SCALABLE ALGORITHMS
for solving
large sparse linear systems of equations

CONTENTS

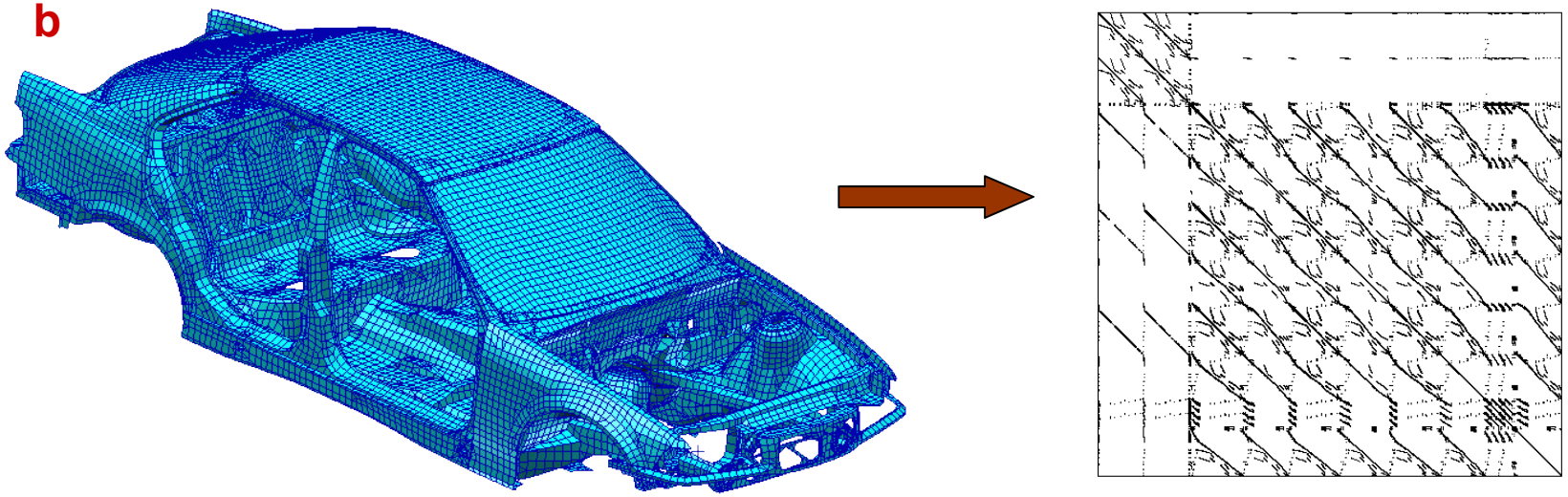
- **Sparse direct solvers (multifrontal)**
- **Substructuring methods (hybrid solvers)**

Jacko Koster, Bergen Center for Computational Science, University of Bergen

(prepared with help from the MUMPS team, esp. Patrick Amestoy and Jean-Yves L'Excellent)

Introduction

Simulation of a physical problem often leads to sparse systems A x = b



- The (repeated) solution of sparse linear systems of equations is often the most computationally intensive part of a simulation process
- Problem: solve **A x = b**, where **A** is square n x n, sparse, and symmetric positive definite, symmetric indefinite, or unsymmetric, and possibly rank deficient, vector **b** of length n is given, vector **x** of length n is to be computed
- Develop software libraries tailored for parallel processing, in particular distributed-memory platforms (loosely connected SMPs, clusters)

Direct methods - Introduction

Solve A x = b :

1. **LU factorization**: P A Q = LU or P A Q = LDL^T

   P is row permutation, Q is column permutation,
   L is lower triangular, U is upper triangular matrix

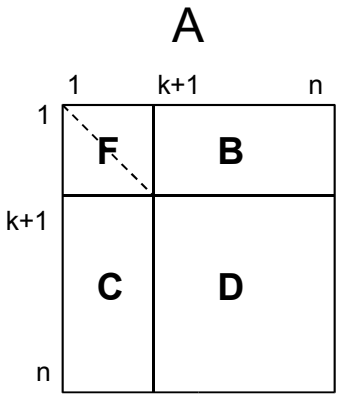
   Dominant part of computation is O(n^3):

```
for k = 1,n          /*  eliminate variable k  */
  for i = k+1,n      /*  overwrite A[ k+1:n, k+1:n ]  */
    for j = k+1,n
```

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)} \, a_{kj}^{(k)}}{a_{kk}^{(k)}}$$

2. Solve L y = P b, for vector y (forward substitution, O(n^2))
3. Solve U [Q]$^{-1}$ x = y, for vector x (back substitution, O(n^2))

Block version: eliminate k variables



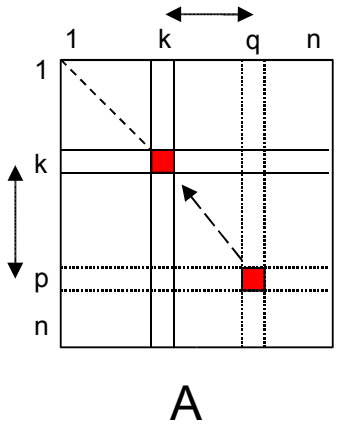
D = D – C [F]$^{-1}$ B

Uses BLAS1, BLAS2, and BLAS3 for high performance

Direct methods - Introduction

Why permutations P and Q in P A Q = LU ?

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)} \, a_{kj}^{(k)}}{a_{kk}^{(k)}}$$



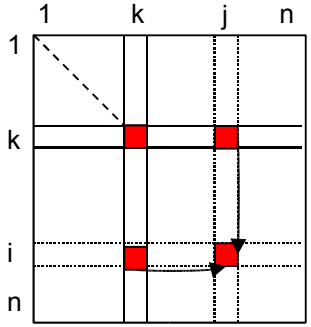
A

P, Q needed to find large (pivots) $\mathbf{a}_{kk}^{(k)}$

→ numerical accuracy

P, Q needed to preserve sparsity: make as few as possible $\mathbf{a}_{ij}^{(k)}$ nonzero (fill-in)

→ keep L and U sparse → computational and memory savings

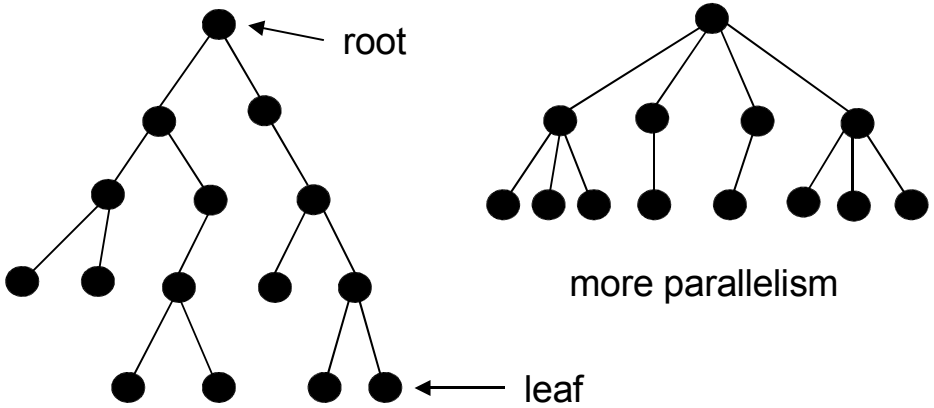


P and Q define variable ordering In the matrix

Multifrontal direct methods

Basic idea: a large sparse matrix is a sum of smaller dense matrices

- The factorization of matrix A is driven by an elimination tree that is determined by the matrix structure and the variable ordering (permutations P and Q).
- A node in the tree represents a partial factorization of a (small) dense matrix.
- An edge in the tree represents data movement between dense matrices.
- The tree defines a partial ordering: a node can only be processed when all its children are processed. Leaves are processed first; root comes last. Nodes that are not ancestors to one another can be processed simultaneously (parallelism).



more parallelism

tridiagonal matrix with natural ordering; no parallelism

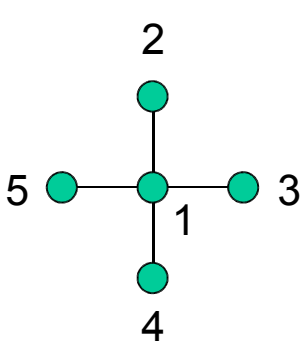
diagonal matrix; embarrasing parallel

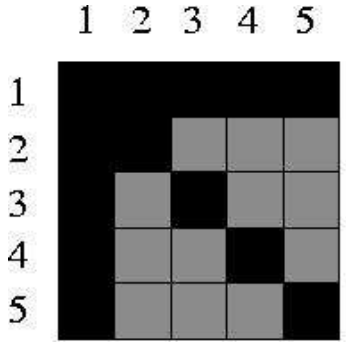
Sparse direct methods

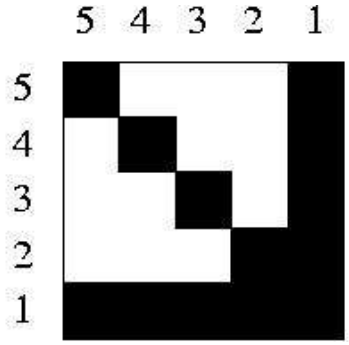
Variable ordering has large impact on performance

- Ordering impacts elimination tree, parallelism, computation, and memory use
- It easily pays off to spend time on analyzing the matrix before factorization
- However, finding the optimal ordering to minimize fill-in is an NP-complete problem

Use heuristics to study/optimize topology of the tree (number of nodes, sizes of nodes, …)





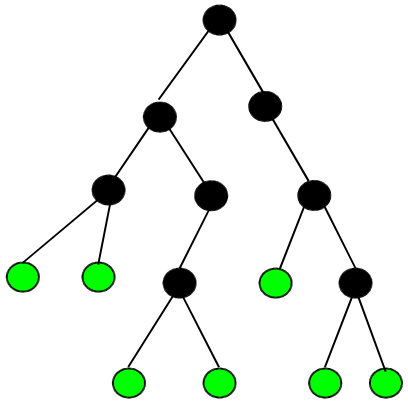


no fill-in

Fill-in (entries created during factorization)

Initial matrix

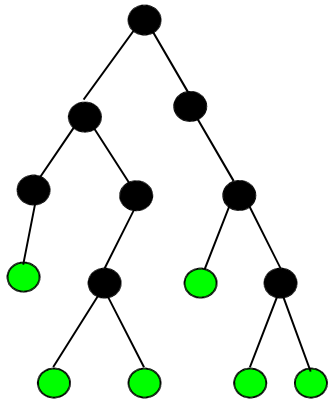
Multifrontal direct methods



7 nodes ready to be processed

8 nodes not ready to be processed

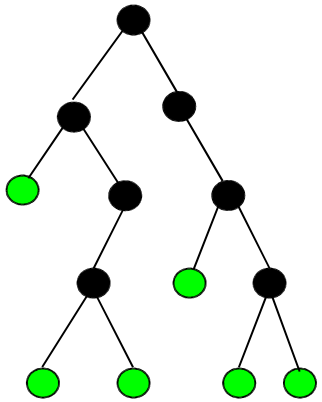
Multifrontal direct methods



6 nodes ready to be processed

8 nodes not ready to be processed

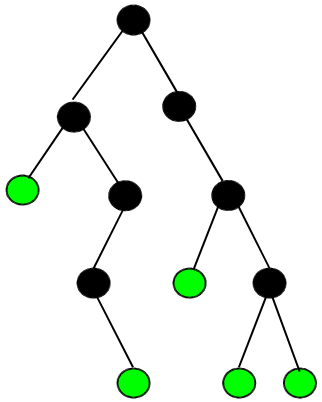
Multifrontal direct methods



6 nodes ready to be processed

7 nodes not ready to be processed

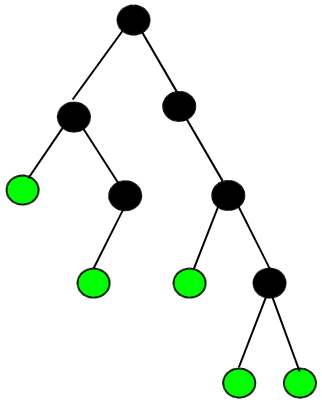
Multifrontal direct methods



5 nodes ready to be processed

7 nodes not ready to be processed

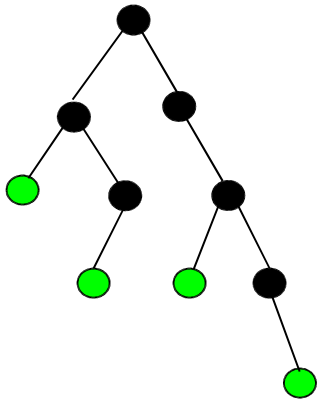
Multifrontal direct methods



5 nodes ready to be processed

6 nodes not ready to be processed

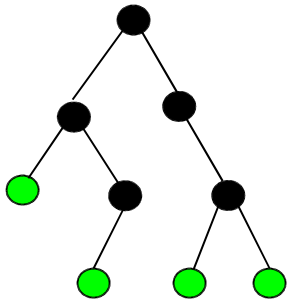
Multifrontal direct methods



4 nodes ready to be processed

6 nodes not ready to be processed

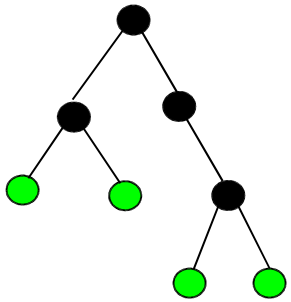
Multifrontal direct methods



4 nodes ready to be processed

5 nodes not ready to be processed

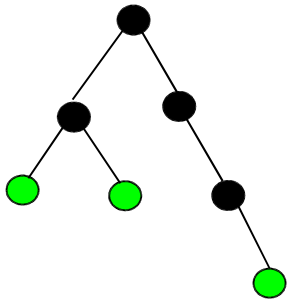
Multifrontal direct methods



4 nodes ready to be processed

4 nodes not ready to be processed

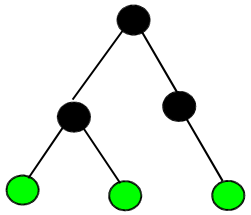
Multifrontal direct methods



3 nodes ready to be processed

4 nodes not ready to be processed

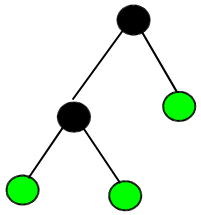
Multifrontal direct methods



3 nodes ready to be processed

3 nodes not ready to be processed

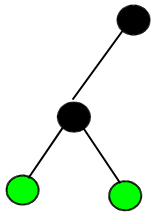
Multifrontal direct methods



3 nodes ready to be processed

2 nodes not ready to be processed

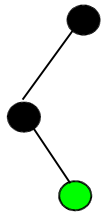
Multifrontal direct methods



2 nodes ready to be processed

2 nodes not ready to be processed

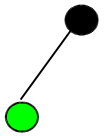
Multifrontal direct methods



1 nodes ready to be processed

2 nodes not ready to be processed

Multifrontal direct methods

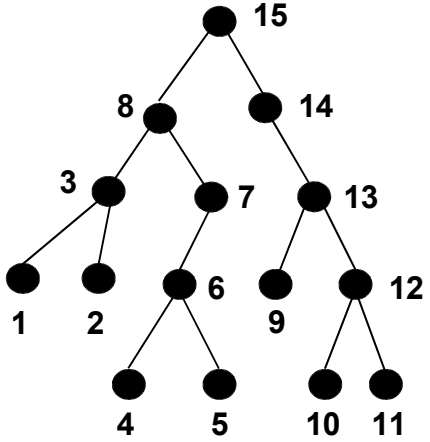


1 nodes ready to be processed

1 nodes not ready to be processed

Multifrontal direct methods

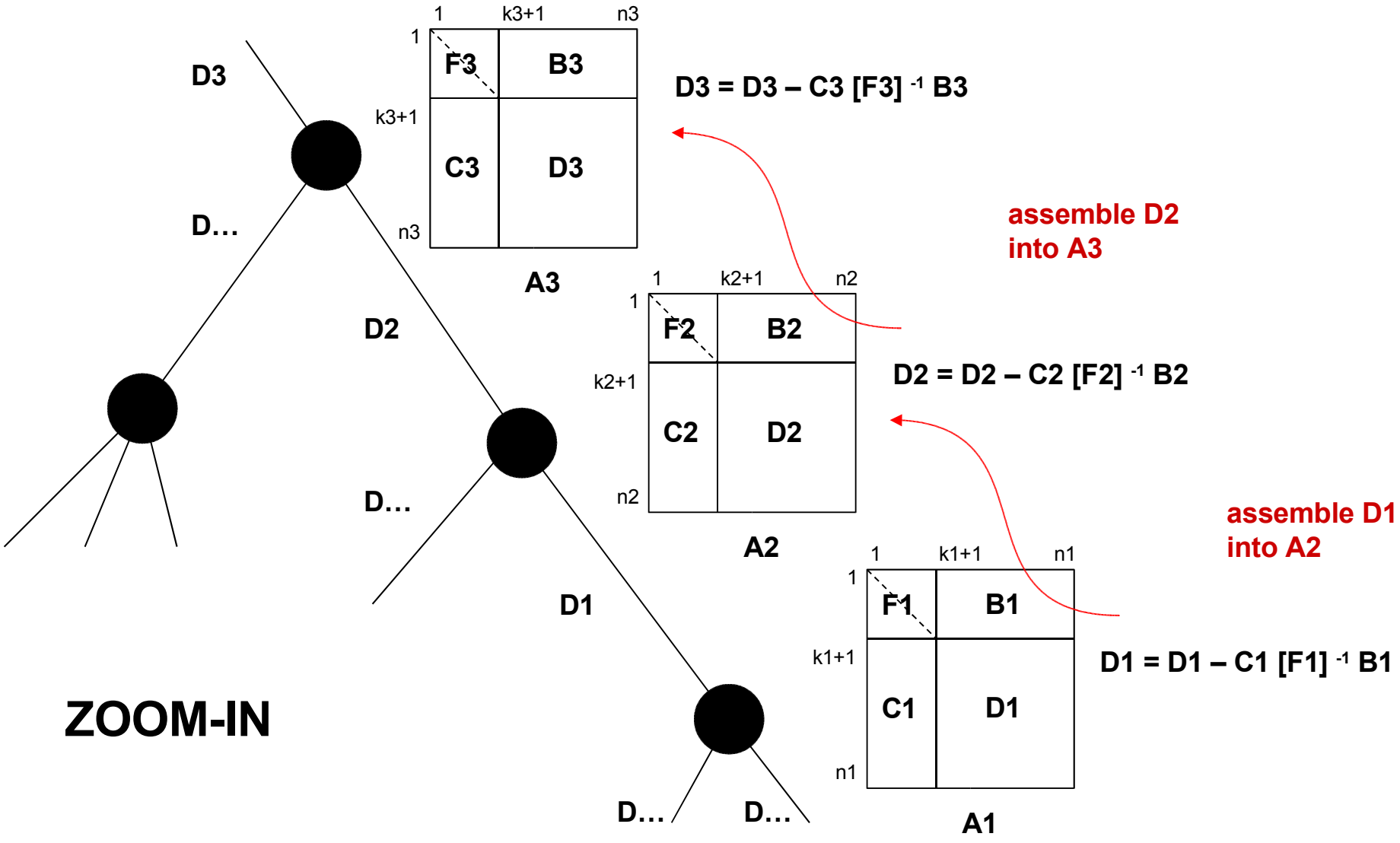
(DFS numbering)



 1 node ready to be processed

ps. In practice, tree could be processed in many orders. A depth-first search (DFS) order allows the frontal matrices to be stored on a stack.

Multifrontal direct methods



Multifrontal direct methods

Three phases to solve A x = b:

1. Analysis of matrix A (symbolic factorization):

Determine appropriate P and Q based on nonzero structure of A to minimize fill-in in L and U. Compute the elimination tree. Prepare for parallel execution, map tree nodes onto processors. Each processor prepares the local data structures.

2. Factorization of A (numerical factorization):

Compute L and U using the tree. Possibly modify P and Q a-posteriori to ensure numerical stability.

3. Forward/back substitution:

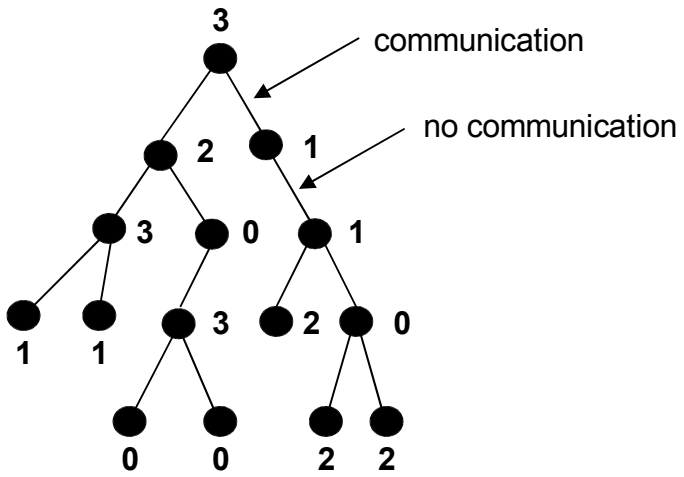
Use P, Q, L, and U to compute $x = [A]^{-1} b$

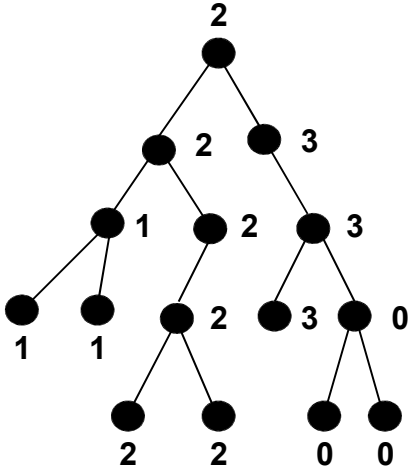
→ For a sequence of systems A x1 = b1, A x2 = b2, … : do steps **1** and **2** only once.

→ For a sequence of systems A1 x1 = b1, A2 x2 = b2, … with A1, A2, … having the same nonzero structure: do step **1** only once.

Multifrontal direct methods

In a distributed-memory environment: map nodes of the tree onto processors



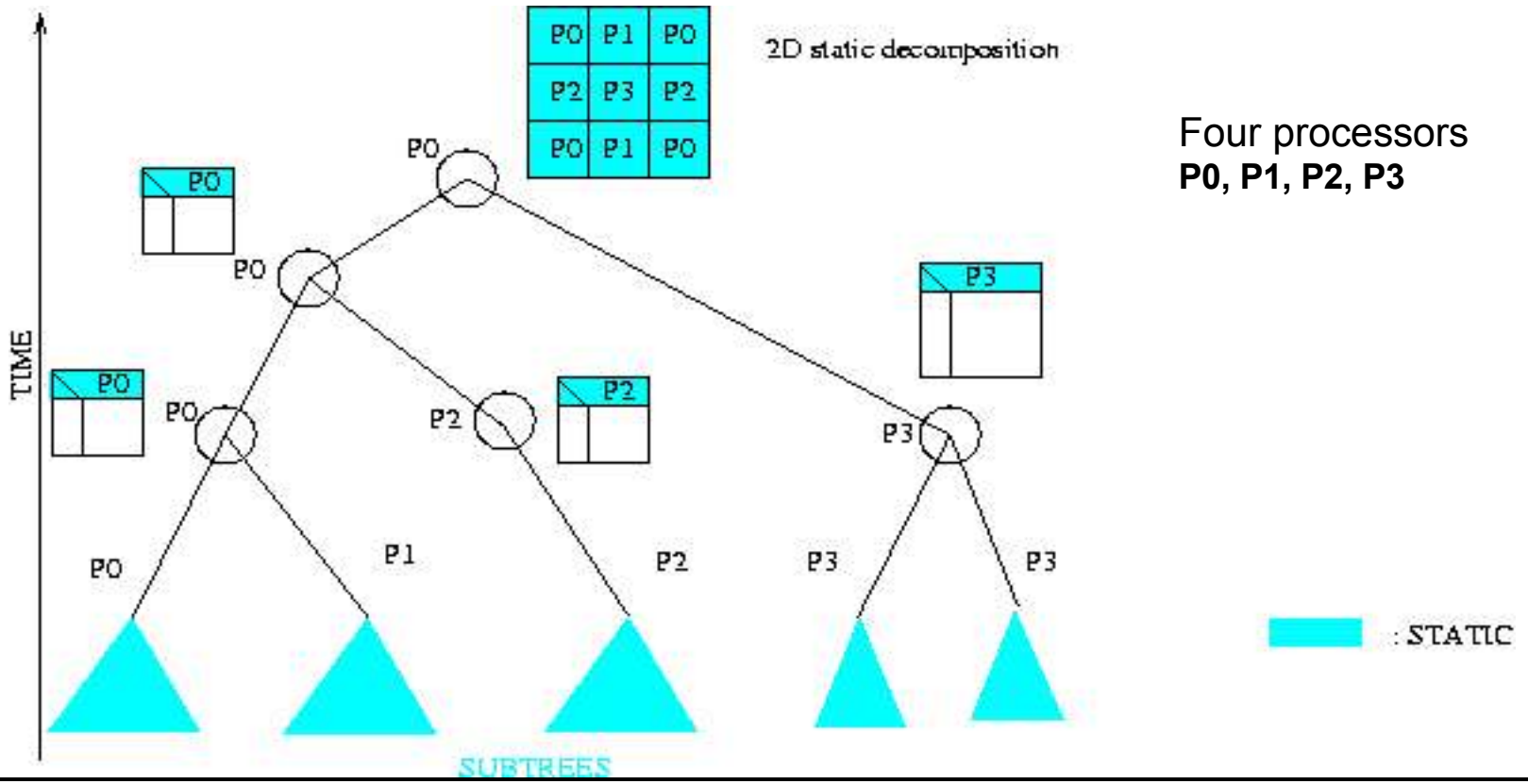


Four processors
0 ,1, 2, 3

Constraints;
- Each node has its own amount of work
- Tree defines node dependencies
- Processors must get roughly the same amount of total work (work load balance)
- Minimize inter-processor communication
- Minimize idling of processors

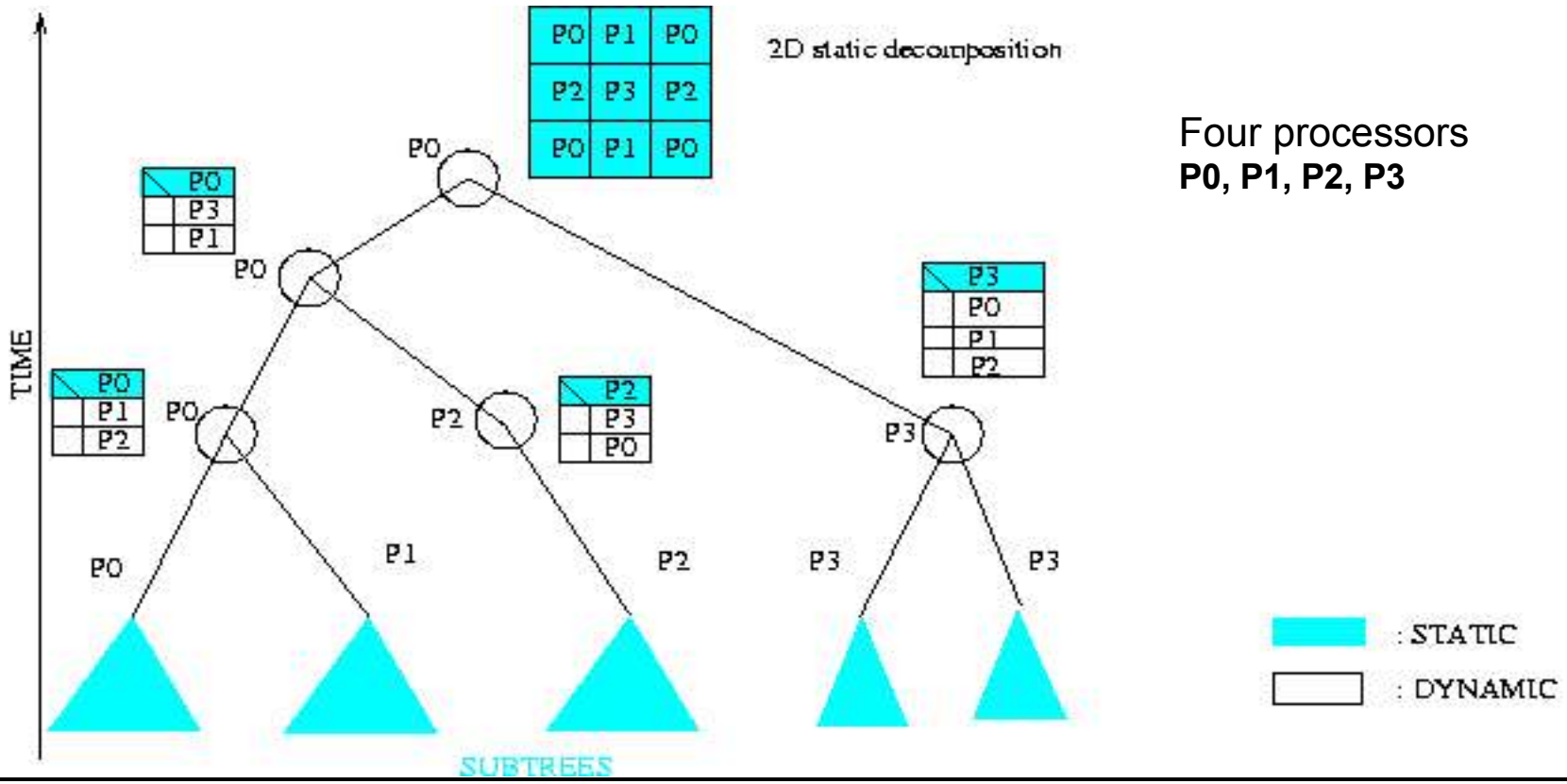
MUMPS parallel multifrontal scheme

- Task granularity: assign subtrees to processors to minimize communication
- In practice, nodes near the root require much work (often 75% of work is in few top levels)
- Reduced parallelism near the root (relatively few nodes, many processors)
- Use multiple processors to process large nodes near the root:



MUMPS parallel multifrontal scheme

- Allow dynamic assignment of matrices during factorization to take care of numerical stability issues



Four processors
P0, P1, P2, P3

MUMPS in a cluster environment

- Flop-based scheduling strategy appears to be most natural (work balance, minimize elapsed time). Works fine in shared-memory environment.

- However, a good work balance does not necessarily imply a good memory balance. On clusters (of small SMPs) with little memory per node, a bad memory balance may make computing a solution impossible.

  → memory scalability and memory load balance are as important

  → need for memory-aware task scheduling

- Similarly, there is a need for interconnect-aware task scheduling to take into account network latencies and bandwidths…

MUMPS package http://graal.ens-lyon.fr/MUMPS

- Background:
  - Project continued from LTR European project PARASOL (1996-1999)
  - Developers and contributors:

    Patrick Amestoy (ENSEEIHT-IRIT, Toulouse)
    Jean-Yves L'Excellent (ReMAP project, INRIA, Lyon)
    Iain Duff (CERFACS, Toulouse and RAL, UK)
    Abdou Guermouche (ReMAP project, Toulouse)
    Jacko Koster (Parallab, BCCS, Norway)
    Stéphane Pralet (CERFACS, Toulouse)
    Christophe Vömel (CERFACS, Toulouse)

- General purpose, competitive, many functionalities
  - Types of matrices: SPD, symmetric, unsymmetric
  - Input matrix format: assembled, elemental, distributed
  - Arithmetic: real, double, complex, double complex
  - Numerical pivoting, scalings, backward error analysis, iterative refinement
  - Written in F90 and MPI; C interface provided

- MUMPS 4.3.2 (latest public release, July 2003)
  - Requested/downloaded by ca. 500 users
  - Ca. 200.000 lines of code and growing …
  - Freely available software

Some MUMPS usage

- PETSc (Argonne National Laboratory)
  - MUMPS available from PETSc 2.2.0 library as an optional package
- Academic and industrial users from various application fields
  - Structural mechanical engineering
  - Biomechanics
  - Heat transfer analysis
  - Medical image processing
  - Geophysics
  - Optical problems
  - Ad-hoc network modeling (Markov processes)
  - Econometric modeling
  - Oil reservoir simulation
  - Computation fluid dynamics
  - Astrophysics
  - Circuit simulation
- Used by EADS, Dassault, CEA, Boeing, NEC, THALES, NASA, MIT, several US national labs, …

SALSA - Introduction

Iterative substructuring for solving A x = b

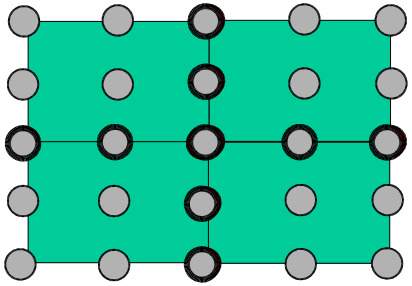
The idea is to combine the best of direct methods (robustness) with that of iterative methods (speed). Reorder A into bordered block diagonal form :

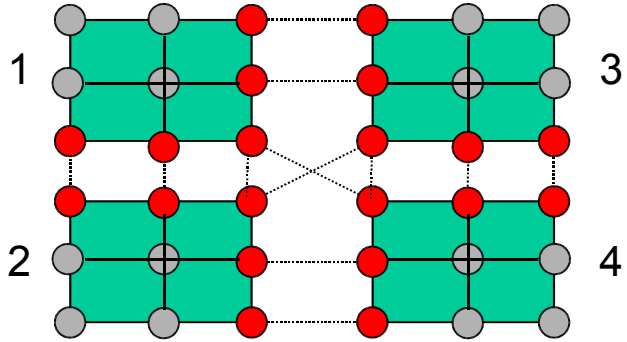
$$\begin{pmatrix} A_{ii}^1 & & & & A_{ir}^1 R_1^T \\ & A_{ii}^2 & & & A_{ir}^2 R_2^T \\ & & \ddots & & \vdots \\ & & & A_{ii}^N & A_{ir}^N R_N^T \\ R_1 A_{ri}^1 & R_2 A_{ri}^2 & \dots & R_N A_{ri}^N & \sum_{p=1}^{N} R_p A_{rr}^p R_p^T \end{pmatrix} \cdot \begin{pmatrix} x_i^1 \\ x_i^2 \\ \vdots \\ x_i^N \\ x_r \end{pmatrix} = \begin{pmatrix} b_i^1 \\ b_i^2 \\ \vdots \\ b_i^N \\ \sum_{p=1}^{N} R_p b_r^p \end{pmatrix}$$

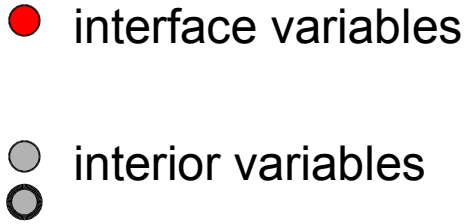
This represents N non-overlapping subdomains. The interior matrices A_{ii} can be processed simultaneously which provides a natural source of parallelism.

SALSA - Introduction

Example : 4 subdomains, 9 interface variables, 16 interior variables







The solution process for A x = b consists of three main steps.

3. Eliminate interior variables in A_{ii}^{p} (in parallel for each subdomain), e.g., with a sparse direct solver like MUMPS
4. Eliminate interface variables in A_{rr}^{p} with use of local Schur complement matrices $S(p) = A_{rr}^{p} - A_{ri}^{p} A_{ii}^{p} A_{ir}^{p}$ typically with preconditioned iterative method (PCG, Bi-CGSTAB, GMRES, ...)
5. Postprocessing to obtain final solution

SALSA features

SALSA accepts matrices in various formats

- Assembled format (traditional Compressed Sparse Row)
- Elemental format (sum of small dense matrices)
- Partitioned format (sum of N matrices in CSR format, one per subdomain)

The number of processors N is independent of number of subdomains P

- P = N : one domain per processor (default)
- N > P : map multiple subdomains onto one processor
  of interest for convergence studies of preconditioners
- P > N : map multiple processors onto one subdomain
  of interest for load balancing and optimal use of available resources

SALSA works with multiple internal Schur complement matrix formats

- Explicit : the matrix S(p) is computed explicitly (e.g., by MUMPS)
- Implicit : the action of S(p) is derived from its definition
  (3 mat-vec products, 1 fw/bw solve)

SALSA features

1-level preconditioners include:

Jacobi/Diagonal

Block (incomplete) LU

Neumann-Neumann (Deroeck & Le Tallec '91)

2-level preconditioners: based on Balancing Neumann-Neumann (Mandel '93)

Requires approximation of local null spaces in case S(p) is singular

1. computed algebraically based on deriving smallest eigenvalue
2. known a-priori depending on problem type (elasticity, etc.)

Direct substructuring:

the interface problem can also solved with a direct method to provide maximum robustness

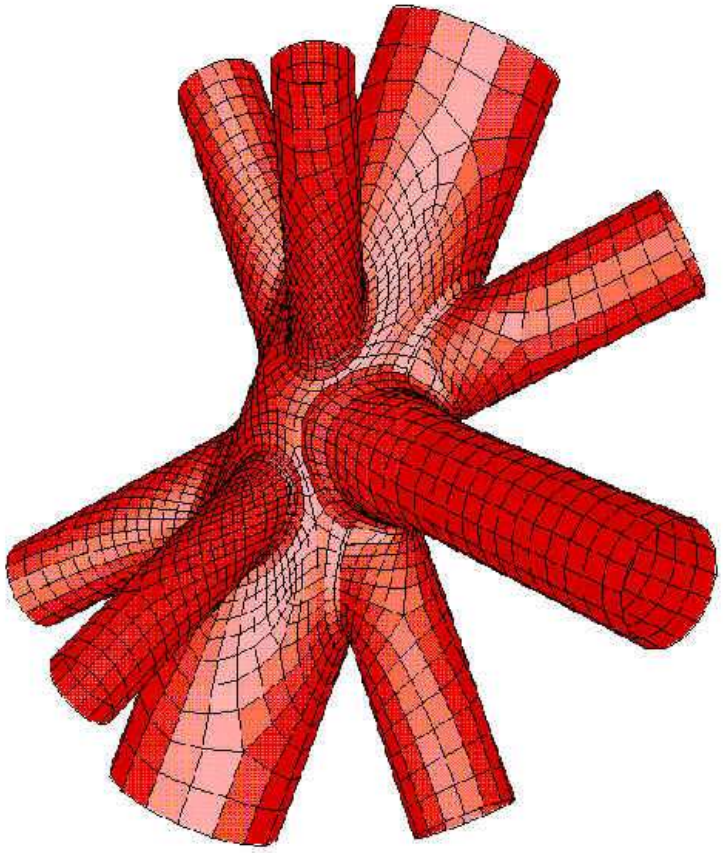
SALSA example of usage

DNV tubular joint problem :

- Partitioned by DNV into 58 subdomains of varying sizes
- Mix of solid, shell, transitional finite elements
- 97,470 dofs, of which 13,920 on interface

nprocs	#iter	preproc (secs)	PCG (secs)	total (secs)
1	32	29.0	16.6	45.7
2	31	16.5	8.7	25.3
4	31	9.7	4.3	13.9
8	31	6.1	2.1	8.3
12	31	5.3	1.8	7.2

Non-optimal speedups for larger number of processors primarily due to differences in domain sizes ('naive' cyclic mapping of domains used). Results obtained on IBM p690 (power4 1.3 Ghz)



SALSA on-going/future work

More robust 2-level preconditioners:

Collaboration with L. Giraud (CERFACS)

Collaboration with J. Mandel (Univ. of Colorado)

Automatic load balancing:

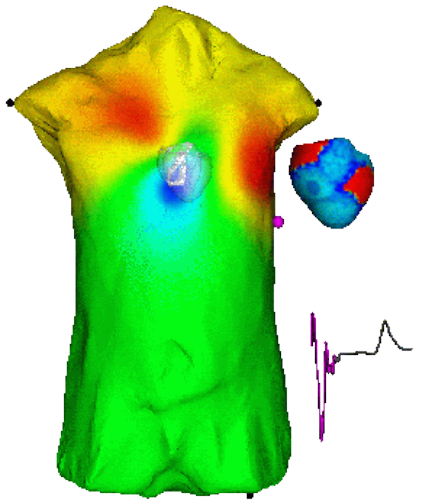
Needed when the number of subdomains differs from the number of processors or when subdomains vary in size and work

Inclusion of more iterative schemes:

Primarily based on user requests

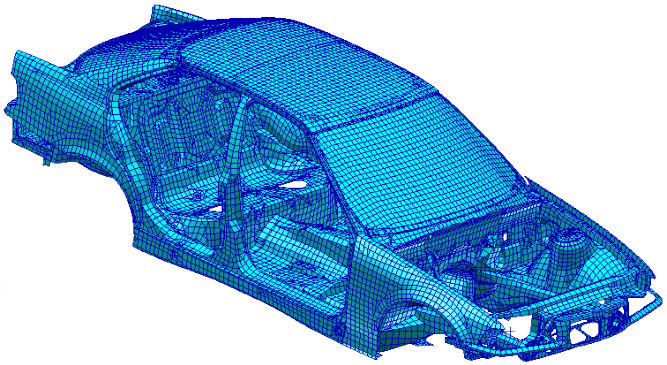
Pursue further collaboration with Simula:

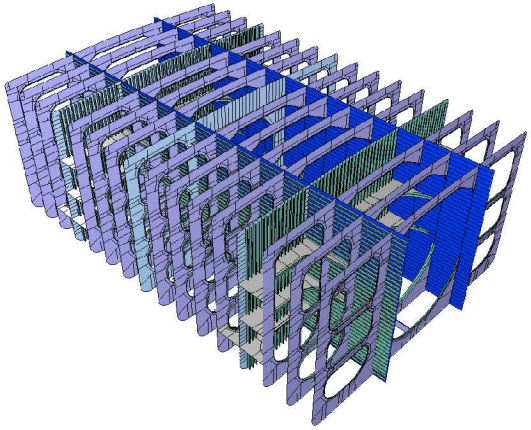
To integrate SALSA into a challenging application for simulating the electrical activity of the heart (talk Xing Cai tomorrow)

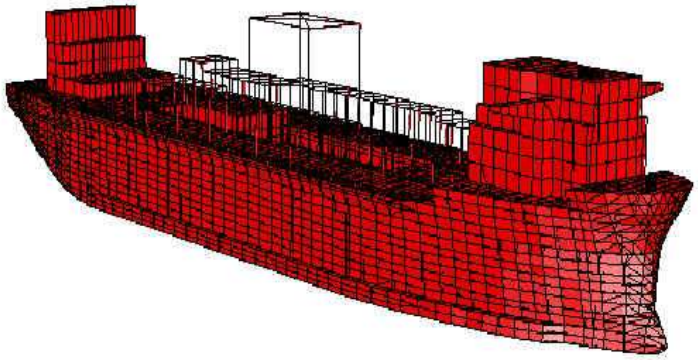


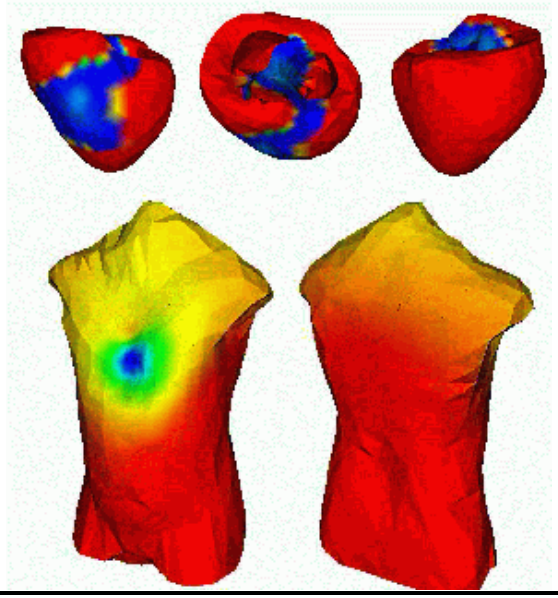
Miscellaneous

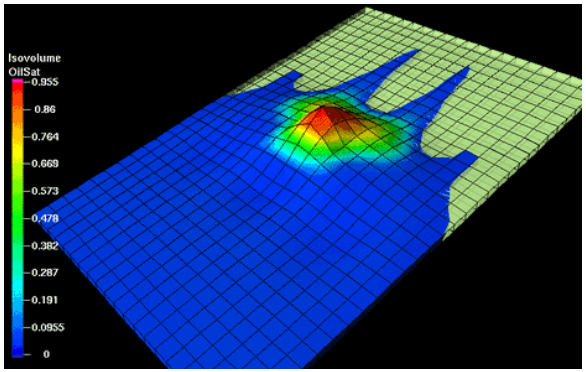
MUMPS and/or SALSA is being tested/developed on problems like:











How it fits together