E6 – Software Development / Numerical Programming II

Olaf Ippisch
email: Olaf.Ippisch@ipvs.uni-stuttgart.de

30. Juli 2009

Inhaltsverzeichnis

1 Introduction **4**
1.1 Subject of the Lecture 4
1.2 Concrete Example 4
1.3 Topics the Lecture and the Exercises 7

2 Groundwater Flow **8**

3 Partial Differential Equations **13**
3.1 Examples for PDE types 15
3.2 Sphere of Influence 17

4 Spatial-Discretisation Methods **18**
4.1 Recapitulation: The Finite-Difference Method 18
4.2 Recapitulation: The Finite-Element Method 19
4.3 The Finite-Volume Method 19
4.4 The Vertex-Centered Finite-Volume Method 26

5 Solution of Linear Equation Systems **28**
5.1 Direct Solution of Sparse Linear Equation Systems 28
5.2 Iterative Solution of Sparse Linear Equation Systems 29
5.2.1 Relaxation Methods 29

6 Parallel Computing **51**
6.1 Introduction 51
6.1.1 Why Parallel Computing ? 52
6.1.2 (Very Short) History of Supercomputers 52
6.2 Single Processor Architecture 55
6.2.1 Von Neumann Architecture 55
6.2.2 Pipelining 56
6.2.3 Superscalar Architecture 57
6.2.4 Caches 58

6.3 Parallel Architectures . . . 61
6.3.1 Classifications . . . 61
6.3.2 Uniform Memory Access Architecture . . . 61
6.3.3 Nonuniform Memory Access Architecture . . . 64
6.4 Things to Remember . . . 66
6.4.1 Private Memory Architecture . . . 66
6.4.2 Things to Remember . . . 69
6.5 Process Model . . . 70
6.5.1 A Simple Notation for Parallel Programs . . . 70
6.5.2 The Critical Section Problem . . . 71
6.5.3 Single Program Multiple Data . . . 72
6.5.4 Condition Synchronisation . . . 73
6.5.5 Things to Remember . . . 75
6.6 OpenMP . . . 75

7 Basics of Parallel Algorithms **79**
7.1 Data Decomposition . . . 79
7.2 Agglomeration . . . 82
7.3 Mapping of Processes to Processors . . . 83
7.4 Load Balancing . . . 83
7.5 Data Decomposition of Vectors and Matrices . . . 84
7.6 Matrix-Vector Multiplication . . . 86

8 Introduction Message Passing **88**
8.1 Synchronous Communication . . . 90
8.2 Asynchronous Communication . . . 91

9 The Message Passing Interface **92**
9.1 Simple Example . . . 93
9.2 Communicators and Topologies . . . 95
9.3 Blocking Communication . . . 96
9.4 Non-blocking communication . . . 98
9.5 Global Communication . . . 99
9.6 Avoiding Deadlocks: Coloring . . . 100

10 Things to Remember **102**

11 Analysis of Parallel Algorithms **103**
11.1 Examples . . . 105
11.1.1 Scalar Product . . . 105
11.1.2 Gaussian Elimination . . . 106
11.2 Scalability . . . 107
11.2.1 Fixed Size . . . 107
11.2.2 Scaled Size . . . 108
11.3 Things to Remember . . . 110

12 Parallel Iterative Solution of Sparse Linear Equation Systems **110**
12.1 Parallelization . . . 111

12.2 MPI Functions for Cartesian Grids 115
12.2.1 Examples 117

13 Debugging of Parallel Programs **119**

14 Time-dependent Problems **120**
14.1 Parabolic Problems 120

1 Introduction

1.1 Subject of the Lecture

Intention of the Lecture

- Other lectures cover theoretical aspects of modeling and simulation (equations, material properties, mathematical aspects of partial differential equations, numerical methods)
- In this lecture we will apply this knowledge to develop a real working model for the solution of a concrete example problem from scratch.
- To realise this we will need some on the iterative solution of linear equation systems, discretisations and parallel programming

Aims

- Get an insight in the operation of a simulation programs
- Get a better understanding for the behaviour of existing solvers for partial differential equations
- Learn modern programming techniques
- Introduction to parallel computing

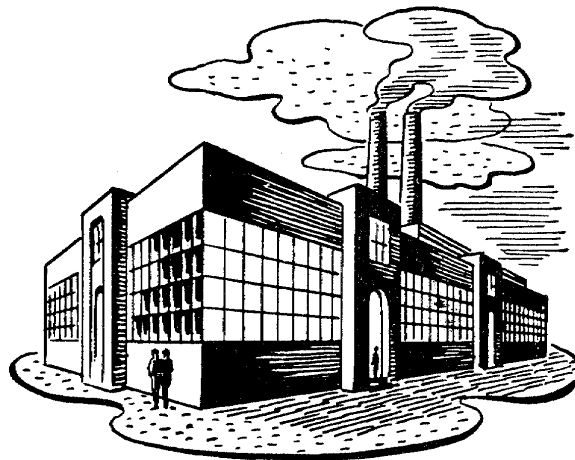
Prerequisites

- Basic knowledge of object-oriented programming techniques (Commas C6 lecture)
- Basic knowledge of numerical mathematics
- Basic knowledge about partial differential equations (Commas C5 lecture)
- Readiness to do some programming in the exercises

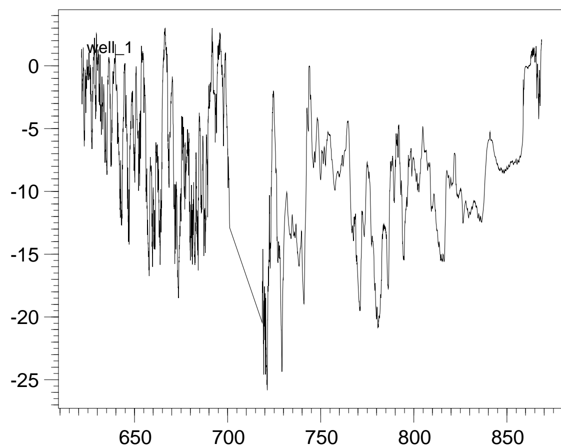
1.2 Concrete Example

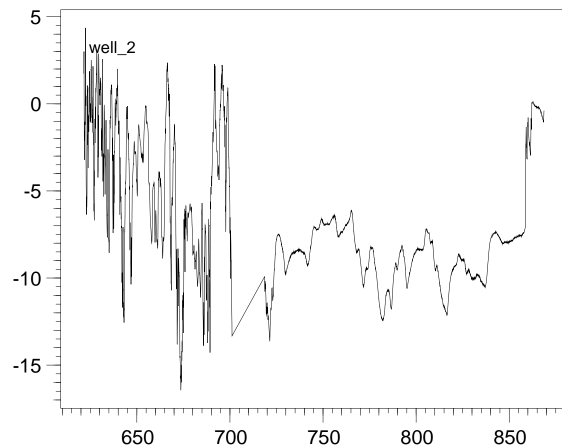
Groundwater contamination problem

The water in several wells is contaminated with a soluble substance.



We know that there was an accident in a factory where the same substance was released to the groundwater.





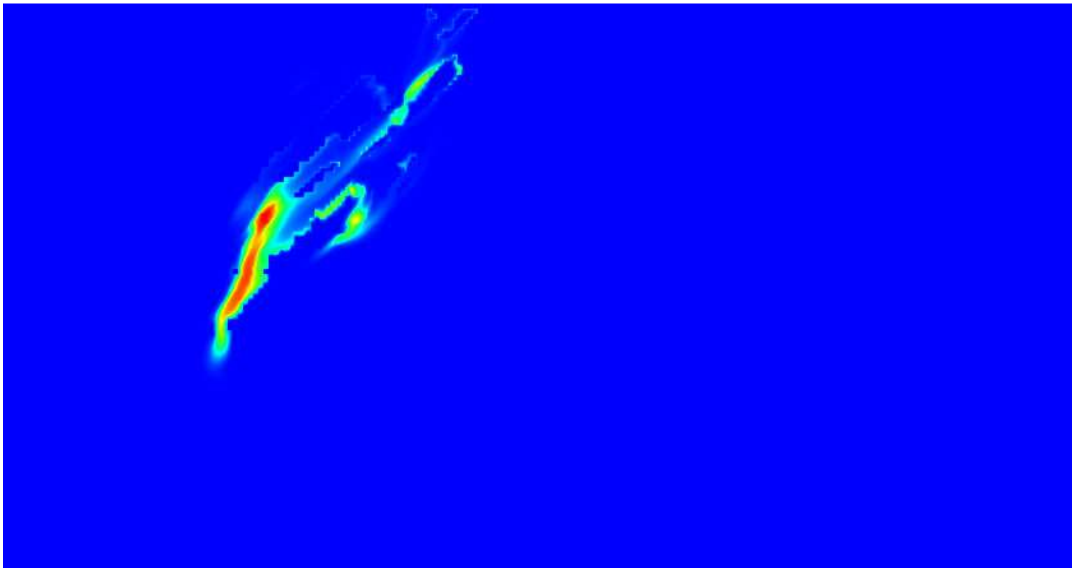
- Does this explain all the contamination?
- Can we reproduce the measurements?
- Is there another source involved?

Why this problem?

The example has nothing to do with structural mechanics, but:

- While problems in structural mechanics usually lead to systems of (nonlinear) partial differential equations, the groundwater flow equation is a linear partial differential equation in one variable only
- Transport in porous media is important for engineers as well(e.g. water (vapour) flow in walls)

- From the numerics and software development point of view the application is not important, only the type of the partial differential equation
- The groundwater flow equation is a typical elliptical, the solute transport equation a typical hyperbolic partial differential equation
- Can be solved in the available time
- We can produce nice pictures ;-)



What do we have to do to solve this problem?

- Compute the flow field for groundwater
- Determine the amount of contamination from the factory
- Solve solute transport
- Compare measurements at wells with the result

How do we do this in this lecture?

- We develop a new groundwater flow model
- Parallelise the groundwater flow model
- Use an existing solute transport solver to solve the solute transport
- Compare measurements at wells with the result

1.3 Topics the Lecture and the Exercises

Topics covered by the Lecture

- Introduction to the problem
- Flow in porous media
- The Finite-Volume-Method
- Iterative linear solvers
  - Basics
  - Advanced methods
  - Multigrid
- Grids and grid generation
- Parallel Computers
  - Basics
  - MPI
  - Parallel iterative solvers
- Time dependend problems, Solute Transport

Exercises

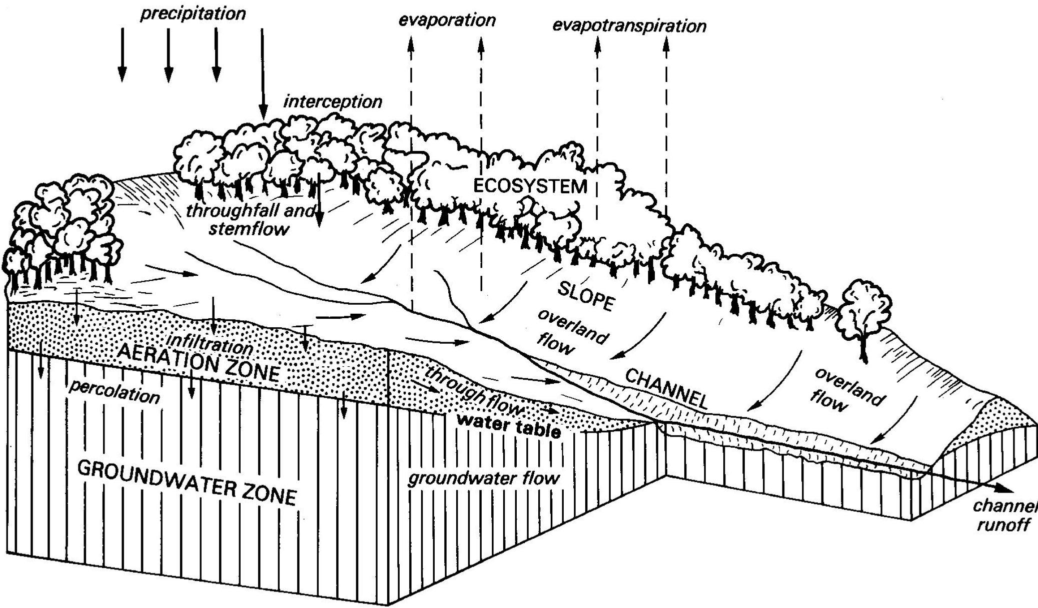
- The exercises are a crucial part of the lecture
- You can only improve your programming skills by programming yourself
- Bit by bit we will implement the necessary routines for the (parallel) groundwater flow solver

Transparencies and Exercises

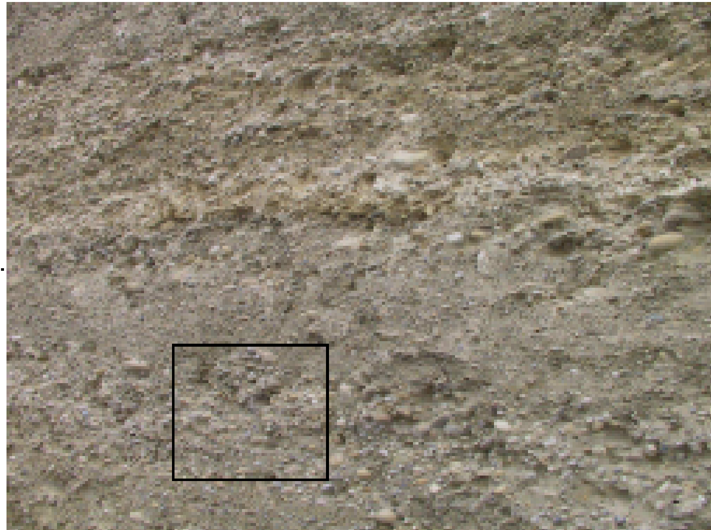
Transparencies
`http://www.ipvs.uni-stuttgart.de/abteilungen/sgs/lehre/lehrveranstaltungen/vorlesungen/SS08/commase6_termine/start/en`

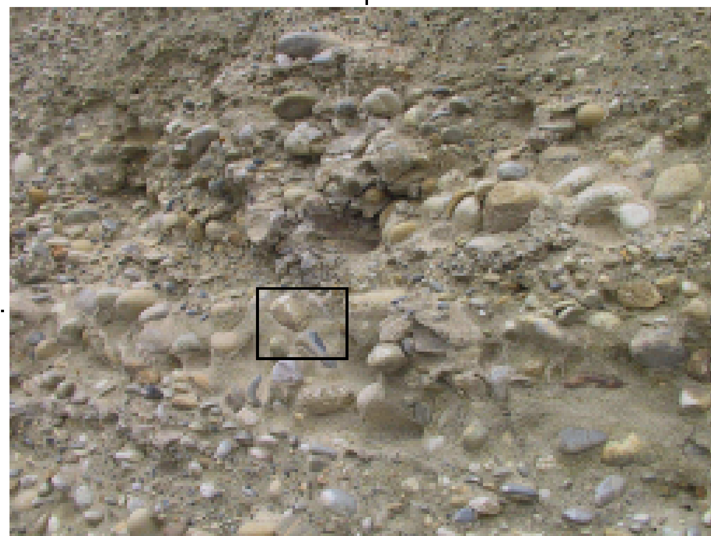
Exercises
`http://www.ipvs.uni-stuttgart.de/abteilungen/sgs/lehre/lehrveranstaltungen/uebungen/SS08/commase6_uebung_termine/start/en`

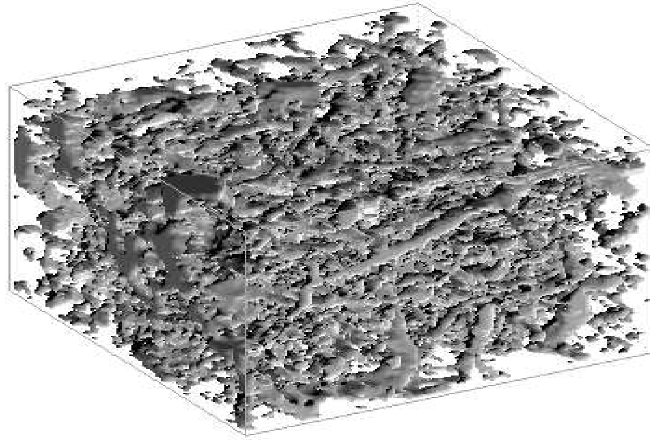
2 Groundwater Flow

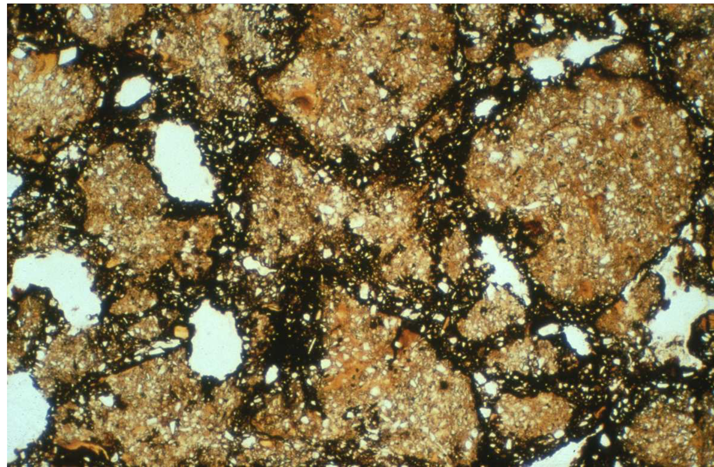


Heterogenity







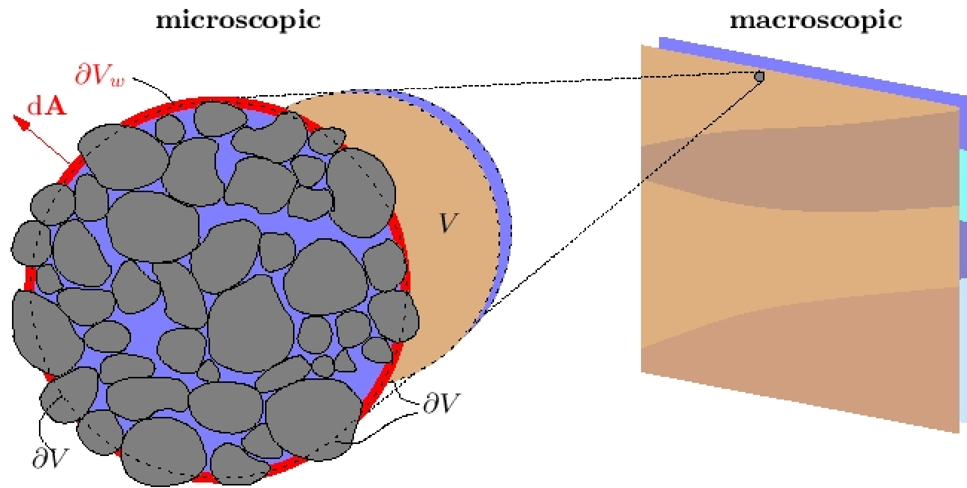


from: K. Roth (2005), Soil Physics - Lecture Notes v1.0, Institut für Umweltphysik, Universität Heidelberg
http://www.iup.uni-heidelberg.de/institut/forschung/groups/ts/students/lecture_notes05/lecture_notes05.html

Anisotropy

from: K. Roth (2005), Soil Physics - Lecture Notes v1.0, Institut für Umweltphysik, Universität Heidelberg
http://www.iup.uni-heidelberg.de/institut/forschung/groups/ts/students/lecture_notes05/lecture_notes05.html

Continuum approach



from: K. Roth (2005), Soil Physics - Lecture Notes v1.0, Institut für Umweltphysik, Universität Heidelberg
http://www.iup.uni-heidelberg.de/institut/forschung/groups/ts/students/lecture_notes05/lecture_notes05.html

Darcy Equation

H. Darcy (1856): Les Fontaines de la Ville de Dijon, Dalmont, Paris.

$$J_w = -K_s \cdot \frac{\Delta p_w}{\Delta x}$$

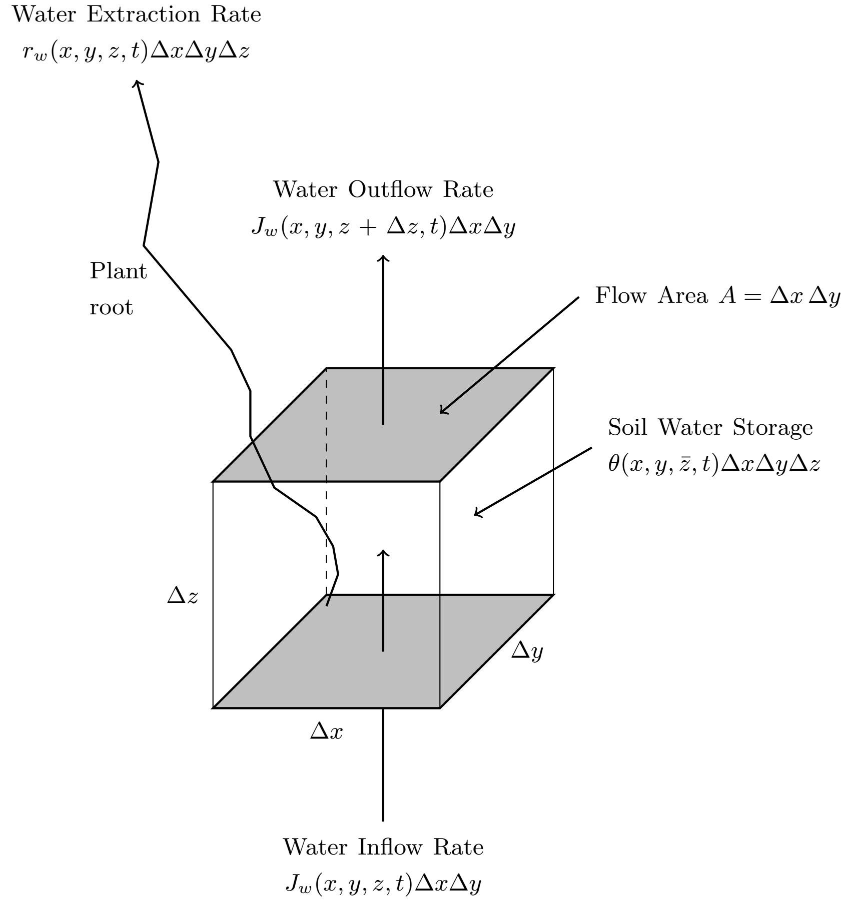
for $\Delta x \to 0$

$$J_w = -K_s \cdot \frac{\partial p_w}{\partial x}$$

in three dimensions:

$$\vec{J}_w = -\bar{K}_s \cdot \begin{pmatrix} \frac{\partial p_w}{\partial x} \\ \frac{\partial p_w}{\partial y} \\ \frac{\partial p_w}{\partial z} \end{pmatrix} = -\bar{K}_s \cdot \nabla p_w$$

Mass Conservation



according to W. A. Jury, R. Horton (2004): Soil Physics, 6th ed, Wiley & Sons, New Jersey

Transport Equation

$$\begin{aligned}
\frac{\partial\theta(\vec{x})}{\partial t} + \nabla\cdot\vec{J}_w(\vec{x}) + r_w(\vec{x}) &= 0 \\
\frac{\partial\theta(\vec{x})}{\partial t} + \nabla\cdot\left[-\bar{K}_s(\vec{x})\cdot\nabla p_w\right] + r_w(\vec{x}) &= 0 \\
\frac{\partial\theta(\vec{x})}{\partial t} - \nabla\cdot\left[\bar{K}_s(\vec{x})\cdot\nabla p_w\right] + r_w(\vec{x}) &= 0
\end{aligned}$$

with gravity:

$$\frac{\partial\theta(\vec{x})}{\partial t} - \nabla\cdot\left[\bar{K}_s(\vec{x})\cdot(\nabla p_w - \rho_w g\vec{e}_z)\right] + r_w(\vec{x}) = 0$$

steady state:

$$-\nabla\cdot\left[\bar{K}_s(\vec{x})\cdot(\nabla p_w - \rho_w g\vec{e}_z)\right] + r_w(\vec{x}) = 0$$

- can be described by Darcy's Law $J_w = -K_s\nabla p_w$ and the continuity equation $\frac{\partial\theta(\vec{x})}{\partial t} + \nabla\cdot\vec{J}_w(\vec{x}) + r_w(\vec{x}) = 0$.
- gravity is included by $\frac{\partial\theta(\vec{x})}{\partial t} - \nabla\cdot\left[\bar{K}_s(\vec{x})\cdot(\nabla p_w - \rho_w g\vec{e}_z)\right] + r_w(\vec{x}) = 0$

- heterogeneity is considered by different values of K_s at different positions of $\vec{x}$
- anisotropy is considered by using a tensor $\bar{K}_s$ instead of a scalar
- in steady state the flux equation is given by: $-\nabla \cdot \left[\bar{K}_s(\vec{x}) \cdot (\nabla p_w - \rho_w g \vec{e}_z)\right] + r_w(\vec{x}) = 0$

The groundwater flow equation can be used to determine the

- velocity of the groundwater at a certain point
- area of influence of a well
- distribution of a substance released at a certain point
- current density in a complex conductor
- temperature distribution e.g. on a computer chip

3 Partial Differential Equations

Identical Flux Laws

- Ohms' law, charge conservation:

Recapitulation: Partial Differential Equations

A partial differential equation

- determines a function $u(x)$ in $n \geq 2$ variables $x = (x_1, \ldots, x_n)^{\mathrm{T}}$.
- is a functional relation between partial derivatives of u at *one* point.

In general:

$$F\left(\frac{\partial^m u}{\partial x_1^m}(x), \frac{\partial^{m-1} u}{\partial x_1^{m-1}}(x), \ldots, \frac{\partial^m u}{\partial x_n^m}(x), \frac{\partial^{m-1} u}{\partial x_n^{m-1}}(x), \ldots, u(x)\right) = 0 \quad \forall x \in \Omega \tag{1}$$

Important:

- PDE's are posed in a domain Ω. Domains may be finite or infinite. Ω does *not* include its boundary
- The boundary of a domain is denoted by $\partial\Omega$
- The highest derivative m determines the order of a PDE
- $u : \Omega \to \mathbb{R}$ is called a solution of a PDE if it satisfies the PDE identically for every point $x \in \Omega$
- Solutions of PDE's are usually not unique unless additional conditions are posed. Typically these are conditions for the function values (and/or derivatives) at the boundary
- A PDE is well posed if the specified boundary conditions are unique and the solution depends continuously on the data

Linear partial PDE's of second order are a case of specific interest. For 2 dimensions and order $m = 2$ the general equation is:

$$\begin{aligned} &a(x,y)\frac{\partial^2 u}{\partial x^2}(x,y) + 2b(x,y)\frac{\partial^2 u}{\partial x \partial y}(x,y) + c(x,y)\frac{\partial^2 u}{\partial y^2}(x,y) \\ &+d(x,y)\frac{\partial u}{\partial x}(x,y) + e(x,y)\frac{\partial u}{\partial y}(x,y) + f(x,y)u(x,y) \\ &+g(x,y) = 0 \end{aligned}$$

At a point (x, y) a PDE can be classified according to the first three terms (main part) into

elliptic if $a(x,y)c(x,y) - b^2(x,y) > 0$

hyperbolic if $a(x,y)c(x,y) - b^2(x,y) < 0$

parabolic if $a(x,y)c(x,y) - b^2(x,y) = 0$ and rank of $\begin{bmatrix} a & b & d \\ b & c & e \end{bmatrix} = 2$ in (x, y)

The general linear PDE of 2nd order in n space dimensions is:

$$\underbrace{\sum_{i,j=1}^{n} a_{ij}(x)\partial_{x_i}\partial_{x_j} u}_{\text{main part}} + \sum_{i=1}^{n} a_i(x)\partial_{x_i} u + a_0(x) = 0 \quad \text{in } \Omega.$$

without loss of generality one can set $a_{ij} = a_{ji}$. With $(A(x))_{ij} = a_{ij}(x)$ the PDE is at a point x

elliptic if all eigenvalues of $A(x)$ have identical sign and no eigenvalue is zero.

hyperbolic if $(n - 1)$ eigenvalues have identical sign, one eigenvalue the opposite sign and no eigenvalue is zero.

parabolic if one eigenvalue is zero, all other eigenvalues have identical sign and the rank$[A(x), a(x)] = n$.

□

- Why this classification? Different solution techniques are necessary for the different types of PDE's.
- The described classification is *complete* for linear PDE's with $n = m = 2$. In higher space dimensions the classification is no longer complete.
- The type is invariant under coordinate transformation $\xi = \xi(x, y)$, $\eta = \eta(x, y)$ and $u(x, y) = \tilde{u}(\xi(x, y), \eta(x, y))$, which yields a new PDE for $\tilde{u}(\xi, \eta)$ with the coefficients $\tilde{a}, \tilde{b}$, etc.. If the equation for u in (x, y) has the type t than $\tilde{u}$ in $(\xi(x, y), \eta(x, y))$ has the same type.
- The type *can* vary at different points (but not in our applications).

- The type is only determined by the main part of the PDE (except for parabolic equations).
- Pathological cases like $\frac{\partial^2 u}{\partial x^2} + \frac{\partial u}{\partial x} = 0; u(x,y) = 0$ are avoided.

Definition 3.1. A linear PDE of 2nd order is called elliptic (hyperbolic, parabolic) in Ω if it is elliptic (hyperbolic, parabolic) for all points $(x, y) \in \Omega$. □

Definition 3.2 (Classification for first-order PDE's)**.** An equation of the form

$$d(x,y)\frac{\partial u}{\partial x}(x,y) + e(x,y)\frac{\partial u}{\partial y}(x,y) + f(x,y)u(x,y) + g(x,y) = 0$$

is called hyperbolic if $|d(x,y)| + |e(x,y)| > 0 \quad \forall (x,y) \in \Omega$ (else it is an ordinary differential equation). For $n \geq 2$ the equation $v(x) \cdot \nabla u(x) + f(x)u(x) + g(x) = 0$ is called hyperbolic. □

In this lecture we only cover scalar PDE's. Systems of PDE's contain several unknown functions $u_1, \ldots, u_n : \Omega \to \mathbb{R}$ and n PDE's. There is also a classification system for systems of PDE's.

3.1 Examples for PDE types

Poisson-Equation

$$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = f(x,y) \qquad \forall (x,y) \in \Omega \tag{2}$$

is called Poisson-Equation.

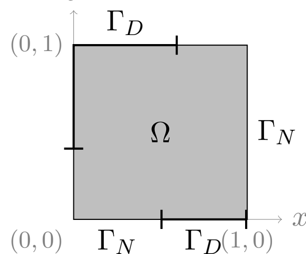
This is the prototype of an *elliptic* PDE. The solution of equation (2) is not unique. If $u(x,y)$ is a solution, then $u(x,y) + c_1 + c_2 x + c_3 y$ are also solutions for arbitrary values of c_1, c_2, c_3. To get a unique solution u values at the boundary have to be specified (we therefore call this a "boundary value problem").

Two types of boundary values are common:

1. $u(x,y) = g(x,y)$ for $(x,y) \in \Gamma_D \subseteq \partial\Omega$ (Dirichlet[1]),
2. $\frac{\partial u}{\partial \nu}(x,y) = h(x,y)$ for $(x,y) \in \Gamma_N \subset \partial\Omega$ (Neumann[2], flux),

and $\Gamma_D \cup \Gamma_N = \partial\Omega$. It is also important that $\Gamma_N \neq \partial\Omega$, as else the solution is only defined up to a constant.

Complete Poisson-Equation



$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= f \text{ in } \Omega \\ u &= g \text{ on } \Gamma_D \subseteq \partial\Omega \\ \frac{\partial u}{\partial \nu} &= h \text{ on } \Gamma_N = \partial\Omega \setminus \Gamma_D \neq \partial\Omega \end{aligned}$$

[1] Peter Gustav Lejeune Dirichlet, 1805-1859, German Mathematician.
[2] John von Neumann, 1903-1957, Austro-Hungarian Mathematician

Generalisation to n space dimensions:

$$\sum_{i=1}^{n} \frac{\partial^2 u}{\partial x_i^2} =: \Delta u = f \text{ in } \Omega$$

$$u = g \text{ on } \Gamma_D \subseteq \partial\Omega$$

$$\nabla u \cdot \nu = h \text{ on } \Gamma_N = \partial\Omega \setminus \Gamma_D$$

This equation is also called elliptic. If $f \equiv 0$ it is called Laplace-Equation. □

General Diffusion Equation

$K : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ is a projection, which relates to each point $x \in \Omega$ a $n \times n$ matrix $K(x)$. We demand also (for all $x \in \Omega$) that $K(x)$

1. $K(x) = K^{\mathrm{T}}(x)$ and $\xi^{\mathrm{T}} K(x) \xi > 0 \;\; \forall \xi \in \mathbb{R}^n,\, \xi \neq 0$ (symmetric positive definite),
2. $C(x) := \min \left\{ \xi^{\mathrm{T}} K(x) \xi \,\middle|\, \|\xi\| = 1 \right\} \geq C_0 > 0$ (uniform ellipticity).

$$\boxed{\begin{aligned} -\nabla \cdot \Big\{ K(x) \nabla u(x) \Big\} &= f \text{ in } \Omega \\ u &= g \text{ on } \Gamma_D \subseteq \partial\Omega \\ -\Big(K(x)\nabla u(x) \Big) \cdot \nu(x) &= h \text{ on } \Gamma_N = \partial\Omega \setminus \Gamma_D \neq \partial\Omega \end{aligned}} \tag{3}$$

is then called General Diffusion Equation (e.g. groundwater flow equation).

For strongly varying K equation (3) can be very difficult to solve. □

Wave-Equation

The prototype of a hyperbolic equation of second order is the Wave-Equation:

$$\frac{\partial^2 u}{\partial x^2}(x,y) - \frac{\partial^2 u}{\partial y^2}(x,y) = 0 \qquad \text{in } \Omega \quad . \tag{4}$$

Possible boundary values for a domain $\Omega = (0,1)^2$ are e.g.:
$x \in [0,1]$:

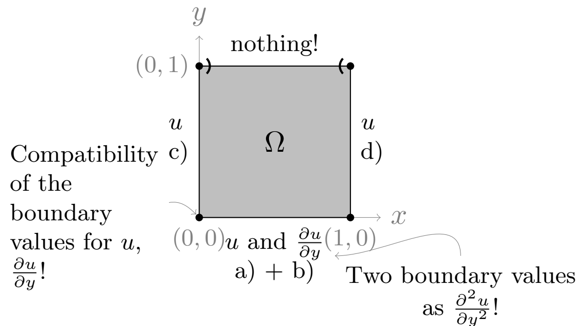
a) $u(x,0) = u_0(x)$

b) $\frac{\partial u}{\partial y}(x,0) = u_1(x)$

$y \in [0,1]$:

c) $u(0,y) = g_0(y)$

d) $u(1,y) = g_1(y)$

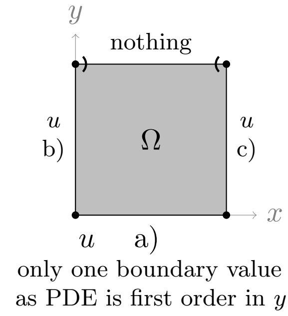


The direction y (usually the time) is special. a) + b) are called initial values and c) + d) boundary values. It is not possible to prescribe values at the whole boundary! □

Heat-Equation

The prototype of a parabolic equation is the heat equation:

$$\frac{\partial^2 u}{\partial x^2}(x,y) - \frac{\partial u}{\partial y}(x,y) = 0 \qquad \text{in } \Omega.$$



For a domain $\Omega = (0,1)^2$ typical boundary values are (with $x \in [0,1], y \in [0,1]$):

$$u(x,0) = u_0(x)$$
$$u(0,y) = g_0(y) \qquad\qquad u(1,y) = g_1(y)$$

□

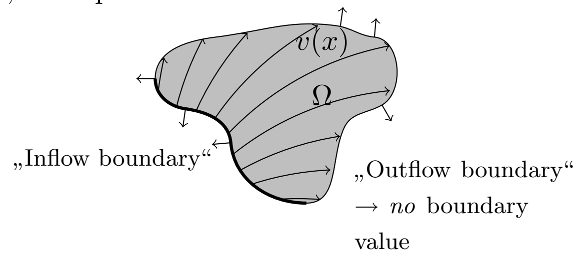
Transport-Equation

If $\Omega \subset \mathbb{R}^n, v : \Omega \to \mathbb{R}^n$ is a given vector field, the equation

$$\nabla \cdot \{v(x)u(x)\} = f(x) \qquad \text{in } \Omega$$

is called stationary transport equation and is a hyperbolic PDE of first order. Possible boundary values are

$$u(x) = g(x)$$



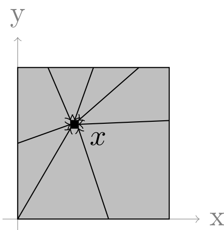
for $x \in \partial\Omega$ with $v(x) \cdot \nu(x) < 0$ (Boundary value depends on the flux field)
$\frac{\partial u}{\partial t} + \nabla \cdot \{v(x,t)u(x,t)\} = f(x,t)$ is also a hyperbolic PDE of first order. □

3.2 Sphere of Influence

The type of a partial differential equation can also be illustrated with the following question:

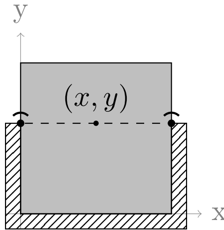
> Given $x \in \Omega$. Which initial/boundary values influence the solution u at the point x?

Elliptic $u_{xx} + u_{yy} = 0$



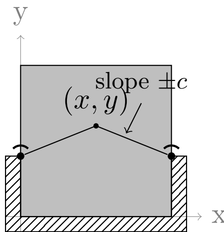
all boundary values influence $u(x)$, i.e. Change in $u(y), y \in \partial\Omega \Rightarrow$ Change in $u(x)$.

Parabolic $u_{xx} - u_y = 0$ Note: The $-$ is crucial, $+$ is parabolic according to the definition *but* it is not well posed (stable)



for (x, y) all (x', y') with $y' \leq y$ influence the value at x.
„infinite velocity of propagation"

Hyperbolic (2nd order) $u_{xx} - u_{yy} = 0$

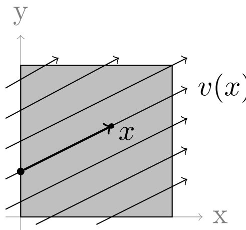


Solution at (x, y) is influenced by all boundary values below the cone

$$\{(x', y') \quad | \quad y' \leq (x' - x) \cdot c + y \quad \wedge \quad y' \leq (x - x') \cdot c + y\} \quad \cap \quad \partial\Omega$$

„finite velocity of propagation"

Hyperbolic (1st order) $u_x + u_y = 0$



Only one boundary point influences the value.

- The steady-state groundwater flow equation $-\nabla \cdot \left[\bar{K}_s(\vec{x}) \cdot (\nabla p_w - \rho_w g \vec{e}_z)\right] + r_w(\vec{x}) = 0$ is an elliptic partial differential equation of second order.
- To get a well posed problem either Dirichlet boundary conditions (the pressure value is given) or Neumann boundary conditions (the flux is given) must be specified at each boundary point.
- At one point of the boundary a Dirichlet boundary condition should be specified (else the equation is only defined up to a constant).
- Each point in the domain is influenced by all boundary conditions.

4 Spatial-Discretisation Methods

4.1 Recapitulation: The Finite-Difference Method

- Partial derivatives are replaced with difference quotients (Taylor series expansion)
- Dirichlet boundary conditions can easily be integrated by rearranging the equation systems and bringing them to the right side of the equation.

- Neumann boundary conditions are integrated by either replacing them with a forward difference formula or by introduction of ghost nodes
- Only on a equidistant grid the Finite-Difference Method is second-order accurate
- Advantages:
  - easy to formulate and implement
  - well suited for structured grids
- Problems:
  - Non-equidistant grids
  - What's the value between two points?
  - Complex domains
  - In general not locally mass-conservative.

4.2 Recapitulation: The Finite-Element Method

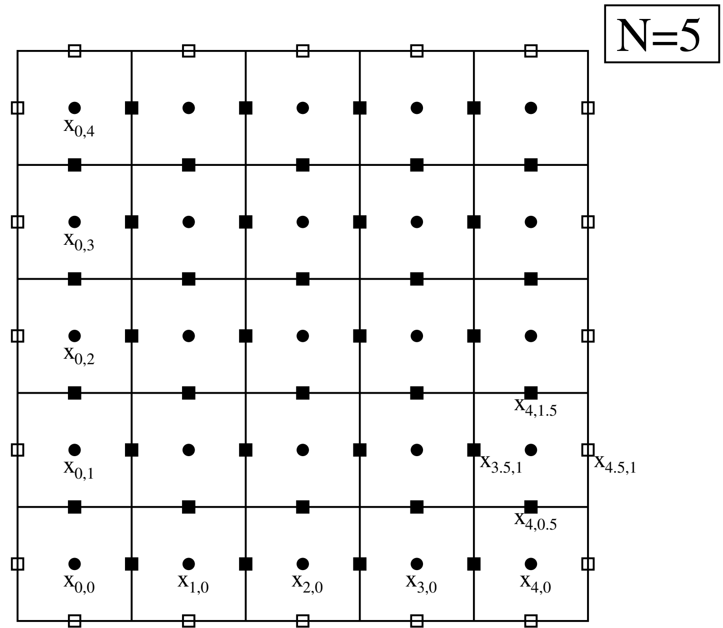
- A trial function is inserted in the partial differential equation, resulting in a residual.
- The free coefficients of the trial functions are chosen such that the integral over the product of the residual and chosen weight functions vanishes (weak formulation).
- Finite element methods differ in the the choice of the trial and weight functions.
- Dirichlet boundary conditions can be directly incorporated into the trial functions.
- Neumann boundary conditions are handled in the integrals.
- Convergence order depends on the choice of weight and trial functions.
- Advantages:
  - can be used for domains with complicated shape
  - well suited for unstructured grids
  - local adaptivity possible
- Problems:
  - grid generation can be complicated (must often fullfill certain conditions)
  - more computationally expensive for simple problems
  - not allways locally mass-conservative

4.3 The Finite-Volume Method

- Only the integral of the partial differential equation over each a grid cell must fullfill the equation.
- Implementation of Dirichlet Boundary and Neumann Boundary conditions straight forward
- Structured and unstructured grids possible

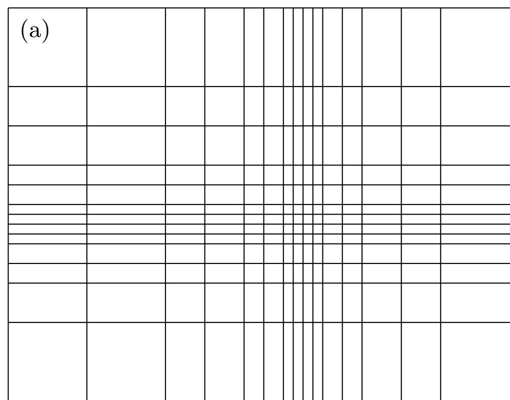
- Dirichlet boundary conditions can easily be integrated by rearranging the equation systems and bringing them to the right side of the equation.
- Neumann boundary conditions can easily be integrated in the flux integrals
- Convergence order can differ dependent on the concrete method.

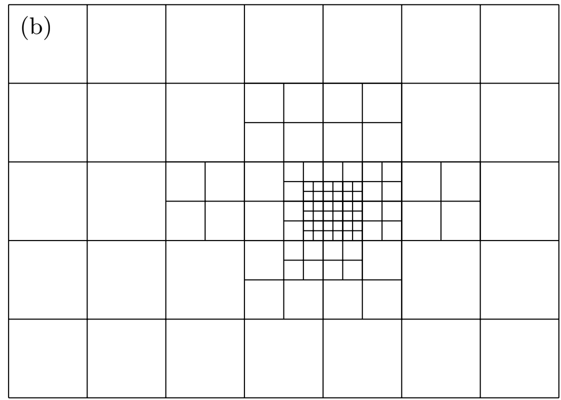
Divide Domain into Grid Cells

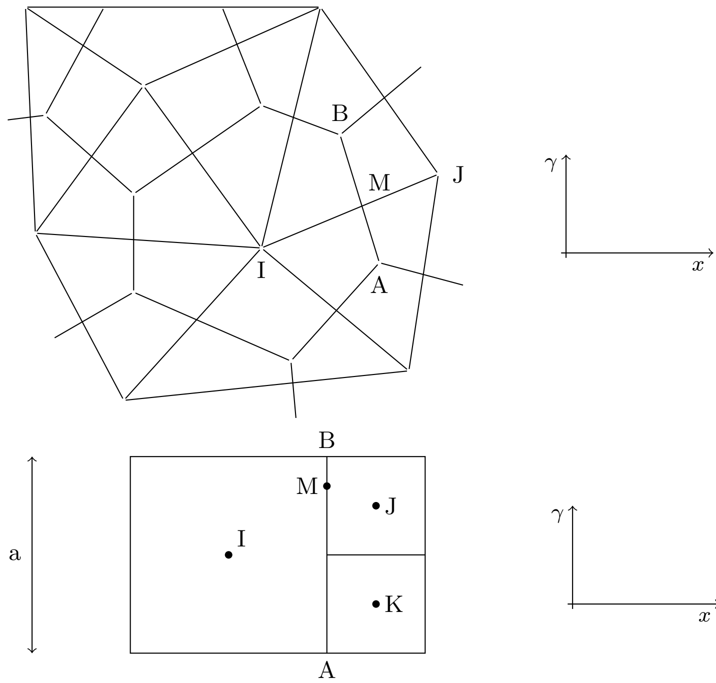


Divide grid into rectangular grid cells g_{ij}

Other Possible Grids with Finite Volume Methods





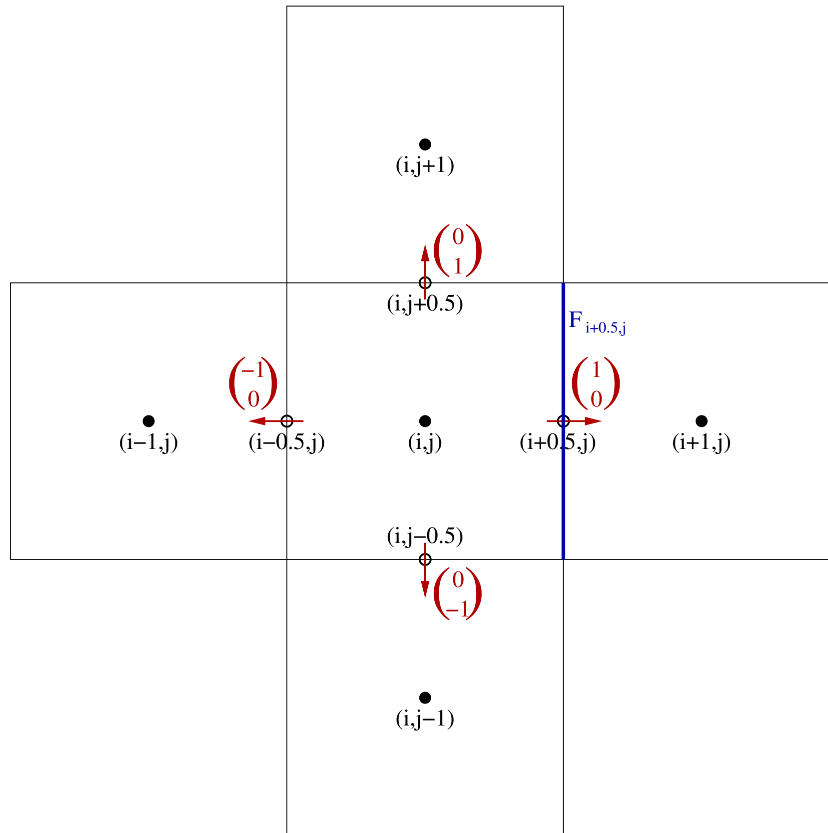


Transformation of Volume Integral into Boundary Integral

$$\int_{g_{ij}} \nabla \cdot \vec{J}_w \;\; dx\, dy = \int_{g_{ij}} r(\vec{x}) \;\; dx\, dy$$

$$\underbrace{\Leftrightarrow}_{Satz\, of\, Gauss} \quad \int_{\partial g_{ij}} \vec{J}_w \cdot \vec{n} \;\; ds = \int_{g_{ij}} r(\vec{x}) \;\; dx\, dy$$

Inner Grid Cell



Finite Volume Discretisation: Split into Sum over Faces

$$\int\limits_{\partial g_{ij}} \vec{J}_w \cdot \vec{n} \;\; ds = \sum_{k=i\pm 0.5} \int\limits_{F_{kj}} \vec{J}_w \cdot \vec{n} \;\; ds + \sum_{l=j\pm 0.5} \int\limits_{F_{il}} \vec{J}_w \cdot \vec{n} \;\; ds$$

$$\underbrace{\approx}_{\text{Midpointrule}} \sum_{k=i\pm 0.5} \vec{J}_w(\vec{x}_{k,j}) \cdot \vec{n} \cdot \underbrace{h}_{\text{Face Area}} + \sum_{l=j\pm 0.5} \vec{J}_w(\vec{x}_{i,l}) \cdot \vec{n} \cdot \underbrace{h}_{\text{Face Area}}$$

with

$$\vec{J}_w(\vec{x}) = -\begin{pmatrix} K_{xx}(\vec{x}) & 0 \\ 0 & K_{yy}(\vec{x}) \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial p}{\partial x}(\vec{x}) \\ \frac{\partial p}{\partial y}(\vec{x}) \end{pmatrix}$$

Finite Volume Discretisation: Insert Flux law

$$
\begin{aligned}
&\sum_{k=i\pm 0.5} \vec{J}_w(\vec{x}_{k,j}) \cdot \vec{n} \cdot h + \sum_{l=j\pm 0.5} \vec{J}_w(\vec{x}_{i,l}) \cdot \vec{n} \cdot h = \\
&-K_{xx}(\vec{x}_{i-0.5,j}) \cdot \frac{\partial p}{\partial x}(\vec{x}_{i-0.5,j}) \cdot \underbrace{(-1)}_{\text{from } n_x} \cdot h \\
&-K_{xx}(\vec{x}_{i+0.5,j}) \cdot \frac{\partial p}{\partial x}(\vec{x}_{i+0.5,j}) \cdot \underbrace{(1)}_{\text{from } n_x} \cdot h \\
&-K_{yy}(\vec{x}_{i,j-0.5}) \cdot \frac{\partial p}{\partial x}(\vec{x}_{i,j-0.5}) \cdot \underbrace{(-1)}_{\text{from } n_y} \cdot h \\
&-K_{yy}(\vec{x}_{i,j+0.5}) \cdot \frac{\partial p}{\partial x}(\vec{x}_{i,j+0.5}) \cdot \underbrace{(1)}_{\text{from } n_y} \cdot h
\end{aligned}
$$

Finite Volume Discretisation: Approximate Derivatives

$$
\begin{aligned}
\underbrace{\approx}_{\text{approx.Derivative}} \quad &+K_{xx}(\vec{x}_{i-0.5,j}) \cdot \frac{p(\vec{x}_{i,j}) - p(\vec{x}_{i-1,j})}{h} \cdot h \\
&-K_{xx}(\vec{x}_{i+0.5,j}) \cdot \frac{p(\vec{x}_{i+1,j}) - p(\vec{x}_{i,j})}{h} \cdot h \\
&+K_{yy}(\vec{x}_{i,j-0.5}) \cdot \frac{p(\vec{x}_{i,j}) - p(\vec{x}_{i,j-1})}{h} \cdot h \\
&-K_{yy}(\vec{x}_{i,j+0.5}) \cdot \frac{p(\vec{x}_{i,j+1}) - p(\vec{x}_{i,j})}{h} \cdot h
\end{aligned}
$$

Midpoint rule for source/sink term:

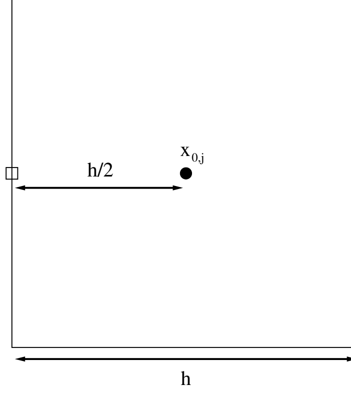
$$
\int_{g_{ij}} r(\vec{x}) dx\, dy \approx h^2 r(\vec{x}_{i,j})
$$

Matrix Contribution of each Grid Cell

$$
\begin{aligned}
-K_{xx}(\vec{x}_{i-0.5,j}) \cdot p_{i-1,j} - K_{xx}(\vec{x}_{i+0.5,j}) \cdot p_{i+1,j} & \\
-K_{yy}(\vec{x}_{i,j-0.5}) \cdot p_{i,j-1} - K_{yy}(\vec{x}_{i,j+0.5}) \cdot p_{i,j+1} & \\
+\left[K_{xx}(\vec{x}_{i-0.5,j}) + K_{xx}(\vec{x}_{i+0.5,j}) + K_{yy}(\vec{x}_{i,j-0.5}) + K_{yy}(\vec{x}_{i,j+0.5})\right] \cdot p_{i,j} &= h^2 r(\vec{x}_{i,j})
\end{aligned}
$$

Dirichlet Boundary

e.g. $x = 0$:



Compute derivative between inner point and boundary point:

$$\frac{\partial p}{\partial x}(\vec{x}_{-0.5,j}) \approx \frac{p(\vec{x}_{0,j}) - p_d(0, y_j)}{h/2}$$

Matrix Contribution at Dirichlet Boundary $x = 0$

$$\begin{aligned} -K_{xx}(\vec{x}_{i+0.5,j}) \cdot p_{i+1,j} & \\ -K_{yy}(\vec{x}_{i,j-0.5}) \cdot p_{i,j-1} - K_{yy}(\vec{x}_{i,j+0.5}) \cdot p_{i,j+1} & \\ +[2K_{xx}(\vec{x}_{i-0.5,j}) + K_{xx}(\vec{x}_{i+0.5,j}) & \\ +K_{yy}(\vec{x}_{i,j-0.5}) + K_{yy}(\vec{x}_{i,j+0.5})] \cdot p_{i,j} &= h^2 r(\vec{x}_{i,j}) + 2K_{xx}(\vec{x}_{i-0.5,j}) \cdot p_d(0, y_j) \end{aligned}$$

Neumann Boundary

For each volume we have to calculate

$$\int\limits_{\partial g_{ij}} \vec{J}_w \cdot \vec{n} \;\, ds = \sum_{k=i\pm 0.5} \int\limits_{F_{kj}} \vec{J}_w \cdot \vec{n} \;\, ds + \sum_{l=j\pm 0.5} \int\limits_{F_{il}} \vec{J}_w \cdot \vec{n} \;\, ds$$

At a Neumann boundary $\vec{J}_w \cdot \vec{n}$ is given by the boundary condition $\phi_n(\vec{x})$, we can therefore use

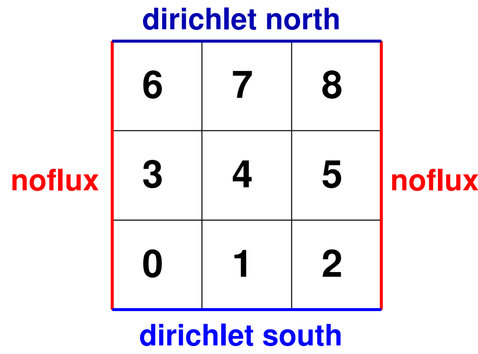
$$\int\limits_{F_{kl}} \vec{J}_w \cdot \vec{n} \;\, ds \underbrace{\approx}_{\text{Midpointrule}} h \cdot \phi_N(\vec{x})$$

at each Neumann boundary.

Matrix Contribution at Neumann Boundary $x = 0$

$$\begin{aligned} -K_{xx}(\vec{x}_{i+0.5,j}) \cdot p_{i+1,j} & \\ -K_{yy}(\vec{x}_{i,j-0.5}) \cdot p_{i,j-1} - K_{yy}(\vec{x}_{i,j+0.5}) \cdot p_{i,j+1} & \\ +[K_{xx}(\vec{x}_{i+0.5,j}) + K_{yy}(\vec{x}_{i,j-0.5}) + K_{yy}(\vec{x}_{i,j+0.5})] \cdot p_{i,j} &= h^2 r(\vec{x}_{i,j}) - h \cdot \phi_N(\vec{x}_{-0.5,j}) \end{aligned}$$

Example: 3×3 **Grid**



$$K(\vec{x}) = \begin{pmatrix} K & 0 \\ 0 & K \end{pmatrix}$$

$$\begin{pmatrix} 4K & -K & 0 & -K & 0 & 0 & 0 & 0 & 0 \\ -K & 5K & -K & 0 & -K & 0 & 0 & 0 & 0 \\ 0 & -K & 4K & 0 & 0 & -K & 0 & 0 & 0 \\ -K & 0 & 0 & 3K & -K & 0 & -K & 0 & 0 \\ 0 & -K & 0 & -K & 4K & -K & 0 & -K & 0 \\ 0 & 0 & -K & 0 & -K & 3K & 0 & 0 & -K \\ 0 & 0 & 0 & -K & 0 & 0 & 4K & -K & 0 \\ 0 & 0 & 0 & 0 & -K & 0 & -K & 5K & -K \\ 0 & 0 & 0 & 0 & 0 & -K & 0 & -K & 4K \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{pmatrix} = \begin{pmatrix} 2Kp_{d_{\text{south}}} \\ 2Kp_{d_{\text{south}}} \\ 2Kp_{d_{\text{south}}} \\ 0 \\ 0 \\ 0 \\ 2Kp_{d_{\text{north}}} \\ 2Kp_{d_{\text{north}}} \\ 2Kp_{d_{\text{north}}} \end{pmatrix}$$

Permeability

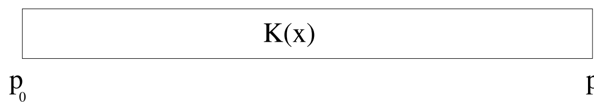
We assume that the permeability is a diagonal Tensor, which is depending on the position, but constant on each grid cell g_{ij}.

We need to evaluate K at the cell boundaries $x_{i\pm 0.5, j\pm 0.5}$.

What is the correct value?

1D-Example

$$\begin{aligned} \frac{dJ_w}{dx} &= 0 \quad \text{in } \Omega = (0, \underbrace{\ell}_{\text{length}}) \\ J_w &= -K(x)\frac{dp}{dx} \end{aligned}$$



with

$$\begin{aligned} p(0) &= p_0 \\ p(\ell) &= p_\ell \end{aligned}$$

because of $\frac{dJ_w}{dx} = 0$ in $\Omega \Leftrightarrow J_w(x) = J_0 \in \mathbb{R}$ this means

$$J_0 = -K(x)\frac{dp}{dx} \Leftrightarrow \frac{dp}{dx} = -\frac{J_0}{K(x)}$$

$$\begin{aligned}
&\frac{dp}{dx} = -\frac{J_0}{K(x)} \\
\Leftrightarrow &\int_0^{\ell} \frac{dp}{dx} dx = [p(x)]_0^{\ell} = p_\ell - p_0 = -J_0 \int_0^{\ell} \frac{1}{K(x)} dx \\
\Leftrightarrow &J_0 = -\underbrace{\frac{\ell}{\int_0^{\ell} \frac{1}{K(x)} dx}}_{\text{eff.permeability}} \cdot \underbrace{\frac{p_\ell - p_0}{\ell}}_{\text{approx.gradient}}
\end{aligned}$$

1D-Example, cell-wise constant Permeability

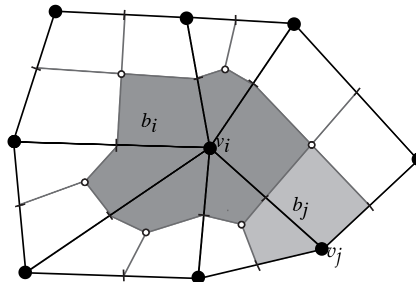
if $K(x) = \begin{cases} K_l & x \leq \frac{\ell}{2} \\ K_r & x > \frac{\ell}{2} \end{cases}$

$$K_{\text{eff}} = \frac{\ell}{\int_0^{\ell} \frac{1}{K(x)} dx} = \frac{\ell}{\frac{\ell}{2}\frac{1}{K_l} + \frac{\ell}{2}\frac{1}{K_r}} = \frac{2}{\frac{1}{K_l} + \frac{1}{K_r}}$$

We therefore choose for cell-wise constant permeabilities

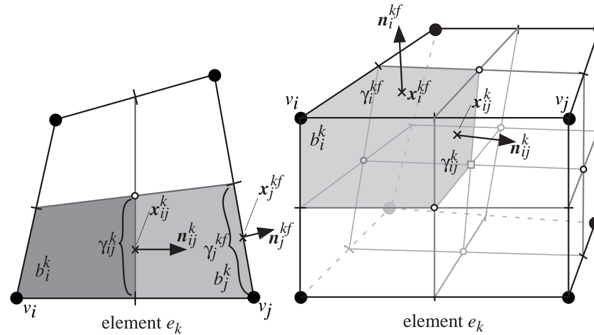
$$K(\vec{x}_{i\pm 0.5,j}) = \frac{2}{\frac{1}{K(\vec{x}_{i,j})} + \frac{1}{K(\vec{x}_{i\pm 1,j})}}$$

4.4 The Vertex-Centered Finite-Volume Method



- The unknowns are located at the edges of the elements (vertices)

- Base functions are used on each element, which are parameterised with the values at the vertices
- A secondary mesh is constructed connecting the face centers and the barycenter of the element
- The flux balance is not calculated over the original grid, but over the secondary mesh, the elements of the secondary mesh are called control-volumes, the parts of a control volume belonging to a specific element of the primary mesh are called subcontrol-volumes.



- Material properties are assumed to be constant for each element
- The volume integrals are calculated as a sum over the subcontrol-volumes using the midpoint rule and the material properties valid for the specific control-volume. $\sum_i b_i^k \cdot r_i^k$
- The face integrals are calculated as a sum over all subcontrol-volume faces with the midpoint rule $\sum_{ij} \gamma_{ij}^k \vec{J}_{ij}^k \vec{n}_{ij}^k$
- The gradient at the face centers is given by the base functions.

Properties of the Vertex-Centered Finite-Volume Method

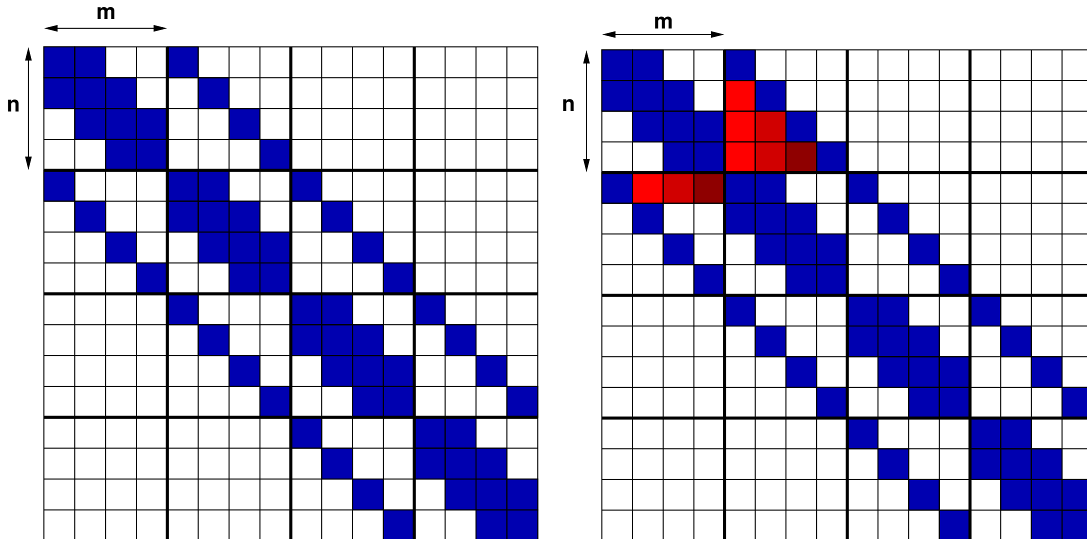
- Advantages:
  - can be used for domains with complicated shape
  - well suited for unstructured grids
  - local adaptivity possible
  - locally mass conservative
- Problems:
  - grid generation can be complicated (must often fullfill certain conditions)
  - more computationally expensive for simple problems

5 Solution of Linear Equation Systems

5.1 Direct Solution of Sparse Linear Equation Systems

Direct Solution of Sparse Linear Equation Systems

We do a Gaussian eliminitation for $A \cdot \vec{x} = \vec{b}$ with $A \in \mathbb{R}^{N \times N}$ regular, $\vec{x}, \vec{b} \in \mathbb{R}^N$ and A is a matrix assembled by the Finite-Volume method.



- As A is symmetric, positive definite the elimination can be done without pivoting
- New non-zero elements are created during the eliminitation ("fill in")
- The "fill in" is created within the outer diagonal

Complexity of the Elimination

Due to the "fill in" '$O(N) = O(n \cdot m)$ matrix entries become $O(n \cdot m \cdot m) = O(n \cdot m^2)$ matrix entries after the elimination.

The complexity of the elimination is:

$$\text{Complexity} \leq \sum_{i=1}^{N} \underbrace{m}_{\substack{\#\text{ elements}\\ \text{to eliminate}\\ \text{till diagonal}\\ \text{in line } i}} \cdot \underbrace{m}_{\substack{\text{lower limit}\\ \text{for}\\ \text{elimination}\\ \text{of one}\\ \text{element}}} = N \cdot m^2 = n \cdot m^3$$

If $n = m$ the complexity of the elimination is $O(N^2)$, with optimal numbering of the nodes $O(N^{3/2})$, compared to $O(N^3)$ with a fully occupied matrix.

In three dimensions: The elimination has a complexity of $O(N^{7/3})$

In one dimension: The elimination has an optimal complexity of $O(N)$

5.2 Iterative Solution of Sparse Linear Equation Systems

Starting from an initial value $\vec{x}^{(0)} \in \mathbb{R}^N$, iterative solution methods create a sequence

$$\vec{x}^{(0)}, \vec{x}^{(1)}, \dots, \vec{x}^{(k)}, \dots$$

with the characteristic

$$\lim_{k \to \infty} \vec{x}^{(k)} = \vec{x}.$$

5.2.1 Relaxation Methods

The ith equation in $A\vec{x} = \vec{b}$ is:

$$\sum_{j=1}^{N} a_{ij} x_j = b_i$$

solve for x_i:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j \right)$$

Precondition: $a_{ii} \neq 0 \quad \forall i = 1 \dots N$. This is not true for all matrices

Gauß-Seidel Iteration: Algorithm

Update all columns one after the other:

given $\vec{x}^{(k)}$

for $(i = 1; i \leq N; i = i + 1)$

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right)$$

yields $\vec{x}^{(k+1)}$

This scheme is called Gauß-Seidel Iteration.

Complexity for calculation of $\vec{x}^{(k+1)}$ from $\vec{x}^{(k)}$ proportional to number of non-zero elements of the matrix, therefore $O(N)$ for sparse matrices.

Open Questions

- Under which conditions is the sequence converging with $\lim_{k \to \infty} \vec{x}^{(k)} = \vec{x}$.
- How many iterations are necessary to reach $||\vec{x}^{(k)} - \vec{x}|| \leq \epsilon$ for a given precision ϵ?
- How can one determine efficiently if $||\vec{x}^{(k)} - \vec{x}|| \leq \epsilon$ is reached? (we don't know the exact solution $\vec{x}$)

Other Relaxation Methods

Jacobi Iteration:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

Damped Jacobi Iteration:

$$x_i^{(k+1)} = (1 - \omega)\, x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

special case: $\omega = 1 \Rightarrow$ Jacobi Iteration

SOR (successive overrelaxation) Iteration:

$$x_i^{(k+1)} = (1 - \omega)\, x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right)$$

$0 < \omega < 1$: underrelaxation $1 < \omega < 2$: overrelaxation special case: $\omega = 1 \Rightarrow$ Gauß-Seidel Iteration

Damped Richardson Iteration:

$$x_i^{(k+1)} = (1 - a_{ii}\omega)\, x_i^{(k)} + \omega \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

Matrix Notation of Relaxation Methods

For an analysis of the convergence behavior it is more convenient to write the iteration schemes as matrix operations:

As $\vec{x} = \vec{x}^{(k)} + \vec{e}^{(k)}$ and $A\vec{e}^{(k)} = \vec{b} - A\vec{x}^{(k)}$ we could calculate $\vec{x}$ from

$$\vec{x} = \vec{x}^{(k)} + A^{-1} \left(\vec{b} - A\vec{x}^{(k)} \right)$$

However inverting A is at least as expensive as calculating the solution of $A\vec{x} = \vec{b}$ with a direct method. We therefore approximate the matrix A^{-1} with a matrix M^{-1}, where M is an approximation of A, which is easy to invert, and get the new formula

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + M^{-1} \left(\vec{b} - A\vec{x}^{(k)} \right)$$

$\vec{x}^{(k+1)}$ is no longer the exact solution but (hopefully) an improvement to $\vec{x}^{(k)}$

Matrix Notation of Common Relaxation Methods

Split $A = L + D + U$ into a strictly lower tridiagonal matrix L, a strictly upper tridiagonal matrix U and a diagonal Matrix D.

Now we can get the iteration methods described above by

$M = \omega^{-1} I$	damped Richardson iteration
$M = D$	Jacobi iteration
$M = \omega^{-1} D$	damped Jacobi iteration
$M = L + D$	Gauß-Seidel iteration
$M = L + \omega^{-1} D$	SOR iteration

Iteration Matrix

For the general iteration scheme we get:

$$\begin{aligned}
\vec{x}^{(k+1)} &= \vec{x}^{(k)} + M^{-1}\left(\vec{b} - A\vec{x}^{(k)}\right) \\
\Leftrightarrow \underbrace{\vec{x} - \vec{x}^{(k+1)}}_{\vec{e}^{(k+1)}} &= \underbrace{\vec{x} - \vec{x}^{(k)}}_{\vec{e}^{(k)}} - M^{-1}\left(\vec{b} - A\vec{x}^{(k)}\right) \\
\vec{e}^{(k+1)} &= \vec{e}^{(k)} - M^{-1}\left(A\vec{x} - A\vec{x}^{(k)}\right) \\
&= \vec{e}^{(k)} - M^{-1}A\left(\vec{x} - \vec{x}^{(k)}\right) \\
&= \underbrace{\left(I - M^{-1}A\right)}_{=:S}\vec{e}^{(k)}
\end{aligned}$$

We call $S = I - M^{-1}A$ the iteration matrix.

Error Propagation

The error propagation is therefore:

$$\vec{e}^{(k+1)} = S \cdot \vec{e}^{(k)}$$

with the iteration matrix $S = I - M^{-1}A$.

Recursive insertion yields:

$$\vec{e}^{(k)} = S \cdot \vec{e}^{(k-1)} = S^2 \cdot \vec{e}^{(k-2)} = \cdots = S^k \cdot \vec{e}^{(0)}$$

If $\lim\limits_{k\to\infty} S^k = 0$ (zero matrix) the scheme converges independently of $\vec{e}^{(0)}$.

This is guaranteed if $\rho(S) < 1$, where $\rho(S) = \max\{|\lambda| \;\big|\; \lambda \text{ is eigenvalue of } S\}$ is called spectral radius of S.

Eigenvalues and Eigenvectors

- If A is symmetric and positive definite (and often if it is not) $\Rightarrow$ there exists a set of N linearly independent eigenvectors $\vec{z}_1, \vec{z}_2, \ldots, \vec{z}_N$.
- If $\vec{z}_i$ is eigenvector of A, $\alpha\vec{z}_i$ with $\alpha \in \mathbb{R}$ is also eigenvector of A.
- The product of A and z_i is equal to z_i times the scalar eigenvalue λ_i:

$$A\vec{z}_i = \lambda_i \vec{z}_i$$

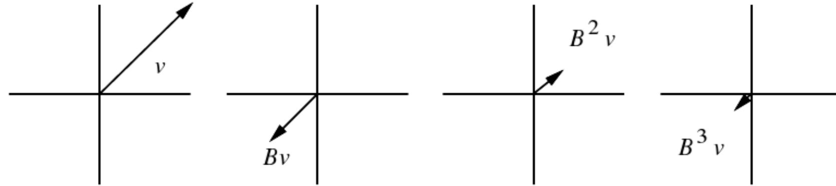
- As the N eigenvectors are linearly independent, they form a basis of $\mathbb{R}^N$, i.e. every vector $\vec{x}$ can be expressed as a linear combination of the eigenvectors.

$$\vec{x} = \sum_{i=1}^{N} \xi_i \vec{z}_i$$

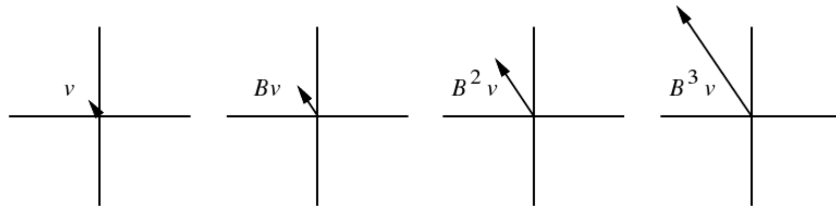
- As matrix-vector multiplication is distributive:

$$A\vec{x} = \sum_{i=1}^{N} \xi_i A \vec{z}_i = \sum_{i=1}^{N} \xi_i \lambda_i \vec{z}_i$$

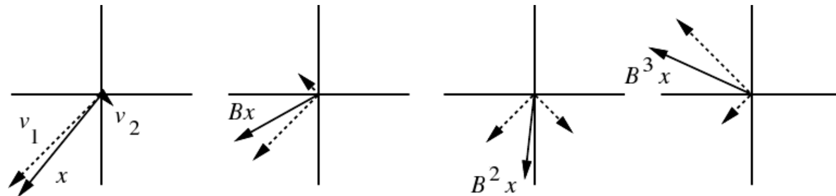
Matrix multiplication with eigenvector if eigenvalue < 1



Matrix multiplication with eigenvector if eigenvalue > 1



Matrix multiplication with vector which is sum of two eigenvectors



figures from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

Convergence Analysis for the Damped Richardson Iteration

If we assume that A is symmetric and positive definite $\Rightarrow$ all eigenvalues of A are real and positive. A convergence analysis can then easily be made for the damped Richardson iteration ($M = \omega^{-1} I$) with the iteration formula:

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \omega \left(\vec{b} - A\vec{x}^{(k)} \right)$$

The iteration matrix S is $S = I - M^{-1}A = I - \omega A$ and has the eigenvalues $\mu_i = 1 - \omega \lambda_i$ where λ_i is an eigenvalue of A.

If we use $\omega = \frac{1}{\lambda_{\mathsf{max}}(A)}$, we get:

$$0 = 1 - \frac{\lambda_{\mathsf{max}}(A)}{\lambda_{\mathsf{max}}(A)} \leq \mu_i \leq 1 - \frac{\lambda_{\mathsf{min}}(A)}{\lambda_{\mathsf{max}}(A)} = 1 - \frac{1}{\kappa_2(A)}$$

We call $\kappa_2(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$ the spectral condition of A.

The spectral Radius of S is:

$$\rho(S) \leq 1 - \frac{1}{\kappa(A)_2}, \text{ with } \kappa(A) \geq 1$$

Convergence for Matrices from PDE-Discretizations

For the solution of the Laplace equation

$$\Delta p = \nabla \cdot (\nabla p) = 0$$

in $\Omega \subset \mathbb{R}^d$ with the Finite-Difference discretisation (i.e. $A \in \mathbb{R}^{N \times N}$) we get $\kappa_2(A) = O(N^{2/d})$.

$\Rightarrow$ the defect reduction is decreasing with increasing matrix size.

Similar results can be obtained for other relaxation methods like Jacobi or Gauss-Seidel iteration.

Further Convergence Results

- If A and $2 \cdot D - A$ are both positive definite the Jacobi iteration converges.
- If A is strictly diagonally dominant ($a_{ii} > \sum_{j \neq i} |a_{ij}| \forall i$) the Jacobi and Gauß-Seidel iterations converge.
- SOR can only converge if $0 < \omega < 2$.
- If A is positive definite, both SOR and Gauß-Seidel converge.
- For many problems occurring in engineering no convergence proofs exist.

Terminating Condition

We call $\vec{e}^{(k)} := \vec{x} - \vec{x}^{(k)}$ the error of the kth iterate. As we do not know the exact solution $\vec{x}$ the error is hard to determine.

With

$$A\vec{e}^{(k)} = A\left(\vec{x} - \vec{x}^{(k)}\right) = A\vec{x} - A\vec{x}^{(k)} = b - A\vec{x}^{(k)} =: \vec{d}^{(k)}$$

we derive the defect vector $\vec{d}^{(k)} := \vec{b} - A\vec{x}^{(k)}$, which can be computed easily.

Because of $A\vec{e}^{(k)} = \vec{d}^{(k)} \Leftrightarrow \vec{e}^{(k)} = A^{-1}\vec{d}^{(k)}$ and therefore $||\vec{e}^{(k)}|| \leq ||A^{-1}|| \cdot ||\vec{d}^{(k)}||$

we can use the norm of the defect $||d^{(k)}||$ as terminating condition.

As $||A^{-1}||$ can be very large, we use a relative termination criterium: $||d^{(k)}|| < \varepsilon ||d^{(0)}||$ with a suitable ε.

Defect Calculation

The new defect is not calculated from $\vec{d}^{(k+1)} = \vec{b} - A\vec{x}^{(k+1)}$ as with this formulation cancelation errors are increasing if the defect gets smaller.

The defect in step $k+1$ is:

$$\begin{aligned}\vec{d}^{(k+1)} &= \vec{b} - A\vec{x}^{(k+1)} = \vec{b} - A\left(\vec{x}^{(k)} + \vec{v}^{(k)}\right) \\ &= \vec{b} - A\vec{x}^{(k)} - A\vec{v}^{(k)} = \vec{d}^{(k)} - A\vec{v}^{(k)}\end{aligned}$$

$\vec{d}^{(k+1)} = \vec{d}^{(k)} - A\vec{v}^{(k)}$ is therefore an equivalent reformulation which reduces the cancelation errors.

Defect Formulation

The iteration scheme can also be reformulated in terms of the defect and the correction:

$$x_i^{(k+1)} = (1-\omega)\, x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}\right)$$

$$x_i^{(k+1)} - x_i^{(k)} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j\geq i} a_{ij} x_j^{(k)}\right)$$

$$v_i^{(k)} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}\left(x_j^{(k)} + v_j^{(k)}\right) - \sum_{j\geq i} a_{ij} x_j^{(k)}\right)$$

$$v_i^{(k)} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j} a_{ij} x_j^{(k)} - \sum_{j<i} a_{ij} v_j^{(k)}\right)$$

$$v_i^{(k)} = \frac{\omega}{a_{ii}}\left(d_i^{(k)} - \sum_{j<i} a_{ij} v_j^{(k)}\right)$$

Relaxation Methods in Defect Formulation

Damped Richardson Iteration:

$$v_i^{(k)} = \omega d_i^{(k)}$$

Damped Jacobi Iteration:

$$v_i^{(k)} = \frac{\omega}{a_{ii}} d_i^{(k)}$$

special case: $\omega = 1 \Rightarrow$ Jacobi Iteration

SOR (successive overrelaxation) Iteration:

$$v_i^{(k)} = \frac{\omega}{a_{ii}}\left(d_i^{(k)} - \sum_{j<i} a_{ij} v_j^{(k)}\right)$$

$0 < \omega < 1$: underrelaxation $1 < \omega < 2$: overrelaxation special case: $\omega = 1 \Rightarrow$ Gauß-Seidel Iteration

Example Algorithm

The initial guess $\vec{x}$, the matrix A and the right side $\vec{b}$ are given.

$$
\begin{aligned}
&\vec{d} = \vec{b} - A\vec{x};\\
&d_0 = ||\vec{d}||;\\
&d_k = d_0;\\
&\text{while } (d_k \geq \varepsilon \cdot d_0)\\
&\{\\
&\quad \text{Solve } M \cdot \vec{v} = \vec{d}\\
&\quad \vec{x} = \vec{x} + \vec{v};\\
&\quad \vec{d} = \vec{d} - A\vec{v};\\
&\quad d_k = ||\vec{d}||;\\
&\}
\end{aligned}
$$

Data Structures for Sparse Matrices

To save memory A should not be stored as ordinary two-dimensional array.
One of the alternatives is called "compressed row storage" (CRS).
If $A \in \mathbb{R}^{N \times N}$ and s with $N < s < N^2$ is the total number of non-zero elements of A.

- All non-zero elements are stored line by line in a one-dimensional floating-point array `a` of size s.
- The corresponding column indices are stored line by line in a one-dimensional integer array `j` of size s.
- The start indices of each line are stored in an one-dimensional integer array `r` of size $N + 1$, where the total number of non-zero elements s is stored as last element of `r` (`r[N]=s`).

Example Matrix

$$
A = \begin{pmatrix}
2.1 & 0 & 3.4 & 0 & 0 \\
0 & 1.3 & 0 & 2 & 6.4 \\
1.1 & 0 & 5.3 & 0 & 0 \\
0 & 7.8 & 0 & 3.9 & 2.3 \\
5.8 & 0 & 0 & 3.1 & 6
\end{pmatrix}
$$

$$
\begin{aligned}
a &= \{2.1, 3.4, 1.3, 2, 6.4, 1.1, 5.3, 7.8, 3.9, 2.3, 5.8, 3.1, 6\}\\
j &= \{0, 2, 1, 3, 4, 0, 2, 1, 3, 4, 0, 3, 4\}\\
r &= \{0, 2, 5, 7, 10, 13\}
\end{aligned}
$$

Memory consumption: if `double` arrays are used for the floating point variables and `int` for the integer arrays: 200 bytes for storing the full matrix, 180 bytes for the CRS matrix (The gain is much larger if the size of the matrix increases).

Access an Element in a CRS-Matrix

```
double &GetA(int row, int column)
{
    for(k=r[row];k<r[row+1];++k)
    {
        if (j[k]==column)
            return(a[k]);
    }
    return(0.);
}
```

Computing $y = A \times x$ for a CRS-Matrix

```
for (i=0;i<N;++i)
{
    y[i]=0.;
    for(k=r[i];k<r[i+1];++k)
        y[i] = y[i] + a[k] * x[j[k]];
}
```

Improved CRS

- Assume that diagonal element does always exist
- Store diagonal element at position `r[row]`
- Do not store diagonal index
- Store number of elements in the row at `j[r[row]]`

Advantages:

- The position of the diagonal element is always clear (necessary for relaxation methods)
- The structure of the matrix (sparsity pattern) can vary a bit

Smoothing Property of Linear Iterative Methods

We assume again that A is symmetric and positive definite. If $\vec{z}_k$ is an eigenvector of A:

$$A\vec{z}_k = \lambda_k \vec{z}_k$$

with $0 < \lambda_{\mathsf{min}} \leq \lambda_k \leq \lambda_{\mathsf{max}}$.

For Richardson's iteration with $\omega = 1/\lambda_{\mathsf{max}}$ and $\vec{e}^{(i)} = \vec{z}_k$ we obtain

$$\vec{e}^{(i+1)} = \left(I - \frac{1}{\lambda_{\mathsf{max}}} A\right) \vec{z}_k = \left(1 - \frac{\lambda_k}{\lambda_{\mathsf{max}}}\right) \vec{e}^{(i)}.$$

This means: $\begin{array}{ll} \lambda_k \text{ close to } \lambda_{\mathsf{max}} & \Rightarrow \left(1 - \frac{\lambda_k}{\lambda_{\mathsf{max}}}\right) \approx 0 \\ \lambda_k \text{ close to } \lambda_{\mathsf{min}} & \Rightarrow \left(1 - \frac{\lambda_k}{\lambda_{\mathsf{max}}}\right) \approx 1 \end{array}$

- Error components corresponding to *large* eigenvalues are damped efficiently.
- Error components corresponding to *small* eigenvalues are damped slowly.

For second order problems we have $\lambda_{\mathsf{min}}/\lambda_{\mathsf{max}} = O(h^2)$, i.e. the asymptotic convergence factor is

$$\rho = 1 - O(h^2).$$

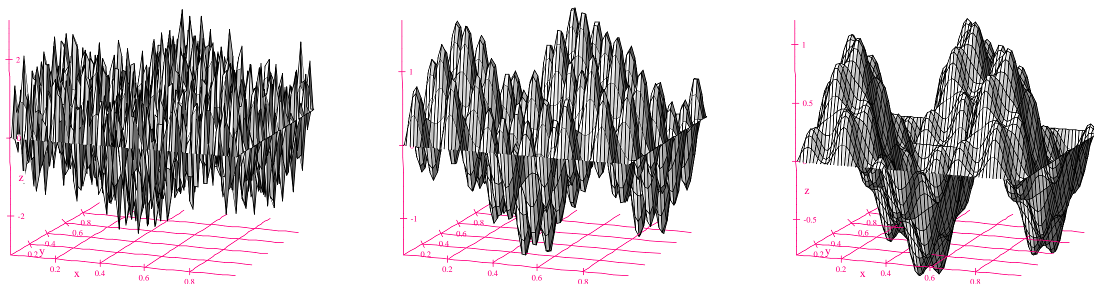
The (damped) Jacobi and Gauß–Seidel iteration have an asymptotically similar behavior in contrast to an optimally damped SOR. However, the optimal damping coefficient for SOR is hard to determine.

Error Smoothing Example

We discretize $-\Delta p = r$ with the cell-centered Finite-Volume method on a structured mesh.

The initial error consists of low and high frequency parts.

The graphs show the initial error and the error after 1 and 5 iterations.

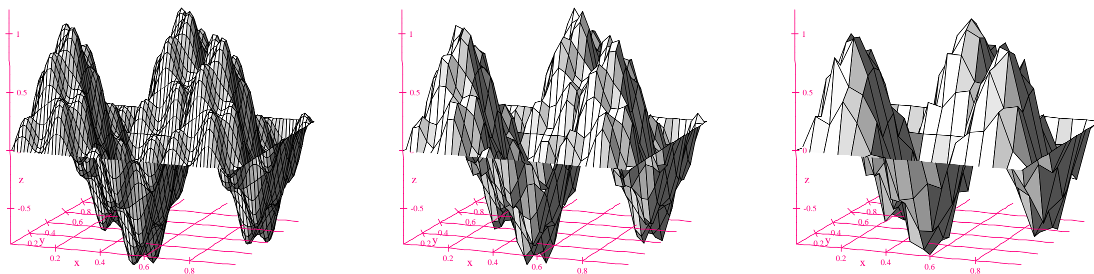


figures P. Bastian (personal communication)

Multigrid Idea

Construct an iteration that is complementary to the smoother reducing *low frequency errors*.

Idea: Low frequency errors can be represented on a coarser grid:

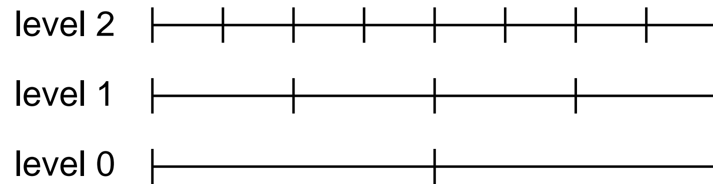


This requires a *hierarchy* of grids $\Omega_0, \Omega_1, \Omega_2, \ldots$

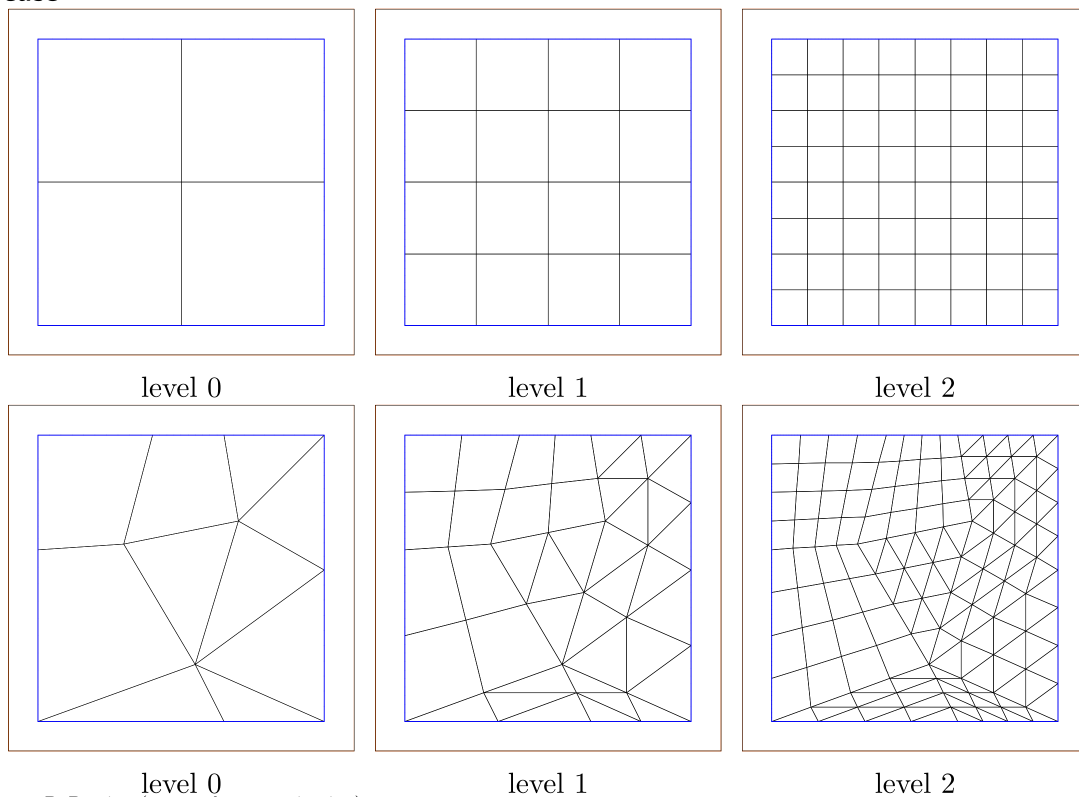
Correspondingly there will be a hierarchy of linear systems

$$A_l \vec{x}_l = \vec{b}_l$$

1D case



2D case



figures P. Bastian (personal communication)

Multigrid Algorithm

- (pre)smoothing of the fine grid solution $\vec{x}_l^{(k)}$ (usually with some steps of a damped Jacobi or Gauß-Seidel iteration)
- compute defect $\vec{d}_l^{(k)}$
- restrict defect $\vec{d}_l^{(k)}$ to coarse grid $\vec{d}_{l-1}^{(k)}$ (either by just using the values at the grid points of the coarse grid or by averaging of fine grid values)

- compute solution $\vec{v}_{l-1}^{(k)}$ of $A_{l-1}\vec{v}_{l-1}^{(k)} = \vec{d}_{l-1}^{(k)}$ (with direct solution, relaxation methods or another coarse grid correction ⇒ multigrid method)
- prolongate $\vec{v}_{l-1}^{(k)}$ to the fine grid Ω_l (interpolate $\vec{v}_l^{(k)}$ at the fine grid points)
- update fine grid solution $\vec{x}_l^{(k+1)} = \vec{x}_l^{(k)} + \vec{v}_l^{(k)}$
- sometimes (post)smoothing of the fine grid solution $\vec{x}_l^{(k+1)}$ (usually with some steps of a damped Jacobi or Gauß-Seidel iteration)

Multigrid Methods

Multigrid methods

- have a overall work, which is still dominated by the finest grid. If C operations are necessary on the fine grid only $C/4$ operations in 2D and $C/8$ operations in 3D are necessary on the next coarser grid ...
- have a optimal complexity of $O(N)$ to solve $Ax = b$ for appropriate matrices (compared to $O(N^2)$ to $O(N^{7/3})$ with Gaussian elimination for banded matrices)
- there are also "Algebraic Multigrid" (AMG) solvers, which do not really construct a coarse grid, but use empirical schemes to generate coarser matrices from the fine-scale matrix. They have a complexity of $O(N \cdot ln(N))$.

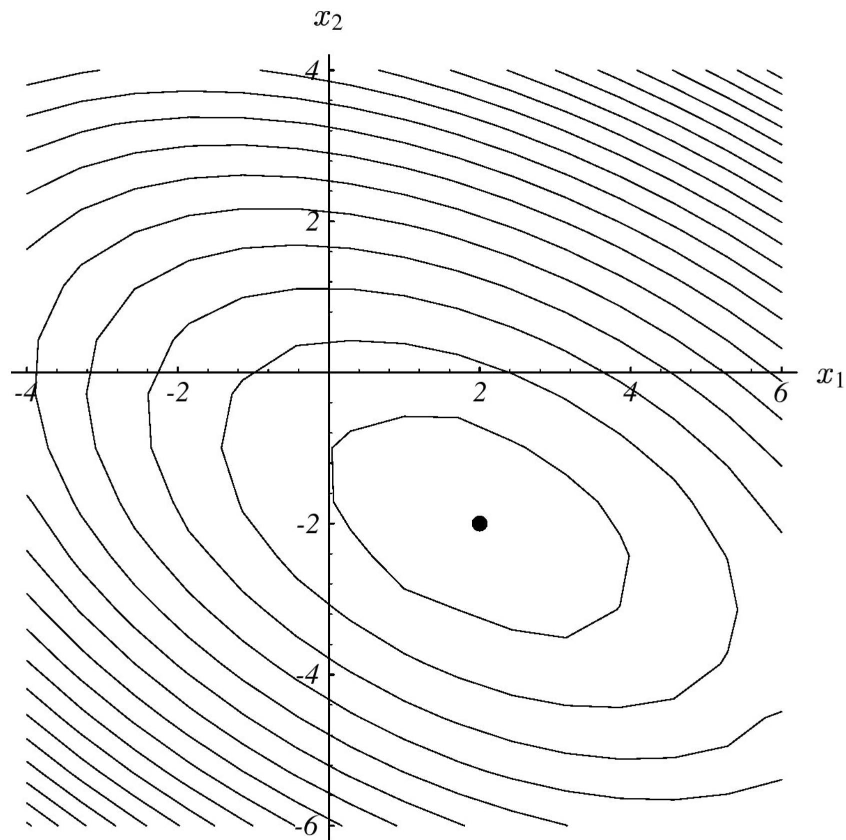
Gradient based iterative methods

If A is symmetric and positive definite then $\vec{x}^T A\vec{x} > 0 \ \forall \vec{x} \neq 0$. Then $A\vec{x} = \vec{b}$ is equivalent to finding the minimum of the quadratic form

$$f(x) := \frac{1}{2}\vec{x}^T A\vec{x} - \vec{b}^T\vec{x} + c$$

where $c \in \mathbb{R}$ is an arbitrary scalar. As A is positive definite, the hypersurface defined by $f(\vec{x})$ forms a paraboloid in $\mathbb{R}^{N+1}$. The minimum $\vec{x}$ is unique and global.

Different gradient based methods depend on the strategy to find this minimum.



from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

Proof of Correspondence

The gradient of $f(\vec{x})$ is

$$f'(\vec{x}) := \frac{1}{2}A^T\vec{x} + \frac{1}{2}A\vec{x} - \vec{b}$$

for symmetric matrices this reduces to

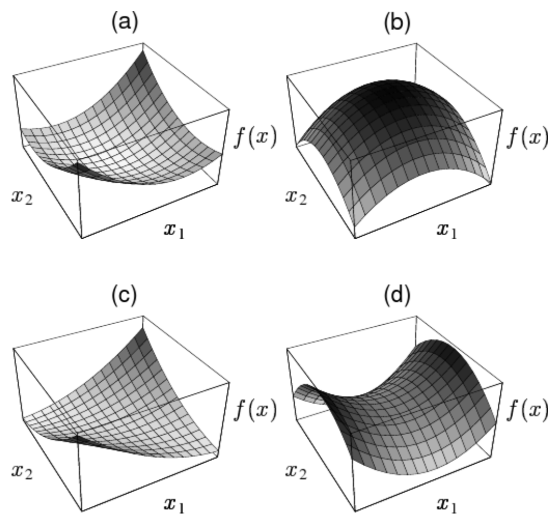
$$f'(\vec{x}) := A\vec{x} - \vec{b}$$

At the minimum the gradient vanishes

$$f'(\vec{x}) := A\vec{x} - \vec{b} = 0$$

Therefore $\vec{x}$ at the minimum solves $A\vec{x} - \vec{b}$

Shape of the quadratic form $f(\vec{x})$



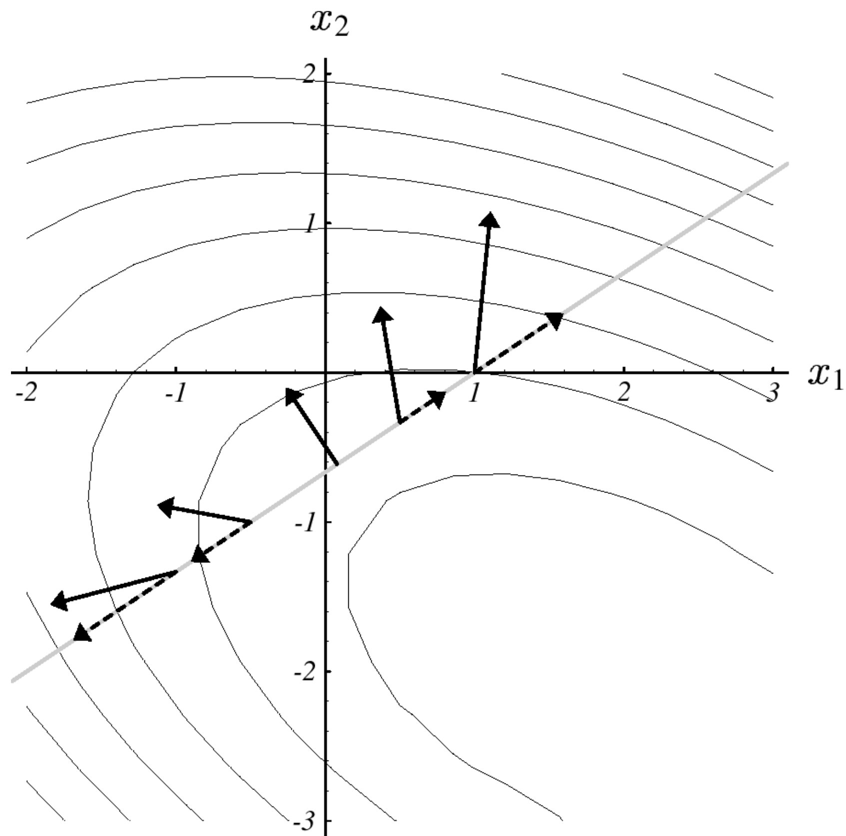
Quadratic form $f(\vec{x})$ for

- (a) a positive-definite matrix
- (b) a negative-definite matrix
- (c) a singular (and positive-definite) matrix
- (d) an indefinite matrix

from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

Method of Steepest Descent

Steepest Descend uses the direction of the negative gradient $-f'(x^{(k)})$.
The improved solution is calculated from $x^{(k+1)} = x^{(k)} - \alpha f'(x^{(k)})$.
The optimal step width α is chosen such that the minimum along the search direction is obtained. This results in the next descend being orthogonal to the search direction.



from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

Optimal step size α

$$f'(\vec{x}^{(k)}) = A\vec{x}^{(k)} - \vec{b} = -\left(\vec{b} - A\vec{x}^{(k)}\right) = -\vec{d}^{(k)}$$
$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \alpha\vec{d}^{(k)}$$

We want to find a minimum along the search direction:

$$\frac{d}{d\alpha} f(\vec{x}^{(k+1)}) = 0$$
$$f'(\vec{x}^{(k+1)})^T \frac{d}{d\alpha}\vec{x}^{(k+1)} = f'(\vec{x}^{(k+1)})^T \vec{d}^{(k)} = 0$$

with $f'(\vec{x}^{(k+1)}) = -\vec{d}^{(k+1)}$:

$$\vec{d}^{(k+1)^T} \vec{d}^{(k)} \quad = \quad 0$$

$$
\begin{aligned}
\vec{d}^{(k+1)^T}\vec{d}^{(k)} &= 0 \\
\left(b - A\vec{x}^{(k+1)}\right)^T \vec{d}^{(k)} &= 0 \\
\left(b - A\left(\vec{x}^{(k)} + \alpha\vec{d}^{(k)}\right)\right)^T \vec{d}^{(k)} &= 0 \\
\left(b - A\vec{x}^{(k)}\right)^T \vec{d}^{(k)} - \alpha\left(A\vec{d}^{(k)}\right)^T \vec{d}^{(k)} &= 0 \\
\alpha\left(A\vec{d}^{(k)}\right)^T \vec{d}^{(k)} &= \left(b - A\vec{x}^{(k)}\right)^T \vec{d}^{(k)} \\
\alpha\vec{d}^{(k)^T} A^T \vec{d}^{(k)} &= \vec{d}^{(k)^T}\vec{d}^{(k)} \\
\alpha &= \frac{\vec{d}^{(k)^T}\vec{d}^{(k)}}{\vec{d}^{(k)^T} A\vec{d}^{(k)}}
\end{aligned}
$$

Steepest Descent Algorithm

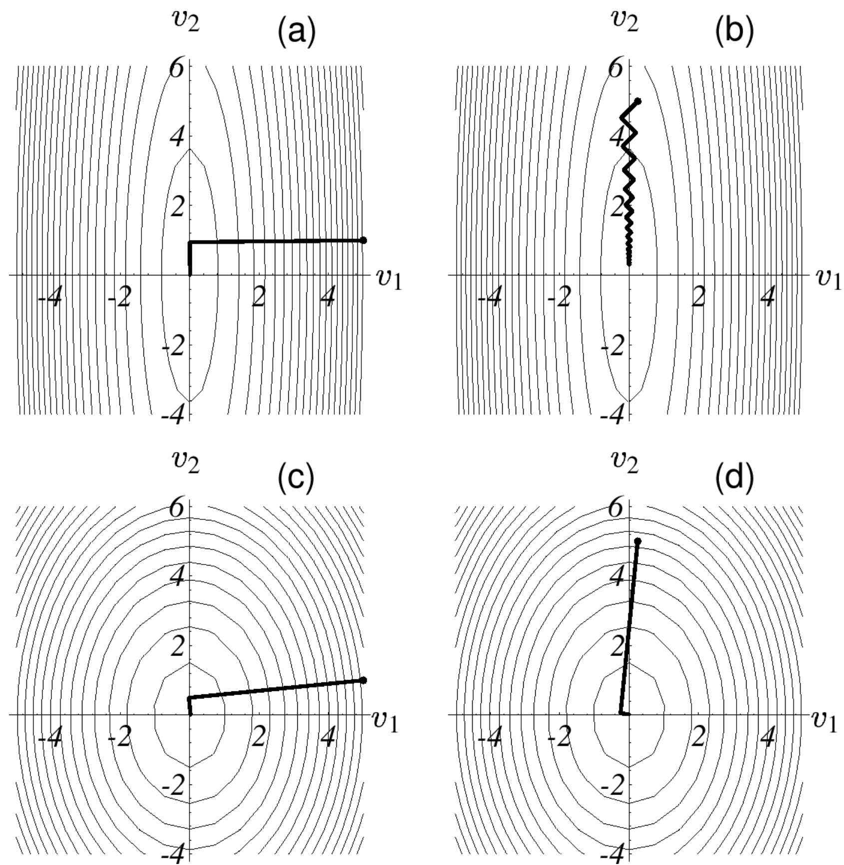
$\vec{d} = \vec{b} - A\vec{x}$
$d_0 = \vec{d}^T\vec{d}$
$d_k = d_0;$
while $(d_k \geq \varepsilon^2 \cdot d_0)$
$\{$
  $\alpha = \left(\vec{d}^T\vec{d}\right) / \left(\vec{d}^T A\vec{d}\right)$
  $\vec{x} = \vec{x} + \alpha\vec{d}$
  $\vec{d} = \vec{d} - \alpha A\vec{d}$
  $d_k = \vec{d}^T\vec{d}$
$\}$

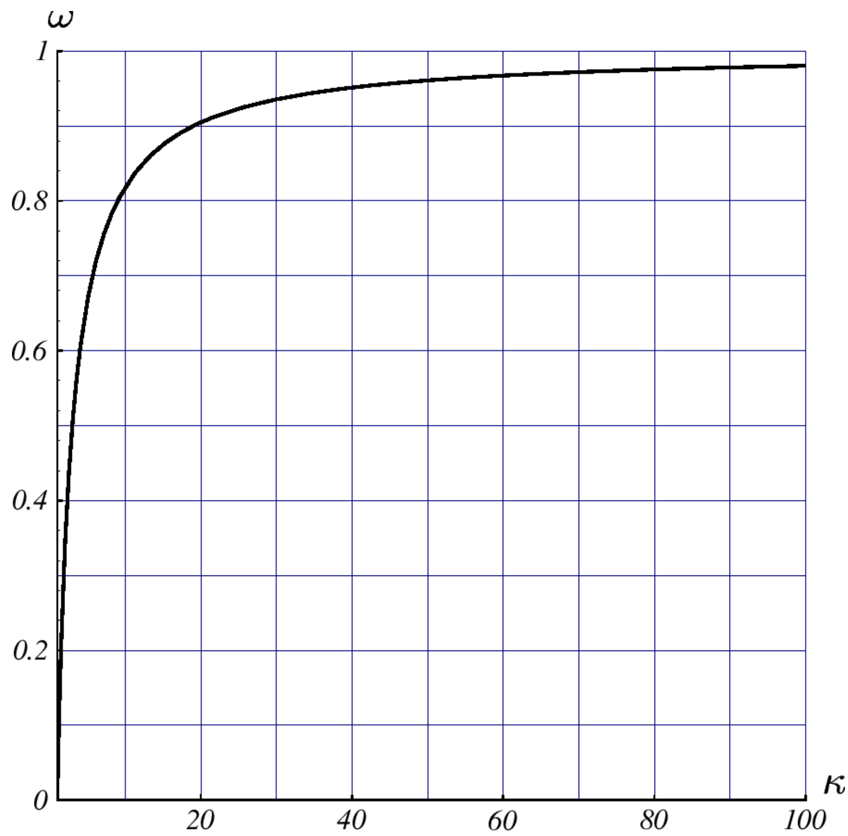
Optimized Steepest Descent Algorithm

$\vec{d} = \vec{b} - A\vec{x}$
$d_0 = \vec{d}^T\vec{d}$
$d_k = d_0;$
while $(d_k \geq \varepsilon^2 \cdot d_0)$
$\{$
  $\vec{t} = A\vec{d}$
  $\alpha = d_k / \left(\vec{d}^T\vec{t}\right)$
  $\vec{x} = \vec{x} + \alpha\vec{d}$
  $\vec{d} = \vec{d} - \alpha\vec{t}$
  $d_k = \vec{d}^T\vec{d}$
$\}$

Convergence of Steepest Descent

Convergence of steepest descend depends strongly on the matrix condition $\kappa(A)$ and on the initial value. Convergence is reduced by the fact that achievements of previous steps can be lost again in later steps.





from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

Convergence Rate

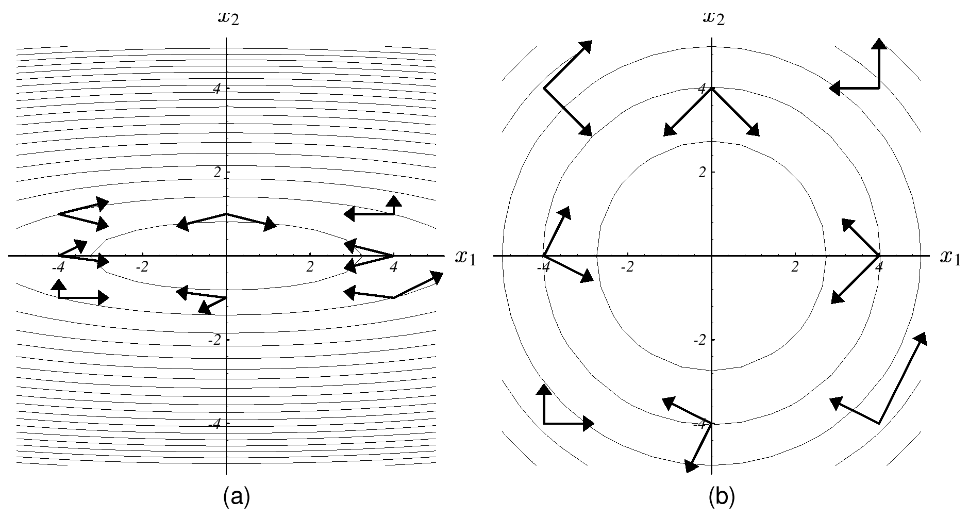
$$||\vec{e}^{(k)}||_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^k ||\vec{e}^{(0)}||_A$$

with the "energy norm"

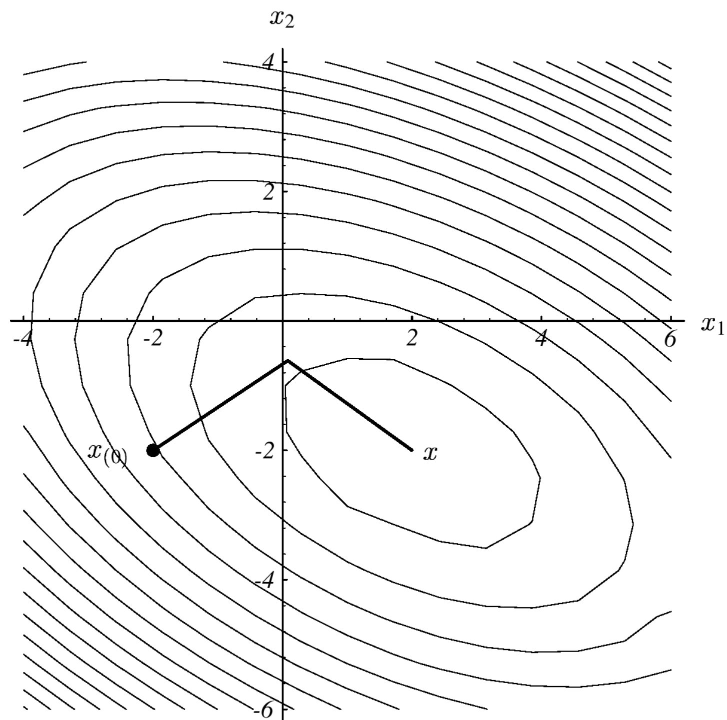
$$||\vec{e}||_A = \sqrt{\vec{e}^T A \vec{e}}$$

Conjugate Gradients (CG)

The Conjugate Gradient method uses a sequence of search directions, where each new search direction is A-orthogonal to *all* previous search directions, i.e. $\vec{v}_{(i)}^T A v_{(j)} = 0$ if $i \neq j$.



In exact arithmetic the minimum is found after at most N iterations (semi-iterative method). However round-off errors make CG to a iterative method.



from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

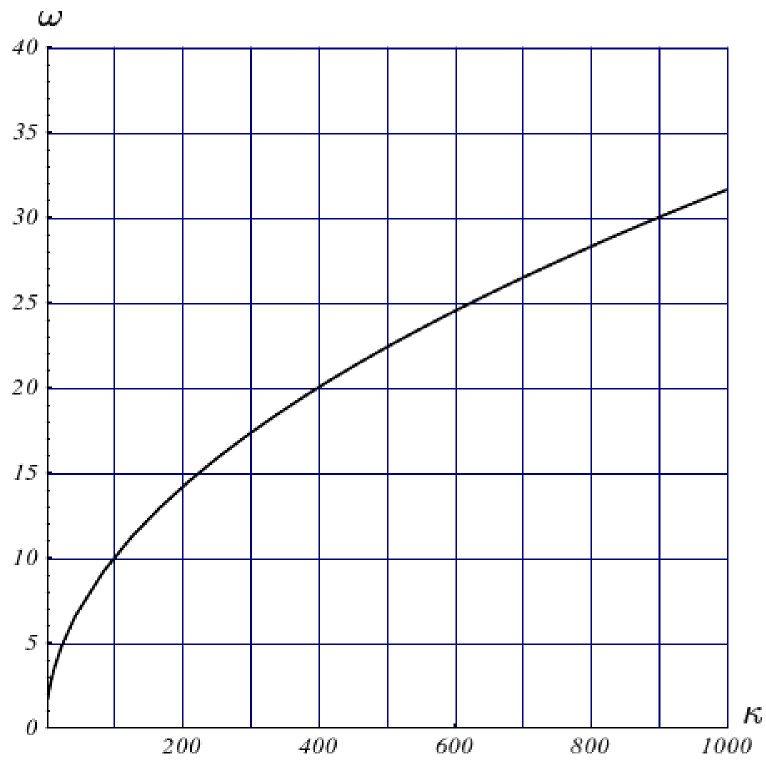
CG Algorithm

$$
\begin{aligned}
&\vec{v} = \vec{d} = \vec{b} - A\vec{x} \\
&d_0 = \vec{d}^T\vec{d} \\
&d_k = d_0; \\
&\text{while } (d_k \geq \varepsilon^2 \cdot d_0) \\
&\{ \\
&\quad \alpha = \left(\vec{d}^T\vec{d}\right) / \left(\vec{v}^T A\vec{v}\right) \\
&\quad \vec{x} = \vec{x} + \alpha\vec{v} \\
&\quad \vec{d}_{\text{new}} = \vec{d} - \alpha A\vec{v} \\
&\quad \beta = \left(\vec{d}_{\text{new}}^T\vec{d}_{\text{new}}\right) / \left(\vec{d}^T\vec{d}\right) \\
&\quad \vec{v} = \vec{d}_{\text{new}} + \beta\vec{v} \\
&\quad \vec{d} = \vec{d}_{\text{new}} \\
&\quad d_k = \vec{d}^T\vec{d} \\
&\}
\end{aligned}
$$

Optimized CG Algorithm

$$
\begin{aligned}
&\vec{v} = \vec{d} = \vec{b} - A\vec{x} \\
&d_0 = \vec{d}^T\vec{d}; \\
&d_k = d_0 \\
&\text{while } (d_k \geq \varepsilon^2 \cdot d_0) \\
&\{ \\
&\quad \vec{t} = A\vec{v} \\
&\quad \alpha = d_k / \left(\vec{v}^T\vec{t}\right) \\
&\quad \vec{x} = \vec{x} + \alpha\vec{v} \\
&\quad \vec{d} = \vec{d} - \alpha\vec{t} \\
&\quad d_{k_{\text{old}}} = d_k; \\
&\quad d_k = \vec{d}^T\vec{d} \\
&\quad \beta = d_k / d_{k_{\text{old}}} \\
&\quad v = \vec{d} + \beta\vec{v} \\
&\}
\end{aligned}
$$

Convergence of Conjugate Gradients versus Steepest Descent

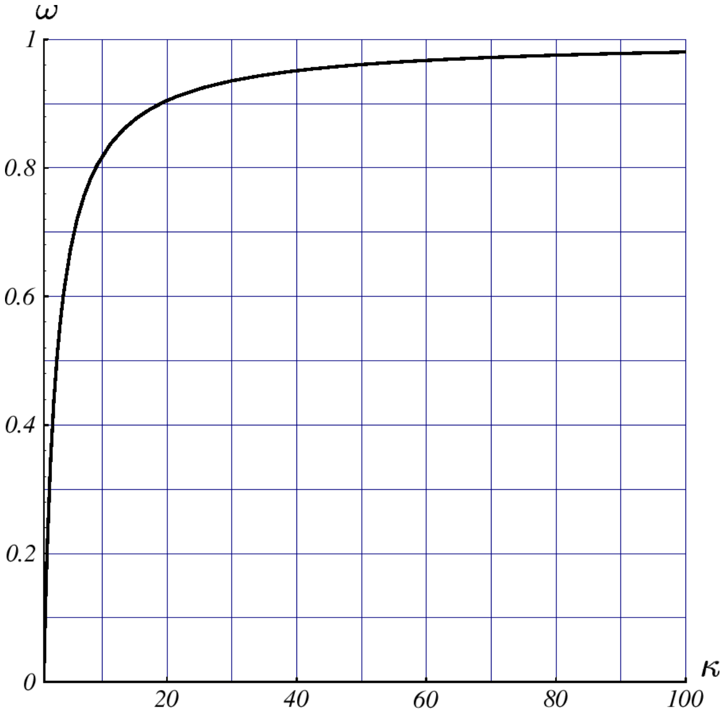


figures from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

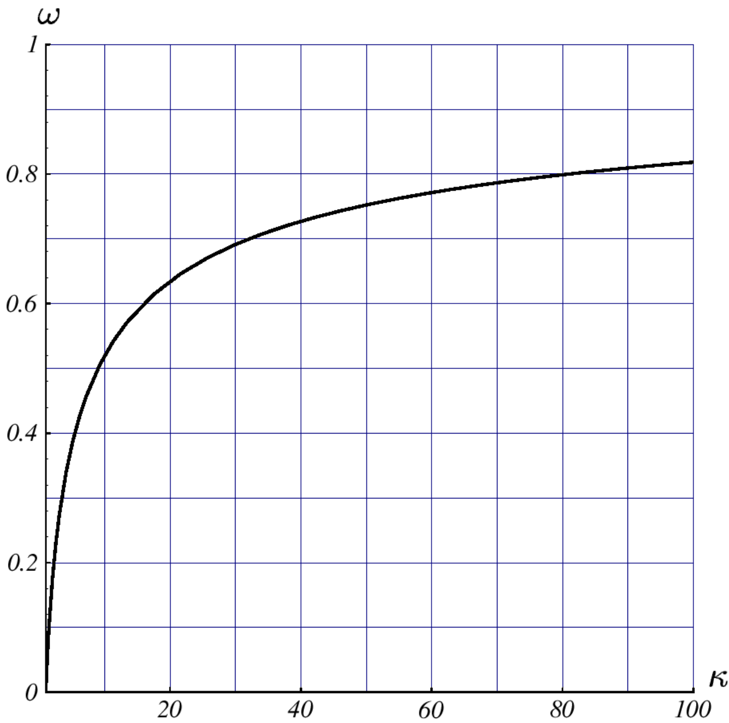
Convergence of Conjugate Gradients

Convergence depends on the condition $\kappa(A)$ of the matrix, but less than in steepest descend. It also depends on the distribution of eigenvalues.

Steepest Descent



Conjugate Gradients



Complexity for discretizations of second-order elliptic PDE's

	two-dimensional	three-dimensional
Steepest Descent	$O(N^2)$	$O(N^{3/2})$
Conjugate Gradients	$O(N^{5/3})$	$O(N^{4/3})$

figures from J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"

Convergence Rate

Steepest Descent

$$||\vec{e}^{(k)}||_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^k ||\vec{e}^{(0)}||_A$$

Conjugate Gradients

$$||\vec{e}^{(k)}||_A \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k ||\vec{e}^{(0)}||_A$$

with the "energy norm"

$$||\vec{e}||_A = \sqrt{\vec{e}^T A \vec{e}}$$

Preconditioning

- While *CG*-methods usually have a better convergence than simple relaxation methods, the convergence still depends on the grid size for matrices generated by discretizations of partial differential equations.
- *CG*-methods therefore are often improved by using so-called preconditioning.
- Instead of $Ax = b$ we solve a system $M^{-1}Ax = M^{-1}b$, where the preconditioner M improves the distribution of eigenvalues or the condition of the matrix and thus provides an improved convergence behavior.

Preconditioners

- A^{-1} would be the optimal preconditioner as the eigenvalues of the resulting identity matrix I would all be identical and thus the system could be solved in one step, but it is of course too expensive to calculate.
- A simple possible choice is $M = D$ (so-called Jacobi preconditioning).
- The best choice is often a multigrid scheme for the coarse grid corrections.
- As the CG-method requires symmetric matrices, the SOR scheme can not be used. However, there is a variant called SSOR (symmetric SOR) which consists of a SOR step followed by a backward SOR step where we start with the last unknown and then decrement the indices. $M^{T^{-1}} M^{-1} A x = M^{T^{-1}} M^{-1} b$

Preconditioned CG Algorithm

$$
\begin{aligned}
&\vec{d} = \vec{b} - A\vec{x}\\
&\text{solve } M\vec{v} = \vec{d}\\
&\rho_k = \rho_0 = \vec{d}^T\vec{s}\\
&\text{while } (\rho_k \geq \varepsilon^2 \cdot \rho_0)\\
&\{\\
&\quad \vec{t} = A\vec{v}\\
&\quad \alpha = \rho_k / \left(\vec{v}^T\vec{t}\right)\\
&\quad \vec{x} = \vec{x} + \alpha\vec{v}\\
&\quad \vec{d} = \vec{d} - \alpha\vec{t}\\
&\quad \text{solve } M\vec{s} = \vec{d}\\
&\quad \rho_{k_{\text{old}}} = \rho_k\\
&\quad \rho_k = \vec{d}^T\vec{s}\\
&\quad \beta = \rho_k / \rho_{k_{\text{old}}}\\
&\quad \vec{v} = \vec{s} + \beta\vec{v}\\
&\}
\end{aligned}
$$

SSOR-Preconditioner

For the SSOR-preconditioner the step solve $M\vec{v} = \vec{d}$ is:

$$
\begin{aligned}
&\text{for } (i = 0; i < n; {+}{+}i)\\
&\quad v_i = \omega \left(d_i - \sum_{j<i} a_{ij} v_i\right) / a_{ii}\\
&\vec{d} = \vec{d} - A\vec{v}\\
&\text{for } (i = n-1; i >= 0; {-}{-}i)\\
&\quad vv_i = \omega \left(d_i - \sum_{j>i} a_{ij} vv_i\right) / a_{ii}\\
&\vec{v} {+}{=} \vec{vv}
\end{aligned}
$$

More information on gradient based methods can be found in

J. R. Shewchuk (1994): "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain"
http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

6 Parallel Computing

6.1 Introduction

The next four lectures give a basic introduction to the subject.
At the end you should have acquired:

- A basic understanding of different parallel computer architectures.

- Know how to write programs using OpenMP.
- Know how to write programs using message passing.
- Knowledge of some parallel algorithms for the iterative solution of linear systems.
- Know how to evaluate the quality of a parallel algorithm and its implementation.

6.1.1 Why Parallel Computing ?

Parallel Computing is Ubiquitous

- Multi-Tasking
  - Several independent computations ("threads of control") can be run quasi-simultaneously on a single processor (time slicing).
  - Developed since 1960s to increase throughput.
  - Interactive applications require "to do many things in parallel".
  - "Hyperthreading" does the simulation in hardware.
  - All relevant coordination problems are already present.
- Distributed Computing
  - Computation is inherently distributed because the information is distributed.
  - Example: Running a world-wide company or a bank.
  - Issues are: Communication across platforms, portability and security.
- High-Performance Computing
  - HPC is the driving force behind the development of computers.
  - All techniques implemented in today's desktop machines have been developed in supercomputers many years ago.
  - Applications run on supercomputers are mostly numerical simulations.
  - Grand Challenges: Cosmology, protein folding, prediction of earth-quakes, climate and ocean flows, ...
  - but also: nuclear weapon simulation
  - ASCI (Advanced Simulation and Computing) Program funding: $ 300 million in 2004.
  - Earth simulator (largest computer in the world from 2002 to 2004) cost about $ 500 million.

6.1.2 (Very Short) History of Supercomputers

What is a Supercomputer?

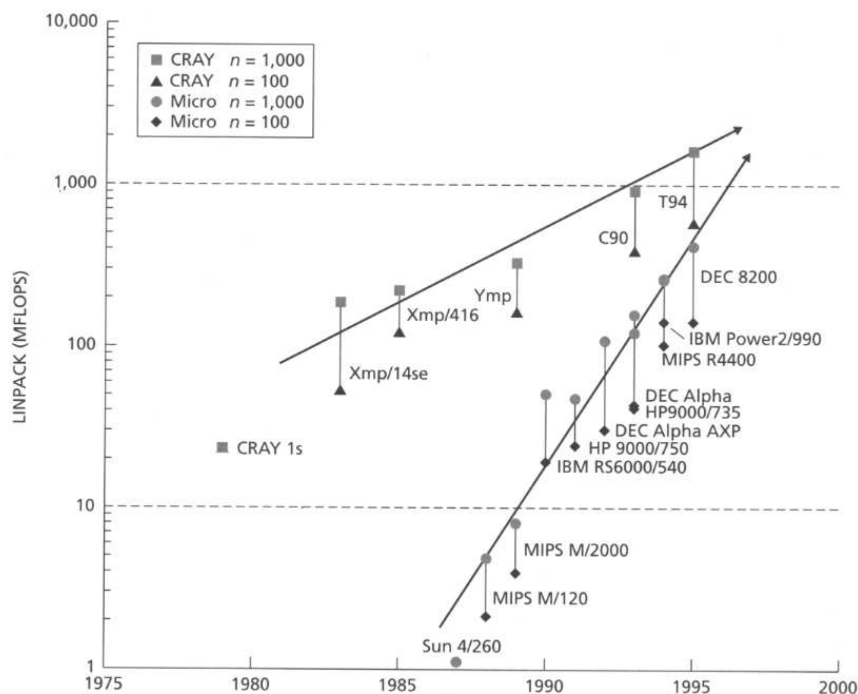
- A computer that costs more than $ 10 million

Computer	Year	$	MHz	MBytes	MFLOP/s
CDC 6600	1964	7M $	10	0.5	3.3
Cray 1A	1976	8M $	80	8	20
Cray X-MP/48	1986	15M $	118	64	220
C90	1996		250	2048	5000
ASCI Red	1997		220	$1.2{\cdot}10^6$	$2.4{\cdot}10^6$
Pentium 4	2002	1500	2400	1000	4800
Core 2 Duo	2007	299	2660	8000	12500

- Speed is measured in floating point operations per second (FLOP/s).
- Current supercomputers are large collections of microprocessors
- Today's desktop PC is yesterdays supercomputer.
- `www.top500.org` compiles list of supercomputers every six months.

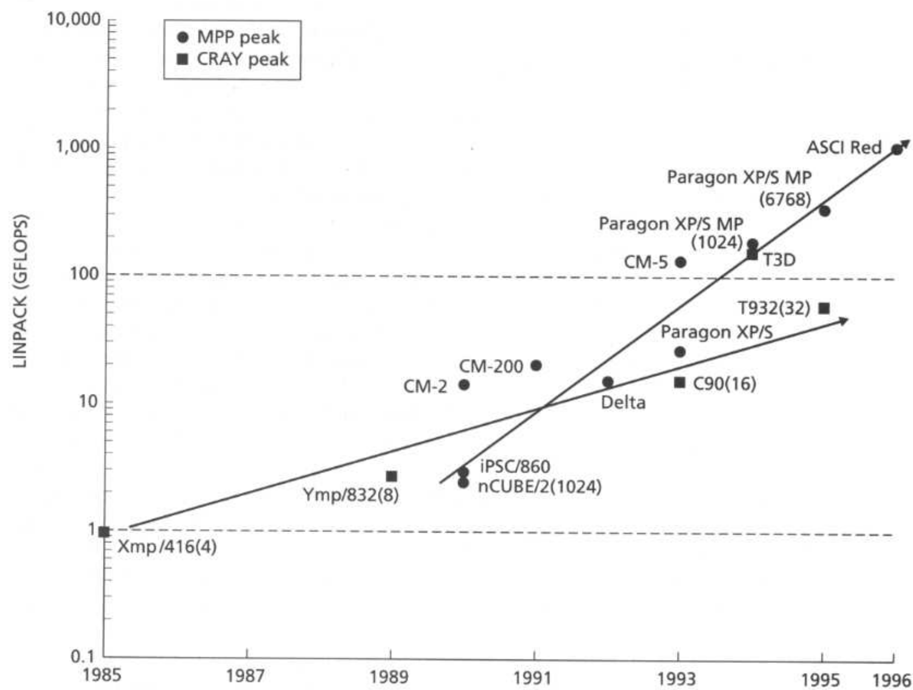
Development of Microprocessors

Microprocessors outperform conventional supercomputers in the mid 90s (from *Culler et al.*).



Development of Multiprocessors

Massively parallel machines outperform vector parallel machines in the early 90s (from *Culler et al.*).



TOP 500 November 2007

Rank	Site	Computer	Cores	Year	R_{max}	R_{peak}
1	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution IBM	212992	2007	478.20	596.38
2	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution IBM	65536	2007	167.30	222.82
3	SGI/New Mexico Computing Applications Center (NMCAC) United States	SGI Altix ICE 8200, Xeon quad core 3.0 GHz SGI	14336	2007	126.90	172.03
4	Computational Research Laboratories, TATA SONS India	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband Hewlett-Packard	14240	2007	117.90	170.88
5	Government Agency Sweden	Cluster Platform 3000 BL460c, Xeon 53xx 2.66GHz, Infiniband Hewlett-Packard	13728	2007	102.80	146.43
6	NNSA/Sandia National Laboratories United States	Red Storm - Sandia/ Cray Red Storm, Opteron 2.4 GHz dual core Cray Inc.	26569	2007	102.20	127.53
7	Oak Ridge National Laboratory United States	Jaguar - Cray XT4/XT3 Cray Inc.	23016	2006	101.70	119.35
8	IBM Thomas J. Watson Research Center United States	BGW - eServer Blue Gene Solution IBM	40960	2005	91.29	114.69
9	NERSC/LBNL United States	Franklin - Cray XT4, 2.6 GHz Cray Inc.	19320	2007	85.37	100.46
10	Stony Brook/BNL, New York Center for Computational Sciences United States	New York Blue - eServer Blue Gene Solution IBM	36864	2007	82.16	103.22

2. BlueGene/P "JUGENE" at FZ Jülich: 65536 processors, 167.3 TFLOP/s
30. Earth Simulator: 5120 processors, 35.9 TFLOP/s
202. SX8/576M72 at HWW/Stuttgart: 576 processors, 8.9 TFLOP/s

Terascale Simulation Facility

BlueGene/L prototype at Lawrence Livermore National Laboratory outperforms Earth Simulator in late 2004: 65536 processors, 136 TFLOP/s,

Final version at LLNL today: 212992 processors, 478 TFLOP/s

Efficient Algorithms are Important!

- Computation time for solving (certain) systems of linear equations on a computer with 1 GFLOP/s.

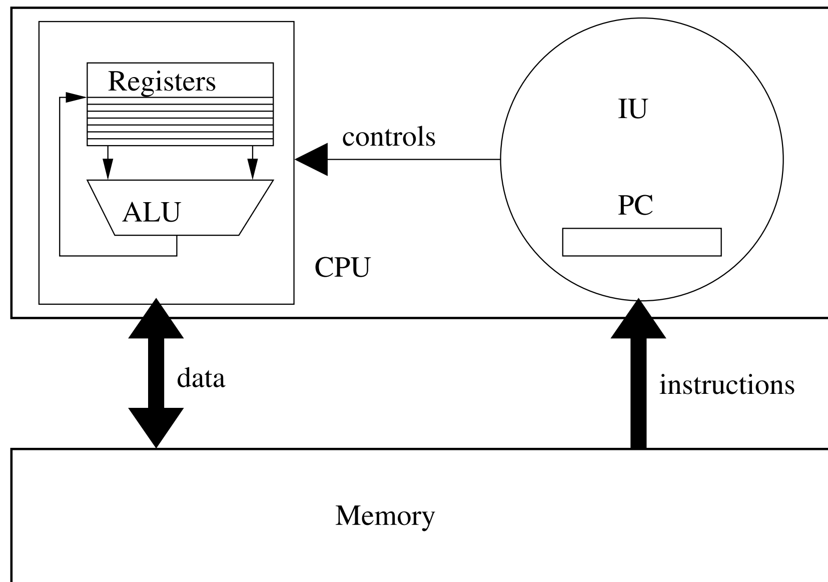
N	Gauß ($\frac{2}{3}N^3$)	Multigrid ($100N$)
1.000	0.66 s	10^{-4} s
10.000	660 s	10^{-3} s
100.000	7.6 d	10^{-2} s
$1 \cdot 10^6$	21 y	0.1 s
$1 \cdot 10^7$	21.000 y	1 s

- Parallelisation does not help an inefficient algorithm.
- We must parallelise algorithms with good sequential complexity.

6.2 Single Processor Architecture

6.2.1 Von Neumann Architecture

Von Neumann Computer



IU: Instruction unit

PC: Program counter

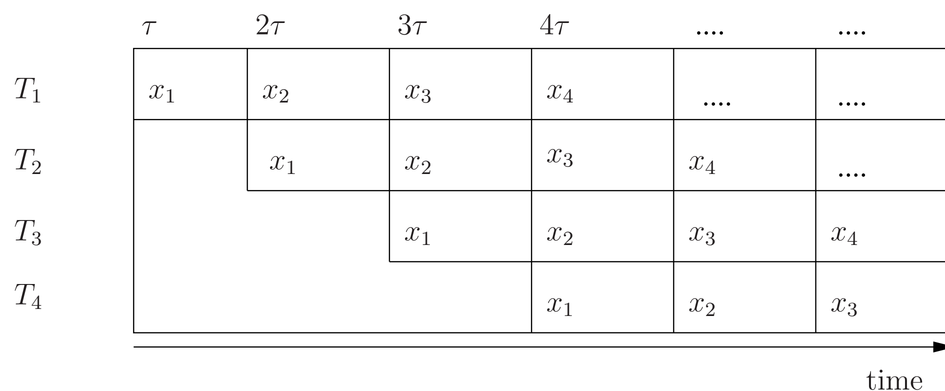
ALU: Arithmetic logic unit

CPU: Central processing unit

Single cycle architecture

6.2.2 Pipelining

Pipelining: General Principle



- Task T can be subdivided into m subtasks $T_1, \ldots, T_m$.
- Every subtask can be processed in the **same** time τ.
- All subtasks are **independent**.

- Time for processing N tasks:
  $T_S(N) = Nm\tau \quad T_P(N) = (m + N - 1)\tau$.
- Speedup
  $S(N) = \frac{T_S(N)}{T_P(N)} = m\frac{N}{m+N-1}$.

Arithmetic Pipelining

- Apply pipelining principle to floating point operations.
- Especially suited for "vector operations" like $s = x \cdot y$ or $x = y + z$ because of independence property.
- Hence the name "vector processor".
- Allows at most $m = 10 \ldots 20$.
- Vector processors typically have a very high memory bandwith.
- This is achieved with *interleaved* memory, which is pipelining applied to the memory subsystem.

Instruction Pipelining

- Apply pipelining principle to the processing of machine instructions.
- Typical subtasks are (m=5):
  - Instruction fetch.
  - Instruction decode.
  - Instruction execute.
  - Memory access.
  - Write back results to register file.
- Reduced instruction set computer (RISC): Use simple and homogeneous set of instructions to enable pipelining (e. g. load/store architecture).
- Conditional jumps pose problems and require some effort such as branch prediction units.
- Optimising compilers are also essential (instruction reordering, loop unrolling, etc.).

6.2.3 Superscalar Architecture

Superscalar Architecture

- Consider the statements

```
(1) a = b+c;
(2) d = e*f;
(3) g = a-d;
(4) h = i*j;
```

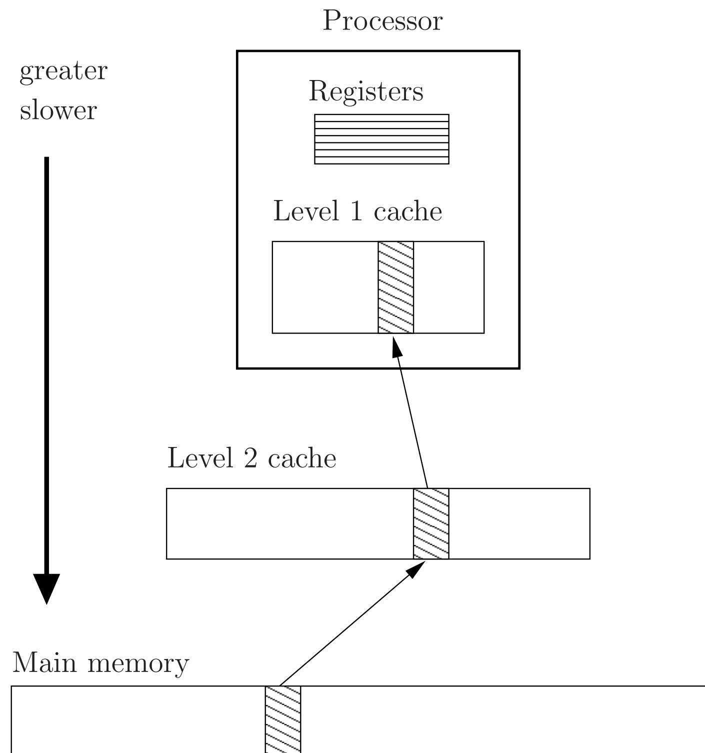
- Statements 1, 2 and 4 can be executed in parallel because they are independent.
- This requires
  - Ability to issue several instructions in one cycle.
  - Multiple functional units.
  - Out of order execution.
  - Speculative execution.
- A processor executing more than one instruction per cycle is called superscalar.
- Multiple issue is possible through:
  - A wide memory access reading two instructions.
  - Very long instruction words.
- Multiple functional units with out of order execution were implemented in the CDC 6600 in 1964.
- A degree of parallelism of $3 \ldots 5$ can be achieved.

6.2.4 Caches

Caches I

While the processing power increases with parallisation, the memory bandwidth usually does not

- Reading a 64-bit word from DRAM memory can cost up to 50 cycles.
- Building fast memory is possible but too expensive per bit for large memories.
- Hierarchical cache: Check if data is in the level l cache, if not ask the next higher level.
- Repeat until main memory is asked.
- Data is transferred in *cache lines* of $32 \ldots 128$ bytes (4 to 16 words).

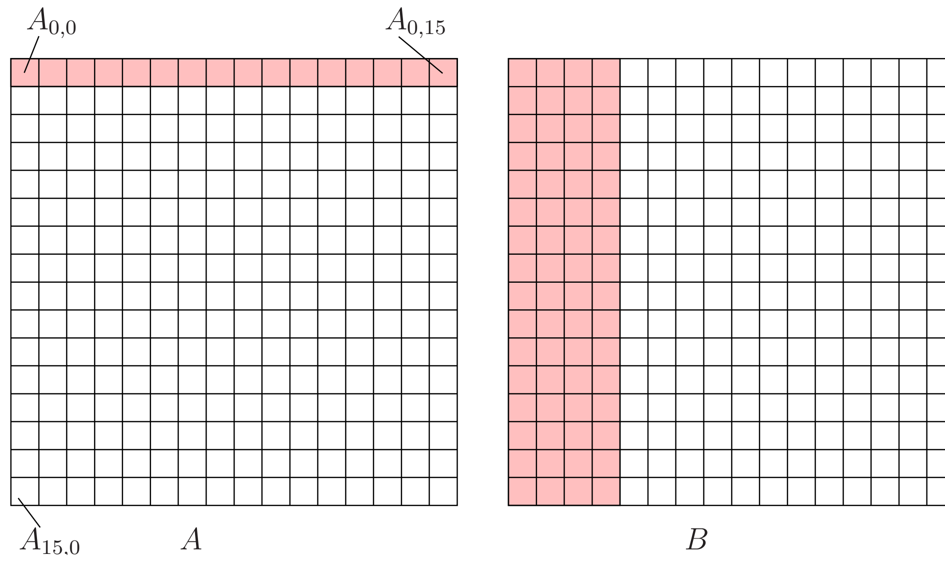


Caches II

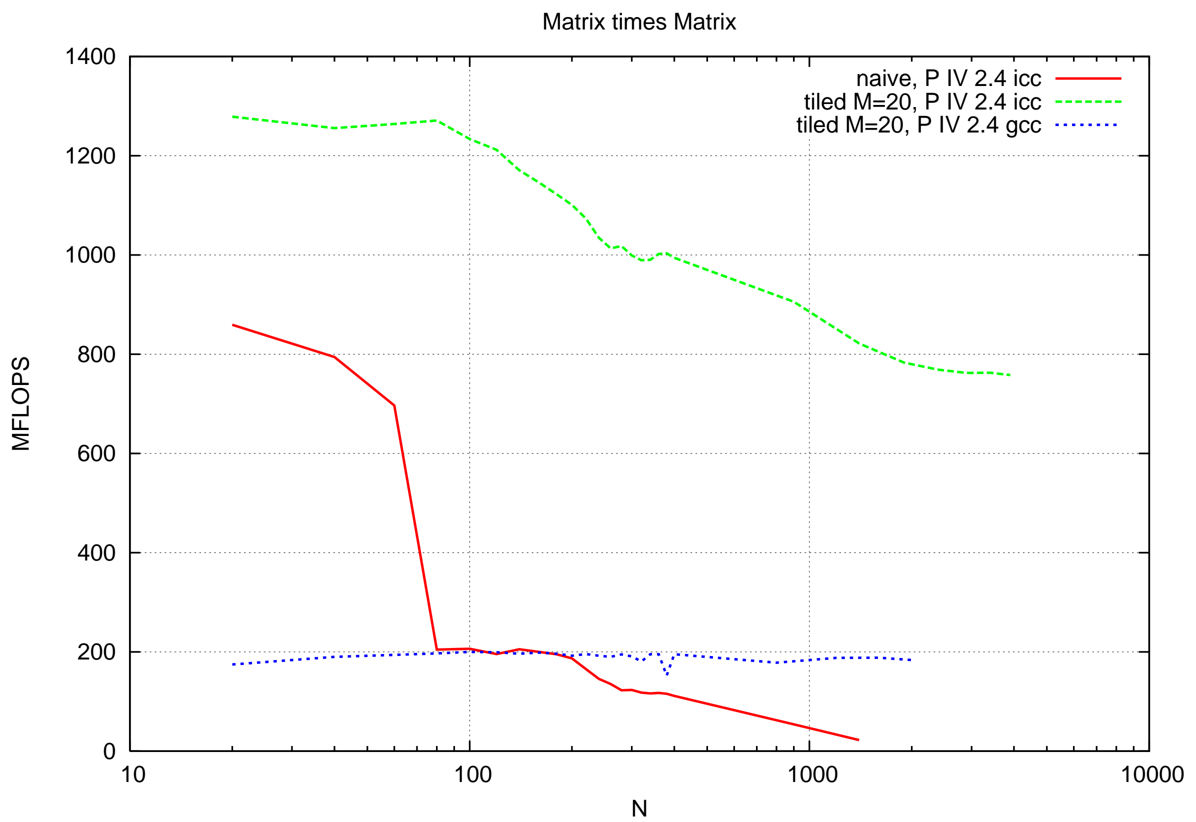
- Caches rely on *spatial and temporal locality.*
- There are four issues to be discussed in cache design:
  - *Placement*: Where does a block from main memory go in the cache? Direct mapped cache, associative cache.
  - *Identification*: How to find out if a block is already in cache?
  - *Replacement*: Which block is removed from a full cache?
  - *Write strategy*: How is write access handled? Write-through and write-back caches.
- Caches require to make code cache-aware. This is usually non-trivial and not done automatically.
- Caches can lead to a slow-down if data is accessed randomly and not reused.

Matrix Multiplication Example I

- Compute product of two matrices $C = AB$, i.e. $C_{ij} = \sum_{k=1}^{N} A_{ik}B_{kj}$
- Assume cache lines containing four numbers. C layout:



Matrix Multiplication Example II



6.3 Parallel Architectures

6.3.1 Classifications

Flynn's Classification (1972)

	Single data stream (One ALU)	Multiple data streams (Several ALUs)
Single instruction stream, (One IU)	**SISD**	**SIMD**
Multiple instruction streams (Several IUs)	—	**MIMD**

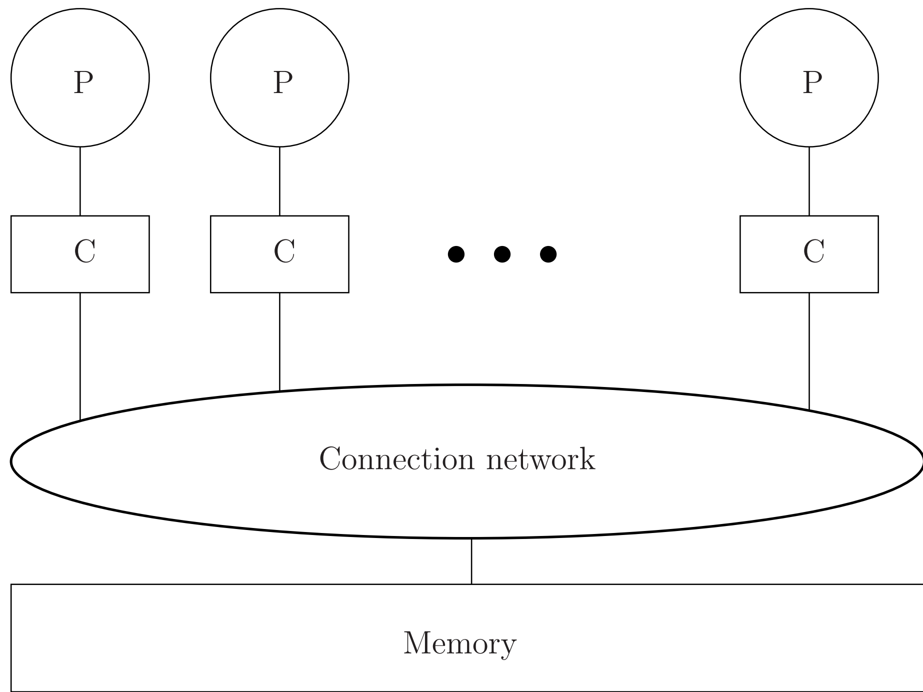
- **SIMD** machines allow the synchronous execution of one instruction on multiple ALUs. Important machines: ILLIAC IV, CM-2, MasPar.
- **MIMD** is the leading concept since the early 90s. All current supercomputers are of this type.

Classification by Memory Access

- Flynn's classification does not state *how* the individual components exchange data.
- There are only two basic concepts.
- **Communication via shared memory**. This means that all processors share a **global address space**. These machines are also called **multiprocessors**.
- **Communication via message exchange.** In this model every processor has its own **local address space**. These machines are also called **multicomputers**.

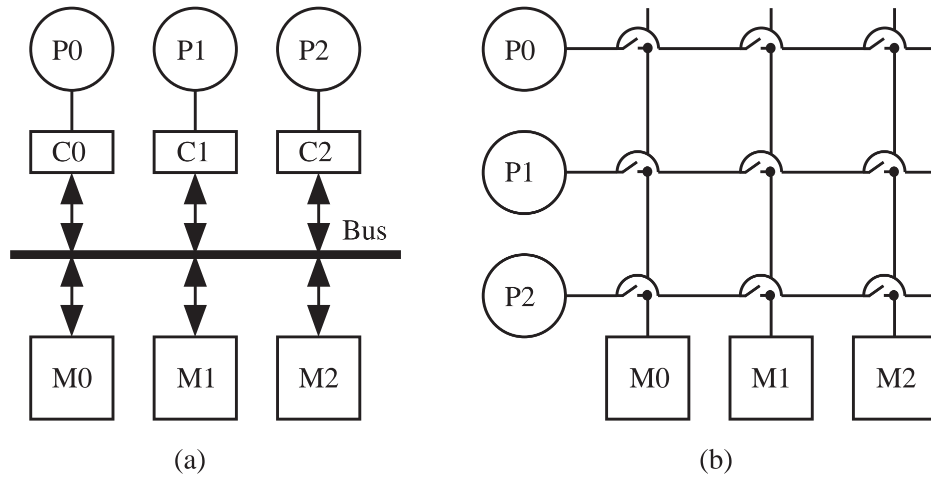
6.3.2 Uniform Memory Access Architecture

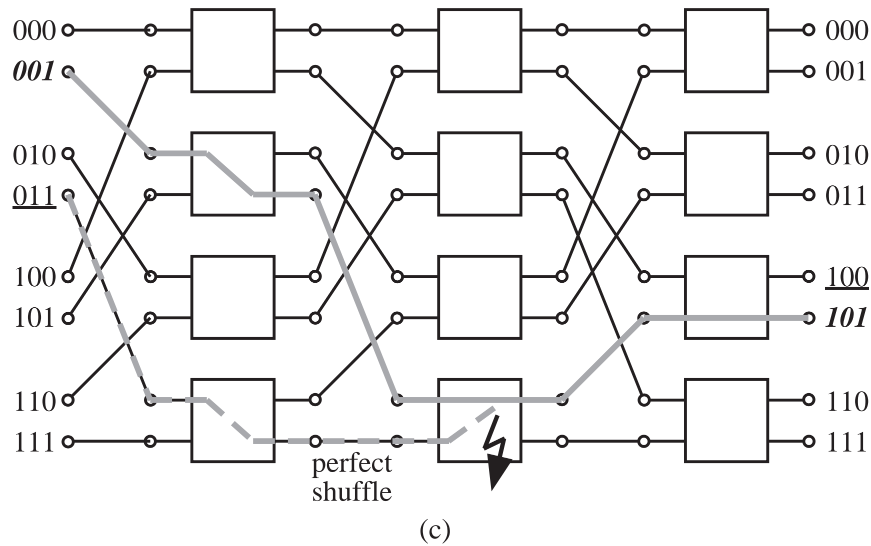
Uniform Memory Access Architecture



- UMA: Access to every memory location from every processor takes the same amount of time.
- This is achieved through a *dynamic network*.
- Caches serve two reasons: Provide fast memory access (migration) and remove traffic from network (replication).
- *Cache coherence problem:* Suppose one memory block is in two or more caches and is written by a processor. What to do now?

Dynamic Connection Networks



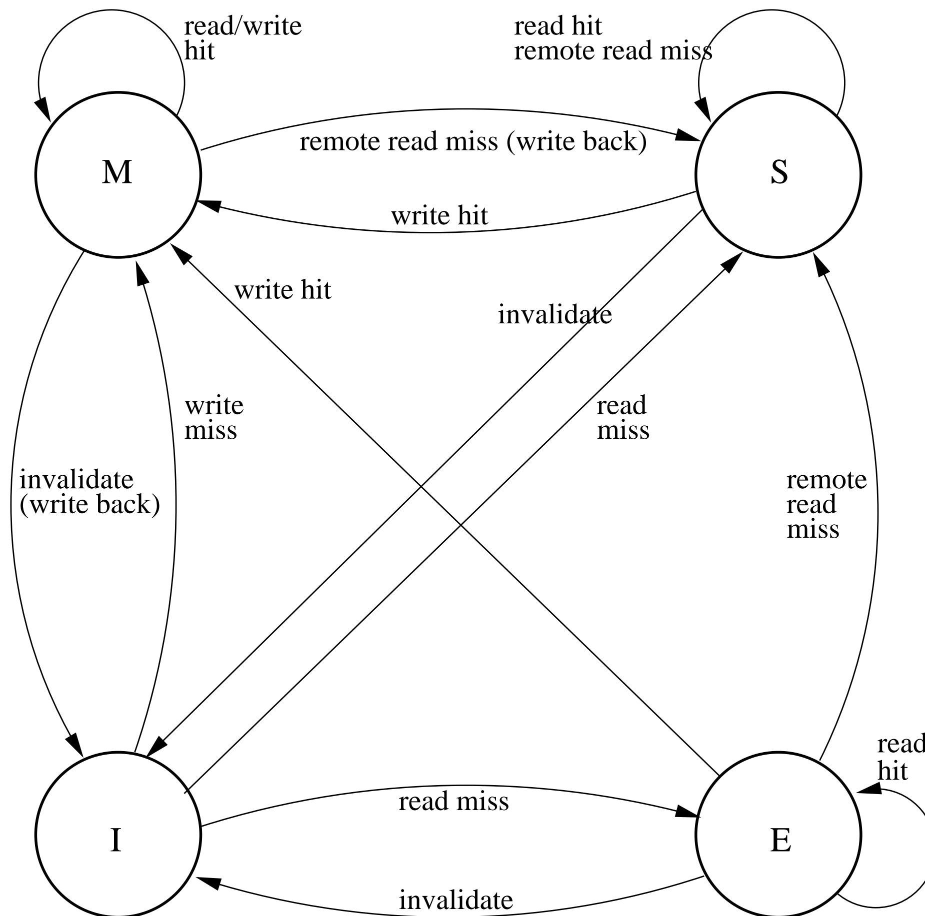


(a) *Bus*: Cheap, cache-coherence problem can be solved easily, not scalable.

(b) *Crossbar*: Expensive because of $O(P^2)$ resources, used in SUN servers, Earth Simulator, etc.

(c) Ω-*Network*: Compromise requiring $O(P \log P)$ resources, not used anymore.

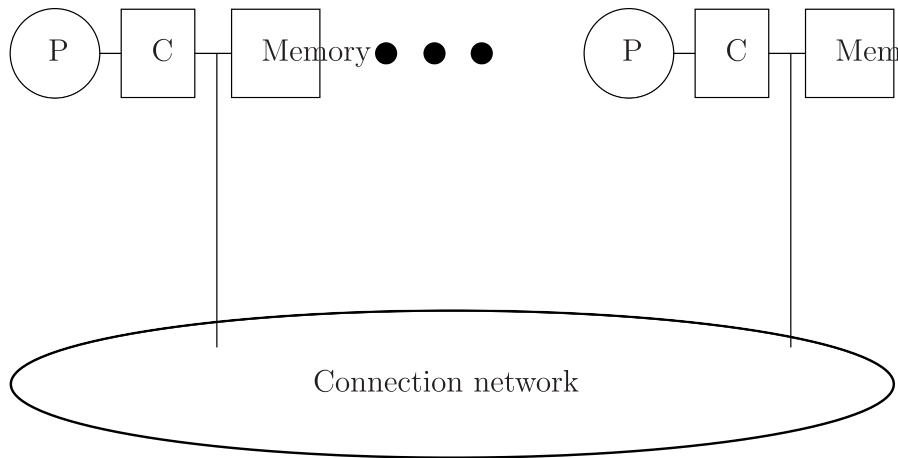
Bus Snooping, MESI-Protocol



- All caches *listen* on the bus whether one of their blocks is affected.
- *MESI-Protocol*: Every block in a cache is in one of four states: *Modified, exclusive, shared, invalid.*
- Write-invalidate, write-back protocol.
- State transition diagram is given on the left.
- Used e.g. in the Pentium.

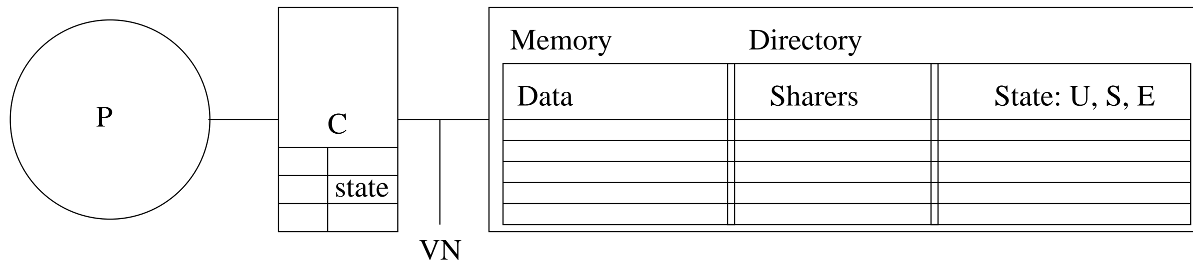
6.3.3 Nonuniform Memory Access Architecture

Nonuniform Memory Access Architecture



- Memories are associated with processors but address space is global.
- Access to local memory is fast.
- Access to remote memories is via the network and slow.
- Including caches there are at least three different access times.
- Solving the cache-coherence problem requires expensive hardware (ccNUMA).
- Machines up to 1024 processors have been built.

Directory Based Cache Coherence



- Main memory is extended by the directory.
- For every block (128 bytes in the SGI-Origin) a state and the sharers are stored.
- For every block in a cache a state is stored (as usual).
- Invalidation is done by sending messages to sharers.
- False sharing: Data processed independently by two procs is in the same block.
- Capacity miss: Proc works on data in a remote memory that does not fit into cache.

6.4 Things to Remember

What you should remember

- Modern microprocessors combine all the features of yesterdays supercomputers.
- Today parallel machines have arrived on the desktop.
- MIMD is the dominant design.
- There are UMA, NUMA and MP architectures.
- Only machines with local memory are scalable.
- Algorithms have to be designed carefully with respect to the memory hierarchy.

Literatur

[1] Grand Challenges applications. http://www.nhse.org/grand_challenge.html

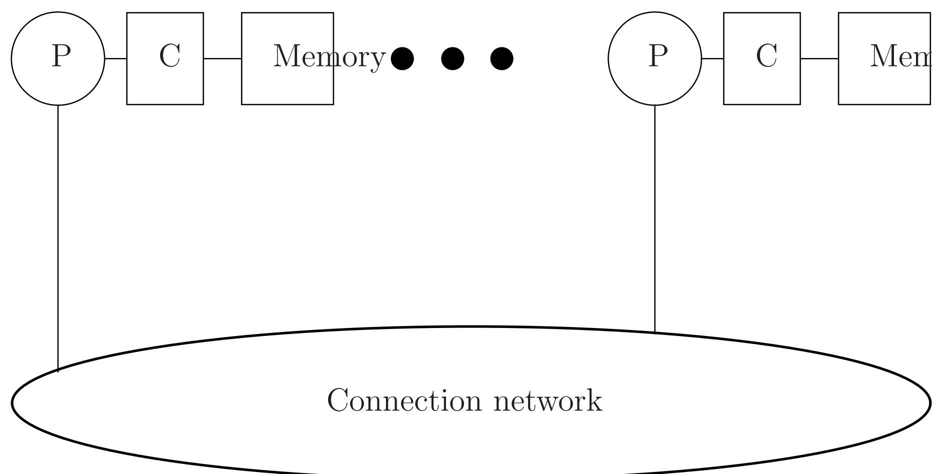
[2] ASCI program website. http://www.llnl.gov/asci/

[3] Achievements of Seymour Cray. http://research.microsoft.com/users/gbell/craytalk/

[4] TOP 500 Supercomputer Sites. http://www.top500.org/

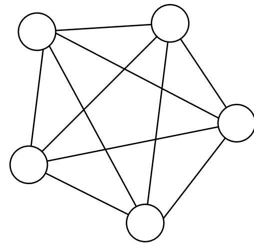
[5] D. E. Culler, J. P. Singh and A. Gupta (1999). *Parallel Computer Architecture*. Morgan Kaufmann.

6.4.1 Private Memory Architecture

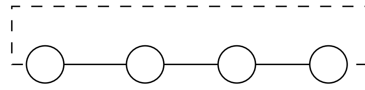


- Processors can only access their local memory.
- Processors, caches and main memory are standard components: Cheap, Moore's law can be fully utilised.
- Network can be anything from fast ethernet to specialised networks.
- Most scalable architecture. Current supercomputers already have more than 10^5 processors.

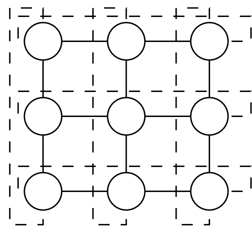
Network Topologies I



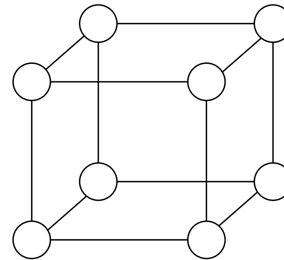
a) fully connected



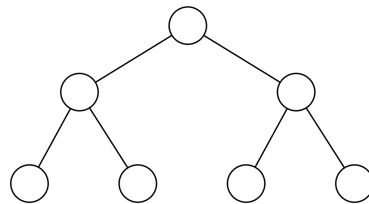
b) 1D-array, ring



c) 2D-array, 2D-torus



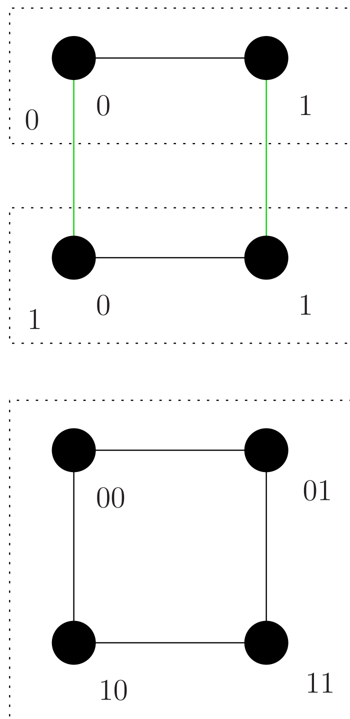
d) Hypercube, 3D-array



e) binary tree

- There are many different types of topologies used for packet-switched networks.
- 1-,2-,3-D arrays and tori.
- Fully connected graph.
- Binary tree.
- Hypercube: HC of dimension $d \geq 0$ has 2^d nodes. Nodes x, y are connected if their bit-wise representations differs in one bit.
- k-dimensional arrays can be embedded in hypercubes.
- Topologies are useful in algorithms as well.

Hypercube



- Hypercubes can be constructed recursively.
- A hypercube of dimension $d = 0$ consists of one node.
- To get a hypercube of dimension $d + 1$:
  - Take two hypercubes of dimension d.
  - Connect corresponding nodes in both hypercubes.
  - Precede each nodes number in the first subcube by 0 and in the second by 1.
- This is shown for $d = 1$ on the left.
- Useful for proving properties of algorithms based on hypercubes.

Network Topologies II

Topology	k	D	l	B	Sym
Fully con.	$P-1$	1	$\frac{P(P-1)}{2}$	$\left(\frac{P}{2}\right)^2$	y
Hypercube	d	d	$d\frac{P}{2}$	$\frac{P}{2}$	y
2D-Torus	4	$2\left\lfloor\frac{r}{2}\right\rfloor$	$2P$	$2r$	y
2D-Array	4	$2(r-1)$	$2P-2r$	r	n
Ring	2	$\left\lfloor\frac{P}{2}\right\rfloor$	P	2	y
Bin. tree	3	$2(h-1)$	$P-1$	1	n

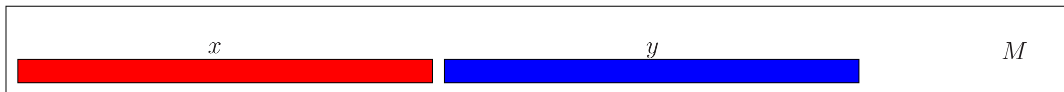
$d = \log_2 P, P = r \times r, h = \lceil \log_2 P \rceil$

- *Degree* k: Max # links per node.
- *Diameter* D: Max distance from one node to any other.

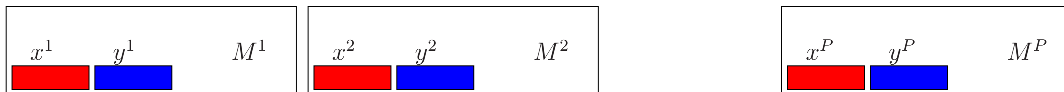
- *Links* l: Total # of links.
- *Bisection width* B: Min # of links one must cut to split machine into two equal-sized halves.
- *Symmetry*: Network looks the same from each node.

Comparison of Architectures by Example

- Given vectors $x, y \in \mathbb{R}^N$, compute scalar product $s = \sum_{i=0}^{N-1} x_i y_i$:
  (1) Subdivide index set into P pieces.
  (2) Compute $s_p = \sum_{i=pN/P}^{(p+1)N/P-1} x_i y_i$ in parallel.
  (3) Compute $s = \sum_{i=0}^{P-1} s_i$. This is treated later.
- *Uniform memory access architecture*: Store vectors as in sequential program:



- *Nonuniform memory access architecture*: Distribute data to the local memories:



- *Message passing architecture*: Same as for NUMA!
- Distributing data structures is hard and not automatic in general.
- Parallelisation effort for NUMA and MP is almost the same.

6.4.2 Things to Remember

What you should remember

- Modern microprocessors combine all the features of yesterdays supercomputers.
- Today parallel machines have arrived on the desktop.
- MIMD is the dominant design.
- There are UMA, NUMA and MP architectures.
- Only machines with local memory are scalable.
- Algorithms have to be designed carefully with respect to the memory hierarchy.

Literatur

[1] Grand Challenges applications. http://www.nhse.org/grand_challenge.html

[2] ASCI program website. http://www.llnl.gov/asci/

[3] Achievements of Seymour Cray. http://research.microsoft.com/users/gbell/craytalk/

[4] TOP 500 Supercomputer Sites. http://www.top500.org/

[5] D. E. Culler, J. P. Singh and A. Gupta (1999). *Parallel Computer Architecture*. Morgan Kaufmann.

6.5 Process Model

6.5.1 A Simple Notation for Parallel Programs

Communicating Sequential Processes

Sequential Program
Sequence of statements. Statements are processed one after another.

(Sequential) Process
A sequential program in execution. The state of a process consists of the values of all variables and the next statement to be executed.

Parallel Computation
A set of interacting sequential processes. Processes can be executed on a single processor (time slicing) or on a separate processor each.

Parallel Program
Specifies a parallel computation.

A Simple Parallel Language

```
parallel <program name> {
    const int P = 8;          // define a global constant
    int flag[P] = {1[P]};     // global array with initialization

    // The next line defines a process
    process <process name 1> [<copy arguments>]
    {
        // put (pseudo-) code here
    }
    ...
    process <process name n> [<copy arguments>]
    { . . . }
}
```

- First all global variables are initialized, then processes are started.
- Computation ends when all processes terminated.
- Processes share global address space (also called threads).

Example: Scalar Product with Two Processes

- We neglect input/output of the vectors.
- Local variables are private to each process.
- Decomposition of the computation is on the **for**-loop.

```
parallel two-process-scalar-product {
    const int N=8;               // problem size
    double x[N], y[N], s=0;      // vectors, result
    process Π1
    {
        int i; double ss=0;
        for (i = 0; i < N/2; i++) ss += x[i]*y[i];
        s=s+ss;                  // danger!
    }
    process Π2
    {
        int i; double ss=0;
        for (i = N/2; i < N; i++) ss += x[i]*y[i];
        s=s+ss;                  // danger!
    }
}
```

6.5.2 The Critical Section Problem

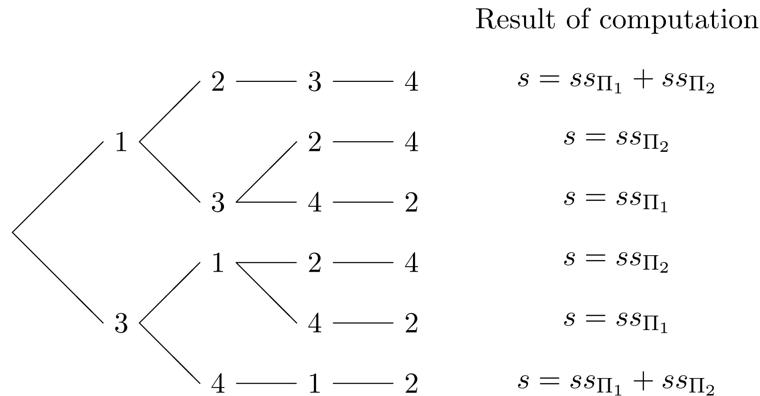
Critical Section

- Statement s=s+ss is not atomic:

	process Π_1		process Π_2
1	load s in R1	3	load s in R1
	load ss in R2		load ss in R2
	add R1, R2, store in R3		add R1, R2, store in R3
2	store R3 in s	4	store R3 in s

- The order of execution of statements of different processes relative to each other is not specified
- This results in an exponentially growing number of possible orders of execution.

Possible Execution Orders



Only some orders yield the correct result!

Mutual Exclusion

- Additional *synchronisation* is needed to exclude possible execution orders that do not give the correct result.
- Critical sections have to be processed under *mutual exclusion*.
- *Mutual exclusion* is one form of synchronisation, the other, *condition synchronisation*, is treated later.
- Mutual exclusion requires:
  - At most one process enters a critical section.
  - No deadlocks.
  - No process waits for a free critical section.
  - If a process wants to enter, it will finally succeed.
- By $[s = s + ss]$ we denote that all statements between "[" and "]" are executed only by *one process at a time*. If two processes attempt to execute "[" at the same time, one of them is delayed.
- We will show later *how* this is implemented.

6.5.3 Single Program Multiple Data

Parametrisation of Processes

- We want to write programs for a variable number of processes:

```
parallel many-process-scalar-product {
    const int N;            // problem size
    const int P;            // number of processors
    double x[N], y[N];      // vectors
    double s = 0;           // result
    process Π [int p ∈ {0, ..., P − 1}]
```

```
    {
        int i; double ss = 0;
        for (i = N * p/P; i < N * (p + 1)/P; i++)
            ss += x[i]*y[i];
        [s = s + ss];                  // sequential execution
    }
}
```

- *Single Program Multiple Data*: Every process has the same code but works on different data depending on p.

6.5.4 Condition Synchronisation

Parallelisation of the Sum

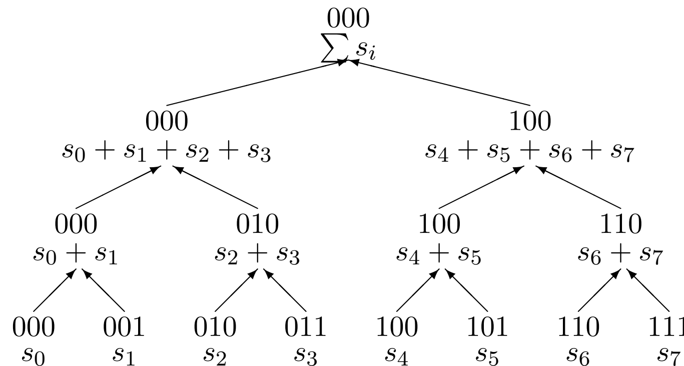
- Computation of the global sum of the local scalar products with $[s = s + ss]$ is not parallel.
- It can be done in parallel as follows ($P = 8$):

$$s = \underbrace{\underbrace{\underbrace{s_0 + s_1}_{s_{01}} + \underbrace{s_2 + s_3}_{s_{23}}}_{s_{0123}} + \underbrace{\underbrace{s_4 + s_5}_{s_{45}} + \underbrace{s_6 + s_7}_{s_{67}}}_{s_{4567}}}_{s}$$

- This reduces the execution time from $O(P)$ to $O(\log_2 P)$.

Tree Combine

Using a binary representation of process numbers, the communication structure forms a binary tree:



Implementation of Tree Combine

```
parallel parallel-sum-scalar-product {
    const int N = 100;              // problem size
    const int d = 4, P = 2^d;       // number of processes
    double x[N], y[N];              // vectors
```

```
    double s[P] = {0[P]};                    // results are global now
    int flag[P] = {0[P]};                    // flags, must be initialized!
    process Π [int p ∈ {0, ..., P − 1}] {
        int i, m, k;
        for (i = 0; i < d; i++) {
            m = 2^i;                         // bit i is 1
            if (p&m) {flag[m]=1; break;}     // I am ready
            while (!flag[p|m]);              // condition synchronisation
            s[p] = s[p] + s[p|m];
        }
    }
}
```

Condition Synchronisation

- A process waits for a condition (boolean expression) to become true. The condition is made true by another process.
- Here some processes wait for the flags to become true.
- Correct initialization of the flag variables is important.
- We implemented this synchronisation using *busy wait*.
- This might not be a good idea in multiprocessing.
- The flags are also called *condition variables*.
- When condition variables are used repeatedly (e.g. when computing several sums in a row) the following rules should be obeyed:
  - A process that waits on a condition variable also resets it.
  - A condition variable may only be set to true again if it is sure that it has been reset before.

Scalar Product on NUMA Architecture

- Every process stores part of the vector as local variables.
- Indices are renumbered from 0 in each process.

```
parallel local-data-scalar-product {
    const int P, N;
    double s = 0;

    process Π [ int p ∈ {0, . . . , P − 1}]
    {
        double x[N/P + 1], y[N/P + 1];
                                        // local part of the vectors
        int i; double ss=0;
```

for $(i = 0, i < (p+1) * N/P - p * N/P; i{+}{+})$ $ss = ss + x[i] * y[i]$;
[$s = s + ss$;]
}
}

6.5.5 Things to Remember

What you should remember

- A parallel computation consists of a set of interacting sequential processes.
- There are two basic synchronisation mechanisms: Mutual exclusion and condition synchronisation.
- Efficient implementation of mutual exclusion requires hardware support (locks).

6.6 OpenMP

- OpenMP is a special implementation of multithreading
- current version 3.0 released in May 2008
- available for Fortran and C/C++
- works for different operating systems (e.g. Linux, Windows, Solaris)
- integrated in various compilers (e.g. Intel icc > 8.0, gcc > 4.2, Visual Studio $>= 2005$, Sun Studio, ...)

Thread Model of OpenMP

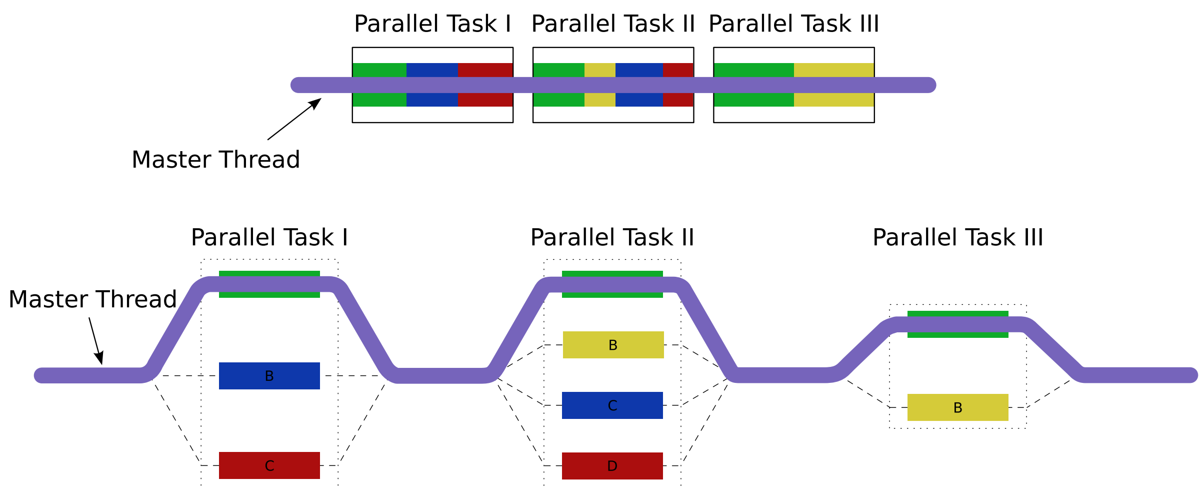


figure from Wikipedia: "OpenMP"

OpenMP Constructs

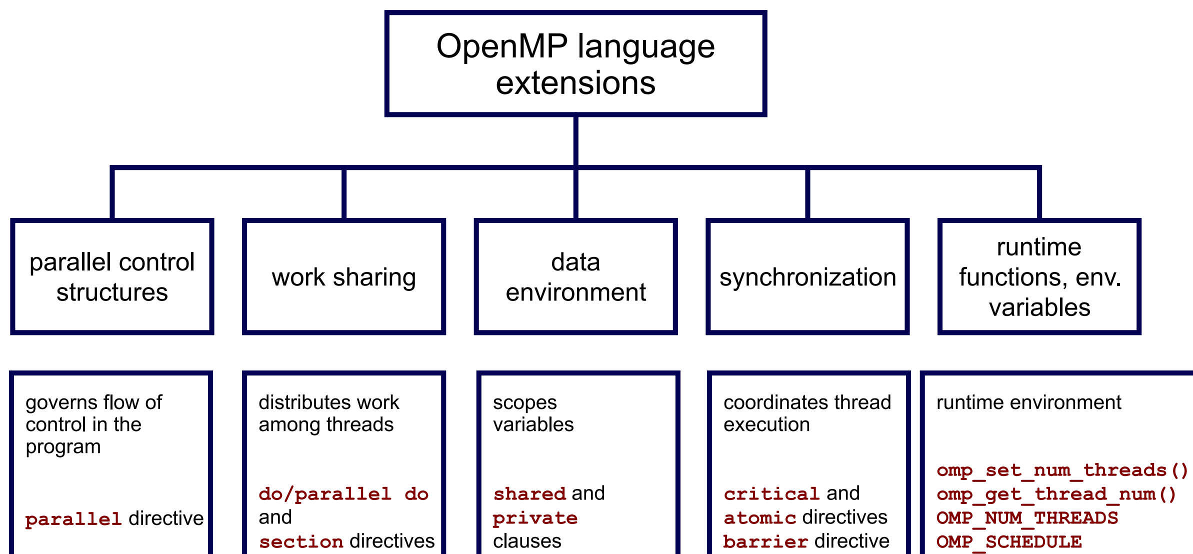


figure from Wikipedia: "OpenMP"

Scalar Product with OpenMP

```
#ifdef _OPENMP
#include<omp.h>
#endif

double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0;
  #pragma omp parallel for shared(a,b,sum)
    for (i=0;i<N;++i)
      sum += a[i] * b[i];
  return(sum);
}
```

Thread Initialisation

parallel for the iterations are distributed among the threads.

sections inside a `sections` block there are several independent `section` blocks which can be executed independently.

single the part inside the `single` block is only executed by one thread. The other threads are waiting at the end of the block.

master the part inside the `master` block is only executed by the master thread. It is skipped by all other threads.

if if an `if` is introduced in the thread initialisation command, the block is only executed in parallel if the condition after `if` is true.

Synchronisation

critical mutual exclusion: the block is only executed by one thread at a time.

atomic same as `critical` but tries to use hardware instructions

ordered is executed like a `for` loop

barrier each thread waits till all threads have reached the barrier

nowait usually threads wait at the end of a block. If `nowait` is used they continue immediately

Accessibility of Variables

shared All threads are accessing the same (shared) data.

private Each thread has its own copy of the data.

default is used to specify what's the default behaviour for variables. Can be `shared`, `private` or `none`.

reduction If `reduction(`*`operator`*`,`*`variable`*`)` is specified, each thread uses a local copy of *`variable`* but all local copies are combined with the operator *`operator`* at the end. Possible operators are `+ * - / & ^ |`

Better Scalar Product with OpenMP

```
#ifdef _OPENMP
#include<omp.h>
#endif

double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0, temp;
  #pragma omp parallel for shared(a,b,sum) private(temp)
    for (i=0;i<N;++i)
    {
      temp = a[i] * b[i];
      #pragma omp atomic
      sum += temp;
    }
  return(sum);
}
```

Improved Scalar Product with OpenMP

```
#ifdef _OPENMP
#include<omp.h>
#endif

double ScalarProduct(std::vector<double> a,
                     std::vector<double> b)
{
  const int N=a.size();
  int i;
  double sum = 0.0;
  #pragma omp parallel for shared(a,b) reduction(+:sum)
    for (i=0;i<N;++i)
      sum += a[i] * b[i];
  return(sum);
}
```

Parallel Execution of different Functions

```
#pragma omp parallel sections
{
#pragma omp section
  {
    A();
    B();
  }
#pragma omp section
  {
    C();
    D();
  }
#pragma omp section
  {
    E();
    F();
  }
}
```

Special OpenMP Functions

There is a number of special functions which are defined in `omp.h`, e.g.

- `int omp_get_num_procs();` returns number of available processors
- `int omp_get_num_threads();` returns number of started threads
- `int omp_get_thread_num();` returns number of this thread
- `void omp_set_num_threads(int i);` set the number of threads to be used

Compiling and Environment Variables

OpenMP is activated with special compiler options. If they are not used, the `#pragma` statements are ignored and a sequential program is created. For `icc` the option is `-openmp`, for `gcc` it is `-fopenmp`

The environment variable `OMP_NUM_THREADS` specifies the maximal number of threads. The call of the function `omp_set_num_threads` in the program has precedence over the environment variable.

Literatur

[1] OpenMP Specification http://www.openmp.org/drupal/node/view/8

[2] Easy OpenMP Tutorial http://www.kallipolis.com/openmp

[3] Very complete OpenMP Tutorial/Reference https://computing.llnl.gov/tutorials/openMP

[4] Intel Compiler (free for non-commercial use) http://www.intel.com/cd/software/products/asmo-na/eng/340679.htm

7 Basics of Parallel Algorithms

Basic Approach to Parallelisation

We want to have a look at three steps of the design of a parallel algorithm:

Decomposition of the problem into independent subtasks to identify maximal possible parallelism.

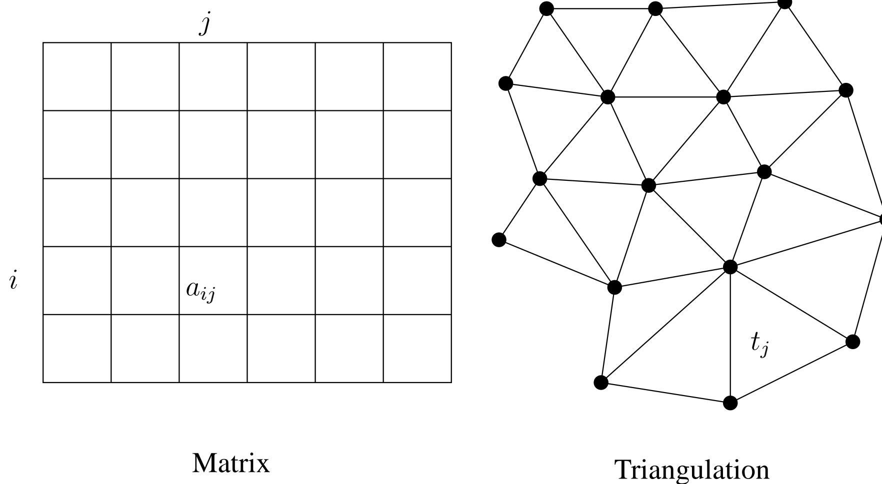
Control of Granularity to balance the expense for computation and communication.

Mapping of Processes to Processors to get an optimal adjustment of logical communication structure and hardware.

7.1 Data Decomposition

Algorithms are often tied to a special data structure. Certain operations have to be done for each data object.

Example: Matrix addition C = A + B, Triangulation



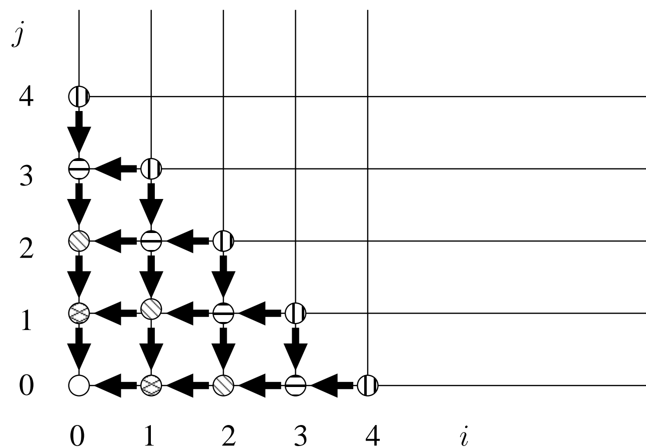
Data Interdependence

Often the operations for all data objects can't be done simultaneously.

Example: Gauß-Seidel/SOR-Iteration with lexicographic numbering.

Calculations are done on a grid, where the calculation at grid point (i, j) depends on the gridpoints $(i-1, j)$ and $(i, j-1)$. Grid point $(0, 0)$ can be calculated without prerequisites. Only grid points on the diagonal $i + j = const$ can be calculated simultaneously.

Data interdependence complicates the data decomposition considerably.

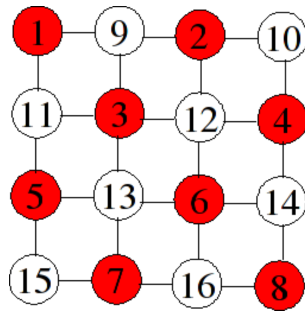


Increasing possible Parallelism

Sometimes the algorithm can be modified to allow for a higher data Independence.

With a different numbering of the unknowns the possible degree of parallelism for the Gauß-Seidel/SOR-iteration scheme can be increased:

Every point in the domain gets a colour such that two neighbours never have the same colour. For structured grids two colours are enough (usually red and black are used). The unknowns of the same colour are numbered first, then the next colour ....

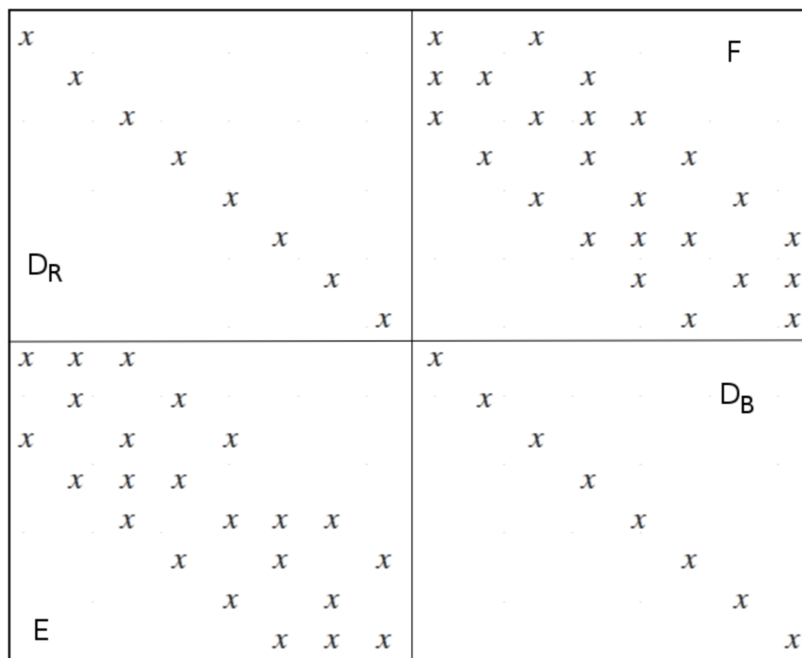


Red-Black-Ordering

The equations for all unknowns with the same colour are then independent of each other. For structured grids we get a matrix of the form

$$A = \begin{pmatrix} D_R & F \\ E & D_B \end{pmatrix}$$

However, while this matrix transformation does not affect the convergence of solvers like Steepest-Descent or CG (as the only depend on the matrix condition) it can affect the convergence rate of relaxation methods.



Functional Decomposition

Functional decomposition can be done, if different operations have to be done on the same data set.

Example: Compiler

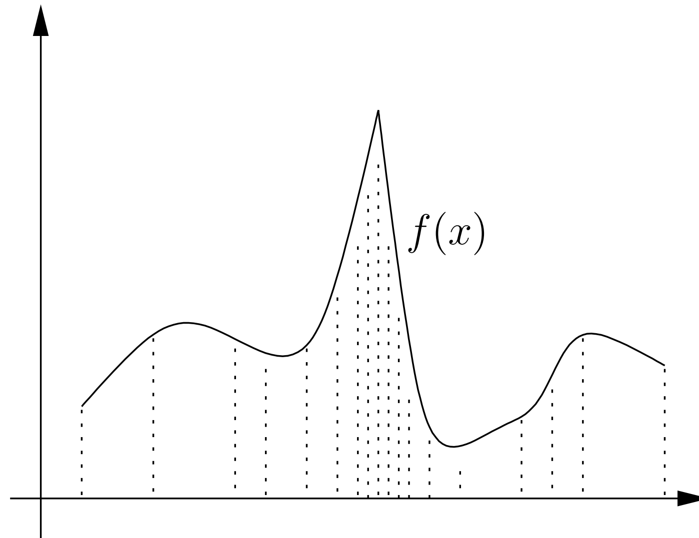
A compiler consists of lexical analysis, parser, code generation, optimisation and assembling. Each of these steps can be assigned to a separate process. The data can run through this steps in portions. This is also called “Macro pipelining”.

Irregular Problems

For some problems the decomposition cannot be determined a priory.

Example: Adaptive Quadrature of a function $f(x)$

The choice of the intervals depends on the function f and results from an evaluation of error predictors during the calculation.



7.2 Agglomeration

Agglomeration and Granularity

The decomposition yields the maximal degree of parallelism, but it does not always make sense to really use this (e.g. one data object for each process) as the communication overhead can get too large.

Several subtasks are therefore assigned to each process and the communication necessary for each subtask is combined in as few messages as possible. This is called “agglomeration”. This reduces the number of messages.

The *granularity* of a parallel algorithm is given by the ratio

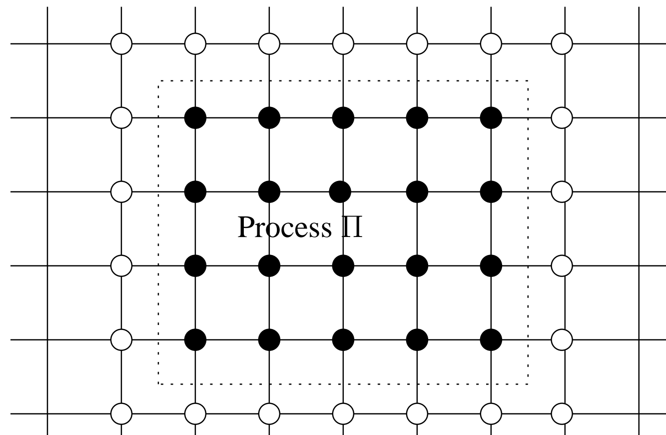
$$\text{granularity} = \frac{\text{number of messages}}{\text{computation time}}$$

Agglomeration reduces the granularity.

Example: Gridbased Calculations

Each process is assigned a number of grid points. In *iterative* calculations usually the value at each node and it’s neighbours is needed. If there is no interdependence all calculations can be done in

parallel. A process with N grid points has to do $O(N)$ operations. With the partition it only needs to communicate $4\sqrt{N}$ boundary nodes. The ratio of communication to computation costs is therefore $O(N^{-1/2})$ and can be made arbitrarily small by increasing N. This is called *surface-to-volume-effect*.



7.3 Mapping of Processes to Processors

Optimal Mapping of Processes to Processors

The set of all process Π forms a undirected graph $G_\Pi = (\Pi, K)$. Two processes are connected if they communicate.

The set of processors P together with the communication network (e.g. hypercube, field, ...) also forms a graph $G_P = (P, N)$.

If we assume $|\Pi| = |P|$, we have to choose which process is executed on which processor. In general we want to find a mapping, such that processes which communicate are mapped to proximate processors. This optimisation problem is a variant of the *graph partitioning problem* and is unfortunately $\mathcal{NP}$-complete.

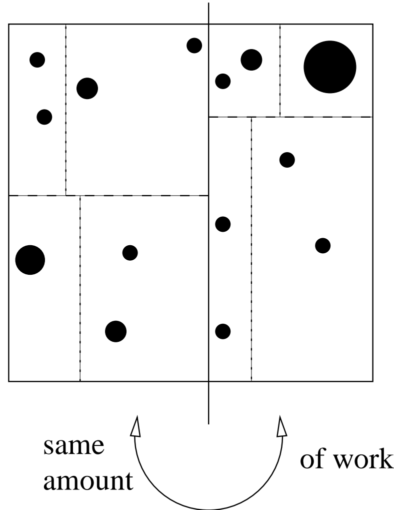
As the transmission time in cut-through networks of state-of-the-art parallel computers is nearly independent of the distance, the problem of optimal process to processor mapping has lost a bit of it's importance. If many processes have to communicate simultaneously (which is often the case!), a good positioning of processes is still relevant.

7.4 Load Balancing

Load Balancing: Static Distribution

Bin Packing At beginning all processors are empty. Nodes are successively packed to the processor with the least work. This also works dynamically.

Recursive Bisection We make the additional assumption, that the nodes have a position in space. The domain is split into parts with an equal amount of work. This is repeated recursively on the subspaces.



7.5 Data Decomposition of Vectors and Matrices

Decomposition of Vectors

A vector $x \in \mathbb{R}^N$ is a ordered list of numbers where each index $i \in I = \{0, \dots, N-1\}$ is associated with a real number x_i.

Data decomposition is equivalent with a segmentation of the index set i in

$$I = \bigcup_{p \in P} I_p, \text{ with } p \neq q \Rightarrow I_p \cap I_q = \emptyset,$$

where P denotes the set of processes. For a good load balancing the subsets I_p, $p \in P$ should contain equal amounts of elements.

To get a coherent index set $\tilde{I}_p = \{0, \dots, |I_p| - 1\}$ we define the mappings

$$\begin{aligned} p\colon &\quad I \to P \text{ and} \\ \mu\colon &\quad I \to \mathbb{N} \end{aligned}$$

which reversibly associate each index $i \in I$ with a process $p(i) \in P$ and a local index $\mu(i) \in \tilde{I}_{p(i)}$: $I \ni i \mapsto (p(i), \mu(i))$.

The inverse mapping $\mu^{-1}(p, i)$ provides a global index to each local index $i \in \tilde{I}_p$ and process $p \in P$ i.e. $p(\mu^{-1}(p,i)) = p$ and $\mu(\mu^{-1}(p,i)) = i$.

Common Decompositions: Cyclic

$$\begin{aligned} p(i) &= i \,\%\, P \\ \mu(i) &= i \div P \end{aligned}$$

$\div$ denotes an integer division and % the modulo function.

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	1	2	3	0	1	2	3	0	1	2	3	0
$\mu(i)$:	0	0	0	0	1	1	1	1	2	2	2	2	3

e.g. $I_1 = \{1, 5, 9\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

Common Decompositions: Blockwise

$$p(i) = \begin{cases} i \div (B+1) & \text{if } i < R(B+1) \\ R + (i - R(B+1)) \div B & \text{else} \end{cases}$$

$$\mu(i) = \begin{cases} i \,\%\, (B+1) & \text{if } i < R(B+1) \\ (i - R(B+1)) \,\%\, B & \text{else} \end{cases}$$

with $B = N \div P$ and $R = N \,\%\, P$

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	0	0	0	1	1	1	2	2	2	3	3	3
$\mu(i)$:	0	1	2	3	0	1	2	0	1	2	0	1	2

e.g. $I_1 = \{4, 5, 6\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

Decomposition of Matrices I

With a matrix $A \in \mathbb{R}^{N \times M}$ a real number a_{ij} is associated with each tupel $(i, j) \in I \times J$, with $I = \{0, \ldots, N-1\}$ and $J = \{0, \ldots, M-1\}$.

To be able to represent the decomposed data on each processor again as a matrix, we limit the decomposition to the one-dimensional index sets I and J.

We assume processes are organised as a two-dimensional field:

$$(p, q) \in \{0, \ldots, P-1\} \times \{0, \ldots, Q-1\}.$$

The index sets I, J are decomposed to

$$I = \bigcup_{p=0}^{P-1} I_p \text{ and } J = \bigcup_{q=0}^{Q-1} J_q$$

Decomposition of Matrices II

Each process (p, q) is responsible for the indices $I_p \times J_q$ and stores it's elements locally as $\mathbb{R}(\tilde{I}_p \times \tilde{J}_q)$-matrix.

Formally the decompositions of I and J are described as mappings p and μ plus q and ν:

$$I_p = \{i \in I \mid p(i) = p\}, \quad \tilde{I}_p = \{n \in \mathbb{N} \mid \exists i \in I : p(i) = p \wedge \mu(i) = n\}$$
$$J_q = \{j \in J \mid q(j) = q\}, \quad \tilde{J}_q = \{m \in \mathbb{N} \mid \exists j \in J : q(j) = q \wedge \nu(j) = m\}$$

Decomposition of a 6×9 Matrix to 4 Processors

$P = 1$, $Q = 4$ (columns), J: cyclic:

J	0	1	2	3	4	5	6	7	8
q	0	1	2	3	0	1	2	3	0
ν	0	0	0	0	1	1	1	1	2

$P = 4$, $Q = 1$ (rows), I: blockwise:

I	p	μ
0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0

$P = 2$, $Q = 2$ (field), I: cyclic, J: blockwise:

J	0	1	2	3	4	5	6	7	8
q	0	0	0	0	0	1	1	1	1
ν	0	1	2	3	4	0	1	2	3

I	p	μ
0	0	0
1	1	0
2	0	1
3	1	1
4	0	2
5	1	2

Optimal Decomposition

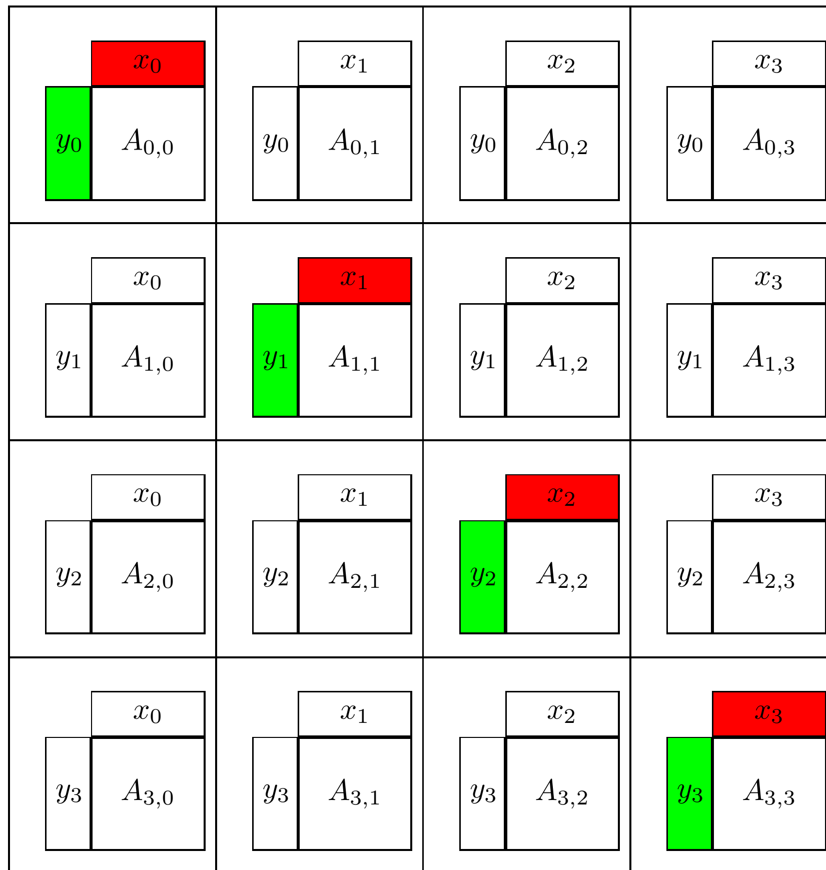
There is no overall best solution for the decomposition of matrices and vectors!

- In general a good load balancing is achieved if the subsets of the matrix are more or less quadratic.
- A good coordination with the algorithm used is usually more important. For example cyclic decomposition is a good solution for LU-decomposition, but not for the solution of the resulting triangular systems.
- Furthermore linear algebra is rarely a goal in it's own, but used in a more general context, like the solution of partial differential equations. The decomposition is then often given by the discretisation and the algorithm has to be flexible enough to deal with it.

7.6 Matrix-Vector Multiplication

Matrix-Vector Multiplication (fully-occupied matrix)

Aim: Calculate the product $y = Ax$ of a matrix $A \in \mathbb{R}^{N\times M}$ and a vector $x \in \mathbb{R}^M$.



Example: Matrix A distributed blockwise in a field, input vector x also blockwise as well as the result vector y.

The vector segment x_q is needed in each processor column.

Then each processor can calculate the local product $y_{p,q} = A_{p,q}x_q$.

Finally the complete result $y_p = \sum_q y_{p,q}$ is collected in the diagonal processor (p, p) with an all-to-one communication.

Matrix-Vector Multiplication: Parallel Runtime

Parallel runtime for a $N \times N$-matrix and $\sqrt{P} \times \sqrt{P}$ processors with a cut-through commu-

nication network:

$$T_P(N,P) = \underbrace{\left(t_s + t_h + t_w \overbrace{\frac{N}{\sqrt{P}}}^{\text{vector}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{Distribution of}\\ x \text{ to column}}} + \underbrace{\left(\frac{N}{\sqrt{P}}\right)^2 2t_f}_{\substack{\text{local matrix-}\\ \text{vector-mult.}}}$$

$$+ \underbrace{\left(t_s + t_h + t_w \frac{N}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}}_{\substack{\text{reduction}\\ (t_f \ll t_w)}} =$$

$$= \operatorname{ld}\sqrt{P}(t_s + t_h)2 + \frac{N}{\sqrt{P}} \operatorname{ld}\sqrt{P} 2t_w + \frac{N^2}{P} 2t_f$$

The contribution of the communication gets arbitrarily small if P and $N \to \infty$.

8 Introduction Message Passing

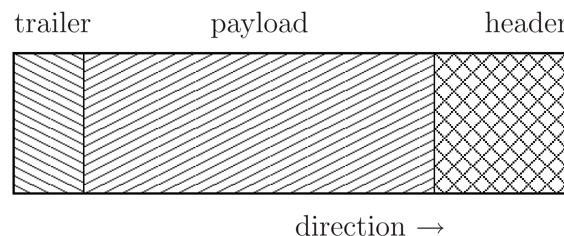
Message Passing

- Developed in the 60s
- Aim: Better structuring of parallel programs (networks didn't exist yet)
- Idea: Data which is needed by other processors is send as messages over the network
- Various powerful solutions available. Differences in elegance.

  Examples:
  - PVM (Parallel Virtual Machine) developed since 1989
  - MPI (Message Parsing Interface) developed since 1994

Message Passing I

- Users view: Copy (contiguous) memory block from one address space to the other.
- Message is subdivided into individual packets.
- Network is packet-switched.
- A packet consists of an envelope and the data:



- Header: Destination, size and kind of data.
- Payload: Size ranges from some bytes to kilobytes.
- Trailer: E.g. checksum for error detection.

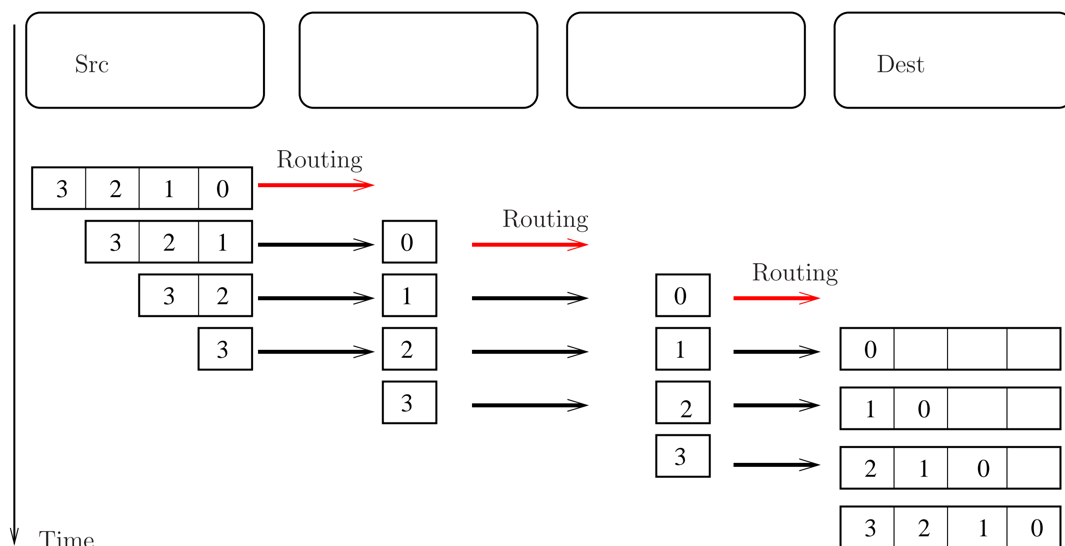
Message Passing II

- Communication protocol: Flow-control, error detection and correction.
- Time to send n bytes can be modelled as

$$t_{mess}(n) = t_s + n * t_b,$$

  t_s: latency, t_b: time per byte, t_b^{-1}: bandwidth.
- Latency is mostly software overhead, not hardware, and depends on the communication protocol.
- TCP/IP is an expensive wide-area network protocol with $t_s \approx 100\mu s$ (in Linux).
- Specialised networks with low-overhead protocols have $t_s \approx 3 \ldots 5\mu s$.

Cut-through Routing



- Pipelining on word level.
- Time to send n bytes: $t_{CT}(n, N, d) = t_s + t_h d + t_b n$.
- Time is essentially independent of distance since $t_h \ll t_s$.

8.1 Synchronous Communication

Send/Receive

The instructions for synchronous point-to-point communication are:

- **send**($dest - process$,$expr_1$,...,$expr_n$) Sends a message to the process $dest - process$ containing the expressions $expr_1$ to $expr_n$.
- **recv**($src - process$, var_1,..., var_n) Receives a message from the process $src - process$ and stores the results of the expressions in the variables var_1 to var_n. The variables have to be of maching type.

Blocking

Both **send** and **recv** are blocking, i.e. they are only finished, after the end of the communication. This synchronises the involved processes. Both sending and receiving process have to execute a matching pair of **send** and **recv** to avoid a deadlock.

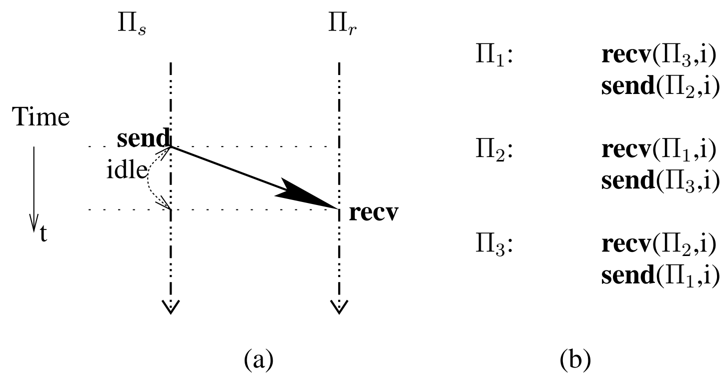


Abbildung 1: (a) Synchronisation of two processes by a **send/recv** pair. (b) Example of a deadlock.

Guards

Blocking communication is not sufficient for all possible tasks.

Sometimes a process does not know which of several partner processes is ready for data exchange.

Possible Solution: Non-blocking functions for detection if a partner process is ready for data reception or sending:

- **int sprobe**($dest - process$)
- **int rprobe**($src - process$).

sprobe returns 1 (*true*), if the specified process is ready for a **recv**-operation, i.e. a **send**-command would *not* block the sender.

rprobe tests, if a **recv**-command would lead to a blockade. To avoid blocking of processes a communication instruction would be written as

- **if** (**sprobe**(Π_d)) **send**(Π_d,...);

The instructions **sprobe** and **rprobe** are also called *"guards"*.

Implementation of rprobe and sprobe

Implementation of **rprobe** is easy. A **send**-instruction sends a message (or a part of it) to the receiver, where it is stored. The receiving process only needs to look up locally if a corresponding message (or part of it) has arrived.

Implementation of **sprobe** is more complicated, as usually the receiving process does not return any information to the sender. As one of the two instructions **sprobe/rprobe** is sufficient (at least in principle) only **rprobe** is used in practise.

Receive Any

- **recv_any**(who,var_1,...,var_n)

  similar effect as **rprobe**

  allows reception of a message from any process

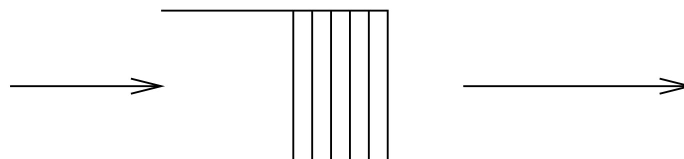
  sender is stored in the variable *who*

8.2 Asynchronous Communication

Asynchronous Communication

- **asend**($dest - process$,$expr_1$,...,$expr_n$)
- **arecv**($src - process$,var_1,...,var_n)

have the same semantic as **send/recv**, but are *non-blocking*.
In principle a process can execute any amount of **asend** instructions without delay.
The involved processes are *not* synchronised implicitly.
Beginning and end of the communication channel can be visualised as a waiting line, buffering all messages until they are accepted by the receiving process.



Check for Success

In practise buffer storage is limited. If the buffer is full, no further **asend** instruction can be executed. Therefore it is necessary to test if a former communication instruction has been completed.

asend/**arecv** are returning a unique identifier

- **msgid asend**(...)
- **msgid arecv**(...)

With this identifier the function

- **int** success(**msgid** m)

returns the state of the communication.

Mixing of Synchronous and Asynchronous Communication

We are going to allow mixing of the functions for asynchronous and synchronous communication. For example the combination of asynchronous sending and synchronous receiving can make sense.

9 The Message Passing Interface

The Message Passing Interface (MPI)

- Portable Library with functions for message exchange between processes
- Developed 1993-94 by a international board
- Available on nearly all computer platforms
- Free Implementations also for LINUX Clusters:**MPICH**[3] and **OpenMPI**[4] (former **LAM**)
- Properties of MPI:
  - library with C-, C++ and Fortran bindings (no language extension)
  - large variety of point-to-point communication functions
  - global communication
  - data conversion for heterogeneous systems
  - subset formation and topologies possible

MPI-1 consits of more than 125 functions, defined in the standard on 200 pages. We therefore only treat a small selection of it's functionality.

[3] http://www-unix.mcs.anl.gov/mpi/mpich
[4] http://www.open-mpi.org/

9.1 Simple Example

Simple Example in C

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>  // provides MPI macros and functions


int main (int argc, char *argv[])
{
    int my_rank;
    int P;
    int dest;
    int source;
    int tag=50;
    char message[100];
    MPI_Status status;

    MPI_Init(&argc,&argv);  // begin of every MPI program

    MPI_Comm_size(MPI_COMM_WORLD,&P);  // number of
                                       // involved processes
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    // number of current process always between 0 and P-1

    sprintf(message,"I␣am␣process␣%d␣of␣%d\n",my_rank,P);
    if (my_rank!=0)
    {
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,  // Send data
                 dest,tag,MPI_COMM_WORLD);            // (blocking)
    }
    else
    {
        puts(message);
        for (source=1; source<P; source++)
        {
            MPI_Recv(message,100,MPI_CHAR,source,tag,  // Receive data
                     MPI_COMM_WORLD,&status);          // (blocking)
            puts(message);
        }
    }

    MPI_Finalize();  // end of every MPI program

    return 0;
}
```

Simple Example in C++

```cpp
#include <iostream>
#include <sstream>
#include <mpi.h>  // provides MPI macros and functions


int main (int argc, char *argv[])
{
    MPI::Init(argc,argv);  // begin of every MPI program

    int P = MPI::COMM_WORLD.Get_size(); // number of
                                        // involved processes
```

```
    int myRank = MPI::COMM_WORLD.Get_rank();
13  int tag=50;

15  // number of current process always between 0 and P-1
    if (myRank != 0)
17  {
        std::ostringstream message;
19      message << "I␣am␣process␣" << myRank << "␣of␣" << P << std::endl;
        int dest = 0;
21      MPI::COMM_WORLD.Send(message.str().c_str(),      // Send data
            message.str().size()+1,MPI::CHAR,dest,tag);  // (blocking)
23  }

24  else
    {
26      std::cout << "I␣am␣process␣0␣of␣" << P << std::endl << std::endl;
        for (int source=1; source<P; ++source)
28      {
            char message[100];
30          MPI::COMM_WORLD.Recv(message,100,MPI_CHAR,  // Receive data
                 source,tag);                          // (blocking)
32          std::cout << message << std::endl;
        }
34  }

36  MPI::Finalize();  // end of every MPI program

38  return 0;
}
```

Compilation of Program

- Sample program is written in SPMD-Stile. This is not prescribed by the MPI Standard, but makes starting of program easier.
- Compiling, linking and execution is different for every implementation.
- Many implementations contain shell-scripts, which hide the location of the libraries. For MPICH the commands to compile the program and start 8 processes are

  ```
  mpicc -o hello hello.c
  mpirun -machinefile machines -np 8 hello
  ```

  In this case the names of the computers used are taken from the file `machines`.

  For C++ programs the command for compilation is

  ```
  mpicxx -o hello hello.c
  ```

Output of the Example Programs (with P=8)

```
1 I am process 0 of 8

3 I am process 1 of 8

5 I am process 2 of 8

7 I am process 3 of 8
```

```
9 I am process 4 of 8

11 I am process 5 of 8

13 I am process 6 of 8

15 I am process 7 of 8
```

Structure of MPI-Messages

MPI-Messages consist of the actual *data* and an *envelope* comprising:

1. number of the sender
2. number of the receiver
3. tag: an integer value to mark different messages between identical communication partners
4. communicator: subset of processes + communication context. Messages belonging to different contexts don't influence each other. Sender and receiver have to use the same communicator. The communicator `MPI_COMM_WORLD` is predefined by MPI and contains all started processes.

   In C++ communicators are objects over which the messages are sent. The communicator object `MPI::COMM_WORLD` is already predefined.

Initialising and Finishing

```
int MPI_Init(int *argc, char ***argv)

void Init(int& argc, char**& argv)
void Init()
```

Before the first MPI functions are used, `MPI_Init` / `MPI::Init` has to be called.

```
int MPI_Finalize(void)

void Finalize()
```

After the last MPI function call `MPI_Finalize` / `MPI::Finalize` must be executed to get a defined shutdown of all processes.

9.2 Communicators and Topologies

Communicator

All MPI communication functions contain an argument of type `MPI_Comm` (in C) or are methods of a communicator object. Such a *communicator* contains the following abstractions:

- *Process group:* builds a subset of processes which take part in a global communication. The predefined communicator `MPI_COMM_WORLD` contains all started processes.
- *Context:* Each communicator defines it's own communication context. Messages can only be received within the same context in which they are send. It's e.g. possible for a numerical library to define it's own communicator. The messages of the library are then completely separated from the messages of the main program.

- *Virtual Topologies:* A communicator represents a set of processes $\{0, \dots, P-1\}$. This set can optionally be provided with an additional structure, e.g. a multi-dimensional field or a general graph.
- *Additional Attributes:* An application (e.g. a library) can associate any static data with a communicator. The communicator is then only a means to preserve this data until the next call of the library.

Communicators in C++

In C++ communicators are objects of classes derived from a base class `Comm`. The available derived classes are

`Intracomm` for the communication inside a group of processes. `MPI::COMM_WORLD` is an object of class `Intracomm` as all processes are included in `MPI::COMM_WORLD`. There exist two derived classes for the formation of topologies of processes

- `Cartcomm` can represent processes which are arranged on a Cartesian topology
- `Graphcomm` can represent processes which are arranged along arbitrary graphs

`Intercomm` for the communication between groups of processes

Determining Rank and Size

```
int MPI_Comm_size(MPI_Comm comm, int *size)

int Comm::Get_size() const
```

The number of processes in a communicator is determined by the function `MPI_Comm_size` / `Comm::Get_size()`. If the communicator is `MPI_COMM_WORLD` this is equal to the total number of started processes.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)

int Comm::Get_rank() const
```

Each process has a unique number inside a group represented by a communicator. This number can be determined by `MPI_Comm_rank` / `Comm::Get_rank()`.

9.3 Blocking Communication

```
int MPI_Send(void *message, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *message, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);

 void Comm::Send(const void* buf, int count,
             const Datatype& datatype, int dest, int tag) const
 void Comm::Recv(void* buf, int count, const Datatype& datatype,
             int source, int tag, Status& status) const
 void Comm::Recv(void* buf, int count, const Datatype& datatype,
             int source, int tag) const
```

The first three arguments `message`, `count`, and `dt`, specify the actual data. `message` points to a contiguous memory block containing `count` elements of type `dt`. The specification of the data type makes data conversion by MPI possible. The arguments `dest`, `tag` and `comm` form

the envelope of the message (the number of the sender/receiver is given implicitly by the invocation).

Data Conversion

MPI implementations for heterogeneous systems are able to do a automatic conversion of the data representation. The conversion method is left to the particular implementation (e.g. by XDR).

MPI provides the architecture independent data types:

MPI_CHAR, MPI_UNSIGNED_CHAR, MPI_BYTE MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG_INT, MPI_UNSIGNED, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE and MPI_LONG_DOUBLE.

IN C++ the datatypes are:

MPI::CHAR, MPI::UNSIGNED_CHAR, MPI::BYTE MPI::SHORT, MPI::INT, MPI::LONG, MPI::LONG_LONG_INT, MPI::UNSIGNED, MPI::UNSIGNED_SHORT, MPI::UNSIGNED_LONG, MPI::FLOAT, MPI::DOUBLE and MPI::LONG_DOUBLE.

The MPI data type `MPI_BYTE` / `MPI::BYTE` is *never* converted.

Status

```
typedef struct {
2     int count;
      int MPI_SOURCE;
4     int MPI_TAG;
      int MPI_ERROR;
6 } MPI_Status;
```

In C `MPI_Status` is a struct containing information about the number of received objects, source rank, tag and an error status.

```
  int Status::Get_source() const
2 void Status::Set_source(int source)
  int Status::Get_tag() const
4 void Status::Set_tag(int tag)
  int Status::Get_error() const
6 void Status::Set_error(int error)
```

In C++ an object of class `Status` provides methods to access the same information.

Varieties of Send

- *buffered send* (`MPI_Bsend` / `Comm::Bsend`): If the receiver has not yet executed a corresponding **recv**-function, the message is buffered by the sender. If enough memory is available, a "buffered send" is always terminated immediately. In contrast to asynchronous communication the sending buffer `message` can be immediately reused.

- *synchronous send* (`MPI_Ssend` / `Comm::Ssend`): The termination of a synchronous send indicates, that the receiver has executed the **recv**-function and has started to read the data.

- *ready send* (`MPI_Rsend` / `Comm::Rsend`): A ready send can only be started if the receiver has already executed the corresponding **recv**. The call leads to an error else .

MPI_Send and MPI_Receive II

The MPI_Send-command either has the semantic of MPI_Bsend or MPI_Ssend, depending on the implementation. Therefore MPI_Send can, but don't have to block. The sending buffer can be reused immediately in any case.

The function MPI_Recv is in any case blocking, e.g. it is only terminated if message contains data. The argument status contains source, tag and error status of the received message.

MPI_ANY_SOURCE / MPI::ANY_SOURCE and MPI_ANY_TAG / MPI::ANY_TAG can be used for the arguments src and tag respectively. Thus MPI_Recv contains the functionality of **recv_any**.

Guard Function

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
bool Comm::Iprobe(int source, int tag, Status& status) const
bool Comm::Iprobe(int source, int tag) const
```

is a non-blocking guard function for the receiving of messages. flag is set to true true ($\neq 0$) if a message with matching source and tag can be received. The arguments MPI_ANY_SOURCE / MPI::ANY_SOURCE and MPI_ANY_TAG / MPI::ANY_TAG are also possible.

9.4 Non-blocking communication

MPI provides the functions

```
int MPI_ISend(void *buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *req);
int MPI_IRecv(void *buf, int count, MPI_Datatype dt,
              int src, int tag, MPI_Comm comm,
              MPI_Request *req);

Request Comm::Isend(const void* buf, int count,
              const Datatype& datatype, int dest, int tag) const
Request Comm::Irecv(void* buf, int count, const Datatype& datatype,
              int source, int tag) const
```

for non-blocking communication. They imitate the respective communication operations. With the MPI_Request / MPI::Request-objects the state of the communication job can be checked (corresponding to our **msgid**).

MPI_Request-Objects

The state of the communication can be checked with MPI_Request-objects returned by the communication functions using the function (among others)

```
int MPI_Test(MPI_Request *req,int *flag, MPI_Status *status);
```

flag is set to true ($\neq 0$) if the communication job designated by req has terminated. In this case status contains information about sender, receiver and error state.

```
bool Request::Test(Status& status)
bool Request::Test()
```

In C++ the Test method of the Request object returned by the communicator method returns true if the job initiated by the method call has terminated.

It is important to mind that the MPI_Request-object becomes invalid as soon as MPI_Test / Request::Test returns flag==true / true. It must not be used thereafter, as the MPI_Request-objects are managed by the MPI-implementation (so-called opaque objects).

9.5 Global Communication

MPI also offers functions for global communication where all processes of a communicator participate.

```
int MPI_Barrier(MPI_Comm comm);

void Intracomm::Barrier() const
```

blocks every process until all processes have arrived (e.g. until they have executed this function).

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
              int root, MPI_Comm comm);

void Intracomm::Bcast(void* buffer, int count,
              const Datatype& datatype, int root) const
```

distributes the message of process root to all other processes of the communicator.

Collection of Data

MPI offers various functions for the collection of data. For example:

```
int MPI_Reduce(void *sbuf, void *rbuf, int count,
        MPI_Datatype dt, MPI_Op op, int root, MPI_Comm comm);

void Intracomm::Reduce(const void* sendbuf, void* recvbuf, int count,
        const Datatype& datatype, const Op& op, int root) const
```

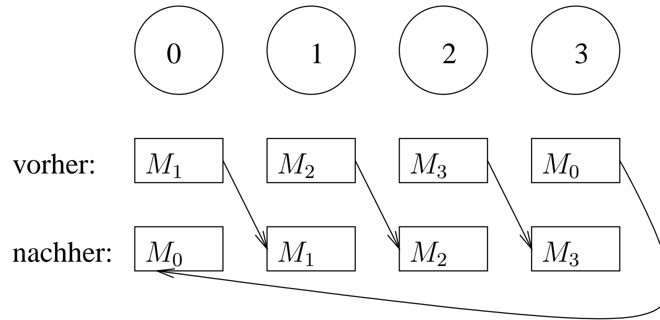
combines the data in the send buffer sbuf of all processes with the associative operation op. The result is available in the receive buffer rbuf of process root. As operations op e.g. MPI_SUM, MPI_MAX and MPI_MIN can be used to calculate the sum, maximum or minimum over all processes.

Remark
Global communications have to be called with maching arguments (e.g. root in MPI_Reduce) by all processes of a communicator.

9.6 Avoiding Deadlocks: Coloring

Shifting along Ring: Creation of Deadlocks

Problem: Each process $p \in \{0, \ldots, P-1\}$ has to transfer data to $(p+1)\%P$.



With *blocking* communication functions a deadlock is created using

...
send($\Pi_{(p+1)\%P}$,msg);
recv($\Pi_{(p+P-1)\%P}$,msg);
...

Deadlock Prevention

A solution with optimal degree of parallelism can be reached by colouring.
Let $G = (V, E)$ be a graph with

$$\begin{aligned} V &= \{0, \ldots, P-1\} \\ E &= \{e = (p, q) | \text{process } p \text{ has to communicate with process } q\} \end{aligned}$$

Each edge then has to be given a colour so that the colours of all edges meeting at each node are unique. The colour assignment is given by the mapping $c\colon E \to \{0, \ldots, C-1\}$, where C is the number of colours necessary. The communication can then be done in C steps, whereas only messages along edges of colour i are exchanged in step $0 \le i < C$.

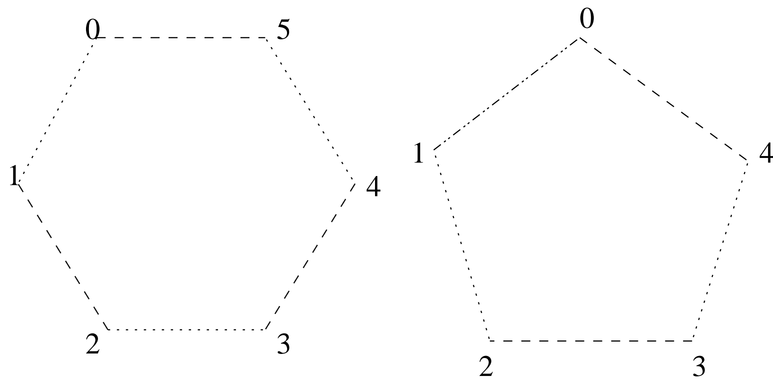
Number of Colours

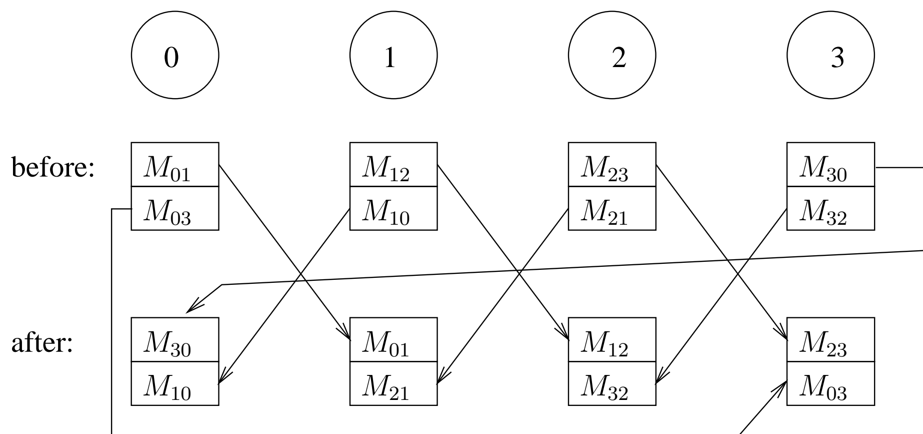
Two colours are needed for shifting along a ring if P is even, but three colours are needed if P is odd.

Exchange with all Neighbours

The algorithm can easily be modified to shift messages in both directions. Each Process exchanges then messages with both neighbours in the graph.

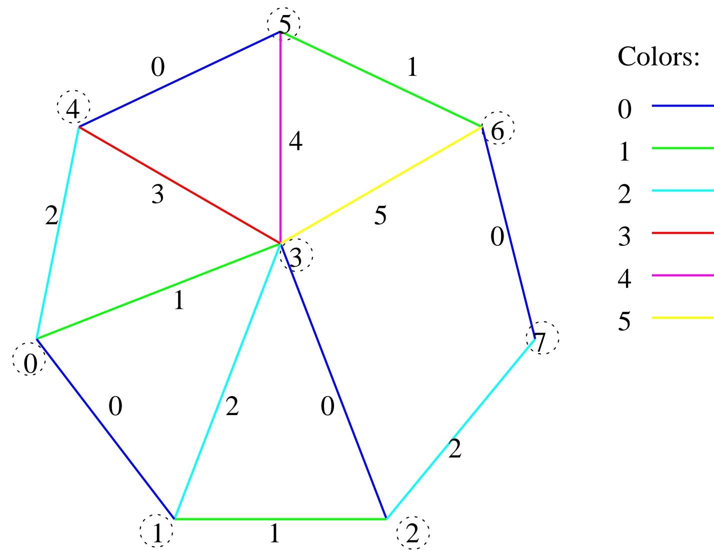
The number of colours is the same as in the case of simple shifting. Two messages are send along each edge. Deadlocks are prevented if the process with the smaller number sends first.


0
5
1
4
2
3
0
1
4
2
3


0
1
2
3
before:
M_{01}
M_{03}
M_{12}
M_{10}
M_{23}
M_{21}
M_{30}
M_{32}
after:
M_{30}
M_{10}
M_{01}
M_{21}
M_{12}
M_{32}
M_{23}
M_{03}

General Graphs

For general, undirected graphs the determination is not so simple and it can also be necessary to determine the colouring at runtime. Distributed algorithms are available to solve this problem.



Alternative: Timestamps

Creating Communicators based on Colours

A new communicator can be created with the function

```
int MPI_Comm_split(MPI_Comm comm, int colour,
                   int key, MPI_Comm *newcomm);

Intracomm Intracomm::Split(int colour, int key) const
```

`MPI_Comm_split` is a collective operation, executed by *all* processes of the communicator `comm`. All processes with the same value of the argument `colour` form a new communicator. The ordering (rank) within the new communicator is controlled by the argument `key`.

10 Things to Remember

What you should remember

- Message passing is a general concept for data exchange. There are different realisations of this concept.
- Two different types of communication exist: Blocking (or synchronous) and Non-blocking (or asynchronous).
- MPI is a standardised message passing system. Different implementations exist implementing the MPI standard (e.g. **MPICH**, **OpenMPI**).
- C and C++ versions of the MPI functions exist

- Deadlocks in communication can occur with blocking communication functions and need to be averted by techniques like colouring or timestamps.

Literatur

[1] *MPI: The different version of the Message-Passing Interface Standard* http://www.mpi-forum.org/docs/

[2] *MPICH-A Portable Implementation of MPI* http://www-unix.mcs.anl.gov/mpi/mpich

[3] *Open MPI: Open Source High Performance Computing* http://www.open-mpi.org/

[4] *Overview of available MPI Tutorials* http://www-unix.mcs.anl.gov/mpi/tutorial/

11 Analysis of Parallel Algorithms

Introduction

- We want to solve a problem Π on a parallel computer.
- Example: "Solve system of linear equations $Ax = b$ with $A \in \mathbb{R}^{N \times N}$ and $x, b \in \mathbb{R}^N$".
- Problem description does not include *how* it is solved.
- Π has a *problem size* parameter $N \in \mathbb{N}$.
- Π is solved on a parallel machine with P identical processors and a given communication network.
- Π is solved
  - on 1 processor with a *sequential* algorithm and
  - on P processors with a *parallel* algorithm.

How "good" is the parallel algorithm ?

Time Measurements

Depending on problem size N and processor number P we can define the following run-times:

- *Sequential run-time* $T_S(N)$: Time needed by a specified algorithm to solve Π for problem size N on one processor of the parallel machine.
- *Best sequential run-time* $T_{best}(N)$: Run-time of a *hypothetical* sequential algorithm that solves Π for any problem size N in the shortest possible time.
- *Parallel run-time* $T_P(N, P)$: Run-time of a given parallel algorithm to solve Π for problem size N on a parallel machine with P processors.

Speedup

Definition 11.1 (Speedup)**.** Given the time measurements we can define the *speedup*

$$S(N,P) = \frac{T_{best}(N)}{T_P(N,P)}.$$

Remark.
The speedup is defined with respect to the best sequential algorithm! If it were defined via some $T_S(N)$, any speedup could be achieved by comparing against a slow sequential algorithm.

Speedup Bound

Theorem 11.2 (Speedup bound)**.** *The speedup $S(N,P)$ fullfills the inequality*

$$S(N,P) \leq P.$$

Beweis. Suppose we have $S(N,P) > P$, then simulating the parallel algorithm on one processor would need time $PT_P(N,P)$ and

$$PT_P(N,P) = P\frac{T_{best}(N)}{S(N,P)} < T_{best}(N)$$

which is a contradiction to the definition of T_{best}. □

Superlinear Speedup

Remark.
The proof above assumes that simulation takes no additional time. There are inherently parallel algorithms where the sequential algorithm *must* simulate a parallel algorithm: Given a program for computing $f(n)$, find $n \in \{1,\ldots,P\}$ where its run-time is minimal.

Superlinear Speedup.
Some people measure speedups greater than P. This is most often because T_{best} must be replaced by some T_S in practise. The most common situation is that for increasing P and fixed N the local problems fit into cache and the sequential code is not cache-aware.

Efficiency

Definition 11.3 (Efficiency)**.** The *efficiency* of a parallel algorithm is defined as

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T_{best}(N)}{PT_P(N,P)}.$$

Theorem 11.4 (Efficiency bound)**.** *The speedup bound immediately gives*

$$E(N,P) \leq 1.$$

Remark.
Interpretation: $E \cdot P$ processors are effectively working on the solution of the problem, $(1-E)\cdot P$ are overhead.

Other Measures

Definition 11.5 (Cost)**.** The *cost* of a parallel algorithm is defined as

$$C(N,P) = PT_P(N,P).$$

In contrast to the previous numbers it is not dimensionless.

Definition 11.6 (Cost optimality)**.** An algorithm is called *cost optimal* if its cost is proportional to T_{best}. Then its efficiency $E(N,P) = T_{best}/C(N,P)$ is constant.

Definition 11.7 (Degree of parallelism)**.** $\Gamma(N)$ is the maximum number of machine instructions that can be executed in parallel. Obviously $\Gamma(N) = O(T_P(N,1))$.

11.1 Examples

11.1.1 Scalar Product

Run-time of best sequential algorithm is given by

$$T_S(N) = N2t_a,$$

t_a: time for a floating point operation.

Run-time of parallel algorithm using tree combine:

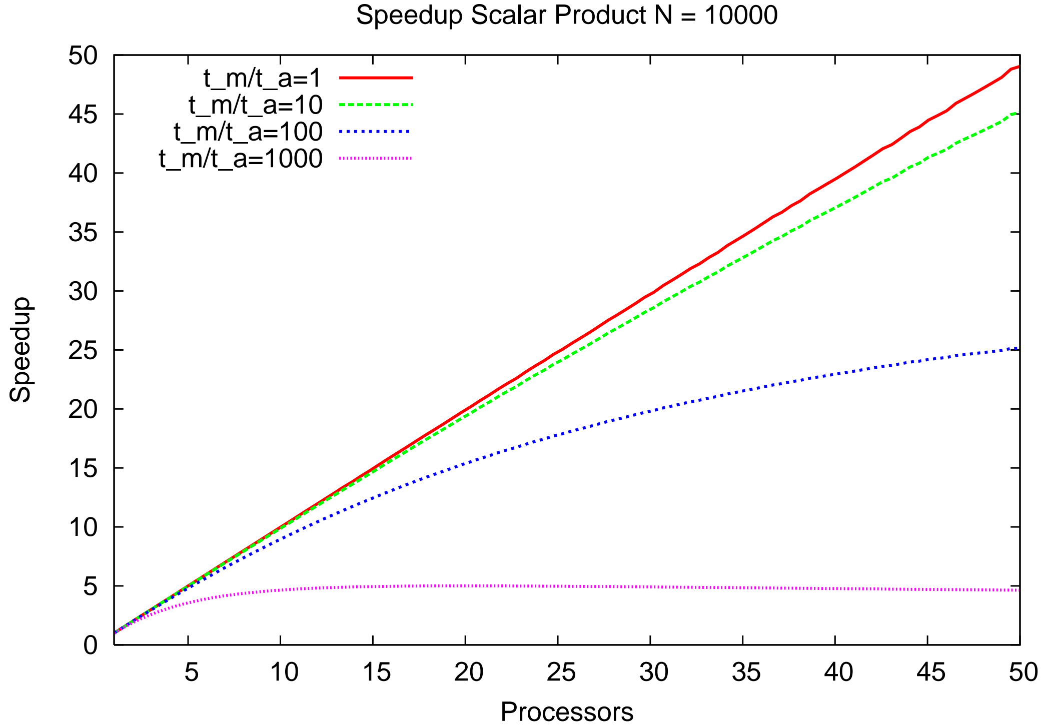
$$T_P(N,P) = \left\lceil \frac{N}{P} \right\rceil 2t_a + \lceil \log_2 P \rceil (t_m + t_a),$$

t_m: time to send a number.

Speedup is

$$S(N,P) = \frac{N2t_a}{\left\lceil \frac{N}{P} \right\rceil 2t_a + \lceil \log_2 P \rceil (t_m + t_a)} = \frac{P}{\underbrace{\left\lceil \frac{N}{P} \right\rceil \frac{P}{N}}_{\text{load imbalance}} + \underbrace{\frac{P\lceil \log_2 P \rceil}{N} \frac{t_m + t_a}{2t_a}}_{\text{communication}}}.$$

Speedup Graph Scalar Product



Reasons For Non-optimal Speedup
We can identify the following reasons for non-optimal speedup:

- *Load imbalance*: Not every processor has the same amount of work to do.
- *Sequential part*: Not all operations of the sequential algorithm can be processes in parallel.
- *Additional operations*: The optimal sequential algorithm cannot be parallelised directly and must be replaced by a slower but parallelisable algorithm.
- *Communication overhead*: Depends relative cost of computation and communication.

11.1.2 Gaussian Elimination

Run-time of best sequential algorithm is given by

$$T_S(N) = N^3 \frac{2}{3} t_a,$$

t_a: time for a floating point operation.
Run-time of parallel algorithm (row-wise cyclic, asynchronous):

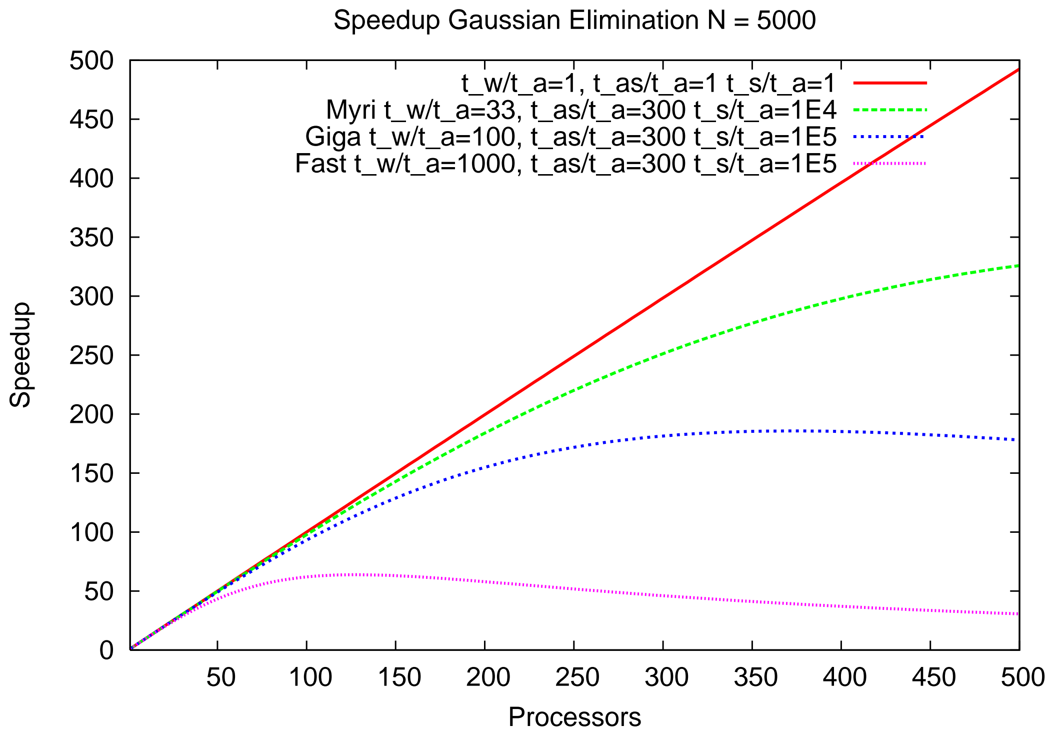
$$T_P(N,P) = (P-1)(t_s + N t_m) + \sum_{m=N-1}^{1} \left(\left\lceil \frac{m}{P} \right\rceil m 2 t_a + t_{as} \right)$$
$$\approx \frac{2}{3} \frac{N^3}{P} t_a + N t_{as} + P t_s + N P t_m,$$

t_s: message latency, t_{as}: asynchronous latency.

Speedup is

$$S(N,P) = \frac{P}{1 + \frac{3}{2}\frac{P}{N^2}\left(P\frac{t_w}{t_a} + \frac{t_{as}}{t_a} + \frac{P}{N}\frac{t_s}{t_a}\right)}$$

Speedup Graph Gaussian Elimination



11.2 Scalability

11.2.1 Fixed Size

- Scalability is the ability of a parallel algorithm to use an increasing number of processors: **How does $S(N,P)$ behave with P?**
- $S(N,P)$ has two arguments. **What about N?**
- We consider several different choices.

Fixed Size Scaling and Amdahl's Law

Fixed size scaling
Very simple: Choose N to be fixed. Equivalently, we can fix the sequential execution time $T_{\text{best}} = T_{\text{fix}}$.

Amdahl's Law (1967)

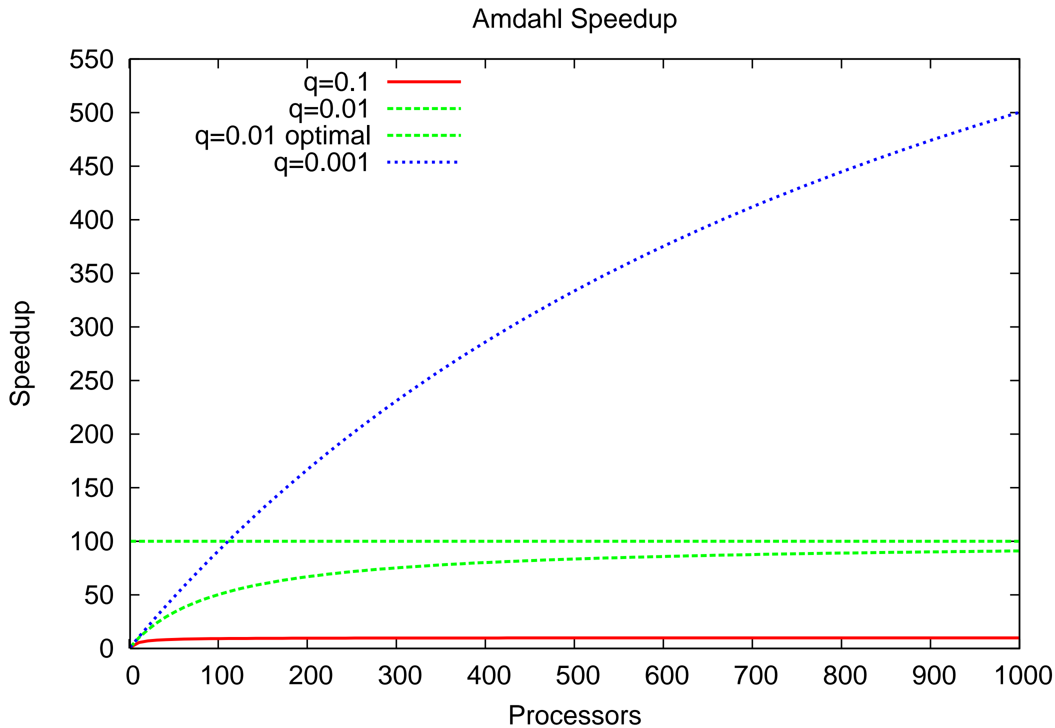
Assume a fixed sequential execution time. The part qT_{fix} with $0 \le q \le 1$ is assumed not to be parallelisable. The remaining part $(1-q)T_{\text{fix}}$ is assumed to be fully parallelisable. Then the speedup is given by

$$S_A(P) = \frac{T_{fix}}{qT_{fix} + (1-q)T_{fix}/P} = \frac{1}{q + \frac{1-q}{P}}.$$

Remark

$$S_A(P) \le 1/q.$$

Speedup Graph Amdahl's Law



11.2.2 Scaled Size

Other Scalings

The sequential part q is a function of N and usually decreases with increasing N. Therefore choose $N = N(P)$.

Gustafson Scaling

Choose $N = N(P)$ such that $T_P(N(P), P) = T_{\text{fix}}$. Motivation: Weather prediction.

Memory Scaling

Choose $N = N(P)$ such that memory requirements scale with the available memory: $M(N(P)) = M_0 P$. Useful for memory-bound applications, e.g. finite element methods.

Isoefficient Scaling

Choose $N = N(P)$ such that $E(N(P), P) = E_0$. This is not always possible! If such $N(P)$ exists the algorithm is called scalable.

Scalar Product: Fixed Sequential Execution Time

Remember the scalar product example:

$$T_S(N) = N2t_a,$$
$$T_P(N,P) = (N/P)2t_a + \log_2 P(t_m + t_a).$$

Fixed size scaling $T_S(N,P) = T_{fix}$:

$$N2t_a = T_{fix} \Longrightarrow N_A = \frac{T_{fix}}{2t_a}.$$

This results in the speedup

$$S_A(P) = \frac{T_{fix}}{T_{fix}/P + \log_2 P(t_m + t_a)} = \frac{P}{1 + P\log_2 P\frac{t_m+t_a}{T_{fix}}}.$$

Scalar Product: Gustafson Scaling

Gustafson scaling $T_P(N(P),P) = T_{fix}$:

$$\frac{N}{P}2t_a + \log_2 P(t_m + t_a) = T_{fix} \Longrightarrow N_G(P) = P\left(\frac{T_{fix} - \log_2 P(t_m + t_a)}{2t_a}\right).$$

There is an upper limit to P !
Assuming $T_{fix} \gg \log_2 P(t_m + t_a)$ we get $N_G(P) \approx PT_{fix}/(2t_a)$.

This results in the speedup

$$S_G(P) = \frac{N_G(P)2t_a}{N_G(P)2t_a/P + \log_2 P(t_m + t_a)} = \frac{P}{1 + \log P\frac{t_m+t_a}{T_{fix}}}.$$

Communication overhead is $O(\log P)$ instead of $O(P\log P)$.

Scalar Product: Memory Scaling

Gustafson scaling $M(N(P)) = M_0P$:

$$M(N(P)) = wN = M_0P \Longrightarrow N_M(P) = P(M_0/w).$$

There is no upper limit to P !

This results in the speedup

$$S_M(P) = \frac{N_M(P)2t_a}{N_M(P)2t_a/P + \log_2 P(t_m + t_a)} = \frac{P}{1 + \log P\frac{(t_m+t_a)w}{M_02t_a}}.$$

Same as Gustafson scaling as long as $T_{fix} \gg \log_2 P(t_m + t_a)$.

Scalar Product: Isoefficient Scaling

Isoefficient scaling

$$E(N(P),P) = S(N(P),P)/P = E_0 \Rightarrow S(N(P),P) = E_0 P.$$

Inserting the speedup gives

$$S = \frac{P}{1 + \frac{P \log_2 P}{N} \frac{(t_m + t_a)}{2t_a}} \stackrel{!}{=} E_0 P \Longrightarrow N_I(P) = P \log_2 P \frac{E_0}{1 - E_0} \frac{t_m + t_a}{2t_a}.$$

The resulting speedup is $S(N_I(P), P) = PE_0$.

11.3 Things to Remember

What you should remember

- Basic performance measures are speedup and efficiency.
- They are defined relative to the best sequential algorithm.
- Most parallel algorithms scale well if the problem size is increased with the number of processors.

Literatur

[1] Chapter 4 in V. Kumar, A. Grama, A. Gupta and G. Karypis (1994). *Introduction to Parallel Computing*. Benjamin/Cummings.

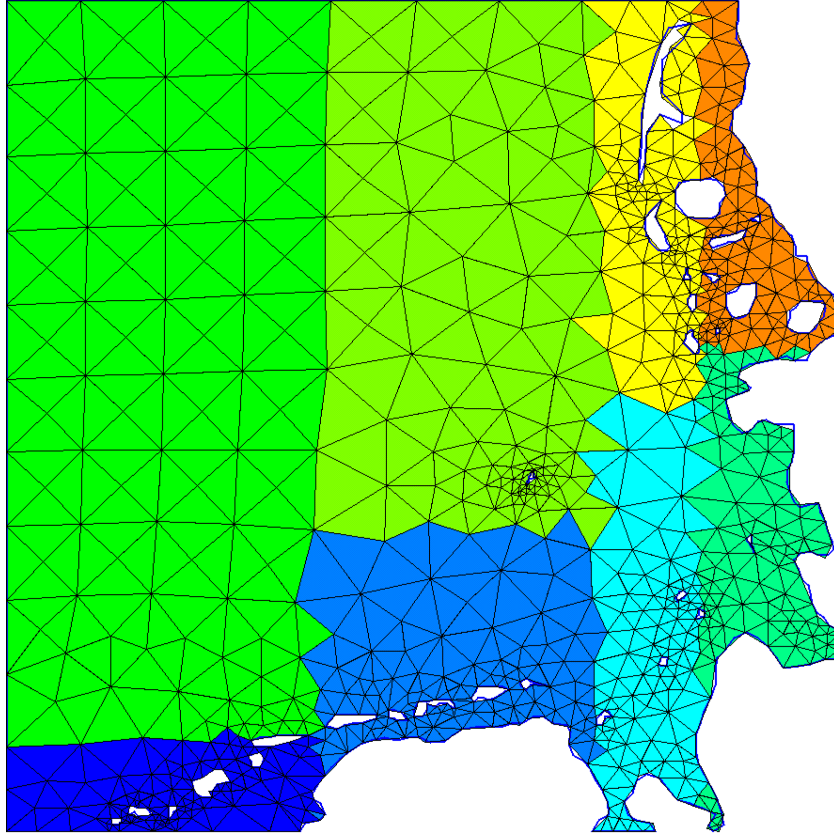
12 Parallel Iterative Solution of Sparse Linear Equation Systems

CG Algorithm

$$\begin{aligned}
&\vec{v} = \vec{d} = \vec{b} - A\vec{x}\\
&d_0 = \vec{d}^T \vec{d};\\
&d_k = d_0\\
&\text{while } (d_k \geq \varepsilon^2 \cdot d_0)\\
&\{\\
&\quad \vec{t} = A\vec{v}\\
&\quad \alpha = d_k / \left(\vec{v}^T \vec{t}\right)\\
&\quad \vec{x} = \vec{x} + \alpha\vec{v}\\
&\quad \vec{d} = \vec{d} - \alpha\vec{t}\\
&\quad d_{k_{\text{old}}} = d_k;\\
&\quad d_k = \vec{d}^T \vec{d}\\
&\quad \beta = d_k / d_{k_{\text{old}}}\\
&\quad v = \vec{d} + \beta\vec{v}\\
&\}
\end{aligned}$$

12.1 Parallelization

Grid Partitioning



Data Decomposition

- The essential operations to be parallelized are:
    1. Matrix vector multiplication $\vec{t} = A\vec{v}$.
    2. Inner products $d_k = \vec{d}^T\vec{d}$ and $\vec{v}^T\vec{t}$
    3. Vector updates $\vec{b} - A\vec{x}$, $\vec{x} = \vec{x} + \alpha\vec{v}$, $\vec{d} = \vec{d} - \alpha\vec{t}$ and $\vec{d} + \beta\vec{v}$.
- Let the index set $I = \{1, \dots, N\}$ be partitioned:

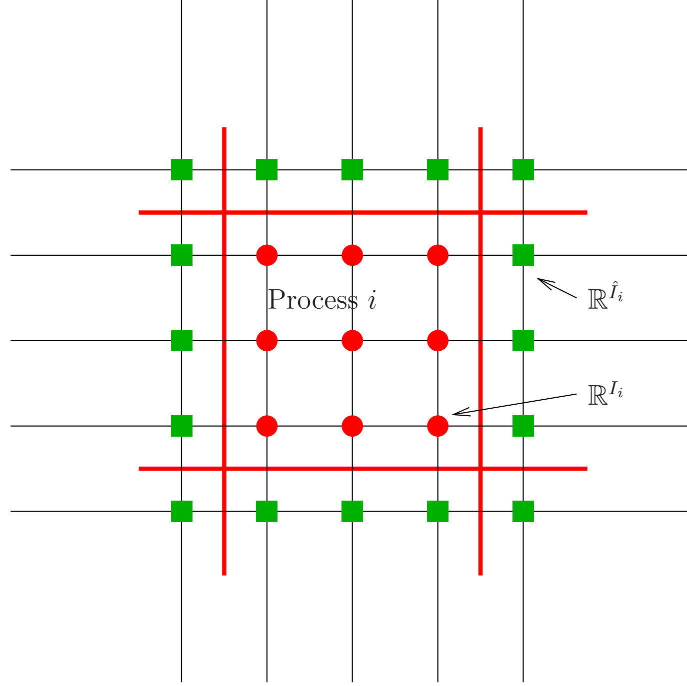
$$I = \bigcup_{i=1}^{P} I_i, \quad I_i \cap I_j = \emptyset \quad \text{for} \quad i \neq j.$$

- Extension of I_i to overlapping decomposition $\hat{I}_i$:

$$\hat{I}_i = \{\delta \in I \mid \exists \gamma \in I_i : (A)_{\gamma,\delta} \neq 0\}.$$

- Introduce vector spaces $\mathbb{R}^I$, $\mathbb{R}^{I_i}$, $\mathbb{R}^{\hat{I}_i}$ and $\mathbb{R}^{\hat{I}_i \times \hat{I}_i}$.

Decomposition for Regular Matrix Graph



Matrix Vector Product

- Vectors $\vec{x}_i, \vec{v}_i, \vec{t}_i, \vec{d}_i \in \mathbb{R}^{\hat{I}_i}$ are with respect to overlapping decomposition.
- Introduce global and local restriction operators:

$$\hat{R}_i : \mathbb{R}^I \to \mathbb{R}^{\hat{I}_i}, \qquad \forall \gamma \in \hat{I}_i : (\hat{R}_i \vec{x})_\gamma = (\vec{x})_\gamma$$

$$R_i : \mathbb{R}^{\hat{I}_i} \to \mathbb{R}^{I_i}, \qquad \forall \gamma \in I_i : (R_i \vec{x}_i)_\gamma = (\vec{x})_\gamma$$

- and the prolongations

$$\hat{R}_i^T : \mathbb{R}^{\hat{I}_i} \to \mathbb{R}^I, \qquad (\vec{x})_\gamma = \begin{cases} \text{if } \gamma \in \mathbb{R}^{\hat{I}_i} \\ 0 \text{ else} \end{cases}$$

$$R_i^T : \mathbb{R}^I \to \mathbb{R}^{\hat{I}_i}, \qquad (\vec{x})_\gamma = \begin{cases} \text{if } \gamma \in \mathbb{R}^{I_i} \\ 0 \text{ else} \end{cases}$$

- Introduce the local matrices $A_i \in \mathbb{R}^{\hat{I}_i \times \hat{I}_i}$:

$$(A_i)_{\gamma,\delta} = \begin{cases} (A)_{\gamma,\delta} & \text{if } \gamma \in \mathbb{R}^{I_i} \\ 1 & \text{if } \gamma = \delta \wedge \gamma \in \mathbb{R}^{\hat{I}_i} \setminus \mathbb{R}^{I_i} \\ 0 & \text{else} \end{cases}$$

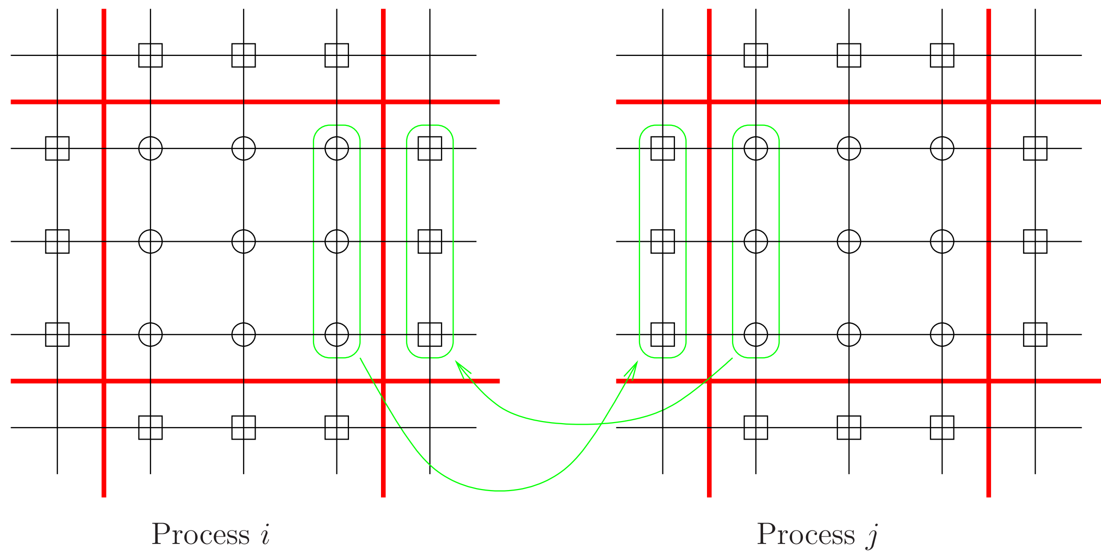
- If $\vec{x}_i = \hat{R}_i \vec{x}$ the matrix vector product can be computed locally on I_i:

$$R_i^T R_i A_i \vec{x}_i = R_i^T R_i \hat{R}_i A \vec{x}.$$

Update and Communication

- For $\vec{x}_i^{(k)}$ we require $\vec{x}_i^{(k)} = \hat{R}_i \vec{x}^{(k)}$, i.e. the global value must be known on the overlapping decomposition.
- The local update would be $\vec{x}_i^{(k+1)} = \vec{x}_i^{(k)} + \alpha \vec{v}_i$
- However, $\vec{v}_i$ is computed on the nonoverlapping decomposition.
- A communication is required to get $\vec{v}_i$ in the overlap region:
$$\forall 1 \leq i \leq P, \forall \gamma \in \hat{I}_i \setminus I_i : \quad (v_i)_\gamma = (v_j)_\gamma \text{ where } \gamma \in I_j \ .$$
- Minimization of the communication requires solution of a *graph partitioning problem.*
- For finite element matrices and reasonable data decompositions the communication involves only local communication with a few neighbors.

Communication for Regular Matrix Graph



Inner Product

- Vectors are stored in overlapping fashion.
- Assuming that for two vectors $\vec{x}, \vec{y}$ we have $R_i \vec{x}_i = R_i \hat{R}_i \vec{x}$ and $R_i \vec{y}_i = R_i \hat{R}_i \vec{y}$ we can compute the inner product via
$$\vec{x}^T \vec{y} = \sum_{i=1}^{P} (R_i \vec{x}_i)^T (R_i \vec{y}_i).$$
or
$$\vec{x}^T \vec{y} = \sum_{i=1}^{P} (R_i^T R_i \vec{x}_i)^T (R_i^T R_i \vec{y}_i).$$

- Obviously, this involves a global communication.

Parallel CG Algorithm

Finally, we arrive at the parallel algorithm for process i:

$$\vec{x}_i = \hat{R}_i \vec{x}; \vec{b}_i = \hat{R}_i b$$
$$\vec{d}_i = R_i^T (R_i \vec{b}_i - R_i A \vec{x}_i)$$
$$\forall \gamma \in \hat{I}_i \setminus I_i : (\vec{d}_i)_\gamma = (\vec{d}_j)_\gamma \text{where } \gamma \in I_j$$
$$\vec{v}_i = \vec{d}_i$$
$$d_0 = \sum_{j=1}^{P} (R_j \vec{d}_j)^T (R_j \vec{d}_j)$$
$$d_k = d_0$$
$$\text{while } (d_k \geq \varepsilon^2 \cdot d_0)$$
$$\{$$
$$\vec{t}_i = R_i^T R_i A_i \vec{v}_i$$
$$\forall \gamma \in \hat{I}_i \setminus I_i : (\vec{t}_i)_\gamma = (\vec{t}_j)_\gamma \text{where } \gamma \in I_j$$

$$\alpha = d_k / \left(\sum_{j=1}^{P} (R_j \vec{v}_j)^T (R_j \vec{t}_j) \right)$$
$$\vec{x}_i = \vec{x}_i + \alpha \vec{v}_i$$
$$\vec{d}_i = \vec{d}_i - \vec{t}_i$$
$$d_{k_{\text{old}}} = d_k$$
$$d_k = \sum_{j=1}^{P} (R_j \vec{d}_j)^T (R_j \vec{d}_j)$$
$$\beta = d_k / d_{k_{\text{old}}}$$
$$v = \vec{d} + \beta \vec{v}$$
$$\}$$
$$\vec{x} = \sum_{i=1}^{P} \hat{R}_i^T \vec{x}_i$$

Speedup of one Iteration

- Assuming a regular matrix graph with N nodes (total) we get the sequential execution time

$$T_S(N) = N t_{t_{op}}$$

.

- The parallel execution time is

$$T_P(N,P) = \underbrace{\frac{N}{P}t_{op}}_{\text{computation}} + \underbrace{\left(t_s + t_w\left(\frac{N}{P}\right)^{\frac{d-1}{d}}\right)2d}_{\text{local comm.}} + \underbrace{2(t_s+t_w)\log_2 P}_{\text{global comm.}}.$$

- Which results in the speedup

$$S(N,P) = \frac{P}{1 + \left(\frac{P}{N}\right)^{\frac{1}{d}}\frac{2dt_w}{t_{op}} + \frac{P}{N}\frac{2dt_s}{t_{op}} + \frac{2P\log_2 P}{N}\frac{t_s+t_w}{t_{op}}}.$$

- Surface to volume effect, good scalability for fixed N/P.

12.2 MPI Functions for Cartesian Grids

Collecting the Result to all Processes

```
int MPI_Allreduce(void* sendbuf, void* recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)

void Intracomm::Allreduce(const void* sendbuf, void* recvbuf,
                          int count, const Datatype& datatype,
                          const Op& op) const
```

The `Allreduce` function performs a Reduce operation and redistributes the result to all involved processors.

Calculate Distribution of Processes

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)

void Compute_dims(int nnodes, int ndims, int dims[])
```

The `Dims_create` function calculates a suitable `ndims`-dimensional distribution of `nnodes`. If an element of `dims` is nonzero, than the number of nodes in this direction is fixed to this value. The result is returned in `dims`.

Create Cartesian Communicator

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
             int *dims, int *periods,
             int reorder, MPI_Comm *comm_cart)

Cartcomm Intracomm::Create_cart(int ndims,
             const int dims[], const bool periods[],
             bool reorder) const
```

`Cart_create` creates a new communicator where the nodes are positioned on a `ndims`-dimensional grid where `dims` gives the number of nodes in each direction. The elements of `periods` determine if the grid is periodic in this dimension. If `reorder` is true, the ranks in the new communicator may be renumbered.

Getting n-d Position of Process

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)

void Cartcomm::Get_coords(int rank, int maxdims,
                    int coords[]) const
```

The function Cart_coords returns the coordinates of the process with rank rank in a maxdims-dimensional Cartesian communicator in the array coords.

Getting Rank of the Neighbour

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
           int disp, int *rank_source, int *rank_dest)

void Cartcomm::Shift(int direction, int disp,
           int& rank_source, int& rank_dest) const
```

Cart_shift returns the rank of the current process in this communicator rank_source and the rank of the process, which is disp positions shifted along direction direction.

Creating Packet Data Formats

```
int MPI_Type_vector(int count, int blocklength,
           int stride, MPI_Datatype oldtype,
           MPI_Datatype *newtype)
int MPI_Type_commit(MPI_Datatype *datatype)

Datatype Datatype::Create_vector(int count,
           int blocklength, int stride) const
Datatype::Commit()
```

Often a number of variables has to be transported, which is not necessarily a direct sequence in a vector. MPI allows the definition of new data types, which pack this variables in a new vector. The vector contains count-times blocklength elements where each of this blocks is separated by stride. Before the datatype can be used the Commit function is called.

Simultaneous Send and Receive

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvtype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)

void Comm::Sendrecv(const void *sendbuf, int sendcount,
        const Datatype& sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount, const Datatype& recvtype,
        int source, int recvtag, Status& status) const
void Comm::Sendrecv(const void *sendbuf, int sendcount,
        const Datatype& sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount, const Datatype& recvtype,
        int source, int recvtag) const
```

For the communication in the grid, nearly all processes need to send as well as receive. To prevent the possibilities for deadlocks, MPI offers a function which defines a simultaneous send

and receive process, which is free of any deadlocks. The number of sent and received objects does not necessarily have to match.

12.2.1 Examples

Subdividing the Grid

```
int numProcs=MPI::COMM_WORLD.Get_size();
int dims[DIM];
for (unsigned int i=0;i<DIM;++i)
    dims[i]=0;
MPI::Compute_dims(numProcs,DIM,dims);
bool periods[DIM];
for (unsigned int i=0;i<DIM;++i)
    periods[i]=false;
MPI::Cartcomm comm=MPI::COMMWORLD.Create_cart(DIM,dims,periods,true);

globalNx = globalNx - nx  % dims[0];
globalNy = globalNy - ny % dims[1];
size_t nx=globalNx/dims[0];
size_t ny=globalNy/dims[1];
size_t NX=nx+2;
size_t NY=ny+2;
```

Creation of Datatypes

```
MPI::Datatype columnType=MPI::DOUBLE.Create_vector(1,ny,ny);
columnType.Commit();
MPI::Datatype rowType=MPI::DOUBLE.Create_vector(nx,1,ny+2);
rowType.Commit();

comm.Get_coords(comm.Get_rank(), DIM, &cartcoords[0]);
size_t offsetX=cartcoords[0]*nx;
size_t offsetY=cartcoords[1]*ny;
size_t start = 1 + NY;

typedef struct direction_t
{
    int sendRank;   // ID of the sending process
    int sendOffset; // Index of first variable to send
    int recvRank;   // ID of the receiving process
    int recvOffset; // Index of first variable to receive
    MPI::Datatype type;
} direction_t;
direction_t direction[2*DIM];
```

Determining the Positions of the Values to Communicate

```
comm.Shift(0, +1, direction[WEST].sendRank, direction[WEST].recvRank);
direction[WEST].type = columnType;
direction[WEST].sendOffset = start;
direction[WEST].recvOffset = start+NY*nx;

comm.Shift(0, -1, direction[EAST].sendRank, direction[EAST].recvRank);
direction[EAST].type = columnType;
direction[EAST].sendOffset = start+NY*(nx-1);
direction[EAST].recvOffset = 1;

comm.Shift(1, +1, direction[NORTH].sendRank, direction[NORTH].recvRank);
direction[NORTH].type = rowType;
direction[NORTH].sendOffset = start;
```

```
direction[NORTH].recvOffset = start+ny;

comm.Shift(1, -1, direction[SOUTH].sendRank, direction[SOUTH].recvRank);
direction[SOUTH].type = rowType;
direction[SOUTH].sendOffset = start+ny-1;
direction[SOUTH].recvOffset = start-1;
```

Communicating the Neighbour Values

```
for (size_t i = 0; i < numNeighbors; ++i)
{
    comm.Sendrecv(&x[direction[i].sendOffset], 1, direction[i].type,
                  direction[i].sendRank, TAGNEIGHBORS,
                  &x[direction[i].recvOffset], 1, direction[i].type,
                  direction[i].recvRank, TAGNEIGHBORS);
}
```

Sending the Result to the Root Process

```
// rectangular section of the world
MPI::Datatype sectionType=MPI::DOUBLE.Create_vector(nx,ny,ny+2);
sectionType.Commit();

// Send results to root process
comm.Isend(&x[NY+1], 1, sectionType, 0, TAGDISTGATHER);
```

Collecting the Result in the Root Process

```
MPI::Datatype subMatrixType=MPI::DOUBLE.Create_vector(nx,ny,globalNy);
subMatrixType.Commit();

if (comm.Get_rank() == 0)
{
    int xy[2], rank;
    for (xy[1] = 0; xy[1] < globalNy/ny; ++xy[1])
    {
        for (xy[0] = 0; xy[0] < globalNx/nx; ++xy[0])
        {
            rank=comm.Get_cart_rank(xy);
            comm.Recv(&world[xy[0] * globalNy * nx + xy[1] * ny],
                      1, subMatrixType, rank, TAGDISTGATHER);
        }
    }
}
```

Renumbering of Vector Indices

RowIndex is just the ordinary index, but for the locally stored data (ny = globalNy/dimY). ColIndex is the index including the copied boundary elements (ny = globalNy/dimY + 2). RowToColumnIndex is a usefull function to convert from one to the other. For example a check for the diagonal element can be performed by `if (j==RowToColumnIndex(i))`.

```
size_t RowIndex(int i, int j) const
{
    return i*ny+j;
}

size_t ColumnIndex(int i, int j) const
{
    return (i+1)*NY+j+1;
```

```
}

size_t RowToColumnIndex(size_t i) const
{
    return (i/ny+1)*NY+(i%ny)+1;
}

index[RowIndex(i,j)]=ColumnIndex(i,j);
```

13 Debugging of Parallel Programs

Debugging parallel programs is a complicate task. Possible tools are:

- Using `printf` or `std::cout`
- Writting to log files
- Using `gdb`
- Using specialised debuggers for parallel programs

Using `printf` or `std::cout`

```
std::cout << MPI_COMM_WORLD.get_rank() << ": a = " << a << std::endl;
```

- Output should be prefixed with the rank of the processor
- Output of different processes is mixed and written by the root process.
- The order of output from different processes must not be chronological
- If job terminates some output may never be written

Writing to log-files

```
    template<class T>
    void DebugOut(const std::string message, T i) const
    {
        std::ostringstream buffer;
        buffer << "debugout" << myRank_;
        std::ofstream outfile(buffer.str().c_str(),std::ios::app);
        outfile << message << " " << i<< std::endl;
    }
```

- Output from each processor goes to a separate file
- Output is complete for each processor (as file is always immediately closed afterwards)
- File is allways appended not erased ⇒ output of several runs can be mixed

Using gdb

Several instances of gdb can be started each attaching to one of the parallel processes using

```
gdb <program name> <PID>
```

To make sure that all processes are still at the beginning you can add an infinite loop

```
1 bool debugStop=true;
  while (debugStop);
```

After gdb has attached the loop can be exited by using the gdb-command

```
set debugStop=true;
```

You may have to compile with

```
1 mpicxx -o <program name> -g -O0 <program source>
```

as the compiler may optimise the variable away if the option -O0 is absent.

Parallel Debuggers

- Parallel debuggers are providing a graphical user interface for the process describe before
- Commercial parallel debuggers are e.g. Totalview and DDT
- There is also a eclipse plugin available: PTP

14 Time-dependent Problems

14.1 Parabolic Problems

Transport Equation

$$\begin{aligned} \frac{\partial \theta(\vec{x})}{\partial t} + \nabla \cdot \vec{J}_w(\vec{x}) + r_w(\vec{x}) &= 0 \\ \frac{\partial \theta(\vec{x})}{\partial t} + \nabla \cdot \left[-\bar{K}_s(\vec{x}) \cdot \nabla p_w\right] + r_w(\vec{x}) &= 0 \\ \frac{\partial \theta(\vec{x})}{\partial t} - \nabla \cdot \left[\bar{K}_s(\vec{x}) \cdot \nabla p_w\right] + r_w(\vec{x}) &= 0 \end{aligned}$$

with gravity:

$$\frac{\partial \theta(\vec{x})}{\partial t} - \nabla \cdot \left[\bar{K}_s(\vec{x}) \cdot (\nabla p_w - \rho_w g \vec{e}_z)\right] + r_w(\vec{x}) = 0$$

steady state:

$$-\nabla \cdot \left[\bar{K}_s(\vec{x}) \cdot (\nabla p_w - \rho_w g \vec{e}_z)\right] + r_w(\vec{x}) = 0$$

Method of Lines

First discretise equations in space:

$$\frac{\partial \left(C(\vec{x}) \cdot T(\vec{x})\right)}{\partial t} = \nabla \cdot [\lambda(\vec{x}) \cdot \nabla T] - r_h(\vec{x})$$

$$\begin{aligned}
\frac{\partial \left(C(\vec{x}_{i,j}) \cdot T(\vec{x}_{i,j}, t)\right)}{\partial t} \quad = \quad & \underbrace{\begin{aligned} & \lambda_{xx}(\vec{x}_{i-0.5,j}, t) \cdot (T(\vec{x}_{i,j}, t) - T(\vec{x}_{i-1,j}, t)) \\ & -\lambda_{xx}(\vec{x}_{i+0.5,j}, t) \cdot (T(\vec{x}_{i+1,j}, t) - T(\vec{x}_{i,j}, t)) \\ & +\lambda_{yy}(\vec{x}_{i,j-0.5}, t) \cdot (T(\vec{x}_{i,j}, t) - T(\vec{x}_{i,j-1}, t)) \\ & -\lambda_{yy}(\vec{x}_{i,j+0.5}, t) \cdot (T(\vec{x}_{i,j+1}, t) - T(\vec{x}_{i,j}, t)) \\ & -h^2 r(\vec{x}_{i,j}, t) \end{aligned}}_{:=F(\vec{x}_{i,j}, t)}
\end{aligned}$$

Integration in Time

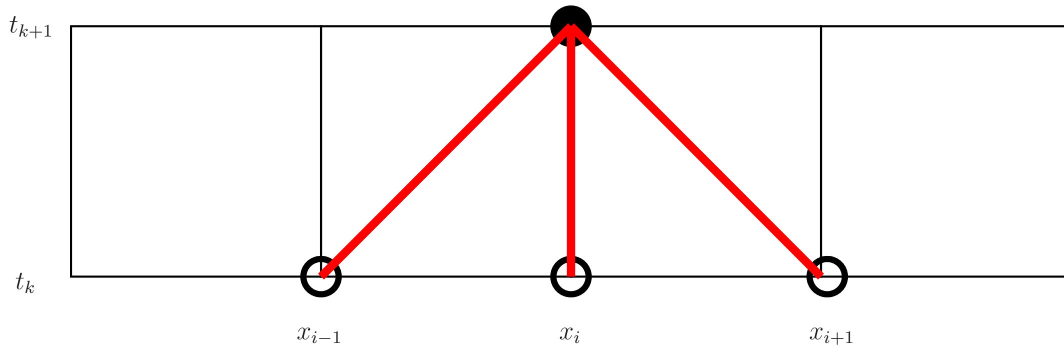
$$\int_{t_k}^{t_{k+1}} \frac{\partial \left(C(\vec{x}_{i,j}) \cdot T(\vec{x}_{i,j}, t)\right)}{\partial t} = \int_{t_k}^{t_{k+1}} F(\vec{x}_{i,j}, t)$$

Numerical Integration of the right side yields:

$$C(\vec{x}_{i,j}) \cdot (T(\vec{x}_{i,j}, t_{k+1}) - T(\vec{x}_{i,j}, t_k)) = \Delta t \cdot [(1 - \Theta) \cdot F(\vec{x}_{i,j}, t_k) + \Theta \cdot F(\vec{x}_{i,j}, t_{k+1})]$$

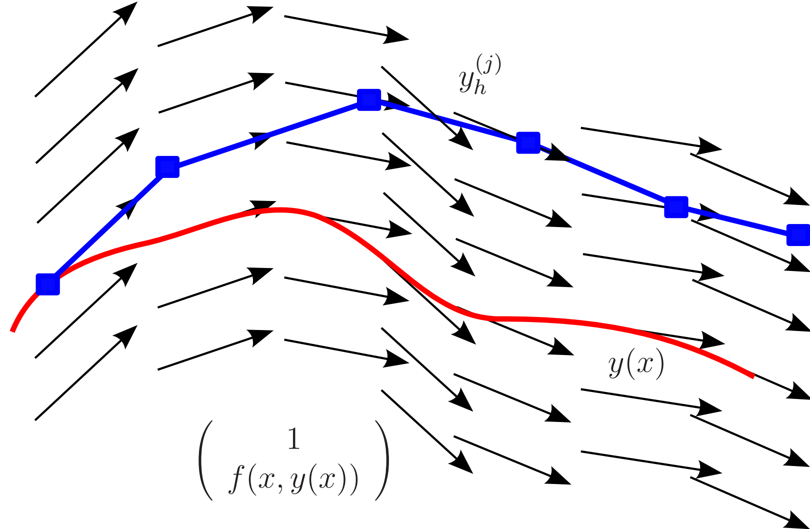
$\Theta = 0.5$: Trapezoidal rule $\Rightarrow$ second-order accurate $\Theta \neq 0.5$: First-order accurate

Explicit Euler Scheme ($\Theta = 0$)

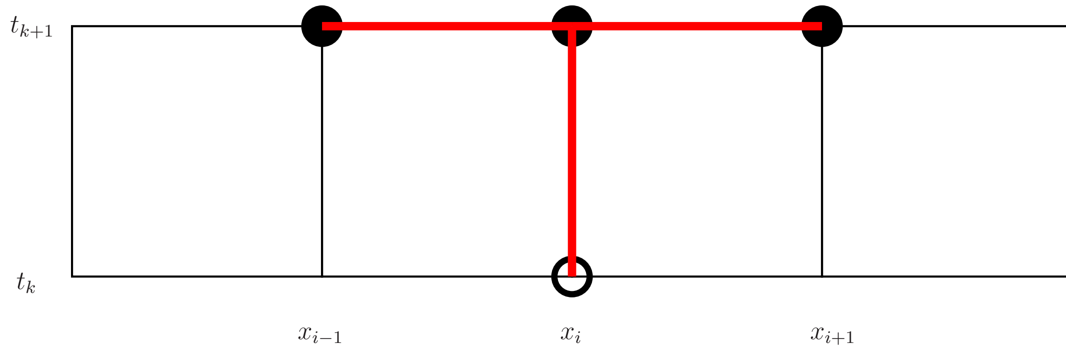


- The explicit Euler or forward Euler scheme yields a diagonal matrix. Therefore the solution of a linear equation system is not necessary.
- Only values of the old time are used
- It is first order accurate
- It is only stable if $\Delta t < \frac{C}{2\lambda} \cdot h^2$

Explicit Euler Scheme (II)

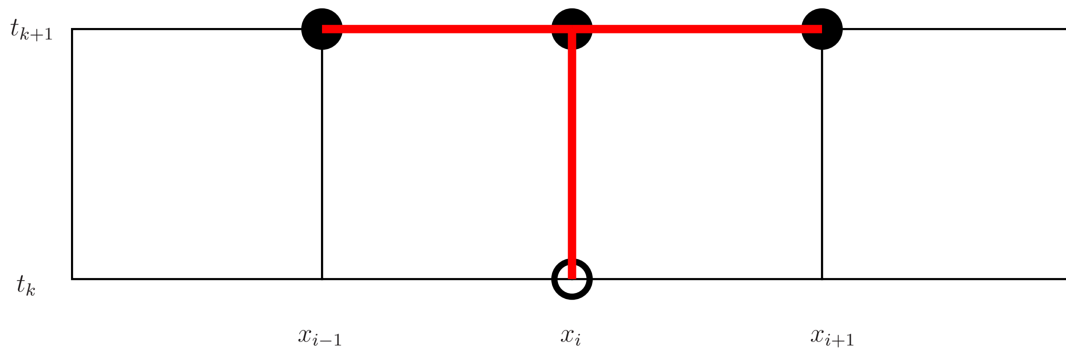


Implicit Euler Scheme ($\Theta = 1$)



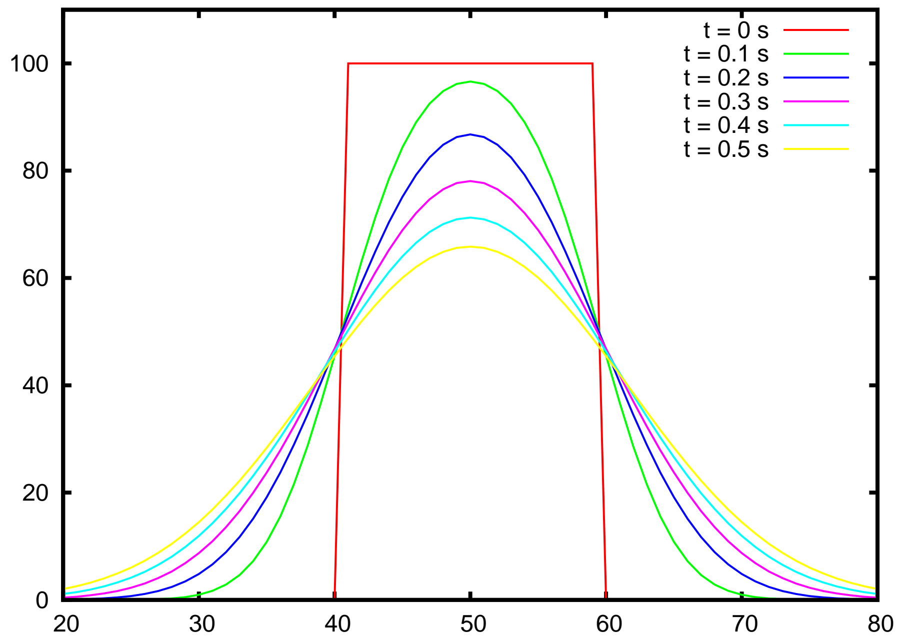
- The implicit Euler scheme requires the solution of a linear equation system.
- Primarily values of the new time are used
- It is first order accurate
- It is unconditionally stable

Crank-Nicolson Scheme ($\Theta = 0.5$)

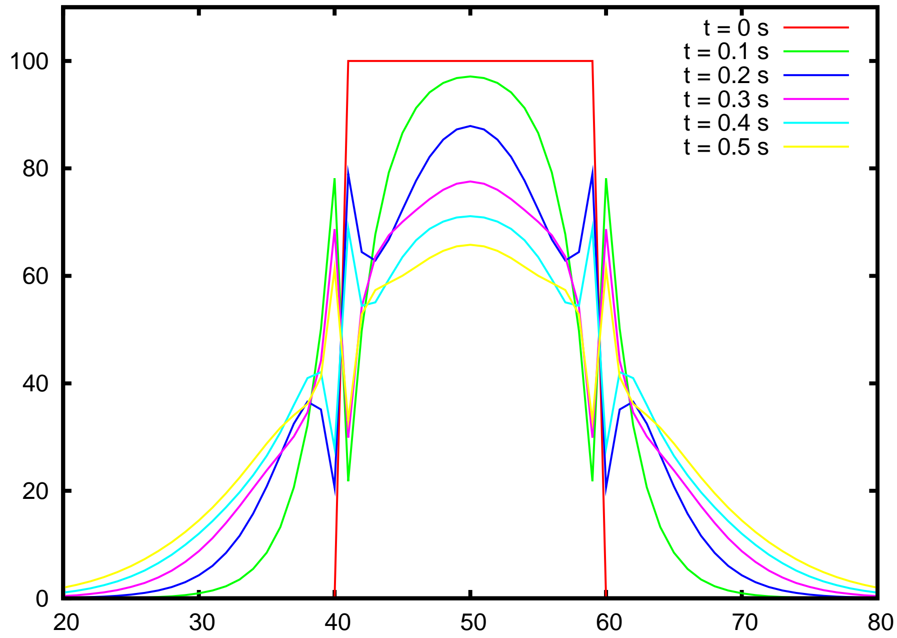


- The Crank-Nicolson scheme requires the solution of a linear equation system.
- Both values of the new and the old time are used
- It is second order accurate
- It is unconditionally stable in the $||\dots||_2$ norm. However oscillations can occur if $\Delta t \geq \frac{C}{\lambda}h^2$

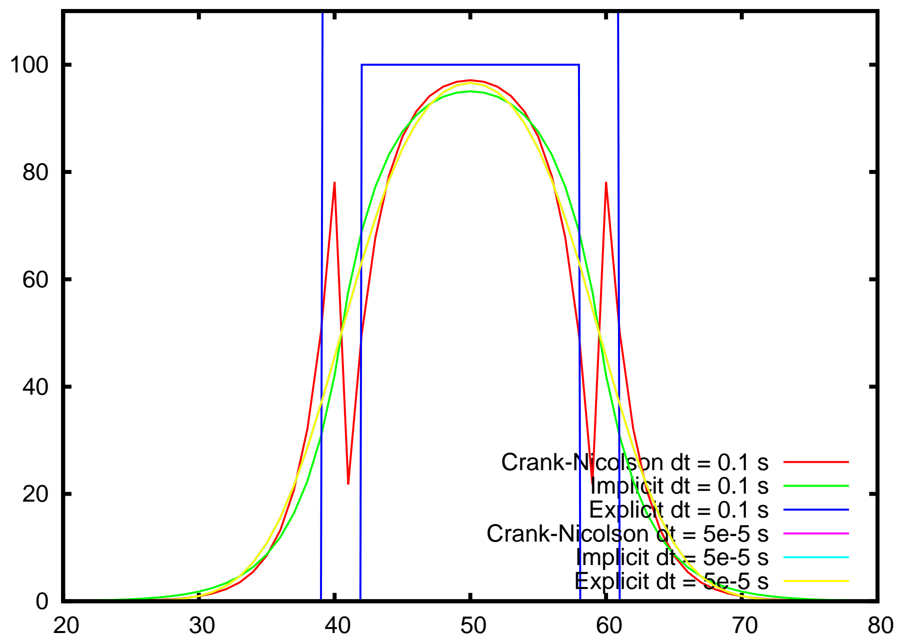
Non-Smooth Initial Condition Crank-Nicolson $\Delta t = 5 \cdot 10^{-5}\ s$



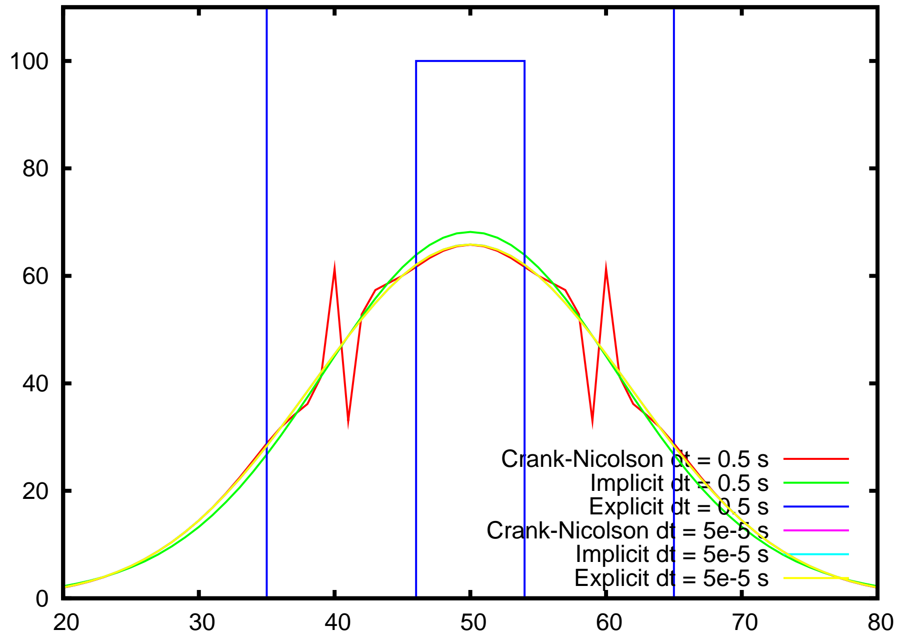
Non-Smooth Initial Condition Crank-Nicolson $\Delta t = 0.1\ s$



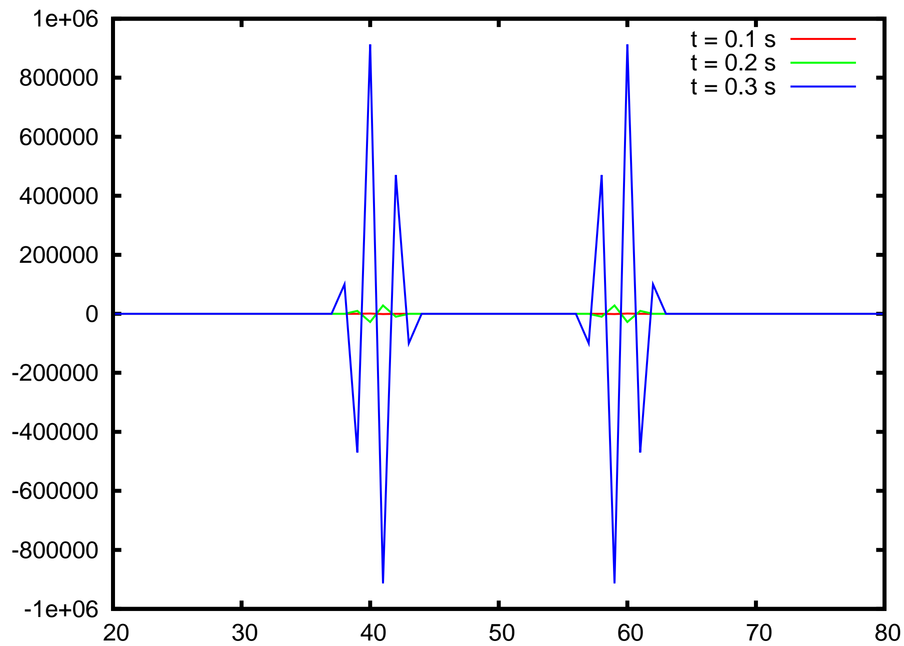
Non-Smooth Initial Condition Different Schemes $t = 0.1\ s$



Non-Smooth Initial Condition Different Schemes $t = 0.5\ s$



Non-Smooth Initial Condition Explicit Euler $dt = 0.1\ s$



Darcy2D::SetupSystem2DTime

```
double alpha=1.0;
#ifdef CRANK_NICOLSON
alpha=0.5;
#elif defined(EXPLICIT)
alpha=0.;
#endif
```

```
// diagonal coefficient and right hand side
A(me,A.ColumnIndex(i,j)) = alpha*(k_w + k_e + k_n + k_s)*dt+C_;
b[me] = (Source(x,y)*hx_*hy_+bflux)*dt+C_*x0[me];
b[me] -= (1.-alpha)*(k_w + k_e + k_n + k_s)*dt*x0[me];

// west coefficient
if (i+offsetX_>0)
{
    A(me,A.ColumnIndex(i-1,j))= - alpha * k_w * dt;
    b[me] += (1.-alpha)*k_w*dt*x0[A.RowIndex(i-1,j)];
}
```

Darcy2D::SetupSystem2DTime

```
// south coefficient
if (j+offsetY_>0)
{
    A(me,A.ColumnIndex(i,j-1))= - alpha * k_s * dt;
    b[me] += (1.-alpha)*k_s*dt*x0[A.RowIndex(i,j-1)];
}

// east coefficient
if (i+offsetX_<(globalNx_-1))
{
    A(me,A.ColumnIndex(i+1,j))= - alpha * k_e * dt;
    b[me] += (1.-alpha)*k_e*dt*x0[A.RowIndex(i+1,j)];
}

// north coefficient
if (j+offsetY_<(globalNy_-1))
{
    A(me,A.ColumnIndex(i,j+1))= - alpha * k_n * dt;
    b[me] += (1.-alpha)*k_n*dt*x0[A.RowIndex(i,j+1)];
}
```

Darcy2D::SetupSystem2DTime

```
void InitX (CRSMatrix &A, std::vector<double> &b,
            std::vector<double> &x0)
{
    x0.resize(nx_*ny_);
    b.resize(nx_*ny_);
    for (int i=0;i<nx_*ny_;++i)
    {
        if ((i*hy_>=0.4)&&(i*hy_<=0.6))
            x0[i]=100.;
        else
            x0[i]=T0_;
    }
}
```

Main Program

```
assembler.InitX(A,b,x);
assembler.Output2D("result0",x);
double dt=1./std::max(nx,ny);
```

```
  dt*=dt/2.;
5 double t=0.;
  double nextOutput=0.1;
7 for (int i=0;t<=0.5 && i<200000000;++i)
  {
9     std::cout << "t␣=␣" << t << std::endl;
      assembler.SetupSystem2DMPITime(A,b,x,dt);
11    #ifdef EXPLICIT
      for (size_t j=0;j<b.size();++j)
13        x[j]=b[j]/A(j,A.RowToColumnIndex(j));
      #else
15    A.SOLVER(x,b,1e-10);
      #endif
17    t+=dt;
      if ((t+dt/2>=nextOutput)&&(MPI::COMM_WORLD.Get_rank()==0))
19    {
          std::ostringstream number;
21        number << t;
          assembler.Output2D("result"+number.str(),x);
23        assembler.WriteDXHeader("result"+number.str());
          nextOutput+=0.1;
25    }
  }
```

Summary Groundwater Flow and Partial Differential Equations

- Groundwater Flow
- Classification of Partial Differential Equations
- Discretisation schemes for partial differential equations
- The cell-centered finite-volume method
- The vertex-centered finite-volume method
- Time-dependent Problems / Parabolic Equations

Summary Iterative Solution of Linear Equation Systems

- Basic solvers for linear equation systems
- Advanced solvers for linear equation systems
- Parallel Iterative Solvers

Summary Parallel computing

- Architectures of Parallel Computers
- Shared Memory Parallel Computing
- Basics of Parallel Programming

- Message Parsing Interface (MPI)
- Analysis of Parallel Algorithms
- Parallel Iterative Solvers