Simulation on High-Performance Computers

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 15/16

Organisational Information

- Course title: Parallel High Performance Computing
- Lecturer: Stefan Lang, Scientific Computing, IWR, Room 425
- Course extend: 4h lectures + 2h exercises, CP: 8
- In total 30 lectures and 12 exercise sheets
- Dates
  - Lectures: Tu 9.00-11.00 (V R532), Th 9.00-11.00 (V R532)
  - Exercises: Th 14.00-16.00 (E in CIP Pool, OMZ, INF 350 U.012)
- Prerequisites:
  Basic lectures in Computer Science and Numerics
- Helpful:
  Knowledge of C/C++
- Literature: see course homepage
  http://conan.iwr.uni-heidelberg.de/teaching/phlr_ws2015/index.html

What means Scientific Computing?

Scientific Computing is a rapdily growing **novel discipline of science**

In particular Numerical Simulation (NS):

- Goal of NS is to **simulate natural or technical processes** with computing machines
- Approach accross disciplines: natural scientists, engineers, mathematicians and computer scientists, **all need to work together**
- Problems, that are relevant in practice, are handled **systematically with formal methods**
- NS enables **insight into areas that are difficult to access** in lab experiments and field studies, for example neuroscience, cell biology, water economics, astrophysics

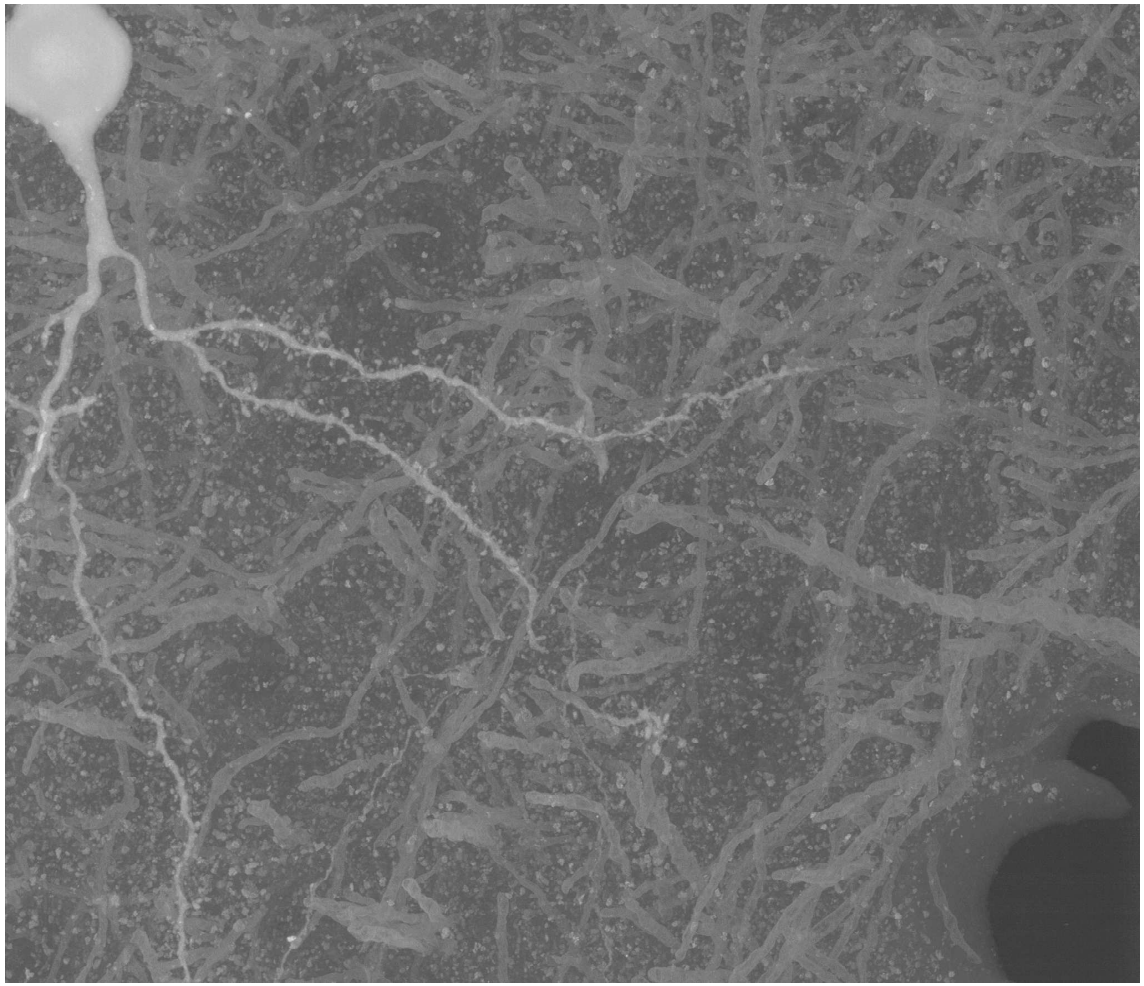
Why Scientific High-Performance Computing?

Trends in **Numerical simulation**:

- Treatment of **global models** instead of local partial models
  (Models of entire brain areas, Complete water resource systems , virtual prototyping in aircraft, naval and automobile design)
- Analysis of **coupled overall systems** instead of isolated individual processes
  (multi-media, multi-phase, multi-scale processes, convection-diffusion-reaction)
- Computer gets a **scientific observation instrument**
  (high resolution capacity,
  measurement in some areas difficult/impossible,
  parameter studies performable)

Computational Neuroscience - Signal Processing

- Goal: Development of a neuronal network model, that reflects an observed or measured system behaviour



- Simulation of neuronal network models, statistical analysis of realisations

A Supercomputer

ASCI Red Storm
Successor of the first TeraFlop computer ASCI Red 1997
Hardware:

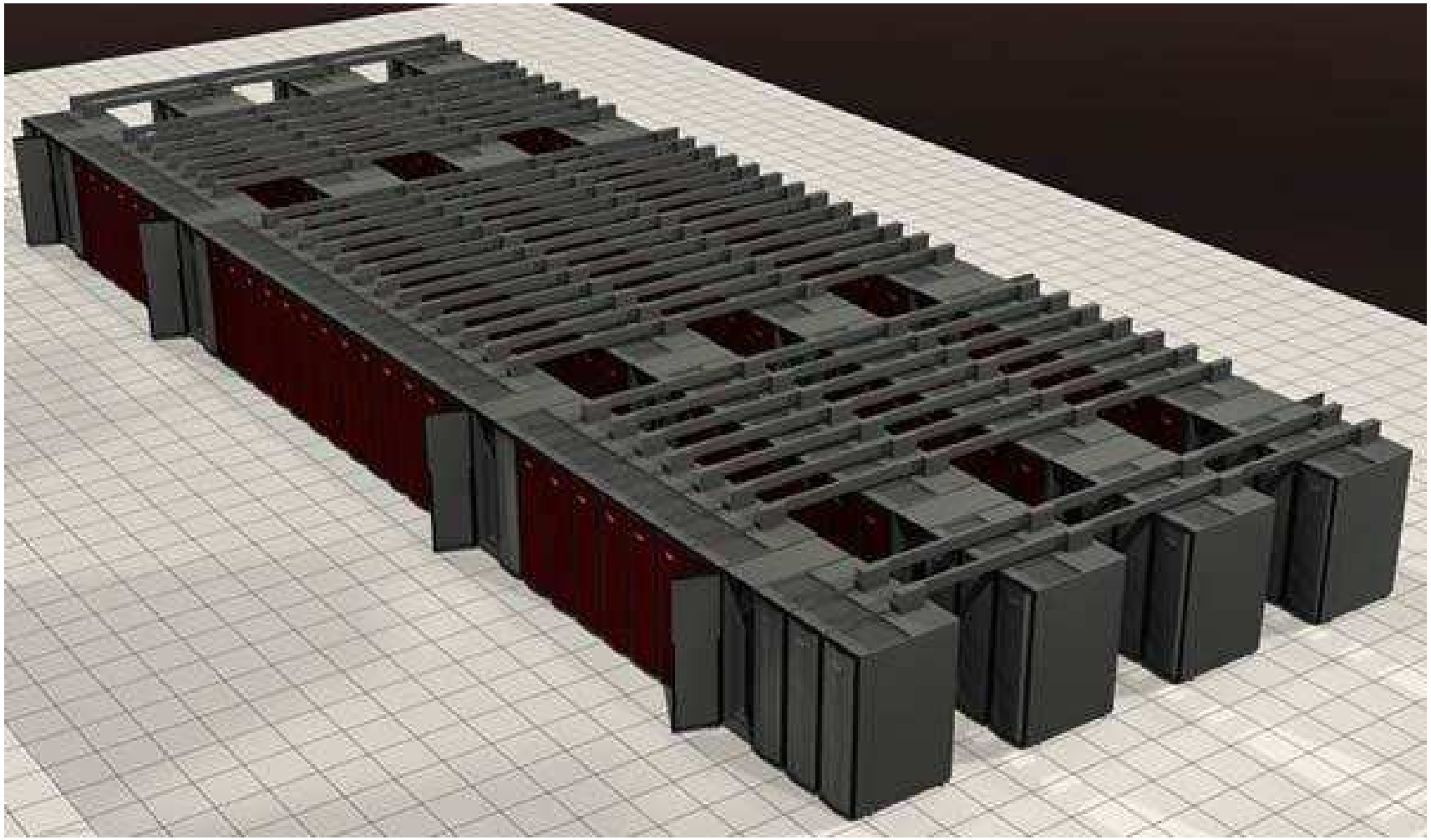
- 11.646 $\times$ 2 GHz AMD Opteron CPUs
- CPU boards vertically mounted in 108 cabinets
- 4 GFLOPS per CPU (40 TFLOPS total)
- 1 GB per CPU (10 TB total)
- Shared memory inside the node
- 3D mesh full interconnect

Software:

- Compute nodes: custom Sandia-developed light-weight OS code-named Catamount
- Service and storage nodes SuSE Linux.

A Supercomputer

ASCI Red Storm



Scalability

Algorithmic complexity using the example of linear solvers

for an equation of the form

$$Ax = b$$

Dimension	$d = 2$	$d = 3$
Gaussian elimination	$O(N^3)$	$O(N^3)$
Banded Gauss	$O(N^2)$	$O(N^{2.33})$
Nested Disection Gauss	$O(N^{1.5})$	$O(N^2)$
Richardson, GS, Jacobi	$O(N^2)$	$O(N^{1.67})$
CG, SOR	$O(N^{1.5})$	$O(N^{1.33})$
SSOR–CG	$O(N^{1.25})$	$O(N^{1.17})$
Multigrid	$O(N)$	$O(N)$
Cascadic Iteration	$O(N)$	$O(N)$

Scalability

Scalability means in simple words
A problem of steadily increasing size can be calculated on a steadily increasing computer (nearly) equally fast.

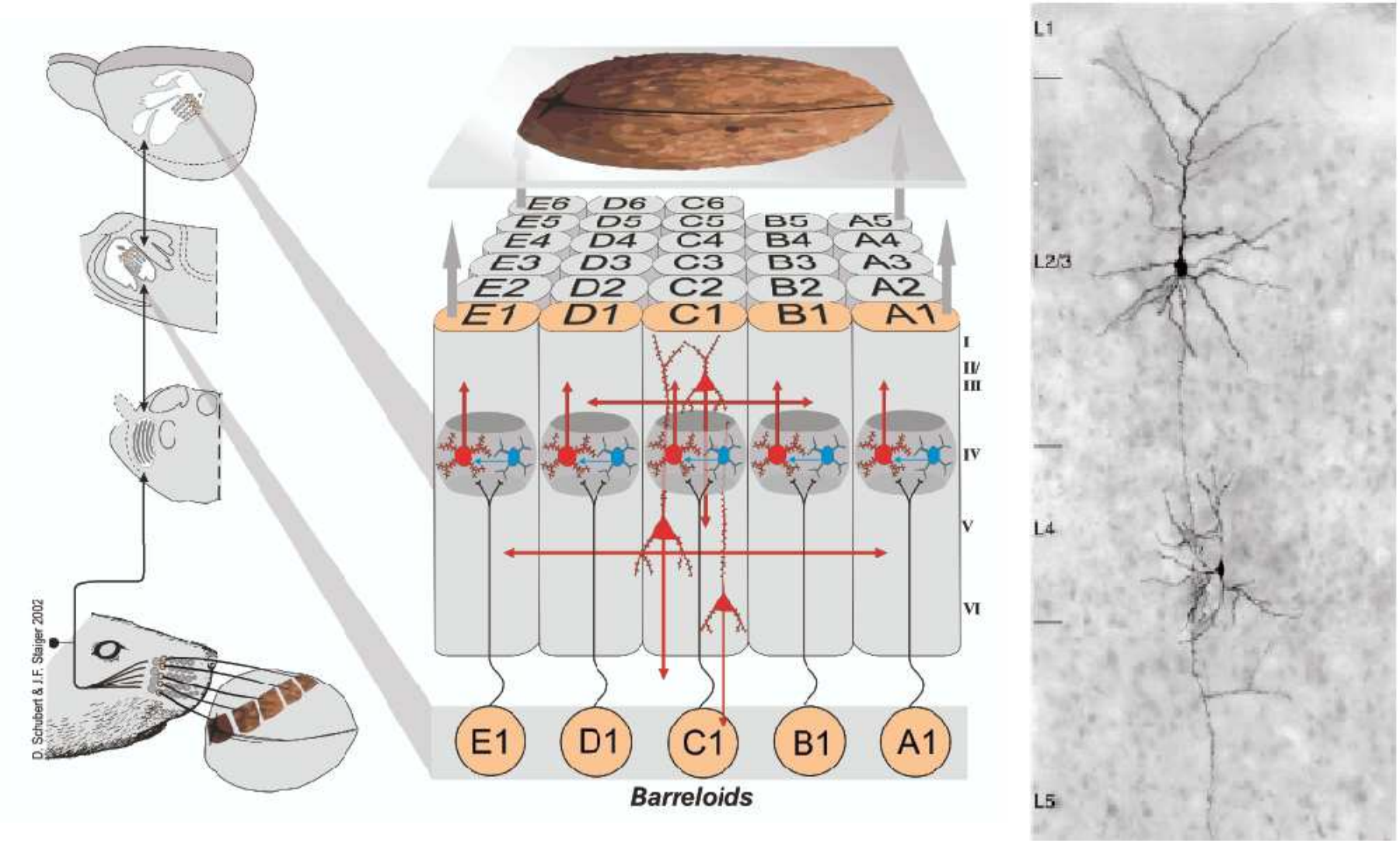
This is seldom the case (unfortunately)!

As a consequence we need

- **enabling software** to handle such powerful computers
- **scalable algorithms + scalable implementations + scalable architectures**

. . . buying huge and expensive computers alone is not enough!!

A Biological System: The Barrel Cortex of the Rat



Goal: Mechanistic understanding of easy decision making

Modeling of Neuronal Activity

Hodgkin-Huxley equation:

The electric potential $v(\mathbf{x}, t)$ and gating particles $\mathbf{c}(\mathbf{x}, t) = (m(\mathbf{x}, t), h(\mathbf{x}, t), n(\mathbf{x}, t))^T$ obey

$$c_m \partial_t v = \partial_{\mathbf{x}} g_a \partial_{\mathbf{x}} v + i_{inj} - \sum_{\nu \in \mathcal{C}} i_\nu(v, \mathbf{c}) - i_s(v)$$

$$\partial_t c_\mu = \alpha_\mu(v) \cdot (1 - c_\mu) \; - \; \beta_\mu(v) \cdot c_\mu,$$

with $\nu \in \mathcal{C} =: \{Na, L, K\}$ and $\mu \in \{m, h, n)$. Boundary and initial conditions are given by

$$\begin{aligned} g_a \partial_{\mathbf{x}} v &= g_N \qquad \text{on } \partial\Omega_N, \\ v &= g_D \qquad \text{on } \partial\Omega_D, \\ (v, \mathbf{c})(\mathbf{x}, 0) &= (v^0, \mathbf{c}^0) \text{ for } t = 0. \end{aligned}$$

The ion currents are modeled with the gating particles and are given by

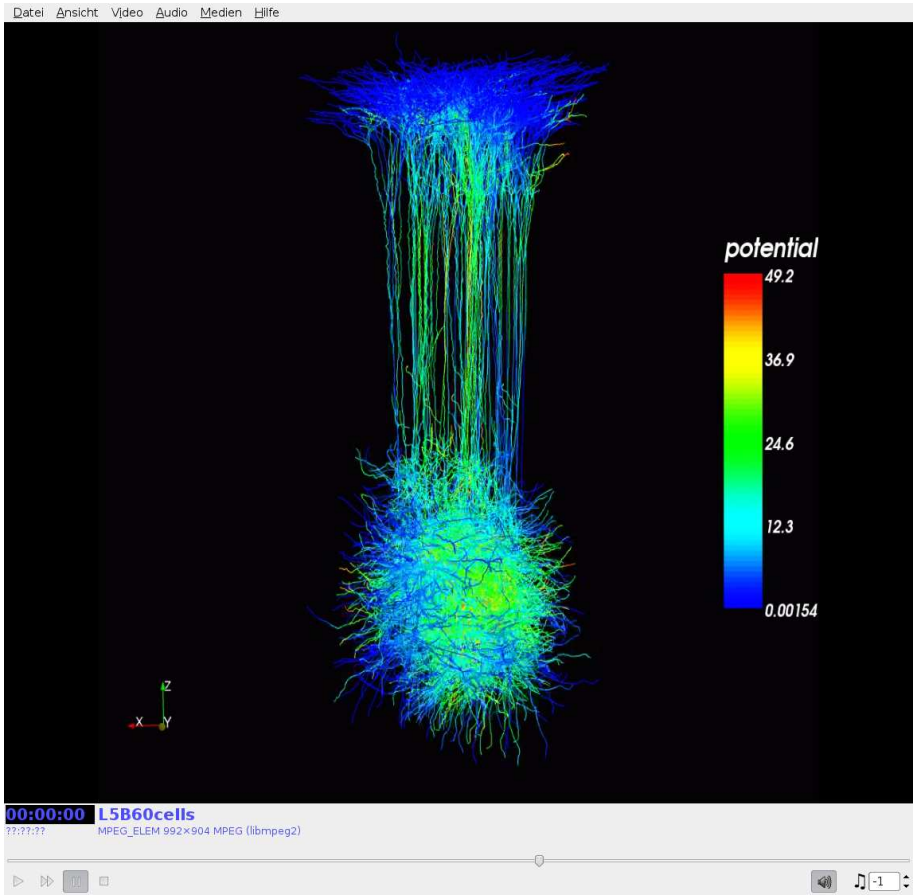
$$\begin{aligned} i_{Na}(v) &= \overline{g_{Na}} \cdot m^3 h \cdot (v - E_{Na}), \\ i_K(v) &= \overline{g_K} \cdot n^4 \cdot (v - E_K), \\ i_L(v) &= \overline{g_L} \cdot (v - E_L) \end{aligned}$$

Moreover currents of synapses are modeled by

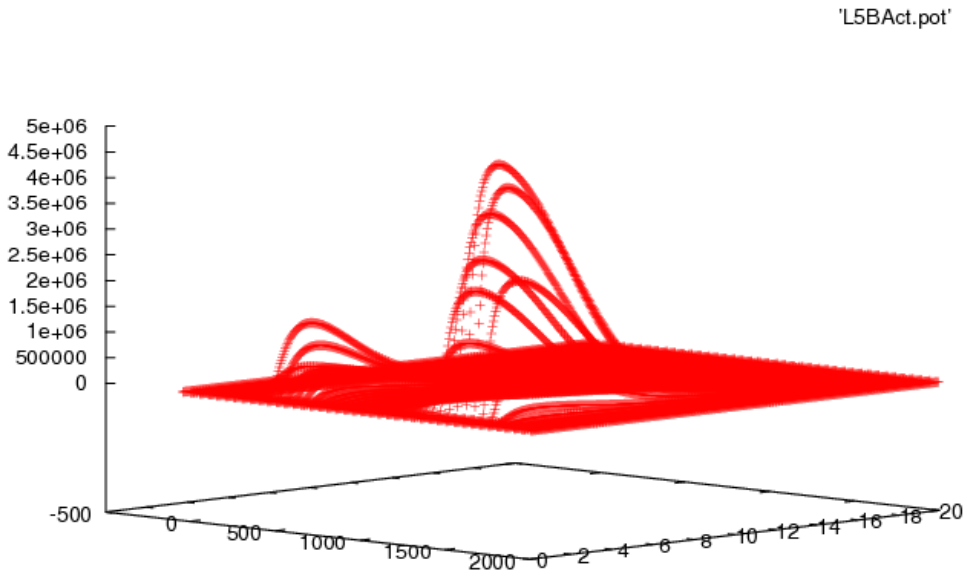
$$i_s(v, t) = g_s(t) \cdot (v - E_s)$$

with time-dependent synaptic strength g_s.

Simulation Study: Passive Deflection of a Whisker

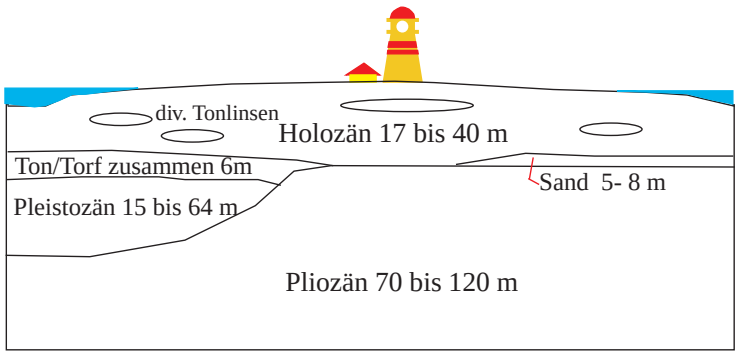


60 L5B neurons activated by VPM



Spatial and temporal activity distribution

Problem study Norderney



- Project "coastal preservation"
- Island typical fresh water lense (-85m)
- Simulation of lense constitution and water facilitation from two pumps
- Strategy to preserve the water quality

Problem study Norderney

10km x 4km
x 150m

- Initial grid (1516 elements, D. Feuchter, geomod2ng)
- 6 geological layers with varying permeability $10^{-10} - 10^{-15}$
- Boundary conditions: influence of fresh water on upper boundary wells are sinks, in/outflow in coastal areas, hydrostatic pressure

Density-driven Groundwater Flow

The equations of density-driven flow are derived from conservation laws.
Formulation uses **salt mass fraction** ω and **pressure** p

$$\partial_t(n\rho) + \nabla \cdot (\rho \mathbf{v}) = Q\rho, \qquad \text{(f low)}$$
$$\partial_t(n\rho\omega) + \nabla \cdot (\rho \mathbf{v}\omega - \rho D \nabla \omega) = Q\rho\omega \qquad \text{(transp.)}$$

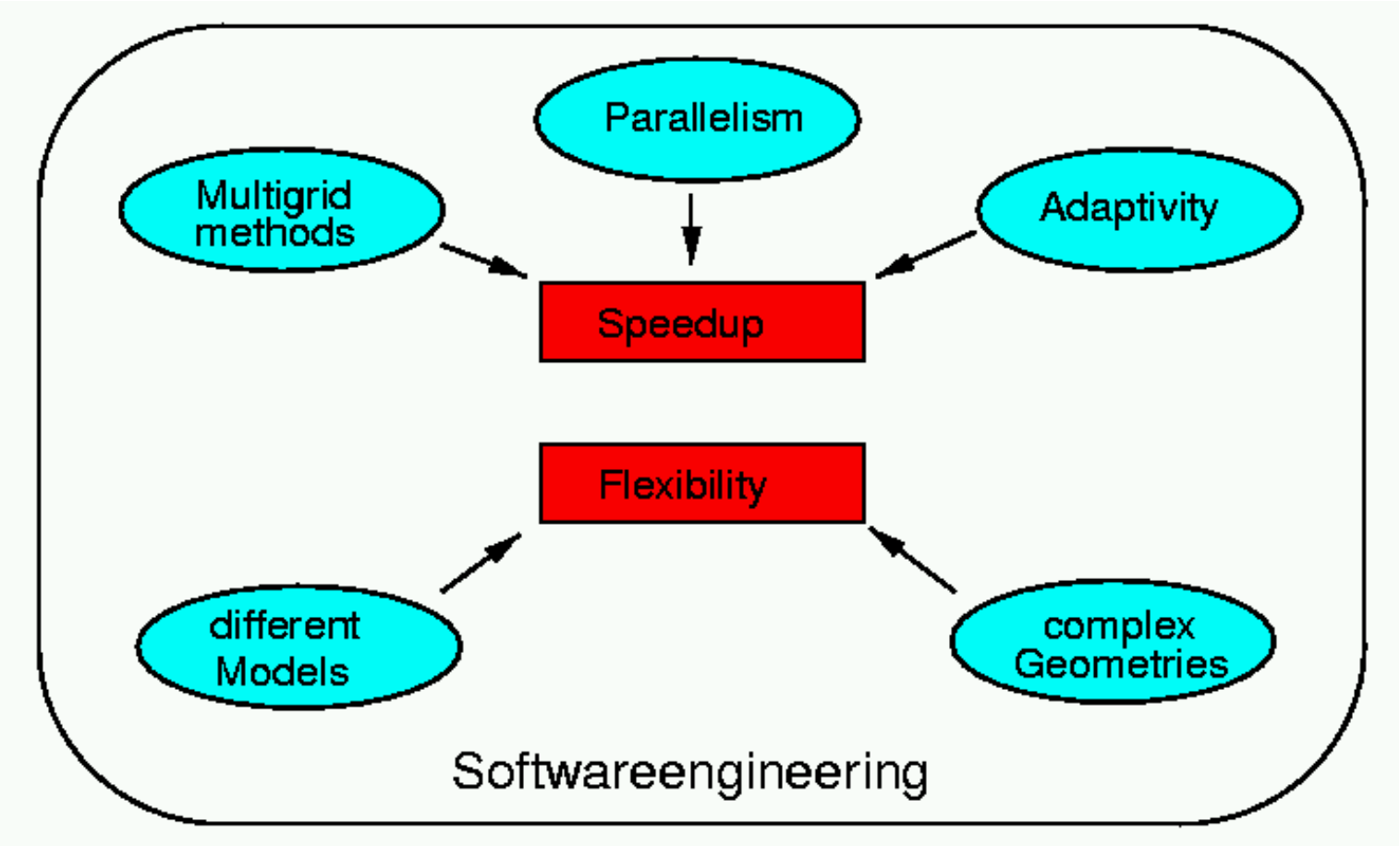
with

$$\mathbf{v} = -K/\mu(\nabla p - \rho \mathbf{g}), \qquad \text{(Darcy's law)}$$
$$D = (\alpha_L - \alpha_T)\mathbf{v}/|\mathbf{v}| + \alpha_T |\mathbf{v}| \qquad \text{(Scheidegger)}$$

Proper initial and boundary conditions have to be defined.

Method and Software Development

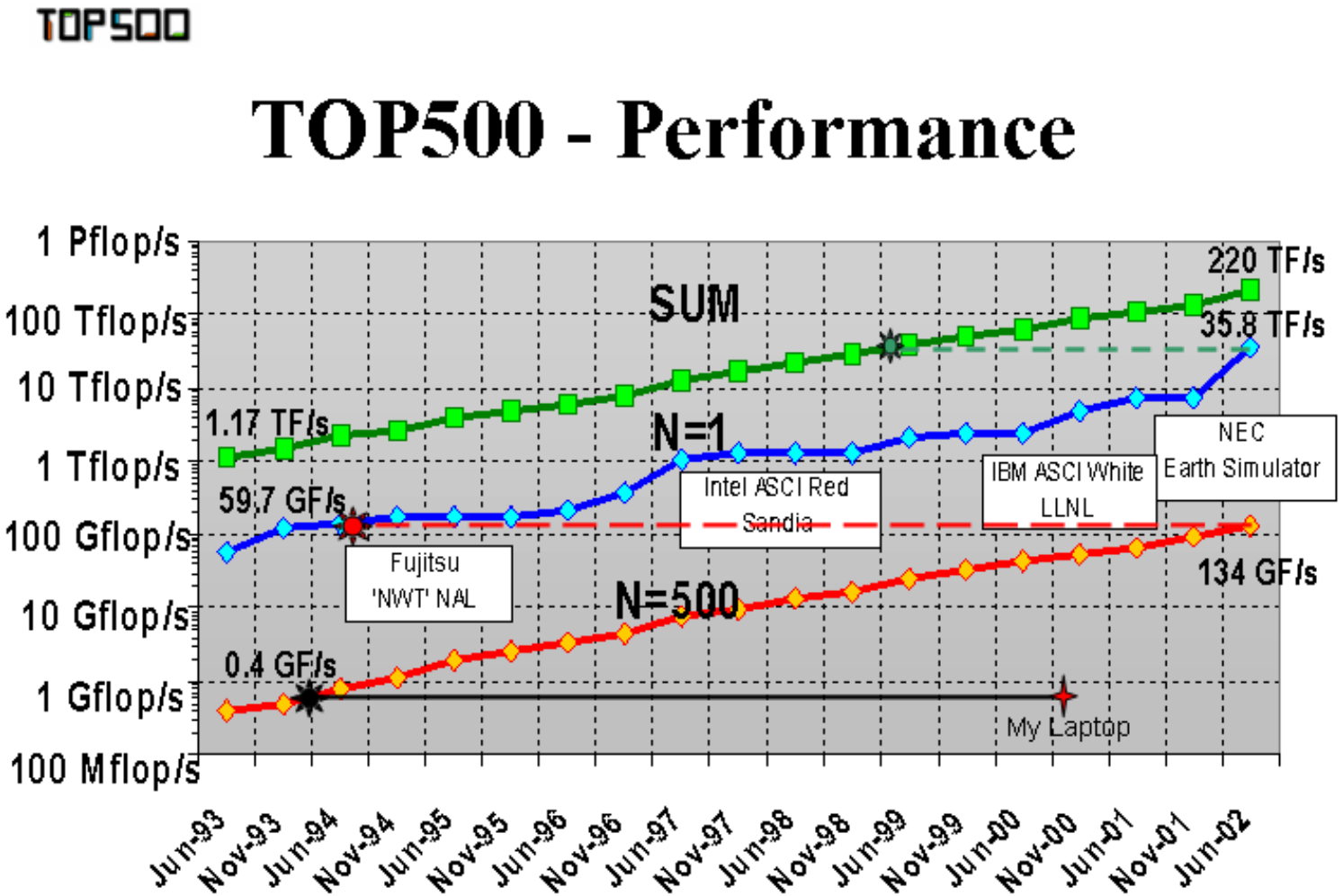
Moreover realistic problems are difficult to handle



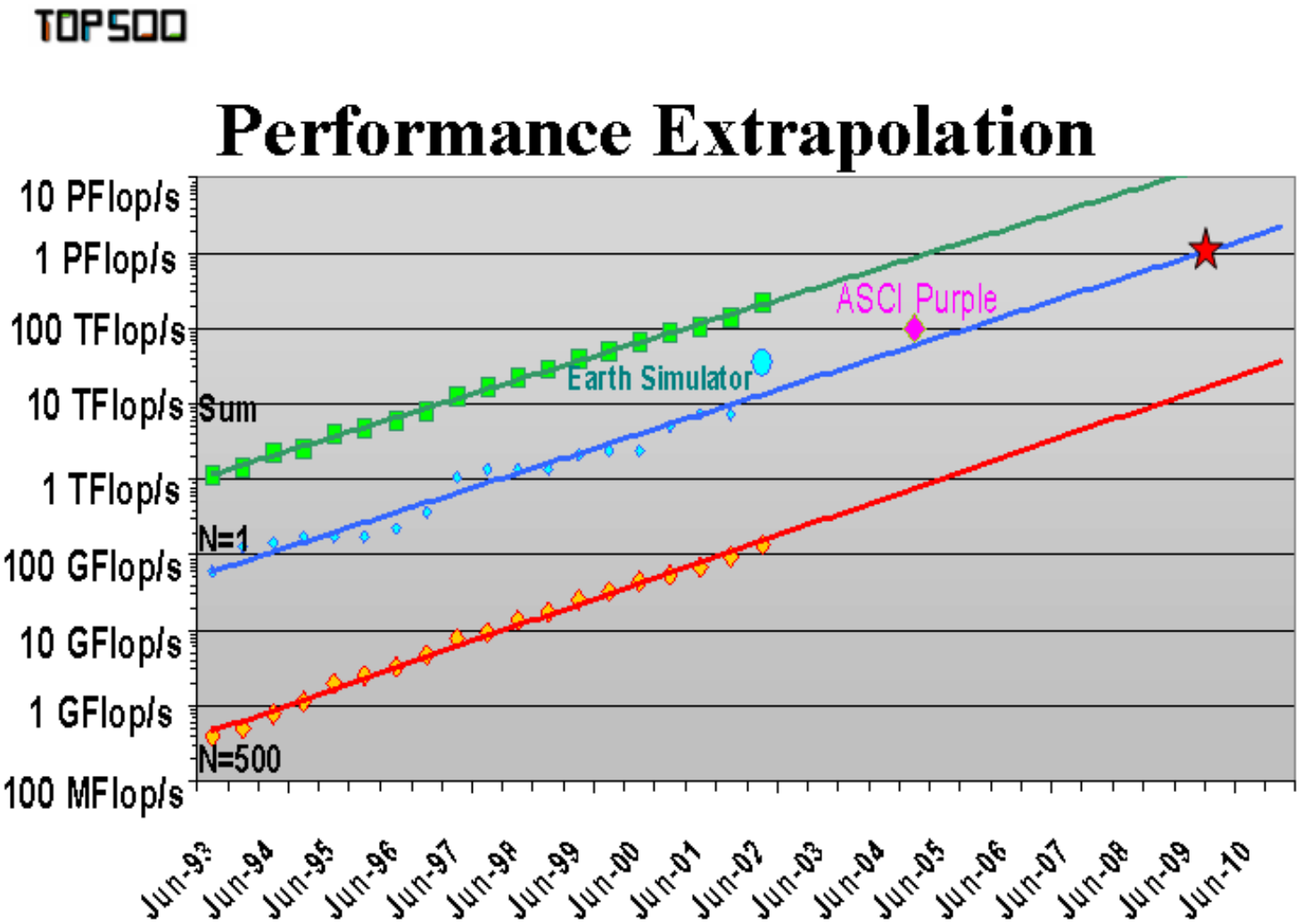
The realization of all these aspects together is a difficult task!

Software engineering is not subject of this course.

TOP500 - List of SuperComputers



TOP500 - Trend



Petaflop machine in 2010! Exaflop until 2018?

Parallelisation: An Introductory Example

Scalarproduct of two vectors

- Introduction of an adequate notation
- Interaction via shared variables
- Interaction via passing of messages
- Evaluation of parallel algorithms

A Simple Problem: Scalar Product Generation

Scalar product of two vectors of length N:

$$s = x \cdot y = \sum_{i=0}^{N-1} x_i y_i .$$

Parallelisation idea:

1. Summands x_i and y_i are independent
2. $N \geq P$, form $I_p \subseteq \{0, \ldots, N-1\}$, $I_p \cap I_q = \emptyset \; \forall p \neq q$
   Each processor calculates now the partial sum $s_p = \sum_{i \in I_p} x_i y_i$
3. Summation of partial sums in example for $P = 8$:

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

Fundamental Conceptions

Sequential program:	Sequence of instructions that are processed sequentially.
Sequential process:	Active execution of a sequential program.
Parallel computation:	Set of interacting sequential processes.
Parallel program:	Describes parallel calculation. Given by a set of sequential programs.

Notation for Parallel Programs I

- Preferably simple and detached of practical details
- Allows different programming models

Program (Pattern of a parallel program)

```
parallel <program name>
{
    // section with global variables (accessible by all processes)
    process <processname-1> [<copyparameters>]
    {
        // local variables, that can be read and written
        // by process <Prozessname-1> only
        // Applications in C-like syntax. Mathematical
        // formula or text allowed for simplification.
    }
    ...
    process <processname-n> [<copyparameters>]
    {
        ...
    }
}
```

Notation for Parallel Programs II

Variable declaration

double $x, y[P]$;

Initialisation

int $n[P] = \{1[P]\}$;

Local/global variables

The execution of the parallel computation starts with initialisation of global variables, then the the processes are executed

Remarks regarding process term: shared variables

Scalarproduct with Two Processes

Program (Scalarproduct with two processes)

```
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!
    }
}
```

- Variables are global, each process works on a part of the indices
- Collision during write access!

Critical Section I

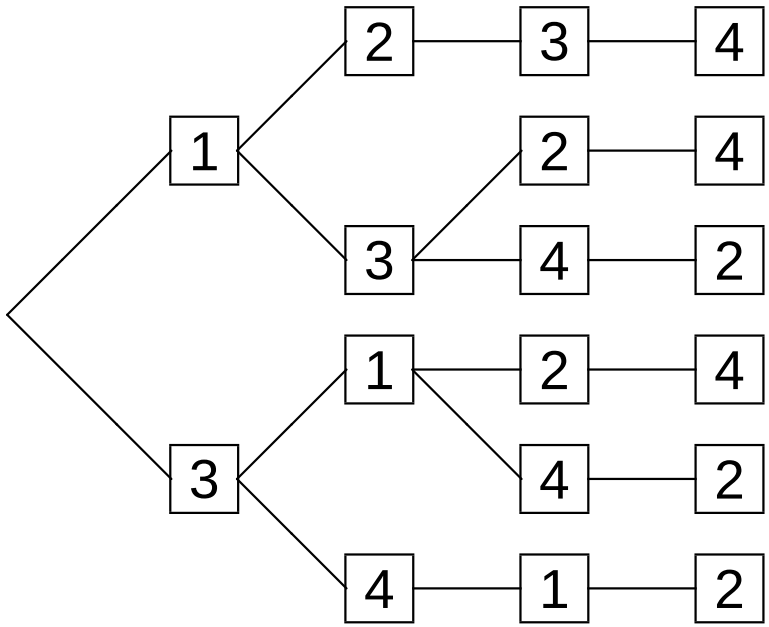
- High level language instruction $s = s + ss$ is transformed in assembly instructions:

	Process Π_1		Process Π_2
1	load s into R1	3	load s into R1
	load ss into R2		load ss into R2
	add R1 and R2 result in R3		add R1 and R2 result in R3
2	store R3 into s	4	store R3 into s

- Execution sequence of instructions of different processes is not determined.

Critical Section II

- Possible execution sequences are:



Result of calculation

$$s = ss_{\Pi_1} + ss_{\Pi_2}$$

$$s = ss_{\Pi_2}$$

$$s = ss_{\Pi_1}$$

$$s = ss_{\Pi_2}$$

$$s = ss_{\Pi_1}$$

$$s = ss_{\Pi_1} + ss_{\Pi_2}$$

- Only sequences 1-2-3-4 and 3-4-1-2 are correct.

Critical Section III

- Instruction block builds a *critical section*, that needs to be processed by *mutual exclusion*.
- We quote this *at first* with squared brackets

$$[\ \langle \text{instruction } 1 \rangle; \ldots; \langle \text{instruction } n \rangle;\ ]$$

- The symbol „[“ selects a process to work on the critical section, all others are waiting.
- Efficient realisation requires hardware instructions, that are introduced later.

Parameterisation of Processes

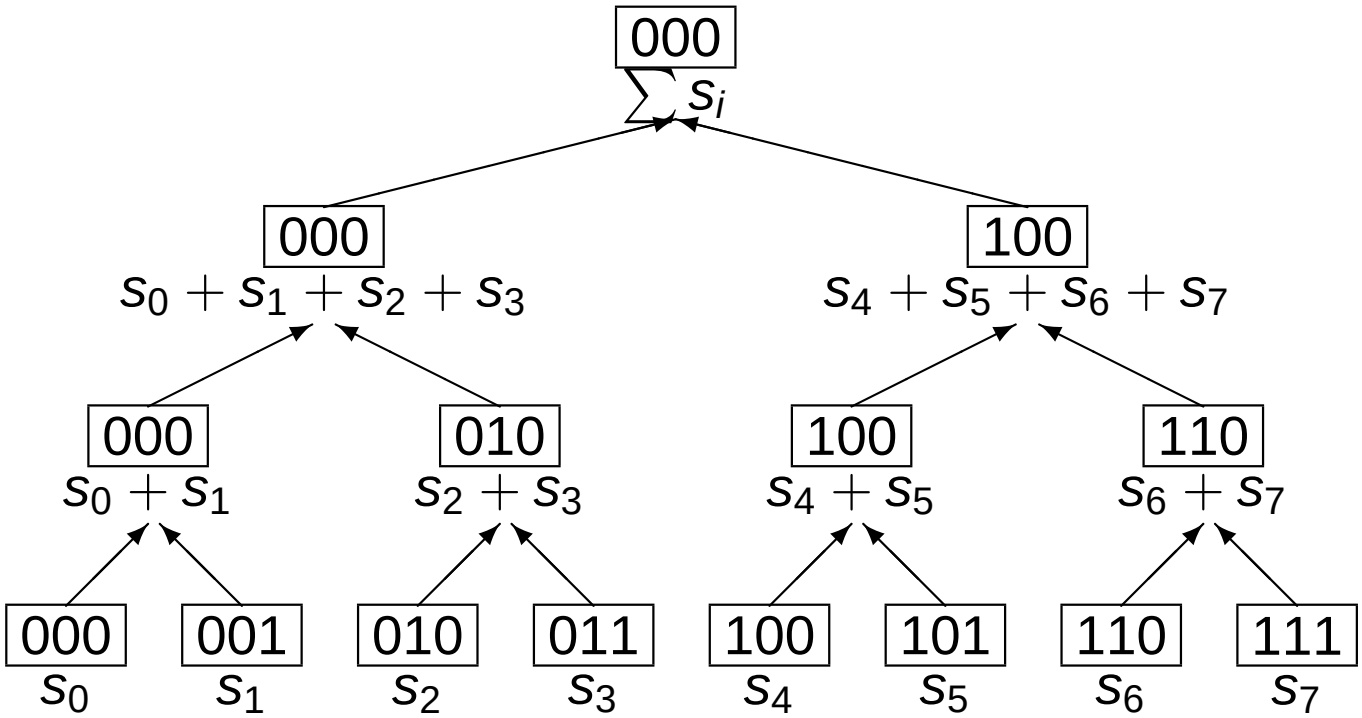
- Processes contain partly identical code (using different data)
- Parameterise the code with a process number, choose the data to be processed using this number
- SPMD = single program multiple data

Program (Scalarproduct with P processors)

```
parallel many-process-scalar-product
{
    const int N;                    // problem size
    const int P;                    // process count
    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];               // Here still all are waiting again
    }
}
```

Communication in Hierarchical Structure

Treelike organisation of the communication sequence with $\mathrm{ld}\, P$ levels



In level $i = 0, 1, \ldots$

- Processes, whose last $i + 1$ bits are 0, fetch
- results of processors whose last i bits are 0 und whose bit i is 1

Program (Parallel Summation)

```
parallel parallel-sum-scalar-product
{
    const int d = 4;
    const int N = 100;                          // problem size
    const int P = 2^d;                          // process count
    double x[N], y[N];                          // vectors
    double s[P] = {0[P]};                       // result
    int flag[P] = {0[P]};                       // process p is ready

    process Π [int p ∈ {0, ..., P − 1}]
    {
        int i, r, m, k;

        for (i = N ∗ p/P; i < N ∗ (p + 1)/P; i++)
            s[p]+ = x[i] ∗ y[i];

        for (i = 0; i < d; i++)
        {
            r = p & [∼ (Σ_{k=0}^{i} 2^k)];      // delete last i + 1 bits
            m = r | 2^i;                        // set bit i
            if (p == m) flag[m]=1;
            if(p == r)
            {
                while(!flag[m]);                // conditional synchronisation
                s[p] = s[p] + s[m];
            }
        }
    }
}
```

Parallelisation of Summation II

- New global variables: $s[P]$ partial results $flag[P]$ indicates, that a processor has finished its computation
- Waiting is called *conditional synchronisation*
- In this example mutual exclusion could be exchanged by conditional synchronisation. This does not work always!
- Reason is that we have fixed the order in advance

Localisation

Goal: Avoid global variables

We advance in two steps: (I) Localise vectors x, y, (II) localise result s

Program (Scalarproduct with local data)

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

    process Π [ int p ∈ {0, ..., P − 1}]
    {
        double x[N/P], y[N/P]; // Assumption N is divisible by P
                               // Local section of 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; ]
    }
}
```

Each stores only N/P indices (one more if not exactly divisible), *these start always with local number 0*

Each local index x equates to a global index in the sequential program:

$$i_{\text{global}}(p) = i_{\text{lokal}} + p * N/P$$

Message Passing I

To completely avoid global variables we need a new concept: *messages*

Syntax:

send(<Process>,<Variable>)
receive(<Process>,<Variable>)

Semantics:
send operation sends content of variable to the specified process,
receive operation waits for message of the specified process and copies it to the variable

send operation waits until the message is received successfully, **receive** operation blocks the process until the message is received

Blocking, or synchronous communication (later other)

Message Passing II

Program (Scalarproduct with message passing)

parallel *message-passing-scalar-product*
{
const int d, $P = 2^d$, N; *// constants!*

process Π [**int** $p \in \{0, \ldots, P-1\}$]
{
double $x[N/P]$, $y[N/P]$; *// local section of vectors*
int i, r, m;
double $s, ss = 0$;

for $(i = 0; i < (p+1) * N/P - p * N/P; i{+}{+})$ $s = s + x[i] * y[i]$;
for $(i = 0; i < d; i{+}{+})$ *// d steps*
{
$r = p \,\&\, \left[\sim \left(\sum_{k=0}^{i} 2^k\right)\right]$;
$m = r \mid 2^i$;
if $(p == m)$
send(Π_r, s);
if $(p == r)$
{
receive(Π_m, ss);
$s = s + ss$;
}
}
}
}

Evaluation of Parallel Algorithms I

Here: asymptotic behaviour in dependance of problem size and processor count

Sequential runtime:

$$T_s(N) = 2Nt_a,$$

t_a: Time for arithmetic operations

Parallel runtime of message-passing variant:

$$T_p(N,P) = \underbrace{2\frac{N}{P}t_a}_{\text{local scalarproduct}} + \underbrace{\operatorname{ld} P(t_m + t_a)}_{\text{parallel sum}},$$

t_m: time to send a number

speedup:

$$S(N,P) = \frac{T_s(N)}{T_p(N,P)} = \frac{2Nt_a}{2\frac{N}{P}t_a + \operatorname{ld} P(t_m + t_a)}$$
$$= \frac{P}{1 + \frac{P}{N}\operatorname{ld} P\frac{t_m + t_a}{2t_a}}$$

It holds $S(N,P) \leq P$!

Evaluation of Parallel Algorithms II

Efficiency:

$$E(N,P) = \frac{S(N,P)}{P} = \frac{1}{1 + \frac{P}{N} \operatorname{ld} P \frac{t_m + t_a}{2t_a}}$$

It applies $E \leq 1$

Consideration of asymptotic limit cases:

- Fixed N, growing P: $\lim_{P \to \infty} E(N,P) = 0$
- Fixed P, growing N: $\lim_{N \to \infty} E(N,P) = 1$
  For which relation of $\frac{P}{N}$ „acceptable“ efficiency values are achieved is regulated by the factor $\frac{t_m + t_a}{t_a}$. This is the relation of communication to computation time.
- Scalability for simultaneously growing of N and P in the form $N = kP$:

$$E(kP,P) = \frac{1}{1 + \operatorname{ld} P \frac{t_m + t_a}{2t_a k}}$$

  Drops off slowly with $P \to$ good scalable.

Exemplary for many algorithms!

Topics

Hardware

- (Parallel) computer architecture: SMP, vector computer, parallel computer
- realisations: some architectures in detail (Paragon, ASCI Red Storm, IBM Blue Gene)

Programming models

- Programming models: OpenMP, MPI, PThreads
- Fundamental numerical algorithms: matrix multiplication

Algorithms

- Performance evaluation: Efficiency, speedup, scalability
- Parallel sorting
- Dense und sparse filled equation systems

Applications

- Parallel (Numerical) applications (neuroscience, biology, soilphysics, astrophysics)

Themes are without commitment.

Organisational Stuff

- Exercises Thursday 14.00-16.00
  Theme: Introduction to (Advanced) C++
  CIP Pool, OMZ, INF 350, U.012
- Certificate: regular participation, 50% points
- Exam: mostly oral after consultation
- Course count: 8 credit points