Sparse Linear Algebra: Direct Methods, advanced features

P. Amestoy and A. Buttari
(INPT(ENSEEIHT)-IRIT)

A. Guermouche (Univ. Bordeaux-LaBRI),
J.-Y. L’Excellent and B. Uçar (INRIA-CNRS/LIP-ENS Lyon)
F.-H. Rouet (Lawrence Berkeley National Laboratory)
C. Weisbecker INPT(ENSEEIHT)-IRIT

2013-2014
Sparse Matrix Factorizations

- \( A \in \mathbb{R}^{m \times m} \), symmetric positive definite \( \rightarrow LL^T = A \)

  \[
  Ax = b
  \]

- \( A \in \mathbb{R}^{m \times m} \), symmetric \( \rightarrow LDL^T = A \)

  \[
  Ax = b
  \]

- \( A \in \mathbb{R}^{m \times m} \), unsymmetric \( \rightarrow LU = A \)

  \[
  Ax = b
  \]

- \( A \in \mathbb{R}^{m \times n} \), \( m \neq n \) \( \rightarrow QR = A \)

  \[
  \min_x \|Ax - b\| \quad \text{if} \quad m > n
  \]

  \[
  \min \|x\| \quad \text{such that} \quad Ax = b \quad \text{if} \quad n > m
  \]
Cholesky on a dense matrix

\[
\begin{pmatrix}
a_{11} \\
a_{21} & a_{22} \\
a_{31} & a_{32} & a_{33} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\]

left-looking Cholesky

\[
\begin{align*}
&\text{for } k = 1, \ldots, n \text{ do} \\
&\quad \text{for } i = k, \ldots, n \text{ do} \\
&\quad\quad \text{for } j = 1, \ldots, k - 1 \text{ do} \\
&\quad\quad\quad a_{ik}^{(k)} = a_{ik}^{(k)} - l_{ij}l_{kj} \\
&\quad\text{end for} \\
&\quad l_{kk}^{(k)} = \sqrt{a_{kk}^{(k-1)}} \\
&\quad \text{for } i = k + 1, \ldots, n \text{ do} \\
&\quad\quad l_{ik} = a_{ik}^{(k-1)} / l_{kk}^{(k)} \\
&\quad\quad \text{for } j = k + 1, \ldots, i \text{ do} \\
&\quad\quad\quad a_{ij}^{(k)} = a_{ij}^{(k)} - l_{ik}l_{jk} \\
&\quad\text{end for} \\
&\quad \text{end for} \\
&\text{end for}
\end{align*}
\]

right-looking Cholesky

\[
\begin{align*}
&\text{for } k = 1, \ldots, n \text{ do} \\
&\quad l_{kk}^{(k)} = \sqrt{a_{kk}^{(k-1)}} \\
&\quad \text{for } i = k + 1, \ldots, n \text{ do} \\
&\quad\quad l_{ik} = a_{ik}^{(k-1)} / l_{kk}^{(k)} \\
&\quad\quad \text{for } j = k + 1, \ldots, i \text{ do} \\
&\quad\quad\quad a_{ij}^{(k)} = a_{ij}^{(k)} - l_{ik}l_{jk} \\
&\quad \text{end for} \\
&\quad \text{end for} \\
&\text{end for}
\end{align*}
\]
The factorization of a sparse matrix is problematic due to the presence of fill-in.

The basic LU step:

\[
a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - \frac{a_{i,k}^{(k)} a_{k,j}^{(k)}}{a_{k,k}^{(k)}}
\]

Even if \(a_{i,j}^{(k)}\) is null, \(a_{i,j}^{(k+1)}\) can be a nonzero

- the factorization is more expensive than \(O(nz)\)
- higher amount of memory required than \(O(nz)\)
- more complicated algorithms to achieve the factorization
Because of the fill-in, the matrix factorization is commonly preceded by an **analysis** phase where:

- the fill-in is reduced through **matrix permutations**
- the fill-in is predicted though the use of (e.g.) **elimination graphs**
- computations are structured using **elimination trees**

The analysis must complete much faster than the actual factorization
Analysis
Three main classes of methods for minimizing fill-in during factorization

Local approaches: At each step of the factorization, selection of the pivot that is likely to minimize fill-in.
- Method is characterized by the way pivots are selected.
- Markowitz criterion (for a general matrix)
- Minimum degree (for symmetric matrices)

Global approaches: The matrix is permuted so as to confine the fill-in within certain parts of the permuted matrix
- Cuthill-McKee, Reverse Cuthill-McKee
- Nested dissection

Hybrid approaches: First permute the matrix globally to confine the fill-in, then reorder small parts using local heuristics.
The elimination process in the graphs

\[ G_U(V, E) \leftarrow \text{undirected graph of } A \]

\[ \text{for } k = 1 : n - 1 \text{ do} \]
\[ \quad V \leftarrow V - \{k\} \{\text{remove vertex } k\} \]
\[ \quad E \leftarrow E - \{(k, \ell) : \ell \in \text{adj}(k)\} \cup \{(x, y) : x \in \text{adj}(k) \text{ and } y \in \text{adj}(k)\} \]
\[ \quad G_k \leftarrow (V, E) \{\text{for definition}\} \]
\[ \text{end for} \]

\[ G_k \] are the so-called elimination graphs (Parter,'61).

\[ H_0 = \]

\[ G_0 : \]

\[ 1 \quad 2 \quad 3 \]
\[ \times \quad \times \quad \times \]
\[ 4 \quad 5 \quad 6 \]

\[ 1 \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \time
The elimination tree is a graph that expresses all the dependencies in the factorization in a compact way.

**Elimination tree: basic idea**

If $i$ depends on $j$ (i.e., $l_{ij} \neq 0$, $i > j$) and $k$ depends on $i$ (i.e., $l_{ki} \neq 0$, $k > i$), then $k$ depends on $j$ for the transitive relation and therefore, the direct dependency expressed by $l_{kj} \neq 0$, if it exists, is redundant.

For each column $j < n$ of $L$, remove all the nonzeros in the column $j$ except the first one below the diagonal.
Let $L_t$ denote the remaining structure and consider the matrix $F_t = L_t + L^T_t$. The graph $G(F_t)$ is a tree called the elimination tree.
The elimination tree has several uses in the factorization of a sparse matrix:

- it expresses the order in which variables can be eliminated:
  - the elimination of a variable only affects (directly or indirectly) its ancestors and ...
  - ... only depends on its descendants

Therefore, any **topological order** of the elimination tree leads to a **correct result** and to the **same fill-in**

- it expresses concurrence: because variables in separate subtrees do not affect each other, they can be eliminated in **parallel**
Matrix factorization
The Cholesky factorization of a sparse matrix can be achieved with a left-looking, right-looking or multifrontal method.

**Reference case**: regular $3 \times 3$ grid ordered by nested dissection. Nodes in the separators are ordered last (see the section on orderings)

Notation:
- $\text{cdiv}(j) = \sqrt{A(j,j)}$ and then $A(j + 1 : n, j)/\sqrt{A(j,j)}$
- $\text{cmod}(j,k) = A(j:n,j) - A(j,k) \times A(j:n,k)$
- $\text{struct}(L(1:k,j))$: the structure of $L(1:k,j)$ submatrix
In the left-looking method, before variable $j$ is eliminated, column $j$ is updated with all the columns that have a nonzero on line $j$. In the example above, $\text{struct}(L(7,1:6)) = \{1, 3, 4, 6\}$.

- this corresponds to receiving updates from nodes lower in the subtree rooted at $j$
- the filled graph is necessary to determine the structure of each row
In the right-looking method, after variable \( k \) is eliminated, column \( k \) is used to update all the columns corresponding to nonzeros in column \( k \). In the example above, \( \text{struct}(L(4:9,3)) = \{7, 8, 9\} \).

- this corresponds to sending updates to nodes higher in the elimination tree
- the filled graph is necessary to determine the structure of each column
The Multifrontal Method

Each node of the elimination tree is associated with a dense front.
The tree is traversed bottom-up and at each node:
1. assembly of the front
2. partial factorization
The Multifrontal Method

Each node of the elimination tree is associated with a dense front. The tree is traversed bottom-up and at each node:

1. assembly of the front
2. partial factorization

\[
\begin{pmatrix}
  a_{44} & a_{46} & a_{47} \\
  a_{64} & 0 & 0 \\
  a_{74} & 0 & 0
\end{pmatrix} \rightarrow
\begin{pmatrix}
  l_{44} & l_{64} & b_{66} & b_{67} \\
  l_{74} & l_{76} & b_{76} & b_{77}
\end{pmatrix}
\]
The Multifrontal Method

Each node of the elimination tree is associated with a dense **front**. The tree is traversed **bottom-up** and at each node:

1. assembly of the front
2. partial factorization

\[
\begin{pmatrix}
a_{55} & a_{56} & a_{59} \\
a_{65} & 0 & 0 \\
a_{95} & 0 & 0
\end{pmatrix} \rightarrow \begin{pmatrix}
l_{55} \\
l_{65} & c_{66} & c_{69} \\
l_{95} & c_{96} & c_{99}
\end{pmatrix}
\]
The Multifrontal Method

Each node of the elimination tree is associated with a dense **front**
The tree is traversed **bottom-up** and at each node
1. assembly of the front
2. partial factorization

\[
\begin{pmatrix}
  a_{66} & 0 & a_{68} & 0 \\
  0 & 0 & 0 & 0 \\
  a_{86} & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
\end{pmatrix}
+ \begin{pmatrix}
  b_{66} & b_{67} & 0 & 0 \\
  b_{76} & b_{77} & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
\end{pmatrix}
+ \begin{pmatrix}
  c_{66} & 0 & 0 & c_{69} \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  c_{96} & 0 & 0 & c_{99} \\
\end{pmatrix}
\rightarrow \begin{pmatrix}
  l_{66} & d_{77} & d_{78} & d_{79} \\
  l_{76} & d_{87} & d_{88} & d_{89} \\
  l_{86} & d_{97} & d_{98} & d_{99} \\
\end{pmatrix}
\]
A dense matrix, called **frontal matrix**, is associated at each node of the elimination tree. The **Multifrontal** method consists in a bottom-up traversal of the tree where at each node two operations are done:

- **Assembly** Nonzeros from the original matrix are assembled together with the contribution blocks from children nodes into the frontal matrix

\[
\begin{array}{ccc}
\text{arrowhead} & + & \text{cb-1} \\
+ & \ldots & + \\
& & \text{cb-n}
\end{array}
\]

- **Elimination** A partial factorization of the frontal matrix is done. The variable associated to the node of the frontal tree (called **fully assembled**) can be eliminated. This step produces part of the final factors and a Schur complement (contribution block) that will be assembled into the father node

(frontal matrix)  \[\rightarrow \]  \[
\begin{array}{c}
\end{array}
\]  \[\rightarrow \]  \[
\begin{array}{c}
\end{array}
\]

\[\text{cb}\]
The Multifrontal Method: example
Solve
Once the matrix is factorized, the problem can be solved against one or more right-hand sides:

\[ AX = LL^T X = B, \quad A, L \in \mathbb{R}^{n \times n}, \quad X \in \mathbb{R}^{n \times k}, \quad B \in \mathbb{R}^{n \times k} \]

The solution of this problem can be achieved in two steps:

**forward substitution** \( LZ = B \)

**backward substitution** \( L^T X = Z \)
Solve: left-looking

Forward Substitution

Backward Substitution
Solve: right-looking

Forward Substitution

Backward Substitution
The solution of a sparse linear system can be achieved in three phases that include at least the following operations:

**Analysis**
- Fill-reducing permutation
- Symbolic factorization
- Elimination tree computation

**Factorization**
- The actual matrix factorization

**Solve**
- Forward substitution
- Backward substitution

These phases, especially the analysis, can include many other operations. Some of them are presented next.
Parallelism
Parallelization: two levels of parallelism

**tree parallelism** arising from sparsity, it is formalized by the fact that nodes in separate subtrees of the elimination tree can be eliminated at the same time.

**node parallelism** within each node: parallel dense $LU$ factorization (BLAS)
Exploiting the second level of parallelism is crucial.

<table>
<thead>
<tr>
<th>Computer</th>
<th>#procs</th>
<th>MFlops (speed-up)</th>
<th>MFlops (speed-up)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alliant FX/80</td>
<td>8</td>
<td>15 (1.9)</td>
<td>34 (4.3)</td>
</tr>
<tr>
<td>IBM 3090J/6VF</td>
<td>6</td>
<td>126 (2.1)</td>
<td>227 (3.8)</td>
</tr>
<tr>
<td>CRAY-2</td>
<td>4</td>
<td>316 (1.8)</td>
<td>404 (2.3)</td>
</tr>
<tr>
<td>CRAY Y-MP</td>
<td>6</td>
<td>529 (2.3)</td>
<td>1119 (4.8)</td>
</tr>
</tbody>
</table>

Performance summary of the multifrontal factorization on matrix BCSSTK15. In column (1), we exploit only parallelism from the tree. In column (2), we combine the two levels of parallelism.
Task mapping and scheduling

Task mapping and scheduling: objective

Organize work to achieve a goal like makespan (total execution time) minimization, memory minimization or a mixture of the two:

- **Mapping**: where to execute a task?
- **Scheduling**: when and where to execute a task?

several approaches are possible:
- **static**: Build the schedule before the execution and follow it at run-time
  - **Advantage**: very efficient since it has a global view of the system
  - **Drawback**: Requires a very-good performance model for the platform
- **dynamic**: Take scheduling decisions dynamically at run-time
  - **Advantage**: Reactive to the evolution of the platform and easy to use on several platforms
  - **Drawback**: Decisions taken with local criteria (a decision which seems to be good at time $t$ can have very bad consequences at time $t + 1$)
- **hybrid**: Try to combine the advantages of static and dynamic
The mapping/scheduling is commonly guided by a number of parameters:

- First of all, a mapping/scheduling which is good for concurrency is commonly not good in terms of memory consumption.

- Especially in distributed-memory environments, data transfers have to be limited as much as possible.
Dynamic scheduling on shared memory computers

The data can be shared between processors without any communication*, therefore a dynamic scheduling is relatively simple to implement and efficient:

- The dynamic scheduling can be done through a pool of “ready” tasks (a task can be, for example, seen as the assembly and factorization of a front)
- as soon as a task becomes ready (i.e., when all the tasks associated with the child nodes are done), it is queued in the pool
- processes pick up ready tasks from the pool and execute them. As soon as one process finishes a task, it goes back to the pool to pick up a new one and this continues until all the nodes of the tree are done
- the difficulty lies in choosing one task among all those in the pool because the order can influence both performance and memory consumption
Decomposition of the tree into levels

- Determination of Level $L_0$ based on subtree cost.

- Scheduling of top of the tree can be dynamic.
Simplifying the mapping/scheduling problem

The tree commonly contains many more nodes than the available processes.

**Objective**: Find a layer $L_0$ such that subtrees of $L_0$ can be mapped onto the processor with a good balance.

**Construction and mapping of the initial level $L_0$**

Let $L_0 \leftarrow$ Roots of the assembly tree

repeat

Find the node $q$ in $L_0$ whose subtree has largest computational cost

Set $L_0 \leftarrow (L_0 \setminus \{q\}) \cup \{children\ \text{of}\ q\}$

Greedy mapping of the nodes of $L_0$ onto the processors

Estimate the load unbalance

**until** load unbalance $< \text{ threshold}$
Static scheduling on distributed memory computers

Main objective: reduce the volume of communication between processors because data is transferred through the (slow) network.

Proportional mapping

- initially assigns all processes to root node.
- performs a top-down traversal of the tree where the processes assign to a node are subdivided among its children in a way that is proportional to their relative weight.

Good at localizing communication but not so easy if no overlapping between processor partitions at each step.
Assumptions and Notations

- **Assumptions** :
  - We assume that each column of $L$ / each node of the tree is assigned to a single processor.
  - Each processor is in charge of computing $\text{cdiv}(j)$ for columns $j$ that it owns.

- **Notation** :
  - $\text{mycols}(p)$ is the set of columns owned by processor $p$.
  - $\text{map}(j)$ gives the processor owning column $j$ (or task $j$).
  - $\text{procs}(L(:,k)) = \{\text{map}(j) \mid j \in \text{struct}(L(:,k))\}$ (only processors in $\text{procs}(L(:,k))$ require updates from column $k$ – they correspond to ancestors of $k$ in the tree).
  - $\text{father}(j)$ is the father of node $j$ in the elimination tree.
The parallel algorithm is characterized by:

- Computational graph dependency
- Communication graph

There are three classical approaches to distributed memory parallelism:

1. **Fan-in**: The fan-in algorithm is very similar to the left-looking approach and is demand-driven: data required are aggregated update columns computed by sending processor.

2. **Fan-out**: The fan-out algorithms is very similar to the right-looking approach and is data driven: data is sent as soon as it is produced.

3. **Multifrontal**: The communication pattern follows a bottom-up traversal of the tree. Messages are contribution blocks and are sent to the processor mapped on the father node.
for $j=1$ to $n$
    $u=0$
    for all $k$ in $(\text{struct}(L(j,1:j-1)) \cap \text{mycols}(p))$
        $\text{cmod}(u,k)$
    end for
    if $\text{map}(j) \neq p$
        send $u$ to processor $\text{map}(j)$
    else
        incorporate $u$ in column $j$
        receive all the updates on column $j$ and incorporate them
    end if
end for
Fan-in variant

∀ i ∈ children map(i) = P₀ and map(father) ≠ P₀ (only) one message sent by P₀ → exploits data locality of proportional mapping.
Fan-in variant

if \( \forall i \in \text{children} \map(i) = P_0 \) and \( \map(father) \neq P_0 \) (only) one message sent by \( P_0 \rightarrow \) exploits data locality of proportional mapping.
if \( \forall i \in \text{children} \ map(i) = P_0 \) and \( \text{map}(\text{father}) \neq P_0 \) (only) one message sent by \( P_0 \rightarrow \) exploits data locality of proportional mapping.
Fan-in variant

\[
\forall i \in \text{children} \quad \text{map}(i) = P_0 \quad \text{and} \quad \text{map}(\text{father}) \neq P_0 \\
\quad \text{(only) one message sent by } P_0 \rightarrow \text{exploits data locality of proportional mapping.}
\]
Fan-in variant

\[
\text{if } \forall i \in \text{children} \map(i) = P_0 \text{ and } \map(\text{father}) \neq P_0 \text{ (only) one message sent by } P_0 \rightarrow \text{exploits data locality of proportional mapping.}
\]
Fan-in variant

∀ \( i \in \text{children} \) \( \text{map}(i) = P_0 \) and \( \text{map}(\text{father}) \neq P_0 \) (only) one message sent by \( P_0 \rightarrow \) exploits data locality of proportional mapping.
\[ \forall i \in \text{children} \, \text{map}(i) = P0 \text{ and } \text{map}(\text{father}) \neq P0 \text{ (only) one message sent by } P0 \rightarrow \text{exploits data locality of proportional mapping.} \]
Fan-out variant (similar to right-looking)

```plaintext
for all leaf nodes j in mycols(p)
    cdiv(j)
    send column L(:,j) to procs(L(:,j))
    mycols(p) = mycols(p) - {j}
end for

while mycols(p) != ∅
    receive any column (say L(:,k))
    for j in struct(L(:,k)) ∩ mycols(p)
        cmod(j,k)
        if column j is completely updated
            cdiv(j)
            send column L(:,j) to procs(L(:,j))
            mycols(p) = mycols(p) - {j}
        end if
    end for
end while
```
Fan-out variant

if $\forall i \in \text{children} \ \map(i) = P_0$ and $\map(father) \neq P_0$ then $n$ messages (where $n$ is the number of children) are sent by $P_0$ to update the processor in charge of the father.
if $\forall i \in \text{children} \; \text{map}(i) = P0$ and $\text{map}(\text{father}) \neq P0$ then $n$ messages (where $n$ is the number of children) are sent by $P0$ to update the processor in charge of the father.
if $\forall i \in \text{children} \ map(i) = P0$ and $map(father) \neq P0$ then $n$ messages (where $n$ is the number of children) are sent by $P0$ to update the processor in charge of the father
if $\forall i \in \text{children} \ map(i) = P0 \text{ and } map(\text{father}) \neq P0$ then $n$ messages (where $n$ is the number of children) are sent by $P0$ to update the processor in charge of the father
if $\forall i \in children \; map(i) = P0$ and $map(father) \neq P0$ then $n$ messages (where $n$ is the number of children) are sent by $P0$ to update the processor in charge of the father.
Properties of fan-out:

- Historically the first implemented.
- Incurs greater interprocessor communications than fan-in (or multifrontal) approach both in terms of:
  - total number of messages
  - total volume
- Does not exploit data locality of proportional mapping.
- Improved algorithm (local aggregation):
  - send aggregated update columns instead of individual factor columns for columns mapped on a single processor.
  - Improve exploitation of data locality of proportional mapping.
  - But memory increase to store aggregates can be critical (as in fan-in).
for all leaf nodes j in mycols(p)
    assemble front j
    partially factorize front j
    send the schur complement to procs(father(j))
    mycols(p) = mycols(p) - {j}
end for
while mycols(p) != ∅
    receive any contribution block (say for node j)
    assemble contribution block into front j
    if front j is completely assembled
        partially factorize front j
        send the schur complement to procs(father(j))
        mycols(p) = mycols(p) - {j}
    end if
end while
Multifrontal variant

Fan-in

Fan-out

Multifrontal
Multifrontal variant

Fan-in

Fan-out

Multifrontal
Multifrontal variant

Fan-in

Fan-out

Multifrontal
Multifrontal variant

Fan-in

Fan-out

Multifrontal
Importance of the shape of the tree

Suppose that each node in the tree corresponds to a task that:

• consumes temporary data from the children,
• produces temporary data, that is passed to the parent node.

• Wide tree
  ◦ Good parallelism
  ◦ Many temporary blocks to store
  ◦ Large memory usage

• Deep tree
  ◦ Less parallelism
  ◦ Smaller memory usage
## Impact of fill reduction on the shape of the tree

<table>
<thead>
<tr>
<th>Reordering technique</th>
<th>Shape of the tree</th>
<th>observations</th>
</tr>
</thead>
</table>
| AMD                  | ![Tree Diagram](image) | • Deep well-balanced  
                           • Large frontal matrices on top |
| AMF                  | ![Tree Diagram](image) | • Very deep unbalanced  
                           • Small frontal matrices |
| **PORD** | ![Diagram of deep unbalanced tree] | - deep unbalanced  
- Small frontal matrices |
|----------|----------------------------------|------------------|
| **SCOTCH** | ![Diagram of very wide well-balanced tree] | - Very wide well-balanced  
- Large frontal matrices |
| **METIS** | ![Diagram of wide well-balanced tree] | - Wide well-balanced  
- Smaller frontal matrices (than SCOTCH) |
Memory consumption in the multifrontal method
A tree can be regarded as a dependency graph where nodes produce data that is then processed by its parent. By this definition, a tree has to be traversed in topological order. But there are still many topological traversals of the same tree.

**Definition**

Two orderings $P$ and $Q$ are equivalent if the structures of the filled graphs of $PAP^T$ and $QAQ^T$ are the same (that is they are isomorphic).

Equivalent orderings result in the same amount of fill-in and computation during factorization. To ease the notation, we discuss only one ordering wrt $A$, i.e., $P$ is an equivalent ordering of $A$ if the filled graph of $A$ and that of $PAP^T$ are isomorphic.
Any topological ordering on $T(A)$ are equivalent

Let $P$ be the permutation matrix corresponding to a topological ordering of $T(A)$. Then, $G^+(PAP^T)$ and $G^+(A)$ are isomorphic.

Any topological ordering on $T(A)$ are equivalent

Let $P$ be the permutation matrix corresponding to a topological ordering of $T(A)$. The elimination tree $T(PAP^T)$ and $T(A)$ are isomorphic.

Because the fill-in won’t change, we have the freedom to choose any specific topological order that will provide other properties
Tree traversal orders

Which specific topological order to choose? **postorder**: why?
Because data produced by nodes is consumed by parents in a **LIFO** order.
In the multifrontal method, we can thus use a stack memory where contribution blocks are **pushed** as soon as they are produced by the elimination on a frontal matrix and **popped** at the moment where the father node is assembled.
This provides a better **data locality** and makes the memory management easier.
The multifrontal method [Duff & Reid '83]

Storage is divided into two parts:
- Factors
- Active memory

Factors
Active frontal matrix
Stack of contribution blocks

Active storage

Elimination tree
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid '83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory

Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ‘83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory

Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory

Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

A = L + U - I

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:
- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory
The multifrontal method [Duff & Reid ’83]

Storage is divided into two parts:

- Factors
- Active memory

Active memory
Storage is divided into two parts:

- Factors
- Active memory
Example 1: Processing a wide tree

Memory

unused memory space
stack memory space
factor memory space
non-free memory space
Example 2: Processing a deep tree

Memory

Allocation of 3

Assembly step for 3

Factorization step for 3

Stack step for 3

Legend:
- unused memory space
- stack memory space
- factor memory space
- non-free memory space
Postorder provides a good data locality and better memory consumption that a general topological order since father nodes are assembled as soon as its children have been processed. But there are still many postorders of the same tree. Which one to choose? the one that **minimizes memory consumption**

**Best (abcdefghi)**

```
  i
 / \
 g  e
 / \  / \
 c  a  d  f
```

**Worst (hfdbacegi)**

```
  i
 / \
 g  e
 / \  / \
 c  a  d  b
```

**Leaves**

```
  a b d f h
```

**Root**

```
  h f d b a
```
Problem model

- $M_i$: memory peak for complete subtree rooted at $i$,
- $temp_i$: temporary memory produced by node $i$,
- $m_{parent}$: memory for storing the parent.

\[
M_{parent} = \max\left( \max_{j=1}^{\text{nbchildren}} (M_j + \sum_{k=1}^{j-1} temp_k),
\ m_{parent} + \sum_{j=1}^{\text{nbchildren}} temp_j \right)
\]
Problem model

- $M_i$: memory peak for complete subtree rooted at $i$,
- $\text{temp}_i$: temporary memory produced by node $i$,
- $m_{\text{parent}}$: memory for storing the parent.

$$M_{\text{parent}} = \max(\max_{j=1}^{\text{nbchildren}} (M_j + \sum_{k=1}^{j-1} \text{temp}_k), m_{\text{parent}} + \sum_{j=1}^{\text{nbchildren}} \text{temp}_j)$$

(1)

**Objective:** order the children to minimize $M_{\text{parent}}$
### Theorem

[Liu,86] The minimum of \( \max_j(x_j + \sum_{i=1}^{j-1} y_i) \) is obtained when the sequence \((x_i, y_i)\) is sorted in decreasing order of \(x_i - y_i\).

### Corollary

An optimal child sequence is obtained by rearranging the children nodes in decreasing order of \(M_i - temp_i\).

**Interpretation:** At each level of the tree, child with relatively large peak of memory in its subtree (\(M_i\) large with respect to \(temp_i\)) should be processed first.

⇒ Apply on complete tree starting from the leaves (or from the root with a recursive approach)
Objective: Minimize peak of stack memory

**Tree_Reorder** \((T)\):
   \[
   \text{for all } i \text{ in the set of root nodes do}
   \]
   \[
   \text{Process\_Node}(i);
   \]
   \[
   \text{end for}
   \]

**Process\_Node**(\(i\)):
   \[
   \text{if } i \text{ is a leaf then}
   \]
   \[
   M_i = m_i
   \]
   \[
   \text{else}
   \]
   \[
   \text{for } j = 1 \text{ to } nbchildren \text{ do}
   \]
   \[
   \text{Process\_Node}(j^{th} \text{ child});
   \]
   \[
   \text{end for}
   \]
   \[
   \text{Reorder the children of } i \text{ in decreasing order of } (M_j - temp_j);
   \]
   \[
   \text{Compute } M_{parent} \text{ at node } i \text{ using Formula (1)};
   \]
   \[
   \text{end if}
   \]
Memory consumption in parallel

In parallel multiple branches have to be traversed at the same time in order to feed all the processes. This means that an higher number of CBs will have to be stored in memory.

Metric: memory efficiency

\[ e(p) = \frac{S_{seq}}{p \times S_{max}(p)} \]

We would like \( e(p) \approx 1 \), i.e. \( S_{seq}/p \) on each processor.

Common mappings/schedulings provide a poor memory efficiency:

- Proportional mapping: \( \lim_{p \to \infty} e(p) = 0 \) on regular problems.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.
Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.

- Targets run time but poor memory efficiency.
- Usually a relaxed version is used: more memory-friendly but unreliable estimates.
Proportional mapping: assuming that the sequential peak is 5 GB,

\[ S_{\text{max}}(p) \geq \max \left\{ \frac{4 \text{ GB}}{26}, \frac{1 \text{ GB}}{6}, \frac{5 \text{ GB}}{32} \right\} = 0.16 \text{ GB} \Rightarrow e(p) \leq \frac{5}{64 \times 0.16} \leq 0.5 \]
A more memory-friendly strategy...

All-to-all mapping: postorder traversal of the tree, where all the processors work at every node:

Optimal memory scalability \( S_{max}(p) = S_{seq}/p \) but no tree parallelism and prohibitive amounts of communications.
"Memory-aware" mapping (Agullo '08): aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

1. Try to apply proportional mapping.
“Memory-aware” mappings (Agullo '08): aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

1. Try to apply proportional mapping.
2. Enough memory for each subtree?

- Ensures the given memory constraint and provides reliable estimates.
- Tends to assign many processors on nodes at the top of the tree $\Rightarrow$ performance issues on parallel nodes.
“Memory-aware” mapping (Agullo ’08): aims at enforcing a given memory constraint \(M_0\), maximum memory per processor):

1. Try to apply proportional mapping.
2. Enough memory for each subtree? If not, serialize them, and update \(M_0\): processors stack equal shares of the CBs from previous nodes.
“Memory-aware” mapping (Agullo '08): aims at enforcing a given memory constraint ($M_0$, maximum memory per processor):

1. Try to apply proportional mapping.
2. Enough memory for each subtree?

- Ensures the given memory constraint and provides reliable estimates.
- Tends to assign many processors on nodes at the top of the tree → performance issues on parallel nodes.
subroutine ma_mapping(r, p, M, S)
! r : root of the subtree
! p(r) : number of processes mapped on r
! M(r) : memory constraint on r
! S( :) : peak memory consumption for all nodes

call prop_mapping(r, p, M)

forall (i children of r)
! check if the constraint is respected
if( S(i)/p(i) > M(i) ) then
! reject PM and revert to tree serialization
  call tree_serialization(r, p, M)
  exit ! from forall block
end if
end forall

forall (i children of r)
! apply MA mapping recursively to all siblings
  call ma_mapping(i, p, M, S)
end forall
end subroutine ma_mapping
**“memory-aware” mappings**

```plaintext
subroutine prop_mapping(r, p, M)
    forall (i children of r)
        p(i)  = share of the p(r) processes from PM
        M(i)  = M(r)
    end forall
end subroutine prop_mapping

subroutine tree_serialization(r, p, M)
    stack_siblings = 0
    forall (i children of r) ! in appropriate order
        p(i) = p(r)
        ! update the memory constraint for the subtree
        M(i) = M(r) - stack_siblings
        stack_siblings = stack_siblings + cb(i)/p(r)
    end forall
end subroutine tree_serialization
```
A finer “memory-aware” mapping? Serializing all the children at once is very constraining: more tree parallelism can be found.

Find groups on which proportional mapping works, and serialize these groups.

Heuristic: follow a given order (e.g. the serial postorder) and form groups as large as possible.
Experiments

- **Matrix**: finite-difference model of acoustic wave propagation, 27-point stencil, $192 \times 192 \times 192$ grid; Seiscope consortium. $N = 7 \text{ M}$, $nnz = 189 \text{ M}$, factors=152 GB (METIS). Sequential peak of active memory: 46 GB.

- **Machine**: 64 nodes with two quad-core Xeon X5560 per node. We use 256 MPI processes.

- **Perfect memory scalability**: $46 \text{ GB}/256 = 180\text{MB}$. 
### Experiments

<table>
<thead>
<tr>
<th>Map</th>
<th>Memory-aware mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_0 = 225$ MB</td>
<td>$M_0 = 380$ MB</td>
</tr>
<tr>
<td>w/o groups</td>
<td>w/ groups</td>
</tr>
<tr>
<td>Max stack peak (MB)</td>
<td>1932</td>
</tr>
<tr>
<td>Avg stack peak (MB)</td>
<td>626</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1323</td>
</tr>
</tbody>
</table>

- Proportional mapping delivers the fastest factorization because it targets parallelism but, at the same time, it leads to a poor memory scaling with $e(256) = 0.09$.
- The memory aware mapping with and without groups respects the imposed memory constraint with a speed that depends on how tight the constraint is.
- Grouping clearly provides a benefit in term of speed because it delivers more concurrency while still respecting the constraint.
Low-Rank approximations
Low-rank matrices

Take a dense matrix $B$ of size $n \times n$ and compute its SVD $B = XSY$: 

$$
B = XSY
$$
Low-rank matrices

Take a dense matrix $B$ of size $n \times n$ and compute its SVD $B = XSY$:

\[
B = X_1 S_1 Y_1 + X_2 S_2 Y_2 \quad \text{with} \quad S_1(k, k) = \sigma_k > \varepsilon, \quad S_2(1, 1) = \sigma_{k+1} \leq \varepsilon
\]
Low-rank matrices

Take a dense matrix $B$ of size $n \times n$ and compute its SVD $B = XSY$:

$$B = X_1 S_1 Y_1 + X_2 S_2 Y_2 \quad \text{with} \quad S_1(k, k) = \sigma_k > \varepsilon, \quad S_2(1, 1) = \sigma_{k+1} \leq \varepsilon$$

If $\tilde{B} = X_1 S_1 Y_1$ then

$$\|B - \tilde{B}\|_2 = \|X_2 S_2 Y_2\|_2 = \sigma_{k+1} \leq \varepsilon$$
Low-rank matrices

Take a dense matrix $B$ of size $n \times n$ and compute its SVD $B = XSY$:

$$B = X_1 S_1 Y_1 + X_2 S_2 Y_2 \text{ with } S_1(k, k) = \sigma_k > \varepsilon, \ S_2(1, 1) = \sigma_{k+1} \leq \varepsilon$$

If $\tilde{B} = X_1 S_1 Y_1$ then

$$\|B - \tilde{B}\|_2 = \|X_2 S_2 Y_2\|_2 = \sigma_{k+1} \leq \varepsilon$$

If the singular values of $B$ decay very fast (e.g. exponentially) then $k \ll n$ even for very small $\varepsilon$ (e.g. $10^{-14}$) $\Rightarrow$ memory and CPU consumption can be reduced considerably with a controlled loss of accuracy ($\leq \varepsilon$) if $\tilde{B}$ is used instead of $B$
**Frontal** matrices are not low-rank but in many applications they exhibit low-rank blocks.

Because each row and column of a frontal matrix are associated with a point of the discretization mesh, blocking the frontal matrix amounts to grouping the corresponding points of the domain. The objective is thus to group (cluster) the points in such a way that the number of blocks with low-rank property is maximized and their respective rank is minimized.
Clustering of points/variables

A block represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away the interaction is weak $\Rightarrow$ rank is low.

Guidelines for computing the grouping of points:

• $\forall \sigma, c_{\min} \geq |\sigma| \geq c_{\max}$: we want many compressible blocks but not too small to maintain a good compression rate and a good efficiency of operations.

• $\forall \sigma, \minimize \text{diam}(\sigma)$: for a given cluster size, this gives well-shaped clusters and consequently maximizes mutual distances.
A block represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away the interaction is weak $\Rightarrow$ rank is low.
Clustering of points/variables

A block represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away the interaction is weak $\Rightarrow$ rank is low.

\[\begin{array}{cccc}
\sigma & \tau \\
\end{array}\]

\[\begin{array}{ccccccccc}
0 & & & & & & & & 1 \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
\end{array}\]

Graph:

- x-axis: distance between $\tau$ and $\sigma$
- y-axis: rank of $A_{\tau}$

Graph shows that as the distance increases, the rank decreases, indicating a weaker interaction between the subdomains.
Clustering of points/variables

A block represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away the interaction is weak $\Rightarrow$ rank is low

Guidelines for computing the grouping of points:

- $\forall \sigma, c_{\text{min}} \geq |\sigma| \geq c_{\text{max}}$: we want many compressible blocks but not too small to maintain a good compression rate and a good efficiency of operations.

- $\forall \sigma$, minimize $\text{diam}(\sigma)$: for a given cluster size, this gives well-shaped clusters and consequently maximizes mutual distances.
A block represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away the interaction is weak $\Rightarrow$ rank is low
A block represents the interaction between two subdomains $\sigma$ and $\tau$. If they have a small diameter and are far away the interaction is weak $\Rightarrow$ rank is low.

Guidelines for computing the grouping of points:

- $\forall \sigma$, $c_{\min} \geq |\sigma| \geq c_{\max}$: we want many compressible blocks but not too small to maintain a good compression rate and a good efficiency of operations.
- $\forall \sigma$, minimize $diam(\sigma)$: for a given cluster size, this gives well shaped clusters and consequently maximizes mutual distances.
Clustering

For example, if the points associated to the rows/columns of a frontal matrix lie on a square surface, it is much better to cluster in a checkerboard fashion (right), rather than into slices (left).

In practice and in an algebraic context (where the discretization mesh is not known) this can be achieved by extracting the subgraph associated with the front variables and by computing a k-way partitioning (with a tool like METIS or SCOTCH, for example).
Here is an example of variables clustering achieved with the x-way partitioning method in the SCOTCH package.

Although irregular, the partitioning provides relatively well-shaped clusters.
Once the clustering of variables is defined, it is possible to compress all the blocks $F(\sigma, \tau)$ that satisfy a (strong) admissibility condition:

$$ Adm_s(\sigma, \tau) : \max(diam(\sigma), diam(\tau)) \leq \eta \ dist(\sigma, \tau) $$

For simplicity, we can use a relaxed version called weak admissibility condition:

$$ Adm_w(\sigma, \tau) : \ dist(\sigma, \tau) > 0. $$

The far field of a cluster $\sigma$ is defined as

$$ F(\sigma) = \{ \tau \mid Adm(\sigma, \tau) \text{ is true} \}. $$

In practice it is easier to consider all block (except diagonal) eligible, try to compress them and eventually revert to non compressed form if rank is too high.
Dense BLR LU factorization

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$

- FACTOR
- SOLVE
- UPDATE
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR** $A_{kk} = L_{kk}U_{kk}$
- **SOLVE**
- **UPDATE**
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$

- FACTOR
- **SOLVE** $U_{kj} = L_{kk}^{-1}A_{kj}$, $L_{ik} = A_{ik}U_{kk}^{-1}$
- UPDATE
Dense BLR LU factorization

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>U</th>
<th>L</th>
<th>U</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **UPDATE** $A_{ij} = A_{ij} - L_{ik}U_{kj}$
Dense BLR LU factorization

<table>
<thead>
<tr>
<th>L</th>
<th>U</th>
<th>U</th>
<th>U</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **UPDATE**
Dense BLR LU factorization

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>U</th>
<th>U</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **UPDATE**
Dense BLR LU factorization

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>U</th>
<th>U</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td></td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **UPDATE**
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **UPDATE**
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
Dense BLR LU factorization

off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$

- FACTOR
- SOLVE
- UPDATE
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- UPDATE
Dense BLR LU factorization

off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- UPDATE
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
Dense BLR LU factorization

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **COMPRESS** $L_{*k} = X_{*k}Y_{*k}^T$, $U_{k*} = X_{k*}Y_{k*}^T$
- **UPDATE**
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- COMPRESS
- UPDATE
Dense BLR LU factorization

<table>
<thead>
<tr>
<th>L</th>
<th>U</th>
<th>U</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- COMPRESS
- UPDATE
Dense BLR LU factorization

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>U</td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **COMPRESS**
- **UPDATE**
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- COMPRESS
- UPDATE
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
Dense BLR LU factorization

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- **FACTOR**
- **SOLVE**
- **COMPRESS**
- **UPDATE**
Dense BLR LU factorization

off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$

- FACTOR
- SOLVE
- COMPRESS
- UPDATE
Dense BLR LU factorization

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}$, $U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- COMPRESS
- UPDATE
off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.
Dense BLR LU factorization

Off-diagonal blocks are compressed in low-rank form right before updating the trailing submatrix. As a result, updating a block $A_{ij}$ of size $b$ using blocks $L_{ik}, U_{kj}$ of rank $r$ only takes $O(br^2)$ operations instead of $O(b^3)$.

- FACTOR
- SOLVE
- COMPRESS
- UPDATE
Setting:

1. **Poisson**: $N^3$ grid with a 7-point stencil with $u = 1$ on the boundary $\partial \Omega$

   $$\Delta u = f$$

2. **Helmholtz**: $N^3$ grid with a 27-point stencil, $\omega$ is the angular frequency, $v(x)$ is the seismic velocity field, and $u(x, \omega)$ is the time-harmonic wavefield solution to the forcing term $s(x, \omega)$.

   $$\left( -\Delta - \frac{\omega^2}{v(x)^2} \right) u(x, \omega) = s(x, \omega)$$
Experimental MF complexity: entries in factor

- \( \varepsilon \) only plays a role in the constant factor
- for Poisson a factor \( \sim 3 \) gain with almost no loss of accuracy
• $\varepsilon$ only plays a role in the constant factor
• for Poisson a factor $\sim 9$ gain with almost no loss of accuracy
Experiments: Poisson

Poisson Equation, 7pt on $128^3$ domain

- Compression improves with higher $\varepsilon$ values
- Time reduction is not as good as flops due to lower efficiency of operations
- Componentwise Scaled Residual is in good accordance with $\varepsilon$
- Iterative Refinement can be used to recover the lost accuracy
- For very high values of $\varepsilon$, the factorization can be used as a preconditioner
• Compression improves with higher $\varepsilon$ values
• Time reduction is not as good as flops due to lower efficiency of operations
• Componentwise Scaled Residual is in good accordance with $\varepsilon$
• Iterative Refinement can be used to recover the lost accuracy
• For very high values of $\varepsilon$, the factorization can be used as a preconditioner
Sparse QR Factorization
In the case $Ax = b$ with $A \in \mathbb{R}^{m \times n}, \quad m > n$, the problem is overdetermined and the existence of a solution is not guaranteed (the solution only exists if $b \in \mathcal{R}(A)$). In this case, the desired solution may be the one that minimizes the norm of the residual (least squares problem):

$$\min_x \|Ax - b\|_2$$

The figure above shows, intuitively, that the residual has minimum norm if it is orthogonal to the $\mathcal{R}(A)$, i.e., $Ax$ is the projection of $b$ onto $\mathcal{R}(A)$.
Least Squares Problems: normal equations

\[ \min_x \| Ax - b \|_2 \Leftrightarrow \min_x \| Ax - b \|_2^2 \]

\[ \| Ax - b \|_2^2 = (Ax - b)^T (Ax - b) = x^T A^T Ax - 2x^T A^T b + b^T b \]

First \( x \)-derivative is equal to zero is \( x = (A^T A)^{-1} A^T b \). This is the solution of the \( n \times n \) fully determined problem:

\[ A^T Ax = A^T b \]

called the normal equations system:

- \( A^T A \) is symmetric, positive definite if \( \text{rank}(A) = n \) (nonnegative definite otherwise)

- can be solved with a Cholesky factorization. Not necessarily backward stable because \( \kappa(A^T A) = \kappa(A)^2 \)

- \( A^T (Ax - b) = A^T r = 0 \) means that the residual is orthogonal to \( \mathcal{R}(A) \) which delivers the desired solution
The **QR factorization** of a matrix $A \in \mathbb{R}^{m \times n}$ consists in finding an orthonormal matrix $Q \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $R \in \mathbb{R}^{n \times n}$ such that

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$$

Assuming $Q = [Q_1 Q_2]$ where $Q_1$ is made up of the first $n$ columns of $Q$ and $Q_2$ by the remaining $m - n$

$$Q^T b = \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} b = \begin{bmatrix} c \\ d \end{bmatrix}$$

we have

$$\|Ax - b\|_2^2 = \|Q^T Ax - Q^T b\|_2^2 = \|Rx - c\|_2^2 + \|d\|_2^2.$$ 

In the case of a full-rank matrix $A$, this quantity is minimized if $Rx = c$. 
Least Squares Problems: the QR factorization

Assume $Q = [Q_1 Q_2]$ where $Q_1$ contains the first $n$ and $Q_2$ the last $m - n$ columns of $Q$:

- $Q_1$ is an orthogonal basis for $\mathcal{R}(A)$ and $Q_2$ is an orthogonal basis for $\mathcal{N}(A^T)$
- $P_1 = Q_1 Q_1^T$ and $P_2 = Q_2 Q_2^T$ are projectors onto $\mathcal{R}(A)$ and $\mathcal{N}(A^T)$. Thus, $Ax = P_1 b$ and $r = b - Ax = P_2 b$

Properties of the solution by QR factorization

- it is **backward stable**
- it requires $3n^2 (m - n/3)$ flops (much more expensive than Cholesky)
Underdetermined Problems

In the case $Ax = b$ with $A \in \mathbb{R}^{m \times n}$, $m < n$, the problem is underdetermined and admits infinite solutions; if $x_p$ is any solution of $Ax = b$, then the complete set of solutions is

$$\{x | Ax = y\} = \{x_p + z | z \in \mathcal{N}(A)\}$$

In this case, the desired solution may be the one with minimum 2-norm (minimum 2-norm solution problem):

$$\begin{align*}
\text{minimize} & \quad \|x\|_2 \\
\text{subject to} & \quad Ax = b
\end{align*}$$
The minimum 2-norm solution is

\[ x_{mn} = A^T (AA^T)^{-1} b \]

In fact, take any other solution \( x \); then \( A(x - x_{mn}) = 0 \) and

\[
(x - x_{mn})^T x_{mn} = (x - x_{mn})^T A^T (AA^T)^{-1} b \\
= (A(x - x_{mn}))^T (AA^T)^{-1} b \\
= 0
\]

which means \((x - x_{mn})^T \perp x_{mn}\). Thus

\[
\|x\|_2^2 = \|x_{mn} + x - x_{mn}\|_2^2 = \|x_{mn}\|_2^2 + \|x - x_{mn}\|_2^2
\]
i.e., \( x_{mn} \) has the smallest 2-norm of any solution.

As for the normal equations system, this approach incurs instability due to \( \kappa(AA^T) \)
Assume

\[ QR = [Q_1Q_2] \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = A^T \]

is the QR factorization of \( A^T \) where \( Q_1 \) is made of the first \( m \) columns of \( Q \) and \( Q_2 \) of the last \( n - m \) ones. Then

\[ Ax = R^T Q^T x = [R_1^T 0] \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = b \]

The minimum 2-norm solution follows by setting \( z_2 = 0 \). Note that \( Q_2 \) is an orthogonal basis for \( \mathcal{N}(A) \) thus, the minimum 2-norm solution is achieved by removing from any solution \( x_p \) all its components in \( \mathcal{N}(A) \).
A matrix $H$ of the form

$$H = I - 2\frac{vv^T}{v^Tv}$$

is called a **Householder Reflection** and $v$ a **Householder Vector**. The transformation

$$P_v = \frac{vv^T}{v^Tv}$$

is a projection onto the space of $v$. 

![Diagram showing Householder reflection and projection](image)
A matrix $H$ of the form

$$H = I - 2 \frac{vv^T}{v^Tv}$$

is called a Householder Reflection and $v$ a Householder Vector. The transformation

$$P_v = \frac{vv^T}{v^Tv}$$

is a projection onto the space of $v$

Householder reflections can be used in such a way to annihilate all the coefficients of a vector $x$ except one which corresponds to reflecting $x$ on one of the original axes. If $v = x \pm \|x\|_2 e_1$ then $Hx$ has all coefficients equal to zero except the first.
Note that in practice the $H$ matrix is never explicitly formed. Instead, it is applied through products with the vector $v$. Also, note that if $x$ is sparse, the $v$ vector (and thus the $H$ matrix) follows its structure. Imagine a sparse $x$ vector and an $\tilde{x}$ one built removing all the zeros in $x$:

$$x = \begin{bmatrix} x_1 \\ 0 \\ x_2 \end{bmatrix}, \quad \tilde{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

If $\tilde{H}$ is the Householder reflector that annihilates all the elements of $\tilde{x}$ but the first, then the equivalent transformation $H$ for $X$ is

$$\tilde{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \quad \tilde{H} = \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix} \Rightarrow v = \begin{bmatrix} v_1 \\ 0 \\ v_2 \end{bmatrix}, \quad H = \begin{bmatrix} H_{11} & 0 & H_{12} \\ 0 & 1 & 0 \\ H_{21} & 0 & H_{22} \end{bmatrix}$$
The complete QR factorization of a matrix is achieved in $n$ steps:

$$H_1 = \left[ \begin{array} { c c c c } 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (I - 2 \frac{v_1 v_1^T}{v_1^T v_1}) \\ 0 & 0 & 0 & (I - 2 \frac{v_2 v_2^T}{v_2^T v_2}) \end{array} \right]$$

$$H_2 = \left[ \begin{array} { c c c c } 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (I - 2 \frac{v_3 v_3^T}{v_3^T v_3}) \\ 0 & 0 & 0 & (I - 2 \frac{v_4 v_4^T}{v_4^T v_4}) \end{array} \right]$$

$$H_3 = \left[ \begin{array} { c c c c } 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (I - 2 \frac{v_5 v_5^T}{v_5^T v_5}) \\ 0 & 0 & 0 & (I - 2 \frac{v_6 v_6^T}{v_6^T v_6}) \end{array} \right]$$

The Householder vectors can be stored in the bottom left corner of the original matrix where the zeros were introduced:

$$H_1 \cdot H_2 \cdot H_3 = Q$$
Depending on the position of the coefficients that have to be annihilated, the Householder transformation only touches certain regions of the matrix. Imagine a matrix

\[
A = \begin{bmatrix} B & D \\ 0 & E \end{bmatrix}
\]

\(Q_B R_B = B\) being the QR factorization of \(B\).

Transformations that modify non overlapping regions of a matrix can be applied independently and thus in parallel.
Depending on the position of the coefficients that have to be annihilated, the Householder transformation only touches certain regions of the matrix. Imagine a matrix

\[ A = \begin{bmatrix} B & D \\ 0 & E \end{bmatrix} \]

\[ Q_B R_B = B \] being the QR factorization of \( B \).

Transformations that modify non overlapping regions of a matrix can be applied independently and thus in parallel.
Sparse Multifrontal QR

Assume a pseudo-sparse matrix $A$ defined as below. Its QR factorization can be achieved in three steps:

**Step-1**

$$Q_B R_B = \begin{bmatrix} B \\ C \end{bmatrix}$$

$$A = \begin{bmatrix} B & F \\ C & G \\ D & H \\ E & L \\ M \end{bmatrix} \rightarrow A_1 = \begin{bmatrix} R_B & \tilde{F} \\ \tilde{G} & \tilde{H} \\ \tilde{L} & M \end{bmatrix}$$

**Step-2**

$$Q_D R_D = \begin{bmatrix} D \\ E \end{bmatrix}$$

$$A_1 \rightarrow A_2 = \begin{bmatrix} R_B & \tilde{F} \\ \tilde{G} & \tilde{H} \\ \tilde{L} & M \end{bmatrix} = P \begin{bmatrix} R_B & \tilde{F} \\ \tilde{D} & \tilde{H} \\ M & \tilde{G} \end{bmatrix}$$

**Step-3**

$$Q_M R_M = \begin{bmatrix} M \\ \tilde{G} \\ \tilde{L} \end{bmatrix}$$

$$A_2 \rightarrow R = \begin{bmatrix} R_B & \tilde{F} \\ R_D & \tilde{H} \\ R_M \end{bmatrix}$$
Sparse Multifrontal QR

We just did a **sparse, multifrontal QR factorization**. The previous operation can be arranged in the following elimination tree:

Most of the theory related to the multifrontal QR factorization can be formalized thanks to structural properties of the normal equation matrix.
After applying an Householder reflection (or a set of reflections) that affects $k$ rows of a sparse matrix, all the columns presenting at least one nonzero along those lines will be completely filled, i.e. will have a nonzero on each of those $k$ rows:
After applying an Householder reflection (or a set of reflections) that affects $k$ rows of a sparse matrix, all the columns presenting at least one nonzero along those lines will be completely filled, i.e. will have a nonzero on each of those $k$ rows:
The strong Hall property

Based on the relation $A^T A = (QR)^T QR = R^T R$, the $R$ factor of a matrix $A$ is equal to the Cholesky factor of the corresponding normal equation $A^T A$. The structure of the $R$ factor can be performed by a symbolic Cholesky factorization of the $A^T A$ matrix is $A$ is a Strong Hall matrix.

**Strong Hall**

A matrix $A \in \mathbb{R}^{m \times n}, m \geq n$ is said to be Strong Hall if every subset of $0 < k < m$ columns, the corresponding submatrix has nonzeros in at least $k + 1$ rows.

In this case, the symbolic Cholesky factorization of $A^T A$ will correctly predict the structure of $R$.

The symbolic Cholesky factorization would not predict structural cancellations that happen in the case where $A$ is not strong Hall.
The strong Hall property: two examples

**NON-Strong Hall**

\[
\begin{align*}
\text{struct}(A) & \quad \begin{bmatrix}
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\end{bmatrix} \\
\text{predicted struct}(R) & \quad \begin{bmatrix}
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\end{bmatrix} \\
\text{actual struct}(R) & \quad \begin{bmatrix}
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\end{align*}
\]

**Strong Hall**

\[
\begin{align*}
\text{struct}(A) & \quad \begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\end{bmatrix} \\
\text{predicted struct}(R) & \quad \begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\end{bmatrix} \\
\text{actual struct}(R) & \quad \begin{bmatrix}
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times \\
\end{bmatrix}
\end{align*}
\]
The strong Hall property

In the case of non-strong Hall matrices, two options are possible:

1. **Reduce the matrix to Block-Triangular Form (BTF).** In this case all the diagonal blocks have the strong Hall property and can be QR-factorized independently:

   ![Coarse BTF and Fine BTF](image)

2. **Rediscover the structure of $R$ during the factorization phase.** The symbolic Cholesky factorization of $A^T A$ overestimates the fill-in in $R$. This analysis can be thus taken as a starting point and can be refined during the numerical computation according to the actual fill-in.

Without loss of generality we can assume $A$ is always strong Hall.
The analysis phase of the sparse, multifrontal QR factorization proceeds in the same way as for the Cholesky or LU factorizations and includes the same operations:

- **fill-reducing ordering**: this operation aims at computing a column permutation that minimizes the amount of fill-in introduced in $R$ during the factorization. This is achieved by computing a fill-reducing order for the Cholesky factor of $A^T A$
- **symbolic factorization**: this operation computes, symbolically, the structure of the Cholesky factor of $A^T A$
- **Elimination tree computation**: computation of the elimination tree for the Cholesky factorization of $A^T A$
- **Tree amalgamation**
- **Tree traversal order**
- **etc.**
The numerical factorization phase also proceeds as in multifrontal LU or Cholesky (a bottom up traversal of the tree) but the factorization and assembly of a frontal matrix are different:

**assembly**: in the QR factorization the contribution blocks from children nodes are not summed but simply appended to coefficients from the original matrix. This can be done in two different ways depending on the factorization strategy (below)

**factorization**: the QR factorization of the frontal matrix can be either partial or full
Assume a frontal matrix of size $m \times n$ with $npiv$ fully assembled variables. In the first strategy, only $npiv$ out of $\min(m, n)$ Householder reflections are applied to the frontal matrix. This leads to a rectangular contribution block.
Assume a frontal matrix of size $m \times n$ with $npiv$ fully assembled variables. In the second strategy, $\min(m, n)$ Householder reflections are applied to the frontal matrix which leads to a triangular contribution block; this results in a relatively sparse frontal matrix which can be row-permuted in order to move most of the zeroes to the bottom-left corner in order to spare the related computations.
Sparse Multifrontal QR: the factorization phase

Strategy 1:
Sparse Multifrontal QR: the factorization phase

Strategy 1:
Sparse Multifrontal QR: the factorization phase

Strategy 1:
Strategy 1:
Sparse Multifrontal QR: the factorization phase

Strategy 1:
Sparse Multifrontal QR: the factorization phase

Strategy 2:
Sparse Multifrontal QR: the factorization phase

Strategy 2:
Sparse Multifrontal QR: the factorization phase

Strategy 2:
Sparse Multifrontal QR: the factorization phase

Strategy 2:
Sparse Multifrontal QR: the factorization phase

Strategy 2:
Sparse Multifrontal QR: the factorization phase

Strategy 2:
Sparse Multifrontal QR: the factorization phase

- **Strategy 1** is much easier to implement
- **Strategy 2:**
  - leads to smaller fronts
  - requires less storage
  - minimizes the overall number of operations
  - reduces communications

<table>
<thead>
<tr>
<th>matrix</th>
<th>Strat.</th>
<th>Max front</th>
<th>flops</th>
<th>size $Q$</th>
<th>facto time</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium2</td>
<td>1</td>
<td>6896</td>
<td>3197633920</td>
<td>4226923</td>
<td>1022.68</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>689</td>
<td>281359552</td>
<td>1065149</td>
<td>254.34</td>
</tr>
<tr>
<td>Nfac70</td>
<td>1</td>
<td>14284</td>
<td>1312200576</td>
<td>4269607</td>
<td>428.38</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>240</td>
<td>24316052</td>
<td>476636</td>
<td>60.34</td>
</tr>
<tr>
<td>WEST2021</td>
<td>1</td>
<td>26</td>
<td>409225</td>
<td>17796</td>
<td>7.78</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>26</td>
<td>390033</td>
<td>22439</td>
<td>8.84</td>
</tr>
</tbody>
</table>
Example: the `FLOWER_4_4` matrix
Sparse Direct LU/Cholesky: resume
1. **Analysis**: a preprocessing phase aiming at improving the structural properties of the problem and preparing the ground for the numerical factorization. Only symbolic operations are done at this phase:
   - Fill-reducing ordering/permutations;
   - Symbolic factorization;
   - Elimination tree computation;
   - Tree amalgamation;
   - Tree traversal order (for memory minimization);
   - Tree mapping (for parallel factorization).

2. **Factorization**: the actual numerical factorization is performed based on the result of the analysis. This can be done in parallel end, eventually, with pivoting to improve numerical stability (no pivoting in the symmetric, positive definite case).

3. **Solve**: this step solves the problem against one or more right-hand sides using the forward and backward substitutions. This can also be done in parallel following the same computational pattern of the factorization phase.
Iterative vs Direct solvers

### Iterative Solvers

- ▲ lower memory consumption
- ▲ fewer flops
- ▲ better scalability (if prec not too complex)
- ▼ less robust: may not converge or converge too slowly

### Direct Solvers

- ▲ more robust
- ▲ ideal for multiple solutions
- ▲ works better with multiple rhs
- ▼ requires more memory and flops (due to fill-in) especially in 3D problems
- ▼ less scalable