SelInv—An Algorithm for Selected Inversion of a Sparse Symmetric Matrix

LIN LIN, Princeton University
CHAO YANG and JUAN C. MEZA, Lawrence Berkeley National Laboratory
JIANFENG LU, Courant Institute of Mathematical Sciences
LEXING YING, University of Texas at Austin
WEINAN E, Princeton University

We describe an efficient implementation of an algorithm for computing selected elements of a general sparse symmetric matrix A that can be decomposed as $A = LDL^T$, where L is lower triangular and D is diagonal. Our implementation, which is called *SelInv*, is built on top of an efficient supernodal left-looking LDL^T factorization of A. We discuss how computational efficiency can be gained by making use of a relative index array to handle indirect addressing. We report the performance of SelInv on a collection of sparse matrices of various sizes and nonzero structures. We also demonstrate how SelInv can be used in electronic structure calculations.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: —*Algorithm design and analysis*; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Algebraic algorithms*

General Terms: Design, Performance

Additional Key Words and Phrases: Electronic structure calculation, elimination tree, selected inversion, sparse LDL^T factorization, supernodes

ACM Reference Format:
Lin, L., Yang, C., Meza, J. C., Lu, J., Ying, L., and E, W. 2010. SelInv—An algorithm for selected inversion of a sparse symmetric matrix. ACM Trans. Math. Softw. 37, 4, Article 40 (February 2011), 19 pages.
DOI = 10.1145/1916461.1916464 http://doi.acm.org/10.1145/1916461.1916464

This work was partially supported by NSF under Contract No. DMS-0708026 and No. DMS-0914336, by Doe under Contract No. DE-FG02-03ER25587, and by ONR under Contract No. N00014-01-1-0674 (L. Lin, J. Lu, and W. E); by an Alfred P. Sloan fellowship and a startup grant from the University of Texas at Austin (L. Ying); and by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 (C. Yang and J. C. Meza). The computational results presented were obtained at the National Energy Research Scientific Computing Center (NERSC), which is supported by the Director, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under contract number DE-AC02-05CH11232.
Authors' addresses: L. Lin, Department of Mathematics, Princeton University, 210 Fine Hall, Washington Road, Princeton, NJ 08544; email: `linlin@math.princeton.edu`; C. Yang and J. C. Meza, Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Rd., Berkeley, CA 94720; email: {`cyang, JCMeza`}`@LbL.gov`; J. Lu, Courant Institute of Mathematical Sciences, 1119 Warren Weaver Hall, 251 Mercer St., New York, NY 10012-1185; email: `jiangeng@cims.nyu.edu`; L. Ying, Department of Mathematics and ICES, University of Texas at Austin, 1 University Station/C1200, Austin, TX 78712; email: `lexing@math.utex.edu`; W. E., Department of Mathematics and Program in Applied Computational Mathematics, Princeton University, 207 Fine Hall Washington Road, Princeton, NJ 08544; email: `weinan@math.princeton.edu`.
© 2011 Association for Computer Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the [U.S.] Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or `permission@acm.org`.
© 2011 ACM 0098-3500/2011/02-ART40 $10.00
DOI 10.1145/1916461.1916464 http://doi.acm.org/10.1145/1916461.1916464

1. INTRODUCTION

In some scientific applications, we need to calculate a subset of the entries of the inverse of a given matrix. A particularly important example is in the electronic structure analysis of materials using algorithms based on pole expansion [Lin et al. 2009a, 2009c] where the diagonal and sometimes subdiagonals of the discrete Green's function or resolvent matrices are needed in order to compute the electron density. Other examples in which particular entries of the Green's functions are needed can also be found in the perturbation analysis of impurities by solving Dyson's equation in solid state physics [Economou 2006], or the calculation of retarded and less-than Green's function in electronic transport [Datta 1997]. We will call this type of calculations a *selected inversion* of a matrix.

From a computational viewpoint, it is natural to ask whether one can develop algorithms for selected inversion that are faster than inverting the whole matrix. This is possible at least in some cases, for example, when A is obtained from a finite difference discretization of a Laplacian operator or from lattice models in statistical or quantum mechanics with a local Hamiltonian. For such matrices, a fast sequential algorithm has been proposed to extract the diagonal or subdiagonal elements of their inverse matrices [Lin et al. 2009b]. The complexity of this fast selected inversion algorithm is $\mathcal{O}(n^{3/2})$ for two-dimensional (2D) problems and $\mathcal{O}(n^2)$ for three-dimensional (3D) problems, with n being the dimension of A. This is lower than the $\mathcal{O}(n^3)$ complexity associated with a direct inversion of a general matrix. For sparse matrices, the implementation of direct inversion can take advantage of the sparsity pattern of the sparse triangular factor of A. Although the complexity of this approach can be made lower than $\mathcal{O}(n^3)$, it is still generally higher than the complexity associated with selected, inversion, as we will see in Section 5. For a narrower class of matrices such as those obtained from discretized elliptic operators, the complexity of direct inversion can be reduced to $\mathcal{O}(n^2)$ in both 2D and 3D by making use of specialized algorithms such as the Fast Multipole Method [Greengard and Rokhlin 1987]. However, the selected inversion algorithm we present here is a more general approach. Hence, it can be applied to a wider class of problems.

The fast selected inversion algorithm in Lin et al. [2009b] contains two steps. The first step produces an LDL^T factorization of the input matrix A. The second step uses the L and D matrices to compute the selected components of A^{-1}. In the following, we will simply refer to the first step as *factorization*, and the second step as *selected inversion*. The parallelization of this algorithm on a distributed memory machine was described in the recent work [Lin et al. 2009d]. We have used the parallel algorithm to perform a selected inversion of a 2D Laplacian of dimension 4.3 billion on 4,096 processors.

The design and implementation of the fast algorithms proposed in Lin et al. [2009b; 2009d] depend explicitly on the domain shape and the discretization stencil for the Laplacian operator. This leads to an efficient implementation, but restricts the application of the algorithm. On the other hand, LDL^T factorization is a general concept. Therefore, it is natural to ask whether it is possible for the selected inversion algorithm to be generalized to any nonsingular symmetric matrix. This question was investigated in Takahashi et al. [1973] and Erisman and Tinney [1975] and discussed recently in Li et al. [2008] and Petersen et al. [2009]. However, no efficient software package is currently available for computing a selected inversion of a general sparse symmetric matrix that admits an LDL^T factorization. The present article intends to fill this gap by describing an efficient algorithm and its implementation for such a task. The algorithm and its implementation described here will be called *SelInv*.

Our article is organized as follows. We begin with the description of some basic concepts underlying a selected inversion algorithm in Section 2, and discuss why the complexity of the algorithm can be made lower than $\mathcal{O}(n^3)$ when A is sparse. We discuss the use of supernodes and block algorithms in Section 3, which is key to achieving high performance. The implementation details of SelInv, which are the main contributions of this article, are provided in Section 4. In particular, we show how a relative index array similar to that used in a sparse LDL^T factorization is set up to handle indirect addressing efficiently. In Section 5, we report the performance of SelInv on a collection of sparse matrices. We also demonstrate how SelInv can be used in electronic structure calculations in Section 6.

Standard linear algebra notation is used for vectors and matrices throughout the article. We use $A_{i,j}$ to denote the (i, j)th element of A. Block indices are denoted by uppercase script letters $\mathcal{I}$, $\mathcal{J}$, etc.. Occasionally, we use a MATLAB [Moler 2004] script to describe a simple algorithm. In particular, we use the MATLAB-style notation `A(i:j,k:l)` to denote a submatrix of A that consists of rows i through j and columns k through l. Another notion we use to denote such a submatrix is $A_{i:j,k:l}$. Furthermore, we use $A_{i,*}$ and $A_{*,j}$ to denote the ith row and the jth column of A, respectively. Similarly, $A_{\mathcal{I},*}$ and $A_{*,\mathcal{J}}$ are used to denote the $\mathcal{I}$th block row and the $\mathcal{J}$th block column of A, respectively.

2. SELECTED INVERSION: BASIC IDEA

An obvious way to obtain selected components of A^{-1} is to compute A^{-1} first and then simply pull out the needed entries. The standard approach for computing A^{-1} is to first decompose A as

$$A = LDL^T, \tag{1}$$

where L is a unit lower triangular matrix and D is a diagonal or a block-diagonal matrix. Equation (1) is often known as the LDL^T *factorization of* A. For positive definite matrices, D can always be kept as a diagonal matrix. For general symmetric matrices, a block LDL^T factorization that allows 2×2 block pivots [Bunch and Parlett 1971; Bunch and Kaufman 1977] or partial pivoting [Gilbert and Peierls 1986] may be used to achieve numerical stability in the factorization. Given such a factorization, one can obtain $A^{-1} = (x_1, x_2, \ldots, x_n)$ by solving a number of triangular systems

$$Ly = e_j, \quad Dw = y, \quad L^T x_j = w, \tag{2}$$

for $j = 1, 2, \ldots, n$, where e_j is the jth column of the identity matrix I. The computational cost of such algorithm is generally $\mathcal{O}(n^3)$, with n being the dimension of A. However, when A is sparse, we can exploit the sparsity structure of L and e_j to reduce the complexity of computing selected components of A^{-1}. We will examine this type of algorithm, which we will refer to as *direct inversion*, further in Section 5 when we compare the performance of direct inversion with that of our new fast algorithm.

The alternative algorithms presented in Lin et al. [2009b; 2009d] and summarized below also perform an LDL^T factorization of A first. However, the algorithm does not require solving (2) directly. Before we present this algorithm, it will be helpful to first review the major operations involved in the LDL^T factorization of A.

Let

$$A = \begin{pmatrix} \alpha & b^T \\ b & \hat{A} \end{pmatrix} \tag{3}$$

be a nonsingular symmetric matrix. The first step of an LDL^T factorization produces a decomposition of A that can be expressed by

$$A = \begin{pmatrix} 1 & \\ \ell & I \end{pmatrix} \begin{pmatrix} \alpha & \\ & \hat{A} - bb^T/\alpha \end{pmatrix} \begin{pmatrix} 1 & \ell^T \\ & I \end{pmatrix},$$

where α is often referred to as a pivot, $\ell = b/\alpha$ and $S = \hat{A} - bb^T/\alpha$ is known as the *Schur complement*. The same type of decomposition can be applied recursively to the Schur complement S until its dimension becomes 1. The product of lower triangular matrices produced from the recursive procedure, which all have the form

$$\begin{pmatrix} I & & \\ & 1 & \\ & \ell^{(i)} & I \end{pmatrix},$$

where $\ell^{(1)} = \ell = b/\alpha$, yields the final L factor. At this last step the matrix in the middle becomes diagonal, which is the D matrix.

To simplify our discussion, we assume here that all pivots produced in the LDL^T factorization are sufficiently large so that no row or column permutation (pivoting) is needed during the factorization. The discussion can be readily generalized if D contains 2×2 blocks.

The key observation made in Lin et al. [2009b; 2009d] is that A^{-1} can be expressed by

$$A^{-1} = \begin{pmatrix} \alpha^{-1} + \ell^T S^{-1} \ell & -\ell^T S^{-1} \\ -S^{-1}\ell & S^{-1} \end{pmatrix}. \tag{4}$$

This expression suggests that, once α and ℓ are known, the task of computing A^{-1} can be reduced to that of computing S^{-1}.

Because a sequence of Schur complements is produced recursively in the LDL^T factorization of A, the computation of A^{-1} can be organized in a recursive fashion too. Clearly, the reciprocal of the last entry of D is the (n, n)th entry of A^{-1}. Starting from this entry, which is also the 1×1 Schur complement produced in the $(n-1)$th step of the LDL^T factorization procedure, we can construct the inverse of the 2×2 Schur complement produced at the $(n-2)$th step of the factorization procedure, using the recipe given by (4). This 2×2 matrix is the trailing 2×2 block of A^{-1}. As we proceed from the lower right corner of L and D toward their upper left corner, more and more elements of A^{-1} are recovered. The complete procedure can be easily described by a MATLAB script shown in Algorithm 1.

Algorithm 1. A MATLAB script for computing the inverse of a dense matrix A given its LDL^T factorization.

Input: A unit triangular matrix L and a diagonal matrix D such that $A = LDL^T$;
Output: The inverse of A denoted by `Ainv`.

```
Ainv(n,n) = 1/D(n,n);
for j = n-1:-1:1
  Ainv(j+1:n,j) = -Ainv(j+1:n,j+1:n)*L(j+1:n,j);
  Ainv(j,j+1:n) = Ainv(j+1:n,j)';
  Ainv(j,j)     = 1/D(j,j) - L(j+1:n,j)'*Ainv(j+1:n,j);
end;
```

For the purpose of clarity, we use a separate array `Ainv` in Algorithm 1 to store the computed A^{-1}. In practice, A^{-1} can be computed in place. That is, we can overwrite the array used to store L and D with the lower triangular and diagonal part of A^{-1} incrementally.

It is not difficult to observe that, if A is a dense matrix, the complexity of Algorithm 1 is $\mathcal{O}(n^3)$ because a matrix vector multiplication involving a $j \times j$ dense matrix is performed at the jth iteration of this procedure, and $(n-1)$ iterations are required to fully recover A^{-1}. Therefore, when A is dense, this procedure does not offer any advantage over the standard way of computing A^{-1}. Furthermore, all elements of A^{-1} are needed and computed. No computational cost can be saved if we just want to extract selected elements (e.g., the diagonal elements) of A^{-1}.

However, when A is sparse, a tremendous amount of savings can be achieved if we are only interested in the diagonal components of A^{-1}. If the vector ℓ in (4) is sparse, computing $\ell^T S^{-1} \ell$ does not require all elements of S^{-1} to be obtained in advance. Only those elements that appear in the rows and columns corresponding to the nonzero rows of ℓ are required.

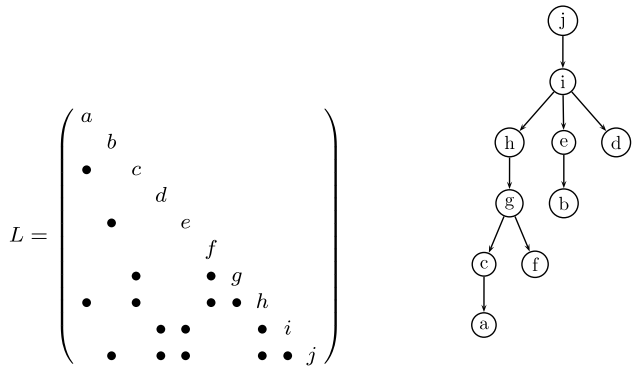
Therefore, to compute the diagonal elements of A^{-1}, we can simply modify the procedure shown in Algorithm 1 so that at each iteration we only compute selected elements of A^{-1} that will be needed by subsequent iterations of this procedure. It turns out that the elements that need to be computed are completely determined by the nonzero structure of the lower triangular factor L. To be more specific, at the jth step of the selected inversion process, we compute $(A^{-1})_{i,j}$ for all i such that $L_{i,j} \neq 0$. Therefore, our algorithm for computing the diagonal of A^{-1} can be easily illustrated by a MATLAB script (which is not the most efficient implementation) shown in Algorithm 2. A more efficient algorithm, which uses a special indirect addressing scheme to retrieve the needed entries of $(A^{-1})_{i,j}$, will be discussed in Section 4.

Algorithm 2. A MATLAB script for computing selected matrix elements of A^{-1} for a sparse symmetric matrix A.

Input: A unit triangular matrix L and a diagonal matrix D such that $A = LDL^T$;
Output: Selected elements of A^{-1} denoted by `Ainv`, i.e. the elements $(A^{-1})_{i,j}$ such that $L_{i,j} \neq 0$.

```
Ainv(n,n) = 1/D(n,n);
for j = n-1:-1:1
   % find the row indices of the nonzero elements in
   % the j-th column of L
   inz = j + find(L(j+1:n,j)~=0);
   Ainv(inz,j) = -Ainv(inz,inz)*L(inz,j);
   Ainv(j,inz) =  Ainv(inz,j)';
   Ainv(j,j)   =  1/D(j,j) - Ainv(j,inz)*L(inz,j);
end;
```

To see why this type of selected inversion is sufficient, we only need to examine the nonzero structure of the kth column of L for all $k < j$ since such a nonzero structure tells us which rows and columns of the trailing subblock of A^{-1} are needed to complete the calculation of the (k,k)th entry of A^{-1}. In particular, we would like to find out



(a) The L factor. (b) The elimination tree.

Fig. 1. The lower triangular factor L of a sparse 10×10 matrix A and the corresponding elimination tree.

which elements in the jth column of A^{-1} are required for computing $A^{-1}_{i,k}$ for any $k < j$ and $i \geq j$.

Clearly, when $L_{j,k} = 0$, the jth column of A^{-1} is not needed for computing the kth column of A^{-1}. Therefore, we only need to examine columns k of L such that $L_{j,k} \neq 0$. A perhaps not so obvious but critical observation is that, for these columns, $L_{i,k} \neq 0$ and $L_{j,k} \neq 0$ implies $L_{i,j} \neq 0$ for all $i > j$. Hence computing the kth column of A^{-1} will not require more matrix elements from the jth column of A^{-1} than those that have already been computed (in previous iterations,), that is, elements $(A^{-1})_{i,j}$ such that $L_{i,j} \neq 0$ for $i \geq j$.

These observations are well known in the sparse matrix factorization literature [Duff and Reid 1987; George and Liu 1981]. They can be made more precise by using the notion of *elimination tree* [Liu 1990]. In such a tree, each node or vertex of the tree corresponds to a column (or row) of A. Assuming A can be factored as $A = LDL^T$, a node p is the parent of a node j if and only if

$$p = \min\{i > j | L_{i,j} \neq 0\}.$$

If $L_{j,k} \neq 0$ and $k < j$, then the node k is a descendant of j in the elimination tree. An example of the elimination tree of a matrix A and its L factor are shown in Figure 1. Such a tree can be used to clearly describe the dependency among different columns in a sparse LDL^T factorization of A. In particular, it is not too difficult to show that constructing the jth column of L requires contributions from descendants of j that have a nonzero matrix element in the jth row [Liu 1990].

Similarly, we may also use the elimination tree to describe which selected elements within the trailing subblock A^{-1} are required in order to obtain the (j, j)th element of A^{-1}. In particular, it is not difficult to show that the selected elements must belong to the rows and columns of A^{-1} that are among the ancestors of j.

3. BLOCK ALGORITHMS AND SUPERNODES

The selected inversion procedure described in Algorithm 1 and its sparse version can be modified to allow a block of rows and columns to be modified simultaneously.

A block algorithm can be described in terms a block factorization of A. For example, if A is partitioned as

$$A = \begin{pmatrix} A_{11} & B_{21}^T \\ B_{21} & A_{22} \end{pmatrix},$$

its block LDL^T factorization has the form

$$A = \begin{pmatrix} I & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} A_{11} & \\ & A_{22} - B_{21}A_{11}^{-1}B_{21}^T \end{pmatrix} \begin{pmatrix} I & L_{21}^T \\ & I \end{pmatrix}, \tag{5}$$

where $L_{21} = B_{21}A_{11}^{-1}$ and $S = A_{22} - B_{21}A_{11}^{-1}B_{21}^T$ is the Schur complement. The corresponding block version of (4) can be expressed by

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} + L_{21}^T S^{-1} L_{21} & -L_{21}^T S^{-1} \\ -S^{-1}L_{21} & S^{-1} \end{pmatrix}.$$

There are at least three advantages of using a block algorithm:

(1) It allows us to use level 3 BLAS (Basic Linear Algebra Subroutine) to develop an efficient implementation by exploiting the memory hierarchy in modern microprocessors.
(2) When applied to sparse matrices, it tends to reduce the amount of indirect addressing overhead.
(3) It allows 2×2 block pivots that can be used to overcome numerical instabilities that may arise when A is indefinite.

When A is sparse, the columns of A and L can be partitioned into *supernodes*. A supernode is a maximal set of contiguous columns $\{j, j+1, \ldots, j+s\}$ of L such that they have the same nonzero structure below the $(j+s)$th row and the lower triangular part of $L_{j:j+s,j:j+s}$ is completely dense. An example of a supernode partition of the lower triangular factor L associated with a 49×49 sparse matrix A is shown in Figure 2. The definition of a supernode can be relaxed to include columns whose nonzero structures are nearly identical with adjacent columns. However, we will not be concerned with such an extension in this article. We will use uppercase script letters such as $\mathcal{J}$ to denote a supernode. Following the convention introduced in Ng and Peyton [1993], we will interpret $\mathcal{J}$ either as a supernode index or a set of column indices contained in that supernode depending on the context.

We should note here that the supernode partition of A or L is completely based on the nonzero structure of A. Although it is desirable to create supernodes that contain all 2×2 block pivots priori to numerical factorization of A, this is generally difficult to do for sparse matrices. When the size of a supernode is larger than 1, we can still use 2×2 block pivots within this supernode to improve numerical stability of the LDL^T factorization. This type of strategy is often used in multifrontal solvers [Duff and Reid 1983; Ashcraft et al. 1998].

We denote the set of row indices associated with the nonzero rows below the diagonal block of the $\mathcal{J}$th supernode by $S_{\mathcal{J}}$. These row indices are further partitioned into $n_{\mathcal{J}}$ disjoint subsets $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n_{\mathcal{J}}}$ such that $\mathcal{I}_i$ contains a maximal set of contiguous row indices and $\mathcal{I}_i \subset \mathcal{K}$ for some supernode $\mathcal{K} > \mathcal{J}$. Here $\mathcal{K} > \mathcal{J}$ means $k > j$ for all $k \in \mathcal{K}$ and $j \in \mathcal{J}$. In Figure 3, we show how the nonzero rows associated with one of the supernodes (the 26th supernode which begins at column 27) are partitioned. The

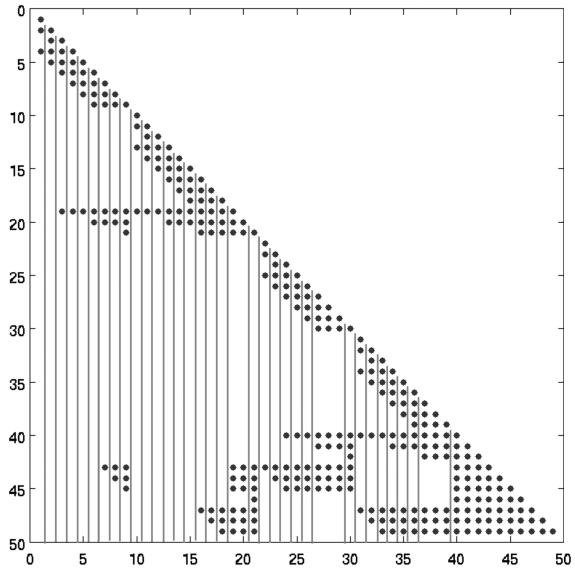


Fig. 2. A supernode partition of L.

purpose of the partition is to create dense submatrices of L that can be easily accessed and manipulated. The reason we impose the constraint $\mathcal{I}_i \subset \mathcal{K}$, which is normally not required in the LDL^T factorization of A, will become clear in Section 4. We should also note that, under this partitioning scheme, it is possible that $\mathcal{I}_i$ and $\mathcal{I}_j$ belong to the same supernode even if $i \neq j$.

The use of supernodes leads to a necessary but straightforward modification of the elimination tree. All nodes associated with columns within the same supernode are collapsed into a single node. The modified elimination tree describes the dependency among different supernodes in a supernode LDL^T factorization of A (see Ng and Peyton [1993]; Rothberg and Gupta [1994]). Such dependency also defines the order by which selected blocks of A^{-1} are computed.

Using the notion of supernodes, we can modify the selected inversion process described by the MATLAB script shown in Algorithm 2 to make it more efficient. If columns of L can be partitioned into n_{sup} supernodes, a supernode-based block selected inversion algorithm can be described by the pseudocode shown in Algorithm 3.

Algorithm 3. A supernode-based algorithm for computing the selected elements of A^{-1}.

Input: (1) The supernode partition of columns of A: $\{1, 2, \ldots, n_{sup}\}$;
(2) A supernode LDL^T factorization of A;
Output: Selected elements of A^{-1}, i.e. $(A^{-1})_{i,j}$ such that $L_{i,j} \neq 0$.

1: Compute $A^{-1}_{n_{sup},n_{sup}} = D^{-1}_{n_{sup},n_{sup}}$;
2: **for** $\mathcal{J} = n_{sup} - 1, n_{sup} - 2, \ldots, 1$
3: Identify the nonzero rows in the $\mathcal{J}$-th supernode $S_{\mathcal{J}}$;
4: Perform $Y = A^{-1}_{S_{\mathcal{J}},S_{\mathcal{J}}} L_{S_{\mathcal{J}},\mathcal{J}}$;
5: Calculate $A^{-1}_{\mathcal{J},\mathcal{J}} = D^{-1}_{\mathcal{J},\mathcal{J}} + Y^T L_{S_{\mathcal{J}},\mathcal{J}}$;
6: Set $A^{-1}_{S_{\mathcal{J}},\mathcal{J}} \leftarrow -Y$;
7: **end for**

4. IMPLEMENTATION DETAILS

We now describe some of the implementation details that allow the selected inversion process described schematically in Algorithm 3 to be carried out in an efficient manner.

We assume a supernode LDL^T factorization has been performed using, for example, an efficient left-looking algorithm described in Ng and Peyton [1993] and Rothberg and Gupta [1994]. Such an algorithm typically stores the nonzero elements of L in a contiguous array using the compressed column format [Duff et al. 1992]. This array will be overwritten by the selected elements of A^{-1}. The row indices associated with the nonzero rows of each supernode are stored in a separate integer array. Several additional integer arrays are used to mark the supernode partition and column offsets.

As we illustrated in Algorithm 3, the selected inversion process proceeds backward from the last supernode n_{sup} towards the first supernode. For all supernodes $\mathcal{J} < n_{sup}$, we need to perform a matrix-matrix multiplication of the form

$$Y = (A^{-1})_{S_{\mathcal{J}},S_{\mathcal{J}}} L_{S_{\mathcal{J}},\mathcal{J}}, \tag{6}$$

where $\mathcal{J}$ serves the dual purposes of being a supernode index and an index set that contains all column indices belonging to the $\mathcal{J}$th supernode, and $S_{\mathcal{J}}$ denotes the set of row indices associated with nonzero rows within the $\mathcal{J}$th supernode of L.

Recall that the row indices contained in $S_{\mathcal{J}}$ are partitioned into a number of disjoint subsets $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n_{\mathcal{J}}}$ such that $\mathcal{I}_i \subset \mathcal{K}$ for some supernode $\mathcal{K} > \mathcal{J}$. Such a partition corresponds to a row partition of the dense matrix block associated with the $\mathcal{J}$th supernode into $n_{\mathcal{J}}$ submatrices. The same partition is applied to the rows and columns of the submatrix $(A^{-1})_{S_{\mathcal{J}},S_{\mathcal{J}}}$ except that this submatrix is not stored in a contiguous array. For example, the nonzero row indices of the 26th supernode in Figure 2, which consists of columns 27, 28, and 29, can be partitioned as

$$S_{26} = \{30\} \cup \{40, 41\} \cup \{43, 44, 45\}.$$

This partition as well as the corresponding partition of the blocks in the trailing A^{-1} that are used in (6) is highlighted in Figure 3.

We carry out the matrix-matrix multiplication (6) by using Algorithm 4. The outer loop (line 2) of this algorithm goes through each block column of $(A^{-1})_{S_{\mathcal{J}},S_{\mathcal{J}}}$ indexed by $\mathcal{I}_j \in S_{\mathcal{J}}$, and accumulates $(A^{-1})_{*,\mathcal{I}_j} L_{\mathcal{I}_j,\mathcal{J}}$ in the dense matrix Y stored in a contiguous work array. The inner loop of this algorithm, which starts from line 6, simply goes through the nonzero blocks of $(A^{-1})_{*,\mathcal{I}_j}$ to perform $(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j} L_{\mathcal{I}_j,\mathcal{J}}$, $i = j+1, \ldots, n_{\mathcal{J}}$, one block at a time. Because A^{-1} is symmetric, we store only the selected nonzero elements in the lower triangular part of the matrix (except the diagonal blocks in which both the upper and lower triangular parts of the matrix are stored in a full dense matrix.) Hence, our implementation of (6) also computes the contribution of $(A^{-1})^T_{*,\mathcal{I}_j} L_{\mathcal{I}_i,\mathcal{J}}$ to Y as the $\mathcal{I}_j$th block column of A^{-1} is accessed (step 10) in the inner loop of Algorithm 4.

An efficient implementation of Algorithm 4 requires each subblock of $A^{-1}_{S_{\mathcal{J}},S_{\mathcal{J}}}$ (within the storage allocated for L) to be identified quickly and the product of $(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}$ and $L_{\mathcal{I}_j,\mathcal{J}}$, as well as the product of $[(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}]^T$ and $L_{\mathcal{I}_i,\mathcal{J}}$, to be placed at appropriate locations in the Y array. To achieve these goals, we use an integer array `indmap` with n entries to record the relative row positions of the first row of $\mathcal{I}_i$ in Y, for $i = 2, 3, \ldots, n_{\mathcal{J}}$. (The relative positions of all other nonzero rows can be easily calculated once the

Algorithm 4. Compute $Y = (A^{-1})_{S_{\mathcal{J}},S_{\mathcal{J}}} L_{S_{\mathcal{J}},\mathcal{J}}$ needed in Step 4 of Algorithm 3.

Input: (1) The $\mathcal{J}$-th supernode of L, $L_{S_{\mathcal{J}},\mathcal{J}}$, where $S_{\mathcal{J}}$ contains the indices of the nonzero rows in $\mathcal{J}$. The index set $S_{\mathcal{J}}$ is partitioned into disjoint $n_{\mathcal{J}}$ subsets of contiguous indices, i.e. $S_{\mathcal{J}} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n_{\mathcal{J}}}\}$;
(2) The nonzero elements of A^{-1} that have been computed previously. These elements are stored in $L_{S_{\mathcal{K}},\mathcal{K}}$ for all $\mathcal{K} > \mathcal{J}$;

Output: $Y = (A^{-1})_{S_{\mathcal{J}},S_{\mathcal{J}}} L_{S_{\mathcal{J}},\mathcal{J}}$;

1: Construct an `indmap` array for nonzero rows in the $\mathcal{J}$-th supernode;
2: **for** $j = 1, 2, \ldots, n_{\mathcal{J}}$
3: Identify the supernode $\mathcal{K}$ that contains $\mathcal{I}_j$;
4: Let $\mathcal{R}_1 = \texttt{indmap}(\mathcal{I}_j)$;
5: Calculate $Y_{\mathcal{R}_1,*} \leftarrow Y_{\mathcal{R}_1,*} + (A^{-1})_{\mathcal{I}_j,\mathcal{I}_j} L_{\mathcal{I}_j,\mathcal{J}}$;
6: **for** $i = j+1, j+2, \ldots n_{\mathcal{J}}$
7: Use `indmap` to find the first nonzero row in the $\mathcal{K}$-th supernode that belongs to $\mathcal{I}_i$ so that $(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}$ can be located;
8: Let $\mathcal{R}_2 = \texttt{indmap}(\mathcal{I}_i)$;
9: Calculate $Y_{\mathcal{R}_2,*} \leftarrow Y_{\mathcal{R}_2,*} + (A^{-1})_{\mathcal{I}_i,\mathcal{I}_j} L_{\mathcal{I}_j,\mathcal{J}}$;
10: Calculate $Y_{\mathcal{R}_1,*} \leftarrow Y_{\mathcal{R}_1,*} + [(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}]^T L_{\mathcal{I}_i,\mathcal{J}}$;
11: **end for**
12: **end for**
13: Reset the nonzero entries of `indmap` to zero;

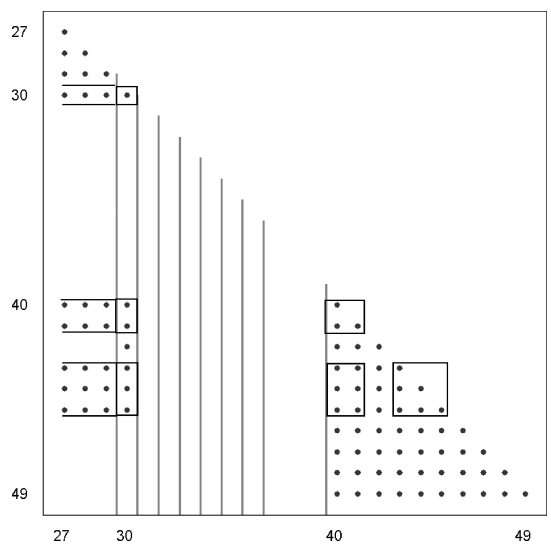
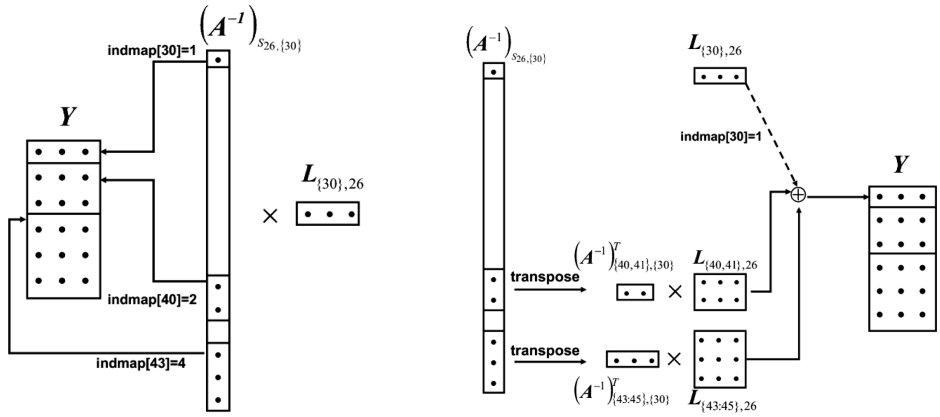


Fig. 3. The partition of the nonzero rows in S_{26} and the matrix elements needed in $A^{-1}_{30:49,30:49}$ for the computation of $A^{-1}_{30:49,30:49} L_{30:39,27:29}$.

relative row position of the first row of $\mathcal{I}_i$ is determined, because the row numbers in $\mathcal{I}_i$ are contiguous.) To be specific, all the entries of `indmap` are initialized to be zero. If k is an element in $\mathcal{I}_i$ (all elements in $\mathcal{I}_i$ are sorted in an ascending order), then `indmap[k]` is set to be the relative distance of row k from the last row of the diagonal block of the $\mathcal{J}$th supernode in L. For example, in Figure 3, the leftmost supernode S_{26},



(a) The use of `indmap` in Step 9.

(b) The use of `indmap` in Step 10. The dashed arrow indicates that $L_{\{30\},26}$ provides only the index for the row of Y that is modified.

Fig. 4. A schematic drawing that illustrates how `indmap` is used in Steps 9 and 10 in the first outer iteration of Algorithm 4 for $\mathcal{J} = 26$ in the example given in Figure 3.

which contains columns 27, 28, 29, contains six nonzero rows below its diagonal block. The nonzero entries of the `indmap` array for S_{26} are

```
indmap[30] = 1,
indmap[40] = 2,
indmap[41] = 3,
indmap[43] = 4,
indmap[44] = 5,
indmap[45] = 6.
```

These entries of the `indmap` array are reset to zeros once the calculation of (6) is completed for each $\mathcal{J}$. A similar indirect addressing scheme was used in Ng and Peyton [1993] for gathering the contributions from the descendants of the $\mathcal{J}$th supernode that have already been updated in the previous steps of a left-looking supernodal LDL^T factorization. Our use of indirect addressing collects contributions from the ancestors of the $\mathcal{J}$th supernode as $(A^{-1})_{S_{\mathcal{J}},\mathcal{J}}$ is being updated.

Once the `indmap` array is properly set up, the sub-block searching process indicated in line 7 of the pseudocode shown in Algorithm 4 goes through the row indices k of the nonzero rows of the $\mathcal{K}$th supernode (which contains $\mathcal{I}_j$) until a nonzero `indmap[k]` is found (step 7). A separate pointer p to the floating point array allocated for L is incremented at the same time. When a nonzero `indmap[k]` is found, the position in the floating point array pointed by p gives the location of $(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}$ required in line 9 of the special matrix-matrix multiplication procedure shown in Algorithm 4. Meanwhile, the value of `indmap[k]` gives the location of the target work array Y at which the product of $(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}$ and $L_{\mathcal{I}_j,\mathcal{J}}$ is accumulated. After lines 9 and 10 are executed in the inner loop of Algorithm 4, the remaining nonzero rows in the $\mathcal{K}$th supernode are examined until the next desired subblock in the $\mathcal{K}$th supernode of A^{-1} is found or until all nonzero rows within this supernode have been examined. Figure 4 shows of how the `indmap` array is used to place the product of $(A^{-1})_{\mathcal{I}_i,\{30\}}$ and $L_{\{30\},26}$ as well as the product of $(A^{-1})^T_{\mathcal{I}_i,\{30\}}$ and $L_{\mathcal{I}_i,26}$ in the Y array at lines 9 and 10 of Algorithm 4 for the example problem given in Figure 3.

Table I. Test Problems

Problem	Description
bcsstk14	Roof of the Omni Coliseum, Atlanta
bcsstk24	Calgary Olympic Saddledome arena
bcsstk28	Solid element model, linear statics
bcsstk18	R.E. Ginna Nuclear Power Station
bodyy6	NASA, Alex Pothen
crystm03	FEM crystal free vibration mass matrix
wathen120	Gould,Higham,Scott: matrix from Andy Wathen, Oxford Univ
thermal1	Unstructured FEM, steady state thermal problem, Dani Schmid, Univ. Oslo
shipsec1	DNV-Ex 4 : Ship section/detail from production run-1999-01-17
pwtk	Pressurized wind tunnel, stiffness matrix
parabolic_fem	Diffusion-convection reaction, constant homogeneous diffusion
tmt_sym	Symmetric electromagnetic problem, David Isaak, Computational_EM_Works
ecology2	Circuitscape: circuit theory applied to animal/gene flow, B. McRae, UCSB
G3_circuit	Circuit simulation problem, Ufuk Okuyucu, AMD, Inc

Table II. Characteristic of Test Problems

Problem	n	$\|A\|$	$\|L\|$
bcsstk14	1,806	32,630	112,267
bcsstk24	3,562	81,736	278,922
bcsstk28	4,410	111,717	346,864
bcsstk18	11,948	80,519	662,725
bodyy6	19,366	77,057	670,812
crystm03	24,696	304,233	3,762,633
wathen120	36,441	301,101	2,624,133
thermal1	82,654	328,556	2,690,654
shipsec1	140,874	3,977,139	40,019,943
pwtk	217,918	5,926,171	56,409,307
parabolic_fem	525,825	2,100,225	34,923,113
tmt_sym	726,713	2,903,837	41,296,329
ecology2	999,999	2,997,995	38,516,672
G3_circuit	1,585,478	4,623,152	197,137,253

Before we copy Y to the appropriate location in the array that stores the $\mathcal{J}$th supernode of L, we need to compute the diagonal block of A^{-1} within this supernode by the following update:

$$(A^{-1})_{\mathcal{J},\mathcal{J}} = (A^{-1})_{\mathcal{J},\mathcal{J}} + Y^T L_{S_{\mathcal{J}},\mathcal{J}},$$

where $(A^{-1})_{\mathcal{J},\mathcal{J}}$, which is stored in the diagonal block of the storage allocated for L, contains the inverse of the diagonal block $D_{\mathcal{J},\mathcal{J}}$ (which may contain 2×2 pivots) produced by the supernode LDL^T factorization before the update is performed.

5. PERFORMANCE

In this section we report the performance of our selected inversion algorithm SelInv. Our performance analysis is carried out on the Franklin Cray XT4 supercomputing system maintained at NERSC.[1] Each compute node consists of a 2.3-GHz single socket quad-core AMD Opteron processor (Budapest) with a theoretical peak performance of 9.2 GFlops/s per core (4 flops/cycle if using SSE128 instructions). Each core has 2 GB of memory. Our test problems are taken from the Harwell-Boeing Test Collection [Duff et al. 1992] and the University of Florida Matrix Collection [Davis 1997]. These matrices are widely used benchmark problems for sparse direct methods. The names of these matrices as well as some of their characteristics are listed in Tables I and II.

[1] http://www.nersc.gov/

Table III. Time Cost, and Flops Results for Factorization and Selected Inversion Process, Respectively; Last Column Reports the Average Flops Reached by SelInv

Problem	Factorization time (s)	Factorization flops (G/s)	Selected inversion time (s)	Selected inversion flops (G/s)	Average flops (G/s)
bcsstk14	0.007	1.43	0.010	2.12	1.85
bcsstk24	0.019	1.75	0.020	3.65	2.71
bcsstk28	0.023	1.63	0.024	3.46	2.54
bcsstk18	0.080	1.80	0.235	1.54	1.60
bodyy6	0.044	1.49	0.090	1.68	1.61
crystm03	0.452	2.26	0.779	2.95	2.70
wathen120	0.251	2.12	0.344	3.47	2.90
thermal1	0.205	1.53	0.443	1.66	1.62
shipsec1	18.48	2.38	17.66	5.45	3.88
pwtk	16.43	2.48	14.55	6.28	4.26
parabolic_fem	6.649	2.34	20.06	1.91	2.02
tmt_sym	10.64	2.35	13.98	4.02	3.30
ecology2	6.789	2.32	16.04	2.35	2.34
G3_circuit	136.5	2.24	218.7	3.27	2.88

All matrices are real and symmetric. The multiple minimum degree (MMD) matrix reordering strategy [Liu 1985] is used to minimize the amount of nonzero fills in L. We used the supernodal left-looking algorithm and code provided by Ng and Peyton [1993] to perform the LDL^T factorization of A. Table III gives the performance result in terms of computational time as well as floating-point operations per second (flops) for both the factorization and the selected inversion algorithms, respectively. We also report the average flops measured on-the-fly using PAPI [Browne et al. 2000]. The dimension of the matrices we tested ranges from 2000 to 1.5 million, and the number of nonzero elements in the L factor ranges from 0.1 million to 0.2 billion. For the largest problem G3_circuit, the overall computation takes only 350 s. Among these problems, the best performance is obtained with the problem pwtk. For this particular problem, the factorization part attains 26% of the peak performance of the machine, and the selected inversion part attains 68% of the peak flops. The average (of the factorization and inversion) flops ratio is 46%. The flops performance is directly related to the supernode size distribution due to the reordering strategy. For pwtk, 90% of the supernodes have sizes larger than 5. By contrast, the dimension of parabolic_fem is more than twice the dimension of pwtk, but 81% of the supernodes contain only one column. Consequently, SelInv cannot take full advantage of level 3 BLAS when it is used to solve this problem. As a result, its performance is worse on this problem than on pwtk.

To demonstrate how much we can gain by using the selected inversion algorithm, we compare the timing statistics of the selected inversion algorithm with that of the *direct inversion* algorithm mentioned in Section 2. In our implementation of the direct inversion algorithm, we compute the diagonal elements of A^{-1} using $e_j^T A^{-1} e_j = (L^{-1}e_j)^T D^{-1}(L^{-1}e_j)$, where e_j is the jth column of the identity matrix. When computing $y = L^{-1}e_j$ (via solving $Ly = e_j$), we modify only the nonzero entries of y. The positions of these entries can be predicted by the traversal of a directed graph constructed from the nonzero structure of L [Gilbert 1984]. This approach reduces the number of floating-point operations significantly compared to a naive approach that does not take into account the sparsity of L or e_j. However, it still has a higher asymptotic complexity compared to the selected inversion algorithm we presented earlier. This can be seen from the following example in which A is a discretized Laplacian operator obtained from applying a five-point stencil on an $m \times m$ grid in 2D where $m = n^{1/2}$. Assuming A is ordered by nested dissection [George 1973] so that the last m columns of A corresponds

Table IV. Timing Comparison Between Selected Inversion and Direct Inversion; Speedup Factor Defined by Direct Inversion Time Divided by Selected Inversion Time

Problem	Selected inversion time (s)	Direct inversion time (s)	Speedup factor
bcsstk14	0.01 sec	0.13 sec	13
bcsstk24	0.02 sec	0.58 sec	29
bcsstk28	0.02 sec	0.88 sec	44
bcsstk18	0.24 sec	5.73 sec	24
bodyy6	0.09 sec	5.37 sec	60
crystm03	0.78 sec	26.89 sec	34
wathen120	0.34 sec	48.34 sec	142
thermal1	0.44 sec	95.06 sec	216
shipsec1	17.66 sec	3346 sec	192
pwtk	14.55 sec	5135 sec	353
parabolic_fem	20.06 sec	7054 sec	352
tmt_sym	13.98 sec	>3 hours	>772
ecology2	16.04 sec	>3 hours	>673
G3_circuit	218.7 sec	>3 hours	>49

to nodes in the largest separator, we can see that solving $Ly = e_j$, for $j = n-m+1, \ldots, n$, would require a total of $\mathcal{O}(m^2) = \mathcal{O}(n)$ operations because the lower triangular part of $L_{n-m+1:n,n-m+1:n}$ is completely dense. Because these columns belong to a supernode that is at the root of the elimination tree, they are all reachable from node j on the directed graph constructed from solving $Ly = e_j$ for $j = 1, 2, \ldots, n-m$. Consequently, the overall complexity for solving $Ly = e_j$ for $j = 1, 2, \ldots, n$ is $\mathcal{O}((n-m)n + n) = \mathcal{O}(n^2)$. This is higher than the $\mathcal{O}(n^{3/2})$ complexity associated with selected inversion. Similarly, if A is a discretized Laplacian operator obtained from applying a seven-point stencil on an $m \times m \times m$ grid in 3D where $m = n^{1/3}$, the complexity of direct inversion becomes $\mathcal{O}(n^{7/3})$ because the largest separator contains $n^{2/3}$ columns, whereas the complexity of selected inversion is $\mathcal{O}(n^2)$.

Although it is difficult to perform such analysis for a general sparse matrix, similar difference in complexity should hold. To provide a more concrete comparison, we list the timing measurements for both approaches in Table IV as well as the speedup factor. The speedup factor is defined by the time for selected inversion divided by the time for direct inversion. In this comparison, selected inversion refers to the second part of SelInv, that is, the time for LDL^T factorization is not counted since factorization is used in both algorithms. We also terminate the direct inversion algorithm if its running time is larger than 3 h. We observe that, for the smallest problem bcsstk14, the speedup factor is already 13. For larger problems, the speedup can be factors of several hundreds or more.

6. APPLICATION TO ELECTRONIC STRUCTURE CALCULATION OF ALUMINUM

In this section, we show how SelInv can be applied to electronic structure calculations within the density functional theory (DFT) framework [Hohenberg and Kohn 1964; Kohn and Sham 1965]. We will focus on methods based on self-consistent field iteration [Martin 2004]. The most time-consuming part of these calculations is the evaluation of the electron density

$$\rho = \mathrm{diag}\,(f_{\beta,\mu}(H)), \tag{7}$$

where $f_{\beta,\mu}(t) = 1/(1 + e^{\beta(t-\mu)})$ is the Fermi-Dirac distribution function with β being a parameter that is proportional to the reciprocal of the temperature and μ being

the chemical potential. The symmetric matrix H in (7) is a discretized Kohn-Sham Hamiltonian [Martin 2004] defined as

$$H = -\frac{1}{2}\Delta + V_{\text{pse}}(\mathbf{r}) + V_H(\mathbf{r}) + V_{\text{xc}}(\mathbf{r}), \tag{8}$$

where Δ is the Laplacian, V_H is the Hartree potential, V_{xc} is the exchange-correlation potential constructed via the local density approximation (LDA) theory [Martin 2004], and V_{pse} is the real space Troullier-Martins ionic pseudopotential [Chelikowsky et al. 1994].

There is another class of methods that are designed to minimize the total energy of an atomistic system directly [Payne et al. 1992]. Although these algorithms do not evaluate the charge density ρ via (7), there are other computational issues one must address in order to make them efficient. We will not discuss these issues in this article.

One of the standard approaches for evaluating (7) is to compute the invariant subspace associated with a few smallest eigenvalues of H. This approach is used in, for example, PARSEC [Alemany et al 2004], which is a real-space electronic structure calculation software package. Other algorithms such as subspace iteration methods [Bekas et al. 2005; Zhou et al. 2006], which reduce the cost of orthogonalization can also be used. The computational cost of these algorithms is generally $\mathcal{O}(n^3)$ for metallic systems that contain a large number of electrons.

An alternative way to evaluate (7) is to use a recently developed pole expansion technique [Lin et al. 2009a] to approximate $f_{\beta,\mu}$. The pole expansion technique expresses electron density ρ as a linear combination of the diagonal of $(H - (\mu + z_i)I)^{-1}$, that is,

$$\rho \approx \sum_{i=1}^{P} \text{Im}\left(\text{diag}\,\frac{\omega_i}{H - (\mu + z_i)I}\right). \tag{9}$$

Here Im (H) stands for the imaginary part of H. The parameters z_i and ω_i are the complex shift and weight associated to the ith pole, respectively. They can be chosen so that the total number of poles P is minimized for a given accuracy requirement. At room temperature, the number of poles required in (9) is relatively small (less than 80). In addition to temperature, the pole expansion (9) also requires an explicit knowledge of the chemical potential μ, which must be chosen so that the condition

$$\text{trace}(f_{\beta,\mu}(H)) = n_e \tag{10}$$

is satisfied. This can be accomplished by solving (10) using a standard Newton's method [Burden et al. 2000] or other root-finding algorithms.

In order to use (9), we need to compute the diagonal of the inverse of a number of complex symmetric (non-Hermitian) matrices $H - (z_i + \mu)I$ $(i = 1, 2, \ldots, P)$. A fast implementation of the SelInv algorithm described in Section 4 can be used to perform this calculation efficiently, as the following example shows.

The example we consider here is a quasi-2D aluminum system with a periodic boundary condition. For simplicity, we only use a local pseudopotential in (8), that is, $V_{\text{pse}}(\mathbf{r})$ is a diagonal matrix. The Laplacian operator Δ is discretized using a second-order seven-point stencil. A room temperature of 300K (which defines the value of β) is used. The aluminum system has a face-centered cubic (FCC) crystal structure. We include five-unit cells along both x and y directions, and a one-unit cell along the z direction in our computational domain. Each unit cell is cubic with a lattice constant of 4.05 Å. Therefore, there are altogether 100 aluminum atoms and 300 valence

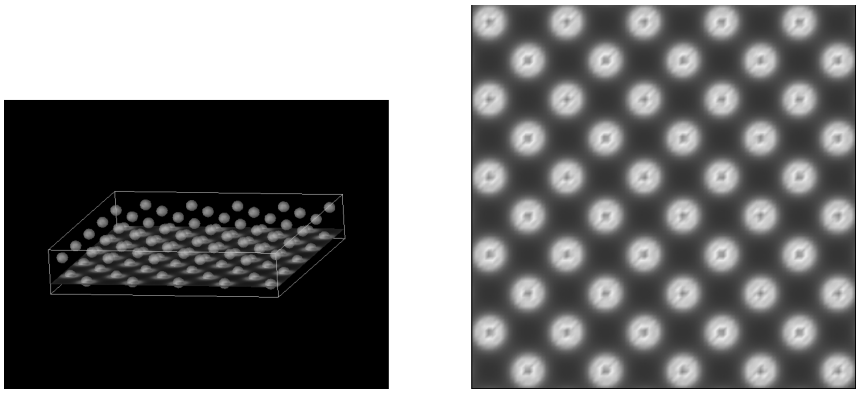


Fig. 5. (a) 3D isosurface plot of the electron density together with the electron density restricted to $z = 0$ plane. (b) The electron density restricted to $z = 0$ plane.

electrons in the experiment. The position of each aluminum atom is perturbed from its original position in the crystal by a random displacement around 10^{-3}Å so that no point group symmetry is assumed in our calculation. The grid size for discretization is set to 0.21 Å. The resulting Hamiltonian matrix size is 159,048.

We compare the density evaluation (7) performed by both PARSEC and the pole expansion technique. In PARSEC, the invariant subspace associated with the smallest 310 eigenvalues is computed using ARPACK [Lehoucq et al. 1998]. Each self-consistent iteration step takes 2490 s. In the pole expansion approach, we use 60 poles in (9), which gives a comparable relative error in electron density on the order of 10^{-5} (in L_1 norm.) The MMD reordering scheme is used to reduce the amount of fill in the LDL^T factorization. In addition to using the selected inversion algorithm to evaluate each term in (9), an extra level of coarse-grained parallelism can be utilized by assigning each pole to a different processor. The evaluation of each term in (9) takes roughly 1950 s. Therefore, the total amount of time required to evaluate (7) for each self-consistent iteration step on a single core is 1950×60 s. As a result, the performance of the selected inversion based pole expansion approach is only comparable to the invariant subspace computation approach used in PARSEC if the extra level of coarse-grained parallelism is used.

A 3D isosurface plot of the electron density as well as the electron density plot restricted on the $z = 0$ plane are shown in Figure 5.

We also remark that the efficiency of selected inversion can be further improved for this particular problem. One of the factors that has prevented the SelInv from achieving even higher performance for this problem is that most of the supernodes produced from the MMD ordering of H contain only 1 column even though many of these supernodes have similar (but not identical) nonzero structures. Consequently, both the factorization and inversion are dominated by level 1 BLAS operations. Further performance gain is likely to be achieved if we relax the definition of a supernode and treat some of the zeros in L as nonzero elements. This approach has been demonstrated to be extremely helpful in Ashcraft and Grimes [1989].

7. CONCLUDING REMARKS

We presented an efficient sequential algorithm for computing selected components of the inverse of a general sparse symmetric matrix A, and described its implementation SelInv. Our algorithm consists of two steps. In the first step, we perform an LDL^T factorization of the matrix A using a supernodal left-looking LDL^T factorization

algorithm developed in Ng and Peyton [1993]. This step can also be implemented using other existing software packages such as those in Ng and Peyton [1993], Amestoy et al. [2001], Schenk and Gartner [2006], Ashcraft and Grimes [1999], Gupta et al. [1997], Gupta [1997], and Raghavan [2002]. In the second step, a selected inversion algorithm specifically designed for the supernodal nonzero structure of L is used to compute the nonzero blocks of A^{-1} that have a corresponding nonzero block in L. The use of supernodes enables us to exploit the memory hierarchy of modern microprocessors to achieve high performance. We described an efficient indirect addressing scheme for gathering contributions from the ancestors of a supernode $\mathcal{J}$ when the selected components of A^{-1} within $(A^{-1})_{*,\mathcal{J}}$ are computed.

We demonstrated the efficiency of our implementation of the selected inversion algorithm by applying our code SelInv to a variety of benchmark problems with dimension as large as 1.5 million. We were able to achieve a relatively high percentage of the peak performance on the high performance machine we used to conduct our experiments. In one case, we were able to achieve 68% of the peak performance.

We also demonstrated how SelInv can be applied to the electronic structure calculation of an aluminum system using a pole expansion technique [Lin et al. 2009a, 2009c]. We compared the efficiency of our algorithm with the standard real-space electronic structure calculation software PARSEC. Our comparison showed that the performance of the pole expansion approach is comparable to that of PARSEC if a coarse-grained parallelization of the poles is used.

ACKNOWLEDGMENTS

We would like to thank Esmond Ng and Sherry Li for helpful discussion. We would also like to thank anonymous referees for helpful comments and suggestions.

REFERENCES

ALEMANY, M., JAIN, M., KRONIK, L., AND CHELIKOWSKY, J. 2004. Real-space pseudopotential method for computing the electronic properties of periodic systems. *Phys. Rev. B 69*, 075101.

AMESTOY, P., DUFF, I., L'EXCELLENT, J.-Y., AND KOSTER, J. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl. 23*, 15–41.

ASHCRAFT, C. AND GRIMES, R. 1989. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw. 15*, 291–309.

ASHCRAFT, C. AND GRIMES, R. 1999. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the 9th SIAM Conference on Parallel Processing*.

ASHCRAFT, C., GRIMES, R. G., AND LEWIS, J. G. 1998. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl. 20*, 513–561.

BEKAS, C., KOKIOPOULOU, E., AND SAAD, Y. 2005. Polynomial filtered Lanczos iterations with applications in Density Functional Theory. *SIAM J. Matrix Anal. Appl. 30,* 1, 397–418.

BROWNE, S., DONGARRA, J., GARNER, N., HO, G., AND MUCCI, P. 2000. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. C. 14*, 189.

BUNCH, J. R. AND KAUFMAN, L. 1977. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.* 163–179.

BUNCH, J. R. AND PARLETT, B. N. 1971. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numer. Anal. 8*, 639–655.

BURDEN, R. L., FAIRES, J. D., AND REYNOLDS, A. C. 2000. *Numerical Analysis*. Brooks Cole, Independence, KY.

CHELIKOWSKY, J., TROULLIER, N., AND SAAD, Y. 1994. Finite-difference-pseudopotential method: Electronic structure calculations without a basis. *Phys. Rev. Lett. 72*, 1240–1243.

DATTA, S. 1997. *Electronic Transport in Mesoscopic Systems*. Cambridge University Press, Cambridge, U.K.

DAVIS, T. 1997. University of Florida sparse matrix collection. *NA Digest 97*, 7.

DUFF, I. AND REID, J. 1987. *Direct Methods for Sparse Matrices*. Oxford University, London, UK.

DUFF, I., GRIMES, R., AND LEWIS, J. 1992. Users guide for the Harwell-Boeing sparse matrix collection. Research and Technology Division, Boeing Computer Services, Seattle, WA.

DUFF, I. S. AND REID, J. K. 1983. The multifrontal solution of indefinite sparse symmetric sets of linear equations. *ACM Trans. Math. Softw. 9*, 302–325.

ECONOMOU, E. 2006. *Green's Functions in Quantum Physics*. Springer Berlin, Germany.

ERISMAN, A. AND TINNEY, W. 1975. On computing certain elements of the inverse of a sparse matrix. *Comm. ACM 18*, 177.

GEORGE, A. 1973. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal. 10*, 345–363.

GEORGE, A. AND LIU, J. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ.

GILBERT, J. 1984. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl. 15,* 4, 62–79.

GILBERT, J. AND PEIERLS, T. 1986. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Stat. Comput. 9*, 5, 862–874.

GREENGARD, L. AND ROKHLIN, V. 1987. A fast algorithm for particle simulations. *J. Comput. Phys. 73,* 2, 325–348.

GUPTA, A. 1997. WSMP: The Watson Sparse Matrix Package. IBM Research rep. RC 21886(98462).

GUPTA, A., KARYPIS, G., AND KUMAR, V. 1997. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parall. Distrib. Syst. 8*, 502–520.

HOHENBERG, P. AND KOHN, W. 1964. Inhomogeneous electron gas. *Phys. Rev. 136*, B864.

KOHN, W. AND SHAM, L. 1965. Self-consistent equations including exchange and correlation effects. *Phys. Rev. 140*, A1133.

LEHOUCQ, R., SORENSEN, D., AND YANG, C. 1998. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA.

LI, S., AHMED, S., KLIMECK, G., AND DARVE, E. 2008. Computing entries of the inverse of a sparse matrix using the FIND algorithm. *J. Comput. Phys. 227*, 9408–9427.

LIN, L., LU, J., CAR, R., AND E, W. 2009a. Multipole representation of the Fermi operator with application to the electronic structure analysis of metallic systems. *Phys. Rev. B 79*, 115133.

LIN, L., LU, J., YING, L., CAR, R., AND E, W. 2009b. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Comm. Math. Sci. 7*, 755–777.

LIN, L., LU, J., YING, L., AND E, W. 2009c. Pole-based approximation of the Fermi-Dirac function. *Chinese Ann. Math. 30B*, 729–742.

LIN, L., YANG, C., LU, J., YING, L., AND E, W. 2009d. A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2D electronic structure calculations. Tech. rep. LBNL-2677E. Lawrence Berkeley National Laboratory, Berkeley.

LIU, J. 1985. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Softw. 11*, 141–153.

LIU, J. 1990. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl. 11*, 134.

MARTIN, R. 2004. *Electronic Structure—Basic Theory and Practical Methods*. Cambridge University Press, Cambridge. UK.

MOLER, C. 2004. *Numerical Computing with MATLAB*. SIAM, Philadelphia, PA.

NG, E. AND PEYTON, B. 1993. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput. 14*, 1034.

PAYNE, M. C., TETER, M. P., ALLEN, D. C., ARIAS, T. A., AND JOANNOPOULOS, J. D. 1992. Iterative minimization techniques for *ab initio* total energy calculation: Molecular dynamics and conjugate gradients. *Rev. Mod. Phys. 64*, 4, 1045–1097.

PETERSEN, D., LI, S., STOKBRO, K., SØRENSEN, H., HANSEN, P., SKELBOE, S., AND DARVE, E. 2009. A hybrid method for the parallel computation of Green's functions. *J. Comput. Phys. 228*, 5020–5039.

RAGHAVAN, P. 2002. DSCPACK: Domain-separator codes for solving sparse linear systems. Tech. rep. CSE-02-004. Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA.

ROTHBERG, E. AND GUPTA, A. 1994. An efficient block-oriented approach to parallel sparse choleskyfactorization. *SIAM J. Sci. Comput. 15*, 1413–1439.

SCHENK, O. AND GARTNER, K. 2006. On fast factorization pivoting methods for symmetric indefinite systems. *Elec. Trans. Numer. Anal. 23*, 158–179.

TAKAHASHI, K., FAGAN, J., AND CHIN, M. 1973. Formation of a sparse bus impedance matrix and its application to short circuit study. In *Proceedings of the 8th PICA Conference*. Minneapolis, MN.

ZHOU, Y., SAAD, Y., TIAGO, M. L., AND CHELIKOWSKY, J. R. 2006. Self-consistent-field calculations using chebyshev-filtered subspace iteration. *J. Comput. Phys. 219*, 172–184.

Received October 2009; revised March 2010; accepted August 2010