

Simulation on High Performance Computers

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Organisational Information

- Lecturer: Stefan Lang, Parallel Computing, IWR
- Lecture: 4h lectures + 2h exercises
- Dates
 - ▶ Lectures: Tu 11.00-13.00 (V in SR11), Th 11.00-13.00 (V in SR11)
 - ▶ Exercises: Th 14.00-16.00 (E in SR11)
- Prerequisites:
Basic lectures in Computer Science and Numerics
- Helpful:
Knowledge of C/C++

What means Scientific Computing?

In particular Numerical Simulation (NS):

- Goal of NS is to simulate natural or technical processes with computing machines
- Interdisciplinary approach: natural scientists, engineers, mathematicians and computer scientists work together
- practical relevant problems are handled systematically with formal methods
- NS enables insight into areas that are difficult to access in lab experiments and field studies, in example neuroscience, cell biology, water economics, astrophysics

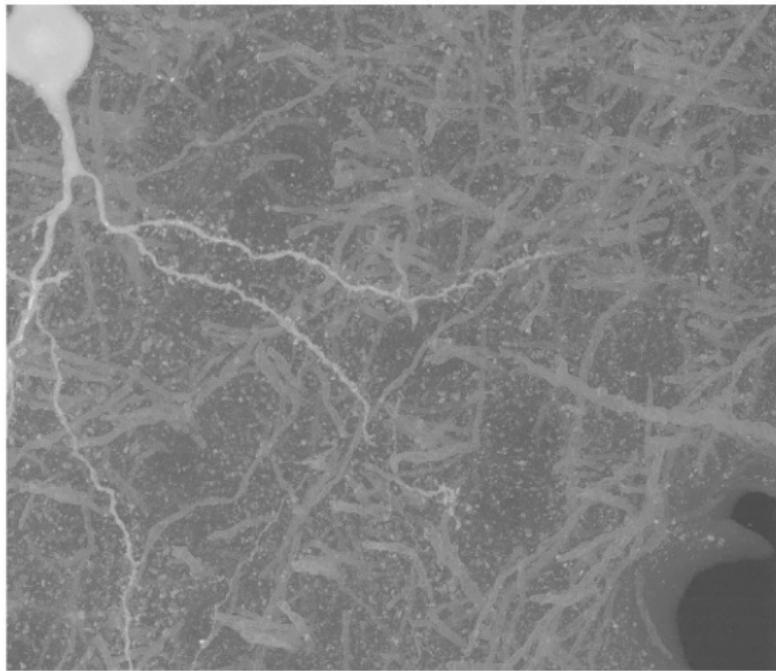
Why Scientific High-Performance Computing?

Trends in **Numerical simulation**:

- treatment of **global models** instead of local partial models
(Models of entire brain areas, 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 an **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, that reflects an observed or measured system behaviour



- Simulation of neuronal networks, statistical analysis of realisations

A Supercomputer

ASCI Red Storm

Successor of the first TeraFlop computer ASCI Red 1997

Hardware:

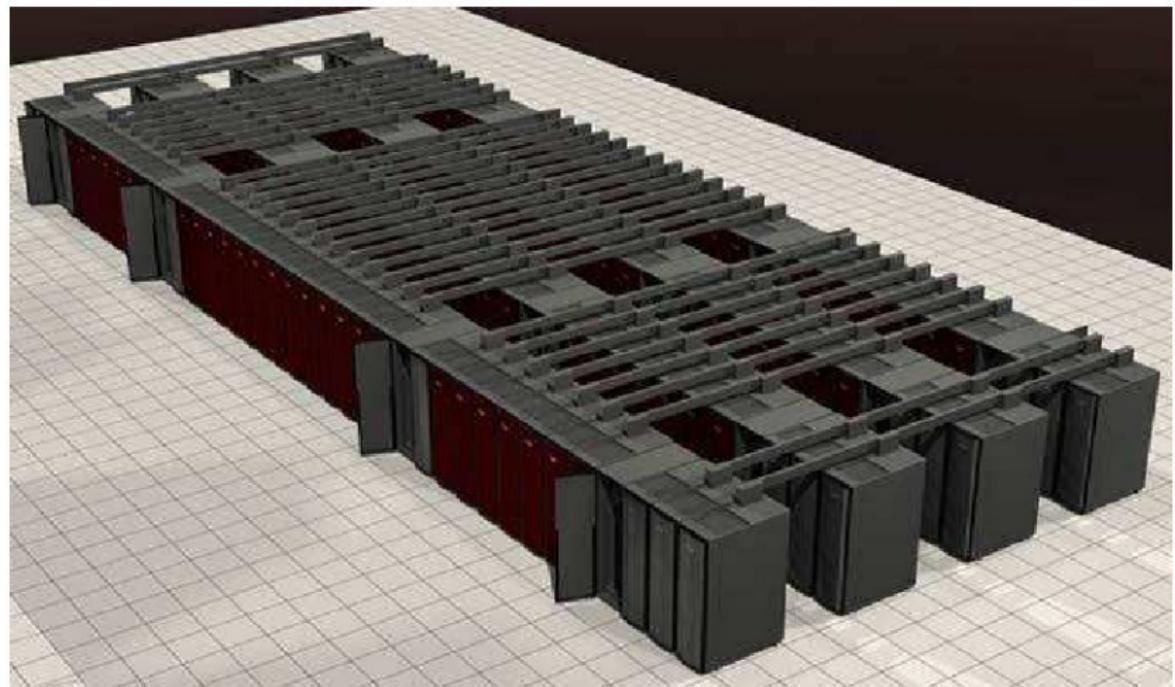
- 11.646×2 GHz AMD Opteron CPUs
- CPU boards vertically mounted in 108 cabinets
- 4 GFLOPS per CPU (40 TFLOPS total)
- 1 GB per CPU (10 TB total)
- shared memory inside the node
- 3D mesh full interconnect

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 Dissection 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.

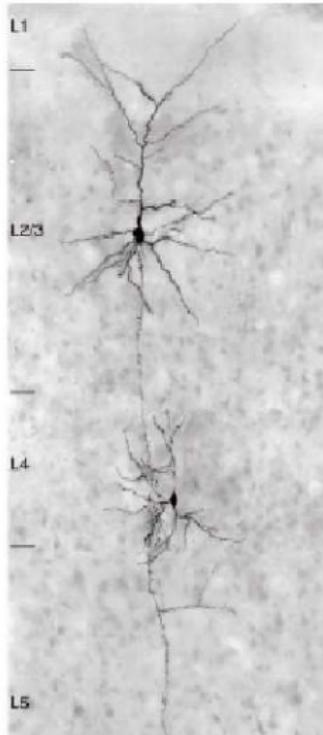
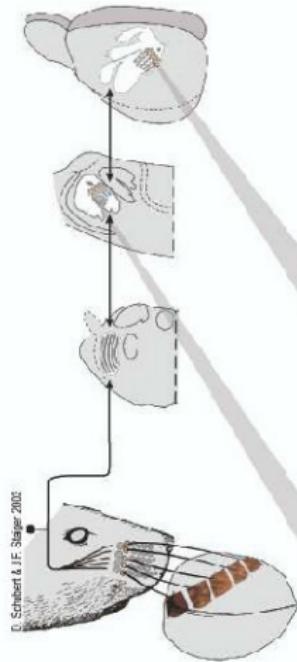
This is seldom the case (unfortunately)!

consequence:

We need

- enabling software to handle powerful computers
 - scalable algorithms + scalable implementations + scalable architectures
- ... buying huge 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

$$g_a \partial_{\mathbf{x}} v = g_N \quad \text{on } \partial \Omega_N,$$

$$v = g_D \quad \text{on } \partial \Omega_D,$$

$$(v, \mathbf{c})(\mathbf{x}, 0) = (v^0, \mathbf{c}^0) \text{ for } t = 0.$$

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

$$i_{Na}(v) = \overline{g_{Na}} \cdot m^3 h \cdot (v - E_{Na}),$$

$$i_K(v) = \overline{g_K} \cdot n^4 \cdot (v - E_K),$$

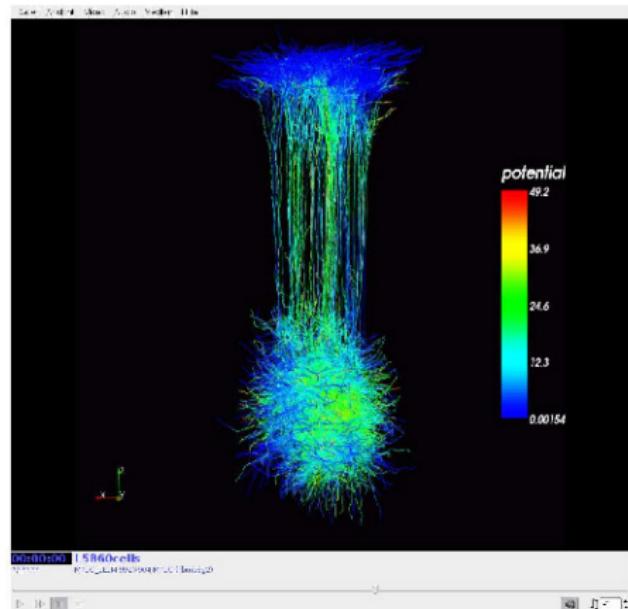
$$i_L(v) = \overline{g_L} \cdot (v - E_L)$$

Moreover currents of synapses are modeled by

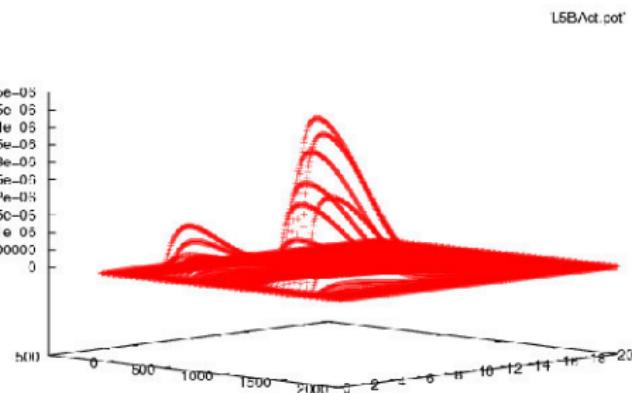
$$i_s(v, t) = g_s(t) \cdot (v - E_s)$$

with time-dependent synaptic strength g_s .

Simulation Study: Passive Deflection of a Whisker

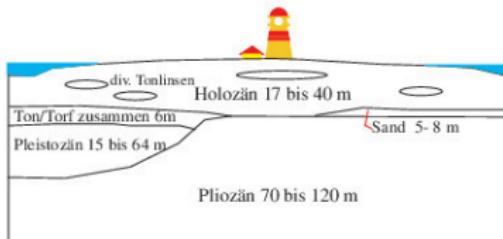


60 L5B neurons activated by VPM



Spatial and temporal activity distribution

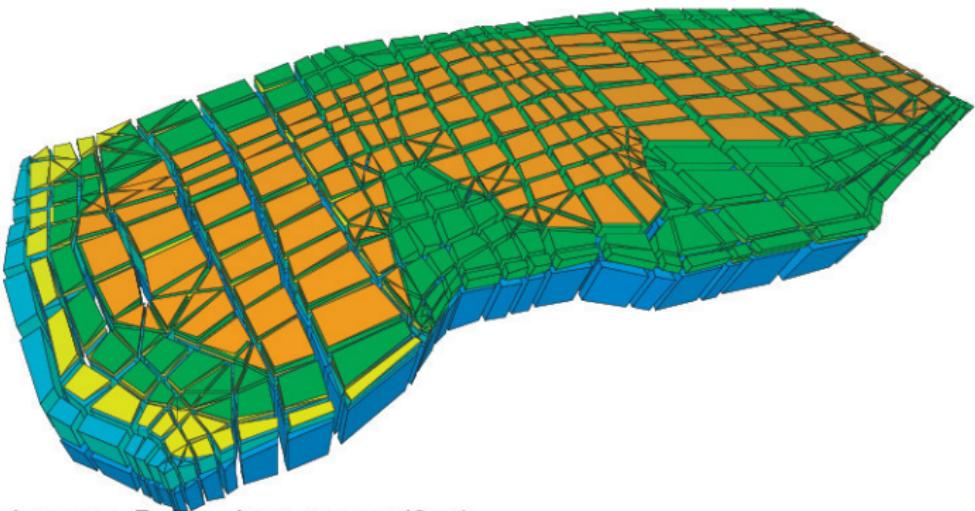
Problem study Norderney



- project "coastal preservation"
- island typical fresh water lens (-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

$$\begin{aligned}\partial_t(n\rho) + \nabla \cdot (\rho\mathbf{v}) &= Q\rho, && \text{(f law)} \\ \partial_t(n\rho\omega) + \nabla \cdot (\rho\mathbf{v}\omega - \rho D\nabla\omega) &= Q\rho\omega && \text{(transp.)}\end{aligned}$$

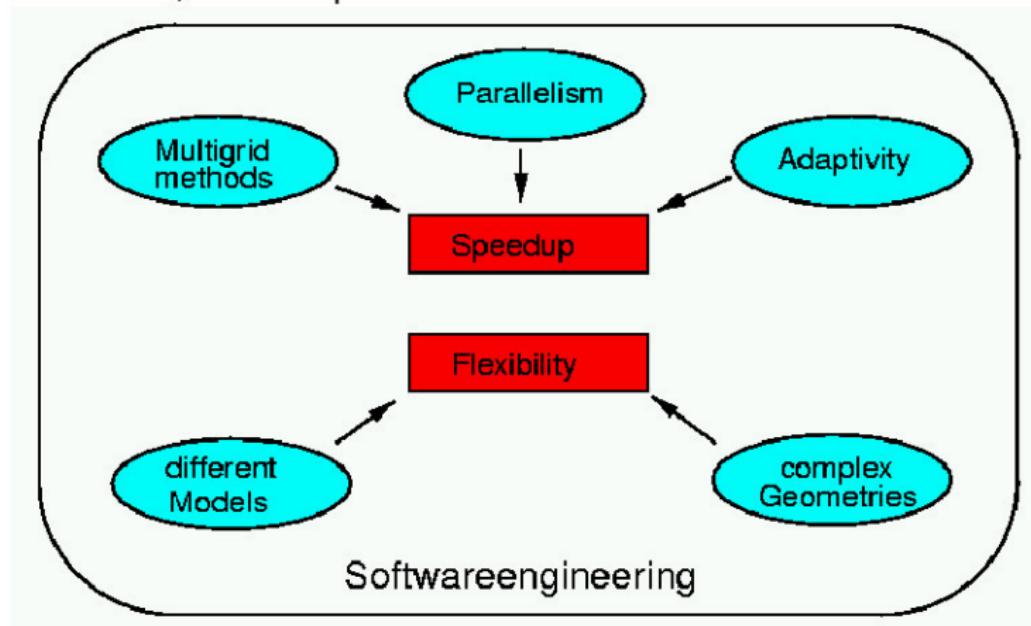
with

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

Proper initial and boundary conditions have to be defined.

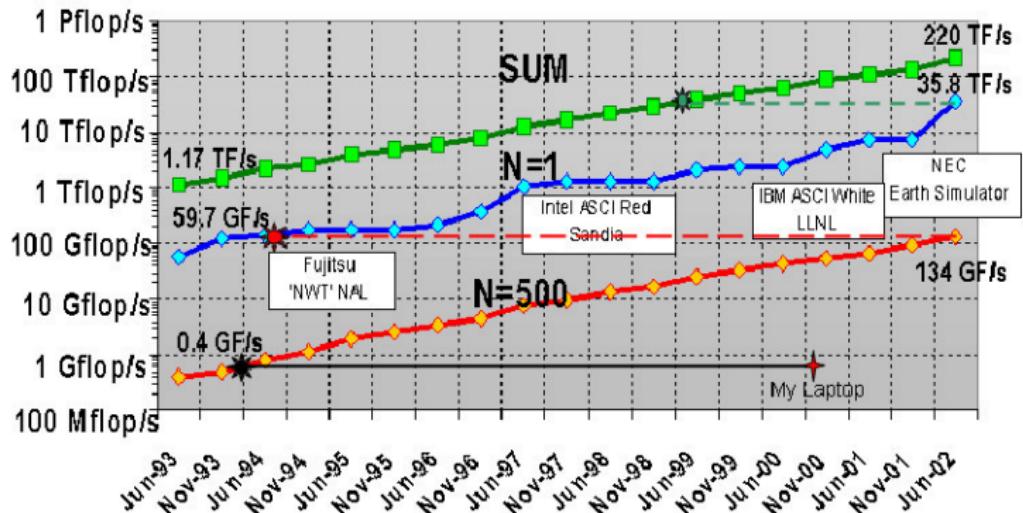
Method and Software Development

Additional, realistic problems are difficult to handle



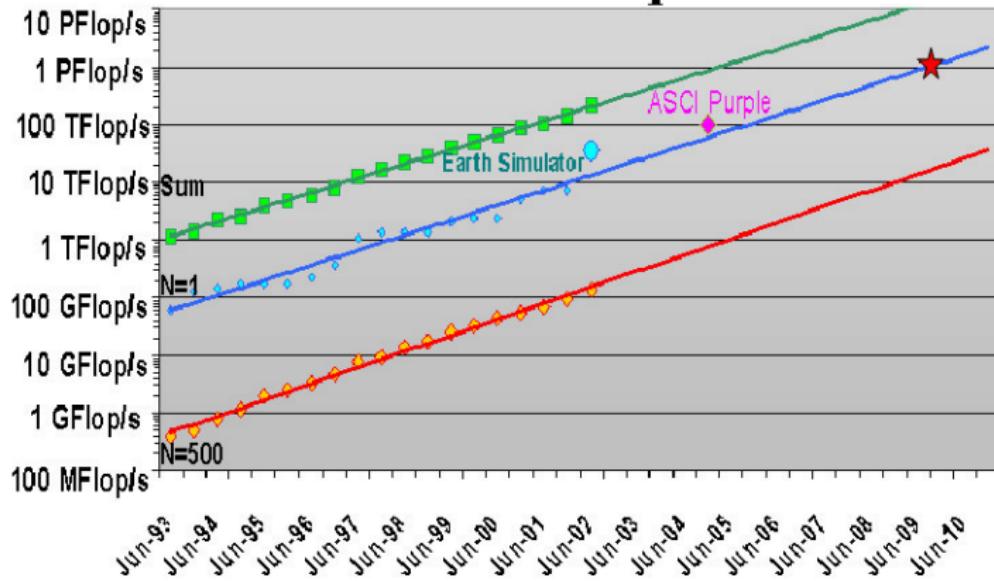
To realize all these aspects together is a difficult task!
Software engineering is not subject of this course.

TOP500 - Performance



TOP500

Performance Extrapolation



Petaflop machine in 2010! Exaflop until 2018?

Parallelisation: An Example as Introduction

Scalarproduct of two vectors

- Introduction of an adequate notation
- Interaction via shared variables
- Interaction via 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:

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

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

Fundamental Conception

Sequential program: Sequence of instructions that are processed sequential.

Sequential process: Active execution of a sequential program.

Parallel calculation: Set of interacting sequential processes.

Parallel program: Describes parallel calculation. Given by a set of sequential programs.

Notation for Parallel Programs

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

Program (Patterns 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

Remarks regarding process term

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 part of indices
- Collision during write access!

Critical Section I

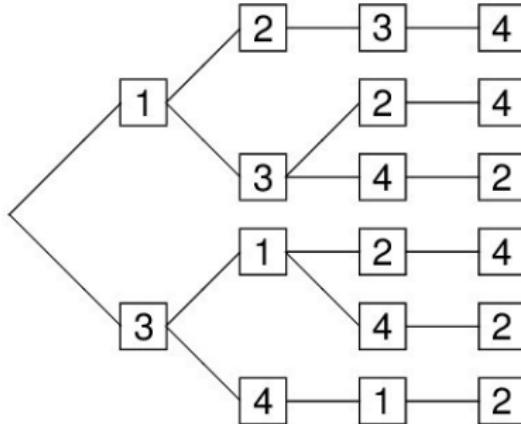
- High level language instruction $s = s + ss$ is transformed in assembly instructions:

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

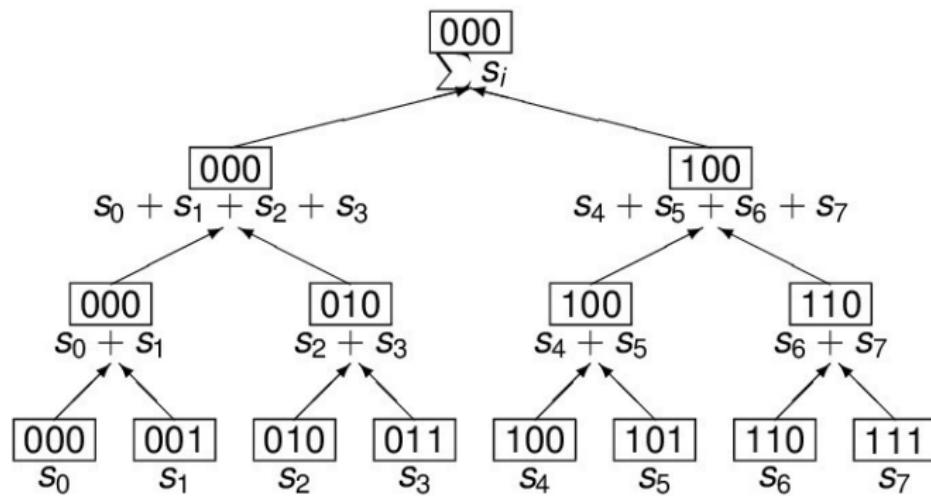
- Processes contain partly identical code (using different data)
- Parameterise the code with a process number, choose the data to be processed using this number
- SPMD = single program multiple data

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 ∏ [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 $\log P$ levels



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

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

```

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  $\Pi$  [int p  $\in \{0, \dots, P - 1\}$ ]
    {
        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 &  $\left[ \sim \left( \sum_{k=0}^i 2^k \right) \right]$ ;           // 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 processor has finished
- 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  $\Pi$  [ int  $p \in \{0, \dots, P - 1\}$  ]
    {
        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:

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

Message Passing I

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

Syntax:

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

Semantics:

send sends content of variable to the specified process,

receive waits for message of the specified process and copies it to the variable

send waits until the message is received successfull, **receive** 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 = 2d, N;                                // constants!

    process  $\Pi$  [int  $p \in \{0, \dots, P - 1\}$ ]
    {
        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 | 2i;
            if ( $p == m$ )
                send( $\Pi_r, s$ );
            if ( $p == r$ )
            {
                receive( $\Pi_m, ss$ );
                s = s + ss;
            }
        }
    }
}
```

Evaluation of Parallel Algorithms I

Here: asymptotic behaviour in dependence 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:

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

t_m : time to send a number

speedup:

$$\begin{aligned} S(N, P) &= \frac{T_s(N)}{T_p(N, P)} = \frac{2Nt_a}{2\frac{N}{P}t_a + \text{Id } P(t_m + t_a)} \\ &= \frac{P}{1 + \frac{P}{N} \text{Id } P \frac{t_m + t_a}{2t_a}} \end{aligned}$$

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

Evaluation of Parallel Algorithms II

Efficiency:

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

It applies $E \leq 1$

asymptotic statements:

- fixed N , growing P : $\lim_{P \rightarrow \infty} E(N, P) = 0$
- fixed P , growing N : $\lim_{N \rightarrow \infty} E(N, P) = 1$

For which relation $\frac{P}{N}$ „acceptable“ efficiency values are achieved,
regulates the factor $\frac{t_m + t_a}{t_a}$, the relation of communication to computation
time.

- Scalability for simultaneously growing of N and P in the form $N = kP$:

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

Drops off slowly with $P \rightarrow$ 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.

Parallel Computer Architecture I

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Parallel Computer Architecture I

- Why parallel computing?
- Von-Neumann architecture
- Pipelining
- Cache
- RISC und CISC
- Scalable computer architectures
- UMA, NUMA
- Protocols for cache coherency
- Examples

Definition of Parallel Machine

What is a parallel machine?

A collection of processing elements that communicate and cooperate to solve large problems fast
(Almasi und Gottlieb 1989)

What is a parallel architecture?

It extends the usual concepts of a computer architecture with a communication architecture

Why Parallel Computing?

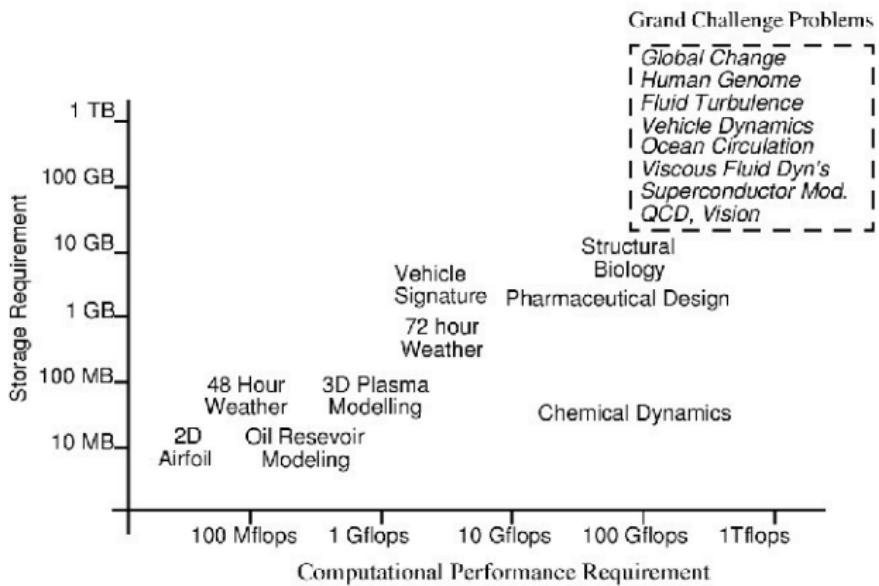
3 flavours of parallel computing

- Solve a problem of fixed size fast
Goal: Minimize time-to-solution and speedup r&d cycle
- Compute very large problems
Goal: exact result, complex systems
- Simulate very large problems fast (respec. in adequate time)
Goal: Grand Challenges

Single processor performance is not sufficient

→ Parallel Architectures

What are Problems?

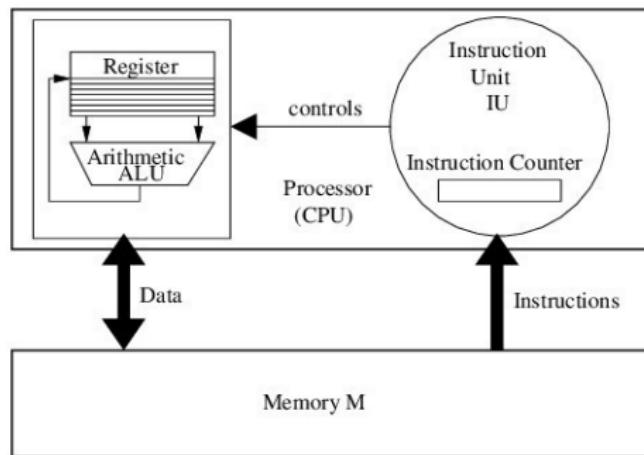


from Culler, Singh, Gupta: Parallel Computer Architecture

- Classification of problems according to memory and computing demands
- Categorisation in 3 types: memory limited, compute-time limited and balanced problems

Von Neumann Architecture

Schematic structure with instruction unit, arithmetic unit and memory



Instruction cycle:

- fetch instruction
- decode instruction
- execute instruction
- store results
- Memory contains program code and data
- Data transfer between processor and memory uses system bus
- Several devices (processors, I/O-Units, Memory) on bus

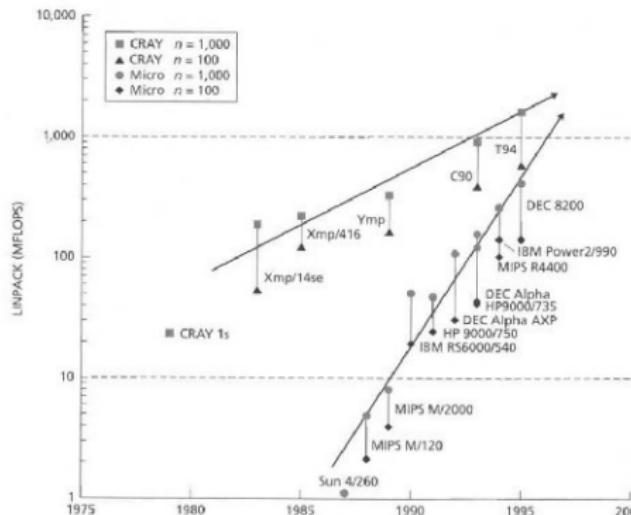
Generations of Electronic Computers

Distinction of 5 + 2 computer generations

Generation	Technology and Architecture	Software and Applications	Representative Systems
First (1945-54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator	Machine/assembly languages, single user, no subroutine programmed I/O using CPU	ENIAC, Princeton IAS, IBM 701
Second (1955-64)	Discrete transistors and core memories, floating-point arithmetic	HLL used with compilers, subroutine libraries, batch processing monitor	IBM 7090, CDC 1604, Univac LARC
Third (1965-74)	Integrated circuits, micro-programming, pipelining cache, lookahead processors	Multiprogramming and time-sharing OS, multiuser applications	IBM 360/370, CDC 6600, TI-ASC, PDP-8
Fourth (1975-90)	LSI/VLSI, semiconductor memory, multiprocessors, vector- and multicompilers	Multiprocessor OS, languages, compilers, environments for parallel processing	VAX 9000, Cray X-MP, IBM 3090
Fifth (1991-1997)	ULSI/VHSIC processors, mems and switches, high-density packaging, scalable archs	Massively parallel processing grand challenge applications heterogeneous processing	Fujitsu VPP-500, Cray/MPP, Intel Paragon
Sixth (1997-2003)	commodity-component cluster high speed interconnects	Standardized Parallel Environments and Tools, Metacomputing	Intel ASCI-Red, IBM SP2, SGI Origin
Seventh (2004-present)	Multicore, Powersaving Extending memory hierarchy	Software for Failure Tolerance, Scalable I/O, Grid Computing,	IBM Blue Gene, Cray XT3

nach Hwang (with additions)

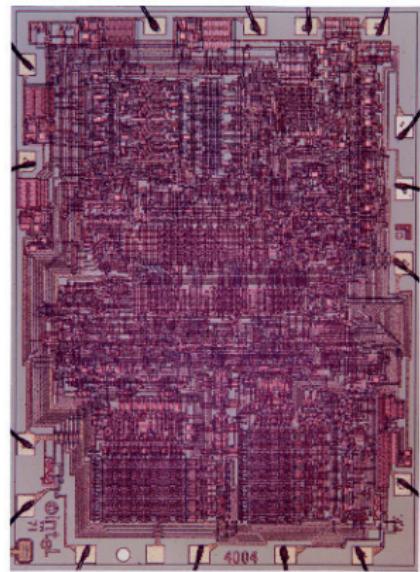
Single-chip Processor Performance



Culler, Singh, Gupta: Parallel Computer Architecture

- Performance development of vector- and superscalar processors
- Earlier: many manufacturers, now: some market leaders
- Speed advantage of vector processors shrinks

Single-chip Processors: Two Examples



1971: Intel 4004, 2700 Trans.,
4 bit, 100 KHz

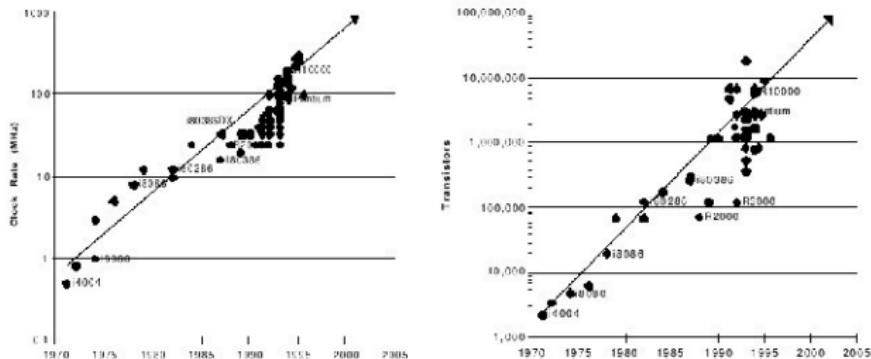


2007: AMD Quadcore, 465 mill. trans.,
64 bit, 2 GHz



Intel founder Andy Grove, Robert Noyce, Gordon Moore in 1978

Integration Density and Clock Frequency



Culler, Singh, Gupta: Parallel Computer Architecture

- Increase according to Moore's law: Doubling within 18 months
- Moore's law is NOT related to performance but to integration density
- Divergence of speed and capacity in storage technologies

Architecture of Single-core Processors

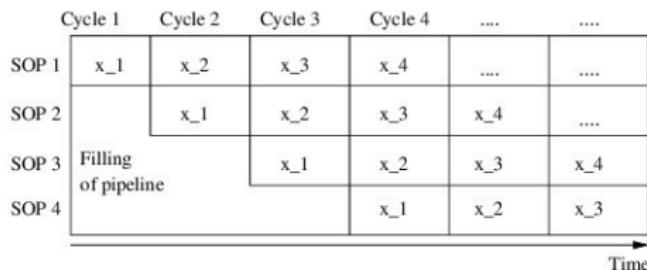
Techniques to increase single-core processor performance

- deep pipelining
- speculative branch prediction
- out-of-order execution
- clock frequency scaling
- superscalar design (instruction level parallelism ILP)
- speculative execution
- thread-level parallelism
- multi-core design

Pipelining I: Principle

Synchronous, overlapping processing of operations

Pipeline with 4 stages:



Requirements:

- An operation $OP(x)$ has to be applied onto many operands x_1, x_2, \dots in sequence.
- The operation can be divided into $m > 1$ sub-operations (or also stages), that can be executed in (preferably) equal time.
- An operand x_i may be with restrictions only a result of former operations.

Gain with pipelining: The time demand for processing of N operands is

$$T_P(N) = (m + N - 1) \frac{T_{OP}}{m}$$

Pipelining II: Speedup

The Speedup is therefore

$$S(N) = \frac{T_S(N)}{T_P(N)} = \frac{N * T_{OP}}{(m + N - 1) \frac{T_{OP}}{m}} = m \frac{N}{m + N - 1}$$

For $N \rightarrow \infty$ the speedup converges towards m .

Utilization inside processors:

- Instruction pipelining: fetch, decode, execute, write back
- Arithmetic pipelining: adapt exponents, add mantissa, norm mantissa

Further applications:

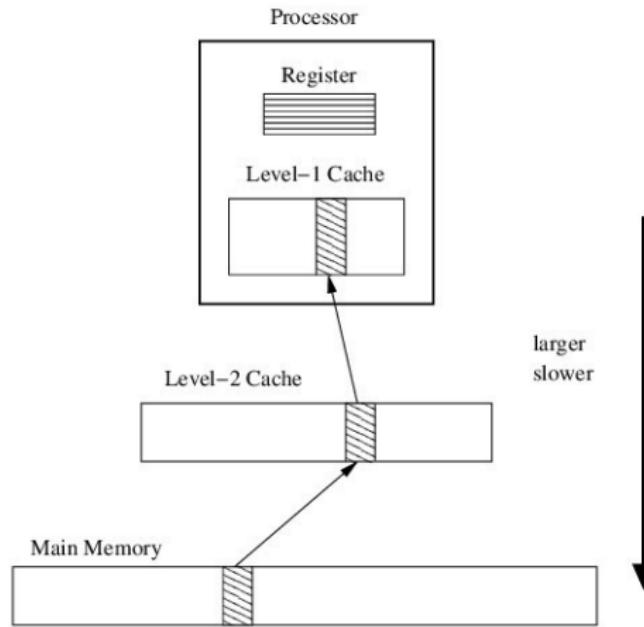
- Memory interleaving
- Cut-through routing
- Wavefront algorithms: LU-decomposition, Gauß–Seidel
- ...

Cache I: Memory Hierarchy

Speed gap:

- Processors are fast: 2-3 GHz clock, ≥ 1 instruction/cycle due to pipelining
- Memory is slow: MHz clock, 7 cycles to read 4 words

Way out: Hierarchy of always slower but larger memories



Cache II: Cache Organisation

Memory contains respectively least recently used data of the next higher hierarchy level

Transfer is managed in blocks (Cache Lines), typical size: 16...128 bytes

Cache organisation:

- Direct mapping: main-memory block i can only be positioned in place $j = i \bmod M$ inside the cache (M : size of cache).
Advantage: easy identification, Disadvantage: aliasing.
- Assoziative cache: main-memory block i can be positioned at each location inside the cache.
Advantage: no aliasing, Disadvantage: costly identification (M comparisons).
- Combination: k -way assoziative cache.

Replacement: LRU (least recently used), random

Storage: write through, write back

Cache III: Locality Principle

Up to now we have assumed, that all memory words can be accessed equally fast.

But with cache least recently fetched data can be accessed faster. This has implications on the implementation of algorithms.

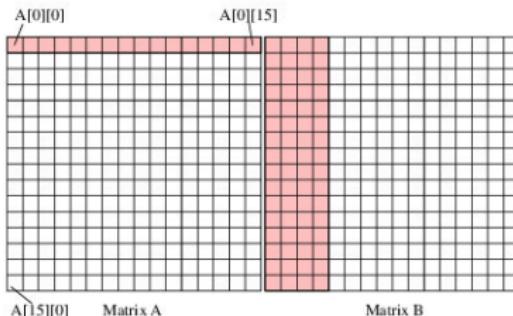
Example: Multiplication of two $n \times n$ -matrices $C = AB$

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Assumption: Cache-line is 32 bytes = 4 floating point numbers.

Cache III: Locality Principle

After calculation of $C[0][0]$ there are the following words stored inside the cache:



A, B, C completely in cache: $2n^3$ arithmetic operations but only $3n^2$ memory accesses

If fewer than $5n$ numbers fit into the cache: slow

Tiling: Process matrix in $m \times m$ blocks with size $3m^2 \leq M$

```
for (i = 0; i < n; i+=m)
    for (j = 0; j < n; j+=m)
        for (k = 0; k < n; k+=m)
            for (s = 0; s < m; s++)
                for (t = 0; t < m; t++)
                    for (u = 0; u < m; u++)
```

RISC und CISC

RISC = „reduced instruction set computer“

CISC= „complex instruction set computer“

Development of processors with increasingly complex instruction sets (i.e. addressing methods): Costly decoding, instructions with variable length

Begin of 1980s : „Back to the roots“. Simple instructions, aggressive usage of pipelining.

The idea was not new: Seymour Cray has always build RISC machines (CDC 6600, Cray 1).

Design principle of RISC machines:

- All instructions are coded in hardware, no micro programming.
- Aggressive usage of instruction pipelining (parallelism on instruction level ILP).
- Preferably execute one instruction/cycle (or more for superscalar machines). This requires a preferably simple and homogeneous instruction set.
- Memory accesses only with special load/store-instructions, no complicated addressing methods.
- Provide many general purpose registers to minimize memory access. The saved chip area in the instruction unit is used for registers or caches.
- Follow the design principle „Make the frequently occurring case fast“.

Today predominantly RISC processors. Intel Pentium is CISC with RISC-core.

Scalable Computer Architecture I

Classification of parallel machines according to FLYNN (1972)

Distinction with regard to data streams and control paths

- *SISD – single instruction single data*: The Von Neumann Computer
- *SIMD – single instruction multiple data*: The machines, also called array processor, possess an instruction set and multiple independent arithmetic units each is connected to its own memory. The arithmetic units are controlled clock synchronous by the instruction unit and execute the same operation on different data.
- *MISD – multiple instruction single data*: This category is empty.
- *MIMD – multiple instruction multiple data*: This correlates to a collection of self-contained computers, each equipped with its own instruction– and arithmetic unit.

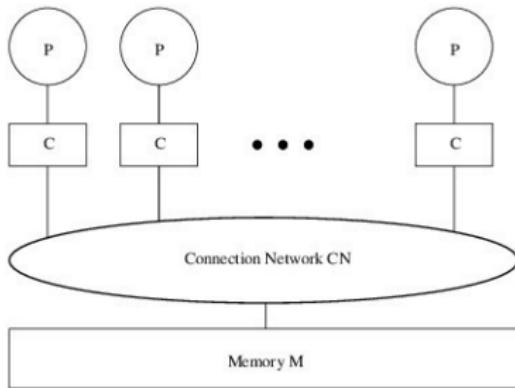
Scalable Computer Architecture III

Classification according to type of data exchange:

- Shared Memory
 - ▶ *UMA – uniform memory access.* Shared memory with uniform access time.
 - ▶ *NUMA – nonuniform memory access.* Shared memory with *non-uniform* access time, with cache-coherency we speak of *ccNUMA*.
- Distributed Memory
 - ▶ *MP – multiprocessor.* Private memory with message passing.

We will consider predominantly MIMD–machines. The SIMD approach exists still in the data parallel programming model (OpenMP, CUDA/OpenCL).

Shared Memory: UMA



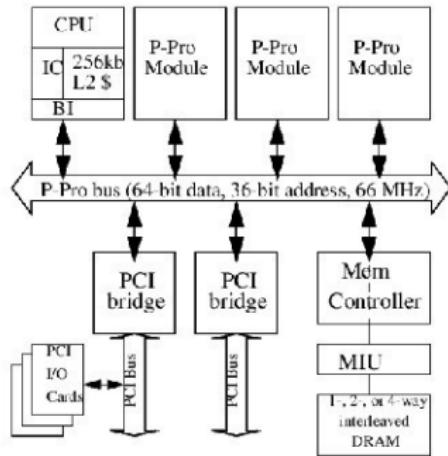
- *Global address space:* Each memory word has its global unique number and can be read and written by all processors.
- Memory access occurs over a *dynamic* connection network that connects processor and memory (therefrom later more).
- Memory organisation: *Low-order interleaving* – consecutive addresses are in consecutive modules. *High-order interleaving* – consecutive addresses are in the same module.

Shared Memory: UMA

Cache is necessary to

- avoid slow down of the processor, and
- to remove load from the connection network.
- Cache coherency problem: A memory block can be stored in several caches. What happens, if a processor writes?
- Write access onto the same block in different caches have to be serialized. Read accesses have to provide up-to-date data.
- UMA enables the usage of up to few 10th of processors.

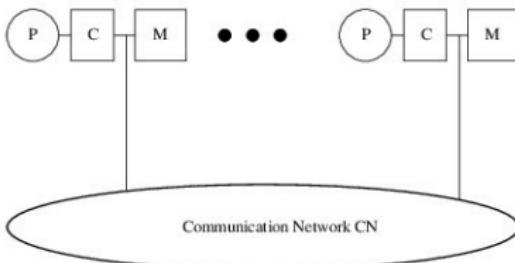
Shared Memory Board: UMA



Quad-processor Pentium Pro Motherboard

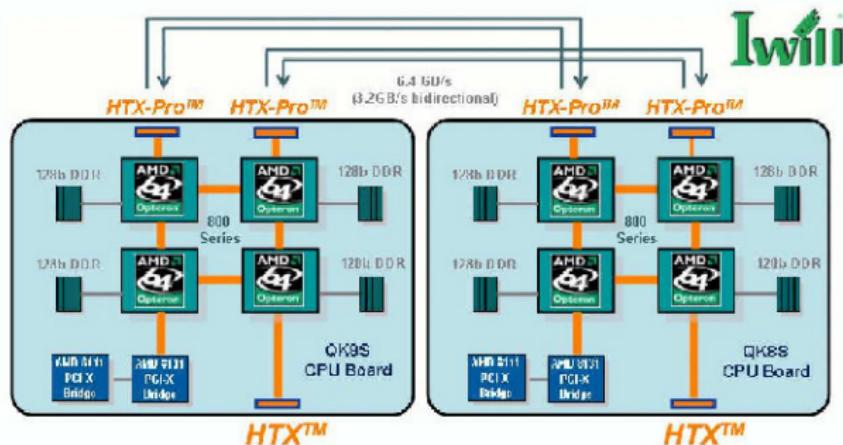
- Symmetric multi processing (SMP)
- Access to each memory word in equal time
- Implementation of cache coherency protocols (MESI)

Shared Memory: NUMA



- Each component consists of processor, memory and cache.
- *Global address space*: Each memory word has a global unique number and can be read and written from all processors.
- Access onto local memory is fast, access onto other memory is (considerably) slower, but transparently possible.
- Cache-coherency problem as in the UMA case
- Extreme memory hierarchy: level-1-cache, level-2-cache, local memory, remote memory
- Scales up to about 1000 processors (SGI Origin)

Shared Memory Board: NUMA

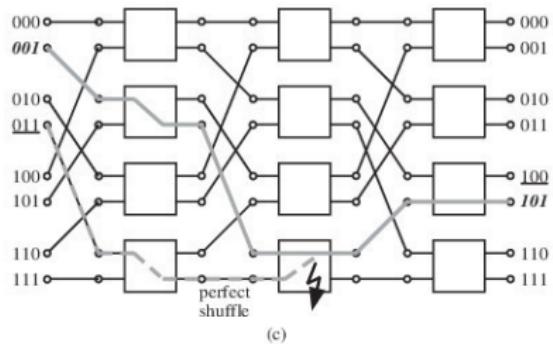
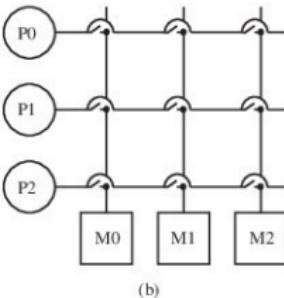
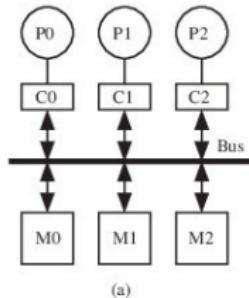


Quad-processor Opteron Motherboard

- Non-uniform memory access (NUMA)
- Intra/Interboard connection with Hypertransport HTX-technology

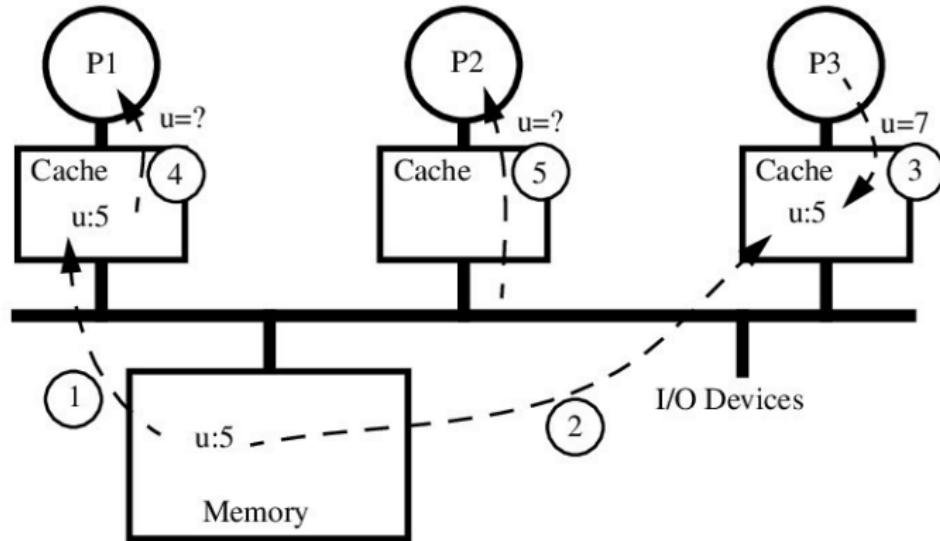
Dynamic Connection Networks

Line transmission: Truly electric connection from source to target.



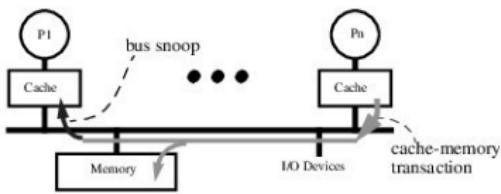
- (a) Bus: connects only two units at a time, thus is not scalable. Advantages: cheap, cache coherency by snooping.
- (b) Crossbar: Complete permutation realisable, but: P^2 switching units.
- (c) Ω network: $(P/2) \log P$ switching units, no complete permutation possible, each stage is *perfect shuffle*, simple routing.

Cache Coherency: An Example

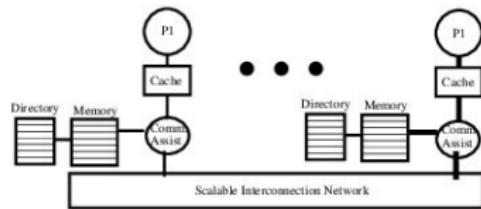


Cache Coherency: Protocol Types

Snooping based protocols



directory based protocols



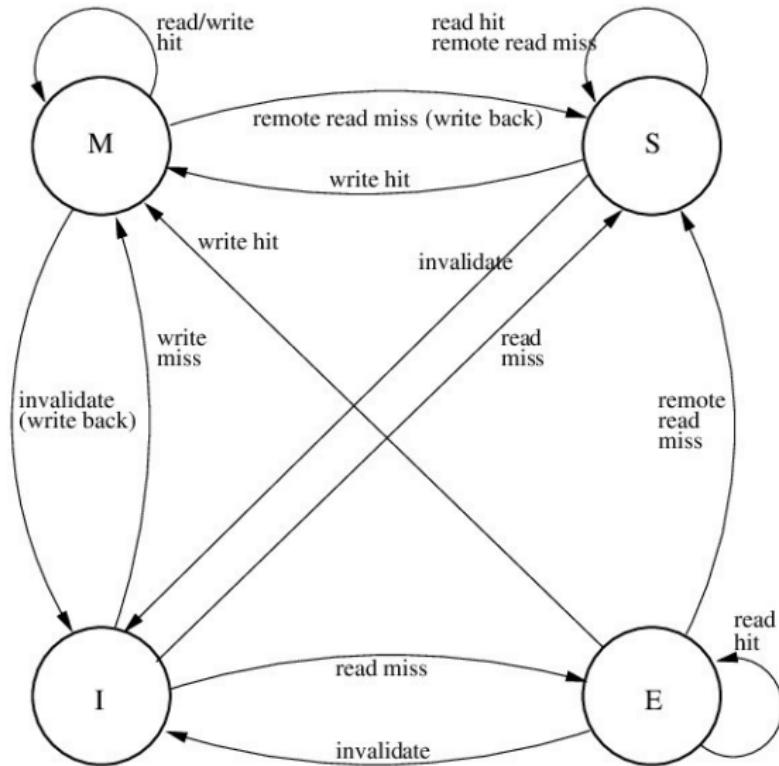
Cache Coherency: Bus Snooping, MESI

- Bus enables simple, efficient protocol for cache-coherency.
- Example MESI: Each cache block has one of the following states:

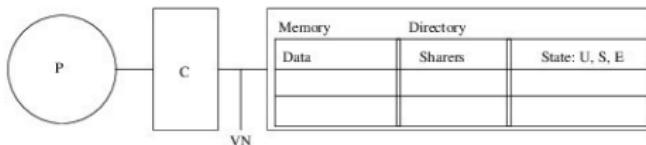
Status	Meaning
E	Entry valid, memory up-to-date, no copies exist
S	Entry valid, memory up-to-date, further copies exist
M	Entry valid, memory invalid, no copies exist
I	Entry is not valid

- Extends write-back protocol by cache coherency.
- Cache controller monitors the bus traffic (snoops) and performs the following state transitions (from the point of view of a controller):

Cache Coherency: Bus Snooping, MESI



Directory-based Cache Coherency I



States:

Cache-Block State	Description	Main Memory Block State	Description
I	Block invalid	U	noone has the block
S	≥ 1 copies exist, caches and memory are up-to-date	S	see left
E	exactly one has written the block (equals M in MESI)	E	see left

Directory-based Cache Coherency II

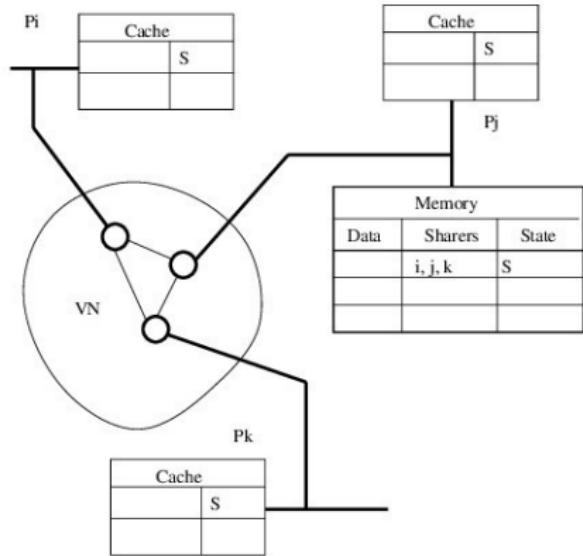
State transitions (view of directory):

Z	Action	Succ.	Description
U	read miss	S	Block is transmitted to cache, bit vector stores who has the copy.
	write miss	E	Block is transmitted to the requesting cache, bit vector contains who has the valid copy.
S	read miss	S	requesting cache gets copy from the memory and is registered in bit vector.
	w miss(hit)	E	Requester gets (if miss) a copy of the memory, directory sends invalidate to all remaining owners of a copy.
E	read miss	S	Owner of the block is informed, this sends block back to home mode and changes to state S, directory sends block to requesting cache.
	write back	U	Owner wants to replace cache block, data are written back, no one has the block.
	write miss	E	Owner changes. Previous owner is informed and sends block to home node, this sends the block to new owner.

Variant: COMA (Cache Only Memory Architecture)

Directory-based Cache Coherency III: Example

Situation: Three processors P_i , P_j , und P_k have a cache line in state shared.
Home node of this memory block is P_j



Actions:

- Processor P_i writes into the cache line (write hit): Message to directory, this informs caches of P_j and P_k , succeeding state is E in P_i
- Processor P_k reads from this block (read miss): Directory fetches block of P_i , directory sends block to P_k

Directory-based Cache Coherency IV: problem cases

Problems of ccNUMA architectures:

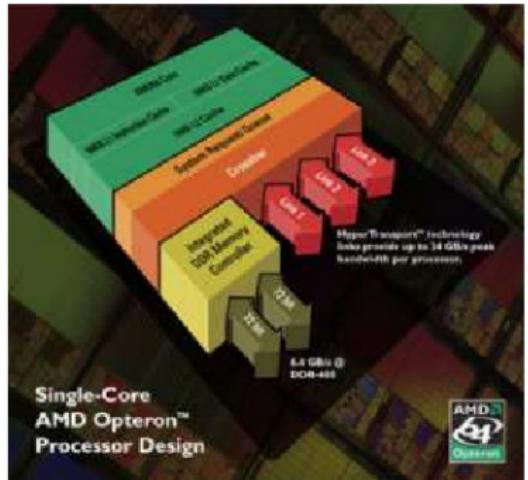
- false sharing: Two processors read and write different memory locations, that are by accident in the same block (probability increases with block size, Origin: 128 byte)
- capacity miss: Data amount, that a processor handles (working set), does not fit into the cache and the data are in the main memory of another processor

Solution for the capacity problem: Cache Only Memory Architecture (COMA), Software Distributed Shared Memory. Pages of the main memory (i.e. 4-16 KB) can be migrated automatically, combination with virtual memory mechanism.

Examples I: Intel Xeon MP

- IA32 architecture (as P4)
- Cache coherency protocol MESI
- Hyperthreading technique (2 logical CPUs)
- Integrated 3-level cache architecture (-1 MB L2, -8 MB L3)
- Machine Check Architecture (MCA) for external und internal buses, cache, translation look-aside buffer and instruction fetch unit
- Intel NetBurst Microarchitecture

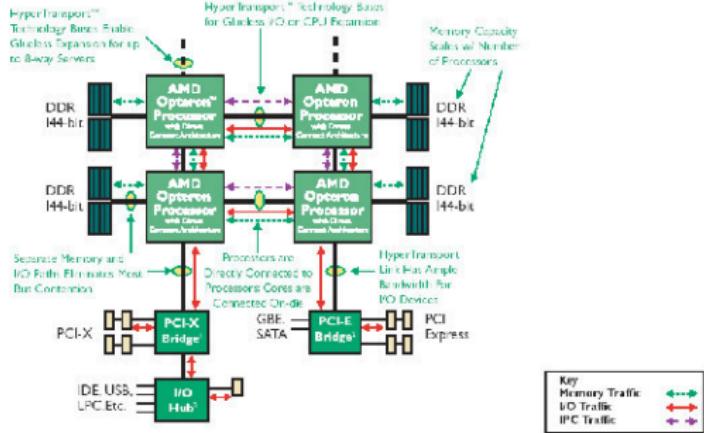
Examples II: AMD Opteron



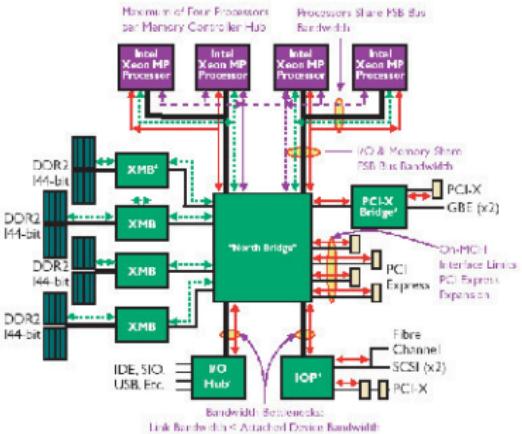
- Direct connect architecture
- On-Chip DDR memory controller
- HyperTransport technology
- Cache coherency protocol MOESI MESI states + 5th state direct data transfer between CPU caches via Hypertransport
- 64-bit Data/Address path, 48-bit virtual address space
- ECC for L1/L2 and DRAM with hardware scrubbing
- 2 additional pipeline stages
- many IPCs (instructions per cycle) through advanced branch prediction

Examples III: Server Architecture AMD vs INTEL

AMD Opteron™ Processor-based 4P Server



Intel Xeon MP Processor-based 4P Server



www.amd.com/us-en/assets/content_type/DownloadableAssets/AMD_Opteron_Streams_041405_LA.pdf

Examples III: Server Architecture AMD vs INTEL

Server System Comparison	AMD Opteron®	Intel Xeon®	Intel Xeon®	Intel Xeon MP®	Intel Itanium 2®
Modular, glueless scalability	Yes	Requires Northbridge	Requires Northbridge	Requires Northbridge	Requires Northbridge
SMP Capabilities	Up to 8-way	Up to 2-way	Up to 2-way	Up to 4-way	Up to 4-way
Direct Connect Architecture	Yes	No	No	No	No
Dual-Core technology	Yes	No	No	No	No
High Performance 32-bit and 64-bit computing	AMD64	No	EM64T	EM64T	No
HyperTransport® technology	Yes	No	No	No	No
Integrated DDR memory controller	Yes	No	No	No	No
Front Side Bus frequency	1.4 – 2.6GHz	533MHz	800MHz	667MHz	400MHz
Front Side Bus bandwidth	11.2 – 20.8GB/s*	4.3GB/s	6.4GB/s	10.6GB/s	6.4GB/s
Maximum Inter-processor bandwidth	8.0GB/s	4.3GB/s	6.4GB/s	10.6GB/s	6.4GB/s
Memory support	DDR200/266/333/400	DDR266	DDR333/DDR2-400	DDR266/333/DDR2-400	DDR200
Memory Bandwidth 2P System	12.8GB/s	4.3GB/s	6.4GB/s	12.8GB/s	6.4GB/s
Memory Bandwidth 4P System	25.6GB/s**	N/A	N/A	12.8GB/s	6.4GB/s
L1 cache size (max.)	64KB (Data) + 64KB (Instruction) per core	8KB + 12k mop	16KB + 12k mop	16KB + 12k mop	32KB
L2 cache size (max.)	1MB per core	512KB	2MB	1MB	256KB
L3 cache size (max.)	N/A	2MB	N/A	8MB	9MB
Maximum I/O bandwidth 2P System	24.0GB/s†	3.2GB/s	12.3GB/s	14.0GB/s	6.4GB/s
Maximum I/O bandwidth 4P System	32.0GB/s**	N/A	N/A	14.0GB/s	6.4GB/s
SIMD Instruction Set Support	SSE, SSE2, SSE3	SSE, SSE2	SSE, SSE2, SSE3	SSE, SSE2, SSE3	N/A

Parallel Computer Architecture II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

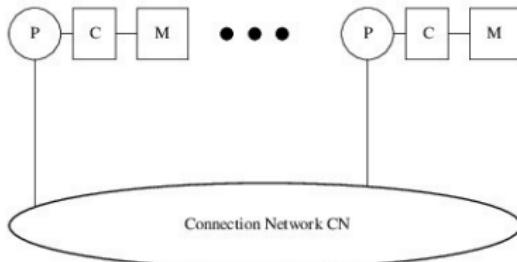
Parallel Computer Architecture II

- Multiprocessor architectures
- Message passing
- Network topologies
- Example architectures
- Routing
- TOP 500
- TOP2 Architectures

Classification of MIMD Architectures

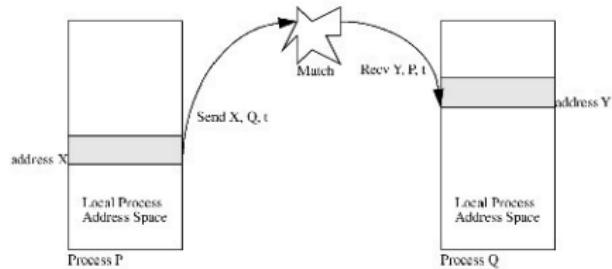
- Physical memory arrangement
 - ▶ shared memory
 - ▶ distributed memory
- Address space
 - ▶ global
 - ▶ local
- Programming model
 - ▶ shared address space
 - ▶ message passing
- Communication structure
 - ▶ Memory coupling
 - ▶ Message coupling
- Synchronization
 - ▶ semaphores
 - ▶ barriers
- Latency treatment
 - ▶ latency hiding
 - ▶ latency minimization

Distributed Memory: MP



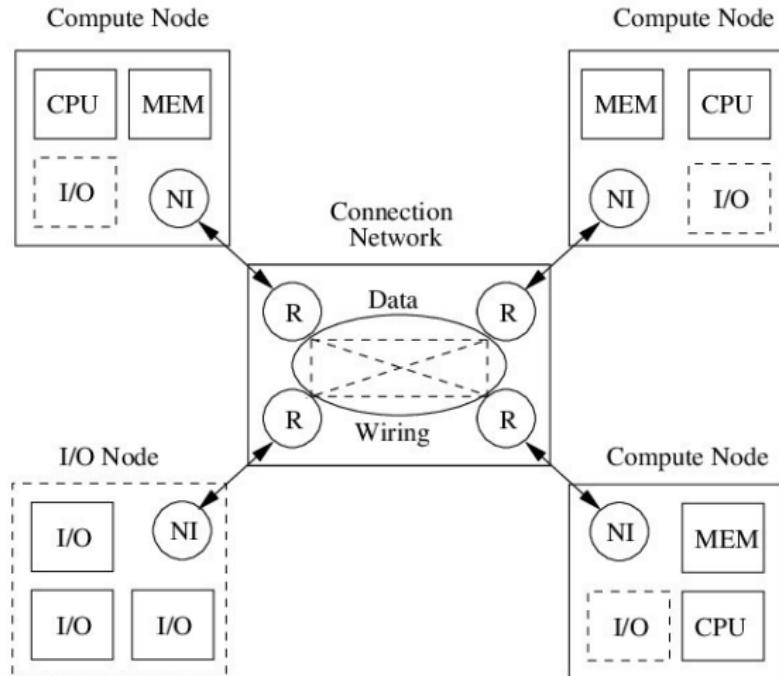
- Multi processors have a *local address space*: Each processor can access only its associated memory.
- Interaction with other processors exclusively by sending of messages.
- Processors, memory and cache are standard components: Full utilization of the price advantage of high quantity of units.
- Connection network ranging from Fast Ethernet to Infiniband.
- Approach with highest scalability: IBM BlueGene > 100 K processors

Distributed Memory: Message Passing



- Processes communicate data between distributed address spaces
- Explicit message passing is necessary
- Send-/Receive operations

A Generic Parallel Computer Architecture



Generic approach of a scalable parallel computer with distributed memory

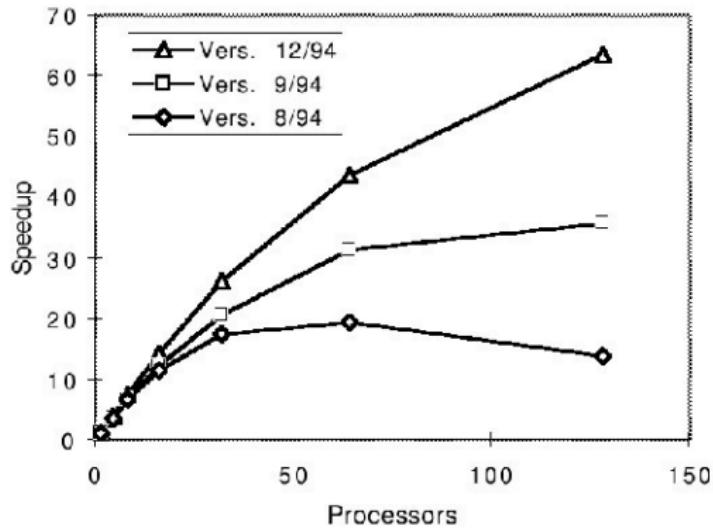
Scalability Parameters

Parameters, that drive the scalability of a computing system:

- Bandwidth [MB/s]
- Latency [μs]
- Costs [\$]
- Physical size [m^2, m^3]
- Power consumption [W]
- Fault tolerance / recovery abilities

A truly scalable architecture should avoid any hard limits!

Influence of Parallel Software?



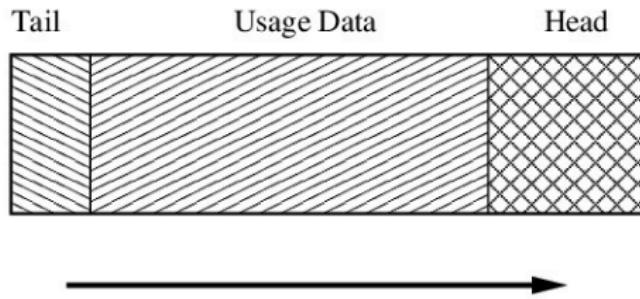
from Culler, Singh, Gupta: Parallel Computer Architecture

Although the parallel architecture scales scalable software is prerequisite for scalable numerical computation.

Message Passing

Memory block (of variable length) shall be copied from one memory to another
More precise: From the address space of one process to the address space
of another process (running on a different processor)

The connection network is packet oriented. Each message is subdivided in
packets of fixed length (e. g. 32 byte to 4 kbyte)

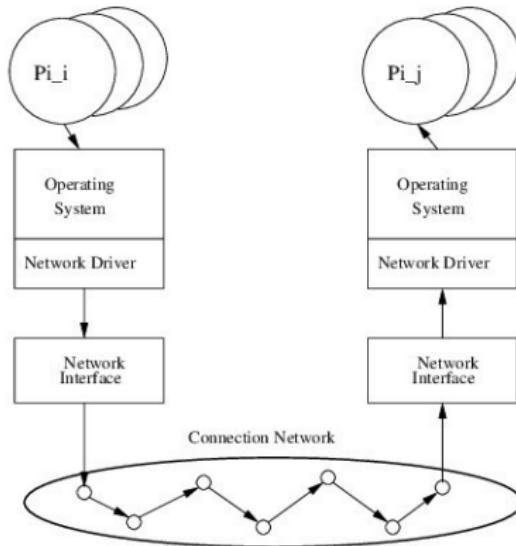


Head: target processor, tail: check sum

Communication protocol: acknowledgment whether packet has arrived valid,
flow control

Message Passing

Layer model (Hierarchy of protocols):

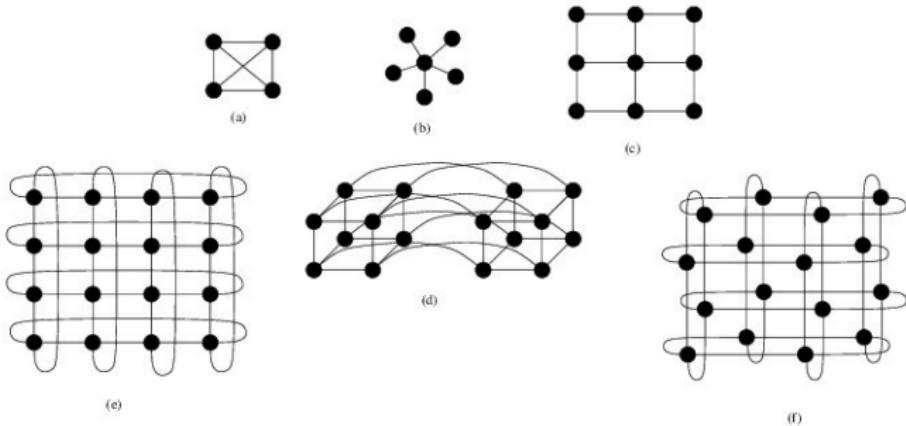


Model of transmission time:

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

t_s : setup time (latency), t_b : time per byte, $1/t_b$: bandwidth, dependent on protocol

Network Topology I



(a) full connected, (b) star, (c) array
(d) hypercube, (e) torus, (f) folded torus

- *Hypercube*: of dimension d has 2^d processors. Processor p is connected to q if their binary representation differs *in exactly one bit*.
- Network node: Earlier (before 1990) this was the processor itself, nowadays it is a dedicated communication processor.

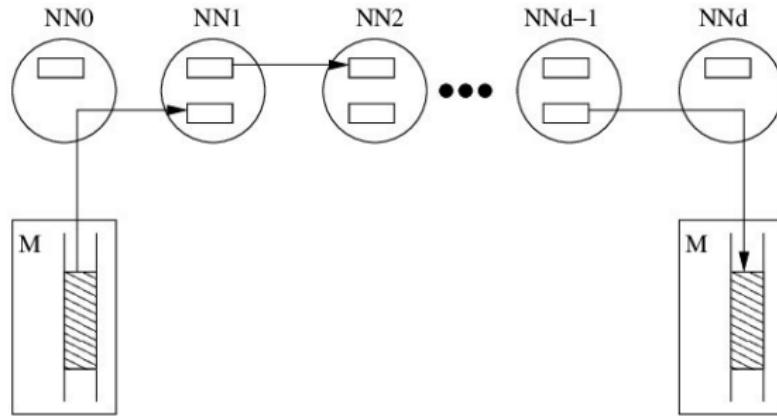
Network Topology II

Reference parameters:

Network topology	Node-degree K	Wire-count L	Diameter D	Bisection-bandwidth B	Symmetry
Full Connectivity	$N - 1$	$N(N - 1)/2$	1	$(N/2)^2$	yes
Star	$N - 1$	$N - 1$	2	$\lfloor N/2 \rfloor$	no
2D Grid	4	$2N - 2\sqrt{N}$	$2(\sqrt{N} - 1)$	\sqrt{N}	no
3D Torus	6	$3N$	$3\lfloor \sqrt{N}/2 \rfloor$	$2\sqrt{N}$	yes
Hypercube	$\log_2 N$	$nN \log_2 N$	n	$N/2$	yes
k-ary n-cube ($N = k^n$)	$3N$	$n\lfloor k/2 \rfloor$	nN	$2k^{n-1}$	yes

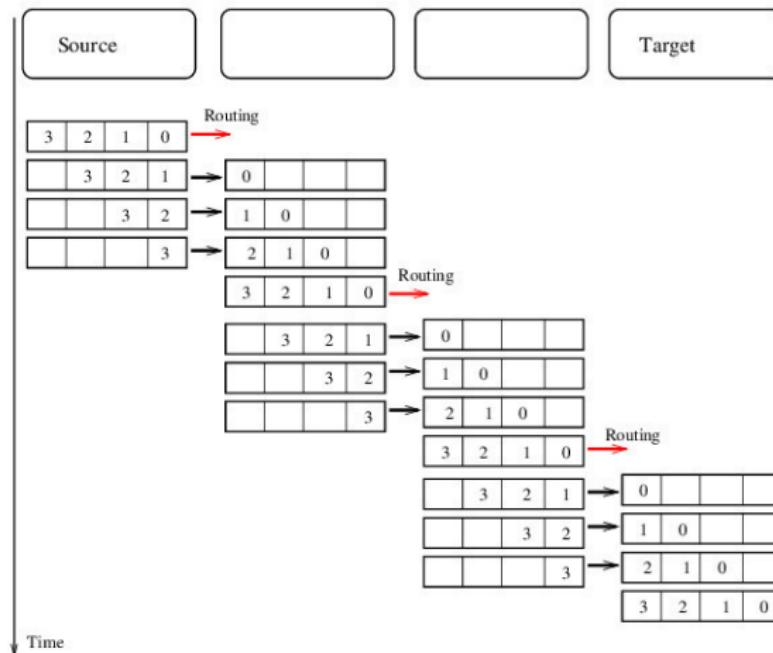
Store & Forward Routing

Store-and-forward routing: Message of length n is subdivided into packets of length N . Pipelining on packet level: Packet is stored completely in the network node.



Store & Forward Routing

Transmission of a packet:



Store & Forward Routing

Run time:

$$\begin{aligned} t_{SF}(n, N, d) &= t_s + d(t_h + Nt_b) + \left(\frac{n}{N} - 1\right)(t_h + Nt_b) \\ &= t_s + t_h \left(d + \frac{n}{N} - 1\right) + t_b(n + N(d - 1)). \end{aligned}$$

t_s : time, that is needed on source and target computer until the network is instructed with the message transmission, respectively until the receiving process is informed. This is the software share of the protocol.

t_h : time that is necessary to transmit the first byte of a message from a network node to another one (node latency, hop-time).

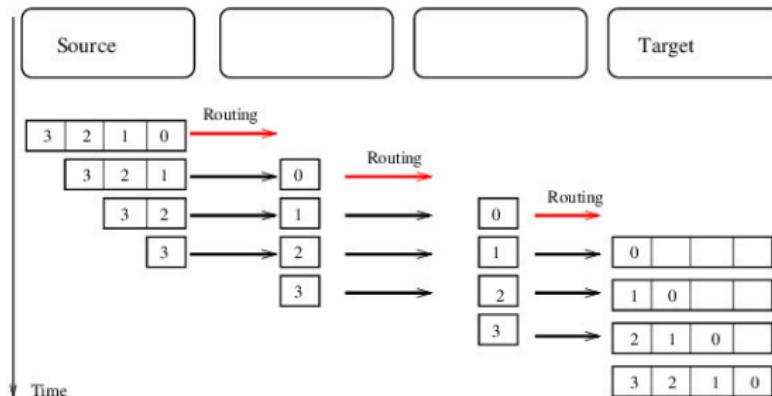
t_b : time for transmission of a byte from network node to network node

d : Hops to reach the target node.

Cut-Through Routing

Cut-through routing or wormhole routing: Packets are not buffered, each word (so called *flit*) is routed immediately to the next network node.

Transmission of a packet:



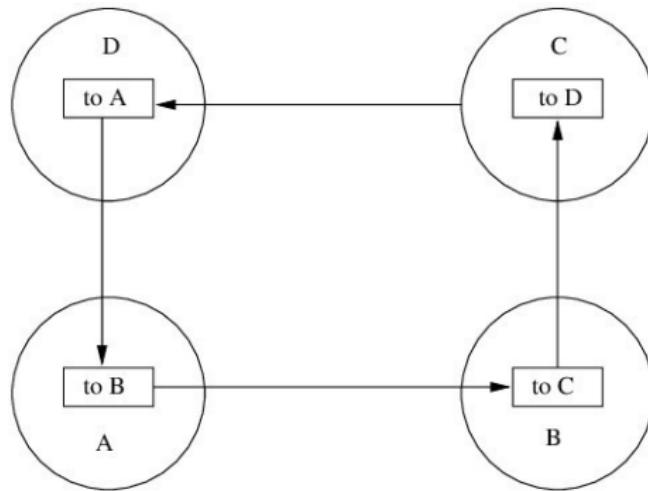
Run time:

$$t_{CT}(n, N, d) = t_s + t_h d + t_b n$$

Time for short message ($n = N$): $t_{CT} = t_s + dt_h + Nt_b$. Because of $dt_h \ll t_s$ (Hardware!) nearly distance independent

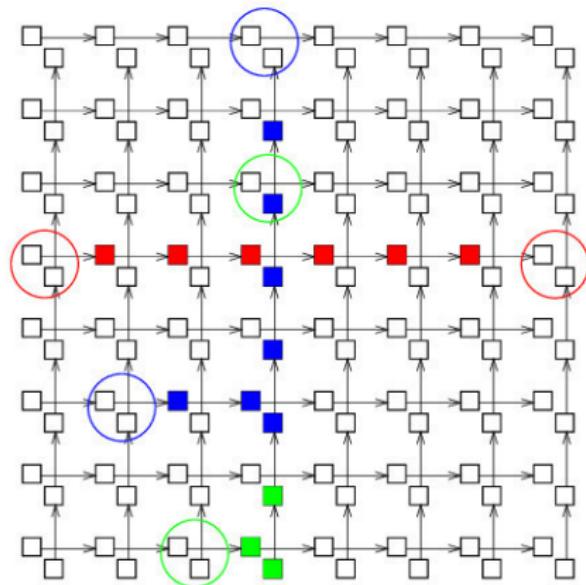
Deadlock

In packet transmitting network the danger of a *store-and-forward deadlock* exists:



Deadlock

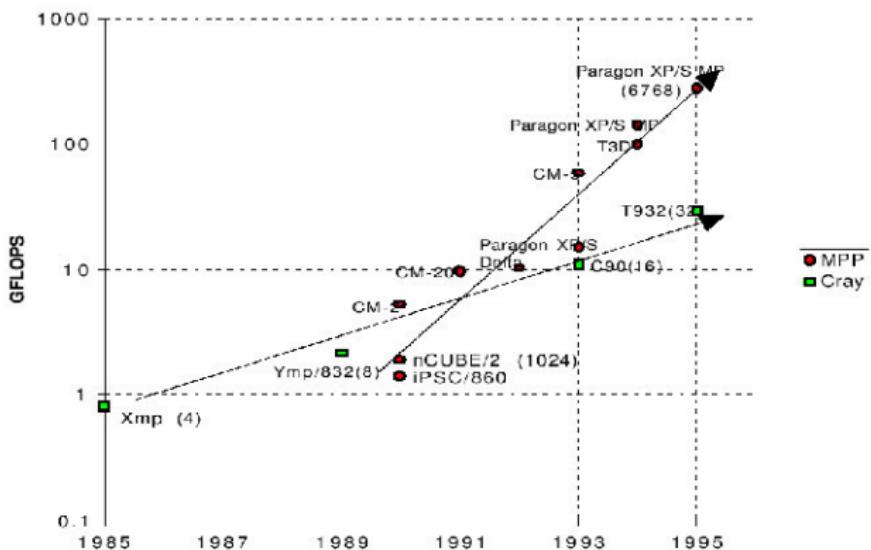
Together with cut-through routing:



Deadlock free „dimension order routing“.

Example 2D grid: Partition network into $+x$, $-x$, $+y$ and $-y$ networks each with individual buffers. Message is routed first in the sender row, then in the receiver column.

Multi-Processor Performance



from Culler, Singh, Gupta: Parallel Computer Architecture

Parallel Computer Architecture III

Stefan Lang

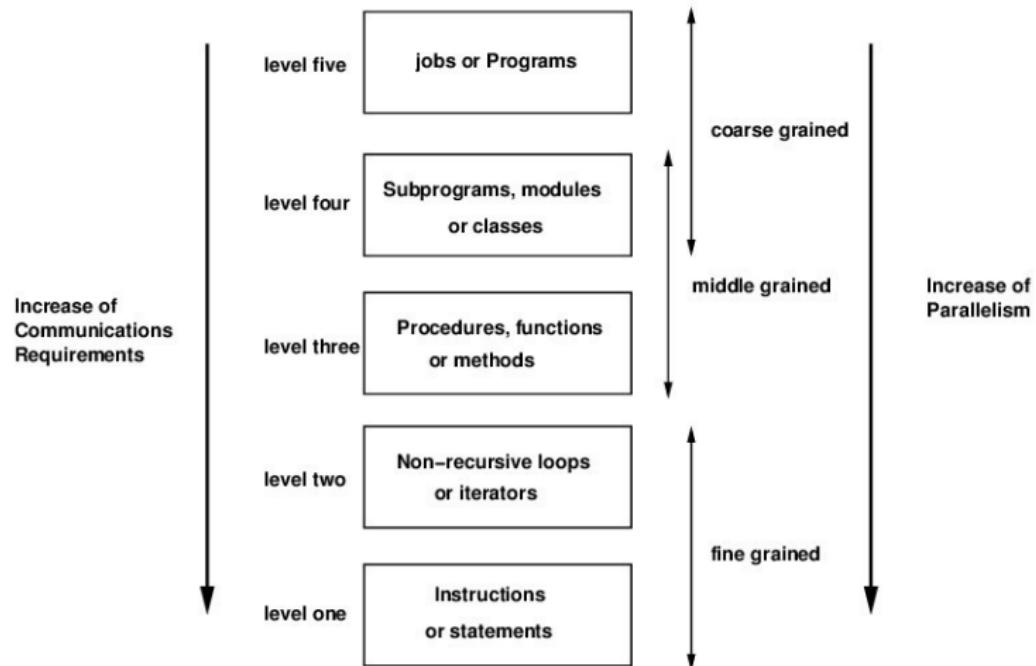
Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Parallel Computer Architecture III

- Parallelism and Granularity
- Graphic cards
- I/O
- Detailed study Hypertransport Protocol

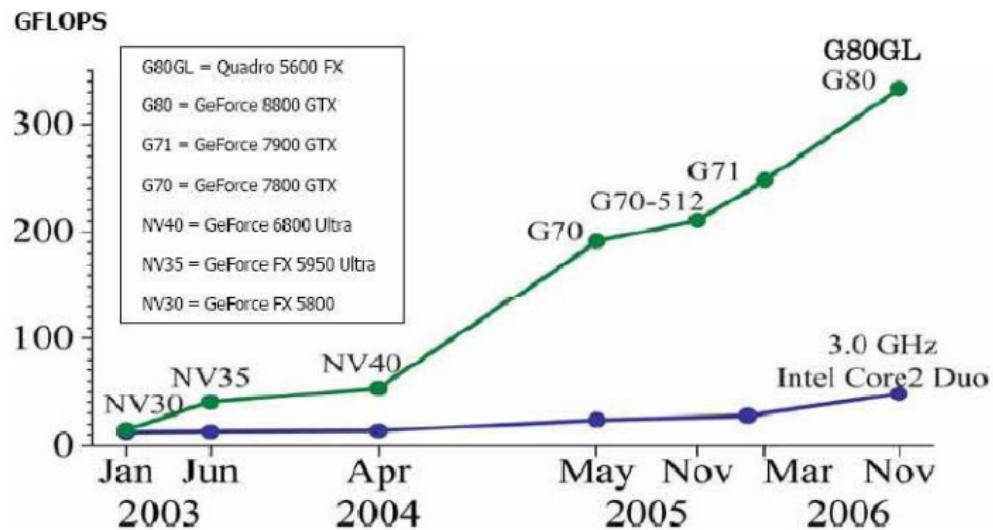
Parallelism and Granularity



Graphics Cards

- GPU = Graphics Processing Unit
- CUDA = Compute Unified Device Architecture
 - ▶ Toolkit by NVIDIA for direct GPU Programming
 - ▶ Programming of a GPU without graphical API
 - ▶ GPGPU
- compared to CPUs strongly increased computing performance and storage bandwidth
- GPUs are cheap and broadly established

Computing Performance: CPU vs. GPU



Graphics Cards: Hardware Specification

GeForce GTX 285

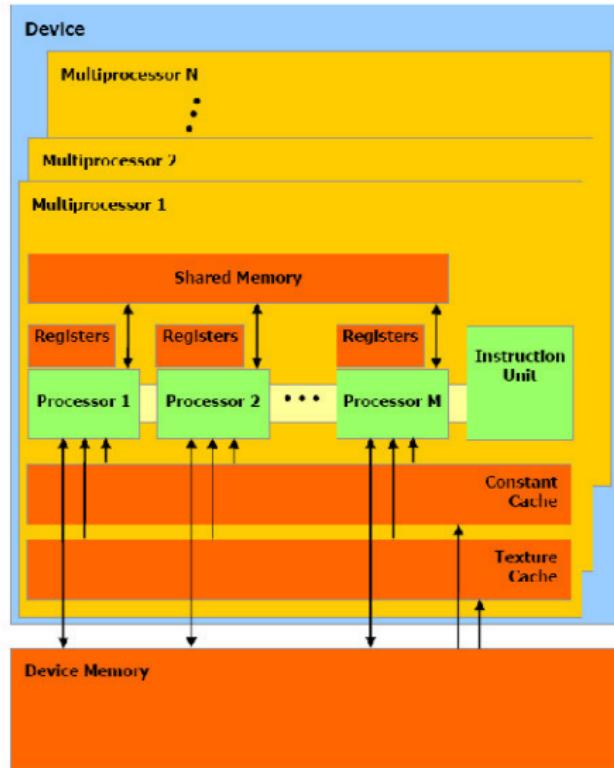
Fab (nm)	55
Transistors (million)	1400
Memory (MB)	1024
Multiprocessors	30
Streaming Processors	240
Shader Clock (MHz)	1476
Memory Bandwidth (GB/s)	159
Memory Bus width (bit)	512
SP GFLOPs (MADD + MUL)	1063
DP GFLOPs (MADD)	89
TDP (Watt)	183
Price (EUR)	~350



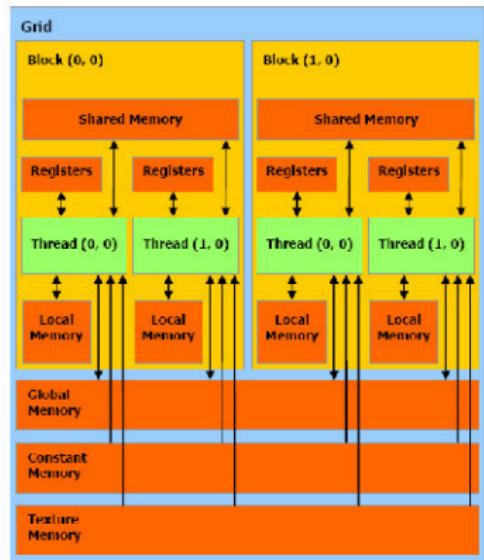
Chip Architecture: CPU vs. GPU



Graphics Cards: Hardware Design

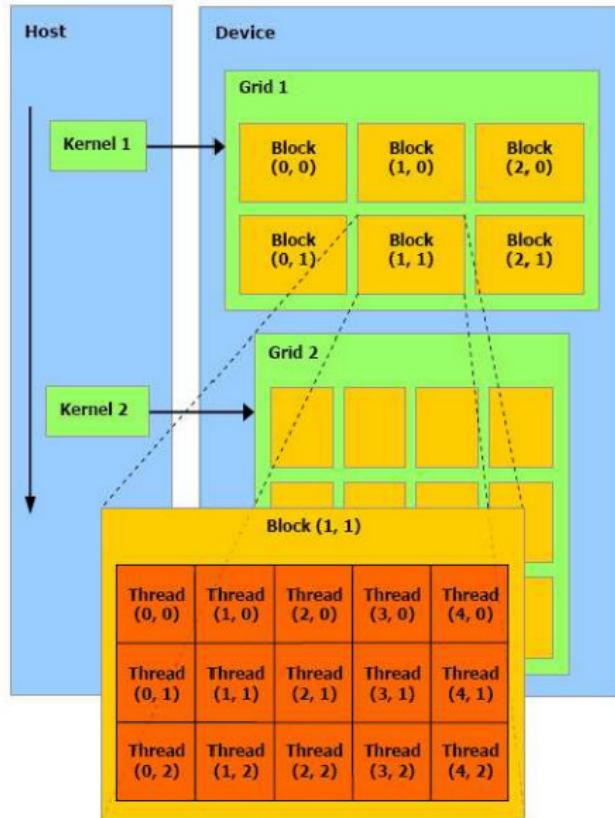


Graphics Cards: Memory Design

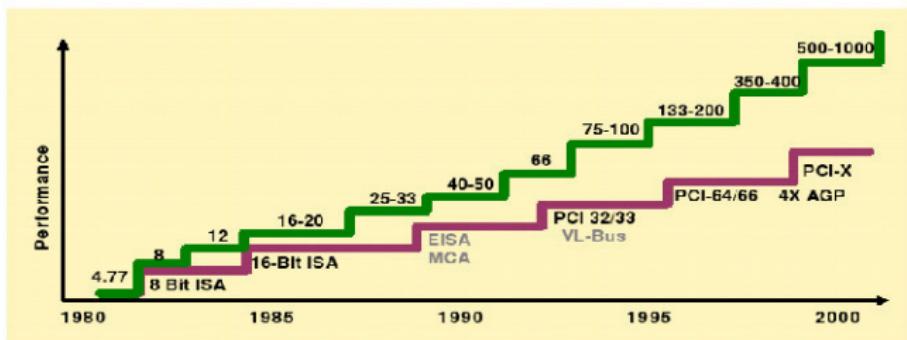


- 8192 registers (32-bit), in total 32KB per multiprocessor
- 16KB of fast shared memory per multiprocessor
- Large global memory (hundreds of MBs, e.g. 512MB-6GB)
- Global memory uncached, latency time 400-600 clock cycles
- Local memory is part of global memory
- Read-only constant memory
- Read-only texture memory
- Registers and shared memory are shared between blocks, that are executed on a single multiprocessor
- Global memory, constant and texture memory are accessible by the CPU

Graphics Card: Programming Modell

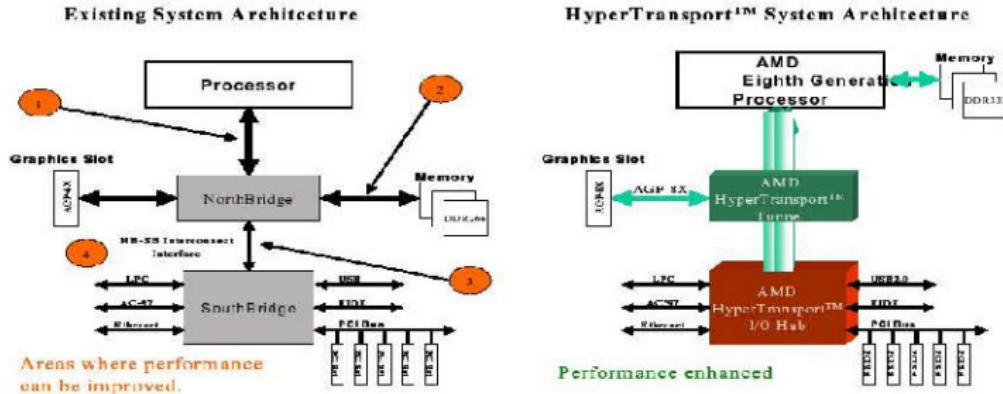


Performance Development concerning I/O



- Micro processor performance doubles about each 18 months
- Performance of the I/O architecture doubles in about 36 months
- Server and workstations necessitate different high speed buses (PCI-X, PCI-E, AGP, USB, SATA)
- High complexity and needless diversity with limited performance
- Requirements that need bandwidth increase: High-resolution 3D graphics and video (CPU – graphics processor), interprocessor (CPU – CPU), high performance networks e.g. Gigabit Ethernet and Infiniband (CPU – I/O)

Bottlenecks in I/O Section



Possible bottlenecks are:

- Front-Side Bus
- Memory interface
- Chip-to-Chip connection
- I/O to other bus systems

Standardization of I/O Requirements

Development of Hypertransport (HT) I/O connection architecture (since 1997)

- HyperTransport is an In-The-Box solution (intraconnect technology)
- Complementary technique to network protocols like Infiniband and 10 Gb/ 100 Gb ethernet as Box-to-box solutions (interconnect technology)
- HT is open industry standard, no product (no license fees)
- Further development and standardization by consortium of industry partners, like AMD, Nvidia, IBM, Apple
- Former code name: Lightning Data Transfer (LDT)

Functional Overview

Feature/Function	HyperTransport Technology
<i>Bus Type</i>	Dual, unidirectional, point-to-point links
<i>Link Width</i>	2, 4, 8, 16, or 32 bits
<i>Protocol</i>	Packet-based, with all packets multiples of four bytes (32 bits). Packet types include Request, Response, and Broadcast, any of which can include commands, addresses, or data.
<i>Bandwidth (Each Direction)</i>	100 to 6400 Mbytes/s
<i>Data Signaling Speeds</i>	100 MHz to 1.6 GHz
<i>Operating Frequencies</i>	400, 600, 800, 1000, 1200, and 1600 Megatransfers/second
<i>Duplex</i>	Full
<i>Max Packet Payload or Burst Length</i>	64-byte packet
<i>Power Management</i>	ACPI-compatible
<i>Signaling</i>	1.2-V Low-Voltage Differential Signaling (LVDS) with a 100-ohm differential impedance
<i>Multiprocessing Support</i>	Yes
<i>Environment</i>	Inside the box
<i>Memory model</i>	Coherent and noncoherent

Design Goals in the Design of HyperTransport

- Improvement of system performance
 - ▶ Higher I/O bandwidth
 - ▶ Avoidance of bottlenecks by slower devices in critical information paths
 - ▶ lower count of system buses
 - ▶ lower response time (low latency)
 - ▶ reduced energy consumption
- Simplification of system design
 - ▶ Unified protocol for In-box connections
 - ▶ Usage of small pin counts for high packaging density and to ensure low costs
- Higher I/O flexibility
 - ▶ Modular bridge architecture
 - ▶ Different bandwidth in upstream/downstream direction
- Compatibility with existing systems
 - ▶ Supplement for standardized, external buses
 - ▶ Minor implications on existing operating systems and drivers
- Extendability for system network architectures (SNA busses)
- High scalability in multiprocessor systems

Flexible I/O Architecture

Hypertransport architecture is organized in 5 layers

Structure is adopted from **Open-System-Interconnection** (OSI) reference model

- **Bit transmission layer:** Physical and electrical properties of data, control, clockwires
- **Data connection layer:** Initialisation and configuration of connections, periodic cyclic redundancy (CRC), Dis-/Reconnection, packet generation control flow and error management,
- **Protocol layer:** Virtual channels and commands
- **Transaction layer:** Read- and write actions by using the data connection layer
- **Session layer:** Power-, interrupt- and system management

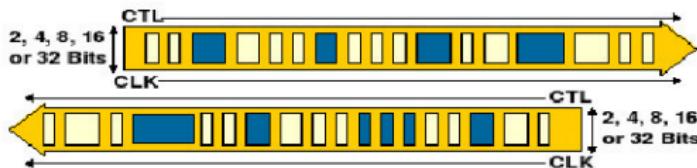
Device classes and -configurations

3 device classes are defined regarding function and location inside the HT-chain: Cave, Tunnel und Bridge

- HT Bridge: Connector between primary side (CPU resp. memory) and sekundary side (HT devices) of a chain
- HT Tunnel: has two sides each with a receive and a send unit, e.g. network card of bridge to further protocol
- HT Cave: marks end of a chain and has only one communication side. By connecting of minimal a HT bridge and a HT Cave an easy HT chain can be established.

Bit Transmission Layer I

Establishing a HT connection



- Two unidirectional point-to-point data paths
- Data path width: 2, 4, 8 and 16 bit depending on device requirements
- Commands, addressing and data (CAD) use the same signal wires
→ smaller wire count, smaller packets, smaller energy consumption, improved thermal properties
- Packets contain CAD and are multiple of 4 Byte (32 bit)
- Connections with lower than 32 bit use subsequent cycles to transmit packets completely

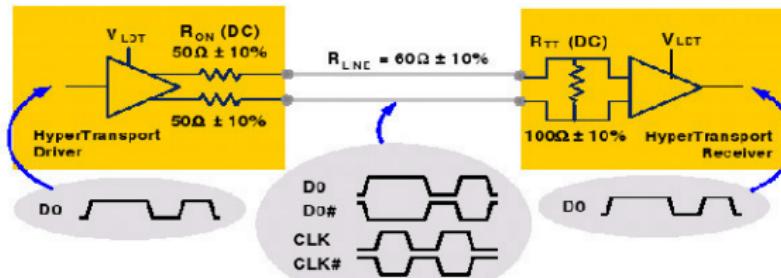
High performance and scalability

- High data rates because of low voltage differential signaling (LVDS, 1,2V ± 5%)
- Scalable bandwidth by scaling of transmission frequency and link width

Bit Transmission Layer II

Hypertransport Low Voltage Differential Signaling

- Differential signal transmission using two wires: voltage difference ($\pm 0,3$ V) represents logic state Logic change by repoling of wires (symmetric signal transmission, double-ended)
- HT signal transmission complies to extended IEEE LVDS standard (1,2 V instead of 2,5 V)
- 60 cm maximal wire length at 800 Mbit/s
- Transceivers are integrated into control chips, 100 Ω impedance avoids reflections
- Simple realizable using standard 4-layer PCBs (Printed Circuit Boards) and ready for future



Bit Transmission Layer III

Electric signals

Signal Name	Description	Comment
CAD	Commands, Addresses and Data: Carries command, address, or data information.	CAD width can be different in each direction.
CTL	Control: Used to distinguish control packets from data packets.	
CLK	Clock: Forwarded clock signal.	Each byte of CAD has a separate clock signal. Data is transferred on each clock edge.
PWROK	Power OK: Power and clocks are stable	Single-ended.
RESET#	HyperTransport Technology Reset: Resets the chain.	Single-ended.
LDTSTOP#	HyperTransport Technology Stop: Enables and disables links during system state transitions.	Used in systems requiring power management. Single-ended.
LDTREQ#	HyperTransport Technology Request: Requests re-enabling links for normal operation.	Used in systems requiring power management. Single-ended.

For any 8 bit data width there is a clock wire from sender to receiver that is used to scan the data on the 8 data wires at the receiver (source synchronous clock)

→ minimal deviation from source clock

Bit Transmission Layer IV

Band width scaling

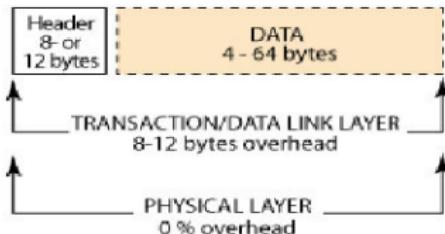
- Data transmission by CAD at rising and falling side of clock signal (DDR)
→ connection cycle at 800 MHz corresponds to 1600 MHz data cycle
- Wire count enables adaptation onto band width needs
 - 2×16 CAD bits + 800 GHz clock = $2 \times 3,2$ GByte bandwidth (103 Pins)
 - 2×2 CAD bits + 400 MHz clock = 2×200 MByte bandwidth (24 Pins)

Link Width (Each Way)	2	4	8	16	32
Data Pins (total)	8	16	32	64	128
Clock Pins (total)	4	4	4	8	16
Control Pins (total)	4	4	4	4	4
Subtotal (High Speed)	16	24	40	76	148
V_{LDT}	2	2	3	6	10
GND	4	6	10	19	37
PWROK	1	1	1	1	1
RESET#	1	1	1	1	1
Total Pins	24	34	55	103	197

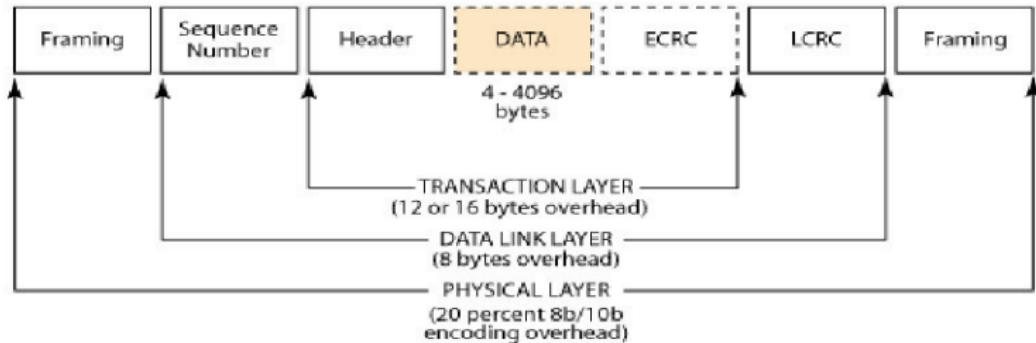
Connection Layer

Packet structure of HT compared to PCI-E

HyperTransport Packet Format



PCI Express Packet Format



Protocol- and Transaction Layer

- Protocol layer: Commands, virtual channels and flow control
- Transaction layer: Performs actions e.g. read requests and responses

Command overview

VIRTUAL CHANNEL	COMMAND	COMMENT
Posted	Posted Write	Followed by data packet(s).
	Broadcast	Issued by host bridge downstream to communicate information to all devices.
	Fence	All posted requests in a stream cannot pass it.
Non-Posted	Non-Posted Write	
	Read	Designates whether response can pass posted requests or not.
	Flush	Forces all posted requests to complete.
	Atomic Read-Modify-Write	Generated by I/O devices or bridges and directed to system memory controlled by the host.
	Read Response	Response to read command, is followed by data packet(s).
	Target Done	A transaction not requiring returned data has completed at its target.
Responses		

Packet Structure

Bit Time	7	6	5	4	3	2	1	0
0		SeqID[3:2]			Cmd[5:0]			
1	PassPW		SeqID[1:0]			UnitID[4:0]		
2				Command-Specific				
3				Command-Specific				
4					Addr[15:8]			
5					Addr[23:16]			
6					Addr[31:24]			
7					Addr[39:32]			

Shared Memory Programming Models I

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Shared Memory Programming Models I

Communication by shared memory

- Critical section
- Mutual exclusion: Petersons algorithm
- OpenMP
- Barriers – Synchronisation of all processes
- Semaphores

Critical Section

What is a critical section?

We consider the following situation:

- Application consists of P concurrent processes, these are thus executed simultaneously
 - Instructions executed by one process are subdivided into interconnected groups
 - ▶ critical sections
 - ▶ uncritical sections
 - Critical section: Sequence of instructions, that perform a read or write access on a *shared variable*.
 - Instructions of a critical section that may not be performed simultaneously by two or more processes.
- it is said only a single process may reside within the critical section.

Mutual Exclusion I

2 types of synchronization can be distinguished

- Conditional synchronisation
- Mutual exclusion

Mutual exclusion consists of an *entry protocol* and an *exit protocol*:

Program (Introduction of mutual exclusion)

parallel *critical-section*

```
{  
    process  $\sqcap$  [int  $p \in \{0, \dots, P - 1\}$ ]  
    {  
        while (1)  
        {  
            entry protocol;  
            critical section;  
            exit protocol;  
            uncritical section;  
        }  
    }  
}
```

Mutual Exclusion II

The following criteria have to be matched:

- ① *Mutual exclusion.* At most one process executed the critical section at a time.
- ② *Deadlock-freeness.* If two or more processes try to enter the critical section exactly one has to succeed within limited time.
- ③ *No unnecessary delay.* If a process wants to enter the critical section while all others process their uncritical sections this may not be prevented.
- ④ *Final entry.* If a process tries to enter the critical section, then this must be allowed after limited waiting time (therefore is assumed, that each process in the critical section also leaves it again).

Petersons Algorithm: A Software Solution

We consider at first only two processes and develop the solution step by step

...

First approach: Wait until the other is *not* inside

```
int in1=0, in2=0;      // 1=inside
```

Π_1 :

```
while (in2) ;  
in1=1;  
crit. section;
```

Π_2 :

```
while (in1) ;  
in2=1;  
crit. section;
```

- No machine instructions are necessary
- Problem: Reading and writing is not atomic

Petersons Algorithm: Second Variant

First set, then test

```
int in1=0, in2=0;
```

Π_1 :

```
in1=1;
```

```
while (in2) ;
```

```
crit. section;
```

Π_2 :

```
in2=1;
```

```
while (in1) ;
```

```
crit. section;
```

Problem: deadlock possible

Petersons Algorithm: Third Variant

Solve deadlock by choosing one process

Program (Petersons Algorithm for two processes)

parallel Peterson-2

```
{  
    int in1=0, in2=0, last=1;
```

process Π_1

```
{  
    while (1) {  
        in1=1;  
        last=1;  
        while (in2  $\wedge$  last==1) ;  
        crit. section;  
        in1=0;  
        uncrit. section;  
    }  
}
```

process Π_2

```
{  
    while (1) {  
        in2=1;  
        last=2;  
        while (in1  $\wedge$  last==2) ;  
        crit. section;  
        in2=0;  
        uncrit. section;  
    }  
}
```

Consistency Models

Previous examples are based on the principle of *Sequential Consistency*:

- ① *Read- and write operations are finished in the order of the program*
- ② *This sequence is for all processors consistently visible*

Here one expects that $a = 1$ is printed:

```
int a = 0, flag=0;           // important!
process Π1                  process Π2
...
a = 1;
flag=1;                      while (flag==0) ;
                               print a;
```

Here one expects that only for one the **if** condition is true:

```
int a = 0, b = 0;           // important!
process Π1                  process Π2
...
a = 1;
if (b == 0) ...
                               b = 1;
                               if (a == 0) ...
```

Consistency Models

Why is there no sequential consistency?

- *Reordering of instructions*: Optimizing compilers can reorder operations for efficiency reasons. Then the first example does not work any more!
- *Out-of-order execution*: e. g. read accesses shall pass slow write accesses (invalidate) (as long as it is not the same memory location). The second example does not work any more!

Total store ordering: Read access may only pass write access

Weak consistency: All accesses may pass each other

In-order sequence can be enforced by special machine instructions, e. g. *fence*: finish all memory accesses before a new one is started

This operations are inserted,

- through annotation of variables („synchronisation variable“)
- in parallel instructions (e. g. FORALL in HPF)
- by the programmer of synchronisation primitives (e. g. Semaphore)

Peterson for P Processes

Idea: Each process passes $P - 1$ stages, respectively the last arriving in a particular stage has to wait

Variables:

- $in[i]$: Stage $\in \{1, \dots, P - 1\}$ (!), that Π_i has reached
- $last[j]$: Number of process that arrived as the latest at stage j

Program (Petersons Algorithm for P processes)

parallel Peterson- P

{

```
    const int P=8;
    int in[P] = {0[P]};
    int last[P] = {0[P]};
```

...

}

Peterson for P Processes

Program (Petersons Algorithm for P Processes cont.)

parallel Peterson- P cont.

```
{  
    process  $\sqcap$  [int  $i \in \{0, \dots, P - 1\}$ ]  
    {  
        int  $j, k$ ;  
        while (1)  
        {  
            for ( $j=1; j \leq P - 1; j++$ ) // Traverse stages  
            {  
                in[ $i$ ] =  $j$ ; // I am in stage  $j$   
                last[ $j$ ] =  $i$ ; // I am the last of stage  $j$   
                for ( $k = 0; k < P; k++$ ) // test all others  
                    if ( $k \neq i$ )  
                        while (in[ $k$ ]  $\geq$  in[ $i$ ]  $\wedge$  last[ $j$ ] ==  $i$ );  
                }  
                critical section;  
                in[ $i$ ] = 0; // exit protocol  
                uncritical section;  
            }  
        }  
    }  
}
```

- $O(P^2)$ tests are necessary for entry
- Strategy is fair, who arrives first enters as first

Hardware Locks

Hardware operations to realize mutual exclusion:

- *test-and-set*: Check whether a memory location has value 0, if yes write the contents of a register into the memory location (as indivisible operation).
- *fetch-and-increment*: Get the content of a memory location in a register and increment the content of the memory location by 1 (as indivisible operation).
- *atomic-swap*: Interchange the content of a register with the content of a memory location in an indivisible operation.

In each of the machine instructions a read access followed by a write access has to be executed without break in between!

Hardware Locks

Goal: Machine instruction and cache coherency model ensure exclusive entry into the critical section and generate low traffic on the interconnection network

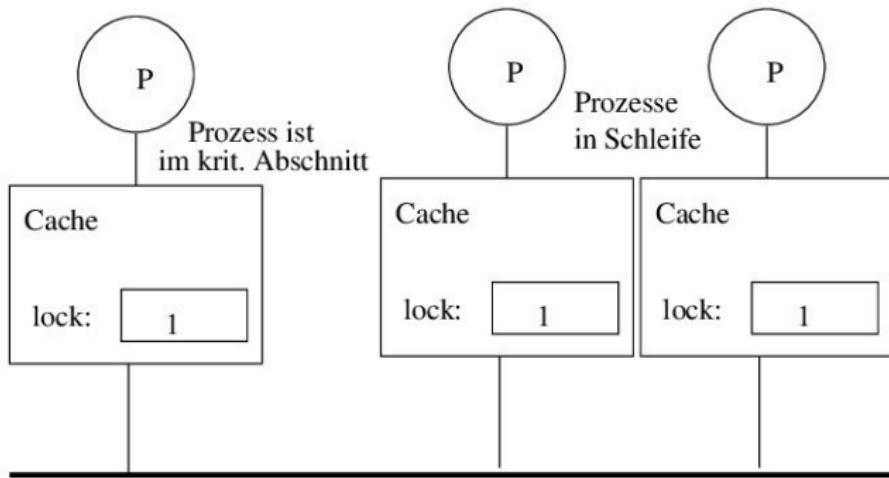
Program (Spin Lock)

```
parallel spin-lock
{
    const int P = 8;           // process count
    int lock=0;               // variable for protection

    process  $\sqcap$  [int  $p \in \{0, \dots, P - 1\}$ ]
    {
        ...
        while ( atomic--swap(& lock)) ;
        ...
        // critical section
        lock = 0;
        ...
    }
}
```

Hardware Locks

What occurs inside the system?



Both waiting processes generate high bus traffic!

Hardware Locks

Activity with MESI protocol: Variable *lock* is 0 and is in none of the caches

- Process Π_0 executes the *atomic – swap* operation
 - Read access induces a read miss, block is fetched from memory and obtains the state E (we take MESI as a basis).
 - Subsequent writing without further bus access, state change from E to M.
- other process Π_1 executes *atomic – swap* operation
 - Read miss induces Write-back of the block by Π_0 , the state of both copies is now S, after the read access.
 - Write access of Π_1 invalidates copy of Π_0 and state in Π_1 is M.
 - *atomic – swap* has result 1 in Π_1 and critical section is not entered by Π_1 .
 - If both processes execute the *atomic – swap* operation simultaneously the bus decides finally who wins.
- Assume cache C_0 of processor P_0 and also C_1 have each a copy of the cache block in state S before execution of the *atomic – swap* operation
 - Both read initially the value 0 from the variable *lock*.
 - In the following write access both compete for the bus to place their own Invalidate message.
 - The winner sets its copy into state M, the loser sets its into state I. The cache logic of the loser finds the state I when it writes and has to arrange that the *atomic – swap* instruction returns after all the value 1 (the atomic-swap instruction is yet not finished at this time).

Improved Lock

Idea: Do not perform any write access as long as the critical section is occupied

Program (Improved Spin Lock)

```
parallel improved-spin-lock
{
    const int P = 8;           // process count
    int lock=0;               // variable for protection

    process  $\sqcap$  [int  $p \in \{0, \dots, P - 1\}$ ] {
        ...
        while (1)
            if (lock==0)
                if (read-and-set(& lock)==0)
                    break;
            ...
            // critical section
        lock = 0;
        ...
    }
}
```

Improved Lock

- ➊ Problem: Strategy guarantees no fairness
- ➋ Situation with three processes: Two always alternate, while the third can enter
- ➌ Effort if P processes want to enter at a time: $O(P^2)$, instruction *lock* = 0 causes P bus transactions for cache block copies
- ➍ Solution is a queuing lock: During exit from the critical section the process chooses a successor

Ticketing algorithmus:

- Fairness with hardware lock
- Idea: Before lining up in the queue one draws a number. The one with the smallest number is the next to be chosen.

Ticketing Algorithm

Program (Ticketing Algorithm for P processes)

```
parallel Ticket
{
    const int P=8;
    int number=0;
    int next=0;

    process  $\sqcap$  [int  $i \in \{0, \dots, P - 1\}$ ]
    {
        int mynumber;
        while (1)
        {
            [mynumber=number; number=number+1];
            while (mynumber  $\neq$  next);
            critical section;
            next = next+1;
            uncritical section;
        }
    }
}
```

Ticketing Algorithm

- ④ Fairness is based on a small duration for drawing a number. Opportunity of a collision is small.
- ⑤ Works also for counter overflow, ($\text{MAXINT} > P$)
- ⑥ Incrementing of *next* is possible without synchronisation, since this always can only be done by one

Conditional Critical Section I

- Producer-Consumer problem:
 - ▶ m processes P_i (producers) generate requests, that shall be finished by n other processes C_j (consumers).
 - ▶ The processes communicate by a central waiting queue (WQ) with k positions.
 - ▶ Is the WQ full the producers have to wait, is the WQ empty the consumers have to wait.
- Problem: Waiting may not block the (exclusive) access onto the WQ!
- Critical section (manipulation of the WQ) may only be entered *if* WQ is not full (for producer), resp. not empty (for consumer).
- Idea: Entry on a trial basis and busy-wait

Conditional Critical Section II

Program (Producer–Consumer Problem with Active Waiting)

parallel producer-consumer-busy-wait

```
{  
    const int m = 8; n = 6; k = 10;  
    int orders=0;  
    process P [int 0 ≤ i < m] {  
        while (1) {  
            produce request;  
            CSenter;  
            while (orders==k){  
                CSexit;  
                CSenter;  
            }  
            store request;  
            orders=orders+1;  
            CSexit;  
        }  
    }  
}
```

```
process C [int 0 ≤ j < n] {  
    while (1) {  
        CSenter;  
        while (orders==0){  
            CSexit;  
            CSenter;  
        }  
        read request;  
        orders=orders-1;  
        CSexit;  
        process request;  
    }  
}
```

Conditional Critical Section III

- Permanent Entry and Exit of the critical section is inefficient if several are waiting (trick of the improved lock doesn't help)
- (Practical) solution: Random delay between *CSenter/CSexit, exponential back-off.*

OpenMP (Open Multi Processing) I: Approach

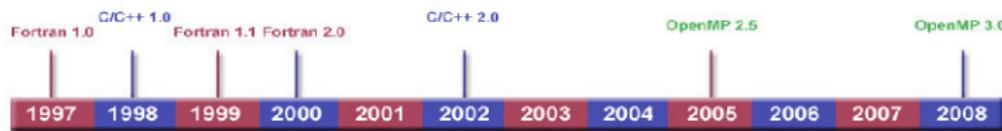
OpenMP is a parallel programming model on the basis of the following assumptions:

- A process uses multiple threads (lightweight processes)
- All threads share the same status variables of the program
- Each thread can own additional private variables
- Threads can run on different processors/cores
- Mechanisms for synchronization and for locking are provided

OpenMP II

What is OpenMP?

- API (application programming interface) to write multi-threaded applications
 - ▶ Compiler directives and library functions
 - ▶ Standardized for C/C++ and Fortran
- New standard under steady enhancement
- Important model, since many important companies are participating
- Parallelisation process using OpenMP
 - ▶ Program is parallelized in steps
 - ▶ Starting point is the serial version, which remains in many cases unchanged
 - ▶ Parallelism is not coded directly, but is influenced by directives



OpenMP Example I

Hello World:

- The original program is preserved
- Execution when an environment variable is set:

```
void main(void)
{
#pragma omp parallel
{
    printf("Hallo Welt\n");
}
}
```

```
(~): export OMP_NUM_THREADS=4
(~): ./hello-openmp
Hallo Welt
Hallo Welt
Hallo Welt
Hallo Welt
```

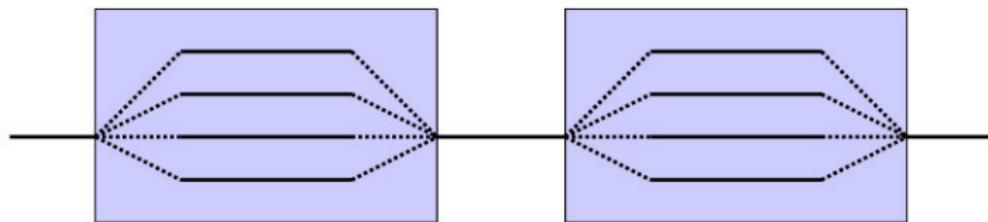
OpenMP Example II

Parallel Regions (blocks)

- By usage of the compiler directive `#pragma omp parallel` the following block is executed in parallel
- Therefore a set of threads is started
 - ▶ the thread count depends on the environment variable `OMP_NUM_THREADS`, that can be changed by the program
 - ▶ we speak of fork-join parallelism
- After all threads are finished, these are either terminated or remain waiting

Control flow in block 1

control flow in block 2



OpenMP Example III

- Primary application area of OpenMP is the parallelisation of loops
- **#pragma omp parallel for**
- i-loop will be executed simultaneously by OMP_NUM_THREADS threads

Matrix-Matrix multiplication

```
#pragma omp parallel for
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)
            C[i,j] = A[i,k] * B[k,j];
```

OpenMP Example IV

Runtime conditions

- If multiple threads read and write the same variable, inconsistencies can occur
- This is comparable to the already known inconsistencies in shared-memory architectures

Example: Scalar product

```
sum = 0.0;  
#pragma omp parallel for  
for (i = 0; i < N; i++)  
    sum = sum + x[i] * y[i];
```

OpenMP Example V

Solution 1: Locking

- To secure the addition of the summing-up one can declare these as atomic or critical
- Disadvantage: This is inefficient, because the threads can not work in parallel anymore

```
sum = 0.0;  
#pragma omp parallel for  
for (i = 0; i < N; i++)  
#pragma omp critical  
{  
    sum = sum + x[i] * y[i];  
}
```

OpenMP Example VI

Solution 2: Private variables

- In parallel regions specific variables can be declared as private
- This can be written in a more compact form

```
sum = 0.0;
#pragma omp parallel private(local_sum)
{
    local_sum = 0.0;
#pragma omp parallel for
    for (i = 0; i < N; i++)
        local_sum = local_sum + x[i] * y[i];

#pragma omp critical
    { sum = sum + local_sum; }
}
```

OpenMP Example VII

Solution 3: Reduction variables

- Such cases are typical, one can declare critical variables as reduction variables
- Within threads these are generated as private and then connected with an appropriate operation at the loop end (synchronized)

```
sum = 0.0;  
#pragma omp parallel for reduction (+ : sum)  
for (i = 0; i < N; i++)  
    sum = sum + x[i] * y[i];
```

OpenMP Pragmas I

- Directives (in C):
 - ▶ **#pragma omp clauses ...**
 - ▶ are ignored by non-OpenMP compilers
- Parallel regions (blocks):
 - ▶ **#pragma omp parallel**
 - ▶ following block `{...}` is executed in parallel
- Variable scoping
 - ▶ **#pragma omp private(...) shared(...) reduction(...) firstprivate(...) lastprivate(...)**
 - ▶ defines which variables are used together and which are used as copies in each thread
 - ▶ **shared is default value**

OpenMP Pragmas II

- Synchronization
 - ▶ **#pragma omp atomic, critical, ordered, barrier, flush**
 - ▶ essential for program correctness
- Parallel loops (work-sharing)
 - ▶ **#pragma omp parallel for**
 - ▶ following **for** is parallelized
 - ▶ type of distribution can be determined with **schedule** clause
 - ▶ e.g. **schedule(dynamic,4)**: each thread is assigned four loop iterations (1...4, 5...8) and new ones, as soon as a thread is ready
 - ▶ other variants are **static**, **guided** and **runtime**

OpenMP Run Time Environment I

Run time environment

- Processor count
 - ▶ `omp_get_num_procs()`
- Thread count
 - ▶ `omp_set_num_thread(int)`
 - ▶ `omp_get_num_thread()`
same as environment variable **OMP_NUM_THREADS**
 - ▶ `omp_get_thread_num()`

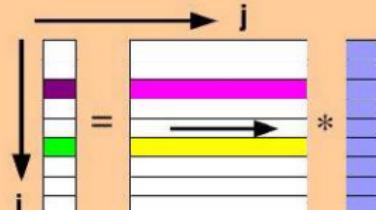
OpenMP Run Time Environment II

Run time environment

- Dynamic mode: Is in different blocks a different number of threads allowed?
 - ▶ `omp_set_dynamic()`, `omp_get_dynamic()`
 - ▶ equal to `OMP_DYNAMIC (TRUE / FALSE)`
- Nesting: Are in parallel regions new thread teams allowed? (nested threads)
 - ▶ `omp_set_nested()`, `omp_get_nested()`
 - ▶ equal to `OMP_NESTED (TRUE / FALSE)`

OpenMP in Practise: Matrix-Vector Product

```
#pragma omp parallel for default(None) \
    private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```



TID = 0

```
for (i=0,1,2,3,4)
    i = 0
    sum = b[i=0][j]*c[j]
    a[0] = sum
    i = 1
    sum = b[i=1][j]*c[j]
    a[1] = sum
```

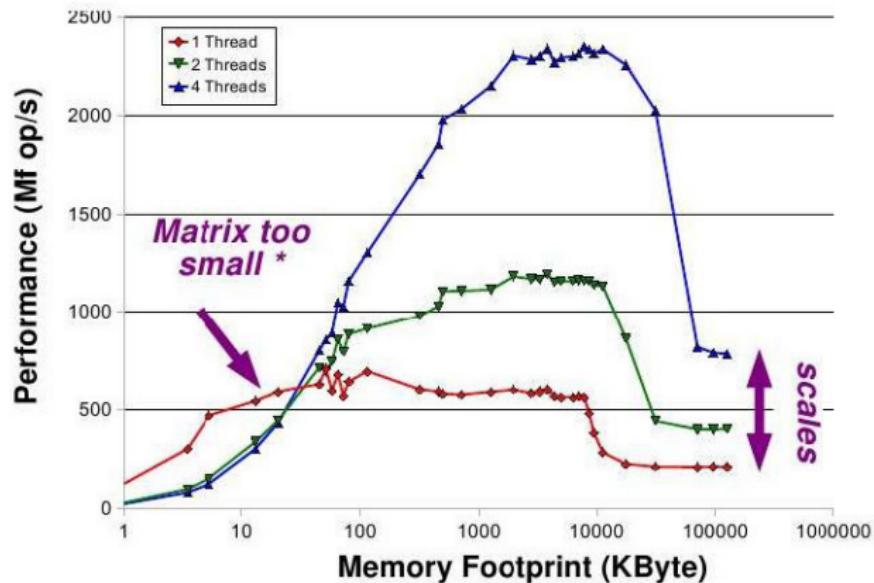
TID = 1

```
for (i=5,6,7,8,9)
    i = 5
    sum = b[i=5][j]*c[j]
    a[5] = sum
    i = 6
    sum = b[i=6][j]*c[j]
    a[6] = sum
```

... etc ...

openmp.org

OpenMP in Practise: Scaling behaviour in MFLOPs



**) With the IF-clause in OpenMP this performance degradation can be avoided*

OpenMP in Practise: IF-Clause

if (scalar expression)

- ✓ *Only execute in parallel if expression evaluates to true*
- ✓ *Otherwise, execute serially*

```
#pragma omp parallel if (n > some_threshold) \
    shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        x[i] += y[i];
} /*-- End of parallel region --*/
```

openmp.org

OpenMP in Practise: More Elaborate Example

```
#pragma omp parallel if (n>limit) default(none) \
    shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
    #pragma omp for nowait
    for (i=0; i<n; i++)
        z[i] = x[i] + y[i];
    #pragma omp for nowait
    for (i=0; i<n; i++)
        a[i] = b[i] + c[i];
    ...
    #pragma omp barrier
    scale = sum(a,0,n) + sum(z,0,n) + f;
}
/*-- End of parallel region --*/
```

parallel region

parallel loop
(work is distributed)

parallel loop
(work is distributed)

synchronization

Statement is executed by all threads

Statement is executed by all threads

OpenMP in Practise: OpenMP Summary

Directives	Runtime environment	Environment variables
<ul style="list-style-type: none">◆ <i>Parallel region</i>◆ <i>Worksharing constructs</i>◆ <i>Tasking</i>◆ <i>Synchronization</i>◆ <i>Data-sharing attributes</i>	<ul style="list-style-type: none">◆ <i>Number of threads</i>◆ <i>Thread ID</i>◆ <i>Dynamic thread adjustment</i>◆ <i>Nested parallelism</i>◆ <i>Schedule</i>◆ <i>Active levels</i>◆ <i>Thread limit</i>◆ <i>Nesting level</i>◆ <i>Ancestor thread</i>◆ <i>Team size</i>◆ <i>Wallclock timer</i>◆ <i>Locking</i>	<ul style="list-style-type: none">◆ <i>Number of threads</i>◆ <i>Scheduling type</i>◆ <i>Dynamic thread adjustment</i>◆ <i>Nested parallelism</i>◆ <i>Stacksize</i>◆ <i>Idle threads</i>◆ <i>Active levels</i>◆ <i>Thread limit</i>

OpenMP in Practise: Locking Mechanism

```
Program Locks
    ...
    Call omp_init_lock (LCK)

!$omp parallel shared(LCK)
    Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
        Call Do_Something_Else()
    End Do

    Call Do_Work()

    Call omp_unset_lock (LCK)

!$omp end parallel

    Call omp_destroy_lock (LCK)

    Stop
End
```

Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

OpenMP in Practise: Scheduling I

```
schedule ( static | dynamic | guided | auto [, chunk] )
schedule (runtime)
```

static [, chunk]

- ✓ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✓ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*
 - *Details are implementation defined*
- ✓ *Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region*

openmp.org

OpenMP in Practise: Scheduling II

dynamic [, chunk]

- ✓ *Fixed portions of work; size is controlled by the value of chunk*
- ✓ *When a thread finishes, it starts on the next portion of work*

guided [, chunk]

- ✓ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

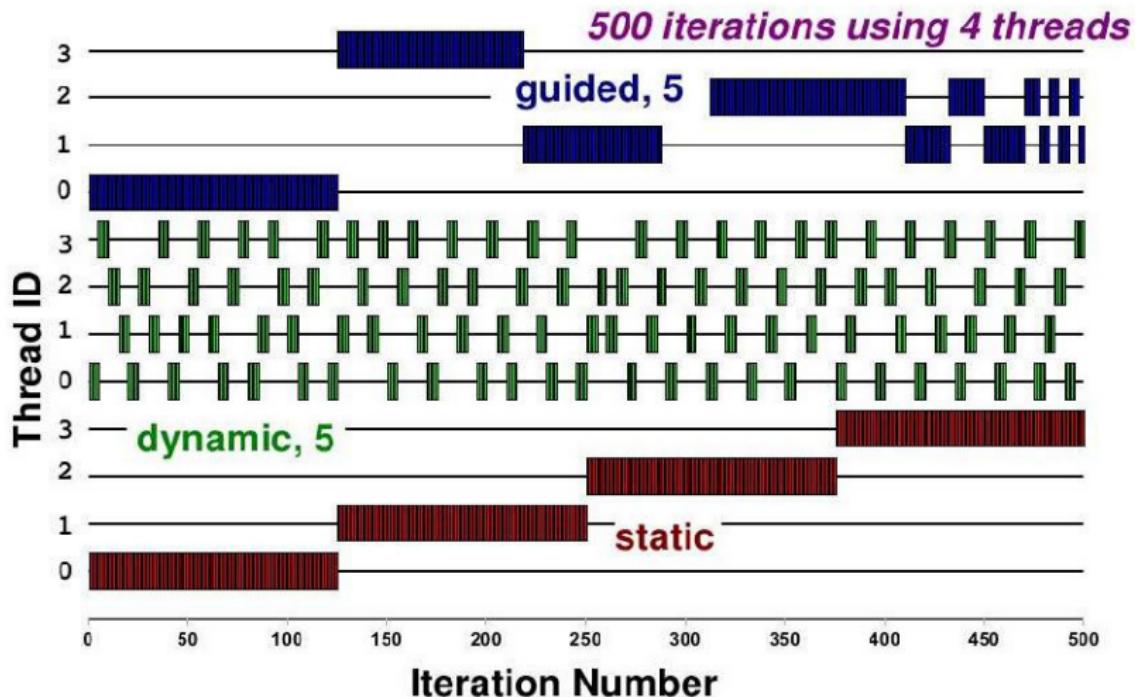
auto

- ✓ *The compiler (or runtime system) decides what is best to use; choice could be implementation dependent*

runtime

- ✓ *Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE*

OpenMP in Practise: Scheduling III



Shared Memory Programming Models II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Parallel Programming Models II

Communication using shared memory

- Barrier – synchronization of all processes
- Semaphores
- Philosophers problem

Global Synchronization

- *Barrier*: All processes shall wait on each other until all have arrived
- Barriers are often executed repeatedly:

```
while (1) {  
    a calculation;  
    Barrier;  
}
```

- Since the calculation is load balanced, all processes arrive simultaneously at the barrier
- First idea: Count all arriving processes

Global Synchronization

Program (First proposal of a barrier)

```
parallel barrier-1
{
    const int P=8;      int count=0;      int release=0;

    process  $\sqcap$  [int  $p \in \{0, \dots, P - 1\}$ ]
    {
        while (1)
        {
            calculation;
            CSenter;                                // entry
            if (count==0) release=0;                  // reset
            count=count+1;                          // increment counter
            CSexit;                                 // exit
            if (count==P) {
                count=0;                            // last resets counter
                release=1;                          // and frees
            }
            else while (release==0);               // waiting
        }
    }
},
```

Barrier with Sense Reversal

Wait reversible for *release==1* and *release==0*

Program (Barrier with direction reversal)

```
parallel sense-reversing-barrier
{
    const int P=8;      int count=0;      int release=0;

    process Π [int p ∈ {0, ..., P - 1}]
    {
        int local_sense = release;
        while (1)
        {
            calculation;
            local_sense = 1-local_sense;          // change direction
            CSenter;                            // entry
            count=count+1;                      // increment counter
            CSexit;                             // exit
            if (count==P) {
                count=0;                        // last resets
                release=local_sense;           // and frees
            } else
                while (release≠local_sense);
        }
    }
}
```

Complexity is $O(P^2)$ since all P processes have to pass through a critical section at a time. Is there a better approach?

Hierarchical Barrier: Variant 1

In the barrier with counter all P processes have to pass through a critical section. This necessitates $O(P^2)$ memory accesses. We now develop a solution with $O(P \log P)$ accesses.

We start with *two* processes and consider the following program segment:

```
int arrived=0, continue=0;
```

Π_0 :

```
while ( $\neg$ arrived) ;  
arrived=0;  
continue=1;
```

Π_1 :

```
arrived=1;
```

```
while ( $\neg$ continue) ;  
continue=0;
```

We use two synchronization variables, so called *flags*

Hierarchical Barrier: Variant 1

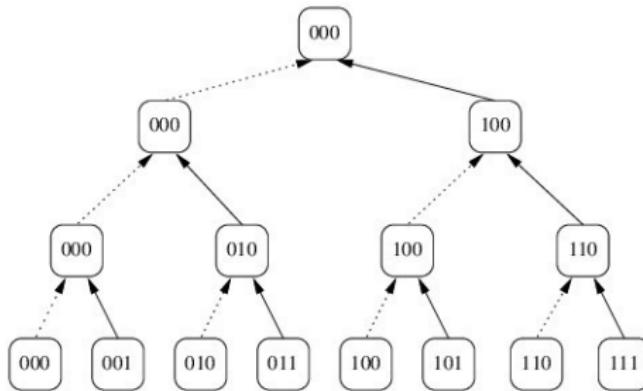
When using flags the following rules have to be met:

- ① The process, that waits for a flag, also resets it.
- ② A flag may first be newly set, if it has been safely reset.

Both rules are respected by our solution.

The solution assumes sequential consistency of the memory!

We now apply this idea in a hierarchical way:



Hierarchical Barrier: Variant 1

Program (Barrier with tree)

```
parallel tree-barrier
{
    const int d = 4, P = 2d; int arrived[P]={0[P]}, continue[P]={0[P]};

    process  $\sqcap$  [int p  $\in$  {0, ..., P - 1}]
    {
        int i, r, m, k;
        while (1) {
            calculation;
            for (i = 0; i < d; i++) { // upward
                r = p &  $\left[ \sim \left( \sum_{k=0}^i 2^k \right) \right]$ ; // reset bits 0 to i
                m = r |  $2^i$ ; // set bit i
                if (p == m) arrived[m]=1;
                if (p == r) {
                    while( $\neg$ arrived[m]); // wait
                    arrived[m]=0;
                }
            }
        } // process 0 knows that all are there
        ...
    }
}
```

Hierarchical Barrier: Variant 1

Program (Barrier with tree cont.)

parallel *tree-barrier cont.*

{

```
    ...
    for ( $i = d - 1; i \geq 0; i--$ ) {           // downward
         $r = p \& \left[ \sim \left( \sum_{k=0}^i 2^k \right) \right];$  // reset bits 0 to i
         $m = r | 2^i;$ 
        if ( $p == m$ ) {
            while ( $\neg continue[m]$ );
             $continue[m] = 0;$ 
        }
        if ( $p == r$ )  $continue[m] = 1;$ 
    }
}
}
```

Caution: Flag variables should be stored in different cache lines, that accesses do not hinder each other!

Hierarchical Barrier: Variant 2

This variant presents a symmetric solution of the barrier with *recursive doubling*.

We consider at first again the barrier for two processes Π_i and Π_j :

Π_i :

```
while (arrived[i]) ;  
arrived[i]=1;  
while ( $\neg$ arrived[j]) ;  
arrived[j]=0;
```

Π_j :

```
while (arrived[j]) ;  
arrived[j]=1;  
while ( $\neg$ arrived[i]) ;  
arrived[i]=0;
```

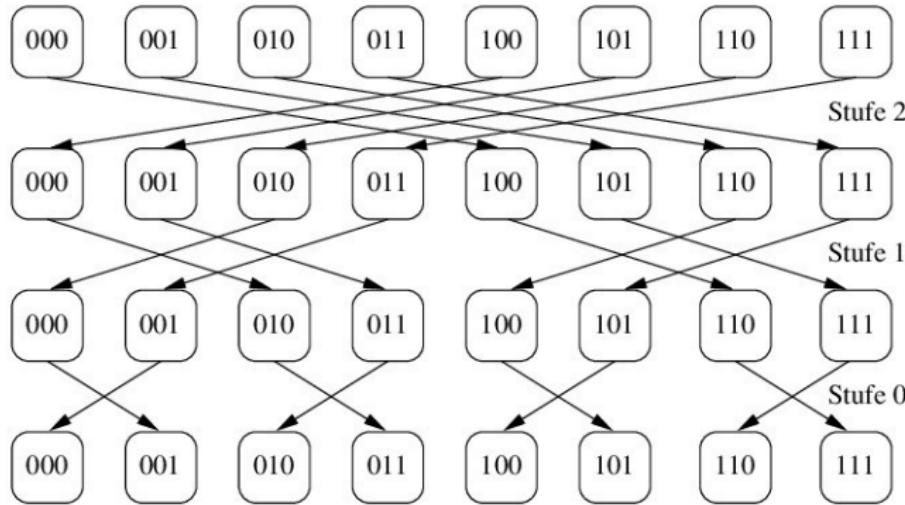
As prerequisite for the general solution the flags are organized as arrays, in the beginning all flags are 0.

Sequence in words:

- Line 2: Each sets *its* flag to 1
- Line 3: Each waits onto the flag of the other
- Line 4: Each resets the flag of the *other*
- Line 1: Because of rule 2 from above wait until the flag is reset
- Now we use this ideas in a recursive manner!

Hierarchical Barrier: Variant 2

Recursive doubling uses the following communication structure:



- No idle processors
- Each step is a two way communication

Hierarchical Barrier: Variant 2

Program (Barrier with recursive doubling)

```
parallel recursive-doubling-barrier
{
    const int d = 4,      P = 2d; int arrived[d][P]={0[P · d]};

    process □ [int p ∈ {0, ..., P - 1}]
    {
        int i, q;
        while (1) {
            calculation;
            for (i = 0; i < d; i++)                      // all steps
            {
                q = p ⊕ 2i;                            // reverse bit i
                while (arrived[i][p]) ;
                arrived[i][p]=1;
                while (¬arrived[i][q]) ;
                arrived[i][q]=0;
            }
        }
    }
}
```

Semaphore

A semaphore is an abstraction of a synchronisation variable, that enables the elegant solution of multiple synchronisation problems

Up-to-now all programs have used *active waiting*. This is very inefficient under quasi-parallel processing of multiple processes on one processor (multitasking). The semaphore enables to switch processes into an idle state.

We understand a semaphore as abstract data type: Data structure with operations, that fulfill particular properties:

A semaphore S has a non-negative integer value $\text{value}(S)$, that is assigned during creation of the semaphore with the value *init*.

For a semaphore S two operations $\mathbf{P}(S)$ and $\mathbf{V}(S)$ are defined with:

- $\mathbf{P}(S)$ decrements the value of S by one if $\text{value}(S) > 0$, otherwise the process *blocks* as long as another process executes a \mathbf{V} operation on S .
- $\mathbf{V}(S)$ frees another process from a \mathbf{P} operation if one is waiting (are several waiting one is selected), otherwise the value of S is incremented by one. \mathbf{V} operations never block!

Semaphore

Is the number of *successfully finished* **P** operations n_P and the one of **V** operations n_V , then for the value of the semaphore applies always:

$$\text{value}(S) = n_V + \text{init} - n_P \geq 0$$

or equivalent $n_P \leq n_V + \text{init}$.

The value of a semaphore is *not* visible from the outside. It shows only by the executability of the **P** operation.

The increment resp. decrement of a semaphore is performed in an atomic way, multiple processes can also perform **P/V** operations concurrently.

Semaphores, that can take a value larger than one, are called *general semaphores*.

Semaphores, that only have values $\{0, 1\}$, are called *binary semaphores*.

Notation:

Semaphore $S=1$;

Semaphore $\text{forks}[5] = \{1 [5]\}$;

Mutual Exclusion with Semaphore

We now present in which way all already treated synchronisation problems can be solved with semaphore variables. The first application is dedicated to mutual exclusion by usage of a single binary semaphore:

Program (Mutual exclusion with semaphore)

```
parallel cs-semaphore
{
    const int P=8;
    Semaphore mutex=1;
    process  $\sqcap$  [int  $i \in \{0, \dots, P - 1\}$ ]
    {
        while (1)
        {
            P(mutex);
            critical section;
            V(mutex);
            uncritical section;
        }
    }
}
```

Mutual Exclusion with Semaphore

By multitasking processes can be switched to the idle state (waiting).

Fairness is easy to integrate into the wake-up mechanism (FCFS).

Memory consistency model can be respected by the implementation,
programs remains portable (e. g. Pthreads)

Barrier with Semaphore

- Each process has to be delayed until the other(s) arrive at the barrier.
- The barrier has to be reusable, since it is usually executed several times.

Program (Barrier with semaphore for two processes)

```
parallel barrier-2-semaphore
{
    Semaphore b1=0, b2=0;
    process Π1
    {
        while (1) {
            calculation;
            V(b1);
            P(b2);
        }
    }
    process Π2
    {
        while (1) {
            calculation;
            V(b2);
            P(b1);
        }
    }
}
```

Barrier with Semaphore

After unrolling of the loop, the code looks as follows:

$\Pi_1:$	$\Pi_2:$
calculation 1;	calculation 1;
$V(b1);$	$V(b2);$
$P(b2);$	$P(b1);$
calculation 2;	calculation 2;
$V(b1);$	$V(b2);$
$P(b2);$	$P(b1);$
calculation 3;	calculation 3;
$V(b1);$	$V(b2);$
$P(b2);$	$P(b1);$
...	...

Assume process Π_1 works in calculation phase i , thus it has executed $P(b2)$ $i - 1$ -times. Assume further Π_2 works in calculation phase $j < i$, therefore it has executed $V(b2)$ $j - 1$ -times. Then holds

$$n_P(b2) = i - 1 > j - 1 = n_V(b2).$$

On the other hand the semaphore rules assure, that

$$n_P(b2) \leq n_V(b2) + 0.$$

This is a contradiction and it can not apply $j < i$. The argument is symmetric and applies also when the processor numbers are exchanged.

Producer/Consumer $m/n/1$

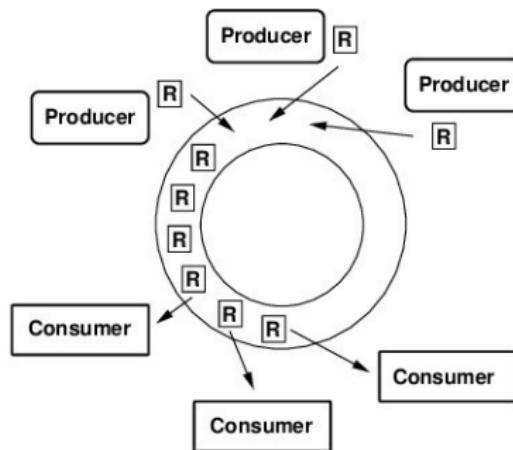
m producers, n consumers, 1 buffer location,

Producer has to block if the buffer location is occupied.

Consumer has to block if no request is stored.

We use two semaphores:

- *empty*: counts number of *free* buffer locations
- *full*: counts number of *occupied* locations (requests)



Produce/Consumer $m/n/1$

Program (m producer, n consumer, 1 buffer location)

```
parallel prod-con-nm1
{
    const int m = 3, n = 5;
    Semaphore empty=1;                                // free buffer location
    Semaphore full=0;                                 // available request
    T buf;                                            // the buffer
    process P [int i ∈ {0, ..., m - 1}] {
        while (1) {
            Generate request t;
            P(empty);                                // Is buffer free?
            buf = t;                                  // store request
            V(full);                                 // request available
        }
    }
    process C [int j ∈ {0, ..., n - 1}] {
        while (1) {
            P(full);                                // Is request available?
            t = buf;                                // remove request
            V(empty);                               // buffer is empty
            Process request t;
        }
    }
}
```

Shared binary semaphore (*split binary semaphore*):

$$0 \leq empty + full \leq 1 \quad (\text{invariant})$$

Producer/Consumer 1/1/k

1 producer, 1 consumer, k buffer locations,

Buffer is array of length k of type T . Insertion and deletion works with

$$\begin{aligned}buf[front] &= t; \quad front = (front + 1) \bmod k; \\t &= buf[rear]; \quad rear = (rear + 1) \bmod k;\end{aligned}$$

Semaphore as above, only initialized with k !

Program (1 producer, 1 consumer, k buffer locations)

```
parallel prod-con-11k
{
    const int k = 20;
    Semaphore empty=k;                                // counts free buffer locs
    Semaphore full=0;                                 // count available requests
    T buf[k];                                         // the buffer
    int front=0;                                       // newest request
    int rear=0;                                        // oldest request
}
```

Producer/Consumer 1/1/k

Program (1 producer, 1 consumer, k buffer locations)

parallel prod-con-11k

{

```
  process P {
    while (1) {
        Generate request t;
        P(empty);           // Is buffer free?
        buf[front] = t;       // store request
        front = (front+1) mod k; // next free location
        V(full);           // request available
    }
  }

  process C {
    while (1) {
        P(full);           // Is request there?
        t = buf[rear];       // remove request
        rear = (rear+1) mod k; // next request
        V(empty);          // buffer is free
        Process request t;
    }
}
```

,

Producer/Consumer $m/n/k$

m producers, n consumers, k buffer locations,

We only have to ensure, that producers among each other and consumers cannot manipulate the buffer at the same time.

Use two additional binary semaphores $mutexP$ und $mutexC$

Program (m producer, n consumer, k buffer locations)

```
parallel prod-con-mnk
{
    const int k = 20, m = 3, n = 6;
    Semaphore empty=k;                                // count free buffer locations
    Semaphore full=0;                                 // count available requests
    T buf[k];                                         // the buffer
    int front=0;                                       // newest request
    int rear=0;                                        // oldest request
    Semaphore mutexP=1;                               // access of producers
    Semaphore mutexC=1;                               // access of consumersr
}
```

Producer/Consumer $m/n/k$

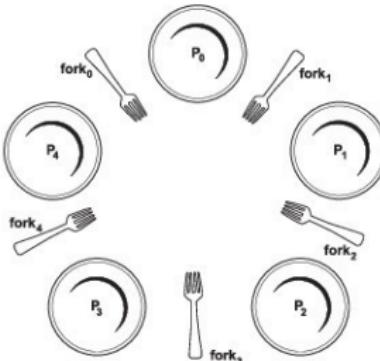
Program (m producer, n consumer, k buffer locations)

parallel process

```
{  
    P [int i ∈ {0, ..., m - 1}] {  
        while (1) {  
            Generate request t;  
            P(empty);                                // Is buffer free?  
            P(mutexP);                               // manipulate buffer  
            buf[front] = t;  
            front = (front+1) mod k;  
            V(mutexP);                               // store request  
            V(full);                                 // next free position  
        }  
    }  
    process C [int j ∈ {0, ..., n - 1}] {  
        while (1) {  
            P(full);                                // Is request there?  
            P(mutexC);                               // manipulate buffer  
            t = buf[rear];  
            rear = (rear+1) mod k;  
            V(mutexC);                               // remove request  
            V(empty);                                // next request  
            Process request t;  
        }  
    }  
}
```

Dining Philosophers

Complex synchronisation task: A process necessitates exclusive access onto several resources to perform a specific task.
→ overlapping critical sections.



Five philosophers sit at a round table. The exercise of each philosopher consists out of interchanging phases of thinking and eating. In between two of the philosophers a fork is positioned and in the center of the table a mountain Spaghetti is located. To eat a philosopher needs two forks – the one laying left and right next to him.

Dining Philosophers

The problem:

Write a parallel program, with one process per philosopher, that

- enables a maximal count of philosophers to eat and
- that avoids a deadlock.

Skeletal structure of a philosopher:

```
while (1)
{
    think;
    take forks;
    eat;
    lay back forks;
}
```

Naive Philosophers

Program (Naive solution of the philosophers problem)

```
parallel philosophers-1
{
    const int P = 5;                                // number of philosophers
    Semaphore forks[P] = { 1 [P] };                 // forks

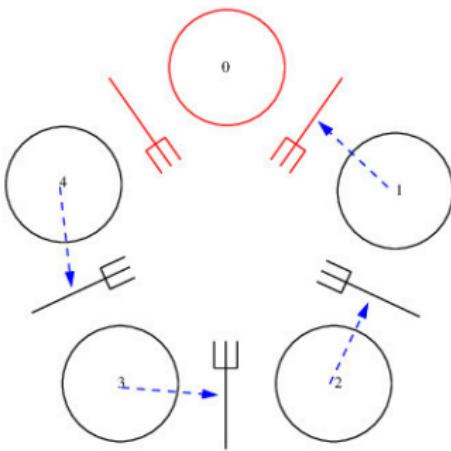
    process Philosopher [int p ∈ {0, ..., P - 1}] {
        while (1) {
            Thinking;
            P(fork[p]);                            // left fork
            P(fork[(p + 1) mod P]);                // right fork
            Eating;
            V(fork[p]);                            // left fork
            V(fork[(p + 1) mod P]);                // right fork
        }
    }
}
```

Naive Philosophers

Philosophers are deadlocked, if all take at first the left fork!

Simple solution of the deadlock problem: Avoid cyclic dependencies,
e. g. philosopher 0 takes his forks in a different sequence right then left.

This solution allows eventually not maximal concurrency:



Clever Philosophers

Take forks only, when *both* are available

Critical section: only one can manipulate the forks

Three states of a philosopher: thinking, hungry, eating

Program (Solution of philosophers problem)

parallel *philosophers*-2

{

```
  const int P = 5;                                // count philosophers
  const int think=0, hungry=1, eat=2;
  Semaphore mutex=1;
  Semaphore s[P] = { 0 [P] };                    // eating philosopher
  int state[P] = { think [P] };                  // state
```

}

Clever Philosophers

Program (Solution of philosophers problem)

parallel process

```
{  
    Philosopher [int  $p \in \{0, \dots, P - 1\}$ ] {  
        void test (int i) {  
            int l=(i + P - 1) mod P, r=(i + 1) mod P;  
            if (state[i]==hungry  $\wedge$  state[l]≠eat  $\wedge$  state[r]≠eat)  
            {  
                state[i] = eat;  
                V(s[i]);  
            }  
        }  
  
        while (1) {  
            Thinking;  
            P(mutex);  
            state[p] = hungry;  
            test(p);  
            V(mutex);  
            P(s[p]);  
            Eating;  
            P(mutex);  
            state[p] = think;  
            test((p + P - 1) mod P);  
            test((p + 1) mod P);  
            V(mutex);  
        }  
    }  
}
```

Shared Memory Programming Models III

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Processes and Threads

A Unix process has

- IDs (process, user, group)
- Environment variables
- Directory
- Program code
- Register, stack, heap
- File descriptors, signals
- Message queues, pipes, shared memory segments
- Shared libraries

Each process owns its individual address space

Threads exist within a single process

Threads share an address space

A thread consists of

- ID
- Stack pointer
- Registers
- Scheduling properties
- Signals

Creation and switching times are shorter

„Parallel function“

PThreads

- Each manufacturer had an own implementation of threads or „light weight processes“
- 1995: IEEE POSIX 1003.1c Standard (there are several „drafts“)
- Standard document is liable to pay costs
- Defines threads in a portable way
- Consists of C data types and functions
- Header file `pthread.h`
- Library name is not normed. In Linux `-lpthread`
- Compilation in Linux: `gcc <file> -lpthread`

PThreads Overview

There are 3 functional groups

All names start with `pthread_`

- `pthread_`
Thread management and other routines
- `pthread_attr_`
Thread attribute objects
- `pthread_mutex_`
All that has to do with mutex variables
- `pthread_mutex_attr_`
Attributes for mutex variables
- `pthread_cond_`
Condition variables
- `pthread_cond_attr_`
Attributes for condition variables

Creation of Threads

- `pthread_t` : Data type for a thread.
- Opaque type: Data type is defined in the library and is processed by its functions. Contents is implementation dependent.
- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)` : Starts the function `start_routine` as thread.
 - ▶ `thread` : Pointer onto a `pthread_t` structure. Serves for identification of a thread.
 - ▶ `attr` : Thread attributes are explained below. Default is `NULL`.
 - ▶ `start_routine` : Pointer onto a function of type `void* func (void*)`;
 - ▶ `arg` : `void*` pointer that is passed as function argument.
 - ▶ Return value that is larger than zero indicates an error.
- Threads can start further threads, maximal count of threads is implementation dependent

Termination of Threads

- There are the following possibilities to terminate a thread:
 - ▶ The thread finishes its `start_routine()`
 - ▶ The thread calls `pthread_exit()`
 - ▶ The thread is terminated by another thread via `pthread_cancel()`
 - ▶ The process is terminated by `exit()` or the end of the `main()` function
- `pthread_exit(void* status)`
 - ▶ Finishes the calling thread. Pointer is stored and can be queried with `pthread_join` (see below) (Return of results).
 - ▶ If `main()` calls this routine existing threads continue and the process is not terminated.
 - ▶ Existing files, that are opened, are not closed!

Waiting for Threads

- Peer model: Several equal threads perform a collective task. Program is terminated if all threads are finished
- Requires waiting of a thread until all others are finished
- This is a kind of synchronisation
- `int pthread_join(pthread_t thread, void **status);`
 - ▶ Waits until the specified thread terminates itself
 - ▶ The thread can return via `pthread_exit()` a `void*` pointer
 - ▶ If the status parameter is chosen as `NULL`, the return value is obsolete

Thread Management Example

```
#include <pthread.h>      /* for threads      */

void* prod (int *i) { /* Producer thread */
    int count=0;
    while (count<100000) count++;
}

void* con (int *j) { /* Consumer thread */
    int count=0;
    while (count<1000000) count++;
}

int main (int argc, char *argv[]) { /* main program */
    pthread_t thread_p, thread_c; int i,j;

    i = 1; pthread_create(&thread_p, NULL, (void*(*)(void*)) prod, (void *) &i);
    j = 1; pthread_create(&thread_c, NULL, (void*(*)(void*)) con, (void *) &j);

    pthread_join(thread_p, NULL); pthread_join(thread_c, NULL);
    return(0);
}
```

Passing of Arguments

- Passing of multiple arguments requires the definition of an individual data type:

```
struct argtype {int rank; int a,b; double c;};
struct argtype args[P];
pthread_t threads[P];

for (i=0; i<P; i++) {
    args[i].rank=i; args[i].a=...
    pthread_create(threads+i,NULL,(void* (*) (void*)) prod,(void *)args+i);
}
```

- The following example contains two errors:

```
pthread_t threads[P];
for (i=0; i<P; i++) {
    pthread_create(threads+i,NULL,(void* (*) (void*)) prod,&i);
}
```

- ▶ Contents of `i` is eventually changed before the thread reads it
- ▶ If `i` is a stack variable it exists eventually no more

Thread Identifiers

- `pthread_t pthread_self(void);`
Returns the own thread-ID
- `int pthread_equal(pthread_t t1, pthread_t t2);`
Returns true (value>0) if the two IDs are identical
- Concept of an „opaque data type“

Join/Detach

- A thread within state PTHREAD_CREATE_JOINABLE releases its resources only, if `pthread_join` has been executed.
- A thread in state PTHREAD_CREATE_DETACHED releases its resources as soon as it is terminated. In this case `pthread_join` is not allowed.
- Default is PTHREAD_CREATE_JOINABLE, but that is not implemented in all libraries.
- Therefore better:

```
pthread_attr_t attr;  
pthread_attr_init(&attr);  
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);  
int rc = pthread_create(&t,&attr, (void* (*) (void*)) func, NULL);  
....  
pthread_join(&t, NULL);  
pthread_attr_destroy(&attr);
```

- Provides example for application of attributes

Mutex Variables

- Mutex variables realize mutual exclusion within PThreads
- Creation and initialisation of a mutex variable

```
pthread_mutex_t mutex;  
pthread_mutex_init (&mutex, NULL);
```

Mutex variable is in state free

- Try to enter the critical section (blocking):

```
pthread_mutex_lock (&mutex);
```

- Leave critical section

```
pthread_mutex_unlock (&mutex);
```

- Release resource of the mutex variable

```
pthread_mutex_destroy (&mutex);
```

Condition Variables

- Condition variables enable *inactive* waiting of a thread until a certain condition has arrived.
- Simplest example: Flag variables (see example below)
- To a condition synchronisation belong *three* things:
 - ▶ A variable of type `pthread_cond_t`, that realizes inactive waiting.
 - ▶ A variable of type `pthread_mutex_t`, that realizes mutual exclusion during condition change.
 - ▶ A global variable, which value enables the calculation of the condition

Condition Variables: Creation/Deletion

- `int pthread_cond_init(pthread_cond_t *cond,
pthread_condattr_t *attr);`
initializes a condition variable
In the simplest case: `pthread_cond_init(&cond, NULL)`
- `int pthread_cond_destroy(pthread_cond_t *cond);`
the resources of a condition variable is released

Condition Variables: Wait

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
blocks the calling thread until for the condition variable the function `pthread_signal()` is called
- When calling the `pthread_wait()` the thread has to be the owner of the lock
- `pthread_wait()` leaves the lock and waits for the signal in an atomic way
- After returning from `pthread_wait()` the thread is again the owner of the lock
- After return the condition has not to be true in any case
- With a single condition variable one should only use exactly one lock

Condition Variables: Signal

- `int pthread_cond_signal(pthread_cond_t *cond);`
Awakes a thread that has executed a `pthread_wait()` onto a condition variable. If no one waits the function has no effect.
- When calling the thread should be owner of the associated lock.
- After the call the lock should be released. First the release of the lock allows the waiting thread to return from `pthread_wait()` function.
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
awakes *all* threads that have executed a `pthread_wait()` on the condition variable. These then apply for the lock.

Condition Variables: Ping-Pong Example

```
#include<stdio.h>
#include<pthread.h>      /* for threads      */

int arrived_flag=0,continue_flag=0;
pthread_mutex_t arrived_mutex, continue_mutex;
pthread_cond_t arrived_cond, continue_cond;

pthread_attr_t attr;

int main (int argc, char *argv[])
{
    pthread_t thread_p, thread_c;

    pthread_mutex_init(&arrived_mutex,NULL);
    pthread_cond_init(&arrived_cond,NULL);
    pthread_mutex_init(&continue_mutex,NULL);
    pthread_cond_init(&continue_cond,NULL);
```

Example cont. I

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
                           PTHREAD_CREATE_JOINABLE);

pthread_create(&thread_p, &attr,
               (void* (*)(void*)) prod, NULL);
pthread_create(&thread_c, &attr,
               (void* (*)(void*)) con , NULL);

pthread_join(thread_p, NULL);
pthread_join(thread_c, NULL);

pthread_attr_destroy(&attr);

pthread_cond_destroy(&arrived_cond);
pthread_mutex_destroy(&arrived_mutex);
pthread_cond_destroy(&continue_cond);
pthread_mutex_destroy(&continue_mutex);

return(0);
```

Example cont. II

```
void prod (void* p) /* Producer thread */
{
    int i;
    for (i=0; i<100; i++) {
        printf("ping\n");

        pthread_mutex_lock(&arrived_mutex);
        arrived_flag = 1;
        pthread_cond_signal(&arrived_cond);
        pthread_mutex_unlock(&arrived_mutex);

        pthread_mutex_lock(&continue_mutex);
        while (continue_flag==0)
            pthread_cond_wait(&continue_cond, &continue_mutex);
        continue_flag = 0;
        pthread_mutex_unlock(&continue_mutex);
    }
}
```

Example cont. III

```
void con (void* p) /* Consumer thread */
{
    int i;
    for (i=0; i<100; i++) {
        pthread_mutex_lock(&arrived_mutex);
        while (arrived_flag==0)
            pthread_cond_wait(&arrived_cond, &arrived_mutex);
        arrived_flag = 0;
        pthread_mutex_unlock(&arrived_mutex);

        printf("pong\n");

        pthread_mutex_lock(&continue_mutex);
        continue_flag = 1;
        pthread_cond_signal(&continue_cond);
        pthread_mutex_unlock(&continue_mutex);
    }
}
```

Thread Safety

- Hereby is understood whether a function/library can be used by multiple threads at the same time.
- A function is *reentrant* if it may be called by several threads synchronously.
- A function, that does not use a global variable, is reentrant
- The runtime system has to use shared resources (e.g. the stack) under mutual exclusion
- The GNU C compiler has to be configured for compilation with an appropriate thread model. With `gcc -v` you can see the type of thread model.
- STL: Allocation is thread save, access of multiple threads onto a single container has to be protected by the user.

Threads and OO

- Obviously are PThreads relatively impractical to code.
- Mutexes, conditional variables, flags and semaphores should be realized in an object-oriented way. Complicated `init/destroy` calls can be hidden in constructors/destructors.
- Threads are transformed into Active Objects.
- An active object „is executed“ independent of other objects.

Active Objects

```
class ActiveObject
{
public:
    //! constructor
    ActiveObject ();

    //! destructor waits for thread to complete
    ~ActiveObject ();

    //! action to be defined by derived class
    virtual void action () = 0;

protected:
    //! use this method as last call in constructor of derived class
    void start ();

    //! use this method as first call in destructor of derived class
    void stop ();

private:
    ...
};
```

Active Objects cont. I

```
#include<iostream>
#include"threadtools.hh"

Flag arrived_flag,continue_flag;

int main (int argc, char *argv[])
{
    Producer prod; // start prod as active object
    Consumer con; // start con as active object

    return(0);
} // wait until prod and con are finished
```

Active Objects cont. II

```
class Producer : public ActiveObject
{
public:
    // constructor takes any arguments the thread might need
    Producer () {
        this->start();
    }

    // execute action
    virtual void action () {
        for (int i=0; i<100; i++) {
            std::cout << "ping" << std::endl;
            arrived_flag.signal();
            continue_flag.wait();
        }
    }

    // destructor waits for end of action
    ~Producer () {
        this->stop();
    }
};
```

Active Objects cont. III

```
class Consumer : public ActiveObject
{
public:
    // constructor takes any arguments the thread might need
    Consumer () {
        this->start();
    }

    // execute action
    virtual void action () {
        for (int i=0; i<100; i++) {
            arrived_flag.wait();
            std::cout << "pong" << std::endl;
            continue_flag.signal();
        }
    }

    // destructor waits for end of action
    ~Consumer () {
        this->stop();
    }
};
```

Distributed-Memory Programming Models I

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Distributed-Memory Programming Models I

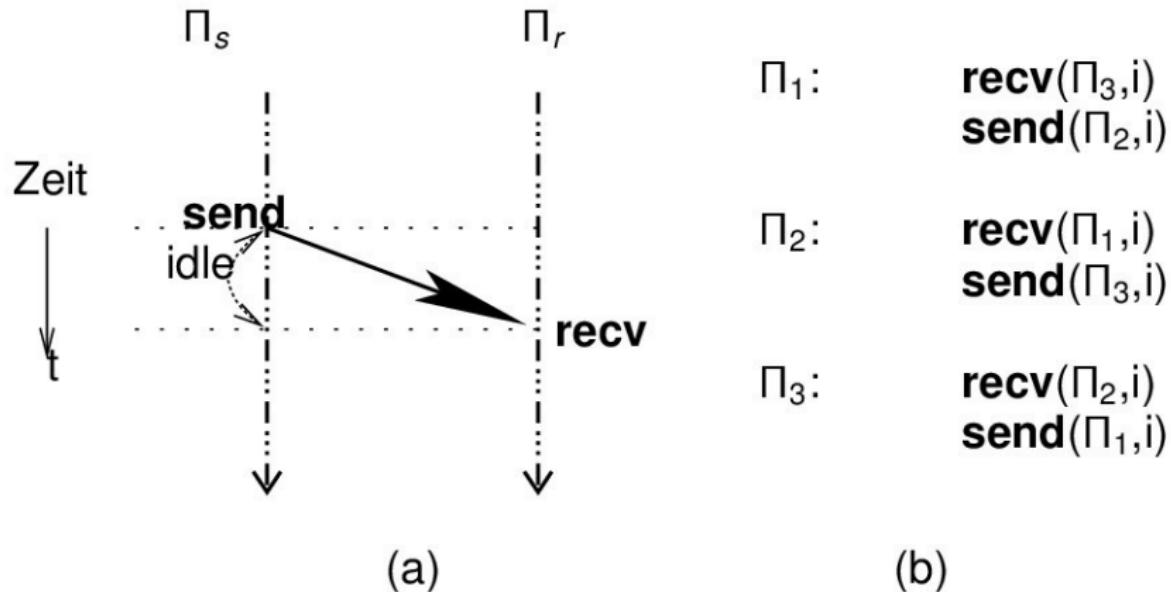
Communication via message passing

- Synchronous message passing
- Asynchronous message passing
- Global communication with
 - ▶ Store-and-Forward or
 - ▶ Cut-Through routing
- Global communication using different topologies
 - ▶ Ring
 - ▶ Array (2D / 3D)
 - ▶ Hypercube

Synchronous Message Passing I

- For the passing of messages we need at least two functions:
 - ▶ **send**: Transmits a memory area from the address space of the source process into the network with specification of the receiver.
 - ▶ **recv**: Receives a memory area from the network and writes it to the address space of the destination process.
- We distinguish:
 - ▶ Point in time at which the communication function is finished.
 - ▶ Point in time at which the communication has really taken place.
- At synchronous communication these points in time are identical,
 - ▶ **send** blocks until the receiver has accepted the message.
 - ▶ **recv** blocks until the message has arrived.
- Syntax in our programming language:
 - ▶ **send**(*dest-process, expr₁, ..., expr_n*)
 - ▶ **recv**(*src-process, var₁, ..., var_n*)

Synchronous Message Passing II



- (a) Synchronisation of two processes by a pair of **send/recv** ops
- (b) Example for a deadlock

Synchronous Message Passing III

There are a series of implementation possibilities.

- Senderinitiated, *three-way handshake*:
 - ▶ Source Q sends *ready-to-send* to target Z.
 - ▶ Target sends *ready-to-receive* if **recv** has been executed.
 - ▶ Source transmits message (variable length, single copy).
- Receiver initiated, *two-phase protocol*:
 - ▶ Z sends *ready-to-receive* to Q if **recv** has been executed.
 - ▶ Q transmits message (variable length, single copy).
- Buffered Send
 - ▶ Q transmits message at once, target has eventually to buffer it.
 - ▶ Here arises the problem of finite memory space!

Synchronous Message Passing IV

- Synchronous **send/recv** is not enough to solve all communication tasks!
- Example: In the producer-consumer-problem the buffer is realized as an individual process. In this case the process cannot know with which of the producers or consumers it has to communicate next. In consequence a blocking **send** can result in a deadlock.
- Solution: Introduction of additional *guard functions*, that check whether a **send** or **recv** would result in a deadlock:
 - ▶ **int sprobe(dest – process)**
 - ▶ **int rprobe(src – process).**

sprobe returns 1 if the receiver process is ready to receive, this means a **send** will not block:

- ▶ **if (sprobe(Π_d)) send(Π_d, \dots);**

Analogous for **rprobe**.

- Guard functions never block!

Synchronous Message Passing V

- Just one of both functions is needed.
 - ▶ **rprobe** is easy to integrate into the sender-initiated protocol.
 - ▶ **sprobe** is easy to integrate into the receiver-initiated protocol.
- An instruction with similar effect as **rprobe** is:
 - ▶ **recv_any**(*who*,*var*₁,...,*var*_{*n*}).

It allows the receiving from an *arbitrary* process, which ID is stored in the variable *who*.
- **recv_any** is implemented simplest with sender-initiated protocol.

Asynchronous Message Passing I

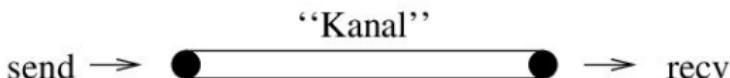
- Instructions for asynchronous message passing:
 - ▶ **asend**(*dest – process, expr₁, ..., expr_n*)
 - ▶ **arecv**(*src – process, var₁, ..., var_n*)
 - Here the return of the communication function does *not* indicate, that the communication has actually taken place. This has to be queried with additional functions.
 - We imagine, that a request is passed to the system to execute the corresponding communication, as soon as it is possible. The calculating process can meanwhile do other things (*communication hiding*).
 - Syntax:
 - ▶ **msgid asend**(*dest – process, var₁, ..., var_n*)
 - ▶ **msgid arecv**(*src – process, var₁, ..., var_n*)
- these do never block! **msgid** is a bill for the communication request.

Asynchronous Message Passing II

- Caution: The variable var_1, \dots, var_n may not be modified anymore when the communication instruction has been initiated!
- This means that the program has to manage the memory space for the communication variable by itself. Alternative would be the buffered send, which is connected with subtleties and need of double-copying.
- Finally one has to test whether the communication has already taken place (this means the request is processed):
 - ▶ `int success(msgid m)`
- Thereafter the communication variables may be modified, the bill is then invalidated.

Synchronous/Asynchronous Message Passing

- Synchronous and asynchronous operations may be mixed. This is specified in the MPI standard.
- Up-to-now all operations have been *without connection*.
- Alternatively there exist channel oriented communication operations (or virtual channels):



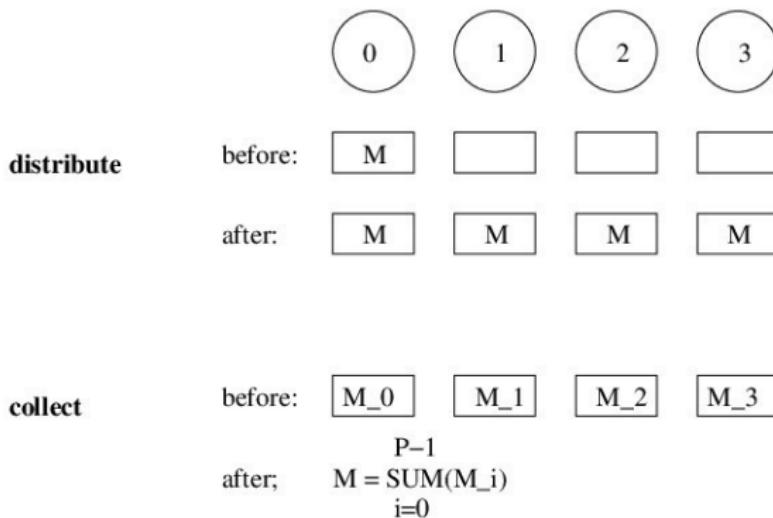
- ▶ Before first send/receive to/from a process a connection has to be established by **connect**.
- ▶ **send/recv** operations are assigned a channel instead of a process id as address.
- ▶ Several processes can send on a channel but only one can receive.
 - ★ **send(channel,expr₁,...,expr_n)**
 - ★ **recv(channel,var₁,...,var_n)**.
- We will use no channel-oriented functions.

Global Communication

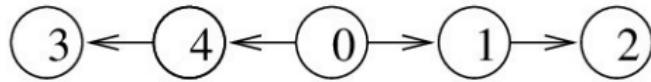
- A process wants to send *identical* data to all other processes
- *one-to-all broadcast*
- dual operation is the collection of individual results on a single process,
e.g. sum generation (all associative operators are possible)
- We consider distribution across different topologies and calculate the
time demand for store & forward as well as cut-through routing
- Algorithms for the collection result from reversing the sequence and
direction of the communications
- The following cases are discussed in detail:
 - ▶ One-to-all
 - ▶ All-to-one
 - ▶ One-to-all with individual messages
 - ▶ All-to-all with individual messages

One-to-all: Ring

A process wants to send *identical* data to all other processes:



Here: Communication in ring topology with store & forward:



One-to-all: Ring

Program (One-to-all in the ring)

```
parallel one-to-all-ring
{
    const int P;
    process  $\Pi$ [int  $p \in \{0, \dots, P - 1\}$ ]
    {
        void one_to_all_broadcast(msg *mptr) {
            // receive messages
            if ( $p > 0 \wedge p \leq P/2$ )
                recv( $\Pi_{p-1}$ , *mptr);
            if ( $p > P/2$ )
                // pass messages to successor
                if ( $p \leq P/2 - 1$ )
                    send( $\Pi_{p+1}$ , *mptr);
                if ( $p > P/2 + 1 \vee p == 0$ )
                    send( $\Pi_{(p+P-1)\%P}$ , *mptr);
        }
        ...
        m=...;
        one_to_all_broadcast(&m);
    }
}
```

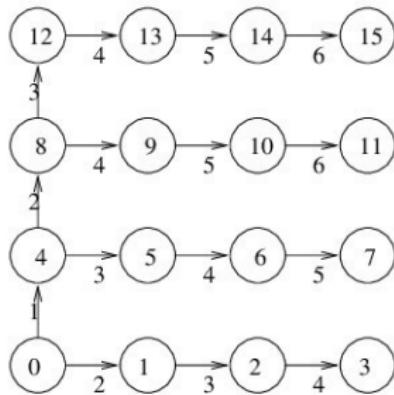
The time consumption for the operation is (nearest-neighbor communication!):

$$T_{one-to-all-ring} = (t_s + t_h + t_w \cdot n) \left\lceil \frac{P}{2} \right\rceil,$$

where $n = |\ast mptr|$ denotes the length of the message.

One-to-all: Array

Now we assume a 2D array structure for communication. The messages are routed along the following paths:



Look at the two-dimensional process index:

One-to-all: Array

Program (One to all on the array)

```
parallel one-to-all-array
{
    int P, Q;                                // Array size in x and y direction
    process Π[int[2]] (p, q) ∈ {0, ..., P - 1} × {0, ..., Q - 1} {
        void one_to_all_broadcast(msg *mptr) {
            if (p == 0)                      // first column
            {
                if (q > 0)                  recv(Π(p,q-1), *mptr);
                if (q < Q - 1)              send(Π(p,q+1), *mptr);
            }
            else
                if (p < P - 1)          recv(Π(p-1,q), *mptr);
                                            send(Π(p+1,q), *mptr);
        }

        msg m=...;
        one_to_all_broadcast(&m);
    }
}
```

The execution time for $P = 0$ is in a 2d array

$$T_{\text{one-to-all-array2D}} = 2(t_s + t_h + t_w \cdot n)(\sqrt{P} - 1)$$

and in a 3d array

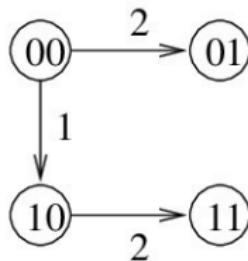
$$T_{\text{one-to-all-array3D}} = 3(t_s + t_h + t_w \cdot n)(P^{1/3} - 1).$$

One-to-all: Hypercube

- We advance in a recursive manner. For a hypercube of dimension $d = 1$ the problem can be solved in a trivial way:



- In a hypercube of dimension $d = 2$ node 0 sends first to node 2 and the problem is reduced to 2 hypercubes of dimension 1:



- In general for step $k = 0, \dots, d - 1$ the processes

send each a message to $\underbrace{p_{d-1} \dots p_{d-k}}_{k \text{ dimens.}}$ 0 $\underbrace{0 \dots 0}_{d-k-1 \text{ dimens.}}$
send each a message to $\underbrace{p_{d-1} \dots p_{d-k}}_{k \text{ dimens.}}$ 1 $\underbrace{0 \dots 0}_{d-k-1 \text{ dimens.}}$

One-to-all: Hypercube

Program (One to all in the hypercube)

```
parallel one-to-all-hypercube
{
    int d, P = 2d;
    process Π[int p ∈ {0, ..., P - 1}]{
        void one_to_all_broadcast(msg *mptr) {
            int i, mask = 2d - 1;
            for (i = d - 1; i ≥ 0; i --){
                mask = mask ⊕ 2i;
                if (p&mask == 0) {
                    if (p&2i == 0)           //the last i bits are 0
                        send(Πp⊕2i, *mptr);
                    else
                        recv(Πp⊕2i, *mptr);
                }
            }
        }
        msg m = „bla“; one_to_all_broadcast(&m);
    }
}
```

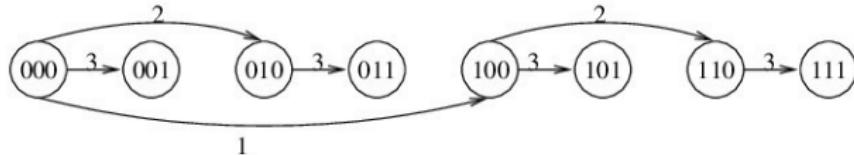
The time consumption is

$$T_{one-to-all-HC} = (t_s + t_h + t_w \cdot n) \lg P$$

Arbitrary source $src \in \{0, \dots, P - 1\}$: Substitute each p by $(p \oplus src)$.

One-to-all: Ring and array with cut-through routing

- If the hypercube algorithm is mapped on the ring, we obtain the following communication structure:

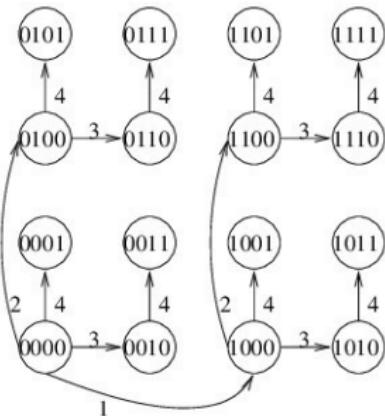


- There are no wires used twice, therefore we obtain by cut-through routing:

$$\begin{aligned} T_{one-to-all-ring-ct} &= \sum_{i=0}^{\lfloor \log_2 P - 1 \rfloor} (t_s + t_w \cdot n + t_h \cdot 2^i) \\ &= (t_s + t_w \cdot n) \lfloor \log_2 P \rfloor + t_h(P - 1) \end{aligned}$$

One-to-all: Ring and Array with Cut-Through Routing

When using an array structure one obtains the following communication structure:



Again there are no wire collisions and we obtain:

$$T_{\text{one-to-all-field-ct}} = \underbrace{2}_{\substack{\text{jede} \\ \text{Entfernung 2} \\ \text{mal}}} \sum_{i=0}^{\frac{\lceil d(P) \rceil}{2}-1} (t_s + t_w \cdot n + t_h \cdot 2^i)$$

One-to-all: Ring and Array with Cut-Through Routing

$$\begin{aligned} T_{\text{one-to-all-field-ct}} &= (t_s + t_w \cdot n) 2^{\frac{\text{Id } P}{2}} + t_h \cdot 2 \underbrace{\sum_{i=0}^{\frac{\text{Id } P}{2}-1} 2^i}_{=2^{\frac{\text{Id } P}{2}}-1} \\ &= \sqrt{P} (t_s + t_w \cdot n) + t_h \cdot 2(\sqrt{P} - 1) \end{aligned}$$

Especially with the array topology the term covering t_h is negligible and we obtain the hypercube performance also for less rich topologies! Also for $P = 1024 = 32 \times 32$ the time is not determined by t_h , thus because of cut-through routing no physical hypercube structures are necessary anymore.

Distributed-Memory Programming Models II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 205
D-69120 Heidelberg
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

Distributed-Memory Programming Models II

Communication by message passing

- MPI Standard
- Global communication for different topologies
 - ▶ Array (1D / 2D / 3D)
 - ▶ Hypercube
- Local exchange

MPI: Introduction

The *Message Passing Interface* (MPI) is a portable library of functions for message exchange between processes.

- MPI has been designed 1993/94 by an international gremium.
- Is available on nearly all platforms, including the free implementations OpenMPI, MPICH and LAM.
- Characteristics:
 - ▶ Library for binding with C-, C++- and FORTRAN programs (no language extension).
 - ▶ Large choice of point-to-point communication functions.
 - ▶ Global communication.
 - ▶ Data conversion for heterogeneous systems.
 - ▶ Creation of partial sets and topologies.
- MPI consists of over 125 functions, that are described on over 800 pages in the standard. Thus we can only discuss a small choice of its functionality.
- MPI-1 has no possibilities for dynamic process generation, this is possible in MPI-2, furthermore in-/output.
MPI-3 is released since 09/2012 with minor extensions.

MPI: Hello World

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

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

    MPI_Init (&argc,&argv);
    MPI_Comm_size (MPI_COMM_WORLD, &P);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

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

    return 0;
}
```

- SPMD style!
- Compilation and startup is done with
 - mpicc -o hello hello.c
 - mpirun -machinefile machines -np 8 hello
- machines contains names of the usable machines.

MPI: Blocking Communication I

- MPI supports different variants of blocking and non-blocking communication, guards for the **receive** function, as well as data conversion during communication between machines with distinct data formats.
- The fundamental blocking communication functions are defined by:

```
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);
```

- A message in MPI consists of plain *data* and an envelope (meta information).
- Data is always an array of elementary data types. This enables MPI to handle data conversion.

MPI: Blocking Communication II

- The envelope consists of:
 - 1 Number of sender,
 - 2 Number of receiver,
 - 3 Tag,
 - 4 and a Communicator.
- Number of sender and receiver is called rank.
- Tag is also an integer number and serves as identifier for different messages between identical communication partners.
- A communicator is defined by a partial set of the processes and a communication context. Messages, that belong to different contexts, do not influence each other, resp. sender and receiver have to use the same communicator.
- Meanwhile we only use the default communicator `MPI_COMM_WORLD` (all started processes).

MPI: Blocking Communication III

- MPI_Send is fundamentally blocking, there are however diverse variants:
 - ▶ *buffered send* (B): If the receiver has still not executed the corresponding **recv** function, the message is buffered on sender side. A „buffered send“ is, while assuming enough buffer space, always immediately finished. In comparison to asynchronous communication can the send buffer message be reused immediately.
 - ▶ *synchronous send* (S): Finishing of synchronous send indicates, that the receiver executes a **recv** function and has started to read the data.
 - ▶ *ready send* (R): A ready send may only be executed, if the receiver has already executed the corresponding **recv**. Otherwise the call results in an error.
- The according calls are designated MPI_Bsend, MPI_Ssend and MPI_Rsend.
- The MPI_Send instruction has either the semantics of MPI_Bsend or MPI_Ssend, according to implementation specifics. Therefore MPI_Send can, but must not block. In every case the send buffer message can be reused immediately after finishing.

MPI: Blocking Communication IV

- The instruction `MPI_Recv` is in every case blocking.
- The argument `status` contains source, tag, and error status of the receiving message.
- For the arguments `src` and `tag` can the values `MPI_ANY_SOURCE` resp. `MPI_ANY_TAG` be inserted. Thus `MPI_Recv` contains the functionality of `recv_any`.
- A non-blocking guard function for the receiving of messages is available by means of

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,  
               int *flag, MPI_Status *status);
```

MPI: Non-blocking and Global Communication I

- For non-blocking communication there are 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);
```

available.

- Via the `MPI_Request` objects it is possible to determine the state of the communication request (corresponds to `msgid` in our pseudo code).
- Therefore exists (beneath other) the function

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status
```

- The `flag` is set to `true` ($\neq 0$), if the communication denoted by `req` has been finished. In this case `status` contains information about sender, receiver and error status.

It needs to be considered, that the `MPI_Request` object gets invalid as soon as `MPI_Test` returns with `flag==true`. It may then not be used again.

MPI: Non-blocking and Global Communication II

- For global communication are available (beneath other):

```
int MPI_Barrier(MPI_Comm comm);
```

blocks all processes of a communicator until all are there.

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

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

- For the collection of data different operations are present. We describe only one of these:

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

combines the data in the input buffer `sbuf` of all processes by the associative operator `op`. The final result is available in the receive buffer `rbuf` of the process `root`. Examples for `op` are `MPI_SUM`, `MPI_MAX`.

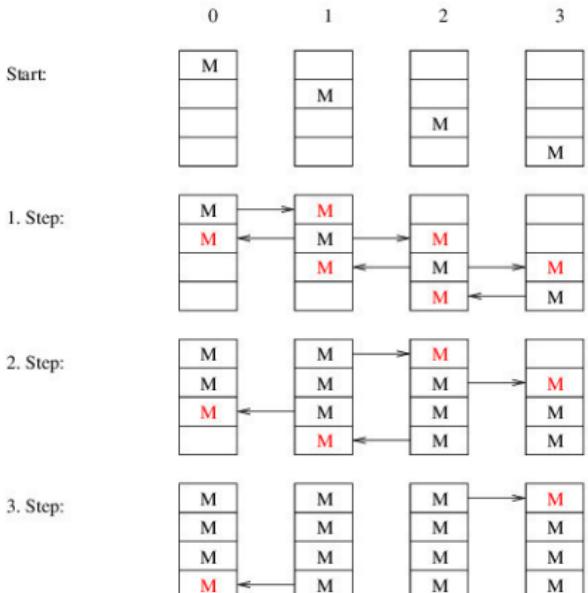
# All-to-all: 1D Array, Principle

We skip the ring topology and consider the 1D array at once: Each process sends into both directions.

Each wants to send data to all (variant: accumulate with associative operator):

|          |       |       |       |       |
|----------|-------|-------|-------|-------|
|          | 0     | 1     | 2     | 3     |
| vorher:  | $M_0$ | $M_1$ | $M_2$ | $M_3$ |
| nachher: | $M_0$ | $M_0$ | $M_0$ | $M_0$ |
|          | $M_1$ | $M_1$ | $M_1$ | $M_1$ |
|          | $M_2$ | $M_2$ | $M_2$ | $M_2$ |
|          | $M_3$ | $M_3$ | $M_3$ | $M_3$ |

Variante mit Akkumulieren

$$\sum_i M_i$$
$$\sum_i M_i$$
$$\sum_i M_i$$
$$\sum_i M_i$$


We use synchronous communication. Decide who sends/receives by black-white coloring:

# All-to-all: 1D Array, Code I

## Program (All-to-all in 1D array)

```
parallel all-to-all-1D-array
{
 const int P;
 process Π[int p ∈ {0, ..., P - 1}]
 {
 void all_to_all_broadcast(msg m[P])
 {
 int i,
 from_left = p - 1, from_right = p + 1,
 to_left = p, to_right = p;
 for (i = 1; i < P; i++)
 {
 if ((p%2) == 1) // black/white coloring
 {
 if (from_left ≥ 0) recv(Πp-1, m[from_left]);
 if (to_right ≥ 0) send(Πp+1, m[to_right]);
 if (from_right < P) recv(Πp+1, m[from_right]);
 if (to_left < P) send(Πp-1, m[to_left]);
 }
 else
 {
 if (to_right ≥ 0) send(Πp+1, m[to_right]);
 if (from_left ≥ 0) recv(Πp-1, m[from_left]);
 if (to_left < P) send(Πp-1, m[to_left]);
 if (from_right < P) recv(Πp+1, m[from_right]);
 }
 }
 ...
 }
 }
}
```

# All-to-all: 1D Array, Code II

Program (All-to-all in 1D array cont.)

```
parallel all-to-all-1D-feld cont.
{
 ...
 from_left--; to_right--;
 from_right++; to_left++;
}
}
...
m[p] = „That is from p!“;
all_to_all_broadcast(m);
...
}
}
```

## All-to-all: 1D Array, Runtime

- For the runtime analysis consider  $P$  odd,  $P = 2k + 1$ :

$$\underbrace{\Pi_0, \dots, \Pi_{k-1}}_k, \Pi_k, \underbrace{\Pi_{k+1}, \dots, \Pi_{2k}}_k$$

|                 |          |          |            |
|-----------------|----------|----------|------------|
| Process $\Pi_k$ | receives | $k$      | from left  |
|                 | sends    | $k + 1$  | to right   |
|                 | receives | $k$      | from right |
|                 | sends    | $k + 1$  | to left.   |
| <hr/>           |          | $\sum =$ | $4k + 2$   |
|                 |          |          | $= 2P$     |

- After that  $\Pi_k$  has all messages. Now the message from 0 has to be send to  $2k$  and vice versa. This needs again additional

$$(\underbrace{k}_{\text{Entfernung}} - 1) \cdot \underbrace{2}_{\text{senden u. empfangen}} + \underbrace{1}_{\text{der Letzte empfängt nur}} = 2k - 1 = P - 2$$

so we have in total

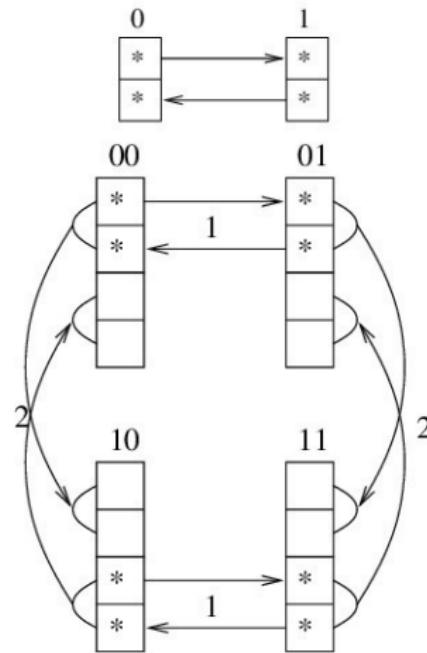
$$T_{\text{all-to-all-array-1d}} = (t_s + t_h + t_w \cdot n)(3P - 2)$$

## All-to-all: Hypercube

The following algorithm for the hypercube is known as *dimension exchange* and is again derived recursively.

Start with  $d = 1$ :

With four processes exchange processes 00 and 01 resp. 10 and 11 first their data, then exchange 00 and 10 resp. 01 and 11 each two data



## All-to-all: Hypercube

```
● void all_to_all_broadcast(msg m[P]) {
 int i, mask = 2d - 1, q;
 for (i = 0; i < d; i++) {
 q = p ⊕ 2i;
 if (p < q) { // who first?
 send(Πq, m[p&mask], ..., m[p&mask + 2i - 1]);
 recv(Πq, m[q&mask], ..., m[q&mask + 2i - 1]);
 }
 else {
 recv(Πq, m[q&mask], ..., m[q&mask + 2i - 1]);
 send(Πq, m[p&mask], ..., m[p&mask + 2i - 1]);
 }
 mask = mask ⊕ 2i;
 }
}
```

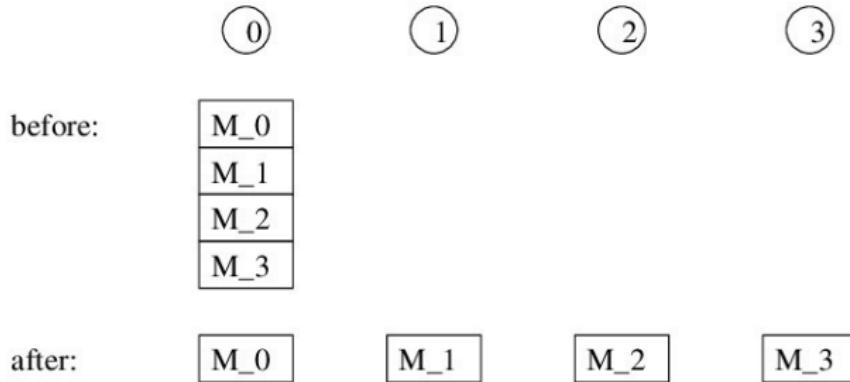
### Runtime analysis:

$$\begin{aligned} T_{all-to-all-bc-hc} &= \underbrace{2}_{\substack{\text{send a.} \\ \text{receive}}} \sum_{i=0}^{\log P-1} t_s + t_h + t_w \cdot n \cdot 2^i = \\ &= 2 \log P (t_s + t_h) + 2t_w n(P-1). \end{aligned}$$

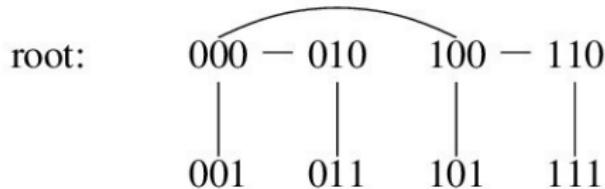
- For large messages the HC has no advantage: Each has to receive  $n$  words from each, whatever the topology looks like.

# One-to-all with indiv. messages: Hypercube, Principle

- Process 0 sends to each a message, but to each a different one!



- Example is the in/output to a *single* file.
- Because of variation purposes we consider the output, this means all-to-one with individual messages.
- We use the well-known hypercube structure:

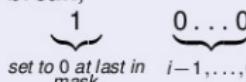


# One-to-all with indiv. messages: Hypercube, Code I

Program (*Collection of individual messages on the hypercube*)

**parallel** all-to-one-personalized

{

```
 const int d, P = 2d;
 process $\Pi[\text{int } p \in \{0, \dots, P - 1\}]$ {
 void all_to_one_pers(msg m) {
 int mask, i, q, root;
 // determine p's root: How many bits from end are zero?
 mask = 2d - 1;
 for (i = 0; i < d; i++)
 {
 mask = mask \oplus 2i;
 if (p&mask \neq p) break;
 } // $p = p_{d-1} \dots p_{i+1}$ 
 if (i < d) root = p \oplus 2i; // my root direction
 // own data
 if (p == 0) self-processing(m);
 else send(root,m); // pass up
 ...
 }
```

# One-to-all with indiv. messages: Hypercube, Code II

Program (*Collection of individual messages on the hypercube cont.*)

parallel all-to-one-personalized cont.

{

...

// process sub-trees:

mask =  $2^d - 1$ ;

for ( $i = 0; i < d; i++$ ) {

    mask = mask  $\oplus 2^i$ ;  $q = p \oplus 2^i$ ;

    if ( $p \& mask == p$ )

//  $p = p_{d-1} \dots p_{i+1} \quad 0 \quad \underbrace{0 \dots 0}_{i-1, \dots, 0}$

//  $q = p_{d-1} \dots p_{i+1} \quad 1 \quad \underbrace{0 \dots 0}_{i-1, \dots, 0}$

//  $\Rightarrow I$  am root of a HC of dim.  $i + 1$ !

        for ( $k = 0; k < 2^i; k++$ ) {

            recv( $\Pi_q, m$ );

            if ( $p == 0$ ) process( $m$ );

            else send( $\Pi_{root}, m$ );

        }

    }

}

}

# One-to-all with indiv. messages: Runtime, Variants

For the *runtime* one has for large ( $n$ ) messages

$$T_{\text{all-to-one-pers}} \geq t_w n(P - 1)$$

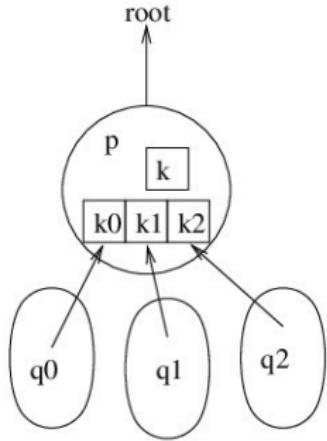
because of the pipelining.

Some variants are worth considering:

- *Individual length of messages*: Here sends one before sending the message itself only the length information (this is practically necessary → MPI).
- *Arbitrary message length* (but only finite intermediate buffer!): subdivide message into packets of fixed length.
- *Sorted output*: Each message  $M_i$  (of process  $i$ ) is associated a sorting key  $k_i$ . The messages should be processed by process 0 in increasing order of keys, *without* intermediate buffering of all messages.

# One-to-all with indiv. messages: Runtime, Variants

- With *sorted output* one may be inspired by the following idea:



$p$  has three „servants“,  $q_0, q_1, q_2$ , that represent complete subtrees.

Each  $q_i$  sends its next smallest key to  $p$ , that searches the smallest key and then itself passes this key with its already transmitted data further.

# Distributed-Memory Programming Models III

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Distributed-Memory Programming Models III

Communication using message passing

- Global communication
- Local exchange
- Synchronisation with time stamps
- Distributed termination
- MPI standard

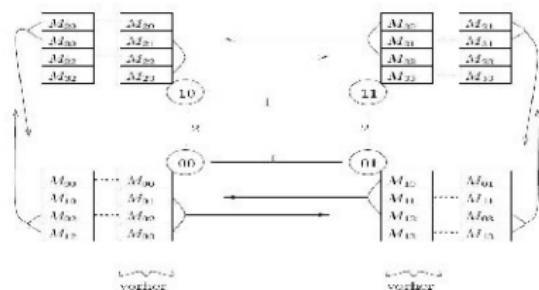
# All-to-all with indiv. Messages: Principle

Here has *each* process  $P - 1$  messages, one for *each other* process. There are thus  $(P - 1)^2$  individual messages to send:

|          | 0        | 1        | 2        | 3        |
|----------|----------|----------|----------|----------|
| vorher:  | $M_{00}$ | $M_{10}$ | $M_{20}$ | $M_{30}$ |
|          | $M_{01}$ | $M_{11}$ | $M_{21}$ | $M_{31}$ |
|          | $M_{02}$ | $M_{12}$ | $M_{22}$ | $M_{32}$ |
|          | $M_{03}$ | $M_{13}$ | $M_{23}$ | $M_{33}$ |
| nachher: | $M_{00}$ | $M_{01}$ | $M_{02}$ | $M_{03}$ |
|          | $M_{10}$ | $M_{11}$ | $M_{12}$ | $M_{13}$ |
|          | $M_{20}$ | $M_{21}$ | $M_{22}$ | $M_{23}$ |
|          | $M_{30}$ | $M_{31}$ | $M_{32}$ | $M_{33}$ |

The figure shows already an application: Matrix transposition for column-wise subdivision.

As always, the hypercube (here  $d=2$ ):



## All-to-all with indiv. Messages: General Derivation I

- In general we have the following situation in step  $i = 0, \dots, d - 1$ :
- Process  $p$  communicates with  $q = p \oplus 2^i$  and sends to him

all data of processes  $p_{d-1} \dots p_{i+1} \quad p_i \quad x_{i-1} \dots x_0$   
for the processes  $y_{d-1} \dots y_{i+1} \quad \bar{p}_i \quad p_{i-1} \dots p_0$ ,

where the  $x$ e and  $\gamma$ s represent all possible entries.

- $\bar{p}_i$  is negation of a bit.
- There are thus always  $P/2$  messages sent in each communication.
- Process  $p$  stores at each point in time  $P$  data.
- An individual data is underway from process  $r$  to process  $s$ .
- Each data is identified by  $(r, s) \in \{0, \dots, P - 1\} \times \{0, \dots, P - 1\}$ .
- We write

$$\mathcal{M}_p^i \subset \{0, \dots, P - 1\} \times \{0, \dots, P - 1\}$$

for the data, that stores process  $p$  at the beginning of step  $i$ , thus before communication.

# All-to-all with indiv. Messages: General Derivation II

- At the start of step 0 process  $p$  owns the data

$$\mathcal{M}_p^0 = \{(p_{d-1} \dots p_0, y_{d-1} \dots y_0) \mid y_{d-1}, \dots, y_0 \in \{0, 1\}\}$$

- After communication in step  $i = 0, \dots, d-1$  has  $p$  the data  $\mathcal{M}_p^{i+1}$ , that result from  $\mathcal{M}_p^i$  and the following rule ( $q = p_{d-1} \dots p_{i+1} \bar{p}_i p_{i-1} \dots p_0$ ):

$$\mathcal{M}_p^{i+1} = \mathcal{M}_p^i$$

  
sends  $p$  to  $q$

$$\{(p_{d-1} \dots p_{i+1} p_i x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} \bar{p}_i p_{i-1} \dots p_0) \mid x_j, y_j \in \{0, 1\} \forall j\}$$

  
receives  $p$  from  $q$

$$\{(p_{d-1} \dots p_{i+1} \bar{p}_i x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} p_i p_{i-1} \dots p_0) \mid x_j, y_j \in \{0, 1\} \forall j\}$$

# All-to-all with indiv. Messages: General Derivation III

- By induction applies therefore for  $p$  after communication in step  $i$ :

$$\mathcal{M}_p^{i+1} = \{(p_{d-1} \dots p_{i+1} x_i \dots x_0, y_{d-1} \dots y_{i+1} p_i \dots p_0) \mid x_j, y_j \in \{0, 1\} \forall j\}$$

because of

$$\begin{aligned}\mathcal{M}_p^{i+1} &= \{(p_{d-1} \dots p_{i+1} \underbrace{p_i}_{\text{what } i \text{ do not need}} x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} p_i \dots p_0) \mid \dots\} \\ &\cup \{(p_{d-1} \dots p_{i+1} \overline{p_i} x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} \overline{p_i} \dots p_0) \mid \dots\} \\ &\backslash \underbrace{\{\dots\}}_{\text{what } i \text{ do not need}} \\ &= \{(p_{d-1} \dots p_{i+1} x_i x_{i-1} \dots x_0, y_{d-1} \dots y_{i+1} p_i \dots p_0) \mid \dots\}\end{aligned}$$

# All-to-all with indiv. Messages: Code

```
void all_to_all_pers(msg m[P])
{
 int i, x, y, q, index;
 msg sbuf[P/2], rbuf[P/2];
 for (i = 0; i < d; i++)
 {
 q = p ⊕ 2i; // my partner

 // assemble send buffer:
 for (y = 0; y < 2d-i-1; y++)
 for (x = 0; x < 2i; x++)
 sbuf[y · 2i + x] = m[y · 2i+1 + (q&2i) + x];
 // < P/2 (!)

 // exchange messages:
 if (p < q)
 { send(Πq, sbuf[0], ..., sbuf[P/2 - 1]); recv(Πq, rbuf[0], ..., rbuf[P/2 - 1]); }
 else
 { recv(Πq, rbuf[0], ..., rbuf[P/2 - 1]); send(Πq, sbuf[0], ..., sbuf[P/2 - 1]); }

 // disassemble receive buffer:
 for (y = 0; y < 2d-i-1; y++)
 for (x = 0; x < 2i; x++)
 m[y · 2i+1 + (q&2i) + x] = sbuf[y · 2i + x];
 // exactly what has been sent is
 // substituted
 }
} // end all_to_all_pers
```

## All-to-all with indiv. Messages: Code

Complexity analysis:

$$\begin{aligned} T_{all-to-all-pers} &= \sum_{i=0}^{\lfloor dP-1 \rfloor} \underbrace{2}_{\text{send and receive}} (t_s + t_h + t_w \underbrace{\frac{P}{2}}_{\text{in every step}} n) = \\ &= 2(t_s + t_h) \lfloor dP \rfloor + t_w n P \lfloor dP \rfloor. \end{aligned}$$

# MPI: Communicators and Topologies I

In all up to now considered MPI communication functions existed an argument of type `MPI_Comm`. Such a *communicator* contains the following abstractions:

- *Process group*: A communicator can be used to build a subset of all processes. Only these then take part in a global communication. The pre-defined communicator `MPI_COMM_WORLD` consists of all started processes.
- *Context*: Each communicator defines an individual communication context. Messages can only be received within the same context, in which they have been sent. Such e.g. a library with numerical functions can use its own communicator. Messages of the library are then completely encapsulated from messages in the user program. Therefore messages of the library can not erroneously be received by the user program and vice versa.
- *Virtual topology*: A communicator represents only a set of processes  $\{0, \dots, P - 1\}$ . Optionally this set can be enhanced by an additional structure, e.g. a multi-dimensional field or a general graph.

# MPI: Communicators and Topologies II

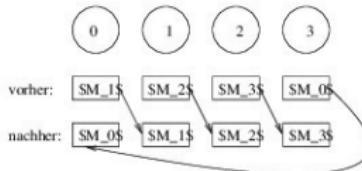
- *Additional attributes:* An application (e.g. a library) can associate with the communicator arbitrary static data. The communicator serves as medium to retain data from a call of the library to the next.
- This is an *intra-communicator*, that only enables communication *within* a process group.
- Furthermore there are *inter-communicators*, that support communication of *distinct* process groups. These are not considered further at the moment!
- As a possibility to create a new (intra-) communicator we have a look at the function

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

- `MPI_Comm_split` is a collective operation, that has to be called by *all* processes of the communicator `comm`. All processes with equal value for the argument `color` create each a new communicator. The sequence (rank) within the new communicator is managed by the argument `key`.

# Local Exchange: Shifting in the Ring I

- Consider the following problem: Each process  $p \in \{0, \dots, P - 1\}$  has to send data to  $(p + 1) \% P$ :



- Naive realisation with synchronous communication results in deadlock:

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

- Avoiding the deadlock (e. g. exchanging of **send/recv** in one process) does not deliver maximal possible parallelism.
- Asynchronous communication is often not preferential because of efficiency reasons.

## Local Exchange: Shifting in the Ring II

- Solution: *Coloring*. Be  $G = (V, E)$  a graph with

$$V = \{0, \dots, P - 1\}$$

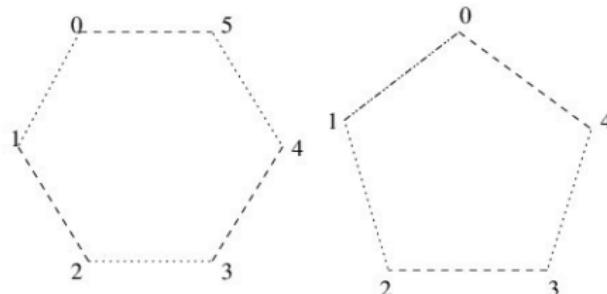
$$E = \{e = (p, q) \mid \text{process } p \text{ has to communicate with process } q\}$$

- There are the *edges* to color in such a way, that each node has only connections to edges with different colors. The assignment of colors is described by the mapping

$$c: E \rightarrow \{0, \dots, C - 1\}$$

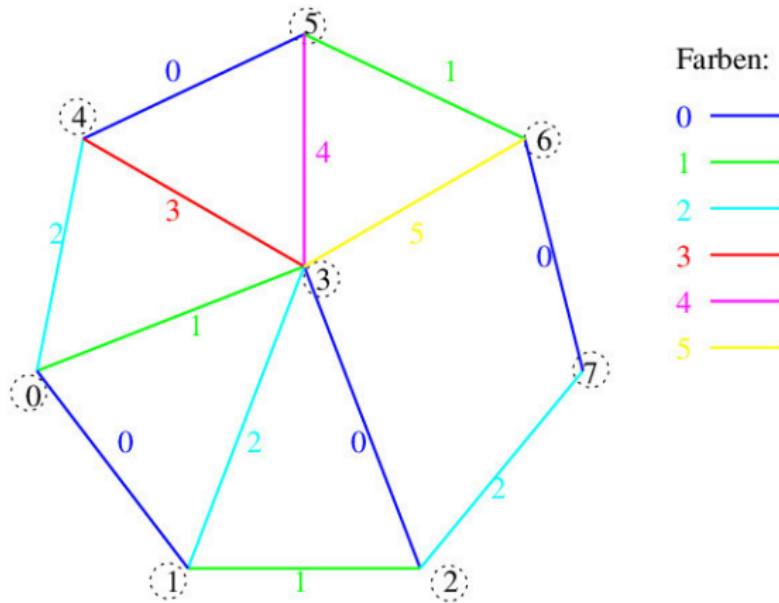
, where  $C$  is the count of necessary colors.

- Shifting in the *ring* needs two colors for  $P$  being even and three color for  $P$  being odd:



# Local Exchange: General Graph I

Establish the communication relations a general graph, then the coloring is determined by an algorithm.



Here a more or less sequential heuristic:

# Local Exchange: General Graph II

## Program (Distributed Coloring)

```
parallel coloring
{
 const int P;
 process Π [int $p \in \{0, \dots, P - 1\}$]
 {
 int nbs; // number of neighbors
 int nb[nbs]; // $nb[i] < nb[i + 1]$!
 int color[nbs]; // the result
 int index[MAXCOLORS]; // free color management
 int i, c, d;
 for (i = 0; i < nbs; i++) index[i] = -1;
 for (i = 0; i < nbs; i++)
 c = 0; // find color for connection to $nb[i]$
 // start with color 0
 while(1)
 c = min{k | index[k] < 0}; // next free color $\geq c$
 if ($p < nb[i]$) { send($\Pi_{nb[i]}, c$); recv($\Pi_{nb[i]}, d$); }
 else { recv($\Pi_{nb[i]}, c$); send($\Pi_{nb[i]}, d$); }
 if (c == d) { // the two have an agreement
 index[c] = i; color[i] = c; break;
 } else c = max(c, d);
 }
 }
 }
}
```

# Lamport Time Stamps I

- Goal: Ordering of events in distributed systems.
- Events: Execution of (marked) instructions.
- The ideal situation would be a global clock, but this is not available in distributed systems, since the sending of messages always is in conjunction with delays.
- *Logical clock*: Time points, that have been assigned to events, shall not be in obvious contradiction to a global clock.

$\Pi_1:$   
 $a = 5;$

$\dots;$   
 $b = 3;$   
**send**( $\Pi_2, a$ );

$\vdots$   
**recv**( $\Pi_2, f$ );

$\Pi_2:$

$\dots;$   
 $c = 4;$   
 $\dots;$   
**recv**( $\Pi_1, b$ );  
 $d = 8;$

$\vdots$   
**recv**( $\Pi_3, e$ );  
 $f = bde;$   
 $\vdots$   
**send**( $\Pi_1, f$ );

$\Pi_3:$

$\vdots$   
 $e = 7;$   
**send**( $\Pi_2, e$ );

## Lamport Time Stamps II

- Be  $a$  an event in process  $p$  and  $C_p(a)$  the time stamp,  $p$  the associated process, e. g.  $C_2(f = bde)$ , then the time stamps should have the following properties:
  - 1 Be  $a$  and  $b$  two events in the same process  $p$ , where  $a$  occurs before  $b$ , then shall be  $C_p(a) < C_p(b)$ .
  - 2 Process  $p$  sends a message to  $q$ , then shall be  $C_p(\text{send}) < C_q(\text{receive})$ .
  - 3 For two arbitrary events  $a$  and  $b$  in arbitrary processes  $p$  resp.  $q$  be  $C_p(a) \neq C_q(b)$ .
- 1 and 2 represent the causality of events: If in a parallel program can surely be said, that  $a$  in  $p$  occurs *before*  $b$  in  $q$ , then applies  $C_p(a) < C_q(b)$  too.
- Only with the properties 1 and 2  $a \leq_C b : \iff C_p(a) < C_q(b)$  would be a half ordering on the set of all events.
- Property 3 results then in a total ordering.

# Lamport Time Stamps: Implementation

## Program (Lamport time stamps)

```
parallel Lamport time stamps
{
 const int P; // whats this?
 int d = min{i| $2^i \geq P\}$; // how many bit positions has P.

 process Π [int p $\in \{0, \dots, P - 1\}$]
 {
 int C=0; // the clock
 int t, s, r; // only for the example
 int Lclock(int c) // output of a new time stamp
 {
 C=max(C, c/2d); // rule 2
 C++; // rule 1
 return C · 2d + p; // rule 3
 // the last d bits contain p
 }

 //application:
 //A local event happens
 t=Lclock(0);

 s=Lclock(0); // send
 send(Π_q , message, s); // the time stamp is sent together!

 recv(Π_q , message, r);
 r=Lclock(r); // receivers also the time stamp of the receiver!
 // thus applies Cp(r) > Cq(s)!

 }
}
```

# Lamport Time Stamps: Implementation

- Management of the time stamps is in response of the user. Ordinarily one necessitates time stamps only for very specific events (see below).
- Overflow of the counter has not been considered.

# Distributed Mutual Exclusion with Time Stamps I

- Problem: From a set of distributed processes exactly one shall do something (e. g. control a device, serve as server, ...). Like in the case of a critical section the processes have to decide which is next.
- A possibility would be, that just one process decides who is next.
- We now present a distributed solution:
  - ▶ Does a process want to enter it sends a message to all others.
  - ▶ As soon as it has gotten an answer from all (there is no no!) it can enter.
  - ▶ A process confirms only, if it doesn't want to enter or if the time stamp of an entry query is larger than that of the others.
- Solution works with a local monitor process.

# Distributed Mutual Exclusion with Time Stamps II

Program (Distributed mutual exclusion with Lamport time stamps)

parallel DME-timestamp // Distributed Mutual Exclusion

```
{
 int P; const int REQUEST=1, REPLY=2; // messages

 process Π[int p ∈ {0, ..., P - 1}]
 {
 int C=0, mytime; // clock
 int is_requesting=0, reply_pending, reply_deferred[P]={0,...}; // deferred processes

 process M[int p' = p] // the monitor
 {
 int msg, time;
 while(1){
 recv_any(π,q,msg,time);
 if (msg==REQUEST)
 {
 [Lclock(time)];

 if(is_requesting ∧ mytime < time)
 reply_deferred[q]=1;
 else
 asend(Mq,p,REPLY,0);
 }
 else reply_pending--;
 }
 }
 ...
 }
}
```

# Distributed Mutual Exclusion with Time Stamps II

Program (Distributed mutual exclusion with Lamport time stamps cont.)

parallel DME-timestamp // Distributed Mutual Exclusion cont.

{

```
 ...
 void enter_cs() // to enter the critical section
 {
 int i;
 [mytime=Lclock(0); is_requesting=1;]
 reply_pending=P - 1; // critical section
 for (i=0; i < P; i++)
 if (i ≠ p) send(Mi,p,REQUEST,mytime); // so many answers do I expect
 while (reply_pending> 0); // busy wait
 }
 void leave_cs()
 {
 int i;
 is_requesting=0;
 for (i=0; i < P; i++) // inform waiting processes
 if (reply_deferred[i])
 {
 send(Mi,p,REPLY,0);
 reply_deferred[i]=0;
 }
 }
 enter_cs(); /* critical section */ leave_cs();
 } // end process
}
```

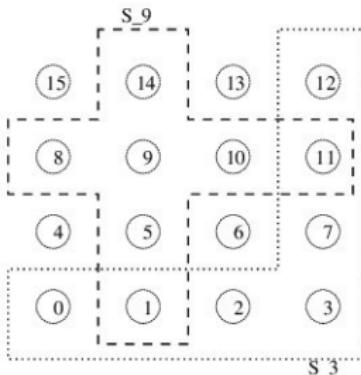
# Distributed Mutual Exclusion with „Voting“ I

- The algorithm above needs  $2P$  messages per process to enter the critical section. With voting we will only need  $O(\sqrt{P})$ .
- Especially a process doesn't need to ask *all* others before it may enter.
- Idea:
  - ▶ The related processes acquire for entry into the critical section. These are called *candidates*
  - ▶ All (or some, see below) vote who may enter. These are called *voters*. Each can be candidate or voter.
  - ▶ Instead of absolute majority we require only relative majority: A process may enter as soon as it knows, that no other can have more votes than itself.
- Each process is assigned a voting district  $S_p \subseteq \{0, \dots, P - 1\}$ . It applies the coverage property:

$$S_p \cap S_q \neq \emptyset \quad \forall p, q \in \{0, \dots, P - 1\}.$$

# Distributed Mutual Exclusion with „Voting“ II

- The voting districts for 16 processes look like this:



- A process  $p$  can enter, if it gets all votes of its voting district. Since no other process  $q$  can enter: According to prerequisite there exists  $r \in S_p \cap S_q$  and  $r$  has decided to vote for  $p$ , thus  $q$  cannot have gotten all votes.
- Danger of deadlock: Is  $|S_p \cap S_q| > 1$  thus one can decide for  $p$  and another for  $q$ , both never may enter. Solution of deadlocks with Lamport time stamps.

# Optimality of Voting Districts I

- Question: How small can the voting districts be?
- Again: Each  $p$  has its voting district  $S_p \subseteq \{0, \dots, P - 1\}$  and we require  $S_p \cap S_q \neq \emptyset$ .
- But this would allow e. g.  $S_p = \{0\}$  for all  $p$ , what we do not want.
- Define  $D_p$  as the set of processes for which  $p$  has to vote:

$$D_p = \{q | p \in S_q\}$$

- We additionally require that for all  $p$ :

$$|S_p| = K, \quad |D_p| = D.$$

This excludes the trivial solution from above.

- With this assumption even holds  $D = K$ , since define the set of all pairs  $(p, q)$  with  $p$  chooses for  $q$ , d.h. :

$$A = \{(p, q) | 0 \leq p < P \wedge q \in D_p\}.$$

## Optimality of Voting Districts II

- On the other side define the set of all pairs  $(p, q)$  where  $p$  has to be voted by  $q$ :

$$B = \{(p, q) \mid 0 \leq p < P \wedge q \in S_p\}.$$

Because of  $q \in S_p \Leftrightarrow p \in D_q$  holds  $(p, q) \in B \Leftrightarrow (q, p) \in A$  thus  $|A| = |B|$ .  
For the sizes applies  $|A| = P \cdot D$  and  $|B| = P \cdot K$  thus  $D = K$ .

- For fixed  $K (= D)$  we maximize now the number of voting districts (processors)  $P$ :

- Choose an arbitrary voting district  $S_p$ . This has  $K$  members.
- Choose an arbitrary  $r \in S_p$ . This  $r$  is member in  $D$  voting districts (set  $D_r$ ) where one is  $S_p$  (obviously is  $p \in D_r$ . Therefore we count  $K(D - 1) + 1$  voting districts.
- More cannot exist, since for arbitrary  $q$  applies: There is a  $r$  with  $r \in S_p \cap S_q$  and thus  $q \in D_r$ . We have thus all gotten.

Thus it holds that

$$P \leq K(K - 1) + 1$$

or

$$K \geq \frac{1}{2} + \sqrt{P - \frac{3}{4}}.$$

# Voting: Implementation I

## Program (Distributed Mutual Exclusion with Voting)

```
parallel DME-Voting
{
 const int P = 7.962;
 const int REQUEST=1, YES=2, INQUIRE=3, RELINQUISH=4, RELEASE=5;
 // „inquire“ = „sich erkundigen“; „relinquish“ = „aufgeben“, „verzichten“
 process Π[int p ∈ {0, ..., P - 1}]
 {
 int C=0, mytime;

 void enter_cs() // wants to enter critical section
 {
 int i, msg, time, yes_votes=0;
 [mytime=Lclock(0);]
 for (i ∈ Sp) asend(Vi,p, REQUEST,mytime); // time of my request
 // send request to voting districts
 while (yes_votes < |Sp|) {
 recv_any(π,q,msg,time);
 if (msg==YES) yes_votes++;
 if (msg==INQUIRE)
 if (mytime==time) // receive from q
 if (q choose) // q choose
 if (q wants vote back) // q wants vote back
 if (now current request) // now current request
 if (there may be old on the way) // there may be old on the way
 asend(Vq,p,RELINQUISH,0); // passes back
 yes_votes--;
 }
 }
 } // end enter_cs
 ...
}
```

# Voting: Implementation II

## Program (Distributed Mutual Exclusion with Voting cont. 1)

```
parallel DME-Voting cont. 1
{
 ...
 void leave_cs()
 {
 int i;
 for (i ∈ Sp) asend(Vi, p, RELEASE, 0);
 // There could be still not processed INQUIRE messages for this
 // critical section exist, that are now obsolete.
 // These are then ignored in enter_cs.
 }

 // Example:
 enter_cs();
 ...; // critical section
 leave_cs();
}

}
```

# Voting: Implementation III

## Program (Distributed Mutual Exclusion with Voting cont. 2)

parallel DME-Voting cont. 2

```
{
 process V[int p' = p] // the voter for Π_p
 {
 int q, candidate, msg, time, have_voted=0, candidate_time, have_inquired=0;
 while(1) // runs forever
 {
 recv_any(π,q,msg,time); // receive it with sender
 if (msg==REQUEST) // request of a candidate
 {
 [Lclock(time);] // increase clock for later requests
 if (!have_voted) {
 asend(Π_q,p,YES,0); // I have still to vote
 candidate_time=time; // back to candidate process
 candidate=q; // remember whom I gave
 have_voted=1; // my vote.
 }
 else{
 store (q, time) in list; // yes, I have already voted
 if (time < candidate_time ∧ !have_inquired)
 {
 asend(Π_candidate,p,INQUIRE,candidate_time);
 // get back vote from candidate!
 // with the candidate_time it recognizes which request
 // it is: it could have happened, that it already entered.
 have_inquired=1;
 }
 }
 }...
 }
 }
}
```

# Voting: Implementation IV

## Program (Distributed Mutual Exclusion with Voting cont. 3)

parallel DME-Voting cont. 3

```
{
 ...
 else if (msg==RELINQUISH) // q is the candidate, that has
 {
 store (candidate, candidate_time) in list;
 take away and delete
 the entry with the smallest time from the list: (q, time)
 asend(Π_q, p , YES, 0); // There could exist others
 candidate_time=time; // vote for q
 candidate=q; // new candidate
 have_inquired=0; // no INQUIRE on the way
 }
 else if (msg==RELEASE) // q leaves the critical section
 {
 if (list is not empty)
 {
 // vote new
 take away and delete
 the entry with the smallest time from list: (q, time)
 asend(Π_q, p , YES, 0);
 candidate_time=time; // new candidate
 candidate=q; // forget all INQUIREs because obsolete
 }
 else
 have_voted=0; // noone need to be voted
 }
}
```

# Distributed Termination I

There are processes  $\Pi_0, \dots, \Pi_{P-1}$  defined, that communicate over a communication graph .

$$G = (V, E)$$

$$V = \{\Pi_0, \dots, \Pi_{P-1}\}$$

$$E \subseteq V \times V$$

With that process  $\Pi_i$  sends messages to the processes

$$N_i = \{j \in \mathbb{N} \mid (\Pi_i, \Pi_j) \in E\}$$

```
process Π_i [int $i \in \{0, \dots, P - 1\}$]
{
 while (1)
 {
 recv_any(who,msg),
 compute(msg);
 for (p $\in N_{msg} \subseteq N_i$)
 {
 msg $_p$ = ... ;
 asend(Π_p , msg $_p$);
 // ignore buffer problems
 }
 }
}
```

# Distributed Termination II

The termination problem consists of finalizing a program only if applies:

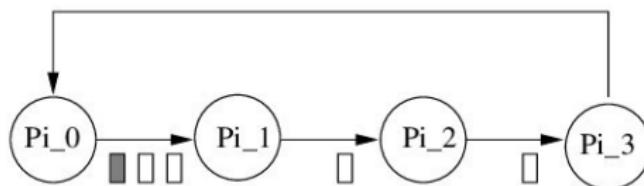
- ① All wait for a message ( are idle )
- ② No messages are underway

Thereby the following assumption are applied regarding the messages:

- ① Ignore problems with buffer overflow
- ② The messages between two processes are processed in the sequence of sending

## 1. variant: termination in the ring

- Token
- Nachricht



## Distributed Termination III

Each process has one of two possible states: red ( active ) or blue ( idle ). For termination recognition a mark is sent around in the ring.

Suppose process  $\Pi_0$  starts the termination process, thus turns first into blue.  
Also suppose,

- ①  $\Pi_0$  is in state blue
- ② mark has arrived at  $\Pi_i$  and  $\Pi_i$  has been recolored into blue

Then we can assume, that the processes  $\Pi_0, \dots, \Pi_i$  are idle and the channels  $(\Pi_0, \Pi_1), \dots, (\Pi_{i-1}, \Pi_i)$  are empty.

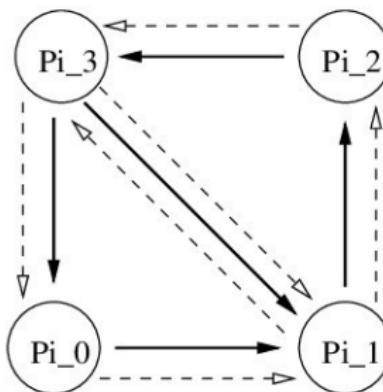
Is the mark again at  $\Pi_0$  and is it still blue ( what it can decide ), then obvious applies:

- ①  $\Pi_0, \dots, \Pi_{P-1}$  are idle
- ② All channels are empty

Then the termination is recognized.

# Distributed Termination IV

## 2. variant: general graph with directed edges



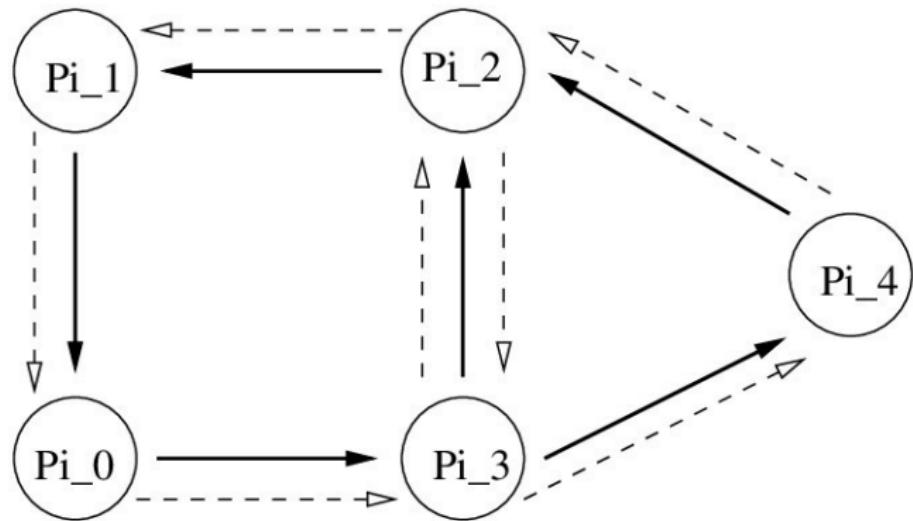
Idea: Over the graph a ring is formed, that includes all nodes, where a node also can be visited more than once.

Algorithm: Choose a path  $\pi = (\Pi_{i_1}, \Pi_{i_2}, \dots, \Pi_{i_n})$  of length n of processes such that applies:

- ① Each edge  $(\Pi_p, \Pi_q) \in E$  exists at least in the path once
- ② A sequence  $(\Pi_p, \Pi_q, \Pi_r)$  exists at most once in the path. Does one reach q from p, then it goes always further to r. r therefore depends on  $\Pi_p$  und  $\Pi_q$  ab:  $r = r(\Pi_p, \Pi_q)$

# Distributed Termination V

Example with  $\pi = (\Pi_0, \Pi_3, \Pi_4, \Pi_2, \Pi_3, \Pi_2, \Pi_1, \Pi_0)$ .



# Distributed Termination VI

```
process Π [int $i \in \{0, \dots, P - 1\}$]
{
 int color = red , token;
 if ($\Pi_i == \Pi_{i_1}$)
 {
 // initialisation of the token
 color = blue;
 token = 0 ,
 asend(Π_{i_2} , TOKEN, token)
 }
 while(1)
 {
 recv_any(who,tag,msg);
 if (tag != TOKEN) { color = red; calculate further }
 else // msg = Token
 {
 if (msg == n) { break; „yeah, ready! “}
 if (color == red)
 {
 color = blue ;
 token = 0 ;
 rcvd = who ;
 }
 else
 if (who == rcvd) token++ ; // a full cycle
 asend($\Pi_{r(who,\Pi_i)}$, TOKEN , token);
 }
 }
}
```

# Distributed Philosophers I

We consider the philosophers problem again, but now with message passing.

- Let a mark circle in the ring. Only who has the mark, may eventually eat.
- State transitions are told to the neighbors, **before** the mark is passed further.
- Each philosopher  $P_i$  is assigned a server  $W_i$ , that performs the state manipulation.
- We use only synchronous communication

```
process P_i [int $i \in \{0, \dots, P - 1\}$]
```

```
{
 while (1) {
 think;
 send(W_i , HUNGRY);
 recv(W_i , msg);
 eat;
 send(W_i , THINK);
 }
}
```

# Distributed Philosophers II

```
process W_i [int $i \in \{0, \dots, P - 1\}$]
{
 int $L = (i + 1) \% P$;
 int $R = (i + p - 1) \% P$;
 int $state = stateL = stateR = THINK$;
 int $stateTemp$;
 if ($i == 0$) send(W_L , TOKEN);
 while (1) {
 recv_any(who, tag);
 if ($who == P_i$) $stateTemp = tag$; // my philosopher
 if ($who == W_L \& \& tag \neq TOKEN$) $stateL = tag$; // state change
 if ($who == W_R \& \& tag \neq TOKEN$) $stateR = tag$; // in neighbor
 if ($tag == TOKEN$){
 if ($state \neq EAT \& \& stateTemp == HUNGRY$
 & & $stateL == THINK \& \& stateR == THINK$){
 $state = EAT$;
 send(W_i , EAT);
 send(W_R , EAT);
 send(P_i , EAT);
 }
 if ($state == EAT \& \& stateTemp == THINK$){
 $state = THINK$;
 send(W_L , THINK);
 send(W_R , THINK);
 }
 send(W_L , TOKEN);
 }
 }
}
```

# Distributed-Memory Programming Models IV

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Client-Server Paradigm I

- **Server:** Process, that processes in an endless loop requests (tasks) of clients.
- **Client:** Sends in irregular distances requests to a server.

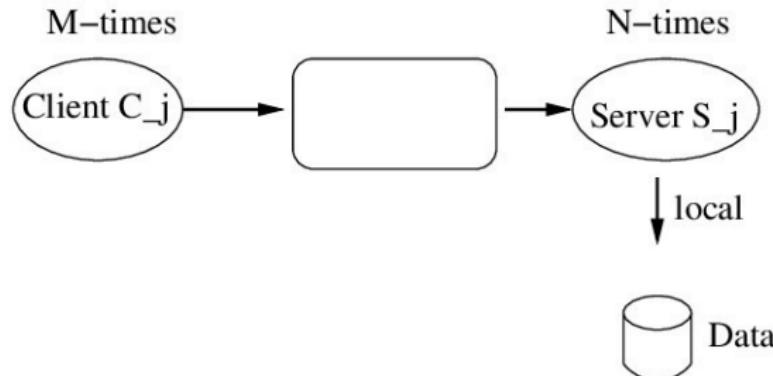
For example the distributed philosophers have been the clients and the servants the servers (that communicate beneath each other).

Practical Examples:

- File Server (NFS: Network File Server)
- Database Server
- HTML Server

Further Example: File Server, Conversational Continuity

Access onto files shall be realized over the network.



# Client-Server Paradigm II

- Client: opens file; performs an arbitrary number of read/write accesses; closes file.
- Server: serves exactly one client, until this closes the file again. Will be released after finalising the communication.
- Allocator: maps a client to a server.

```
process C [int $i \in \{0, \dots, M - 1\}$]
{
 send(A, OPEN , „foo.txt “);
 recv(A , ok , j);
 send(S_j , READ , where);
 recv(S_j , buf);
 send(S_j , WRITE , buf , where);
 recv(S_j , ok);
 send(S_j , CLOSE);
 recv(S_j , ok);
}
```

# Client-Server Paradigm III

```
process A // Allocator
{
 int free [N] = {1[N]} ;
 int cut = 0; // all servers free
 while (1) {
 if (rprobe(who)) { // from whom may I receive?
 if (who ∈ {C0, ..., CM-1} & & cut == N)
 continue; // no servers free
 recv(who , tag , msg);
 if (tag == OPEN){
 Find free server j ;
 free [j] = 0 ;
 cut++;
 send(Sj , tag , msg , who);
 recv(Sj , ok);
 send(who , ok , j);
 }
 if (tag == CLOSE)
 for (j ∈ {0, ..., N - 1})
 if (Sj == who) {
 free [j] = 1;
 cut = cut - 1 ;
 }
 }
}
```

# Client-Server Paradigm IV

```
process S [int j ∈ {0, . . . , N − 1}]
{
 while (1) {
 // wait for message of A
 recv(A , tag , msg , C); // my client
 if (tag ≠ OPEN) → error;
 open file msg
 send(A , ok);
 while (1) {
 recv(C , tag , msg);
 if (tag == READ) {
 ...
 send(C , buf);
 }
 if (tag == WRITE) {
 ...
 send(C , ok);
 }
 if (tag == CLOSE) {
 close file;
 send(C , ok);
 send(A , CLOSE , dummy);
 break;
 }
 }
 }
}
```

# Remote Procedure Call I

- Is abbreviated with RPC ( Remote Procedure Call ). A process calls a procedure/function of another process.

- $\Pi_1$ :

```
:
y = Square(x);
:
:
```

- $\Pi_2$ :

```
int Square(int x)
{
 return x · x;
}
```

- It applies thereby:

- ▶ The processes can run on distinct (remote) processors.
- ▶ The caller blocks as long as the results have not arrived.
- ▶ A two-way communication is established, this means arguments are sent forth and results are sent back. For the client-server paradigm this is the ideal configuration.
- ▶ Many clients can call a remote procedure at a time.

# Remote Procedure Call II

- We realise the RPC by assigning the key word `remote` to the procedure of interest. These can then be called by other processes.

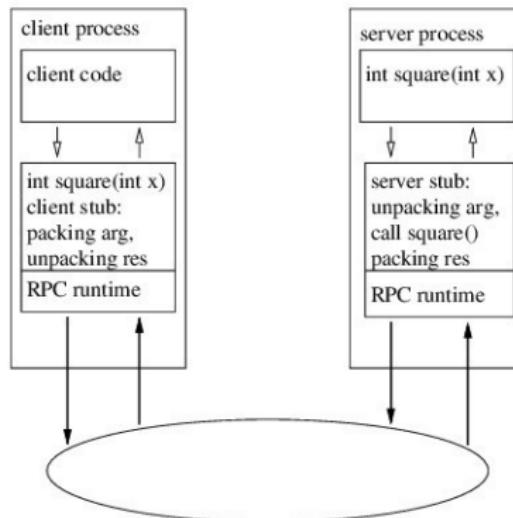
## Program (RPC-Syntax)

```
parallel rpc-example
{
 process Server
 {
 remote int Square(int x)
 {
 return x · x;
 }
 remote long Time (void)
 {
 return time_of_day;
 }
 ... initialisation code
 }
 process Client
 {
 y = Server.Square(5);
 }
}
```

# Remote Procedure Call III

During a call of a function in another process via RPC the following happens:

- The arguments are packed on the caller side into a message, sent across the network and unpacked on the other side.
- Now the function can be called completely normal.
- The return value of the function is sent back to the caller in the same kind.



## Remote Procedure Call IV

A quite frequently used implementation of RPC comes from the company SUN. The most important properties are:

- Portability (client/server applications on different architectures). This means, that the arguments and return values have to be transported in a architecture-independent representation over the network. This is performed by the XDR library (external data representation).
- Few knowledge about network programming is necessary.

We now realize step by step the example from above via SUN's RPC.

# Client-Server Paradigm with RPC I

- (1) Construct a RPC specification in file square.x

```
struct square_in { /* first argument */
 int arg1;
} ;

struct square_out { /* return value */
 int res1;
} ;

program SQUARE_PROG {
 version SQUARE_VERS { /* procedure number */
 square_out SQUAREPROC(square_in) = 1;
 } = 1; /* version number */
} = 0x31230000; /* program number */
```

- (2) Compile the description with the command

```
rpcgen -C square.x
```

# Client-Server Paradigm with RPC II

generates the following 4 files in a **completely automatic** way:

square.h: data types for arguments, procedure heads (cutout)

```
#define SQUAREPROC 1
extern square_out * squareproc_1(square_in *, CLIENT *); /* die ruft Client */
extern square_out * squareproc_1_svc(square_in *, struct svc_req *); /* Server */
```

square\_clnt.c: client side of the function, packing of arguments

```
#include <memory.h> /* for memset */
#include "square.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

square_out * squareproc_1(square_in *argp, CLIENT *clnt)
{
 static square_out clnt_res;

 memset((char *)&clnt_res, 0, sizeof(clnt_res));
 if (clnt_call (clnt, SQUAREPROC,
 (xdrproc_t) xdr_square_in, (caddr_t) argp,
 (xdrproc_t) xdr_square_out, (caddr_t) &clnt_res,
 TIMEOUT) != RPC_SUCCESS) {
 return (NULL);
 }
 return (&clnt_res);
}
```

# Client-Server Paradigm with RPC III

square\_svc.c: Complete server, that reacts on the procedure call.  
square\_xdr.c: Function for data conversion in a heterogeneous environment:

```
#include "square.h"

bool_t xdr_square_in (XDR *xdrs, square_in *objp)
{
 register int32_t *buf;

 if (!xdr_int (xdrs, &objp->arg1))
 return FALSE;
 return TRUE;
}

bool_t xdr_square_out (XDR *xdrs, square_out *objp)
{
 register int32_t *buf;

 if (!xdr_int (xdrs, &objp->res1))
 return FALSE;
 return TRUE;
}
```

# Client-Server Paradigm with RPC IV

- (3) Now the client needs to be written, that calls the procedure.

(client.c):

```
#include "square.h" /* includes also rpc/rpc.h */
int main (int argc, char **argv)
{
 CLIENT *cl;
 square_in in;
 square_out *outp; /* can only return a pointer */

 if (argc!=3) {
 printf("usage: client <hostname> <integer-value>\n");
 exit(1);
 }

 cl = clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");
 if (cl==NULL) {
 printf("clnt_create failed\n");
 exit(1);
 }
 in.arg1 = atoi(argv[2]);
 outp = squareproc_1(&in,cl); /* remote procedure call */
 if (outp==NULL) {
 printf("%s",clnt_sperror(cl,argv[1]));
 exit(1);
 }

 printf("%d\n",outp->res1);
 exit(0);
}
```

# Client-Server Paradigm with RPC V

- (4) Now the client can be build:

```
gcc -g -c client.c
gcc -g -c square_xdr.c
gcc -g -c square_clnt.c
gcc -o client client.o square_xdr.o square_clnt.o
```

- (5) Finally the function on the server side has to be written (`server.c`):

```
square_out * squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
{
 static square_out out; /* since we return pointers */

 out.res1 = inp->arg1 * inp->arg1;
 return (&out);
}
```

- (6) Now the server can be build:

```
gcc -g -c server.c
gcc -g -c square_xdr.c
gcc -g -c square_svc.c
gcc -o server server.o square_xdr.o square_svc.o
```

# Client-Server Paradigm with RPC VI

- (7) Starting of the processes works as follows:

Test, whether the portmapper runs: `rpcinfo -p`

Start server via `server &`

Start client:

```
josh> client troll 123
15129
```

By default the server answers the request sequentially after each other. A multi-threaded server is created as follows:

- generate RPC code via `rpcgen -C -M ...`
- make the procedures reentrant. Trick with `static` variables does not work anymore. Solution: Pass the result back in a call-by-value parameter.

# Client-Server Paradigm: CORBA I

Example works with MICO (<http://www.mico.org>), a free CORBA implementation (C++), that has been developed at the university of Frankfurt.

- (1) IDL definition of the class account.idl:

```
interface Account {
 void deposit(in unsigned long amount);
 void withdraw(in unsigned long amount);
 long balance();
};
```

- (2) Automatic generation of client/server classes

```
idl account.idl
```

generates the files account.h (class definitions) and account.cc (implementation of the client side).

# Client-Server Paradigm: CORBA II

- (3) Call of the client side: `client.cc`

```
#include <CORBA-SMALL.h>
#include <iostream.h>
#include <fstream.h>
#include "account.h"

int main(int argc, char *argv[])
{
 // ORB initialization
 CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "mico-local-orb");
 CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

 // read stringified object reference
 ifstream in ("account.objid");
 char ref[1000];
 in >> ref;
 in.close();

 // client side
 CORBA::Object_var obj = orb->string_to_object(ref);
 assert (!CORBA::is_nil (obj));
 Account_var client = Account::_narrow(obj);

 client->deposit(100);
 client->deposit(100);
 client->deposit(100);
 client->deposit(100);
 client->deposit(100);
 client->withdraw(240);
 client->withdraw(10);
 cout << "Balance is " << client->balance() << endl;

 return 0;
}
```

# Client-Server Paradigm: CORBA III

- (4) Server contains the implementation of the class, generates the objects and the server itself: `server.cc`:

```
#define MICO_CONF_IMR
#include <CORBA-SMALL.h>
#include <iostream.h>
#include <fstream.h>
#include <unistd.h>
#include "account.h"

class Account_impl : virtual public Account_skel {
 CORBA::Long _current_balance;
public:
 Account_impl ()
 {
 _current_balance = 0;
 }
 void deposit(CORBA::ULong amount)
 {
 _current_balance += amount;
 }
 void withdraw(CORBA::ULong amount)
 {
 _current_balance -= amount;
 }
 CORBA::Long balance()
 {
 return _current_balance;
 }
}
```

# Client-Server Paradigm: CORBA IV

```
int main(int argc, char *argv[])
{
 cout << "server init" << endl;

 // initialize CORBA
 CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "mico-local-orb");
 CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

 // create object, produce global reference
 Account_impl *server = new Account_impl;
 CORBA::String_var ref = orb->object_to_string(server);
 ofstream out ("account.objid");
 out << ref << endl;
 out.close();

 // start server
 boa->impl_is_ready(CORBA::ImplementationDef::_nil());
 orb->run ();

 CORBA::release(server);
 return 0;
}
```

## Client-Server Paradigm: CORBA V

To start the server is run again: `server &`

And the client is called:

```
josh > client
Balance is 250
josh > client
Balance is 500
josh > client
Balance is 750
```

Object naming: Here over a „stringified object reference“. Exchange over shared readable file, email, etc. Is global unique and contains IP numbers, server process, object.

Alternatively: Separate naming services.

## Some innovative aspects of MPI-2

- Dynamic process creation and management
- Communicators: Inter- and Intracomunicators
- MPI and Threads
- One-sided communication

# MPI-2 Process Control

- MPI-1 specifies neither how the processes are spawned nor how they create a communication infrastructure
- MPI-2 enables dynamic creation of processes
  - ▶ `MPI_Comm_spawn()` starts MPI processes and creates a communication infrastructure
  - ▶ `MPI_Comm_spawn_multiple()` starts binary-distinct programs or the same program with different arguments below the same communicator `MPI_COMM_WORLD`
- MPI uses the existing group abstractions to represent processes. A (group,rank) pair identifies a process in a unique way. A process determines a unique (group,rank) pair, since it may be part of several groups.
- MPI does not provide any operating system services, e.g. starting and stopping of processes , and therefore implies implicitly the existence of a runtime environment, within which a MPI-application can run.
- The newly created child processes possess their own communicator `MPI_COMM_WORLD`. With `int MPI_Comm_get_parent(MPI_Comm *parent)` you receive the same intercommunicator, that the parent processes have received during their creation.

# MPI-2 Process Control

Interface to create new processes during runtime

- Syntax:

```
int MPI_Comm_spawn(command, argv, maxprocs, info,
root, comm, intercomm, errorcodes)
```

- int MPI\_Comm\_spawn () is a collective function. First if all child processes have called MPI\_Init () it is finished.
- Arguments are specified in the following:

| argument type    | name          | description                                                                                                                |
|------------------|---------------|----------------------------------------------------------------------------------------------------------------------------|
| char * (IN)      | command       | name of the program to be created (only root)                                                                              |
| char * (IN)      | argv          | arguments for command (only root)                                                                                          |
| int (IN)         | maxprocs      | maximal count of processes to be created                                                                                   |
| MPI_Info (IN)    | info          | a set of key-value pairs, that provides the runtime system info, where and how the processes are to be created (only root) |
| int (IN)         | root          | the rank of the process in which argv is evaluated                                                                         |
| MPI_Comm (IN)    | comm          | Intracommunicator for generated processes                                                                                  |
| MPI_Comm * (OUT) | intercomm     | Intercommunicator between original group and newly generated group                                                         |
| int (OUT)        | errorcodes [] | A code per process                                                                                                         |

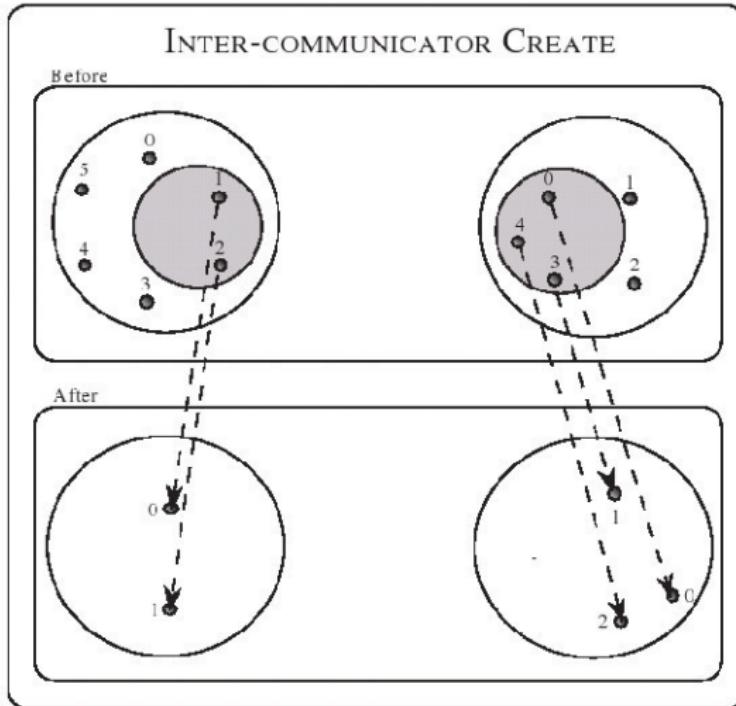
# MPI-2 Enhanced Shared Communication

- MPI-1: shared communication operations for **intracomunicators**, only `MPI_Intercomm_create()` and `MPI_Comm_dup()` to create **intercommunicators**
- MPI-2: extension of many MPI-1 communication operations to intercommunicators, further possibilities to create intercommunicators, 2 new routines for shared communication.

constructors for intercommunicators:

- `MPI::Intercomm MPI::Intercomm::Create(const Group& group) const`  
`MPI::Intracomm MPI::Intracomm::Create(const Group& group) const`

# MPI-2: Intercommunicator Construction



from MPI-2 standard document

# MPI-2: Collective Communication inside Intercommunicator

- **All-To-All**

- ▶ `MPI_Allgather, MPI_Allgatherv`
- ▶ `MPI_Alltoall, MPI_Alltoallv`
- ▶ `MPI_Allreduce, MPI_Reduce_scatter`

- **All-To-One**

- ▶ `MPI_Gather, MPI_Gatherv`
- ▶ `MPI_Reduce`

- **One-To-All**

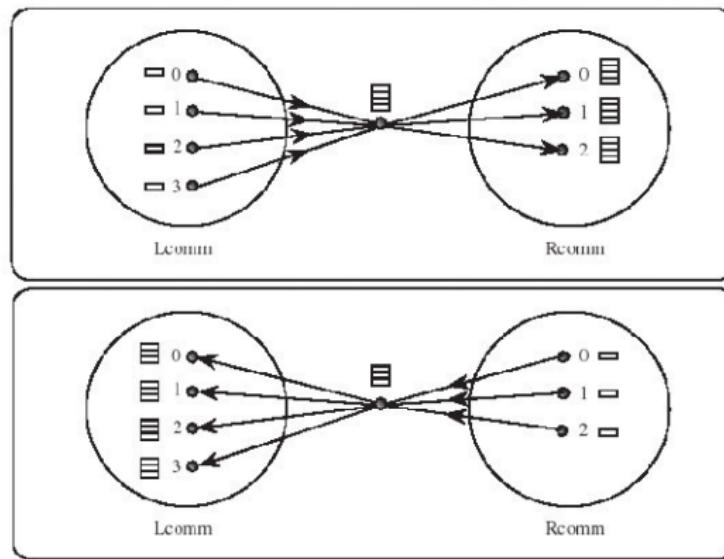
- ▶ `MPI_Bcast`
- ▶ `MPI_Scatter, MPI_Scatterv`

- **Other**

- ▶ `MPI_Scan`
- ▶ `MPI_Barrier`

# MPI-2: Collective Communication in Intercommunicator

- Description of operations with source and target group.
  - ▶ within intracomunicators these groups are identical
  - ▶ within intercommunicators these groups are distinct
- Messages and data flow within MPI\_Allgather()



from MPI-2 standard document

# MPI-2: Collective Communication in the Intracommunicator

Generalised Alltoall function (w) (we already known this one!)

- Declaration:

```
void MPI::Comm::Alltoallw (const void* sendbuf, const int sendcounts[], const
int sdispls[], const MPI::Datatype sendtype[], void *recvbuf, const int
recvcounts[], const int rdispls[], const MPI::Datatype recvtypes[]) const =
0;
```

- The j-th block that sends process i is stored by process j in the i-th block of recvbuf.
- The blocks can have different size
- Type signatures and data extend have to be consistent:  
 $\text{sendcounts[j]}, \text{sendtypes[j]}$  of process i fits to  
 $\text{sendcounts[i]}, \text{sendtypes[i]}$  of process j
- No in-place option

# MPI-2: Collective Communication in the Intracommunicator

Exclusive scan operation, inclusive scan already in MPI-1

- Declaration:

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

- Performs a prefix reduction on data, that are distributed across the group
- Value in `recvbuf` of process 0 is undefined
- Value in `recvbuf` of process 1 is defined by the value of `sendbuf` of process 0
- Value in `recvbuf` of process  $i$  with  $i > 1$  is the value of reduction operation `op` applied to the `sendbufs` of processes  $0, \dots, i - 1$
- no in-place option

# Hybrid Programming: MPI and Threads I

## Basic Assumptions

- Thread library according to POSIX standard
- MPI process can be run multithreaded without limitations
- Each thread can call MPI functions
- Threads of an MPI process can not be distinguished  
rank specifies a MPI process not thread
- The user has to avoid conditions, that can be generated by  
contradictionary communication calls  
This can e.g. occur by thread specific communicators

## Minimal requirements for thread-aware MPI

- All MPI calls are thread save, this means two concurrent threads may execute MPI calls, the result is invariant concerning the call sequence, also by interleaving of the calls in time
- Blocking MPI calls block only the calling thread, while further threads can be active, especially these may execute MPI calls.
- MPI calls can be made thread save when one only executes one call at a time. This can be performed with one MPI process with individual lock.

# Hybrid Programming: MPI and Threads II

- `MPI_Init()` and `MPI_Finalize()` should be called by the same thread, so called main thread
- Initialisation of MPI and thread environment with

```
int MPI::Init_thread (int& argc, char **& argv, int required)
```

The argument `required` specifies a necessary thread level

- ▶ `MPI_THREAD_SINGLE`: only a thread will be executed
  - ▶ `MPI_THREAD_FUNNELED`: the process can be multi-threaded, MPI calls are performed only by the main thread
  - ▶ `MPI_THREAD_SERIALIZED`: the process can be multi-threaded and several threads may execute MPI calls, but at each point in time only one (thus no concurrency of MPI calls)
  - ▶ `MPI_THREAD_MULTIPLE`: Several threads may call MPI without constraints
- The user has to ensure the correspondence of MPI collective operations on a communicator via interthread synchronisation
  - It is not guaranteed, that the exception handling is done by the same thread, that has executed the MPI call causing the exception.
  - Request of the current thread level with `int MPI::Query_thread()` determination whether main thread `bool MPI::Is_thread_main()`

# Evaluation of Parallel Algorithms

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Themes

## Evaluation of parallel algorithms

- Speedup, efficiency
- Degree of parallelism, costs
- Iso-efficiency
- Amdahl's law
- Gustafson scaling
- Scalability

# Evaluation of Parallel Algorithms I

How can the properties of a parallel algorithm for the solution of a problem  $\Pi(N)$  be analyzed?

- Problem size  $N$  can be chosen arbitrary
- Solution of the problem with sequential resp. parallel algorithm
- Hardware assumptions:
  - ▶ MIMD parallel computer with  $P$  identical computing nodes
  - ▶ Communication network scales with number of computing nodes
  - ▶ Latency, band width and node performance are known
- Execution of the sequential program on a single node
- Parallel algorithm + parallel implementation + parallel hardware = parallel system
- The notion of scalability characterizes the ability of a parallel system to handle increasing resources provided by processors  $P$  or demands by the problem size  $N$ .

Goal: Analysis of scalability properties of a parallel system

# Evaluation of Parallel Algorithms II

## Measures for Parallel Algorithms

- Runtime
- Speedup and Efficiency
- Costs
- Degree of parallelism

# Evaluation of Parallel Algorithms III

Definition of different execution times:

- The **sequential execution time**  $T_S(N)$  denotes the runtime of a sequential algorithm for solution of problem  $\Pi$  for input size  $N$ .
- The **optimal execution time**  $T_{best}(N)$  characterizes the runtime of the best (existing) sequential algorithm to solve the problem  $\Pi$  with input size  $N$ . This algorithm has for nearly all sizes of  $N$  the lowest time demands.
- The **parallel runtime**  $T_P(N, P)$  describes the runtime of the parallel systems, to be investigated, for solution of  $\Pi$  in dependence of input size  $N$  and the processor count  $P$ .

# Evaluation of Parallel Algorithms IV

The measurement of runtimes allows the definition of further units:

- **Speedup**

$$S(N, P) = \frac{T_{best}(N)}{T_P(N, P)}. \quad (1)$$

For all  $N$  and  $P$  applies  $S(N, P) \leq P$ .

Assume that holds  $S(N, P) > P$ , then exists a sequential program, that simulates the parallel program (processing in time slices mode). This hypothetical program would then have a runtime  $PT_P(N, P)$  and it would be

$$PT_P(N, P) = P \frac{T_{best}(N)}{S(N, P)} < T_{best}(N), \quad (2)$$

In obvious contradiction to former definitions.

- **Efficiency**

$$E(N, P) = \frac{T_{best}(N)}{PT_P(N, P)}. \quad (3)$$

It applies  $E(N, P) \leq 1$ . The efficiency represents the share of the maximal achievable speedup. We say that  $E \cdot P$  processors really work for the solution of  $\Pi$  and the rest  $(1 - E)P$  does not contribute effectively to the problem solution.

# Evaluation of Parallel Algorithms V

## • Costs

As costs  $C$  the product

$$C(N, P) = PT_P(N, P), \quad (4)$$

is defined, since one would have to pay this duration of computing time in the computing center.

We denote an algorithm as *cost optimal*, if  $C(N, P) = const T_{best}(N)$ . Obviously applies then

$$E(N, P) = \frac{T_{best}(N)}{C(N, P)} = 1/const, \quad (5)$$

the efficiency remains thus constant.

# Evaluation of Parallel Algorithms VI

- **Degree of parallelism**

With  $\Gamma(N)$  we denote the *degree of parallelism*. That is the maximal number of operations, that can be executed synchronously, in the best sequential algorithm.

- ▶ Obviously could be in principle executed the more operations the more operations had to be executed overall, thus the larger  $N$  is. The degree of parallelism is such dependent on  $N$ .
- ▶ On the other side the degree of parallelism can not be larger than the number of operations that have to be executed in total. Since this number is proportional to  $T_S(N)$ , we can say, that

$$\Gamma(N) \leq O(T_S(N)) \tag{6}$$

holds.

# Evaluation of Parallel Algorithms: Speedup

Elementary is the behaviour of the speedup  $S(N, P)$  of a parallel system in dependence of  $P$ .

With the second parameter  $N$  we have the choice of different scenarios.

## 1. Fixed sequential execution time

- We determine  $N$  from the relation

$$T_{best}(N) \stackrel{!}{=} T_{fix} \rightarrow N = N_A \quad (7)$$

where  $T_{fix}$  is a parameter. The speedup is then

$$S_A(P) = S(N_A, P), \quad (8)$$

therefore  $A$  stands for the name *Amdahl*.

- How behaves the scaled speedup?

Assumption: the parallel program is created from the best sequential program with sequential share  $0 < q < 1$  and a completely parallelisable rest ( $1 - q$ ). The parallel runtime (for fixed  $N_A$ !) is then

$$T_P = qT_{fix} + (1 - q)T_{fix}/P. \quad (9)$$

# Evaluation of Parallel Algorithms: Amdahl

For the speedup applies then

$$S(P) = \frac{T_{fix}}{qT_{fix} + (1 - q)T_{fix}/P} = \frac{1}{q + \frac{1-q}{P}} \quad (10)$$

Thus the Amdahl's law holds

$$\lim_{P \rightarrow \infty} S(P) = 1/q. \quad (11)$$

## Consequences:

- The maximal achievable speedup is then determined purely by the sequential share.
- The efficiency strongly decreases, if one nearly wants to reach the maximal speedup.
- This achievement led at the end of the 60th to a very pessimistic estimation of the possibilities by parallel computing.
- This has changed first, when it has been recognized, that for most parallel algorithms the sequential share  $q$  *decreases* with increasing  $N$ .

The way out of this dilemma consists in solving with more processors always larger problems!

We now present three approaches how  $N$  can be increased with  $P$ .

# Evaluation of Parallel Algorithms: Gustafson

## 2. Fixed parallel execution time

We determine  $N$  from the equation

$$T_P(N, P) \stackrel{!}{=} T_{fix} \rightarrow N = N_G(P) \quad (12)$$

for given  $T_{fix}$  and then consider the speedup

$$S_G(P) = S(N_G(P), P). \quad (13)$$

- This kind of scaling is also called „Gustafson scaling“.
- Motivation are for example applications in the area of weather forecast.  
Here one has a fixed time slot  $T_{fix}$  that is used to solve a problem as large as possible.

# Evaluation of Parallel Algorithms: Memory Limitation

## 3. Fixed memory consumption per processor

Many simulation applications are memory constraint, the memory need grows as function  $M(N)$ . According to memory complexity not computing time since the memory needs determine what problems can be calculated with a machine.

Assumption: Let us assume, that the parallel computer consists of  $P$  identical processors, that each have memory of size  $M_0$ , thus the scaling provides

$$M(N) \stackrel{!}{=} PM_0 \rightarrow N = N_M(P) \quad (14)$$

and we consider

$$S_M(P) = S(N_M(P), P). \quad (15)$$

as scaled speedup.

# Evaluation of Parallel Algorithms: Efficiency Limitation

## 4. Constant Efficiency

We choose  $N$  such, that the parallel efficiency remains constant.

We require

$$E(N, P) \stackrel{!}{=} E_0 \rightarrow N = N_l(P). \quad (16)$$

This is denoted as *iso-efficient scaling*. Obviously is  $E(N_l(P), P) = E_0$  thus

$$S_l(P) = S(N_l(P), P) = PE_0. \quad (17)$$

An iso-efficient scaling is not possible for each parallel system. One does not necessarily find a function  $N_l(P)$ , that fulfills (16) identical. Thus one can require on the other side, that a system is scalable exactly if such a function can be found.

# Evaluation of Parallel Algorithms: Example I

For a deeper understanding of the notions we now consider an **example**

- We want to add  $N$  numbers on a hypercube with  $P$  processors. The approach is as follows:
  - ▶ Each has  $N/P$  numbers, that are added in the first step.
  - ▶ These  $P$  intermediate results are then added in a tree.
- We then get for the sequential computing time

$$T_{best}(N) = (N - 1)t_a \quad (18)$$

- The parallel computing time is

$$T_P(N, P) = (N/P - 1)t_a + \text{Id } Pt_m, \quad (19)$$

where  $t_a$  is the time for the addition of two numbers and  $t_m$  the time for the message exchange (we assume, that  $t_m \gg t_a$ ).

# Evaluation of Parallel Algorithms: Example II

## 1. Fixed sequential execution time (Amdahl)

If we set  $T_{best}(N) = T_{fix}$  then we get, if  $T_{fix} \gg t_a$ , in good approximation

$$N_A = T_{fix}/t_a.$$

For meaningful processor counts  $P$  applies:  $P \leq N_A$ .

For the speedup we obtain in the case of  $N_A/P \gg 1$

$$S_A(P) = \frac{T_{fix}}{T_{fix}/P + \text{ld } Pt_m} = \frac{P}{1 + P \text{ld } P \frac{t_m}{T_{fix}}}. \quad (20)$$

## 2. Fixed parallel execution time (Gustafson)

Here one obtains

$$\left(\frac{N}{P} - 1\right) t_a + \text{ld } Pt_m = T_{fix} \implies N_G = P \left(1 + \frac{T_{fix} - \text{ld } Pt_m}{t_a}\right). \quad (21)$$

The maximal usable processor count is again limited:  $2^{T_{fix}/t_m}$ . Is  $\text{ld } Pt_m = T_{fix}$ , then when using more processors than that in every case the maximal allowed computing time is exceeded.

Despite that we can suppose, that  $2^{T_{fix}/t_m} \gg T_{fix}/t_a$  holds.

## Evaluation of Parallel Algorithms: Example III

The scaled speedup  $S_G$  is under the assumption  $N_G(P)/P \gg 1$ :

$$S_G(P) = \frac{N_G(P)t_a}{N_G(P)t_a/P + \text{ld } Pt_m} = \frac{P}{1 + \text{ld } P \frac{t_m}{T_{fix}}}. \quad (22)$$

It applies  $N_G(P) \approx PT_{fix}/t_a$ .

For the same processor count is then  $S_G$  greater than  $S_A$ .

### 3. Fixed memory per processor (memory limitation)

If the memory demands are  $M(N) = N$ , then applies for  $M(N) = M_0 P$  the scaling

$$N_M(P) = M_0 P.$$

We can now use an unlimited number of processors, on the other hand the parallel computing time increases also unlimited. For the scaled speedup we get:

$$S_M(P) = \frac{N_M(P)t_a}{N_M(P)t_a/P + \text{ld } Pt_m} = \frac{P}{1 + \text{ld } P \frac{t_m}{M_0 t_a}}. \quad (23)$$

For the choice  $T_{fix} = M_0 t_a$  this is the same formula as  $S_G$ . In both cases we see, that the efficiency decreases with  $P$ .

# Evaluation of Parallel Algorithms: Example IV

## 4. Iso-efficient scaling

We choose  $N$  such, that the efficiency remains constant, resp. the speedup grows linearly:

$$S = \frac{P}{1 + \frac{P \text{Id} P}{N} \frac{t_m}{t_a}} \stackrel{!}{=} \frac{P}{1 + K} \implies N_l(P) = P \text{Id} P \frac{t_m}{K t_a},$$

for an arbitrary choosable  $K > 0$ . Since  $N_l(P)$  exists, the algorithms can be regarded as scalable. For the speedup applies  $S_l = P/(1 + K)$ .

# Iso-efficiency Analysis I

We now introduce further a formalism to clarify the principle of iso-efficient scaling

- Goal answering of questions:  
„Is this algorithm for matrix multiplication on the hypercube better scalable than that for fast fourier transform on the array topology“
- Problem size: Parameter  $N$  has been chosen up to now arbitrary.
- $N$  can denote in matrix multiplication either the number of matrix elements or too the number of elements per row.
- In this situation the first case would lead to  $2N^{3/2}t_f$ , whilst the second case  $2N^3t_f$  for the sequential runtime.
- Meaningful comparison of algorithms necessitates invariance of the cost measure regarding the choice of the parameter for the problem size.

## Iso-efficiency Analysis II

- We choose as measure for the costs  $W$  of a (sequential) algorithm its execution time, we therefore define

$$W = T_{best}(N) \quad (24)$$

itself. This execution time is furthermore proportional to the number of operations to be executed in the algorithms.

- For the degree of parallelism  $\Gamma$  we obtain:

$$\Gamma(W) \leq O(W),$$

since there can not be executed more operations in parallel as there are operations in total.

- Via  $N = T_{best}^{-1}(W)$  we can write

$$\tilde{T}_P(W, P) = T_P(T_{best}^{-1}(W), P),$$

where we however leave away the  $\sim$  sign in the following.

# Iso-efficiency Analysis III

We define the *overhead* as

$$T_o(W, P) = PT_P(W, P) - W \geq 0. \quad (25)$$

$PT_P(W, P)$  is the time, that a simulation of the parallel program would need on one processor. This is in every case not smaller than the best sequential execution time  $W$ . The overhead contains additional computing time because of communication, load imbalance and „superfluous“ calculations.

**Iso-efficiency function** From the overhead we obtain

$$T_P(W, P) = \frac{W + T_o(W, P)}{P}.$$

thus we obtain for the speedup

$$S(W, P) = \frac{W}{T_P(W, P)} = P \frac{1}{1 + \frac{T_o(W, P)}{W}},$$

resp. for the efficiency

$$E(W, P) = \frac{1}{1 + \frac{T_o(W, P)}{W}}.$$

In the sense of an iso-efficient scaling we now ask: How needs  $W$  to grow as function of  $P$  that the efficiency remains constant. Because of the formula above this is the case when  $T_o(W, P)/W = K$ , with an arbitrary constant  $K \geq 0$ . The efficiency is then  $1/(1 + K)$ .

## Iso-efficiency Analysis IV

- A function  $W_K(P)$  is called *iso-efficiency function* if it fulfills the equation

$$T_o(W_K(P), P) = KW_K(P)$$

identical.

- A parallel system is called scalable (exactly) iif it has an iso-efficiency function.
- The asymptotic growing of  $W$  with  $P$  is a measure for the scalability of the system:  
Has for example a system  $S_1$  an iso-efficiency function  $W = O(P^{3/2})$  and a system  $S_2$  an iso-efficiency function  $W = O(P^2)$  then  $S_2$  scales *worse* than  $S_1$ .

# Iso-efficiency Analysis V

## When is there an iso-efficiency function?

- We progress from the efficiency

$$E(W, P) = \frac{1}{1 + \frac{T_o(W, P)}{W}}.$$

- for *fixed W* and growing *P*. It holds for each parallel system, that

$$\lim_{P \rightarrow \infty} E(W, P) = 0$$

as can be seen by the following thoughts: Since *W* is fixed, also the degree of parallelism is fixed and then there exists a lower bound for the parallel computing time:  $T_P(W, P) \geq T_{min}(W)$ , this means the calculation can not be faster than  $T_{min}$ , without dependance on the number of used processors. Thus however implies asymptotically

$$\frac{T_o(W, P)}{W} \geq \frac{PT_{min}(W) - W}{W} = O(P)$$

and therefore the efficiency drops against 0.

# Iso-efficiency Analysis VI

If we consider now the efficiency at *fixed P* and growing work *W*, then applies for many (not all!) parallel systems, that

$$\lim_{W \rightarrow \infty} E(W, P) = 1.$$

Obviously this means regarding the efficiency formula, that

$$T_o(W, P)|_{P=const} < O(W) \quad (26)$$

for fixed *P* the overhead grows less than linear with *W*. In this case for each *P* a *W* can be found such that a desired efficiency is achieved. Equation (26) ensures such the existence of an iso-efficiency function. For example the matrix transposition can be encountered as a not scalable system. We will later derive, that the overhead amounts in this case  $T_o(W, P) = O(W \text{Id } P)$ . Such no iso-efficiency function can exist.

## Optimal parallelisable systems

We want to analyze now the question how iso-efficiency functions have to grow at least. For this we remark finally, that

$$T_P(W, P) \geq \frac{W}{\Gamma(W)},$$

since  $\Gamma(W)$  (dimensionless) is the maximal count of operations that can be executed synchronously in the sequential algorithms for effort *W*. Therefore  $W/\Gamma(W)$  is a lower bound for the parallel computing time.

# Iso-efficiency Analysis VII

Now there can surely not be executed more operations in parallel than can be executed in total, thus holds  $\Gamma(W) \leq O(W)$ . We want to denote a system as *optimal parallelisable*, if

$$\Gamma(W) = cW$$

holds with a constant  $c > 0$ . Now applies

$$T_P(W, P) \geq \frac{W}{\Gamma(W)} = \frac{1}{c},$$

the minimal parallel computing time remains constant. For the overhead we obtain that in this case

$$T_o(W, P) = PT_P(W, P) - W = P/c - W$$

and such for the iso-efficiency function

$$T_o(W, P) = P/c - W \stackrel{!}{=} KW \iff W = \frac{P}{(K+1)c} = O(P).$$

Optimal parallelisable systems such have an iso-efficiency function  $W = O(P)$ . We remark thus, that a iso-efficiency function grows at least linear with  $P$ .

In the following lectures we will determine the iso-efficiency functions for a series of algorithms, therefore we relinquish for an extensive example here.

# Fundamentals of Parallel Algorithms

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

- Foundations of parallel algorithms
- Load balancing

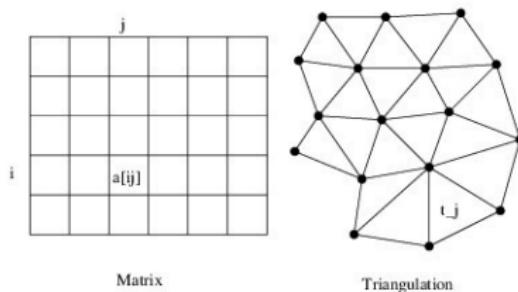
# Foundations of Parallel Algorithms

Parallelisation approaches for the design of parallel algorithms:

- ① Data partitioning: *Subdivide* a problem into independent subtasks. This serves for the identification of the maximal possible parallelism.
- ② Agglomeration: Control of *granularity* to balance computational needs and communication.
- ③ Mapping: Map processes onto processors. Goal is an optimal tuning of the logical communication structure onto the machine structure.

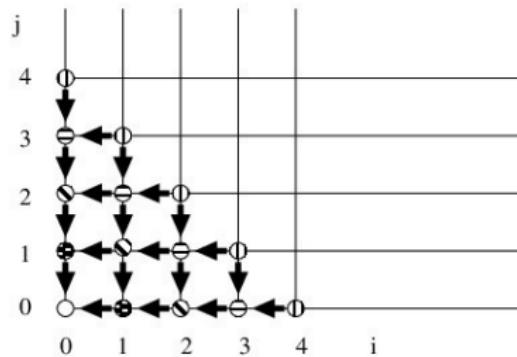
# Foundations of Parallel Algorithms: Data Partitioning

- Calculations are directly associated to specific data structures.
- For each data object certain operations have to be executed, often the same sequence of operations has to be applied onto different data. Thus a part of the data (objects) can be assigned to each process.



- Matrix addition: For  $C = A + B$  elements  $c_{ij}$  can be processed completely in parallel. In this case one would assign each process  $\Pi_{ij}$  the matrix elements  $a_{ij}$ ,  $b_{ij}$  and  $c_{ij}$ .
- Triangulation for the numerical solution of partial differential equations: Here the calculations occur per triangle, that all can be performed at a time, each process could be assigned thus a partial set of the triangles.

# Foundations of Parallel Algorithms: Data Dependencies



Data dependencies in the Gauß–Seidel method.

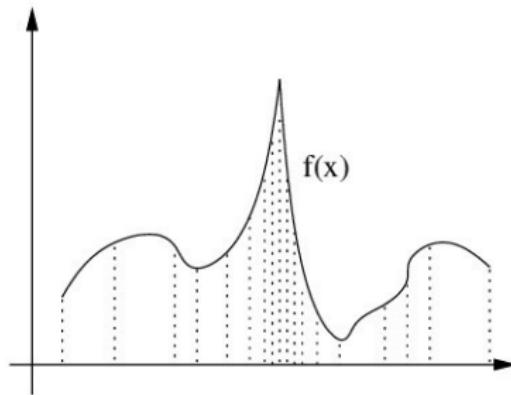
- Operations often can not be performed for all data objects synchronously.
- Example: Gauß-Seidel iteration with lexicographic numbering.  
Calculation at grid point  $(i, j)$  depends on the result of calculations at the grid points  $(i - 1, j)$  and  $(i, j - 1)$ .
- The grid point  $(0, 0)$  can be calculated without any prerequisite.
- All grid points on the diagonal  $i + j = \text{const}$  can be processed in parallel.

# Foundations of Parallel Algorithms: Funct. Partitioning

## Functional partitioning

- For different operations on same data.
- Example compiler: This performs the steps: lexical analysis, parsing, code generation, optimization and assembling. Each step can be assigned to a separate process. („macro pipelining“).

## Irregular problems



- No a-priori partitioning possible.
- Calculation of the integral of a function  $f(x)$  by adaptive quadrature.
- Intervall choice depends on  $f$  and is determined during the calculation.

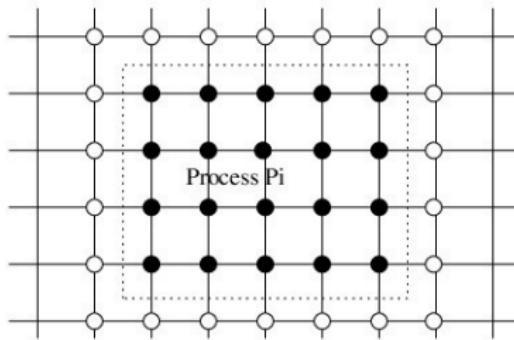
# Foundations of Parallel Algorithms: Agglomeration I

- Partition step determines the maximal possible parallelism.
- Realisation (in the sense of a data object per process) is in most cases not meaningful (communication overhead).
- Agglomeration: Mapping of several subtasks to one process, thus the communication is collected for these subtasks in as less as possible messages.
- Reduction of number of messages to sent, further savings for *data locality*
- As *granularity* of a parallel algorithm we denote the relationship:

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

- Agglomeration reduces also the granularity.

# Foundations of Parallel Algorithms: Agglomeration II



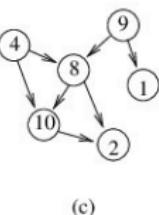
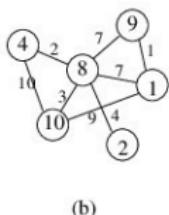
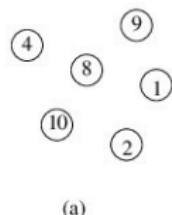
## Grid based calculations

- Calculations are performed for all grid points in parallel.
- Assignment of a set of grid points to a process
- All modifications on grid points can be executed under specific circumstances simultaneously. There are thus no data dependencies.

# Foundations of Parallel Algorithms: Agglomeration III

- A process owns  $N$  grid points and has thus to execute  $O(N)$  computing operations.
- Only for grid points at the boundary of the partition a communication is needed.
- Therefore it is only for in total  $4\sqrt{N}$  grid points communication necessary.
- Relationship of communication needs towards computing needs is thus in the order of  $O(N^{-1/2})$
- Increase of the number of grid points per processor shrinks the needs for communication relatively to the calculations to an arbitrary small size (*surface-to-volume effect*).

How has the agglomeration to be performed?



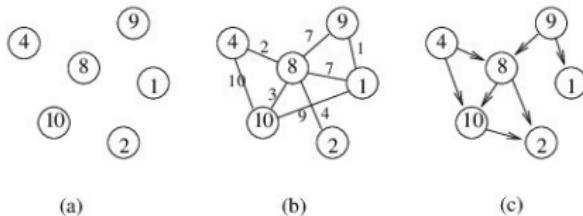
- ➊ Uncoupled calculations
- ➋ Coupled calculations
- ➌ Coupled calculations with time dependency

# Foundations of Parallel Algorithms: Partitioning (a)

## (a) Uncoupled calculations

- Calculation consists of subproblems, that can be calculated completely independent of each other.
  - Each subproblem may need different computing time.
  - Representable as set of nodes with weights. Weights are a measure for the required computing time.
  - Agglomeration is trivial. One assigns the nodes one by one (e.g. ordered by its size or randomly) each to the process, that has the least work (this is the sum of all its node weights).
  - Agglomeration gets more complicated if the number of nodes is only known during the calculation (as with the adaptive quadrature) and/or the node weights are not known a-priori (as e.g. at depth first search).
- Solution by dynamic load balancing
- ▶ central: a process makes the load balancing
  - ▶ decentral: a process gets work of others, that have to much

# Foundations of Parallel Algorithms: Partitioning (b)



## (b) Coupled calculations

- Standard model for static, data local calculations.
- Calculation is described by an undirected graph.
- First a calculation per node is required, whose computing time is modelled by the node weight. Then each node exchanges data with its neighbor nodes.
- Count of data to be sent is proportional to the associated edge weight.
- Uniform graph with constant weights: trivial agglomeration.

# Foundations of Parallel Algorithms: Partitioning

- General graph  $G = (V, E)$  for  $P$  processors:  
node set  $V$  is to partition such, that

$$\bigcup_{i=1}^P V_i = V, \quad V_i \cap V_j = \emptyset, \quad \sum_{v \in V_i} g(v) = \sum_{v \in V} g(v)/|V|$$

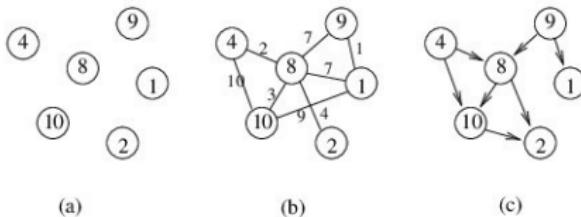
and the separator costs

$$\sum_{(v,w) \in S} g(v, w) \rightarrow \min, \quad S = \{(v, w) \in E | v \in V_i, w \in V_j, i \neq j\}$$

are minimal.

- This problem is denoted as *graph partitioning problem*.
- $\mathcal{NP}$ -complete, already in the case of constant weights and  $P = 2$  (graph bisection)  $\mathcal{NP}$ -complete.
- There exist good heuristics, that generate in linear time (concerning number of nodes,  $O(1)$  neighbors) a (reasonably) well partitioning.

# Foundations of Parallel Algorithms: Partitioning (c)



## (c) Coupled calculations with temporal dependency

- Model is a directed graph.
- A node can only be calculated, if all nodes of ingoing edges are calculated.
- When a node is calculated, the result is passed further over the outgoing edges. The computing time corresponds to the node weights, the communication time correlates to the edge weights.
- In general very difficult to solve problem.
- Theoretically not „more difficult than graph partitioning“ (also  $\mathcal{NP}$ -complete)
- Practically no simple and good heuristics are known.
- For special problems, e.g. adaptive multigrid methods, one can however find good heuristics.

# Foundations of Parallel Algorithms: Mapping

## Mapping: Map the processes on to processors

- Set of processes  $\Pi$  form a undirected graph  $G_\Pi = (\Pi, K)$ : Two processes are connected with each other, if they communicate together (edge weights could model the extent of the communication).
- Likewise the set of processors  $P$  with the communication network forms a graph  $G_P = (P, N)$ : Hypercube, array.
- Be  $|\Pi| = |P|$  and the following question has to be asked:  
Which process shall be executed on which processor?
- In general we want to perform the mapping in such a way, that processes that communicate with each other are mapped onto neighboring or near processors.
- This optimization problem is denoted as *graph assignment problem*
- This is again  $\mathcal{NP}$ -complete (unfortunately).
- Contemporary multi processors possess very powerful communication networks: In cut-through networks the transmission time of a message is practically distance independent.
- The problem of optimal process placement is thus not preferential any more.
- A good process placement is nevertheless important if many processes communicate *synchronously* with each other.

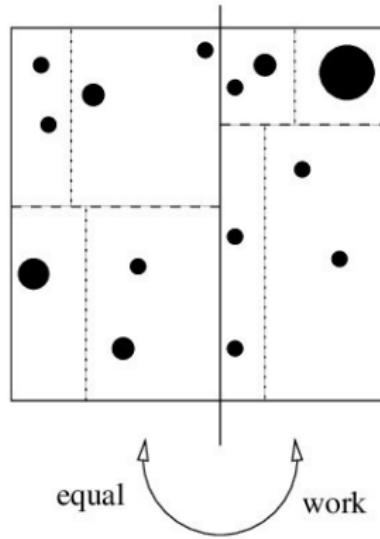
## Situation 1: Static distribution of uncoupled problems

- Task: Partitioning of accumulated work on to the different processors.
- This corresponds to the agglomeration step at which we have again combined the subproblems, that are processable in parallel.
- The measure for the work is hereby known.
- **Bin Packing**
  - ▶ Initially all processors are empty.
  - ▶ Nodes, that are available either in arbitrary sequence or sorted (e.g. corresponding to their size), are packed after each other on the processor that has currently the least work.
  - ▶ This also functions dynamically, if new work is generated in the calculation.

# Foundations of Parallel Algorithms: Load Balancing

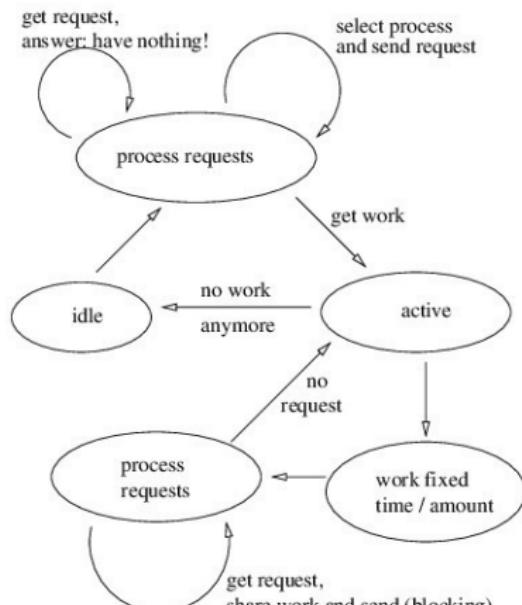
## • Recursive Bisection

- ▶ Each node is assigned a position in space.
- ▶ The space is divided orthogonal to the coordinate system such, that in each part is about the same amount of work.
- ▶ This approach is then recursively applied onto the generated subvolumes with alternating coordinate direction.



# Foundations of Parallel Algorithms: Load Balancing

## Situation 2: Dynamic distribution of uncoupled problems



activity/state diagram

- The measure for the work is not known.
- Process is either active (performs work) or idle (without work).
- During work division the following questions have to be considered:
  - ▶ What do I want to pass away? For travelling-salesman problem e.g. preferably nodes from the stack, that reside far low.
  - ▶ How much do I want to pass away? E.g. half of the work (half split).
- Beside the work sharing further communication can take place (Forming of the global minimum for branch-and-bound during travelling-salesman).
- Furthermore the problem of termination recognition exists.  
When are all processes idle?

# Foundations of Parallel Algorithms: Load Balancing

Which idle process shall be addressed next?

Different selection strategies:

- **Master/Slave (Worker) principle**

A process distributes work. It knows, who is active resp. free and forwards the request. It controls (since it knows, who is idle) also the termination problem. Disadvantage this method does not scale.

Alternatively: hierarchical structure of masters.

- **Asynchrones Round Robin**

The process  $\Pi_i$  has a variable target<sub>i</sub>. It sends its requests to  $\Pi_{\text{target}_i}$  and sets then  $\text{target}_i = (\text{target}_i + 1) \% P$ .

- **Globales Round Robin**

There is only a single global variable target. Advantage: no synchronous request onto the same process. Disadvantage: access on the global variable (what e.g. a server process can do).

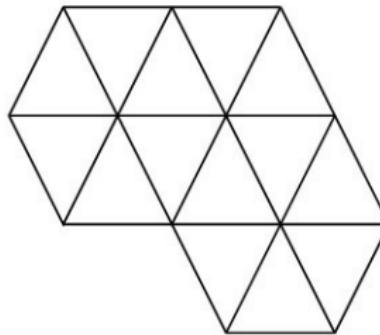
- **Random Polling**

Each chooses randomly a process with same probability ( $\rightarrow$  parallel random generator, watch out at least for the distribution of seeds). This approach provides a uniform distribution of the request and needs no global resource.

# Foundations of Parallel Algorithms: Load Balancing

## Graph partitioning

Consider a Finite-Element mesh:



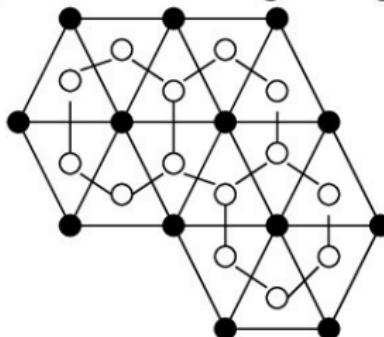
It consists of triangles  $T = \{t_1, \dots, t_N\}$  with

$$\bar{t}_i \cap \bar{t}_j = \begin{cases} \emptyset \\ \text{a node} \\ \text{an edge} \end{cases}$$

- In the method of Finite-Elements the work is associated to the nodes.
- Alternatively one can also put a node in each triangle and connect these with edges over the triangle neighbors. Now the evolving dual graph can be considered.

# Foundations of Parallel Algorithms: Load Balancing

Graph and according dual graph



The partitioning of the graph to processors leads to the graph partitioning problem. Therefore we define the following notation:

$$G = (V, E) \quad \text{(Graph or dual graph)}$$
$$E \subseteq V \times V \quad \text{symmetric (undirected)}$$

The weighting functions

$$w : V \longrightarrow \mathbb{N} \quad \text{(computing demand)}$$
$$w : E \longrightarrow \mathbb{N} \quad \text{(communication)}$$

# Foundations of Parallel Algorithms: Load Balancing

The total work

$$W = \sum_{v \in V} w(v)$$

Furthermore let  $k$  be the number of partitions to be constituted, where holds  $k \in \mathbb{N}$  and  $k \geq 2$ . We seek now the partition mapping

$$\pi : V \longrightarrow \{0, \dots, k - 1\}$$

and the associated edge separator

$$X_\pi := \{(v, v') \in E \mid \pi(v) \neq \pi(v')\} \subseteq E$$

The graph partitioning problem consists of finding the mapping  $\pi$  such that the cost function (communication costs)

$$\sum_{e \in X_\pi} w(e) \rightarrow \min$$

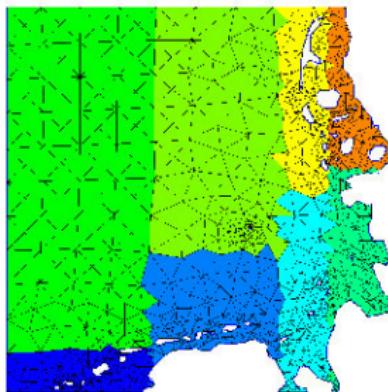
gets minimal under the restriction (equal work balancing)

$$\sum_{v, \pi(v)=i} w(v) \leq \delta \frac{W}{k} \quad \text{for all } i \in \{0, \dots, k - 1\}$$

where  $\delta$  determines the tolerated imbalance ( $\delta = 1.1$  10% deviation).

# Foundations of Parallel Algorithms: Load Balancing

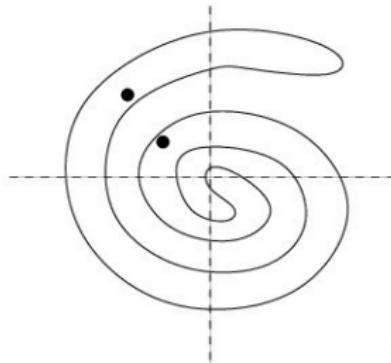
- Calculation costs dominate communication costs. Otherwise the partitioning might eventually be not necessary because of the high communication costs. This is albeit only a model for the run time!
- For binary partitioning one speaks about the graph bisection problem. By recursive bisection  $2^d$ -way partitionings can be generated.
- Problematically the graph partitioning is NP-complete for  $k \geq 2$ .
- Optimal solution thus would dominate the original calculation as parallel overhead and is not acceptable.  
→ Necessity of fast heuristics.



# Foundations of Parallel Algorithms: Load Balancing

## Recursive coordinate bisection(RCB)

- One needs the positions of the nodes in space (for Finite-Element applications these exist).
- Up to now we have seen the methods under the name **recursive bisection**.
- This time the problem is coupled. Hence it is important, that the space, in whose coordinates the bisection is performed, coincides with the space, in which the nodes reside.
- In the picture this is not the case. Two nodes may be near each other in space, a coordinate bisection does not make sense since the points here are not coupled, thus a processor has no advantage of storing both.



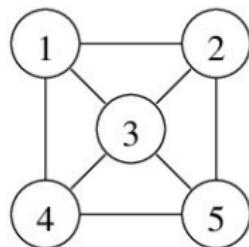
Counter example for application of RCB

# Foundations of Parallel Algorithms: Load Balancing

## Rekursive Spectral Bisektion(RSB)

Here the positions of nodes in space are not needed. One constitutes first the Laplacian matrix  $A(G)$  for the given graph  $G$ . This is defined in the following way:

$$A(G) = \{a_{ij}\}_{i,j=1}^{|V|} \quad \text{with} \quad a_{ij} = \begin{cases} \text{degree}(v_i) & i = j \\ -1 & (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$



|   | 1  | 2  | 3  | 4  | 5  |
|---|----|----|----|----|----|
| 1 | 3  | -1 | -1 | -1 | 0  |
| 2 | -1 | 3  | -1 | 0  | -1 |
| 3 | -1 | -1 | 4  | -1 | -1 |
| 4 | -1 | 0  | -1 | 3  | -1 |
| 5 | 0  | -1 | -1 | -1 | 3  |

graph and related Laplacian matrix

# Foundations of Parallel Algorithms: Load Balancing

Then solve the eigen value problem

$$A\mathbf{e} = \lambda \mathbf{e}$$

The smallest eigen value  $\lambda_1$  equals zero, since with  $\mathbf{e}_1 = (1, \dots, 1)^T$  applies  $A\mathbf{e}_1 = 0 \cdot \mathbf{e}_1$ . The second smallest eigen value  $\lambda_2$  however is not zero, if the graph is connected.

The bisection now is performed by means of the components of the eigen vector  $\mathbf{e}_2$ , one builds for  $c \in \mathbb{R}$  the two index sets

$$\begin{aligned}I_0 &= \{i \in \{1, \dots, |V|\} \mid (\mathbf{e}_2)_i \leq c\} \\I_1 &= \{i \in \{1, \dots, |V|\} \mid (\mathbf{e}_2)_i > c\}\end{aligned}$$

and the partition mapping

$$\pi(v) = \begin{cases} 0 & \text{if } v = v_i \wedge i \in I_0 \\ 1 & \text{if } v = v_i \wedge i \in I_1 \end{cases}$$

Tough one chooses  $c$  such that the work is equally distributed (median).

## Kernighan/Lin

- Iteration method, that improves a given partitioning under consideration of the cost functional.
- We restrict ourselves to bisection ( $k$ -way extension is possible) and assume the node weights as 1.
- Furthermore the count of nodes shall be even.
- The method of Kernighan/Lin is mostly used in combination with other methods.

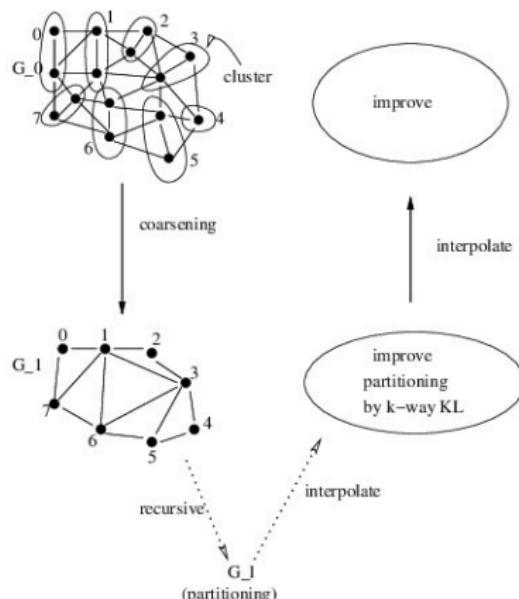
# Foundations of Parallel Algorithms: Load Balancing KL

```
i = 0; | V | = 2n;
// Generate Π_0 such, that equal distribution is given.
while (1) { // iteration step
 $V_0 = \{v \mid \pi(v) = 0\};$
 $V_1 = \{v \mid \pi(v) = 1\};$
 $V_0' = V_1' = \emptyset;$
 $\bar{V}_0 = V_0;$
 $\bar{V}_1 = V_1;$
 for (i = 1 ; i \leq n ; i++)
 {
 // choose $v_i \in V_0 \setminus V_0'$ und $w_i \in V_1 \setminus V_1'$ such, that
 $\sum_{e \in (\bar{V}_0 \times \bar{V}_1) \cap E} w(e) - \sum_{e \in (V_0'' \times V_1'') \cap E} w(e) \rightarrow \max$
 // where
 $V_0'' = \bar{V}_0 \setminus \{v_i\} \cup \{w_i\}$
 $V_1'' = \bar{V}_1 \setminus \{w_i\} \cup \{v_i\}$
 // set
 $\bar{V}_0 = \bar{V}_0 \setminus \{v_i\} \cup \{w_i\};$
 $\bar{V}_1 = \bar{V}_1 \setminus \{w_i\} \cup \{v_i\};$
 } // for
 // rem.: max can be negative, thus worsening of the separator costs
 // result at this point: sequence of pairs $\{(v_1, w_1), \dots, (v_n, w_n)\}.$
 // V_0, V_1 have not been altered.
 // Now choose a partial sequence up to $m \leq n$, that performs a
 // maximal improvement of the costs („hill climbing“)
 $V_0 = V_0 \setminus \{v_1, \dots, v_m\} \cup \{w_1, \dots, w_m\};$
 $V_1 = V_1 \setminus \{w_1, \dots, w_m\} \cup \{v_1, \dots, v_m\};$
 if (m == 0) break; // end
} // while
```

# Foundations of Parallel Algorithms: Load Balancing

## Multilevel k-way partitioning

- ➊ Agglomeration of nodes of the starting graph  $G^0$  (e.g. random or by means of heavy edge weight) in clusters.
- ➋ These clusters define the nodes in a coarsened graph  $G^1$ .
- ➌ By recursion this behaviour leads to a graph  $G^l$ .



- ➊  $G^l$  is now partitioned (e.g. RSB/KL).
- ➋ After that the partition function is interpolated onto the finer graph  $G^{l-1}$ .
- ➌ This interpolated partitioning function can now be improved again via KL recursively and then be interpolated onto the next finer graph.
- ➍ In this way we continue recursively up to the starting graph.
- ➎ The implementation is then possible in  $\mathcal{O}(n)$  steps. The method provides qualitatively profound partitions.

## Further problems

Further problems for the partitioning are

- **Dynamic repartitioning:** The graph partitioning shall be changed with the least possible local rebalancing to regain work balance.
- **Constraint partitioning:** From other algorithmic parts additional data dependencies exist.
- **Parallelisation of partitioning methods:** Is necessary for large data sets (Here finished software exists)

# Algorithms for Dense Matrices I

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

Algorithms for dense matrices as data parallel algorithms

- Data distribution of vectors and matrices
- Matrix transposition

# Partitioning of Vectors I

- Vector  $x \in \mathcal{R}^N$  corresponds to an ordered list of numbers.
- Each index  $i$  of the index set  $I = \{0, \dots, N - 1\}$  is assigned a real number  $x_i$ .
- Instead of  $\mathcal{R}^N$  we write  $\mathcal{R}(I)$  to emphasize the dependency of the index set.
- The natural (and most efficient) data structure for a vector is the array.
- Since arrays start in many programming languages with index 0, this is also the case for the index set  $I$ .

## Partitioning of Vectors II

- A data partitioning matches now a segmentation of the index set  $I$

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

where  $P$  is the process set.

- With good load balancing the index sets  $I_p$ ,  $p \in P$  should contain each (nearly) an equal number of elements.
- Process  $p \in P$  stores such the components  $x_i$ ,  $i \in I_p$  of the vector  $x$ .
- In each process we would again like to work with a contiguous index set  $\tilde{I}_p$ , that starts at 0, this means

$$\tilde{I}_p = \{0, \dots, |I_p| - 1\}.$$

# Partitioning of Vectors III

The mappings

$$\begin{aligned} p: \quad I &\rightarrow P \text{ resp.} \\ \mu: \quad I &\rightarrow \mathbf{N} \end{aligned}$$

assign each index  $i \in I$  invertible unique 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 invertible mapping

$$\mu^{-1}: \underbrace{\bigcup_{p \in P} \{p\} \times \tilde{I}_p}_{\subset P \times \mathbf{N}} \rightarrow I$$

provides for each local index  $i \in \tilde{I}_p$  and process  $p \in P$  the global index  $\mu^{-1}(p, i)$ , thus

$$p(\mu^{-1}(p, i)) = p \text{ and } \mu(\mu^{-1}(p, i)) = i.$$

## Partitioning of Vectors IV

Common partitionings are especially the *cyclic partitioning* with<sup>1</sup>

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

and the *blockwise partitioning* with

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

with  $B = N \div P$  and  $R = N \% P$ . Here is the idea, that the first  $R$  processes get  $B + 1$  indices and the remaining  $B$  indices each.

---

<sup>1</sup>  $\div$  means integer division;  $\%$  the modulo function

# Partitioning of Vectors V

Cyclic and blockwise partitioning for  $N = 13$  and  $P = 4$ :  
cyclic partitioning:

|           |   |   |   |   |   |   |   |   |   |   |    |    |    |
|-----------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| $I:$      | 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  |

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

blockwise partitioning

|           |   |   |   |   |   |   |   |   |   |   |    |    |    |
|-----------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| $I:$      | 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  |

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

# Partitioning of Matrices I

- For a matrix  $A \in \mathcal{R}^{N \times M}$  each tuple  $(i, j) \in I \times J$ , with  $I = \{0, \dots, N - 1\}$  and  $J = \{0, \dots, M - 1\}$ , is assigned a real number  $a_{ij}$ .
- In principle the assignment of matrix elements to processors is arbitrary
- However the elements assigned to a processor can in general *not* be represented as matrix again.
- Exception: separate segmentation of the one-dimensional index sets  $I$  and  $J$ .
- Therefore we assume the processes as being organized as a two-dimensional field , thus

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

## Partitioning of Matrices II

- The index sets  $I, J$  are partitioned into

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

- process  $(p, q)$  is then responsible for the indices  $I_p \times J_q$ .
- Locally process  $(p, q)$  stores its elements then as  $\mathcal{R}(I_p \times J_q)$  matrix.
- The partitioning of  $I$  and  $J$  are formally described by the mappings  $p$  and  $\mu$  as well as  $q$  and  $\nu$ :

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

# Partitioning of Matrices III

Examples for partitioning of a  $6 \times 9$  matrix onto four processors

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

|   |   |   |   |   |   |   |   |   |       |
|---|---|---|---|---|---|---|---|---|-------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $J$   |
| 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | $q$   |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | $\nu$ |

A diagram showing a 6x9 matrix partitioned into 4 columns (P=1, Q=4). The columns are colored red, blue, green, and yellow. The first column has 6 red blocks. The second column has 6 blue blocks. The third column has 6 green blocks. The fourth column has 6 yellow blocks. The matrix is partitioned cyclically across 4 processors.

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

|     |     |       |     |     |     |     |     |     |     |
|-----|-----|-------|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0     |     |     |     |     |     |     |     |
| 1   | 0   | 1     |     |     |     |     |     |     |     |
| 2   | 1   | 0     |     |     |     |     |     |     |     |
| 3   | 1   | 1     |     |     |     |     |     |     |     |
| 4   | 2   | 0     |     |     |     |     |     |     |     |
| 5   | 3   | 0     |     |     |     |     |     |     |     |
| $I$ | $p$ | $\mu$ | ... | ... | ... | ... | ... | ... | ... |

# Partitioning of Matrices IV

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

|     | 0   | 1     | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $J$   |
|-----|-----|-------|---|---|---|---|---|---|---|-------|
|     | 0   | 0     | 0 | 0 | 0 | 1 | 1 | 1 | 1 | $q$   |
|     | 0   | 1     | 2 | 3 | 4 | 0 | 1 | 2 | 3 | $\nu$ |
| 0   | 0   | 0     |   |   |   |   |   |   |   |       |
| 1   | 1   | 0     |   |   |   |   |   |   |   |       |
| 2   | 0   | 1     |   |   |   |   |   |   |   |       |
| 3   | 1   | 1     |   |   |   |   |   |   |   |       |
| 4   | 0   | 2     |   |   |   |   |   |   |   |       |
| 5   | 1   | 2     |   |   |   |   |   |   |   |       |
| $I$ | $p$ | $\mu$ |   |   |   |   |   |   |   |       |

# Partitioning of Matrices V

Which data partitioning is now the best one?

- In general the organisation of the processes as a nearly quadratic array leads to a partitioning with good load balancing.
- More important is however that different partitionings are suited differently good for distinct algorithms.
- We will see, that a process array with cyclic partitioning is suited quite well for row as well as column indices for the  $LU$  partitioning.
- This partitioning is however not optimal for the solution of the resulting triangular systems. If one has to solve the equation system for many righthand sides then a compromise has to be achieved.
- This generally holds for nearly all tasks of linear algebra: The multiplication of two matrices or the transposition of a matrix represents only a step in a larger algorithm.
- The data partitioning can thus not be optimized towards a partial step, but should give a meaningful tradeoff. Eventually can be thought whether rearranging (copying) the data into a different structure is advantageous.

# Transposition of a Matrix I

## Task description

Given:  $A \in \mathcal{R}^{N \times M}$  distributed onto a set of processes;

Determine:  $A^T$  with the same data partitioning as  $A$ .

- In principle the problem is trivial.
- We could distribute the matrix onto the processors such, that only communication with nearest neighbors is necessary (since the processes communicate pairwise).

|    |    |    |    |
|----|----|----|----|
| 12 | 1  | 3  | 5  |
| 0  | 13 | 7  | 9  |
| 2  | 6  | 14 | 11 |
| 4  | 8  | 10 | 15 |

Optimal data distribution for the matrix transposition (the numbers denote the processor numbers).

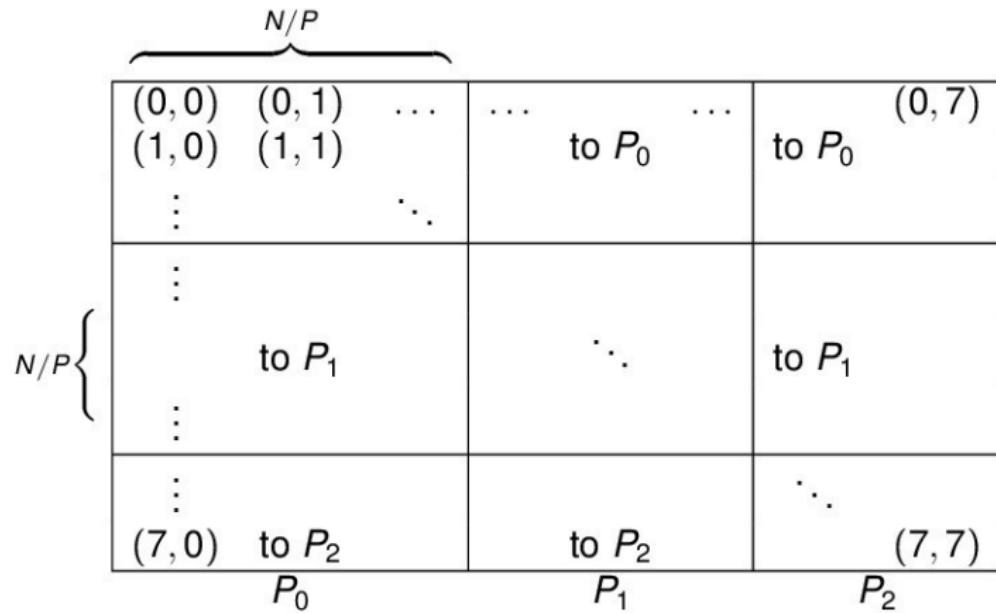
## Transposition of a Matrix II

Example with ring topology:

- Obviously only communication is necessary between direct neighbors ( $0 \leftrightarrow 1, 2 \leftrightarrow 3, \dots, 10 \leftrightarrow 11$ ).
- Albeit these data partitioning does not coincide with the scheme, that we just have introduced and is for example less suited for the multiplication of two matrices.

# Transposition of a Matrix: 1D Partitioning

Let us consider without loss of generality a column-wise, blocked partitioning



$8 \times 8$  matrix on three processors in column-wise, blocked distribution.

# Transposition of a Matrix: 1D Partitioning

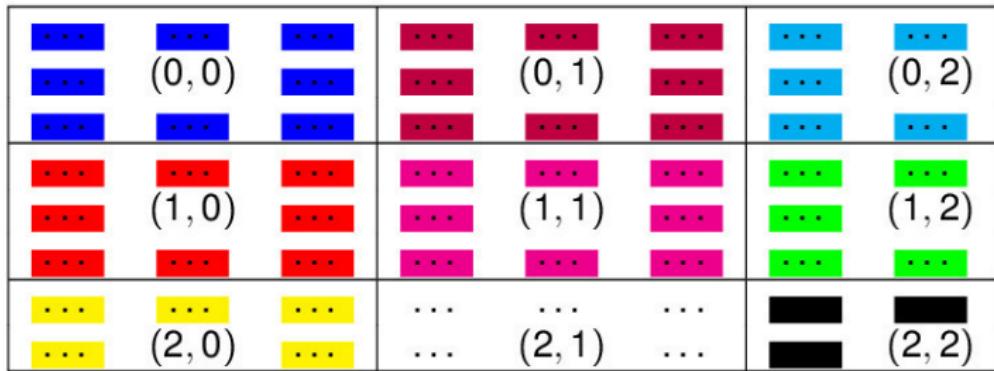
- Obviously in this case each processor has to send data to each other.
- Thus an all-to-all communication with individual messages has to be performed.
- Let us assume a hypercube structure as connection topology, then we get the following parallel runtime for a  $N \times N$  matrix and  $P$  processors:

$$\begin{aligned} T_P(N, P) &= \underbrace{2(t_s + t_h) \text{Id } P}_{\text{setup}} + \underbrace{t_w \frac{N^2}{P^2} P \text{Id } P}_{\text{data trans-mission}} + \underbrace{(P - 1) \frac{N^2}{P^2} \frac{t_e}{2}}_{\text{transposition}} \approx \\ &\approx \text{Id } P(t_s + t_h)2 + \frac{N^2}{P} \text{Id } Pt_w + \frac{N^2}{P} \frac{t_e}{2} \end{aligned}$$

- Also for fixed  $P$  and increasing  $N$  we cannot make the communication share of the total runtime arbitrary small.
- This is the same for all algorithms for transposition (also for an optimal distribution as above).
- Matrix transposition has therefore no iso-efficiency function and is not scalable.

# Transposition of a Matrix: 2D Partitioning

We consider now the two-dimensional, blocked distribution of a  $N \times N$  matrix onto a  $\sqrt{P} \times \sqrt{P}$  processor array:

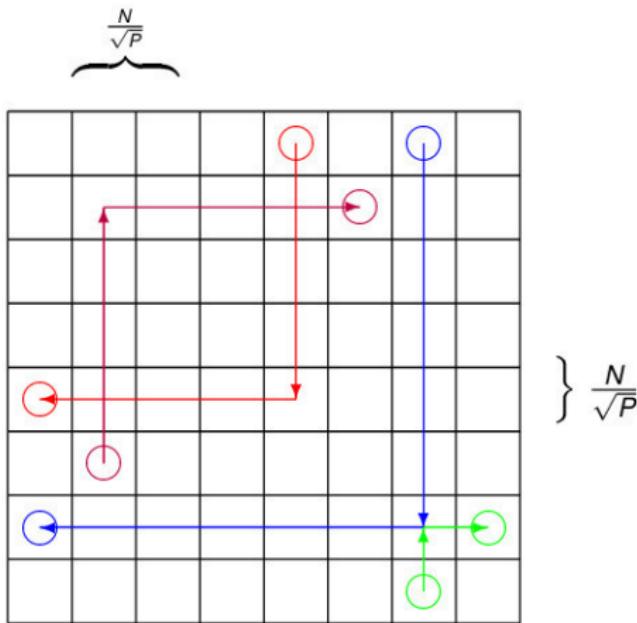


Example for a two-dimensional, blocked distribution  $N = 8$ ,  $\sqrt{P} = 3$ .

# Transposition of a Matrix

- Each processor has to exchange its partial matrix with exactly one other.
- A naive transposition algorithm for these configuration is:
  - ▶ Processors  $(p, q)$  below the main diagonal ( $p > q$ ) send the partial matrix in the column to above up to processor  $(q, q)$ , thereafter the partial matrix is routed to the right up to the final column to processor  $(q, p)$ .
  - ▶ Corresponding the data of processors  $(p, q)$  are routed above the main diagonal ( $q > p$ ) first in the column  $q$  to below up to  $(q, q)$  and then to the left until  $(q, p)$  is reached.

# Transposition of a Matrix



Diverse paths of partial matrices for  $\sqrt{P} = 8$ .

# Transposition of a Matrix

- Obviously route the processors  $(p, q)$  with  $p > q$  data from below to above resp. right to left and processors  $(p, q)$  with  $q > p$  correspondingly data from above to below and left to right.
- For synchronous communication in each step four send- resp. receive operations are necessary, and in total one needs  $2(\sqrt{P} - 1)$  steps.
- The parallel runtime therefore amounts

$$\begin{aligned} T_P(N, P) &= 2(\sqrt{P} - 1) \cdot 4 \left( t_s + t_h + t_w \left( \frac{N}{\sqrt{P}} \right)^2 \right) + \frac{1}{2} \left( \frac{N}{\sqrt{P}} \right)^2 t_e \approx \\ &\approx \sqrt{P}8(t_s + t_h) + \frac{N^2}{P} \sqrt{P}8t_w + \frac{N^2}{P} \frac{t_e}{2} \end{aligned}$$

- In comparison to a one-dimensional distribution with hypercube one has in the data transmission the factor  $\sqrt{P}$  instead of  $\text{Id } P$ .

# Recursive Transposition Algorithm

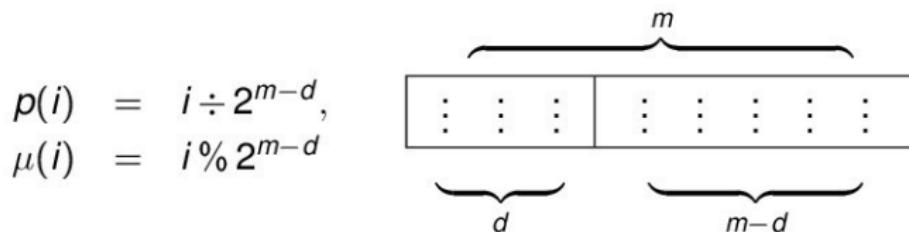
This algorithm is based on the following observation: For a  $2 \times 2$  block matrix partitioning of  $A$  applies

$$A^T = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \end{pmatrix}$$

thus the off-diagonal blocks change the places and then each partial matrix has to be transposed. This of course happens recursively until a  $1 \times 1$  matrix is reached. Is  $N = 2^n$ , then  $n$  recursion steps are necessary.

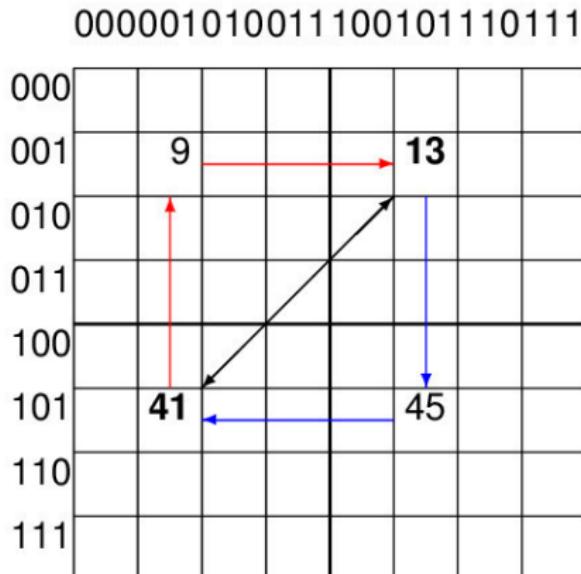
## Recursive Transposition Algorithm

- The hypercube is the ideal connection topology for this algorithm.
- With  $N = 2^n$  and  $\sqrt{P} = 2^d$  with  $n \geq d$  this mapping of indices  $I = \{0, \dots, N - 1\}$  is done on the processors via



- The upper  $d$  bits of an index describe the processor, on which the index is mapped.
- Consider as example  $d = 3$ , thus  $\sqrt{P} = 2^3 = 8$ .
- In the recursion step the matrix has to be divided into  $2 \times 2$  blocks from  $4 \times 4$  partial matrices and  $2 \cdot 16$  processors have to exchange data, for example processor  $101001 = 41$  and  $001101 = 13$ . This happens in two steps over the processors  $001001 = 9$  and  $101101 = 45$ .
- These are both *direct* neighbors of the processors 41 and 13 in the hypercube.

# Recursive Transposition Algorithm



Communication in the recursive transposition algorithm for  $d = 3$ .

The recursive transposition algorithm works now recursive on the processor topology. Is a processor reached, the transposition is continued with the sequential algorithms. The parallel runtime is described with

$$T_P(N, P) = \text{Id } P(t_s + t_h)2 + \frac{N^2}{P} \text{Id } \sqrt{P}2t_w + \frac{N^2}{P} \frac{t_e}{2}$$

# Recursive Transposition Algorithm

Program (Recursive transposition algorithm on hypercube)

parallel recursive transpose

```
{
 const int d = ... , n = ... ;
 const int P = 2^d , N = 2^n ;

 process $\Pi[\text{int } (p, q) \in \{0, \dots, 2^d - 1\} \times \{0, \dots, 2^d - 1\}]$
 {
 Matrix A, B; // A is the input matrix
 void rta(int r, int s, int k)
 {
 if ($k == 0$) {A = A^T ; return;}
 int i = p - r, j = q - s, l = 2^{k-1} ;
 if ($i < l$)
 {
 if ($j < l$)
 {
 recv(B, $\Pi_{p+l, q}$); send(B, $\Pi_{p, q+l}$);
 rta(r, s, k - 1);
 }
 else
 {
 send(A, $\Pi_{p+l, q}$); recv(A, $\Pi_{p, q-l}$);
 rta(r, s + l, k - 1);
 }
 }
 ...
 }
 }
}
```

# Recursive Transposition Algorithm cont.

Program (Recursive transposition algorithm on hypercube cont.)

**parallel** recursive transpose cont.

```
{
 ...
 else
 {
 if ($j < l$) {
 send($A, \Pi_{p-l,q}$); recv($A, \Pi_{p,q+l}$);
 rta($r + l, s, k - 1$);
 }
 else
 {
 recv($B, \Pi_{p-l,q}$); send($B, \Pi_{p,q-l}$);
 rta($r + l, s + l, k - 1$);
 }
 }
 rta(0,0,d);
}
}
```

# Algorithms for Dense Matrices II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

Data parallel algorithms for dense matrices

- Matrix-Vector Multiplication
- Matrix-Matrix Multiplication

# Matrix-Vector Multiplication

Compute  $y = Ax$ , matrix  $A \in \mathcal{R}^{N \times M}$  and vector  $x \in \mathcal{R}^M$

- Different possibilities for data partitioning
- Distribution of the matrix and the vector have to fit together
- Distribution of the result vector  $y \in \mathcal{R}^N$  same as of input vector  $x$

Example:

- Matrix is blockwise distributed onto an array topology
- Input vector  $x$  is correspondingly distributed blockwise across the diagonal processors
- The processor array is quadratic
- Vector segment  $x_q$  is needed in each processor column and is therefore to copy in each column (one-to-all).
- Local computation of the product  $y_{p,q} = A_{p,q}x_q$ .
- Complete segment  $y_p$  results first from the summation  $y_p = \sum_q y_{p,q}$ . (further all-to-one communication)
- Result can immediately used for further matrix-vector multiplications

# Matrix-Vector Multiplication: Partitioning

Partitioning for the Matrix-Vector product

|                             |                             |                             |                             |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| $x_0$<br>$y_0$<br>$A_{0,0}$ | $x_1$<br>$y_0$<br>$A_{0,1}$ | $x_2$<br>$y_0$<br>$A_{0,2}$ | $x_3$<br>$y_0$<br>$A_{0,3}$ |
| $x_0$<br>$y_1$<br>$A_{1,0}$ | $x_1$<br>$y_1$<br>$A_{1,1}$ | $x_2$<br>$y_1$<br>$A_{1,2}$ | $x_3$<br>$y_1$<br>$A_{1,3}$ |
| $x_0$<br>$y_2$<br>$A_{2,0}$ | $x_1$<br>$y_2$<br>$A_{2,1}$ | $x_2$<br>$y_2$<br>$A_{2,2}$ | $x_3$<br>$y_2$<br>$A_{2,3}$ |
| $x_0$                       | $x_1$                       | $x_2$                       | $x_3$                       |

# Matrix-Vector Multiplication: Parallel Runtime

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

$$\begin{aligned} T_P(N, P) &= \underbrace{\left( t_s + t_h + t_w \frac{\overbrace{N}^{\text{vector}}}{\sqrt{P}} \right) \text{Id } \sqrt{P}}_{\text{Distribute } x \text{ across column}} + \underbrace{\left( \frac{N}{\sqrt{P}} \right)^2 2t_f}_{\text{local matrix-vector mult.}} \\ &\quad + \underbrace{\left( t_s + t_h + t_w \frac{N}{\sqrt{P}} \right) \text{Id } \sqrt{P}}_{\text{reduction } (t_f \ll t_w)} = \\ &= \text{Id } \sqrt{P}(t_s + t_h)2 + \frac{N}{\sqrt{P}} \text{Id } \sqrt{P}2t_w + \frac{N^2}{P}2t_f \end{aligned}$$

For fixed  $P$  and  $N \rightarrow \infty$  the communication share get arbitrary small, thus an iso-efficiency function exists, the algorithm is scalable.

# Matrix-Vector Multiplication: Work/Overhead

Let us compute work and overhead:

Recalculate to the work  $W$ :

$$W = N^2 2t_f \text{ (seq. runtime)}$$

$$\Rightarrow N = \frac{\sqrt{W}}{\sqrt{2t_f}}$$

$$T_P(W, P) = \text{Id } \sqrt{P}(t_s + t_h)2 + \frac{\sqrt{W}}{\sqrt{P}} \text{Id } \sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + \frac{W}{P}$$

Overhead:

$$T_O(W, P) = PT_P(W, P) - W =$$

$$= \sqrt{W} \sqrt{P} \text{Id } \sqrt{P} \frac{2t_w}{\sqrt{2t_f}} + P \text{Id } \sqrt{P}(t_s + t_h)2$$

# Matrix-Vector Multiplication: Iso-Efficiency

and now the iso-efficiency function:

Iso-efficiency ( $T_O(W, P) \stackrel{!}{=} KW$ ):  $T_O$  has two terms.

For the first we achieve

$$\begin{aligned} \sqrt{W} \sqrt{P} \text{Id} \sqrt{P} \frac{2t_w}{\sqrt{2t_f}} &= KW \\ \iff W &= P(\text{Id} \sqrt{P})^2 \frac{4t_w^2}{2t_f K^2} \end{aligned}$$

and for the second

$$\begin{aligned} P \text{Id} \sqrt{P} (t_s + t_h) 2 &= KW \\ \iff W &= P \text{Id} \sqrt{P} \frac{(t_s + t_h) 2}{K}; \end{aligned}$$

thus  $W = O(P(\text{Id} \sqrt{P})^2)$  is the desired iso-efficiency function.

# Matrix-Matrix Multiplication

## Algorithm of Cannon

It is to calculate  $C = A \cdot B$ .

- The  $N \times N$  matrices  $A$  and  $B$ , that are to multiply, are blockwise distributed onto a 2D-array topology ( $\sqrt{P} \times \sqrt{P}$ )
- For practical reasons should be the result  $C$  again be stored in the same partitioning.
- Process  $(p, q)$  has thus

$$C_{p,q} = \sum_k A_{p,k} \cdot B_{k,q}$$

to calculate, needs therefore block row  $p$  of  $A$  and block column  $q$  of  $B$ .

# Matrix-Matrix Multiplication

The two phases of Cannon's algorithm are

- ① *Alignment phase:* The blocks of  $A$  are shifted in each row cyclic to the left, until the diagonal blocks reside in the first column. Correspondingly one shifts the blocks of  $B$  in the columns to above, until all diagonal blocks reside in the first row.

After the alignment phase processor  $(p, q)$  has the blocks

$$A_{\underbrace{p, (q+p) \% \sqrt{P}}}_{(row p shifts p times to the left)}$$

$$B_{\underbrace{(p+q) \% \sqrt{P}, q}_{(column q shifts q time to above)}}.$$

- ② *Computing phase:* Obviously now each process stores two fitting blocks, that it can multiply. Are the blocks of  $A$  in each row of  $A$  shifted for one position to the left and the one of  $B$  in each column to the above, then each owns again two fitting blocks. After  $\sqrt{P}$  steps the result is ready.

# Cannon's Algorithm

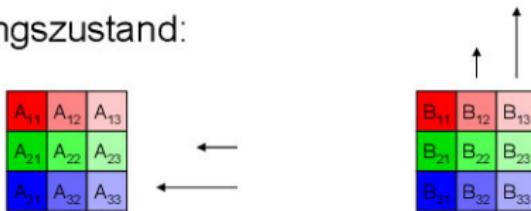
- Is based on blockwise partitioning of the matrices
- Setup phase
  - ▶ Rotation of the matrizen  $A$  and  $B$
- Iteration over  $\sqrt{p}$  steps
  - ▶ Compute locally block matrix product
  - ▶ Shift  $A$  horizontally and  $B$  vertically

$$\begin{matrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{matrix} = \begin{matrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{matrix} \times \begin{matrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{matrix}$$

$C$        $A$        $B$

# Cannon's Algorithm - Rotation

- Anfangszustand:



- Verdrehen: Rotieren der i. Zeile (Spalte) von A (B) um i-Schritte:



# Cannon's Algorithm - Iteration

1.

$$\begin{matrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{matrix} = \begin{matrix} A_{11} & A_{12} & A_{13} \\ A_{22} & A_{23} & A_{21} \\ A_{33} & A_{31} & A_{32} \end{matrix} \times \begin{matrix} B_{11} & B_{22} & B_{33} \\ B_{21} & B_{32} & B_{13} \\ B_{31} & B_{12} & B_{23} \end{matrix}$$

*C*                    *A*                    *B*

$$C_{11} = A_{11}B_{11}$$

2.

$$\begin{matrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{matrix} = \begin{matrix} A_{12} & A_{13} & A_{11} \\ A_{23} & A_{21} & A_{22} \\ A_{31} & A_{32} & A_{33} \end{matrix} \times \begin{matrix} B_{11} & B_{32} & B_{13} \\ B_{31} & B_{12} & B_{23} \\ B_{11} & B_{22} & B_{33} \end{matrix}$$

←                    ←

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

3.

$$\begin{matrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{matrix} = \begin{matrix} A_{13} & A_{11} & A_{12} \\ A_{21} & A_{22} & A_{23} \\ A_{32} & A_{33} & A_{31} \end{matrix} \times \begin{matrix} B_{11} & B_{12} & B_{23} \\ B_{11} & B_{22} & B_{33} \\ B_{21} & B_{32} & B_{13} \end{matrix}$$

←                    ←

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$

# Cannon with MPI (Init)

```
/* Baue Gitter und hole Koordinaten */
int dims[2], periods[2] = {1, 1};
int mycoords[2];

dims[0] = sqrt(num_procs);
dims[1] = num_procs / dims[0];

MPI_Cart_create(MPI_COMM_WORLD, /* kollektiv */
 2, dims, periods,
 0, &comm_2d);
MPI_Comm_rank(comm_2d, &my2drank);
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

/* Lokale Blöcke der Matrizen */
double *a, *b, *c;

/* Lade a, b und c entsprechend der Koordinaten */
...
```

# Cannon with MPI (Rotate)

```
/* Matrix-Verdrehung A */
MPI_Cart_shift(comm_2d, 0, -mycoords[0],
 &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
 shiftdest, 77, shiftsource, 77,
 comm_2d, &status);

/* Matrix-Verdrehung B */
MPI_Cart_shift(comm_2d, 1, -mycoords[1],
 &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
 shiftdest, 77, shiftsource, 77,
 comm_2d, &status);
```

# Cannon with MPI (Iteration)

```
/* Finde linken und oberen Nachbarn */
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);

for (i=0; i<dims[0]; i++)
{
 dgemm(nlocal, a, b, c); /* c = c + a * b */

 /* Matrix A nach links rollen */
 MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
 leftrank, 77, rightrank, 77,
 comm_2d, &status);

 /* Matrix B nach oben rollen */
 MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
 uprank, 77, downrank, 77,
 comm_2d, &status);
}

/* A und B zurück in Ursprungs-Zustand */
...
```

# Cannon - Practical Aspects

- Efficient, but not simple generalisation, if
  - ▶ Matrices are not quadratic
  - ▶ Dimensions are not without rest divisible by p
  - ▶ Other matrix partitionings are needed
- Iso-efficiency function of Cannon's Algorithmus:  $O(P^{3/2})$ ,  
 $N/\sqrt{P} = \text{const} \rightarrow$  Efficiency remains constant for fixed block sizes per processor and increasing processor count
- Dekel-Nassimi-Salmi algorithm enables the usage of  $N^3$  processors (Cannon  $N^2$ ) with better iso-efficiency function.

# Matrix-Matrix Multiplication: Iso-efficiency Analysis

Consider the corresponding iso-efficiency function.

Sequential runtime :

$$W = T_S(N) = N^3 2t_f$$

$$\Rightarrow N = \left( \frac{W}{2t_f} \right)^{\frac{1}{3}}$$

Parallel runtime:

$$\begin{aligned} T_P(N, P) &= \underbrace{\left( \sqrt{P} - 1 \right) \left( t_s + t_h + t_w \frac{N^2}{P} \right)}_{\text{alignment}} \overbrace{4}^{\text{send/recv } A/B} \\ &\quad + \sqrt{P} \left( \underbrace{\left( \frac{N}{\sqrt{P}} \right)^3 2t_f}_{\text{multiplic. of a block}} + \left( t_s + t_h + t_w \frac{N^2}{P} \right) 4 \right) \approx \\ &\approx \sqrt{P} (t_s + t_h) 8 + \frac{N^2}{\sqrt{P}} t_w 8 + \frac{N^3}{P} 2t_f \\ T_P(W, P) &= \sqrt{P} (t_s + t_h) 8 + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \frac{8t_w}{(2t_f)^{\frac{1}{3}}} + \frac{W}{P} \end{aligned}$$

# Matrix-Matrix Multiplication: Iso-efficiency Analysis

Overhead:

$$T_O(W, P) = PT_P(W, P) - W = P^{\frac{3}{2}}(t_s + t_h)8 + \sqrt{P}W^{\frac{2}{3}} \frac{8t_w}{(2t_f)^{\frac{1}{3}}}$$

Result:

- Thus is  $W = O(P^{3/2})$ .
- Because of  $N = \left(\frac{W}{2t_f}\right)^{1/3}$  applies  $N/\sqrt{P} = \text{const}$
- Thus for *fixed* block size in each processor and increasing processor count the efficiency keeps constant.
- If we restrict in the algorithm of Cannon to  $1 \times 1$  blocks per processor, thus  $\sqrt{P} = N$ , then we can use for the required  $N^3$  multiplications only  $N^2$  processors.
- This is the reason for the iso-efficiency function of order  $P^{3/2}$ .

# Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi-Alg.

## Dekel-Nassimi-Salmi-Algorithm

- Now we consider an algorithm that renders the usage of up to  $N^3$  processors for a  $N \times N$  matrix possible.
- Given are then  $N \times N$  matrices  $A$  and  $B$  as well as a 3D array of processors of dimension  $P^{1/3} \times P^{1/3} \times P^{1/3}$ .
- The processors are addressed by the coordinates  $(p, q, r)$ .
- To calculate the block  $C_{p,q}$  of the result matrix  $C$  via

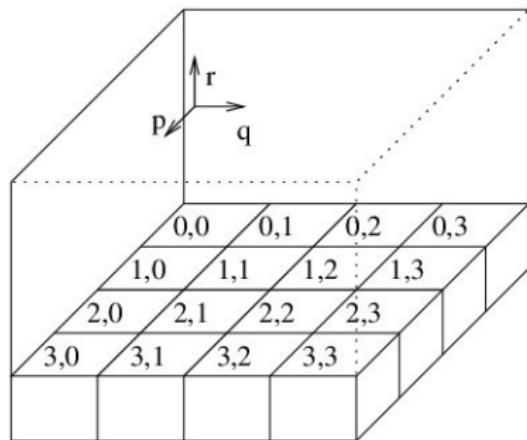
$$C_{p,q} = \sum_{r=0}^{P^{\frac{1}{3}}-1} A_{p,r} \cdot B_{r,q} \quad (1)$$

we use  $P^{1/3}$  processors, in detail processor  $(p, q, r)$  is exactly responsible for the product  $A_{p,r} \cdot B_{r,q}$ .

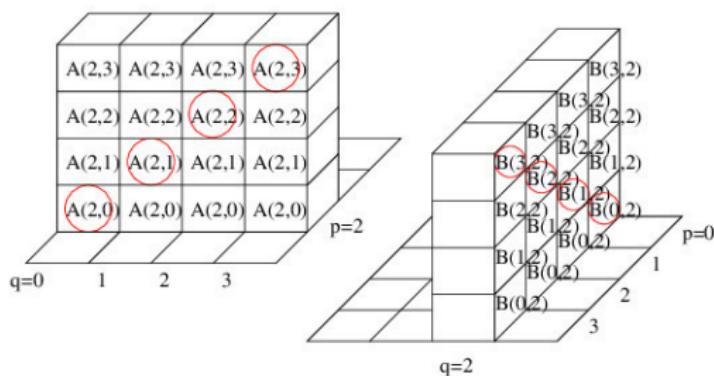
- Now it still to decide, how the input and result matrices shall be distributed.
- Both  $A$  and  $B$  are partitioned into  $P^{1/3} \times P^{1/3}$  blocks of size  $\frac{N}{P^{1/3}} \times \frac{N}{P^{1/3}}$ .
- $A_{p,q}$  and  $B_{p,q}$  is stored in the beginning in processor  $(p, q, 0)$ , also the result  $C_{p,q}$  shall reside there.
- The processors  $(p, q, r)$  for  $r > 0$  are only used temporarily.

# Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

Distribution of  $A$ ,  $B$ ,  $C$  for  $P^{1/3} = 4$  ( $P=64$ ).



Partitioning of the blocks of  $A$  and  $B$  (at the beginning) and  $C$  (at the end)



Distribution of  $A$  and  $B$  for the multiplication

## Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

- That now each processor  $(p, q, r)$  can perform „its“ multiplication  $A_{p,r} \cdot B_{r,q}$ , the involved blocks of  $A$  and  $B$  first have to be moved to the right position.
- All processors require  $(p, *, r)$  the block  $A_{p,r}$  and all processors  $(*, q, r)$  the block  $B_{r,q}$ .
- The distribution is achieved in the following way:

*Processor  $(p, q, 0)$  sends  $A_{p,q}$  to processor  $(p, q, q)$  and sends then  $(p, q, q)$  the  $A_{p,q}$  to all  $(p, *, q)$  via a one-to-all communication on  $P^{1/3}$  processors. Corresponding sends  $(p, q, 0)$  the  $B_{p,q}$  to processor  $(p, q, p)$ , and this distributed then to  $(*, q, p)$ .*

- After the multiplication in each  $(p, q, r)$  the results of all  $(p, q, *)$  are still to collect in  $(p, q, 0)$  via a all-to-one communication on  $P^{1/3}$  processors.

# Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

Let us analyse the method in detail (3D-cut-through network):

$$W = T_S(N) = N^3 2t_f \Rightarrow N = \left( \frac{W}{2t_f} \right)^{\frac{1}{3}}$$

$$\begin{aligned} T_P(N, P) &= \underbrace{\left( t_s + t_h + t_w \left( \frac{N}{P^{\frac{1}{3}}} \right)^2 \right)}_{(p,q,0) \longrightarrow (p,q,q), (p,q,p)} \underbrace{\overbrace{2}^{A_{p,q} \text{ u. } B_{p,q}}}_{\text{one-to-all}} + \underbrace{\left( t_s + t_h + t_w \left( \frac{N}{P^{\frac{1}{3}}} \right)^2 \right) \text{Id } P^{\frac{1}{3}} \overbrace{2}^{A,B}}_{\text{all-to-one } (t_f \ll t_w)} \\ &\quad + \underbrace{\left( \frac{N}{P^{\frac{1}{3}}} \right)^3 2t_f}_{\text{multiplication}} + \underbrace{\left( t_s + t_h + t_w \left( \frac{N}{P^{\frac{1}{3}}} \right)^2 \right) \text{Id } P^{\frac{1}{3}} \approx}_{\text{all-to-one } (t_f \ll t_w)} \\ &\approx 3 \text{Id } P^{\frac{1}{3}} (t_s + t_h) + \frac{N^2}{P^{\frac{2}{3}}} 3 \text{Id } P^{\frac{1}{3}} t_w + \frac{N^3}{P} 2t_f \\ T_P(W, P) &= 3 \text{Id } P^{\frac{1}{3}} (t_s + t_h) + \frac{W^{\frac{2}{3}}}{P^{\frac{2}{3}}} 3 \text{Id } P^{\frac{1}{3}} \frac{t_w}{(2t_f)^{\frac{2}{3}}} + \frac{W}{P} \\ T_O(W, P) &= P \text{Id } P^{\frac{1}{3}} 3(t_s + t_h) + W^{\frac{2}{3}} P^{\frac{1}{3}} \text{Id } P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}} \end{aligned}$$

# Matrix-Matrix Multiplication: Dekel-Nassimi-Salmi Alg.

- From the second term of  $T_O(W, P)$  we approximate the iso-efficiency function:

$$\begin{aligned} W^{\frac{2}{3}} P^{\frac{1}{3}} \text{Id } P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}}} &= KW \\ \iff W^{\frac{1}{3}} &= P^{\frac{1}{3}} \text{Id } P^{\frac{1}{3}} \frac{3t_w}{(2t_f)^{\frac{2}{3}} K} \\ \iff W &= P \left( \text{Id } P^{\frac{1}{3}} \right)^3 \frac{27t_w^3}{4t_f^2 K^3}. \end{aligned}$$

- Thus we achieve the iso-efficiency function  $O(P(\text{Id } P^{\frac{1}{3}})^3)$  and therefore a better scalability than for Cannon's algorithm.
- We have always assumed, that the optimal sequential complexity of the matrix multiplication is  $N^3$ . The algorithm of Strassen has however a complexity of  $O(N^{2.87})$ .
- For an efficient implementation of the multiplication of two matrix blocks on a processor one has to ensure cache efficiency.

# Algorithms for Dense Matrices III

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

Data parallel algorithms for dense matrices

- LU decomposition

# LU Decomposition: Problem Formulation

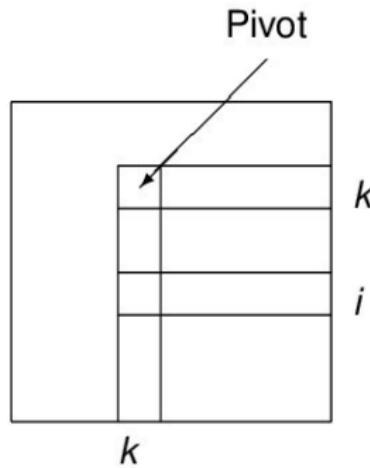
Be the linear equation system to solve

$$Ax = b \quad (1)$$

with a  $N \times N$  matrix  $A$  and according vectors  $x$  and  $b$ .

Gaussian Elimination Method (sequential)

```
(1) for (k = 0; k < N; k++)
(2) for (i = k + 1; i < N; i++) {
(3) lik = aik / akk;
(4) for (j = k + 1; j < N; j++)
(5) aij = aij - lik * akj;
(6) bi = bi - lik * bk;
 }
```



transforms the equation system (1) into the equation system

$$Ux = d \quad (2)$$

with an upper triangular matrix  $U$ .

# LU Decomposition: Properties

Above formulation has the following properties:

- The matrix elements  $a_{ij}$  for  $j \geq i$  contain the according entries of  $U$ , this means  $A$  will be overwritten.
- Vector  $b$  is overwritten with the elements of  $d$ .
- It is assumed, that the  $a_{kk}$  in line (3) is always non zero (no pivoting).

# LU Decomposition: Derivation of Gaussian Elimination

The  $LU$  decomposition can be derived from Gaussian elimination:

- Each individual transformation step, that consists for fixed  $k$  and  $i$  from the lines (3) to (5), can be written as a multiplication of the equation system with a matrix  $\hat{L}_{ik}$  from left:

$$\hat{L}_{ik} = \begin{pmatrix} 1 & & & & k \\ & 1 & & & \\ i & & \ddots & & \\ & & & \ddots & \\ & & & & -l_{ik} & \ddots & \\ & & & & & & 1 \end{pmatrix} = I - l_{ik} E_{ik}$$

$E_{ik}$  is the matrix whose single element is  $e_{ik} = 1$ , and that otherwise consists of zeros, with  $l_{ik}$  from line (3) of the Gaussian elimination method.

# LU Decomposition

- Thus applies

$$\begin{aligned}\hat{L}_{N-1,N-2} \cdot \dots \cdot \hat{L}_{N-1,0} \cdot \dots \cdot \hat{L}_{2,0} \hat{L}_{1,0} A &= \\ = \hat{L}_{N-1,N-2} \cdot \dots \cdot \hat{L}_{N-1,0} \cdot \dots \cdot \hat{L}_{2,0} \hat{L}_{1,0} b\end{aligned}\tag{3}$$

and because of (2) applies

$$\hat{L}_{N-1,N-2} \cdot \dots \cdot \hat{L}_{N-1,0} \cdot \dots \cdot \hat{L}_{2,0} \hat{L}_{1,0} A = U.\tag{4}$$

# LU Decomposition: Properties

- There apply the following properties:

- 1  $\hat{L}_{ik} \cdot \hat{L}_{i',k'} = I - l_{ik}E_{ik} - l_{i'k'}E_{i'k'} \text{ for } k \neq i' (\Rightarrow E_{ik}E_{i'k'} = 0)$ .
- 2  $(I - l_{ik}E_{ik})(I + l_{ik}E_{ik}) = I \text{ für } k \neq i, \text{ thus } \hat{L}_{ik}^{-1} = I + l_{ik}E_{ik}$ .

- Because of 2 and the relationship (4)

$$A = \underbrace{\hat{L}_{1,0}^{-1} \cdot \hat{L}_{2,0}^{-1} \cdots \hat{L}_{N-1,0}^{-1} \cdots \cdots \hat{L}_{N-1,N-2}^{-1}}_{=:L} U = LU \quad (5)$$

- Because of 1, which also holds in its meaning for  $\hat{L}_{ik}^{-1} \cdot \hat{L}_{i'k'}$ ,  $L$  is a lower triangular matrix with  $L_{ik} = l_{ik}$  for  $i > k$  and  $L_{ii} = 1$ .
- The algorithm for  $LU$  decomposition of  $A$  is obtained by leaving out line (6) in the Gaussian algorithm above. The matrix  $L$  will be stored in the lower triangle of  $A$ .

# LU Decomposition: Parallel Variant with Row-wise Partitioning

Row-wise partitioning of a  $N \times N$  matrix for the **case  $N = P$** :

|       |  |  |  |  |  |  |  |  |
|-------|--|--|--|--|--|--|--|--|
| $P_0$ |  |  |  |  |  |  |  |  |
| $P_1$ |  |  |  |  |  |  |  |  |
| $P_2$ |  |  |  |  |  |  |  |  |
| $P_3$ |  |  |  |  |  |  |  |  |
| $P_4$ |  |  |  |  |  |  |  |  |
| $P_5$ |  |  |  |  |  |  |  |  |
| $P_6$ |  |  |  |  |  |  |  |  |
| $P_7$ |  |  |  |  |  |  |  |  |

- In step  $k$  processor  $P_k$  sends the matrix elements  $a_{k,k}, \dots, a_{k,N-1}$  to all processors  $P_j$  with  $j > k$ , and these eliminate in their row.
- Parallel runtime:

$$\begin{aligned} T_P(N) &= \sum_{\substack{m=N-1 \\ \text{Number of rows to eliminate}}}^1 (t_s + t_h + \underbrace{t_w \cdot m}_{\text{Rest of row } k}) \underbrace{\text{Id } N}_{\text{Broadcast}} + \underbrace{m2t_f}_{\text{Elimination}} \quad (6) \\ &= \frac{(N-1)N}{2} 2t_f + \frac{(N-1)N}{2} \text{Id } N t_w + N \text{Id } N (t_s + t_h) \\ &\approx N^2 t_f + N^2 \text{Id } N \frac{t_w}{2} + N \text{Id } N (t_s + t_h) \end{aligned}$$

# LU Decomposition: Analysis of Parallel Variant

- Sequential runtime of LU decomposition:

$$\begin{aligned} T_S(N) &= \sum_{m=N-1}^1 \underbrace{m}_{\text{rows are to elim.}} \underbrace{2mt_f}_{\text{Elim. of a row}} = \\ &= 2t_f \frac{(N-1)(N(N-1)+1)}{6} \approx \frac{2}{3} N^3 t_f. \end{aligned} \tag{7}$$

- As you can see from (6),  $N \cdot T_P = O(N^3 \ln N)$  (consider  $P = N!$ ) increases asymptotically faster than  $T_S = O(N^3)$ .
- The algorithm is thus not cost optimal (efficiency cannot be kept constant for  $P = N \rightarrow \infty$ ).
- The reason is, that processor  $P_k$  waits within its broadcast until all other processors have received the pivot row.
- We describe now an *asynchronous* variant, where a processor immediately starts calculating as soon as it receives the pivot row.

# LU Decomposition: Asynchronous Variant

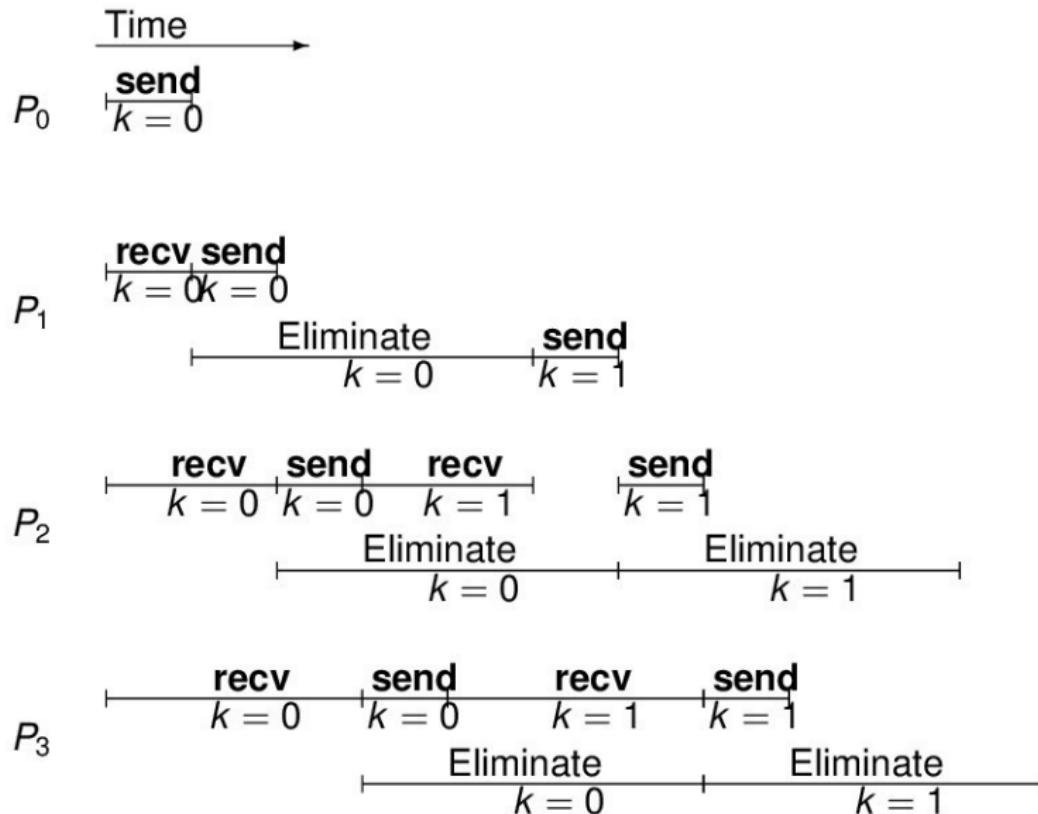
Program ( Asynchronous LU decomposition for  $P = N$ )

```
parallel lu-1
{
 const int N = ...;
 process Π[int p ∈ {0, ..., N - 1}]
 {
 double A[N];
 double rr[2][N]; // my row
 double *r; // buffer for pivot row
 msgid m;
 int j, k;

 if (p > 0) m =arecv(Πp-1, rr[0]);
 for (k = 0; k < N - 1; k++)
 {
 if (p == k) send(Πp+1, A);
 if (p > k)
 {
 while (!success(m)); // wait for pivot row
 if (p < N - 1) asend(Πp+1, rr[k%2]);
 if (p > k + 1) m =arecv(Πp-1, rr[(k + 1)%2]);
 r = rr[k%2];
 A[k] = A[k] / r[k];
 for (j = k + 1; j < N; j++)
 A[j] = A[j] - A[k] · r[j];
 }
 }
 }
}
```

# LU Decomposition: Temporal Sequence

How does the parallel algorithm behave over time?



## LU Decomposition: Parallel Runtime and Efficiency

- After a fill-in time of  $p$  message transmissions the pipeline is filled completely, and all processors are always busy with elimination. Then one obtains the following runtime ( $N = P$ , still!):

$$\begin{aligned} T_P(N) &= \underbrace{(N-1)(t_s + t_h + t_w N)}_{\text{fill-in time}} + \sum_{m=N-1}^1 \left( \underbrace{2mt_f}_{\text{elim.}} + \underbrace{t_s}_{\substack{\text{setup time} \\ (\text{compute+send parallel})}} \right) = \quad (8) \\ &= \frac{(N-1)N}{2} 2t_f + (N-1)(2t_s + t_h) + N(N-1)t_w \approx \\ &\approx N^2 t_f + N^2 t_w + N(2t_s + t_h). \end{aligned}$$

- The factor  $\text{Id } N$  of (6) is now vanished. For the efficiency we obtain

$$\begin{aligned} E(N, P) &= \frac{T_S(N)}{NT_P(N, P)} = \frac{\frac{2}{3}N^3 t_f}{N^3 t_f + N^3 t_w + N^2(2t_s + t_h)} = \quad (9) \\ &= \frac{2}{3} \frac{1}{1 + \frac{t_w}{t_f} + \frac{2t_s + t_h}{Nt_f}}. \end{aligned}$$

- The efficiency is such limited by  $\frac{2}{3}$ . The reason for this is, that processor  $k$  remains after  $k$  steps idle. This can be avoided by more rows per processor (coarser granularity).

# LU Decomposition: The Case $N \gg P$

LU decomposition for the **case  $N \gg P$** :

- Program 0.1 from above can be easily extended to the case  $N \gg P$ . Therefore the rows are distributed cyclically onto the processors  $0, \dots, P - 1$ . A processor's current pivot row is obtained from the predecessor in the ring.
- The parallel runtime is

$$\begin{aligned} T_P(N, P) &= \underbrace{(P-1)(t_s + t_h + t_w N)}_{\text{fill-in time of pipeline}} + \sum_{m=N-1}^1 \left( \underbrace{\frac{m}{P}}_{\text{rows per processor}} \cdot m2t_f + t_s \right) = \\ &= \frac{N^3}{P} \frac{2}{3} t_f + Nt_s + P(t_s + t_h) + NPt_w \end{aligned}$$

and thus one has the efficiency

$$E(N, P) = \frac{1}{1 + \frac{Pt_s}{N^2 \frac{2}{3} t_f} + \dots}.$$

## LU Decomposition: The case $N \gg P$

- Because of row-wise partitioning applies however in average, that some processors have a row more than others.
- A still better load balancing is achieved by a two-dimensional partitioning of the matrix. Herefore we assume that the segmentation of the row and column index set

$$I = J = \{0, \dots, N - 1\}$$

is done with the mappings  $p$  and  $\mu$  for  $I$  and  $q$  and  $\nu$  for  $J$ .

# LU decomposition: General Partitioning

- The following implementation is simplified, if we additionally assume, that the data partitioning fulfills the following monotony condition:

Ist  $i_1 < i_2$  und  $p(i_1) = p(i_2)$  such applies  $\mu(i_1) < \mu(i_2)$

Ist  $j_1 < j_2$  und  $q(j_1) = q(j_2)$  such applies  $\nu(j_1) < \nu(j_2)$

- Therefore an interval of global indices  $[i_{min}, N - 1] \subseteq I$  corresponds to a number of intervals of local indices in different processors, that can be calculated by:

Set

$$\tilde{I}(p, k) = \{m \in \mathbf{N} \mid \exists i \in I, i \geq k : p(i) = p \wedge \mu(i) = m\}$$

and

$$ibegin(p, k) = \begin{cases} \min \tilde{I}(p, k) & \text{if } \tilde{I}(p, k) \neq \emptyset \\ N & \text{otherwise} \end{cases}$$

$$iend(p, k) = \begin{cases} \max \tilde{I}(p, k) & \text{if } \tilde{I}(p, k) \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

- Then one can substitute a loop

**for** ( $i = k; i < N; i++$ ) ...

by local loops in the processors  $p$  of shape

**for** ( $i = ibegin(p, k); i \leq iend(p, k); i++$ ) ...

# LU Decomposition: General Partitioning

Analogous we perform with the column indices:

Set

$$\tilde{J}(q, k) = \{n \in \mathbf{N} \mid \exists j \in j, j \geq k : q(j) = q \wedge \nu(j) = n\}$$

and

$$jbegin(q, k) = \begin{cases} \min \tilde{J}(q, k) & \text{if } \tilde{J}(q, k) \neq \emptyset \\ N & \text{otherwise} \end{cases}$$

$$jend(q, k) = \begin{cases} \max \tilde{J}(q, k) & \text{if } \tilde{J}(q, k) \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Now we can go on with the implementation of the *LU* decomposition for a general data partitioning.

# LU Decomposition: Algorithm with General Partitioning

Program ( Synchronous LU decompositon with general data partitioning)

parallel lu-2

{  
    **const int**  $N = \dots, \sqrt{P} = \dots;$

process  $\Pi[\text{int } (p, q) \in \{0, \dots, \sqrt{P}-1\} \times \{0, \dots, \sqrt{P}-1\}]$

{  
    **double**  $A[N/\sqrt{P}][N/\sqrt{P}], r[N/\sqrt{P}], c[N/\sqrt{P}];$   
    **int**  $i, j, k;$

**for** ( $k = 0; k < N - 1; k++$ )

{

$I = \mu(k); J = \nu(k);$  // local indices

// distribute pivot row:

**if** ( $p == p(k)$ )

{

**for** ( $j = jbegin(q, k); j \leq jend(q, k); j++$ ) // I have pivot row

$r[j] = A[I][j];$

            Send  $r$  to all processors ( $x, q$ )  $\forall x \neq p$

}

**else recv**( $\Pi p(k), q, r$ );

// distribute pivot column:

**if** ( $q == q(k)$ )

{

**for** ( $i = ibegin(p, k + 1); i \leq iend(p, k + 1); i++$ ) // I have part of column k

$c[i] = A[i][J] = A[i][J]/r[J];$

            Send  $c$  to all processors ( $p, y$ )  $\forall y \neq q$

}

**else recv**( $\Pi p, q(k), c$ );

// elimination:

**for** ( $i = ibegin(p, k + 1); i \leq iend(p, k + 1); i++$ )

**for** ( $j = jbegin(q, k + 1); j \leq jend(q, k + 1); j++$ )

$A[i][j] = A[i][j] - c[i] \cdot r[j];$

}

# LU Decomposition: Analysis I

- Let us analyse this implementation (synchronous variant):

$$\begin{aligned} T_P(N, P) &= \sum_{m=N-1}^1 \underbrace{\left( t_s + t_h + t_w \frac{m}{\sqrt{P}} \right)}_{\text{Broadcast pivot row/-column}} \text{Id } \sqrt{P} 2 + \left( \frac{m}{\sqrt{P}} \right)^2 2t_f = \\ &= \frac{N^3}{P} \frac{2}{3} t_f + \frac{N^2}{\sqrt{P}} \text{Id } \sqrt{P} t_w + N \text{Id } \sqrt{P} 2(t_s + t_h). \end{aligned}$$

- Mit  $W = \frac{2}{3}N^3 t_f$ , d.h.  $N = \left(\frac{3W}{2t_f}\right)^{\frac{1}{3}}$ , gilt

$$T_P(W, P) = \frac{W}{P} + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \text{Id } \sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} + W^{\frac{1}{3}} \text{Id } \sqrt{P} \frac{3^{1/3} 2(t_s + t_h)}{(2t_f)^{\frac{1}{3}}}.$$

# LU Decomposition: Analysis II

- The isoefficiency function can be obtained from  $PT_P(W, P) - W \stackrel{!}{=} KW$ :

$$\begin{aligned} & \sqrt{P}W^{\frac{2}{3}} \text{Id} \sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} = KW \\ \iff & W = P^{\frac{3}{2}} (\text{Id} \sqrt{P})^3 \frac{9t_w^3}{4t_f^2 K^3} \end{aligned}$$

thus

$$W \in O(P^{3/2}(\text{Id} \sqrt{P})^3).$$

- Program 0.2 can also be realized in an asynchronous variant. Hereby the communication shares can be effectively hidden behind the calculation.

# LU Decomposition: Pivoting

- The  $LU$  factorisation of general, invertible matrices requires pivoting and is also meaningful by reasons of minimisation of rounding errors.
- One speaks of full pivoting, if the pivot element in step  $k$  can be chosen from all  $(N - k)^2$  remaining matrix elements, resp. of partial pivoting, if the pivot element can only be chosen from a part of the elements. Usual for example is the maximal row- or column pivot this means one chooses  $a_{ik}$ ,  $i \geq k$ , with  $|a_{ik}| \geq |a_{mk}| \quad \forall m \geq k$ .
- The implementation of  $LU$  decomposition has now to consider the choice of the new pivot element during the elimination. Therefore one has two possibilities:
  - ▶ Explicit exchange of rows and/or columns: Here a rest of the algorithm then remains unchanged (for row exchanges the righthand side has to be permuted).
  - ▶ The actual data is not moved, but one remembers the interchange of indices (in an integer array, that maps old indices to new).

# LU Decomposition: Pivoting

- The parallel versions have different properties regarding pivoting.  
The following points have to be considered for the parallel LU partitioning with partial pivoting:
  - ▶ If the area, in which the pivot element is searched, is stored in a single processor (e.g. row-wise partitioning with maximal row pivot), then the search is to be performed purely sequential. In the other case it can be parallelized.
  - ▶ But this parallel search for a pivot element requires communication (and such synchronisation), that renders the pipelining in the asynchronous variant impossible.
  - ▶ To permute the indices is faster than explicit exchange, especially if the exchange requires data exchange between processors. Besides that a favourable load balancing can such be destroyed, if randomly the pivot elements reside always in the same processor.
- A quite good compromise is given by the row-wise cyclic partitioning with maximal row pivot and explicit exchange, since:
  - ▶ pivot search in row  $k$  is pure sequential, but needs only  $O(N - k)$  operations (compared to  $O((N - k)^2 / P)$  for the elimination); besides the pipelining is not destroyed.
  - ▶ explicit exchange requires only communication of the index of the pivot column, but no exchange of matrix elements between processors. The pivot column index is sent with the pivot row.
  - ▶ load balancing is not influenced by the pivoting.

# LU Decomposition: Solution of Triangular Systems

- We assume the matrix  $A$  be factorized into  $A = LU$  as above, and continue with the solution of the system of the form

$$LUx = b. \quad (10)$$

This happens in two steps:

$$Ly = b \quad (11)$$

$$Ux = y. \quad (12)$$

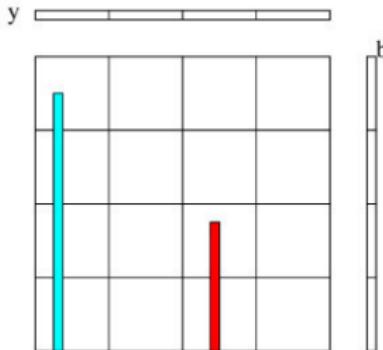
- We shortly consider the sequential algorithm:

```
// Ly = b:
for (k = 0; k < N; k++) {
 y_k = b_k; l_{kk} = 1
 for (i = k + 1; i < N; i++)
 b_i = b_i - a_{ik}y_k;
}
// Ux = y:
for (k = N - 1; k ≥ 0; k--) {
 x_k = y_k / a_{kk}
 for (i = 0; i < k; i++)
 y_i = y_i - a_{ik}x_k;
}
```

- This is a column oriented version, since after calculation of  $y_k$  resp.  $x_k$  immediately the righthand side is modified for all indices  $i > k$  resp.  $i < k$ .

# LU Decomposition: Parallelisation

- The parallelisation has of course to be oriented at the data partitioning of the LU decomposition (if one wants to avoid copying, which seems not to be meaningful because of  $O(N^2)$  data and  $O(N^2)$  operations). We consider for this a two-dimensional block-wise partitioning of the matrix:



- The sections of  $b$  are copied across processors rows and the sections of  $y$  are copied across the processor columns. Obviously after calculation of  $y_k$  only the processors of column  $q(k)$  can be busy with the modification of  $b$ . According to that during the solution of  $Ux = y$  only the processors  $(*, q(k))$  can be busy at a time. Thus, with a row-wise partitioning ( $Q = 1$ ) always all processors can be kept busy.

# LU Decomposition: Parallelisation for General Partitioning

Program (Resolving of  $LUX = b$  for general data partitioning)

```
parallel lu-solve
{
 const int N = . . . ;
 const int √P = . . . ;
 process Π[int (p, q) ∈ {0, . . . , √P − 1} × {0, . . . , √P − 1}]
 {
 double A[N / √P][N / √P];
 double b[N / √P]; x[N / √P];
 int i, j, k, l, K;

 // Solve Ly = b, store y in x.
 // b column-wise distributed onto diagonal processors.
 if (p == q) send b to all (p, *);
 for (k = 0; k < N; k++)
 {
 l = μ(k); K = ν(k);
 if(q(k) == q)
 // only they have something to do
 {
 if (k > 0 ∧ q(k) ≠ q(k − 1))
 // need current b
 recv(Πp, q(k−1), b);
 if (p(k) == p)
 // have diagonal element
 // store y in x!
 {
 x[K] = b[l];
 send x[K] to all (*, q);
 }
 else recv(Πp(k), q(k), x[K]);
 for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i++)
 b[i] = b[i] − A[i][K] · x[K];
 if (k < N − 1 ∧ q(k + 1) ≠ q(k))
 send(Πp, q(k+1), b);
 }
 }
 ...
 }
}
```

# LU Decomposition: Parallelisation

Program (Resolving of  $LUX = b$  for general data partitioning cont.)

parallel lu-solve cont.

{

// {  
// }  
// {  
// It is such to copy  $x$  into  $b$ , where  $b$  shall be distributed row-wise and copied column-wise.  
for ( $i = 0; i < N/\sqrt{P}; i++$ )

// extinguish

$b[i] = 0;$

    for ( $j = 0; j < N - 1; j++$ )

        if ( $q(j) = q \wedge p(j) = p$ )

// one has to be it

$b[\mu(j)] = x[\nu(j)];$

sum  $b$  across all  $(p, *)$ , result in  $(p, p)$ ;

// Resolving of  $UX = y$  ( $y$  is stored in  $b$ )

if ( $p == q$ ) send  $b$  and all  $(p, *)$ ;

for ( $k = N - 1; k \geq 0; k--$ )

{

$I = \mu(k); K = \nu(k);$

    if ( $q(k) == q$ )

{

        if ( $k < N - 1 \wedge q(k) \neq q(k + 1)$ )

            recv( $\Pi_{p, q(k+1)}, b$ );

        if ( $p(k) == p$ )

{

$x[K] = b[I]/A[I][K];$

            send  $x[K]$  to all  $(*, q)$ ;

}

        else recv( $\Pi_{p(k), q(k)}, x[K]$ );

        for ( $i = ibegin(p, 0); i \leq iend(p, 0); i++$ )

$b[i] = b[i] - A[i][K] \cdot x[K];$

        if ( $k > 0 \wedge q(k) \neq q(k - 1)$ )

            send( $\Pi_{p, q(k-1)}, b$ );

}

}

}

# LU Decomposition: Parallelisation

- Since at a time always only  $\sqrt{P}$  processors are busy, the algorithm cannot be cost optimal. The total scheme consisting of *LU* decomposition and solution of triangular systems can still always be scaled iso-efficiently, since the sequential complexity of solution is only  $O(N^2)$  compared to  $O(N^3)$  for the factorisation.
- If one needs to solve the equation system for many righthand sides, one should use a rectangular processor array  $P \times Q$  with  $P > Q$ , or in the extreme case choose as  $Q = 1$ . If pivoting has been required, this was already a meaningful configuration.

# Particle Methods I + II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

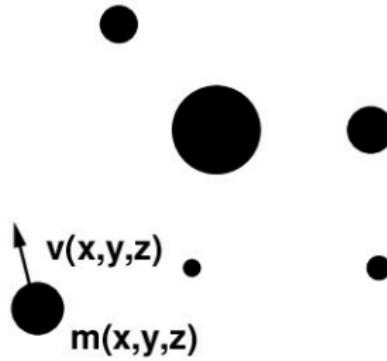
## Particle methods

- Problem formulation
- Standard method
- Parallelisation
- Improvement of the method

# Particle Methods

Task formulation:

- With particle methods one simulates the movement of  $N$  particles (or bodies) that move under the influence of a force field.
- The force field itself depends again on the position of the particles.
- The particles are characterised by point masses  $m(x, y, z)$  and move with velocity  $v(x, y, z)$  in the system.



# The $N$ -Body Problem I

$N$ -Body problem:

- Given being  $N$  point shaped masses  $m_1, \dots, m_N$  at positions  $x_1, \dots, x_N \in \mathbb{R}^3$ .
- The gravitational force, exerted from body  $j$  onto body  $i$ , is given by Newton's law of gravitation

$$F_{ij} = \underbrace{\frac{\gamma m_i m_j}{\|x_j - x_i\|^2}}_{\text{strength}} \cdot \underbrace{\frac{x_j - x_i}{\|x_j - x_i\|}}_{\text{unit vector of } x_i \text{ in direction } x_j} \quad (1)$$



- The gravitational force can be written as potential gradient:

$$F_{ij} = m_i \frac{\gamma m_j (x_j - x_i)}{\|x_j - x_i\|^3} = m_i \nabla \left( \frac{\gamma m_j}{\|x_j - x_i\|} \right) = m_i \nabla \phi_j(x_i). \quad (2)$$

$\phi_y(x) = \frac{\gamma m}{\|y-x\|}$  is denoted gravitational potential of the mass  $m$  at position  $y \in \mathbb{R}^3$ .

## The $N$ -Body Problem II

- The movement equations of the considered  $N$  bodies are a consequence of the law force=mass $\times$ acceleration:

für  $i = 1, \dots, N$

$$m_i \frac{dv_i}{dt} = m_i \nabla \left( \sum_{j \neq i} \frac{\gamma m_j}{\|x_j - x_i\|} \right) \quad (3)$$

$$\frac{dx_i}{dt} = v_i \quad (4)$$

Here is  $v_i(t) : \mathcal{R} \rightarrow \mathcal{R}^3$  the velocity of body  $i$  and  $x_i(t) : \mathcal{R} \rightarrow \mathcal{R}^3$  the position in dependence of time.

- We get thus a system of ordinary differential equations of dimension  $6N$  (three space dimensions). For its solution are still initial conditions for position and velocity required:

$$x_i(0) = x_i^0, \quad v_i(0) = v_i^0, \quad \text{for } i = 1, \dots, N \quad (5)$$

# Numerical Computation

- The integration of the movement equations (3) and (4) is performed numerically, since only for  $N = 2$  analytical solutions are possible (Kegelcuts, Kepler's laws).
- The simplest method is the explicit Euler method:

$$t^k = k \cdot \Delta t,$$

Discretisation in time:  $v_i^k = v_i(t^k),$   
 $x_i^k = x_i(t^k).$

$$\left. \begin{aligned} v_i^{k+1} &= v_i^k + \Delta t \cdot \nabla \left( \sum_{j \neq i} \underbrace{\frac{\gamma m_j}{\|x_j^k - x_i^k\|}}_{\text{"explicit"}} \right) \\ x_i^{k+1} &= x_i^k + \Delta t \cdot v_i^k \end{aligned} \right\} \text{for } i = 1, \dots, N \quad (6)$$

# Problematique of Force Evaluation

- Surely there are better methods than the explicit Euler method, that only has a convergence order of  $O(\Delta t)$ , for the parallelisation this does not have a large impact, since the structure of other schemes is similar.
- The problem of force evaluation:  
In the force calculation the force of a body  $i$  depends on the position of *all* other bodies  $j \neq i$ . The effort for a force evaluation (that is at least once necessary for each time step) increases such as  $O(N^2)$ . In practical applications  $N$  can be in the range  $10^6, \dots, 10^9$ , which means an enormous computing effort.
- The main point of this chapter is therefore the presentation of improved sequential algorithms for fast force evaluation. These calculate the forces approximately with an effort of  $O(N \log N)$  (There are methods with a complexity of  $O(N)$  too, that we leave away for time reasons).

# Parallelisation of the Standard Method

- The  $O(N^2)$  algorithm is quite simple to parallelise. Each of the  $P$  processors gets  $\frac{N}{P}$  bodies. To calculate the forces for all its bodies, the processor needs all other bodies. Therefore the data is shifted cyclic once through a ring topology.
- Analyse:

$$\begin{aligned}T_S(N) &= c_1 N^2 \\T_P(N, P) &= c_1 P \left( \underbrace{\frac{N}{P} \cdot \frac{N}{P}}_{\text{block } p \text{ with } q} \right) + \underbrace{c_2 P \frac{N}{P}}_{\text{communication}} = \\&= c_1 \frac{N^2}{P} + c_2 N \\E(P, N) &= \frac{c_1 N^2}{\left( c_1 \frac{N^2}{P} + c_2 N \right) P} = \frac{1}{1 + \frac{c_2}{c_1} \cdot \frac{P}{N}}\end{aligned}$$

constant efficiency for  $N = O(P)$ .

- Therefore is the iso-efficiency function because of  $W = c_1 N^2$

$$W(P) = O(P^2)$$

(of course in relation to the sub-optimal standard algorithm).

# Fast Multipole Methods

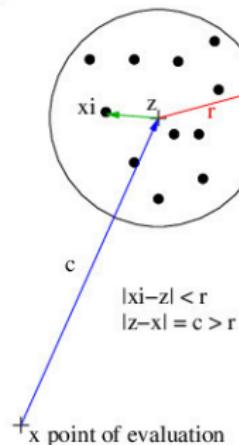
## Fast Multipole Methods:

- The first basic idea has been published by *Pincus und Scherega* in 1977. Essential idea was the representation of a group of particles by a abstraction called pseudo particle. This represents the group properties and thus the resulting potential. The relationship with another particle group can then be calculated with a single multipole expansion.
- A second concept is the hierarchical coarsening of space in separate sub-areas.
- Both methods have been combined by Appel, 1985 and Barnes and Hut, 1986 within a single algorithm. The effort has a complexity of  $O(N \log N)$ .
- The fast multipole method has been published in 1987 by Greengard and Rokhlin. To both mentioned ideas they further introduce a local expansion of potentials. In specific cases, e.g. for uniform particle distribution, the effort reduces to  $O(N)$ .

# Fast Summation Methods I

Accelerated method for force evaluation:

- We consider the figured cluster of bodies:  $M$  mass points be contained in a circle around  $z$  with radius  $r$ . We evaluate the potential  $\phi$  of all mass points in point  $x$  with  $\|z - x\| = c > r$ .



- Let us consider first a mass point at position  $\xi$  with  $\|\xi - z\| < r$ . The potential of mass in  $\xi$  be (the multiplicative factor  $\gamma m$  is neglected)

$$\phi_\xi(x) = \frac{1}{\|\xi - x\|} = f(\xi - x).$$

# Fast Summation Methods II

- The potential depends only on the distance  $\xi - x$ .
- Now we insert the point  $z$  and develop in a Taylor series around  $(z - x)$  up to order  $p$  (do not interchange with processors):

$$\begin{aligned} f(\xi - x) &= f((z - x) + (\xi - z)) = \\ &= \sum_{|\alpha| \leq p} \frac{D^\alpha f(z - x)}{|\alpha|!} (\xi - z)^{|\alpha|} + \underbrace{\sum_{|\alpha|=p+1} \frac{D^\alpha f(z - x + \theta(\xi - z))}{|\alpha|!} (\xi - z)^{|\alpha|}}_{\text{remainder term}} \end{aligned}$$

for a  $\theta \in [0, 1]$ . Important is the separation of variables  $x$  and  $\xi$ .

- The size of the error (remainder term) depends on  $p$ ,  $r$  and  $c$ .
- How can the series expansion be used to accelerate the potential evaluation?
- Herefore we assume that the evaluation of the potential of  $M$  masses is to be computed at  $N$  points, which normally would require  $O(MN)$  operations.

# Fast Summation Methods III

- For the evaluation of the potential at the position  $x$  we calculate

$$\begin{aligned}\phi(x) &= \sum_{i=1}^M \gamma m_i \phi_{\xi_i}(x) = \sum_{i=1}^M \gamma m_i f((z-x) + (\xi_i - z)) \approx \\ &\underset{\text{(taylor series up to order } p\text{)}}{\approx} \sum_{i=1}^M \gamma m_i \sum_{|\alpha| \leq p} \frac{D^\alpha f(z-x)}{|\alpha|!} (\xi_i - z)^{|\alpha|} = \\ &\underset{\text{(permute sum)}}{=} \sum_{|\alpha| \leq p} \frac{D^\alpha f(z-x)}{|\alpha|!} \underbrace{\left( \sum_{i=1}^M \gamma m_i (\xi_i - z)^{|\alpha|} \right)}_{=: M_\alpha, \text{ independent of } x!} = \\ &= \sum_{|\alpha| \leq p} \frac{D^\alpha f(z-x)}{|\alpha|!} M_\alpha\end{aligned}$$

- The calculation of coefficients  $M_\alpha$  requires once  $O(Mp^3)$  operations.
- Are the  $M_\alpha$  known, then a evaluation of  $\phi(x)$  costs  $O(p^5)$  operations.
- For evaluation at  $N$  points we get such the total complexity of  $O(Mp^3 + Np^5)$ .

# Fast Summation Methods IV

- It is clear that the potential, calculated in this way, is not exact. The algorithm only makes sense, if the error can be controlled such that it is neglectable (e.g. smaller as the discretization error).
- A criteria for error control provides the error estimation:

$$\frac{\phi_\xi(x) - \sum_{|\alpha| \leq p} \frac{D^\alpha f(z-x)}{|\alpha|!} (\xi-z)^{|\alpha|}}{\phi_\xi(x)} \leq c(2h)^{p+1},$$

with  $\frac{r}{c} \leq h < \frac{1}{4}$ . For the case  $c > 4r$  the error reduces by  $(1/2)^{p+1}$ .

- The approximation is then the more accurate
  - the smaller  $\frac{r}{c}$
  - the larger the degree of expansion  $p$ .

# Gradient Calculation

- In the  $N$ -body problem one does not want to calculate the potential, but the force, thus the gradient of the potential.
- This works via

$$\frac{\partial \phi(x)}{\partial x_{(j)}} \underset{\substack{\text{series dev.} \\ \uparrow \\ \text{space dimension}}}{\approx} \frac{\partial}{\partial x_{(j)}} \sum_{|\alpha| \leq p} \frac{D^\alpha f(z-x)}{|\alpha|!} M_\alpha = \sum_{|\alpha| \leq p} \frac{D^\alpha \partial_{x_{(j)}} f(z-x)}{|\alpha|!} M_\alpha$$

- One has thus only to calculate  $D^\alpha \partial_{x_{(j)}} f(z-x)$  instead  $D^\alpha f(z-x)$ .
- Above we have used a Taylor series. This is not the only possibility of a series expansion. Besides there are for the considered potentials  $\frac{1}{\log(\|\xi-x\|)}$  (2D) and  $\frac{1}{\|\xi-x\|}$  (3D) other expansions, the so called multipole expansions, that are fitting better.
- For this series a better error estimation applies in the sense, that they have the form

$$\text{error} \leq \frac{1}{1 - \frac{r}{c}} \left(\frac{r}{c}\right)^{p+1}$$

and therefore are already for  $c > r$  satisfied.

- Moreover the complexity in relation to  $p$  is better ( $p^2$  in 2D,  $p^4$  in 3D).

# Gradient Calculation

- An approximation of the gravitation potential, that is often used by physicians, is a taylor expansion of

$$\phi(x) = \sum_{i=1}^M \frac{\gamma m_i}{\|(s - x) + (\xi_i - s)\|},$$

where  $s$  is the *point of gravity* of  $M$  masses (and not a fictitious circle midpoint).

- The so-called monopole expansion reads

$$\phi(x) \approx \frac{\sum_{i=1}^M \gamma m_i}{\|s - x\|}$$

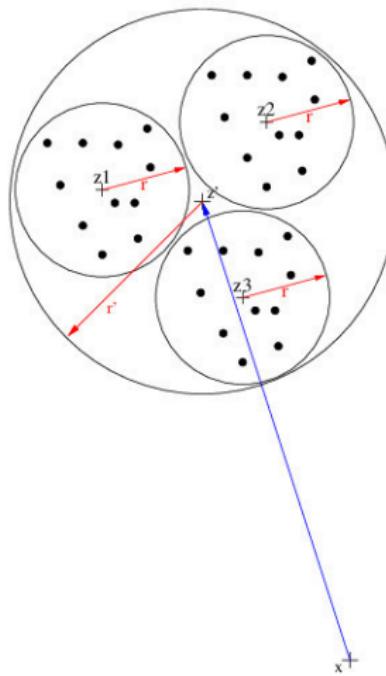
(this means a body of mass  $\sum m_i$  in  $s$ ).

- The accuracy is then only controlled by the relationship  $r/c$ .

# Shifting of an Expansion

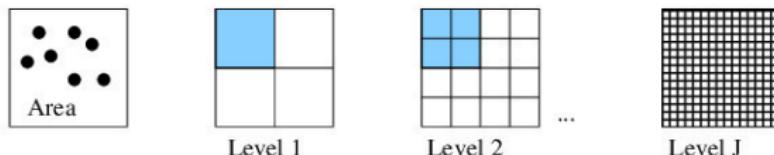
- In the following algorithms we still need a further tool, that concerns the shifting of expansions.
- The mapping shows three clusters of bodies in circles around  $z_1, z_2, z_3$  each with radius  $r$ . The three circles are contained in a larger circle around  $z'$  with radius  $r'$ .
- If we want to evaluate the potential of all masses in  $x$  with  $\|x - z'\| > r'$ , we could use the series expansion around  $z'$ .
- If already series expansions have been calculated in the three smaller circles (this means the coefficients  $M_\alpha$ ), then the expansion coefficients of the new series can be computed from the one of the old series with an effort of  $O(p^\alpha)$ , thus independent of the number of masses.
- Here *no* additional error arises, and it applies also the error estimation with appropriate larger  $r'$ .

# Shifting of an Development



# Uniform Point Distribution

- First we develop an algorithm, that is usable for a uniform point distribution. This has the advantage of simpler data structures and the possibility of simpler load balancing. We present the ideas for the two-dimensional case, since this can be drawn easier. However, all can be performed analogous for three space dimensions.
- All bodies be contained in a square  $\Omega = (0, D_{max})^2$  of side length  $D_{max}$ . We now introduce for  $\Omega$  a hierarchy of steadily finer grids. Level  $l$  is developed from level  $l - 1$  by quatering of elements.



$ID\{children\}(b)$



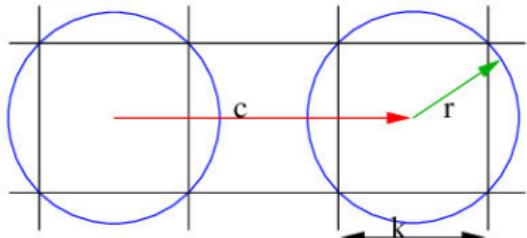
$b = ID\_V(b') \text{ for all } b' \text{ in } ID\_K(b)$

construction of grids

# Uniform Point Distribution

- If we want  $s$  bodies per fine grid box, then applies  $J = \log_4 \left(\frac{N}{s}\right)$ . For two not-neighbored squares we get the following estimation for the  $r/c$  relationship (masses in  $b$ , evaluation in  $a$ )

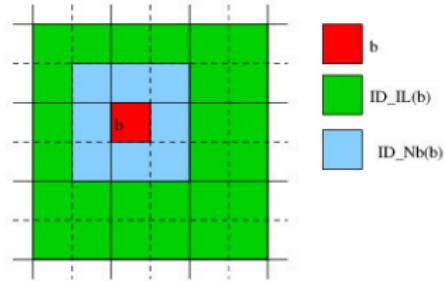
$$\begin{aligned}r &= \sqrt{2} \frac{k}{2} \\c &= 2k \\\Rightarrow \frac{r}{c} &= \frac{\sqrt{2} k}{4k} = \frac{\sqrt{2}}{4} \approx 0.35.\end{aligned}$$



$r/c$  evaluation for two not-neighbored squares

- For an element  $b$  on level  $l$  one defines the following regions in the neighborhood of  $b$ :

- $Nb(b) = \text{alle neighbors } b'$  of  $b$  on level  $l$  ( $\partial b \cap \partial b' \neq \emptyset$ ).
- $IL(b) = \text{interaction list of } b:$  children of neighbors of  $\text{father}(b)$ , that are not neighbors of  $b$ .



# Uniform Point Distribution

The following algorithm calculates the potential at all positions  $x_1, \dots, x_N$ :

## 1. Preparation phase:

For each fine grid box calculate a far field expansion;

effort

$$O\left(\frac{N}{s} sp^\alpha\right)$$

For all levels  $l = J - 1, \dots, 0$

For each box  $b$  on level  $l$

calculate expansion in  $b$  from expansion in  $\text{children}(b)$ ;

$$O\left(\frac{N}{s} sp^\gamma\right)$$

## 2. Evaluation phase:

For each fine grid box  $b$

For each body  $q$  in  $b$

{

calculate exact potential of all  $q' \in B, q' \neq q$ ;

$$O(Ns)$$

For all  $b' \in Nb(b)$

For all  $q' \in b'$

calculate potential of  $q'$  in  $q$ ;

$$O(N8s)$$

$\bar{b} = b$ ;

For all levels  $l = J, \dots, 0$

{

For all  $b' \in IL(\bar{b})$

Evaluate the far field expansion of  $b'$  in  $q$ ;

$$O\left(\frac{N}{s} sp^\beta\right)$$

}

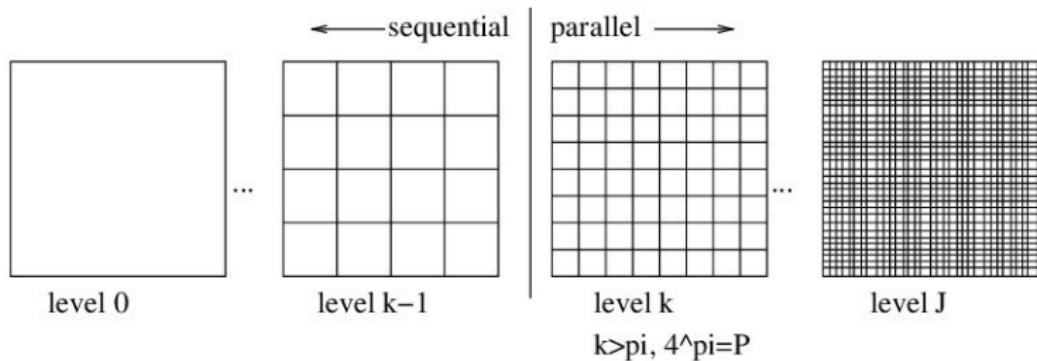
}

# Uniform Point Distribution

- Total effort:  $O(N \log N p^\gamma + Ns + Np^\alpha + \frac{N}{s} p^\beta)$ , thus asymptotically  $O(N \log N)$ .
- Here denotes  $\alpha$  the exponents for the building of the far field expansion and  $\beta$  the exponents for shifting.
- One considers, that one has  $N/s$  bodies per box on level  $J$  because of the uniform distribution.
- The accuracy is controlled here by the expansion degree  $p$ , the relationship  $r/c$  is fixed.

# Parallelisation: Load Balancing

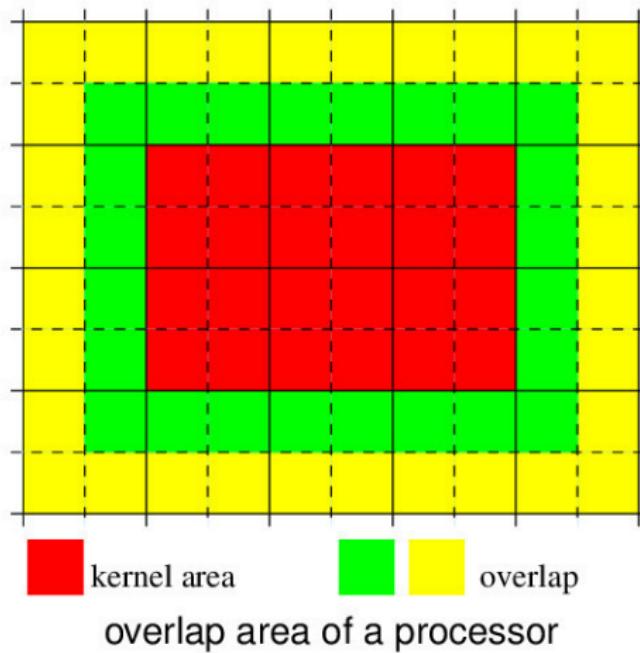
- Load balancing:  
The grid with the associated bodies is distributed onto the processors.
- Since we now have a hierarchy of grids we progress as follows:
  - ▶ Each processor shall at least get  $2 \times 2$  grid cells.
  - ▶ Be  $P = 4^\pi$ , then choose  $k = \pi + 1$  and distribute the grid on level  $k$  onto all processors (each has  $2 \times 2$  elements).
  - ▶ All levels  $l < k$  are stored on all processors.
  - ▶ All levels  $l > k$  are distributed such, that  $\text{children}(b)$  are processed in the same processor than  $b$ .
  - ▶ Example for  $P = 16 = 4^2$ .



Distribution of boxes for the parallelisation

# Parallelisation: Overlap

- Additional to the assigned elements each processor stores further an overlap area of two element rows:



# Parallelisation: Analysis I

- Since each has at least  $2 \times 2$  elements, the overlap area lies always in directly neighbored processors.
- The evaluation of  $IL$  requires at most communication with nearest neighbors.
- To establish the far field expansion for the levels  $l < k$  is a all-to-all communication required.
- Scalability estimation: Be  $\frac{N}{sP} \gg 1$ . Because of

$$J = \log_4 \left( \frac{N}{s} \right) = \log_4 \left( \frac{N}{sP} P \right) = \underbrace{\log_4 \left( \frac{N}{sP} \right)}_{J_p} + \underbrace{\log_4 P}_{J_s}$$

$J_s$  levels are computed sequentially and  $J_p = O(1)$  levels in parallel.

# Parallelisation: Analysis II

- Then we get for *fixed expansion degree*:

$$\begin{aligned} T_P(N, P) &= \underbrace{\left(\frac{N}{P} = \text{const.}\right)}_{\substack{\text{FFE level } J \dots K \\ \text{evaluate near field}}} + \underbrace{c_2 \text{Id } P + c_3 P}_{\substack{\text{all-to-all. This is always} \\ \text{data for four cells}}} + \underbrace{c_4 p}_{\substack{\text{compute FFE in whole } \Omega \\ \text{for } l = k-1 \dots 0 \text{ in all} \\ \text{processors}}} + \underbrace{c_5 J_p \frac{N}{P}}_{\substack{\text{FFE levels } l \geq k}} + \underbrace{c_5 \frac{N}{P} J_s}_{\substack{\text{FFE levels } l < k}} \end{aligned}$$

- Thus:

$$\begin{aligned} E(N, P) &= \frac{c_5 N \log N}{\underbrace{(c_5 \frac{N}{P} (J_s + J_p) + (c_3 + c_4)P + c_2 \text{Id } P)}_{\substack{\log N \\ \text{all to all coarse} \\ \text{grid FFE}}} + \underbrace{c_1 \frac{N}{P}}_{\substack{\text{nearfield local FFE}}})P} = \\ &= \frac{1}{1 + \frac{c_3 + c_4}{c_5} \cdot \frac{P^2}{N \log N} + \frac{c_2}{c_5} \cdot \frac{P \text{Id } P}{N \log N} + \frac{c_1}{c_5} \cdot \frac{1}{\log N}} \end{aligned}$$

- For an iso-efficient scaling  $N$  has thus to grow nearly like  $P^2$ !

# Parallelisation: Irregular Distribution

- The assumption of a uniform distribution of bodies is in some application (e.g. astro physics) not realistic.
- If we want to have in each fine grid box exactly a single body and is  $D_{min}$  the minimal distance between two bodies, then one needs a grid with

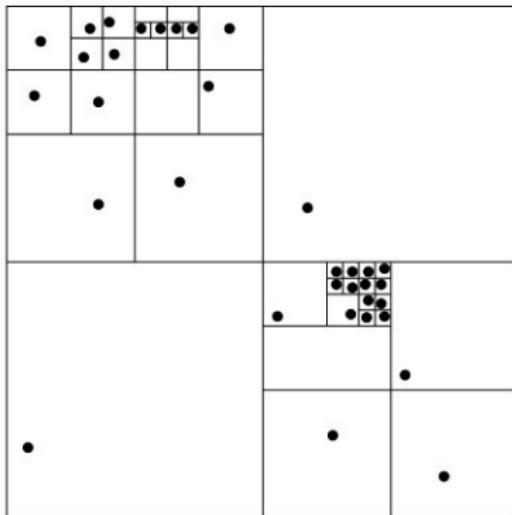
$$\log L = \log \frac{D_{max}}{D_{min}}$$

grid levels.  $L$  is called „*separation ratio*“.

- But from these the most are empty. As with sparse matrices one now avoids to store the empty cells. In two space dimensions this construction is called „adaptive quadtree“.
- The adaptive quadtree is constructed in the following way:
  - ▶ Initialisation: root contains all bodies in the square  $(0, D_{max})$ .
  - ▶ As long as a leaf  $b$  with more than two bodies exists:
    - ★ Subdivide  $b$  into four parts
    - ★ Put each body of  $b$  into the appropriate child
    - ★ Leave out empty children.

# Parallelisation: Irregular Distribution

- Example of an adaptive quadtree:



- The effort amounts (sequentially) to  $O(N \log L)$ .
- The first (successful) fast evaluation algorithm for irregular distributed bodies has been proposed by Barnes and Hut
- As in the uniform case the far field expansion is constructed from the leafs up to the root (at Barnes & Hut: monopole expansion).

# Irregular Distribution

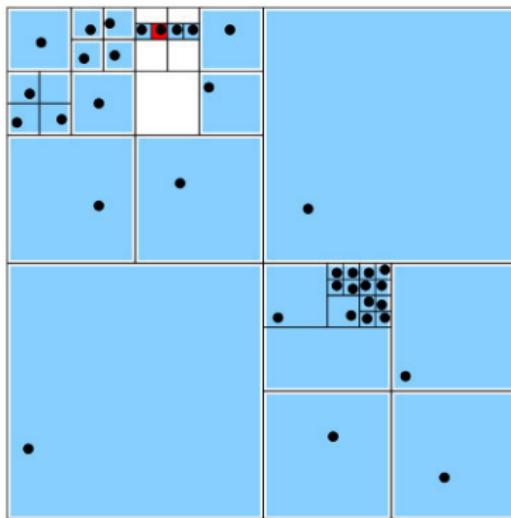
- For a body  $q$  one can then calculate the potential in  $q$  with the following recursive algorithm:

```
Pot(body q, box b)
{
 double pot = 0;
 if (b is leaf \wedge q = b.q) return 0; // end
 if (children(b) == \emptyset)
 return $\phi(b.q, q)$; // direct evaluation
 if ($\frac{r(b)}{\text{dist}(q, b)} \leq h$)
 return $\phi_b(q)$; // evaluate FFE
 for ($b' \in \text{children}(b)$)
 pot = pot + Pot(q, b'); // recursive descent
 return pot;
}
```

- For the calculation  $Pot$  is called with  $q$  and the root of the quadtree.
- In the algorithm of Barnes & Hut the accuracy of the evaluation is controlled with the parameters  $h$ .

# Irregular Distribution

Which cells of the quadtree are visited in the Barnes & Hut algorithm?



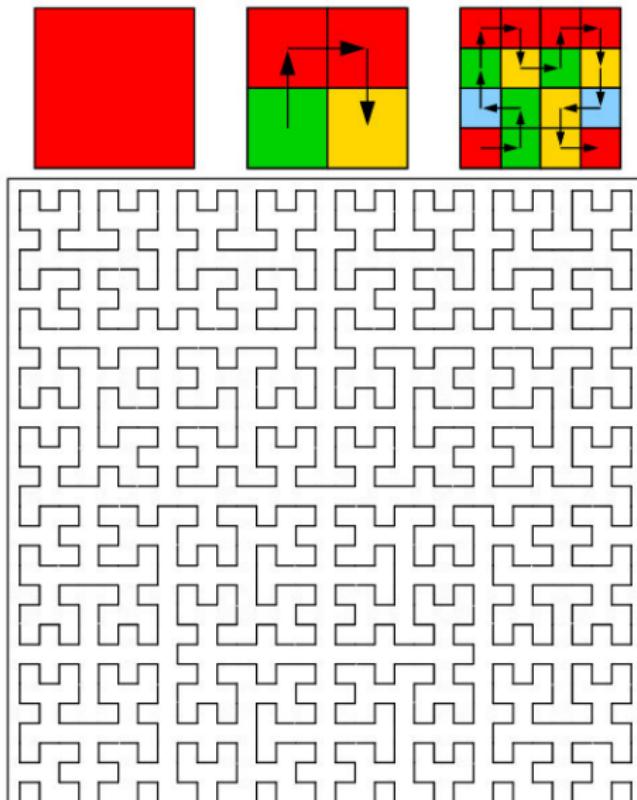
Example to the evaluation in the Barnes & Hut algorithm

# Irregular Distribution: Parallelisation I

- The parallelisation of this algorithm is relatively complex, such that we can only provide some hints. For details we point out Salmon, 1994.
- Since the positions of the particle change with time, the adaptive quadtree has to be constructed in each time step. Furthermore the partitioning of the bodies onto the processors has to be done in such a way, that close neighboring bodies are also stored in close neighboring processors.

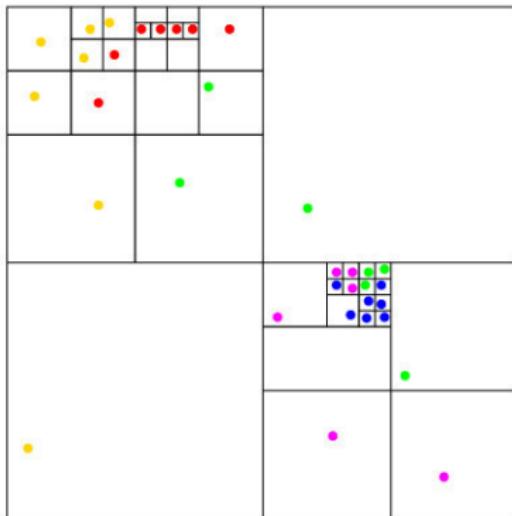
# Irregular Distribution: Parallelisation II

- A very fancy load balancing method works with „space filling curves“. The so called Peano or Hilbert curves have the following shape:



# Irregular Distribution: Parallelisation III

- A Hilbert curve of appropriate depth can be used to find a linear ordering of the bodies (resp. the leafs of in the quad tree). This can then easily be partitioned into  $P$  sections of length  $N/P$ .



- Salmon & Warren show that with this data distribution the adaptive quadtree can be constructed in parallel with few communication. Similar to the uniform algorithm the coarse grid information, that all processors store, can then be constructed by an all-to-all communication.

# Iterative Solution of Sparse Equation Systems

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

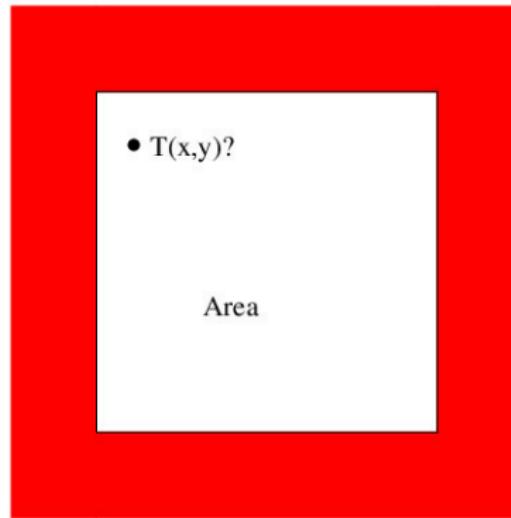
## Iterative Solution of Sparse Linear Equation Systems

- Problem formulation
- Iteration methods
- Parallelisation
- Multigrid methods
- Parallelisation of multigrid methods

# Problem Formulation: Example

A continuous problem and its discretisation:

- Example: A thin, quadratic metal plate is fixed at every side.
- The temporal constant temperature distribution at the boundary of the metal plate is known.
- Which temperature exists at each inner point of the metal plate, if the system is in a stationary state?



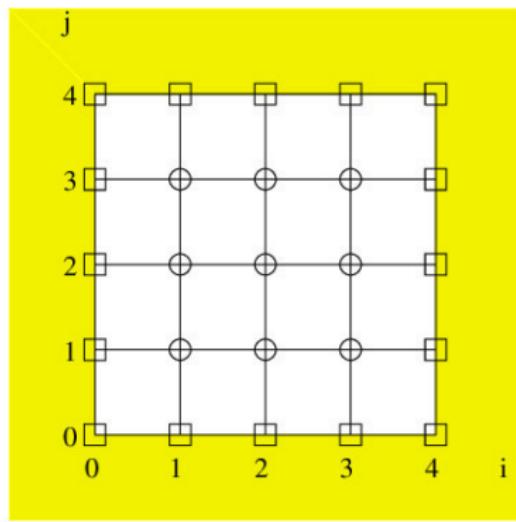
## Problem Formulation: Continuous

- The process of heat conduction can be described (approximately) by a mathematical model.
- The geometry of the metal plate is described by an area  $\Omega \subset \mathbb{R}^2$ .
- Wanted is the temperature distribution  $T(x, y)$  for all  $(x, y) \in \Omega$ .
- The temperature  $T(x, y)$  for  $(x, y)$  on the boundary  $\partial\Omega$  is known.
- If the metal plate homogeneous (same heat conduction coefficient everywhere), then the temperature in the inner domain is described by the partial differential equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0, \quad T(x, y) = g(x, y) \text{ auf } \partial\Omega. \quad (1)$$

## Problem Formulation: Discrete

- In the computer one cannot determine the temperature at every position  $(x, y) \in \Omega$  (innumerable many), but only at some selected ones.
- For that let  $\Omega = [0, 1]^2$  chosen specially (unit square).
- Via the parameter  $h = 1/N$ , for a  $N \in \mathbb{N}$ , we choose specially the points  $(x_i, y_j) = (ih, jh)$ , for all  $0 \leq i, j \leq N$ .
- One denotes this set of points also as regular, equidistant grid.



The points at the boundary have been marked with other symbols (squares) as the inner ones (circles).

# Discretisation I

How can the temperature  $T_{ij}$  at point  $(x_i, y_j)$  be determined?

- By a standard method: the method of „Finite Differences“
- Idea: The temperature at point  $(x_i, y_j)$  is expressed by the values of the four neighboring points:

$$T_{i,j} = \frac{T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}}{4} \quad (2)$$

$$\iff T_{i-1,j} + T_{i+1,j} - 4T_{i,j} + T_{i,j-1} + T_{i,j+1} = 0 \quad (3)$$

für  $1 \leq i, j \leq N - 1$ .

- From the form of (3) one recognises, that all  $(N - 1)^2$  equations for  $1 \leq i, j \leq N - 1$  together form a linear equation system:

$$AT = b \quad (4)$$

## Discretisation II

- Here  $G(A)$  corresponds exactly to the above drawn grid, if one neglects the boundary points (squares). The righthand side  $b$  of (3) is not even zero, but contains the temperature values at the boundary!
- The such calculated temperature values  $T_{i,j}$  at the points  $(x_i, y_j)$  are *not* identical with the solution  $T(x_i, y_i)$  of the partial differential equation (1). Furthermore applies

$$|T_{i,j} - T(x_i, y_i)| \leq O(h^2) \quad (5)$$

- This error is denoted as „discretisation error“. An increase in the size of  $N$  corresponds such to an exacter temperature calculation.

# Iteration Methods I

- We now want to solve the equation system (4) „iteratively“. Herefore we determine an arbitrary value of the temperature  $T_{i,j}^0$  at each point  $1 \leq i, j \leq N - 1$  (the temperature at the boundary is wellknown).
- Starting from this approximate solution we now want to calculate an improved solution. Herefore we use the formula (2) and set

$$T_{i,j}^{n+1} = \frac{T_{i-1,j}^n + T_{i+1,j}^n + T_{i,j-1}^n + T_{i,j+1}^n}{4} \quad \text{für alle } 1 \leq i, j \leq N - 1. \quad (6)$$

- Obviously the improved values  $T_{i,j}^{n+1}$  can be calculated simultaneously for each of the indices  $(i, j)$ , since they only depend on the old values  $T_{i,j}^n$ .

## Iteration Methods II

- One can indeed show, that

$$\lim_{n \rightarrow \infty} T_{i,j}^n = T_{i,j} \quad (7)$$

applies.

- The error  $|T_{i,j}^n - T_{i,j}|$  in the  $n$ -th approximate solution is denoted as „iteration error“.
- How large is this iteration error then? One needs a criterium up to which  $n$  one needs to compute.

# Iteration Methods III

- Therefore one considers how well the values  $T_{i,j}^n$  fulfill the equation (3), this means we set

$$E^n = \max_{1 \leq i,j \leq N-1} |T_{i-1,j}^n + T_{i+1,j}^n - 4T_{i,j}^n + T_{i,j-1}^n + T_{i,j+1}^n|$$

- Commonly one uses this error only relatively, thus one iterates as long until

$$E^n < \epsilon E^0$$

applies. Then the initial error  $E^0$  has been reduced by the reduction factor  $\epsilon$ .

- This leads us to the sequential method:

choose  $N, \epsilon$ ;

choose  $T_{i,j}^0$ ;

$$E^0 = \max_{1 \leq i,j \leq N-1} |T_{i-1,j}^0 + T_{i+1,j}^0 - 4T_{i,j}^0 + T_{i,j-1}^0 + T_{i,j+1}^0|;$$

$n = 0$ ;

**while** ( $E^n \geq \epsilon E^0$ )

{

**for** ( $1 \leq i,j \leq N-1$ )

$$T_{i,j}^{n+1} = \frac{T_{i-1,j}^n + T_{i+1,j}^n + T_{i,j-1}^n + T_{i,j+1}^n}{4};$$

$$E^{n+1} = \max_{1 \leq i,j \leq N-1} |T_{i-1,j}^{n+1} + T_{i+1,j}^{n+1} - 4T_{i,j}^{n+1} + T_{i,j-1}^{n+1} + T_{i,j+1}^{n+1}|;$$

$n = n + 1$ ;

}

# Iteration Methods IV

- All that can be written more compact in vector notation. Then  $AT = b$  is the equation system (4) to be solved. The approximation values  $T_{i,j}^n$  correspond to vectors  $T^n$  each.
- Formally  $T^{n+1}$  is calculated as

$$T^{n+1} = T^n + D^{-1}(b - AT^n)$$

with the diagonal matrix  $D = \text{diag}(A)$ . This scheme is denoted as Jacobi method.

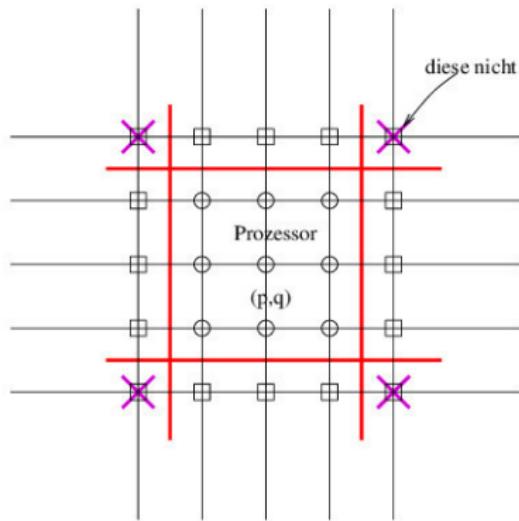
- The error  $E^n$  is constituted by

$$E^n = \|b - A \cdot T^n\|_\infty,$$

where  $\|\cdot\|_\infty$  is the maximum norm of a vector.

# Parallelisation I

- The algorithm allows again a data parallel formulation.
- Therefore the  $(N + 1)^2$  grid points are subdivided onto a  $\sqrt{P} \times \sqrt{P}$  processor array by partitioning of the index set  $I = \{0, \dots, N\}$ .
- The partitioning happens here *block-wise*:



## Parallelisation II

- Processor  $(p, q)$  computes then the values  $T_{i,j}^{n+1}$  with  $(i,j) \in \{start(p), \dots, end(p)\} \times \{start(q), \dots, end(q)\}$ .
- To do this, he needs however also the values  $T_{i,j}^n$  from the neighboring processors with  $(i,j) \in \{start(p) - 1, \dots, end(p) + 1\} \times \{start(q) - 1, \dots, end(q) + 1\}$ .
- These are the nodes, that have been marked with squares in the figure above!
- Each processor stores such beyond its assigned grid points an additional layer of grid points.

# Parallelisation III

- The parallel algorithm therefore consists of the following steps:

initial values  $T_{i,j}^0$  are known by all processors.

**while** ( $E^n > \epsilon E^0$ )

{

    calculate  $T_{i,j}^{n+1}$  for  $(i,j) \in \{start(p), \dots, end(p)\} \times \{start(q), \dots, end(q)\}$ ;

    exchange boundary values (squares) with neighbors;

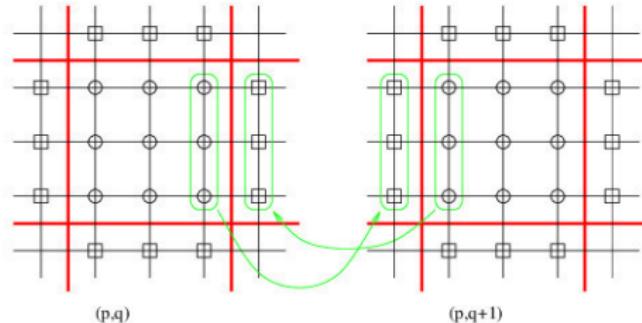
    calculate  $E^{n+1}$  for  $(i,j) \in \{start(p), \dots, end(p)\} \times \{start(q), \dots, end(q)\}$ ;

    Determine global maximum;

$n = n + 1$ ;

}

- In the exchange step two neighboring processors exchange values:



# Parallelisation IV

- For this exchange step one uses either asynchronous communication or synchronous communication with coloration.
- We calculate the scalability of a *single* iteration:

$$W = T_S(N) = N^2 t_{op} \implies N = \sqrt{\frac{W}{t_{op}}}$$

$$T_P(N, P) = \underbrace{\left( \frac{N}{\sqrt{P}} \right)^2 t_{op}}_{\text{calculation}} + \underbrace{\left( t_s + t_h + t_w \frac{N}{\sqrt{P}} \right)}_{\text{boundary exchange}} 4 + \underbrace{(t_s + t_h + t_w) \text{ld } P}_{\substack{\text{global} \\ \text{comm.: max.} \\ \text{for } E^n}}$$

$$T_P(W, P) = \frac{W}{P} + \frac{\sqrt{W}}{\sqrt{P}} \frac{4t_w}{\sqrt{t_{op}}} + (t_s + t_h + t_w) \text{ld } P + 4(t_s + t_h)$$

$$\begin{aligned} T_O(W, P) &= PT_P - W = \\ &= \sqrt{W} \sqrt{P} \frac{4t_w}{\sqrt{t_{op}}} + P \text{ld } P(t_s + t_h + t_w) + P4(t_s + t_h) \end{aligned}$$

## Parallelisation V

- Asymptotically we obtain the iso-efficiency function  $W = O(P \ln P)$  from the second term, albeit the first term will be dominant for practical values of  $N$  dominant. The algorithm is nearly optimal scalable.
- Because of the block-wise partitioning one has a surface-to-volume effect:  $\frac{N}{\sqrt{P}} / \left(\frac{N}{\sqrt{P}}\right)^2 = \frac{\sqrt{P}}{N}$ . In three space dimensions one obtains  $\left(\frac{N}{P^{1/3}}\right)^2 / \left(\frac{N}{P^{1/3}}\right)^3 = \frac{P^{1/3}}{N}$ .
- For same  $N$  and  $P$  the efficiency is such a little bit worse compared to two dimensions.

# Multigrid Methods I

- If we ask about the total efficiency of a method, then the number of operations is distinctive.
- Hereby is

$$T_S(N) = IT(N) \cdot T_{IT}(N)$$

- How many iterationen have now indeed to be executed depends besides N of course on the used method.
- For that one obtains the following classifications:

Jacobi, Gauß-Seidel :  $IT(N) = \mathcal{O}(N^2)$

SOR with  $\omega_{\text{opt}}$  :  $IT(N) = \mathcal{O}(N)$

Conjugated gradients (CG) :  $IT(N) = \mathcal{O}(N)$

Hierarchical basis d=2 :  $IT(N) = \mathcal{O}(\log N)$

Multigrid methods :  $IT(N) = \mathcal{O}(1)$

- The time for an iteration  $T_{IT}(N)$  is there for all schemes in  $\mathcal{O}(N^d)$  with comparable constant (in the region of 1 to 5).

## Multigrid Methods II

- We can see, that e.g. the multigrid method is much faster than the Jacobi method.
- Also the parallelisation of the Jacobi method does not help, since it applies:

$$T_{P,\text{Jacobi}}(N, P) = \frac{\mathcal{O}(N^{d+2})}{P} \quad \text{und} \quad T_{S,\text{MG}}(N) = \mathcal{O}(N^d)$$

- A doubling of N results in a fourfold increase of the effort for the parallelised Jacobi scheme in comparison to the sequential multigrid method!
- This leads to a fundamental paradigm of parallel programming:

Parallelise the best sequential algorithm, if possible anyhow!

## Multigrid Methods III

- Let us consider again the discretisation of the Laplace equation  $\Delta T = 0$ .
- This leads to the linear equation system

$$Ax = b$$

- Here the vector  $b$  is determined by the Dirichlet boundary values. Now be an approximation of the solution given by  $x^i$ . Herefore set the iteration error

$$e^i = x - x^i$$

- Because of the linearity of  $A$  we can conclude the following:

$$Ae^i = \underbrace{Ax}_b - Ax^i = b - Ax^i =: d^i$$

- Here we call  $d^i$  the defect .
- A good approximation for  $e^i$  is calculated by the solution of

$$Mv^i = d^i \quad \text{also} \quad v^i = M^{-1}d^i$$

- Herefore be  $M$  easier to solve than  $A$  ( in  $\mathcal{O}(N)$  steps, if  $x \in \mathbb{R}^N$  ).

# Multigrid Methods IV

- For special  $M$  we get the already known iteration method :

$$M = I \quad \rightarrow \text{Richardson}$$

$$M = \text{diag}(A) \quad \rightarrow \text{Jacobi}$$

$$M = L(A) \quad \rightarrow \text{Gauß-Seidel}$$

- We obtain the linear iteration method of the form

$$x^{i+1} = x^i + \omega M^{-1}(b - Ax^i)$$

- Here the  $\omega \in [0, 1]$  is a damping factor .
- For the error  $e^{i+1} = x - x^{i+1}$  applies:

$$e^{i+1} = (I - \omega M^{-1}A)e^i$$

- Here we denote the iteration matrix  $I - \omega M^{-1}A$  with  $S$ .
- The scheme is exactly then convergent if applies ( $\lim_{i \rightarrow \infty} e^i = 0$ ). This holds if the largest absolut eigen value of  $S$  is smaller than one.

# Smoothing Property I

- If the matrix  $A$  is symmetric and positive definite, then it has only real, positive eigen values  $\lambda_k$  for eigen vectors  $z_k$ .
- The Richardson iteration

$$x^{i+1} = x^i + \omega(b - Ax^i)$$

leads because of  $M = I$  to error

$$e^{i+1} = (I - \omega A)e^i$$

- Now we set the damping factor  $\omega = \frac{1}{\lambda_{\max}}$  and consider  $e^i = z_k (\forall k)$ .
- Then we obtain

$$\begin{aligned} e^{i+1} &= \left( I - \frac{1}{\lambda_{\max}} A \right) z_k = z_k - \frac{\lambda_k}{\lambda_{\max}} z_k = \left( 1 - \frac{\lambda_k}{\lambda_{\max}} \right) e^i \\ \left( 1 - \frac{\lambda_k}{\lambda_{\max}} \right) &= \begin{cases} 0 & \lambda_k = \lambda_{\max} \\ \approx 1 & \lambda_k \text{ small } (\lambda_{\min}) \end{cases} \end{aligned}$$

## Smoothing Property II

- In the case of small eigenvalues we thus have a bad damping of the error (it has the order of magnitude of  $1 - \mathcal{O}(h^2)$ ).
- This behaviour is qualitatively identical for the Jacobi and Gauß-Seidel iteration methods.
- But, to small eigenvalues belong long-wave eigenfunctions.
- These long-wave errors are such damped only very badly.
- With a pictorial view the iteration methods only offer a local smoothing of the error on which they work, since they get new iteration values only from values in the local neighborhood.
- Fast oscillations could be smoothed out fast, whilst long-wave errors survive the local smoothing operations quite unmodified.

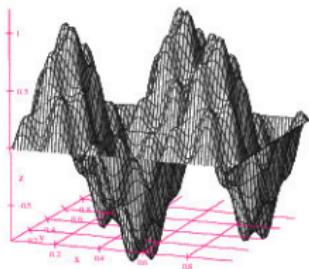
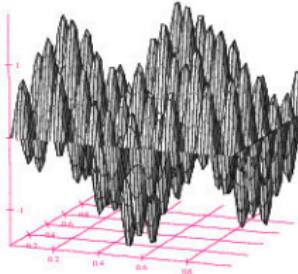
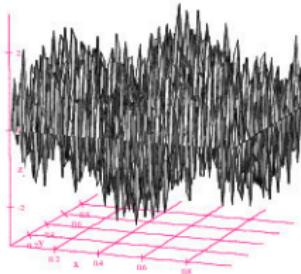
# Smoothing Property III

For illustration purposes we consider the following example:

The Laplacian equation  $-\Delta u = f$  is discretized via a five-point stencil on a structured grid. The associated eigenfunctions are  $\sin(\nu\pi x)\sin(\mu\pi y)$ , where  $1 \leq \nu$  and  $\mu \leq h^{-1} - 1$  apply. We set  $h = \frac{1}{32}$  and the initial error to

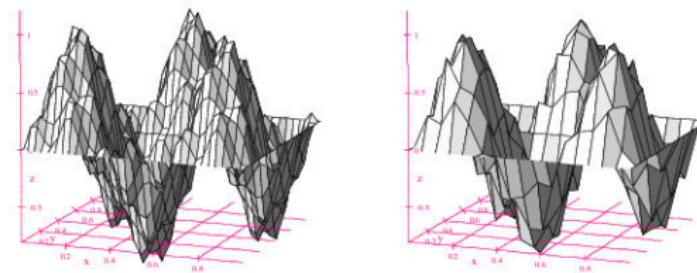
$$e^0 = \sin(3\pi x)\sin(2\pi y) + \sin(12\pi x)\sin(12\pi y) + \sin(31\pi x)\sin(31\pi y).$$

With  $\omega = \frac{1}{\lambda_{\max}}$  one obtains the damping factors (per iteration) for the Richardson iteration as 0.984, 0.691 and 0 for the individual eigenfunctions. The graphs below show the initial error and the error after one resp. five iterations.



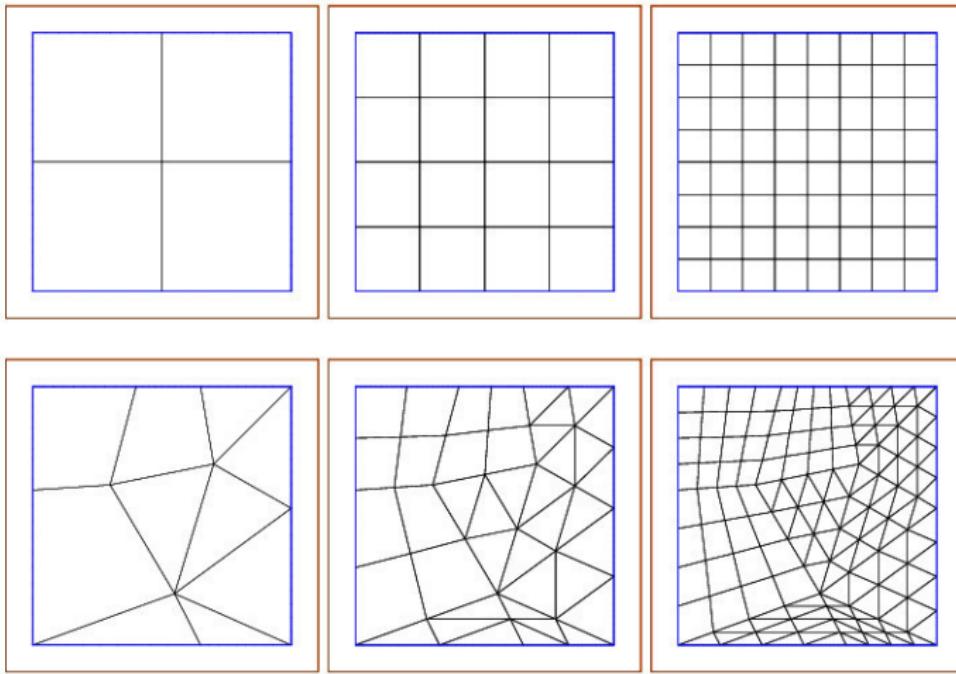
# Smoothing Property IV

- From this the idea arises to represent the long wave error on coarser grids, after smoothing out the fast oscillations.
- On this coarser grids the effort is then smaller to smooth the error curve.
- Because the curve is somehow smooth after presmoothing, this restriction onto fewer grid points is well possible.



# Grid Hierarchy I

- One constructs thus a complete sequence of grids of different accuracy.
- At first one smoothes out on the finest grid the short wave error functions.
- Then we restrict to the next coarser grid, and so on.



## Grid Hierarchy II

- According to that one obtains a complete sequence of linear systems

$$A_I x_I = b_I,$$

since the number of grid points  $N$  and therefore the length of  $x$  decreases on coarser grids.

- Of course one wants to return after this restriction again to the original fine grid.
- Therefore we perform a coarse grid correction.
- Let us assume we are on grid level  $I$ , thus we consider the LES

$$A_I x_I = b_I$$

- On this level the iterate  $x_I^j$  is given with an error of  $e_I^j$ , thus the error equation

$$A e_I^j = b_I - A_I x_I^j$$

Lets suppose,  $x_I^j$  is the result of  $\nu_1$  Jacobi, Gauß-Seidel or similar iterations.

- Then  $e_I^j$  is relatively smooth and thus also properly representable on a coarser grid, this means it can be interpolated well from a coarser grid to a finer one.

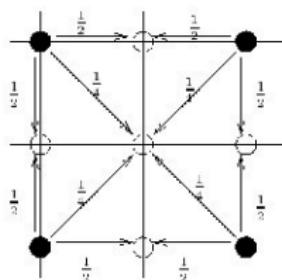
## Grid Hierarchy III

- For this be  $v_{I-1}$  the error on the coarser grid.
- Then with good approximation applies

$$e_I^i \approx P_I v_{I-1}$$

- Here  $P_I$  is an interpolation matrix (prolongation), that performs a linear interpolation and changes the coarse grid vector into a fine grid vector.

|               |               |               |               |
|---------------|---------------|---------------|---------------|
| 1             | 0             | 0             | 0             |
| $\frac{1}{2}$ | $\frac{1}{2}$ | 0             | 0             |
| 0             | 1             | 0             | 0             |
| $\frac{1}{2}$ | 0             | $\frac{1}{2}$ | 0             |
| $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ | $\frac{1}{4}$ |
| 0             | $\frac{1}{2}$ | 0             | $\frac{1}{2}$ |
| 0             | 0             | 1             | 0             |
| 0             | 0             | $\frac{1}{2}$ | $\frac{1}{2}$ |
| 0             | 0             | 0             | 1             |



# Two-grid and Multigrid Methods I

- Through combination of the equations above one obtains the equation for  $v_{l-1}$  by

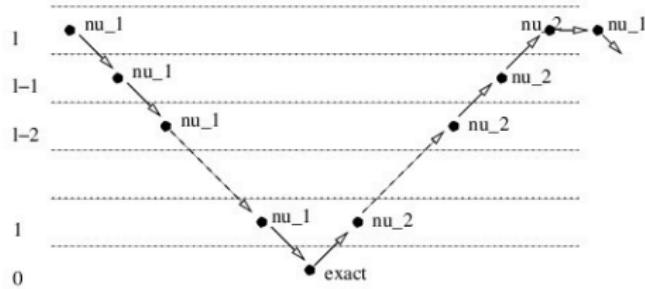
$$R_l A P_l v_{l-1} = R_l(b_l - A_l x_l^i)$$

- Here is  $R_l A P_l =: A_{l-1} \in \mathbb{R}^{N_{l-1} \times N_{l-1}}$  and  $R_l$  is the restriction matrix, for that one takes e.g.  $R_l = P_l^T$ .
- The so called two-grid method consists now of the two steps:
  - 1  $\nu_1$  Jacobi iterations (on level l)
  - 2 coarse grid correction  $x_l^{i+1} = x_l^i + P_l A_{l-1}^{-1} R_l(b_l - A_l x_l^i)$
- The recursive application leads to the multigrid methods.

```
mgc(l, x_l, b_l)
{
 if (l == 0) x_0 = A_0-1b_0;
 else {
 ν1 iterations one-grid method on Alxl = bl; // presmoothing
 dl-1 = Rl(bl - Alxl);
 vl-1 = 0;
 for (g = 1, ..., γ)
 mgc(l - 1, vl-1, dl-1);
 xl = xl + Plvl-1;
 ν2 iterations one-grid method on Alxl = bl; // postsmoothing
 }
}
```

# Two-grid and Multigrid Methods II

- It is sufficient to set  $\gamma = 1, \nu_1 = 1, \nu_2 = 0$  to get a iteration count of  $\mathcal{O}(1)$ .
- A single pass from level I to level 0 and back is denoted as V-cycle:



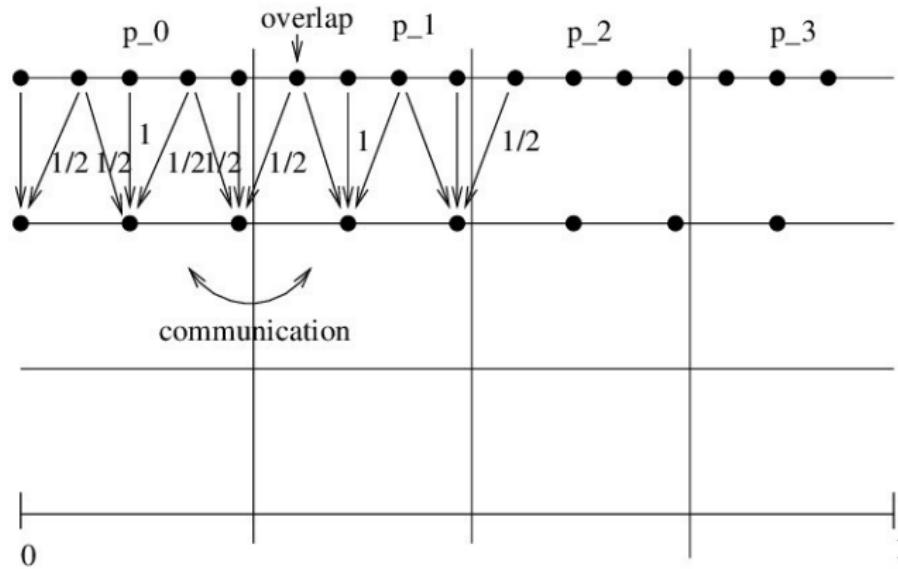
- Effort for two-dimensional structured grids:  $N$  is number of grid points in a row on the finest level, and  $C := t_{op}$ :

$$\begin{aligned} T_{IT}(N) &= \underbrace{CN^2}_{\text{level I}} + \underbrace{\frac{CN^2}{4}}_{\text{level I-1}} + \underbrace{\frac{CN^2}{16}}_{\text{coarse grid}} + \dots + \underbrace{G(N_0)}_{\text{coarse grid}} \\ &= CN^2 \underbrace{\left(1 + \frac{1}{4} + \frac{1}{16} + \dots\right)}_{\frac{4}{3}} + G(N_0) = \frac{4}{3}CN + G(N_0) \end{aligned}$$

# Parallelisation I

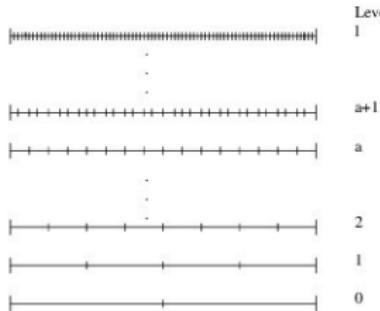
For data partitioning in the grid hierarchy on the individual processors one has to consider:

- In the coarse grid correction has to be checked, whether communication is necessary to calculate the node values in the coarse grid.
- How to handle the coarsest grids, where the number of unknowns in each dimension get smaller than the number of processors?



# Parallelisation II

- For illustration of the method we only consider the one-dimensional case. The distribution in the high-dimensional case is according to the tensor product (chessboard-like).
- The processor limits are chosen at  $p \cdot \frac{1}{P} + \epsilon$ , such the nodes, that reside on the boundary between two processors, are still assigned to the „previous“.
- It is to remark, that the defect, that is restringated in the coarse grid correction, can only calculated on the single master node, but not in the overlap!
- To solve the problem with the decreasing node count in the coarsest grids, one uses successively fewer processors. Be for that  $a := \text{Id } P$  and again  $C := t_{\text{op}}$ . On level 0 only one processor calculates, first on level  $a$  all are busy.



# Parallelisation III

| level | nodes               | processors    | effort                                        |
|-------|---------------------|---------------|-----------------------------------------------|
| l     | $N_l = 2^{l-a} N_a$ | $P_l = P$     | $T = 2^{l-a} CN_0$                            |
| a+1   | $N_{a+1} = 2N_a$    | $P_{a+1} = P$ | $T = 2CN_0$                                   |
| a     | $N_a$               | $P_a = P$     | $T = CN_0$                                    |
| 2     | $N_2 = 4N_0$        | $P_2 = 4P_0$  | $T = \frac{CN_2}{4} = CN_0$                   |
| 1     | $N_1 = 2N_0$        | $P_1 = 2P_0$  | $T = \frac{CN_1}{2} = \frac{C2N_0}{2} = CN_0$ |
| 0     | $N_0$               | $P_0 = 1$     | $G(N_0) \stackrel{\text{be}}{\approx} CN_0$   |

- Let's consider the total effort: From level 0 to level a  $\frac{N}{P}$  is constant, thus  $T_P$  grows like  $\text{Id } P$ . Therefore we get

$$T_P = \text{Id } P \cdot CN_0$$

- In the higher levels we get

$$T_P = C \cdot \frac{N_l}{P} \cdot \left( 1 + \frac{1}{2} + \frac{1}{4} + \dots \right) = 2C \frac{N_l}{P}$$

- The total effort is then given by the sum of both partial efforts.
- Here we have not taken into account the communication between the processors.

## Parallelisation IV

How effects the usage of the multigrid method the number of iteration steps to be executed?

- We show the number of used processors against the choice of the grid spacing, that has been used.
- The minimal error reduction has been set to  $10^{-6}$ .

| P/I | 5  | 6  | 7  | 8  |
|-----|----|----|----|----|
| 1   | 10 |    |    |    |
| 4   | 10 | 10 |    |    |
| 16  | 11 | 12 | 12 |    |
| 64  | 13 | 12 | 12 | 12 |

- The table shows the according iteration times in seconds for 2D (factor 4 of grid growths):

| P/I | 5    | 6    | 7    | 8    |
|-----|------|------|------|------|
| 1   | 3.39 |      |      |      |
| 4   | 0.95 | 3.56 |      |      |
| 16  | 0.32 | 1.00 | 3.74 |      |
| 64  | 0.15 | 0.34 | 1.03 | 3.85 |

## Most Important Knowledge

- Jacobi scheme is one of the most simple iteration methods for the solution of linear equation systems.
- For fixed reduction factor  $\epsilon$  one necessitates a specific number of iterations  $IT$  to reach a certain error reduction.
- $IT$  is *independent* on the choice of the starting value, but depends directly from the choice of the method (e.g. Jacobi scheme) and the grid spacing  $h$  (thus  $N$ ).
- For the Jacobi method applies  $IT = O(h^{-2})$ . For a halvening of the grid width  $h$  one needs the fourfold number of iterations to get the error reduction  $\epsilon$ . Since an iteration costs also four times more, the effort has increased by a factor of 16!
- There are a series of better iteration schemes for which e.g.  $IT = O(h^{-1})$ ,  $IT = O(\log(h^{-1}))$  or even  $IT = O(1)$  applies (CG method, hierarchical basis, multigrid method).
- Of course asymptotically ( $h \rightarrow \infty$ ) each of these methods is superior to a parallel naive scheme.
- One should therefore at all parallelize the method with optimal sequential complexity, especially because we want to solve large problems ( $h$  small) on parallel machines.

# Solution of Tridiagonal and Sparse Linear Equation Systems

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)  
University of Heidelberg  
INF 205  
D-69120 Heidelberg  
email: Stefan.Lang@iwr.uni-heidelberg.de

SoSe 2017

# Topics

Solution of tridiagonal and sparse linear equation systems

- Optimal sequential algorithm
- Cyclic reduction
- Domain decomposition
- $LU$  decomposition of sparse matrices
- Parallelisation

# Optimal Sequential Algorithm

- As an extreme case of sparse equation systems we consider

$$Ax = b \quad (1) \quad \left( \begin{array}{cccccc} * & * & & & & \\ * & * & * & & & \\ * & * & * & * & & \\ & * & * & * & * & \\ & * & * & * & * & * \\ & * & * & * & * & * \end{array} \right)$$

with  $A \in \mathbb{R}^{N \times N}$  tridiagonal.

- The optimal algorithm is the Gaussian elimination, sometimes also called Thomas algorithm.

# Optimal Sequential Algorithm

- Gaussian elimination for tridiagonal systems

// Forward elimination (here solution, not  $LU$  decomposition):

```
for ($k = 0; k < N - 1; k ++$) {
```

$$l = a_{k+1,k} / a_{k,k};$$

$$a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$$

$$b_{k+1} = b_{k+1} - l \cdot b_k;$$

```
} // $(N - 1) \cdot 5$ fp operations
```

// Backward substitution

$$x_{N-1} = b_{N-1} / a_{N-1,N-1};$$

```
for ($k = N - 2; k \geq 0; k --$) {
```

$$x_k = (b_k - a_{k,k+1} \cdot x_{k+1}) / a_{k,k};$$

```
} // $(N - 1)3 + 1$ fp operations
```

- The sequential complexity amounts to

$$T_S = 8Nt_f$$

Obviously the algorithm is strictly sequential!

# Cyclic Reduction

- Consider a tridiagonal matrix with  $N = 2M$  ( $N$  even).
- *Idea:* Eliminate in each *even* row  $k$  the elements  $a_{k-1,k}$  and  $a_{k+1,k}$  with the help of the odd rows above resp. beneath.
- Each even row is therefore only coupled with the second previous and second next; since these are just even, the dimension has been reduced to  $M = N/2$ .
- The remaining system is again tridiagonal, and the idea can be applied recursively.

|   |   |     |     |     |     |     |     |
|---|---|-----|-----|-----|-----|-----|-----|
| 0 | * | (*) | □   |     |     |     |     |
| 1 | * | *   | *   |     |     |     |     |
| 2 | □ | (*) | *   | (*) | □   |     |     |
| 3 |   | *   | *   | *   |     |     |     |
| 4 |   | □   | (*) | *   | (*) | □   |     |
| 5 |   |     |     | *   | *   | *   |     |
| 6 |   |     | □   | (*) | *   | (*) | □   |
| 7 |   |     |     |     | *   | *   | *   |
| 8 |   |     |     |     | □   | (*) | *   |
| 9 |   |     |     |     |     | *   | (*) |

(\*) are removed, thereby fill-in (□) is generated.

# Cyclic Reduction

- Algorithm of cyclic reduction

// Elimination of all odd unknowns in even rows:

**for** ( $k = 1; k < N; k += 2$ )

{ // row  $k$  modifies row  $k - 1$

$l = a_{k-1,k} / a_{k,k};$

$a_{k-1,k-1} = a_{k-1,k-1} - l \cdot a_{k,k-1};$

$a_{k-1,k+1} = -l \cdot a_{k,k+1}; // \text{fill-in}$

$b_{k-1} = b_{k-1} - l \cdot b_k;$

} //  $\frac{N}{2} 6t_f$

**for** ( $k = 2; k < N; k += 2$ );

{ // row  $k - 1$  modifies row  $k$

$l = a_{k,k-1} / a_{k-1,k-1};$

$a_{k,k-2} = l \cdot a_{k-1,k-2}; // \text{fill-in}$

$a_{k,k} = l \cdot a_{k-1,k};$

} //  $\frac{N}{2} 3t_f$

- All traversals of both loops can be processed in parallel (if we assume a machine with shared memory)!

# Cyclic Reduction

- Result of this elimination is

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | * | * |   |   |   |   |   |   |   |   |
| 1 | * | * | * |   |   |   |   |   |   |   |
| 2 | * | * | * | * |   |   |   |   |   |   |
| 3 |   | * | * | * |   |   |   |   |   |   |
| 4 |   | * | * | * | * |   |   |   |   |   |
| 5 |   |   | * | * | * |   |   |   |   |   |
| 6 |   |   | * | * | * |   | * |   |   |   |
| 7 |   |   |   | * | * | * |   |   |   |   |
| 8 |   |   |   |   | * | * |   |   |   | * |
| 9 |   |   |   |   |   | * |   |   |   | * |

| resp. after reordering |   |   |   |   |   |   |   |   |   |  |
|------------------------|---|---|---|---|---|---|---|---|---|--|
|                        | 1 | 3 | 5 | 7 | 9 | 0 | 2 | 4 | 6 |  |
| 1                      | * |   |   |   |   | * | * |   |   |  |
| 3                      |   | * |   |   |   | * |   | * | * |  |
| 5                      |   |   | * |   |   |   |   | * |   |  |
| 7                      |   |   |   | * |   |   |   |   | * |  |
| 9                      |   |   |   |   | * |   |   |   |   |  |
| 0                      |   |   |   |   |   | * | * |   |   |  |
| 2                      |   |   |   |   |   | * | * | * |   |  |
| 4                      |   |   |   |   |   |   | * | * | * |  |
| 6                      |   |   |   |   |   |   |   | * | * |  |
| 8                      |   |   |   |   |   |   |   |   | * |  |

- Are the  $x_{2k}$ ,  $k = 0, \dots, M - 1$ , calculated, then the odd unknowns can be calculated with

**for** ( $k = 1; k < N - 1; k += 2$ )

$$x_k = (b_k - a_{k,k-1} \cdot x_{k-1} - a_{k,k+1} \cdot x_{k+1}) / a_{k,k};$$

//  $\frac{N}{2} 5t_f$

$$x_{N-1} = (b_{N-1} - a_{N-1,N-2} \cdot x_{N-2}) / a_{N-1,N-1};$$

completely in *parallel*.

# Cyclic Reduction

- The *sequential* effort for the cyclic reduction is therefore

$$\begin{aligned}T_S(N) &= (6 + 3 + 5)t_f \left( \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots + 1 \right) \\&= 14Nt_f\end{aligned}$$

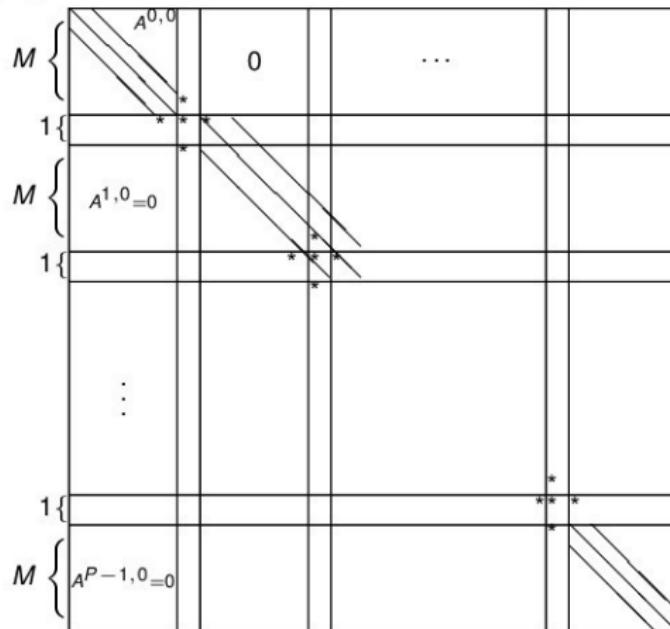
- This is nearly twice as much as the optimal sequential algorithm needs. Therefore the cyclic reduction can be parallelised. The maximal achievable efficiency is however

$$E_{\max} = \frac{8}{14} \approx 0.53,$$

where we have assumed, that all operations are executed optimally parallel and communication is for free (backward substitution needs only  $\frac{N}{2}$  processors!). We have not taken into account that cyclic reduction necessitates more index calculation!

# Domain Decomposition Methods

- Another approach can be in principle be extended to more general problem formulations: domain decomposition methods
  - Be  $P$  the number of processors and  $N = MP + P - 1$  for a  $M \in \mathbf{N}$ . We subdivide then the  $N \times N$  matrix  $A$  in  $P$  blocks à  $M$  rows with a single row between the blocks:



# Domain Decomposition Methods

- The unknowns between the blocks form the *interface*. Each block is at most coupled to two interface unknowns.
- Now we sort rows and columns of the matrix such that the interface unknowns are moved to the end. This results in the following shape:

$$\begin{array}{|c|c|c|} \hline A^{0,0} & & A^{0,I} \\ A^{1,1} & & A^{1,I} \\ \ddots & & \vdots \\ & A^{P-1,P-1} & A^{P-1,I} \\ \hline A^{I,0} & A^{I,1} & \dots & A^{I,P-1} & A^{I,I} \\ \hline \end{array},$$

where  $A^{p,p}$  are the  $M \times M$  tridiagonal partial matrices from  $A$  and  $A^{I,I}$  is a  $P-1 \times P-1$  diagonal matrix. The  $A^{p,I}$  have the general form

$$A^{p,I} = \left( \begin{array}{c|c|c} \dots & * & \dots \\ \hline & * & \end{array} \right).$$

# Domain Decomposition Methods

- Idea: Eliminate blocks  $A^{I,*}$  in the block representation. Thereby  $A^{I,I}$  is modified, more exact the following block representation is created:

$$\begin{array}{|c|c|c|c|} \hline & A^{0,0} & & A^{0,I} \\ & A^{1,1} & & A^{1,I} \\ & \ddots & & \vdots \\ & A^{P-1,P-1} & & A^{P-1,I} \\ \hline 0 & 0 & \dots & 0 \\ \hline & & & S \\ \hline \end{array},$$

$$\text{mit } S = A^{I,I} - \sum_{p=0}^{P-1} A^{I,p} (A^{p,p})^{-1} A^{p,I}.$$

- $S$  is in general denoted as „Schurcomplement“. All eliminations in  $\sum_{p=0}^{P-1}$  can be executed *parallel*.
- After solution of a system  $Sy = d$  for the interface unknowns the inner unknowns can again be calculated in parallel.
- $S$  has dimension  $P - 1 \times P - 1$  and is itself sparse, as we can see soon.

# Execution of the Plan

1. Transform  $A^{p,p}$  to diagonal shape.

( $a_{i,j}$  denotes  $(A^{p,p})_{i,j}$ , if not stated otherwise):

$\forall p$  parallel

**for** ( $k = 0; k < M - 1; k ++$ ) // lower diagonal

{

$$l = a_{k+1,k} / a_{k,k};$$

$$a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1};$$

$$\text{if } (p > 0) \quad a_{k+1,p-1}^{p,l} = a_{k+1,p-1}^{p,l} - l \cdot a_{k,p-1};$$

// fill-in left boundary

$$b_{k+1}^p = b_{k+1}^p - l \cdot b_k^p;$$

} //  $(M - 1)7t_f$

**for** ( $k = M - 1; k > 0; k --$ ) // upper diagonal

{

$$l = a_{k-1,k} / a_{k,k};$$

$$b_{k-1}^p = b_{k-1}^p - l \cdot b_k^p;$$

$$\text{if } (p > 0) \quad a_{k-1,p-1}^{p,l} = a^{p,l} - l \cdot a_{k,p-1}^{p,l}; \quad // \text{left boundary}$$

$$\text{if } (p < P - 1) \quad a_{k-1,p}^{p,l} = a_{k-1,p}^{p,l} - l \cdot a_{k,p}^{p,l}; \quad // \text{right boundary, fill-in}$$

} //  $(M - 1)7t_f$

# Execution of the Plan

2. Eliminate in  $A^{l,*}$ .

$\forall p$  parallel:

**if** ( $p > 0$ )

{

$$l = a_{p-1,0}^{l,p} / a_{0,0}^{p,p};$$

$$a_{p-1,p-1}^{l,l} = a_{p-1,p-1}^{l,l} - l \cdot a_{0,p-1}^{p,l};$$

$$\text{if } (p < P-1) \quad a_{p-1,p}^{l,l} = a_{p-1,p}^{l,l} - l \cdot a_{0,p}^{p,l};$$

$$b_{p-1}^l = b_{p-1}^l - l \cdot b_0^p;$$

// left boundary  $P-1$  in interface

// diagonal in  $S$

// upper diag. in  $S$ , fill-in

}

**if** ( $p < P-1$ )

{

$$l = a_{p,M-1}^{l,p} / a_{M-1,M-1}^{p,p};$$

$$\text{if } (p > 0) \quad a_{p,p-1}^{l,l} = a_{p,p-1}^{l,l} - l \cdot a_{M-1,p-1}^{p,l};$$

// right boundary

$$a_{p,p}^{l,l} = a_{p,p}^{l,l} - l \cdot a_{M-1,p}^{p,l};$$

$$b_p^l = b_p^l - l \cdot b_{M-1}^p;$$

// fill-in lower diag of  $S$

}

# Execution of the Plan

3. Solve Schurcomplement.

$S$  is *tridiagonal* with dimension  $P - 1 \times P - 1$ . Assume that  $M \gg P$  and solve sequential.  $\rightarrow 8Pt_f$  effort.

4. Calculate inner unknowns.

Here, only one diagonal matrix has to be solved per processor.

$\forall p$  parallel:

```
for (k = 0; k < M - 1; k++)
 $x_k^p = (b_k^p - a_{k,p-1}^{p,I} \cdot x_{p-1}^I - a_{k,p}^{p,I} \cdot x_p^I) / a_{k,k}^{p,p};$
// M5tf
```

# Analysis

- Total effort parallel:

$$\begin{aligned}T_P(N, P) &= 14Mt_f + O(1)t_f + 8Pt_f + 5Mt_f = \\&= 19Mt_f + 8Pt_f\end{aligned}$$

(without communication!)

$$\begin{aligned}E_{\max} &= \frac{8(MP + P - 1)t_f}{(19Mt_f + 8Pt_f)P} \approx \\&\approx \underbrace{\frac{1}{\frac{19}{8} + \frac{P}{M}}}_{\text{für } P \ll M} \leq \frac{8}{19} = 0.42\end{aligned}$$

- The algorithm needs additional memory for the fill-in. Cyclic reduction works with overwriting of old entries.

# *LU* Decomposition of Sparse Matrices

What is a sparse matrix

- In general one speaks of a sparse matrix, if it has in (nearly) each row only a constant number of non-zero elements.
- If  $A \in \mathbb{R}^{N \times N}$ , then  $A$  has only  $O(N)$  instead of  $N^2$  entries.
- For large enough  $N$  it is then advantageous regarding computing time and memory not to process resp. to store this large number of zeros.

# *LU* Decomposition of Sparse Matrices

## Fill-in

- While *LU* decomposition elements, that initially have been zero, can get non-zero during the elimination process.
- One speaks then of „Fill-in“.
- This heavily depends on the structure of the matrix. As an extreme example consider the „arrow matrix“

|   |   |   |   |
|---|---|---|---|
| * |   | * |   |
|   | * |   |   |
|   |   | * | 0 |
| * |   | * |   |
| * |   | . | . |
|   | 0 | . | . |
|   |   | * |   |
|   |   |   | * |

- During elimination in the natural sequence (without pivoting) the whole matrix is „filled-in“.

# LU Decomposition of Sparse Matrices

## Reordering of the matrix

- If we rearrange the matrix by row and column permutations to the form

$$\left[ \begin{array}{ccc|c} * & & & 0 \\ * & & & * \\ * & & \ddots & \\ \vdots & & \ddots & * \\ 0 & & * & \\ & * & & * \end{array} \right],$$

- Obviously no fill-in is produced.
- An important point in the *LU* decomposition of sparse matrices is to find a matrix ordering such that the fill-in is minimised.
- Reordering is strongly coupled to pivoting.

# Pivoting

- If the matrix  $A$  is symmetric positive definite (SPD), then the  $LU$  factorisation is always numerically stable, and *no* pivoting is necessary.
  - The matrix can thus be reordered in advance, that the fill-in gets small.
- For a general, invertible matrix one will need to use pivoting.
  - Then during elimination a compromise between numerical stability and fill-in has to be found dynamically.
- Therefore nearly all codes restrict to the symmetric positive case and determine an elimination sequence that minimizes the fill-in in advance.
- The exact solution of the minimization problem is  $\mathcal{NP}$  complete.
  - One therefore uses heuristical methods.

# Graph of a Matrix

## Matrix graph

- In the symmetric positive case fill-in can be investigated purely by the zero structure of the matrix.
- For an arbitrary, now not necessarily symmetric  $A \in \mathbb{R}^{N \times N}$  we define an undirected graph  $G(A) = (V_A, E_A)$  by

$$\begin{aligned} V_A &= \{0, \dots, N-1\} \\ (i, j) \in E_A &\iff a_{ij} \neq 0 \vee a_{ji} \neq 0. \end{aligned}$$

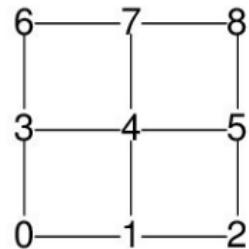
- This graph describes the direct dependencies of the unknowns beneath each other.

# Graph of a Matrix

Example:

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | * | * |   | * |   |   |   |   |   |
| 1 | * | * | * |   | * |   |   |   |   |
| 2 |   | * | * |   |   | * |   |   |   |
| 3 | * |   |   | * | * |   |   | * |   |
| 4 |   | * |   | * | * | * |   | * |   |
| 5 |   |   | * |   | * | * |   |   | * |
| 6 |   |   |   | * |   |   | * | * |   |
| 7 |   |   |   |   | * |   | * | * | * |
| 8 |   |   |   |   | * |   | * | * | * |

$A$

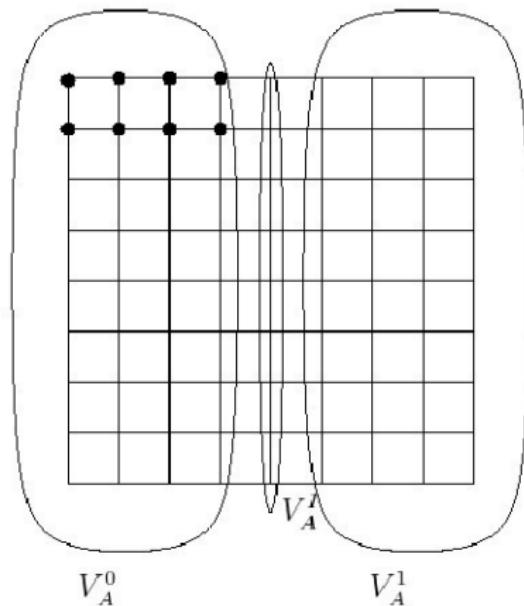


$G(A)$

# Matrix Ordering Strategies

## Nested Dissection

- An important method to order SPD matrices for the purpose of fill-in minimisation is the „*nested dissection*“.
- Example:  
The graph  $G(A)$  of the matrix  $A$  be a quadratic grid



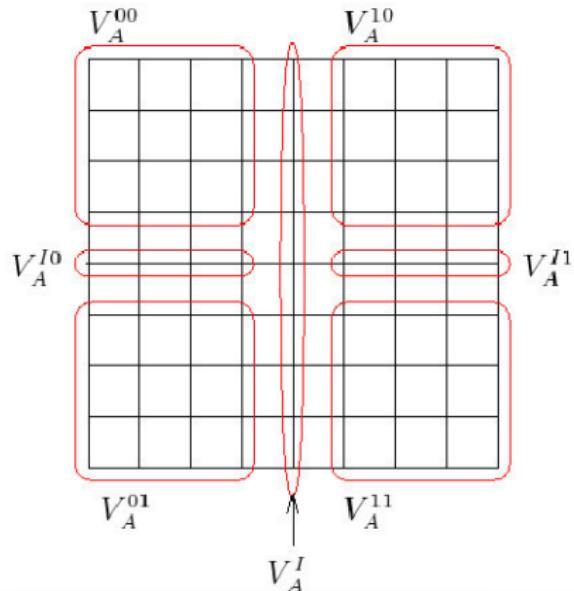
# Matrix Ordering Strategies

Now we divide the node set  $V_A$  in three parts:  $V_A^0$ ,  $V_A^1$  and  $V_A^I$ , such that

- $V_A^0$  and  $V_A^1$  are as large as possible,
- $V_A^I$  is a separator, this means when  $V_A^I$  is removed from the graph this is split into two parts. Thus there is *no*  $(i, j) \in E_A$ , such that  $i \in V_A^0$  and  $j \in V_A^1$ .
- $V_A^I$  is as small as possible,
- The figure shows a possibility for such a partitioning.

# Matrix Ordering Strategies

- Now one reorders the rows and columns such that first the indices  $V_A^0$  are present, then  $V_A^1$  and finally  $V_A^I$ .
- Then we apply the method *recursively* to the partial graphs with the node sets  $V_A^0$  and  $V_A^1$ .
- The method stops, if the graphs has reached a predefined size.
- Example graph after two steps:



# Matrix Ordering Strategies

|            | $V_A^{00}$ | $V_A^{01}$ | $V_A^{I0}$ | $V_A^{10}$ | $V_A^{11}$ | $V_A^{I1}$ | $V_A^I$ |
|------------|------------|------------|------------|------------|------------|------------|---------|
| $V_A^{00}$ | *          | 0          | *          |            |            |            | *       |
| $V_A^{01}$ | 0          | *          | *          |            |            |            | *       |
| $V_A^{I0}$ | *          | *          | *          |            |            |            | *       |
| $V_A^{10}$ |            |            |            | *          |            | *          | *       |
| $V_A^{11}$ |            |            |            |            | *          | *          | *       |
| $V_A^{I1}$ |            |            |            | *          | *          | *          | *       |
| $V_A^I$    | *          | *          | *          | *          | *          | *          | *       |

$A$ , reordered

## Complexity of the nested dissection

- For the example above the nested dissection numbering leads to a complexity of  $O(N^{3/2})$  for the  $LU$  decomposition.
- For comparison one needs with lexicographic numbering (band matrix)  $O(N^2)$  operations.

# Data Structures for Sparse Matrices

- There are a series of data structures for storage of sparse matrices.
- Goal is an efficient implementation of algorithms.
- Thus one has to watch out for data locality and as few overhead as possible by additional index calculation.
- An often used data structure is „*compressed row storage*“
- If the  $N \times N$  matrix has in total  $M$  non-zero elements, then one stores the matrix elements row-wise in an one-dimensional field:

**double**  $a[M];$

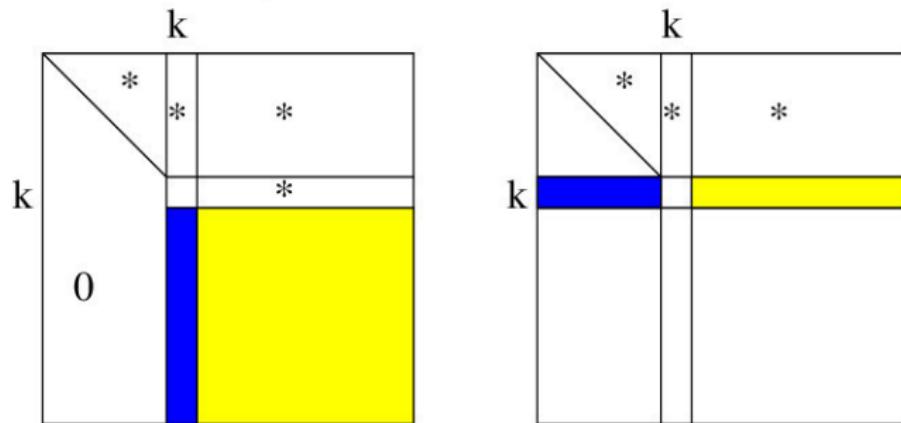
- The management of index information happens via three arrays  
**int**  $s[N], r[N], j[M];$
- Their meaning shows the realisation of the matrix-vector product  $y = Ax$ :

```
for ($i = 0; i < N; i ++$) {
 $y[i] = 0;$
 for ($k = r[i]; k < r[i] + s[i]; k ++$)
 $y[i] += a[k] \cdot x[j[k]];$
}.
```

- $r$  provides row start,  $s$  the row length, and  $j$  the column index.

# Elimination Forms

- In the  $LU$  decomposition of dense matrices we have used the so-called  $kij$  form of the  $LU$  decomposition.



- Here in each step  $k$  all  $a_{ik}$  for  $i > k$  are eliminated, which requires a modification of all  $a_{ij}$  with  $i, j > k$ .
- This situation is shown left.
- In the  $kji$  variant one eliminates in step  $k$  all  $a_{kj}$  with  $j < k$ .
- Here the  $a_{ki}$  with  $i \geq k$  are modified. Watch out, that the  $a_{kj}$  have to be eliminated from left to right!
- For the following sequential  $LU$  decomposition for sparse matrices we are going to start from this  $kji$  variant.

# Sequential Algorithm

- In the following we assume:

- ▶ The matrix  $A$  can be factorized in the given ordering without pivoting. The ordering has been chosen in an appropriate way to minimize fill-in.
- ▶ The data structure stores all elements  $a_{ij}$  with  $(i, j) \in G(A)$ . Because of the definition of  $G(A)$  applies:

$$(i, j) \notin G(A) \Rightarrow a_{ij} = 0 \wedge a_{ji} = 0.$$

If  $a_{ij} \neq 0$ , then in every case also  $a_{ji}$  is stored, *also if this is zero*. The matrix does not have to be symmetric.

- The extension of the structure of  $A$  happens purely because of the information in  $G(A)$ . Thus also a  $a_{kj}$  is formally eliminated, if  $(k, j) \in G(A)$  applies. This can possibly create a fill-in  $a_{ki}$ , albeit applies  $a_{ki} = 0$ .
- The now presented algorithm uses the sets  $S_k \subset \{0, \dots, k - 1\}$ , that contain in step  $k$  exactly the column indices, that have to be eliminated.

# Sequential Algorithm

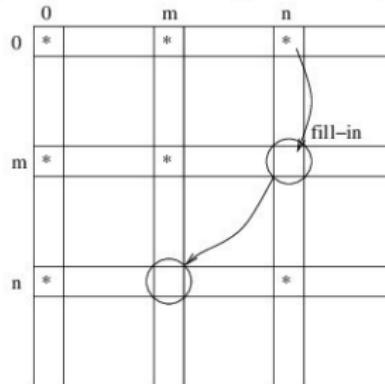
```
for (k = 0; k < N; k++) $S_k = \emptyset$;
for (k = 0; k < N; k++)
{
 // 1. extend matrix graph
 for (j ∈ S_k)
 {
 $G(A) = G(A) ∪ \{(k, j)\};$
 for (i = k; i < N; i++)
 if ((j, i) ∈ $G(A)$)
 $G(A) = G(A) ∪ \{(k, i)\};$
 }

 // 2. Eliminate
 for (j ∈ S_k)
 {
 $a_{k,j} = a_{k,j} / a_{j,j}$; // eliminate $a_{k,j}$
 for (i = j + 1; i < N; i++)
 $a_{k,i} = a_{k,i} - a_{k,j} \cdot a_{j,i};$ // L factor
 }

 // 3. update S_i for $i > k$, holds because of symmetry of E_A
 for (i = k + 1; i < N; i++)
 if ((k, i) ∈ $G(A)$) // $\Rightarrow (i, k) \in G(A)$
 $S_i = S_i ∪ \{k\};$
}
```

# Sequential Algorithm

- We consider in an example, how the  $S_k$  are mapped.



- At start  $G(A)$  shall contain the elements

$$G(A) = \{(0,0), (m,m), (n,n), (0,n), (n,0), (0,m), (m,0)\}$$

- For  $k = 0$  is set in step 1  $S_m = \{0\}$  and  $S_n = \{0\}$ .
- Now, next for  $k = m$  the  $a_{m,0}$  is eliminated.
- This generates the fill-in  $(m,n)$ , which again has in step 3 for  $k = m$  as consequence the instruction  $S_n = S_n \cup \{m\}$ .
- Thus applies at start of traversal  $k = n$  correctly  $S_n = \{0, m\}$  and in step 1 the fill-in  $a_{n,m}$  is correctly generated, before the elimination of  $a_{n,0}$  is performed. This is enabled because of the symmetry of  $G_A$ .

# Parallelisation

*LU* decomposition of sparse matrices has the following possibilities for a parallelisation:

- *coarse granularity*: In all  $2^d$  partial sets of indices, that has been created by nested dissection of depth  $d$ , one can start in parallel with the elimination. First for the indices, that correspond to the separators, communication is necessary.
- *medium granularity*: Single rows can be processed in parallel, as soon as the according pivot row is locally available. This corresponds to the parallelisation of the dense *LU* decomposition.
- *fine granularity*: modifications of an individual row can be processed in parallel, as soon as the pivot line and the multiplicator are available. This is used for the two-dimensional data distribution in the dense case.

# Parallelisation: Case $N = P$

Program (LU decomposition for sparse matrices and  $N = P$ )

parallel sparse-lu-1

```
{
 const int N = ...;
 process Π[int k ∈ {0, ..., N - 1}]
 {
 // (only pseudo code!)
 S = ∅; // 1. set S
 for (j = 0; j < k; j++)
 if ((k, j) ∈ G(A)) S = S ∪ {j};
 for (j = 0; j < k; j++)
 if (j ∈ Sk)
 {
 recv(Πj, r);
 // extend pattern
 for (i = j + 1; i < N; i++)
 {
 if (i < k ∧ (j, i) ∈ G(A))
 S = S ∪ {i};
 if ((j, i) ∈ G(A) ∪ {(k, i)})
 G(A) = G(A) ∪ {(k, i)};
 }
 // eliminate ak,j = ak,j / aj,j;
 for (i = j + 1; i < N; i++)
 ak,i = ak,i - ak,j · aj,i;
 }
 for (i = k + 1; i < N; i++)
 if ((k, i) ∈ G(A))
 send row k at Πi;
 }
}
```

# Parallelisation for $N \gg P$

For case  $N \gg P$

- Each processor has now a complete block of rows. Three things have to happen:
- *Receive pivot rows* of other processors and store them in the set  $R$ .
- *Send finished rows* from the send buffer  $S$  to the target processors.
- *Local elimination*
  - ▶ choose a row  $j$  from the receive buffer  $R$ .
  - ▶ eliminate with this row locally all possible  $a_{k,j}$ .
  - ▶ if a row is going to be ready, put it into the send buffer (there may be several target processors).