JCTC Journal of Chemical Theory and Computation

Fast Sparse Cholesky Decomposition and Inversion using Nested Dissection Matrix Reordering

Kai Brandhorst*,† and Martin Head-Gordon*,†,‡

Department of Chemistry, University of California, Berkeley, California 94720, United States, and Chemical Sciences Division, Lawrence Berkeley National Laboratory, Berkeley, California, United States

Received October 29, 2010

Abstract: Here we present an efficient, yet nonlinear scaling, algorithm for the computation of Cholesky factors of sparse symmetric positive definite matrices and their inverses. The key feature of this implementation is the separation of the task into an algebraic and a numeric part. The algebraic part of the algorithm attempts to find a reordering of the rows and columns which preserves at least some degree of sparsity and afterward determines the exact nonzero structure of both the Cholesky factor and its corresponding inverse. It is based on graph theory and does not involve any kind of numerical thresholding. This preprocessing then allows for a very efficient implementation of the numerical factorization step. Furthermore this approach even allows use of highly optimized dense linear algebra kernels which leads to yet another performance boost. We will show some illustrative timings of our sparse code and compare it to the standard library implementation and a recent sparse implementation using thresholding. We conclude with some comments on how to deal with positive semidefinite matrices.

1. Introduction

The Cholesky factorization has become increasingly significant in quantum chemistry, especially with respect to applications where large sparse matrices occur. This can be attributed to the fact that a Cholesky factor of a sparse symmetric positive definite matrix usually retains some degree of sparsity that can be exploited in low-order scaling algorithms.

Beebe and Linderberg[1] were probably the first who utilized the Cholesky factorization of the two-electron integral matrix in order to achieve savings in computational time for its generation and transformation. After this seminal work other groups adopted this approach,[2] and it has proven to be useful for factorizing overlap[3,4] and density matrices[5,6] in order to generate sparse transformation matrices. Furthermore, Cholesky factorizations have been used for coordinate transformations,[7] factorization of the amplitude matrix in scaled-opposite-spin MP2 (SOS-MP2)[8,9] and generation of auxiliary basis sets.[10]

While the computational savings gained by factorizing two-electron integral and density matrices stem from the fact that both are semidefinite and thus their Cholesky factors have less columns than rows, overlap matrices are strictly positive definite as long as there are no linear dependencies among the basis functions, and computational savings have been obtained by preserving the sparsity during the factorization. This approach is particularly useful for density matrix-based schemes[3,11−14] where transformations between co- and contravariant quantities are necessary.[3,13,15] For these transformations, the inverse of the metric, i.e., the inverse overlap matrix, is required, which in a dense implementation scales cubically with respect to the size of the matrix.

As two-center overlap matrices in an atomic orbital base tend to become very sparse in the large molecule limit, by contrast to their inverses, their Cholesky factors (or square roots)[16] may retain at least some degree of sparsity. It has already been pointed out[3] that this factorization can be done very efficiently by exploiting sparsity, and recently the group of Ochsenfeld[4] has devised an algorithm which is able to

* Corresponding author. E-mail: k.brandhorst@berkeley.edu; mhg@cchem.berkeley.edu.

† Department of Chemistry, University of California.

‡ Chemical Sciences Division, Lawrence Berkeley National Laboratory.

10.1021/ct100618s © 2011 American Chemical Society
Published on Web 01/10/2011

compute a sparse Cholesky factorization and its inverse by neglecting values that fall below a specified threshold, which they claim to be asymptotically linear scaling. Although this algorithm performs quite well for very sparse matrices, the performance of routines that exploit sparsity is of utmost importance in that respect, that the crossover in runtime between them and their usually highly optimized dense equivalents has to occur reasonably early in order to be advantageous. The aim of this report will be to present a more precise and yet more efficient algorithm that relies on purely algebraic methods based on graph theory, which allows the prediction of the exact nonzero structure of the Cholesky factor and its inverse before the actual numerical factorization is started. By avoiding numerical thresholding, we are even able to employ highly optimized linear algebra kernels from the BLAS[17] and LAPACK[18] libraries. We have implemented all of these improvements in a library and included it into a developer version of Q-CHEM.[19]

We want to stress that this report is more or less a brief introductory review of established mathematical methods[20] and efficient libraries for sparse Cholesky factorization have already been developed (e.g., CHOLMOD[21] and PARDISO).[22] However, none of these libraries is able to compute a sparse inverse of a Cholesky factor which is often required in quantum chemical calculations, e.g., for the transformation between orthogonalized and regular atomic orbital basis sets.[3,12,23]

2. Theory

A symmetric $N \times N$ matrix $\mathbf{A}$ is positive definite if all its eigenvalues are positive. In the case that some of the eigenvalues are zero, the matrix is positive semidefinite. Every positive semidefinite matrix $\mathbf{A}$ can be decomposed into the form $\mathbf{A} = \mathbf{X}\mathbf{X}^{\mathrm{T}}$, with $\mathbf{X}$ having full rank if $\mathbf{A}$ is positive definite and reduced rank if $\mathbf{A}$ is positive semidefinite.[24] Among the infinite number of possible matrices $\mathbf{X}$, however, there exists a unique triangular matrix $\mathbf{L}$. This particular matrix is called the Cholesky factor[24] of $\mathbf{A}$, and its elements $L_{i,j}$ are algebraically given by:

$$L_{i,j} = \begin{cases} \frac{1}{L_{j,j}}\left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k}\right), & i > j,\ L_{j,j} \neq 0 \\ 0, & i > j,\ L_{j,j} = 0 \\ \sqrt{A_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2}, & i = j \\ 0, & i < j \end{cases} \quad (1)$$

We want to stress that the case $L_{j,j} = 0$ only happens if the matrix to be composed is semidefinite, and although the given formula algebraically holds true for any positive (semi)definite matrix $\mathbf{A}$, it is not recommended to use it for factorizing semidefinite matrices in the presence of rounding error.[25] This is due to the fact that rounding errors can accumulate and lead to almost arbitrary results. One thus has to use pivoting techniques in order to get meaningful results,[25] and we will come back to this point later. For now we will assume the matrix $\mathbf{A}$ to be strictly positive definite.

Most often the Cholesky factorization is used for solving linear sets of equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{y} \quad (2)$$

where $\mathbf{A}$ is a positive definite coefficient matrix, $\mathbf{y}$ denotes one or more right-hand side vectors, and $\mathbf{x}$ is the solution to be determined. While the naïve solution to this problem can be found by multiplying eq 2 from the left by $\mathbf{A}^{-1}$

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} \quad (3)$$

and thus requires the explicit formation of the inverse of $\mathbf{A}$, this way is not recommended for practical applications for several reasons: The computation of the inverse is a quite expensive task from a computational point of view. The inverse is usually completely dense[26] even if $\mathbf{A}$ is sparse, and this approach is not backward stable[27] and can thus introduce rounding errors.

For a strictly positive definite matrix $\mathbf{A}$, however, the computationally more efficient way[28] for solving eq 2 consists of computing the Cholesky factorization of $\mathbf{A}$

$$\mathbf{L}\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y} \quad (4)$$

and finding the solution by forward and back substitution.[29] This approach is far superior to computing the inverse $\mathbf{A}^{-1}$ explicitly since computing the Cholesky factor is much less demanding. (The most efficient way of computing an inverse of a symmetric positive definite matrix is by computing its Cholesky factor, inverting it, and forming $\mathbf{A}^{-1} = \mathbf{L}^{\mathrm{T}-1}\mathbf{L}^{-1}$.) Furthermore, this approach is backward stable, and thus the solution obtained by forward and back substitution is usually more accurate when computed in the presence of rounding error. The most appealing feature of this approach however is the fact, that by contrast to inverses, Cholesky factors usually retain some degree of sparsity if the matrix to be decomposed is already sparse.

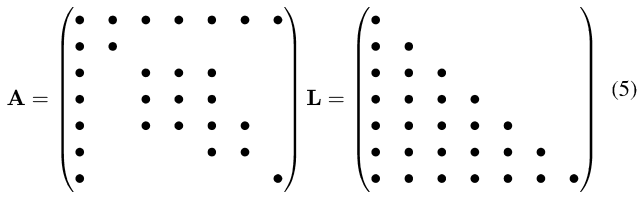
3. Sparse Cholesky Factorization

As stated above, Cholesky factors of sparse matrices tend to remain quite sparse, although they are usually not as sparse. This is due to the effect of fill-in, i.e., some elements that have been zero in the symmetric matrix become nonzero in the Cholesky factor.

3.1. Fill-in. In order to understand fill-in we need to take a closer look at eq 1. Column j of the Cholesky factor depends on the elements of all previous columns of the Cholesky factor, since these terms appear in the sum $\Sigma_{k=1}^{j-1} L_{i,k}L_{j,k}$. Note that instead of performing this summation in a single step right before the factorization of column j, we could also have performed a rank update to the right after the factorization of all columns 1, ..., $j - 1$. If the columns are factorized one at a time, then the rank-1 update is just the product of the vector with elements $(L_{i+1,j}, ..., L_{N,j})$ times its transpose. This product results in a matrix which has to be subtracted from the lower right-hand submatrix of $\mathbf{A}$.

To illustrate this effect consider the schematic representation of a sparse matrix $\mathbf{A}$ and its Cholesky factor $\mathbf{L}$. Since the algebraic nonzero structure of the Cholesky factor does not depend on the actual numerical values of $\mathbf{A}$, we simply use the symbol • to indicate nonzeros:

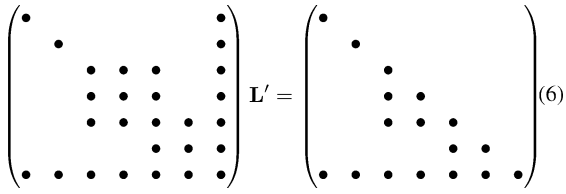
$$\mathbf{A} = \quad \mathbf{L} = \quad (5)$$



Although **A** has at least some degree of sparsity, its Cholesky factor **L** is completely dense. This is due to the rank update applied after having factorized the first column. Since the first column of **A** is dense, the product of the first column without the diagonal element times its transpose results in a completely dense matrix which has to be subtracted from the lower right-hand side submatrix of **A**. Thus the factorization continues on a completely dense matrix, resulting in a dense Cholesky factor **L**.

As a second example consider the Cholesky factorization of the matrix **A′**:

$$\mathbf{A}' = \quad \mathbf{L}' = \quad (6)$$



Now the Cholesky factor **L′** has the same nonzero structure as the matrix **A′**. As the reader can verify, all matrices used for the rank updates have nonzeros in positions that are already nonzero in **A′**, and thus no fill-in occurs at all.

From these two simple examples it becomes clear that the degree of sparsity in the Cholesky factor not only depends on the number of nonzeros in the matrix being decomposed but also strongly depends on the nonzero pattern.

3.2. Permutation Matrices. Of course the matrices **A** and **A′** are quite similar. In fact they are related by the unitary transformation

$$\mathbf{A}' = \mathbf{PAP}^{\mathbf{T}} \quad (7)$$

with

$$\mathbf{P} = \begin{pmatrix} & & & & & & 1 \\ & 1 & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ 1 & & & & & & \end{pmatrix} \quad (8)$$

Matrices like **P** are usually called permutation matrices, since if they are applied to a matrix **A** according to eq 7, then they effectively permute rows and columns. Permutation matrices themselves can be generated by exchanging rows or columns of the identity matrix **I**.

We have already seen that the Cholesky factor **L′** of **A′** does not suffer from any fill-in, while on the other hand **L** is completely dense. Now suppose that we want to solve the linear set of eq 2 with matrix **A** from eq 5 via a Cholesky factorization and a forward/back substitution. Obviously, we cannot take advantage of sparsity in the Cholesky factors right away. However, due to $\mathbf{P}^{\mathrm{T}}\mathbf{P} = \mathbf{I}$, we can reformulate the set of equations:

$$\mathbf{P}^{\mathrm{T}}\mathbf{PAP}^{\mathrm{T}}\mathbf{Px} = \mathbf{y} \quad (9)$$

Since eq 9 is exactly equal to eq 2, one can solve the equivalent linear set of equations

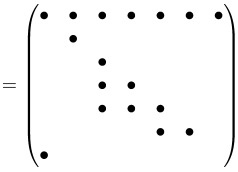
$$\mathbf{PAP}^{\mathrm{T}}\mathbf{Px} = \mathbf{Py} \quad (10)$$

$$\mathbf{A}'\mathbf{x}' = \mathbf{y}' \quad (11)$$

by applying the permutation to the right-hand side and computing the Cholesky factorization $\mathbf{A}' = \mathbf{L}'\mathbf{L}'^{\mathrm{T}}$ instead. This now allows to take full advantage of the sparsity of **A′**, since **L′** does not suffer from any fill-in during its generation. Of course we now obtain a different solution **x′**, however, the solution **x** can easily be derived by unapplying the permutation $\mathbf{x} = \mathbf{P}^{\mathrm{T}}\mathbf{x}'$.

Our focus however, is not the efficient solution of linear sets of equations but the computation of sparse Cholesky factors and their inverses, and of course **L′** is not the Cholesky factor of **A**, nevertheless

$$\mathbf{P}^{\mathrm{T}}\mathbf{L}' = \quad (12)$$

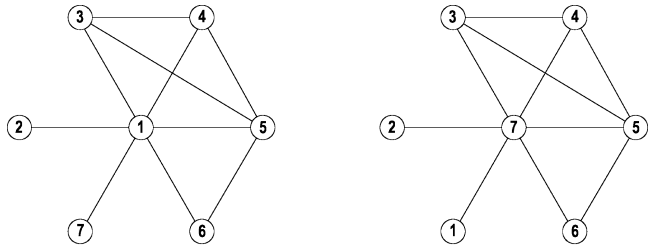


is a sparse, though nontriangular, matrix **X** which fulfills $\mathbf{A} = \mathbf{XX}^{\mathrm{T}}$. Often, this property is sufficient for **X** being useful as a transformation matrix, and it is more important that it is as sparse as possible.

As shown in this simple example, choosing an appropriate permutation matrix **P** can have a dramatic effect on the amount of fill-in occurring during the Cholesky factorization, and we will now outline how such permutation matrices can be found. We would like to stress that we have not made any assumptions on the actual numerical entries of **A** other than that the matrix be positive definite, and we will continue to do so. In what follows we will illustrate that knowing the nonzero structure of a sparse symmetric positive matrix **A** suffices to determine a permutation that results in a fairly low amount of fill-in during the factorization.

3.3. Symbolic Cholesky Factorization. The task of finding a fill-reducing permutation matrix **P** and the prediction of the exact nonzero pattern of a Cholesky factor is commonly termed *symbolic Cholesky factorization*,[20,30] since all these steps can be carried out by using purely algebraic methods from graph theory. Knowing the precise structure of Cholesky factors enables the design of efficient codes for their computation, since this allows allocation of only the actually required amount of memory which can even be done in a single step, and expensive numerical thresholding is not required at all. Furthermore, by using the so-called *supernode*[21,31,32] technique, we will show how highly optimized dense level 3 BLAS[17] and LAPACK[18] kernels can be employed even for the factorization of sparse matrices.

3.3.1. Graph Theory. Undirected graphs are useful tools in the study of symmetric matrices. Any given sparse



(a) Graph for matrix **A** (b) Graph for matrix **A′**

Figure 1. Connectivity graphs for the symmetric matrices before and after applying a permutation.

symmetric matrix **A** can be structurally represented by its associated adjacency graph $G(\mathbf{A}) = [X(\mathbf{A}), E(\mathbf{A})]$, where $X(\mathbf{A}) = \{1, 2, ..., N\}$ is the set of nodes corresponding to rows and columns of the matrix, and $E(\mathbf{A}) = \{\{X_i, X_j\}, ...\}$ is the set of edges corresponding to nonzero entries.[33] In Figure 1 the corresponding graphs for the above-mentioned matrices **A** and **A′** are depicted.

In terms of graph theory, a symmetric permutation does not affect the structure of the graph but only changes the numbering of the nodes in the graph, and all adjacency graphs for any symmetric permutation of a sparse symmetric matrix are isomorphic. However, the nonzero pattern of the Cholesky factors **L** and **L′** are very different, and in fact it is the numbering of the nodes in the adjacency graph that determines the amount of fill-in occurring during the factorization.

Recall from eq 1 that the entries of a particular column in the Cholesky factor depend on the columns to its left already being factorized. That means we have to start the factorization beginning with column 1. The relationship between the graph theoretical representation and the factorization is now that the calculation of the elements of a given column corresponds to removing the corresponding node from the adjacency graph. However, after having factorized a column, a rank update to all subsequent columns has to be performed, which as we have already seen might result in additional fill-in. Since every off-diagonal nonzero is represented by an edge in the adjacency graph, upon removal of a node, some additional edges may have to be added to the graph. In graph theoretical terms, all nodes that have been connected to the removed node have to form a *clique* afterward, i.e., all these nodes have to be connected to each other, which thus might require additional edges to be introduced.

As an example consider the graph in Figure 1a. The first column of **L** is completely dense since node 1 is connected to every other node in that graph. Upon removing node 1 from this graph, one has to add all possible missing edges between all remaining nodes to form the clique, i.e. after eliminating the first node the resulting graph is fully connected and thus the resulting Cholesky factor is completely dense.

If on the other hand one applies the same elimination procedure to the graph of matrix **A′** (Figure 1b), then one can easily verify that no additional edges have to be introduced at all, thus the resulting Cholesky factor does not suffer from any fill-in and has the same nonzero structure as **A′**.

3.3.2. Finding Fill-Reducing Reorderings. Since all adjacency graphs for any symmetric permutation applied to a sparse symmetric matrix are isomorphic and the amount of fill-in only depends on the numbering of the nodes, finding a permutation that results in a small amount of fill-in is equivalent to determining an appropriate elimination sequence.

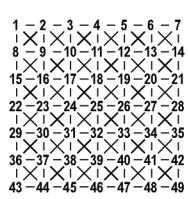
For the special case that the adjacency graph of **A** is a tree (i.e., there are no cycles in the graph), there always exists a reordering that does not introduce any additional fill-in. More generally, any chordal graph (i.e., a graph where every cycle of length at least four has an edge that connects two nonconsecutive nodes on the cycle) has a perfect elimination ordering.[33] The adjacency graph, e.g., of matrix **A** is chordal. While linear time implementations exist[34] that can test whether or not a given graph is chordal, the adjacency graphs of most matrices, however, do not fall into this category. Nevertheless a good fill-reducing reordering may still exist.

Unfortunately the task of finding an optimal, i.e., least fill-in, reordering is known to be NP-hard[35] and is thus not feasible. However, several methods have been established that provide low fill-in reorderings in polynomial time by using heuristics.

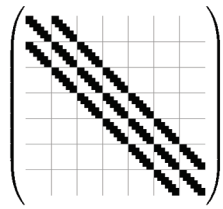
Among the most popular ones of these are the Reverse Cuthill−McKee (RCM),[36] the Lexicographic Breadth First Search (LexBFS),[34,37] the Minimum-Degree (MD),[28,38] (or approximate derivations (AMD) thereof),[39] and Nested Dissection (ND)[30,40−43] algorithms. The first two mentioned strategies have been implemented as $\mathcal{O}(|\mathbf{A}|)$ algorithms, i.e., their runtime is bound from above to be proportional to the number of nonzeros $|\mathbf{A}|$ in the matrix, and we would like to stress that RCM has already been applied in a chemically motivated context,[44] where it has been applied to the connectivity matrix of large molecules in order to reduce its bandwidth.

Strictly speaking, MD is an $\mathcal{O}(N^3)$ algorithm,[45] however it needs this time only for dense matrices and much less if the matrix is sparse. For ND, except for special cases,[30] no strict runtime bound has been established yet. Nevertheless, we used the METIS library[43] for our implementation, and we found that it can produce high-quality reorderings in reasonable amounts of time. Although its runtime is far higher than any other mentioned reordering strategy, this increased demand is itself far outweighed by the gains achieved during the subsequent numerical factorization. We are not going to describe the ND algorithm in great detail and rather refer the interested reader to consult the orginal research papers.[30,40−43]

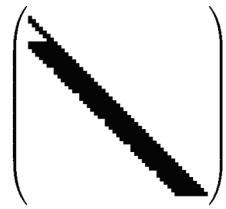
Just briefly, ND is a divide and conquer algorithm that tries to find separator nodes in a graph, i.e., nodes which upon removal would let the graph fall apart into two or more disconnected subgraphs of similar size. Those nodes are then assigned the highest node labels, i.e., they will be eliminated last. The algorithm is then applied recursively to the subgraphs until all nodes have been labeled. In Figure 2 the nonzero structures of matrices for the prominent example[46] of a regular 7×7 grid are depicted. We have used a color code in order to



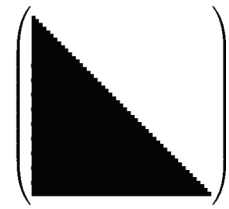
(a) Adjacency graph of a regular 7×7 grid.



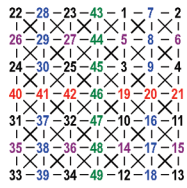
(b) Non-zero pattern of the matrix **A** corresponding to the adjacency graph in Figure 2(a)



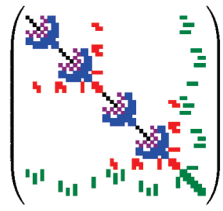
(c) Non-zero pattern of the Cholesky factor **L** of **A**



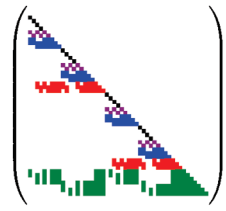
(d) Non-zero pattern of the inverse $\mathbf{L}^{-1}$ of the Cholesky factor



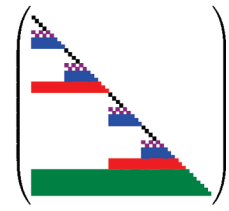
(e) Adjacency graph of a regular 7×7 grid after having applied a Nested Dissection reordering.



(f) Non-zero pattern of the matrix **A** corresponding to the adjacency graph in Figure 2(e)



(g) Non-zero pattern of the Cholesky factor $\mathbf{L}'$ of $\mathbf{A}'$



(h) Non-zero pattern of the inverse $\mathbf{L}'^{-1}$ of the Cholesky factor

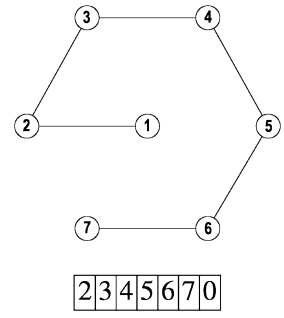
Figure 2. Illustration of a ND reordering applied to a 7×7 regular grid. (a−d) refer to matrices corresponding to the original numbering scheme, whereas (e−h) depict the matrices after applying a ND reordering. We have used a color code to highlight the individual sets of separator nodes at each level of dissection. Note the recurring patterns in (f−h).

highlight the effect of the individual steps during the ND. Matching colors indicate separator nodes of the same level, e.g., the first set of node separators is colored green. Upon their removal, the graph is separated into two disjoint subgraphs, which by removing the red nodes are further subdivided and so on and so forth. The resulting matrix $\mathbf{A}'$ (Figure 2f) shows a very characteristic pattern of recurring nonzero patterns, and these patterns will turn out to be most important for the computation of the inverse of the Cholesky factor.

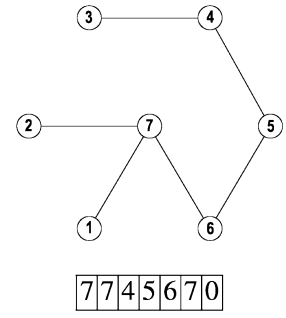
In terms of their sparsity both matrices **L** and $\mathbf{L}'$ are quite similar. This is due to the fact that the initial ordering is already quite good and is similar to a RCM ordering. In fact, all mentioned heuristics are able to find good fill-in reducing orderings, and in terms of the number of nonzeros in the Cholesky factor, they are more or less equivalent. ND, however, has one distinctive feature which makes it by far the most useful strategy if one is also interested in computing the inverse of the Cholesky factor. However, before we can explain the reasons for this, it is necessary to introduce the concept of elimination trees.[33]

3.3.3. Elimination Trees. The elimination tree[33] is defined as the adjacency graph of the Cholesky factor **L** from which all nonzeros below the diagonal except for the first one of each column have been removed, as indicated by °. For instance, referring back to the examples of eqs 5 and 6 we have

$$\mathbf{L} = \begin{pmatrix} \bullet & & & & & & \\ \bullet & \bullet & & & & & \\ \circ & \bullet & \bullet & & & & \\ \circ & \circ & \bullet & \bullet & & & \\ \circ & \circ & \circ & \bullet & \bullet & & \\ \circ & \circ & \circ & \circ & \bullet & \bullet & \\ \circ & \circ & \circ & \circ & \circ & \bullet & \bullet \end{pmatrix} \quad \mathbf{L}' = \begin{pmatrix} \bullet & & & & & & \\ & \bullet & & & & & \\ & & \bullet & & & & \\ & & \circ & \bullet & \bullet & & \\ & & & \circ & \bullet & \bullet & \\ & & & & & \bullet & \bullet \\ \bullet & \bullet & \circ & \circ & \circ & \bullet & \bullet \end{pmatrix} \quad (13)$$



(a) Elimination tree for matrix **A**



(b) Elimination tree for matrix $\mathbf{A}'$

Figure 3. Elimination trees and their corresponding vectorial representations for the symmetric matrices before and after applying a permutation.

Elimination trees always have a root, which is the highest numbered node. A node which is directly connected to the root is a child node of the root, while the root is its parent. Every node has only one parent but can have several children. Nodes which do not have any children are called leaf nodes. Every elimination tree has at least one leaf node but can have several.

In Figure 3 the elimination trees for the matrices **A** and $\mathbf{A}'$ are depicted, and in both cases, the node with number 7 is the root. The elimination tree of **A** (Figure 3a) has only one leaf node (1), while that of $\mathbf{A}'$ (Figure 3b) has three leaf nodes (1, 2, 3). Elimination trees can easily be stored in the form of a parent vector, i.e., for each node its corresponding parent node is stored in an array of size *N*, as also illustrated in Figure 3.

Given the adjacency structure of a sparse matrix and an elimination sequence (i.e., a reordering of the rows and columns), the elimination tree can be determined from the actual graph as outlined in greater detail by Liu.[47] We will not describe the actual algorithm here but only mention its runtime bounds. The original algorithm presented by Liu using *path compression* has a runtime complexity of $\mathcal{O}(|\mathbf{A}| \log_2 N)$,[48] where $|\mathbf{A}|$ stands for the number of nonzeros in the matrix subject to factorization. He mentions that another version of this algorithm exists that uses *path compression and balancing*[49−51] having a lower runtime bound of $\mathcal{O}(|\mathbf{A}| \alpha(N, |\mathbf{A}|))$ with $\alpha(N, |\mathbf{A}|)$ being the functional inverse[33,49−51] of the Ackermann function.[52] This implementation relies on a more sophisticated implementation of the set/union problem,[48,49,53] however it has been found[51] that the first version is much more efficient to implement, so the latter algorithm would have a runtime advantage only for very large graphs. Strictly speaking, none of these algorithms is truly linear, but the second one is usually considered as being *almost linear*, since α can essentially be regarded as a constant for all practically relevant integers N and $|\mathbf{A}|$.

Having determined the elimination tree for a particular reordering of a sparse symmetric matrix, the next step consists of finding what is called a postordering. This is a particular form of an equivalent reordering of the nodes in the elimination tree that changes neither the number of arithmetic operations for computing the Cholesky factorization nor the structure of the resulting adjacency graph. Therefore, in terms of both storage and computational costs, any postordering is as good as the original ordering. However, we may use them to take advantage of other aspects of elimination.

In a postordering, the nodes within every subtree of the elimination tree are numbered consecutively. The root of a subtree will always be labeled last among nodes in the subtree, and it turns out that given an elimination tree, a postorder numbering can be computed in linear time by a depth-first search.[54] As a byproduct from this step we also obtain a vector *depth* of length N that contains the distance of each node in the elimination tree from the root. This vector will turn out to be useful later. For the elimination trees depicted in Figure 3 however the nodes are already numbered in accordance with a postordering, with node 7 being the root in both cases.

The elimination tree contains useful information about data dependencies. The straightforward Cholesky factorization algorithm starts with eliminating the lowest-numbered node. If the nodes in the elimination tree are labeled according to a postordering, then the lowest-numbered node is always a leaf. Note however, that for elimination trees which have more than one leaf, we could have assigned any of them the lowest number. The factorization may thus start with removing any leaf node from the elimination tree, which is equivalent to computing the corresponding column of the Cholesky factor. Since leaf nodes do not have any children, all leaf nodes can even be removed at once, and algorithms exploiting this in a parallel manner have been developed[42,55,56] as well. Repeating this process of factorizing those columns that correspond to leaf nodes can continue as long as the root is present and will stop once this node has been eliminated as well.

Although there are no strict runtime bounds known for ND, its appealing property is that it produces broad elimination trees, i.e., trees which have many leaves. Thus these reorderings are especially useful for parallel factorization. But even if one does not intend to use any form of parallelism for the factorization step, the number of leaves in the elimination tree, or more precisely the distance from the root to the leaves, determines the nonzero structure of the inverse of the Cholesky factor as we will show later.

We also note that techniques exist that aim at finding elimination trees of minimum height,[56−58] and it is known that for every graph there exists a nested dissection ordering with minimal separators which produces an elimination tree of minimum height,[59] but unfortunately this problem is NP-hard[60,61] as well. However, we have made no attempt to find an elimination tree of minimum height, since the METIS reordering already produces well-balanced elimination trees of low height.

3.3.4. Nonzero Pattern of the Cholesky Factor. After having determined an elimination sequence that leads to a reasonable amount of fill-in, the concluding step of the symbolic Cholesky factorization is the determination of the actual nonzero pattern. In order to be memory efficient, we will use the compressed sparse column (CSC) storage scheme,[62] i.e., for every column we will store the row indices and the numerical data for the nonzero elements only, and while the computation of the numerical values will be postponed to the numerical factorization step, the determination of the row indices needs to be performed during the symbolic step. Note, that due to $\mathbf{A}'$ being symmetric, we actually only need to store the lower triangular part of this matrix.

However, before we can proceed to calculate the actual row indices we need to allocate the appropriate amount of memory required to store them, and thus we need to know how many nonzeros the Cholesky factor will have. It turns out that efficient algorithms for this task have already been developed,[51] and as before, we will only report on their runtime bounds instead of explaining the algorithm in detail. Actually the algorithm calculates the number of nonzeros for every row/column and is related to the algorithm employed for the determination of the elimination tree. It is thus not surprising that the computational complexity again depends on how the set/union problem[48,49,53] is implemented, and while the best known implementation has a complexity of $\mathcal{O}(|\mathbf{A}| \alpha(N, |\mathbf{A}|))$, still, the computationally more efficient implementation[51] has a slightly higher complexity of $\mathcal{O}(|\mathbf{A}| \log_2 N)$.

Knowing the exact number of nonzeros now allows us to allocate the actual storage space for the CSC structure of the Cholesky factor, and we can proceed with determining the actual row and column indices now. Since all $|\mathbf{L}|$ nonzero elements of the Cholesky factor need to be determined, the best runtime complexity that can be expected for this task is $\mathcal{O}(|\mathbf{L}|)$. Indeed an efficient and quite simple algorithm exists[63] that operates on the nonzero structure of the permuted matrix $\mathbf{A}$ and its corresponding elimination tree which meets this runtime bound.

3.3.5. Supernodes. Now that we have determined a proper fill-reducing reordering and the actual nonzero structure of the resulting Cholesky factor, we are almost ready to perform the actual numerical factorization. But before we do so, it is important to introduce the concept of supernodes.

Supernodes[21,31,32] are collections of columns in the Cholesky factor that share a similar nonzero pattern. Let x_p be a parent node of x_i in the elimination tree. Both nodes belong to the same supernode if the nonzero structure of column x_p is the same as that for x_i, except for the diagonal element of column x_i, and x_i is the only child of x_p. Since we assume the nodes in the elimination tree to be labeled according to a postordering, a supernode is thus a collection of columns in **L** that form a clique in the graph and share the same set of adjacent nodes outside the clique. Thus each supernode corresponds to a completely dense triangular diagonal block of the Cholesky factor and has the same nonzero pattern below the diagonal block. For the simple example given earlier, the supernodal partioning scheme is as follows:

$$\mathbf{L}' = \begin{pmatrix} \bullet & & & & \\ & \bullet & & & \\ & & \bullet & & \\ & & \bullet & \bullet & \\ & & \bullet & \bullet & \\ & & & \bullet & \\ & & & \bullet & \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \tag{14}$$

It will turn out that knowing the supernodal partitioning scheme allows for a very efficient numerical factorization at the cost of only slightly increased memory requirements. This cost however is more than outweighed by the gains in speed one obtains, since this partitioning scheme allows to employ highly optimized dense BLAS and LAPACK kernels for the numerical factorization step even for sparse matrices.

3.4. Numeric Factorization. After having completed the symbolic Cholesky factorization, we will now outline how the numeric factorization can be carried out efficiently. As mentioned above, we want to take advantage of the supernodal partitioning scheme of the resulting sparse Cholesky factor by employing highly optimized dense BLAS and LAPACK kernels. For didactic reasons however, it is useful to briefly explain how a dense Cholesky factorization is implemented in LAPACK.

3.4.1. Level 3 BLAS Factorization. Most modern computers have cache architectures,[64] i.e., besides the main memory (RAM) they have up to three layers of memory that are located near the CPU which provide faster data access than to RAM. Accessing data in the level 1 cache is the fastest and has almost no latency, i.e., the level 1 cache can deliver data at every clock cycle of the CPU. However, cache is much more expensive than RAM, and thus the size of the level 1 cache is limited to usually a few kilobytes. By contrast, the RAM is the slowest memory, and as a rule of thumb, fetching data from RAM takes up to 20 clock cycles.[65] This is the reason why a naïve implementation of a matrix−matrix multiply will run at ∼5% of the theoretical peak performance, since ∼95% of the time the CPU is idle while waiting for data. More sophisticated implementations of basic linear algebra kernels like those in the BLAS library take this cache hierarchy into account and operate on smaller blocks of matrices that completely fit into the cache.[66] This effectively reduces the amount of memory fetches from RAM and leads to much more efficient codes which can run as fast as ∼95% of the theoretical peak performance of the CPU, i.e., they are about 20 times as fast.

The LAPACK library contains a lot of useful and highly optimized kernels that make use of this blocking technique as well and the routine DPOTRF, e.g., is the level 3 BLAS implementation of the Cholesky factorization for dense matrices. We will briefly review this algorithm now.

Let the lower triangular part of a symmetric positive definite matrix **A** be divided into six blocks A_1, ..., A_6 with the diagonal blocks A_1, A_4, A_6 being quadratic

A_1		
A_2	A_4	
A_3	A_5	A_6

then we start the factorization on the first diagonal block by using the LAPACK function DPOTRF:

L_1		
A_2	A_4	
A_3	A_5	A_6

As the reader can verify, the entries of the blocks below the diagonal one can be computed as the solution of a linear set of equations $\mathbf{y} = \mathbf{Tx}$, where **T** is a triangular matrix, namely the Cholesky factor of the diagonal block. We can thus simply use the level 3 BLAS function DTRSM for this task:

L_1		
$A_2 \cdot L_1^{\mathrm{T}-1}$	A_4	
$A_3 \cdot L_1^{\mathrm{T}-1}$	A_5	A_6

This concludes the factorization of the first column block, but before we can proceed, we have to update all remaining column blocks to the right of the current column block. This is essentially a symmetric rank update and we can use the BLAS kernel DSYRK for this task:

L_1		
L_2	$A_4 - L_2 \cdot L_2^{\mathrm{T}}$	
L_3	$A_5 - L_3 \cdot L_2^{\mathrm{T}}$	$A_6 - L_3 \cdot L_3^{\mathrm{T}}$

Now we can continue with factorizing the second diagonal block, by using DPOTRF again

L_1		
L_2	L_4	
L_3	A_5'	A_6'

and by using DTRSM we can factorize the remaining blocks of the current column:

L_1		
L_2	L_4	
L_3	$A'_5 \cdot L_4^{T-1}$	A'_6

Again, we apply the rank update to the remaining column with DSYRK

L_1		
L_2	L_4	
L_3	L_5	$A'_6 - L_5 \cdot L_5^T$

and likewise we would continue with this process until the last diagonal block has been factorized

L_1		
L_2	L_4	
L_3	L_5	L_6

which concludes the factorization.

After this brief explanation on how to compute the Cholesky factorization of a dense matrix, we will now turn to the discussion of a corresponding sparse implementation. As stated above, the collection of nodes that belong to the same supernode forms a clique in the adjacency graph of $\mathbf{L}$, and therefore, the diagonal block of each supernode is always completely dense, and we can employ the LAPACK kernel DPOTRF for its factorization. In order to be able to do so however, we have to set up an appropriate data structure that allows for the invocation of this function, i.e., we have to store the diagonal blocks as a dense matrix, and although the function DPOTRF only operates on the lower (or upper) triangular part of the matrix to be factorized, the data has still to be passed as a full $n \times n$ array, where n is the dimension of the matrix. That means, we have to allocate the upper triangular part of each diagonal block of every supernode as well as indicated by °. This additional memory

$$\mathbf{L}' = \begin{pmatrix} \bullet & & & & \\ & \bullet & & & \\ & & \bullet\ \circ & & \\ & & \bullet\ \bullet & & \\ & & \bullet\ \bullet & \bullet\ \circ\ \circ & \\ & & & \bullet\ \bullet\ \circ & \\ \bullet & \bullet & \bullet\ \bullet & \bullet\ \bullet\ \bullet & \end{pmatrix} \tag{15}$$

requirement thus causes a small overhead which however is more than outweighed by the increased performance compared to a sparse indexing based factorization.

So far of course this only allows for the efficient computation of the diagonal blocks of $\mathbf{L}'$. However, we know that all elements below the diagonal block share the same row indices and can thus, by omitting the zero rows, be stored compactly as a dense rectangular matrix as well. Note that we can actually store the whole supernode as a dense matrix if we use the LDA parameter of the BLAS and LAPACK routines properly. The proper data structure for the numerical factorization is thus essentially a collection of dense matrices (one for every supernode) that contain the numerical values of the lower triangular part of the matrix $\mathbf{A}'$, as illustrated in Figure 4.

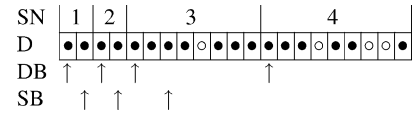


Figure 4. Illustration of the supernodal storage scheme used for the Cholesky factor $\mathbf{L}'$. SN indicates the number of the supernode, D holds the actual numerical data stored in column major order, where ° denotes the additional storage required in order to be able to invoke BLAS and LAPACK routines. DB and SB indicate the pointers to the beginning of the diagonal and subdiagonal blocks, respectively.

This storage scheme now allows us to proceed with the remaining steps of the factorization in a similar way as illustrated for the dense factorization. After having factorized the diagonal block of a supernode, we can employ the dense matrix kernel DTRSM to compute the remaining parts below the diagonal block, and since the subdiagonal part is stored as a dense matrix, we can yet use the level 3 BLAS function DSYRK for the computation of the rank update of the columns to the right of the current supernode. However, we have to store the result in a temporary array first and then scatter the data into their final destinations, since the nonzero pattern of subsequent supernodes is not necessarily the same as that of the current supernode. That means, we need one more temporary array to store the rank update. The size of this array can be determined prior to the computation since it only depends on the size of the largest subdiagonal block and is quite small for sparse matrices.

This sequence can then be carried out for every supernode and will eventually lead to the Cholesky factor of the matrix $\mathbf{A}'$. Note that the last supernode always consists of just a dense square block. The size of this block tends to become larger the denser the Cholesky factor is. Thus in case the matrix $\mathbf{A}'$ is already quite dense, there will be only one large supernode which is factorized by using the highly optimized LAPACK kernel DPOTRF. That means, we can expect a seamless transition in performance from very sparse to completely dense matrices. Although there is an additional overhead for the symbolic Cholesky factorization, that can of course be avoided if one knows that the Cholesky factor is going to be completely dense; there will be no significant loss in performance as compared to DPOTRF if the matrix is more or less dense. This behavior is probably the most important advantage of using reordering and supernode techniques for the factorization, since the performance will not be affected as much as it is often the case for sparse matrix routines if the matrix is dense.

4. Inverses of Cholesky Factors

The elements $L_{i,j}^{-1}$ of the inverted Cholesky factor $\mathbf{L}^{-1}$ are given by

$$L_{i,j}^{-1} = \begin{cases} \dfrac{1}{L_{i,i}}\left(-\sum_{k=j}^{i-1} L_{i,k} L_{k,j}^{-1}\right), & i > j \\ \dfrac{1}{L_{i,i}}, & i = j \end{cases} \tag{16}$$

As stated above, for the solution of linear sets of equations like eq 2 there is usually no need to explicitly compute $\mathbf{A}^{-1}$ or

$\mathbf{L}^{-1}$ since one can find the solution by forward/back substitution. In the presence of rounding errors, this is not only the most efficient way but also the most accurate, since the computation of $\mathbf{L}^{-1}$ is not backward stable[27] and thus can lead to unnecessary rounding errors. Furthermore, even if the right-hand side vectors $\mathbf{y}$ are sparse, the solution $\mathbf{x}$ is usually dense, due to the fact that the inverse of $\mathbf{A}$ is usually[26] dense. In light of this, it can easily be understood why all available sparse solvers[21,22,67] always operate on dense right-hand side vectors.

However, if one intends to find the solution to the forward substitution step only and the right-hand side is very sparse, then chances are good that the result remains sparse. This is due to the fact that the result of the forward substitution is given by $\mathbf{L}^{-1}\mathbf{y}$, and as long as both matrices are sufficiently sparse, the result may still be. Furthermore, if one intends to use the inverse Cholesky factor as a transformation matrix, then it becomes obvious that for reasons of efficiency it should be as sparse as possible.

Again, we assume the Cholesky factor has been computed for the permuted matrix $\mathbf{A}'$ in order to reduce the fill-in during the factorization. Then the inverse of the original (i.e., unpermuted) matrix is given by

$$\mathbf{A}^{-1} = (\mathbf{P}^{\mathrm{T}}\mathbf{P}\mathbf{A}\mathbf{P}^{\mathrm{T}}\mathbf{P})^{-1} \tag{17}$$

$$= \mathbf{P}^{\mathrm{T}}(\mathbf{P}\mathbf{A}\mathbf{P}^{\mathrm{T}})^{-1}\mathbf{P} \tag{18}$$

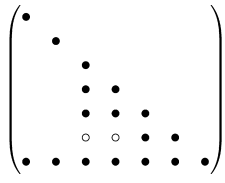
$$= \mathbf{P}^{\mathrm{T}}(\mathbf{L}'\mathbf{L}'^{\mathrm{T}})^{-1}\mathbf{P} \tag{19}$$

$$= \mathbf{P}^{\mathrm{T}}\mathbf{L}'^{-1^{\mathrm{T}}}\mathbf{L}'^{-1}\mathbf{P} \tag{20}$$

and thus a potentially sparse, though nontriangular, equivalent of the inverse of the orginal Cholesky factor is given as $\mathbf{L}'^{-1}\mathbf{P}$.

As for the Cholesky factorization, the computation of the inverse can be split up into a symbolic and a numeric part.

4.1. Symbolic Cholesky Inversion. Given the nonzero structure of the Cholesky factor, the nonzero structure of its inverse is quite easy to predict. It is defined as the *transitive closure* of the elimination tree,[26,40,61] i.e., the nonzero entries of every column are given by following the path from the corresponding node up to the root. Every node visited on the path corresponds to a nonzero element in the column. Thus for the above given example, the structure of the inverse Cholesky factor can be determined from the elimination tree depicted in Figure 3b:

$\mathbf{L}'^{-1} =$  (21)

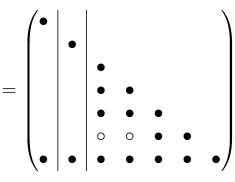
Here we use ° to indicate fill-in positions during the inversion. Fill-in usually occurs and can be avoided only in special cases (e.g., if the adjacency graph of the Cholesky factor is a tree), which means that the inverse of a Cholesky factor is usually denser than the Cholesky factor, which itself is usually denser than the matrix subject to the factorization. This is of course a huge drawback, and thus computational gains by using sparse algorithms can only be expected if the matrix is very sparse and its elimination tree is very broad. The latter point is by far the most important since the shorter the paths from the nodes to the root are the less nonzeros the inverse of the Cholesky factor will have. This observation clarifies the importance of finding fill-reducing orderings that result in an elimination tree with many leafs.

We have tested all of the above-mentioned reordering heuristics, and while all perform very well for the Cholesky factorizaton, only ND is able to produce "bushy" elimination trees, while all other reorderings tend to produce elimination trees that are more or less straight lines like the one in Figure 3a. Although ND is the most expensive reordering strategy, the quality of the reordering in terms of its supernode structure and number of nonzeros in the inverse is far superior, and the higher computational demand is by far outweighed by the savings gained during the numerical steps.

4.1.1. Nonzero Pattern of the Inverted Cholesky Factor. The symbolic part of the inversion is focused on determining the row and column indices of the nonzeros, and in order to avoid multiple memory allocation steps or a waste of memory, again we first need to know the exact number of nonzeros. Since the nonzero pattern is determined by the transitive closure of the elimination tree, the number of nonzeros in one column is given by the distance of the corresponding node from the root of the elimination tree, and as mentioned above, this information has already been computed as a byproduct of the postordering. The number of nonzeros in the whole inverse can thus easily be computed by summing over all elements in the vector *depth*, which has a complexity of $\mathcal{O}(N)$.

After having allocated the sparse data structure, we can then proceed with the determination of the actual nonzero indices. Since we have to determine all of them, the best computational complexity for this task is expected to be $\mathcal{O}(|\mathbf{L}^{-1}|)$. Since we have stored the elimination tree as a vector in which the parent of every node is stored, it is easy to verify that by following the path from a given node to the root of the elimination tree; all nonzero elements can be computed within the given runtime bound.

4.1.2. Supernodes during Inversion. We have stressed that taking into account the supernodal structure can lead to much more efficient numerical factorizations, and it turns out that we can use the same technique for the inversion as well. Due to the additional amount of fill-in, however, the supernodal structure for computing the inverse is usually different from the one used for the factorization, as illustrated for the given example:

$\mathbf{L}'^{-1} =$  (22)

The data structure that allows utilizing BLAS and LAPACK kernels is essentially the same as that used for the factorization, i.e., we store each supernode as a dense rectangular matrix.

4.2. Numeric Cholesky Inversion. The biggest difference between the computation of the Cholesky factor and the

calculation of its inverse is probably the order in which the columns have to be computed. As the reader can verify from eq 16, by contrast to the factorization, the elements of the inverse depend on elements of the inverse to their right, i.e., we have to start the inversion with the utmost right column and then proceed to the left.

As for the Cholesky factorization, the LAPACK library already contains a function DTRTRI that is able to compute the inverse of a dense triangular matrix by invoking level 3 BLAS calls, and since we are going to adopt this algorithm to the case when the triangular matrix is sparse, for didactic reasons we will illustrate the basic steps for a dense matrix.

Let the Cholesky factor **L** be divided into six blocks L_1, ..., L_6 with the diagonal blocks L_1, L_4, L_6 being quadratic

L_1		
L_2	L_4	
L_3	L_5	L_6

then we can start the factorization on the last diagonal block by using the LAPACK kernel DTRTRI:

L_1		
L_2	L_4	
L_3	L_5	L_6^{-1}

As the next step we have to update the subdiagonal block of the second column block, and as the reader can verify, this is equivalent to a matrix multiply between a triangular and a rectangular matrix. We can thus use the BLAS function DTRMM for this task:

L_1		
L_2	L_4	
L_3	$L_6^{-1} \cdot L_5$	L_6^{-1}

After this multiplication the final block L_5^{-1} can be computed as the solution to a linear set of equations with a triangular coefficient matrix by invoking the BLAS kernel DTRSM

L_1		
L_2	L_4	
L_3	$L_5' \cdot L_4^{-1}$	L_6^{-1}

before we invert the diagonal block of the second row by using DTRTRI again:

L_1		
L_2	L_4^{-1}	
L_3	L_5^{-1}	L_6^{-1}

At this point the last two columns already contain the right numerical values, and we can proceed with the first column. In principle this can be carried out by multiplying the already inverted lower right-hand submatrix with the subdiagonal block of the first column. In order to be able to do this in place, however, we have to break this up into smaller steps, and we will start by modifying block L_3 by calling DTRMM:

L_1		
L_2	L_4^{-1}	
$L_6^{-1} \cdot L_3$	L_5^{-1}	L_6^{-1}

The next step requires a matrix multiply between two rectangular matrices, which is best been done by the prominent BLAS kernel DGEMM:

L_1		
L_2	L_4^{-1}	
$L_3' + L_5^{-1} \cdot L_2$	L_5^{-1}	L_6^{-1}

Note that for this step the unmodified block L_2 is required. This is the reason why the update of L_2 by invoking DTRMM can only take place after having updated all blocks below it:

L_1		
$L_4^{-1} \cdot L_2$	L_4^{-1}	
L_3''	L_5^{-1}	L_6^{-1}

Now the multiplication by the inverse of the diagonal block of the current row can be done by using DTRSM again

L_1		
$L_2' \cdot L_1^{-1}$	L_4^{-1}	
$L_3'' \cdot L_1^{-1}$	L_5^{-1}	L_6^{-1}

and the inversion of the diagonal block L_1 with calling DTRTRI concludes the inversion:

L_1^{-1}		
L_2^{-1}	L_4^{-1}	
L_3^{-1}	L_5^{-1}	L_6^{-1}

Of course the actual sparse implementation is a little more complicated, but it is essentially identical with a slight overhead for bookkeeping. The appealing advantage in this implementation is that all operations can be done by directly calling BLAS and LAPACK functions and that there is not a single step where temporary results have to be scattered into their destinations. This and the fact that the individual supernodes are larger compared to those in the factorization step account for this algorithms' efficiency.

Table 1. CPU Timings for Computation of the Cholesky Factor **L** and Its Inverse $\mathbf{L}^{-1}$ of Two-Center Overlap Matrices **S** in the Basis Set 6-31G(d) for a Series of Linear Alkanes, Single Graphite Layers, and Spherical Diamond Blocks of Various Sizes[a]

				thresholding			supernodal			LAPACK
	n	dim(**S**)	% (**S**)	% (**L**)	% ($\mathbf{L}^{-1}$)	time (s)	% (**L′**)	% ($\mathbf{L'}^{-1}$)	time (s)	time (s)
alkanes (C_nH_{2n+2})	100	1904	8.1	11.8	30.6	2.98	9.1	16.2	0.20	0.78
	200	3804	4.1	6.8	18.5	9.41	5.1	10.5	0.48	5.64
	300	5704	2.7	4.7	13.1	16.06	3.5	8.0	0.82	18.37
	400	7604	2.1	3.6	10.1	22.76	2.7	6.5	1.19	42.31
	500	9504	1.7	2.9	8.2	29.26	2.2	5.6	1.59	82.16
	600	11404	1.4	2.4	6.9	35.12	1.8	4.8	1.96	140.58
	700	13304	1.2	2.1	6.0	42.06	1.6	4.3	2.39	220.58
	800	15204	1.0	1.8	5.2	48.18	1.4	3.9	2.82	328.05
	900	17104	0.9	1.6	4.7	56.75	1.2	3.6	3.28	465.61
	1000	19004	0.8	1.5	4.2	64.88	1.1	3.3	3.73	634.87
graphite (C_n)	100	1500	22.1	30.2	30.5	3.36	22.3	26.9	0.44	0.39
	200	3000	12.9	28.5	30.5	29.86	17.5	24.2	1.77	2.81
	300	4500	9.2	26.3	30.5	92.94	14.3	21.7	4.18	9.14
	400	6000	7.1	24.4	30.5	195.77	12.3	20.2	7.09	21.15
	500	7500	5.9	22.7	30.5	351.83	10.8	18.7	10.15	40.61
	600	9000	5.0	21.2	30.4	566.38	9.8	17.8	15.31	69.36
	700	10500	4.3	20.0	30.4	830.96	9.0	16.9	19.43	109.34
	800	12000	3.8	18.9	30.4	1156.21	8.6	16.4	27.44	162.50
	900	13500	3.4	17.9	30.4	1532.60	7.8	15.8	32.41	229.13
	1000	15000	3.1	17.1	30.4	2024.51	7.5	15.1	39.34	313.79
diamond (C_n)	100	1500	62.4	49.9	50.0	8.81	46.8	49.5	1.57	0.39
	200	3000	47.0	49.9	50.0	77.36	43.9	48.2	10.18	2.81
	300	4500	38.0	49.9	50.0	253.42	42.1	47.3	26.35	9.12
	400	6000	31.7	49.9	50.0	605.00	39.9	46.4	48.76	21.08
	500	7500	27.3	49.9	50.0	1165.08	37.9	45.2	94.72	40.65
	600	9000	24.1	49.9	50.0	2022.88	36.0	42.8	139.21	69.25
	700	10500	21.6	49.8	50.0	3231.13	35.6	43.6	226.14	109.09
	800	12000	19.5	49.7	50.0	4643.79	33.5	41.3	255.94	162.29
	900	13500	17.8	49.6	50.0	6570.94	33.1	42.2	365.23	229.99
	1000	15000	16.5	49.5	50.0	9161.83	32.0	40.4	451.80	313.92

[a] All timings are in seconds and have been carried out on a single core/single thread (`OMP_NUM_THREADS=1`) on a 2.31 GHz AMD Opteron 2376 architecture. They include memory allocation times for all sparse matrices and temporary data structures generated. The time required to calculate the overlap matrices, however, is not included. Thresholding indicates the implementation of the Ochsenfeld group, while our implementation is the supernodal. For comparison, the cumulative execution times for the LAPACK calls DPOTRF and DTRTRI as implemented in the Intel MKL are provided. Dim(**S**) stands for the dimension of the overlap matrix and % () indicates the density in percent, respectively.

5. Illustrative Timings

After having explained how the supernodal Cholesky factorization and inversion can be done, it is now time to justify our claims about its superior efficiency.

Because they tend to become very sparse in the large molecule limit and are always strictly positive definite, as long as the basis set does not contain linear dependencies among the basis functions, we choose two-center overlap matrices as our test targets. We consider linear alkanes (C_nH_{2n+2}), graphite (C_n), and diamond (C_n), with $n = 100, ..., 1000$ as representatives of one-, two- and three-dimensional geometries, respectively. For the construction of the geometries standard values were assumed for the geometrical parameters (alkanes: $r(C-C) = 1.54$ Å, $r(C-H) = 1.10$ Å, graphite: $r(C-C) = 1.42$ Å, and diamond: $r(C-C) = 1.54$ Å).

All calculations were carried out using a single core/single thread (`OMP_NUM_THREADS=1`) on a 2.31 GHz AMD Opteron 2376 architecture running linux. All timings include memory allocation times for all sparse matrices and temporary data structures generated by the routines. The computation of the overlap matrices itself, however, is not included. Both codes have been compiled with full optimization using the Intel compilers.

After having computed the overlap matrices, we disregarded all entries below a threshold of 10^{-15} and stored the remaining entries in the CSC format, which we then supplied to both our own implementation and the one of the Ochsenfeld group. (Here we used the same threshold of 10^{-15} throughout.) Since this algorithm uses thresholding criteria for retaining sparsity, we will refer to this as the "thresholding" implementation, while our own code is termed "supernodal".

The collective timings for the sparse Cholesky factorization and the subsequent inversion for the overlap matrices in the 6-31G(d) and 6-311G(2df) basis set are listed in Tables 1 and 2 and depicted in Figure 5, respectively. For comparison, we have also performed timings for the LAPACK routines DPOTRF and DTRTRI from the Intel Math Kernel (MKL) library (version 10.2).

For the linear alkanes both implementations are faster in the large molecule limit as compared to the dense LAPACK codes. This is not surprising since for both basis sets with incresing system size the density of the overlap matrix rapidly

Table 2. CPU Timings for Computation of the Cholesky Factor **L** and Its Inverse $\mathbf{L}^{-1}$ of Two-Center Overlap Matrices **S** in the Basis Set 6-311G(2df) for a Series of Linear Alkanes, Single Graphite Layers, and Spherical Diamond Blocks of Various Sizes[a]

				thresholding			supernodal			LAPACK
	n	dim(**S**)	% (**S**)	% (**L**)	% ($\mathbf{L}^{-1}$)	time (s)	% (**L′**)	% ($\mathbf{L'}^{-1}$)	time (s)	time (s)
alkanes (C_nH_{2n+2})	100	4106	6.8	10.1	31.2	27.45	8.0	15.3	1.05	7.40
	200	8206	3.5	6.2	24.4	118.46	4.4	9.9	2.76	55.14
	300	12306	2.3	4.6	20.4	260.62	3.1	7.6	4.73	178.46
	400	16406	1.7	3.6	17.6	443.93	2.4	6.2	7.05	419.51
	500	20506	1.4	3.0	15.4	644.27	1.9	5.3	9.38	811.10
	600	24606	1.2	2.6	13.7	848.45	1.6	4.7	12.12	1381.50
	700	28706	1.0	2.2	12.3	1080.14	1.4	4.1	14.63	2168.69
	800	32806	0.9	2.0	11.0	1303.58	1.2	3.8	17.65	3253.99
	900	36906	0.8	1.8	10.0	1599.31	1.1	3.4	20.43	4605.42
	1000	41006	0.7	1.6	9.2	1762.28	1.0	3.2	23.19	6304.39
graphite (C_n)	100	3500	18.0	28.2	28.5	44.19	19.8	24.5	3.12	4.41
	200	7000	10.4	27.0	28.5	331.64	14.6	21.5	13.41	33.08
	300	10500	7.4	25.1	28.5	1015.73	12.5	19.4	30.94	109.08
	400	14000	5.7	23.4	28.5	2180.21	10.8	18.0	51.01	255.74
	500	17500	4.7	21.8	28.5	3853.73	9.9	17.2	82.85	496.33
	600	21000	4.0	20.5	28.5	6181.12	8.6	16.1	114.74	852.52
	700	24500	3.4	19.3	28.5	9212.60	7.9	15.3	150.01	1352.71
	800	28000	3.0	18.3	28.4	12697.22	7.3	14.6	195.04	2017.46
	900	31500	2.7	17.4	28.4	16952.24	6.7	14.1	239.24	2864.93
	1000	35000	2.5	16.6	28.4	22187.31	6.4	13.5	296.49	3923.51
diamond (C_n)	100	3500	55.2	49.8	50.0	122.12	45.8	49.0	17.41	4.42
	200	7000	40.3	49.9	50.0	948.06	43.2	47.8	87.65	33.07
	300	10500	32.1	49.9	50.0	3143.71	39.4	44.7	268.13	108.98
	400	14000	26.6	49.9	50.0	7476.32	37.1	43.7	468.03	256.24
	500	17500	22.7	49.9	50.0	14647.72	35.5	42.7	786.33	495.17
	600	21000	20.0	49.9	50.0	24621.83	33.8	41.7	1082.80	856.80
	700	24500	17.9	49.9	50.0	38711.14	32.6	40.9	1704.99	1358.30
	800	28000	16.1	49.8	50.0	59171.17	31.2	40.0	2111.56	2025.69
	900	31500	14.7	49.8	50.0	85512.58	31.1	41.5	3548.40	2860.03
	1000	35000	13.5	49.7	50.0	115717.98	28.9	38.7	3882.37	3924.98

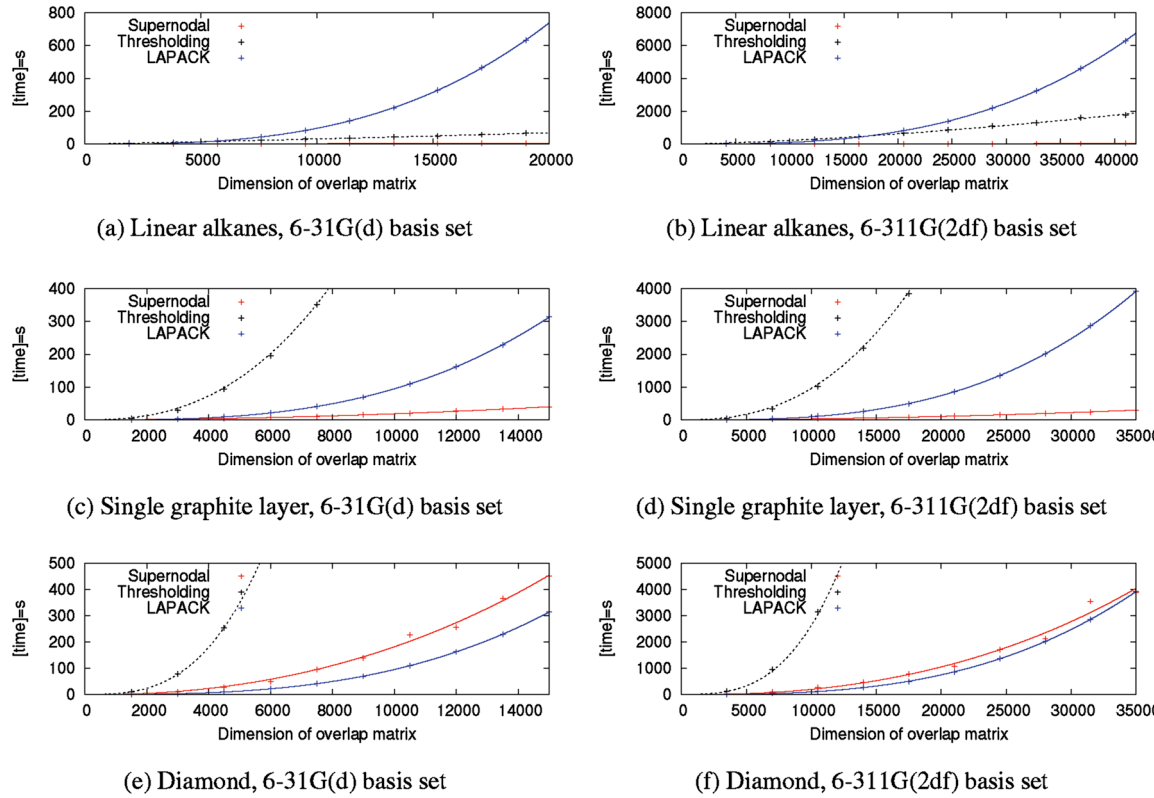
[a] All timings are in seconds and have been carried out on a single core/single thread (`OMP_NUM_THREADS=1`) on a 2.31 GHz AMD Opteron 2376 architecture. They include memory allocation times for all sparse matrices and temporary data structures generated. The time required to calculate the overlap matrices, however, is not included. Thresholding indicates the implementation of the Ochsenfeld group, while our implementation is the supernodal. For comparison, the cumulative execution times for the LAPACK calls DPOTRF and DTRTRI as implemented in the Intel MKL are provided. Dim(**S**) stands for the dimension of the overlap matrix and % () indicates the density in percent, respectively.

decays from 8.1% and 6.8% to 0.8% and 0.7%, respectively. However, while the crossover between the dense kernel and the thresholding algorithm occurs at roughly 5000 basis functions in the 6-31G(d) basis and at 17500 basis functions in the 6-311G(2df) basis, our supernodal implementation is faster than both codes throughout and runs by a factor of 14−78 faster than the thresholding implementation. This indicates that even for very sparse matrices the supernodal technique is advantageous even though the thresholding code shows a lower scaling with system size (with a larger prefactor though) according to the polynomial fit (see Table 3) for the 6-31G(d) basis. We would like to stress that strictly speaking the thresholding code as it is implemented is quadratically scaling since it requires the allocation of a full matrix in order to hold the Cholesky factor. This step of course has a tiny prefactor and could even be avoided by subsequent reallocation of memory once it is actually needed.

Since the supernodal algorithm relies on applying a fill-reducing reordering to the initial matrix, which also reduces the number of arithmetic operations, both matrices $\mathbf{L}'$ and $\mathbf{L}'^{-1}$ are less dense than their unpermuted counterparts. It is thus likely that the supernodal code will have runtime advantage for any reasonable number of basis functions and that the scaling might even further decrease for larger systems.

Of course these test systems are far from being linear in reality and should be seen as ideal test cases for the algorithms. We thus have included more realistic and less ideal test cases as well, one of them being a single graphite layer as a representative of a two-dimensional system.

For graphite the difference in performance between the two sparse implementations is even more pronounced. While the thresholding algorithm is roughly a factor of 5−10 slower than the LAPACK functions, with one exception our supernodal implementation is faster than the optimized dense kernels by up to a factor of 13. Here the superior quality of the ND reordering becomes apparent, which is able to produce Cholesky factors and their corresponding inverses, which are roughly half as dense as those obtained by the thresholding algorithm. Once again, we would like to stress that we have not applied any sort of thresholding, i.e., it is likely that the number of significant elements in the inverse could even be reduced by eliminating those values that fall below a given threshold.



(a) Linear alkanes, 6-31G(d) basis set

(b) Linear alkanes, 6-311G(2df) basis set

(c) Single graphite layer, 6-31G(d) basis set

(d) Single graphite layer, 6-311G(2df) basis set

(e) Diamond, 6-31G(d) basis set

(f) Diamond, 6-311G(2df) basis set

Figure 5. Graphical representation of the data in Tables 1 and 2.

Table 3. Coefficients *a*, *b* for the Fitting of the Data in Tables 1 and 2 According to a Fit Function of the Form $a \cdot \dim(\mathbf{S})$

		6-31G(d)		6-311G(2df)	
		a	*b*	*a*	*b*
alkanes	thresholding	6.17×10^{-4}	1.17	1.76×10^{-4}	1.52
	supernodal	1.61×10^{-5}	1.25	2.13×10^{-5}	1.31
	LAPACK	1.42×10^{-10}	2.96	1.23×10^{-10}	2.97
graphite	thresholding	6.52×10^{-8}	2.51	9.19×10^{-8}	2.50
	supernodal	5.02×10^{-7}	1.89	8.98×10^{-7}	1.87
	LAPACK	1.48×10^{-10}	2.95	1.08×10^{-10}	2.98
diamond	thresholding	3.63×10^{-9}	2.97	2.14×10^{-9}	3.02
	supernodal	1.58×10^{-7}	2.26	3.79×10^{-8}	2.43
	LAPACK	1.41×10^{-10}	2.96	1.17×10^{-10}	2.98

As a concluding challenging example we choose to test the performance of our sparse implementation on rather dense matrices, namely the overlap matrices of diamond. Of course no one would use a sparse code for a dense matrix since highly optimized BLAS and LAPACK kernels have been designed exactly for this purpose, however if the density of the matrix subject to factorization is not known in advance, then one might give a sparse implementation a try if its performance is not too bad as compared to a dense code. As can be seen from Tables 1 and 2, the thresholding algorithm is slower by a factor of 22−30 as compared to LAPACK, and the scaling is even worse. Not surprisingly, our supernodal implementation is slower than the dense equivalent as well, however it is at most 4 times slower, and by contrast to the thresholding implementation, the differences more and more vanish with increasing matrix size and even break even for the diamond cluster with 1000 atoms in the 6-311G(2df) basis. This behavior once again highlights the importance of the supernode technique, which allows for a seamless transition between really sparse systems and more or less dense ones that are treated as if they were sparse. For a completely dense matrix, our implementation would have an overhead for the symbolic Cholesky factorization. The numeric part however would be the same single LAPACK call as for the dense matrix, since the matrix would have just one supernode, and this property renders our implementation more or less generally applicable.

6. Semidefinite and Idempotent Matrices

So far we have only considered strictly positive definite matrices, and as stated above, symmetric positive semidefinite matrices do have Cholesky factors as well. However, in the presence of rounding error, one is forced to use the *full pivoting* technique[25] in order to obtain reliable results, i.e., the factorization is carried out by exchanging the current row/column with that one having the largest diagonal element among those that have not yet been factorized. This process is repeated until all remaining diagonal elements fall below a predefined threshold. The huge drawback of this approach however is that the matrix has to be reordered at every step of the factorization, and since this reordering now depends on the actual numerical values of the matrix, we cannot apply a symbolic Cholesky factorization anymore.

For semidefinite matrices which are very rank deficient (density and two-electron integral matrices, e.g.), this approach is fine. However, although overlap matrices are positive definite as long as there are no linear dependencies among the basis functions, they tend to become numerically semidefinite for larger basis sets. If this happens to be the case, then one would encounter numerical problems during the regular Cholesky factorization. Furthermore the Cholesky factor would not have a regular inverse anymore, and one

- **Prerequisites**
  - compute overlap matrix
  - disregard all entries below a defined threshold
  - store remaining significant elements in CSC format
- **Symbolic Cholesky factorization**
  - create graph structure
  - determine Nested Dissection reordering
  - determine elimination tree
  - determine postordering
  - determine number of non-zeros per row/column and supernodal structure
  - (if desired) determine number of non-zeros in inverse Cholesky factor
  - determine storage space for numerical factorization
- **Numeric Cholesky factorization**
  - create CSC structure of Cholesky factor
  - setup numerical data for supernodal factorization
  - perform numeric factorization
- **Symbolic Cholesky inversion**
  - determine actual non-zero structure of inverted Cholesky factor
  - determine supernodal structure of inverted Cholesky factor
  - determine storage space required for inversion
- **Numeric Cholesky inversion**
  - setup numerical data for inversion
  - perform numeric inversion

Figure 6. Brief summary of the algorithm for the computation of the Cholesky factorization and the inverted Cholesky factor.

would be forced to compute its Moore−Penrose inverse[68] in order to obtain numerically stable and meaningful results.

Fortunately, the Cholesky factorization is generally the most efficient method for testing positive definiteness of matrices,[69] and for overlap matrices, it might thus be best to try a regular Cholesky factorization as outlined further above. In case the Cholesky factorization fails, probably the best alternative in order to get around the demanding SVD[70] required for the computation of the Moore−Penrose inverse is to eliminate the linear dependencies among the basis functions. Although we have not yet implemented it, we want to note that this can be done by computing a LU[71] or QR[72] decomposition first and then using an iterative method for the determination of a basis of the null space.[73] Once this basis is known, it is possible to identify those columns that are linearly dependent and thus would cancel during the Cholesky factorization. These columns can then be permuted to the end of the matrix,[74] and the symbolic Cholesky factorization can then still be applied to the leading columns in order to reduce the fill-in.

Furthermore, we would like to note that for idempotent matrices the Cholesky factorization can also be computed via a QR factorization[75] and although we have not tested the reliability it has been found to yield acceptable results.[76]

7. Conclusion and Outlook

We have presented an efficient algorithm for the computation of the Cholesky factor and its corresponding inverse of a sparse symmetric positive definite matrix, the individual steps of which are briefly summarized in Figure 6. The high efficiency stems from splitting the factorization into a symbolic and a numeric part. The symbolic part is a strictly algebraic algorithm based on graph theory that first tries to find an appropriate reordering of the rows/columns in order to reduce the amount of fill-in and in a second step determines the exact nonzero structure of the resulting Cholesky factor. Furthermore this step finds blocks of columns which share a similar nonzero pattern, the so-called supernodes. These two steps are the most important features of this implementation and account for the superior efficiency of this algorithm as compared to a straightforward sparse implementation.

Furthermore, we have shown that this approach turns out to be useful for the computation of inverted Cholesky

factors as well, and we have stressed that it is crucial to use a ND reordering algorithm in order to preserve at least some degree of sparsity in the inverse. Although we have not made any attempt of implementation so far, we would like to stress that the ND reordering also allows for a parallel implementation.

Although the ND algorithm used for our algorithm is a rather small part of our code, we are convinced that this is the most important ingredient, and this divide and conquer strategy might also turn out to be useful for the design of other low-order scaling methods as well. For example, one might think of dividing a large molecule into many local domains based on decomposing two-center overlap matrices in the context of local correlation methods.

Acknowledgment. We thank Daniel S. Lambrecht and Eric J. Sundstrom for valuable discussions. This work was supported by the Director, Office of Energy Research, Office of Basic Energy Sciences, Chemical Sciences Division of the U.S. Department of Energy under contract no. DE-AC0376SF00098. M.H.-G. is a part-owner of Q-CHEM Inc.

References

(1) Beebe, N. H. F.; Linderberg, J. Simplifications in the generation and transformation of two-electron integrals in molecular calculations. *Int. J. Quantum Chem.* **1977**, *12*, 683–705.

(2) (a) Røeggen, I.; Wisløff-Nilssen, E. On the Beebe-Linderberg two-electron integral approximation. *Chem. Phys. Lett.* **1986**, *132*, 154–160. (b) O'Neal, D.; Simons, J. Application of Cholesky-like matrix decomposition methods to the evaluation of atomic orbital integrals and integral derivatives. *Int. J. Quantum Chem.* **1989**, *36*, 673–688. (c) Koch, H.; Sánchez de Merás, A.; Pedersen, T. B. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.* **2003**, *118*, 9481–9484. (d) Aquilante, F.; Pedersen, T. B.; Lindh, R. Low-cost evaluation of the exchange Fock matrix from Cholesky and density fitting representations of the electron repulsion integrals. *J. Chem. Phys.* **2007**, *126*, 194106. (e) Aquilante, F.; Malmqvist, P.-A.; Bondo Pedersen, T.; Ghosh, A.; Roos, B. O. Cholesky Decomposition-Based Multiconfiguration Second-Order Perturbation Theory (CD-CASPT2): Application to the Spin-State Energetics of Co^{III} (diiminato)(NPh). *J. Chem. Theory Comput.* **2008**, *4*, 694–702. (f) Aquilante, F.; Lindh, R.; Pedersen, T. B. Analytic derivatives for the Cholesky representation of the two-electron integrals. *J. Chem. Phys.* **2008**, *129*, 034106. (g) Røeggen, I.; Johansen, T. Cholesky decomposition of the two-electron integral matrix in electronic structure calculations. *J. Chem. Phys.* **2008**, *128*, 194107. (h) Weigend, F.; Kattannek, M.; Ahlrichs, R. Approximated electron repulsion integrals: Cholesky decomposition versus resolution of the identity methods. *J. Chem. Phys.* **2009**, *130*, 164106. (i) Chwee, T. S.; Carter, E. A. Cholesky decomposition within local multireference singles and doubles configuration interaction. *J. Chem. Phys.* **2010**, *132*, 074104.

(3) Millam, J. M.; Scuseria, G. E. Linear scaling conjugate gradient density matrix search as an alternative to diagonalization for first principles electronic structure calculations. *J. Chem. Phys.* **1997**, *106*, 5569–5577.

(4) Schweizer, S.; Kussmann, J.; Doser, B.; Ochsenfeld, C. Linear-Scaling Cholesky decomposition. *J. Comput. Chem.* **2008**, *29*, 1004–1010.

(5) Aquilante, F.; Bondo Pedersen, T.; Sánchez de Merás, A.; Koch, H. Fast noniterative orbital localization for large molecules. *J. Chem. Phys.* **2006**, *125*, 174101.

(6) Zienau, J.; Clin, L.; Doser, B.; Ochsenfeld, C. Cholesky-decomposed densities in Laplace-based second-order Møller-Plesset perturbation theory. *J. Chem. Phys.* **2009**, *130*, 204112.

(7) Farkas, O.; Schlegel, B. H. Geometry optimization methods for modeling large molecules. *J. Mol. Struct. Theochem* **2003**, *666−667*, 31–39.

(8) Aquilante, F.; Pedersen, T. B. Quartic scaling evaluation of canonical scaled opposite spin second-order Møller Plesset correlation energy using Cholesky decompositions. *Chem. Phys. Lett.* **2007**, *449*, 354–357.

(9) Jung, Y.; Lochan, R. C.; Dutoi, A. D.; Head-Gordon, M. Scaled opposite-spin second or-der Møller-Plesset correlation energy: An economical electronic structure method. *J. Chem. Phys.* **2004**, *121*, 9793–9802.

(10) (a) Aquilante, F.; Lindh, R.; Bondo Pedersen, T. Unbiased auxiliary basis sets for accurate two-electron integral approximations. *J. Chem. Phys.* **2007**, *127*, 114107. (b) Boman, L.; Koch, H.; Sánchez de Merás, A. Method specific Cholesky decomposition: Coulomb and exchange energies. *J. Chem. Phys.* **2008**, *129*, 134107. (c) Aquilante, F.; Gagliardi, L.; Pedersen, T. B.; Lindh, R. Atomic Cholesky decompositions: A route to unbiased auxiliary basis sets for density fitting approximation with tunable accuracy and efficiency. *J. Chem. Phys.* **2009**, *130*, 154107.

(11) Li, X.-P.; Nunes, R. W.; Vanderbilt, D. Density-matrix electronic-structure method with linear system-size scaling. *Phys. Rev. B: Condens. Matter Mater. Phys.* **1993**, *47*, 10891–10894.

(12) (a) Challacombe, M. A simplified density matrix minimization for linear scaling self-consistent field theory. *J. Chem. Phys.* **1999**, *110*, 2332–2342. (b) Nunes, R. W.; Vanderbilt, D. Generalization of the density-matrix method to a nonorthogonal basis. *Phys. Rev. B: Condens. Matter Mater. Phys.* **1994**, *50*, 17611–17614. (c) Daniels, A. D.; Millam, J. M.; Scuseria, G. E. Semiempirical methods with conjugate gradient density matrix search to replace diagonalization for molecular systems containing thousands of atoms. *J. Chem. Phys.* **1997**, *107*, 425–431. (d) Bates, K. R.; Daniels, A. D.; Scuseria, G. E. Comparison of conjugate gradient density matrix search and Chebyshev expansion methods for avoiding diagonalization in large-scale electronic structure calculations. *J. Chem. Phys.* **1998**, *109*, 3308–3312. (e) Daniels, A. D.; Scuseria, G. E. What is the best alternative to diagonalization of the Hamiltonian in large scale semiempirical calculations? *J. Chem. Phys.* **1999**, *110*, 1321–1328. (f) Helgaker, T.; Larsen, H.; Olsen, J.; Jørgensen, P. Direct optimization of the AO density matrix in Hartree-Fock and Kohn-Sham theories. *Chem. Phys. Lett.* **2000**, *327*, 397–403. (g) Larsen, H.; Olsen, J.; Jørgensen, P.; Helgaker, T. Direct optimization of the atomic-orbital density matrix using the conjugate-gradient method with a multilevel preconditioner. *J. Chem. Phys.* **2001**, *115*, 9685–9697.

(13) (a) Ochsenfeld, C.; Head-Gordon, M. A reformulation of the coupled perturbed self-consistent field equations entirely within a local atomic orbital density matrix-based scheme. *Chem. Phys. Lett.* **1997**, *270*, 399–405. (b) Shao, Y.; Saravanan, C.; Head-Gordon, M. Curvy steps for density matrix-based energy minimization: Application to large-scale self-consistent-field calculations. *J. Chem. Phys.* **2003**, *118*, 6144–6151. (c) Head-Gordon, M.; Shao, Y.; Saravanan, C.; White, C. A. Curvy steps for density matrix based energy minimization: tensor formulation

and toy applications. *Mol. Phys.* **2003**, *101*, 37–43. (d) Ochsenfeld, C.; Kussmann, J.; Koziol, F. Ab Initio NMR Spectra for Molecular Systems with a Thousand and More Atoms: A Linear-Scaling Method. *Angew. Chem.* **2004**, *116*, 4585–4589. (e) Ochsenfeld, C.; Kussmann, J.; Koziol, F. Ab Initio NMR Spectra for Molecular Systems with a Thousand and More Atoms: A Linear-Scaling Method. *Angew. Chem., Int. Ed.* **2004**, *43*, 4485–4489.

(14) Guidon, M.; Hutter, J.; Vande Vondele, J. Auxiliary Density Matrix Methods for Hartree-Fock Exchange Calculations. *J. Chem. Theory Comput.* **2010**, *6*, 2348–2364.

(15) (a) Head-Gordon, M.; Maslen, P. E.; White, C. A. A tensor formulation of many-electron theory in a nonorthogonal single-particle basis. *J. Chem. Phys.* **1998**, *108*, 616–625. (b) Scuseria, G. E. Linear Scaling Density Functional Calculations with Gaussian Orbitals. *J. Phys. Chem. A* **1999**, *103*, 4782–4790.

(16) Jansík, B.; Høst, S.; Jørgensen, P.; Olsen, J.; Helgaker, T. Linear-scaling symmetric square-root decomposition of the overlap matrix. *J. Chem. Phys.* **2007**, *126*, 124104.

(17) Basic Linear Algebra Subprograms; http://www.netlib.org/blas. Accessed December 02, 2010.

(18) Linear Algebra Package; http://www.netlib.org/lapack. Accessed December 02, 2010.

(19) Shao, Y.; et al. Advances in methods and algorithms in a modern quantum chemistry program package. *Phys. Chem. Chem. Phys.* **2006**, *8*, 3172–3191.

(20) George, A. In Algorithms for Large Scale Linear Algebraic Systems; *NATO ASI Series C: Mathematical and Physical Sciences*; Althaus, G. W., Spedicato, E., Eds.; Kluwer Academic Publishers: New York, 1998; Vol. 508; pp 73−105.

(21) (a) Chen, Y.; Davis, T. A.; Hager, W. W.; Rajamanickam, S. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM T. Math. Software* **2008**, *35*, 22:1–14. (b) Davis, T. A.; Hager, W. W. Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves. *ACM T. Math. Software* **2009**, *35*, 27:1–23.

(22) (a) Schenk, O. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. Ph.D. thesis, ETH Zürich, 2000; (b) Schenk, O.; Gärtner, K. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generat. Comput. Syst.* **2004**, *20*, 475–487. (c) Schenk, O.; Gärtner, K. On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems. *Electron. T. Numer. Ana.* **2006**, *23*, 158–179.

(23) Liang, W.; Head-Gordon, M. An exact reformulation of the diagonalization step in electronic structure calculations as a set of second order nonlinear equations. *J. Chem. Phys.* **2004**, *120*, 10379–10384.

(24) Higham, N. J. Cholesky factorization. *WIREs Comput. Stat.* **2009**, *1*, 251–254.

(25) Higham, N. J. In *Reliable Numerical Computation*; Cox, M. G., Hammarling, S. J., Eds.; Oxford University Press: Oxford, U.K., 1990; pp 161−185.

(26) Gilbert, J. R. Predicting Structure in Sparse Matrix Computations. *SIAM J. Matrix Anal. Appl.* **1994**, *15*, 62–79.

(27) Higham, N. J.; Pothen, A. Stability of the partitioned inverse method for parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.* **1994**, *15*, 139–148.

(28) Tinney, W.; Walker, J. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* **1967**, *55*, 1801–1809.

(29) George, A.; Liu, J. W.-H. *Computer Solution of Large Sparse Positive Definite Systems*; Prentice-Hall: Englewood Cliffs, NJ, 1981.

(30) George, A.; Liu, J. W. H. An optimal algorithm for symbolic factorization of symmetric matrices. *SIAM J. Comput.* **1980**, *9*, 583–593.

(31) Ng, E.; Peyton, B. W. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Sci. Comput.* **1993**, *14*, 761–769.

(32) Liu, J. W. H.; Ng, E. G.; Peyton, B. W. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.* **1993**, *14*, 242–252.

(33) Liu, J. W. H. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.* **1990**, *11*, 134–172.

(34) (a) Rose, D. J.; Tarjan, R. E.; Lueker, G. S. Algorithmic aspects of vertex elimination. *SIAM J. Comput.* **1976**, *5*, 266–283. (b) Tarjan, R. E.; Yannakakis, M. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.* **1984**, *13*, 566–579.

(35) Yannakakis, M. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.* **1981**, *2*, 77–79.

(36) (a) Cuthill, E.; McKee, J. Reducing the bandwidth of sparse symmetric matrices. 1969;(b) Chan, W. M.; George, A. A linear time implementation of the reverse Cuthill-McKee algorithm. *BIT* **1980**, *20*, 8–14.

(37) Biermann, M. *Erkennen von Graphenklassen mittels lexikographischer Breitensuche*. M. Sc. Thesis, FernUniversität Hagen, 2007.

(38) (a) Liu, J. W. H. Modification of the minimum-degree algorithm by multiple elimination. *ACM T. Math. Software* **1985**, *11*, 141–153. (b) George, A.; Liu, W. H. The evolution of the minimum degree ordering algorithm. *SIAM Rev.* **1989**, *31*, 1–19.

(39) Amestoy, P. R.; Davis, T. A.; Duff, I. S. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl.* **1996**, *17*, 886–905.

(40) Khaira, M. S.; Miller, G. L.; Sheffler, T. J. Nested Dissection: A survey and comparison of various nested dissection algorithms; Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

(41) (a) George, A. Nested Dissection of a regular finite element mesh. *SIAM J. Numer. Anal.* **1973**, *10*, 345–363. (b) Armon, D.; Reif, J. Space and time efficient implementations of parallel nested dissection. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 1992; pp 344−352.

(42) (a) Pothen, A.; Rothberg, E.; Simon, H.; Wang, L. Parallel sparse Cholesky factorization with spectral nested dissection ordering. In *Proceedings of the Fifth SIAM Conference on Applied Linear Algebra*, 1994; pp 418−422. (b) Schulze, J.; Diekmann, R.; Preis, R. Comparing nested dissection orderings for parallel sparse matrix factorization. In *Proceedings of PDPTA '95, CSREA 96-1103*, 1995; pp 280−289. (c) Bornstein, C. F.; Maggs, B. M.; Miller, G. L. Tradeoffs between parallelism and fill in nested dissection. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 1999; pp 191−200. (d) Boman, E. G.; Wolf, M. M. A nested dissection approach to sparse matrix partitioning for parallel computations. Technical report, Sandia National Laboratories, NM, 2008.

(43) Karypis, G.; Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* **1998**, *20*, 359–392.

(44) (a) Kussmann, J.; Ochsenfeld, C. Linear-scaling method for calculating nuclear magnetic resonance chemical shifts using gauge-including atomic orbitals within Hartree-Fock and density-functional theory. *J. Chem. Phys.* **2007**, *127*, 054103. (b) Kussmann, J.; Ochsenfeld, C. A density matrix-based method for the linear-scaling calculation of dynamic second- and third-order properties at the Hartree-Fock and Kohn-Sham density functional theory levels. *J. Chem. Phys.* **2007**, *127*, 204103.

(45) Heggernes, P.; Eisenstat, S. C.; Kumfert, G.; Pothen, A. The computational complexity of the minimum degree algorithm. In *Proceedings of 14th Norwegian Computer Science Conference, NIK 2001*, University of Troms, Norway. Also available as ICASE Report 2001-42, NASA/CR2001-211421, NASA Langley Research, pages 98–109.

(46) Conroy, J. M. Parallel nested dissection. *Parallel Comput.* **1990**, *16*, 139–156.

(47) Liu, J. W. A compact row storage scheme for Cholesky factors using elimination trees. *ACM T. Math. Software* **1986**, *12*, 127–148.

(48) Tarjan, R. E. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* **1975**, *22*, 215–225.

(49) Tarjan, R. E. Applications of Path Compression on Balanced Trees. *J. ACM* **1979**, *26*, 690–715.

(50) Tarjan, R. E. Data Structures and Network Algorithms. Number 44 in CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, 1983.

(51) Gilbert, J. R.; Ng, E. G.; Peyton, B. W. An Efficient Algorithm to Compute Row and Column Counts for Sparse Cholesky Factorization. *SIAM J. Matrix Anal. Appl.* **1994**, *15*, 1075–1091.

(52) Ackermann, W. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann.* **1928**, *99*, 118–133.

(53) Anderson, R. J.; Woll, H. Wait-free parallel algorithms for the union-find problem. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, New York, NY, USA, 1991; pp 370–380.

(54) Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* **1972**, *1*, 146–160.

(55) (a) Zmijewski, E.; Gilbert, J. A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor. *Parallel Comput.* **1988**, *7*, 199–210. (b) Lewis, J. G.; Peyton, B. W.; Pothen, A. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM J. Sci. Stat. Comput.* **1989**, *10*, 1146–1173. (c) Geist, G. A.; Ng, E. Task scheduling for parallel sparse Cholesky factorization. *Int. J. Parallel Program.* **1990**, *18*, 291–314. (d) Gupta, A.; Kumar, V. A scalable parallel algorithm for sparse Cholesky factorization. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Washington, D.C., 1994; pp 793–802. (e) Rothberg, E.; Schreiber, R. Improved load distribution in parallel sparse Cholesky factorization. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Washington, D.C., 1994; pp 783–792. (f) Kumar, B.; Eswar, K.; Sadayappan, P.; Huang, C.-H. A reordering and mapping algorithm for parallel sparse Cholesky factorization. In *Proc. Scalable High Performance Computing Conference*, 1994.

(56) Liu, J. W. H. Reordering sparse matrices for parallel elimination. *Parallel Comput.* **1989**, *11*, 73–91.

(57) Liu, J. W. H. Equivalent sparse matrix reordering by elimination tree rotations. *SIAM J. Sci. Stat. Comput.* **1988**, *9*, 424–444.

(58) (a) Bird, R. S. On building trees with minimum height. *J. Funct. Program.* **1997**, *7*, 441–445. (b) Hsu, C.-H.; Peng, S.-L.; Shi, C.-H. Constructing a minimum height elimination tree of a tree in linear time. *Inf. Sci.* **2007**, *177*, 2473–2479.

(59) Manne, F. Reducing The Height Of An Elimination Tree Through Local Reorderings; Technical Report CS-51-91, University of Bergen, Norway, 1991.

(60) Pothen, A. The complexity of optimal elimination trees; Technical Report CS-88-16, Pennsylvania State University, USA, 1988.

(61) Benzi, M.; Tuma, M. Orderings for Factorized Sparse Approximate Inverse Preconditioners. *SIAM J. Sci. Comput.* **2000**, *21*, 1851–1868.

(62) Dongarra, J. In *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*; Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H., Eds.; SIAM: Philadelphia, PA , 2000.

(63) van Grondelle, J. *Symbolic Sparse Cholesky Factorisation Using Elimination Trees*. M.Sc. Thesis, Utrecht University, 1999.

(64) Flake, J. Getting the Best from your Cache Architecture. *Info. Quarterly* **2004**, *3*, 14–15.

(65) Patterson, D. A.; Hennessy, J. *Computer Organization and Design*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, 2004.

(66) (a) Goto, K.; van de Geijn, R. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, USA, 2002. (b) Goto, K.; van de Geijn, R. A. Anatomy of a High-Performance Matrix Multiplication. *ACM T. Math. Software* **2008**, *34*, 1–25. (c) Goto, K.; van de Geijn, R. High Performance Implementation of the Level-3 BLAS. *ACM T. Math. Software* **2008**, *35*, 4:1–14.

(67) Scott, J.; Hu, Y.; Gould, N. In *Applied Parallel Computing*; Dongarra, J., Madsen, K., Wasniewski, J., Eds.; Springer: Berlin/Heidelberg, 2006; Vol. 3732 pp 818−827.

(68) (a) Moore, E. H. On the reciprocal of the general algebraic matrix. *B. Am. Math. Soc.* **1920**, *26*, 394–395. (b) Penrose, R. A generalized inverse for matrices. *Math. Proc. Cambridge Philos. Soc.* **1955**, *51*, 406–413.

(69) (a) Hansen, P. C. Detection of near-singularity in Cholesky and LDLT factorizations. *J. Comput. Appl. Math.* **1987**, *19*, 293–299. (b) Barlow, J. L.; Vemulapati, U. B. Rank Detection Methods for Sparse Matrices. *SIAM J. Matrix Anal. A.* **1992**, *13*, 1279–1297.

(70) (a) Golub, G.; Kahan, W. Calculating the Singular Values and Pseudo-Inverse of a Matrix. *J. Soc. Ind. Appl. Math. B* **1965**, *2*, 205–224. (b) Loan, C. F. V. Generalizing the Singular Value Decomposition. *SIAM J. Numer. Anal.* **1976**, *13*, 76–83. (c) Golub, G.; Sølna, K.; Dooren, P. V. Computing the SVD of a General Matrix Product/Quotient. *SIAM J. Matrix Anal. A* **2000**, *22*, 1–19.

(71) Gilbert, J. R.; Ng, E. G. Predicting structure in nonsymmetric sparse matrix factorizations. In *Graph Theory and Sparse Matrix Computation*; Springer-Verlag, 1992; pp 107–139.

(72) Davis, T. Multifrontral multithreaded rank-revealing sparse QR factorization. In *Combinatorial Scientific Computing*, number 09061; Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Naumann, U., Schenk, O., Simon, H. D.,

Toledo, S., Eds.; Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

(73) (a) Berry, M. W.; Heath, M. T.; Kaneko, I.; Lawo, M.; Plemmons, R. J.; Ward, R. C. An Algorithm to Compute a Sparse Basis of the Null Space. *Numer. Math.* **1985**, *47*, 483–504. (b) Foster, L. V. Rank and Null Space Calculations Using Matrix Decompositions without Column Interchanges. *Linear Algebra Appl.* **1986**, *74*, 47–71. (c) Choi, S.-C. *Iterative Methods for Singular Linear Equations and Least-Squares Problems*, Ph.D. Thesis, Stanford University, CA, 2006; (d) Le Borne, S. Block computation and representation of a sparse nullspace basis of a rectangular matrix. *Linear Algebra Appl.* **2008**, *428*, 2455–2467. (e) Gotsman, C.; Toledo, S. On the Computation of Null Spaces of Sparse Rectangular Matrices. *SIAM J. Matrix Anal. A* **2008**, *30*, 445–463. (f) Wu, J.; Lee, Y.-J.; Xu, J.; Zikatanov, L. Convergence Analysis on Iterative Methods for Semidefinite Systems. *J. Comput. Math.* **2008**, *26*, 797–815.

(74) Arbenz, P.; Drmac, Z. On Positive Semidefinite Matrices with Known Null Space. *SIAM J. Matrix Anal. Appl.* **2002**, *24*, 132–149.

(75) Moler, C. B.; Stewart, G. W. On the Householder-Fox algorithm for decomposing a projection. *J. Comput. Phys.* **1978**, *28*, 82–91.

(76) Fox, K.; Krohn, B. J. Computation of cubic harmonics. *J. Comput. Phys.* **1977**, *25*, 386–408.

CT100618S