Outline

1. Sparse Matrices
2. Sparse Triangular Solve
3. Cholesky Factorization
4. Sparse Cholesky Factorization
Matrix is *sparse* if most of its entries are zero

For efficiency, store and operate on only nonzero entries, e.g., $cmod(j, k)$ need not be done if $a_{jk} = 0$

But more complicated data structures required incur extra overhead in storage and arithmetic operations

Matrix is "usefully" sparse if it contains enough zero entries to be worth taking advantage of them to reduce storage and work required

In practice, grid discretizations often yield matrices with $\Theta(n)$ nonzero entries, i.e., (small) constant number of nonzeros per row or column
**Graph Model**

- **Adjacency Graph** $G(A)$ of symmetric $n \times n$ matrix $A$ is an undirected graph having $n$ vertices, with edge between vertices $i$ and $j$ if $a_{ij} \neq 0$

- Number of edges in $G(A)$ is the number of nonzeros in $A$

- For a grid-based discretization, $G(A)$ is the grid

- Adjacency graph provides visual representation of algorithms and highlights connections between numerical and combinatorial algorithms

- For nonsymmetric $A$, $G(A)$ would be directed

- Often convenient to think of $a_{ij}$ as the weight of edge $(i, j)$
Sparse Matrix Representations

- **Coordinate (COO)** (naive) format – store each nonzero along with its row and column index
- **Compressed-sparse-row (CSR)** format
  - Store value and column index for each nonzero
  - Store index of first nonzero for each row

- Storage for CSR is less than COO and CSR ordering is often convenient
- **CSC** (compressed-sparse column), blocked versions (e.g. CSB), and other storage formats are also used
Dense matrix mappings can be adapted to sparse matrices

- 1-D blocked mapping – store all nonzeros in \( n/p \) consecutive rows on each processor
- 1-D cyclic or randomized mapping – store all nonzeros in some subset of \( n/p \) rows on each processor
- 2-D block mapping – store all nonzeros in a \( n/\sqrt{p} \times n/\sqrt{p} \) block of matrix

- 1-D blocked mappings are best for exploiting locality in graph, especially when there are \( \Theta(n) \) nonzeros

- Row ordering matters for all mappings, randomization and cyclicity yield load balance, blocking can yield locality
Sparse Matrix Vector Multiplication

- **Sparse matrix vector multiplication (SpMV)** is
  
  \[ y = Ax \]
  
  where \( A \) is sparse and \( x \) is dense

- CSR-based matrix-vector product, for all \( i \) (in parallel) do
  
  \[ x_i = \sum_j a_{i,c(j)} x_{c(j)} = \sum_{j=1}^{n} a_{ij} x_j \]
  
  where \( c(j) \) is the index of the \( j \)th nonzero in row \( i \)

- For random 1-D or 2-D mapping, cost of vector communication is same as in corresponding dense case
SpMV with 1-D Mapping

- For 1D blocking (each processor owns $n/p$ rows), number of elements of $x$ needed by a processor is the number of columns with a nonzero in the rows it owns.
- In general, want to order rows to minimize maximum number of vector elements needed on any processor.
- Graphically, we want to partition the graph into $p$ subsets of $n/p$ nodes, to minimize the maximum number of nodes to which any subset is connected, i.e., for $G(A) = (V, E)$,

$$V = V_1 \cup \cdots \cup V_p, \quad |V_i| = n/p$$

is selected to minimize

$$\max_i(|\{v : v \in V \setminus V_i, \exists w \in V_i, (v, w) \in E\}|)$$
Surface Area to Volume Ratio in SpMV

- The number of external vertices the maximum partition is adjacent to depends on the *expansion* of the graph.
- Expansion can be interpreted as a measure of the surface-area to volume ratio of the subgraphs.
- For example, for a $k \times k \times k$ grid, a subvolume of $k/p^{1/3} \times k/p^{1/3} \times k/p^{1/3}$ has surface area $\Theta(k^2/p^{2/3})$.
- Communication for this case becomes a neighbor *halo exchange* on a 3-D processor mesh.
- Thus, finding the best 1-D partitioning for SpMV often corresponds to *domain partitioning* and depends on the physical geometry of the problem.
Other Sparse Matrix Products

- SpMV is of critical importance to many numerical methods, but suffers from a low flop-to-byte ratio and a potentially high communication bandwidth cost.

- In graph algorithms, $SpMSpV$ ($x$ and $y$ are sparse) is prevalent, which is even harder to perform efficiently (e.g., to minimize work need layout other than CSR, like CSC).

- $SpMM$ ($x$ becomes dense matrix $X$) provides a higher flop-to-byte ratio and is much easier to do efficiently.

- $SpGEMM$ ($SpMSpM$) (matrix multiplication where all matrices are sparse) arises in e.g., algebraic multigrid and graph algorithms, efficiency is highly dependent on sparsity.
Solving Triangular Sparse Linear Systems

Given sparse lower-triangular matrix $L$ and vector $b$, solve

$$Lx = b$$

- all nonzeros of $L$ must be in its lower-triangular part
- Sequential algorithm: take $x_i = b_i/l_{ii}$, update

$$b_j = b_j - l_{ji}x_i \quad \text{for all} \quad j \in \{i+1, \ldots, n\}$$

- If $L$ has $m > n$ nonzeros, require $Q_1 \approx 2m$ operations
Parallelism in Sparse Triangular Solve

- We can adapt any dense parallel triangular solve algorithm to the sparse case
  - Again have fan-in (left-looking) and fan-out (right-looking) variants
  - Communication cost stays the same, computational cost decreases

- In fact there may be additional sources of parallelism, e.g., if $l_{21} = 0$, we can solve for $x_1$ and $x_2$ concurrently

- More generally, can \textit{concurrently prune leaves} of directed acyclic adjacency graph (DAG) $G(A) = (V, E)$, where $(i, j) \in E$ if $l_{ij} \neq 0$

- Depth of algorithm corresponds to diameter of this DAG
Parallel Algorithm for Sparse Triangular Solve

- **Partition**: associate fine-grain tasks with each \((i, j)\) such that \(l_{ij} \neq 0\)
- **Communicate**: task \((i, i)\) communicates with task \((j, i)\) and \((i, j)\) for all possible \(j\)
- **Agglomerate**: form coarse-grain tasks for each column of \(L\), i.e., do 1-D agglomeration, combining fine-grain tasks \((*, i)\) into agglomerated task \(i\)
- **Map**: assign coarse-grain tasks (columns of \(L\)) to processors with blocking (for locality) and/or cyclicity (for load balance and concurrency)
Cost of 1-D algorithm will clearly be less than the corresponding algorithm for the dense case.

Load balance depends on distribution of nonzeros, cyclicity can help distribute dense blocks.

Naive algorithm with 1-D column blocking exploits concurrency only in fan-out updates.

Communication bandwidth cost depends on surface-to-volume ratio of each subset of vertices associated with a block of columns.

Higher concurrency and better performance possible with dynamic/adaptive algorithms.
Symmetric positive definite matrix $A$ has **Cholesky factorization**

$$A = LL^T$$

where $L$ is lower triangular matrix with positive diagonal entries.

Linear system

$$Ax = b$$

can then be solved by forward-substitution in lower triangular system $Ly = b$, followed by back-substitution in upper triangular system $L^Tx = y$.
Algorithm for computing Cholesky factorization can be derived by equating corresponding entries of $A$ and $LL^T$ and generating them in correct order.

For example, in $2 \times 2$ case

\[
\begin{bmatrix}
a_{11} & a_{21} \\
a_{21} & a_{22}
\end{bmatrix} =
\begin{bmatrix}
\ell_{11} & 0 \\
\ell_{21} & \ell_{22}
\end{bmatrix}
\begin{bmatrix}
\ell_{11} & \ell_{21} \\
0 & \ell_{22}
\end{bmatrix}
\]

so we have

\[
\ell_{11} = \sqrt{a_{11}}, \quad \ell_{21} = \frac{a_{21}}{\ell_{11}}, \quad \ell_{22} = \sqrt{a_{22} - \ell_{21}^2}
\]
for $k = 1$ to $n$
    $a_{kk} = \sqrt{a_{kk}}$
    for $i = k + 1$ to $n$
        $a_{ik} = a_{ik}/a_{kk}$
    end
    for $j = k + 1$ to $n$
        for $i = j$ to $n$
            $a_{ij} = a_{ij} - a_{ik} a_{jk}$
        end
    end
end
All \( n \) square roots are of positive numbers, so algorithm well defined

Only lower triangle of \( A \) is accessed, so strict upper triangular portion need not be stored

Factor \( L \) computed in place, overwriting lower triangle of \( A \)

Pivoting is not required for numerical stability

About \( n^3/6 \) multiplications and similar number of additions are required (about half as many as for LU)
Parallel Algorithm

Partition

- For $i, j = 1, \ldots, n$, fine-grain task $(i, j)$ stores $a_{ij}$ and computes and stores
  \[
  \begin{cases}
  \ell_{ij}, & \text{if } i \geq j \\
  \ell_{ji}, & \text{if } i < j
  \end{cases}
  \]
  yielding 2-D array of $n^2$ fine-grain tasks

- Zero entries in upper triangle of $L$ need not be computed or stored, so for convenience in using 2-D mesh network, $\ell_{ij}$ can be redundantly computed as both task $(i, j)$ and task $(j, i)$ for $i > j$
Fine-Grain Tasks and Communication

\[ a_{11} \rightarrow a_{21} \rightarrow a_{31} \rightarrow a_{41} \rightarrow a_{51} \rightarrow a_{61} \]
\[ \ell_{11} \rightarrow \ell_{21} \rightarrow \ell_{31} \rightarrow \ell_{41} \rightarrow \ell_{51} \rightarrow \ell_{61} \]
\[ a_{21} \rightarrow a_{22} \rightarrow a_{32} \rightarrow a_{42} \rightarrow a_{52} \rightarrow a_{62} \]
\[ \ell_{21} \rightarrow \ell_{22} \rightarrow \ell_{32} \rightarrow \ell_{42} \rightarrow \ell_{52} \rightarrow \ell_{62} \]
\[ a_{31} \rightarrow a_{32} \rightarrow a_{33} \rightarrow a_{43} \rightarrow a_{53} \rightarrow a_{63} \]
\[ \ell_{31} \rightarrow \ell_{32} \rightarrow \ell_{33} \rightarrow \ell_{43} \rightarrow \ell_{53} \rightarrow \ell_{63} \]
\[ a_{41} \rightarrow a_{42} \rightarrow a_{43} \rightarrow a_{44} \rightarrow a_{54} \rightarrow a_{64} \]
\[ \ell_{41} \rightarrow \ell_{42} \rightarrow \ell_{43} \rightarrow \ell_{44} \rightarrow \ell_{54} \rightarrow \ell_{64} \]
\[ a_{51} \rightarrow a_{52} \rightarrow a_{53} \rightarrow a_{54} \rightarrow a_{55} \rightarrow a_{65} \]
\[ \ell_{51} \rightarrow \ell_{52} \rightarrow \ell_{53} \rightarrow \ell_{54} \rightarrow \ell_{55} \rightarrow \ell_{65} \]
\[ a_{61} \rightarrow a_{62} \rightarrow a_{63} \rightarrow a_{64} \rightarrow a_{65} \rightarrow a_{66} \]
\[ \ell_{61} \rightarrow \ell_{62} \rightarrow \ell_{63} \rightarrow \ell_{64} \rightarrow \ell_{65} \rightarrow \ell_{66} \]
Fine-Grain Parallel Algorithm

for $k = 1$ to $\min(i, j) - 1$
  recv broadcast of $a_{kj}$ from task $(k, j)$
  recv broadcast of $a_{ik}$ from task $(i, k)$
  $a_{ij} = a_{ij} - a_{ik} a_{kj}$
end
if $i = j$ then
  $a_{ii} = \sqrt{a_{ii}}$
  broadcast $a_{ii}$ to tasks $(k, i)$ and $(i, k)$, $k = i + 1, \ldots, n$
else if $i < j$ then
  recv broadcast of $a_{ii}$ from task $(i, i)$
  $a_{ij} = a_{ij} / a_{ii}$
  broadcast $a_{ij}$ to tasks $(k, j)$, $k = i + 1, \ldots, n$
else
  recv broadcast of $a_{jj}$ from task $(j, j)$
  $a_{ij} = a_{ij} / a_{jj}$
  broadcast $a_{ij}$ to tasks $(i, k)$, $k = j + 1, \ldots, n$
end
Agglomeration Schemes

**Agglomerate**

- Agglomeration of fine-grain tasks produces
  - 2-D
  - 1-D column
  - 1-D row

parallel algorithms analogous to those for LU factorization, with similar performance and scalability
Loop Orderings for Cholesky

Each choice of $i$, $j$, or $k$ index in outer loop yields different Cholesky algorithm, named for portion of matrix updated by basic operation in inner loops

- **Submatrix-Cholesky**: (fan-out) with $k$ in outer loop, inner loops perform rank-1 update of remaining unreduced submatrix using current column

- **Column-Cholesky**: (fan-in) with $j$ in outer loop, inner loops compute current column using matrix-vector product that accumulates effects of previous columns

- **Row-Cholesky**: (fan-in) with $i$ in outer loop, inner loops compute current row by solving triangular system involving previous rows
Memory Access Patterns

Submatrix-Cholesky

Column-Cholesky

Row-Cholesky

- read only
- read and write

Sparse Matrices
Sparse Triangular Solve
Cholesky Factorization
Sparse Cholesky Factorization

Michael T. Heath and Edgar Solomonik
Parallel Numerical Algorithms
Column-Oriented Cholesky Algorithms

**Submatrix-Cholesky**

\[
\text{for } k = 1 \text{ to } n \\
\quad a_{kk} = \sqrt{a_{kk}} \\
\quad \text{for } i = k + 1 \text{ to } n \\
\quad \quad a_{ik} = a_{ik}/a_{kk} \\
\quad \text{end} \\
\quad \text{for } j = k + 1 \text{ to } n \\
\quad \quad \text{for } i = j \text{ to } n \\
\quad \quad \quad a_{ij} = a_{ij} - a_{ik}a_{jk} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end}
\]

**Column-Cholesky**

\[
\text{for } j = 1 \text{ to } n \\
\quad \text{for } k = 1 \text{ to } j - 1 \\
\quad \quad \text{for } i = j \text{ to } n \\
\quad \quad \quad a_{ij} = a_{ij} - a_{ik}a_{jk} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\quad a_{jj} = \sqrt{a_{jj}} \\
\quad \text{for } i = j + 1 \text{ to } n \\
\quad \quad a_{ij} = a_{ij}/a_{jj} \\
\quad \text{end} \\
\text{end}
\]

Michael T. Heath and Edgar Solomonik
Parallel Numerical Algorithms 25 / 63
Column Operations

Column-oriented algorithms can be stated more compactly by introducing column operations

- $cdiv(j)$: column $j$ is divided by square root of its diagonal entry

\[
a_{jj} = \sqrt{a_{jj}}
\]

\[
\text{for } i = j + 1 \text{ to } n
\]

\[
a_{ij} = a_{ij} / a_{jj}
\]

\[
\text{end}
\]

- $cmod(j, k)$: column $j$ is modified by multiple of column $k$, with $k < j$

\[
\text{for } i = j \text{ to } n
\]

\[
a_{ij} = a_{ij} - a_{ik} a_{jk}
\]

\[
\text{end}
\]
Column-Oriented Cholesky Algorithms

**Submatrix-Cholesky**

\[
\text{for } k = 1 \text{ to } n \\
\quad \text{cdiv}(k) \\
\quad \text{for } j = k + 1 \text{ to } n \\
\quad \quad \text{cmod}(j, k) \\
\text{end} \\
\text{end}
\]

- right-looking
- immediate-update
- data-driven
- fan-out

**Column-Cholesky**

\[
\text{for } j = 1 \text{ to } n \\
\quad \text{for } k = 1 \text{ to } j - 1 \\
\quad \quad \text{cmod}(j, k) \\
\quad \text{end} \\
\quad \text{cdiv}(j) \\
\text{end}
\]

- left-looking
- delayed-update
- demand-driven
- fan-in
Data Dependences

\[ \text{cdiv}(k) \]

\[ \text{cmod}(k + 1, k) \quad \text{cmod}(k + 2, k) \quad \ldots \quad \text{cmod}(n, k) \]

\[ \text{cmod}(k, 1) \quad \text{cmod}(k, 2) \quad \ldots \quad \text{cmod}(k, k - 1) \]
Data Dependences

- $cmod(k, \star)$ operations along bottom can be done in any order, but they all have the same target column, so updating must be coordinated to preserve data integrity.

- $cmod(\star, k)$ operations along top can be done in any order, and they all have different target columns, so updating can be done simultaneously.

Performing $cmods$ concurrently is most important source of parallelism in column-oriented factorization algorithms.

For dense matrix, each $cdiv(k)$ depends on immediately preceding column, so $cdivs$ must be done sequentially.
For sparse matrix \( M \), let \( M_{i \star} \) denote its \( i \)th row and \( M_{\star j} \) its \( j \)th column.

Define \( \text{Struct}(M_{i \star}) = \{ k < i \mid m_{ik} \neq 0 \} \), nonzero structure of row \( i \) of strict lower triangle of \( M \).

Define \( \text{Struct}(M_{\star j}) = \{ k > j \mid m_{kj} \neq 0 \} \), nonzero structure of column \( j \) of strict lower triangle of \( M \).
Sparse Cholesky Algorithms

**Submatrix-Cholesky**

```
for \( k = 1 \) to \( n \)
    \( \text{cdiv}(k) \)
    for \( j \in \text{Struct}(L_{*k}) \)
        \( \text{cmod}(j, k) \)
    end
end
```

- right-looking
- immediate-update
- data-driven
- fan-out

**Column-Cholesky**

```
for \( j = 1 \) to \( n \)
    for \( k \in \text{Struct}(L_{j*}) \)
        \( \text{cmod}(j, k) \)
    end
    \( \text{cdiv}(j) \)
end
```

- left-looking
- delayed-update
- demand-driven
- fan-in
Recall that adjacency graph $G(A)$ of symmetric $n \times n$ matrix $A$ is undirected graph with edge between vertices $i$ and $j$ if $a_{ij} \neq 0$

At each step of Cholesky factorization algorithm, corresponding vertex is eliminated from graph

- Neighbors of eliminated vertex in previous graph become *clique* (fully connected subgraph) in modified graph
- Entries of $A$ that were initially zero may become nonzero entries, called *fill*
Example: Graph Model of Elimination
Elimination Tree

- $\text{parent}(j)$ is row index of first offdiagonal nonzero in column $j$ of $L$, if any, and $j$ otherwise

- **Elimination tree** $T(A)$ is graph having $n$ vertices, with edge between vertices $i$ and $j$, for $i > j$, if $i = \text{parent}(j)$

- If matrix is irreducible, then elimination tree is single tree with root at vertex $n$; otherwise, it is more accurately termed **elimination forest**

- $T(A)$ is spanning tree for **filled graph**, $F(A)$, which is $G(A)$ with all fill edges added

- Each column of Cholesky factor $L$ depends only on its descendants in elimination tree
Example: Elimination Tree

\[ A \times L = G \times F \times T (A) \]
Effect of Matrix Ordering

- Amount of fill depends on order in which variables are eliminated
- Example: “arrow” matrix — if first row and column are dense, then factor fills in completely, but if last row and column are dense, then they cause no fill
General problem of finding ordering that minimizes fill is NP-complete, but there are relatively cheap heuristics that limit fill effectively

- **Bandwidth or profile reduction**: reduce distance of nonzero diagonals from main diagonal (e.g., RCM)
- **Minimum degree**: eliminate node having fewest neighbors first
- **Nested dissection**: recursively split graph into pieces using a *vertex separator*, numbering separator vertices last
For symmetric positive definite (SPD) matrices, ordering can be determined in advance of numeric factorization.

Only locations of nonzeros matter, not their numerical values, since pivoting is not required for numerical stability.

Once ordering is selected, locations of all fill entries in $L$ can be anticipated and efficient static data structure set up to accommodate them prior to numeric factorization.

Structure of column $j$ of $L$ is given by union of structures of lower triangular portion of column $j$ of $A$ and prior columns of $L$ whose first nonzero below diagonal is in row $j$. 
Solving Sparse SPD Systems

Basic steps in solving sparse SPD systems by Cholesky factorization

1. **Ordering**: Symmetrically reorder rows and columns of matrix so Cholesky factor suffers relatively little fill

2. **Symbolic factorization**: Determine locations of all fill entries and allocate data structures in advance to accommodate them

3. **Numeric factorization**: Compute numeric values of entries of Cholesky factor

4. **Triangular solve**: Compute solution by forward- and back-substitution
In sparse submatrix- or column-Cholesky, if $a_{jk} = 0$, then $c_{mod}(j, k)$ is omitted.

Sparse factorization thus has additional source of parallelism, since “missing” $c_{mods}$ may permit multiple $c_{divs}$ to be done simultaneously.

Elimination tree shows data dependences among columns of Cholesky factor $L$, and hence identifies potential parallelism.

At any point in factorization process, all factor columns corresponding to $leaves$ in the elimination tree can be computed simultaneously.
Parallel Sparse Cholesky

- **Height** of elimination tree determines longest serial path through computation, and hence parallel execution time.

- **Width** of elimination tree determines degree of parallelism available.

- Short, wide, well-balanced elimination tree desirable for parallel factorization.

- Structure of elimination tree depends on ordering of matrix.

- So ordering should be chosen *both* to preserve sparsity and to enhance parallelism.
Levels of Parallelism in Sparse Cholesky

- **Fine-grain**
  - Task is one multiply-add pair
  - Available in either dense or sparse case
  - Difficult to exploit effectively in practice

- **Medium-grain**
  - Task is one $cmod$ or $cdiv$
  - Available in either dense or sparse case
  - Accounts for most of speedup in dense case

- **Large-grain**
  - Task computes entire set of columns in subtree of elimination tree
  - Available only in sparse case
Example: Band Ordering, 1-D Grid

\[ G(A), A, L, T(A) \]
Example: Minimum Degree, 1-D Grid

\[ G(A) \]

\[ A \]

\[ L \]

\[ T(A) \]
Example: Nested Dissection, 1-D Grid

\[ A \] and \[ L \] are shown with corresponding \( G(A) \) and \( T(A) \) structures.
Example: Band Ordering, 2-D Grid

\[ G(A) \]

\[ A \]

\[ L \]

\[ T(A) \]
Example: Minimum Degree, 2-D Grid
Example: Nested Dissection, 2-D Grid
Cyclic mapping of columns to processors works well for dense problems, because it balances load and communication is global anyway.

To exploit locality in communication for sparse factorization, better approach is to map columns in subtree of elimination tree onto local subset of processors.

Still use cyclic mapping within dense submatrices ("supernodes")
Example: Subtree Mapping
Fan-Out Sparse Cholesky

$$\text{for } j \in \text{mycols}$$

$$\text{if } j \text{ is leaf node in } T(A) \text{ then}$$

$$\text{cdiv}(j)$$

$$\text{send } L_{*j} \text{ to processes in } \text{map}(\text{Struct}(L_{*j}))$$

$$\text{mycols} = \text{mycols} - \{ j \}$$

$$\text{end}$$

$$\text{end}$$

$$\text{while } \text{mycols} \neq \emptyset$$

$$\text{receive any column of } L, \text{ say } L_{*k}$$

$$\text{for } j \in \text{mycols} \cap \text{Struct}(L_{*k})$$

$$\text{cmod}(j, k)$$

$$\text{if column } j \text{ requires no more } \text{cmods} \text{ then}$$

$$\text{cdiv}(j)$$

$$\text{send } L_{*j} \text{ to processes in } \text{map}(\text{Struct}(L_{*j}))$$

$$\text{mycols} = \text{mycols} - \{ j \}$$

$$\text{end}$$

$$\text{end}$$

$$\text{end}$$

Michael T. Heath and Edgar Solomonik
Fan-In Sparse Cholesky

\[
\text{for } j = 1 \text{ to } n \\
\text{if } j \in \text{mycols or mycols} \cap \text{Struct}(L_{j*}) \neq \emptyset \text{ then} \\
\quad u = 0 \\
\quad \text{for } k \in \text{mycols} \cap \text{Struct}(L_{j*}) \\
\quad \quad u = u + \ell_{jk} L_{*k} \\
\text{if } j \in \text{mycols then} \\
\quad \text{incorporate } u \text{ into factor column } j \\
\quad \text{while any aggregated update column} \\
\quad \quad \text{for column } j \text{ remains, receive one} \\
\quad \quad \quad \text{and incorporate it into factor column } j \\
\quad \text{end} \\
\quad cdiv( j ) \\
\text{else} \\
\quad \text{send } u \text{ to process } map( j ) \\
\text{end} \\
\text{end} \\
\text{end}
\]
Multifrontal Sparse Cholesky

- Multifrontal algorithm operates recursively, starting from root of elimination tree for $A$
- Dense frontal matrix $F_j$ is initialized to have nonzero entries from corresponding row and column of $A$ as its first row and column, and zeros elsewhere
- $F_j$ is then updated by $extend_add$ operations with update matrices from its children in elimination tree
- $extend_add$ operation, denoted by $\oplus$, merges matrices by taking union of their subscript sets and summing entries for any common subscripts
- After updating of $F_j$ is complete, its partial Cholesky factorization is computed, producing corresponding row and column of $L$ as well as update matrix $U_j$
Example: *extend_add*

\[
\begin{bmatrix}
  a_{11} & a_{13} & a_{15} & a_{18} \\
  a_{31} & a_{33} & a_{35} & a_{38} \\
  a_{51} & a_{53} & a_{55} & a_{58} \\
  a_{81} & a_{83} & a_{85} & a_{88}
\end{bmatrix}
\oplus
\begin{bmatrix}
  b_{11} & b_{12} & b_{15} & b_{17} \\
  b_{21} & b_{22} & b_{25} & b_{27} \\
  b_{51} & b_{52} & b_{55} & b_{57} \\
  b_{71} & b_{72} & b_{75} & b_{77}
\end{bmatrix}
= \\
\begin{bmatrix}
  a_{11} + b_{11} & b_{12} & a_{13} & a_{15} + b_{15} & b_{17} & a_{18} \\
  b_{21} & b_{22} & 0 & b_{25} & b_{27} & 0 \\
  a_{31} & 0 & a_{33} & a_{35} & 0 & a_{38} \\
  a_{51} + b_{51} & b_{52} & a_{53} & a_{55} + b_{55} & b_{57} & a_{58} \\
  b_{71} & b_{72} & 0 & b_{75} & b_{77} & 0 \\
  a_{81} & 0 & a_{83} & a_{85} & 0 & a_{88}
\end{bmatrix}
\]
Multifrontal Sparse Cholesky

Factor(\(j\))

Let \(\{i_1, \ldots, i_r\} = \text{Struct}(L_{xj})\)

Let \(F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j} & 0 & \cdots & 0 \end{bmatrix}\)

for each child \(i\) of \(j\) in elimination tree

Factor(\(i\))

\(F_j = F_j \oplus U_i\)

end

Perform one step of dense Cholesky:

\(F_j = \begin{bmatrix} l_{j,j} & 0 \\ l_{i_1,j} & I \\ \vdots & \vdots \\ l_{i_r,j} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & U_j \end{bmatrix} \begin{bmatrix} l_{j,j} & l_{i_1,j} & \cdots & l_{i_r,j} \end{bmatrix}\)
Advantages of Multifrontal Method

- Most arithmetic operations performed on dense matrices, which reduces indexing overhead and indirect addressing.
- Can take advantage of loop unrolling, vectorization, and optimized BLAS to run at near peak speed on many types of processors.
- Data locality good for memory hierarchies, such as cache, virtual memory with paging, or explicit out-of-core solvers.
- Naturally adaptable to parallel implementation by processing multiple independent fronts simultaneously on different processors.
- Parallelism can also be exploited in dense matrix computations within each front.
Summary for Parallel Sparse Cholesky

Principal ingredients in efficient parallel algorithm for sparse Cholesky factorization

- Reordering matrix to obtain relatively short and well balanced elimination tree while also limiting fill
- Multifrontal or supernodal approach to exploit dense subproblems effectively
- Subtree mapping to localize communication
- Cyclic mapping of dense subproblems to achieve good load balance
- 2-D algorithm for dense subproblems to enhance scalability
Scalability of Sparse Cholesky

- Performance and scalability of sparse Cholesky depend on sparsity structure of particular matrix.

- Sparse factorization with nested dissection requires factorization of dense matrix of dimension $\Theta(\sqrt{n})$ for 2-D grid problem with $n$ grid points ($\sqrt{n}$ is the size of the root vertex separator), for which unconditional weak scalability is possible.

- However, efficiency often deteriorates as a result of the rest of the sparse factorization taking more time.
References – Dense Cholesky


References – Sparse Cholesky


References – Multifrontal Methods

References – Scalability

References – Nonsymmetric Sparse Systems


