To the Graduate Council:

I am submitting herewith a dissertation written by Piotr Rafal Luszczek entitled "Performance Improvements of Common Sparse Numerical Linear Algebra Computations." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

DR. JACK J. DONGARRA
Major Professor

We have read this dissertation
and recommend its acceptance:

DR. ALLEN BAKER

DR. MICHAEL BERRY

DR. VICTOR EIJKHOUT

Accepted for the Council:

DR. ANNE MAYHEW
Vice Provost and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

Performance Improvements of Common Sparse Numerical Linear Algebra Computations

A Thesis
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

Piotr Rafal Luszczek
May 2003

Acknowledgments

I would like to thank my advisor, Prof. Dongarra, for letting me first become his student and then for his guidance, mentorship and continuing support that made this research possible.

I am thankful to Profs. Baker, Berry, and Eijkhout for agreeing to serve on my graduate committee, providing me with helpful suggestions, and contributing to seamless cooperation.

In addition, I would like to express my appreciation to Profs. Langston and Raghavan who offered invaluable advices on technical issues that I was facing while working on this dissertation.

Also, the sponsorship of the National Science Foundation and the Department of Energy is acknowledged as this work was supported in part by the University of California Berkeley through subcontract number SA2283JB, as a part of the prime contract ACI-9813362 from the National Science Foundation; and by the University of California Berkeley through subcontract number SA1248PG, as a part of the prime contract DEFG03-94ER25219 from the U.S. Department of Energy.

Abstract

Manufacturers of computer hardware are able to continuously sustain an unprecedented pace of progress in computing speed of their products, partially due to increased clock rates but also because of ever more complicated chip designs. With new processor families appearing every few years, it is increasingly harder to achieve high performance rates in sparse matrix computations. This research proposes new methods for sparse matrix factorizations and applies in an iterative code generalizations of known concepts from related disciplines.

The proposed solutions and extensions are implemented in ways that tend to deliver efficiency while retaining ease of use of existing solutions. The implementations are thoroughly timed and analyzed using a commonly accepted set of test matrices. The tests were conducted on modern processors that seem to have gained an appreciable level of popularity and are fairly representative for a wider range of processor types that are available on the market now or in the near future.

The new factorization technique formally introduced in the early chapters is later on proven to be quite competitive with state of the art software currently available. Although not totally superior in all cases (as probably no single approach could possibly be), the new factorization algorithm exhibits a few promising features.

In addition, an all-embracing optimization effort is applied to an iterative algorithm that stands out for its robustness. This also gives satisfactory results on the tested computing platforms in terms of performance improvement. The same set of test matrices is used to enable an easy comparison between both investigated techniques, even though they are customarily treated separately in the literature.

Possible extensions of the presented work are discussed. They range from easily conceivable merging with existing solutions to rather more evolved schemes dependent on hard to predict progress in theoretical and algorithmic research.

Contents

List of Tables

List of Figures

Chapter 1

Problem Description

1.1 Introduction

The computer hardware industry (and its high performance branch in particular) continues to follow Moore's law [122, 124] even though there exist skeptical opinions [158]. This development trend on the one hand makes the integrated circuits faster, but on the other hand, more complex and harder to use. At the same time, there exists an ever increasing demand from science and industry for yet higher computation rates – Grand Challenge problems [109, 121, 126, 127] and the interest in the Grid [72] exemplify the need. In a nutshell, this work shows a way of applying recent algorithmic advances in dense numerical linear algebra and matrix ordering optimizations in sparse computational kernels, namely direct and iterative codes.

1.2 Motivation

The prevalent processor architecture for high performance computers has been Reduced Instruction Set Computer (RISC) and recently (for the second time in history) also Complex Instruction Set Computer (CISC) [44, 122] with introduction of increasingly powerful commodity desktop computers. Despite the differences between the two, when solving applied numerical linear algebra problems they benefit from similar optimization techniques [51, 88] – those that are used in tuned versions of Basic Linear Algebra Subroutines (BLAS) [56, 163] and deliver a rather high percentage of theoretical peak performance [55]. Unfortunately, many interesting scientific and commercial problems cannot be formulated in terms of dense linear algebra kernels since that would require prohibitive amounts of storage and computational capacity. In effect, possible formulations are the main source of sparse matrices that have had customarily large dimensions because they commonly originate in discretizations of rather large multi-dimensional problems. A positive feature of sparse matrices is that relatively few entries that need to be considered during computations and while storing them. This has been taken advantage of in many ways, most commonly to reduce storage requirements. The computational efficiency suffers because intricate data structures do not match the execution patterns favored by modern Central Processing Units (CPU). Clearly, BLAS is for now the only way to achieve the levels of performance that are commonly advertised by hardware vendors. To address this problem, it is necessary to consider how a contemporary CPU handles floating point (FP) data, how BLAS use this information, and why sparse computations fail in this respect.

1.2.1 Application of Recursive Methods

On modern CPUs, the two major issues that hinder efficient calculations of sparse codes, as opposed to dense codes, are an extra demand for memory bandwidth and additional fixed point calculations. Both of these are directly related to sparse data storage schemes which require transfer and processing of integer data [156]. The former may be addressed in a novel way – by use of recursion [53, 54]. It was possible because of recent introduction of recursion-based formulation of matrix decomposition algorithms [67, 95, 157] that reduce the memory bus traffic. In addition, if the recursive LU algorithm is combined with a recursive layout [93] then the resulting code is superior to any known software techniques as shown later on. This is chosen as the premise for possible application in a direct sparse solver. The latter issue of integer arithmetic may be addressed with direct use of BLAS. This, however, will require reformulating FP computations, but there exist ways to do so and they will be elaborated on later. The point needs to be stressed here that the transfer of data that describes the sparsity structure (not present in dense computations) is more important than the extra calculations on such data because of the widening gap between the processor and system bus speeds – a fact illustrated quite well by the ratio of performance of BLAS `xGEMV` and `xGEMM` routines as presented in section 4.2.

A quick argument in favor of recursive formulation of a computational task may easily be made with a simple example of the Fibonacci series. The traditional way of defining the Fibonacci series is this:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 2 \\ F(n) &= F(n-1)+F(n-2) \end{aligned}$$

This is a, so called, *tail* recursion. In order to calculate a single element of the series all preceeding elements have to be calculated first. It may be (rather easily) proven that the following definition is an equivalent one:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 2 \\ F(2) &= 3 \\ F(n) &= \begin{cases} F(\frac{n-1}{2})^2+F(\frac{n-1}{2}-1)^2 & \text{if } n \text{ is odd} \\ F(n/2)^2-F(n/2-2)^2 & \text{if } n \text{ is even} \end{cases} \end{aligned}$$

The latter formulation uses a *divide and conquer* recursion and even intuitively should be faster. The intuition proves correct in practice because the latter formulation is typically orders of magnitude faster. The former formulation is in a sense analogous to the LU decomposition method used in LINPACK [46] – a single column of the matrix is factored at a time. The latter formulation of the Fibonacci series corresponds the aforementioned recursive LU factorization – at any given point of the algorithm both halves of the matrix are processed recursively.

The are more examples of rather elegant use of recursion to formulate a superior algorithm. To name the some of them, the Fast Fourier Transform (FFT) [123] and Nested Disection [76] algorithms should be mentioned. In these cases, the divide and conquer recursion was successfully used to reduce the computational cost of a computational task.

1.2.2 Iterative Methods

In practice, direct methods could fail for two main reasons: prohibitive amount of storage requirements due to fill-in and/or extremely long factorization time (again due to fill-in). In such cases, an iterative method is a viable solution. On one hand, it does not incur fill-in and on the other (if matrix properties permit) executes far less FP operations in comparison with a direct method. However, iterative methods are known for lack of robustness, i.e. might not converge for a well-conditioned matrix. It is less of a problem with emergence of techniques such as the Bi-CGSTAB [161] method designed in a sense for unsymmetric matrices. Once the choice of the method has been made, performance optimization options need to be considered. As it was the case in direct methods, integer indices are the main performance concern. A different optimization goal is striven here as the dominant operation is matrix-vector multiplication (contrasting with matrix-matrix multiplication for direct codes). Consequently, a different approach must be taken – the one that optimizes system bus utilization and takes advantage of RISC architectures for relatively small dense structures present in sparse matrices – be it short vectors or elongated rectangular matrices. In particular, in such a setting BLAS cannot be considered to leverage hardware's high computational rate as the function call overhead cannot be offset by possible performance gains.

1.3 Problem Statement

The main objective of this work is to improve solution time of the following system of linear equations:

$$Ax = b, \tag{1.1}$$

where A is a n by n real matrix ($A \in \mathbb{R}^{n \times n}$), and x and b are n-dimensional real vectors ($b, x \in \mathbb{R}^n$). The values of A and b are known and the task is to find x satisfying (1.1). It is assumed that the matrix A is large (of order commonly exceeding ten thousand) and sparse, in other words, 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 [146] and direct methods [58].

Iterative methods, in particular techniques based on Krylov subspace such as the Conjugate Gradient [97] 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 [41], one has to revert to direct methods [58]. For such methods, this study analyzes how the recursive formulation of the LU factorization algorithm can be efficiently applied to sparse matrices. It involves consideration of a matrix ordering scheme applied prior to factorization. Various schemes for sparse matrix storage are considered to exploit most commonly used computer architectures and existing software for matrix computations.

An attractive meeting point between direct and iterative methods is preconditioning [19]. In a sense, it addresses shortcomings of both approaches. Common causes of failure for direct methods are excessive storage requirements to accommodate fill-in or prohibitive amount of FP operations. On the other hand, iterative methods (especially non-stationary ones) are quite capricious when it comes to matrix spectrum considerations. Preconditioning fits perfectly in such a situation. It allocates space for a closely controlled amount of fill-in and consequently has predictable computational complexity. L and U factors resulting from incomplete LU preconditioning are often less numerically accurate than those from the proper Gaussian elimination process, but are good enough to bring the iteration matrix close to the identity matrix. As the name – incomplete LU – indicates, there is a notion of incompleteness involved in the methods that are used in preconditioning. This favorably coincides with the fact that the recursive LU factorization for sparse matrices does not perform pivoting – a possibly incomplete solution for some matrices but quite likely sufficient for the purposes of applications involving iterative methods.

1.4 Limitations

This study is concerned with application of recursion to direct methods, which have a drawback of incurring fill-in in the factored matrix which, in turn, may prohibit factorization of large matrices due to excessive memory requirements. To increase the performance, the Level 3 BLAS have to be used. This places extra burden on the matrix ordering which has to fulfill one more requirement, i.e., it has to introduce dense sub-matrices in the original matrix structure. This, again, may not be possible for certain types of matrices. Thus, the study aims at optimizing performance for as many types of matrices as possible with the provision of less satisfactory results for remaining matrices.

Another decision made early in the development process was to not include pivoting in the recursive sparse LU code. Obviously it simplifies the algorithm but simplification was not the only reason. There exist strong evidence [114, 115] that other techniques (mainly a form of equilibration [60]) provide enough countermeasures against excessive pivot growth during the factorization for most cases. However, if during the factorization a diagonal value occurs that is dangerously close to 0, it is dealt with techniques described later on (see section 2.5).

The iterative method research is limited to the unpreconditioned Bi-CGSTAB method. Other researchers have considered the use of multiple iterative methods [20]. In addition, even though preconditioning is an attractive option to increase robustness [19] and serves as an interesting conceptual bridge between a true direct code and a purely iterative method but consequently it poses new challenges which were decided to be left out of this effort. However, this research provides concepts, tools, and techniques to create a preconditioned iterative method.

Yet another constraint imposed on this research is the rather early maturity state of parallel implementations of the recursive LU code for dense matrices [106, 107]. Naturally, a high quality parallel dense recursive solver would be a perfect starting point for a parallel sparse recursive solver as it was the case in the sequential case. More thorough scalability considerations need to be provided prior to competing with existing sparse solvers for parallel machines. A contrasting argument is raised against including parallel iterative method techniques here – namely a rather interesting concepts were proposed by others [20, 63] which do not fit directly into the problem addressed by this writing.

Lastly, techniques specific to Vector Processing Units (VPU) are not included here as the processors traditionally have been addressing the problem of sparse computations with `gather` and `scatter` instructions and the optimizations considered here result in rather short (in VPU vernacular) vector lengths which quite certainly would not yield substantial benefits.

Chapter 2

Theoretical Framework

2.1 Decompositional Techniques

Given a system of linear equation denoted with

$$Ax = b, \tag{2.1}$$

where A is a n by n real matrix ($A \in \mathbb{R}^{n\times n}$), and x and b are n-dimensional real vectors ($b, x \in \mathbb{R}^n$) the Gaussian elimination with partial pivoting [89] is performed to find a solution of (2.1). Most commonly, the factored form of A is given by means of matrices L, U, P and Q such that:

$$LU = PAQ, \tag{2.2}$$

where:

- L is a lower triangular matrix with unitary diagonal, i.e. $\exists_{B\in\mathbb{R}^{n\times n}} L = \mathcal{L}_1(B)$
- U is an upper triangular matrix with arbitrary diagonal, $\exists_{B\in\mathbb{R}^{n\times n}} U = \mathcal{U}(B)$
- P and Q are row and column permutation matrices ($P, Q \in \mathbb{P}^n$), respectively.

A simple transformation of (2.1) yields:

$$(PAQ)Q^{-1}x = Pb, \tag{2.3}$$

which in turn, after applying (2.2), gives:

$$LU(Q^{-1}x) = Pb, \tag{2.4}$$

Solution to (2.1) may now be obtained in two steps:

$$\begin{aligned} Ly &= Pb && (2.5)\\ U(Q^{-1}x) &= y && (2.6) \end{aligned}$$

and these steps are performed through so called forward and backward substitutions. The matrices involved in the substitutions are triangular and therefore solving (2.5) and (2.6) is faster than solving (2.1). More precisely, the most computationally intensive part of solving (2.1) is the LU factorization defined by (2.2). This operation has computational complexity of order $O(n^3)$ when A is a dense matrix, as compared to $O(n^2)$ for the substitution phase. Therefore, optimization of the factorization is the main determinant of the overall performance.

When both of the matrices P and Q of (2.2) 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 [152]. Moderate values ($\kappa_p \ll \varepsilon^{-1}$) of the condition number $\kappa_p = \|A^{-1}\|_p \|A\|_p$ guarantee a success for a direct method as opposed to matrix norm and spectrum considerations required for iterative methods.

It is assumed that A is sparse, which may be formulated as

$$n \leq \eta(A) \ll n^2, \tag{2.7}$$

where $\eta(A)$ is the number of (structural) nonzero entries in A. Many authors avoid an explicit definition of sparsity in quantitative terms. It is commonly accepted, though, that the aim is for the amount of fill-in and computational complexity to be proportional to $O(n) + O(\eta(A))$ [51]. If the matrix A is sparse, 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 (2.1). 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 non-zeros 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 (2.2) are non-trivial. Matrix Q induces a column reordering that minimizes fill-in and P permutes rows so that pivots selected during the Gaussian elimination warrant numerical stability.

2.2 Recursive LU Factorization

Figure 2.1 shows the classical LU factorization code which uses Gaussian elimination. Rearrangement of the loops and introduction of blocking techniques can significantly increase performance rates achieved by this code [11, 40]. However, the recursive formulation of the Gaussian elimination shown in Figure 2.2 exhibits superior performance [95, 157]. It does not contain any looping statements and most of the floating point operations are performed by the Level 3 BLAS routines: `xTRSM` and `xGEMM`. These routines achieve near-peak Mflop/s rates on modern computers with deep memory hierarchy. They are incorporated in many vendor-optimized libraries, and also the ATLAS project [56, 163] automatically generates implementations tuned to specific platforms. Some properties of the recursive LU algorithm are established by the following theorems.

Theorems 1 and 2 compare the amount of memory traffic – called I/O for short – for LAPACK's blocked implementation of LU factorization and the recursive one. The memory traffic that is being accounted for in the theorems is the one that takes place between the Level 1 cache (referred to as primary memory) and the rest of the memory hierarchy (most commonly Level 2 cache).

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
        $a_{ij} := \frac{1}{a_{jj}}(a_{ij} - \sum_{k=1}^{j-1} a_{ik}a_{kj}) \quad j = 1,\ldots,i-1$
        $a_{ij} := (a_{ij} - \sum_{k=1}^{i-1} a_{ik}a_{kj}) \qquad j = i,\ldots,n$
    end
end

Figure 2.1: LU factorization function of a dense matrix A that uses Gaussian elimination and does not perform pivoting.

Theorem 1 *([157]) Given a matrix multiplication subroutine whose I/O performance satisfies equation*

$$IO_{MM}(n,n,m) \leq \begin{cases} 3nm + m^2 & \text{if } m \leq \sqrt{M/3} \\ 2nm^2/\sqrt{M/3} + 2nm & \text{if } m > \sqrt{M/3} \end{cases}$$

and a subroutine for solving triangular linear systems whose I/O performance satisfies equation

$$IO_{TS}(m,m) \leq \begin{cases} 2.5m^2 & \text{if } m \leq \sqrt{M/3} \\ m^3/\sqrt{M/3} + m^2 & \text{if } m > \sqrt{M/3} \end{cases}$$

the recursively partitioned LU decomposition algorithm running on a computer with M words of primary memory [(Level 1 cache)] computes the LU decomposition with partial pivoting of an n-by-m matrix using at most

$$IO_{RP}(n,m) \leq 2nm\left(\frac{m}{2\sqrt{M/3}} + \log m\right) + 2n^2(1 + \log m)$$

I/Os. □

Theorem 2 *([157]) Given a matrix multiplication subroutine whose I/O performance satisfies equation ($r \leq s \leq t$)*

$$IO_{MM}(t,r,s) \leq \begin{cases} 2ts + rs + rt & \text{if } r < \sqrt{M/3} \\ 2trs/\sqrt{M/3} + 2ts & \text{if } r \geq \sqrt{M/3} \end{cases}$$

and a subroutine for solving triangular linear systems whose I/O performance satisfies equation ($r \leq s$)

$$IO_{TS}(r,s) \leq \begin{cases} 2rs + r^2/2 & \text{if } r < \sqrt{M/3} \\ r^2s/\sqrt{M/3} + rs & \text{if } r \geq \sqrt{M/3} \end{cases}$$

the right-looking LU decomposition algorithm running on a computer with M words of primary memory [(Level 1 cache)] computes the LU decomposition with partial pivoting of an n-by-m matrix using at least

function `xGETRF`(matrix $A \in \mathbb{R}^{m\times n}$)	
if ($A \in \mathbb{R}^{m\times 1}$)	*If the matrix has just one column.*
$\lvert a_{k1}\rvert := \max\limits_{1\le i\le n} \lvert a_{i1}\rvert$	*Find a suitable pivot – equivalent to the Level 1 BLAS `IxMAX`.*
$a_{11} :=: a_{k1}$	*Exchange a_{11} with a_{k1}.*
$A := \frac{1}{a_{11}} A$	*Scale the matrix – equivalent to the Level 1 BLAS `xSCAL`.*
else begin	
$k := \min\{m, n\}$	
$k_1 := \lfloor k/2 \rfloor$	
$m_1 := m - k_1$	
$n_1 := n - k_1$	*Divide matrix A into four sub-matrices:*
$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$	$\begin{bmatrix} \mathbb{R}^{k_1\times k_1} & \mathbb{R}^{k_1\times n_1} \\ \mathbb{R}^{m_1\times k_1} & \mathbb{R}^{m_1\times n_1} \end{bmatrix}$
`xGETRF`($[A_{11}\, A_{21}]^T$)	*Recursive call.*
`xLASWP`($[A_{12}\, A_{22}]^T$)	*Apply pivoting from the recursive call.*
$A_{12} := \mathcal{L}_1(A_{11})^{-1} A_{12}$	*Perform a lower triangular solve which is equivalent to the Level 3 BLAS `xTRSM` function.*
$A_{22} := A_{22} - A_{21}A_{12}$	*Compute the Schur's complement which is equivalent to a matrix-matrix multiply performed by the Level 3 BLAS `xGEMM` function.*
`xGETRF`(A_{22})	*Recursive call.*
`xLASWP`($[A_{11}\, A_{21}]^T$)	*Apply pivoting from the recursive call.*
end	
end.	

Figure 2.2: Recursive LU factorization function of a dense matrix A equivalent to the LAPACK's `xGETRF` function (partial row pivoting is performed).

(r is so-called blocking factor)

$$IO_{RL}(n,m) \geq \begin{cases} \frac{1}{4}nm^2/\sqrt{M/3} + 2nm & \text{if } r = M/n \\ \frac{1}{4}nm^{3/2} & \text{if } r \approx \sqrt{m} < \sqrt{M/3} \\ \frac{1}{4}nm^{3/2} & \text{if } r = \sqrt{M/3} \end{cases}$$

I/Os. If the matrix is not very large [...], $n^2/3 \leq M$, [...] $\sqrt{M/3} \leq r \leq M/n$ [...] the total number of I/Os is at least

$$\frac{nm^2}{4\sqrt{M/3}} + 2nm \geq \frac{n^2m^2}{4M} + 2nm. \square$$

Theorem 3 provides another insight into the superiority in performance exhibited by the recursive factorization. The theorem introduces quantity called *area of the matrix operands passed to BLAS* . Clearly, a BLAS subroutine references memory locations that hold matrix entries. Such references incur memory bus traffic which needs to be optimized since it is the bottleneck of the factorization. The theorem shows that the recursive algorithm is more optimal with respect to area of the matrix operands passed to BLAS.

Theorem 3 *([95]) Let A be a M-by-N matrix ($A \in \mathbb{R}^{M\times N}$) where $M = mN_B$, $N = nN_B$, and N_B is a blocking factor. The total area of the matrix operands passed to BLAS for the recursive LU algorithm is*

$$A_R(mN_B, nN_B) = (((2 + \lfloor \log n \rfloor)n - 2^{1+\lfloor \log n \rfloor})m + f(n))N_B^2$$

where

$$f(2^k) = 2^k(k(1 - 2\times 2^k) + 5(2^k - 1))/4$$

In particular, if n is a power of 2 ($n = 2^k$) and $m = n$, then

$$A_R(nN_B, nN_B) = 2^{k-2}(k(2^{k+1} + 1) + 5(2^k - 1)]N_B^2.$$

The total area of the LU right-looking LAPACK algorithm `DGETRF` is

$$A_L(mN_B, nN_B) = ((n(n+1)/2)(m - n/3 + 4/3) - m - 1)N_B^2.$$

For $n > 3$, $A_L(m,n) > A_R(m,n)$, and the $A_L(n,n)/A_R(n,n)$ ratio is approximately $\frac{4}{3}n/\log n$. $\square$

Finally, Lemma 1 and Theorem 4 show that asymptotically the recursive and iterative (loop-based) factorizations perform the same number of FP operations.

Lemma 1 *The right-looking LU factorization algorithm of an $m \times n (m \geq n)$ matrix performs $mn^2 - n^3/3 - n^2/2 - n/6$ FP operations.*

Proof Let F_D denote divisions by pivot and F_U – additions and multiplications during the update phase. The outer-product update version of the factorization will be used in the proof.

In each column, all entries below the diagonal need to be divided by the pivot element. And so there are $m-1$ divisions in the first column, $m-2$ in the second, and so on, until the nth column where there are $m-n$ divisions. Thus:

$$\begin{aligned}F_D &= (m-1)+(m-2)+\ldots+(m-n) = \\ &= mn - \sum_{i=1}^{n} i = mn - n(n+1)/2 = mn - n^2/2 - n/2\end{aligned}$$

There are $n-1$ updates (there is no update in the last column) – an update in column i ($1 \le i \le n-1$) is an outer-product update which performs $2(m-i)(n-i)$ FP operations (it may be regarded as a `xGEMM` call with the third dimension equal to one). Thus:

$$\begin{aligned}F_U = {} & 2(m-1)(n-1)+2(m-2)(n-2)+\ldots+ \\ & +2(m-(n-1))(n-(n-1)) = \\ & = 2\sum_{i=1}^{n-1}(mn - i(m+n)+i^2) = \\ & = 2((n-1)mn-(m+n)(n-1)n/2+n(n-1)(2n-1)/6) = \\ & = mn^2 - n^3/3 - mn + mn + n/3\end{aligned}$$

And finally:

$$F_D + F_U = mn^2 - n^3/3 - n^2/2 - n/6 \quad \square$$

Theorem 4 *The recursive LU factorization algorithm of an $m \times n (m \ge n)$ matrix performs $mn^2 - n^3/3 + O(m^2, n^2, mn)$ FP operations.*

Proof For simplicity, it is assumed that m and n are powers of 2. Let $f(m,n)$ denote the number of FP operations in the recursive LU factorization of an $m \times n$ matrix. At each (other than a single column case) recursion level the following operations are performed:

- the left hand side of the matrix is factored recursively – this performs $f(m, n/2)$ FP operations,
- `xTRSM` is called on the top right part of the matrix – this performs $(n/2)^3$ FP operations,
- `xGEMM` is called to compute Schur's complement – this performs $2(n/2)^2(m-n/2)$ FP operations,
- the lower right portion of the matrix is factored recursively – this performs $f(m-n/2, n/2)$ FP operations,

Thus, the function f satisfies the following recursive equation:

$$f(m,n) = f(m,n/2) + (n/2)^3 + 2(n/2)^2(m-n/2) + f(m-n/2,n/2) \tag{2.8}$$

Since for the proof only the high order terms are relevant then function f is of the form:

$$f(m,n) = \alpha mn^2 + \beta n^3 + \gamma m^2 n + \delta m^3 \tag{2.9}$$

Coefficients α, β, γ, and δ are determined by substituting equation (2.9) in equation (2.8):

$$\begin{aligned}\alpha mn^2 + \beta n^3 + \gamma m^2 n + \delta m^3 \equiv\ & \alpha mn^2/4 + \beta n^3/8 + \gamma m^2 n/2 + \\ & + \delta m^3 + n^3/8 + mn^2/2 - n^3/4 + \alpha(m-n/2)n^2/4 + \\ & + \beta n^3/8 + \gamma(m-n/2)^2 n/2 + \delta(m-n/2)^3\end{aligned}$$

To simplify, we immediately observe that:

$$\delta m^3 \equiv \delta m^3 + \delta m^3 \Rightarrow \delta = 0$$

Gathering "mn^2 terms" yields:

$$\alpha mn^2 \equiv \alpha mn^2/4 + mn^2/2 + \alpha mn^2/4 - \gamma mn^2/2$$

and thus:

$$\alpha = \alpha/4 + 1/2 + \alpha/4 - \gamma/2 \Rightarrow \alpha = 1 - \gamma.$$

Similarly "n^3 terms" give:

$$\beta n^3 \equiv \beta n^3/8 + n^3/8 - n^3/4 - \alpha n^3/8 + \beta n^3/8 + \gamma n^3/8$$

so that:

$$\beta = \beta/8 + 1/8 - 1/4 - \alpha/8 + \beta/8 + \gamma/8 \Rightarrow \beta = \frac{\gamma - 1}{3}.$$

To determine the value of γ the "initial condition" $f(2,2) = 2$ is used, which is combined with (2.9): $f(2,2) = 8\alpha + 8\beta + 8\gamma$. This gives $\gamma = 0$ and consequently $\beta = -1/3$ and $\alpha = 1$. Finally then (2.9) may be written as

$$f(m,n) = mn^2 - n^3/3. \square$$

An alternative formulation of the recursive algorithm is proposed here and shown in Figure 2.3. Most notable difference is the lack of pivoting code and additional calls to Level 3 BLAS. Experiments show that this code performs equally well as the code from Figure 2.2. The experiments also provide indications that further performance improvements are possible, if the matrix is stored recursively [93]. A simplified but still recursive storage scheme is proposed and illustrated in Figures 2.4 and 2.5. This scheme causes dense square sub-matrices to be aligned recursively in memory which is a discrete mapping between one dimensional main memory and matrix entries that form a two dimensional array. The recursive algorithm from Figure 2.3 traverses the recursive matrix structure all the way down to the level of a single dense sub-matrix. At this point an appropriate computational routine is called (either BLAS function xGETRF(matrix $A \in \mathbb{R}^{n \times n}$) begin

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

$n_1 := \lfloor n/2 \rfloor$

$n_2 := n - \lfloor n/2 \rfloor$ — *Divide matrix A into four sub-matrices:* $\left[\begin{array}{cc} \mathbb{R}^{n_1 \times n_1} & \mathbb{R}^{n_1 \times n_2} \\ \mathbb{R}^{n_2 \times n_1} & \mathbb{R}^{n_2 \times n_2} \end{array}\right]$

$A = \left[\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array}\right]$

xGETRF(A_{11}) — *Recursive call.*

$A_{21} := A_{21}\,\mathcal{U}(A_{11})^{-1}$ — *Perform a upper triangular solve which is equivalent to the Level 3 BLAS* `xTRSM` *function.*

$A_{12} := \mathcal{L}_1(A_{11})^{-1}A_{12}$ — *Perform a lower triangular solve which is equivalent to the Level 3 BLAS* `xTRSM` *function.*

$A_{22} := A_{22} - A_{21}A_{12}$ — *Compute the Schur's complement which is equivalent to a matrix-matrix multiply performed by the Level 3 BLAS* `xGEMM` *function.*

xGETRF(A_{22}) — *Recursive call.*

end.

Figure 2.3: A recursive LU factorization function suitable for sparse matrices (no pivoting is performed).

or xGETRF). Depending on the size of the sub-matrices (referred to as a block size [11]), 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 due to lower cache miss rate [93]. This observation, and the aforementioned lack of pivoting (that excessively complicates sparse codes) are primary incentives to adopt the code from Figure 2.3 as the base for the sparse recursive algorithm presented below. The pivot growth effect, however, is still a concern as far as numerical stability is concerned and so it is dealt with with rank-one matrix perturbations as described later on (see section 2.5).

The following theorem establishes computational equivalence between the algorithms from Figures 2.2 and 2.3 (the latter is called pivot-free).

Theorem 5 *The recursive pivot-free LU factorization algorithm of a $n \times n$ matrix asymptotically performs $\frac{2}{3}n^3$ FP operations.*

Proof For simplicity, it is assumed that n is a power of 2. Let $f(n)$ denote the number of FP operations in the recursive pivoting-free LU factorization of an $n \times n$ matrix. At each (other than a single element case) recursion level the following operations are performed:

- the upper left part of the matrix is factored recursively – this performs $f(n/2)$ FP operations, 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 2.4: A column-major storage scheme versus a recursive one (left) and a function for converting a square matrix from the column-major to recursive storage (right). Block size is 1.

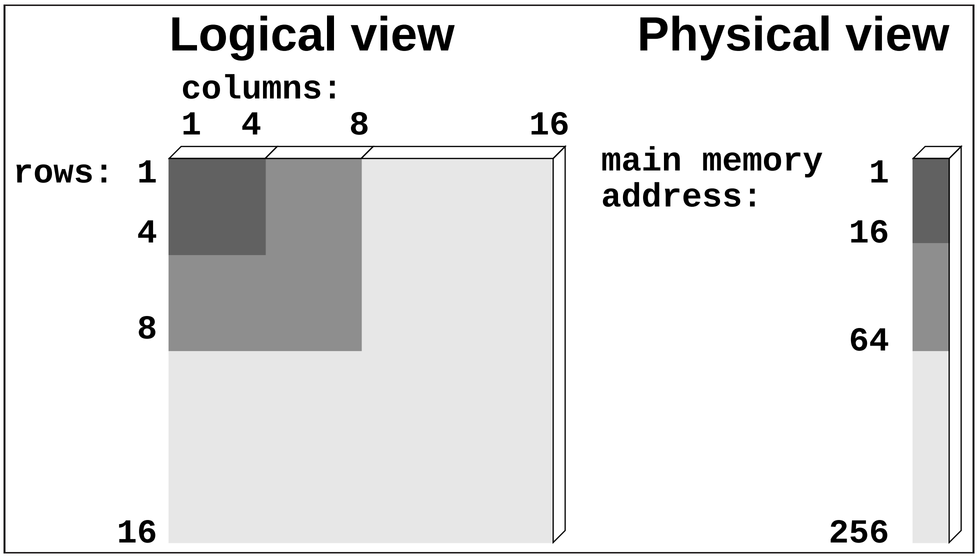


Figure 2.5: Recursive matrix layout in logical (algorithmic) and physical (main memory) views.

- xTRSM is called on the top right part of the matrix – this performs $(n/2)^3$ FP operations,
- xTRSM is called on the bottom left part of the matrix – this performs $(n/2)^3$ FP operations,
- xGEMM is called to compute Schur's complement – this performs $2(n/2)^3$ FP operations,
- the lower right portion of the matrix is factored recursively – this performs $f(n/2)$ FP operations,

Thus, the function f satisfies the following recursive equation:

$$f(n) = f(n/2) + (n/2)^3 + (n/2)^3 + 2(n/2)^3 + f(n/2) \tag{2.10}$$

or more concisely:

$$f(n) = 2f(n/2) + n^3/2 \tag{2.11}$$

It is easily verifiable that the highest order term that satisfies equation (2.11) is of the form: αn^3. By substitution it easily follows that:

$$\alpha n^3 = 2\alpha n^3/8 + n^3/2 \tag{2.12}$$

from which it is established that $\alpha = 2/3$.$\square$

2.3 Sparse Matrix Factorization

Matrices originating from the Finite Element Method [28, 153], 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 FP operations and execution time. To reap the benefits however, one needs to address the major issue of the sparse factorization – the aforementioned fill-in phenomenon. It turns out that a proper ordering of the matrix, represented by the matrices P and Q from equation (2.2), may reduce the amount of fill-in. However, the search for the optimal ordering is an $\mathcal{NP}$-complete problem [165]. Therefore, many heuristics have been devised to find an ordering which approximates the optimal one. These heuristics range from the divide and conquer approaches such as Nested Dissection [76, 116] to the greedy schemes such as Minimum Degree [2, 155]. For certain types of matrices, bandwidth and profile reducing orderings such as Reverse Cuthill-McKee [33, 84] and the Sloan ordering [151] may perform well. Once the amount of fill-in is minimized through an appropriate ordering, it is still desirable to use the optimized BLAS to perform the FP 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 [61] and supernodal [15] methods. The SuperLU package [42, 113] is an example of a supernodal code, whereas UMFPACK [35, 39] is a multifrontal one.

The factorization algorithm for sparse matrices includes the following phases:

- matrix ordering,
  - reduces fill-in (for reduction of memory storage),
  - introduces dense sub-matrices (for efficient use of Level 3 BLAS),

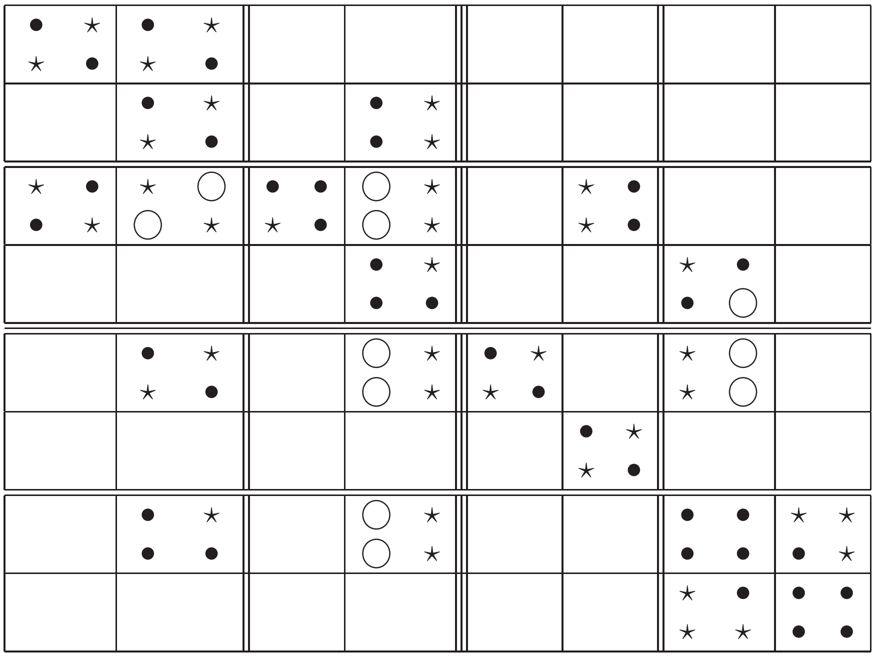
    - transforms matrix so it is almost diagonally dominant (for numerical stability).

- symbolic factorization,
    - estimates memory requirements for the L and U factors,
    - prepares the data structures to store factored matrix.
- numerical factorization,
    - performs Gaussian elimination to compute L and U factors.
- triangular solves,
    - finds the solution of the original system by backward substitution.
- iterative refinement,
    - improves solution found in the previous step,
    - compensates for inaccuracies introducied by lack of pivoting and round-off errors during the factorization.

The first phase – matrix ordering – 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 [60, 115]). Here, 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 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 even in parallel codes [6, 7].

The next phase – symbolic factorization – 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 [85] are efficiently utilized. In essence, for the LL^T (Cholesky) factorization, fill-in may be determined without additional storage with the use of undirected *quotient graphs* [79]. For an unsymmetric matrix the amount of fill-in cannot be bound in a practical way and directed graphs need to be used. It makes the process much slower and therefore, the matrix is implicitly split into two components one of which is symmetric and can take advantage of existing algorithms for Cholesky factorization [65, 66].

The following three phases – numerical factorization, triangular solves, and iterative refinement – share similar goals and implementation techniques. 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 [42, 43, 113]. UMFPACK uses frontal matrices that are formed during the factorization



● – original nonzero value
○ – zero value introduced due to blocking
⋆ – zero value introduced due to fill-in

Figure 2.6: Sparse recursive blocked storage scheme with the blocking factor equal 2.

process. They are stored as dense matrices and may be passed to the Level 3 BLAS. The recursive code uses dense regular sub-matrices – blocks – that include original and fill-in entries and may be directly passed on to BLAS.

2.4 Sparse Recursive Factorization Algorithm

An 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 Figure 2.6. It has the following characteristics:

- the data structure that describes the sparsity pattern is recursive as shown in Figure 2.4,
- the storage scheme for matrix structure and entries has two levels:
  - the (lower) level of numerical values, which consists of dense square sub-matrices (blocks) which enable direct use of the Level 3 BLAS, and
  - the (upper) level of integer pointers, which describes the sparsity pattern of the blocks.

There are two important ramifications of this storage scheme. First, the number of integer indices that describe the sparsity pattern is smaller than in other codes because each of the integer indices refers to a block of values rather than individual values or small set of 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. This is in contrast to a code that relies on sophisticated compiler optimization to produce efficient code for integer computations. Second, the blocking introduces additional nonzero entries that would not be present otherwise. These artificial non-zeros amount to an increase in storage requirements. Also, the arithmetic complexity is higher because floating point operations are performed on the artificial zero values. This leads to the conclusion that the sparse recursive storage scheme performs best when natural dense (or almost dense) blocks exist in the L and U factors of the matrix. Such a structure may be achieved with 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 Figure 2.3 remains almost unchanged in the sparse case – the differences being the calls to BLAS which are replaced by the calls to their sparse recursive counterparts and that the data structures are no longer the same. Figures 2.7, 2.8 and 2.9 show the recursive BLAS routines used by the sparse recursive factorization algorithm. They traverse the sparsity pattern and upon reaching a single dense block they call the BLAS which perform actual FP operations.

The system matrix is converted into a blocked form prior to the factorization. The following provides quantitative rationale as to why the blocked form should not be used to determine the fill-in. First, two matrix operators are defined (a matrix A is assumed to be the set of its entries and $|A|$ is the cardinality of this set, i.e. the number of non-zero entries of A).

Definition 1 *Given an n by n matrix A let $\mathcal{F}(A)$ denote the set of nonzero entries of A and (possibly none) additional entries incurred by structural fill-in resulting from LU factorization.*

Definition 2 *Given an n by n matrix A let $\mathcal{B}_f(A)$ denote a set of nonzero entries of A and (possibly none) additional entries incurred by regular square blocked storage with blocking factor f.*

The intuitive properties of the above operators may be summarized as follows.

Observation 1 *Given an n by n matrix A and operators $\mathcal{F}$ and $\mathcal{B}_f$ (fill-in and blocked form sets, respectively) the following holds:*

1. $\mathcal{F}(A) = \mathcal{F}(\mathcal{F}(A))$
2. $\mathcal{B}_f(A) = \mathcal{B}_f(\mathcal{B}_f(A))$
3. $\mathcal{F}(\mathcal{B}_f(A)) \supseteq \mathcal{B}_f(\mathcal{F}(A)) \supseteq \mathcal{F}(A)$

It is a rather intuitive result that calculating fill-in after the matrix is converted into a blocked form (even though faster) requires more storage than doing it afterwords. The following substantiates intuition.

$\boxed{\begin{array}{l} C := C - A \cdot B \\ A,B,C \text{ are rectangular matrices } (A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}, C \in \mathbb{R}^{m \times n}) \end{array}}$

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

$\boxed{\text{if any of } A, B, C \text{ are smaller than block size } n_B \text{ – use BLAS}}$

if $m < n_B$ **or** $k < n_B$ **or** $n < n_B$ **then**

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

return

end if

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

$\boxed{C_{11} := C_{11} - A_{11} \cdot B_{11}}$

`xRGEMM('N','N',`$\alpha = -1, A_{11}, B_{11},\ \beta = 1, C_{11}$`)`

$\boxed{C_{21} := C_{21} - A_{21} \cdot B_{11}}$

`xRGEMM('N','N',`$\alpha = -1, A_{21}, B_{11},\ \beta = 1, C_{21}$`)`

$\boxed{C_{11} := C_{11} - A_{12} \cdot B_{21}}$

`xRGEMM('N','N',`$\alpha = -1, A_{12}, B_{21},\ \beta = 1, C_{11}$`)`

$\boxed{C_{21} := C_{21} - A_{22} \cdot B_{21}}$

`xRGEMM('N','N',`$\alpha = -1, A_{22}, B_{21},\ \beta = 1, C_{21}$`)`

$\boxed{C_{12} := C_{12} - A_{11} \cdot B_{12}}$

`xRGEMM('N','N',`$\alpha = -1, A_{11}, B_{12},\ \beta = 1, C_{12}$`)`

$\boxed{C_{12} := C_{12} - A_{12} \cdot B_{22}}$

`xRGEMM('N','N',`$\alpha = -1, A_{12}, B_{22},\ \beta = 1, C_{12}$`)`

$\boxed{C_{22} := C_{22} - A_{21} \cdot B_{12}}$

`xRGEMM('N','N',`$\alpha = -1, A_{21}, B_{12},\ \beta = 1, C_{22}$`)`

$\boxed{C_{22} := C_{22} - A_{22} \cdot B_{22}}$

`xRGEMM('N','N',`$\alpha = -1, A_{22}, B_{22},\ \beta = 1, C_{22}$`)`

end.

Figure 2.7: Recursive formulation of the `xRGEMM` function which is used in the sparse recursive factorization to perform the Schur's complement operation.

$B := B \cdot U^{-1}$
U is an upper triangular matrix with non-unitary diagonal without zero entries ($B \in \mathbb{R}^{m\times n}, U \in \mathbb{R}^{n\times n}$)

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

$\boxed{\text{if } U \text{ or } B \text{ are smaller than block size } n_B \text{ -- use BLAS}}$

if $m < n_B$ **or** $n < n_B$ **then**

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

return

end if

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

$\boxed{B_{11} := B_{11} \cdot U_{11}^{-1}}$

`xRTRSM('R','U','N','N',`U_{11}, B_{11}`)`

$\boxed{B_{21} := B_{21} \cdot U_{11}^{-1}}$

`xRTRSM('R','U','N','N',`U_{11}, B_{21}`)`

$\boxed{B_{22} := B_{22} - B_{21} \cdot U_{12}}$

`xRGEMM('N','N',`$\alpha = -1, B_{21}, U_{12}, \beta = 1, B_{22}$`)`

$\boxed{B_{22} := B_{22} \cdot U_{22}^{-1}}$

`xRTRSM('R','U','N','N',`U_{22}, B_{22}`)`

$\boxed{B_{12} := B_{12} - B_{11} \cdot U_{12}}$

`xRGEMM('N','N',`$\alpha = -1, B_{11}, U_{12}, \beta = 1, B_{12}$`)`

$\boxed{B_{12} := B_{12} \cdot U_{22}^{-1}}$

`xRTRSM('R','U','N','N',`U_{22}, B_{12}`)`

end.

Figure 2.8: Recursive formulation of the `xRTRSM` function for upper triangular matrices as it is used in the sparse recursive factorization.

$B := L^{-1} \cdot B$

L is a lower triangular matrix with unitary diagonal without zero entries ($B \in \mathbb{R}^{m\times n}, L \in \mathbb{R}^{m\times m}$)

function `xRTRSM('L','L','N','U',`L`,`B`)`

begin

    if L or B are smaller than block size n_B – use BLAS

    if $m < n_B$ **or** $n < n_B$ **then**

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

        return

    end if

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

    $B_{11} := L_{11}^{-1} \cdot B_{11}$

    `xRTRSM('L','L','N','U',`L_{11}`,`B_{11}`)`

    $B_{21} := B_{21} - L_{21} \cdot B_{11}$

    `xRGEMM('N','N',`$\alpha = -1$`,`L_{21}`,`B_{11}`,` $\beta = 1$`,`B_{21}`)`

    $B_{21} := L_{22}^{-1} \cdot B_{21}$

    `xRTRSM('L','L','N','U',`L_{22}`,`B_{21}`)`

    $B_{12} := L_{11}^{-1} \cdot B_{12}$

    `xRTRSM('L','L','N','U',`L_{11}`,`B_{12}`)`

    $B_{22} := B_{22} - L_{12} \cdot B_{12}$

    `xRGEMM('N','N',`$\alpha = -1$`,`L_{12}`,`B_{12}`,` $\beta = 1$`,`B_{22}`)`

    $B_{22} := L_{22}^{-1} \cdot B_{22}$

    `xRTRSM('L','L','N','U',`L_{22}`,`B_{22}`)`

end.

Figure 2.9: Recursive formulation of the `xRTRSM` function for lower triangular matrices as it is used in the sparse recursive factorization.

Observation 2 *Let A be an n by n matrix and B_{ij}, B_{jk}, B_{ik} ($i > j$ and $k > j$) sub-matrices such that $\mathcal{B}_f(A) \supset B_{ij} \cup B_{jk}$ (sub-matrices B_{ij}, B_{jk} are present in the original matrix) and $\mathcal{B}_f(A) \not\supset B_{ik}$ (sub-matrix B_{ik} is not present in the original matrix) then*

$$\eta(B_{ij} + B_{jk}) > f^2 \Rightarrow B_{ij} \in \mathcal{B}_f(\mathcal{F}(A))$$

and

$$\eta(B_{ij}) + \eta(B_{jk}) > 1 \Rightarrow B_{ij} \in \mathcal{F}_f(\mathcal{B}(A)).$$

In essence, to produce a fill-in sub-matrix it takes for the matrix to have either a certain structure or simply have enough entries. In contrast, if fill-in is calculated on a matrix in blocked form then a single entry per block produces fill-in.

2.5 Pivoting and Numerical Stability of LU Factorization

The main problem in robust implementations of Gaussian elimination as far as numerical stability of the algorithm and error propagation are concerned is so called *pivot growth*. Each column of the matrix is divided by the diagonal element – a *pivot* – of that column. Partial pivoting makes sure through row rearrangements that the pivot is as large in absolute value as possible. However, the recursive LU factorization algorithm that has been chosen for the implementation for sparse matrices does not perform any kind of pivoting. Consequently, a pathological case may be encountered – the current pivot is very small (may be regarded as zero for practical purposes). A rather simple solution is proposed here – the offending pivot value is replaced with $\varepsilon\|A\|$ (ε being the FP precision constant for a given machine). This clearly perturbs the original system from equation (2.1) and the procedure ends up solving

$$(A+E)x = b \tag{2.13}$$

rather than (2.1). Commonly known identities allow to recover the solution of the original system from the solution of the perturbed system. The Sherman-Morrison [148, 149] formula may be used for such a recovery in case of rank-one perturbation, i.e. if only one pivot value had to be replaced:

$$(A + huv^T)^{-1} = A^{-1} - h\frac{A^{-1}uv^TA^{-1}}{1 + hv^TA^{-1}u} \tag{2.14}$$

where: $h \in \mathbb{R}$; $u, v \in \mathbb{R}^n$; $A \in \mathbb{R}^{n\times n}$ and A^{-1} exists.
The more general Sherman-Morrison-Woodbury [129, 164] formula works for higher rank modifications:

$$(A + hUV^T)^{-1} = A^{-1} - hA^{-1}U(I + hV^TA^{-1}U)^{-1}V^TA^{-1} \tag{2.15}$$

where: $h \in \mathbb{R}$; $U, V \in \mathbb{R}^{n\times m}$; $A \in \mathbb{R}^{n\times n}$ and A^{-1} exists.
The Bartlett-Sherman-Morrison-Woodbury [21, 26] formula generalizes the previous result even further:

$$(A + BCD)^{-1} = A^{-1} - A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1} \tag{2.16}$$

where: $A \in \mathbb{R}^{n\times n}$; $B \in \mathbb{R}^{n\times m}$; $C \in \mathbb{R}^{m\times m}$; $D \in \mathbb{R}^{m\times n}$ and A^{-1} exists.
It is worth mentioning that all of the matrix inverses from equation (2.14) do not need to be computed explicitly but rather the result of matrix-vector multiplication needs to be provided. This is readily available through computationally inexpensive backward and forward substitution operations involving the L and U factors of the perturbed matrix A. In equations (2.15) and (2.16), computational complexity of some of the inverses depends on dimensions and numerical properties of the rank modyfing matrices: U, V, B, C, and D.

Chapter 3

Review of Literature

3.1 Dense Linear Algebra Computations

One of the most common computational kernels in linear algebra is the LU factorization of a matrix. In popular software packages [11, 56, 163] this operation, performed on a dense matrix, achieves very good performance on modern architectures through the use of block operations with BLAS [47, 48, 49, 50]. Recently these factorization codes have been formulated and implemented using recursion, achieving further improvement of performance [9, 10, 94, 95, 157]. 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. There two are the determining factors of the performance of sparse code.

As a side remark, it is worth mentioning that the recursive LU codes are related to space-filling curves [98, 130] in the sense of bridging the gap between single dimension of column-oriented pivoting as well as linear memory storage model and an intuitive two-dimensional view of matrix elements. There exist proposals [93] for new storage schemes that would narrow the gap even further. Yet other, arguably more related, research efforts are recursive matrix multiplication algorithms [32, 154] which considerably lower the computational cost of recursive matrix decompositions [157].

Parallel implementation of the recursive factorization algorithm has not been studied as thoroughly as its sequential counterpart and there still exists a need for further improvement [106, 107]. Despite the fact that recursive algorithm uses well studied BLAS routines it is still not clear whether it will be scalable and deliver superior performance rates with respect to the existing parallel codes [31]. The main issue to deal with is the communication cost [145]. The ever-recurring issue of matrix sparsity plays paramount role in the data distribution and work assignment in the case of a distributed memory implementation.

3.2 Direct Sparse Methods for Linear Equations Solving

This section gives historical perspective and a brief overview of the evolution of sparse factorization concepts. The review considers: factorization types (Cholesky, LU), algorithmic approaches (multifrontal, supernodal), and ordering techniques (Nested Dissection, Minimum Degree, Reverse Cuthill-McKee, and Sloan).

Two main factorization approaches: multifrontal [61] and supernodal [15] were originally introduced in Cholesky factorization [147]. It was studied from the graph-theoretic stand point and the *elimination trees* [119] were used to model and speed up factorization process, as they were used in both: symbolic and numerical phases of the factorization. In order to benefit from some of these concepts in LU factorization they had to be generalized for unsymmetric matrices. The structure of such matrices may be represented with directed graphs. Thus, e.g., elimination trees became *elimination dags* (direct acyclic graphs) [85] and symmetrical reductions were used to speed up symbolic factorization [65].

The concept of supernodes from Cholesky factorization [80] could be generalized in number of ways and experimental results led to selection of the best option [42, 113]. The trade-off between speed and precision of the solution was being balanced by introduction of the threshold pivoting [58] and, later on, partial pivoting was replaced by the static pivoting [114].

Historically, evolution of the supernodal approach may be followed with the software packages that implemented it: GP [86], GP-Mod [66], SupCol [64]. The multifrontal approach, on the other hand, was implemented in: MA48 [62], MUPS [5], and UMFPACK [35, 39]. The most recent supernodal code is SuperLU [43, 115] and multifrontal one is MUMPS [6, 7].

In the early codes, matrix ordering was used primarily to reduce fill-in generated by the factorization. It has been proven that finding optimal ordering is $\mathcal{NP}$-complete [165] and therefore numerous heuristics have been devised to approximate the optimal solution to the problem. For structurally symmetric matrices the Nested Dissection ordering [76] was used as it yields minimal fill-in for regular meshes. However, it is also suitable for broader class of matrices [116]. It represents *divide and conquer* (also referred to as *global*) approach to matrix ordering which is in contrast with *greedy* (or *local*) approaches represented by Minimum Degree ordering [155] and its modern variants [3, 81, 117]. It was originally presented as the Markowitz ordering [120] which balances minimum fill-in criterion with numerical stability for unsymmetric matrices which require pivoting. In addition, matrices originating from discretization of certain partial differential equations are amenable to the Reverse Cuthill-McKee [33, 84] and Sloan [151] orderings which strive to minimize matrix bandwidth. After such a transformation, the factored matrix will become a dense banded matrix for which the dense factorization method may be favorable. Yet another function of matrix ordering may be improvement of numerical properties of the matrix so that the factorization process can remain numerically stable without use of pivoting [60].

3.3 Vector and Parallel Algorithms for Sparse Matrix Factorizations

Most likely the most important insight into issues regarding vector and parallel processing is the fact that communication complexity of dense LU factorization [145] is only and asymptotic upper bound for the sparse case. In fact, communication amount and its patterns in the sparse case are determined by the sparsity structure of the matrix and how it is taken advantage of in a particular implementation – hardly a non-trivial remark in the light what has already been said here for sequential codes. In the following, a rather brief overview is presented of relatively recent techniques applied in the area. Since their introduction in the late eighties and early nineties most of the computer architectures that were tested are long gone but some of the ideas are applicable to the contemporary hardware and thus are used (possibly with some modifications) in modern codes. More detailed descriptions are available [57, 96] – they give more technical record of the developments that are only briefly and selectively mentioned here.

Even though the basic sequential algorithm for Cholesky factorization needs to be changed for efficient use on vector computers [4, 13, 15, 34] the changes are not as extensive as those that are required for a parallel implementation. The key concept are supernodes that allow to substantially increase data reuse in vector registers and thus improve computational rates on vector supercomputers [4, 13, 15, 34]. In a similar fashion, supernodes reduce synchronization overhead in a shared memory parallel environment [78, 125] that commonly is used with "pool of tasks" approach that naturally performs load balancing.

A number of interesting experiments have been reported [75, 77, 87, 100] for massively parallel machines of the past. While not as relevant today as they were in the time of their writing, they provide a background for other developments that continue to influence contemporary software. A non-trivial issue of mapping the matrix data on the parallel computing nodes. The more optimal the mapping the less communication that is an unwelcome but unavoidable overhead of the parallel setting. A *wrap mapping* [74] might be a good choice for dense parallel computations but it needs to be modified for the sparse case – a *sparse-wrap* mapping [77] is one of the options. Asymptotically a better alternative for regular finite element grids is a *subtree-to-subcube* mapping [82] that uses the elimination as a guide in data distribution. The same concept may be reused with different granularity: instead column, cliques are used to construct the tree and perform distribution [133]. Even unbalanced trees – the major cause for poor quality mapping with the subtree-to-subcube scheme – may be sometimes rectified with a *tree rotation* technique that improves load balance for the numerical phase of the factorization [118]. Still, it might not be sufficient in certain cases, but a subtree-to-subcube mapping may be generalized with the use of combination of breadth first search on the elimination tree, bin packing algorithm and estimation of workload on each node [73]. A different approach is to extend the concept of Nested Dissection by artificially mapping grid points onto a plain and performing so called Cartesian Nested Dissection [135, 136]. Some of the aforementioned concepts have influenced in one way or another modern sparse solvers like SuperLU [43, 115] and MUMPS [6, 7].

A completely different approach has been taken by some researchers. It is quite closely related to some of the ideas from this writing, as it involves variable blocking [90, 141, 142, 143, 144, 160]. Others, on the other hand, have stressed high quality of the implementation by incorporating many of the techniques described above and delivering extraordinary result of 20 Gflop/s execution rate on the Cray T3D computer [92].

3.4 Iterative Sparse Methods for Linear Equations Solving

Ever since its introduction more than half a century ago, the Conjugate Gradient method [97] has seen a wide range of modifications, additions and extensions [19, 63]. Rather then trying to enumerate them all or even to provide some generalizations, only the most relevant results are given. Namely, the Bi-CGSTAB method [161] is probably the safest choice for unsymmetric matrices. There exist generalizations of the CG method for multiple right hand sides [29, 128]. Also, a more practical termination criterion has been proposed [108].

On the implementation side, majority of effort has gone into improving performance of sparse matrix-vector multiply. Experiments have been performed to utilize small blocking factor and assembly level coding techniques to make sparse computational kernels more suitable for modern RISCs [1, 156]. Also, a more generic approach has been proposed – it relies on compiler optimizations and high quality heuristics developed for the Traveling Salesman Problem as it can be proven to be related to matrix blocking problem [132]. Provisions for blocked sparse data structures are made in general purpose solvers – PETSc being one of them [16, 17, 18]. Such blocking structures may be efficiently found in matrices originating from finite difference and finite element methods [14]. Finally, a rather ambitious project called Sparsity [101, 102] aims at fully automated performance tuning comparable with quite successful efforts in the dense matrix computations arena [24, 56, 163] which makes it pertinent to the Self Adapting Numerical Software (SANS) framework [52].

3.5 Contribution of the Study

Direct methods such as LU factorization are a well know technique for solving problems that involve sparse matrices which originate from the Finite Element Method [28, 153] or other discretization methods of Partial Differential Equations. Their significance grows as the resultant sparse matrices cease to have properties required for efficient application of any of the iterative methods [146].

There are two main approaches in direct methods, namely, multifrontal [61] (which is the generalization of the frontal approach [105]) and supernodal [15]. Both of them tend to exploit efficient linear algebra computational kernels to achieve high Mflop/s rates on the modern computer architectures. Drawing on this experience, the recursive approach introduces yet another technique which could exhibit competitive performance for certain types of matrices.

In the context of iterative methods, a thorough analysis of implementations of the Bi-CGSTAB method [161] is presented. The implementations are tested on the same sparse matrix set as the direct codes to allow true comparison of the two approaches. An additional purpose is to reveal runtime behavior of a more sophisticated (in terms of complexity and robustness) iterative code so that a convincing argument may be made about optimizations that should to be performed. Finally, a result from sparse matrix ordering discipline is borrowed and generalized for unsymmetric matrices to allow possibly very high performance gains in implementation of sparse matrix-vector multiplication.

Chapter 4

Research Design

4.1 Test Matrices

To evaluate performance and behavior of the implementations of the algorithms presented earlier, matrices from Harwell-Boeing [59] and Tim Davis' [36, 37, 38] collections are used. The collections are rather large and contain matrices originating in many scientific disciplines. To reduce the time for tests but at the same time retain viability of findings, only a subset of the matrices is used. The subset is chosen to be the same as the one that was used to evaluate the performance of SuperLU [42, 113]. Matrices from that subset are described in Table 4.1. A disadvantage of using these matrices is relative small size of some of them as compared with other matrices from the aforementioned collections. This means that researchers around the world are solving linear systems of larger size. To defend the choice of matrices presented here, it needs to be pointed out that due to availability of parallel codes, bigger matrices are solved in parallel rather than sequentially. Also, the matrices selected here for tests are from many different disciplines and they have a similar sparsity pattern to bigger matrices from the same disciplines. In addition, the test matrices from Table 4.1 were used by others to compare against many other software packages. Thus, a smaller set of codes needs to be tested here and the saved time and focus could be directed toward other issues.

4.2 Hardware Description

Tables 4.2, 4.3, 4.4, and 4.5 show parameters of processors that are used to conduct the tests. The SGI Octane computer uses a rather typical RISC processor while Pentium III and Pentium 4 are CISCs (but internally they use RISC-type instructions – a microcode). Even though they are so different they benefit from the optimization techniques that were described earlier. The operating systems used were Unix variants: IRIX, Linux, and Solaris.

Table 4.1: Parameters of the test matrices (all of them are square).

Matrix name	n	η	Originating discipline or application
af23560	23560	460 598	fluid flow
ex11	16614	1 096 948	3D steady flow calculation
gemat11	4929	33 185	optimal power flow (western U.S.)
goodwin	7320	324 772	fluid dynamics
jpwh_991	991	6 027	circuit physics
mcfe	765	24 382	astrophysics
memplus	17758	99 147	circuit simulation
olafu			structure from NASA Langley,
(inaccura)	16146	1 015 156	inaccuracy problem
orsreg_1	2205	14 133	petroleum engineering
psmigr_1	3140	543 162	demography
raefsky3	21200	1 488 768	computational fluid dynamics
raefsky4	19779	1 316 789	computational fluid dynamics
saylr4	3564	22 316	petroleum engineering
sherman3	5005	20 033	petroleum engineering
sherman5	3312	20 793	petroleum engineering
wang3	26064	177 168	semiconductor device simulation
venkat01	62424	1 717 792	2D implicit Euler solver for flow simulation

Table 4.2: Parameters of the SGI Octane (running IRIX OS) machine used in tests. The DGEMV and DGEMM rates were measured with ATLAS 3.2.1.

Hardware specifications	
Machine name	SGI Octane
Clock rate	270 MHz
CPU	MIPS R12000
FPU	MIPS R12010
Level 1 instruction cache	32 KiB
Level 1 data cache	32 KiB
Level 2 unified cache	2 MiB
Main memory	256 MiB
Performance of a single CPU	
Peak FP performance	540 Mflop/s
Matrix-matrix multiply – DGEMM	≈450 Mflop/s
Matrix-vector multiply – DGEMV	≈80 Mflop/s

4.3 Numerical Stability Techniques for LU Factorization

LU factorization, as any other numerical algorithm, is prone to roundoff errors and a good quality implementation should provide means to alleviate the problem. Traditionally, the countermeasures included [41]:

- equilibration (prior to factorization),
- pivoting (prior and during factorization),
- iterative refinement (after factorization and backward substitutions).

Equilibration strives to reduce the system matrix' condition number by replacing $Ax = b$ with either $DAx = Db$ or $D_r A D_c x' = D_r b$ with $x = D_c x'$. Similar functionality is available for sparse matrices [60] and is considered during the subsequent testing procedures.

There exist multiple variants of pivoting:

- full,
- partial,
- threshold, and
- static.

But here only the last one is used as it was reported to yield satisfactory results [114] and allows for the numerical factorization phase to be easier to implement and faster to execute. However, if a zero (or almost zero) pivot occurs during the numerical phase then it is replaced by a value that is small relative

Table 4.3: Parameters of the Ultra SPARC II machine (running Solaris OS) used in tests. The `DGEMV` and `DGEMM` rates were measured with ATLAS 3.2.1.

Hardware specifications	
CPU type	Ultra SPARC II
CPU clock rate	296 MHz
System bus clock rate	100 MHz
Level 1 data cache	16 KiB
Level 1 instruction cache	16 KiB
Level 2 unified cache	2 MiB
Main memory	1024 MiB
CPU performance	
Peak FP performance	592 Mflop/s
Matrix-matrix multiply – `DGEMM`	≈450 Mflop/s
Matrix-vector multiply – `DGEMV`	≈55 Mflop/s

Table 4.4: Parameters of the Pentium III machine (running Linux OS) used in tests. The `DGEMV` and `DGEMM` rates were measured with ATLAS 3.2.1.

Hardware specifications	
CPU type	Intel Pentium III
CPU clock rate	550 MHz
System bus clock rate	100 MHz
Level 1 data cache	16 KiB
Level 1 instruction cache	16 KiB
Level 2 unified cache	512 KiB
Main memory	512 MiB
CPU performance	
Peak FP performance	550 Mflop/s
Matrix-matrix multiply – `DGEMM`	≈390 Mflop/s
Matrix-vector multiply – `DGEMV`	≈100 Mflop/s

Table 4.5: Parameters of the Pentium 4 machine (running Linux OS) that was used in tests. The `DGEMV` and `DGEMM` rates were measured with ATLAS 3.2.1.

Hardware specifications	
CPU type	Intel Pentium 4
CPU clock rate	1700 MHz
System bus clock rate	400 MHz
Level 1 data cache	8 KiB
Level 1 instruction cache	12 KiB
Level 2 unified cache	256 KiB
Main memory	512 MiB
CPU performance	
Peak FP performance	3400 Mflop/s
Matrix-matrix multiply – `DGEMM`	≈2200 Mflop/s
Matrix-vector multiply – `DGEMV`	≈350 Mflop/s

to a norm of the matrix: $\|A\|\varepsilon$ (ε being machine FP precision). In this way, the obtained solution satisfies a slightly perturbed system rather than the original one. That, in practice, is as good as the exact solution and (if more correct digits are necessary) is a very good starting point for the iterative refinement process – see below for details.

The iterative refinement method is a well know strategy for adding more significant digits to a solution x_0 of a $Ax = b$ linear system: $x_i = x_{i-1} - A^{-1}(Ax_{i-1} - b)$ $(i = 1, 2, \ldots)$. Based on the thoroughly studied Newton's method for nonlinear equations of the form $f(x) = 0$ with $f(x) = Ax - b$, it performs considerably well especially when the iterative refinement calculations are performed in extended precision FP arithmetic. The method is used in the following tests.

4.4 Programming Language Considerations

The position of the Fortran programming language [103] is undoubtfully very strong in the applied numerical analysis community and many high quality software pieces have been successfully designed and implemented [8, 11, 25, 47, 48, 49, 50, 60] – far too many to try to even count them all. In the context of the recursive LU factorization, however, the shortcomings of the language seem to be more than just annoyances. The problem is posed by the use of recursion. In standard Fortran 77 there is no support for recursion and thus it has to be handled explicitly. Admittedly a matter of opinion and taste, a loop-based implementation of the recursive LU obscures the expressiveness of the C code from Figure 4.1 (the code is just a C implementation of the algorithm presented in Figure 2.2). Seemingly, the problem is solvable in Fortran 90 (and later versions) that permits recursion within the standard. Unfortunately, the overloaded semantics of array operations and pseudo-pointers support forced many implementors of the language to institute semantically safe, but extremely inefficient in practice, data copying procedures that are executed upon every function call. The data copying may be disabled with either command line options or by compiler hints inlcluded in the code, but this defeats portability. –

At this point it should be clear that the C language [104, 110] is preferable to Fortran, at least in the context of this work. In addition, to the aforementioned problems with recursion, C has numerous success stories in the mathematical software field [30, 56, 163, 162] and is very competitive in terms of the quality of the generated assembly code on the targeted systems [88].

Lastly, Python programming language [134, 137, 138, 139, 140] is used since it has a maturing numerical extension [12]. It has also been successfully applied to large scientific projects [22, 23, 99] and to refactor a mixed language sparse eigensolver [83]. On top of the above facts, it encourages interactivity and thus is commonly known for its fast prototyping facilities but at the same time facilitates clarity of style and object-oriented design [159].

4.5 Existing Software Packages for LU Factorization of Sparse Matrices

There is a number of existing software packages available [45] that could be used for sparse LU factorization. Out of them, only the fairly modern ones are considered that are still maintained by their respective

```
int
dgrtrf(int m, int n, double *a,
       int lda, int *ip) {
  int i, k = m < n ? m : n, r=0;

  if (1 == k) {
    double t;
    *ip = i =BLAS_idamax(m,a,1);
    t = a[i];
    if (t != 0.0 && t != -0.0) {
      BLAS_dscal(m,1.0/t, a, 1);
      a[i] = *a;
      *a = t;
    } else return m;
  } else { /* k > 1 */
    int k1 = k >> 1, m2, n2;
    double *a12;
    m2 = m - k1; n2 = n - k1;
    a12 = a + k1 * lda;

    r = LAPACK_dgrtrf(m, k1, a,
                      lda, ip);
    BLAS_dge_permute(k1, n2, ip,
                     1,a12,lda);
    BLAS_dtrsm(blas_colmajor,
               blas_left_side,
               blas_lower,
               blas_no_trans,
               blas_unit_diag,
               k1, n2, 1.0, a,
               lda, a12, lda );
    BLAS_dgemm(blas_colmajor,
               blas_no_trans,
               blas_no_trans,
               m2, n2, k1, -1.0,
               a + k1, lda, a12,
               lda, 1.0, a12+k1,
               lda );
    i = LAPACK_dgrtrf(m2, n2,
                      a12 + k1,
                      lda,
                      ip + k1 );
    BLAS_dge_permute(k - k1, k1,
                     ip + k1, 1,
                     a + k1,
                     lda );
    if (i && ! r) r = i;
    for (ip += k1, k -= k1;
         k!=0; k--) *ip++ += k1;
  }
  return r;
} /* dgrtrf */
```

Figure 4.1: C implementation of the recursive LU factorization with pivoting.

authors:

- SuperLU [42, 113],
- UMFPACK [35, 39],
- MA41 [8],
- Watson Sparse Matrix Package [91] (WSMP).

The SuperLU package is a supernodal code and is available for sequential, shared memory and distributed memory machines[1]. It is portable due to the use of open standards for threading and message passing. The sequential version performs threshold pivoting. The orderings that are available with SuperLU are:

- natural ordering (the system matrix is not reordered prior to factorization),
- Multiple Minimum Degree (MMD) ordering applied to the (implicitly formed) structure of $A^T A$,
- MMD ordering applied to the (implicitly formed) structure of $A^T + A$,
- approximate minimum degree column ordering.

The UMFPACK[2] software package is a multifrontal code and was originally written in Fortran 77 (versions up to and including 2.2) but later on (versions 3.0 and above) it was rewritten in C. The change in programming language allowed for better modularization of the code and rendered the API more flexible – it is now possible to run the symbolic and numerical phases separately. An interesting feature of the package is that during the symbolic factorization the assembly tree is created which is the main data structure used in the numerical factorization for scheduling FP calculations. Since the symbolic factorization depends only on the structure of the matrix, the assembly tree, and the work invested in constructing it, may be reused between matrices that differ only in numerical content. The ordering available for UMFPACK is the Approximate Minimum Degree (AMD) ordering.

An example of a commercial package is MA41 – a symmetric structure multifrontal code. It is written in portable Fortran 77 and can take advantage of an SMP system through threads. It was used as a basis for a distributed memory implementation – MUMPS [6, 7]. MA41 can use AMD ordering and allows for independent use of symbolic and numerical factorizations. The former phase, just as in UMFPACK, creates the assembly tree for the latter phase. Even though MA41 is a multifrontal code somewhat similar to UMFPACK, the two codes differ in the use of symmetry. The former code's performance heavily depends on structural symmetry in the matrix while the latter code performs equally well even for matrices with relatively few symmetric entries.

The last package (certainly not the least, though) is WSMP. It is a multifrontal code that was highly tuned and refined for IBM SP-series systems. It takes advantage of thread-based and message passing paradigms – all at the same time yielding exceptional levels of performance. It is freely available but

[1] It is freely available at http://www.nersc.gov/~xiaoye/SuperLU/.

[2] It is freely available at http://www.cise.ufl.edu/research/sparse/umfpack/.

may only be used on IBM systems since it comes in a form of binary libraries that may be linked by IBM compilers for C or Fortran. It has been excluded from comparisons with the other packages due to its closed-source form and specific hardware demands. This decision is especially regretful since WSMP is very competitive on IBM systems. Still, the reader is referred to the supplied reference to see the performance levels achieved by WSMP.

Due to the recent interest of the IBM company in the Linux operating system, WSMP is now available for Linux clusters from IBM. However, tests of WSMP were not performed for purposes of this study due to the following considerations: a Linux release is a rather recent development (the Linux version was announced at the beginning of 2003 – much later than any of the tests presented later in this writing), its availability is still limited, and the maturity of the release is (understandably) in its early stages which could possibly require extensive tuning to achieve competitive performance.

4.6 Common Optimizations for Iterative Solvers

A straightforward implementation of a commonly accepted rendition of the Bi-CGSTAB algorithm [19] for sparse matrices has a number of preformance problems which cannot be easily resolved by a vast majority of optimizing compilers. An attempt has been made to optimize this reference implementation for the set of test matrices presented earlier. All of the optimizations applied to the code are described in this and the following section.

BLAS cannot be used directly for optimization of the sparse Bi-CGSTAB implementation targetted at the test matrices mentioned before. The reason is twofold:

1. Level 3 BLAS cannot be used because only matrix-vector multiplications are performed and therefore no data reuse is possible to the extent observed for direct methods, and
2. sub-matrix dimensions and vector lengths are much smaller than those for direct codes, mostly because of lack of fill-in.

Regardless, the optimization techniques used by vendors to tune their BLAS implementations may successfully be utilized in the iterative solver computations bearing in mind the aforementioned limitations.

The following are the common practices used for optimization of sparse codes, especially matrix-vector multiplication [1, 16, 17, 18, 55, 83, 88, 101, 102, 132, 156]. Here, they are applied throughout the entire iterative solver code:

- software pipelining,
- functional unit scheduling,
- register blocking,
- matrix reordering,
- decreasing loop overhead through loop unrolling,
- lowering memory traffic,

- data prefetching,
- loop fusion.

Software pipelining makes sure that the pipelined architecture of modern CPUs is taken advantage of. In essence, it is desirable that at any given point in time there are a few (ideally as many as the length of the CPU pipeline) independent instructions in the CPU (certain types of dependences are allowed but for simplicity of explanation they are not described here). All of these instructions will be processed simultaneously and consequently achieve appreciable speed up.

Explicit functional unit scheduling cannot be done portably and not even in the assembly language since tha majority of modern CPUs have dynamic schedulers that decide the order of execution at runtime rather than compilation time. However, since it is known that a dynamic scheduler feeds the execution core from a straight line of code (free of test and jump operations), thus, if a code already contains a dependence-free instruction list, it will be utilized by the scheduler. The previous statement assumes that the compiler will not spoil the handcrafted code – it is a rather safe assumption to make.

Register blocking allows for all the operands and results to reside in the CPU's registers so that no memory loads and stores need to be issued. Since the number of registers is usually very small integer power of 2, register blocking is achieved by grouping matrix entries into sub-matrices of dimensions up to 2 [156] or 3 [83]. For the `xGEMM` operation: $C \leftarrow \alpha AB + \beta C$ it would require up to 18 FP registers – still very few considering how many there are available on some modern CPUs but another important limiting factor is the dense (or almost dense) sub-matrices readily available in the sparse matrix itself and those tend to be of rather small dimensions.

Matrix reordering mainly strives to increase data reuse in cache but may also help with register blocking as the rearranged matrix has more dense sub-matrices. Commonly, the Reverse Cuthill-McKee [33, 84] and Sloan [151] orderings produce favorable band-like matrix structures that for $y \leftarrow \beta y + \alpha Ax$ matrix-vector multiplication allow for better cache utilization as far as entries of x are concerned.

Loop unrolling quite often results from the previously described optimizations and so the compiler usually does not perform the unrolling automatically because it assumes that it has been done already by the programmer to a sufficient extent. However, more often than not it is a false assumption and more unrolling may be performed manually to reduce the loop overhead and expose more instruction-level parallelism to the CPU's dynamic scheduler. This is yet more relevant for sparse computations where the compiler can infer much less information about the data structures which tend to complicated to accommodate matrix' sparsity.

Lowering memory traffic is obviously necessary in any code due to the widening performance gap between memory system and CPU. For sparse iterative codes it is especially important as they rely on memory-bound matrix-vector multiplication and Level 1 BLAS calls as well as transfer of data describing matrix sparsity structure. Reduction of the latter by blocking (i.e. grouping together) nearby entries is a common optimization technique.

Data prefetching is a very useful feature of superscalar RISCs that allows them to request transfers of data to Level 1 cache ahead of time and perform useful operations while the transfers are performed. It is done so that the cache miss penalty is not paid when later on the requested data is used by the CPU. The challenging part is to perform prefetching portably as resorting to low level coding cannot possibly be done for every piece of hardware available today. Extra store or load operations (which are useless from the stand point of the original code) allow to start memory transfer that would fulfill future data request. However, these standard programming techniques do not always allow to bypass compiler's optimizations and emulate functionality provided by specialized assembly instructions for prefetching.

Loop fusion is unlikely to be used in sparse matrix-vector multiplication due to its rather simple structure that (for the reference implementation) involves only two nested loops. For iterative solvers, however, it seems to be a very useful approach. The observation is that a typical iterative code involves many operations equivalent or similar to to Level 1 BLAS. Rather than performing them in sequence one after another, they should be computed jointly whenever possible (some of the operations might depend upon completion of others in which case they need to be performed in sequence). By doing so, a better opportunity is created for application of other optimizations described earlier – manually and automatically by the compiler.

4.7 New Register Blocking Technique

It has been shown that graph compression improves performance of the Minimum Degree ordering for symmetric matrices [14]. A natural extension of this optimization is to use it with unsymmetric matrices for matrix-vector multiplication. Such an extension is described in detail below.

The non-zero structure of an arbitrary sparse n by n matrix $A = [a_{ij}]$ $(1 \le i, j \le n)$ may be modeled with a graph $G_A = (V, E)$ where V is a set of vertices: $V = \{1, 2, \ldots, n\}$ and E is a set of edges: $E = \{(i, j) \mid a_{ij} \neq 0 \ ; 1 \le i, j \le n\} \subseteq V \times V$. An *adjacency set* of vertex $u \in V$ is defined as: $\mathrm{adj}(u) = \{v \in V \mid (u, v) \in E\}$. Vertices u and v are *indistinguishable* if $\mathrm{adj}(u) = \mathrm{adj}(v)$. Since a vertex in G_A corresponds to a (say) row of A, for two indistinguishable vertices their corresponding matrix rows have the same sparsity structure and thus they may share the data that describes this structure. This sharing allows savings in storage, but primarily it lowers memory traffic and reduces the number of fixed-point operations to allow higher FP throughput. Depending on the compression level, the code speed up could be much higher than the one offered by small fixed block sizes. Finally, a successful application of the compression must be able to determine the indistinguishable sets of vertices in a timely manner – a trivial $O(n^2)$ algorithm is by far not practical. Fortunately, the algorithm for symmetric matrices [14] is applicable to nonsymetric ones and thus it is possible to perform graph compression in time $O(|E| + |V| \log |V|)$ or in matrix terms: $O(\eta(A) + n \log n)$. This fast compression algorithm is described next.

The compression algorithm starts with calculating a checksum value for each row of the matrix. Each checksum number is just a sum of integer indices that describe the sparsity pattern of the row (compressed row storage is assumed) – this may be easily calculated in $O(\eta(A))$ time. Next, the checksums are sorted which consumes $O(n \log n)$ time. Finally, a simple (linear in complexity) scan of the checksums allows to determine which rows truly have identical structure since two rows of different structure may have the same checksum.

The graph compression technique helps to take greater advantage of all of the optimizations described in the previous section. A major reason is that it exposes regularity in the matrix' structure that neither the programmer or the compiler can know of prior and at the compilation time.

Chapter 5

Research Findings

5.1 Experiments with Recursive LU Factorization Code for Dense Matrices

In order to better understand the behavior of the dense recursive LU factorization code from Figure 2.3, comparative experiments were performed on the Ultra SPARC II computer. Figure 5.1 shows results of these experiments. The figure compares performance of vendor implementation of `DGETRF` routine (achieving 403 Mflop/s) with the recursive code that uses varying block size. In this experiment, block size is the size of the largest of a square sub-matrix which is not divided any further but passed to vendor BLAS to perform the actual calculations. The recursive code was using the recursive data layout from Figures 2.4 and 2.5. Since the matrix size is a power of 2 (4096 to be exact) the code was much more simplified than it would have been if it was to deal with matrices of arbitrary size. Still, the point is made that recursive layout of matrix data has superior memory access patterns if compared with highly tuned algorithm supplied by the vendor. It needs to be stressed that no floating point operations are performed in the recursive code. Instead, all of them take place in the software libraries supplied by the vendor. Only the scheduling of the operations was different and matrix layout was recursive (the conversion time from Fortran's column-major to recursive layout are not accounted for on the figure but it is only an $O(n^2)$ cost and should become negligible as the matrix dimension increases). Similar results were obtained on the Pentium III computer – they are not included here for brevity and lack of any additional information on top of what is already shown in Figure 5.1.

5.2 Performance Comparison of Direct Codes

Tables 5.1, 5.2, and 5.3 show timing results (the time shown in tables and figures is in seconds unless explicitly stated otherwise) and error estimates for SuperLU Version 2.0 [42, 113] (for sequential machines), UMFPACK Version 3.0 [35, 39] and for the recursive approach. The tables show the total execution time of factorization (including symbolic and numerical phases) and forward error estimates. The matrices used in the tests are selected matrices from the Harwell-Boeing collection [59], and Tim Davis' [36, 37, 38] matrix collection, which were used to evaluate the performance of SuperLU [42, 113].

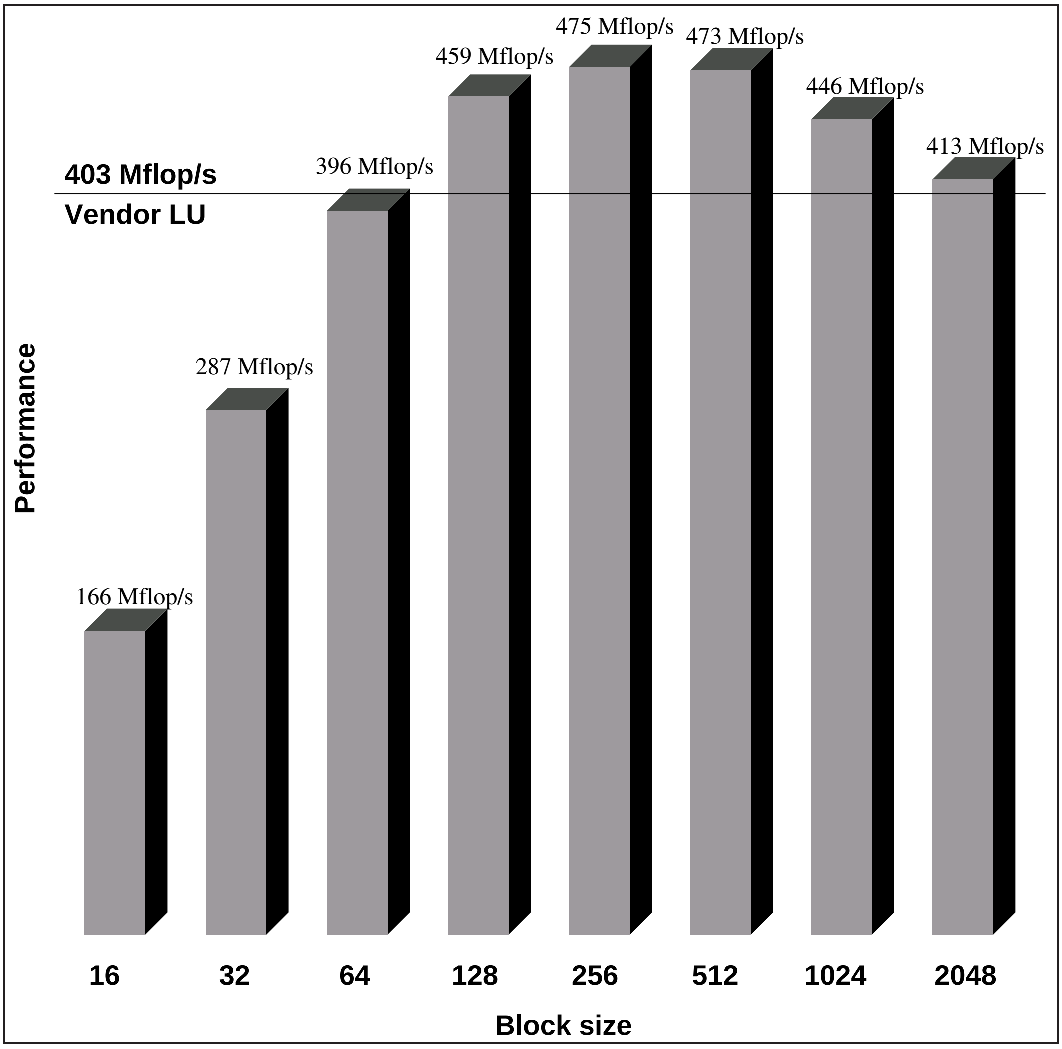


Figure 5.1: Performance of a dense recursive LU factorization with recursive layout of data with varying block sizes compared with vendor LU on the Ultra SPARC II computer for random matrix of size 4096 (the recursive code uses vendor BLAS at a single sub-matrix block level).

Table 5.1: Factorization time (t) and forward/backward error estimates after factorization (ξ_0, r_0) and one iteration of iterative refinement (ξ_1, r_1) for the test matrices on the SGI Octane computer.

Matrix	SuperLU		Recursive algorithm				
name	t [s]	ξ_0	t [s]	ξ_0	ξ_1	r_0	r_1
`af23560`	34.57	7×10^{-14}	34.06	2×10^{-14}	1×10^{-15}	2×10^{-6}	2×10^{-7}
`ex11`	85.75	6×10^{-05}	33.47	3×10^{-07}	2×10^{-07}	2×10^{-7}	7×10^{-7}
`goodwin`	4.48	3×10^{-08}	6.32	5×10^{-09}	3×10^{-11}	7×10^{-5}	1×10^{-6}
`jpwh_991`	0.18	3×10^{-13}	0.16	5×10^{-08}	3×10^{-15}	$3\times10^{+3}$	5×10^{-5}
`mcfe`	0.07	1×10^{-13}	0.06	3×10^{-13}	3×10^{-16}	1×10^{-3}	2×10^{-5}
`olafu`	18.92	1×10^{-06}	18.83	5×10^{-10}	9×10^{-11}	1×10^{-6}	4×10^{-7}
`orsreg1`	0.40	2×10^{-10}	0.22	5×10^{-02}	2×10^{-09}	$9\times10^{+4}$	1×10^{-1}
`psmigr_1`	69.44	7×10^{-11}	45.64	4×10^{-07}	8×10^{-12}	3×10^{-4}	4×10^{-4}
`raefsky3`	46.26	3×10^{-09}	56.2	8×10^{-14}	8×10^{-15}	3×10^{-6}	1×10^{-7}
`raefsky4`	62.15	3×10^{-06}	64.68	3×10^{-07}	2×10^{-06}	4×10^{-6}	6×10^{-7}
`saylr4`	0.61	5×10^{-11}	0.52	2×10^{-10}	3×10^{-11}	5×10^{-6}	2×10^{-6}
`sherman3`	0.45	5×10^{-13}	0.49	2×10^{-13}	1×10^{-13}	1×10^{-6}	7×10^{-7}
`sherman5`	0.23	9×10^{-14}	0.26	2×10^{-15}	5×10^{-15}	1×10^{-5}	3×10^{-6}
`wang3`	73.47	8×10^{-14}	62.33	1×10^{-13}	2×10^{-15}	1×10^{-6}	9×10^{-8}

$\xi_i = \|\hat{x}_i - x\| / \|\hat{x}_i\|$ (relative error at step i)
$r_i = \|A\hat{x}_i - b\| / ((\|A\|\|\hat{x}_i\| + \|b\|)n\varepsilon)$ (backward error at step i)

Table 5.2: Factorization time and forward error estimates for the test matrices for three factorization codes on the Pentium III computer.

Matrix name	SuperLU t [s]	SuperLU FERR	UMFPACK 3.0 t [s]	UMFPACK 3.0 FERR	Recursion t [s]	Recursion FERR
af23560	44.2	5×10^{-14}	29.3	4×10^{-04}	31.3	2×10^{-14}
ex11	109.7	3×10^{-05}	66.2	2×10^{-03}	55.3	1×10^{-06}
goodwin	6.5	1×10^{-08}	17.8	2×10^{-02}	6.7	5×10^{-06}
jpwh_991	0.2	3×10^{-15}	0.1	2×10^{-12}	0.3	3×10^{-15}
mcfe	0.1	1×10^{-13}	0.2	2×10^{-13}	0.2	9×10^{-13}
memplus	0.3	2×10^{-12}	20.1	4×10^{-11}	12.7	7×10^{-13}
olafu	26.2	1×10^{-06}	19.6	2×10^{-06}	22.1	4×10^{-09}
orsreg_1	0.5	1×10^{-13}	0.3	2×10^{-12}	0.5	2×10^{-13}
psmigr_1	110.8	8×10^{-11}	242.6	2×10^{-08}	88.6	1×10^{-05}
raefsky3	62.1	1×10^{-09}	52.4	5×10^{-10}	69.7	4×10^{-13}
raefsky4	82.5	2×10^{-06}	101.9	$5\times10^{+01}*$	104.3	4×10^{-06}
saylr4	0.9	3×10^{-11}	0.7	2×10^{-07}	1.0	1×10^{-11}
sherman3	0.6	6×10^{-13}	0.5	2×10^{-11}	0.7	5×10^{-13}
sherman5	0.3	1×10^{-13}	0.3	4×10^{-12}	0.3	6×10^{-15}
wang3	84.1	2×10^{-14}	100.1	5×10^{-08}	79.2	2×10^{-14}

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

Table 5.3: Total factorization time t, forward error estimate ξ_0, and storage requirement $\eta(L+U)$ for the test matrices for two factorization codes on the Intel Pentium 4 computer.

Matrix	SuperLU			Recursion		
name	t [s]	ξ_0	$\eta(L+U)$	t [s]	ξ_0	$\eta(L+U)$
`af23560`	14.1	5×10^{-14}	132.2	13.9	1×10^{-14}	155.7
`ex11`	36.8	2×10^{-05}	210.2	22.0	1×10^{-06}	159.3
`goodwin`	2.1	1×10^{-08}	31.3	2.9	1×10^{-05}	41.9
`jpwh_991`	0.1	2×10^{-15}	1.4	0.1	2×10^{-15}	3.4
`mcfe`	0.01	1×10^{-13}	0.9	0.1	9×10^{-13}	2.6
`memplus`	0.1	2×10^{-12}	5.9	2.7	4×10^{-13}	98.7
`olafu`	8.3	1×10^{-06}	83.9	9.0	7×10^{-09}	104.0
`orsreg_1`	0.2	1×10^{-13}	3.6	0.2	1×10^{-13}	5.7
`psmigr_1`	35.8	7×10^{-11}	64.6	32.4	1×10^{-05}	78.4
`raefsky_3`	20.0	1×10^{-09}	147.2	27.2	$1\times10^{+00}$	189.5
`raefsky_4`	26.9	2×10^{-06}	156.2	51.7	4×10^{-06}	241.5
`saylr4`	0.3	3×10^{-11}	6.0	0.5	1×10^{-11}	11.7
`sherman3`	0.2	6×10^{-13}	5.0	0.4	9×10^{-01}	8.6
`sherman5`	0.1	1×10^{-13}	3.0	0.2	8×10^{-15}	5.4
`wang3`	29.3	2×10^{-14}	116.7	48.9	1×10^{-14}	246.4

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. UMFPACK supports only one kind of ordering (a column oriented version of the Approximate Minimum Degree algorithm [2, 3]), in addition, the code was used with its default values of the tuning parameters and threshold pivoting disabled. For the recursive approach almost all of the matrices were ordered using Reverse Cuthill-McKee ordering [33] except for `goodwin` and `mcfe` which were used with their natural ordering. For the recursive approach it is possible to select different block sizes, which yield slightly different execution times. Generally, block size 40 seemed to be optimal; however, for some matrices a better time may be obtained with a different block size (the block sizes tried were between 40 and 120). This coincides with the internal blocking factor of ATLAS [56, 163] that was used as a BLAS implementation. To accommodate CPU and memory hierarchy parameters, ATLAS performs computations on 40 rows or columns at a time and does so in an extremely efficient manner. If the number of rows or columns is not divisible by 40, a so called *clean-up code* is involved which is slightly less efficient. Consequently, block sizes that are multiples of 40 are preferred when ATLAS is used.

The total factorization time from Tables 5.1, 5.2, and 5.3 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 group of matrices. 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 non-zeros were introduced by the blocking technique. Whenever this factor drops below 70%, i.e. more than 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 relative and backward errors are similar for all the codes despite the fact that two different approaches to pivoting are used. SuperLU uses threshold pivoting while in UMFPACK and the recursive code there is no pivoting but instead the iterative refinement method is used.

Tables 5.4 and 5.5 show storage requirements and operation counts for the test matrices. On average, it may be observed that SuperLU and UMFPACK use slightly less memory and perform fewer FP operations. This can be attributed to the minimum degree algorithm and its variations used in SuperLU and UMFPACK which minimize the fill-in and thus the space required to store the factored matrix. The large differences between operation counts come from the fact that the recursive approach stores many more floating point values (most of which are zero). This is not evident from the memory requirements because the storage scheme is very sparing in storage of indices. However, it becomes noticeable in the floating point operation counts which is proportional to the third power of the number of FP values. Nevertheless, the performance in terms of time to solution of the recursive code is still very competitive with SuperLU and UMFPACK due to the judicious use of Level 3 BLAS.

Finally, Table 5.6 shows comparison of performance of `MA41` and the recursive code. The table shows only the numerical factorization time since this is the part of the recursive code that was optimized the most. This is in contrast with `MA41` which has been thoroughly tuned in every aspect and in its

Table 5.4: Parameters of the test matrices, their storage requirements and floating point operation counts for SuperLU and the recursive algorithm on the SGI Octane computer.

Matrix parameters			SuperLU		Recursive algorithm		
Name	n	$\eta(A)$	$\eta(L+U)$ [MB]	no. flops [Mflop]	n_B [-]	$\eta(L+U)$ [MB]	no. flops [Mflop]
`af23560`	23560	460598	127.6	4515	50	138.9	5649
`ex11`	16614	1096948	206.7	14249	50	138.1	10320.5
`goodwin`	7320	324772	29.9	506	40	41.4	2097.82
`jpwh_991`	991	6027	1.5	16.1	36	2.3	34.2
`mcfe`	765	24382	0.7	2.7	34	1.2	11.94
`olafu`	16146	1015156	80.7	2960	50	90.0	4937
`orsreg_1`	2205	14133	3.6	43.6	33	3.4	49.05
`psmigr_1`	3140	543162	64.0	9687	45	73.7	20075.4
`raefsky3`	21200	1488768	143.1	7752	50	199.7	17618.5
`raefsky4`	19779	1316789	152.3	10575.6	50	214.3	21593.8
`saylr4`	3564	22316	5.3	62.3	36	6.3	101.55
`sherman3`	5005	20033	4.0	46.2	49	5.8	93.95
`sherman5`	3312	20793	2.3	16.3	40	3.3	44.21
`wang3`	26064	177168	111.6	9966.5	50	214.9	18140

Table 5.5: Parameters of the test matrices and their storage requirements for three factorization codes on the Pentium III computer (for the recursive code the blocking factor n_B for the optimal run is also given).

Matrix parameters			SuperLU	UMFPACK 3.0	Recursion	
Name	n	$\eta(A) \times 10^{-3}$	$\eta(L+U)$ [MB]	$\eta(L+U)$ [MB]	$\eta(L+U)$ [MB]	n_B
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

Table 5.6: Numerical factorization time for two factorization codes – MA41 and the recursive one – on the Pentium III computer. Symmetry factor was reported by MA41 (100% means structural symmetry).

Matrix parameters			Symmetry	MA41	Recursion
Name	n	η	factor [%]	t [s]	t [s]
af23560	23560	460598	100	12.9	16.2
ex11	16614	1096948	100	29.1	30.9
goodwin	7320	324772	64	1.3	4.5
jpwh_991	991	6027	95	0.05	0.2
mcfe	765	24382	71	0.1	0.1
memplus	17758	126150	100	0.2	4.6
olafu	16146	1015156	100	9.8	10.6
orsreg_1	2205	14133	100	0.1	0.2
psmigr_1	3140	543162	48	39.9	60.1
raefsky3	21200	1488768	100	13.6	36.6
raefsky4	19779	1316789	100	34.5	64.5
saylr4	3564	22316	100	0.2	0.5
sherman3	5005	20033	100	0.2	0.3
sherman5	3312	20793	78	0.1	0.2
wang3	26064	177168	100	39.2	52.6

current form benefits from theoretical and implementation optimizations known in the field of sparse direct methods for symbolic as well as numerical phases of the factorization. As the Table 5.6 shows, MA41 is faster on all of the tested matrices except for mcfe for which the times are the same. Still, the recursive code offers comparable performance levels for matrices af23560, ex11, and olafu. For the rest of the matrices MA41 is a clear winner. Further analysis, that is presented later on, reveals the reason for such extraordinarily good behavior. Namely, the structural symmetry factor has very favorable values for the test matrices and MA41 takes full advantage of it.

5.3 Performance Analysis of Direct Codes

This section focuses on analysis of performance of the direct solvers presented earlier. In particular, time breakdowns and timings for synthetic data sets are presented to reveal performance characteristics of each code so it becomes possible to draw more insightful conclusions from the previously presented tests as to under what circumstances and in what manner the software packages should be used.

Table 5.7 gives more insight into performance characteristics of UMFPACK. First and foremost, for matrices memplus, olafu, psmigr_1, and raefsky3 the older Fortran version is significantly faster than the new version written in C. It is hard to imagine that the choice of programming language would trigger such a drastic change, especially bearing in mind that for all the other matrices the new version clearly outperforms the older one. More likely explanation is the change in internal algorithms when migrating

Table 5.7: Breakdown of time spent in factorization for UMFPACK 3.0 in C and UMFPACK 2.2 in Fortran on the Pentium III computer. (Threshold pivoting was disabled; Fortran version did not allow separation of factorization phases.)

Matrix	UMFPACK 3.0			UMFPACK 2.2
name	Symbolic	Numerical	Combined	Combined
af23560	4.59	23.95	28.54	71.47
ex11	5.35	61.67	67.02	268.54
goodwin	0.89	18.08	18.97	17.19
jpwh_991	0.03	0.11	0.14	0.09
mcfe	0.06	0.1	0.16	0.12
memplus	6.78	11.74	18.52	0.65
olafu	3.04	22.86	25.9	13.85
orsreg_1	0.04	0.31	0.35	0.51
psmigr_1	3.31	247.57	250.88	61.34
raefsky3	1.59	51.17	52.76	45.74
raefsky4	5.10	97.17	102.27	134.04
saylr4	0.07	0.51	0.58	0.88
sherman3	0.06	0.36	0.42	0.71
sherman5	0.08	0.24	0.32	0.63
wang3	0.66	99.36	100.02	331.42

to C from Fortran. It seems to be supported by more flexible interface in the C version which allows to perform symbolic and numerical phases separately (and possibly reuse the former for structurally identical matrices.)

Figures 5.2 and 5.3 show time breakdown for the recursive code for block sizes 40 and 100, respectively. The breakdown lists relative time spent in the following phases of the algorithm:

- Symbolic factorization – accommodating storage for fill-in entries,
- Block conversion – converting the original matrix and the fill-in entries into blocked form so that dense sub-matrices rather than single entries are accounted for,
- Recursive conversion – converting the blocked form of the matrix into the recursive layout; it is equivalent to a sort function on the blocks (sub-matrices),
- Search for embedded blocks – withing each non-zero block sub-matrix, the smallest sub-matrix is determined that encloses original non-zeros or fill-in entries; the block sub-matrix region that is outside of an embedded block is not being used in FP calculations,
- Numerical factorization – the actual operations FP operation that comprise Gaussian elimination are performed.

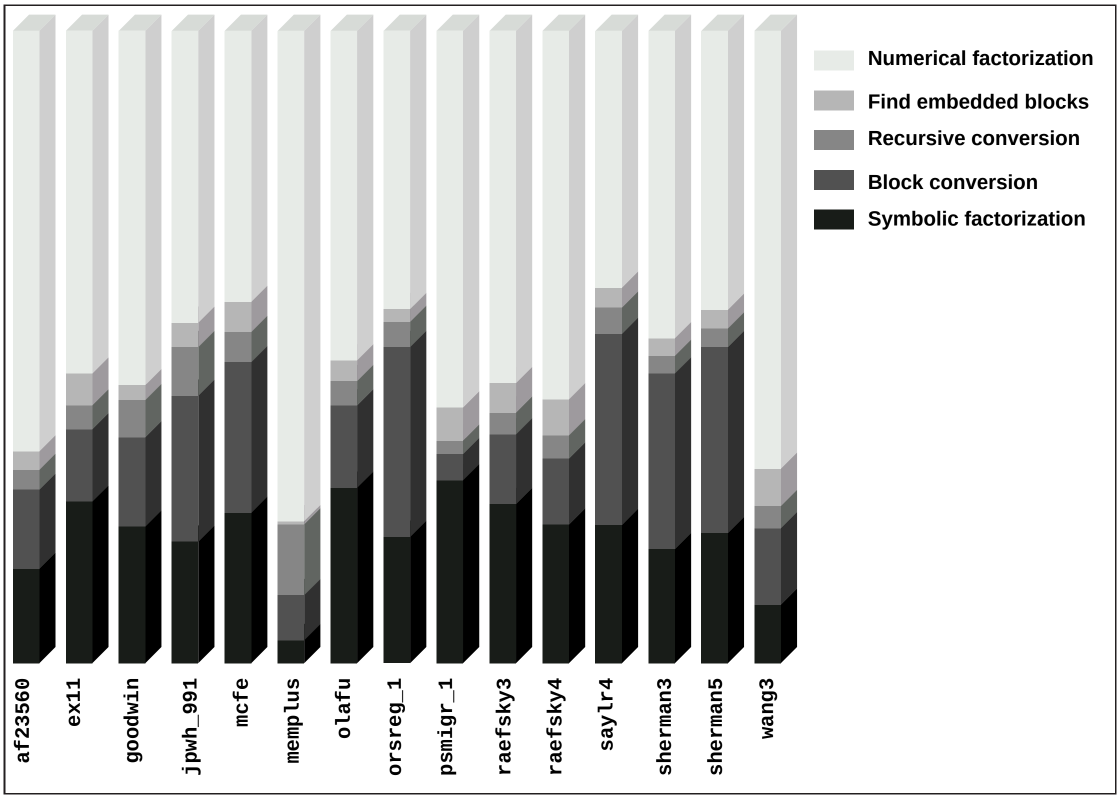


Figure 5.2: Breakdown of time spent in the recursive factorization code with block size 40 on the Pentium III computer.

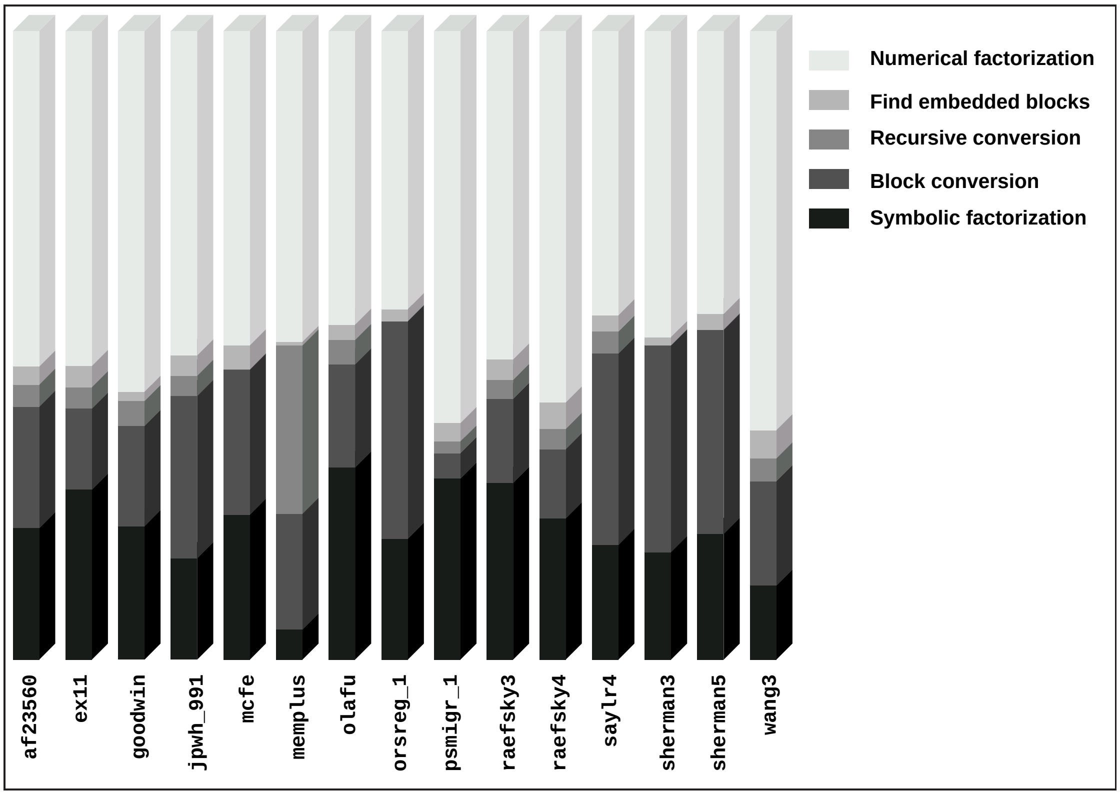


Figure 5.3: Breakdown of time spent in the recursive factorization code with block size 100 on the Pentium III computer.

Comparison of Figures 5.2 and 5.3 results in a predictable conclusion that for larger block sizes more FP operations are performed. This may be easily explained by the fact that for larger block sizes more artificial non-zero entries are introduced and therefore there is more data to operate on. The difference is the most conspicuous for the `memplus` matrix which is more sparse than any other matrix from the tested group. Such an insight offers a guideline for selecting a block size – a balance must be achieved between performance of BLAS – the block size cannot be too small – and excessive number of FP operations – the block size cannot be too large.

Table 5.8 compares numerical factorization times of various block-oriented sparse direct codes (total factorization time is approximately twice as large as the numerical part alone, refer to Figures 5.2 and 5.3 for detailed time breakdown). The table compares two types of codes: the recursive one and a block one operating on symmetric matrix structure (it is a straightforward, loop-based, symmetric-structure code that instead of individual values operates on the same submatrices that the recursive algorithm uses). The former can handle matrices with unsymmetric structure. The latter assumes that the system matrix is structurally symmetric and consequently have the storage and processing is needed to describe and operate on matrix' structure. If, however, the original matrix is not symmetric in structure then artificial non-zero entries are introduced such that the structure of the matrix becomes symmetric but numerical equivalence is retained. Needless to say, such a naïve procedure might be disastrous in terms of performance for matrices whose structure is far from symmetric and many artificial non-zero entries need to be introduced. The block symmetric code was implemented in C [104, 110] and Python [134] programming languages. A number of interesting observation can be made. On average the recursive code is slightly faster than the iterative code even though the iterative code has roughly twice as little integer arithmetic to perform. This may be attributed to superior memory performance of the recursive code. It also explains why the larger block size yields for quite a few matrices better performance as far as the recursive code is concerned whereas the iterative code almost invariably has worse performance for the larger block size (except for the `psmigr_1` matrix which is very dense after fill-in and could use as large block size as possible). The column with timings for the Python code shows how few integer operations are performed in the symmetric code. The Python implementation is comparable to its C counterpart even though it is being interpreted (rather than first compiled and then executed on the CPU). The Python code uses the same optimized BLAS as the other two codes and clearly it is the main prerequisite for performance.

Table 5.9 shows results from experiments on randomly generated skyline matrices. The matrices were generated so they had a changing symmetry factor, i.e. the ratio of all non-zero matrix entries to the number of entries in the minimal numerically equivalent matrix that was symmetric in structure. The ratio is 0% for matrices without any symmetry and 100% for structurally symmetric matrices. The experiments were mainly to show the weakness of the `MA41` code – its dependence on matrix' structural symmetry. While on the tests matrices `MA41` performed exceptionally well (see Table 5.6), it is rather slow for highly structurally unsymmetric matrices and becomes more and more competitive as the matrix' inherent structural symmetry rises. In a sense though, it cannot become competitive enough since the blocked symmetric code, that takes advantage of symmetry just like `MA41`, is almost consistently better. Surprisingly, SuperLU seems to be well (if not the best) suited for this experimental setting having extremely good overall performance, especially bearing in mind that for SuperLU the total time is reported rather than just numerical factorization time. This turns out to be even more unexpected considering the

Table 5.8: Numerical factorization times for various block-oriented codes on the Pentium III computer. For the implementation in the Python programming language the time and block size of the fastest run are shown.

Matrix	Recursive		Symmetric		Python symmetric	
name	$n_B = 40$	$n_B = 100$	$n_B = 40$	$n_B = 100$	t [s]	n_B
`af23560`	30.58	17.51	16.2	26.7	17.26	40
`ex11`	31.51	29.23	30.9	38.7	33.98	40
`goodwin`	3.81	3.92	4.5	6.5	7.08	40
`jpwh_991`	0.12	0.16	0.2	0.3	0.21	40
`mcfe`	0.09	0.13	0.1	0.3	0.13	20
`memplus`	7.63	3.6	4.6	16.1	5.52	30
`olafu`	12.86	10.37	10.6	18.1	16.39	40
`orsreg_1`	0.22	0.23	0.2	0.5	0.26	40
`psmigr_1`	50.11	52.89	60.1	56.7	57.92	100
`raefsky3`	42.61	35.91	36.6	44.4	38.41	40
`raefsky4`	61.45	60.76	64.5	75.1	71.02	40
`saylr4`	0.39	0.52	0.5	1.2	0.51	40
`sherman3`	0.35	0.37	0.3	0.6	0.36	40
`sherman5`	0.15	0.18	0.2	0.4	0.17	40
`wang3`	70.20	50.80	52.6	61.2	54.85	40

Table 5.9: Numerical (total in the SuperLU case) factorization time for random skyline matrices of order 30000 for various direct codes. Symmetry factor is $\eta(A)/\eta(A+A^T)$ and is 100% for structurally symmetric matrices.

Symmetry factor [%]	`MA41` t [s]	Unsymmetric recursion, t [s]	Symmetric blocked, t [s]	SuperLU t [s]
37	22.7	11.6	38.6	9.7
56	36.8	19.9	35.7	19.9
64	54.5	34.5	34.7	29.9
67	136.1	153.2	35.6	39.9

fact that SuperLU does not use Level 3 BLAS – only Level 2 or lower.

5.4 Improving Performance of an Iterative Solver

Table 5.10 shows performance results of a reference (unoptimized) implementation of sparse matrix-vector multiplication routine in row-oriented storage format together with a code that utilizes the graph compression technique mentioned earlier (see section 4.7) which allows to take better advantage of standard RISC-oriented optimizations (also described earlier – see section 4.6). The set of the test matrices, although not selected for practical purposes of this study, but rather influenced by external references, turned out to be diversified enough to show both strengths and weakness of the optimization based on graph compression. For matrix `gemat11` a 100% performance improvement was possible. While still it might be due to really low performance of the reference code, nevertheless, for this matrix there should be a significant improvement against any other known optimization techniques. Such a good result should be mainly attributed to a high level compression that was possible for this matrix. Unfortunately, the Bi-CGSTAB algorithm does not converge for this matrix and so the gains from the optimization cannot be used to obtain a solution to a linear system (only to detect quicker a breakdown of the solver). On the other hand for quite many (5 in all) matrices there was no visible improvement against reference implementation. For these matrices, no graph compression occurs – the original and compressed graphs are isomorphic. This suggests that the graph compression method should be supplied with other blocking techniques to get at least improvement level. On the positive side, it has to be noted that compression did not slow the multiplication routine. Other than the time that was used to compress the graph, no additional overhead is incurred even if the optimization does not pay off.

Tables 5.11 and 5.12 show performance results on the Pentium III computer for reference (unoptimized) and optimized implementations, respectively, of the Bi-CGSTAB algorithm. Similarly, Tables 5.13 and 5.14 show results for unoptimized and optimized codes but on the Pentium 4 computer. The tables include timings for matrix `mat64x32` which results from circular boundary conditions on a regular grid – it may be regarded as a generic matrix for many model 2D problems that are commonly tackled with iterative methods. To better show how time is distributed between various components of the codes, the time spend in matrix-vector routine (the routine itself is called twice for every iteration – each

Table 5.10: Performance of the reference and optimized sparse matrix-vector multiplication routines on the Intel Pentium 4 computer.

Matrix name	Reference [Mflop/s]	Optimized [Mflop/s]	Improvement [%]
af23560	262.5	295.3	12.5
ex11	279.1	294.2	5.4
gemat11	153.6	307.2	100.0
goodwin	267.6	291.9	9.1
mcfe	240.0	240.0	0.0
memplus	195.5	195.5	0.0
olafu	272.2	287.7	5.7
psmigr_1	285.0	300.9	5.6
raefsky3	268.8	301.7	12.2
raefsky4	269.1	286.7	6.5
saylr4	205.3	205.3	0.0
sherman3	175.3	175.3	0.0
sherman5	191.4	191.4	0.0

Table 5.11: Performance data for the reference (not optimized) Bi-CGSTAB implementation on the Intel Pentium III computer.

Matrix Name	Number of Iterations	Correct Digits	$\|\|Ax-b\|\|_\infty$ [-]	t_{Total} [s]	$t_{\texttt{SDGEMV}}$ [s]	Perf. [Mflop/s]	tol [-]
jpwh_991	39	9	2×10^{-09}	0.028	0.016	91	10^{-09}
mat64x32	1739	7	2×10^{-09}	66.5	47.4	91	10^{-09}
memplus	1499	5	1×10^{-10}	41	26	53	10^{-09}
orsreg_1	545	6	9×10^{-06}	0.95	0.56	69	10^{-09}
psmigr_1	1524	3	1×10^{-06}	95.5	93.1	44	10^{-12}
saylr4	3387	11	6×10^{-11}	12.3	7.9	73	10^{-09}
sherman3	22737	2	4×10^{-08}	100.4	54.0	53	10^{-14}
sherman5	2259	6	2×10^{-06}	7.0	4.2	59	10^{-09}
venkat01	7317	2	1×10^{-06}	1744	1436	33	10^{-07}
wang3	264	8	7×10^{-12}	10.5	6.4	38	10^{-09}

Table 5.12: Performance data for the optimized Bi-CGSTAB implementation on the Intel Pentium III computer.

Matrix Name	Number of Iterations	Correct Digits	$\|\|Ax-b\|\|_\infty$ [-]	t_{Total} [s]	$t_{\texttt{SDGEMV}}$ [s]	Perf. [Mflop/s]	`tol` [-]
`jpwh_991`	39	9	5×10^{-09}	0.025	0.014	82	10^{-09}
`mat64x32`	1765	7	2×10^{-09}	62.3	43.0	38	10^{-09}
`memplus`	1474	5	2×10^{-10}	43	27	34	10^{-09}
`orsreg_1`	484	6	9×10^{-06}	0.73	0.43	79	10^{-09}
`psmigr_1`	1399	3	1×10^{-06}	83.1	80.9	38	10^{-12}
`saylr4`	3730	11	6×10^{-11}	11.7	6.8	60	10^{-09}
`sherman3`	17849	2	4×10^{-08}	78.7	43.1	51	10^{-14}
`sherman5`	2478	6	2×10^{-06}	7.0	4.4	62	10^{-09}
`venkat01`	6826	2	1×10^{-06}	1652	1354	36	10^{-09}
`wang3`	259	8	7×10^{-12}	9.8	5.9	38	10^{-09}

Table 5.13: Performance data for the reference (not optimized) Bi-CGSTAB implementation on the Intel Pentium 4 computer.

Matrix Name	Number of Iterations	Correct Digits	$\|\|Ax-b\|\|_\infty$ [-]	t_{Total} [s]	$t_{\texttt{SDGEMV}}$ [s]	`tol` [-]
`jpwh_991`	39	9	2×10^{-09}	0.006	0.003	10^{-09}
`mat64x32`	1739	7	2×10^{-09}	10.1	6.0	10^{-09}
`memplus`	1499	5	1×10^{-10}	7.17	3.9	10^{-09}
`orsreg_1`	545	6	9×10^{-06}	0.22	0.13	10^{-09}
`psmigr_1`	1524	3	1×10^{-06}	11.22	10.7	10^{-12}
`saylr4`	3387	11	6×10^{-11}	2.46	1.4	10^{-09}
`sherman3`	22737	2	4×10^{-08}	21.17	10.8	10^{-14}
`sherman5`	2259	6	2×10^{-06}	1.58	0.9	10^{-09}
`venkat01`	7317	2	1×10^{-06}	250	186	10^{-07}
`wang3`	264	8	7×10^{-12}	1.69	0.9	10^{-09}

Table 5.14: Performance data for the optimized Bi-CGSTAB implementation on the Intel Pentium 4 computer.

Matrix Name	Number of Iterations	Correct Digits	$\|\|Ax-b\|\|_\infty$ [-]	t_{Total} [s]	$t_{\texttt{SDGEMV}}$ [s]	`tol` [-]
`jpwh_991`	39	9	5×10^{-09}	0.007	0.003	10^{-09}
`mat64x32`	1765	7	1×10^{-09}	9.6	5.8	10^{-09}
`memplus`	1474	5	2×10^{-10}	6.8	3.7	10^{-09}
`orsreg_1`	484	6	9×10^{-06}	0.19	0.1	10^{-09}
`psmigr_1`	1399	3	1×10^{-06}	9.98	9.5	10^{-12}
`saylr4`	3730	11	6×10^{-11}	2.7	1.6	10^{-09}
`sherman3`	17849	2	4×10^{-08}	16.6	8.4	10^{-14}
`sherman5`	2478	6	2×10^{-06}	1.8	1.0	10^{-09}
`venkat01`	6826	2	1×10^{-06}	222	169	10^{-09}
`wang3`	259	8	7×10^{-12}	1.6	0.8	10^{-09}

call is timed and accumulated for all calls) $t_{\texttt{SDGEMV}}$ is also included in the tables. Judging by the number of publications on improvements made to the sparse matrix-vector multiplication routine, it should be natural to conclude that this is where most of the time is spent (barring the use of a preconditioner). The data from the tables to an extent disagree with this assertion – there are matrices (`jpwh_991`, `sherman3`, and `sherman5`) for which only half of the time is spent in the multiplication routine. Thus, it is justifiable to optimize the rest of the code as it was done for the optimized version in Tables 5.12 and 5.14. By using all the optimizations described earlier (see section 4.7), except for graph compression whose influence was shown earlier, it is possible to reduce the time to solution by about 10% on average. An interesting result was obtained for matrix `wang3` for which the time is much shorter than for any of the direct methods presented in the preceding sections. The only superiority exhibited by direct methods is accuracy of the solution – twice as many correct digits may be worth spending about six times as much on computations. The rest of the matrices do not pose such a dilemma – any of the direct methods are superior in terms of time and accuracy. The difference is not in order of magnitude, though.

Another comment is due at this point as to why time to solution was used as the performance metric in Tables 5.11, 5.12, 5.13, and 5.14. It might be tempting to report time per iteration and compare it for codes prior and after optimizations. This is especially true bearing in mind that the unoptimized and optimized codes perform different number of iterations (for most of the matrices the optimized code performs less iterations but the difference does not exceed 10%) to achieve the same level of accuracy and trigger the stopping criterion. However, a user-centric view was adopted in reporting the data for the tables. From the user perspective, only the time to obtain a solution (or indication of failure for that matter) and its quality is important. The details of the implementation are of a lesser, if not the least, concern.

Finally, it should be mentioned that Tables 5.11, 5.12, 5.13, and 5.14 do not show results for all the matrices that were used with the direct solvers in the preceding sections, the reason being lack of convergence for the missing matrices. From a theoretical standpoint an iterative method that does not converge should be replaced with a different method. From the user standpoint, however, learning about the failure is as time consuming as delivering a satisfactory solution. Consequently, even an optimized failing code has an advantage over an unoptimized one as the former is able to provide quicker information about failure.

Chapter 6

Conclusions and Implications

6.1 Concluding Remarks

This research completed the set of theoretical results pertaining to the recent algorithmic advances in decompositional techniques of numerical computational linear algebra for dense matrices [67, 95, 157]. In addition, a new formulation of recursive LU factorization algorithm has been proposed and applied in sparse matrix computations. The application was compared with modern software packages. The comparison yielded promising results on a range of modern computing platforms. Aside from competitive studies, thorough performance analysis and time breakdown were presented. Also, mixed-language implementation was tested to give yet greater insight into runtime characteristics of the proposed algorithm.

In the area of iterative methods, time breakdown was presented for the Bi-CGSTAB algorithm while applying it to an interdisciplinary set of sparse matrices. Based on the obtained results, a more holistic approach to optimization of Bi-CGSTAB was shown. The approach incorporates a newly proposed generalization of theoretical advances in symmetric matrix orderings [14] which results in appreciable speed-up of matrix-vector multiplication for certain types of sparse matrices. Additionally, a set of known optimizations was used to improve performance of the remaining part of the Bi-CGSTAB algorithm to provide yet more gains in terms of efficiency and revealing some convergence subtleties which, however, do not diminish the improvements.

6.2 Recommendations for Further Research

The material from this writing could conceivably be extended in ways that might lead to potentially useful results.

The use of recursive matrix layout (or any of its suitable variants [93]) could be incorporated into existing multifrontal or supernodal approaches. This could shed more light on an interesting question as to where the performance comes from in the current implementation, whether it is the algorithm or the data layout.

Extending the existing code so it handles variable size sub-matrices is also an interesting undertaking as it has been successfully used before [90, 141, 142, 143, 144, 160] but without recursive algorithm. Another possibility is to let the recursive algorithm be guided by symbolic data structures like an elimination tree or assembly tree that are naturally created during the symbolic phase of the factorizations. In this context, different recursive tree traversal schemes give wide range of research possibilities. And finally, employing the recursive code as a preconditioner might yield a viable solution in terms of efficiency and numerical terms. A new notion of incompleteness in preconditioning is required, though.

In the context of vector and parallel processing, the basic precept of reduction of memory traffic by the recursive algorithm needs to be revisited and somehow achieve properties that currently favor block-cyclic schemes [131]. Moreover, the use of recursion could be a welcome innovation for factorization codes that need to initially obtain the data from a disk and stage the factored matrix on a disk afterwords.

Certainly, the graph compression technique could use an improvement for matrices without large subsets of vertices with identical adjacency sets – the very matrices that show only moderate, none at all, performance increase. The requirement for the adjacency sets to be identical simplifies and speeds-up the implementation but on the other hand makes its success extensively dependent on matrix structure. Experiments with efficient algorithms for 0-1 matrices [27] could give some insight into possible progress in this areas. Also, naïve algorithms (with exponential computational complexity) for testing similarities (rather than identity) between adjacency sets could be tackled with techniques from the field of Fixed Parameter Tractability [68, 69, 70, 71, 111, 112].

As far as iterative methods are concerned, it would be interesting to see how much slower an iterative solver is for, say, two right-hand sides. If it is comparable, then it might be possible to iterate simultaneously on more than one vector which would increase robustness of any iterative method. Another question is how these simultaneous vectors and their corresponding right-hand sides should be related to yield faster convergence.

The results for block variants of popular iterative solvers do not give enough guidance for possible implementations. The common result on convergence rate for the CG algorithm states that [128, 146]:

$$\|x - x_i\|_A \leq 2\|x - x_0\|_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^i \tag{6.1}$$

for $Ax = b$ linear system with κ being a condition number of A and x_i being the iterates. The result (6.1) is proven for each component separately in the case of a block method with multiple right-hand side. Maybe a better monitoring strategy be possible if one or more of the iterated vectors is known from the onset – this would probably require a generalization of equation (6.1).

Bibliography

Bibliography

[1] R. Agarwal, Fred Gustavson, and M. Zubair. A high-performance algorithm using preprocessing for the sparse matrix-vector multiplication. In *Proceedings of International Conference on Supercomputing*, 1992.

[2] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree algorithm. Technical Report TR/PA/95/09, CERFACS, Toulouse, France, 1995.

[3] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, October 1996.

[4] Patrick R. Amestoy and Iain S. Duff. Vectorization of a multiprocessor multifrontal code. *Internat. J. Supercomp. Appl.*, 3:41–59, 1989.

[5] Patrick R. Amestoy and Iain S. Duff. MUPS: a parallel package for solving sparse unsymmetric sets of linear equations. Technical report, CERFACS, Toulouse, France, 1994.

[6] Patrick R. Amestoy, Iain S. Duff, Jacko Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RT/APO/99/2, ENSEEIHT-IRIT, Toulouse, France, 1999.

[7] Patrick R. Amestoy, Iain S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, pages 501–520, 2000.

[8] Patrick R. Amestoy and Chiara Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM J. Matrix Anal. Applic.*, 24(2):553–569, 2002.

[9] B. Andersen, Fred Gustavson, A. Karaivanov, Jerzy Wasniewski, and P. Yalamov. LAWRA – Linear Algebra with Recursive Algorithms. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, 1999.

[10] B. Andersen, Fred Gustavson, and Jerzy 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, 24-27 March 1999.

[11] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, Third edition, 1999.

[12] D. Ascher, P. F. Dubois, K Hinsen, J. Hugunin, and T. Oliphant. *Numerical Python*. Lawrence Livermore National Laboratory, September 2001.

[13] Cleve Ashcraft. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. Technical Report ETA-TR-51, Engineering Technology Applications Divisions, Boeing Computer Services, Seattle, WA, 1987.

[14] Cleve Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Scientific and Statistical Computing*, 16:1404–1411, 1995.

[15] Cleve Ashcraft, R. Grimes, J. Lewis, Barry W. Peyton, and Horst Simon. Progress in sparse matrix methods in large sparse linear systems on vector supercomputers. *Intern. J. of Supercomputer Applications*, 1:10–30, 1987.

[16] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[17] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 Users Manual. Technical Report ANL-95/11 - Revision 2.0.28, Argonne National Laboratory, 2000.

[18] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. `http://www.mcs.anl.gov/petsc/`, 2000.

[19] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, J. Donato, Jack J. Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994. Available at `http://www.netlib.org/templates/Templates.html`.

[20] Richard Barrett, Michael Berry, Jack J. Dongarra, Victor Eijkhout, and Charles Romine. Algorithmic bombardment for the iterative solution of linear systems: A poly-iterative approach. *Journal of Computational and Applied Mathematics*, 74(1-2):91–109, 1996.

[21] M. S. Bartlett. An inverse matrix adjustment arising in discriminant analysis. *Ann. Math. Statist.*, 22:107–111, 1951.

[22] D. M. Beazly and P. S. Lomdahl. Controlling the data glut in large-scale molecular-dynamics simulations. *Computers in Physics*, 11(3):230–238, May/June 1997.

[23] D. M. Beazly and P. S. Lomdahl. Feeding a large-scale physics application to python. In *Proceedings of the 6th International Python Conference*, October 1997.

[24] J. Bilmes et al. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, 1997. ACM SIGARC.

[25] Suzan L. Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[26] E. Bodewig. *Matrix Calculus*. North Holland, Amsterdam, 1959.

[27] K. S. Booth and G. S. Leuker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.*, 13:335–379, 1976.

[28] D. Braess. *Finite Elements*. Cambridge University Press, 1997.

[29] Tony F. Chan and Michael K. Ng. Galerkin projection methods for solving multiple linear systems. Technical Report CAM Rep. 96-31, Department of Math., UCLA, September 1996.

[30] J. Choi. A proposal for a set of parallel basic linear algebra subprograms. Technical report, University of Tennessee Knoxville, 1995. LAPACK Working Note 100.

[31] J. Choi, Jack J. Dongarra, Susan L. Ostrouchov, Antoine Petitet, David W. Walker, and Clint R. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996. (also LAPACK Working Note #80).

[32] Don Coppersmith and Samuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9:251–280, 1990.

[33] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th ACM National Conference*, pages 157–172, Association of Computing Machinery, New York, 1969.

[34] A. Dave and Iain S. Duff. Sparse matrix calculations on the Cray-2. *Parallel Comput.*, 5:55–64, 1987.

[35] Timothy A. 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. (The software is freely available at `http://www.cise.ufl.edu/research/sparse/umfpack/`).

[36] Timothy A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 94(42), 16 October 1994. See `http://www.cise.ufl.edu/~davis/sparse/`.

[37] Timothy A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 96(28), 23 July 1996. See `http://www.cise.ufl.edu/~davis/sparse/`.

[38] Timothy A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), 7 June 1997. See `http://www.cise.ufl.edu/~davis/sparse/`.

[39] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse lu factorization. Technical Report RAL-93-036, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, 1994.

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

[41] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[42] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W.-H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Applic.*, 20(3):720–755, 1999.

[43] James W. Demmel, Stanley C. Eisenstat, Jonh R. Gilbert, Xiaoye S. Li, and Joseph W.-H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, University of California Berkeley, Berkeley, California, 1995.

[44] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, 1989. (An updated version of this report can be found at `http://www.netlib.org/benchmark/performance.ps`).

[45] Jack J. Dongarra. Freely available software for linear algebra on the web, July 2001. (An updated version of this document is available at `http://www.netlib.org/utk/people/JackDongarra/la-sw.html`).

[46] Jack J. Dongarra, J. Bunch, Cleve Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[47] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990.

[48] Jack J. Dongarra, J. Du Croz, Iain S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:18–28, March 1990.

[49] Jack J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, March 1988.

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

[51] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[52] Jack J. Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. Technical report, Innovative Computing Laboratory, University of Tennessee, August 2002. To appear in the International Journal of High Performance Computing Applications in 2003. Available at `http://icl.cs.utk.edu/iclprojects/pages/sans.html`.

[53] Jack J. Dongarra, Victor Eijkhout, and Piotr Luszczek. Recursive approach in sparse matrix LU factorization. In *Proceedings of the 1st SGI Users Conference*, pages 409–418, Cracow, Poland, October 2000. ACC Cyfronet UMM.

[54] Jack J. Dongarra, Victor Eijkhout, and Piotr Luszczek. Recursive approach in sparse matrix LU factorization. *Scientific Programming*, 9(1):51–60, 2001.

[55] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[56] Jack J. Dongarra and Clint R. Whaley. Automatically tuned linear algebra software (ATLAS). In *Proceedings of SC'98 Conference*. IEEE, 1998.

[57] Iain S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. Technical Report RAL-TR-96-047, Rutherford Appleton Laboratory, Oxon, U.K., April 1996. Also available as Report TR-PA-96-22, CERFACS, Toulouse, France.

[58] Iain S. Duff, I. M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

[59] Iain S. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1-14), 1989.

[60] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Applic.*, 20:889–901, 1999.

[61] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, September 1983.

[62] Iain S. Duff and John K. Reid. MA48, a Fortran code for direct solution of unsymmetric linear systems of equations. Technical Report RAL-93-072, Rutherford Appleton Laboratory, Oxon, UK, 1993.

[63] Victor Eijkhout. Computational variants of the CGS and BiCGSTAB methods. Technical Report CS-94-241, Computer Science Department, The University of Tennessee Knoxville, August 1994. (Also LAPACK Working Note No. 78).

[64] Stanley C. Eisenstat, John R. Gilbert, and Joseph W.-H. Liu. A supernodal approach to a sparse partial pivoting code. In *Housholder Symposium XII*, 1993.

[65] Stanley C. Eisenstat and Joseph W.-H. Liu. Exploiting structural symmetry in unsymmetric sparse symbolic factorization. *SIAM J. Matrix Anal. Applic.*, 13(1):202–211, January 1992.

[66] Stanley C. Eisenstat and Joseph W.-H. Liu. Exploiting structural symmetry in sparse partial pivoting code. *SIAM J. Scientific and Statistical Computing*, 14:253–257, 1993.

[67] 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):605–624, 2000.

[68] Michael R. Fellows and Michael A. Langston. Nonconstructive advances in polynomial time complexity. *Info. Proc. Letters*, 26:157–162, 1987.

[69] Michael R. Fellows and Michael A. Langston. Nonconstructive tools for proving polynomial-time decidability. *J. of the ACM*, 35:727–739, 1988.

[70] Michael R. Fellows and Michael A. Langston. Fast search algorithms for layout permutation problems. *International Journal of Computer Aided VLSI Design*, 3:325–342, 1991.

[71] Michael R. Fellows and Michael A. Langston. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM J. Disc. Math.*, 5:117–126, 1992.

[72] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, 1999.

[73] G. A. Geist. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[74] G. A. Geist and Michael T. Heath. Parallel Cholesky factorization on a hypercube multiprocessor. Technical Report ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.

[75] G. A. Geist and Michael T. Heath. Matrix factorization on a hypercube multiprocessor. In Michael T. Heath, editor, *Hypercube Multiprocessors*, pages 161–180. SIAM, Philadelphia, PA, 1986.

[76] J. Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10:345–363, 1973.

[77] J. Alan George, Michael T. Heath, Joseph W.-H. Liu, and Esmond G.-Y. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Scientific and Statistical Computing*, 9(2):327–340, March 1988.

[78] J. Alan George, Micheal T. Heath, and Joseph W.-H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and Its Applications*, 77:165–187, 1986.

[79] J. Alan George and Joseph W.-H. Liu. A fast implementation of the minimum degree algorhthm using quotient graphs. *ACMTOMS*, 6:337–358, September 1980.

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

[81] J. Alan George and Joseph W.-H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[82] J. Alan George, Joseph W.-H. Liu, and Esmond G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[83] Roman Geus. *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 2002.

[84] 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.

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

[86] John R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Scientific and Statistical Computing*, 9:862–874, 1988.

[87] John R. Gilbert and Robert Schreiber. Highly parallel sparse Cholesky factorization. *SIAM J. Scientific and Statistical Computing*, 13(5):1151–1172, September 1992.

[88] Stefan Goedecker and Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, Philadelphia, PA, 2001.

[89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, third edition, 1996.

[90] Anoop Gupta and Edward Rothberg. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '93 Proceedings*, 1993.

[91] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM J. Matrix Anal. Applic.*, 24(2):529–552, 2002.

[92] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report TR-94-63, Department of Computer Science, University of Minnesota, 1994.

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

[94] Fred Gustavson and I. Jonsson. Minimal-storage high-performance cholesky factorization via blocking and recursion. *IBM Journal of Research and Development*, 44(6):823–850, November 2000.

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

[96] Michael T. Heath, Esmond G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, September 1991. Also appears in K. A. Gallivan et al., *Parallel Algorithms for Matrix Computations*, SIAM, Philadelphia, PA, 1990.

[97] M. R. Hestens and E. L. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards, Section B*, 49:409–436, 1952.

[98] David Hilbert. Über stetige abbildung einer linie auf ein flächenstük. *Mathematische Annalen*, 38:459–460, 1891.

[99] K. Hinsen. High level scientific programming with Python. In Peter M. A. Sloot, C. J. K. Tan, Jack J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002*, pages 691–700. Springer, 2002. Lecture Notes in Computer Science 2331.

[100] Laurie Hulbert and Earl Zmijewski. Limiting communication in parallel sparse Cholesky factorization. *SIAM J. Scientific and Statistical Computing*, 12(5):1184–1197, September 1991.

[101] E.-J. Im. *Automatic optimization of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, California, 2000.

[102] E.-J. Im and Kathy Yelick. Optimizing sparse matrix-vector multiplication on SMPs. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[103] International Organization for Standardization. Information technology – Programming languages – Fortran. ISO/IEC 1539-1:1997, Geneva, Switzerland, 1997.

[104] International Organization for Standardization. Programming languages – C. ISO/IEC 9899:1999, Geneva, Switzerland, 1999.

[105] B. Irons. A frontal solution program for finite element analysis. *Int. J. Numer. Methods in Engrg.*, 2:5–32, 1970.

[106] Dror Irony and Sivan Toledo. Communication-efficient parallel dense LU using 3-dimensional approach. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 2001.

[107] Dror Irony and Sivan Toledo. Trading replication for communication in parallel distributed-memory dense solvers. *Paralle Processing Letters*, 12:79–94, 2002.

[108] E. F. Kaasschieter. A practical termination criterion for the Conjugate Gradient method. *BIT*, 28:308–322, 1988.

[109] W. Kaufmann and L. Smarr. *Supercomputing and the Transformation of Science*. Scientific American Library, 1993.

[110] Brian Kernighan and Dennis Ritchie. *The C programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Second edition, 1988.

[111] Michael A. Langston. WQO-based methods. In *International Workshop on Combinatorial Methods for Curcuit Design*, Schloss Dagstuhl, Germany, October 1993.

[112] Michael A. Langston and Barbara C. Plaut. On algorithmic applications of the immersion order. *Discrete Mathematics*, 182:191–196, 1998.

[113] Xiaoye S. Li. *Sparse Gaussian Elimination on High Performance Computers*. Computer Science Department, University of California at Berkeley, 1996. Ph.D. thesis (SuperLU software available at `http://www.nersc.gov/~xiaoye/SuperLU/`).

[114] Xiaoye S. Li and James W. Demmel. Making sparse gaussian elimination scalable by static pivoting. In *Proceedings of SC'98 Conference*, Orlando, Florida, 7-13 November 1998.

[115] Xiaoye S. Li and James W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 22-24 March 1999.

[116] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16:346–358, 1979.

[117] Joseph W.-H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[118] Joseph W.-H. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11:73–91, 1989.

[119] Joseph W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Applic.*, 11(1):134–172, 1990.

[120] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, 1957.

[121] Mathematical Committee on Physical and Engineering Sciences. *Grand Challenges: High Performance Computing and Communications*. NSF/CISE, 1800 G Street NW, Washington, DC, 20550, 1991.

[122] Hans W. Meuer, Erik Strohmaier, Jack J. Dongarra, and Horst D. Simon. *Top500 Supercomputer Sites*, 20th edition edition, November 2002. (The report can be downloaded from `http://www.netlib.org/benchmark/top500.html`).

[123] D. Mirkovic and S. L. Johnsson. Automatic performance tuning in the UHFFT library. In *2001 International Conference on Computational Science*, San Francisco, California, USA, 2001.

[124] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 19 April 1965.

[125] Esmond G.-Y. Ng and Barry W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 14:761–769, 1993.

[126] Office of Science and Technology Policy, editors. *A Research and Development Strategy for High Performance Computing*. Executive Office of the President, 1987.

[127] Office of Science and Technology Policy, editors. *The Federal High Performance Computing Program*. Executive Office of the President, 1989.

[128] Dianne P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and Its Applications*, 29:293–322, 1980.

[129] J. M. Ortega and W. C. Rheinboldt. *Iterative solution of nonlinear equations in several variables*. McGraw-Hill, New York, 1970.

[130] Giuseppe Peano. Sur une courbe qui remplit toute une aire plaine. *Mathematische Annalen*, 36:157–160, 1890.

[131] Antoine Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions*. Computer Science Department, University of Tennessee, Knoxville, December 1996. PhD dissertation.

[132] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceddings of SC'99*, 1999.

[133] Alex Pothen and Chunguang Sun. Distributed multifrontal factorization using clique trees. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 34–40, 1991.

[134] Python Software Foundation. Python programming language. For details refer to `http://www.python.org/`.

[135] Padma Raghavan. *Distributed sparse matrix factorization: QR and Cholesky factorizations*. PhD thesis, Pennsylvania State University, University Park, PA, 1991.

[136] Padma Raghavan and Alex Pothen. Parallel orthogonal factorization. In *SIAM Symposium on Sparse Matrices*, Gleneden Beach, OR, 1989. SIAM, Philadelphia, PA.

[137] Guido van Rossum and Fred L. Drake. *Extending and Embedding the Python Interpreter*. PythonLabs, April 2002.

[138] Guido van Rossum and Fred L. Drake. *Python Library Reference*. PythonLabs, April 2002.

[139] Guido van Rossum and Fred L. Drake. *Python Tutorial*. PythonLabs, April 2002.

[140] Guido van Rossum and Fred L. Drake. *Python/C API Reference Manual*. PythonLabs, April 2002.

[141] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon systems. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[142] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '92 Proceedings*, 1992.

[143] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 15(6):1413–1439, November 1994.

[144] Edward Rothberg and Robert Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing '94 Proceedings*, 1994.

[145] Youcef Saad. Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra and Its Applications*, 77:315–340, 1986.

[146] Youcef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, New York, 1996.

[147] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[148] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *Ann. Math. Statist.*, 20:621, 1949.

[149] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Ann. Math. Statist.*, 21:124–127, 1950.

[150] Deszo Sima, Terence Fountain, and Péter Kacsuk. *Advanced computer architectures: a desing space approach*. Addison-Wesley, Reading, MA, 1997.

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

[152] Gilbert W. Stewart. *Introduction to matrix computations*. Academic Press, New York, 1973.

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

[154] V. Strassen. Gaussian elimination is not optimal. *Numerical Mathematics*, 13:354–356, 1969.

[155] W. Tinney and J. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. In *Proceedings of the IEEE*, volume 55, pages 1801–1809, 1967.

[156] Sivan Toledo. Improving the memory-system performance of sparse matrix-vector multiplication. *IBM Journal of Research and Development*, 41(6), November 1997.

[157] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, October 1997.

[158] Ilkka Tuomi. The lives and death of Moore's law. *First Monday*, 7(11), 4 November 2002. (Available at `http://firstmonday.org/`).

[159] Bill Venners. Programming at Python speed: A conversation with Guido van Rossum, 27 January 2003. Available at `http://www.artima.com/intv/speed.html`.

[160] Sesh Venugopal and Vijay K. Naik. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. In *Supercomputing '91 Proceedings*, pages 866–875, 1991.

[161] Henk van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 13:631–644, 1992.

[162] R. Clint Whaley. *A User's Guide to the BLACS v1.1.* Computer Science Departement, University of Tennessee Knoxville, 5 May 1997. LAPACK Working Note 94.

[163] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[164] Max A. Woodbury. Inverting modified matrices. Technical Report Mem. Rep., No. 42, Statistical Research Group, Princeton University, Princeton, N. J., 1950.

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

Appendix

Definition of Terms

Acronyms

AMD Approximate Minimum Degree (ordering)

API Application Programming Interface

ATLAS Automatically Tuned Linear Algebra Software [56, 163]

CISC Complex Instruction Set Computer

CG Conjugate Gradient

CPU Central Processing Unit

BLAS Basic Linear Algebra Subroutines [47, 48, 50, 49] Routines referenced in the text:

- `xSCAL` – multiply a vector by a scalar,
- `IxAMAX` – find an element of a vector having the largest absolute value,
- `xGEMV` – matrix-vector multiplication
- `xTRSM` – triangular solve for multiple right-hand sides,
- `xGEMM` – matrix-matrix multiplication

Symbol `x` in each name specifies the floating point precision of matrix/vector it may be either `S`, `D`, `C`, or `Z` which corresponds to single and double precision real and single and double precision complex numbers.

flop floating point operation (also flop/s for number of floating point operations per second)

FP floating point (arithmetic)

LAPACK Linear Algebra PACKage [11]. Routines referenced in the text:

- `xGETRF` – performs LU factorization with partial row pivoting based on Level 3 BLAS routines
- `xLASWP` – performs row pivoting

Symbol x has the same meaning as in BLAS.

FFT Fast Fourier Transform

IBM International Business Machines

KiB kibibyte[1] (kilobinary): 2^{10} (1024) bytes

MiB mebibyte (megabinary): 2^{20} (1048576) bytes

MMD Multiple Minimum Degree (ordering)

MUMPS MUltifrontal Massively Parallel sparse direct Solver

NP Non-deterministic Polynomial (algorithm)

OS Operating System

RISC Reduced Instruction Set Computer, also Superscalar RISC [150] – a RISC processor with multiple functional units

SANS Self Adapting Numerical Software

SMP Symmetric Multi-Processor

UMFPACK Unsymmetric Multifrontal PACKage

WSMP Watson Sparse Matrix Package

VPU Vector Processing Unit

Mathematical Symbols

$\mathbb{R}$	set of real numbers
A, L, U	real matrices: $A, L, U \in \mathbb{R}^{m\times n}$, $A = [a_{ij}]$ $1 \le i \le m, 1 \le j \le n$
$\|\cdot\|_p$	matrix or vector norm:
	$\|x\|_p = (\sum_{i=1}^{n} \|x_i\|^p)^{1/p}$
	$\|A\|_p = \sup_{x\neq 0} \|Ax\|_p / \|x\|_p$
	$\|A\|_F = (\sum_{i=1}^{m} \sum_{j=1}^{n} \|a_{ij}\|^2)^{1/2}$
	where: $A \in \mathbb{R}^{m\times n}$; $x \in \mathbb{R}^n$ $m, n \in \{1, 2, \ldots\}; 1 \le p \le +\infty$
$\|\cdot\|_A$	A-norm of a vector (A must be positive definite):
	$\|x\|_A = \sqrt{x^T A x}$ $A \in \mathbb{R}^{n\times n}; x \in \mathbb{R}^n$
$\kappa_p(A)$	p-norm matrix condition number: $\kappa_p(A) = \|A\|_p \, \|A^{-1}\|_p$
$\mathcal{U}(A)$	upper triangular part of the matrix including the diagonal:

[1] See `http://physics.nist.gov/cuu/Units/binary.html`.

	$\mathcal{U}(A) = [u_{ij}] \quad u_{ij} = \begin{cases} a_{ij} & \text{if } i \leq j \\ 0 & \text{if } i > j \end{cases}$
$\mathcal{L}_1(A)$	lower triangular part of the matrix with unitary diagonal: $\mathcal{L}_1(A) = [l_{ij}] \quad l_{ij} = \begin{cases} 0 & \text{if } i < j \\ 1 & \text{if } i = j \\ a_{ij} & \text{if } i > j \end{cases}$
$\mathbb{P}^n$	set of n by n permutation matrices, i.e. identity matrices with rearranged rows
$\eta(A)$	number of (structural) nonzero entries in matrix A
$n(A)$	dimension of matrix A
m_B, n_B	blocking factors for the first and second matrix dimensions
$\text{float}(x)$	floating point representation of x
ε	machine floating point precision: $\varepsilon = \max_{e>0} \text{float}(1+e) = 1$
$\hat{x}$	approximation of x
$\xi(\hat{x}, x)$	relative forward error: $\xi(\hat{x}, x) = (x - \hat{x})/x \equiv \texttt{FERR}(\hat{x}, x)$
$r(\hat{x})$	scaled residual (backward error) of $Ax = b$: $r(\hat{x}) = (\lVert A\hat{x} - b \rVert)/(\lVert A \rVert \lVert \hat{x} \rVert n\varepsilon)$
$\log x$	logarithm of base 2: $\log x \equiv \log_2 x$ (unless stated otherwise)
$\min\{\ldots\}$	smallest value
$O(f(n))$	functions of order $f(n)$: $O(f(n)) = \{g \mid \exists_{a,b,N} \forall_{n>N}\, af(n) \leq g(n) \leq bf(n)\}$

Vita

Piotr Luszczek was born in Kraków, Poland on October 19, 1974. He graduated from high school in 1989 with the title of Electronics Technician, and received a Master of Science degree in Computer Science in 1999 from the University of Mining and Metallurgy in Kraków, Poland under supervision of Dr. Marian Bubak. His thesis was about parallel runtime support for distributed-memory irregular applications and was a part of joint research effort with the University of Vienna, Austria.

Immediately after completion of the Master's degree he came to the University of Tennessee Knoxville to pursue the PhD degree under supervision of Prof. Jack Dongarra. While enrolled at the University of Tennessee, he worked on research projects in cooperation with the Hewlett-Packard Company, the Joint Institute for Computational Science and Oak Ridge National Laboratory.