
Parallelization of Explicit and Implicit Solvers

Rolf Rabenseifner, Christoph Niethammer

University of Stuttgart

High-Performance Computing-Center Stuttgart

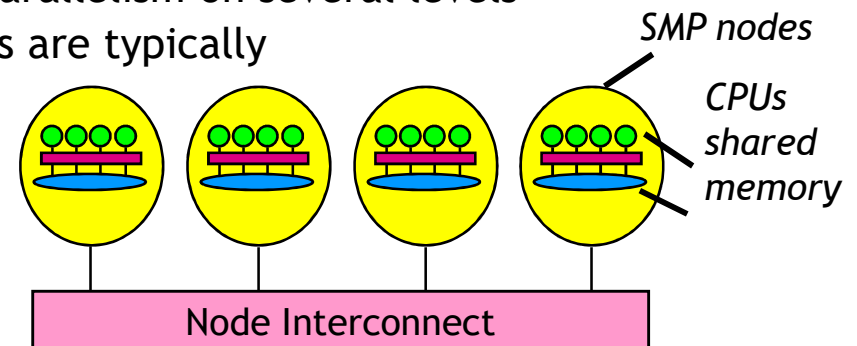


Outline

- Parallelization of explicit or implicit solver [slides 1-42]
 - PDE [4] → Discretization [4] → Explicit time-step integration [5]
 - Algebraic viewpoint [6] → Implicit time-step [8] → no principle differences [10]
 - Parallelization [11] → Domain Decomposition [12] → Load Balancing [13-17]
 - Halo [18-20] → Speedup & Amdahl's Law [20-26]
 - Parallelization of Implicit Solver [27-31] → Optimization Hints [32-35]
 - Vectorization & Cache Optimization [35-38]
 - Solver-Classes & Red/Black (checkerboarder) [39-41]
 - Literature [42]
- Parallel hardware [slides 43-49]
- Parallel programming models [slides 50-69]
- Parallelization scheme [slides 70-74]

Motivation

- Most systems have some kind of parallelism
 - Pipelining -> vector computing
 - Functional Parallelism -> modern processor technology
 - Combined instructions -> e.g. multiply-add as one instruction
 - Hyperthreading
 - Several CPUs on Shared Memory (SMP) with Multithreading
 - Distributed memory with
 - Message Passing or
 - Remote Memory Access
- Most systems are hybrid architectures with parallelism on several levels
- High Performance Computing (HPC) platforms are typically
 - Clusters (distributed memory) of
 - SMP nodes with several CPUs
 - Each CPU with several
 - Floating point units, pipelining ...



Partial Differential Equation (PDE) and Discretization

- $\partial T / \partial t = f(T, t, x, y, z)$
- Example: Heat conduction $\partial T / \partial t = \alpha \Delta T$
- Discretization: lower index $i, j \leftrightarrow$ continuous range x, y (2-dim. example)
upper index $t \leftrightarrow$ continuous range t
- $\partial T / \partial t = (T_{ij}^{t+1} - T_{ij}^t) / dt, \quad \partial^2 T / \partial x^2 = (T_{i+1,j} - 2T_{i,j} + T_{i-1,j}) / dx^2, \quad \dots$
- $(T_{ij}^{t+1} - T_{ij}^t) / dt = \alpha((T_{i+1,j}^? - 2T_{i,j}^? + T_{i-1,j}^?) / dx^2 + (T_{i,j+1}^? - 2T_{i,j}^? + T_{i,j-1}^?) / dy^2)$

Explicit time-step integration

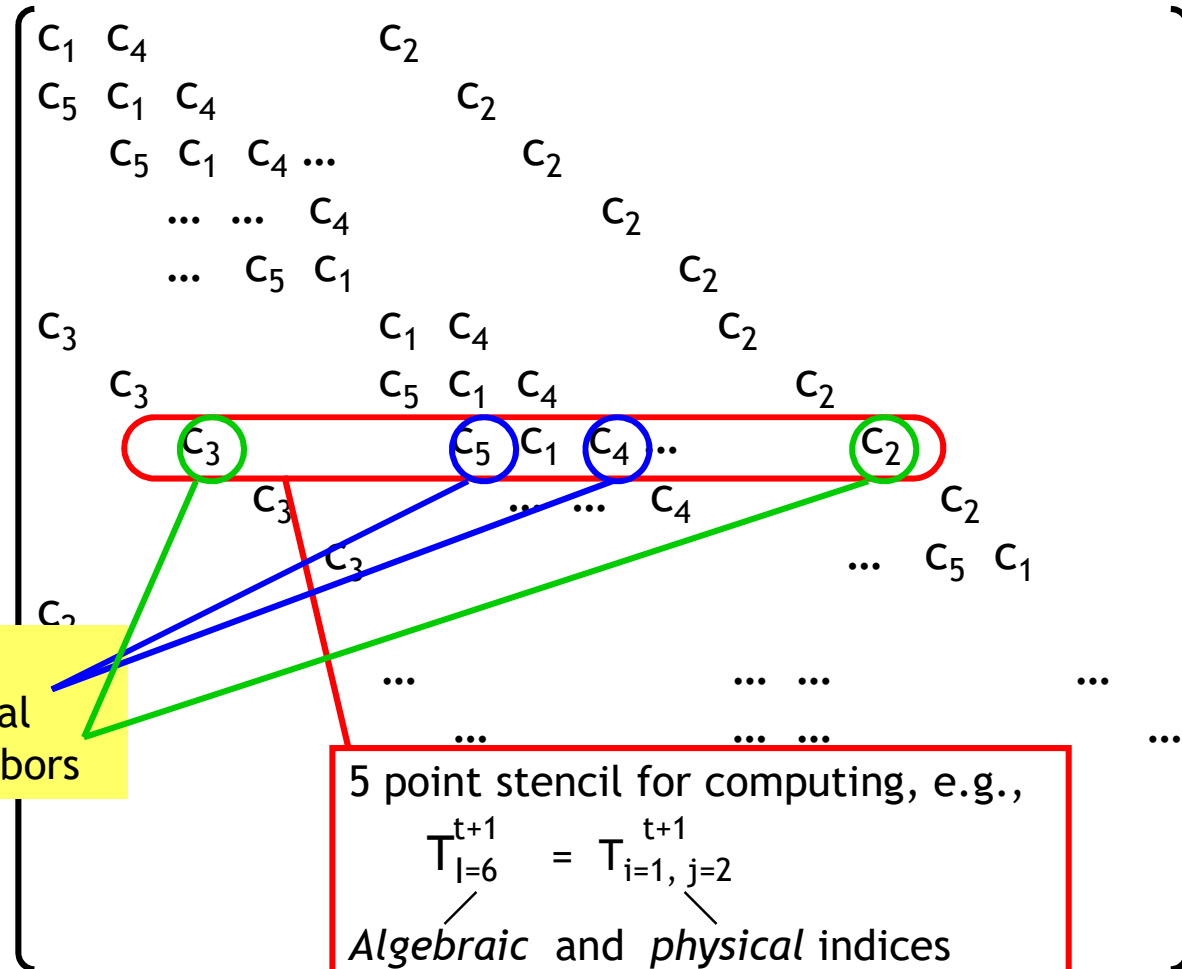
- If the right side depends only on old values T^t , i.e., $(?) = t$
- $$T_{ij}^{t+1} = T_{ij}^t + \alpha((T_{i+1,j}^t - 2T_{i,j}^t + T_{i-1,j}^t)/dx^2 + (T_{i,j+1}^t - 2T_{i,j}^t + T_{i,j-1}^t)/dy^2)dt$$
- You can implement this, e.g., as two nested loops:
 do i=0,m-1
 do j=0,n-1
 Tnew(i,j) = (1+c₁)T(i,j) + c₂T(i+1,j) + c₃T(i-1,j) + c₄T(i,j+1) +
c₅T(i,j-1)
 end do
 end do
- Vectorizable loop, without indirect addressing!

Algebraic view-point

- Explicit scheme:
- $T_{ij}^{t+1} = (1+c_1)T_{ij}^t + c_2T_{i+1,j}^t + c_3T_{i-1,j}^t + c_4T_{i,j+1}^t + c_5T_{i,j-1}^t$
- Can be **viewed** as a sparse-matrix-multiply
 - Choose a global numbering
 - $i,j = 0,0; 0,1; \dots 0,n-1; 1,0; 1,1; \dots 1,n-1; \dots m-1,0; \dots m-1,n-1$
 - $\rightarrow l = 0; 1; \dots n-1; n; n+1; \dots 2n-1; \dots (m-1)n; \dots mn-1$
 - $(T_{ij})_{i=0..m-1, j=0..n-1}$ is view as a vector $(T_l)_{l=0..mn-1}$
 - $T^{t+1} = (I+A)T^t$
- Is **never programmed** as a general sparse-matrix-multiply!
- This algebraic view-point is important to understand the parallelization of iterative solvers on the next slides

Matrix notation $T = AT$

$$A = (A_{IJ})_{\substack{I=0, mn-1 \\ J=0, mn-1}}$$



Representing physical relation to vertical and to horizontal neighbors

5 point stencil for computing, e.g.,
 $T_{l=6}^{t+1} = T_{i=1, j=2}^{t+1}$
 Algebraic and physical indices

Implicit time-step: Solving a PDE

- The right side depends also on **new** values T^{t+1} ,
i.e., T^{t+1} or a combination of old and new values
- $$T_{ij}^{t+1} = T_{ij}^t + \alpha((T_{i+1,j}^{t+1} - 2T_{i,j}^{t+1} + T_{i-1,j}^{t+1})/dx^2 + (T_{i,j+1}^{t+1} - 2T_{i,j}^{t+1} + T_{i,j-1}^{t+1})/dy^2)dt$$
- You have to implement a global solver in each time-step
- $$(1-c_1)T_{ij}^{t+1} - c_2T_{i+1,j}^{t+1} - c_3T_{i-1,j}^{t+1} - c_4T_{i,j+1}^{t+1} - c_5T_{i,j-1}^{t+1} = T_{ij}^t$$
- Using global numbering $l=0..(nm-1)$ and matrix notation $(I-A)T^{t+1} = T^t$
- c_1, c_2, \dots normally depend also on i, j (and possibly also on t)
- $(I-A)T^{t+1} = T^t$ can be solved with iterative solvers, e.g., CG,
with major internal compute step $p_{\text{new}} = Ap_{\text{old}}$ (**sparse-matrix-vector-multiply**)

Solver Categories (used in this talk)

- **Explicit:** (In each [time] step,) field variables are updated using neighbor information (no global linear or nonlinear solves)
- **Implicit:** Most or all variables are updated in a single global linear or nonlinear solve
- **Both categories** can be expressed (in the linear case) with a **sparse-matrix-vector-multiply**
 - Explicit: $T^{t+1} = (I+A)T^t$ [the 2- or 3-dim T is here expressed as a vector
 - Implicit: $(I-A)T^{t+1} = T^t$ over the global index $I=0..(mn-1)$]
- Vector T is a *logically* serialized storage of the field variables
- Matrix A is sparse
 - The rows reflect same position as in T, i.e., corresponds to one field variable
 - Elements reflect needed neighbor information

No principle differences between implicit and explicit

- Both categories can be expressed (in the linear case) with a sparse matrix
 - Explicit: $T^{n+1} = (I+A)T^n$ [the 2- or 3-dim T is here expressed as a vector]
 - Implicit: $(I-A)T^{n+1} = T^n$
- Implicit iterative solver:
 - Major (time-consuming) operation is sparse-matrix-vector-multiply
 - A_p with p is an interim vectors
 - Same operation as in the explicit scheme
- → Focus of this talk
 - Parallelization of simulation codes based on
 - Sparse matrix-vector-multiply
 - Domain decomposition for explicit time-step integration
 - Same methods can be used for A_p in implicit solvers

Parallelization

- Shared memory:
 - Independent iterations are distributed among threads,
 - Threads = parallel execution streams (on several CPUs) on the same shared memory
 - Mainly used to parallelize DO / FOR loops
 - E.g., with OpenMP
- Distributed memory:
 - Parallel processes, each with own set of variables
 - Message Passing between the processes, e.g., with MPI
 - Matrix (physically stored, or only logically) and all vectors are distributed among the processes
 - Optimal data distribution based on domain decomposition

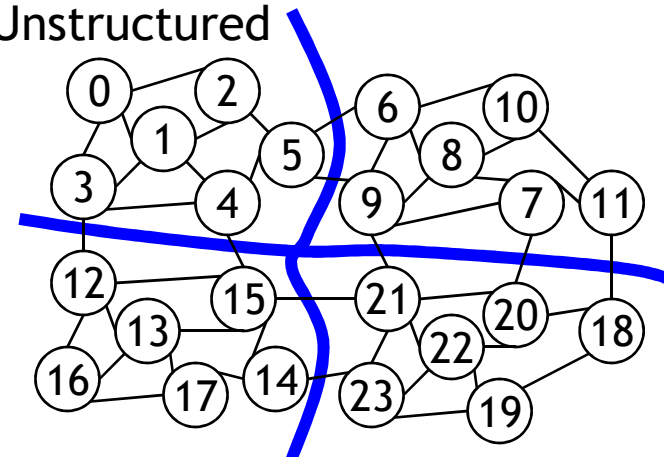
Domain Decomposition

- The simulation area (grid, domain) must be divided into several sub-domains
- Each sub-domain is stored in and calculated by a separate process

Cartesian

0	1	2	6	7	8
3	4	5	9	10	11
12	13	14	18	19	20
15	16	17	21	22	23

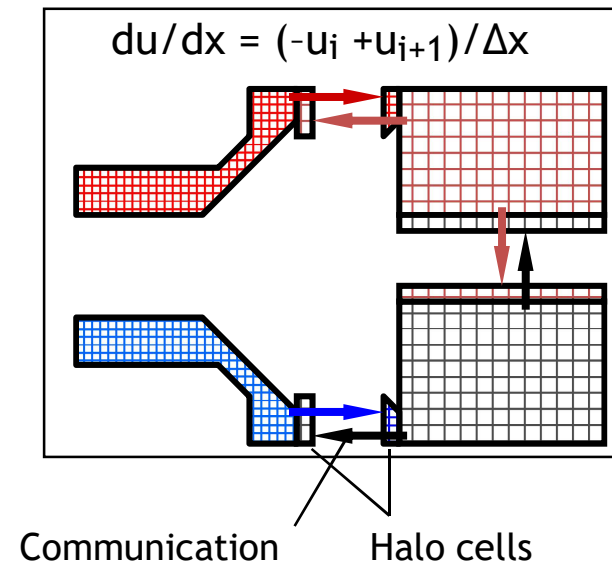
Unstructured



Examples with 4 sub-domains

Load Balancing and Communication Optimization

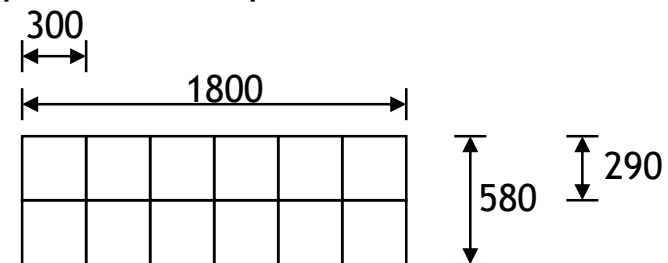
- Distribution of data and work implies
 - Idle time, if the work load distribution is not balanced
 - Additional overhead due to communication needs on sub-domain boundaries
 - Additional memory needs for halo (shadow, ghost) cells to store data from neighbors
- Major optimization goals:
 - Each sub-domain has the same work load
→ optimal load balance
 - The maximal boundary of all sub-domains is minimized
→ minimized communication



Cartesian Grids

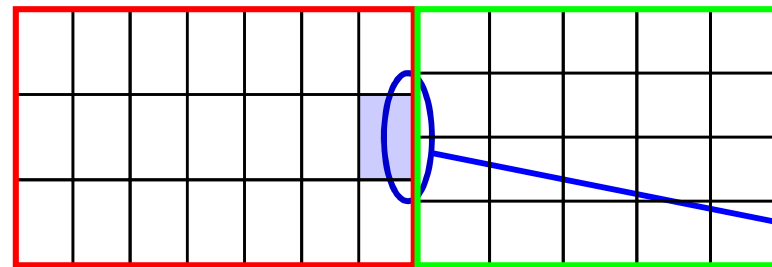
If each grid point requires same work:

- 2 dimensions:
Each sub-domain (computed by one CPU) should
 - Have the same size → optimal load balance
 - And should be quadratic → minimal communication
- Solution with factorization of the number of available processors
 - With MPI_Dims_create()
 - Caution: MPI_Dims_create tries to factorize the number of processes as quadratic as possible, e.g., $12 = 4 \times 3$,
 - But one must make the number of grid points quadratic!
 - Example - Task: Grid with 1800×580 grid points on 12 processors
Solution: 6 x 2 processes



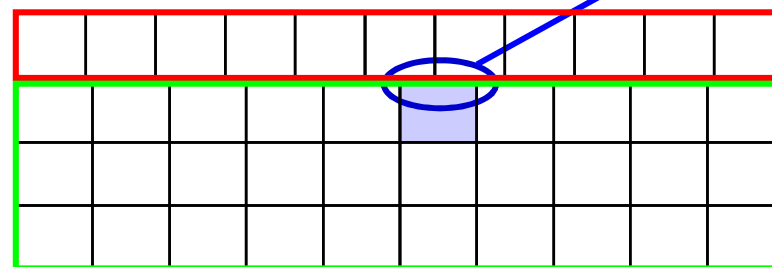
Cartesian Grids (2-dim, continued)

- Solution for any number of available processors
 - Two areas with different shape of their sub-domains
 - Horizontal split



Sub-domains at the split boundary have a more complicated neighborhood

- Vertical split



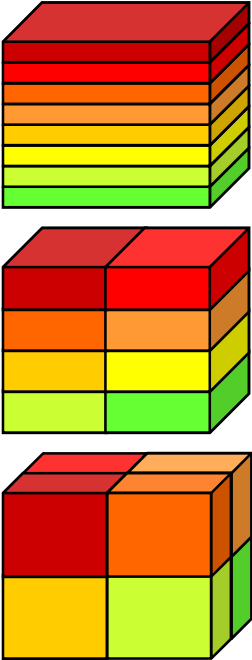
Examples with 41 sub-domains and 1800 x 580 grid



Cartesian Grids (3-dim)

- 3 dimensions
 - Same rules as for 2 dimensions
 - Usually optimum with 3-dim. domain decomposition & **cubic** sub-domains

Splitting in



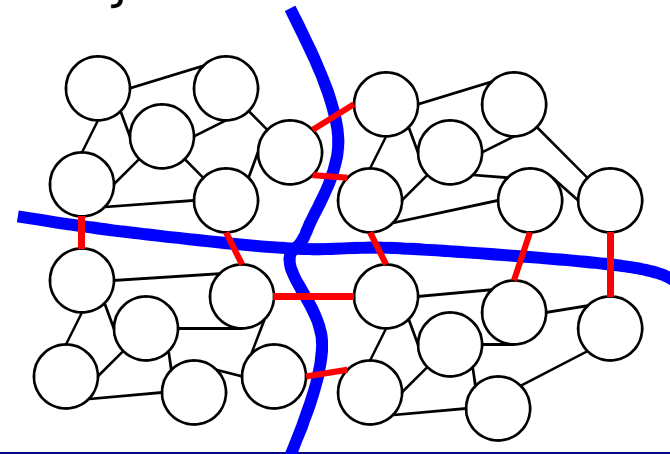
- **one** dimension:
communication
 $= n^2 * 2 * w * 1$
- **two** dimensions:
communication
 $= n^2 * 2 * w * 2 / p^{1/2}$
- **three** dimensions:
communication
 $= n^2 * 2 * w * 3 / p^{2/3}$

w = width of halo
 n^3 = size of matrix
 p = number of processors
 cyclic boundary
 → **two** neighbors
 in each direction

optimal for $p > 11$

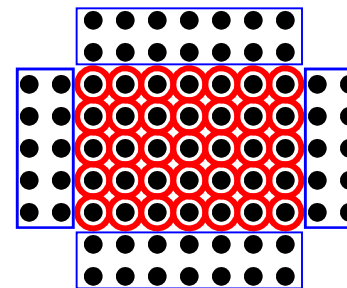
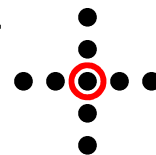
Unstructured Grids

- Mesh partitioning with special load balancing libraries
 - Metis (George Karypis, University of Minnesota)
 - <http://www.cs.umn.edu/~karypis/metis/metis.html>
 - ParMetis (internally parallel version of Metis)
 - <http://www.labri.fr/perso/pelegrin/scotch/>
 - Scotch & PT-Scotch (Francois Pellegrini, LaBRI, France)
 - <http://www.labri.fr/perso/pelegrin/scotch/>
 - Jostle (Chris Walshaw, University of Greenwich, GB)
 - <http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/>
 - Goals:
 - Same work load in each sub-domain
 - Minimizing the maximal number of neighbor-connections between sub-domains




Halo

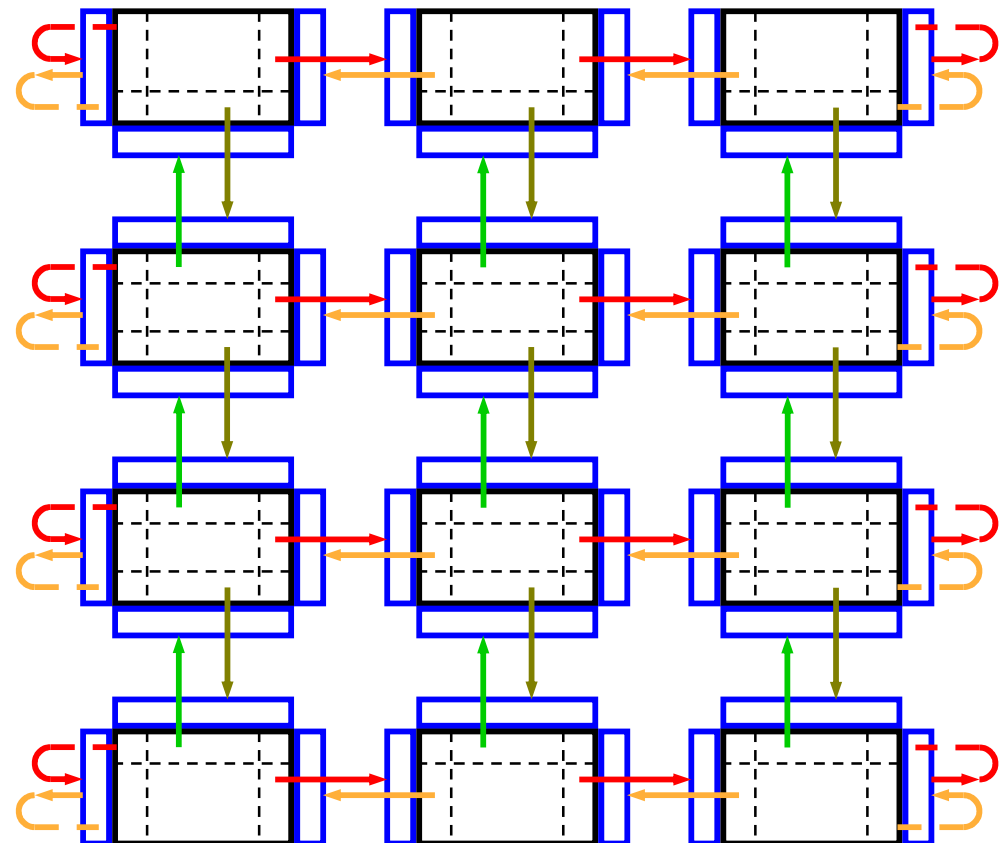
- Stencil:
 - To calculate a new grid point (○), old data from the stencil grid points (●) are needed
 - E.g., 9 point stencil
- Halo
 - To calculate the new grid points of a sub-domain, additional grid points from other sub-domains are needed.
 - They are stored in halos (ghost cells, shadows)
 - Halo depends on form of stencil



Communication: Send inner data into halo storage

One iteration in the

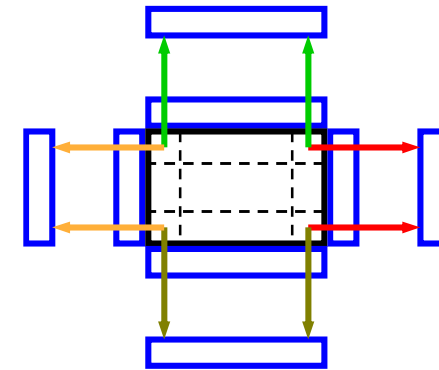
- **Serial code:**
 - $X_{\text{new}} = \text{function}(x_{\text{old}})$
 - $X_{\text{old}} = X_{\text{new}}$
- **Parallel code:**
 - Update halo
[=Communication, e.g., with
4 x MPI_Sendrecv ]
 - $X_{\text{new}} = \text{function}(x_{\text{old}})$
 - $X_{\text{old}} = X_{\text{new}}$




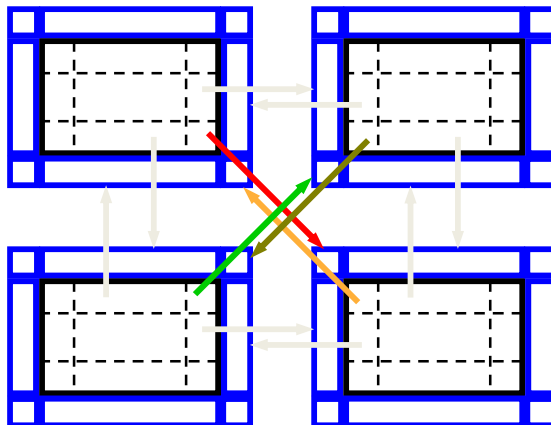
Examples with 12 sub-domains and horizontally cyclic boundary conditions

Corner problems

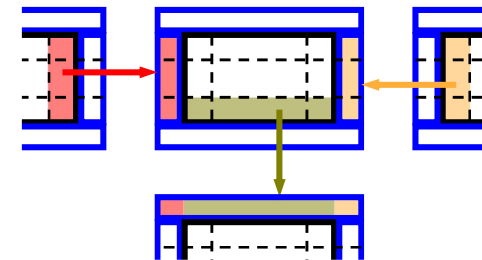
- MPI non-blocking send must not send inner corner data into more than one direction
 - Use `MPI_Sendrecv`
 - Or non-blocking `MPI_Irecv`



- Stencil with diagonal point, e.g., 
 - i.e., halos include corners →→→ substitute small corner messages:



- one may use 2-phase-protocol:
- normal horizontal halo communication
- include corner into vertical exchange



Speedup

$$T_{\text{parallel, } p} = f T_{\text{serial}} + (1-f) T_{\text{serial}} / p + T_{\text{communication}} + T_{\text{idleCPU}} / p$$

T_{serial} ,	wall-clock time needed with one processor
f	percentage T_{serial} of that can not be parallelized
$T_{\text{parallel, } p}$	wall-clock time needed with p processor
$T_{\text{communication}}$	average wall-clock time needed communication on each CPU
T_{idleCPU}	idle CPU-time due to bad load balancing
S_p	speedup on p processors := $T_{\text{serial}} / T_{\text{parallel, } p}$
E_p	efficiency on p processors := S_p / p

$$T_{\text{parallel, } p} = f T_{\text{serial}} + (1-f) T_{\text{serial}} / p + T_{\text{communication}} + T_{\text{idleCPU}} / p$$

$$E_p = (1 + f(p-1) + T_{\text{communication}} / (T_{\text{serial}}/p) + T_{\text{idleCPU}} / T_{\text{serial}})^{-1}$$

$$\approx 1 - \underbrace{f(p-1)}_{\ll 1} - \underbrace{T_{\text{communication}} / (T_{\text{serial}}/p)}_{\ll 1} - \underbrace{T_{\text{idleCPU}} / T_{\text{serial}}}_{\ll 1}$$

should be $\ll 1$

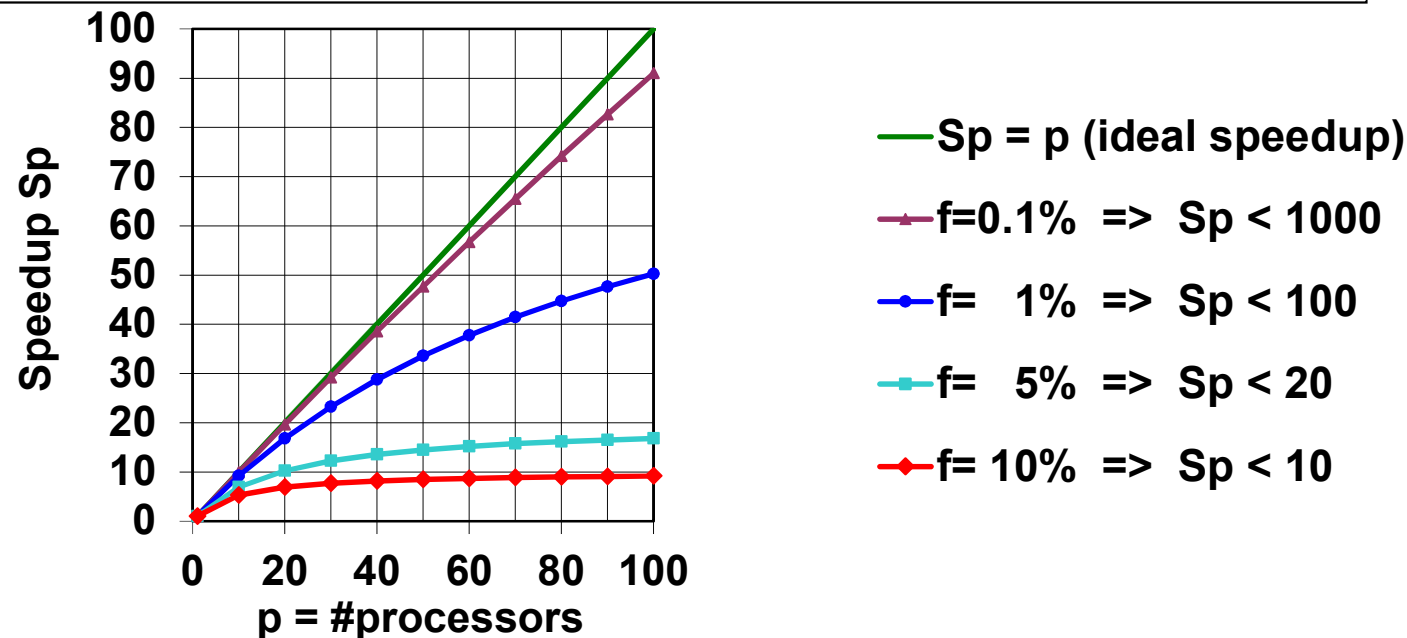
Amdahl's Law (if neglecting $T_{\text{communication}}$ and T_{idleCPU})

$$T_{\text{parallel}, p} = f \cdot T_{\text{serial}} + (1-f) \cdot T_{\text{serial}} / p$$

f ... sequential part of code that can not be done in parallel

$$S_p = T_{\text{serial}} / T_{\text{parallel}, p} = 1 / (f + (1-f) / p)$$

For $p \rightarrow$ infinity, speedup is limited by $S_p < 1 / f$



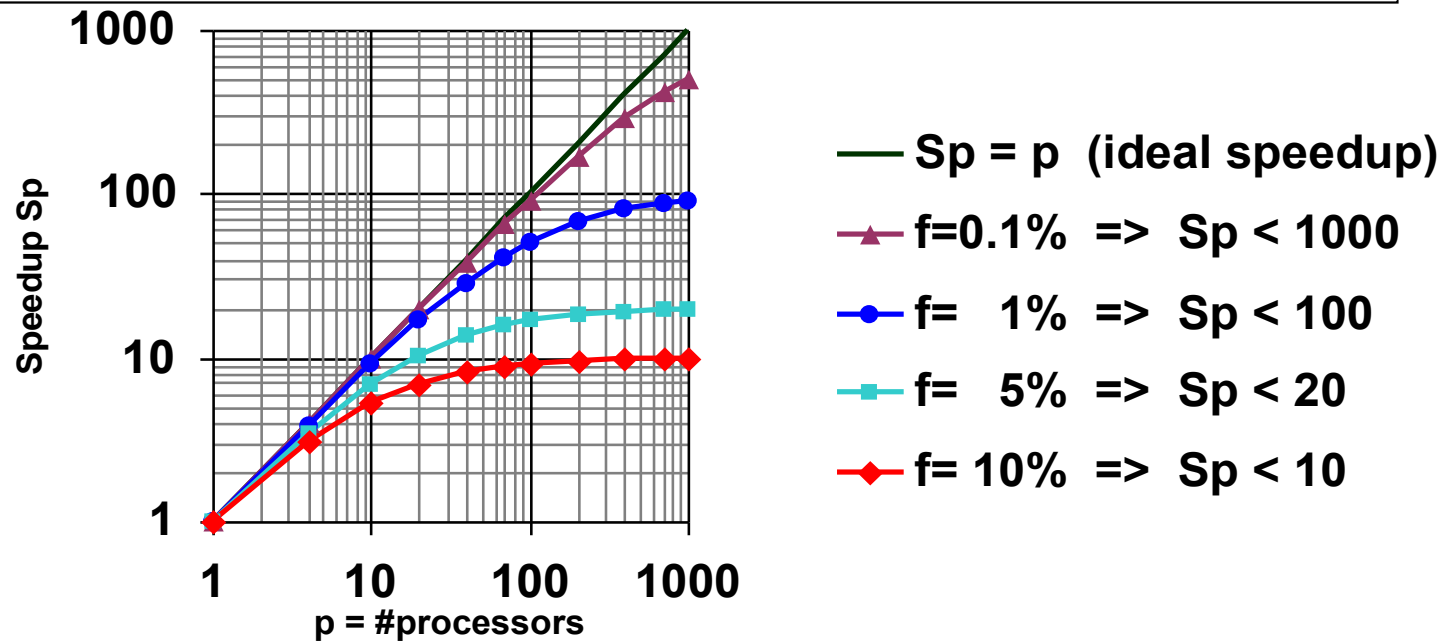
Amdahl's Law (double-logarithmic)

$$T_{\text{parallel}, p} = f \cdot T_{\text{serial}} + (1-f) \cdot T_{\text{serial}} / p$$

f ... sequential part of code that can not be done in parallel

$$S_p = T_{\text{serial}} / T_{\text{parallel}, p} = 1 / (f + (1-f) / p)$$


For $p \rightarrow \text{infinity}$, speedup is limited by $S_p < 1 / f$




Speedup problems

- Only ratio - no **absolute** performance value!
- Sometimes **super-scalar speedup**: $S_p > p$
 - Reason:
For speedup measurement, the total problem size is constant
→ The local problem size in each sub-domain may fit into **cache**
- **Scale-up**:
 - $S_c(p,N) = N / n$ with $T(1,n) = T(p,N)$
 - With $T(p,N)$ = **Time to solve problem of size N on p processors**
 - Compute **larger problem** with more processors in **same time**
- **Weak scaling**:
 - $T(p, p \cdot n) / T(1,n)$ is reported,
 - I.e., problem size per process ($N = p \cdot n$) is fixed
 - Constant ratio = 100% efficiency

Example (2-dim)

- 2-dim:
 - 9-point-stencil  or
 - 300x300 grid points on each sub-domain
 - 16 byte communication data per grid point
 - 100 FLOP per grid point
 - 20 MB/s communication bandwidth per process
(this bandwidth must be available on all processes at the same time)
 - 1 GFLOP/s peak processor speed
 - 10% = real application / peak processor speed
 - $T_{\text{communication}} = (9-1) \cdot 300 \cdot 16 \text{ byte} / 20 \text{ MB/s} = 1.92 \text{ ms}$
 - $T_{\text{serial}} / p = 300 \cdot 300 \cdot 100 \text{ FLOP} / (1 \text{ GFLOP/s} \cdot 10\%) = 90 \text{ ms}$
 - $T_{\text{communication}} / (T_{\text{serial}}/p) = 1.92 \text{ ms} / 90 \text{ ms} = 0.021 \ll 1$
 - Only 2.1 % reduction of the parallel efficiency due to communication

Example (3-dim)

- 3-dim:
 - 13-point-stencil 
 - 50x50x50 grid points on each sub-domain
 - 16 byte communication data per grid point
 - 100 FLOP per grid point
 - 20 MB/s communication bandwidth per process
(this bandwidth must be available on all processes at the same time)
 - 1 GFLOP/s peak processor speed
 - 10% = real / peak processor speed
 - $T_{\text{communication}} = (13-1) \cdot 50 \cdot 50 \cdot 16 \text{ byte} / 20 \text{ MB/s} = 24 \text{ ms}$
 - $T_{\text{serial}} / p = 50 \cdot 50 \cdot 50 \cdot 100 \text{ FLOP} / (1 \text{ GFLOP/s} \cdot 10\%) = 125 \text{ ms}$
 - $T_{\text{communication}} / (T_{\text{serial}}/p) = 24 \text{ ms} / 125 \text{ ms} = 0.192 < 1$
 - 19 % reduction of the parallel efficiency due to communication

Implicit Iterative Solver

- The solution path:
 - Real world
 - Partial differential equation
 - Discretization (2/3-dimensions = indices i, j, k)
 - Global index $(i, j, k) \rightarrow I$
 - Algebraic equation $Ax=b$
 - with sparse-matrix $A = (a_{I,J})_{I=1..N, J=1..N}$
 - boundary vector $b = (b_I)_{I=1..N}$
 - solution vector $x = (x_I)_{I=1..N}$
- Solve $Ax=b$ with iterative solver:
 - Major computational steps:
 - Sparse-matrix-vector-multiply: Av , with v =interims vector
 - Scalar product: (v_1, v_2)

Example: CG Solver

```

Initialize matrix A;           Initialize boundary condition vector b;
Initialize i_max (≤ size of A); Initialize ε (>0);   Initialize solution vector x;
/* p = b - Ax ;   */         p = x;           /* Reason: */
/* substituted by */         v = Ap;          /* Parallelization halo needed */
                             p = b - v;      /* For same vector (p) as in loop */

r = p;
α = (|| r ||2)2;
for ( i=0; (i < i_max) && (α > ε); i++)
{
  v = Ap;
  λ = α / (v,p)2;
  x = x + λp;
  r = r - λv;
  αnew = ( || r ||2 )2;
  p = r + (αnew/α)p;
  α = αnew;
}
Print x, √α, ||b-Ax||2;

```

←

See, e.g.,
 Andreas Meister: Numerik linearer Gleichungssysteme.
 Vieweg, 2nd ed., 2005, p. 124.

Parallel Iterative Solver

To implement domain decomposition:

- Go back to 2- or 3-dim domain with the 2 or 3 index variables (i,j) or (i,j,k)
 - $A = (a_{i,j,k; i',j',k'})_{i=1..l, j=1..m, k=1..n; i'=1..l, j'=1..m, k'=1..n}$
 - $p = (p_{i,j,k})_{i=1..l, j=1..m, k=1..n}$
 - Matrix-vector-multiply:


```

do (i=1, i<l, i++)
  do (j=1, j<m, j++)
    do (k=1, k<n, k++)
      vi,j,k = 0
      sparse (unrolled) loops over i', j', k'
      vi,j,k = vi,j,k + ai,j,k; i',j',k' * pi',j',k'
          
```
- Domain decomposition in the 2/3-dim space (and not in the 1-dim algebraic space $I=1..N$)

Distributed Data

- Matrix A
 - Boundary condition vector b
 - Solution vector x
 - Residual vector r
 - Gradient vector p
-
- Halos are needed in this algorithm
only for p
(only p is multiplied with A)

```
Initialize matrix A;  
Initialize boundary condition vector b;  
Initialize i_max ( $\leq$  size of A); Initialize  $\epsilon$  ( $>0$ );  
Initialize solution vector x;  
p = x;  
v = Ap;  
p = b - v;  
r = p;  
 $\alpha = (\|r\|_2)^2$ ;  
for ( i=0; (i < i_max) && ( $\alpha > \epsilon$ ); i++)  
{  
    v = Ap;  
     $\lambda = \alpha / (v,p)_2$ ;  
    x = x +  $\lambda p$ ;  
    r = r -  $\lambda v$ ;  
     $\alpha_{new} = (\|r\|_2)^2$ ;  
    p = r + ( $\alpha_{new}/\alpha$ )p;  
     $\alpha = \alpha_{new}$ ;  
}  
Print x,  $\sqrt{\alpha}$ ,  $\|b - Ax\|_2$ ;
```

Parallel Operations

Operation that include communication

- Halo exchange for vector p to prepare matrix-vector-multiply Ap
- Scalar product (v_1, v_2)
 - Algorithm:
 - Compute local scalar product
 - Compute global scalar product with `MPI_Allreduce(..., MPI_SUM, ...)` over all local scalar product values
- Norm $\|r\|_2$
 - Algorithm: same as *scalar product*

Operations without communication

- Matrix-vector-multiply: $v = Ap$
 - requires updated halo
- AXPY: x or $y = \alpha x + y$

```

Initialize matrix A;
Initialize boundary condition vector b;
Initialize i_max ( $\leq$  size of A); Initialize  $\epsilon$  ( $>0$ );
Initialize solution vector x;
p = x;
v = Ap;
p = b - v;
r = p;
 $\alpha = (\|r\|_2)^2$ ;
for ( i=0; (i < i_max) && ( $\alpha > \epsilon$ ); i++)
{
    v = Ap;
     $\lambda = \alpha / (v,p)_2$ ;
    x = x +  $\lambda p$ ;
    r = r -  $\lambda v$ ;
     $\alpha_{new} = (\|r\|_2)^2$ ;
    p = r + ( $\alpha_{new}/\alpha$ )p;
     $\alpha = \alpha_{new}$ ;
}
Print x,  $\sqrt{\alpha}$ ,  $\|b-Ax\|_2$ ;

```

Parallel Solver - Optimization Hints

- Preserve regular pattern of the matrix!
- Don't use indexed array access (`p(indexarr(i))`), if it is not really necessary
- Always use many arrays
 `REAL :: t(1000000), p(1000000), v(1000000)`

- (instead of one array of a structure)

```
TYPE data_struct_of_one_point  
    REAL :: t  
    REAL :: p  
    REAL :: v  
END TYPE data_struct_of_one_point  
TYPE (data_struct_of_one_point) :: points(1000000)
```


General Optimization Hints

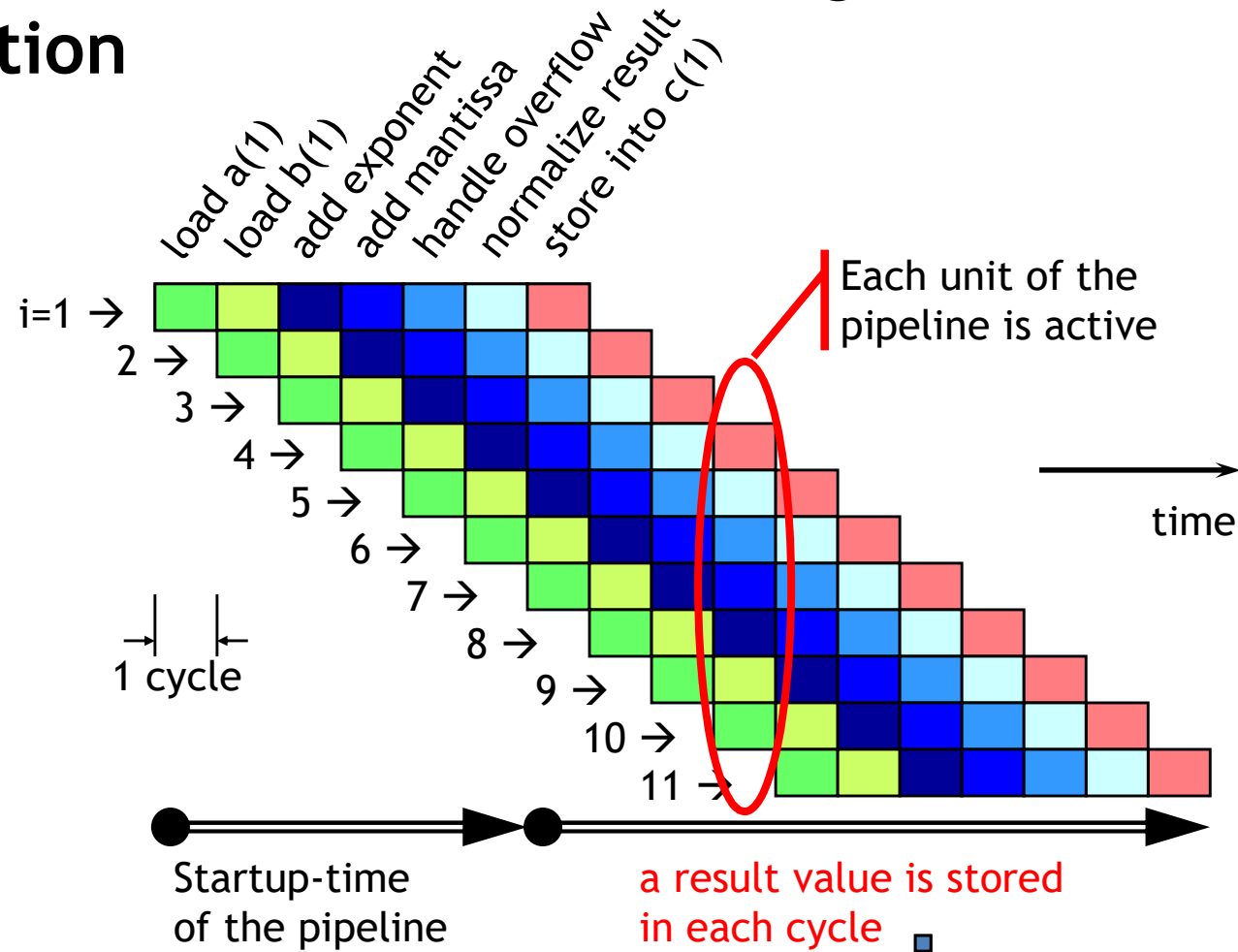
- Non-cubic may cause better computational efficiency
 - 50x50x50 cubic \rightarrow boundary = $6 \times 50 \times 50 = 15,000$
 - vs. 100x25x50 \rightarrow boundary = $2 \times 100 \times 25$
 $+ 2 \times 100 \times 50$
 $+ 2 \times 25 \times 50 = 17,500$
 - 16 % larger boundary, and
 - (expecting totally ~10% communication)
 \rightarrow 1.6% additional communication overhead
 - 100% longer most inner loop,
which may cause more than 1.6 % computational speedup!!!

General Optimization Hints (continued)

- Overlapping of communication and computation
 - On MPP (massively parallel processors) systems and clusters of single-CPU-nodes:
Overlapping normally not needed
 - Advantages on clusters of SMP (shared memory) nodes (hybrid hardware with hybrid programming model):
1 CPU communicates while other CPUs compute
 - One must separate
 - Computation that needs halo data
→ cannot be overlapped with communication
 - Computation of grid points that do not need halo data
→ can be overlapped with communication
- Preserve pipelining / vectorization with your parallelization

Pipelining and Instruction Chaining / Vectorization

- $\vec{c} = \vec{a} + \vec{b}$



How to implement sparse-matrix-vector-multiply I

- How can I implement the loops efficiently

```
do i=0,m-1
  do j=0,n-1
    Tnew(i,j) = (1+c1)T(i,j) + c2T(i+1,j) + c3T(i-1,j) + c4T(i,j+1) + c5T(i,j-1)
  end do
end do
```

- On vector-systems:
 - T and Tnew are defined on (-1:m, -1:n),
 - But the loop is done only on (0:m-1, 0:n-1)
 - The most-inner loop may be too small for good vectorization [e.g., on NEC SX-6, vector length should be a multiple of 256]
 - Interpret arrays as 1-dimensional T, Tnew(0 : (m+2)(n+2)-1)
 - One loop over all elements
 - Ignore senseless values in Tnew on boundary

How to implement sparse-matrix-vector-multiply II

- On cache-based systems:
 - Move small squares (2-dim) or cubes (3-dim) over the total area:

```

do iout=0,m-1,istride
  do jout=0,n-1,jstride
    do i=iout, min(m-1, iout+istride-1)
      do j=jout, min(n-1, jout+jstride-1)
        Tnew(i,j) = (1+c1)T(i,j) + c2T(i+1,j) + c3T(i-1,j) + c4T(i,j+1) + c5T(i,j-1)
      end do
    end do
  end do
end do
    
```

5 loaded stencil values are reused via cache in the next i or i iterations

e.g., istride=jstride=10

→ 100 inner iterations need 500 T-values
 → 140 from memory + 360 from cache } used for 900 FLOP

How to implement sparse-matrix-vector-multiply III

Important principle → Single source!!!

```
#ifdef _OPENMP
    special OpenMP parallelization features
#endif
```

```
• #ifdef USE_MPI
    MPI_Init(...);
    MPI_Comm_size(..., &size); MPI_Comm_rank(..., &my_rank);
#else
    size=1; my_rank=0;
#endif
...
```

```
#ifdef USE_CACHE
    cache-version of sparse-matrix-vector-multiply
#else
    vector-version
#endif
```

Classes of iterative solvers

- Parallel step algorithms:
 - $x_{\text{iter}} := \text{func}(x_{\text{iter}-1})$
 - e.g. Jacobi, CG, Richardson, ...
 - No problems with vectorization and parallelization
- Single step algorithms:
 - $x_{\text{iter}} := \text{func}(x_{\text{iter}-1}$, some elements of x_{iter})
 - E.g. Gauß-Seidel, SOR, ...
 - Vectorization and parallelization is possible with red/black (checkerboard) method

Parallelization of single-step algorithms

Single step algorithms

- Example: SOR

$$- x_{m+1,i} := (1-\omega)x_{m,i} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_{m+1,j} - \sum_{j=i}^n a_{ij} x_{m,j} \right) \quad (m = \#iteration)$$

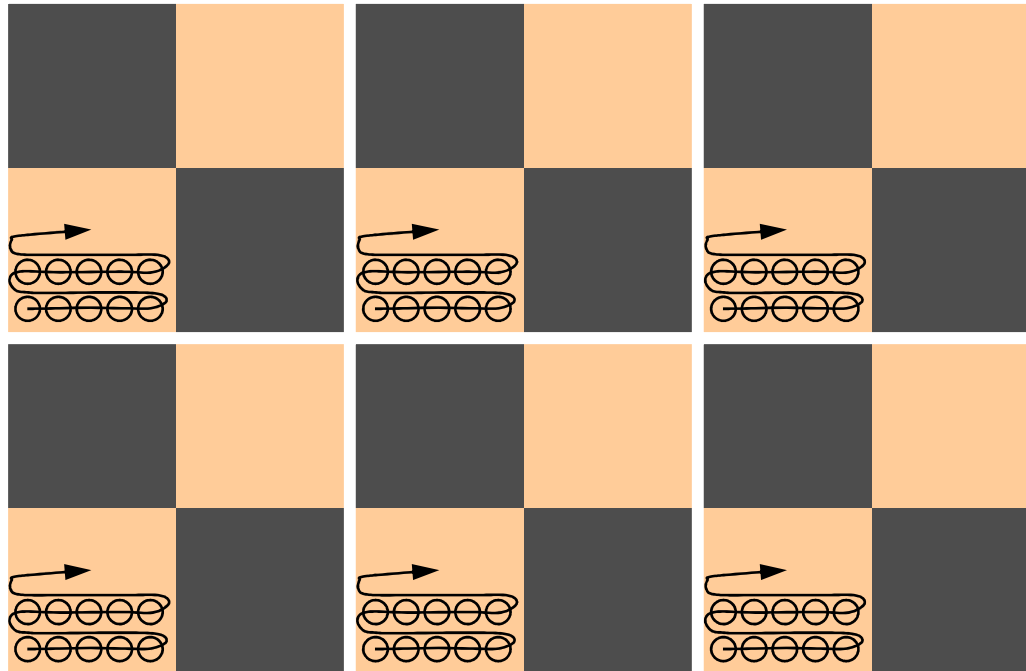
- If only direct neighbor exists,
i.e. $a_{ij} \neq 0$ for $j = "i+x", "i-x", "i+y", "i-y"$
- and "i-x" and "i-y" are indexes less than i, then

→ $x_{m+1,i}$

$$:= (1-\omega)x_{m,i} + \frac{\omega}{a_{ii}} \left(b_i - \underbrace{a_{i,i-x}x_{m+1,i-x} - a_{i,i-y}x_{m+1,i-y}}_{\text{left and lower x value must be already computed!}} - a_{i,i+x}x_{m,i+x} - a_{i,i+y}x_{m,i+y} \right)$$

Left and lower x value must be already computed!
 Problem for parallelization and vectorization!

Red/black (checkerboard) ordering



- 6 nodes
- Each node has
 - 2 red and
 - 2 black checkers

- First, compute all red checkers, then communicate boundary
- Second, compute all black checkers and communicate boundary
- Inside of each checker: Use original sequence
- Parallel version is **not** numerically identical to serial version!!!

Literature

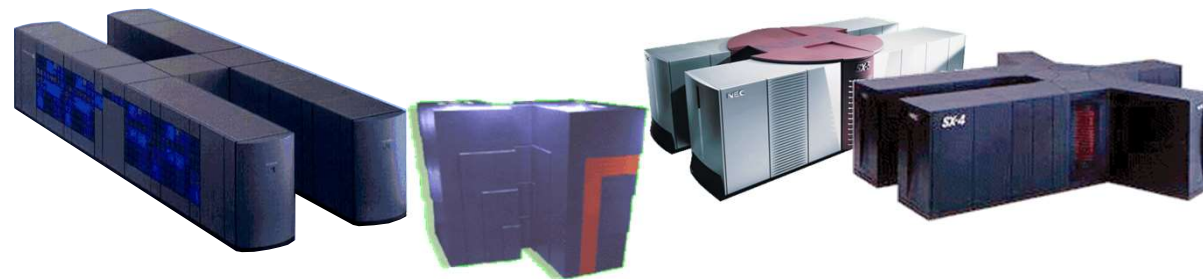
- G. Fox, M. Johnson, G. Lyzenga, S. Otto, S. Salmon, D. Walker:
Solving Problems on Concurrent Processors.
Prentice-Hall, 1988.
- Barry F. Smith, Petter E. Bjørstad, William D. Gropp:
Domain Decomposition
Parallel Multilevel Methods for Elliptic Partial Differential Equations.
Cambridge University Press, 1996.
- Andreas Meister:
Numerik linearer Gleichungssysteme.
Vieweg, 1999.

Outline

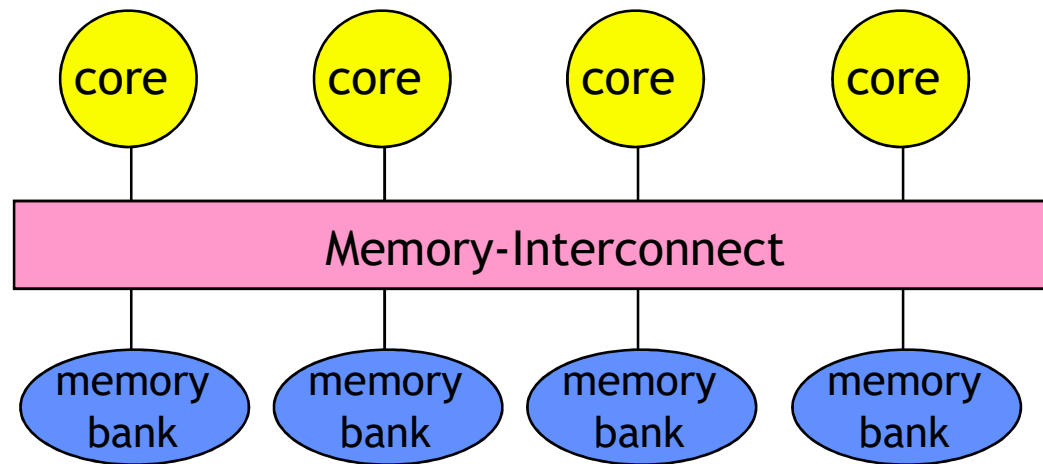
- Parallelization of explicit or implicit solver
- Parallel hardware
- Parallel programming models
- Parallelization scheme

Major Parallel Hardware Architectures

- Shared Memory
 - SMP = Symmetric multiprocessing
- Distributed Memory
 - DMP = Distributed memory parallel
- Hierarchical memory systems
 - Combining both concepts

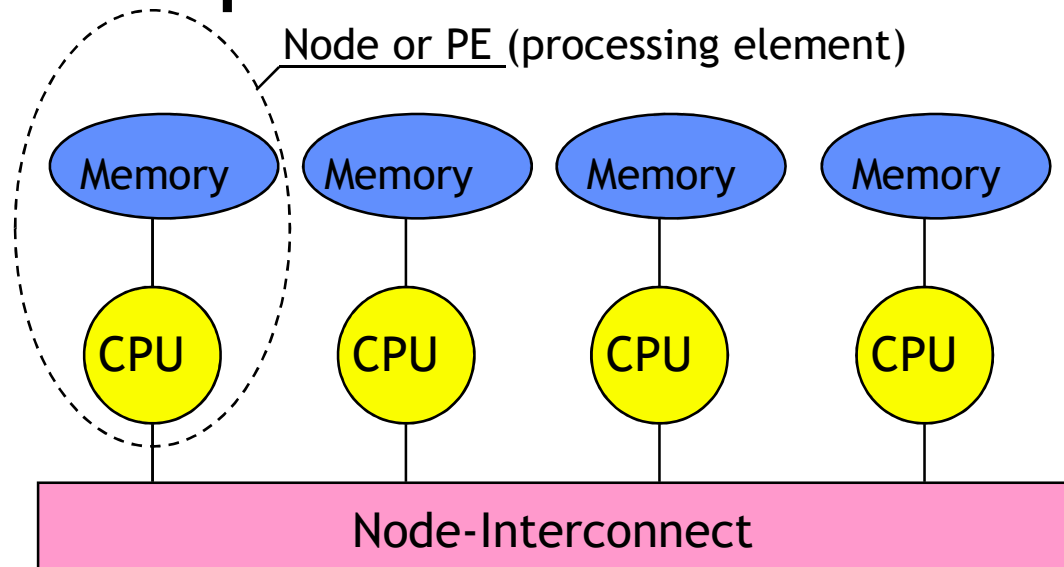


Multiprocessor - shared memory



- All CPUs are connected to all memory banks with same speed
- Uniform Memory Access (UMA)
- Symmetric Multi-Processing (SMP)
- Network types, e.g.
 - Crossbar → independent access from each CPU
 - BUS → one CPU can *block* the memory access of the other CPUs

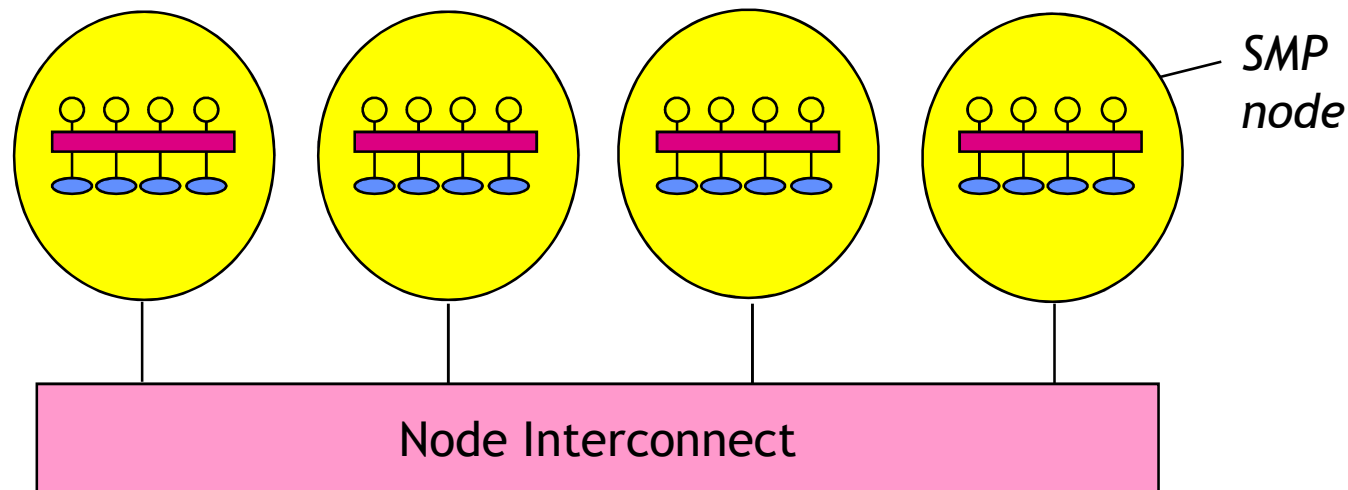
Multicomputer - distributed memory



- Nodes are coupled by a node-interconnect
- Each CPU:
 - Fast access to its own memory
 - but slower access to other CPU's memories
- Non-Uniform memory Access (NUMA)
- Different network types, e.g. BUS, torus, crossbar

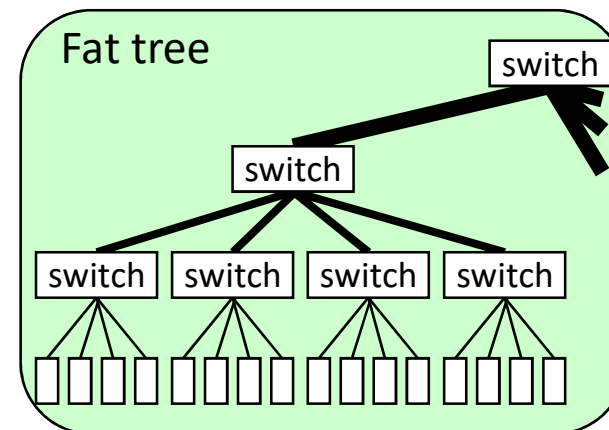
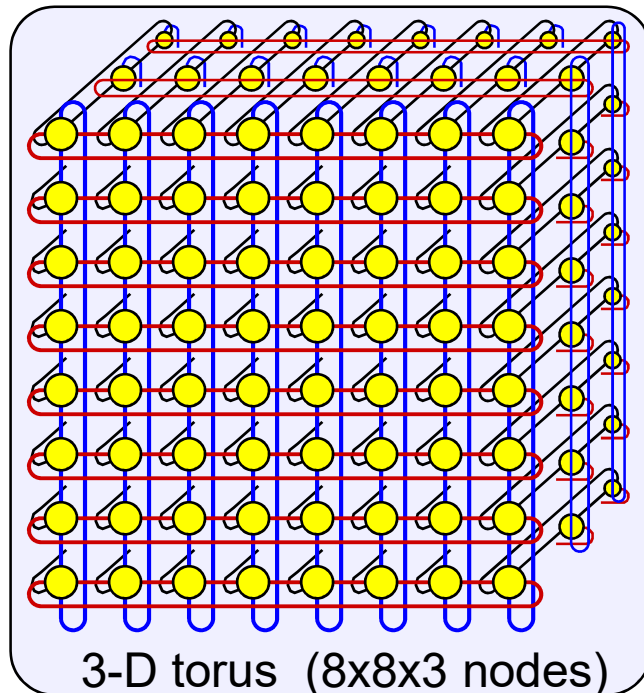
Hybrid architectures

- Most modern high-performance computing (HPC) systems are clusters of SMP nodes



- SMP (symmetric multi-processing) inside of each node
- DMP (distributed memory parallelization) on the node interconnect

Interconnect topologies



Outline

- Parallelization of explicit or implicit solver
- Parallel hardware
- Parallel programming models
 - Parallelization Strategies [51-55] → Models [56] → OpenMP [56-58] → OpenMP-tasks [59-61] → MPI [62-66]
 - Limitations [67-68] → Advantages & Challenges [69]
- Parallelization scheme

Parallelization strategies – hardware resources

- Two major resources of computation:
 - Processor
 - Memory
- Parallelization means
 - **Distributing work** to processors
 - **Distributing data** (if memory is distributed)and
 - **Synchronization** of the distributed work
 - **Communication** of *remote* data to *local* processor (if memory is distr.)
- Programming models offer a combined method for
 - Distribution of work & data, synchronization and communication

Distributing Work & Data

Work decomposition

- Based on loop decomposition

do i=1,100

→ i=1,25

i=26,50

i=51,75

i=76,100

Data decomposition

- All work for a local portion of the data is done by the local processor

A(1:20, 1: 50)

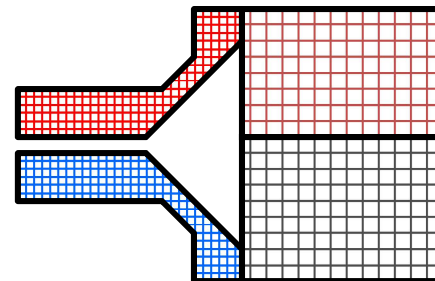
A(1:20, 51:100)

A(21:40, 1: 50)

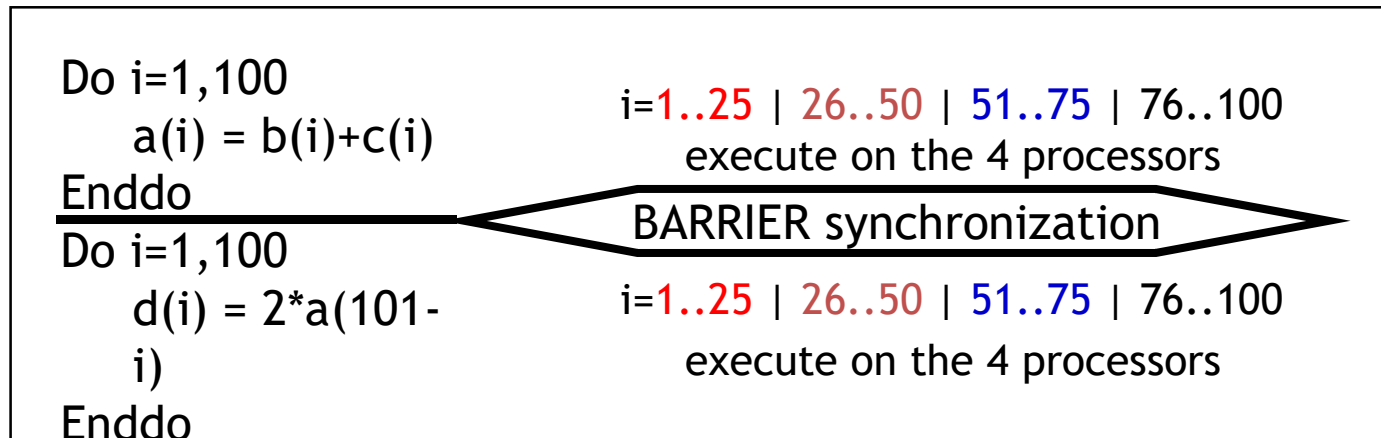
A(21:40, 51:100)

Domain decomposition

- Decomposition of work and data is done in a higher model, e.g. in the reality



Synchronization



- **Synchronization**
 - Is necessary
 - May cause
 - Idle time on some processors
 - Overhead to execute the synchronization primitive

Communication

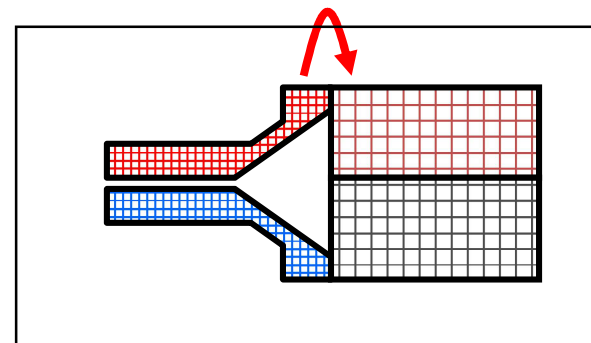
```
Do i=2,99
  b(i) = a(i) + f*(a(i-1)+a(i+1)-2*a(i))
Enddo
```

- Communication is necessary on the boundaries

- E.g. $b(26) = a(26) + f*(a(25)+a(27)-2*a(26))$

a(1:25),	b(1:25)
a(26,50),	b(51,50)
a(51,75),	b(51,75)
a(76,100),	b(76,100)

- E.g. at domain boundaries



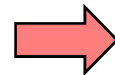
Major Programming Models

① OpenMP

- Shared Memory **Directives**
- To define the work decomposition
- No data decomposition
- Synchronization is implicit (can be also user-defined)
- **OpenMP - task based parallelization**
 - Task based parallelization
 - User specifies tasks and task dependencies with **directives**
 - Parallelization (and synchronization) is implicit
- **MPI (Message Passing Interface)**
 - User specifies how work & data is distributed
 - User specifies how and when communication has to be done
 - By calling MPI communication **library-routines**

Shared Memory Directives - OpenMP I

Real :: A(n,m), B(n,m)


 Data definition

!\$OMP PARALLEL DO




do j = 2, m-1


 Loop over y-dimension

do i = 2, n-1


 Vectorizable loop over x-dimension

B(i,j) = ... A(i,j)
 ... A(i-1,j) ... A(i+1,j)
 ... A(i,j-1) ... A(i,j+1)

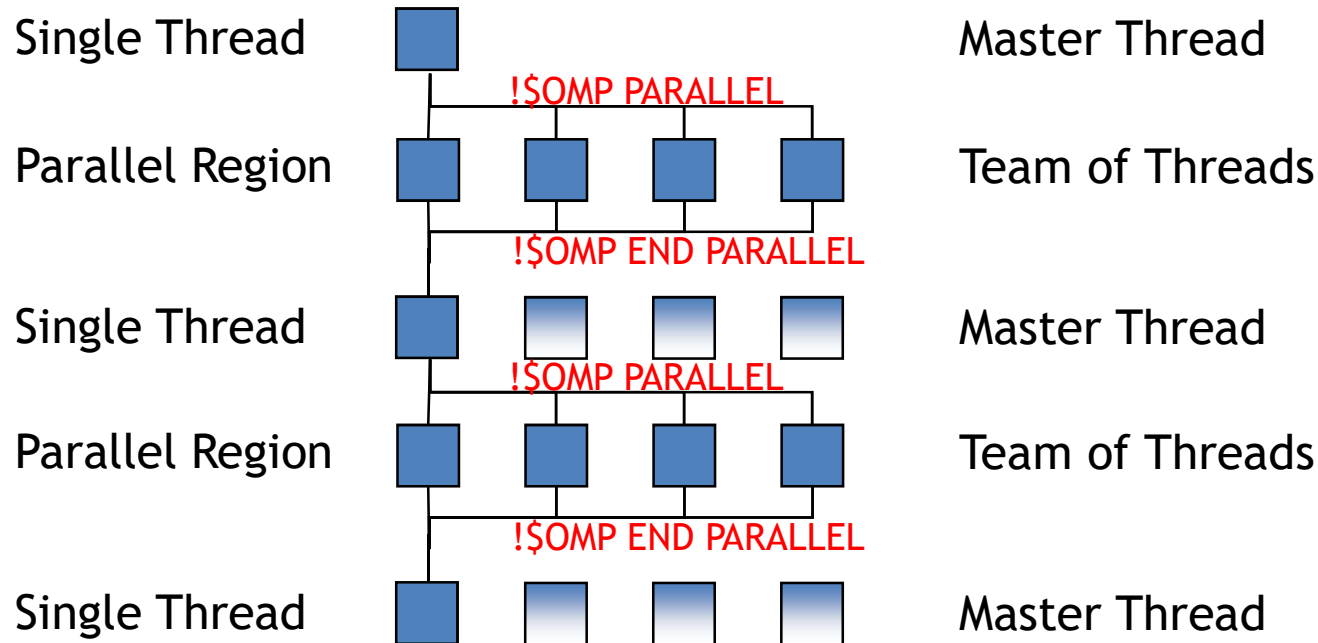

 Calculate B,

 using upper and lower,

 left and right value of A

end do

end do

!\$OMP END PARALLEL DO

Shared Memory Directives - OpenMP II



Shared Memory Directives - OpenMP III

- OpenMP
 - Standardized shared memory parallelism
 - Thread-based
 - The user has to specify the work distribution explicitly with directives
 - No data distribution, no communication
 - Mainly loops can be parallelized
 - Compiler translates OpenMP directives into thread-handling
 - Standardized since 1997
- Automatic SMP-Parallelization
 - E.g., Compas (Hitachi), Autotasking (NEC)
 - Thread based shared memory parallelism
 - With directives (similar programming model as with OpenMP)
 - Supports automatic parallelization of loops
 - Similar to automatic vectorization

Major Programming Models - Task based programming

① OpenMP

- Shared Memory Directives
- to define the work decomposition
- no data decomposition
- synchronization is implicit (can be also user-defined)

② OpenMP - task based parallelization

- Task based parallelization
- User specifies tasks and task dependencies with **directives**
- Parallelization (and synchronization) is implicit

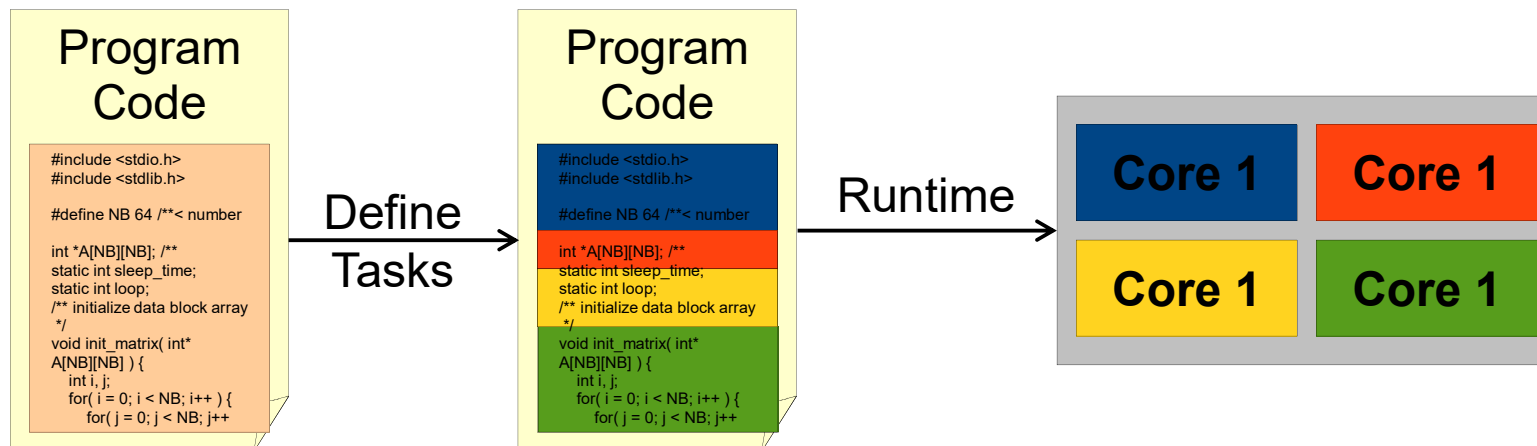
- MPI (Message Passing Interface)

- User specifies how work & data is distributed
- User specifies how and when communication has to be done
- by calling MPI communication **library-routines**

Task based Parallelisation

Basic Idea:

- **Programmer defines tasks** as basic units for parallel execution where a task represents a more or less self-contained part of the code.
- **Runtime decides on the execution** of the tasks, managing the difficult problem of their ordering and hardware placement



Task based Parallelisation - OpenMP

- Task + dependency model introduced with OpenMP 4.0
- OpenMP tasks defined with `#prgams omp task`
- Dependencies between tasks specified via input and output parameters using `depend(in|out)` clause
- Uses the task set of a surrounding parallel region as workers to execute the tasks

Task based Parallelisation - OpenMP Example

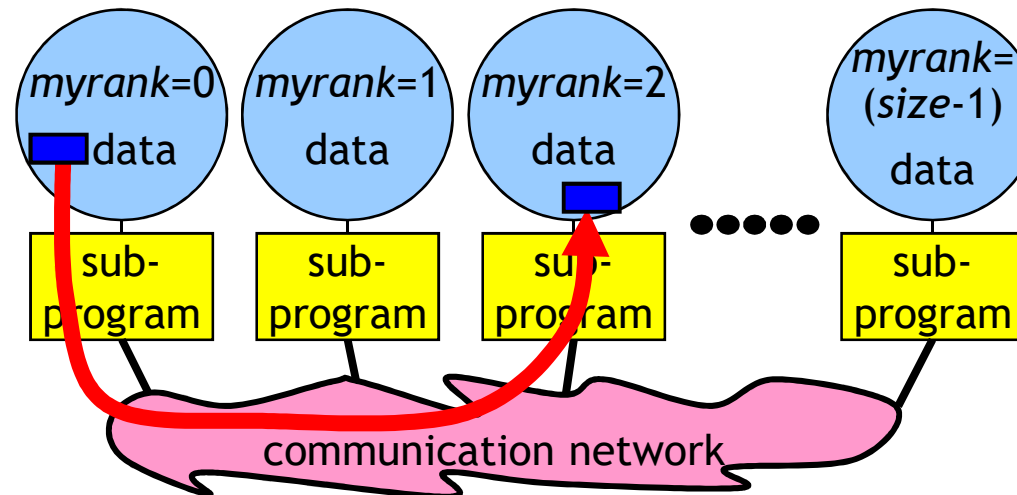
```
#pragma omp parallel
{
#pragma omp single
{
    int x, y, z;
    #pragma omp task depend( out: x )
    x = init();
    #pragma omp task depend( in: x ) depend( out: y )
    y = f(x);
    #pragma omp task depend( in: x ) depend( out: z )
    z = g(x);
    #pragma omp task depend( in: y, z )
    finalize(y, z);
}}
```

Major Programming Models - MPI

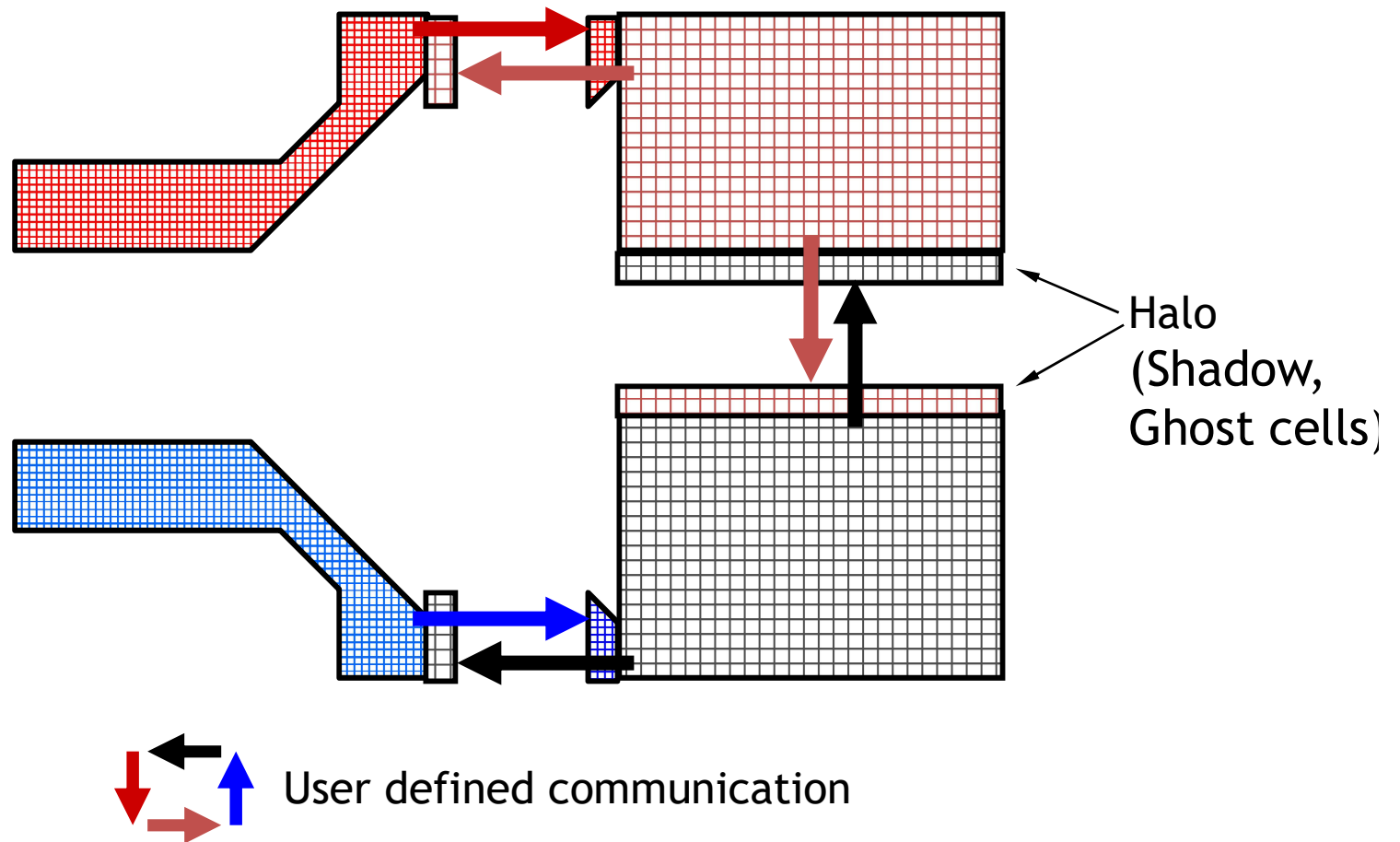
- ① OpenMP
 - Shared Memory Directives
 - to define the work decomposition
 - no data decomposition
 - synchronization is implicit (can be also user-defined)
- ② OpenMP - task based parallelisation
 - Task based parallelisation
 - User specifies tasks and task dependencies with directives
 - Parallelisation (and synchronization) is implicit
- ③ MPI (Message Passing Interface)
 - User specifies how work & data is distributed
 - User specifies how and when communication has to be done
 - By calling MPI communication **library-routines**

Message Passing Program Paradigm - MPI I

- Each processor in a message passing program runs a *sub-program*
 - Written in a conventional sequential language, e.g., C or Fortran,
 - Typically the same on each processor (SPMD)
- All work and data distribution is based on value of *myrank*
 - Returned by special library routine
- Communication via special send & receive routines (*message passing*)



Additional Halo Cells - MPI II



Message Passing - MPI III

```
Call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
Call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierror)
m1 = (m+size-1)/size; ja=1+m1*myrank; je=max(m1*(myrank+1), m)
jax=ja-1; jex=je+1 // extended boundary with halo
```

```
Real :: A(n, jax:jex), B(n, jax:jex)
do j = max(2,ja), min(m-1,je)
  do i = 2, n-1
    B(i,j) = ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
```

- Data definition
- Loop over y-dimension
- Vectorizable loop over x-dimension
- Calculate B,
- using upper and lower,
- left and right value of
- A

```
Call MPI_Send(.....) ! - sending the boundary data to the neighbors
Call MPI_Recv(.....) ! - receiving from the neighbors,
                    ! storing into the halo cells
```

Summary – MPI IV

- MPI (Message Passing Interface)
 - Standardized distributed memory parallelism with message passing process-based
 - The user has to specify the work distribution & data distribution & all communication
 - Synchronization implicit by completion of communication
 - The application processes are calling MPI library-routines
 - Compiler generates normal sequential code
 - Typically domain decomposition is used
 - Communication across domain boundaries
 - Standardized
 - MPI-1: Version 1.0 (1994), Version 1.1 (1995), Version 1.2 (1997)
 - MPI-2: Version 2.0 (1997), Version 2.1 (2008), Version 2.2 (2009)
 - MPI-3: Version 3.0 (2012), Version 3.1 (2015)

Limitations I

- Automatic Parallelization
 - The compiler
 - Has no global view
 - Cannot detect independencies, e.g., of loop iterations
 - Parallelizes only parts of the code
 - Only for shared memory and ccNUMA systems, see OpenMP
- OpenMP
 - Only for shared memory and ccNUMA systems
 - Mainly for loop parallelization with directives
 - Only for medium number of processors
 - Explicit domain decomposition also via rank of the threads

Limitations II

- HPF
 - Set-compute-rule may cause a lot of communication
 - HPF-1 (and 2) not suitable for irregular and dynamic data
 - JaHPF may solve these problems, but with additional programming costs
 - Can be used on any platform
- MPI
 - The amount of your hours available for MPI programming
 - Can be used on any platform, but communication overhead on shared memory systems

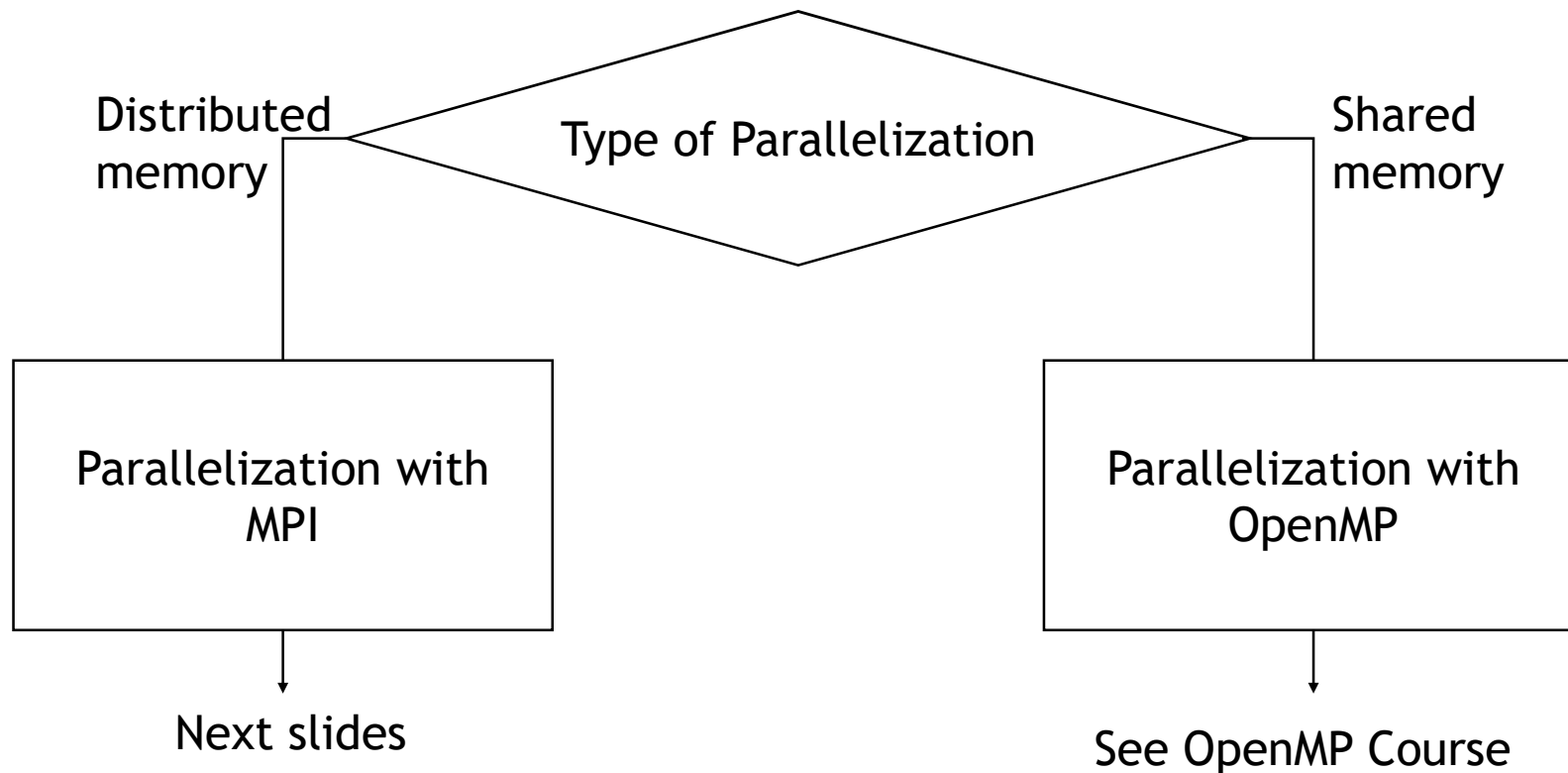
Advantages and Challenges

	OpenMP	HPF	MPI
Maturity of programming model	++	+	++
Maturity of standardization	+	+	++
Migration of serial programs	++	0	--
Ease of programming (new progr.)	++	+	-
Correctness of parallelization	-	++	--
Portability to any hardware architecture	-	++	++
Availability of implementations of the stand.	+	+	++
Availability of parallel libraries	0	0	0
Scalability to hundreds/thousands of processors	--	0	++
Efficiency	-	0	++
Flexibility – dynamic program structures	-	-	++
– irregular grids, triangles, tetrahedrons, load balancing, redistribut.	-	-	++

Outline

- Parallelization of explicit or implicit solver
- Parallel hardware
- Parallel programming models
- Parallelization scheme

Parallelizing an Application



Parallelizing an Application with MPI

- Designing the domain decomposition
 - How to achieve optimal load balancing
 - **And** minimal data transfer between the sub-domains
- Estimating [for a given platform]
 - Idle time due to non-optimal load balancing
 - Communication time
 - Calculating the estimated speedup
- Implementation
 - Domain decomposition with load balancing
 - Halo storage
 - Communication: Calculated data → halo cells of the neighbors
[e.g., with MPI_Sendrecv (Cartesian grids)
or non-blocking point-to-point communication (unstructured grids)]
 - Checking for global operations, e.g., dot-product, norm, abort criterion
[to be implemented, e.g., with MPI_Allreduce]

Problems

- Scalability
 - Memory:
All large data should be distributed
[and not duplicated on each MPI process]
 - Compute time:
How many processes can be used to have
95%, 90%, 80%, or 50% parallel efficiency?
- Efficient numerical schemes:
 - Multigrid only inside of a MPI process
[and not over the total simulation domain]
 - Full data exchange between all processes
[e.g., a redistribution of the data, (with MPI_Alltoall)]

Summary

- Parallelization of explicit or implicit solver
 - Domain decomposition
 - Halo data communication
 - Global operations
- Parallel hardware
 - Shared memory [SMP] / distributed memory / hybrid [cluster of SMPs]
- Parallel programming models
 - Distributing work and data
 - Additional overhead due to:
 - Communication / Synchronization / Non-optimal load balancing
 - OpenMP / HPF / MPI
- Parallelization scheme
 - Design / Estimation of Speedup / Implementation
 - Scalability problems

Data Parallelism - HPF, I.

Real :: A(n,m), B(n,m)	→	Data definition
!HPF\$ DISTRIBUTE A(block,block), B(...)		
do j = 2, m-1	→	Loop over y-dimension
do i = 2, n-1	→	Vectorizable loop over x-dimension
B(i,j) = ... A(i,j)	→	Calculate B, using upper and lower, left and right value of A
... A(i-1,j) ... A(i+1,j)	→	
... A(i,j-1) ... A(i,j+1)	→	
end do		
end do		

■

Data Parallelism - HPF, II.

- HPF (High Performance Fortran)
 - standardized data distribution model
 - the user has to specify the data distribution explicitly
 - Fortran with language extensions and directives
 - compiler generates message passing or shared memory parallel code
 - work distribution & communication is implicit
 - set-compute-rule:
the owner of the left-hand-side object computes the right-hand-side
 - typically arrays and vectors are distributed
 - draft HPF-1 in 1993, standardized since 1996 (HPF-2)
 - JaHPF since 1999