A Distributed Functional Framework for Recursive FE-Simulations

Ralf Ebner

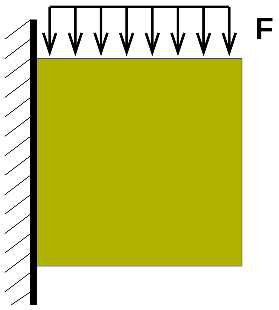
Technische Universität München

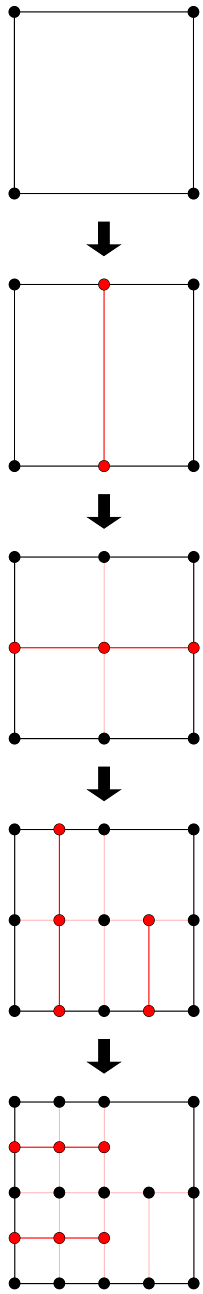
ebner@informatik.tu-muenchen.de
http://wwwzenger.informatik.tu-muenchen.de/

The basic algorithm:

Adaptive Recursive Substructuring

Example:
Short
Cantilever





Divide & conquer algorithm ARESO: adaptive recursive splitting of domain into subdomains

$\Rightarrow$ tree of finite elements with equation system in each node

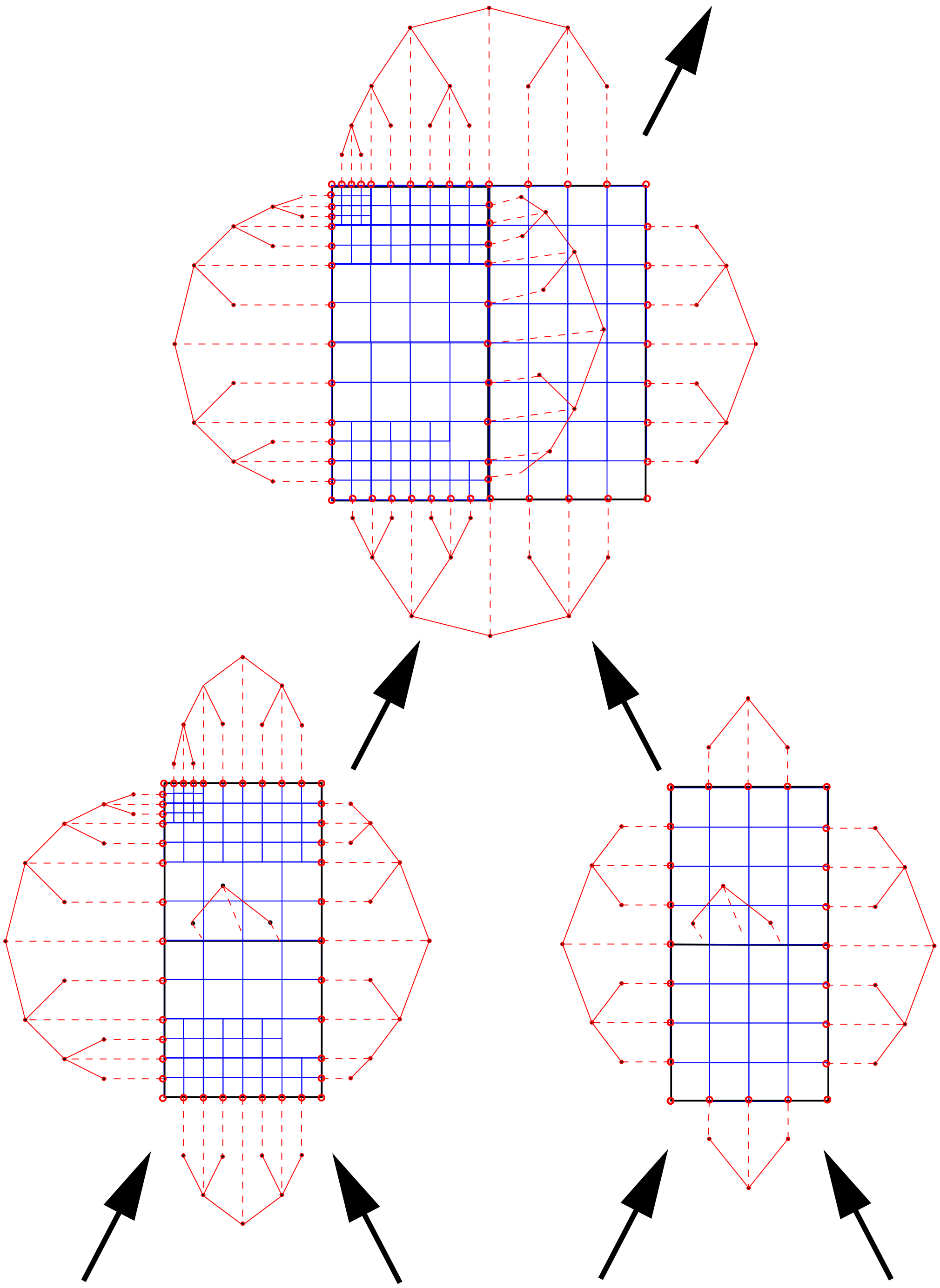
$$\begin{pmatrix} K_{ee} & K_{ei} \\ K_{ie} & K_{ii} \end{pmatrix} \begin{pmatrix} u_e \\ u_i \end{pmatrix} = \begin{pmatrix} f_e \\ f_i \end{pmatrix}$$

Tree Traversal:

- **Building** phase: refinement, bottom-up accumulation of matrices
- **Solving** phase: direct or iterative

Iteration of traversal, until discretization error is small enough

Data Structures



Supported Problem Classes

Requirements:

- Separator as central concept: problem parts must be well-defined by “boundary conditions”
  $\hookrightarrow$ locally limited coupling of degrees of freedom
- Linear combination of problem parts: Additive bottom-up accumulation of equation systems
- External FE programs for leaves: possibility to reduce the restart overhead

Advantages:

- Complexity: Dimension reduction in upper nodes
- Parallelization: Slim root
- Teamwork: distributed design by engineers

Examples:

- Static heat distribution
- Structural mechanics

Implementation Problems

1. static condensation $\Rightarrow$ full matrices
2. commercial applications: unknown stiffness matrix
3. difficult hand-coded parallelization
4. data locality, distributed data structures
5. adaptivity $\Rightarrow$ unbalanced tree

Static condensation vs. iterative solver

$$(K_{ee} - K_{ei}K_{ii}^{-1}K_{ie})u_e = f_e - K_{ei}K_{ii}^{-1}f_i$$

Problem:

The initially sparse matrix is filled up by conventional static condensation $\Rightarrow$ root node memory size limits recursion (and refinement) depth

Alternative: iterative solver with recursive relaxation and adaption of RHS within each tree node

Top-Down:

$$\tilde{f}_i^{(n)} = f_i^{(n)} - K_{ie}u_e^{(n)}$$

$$u_i^{(n+1)} = \text{GaußSeidel}(K_{ii}, \tilde{f}_i^{(n)}, u_i^{(n)})$$

$$\theta_i^{(n+1)} = u_i^{(n+1)} - u_i^{(n)}$$

Bottom-Up:

$$r^{(n+\frac{1}{2})} = \sum_{s \in sons} (r_s)_e^{(n+1)}$$

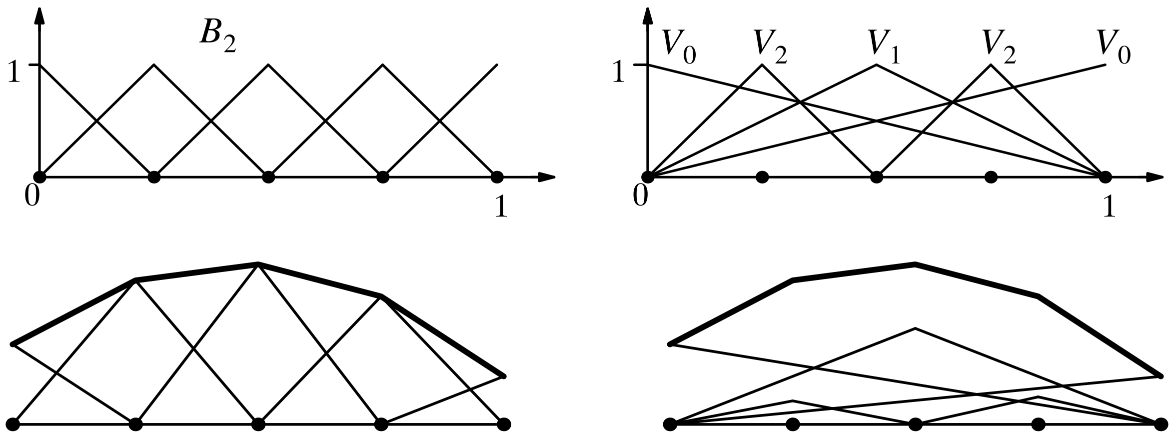
$$f^{(n+1)} = f^{(n)} + r^{(n+\frac{1}{2})}$$

$$r_e^{(n+1)} = r_e^{(n+\frac{1}{2})} - K_{ei}\theta_i^{(n+1)}$$

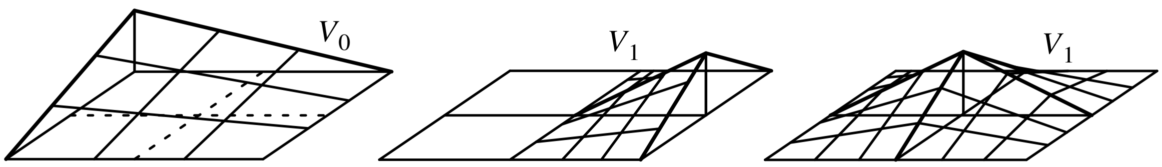
Convergence

Ad hoc implementation of iterative solver converges very slowly at fine discretization (large equation systems)
Transformation to a **hierarchical basis** improves the condition of the system considerably ([Hüttl96])

1D:



2D:



Restrictions

Commercial finite element solvers

- graphical user interface
- interface definition language
- complicated and real world simulations

Drawbacks:

– only sequential programs
  $\hookrightarrow$ combined with ARESO

– expensive licence for each workstation
  $\hookrightarrow$ team work with ARESO

– no access to system matrix
  $\hookrightarrow$ solver requiring only matrix vector product (e. g. CG, GMRES)

Parallelization

Recursion and adaptivity ⇒ handcoding is tedious and a source of errors (deadlocks, races,...)

Automatic parallelization does not work well for recursion in general programming languages

Better approach: Functional programming

- Functions describe mapping from input to output data ⇒ no side effects
- Identifiers keep their assigned value
  no „procedure variable“, no pointers
  ⇒ no implicit global computation state
- Functions can be evaluated whenever all input data is available (Church-Rosser property)
  ⇒ no explicit execution sequence

Therefore:

+ clear data dependence
+ programs easier to understand
+ soundness proofs
+ automatic detection of potential parallelism
+ free of deadlocks

Examples: Haskel GHC, SISAL, FASAN

Functional Framework: Tree Construction

```
data Node = Node {K     ::Matrix,
                  U,F,R::Vector,
                  Ixtree::IndexTree,
                  Ixtrafo::Vector}
data Tree = Leaf   {Nd::Node}
          | Branch {Nd::Node, Sons::[Tree]}

createLeaf  :: [BoundarySpec] -> Node
combineNodes:: Node -> [Node] -> Node
dontRefine  :: Tree -> Bool

buildTree :: Tree -> Tree
buildTree t =
    if dontRefine(t) then t
    else let sons_ = case t of
            Leaf nd -> map createLeaf spec
            Branch (nd,sons) -> map buildTree sons
         in Branch (combineNodes nd (map Nd sons_), sons_)
```

Functional Framework: Iterative Solver

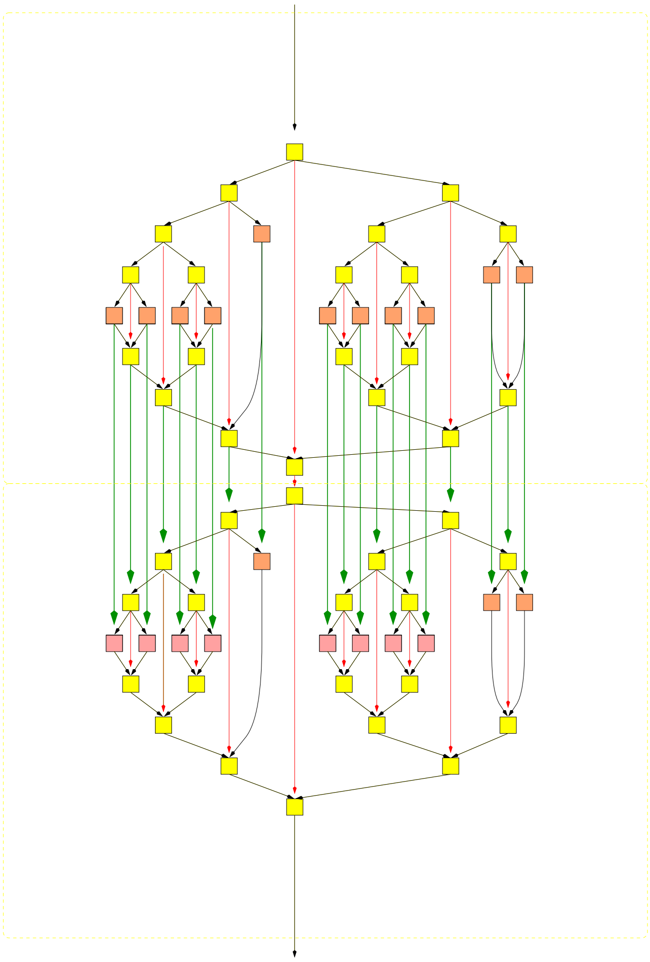
```
displace :: Node -> Node -> Node
react    :: Node -> [Node] -> Node
leafSolve:: Node -> Node

relax :: Node -> Tree -> (Tree, Float)
relax    parent  t    =  (t_,   resid)
  where
    t_ = case t of
      Leaf nd -> Leaf (leafSolve nd)
      Branch (nd, sons) -> let
          nd_ = displace nd par
          sons_ = map (relax nd_) sons
        in Branch (react nd_ (map Nd sons_), sons_)
    resid = norm (R nd)

solveIter :: Tree -> Tree
solveIter t =
   let (t_, res) = relax (Nd t) t
   in if not tooBig(res) then t_
         else solveIter t_
```

Locality of data

In large simulations, main memory of a computer/processor is sufficient for just a few tree nodes.
$\Rightarrow$ **Implementation must keep data local**



But:
parallelizing compilers for functional programs usually distribute function calls and are not concerned in keeping data local.

Haskell (GHC):
monads with references to tree nodes as allocated C data structures.

FASAN system:
small coordinating language that wraps tree components in streams without copying their contents.

Adaptivity

Imbalanced tree and different work load in subtrees
⇒ **Dynamic load balancing necessary**

Implementation: redistribution of tree nodes after each iteration step

- Bisectors of trees (acceptable worst cases)
- Hashing techniques
- Thread environment (Java, POSIX Threads with scheduler; SMP model)
- Distribution by some load balancing library (TUM SFB 342)

Summary

fat root node with static condensation	$\succ$	iterative solver preserves sparse matrix
slow convergence	$\succ$	transformation to hierarchical basis
unknown stiffness matrix in leaves	$\succ$	only one level; CG, GMRES
difficult hand-coded parallelization	$\succ$	**functional framework**: automatic parallelizing layer
data locality, distributed data structures	$\succ$	Haskell: monads FASAN: stream semantics and wrapper streams
adaptivity $\Rightarrow$ unbalanced tree and work load	$\succ$	dynamic load balancing: hashing, bisectors, library, thread scheduler