Recursive approach in sparse matrix LU factorization

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

Abstract

This paper describes the recursive LU factorization of sparse matrices. The recursive formulation of common linear algebra codes has been proven very successful in dense matrix computations. We present an extension of this idea to sparse matrices. Our experiments show that the recursive approach can be very competitive with leading tools for sparse matrix factorization in terms of time, required storage space, and error estimation.

1 Introduction

One of the most commonly used linear algebra computational kernels is the LU factorization of a matrix. Given square real matrix A ($A \in \mathbf{R}^{n\times n}$) the goal of the factorization is to find matrices L, U, P and Q such that:

$$LU = PAQ, \tag{1}$$

where:

- L is lower triangular matrix with unitary diagonal,
- U is upper triangular matrix,
- P, Q are row and column permutation matrices, respectively, for which the following holds: $PP^T = QQ^T = I$ with I being identity matrix.

After the factorization, the following linear system of equations:

$$Ax = b, \tag{2}$$

can be solved with:

$$x = QU^{-1}L^{-1}Pb, \tag{3}$$

where $x, b \in \mathbf{R^n}$ are vectors.
However, matrix inverse operations in (3) do not have to be performed explicitly since L and U are triangular matrices. Thus, the solve operation becomes very fast which in turn renders the iterative improvement method feasible.

When both of the matrices P and Q of (1) are non-trivial then 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 matrix A of (2) is too close to being singular.

When matrix A is sparse, i.e. most of its entries are zero, it is of great importance to confine the factorization process to nonzero entries of the matrix. This creates additional problem of fill-in and to deal with this, usually both of the matrices: P and Q of (1) are non-trivial.

Typical linear algebra package such as LAPACK [2] which includes factorization performs it iteratively with blocking technique applied to enhance the performance of the computer implementation. Surprisingly, recursive approach was proven to outperform its loop-oriented counterpart [13]. For sparse matrices this approach cannot be applied directly because the sparsity pattern of a matrix has to be taken into account in order to reduce both storage requirements and floating point operation count which are the determining factors of the performance of sparse code.

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

Figure 1: The recursive LU factorization function of a dense matrix A equivalent to LAPACK's `xGETRF()` function performed using Gaussian elimination (without pivoting code).

2 Dense Recursive LU factorization

Figure 1 shows classical LU factorization code which uses Gaussian elimination. Rearrangement of the loops and introduction of blocking techniques can significantly increase performance of this code [2]. However, the recursive formulation of this algorithm shown in Figure 2 exhibits superior performance [13].

Figure 2 shows recursive algorithm of the LU factorization with pivoting code. Most notably, it does not contain any loop constructs. Also, most of the floating point operations are performed by calls to Level 3 BLAS [6] routines: `xTRSM()` and `xGEMM()`. They achieve the highest MFLOP/s rates on modern computer architectures with deep memory hierarchy as they come in the vendor-optimized libraries or are tuned for a given machine in some other way [8].

Yet another variation of recursive algorithm is shown in Figure 3 with pivoting omitted. Experiments show that this code performs equally well as the code from Figure 2. It is still possible to improve performance even further when the matrix is stored recursively. Such a storage scheme is illustrated in Figure 4. The scheme recursively aligns dense submatrices in memory. Recursive algorithm from Figure 3 recurses down to the level of a single submatrix and calls appropriate computational routine (either BLAS or `xGETRF()`) which works within boundaries of a single submatrix. Depending on the size of submatrices, it is possible to achieve execution rates higher than when matrix is stored by columns. Influenced by this result, we use code from Figure 3 as a base for our sparse algorithm.

3 Sparse Matrix Factorization

Matrices originating from the Finite Element Method [17], 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 in order to significantly reduce the number of floating point operations and running time. There is a major problem connected with it which is fill-in, i.e. new nonzero entries that are introduced in L, U factors which are not present in original matrix A of (1). It turns out that proper ordering of a matrix (matrices P and Q) can reduce amount of fill-in, however, search for the optimal ordering is $\mathcal{NP}$-complete problem. Thus, many heuristics has been devised, ranging from divide and conquer approaches such as Nested Dissection [10, 15] to greedy schemes such as Minimum Degree [1, 18]. For certain types of matrices bandwidth and profile reducing orderings such as Reverse Cuthill-McKee [4, 11] and Sloan [16] may perform well.

Once the amount of fill-in is minimized through the appropriate ordering, it is still desirable to use optimized BLAS. This poses additional problems since sparse matrix data is usually stored in a form which is not suitable for the BLAS. The two major approaches that cope with this are multifrontal and supernodal. We compare our recursive code only with SuperLU [14] which is a supernodal code, because until recently, multifrontal packages were suited only for certain class of matrices.

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

        $\boxed{\begin{array}{l} \textit{Divide matrix A into four submatrices.} \\ A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \in \mathbf{R}^{m\times n} \\ A_{11} \in \mathbf{R}^{m_1\times n_1},\ A_{21} \in \mathbf{R}^{m_2\times n_1},\ A_{12} \in \mathbf{R}^{m_1\times n_2},\ A_{22} \in \mathbf{R}^{m_2\times n_2} \end{array}}$
          $m_1 := \lfloor m/2 \rfloor;$          $n_1 := \lfloor n/2 \rfloor;$
          $m_2 := m - \lfloor m/2 \rfloor;$  $n_2 := n - \lfloor n/2 \rfloor;$

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

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

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

        $\boxed{\begin{array}{l} \textit{Compute the Shur's complement which is} \\ \textit{equivalent to a matrix-matrix multiply performed} \\ \textit{by Level 3 BLAS } \texttt{xGEMM()} \textit{ function.} \end{array}}$
        $A_{22} := A_{22} - A_{21} \cdot A_{12};$

        xGETRF($A_{22}$);                          $\boxed{\textit{Recursive call.}}$
end.
```

Figure 2: The recursive LU factorization function of a dense matrix A equivalent to LAPACK's `xGETRF()` function with pivoting code.

```
function xGETRF(matrix $A$)
begin
        if ($A \in \mathbf{R}^{1\times 1}$) return;        $\boxed{\textit{Do nothing for 1x1 matrices.}}$

        $\boxed{\begin{array}{l}\textit{Divide matrix A into four submatrices.}\\ A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \in \mathbf{R}^{n\times n} \\ A_{11} \in \mathbf{R}^{n_1\times n_1},\ A_{21} \in \mathbf{R}^{n_2\times n_1},\ A_{12} \in \mathbf{R}^{n_1\times n_2},\ A_{22} \in \mathbf{R}^{n_2\times n_2}\end{array}}$
        $n_1 := \lfloor n/2 \rfloor;$
        $n_2 := n - \lfloor n/2 \rfloor;$

        xGETRF($A_{11}$);        $\boxed{\textit{Recursive call.}}$

        $\boxed{\begin{array}{l}\textit{Upper triangular solve which is equivalent to}\\ \textit{Level 3 BLAS xTRSM() function.}\end{array}}$
        Find $X_{21}$ such that: $X_{21} \cdot U_{11} = A_{21}$
                where:
                $U_{11} \in \mathbf{R}^{n_1\times n_1}$ is upper triangular matrix of $A_{11}$
                (including diagonal)
        $A_{21} := X_{21};$

        $\boxed{\begin{array}{l}\textit{Lower triangular solve which is equivalent to}\\ \textit{Level 3 BLAS xTRSM() function.}\end{array}}$
        Find $X_{12}$ such that: $L_{11} \cdot X_{12} = A_{12}$
                where:
                $L_{11} \in \mathbf{R}^{n_1\times n_1}$ is lower triangular matrix of $A_{11}$
                with unitary diagonal
        $A_{12} := X_{12};$

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

        xGETRF($A_{22}$);        $\boxed{\textit{Recursive call.}}$
end.
```

Figure 3: The recursive LU factorization function used for sparse matrices equivalent to LAPACK's `xGETRF()` function with pivoting code omitted.

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 $A$)
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 square matrix from column-major to recursive storage (right.)

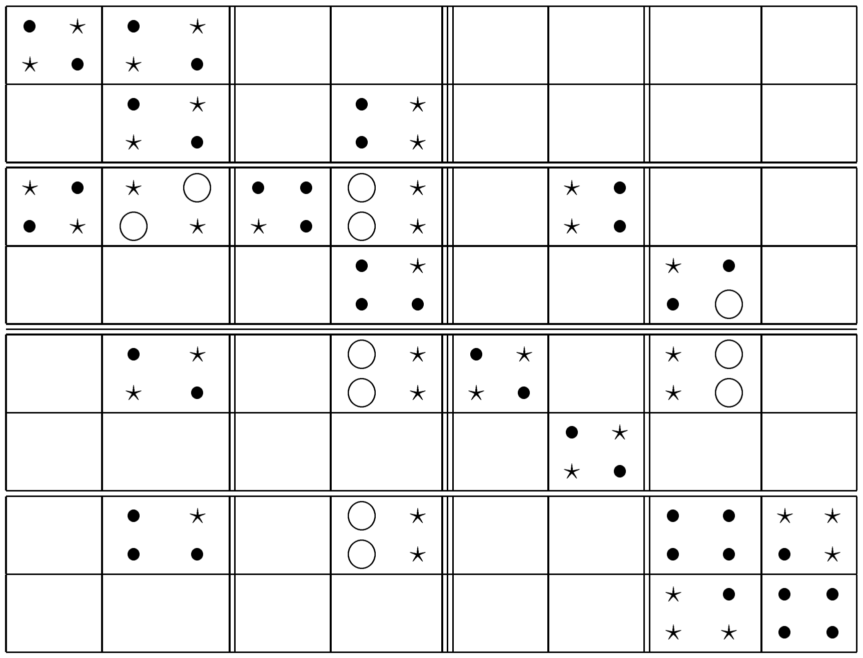
Factorization algorithms for sparse matrices typically include the following phases:

- matrix ordering that reduces fill-in,
- symbolic factorization,
- search for so called supernodes,
- numerical factorization.

The first phase is aimed at reducing the aforementioned amount of fill-in and sometimes at improving numerical stability of the factorization (it is referred to as static pivoting). In our code, this phase serves both of these purposes, whereas in SuperLU pivoting is performed during the factorization. The actual pivoting strategy is called threshold pivoting, i.e. the pivot that is being chosen is not necessarily the largest in absolute value in a current column (which is the case in a dense codes) but instead it has to satisfy less rigid requirements. This makes the pivoting much more efficient considering complexity of data structures involved in a sparse factorization.

The next stage finds the fill-in and allocates storage space for them. This process can be performed solely based on the matrix structure information without taking matrix values into consideration. Substantial performance improvements are obtained in this phase if graph-theoretic concepts such as elimination trees and elimination dags [12] are efficiently utilized.

Then, supernodes are found which are sets of columns that have a similar sparsity structure. They are used in the next stage: the numerical factorization. Supernodes enable the use of BLAS routines to improve the performance of the last phase. In case of SuperLU, the structure of supernodes allows only calls to Level 2 BLAS routines which have a performance limited by the CPU-memory bandwidth. To alleviate this problem, SuperLU reorganizes calls to BLAS routines to gain extra reuse of data already present in cache (this technique is referred to as the use of Level 2.5 BLAS [14]).



• – 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 blocking factor equal two.

4 Sparse Recursive Factorization Algorithm

In order to implement efficient factorization code we took into account all the issues mentioned above and developed a storage scheme with the following characteristics:

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

Figure 5 illustrates the sparse blocked recursive storage scheme that we use. 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. Second, blocking introduces additional nonzero elements which would not be present otherwise. The former implies more compact data structures, and during factorization translates into shorter execution time because there is less sparsity pattern data to traverse and more floating operations are performed within efficient BLAS codes.

The additional zero values that arise from blocking lead to an increase in storage requirements, which not only increases memory demands, but also execution time since floating operations are performed on zero values. This leads to the conclusion that the sparse recursive storage scheme will perform best when dense blocks exist in the L and U factors of a matrix. Such a structure may be achieved using band reducing orderings such as Reverse Cuthill-McKee [4] or Sloan [16]. These orderings incur more fill-in than others such as Minimum Degree [1, 18] or Nested Dissection [10, 15] but this effect is alleviated by aforementioned compactness of the data storage and utilization of Level 3 BLAS.

The algorithm from Figure 3 remains almost unchanged in the sparse case – the only difference are the calls to BLAS which are replaced by the calls to their sparse recursive counterparts and the data structure is

```
$\boxed{C := C - A \cdot B \quad A,B,C \text{ are arbitrary rectangular matrices}}$
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 `xGEMM()` function used in sparse recursive factorization.

no longer the same. Figures 6 and 7 show the recursive BLAS routines used by the factorization algorithm. They traverse the sparsity pattern and upon reaching single dense block level they call dense BLAS which perform actual floating point operations.

5 Performance Results

We have performed our experiments on an Pentium III workstation running Linux operating system. Table 1 summarizes characteristics of a machine used in tests. Table 2 shows timing results and error estimates for SuperLU Version 2.0 (available at `http://www.nersc.gov/~xiaoye/SuperLU/`) and for the recursive approach, operating on selected matrices from the Harwell-Boeing collection [9], and Tim Davis' [5] matrix collection, which were used in [14] to evaluate the performance of SuperLU. Performance of the sparse factorization code heavily depends on the initial ordering of the matrix. Thus, we have selected the best time we could obtain using all the available ordering schemes that come with SuperLU. For the recursive approach all of the matrices were ordered using Reverse Cuthill-McKee ordering [4, 11]. For the recursive approach it is possible to select different block sizes, which yields slightly different times. We then show the best running time out of block sizes ranging between 40 and 120. The computed forward and backward errors are similar for both codes despite the fact that two different approaches to pivoting are used. SuperLU uses threshold pivoting while in the recursive code there is no pivoting but instead the iterative improvement method is used.

Table 3 shows matrix parameters and storage requirements. It can be seen that SuperLU uses slightly less memory and consequently performs much fewer floating point operations. This may be attributed to the minimum degree ordering algorithm used by SuperLU which minimizes the fill-in and thus the space required to store factored matrix. The large difference between operation counts is not evident from the memory requirements because the recursive storage scheme is very compact. It comes from the fact that the recursive

```
$\boxed{B := B \cdot U^{-1}}$
$U$  is upper triangular matrix
     with non-unitary diagonal
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.
```

```
$\boxed{B := L^{-1} \cdot B}$
$L$  is lower triangular matrix
     with unitary diagonal
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 xTRSM() functions used in sparse recursive factorization.

Hardware specifications	
Machine type	Dual Pentium III
Clock rate	550 MHz
Bus clock rate	100 MHz
CPU(s)	2 x Pentium III
L1 data cache	16 Kbytes
L1 instruction cache	16 Kbytes
L2 unified cache	512 Kbytes
Main memory	512 MBytes
Performance of single CPU	
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.

Matrix	SuperLU			Recursive approach		
name	time [s]	FERR	BERR	time [s]	FERR	BERR
`af23560`	44.19	5.8e-14	1.3e-03	31.34	1.8e-14	1.1e-04
`ex11`	109.67	2.5e-05	8.1e-04	55.3	1.3e-06	1.3e-04
`goodwin`	6.49	1.2e-08	2.7e-04	6.74	4.6e-06	1.2e+01
`jpwh_991`	0.19	2.9e-15	2.7e-03	0.25	2.6e-15	1.3e-03
`mcfe`	0.07	1.2e-13	6.2e-04	0.22	9.1e-13	4.8e-04
`memplus`	0.29	2.1e-12	2.4e-04	12.67	6.6e-13	4.0e-05
`olafu`	26.16	1.1e-06	2.4e-05	22.1	3.7e-09	1.2e-05
`orsreg_1`	0.46	1.3e-13	2.5e-03	0.45	2.1e-13	8.7e-04
`psmigr_1`	110.79	7.9e-11	4.3e-03	88.61	1.2e-05	5.6e+01
`raefsky3`	62.07	1.4e-09	9.8e-04	69.67	4.4e-13	2.1e-04
`raefsky4`	82.45	2.3e-06	3.0e-04	104.29	3.5e-06	4.5e-05
`saylr4`	0.85	3.2e-11	8.3e-04	0.95	1.2e-11	5.6e-04
`sherman3`	0.61	6.0e-13	1.3e-04	0.67	4.8e-13	6.1e-05
`sherman5`	0.28	1.4e-13	1.2e-04	0.32	6.2e-15	1.7e-04
`wang3`	84.14	2.4e-14	3.9e-04	79.18	1.6e-14	1.8e-04

forward error: $\text{FERR} = \frac{||\hat{\mathbf{x}}-\mathbf{x}||_\infty}{||\mathbf{x}||_\infty}$

backward error: $\text{BERR} = \frac{||\mathbf{A}\hat{\mathbf{x}}-\mathbf{b}||_\infty}{(||\mathbf{A}||_\infty ||\hat{\mathbf{x}}||_\infty + ||\mathbf{b}||_\infty)\cdot\varepsilon\cdot\mathbf{n}}$

Table 2: Factorization time and error estimation for the test matrices.

Matrix parameters			SuperLU	Recursive approach	
Name	N	nonzeros	L+U [MBytes]	L+U [MBytes]	block size
af23560	23560	460598	132.2	149.7	120
ex11	16614	1096948	210.2	150.6	80
goodwin	7320	324772	31.3	35.0	40
jpwh_991	991	6027	1.4	2.3	40
mcfe	765	24382	0.9	1.8	40
memplus	17758	126150	5.9	195.7	60
olafu	16146	1015156	83.9	96.1	80
orsreg_1	2205	14133	3.6	3.9	40
psmigr_1	3140	543162	64.6	78.4	100
raefsky3	21200	1488768	147.2	193.9	120
raefsky4	19779	1316789	156.2	234.4	80
saylr4	3564	22316	6.0	7.2	40
sherman3	5005	20033	5.0	7.3	60
sherman5	3312	20793	3.0	3.1	40
wang3	26064	177168	116.7	256.7	120

Table 3: Parameters of the test matrices and their storage requirements for SuperLU and recursive approach.

approach stores many more floating point values (most of which are zeros). However, it becomes noticeable in the floating point operation count which is proportional to the third power of the number of floating point values. Nevertheless, the performance in terms of time to solution of the recursive code is still very competitive with SuperLU due to the use of Level 3 BLAS.

6 Conclusions and Future Work

We have shown that a recursive approach to sparse matrix factorization may lead to a very efficient implementation which is competitive with the best supernodal codes in terms of execution time, storage requirements, and error estimation of the solution. However, there are still matrices for which our code does not perform well. We plan to investigate further these cases and devise matrix metrics which would allow us to select the best factorization method for a given matrix [3].

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] 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.

[4] 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.

[5] 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, **92**(42), October 16, 1994, NA Digest, **96**(28), July 23, 1996, and NA Digest, textbf97(23), June 7, 1997.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] F. Gustavson, Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms, IBM Journal of Research and Development, Volume 41, Number 6, November 1997.

[14] X. Li, Sparse Gaussian Elimination on High Performance Computers, PhD thesis, University of California at Berkeley, Computer Science Department, 1996.

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

[16] 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.

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

[18] 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.