Parallelization of Explicit and Implicit Solver

Rolf Rabenseifner
rabenseifner@hlrs.de

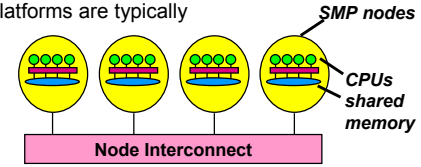
University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

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 …



Parallelization of Explicit and Implicit Solvers [CFD-20]

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 ↔ continuous range *x*,y (2-dim. example)
  upper index t ↔ continuous range *t*
- $\partial T/\partial t = (T_{ij}^{t+1} - T_{ij}^{t})/dt$, $\partial^2 T/\partial x^2 = (T_{i+1,j} - 2T_{i,j} + T_{i-1,j})/dx^2$, …
- $(T_{ij}^{t+1} - T_{ij}^{t})/dt = \alpha((T_{i+1,j}^{?} - 2T_{i,j}^{?} + T_{i-1,j}^{?})/dx^2 + (T_{i,j+1}^{?} - 2T_{i,j}^{?} + T_{i,j-1}^{?})/dy^2)$

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
```

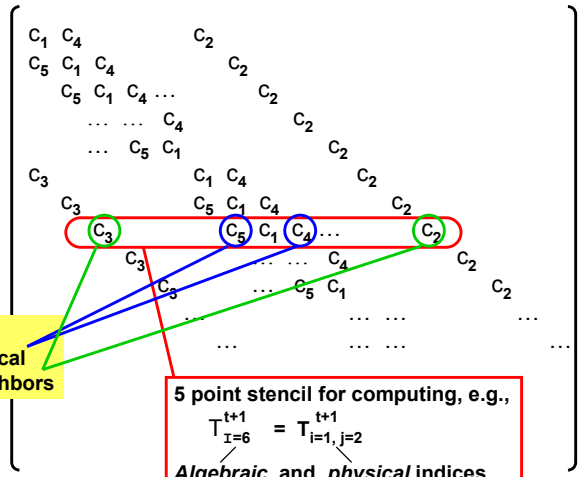
- Vectorizable loop, without indirect addressing !

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; … 0,n-1; 1,0; 1,1; … 1,n-1; … m-1,0; … m-1,n-1
    ➔ **I** = 0; 1; … n-1; n; n+1; … 2n-1; … (m-1)n; … mn-1
  - $(T_{ij})_{i=0..m-1,\ j=0..n-1}$ is view as a vector $(T_I)_{I=0..mn-1}$
  - $T^{t+1} = (I+A)T^t$
- 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 $T^{t+1} = AT^t$

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



Representing physical relation to vertical and to horizontal neighbors

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

Implicit time-step: Solving a PDE

- The right side depends also on **new** values T^t,
  i.e., ? = 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 solve 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 **I**=0..(nm-1) and matrix notation $(I-A)T^{t+1} = T^t$
- $c_1, c_2, \ldots$ normally depend also on i,j (and possibly also on t)

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: $T^{t+1} = (I+A)T^t$
  - Implicit: $(I-A)T^{t+1} = T^t$

  [the 2- or 3-dim T is here expressed as a vector 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: $T^{t+1} = (I+A)T^t$ **[the 2- or 3-dim T is here expressed as a vector]**
  - Implicit: $(I-A)T^{t+1} = T^t$
- Implicit iterative solver:
  - Major (time-consuming) operation is sparse matrix-vector-multiply
    - **Ap with p is an interims 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 Ap 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

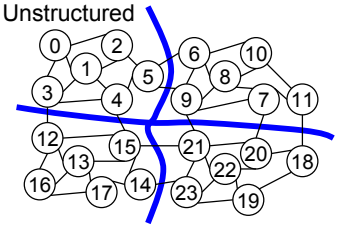
Domain Decomposition

- 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

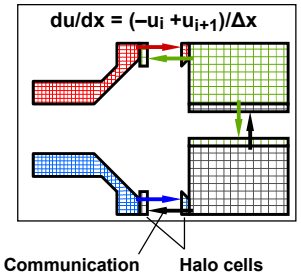


Examples with 4 sub-domains

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

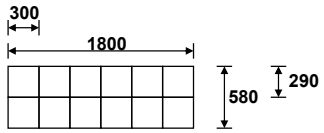


Cartesian Grids

If each grid point requires same work:

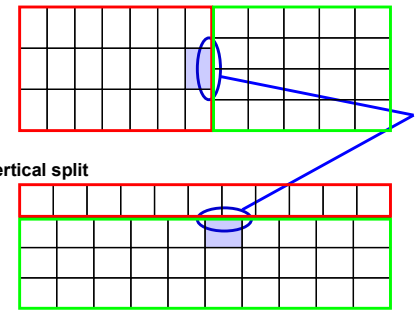
- 2 dimensions:
  each sub-domain (computed by one CPU) should
  - have the same size → optimal load balance
  - and should be quadratic → minimal communication

- 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 x 3,**
    - **but one must make the number of grid points quadratic!**
    - **Example – Task: Grid with 1800 x 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**
    - **Vertical split**

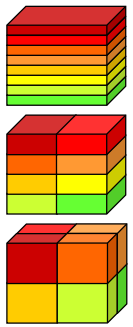


Sub-domains at the split boundary have a more complicated neighborhood

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

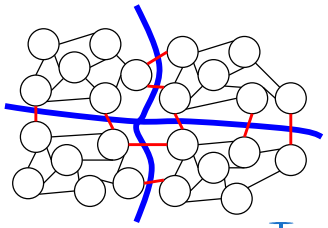
- **one** dimension:
  communication
  $= n^2 * 2 * w * 1$
- **two** dimensions:
  communication
  $= n^2 * 2 * w * 2 / p^{1/2}$
- **three** dimensions:
  communication
  $= n^2 * 2 * w * 3 / p^{2/3}$

optimal for p>11

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

Unstructured Grids

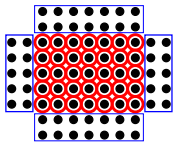
- Mesh partitioning with special load balancing libraries
  - Metis (George Karypis, University of Minnesota)
  - ParMetis (internally parallel version of Metis)
    - **http://www.cs.umn.edu/~karypis/metis/metis.html**
    - **http://www.hlrs.de/organization/par/services/tools/loadbalancer/metis.html**
  - Jostle (Chris Walshaw, University of Greenwich)
    - **http://www.gre.ac.uk/jostle**
    - **http://www.hlrs.de/organization/par/services/tools/loadbalancer/jostle.html**
  - 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

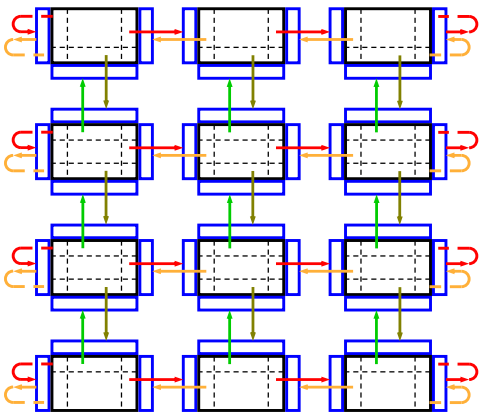


Communication: Send inner data ⇄ into halo storage

One iteration in the

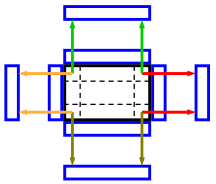
- **serial** code:
  - X_{new} = function(x_{old})
  - X_{old} = x_{new}
- **parallel** code:
  - Update halo
    [=Communication, e.g., with **4 x MPI_Sendrecv** ⇄ ⇅]
  - X_{new} = function(x_{old})
  - X_{old} = x_{new}

Examples with 12 sub-domains and horizontally cyclic boundary conditions



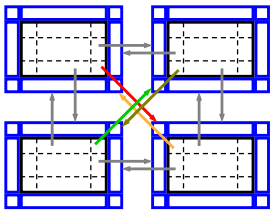
Corner problems

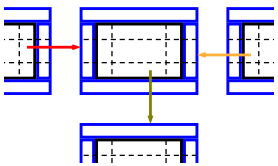
- MPI non-blocking send must not send inner corner data into more than one direction
  - Use MPI_Sendrecv
  - Or non-blocking MPI_Irecv



- 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

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

T_{serial}, f	wall-clock time needed with one processor percentage T_{serial} of that can not be parallelized
$T_{parallel,\ p}$	wall-clock time needed with p processor
$T_{communication}$	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 := $T_{serial} / T_{parallel,\ p}$
E_p	efficiency on p processors := S_p / p

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

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

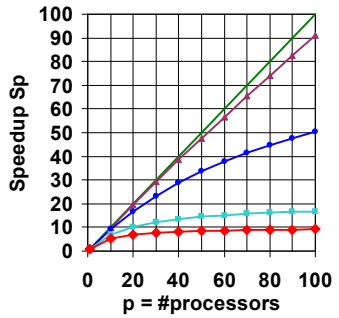
$$\approx 1 - \underbrace{f(p-1)}_{\text{should be } \ll 1} - \underbrace{T_{communication} / (T_{serial}/p)}_{\ll 1} - \underbrace{T_{idleCPU} / T_{serial}}_{\ll 1}$$

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

$T_{parallel,\ p} = f \cdot T_{serial} + (1-f) \cdot T_{serial} / p$
f ... sequential part of code that can not be done in parallel

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

For p —> infinity, speedup is limited by $S_p < 1 / f$



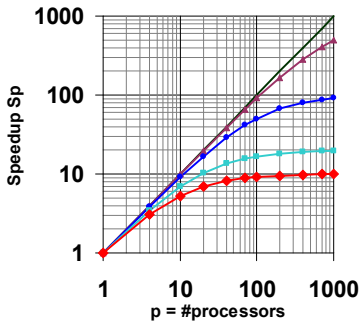
- Sp = p (ideal speedup)
- f=0.1% => Sp < 1000
- f= 1% => Sp < 100
- f= 5% => Sp < 20
- f= 10% => Sp < 10

Amdahl's Law (double-logarithmic)

$T_{parallel,\ p} = f \cdot T_{serial} + (1-f) \cdot T_{serial} / p$
f ... sequential part of code that can not be done in parallel

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

For p —> infinity, speedup is limited by $S_p < 1 / f$



- Sp = p (ideal speedup)
- f=0.1% => Sp < 1000
- f= 1% => Sp < 100
- f= 5% => Sp < 20
- f= 10% => Sp < 10

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:**
  - Sc(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**

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
  - $T_{communication}$ = (9-1) · 300 · 16 byte / 20 MB/s = 1.92 ms
  - T_{serial} / p = 300 · 300 · 100 FLOP / (1 GFLOP/s · 10%) = 90 ms
  - → $T_{communication}$ / (T_{serial}/p) = 1.92 ms / 90 ms = 0.021 << 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
  - $T_{communication}$ = (13-1) · 50 · 50 · 16 byte / 20 MB/s = 24 ms
  - T_{serial} / p = 50 · 50 · 50 · 100 FLOP / (1 GFLOP/s · 10%) = 125 ms
  - → $T_{communication}$ / (T_{serial}/p) = 24 ms / 125 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) → `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: Ax
  - Scalar product: (v_1,v_2)

Parallelization of Explicit and Implicit Solvers [CFD-20]

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;
$\alpha = (|| r ||_2)^2$;
for (i=0; (i < i_max) && ($\alpha > \varepsilon$); i++)
{ v = **Ap**;
$\lambda = \alpha$ / **$(v,p)_2$** ;
x = x + λp;
r = r – λv;
$\alpha_{new} = ($ **$|| r ||_2$** $)^2$;
p = r + (α_{new}/α)p;
$\alpha = \alpha_{new}$;
}
Print x, $\sqrt{\alpha}$, $||b-Ax||_2$;

See, e.g.,
Andreas Meister: Numerik linearer Gleichungssysteme.
Vieweg, 1999, 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++)
            v_{i,j,k} = 0
            sparse (unrolled) loops over i', j', k'
                v_{i,j,k} = v_{i,j,k} + a_{i,j,k; i',j',k'} * p_{i',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 (≤ size of A);  Initialize ε (>0);
Initialize solution vector x;
p = x;
v = Ap;
p = b – v;
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;
      α_new = (|| r ||_2)² ;
      p  = r + (α_new/α)p;
      α = α_new;
}
Print x, √α, ||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 (≤ size of A);  Initialize ε (>0);
Initialize solution vector x;
p = x;
v = Ap;
p = b – v;
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;
      α_new = (|| r ||_2)² ;
      p  = r + (α_new/α)p;
      α = α_new;
}
Print x, √α, ||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

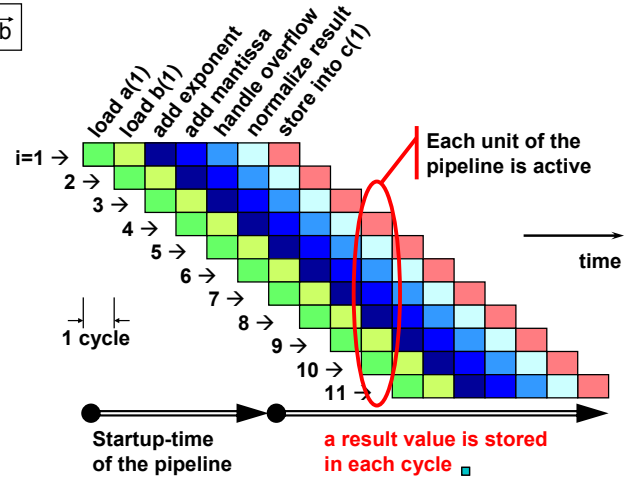
- Non-cubic may cause better computational efficiency
  - 50x50x50 cubic → boundary = 6 x 50 x 50 = 15,000
  - vs. 100x25x50 → boundary = 2 x 100 x 25
+ 2 x 100 x 50
+ 2 x 25 x 50 = 17,500
  - 16 % larger boundary, and
  - 100% longer most inner loop,
that may cause more than 16 % computational speedup

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 <u>needs halo data</u>**
➔ <u>cannot</u> be overlapped with communication
    - **Computation of grid points that do <u>not need halo data</u>**
➔ <u>can</u> 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, 0:n)
  - The most-inner loop may be to 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.

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

Parallelization of single-step algorithms

Single step algorithms

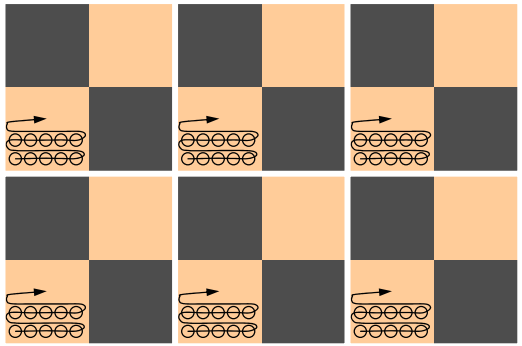
- Example: SOR
  - $x_{m+1,i} := (1-\omega)x_{m,i} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_{m+1,j} - \sum_{j=i}^{n} a_{ij}\, x_{m,j}\right)$ (m = #iteration)
  - if only direct neighbor exists, i.e. $a_{ij} \neq 0$ for j = "i+x", "i–x", "i+y", "i–y"
  - and "i–x" and "i–y" are indexes less than i, then

→ $x_{m+1,i}$

$:= (1-\omega)x_{m,i} + \frac{\omega}{a_{ii}}\left(b_i - a_{i,i-x}x_{m+1,i-x} - a_{i,i-y}x_{m+1,i-y} - a_{i,i+x}x_{m,i+x} - a_{i,i+y}x_{m,i+y}\right)$

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.

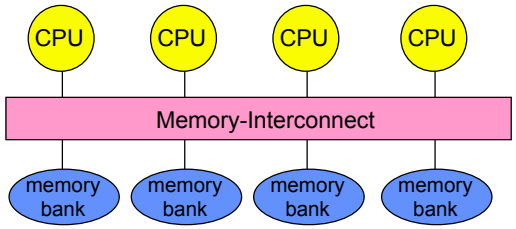
Outline

- 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

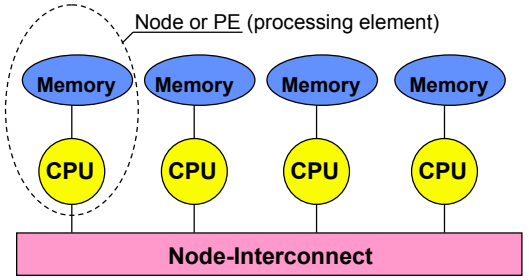
Multiprocessor - shared memory



- 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

Thanks to Alfred Geiger and Michael Resch (HLRS):
Some pictures and slides are from their *Parallel Programming* lectures.

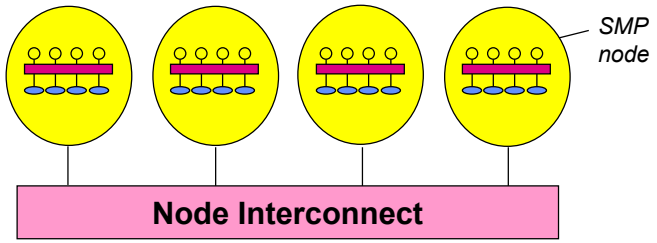
Multicomputer - distributed memory



- 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

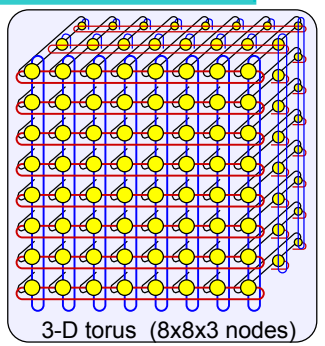
Interconnects

- Memory interconnect
  - bus
  - cross-bar
- Node interconnect
  - bus based networks
    -
  - multi-link networks, **e.g.,**
    - **ring with independent connections**
  - 2-D or 3-D torus
    - **each processor is connected by a link with 4 or 6 neighbors**
  - hierarchical networks
    - **multi-level cross-bars**
  - cross-bar (single level)
  - full interconnect

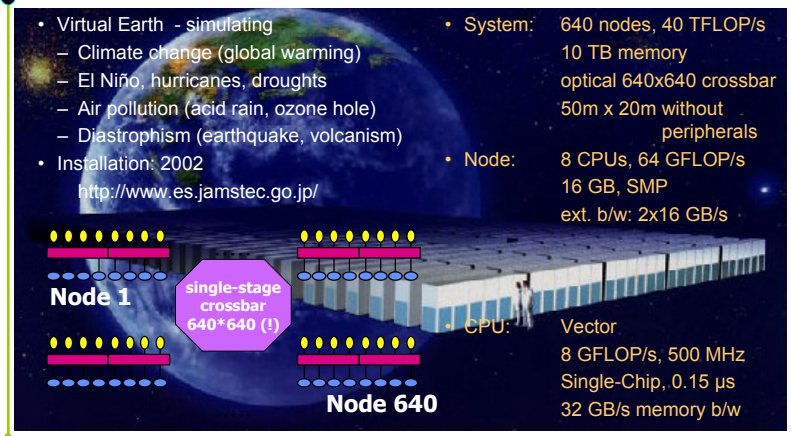
cheap, but poor inter-connect

scalable network costs, high accumulated bandwidth

not scalable! — n*(n-1)/2 links


3-D torus (8x8x3 nodes)

Earth Simulator Project ESRDC / GS 40 (NEC)



Outline

- Parallelization of explicit or implicit solver
- Parallel hardware
- Parallel programming models
  - **Parallelization Strategies → Models [56] → OpenMP [56-58] → HPF [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

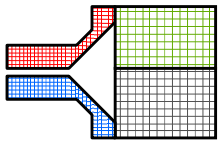
Work decomposition
- based on loop decomposition

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

```
Do i=1,100                    i=1..25 | 26..50 | 51..75 | 76..100
    a(i) = b(i)+c(i)              execute on the 4 processors
Enddo
                              BARRIER synchronization
Do i=1,100
    d(i) = 2*a(101-i)         i=1..25 | 26..50 | 51..75 | 76..100
Enddo                             execute on the 4 processors
```

- **Synchronization**
  - is necessary
  - may cause
    - **idle time on some processors**
    - **overhead to execute the synchronization primitive**

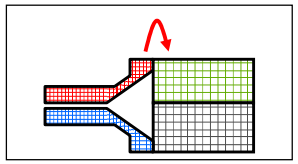
Communication

```
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

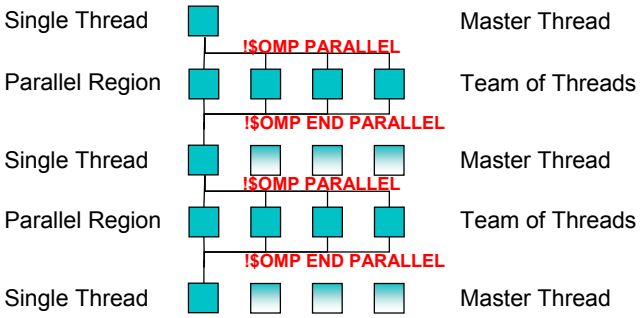
1. **OpenMP**
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
- **HPF (High Performance Fortran)**
  - Data Parallelism
  - User specifies data decomposition with **directives**
  - Communication (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

!$OMP PARALLEL DO
do j = 2, m-1                                   ⇨ Loop over y-dimension
  do i = 2, n-1                                 ⇨ Vectorizable loop over x-dimension
    B(i,j) =  ... A(i,j)                        ⇨ Calculate B,
            ... A(i-1,j) ... A(i+1,j)                 using upper and lower,
            ... A(i,j-1) ... A(i,j+1)                       left and right value of A
  end do
end do
!$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 – HPF

1. **OpenMP**
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
2. **HPF (High Performance Fortran)**
   - Data Parallelism
   - User specifies data decomposition with **directives**
   - Communication (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**

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,
             ... A(i-1,j) ... A(i+1,j)    ⇨   using upper and lower,
             ... A(i,j-1) ... A(i,j+1)    ⇨     left and right value of A
  end do
end do
```

Data Parallelism – HPF, II.

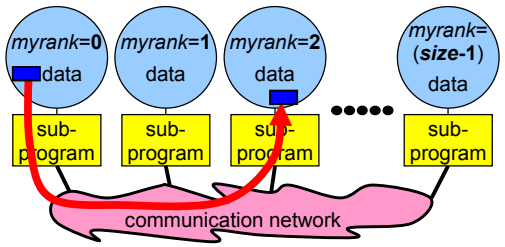
- HPF (High Performance Fortran)
  - standardized data distribution model
  - the user has to specify the data distribution explicitly
  - Fortran with language extensions and directives
  - compiler generates message passing or shared memory parallel code
  - work distribution & communication is implicit
  - 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

Major Programming Models – MPI

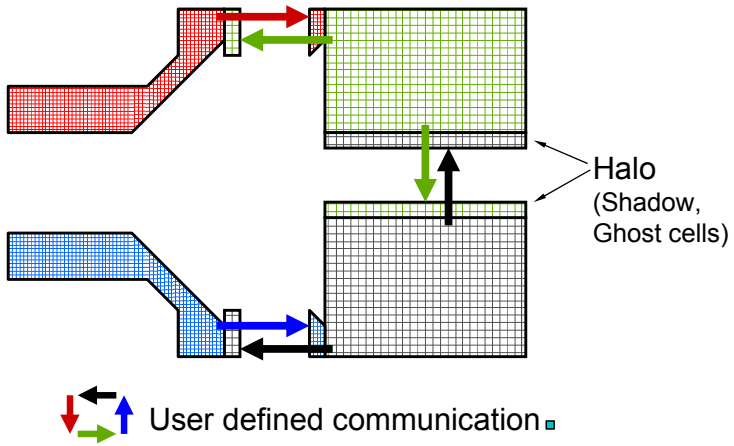
1. **OpenMP**
   - Shared Memory **Directives**
   - to define the work decomposition
   - no data decomposition
   - synchronization is implicit (can be also user-defined)
2. **HPF (High Performance Fortran)**
   - Data Parallelism
   - User specifies data decomposition with **directives**
   - Communication (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)        ⇨ Data definition
do j = max(2,ja), min(m-1,je)               ⇨ Loop over y-dimension
 do i = 2, n-1                              ⇨ Vectorizable loop over x-dimension
  B(i,j) = ... A(i,j)                       ⇨ Calculate B,
           ... A(i-1,j) ... A(i+1,j)        ⇨   using upper and lower,
           ... A(i,j-1) ... A(i,j+1)        ⇨         left and right value of A
 end do
end do

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), 1.1 (1995), 1.2 (1997)
    MPI-2: since 1997

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

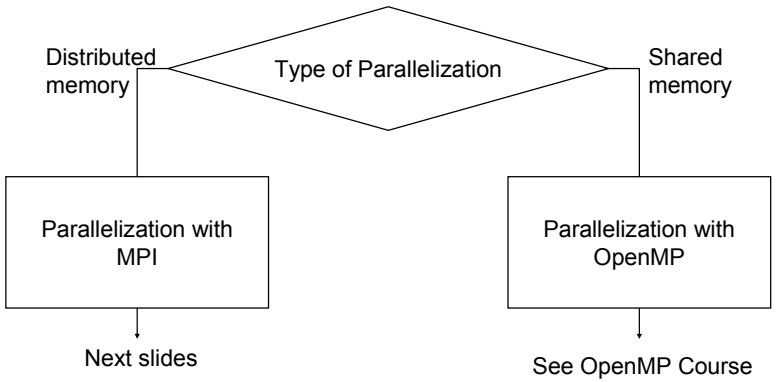
Advantages and Challenges

	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, tetra-hedrons, load balancing, redistribut.	–	–	++

Outline

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

Parallelizing an Application

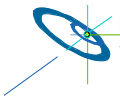


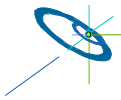
Parallelizing an Application with MPI

- 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 an 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