Recursive Approach in Sparse Matrix LU Factorization

Jack Dongarra[1], Victor Eijkhout[1], Piotr Łuszczek[1]

[1] University of Tennessee
Department of Computer Science
Knoxville, TN 37996-3450
USA
phone: (+865) 974-8295
fax: (+865) 974-8296

Address for correspondence:
Piotr Luszczek
Department of Computer Science
1122 Volunteer Blvd., Suite 203
Knoxville, TN 37996-3450
U.S.A.
phone: (+865) 974-8295
fax: (+865) 974-8296
e-mail: `luszczek@cs.utk.edu`

Abstract

This paper describes a recursive method for the LU factorization of sparse matrices. The recursive formulation of common linear algebra codes has been proven very successful in dense matrix computations. An extension of the recursive technique for sparse matrices is presented. Performance results given here show that the recursive approach may perform comparable to leading software packages for sparse matrix factorization in terms of execution time, memory usage, and error estimates of the solution.

1 Introduction

Typically, a system of linear equations has the form:

$$Ax = b, \tag{1}$$

where A is n by n real matrix ($A \in \mathbf{R}^{n\times n}$), and x and b are n-dimensional real vectors ($b,x \in \mathbf{R}^n$). The values of A and b are known and the task is to find x satisfying (1). In this paper, it is assumed that the matrix A is large (of order commonly exceeding ten thousand) and sparse (there are enough zero entries in A that it is beneficial to use special computational methods to factor the matrix rather than to use a dense code). There are two common approaches that are used to deal with such a case, namely, iterative [33] and direct methods [17].

Iterative methods, in particular Krylov subspace techniques such as the Conjugate Gradient algorithm, are the methods of choice for the discretizations of elliptic or parabolic partial differential equations where the resulting matrix is often guaranteed to be positive definite or close to it. However, when the linear system matrix is strongly unsymmetric or indefinite, as is the case with matrices originating from systems of ordinary differential equations or the indefinite matrices arising from shift-invert techniques in eigenvalue methods, one has to revert to direct methods which are the focus of this paper.

In direct methods, Gaussian elimination with partial pivoting is performed to find a solution of (1). Most commonly, the factored form of A is given by means of matrices L, U, P and Q such that:

$$LU = PAQ, \tag{2}$$

where:

- L is a lower triangular matrix with unitary diagonal,
- U is an upper triangular matrix with arbitrary diagonal,
- P and Q are row and column permutation matrices, respectively (each row and column of these matrices contains single a non-zero entry which is 1, and the following holds: $PP^T = QQ^T = I$, with I being the identity matrix).

A simple transformation of (1) yields:

$$(PAQ)Q^{-1}x = Pb, \tag{3}$$

which in turn, after applying (2), gives:

$$LU(Q^{-1}x) = Pb, \tag{4}$$

Solution to (1) may now be obtained in two steps:

$$Ly = Pb \tag{5}$$
$$U(Q^{-1}x) = y \tag{6}$$

and these steps are performed through forward/backward substitution since the matrices involved are triangular. The most computationally intensive part of solving (1) is the LU factorization defined by (2). This operation has computational complexity of order $O(n^3)$ when A is a dense matrix, as compared to $O(n^2)$ for the solution phase. Therefore, optimization of the factorization is the main determinant of the overall performance.

When both of the matrices P and Q of (2) are non-trivial, i.e. neither of them is an identity matrix, then the factorization is said to be using complete pivoting. In practice, however, Q is an identity matrix and this strategy is called partial pivoting which tends to be sufficient to retain numerical stability of the factorization, unless the matrix A is singular or nearly so. Moderate values of the condition number $\kappa = \|A^{-1}\| \cdot \|A\|$ guarantee a success for a direct method as opposed to matrix structure and spectrum considerations required for iterative methods.

When the matrix A is sparse, i.e. enough of its entries are zeros, it is important for the factorization process to operate solely on the non-zero entries of the matrix. However, new nonzero entries are introduced in the L and U factors which are not present in the original matrix A of (2). The new entries are referred to as fill-in and cause the number of non-zero entries in the factors (we use the notation $\eta(A)$ for the number of nonzeros in a matrix) to be (almost always) greater than that of the original matrix A: $\eta(L+U) \geq \eta(A)$. The amount of fill-in can be controlled with the matrix ordering performed prior to the factorization and consequently, for the sparse case, both of the matrices P and Q of (2) are non-trivial. Matrix Q induces the column reordering that minimizes fill-in and P permutes rows so that pivots selected during the Gaussian elimination guarantee numerical stability.

Recursion started playing an important role in applied numerical linear algebra with the introduction of Strassen's algorithm [6, 31, 36] which reduced the complexity of the matrix-matrix multiply operation from $O(n^3)$ to $O(n^{\log_2 7})$. Later on it was recognized that factorization codes may also be formulated recursively [3, 4, 21, 25, 27] and codes formulated this way perform better [38] than leading linear algebra packages [2] which apply only a blocking technique to increase performance. Unfortunately, the recursive approach cannot be applied directly for sparse matrices because the sparsity pattern of a matrix has to be taken into account in order to reduce both the storage requirements and the floating point operation count, which are the determining factors of the performance of a sparse code.

2 Dense Recursive LU factorization

Fig. 1 shows the classical LU factorization code which uses Gaussian elimination. Rearrangement of the loops and introduction of blocking techniques can significantly increase the performance of this code [2, 9]. However, the recursive formulation of the Gaussian elimination shown in Fig. 2 exhibits superior performance [25]. It does not contain any looping statements and most of the floating point operations are performed by the Level 3 BLAS [14] routines: `xTRSM()` and `xGEMM()`. These routines achieve near-peak MFLOP/s rates on modern computers with a deep memory hierarchy. They are incorporated in many vendor-optimized libraries, and they are used in the Atlas project [16] which automatically generates implementations tuned to specific platforms.

Yet another implementation of the recursive algorithm is shown in Fig. 3, this time without pivoting code. Experiments show that this code performs equally well as the code from Fig. 2. The experiments also provide indications that further performance improvements are possible, if the matrix is stored recursively [26]. Such a storage scheme is illustrated in Fig. 4. This scheme causes the dense submatrices to be aligned recursively in memory. The recursive algorithm from Fig. 3 then traverses the recursive matrix structure all the way down to the level of a single dense submatrix. At this point an appropriate computational routine is called (either BLAS or `xGETRF()`). Depending on the size of the submatrices (referred to as a block size [2]), it is possible to achieve higher execution rates than for the case when the matrix is stored in the column-major or row-major order. This observation made us adopt the code from Fig. 3 as the base for the sparse recursive algorithm presented below.

3 Sparse Matrix Factorization

Matrices originating from the Finite Element Method [35], or most other discretizations of Partial Differential Equations, have most of their entries equal to zero. During factorization of such matrices it pays off to take advantage of the sparsity pattern for a significant reduction in the number of floating point operations and execution time. The major issue of the sparse factorization is the aforementioned fill-in phenomenon. It turns out that the proper ordering of the matrix, represented by the matrices P and Q, may reduce the amount of fill-in. However, the search for the optimal ordering is an $\mathcal{NP}$-complete problem [39]. Therefore, many heuristics have been devised to

function xGETRF(matrix $(\mathbf{R}^{n\times n} \ni) A \equiv [a_{ij}];\ i,j = 1,\ldots,n$)
begin
    for $i = 2,\ldots,n$ do
    begin
$$a_{ij} := \frac{1}{a_{jj}}(a_{ij} - \sum_{k=1}^{j-1} a_{ik}a_{kj});\quad j = 1,\ldots,i-1$$
$$a_{ij} := (a_{ij} - \sum_{k=1}^{i-1} a_{ik}a_{kj});\quad j = i,\ldots,n$$
    end
end

Figure 1: Iterative LU factorization function of a dense matrix A. It is equivalent to LAPACK's `xGETRF()` function and is performed using Gaussian elimination (without a pivoting clause).

find an ordering which approximates the optimal one. These heuristics range from the divide and conquer approaches such as Nested Dissection [22, 29] to the greedy schemes such as Minimum Degree [1, 37]. For certain types of matrices, bandwidth and profile reducing orderings such as Reverse Cuthill-McKee [8, 23] and the Sloan ordering [34] may perform well.

Once the amount of fill-in is minimized through the appropriate ordering, it is still desirable to use the optimized BLAS to perform the floating point operations. This poses a problem since the sparse matrix coefficients are usually stored in a form that is not suitable for BLAS. There exist two major approaches that efficiently cope with this, namely the multifrontal [20] and supernodal [5] methods. The SuperLU package [28] is an example of a supernodal code, whereas UMFPACK [11, 12] is a multifrontal one.

Factorization algorithms for sparse matrices typically include the following phases, which sometimes are intertwined:

- matrix ordering to reduce fill-in,
- symbolic factorization,
- search for dense submatrices,
- numerical factorization.

The first phase is aimed at reducing the aforementioned amount of fill-in. Also, it may be used to improve the numerical stability of the factorization (it is then referred to as a static pivoting [18]). In our code, this phase serves both of these purposes, whereas in SuperLU and UMFPACK the pivoting is performed only during the factorization. The actual pivoting strategy being used in theses packages is called a threshold pivoting: the pivot is not necessarily the largest in absolute value in the current column (which is the case in the dense codes) but instead, it is just large enough to preserve numerical stability. This makes the pivoting much more efficient, especially with the complex data structures involved in sparse factorization.

The next phase finds the fill-in and allocates the required storage space. This process can be performed solely based on the matrix sparsity pattern information without considering matrix values. Substantial performance improvements are obtained in this phase if graph-theoretic concepts such as elimination trees and elimination dags [24] are efficiently utilized.

The last two phases are usually performed jointly. They aim at executing the required floating point operations at the highest rate possible. This may be achieved in a portable fashion through the use of BLAS. SuperLU uses supernodes, i.e. sets of columns of a similar sparsity structure, to call the Level 2 BLAS. Memory bandwidth is the limiting factor of the Level 2 BLAS, so, to reuse the data in cache and consequently improve the performance, the BLAS calls are reorganized yielding the so-called Level 2.5 BLAS technique [13, 28]. UMFPACK uses frontal matrices that are formed during the factorization process. They are stored as dense matrices and may be passed to the Level 3 BLAS.

4 Sparse Recursive Factorization Algorithm

The essential part of any sparse factorization code is the data structure used for storing matrix entries. The storage scheme for the sparse recursive code is illustrated in Fig. 5. It has the following characteristics:

- the data structure that describes the sparsity pattern is recursive,
- the storage scheme for numerical values has two levels:
  - the lower level, which consists of dense square submatrices (blocks) which enable direct use of the Level 3 BLAS, and
  - the upper level, which is a set of integer indices that describe the sparsity pattern of the blocks.

function xGETRF(matrix $A \in \mathbf{R}^{m \times n}$)
begin
    if ($A \in \mathbf{R}^{m \times 1}$) — *If the matrix has just one column.*
    begin
        $|a_{k1}| := \max_{1 \le i \le n} |a_{i1}|$ — *Find a suitable pivot.*
        $a_{11} :=: a_{k1}$ — *Exchange a_{11} with a_{k1}.*
        $A := \frac{1}{a_{11}} \cdot A$ — *Scale the matrix – equivalent to the Level 1 BLAS xSCAL().*
    end
    else begin

    $m_1 := \lfloor m/2 \rfloor$; $n_1 := \lfloor n/2 \rfloor$;
    $m_2 := m - \lfloor m/2 \rfloor$; $n_2 := n - \lfloor n/2 \rfloor$;

    Divide the matrix A into four submatrices.
    $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \in \mathbf{R}^{m \times n}$
    $A_{11} \in \mathbf{R}^{m_1 \times n_1}, A_{21} \in \mathbf{R}^{m_2 \times n_1}, A_{12} \in \mathbf{R}^{m_1 \times n_2}, A_{22} \in \mathbf{R}^{m_2 \times n_2}$

    xGETRF($\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}$); — *Recursive call.*

    xLASWP($\begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix}$); — *Apply pivoting from the recursive call.*

    Perform a lower triangular solve which is equivalent to the Level 3 BLAS xTRSM() function.
    Find X_{12} such that: $L_{11} \cdot X_{12} = A_{12}$
        where:
        $L_{11} \in \mathbf{R}^{m_1 \times n_1}$ is a lower triangular matrix of A_{11}
        with unitary diagonal
    $A_{12} := X_{12}$;

    Compute the Schur's complement which is equivalent to a matrix-matrix multiply performed by the Level 3 BLAS xGEMM() function.
    $A_{22} := A_{22} - A_{21} \cdot A_{12}$;

    xGETRF(A_{22}); — *Recursive call.*
    xLASWP($\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}$); — *Apply pivoting from the recursive call.*

    end
end.

Figure 2: Recursive LU factorization function of a dense matrix A equivalent to the LAPACK's xGETRF() function with partial pivoting code.

function xGETRF(matrix $A \in \mathbf{R}^{n \times n}$)
begin

if ($A \in \mathbf{R}^{1 \times 1}$) return; *Do nothing for matrices of order* 1.

$n_1 := \lfloor n/2 \rfloor$;
$n_2 := n - \lfloor n/2 \rfloor$;

Divide the matrix A into four submatrices.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \in \mathbf{R}^{n \times n}$$

$A_{11} \in \mathbf{R}^{n_1 \times n_1}, A_{21} \in \mathbf{R}^{n_2 \times n_1}, A_{12} \in \mathbf{R}^{n_1 \times n_2}, A_{22} \in \mathbf{R}^{n_2 \times n_2}$

xGETRF(A_{11}); *Recursive call.*

Perform an upper triangular solve which is equivalent to the Level 3 BLAS `xTRSM()` *function.*

Find X_{21} such that: $X_{21} \cdot U_{11} = A_{21}$
where:
$U_{11} \in \mathbf{R}^{n_1 \times n_1}$ is an upper triangular matrix of A_{11}
(including diagonal)

$A_{21} := X_{21}$;

Perform a lower triangular solve which is equivalent to the Level 3 BLAS `xTRSM()` *function.*

Find X_{12} such that: $L_{11} \cdot X_{12} = A_{12}$
where:
$L_{11} \in \mathbf{R}^{n_1 \times n_1}$ is a lower triangular matrix of A_{11}
with unitary diagonal

$A_{12} := X_{12}$;

Compute the Schur's complement which is equivalent to a matrix-matrix multiply performed by the Level 3 BLAS `xGEMM()` *function.*

$A_{22} := A_{22} - A_{21} \cdot A_{12}$;

xGETRF(A_{22}); *Recursive call.*

end.

Figure 3: Recursive LU factorization function used for sparse matrices (no pivoting is performed).

Column-major storage scheme:

1	8	15	22	29	36	43
2	9	16	23	30	37	44
3	10	17	24	31	38	45
4	11	18	25	32	39	46
5	12	19	26	33	40	47
6	13	20	27	34	41	48
7	14	21	28	35	42	49

Recursive storage scheme:

1	4	5	22	23	28	29
2	6	8	24	26	30	32
3	7	9	25	27	31	33
10	14	16	34	36	42	44
11	15	17	35	37	43	45
12	18	20	38	40	46	48
13	19	21	39	41	47	49

function `convert`(matrix $A \in \mathbf{R}^{n \times n}$)
begin
    if $(A \in \mathbf{R}^{1 \times 1})$
        Copy current element of A;
        Go to the next element of A;
    else
    begin
        $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix};$
        `convert`(A_{11});
        `convert`(A_{21});
        `convert`(A_{12});
        `convert`(A_{22});
    end
end.

Figure 4: Column-major storage scheme versus recursive storage (left) and function for converting a square matrix from the column-major to recursive storage (right.)

There are two important ramifications of this scheme. First, the number of integer indices that describe the sparsity pattern is decreased because each of these indices refers to a block of values rather than individual values. It allows for more compact data structures and during the factorization it translates into a shorter execution time because there is less sparsity pattern data to traverse and more floating operations are performed by efficient BLAS codes – as opposed to in code that relies on compiler optimization. Second, the blocking introduces additional nonzero entries that would not be present otherwise. These artificial nonzeros amount to an increase in storage requirements. Also, the execution time is longer because it is spent on floating point operations that are performed on the additional zero values. This leads to the conclusion that the sparse recursive storage scheme performs best when almost dense blocks exist in the L and U factors of the matrix. Such a structure may be achieved with the band-reducing orderings such as Reverse Cuthill-McKee or Sloan. These orderings tend to incur more fill-in than others such as Minimum Degree or Nested Dissection, but this effect can be expected to be alleviated by the aforementioned compactness of the data storage scheme and utilization of the Level 3 BLAS.

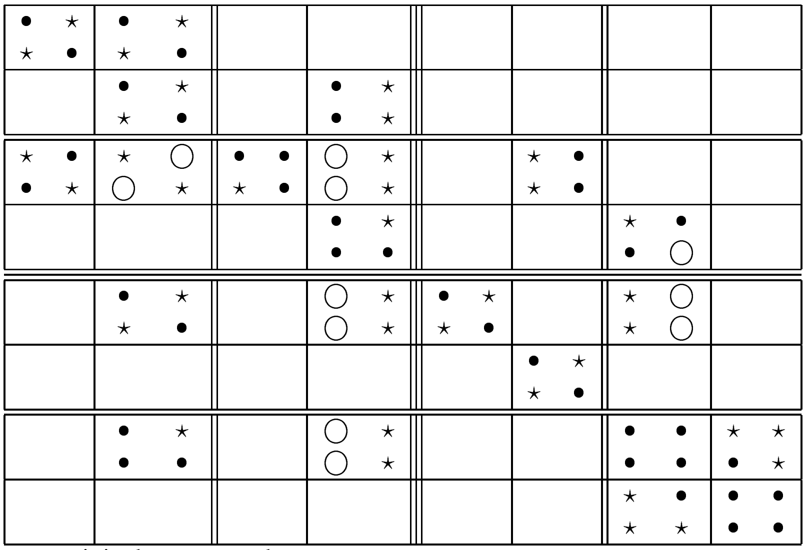
The algorithm from Fig. 3 remains almost unchanged in the sparse case – the differences being that calls to BLAS are replaced by the calls to their sparse recursive counterparts and that the data structure is no longer the same. Fig. 6 and Fig. 7 show the recursive BLAS routines used by the sparse recursive factorization algorithm. They traverse the sparsity pattern and upon reaching a single dense block level they call the dense BLAS which perform actual floating point operations.

5 Performance Results

To test the performance of the sparse recursive factorization code it was compared to SuperLU Version 2.0 (available at `http://www.nersc.gov/~xiaoye/SuperLU/`) and UMFPACK Version 3.0 (available at `http://www.cise.ufl.edu/research/sparse/umfpack/`). The tests were performed on a Pentium III Linux workstation whose characteristics are given in Table 1.

Each of the codes were used to factor selected matrices from the Harwell-Boeing collection [19], and Tim Davis' [10] matrix collection. These matrices were used to evaluate the performance of SuperLU. The matrices are unsymmetric so they cannot be used directly with the Conjugate Gradient method and there is no general method for finding the optimal iterative method other than trying each one in turn or running all of the methods in parallel [7]. Table 2 shows the total execution time of factorization (including symbolic and numerical phases) and forward error estimates.

The performance of a sparse factorization code can be tuned for a given computer architecture and a particular matrix. For SuperLU, the most influential parameter was the fill-in reducing ordering used prior to factorization. All of the available ordering schemes that come with SuperLU were used and Table 2 gives the best time that was obtained. UMFPACK supports only one kind of ordering (a column oriented version of the Approximate Minimum Degree algorithm [1]) so it was used with the default values of its tuning parameters and threshold pivoting disabled. For the recursive approach all of the matrices were ordered using the Reverse Cuthill-McKee ordering. However, the block size selected somewhat influences the execution time. Table 2 shows the best running time out of the block sizes ranging between 40 and 120. The block size depends mostly on the size of the Level 1 cache but also on the sparsity pattern of the matrix. Nevertheless, running times for the different block sizes are comparable. SuperLU and UMFPACK also have tunable parameters that functionally resemble the block size parameter but their importance is marginal as compared to that of the matrix ordering.



• – original nonzero value
0 – zero value introduced due to blocking
x – zero value introduced due to fill-in

Figure 5: Sparse recursive blocked storage scheme with the blocking factor equal 2.

$\boxed{\begin{array}{l} C := C - A \cdot B \\ A,B,C \text{ are arbitrary rectangular matrices} \end{array}}$

function `xGEMM('N','N',`$\alpha = -1, A, B,\ \beta = 1, C$`)`
begin

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}; C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix};$$

$\boxed{C_{11} := C_{11} - A_{11} \cdot B_{11}}$
`xGEMM('N','N',`$\alpha = -1, A_{11}, B_{11},\ \beta = 1, C_{11}$`);`
$\boxed{C_{21} := C_{21} - A_{21} \cdot B_{11}}$
`xGEMM('N','N',`$\alpha = -1, A_{21}, B_{11},\ \beta = 1, C_{21}$`);`
$\boxed{C_{11} := C_{11} - A_{12} \cdot B_{21}}$
`xGEMM('N','N',`$\alpha = -1, A_{12}, B_{21},\ \beta = 1, C_{11}$`);`
$\boxed{C_{21} := C_{21} - A_{22} \cdot B_{21}}$
`xGEMM('N','N',`$\alpha = -1, A_{22}, B_{21},\ \beta = 1, C_{21}$`);`
$\boxed{C_{12} := C_{12} - A_{11} \cdot B_{12}}$
`xGEMM('N','N',`$\alpha = -1, A_{11}, B_{12},\ \beta = 1, C_{12}$`);`
$\boxed{C_{12} := C_{12} - A_{12} \cdot B_{22}}$
`xGEMM('N','N',`$\alpha = -1, A_{12}, B_{22},\ \beta = 1, C_{12}$`);`
$\boxed{C_{22} := C_{22} - A_{21} \cdot B_{12}}$
`xGEMM('N','N',`$\alpha = -1, A_{21}, B_{12},\ \beta = 1, C_{22}$`);`
$\boxed{C_{22} := C_{22} - A_{22} \cdot B_{22}}$
`xGEMM('N','N',`$\alpha = -1, A_{22}, B_{22},\ \beta = 1, C_{22}$`);`

end.

Figure 6: Recursive formulation of the `xGEMM()` function which is used in the sparse recursive factorization.

$$\boxed{\begin{array}{ll} B := B \cdot U^{-1} & \\ U & \text{is an upper triangular matrix} \\ & \text{with non-unitary diagonal} \end{array}}$$

function `xTRSM('R','U','N','N',U,B)`
begin

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}; U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix};$$

$\boxed{B_{11} := B_{11} \cdot U_{11}^{-1}}$
`xTRSM('R','U','N','N',`U_{11}`,`B_{11}`);`
$\boxed{B_{21} := B_{21} \cdot U_{11}^{-1}}$
`xTRSM('R','U','N','N',`U_{11}`,`B_{21}`);`
$\boxed{B_{22} := B_{22} - B_{21} \cdot U_{12}}$
`xGEMM('N','N',`$\alpha = -1$`,`B_{21}`,`U_{12}`, `$\beta = 1$`,`B_{22}`);`
$\boxed{B_{22} := B_{22} \cdot U_{22}^{-1}}$
`xTRSM('R','U','N','N',`U_{22}`,`B_{22}`);`
$\boxed{B_{12} := B_{12} - B_{11} \cdot U_{12}}$
`xGEMM('N','N',`$\alpha = -1$`,`B_{11}`,`U_{12}`, `$\beta = 1$`,`B_{12}`);`
$\boxed{B_{12} := B_{12} \cdot U_{22}^{-1}}$
`xTRSM('R','U','N','N',`U_{22}`,`B_{12}`);`

end.

$$\boxed{\begin{array}{ll} B := L^{-1} \cdot B & \\ L & \text{is a lower triangular matrix} \\ & \text{with unitary diagonal} \end{array}}$$

function `xTRSM('L','L','N','U',L,B)`
begin

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}; L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix};$$

$\boxed{B_{11} := L_{11}^{-1} \cdot B_{11};}$
`xTRSM('L','L','N','U',`L_{11}`,`B_{11}`);`
$\boxed{B_{21} := B_{21} - L_{21} \cdot B_{11};}$
`xGEMM('N','N',`$\alpha = -1$`,`L_{21}`,`B_{11}`, `$\beta = 1$`,`B_{21}`);`
$\boxed{B_{21} := L_{22}^{-1} \cdot B_{21};}$
`xTRSM('L','L','N','U',`L_{22}`,`B_{21}`);`
$\boxed{B_{12} := L_{11}^{-1} \cdot B_{12};}$
`xTRSM('L','L','N','U',`L_{11}`,`B_{12}`);`
$\boxed{B_{22} := B_{22} - L_{12} \cdot B_{12};}$
`xGEMM('N','N',`$\alpha = -1$`,`L_{12}`,`B_{12}`, `$\beta = 1$`,`B_{22}`);`
$\boxed{B_{22} := L_{22}^{-1} \cdot B_{22};}$
`xTRSM('L','L','N','U',`L_{22}`,`B_{22}`);`

end.

Figure 7: Recursive formulation of the `xTRSM()` functions used in the sparse recursive factorization.

Hardware specifications	
CPU type	Pentium III
CPU clock rate	550 MHz
System bus clock rate	100 MHz
L1 data cache	16 Kbytes
L1 instruction cache	16 Kbytes
L2 unified cache	512 Kbytes
Main memory	512 MBytes
CPU performance	
Peak	550 MFLOP/s
Matrix-matrix multiply	390 MFLOP/s
Matrix-vector multiply	100 MFLOP/s

Table 1: Parameters of the machine used in the tests.

The total factorization time from Table 2 favors the recursive approach for some matrices, e.g., `ex11`, `psmigr_1` and `wang3`, and for others it strongly discourages its use (matrices `mcfe`, `memplus` and `raefsky4`). There are two major reasons for the poor performance of the recursive code on the second class. First, there is an average density factor which is the ratio of the true nonzero entries of the factored matrix to all the entries in the blocks. It indicates how many artificial nonzeros were introduced by the blocking technique. Whenever this factor drops below 70%, i.e. 30% of the factored matrix entries do not come from the L and U factors, the performance of the recursive code will most likely suffer. Even when the density factor is satisfactory, still, the amount of fill-in incurred by the Reverse Cuthill-McKee ordering may substantially exceed that of other orderings. In both cases, i.e. with a low value of the density factor or excessive fill-in, the recursive approach performs too many unnecessary floating point operations and even the high execution rates of the Level 3 BLAS are not able to offset it.

The computed forward error is similar for all of the codes despite the fact that two different approaches to pivoting were employed. Only SuperLU was doing threshold pivoting while the other two codes had the threshold pivoting either disabled (UMFPACK) or there was no code for any kind of pivoting (recursive code).

Table 3 shows the matrix parameters and storage requirements for the test matrices. It can be seen that SuperLU and UMFPACK use slightly less memory and consequently perform fewer floating point operations. This may be attributed to the Minimum Degree algorithm used as an ordering strategy by these codes which minimizes the fill-in and thus the space required to store the factored matrix.

6 Conclusions and Future Work

We have shown that the recursive approach to the sparse matrix factorization may lead to an efficient implementation. The execution time, storage requirements, and error estimates of the solution are comparable to that of supernodal and multifrontal codes. However, there are still matrices for which the recursive code does not perform well. These cases should be investigated further and possibly a metric devised that would allow selecting the best factorization method for a given matrix. This metric will probably include the aforementioned density factor. During a preprocessing phase, the density factor is computed and only if it exceeds a certain threshold the recursive code is used. An open question is which code to choose when the recursive one is not appropriate. A performance model is necessary that links together the features of the multifrontal and supernodal approaches with the characteristics of the matrix to be factored and machnie it is to be used on.

The problem with low values of the density factor may be regarded as a future research direction. The aim should be to make the recursive code more adaptive to the matrix sparsity pattern. It could allow the use of matrix orderings other than Reverse Cuthill-McKee because the high average density of the blocks will not be as crucial any more.

Another outstanding issue is the numerical stability of the factorization process. As it is now, it does not perform pivoting and still delivers acceptable accuracy. On matrices other than those tested, the method may still fail, and even iterative refinement may be unable to regain sufficient accuracy. Therefore, an extended version that performs at least some form of pivoting would likely be much more robust.

A parallel version of the recursive approach for sparse matrices is also under consideration. At this point, there are many issues to be resolved and the main direction is still not clear. Supernodal and multifrontal approaches use symbolical data structures from the sequential algorithm to assist the parallel implementation. In the recusive approach no such structures are used and consequently parallelism has to be exploited in some other way. On the other hand, dense codes [21, 30] use recursion only locally and resort to other techniques in order to expose parallelism inherent in the factorization process [32].

Matrix name	SuperLU T [s]	SuperLU FERR	UMFPACK T [s]	UMFPACK FERR	Recursion T [s]	Recursion FERR
af23560	44.2	$5 \cdot 10^{-14}$	29.3	$4 \cdot 10^{-04}$	31.3	$2 \cdot 10^{-14}$
ex11	109.7	$3 \cdot 10^{-05}$	66.2	$2 \cdot 10^{-03}$	55.3	$1 \cdot 10^{-06}$
goodwin	6.5	$1 \cdot 10^{-08}$	17.8	$2 \cdot 10^{-02}$	6.7	$5 \cdot 10^{-06}$
jpwh_991	0.2	$3 \cdot 10^{-15}$	0.1	$2 \cdot 10^{-12}$	0.3	$3 \cdot 10^{-15}$
mcfe	0.1	$1 \cdot 10^{-13}$	0.2	$2 \cdot 10^{-13}$	0.2	$9 \cdot 10^{-13}$
memplus	0.3	$2 \cdot 10^{-12}$	20.1	$4 \cdot 10^{-11}$	12.7	$7 \cdot 10^{-13}$
olafu	26.2	$1 \cdot 10^{-06}$	19.6	$2 \cdot 10^{-06}$	22.1	$4 \cdot 10^{-09}$
orsreg_1	0.5	$1 \cdot 10^{-13}$	0.3	$2 \cdot 10^{-12}$	0.5	$2 \cdot 10^{-13}$
psmigr_1	110.8	$8 \cdot 10^{-11}$	242.6	$2 \cdot 10^{-08}$	88.6	$1 \cdot 10^{-05}$
raefsky3	62.1	$1 \cdot 10^{-09}$	52.4	$5 \cdot 10^{-10}$	69.7	$4 \cdot 10^{-13}$
raefsky4	82.5	$2 \cdot 10^{-06}$	101.9	$5 \cdot 10^{+01}$*	104.3	$4 \cdot 10^{-06}$
saylr4	0.9	$3 \cdot 10^{-11}$	0.7	$2 \cdot 10^{-07}$	1.0	$1 \cdot 10^{-11}$
sherman3	0.6	$6 \cdot 10^{-13}$	0.5	$2 \cdot 10^{-11}$	0.7	$5 \cdot 10^{-13}$
sherman5	0.3	$1 \cdot 10^{-13}$	0.3	$4 \cdot 10^{-12}$	0.3	$6 \cdot 10^{-15}$
wang3	84.1	$2 \cdot 10^{-14}$	132.1	$5 \cdot 10^{-08}$	79.2	$2 \cdot 10^{-14}$

T - combined time for symbolic and numerical factorization
FERR $= \frac{\|\hat{\mathbf{x}}-\mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty}$ (forward error)
* the matrix raefsky4 requires the threshold pivoting in UMFPACK to be enabled in order to give a satisfactory forward error

Table 2: Factorization time and error estimates for the test matrices for three factorization codes.

Name	N	NZ·10^3	SuperLU $L+U$ [MB]	UMFPACK $L+U$ [MB]	Recursion $L+U$ [MB]	Recursion block size
af23560	23560	461	132.2	96.6	149.7	120
ex11	16614	1097	210.2	129.2	150.6	80
goodwin	7320	325	31.3	57.0	35.0	40
jpwh_991	991	6	1.4	1.4	2.3	40
mcfe	765	24	0.9	0.7	1.8	40
memplus	17758	126	5.9	112.5	195.7	60
olafu	16146	1015	83.9	63.3	96.1	80
orsreg_1	2205	14	3.6	2.8	3.9	40
psmigr_1	3140	543	64.6	76.2	78.4	100
raefsky3	21200	1489	147.2	150.1	193.9	120
raefsky4	19779	1317	156.2	171.5	234.4	80
saylr4	3564	22	6.0	4.6	7.2	40
sherman3	5005	20	5.0	3.5	7.3	60
sherman5	3312	21	3.0	1.9	3.1	40
wang3	26064	177	116.7	249.7	256.7	120

N - order of the matrix
NZ - number of nonzero entries in the matrix
$L+U$ - size of memory required to store the L and U factors

Table 3: Parameters of the test matrices and their storage requirements for three factorization codes.

7 Acknowledgments

This work was supported in part by the University of California Berkeley through subcontract number SA2283JB, as part of the prime contract ACI-9813362 from the National Science Foundation; and by the University of California Berkeley through subcontract number SA1248PG, as part of the prime contract DEFG03-94ER25219 from the Department of Energy.

References

[1] R. Amestoy, T. Davis and I. Duff, An approximate minimum degree algorithm, Technical Report TR/PA/95/09, CERFACS, Toulouse, France.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, Third edition, 1999.

[3] B. Andersen, F. Gustavson, and J. Wasniewski, A recursive formulation of the Cholesky factorization operating on a matrix in packed storage form, in Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, USA, March 24-27, 1999.

[4] B. Andersen, F. Gustavson, A. Karaivanov, J. Wasniewski, and P. Yalamov, LAWRA – linear algebra with recursive algorithms, in Proceedings of the Conference on Parallel Processing and Applied Mathematics, Kazimierz Dolny, Poland, September 14-17, 1999.

[5] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon, Progress in sparse matrix methods in large sparse linear systems on vector supercomputers, Intern. J. of Supercomputer Applications **1**, pp. 10-30, 1987.

[6] D. Bailey, K. Lee, and H. Simon, Using Strassen's algorithm to accelerate the solution of linear systems, The Journal of Supercomputing **4**, pp. 357-371, 1990.

[7] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout and C. Romine, Algorithmic bombardment for the iterative solution of linear systems: a poly-iterative approach, JCAM **74**, pp. 91-109, 1996.

[8] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, in Proceedings of ACM National Conference, Association of Computing Machinery, New York, 1969.

[9] M. Dayde and I. Duff, Level 3 BLAS in LU factorization on Cray-2, ETA-10P and IBM 3090-200/VF, The International Jorunal of Supercomputer Applications **3**, pp. 40-70, 1989.

[10] T. Davis, University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/~davis/sparse/`, `ftp://ftp.cise.ufl.edu/pub/faculty/davis/matrices`, NA Digest **94**(42), October 16, 1994, NA Digest **96**(28), July 23, 1996, and NA Digest **97**(23), June 7, 1997.

[11] T. Davis and I. Duff, An unsymmetric-pattern multifrontal method for sparse LU factorization, Technical Report RAL-93-036, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, 1994.

[12] T. Davis, User's guide for the unsymmetric-pattern multifrontal package (UMFPACK), Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, June, 1993.

[13] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu, A supernodal approach to sparse partial pivoting, Technical report UCB//CSD-95-883, Computer Science Division, U. C. Berkeley, Berkeley, California, 1995.

[14] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, A set of Level 3 FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software **16**, pp. 1-17, March 1990.

[15] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software **14**, pp. 1-17, March 1988.

[16] J. Dongarra, and R. Whaley, Automatically Tuned Linear Algebra Software (ATLAS), in Proceedings of SC'89, 1989.

[17] I. Duff, A. Erisman, and J. Reid, Direct methods for sparse matrices, Oxford University Press, 1989.

[18] I. Duff and J. Koster, The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, SIAM J. Matrix Anal. Appl. **20**, pp. 889-901, 1999.

[19] I. Duff, R. Grimes and J. Lewis, Sparse matrix test problems, ACM Transactions on Mathematical Software **15**, pp. 1-14, 1989.

[20] I. Duff and J. Reid, The multifrontal solution of indefinite sparse symmetric linear equations, ACM Transactions on Mathematical Software **9**(3), pp. 302-325, September, 1983.

[21] E. Elmroth and F. Gustavson, Applying recursion to serial and parallel QR factorization leads to better performance, IBM Journal of Research and Development **44**(4), pp. 605-624, 2000.

[22] A. George, Nested dissection of a regular finite element mesh, SIAM Journal of Numerical Analysis **10**, pp. 345-363, 1973.

[23] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, An algorithm for reducing the bandwidth and profile of a sparse matrix, SIAM Journal of Numerical Analysis **13**(2), April, 1976.

[24] John R. Gilbert, and Joseph W. H. Liu, Elimination structures for unsymmetric sparse LU factors, SIAM J. Matrix Anal. Appl. **14**(2), pp. 334-352, April, 1993.

[25] F. Gustavson, Recursion leads to automatic variable blocking for dense linear-algebra algorithms, IBM Journal of Research and Development **41**(6), pp. 737-755, November 1997.

[26] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling, Recursive blocked data formats and BLAS's for dense linear algebra algorithms, In Proceedings of Applied Parallel Computing, PARA'98, B. Kågström, J. Dongarra and E. Elmroth, and J. Waśniewski eds., Lecture Notes in Computer Science **1541**, Springer-Verlag, Berlin pp. 195–206, 1998.

[27] F. Gustavson and I. Jonsson, Minimal-storage high-performance Cholesky factorization via blocking and recursion, IBM Journal of Research and Development **44**(6), pp. 823-850, Novemeber 2000.

[28] X. Li, Sparse Gaussian elimination on high performance computers, Ph.D. thesis, University of California at Berkeley, Computer Science Department, 1996.

[29] R. J. Lipton, D. J. Rose, and R. E. Tarjan, Generalized Nested Dissection, SIAM Journal on Numerical Analysis **16**, pp. 346-358, 1979.

[30] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, HPL – A portable implementation of the high-performance Linpack benchmark for distributed-memory computers, `http://icl.cs.utk.edu/hpl/`, `http://www.netlib.org/benchmark/hpl/`.

[31] M. Paprzycki and C. Cyphers, Using Strassen's matrix multiplication in high performance solution of linear systems, Computers Math. Applic. **31**(4/5), pp. 55-61, 1996.

[32] Y. Saad, Communication complexity of the Gaussian elimination algorithm on multiprocessors, Linear Algebra and Its Applications **77**, pp. 315-340, 1986.

[33] Y. Saad, Iterative methods for sparse linear systems, PWS Publishing Company, New York, 1996.

[34] S. W. Sloan, An algorithm for profile and wavefront reduction of sparse matrices, International Journal for Numerical Methods in Engineering **23**, pp. 239-251, 1986.

[35] G. Strang and G. Fix, An analysis of the Finite Element Method, Prentice-Hall, Inc., 1973.

[36] V. Strassen, Gaussian elimination is not optimal, Numerical Mathematics **13**, pp. 354-356, 1969.

[37] W. Tinney and J. Walker, Direct solutions of sparse network equations by optimally ordered triangular factorization, in Proceedings of the IEEE **55**, pp. 1801-1809, 1967.

[38] S. Toledo, Locality of Reference in LU Decomposition with partial pivoting, SIAM J. Matrix Anal. Appl., **18**(4), pp. 1065-1081, October 1997.

[39] M. Yannakakis, Computing the minimum fill-in is NP-complete, SIAM Journal on Algebraic and Discrete Methods **2**(1), pp. 77-79, March 1981.