LOAD MIGRATION

Klaus Birken

Institute for Computer Applications III, University of Stuttgart,
Pfaffenwaldring 27, D-70569 Stuttgart, Germany
email: birken@ica3.uni-stuttgart.de

Outline

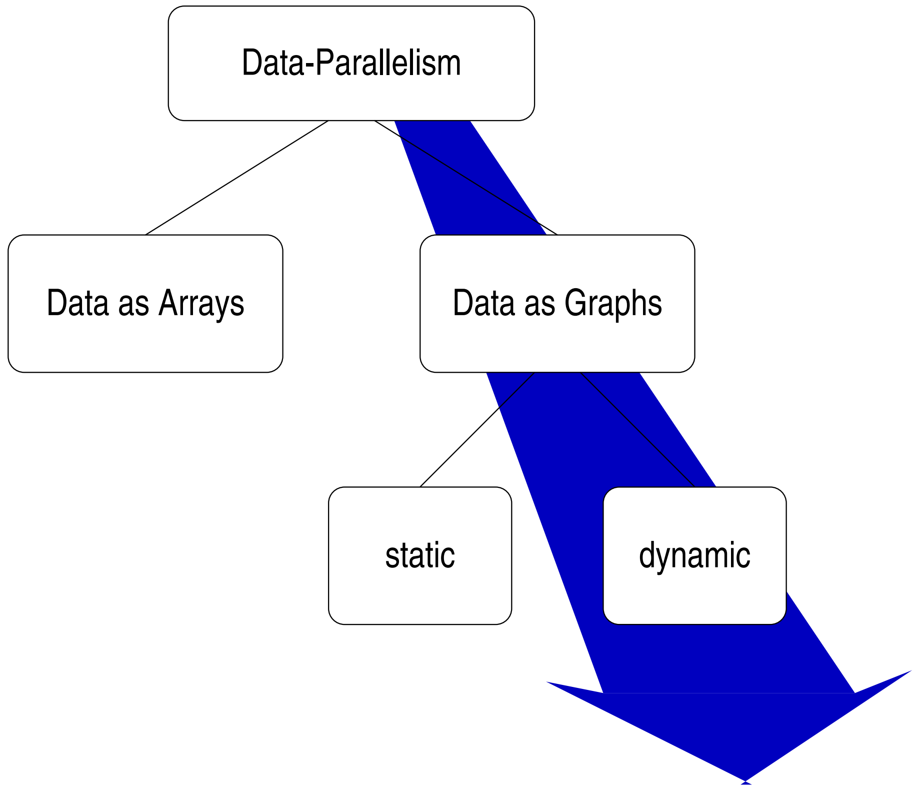
- ⋆ The (Existing) Applications
- ⋆ The (Abstract) Machines
- ⋆ DDD – Objectives & Programming Model
- ⋆ DDD – An Application Example
- ⋆ DDD – A Real-World Application
- ⋆ The Future: Automization

Restriction: This is *not* a talk on OS features.

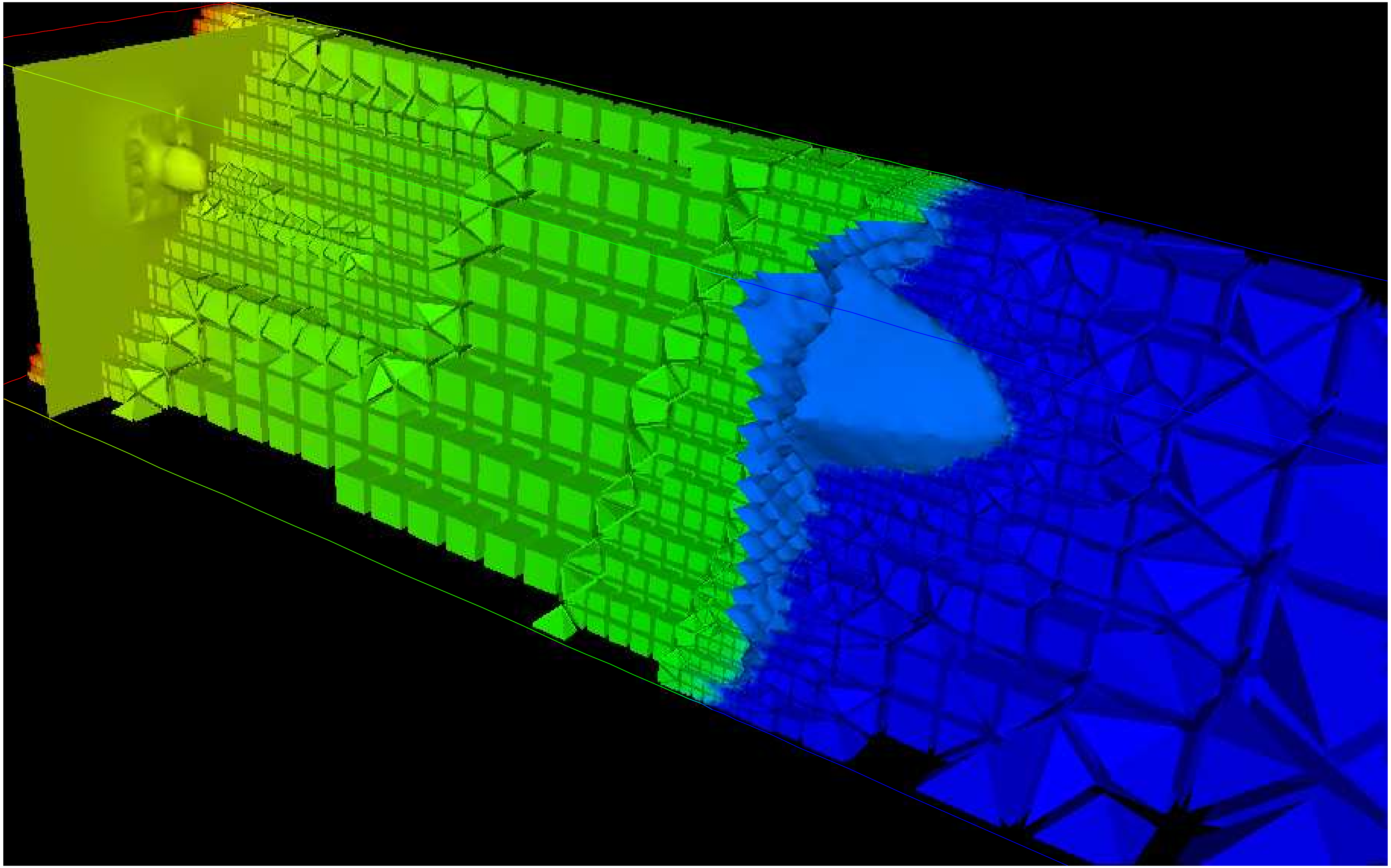
Process-level migration is not an issue,
at least for the next two hours.

1. Parallelization of dynamic data programs is complicated.
2. A high-level programming model can help.
3. Careful software reuse is necessary.

The (Existing) Applications

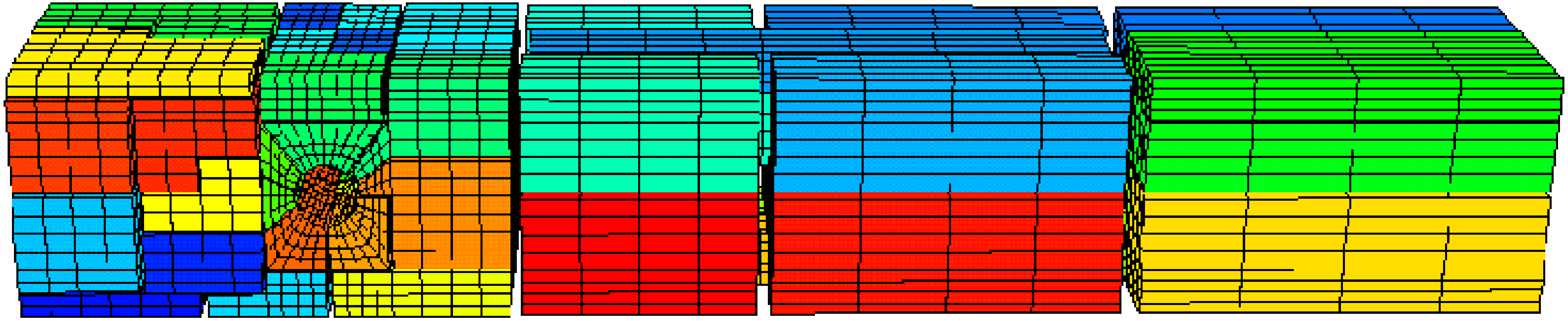


3D Example: Multi-Phase Flow



P. Bastian, ”Viscous Fingering” (Buckley–Leverett Problem, 3D)

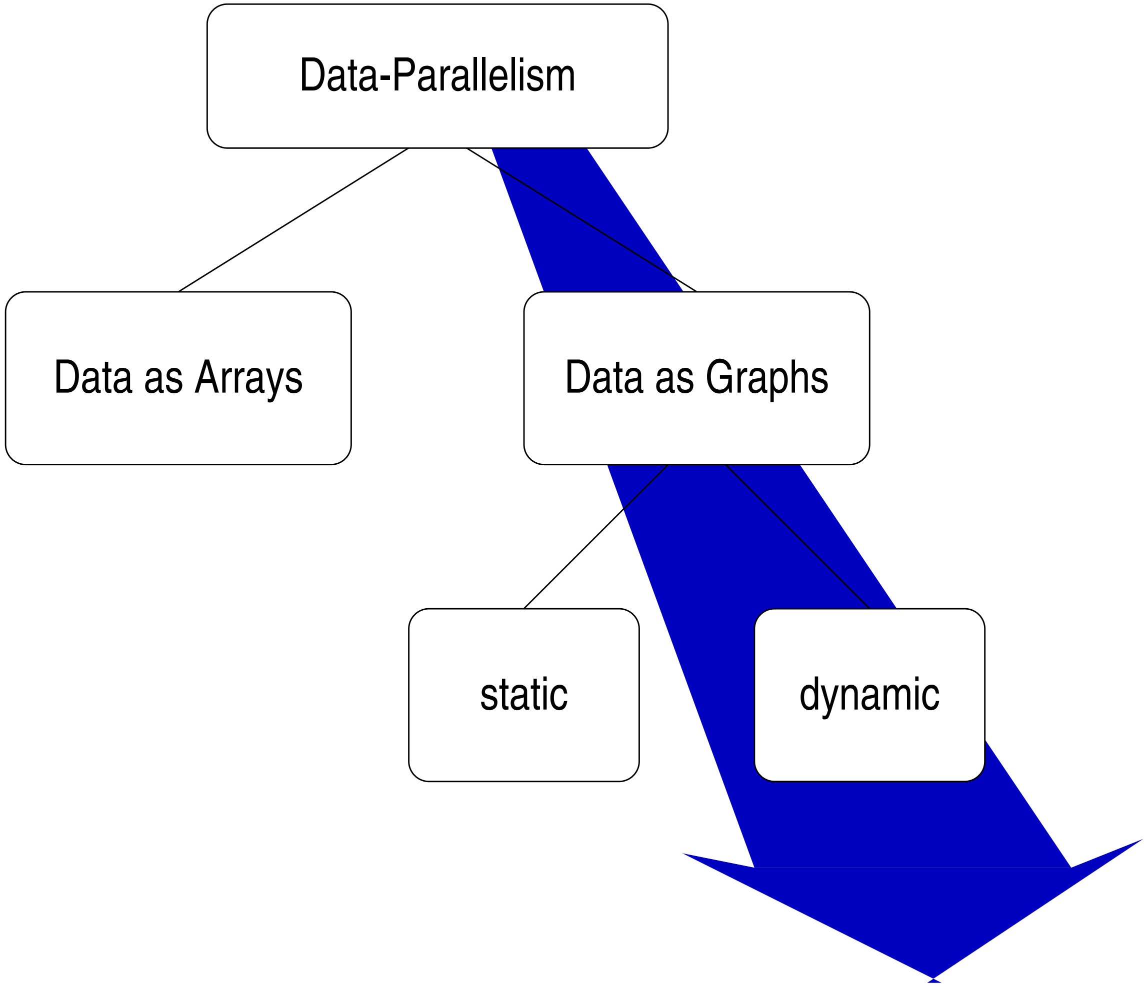
Example: Navier-Stokes Solver



V. Reichenberger, Load Balancing on 54 procs

* ⋆ for computational intensive problems on distributed-memory: partitioning of data, control flow in SPMD
* ⋆ arrays vs. graphs
  graphs are much harder, due to:
  * no global information
  * no references across address spaces
* ⋆ static vs. dynamic
  * dynamic load balancing
  * dynamic load migration

An Application Classification Scheme



A typical application

- ⋆ computationally intensive
  e.g. PDE solver
- ⋆ unstructured meshes
  e.g. tetrahedra and hexahedra in 3D
- ⋆ state-of-the-art numerics
  e.g., multi-level methods
- ⋆ dynamic mesh changes
  e.g., locally adaptive refinement/derefinement

Objectives

⋆ parallelization of existing sequential programs

- minor effort
- don't change program code!
- continuous development/maintenance process

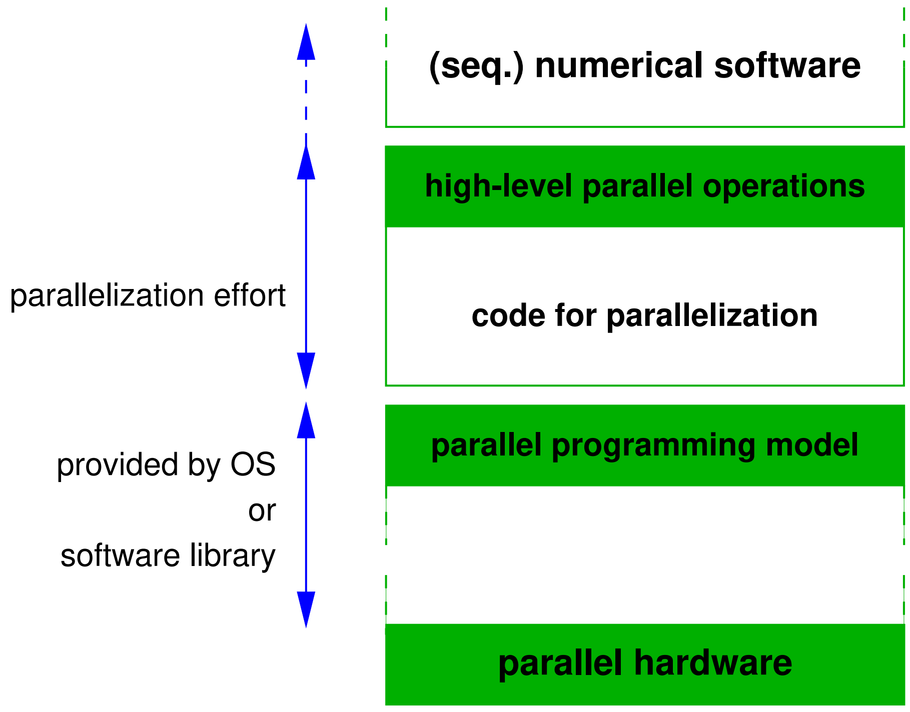
⋆ development of new parallel programs

- useful abstract programming model
- problem-oriented, not machine-oriented

⋆ portability and efficiency

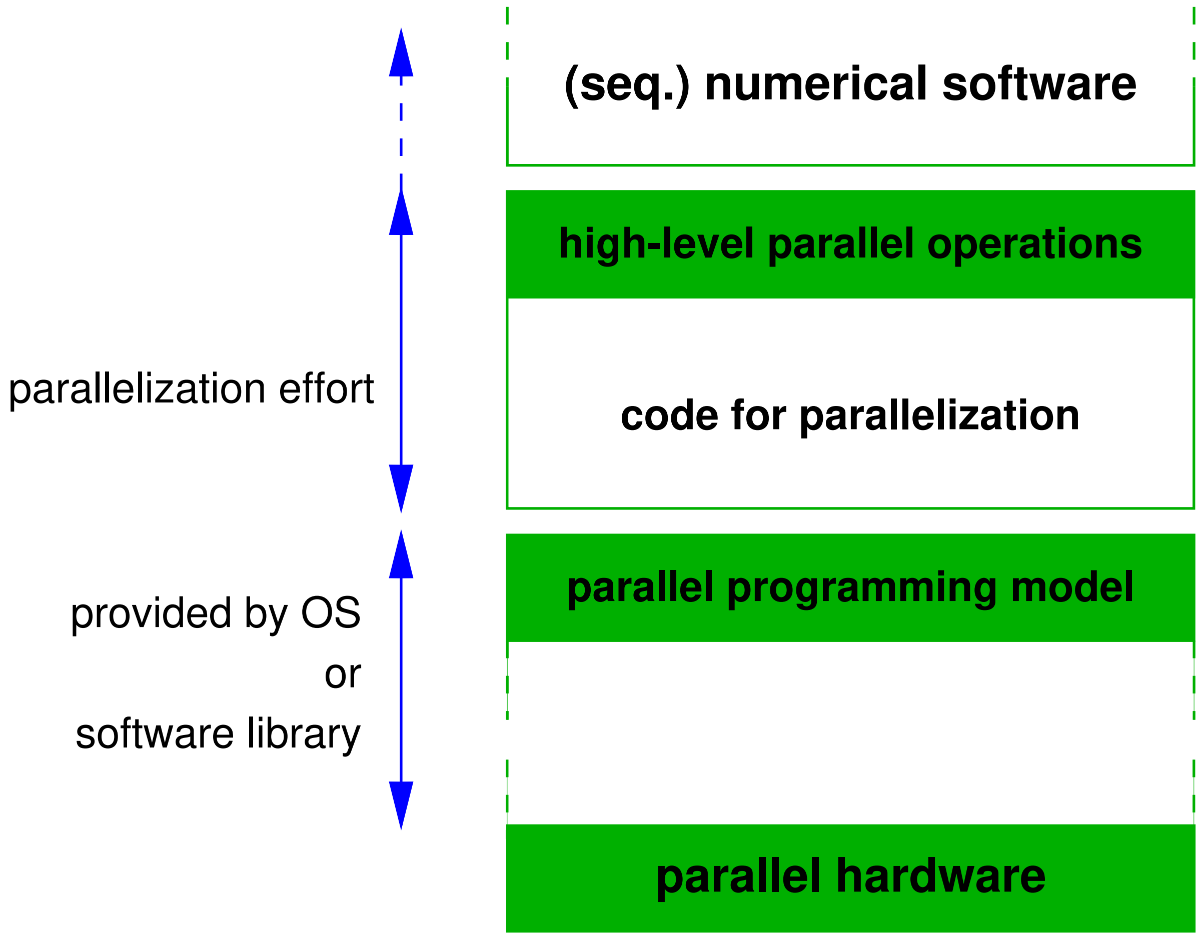
- porting with minimal effort
- efficiency without manual tuning
- explicit information about data-locality

The (Abstract) Machines



Complicated data structures
and
Parallelization effort
$\Downarrow$
High software complexity

A layer-based strategy will be necessary for implementing data-dynamic parallel programs.


(seq.) numerical software
high-level parallel operations
parallelization effort
code for parallelization
parallel programming model
provided by OS
or
software library
parallel hardware

$\Leftarrow$ Message-Passing

⋆ many address spaces
⋆ management of object copies
⋆ local object references
⋆ user is responsible for consistency
⋆ explicit knowledge about locality et al.

$\Rightarrow$ Shared-Memory

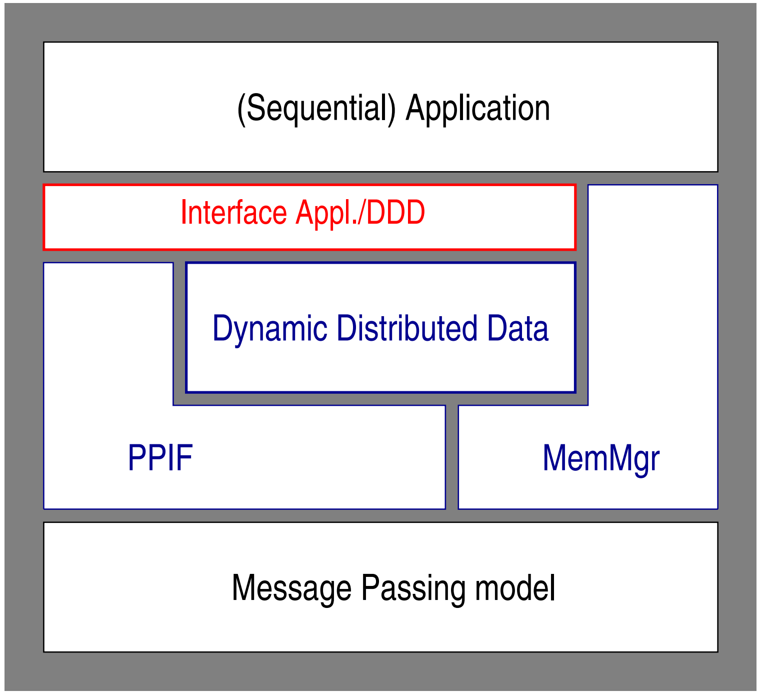
⋆ one global address space
⋆ no object copies, global references
⋆ system is responsible for consistency
⋆ implicit knowledge $\rightarrow$ potential bottleneck

Goal: Find trade-off!

Tools for parallel distributed graphs

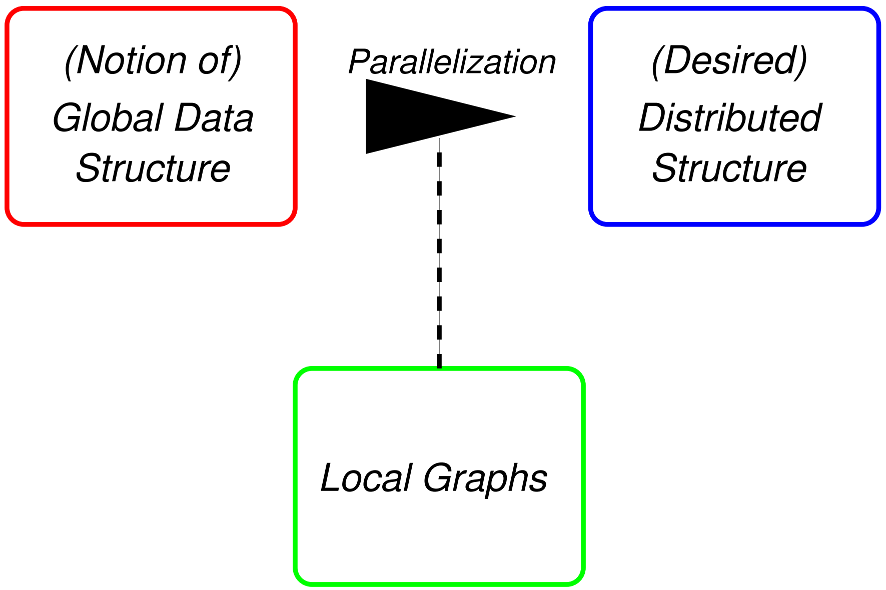
A++P++	blocks	dynamic
BlockComm	blocks	static
CHAOS	unstructured	dynamic
CHAOS++	general	dynamic
CLIC	blocks	dynamic
Canopy	blocks	static
Concurrent Graph	general	dynamic
DAGH	blocks	dynamic
DDD	general	dynamic
DIME	triangles	dynamic
Dome	grids	dynamic
GA	grids	dynamic
GRIDS	unstructured	static
LOCO	blocks	dynamic
LPARX	blocks	dynamic
NNT	grids	static
OPlus	unstructured	static
PARTI	unstructured	static
PetSc	grids	static
RSL	grids	dynamic
SUMAA3d	triangles	dynamic

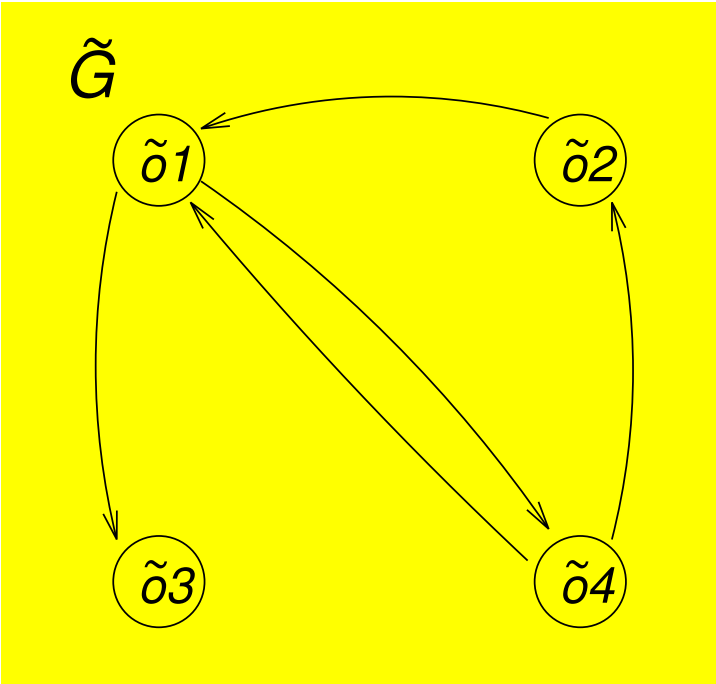
DDD – Objectives & Programming Model



Which data structures can be treated using DDD?

How can one define *parallelization/distribution*?



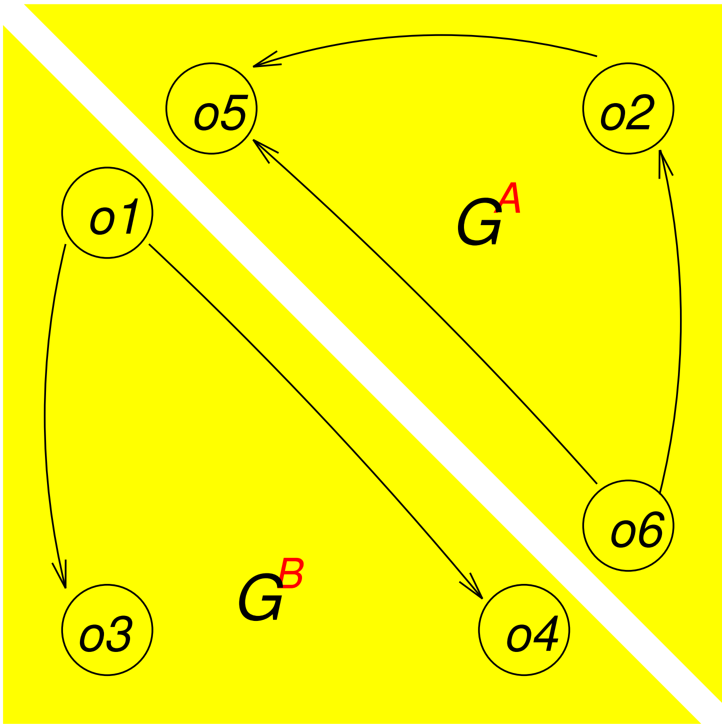


Objects $\tilde{O}$ are structured memory regions:

$$\tilde{o} = (\texttt{type},\texttt{data}).$$

Reference notation $\tilde{o}_i \xrightarrow{\text{ref}} \tilde{o}_j$ defines global relations by

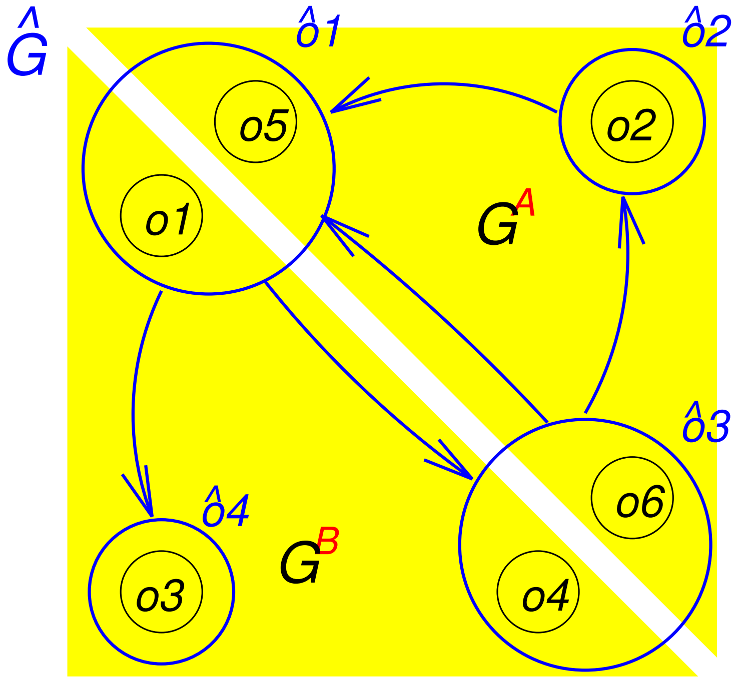
$$(\tilde{o}_i, \tilde{o}_j) \in \tilde{R} \;\Leftrightarrow\; \tilde{o}_i \xrightarrow{\text{ref}} \tilde{o}_j.$$



Object definition enhanced by processor id.

Local graphs G^p defined by *local reference relations* R^p:

$$\forall p \in P : \ (o_i, o_j) \in R^p \ \Leftrightarrow \ o_i \in O^p \wedge o_j \in O^p \wedge \ ptr o_i o_j$$



Mapping *id* : $O \mapsto \mathcal{I}$ assigns arbitrary index to objects.

This induces *sh* : $\mathcal{I} \mapsto \mathcal{P}(O)$ as follows:

$$sh(i \in \mathcal{I}) := \{o \mid id(o) = i\}$$

Thus, local objects are grouped into *distributed objects.*

$$\forall i \in \mathcal{I} : \quad \hat{o}_i := \mathrm{sh}(i).$$

⋆ Priority

- definition of local object is enhanced by

  $$\texttt{prio}(o) \in \Pi \subset \mathbf{N}$$

- this allows definition of consistency-protocols on application level
- e.g., priviledged treatment of certain object types

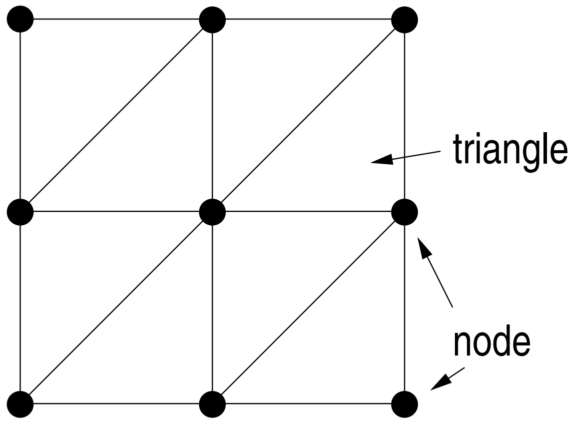
⋆ Attribute

- definition of distributed object is enhanced by

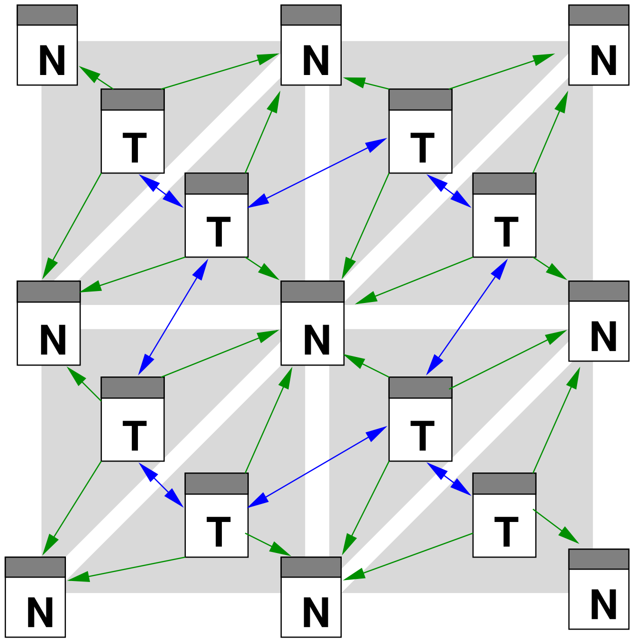
  $$\texttt{attr}(\hat{o}) \in \mathcal{A}$$

- decomposition of object set into several subsets
- e.g., mapping of objects onto multilevel-hierarchy

Mini-Example: Triangular Grid

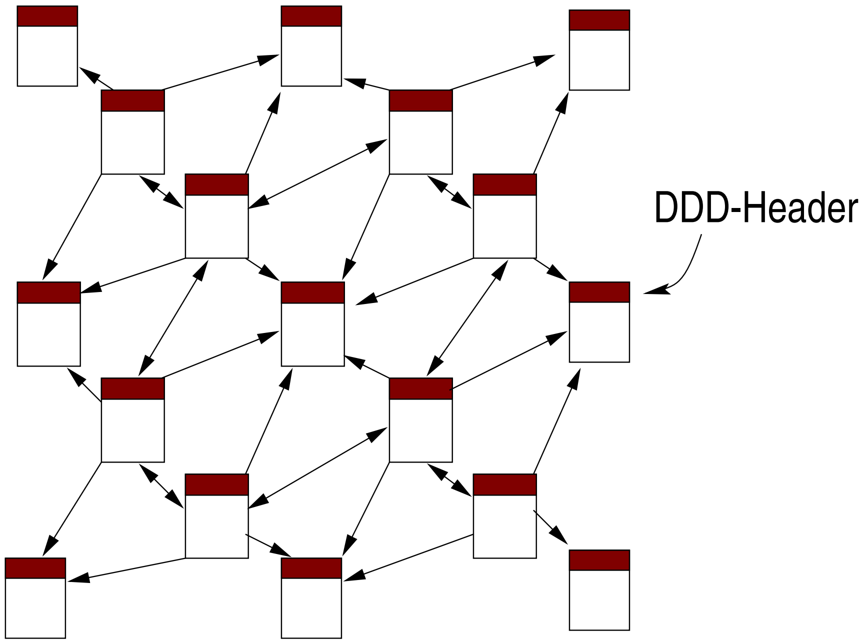


simple grid with triangles and nodes
(e.g., for FE computations)



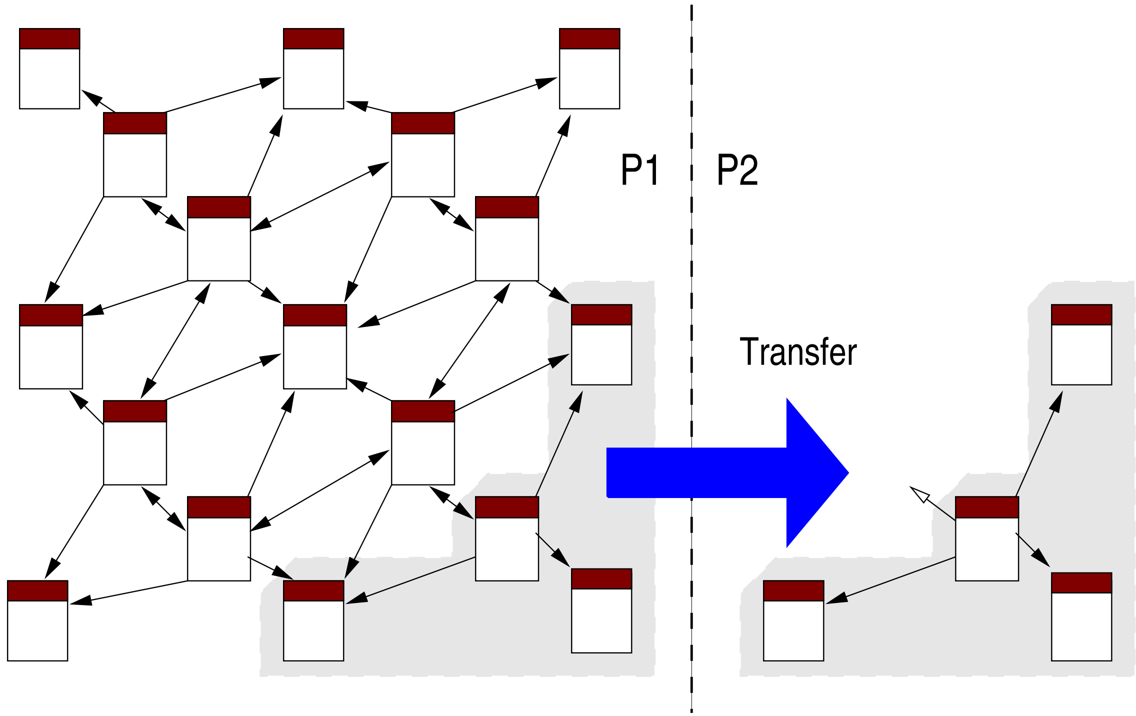
- one datatype each for triangles (T) and nodes (N)
- triangle references three nodes
- triangle references neighbours
- here: no edge-datatype

Mini-Example: Data Structures with DDD



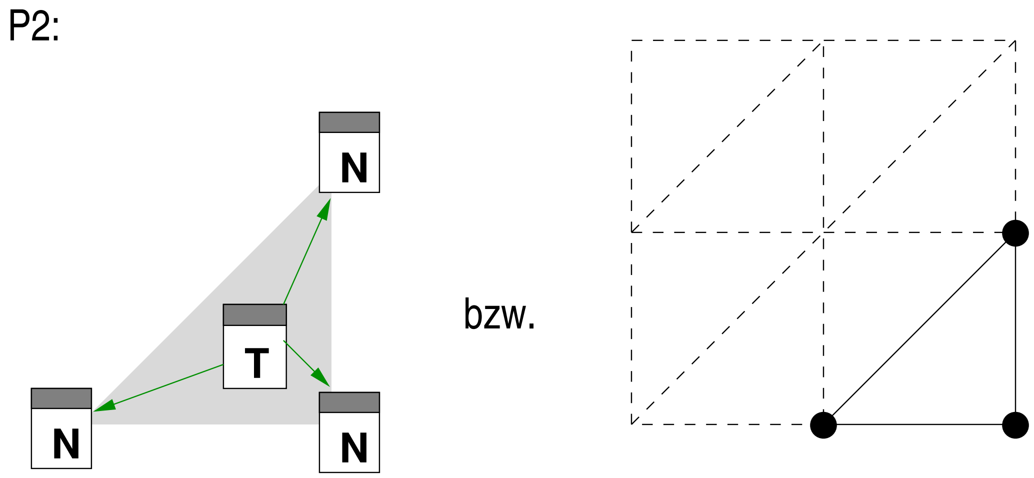
Data type `DDD_HEADER`			
type	name	description	unique in:
`DDD_TYPE`	*type*	DDD-type	distributed object
`DDD_GID`	*gid*	global ID	distributed object
`DDD_ATTR`	*attr*	attribute	distributed object
`DDD_PRIO`	*prio*	priority	local object
int	*index*	local index	local object

Mini-Example: Grid Transfer



- ⋆ transfer of single objects
- ⋆ management of object references
- ⋆ redundancy/overlapping

Mini-Example: Grid-Part after Transfer



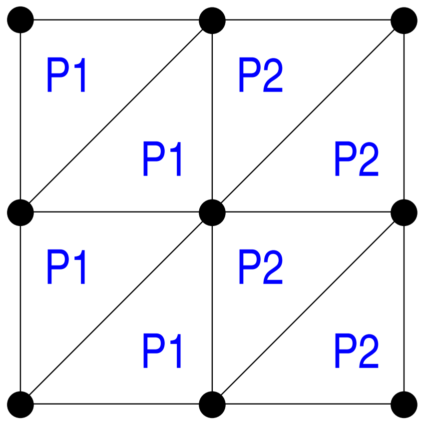
⋆ receiver gets consistent grid

⋆ doesn't know other parts of grid

⋆ no references to remote objects

Mini-Example: Load balancing I

Partitioning on two processors:

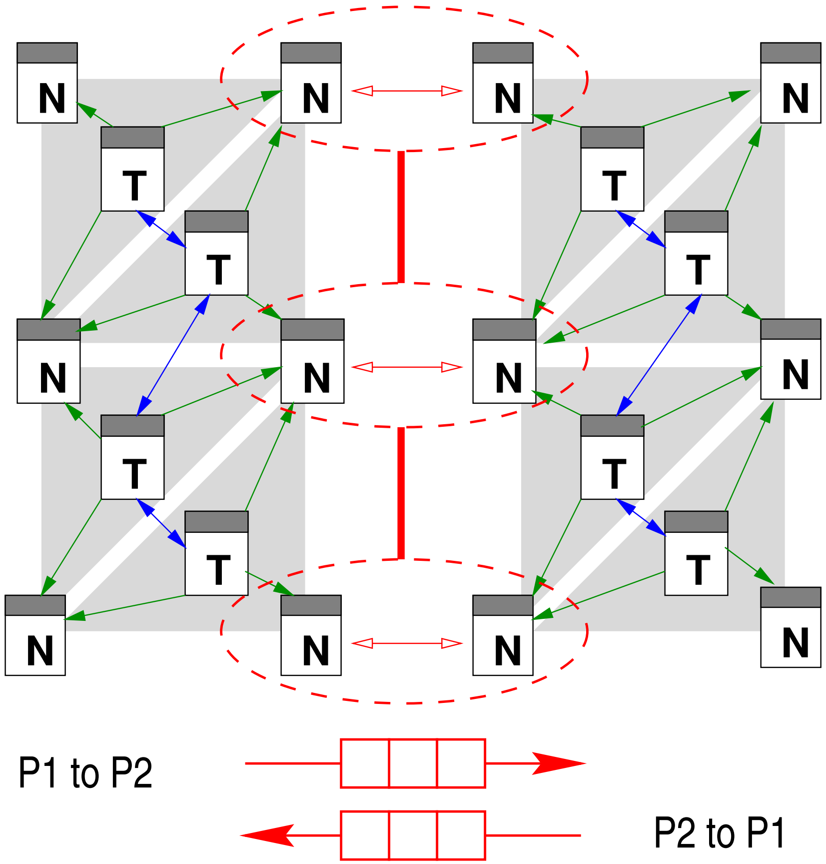


⋆ load balancer marks triangles

⋆ DDD carries out actual load migration

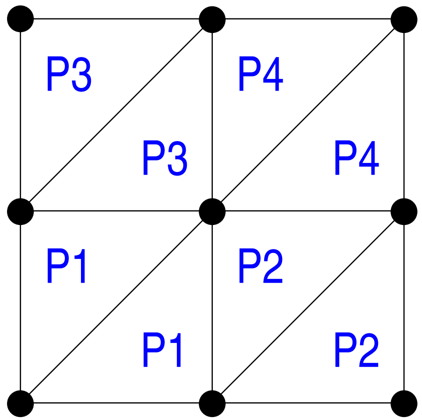
⋆ realization of overlapping strategy

⋆ other data-types depend on triangles



- ⋆ due to overlapping: distributed objects
- ⋆ Interfaces as abstract communication scheme
- ⋆ specified at define time:
  data-types in Interface, priorities
- ⋆ specified at usage time:
  data per object, gather/scatter

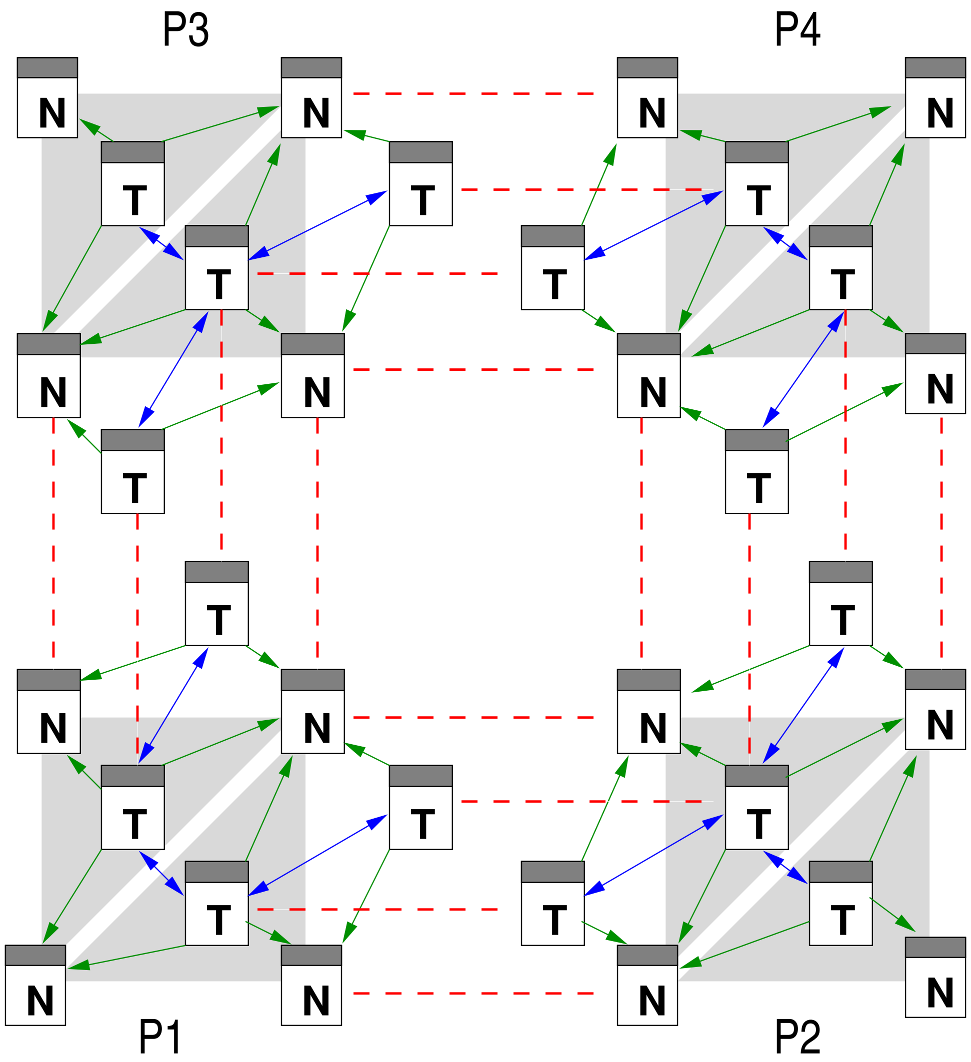
Partitioning on four processors:



Additionally we require:

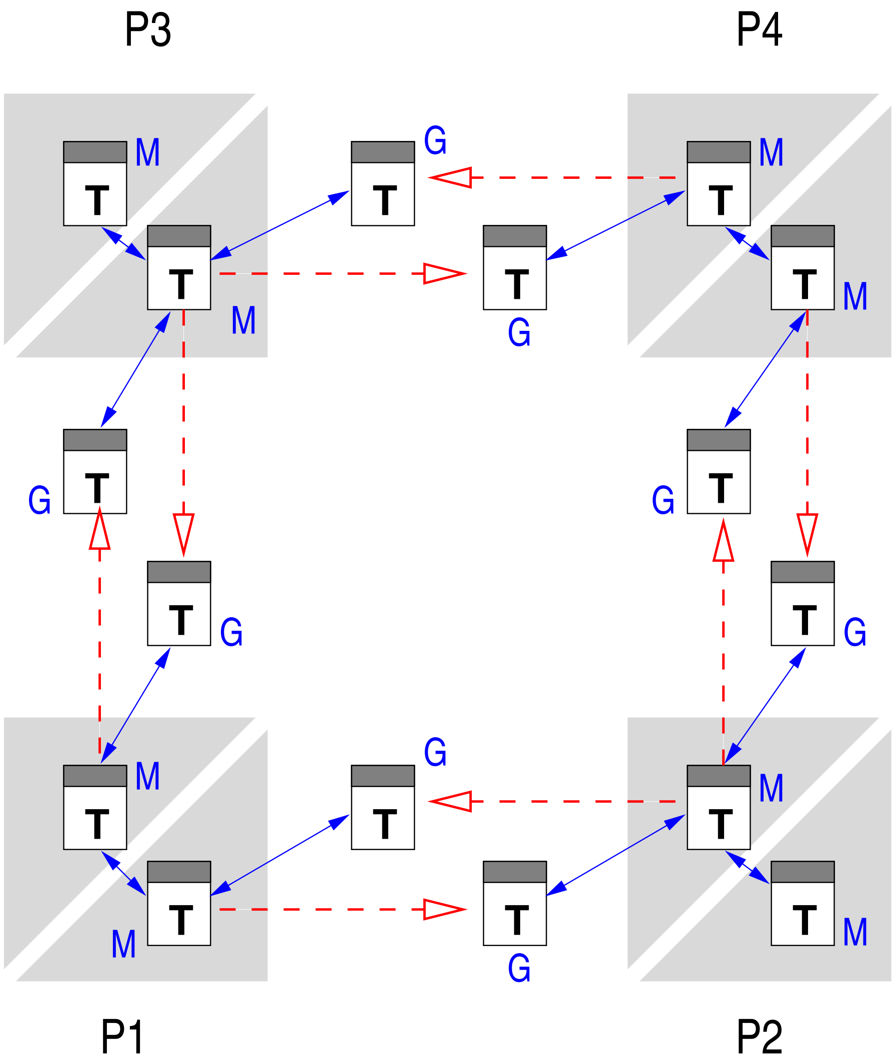
$\forall$triangle t_1, t_2 : $\forall$proc p :

$$nb(t_1, t_2) \wedge master(t_1, p) \rightarrow master(t_2, p) \vee ghost(t_2, p)$$



new:
multiple copies,
different object types,
different priorities,
uncomplete triangle-objects

Mini-Example: Interfaces II



- ⋆ two priorities (M and G)
- ⋆ directed communication

⋆ definition of an Interface:

$$\mathcal{X} = (O_{\mathcal{X}}, A, B)$$

⋆ communication only *inside* each distributed object

$A \to B$: each object copy

$$o \in O_{\mathcal{X}} \cap O^p \quad \text{mit} \quad \texttt{prio}^p(o) \in A$$

sends data to each copy

$$o \in O_{\mathcal{X}} \cap O^q \quad q \neq p \quad \text{with} \quad \texttt{prio}^q(o) \in B.$$

$B \to A$: as before, A and B switched

$A \leftrightarrow B$: exchange in both directions

⋆ transaction-oriented operations

⋆ example:

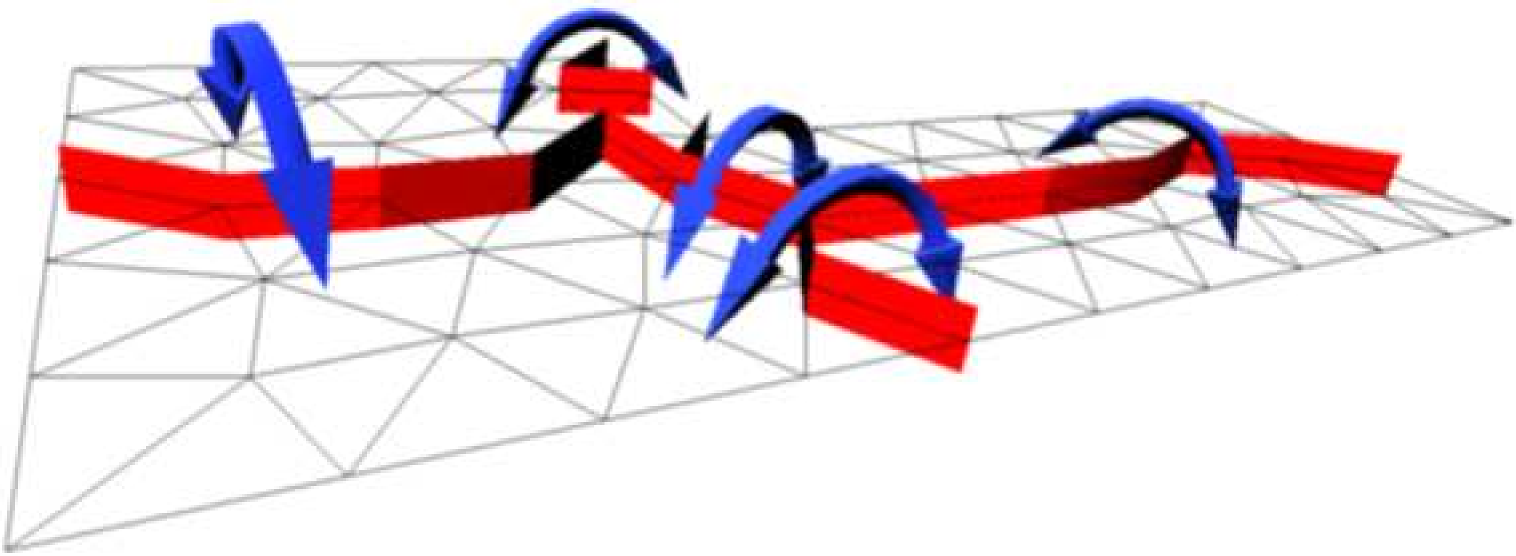
```
XferBegin();
    ...
    XferCopyObj(object, proc, prio);
    ...
    XferDeleteObj(object);
    ...
    XferSetPriority(object, prio);
    ...
XferEnd();
```

⋆ minimal number of messages

⋆ interfaces are kept consistent

- ⋆ F77 Frontend
  mapping of DDD-functionality to F77-specials with same DDD source-code.
- ⋆ C++ Frontend
  design of a class-hierarchy for integration of DDD and OO-software.
- ⋆ debugging-support
  application-dependend debugger, control of parameters and consistency checking, all at run-time.
- ⋆ performance-analyser
  complex benchmarking-application for measuring performance of hardware, PPIF and DDD.
- ⋆ consistency-checking
  automated testcase-generation (in Prolog), formal specification for atomic DDD Transfer-commands.

DDD – An Application Example



* Designing control flow
* Choosing objects
* Defining distribution
* Implementing handlers
* Load balancing
* Synchronization calls
* Additional requirements

Step 1: Designing Control Flow

```
FiniteElementMain()
{
   Mesh   *m;

   DDD_Init();
   if (me==master)
      CreateMeshStructure(&m);
   DistributeMesh(m);

   ComputeLocalMatrices(m);
   ComputeGlobalMatrix(m);
   Solve(m, 1E-8);

   PostProcessing(m);
   DDD_Exit();
}
```

Step 2: Choosing DDD-Objects

```
struct Edge
{
   DDD_HEADER;

   /* numerical information */
   Real    aij, aji;

   /* connectivity */
   Node    *ni, *nj;
   Element *epos, *eneg;

   /* storage management */
   Edge    *next;
};
```

Use DDD Transfer operations to implement load migration strategy.

Issues:

- ⋆ data locality
- ⋆ overlapping (cf. formal specification)
- ⋆ consistency (definition of DDD Interfaces)
- ⋆ supplements, e.g., multi-level

Handlers are user-defined callback functions.

Example:

```
void CopyHandler (Elem *elem, Proc dest)
{
   int     i;

   for(i=0; i<3; i++)
      DDD_XferCopyObj(elem->node[i], dest);

   for(i=0; i<3; i++)
      DDD_XferCopyObj(elem->edge[i], dest);
}
```

Step 5: Load Balancing

⋆ external load-balancing-strategy
e.g. simple RCB or third-party: Jostle, Metis, Chaco

⋆ DDD-Transfer-module is used for

- initial static distribution
- runtime dynamic distributions

⋆ references are maintained

⋆ object copies are created

⋆ interfaces are transparent

Step 6: Synchronization

```
Solve (Mesh *m, Real accuracy)
{
   Real   residuum;

   DDD_IFExchange(theNodeIF,
      sizeof(Real),
      LocalAii, AddupAii);

   do {
      ComputeDefect(m);
      DDD_IFExchange(theNodeIF,
         sizeof(Real),
         LocalDi,AddupDi);

      residuum = UpdateSolution(m);
      residuum = GlobalSum(residuum);
   }
   while (residuum > accuracy)
}
```

Step 7: Additional Requirements

- ⋆ distributed mesh refinement
- ⋆ other mesh topology changes
- ⋆ parallel I/O
- ⋆ visualization
- ⋆ . . .

1. Parallelization of dynamic data programs is complicated.
2. A high-level programming model can help.
3. Careful software reuse is necessary.

For load migration, DDD will do the job.

Klaus Birken
birken@ica3.uni-stuttgart.de
http://www.ica3.uni-stuttgart.de/∼birken

DDD Homepage
http://www.ica3.uni-stuttgart.de/∼birken/ddd