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

Lin Lin [*] Chao Yang [†] Juan C. Meza [‡] Jianfeng Lu [§] Lexing Ying [¶]
Weinan E [‖]

October 16, 2009

Abstract

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 calcuations.

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 [18, 20] where the diagonal and sometimes sub-diagonals 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 [10], or the calculation of retarded and less-than Green's function in electronic transport [6]. We will call this type of calculations a *selected inversion* of a matrix.

[*]Program in Applied and Computational Mathematics, Princeton University, Princeton, NJ 08544. Email: linlin@math.princeton.edu

[†]Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. Email: cyang@lbl.gov

[‡]Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. Email: JCMeza@lbl.gov

[§]Department of Mathematics, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012. Email:jianfeng@cims.nyu.edu

[¶]Department of Mathematics and ICES, University of Texas at Austin, 1 University Station/C1200, Austin, TX 78712. Email: lexing@math.utexas.edu

[‖]Department of Mathematics and PACM, Princeton University, Princeton, NJ 08544. Email: weinan@math.princeton.edu

From a computational viewpoint, it is natural to ask whether one can develop algorithms for selected inversion that is 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 sub-diagonal elements of their inverse matrices [19]. The complexity of the fast algorithm is $O(n^{1.5})$ for two dimensional (2D) problems and $O(n^2)$ for three dimensional (3D) problems, with n being the dimension of A. This is lower than the $O(n^3)$ complexity associated with a direct inversion of the full matrix. The algorithm presented in [19] contains two steps. The first step produces an LDL^T factorization of the input matrix A. The second step uses 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 is described in the recent work [21]. We have used the parallel algorithm to perform a selected inversion a 2D Laplacian of dimension 4.3 billion on 4,096 processors.

The design and implementation of the fast algorithm proposed in [19] and [21] depend explicitly on the domain shape and 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 [11], and discussed recently in [17, 27, 21]. 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 paper intends to fill such a 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 paper 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 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 paper. 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 [25] 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 i-th row and the j-th 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 matrix. Equation (1) is often known as the LDL^T factorization of A. 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.

An alternative algorithm presented in [21] and summarized below also performs 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

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

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

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

where α is often referred to as a pivot, $\ell = a/\alpha$ and $S = \hat{A} - aa^T/\alpha$ 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 $\ell^{(1)} = \ell = a/\alpha$, yields the final L factor. The $(1,1)$ entry of each Schur complement together with α become the diagonal entries of 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 key observation made in [19] and [21] 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 are produced recursively in the LDL^T factorization of A, the computation of A^{-1} can be organized in a recursive fashion also. 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 towards 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.

```
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 clarity purpose, 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 computation 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 saving 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 described by a MATLAB script (which is not the most efficient implementation) shown in Algorithm 2.

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 sub-block of A^{-1} are needed to complete the calculation of the (k, k) entry of A^{-1}. In particular, we would like to find out which

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

```
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) - Ainv(j,inz)*L(inz,j);
end;
```

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

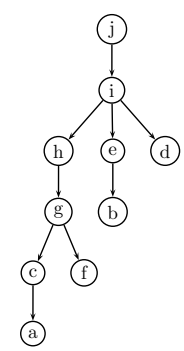
Clearly, when $L_{j,k} = 0$, the jth column of A^{-1} is not needed for computing the kth column of A^{-1}. Therefore, we only need to examine columns k of L such that $L_{j,k} \neq 0$. A perhaps not so obvious but critical observation is that for these columns, $L_{i,k} \neq 0$ and $L_{j,k} \neq 0$ 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,) i.e. 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 [9, 12]. They can be made more precise by using the notion of *elimination tree* [23]. 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

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

$$L = \begin{pmatrix} a & & & & & & & & & \\ & b & & & & & & & & \\ \bullet & & c & & & & & & & \\ & & & d & & & & & & \\ & \bullet & & & e & & & & & \\ & & & & & f & & & & \\ & & \bullet & & & \bullet & g & & & \\ \bullet & & \bullet & & & \bullet & \bullet & h & & \\ & & & \bullet & \bullet & & & \bullet & i & \\ & \bullet & & \bullet & \bullet & & & \bullet & \bullet & j \end{pmatrix}$$

(a) The L factor.



(b) The elimination tree.

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

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.

Similarly, we may also use the elimination tree to describe which selected elements within the trailing sub-block 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 two 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 memory hierarchy in modern microprocessors;

2. When applied to sparse matrices, it tends to reduce the amount of indirect addressing overhead.

When A is sparse, the columns of A and L can be partitioned into *supernodes*. A supernode is a maximal set of contiguous columns $\{j, j+1, ..., j+s\}$ of L that 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 paper. We will use upper case script letters such as $\mathcal{J}$ to denote a supernode. Following the convention introduced in [26], we will interpret $\mathcal{J}$ either as a supernode index or a set of column indices contained in that supernode depending on the context.

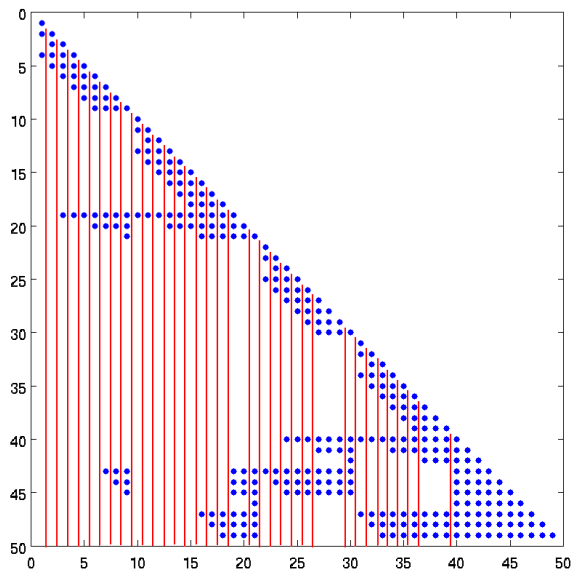


Figure 2: A supernode partition of L.

We denote the set of row indices associated with the nonzeros 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 $\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_{n_{\mathcal{J}}}$ 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 26-th supernode which begins at column 27) are partitioned. The 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 [26, 28]). 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.

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.

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

Input: (1) The supernode partition of columns of A: $\{1, 2, ..., n_{sup}\}$;
(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** $\mathcal{J} = n_{sup} - 1, n_{sup} - 2, ..., 1$ **do**
3: Identify the non-zero 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**

We assume a supernode LDL^T factorization has been performed using, for example, an efficient left-looking algorithm described in [26, 28]. Such an algorithm typically stores the nonzero elements of L in a contiguous array using the compressed column format [8]. 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 2, the selected inversion process proceeds backwards 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 $\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_{n_{\mathcal{J}}}$ 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 26-th 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.

In order to carry out the matrix-matrix multiplication (6) we must identify the location of each subblock of $A^{-1}_{S_{\mathcal{J}},S_{\mathcal{J}}}$ (within the storage allocated for L) one by one as we accumulate the products of these subblocks and the corresponding subblocks of $L_{S_{\mathcal{J}},\mathcal{J}}$ in a work array which we denote by Y. Furthermore, because A^{-1} is symmetric, we store only the selected nonzero elements in the lower triangular part of the matrix. Hence, our implementation of (6), which is carefully described in Algorithm 4, makes uses of the transposes of the subblocks in the lower triangular part of $A^{-1}_{S_{\mathcal{J}},S_{\mathcal{J}}}$ (line 10).

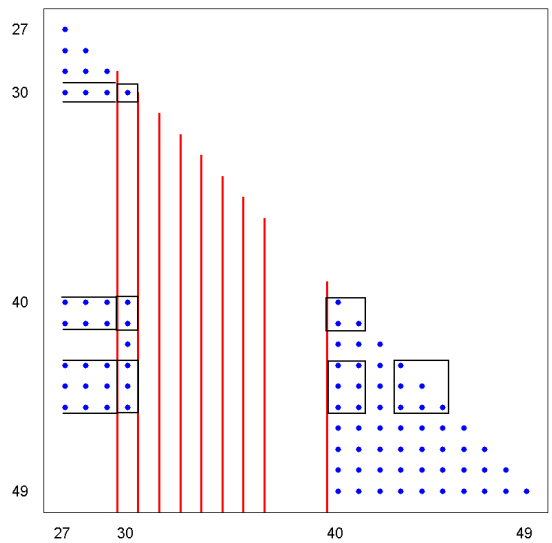


Figure 3: The partition of the nonzero rows in S_{26} and the matrix elements needed in $A^{-1}_{30:49,30:49}$ for the computation of $A^{-1}_{30:49,30:49}L_{30:39,27:29}$.

To identify the location of each subblock of $A^{-1}_{S_{\mathcal{J}},S_{\mathcal{J}}}$ required in (6), 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 $i = 2, 3, ..., n_{\mathcal{J}}$. To be specific, the `indmap` array is initialized to have 0's in all its entries. 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 in the diagonal block of the $\mathcal{J}$th supernode in L. For example, S_{26} contains columns $27, 28, 29$ and is shown as the leftmost supernode in Figure 3. 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.
```

A similar indirect addressing scheme is used in [26] for the gathering and scattering operations used in the LDL^T factorization of A.

Once the `indmap` array is properly set up, the subblock 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 until a nonzero `indmap[k]` is found. 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 $(A^{-1})_{\mathcal{I}_j,\mathcal{I}_j}$ 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 $(A^{-1})_{\mathcal{I}_j,\mathcal{I}_j}$ and $L_{\mathcal{I}_j,\mathcal{J}}$ is accumulated.

Algorithm 4 Sparse matrix-matrix multiplication.

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. $S_\mathcal{J} = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_{n_\mathcal{J}}\}$;
(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: $Y = (A^{-1})_{S_\mathcal{J},S_\mathcal{J}} L_{S_\mathcal{J},\mathcal{J}}$;

Construct an `indmap` array for nonzero rows in the $\mathcal{J}$th supernode;
for $j = 1, 2, ..., n_\mathcal{J}$ **do**
  Identify the supernode $\mathcal{K}$ that contains $\mathcal{I}_j$;
  Let $\mathcal{R}_1 = \texttt{indmap}(\mathcal{I}_j)$;
  Calculate $Y_{\mathcal{R}_1,*} \leftarrow Y_{\mathcal{R}_1,*} + (A^{-1})_{\mathcal{I}_j,\mathcal{I}_j} L_{\mathcal{I}_j,\mathcal{J}}$;
  for $i = j+1, j+2, ...n_\mathcal{J}$ **do**
    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;
    Let $\mathcal{R}_2 = \texttt{indmap}(\mathcal{I}_i)$;
    Calculate $Y_{\mathcal{R}_2,*} \leftarrow Y_{\mathcal{R}_2,*} + (A^{-1})_{\mathcal{I}_i,\mathcal{I}_j} L_{\mathcal{I}_j,\mathcal{J}}$;
    Calculate $Y_{\mathcal{R}_1,*} \leftarrow Y_{\mathcal{R}_1,*} + [(A^{-1})_{\mathcal{I}_i,\mathcal{I}_j}]^T L_{\mathcal{I}_i,\mathcal{J}}$;
  end for
end for
Reset the nonzero entries of `indmap` to zero;

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 $D_{\mathcal{J},\mathcal{J}}$ 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 cluster maintained at NERSC. 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/sec per core (4 flops/cycle if using SSE128 instructions). Each core has 2 GB of memory. Our test problems are taken from Harwell-Boeing Test Collection [8] and the University of Florida Matrix Collection[7]. 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 Table 1 and 2. All matrices are real and symmetric. The multiple minimum degree (MMD) matrix reordering strategy [22] is used to minimize the amount of nonzero fills in L. We used the supernodal left-looking algorithm and code provided by the authors of [26] to perform the LDL^T factorization of A. Table 3 gives the performance result in terms of computational time as well

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 1: Test problems

as floating point operations per second (flops) for both the factorization and the selected inversion algorithms respectively. We also report the average flops.

The dimension of the matrices we tested ranges from problem size ranges from $2,000$ to 1.5 million, and the number of non-zero 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 $350s$. 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 flops ratio is 46%.

To demonstrate how much we can gain by using the selected inversion algorithm instead of the naive approach of inverting A directly through (2), which we will refer to as the *direct inversion*, we list the timing statistics for both approaches in Table 4 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 just refers to the second part of SelInv. The time for factorization part is not counted, since factorization is shared between both algorithms. Even for the smallest problem bcsstk14, the speedup factor is 108. For the largest problem, selected inversion uses $219s$, while the direct inversion requires an estimated 21.8 days. Among these problems, the largest speedup gain is achieved for problem ecology2. In this case the selected inversion is $18,000$ times faster than the direct full inversion.

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

Table 2: Characteristic of the test problems

problem	factorization time(sec)	factorization flops(G/sec)	selected inversion time(sec)	selected inversion flops(G/sec)	average flops(G/sec)
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

Table 3: The time cost, and flops result for factorization and selected inversion process respectively. The last column reports the average flops reached by SelInv.

problem	selected inversion time	direct inversion time	speedup
bcsstk14	0.010 sec	1.080 sec	108
bcsstk24	0.020 sec	5.360 sec	268
bcsstk28	0.024 sec	8.930 sec	372
bcsstk18	0.235 sec	53.37 sec	227
bodyy6	0.090 sec	104.6 sec	1162
crystm03	0.779 sec	577.9 sec	742
wathen120	0.344 sec	655.9 sec	1906
thermal1	0.443 sec	0.539 hr	4308
shipsec1	17.66 sec	8.933 hr	1820
pwtk	14.55 sec	19.56 hr	4839
parabolic_fem	20.06 sec	1.521 day	6551
tmt_sym	13.98 sec	2.486 day	15364
ecology2	16.04 sec	3.374 day	18174
G3_circuit	218.7 sec	21.78 day	8604

Table 4: Time cost comparison between selected inversion and direct inversion. The speedup factor is defined by the time cost of direct inversion divided by the time cost of selected inversion.

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 [15, 16]. The most time consuming part of these calculations is the evaluation of the electron density

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

where and $f_{\beta,\mu}(t) = 1/(1 + e^{\beta(t-\mu)})$ 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 [24] 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 [24] and V_{pse} is the real space Troullier-Martins ionic pseudopotential [5].

The standard approach 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 [1], which is a real space electronic structure calculation software package.

An alternative way to evaluate (7) is to use a recently developed pole expansion technique [18, 20] to approximate $f_{\beta,\mu}$. The pole expansion technique expresses electron density ρ as a linear combination of the diagonal of $(H - (\mu + z_i)I)^{-1}$, i.e.

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

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 i-th 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

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

is satisfied. This can be accomplished by solving (10) using the standard Newton's method.

In order to use (9), we need to compute the diagonal of the inverse of a number of complex symmetric (non-Hermitian) matrices $H-(z_i+\mu)I$ $(i = 1, 2, ..., P)$. 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), i.e. $V_{\mathrm{pse}}(\mathbf{r})$ is a diagonal matrix. The Laplacian operator Δ is discretized using a second-order five-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 5 unit cells along both x and y directions, and 1 unit cell along the z direction in our computational domain. Each unit cell is cubic with a lattice constant of 4.05Å. Therefore, there are altogether 100 aluminum atoms and 300 valence 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 0.21Å. 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 are computed using ARPACK. This calculation takes $2,490$ seconds. 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 $1,950$ seconds. Therefore, the total amount of time required to evaluate (7) on a single core is $1,950 \times 60$ seconds. 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 4.

We also remark that the efficiency of selected inversion can be further improved. One of the factors that have prevented the SelInv from achieving even higher performance is that most of the supernodes produced from the MMD ordering of H contains 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

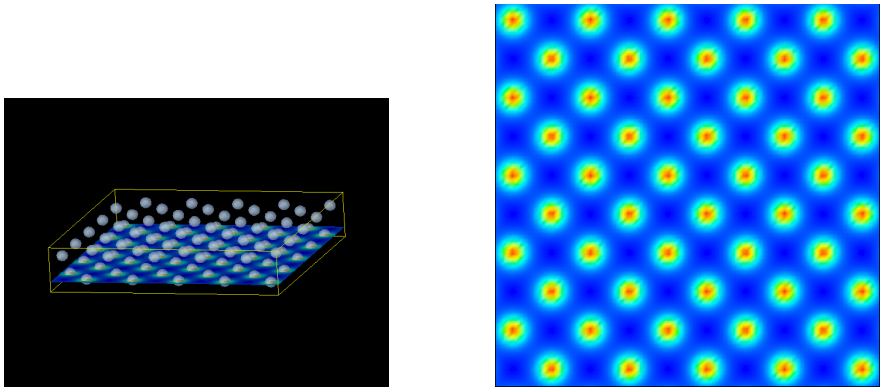


Figure 4: (a)3D isosurface plot of the electron density together with the electron density restricted to $z = 0$ plane. (b) The electron density restricted to $z = 0$ plane.

and treat some of the zeros in L as non-zeros. This approach has been demonstrated to be extremely helpful in [3].

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 [26]. This step can also be implemented using other existing software packages such as [26, 2, 29, 4, 14, 13]. 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 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 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 [18, 20]. 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 expansion is used.

To solve problems with more degrees of freedom, the selected inversion algorithm itself must be parallelized on distributed memory parallel computers. This is a research direction that we plan to pursue in the near future.

Acknowledgements

This work was partially supported by DOE under Contract No. DE-FG02-03ER25587 and by ONR under Contract No. N00014-01-1-0674 (L. L., J. L. and W. E), by an Alfred P. Sloan fellowship and a startup grant from the University of Texas at Austin (L. Y.), and by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231 (C. Y. and J. C. M.). 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. We would like to thank Esmond Ng and Sherry Li for helpful discussion.

References

[1] M. Alemany, M. Jain, L. Kronik, and J. Chelikowsky. Real-space pseudopotential method for computing the electronic properties of periodic systems. *Phys. Rev. B*, 69:075101, 2004.

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

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

[4] C. Ashcraft and R. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Ninth SIAM Conference on Parallel Processing*, San Antonio, TX, 1999.

[5] J. Chelikowsky, N. Troullier, and Y. Saad. Finite-difference-pseudopotential method: Electronic structure calculations without a basis. *Phys. Rev. Lett.*, 72:1240–1243, 1994.

[6] S. Datta. *Electronic transport in mesoscopic systems.* Cambridge Univ Pr, 1997.

[7] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97:7, 1997.

[8] I. Duff, R. Grimes, and J. Lewis. Users guide for the Harwell-Boeing sparse matrix collection. *Research and Technology Division, Boeing Computer Services, Seattle, Washington, USA*, 1992.

[9] I. Duff and J. Reid. *Direct Methods for Sparse Matrices.* Oxford University, London, 1987.

[10] E. Economou. *Green's functions in quantum physics.* Springer Berlin, 2006.

[11] A. Erisman and W. Tinney. On computing certain elements of the inverse of a sparse matrix. *Comm. ACM*, 18:177, 1975.

[12] J. George and J. Liu. *Computer Solution of Large Sparse Positive Definite systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[13] A. Gupta. WSMP: the Watson Sparse Matrix Package. *IBM Research Report*, (RC 21886(98462)), 1997.

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

[15] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864, 1964.

[16] W. Kohn and L. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133, 1965.

[17] S. Li, S. Ahmed, G. Klimeck, and E. Darve. Computing entries of the inverse of a sparse matrix using the FIND algorithm. *J. Comput. Phys.*, 227:9408 – 9427, 2008.

[18] L. Lin, J. Lu, R. Car, and W. E. Multipole representation of the Fermi operator with application to the electronic structure analysis of metallic systems. *Phys. Rev. B*, page 115133, 2009.

[19] L. Lin, J. Lu, L. Ying, R. Car, and W. E. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Comm. Math. Sci.*, accepted.

[20] L. Lin, J. Lu, L. Ying, and W. E. Pole-based approximation of the Fermi-Dirac function. *Chinese Ann. Math Ser. B*, in press.

[21] L. Lin, C. Yang, J. Lu, L. Ying, and W. E. A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2D electronic structure calculations. in preparation, 2009.

[22] J. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

[23] J. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11:134, 1990.

[24] R. Martin. *Electronic Structure – Basic Theory and Practical Methods.* Cambridge University Press, 2004.

[25] C. Moler. *Numerical Computing with MATLAB.* SIAM, 2004.

[26] E. Ng and B. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14:1034, 1993.

[27] D. Petersen, S. Li, K. Stokbro, H. Sørensen, P. Hansen, S. Skelboe, and E. Darve. A hybrid method for the parallel computation of Green's functions. *J. Comput. Phys.*, 228:5020 – 5039, 2009.

[28] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse choleskyfactorization. *SIAM J. Sci. Comput.*, 15:1413–1439, 1994.

[29] O. Schenk and K. Gartner. On fast factorization pivoting methods for symmetric indefinite systems. *Elec. Trans. Numer. Anal.*, 23:158–179, 2006.