Parallel Computing 74 (2018) 84–98

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

PSelInv – A distributed memory parallel algorithm for selected inversion: The non-symmetric case

Mathias Jacquelin[a,*], Lin Lin[b,a], Chao Yang[a]

[a] Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[b] Department of Mathematics, University of California, Berkeley, Berkeley, CA 94720, USA

ARTICLE INFO

Article history:
Received 28 December 2016
Revised 14 October 2017
Accepted 29 November 2017
Available online 2 December 2017

Keywords:
Selected inversion
Parallel algorithm
Non-symmetric
High performance computation

ABSTRACT

This paper generalizes the parallel selected inversion algorithm called PSelInv to sparse non-symmetric matrices. We assume a general sparse matrix A has been decomposed as $PAQ = LU$ on a distributed memory parallel machine, where L, U are lower and upper triangular matrices, and P, Q are permutation matrices, respectively. The PSelInv method computes selected elements of A^{-1}. The selection is confined by the sparsity pattern of the matrix A^T. Our algorithm does not assume any symmetry properties of A, and our parallel implementation is memory efficient, in the sense that the computed elements of A^{-T} overwrites the sparse matrix $L+U$ *in situ*. PSelInv involves a large number of collective data communication activities within different processor groups of various sizes. In order to minimize idle time and improve load balancing, tree-based asynchronous communication is used to coordinate all such collective communication. Numerical results demonstrate that PSelInv can scale efficiently to 6,400 cores for a variety of matrices.

1. Introduction

Let $A \in \mathbb{C}^{N\times N}$ be a sparse matrix. If A is symmetric, the selected inversion algorithm [1–4] and its variants [5–13] are efficient ways for computing certain selected elements of A^{-1}, defined as $(A^{-1})_S := \{(A^{-1})_{i,j} |\ \text{for}\ 1 \le i, j \le N,\ \text{such that}\ A_{i,j} \neq 0\}$. The algorithm actually computes more elements of A^{-1} than $(A^{-1})_S$. The set of computed elements is a superset of $(A^{-1})_S$, defined as $\{(A^{-1})_{i,j} |\ (L+U)_{i,j} \neq 0\}$. Here, for simplicity, we have omitted the range of indices for i, j. The LU factorization of A is given by $A = LU$, and the sparsity pattern of U is the same as that of L^T. Selected inversion algorithms have already been used in a number of applications such as density functional theory [12,14,15], quantum transport theory [6–8,13], dynamical mean field theory (DMFT) [16], Poisson–Boltzmann equation [17], to name a few.

In [2], Erisman and Tinney demonstrated that a selected inversion procedure can be applied to non-symmetric matrices. In such a case, the selected inversion algorithm computes $\{(A^{-1})_{i,j} |\ (L+U)_{j,i} \neq 0\}$, and therefore the definition of selected elements should be modified to

$$(A^{-1})_S := \{(A^{-1})_{i,j} | A_{j,i} \neq 0\}. \tag{1}$$

Let us consider two extreme cases. 1) When A is symmetric, the general definition of selected elements agree with the previous definition. The same argument holds for structurally symmetric matrices (i.e. $A_{i,j} \neq 0 \Leftrightarrow A_{j,i} \neq 0$). 2) When A is an

* Corresponding author.
E-mail addresses: mjacquelin@lbl.gov (M. Jacquelin), linlin@math.berkeley.edu (L. Lin), cyang@lbl.gov (C. Yang).

https://doi.org/10.1016/j.parco.2017.11.009
0167-8191/

upper triangular or a lower triangular matrix, the selected inversion algorithm only computes the diagonal elements of A^{-1}. Indeed, these entries are easy to compute since $(A^{-1})_{i,i} = (A_{i,i})^{-1}$, while $\{(A^{-1})_{i,j} | A_{i,j} \neq 0\}$ would include all the nonzero entries of A^{-1}.

At first glance, it may seem restrictive that the selected inversion algorithm for general matrices cannot even compute the entries of A^{-1} corresponding to the sparsity pattern of A. Fortunately, this modified definition of selected elements is already sufficient in a number of applications. One case is the computation of the diagonal elements of A^{-1}. Another case is the computation of traces of the form $\mathrm{Tr}[BA^{-1}] = \sum_{ij} B_{i,j}(A^{-1})_{j,i}$, where the sparsity pattern of $B \in \mathbb{C}^{N\times N}$ is contained in the sparsity pattern of A, i.e. $\{(i,j)|B_{i,j} \neq 0\} \subset \{(i,j)|A_{i,j} \neq 0\}$. This type of trace calculation appears in a number of contexts, such as the computation of electron energy in density functional theory calculations. It is also a useful way to numerically validate the identity $\mathrm{Tr}[AA^{-1}] = N$, which serves as a quick and useful indicator of the accuracy of the computed selected elements of A^{-1}, especially for large matrices of which the full inverse is too expensive to compute.

Although the non-symmetric version of the selected inversion algorithm was proposed more than four decades ago, to our knowledge, there is no efficient implementation of the selected inversion algorithm for general non-symmetric matrices, either sequential or parallel. This paper fills this gap by extending the PSelInv implementation reported in [4] to non-symmetric matrices and on distributed memory parallel architecture. We remark that such a general treatment may be of interest even for symmetric matrices, when additional static pivoting is performed to improve numerical stability [18,19]. In such cases, the selected inversion algorithm needs to be applied to the non-symmetric matrix $\widetilde{A} = PAQ$, where P, Q are permutation matrices.

There are some notable differences between the implementation of PSelInvfor symmetric and non-symmetric matrices. First, a non-symmetric matrix only permits a LU factorization, while the LU, Cholesky (LL^T), and the LDL^T factorization can all be used for symmetric matrices. Second, for non-symmetric matrices, one can in principle perform a structural symmetrization procedure by treating certain zero elements as nonzeros and use the selected inversion algorithm for structurally symmetric matrices. However, such treatment is generally inefficient in terms of both the storage cost and the computational cost. As an extreme case, structurally symmetrizing a dense upper triangular matrix would mean that the matrix A is treated as a full dense square matrix, while the selected inversion algorithm for the upper triangular matrix according to the definition of Eq. (1) means that only the diagonal entries need to be inverted. From this perspective, our parallel implementation is memory efficient, in the sense that no symmetrization process is involved, and the selected elements of A^{-T} overwrites the sparse matrix $L+U$ *in situ*. Here the transpose corresponds to the definition of the selected elements (1) and will be explained in detail later. Third, more complicated data communication pattern is required to implement the parallel selected inversion algorithm for non-symmetric matrices, and the selected elements of the inverse in the upper and lower triangular parts need to be treated separately. In [4] we explicitly take advantage of the symmetry of the matrix to simplify some of the data communication. This is no longer an option for non-symmetric matrices. We develop a general point-to-point data communication strategy to efficiently handle collective data communication operations. This general point-to-point data communication strategy allows us to use a recently developed tree based asynchronous collective communication method to improve load balancing when a large number of cores are used, as recently demonstrated for the symmetric case of the PSelInvalgorithm [20]. Our numerical results indicate that the non-symmetric version of PSelInvcan be scalable to up to 6400 cores depending on the size and sparsity of the matrix. Our implementation of PSelInvis publicly available.[1]

The rest of the paper is organized as follows. We review the basic idea of the selected inversion method for non-symmetric matrices in Section 2, and discuss various implementation issues for the distributed memory parallel selected inversion algorithm for non-symmetric matrices in Section 3. The numerical results with applications to various matrices from including Harwell–Boeing Test Collection [21], the SuiteSparse Matrix Collection[22], and from density functional theory in Section 4, followed by the conclusion and the future work discussion in Section 5.

Standard linear algebra notation is used for vectors and matrices throughout the paper. We use $A_{i,j}$ to denote the (i,j)th entry of the matrix A, and f_i to denote the ith entry of the vector f. With slight abuse of notation, both a supernodal index and the set of column indices associated with a supernode are denoted by uppercase script letters such as $\mathcal{I}, \mathcal{J}, \mathcal{K}$ *etc.*. $A^{-1}_{\mathcal{I},\mathcal{J}}$ denotes the $(\mathcal{I},\mathcal{J})$th block of the matrix A^{-1}, i.e. $A^{-1}_{\mathcal{I},\mathcal{J}} \equiv (A^{-1})_{\mathcal{I},\mathcal{J}}$. When the block $A_{\mathcal{I},\mathcal{J}}$ itself is invertible, its inverse is denoted by $(A_{\mathcal{I},\mathcal{J}})^{-1}$ to distinguish from $A^{-1}_{\mathcal{I},\mathcal{J}}$. We also use $(A^{-T})_{\mathcal{I},\mathcal{J}}$ to denote the $(\mathcal{I},\mathcal{J})$th matrix block of the transpose of the matrix A^{-1}.

2. Selected inversion algorithm for non-symmetric matrices

The standard approach for computing A^{-1} is to first decompose A using the LU factorization

$$A = LU \tag{2}$$

where L is a unit lower triangular matrix and U is an upper triangular matrix. In order to stabilize the computation, matrix reordering and row pivoting (or partial pivoting) [18] are usually applied to the matrix of A, and the general form of the LU factorization can be given as

$$PAQ = \widetilde{A} = LU, \tag{3}$$

[1] http://www.pexsi.org/, distributed under the BSD license

where P and Q are two permutation matrices. Care must be taken when non-symmetric row and column permutations are used, i.e. $P \neq Q^T$. To simplify the discussion for now, we use Eq. (2) and assume A has already been permuted.

The selected inversion algorithm can be *heuristically* understood as follows. We first partition the matrix A into 2×2 blocks of the form

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, \tag{4}$$

where $A_{1,1}$ is a scalar of size 1×1. We can write $A_{1,1}$ as a product of two scalars $L_{1,1}$ and $U_{1,1}$. In particular, we can pick $L_{1,1} = 1$ and $U_{1,1} = A_{1,1}$. Then

$$A = \begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & I \end{pmatrix} \begin{pmatrix} U_{1,1} & U_{1,2} \\ 0 & S_{2,2} \end{pmatrix} \tag{5}$$

where

$$L_{2,1} = A_{2,1}(U_{1,1})^{-1}, \quad U_{1,2} = (L_{1,1})^{-1} A_{1,2}, \tag{6}$$

and

$$S_{2,2} = A_{2,2} - L_{2,1} U_{1,2} \tag{7}$$

is the Schur complement.

Using the decomposition given by Eq. (5), we can express A^{-1} as

$$A^{-1} = \begin{pmatrix} (U_{1,1})^{-1}(L_{1,1})^{-1} + (U_{1,1})^{-1} U_{1,2} S_{2,2}^{-1} L_{2,1} (L_{1,1})^{-1} & -(U_{1,1})^{-1} U_{1,2} S_{2,2}^{-1} \\ -S_{2,2}^{-1} L_{2,1} (L_{1,1})^{-1} & S_{2,2}^{-1} \end{pmatrix}. \tag{8}$$

With slight abuse of notation, define $\mathcal{C}_L := \{i | L_{i,1} \neq 0\}$ and $\mathcal{C}_U := \{j | U_{1,j} \neq 0\}$. Here $L_{i,1}$ is the ith component of the column vector $(L_{1,1}, L_{2,1})^T$ as in Eq. (5), and $U_{1,j}$ is the jth component of the row vector $(U_{1,1}, U_{1,2})$. The sets $\mathcal{C}_L$ and $\mathcal{C}_U$ are defined purely in terms of the nonzero structures of L and U, i.e., $L_{i,1}$ and $U_{1,j}$ treated as nonzeros even if their numerical values are coincidentally 0. For non-symmetric matrices, $\mathcal{C}_L$ and $\mathcal{C}_U$ may not be the same.

We assume $S_{2,2}^{-1}$ has already been computed. From Eq. (8) it can be readily observed that, if L and U are sparse, the (1, 1) entry of A^{-1} can be computed from the nonzero elements of $L_{2,1}$ and $U_{1,2}$ together with the corresponding selected entries of $S_{2,2}^{-1}$. Because $A_{2,2}^{-1} = S_{2,2}^{-1}$, the selected entries of $S_{2,2}^{-1}$ belong to a subset of

$$\left\{ A_{i,j}^{-1} | i \in \mathcal{C}_U, j \in \mathcal{C}_L \right\}. \tag{9}$$

which also include $\{A_{1,j}^{-1} | j \in \mathcal{C}_L\}$ and $\{A_{i,1}^{-1} | i \in \mathcal{C}_U\}$. The latter can be computed from the same selected elements of $S_{2,2}^{-1}$, $L_{2,1}$ and $U_{1,2}$. Repeating the procedure above recursively for $S_{2,2}$, we can see how the selected elements of $A_{i,k}^{-1}$ and $A_{k,j}^{-1}$ that are required to compute the selected elements of A^{-1} in the rows and columns preceding k can be computed from selected elements of the trailing $(n-k) \times (n-k)$ block of A^{-1}. This argument can be stated more precisely in Theorem 1.

Theorem 1 (Erisman and Tinney [2]). *For a matrix $A \in \mathbb{C}^{N \times N}$, let $A = LU$ be its LU factorization, and L, U are invertible matrices. For any $1 \leq k < N$, define*

$$\mathcal{C}_L = \{i | L_{i,k} \neq 0\}, \quad \mathcal{C}_U = \{j | U_{k,j} \neq 0\}. \tag{10}$$

Then all entries $\{A_{i,k}^{-1} | i \in \mathcal{C}_U\}$, $\{A_{k,j}^{-1} | j \in \mathcal{C}_L\}$, and $A_{k,k}^{-1}$ can be computed using only $\{L_{j,k} | j \in \mathcal{C}_L\}$, $\{U_{k,i} | i \in \mathcal{C}_U\}$ and $\{A_{i,j}^{-1} | (L+U)_{j,i} \neq 0, i, j \geq k\}$.

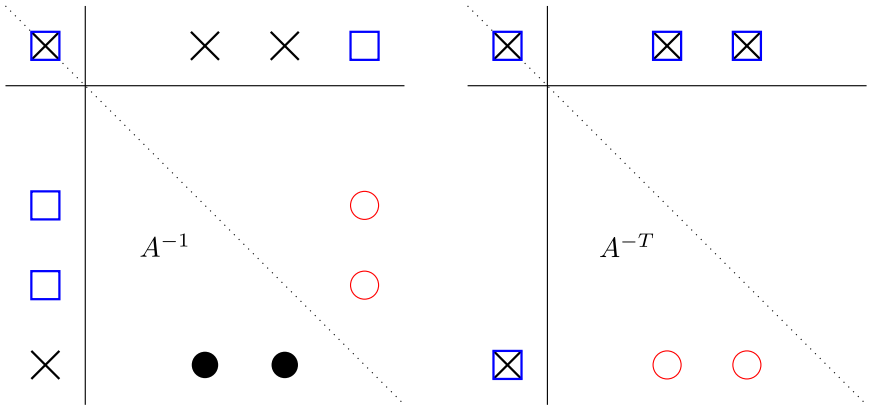
Proof. First consider $\{A_{i,k}^{-1} | i \in \mathcal{C}_U\}$. Similar to Eq. (8) we can derive

$$A_{i,k}^{-1} = -\sum_{j=k+1}^{N} A_{i,j}^{-1} L_{j,k} (L_{k,k})^{-1}, \quad i \in \mathcal{C}_U. \tag{11}$$

If $L_{j,k} \neq 0$, then $A_{i,j}^{-1}$ is needed in the sum. Since we are only interested in computing $A_{i,k}^{-1}$ for $i \in \mathcal{C}_U$, the i and j indices are constrained to satisfy the conditions $L_{j,k} \neq 0$ and $U_{k,i} \neq 0$. This constraint implies $(L+U)_{j,i} \neq 0$ because the nonzero fill-in pattern of the trailing blocks of L and U are determined by the nonzero patterns of the kth column of L and the kth row of U respectively. A similar argument can be made for $\{A_{k,j}^{-1} | j \in \mathcal{C}_L\}$. Finally for the diagonal entry, we have

$$A_{k,k}^{-1} = (U_{k,k})^{-1}(L_{k,k})^{-1} - \sum_{i=k+1}^{N} (U_{k,k})^{-1} U_{k,i} A_{i,k}^{-1}, \tag{12}$$

which can be readily computed given $\{A_{i,k}^{-1} | i \in \mathcal{C}_U\}$ is available. □



(a) $A^{-1}_{k,k}, \{A^{-1}_{i,k} | i \in \mathcal{C}_U\}$ and $\{A^{-1}_{k,j} | j \in \mathcal{C}_L\}$ for a particular k are marked by ☐.

(b) $\{L_{i,k} | i \in \mathcal{C}_L\}$ and $\{U_{k,j} | j \in \mathcal{C}_U\}$ are overwritten by the corresponding entries in A^{-T}.

Fig. 1. (a) $\{L_{i,k} | i \in \mathcal{C}_L\}$ and $\{U_{k,j} | j \in \mathcal{C}_U\}$ are marked by ✕. The nonzero fills introduced by the kth column of L and kth row of U are represented by ●. (b) A^{-T} in the selected inversion algorithm can directly overwrite the L, U factors.

Fig. 1(a) illustrates one step of the selected inversion procedure for a general matrix. For example, according to Eqs. (11) and (12), computing the (k, k)th element of A^{-1} shown at the upper left corner of the figure requires previously computed element of A^{-1} marked by red circles. To compute $A^{-1}_{k,k}$ we effectively have to compute the selected element of A^{-1} marked by the blue squares. By pretending that we are computing the selected elements of A^{-T} instead, we can overwrite the corresponding elements of U and L as shown in Fig. 1 (b). Theorem 1 directly indicates that any element of A^{-1} corresponding to the sparsity pattern of $(L+U)^T$ can be evaluated using L, U and other elements of A in this subset of entries. In particular, the selected elements $\{A^{-1}_{i,j} | A_{j,i} \neq 0\}$ can be evaluated efficiently.

So far we have not explicitly taken into account row and column permutation. Theorem 2 demonstrates that the same result holds when permutation is involved.

Theorem 2. *For $A \in \mathbb{C}^{N \times N}$, let $PAQ = \widetilde{A} = LU$ be its LU factorization. Here L, U are invertible matrices, and P, Q are permutation matrices. Then $\{A^{-1}_{i,j} | A_{j,i} \neq 0\}$ can be evaluated using L, U and $\{\widetilde{A}^{-1}_{i,j} | (L+U)_{j,i} \neq 0\}$.*

Proof. Since P, Q are permutation matrices, $PP^T = QQ^T = I$, and we have the identity

$$A^T = Q\widetilde{A}^T P, \quad A^{-1} = Q\widetilde{A}^{-1} P. \tag{13}$$

Since the entries $\{\widetilde{A}^{-1}_{i,j} | \widetilde{A}_{j,i} \neq 0\} \subset \{\widetilde{A}^{-1}_{i,j} | (L+U)_{j,i} \neq 0\}$ have been computed, undo the permutation of $\widetilde{A}^{-1}$ and we obtain $\{A^{-1}_{i,j} | A_{j,i} \neq 0\}$, which are the required selected elements of A^{-1}. □

In practice, a column-based sparse factorization and selected inversion algorithm may not be efficient due to the lack of level 3 BLAS operations. For a sparse matrix A, the columns of A and the L factor can be partitioned into supernodes. A supernode is a maximal set of contiguous columns $\mathcal{J} = \{j, j+1, \ldots, j+s\}$ of the L factor that have the same nonzero structure below the $(j+s)$th row, and the lower triangular part of $L_{\mathcal{J},\mathcal{J}}$ is dense. However, this strict definition can produce supernodes that are either too large or too small, leading to memory usage, load balancing and efficiency issues. Therefore, in our work, we relax this definition to limit the maximal number of columns in a supernode (i.e. sets are not necessarily maximal). The relaxation also allows a supernode to include columns for which nonzero patterns are nearly identical to enhance the efficiency [23], and this approach is also used in `SuperLU_DIST` [19]. We assume the same supernode partitioning is usually applied to the row partition as well, even though the nonzero pattern of the L and U can be different from each other. The total number of supernodes is denoted by $\mathcal{N}$. Using the notation of supernodes (e.g. 1 means the first supernode instead of the first column index), $L_{1,1}$ is no longer a scalar 1 or an identity matrix, but a lower triangular matrix. To simplify the notation of the selected inversion algorithm, in Eq. (8) we can define the normalized LU factors as

$$\hat{L}_{1,1} = L_{1,1}, \quad \hat{U}_{1,1} = U_{1,1}, \quad \hat{L}_{2,1} = L_{2,1}(L_{1,1})^{-1}, \quad \hat{U}_{1,2} = (U_{1,1})^{-1} U_{1,2}. \tag{14}$$

This definition can be directly generalized for other columns and for the case when supernodes are used. Furthermore, from an implementation perspective, the definition of selected elements indicates that it is most natural to formulate the selected inversion algorithm to compute A^{-T}, so that A^{-T} can directly overwrite the L, U factors (see Fig. 1(b)). A pseudo-code for the selected inversion algorithm for non-symmetric matrices is given in Algorithm 1, which can readily be used as a sequential implementation of the selected inversion algorithm. Note that in step 3, the diagonal entry can be equivalently

Algorithm 1. Selected inversion algorithm for a general sparse matrix A.

Input: (1) Permutation matrices P, Q.
(2) The supernodal partition $\{1, 2, ..., \mathcal{N}\}$.
(3) A supernodal sparse LU factorization $PAQ = \widetilde{A} = LU$.

Output: $\{A^{-1}_{\mathcal{I},\mathcal{J}} | A_{\mathcal{J},\mathcal{I}}$ is a nonzero block, $\mathcal{I}, \mathcal{J} = 1, \ldots, \mathcal{N}\}$.

for $\mathcal{K} = \mathcal{N}, \mathcal{N}-1, ..., 1$ **do**

Find the collection of indices

$\mathcal{C}_L = \{\mathcal{I} | \mathcal{I} > \mathcal{K}, L_{\mathcal{I},\mathcal{K}}$ is a nonzero block$\}$

$\mathcal{C}_U = \{\mathcal{J} | \mathcal{J} > \mathcal{K}, U_{\mathcal{K},\mathcal{J}}$ is a nonzero block$\}$

1 $\hat{L}_{\mathcal{C}_L,\mathcal{K}} \leftarrow L_{\mathcal{C}_L,\mathcal{K}}(L_{\mathcal{K},\mathcal{K}})^{-1}, \hat{U}_{\mathcal{K},\mathcal{C}_U} \leftarrow (U_{\mathcal{K},\mathcal{K}})^{-1} U_{\mathcal{K},\mathcal{C}_U}$

end

for $\mathcal{K} = \mathcal{N}, \mathcal{N}-1, ..., 1$ **do**

Find the collection of indices

$\mathcal{C}_L = \{\mathcal{I} | \mathcal{I} > \mathcal{K}, L_{\mathcal{I},\mathcal{K}}$ is a nonzero block$\}$

$\mathcal{C}_U = \{\mathcal{J} | \mathcal{J} > \mathcal{K}, U_{\mathcal{K},\mathcal{J}}$ is a nonzero block$\}$

2 Calculate $(\widetilde{A}^{-T})_{\mathcal{K},\mathcal{C}_U} \leftarrow -(\hat{L}_{\mathcal{C}_L,\mathcal{K}})^T (\widetilde{A}^{-T})_{\mathcal{C}_L,\mathcal{C}_U}$

3 Calculate $(\widetilde{A}^{-T})_{\mathcal{K},\mathcal{K}} \leftarrow (L_{\mathcal{K},\mathcal{K}})^{-T}(U_{\mathcal{K},\mathcal{K}})^{-T} - (\widetilde{A}^{-T})_{\mathcal{K},\mathcal{C}_U}(\hat{U}_{\mathcal{K},\mathcal{C}_U})^T$

4 Calculate $(\widetilde{A}^{-T})_{\mathcal{C}_L,\mathcal{K}} \leftarrow -(\widetilde{A}^{-T})_{\mathcal{C}_L,\mathcal{C}_U}(\hat{U}_{\mathcal{K},\mathcal{C}_U})^T$

end

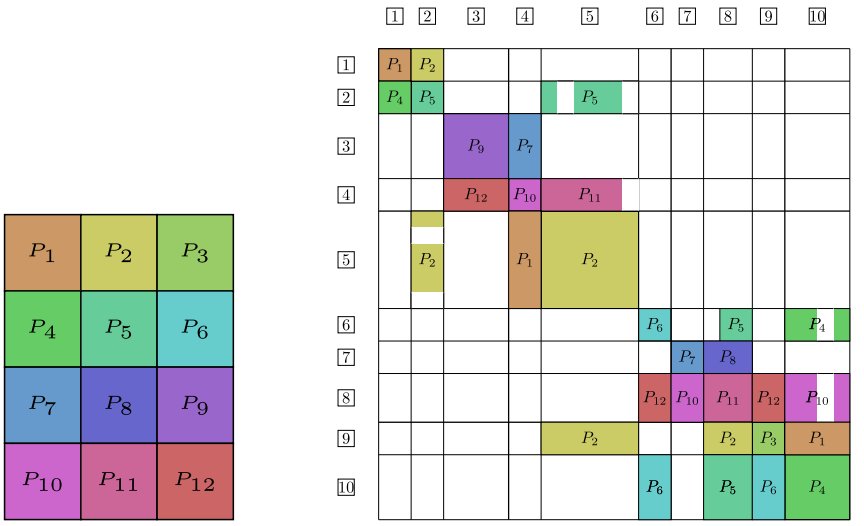
5 Extract the matrix blocks $\{(\widetilde{A}^{-T})_{\mathcal{J},\mathcal{I}} | \widetilde{A}_{\mathcal{J},\mathcal{I}}$ is a nonzero block$\}$, undo the permutation and apply matrix transpose to obtain $\{A^{-1}_{\mathcal{I},\mathcal{J}} | A_{\mathcal{J},\mathcal{I}}$ is a nonzero block, $\mathcal{I}, \mathcal{J} = 1, \ldots, \mathcal{N}\}$

computed using the formula $(\widetilde{A}^{-T})_{\mathcal{K},\mathcal{K}} \leftarrow (L_{\mathcal{K},\mathcal{K}})^{-T}(U_{\mathcal{K},\mathcal{K}})^{-T} - (\hat{L}_{\mathcal{C}_L,\mathcal{K}})^T (\widetilde{A}^{-T})_{\mathcal{C}_L,\mathcal{K}}$. We also note that the normalized factors $\hat{L}, \hat{U}$ can overwrite the L, U factors, and the intermediate matrix $\widetilde{A}^{-T}$ can overwrite the normalized factors whenever the computation for a given supernode $\mathcal{K}$ is finished. However, we keep these matrices with distinct notations in Algorithm 1 for clarity.

3. Distributed memory parallel selected inversion algorithm for non-symmetric matrices

In this section, we present the PSelInv method for general non-symmetric matrices on distributed memory parallel architecture. The selected inversion algorithm described in Algorithm 1 requires a sparse LU factorization of the permuted matrix $\widetilde{A} = PAQ$ to be computed first. We compute the LU decomposition using the SuperLU_DIST software package [19], which has been shown to be scalable to a large number of processors on distributed memory parallel machines. SuperLU_DIST allows the sparse L and U factors to be accessed through relatively simple data structures. However, it should be noted that the ideas developed in this section can be combined with other sparse matrix solvers such as MUMPS [24] or PARDISO [25] too, provided that the factors are available.

As discussed at the end of Section 2, in order to achieve a memory efficient implementation, we work with the transposed matrix inverse $\widetilde{A}^{-T}$, which can directly overwrite the LU factors. To simplify the notation, in this section we do not distinguish A and the permuted matrix $\widetilde{A}$. We use the same 2D block cyclic distribution scheme employed in SuperLU_DIST to partition and distribute both the L, U factors and the selected elements of A^{-T} to be computed. We will review the main features of this type of distribution in Section 3.1. In the 2D block cyclic distribution scheme, each supernode $\mathcal{K}$ is assigned to and partitioned among a subset of processors. However, computing the selected elements of A^{-T} associated with the supernode $\mathcal{K}$ requires retrieving some previously computed selected elements of A^{-T} that belong to ancestors of $\mathcal{K}$ in the elimination tree. These selected elements may reside on other processors. As a result, communication is required to transfer data among different processors to complete steps 2 to 4 of Algorithm 1 in each iteration. We will discuss how this is done in Section 3.2. Furthermore, in order to achieve scalable performance on thousands of cores, it is important to overlap communication with computation using asynchronous point-to-point MPI functions. In the PSelInv method, most of these communication operations are collective in nature (e.g., broadcast and reduce) within communication subgroups. The sizes of the communication groups can vary widely for operations associated with different supernodes. We will describe how such collective communication operations can be efficiently performed asynchronously in Section 3.3.



(a) A 4-by-3 example of a 2D processor grid

(b) 2D block cyclic distribution of PMatrix data structure on a 4-by-3 processor grid

Fig. 2. Data layout of the non-symmetric PMatrixdata structure used by `PSelInv`.

3.1. Distributed data layout and structure

As discussed in Section 2, the columns of *A, L* and *U* are partitioned into supernodes. Different supernodes may have different sizes. The same partition is applied to the rows of these matrices to create a 2D block partition of these matrices. The submatrix blocks are mapped to processors that are arranged in a virtual 2D grid of dimension $\mathrm{Pr} \times \mathrm{Pc}$ in a cyclic fashion as follows: The $(\mathcal{I}, \mathcal{J})$th matrix block is held by the processor labeled by

$$P_{\mathrm{mod}(\mathcal{I}-1,\mathrm{Pr})\times \mathrm{Pc}+\mathrm{mod}(\mathcal{J}-1,\mathrm{Pc})+1}. \tag{15}$$

This is called a 2D block cyclic data-to-processor mapping. The mapping itself does not take the sparsity of the matrix into account. If the $(\mathcal{I}, \mathcal{J})$th block contains only zero elements, then that block is not stored. It is possible that some nonzero blocks may contain several rows of zeros. These rows are not stored either. As an example, a 4-by-3 grid of processors is depicted in Fig. 2(a). The mapping between the 2D supernode partition of a sparse matrix and the 2D processor grid in Fig. 2(a) is depicted in Fig. 2(b). Each supernodal block column of *L*is distributed among processors that belong to a column of the processor grid. Each processor may own multiple matrix blocks. For instance, the nonzero rows in the second supernode are owned by processors P_2 and P_5. More precisely, P_2 owns two nonzero blocks, while P_5 is responsible for one block. Note that these nonzero blocks are not necessarily contiguous in the global matrix. Though the nonzero structure of *A*is not taken into account during the distribution, it has been shown in practice that 2D layouts leads to higher scalability for both dense [26] and sparse Cholesky factorization [27].

In the current implementation, `PSelInv` contains an interface that is compatible with the `SuperLU_DIST` software package. In order to allow `PSelInv` to be easily integrated with other *LU* factorization codes, we create some intermediate sparse matrix objects to hold the distributed *L* and *U* factors. Such intermediate sparse matrix objects will be overwritten by matrix blocks of A^{-T} in the selected inversion process. Each nonzero block $L(\mathcal{I}, \mathcal{J})$ is stored as follows. Diagonal blocks are always stored as dense matrices which includes both $L(\mathcal{I}, \mathcal{I})$ and $U(\mathcal{I}, \mathcal{I})$. Nonzero entries of $L(\mathcal{I}, \mathcal{J})$ $(\mathcal{I} > \mathcal{J})$ are stored contiguously as a dense matrix in a column-major order even though row indices associated with the stored matrix elements are not required to be contiguous. Nonzero entries of within $U(\mathcal{I}, \mathcal{J})$ $(\mathcal{I} < \mathcal{J})$ are also stored as a dense matrix in a contiguous array in a column major order. The nonzero column indices associated with the nonzeros entries in $U(\mathcal{I}, \mathcal{J})$ are not required to be continuous either. We remark that for matrices with highly non-symmetric sparsity patterns, it is more efficient to store the upper triangular blocks using the skyline structure shown in [19]. However, we choose to use a simpler data layout because it allows level-3 BLAS (GEMM) to be used in the selected inversion process.

3.2. Computing selected elements of A^{-T} within each supernode in parallel

In this section, we detail how steps 2 to 4 in Algorithm 1 can be completed in parallel. We perform step 1 of Algorithm 1 in a separate pass, since the data communication required in this step is relatively simple. The processor that owns the block $L_{\mathcal{K},\mathcal{K}}$ broadcasts $L_{\mathcal{K},\mathcal{K}}$ to all other processors within the same column processor group owning nonzero blocks in the supernode $\mathcal{K}$. Each processor in that group performs the triangular solve $\hat{L}_{\mathcal{I},\mathcal{K}} = L_{\mathcal{I},\mathcal{K}}(L_{\mathcal{K},\mathcal{K}})^{-1}$ for each nonzero block contained in the set $\mathcal{C}$ defined in step 1 of the algorithm. Because $L_{\mathcal{I},\mathcal{K}}$ is not used in the subsequent steps of selected inver-

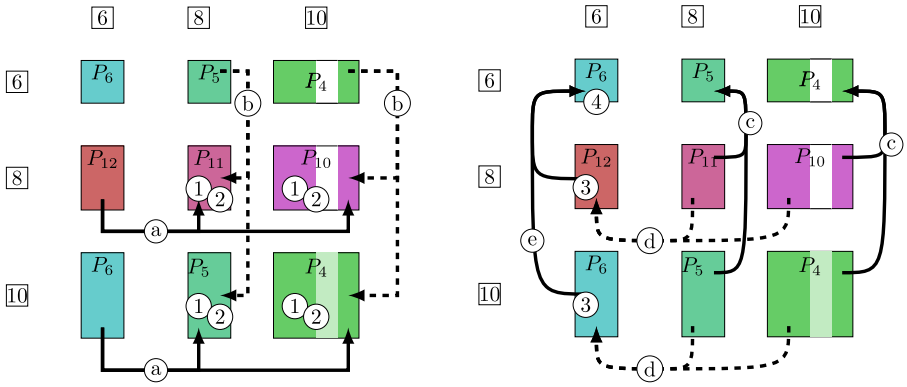


Fig. 3. Communication and computational events for computing selected elements of A^{-1} within $\boxed{6}$. The ⓐ-①-ⓒ sequence of events yields $\{A^{-1}_{i,6} | i \in \mathcal{C}_U\}$ and overwrites the corresponding elements in $\hat{U}_{6,i}$. The ⓑ-②-ⓓ sequence yields $\{A^{-1}_{6,j} | j \in \mathcal{C}_L\}$. Before overwriting the corresponding elements in $\hat{L}_{j,6}$, the ③-ⓔ-④ sequence yields $A^{-1}_{6,6}$. Note that the shaded area of $(A^{-T})_{10,10}$ does not contribute to supernode $\boxed{6}$.

sion once $\hat{L}_{\mathcal{I},\mathcal{K}}$ has been computed, it is overwritten by $\hat{L}_{\mathcal{I},\mathcal{K}}$. Similarly, $U_{\mathcal{K},\mathcal{K}}$ is broadcast to all other processors within the same row processor group owning nonzero blocks in the supernode $\mathcal{K}$. Each processor in that group performs the triangular solve $\hat{U}_{\mathcal{K},\mathcal{I}} = (U_{\mathcal{K},\mathcal{K}})^{-1} U_{\mathcal{K},\mathcal{I}}$ for each nonzero block contained in the set $\mathcal{C}$ defined in step 1 of the algorithm.

A more complicated communication pattern is required to complete steps 2 to 4 in parallel. Because $(A^{-T})_{\mathcal{C}_L,\mathcal{C}_U}$ and $\hat{L}_{\mathcal{C}_L,\mathcal{K}}$ (resp. $\hat{U}_{\mathcal{K},\mathcal{C}_U}$) are generally owned by different processor groups, using the approach discussed in [4], we need to send blocks of $\hat{L}_{\mathcal{C}_L,\mathcal{K}}$ to processors that own *matching* blocks of $(A^{-T})_{\mathcal{C}_L,\mathcal{C}_U}$, so that matrix-matrix multiplication can be performed on the group of processors owning $(A^{-T})_{\mathcal{C}_L,\mathcal{C}_U}$. More specifically, the processor owning the $\hat{L}_{\mathcal{I},\mathcal{K}}$ block sends to all processors within the same row group of processors among which $(A^{-T})_{\mathcal{I},\mathcal{C}_U}$ is distributed in step 2.

However, the set of processors owning $\hat{L}_{\mathcal{I},\mathcal{K}}$ and the owners of $(A^{-T})_{\mathcal{I},\mathcal{C}_U}$ generally form a small subset of all processors, and this set can largely vary across different supernodes. In order to perform such collective communication operations efficiently within the MPI framework, one would have to create a communicator per distinct communication pattern. We have shown in [20] that in the context of `PSelInv`, this can result in more communicators than what was handled by most MPI implementations for matrices of large sizes. Therefore, one way to complete this step of data communication is to use a number of point-to-point asynchronous MPI sends from the processor that owns $\hat{L}_{\mathcal{I},\mathcal{K}}$ to the group of processors that own the nonzero blocks of $(A^{-T})_{\mathcal{I},\mathcal{C}_U}$. Similarly, in step 4 the processor that owns $\hat{U}_{\mathcal{K},\mathcal{J}}$ has to send it to the group of processors that own the nonzero blocks of $(A^{-T})_{\mathcal{C}_L,\mathcal{J}}$. Then $\hat{L}^T_{\mathcal{I},\mathcal{K}}(A^{-T})_{\mathcal{I},\mathcal{J}}$ and $(A^{-T})_{\mathcal{I},\mathcal{J}}\hat{U}^T_{\mathcal{K},\mathcal{J}}$ are performed locally on each processor owning $(A^{-T})_{\mathcal{I},\mathcal{J}}$ using the GEMM subroutine in BLAS3, and the local matrix contributions $\hat{L}^T_{\mathcal{I},\mathcal{K}}(A^{-T})_{\mathcal{I},\mathcal{J}}$ are reduced within each column communication groups owning $\hat{U}_{\mathcal{K},\mathcal{J}}$ to produce the $(A^{-T})_{\mathcal{K},\mathcal{J}}$ block in step 2 of Algorithm 1. Respectively, local matrix contributions $(A^{-T})_{\mathcal{I},\mathcal{J}}\hat{U}^T_{\mathcal{K},\mathcal{J}}$ are reduced within each row communication groups owning $\hat{L}_{\mathcal{I},\mathcal{K}}$ to produce the $(A^{-T})_{\mathcal{I},\mathcal{K}}$ block in step 4 of Algorithm 1. We will discuss in more detail how these asynchronous point-to-point exchanges can be organized to form efficient broadcast and reduction operations in Section 3.3.

Fig. 3 illustrates how this step is completed for a specific supernode $\mathcal{K} = \boxed{6}$, for the matrix depicted in Fig. 2(b). We use circled letters ⓐ, ⓑ, ⓒ, ⓓ, ⓔ to label communication events, and circled numbers ①, ②, ③, ④ to label computational events. We can see from this figure that $\hat{L}_{8,6}$ is sent by P_{12} to all processors within the same row processor group to which P_{12} belongs (ⓐ). This group includes P_{10}, P_{11}, and P_{12}. Similarly $\hat{L}_{10,6}$ is broadcast from P_6 to all other processors within the same row group to which P_6 belongs (ⓐ). For the upper triangular part, $\hat{U}_{6,8}$ is sent by P_5 along the column processor group to which it belongs (ⓑ). P_4 does a similar communication operation for $\hat{U}_{6,10}$.

Local matrix-matrix multiplications are then performed on P_{11}, P_{10}, P_4 and P_5 simultaneously, corresponding to events ① and ②. Contributions to $(A^{-T})_{\mathcal{C}_L,\mathcal{K}}$ are then reduced onto P_{12} and P_6 within the row processor groups they belong to respectively (communication step ⓓ). Similarly, communication step ⓒ corresponds to reductions of contributions to $(A^{-T})_{\mathcal{K},\mathcal{C}_U}$ onto P_5 and P_4. After this step, $(A^{-T})_{8,6}$ and $(A^{-T})_{10,6}$ become available on P_{12} and P_6 respectively. The matrix product $\hat{L}^T_{\mathcal{C}_L,\mathcal{K}}(A^{-T})_{\mathcal{C}_L,\mathcal{K}}$ is first computed locally on the processor holding blocks of $\hat{L}_{\mathcal{C}_L,\mathcal{K}}$ (step ③), and then reduced to the processor that owns the diagonal block $\hat{L}_{\mathcal{K},\mathcal{K}}$ within the column processor group to which supernode $\mathcal{K}$ is mapped (step ⓔ). The result of this reduction is added to the diagonal block during step ④. This completes the computation for the current supernode $\mathcal{K}$, and the algorithms moves to the next supernode.

3.3. Task scheduling and asynchronous collective communication

In Section 3.2, we have discussed how to exploit parallelism within a given supernode. Besides such intra-node parallelism, there is potentially a large amount of inter-node concurrency across the work associated with different supernodes.

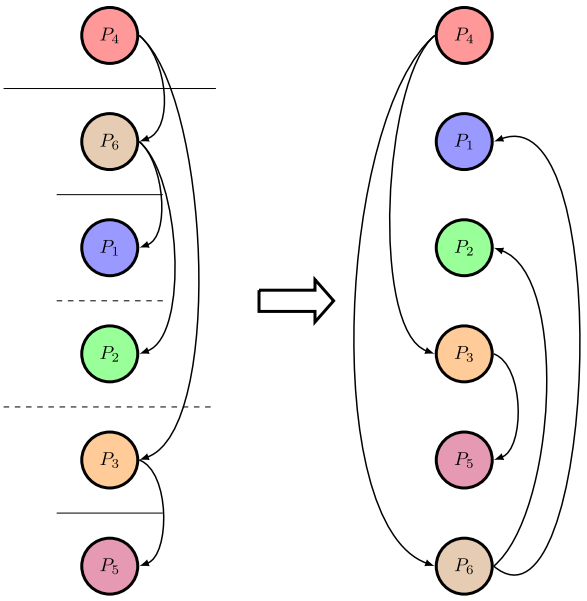


Fig. 4. A random shifted binary tree broadcast: ranks are randomly shifted before organizing the broadcast along a binary tree.

In [4] we have demonstrated that exploiting such inter-node parallelism is crucial for improving the parallel scalability of the `PSelInv` method for symmetric matrices. The basic idea is to use the elimination tree [28] associated with the sparse *LU* factorization to add an additional coarse-grained level of parallelism at the `for` loop level in Algorithm 1. For non-symmetric matrices we use the same strategy to exploit the inter-node parallelism.

We create a basic parallel task scheduler to launch different iterates of the `for` loop in a certain order. This order is defined by a priority list S, which is indexed by integer priority numbers ranging from 1 to n_s, where n_s is bounded from above by the depth of the elimination tree. The task performed in each iteration of the `for` loop is assigned a priority number $\sigma(\mathcal{I})$. The lower the number, the higher the priority of the task, hence the sooner it is scheduled. The supernode $\mathcal{N}$ associated with the root of the elimination tree clearly has to be processed first. If multiple supernodes or tasks have the same priority number, they are executed in a random order. Even though we use a priority list to help launch tasks, we do not place extra synchronization among launched tasks other than requiring them to preserve data dependency. Tasks associated with different supernodes can be executed concurrently if these supernodes are on different critical paths of the elimination tree, and if there is no overlap among processors mapped to these critical paths. We refer readers to [4] for more details on how to create such a task scheduler.

Collective communication operations such as broadcast and reduction in Section 3.2 dominate the communication cost of the `PSelInv` method. Each communication event involves potentially a different group of processors, and it is not practical to create an MPI communicator per group especially when a large number of processors are used. Instead, our implementation relies on asynchronous point-to-point MPI_Isend/MPI_Irecv routines to communicate between the processors. Take the broadcast operation for example, the simplest strategy is to let one processor to send information to all other processors within the relevant communication group. However, such a simple strategy can result in a highly imbalanced communication volume, as demonstrated in [20] for symmetric matrices. Instead, we employ the *shifted binary tree* method developed in [20] for asynchronous communication operations. Assuming that ranks are sorted, this type of tree is built by first shifting ranks of the recipients around a random position, and then by building a binary tree from the root to those shifted ranks. This technique prevents from always picking the same ranks as forwarding nodes in the binary tree, and thus further smooths communication load balance. An example of a such tree depicted in Fig. 4.

In the non-symmetric implementation of `PSelInv`, we therefore use non-blocking random shifted binary trees for the following operations:

1. broadcasting $\hat{L}_{\mathcal{C}_L,\mathcal{K}}$ to processors owning $(A^{-T})_{\mathcal{C}_L,\mathcal{C}_U}$ (step ⓐ),
2. broadcasting $\hat{U}_{\mathcal{K},\mathcal{C}_U}$ to processors owning $(A^{-T})_{\mathcal{C}_L,\mathcal{C}_U}$ (step ⓑ),
3. reducing contributions to $(A^{-T})_{\mathcal{K},\mathcal{C}_U}$ (step ⓒ),
4. reducing contributions to $(A^{-T})_{\mathcal{C}_L,\mathcal{K}}$ (step ⓓ),
5. reducing contributions to $(A^{-T})_{\mathcal{K},\mathcal{K}}$ (step ⓔ).

Table 1
Description of test problems for PSelInv.

Problem	Description
SIESTA_Si_512_k	KSDFT, Si with 512 atoms (complex structurally symmetric)
SIESTA_DNA_25_k	KSDFT, DNA with 17875 atoms (complex structurally symmetric)
SIESTA_DNA_64_k	KSDFT, DNA with 45760 atoms (complex structurally symmetric)
SIESTA_CBN_0.00_k	KSDFT, C-BN sheet with 12770 atoms (structurally symmetric)
SIESTA_Water_4x4x4_k	KSDFT, Water with 12288 atoms (complex structurally symmetric)
audikw_1	Automotive crankshaft model with over 900000 TETRA elements (real symmetric)
shyy161	Direct, fully-coupled method for solving the Navier-Stokes equations for viscous flow calculations (real non-symmetric)
stomach	Electro-physiological model of a Duodenum (real non-symmetric)
DG_DNA_715_64cell	KSDFT, DNA with 45760 atoms (complex symmetric)
DG_Graphene8192	KSDFT, Graphene sheet with 8192 atoms (complex symmetric)
SIESTA_C_BN_1x1	KSDFT, C-BN sheet with 2532 atoms (complex symmetric)
SIESTA_C_BN_2x2	KSDFT, C-BN sheet with 10128 atoms (complex symmetric)
SIESTA_C_BN_4x2	KSDFT, C-BN sheet with 20256 atoms (complex symmetric)

Table 2
The dimension n, the number of nonzeros $|A|$, and the number of nonzeros of the factors $|L+U|$ of the test problems.

problem	n	$\|A\|$	$\|L+U\|$
SIESTA_Si_512_k	6656	5,016,064	32,686,104
SIESTA_DNA_25_k	179,575	87,521,775	351,534,751
SIESTA_DNA_64_k	459,712	224,055,744	904,281,098
SIESTA_CBN_0.00_k	166,010	251,669,372	2,907,670,098
SIESTA_Water_4x4x4_k	94,208	32,706,432	1,388,275,840
audikw_1	943,695	77,651,847	2,530,341,547
shyy161	76,480	329,762	4,467,806
stomach	213,360	3,021,648	83,840,514
DG_DNA_715_64cell	459,712	224,055,744	898,749,546
DG_Graphene8192	327,680	238,668,800	1,968,211,450
SIESTA_C_BN_1x1	32,916	23,857,418	274,338,850
SIESTA_C_BN_2x2	131,664	95,429,672	1,655,233,542
SIESTA_C_BN_4x2	263,328	190,859,344	3,591,750,262

4. Numerical results

We evaluate the performance of PSelInv on a variety of problems, taken from sources including the SuiteSparse Matrix Collection[22], and matrices generated from the SIESTA [29] and DGDFT [30], two software packages for performing Kohn-Sham density functional theory [31] calculations using two different types of basis sets. The first matrix collection is a widely used benchmark set of problems for testing sparse direct methods, while the other set comes from practical large scale electronic structure calculations. The names of these matrices as well as some of their characteristics are listed in Tables 1 and 2. The matrices labeled by SIESTA_XXX_k are obtained from the SIESTA package with k-point sampling. These matrices are complex structurally symmetric matrices, but are neither complex symmetric nor Hermitian. The matrices labeled by DG_XXX and by SIESTA_XXX are complex symmetric matrices. We include these matrices in the test that compare the performance of the non-symmetric PSelInv solver with that of PSelInv for symmetric matrices.

In all of our experiments, we used the NERSC Edison platform with Cray XC30 nodes. Each node has 24 cores partitioned among two Intel Ivy Bridge processors. Each 12-core processor runs at 2.4 GHz. A single node has 64 GB of memory, providing more than 2.6 GB of memory per core. We run one MPI rank per core as an efficient multithreaded scheme is not yet available in PSelInv implementation. Computations are performed in complex arithmetic for all packages. Sparse matrices were reordered to reduce the amount of fill using PARMetis 4.0.3 [32] in all experiments. Before applying PSelInv, a *LU* factorization is first computed using SuperLU_DIST 5.1.0. In Section 4.3, we compare PSelInv to the MUMPS5.0.0 [9,10,24] package mainly to demonstrate the numerical accuracy, and get insight on the efficiency of our implementation as well.

4.1. Strong scaling experiments

We illustrate the strong scalability of PSelInv using several non-symmetric and symmetric matrices. In the latter case, the non-symmetric storage format is used and performance is compared against the symmetric implementation of PSelInv presented in [4,20] and available in the PEXSI package.[2] Each experiment is repeated 10 times and the average timing measurements are reported, together with error bars representing standard deviations in the plots.

[2] version 0.10.1 on http://www.pexsi.org/

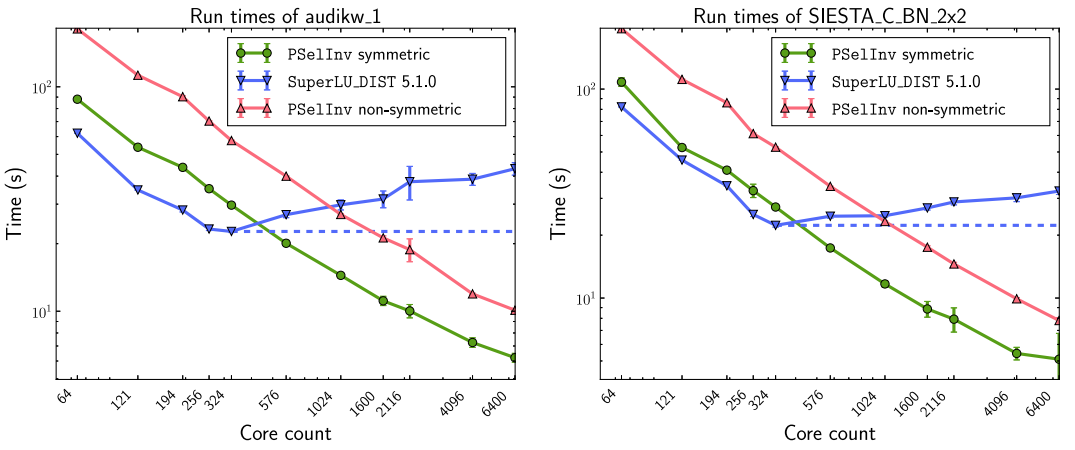


Fig. 5. Strong scaling of PSelInvon audikw_1 and SIESTA_C_BN_2x2 matrices.

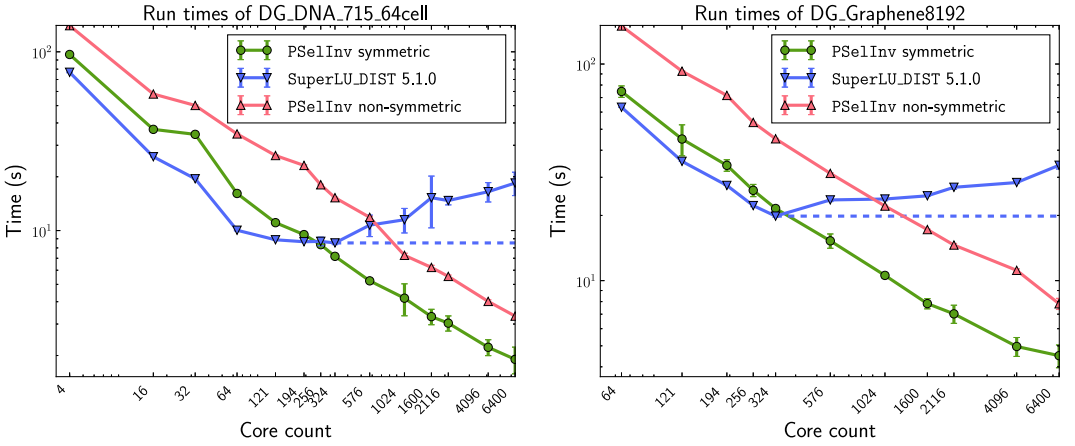


Fig. 6. Strong scaling of PSelInvon DG_DNA_64 and DG_Graphene8192 matrices.

Factorization timing measurements from SuperLU_DIST are provided as a reference. *LU* factorization is a pre-processing step of PSelInv, and needs to be added to the selected inversion time to reflect the overall cost required to compute the selected elements of the inverse matrix. Moreover, *LU* factorization and selected inversion have the same asymptotic computational cost but the actual cost may differ in practice. For the SIESTA_C_BN_2x2 matrix for instance, the *LU* factorization requires 1.78373×10^{13} floating point operations (flops). The selected inversion requires 3.59698×10^{13} flops, which is around 2 times larger. This needs to be taken into consideration when comparing the factorization times to the selected inversion times.

The first set of experiments (Figs. 5 and 6) demonstrate that the strong scalability of the non-symmetric version of PSelInv rivals that of the symmetric version. Over these 4 matrices, PSelInv can scale up to 6,400 cores. We also note that SuperLU_DIST can scale up to only 256 processors. Based on the study in [4], the scalability of PEXSI greatly benefits from the strategy for handling collective communication operations as well as the coarse-grain level parallelism. The runtime of the non-symmetric version of PSelInv is 1.5–2.1 times of that of the symmetric version, which illustrates the efficiency of the non-symmetric implementation despite the more complex communication pattern. In particular, we observe that such ratio tends to be smaller than 2.0 when more than 2000 cores are used. This is because we have removed some redundant data communication in the non-symmetric implementation of PSelInv, and we plan to pursue such improved implementation for the symmetric version of PSelInv in the future as well.

The next set of experiments focuses on assessing the efficiency of the PSelInv for the SIESTA_XXX_k matrices, which are only structurally symmetric. These matrices correspond to electronic structure calculations of 1D, 2D and 3D quantum systems. This results in the large difference in the ratio $|L+U|/|A|$ for different matrices. We also stress that we do not explicitly take advantage of the structural symmetry of the matrix. The results depicted in Figs. 7–9 demonstrate that the performance of PSelInv for non-symmetric matrices is comparable to that for symmetric matrices. PSelInv can scale to up to 6,400 cores on all problems except the SIESTA_Si_512_k matrix, which is significantly smaller in size. On the other hand, SuperLU_DIST can only scale to around 300 processors.

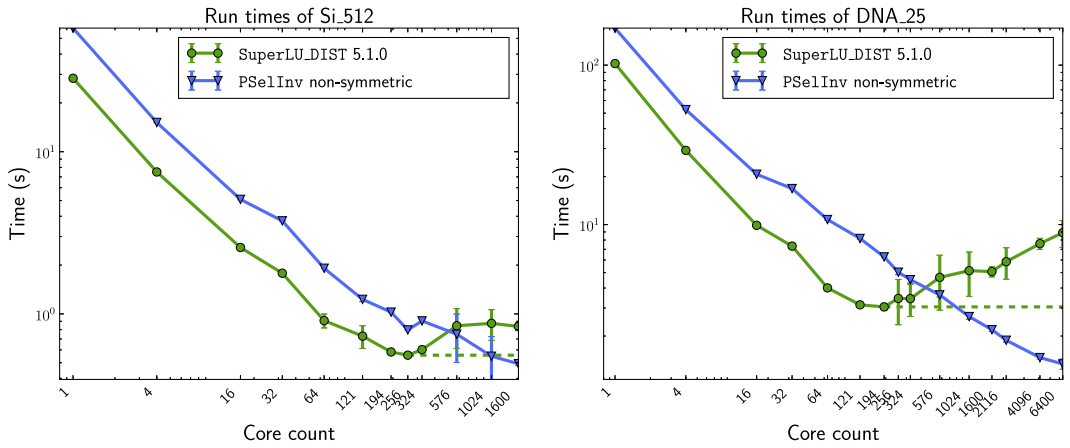


Fig. 7. Strong scaling of PSelInvon SIESTA_Si_512_k and SIESTA_DNA_25_k matrices.

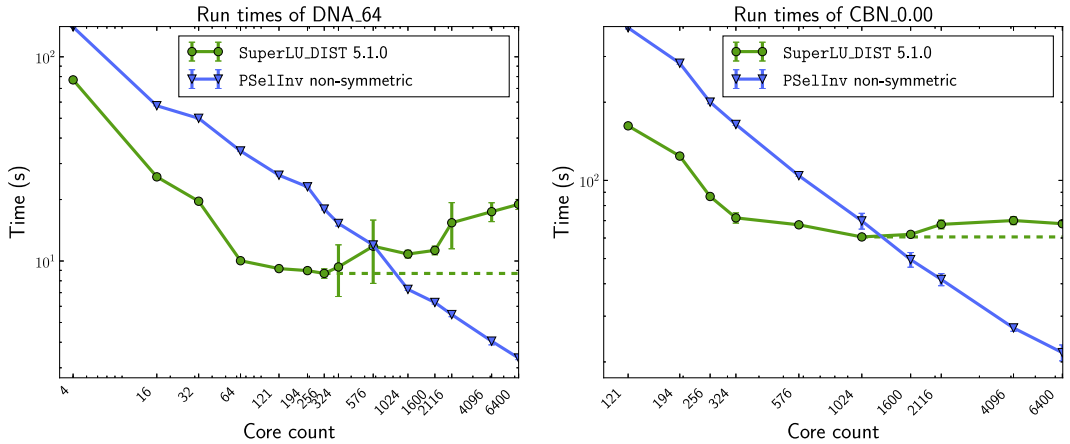


Fig. 8. Strong scaling of PSelInvon SIESTA_DNA_64_k and SIESTA_CBN_0.00_k matrices.

Table 3
Configurations used in the weak scaling experiments.

Problem	P	*flops*	*flops*/P
SIESTA_C_BN_1x1	30	1.6×10^{13}	533×10^{9}
SIESTA_C_BN_2x2	256	1.3×10^{14}	508×10^{9}
SIESTA_C_BN_4x2	576	3.0×10^{14}	520×10^{9}

4.2. Weak scaling experiment on symmetric matrices

In this section we evaluate the weak scalability of the non-symmetric version of PSelInv. Since the workload, measured by the flops of PSelInv, generally does not scale linearly with respect to the matrix size, we perform weak scaling tests by keeping the flops per core close to be constant while increasing the matrix size and the number of processors simultaneously. We choose the SIESTA_C_BN_XXX matrices for demonstrating both the weak scaling and the computational complexity of PSelInv. These matrices correspond to electronic structure calculations of two dimensional C-BN sheets of increasing sizes. For such matrices, asymptotic complexity analysis [12] shows that the flop count should increase by a factor of 8 from SIESTA_C_BN_1x1 to SIESTA_C_BN_2x2, but only by a factor of 2 from SIESTA_C_BN_2x2 to SIESTA_C_BN_4x2, respectively. The nonlinear growth behavior can be explained in terms of the size of the largest separator of the graph associated with the sparsity pattern of the matrix. In the former case, the size of the largest separator increases by a factor of 2. The dense matrix inversion corresponding to this separator leads to a factor of $2^3 = 8$ increase in flops. In the latter case, the size of the largest separator remains approximately the same despite the increase of the matrix size. Hence the flops approximately increases linearly with respect to the matrix size. Table 3 shows that the actual flop count obtained from PSelInv agrees well with the theoretical prediction: From SIESTA_C_BN_1x1 to SIESTA_C_BN_2x2 the flops increase by a factor of 8.1, while an increase by a factor of 2.3 is seen from SIESTA_C_BN_2x2 to SIESTA_C_BN_4x2. We choose the number

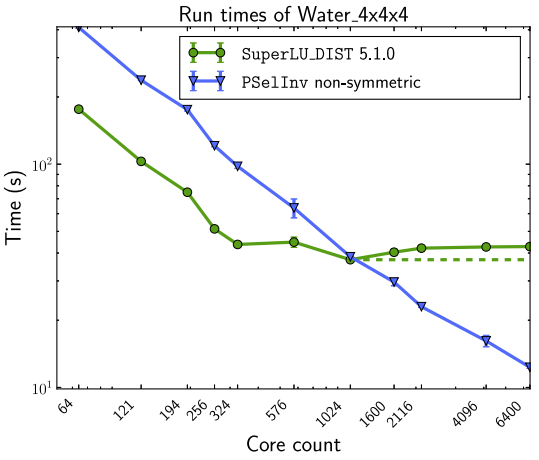


Fig. 9. Strong scaling of PSelInvon SIESTA_Water_4x4x4_k matrix.

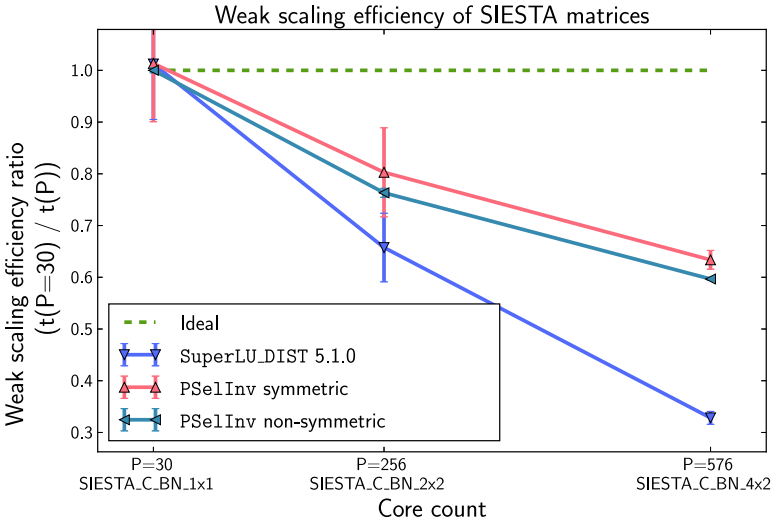


Fig. 10. Weak scaling of PSelInvand SuperLU_DISTon SIESTA sparse matrices.

of cores so that the number of flops per core is approximately 5×10^9. The largest number of cores we used for this test is 576 processors. This is due to the limitation of the strong scalability of SuperLU_DIST as observed in Section 4.1.

Fig. 10 shows that the non-symmetric implementation of PSelInv exhibits similar weak scalability compared to that of the symmetric case. We again repeat each experiment 10 times and report the averaged timing results, while error bars represent standard deviations. The line labeled by the "ideal" weak scaling is constructed by using the timing measurements obtained from a 30-core run. We observe that that both the symmetric and the non-symmetric versions of PSelInv exhibit better weak scalability than that of *LU* factorization implemented in SuperLU_DIST. The non-symmetric version achieves weak scaling efficiency of 59% on 576 cores, while the weak scaling efficiency of the symmetric version of PSelInv is slightly higher at 63%. The weak scaling efficiency of SuperLU_DIST is 33% when 576 cores are used.

4.3. Comparison against the MUMPS state-of-the-art solver

In this section, we provide a comparative study of the performance of the non-symmetric implementation of PSelInv against that of MUMPS (version 5.0.0), which is a state-of-the-art sparse matrix solver. In addition to *LU* factorization, the MUMPS package also offers an optimized algorithm for solving multiple sparse right-hand sides which can be used to perform selected inversion as well [9,10]. This approach is more generic than the one presented in this paper which is more restrictive on the element selection in the matrix inverse. Similarly to PSelInv, MUMPS first need to compute the *LU* factorization prior to computing the entries of the inverse. In the following, we use MUMPS to compute only the diagonal elements of the inverse matrix, or the full pattern described in Eq. (1). PSelInv computes all entries corresponding to Eq. (1) including the diagonal elements. Each experiment is repeated 5 times and average times are reported.

The results in Fig. 11 and Table 4 demonstrate that PSelInv can be orders of magnitude faster than the inversion available in MUMPS. Another interesting fact is that computing only the diagonal entries of the inverse with MUMPS is marginally

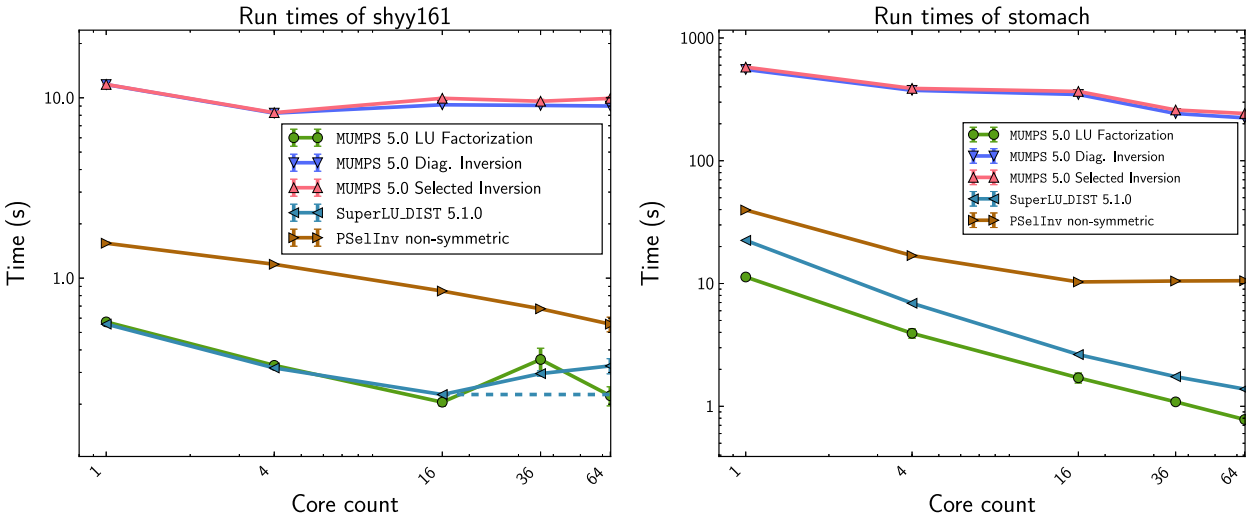


Fig. 11. Strong scaling of PSelInv and MUMPS5.0.0 on shyy161 and stomach matrices.

Table 4
Run times of PSelInvand MUMPS5.0.0 on shyy161 and stomach matrices.

p	MUMPS *LU* Factorization	MUMPS Diag. Inversion	MUMPS Selected Inversion	SuperLU_DIST Factorization	PSelInv
1	0.572069	11.841920	11.859480	0.556076	1.561718
4	0.328426	8.221570	8.272150	0.317725	1.193488
16	0.205184	9.149670	9.938566	0.226287	0.847977
36	0.354054	9.070224	9.571224	0.295593	0.676077
64	0.222778	9.003740	9.926730	0.326116	0.555733

(a) shyy161 matrix

p	MUMPS *LU* Factorization	MUMPS Diag. Inversion	MUMPS Selected Inversion	SuperLU_DIST Factorization	PSelInv
1	11.299000	553.8450	578.7244	22.455480	39.68122
4	3.932046	374.4712	388.2246	6.904646	16.89038
16	1.706144	346.5160	366.8144	2.638772	10.29646
36	1.086744	242.4746	259.1544	1.743026	10.48112
64	0.780408	223.562	242.4732	1.380420	10.53874

(b) stomach matrix

Table 5
Numerical error of values computed using PSelInvw.r.t. values computed by MUMPS5.0.0.

P	shyy161 $\|\|diag(A^{-1}_{\text{MUMPS}}) - diag(A^{-T}_{\text{PSelInv}})\|\|$	stomach $\|\|diag(A^{-1}_{\text{MUMPS}}) - diag(A^{-T}_{\text{PSelInv}})\|\|$
1	4.1813×10^{-15}	N.A.
4	4.1837×10^{-15}	3.0468×10^{-13}
16	4.1832×10^{-15}	3.0460×10^{-13}
36	4.1906×10^{-15}	3.0451×10^{-13}
64	4.1809×10^{-15}	3.0414×10^{-13}

less expensive than the full pattern, as most of these elements need to be computed internally to get the diagonal entries. The speedup achieved by PSelInv over MUMPS inversion reaches 9.75 for the shyy161 matrix, and 20.40 for the stomach matrix. The main reason to this is that MUMPS inversion is a more general algorithm, and it does not reuse computations as efficiently as PSelInv at the algorithm level.

Table 5 illustrates the accuracy of PSelInv is fully comparable to that of MUMPS, measured in terms of the diagonal entries of the matrix inverse. We remark that the shyy161 matrix, the diagonal contains elements with very small magnitude (some are zero elements). Therefore, row pivoting has to be used to move these elements to off-diagonal positions. SuperLU_DIST uses a static row pivoting strategy, while MUMPS employs a dynamic one. Table 5 shows that for the matrix we tested, the static row pivoting strategy is sufficient to obtain accurate matrix inverse elements. Note that the pivoting strategy impacts the *LU* factorization time only.

5. Conclusion

In this paper, we extend the parallel selected inversion algorithm called PSelInv, which is originally developed for symmetric matrices, to handle general non-symmetric matrices. The selected inversion algorithm can efficiently evaluate

the elements of A^{-1} indexed by the sparsity pattern of A^T. From an implementation perspective, it is more convenient and economical to formulate the selected inversion algorithm to compute selected elements of A^{-T} indexed by the sparsity pattern of $L+U$, where L, U are the LU factors for the possibly permuted matrix of A, because such a formulation allows us to overwrite the sparse matrix $L+U$ by the computed elements of A^{-T} *in situ*. We present the data distribution and communication patterns required to perform selected inversion in parallel. When a large number of processors are used, it is important to exploit coarse-grained level of concurrency available within the elimination trees to achieve high scalability. We also employ a tree-based asynchronous communication structure for handling various collective communication operations in the selected inversion algorithm. Our implementation of `PSelInv` is publicly available. Our numerical results demonstrates excellent scalability of `PSelInv` up to 6400 cores depending on the size and sparsity of the matrix. In the near future, we will explore the efficient implementation of `PSelInv` on heterogeneous many-core architecture such as GPU and Intel Knights Landing (KNL).

Acknowledgment

This work was partially supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences at Lawrence Berkeley National Laboratory (LBNL) through Contract No. DE-AC02-05CH11231 (M. J., L. L. and C. Y.), the National Science Foundation under Grant No. 1450372, and the Center for Applied Mathematics for Energy Research Applications (CAMERA) (L. L. and C. Y.). This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank Volker Blum, Alberto García, Xiaoye S. Li, François-Henry Rouet and Pieter Vancraeyveld for helpful discussion.

References

[1] K. Takahashi, J. Fagan, M. Chin, Formation of a sparse bus impedance matrix and its application to short circuit study, in: 8th PICA Conf. Proc., 1973.
[2] A. Erisman, W. Tinney, On computing certain elements of the inverse of a sparse matrix, Commun. ACM 18 (1975) 177.
[3] L. Lin, C. Yang, J. Meza, J. Lu, L. Ying, W. E, SelInv – an algorithm for selected inversion of a sparse symmetric matrix, ACM. Trans. Math. Softw. 37 (2011) 40.
[4] M. Jacquelin, L. Lin, C. Yang, Pselinv – a distributed memory parallel algorithm for selected inversion: the symmetric case, ACM Trans. Math. Softw. 43 (3) (2016) 21:1–21:28.
[5] Y.E. Campbell, T.A. Davis, Computing the sparse inverse subset: an inverse multifrontal approach, Technical Report, TR-95-021, University of Florida, 1995.
[6] S. Li, S. Ahmed, G. Klimeck, E. Darve, Computing entries of the inverse of a sparse matrix using the FIND algorithm, J. Comput. Phys. 227 (2008) 9408–9427.
[7] S. Li, E. Darve, Extension and optimization of the find algorithm: computing greens and less-than greens functions, J. Comput. Phys. 231 (4) (2012) 1121–1139.
[8] U. Hetmaniuk, Y. Zhao, M.P. Anantram, A nested dissection approach to modeling transport in nanodevices: algorithms and applications, Int. J. Numer. Method Eng. (2013).
[9] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, B. Uçar, On computing inverse entries of a sparse matrix in an out-of-core environment, SIAM J. Sci. Comput. 34 (2012) A1975–A1999.
[10] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, F.-H. Rouet, Parallel computation of entries of a^{-1}, SIAM J. Sci. Comput. 37 (2015) C268–C284.
[11] D.E. Petersen, S. Li, K. Stokbro, H.H.B. Sørensen, P.C. Hansen, S. Skelboe, E. Darve, A hybrid method for the parallel computation of Green's functions, J. Comput. Phys. 228 (2009) 5020–5039.
[12] L. Lin, J. Lu, L. Ying, R. Car, W. E, Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems, Commun. Math. Sci. 7 (2009) 755.
[13] A. Kuzmin, M. Luisier, O. Schenk, Fast methods for computing selected elements of the Greens function in massively parallel nanoelectronic device simulations, in: Euro-Par 2013 Parallel Processing, Springer, 2013, pp. 533–544.
[14] L. Lin, M. Chen, C. Yang, L. He, Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion, J. Phys. Condens. Matter 25 (2013) 295501.
[15] L. Lin, A. García, G. Huhs, C. Yang, SIESTA-PEXSI: massively parallel method for efficient and accurate *ab initio* materials simulation without matrix diagonalization, J. Phys. 26 (2014) 305503.
[16] G. Kotliar, S.Y. Savrasov, K. Haule, V.S. Oudovenko, O. Parcollet, C. Marianetti, Electronic structure calculations with dynamical mean-field theory, Rev. Mod. Phys. 78 (2006) 865–952.
[17] Z. Xu, A.C. Maggs, Solving fluctuation-enhanced Poisson-Boltzmann equations, arXiv:1310.4682(2013).
[18] G.H. Golub, C.F. Van Loan, Matrix Computations, third, Johns Hopkins Univ. Press, Baltimore, 1996.
[19] X. Li, J. Demmel, SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Softw. 29 (2003) 110.
[20] M. Jacquelin, L. Lin, N. Wichmann, C. Yang, Enhancing the scalability and load balancing of the parallel selected inversion algorithm via tree-based asynchronous communication, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 192–201.
[21] I. Duff, R. Grimes, J. Lewis, User's Guide for the Harwell-Boeing Sparse Matrix Collection, Research and Technology Division, Boeing Computer Services, Seattle, Washington, USA, 1992.
[22] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Software 38 (2011) 1.
[23] C. Ashcraft, R. Grimes, The influence of relaxed supernode partitions on the multifrontal method, ACM Trans. Math. Softw. 15 (1989) 291–309.
[24] P. Amestoy, I. Duff, J.-Y. L'Excellent, J. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM J. Matrix Anal. Appl. 23 (2001) 15–41.
[25] O. Schenk, K. Gartner, On fast factorization pivoting methods for symmetric indefinite systems, Electron. Trans. Numer. Anal. 23 (2006) 158–179.
[26] L.S. Blackford, ScaLAPACK User's Guide, 4, SIAM, 1997.
[27] E. Rothberg, A. Gupta, An efficient block-oriented approach to parallel sparse Cholesky factorization, SIAM J. Sci. Comput. 15 (1994) 1413–1439.
[28] J. Liu, The role of elimination trees in sparse factorization, SIAM J. Matrix Anal. Appl. 11 (1990) 134.

[29] J.M. Soler, E. Artacho, J.D. Gale, A. García, J. Junquera, P. Ordejón, D. Sánchez-Portal, The SIESTA method for ab initio order-N materials simulation, J. Phys. 14 (2002) 2745–2779.
[30] L. Lin, J. Lu, L. Ying, W. E, Adaptive local basis set for Kohn-Sham density functional theory in a discontinuous Galerkin framework I: total energy calculation, J. Comput. Phys. 231 (2012) 2140–2154.
[31] W. Kohn, L. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. 140 (1965) A1133–A1138.
[32] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1998) 359–392.