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

Cholesky Factorization

- Symmetric positive definite matrix $\boldsymbol{A}$ has *Cholesky factorization*

$$\boxed{\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T}$$

  where $\boldsymbol{L}$ is lower triangular matrix with positive diagonal entries

- Linear system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

  can then be solved by forward-substitution in lower triangular system $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b}$, followed by back-substitution in upper triangular system $\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{y}$

Computing Cholesky Factorization

- Algorithm for computing Cholesky factorization can be derived by equating corresponding entries of $\boldsymbol{A}$ and $\boldsymbol{L}\boldsymbol{L}^T$ and generating them in correct order
- For example, in 2×2 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

$$\ell_{11} = \sqrt{a_{11}}, \quad \ell_{21} = a_{21}/\ell_{11}, \quad \ell_{22} = \sqrt{a_{22} - \ell_{21}^2}$$

Cholesky Factorization Algorithm

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

Cholesky Factorization Algorithm

- All n square roots are of positive numbers, so algorithm well defined
- Only lower triangle of $\boldsymbol{A}$ is accessed, so strict upper triangular portion need not be stored
- Factor $\boldsymbol{L}$ is computed in place, overwriting lower triangle of $\boldsymbol{A}$
- 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

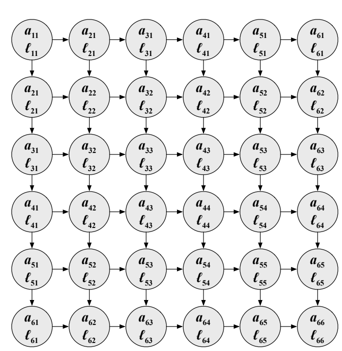
- For $i, j = 1, \ldots, n$, fine-grain task (i, j) stores a_{ij} and computes and stores

$$\begin{cases} \ell_{ij}, & \text{if } i \geq j \\ \ell_{ji}, & \text{if } i < j \end{cases}$$

  yielding 2-D array of n^2 fine-grain tasks

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

Fine-Grain Tasks and Communication



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

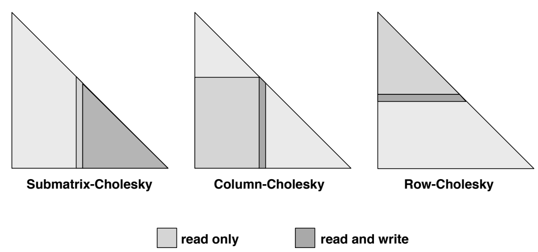
  parallel algorithms analogous to those for LU factorization, with similar performance and scalability
- Rather than repeat analyses for dense matrices, we focus instead on sparse matrices, for which column-oriented algorithms are typically used

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*: 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



Column-Oriented Cholesky Algorithms

Submatrix-Cholesky

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

Column-Cholesky

for $j = 1$ **to** n
  for $k = 1$ **to** $j - 1$
    for $i = j$ **to** n
      $a_{ij} = a_{ij} - a_{ik}\, a_{jk}$
    end
  end
  $a_{jj} = \sqrt{a_{jj}}$
  for $i = j + 1$ **to** n
    $a_{ij} = a_{ij}/a_{jj}$
  end
end

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

  $a_{jj} = \sqrt{a_{jj}}$
  for $i = j + 1$ **to** n
    $a_{ij} = a_{ij}/a_{jj}$
  end
- *cmod* (j, k): column j is modified by multiple of column k, with $k < j$

  for $i = j$ **to** n
    $a_{ij} = a_{ij} - a_{ik}\, a_{jk}$
  end

Column-Oriented Cholesky Algorithms

Submatrix-Cholesky

for $k = 1$ **to** n
  cdiv (k)
  for $j = k + 1$ **to** n
    cmod (j, k)
  end
end

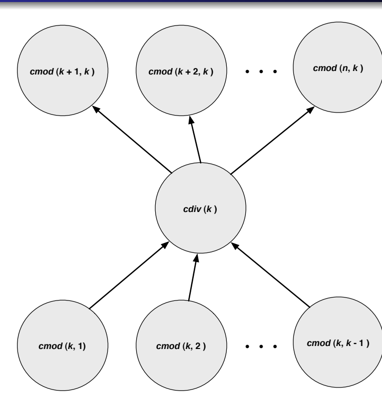
- right-looking
- immediate-update
- data-driven
- fan-out

Column-Cholesky

for $j = 1$ **to** n
  for $k = 1$ **to** $j - 1$
    cmod (j, k)
  end
  cdiv (j)
end

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

Data Dependences



Data Dependences

- *cmod* $(k,*)$ 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* $(*,k)$ 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

Sparse Matrices

- 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, 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

Sparsity Structure

- For sparse matrix $\boldsymbol{M}$, let $\boldsymbol{M}_{i*}$ denote its ith row and $\boldsymbol{M}_{*j}$ its jth column
- Define *Struct* $(\boldsymbol{M}_{i*}) = \{k < i \mid m_{ik} \neq 0\}$, nonzero structure of row i of strict lower triangle of $\boldsymbol{M}$
- Define *Struct* $(\boldsymbol{M}_{*j}) = \{k > j \mid m_{kj} \neq 0\}$, nonzero structure of column j of strict lower triangle of $\boldsymbol{M}$

Sparse Cholesky Algorithms

Submatrix-Cholesky

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

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

Column-Cholesky

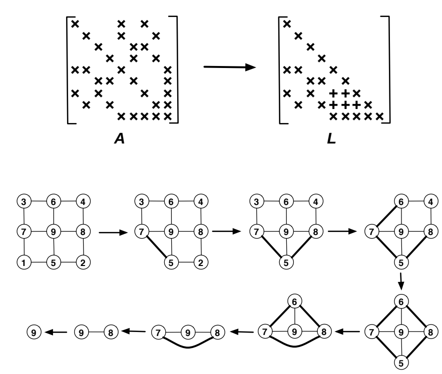
```
for j = 1 to n
    for k ∈ Struct(L_{j*})
        cmod(j, k)
    end
    cdiv(j)
end
```

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

Graph Model

- *Graph* $G(\boldsymbol{A})$ of symmetric $n \times n$ matrix $\boldsymbol{A}$ 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 $\boldsymbol{A}$ that were initially zero may become nonzero entries, called *fill*

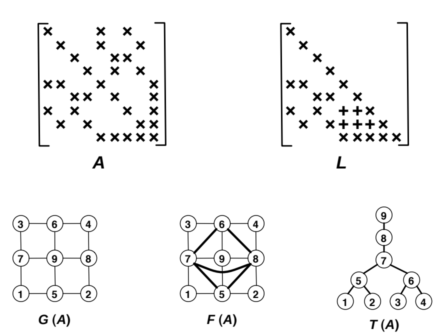
Example: Graph Model of Elimination



Elimination Tree

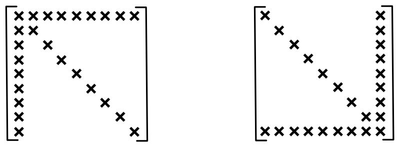
- *parent* (j) is row index of first offdiagonal nonzero in column j of $\boldsymbol{L}$, if any, and j otherwise
- *Elimination tree* $T(\boldsymbol{A})$ is graph having n vertices, with edge between vertices i and j, for $i > j$, if $i =$ *parent* (j)
- If matrix is irreducible, then elimination tree is single tree with root at vertex n; otherwise, it is more accurately termed *elimination forest*
- $T(\boldsymbol{A})$ is spanning tree for *filled graph* $F(\boldsymbol{A})$, which is $G(\boldsymbol{A})$ with all fill edges added
- Each column of Cholesky factor $\boldsymbol{L}$ depends only on its descendants in elimination tree

Example: Elimination Tree



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



Ordering Heuristics

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 $\boldsymbol{L}$ can be anticipated and efficient static data structure set up to accommodate them prior to numeric factorization
- Structure of column j of $\boldsymbol{L}$ is given by union of structures of lower triangular portion of column j of $\boldsymbol{A}$ and prior columns of $\boldsymbol{L}$ 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

Parallel Sparse Cholesky

- 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 $\boldsymbol{L}$, 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

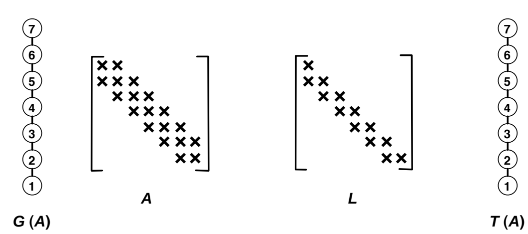
Parallel Sparse Cholesky

- *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

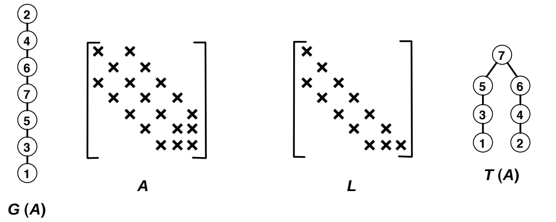
Levels of Parallelism in Sparse Cholesky

- *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

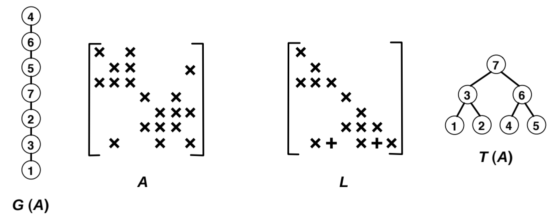
Example: Band Ordering, 1-D Grid



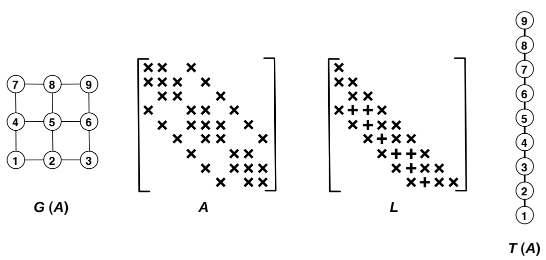
Example: Minimum Degree, 1-D Grid



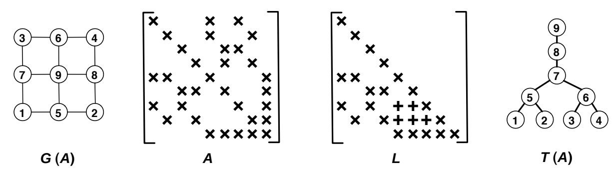
Example: Nested Dissection, 1-D Grid



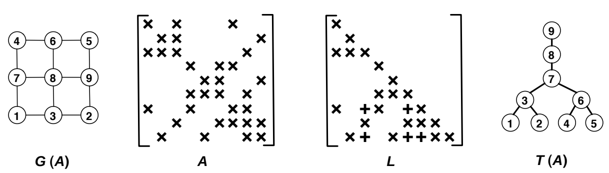
Example: Band Ordering, 2-D Grid



Example: Minimum Degree, 2-D Grid



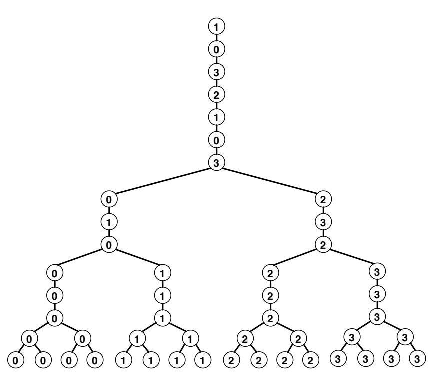
Example: Nested Dissection, 2-D Grid



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

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

Fan-In Sparse Cholesky

for $j = 1$ **to** n
  if $j \in$ *mycols* or *mycols* $\cap$ *Struct*$(\boldsymbol{L}_{j*}) \neq \emptyset$ **then**
    $u = 0$
    for $k \in$ *mycols* $\cap$ *Struct*$(\boldsymbol{L}_{j*})$
      $u = u + \ell_{jk}\, \boldsymbol{L}_{*k}$
    if $j \in$ *mycols* **then**
      incorporate u into factor column j
      while any aggregated update column
        for column j remains, receive one
        and incorporate it into factor column j
      end
      cdiv$(\,j)$
    else
      send u to process *map*$(\,j)$
    end
  end
end

Multifrontal Sparse Cholesky

- Multifrontal algorithm operates recursively, starting from root of elimination tree for $\boldsymbol{A}$
- Dense frontal matrix $\boldsymbol{F}_j$ is initialized to have nonzero entries from corresponding row and column of $\boldsymbol{A}$ as its first row and column, and zeros elsewhere
- $\boldsymbol{F}_j$ 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 $\boldsymbol{F}_j$ is complete, its partial Cholesky factorization is computed, producing corresponding row and column of $\boldsymbol{L}$ as well as update matrix $\boldsymbol{U}_j$

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 $\{i_1, \ldots, i_r\} =$ *Struct*$(\boldsymbol{L}_{*j})$

$$\text{Let } \boldsymbol{F}_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{i_1,j} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j} & 0 & \ldots & 0 \end{bmatrix}$$

  for each child i of j in elimination tree
    Factor(i)
    $\boldsymbol{F}_j = \boldsymbol{F}_j \oplus \boldsymbol{U}_i$
  end
  Perform one step of dense Cholesky:

$$\boldsymbol{F}_j = \begin{bmatrix} \ell_{j,j} & \boldsymbol{0} \\ \ell_{i_1,j} & \\ \vdots & \boldsymbol{I} \\ \ell_{i_r,j} & \end{bmatrix} \begin{bmatrix} 1 & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{U}_j \end{bmatrix} \begin{bmatrix} \ell_{j,j} & \ell_{i_1,j} & \ldots & \ell_{i_r,j} \\ \boldsymbol{0} & & \boldsymbol{I} & \end{bmatrix}$$

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 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 – Dense Cholesky

- G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, Communication-optimal parallel and sequential Cholesky decomposition, *SIAM J. Sci. Comput.* 32:3495-3523, 2010
- J. W. Demmel, M. T. Heath, and H. A. van der Vorst, Parallel numerical linear algebra, *Acta Numerica* 2:111-197, 1993
- D. O'Leary and G. W. Stewart, Data-flow algorithms for parallel matrix computations, *Comm. ACM* 28:840-853, 1985
- D. O'Leary and G. W. Stewart, Assignment and scheduling in parallel matrix factorization, *Linear Algebra Appl.* 77:275-299, 1986

References – Sparse Cholesky

- M. T. Heath, Parallel direct methods for sparse linear systems, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., *Parallel Numerical Algorithms*, pp. 55-90, Kluwer, 1997
- M. T. Heath, E. Ng and B. W. Peyton, Parallel algorithms for sparse linear systems, *SIAM Review* 33:420-460, 1991
- J. Liu, Computational models and task scheduling for parallel sparse Cholesky factorization, *Parallel Computing* 3:327-342, 1986
- J. Liu, Reordering sparse matrices for parallel elimination, *Parallel Computing* 11:73-91, 1989
- J. Liu, The role of elimination trees in sparse factorization, *SIAM J. Matrix Anal. Appl.* 11:134-172, 1990

References – Multifrontal Methods

- I. S. Duff, Parallel implementation of multifrontal schemes, *Parallel Computing* 3:193-204, 1986
- A. Gupta, Parallel sparse direct methods: a short tutorial, IBM Research Report RC 25076, November 2010
- J. Liu, The multifrontal method for sparse matrix solution: theory and practice, *SIAM Review* 34:82-109, 1992
- J. A. Scott, Parallel frontal solvers for large sparse linear systems, *ACM Trans. Math. Software* 29:395-417, 2003

References – Scalability

- A. George, J. Lui, and E. Ng, Communication results for parallel sparse Cholesky factorization on a hypercube, *Parallel Computing* 10:287-298, 1989
- A. Gupta, G. Karypis, and V. Kumar, Highly scalable parallel algorithms for sparse matrix factorization, *IEEE Trans. Parallel Distrib. Systems* 8:502-520, 1997
- T. Rauber, G. Runger, and C. Scholtes, Scalability of sparse Cholesky factorization, *Internat. J. High Speed Computing* 10:19-52, 1999
- R. Schreiber, Scalability of sparse direct solvers, A. George, J. R. Gilbert, and J. Liu, eds., *Graph Theory and Sparse Matrix Computation*, pp. 191-209, Springer-Verlag, 1993

References – Nonsymmetric Sparse Systems

- I. S. Duff and J. A. Scott, A parallel direct solver for large sparse highly unsymmetric linear systems, *ACM Trans. Math. Software* 30:95-117, 2004
- A. Gupta, A shared- and distributed-memory parallel general sparse direct solver, *Appl. Algebra Engrg. Commun. Comput.*, 18(3):263-277, 2007
- X. S. Li and J. W. Demmel, SuperLU_Dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, *ACM Trans. Math. Software* 29:110-140, 2003
- K. Shen, T. Yang, and X. Jiao, S+: Efficient 2D sparse LU factorization on parallel machines, *SIAM J. Matrix Anal. Appl.* 22:282-305, 2000