$ reside on processor 1, $N/p + 1 \cdots 2N/p$ on processor 2 and so on. Now each processor can calculate in parallel the forces acting on the particles residing on it. On processor 1 the forces acting on the first N/p particles are evaluated, while on processor 2 the forces acting on particles $N/p+1 \cdots 2N/p$ are evaluated and so on. In principle, each particle can interact with any other particle, as there is usually no relation between particle number and relative position with respect to other particles. Hence each processor needs to know the positions of all other particles. These positions need to be communicated from the other processors through the communication network of the parallel computer. Once the forces are known the equations of motion can be integrated. Since this involves only velocities and forces of the particles themselves this can again be done in parallel.
  The precise implementation of the parallel algorithm may depend on the topology of the communication network of the computer. It is easiest to consider the algorithm for a ring topology of processors, as shown in Figure 7.5.1. Each processor can communicate with its two nearest neighbors. Assume there are P processors, numbered $0, \cdots, P-1$. Then, in line of the ring structure, if $s = p + q \geq P$ then $s \to s - P$, whereas if $s = p - q < 0$ then $s \to s + P$. The force calculation then proceeds as follows:

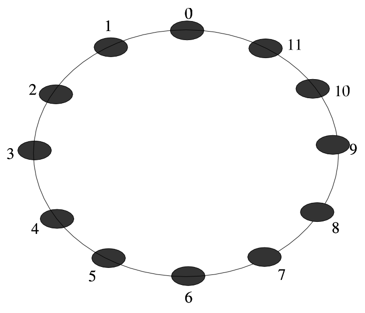


Figure 7.5.19: Simple ring topology for a set of 12 processors.

```
Parallel algorithm

step 1
On each processor *compute* forces between the $N/p$ particles residing on it.
step 2
do $p = 1, P/2$
    for all processors $q$: *send* coordinates on processor $q + p$ to processor $q$.
    Note that this can be done in successive steps each time passing on the coordinates to a neighbouring processor.
    *Compute* the forces $Fij$ between particles $i$ on processor $q$ and particles $j$ on processor $q + p$;
    *add* pair-forces $Fij$ to the forces $Fi$ acting on particle $i$ on processor q;
    *send* the forces $Fji = -Fij$ to processor $p + q$ and add these to the force $Fj$ acting on particle $j$ on that processor.
enddo
```

Because of Newton's third law we only have to make $P/2$ steps communicating the particle positions and an equal number of steps for communicating the forces. If there is no correlation between the positions of the particles and their index each processor will have approximately an equal number of force evaluations, so that load balancing is maintained in the algorithm. Moreover, the force calculation is almost scalable if an efficient local algorithm, such as the linked cell method, Appendix A.0.1 (or neighborhood tables), is used for the force evaluation. Then, if we double the number of particles N and the number of processors P, the same number of force evaluations have to be carried out. There is some overhead as a somewhat larger set of particles has to be searched through, but this is usually very small. However, the number of communication steps doubles if we double the number of processors, so that this is clearly not scalable. If the communication time T_{comm} cannot be neglected, as it is usually the case, the particle decomposition algorithm is not scalable. This is a serious limitation of the particle decomposition technique.

- *Domain Decomposition*
  The domain decomposition algorithm divides the physical space of the system, i.e. the simulation cell, into contiguous domains and maps these onto the processors. This decomposition works well when the effective range of the interactions between particles is small compared to the domain size. Particles inside a given domain interact with particles in other domains only if they reside in a boundary region with a thickness equal to the range of the interactions. This can be handled by passing boundary region information between neighbouring processors, as shown in Figure 7.5.1. As the simulation proceeds particles will move from their domain to another. Obviously the particles coordinates, velocities and other characteristics have to be removed from the processor associated with the old domain and transferred to the processor associated with the new domain. Thus, the number of particles in any processor fluctuates as the simulation proceeds.

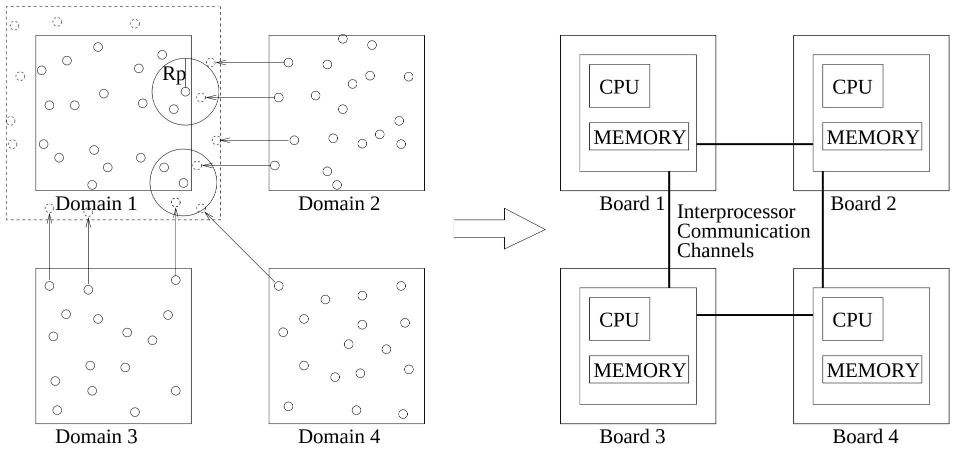


Figure 7.5.20: Domain decomposition and mapping onto four processor boards. The boundary regions to be copied for domain 1 are shown explicitly. The minimum width of these regions is the radius of the pair-list R_p.

The domain decomposition can be done in a number of ways. In case of a 3-dimensional molecular dynamics simulation the simulation cell can be divided into slabs (1-dimensional decomposition), columns (2-dimensional

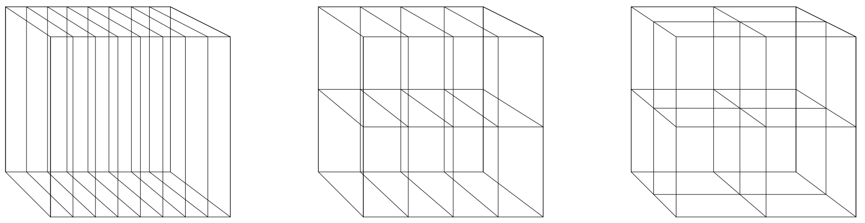

a. 1D decomposition. b. 2D decomposition. c. 3D decomposition.

Figure 7.5.21: Domain decomposition in 1,2 and 3 dimensions over 8 processors.

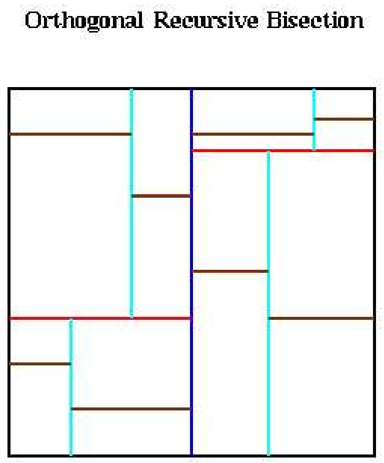
decomposition) and cubes (truly 3 dimensional decomposition). This is shown schematically in Figure 7.5.21 for a cubic box and eight processors. The choice of a particular decomposition should be taken with care because it can affect the efficiency of the calculation. One obviously wishes to minimize the amount of time spend on communication. If the width of the boundary regions is small compared to the width of the domain the amount of communication is roughly proportional to the total surface area of the interfaces between the domains. We should therefore aim for a small surface-to-volume ratio to minimize the communication time. For large numbers of processors (≈ 16 or higher) a truly 3D decomposition is then the most efficient.

For homogeneous systems a decomposition into domains of equal size will ensure good load balancing, since fluctuations in the numbers of particles are relatively small. The compressibility theorem makes this plausible. For systems with inhomogeneous particle distributions is may be necessary to vary the size of the domains according to the workload on a processor. When the inhomogeneous distribution varies in time *dynamic load balancing* may be required. This is relatively easy for one-dimensional decomposition, but rather involved in 2 and 3 dimensions.

7.6 Parallelization of Barnes-Hut Algorithm

There have been many parallelizations of Barnes-Hut algorithms both on shared memory machines and distributed memory machines. In general, two similar strategies have been pursued on both architectures: the spatial partitioning and the tree partitioning approaches (named according to the way the work is divided).

Spatial Partitioning This method divides the large square in which all the particles lie into p non-overlapping sub-rectangles, each of which contains an approximately equal number of particles, and assigns each rectangle to a processor.



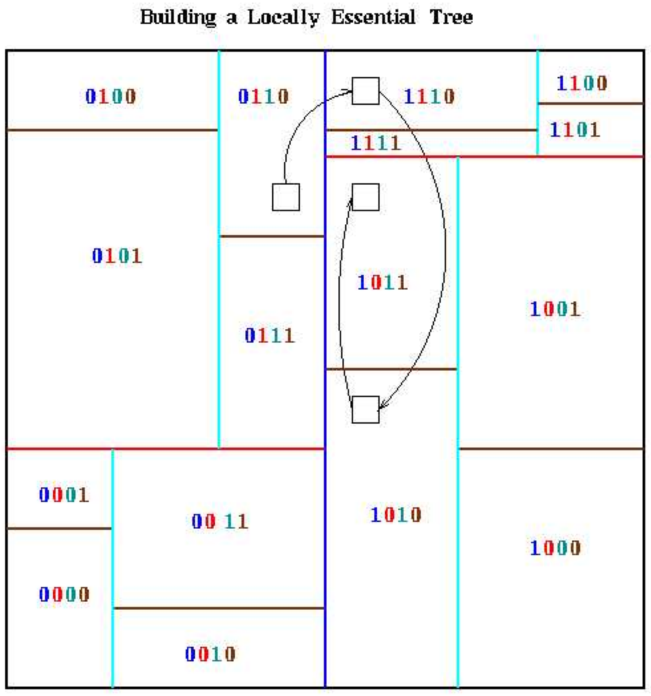
This is also called Orthogonal Recursive Bisection (ORB) in the literature. It works as follows. First, a vertical dividing line is found such that half the particles are on one side, and half are on the other, as shown below with the blue line. Next, the particles in the left and right halves are divided in half by horizontal red lines, to get 4 partitions with equally many particles. These are in turn bisected with vertical cyan lines, horizontal brown lines, and so on, until there are as many partitions as processors.

In order for processor k to run Barnes-Hut on the particles it owns, it needs that subset of the quadtree that would be accessed while computing forces for its own particles; this subset is called the Locally Essential Tree (LET). The LET includes the whole subset of the quadtree whose leaves contain the particles inside processor k, as well as "neighboring" tree nodes. Here is a somewhat simplified description of the LET. Recalling the Barnes-Hut algorithm, we let n be a node in the quadtree not containing any particles in processor k, let $D(n)$ be the length of a side of the square corresponding to n, and let $r(n)$ be the shortest distance from any point in n to any point in the rectangle owned by processor k. Then if

a. $\frac{D(n)}{r(n)} < \theta$ and $\frac{D(parent(n))}{r(parent(n))} \leq \theta$, or

b. $\frac{D(n)}{r(n)} \leq \theta$

n is part of the LET of processor k. The idea behind condition (1) is that the mass and center of mass of n can be used to evaluate the force at any point in processor k, but no ancestor of n in the tree has this property. In particular, no children of n are in the LET. Condition (2) says we need the ancestors of all such nodes as well. The advantage of conditions (1) and (2) is that processor k can examine its own local subtree, and decide how much of it any other processor j needs for processor j's LET, knowing only the size and location of the rectangle



owned by processor j (this is needed to determine $r(n)$). The parallel algorithm given below for computing the LETs takes advantage of this.

To describe the algorithm, we assume there are $p = 2^m$ processors. Each m-bit processor number is determined by the orthogonal bisection process; bit i is 1 or 0 depending on the side of the i-th bisecting line on which the processor lies. This is illustrated below for the same partition as above, with bits in processor numbers color coded to indicate to which bisecting line they correspond.

The following algorithm is of the type of SPMD code.MYPROC is a variable equal to the local processor number.

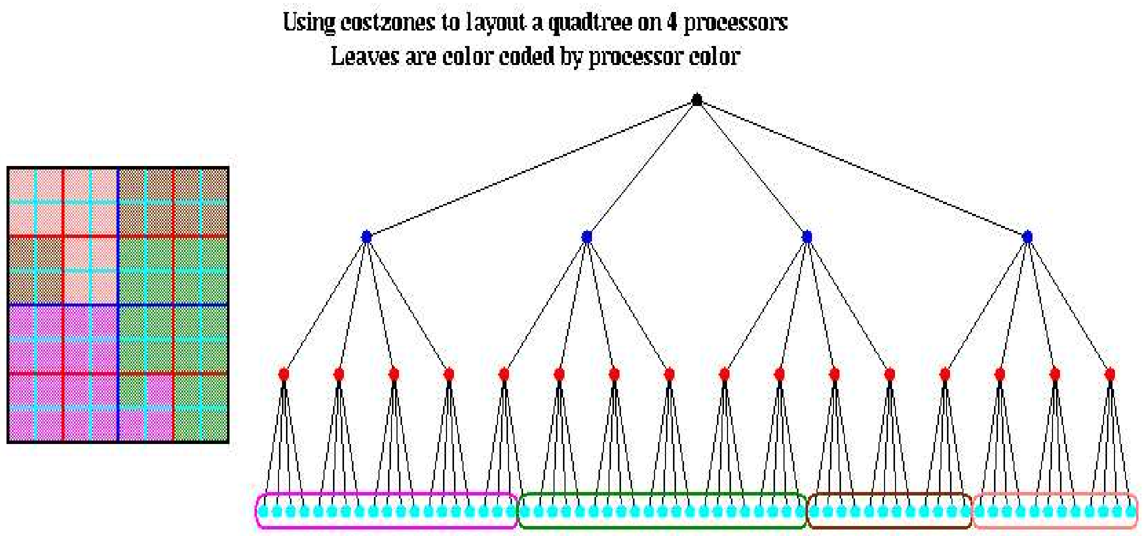
```
Compute local quadtree T
LET = T
for $i = 0$ to $m - 1$ /* ... $p = 2^m$ */
    OTHERPROC = MYPROC xor $2^i$ /* ... flip $i$-th bit of MYPROC */
    Compute the subset of LET that any processor on other side
        of bisecting line $i$ might want
    Send this subset to OTHERPROC
    Receive corresponding subset from OTHERPROC, and
        incorporate it into LET
end for
prune LET of unnecessary parts
```

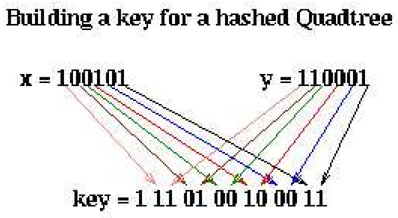
The path a particular part of the quadtree might take to reach its final destination is circuitous, as illustrated by the small black box in the last figure. To get from processor 0110 to 1011, which needs it, it first goes to processor 1110 (leftmost bit flipped), then to processor 1010 (next bit flipped), then it remains in 1010 because it is on the correct side of the cyan line, and finally it goes to processor 1011 (rightmost bit flipped).

This algorithm was implemented [173] both on the Intel Delta, a distributed memory machine, and the Stanford Dash, a shared memory machine. The Delta implementation was the largest astrophysical simulation ever done at that time, and won the Gordon Bell Prize at Supercomputing 92. The Intel Delta has 512 processors, each one an Intel i860. Over 17 million particles were simulated for over 600 time steps. The elapsed time was about 24 hours. The shared memory implementation was somewhat slower than the Tree Partitioning algorithm, and much more complicated to program.

Tree Partitioning This approach is sometimes called *costzones* in the shared memory implementation, and *hashed octtree* in the distributed memory implementation. The shared memory implementation is much simpler, so we describe it first. After the quadtree is formed, a *work estimate* is associated with each leaf (particle), indicating how much work is needed to compute the force for that leaf. If the tree is being constructed for the first time, this estimate may simply be the depth of the leaf in the tree. If, as is more common, the tree has been computed before during a previous time step, then an estimate from the last call to Barnes-Hut is available (if particles do not move quickly, then the tree structure changes little from time step to time step). Given the work estimate for each leaf, we can compute an estimate of the total work W. Now the leaves may be partitioned by breaking them into blocks, or costzones, each of which has a total amount of work close to W/p, so the load is balanced. Such a partition is shown below, where we have circled leaves belonging to a costzone, as well as correspondingly color coded the squares to which they correspond. Note that it is possible for a costzone to consist of noncontiguous squares (such as the brown processor), although this method generally assigns contiguous squares to processors. This contiguity is important for memory locality reasons, since the algorithm accesses more nearby squares than distant ones.

The shared memory implementation simply used reads of remote data to fetch the parts of the quadtree needed on demand. This data is read-only while traversing the tree to compute forces, and is cached in each processor. Because each processor works on (mostly) contiguous regions of space, the same parts of the quadtree are needed over and over again, so caching is very effective. The speedup was 15 on a 16 processor Dash, which is quite good. The disadvantage of the shared memory approach is that cache sizes have to grow proportionally to the number of processors to maintain high efficiency. Also, the locality afforded


Using costzones to layout a quadtree on 4 processors
Leaves are color coded by processor color

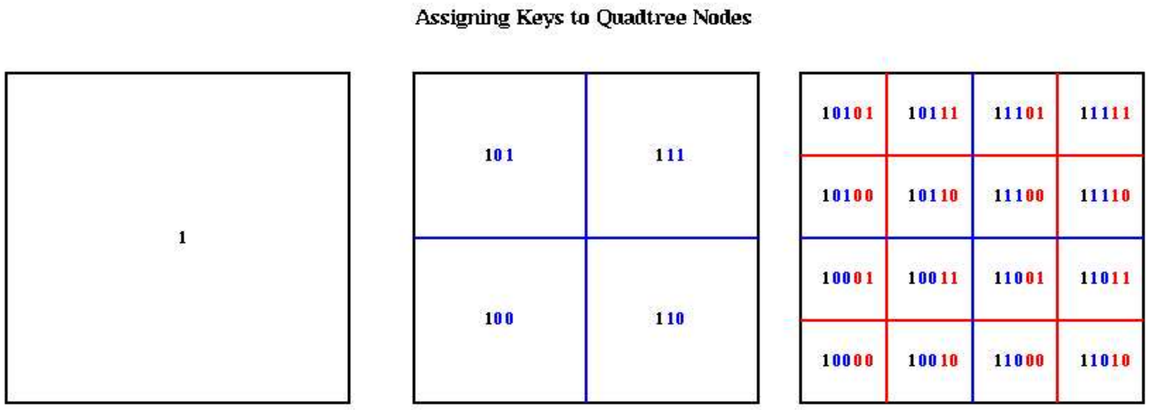

Building a key for a hashed Quadtree
x = 100101
y = 110001
key = 1 11 01 00 10 00 11

by this approach is not perfect, as illustrated by the brown processor (the brown squares are scattered) in the above figure.

The distributed memory algorithm is much more complex, and uses a distributed data structure called a *hashed octtree*. The idea is to associate a unique key with each node in the quadtree (as before, we limit ourselves to 2D for ease of exposition). Then a *hashing function* hash(key) will map each key to a global address in a hash table at which to find the data associated with the node. The hash table is then distributed across processors using a technique much like costzones. The hash function permits each processor (with high probability) to find exactly where each node is stored without traversing links in a linked tree structure, and so minimizing the number of communications necessary find get the data at a node.

Here is how the key is computed. Let (x, y) be the coordinates of the center of a square. We may think of them as bit strings, as shown below. The corresponding key is obtained by interleaving these bits, from right (least significant) to left (most significant), and putting an extra 1 at the left (to act as a "sentinel").

The nodes at the top of the quadtree are larger, and therefore may be repre-



sented by shorter bit patterns. A more complete picture of the keys for each level of the quadtree is shown below. The bits of each key are color coded according to which level of the quadtree determines them.

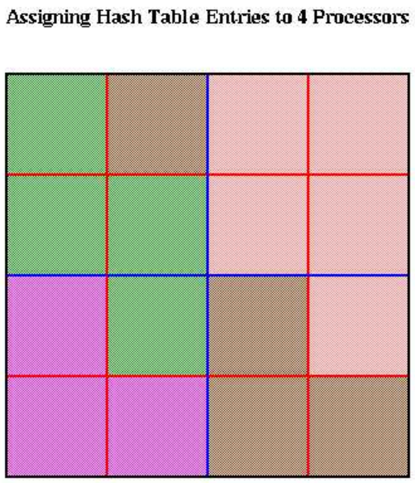
The hash function hash(key) is a simple bit-mask of the bottom h bits of each key. Thus, the top $(h/2)+1$ quadtree levels ($(h/3)+1$ octtree levels) have unique value of hash(key). Each hash table entry include a pointer to a list of boxes with the same value of hash(key), a list of which child cells are actually present, and mass and center of mass data.

The hash table is partitioned in a method similar to costzones. Suppose $h=4$, so there is one cell in the hash table for each of the 16 squares on level 3 of the quadtree (the rightmost figure of the three above). Then the keys for each of the 16 squares determine a linear order for these squares, from 10000 to 11111. We want to break this sequence of 16 squares into p contiguous subsequences to assign to processors. To balance the load, we want each each subsequence to require the same work. We do this similarly to costzones, where to each square we associate a cost, and choose the subsequences so the sum of the costs in each subsequence is nearly equal. This is illustrated below, for $p=4$. Note that a processor may be assigned noncontiguous regions of space, as with costzones.

Finally, the Barnes-Hut tree is traversed by sending messages to access remote parts of the hash table. There are many standard techniques to lower the communication needed, including overlapping communication and computation, prefetching parts of the tree, processing tree nodes in the order they become available rather in the order they appear in the tree, and caching.

Here are some performance results of an astrophysics calculation taken from literature, again on a 512 processor Intel Delta. 8.8 million particles are involved in the simulation, which are initially uniformly distributed. The time for one Barnes-Hut step is 114 seconds, or 5.8 Gflops. The breakdown of the time is

```
Decomposing the domain          7 seconds
```



Building the tree	7 seconds
Tree traversal	33 seconds
Communication during traversal	6 seconds
Force evaluation	54 seconds
Load Imbalance	7 seconds
Total	114 seconds

Thus, the floating point work of force evaluation takes almost half the time, so the implementation is reasonably efficient. At later stages of the calculation, the particles become highly clustered, and the time increases to 160 seconds, but with about the same timing breakdown as above.

7.7 A Chronology of Computer History

In the following, we briefly list chronologically some important historical facts in the development of fast computers and efficient computing. The development in the 20th century is especially impressive.

3000 BC : Dust abacus is invented, probably in Babylonia (Fig. 7.7.22).

1800 BC : Babylonian mathematician develops algorithms to resolve numerical problems.

200 AD : Soroban computing tray used in Japan.

1624 : *Wilhelm Schickard* builds first four-function calculator-clock at the University of Heidelberg.

1642 : *Blaise Pascal* builds the first numerical calculating machine in Paris (Fig. 7.7.22).

1673 : *Gottfried Leibniz* builds a mechanical calculating machine that multiplies, divides, adds and subtracts.

1780 : American *Benjamin Franklin* discovers electricity.

1833 : *Charles Babbage* designs the Analytical Machine that follows instructions from punched-cards. It is the first general purpose computer (Fig. 7.7.23).

1854 : Irishman *George Boole* publishes The Mathematical Analysis of Logic using the binary system.

1876 : Telephone is invented by *Alexander Graham Bell*.

1924 : Computing-Tabulating-Recording Company changes its name to *International Business Machines* (IBM).

1927: First public demonstration of television. Radio-telephone becomes operational between London and New York.

1940: First color TV broadcast.

1941: *Konrad Zuse* builds the Z3 computer, the first calculating machine with automatic control of its operations.

1946: ENIAC (Electronic Numerical Integrator and Computer), with 18,000 vacuum tubes, is introduced (see Figure 7.7.24). It was 2.5 by 30.5 meters and weighted 72500 kilograms. It could do 5,000 additions and 360 multiplications per second.

1946: Term bit for binary digit is used for first time.

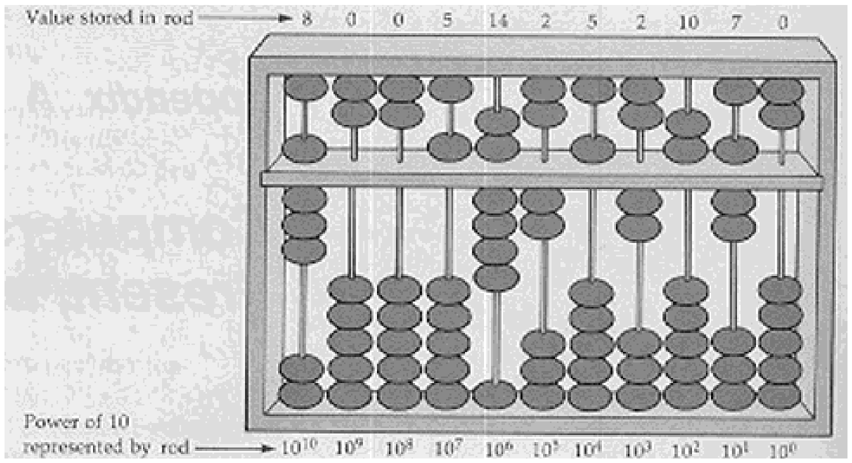
1948: *Transistor* is invented by William Bradford Shockley

1953: IBM ships its first stored-program computer, the 701. It is a vacuum tube, or first generation, computer.

1954: FORTRAN is created by John Backus at IBM. Harlan Herrick runs the first successful FORTRAN program.

1954: *Gene Amdahl* develops the first operating system, used on IBM 704.

1958: First electronic computers are built in Japan by NEC: the NEC-1101 and -1102.



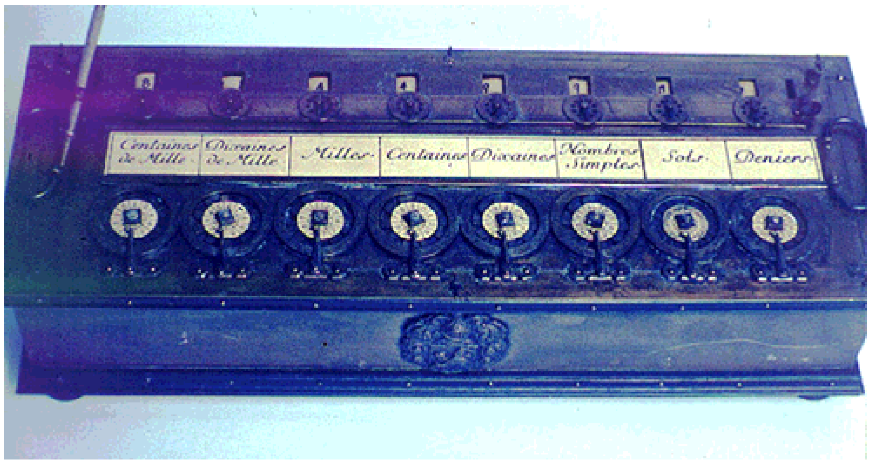


Figure 7.7.22: The abacus and Pascals "computer" (1642)

1958: Seymour Cray builds the first fully transistorized supercomputer, the CDC 1604.

1958: Jack Kilby of Texas Instruments makes the first *integrated circuit* (IC).

1959: IBM ships its first transistorized, or second generation, computers, the 1620 and 1790.

1961: IBM delivers the Stretch computer to Los Alamos. This transistorized computer with 64-bit data paths is the first to use eight-bit bytes; it remains operational until l971.

1964: Control Data Corporation introduces the CDC 6600, the most powerful computer for several years. It was designed by Seymour Cray.

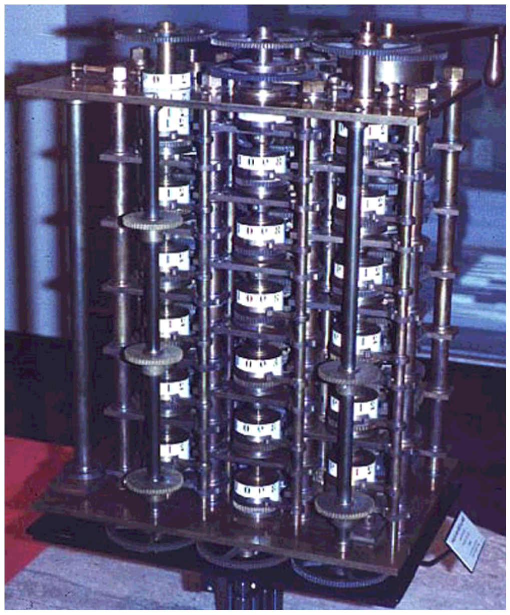


Figure 7.7.23: The Babbage machine (1833)

1965: IBM ships the first System 360, its first integrated circuit-based, or third generation, computer.

1968: Integrated Electronics (Intel) Corp. is founded.

1975: Microsoft is founded after Bill Gates and Paul Allen adapt and sell BASIC to MITS for the Altair PC.

1976: Seymour Cray engineers and delivers the *Cray 1* vectorcomputer (Figure 7.7.24) with 200,000 freon-cooled ICs and 100 million floating point operations per second (MFLOP) performance.

1983: Cray 2 vectorcomputer introduced with one billion Flops performance rating.

1983: NEC announces the SX-1 and SX-2 supercomputers.

1988: Cray Research introduces the Cray Y-MP, a $20M supercomputer.

1989: Intel announces the I860 RISC/coprocessor chip with over one million transistors. and launches a *parallel supercomputer* using over 500 860 RISC microprocessors.

1991: Japan announces plans for sixth-generation computers based on neural networks.

1992: Intel says its next microprocessor will be called Pentium.

1993: IBM presents its SP series, RISC based parallel computers.

1995 $--$ >: Huge parallel machines exist in the USA, the ASCI machines (Intel, IBM), the numerical wind channel in Japan (Fujitsu VPP-based), the earth simulator by NEC (Figure 7.7.25).

1994: Leonard Adleman introduces ideas in which DNA is used as computing medium.

1995$--$ >: Clusters of PCs become interesting alternatives for parallel computing. With fast connection networks between the PCs, they are competitive with classical supercomputers.

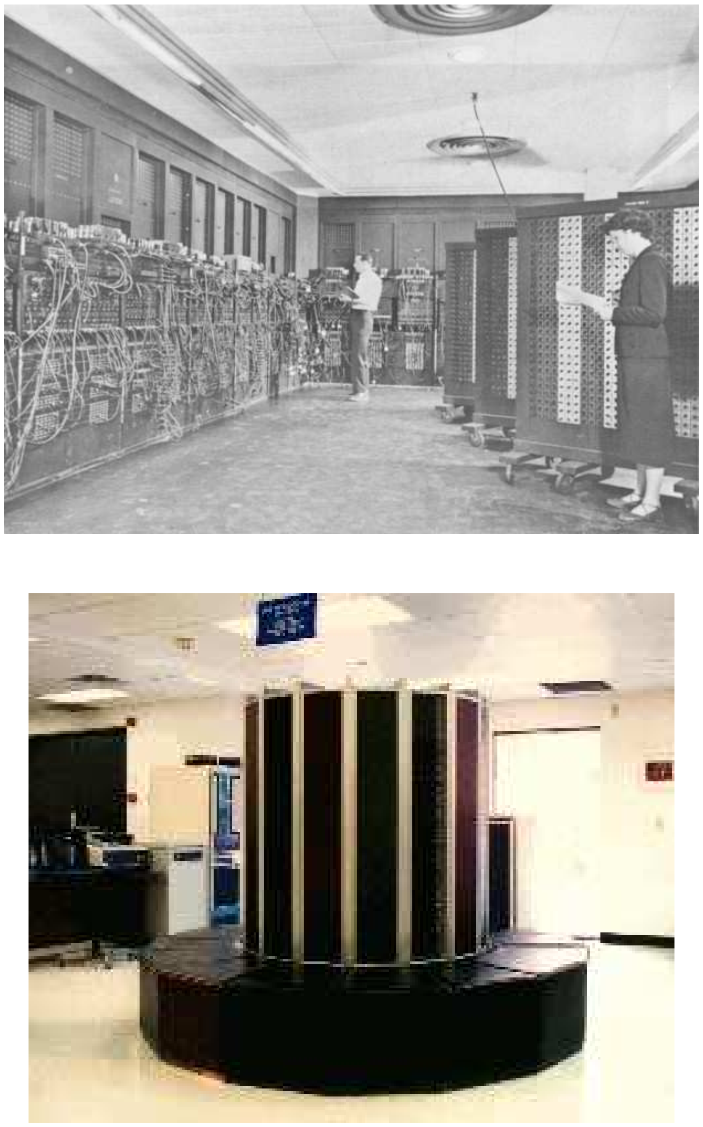


Figure 7.7.24: Two classic computers from the 20th century: The ENIAC computer, and the CRAY1 vector computer.

Figure 7.7.25: Two fast parallel computer systems from the 20th century, with thousands of processors in a parallel machine: The ASCI machines in USA, and NEC's earth simulator in Japan.

A Force Calculation in Molecular Dynamics

A.0.1 Fortran Code

Below we give Fortran code for the force calculation. The box length is denoted by *boxl* and $rcsq = r_c^2$, where r_c is the cut-off distance for the potential, i.e. $\phi(r) = 0$ for $r \geq r_c$. We have also included the calculation of the virial Ψ

$$\Psi = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} r_{ij} \frac{d\phi(r_{ij})}{dr_{ij}}, \tag{1.0.1}$$

which is required for the calculation of the pressure. This quantity is usually calculated along with the forces. The *int*-function is a Fortran defined function, which truncates to an integer.

```
set accumulators to zero
vlj = 0.0
vir = 0.0


do i=1,N
   fx(i) = 0.0
   fy(i) = 0.0
   fz(i) = 0.0
enddo

force calculation

do i= 1, N-1
   do j=i+1, N
      dx = x(i)-x(j)
      dy = y(i)-y(j)
      dz = z(i)-z(j)
implement periodic boundary conditions
      dx = dx - boxl*int(2*dx/boxl)
      dy = dy - boxl*int(2*dy/boxl)
      dz = dz - boxl*int(2*dz/boxl)
      rsq = dx*dx+dy*dy+dz*dz
      if (rsq.lt.rcsq) then
        rrsq = 1.0/rsq
        rr6 = rrsq*rrsq*rrsq
        rr12 = rr6*rr6
        vlj = vlj + rr12 - rr6
        vir = vir + rr12 + rr12 - rr6
        fr = rrsq*(rr12 + rr12 - rr6)
        fx(i) = fx(i) + dx*fr
```

```
            fy(i) = fy(i) + dy*fr
            fz(i) = fz(i) + dz*fr
            fx(j) = fx(j) - dx*fr
            fy(j) = fy(j) - dy*fr
            fz(j) = fz(j) - dz*fr
        endif
    enddo
enddo


vlj = 4.0*vlj
vir = 24.0*vir

do i=1,N
    fx(i) = 24*fx(i)
    fy(i) = 24*fy(i)
    fz(i) = 24*fz(i)
enddo
```

This is by no means the most elegant or efficient piece of code, but it clearly illustrates the structure of the force loop. Note that in this form all possible pairs are considered, so that the total computation time $\propto N^2$, which is not very efficient. In most cases $\phi(r)$ is short-ranged, so that efficient search techniques will improve the performance of the program considerably. Two techniques, which make the time of the computation effectively $\propto N$ are the *linked list* method, originally introduced by Hockney [159], and *neighbour tables*, proposed by Verlet [172].

In the *linked list* technique an *MD* cell, with length l, is divided into M^3 smaller cells of size $l/M \geq r_c$. A particle in a given cell only interacts with particles in the same or neighbouring cells. Therefore, we only need to search the cell a particle resides in and (half of) its neighbouring cells. This can be done very efficiently with a *link list.* In the linked list technique we construct a header array, which points, for each cell, to the first particle in that cell. Then, to each particle we assign a pointer, which points to the next particle in the same cell. The link list ends, if the pointer of a particle is zero. This is implemented below. We assume that the particle coordinates lie between 0 and the box length *boxl*. N is the number of particles and the molecular dynamics cell has been divided into M^3 cells.

```
dimension header(M,M,M), link(N)

do k=1,M
    do l=1,M
        do m=1,M
            header(k,l,m) = 0
        enddo
    enddo
enddo
do $i = 1, N$
    link(i) = 0
enddo
do $i = 1, N$
    ixi = int(M*x(i)/boxl)+1
    iyi = int(M*y(i)/boxl)+1
    izi = int(M*z(i)/boxl)+1
    link(i) = header(ixi,iyi,izi)
    header(ixi,iyi,izi) = i
enddo
```

A particle in a given cell now interacts only with particles in the same cell and its neighbours (26 in total). Because of Newton's third law we only have to scan half of them, each time. The algorithm above creates the list of particles within each cell. An example of the force calculation using the link list is given below. NL is a number, which should be large enough to accommodate all particles in 14 cells (= 13 neighbouring cells and the current cell). The arrays xt, yt and zt store the coordinates in the 14 cells and fxt, fyt and fzt the corresponding forces temporarily.

```
dimension header(M,M,M), link(N)
dimension fx(N), fy(N), fz(N), fxt(NL), fyt(NL), fzt(NL)
dimension xt(NL), yt(NL), zt(NL)
dimension mx(13), my(13), mz(13)
dimension jpart(NL)
data mx/ 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,-1/
data my/ 0, 1, 0, 1, 0, 1, 1, 0,-1, 1, 1,-1, 1/
data mx/ 1, 0, 0, 1, 1, 0,-1,-1, 0, 1,-1, 1, 1/

the arrays mx, my and mz are used to find neighbouring cells

set accumulators to zero
vir=0.0
vpot=0.0
```

```
do i=1,N
   fx(i)=0.0
   fy(i)=0.0
   fz(i)=0.0
enddo
```
loop over cell
```
do ix = 1,M
   do iy = 1,M
      do iz =1,M
         do i=1,NL
            fxt(i)=0.0
            fyt(i)=0.0
            fzt(i)=0.0
         enddo
```
first find particles in cell (ix,iy,iz)
```
         j = header(ix,iy,iz)
         ip = 1
         xt(ip)=x(j)
         yt(ip)=y(j)
         zt(ip)=z(j)
         jpart(ip)=j
         do i=1,NL
            j = link(j)
            if (j.eq.0) goto 100
            ip=ip+1
            xt(ip)=x(j)
            yt(ip)=y(j)
            zt(ip)=z(j)
            jpart(ip)=j
         enddo
100      continue
         npart1=ip
```
find particles in neighbouring cells
```
         do m = 1,13
            ixm = ix+mx(m)
            iym = iy+my(m)
            izm = iz+mz(m)
            if (ixm.eq.M+1) then
            ixm = 1
            else if (ixm.eq.0)
            ixm = M
```

```
              endif
              if (iym.eq.M+1) then
              iym = 1
              else if (iym.eq.0)
              iym = M
              endif
              if (izm.eq.M+1) then
              izm = 1
              else if (izm.eq.0)
              izm = M
              endif
              j=header(ixm,iym,izm)
              ip=ip+1

              xt(ip)=x(j)
              yt(ip)=y(j)
              zt(ip)=z(j)
              jpart(ip)=j
              do i=1,NL
              j = link(j)
              if (j.eq.0) goto 101
              ip=ip+1
              xt(ip)=x(j)
              yt(ip)=y(j)
              zt(ip)=z(j)
              jpart(ip)=j
              enddo
101           continue
          enddo
          npart2=ip
calculate forces between particles
          do i= 1,npart1
              do j=i+1,npart2
                  dx=xt(i)-xt(j)
                  dy=yt(i)-yt(j)
                  dz=zt(i)-zt(j)
                  dx=dx-boxl*int(2*dx/boxl)
                  dy=dy-boxl*int(2*dy/boxl)
                  dz=dz-boxl*int(2*dz/boxl)
                  rsq=dx*dx+dy*dy+dz*dz
                  if (rsq.lt.rcsq) then
                  rrsq = 1.0/rsq
                  rr6 = rrsq*rrsq*rrsq
```

```
                rr12 = rr6*rr6
                vpot = vpot + rr12 - rr6
                fr = rr12 + rr12 - rr6
                vir = vir + fr
                fr = rrsq*fr
                fxt(i) = fxt(i) + dx*fr
                fyt(i) = fyt(i) + dy*fr
                fzt(i) = fzt(i) + dz*fr
                fxt(j) = fxt(j) - dx*fr
                fyt(j) = fyt(j) - dy*fr
                fzt(j) = fzt(j) - dz*fr
                endif
             enddo
          enddo
          do j = 1,npart2
             i = jpart(j)
             fx(i) = fx(i) + fxt(j)
             fy(i) = fy(i) + fyt(j)
             fz(i) = fz(i) + fzt(j)
          enddo
       enddo
   enddo
enddo


vir = 24.0* vir
vpot = 4.0* vpot


do i = 1,N
   fx(i) = 24.0*fx(i)
   fy(i) = 24.0*fy(i)
   fz(i) = 24.0*fz(i)
enddo
```

The *neighbour table* of a particle is a list of particles, which at some time t reside within a radius $r_t \geq r_c$ of that particle. By going through the *neighbour table* of each particle we find all the other particles, with which the particle interacts. Depending on the size of r_t and the diffusion rate of the particles one needs to update the table every so often. The update is most efficiently carried out with the linked list method. We refer to the literature [151] for a fuller discussion of the optimal choice of the algorithm and the optimal value of r_t.

References

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D.A. Yeung. "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessors", *Proc. Workshop Multi-threaded Computers, Supercomputing 91*, 1991.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] A.V. Aho, J.D. Ullman. *Foundations of Computer Science*, Computer Science Press, New York, 1992.

[4] S.G. Akl. *The Design and Analysis of Parallel Algorithms*, Prentice-Hall Int., London, 1989.

[5] R.E. Alcouffe, A. Brandt, J.E. Dendy and J.W. Painter, The multi-grid method for the diffusion equation with strongly discontinuous coefficients. *SIAM J. Sci. Comput.* 2: 430-454, 1981.

[6] G. Almasi and A. Gottlieb. *Highly Parallel Computing*, 2nd edition, Benjamin/Cummings, 1994.

[7] O. Axelsson and V.A. Barker, *Finite element solution of boundary value problems.* Academic Press, Orlando, 1984.

[8] O. Axelsson and G. Lindskog. On the eigenvalue distribution of a class of preconditioning methods. *Numer. Math.*, 48:479–498, 1986.

[9] I. Babuska, The p and $h-p$ versions of the finite element method: The state of the art. *In:* D.L Dwoyer et al. (eds.), Finite elements, theory and applications, Springer New York, 1988.

[10] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA J. Num. Anal.*, 14:563–581, 1994.

[11] N.S. Bakhvalov, On the convergence of a relaxation method with natural constraints on the elliptic operator. *USSR Comp. Math. and Math. Phys.* 6: 101-135, 1966.

[12] R.E. Bank, PLTMG: A Software Package for Solving Elliptic Partial Differential Equations. User's Guide 7.0, *Frontiers Appl. Math.* 15, SIAM Philadelphia, 1994.

[13] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[14] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, 1989.

[15] A. Björck and T. Elfving. Accelerated projection methods for computing pseudo-inverse solution of systems of linear equations. *BIT*, 19:145–163, 1979.

[16] D. Braess, *Finite Elements. Theory, fast solvers and applications in solid mechanics.* Cambridge University Press, Cambridge, 1997.

[17] D. Braess and W. Hackbusch, A new convergence proof for the multigrid method including the V-Cycle. *SIAM J. Numer. Anal.* 20: 967–975, 1983.

[18] E. Brakkee, A. Segal, and C.G.M. Kassels. A parallel domain decomposition algorithm for the incompressible Navier-Stokes equations. *Simulation Practice and Theory*, 3:185–205, 1995.

[19] E. Brakkee, C. Vuik, and P. Wesseling. Domain decomposition for the incompressible Navier-Stokes equations: solving subdomain problems accurately and inaccurately. *Int. J. Num. Meth. in Fluids*, 26:1217–1237, 1998.

[20] J.H. Bramble, Multigrid Methods. *Pitman Res. Notes Math. Ser.* 294, Longman Sci.& Tech., Harlow, 1993.

[21] A. Brandt, Multi-level adaptive solutions to boundary-value problems. *Math. Comput.* 31: 333–390, 1977.

[22] A. Brandt, Multigrid solvers for non-elliptic and singular-perturbation steady-state problems. Weizmann Institute of Science, Rehovot, December 1981.

[23] A. Brandt, Guide to multigrid development. In: [67], 220–312.

[24] A. Brandt, *Multigrid techniques: 1984 guide with applications to fluid dynamics.* GMD-Studie Nr. 85, Sankt Augustin, Germany, 1984.

[25] A. Brandt and V. Mikulinsky, Recombining iterants in multigrid algorithms and problems with small islands. *SIAM J. Sci. Comput.* 16: 20–28, 1995.

[26] W.L. Briggs, *A multigrid tutorial*, SIAM, Philadelphia (1987).

[27] A.M. Bruaset. *A Survey of Preconditioned Iterative Methods.* Pitman research notes in mathematics series 328. Longman Scientific and Technical, Harlow, 1995.

[28] H. Bulgak and C. Zenger, *Error control and Adaptivity in Scientific Computing*, NATO Science Series, Series C: Math. and Phys. Sciences, Vol. 536, Kluwer Acad. Publ., 1999.

[29] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 21:792–797, 1999.

[30] R. Courant, K.O. Friedrichs and H. Lewy, Über die partiellen Differenzengleichungen der mathematischen Physik. *Math. Annal.* 100: 32–74, 1928.

[31] I. Danesh. "Physical limitations of a computer", *Computer Architecture News* 21: 40-45, 1993.

[32] L. Dekker. "Delft Parallel Processor: a missing link to the future of simulation", Proc. of SIMS'82 Annual Meeting, University of Trondheim, Norway, May, 1982.

[33] L. Dekker. "Expandability of an MIMD multiprocessor system to a large size". *Proc. SCS Eastern Simulation Conf.*, 157-162, 1985.

[34] J.W. Demmel, M.T. Heath, and H.A. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica*, pages 111–197. Cambridge University Press, Cambridge, UK, 1993.

[35] R.B.K. Dewar and M. Smosna. *Microprocessors: A Programmers View*, McGraw-Hill, New York, 1990.

[36] S. Doi, *Applied Numerical Mathematics*, 7: 417, 1991.

[37] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, 1991.

[38] M. Dryja and O. B. Widlund, Domain decomposition algorithms with small overlap. *SIAM J. Sci. Comput.* 15: 604–620, 1994.

[39] I.S. Duff and G.A. Meurant, *BIT*, 29: 635, 1989.

[40] S.C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[41] S.C. Eisenstat, H.C. Elman, and M.H. Schultz. Variable iterative methods for nonsymmetric systems of linear equations. *SIAM J. Num. Anal.*, 20:345–357, 1983.

[42] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Num. Anal.*, 21:356–362, 1984.

[43] R.P. Fedorenko, A relaxation method for solving elliptic difference equations. *USSR comp. Math. and Math. Phys.* 1(5): 1092–1096, 1962.

[44] R.P. Fedorenko, The speed of convergence of one iterative process. *USSR comp. Math. and Math. Phys.* 4(3): 227–235, 1964.

[45] R. Fletcher. Factorizing symmetric indefinite matrices. *Lin. Alg. and its Appl.*, 14:257–277, 1976.

[46] M.J. Flynn. "Some computer organizations and their effectiveness", *IEEE trans. on Computers* 21, No.9, 1972.

[47] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. MAS-R 0009, CWI, Amsterdam, 2000.

[48] R.W. Freund, G.H. Golub, and N.M. Nachtigal. Iterative solution of linear systems. In A. Iserles, editor, *Acta Numerica*, pages 57–100. Cambridge University Press, Cambridge, UK, 1992.

[49] R.W. Freund, M.H. Gutknecht, and N.M. Nachtigal. An implimentation of the look-ahead Lanczos algorithm for non-Hermitian matrices. *SIAM J. Sci. Comp.*, 14:137–156, 1993.

[50] R.W. Freund and N.M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[51] D.D. Gajski, D.J. Kuck, D.H. Lawrie and A.H. Sameh. "Cedar—A large scale multiprocessor", *Proc. 1983 Int. Conf. Parallel Processing*, 524-529, 1983.

[52] D.D. Gajski, and J.K. Peir. "Essential issues in multiprocessor systems", *IEEE Computer* 18(6): 9-27, 1985.

[53] P.R. Garabedian, *Partial differential equations.* Wiley, New York, 1964.

[54] C.W. Gear, *Numerical initial value problems for ordinary differential equations.* Prentice Hall, Englewood Cliffs, 1971.

[55] A. Gibbons, and W. Rytter. *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

[56] T. Ginsburg. The conjugate gradient method. In J.H. Wilkinson and C. Reinsch, editors, *Handbook for Automatic Computation, 2, Linear Algebra*, pages 57–69, Berlin, 1971. Springer.

[57] G.H. Golub and C.F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, 1996. Third edition.

[58] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir. "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer", *IEEE Trans. on Computers*, Vol. C-32, No. 2, Feb. 1983.

[59] "Grand Challenges: High Performance Computing and Communications", Report by the Committee on Physical, Mathematical, and Engineering Sciences, Office of Science and Technology Policy, U.S.A., 1991.

[60] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, SIAM, Philadelphia, 1997.

[61] M. Griebel, Multilevel algorithms considered as iterative methods on semidefinite systems. *SIAM J. Sci. Comput.* 15: 547–565, 1994.

[62] A. Gupta, and V. Kumar. "The scalability of FFT on parallel computers", *IEEE Trans. Parallel and Distributed Systems* 4(7), July, 1993.

[63] G.R. Gurd C.C. Kirkham and I. Watson. "The Manchester Data Flow Computer", *Comm. ACM*, 28(1): 34-52, 1985.

[64] I.A. Gustafsson. A class of first order factorization methods. *BIT*, 18:142–156, 1978.

[65] W. Hackbusch, On the convergence of a multi-grid iteration applied to finite element equations. Report 77-8, Institute for Applied Mathematics, University of Cologne, 1977.

[66] W. Hackbusch, Convergence of multi-grid iterations applied to difference equations. *Math. Comp.* 34: 425–440, 1980.

[67] W. Hackbusch and U. Trottenberg (eds.), *Multigrid methods*, Lecture Notes in Mathematics 960, Springer, Berlin, 1982.

[68] W. Hackbusch, *Multi-grid methods and applications.* Springer, Berlin, 1985.

[69] W. Hackbusch and U. Trottenberg (eds.), *Multigrid Methods II*, Lecture Notes in Mathematics 1228, Springer, Berlin 1986.

[70] W. Hackbusch and U. Trottenberg (eds.), *Multigrid Methods III*, Proc. 3rd Int. Conf. on Multigrid Methods. Int. Series of Num. Math. 98, Birkhäuser, Basel 1991.

[71] W. Hackbusch, *Theory and numerical treatment of elliptic differential equations*. Springer New York, 1992.

[72] W. Hackbusch, *Iterative solution of large sparse systems of equations*. Springer New York, 1994.

[73] W. Hackbusch, G. Wittum (eds.) *Multigrid Methods V*, Proc. 5th European Multigrid Conf. in Stuttgart Germany, Lecture Notes in Computational Science and Engineering 3, Springer Berlin, 1998.

[74] P.W. Hemker and P. Wesseling (eds.), *Multigrid Methods IV.* Proc. of the Fourth European Multigrid Conference, Birkhäuser Basel, 1994.

[75] R. Hempel and A. Schüller, *Experiments with parallel multigrid algorithms using the SUPRENUM communications subroutine library.* GMD Arbeitspapier 141, GMD, St. Augustin, Germany, 1988.

[76] M.R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bur. Stand. Sect. B* 49: 409–436, 1952.

[77] R.W. Hockney and C.R. Jesshope. *Parallel computers: architecture, programming and algorithms*, 1st edition, Adam Hilger Ltd., Bristol, 1981.

[78] R.W. Hockney and C.R. Jesshope. *Parallel computers 2: architecture, programming and algorithms*, 2nd edition, Adam Hilger Ltd., Bristol, 1988.

[79] H. Hoppe and H. Mühlenbein, Parallel adaptive full-multigrid methods on message-based multiprocessors. *Parallel Comput.* 3: 269–287, 1986.

[80] T.C. Hu. "Parallel Sequencing and Assembly Line Problems", *Operations Research* 9: 841-848, 1961.

[81] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., 1993.

[82] A. Jameson, W. Schmidt and E. Turkel, Numerical simulation of the Euler equations by finite volume methods using Runge -Kutta time stepping schemes. *AIAA Paper* 81-1259, 1981.

[83] A. Jameson, *Transonic flow calculations for aircraft*, *In:* F. Brezzi (ed.), Num. Methods in Fluid Mech., Lecture Notes in Math. 1127: 156–242, Springer, Berlin, 1985.

[84] A. Jameson, Analysis and design of numerical schemes for gas dynamics, 1: artificial diffusion, upwind biasing, limiters and their effect on accuracy and multigrid convergence, *Int. J. Comp. Fluid Dyn.* 4: 171–218, 1995.

[85] E.F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *J. of Comp. Appl. Math.*, 24:265–275, 1988.

[86] A.H. Karp. "Programming for parallelism", *IEEE Computer* 20: 43-57, 1987.

[87] R. Kettler, Analysis and comparison of relaxation schemes in robust multigrid and preconditioned conjugate gradient methods. *In:* [67], 502–534.

[88] A. Klawonn and O. B. Widlund, FETI and Neumann-Neumann Iterative Substructuring Methods: Connections and New Results. Techn. Rep. 796, Comp. Sci. Dep., Courant Inst. Math. Sciences, 1999.

[89] B. Koren, Defect correction and multigrid for an efficient and accurate computation of airfoil flows. *J. Comp. Phys.* 183: 193–206, 1988.

[90] A. Krechel, H-J. Plum and K. Stüben, Parallelization and vectorization aspects of the solution of tridiagonal linear systems. *Parallel Comput.* 14: 31–49, 1990.

[91] V. Kumar, and V.N. Rao. "Parallel depth-first search, Part II: Analysis", *Int. J. of Parallel Programming* 16(6): 501-519, 1987.

[92] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing; design and analysis of algorithms.* Benjamin/Cummings, Redwood City, 1994.

[93] C.E. Leiserson. "Fat-trees: universal networks for hardware-efficient supercomputing", *IEEE Trans. Computers* 34: 892-901, 1985.

[94] K. Li and R. Schaefer. "A hypercube shared virtual memory system", *Proc. of 1989 International Conference on Parallel Processing*, vol. I, 125-132, 1989.

[95] J. Linden, G. Lonsdale, H. Ritzdorf and A. Schüller, Scalability aspects of parallel multigrid. *Fut. Generation Comp. Systems* 10: 429–439, 1994.

[96] J. Linden, B. Steckel and K. Stüben, Parallel multigrid solution of the Navier-Stokes equations on general 2D-domains. *Parallel Comput.* 7: 461–475, 1988.

[97] J. Mandel and R. Tezaur, Convergence of a substructuring method with Lagrange multipliers. *Numer. Math.* 73: 473–487, 1996.

[98] D.J. Mavripilis, Multigrid strategies for viscous flow solvers on anisotropic unstructured meshes. *J. Comp. Phys.* 145: 141–165, 1998.

[99] O.A. McBryan, P.O. Frederickson, J. Linden, A. Schüller, K. Solchenbach, K. Stüben, C.A. Thole and U. Trottenberg, Multigrid methods on parallel computers – a survey of recent developments. *Impact Comput. Sci. Engrg.* 3: 1–75, 1991.

[100] S.F. Mc.Cormick, *Multilevel adaptive methods for partial differential equations,* Frontiers in Appl. Math. 6, SIAM Philadelphia, 1989.

[101] J.A. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[102] N.M. Nachtigal, S.C. Reddy, and L.N. Trefethen. How fast are non symmetric matrix iterations. *SIAM J. Matrix Anal. Appl.*, 13:778–795, 1992.

[103] C.W. Oosterlee and T. Washio, An evaluation of parallel multigrid as a solver and as a preconditioner for singularly perturbed problems. *SIAM J. Sci. Comput.* 19: 87–110, 1998.

[104] C.C. Paige and M.A. Saunders. Solution of sparse indefinite system of linear equations. *SIAM J. Num. Anal.*, 12:617–629, 1975.

[105] C.C. Paige and M.A. Saunders. LSQR: an algorithm for sparse linear equations and sparse least square problem. *ACM Trans. Math. Softw.*, 8:44–71, 1982.

[106] B.N. Parlett, D.R. Taylor, and Z.A. Liu. A look-ahead Lanczos algorithm for unsymmetric matrices. *Math. Comp.*, 44:105–124, 1985.

[107] "Technical Summary Parsytec GC", Parsytec Computer GmbH, 1991.

[108] F. Peper. "Efficient network topologies for extensible massively parallel computers", Ph.D. thesis, Delft University of Technology, 1989.

[109] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton and J. Weiss. "The IBM Research Parallel Processor Prototype (RP3); introduction and architecture", *Proc. 1985 Int. Conf. Parallel Processing*, Chicago, 764-771, 1985.

[110] A. Quarteroni and A. Valli, *Domain Decomposition Methods for Partial Differential Equations.* Oxford Science Publi., 1999.

[111] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

[112] G. Radicati di Brozolo and Y. Robert. Vector and parallel CG-like algorithms for sparse non-symmetric systems. Technical Report 681-M, IMAG/TIM3, Grenoble, 1987.

[113] J.K. Reid. The use of conjugate for systems of linear equations posessing property A. *SIAM J. Num. Anal.*, 9:325–332, 1972.

[114] L.F. Richardson. *Weather prediction by numerical process*, Cambridge University Press, 1922.

[115] H. Ritzdorf and K. Stüben, *Adaptive multigrid on distributed memory computers. In:* [74], 77–96.

[116] D. Rixen and C. Farhat, A simple and efficient extension of a class of substructure based preconditioners to heterogeneous structural mechanics problems. *Int. J. Numer. Meth. Engng.*, 44:489–516, 1999.

[117] J. Rothnie. "Overview of the KSR1 Computer System", Kendall Square Research Report TR 9202001, March, 1992

[118] U. Rüde, Mathematical and computational techniques for multilevel adaptive methods. *Frontiers in Applied Mathematics*, Vol. 13, SIAM, Philadelphia, 1993.

[119] J.W. Ruge and K. Stüben, Algebraic Multigrid (AMG). *In:* S.F. McCormick (ed.), Multigrid Methods, Frontiers in Appl. Math., SIAM Philadelphia, 5: 73–130, 1987.

[120] Y. Saad. Preconditioning techniques for non symmetric and indefinite linear system. *J. Comp. Appl. Math.*, 24:89–105, 1988.

[121] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Stat. Comput.*, 14:461–469, 1993.

[122] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, Boston, 1996.

[123] Y. Saad and M.H. Schultz. GMRES: a generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[124] R. Schreiber, and H.D. Simon. Towards the Teraflops Capability for CFD. *Parallel Computational Fluid Dynamics; Implementations and Applications*, H.D. Simon (eds.), MIT Press, 1992.

[125] M.K. Seager. Parallelizing conjugate gradient for the Cray X_MP. *Parallel Computing*, 3:35–47, 1986.

[126] D.B. Skillicorn. "A taxonomy for computer architectures", *IEEE Computer* 21: 46-57, 1988.

[127] A. van der Sluis. Conditioning, equilibration, and pivoting in linear algebraic systems. *Numer. Math.*, 15:74–86, 1970.

[128] A. van der Sluis and H.A. van der Vorst. The rate of convergence of conjugate gradients. *Numer. Math.*, 48:543–560, 1986.

[129] B. Smith, P. Bjorstad and W. Gropp, *Domain decomposition - Parallel multilevel methods for elliptic partial differential equations*. Cambridge Univ. Press, Cambridge, 1996.

[130] P. Sonneveld, P. Wesseling and P.M. de Zeeuw, Multigrid and conjugate gradient methods as convergence acceleration techniques. *In:* D.J. Paddon and H. Holstein (eds.), Multigrid Methods for Integral and Differential Equations, Clarendon Press, 1985.

[131] P. Sonneveld. CGS: a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[132] P. Stenstrom, T. Joe, and A. Gupta. "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures", *Proceedings 19th International Symposium on Computer Architecture*, 1992.

[133] K. Stüben and U. Trottenberg, *Multigrid methods: fundamental algorithms, model problem analysis and applications. In:* [67], 1–176.

[134] E. de Sturler and H.A. van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Appl. Num. Math.*, 18:441–459, 1995.

[135] "The Connection Machine CM-5 Technical Summary", Thinking Machines Corporation, 1991.

[136] J.F. Thompson, Z.U.A. Warsi and C.W. Mastin, *Numerical grid generation.* North Holland, Amsterdam, 1985.

[137] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. ”Data-driven and demand-driven computer architecture”, *Computing surveys* 14: 93-143, 1982.

[138] H.A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comp.*, 10:1174–1185, 1989.

[139] H.A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for solution of non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

[140] H.A. van der Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *J. Comput. Appl. Math.*, 48:327–341, 1993.

[141] H.A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. *Num. Lin. Alg. Appl.*, 1:369–386, 1994.

[142] R.S. Varga, *Matrix iterative analysis*, Prentice-Hall, Englewood Cliffs, 1962.

[143] C. Vuik. Solution of the discretized incompressible Navier-Stokes equations with the GMRES method. *Int. J. Num. Meth. in Fluids*, 16:507–523, 1993.

[144] C. Vuik, A. Segal, and J.A. Meijerink. An efficient preconditioned CG method for the solution of a class of layered problems with extreme contrasts in the coefficients. *J. Comp. Phys.*, 152:385–403, 1999.

[145] C. Vuik, R.R.P. van Nooyen, and P. Wesseling. Parallelism in ILU-preconditioned GMRES. *Paral. Comp.*, 24:1927–1946, 1998.

[146] P. Wesseling, *An introduction to multigrid methods.* John Wiley, Chichester, 1992.

[147] O. Widlund, DD methods for elliptic partial differential equations, In [28], 325–354.

[148] J. Xu, Iterative methods by space decomposition and subspace correction. *SIAM Review* 34: 581–613, 1992.

[149] D. Young, *Iterative solution of large linear systems.* Academic Press, New York, 1971.

Publications: Particle Simulations

[150] B.J. Alder and T. E. Wrainwright, Decay of the velocity autocorrelation function, *Phys. Rev.* A1: 18-21, 1970.

[151] M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids*, (Clarendon, Oxford, 1987.

[152] J. Barnes and P. Hut, A Hierarchical O(n log(n)) force calculation algorithm, *Nature* 324: 446-449, 1986.

[153] G. Ciccotti and J.P. Ryckaert, Molecular dynamics simulation of rigid molecules, *Computer Physics Rep.* 4: 345-392, 1985.

[154] M.J. Ernst, E.H. Hauge and J.M.J. van Leeuwen, Asymptotic time behaviour of correlation functions. I. Kinetic terms, *Phys. Rev.* A4: 2055-2065, 1971.

[155] D.J. Evans and S. Murad A singularity-free algorithm for molecular dynamics simulation of rigid polyatomics, *Molec. Phys.* 34: 327-331, 1977.

[156] P.P. Ewald, Die Berechnung optischer und electrostatischer Gitterpotentiale, *Ann. Phys* 64: 253-287, 1921.

[157] H. Goldstein, *Classical Mechanics*, 2nd ed. Addison-Wesley, Reading, Ma. 1980.

[158] L. Greengard and V. Rokhlin, A Fast Algorithm for Particle Simulations, *J. Comp. Phys.* 73: 282-292, 1987.

[159] R.W. Hockney and J.W. Eastwood, *Computer simulations using particles*, MacGraw-Hill, New York, 1981.

[160] W.G. Hoover, Canonical dynamics: equilibrium phase space distributions, *Phys. Rev.* A31: 1695-1703, 1985.

[161] K. Hwang and Z. Xu, *Scalable Parallel Computing*, McGraw-Hill, Boston, 1998.

[162] H. Jeffreys and B. Jeffreys, *Methods of Mathematical Physics*, Cambridge University Press, Cambridge, page 361, 1980.

[163] L.D. Landau and E.M. Lifshitz, *A course of theoretical physics vol. 5: Statistical Physics*, Pergamon Press, Oxford, Chapter 12, 1970.

[164] J.L. Lebowitz, J.K. Percus and L. Verlet, Ensemble dependence of fluctuations with application to machine calculations, *Phys. Rev.* 253: 250-254, 1967.

[165] L. Onsager, Reciprocal relations in irreversible processes. I., *Phys. Rev.* 37: 405-417, 1931.

[166] L. Onsager, Reciprocal relations in irreversible processes. II., *Phys. Rev.* 38: 2265-2278, 1931.

[167] J.W. Perram and S.W. de Leeuw, Statistical Mechanics of two-dimensional Coulomb systems. I. Lattice sums and simulation methodology, *Physica* 109A: 237-250, 1980.

[168] H.G. Petersen and H. Flyvbjerg, Error estimates in molecular dynamics simulations, *J. Chem. Phys.* 91: 461-467, 1989.

[169] J.P. Ryckaert and G. Ciccotti and H.J.C. Berendsen, Numerical integration of the cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes, *J. Computational Physics* 23: 327-341, 1977.

[170] W.C. Swope, H.C. Andersen, P.H. Berens and K.R. Wilson, A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: application to small water clusters, *J. Chem. Phys.* 76: 637-649, 1982.

[171] M. Tuckerman, B.J. Berne and G.J. Martine, Reversible multiple time scale molecular dynamics, *J. Chem. Phys.* 97: 1990-2001, 1992.

[172] L. Verlet, Computer 'experiments' on classical liquids. I. Thermodynamical properties of Lennard-Jones molecules, *Phys. Rev.* 159: 98-103, 1967.

[173] M. Warren and J. Salmon, "Astrophysical N-body Simulations using Hierarchical Tree Data Structures", in *Proceedings of Supercomputing 92*, 1992.

[174] M. Warren and J. Salmon, "A parallel hashed octtree N-body algorithm", in *Proceedings of Supercomputing 93*.

[175] An online bibliography of papers on parallel Barnes-Hut can be found in http://www.ccsf.caltech.edu/ johns/papers.html

[176] E.T. Whittaker and G.N. Watson, *A course in modern analysis*, Cambridge University Press, 4th edition, Chapter 21, 1978.