Parallel Numerical Algorithms
Chapter 7 – Cholesky Factorization

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512
Outline

1. Cholesky Factorization
2. Parallel Dense Cholesky
3. Parallel Sparse Cholesky
Symmetric positive definite matrix $A$ has \textit{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 $L^T x = y$.
Computing Cholesky Factorization

- 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 $L$ is 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

Cholesky Factorization
Parallel Dense Cholesky
Parallel Sparse Cholesky
Parallel Algorithm
Loop Orderings
Column-Oriented Algorithms

Michael T. Heath Parallel Numerical Algorithms 8 / 52
Fine-Grain Parallel Algorithm

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

- Rather than repeat analyses for dense matrices, we focus instead on sparse matrices, for which column-oriented algorithms are typically used
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**: with \( k \) in outer loop, inner loops perform rank-1 update of remaining unreduced submatrix using current column

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

- **Row-Cholesky**: 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
Column-Oriented Cholesky Algorithms

**Submatrix-Cholesky**

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

**Column-Cholesky**

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

Michael T. Heath
Parallel Numerical Algorithms 13 / 52
Column Operations

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

- \( cd\text{div}(j) \): column \( j \) is divided by square root of its diagonal entry

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

\[
\text{for } i = j + 1 \text{ to } n \\
a_{ij} = a_{ij} / a_{jj}
\]

end

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

\[
\text{for } i = j \text{ to } n \\
a_{ij} = a_{ij} - a_{ik} a_{jk}
\]

end
Column-Oriented Cholesky Algorithms

**Submatrix-Cholesky**

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

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

**Column-Cholesky**

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

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

\[
cmod(k + 1, k) \quad \cdots \quad cmod(n, k)
\]

\[
cmod(k + 2, k) \quad \cdots \quad cmod(k, k - 1)
\]

\[
cdiv(k)
\]
Data Dependences

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

- \( cmod(\ast, k) \) 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 \( cdvs \) must be done sequentially.
Sparse Matrices

- Matrix is \textit{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, sparsity worth exploiting for family of matrices if there are \( \Theta(n) \) nonzero entries, i.e., (small) constant number of nonzeros per row or column.
For sparse matrix $M$, let $M_{i*}$ denote its $i$th row and $M_{*j}$ its $j$th column.

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

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

**Submatrix-Cholesky**

```plaintext
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**

```plaintext
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
Graph Model

- **Graph** $G(A)$ of symmetric $n \times n$ matrix $A$ is undirected graph having $n$ vertices, 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

$$A \rightarrow L$$
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

\[ A \]

\[ L \]

\[ G(A) \]

\[ F(A) \]

\[ T(A) \]
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.

\[
\begin{bmatrix}
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
  \times & \times & \times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\]
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, numbering nodes in *separators* last
Symbolic Factorization

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

Sparse factorization thus has additional source of parallelism, since “missing” $cmods$ may permit multiple $cdivs$ 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 leaf nodes of 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

\[ A \]

\[ L \]

\[ G(A) \]

\[ T(A) \]
Example: Band Ordering, 2-D Grid

\[ G(A) \]

\[
\begin{bmatrix}
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\times & \times & \times \\
\end{bmatrix}
\]

\[ L \]

\[
\begin{bmatrix}
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times \\
\end{bmatrix}
\]

\[ T(A) \]

\[
\begin{bmatrix}
9 & & & & & & \\
8 & 7 & & & & & \\
6 & 5 & 4 & & & & \\
3 & 2 & 1 & & & & \\
\end{bmatrix}
\]
Example: Minimum Degree, 2-D Grid

- $G(A)$
- $A$
- $L$
- $T(A)$
Example: Nested Dissection, 2-D Grid

- **G(A)**: Graph representation of the matrix A.
- **A**: Sparse matrix A.
- **L**: Lower triangular matrix L.
- **T(A)**: Tree representation of the matrix A, showing the nested dissection order.

The matrix A is factorized into L and U, where U is the transpose of L. This representation is used in parallel sparse Cholesky factorization methods, particularly in nested dissection and 2-D grid computations.
Mapping

- 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

\[ \text{for } j \in \text{mycols} \]
\[ \text{if } j \text{ is leaf node in } T(A) \text{ then} \]
\[ cdiv(j) \]
\[ \text{send } L*_{j} \text{ to processes in } map(Struct(L*_{j})) \]
\[ \text{mycols} = \text{mycols} - \{ j \} \]
\[ \text{end} \]
\[ \text{end} \]
\[ \text{while } \text{mycols} \neq \emptyset \]
\[ \text{receive any column of } L, \text{ say } L*_{k} \]
\[ \text{for } j \in \text{mycols} \cap Struct(L*_{k}) \]
\[ cmod(j, k) \]
\[ \text{if column } j \text{ requires no more } cmods \text{ then} \]
\[ cdiv(j) \]
\[ \text{send } L*_{j} \text{ to processes in } map(Struct(L*_{j})) \]
\[ \text{mycols} = \text{mycols} - \{ j \} \]
\[ \text{end} \]
\[ \text{end} \]
\[ \text{end} \]
Fan-In Sparse Cholesky

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

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

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

Multifrontal Sparse Cholesky

Factor(\( j \))
Let \( \{i_1, \ldots, i_r\} = \text{Struct}(L_{\ast j}) \)
\[
\begin{bmatrix}
a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\
a_{i_1,j} & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
a_{i_r,j} & 0 & \cdots & 0
\end{bmatrix}
\]
Let \( F_j = \)
\[
\begin{bmatrix}
\ell_{j,j} & 0 \\
\ell_{i_1,j} & I \\
\vdots & \vdots \\
\ell_{i_r,j} & I
\end{bmatrix}
\begin{bmatrix}
1 & 0 \\
0 & U_j
\end{bmatrix}
\begin{bmatrix}
\ell_{j,j} & \ell_{i_1,j} & \cdots & \ell_{i_r,j}
\end{bmatrix}
\]
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:
\[
F_j = \begin{bmatrix}
\ell_{j,j} & 0 \\
\ell_{i_1,j} & I \\
\vdots & \vdots \\
\ell_{i_r,j} & I
\end{bmatrix}
\begin{bmatrix}
1 & 0 \\
0 & U_j
\end{bmatrix}
\begin{bmatrix}
\ell_{j,j} & \ell_{i_1,j} & \cdots & \ell_{i_r,j}
\end{bmatrix}
\]
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
Performance and scalability of sparse Cholesky depend on sparsity structure of particular matrix.

Sparse factorization requires factorization of dense matrix of size $\Theta(\sqrt{n})$ for 2-D grid problem with $n$ grid points, so isoefficiency function is at least $\Theta(p^3)$ for 1-D algorithm and $\Theta(p\sqrt{p})$ for 2-D algorithm.

Scalability analysis is difficult for arbitrary sparse problems, but best current parallel algorithms for sparse factorization can achieve isoefficiency $\Theta(p\sqrt{p})$ for important classes of problems.


References – Sparse Cholesky


References – Multifrontal Methods


References – Scalability


References – Nonsymmetric Sparse Systems


