RC25076 (W1011-074) November 10, 2010
Computer Science

IBM Research Report

Sparse Direct Methods

Anshul Gupta
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Sparse Direct Methods

Anshul Gupta
Business Analytics and Mathematical Sciences
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
anshul@watson.ibm.com

November 9, 2010

1 Definition

Direct methods for solving linear systems of the form $Ax = b$ are based on computing $A = LU$, where L and U are lower and upper triangular, respectively. Computing the triangular factors of the coefficient matrix A is also known as *LU decomposition*. Following the factorization, the original system is trivially solved by solving the triangular systems $Ly = b$ and $Ux = y$. If A is symmetric, then a factorization of the form $A = LL^T$ or $A = LDL^T$ is computed via *Cholesky factorization*, where L is a lower triangular matrix (unit lower triangular in the case of $A = LDL^T$ factorization) and D is a diagonal matrix. One set of common formulations of LU decomposition and Cholesky factorization for dense matrices are shown in Figures 1 and 2, respectively. Note that other mathematically equivalent formulations are possible by rearranging the loops in these algorithms. These algorithms must be adapted for sparse matrices, in which a large fraction of entries are zero. For example, if $A[j, i]$ in the division step is zero, then this operation need not be performed. Similarly, the update steps can be avoided if either $A[j, i]$ or $A[i, k]$ ($A[k, i]$ if A is symmetric) is zero.

When A is sparse, the triangular factors L and U typically have nonzero entries in many more locations than A does. This phenomenon is known as *fill-in*, and results in a superlinear growth in the memory and time requirements of a direct method to solve a sparse system with respect to the size of the system. Despite a high memory requirement, direct methods are often used in many real applications due to their generality and robustness. In applications requiring solutions with respect to several right-hand side vectors and the same coefficient matrix, direct methods are often the solvers of choice because the one-time cost of factorization can be amortized over several inexpensive triangular solves.

2 Discussion

The direct solution of a sparse linear system typically involves four phases. The two computational phases, *factorization* and *triangular solutions* have already been mentioned. The number of nonzeros in the factors and sometimes their numerical properties are functions of the initial permutation of the rows and columns of the coefficient matrix. In many parallel formulations of sparse factorization, this permutation can also have an effect on load balance. The first step in the direct solution of a sparse linear system, therefore, is to apply heuristics to compute a desirable permutation the matrix. This step is known as *ordering*. A sparse matrix can be viewed

```
1.  begin LU_Decomp (A, n)
2.      for i = 1, n
3.          for j = i + 1, n
4.              A[j, i] = A[j, i]/A[i, i]; /* division step, computes column i of L */
5.          end for
6.          for k = i + 1, n
7.              for j = i + 1, n
8.                  A[j, k] = A[j, k] − A[j, i] × A[i, k]; /* update step */
9.              end for
10.         end for
11.     end for
12. end LU_Decomp
```

Figure 1: A simple column-based algorithm for LU decomposition of an $n \times n$ dense matrix A. The algorithm overwrites A by L and U such that $A = LU$, where L is unit lower triangular and U is upper triangular. The diagonal entries after factorization belong to U; the unit diagonal of L is not explicitly stored.

as the adjacency matrix of a graph. Ordering heuristics typically use the graph view of the matrix and label the vertices in a particular order that is equivalent to computing a permutation of the coefficient matrix with desirable properties. In the second phase, known as *symbolic factorization*, the nonzero pattern of the factors is computed. Knowing the nonzero pattern of the factors before actually computing them is useful for several reasons. The memory requirements of numerical factorization can be predicted during symbolic factorization. With the number and locations of nonzeros known before hand, a significant amount of indirect addressing can be avoided during numerical factorization, thus boosting performance. In a parallel implementation, symbolic factorization helps in the distribution of data and computation among processing units. The ordering and symbolic factorization phases are also referred to as preprocessing or analysis steps. A fifth phase of *iterative refinement* is sometimes used after the solution phase to improve the accuracy of the solution.

Of the four phases, numerical factorization typically consumes the most memory and time. Many applications involve factoring several matrices with different numerical values but the same sparsity structure. In such cases, some or all of the results of the ordering and symbolic factorization steps can be reused. This is also advantageous for parallel sparse solvers because parallel ordering and symbolic factorization are typically less scalable. Amortization of the cost of these steps over several factorization steps helps maintain the overall scalability of the solver close to that of numerical factorization. The parallelization of the triangular solves is highly dependent on the parallelization of the numerical factorization phase. The parallel formulation of numerical factorization dictates how the factors are distributed among parallel tasks. The subsequent triangular solution steps must use a parallelization scheme that works on this data distribution, particularly in a distributed-memory parallel environment. Given its prominent role in the parallel direct solution of sparse linear system, the numerical factorization phase is the primary focus of this article.

The algorithms used for preprocessing and factoring a sparse coefficient matrix depend on the properties of the matrix, such as symmetry, diagonal dominance, positive definiteness, etc. However, there are common elements in most sparse factorization algorithms. Two of these, namely *task graphs* and *supernodes*, are key to

```
1.  begin Cholesky (A, n)
2.      for i = 1, n
3.          A[i, i] = sqrt(A[i, i]);
4.          for j = i + 1, n
5.              A[j, i] = A[j, i]/A[i, i]; /* division step, computes column i of L */
6.          end for
7.          for k = i + 1, n
8.              for j = k, n
9.                  A[j, k] = A[j, k] − A[j, i] × A[k, i]; /* update step */
10.             end for
11.         end for
12.     end for
13. end Cholesky
```

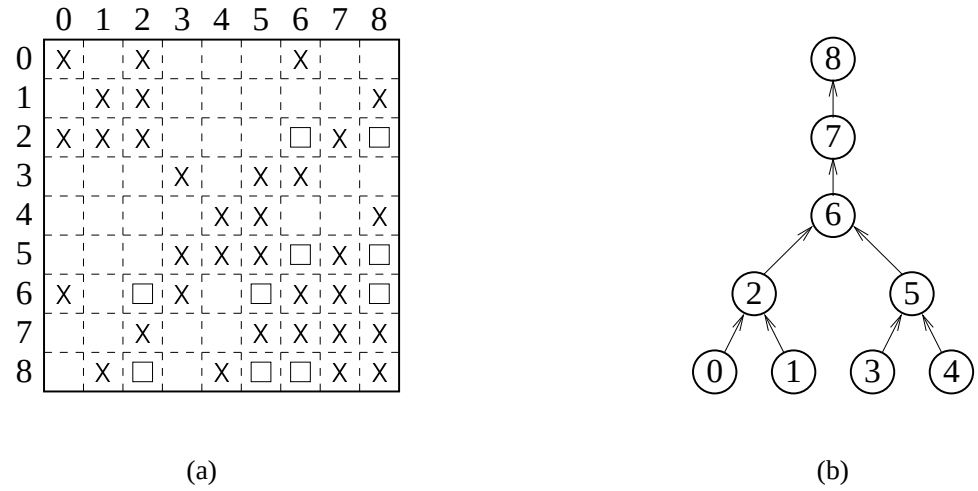
Figure 2: A simple column-based algorithm for Cholesky factorization of an $n \times n$ dense symmetric positive definite matrix A. The lower triangular part of A is overwritten by L, such that $A = LL^T$.

the discussion of parallel sparse matrix factorization of all types for both practical and pedagogical reasons.

2.1 Task graph model of sparse factorization

A parallel computation is usually the most efficient when running at the maximum possible level of granularity that ensures a good load-balance among all the processors. Dense matrix factorization is computationally rich and requires $O(n^3)$ operations for factoring an $n \times n$ matrix. Sparse factorization involves a much smaller overall number of operations per row or column of the matrix than its dense counterpart. The sparsity results in additional challenges, as well as additional opportunities to extract parallelism. The challenges are centered around finding ways of orchestrating the unstructured computations in a load-balanced fashion and of containing the overheads of interaction between parallel tasks in the face of a relatively small number or operations per row or column of the matrix. The added opportunity for parallelism results from the fact that, unlike the dense algorithms of Figures 1 and 2, the columns of the factors in the sparse case do not need to be computed one after the other. Note that in the algorithms shown in Figures 1 and 2, row and column i are updated by rows and columns $1 \ldots i-1$. In the sparse case, column i is updated by a column $j < i$ only if $U[j, i] \neq 0$, and a row i is updated by a row $j < i$ only if $L[i, j] \neq 0$. Therefore, as the sparse factorization begins, the division step can proceed in parallel for all columns i for which $A[i, j] = 0$ and $A[j, i] = 0$ for all $j < i$. Similarly, at any stage in the factorization process, there could be large pool of columns that are ready of the division step. Any unfactored column i would belong to this pool iff all columns $j < i$ with a nonzero entry in row i of L and all rows $j < i$ with a nonzero entry in column i of U have been factored.

A task dependency graph is an excellent tool for capturing parallelism and the various dependencies in sparse matrix factorization. It is a directed acyclic graph (DAG) whose vertices denote tasks and the edges specify the dependencies among the tasks. A task is associated with each row and column (column only in the symmetric case) of the sparse matrix to be factored. The vertex i of the task graph denotes the task responsible for computing column i of L and row i of U. A task is ready for execution if and only if all tasks with incoming



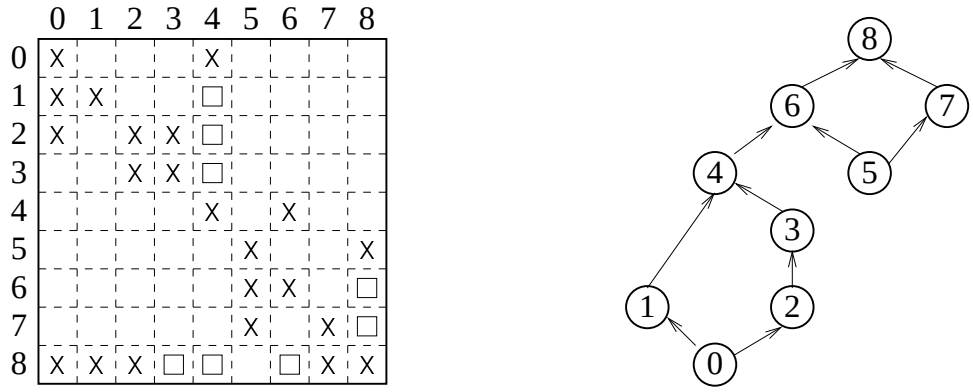
(a) (b)

Figure 3: A structurally symmetric sparse matrix and its elimination tree. An X indicates a nonzero entry in the original matrix and a box denotes a fill-in.

edges to it have completed. Task graphs are often explicitly constructed during symbolic factorization to guide the numerical factorization phase. This permits the numerical factorization to avoid expensive searches in order to determine which tasks are ready for execution at any given stage of the parallel factorization process. The task graphs corresponding to matrices with a symmetric structure are trees and are known as *elimination trees* in the sparse matrix literature.

Figure 3 shows the elimination tree for a structurally symmetric sparse matrix and Figure 4 shows the task DAG for a structurally unsymmetric matrix. Once a task graph is constructed, then parallel factorization (and even parallel triangular solution) can be viewed as the problem of scheduling the tasks onto parallel processes or threads. Static scheduling is generally preferred in a distributed-memory environment and dynamic scheduling in a shared-memory environment. The shape of the task graph is a function of the initial permutation of rows and columns of the sparse matrix, and is therefore determined by the outcome of the ordering phase. Figure 5 shows the elimination tree corresponding to the same matrix as in Figure 3(a), but with a different initial permutation. The structure of the task DAG usually affects how effectively it can be scheduled for parallel factorization. For example, it may be intuitively recognizable to readers that the elimination tree in Figure 3 is more amenable to parallel scheduling than the tree corresponding to a different permutation of the same matrix in Figure 5. Figure 6 illustrates that the matrices in Figures 3 and 5 have the same underlying graph. The only difference is in the labeling of the vertices of the graph, which results in a different permutation of the rows and columns of the matrix, different amount of fill-in, and different shapes of task graphs. In general, long and skinny task graphs result in limited parallelism and a long critical path. Short and broad task graphs have a high degree of parallelism and shorter critical paths.

Since the shape, and hence the amenability to efficient parallel scheduling of the task graph is sensitive to ordering, heuristics that result in balanced and broad task graphs are preferred for parallel factorization. The best known ordering heuristic in this class is called *nested dissection.* Nested dissection is based on recursively computing balanced bisections of a graph by finding small vertex separators. The vertices in the two disconnected partitions of the graph are labeled before the vertices of the separator. The same heuristic is applied recursively for labeling the vertices of each partition. The ordering in Figure 3 is actually based on nested dissection. Note that the vertex set 6, 7, 8 forms a separator, dividing the graph into two disconnected components,



(a) (b)

Figure 4: An unsymmetric sparse matrix and the corresponding task DAG. An X indicates a nonzero entry in the original matrix and a box denotes a fill-in.

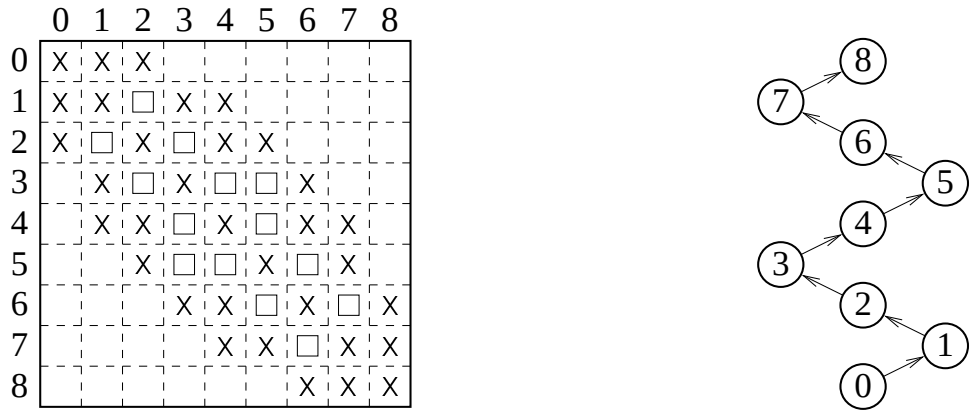
0, 1, 2 and 3, 4, 5. Within the two components, vertices 2 and 5 are the separators, and hence have the highest label in their respective partitions.

2.2 Supernodes

In sparse matrix terminology, a set of consecutive rows or columns that have the same nonzero structure is loosely referred to as a supernode. The notion of supernodes is crucial to efficient implementation of sparse factorization for a large class of sparse matrices arising in real applications.

Coefficient matrices in many applications have natural supernodes. In graph terms, there are sets of vertices with identical adjacency structures. Graphs like this can be compressed by having one *supervertex* represent the whole set of vertices with the same adjacency structures. When most vertices of a graph belong to supernodes and the supernodes are of roughly the same size (in terms of the number of vertices in them) with an average size of, say, m, then it can be shown that the compressed graph has $O(m)$ fewer vertices and $O(m^2)$ fewer edges than in the original graph. It can also be shown that an ordering of original graph can be derived from an ordering of the compressed graph, while preserving the properties of the ordering, by simply labeling the vertices of the original graph consecutively in the order of the supernodes of the compressed graph. Thus, the space and the time requirements of ordering can be dramatically reduced. This is particularly useful for parallel sparse solvers because parallel ordering heuristics often yield orderings of lower quality than their serial counterparts. For matrices with highly compressible graphs, it is possible to compute the ordering in serial with only a small impact on the overall scalability of the entire solver because ordering is performed on a graph with $O(m^2)$ fewer edges.

While the natural supernodes in the coefficient matrix, if any, can be useful during ordering, it is the presence of supernodes in the factors that have the biggest impact on the performance of the factorization and triangular solution steps. Although there can be multiple ways of defining supernodes in matrices with an unsymmetric structure, the most useful form involves groups of indices with identical nonzero pattern in the corresponding columns of L and rows of U. Even if there are no supernodes in the original matrix, supernodes in the factors are almost inevitable for matrices in most real applications. This is due to fill-in. Examples of supernodes



(a) (b)

Figure 5: A permutation of the sparse matrix of Figure 3(a) and its elimination tree.

in factors include indices 6–8 in Figure 3(a), indices 2–3 and 7–8 in Figure 4(a), and indices 4–5 and 6–8 in Figure 5. Some practitioners prefer to artificially increase the size (i.e., the number of member rows and columns) of supernodes by padding the rows and columns that have only slightly different nonzero patterns, so that they can be merged into the same supernode. The supernodes in the factors are typically detected and recorded as they emerge during symbolic factorization. In the remainder of this chapter, the term supernode refers to a supernode in the factors.

It can be seen from the algorithms in Figures 1 and 2 that there are two primary computations in a column-based factorization: the division step and the update step. A supernode-based sparse factorization too has the same two basic computation steps, except that these are now matrix operations on row/column blocks corresponding to the various supernodes.

Supernodes impart efficiency to numerical factorization and triangular solves because they permit floating point operations to be performed on dense submatrices instead of individual nonzeros, thus improving memory hierarchy utilization. Since rows and columns in supernodes share the nonzero structure, indirect addressing is minimized because the structure needs to be stored only once for these rows and columns. Supernodes help to increase the granularity of tasks, which is useful for improving computation to overhead ratio in a parallel implementation. The task graph model of sparse matrix factorization was introduced earlier with a task k defined as the factorization of row and column k of the matrix. With supernodes a task can be defined as the factorization of all rows and columns associated with a supernode. Actual task graphs in practical implementations of parallel sparse solvers are almost always supernodal task graphs.

Note that some applications, such as power grid analysis, in which the basis of the linear system is not a finite-element or finite-difference discretization of a physical domain, can give rise to sparse matrices that incur very little fill-in during factorization. The factors of these matrices may have very small supernodes.

2.3 An effective parallelization strategy

The task graphs for sparse matrix factorization have some typical properties that make scheduling somewhat different from traditional DAG scheduling. Note that the task graphs corresponding to irreducible matrices have a distinct root; i.e., one node that has no outgoing edges. This corresponds to the last (rightmost) supernode

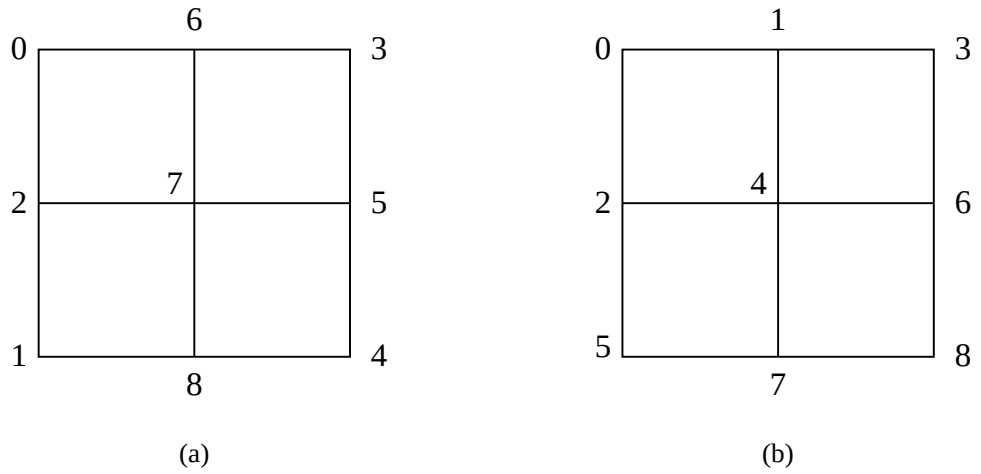


Figure 6: An illustration of the duality between graph vertex labeling and row/column permutation of a structurally symmetric sparse matrix. Grid (a), with vertices labeled based on nested dissection, is the adjacency graph of the matrix in Figure 3(a) and grid (b) is the adjacency graph of the matrix in Figure 5(a).

in the matrix. The number of member rows and columns in supernodes typically increases away from the leaves and towards the root of the task graph. The reason is that a supernode accumulates fill-in from all its predecessors in the task graph. As a result, the portions of the factors that correspond to task graph nodes with a large number of predecessors tend to get denser. Due to their larger supernodes, the tasks that are relatively close to the root tend to have more work associated with them. On the other hand, the width of the task graph shrinks close to the root. In other words, a typical task graph for sparse matrix factorization tends to have a large number of small independent tasks closer to the leaves, but a small number of large tasks closer to the root. An ideal parallelization strategy that would match the characteristics of the problem is as follows. Starting out, the relatively plentiful independent tasks at or near the leaves would be scheduled to parallel threads or processes. As tasks complete, other tasks become available and would be scheduled similarly. This could continue until there are enough independent tasks to keep all the threads or processes busy. When the number of available parallel tasks becomes smaller than the number of available threads or processes, then the only way to keep the latter busy would be to utilize more than one of them per task. The number of threads or processes working on individual tasks would increase as the number of parallel tasks decreases. Eventually, all threads or processes would work on the root task. The computation corresponding to the root task is equivalent to factoring a dense matrix of the size of the root supernode.

2.4 Sparse factorization formulations based on task roles

So far in this article, the tasks have been defined somewhat ambiguously. There are multiple ways of defining the tasks precisely, which can result in different parallel implementations of sparse matrix factorization. Clearly, a task is associated with a supernode and is responsible for computing that supernode of the factors; i.e., performing the computation equivalent to the division steps in the algorithms in Figures 1 and 2. However, a task does not own all the data that is required to compute the final values of its supernode's rows and columns. The data for performing the update steps on a supernode may be contributed by many other supernodes. Based on the tasks' responsibilities, sparse LU factorization has traditionally been classified into three categories,

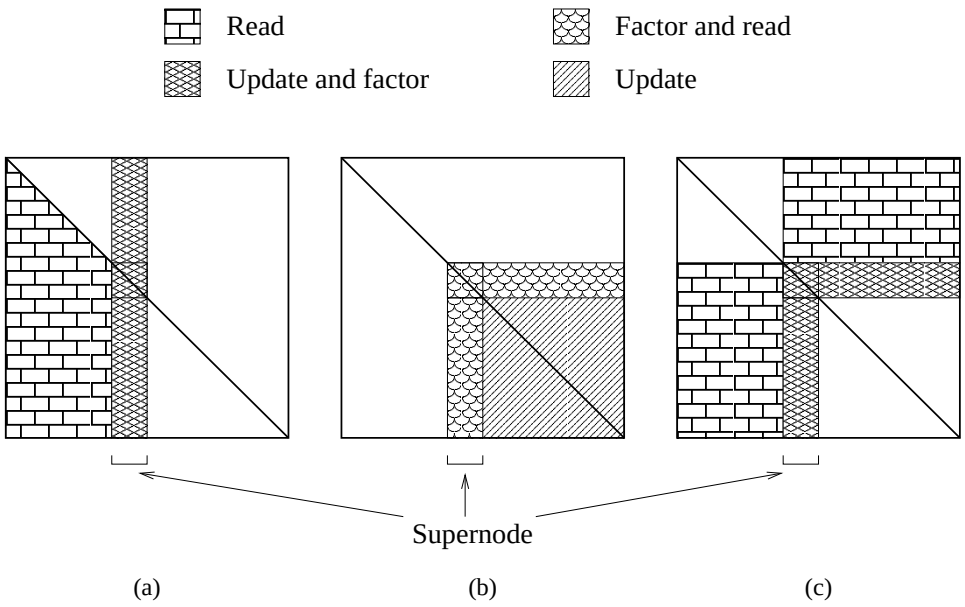


Figure 7: The left-looking (a), right-looking (b), and Crout (c) variations of sparse LU factorization. Different patterns indicate the parts of the matrix that are read, updated, and factored by the task corresponding to a supernode. Blank portions of the matrix are not accessed by this task.

namely *left-looking*, *right-looking*, and *Crout*. These variations are illustrated in Figure 7. The traditional left-looking variant uses nonconforming supernodes made up of columns of both L and U, which are not very common in practice. In this variant, a task is responsible for gathering all the data required for its own columns from other tasks and for updating and factoring its columns. The left-looking formulation is rarely used in modern high-performance sparse direct solvers. In the right-looking variation of sparse LU, a task factors the supernode that it owns and performs all the updates that use the data from this supernode. In the Crout variation, a tasks is responsible for updating and factoring the supernode that it owns. Only the right-looking and Crout variants have symmetric counterparts.

A fourth variation, known as the *multifrontal method*, incorporates elements of both right-looking and Crout formulations. In the multifrontal method, the task that owns a supernode computes its own contribution to updating the remainder of the matrix (like the right-looking formulation), but does not actually apply the updates. Each task is responsible for collecting all relevant precomputed updates and applying them to its supernode (like the Crout formulation) before factoring the supernode. The supernode data and its update contribution in the multifrontal method is organized into small dense matrices called *frontal matrices.* Integer arrays maintain a mapping of the local contiguous indices of the frontal matrices to the global indices of the sparse factor matrices. Figure 8 illustrates the complete supernodal multifrontal Cholesky factorization of the symmetric matrix shown in Figure 3(a). Note that, since rows and columns with indices 6–8 form a supernode, there would be only one task (Figure 8(g)) corresponding to these in the supernodal task graph (elimination tree).

When a task is ready for execution, it first constructs its frontal matrix by accumulating contributions from the frontal matrices of its children and from the coefficient matrix. It then factors its supernode, which is the portion of the frontal matrix that is shaded in Figure 8. After factorization, the unshaded portion (this submatrix of a frontal matrix is called the *update matrix*) is updated based on the update step of the algorithm in Figure 2.

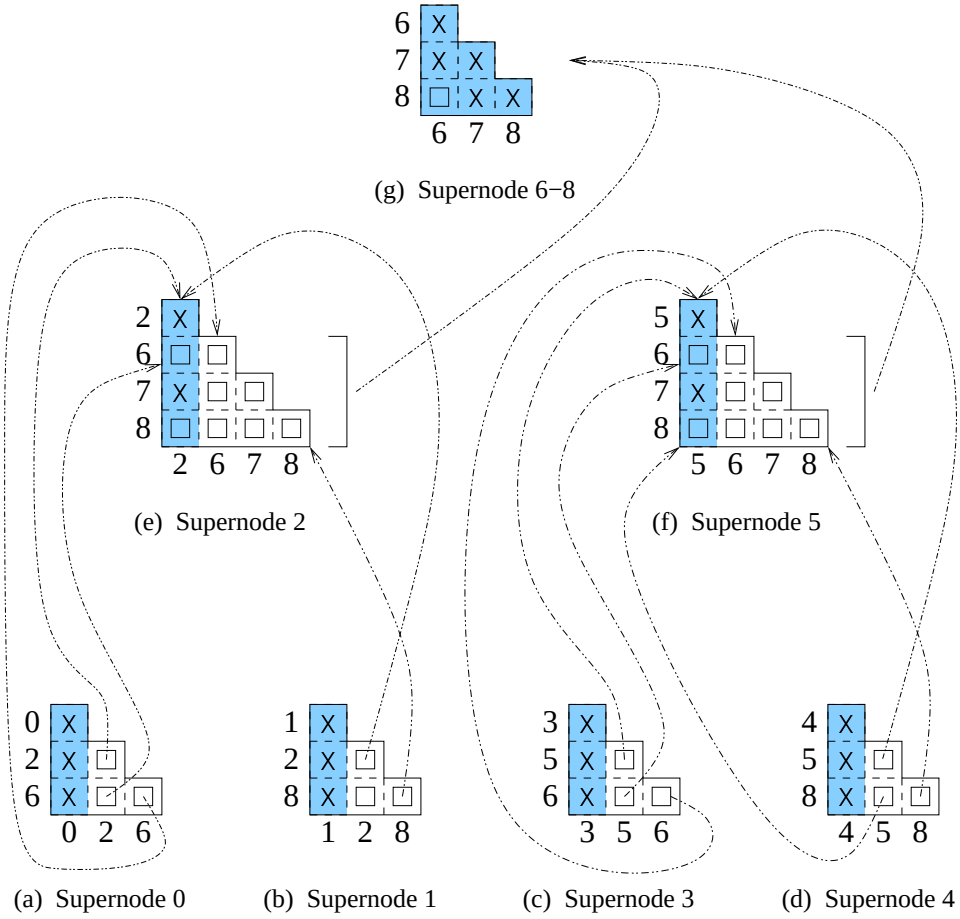


Figure 8: Frontal matrices and data movement among them in the supernodal multifrontal Cholesky factorization of the sparse matrix shown in Figure 3(a).

The update matrix is then used by the parent task to construct its frontal matrix.

Note that Figure 8 illustrates a symmetric multifrontal factorization; hence, the frontal and update matrices are triangular. For general LU decomposition, these matrices would be square or rectangular. In symmetric multifrontal factorization, a child's update matrix in the elimination tree contributes only to its parent's frontal matrix. The task graph for general matrices is usually not a tree, but a DAG, as shown in Figure 4. Apart from the shape of the frontal and update matrices, unsymmetric pattern multifrontal method differs from its symmetric counterpart in two other ways. First, an update matrix can contribute to more than one frontal matrices. Secondly, the frontal matrices receiving data from an update matrix can belong to the contributing supernode's ancestors (not necessarily parents) in the task graph.

The multifrontal method is often the formulation of choice for highly parallel implementations of sparse matrix factorization. This is because of its natural data locality (most of the work of the factorization is performed in the well-contained dense frontal matrices) and the ease of synchronization that it permits. In general, each supernode is updated by multiple other supernodes and it can potentially update many other supernodes during the course of factorization. If implemented naively, all these updates may require excessive locking and synchronization in a shared-memory environment or generate excessive message-traffic in a distributed environment. In the multifrontal method, the updates are accumulated and channeled along the paths from the

leaves of the task graph to its the root. This gives a manageable structure to the potentially haphazard interaction among the tasks.

Recall that the typical supernodal sparse factorization task graph is such that the size of tasks generally increases and the number of parallel tasks generally diminishes on the way to the root from the leaves. The multifrontal method is well suited for both task parallelism (close to the leaves) and data parallelism (close to the root). Larger tasks working on large frontal matrices close to the root can readily employ multiple threads or processes to perform parallel dense matrix operations, which not only have well understood data-parallel algorithms, but also a well developed software base.

2.5 Pivoting in parallel sparse LDL^T and LU factorization

The discussion in this article so far has focussed on the scenario in which the rows and columns of the matrix are permuted during the ordering phase and this permutation stays static during numerical factorization. While this assumption is valid for a large class of practical problems, there are applications that would generate matrices that could encounter a zero or a very small entry on the diagonal during the factorization process. This will cause the division step of the LU decomposition algorithm to fail or to result in numerical instability. For nonsingular matrices, this problem can be solved by interchanging rows and columns of the matrix by a process known as *partial pivoting*. When a small or zero entry is encountered at $A[i, i]$ before the division step, then row i is interchanged with another row j ($i < j \leq n$) such that $A[j, i]$ (which would occupy the location $A[i, i]$ after the interchange) is sufficiently greater in magnitude compared to other entries $A[k, i]$ ($i < k \leq n$, $k \neq j$). Similarly, instead of row i, column i could be exchanged with a suitable column j ($i < j \leq n$). In symmetric LDL^T factorization, both row and column i are interchanged simultaneously with a suitable row-column pair to maintain symmetry.

Until recently, it was believed that due to unpredictable changes in the structure of the factors due to partial pivoting, a priori ordering and symbolic factorization could not be performed, and these steps needed to be combined with numerical factorization. Keeping the analysis and numerical factorization steps separate has substantial performance and parallelization benefits, which would be lost if these steps are combined. Fortunately, modern parallel sparse solvers are able to perform partial pivoting and maintain numerical stability without mixing the analysis and numerical steps. The multifrontal method permits effective implementation of partial pivoting in parallel and keeps its effects as localized as possible.

Before computing a fill-reducing ordering, the rows or columns of the coefficient matrix are permuted such that the absolute value of the product of the magnitude of the diagonal entries is maximized. Special graph matching algorithms are used to compute this permutation. This step ensures that the diagonal entries of the matrix have relatively large magnitudes at the beginning of factorization. It has been observed that once the matrix has been permuted this way, in most cases, very few interchanges are required during the factorization process to keep it numerically stable. As a result, factorization can be performed using the static task graph and the static structures of the supernodes of L and U predicted by symbolic factorization. When an interchange is necessary, the resulting changes in the data structures are registered. Since such interchanges are rare, the resulting disruption and the overhead is usually well contained.

The first line of defense against numerical instability is to perform partial pivoting within a frontal matrix. Exchanging rows or columns within a supernode is local, and if all rows and columns of a supernode can be successfully factored by simply altering their order, then nothing outside the supernode is affected. Sometimes, a supernode cannot be factored completely by local interchanges. This can happen when all candidate rows

or columns for interchange have indices greater than that of the last row-column pair of the supernode. In this case, a technique known as *delayed pivoting* is employed. The unfactored rows and columns are simply removed from the current supernode and passed onto the parent (or parents) in the task graph. Merged with the parent supernode, these rows and columns have additional candidate rows and columns available for interchange, which increases the chances of their successful factorization. The process of upward migration of unsuccessful pivots continues until they are resolved, which is guaranteed to happen at the root supernode for a nonsingular matrix.

In the multifrontal framework, delayed pivoting simply involves adding extra rows and columns to the frontal matrices of the parents of supernode with failed pivots. The process is straightforward for the supernodes whose tasks are mapped onto individual threads or processes. For the tasks that require data-parallel involvement of multiple threads or processes, the extra rows and columns can be partitioned using the same strategy that is used to partition the original frontal matrix.

2.6 Parallel solution of triangular systems

As mentioned earlier, solving the original system after factoring the coefficient matrix involves solving a sparse lower triangular and a sparse upper triangular system. The task graph constructed for factorization can be used for the triangular solves too. For matrices with an unsymmetric pattern, a subset of edges of the task DAG may be redundant in each of the solve phases, but these redundant edges can be easily marked during symbolic factorization. Just like factorization, the computation for the lower triangular solve phase starts at the leaves of the task graph and proceeds towards the root. In the upper triangular solve phase, computation starts at the root and fans out towards the leaves (in other words, the direction of the edges in the task graph is effectively reversed).

3 Related Entries

- Cholesky factorization
- LU decomposition
- Multifrontal method
- Reordering

4 Bibliographic Notes

Books by George and Liu [6] and Duff, Erisman, and Reid [5] are excellent sources for a background on sparse direct methods. A comprehensive survey by Demmel, Heath, and van der Vorst [4] sums up the developments in parallel sparse direct solvers until the early 1990s. Some remarkable progress was made in the development of parallel algorithms and software for sparse direct methods during a decade starting in the early 1990s. Gupta et al. [9] developed the framework for highly scalable parallel formulations of symmetric sparse factorization based on the multifrontal method (see tutorial by Liu [12] for details), and recently demonstrated scalable performance of an industrial strength implementation of their algorithms on thousands of cores [10].

Demmel, Gilbert, and Li [3] developed one of the first scalable algorithms and software for solving unsymmetric sparse systems without partial pivoting. Amestoy et al. [2, 1] developed parallel algorithms and software that incorporated partial pivoting for solving unsymmetric systems with (either natural or forced) symmetric pattern. Hadfield [11] and Gupta [7] laid the theoretical foundation for a general unsymmetric pattern parallel multifrontal algorithm with partial pivoting, with the latter following up with a practical implementation [8].

References

[1] Patrick R. Amestoy, Iain S. Duff, Jacko Koster, and J. Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] Patrick R. Amestoy, Iain S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computational Methods in Applied Mechanical Engineering*, 184:501–520, 2000.

[3] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[4] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, pages 111–197, 1993.

[5] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, UK, 1990.

[6] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, NJ, 1981.

[7] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 24(2):529–552, 2002.

[8] Anshul Gupta. A shared- and distributed-memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communication, and Computing*, 18(3):263–277, 2007.

[9] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.

[10] Anshul Gupta, Seid Koric, and Thomas George. Sparse matrix factorization on massively parallel computers. In *SC09 Proceedings.*

[11] Steven M. Hadfield. *On the LU Factorization of Sequences of Identically Structured Sparse Matrices within a Distributed Memory Environment.* PhD thesis, University of Florida, Gainsville, FL, 1994.

[12] Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.