THESE

pour obtenir

LE TITRE DE DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité: Informatique et Télécommunications
par

Christof VÖMEL
CERFACS

Contributions à la recherche en calcul scientifique haute performance pour les matrices creuses

Contributions to research in high performance scientific computing for sparse matrices

---

Thèse présentée le 20 mars 2003 devant le jury composé de:

P. R. Amestoy	Maître de conférence, ENSEEIHT	*Directeur de thèse*
M. Arioli	Directeur de recherche, RAL	
I. S. Duff	Directeur de recherche, CERFACS et RAL	
E. Ng	Directeur de recherche, NERSC, LBNL	*Rapporteur*
A. Pothen	Professeur, ODU	
J. Roman	Professeur, ENSEIRB	*Rapporteur, Président*

Résumé

Dans cette thèse, nous présentons le résultat de nos recherches dans le domaine du calcul en algèbre linéaire creuse. En particulier, nous nous intéressons au développement d'un nouvel algorithme pour estimer la norme d'une matrice de manière incrémentielle, à l'implantation d'un modèle de référence des 'Basic Linear Algebra Subprograms for sparse matrices (Sparse BLAS)', et à la réalisation d'un nouveau gestionnaire de tâches pour un solver multifrontal creux pour architectures à mémoire répartie.

Notre méthode pour estimer la norme d'une matrice a l'avantage de s'appliquer à tout type de matrices, dense et creux, contrairement aux algorithmes incrémentiels existants auparavant. Elle peut s'avérer utile par exemple dans le cadre des factorisations QR, Cholesky, ou LU.

En ce qui concerne le BLAS creux, notre modèle de référence en Fortran 95 est actuellement la seule implantation existante des interfaces spécifiées par le standard. Afin de laisser assez de liberté à l'implantation la plus efficace, le standard définit des interfaces génériques et reste général quant à la structure des données. Nous avons donc été amenés à répondre aux questions complexes concernant la représentation et la gestion des données.

Le séquencement de tâches devient un enjeu important dès que nous travaillons sur un grand nombre de processeurs (entre 100 et 500). L'algorithme introduit dans cette thèse permet d'améliorer le passage à l'échelle du solveur d'une façon significative. Une étude des gains en mémoire et en temps de calcul obtenus sur une architecture possédant plus de 500 processeurs montre tout l'intérêt de cette nouvelle approche.

Mots clés: Algorithmique numérique, calcul à mémoire répartie, systèmes linéaires creux, élimination de Gauss multifrontal, séquencement dynamique de tâches, structures de données creuses, standard de programmation, BLAS creux, norme d'une matrice, conditionnement, estimation incrémentielle.

Abstract

In this thesis, we present our research on methods in high performance scientific computing for sparse matrices. Specifically, we are concerned with the development of a new algorithm for incremental norm estimation, the reference model implementation of the standard for Basic Linear Algebra Subprograms for sparse matrices (Sparse BLAS), and the design of a new task scheduler for MUMPS, an asynchronous distributed memory multifrontal direct solver for sparse linear systems.

Our new incremental norm estimator has the advantage of being applicable to both dense and sparse systems, in contrast to previously existing incremental schemes. Applications include for example monitoring of a QR, a Cholesky, or an LU factorization.

Our Fortran 95 reference model is currently the only existing implementation of the interfaces in the standard for the Sparse BLAS. We identify many of the complicated issues regarding the representation and the handling of sparse data structures that underlie the high-level description of sparse vector and matrix operations in the standard but are avoided there in order to leave enough freedom for vendors to provide efficient implementations.

With our new task scheduler for MUMPS, we address concerns about lack of scalability and performance on large numbers of processors that arose in a comparison with the SuperLU parallel solver. In the new approach, we determine, during the analysis of the matrix, candidate processes for the tasks that will be dynamically scheduled during the subsequent factorization. This approach significantly improves the scalability of the solver in terms of time and memory needed, as we show by comparison with the previous version.

Keywords: High performance computing, sparse linear systems, MUMPS, multifrontal Gaussian elimination, distributed memory code, dynamic task scheduling, sparse data structures, programming standard, Sparse BLAS, reference implementation, matrix norm, condition number, incremental estimators.

Acknowledgements

I am grateful to I. Duff for allowing me to write my thesis at CERFACS, for letting me find my way, and for supporting my ideas and projects. I also thank P. Amestoy for his continuing interest in my work, his constructive criticism, and his guidance. I recognize that without their help, this thesis would not have been possible.

I wish to thank Mr J. Roman and Mr E. Ng for their interest in my work and acting as referees on my thesis commitee. Likewise, I acknowledge with gratitude Mr M. Arioli and Mr A. Pothen as members of my jury.

It is my great pleasure to thank all members of the Parallel Algorithms Group who, explicitly or implicitly through their friendship, in many ways contributed to this work. Merci, Brigitte, Bruno, Daniel, Dominique, Elisabeth, Emeric, Enric, Hamid, Jean-Christophe, Julien, Luc, Marielba, Martin, Serge, Songklod, Stéphane, Tadas.

Contents

Introduction

The solution of large linear systems of equations is a central part of many scientific calculations. Whatever physical phenomenon might be modelled, at the end of the discretization process usually a linear system must be solved for unknowns that represent the physical quantities of the underlying problem. We are particularly interested in the class of sparse matrices which arise from diverse fields such as finite element models in mechanics, finite volume discretizations in fluid dynamics, or circuit theory in electrical engineering.

We call a matrix sparse if it is advantageous, for example with respect to work and storage, to exploit the zero entries. By focusing only on the nonzero part of the matrix entries, sparse matrix algorithms can considerably reduce computations and memory space, and thus be much more efficient than their dense counterparts. However, the amount of work required for developing the sparse version of a given algorithm can be important and involve complicated issues like the efficient handling of sparse data structures.

Because of its prominent role and the complexity of the related issues, research on the efficient solution of sparse linear systems is being given a great deal of attention and is going through an exciting evolution that we describe in the following.

In the first part of this introductory chapter, we give an overview of the area of high performance scientific computing for the direct solution of sparse linear systems. In particular, we identify those issues that have inspired our own research. Then, in the second part, we present the results that we have obtained in the framework of this thesis.

Current topics of research in high performance scientific computing for the direct solution of sparse linear systems

As an introduction to the results of our research, we describe in this section recent work on the efficient solution of sparse linear systems. However, we are mainly concerned with *direct* methods for *unstructured* sparse matrices, that is matrices without a special form. These methods are often preferable to iterative solvers when trying to solve an ill-conditioned system. Furthermore, they are commonly exploited to construct preconditioners when because of limited storage, an iterative approach must be used [46]. The iterative solution of sparse linear systems and the calculation of eigenvalues are complicated issues in their own right, and we refer the reader to the recent surveys [69] and [121]. Furthermore, direct techniques for structured sparse matrices, for example in tridiagonal or general banded form, are reviewed in [47, 56].

The evolution of computer architectures and programming paradigms has greatly influenced research and developments in sparse matrix computations. To make efficient use of modern machines with a multi-layered memory hierarchy, algorithm designers now try to increase the ratio of computational operations to memory access [41, 56]. In order to facilitate this task, but also to provide a level of abstraction for programming and to allow a greater robustness and program portability, the Basic Linear Algebra Subprograms (BLAS) were developed [39, 40, 100]. This initial set of standardized kernels for operations with dense matrices was adopted by vendors who provided optimised implementations, offering to users a high performance API for their programs. Thanks to this support, the gains through using BLAS kernels on modern system architectures are so high that we usually cannot afford not to use them. Another illustration of their huge impact on software development is given by the fact that standard numerical libraries like the Linear Algebra PACKage (LAPACK [13]) make heavy use of the BLAS kernels [42]. With this success of the BLAS, other efforts followed: first, the development of the Basic Linear Algebra Communication Routines (BLACS [43]) used within the framework of the ScaLAPACK [29] project, then the Automatically Tuned Linear Algebra Software (ATLAS [44]), and finally the design of a new, extended BLAS standard [1, 24] including kernels for mixed precision calculations [103] and sparse matrices [50, 57].

The performance of numerical algorithms on modern computer architectures is often limited by the speed of data movement between different layers of the memory hierarchy rather than by the performance of the floating point processing units of the processor executing the numerical operations of the algorithm itself [41]. In consequence, a key strategy of the BLAS is to partition the data into blocks which are subsequently loaded into the cache or the local memory with the goal of maximizing reuse and increasing the ratio of floating point to memory access operations. In this respect, an important application of the BLAS in the framework of sparse matrices is the class of blocked algorithms such as supernodal [16, 54] and multifrontal methods [54, 55]. There, one obtains a significant gain in performance through using the BLAS, that is dense matrix kernels, for the frontal matrices. One example of such a code making use of the Level 3 BLAS is MA41 [7, 12] from HSL [91]. While MA41 was designed for shared memory computers, the development of distributed memory machines and the message passing paradigm for communication in the 90's required the design of new multifrontal codes and led, for example, to the recent development of MUMPS [9, 10, 11] and WSMP [72, 73]. Furthermore, alternative distributed memory codes, based for example on supernodal techniques, were developed, including PaStiX [79, 80, 81], SPOOLES [15], and SuperLU [35, 36]. For a complete review on the history of these different techniques, we refer to [41, 56, 76]. For any of these direct solvers, a crucial issue when working with sparse matrices is the ordering of the rows and columns so that the factorization preserves sparsity and/or less work needs to be performed. Furthermore, in a parallel environment, the ordering must offer parallelism for the solver. Classic orderings based on a minimum degree heuristic [126] or the nested dissection approach [65] have been refined and have led to the development of modern algorithms including approximate minimum degree (AMD [6]) and approximate minimum fill (AMF [109, 119]) orderings as well as hybrid techniques [17, 78, 123] which are implemented in ordering packages such as CHACO [77], METIS [98], and SCOTCH [112, 113]. Moreover, in order to facilitate pivoting or to avoid it entirely, one often tries to permute large matrix entries to the diagonal [51, 52]. While the stability of an LU factorization with diagonal pivoting cannot be guaranteed even with this permutation, it can still mean that the number of off-diagonal pivots is substantially reduced in solvers like MA41, MUMPS, and WSMP. In the SuperLU distributed memory solver, static pivoting is used instead of threshold-based partial pivoting, that is off-diagonal pivots are entirely avoided. Small or zero diagonal entries are perturbed when encountered and the computation of a solution proceeds via iterative refinement [47, 85, 86]. Thus, the permutation of large entries to the diagonal is also very beneficial in this context [102].

In the special case of symmetric positive definite systems, the Cholesky factorization is preferable to the LU factorization because it is backward stable [86] and needs only half of the number of operations. Furthermore, since no pivoting is required, the sparse Cholesky factorization can avoid the use of dynamic data structures and work instead with static memory, as do for example PSPASES [74] and PaStiX [79, 80, 81].

Also from the point of view of scheduling, the sparse Cholesky factorization is very attractive. Due to the absence of pivoting, the elimination process depends only on the matrix structure and can be simulated efficiently without taking account of the values of the matrix entries. In particular, in a parallel environment, it is possible to calculate a static schedule prior to factorization that balances the work among the parallel processes based on a careful estimation of the speed of communication and of the computational operations of the underlying computer architecture, as does for example PaStiX [79, 80, 81]. Dynamic load balancing during the actual factorization as used by MUMPS [9, 10, 11] is generally less important in this case. However, the scheduling problems arising in the context of sparse multifrontal factorization, as far as they concern the minimisation of the overall completion time for the jobs to be scheduled, are NP hard in general [62]. Consequently, one usually calculates only an approximate solution based on relatively inexpensive heuristics or so-called approximating algorithms in order not to increase the overall complexity of the linear solver [89, 101].

Once we have solved a linear system, another important step consists of determining the quality of the numerical solution that has been computed in finite precision. As already mentioned in the discussion of SuperLU, iterative refinement can be used to improve the accuracy of the computed solution. While backward error bounds usually are computed from a residual [86], the computation of forward error bounds requires knowledge of the matrix condition number. As the computation of singular values is costly, these are commonly not computed exactly but obtained via relatively inexpensive so-called condition estimators. These are included in standard numerical libraries such as LINPACK [30] or LAPACK [75, 88] for dense matrices, for a survey we refer also to [83, 86]. Furthermore, an alternative, so-called incremental, approach was designed specifically to monitor the conditioning of a triangular factorization on the fly [22]. However, the application of this incremental technique to sparse matrices is difficult and requires additional care, see [23, 59]. A related question concerns the determination of the numerical null space of a matrix by a direct factorization and led to the development of the class of so-called rank-revealing methods, see for example [26, 27, 28, 110]. Initially, these were designed for dense matrices but have recently been adopted also for sparse solvers, see [114]. In all these methods, a condition estimator plays a central role for determining the rank-revealing factor.

A description of our research and the contents of this manuscript

In the first part of this introduction, we have given an overview of recent research on the direct solution of sparse linear systems. We now give an outline of the subsequent chapters of this thesis. Specifically, we describe our contributions to the different topics of research that we have identified before. Our presentation consists of three parts.

- The first part of this thesis is concerned with the development of a new incremental algorithm for norm estimation. Offering more flexibility than previously existing schemes, our new algorithm not only adapts well to dense but also to sparse matrices. We give both a theoretical analysis of the properties of the new technique and test its performance in practice. Through our investigation, we demonstrate the reliability of the new algorithm which is related to condition estimation and has applications in rank-revealing algorithms.

- In the second part of this thesis we describe the design and the implementation of the Sparse BLAS kernels, providing Basic Linear Algebra Subprograms for sparse matrices. These are defined by the BLAS Technical Forum with the principal goal of aiding in the development of modern iterative solvers like Krylov subspace methods for large sparse linear systems. The Sparse BLAS standard, as part of the BLAS project, specifies interfaces for a high-level description of vector and matrix operations for the algorithm developer but also leaves enough freedom for vendors to provide the most efficient implementation of the underlying algorithms for their specific architectures. Our Fortran 95 reference model implementation of the Sparse BLAS is currently the only existing complete implementation. It represents a first step towards widely available efficient kernels, as we identify many of the underlying complicated issues of the representation and the handling of sparse matrices and give suggestions to other implementors of how to address them.

- The development of a new scheduling algorithm for MUMPS, a MUltifrontal

Massively Parallel Solver, is presented in the third part of this thesis. Our new approach addresses concerns about the scalability of MUMPS on a large number of processors with respect to computation time and use of memory that arose in a comparison with SuperLU, another state-of-the-art parallel direct solver for linear systems. An investigation shows that the dynamic scheduling has a significant impact on the performance of MUMPS as a major part of the computational work is distributed dynamically. We develop new algorithms to improve the treatment of the assembly tree during the static analysis and to better guide the dynamic task scheduler during factorization. With this new approach, we can increase the scalability of the solver in terms of time and memory needed.

I

Chapter 1

Incremental Norm Estimation for Dense and Sparse Matrices

Error analysis is a very important field in numerical computation. In this field, we try to measure the quality of a solution that has been computed in finite precision. According to [47], we call a problem *ill-conditioned* if small changes in the data can produce large changes in the solution. In the special case of the solution of linear systems $Ax = b$, we assign a *condition number*

$$\kappa(A) = \|A\| \, \|A^{-1}\| \tag{1.1}$$

to the matrix A and speak of the matrix being ill-conditioned if $\kappa(A) \gg 1$.

Standard bounds on the forward error in the numerical solution of a linear system are usually expressed in terms of the matrix condition number [83, 86]. Furthermore, it can be shown that the common approach of improving the numerical solution of a linear system through fixed precision iterative refinement is guaranteed to yield a relative forward error bounded by the unit roundoff times the condition number, as long as the matrix A is not too ill-conditioned and the solver is not too unstable. For a precise presentation of this result, we refer to [86].

However, we want to know in general only the magnitude of the error and not the precise value. For this reason, we often accept estimates that are correct up to a certain factor but are significantly cheaper to compute than the solution itself. For dense linear systems, we are interested in algorithms that compute an estimate with about $\mathcal{O}(n^2)$ operations and that is correct to within a factor 10 [83, 86]. One example of such an algorithm is the matrix 1-norm power method that is available in LAPACK [13].

In the direct solution of linear systems where we compute a triangular factorization of the initial matrix A, condition estimators for triangular matrices are of particular interest. According to [86], the first condition estimator of this kind that was widely used is the one included in LINPACK [30]. Several other estimators were developed later on; for a survey on these methods we refer to [83, 86]. Here, we are particularly interested in a method developed by Bischof in [22], the so called *incremental* condition estimation (ICE). This algorithm allows us to monitor an ongoing triangular factorization to detect ill-conditioning. However, while working well in the dense case, ICE is difficult to adapt to *sparse* matrices, an observation that motivated our research.

In this chapter of the thesis, we present a new incremental approach to 2-norm estimation. Our investigation covers both dense and sparse matrices which can arise for example from a QR, a Cholesky, or an LU factorization. If the explicit inverse of a triangular factor is available, as in the case of an implicit version of the LU factorization, we can relate our results to ICE. This will be explained later on. Incremental norm estimation (INE) extends directly from the dense to the sparse case without needing the modifications that are necessary for the sparse version of ICE. INE can be applied to complement ICE, since the product of the two estimates gives an estimate for the matrix condition number. Furthermore, when applied to matrix inverses, INE can be used as the basis of a rank-revealing factorization.

In order to avoid confusion of the reader, we point out that the term condition estimation frequently refers to estimating the norm of the matrix *inverse*. In the special case of the Euclidean norm, this corresponds to estimating the size of the smallest singular value of a matrix. The algorithm we present in the following is explicitly designed to estimate the largest singular value of a given matrix. Furthermore, in contrast to Bischof who implicitly works with the matrix inverse, our algorithm is based on the triangular matrix itself. For these reasons, and in order to underline the ingenuity of our approach, we refer to our scheme as incremental *norm* estimator. However, both Bischof's and our incremental algorithm can provide estimates of the largest and smallest singular values, hence both can be used as *condition number* estimators.

1.1 Introduction to incremental estimation

There are many cases when it is interesting and important to detect the ill-conditioning of a square matrix A from the triangular factors arising in its LU or QR factorization. Applications include the calculation of forward error bounds based on the condition number of A and robust pivot selection criteria.

Another particularly interesting field of applications is provided by rank-revealing factorizations. During the process of determining a rank-revealing permutation, several (and, in the extreme case, an exponential number of) leading or trailing submatrices have to be investigated for their conditioning, see for example the survey [28]. A condition estimator is used to determine the conditioning of these matrices. Conceptually, there are two major classes of these estimators. The estimators belonging to the first class are *static* in the sense that they estimate the condition number of a fixed triangular matrix. These methods are surveyed in [83]. The second class can be used for *dynamic* estimation when a triangular matrix is calculated one column or row at a time. These incremental schemes (often called incremental condition estimation or ICE) for estimating ill-conditioning in the Euclidean norm were originally presented in [22] and are particularly attractive for monitoring a factorization as it proceeds. This was exploited in [114] where a generalization of the original scheme to sparse matrices [23] was incorporated in a multifrontal QR algorithm to generate a good initial permutation for rank-revealing 'on-the-fly'. Finally, the idea of incremental updates was also applied in an algorithm for the calculation of the condition number in the Frobenius norm [125]. This algorithm computes the condition number rather than estimating it. However, it is comparatively expensive and has, as far as we know, not found widespread use.

A completely different rank-revealing strategy is proposed in [108]. Instead of using condition estimation together with the triangular factors from a LU factorization (as for example in [26, 92, 93, 110]), a method based on an *implicit* LU factorization is employed. This so-called Direct Projection Method [20] calculates an upper triangular matrix Z such that $AZ = L$ is a lower triangular matrix, with $Z = U^{-1}$ where U is the triangular factor of Crout's LU factorization. To our knowledge, this is the first time information on the *inverse* of a triangular factor was used to detect ill-conditioning. Speaking in terms of the Euclidean matrix norm, all the previous approaches only used the triangular factors themselves so that the condition estimators had to estimate the reciprocal of the smallest singular value. On the contrary, working with the matrix inverse implies the estimation of the largest singular value. This motivated us to think about the design of an efficient norm estimator which can be applied in that framework.

When we were reformulating the ICE scheme from [22] to the task of norm estimation, we discovered that this scheme has a major shortcoming. Namely that the scheme allows the use of approximate vectors for only one side; that is, approximate right singular vectors for lower triangular matrices and approximate left singular vectors for upper triangular matrices. While this might at first glance not seem very critical, it has a severe implication on the use of ICE on *sparse* matrices and can degrade the quality of the estimates drastically. This has been observed in [23] where sophisticated modifications have been introduced to adapt the scheme to the sparse case.

The topic of this chapter of the thesis is the development of an incremental 2-norm estimation (INE) scheme that can be based on approximate singular vectors of either the left or the right side. We observe from experiments on dense matrices that INE estimates the largest singular value with the same quality as ICE estimates the smallest one. The great advantage of the norm estimator is that it applies without modifications to the sparse case. The right incremental approach for sparse matrices will prove to be as reliable as it is for dense ones. An immediate application of our scheme lies in its combination with ICE in order to obtain an incremental estimator for the Euclidean condition number. Another interesting application arises from calculations that involve matrix inverses, for example the Direct Projection Method [20] and the rank-revealing approach of [108]. Here, INE can be used to estimate the smallest singular value.

In Section 1.2, we first briefly discuss the original condition estimation scheme of Bischof [22] and then describe how we calculate the matrix norm of a triangular matrix in incremental fashion. We show for which matrices the incremental approach leads to exact results and investigate its behaviour in the sparse case. Afterwards, we give an assessment of its work and storage demands.

Of course, our norm estimation is of particular interest when the inverse of the triangular factor is available. In Section 1.3, we consider, as an example, a QR factorization with inverted triangular factor. This algorithm will later be used for testing our norm estimator.

The inversion of sparse matrices is additionally associated with the problem of fill-in. However, in the case of triangular matrices fill-in can be avoided by storing the inverse in factored form as proposed by [4]. We describe the details of this approach in Section 1.4 and illustrate problems that can occur when we try to detect ill-conditioning from the factored form.

We show the reliability of our incremental norm estimator in Section 1.5, by presenting results obtained from a variety of dense and sparse test cases from standard matrix collections [48, 84].

Finally, we give our conclusions in Section 1.6.

1.2 Incremental estimators

In this section, we present the details of our incremental norm estimator. The principal conceptual difference between our scheme and the original incremental condition estimator (ICE) [22] is that ours uses matrix-vector multiplications whereas ICE is based on the solution of triangular systems. A more detailed comparison between the schemes is given in Section 1.2.6.

1.2.1 The original incremental condition estimator (ICE)

In order to appreciate the general difficulties of determining the conditioning by examining the triangular factors, we first present two classical test matrices from Kahan [97]:

Example 1.2.1 *Consider $T_n \in \mathbf{R}^{n\times n}$ where*

$$T_n = \begin{bmatrix} 1 & -\gamma & \dots & -\gamma \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\gamma \\ 0 & \dots & 0 & 1 \end{bmatrix}$$

with $\gamma > 0$. The components of the inverse $T_n^{-1} = (\alpha_{ij})$ satisfy the recursion $(\alpha_{i-1j}) = (1+\gamma)(\alpha_{ij}), i = j-2, \dots, 1$, hence it is given by

$$\alpha_{ij} = \begin{cases} 1, & i = j \\ \gamma(1+\gamma)^{j-i-1}, & i < j \\ 0, & i > j. \end{cases}$$

Example 1.2.2 *Consider* $K_n(c) \in \mathbf{R}^{n\times n}$ *with*

$$K_n(c) = \mathrm{diag}(1, s, s^2, \dots, s^{n-1}) \begin{bmatrix} 1 & -c & \dots & -c \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -c \\ 0 & \dots & 0 & 1 \end{bmatrix}$$

where $c, s \in (0,1)$ *with* $c^2 + s^2 = 1$. *Its inverse is given by* $K_n^{-1}(c) = (\alpha_{ij})$ *with*

$$\alpha_{ij} = \begin{cases} s^{1-i}, & i = j \\ s^{1-i}c(1+c)^{j-i-1}, & i < j \\ 0, & i > j. \end{cases}$$

Both matrices are very ill-conditioned which is easy to see from the entries of the inverses but is not evident from the entries of the matrices themselves. A Householder QR factorization with column pivoting [70] will reveal the ill-conditioning of Example 1.2.1 but will not work on Example 1.2.2. We will use both these matrices in our experiments in Section 1.5.

In [22], an elegant approach to condition estimation is presented which updates an estimate of the smallest singular value of a triangular matrix when it is augmented by adding another column. We now describe this approach applied to upper triangular matrices.

Given an upper triangular matrix $T \in \mathbf{R}^{n\times n}$, we can calculate its smallest singular value by finding a vector $d \in \mathbf{R}^n$ of unit length so that the solution x of $x^T T = d^T$ has maximum norm. That is, we find

$$d^* = \arg\max_{\|d\|_2=1} \|d^T T^{-1}\|_2.$$

Once we have solved this problem (at least approximately), it is shown in [22] how to compute a cheap estimate of the smallest singular value for the augmented matrix

$$\hat{T} = \begin{bmatrix} T & v \\ & \gamma \end{bmatrix}.$$

The right-hand side $\hat{d}$ for the augmented system $\hat{x}^T\hat{T} = \hat{d}^T$ can be chosen as

$$\hat{d} = \hat{d}(s,c) = \begin{pmatrix} sd \\ c \end{pmatrix}, \tag{1.2}$$

where $s^2 + c^2 = 1$, and the solution to this augmented system has the form

$$\hat{x} = \begin{pmatrix} sx \\ \frac{c - s\alpha}{\gamma} \end{pmatrix} \tag{1.3}$$

with $\alpha = x^T v$. In other words, $\hat{x}^T\hat{T} = \hat{d}^T$ can be solved for $\hat{x}$ *without any back-substitution involving* T.

The parameters $(s,c) \in \mathbf{R}^2$ are chosen to maximize the norm of $\hat{x}$. This maximization problem can be treated analytically, and we refer the reader to the very elegant demonstration in [22].

The low cost of this approach together with the quality of the estimates obtained have made it an attractive safeguard for the computation of the QR factorization, as was already suggested in [22] and later on was successfully employed in the sparse multifrontal rank revealing QR factorization [114].

In [60], it is shown that one step of Lanczos iteration for improving an incremental estimate can be applied in an incremental fashion at a cost of $\mathcal{O}(n)$, where n is the order of the matrix. This yields a better estimate while preserving the overall costs of order $\mathcal{O}(n)$ for an incremental step.

1.2.2 Incremental norm estimation by approximate left singular vectors

Analogously to Section 1.2.1, we seek a cheap incremental *norm* estimator when augmenting an upper triangular matrix. We can design an efficient scheme by proceeding in a very similar way to the ICE construction.

Computing the matrix norm using a left singular vector means we wish to find a vector y of unit length such that

$$y^* = \arg \max_{\|y\|_2=1} \|y^T T\|_2.$$

An incremental norm estimator has then to specify a cheap heuristic for the computation of $\hat{y}$ corresponding to the augmented matrix

$$\hat{T} = \begin{bmatrix} T & v \\ & \gamma \end{bmatrix}. \tag{1.4}$$

We will see that we can avoid a matrix-vector product in this computation if we restrict the search to vectors $\hat{y}$ of the form

$$\hat{y} = \hat{y}(s,c) = \begin{pmatrix} sy \\ c \end{pmatrix}, \tag{1.5}$$

where $s^2 + c^2 = 1$.

Since

$$\begin{aligned} \|\hat{y}^T \hat{T}\|_2^2 &= \hat{y}^T \hat{T} \hat{T}^T \hat{y} \\ &= (sy^T, c) \begin{bmatrix} T & v \\ & \gamma \end{bmatrix} \begin{bmatrix} T^T & \\ v^T & \gamma \end{bmatrix} \begin{pmatrix} sy \\ c \end{pmatrix} \\ &= (s,c) \begin{bmatrix} y^T T T^T y + (y^T v)^2 & \gamma(y^T v) \\ \gamma(y^T v) & \gamma^2 \end{bmatrix} \begin{pmatrix} s \\ c \end{pmatrix} \\ &= (s,c)\, B \begin{pmatrix} s \\ c \end{pmatrix}, \end{aligned}$$

we can rewrite the objective function as a quadratic form, where $B \in \mathbf{R}^{2\times 2}$.

Theorem 1.2.1 *The matrix B is symmetric positive definite (s.p.d.) if $\hat{T}$ is non-singular. Hence the maximization problem*

$$\max_{\|(s,c)\|_2=1} \|\hat{y}(c,s)^T\hat{T}\|_2^2 \tag{1.6}$$

has as solution the eigenvector (s^*, c^*) of unit length belonging to the largest eigenvalue of B.

The calculation of $\|\hat{y}^T\hat{T}\|_2$ by a matrix-vector product at every step can be avoided by using the updating formula

$$\|\hat{y}^T\hat{T}\|_2 = \sqrt{s^2\|y^TT\|_2^2 + (s(y^Tv)+c\gamma)^2} \tag{1.7}$$

which is a consequence of

$$\hat{y}^T\hat{T} = \begin{pmatrix} sy^TT \\ sy^Tv + c\gamma \end{pmatrix}. \tag{1.8}$$

If we introduce the quantities

$$\alpha = y^Tv, \quad \delta = \|y^TT\|_2$$

and

$$\eta^2 = \alpha^2 + \delta^2, \quad \mu = \delta\gamma, \quad \nu = \alpha\gamma,$$

we find as the solution of (1.6):

for the case $\alpha \neq 0$

$$\begin{pmatrix} s \\ c \end{pmatrix} = \frac{u}{\|u\|_2}, \; u = \begin{pmatrix} \eta^2 - \gamma^2 + \sqrt{\eta^4 + 2\nu^2 - 2\mu^2 + \gamma^4} \\ 2\nu \end{pmatrix}$$

for the case $\alpha = 0$

$$\begin{pmatrix} s \\ c \end{pmatrix} = \begin{cases} \begin{pmatrix} 1 \\ 0 \end{pmatrix}, & \text{if } \delta > |\gamma|, \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Using (1.7), we can completely omit the calculation of $\hat{y}^T\hat{T}$ and compute $\hat{\delta}$ directly from δ.

1.2.3 Incremental norm estimation by approximate right singular vectors

In the previous section, we showed how to construct incrementally an approximation of the left singular vector corresponding to the largest singular value. We will now develop the scheme for the corresponding right singular vector. This might seem very natural, however we emphasize that it is not possible to extend the original ICE scheme described in Section 1.2.1 to right singular vectors. In Section 1.2.6, we look more closely at this problem of extending ICE. In order to make the following derivation more transparent, we point out that the key idea lies in focusing on the vector Tz rather than the approximate right singular vector z.

For an upper triangular matrix T, the issue is now to find a vector z of unit length so that

$$z^* = \arg \max_{\|z\|_2=1} \|Tz\|_2.$$

With the augmented matrix $\hat{T}$ defined as in (1.4), our approximate right singular vector is assumed to be of the form

$$\hat{z} = \hat{z}(s,c) = \begin{pmatrix} sz \\ c \end{pmatrix}, \tag{1.9}$$

where $s^2 + c^2 = 1$, exactly as in (1.5).

We state again the objective function as a quadratic form

$$\begin{aligned} \|\hat{T}\hat{z}\|_2^2 &= (s,c) \begin{bmatrix} z^T T^T T z & z^T T^T v \\ v^T T z & v^T v + \gamma^2 \end{bmatrix} \begin{pmatrix} s \\ c \end{pmatrix} \\ &= (s,c)\, C \begin{pmatrix} s \\ c \end{pmatrix}, \end{aligned}$$

and see, by the same arguments as in Section 1.2.2, that the solution (s^*, c^*) can be calculated analytically.

By exploiting the recurrence

$$\hat{T}\hat{z} = \begin{pmatrix} sTz + cv \\ c\gamma \end{pmatrix}, \tag{1.10}$$

we see that, as in Section 1.2.2, we can avoid a matrix-vector product at each stage.

If we define

$$\beta = v^T T z, \quad \epsilon = \|Tz\|_2 \quad \kappa^2 = v^T v + \gamma^2, \tag{1.11}$$

we have

for the case $\beta \neq 0$

$$\begin{pmatrix} s \\ c \end{pmatrix} = \frac{u}{\|u\|_2}, \; u = \begin{pmatrix} \epsilon^2 - \kappa^2 + \sqrt{\epsilon^4 + 4\beta^2 - 2\epsilon^2\kappa^2 + \kappa^4} \\ 2\beta \end{pmatrix}$$

and for the case $\beta = 0$

$$\begin{pmatrix} s \\ c \end{pmatrix} = \begin{cases} \begin{pmatrix} 1 \\ 0 \end{pmatrix}, & \text{if } \epsilon > |\gamma|, \\ \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \text{otherwise.} \end{cases}$$

Then, we can compute $\hat{\epsilon}$ from

$$\hat{\epsilon} = \|\hat{T}\hat{z}\|_2 = \sqrt{s^2\epsilon^2 + 2sc\beta + c^2\kappa^2}. \tag{1.12}$$

We point out that our algorithm works only with the vector Tz, the approximate right singular vector z is neither needed nor computed. However, it might become useful to know z. In this case, we suggest multiplying Tz by T^T and normalizing the result. This strategy of performing one step of power iteration with an appropriately chosen vector z was originally used by the LINPACK condition estimator [30], but we choose z differently.

1.2.4 Quality of incremental norm estimates

We investigate in this section when the updating formulae (1.5) and (1.9) lead to exact norm estimates.

We consider the singular value decomposition $T = U\Sigma V^T$ with singular values $\sigma_1 \geq \ldots \geq \sigma_n$ and orthogonal matrices $U = [u_1, \ldots, u_n]$ and $V = [v_1, \ldots, v_n]$.

We define as before

$$\hat{T} = \begin{bmatrix} T & v \\ & \gamma \end{bmatrix},$$

and assume in the following that $v \neq 0$. Then

$$\begin{aligned} \hat{T}\hat{T}^T &= \begin{bmatrix} TT^T + vv^T & \gamma v \\ \gamma v^T & \gamma^2 \end{bmatrix} \\ &= \begin{bmatrix} U & \\ & 1 \end{bmatrix} \begin{bmatrix} \Sigma^2 + U^T vv^T U & \gamma U^T v \\ \gamma v^T U & \gamma^2 \end{bmatrix} \begin{bmatrix} U^T & \\ & 1 \end{bmatrix} \\ &= \begin{bmatrix} U & \\ & 1 \end{bmatrix} \left\{ \begin{bmatrix} \Sigma^2 & \\ & 0 \end{bmatrix} + \begin{pmatrix} U^T v \\ \gamma \end{pmatrix} \begin{pmatrix} v^T U & \gamma \end{pmatrix} \right\} \begin{bmatrix} U^T & \\ & 1 \end{bmatrix}, \end{aligned}$$

and an eigenvalue λ of $\hat{T}\hat{T}^T$ is a solution of

$$\begin{aligned} 0 &= \det\left(\begin{bmatrix} \Sigma^2 - \lambda I & \\ & -\lambda \end{bmatrix} + \begin{pmatrix} U^T v \\ \gamma \end{pmatrix} \begin{pmatrix} v^T U & \gamma \end{pmatrix} \right) \\ &= \det(D - \lambda I + mm^T). \end{aligned}$$

Theorem 1.2.2 *If the vector m has no zero components, and the entries d_i of the diagonal matrix D satisfy $d_1 > \ldots > d_{n+1} = 0$, then any eigenvalue λ of $\hat{T}\hat{T}^T$ is a solution of*

$$\det\left(I + \begin{bmatrix} \Sigma^2 - \lambda I & \\ & -\lambda \end{bmatrix}^{-1} \begin{pmatrix} U^T v \\ \gamma \end{pmatrix} \begin{pmatrix} v^T U & \gamma \end{pmatrix} \right) = 0.$$

We have to show that for each eigenvalue λ_i of $\hat{T}\hat{T}^T$ and $v_i \neq 0$ satisfying $(D + mm^T)v_i = \lambda_i v_i$, the matrix $D - \lambda_i I$ is nonsingular. If $D - \lambda_i I$ is singular, then there exists a corresponding unit vector e_i with $(D - \lambda_i I)e_i = 0$. As $0 = e_i^T(D - \lambda_i I + mm^T)v_i = (e_i^T m)(m^T v_i)$, it follows that $(m^T v_i) = 0$ since m has no zero components. But then, $(D + mm^T)v_i = Dv_i = \lambda_i v_i$ and v_i has to be also a multiple of e_i, since the eigenvalues of D are distinct and all eigenspaces one-dimensional. Then $0 = (m^T v_i) = (m^T e_i)$ is a contradiction to the assumption that the vector m has only nonzero components.

Theorem 1.2.3 *Under the assumptions of Theorem 1.2.2, any eigenvalue λ of $\hat{T}\hat{T}^T$ is a solution of the secular equation*

$$0 = 1 + \sum_{i=1}^{n} \frac{(v^T u_i)^2}{\sigma_i^2(T) - \lambda} - \frac{\gamma^2}{\lambda}. \tag{1.13}$$

An eigenvector $\widetilde{u}_i$ of the matrix $D + mm^T$ corresponding to λ_i is a multiple of $(D - \lambda_i I)^{-1} m$.

The first part of this theorem is an immediate consequence of the previous one and the well known formula for the determinant of a rank-one update of the identity matrix $\det(I + xy^T) = 1 + y^T x$. Furthermore, since $(D - \lambda_i I)\widetilde{u}_i + (m^T \widetilde{u}_i)m = 0$ and $(D - \lambda_i I)$ is nonsingular, $\widetilde{u}_i$ and $(D - \lambda_i I)^{-1} m$ must be linearly dependent.

Theorem 1.2.4 *Let T be upper triangular and $T = U\Sigma V^T$ its singular value decomposition, with singular values $\sigma_1 > \ldots > \sigma_n > 0$ and orthogonal matrices $U = [u_1, \ldots, u_n]$ and $V = [v_1, \ldots, v_n]$. If*

$$\hat{T} = \begin{bmatrix} T & v \\ & \gamma \end{bmatrix},$$

then the vector recurrence $\hat{u}_1 = (s u_1^T, c)^T$ can only lead to an exact left singular vector belonging to the maximum singular value $\hat{\sigma}_1$ of $\hat{T}$, if u_1 and v are parallel.

By Theorem 1.2.3, an eigenvector of the matrix $D + mm^T$ corresponding to the eigenvalue λ_i is given by $\widetilde{u}_i = \alpha(D - \lambda_i I)^{-1} m$. Since

$$\hat{T}\hat{T}^T = \begin{bmatrix} U & \\ & 1 \end{bmatrix} (D + mm^T) \begin{bmatrix} U^T & \\ & 1 \end{bmatrix},$$

the corresponding eigenvector of the matrix $\hat{T}\hat{T}^T$ is given by

$$\hat{u}_i = \alpha \begin{bmatrix} U & \\ & 1 \end{bmatrix} (D - \lambda_i I)^{-1} m.$$

Assume that u_1 is the exact left singular vector belonging to the maximum singular value of T and $\hat{u}_1$ can be expressed by the recurrence relation (1.5) as $\hat{u}_1 = (su_1^T, c)^T$. Since for all j

$$\hat{u}_1^T \begin{pmatrix} u_j \\ 0 \end{pmatrix} = su_1^T u_j = \alpha m^T (D - \lambda_1 I)^{-T} \begin{bmatrix} U^T & \\ & 1 \end{bmatrix} \begin{pmatrix} u_j \\ 0 \end{pmatrix} = \widetilde{\alpha} m^T e_j,$$

it follows from $u_1^T u_j = 0$ for $j = 2, \ldots, n$ that $m(1:n)$ is a multiple of e_1. By the definition of

$$m = \begin{pmatrix} U^T v \\ \gamma \end{pmatrix},$$

the left singular vector u_1 and v must be parallel.

An investigation of the recurrence for the corresponding right singular vector leads to the same necessary condition as in Theorem 1.2.4.

Theorem 1.2.5 *Let $T = U\Sigma V^T$ and $\hat{T}$ be defined as before. The vector recurrence $\hat{v}_1 = (sv_1^T, c)^T$ can only lead to an exact right singular vector belonging to the maximum singular value $\hat{\sigma}_1$ of $\hat{T}$ if u_1 and v are parallel.*

From $\hat{v}_1 = (sv_1^T, c)^T$ and

$$\begin{aligned} \hat{\sigma}_1 \hat{v}_1^T &= \hat{u}_1^T \hat{T} \\ &= \alpha m^T (D - \lambda_i I)^{-T} \begin{bmatrix} U^T & \\ & 1 \end{bmatrix} \begin{bmatrix} T & v \\ & \gamma \end{bmatrix} \\ &= \alpha m^T (D - \lambda_i I)^{-T} \begin{bmatrix} \Sigma V^T & U^T v \\ & \gamma \end{bmatrix} \end{aligned}$$

follows again $m_j = 0$ for $j = 2, \ldots, n$, because

$$\begin{aligned} \hat{v}_1^T \begin{pmatrix} v_j \\ 0 \end{pmatrix} &= \alpha m^T (D - \lambda_1 I)^{-T} \begin{bmatrix} \Sigma V^T & U^T v \\ & \gamma \end{bmatrix} \begin{pmatrix} v_j \\ 0 \end{pmatrix} \\ &= \alpha m^T (D - \lambda_1 I)^{-T} \begin{pmatrix} \sigma_j e_j \\ 0 \end{pmatrix}. \end{aligned}$$

The next theorem says that the necessary condition $|u_1^T v| = \|v\|_2$ from Theorem 1.2.4 and 1.2.5 is also sufficient and gives an explicit formula for the 2-norm of the augmented triangular matrix. This formula has already appeared in [22].

Theorem 1.2.6 *Let $T = U\Sigma V^T$ and $\hat{T}$ be defined as in Theorem 1.2.4 and 1.2.5. If the condition $|u_1^T v| = \|v\|_2$ holds, then*

$$\|\hat{T}\|_2^2 = \hat{\sigma}_1^2 = \frac{\tau}{2} + \sqrt{\frac{\tau^2}{4} - \sigma_1^2\gamma^2}, \quad \tau = \sigma_1^2 + \|v\|_2^2 + \gamma^2. \tag{1.14}$$

Moreover, if the left singular vector u_1 of the matrix T is taken as the initial vector, then the vector $\hat{u}_1$ constructed by the left incremental norm estimator is exact, that is, it satisfies $\|\hat{u}_1^T\hat{T}\|_2 = \hat{\sigma}_1$. Likewise, the right incremental estimate $\hat{v}_1$ satisfies $\|\hat{T}\hat{v}_1\|_2 = \hat{\sigma}_1$, if the right singular vector v_1 is taken as the initial vector.

If $|u_1^T v| = \|v\|_2$ holds, then, because of the pairwise orthogonality of the vectors u_i, the secular equation (1.13) is simplified to

$$0 = 1 + \frac{\|v\|_2^2}{\sigma_1^2(T) - \lambda} - \frac{\gamma^2}{\lambda},$$

which has as largest root $\lambda_1 = \hat{\sigma}_1^2$ from (1.14).
From the proof of Theorem 1.2.4 follows with $u_1^T v = \pm\|v\|_2 e_1$ that the singular vector $\hat{u}_1$ can be expressed as

$$\begin{aligned}\hat{u}_1 &= \alpha \begin{bmatrix} U & \\ & 1 \end{bmatrix} (D - \lambda_1 I)^{-1} m \\ &= \alpha \begin{bmatrix} U & \\ & 1 \end{bmatrix} (D - \lambda_1 I)^{-1} \begin{pmatrix} \pm\|v\|_2 e_1 \\ \gamma \end{pmatrix} = \begin{pmatrix} \widetilde{s} u_1 \\ \widetilde{c} \end{pmatrix},\end{aligned}$$

hence $\hat{u}_1$ satisfies the vector recurrence (1.5) defining the incremental construction of the approximate left singular vectors. Since $\hat{u}_1$ corresponds to the maximum singular value, it holds that

$$(\widetilde{s}, \widetilde{c}) = \arg \max_{\|(s,c)\|_2 = 1} \|\hat{u}_1(c,s)^T \hat{T}\|_2^2,$$

and the incremental estimate is exact. Analogously, we have from the proof of Theorem 1.2.5 that

$$\hat{v}_1 = \frac{\alpha}{\hat{\sigma}_1} \begin{bmatrix} V\Sigma & \\ v^T U & \gamma \end{bmatrix} (D - \lambda_1 I)^{-1} \begin{pmatrix} \pm\|v\|_2 e_1 \\ \gamma \end{pmatrix} = \begin{pmatrix} \bar{s} v_1 \\ \bar{c} \end{pmatrix},$$

therefore $\hat{v}_1$ satisfies the vector recurrence (1.9). The incremental estimate is exact because

$$(\bar{s}, \bar{c}) = \arg \max_{\|(s,c)\|_2 = 1} \|\hat{T}\hat{v}_1(c,s)\|_2^2.$$

1.2.5 Incremental norm estimation for sparse matrices

The incremental condition estimator described in Section 1.2.1 is intended to be used with dense matrices. In its original form, ICE cannot be applied to sparse matrices as we illustrate through Example 1.2.3.

Example 1.2.3 *Consider the triangular matrix $A \in \mathbf{R}^{(n_B+n_C+n_D)\times(n_B+n_C+n_D)}$ with*

$$A = \left[\begin{array}{ccc} B & 0 & E_B \\ 0 & C & E_C \\ 0 & 0 & D \end{array} \right] \begin{array}{l} \} \, n_B \\ \\ \\ \end{array} .$$

After step n_B, we obtain from ICE an approximate left singular vector x_B. In the next step, the first column of the second block column is appended, where $\gamma = a(n_B+1, n_B+1)$ is the only nonzero entry. ICE will now take either $\hat{x} = (x_B, 0)^T$ or $\hat{x} = (0, 1)^T$ as an approximate left singular vector for the augmented triangular matrix. But once a component in the approximate left singular vector is set to zero, this choice cannot be undone later in the calculation, independent of the entries in the right border. Thus, the sparsity of the matrix A can cause the quality of the estimate to be very poor.

The modifications proposed to ICE in [23] to overcome this problem are as follows: for each block on the diagonal generate a separate approximate left singular vector, and then merge these subvectors together where the weights of each subvector are computed using a block version of ICE. This again requires the computation of the eigenvector belonging to the largest eigenvalue of a s.p.d. matrix, but this matrix will be of order k where k is the number of diagonal blocks rather than of order 2. As this eigenvector (for $k > 4$) can no longer be computed analytically, the solution of a secular equation using rational approximations is used.

The reason for the failure of ICE on sparse matrices is that, while the upper triangular matrix is augmented *column by column*, the incremental condition estimator uses *left* approximate singular vectors and thus calculates a weighted linear combination of the *rows*. This problem would not occur if it was possible to base ICE on approximate right singular vectors.

What does this imply for the incremental norm estimation? By the same reasoning as in the case of ICE, we expect to encounter similar problems to the incremental norm estimator for sparse matrices if we use approximate singular vectors from the left side. Fortunately, we can use approximate right singular vectors in the case of columnwise augmentation of the matrix as we have shown in Section 1.2.3. This allows us to use the scheme for dense matrices in the sparse case, too. Mathematically, the difference between the left and the right incremental norm estimator can be seen by comparing the two update formulas (1.5) and (1.10). The update of the approximate left singular vector $\hat{y} = (sy^T, c)^T$ does not change a vector component once it becomes zero; this can result in poor estimates as shown in the example above. However, the formula (1.10)

$$\hat{T}\hat{z} = \begin{pmatrix} sTz + cv \\ c\gamma \end{pmatrix}$$

shows that the vector $\hat{T}\hat{z}$ is essentially a linear combination of the old vector Tz and the new column appended to the triangular matrix. Even if a component of Tz has become zero during the previous steps of the algorithm, it can change again if the corresponding component of the vector v is nonzero. In this sense, the right incremental norm estimator can 'recover' from zeros introduced by sparsity, in contrast to the left scheme.

1.2.6 The relationship between incremental norm and condition estimation

We now present a more detailed investigation of the relationship between the incremental norm and incremental condition estimators described in the previous sections. In particular, we show why incremental condition estimation is less flexible with respect to the use of approximate singular vectors from both sides.
For the following discussion, we use the nomenclature

$$\hat{T} = \begin{bmatrix} T & v \\ & \gamma \end{bmatrix}, \quad \hat{T}^{-1} = \begin{bmatrix} T^{-1} & u \\ & \gamma^{-1} \end{bmatrix}.$$

Let us first look at the incremental condition estimator ICE. The scheme constructs from a vector d of unit norm the next vector $\hat{d}$ incrementally as

$$\hat{d} = \hat{d}(s,c) = \begin{pmatrix} sd \\ c \end{pmatrix}.$$

The elegance of the scheme lies in the fact that it is *not* necessary to solve the equation $\hat{x}^T\hat{T} = \hat{d}^T$ if the solution of the previous equation $x^T T = d$ is known. Instead, $\hat{x}$ can be computed directly from x through the update formula given in equation (1.3).

The problem is that the analogous update formula for ICE based on approximate right singular vectors is not practical. An investigation of the equation $\hat{T}\hat{z} = \hat{d}$ reveals the update formula

$$\hat{z} = \begin{pmatrix} sz + cu \\ c/\gamma \end{pmatrix}. \tag{1.15}$$

Note that this formula involves the vector u which is part of the *inverse* matrix $\hat{T}^{-1}$. (Theoretically, u could be computed from $u = -1/\gamma T^{-1}v$, however, this would increase the costs of one incremental step from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$!) As was shown in Sections 1.2.2 and 1.2.3, the incremental norm estimator does not have the same problem. Estimators based on both left as well as right singular vectors have update formulae that involve only terms of the original matrix T (see equations (1.8) and (1.10)).

The incremental approach to norm estimation is a direct generalization of the concepts used in ICE. Indeed it is the case that the application of the incremental norm estimator, using approximate left singular vectors, to the matrix T^{-1} is mathematically *equivalent* to applying the incremental condition estimator ICE to the matrix T. This follows by substituting the matrix T by its inverse T^{-1} in the derivation of Section 1.2.2.

1.2.7 Computational costs of incremental norm estimation

We now give an analysis of the work and storage demands of the incremental norm estimators. We regard the transition from the triangular matrix $T \in \mathbf{R}^{n\times n}$ to the augmented matrix $\hat{T} \in \mathbf{R}^{(n+1)\times(n+1)}$.

We begin with the left incremental norm estimator described in Section 1.2.2. The key formulas for this technique are given by the update of the approximate left singular vector (1.5) and the update of the norm estimate (1.7). From (1.7), it follows that the norm estimate $\delta = \|y^T T\|_2$ and the approximate left singular vector y of the matrix T are required for the computation of the new norm estimate $\hat{\delta} = \|\hat{y}^T \hat{T}\|_2$. The storage requirements for the update are therefore $n+1$. The new approximate singular vector $\hat{y}^T$ is essentially a scaling of y, and the expensive part of the norm estimate update is given by the dot product $y^T v$. Thus, the total computational costs of the update step are approximately $3n$.

We now consider the right incremental norm estimator described in Section 1.2.3. Fundamental for this algorithm is the update of the vector Tz as given in (1.10) and the update of the norm estimate (1.12). We see that $\epsilon = \|Tz\|_2$ needs to be stored for the computation of the new estimate $\hat{\epsilon} = \|\hat{T}\hat{z}\|_2$, and Tz is needed for $\hat{T}\hat{z}$. Overall, we need to store $n+1$ numbers as for the left incremental norm estimator. However, the right incremental norm estimator is more expensive as can be seen from (1.10) and the constants defined in (1.11). The computation of the vector $\hat{T}\hat{z}$ costs $3n$ operations, since the sum of the scaled vector Tz and the scaled rightmost column of $\hat{T}$ has to be formed. The update of the norm estimate requires the computation of the two dot products $v^T(Tz)$ and $v^T v + \gamma^2$, together $4n$ operations. Thus, the overall costs for the update step are $7n$.

The above analysis applies for the case of a dense matrix T and shows that in this case, the left incremental approach is clearly better with respect to computational costs. If we assume that the vector v is sparse with $m \ll n$ entries, the difference between the two schemes becomes significantly less important. Since also in the case of a sparse T, both the left and the right approximate singular vectors are generally dense, the scaling of y and Tz by the scalar s will still cost n operations. However, the cost of any operation involving v will now be reduced from n to m. Therefore, we obtain $n+2m$ as the operation count for the left incremental scheme and $n+6m$ for the right incremental scheme.

1.3 Triangular factorizations with inverse factors

In this section, we briefly describe the incorporation of the inversion of a triangular factor into a QR factorization. This algorithm will be the basis for our numerical tests which are reported in Section 1.5. We remark that the explicit computation of matrix inverses arises for example in signal processing applications [31, 111].

1.3.1 The QR factorization with simultaneous inversion of R

There are several ways to combine a standard QR factorization with a simultaneous inversion of R. It is important to consider both the performance and the stability of the inversion algorithm. Both aspects were investigated in [45]. Of all the methods discussed in that paper, we decided to implement a method rich in Level 2 BLAS matrix-vector multiplies. This algorithm was already considered by Householder [90]. Lemma 1.3.1 describes the basis of the inversion algorithm.

Lemma 1.3.1 *Assume that* $R \in \mathbf{R}^{i\times i}$ *and that the first* $i-1$ *columns of* $Y = R^{-1}$ *have already been computed. Then, the* i^{th} *column of* Y *can be computed from*

$$\begin{aligned} Y(i,i) * R(i,i) &= 1, \\ Y(1:i-1,i) * R(i,i) &= -Y(1:i-1,1:i-1) * R(1:i-1,i). \end{aligned}$$

This is a consequence of a columnwise evaluation of $YR = I$.

If we combine a QR factorization based on the modified Gram-Schmidt algorithm [70] with the simultaneous inversion described by Lemma 1.3.1, we get Algorithm 1.

Algorithm 1 QR factorization with simultaneous inversion.

```
  $[n,n] = size(A);$
  $Q = zeros(m,n);$
  $R = zeros(n);$
  $Y = zeros(n);$
  for i = 1:n do
    $R(i,i) = norm(A(:,i),2);$
    $Q(:,i) = A(:,i)/R(i,i);$
    for j = i+1:n do
      $R(i,j) = (Q(:,i))' * A(:,j);$
      $A(:,j) = A(:,j) - R(i,j) * Q(:,i);$
    end for
    $Y(i,i) = 1/R(i,i);$
    if $i > 1$ then
      $Y(1:i-1,i) = Y(1:i-1,1:i-1) * R(1:i-1,i);$
      $Y(1:i-1,i) = -Y(i,i) * Y(1:i-1,i);$
    end if
  end for
```

1.3.2 Stability issues of triangular matrix inversion

The numerical stability properties of general triangular matrix inversion were investigated in [45]. The inversion in Algorithm 1 is just Method 2 from [45] adapted for upper triangular matrices. An error analysis similar to the one performed there establishes the following componentwise residual bound for the computed inverse $\bar{Y}$:

$$|\bar{Y}R - I| \leq c_n u |\bar{Y}||R| + \mathcal{O}(u^2)$$

where c_n denotes a constant of order n and u the unit roundoff. The interpretation of this result is that the residual $\bar{Y}R - I$ can be bounded componentwise by a small multiple of the unit roundoff times the size of the entries in $\bar{Y}$ and R. The bound illustrates the reliability of our algorithm for matrix inversion, but the remark on page 18 of [45] should be recalled:

... we wish to stress that all the analysis here pertains to matrix inversion alone. It is usually the case that when a computed inverse is used as part of a larger computation the stability properties are less favourable, and this is one reason why matrix inversion is generally discouraged.

Indeed, the authors give an example illustrating that the solution of $Rx = b$ by the evaluation $R^{-1}b$ need *not* be backward stable if R^{-1} has first to be computed from R.

We conclude that one would not compute R^{-1} instead of R just to replace ICE by INE. ICE is the condition estimator for standard factorizations in the classical sense of giving an estimate of the smallest singular value. However, if an inverse matrix arises from the application, the norm estimator INE can serve as a condition estimator.

1.4 Inverses in sparse factored form

1.4.1 Sparse storage of triangular inverses

The inverse of a sparse matrix A is generally less sparse than A itself and indeed, if the matrix is irreducible, its inverse is structurally dense, see for example [47]. As was observed by the pioneers in linear programming some few decades ago [19], [61], the inverse of a sparse triangular matrix can be stored with exactly the same storage as the matrix itself, that is without fill-in. This is described in Lemma 1.4.1.

Lemma 1.4.1 *Let $R \in \mathbf{R}^{n \times n}$ be an upper triangular matrix. Denote by R_i an elementary matrix, equal to the identity matrix except for row i where it is identical to the i-th row of R. Then:*

1. $R = R_n R_{n-1} \ldots R_1$
2. *Let $S_i = R_i^{-1}$. Then S_i has exactly the same sparsity structure as R_i and is, apart from row i equal to the identity matrix. Note that S_i does not contain the i-th row of Y.*

Both results can be checked by calculation.

The lemma suggests that we can obtain a no fill-in representation of the inverse by storing the factors $S_i, i = 1, \ldots, n$ instead of R^{-1}.

Although this is very good from the point of view of sparsity it unfortunately causes problems for the detection of ill-conditioning. For example, the factored representation of T_n^{-1}, where T_n is the matrix of Example 1.2.1, is given by the tableau

$$\left[\begin{array}{c|c|c|c} 1 & \gamma & \ldots & \gamma \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \gamma \\ 0 & \ldots & 0 & 1 \end{array}\right].$$

Here, row i of Y holds the non-trivial row of the elementary matrix S_i. We see that the exponential growth of the matrix entries does not show up in the factored form, that is the ill-conditioning is *hidden* by this implicit representation of the inverse. From this example, we conclude that we need to calculate the inverse explicitly to avoid hiding the ill-conditioning. For most matrices, it is not possible to do this without fill-in, however, in [4], it is shown how the number of factors in the sparse factored form of the inverse can be reduced while still avoiding fill-in so long as the matrix satisfies a certain condition. The original intention of [4] was to enhance parallelism in the solution of triangular systems, but we use the idea here to help detect ill-conditioning.

In order to explain the method, we introduce the following nomenclature:
For an upper triangular matrix $R \in \mathbf{R}^{n \times n}$, its directed acyclic graph $G(R)$ is the pair (V, E) where $V = \{1, \ldots, n\}$ and $E = \{(i,j) | i \neq j \text{ and } R(i,j) \neq 0\}$. For $(i,j) \in E$, i is called a predecessor of j and j a successor of i. The transitive closure of a directed graph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{(i,j) | \exists \text{ path } i \rightarrow j \text{ in } G\}$.

Theorem 1.4.2 *[67] Let R be a nonsingular upper triangular matrix. Then*

$$G(R^{-1}) = (G(R))'.$$

This theorem allows us to extend Lemma 1.4.1 by showing that the restriction to using elementary matrices as factors is not necessary. Instead, we can consider *blocks* of rows of R where the corresponding *generalized* elementary matrix has a transitively closed directed graph. By generalized elementary matrix, we mean the matrix which is equal to the identity except for the rows belonging to the block where it is identical to the corresponding rows of R.

Example 1.4.1 *Consider the matrix*

$$R = \begin{bmatrix} 11 & 12 & 13 & 14 \\ & 22 & 23 & 0 \\ & & 33 & 34 \\ & & & 44 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 33 & 34 \\ & & & 44 \end{bmatrix} \begin{bmatrix} 11 & 12 & 13 & 14 \\ & 22 & 23 & 0 \\ & & 1 & \\ & & & 1 \end{bmatrix}.$$

From Theorem 1.4.2, we see that each of the factors can be inverted without fill-in.

It is desirable to look for a representation of the inverse with the smallest number of factors possible. The inclusion of this row blocking strategy into Algorithm 1 will then result in a hybrid algorithm that uses the sparse representation of the inverse but also reveals possible hidden ill-conditioning of dense submatrices. In particular, this algorithm can handle the pathological matrices in Examples 1.2.1 and 1.2.2.

To formalize the objectives, the algorithm should find a partition $0 = e_1 < e_2 < \ldots < e_{m+1} = n$ so that

$$R^{-1} = S_1 \ldots S_m \tag{1.16}$$

where S_k is the inverse of the generalized elementary matrix corresponding to the rows $e_k + 1, \ldots, e_{k+1}$ of R and the number of factors m is as small as possible.

The following Lemma is now an immediate consequence of Theorem 1.4.2.

Lemma 1.4.3 *Assume that the generalized elementary matrix corresponding to the rows $e_k + 1, \ldots, j-1$ of R is invertible without fill-in. Then the augmented generalized elementary matrix corresponding to the rows $e_k + 1, \ldots, j$ of R is invertible without fill-in if and only if the Condition 1.4.1 is satisfied.*

Condition 1.4.1 *Every successor s of j is also a successor of all predecessors $p \geq e_k + 1$ of j.*

The following theorem shows the optimality of the partition.

Theorem 1.4.4 *[4] A partitioning with maximum row blocking based on Condition 1.4.1 leads to a sparse representation of R^{-1} with the smallest possible number of factors.*

It is interesting to see how easily the row blocking can be incorporated into our inversion algorithm for triangular matrices. The following analogue to Lemma 1.3.1 shows how Algorithm 1 has to be modified.

Lemma 1.4.5 *Assume that Condition 1.4.1 holds for $e_k + 1, \ldots, i$ and that the columns $e_k + 1, \ldots, i-1$ of S_k have already been computed. Then, the i^{th} column of S_k can be computed from*

$$\begin{aligned} Y(i,i) * R(i,i) &= 1, \\ Y(e_k+1:i-1, i) * R(i,i) &= -Y(e_k+1:i-1, e_k+1:i-1) \\ &\quad *R(e_k+1:i-1, i). \end{aligned}$$

We remark that the stability of the partitioned inverse method in the context of solving triangular linear systems has been studied in [87]. Generally, the comments given at the end of Section 1.3.2 also apply here.

1.4.2 Incremental norm estimation for sparse factored inverses

The application of any incremental scheme to a factored representation is a difficult problem. As can be seen from (1.7) and (1.10), it is always assumed that we have access to the full column being appended. However, in the factored approach a column might not be stored explicitly because of fill-in, see Section 1.4. The column could be generated but the high cost of its computation from the factors might spoil the effectiveness of the scheme.

Although we cannot give a full solution to this problem, we suggest at least a partial remedy as follows:

1. It is possible to use

$$\|R^{-1}\|_2 \approx \prod_{i=1}^{m} \|S_i\|_2. \tag{1.17}$$

   to obtain an estimated upper bound for the condition number of $Y = R^{-1}$. In our tests, we found that the product on the right-hand side is often a severe overestimation of $\|R^{-1}\|_2$, even if each factor $\|S_i\|_2$ is an underestimate. Although there are circumstances where an overestimate is useful (for example, if the value is not too large then we are fairly sure the matrix is not ill-conditioned), the use of (1.17) can be very unreliable.

2. The cost for the computation of an off-diagonal block depends on the number of factors in the sparse representation, the graph $G(R^{-1})$, and the position of the block. The example

$$\begin{aligned} Y &= \begin{bmatrix} S_{11} & S_{12} & S_{13} \\ & I & \\ & & I \end{bmatrix} \begin{bmatrix} I & & \\ & S_{22} & S_{23} \\ & & I \end{bmatrix} \begin{bmatrix} I & & \\ & I & \\ & & S_{33} \end{bmatrix} \\ &= \begin{bmatrix} S_{11} & S_{12}S_{22} & (S_{12}S_{23} + S_{13})S_{33} \\ & S_{22} & S_{23}S_{33} \\ & & S_{33} \end{bmatrix} \end{aligned}$$

   illustrates this. If R^{-1} is very sparse, the computation of the columns of Y from its factors becomes affordable. We can use a blocked version of our scheme to first calculate approximate singular vectors corresponding to the diagonal blocks and afterwards merge them together to obtain an approximate singular vector for the whole system.

1.5 Numerical tests

In this section, we present tests of our incremental schemes with dense and sparse matrices. We use the algorithm described in Section 1.3.1 which allows us to use our norm estimator on both the triangular factor and its inverse.

In Table 1.1, we show the incremental estimates for the QR factorization of sparse matrices from the Harwell-Boeing collection [48]. Here, the second column displays the exact matrix norm of R as calculated by MATLAB, the third and fourth columns show estimations based on approximate left singular vectors (Section 1.2.2) and on approximate right singular vectors (Section 1.2.3), respectively. If we denote by c_l the costs for applying the left incremental estimator to the matrix R, we obtain from Section 1.2.7 that $c_l = \sum_{i=1}^{n} i + 2n_{nz}(R) = n^2/2 + 2n_{nz}(R)$, where n denotes the size of the matrix and $n_{nz}(R)$ the number of nonzero elements. Likewise, we obtain $c_r = n^2/2 + 6n_{nz}(R)$ as costs for the right incremental estimator. The cost quotient $q_c = c_r/c_l$ is given in the fifth column of the table. The following four columns of the table hold the corresponding numbers for the inverse factor R^{-1}. Furthermore, preordering the matrix A is potentially beneficial for the sparsity of the triangular factors R and R^{-1}. Following the suggestions in [21] and [25], we test our scheme also on $A\Pi$, where Π is the column minimum degree ordering. The results for the permuted matrices are reported below those for the corresponding unpermuted ones and are marked by *colmmd*.

In general, both of our estimators give a good approximation to the norm of R. Note that, because we compute the norm using an approximate singular vector, our estimate will always be a lower bound for the norm. We cannot find a general superiority of the right incremental estimates over the left incremental ones. However, as indicated in Section 1.2.5, the incremental approach for upper triangular matrices based on approximate left singular vectors can lead to problems for sparse matrices and we see this in a few of our test cases, most noticeably the case of the matrix `arc130` and its inverse, where the incremental approach based on right singular vectors gives a much better estimate than the one using left singular vectors. While the column minimum degree ordering does not generally improve the norm estimate, our results show that in most cases it reduces substantially the costs of the schemes. Furthermore, the difference between the two schemes in terms of costs can become significantly less pronounced as for example seen for the permuted matrices `bfw398a` and `impcol_c`.

In Table 1.2, we show the incremental estimates for the norm of R^{-1} from the QR factorization of dense matrices from the Matlab Test Matrix Toolbox [84]. Specifically, we use the matrices from Example 1.2.1, called `condex` in [84], and Example 1.2.2, named `kahan`. Furthermore, we include tests with Hilbert matrices (`hilb`). We apply our incremental norm estimator to the inverse of the matrices, since the ill-conditioning is evident there rather than in the matrices themselves. We note that both of our estimates are always very close to the real norm, however, as the cost difference between the two schemes is $c_r - c_l = 4n_{nz}(R) = 2n^2$ for dense matrices, the left incremental scheme is preferable.

In Tables 1.3 and 1.4, we show the incremental estimates for the QR factorization of random matrices with uniform and exponentially distributed singular values, respectively. For each of the different matrix sizes n, we created 50 random matrices $A = U\Sigma V^T$ choosing different random matrices U, V, and singular values either

Name	$\|R\|_2$	l-est.	r-est.	q_c	$\|R^{-1}\|_2$	l-est.	r-est.	q_c
arc130	2.39e+5	1.91e+2	2.37e+5	2.3	2.52e+5	2.15e+2	2.52e+5	2.3
(colmmd)	2.39e+5	2.37e+5	2.37e+5	1.7	2.52e+5	2.15e+2	1.81e+5	1.9
bfw398a	1.04e+1	9.45e+0	8.69e+0	2.2	2.87e+2	9.06e+1	2.86e+2	2.3
(colmmd)	1.04e+1	8.28e+0	9.32e+0	1.5	2.87e+2	2.91e+1	2.77e+2	1.9
cavity04	7.12e+1	3.53e+1	6.39e+1	1.9	4.97e+4	1.03e+4	4.87e+4	2.0
(colmmd)	7.12e+1	5.33e+1	6.99e+1	1.8	4.97e+4	5.70e+3	4.94e+4	1.9
e05r0400	4.59e+1	1.79e+1	4.13e+1	2.2	1.10e+4	2.67e+3	1.06e+4	2.3
(colmmd)	4.59e+1	3.64e+1	3.86e+1	2.1	1.10e+4	2.76e+3	1.09e+4	2.2
fidap001	1.30e-1	1.14e-1	1.19e-1	2.0	2.53e+5	1.81e+5	2.08e+5	2.3
(colmmd)	1.30e-1	1.23e-1	1.27e-1	2.0	2.53e+5	1.79e+5	2.09e+5	2.3
fs_183_1	1.12e+9	8.22e+8	1.27e+7	2.3	1.94e+4	1.08e+3	1.93e+4	2.3
(colmmd)	1.12e+9	1.12e+9	1.12e+9	1.9	1.94e+4	1.88e+4	1.94e+4	2.2
impcol_b	8.63e+0	3.18e+0	8.48e+0	2.0	1.89e+4	1.30e+4	1.89e+4	2.3
(colmmd)	8.63e+0	2.41e+0	8.58e+0	1.7	1.89e+4	8.74e+3	1.89e+4	2.0
impcol_c	1.20e+2	9.77e+0	1.20e+2	1.9	1.47e+2	3.54e+1	1.47e+2	2.3
(colmmd)	1.20e+2	1.20e+2	1.20e+2	1.4	1.47e+2	4.59e+1	1.47e+2	1.9
lns__131	9.77e+9	9.54e+9	9.10e+9	2.2	1.30e+5	5.83e+4	1.30e+5	2.3
(colmmd)	9.77e+9	3.98e+9	6.32e+9	1.6	1.30e+5	5.84e+4	1.30e+5	2.0
saylr1	4.83e+8	4.14e+8	4.81e+8	1.6	1.61e+0	7.61e-1	1.10e+0	2.3
(colmmd)	4.83e+8	2.37e+8	4.35e+8	1.5	1.61e+0	6.02e-1	1.59e+0	1.9
steam1	2.17e+7	1.88e+7	2.17e+7	2.1	1.30e+0	6.58e-1	1.18e+0	2.3
(colmmd)	2.17e+7	8.71e+6	2.17e+7	1.7	1.30e+0	1.29e+0	1.29e+0	2.1
str____0	1.39e+1	7.33e+0	1.39e+1	1.1	1.96e+1	2.97e+0	1.69e+1	1.3
(colmmd)	1.39e+1	1.03e+1	1.31e+1	1.2	1.96e+1	9.26e+0	1.95e+1	1.5
west0381	1.71e+3	1.39e+3	1.71e+3	2.2	7.31e+2	2.10e+2	7.18e+2	2.3
(colmmd)	1.71e+3	5.36e+2	1.71e+3	1.7	7.31e+2	9.37e+1	7.16e+2	2.1

Table 1.1: Results with matrices from the Harwell-Boeing collection. The column headers *l-est.* and *r-est.* denote left and right estimates, respectively. The cost quotient q_c describes the ratio of costs between right and left estimates in number of operations.

uniformly distributed as

$$\sigma_i = norm(A)/i, \quad 1 \le i \le n,$$

or exponentially distributed as

$$\sigma_i = \alpha^i, \quad 1 \le i \le n, \quad \alpha^n = norm(A),$$

where the norm of A was chosen in advance. The random orthogonal matrices U and V were generated using a method of Stewart [124] which is available under the name *qmult* in [84]. The values displayed in the table are the averages from 50 tests each. These tests seem to indicate that in the case of dense upper triangular matrices, the norm estimation based on approximate right singular vectors is more accurate than that based on approximate left singular vectors. However, we don't have a theoretical foundation for this.

Name	Size	$\|R^{-1}\|_2$	l-est.	r-est.	$c_r - c_l$
`condex(n,3)`	50	3.75e+14	3.72e+14	3.75e+14	5000
	75	1.26e+22	1.25e+22	1.26e+22	11250
	100	4.23e+29	4.19e+29	4.23e+29	20000
`kahan(n)`	50	6.43e+07	6.09e+07	6.43e+07	5000
	75	8.50e+11	8.06e+11	8.50e+11	11250
	100	1.12e+16	1.07e+16	1.12e+16	20000
`hilb(n)`	50	8.95e+17	3.74e+17	8.37e+17	5000
	75	1.77e+18	5.29e+17	1.71e+18	11250
	100	1.82e+18	7.26e+17	1.63e+18	20000

Table 1.2: Results with matrices from the Matlab Test Matrix Toolbox. The column headers *l-est.* and *r-est.* denote left and right estimates, respectively. The rightmost column shows the difference between the costs for right and left estimate $c_r - c_l$ in number of operations.

Size	$\|R\|_2$	l-est.	r-est.
50	1.00e+01	8.82e+00	9.24e+00
	1.00e+06	8.79e+05	9.21e+05
	1.00e+12	8.77e+11	9.20e+11
75	1.00e+01	8.82e+00	9.23e+00
	1.00e+06	8.82e+05	9.24e+05
	1.00e+12	8.80e+11	9.21e+11
100	1.00e+01	8.81e+00	9.22e+00
	1.00e+06	8.78e+05	9.18e+05
	1.00e+12	8.73e+11	9.17e+11

Table 1.3: Results (averages) with random matrices, σ_i uniformly distributed. The column headers *l-est.* and *r-est.* denote left and right estimates, respectively.

When we regard the results of all tests together, we find it difficult to give a general statement on which heuristics should be preferred. From its derivation, it is clear that, for sparse matrices, the right incremental approach is more accurate than the left incremental one. However, even for sparse matrices it is possible that the left incremental estimator gives a better approximation. For better robustness, we suggest applying the left and right incremental estimator *together* and take the best of the two estimates, as in all our tests at least one of them is accurate.

Size	$\|R\|_2$	l-est.	r-est.
50	1.00e+01	8.55e+00	9.10e+00
	1.00e+06	8.52e+05	9.59e+05
	1.00e+12	8.89e+11	9.92e+11
75	1.00e+01	8.52e+00	9.11e+00
	1.00e+06	8.20e+05	9.52e+05
	1.00e+12	8.32e+11	9.82e+11
100	1.00e+01	8.54e+00	9.11e+00
	1.00e+06	8.08e+05	9.31e+05
	1.00e+12	8.20e+11	9.58e+11

Table 1.4: Results (averages) with random matrices, σ_i exponentially distributed. The column headers *l-est.* and *r-est.* denote left and right estimates, respectively.

1.6 Conclusions and future work

In the context of the direct solution of linear systems, one often employs condition estimators for triangular matrices in order to have a cheap but reliable statement on the quality of the computed solution. The obtained estimates for the condition numbers of the triangular systems allow to estimate the magnitude of the error in the solution. Incremental condition estimators are particularly adapted to monitoring an ongoing triangular factorization to detect ill-conditioning. However, while working well in the dense case, previously existing schemes have been difficult to adapt to sparse matrices.

In this chapter of the thesis, we have shown how a new incremental norm estimator can be developed for a triangular factor. We have pointed out the suitability of the scheme for both dense and sparse matrices due to the fact that it can use approximate singular vectors both from the left and the right side. We have stated necessary and sufficient conditions for the applicability of the new algorithm. Furthermore, we have given an analysis of its complexity in terms of floating point operations and memory. To demonstrate the efficacy of our approach in practice, we have shown results on standard test examples including both sparse and dense matrices. These studies have been published in [59]. We point out that while we have been concerned with estimating the largest singular value of a matrix, we can also generalize the new incremental approach of Section 1.2.3 to estimate the *smallest* singular value. This can be done by looking in each incremental step for the pair (s, c) that minimizes the norm of $\hat{z}$ in (1.9).

Some topics remain subject of further research. A first question concerns matrix inverses which are stored in sparse factored form. These are difficult to treat by incremental estimators because they need access to appended columns, whereas in the factored approach, a column might not be stored explicitly because of fill-in, A second direction of research concerns the application of incremental norm estimation in the framework of the implicit LU factorization and, more generally, in the context of rank revealing. In particular, apart from [114] little has been done to explore the potential of rank revealing methods for sparse matrices where the capabilities of our algorithm to treat also these matrices could become important.

II

Chapter 2

Computational kernels for unstructured sparse matrices

A sparse matrix is a matrix many of whose entries are zero and for which some advantage can be taken of this fact, either for storage or operations with the matrix. In this part of the thesis, we consider the design and implementation of the Sparse BLAS, a set of computational kernels for unstructured sparse matrices. These Basic Linear Algebra Subprograms for sparse matrices are part of the new BLAS standard [1] which has been developed by the BLAS Technical Forum, and for which we have provided a reference implementation [57]. Our implementation of the Sparse BLAS has occurred in parallel to the design of the standard. In fact, our experience actually has influenced the final definition of the standard [50]. For example, the error flags in the interface were introduced in the final version of the standard after the implementation of an earlier version. These diagnostic error flags are used to indicate the success or failure of a computation in order to provide verifiable reliability to users.

The development of the Sparse BLAS standard is motivated by a shortage of sparse matrix support in the original BLAS. The Basic Linear Algebra Subprograms in their initial version [39, 40] have some support for sparse matrices, specifically, banded and packed matrices, and LAPACK [13] provides kernels for tridiagonal matrices, but neither library offers functionalities for matrices with more irregular sparsity patterns.

The problem of providing kernels for unstructured matrices arises from the fact that data access changes from one matrix to another, depending on its pattern. This stands in contrast to regularly structured sparse and dense matrices, where one *a priori* knows the shape of the matrix and can tune data structures and data access accordingly. Furthermore, the way of storing and accessing the nonzero entries of a sparse matrix often depends on the underlying application, see the remarks in [50].

However, it is important to provide a common programming interface for sparse matrix computations that is not limited to one specific storage format or matrix layout. With such interfaces, an application programmer can write portable, generic algorithms for sparse matrices without having to care about the underlying data handling. The programmer then can perform high-level operations such as matrix-vector multiplications with abstract matrix representations with a single call of a Sparse BLAS subroutine, which aids the development of algorithms. Furthermore, depending on the specific machine architecture, a vendor can provide efficient implementations of the algorithms that are hidden behind the Sparse BLAS interfaces. Thus, from the viewpoint of application development, the standardized Sparse BLAS kernels ensure easier coding as well as enhanced code portability.

2.1 Introduction, history, and overview of the development of the Sparse BLAS

The Basic Linear Algebra Subprograms (BLAS), which provide essential functionalities for dense matrix and vector operations, are a milestone in the history of numerical software. BLAS have been proposed for operations on dense matrices for some time, with the original paper for vector operations (Level 1 BLAS) appearing in 1979 [100]. This was followed by the design of kernels for matrix-vector operations (Level 2 BLAS) [40] and matrix-matrix operations (Level 3 BLAS) [39]. The Level 3 BLAS have proved to be particularly powerful for obtaining close to peak performance on many modern architectures since they amortize the cost of obtaining data from main memory by reusing data in the cache or high level memory.

For some years it has been realized that the BLAS standard needed updating and a BLAS Technical Forum was coordinated and has recently published a new standard [1]. Some of the main features included in the new standard are added functionality for computations in extended and mixed precision, and basic subprograms for sparse matrices (the Sparse BLAS). The need for the latter is particularly important for the iterative solution of large sparse systems of linear equations and eigenvalue problems.

As in the dense case, the Sparse BLAS enables the algorithm developer to rely on a standardized library of frequently occurring linear algebra operations and allows code to be written in a meta-language that uses these operations as building blocks. Additionally, vendors can provide implementations that are specifically tuned and tested for individual machines to promote the use of efficient and robust codes. The development of the Sparse BLAS standard has its roots in [37], [38] and [53], the first proposals for Level 1 and Level 3 kernels for the Sparse BLAS. While the final standard [50] has evolved from these proposals, these papers are not only of historical interest but also contain suggestions for the implementor which are deliberately omitted in the final standard.

Similarly to the BLAS, the Sparse BLAS provides operations at three levels, although it includes only a small subset of the BLAS functionality. Level 1 covers basic operations on sparse and dense vectors, Level 2 and Level 3 provide sparse matrix multiplication and sparse triangular solution on dense systems that may be vectors (Level 2) or matrices (Level 3). We emphasize that the standard is mainly intended for sparse matrices without a special structure. This has a significant influence on the complexity of the internal routines for data handling. Depending on the matrix, the algorithm used, and the underlying computing architecture, an implementor has to choose carefully an internal representation of the sparse matrix.

However, there is one important difference between the Sparse BLAS and the other BLAS interfaces in the standard. The Sparse BLAS supports the use of abstract matrix representations. Level 2 and Level 3 operations take as input not a data structure holding the matrix entries, but rather a pointer, or a *handle* to a previously created sparse matrix object. The result is a more general, portable code that can be run under different implementations of the Sparse BLAS that might be optimized for a special machine, a given matrix, or an application. The *internal* representation of the matrix data within the sparse matrix object is hidden from the user who accesses and manipulates the data through the available kernels.

The standard defines the following procedure for the use of the Sparse BLAS. First, the given matrix data has to be passed to an initialization routine that creates a handle referencing the matrix. Afterwards, the user can call the necessary Sparse BLAS routines with the handle as a means to reference the data. The implementation chooses the data-dependent algorithms internally, without the user being involved. When the matrix is no longer needed the matrix handle can be released and a cleanup routine is called to free any internal storage resources associated with that handle.

In the following sections, we describe in more detail each step of the above procedure and comment on our implementation [57] in Fortran 95. First, we give an overview of the Sparse BLAS functionalities in Section 2.2. In Section 2.3, we discuss how we organize, create, and use the data structures in Fortran for the sparse data. In Section 2.4 we discuss by the corresponding Fortran 95 interfaces the Sparse BLAS operations for the different Levels 1, 2, and 3. We illustrate the features of the Sparse BLAS by sample programs in Section 2.5. Finally we discuss the availability of our software in Section 2.6.

2.2 The Sparse BLAS functionalities

In this section, we briefly review the functionalities provided by the Sparse BLAS so that we can reference them in later sections where they are described further together with their implementation. For a complete specification, we refer to the standard [1].

Our notation is as follows. For the Level 1 BLAS, r and α are scalars, x is a compressed sparse vector and y a dense vector. Furthermore, the symbol $y|_x$ refers to the entries of y that have the same indices as the stored nonzero components of the sparse vector x. For the Level 2 and 3 BLAS, both x and y represent dense vectors, and α is a scalar. Moreover, A represents a general sparse matrix, T a sparse triangular matrix, and B and C are dense matrices.

2.2.1 Level 1 Sparse BLAS functionalities

The Level 1 Sparse BLAS covers basic operations on sparse and dense vectors, see Table 2.1. The functionalities provided are a sparse dot product (`USDOT`), a sparse vector update (`USAXPY`), sparse gather operations (`USGA`, `USGZ`), and a sparse scatter function (`USSC`). Here, `US` stands for 'Unstructured Sparse' data.

`USDOT`	sparse dot product	$r \leftarrow x^T y$, $r \leftarrow x^H y$
`USAXPY`	sparse vector update	$y \leftarrow \alpha x + y$
`USGA`	sparse gather	$x \leftarrow y\|_x$
`USGZ`	sparse gather and zero	$x \leftarrow y\|_x; y\|_x \leftarrow 0$
`USSC`	sparse scatter	$y\|_x \leftarrow x$

Table 2.1: Level 1 Sparse BLAS: sparse vector operations.

2.2.2 Level 2 Sparse BLAS functionalities

Table 2.2 lists the Level 2 operations on sparse matrices and a dense vector. The functionalities available are matrix-vector multiplication with a sparse matrix A or its transpose (`USMV`) and the solution of sparse triangular systems (`USSV`).

`USMV`	sparse matrix-vector multiplication	$y \leftarrow \alpha A x + y$ $y \leftarrow \alpha A^T x + y$ $y \leftarrow \alpha A^H x + y$
`USSV`	sparse triangular solution	$x \leftarrow \alpha T^{-1} x$ $x \leftarrow \alpha T^{-T} x$ $x \leftarrow \alpha T^{-H} x$

Table 2.2: Level 2 Sparse BLAS: sparse matrix-vector operations.

2.2.3 Level 3 Sparse BLAS functionalities

The Level 3 Sparse BLAS provides kernels for sparse matrices operating on dense matrices, those are listed in Table 2.3. The functionalities are similar to those of Level 2 and consist of matrix-matrix multiplication (`USMM`) and solution of sparse triangular systems with several right-hand sides (`USSM`).

2.2.4 Routines for the creation of sparse matrices

The routines for the creation of a sparse matrix representation and its associated handle are listed in Table 2.4.

The Sparse BLAS can deal with general sparse matrices and with sparse block matrices with a fixed or variable block size. After the creation of the corresponding handle with `USCR_BEGIN`, `USCR_BLOCK_BEGIN`,or `USCR_VARIABLE_BLOCK_BEGIN`,

USMM	sparse matrix-matrix multiplication	$C \leftarrow \alpha AB + C$ $C \leftarrow \alpha A^T B + C$ $C \leftarrow \alpha A^H B + C$
USSM	sparse triangular solution	$B \leftarrow \alpha T^{-1} B$ $B \leftarrow \alpha T^{-T} B$ $B \leftarrow \alpha T^{-H} B$

Table 2.3: Level 3 Sparse BLAS: sparse matrix-matrix operations.

USCR_BEGIN	begin point-entry construction
USCR_BLOCK_BEGIN	begin block-entry construction
USCR_VARIABLE_BLOCK_BEGIN	begin variable block-entry construction
USCR_INSERT_ENTRY	add point-entry
USCR_INSERT_ENTRIES	add list of point-entries
USCR_INSERT_COL	add a compressed column
USCR_INSERT_ROW	add a compressed row
USCR_INSERT_CLIQUE	add a dense matrix clique
USCR_INSERT_BLOCK	add a block entry
USCR_END	end construction
USSP	set matrix property
USGP	get/test for matrix property
USDS	release matrix handle

Table 2.4: Sparse BLAS: operations for the handling of sparse matrices.

the entries must be input using the appropriate insertion routines. A single point entry can be added by using USCR_INSERT_ENTRY, a list of multiple entries by USCR_INSERT_ENTRIES. For insertion of a list of row or column entries, one uses USCR_INSERT_ROW and USCR_INSERT_COL, respectively. Cliques are dense submatrices that arise for example in finite-element computations, these can be inserted by USCR_INSERT_CLIQUE. For a block matrix, we also can use USCR_INSERT_BLOCK to add a block entry.

Furthermore, we can optionally specify via USSP various properties of the matrix in order to assist possible optimization of storage and computation. As an example, we mention blas_lower_symmetric which indicates that the matrix is symmetric and only the lower half of the entries is given during construction. Calls to USSP should be made after a call to the BEGIN routine but before the first call to an INSERT routine for the same handle. A complementary routine, USGP, can be used to obtain information about the properties of a sparse matrix. Table 2.5 lists and explains the properties that can be associated with a sparse matrix representation. The construction is finished by calling USCR_END.

`blas_non_unit_diag`	nonzero diagonal entries are stored (Default)
`blas_unit_diag`	diagonal entries are not stored and assumed to be 1.0
`blas_no_repeated_indices`	indices are unique (Default)
`blas_repeated_indices`	nonzero values of repeated indices are summed
`blas_lower_symmetric`	only lower half of symmetric matrix is specified by user.
`blas_upper_symmetric`	only upper half of symmetric matrix is specified by user.
`blas_lower_hermitian`	only lower half of Hermitian matrix is specified by user.
`blas_upper_hermitian`	only upper half of Hermitian matrix is specified by user.
`blas_lower_triangular`	sparse matrix is lower triangular
`blas_upper_triangular`	sparse matrix is upper triangular
`blas_zero_base`	indices of inserted items are 0-based (Default for C)
`blas_one_base`	indices of inserted items are 1-based (Default for Fortran)
`blas_rowmajor`	dense blocks stored in row major order (C-default)
`blas_colmajor`	dense blocks stored in column major order (Fortran-default)
`blas_irregular`	general unstructured matrix
`blas_regular`	structured matrix
`blas_block_irregular`	unstructured matrix best represented by blocks
`blas_block_regular`	structured matrix best represented by blocks
`blas_unassembled`	matrix is best represented by cliques

Table 2.5: Matrix properties of Sparse BLAS matrices.

2.2.5 Remarks on the Sparse BLAS functionalities

Compared to the dense BLAS, the Sparse BLAS provides much less functionality. In particular, the Level 2 and 3 kernels exclusively support the product of a sparse matrix with a dense vector or matrix. Operations on two sparse structures are not supported. The reason given in [50] is the complexity of potentially mixing different representations of the two sparse structures, which is believed too complicated for efficient low-level kernels.

Arguably, there are many cases where such operations can be avoided and where it is computationally more efficient to do so: for example, the product ABx of two sparse matrices A and B and the dense vector x which can be computed as $A*(B*x)$ rather than $(A*B)*x$. In particular, this applies to the development and use of iterative solvers and eigenvalue methods, two fields of primary interest for the Sparse BLAS [50].

However, there exist numeric libraries that provide routines for use in situations where the multiplication of two sparse matrices is requested. Examples are given by SMMP [18] and SPARSKIT [120].

2.3 Representation of sparse matrices and vectors

The data structures for sparse matrices are often much more complicated than the ones for sparse vectors and can depend on the underlying application, the machine architecture, and other factors. For this reason, the Sparse BLAS standard deliberately avoids the prescription of a data structure for storing the matrix entries and allows the implementor to choose for his purposes the best implementation of the interfaces. In this section, we first give a short presentation of the data structures for sparse vectors and afterwards describe in detail the sparse matrix data structures that are used in the reference implementation [57].

2.3.1 The representation of sparse vectors

Sparse vectors are exclusively used in Level 1 operations. Their storage is much less complicated than that for sparse matrices and greatly facilitates the implementation of the Level 1 routines.

Generally, only the nonzero entries of a sparse vector x will be stored which leads to a representation of x by a pair of one-dimensional arrays, one for the real or complex nonzero entries, and the other one for their indices. A major motivation for prescribing this data structure in the standard is the observation that the representation by two arrays is one of the most commonly used, straightforward to implement, and avoiding possible overhead of derived data types or structures in Fortran or C, respectively [50].

For example, the sparse vector

$$x = (1.0, 0, 3.0, 4.0, 0)^T$$

can be represented as

$$\begin{array}{rcl} \mathit{VAL} & = & (1.0, 3.0, 4.0), \\ \mathit{INDX} & = & (1, 3, 4). \end{array} \tag{2.1}$$

In contrast to sparse matrices, the standard does not allow repeated indices in sparse vectors. Furthermore, the internal ordering of the vector elements in the two arrays is not specified, and multiple equivalent storage representations for the same sparse vector are valid. The *index base* of the sparse vector, that is whether one starts entry numbering at zero or one, can be chosen. A zero index base is commonly used in the C programming language, whereas Fortran arrays usually start with one. Thus, the representation in the above example corresponds to the Fortran convention of starting with index one for the first vector entry.

2.3.2 Internal data structures for sparse matrix data

The Sparse BLAS standard advocates the use of handles to refer to a sparse matrix data structure that has been created. The user inputs the matrix entries which are copied into internal storage. Then the matrix can be manipulated through the various kernels that access the corresponding data through its handle. In practice, the storage of the matrix entries depends on the nonzero structure of the matrix, and the user can aid matrix storage by specifying additional matrix properties, for example symmetry.

In this section, we discuss the internal data structures and manipulation related to the creation of a matrix handle that will be used to represent the sparse matrix in the later calls to the Sparse BLAS reference implementation [57]. From the implementor's point of view, the choice of the internal data structures is perhaps the most important part of the implementation as it will influence the design, implementation, and performance of all subsequent operations.

Conceptually, the Sparse BLAS distinguishes between three different types of sparse matrices; ordinary sparse matrices consisting of a set of single point entries, sparse matrices with a regular block structure, and sparse matrices with a variable block structure. These are discussed in the following.

1. Point entry matrices.
   The entries of this type of matrix are simple scalar values. The sparsity structure describes the layout of these entries in the matrix. An example is given in (2.2).

2. Block entry matrices with constant block size.
   Entries in these matrices are dense matrices themselves, such that all entries have the same row and column dimension. Block entry matrices can be characterized by any two of the following three different types of dimensions. The first one is the number of block rows and columns, the second one is the number of row and columns in the block entries (which is the same for each entry), and third one is the number of equations and variables, counted as standard row and column dimensions, respectively. We give an example in (2.3).

3. Block entry matrices with varying block sizes.
   Entries in these matrices are also dense matrices, but block sizes may vary from block row to block row. The blocks on the matrix diagonal are assumed to be square. One example for a variable block matrix is given in (2.4).

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 \\ 0 & 0 & 23 & 24 & 0 \\ 31 & 32 & 33 & 34 & 0 \\ 0 & 42 & 0 & 44 & 0 \\ 51 & 52 & 0 & 0 & 55 \end{pmatrix} \qquad (2.2)$$

$$B = \left(\begin{array}{cc|cc|cc} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ \hline 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \\ \hline 51 & 52 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 \end{array}\right), \tag{2.3}$$

$$C = \left(\begin{array}{rr|rrr|r|rrr|rr} 4 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 \\ 1 & 5 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\ \hline 0 & 0 & 6 & 1 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 9 & 3 & 0 & 0 & 0 & 0 & 0 \\ \hline 2 & 1 & 3 & 4 & 5 & 10 & 4 & 3 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 4 & 13 & 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 3 & 11 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 7 & 0 & 0 \\ \hline 8 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 & 3 \\ -2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 12 \end{array}\right). \tag{2.4}$$

Block entry matrices naturally occur in mathematical models where multiple degrees of freedom are associated with a single physical location. Examples are given by discretizations of problems in mechanical engineering where field equations couple several physical quantities. There, a single spatial mesh point is associated with a value of each of the quantities, whose internal coupling is described by the values in the associated matrix block. Variable block entry matrices occur when some degrees of freedom are tracked in one subregion and not another. In [50], the authors mention the example of chemical species concentration.

Furthermore, the distinction between the different matrix classes will allow a vendor to provide optimized algorithms for blocked data. First, one can save on integer storage by indexing matrix blocks instead of single entries. Second, computations with dense block entries can often be tuned to achieve higher performance. If the block size is large enough, dense BLAS kernels can be used that improve cache reuse and increase the ratio of floating point operations over memory access, resulting in a speedup of the code [41].

We now consider the process of constructing appropriate internal data structures for sparse matrices in our Fortran 95 reference implementation [57]. The actual creation of a sparse matrix with its handle consists of three or four basic steps and involves the routines listed in Table 2.4.

1. An internal data structure is initialized by calling one of the `USCR_BEGIN` routines.

2. (Optionally, the user can set matrix properties by `USSP`.)

3. The matrix data is passed to the internal data structure by one or more calls to `USCR_INSERT` routines.

4. The construction is completed by calling the `USCR_END` routine.

Intentionally, the standard [1] is written so that that a user need *not* know during the creation procedure how many matrix entries will be input. Instead, he can simply pass the data to the Sparse BLAS using the insert routines. Thus, for the kernel implementor it is not possible *a priori* to predict how much memory should be allocated. Consequently, the Sparse BLAS must use dynamic memory allocation and a dynamic data structure for the construction phase of the matrix. We use linked lists which are augmented dynamically when new matrix entries are added. The nodes of the lists contain the matrix entries together with their indices and a pointer to the next list node. In order to limit the size of the list, we keep the matrix entries grouped together in the same way and in the same order as the user passes them to the Sparse BLAS. If a single matrix entry is inserted, the list node contains only this single entry; if a row, a column, or a block is inserted, a different list is used and the list node contains all entries of the row, the column, or the block, respectively. In order to identify which kind of data is associated with each node, we use different pointers for row, column, and block data, respectively. However, using this structure for the Level 2 and Level 3 algorithms would imply a serious performance loss because of the amount of indirect addressing involved. At the call to `USCR_END`, all the data for the matrix is known. We now allocate one contiguous block of memory with appropriate size and copy the data into a static data structure for better performance. Additionally, it is possible to sort the matrix entries during this copying process so that the matrix data is held by rows or columns if this is beneficial for the performance of the Level 2 and Level 3 algorithms.

Before we describe the layout of the static internal storage schemes, we discuss alternative approaches to handling the dynamic memory allocation. Generally, it is important to limit memory fragmentation by allocating storage by blocks. We respect this by allocating the space for matrix rows, columns or blocks 'in one shot'. Another possibility is to preallocate a certain amount of memory and add the matrix entries as long as space is available; more space can be allocated when it is needed. This preallocation allows us to allocate contiguous memory blocks independently from the way a user inserts the matrix data.

We now present the details of our internal storage schemes:

1. Ordinary sparse matrices consisting of point entries are stored in coordinate format (COO), that is the entries of the matrix are stored along with their corresponding row and column indices. This requires the three arrays:

   - *VAL* - a real or complex array containing the entries of A, in any order.
   - *INDX* - an integer array containing the corresponding row indices of the entries of A.
   - *JNDX* - an integer array containing the corresponding column indices of the entries of A.

For example, a representation of the matrix A in equation (2.2) in COO format could be:

$$\begin{array}{rcllllllllllllllll} \mathit{VAL} & = & (& 11 & 51 & 31 & 32 & 34 & 52 & 13 & 23 & 33 & 14 & 24 & 42 & 55 & 44 &), \\ \mathit{INDX} & = & (& 1 & 5 & 3 & 3 & 3 & 5 & 1 & 2 & 3 & 1 & 2 & 4 & 5 & 4 &), \\ \mathit{JNDX} & = & (& 1 & 1 & 1 & 2 & 4 & 2 & 3 & 3 & 3 & 4 & 4 & 2 & 5 & 4 &). \end{array}$$

2. Systems with a regular block structure, where each entry is an LB-by-LB dense block, are stored internally in block coordinate format (BCO). Systems of this form typically arise, for example, when there are multiple unknowns per grid point of a discretized partial differential equation. Typically LB is a small number, less than twenty, determined by the number of quantities measured at each grid point, for example velocity, pressure, temperature, etc. The BCO format is defined similarly to the COO format. Entries are stored block-wise together with their block row and block column indices. This again requires three arrays.

   - *VAL* - a real or complex array containing the entries of the matrix, grouped and stored as dense blocks.
   - *BINDX* - an integer array containing the block row indices.
   - *BJNDX* - an integer array containing the block column indices.

   For example, a representation of the matrix B in equation (2.3) in BCO format could be:

$$\begin{array}{rclllllllllll} \mathit{VAL} & = & (& 11, & 21, & 12, & 22, & 15, & 25, & 16, & 26, & 33, & 43, \\ & & & 0, & 44, & 35, & 45, & 36, & 46, & 51, & 61, & 52, & 62 \;), \\ \mathit{BINDX} & = & (& 1, & 1, & 2, & 2, & 3 &), \\ \mathit{BJNDX} & = & (& 1, & 3, & 2, & 3, & 1 &). \end{array}$$

   Note that we choose the block internal storage to be in 'Fortran style', that is in column major order.

3. Systems with an irregular block structure are stored internally in the Variable Block Row format (VBR). VBR stores the nonzero block entries in each of the block rows as a sparse vector of dense matrices. The matrix is *not* assumed to have uniform block partitioning, that is, the blocks may vary in size. The VBR data structure is defined as follows. Consider an m-by-k sparse matrix along with a row partition $P_r = \{i_1, i_2, \ldots, i_{m_b+1}\}$ and column partition $P_c = \{j_1, j_2, \ldots, j_{k_b+1}\}$ such that $i_1 = j_1 = 1$, $i_{m_b+1} = m + 1$, $j_{k_b+1} = k + 1$, $i_p < i_{p+1}$ for $p = 1, \ldots, m_b$, and $j_q < j_{q+1}$ for $q = 1, \ldots, k_b$. The matrix C in equation (2.4) is an example of a block matrix where the blocks C_{ij} are defined according to the row and column partition shown. The block entries are stored block row by block row and each block entry is stored as a dense matrix in standard column major form. Six arrays are associated with this form of storage.

- *VAL* - a real or complex array containing the block entries of C. Each block entry is a dense rectangular matrix stored column by column.
- *INDPTR* - an integer array, the i-th entry of *INDPTR* points to the location in *VAL* of the (1,1) entry of the i-th block entry.
- *BINDX* - An integer array containing the block column indices of the nonzero blocks of C.
- *RPNTR* - An integer array of length m_b+1 containing the row partition P_r of C. $RPNTR(i)$ is set to the row index of the first row in the i-th block row.
- *CPNTR* - An integer array of length k_b+1 containing the column partition P_c of C. $CPNTR(j)$ is set to the column index of the first column in the j-th block column.
- *BPNTR* - An integer array of length m_b such that $BPNTR(i)$ points to the location in *BINDX* of the first nonzero block entry of block row i. If the i-th block row contains only zeros then set $BPNTR(i+1) = BPNTR(i)$.

For example, the matrix C in equation (2.4) is stored in VBR format as follows:

$$
\begin{array}{lcrrrrrrrrrrrrl}
VAL & = & (4, & 1, & 2, & 5, & 1, & 2, & -1, & 0, & 1, & -1, & 6, & 2, & \\
 & & -1, & 1, & 7, & 2, & 2, & 1, & 9, & 2, & 0, & 3, & 2, & 1, & \\
 & & 3, & 4, & 5, & 10, & 4, & 3, & 2, & 4, & 3, & 0, & 13, & 3, & \\
 & & 2, & 4, & 11, & 0, & 2, & 3, & 7, & 8, & -2, & 4, & 3, & & \\
 & & 25, & 8, & 3, & 12 &), & & & & & & & & \\
INDPTR & = & (1, & 5, & 7, & 11, & 20, & 23, & 25, & 28, & 29, & 32, & 35, & 44, & \\
 & & 48, & 52 &), & & & & & & & & & & \\
BINDX & = & (1, & 3, & 5, & 2, & 3, & 1, & 2, & 3, & 4, & 3, & 4, & 1, & \\
 & & 5 &), & & & & & & & & & & & \\
RPNTR & = & (1, & 3, & 6, & 7, & 10, & 12 &), & & & & & & \\
CPNTR & = & (1, & 3, & 6, & 7, & 10, & 12 &), & & & & & & \\
BPNTR & = & (1, & 4, & 6, & 10, & 12, & 15 &), & & & & & &
\end{array}
$$

We emphasize that our choice of the internal storage schemes is only one among several possibilities. The coordinate representation is very simple and is, for example, used as the basis for the *Matrix Market* sparse matrix storage format. However, the drawback of both the COO and the BCO storage format is that the matrix entries are not necessarily ordered and can degrade the efficiency of the Level 2 and Level 3 algorithms. Alternative matrix formats include the storage of the matrix entries by compressed columns or rows. Specifically, the Compressed Sparse Column (CSC) storage scheme is used for the matrices of the *Harwell-Boeing* collection [48] and forms also the basis of the *Rutherford-Boeing* format [49]. It is up to the vendor to choose the most appropriate representation.

We now present the fundamental datatype that accommodates all the data belonging to a sparse matrix. Its design is derived from [53]. When `USCR_END` is called, an instantiation of this datatype is created that will then be referenced by its handle in the calls to the Level 2 and Level 3 routines.

```
TYPE DSPMAT
   INTEGER :: M,K
   CHARACTER*5 :: FIDA
   CHARACTER*11 :: DESCRA
   INTEGER, DIMENSION(10) :: INFOA
   REAL(KIND=DP), POINTER, DIMENSION(:) :: VALUES
   INTEGER, POINTER, DIMENSION(:) :: IA1,IA2,PB,PE,BP1,BP2
END TYPE DSPMAT
```

(This is the datatype for a matrix with real entries in double precision. The other datatype formats are analogous. Here, the Fortran KIND parameters `sp` and `dp` specifying single and double precision are selected as `SELECTED_REAL_KIND(6,37)` and `SELECTED_REAL_KIND(15,307)`, respectively.)

Since the meaning of most of the components is already obvious from the above discussion of the internal storage formats, we give only short general remarks on them.

- The integers M and K represent the dimensions of the sparse matrix.

- `FIDA` holds a string representation of the matrix format, for example, 'COO'.

- `DESCRA` stores possible matrix properties such as symmetry.

- `INFOA` holds complementary information on the matrix such as the number of nonzero entries.

- The array `VALUES` keeps the values of the matrix entries. The way in which these entries are stored can be deduced from the following character and integer arrays.

- The arrays `IA1,IA2,PB,PE,BP1,BP2` are used to provide the necessary information on the sparsity structure of the matrix. The pointer arrays `PB,PE,BP1,BP2` are only used for block matrices. Note that we use generic array names, since their use depends on the matrix format. For example, in COO format, the arrays `IA1` and `IA2` represent $INDX$ and $JNDX$, while in VBR format, they represent $BINDX$ and $INDPTR$.

We decided to group the administration of all handle-matrix pairs according to their floating-point data type, that is, we keep a separate list of all valid matrix handles for each of the five floating-point data types supported by the Sparse BLAS.

2.4 Sparse BLAS operations on sparse matrices and vectors

In this section, we discuss in more detail the Sparse BLAS interface for Levels 1, 2, and 3 and remark on the kernel design in the Fortran 95 reference implementation [57]. As discussed in Section 2.3.1, the Sparse BLAS explicitly prescribes a data structure for the representation of sparse vectors. The choice of a single vector representation makes the implementation of the Level 1 kernels of the Sparse BLAS much less complex than one of the Level 2 and 3 routines for sparse matrices. In the Sparse BLAS Levels 2 and Level 3, the matrix is referenced by its handle referencing the internal matrix presentation that has been constructed before. The discussion in Section 2.3.2 on the different internal storage schemes shows that every Level 2 and Level 3 routine must be implemented for each scheme. Hidden from the user who uses the matrix handle in a generic subroutine call, the software chooses the appropriate routine according to the type of data.

2.4.1 Sparse vector operations

The storage of sparse vectors is much less complicated than that for sparse matrices and greatly facilitates the implementation of the Level 1 routines.

The Sparse BLAS Level 1 functionalities are described in Section 2.2.1, and the interface implementation is similar to the one used for the Level 1 kernels of the dense BLAS [1, 39, 40] for which a reference implementation is available from Netlib [2].

In our implementation [57], we generally do not assume that the entries of sparse vectors are ordered. By ordering and grouping the sparse vector according to its indices, it can be ensured that the dense vector involved in the Level 1 operations is accessed in blocks and cache reuse is enhanced. However, this is not prescribed by the Sparse BLAS standard [1].

One peculiarity of the Level 1 routines, in contrast to the sparse matrix operations of Level 2 and Level 3, is that the sparse vector operations do not return an error flag. Level 2 and Level 3 routines have to provide some checking of the input arguments, for example matching matrix dimensions, and can detect at least some of these errors and signal them to the user by setting an error flag. Because of the simplicity, the representation of sparse vectors is left to the user who is thus responsible for ensuring the correctness of the data. Furthermore, the overhead for checking in the Level 2 and Level 3 operations is less important because of their greater granularity.

2.4.2 Product of a sparse matrix with one or many dense vectors

In this section, we discuss the interface of the multiplication of a sparse matrix with a dense vector or a dense matrix. We concentrate on the matrix-vector multiplication, since in our code, the multiplication of a sparse matrix and a dense matrix is performed columnwise. Thus, we discuss the realization of

$$y \leftarrow \alpha A x \quad + \quad y,$$
and
$$y \leftarrow \alpha A^T x \quad + \quad y.$$

Compared to the corresponding matrix-vector multiplication kernel in the dense BLAS [2], the interface of the Sparse BLAS is lacking a scalar β for the scaling of the vector y. In [50], the authors argue that such a scaling operation should be performed by using the scaling available from the dense BLAS.

In our reference implementation, we use the generic functions of Fortran 95 extensively, allowing the software to choose the appropriate kernel according to the type of data.

We provide the following interface for the support of the different types

```
interface usmv
   module procedure susmv
   module procedure dusmv
   module procedure cusmv
   module procedure zusmv
   module procedure iusmv
end interface
```

with an implementation for matrices in double precision as follows:

```
subroutine dusmv(a, x, y, ierr, transa, alpha)
integer,  intent(in) :: a
real(kind=dp), dimension(:), intent(in) :: x
real(kind=dp), dimension(:), intent(inout) :: y
integer, intent(out) :: ierr
integer, intent(in),  optional :: transa
real(kind=dp), intent(in), optional :: alpha
```

where

- a denotes the matrix handle.
- x, y denote the dense vectors.
- $ierr$ is used as an error flag.
- $transa$ allows the optional use of the transposed sparse matrix.
- $alpha$ is an optional scalar factor.

The error flag encoding used in the Sparse BLAS is consistent with the error-handling framework described in Section 1.8 of the BLAS Standard [1]. Return codes are assigned to signify the success or failure of the operation. Typically, a Sparse BLAS routine returns a value of 0 (zero) if its operation completed successfully.

In the case of 'COO' sparse matrix storage, we perform the multiplication entry by entry, whereas for both regular block matrices in 'BCO' and irregular block matrices in 'VBR' format, we perform a dense matrix-vector multiplication with the subblocks.

We remark that, even if we perform the multiplication of a sparse matrix and a dense matrix column by column, there can be more efficient ways of doing this. Depending on the size of the dense matrix, a vendor could use blocking also on the dense matrix to gain performance.

2.4.3 Solution of a sparse triangular system with one or many dense right-hand sides

In this section, we show the interface of the solution of a sparse triangular system with a dense vector or a dense matrix. As in Section 2.4.2, we focus on the case of a single right-hand side:

$$x \leftarrow \alpha T^{-1} x,$$

and

$$x \leftarrow \alpha T^{-T} x.$$

Similarly to the multiplication routines, we provide a generic interface for the support of the different types in a similar way to that discussed for `usmv` in the previous section. The implementation of the different floating-point data types, for example `dussv`, is given by the following header:

```
subroutine dussv(a,x,ierr,transa,alpha)
integer, intent(in) :: a
real(kind=dp), intent(inout) :: x(:)
integer, intent(out) :: ierr
integer, intent(in), optional :: transa
real(kind=dp), intent(in), optional :: alpha
```

where the parameters are defined as for `usmv`, see Section 2.4.2.

For block matrices in either 'BCO' or 'VBR' format, the triangular solution is blocked and uses dense matrix kernels for matrix-vector multiplication and triangular solution on the subblocks.

In the case of a simultaneous triangular solution for more than one right-hand side, we have chosen internally to perform the solution step separately on each of them. However, the remark given at the end of Section 2.4.2 applies here, too. Blocking should be applied to the right-hand side if the matrix is big enough and offers enough potential for efficient dense matrix kernels.

2.4.4 Releasing matrix handles

In the Sparse BLAS, matrix handles and the corresponding internal data structures are created dynamically. Thus, as an implementor, we have to be careful with our memory management. In particular, we will want to return allocated memory to the system when we do not need it any more. The Sparse BLAS provides a routine for releasing a created matrix handle and freeing all associated memory.

The Fortran 95 binding of the handle release routine is:

```
subroutine usds(a,ierr)
integer, intent(in) :: a
integer, intent(out) :: ierr
```

Here, a denotes the matrix handle to be released and $ierr$ a variable to signal possible internal errors occurring on the attempt to release the handle.

We have already remarked in Section 2.3.2 that we use different internal data structures for the handle initialization and the Level 2 and Level 3 routines. The linked lists used in the handle initialization procedure can already be deallocated when `USCR_END` is called. The call to `USDS` will then result in a deallocation of the memory associated with the fundamental internal datatype shown at the end of Section 2.3.2.

When a matrix handle is released, one can either re-use the handle, that is, its `integer` value, for the next matrix that will be created, or prevent it from being used again. We decided to assign a new handle to each created matrix and not to re-use the released handles. This ensures that matrices are not confused by accident, as no matrix handle can represent more than one matrix simultaneously in the context of the program.

2.4.5 Some remarks on using Fortran 95

Fortran is widely recognized as very suitable for numerical computation. Fortran 95 includes Fortran 77 as a subset but additionally includes some useful features of other modern programming languages.

In our reference implementation [57], we benefit from the following features of Fortran 95:

1. Modules allow the code to be structured by grouping together related data and algorithms. An example is given by the Sparse BLAS module itself as it is used by the test programs described in Section 2.5.

2. Generic interfaces as shown in Sections 2.4.2 and 2.4.3 allow the *same* subroutine call to be used for each of the 5 floating-point data types supported.

3. Dynamic memory allocation allows us to create the linked lists for matrix entry management, as described in Sections 2.3.2 and 2.4.4.

4. Vector operation facilities instead of loops are employed wherever possible in the Level 2 and Level 3 algorithms from Sections 2.4.2 and 2.4.3, in particular for block matrix algorithms.

5. Numerical precision is consistently defined via the KIND parameters `sp` and `dp` for single and double precision, respectively. These are set in the module `blas_sparse`. We gave an example in the definition of the datatype `DSPMAT` at the end of Section 2.3.2.

2.5 Sample programs

We now demonstrate use of the Sparse BLAS by two examples. The sample program in Section 2.5.1 demonstrates the use of all Level 2 and Level 3 kernels by a triangular matrix. In Section 2.5.2, we present the power method for eigenvalue calculations as an application.

2.5.1 A sample program

In this section, we show how to use the Sparse BLAS by the example of a sparse triangular matrix T. We illustrate all steps from the creation of the matrix handle for the sample matrix T and the calls to Level 2 and Level 3 routines up to the release of the handle.

It is worth mentioning that the whole Sparse BLAS is available as a Fortran 95 module which has to be included by the statement *use blas_sparse* as shown in the fourth line of the source code. This module contains all Sparse BLAS routines and predefined named constants like the matrix properties mentioned in Section 2.2.4.

```
 program test
!
!----------------- Use the Sparse BLAS module ----------------------
use blas_sparse
!
!----------------- The test matrix data   -------------------------
!
!    / 1  1  1  1  1\
!    |    1  1  1  1|
! T= |       1  1  1|
!    |          1   |
!    \             1/
!
implicit none
real(kind=dp),dimension(14):: T_VAL=1.0_dp
integer,dimension(14):: T_indx=(/1,1,2,1,2,3,1,2,3,4,1,2,3,5/)
integer,dimension(14):: T_jndx=(/1,2,2,3,3,3,4,4,4,4,5,5,5,5/)
```

```
integer,parameter:: T_m=5, T_n=5, T_nz=14
real(kind=dp):: Tx(5) =(/15.0_dp,14.0_dp,12.0_dp,4.0_dp,5.0_dp/)
!
!----------------- Declaration of variables  -----------------------
real(kind=dp),dimension(:),allocatable:: x, y, z
real(kind=dp),dimension(:,:),allocatable:: dense_B,dense_C,dense_D
integer:: i,prpty,a,ierr
!
      open(UNIT=5,FILE='output',STATUS='new')
!
!----------------- Begin point entry construction  -----------------
      call duscr_begin(T_m, T_n, a, istat)
!
!----------------- Insert all entries -----------------------------
      call uscr_insert_entries(a, T_VAL, T_indx, T_jndx, istat)
!
!----------------- Set matrix properties ---------------------------
      prpty = blas_upper_triangular + blas_one_base
      call ussp(a, prpty,istat)
!
!----------------- End of construction  ----------------------------
      call uscr_end(a, istat)
!
      allocate(x(5),y(5),z(5),&
                  dense_B(size(y),3),dense_C(size(x),3),&
                  dense_D(size(x),3),STAT=ierr)
      if(ierr.ne.0) then
         write(UNIT=5, FMT='(A)') 'Allocation error'
         close(UNIT=5)
         stop
      endif
      do i=1, size(x)
         x(i) = dble(i)
      end do
      y=0.
      z = x
      do i = 1, 3
         dense_B(:, i) = x
         dense_C(:, i) = 0.
         dense_D(:, i) = Tx
      end do
!
!----------------- Matrix-Vector product  --------------------------
      write(UNIT=5, FMT='(A)') '* Test of MV multiply *'
      call usmv(a, x, y, istat)
      write(UNIT=5, FMT='(A)') 'Error : '
      write(UNIT=5, FMT='(D12.5)') maxval(abs(y-Tx))
!
!----------------- Matrix-Matrix product ---------------------------
      write(UNIT=5, FMT='(A)') '* Test of MM multiply *'
      call usmm(a, dense_B, dense_C, istat)
      write(UNIT=5, FMT='(A)') 'Error: '
      write(UNIT=5, FMT='(D12.5)') maxval(abs(dense_C-dense_D))
!
!----------------- Triangular Vector solve -------------------------
      write(UNIT=5, FMT='(A)') '* Test of tri. vec. solver *'
      call ussv(a, y, istat)
      write(UNIT=5, FMT='(A)') 'Error : '
      write(UNIT=5, FMT='(D12.5)') maxval(abs(y-x))
!
!----------------- Triangular Matrix solve -------------------------
      write(UNIT=5, FMT='(A)') '* Test of tri. mat. solver *'
      call ussm(a, dense_C, istat)
      write(UNIT=5, FMT='(A)') 'Error : '
      write(UNIT=5, FMT='(D12.5)') maxval(abs(dense_C-dense_B))
!
!------------------ Deallocation ----------------------------------
      deallocate(x,y,z,dense_B,dense_C,dense_D)
      call usds(a,istat)
      close(UNIT=5)
 end program test
```

2.5.2 An application: the power method

The power method is a well-known iteration for calculating the dominant eigenvalue and the corresponding eigenvector of a diagonizable matrix A [70].

Algorithm 2 *The power method.*

```
Given a random vector $z_1 \neq 0$
for $k = 1, \ldots, niter$ do
    $q_k = z_k / \|z_k\|_2$
    $z_{k+1} = A q_k$
    $\lambda_k = q_k^T A q_k = z_{k+1}^T q_k$
end for
```

In the program below, we show how the Sparse BLAS can be used to implement Algorithm 2. The essential part of the algorithm is the matrix-vector product USMV with a sparse matrix represented by its handle. At the end of the program, that is when the maximum number of iterations is reached, the eigenvalue approximation λ is printed.

```
 program power_method_test
!
!----------------- Use the Sparse BLAS module ----------------------
use blas_sparse
!----------------- Declaration of variables  -----------------------
      implicit none
      integer,parameter:: nmax= 4, nnz= 6
      integer:: a,i,n,niters,istat
      integer,dimension(:),allocatable:: indx,jndx
      real(kind=dp),dimension(:),allocatable:: val,q,work
      real(kind=dp):: lambda
      allocate (val(nnz),indx(nnz),jndx(nnz))
      allocate (q(nmax),work(nmax))
!
!----------------- The test matrix data   -------------------------
      val  = (/ 1.1_dp, 2.2_dp, 2.4_dp, 3.3_dp, 4.1_dp, 4.4_dp/)
      indx = (/  1,  2,  2,  3,  4,  4/)
      jndx = (/  1,  2,  4,  3,  1,  4/)

      n = nmax
!
!----------------- Begin point entry construction  -----------------
      call duscr_begin(n, n, a, istat)
!
!----------------- Insert all entries ------------------------------
      call uscr_insert_entries(a, val, indx, jndx, istat)
!
!----------------- End of construction  ----------------------------
      call uscr_end(a, istat)
!
!----------------- Call power method routine -----------------------
!     q      - eigenvector approximation.
!     lambda - eigenvalue approximation.
      niters = 100
      call POWER_METHOD(a, q, lambda, n, niters, work, istat)
      if (istat.ne.0) then
         write(UNIT=5,*) 'error in power_method = ',istat
      else
         write(UNIT=5,*) 'number of iterations = ',niters
         write(UNIT=5,*) 'approximate dominant eigenvalue = ',lambda
      endif
!
```

```
!------------------ Release matrix handle --------------------------
      call usds(a,istat)

    CONTAINS

      subroutine POWER_METHOD(a, q, lambda, n, niters, z, istat)
!
!----------------- Use the Sparse BLAS module ----------------------
      use blas_sparse
      implicit none
      real(kind=dp),dimension(:),intent(inout):: q(n),z(n)
      real(kind=dp),intent(out):: lambda
      integer,intent(in):: a,n,niters
      integer,intent(out):: istat
      integer:: i,iter,iseed
      real(kind=dp):: normz
      real(kind=sp):: y
      intrinsic random_number,dot_product
!
!-----------------  Fill z by random numbers -----------------------
      do i = 1,n
         call random_number(harvest=y)
         z(i)=dble(y)
      end do

      do iter = 1, niters
!
!-----------------  Compute 2-norm of z ----------------------------
         normz = sqrt(dot_product(z(1:n),z(1:n)))
!
!-----------------  Normalize z ------------------------------------
         if (normz.ne.0) z(1:n) = z(1:n)/normz
!
!-----------------  Copy z to q ------------------------------------
         q=z
!
!-----------------  Set z to 0 -------------------------------------
         z=0.0_dp
!
!----------------- Matrix-Vector product: Compute new z -----------
         call usmv(a, q, z, istat)
         if (istat.ne.0) return
!
!-----------------  Calculate new eigenvalue approximation lambda --
```

```
         lambda = dot_product(q,z)
      end do
      return
      end subroutine POWER_METHOD
!
 end program power_method_test
```

2.6 Conclusions and future work

In this chapter, we have described the design and part of the implementation of the Sparse BLAS. The Sparse BLAS standard is defined by the BLAS Technical forum [1] and provides interfaces for portable, high-performance kernels for operations with sparse vectors and matrices. Rationals of the Sparse BLAS design are described in [50], a reference model implementation in Fortran 95 has been developed in the framework of this thesis. A description of the implementation has been published in [57], and the software is available in the CALGO repository of ACM TOMS [58].

Our contributions are twofold: As our implementation has been developed in parallel to the design of the standard, it has influenced and shaped its design. Moreover, the standard avoids the discussion of data structures as well as language-dependent implementation issues. In this respect, we have proposed and described decisions taken in our software concerning these topics.

Our software is currently the only existing reference implementation. Both the C and the Fortran 77 reference implementation are still under development and user guides are planned [82]. The internal report [118] is currently the only existing reference. Vendors may supply optimized versions which exploit special features of high performance computing architectures. Suggestions for such optimizations have been given in the relevant sections of this chapter.

However, there is scope for more sophisticated optimizations that can be part of a future implementation. As an example, we point out the Sparsity project at the University of California in Berkeley which provides kernels for sparse matrix-vector multiplication that are automatically tuned for a given matrix on a given machine [94, 95, 96]. The major techniques which are investigated there consist of register and cache blocking as well as the careful design of the underlying sparse matrix data structures.

In the overall framework of the BLAS project, the design of the standard and the development of the reference implementation are only a first step. The success of the Sparse BLAS depends on future efforts of programmers, commercial library developers, and computer vendors to design optimized and efficient implementations.

III

Chapter 3

Task scheduling in a multifrontal sparse linear solver

The direct solution of sparse linear systems by Gaussian elimination is more robust than iterative solution methods, and is thus often the preferred approach when trying to solve an ill-conditioned system. However, the problem of designing an *efficient* sparse direct solver is very complicated, as it involves numerical issues such as the careful control of fill-in and pivoting as well as computer science related questions on how to exploit modern architectures with a multi-layer memory hierarchy or a parallel distributed environment.

The asynchronous distributed memory multifrontal solver MUMPS is an example of such a modern algorithm addressing all these issues. Specifically, it provides fill-reducing matrix orderings during the analysis and dynamic data structures to support threshold pivoting during the numerical factorization. Furthermore, it makes use of BLAS kernels and data blocking to enhance cache reuse, exploits parallelism arising from independent computations due to sparsity, and allows dynamic task scheduling during factorization in order to automatically adapt to a varying computer load. Additionally, it includes features such as automatic error analysis, iterative refinement, matrix scaling, support for matrix input in elemental format, and determination of the null space basis of a rank-deficient matrix.

MUMPS can solve a wide range of problems including symmetric and unsymmetric systems with real or complex entries by either an LU or an LDL^T factorization. It has initially been developed in the framework of the PARASOL Project and is now in the public domain.

In this chapter of the thesis, we are concerned with the question of designing a new efficient task scheduler for MUMPS that improves scalability on a large number of processors. In our approach, we determine, during the analysis of the matrix, candidate processes for the tasks that will be dynamically scheduled during the subsequent factorization. The new algorithm significantly improves the scalability of the solver on a large number of processors in terms of execution time and storage, as we show by experiments with matrices from regular grids and irregular ones from real-life applications.

3.1 Introduction to MUMPS

We consider the direct solution of sparse linear systems on distributed memory computers. Two state-of-the-art codes for this task, MUMPS and SuperLU, have been extensively studied and compared in [10]. Specifically, the authors show that on a large number of processors, the scalability of the multifrontal approach used by MUMPS [9, 10] with respect to computation time and use of memory could be improved. This observation is the starting point for this current work.

The solution of a linear system of equations using MUMPS consists of three phases. In the *analysis* phase, the matrix structure is analysed and a suitable ordering and data structures for an efficient factorization are produced. In the subsequent *factorization* phase, the numerical factorization is performed. The final *solution* phase computes the solution of the system by forward and backward substitutions using the factors that were just computed.

The numerical factorization is the most expensive of these three phases, and we now describe how parallelism is exploited in this phase. The task dependency graph of the multifrontal factorization is a tree, the so-called *assembly tree*. A node of this tree corresponds to the factorization of a dense submatrix, and an edge from one node to another describes the order in which the corresponding submatrices can be factorized. In particular, independent branches of the assembly tree can be factorized in parallel as the computations associated with one branch do not depend on those performed in the others. Furthermore, each node in the tree can itself be a source of parallelism. The ScaLAPACK library [29] provides an efficient parallel factorization of dense matrices and is used for the matrix associated with the root of the assembly tree. But MUMPS offers another possibility for exploiting parallelism for those nodes that are large enough. Such nodes can be assigned a master process during analysis that chooses, during numerical factorization, a set of slave processes to work on subblocks of the dense matrix. This *dynamic* decision about the slaves is based on the load of the other processors, only the less loaded ones are selected to participate as slaves.

In order to address the scalability issues, we have modified this task scheduling and the treatment of the assembly tree during analysis and factorization. We now give a brief description of these new modifications to Version 4.1 of MUMPS (to which we sometimes refer as the old code or the previous version of MUMPS).

The objective of the dynamic task scheduling is to balance the work load of the processors at run time. However, two major problems arise from offering too much freedom to the dynamic scheduling. In the previous version of MUMPS, a master process is free to choose its slaves among all available processes. Since this choice is taken dynamically during the factorization phase, we have to anticipate it by providing enough memory on every process for the corresponding computational tasks. Since typically not all processes are actually used as slaves (and, on a large number of processors, often only relatively few are needed), the prediction of the required workspace will be overestimated. Thus, the size of the problems that can be solved is reduced unnecessarily because of this difference between the prediction and the allocation of memory by the analysis phase and the memory actually used during the factorization. Secondly, decisions concerning a node should take into account global information on the assembly tree to localize communication. For example, by mapping independent subtrees to disjoint sets of processors so that all data movements related to a subtree are performed within the set, we can improve locality of communication and increase performance.

With the concept of *candidate processors*, it is possible to *guide* the dynamic task scheduling and to address these issues. The concept originates in an algorithm presented in [115, 117] and has also been used in the context of static task scheduling for sparse Cholesky factorization [81]. In this chapter of the thesis, we show how it also extends efficiently to dynamic scheduling. For each node that requires slaves to be chosen dynamically during the factorization, we introduce a limited set of processors from which the slaves can be selected. While the master previously chose slaves from among all less loaded processors, the freedom of the dynamic scheduling is reduced so that the slaves are only chosen from among the candidates. This allows us to exclude all non-candidates from the estimation of workspace during the analysis phase and leads to a more realistic prediction of the workspace needed. Furthermore, the candidate concept allows us to better structure the computation since we can explicitly restrict the choice of the slaves to a certain group of processors and enforce for example a 'subtree-to-subcube' mapping principle [66]. (Throughout this chapter, we assume that every processor has one single MPI process associated with it so that we can unambiguously identify a processor and a corresponding MPI process.)

We illustrate the benefits of the new approach by tests using a number of performance metrics including execution time, memory usage, communication volume, and scalability. Our results demonstrate significant improvements for all these metrics, in particular when performing the calculations on a large number of processors.

The rest of this chapter is organized as follows. In Section 3.2, we review briefly the general concepts of the multifrontal direct solution of sparse linear systems, most notably the assembly tree. We describe in Section 3.3 the possibilities for exploiting parallelism. Section 3.4 gives a short presentation of possible improvements to the assembly tree, and we then introduce, in Section 3.5, the concept of candidate processors. In Section 3.6, we give an overview of how the candidate concept fits into the scheduling algorithm and present the algorithmic details in Section 3.7. Section 3.8 gives an overview of the test problems used. The presentation of our experimental results begins with parameter studies and detailed investigations of the improved algorithms in Section 3.9. Afterwards, we present a systematic comparison of the previous with the new version of the code on regular grid problems and general matrices in Section 3.10. Finally, we discuss possible extensions of our algorithm in Section 3.11 and present our conclusions and a brief summary in Section 3.12.

3.2 Multifrontal solution of sparse linear systems

We consider the direct solution of large sparse systems of linear equations

$$Ax = b$$

on distributed memory parallel computers using multifrontal Gaussian elimination. For an unsymmetric matrix, we compute its LU factorization; if the matrix is symmetric, its LDL^T factorization is computed.

The multifrontal method was initially developed for indefinite sparse symmetric linear systems [54] and was then extended to unsymmetric matrices [55]. Because of numerical stability, pivoting is required in these cases in contrast to symmetric positive definite sparse systems where pivoting can be avoided. We are concerned with general unsymmetric and symmetric indefinite matrices in the following, for an overview of the multifrontal method for symmetric positive definite systems we refer to [47, 54, 106].

3.2.1 General principles of sparse Gaussian elimination

The multifrontal direct solution of sparse linear systems generally consists of three steps [47, 54]:

1. The *analysis phase* works mostly with the matrix structure and ignores numerical values. (However, numerical values might be used in a preprocessing step, for example to permute large matrix entries to the diagonal [51, 52, 99].) From a symbolic analysis of the matrix, it is possible to estimate the work and storage requirements, and to set up the work space for the subsequent factorization and solution phases. By reordering the matrix through a permutation of the rows and columns, we can usually reduce the memory and floating-point operations required during the subsequent steps, and possibly also improve their parallelism. (We remark that the ordering step may also include numerical values.)

2. The *factorization phase* computes the triangular factors L and U of the matrix A by Gaussian elimination, or the matrices L and D in the case of a symmetric A. The multifrontal approach is governed by the so-called *assembly tree* which describes the tasks and task dependencies of the algorithm. Each node of the assembly tree is associated with the factorization of a dense *frontal* matrix of A which can be performed once all the frontal matrices in the subtree below the node have been processed. This partial ordering can be exploited effectively by the parallel algorithm. The stability of Gaussian elimination depends on the choice of the pivots. A pivot should be chosen so that the growth in the magnitude of the factors is bounded, as this will limit the effects of roundoff error. However, for sparse matrices, we have not

only to consider numerical stability but also the fill-in. This occurs when the fundamental operation of Gaussian elimination

$$a_{ij}{}^{(k+1)} \quad = \quad a_{ij}{}^{(k)} \quad - \quad \frac{a_{ik}{}^{(k)} \times a_{kj}{}^{(k)}}{a_{kk}{}^{(k)}}$$

at step k of the factorization creates a nonzero entry $a_{ij}{}^{(k+1)}$ when $a_{ij}{}^{(k)}$ is zero. For this reason, partial pivoting with a threshold criterion (so called *threshold pivoting*) is often the method of choice because it gives some freedom to choose pivots to reduce fill-in [47].

3. The *solution phase* computes the solution of the linear system by using the factors determined during the previous phase. The solution process is again driven by the assembly tree. A subsequent error analysis can reveal the accuracy of the computed solution and iterative refinement can be used if required.

3.2.2 Tasks and task dependencies in the multifrontal factorization

In this section, we describe the tasks arising in the factorization phase of a multifrontal algorithm. Specifically, we investigate the work associated with the factorization of individual frontal matrices and the order in which these factorizations can be performed.

We start with the description of task dependencies in the multifrontal Cholesky factorization. While we will be concerned with the more general case of unsymmetric and symmetric indefinite matrices, the fundamental concept of the elimination tree [54, 105, 122] originates from the Cholesky factorization and can be described most clearly for the symmetric positive definite case.

We consider the computation of the Cholesky factor L of a sparse symmetric positive definite matrix of order n. We define an associated graph with vertices $v \in \{1, \dots, n\}$ representing the columns of L and a set of undirected edges $e = \{j, i_j\}, j = 1, \dots, n-1, i_j > j$ connecting the vertex of column j with that corresponding to its first off-diagonal nonzero i_j in L. Assuming that, in each column of L except the last, there exists at least one such nonzero off-diagonal entry, the graph is connected and is also acyclic since it consists of n vertices and $n-1$ edges. Hence, the defined graph is a tree which is commonly called the *elimination tree* of the sparse symmetric positive definite matrix A. The importance of the elimination tree stems from the fact that it represents the order in which the matrix can be factorized, that is, in which the unknowns from the underlying linear system of equations can be eliminated. Each pivot associated with a vertex (or *node*) of the elimination tree can only be eliminated after all the pivots associated with the subtree below the node are eliminated. For a dense matrix, the elimination tree is a chain and defines a complete ordering of the eliminations. However, for a general sparse matrix, the definition yields only a *partial* ordering which allows some freedom for the sequence in which pivots can be eliminated. For example, the leaf nodes of the tree can be processed in any order. We note that the construction of the elimination tree can be done symbolically (if we neglect coincidental numerical cancellation) and is usually done during the analysis phase of a direct solver. For information on the reordering of nodes to reduce the size of the work space for the solution process, we refer to [71, 104].

There are two main possibilities for extending the concept of the elimination tree to an unsymmetric matrix A. The first approach works with the symmetrized pattern of $A + A^T$ which may imply an increased demand on storage but which has the advantage of offering a unified concept for an integrated solver for both symmetric and unsymmetric matrices, for example MUMPS [9, 10]. The second approach considers dependency graphs separately for the L and the U factor, this concept of *elimination dags* was introduced in [68] and is used in SuperLU [35, 36] and UMFPACK/MA38 [33, 34]. In the following discussion, we focus on the first approach that is used by MUMPS.

One central concept of the multifrontal approach [54] is to group (or *amalgamate*) columns with the same sparsity structure to create bigger *supervariables* or *supernodes* [54, 107] in order to make use of efficient dense matrix kernels. We will discuss later the advantages and dangers of amalgamation in the context of a distributed memory multifrontal code. We mention here that it is common to relax the criterion for amalgamation and permit the creation of coarser supernodes with extra fill-in that, however, improve the performance of the factorization, see [14, 54]. The amalgamated elimination tree is called the *assembly tree*.

We illustrate these concepts by the following example. Figure 3.1 shows the nonzero pattern of a symmetric positive definite matrix A together with the pattern of its Cholesky factor L, the fill-in is indicated by $\bullet$. In Figure 3.2 we show the elimination tree of A and in Figure 3.3 the resulting assembly tree when we amalgamate the nodes 3, 5, and 6 together which form a dense submatrix in the filled matrix $L + L^T$. If the amalgamation was relaxed to allow the zero entry in the $(4,3)$ position of L to be stored, we could also amalgamate node 4 together with 3, 5, and 6.

$$\begin{bmatrix} \times & & & \times & & \\ & \times & & \times & \times & \\ & & \times & & \times & \times \\ \times & \times & & \times & & \times \\ & \times & \times & & \times & \\ & & \times & \times & & \times \end{bmatrix} \qquad \begin{bmatrix} \times & & & & & \\ & \times & & & & \\ & & \times & & & \\ \times & \times & & \times & & \\ & \times & \times & \bullet & \times & \\ & & \times & \times & \bullet & \times \end{bmatrix}$$

Figure 3.1: Matrix A and Cholesky factor L.

We have already emphasized the importance of the assembly tree as a data structure for driving the factorization and the solution phases of the multifrontal algorithm. We now investigate more closely the work associated with the factorization of the frontal matrix at an individual node of the assembly tree. Frontal

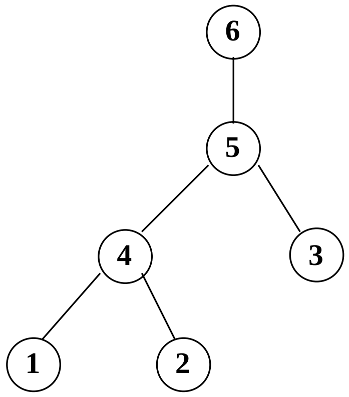

Figure 3.2: Elimination tree of matrix A from Figure 3.1.

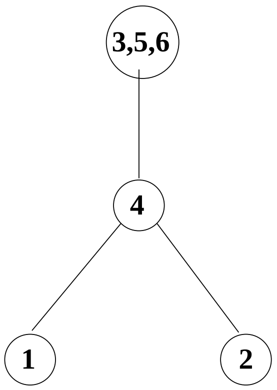

Figure 3.3: Assembly tree of matrix A from Figure 3.1.

matrices are always considered as dense matrices. We can make use of the efficient BLAS kernels and avoid indirect addressing, see for example [41]. Frontal matrices can be partitioned as shown in Figure 3.4.

$$\begin{array}{rc}
 & \begin{array}{cc} \text{fully summed columns} & \text{partially summed columns} \\ \downarrow & \downarrow \end{array} \\
\begin{array}{r} \text{fully summed rows} \rightarrow \\ \text{partially summed rows} \rightarrow \end{array} & \left[\begin{array}{cc} F_{11} & F_{12} \\ F_{21} & F_{22} \end{array}\right]
\end{array}$$

Figure 3.4: A frontal matrix.

Here, pivots can be chosen only from within the block of fully summed variables F_{11}. Once all eliminations have been performed, the Schur complement matrix $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is computed and used to update later rows and columns of the overall matrix which are associated with the parent nodes. We call this Schur complement matrix the *contribution block* of the node.

The notion of children nodes which send their contribution block to their parents leads to the following interpretation of the factorization process. When a node of the assembly tree is being processed, it assembles the contribution blocks from all its children nodes into its frontal matrix. Afterwards, the pivotal variables from the fully summed block are eliminated and the contribution block computed. The contribution block is then sent to the parent node to be assembled once all children of the parent (which are the siblings of the current node) have been processed. This is illustrated in Figure 3.5 for the matrix from Figure 3.1.

We remark that possibly some variables cannot be eliminated safely from a frontal matrix because of possible numerical instability. In this case, their elimination will be delayed until stable pivots can be found. The corresponding fully summed rows and columns are moved to the contribution block and are assembled at the parent node. The contribution block becomes larger than was predicted during the analysis phase, and the data structure used in the factorization needs to be

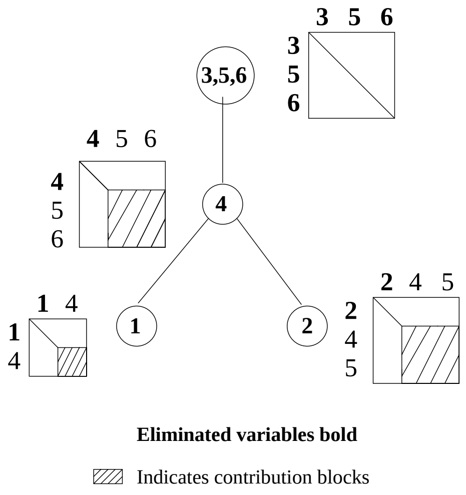

Figure 3.5: Illustration of the multifrontal factorization driven by an assembly tree.

modified dynamically. For this reason, it is common to provide some extra memory for the factorization in case delayed pivoting occurs.

3.3 Parallelism in the multifrontal factorization

In the following, we identify different sources of parallelism in the multifrontal factorization and describe how these are exploited in MUMPS [9].

3.3.1 The different types of parallelism

In Section 3.2.2, we mentioned that the tasks of multifrontal Gaussian elimination for sparse matrices are only *partially* ordered and that the task dependencies are represented by the assembly tree. A pair of nodes where neither is an ancestor of the other can be factorized independently from each other, in any order or in parallel. Consequently, independent branches of the assembly tree can be processed in parallel, and we refer to this as *tree parallelism* or *type 1 parallelism.*

A fundamental concept for the complete static mapping of assembly trees from model grid problems, the *subtree-to-subcube* mapping, was given by George, Liu, and Ng [66]. This algorithm was then generalized for problems with irregular sparsity structure and unbalanced trees to the *bin-pack* mapping scheme by Geist and Ng [63] and the *proportional* mapping approach by Pothen and Sun [115, 117]. We will describe these algorithms in Section 3.3.2.

It is obvious that in general, tree parallelism can be exploited more efficiently in the lower part of the assembly tree than near the root node. Experimental results presented in [8] showed a typical speedup obtained from tree parallelism of less than five on 32 processors. These results are related to the observation [7] that often more than 75% of the computations are performed in the top three levels of the assembly tree where tree parallelism is limited. For better scalability, additional parallelism is created from parallel blocked versions of the algorithms that handle the factorization of the frontal matrices.

The computation of the Schur complement of frontal matrices with a large enough contribution block can be performed in parallel using a Master-Slave computational model. The contribution block is partitioned and each part of it assigned to a slave. The master processor is responsible for the factorization of the block of fully summed variables and sends the triangular factors to the slave processors which then update their own share of the contribution block independently from each other and in parallel. We refer to this approach as *type 2 parallelism* and call the concerned nodes *type 2 nodes*.

Furthermore, the factorization of the dense root node can be treated in parallel with ScaLAPACK [29]. The root node is partitioned and distributed to the processors using a 2D block cyclic distribution. This is referred to as *type 3 parallelism*. Note that a 2D distribution could also be used for frontal matrices other than the root node, but this is not exploited in MUMPS.

3.3.2 Parallel task scheduling: main principles

From the point of view of scheduling, the different types of parallelism vary in their degree of difficulty. Apart from looking at each type of parallelism individually, it is also necessary to investigate their interaction. The main objectives of the scheduling approaches are to control the communication costs, and to balance the memory and computation among the processors. We describe in this section the techniques implemented in Version 4.1 of MUMPS which is described in [8, 9], and which has been extensively tested and compared with SuperLU [10] and WSMP [73, 72]. We also present the proportional mapping by Pothen and Sun [115, 117] from which we develop, in Section 3.5, our idea of the candidate-based scheduling that is used in the new version of MUMPS.

3.3.2.1 Geist-Ng mapping and layers in the assembly tree

We mentioned in Section 3.3.1 that in general, only the lower part of an assembly tree can be exploited efficiently for tree parallelism. Our previous scheduling approach consists therefore of two phases. At first, we find the lower part of the assembly tree where enough tree parallelism can be obtained. Afterwards we process the remaining upper part of the tree exploiting additionally type 2 and type 3 parallelism.

The mapping algorithm by Geist and Ng [63] allows us to find a layer in the assembly tree so that the subtrees rooted at the nodes of this layer can be mapped onto the processors for a good balance with respect to floating-point operations. Processor communication is avoided by mapping each subtree completely to a single designated processor. We call the constructed layer L_0. It marks the boundary between the lower part where scheduling exploits only tree parallelism (type 1), and the upper part where all three types of parallelism are used.

We consider the following top-down tree-processing approach [63]. We take as potential layer L_0 the root nodes (or the root node, for an irreducible matrix) of the assembly tree. We check whether the nodes can be mapped onto the processors so that the load on each processor is balanced up to a threshold. In general, this will not be the case, in particular if the number of processors used for the factorization is larger than the number of root nodes. We then modify the potential layer L_0 by replacing the node whose subtree has largest computational cost by its children. Again, we check if the mapping of the new layer is balanced up to the threshold, otherwise we repeat the previous substitution step for the node which has now the highest computational subtree costs, and so forth. The algorithm stops once the nodes in the potential layer L_0 allow a threshold-balanced mapping. Intuitively, we can think of the algorithm descending down in the assembly tree, as illustrated in Figure 3.6.

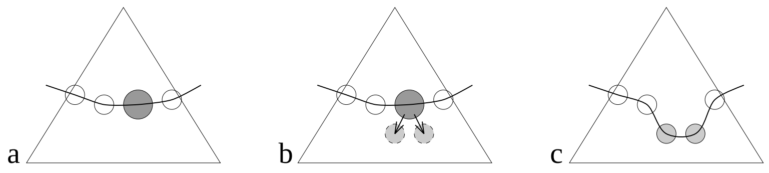

Figure 3.6: Geist-Ng algorithm for the construction of layer L_0 .

The constructed initial layer L_0 induces a layer partition of the upper part of the assembly tree. Before the frontal matrix belonging to a node of the tree can be processed, all contributions from the descendants of the node have to be gathered. This leads to the following recursive definition. Given a node in layer L_{i-1} , the parent of this node belongs to L_i if and only if all the children of this parent node belong to the layers $L_0, \ldots , L_{i-1}$. As the nodes in one layer can be only processed if all their children, belonging to the lower layers, have already been treated, the layer partition not only represents dependency but also concurrency of the multifrontal factorization. An example is shown in Figure 3.7.

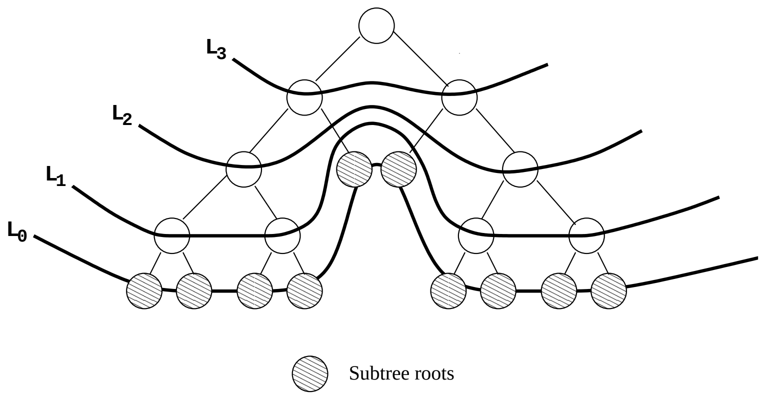

Figure 3.7: Layers in the assembly tree.

3.3.2.2 The proportional mapping of Pothen and Sun

The proportional mapping approach by Pothen and Sun [115, 117] represents an alternative approach to task scheduling in both regular and possibly irregular assembly trees. It consists of a recursive assignment of processors to subtrees according to their associated computational work.

The assembly tree is processed from top to bottom, starting with the root nodes. For each root node, we calculate the work associated with the factorization of all nodes in its subtree, and the available processors are distributed among the root nodes according to their weight. Each node thus gets its set of so-called *preferential* processors. The same partitioning is now repeated recursively. The processors that have been previously assigned to a node are now distributed among the children proportional to their weight given by the computational costs of their subtrees. The recursive partitioning stops once a subtree has only one processor assigned to it.

A main benefit of the proportional mapping is that communication is effectively localized among the processors assigned to a given subtree, with the partitioning guided from a *global* point of view taking account of the weight of subtrees. An illustration of the proportional mapping algorithm is given in Figure 3.8.

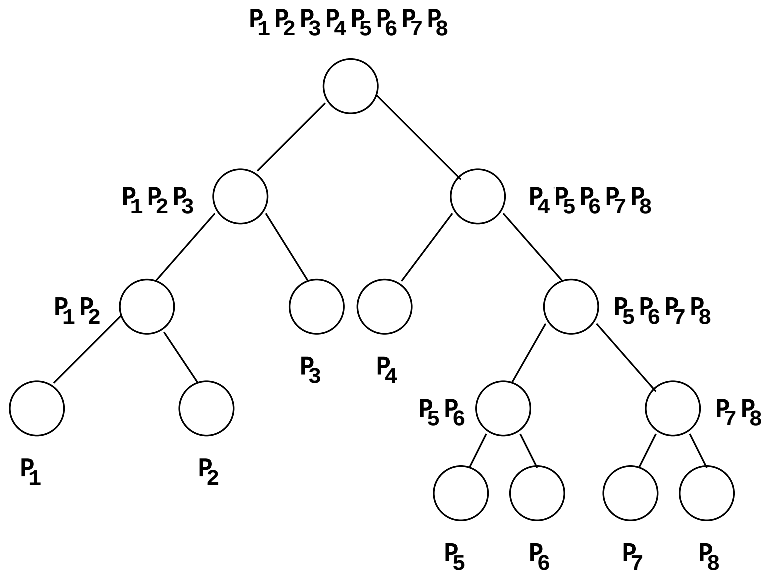

Figure 3.8: Proportional mapping of an assembly tree on eight processors.

While the proportional mapping approach has not been previously used in MUMPS, we mention it here because of its central importance for other sparse linear solvers like PaStiX [79, 80, 81] and because the concept of *candidate* processors for type 2 parallel nodes presented later on exploits this idea.

3.3.2.3 Dynamic task scheduling for type 2 parallelism

It is possible to extend the static mapping to the tasks arising in the Master-Slave computational model for the factorization of type 2 parallel nodes. However, the static mapping is performed during the analysis phase on the basis of *estimated* costs of computational work and communication. These estimates can be inaccurate if pivots have to be delayed for numerical reasons. For a better equilibration of the actual computational work at run time, both the number and the choice of the slaves of type 2 nodes are determined *dynamically* during factorization in the following way [8, 9]. When the master of a type 2 node receives the symbolic information on the structure of the contribution blocks of the children, the slaves for the factorization are selected based on their current work load, the least loaded processors being chosen. The master then informs the processes handling the children nodes which slaves are participating in the factorization of the node so that they can send the entries in their contribution blocks directly to the appropriate slaves.

The previous version of MUMPS exploits type 2 parallelism above layer L_0 as follows. If a node possesses a contribution block larger than a given threshold and the number of eliminated variables in its pivot block is large enough, then it will be declared a type 2 node and will be involved in the dynamic decision to schedule new activities. In the new version of MUMPS, we leave this concept principally unchanged; however, we restrict the freedom for the dynamic choice of the slaves. While, in the earlier algorithm, potentially every processor could be chosen as a slave during run time, in the new approach we restrict this selection to the candidates that have been chosen for a given node during the analysis phase. This is explained in detail in Section 3.5.2.

3.4 Modifications to the assembly tree

In this section, we present and illustrate the concepts of amalgamation and splitting as possible modifications to an assembly tree during the analysis phase in order to improve the subsequent factorization. In this context, we also describe delayed pivoting which constitutes an on-the-fly modification of the assembly tree during factorization.

The concepts of amalgamation and splitting were already exploited in Version 4.1 of MUMPS. However, the capabilities were relatively limited. Amalgamation was only performed near leaf nodes, and splitting was used only up to a certain distance from the root nodes.

In the new version of MUMPS, we have significantly increased the possibilities for splitting and amalgamation; we refer to Sections 3.6 and 3.7 for the details. The positive impact of these modifications on the performance of the algorithm is illustrated by the results in Sections 3.9.4 and 3.9.5.

3.4.1 The benefits and drawbacks of amalgamation

Amalgamation of a node with its parent is performed when the node has a large contribution block but only a few fully summed variables. The frontal matrices of parent and child are merged and treated together instead of being processed sequentially.

The immediate impact of node amalgamation is on the memory. If a large contribution block has to be kept on one processor and then sent to another one, the communication buffer must be large enough to hold the data. If the node is amalgamated, it effectively disappears from the tree, its stacking is avoided and the buffer can possibly be smaller. Amalgamated nodes may still have a large contribution block that could be a problem for the stack and buffer size. However, we can often exploit type 2 parallelism more efficiently for such nodes since the number of fully summed rows has been increased. In this case, the update of the contribution block is done in parallel on several slaves which eliminates the need for stacking the whole block on the master. Instead, each slave stacks its own, smaller part of the contribution block, and communication from one process to another will involve smaller amounts of data corresponding to only part of the contribution block.

Furthermore, it has been observed in [5] that even in a multifrontal factorization on a single processor, the assembly process runs at a much lower megaflop rate than the elimination process. Since, in a distributed environment, the assembly requires additional data movement from the master processor of the parent node with all processors working on the children of the node, the costs of assembly, and consequently the benefits of amalgamation, can be even higher.

However, we have to also consider possible negative effects of amalgamation. Its main drawback stems from the fact that it increases the front size of the parent node and creates extra fill-in and operations. While the total amount of data to be stacked and/or communicated is reduced, the memory for the storage of the matrix factors is increased. This is illustrated in Figure 3.9.

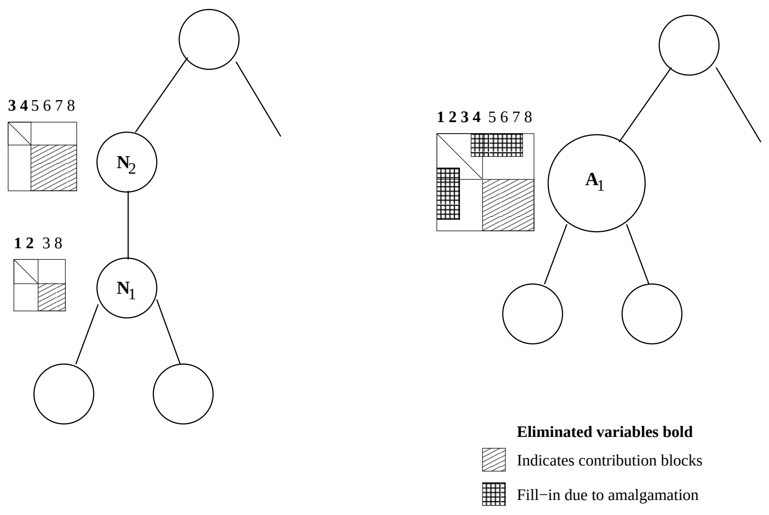
3 4 5 6 7 8
N$_2$
1 2 3 8
N$_1$
1 2 3 4 5 6 7 8
A$_1$
Eliminated variables bold
▨ Indicates contribution blocks
▦ Fill−in due to amalgamation

Figure 3.9: Amalgamation of two nodes N_1 and N_2 into a single node A_1.

3.4.2 The benefits and drawbacks of splitting

MUMPS performs the factorization of the pivot rows of a frontal matrix on a single processor. This can lead to performance problems, particularly if the frontal matrix has a large block of fully summed variables and only a relatively small contribution block. However, if the front size is big enough, it is possible to create artificial type 2 parallelism in the following way.

The factorization of the pivot block is divided into two parts performed sequentially one after the other. Instead of factorizing the pivot block at once and then updating the contribution block, a two-step process is invoked. The first part of the pivot block is factorized and the remaining part of the frontal matrix is updated, that is, the original contribution block *and* the second part of the pivot block. Afterwards, the second part of the pivot block is factorized and the update applied to the contribution block. Conceptually, we can think of this process as splitting a node of the assembly tree into a chain of two nodes, see Figure 3.10.

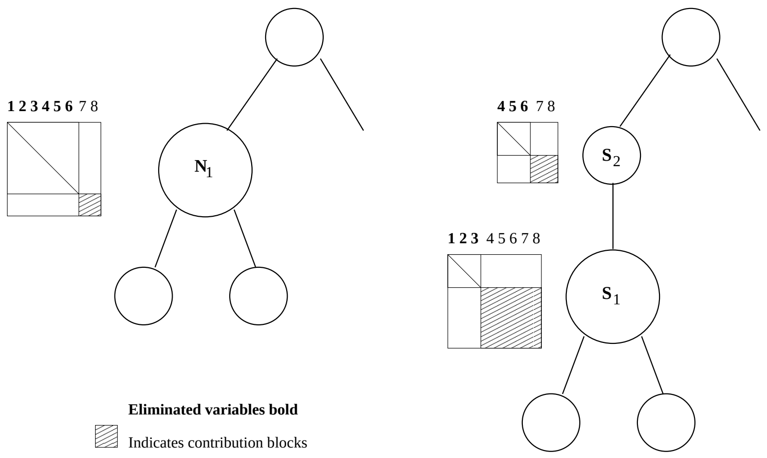

Figure 3.10: Splitting of a node N_1 into S_1 and S_2.

Potential benefits of splitting can be expected in the first step of the process. The outer product update with the factors of the first part of the pivot block is applied to a *larger* contribution block (consisting of the original contribution block and the second part of the pivot block) which offers possible potential for type 2 parallelism.

When deciding whether or not a node should be split, we also have to consider the drawbacks. While the number of factorization operations doesn't change, the number of assembly operations increases. Moreover, the type 2 parallel update of the contribution block is associated with additional costs for the communication between master and slaves. Lastly, if splitting is applied recursively, then long chains of nodes might be created that could lead to more imbalanced assembly trees. A node should therefore only be split if the benefits from splitting clearly outweigh the additional costs.

3.4.3 Delayed pivots and numerical stability

Numerical pivoting is required for the numerical stability of the factorization of each frontal matrix. In general, partial pivoting with a threshold criterion is used [41, 47].

If a fully summed variable cannot be eliminated within a frontal matrix because of numerical stability, then the elimination is postponed and the non-eliminated block is sent to the parent node.

The delay of eliminations corresponds to a dynamic modification of the assembly tree during the factorization phase. In contrast to amalgamation and splitting, discussed in Sections 3.4.1 and 3.4.2, that are performed during the analysis phase prior to the factorization, the delay corresponds to an *a posteriori* modification of the assembly tree [9]. It introduces so-called *numerical fill-in* as is illustrated in Figure 3.11.

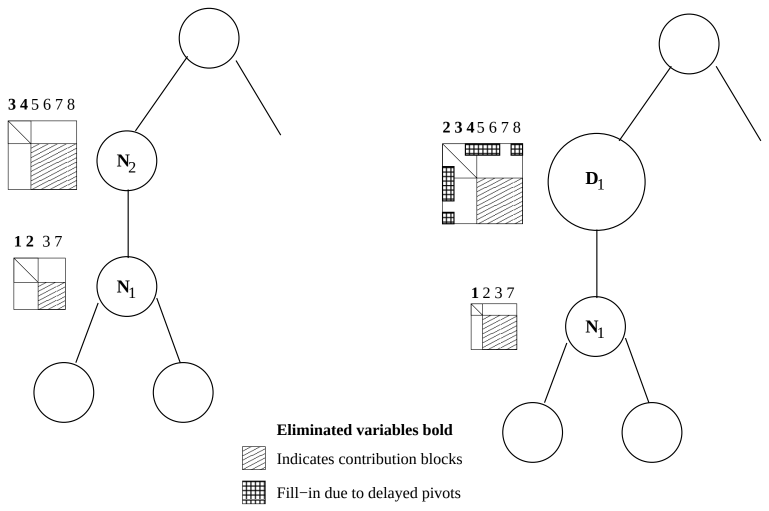
3 4 5 6 7 8
N2
1 2 3 7
N1
2 3 4 5 6 7 8
D1
1 2 3 7
N1
Eliminated variables bold
Indicates contribution blocks
Fill−in due to delayed pivots

Figure 3.11: The elimination of a pivot in node N_1 is postponed, the delayed pivot is assembled into node D_1.

3.5 Combining the concept of candidates with dynamic task scheduling

The dynamic choice of the slaves of type 2 nodes during the factorization phase is an attempt to detect and adjust an imbalance of the workload between the processors at run time. It was shown to work very well on a small to medium (64) number of processors [9, 10]. However, the straightforward extension of this technique to a large number of processors often offers more freedom to the dynamic scheduling than can be exploited effectively. In this section, we first give a more detailed illustration of these shortcomings, and then propose as a solution an algorithm that exploits the concept of candidate processors.

3.5.1 Issues of dynamic scheduling

The first issue of dynamic scheduling concerns the memory management. In MUMPS, the amount of memory needed for each processor is estimated during the analysis phase and is reserved as workspace for the factorization. Consequently, if every processor can be possibly taken as a slave of a type 2 node, then enough workspace has to be reserved on each processor during the analysis phase for the potential corresponding computational task. However, during the factorization, typically not all processors are actually used as slaves. In particular, it is common that when using a large number of processors, only relatively few slaves are needed for most of the type 2 nodes in the tree. This leads to a severe overestimation by the analysis phase of the required work space with the possible consequence of exhausting the memory available on the processors.

Secondly, the choice of the slaves is completely local. When a type 2 node is to be processed, its master greedily takes the slaves that seem best to it; those processors that are less loaded (with respect to the number of floating-point operations) than itself at the time of the scheduling decision are selected as slaves. Thus, the decision about the slaves depends crucially on the instant when the master chooses the slaves (locality in time). Furthermore, no account is taken of other type 2 nodes in the tree that have to be processed (locality in space). Instead of sharing the available slaves so that other nodes can be processed in parallel, a master might decide to take all of them, hindering the work on the other type 2 nodes and the treatment of other branches of the assembly tree. Furthermore, it is not possible in this approach to guarantee any locality of communication and data movement as, in principle, every processor can work on any type 2 node in the assembly tree. However, controlling locality is of great importance for modern computer architectures, for example SMPs like the IBM SP, where, for an MPI programming model, data movement within the shared memory of a node is cheap compared to communication across nodes.

3.5.2 Candidate processors for type 2 parallel nodes

In the following, we present a concept of *candidate processors* that naturally addresses the issues raised in Section 3.5.1. For each type 2 node that requires slaves to be chosen dynamically during the factorization because of the size of its contribution block, we introduce a limited set of processors from which the slaves can be selected. (We postpone here a detailed algorithmic description and refer to Sections 3.7.3 and 3.7.4.) While the master previously chose slaves from among all less loaded processors, slaves are now only chosen from this list of candidates. This effectively allows us to exclude all non-candidates from the estimation of workspace during the analysis phase and leads to a tighter and more realistic estimation of the workspace needed. Secondly, we can expect a performance gain in cases as described in the previous section where greedy decisions of one type 2 master can no longer hinder processors from processing another node.

The candidate concept can be thought of as an intermediate, or *semi-static*, step between full static and full dynamic scheduling. While we leave some freedom for dynamic decisions at run time, this is guided by static decisions about the candidate assignment during the analysis phase. We refer to Section 3.7.6 for a full description of the algorithmic details.

In Section 3.3.2.1, we described the layer structure of the assembly tree. As each layer of the assembly tree represents a view of concurrent execution, all type 2 nodes on the same layer are potential rivals for slave processors, see Section 3.5.1. By assigning the candidates to all type 2 nodes of a given layer simultaneously, we avoid isolated treatment of nodes and direct our candidate concept from a global view of complete layers.

The assignment and the choice of the candidate processors are guided using a proportional mapping as described in Sections 3.3.2.2 and 3.6.1. We partition the set of processors recursively, starting from the root, so that for each subtree there is a well defined subset of *preferential* processors which guides the selection of the candidates.

With this approach, we achieve

- Locality of communication as we limit the communication to those processors belonging to the subtree.
- Independence of computation as we limit the interaction of the processing of one subtree with the treatment of another independent one.

3.6 Task mapping and task scheduling in MUMPS

In this section, we give a generic description of the algorithm used by Version 4.1 of MUMPS [9, 10] and discuss in general terms our improvements to it as they have been integrated into the new version. A detailed discussion of the key modifications is given in Section 3.7. We speak, in the following, of task *mapping* when we refer to the assignment of master processors and candidates during the analysis phase, and of task *scheduling* when we refer to the dynamic choice of type 2 slaves during the factorization phase.

3.6.1 Task mapping algorithm during the analysis phase

We consider the task mapping during the analysis phase and compare the previous with the new version of MUMPS. A first major point to emphasize is the greater flexibility and adaptivity of the new algorithm when mapping the upper part of the assembly tree (that is, above layer L_0). The former version, shown in Algorithm 3, performs a simple mapping of only the master nodes, while the new version, shown in Algorithm 4, treats the upper part layer-wise, mapping both master nodes and type 2 candidates. Using a layer-wise approach we take better account of the task dependency that will control the later factorization phase and, by analysing the quality of mapping decisions taken on previous layers, we can try to correct problems by influencing the mapping of the current layer. This adaptivity was conceptually impossible in the old mapping algorithm.

The second contribution of the new algorithm is of course the added features. A very important feature is the candidate concept guided by a proportional mapping partition of the processors. Furthermore, we have added to the treatment of each layer a preprocessing step that performs amalgamations and node splitting. Moreover, we have improved the construction of layer L_0 for better memory scalability. Lastly, we treat memory imbalances due to type 2 node mapping using a post-processing step.

We now present in more detail the previous version of the task mapping (Algorithm 3) and compare it afterwards with the new one, Algorithm 4.

Algorithm 3 *Old task mapping algorithm.*

```
(1) Given the assembly tree of a sparse matrix $A$
(2) Build and map initial layer $L_0$
(3) Decide type of parallelism for nodes in upper part of tree
(4) Map master nodes of upper part of tree
```

The starting point (1) of the original algorithm is the assembly tree that was constructed from the elimination tree of a given sparse matrix using basic amalgamation and node splitting. From this assembly tree, the algorithm constructs, in step (2), an initial layer L_0 following the Geist-Ng approach presented in Section 3.3.2.1 with the objective of balancing the work among the processors. Afterwards, it is decided for which nodes type 2 or type 3 parallelism is exploited (3), and finally the masters of all nodes above layer L_0 are mapped (4) with the objective of balancing the memory. The choice of the slave processors for the type 2 nodes is left entirely to the dynamic scheduler during factorization, see Section 3.6.2.

Algorithm 4 *New task mapping algorithm.*

```
($1'$) Given the assembly tree of a sparse matrix $A$
($2'$) Calculate relaxed proportional mapping, i.e. the preferential processors
($3'$) Build and map modified initial layer $L_0$
current_layer = 1
while there exist unmapped nodes on or above current_layer do
    ($4'$) Perform tree modifications if necessary
    ($5'$) Decide type of parallelism for the nodes on current_layer
    ($6'$) Map the tasks associated with the nodes on current_layer
    current_layer = current_layer + 1
end while
($7'$) Post-processing of the candidate selection to improve memory balance
```

The starting point ($1'$) of the new algorithm is the same assembly tree as for the old approach (1). In step ($2'$), we calculate a variant of the proportional mapping as introduced in Section 3.3.2.2 and whose algorithmic description is given later in Section 3.7.1. For each node in the assembly tree, we obtain a set of *preferential processors* that will guide the selection and mapping of the candidate processors in step ($6'$). Step ($3'$) differs from the corresponding step (2) in the old algorithm insofar as the constructed initial layer has one additional property. Not only can it be mapped so that the computational work is balanced between the processors, but we also control better the memory demands of the subtree roots, see Section 3.7.2 for the details. Step ($4'$) performs amalgamations and node splitting to improve the nodes of the current layer. In step ($5'$), we decide which type of parallelism we exploit for the nodes of the current layer. Nodes are of type 2 when their contribution block is large enough, that is greater than a minimum block size. The list of tasks associated with the current layer includes the masters for the type 1 and type 2 nodes, and the type 2 candidates which are derived from the proportional mapping ($2'$), see Section 3.7.1. For the task mapping, we use a list scheduling algorithm that is described in Section 3.7.4. The main difference of the new mapping ($6'$) from the old one (4) is that we now pre-assign candidate processors for the type 2 nodes while in the former version, every processor was a potential type 2 slave. The post-processing step ($7'$) affects mostly the *LU* factorization: as the flop-based equilibration of the mapping step ($6'$) can lead to a particularly bad memory balance, we perform a remapping of the type 2 masters for improved memory balance, as described in Section 3.7.5.

3.6.2 Task scheduling during the factorization phase

In this section, we describe the task management of a processor during the factorization phase. Here, the old and the new version of the algorithm differ only in the way the type 2 slaves are chosen. In the previous algorithm, every processor was a potential slave for a type 2 node, whereas now only the candidates can be selected to work on the parallel update of the contribution block.

Algorithm 5 *Dynamic task scheduling performed on each processor during the factorization.*

```
(1) Given the task pool of one processor
while (2) Not all tasks processed do
  if Work is received from another processor then
    (3) Store work in pool of tasks
  else
    (4) Extract work from the task pool
    if Task is master of type 2 node then
      (5) Choose and notify the slaves for the type 2 node
    end if
    (6) Perform pivot elimination and/or contribution block update
  end if
end while
```

The task pool (1) of a processor can contain the following tasks: master of a type 1 node, master of a type 2 node, or slave of a type 2 node. MUMPS uses a stack as the data structure for the task pool; the processor adds new tasks (3) or extracts them from the pool (4), respectively. If, during the factorization, the task pool of the processor is empty, it will wait until it receives new tasks and then re-enter loop (2). If the processor works as a type 2 master, it chooses the slaves that will participate in the parallel contribution block update (5) before it starts the elimination of the pivotal block (6). Otherwise, if the processor is a type 1 master or a type 2 slave, it begins directly with the pivot elimination or the contribution block update, respectively (6).

In the new version of the algorithm, only step (5) is modified to ensure that the type 2 slaves are selected from among the candidates allocated for the type 2 node. We give the details of the algorithm for choosing the slaves in Section 3.7.6.

3.7 Details of the improved task mapping and scheduling algorithms

After the general comparison of the old and new versions of MUMPS task mapping and scheduling in Section 3.6, we describe in this section the key points of the new algorithm in detail.

3.7.1 The relaxed proportional mapping

We give below an algorithmic description of one step of the proportional mapping presented in Section 3.3.2.2. The preferential processors given to a node are distributed among its children according to their weight. Note that we can *relax* the strict proportional mapping by multiplying the number of preferential processors n_a by a relaxation factor $\rho \geq 1$ in step (2).

Algorithm 6 *One step of proportional mapping.*

```
Given a node $n$ with preferential processors $p_1, \dots, p_{n_a(n)}$ and children $s_1, \dots, s_i$
for each child $s$ of $n$ do
    (1) Calculate relative costs $c_r(s)$ of child $s$, $0 \leq c_r(s) \leq 1$
    (2) Calculate number of preferentials $n_a(s) = \min\{\rho \times c_r(s) \times n_a(n), n_a(n)\}$
        for child $s$
end for
(3) Cyclic assignment of the preferential processors for all children $s_1, \dots, s_i$
```

In step (1), we calculate the relative costs $c_r(s)$ of a child $s, s \in \{s_1, \dots, s_i\}$ from the costs $c(s)$ for the factorization of all nodes in the subtree rooted at s as

$$c_r(s) = \frac{c(s)}{\sum_{k=1}^{i} c(s_k)}. \tag{3.1}$$

From the relative weight of child s, we obtain its share of preferential processors in step (2) that can be relaxed by the factor ρ. A fundamental property of the proportional mapping is that the preferential processors of a child s are a subset of the preferential processors of its parent node n. This is ensured because $n_a(s) \leq n_a(n)$ in (2) and we always choose from the preferential processors of the parents at step (3). Here, we also make sure that each child has at least one processor, even if its cost is negligible. After we have calculated the number of preferential processors for all children, we distribute in step (3) the processors $p_1, \dots, p_{n_a(n)}$ among the children. If the proportional mapping is strict ($\rho = 1$) and the number of preferential processors calculated from the relative weight $c_r(s)$ in equation 3.1 is an integer, each processor is assigned exactly to one child. Otherwise, and in particular if the proportional mapping is relaxed with $\rho > 1$, processors can become preferential for more than one child. Consequently, large values of ρ will dilute the strict partition of the processors so that in the extreme case, $\rho \to \infty$, all processors become preferential for each node.

3.7.2 The Geist-Ng construction of layer L_0

We now give an algorithmic description of the construction of the initial layer L_0 that extends the Geist-Ng approach presented in Section 3.3.2.1.

Algorithm 7 *The Geist-Ng algorithm.*

```
(1) Let $L_0$ contain all root nodes of the assembly tree
(2) Map layer $L_0$
while (3) Layer $L_0$ is not acceptable do
   (4) Find node in $L_0$ with highest computational costs
   (5) Replace this node by its children in $L_0$
   (6) Map new layer $L_0$
end while
```

Starting with a potential layer L_0 consisting of the root nodes of the assembly tree (1), we first compute (2) a mapping of L_0 with the list scheduling heuristics described in Section 3.7.4. The former criterion for accepting the layer in step (3) demands that the load imbalance between the processors is smaller than a threshold. Here, the work associated with a node in L_0 is defined as the costs for computing the factors of the subtree rooted at the node and can be estimated during the analysis phase. We discuss below what problems can arise and why we have modified this criterion of acceptability for L_0 . If the mapping of layer L_0 is not acceptable, then the node with the highest costs is eliminated from the layer and replaced by its children (4, 5). A new mapping is computed (6) with the same algorithm as in (2).

The main problem of the algorithm is that balancing the computational work does not necessarily imply balancing the memory. Consider a node with a very small number of pivots but a big contribution block. The costs for the factorization depend mainly on the size of the pivotal block and are small, while the memory required to stack the contribution block is large. In order to take care of such situations, we propose the following approach. If a node with a large contribution block was in the upper part of the tree above L_0 , it could either be amalgamated with its parent or become a type 2 node, and, in both cases, the memory problems would vanish. Thus, by eliminating such nodes from layer L_0 and replacing them by their children, we control the memory required for the subtrees in addition to balancing the work on layer L_0. The idea is to treat critical nodes in the latter part of the algorithm by moving them into the upper part of the tree.

Summarizing, we modify the criterion of acceptability (3) to demand that both the load imbalance for the mapping of L_0 is smaller than a threshold *and* that L_0 contains no nodes that would need to be amalgamated.

3.7.3 Choosing the number of candidates for a type 2 node

We now describe the role of the proportional mapping for the decision of which processors become candidates for a type 2 node. Our approach consists of two steps. For a given layer, we first determine for each type 2 node the *number* of candidate processors. In the second step, we choose the candidates from the available processors. (Thus, for a given node n we determine first the number of candidate processors, $n_c(n)$, that have to be chosen in the second step.) The selection of a candidate processor is conceptually similar to the selection of the master processors for the type 1 and type 2 nodes; all these are tasks that need to be mapped onto the set of processors. In Section 3.7.4, we describe the algorithm that we use to map the tasks associated with one layer in the assembly tree. By mapping the master and candidate processors together, we hope to obtain better load balancing.

We have experimented with two different ways for determining the number of candidates for a given type 2 node and describe these in Algorithm 8. In the first approach, we select its preferential processors as candidates, thus setting the number of candidates equal to the number of preferentials. We emphasize that this approach is *not* equivalent to a relaxed proportional mapping as the candidates are only *potential* slaves for the factorization. In a second approach, we employ an additional post-processing step where we redistribute the candidates of the layer according to the relative weight of the nodes. As described in Section 3.3.2.2, the proportional mapping is calculated from the costs of complete subtrees, not individual nodes. So it might happen that a small node has a large number of preferentials because it is the root of a large subtree, while a relatively large node on the same layer has only a small number of preferentials. In order to correct this, we can reassign candidates from small type 2 nodes as candidates of large type 2 nodes on the same layer by the optional step in Algorithm 8.

Algorithm 8 *Determining the number of candidates using the preferentials.*

```
Given a layer in the assembly tree
for each Type 2 node $n$ with $n_a(n)$ preferential processors in the layer do
    (1) Determine the number of candidates by $n_c(n) = n_a(n)$.
end for
(2) OPTIONAL: Redistribute the total number of candidates of the layer among
              the layer's type 2 nodes according to their relative weight.
```

3.7.4 Layer-wise task mapping

The algorithm that we use for the mapping of the tasks of each layer is a variant of the well known list scheduling algorithm [89] where we first make a list of the tasks sorted by decreasing costs, and then maps the tasks in this order one after another to the processor that has the least work assigned so far. We remark that this heuristic can be proved to construct a schedule whose total makespan (that is, the time by which all jobs complete their processing) never exceeds twice the makespan of an optimal schedule [89].

As described in the previous sections, the tasks associated with a layer can include the following:

- The masters of all type 1 and type 2 nodes.
- For each type 2 node, the number of candidate processors determined using the node's preferential processors, see Section 3.7.3.
- The type 3 parallel node.

In the case of layer L_0, we employ the original list scheduling, however, for all upper layers $L_1, L_2, \ldots$ our algorithm is more complicated for two reasons. First, we want to guide mapping decisions by the proportional mapping representing a global view of the tree. Second, we have to take care of constraints that arise either from explicit user-given limits on memory or work for each processor, or implicitly from the fact that any two candidate processors or any candidate and the master of a type 2 node have to be different from each other.

Algorithm 9 *Generic mapping algorithm.*

```
(1) Create an ordered task list
while Task list not empty do
  (2) Extract the next task $t_i$ from the list
  (3) Make a preference list for the processors
  while Task $t_i$ not mapped to a processor do
    (4) Try to map $t_i$ to next processor from the preference list
  end while
end while
```

The first two steps (1) and (2) of Algorithm 9 are identical to the original list scheduling approach: we create a list of all tasks that have to be mapped on the layer, that is, the work of the type 1 node masters, of type 2 node masters, and of type 2 node candidates (which have been obtained from the proportional mapping, as described in Section 3.7.1). This list is then ordered by decreasing costs and the tasks are mapped in the order that they appear in the list.

Steps (3) and (4) are the generalization of the idea of mapping to the least loaded processor. In order to take account of the proportional mapping, we can simply propose mapping the task on the least loaded of the preferential processors coming from the proportional mapping. However, this is actually too simple as the mapping constraints for type 2 nodes that we mentioned above have to be respected. Our solution is that we create a *preference* list containing all the processors, at first the preferential ones ordered by decreasing workload and then the non-preferential ones ordered separately. The first processor in the preference list that doesn't violate the mapping constraints will be the one to which the task is mapped.

3.7.5 Post-processing of the assembly tree for an improved memory balance in the LU factorization

.

The mapping algorithm from Section 3.7.4 tries to balance the work between the processors. However, there is an important difference between symmetric and unsymmetric factorization with respect to memory. In the LDL^T factorization, the master of a type 2 node only holds the pivotal block whereas, in the LU factorization, the master stores the *complete* fully summed rows. The additional memory that a master requires for storing its part of the factors in the LU factorization (with respect to the LDL^T factorization) is illustrated in Figure 3.12. In both the LU and the LDL^T factorization, a type 2 slave reserves space for a part of the L factor below the pivotal block as shown in Figure 3.13. Thus, in the case of the LU factorization, the work equilibration can lead to memory imbalances if the same processor becomes master of several type 2 nodes.

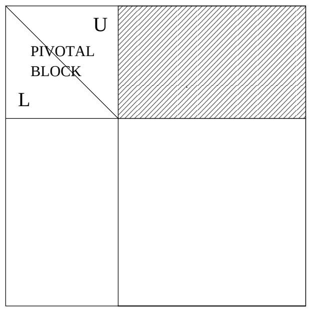

Figure 3.12: Additional memory needed by a type 2 master in the unsymmetric factorization.

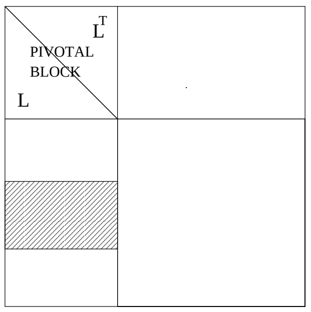

Figure 3.13: Memory reserved for the L factor on a type 2 slave in both LU and LDL^T factorization.

We propose the following simple remedy. After the whole tree is mapped with the objective of balancing the work, we use a post-processing step to correct obvious memory problems.

We process the upper part of the assembly tree from the top down (1), as the type 2 nodes creating the biggest problems are often near a root of the tree. By swapping a master processor with one of the candidates, we still guarantee the benefits of the proportional mapping used in the candidate assignment, but locally improve the memory imbalance (steps 2 and 3).

Algorithm 10 *Post-processing for better memory equilibration in the LU factorization.*

```
(1) Process the type 2 nodes in the tree from the root downwards
(2) For a node $n$ with master $p^M(n)$ select candidate $c^*(n)$ with smallest memory
if memory imbalance can be improved by swapping $p^M(n)$ and $c^*(n)$ then
    (3) Exchange the roles of master and candidate processor $p^M(n) \Leftrightarrow c^*(n)$
end if
```

3.7.6 The dynamic scheduling algorithm used at run time

We describe the dynamic scheduling algorithm used in MUMPS for the mapping of the slaves of a type 2 node at run time [9] and show how the candidate concept influences the original approach. Furthermore, we identify and describe the role of the algorithmic parameter $k_{\max}$ controlling the minimum granularity for type 2 parallelism at run time.

Algorithm 11 *Dynamic choice of the slaves of a type 2 node.*

```
Given a type 2 node $n$ with master processor $p^M(n)$ and children $s_1, \ldots, s_i$
(1) The masters of the children $p^M(s_1), \ldots, p^M(s_i)$ send symbolic data to $p^M(n)$
(2) $p^M(n)$ analyses its information concerning the load of all processors
(3) $p^M(n)$ decides the partitioning of the frontal matrix of node $n$ and chooses
    the slave processors $p^S_1(n), \ldots, p^S_j(n)$
(4) $p^M(n)$ informs all processors working on the children about the partition
(5) The numerical data is sent directly to the slaves $p^S_1(n), \ldots, p^S_j(n)$
```

Instead of first assembling all numerical data on the master of the type 2 node and distributing it afterwards to the slaves, a two-phase assembly process is used. At step (1), the master receives only the integer data describing the symbolic structure of the front. At step (2), the master analyses the information on the work-load of the other processors: Each processor is responsible for monitoring its own work-load status and for broadcasting significant changes to all other processors so that everyone has accurate information on the overall progress of the factorization. At step (3), the master processor $p^M(n)$ selects the least loaded among all processors as slaves. As a general rule, all processors that are less loaded than $p^M(n)$ are chosen as slaves. Then, a partition of the frontal matrix onto the slaves is calculated.

The following constraint on the number of slaves, n_{s}, for a type 2 node selected during factorization is imposed. It must always satisfy

$$n_{\mathrm{s}} \geq \max(\lfloor \frac{ncb}{k_{\max}} \rfloor, 1), \tag{3.2}$$

where ncb denotes the number of rows in the contribution block. (The result of the division on the right-hand side is truncated to the next smaller integer but must always be at least one.) The parameter $k_{\max}$ controls the maximum work of a type 2 slave and thus the maximum buffer size permitted for the factorization of a type 2 node.

Once the slaves participating in the parallel update of the contribution block have been selected, they obtain the part of the symbolic information from the master $p^M(n)$ that is relevant for their work (4). Furthermore, they receive the corresponding numerical data from the processors working on the children (5).

In the candidate-based scheduling approach, we modify step (3) so that the slaves are *always* chosen among the candidates provided for the node. At first, we select all those candidates that are less loaded than the master processor. If the inequality (3.2) is not satisfied, additional candidates are chosen so that it holds. In order to be able to choose the slaves at factorization time among the candidates so that (3.2) is never violated, we must take care to provide enough candidates during analysis. If we provide only a minimum number of candidates so that (3.2) holds as equality, we enforce a static scheduling. In this case, all candidates must be selected as slaves during factorization. We have freedom for dynamic choices during the factorization only if we provide a number of candidates greater than $ncb/k_{\max}$. Consequently, the freedom offered to the dynamic scheduling can be measured by the number of extra candidates given for a type 2 node. On the other hand, the larger the number of candidates for a given node, the closer we come to the case of fully dynamic scheduling with all the possible drawbacks discussed in Section 3.5.1. In the following experiments, we will see that the dynamic scheduling works most effectively when $k_{\max}$ is large and (3.2) does not impose a significant restriction. Because of overall memory constraints, the scope for increasing the parameter is limited; however, we will see that the better memory estimates from the candidate approach greatly increase the range from which $k_{\max}$ can be chosen.

We remark that, in the case of the LDL^T factorization, MUMPS precomputes a partition of the contribution block in order to guarantee that each of the slaves performs approximately the same amount of work [8]. As the frontal matrix is symmetric (and only the lower triangular part is stored), rows at the bottom of the frontal matrix are longer (and thus associated with more work) than rows at the top.

3.8 The test environment

In this section, we present the test matrices that we use to illustrate the behaviour of our algorithm. Specifically, we consider in Section 3.8.1 matrices from regular grids and in Section 3.8.2 irregular ones from real-life applications. We mention that our set of regular grid problems includes those used in [10] which allows us to compare the performance of the new code with results already published.

For our tests, we use both a CRAY T3E-900 (512 processors, 256 MBytes RAM and 900 peak MFlops per processor) and an SGI Origin 2000 (32 processors, 16 GBytes shared memory, 500 peak MFlops per processor). We consider different orderings including nested dissection from SPARSPAK [64], METIS [98], and SCOTCH [112, 113], and Approximate Minimum Fill [119, 109].

3.8.1 Regular grid test problems

We consider a set of test matrices obtained from an 11-point discretization of the Laplacian on 3D grids of either cubic or rectangular shape, the grid sizes are reported in Table 3.1. The set of problems is chosen as in [10] and is designed so that when the number of processors increases, the number of operations per processor in the *LU* factorization stays approximately constant when employing a nested dissection ordering [64].

Processors	Rectangular grid sizes			Cubic grid size
1	96	24	12	29
2	100	20	20	33
4	120	30	15	36
8	136	32	16	41
16	152	38	19	46
32	168	42	21	51
48	172	44	22	55
64	184	46	23	57
128	208	52	26	64
256	224	56	28	72
512	248	62	31	80

Table 3.1: 3D grid problems.

In Tables 3.2 and 3.3, we show for the grid problems the distribution of work for type 1 masters (T1), type 2 masters (T2M) and slaves (T2S), and the type 3 root node (T3). It can be seen that, when increasing the problem size and the number of processors used, the work of the type 2 slaves becomes a major part of the overall work. Thus, improving the mapping of the type 2 slaves through the candidate concept will have a great influence on the overall performance of the factorization, in particular on larger problems.

3.8.2 General symmetric and unsymmetric matrices

The matrices described in this section all arise from industrial applications and include test matrices from the PARASOL Project [3], the Rutherford-Boeing Collection [49], and the University of Florida sparse matrix collection [32].

In Table 3.4, we describe the characteristics of the test matrices arising from real life problems. In Table 3.5, we show for the irregular problems on 64 processors the distribution of work for type 1 masters (T1), type 2 masters (T2M) and slaves (T2S), and the type 3 root node (T3). We see that the work distribution depends heavily on the ordering used. The AMF ordering produces assembly trees that are rich in type 2 parallelism; on the other hand, the root nodes are so small that

	LU				*LDL^T*			
Processors	T1	T2M	T2S	T3	T1	T2M	T2S	T3
1	100	0	0	0	100	0	0	0
2	85	0	0	15	85	0	0	15
4	45	7	34	14	45	2	39	14
8	28	7	49	14	28	2	56	14
16	18	5	63	14	18	2	65	15
32	7	4	75	14	8	1	77	14
48	7	4	75	15	8	1	77	14
64	5	3	78	14	5	1	81	13

Table 3.2: Percentage distribution of work for 3D cubic grid problems (nested dissection ordering).

	LU				*LDL^T*			
Processors	T1	T2M	T2S	T3	T1	T2M	T2S	T3
1	100	0	0	0	100	0	0	0
2	88	0	0	12	88	0	0	12
4	84	1	3	12	84	1	3	12
8	49	5	34	12	49	3	36	12
16	25	5	58	12	25	2	61	12
32	16	4	68	12	16	2	70	12
48	14	4	70	12	12	2	74	12
64	10	4	74	12	10	1	77	12

Table 3.3: Percentage distribution of work for 3D rectangular grid problems (nested dissection ordering).

Matrix name	Matrix type	Matrix order	Nmb. entries	Origin
`bbmat`	symmetric	38744	1771722	Rutherford-Boeing
`ecl32`	symmetric	51993	380415	Rutherford-Boeing
`g7jac200`	symmetric	59310	837936	University of Florida
`twotone`	symmetric	120750	1224224	Rutherford-Boeing
`ship003`	unsymmetric	121728	8086034	PARASOL
`bmwcra_1`	unsymmetric	148770	10644002	PARASOL

Table 3.4: Matrix order, type, and number of entries for the irregular test matrices.

type 3 parallelism cannot be exploited effectively, in contrast to both SCOTCH and METIS. For all matrices apart from `bbmat`, the major part of the work is associated with the factorization of type 2 nodes, similar to the regular grid problems.

	AMF				METIS				SCOTCH			
Matrix	T1	T2M	T2S	T3	T1	T2M	T2S	T3	T1	T2M	T2S	T3
`bbmat`	43	3	54	0	57	7	30	6	75	3	19	3
`ecl32`	14	8	78	0	29	8	52	11	25	9	51	15
`g7jac200`	9	2	89	0	12	5	71	12	3	12	69	16
`twotone`	6	6	90	0	7	8	79	6	11	7	52	30
`ship003`	7	7	85	1	14	10	65	11	15	12	63	10
`bmwcra_1`	22	8	70	0	36	12	51	1	40	9	49	2

Table 3.5: Percentage distribution of work for irregular problems on 64 processors with different orderings.

3.9 Experimental investigation of algorithmic details

In this section, we study the influence and scope of parameters in the algorithms used by Version 4.1 [9, 10] and by the new version of MUMPS. Furthermore, we present a detailed investigation of isolated parts of the improved algorithm by typical examples of phenomena that we have observed in our experiments.

3.9.1 The impact of k_{max} on volume of communication and memory

We first show the impact of the parameter $k_{\max}$ that controls the minimum granularity of the type 2 parallelism, on the volume of communication and memory.

Our test matrix is from Section 3.8.1 and comes from an 11-point discretization of the Laplacian on a cubic grid of order 46, ordered by nested dissection. Here, we perform an LU factorization on an SGI Origin 2000 with 16 processors. This platform is well suited for testing the $k_{\max}$ parameter over a wide range of values because of its shared-memory architecture where a large amount of memory is available to all processors.

At first, we study the behaviour of Version 4.1 of MUMPS and then compare it with the new code.

The two graphs in the upper row of Figure 3.14 illustrate that with increasing $k_{\max}$, both the total volume of communication and the number of messages associated with dynamic scheduling decrease. If $k_{\max}$ is small, the required minimum number of slaves for a type 2 node and the corresponding communication volume will be large. With increasing $k_{\max}$, a single type 2 slave might be authorized to

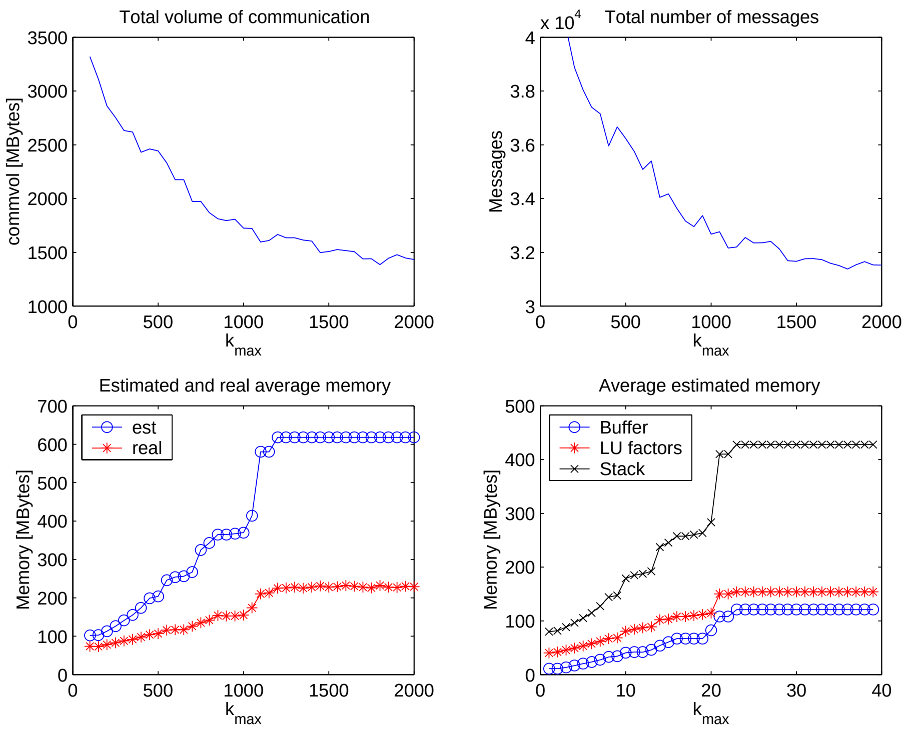

Figure 3.14: Impact of $k_{\max}$ on volume of communication and memory in Version 4.1 of MUMPS (Origin 2000, 16 processors).

work on larger parts of a contribution block and the minimum number of slaves required during factorization becomes smaller. With $k_{\max}$ sufficiently large and thus the guaranteed minimum number of slaves from inequality (3.2) being no longer a constraint, the dynamic scheduling can freely choose slaves among the least loaded processors. Thus, further increases in $k_{\max}$ do not further reduce the volume of communication.

The graph in the left lower corner of Figure 3.14 shows the increase in estimated and actually used memory with increasing $k_{\max}$, and the graph in the right lower corner shows the decomposition of the estimated memory into the space reserved for the communication buffers, the LU factors, and the stack. As potentially every processor can be selected as a slave during the factorization and the memory predicted depends monotonically on $k_{\max}$, the prediction during the analysis phase will lead to an increasing gap between real and estimated memory as can be seen in the graph on the lower left. On the lower right, we see that the main contribution to the overestimation of the memory is the stack. As slaves stack their part of the contribution block until it can be received by the processors working on the parent of the node, the stack has to grow when $k_{\max}$ increases. Furthermore, a single type 2 slave is authorized to work on larger parts of a contribution block.

When weighing memory estimation and communication volume against each other, the best value for $k_{\max}$ is so that it reduces the memory over-estimation but at the same time limits the communication volume sufficiently.

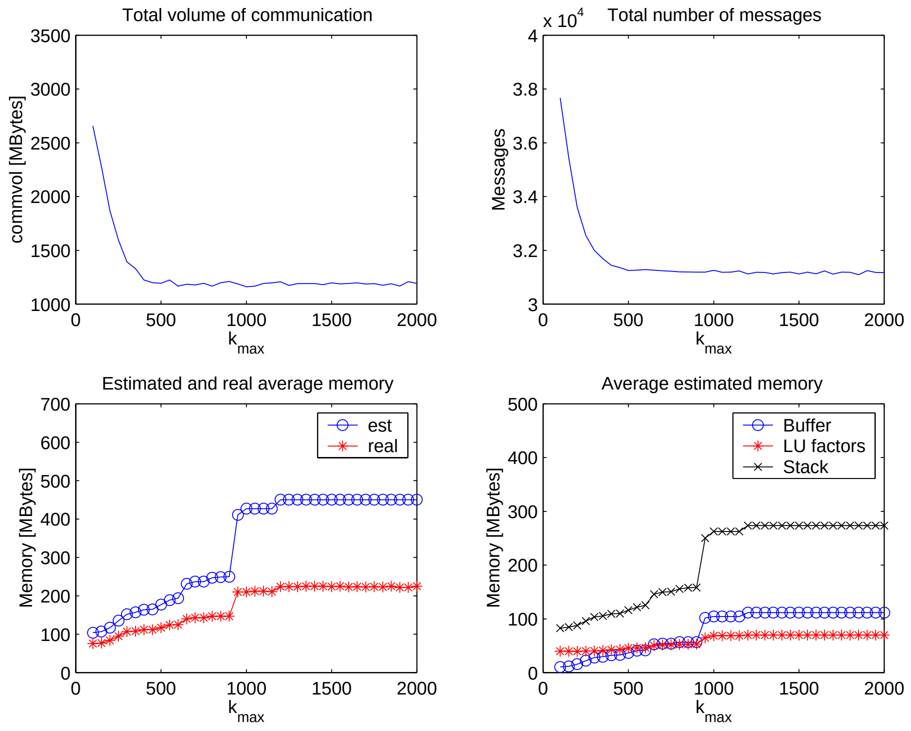

Figure 3.15: Impact of $k_{\max}$ on volume of communication and memory in the new version of MUMPS (Origin 2000, 16 processors).

We now investigate the behaviour of the new candidate-based code on the same test matrix. Candidates are assigned without relaxation and layerwise redistribution, following the proportional mapping of the assembly tree. From the two graphs in the bottom row of Figure 3.15 we observe the expected better estimation of memory. Compared to the corresponding graphs in Figure 3.14, the growth of the gap between estimated and real memory is significantly smaller. As the type 2 slaves can only be chosen from the candidates, the non-candidates can be excluded thus making the estimation tighter and more realistic. Furthermore, the two graphs in the top row of Figure 3.15 indicate that the communication volume in the new version of MUMPS drops faster with increasing $k_{\max}$ than it does for the previous version. This can be explained by the restricted freedom for the dynamic scheduling, so that actually less parallelism is created and fewer slaves are chosen during factorization. Thus, if we want to reduce the communication volume in the new code, we are not obliged to increase $k_{\max}$ substantially with the consequent drawback of overestimating the memory. Instead, we can choose $k_{\max}$ relatively small and have the benefits of a relatively realistic memory estimation together with a reduced communication volume.

3.9.2 The impact of k_{max} on performance

In the following example, we show the impact of the parameter $k_{\max}$ on the factorization time.

Our test matrix arises from an 11-point discretization of the Laplacian on a cubic grid of order 51, ordered by nested dissection; we perform an LU factorization on a CRAY T3E with 64 processors. (Compared to Table 3.1, we have reduced the problem size to have enough flexibility with respect to memory for this parameter study.) Furthermore, because of limited memory and in order to separate the different algorithmic parameters, we use a candidate assignment without relaxation. For a study of the influence of relaxation we refer to Section 3.9.3.

The T3E is well suited for providing reliable timing for performance measures because the processors are guaranteed to run in dedicated mode for a single task. On the other hand, the T3E has a distributed-memory architecture with a fairly small amount of memory per processor, so that we can vary the parameter $k_{\max}$ only in a relatively small range compared to the range possible on the Origin 2000 which has a shared memory.

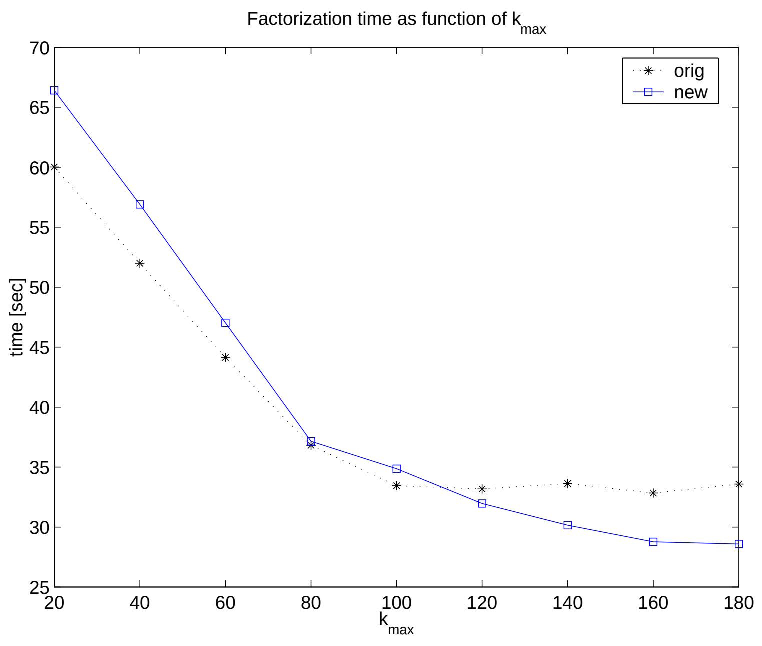

Figure 3.16: Impact of $k_{\max}$ on the performance of the original and the new version of the LU factorization time (CRAY T3E, 64 processors).

From Figure 3.16, we see that with increasing $k_{\max}$, the factorization time decreases in both versions of the code as the minimum number of slaves required during factorization gets smaller and the dynamic scheduler is free to decrease unnecessary parallelism. However, the previous version of MUMPS needs much more memory than the candidate-based version, and thus the flexibility for increasing $k_{\max}$ is more strictly limited.

Once $k_{\max}$ is sufficiently large, a further increase in $k_{\max}$ shows no further improvements in performance. This corresponds to the results on the limited reduction in the volume of communication obtained in Section 3.9.1. We note that for the larger values of $k_{\max}$, the new version of the code performs better. We will confirm this observation by systematic studies on our set of test matrices in Section 3.10.

3.9.3 Modifying the freedom offered to dynamic scheduling

We now investigate the behaviour of the new code when modifying the assignment of candidates. We study two different approaches. As described in Section 3.7.3, we can increase the number of candidates given to a node by relaxing its number of preferentials through the proportional mapping. Furthermore, according to Algorithm 8, we can modify the candidate assignment for a given layer by an optional redistribution of the candidates that takes account of the weight of the nodes relative to each other.

We first compare the performance of the candidate assignment with and without layerwise redistribution. Afterwards, we show the impact of relaxation on the two assignment strategies.

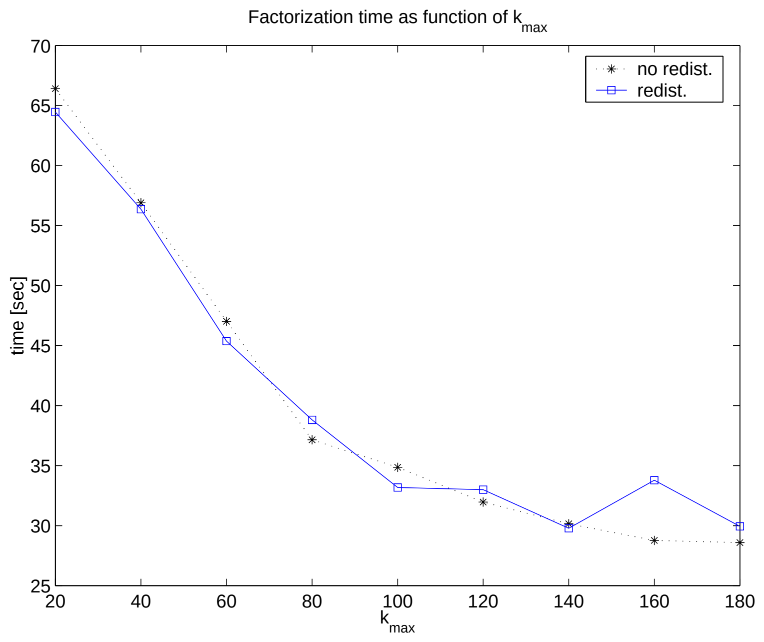

Figure 3.17: Comparison of the candidate assignment with (solid) and without (dotted) layer-wise candidate redistribution when increasing minimum granularity (LU factorization time on CRAY T3E, 64 processors, no candidate relaxation).

For our study, we use the same test case as in Section 3.9.2. Figure 3.17 shows the factorization time of the new version of MUMPS for the candidate assignment with and without layer-wise candidate redistribution as a function of the minimum granularity. We cannot find significant differences in the behaviour of the two approaches. This example is representative for the results we obtain on the complete set of test problems. With or without layer-wise redistribution of the candidates, the performance of the algorithm is similar.

We now investigate the impact of relaxation on the volume of communication, memory, and performance. In Figures 3.18 and 3.19, the horizontal axis denotes the percentage relaxation factor. We present the behaviour of the new version of MUMPS for the candidate assignment with and without layer-wise candidate redistribution as a function of the relaxation.

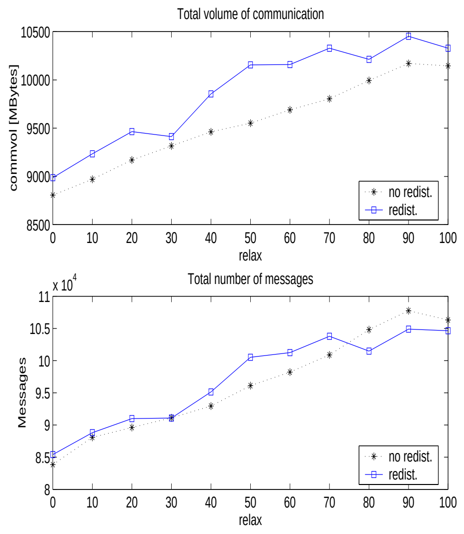

Figure 3.18: Amount of communication in original and modified candidate assignment when increasing the relaxation (CRAY T3E, 64 processors).

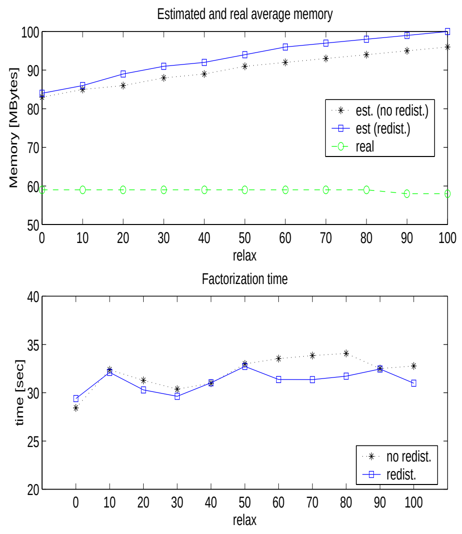

Figure 3.19: Memory estimation and performance of original and modified candidate assignment when increasing the relaxation (CRAY T3E, 64 processors).

The two graphs in Figure 3.18 illustrate that, with increasing relaxation, both the total volume of communication and the number of messages related to dynamic scheduling increase because the flexibility for choosing the slaves during factorization becomes greater. Likewise, the memory estimation grows with increasing relaxation, see Figure 3.18. However, we do not observe a positive impact of relaxation on the performance of the algorithm; a possible interpretation is that, through the relaxation, we create additional parallelism that is not actually needed at run time. In general, we already have without relaxation enough freedom for a dynamic choice of the slaves. While this observation holds for all the experiments we have conducted, we are convinced that relaxation might show a positive impact on irregular problems from real-life applications. Unfortunately, the irregular problems available to us are not large enough to effectively exploit parallelism on a large number of processors, and we plan to investigate this further in the future, see the remarks in Section 3.11.2.

In conclusion, we note that the candidate approach without layer-wise redistribution and without additional relaxation already offers good results in our experiments.

In the following, we focus therefore on the presentation of the results obtained with this algorithmic configuration.

3.9.4 Improved node splitting

We illustrate the additional capabilities for node splitting on the set of test matrices from Section 3.8.1. Those matrices are obtained from an 11-point discretization of the Laplacian on 3D cubic or rectangular grids with Approximate Minimum Fill (AMF) ordering and are described in Table 3.1. The AMF ordering produces long and thin trees from the regular grid problems which we can use to illustrate the problems of the splitting criterion used in the previous version of the code. Splitting was done in a preprocessing step and before the layer structure of the assembly tree was known. In order to prevent useless splitting below layer L_0 where no type 2 parallelism is exploited, the algorithm authorized node splitting only up to a fixed distance from the root node, where this distance depended only on the number of processors but not on the matrix. So it could happen that even though there were nodes in the upper part of the tree that should have been split for better performance, the splitting was not performed.

	Cubic grids (AMF)			Rectangular grids (AMF)		
	Number of splittings			Number of splittings		
Processors	added by new code	total in new code	Nodes in upper tree	Added by new code	Total in new code	Nodes in upper tree
1	0	0	1	0	0	1
2	0	0	13	0	0	118
4	0	0	14	0	0	285
8	2	5	31	0	0	246
16	2	7	41	0	2	188
32	7	21	96	0	2	175
48	8	37	120	3	10	136
64	14	32	140	1	4	194
128	2	13	192	0	7	196
256	2	33	348	50	80	414
512	Not enough memory in analysis			61	107	830

Table 3.6: Improved splitting of the new code.

The new algorithm incorporates the splitting systematically in the upper part of the tree. Once layer L_0 is known, we can authorize splitting everywhere in the upper part of the tree to create more parallelism if this is useful. We illustrate the additional splitting in Table 3.6 for both cubic and rectangular grids. For each grid type, we show in the first of the three columns the additional number of splittings of the new code and compare them to the total number of splittings (including the splittings already performed by Version 4.1 of the code) and the total number of nodes in the upper part of the tree after splitting in the second and third columns, respectively. For example, for the rectangular grid on 512 processors, and with the same splitting criteria, the new code performs 61 splittings in addition to those already done by the old code, so that altogether 107 splittings are performed, resulting in an assembly tree with 830 nodes. In other words, in this example, the previous version of MUMPS missed 57% of the possible splittings.

We illustrate, in Table 3.7, the properties and benefits of the improved splitting in the case of the cubic grid of order 57 on 64 processors. The additional splitting slightly increases the number of assembly operations and also the average amount of memory per processor. However, it creates additional parallelism by augmenting the number of type 2 nodes. This significantly improves the performance of the factorization. Moreover, memory can be balanced better between the processors because of the additional type 2 parallelism.

	Nmb	Operations		Mem. est.		Mem. real		Facto.
Algorithm	type 2	elim.	assem.	max	avg	max	avg	time
no splitting	126	7.98e+11	1.10e+09	196	143	141	92	182
with splitting	140	7.98e+11	1.30e+09	167	150	119	96	145

Table 3.7: Comparison of the candidate-based LU factorization with and without improved node splitting, cubic grid of order 57 on CRAY T3E with 64 processors (AMF).

3.9.5 Improved node amalgamation

In this section, we illustrate the improvements we have made concerning node amalgamation by using our test examples from Table 3.1 with a nested dissection ordering.

In Table 3.8, we show, for both cubic and rectangular grids, the number of extra amalgamations the new code performed in layer L_0 of the Geist-Ng algorithm as described in Section 3.7.2 and the total number of extra amalgamations performed on all layers of the tree. We also give the total number of nodes in the upper part of the tree after all amalgamations have been performed.

We emphasize that, in the new version, we use the same amalgamation criteria as in the previous version and show, in the table, the amalgamations that are performed *in addition* to those performed before. Amalgamation in the previous version was only possible between a parent node and its oldest child; the greater freedom in the new code allows many more amalgamations as can be seen in particular for the large test matrices on 512 processors. For example, for the cubic grid on 512 processors, the new code performed 119 additional amalgamations, that is, 119 amalgamations more than the old code with the same amalgamation criteria. Among the additional amalgamations of the new code are 77 for layer L_0, so that the amalgamated assembly tree has 1371 nodes in the upper part.

	Cubic grids (ND)			Rectangular grids (ND)		
	extra amalg		total	extra amalg		total
Processors	L_0	total	nodes	L_0	total	nodes
1	0	0	1	0	0	1
2	0	0	1	0	0	1
4	2	2	5	0	0	3
8	2	3	18	0	0	17
16	1	4	43	0	1	53
32	7	12	121	2	2	90
48	13	18	126	0	1	104
64	9	14	157	3	5	156
128	0	7	250	2	2	271
256	36	54	528	2	7	525
512	77	119	1371	66	93	1326

Table 3.8: Improved amalgamation of the new code.

We illustrate in Table 3.9 the properties and benefits of the improved amalgamation in the case of the cubic grid of order 46 on 16 processors. The additional amalgamation decreases the number of assembly operations and allows a better memory balance because the stacking of several large type 1 nodes can be avoided.

Type of	Operations		Mem		Mem real		Fact.
amalgamation	assem.	elim.	max	avg	max	avg	time
Old	2.44e+08	5.91e+10	187	121	175	97	19.4
New	2.35e+08	5.91e+10	108	95	82	71	18.7

Table 3.9: Comparison of the candidate-based LDL^T factorization with and without improved node amalgamation, cubic grid of order 46 on CRAY T3E with 17 processors.

3.9.6 Post-processing for a better memory balance

On the CRAY T3E, we illustrate a shortcoming that we detected when testing an initial version of the candidate-based LU factorization. We consider the cubic grid problem of order 72 from Table 3.1 ordered by nested dissection. We show the benefits obtained by remapping the masters of type 2 nodes for better memory balance as described in Section 3.7.5 and conclude that the post-processing is also crucial for obtaining good speedup.

Looking at the first rows (no postp.) of Table 3.10, we see that the flop-based equilibration of the scheduling algorithm leads to severe memory imbalance both in the estimated and the actual memory. In particular, the process needing the largest amount of (estimated) memory requests 179 megabytes, about 70% of the memory of the processor. For performance reasons, it is necessary to increase $k_{\max}$, see Section 3.9.2. However, this is impossible because of the strong memory imbalance, as augmenting $k_{\max}$ increases the memory estimations of the analysis phase considerably.

Algorithm	$k_{\max}$	Max est	Avg est	Max real	Avg real	Fact. time
no postp.	80	179	117	172	102	165.5
	160	Not enough memory				
w. postp.	80	136	117	123	102	152.1
	160	193	164	162	132	123.6

Table 3.10: Memory (in MBytes) and factorization time (in seconds) of the candidate-based LU factorization with and without post-processing, cubic grid of order 72 with nested dissection.

In the last two rows of Table 3.10, we show the memory statistics when the post-processing is performed. We observe that the difference between average and maximum values for both the estimated and actual memory are much reduced. This allows us to double the $k_{\max}$ parameter for this test case and obtain better performance for the factorization. However, note that the estimate for the most loaded processor is more accurate without post-processing. This is because the differences between memory estimation and actually used memory are mainly related to a processor being a type 2 candidate but not being chosen as a slave during factorization. Without post-processing, the major activity for the most loaded processor probably involves work associated with being master of several type 2 nodes, see Section 3.7.5. But after the post-processing, the processor exchanges its role as master with another and becomes a type 2 candidate so that its memory estimate can become less accurate.

3.10 Performance analysis

In the following, we compare the performance of the new MUMPS code with the previous version [10] on the complete set of test problems presented in Section 3.8. All these tests were performed on the CRAY T3E of the NERSC computing center at Lawrence Berkeley National Lab in Berkeley, California.

3.10.1 Nested dissection ordering

In this section, we use the test matrices from Table 3.1 ordered by nested dissection.

We observe, from the results in Table 3.11, that for up to 64 processors, the new version has similar performance to the good results of the previous version.

However, when more processors are used and the matrices become larger, the new code performs significantly better. Looking at the results on 128, 256 and 512 processors, we note the greatly improved scalability of the candidate-based code.

	Cubic grids (ND)			Rectangular grids (ND)		
Processors	flops	old	new		old	new
1	7.2e+09	23.2	23.2	4.5e+09	16.6	16.6
2	1.6e+10	29.1	29.0	9.5e+09	17.2	17.1
4	2.7e+10	27.4	23.9	1.8e+10	16.6	16.9
8	6.0e+10	30.1	29.5	3.7e+10	20.6	19.2
16	1.2e+11	30.8	31.8	7.3e+10	22.4	23.3
32	2.3e+11	43.3	42.2	1.4e+11	25.7	27.4
48	3.6e+11	53.0	57.5	1.8e+11	26.0	23.9
64	4.5e+11	59.0	52.9	2.4e+11	31.2	30.2
128	8.9e+11	93.4	72.7	4.9e+11	44.9	38.5
256	1.8e+12	163.5	119.4	7.7e+11	75.4	47.1
512	3.4e+12	599.6	189.1	1.4e+12	135.5	73.7

Table 3.11: Performance of the old and new LU factorization (time in seconds on the CRAY T3E).

Another major advantage of the new candidate-based code is that it better estimates the memory used for the factorization. In Table 3.12, we show the memory space for the LU factors of the old and the new version of MUMPS. We see that the candidate-based code significantly reduces the overestimation of the storage required, and that the gains increase with the matrix size and the number of processors.

The big gains of the new candidate-based code are a result of the individual improvements concerning splitting and amalgamation, reduced communication and the better locality of the computation as illustrated in Section 3.9. Furthermore, we need to decrease $k_{\max}$ in the large problems for the old version of MUMPS because of memory. This limits the performance as we saw in Section 3.9.2. On the other hand, we do not need to decrease $k_{\max}$ in the candidate-based code as the tighter estimates stay within the memory available.

As all regular test matrices are symmetric, we can also compare the old with the new candidate-based LDL^T factorization. The results presented in Table 3.13 confirm those obtained for the LU factorization. The candidate-based code shows a much better performance in particular for the large problems on a large number of processors due to improved locality of communication and computation, and because of the bigger scope for increasing the $k_{\max}$ parameter.

Note that thanks to the improvements in the scalability of the new code, MUMPS now compares favourably to SuperLU on a large number of processors. (The factorization time for SuperLU on 128 processors and the same nested dissection ordering according to [10] is 71.1 seconds for the cubic and 56.1 seconds for the rectangular

	Cubic grids (ND)			Rectangular grids (ND)		
Processors	space used	Estimate old	Estimate new	space used	Estimate old	Estimate new
1	11.4	11.4	11.4	10.2	10.2	10.2
2	19.7	19.7	19.7	17.2	17.2	17.2
4	28.1	28.1	28.1	26.5	26.6	26.6
8	49.1	49.2	49.2	43.9	44.6	44.0
16	77.9	84.3	78.6	70.4	82.3	70.9
32	121.2	181.5	122.8	107.7	166.8	110.0
48	165.9	289.5	170.7	130.2	255.5	134.7
64	193.7	412.6	203.8	158.4	407.2	166.7
128	309.7	897.9	357.0	260.1	1108.0	296.4
256	504.4	2678.5	924.6	353.9	2420.5	478.0
512	780.4	4594.0	1369.7	541.6	5759.0	921.5

Table 3.12: Space for the LU factors (number of reals $\times 10^6$).

	Cubic grids (ND)			Rectangular grids (ND)		
Processors	flops	old	new	flops	old	new
1	3.6e+09	19.1	18.7	2.2e+09	13.5	13.1
2	8.0e+09	21.3	20.7	4.8e+09	13.1	12.9
4	1.3e+10	19.7	16.7	9.0e+09	11.5	12.4
8	3.0e+10	18.1	18.3	1.8e+10	15.2	12.9
16	5.9e+10	18.8	19.8	3.6e+10	13.8	13.2
32	1.1e+11	25.8	22.2	6.8e+10	15.5	15.3
48	1.8e+11	28.7	30.4	9.0e+10	14.2	14.8
64	2.2e+11	30.7	25.6	1.2e+11	17.6	16.8
128	4.4e+11	45.6	33.0	2.4e+11	33.5	20.3
256	9.1e+11	109.1	43.0	3.8e+11	45.2	18.4
512	1.7e+12	421.9	64.0	7.1e+11	195.5	24.3

Table 3.13: Performance of the LDL^T factorization (time in seconds on the CRAY T3E).

grid. MUMPS results are reported in Table 3.11.)

3.10.2 Hybrid nested dissection with SCOTCH

In this section, we consider the previous set of test matrices ordered by SCOTCH [112, 113] which uses a hybrid method of nested dissection and the Halo Approximate Minimum Degree ordering. On regular grids, the SCOTCH ordering usually produces quite well balanced assembly trees but requires more memory so that we are only able to show results for up to 64 processors.

	Cubic grids (SCOTCH)			Rectangular grids (SCOTCH)		
Processors	flops	old	new	flops	old	new
1	6.1e+09	18.7	18.7	3.3e+09	12.0	12.0
2	1.3e+10	21.3	21.3	7.1e+09	13.1	13.1
4	2.4e+10	19.2	19.2	1.3e+10	12.0	12.0
8	6.0e+10	26.1	25.6	3.0e+10	15.3	14.6
16	1.2e+11	32.4	32.0	6.5e+10	18.2	18.6
32	2.5e+11	37.9	38.2	1.1e+11	20.0	20.3
48	3.1e+11	41.4	40.7	1.8e+11	27.0	26.3
64	3.9e+11	46.4	41.4	2.1e+11	26.8	25.7

Table 3.14: Performance of the LU factorization (time in seconds on the CRAY T3E).

For the results of both the LU and the LDL^T factorization presented in Tables 3.14 and 3.15, we do not see a significantly improved scalability of the new code. This is not surprising since we saw in Section 3.10.1 that the major gains came on the largest problems on more than 128 processors. Additionally, the test matrices were designed so that the number of operations per processor is approximately constant when using the nested dissection ordering from Section 3.10.1, but this does not hold for the SCOTCH ordering as we show by the flop counts in the table. We see, for example, that when doubling the number of processors from 4 to 8, the number of operations for the elimination increases by a factor of 2.5 so that we cannot expect the factorization time to be nearly constant.

	Cubic grids (SCOTCH)			Rectangular grids (SCOTCH)		
Processors	flops	old	new	flops	old	new
1	3.0e+09	15.1	15.1	1.7e+09	9.8	9.8
2	6.3e+09	14.9	14.9	3.6e+09	9.8	9.8
4	1.2e+10	13.1	13.2	6.6e+09	9.3	9.2
8	3.0e+10	15.9	16.1	1.5e+10	10.3	9.3
16	6.3e+10	16.8	16.7	3.2e+10	11.2	11.1
32	1.2e+11	19.6	21.5	5.4e+10	12.0	12.6
48	1.6e+11	20.4	22.7	7.0e+10	10.8	11.0
64	1.9e+11	21.8	24.2	1.1e+11	12.6	14.7

Table 3.15: Performance of the LDL^T factorization (time in seconds on the CRAY T3E).

3.10.3 Approximate Minimum Fill (AMF) ordering

Recently a fairly large number of experiments have been conducted with several heuristics to reduce the fill-in (deficiency) during the elimination process [109, 119]. The approximation of the deficiency used in our AMF code is based on the observation that, because of the approximate degree, we count variables twice that belong to the intersection of two elements adjacent to a variable in the current pivot list. This property of the approximate degree can be exploited to improve the estimation of the deficiency and the accuracy of the approximation proposed in [119].

The AMF ordering produces trees that are difficult to exploit in MUMPS. The upper part of the tree where type 2 and type 3 parallelism can be exploited is usually a long and thin chain. In Table 3.16, we show the memory required to store the factors of the LU factorization for the different orderings. In the case of the cubic grid, the real space used by the factors is significantly larger than when using the nested dissection or SCOTCH ordering. Furthermore, we note that for the rectangular grid, AMF actually needs the least space for the factors. However, the shape of the assembly tree still offers less potential for parallelism and we expect the factorization time for AMF-ordered matrices to be considerably longer than for the case of nested dissection or SCOTCH. This is confirmed by the results in Tables 3.17 and 3.18. We remark that, because of the increased space necessary for the factors, we are unable to perform the analysis for the matrix from the largest cubic grid because of insufficient memory.

	AMF	ND	SCOTCH
Grid	Factors	Factors	Factors
Cubic	247,804,999	193,785,687	176,628,109
Rect	148,102,032	158,402,018	154,038,836

Table 3.16: Number of entries in the factors by ordering for the LU factorization on 64 processors, grid sizes according to Table 3.1.

3.10.4 Analysis of the speedup for regular grid problems

We now summarize the results of the previous sections by presenting a comparison of the speedup on the 3D grid problems.

Let t_j denote the time to execute a given job involving ops_j floating point operations on j parallel processors. Then, we define the scaled *speedup*, S_p, for p processors to be

$$S_p = \frac{t_1/ops_1}{t_p/ops_p}. \qquad (3.3)$$

In Figures 3.20 and 3.21, we show the scaled speedup for the matrices ordered by nested dissection, and in Figures 3.22 and 3.23 for SCOTCH and in Figures 3.24

	Cubic grids (AMF)			Rectangular grids (AMF)		
Processors	flops	old	new	flops	old	new
1	8.6e+09	25.7	25.7	3.1e+09	13.4	13.7
2	2.1e+10	47.4	48.3	5.8e+09	21.7	22.1
4	3.8e+10	35.1	35.0	1.0e+10	22.5	23.7
8	1.0e+11	54.5	47.8	2.2e+10	27.8	27.6
16	1.9e+11	55.5	54.9	5.4e+10	34.8	32.6
32	3.8e+11	96.3	81.3	1.0e+11	50.7	49.8
48	4.8e+11	114.6	98.2	1.9e+11	71.0	67.4
64	8.0e+11	188.0	145.4	1.8e+11	46.4	43.3
128	1.7e+12	302.6	242.7	4.6e+11	118.9	114.6
256	4.1e+12	740.9	484.1	8.6e+11	262.5	208.6
512	Not enough memory in analysis			1.2e+12	325.7	264.7

Table 3.17: Performance of the LU factorization (time in seconds on the CRAY T3E).

	Cubic grids (AMF)			Rectangular grids (AMF)		
Processors	flops	old	new	flops	old	new
1	4.3e+09	19.5	19.5	1.6e+09	11.3	11.3
2	1.1e+10	32.8	33.8	2.9e+09	18.6	18.7
4	1.9e+10	24.0	24.6	5.2e+09	19.5	19.9
8	5.1e+10	28.0	27.9	1.1e+10	15.0	14.9
16	9.5e+10	29.0	29.2	2.7e+10	15.4	16.7
32	1.9e+11	34.1	33.8	5.1e+10	20.1	20.9
48	2.4e+11	36.3	36.5	9.3e+10	24.6	25.0
64	4.0e+11	51.8	48.6	8.9e+10	23.6	23.7
128	8.4e+11	86.1	67.8	2.3e+11	38.6	34.5
256	2.1e+12	237.7	117.3	4.3e+11	74.6	67.7
512	Not enough memory in analysis			6.2e+11	196.1	73.0

Table 3.18: Performance of the LDL^T factorization (time in seconds on the CRAY T3E).

and 3.25 for AMF ordering, respectively. We see that the scaled speedup for the new candidate-based code is significantly improved on a large number of processors.

3.10.5 Performance analysis on general symmetric and unsymmetric matrices

In this section, we compare the performance of the new mapping algorithm with the previous version on general symmetric and unsymmetric matrices. The main problem with this comparison is that our algorithm offers the biggest performance

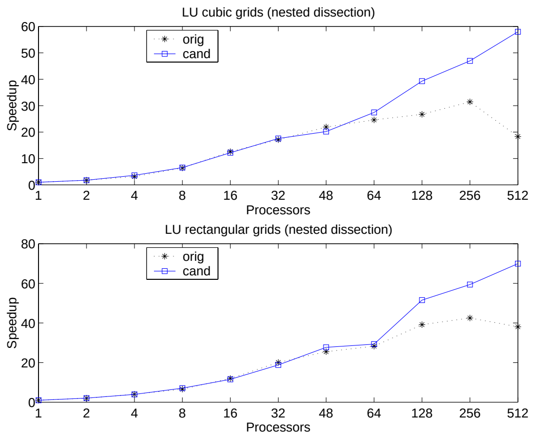

Figure 3.20: Comparison of the speedup of the LU factorization for 3D grid problems ordered by nested dissection.

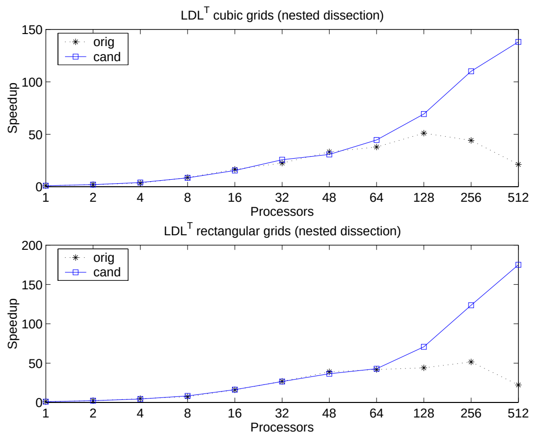

Figure 3.21: Comparison of the speedup of the LDL^T factorization for 3D grid problems ordered by nested dissection.

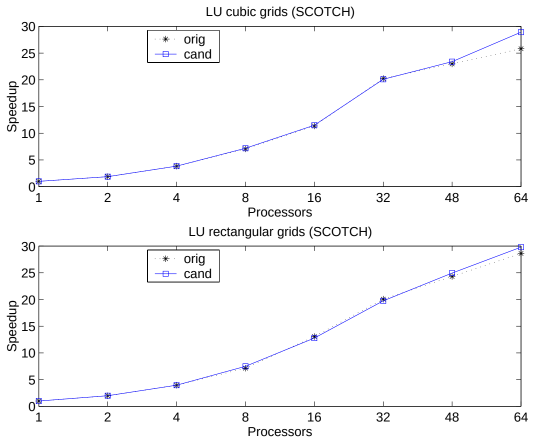

Figure 3.22: Comparison of the speedup of the LU factorization for 3D grid problems ordered by SCOTCH.

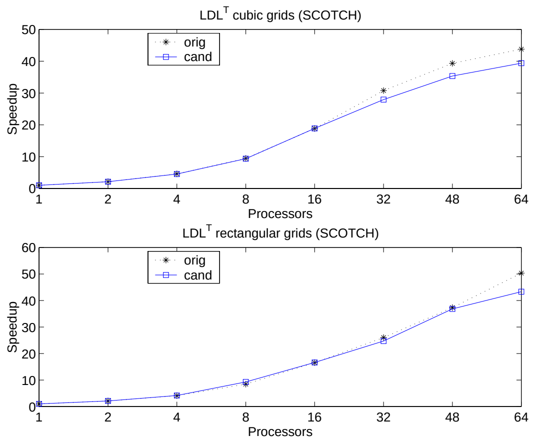

Figure 3.23: Comparison of the speedup of the LDL^T factorization for 3D grid problems ordered by SCOTCH.

gains only on a large number of processors. However, the unsymmetric matrices available to us are either too small to offer enough potential for scalability on more than 64 processors, or they are too large to do the analysis (which is performed on only one processor). This was already observed in the analysis of the scalability of both MUMPS and SuperLU [10] .

In order to compare the quality of the different orderings, we show the number of entries in the factors for the test matrices in Table 3.19.

While it is not always the best ordering, METIS consistently provides a good overall performance with respect to the number of entries in the factors.

From Table 3.20 we see that in general the new mapping algorithm performs similarly to the old one. As already noted, we would expect significant improvements

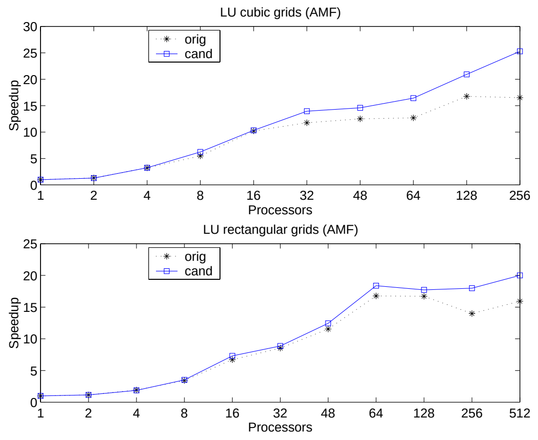

Figure 3.24: Comparison of the speedup of the LU factorization for 3D grid problems ordered by AMF.

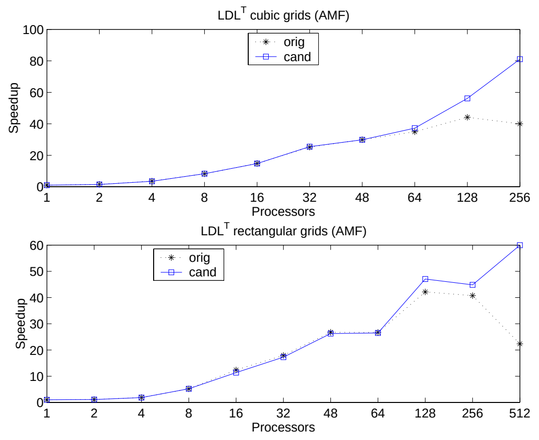

Figure 3.25: Comparison of the speedup of the LDL^T factorization for 3D grid problems ordered by AMF.

Matrix	AMF		METIS		SCOTCH	
name	Factors	Flops	Factors	Flops	Factors	Flops
`bbmat`	37,734,384	2.8e+10	37,429,544	2.8e+10	37,347,812	2.5e+10
`ecl32`	31,862,069	3.5e+10	25,190,381	2.1e+10	29,030,953	2.6e+10
`g7jac200`	33,245,736	3.5e+10	43,496,678	5.5e+10	76,451,656	1.6e+11
`twotone`	22,653,594	2.9e+10	25,537,506	2.9e+10	24,882,282	2.6e+10
`ship003`	68,199,143	9.6e+10	71,388,126	8.3e+10	77,085,965	9.2e+10
`bmwcra_1`	95,816,634	9.9e+10	78,012,686	6.1e+10	140,412,515	2.6e+11

Table 3.19: Number of entries in the factors and number of operations during factorization by ordering (LDL^T factorization for symmetric and LU factorization for unsymmetric matrices).

on large matrices and on a large number of processors greater than 64. However, we notice some improvements for the AMF ordering on `bbmat` and `g7jac200`. But since both METIS and SCOTCH generally provide better orderings, those improvements on AMF are in fact not so relevant and only show the capacity of our algorithm to correctly handle irregular trees.

3.11 Perspectives and future work

We now present possible extensions of our new scheduling algorithm. With the IBM SP, an architecture based on clusters of shared memory (SMP) nodes, we consider a distributed environment where the costs of communications are non-uniform. Here, the candidate concept allows us to enforce locality of communication and thus to improve performance. Then, we discuss other future directions of research.

Matrix	Order	Alg	4	8	16	32	64
`bbmat`	AMF	old	119.7	71.1	50.5	44.3	44.1
		new	114.2	69.6	44.1	27.6	21.7
	METIS	old	39.5	24.2	14.5	11.8	9.6
		new	37.7	22.2	14.1	10.8	8.8
	SCOTCH	old	20.9	13.4	8.3	6.4	5.4
		new	21.0	12.0	7.4	5.6	4.9
`ecl32`	AMF	old	45.2	25.7	19.9	16.6	16.0
		new	44.4	24.2	19.0	16.0	14.5
	METIS	old	28.4	16.7	10.7	7.7	6.3
		new	29.4	16.0	11.4	7.7	5.6
	SCOTCH	old	24.7	13.7	8.6	6.6	5.7
		new	23.0	13.2	9.1	6.6	5.4
`g7jac200`	AMF	old	166.0	77.3	63.4	40.2	41.8
		new	171.6	78.3	61.3	38.6	33.7
	METIS	old	-	48.2	27.4	20.3	15.7
		new	-	41.4	26.7	19.9	13.6
	SCOTCH	old	-	69.2	50.0	43.8	33.3
		new	-	68.3	44.0	35.1	28.2
`twotone`	AMF	old	105.8	47.1	28.3	20.8	19.1
		new	102.4	47.4	29.0	20.9	18.7
	METIS	old	-	26.9	19.1	13.3	11.4
		new	-	27.9	17.7	11.9	11.2
	SCOTCH	old	22.2	13.0	8.8	6.7	6.1
		new	23.8	13.2	9.7	7.1	5.8
`ship003`	AMF	old	-	66.0	34.0	24.4	22.1
		new	-	62.2	33.5	24.2	20.4
	METIS	old	-	-	29.2	18.2	12.3
		new	-	-	28.4	18.0	12.0
	SCOTCH	old	-	-	25.2	15.6	11.5
		new	-	-	23.5	16.7	13.4
`bmwcra_1`	AMF	old	-	-	44.6	30.3	27.6
		new	-	-	42.4	28.5	26.9
	METIS	old	-	36.6	20.1	13.5	8.5
		new	-	35.7	20.9	13.2	8.4
	SCOTCH	old	-	-	-	53.5	30.8
		new	-	-	-	49.9	31.3

Table 3.20: Performance of old and new code on the irregular test matrices (factorization time in seconds on the CRAY T3E).

3.11.1 Adapting the new scheduling algorithm to include communication costs

We discuss an approach for SMP architectures presented in [116]. In order to take account of the system architecture, both the task mapping during the analysis phase and the task scheduling during the factorization phase are slightly modified. In the analysis phase, the mapping is changed to decrease the costs of the communications associated with the static mapping. For the dynamic choice of the type 2 slaves in the factorization phase, processors which would require expensive communications are penalized so that the master-slave communication costs are reduced.

We first describe the modifications to the mapping decisions during the analysis phase. As discussed in Section 3.7.4, the mapping of a task is done via a preference list giving priority to the preferential processors from the proportional mapping. The proportional mapping is modified so that the recursive assignment of the processors to subtrees also takes account of communication costs. It turns out that a small modification suffices to adapt the assignment of the preferential processors. Instead of assigning the preferentials cyclicly by processor numbers as described in Section 3.7.1, we can also perform the cyclic assignment from a processor list that is *ordered* so that the average communication cost between list neighbours is as small as possible. As an example, we consider a simple communication cost model where data movement within an SMP node is cheap, and where it is expensive between two SMP nodes. Then, a well ordered list holds the processors of one SMP node next to each other, followed by those of another SMP node, a further SMP node, and so forth. Ideally, the cyclic assignment of the preferentials from the ordered processor list will then map subtrees to SMP nodes and limit expensive communications between different nodes.

The modifications to the scheduling decisions during the factorization phase are as follows. As described in Section 3.6.2, all candidates that are less loaded than the master of a type 2 node are chosen as its slaves. In order to give preference to the candidates within the SMP node of the master, the workload information for the processors outside the SMP node is modified so that expensive communication is penalized. When the master selects its slaves from among the candidates, it uses the modified workload information as a basis for its decision. Thus, when a master chooses its slaves, it gives priority to those processors that are located on the same SMP node.

		ND			SCOTCH			AMF		
Proc	grid	old	new	ext.	old	new	ext.	old	new	ext.
16	$184 \times 46 \times 23$	34.9	35.2	35.2	36.2	37.9	37.9	34.0	43.3	43.3
32	$208 \times 52 \times 26$	66.9	59.7	49.4	72.1	63.3	60.5	101.5	82.4	71.6
64	$224 \times 56 \times 28$	95.4	70.1	56.8	93.2	76.6	68.6	212.2	199.0	163.9
128	$248 \times 62 \times 31$	303.8	97.5	83.8	365.9	136.1	111.9	544.6	298.0	212.0

Table 3.21: Performance of the old, the new, and the extended implementation of the *LU* factorization on 3D rectangular grids problems (factorization time in seconds on the IBM SP3).

In the following, we present some results obtained in [116] that show the performance improvements of this extension of the new scheduling algorithm. The experiments are conducted on an IBM SP3 at CINES which consists of 29 SMP nodes of 16 processors each. For the analysis phase, the simple two-cost model described above is used. In the factorization phase, the workload penalty function for processors outside of the master's SMP node depends on the size of the message to be communicated. For small messages, the penalty function is linear in the workload of the processor. For large messages, the function practically excludes all processors outside of the SMP node from being a slave (when the number of candidates from within the node already satisfies the restriction (3.2) imposed by $k_{\max}$, see Section 3.7.6.)

		ND			SCOTCH			AMF		
Proc	grid	old	new	ext.	old	new	ext.	old	new	ext.
16	57^3	72.6	70.8	70.8	56.7	60.9	60.9	131.4	132.9	132.9
32	64^3	140.1	136.2	120.8	96.5	86.8	82.4	304.3	264.8	280.8
64	72^3	243.7	229.8	204.2	252.7	229.8	204.2	755.8	686.5	626.2
128	80^3	890.2	446.7	366.1	896.8	294.6	272.9	2190.8	1250.0	1011.4

Table 3.22: Performance of the old, the new, and the extended implementation of the *LU* factorization on 3D cubic grids problems (factorization time in seconds on the IBM SP3).

Table 3.21 shows the results for the rectangular grids; the results for the cubic grids are presented in Table 3.22. We compare the old version of MUMPS with the new candidate-based version and the new extended version which takes account of communication costs. In order to increase the volume of communication and because of the greater amount of memory available, the grid size with respect to Table 3.1 has been modified: the grids for the 16 processors of the IBM SP correspond to those used on 64 processors of the CRAY T3E, the 32 processor grids for the IBM SP are those used on 128 processors of the T3E, and so on.

The experimental results on 16 processors are obtained on a single SMP node where the new extended version does not bring benefits. Looking at the set of results reported, it can be seen that the gains from the new candidate-based algorithm can again be increased through the new extended version.

3.11.2 Future directions of research

In this section, we summarize the open questions that need further investigation.

In Section 3.9.3, we investigated the behaviour of the new code when modifying the assignment of candidates through relaxation and layer-wise redistribution. On the test cases that we have studied in the framework of this chapter of the thesis, these modifications have not shown a positive effect on the overall performance of the code. Still, there is an intuitive argument suggesting further experiments. The analysis phase tries to predict the actual factorization of the matrix and takes mapping decisions based on this symbolic factorization. However, there are cases where this approach might not be accurate enough. First, one example where the symbolic factorization is inaccurate is the case of delayed pivots. Second, another problem is our assumption during the analysis that all candidate processors of type 2 nodes can be chosen as slaves during factorization, which might not be the case. Third, and probably the most critical, is the fact that our static mapping decisions are based on flop equilibration which might not accurately model the time. For example, we do not take into account costs of communication between the processors as is done, for example, by the static scheduler of PaStiX [79, 80, 81]. Thus, mapping decisions might need to be corrected, for example, by the dynamic scheduler. An approach combining the techniques presented in this chapter of the thesis could result from the following observation. Since, during factorization, the assembly tree is treated from bottom up, we might expect mapping problems to have more severe influence towards the root of the tree. For this reason, we could decide to offer more freedom to dynamic scheduling near the root nodes so that unfortunate mapping decisions can be corrected dynamically there.

Furthermore, we have presented in Section 3.10.5 test results on a few large irregular test matrices from real life applications. We have already remarked that these matrices are still relatively small and do not offer enough sources of parallelism on a large number of processors. This study needs to be extended in order to be able to give reliable statements on the scalability of the new code also in real life applications.

3.12 Summary and conclusions

Previous studies of MUMPS, a distributed memory direct multifrontal solver for sparse linear systems, indicated that its scalability with respect to computation time and use of memory should be improved. In this chapter of the thesis, we have presented a new task scheduling algorithm designed to address these problems. It consists of an approach that treats the assembly tree layer by layer and integrates tree modifications, such as amalgamation and splitting, with the mapping decisions. As a major feature, we have introduced the concept of candidate processors that are determined during the analysis phase of the solver in order to guide the dynamic scheduling during the factorization.

We have illustrated key properties of the new algorithm by detailed case studies on selected problems. Afterwards, by comparison of the old with the new code on a large set of regular and irregular test problems, we have illustrated the main benefits of the new approach. These include an improved scalability on a large number of processors, reduced memory demands and a smaller volume of communication, and the easier handling of parameters relevant for the performance of the algorithm. Finally, we have discussed possible extensions of our algorithm, in particular with respect to its use on SMP architectures.

We remark that the semi-static approach of our new scheduling algorithm could also be used for sparse linear solvers other than MUMPS. As long as such a solver is driven by an assembly tree and uses dynamic scheduling, we can implement our two-phase approach. In a first static phase, choices taking account of global tree information are made that will afterwards, in the second phase, influence and guide dynamic decisions. As in the context of MUMPS, this can improve the performance of the solver through better control of activities related to dynamic scheduling.

Bibliography

[1] BLAS Technical Forum Standard. The International Journal of High Performance Computing Applications 15(3-4), 2001.

[2] BLAS (Basic Linear Algebra Subprograms). `http://www.netlib.org/blas`, 2002.

[3] PARASOL test data. `http://www.parallab.uib.no/parasol/data.html`, 2002.

[4] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.*, 14(2):446–460, 1993.

[5] P. R. Amestoy. *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment.* Ph.D. thesis, Institut National Polytechnique de Toulouse, 1991. Available as CERFACS report TH/PA/91/2.

[6] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17:886–905, 1996.

[7] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Appl.*, 7:64–82, 1993.

[8] P. R. Amestoy, I. S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Appl. Mech. Eng.*, pages 501–520, 2000.

[9] P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[10] P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and X. S. Li. Analysis, Tuning and Comparison of Two General Sparse Solvers for Distributed Memory Computers. *ACM Trans. Math. Software*, 27(4):388–421, 2001.

[11] P. R. Amestoy, I. S. Duff, and C. Vömel. Task scheduling in an asynchronous distributed memory multifrontal solver. Technical Report TR/PA/02/105,

CERFACS, Toulouse, France, 2002. Submitted to SIAM Journal of Matrix Analysis and Applications.

[12] P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM J. Matrix Anal. Appl.*, 24:553–569, 2002.

[13] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 3. edition, 1999.

[14] C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[15] C. Ashcraft and R. G. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[16] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector computers. *Int. J. Supercomputer Appl.*, 1(4):10–30, 1987.

[17] C. Ashcraft and J. W. H. Liu. Robust ordering of sparse matrices using multisection. *SIAM J. Matrix Anal. Appl.*, 19(3):816–832, 1998.

[18] R. E. Bank and C. C. Douglas. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics*, 1:127–137, 1993.

[19] R. H. Bartels and G. H. Golub. The simplex method of linear programming using LU decompositions. *Comm. ACM*, 12:266–268, 1969.

[20] M. Benzi and C. D. Meyer. A direct projection method for sparse linear systems. *SIAM J. Sci. Comput.*, 16:1159–1176, 1995.

[21] M. Benzi and M. Tuma. Orderings for factorized sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21:1851–1868, 2000.

[22] C. H. Bischof. Incremental condition estimation. *SIAM J. Matrix Anal. Appl.*, 11:312–322, 1990.

[23] C. H. Bischof, D. J. Pierce, and J. G. Lewis. Incremental condition estimation for sparse matrices. *SIAM J. Matrix Anal. Appl.*, 11:644–659, 1990.

[24] L. S. Blackford, J. Demmel, J. J. Dongarra, I. S. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software*, 28(2):135–151, 2002.

[25] R. Bridson and W.-P. Tang. Ordering, anisotropy and factored approximate inverses. *SIAM J. Sci. Comput.*, 21:867–882, 1999.

[26] T. F. Chan. On the existence and computation of LU-factorizations with small pivots. *Math. Comp.*, 42:535–547, 1984.

[27] T. F. Chan. Rank revealing QR-factorizations. *Linear Algebra and Appl.*, 88/89:67–82, 1987.

[28] S. Chandrasekaran and I. Ipsen. On rank-revealing QR factorizations. *SIAM J. Matrix Anal. Appl.*, 15(2):592–622, 1994.

[29] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also as LAPACK Working Note #95).

[30] A. K. Cline, C. B. Moler, G. W. Stewart, and J. H. Wilkinson. An estimate for the condition number of a matrix. *SIAM J. Numer. Anal.*, 16:368–375, 1979.

[31] G. Cybenko. Fast Toeplitz orthogonalization using inner products. *SIAM J. Sci. Stat. Comput.*, 8:734–740, 1987.

[32] T. A. Davis. University of Florida sparse matrix collection, 2002. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[33] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse *LU* factorization. *SIAM J. Matrix Anal. Appl.*, 18(1):140–158, 1997.

[34] T. A. Davis and I. S. Duff. A Combined Unifrontal/Multifrontal Method for Unsymmetric Sparse Matrices. *ACM Trans. Math. Software*, 25(1):1–19, 1999.

[35] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, 1999.

[36] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, 1999.

[37] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Algorithm 692: Model implementation and test package for the Sparse Basic Linear Algebra Subroutines. *ACM Trans. Math. Software*, 17(2):264–272, 1991.

[38] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 17(2):253–263, 1991.

[39] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[40] J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. J. Hanson. An extented set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–17, 1988.

[41] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* SIAM Press, Philadelphia, 1998.

[42] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[43] J. J. Dongarra and R. C. Whaley. A user's guide to the blacs v1.0. Technical Report UT CS-95-281, University of Tennessee, 1995. (also as LAPACK Working Note #94).

[44] J. J. Dongarra and R. C. Whaley. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, University of Tennessee, 1997.

[45] J. J. Du Croz and N. J. Higham. Stability of methods for matrix inversion. *IMA J. Numer. Anal.*, 12:1–19, 1992.

[46] I. S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. In I. S. Duff and G. A. Watson, editors, *The State of the Art in Numerical Analysis*, pages 27–62, Oxford, 1997. Oxford University Press.

[47] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, London, 1986.

[48] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[49] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report RAL-TR-97-031, Atlas Centre, Rutherford Appleton Laboratory, 1997. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.

[50] I. S. Duff, M. Heroux, and R. Pozo. The Sparse BLAS. *ACM Trans. Math. Software*, 28(2):239–267, 2002.

[51] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[52] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2001.

[53] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 Basic Linear Algebra Subprograms for sparse matrices: a user level interface. *ACM Trans. Math. Software*, 23(3):379–401, 1997.

[54] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Software*, 9:302–325, 1983.

[55] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM J. Sci. Stat. Comput.*, 5:633–641, 1984.

[56] I. S. Duff and H. A. van der Vorst. Developments and trends in the parallel solution of linear systems. *Parallel Computing*, 25(13-14):1931–1970, 1999.

[57] I. S. Duff and C. Vömel. Algorithm 818: A Reference Model Implementation of the Sparse BLAS in Fortran 95. *ACM Trans. Math. Software*, 28(2):268–283, 2002.

[58] I. S. Duff and C. Vömel. Algorithm 818: A Reference Model Implementation of the Sparse BLAS in Fortran 95. `http://www.netlib.org/netlib/toms/818`, 2002.

[59] I. S. Duff and C. Vömel. Incremental Norm Estimation for Dense and Sparse Matrices. *BIT*, 42(2):300–322, 2002.

[60] W. R. Ferng and D. Pierce. Incremental Lanczos Condition Estimation (or The Robustification of ICE). Technical Report AMS-TR-184, Boeing Computer Services, 1992.

[61] J. J. H. Forrest and J. A. Tomlin. Updating triangular factors of the basis to maintain sparsity in the product form simplex method. *Math. Prog.*, 2(3):263–278, 1972.

[62] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman, New York, 1979.

[63] A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

[64] A. George and E. Ng. SPARSPAK: Waterloo sparse matrix package user's guide for SPARSPAK-B. Research Report CS-84-37, Dept. of Computer Science, University of Waterloo, 1984.

[65] J. A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1971.

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

[67] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 15(1):62–79, 1994.

[68] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Anal. Appl.*, 14(2):334–352, 1993.

[69] G. H. Golub and H. A. van der Vorst. Eigenvalue Computations in the 20th Century. *J. Comp. Appl. Math.*, 123:35–65, 2000.

[70] G. H. Golub and C. van Loan. *Matrix Computations.* The John Hopkins University Press, Baltimore, Maryland, 3. edition, 1996.

[71] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of sparse matrix reordering techniques on the memory usage of a parallel multifrontal solver. In *Proceedings of the 2nd International Workshop on Parallel Matrix Algorithms and Applications (PMMA'02)*, 2002.

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

[73] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. Math. Software*, 28(3):301–324, 2002.

[74] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel and Distributed Systems*, 8:502–520, 1997.

[75] W. W. Hager. Condition estimates. *SIAM J. Sci. Stat. Comput.*, 5(2):311–316, 1984.

[76] M. T. Heath, E. G. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[77] B. Hendrickson and R. Leland. The CHACO User's Guide. Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, 1994.

[78] B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2):468–489, 1999.

[79] P. Hénon, P. Ramet, and J. Roman. A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization. In *Proceedings of EuroPAR'99, Toulouse, France*, number 1685 in Lecture Notes in Computer Science, pages 1059–1067. Springer Verlag, 1999.

[80] P. Hénon, P. Ramet, and J. Roman. PaStiX: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Proceedings of Irregular'2000, Cancun, Mexique*, number 1800 in Lecture Notes in Computer Science, pages 519–525. Springer Verlag, 2000.

[81] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, 2002.

[82] M. Heroux and R. Pozo. Personal communication.

[83] N. J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29:575–596, 1987.

[84] N. J. Higham. The Test Matrix Toolbox for MATLAB (Version 3.0). Numerical Analysis Report No. 276, Manchester Centre for Computational Mathematics, Manchester, England, 1995.

[85] N. J. Higham. Iterative refinement for linear systems and LAPACK. *IMA J. Numer. Anal.*, 17(4):495–509, 1997.

[86] N. J. Higham. *Accuracy and Stabilty of Numerical Algorithms.* SIAM, Philadelphia, 2. edition, 2002.

[87] N. J. Higham and A. Pothen. Stability of the partitioned inverse method for parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.*, 15(1):139–148, 1994.

[88] Nicholas J. Higham. Algorithm 674: FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Software*, 14(4):381–396, 1988.

[89] D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*, chapter Approximation algorithms for scheduling, pages 1–45. PWS Publishing, Boston, 1996.

[90] A. S. Householder. *The Theory of Matrices in Numerical Analysis.* Blaisdell Publishing Co., New York, 1964.

[91] HSL. HSL 2002: A collection of Fortran codes for large scale scientific computation, 2002. `http://www.cse.clrc.ac.uk/nag/hsl`.

[92] T.-M. Hwang, W.-W. Lin, and D. Pierce. Improved Bound for Rank Revealing LU-factorizations. *Linear Algebra and Appl.*, 261:173–186, 1997.

[93] T.-M. Hwang, W.-W. Lin, and E. K. Yang. Rank Revealing LU-factorizations. *Linear Algebra and Appl.*, 175:115–141, 1992.

[94] E. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication.* PhD thesis, University of California, Berkeley, 2000.

[95] E. Im and K. Yelick. Optimizing Sparse Matrix-Vector Multiplication on SMPs. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1999.

[96] E. Im and K. Yelick. Optimizing Sparse Matrix-Vector Multiplication for Register Reuse in SPARSITY. In *International Conference on Computational Science, San Francisco, California*, pages 127–136, 2001.

[97] W. Kahan. Numerical linear algebra. *Canadian Mathematical Bulletin*, 9:757–801, 1966.

[98] G. Karypis and V. Kumar. *MeTis - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices - Version 4.0.* University of Minnesota, 1998.

[99] J. Koster. *On the parallel solution and the reordering of unsymmetric sparse matrices.* Ph.D. thesis, Institut National Polytechnique de Toulouse, 1997. Available as CERFACS report TH/PA/97/51.

[100] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[101] A. Legrand and Y. Robert. *Algorithmique Parallèle – Cours et exercices corrigés.* Dunod, 2002.

[102] X. S. Li and J. 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, 1999.

[103] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Trans. Math. Software*, 28(2):152–205, 2002.

[104] J. W. H. Liu. Equivalent sparse matrix reordering by elimination tree rotations. *SIAM J. Sci. Stat. Comput.*, 9:424–444, 1988.

[105] J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl.*, 11:134–172, 1990.

[106] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.

[107] J. W. H. Liu, E. G. Ng, and W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14:242–252, 1993.

[108] C. D. Meyer and D. Pierce. Steps towards an iterative rank-revealing method. Technical Report ISSTECH-95-013, Boeing Information and Support Services, 1995.

[109] E. Ng and P. Raghavan. Performance of greedy heuristics for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 20:902–914, 1999.

[110] C.-T. Pan. On the existence and computation of rank-revealing LU factorizations. *Linear Algebra and Appl.*, 316:199–222, 2000.

[111] C.-T. Pan and R. J. Plemmons. Least squares modifications with inverse factorizations: parallel implications. *J. Comp. Appl. Math.*, 27:109–127, 1989.

[112] F. Pellegrini and J. Roman. Sparse matrix ordering with Scotch. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, April 1997.

[113] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12(2-3):69–84, 2000.

[114] D. J. Pierce and J. G. Lewis. Sparse multifrontal rank revealing QR factorization. *SIAM J. Matrix Anal. Appl.*, 18(1):159–180, 1997.

[115] A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM J. Sci. Comput.*, 14(5):1253–1257, 1993.

[116] S. Pralet. Study of a parallel sparse direct linear solver on an SMP architecture. Technical report, CERFACS, Toulouse, France, 2002. (in preparation).

[117] P. Raghavan. *Distributed sparse matrix factorization: QR and Cholesky decompositions*. Ph.D. thesis, Department of Computer Science, Pennsylvania State University, 1991.

[118] K. A. Remington and R. Pozo. NIST Sparse BLAS user's guide. Internal Report NISTIR 6744, National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2001.

[119] E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal. Appl.*, 19(3):682–695, 1998.

[120] Y. Saad. SPARSKIT: A basic tool kit for sparse computations, VERSION 2. Technical report, Computer Science Department, University of Minnesota, June 1994.

[121] Y. Saad and H. A. van der Vorst. Iterative Solution of Linear Systems in the 20-th Century. *J. Comp. Appl. Math.*, 123:1–33, 2000.

[122] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Math. Software*, 8:256–276, 1982.

[123] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.

[124] G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM J. Numer. Anal.*, 17:403–404, 1980.

[125] G. W. Stewart. Incremental condition calculation and column selection. Technical Report TR-90-87, Institute for Advanced Computer Studies, University of Maryland, College Park, MD20742, 1990.

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