Sparse LA

Sathish Vadhiyar

Motivation

- Sparse computations much more challenging than dense due to complex data structures and memory references
- Many physical systems produce sparse matrices

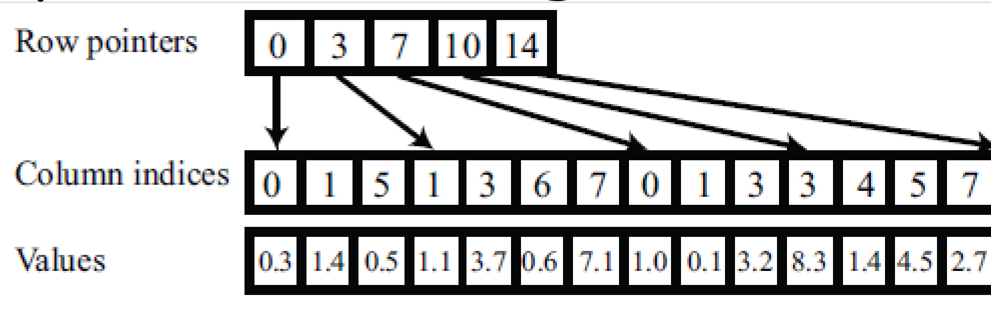
Sparse Matrix-Vector Multiplication

Cache in GPUs

- Fermi has 16KB/48KB L1 cache per SM, and a global 768 KB L2 cache
- Each cache line is 128 bytes, to provide high memory bandwidth
- Thus when using 48KB cache, only (48KB/128bytes=384) cache lines can be stored
- GPUs execute up to 1536 threads per SM
- If all threads access different cache lines, 384 of them will get cache hits, and others will get cache miss
- Thus threads in the same block should work on the same cache lines

Compressed Sparse Row (CSR) Format

- SpMV (Sparse Matrix-Vector Multiplication) using CSR



Algorithm 1 Element by element assembly of the stiffness matrix and the load vector.

for i = 0 to dimRow-1 **do**
  $row = rowPtrs[i]$
  $y[i] = 0$
  for j = 0 to rowPtrs[i+1]-row-1 **do**
    $y[i] += values[row + j] * x[colIdxs[row + j]]$
  end for
end for

Naïve CUDA Implementation

- Assign one thread to each row
- Define block size as multiple of warp size, e.g., 128
- Can handle max of 128x65535=8388480 rows, where 65535 is the max number of blocks

```
int i = blockIdx.x*blockSize + threadIdx.x;
float rowSum = 0;
int rowPtr = rowPtrs[i];
for (int j = 0; j<rowPtrs[i+1]-rowPtr; j+=1) {
        rowSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
}
y[i] = rowSum;
```

Naïve CUDA Implementation - Drawbacks

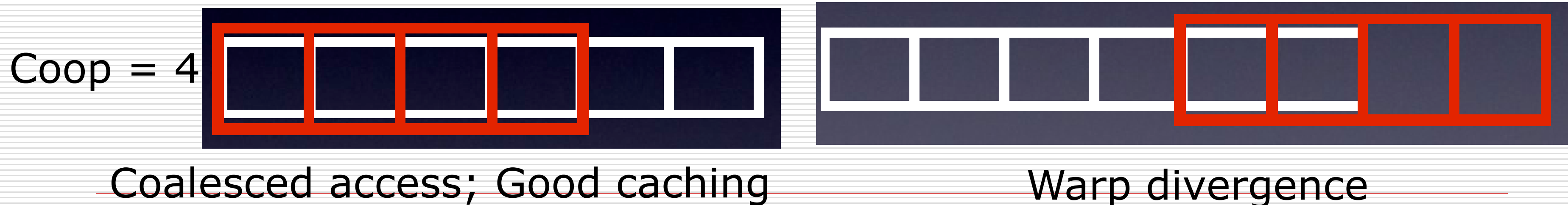
- For large matrices with several elements per row, the implementation suffers from a high cache miss rate since cache can’t hold all cache lines being used
- Thus coalescing/caching is poor for long rows
- If nnz per row have high variance, warp divergence will occur

Thread Cooperation

- Multiple threads can be assigned to work on the same row
- Cooperating threads act on adjacent elements of the row; perform multiplication with elements of vector x; add up their results in shared memory using reduction
- First thread of the group writes the result to vector y
- If the number of cooperating threads, coop, is less than warp size, the synchronization between cooperating threads is implicit

Analysis

- Same cache lines are used by cooperating threads
- Improves coalescing/caching
- If the length of the row is not a multiple of 32, can lead to warp divergence, and loss in performance



Granularity

- If a group of cooperating threads act on only row, then the number of blocks required for entire matrix may be more than 65535
- Thus, more than one row per cooperating group can be processed
- Number of rows processing by a cooperating group is denoted as repeat
- A thread block processes repeat*blockSize/coop consecutive rows
- An algorithm can be parametrized by blockSize, coop and repeat

Parametrized Algorithm

```
__global__ void csrmv(float *values, int *rowPtrs,
                                    int *colIdxs, float *x, float *y,
                                    int dimRow, int repeat, int coop) {
    int i = (repeat*blockIdx.x*blockDim.x + threadIdx.x)/coop;
    int coopIdx = threadIdx.x%coop;
    int tid = threadIdx.x;
    extern __shared__ volatile  float sdata[];
    for (int r = 0; r<repeat; r++) {
        float localSum = 0;
        if (i<dimRow) {
            // do multiplication
            int rowPtr = rowPtrs[i];
            for (int j = coopIdx; j<rowPtrs[i+1]-rowPtr; j+=coop) {
                localSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
            }
            // do reduction in shared mem
            sdata[tid] = localSum;
            for(unsigned int s=coop/2; s>0; s>>=1) {
                if (coopIdx < s) sdata[tid] += sdata[tid + s];
            }
            if (coopIdx == 0) y[i] = sdata[tid];

            i += blockDim.x/coop;
        }
    }
}
```

References

- Efficient Sparse Matrix-Vector Multiplication on cache-based GPUs. Reguly, Giles. InPar 2012.

Motivation

- Sparse computations much more challenging than dense due to complex data structures and memory references
- Many physical systems produce sparse matrices
- Commonly used computations – sparse Cholesky factorization

Sparse Cholesky

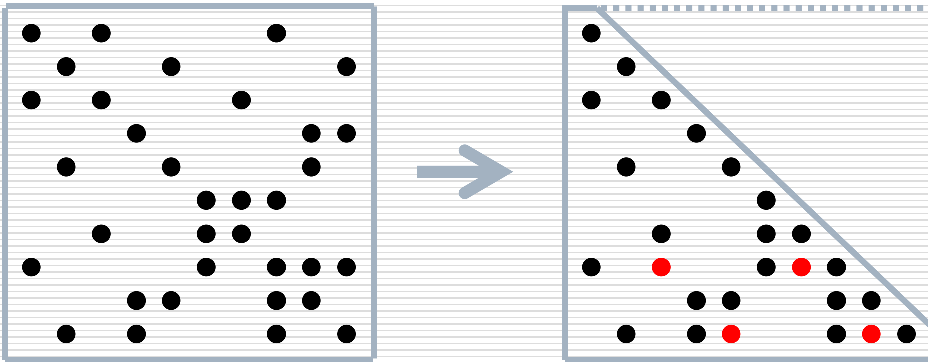
- To solve $Ax = b$;
- Most of the research and the base case are in sparse symmetric positive definite matrices
- $A = LL^T$; $Ly = b$; $L^Tx = y$;
- Cholesky factorization introduces *fill-in*

Column oriented left-looking Cholesky

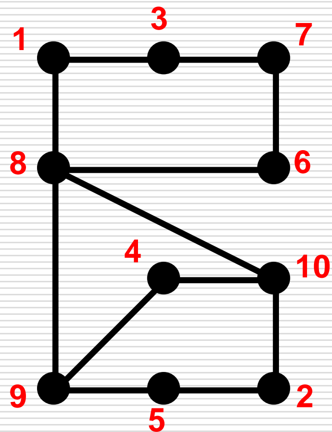
Column Oriented Cholesky factorization

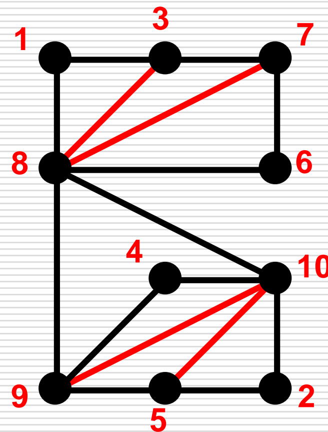
$$
\begin{array}{l}
for\ j = 1,\ n \\
\quad for\ k = 1,\ j-1 \\
\quad\quad for\ i = j,\ n \qquad \{\mathrm{cmod}(j,k)\} \\
\quad\quad\quad a_{ij} = a_{ij} - a_{ik} \cdot a_{jk} \\
\quad a_{jj} = \sqrt{a_{jj}} \\
\quad for\ k = j+1,\ n \qquad \{\mathrm{cdiv}(j)\} \\
\quad\quad a_{kj} = a_{kj}/a_{jj}
\end{array}
$$

Fill-in



Fill: new nonzeros in factor





Permutation Matrix or Ordering

- Thus ordering to reduce fill or to enhance numerical stability
- Choose permutation matrix P so that Cholesky factor L' of PAP^T has less fill than L.
- Triangular solve:

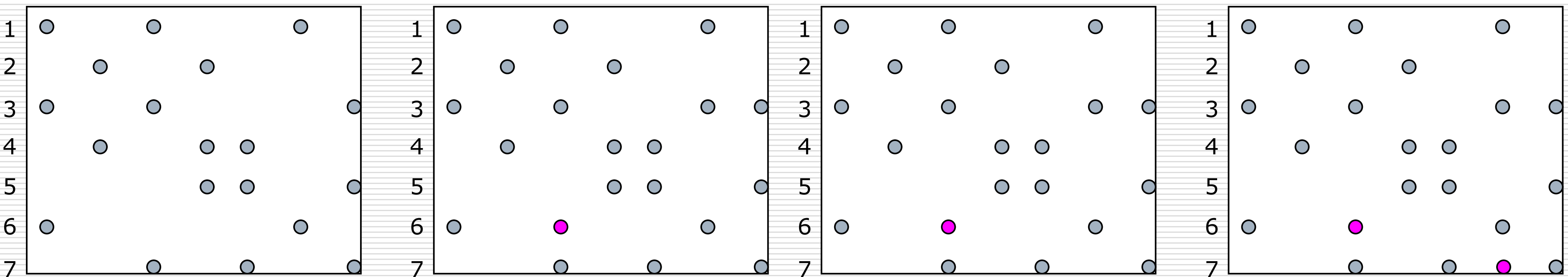
$L'y = Pb; L'^T z = y; x = P^T z$

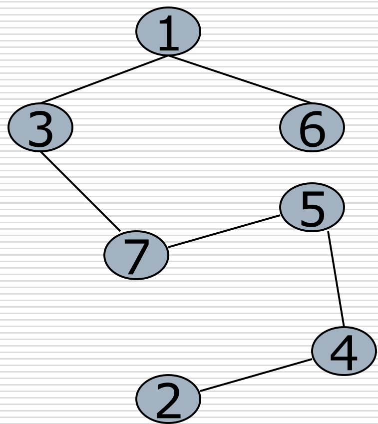
- The fill can be predicted in advance
- Static data structure can be used – *symbolic factorization*

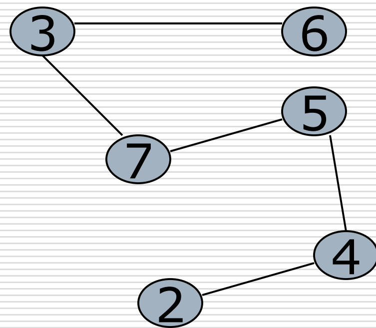
Steps

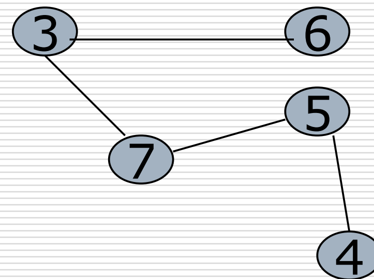
- **Ordering:**
  - Find a permutation P of matrix A,
- **Symbolic factorization:**
  - Set up a data structure for the Cholesky factor L of PAP^T,
- **Numerical factorization:**
  - Decompose PAP^T into LL^T,
- **Triangular system solution:**
  - $Ly = Pb$; $L^Tz = y$; $x = P^Tz$.

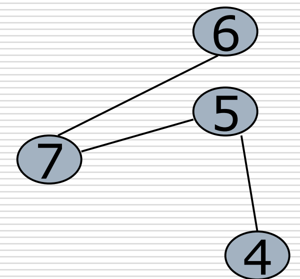
Sparse Matrices and Graph Theory





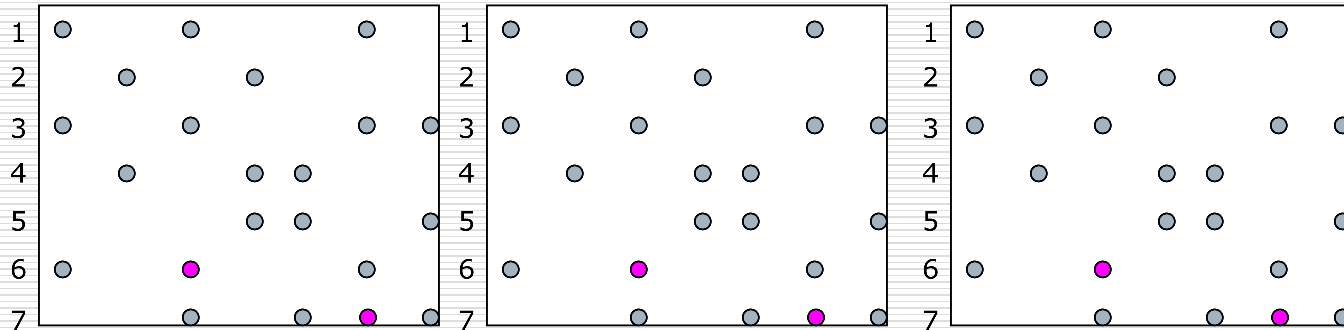


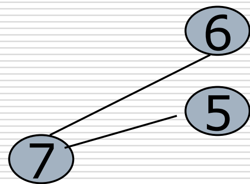


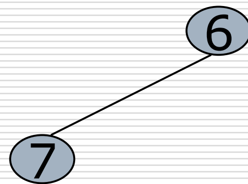


G(A)

Sparse and Graph





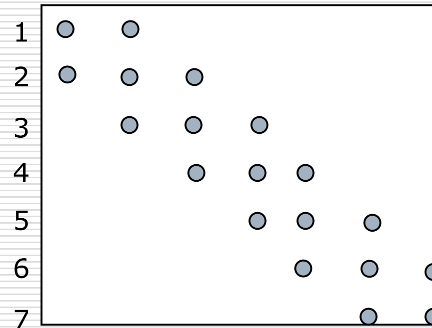


F(A)

Ordering

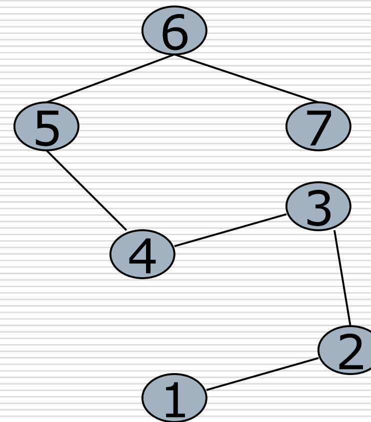
- The above order of elimination is “natural”
- The first heuristic is minimum degree ordering
- Simple and effective
- But efficiency depends on tie breaking strategy
- Difficult to parallelize!

Minimum degree ordering for the previous Matrix



Ordering – {2,4,5,7,3,1,6}

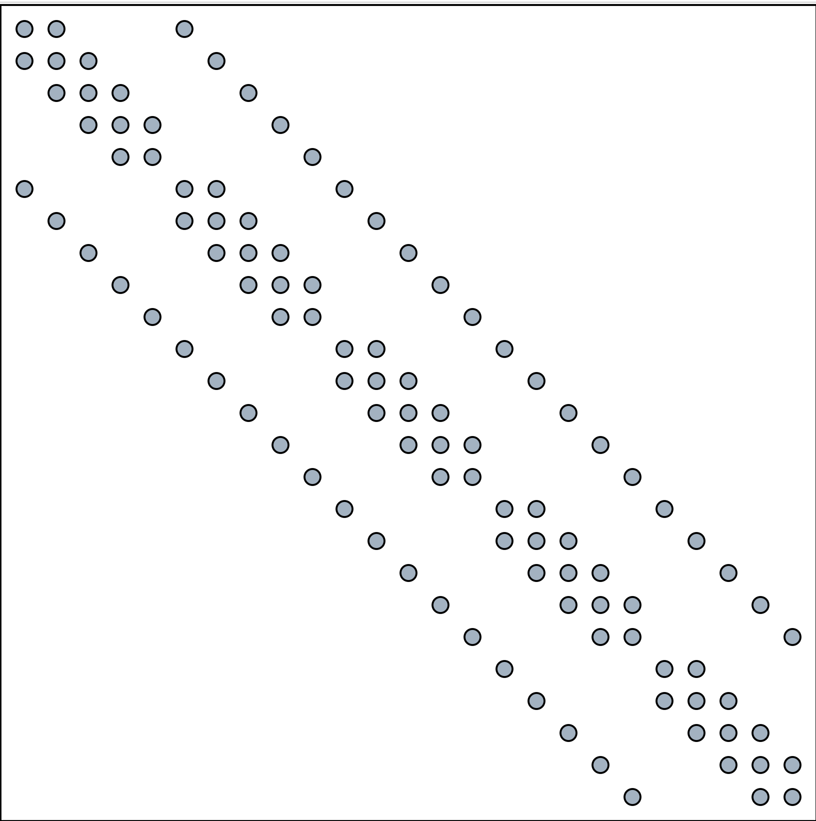
No fill-in !

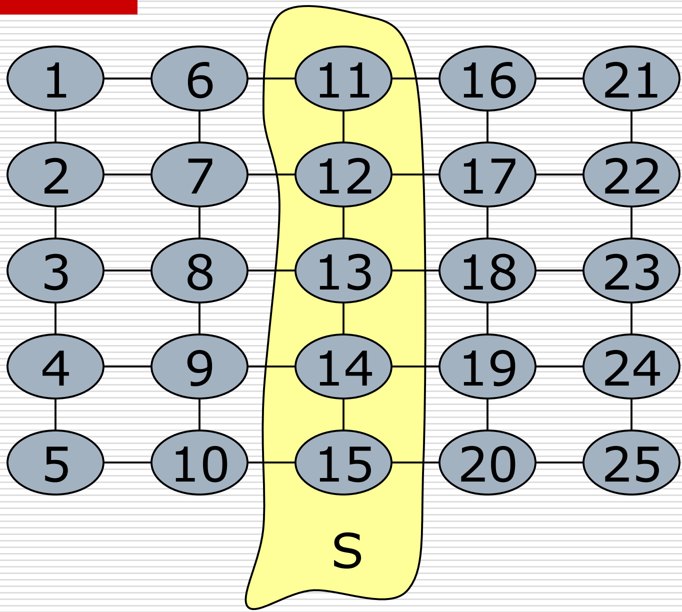


Ordering

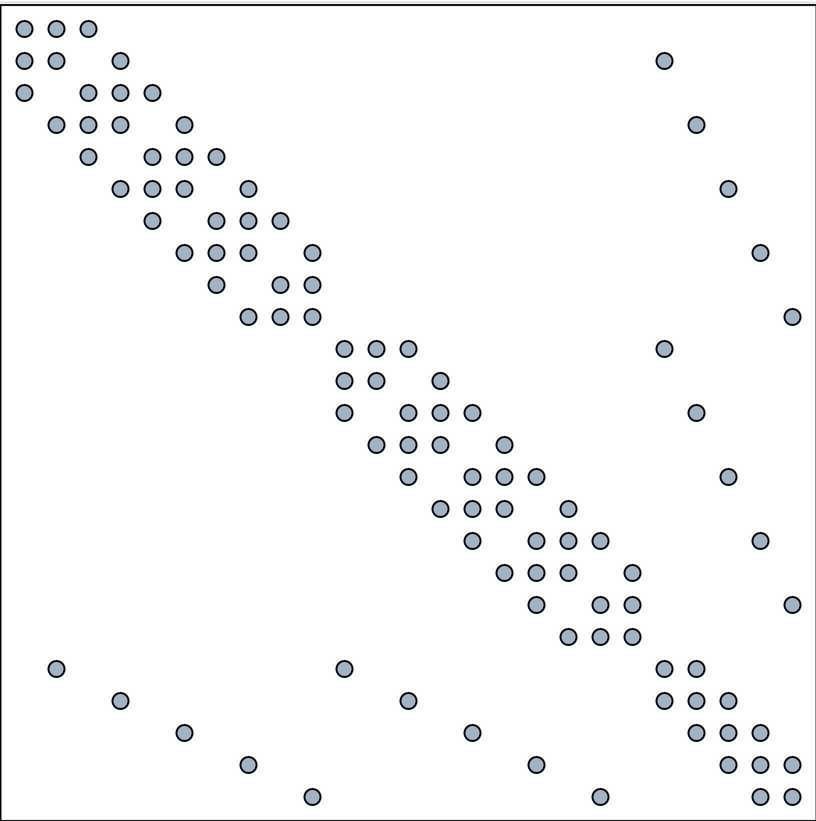
- Another ordering is nested dissection (divide-and-conquer)
- Find separator S of nodes whose removal (along with edges) divides the graph into 2 disjoint pieces
- Variables in each pieces are numbered contiguously and variables in S are numbered last
- Leads to bordered block diagonal non-zero pattern
- Can be applied recursively
- Can be parallelized using divide-and-conquer approach

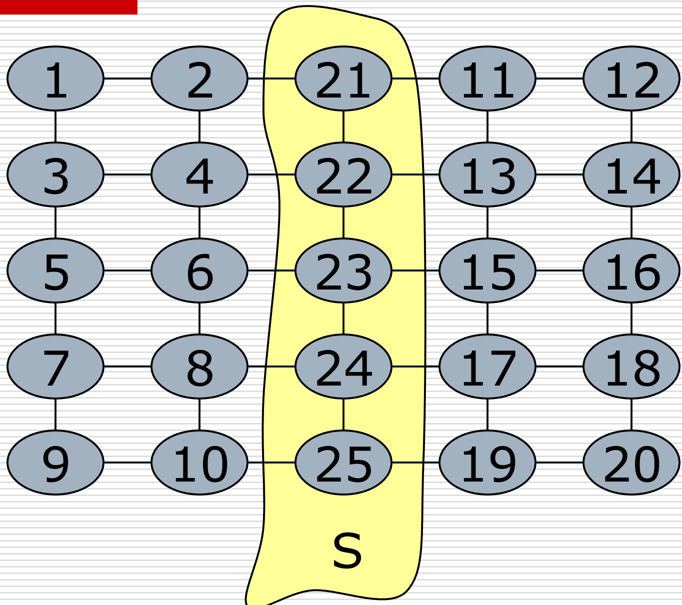
Nested Dissection Illustration



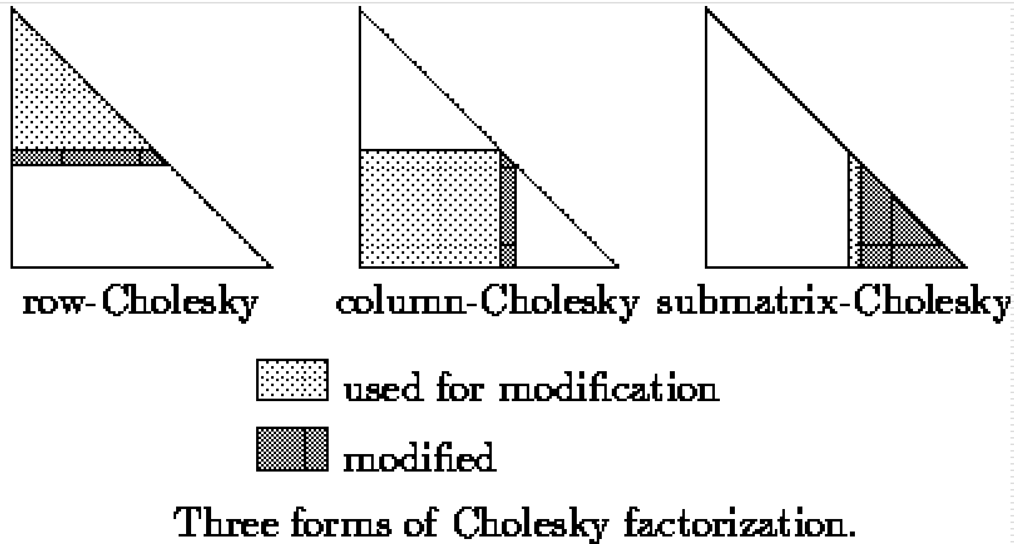


Nested Dissection Illustration





Numerical Factorization



Three forms of Cholesky factorization.

cmod(j, k): modification of column j by column k, k < j

cdiv(j) : division of column j by a scalar

Algorithms

Sparse column-Cholesky factorization

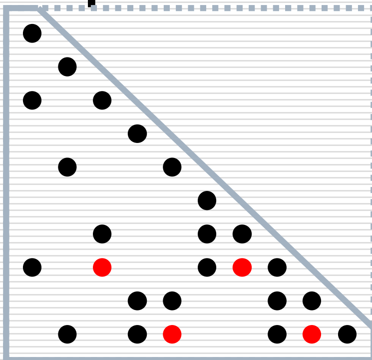
$$
\begin{aligned}
&\textit{for } j = 1, n \\
&\quad \textit{for } k \in \mathrm{Struct}(L_{j*}) \\
&\quad\quad \mathrm{cmod}(j, k) \\
&\quad \mathrm{cdiv}(j)
\end{aligned}
$$

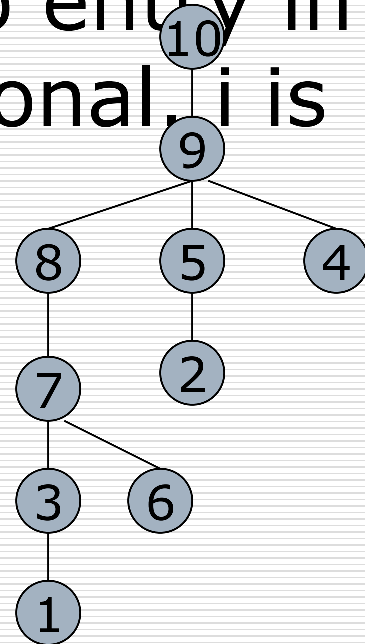
Sparse submatrix-Cholesky factorization

$$
\begin{aligned}
&\textit{for } k = 1, n \\
&\quad \mathrm{cdiv}(k) \\
&\quad \textit{for } j \in \mathrm{Struct}(L_{*k}) \\
&\quad\quad \mathrm{cmod}(j, k)
\end{aligned}
$$

Elimination Tree

- T(A) has an edge between two vertices i and j, with i > j, if i = p(j), i.e., L(i, j) is first non-zero entry in the jth column below diagonal. i is the parent of j.





Parallelization of Sparse Cholesky

- Most of the parallel algorithms are based on elimination trees
- Work associated with two disjoint subtrees can proceed independently
- Same steps associated with sequential sparse factorization
- One additional step: assignment of tasks to processors

Ordering in Parallel – Nested dissection

- Nested dissection can be carried in parallel
- Also leads to elimination trees that can be parallelized during subsequent factorizations
- But parallelization only in the later levels of dissection
- Can be applied to only limited class of problems

Nested Dissection Algorithms

- Use a graph partitioning heuristic to obtain a small edge separator of the graph
- Transform the small edge separator into a small node separator
- Number nodes of the separator last and recursively apply

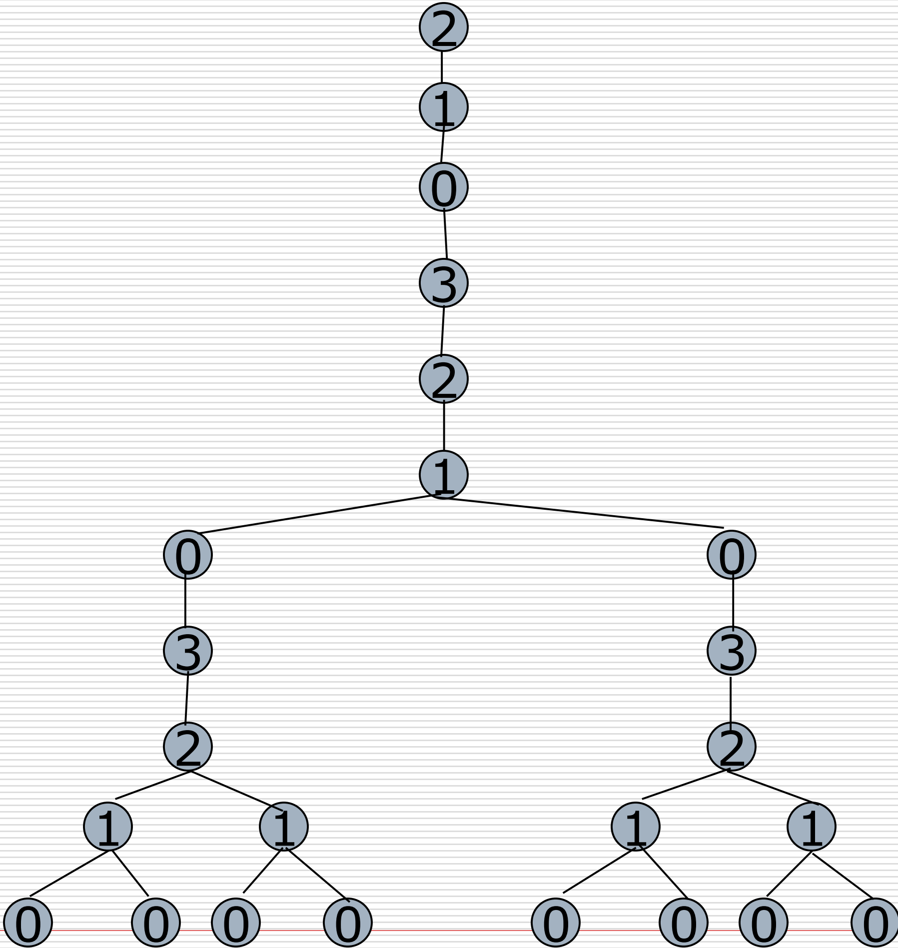
Kernighan-Lin for ND

- Form a random initial partition
- Form edge separator by applying K-L to form partitions P1 and P2
- Let V1 in P1 such that nodes in V1 incident on atleast one edge in the separator set. Similarly V2
- V1 U V2 (wide node separator),
- V1 or V2 (narrow node separator) by Gilbert and Zmijewski (1987)

Step 2: Mapping Problems on to processors

- Based on elimination trees
- Various strategies to map columns to processors based on elimination trees.
- Two algorithms:
  - Subtree-to-Subcube
  - Bin-Pack by Geist and Ng

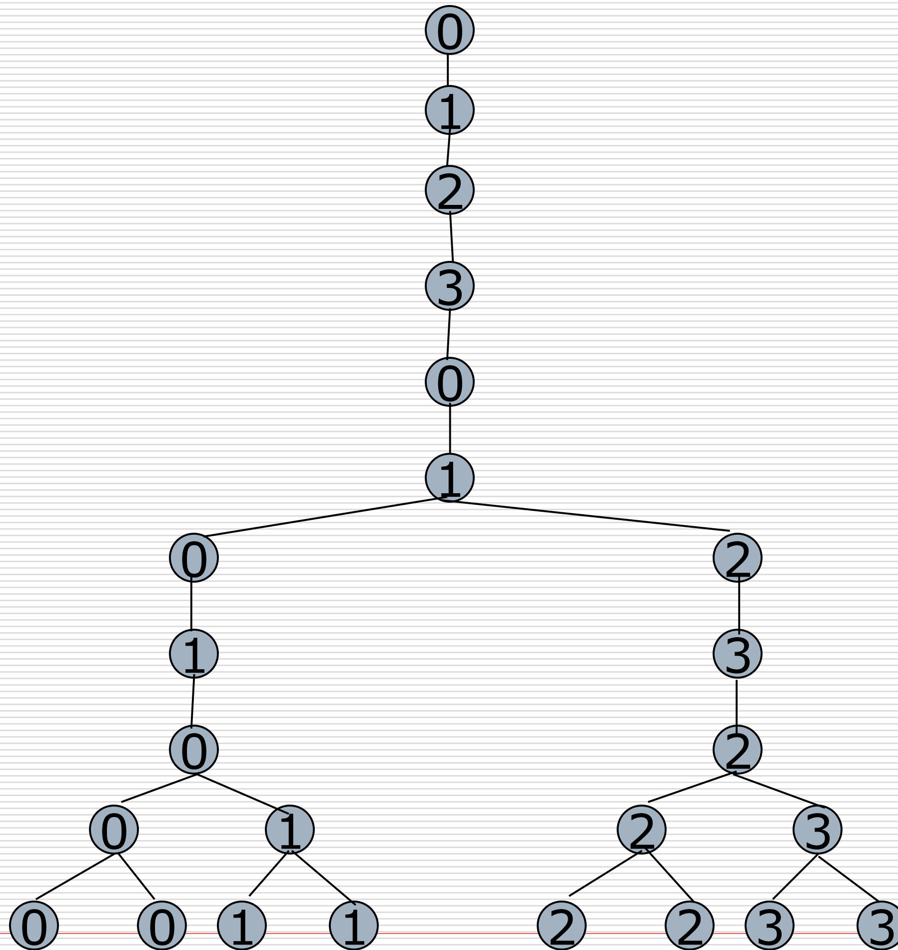
Naïve Strategy



Strategy 2 – Subtree-to-subcube mapping

- Select an appropriate set of P subtrees of the elimination tree, say T0, T1…
- Assign columns corresponding to Ti to Pi
- Where two subtrees merge into a single subtree, their processor sets are merged together and wrap-mapped onto the nodes/columns of the separator that begins at that point.
- The root separator is wrap-mapped onto the set of all processors.

Strategy 2



Strategy 3: Bin-Pack (Geist and Ng)

- Subtree-to-subcube mapping is not good for unbalanced trees
- Try to find disjoint subtrees
- Map the subtrees to p bins based on first-fit-decreasing bin-packing heuristic
  - Subtrees are processed in decreasing order of workloads
  - A subtree is packed into the current lightest bin
- Weight imbalance, α – ratio between lightest and heaviest bin
- If α >= user-specified tolerance, γ, stop
- Else explore the heaviest subtree from the heaviest bin and split into subtrees. These subtrees are then mapped to p bins and repacked using bin-packing again
- Repeat until α >= γ or the largest subtree cannot be split further
- Load balance based on user-specified tolerance
- For the remaining nodes from the roots of the subtrees to the root of the tree, wrap map.

Parallel Numerical Factorization – Submatrix Cholesky

Sparse submatrix-Cholesky factorization

$$
\begin{aligned}
&\textit{for } k = 1,\ n \\
&\quad \mathrm{cdiv}(k) \\
&\quad \textit{for } j \in \mathrm{Struct}(L_{*k}) \\
&\qquad \mathrm{cmod}(j, k)
\end{aligned}
\quad \Bigg\} \ \text{Tsub(k)}
$$

Tsub(k) is partitioned into various subtasks Tsub(k,1),…,Tsub(k,P) where

Tsub(k,p) := {cmod(j,k) | j C Struct(L_{*k}) ∩ mycols(p)}

Definitions

- mycols(p) – set of columns owned by p
- map[k] – processor containing column k
- procs(L_{*k}) = {map[j] | j in Struct(L_{*k})}

Parallel Submatrix Cholesky

```
for j in mycols(p) do
  if j is a leaf node in T(A) do
    cdiv(j)
    send L*j to the processors in procs(L*j)
    mycols(p) := mycols(p) – {j}

while mycols(p) ≠ 0 do
    receive any column of L, say L*k
    for j in Struct(L*k) ∩ mycols(p) do
      cmod(j, k)
      if column j required no more cmod's do
        cdiv(j)
        send L*j to the processors in procs(L*j)
        mycols(p) := mycols(p) – {j}
```

Disadvantages:

1. Communication is not localized

Parallel Numerical Factorization – Sub column Cholesky

Sparse column-Cholesky factorization

$$
\begin{aligned}
&\textit{for } j = 1,\ n\\
&\quad \textit{for } k \in \mathrm{Struct}(L_{j*})\\
&\quad\quad \mathrm{cmod}(j,k)\\
&\quad \mathrm{cdiv}(j)
\end{aligned}
$$

Tcol(j) is partitioned into various subtasks Tcol(j,1),…,Tcol(j,P) where

Tcol(j,p) aggregates into a single update vector every update vector u(j,k) for which k C Struct(L_{j*}) ∩ mycols(p)

Definitions

- mycols(p) – set of columns owned by p
- map[k] – processor containing column k
- procs(L_{j*}) = {map[k] | k in Struct(L_{j*})}
- u(j, k) – scaled column accumulated into the factor column by cmod(j, k)

Parallel Sub column Cholesky

```
for j:= 1 to n do
  if j in mycols(p) or Struct(L_j*) ∩ mycols(p) ≠ 0 do
    u = 0
    for k in Struct(L_j*) ∩ mycols(p) do
      u = u + u(j,k)
    if map[j] ≠ p do
      send u to processor q = map[j]
   else
      incorporate u into the factor column j
      while any aggregated update column for column j remains unreceived do
        receive in u another aggregated update column for column j
        incoprporate u into the factor column j
      cdiv(j)
```

Has uniform and less communication than sub matrix version for subtree-subcube mapping

A refined version – compute-ahead fan-in

- The previous version can lead to processor idling due to waiting for the aggregates for updating column j
- Updating column j can be mixed with compute-ahead tasks:

1. Aggregate u(i, k) for i > j for each completed column k in Struct(L_{i*}) ∩ mycols(p)
2. Receive aggregate update column for i > j and incorporate into factor column i

Sparse Iterative Methods

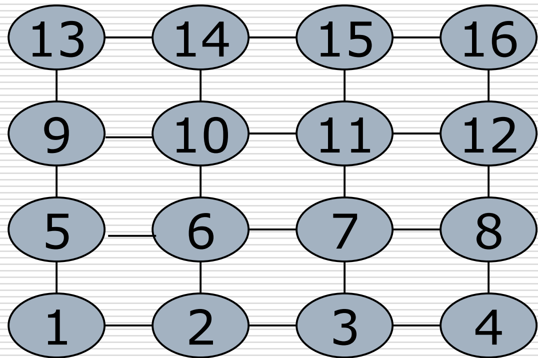
Iterative & Direct methods – Pros and Cons.

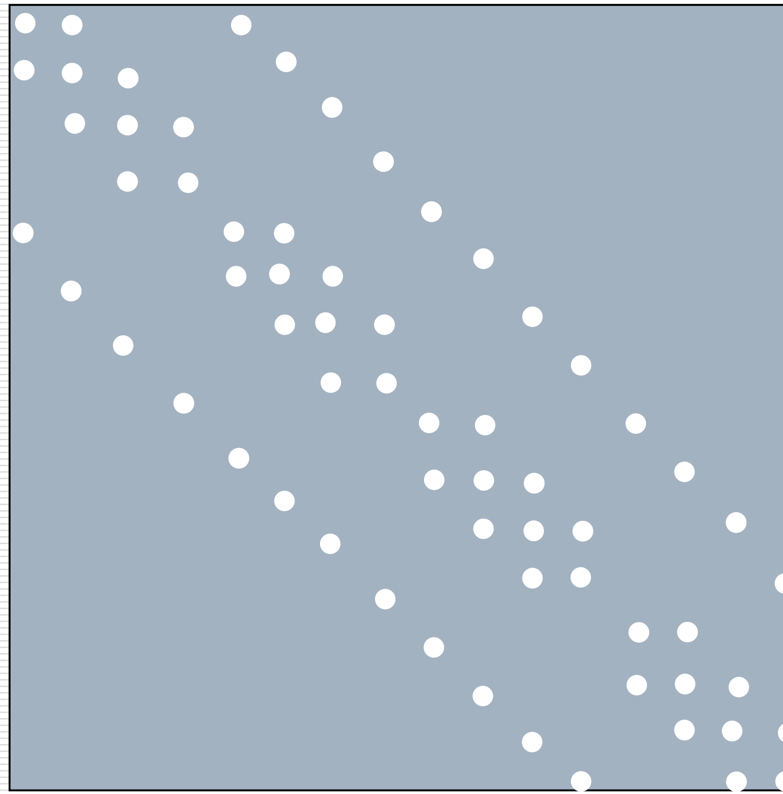
- Iterative methods do not give accurate results.
- Convergence cannot be predicted
- But absolutely no fills.

Parallel Jacobi, Gauss-Seidel, SOR

- For problems with grid structure (1-D, 2-D etc.), Jacobi is easily parallelizable
- Gauss-Seidel and SOR need recent values. Hence ordering of updates and sequencing among processors
- But Gauss-Seidel and SOR can be parallelized using red-black ordering or checker board

2D Grid example

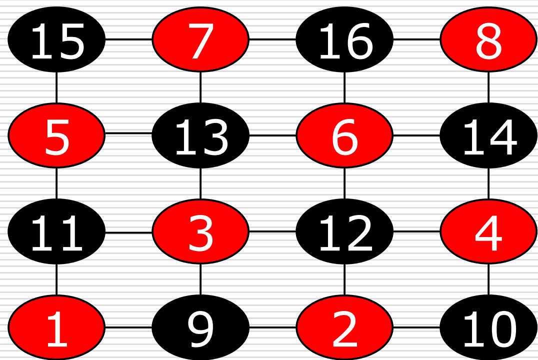


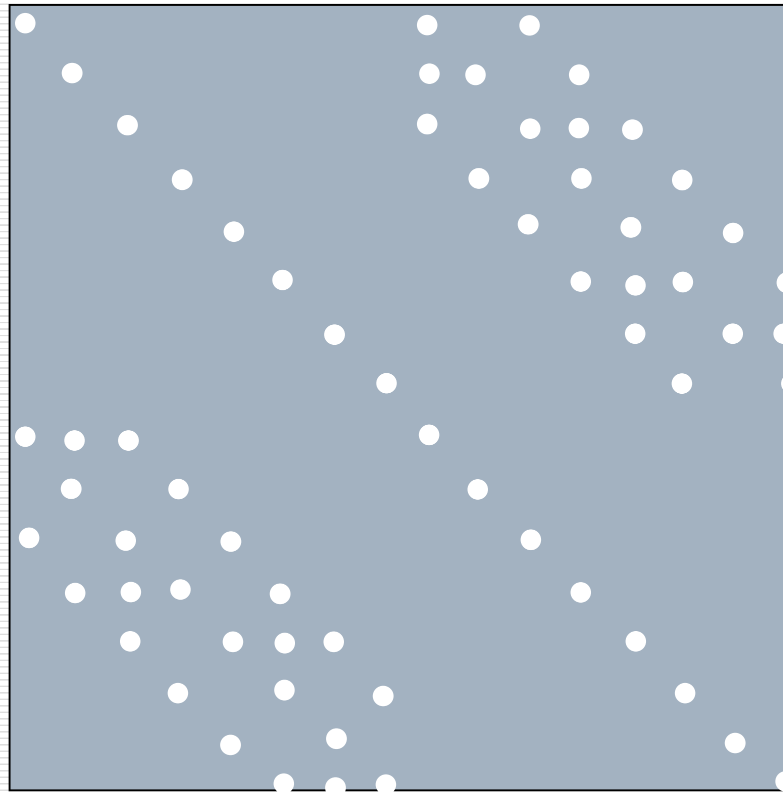


Red-Black Ordering

- Color alternate nodes in each dimension red and black
- Number red nodes first and then black nodes
- Red nodes can be updated simultaneously followed by simultaneous black nodes updates

2D Grid example – Red Black Ordering





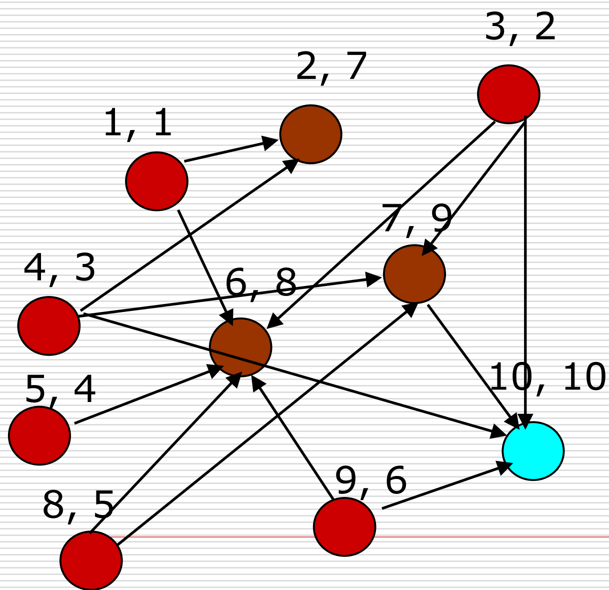
- In general, reordering can affect convergence

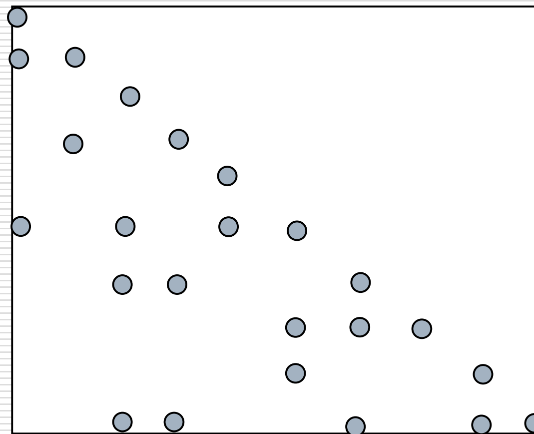
Graph Coloring

- In general multi-colored graph coloring Ordering for parallel computing of Gauss-Seidel and SOR
- Graph coloring can also be used for parallelization of triangular solves
- The minimum number of parallel steps in triangular solve is given by the chromatic number of symmetric graph
- Unknowns corresponding to nodes of same color are solved in parallel; computation proceeds in steps
- Thus permutation matrix, P based on graph color ordering

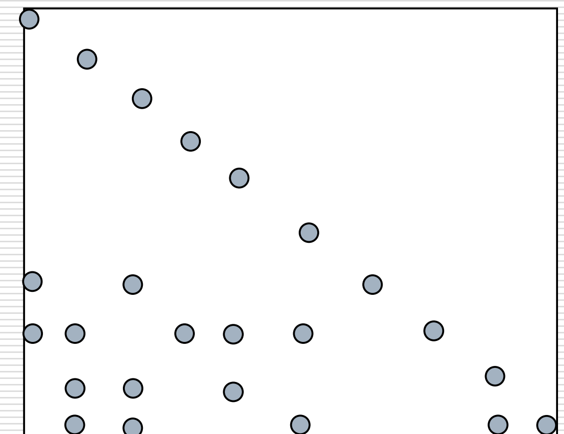
Parallel Triangular Solve based on Multi-Coloring

- Unknowns corresponding to the vertices of same color can be solved in parallel
- Thus parallel triangular solve proceeds in steps equal to the number of colors





Original Order



New Order

References in Graph Coloring

- M. Luby. A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing. 15(4)1036-1054 (1986)
- M.T.Jones, P.E. Plassmann. A parallel graph coloring heuristic. SIAM journal of scientific computing, 14(3): 654-669, May 1993
- L. V. Kale and B. H. Richards and T. D. Allen. Efficient Parallel Graph Coloring with Prioritization, Lecture Notes in Computer Science, vol 1068, August 1995, pp 190-208. Springer-Verlag.
- A.H. Gebremedhin, F. Manne, Scalable parallel graph coloring algorithms, Concurrency: Practice and Experience 12 (2000) 1131-1146.
- A.H. Gebremedhin , I.G. Lassous , J. Gustedt , J.A. Telle, Graph coloring on coarse grained multicomputers, Discrete Applied Mathematics, v.131 n.1, p.179-198, 6 September 2003

References

- M.T. Heath, E. Ng, B.W. Peyton. Parallel Algorithms for Sparse Linear Systems. SIAM Review. Vol. 33, No. 3, pp. 420-460, September 1991.
- A. George, J.W.H. Liu. The Evolution of the Minimum Degree Ordering Algorithm. SIAM Review. Vol. 31, No. 1, pp. 1-19, March 1989.
- J. W. H. Liu. Reordering sparse matrices for parallel elimination. Parallel Computing 11 (1989) 73-91

References

- Anshul Gupta, Vipin Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. Conference on High Performance Networking and Computing. Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM).
- P. Raghavan. Efficient Parallel Triangular Solution Using Selective Inversion. Parallel Processing Letters, Vol. 8, No. 1, pp. 29-40, 1998

References

- Joseph W. H. Liu. The Multifrontal Method for Sparse Matrix Factorization. SIAM Review. Vol. 34, No. 1, pp. 82-109, March 1992.
- Gupta, Karypis and Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. TPDS. 1997.