SIAM J. SCI. COMPUT.
Vol. 33, No. 3, pp. 1329–1351

A FAST PARALLEL ALGORITHM FOR SELECTED INVERSION OF STRUCTURED SPARSE MATRICES WITH APPLICATION TO 2D ELECTRONIC STRUCTURE CALCULATIONS*

LIN LIN†, CHAO YANG‡, JIANFENG LU§, LEXING YING¶, AND WEINAN E‖

Abstract. An efficient parallel algorithm is presented for computing selected components of A^{-1} where A is a structured symmetric sparse matrix. Calculations of this type are useful for several applications, including electronic structure analysis of materials in which the diagonal elements of the Green's functions are needed. The algorithm proposed here is a direct method based on a block LDL^T factorization. The selected elements of A^{-1} we compute lie in the nonzero positions of $L+L^T$. We use the elimination tree associated with the block LDL^T factorization to organize the parallel algorithm, and reduce the synchronization overhead by passing the data level by level along this tree using the technique of local buffers and relative indices. We demonstrate the efficiency of our parallel implementation by applying it to a discretized two dimensional Hamiltonian matrix. We analyze the performance of the parallel algorithm by examining its load balance and communication overhead, and show that our parallel implementation exhibits an excellent weak scaling on a large-scale high performance distributed-memory parallel machine.

Key words. selected inversion, parallel algorithm, electronic structure calculation

AMS subject classifications. 65F05, 65F50, 65Z05

DOI. 10.1137/09077432X

1. Introduction. In some scientific applications, we need to calculate a subset of entries of the inverse of a sparse symmetric matrix. One important example is the pole expansion algorithm [27, 29] for electronic structure analysis where the diagonal and sometimes subdiagonals of the discrete Green's function or resolvent matrices are needed in order to compute the electron density. Other examples in which particular entries of the Green's functions are needed can also be found in the perturbation analysis of impurities by solving Dyson's equation in solid state physics [15], or the

*Submitted to the journal's Methods and Algorithms for Scientific Computing section October 20, 2009; accepted for publication (in revised form) March 29, 2011; published electronically June 7, 2011. 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. DOE under contract number DE-AC02-05CH11232.
http://www.siam.org/journals/sisc/33-3/77432.html

†Program in Applied and Computational Mathematics, Princeton University, Princeton, NJ 08544 (linlin@math.princeton.edu). This author was partially supported by DOE under contract DE-FG02-03ER25587 and by ONR under contract N00014-01-1-0674. This author was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. DOE under contract number DE-AC02-05CH11231.

‡Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (cyang@lbl.gov). This author was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. DOE under contract number DE-AC02-05CH11231.

§Department of Mathematics, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012 (jianfeng@cims.nyu.edu). This author was partially supported by DOE under contract DE-FG02-03ER25587 and by ONR under contract N00014-01-1-0674.

¶Department of Mathematics and ICES, University of Texas at Austin, 1 University Station/C1200, Austin, TX 78712 (lexing@math.utexas.edu). This author was supported by an Alfred P. Sloan fellowship and NSF CAREER grant DMS-0846501.

‖Department of Mathematics and PACM, Princeton University, Princeton, NJ 08544 (weinan@math.princeton.edu). This author was partially supported by DOE under contract DE-FG02-03ER25587 and by ONR under contract N00014-01-1-0674.

calculation of retarded and less-than Green's function in electronic transport [10]. We will call this set of problems *selected inversion* of a matrix. This terminology is not to be confused with the concept of selective inversion used in [38, 39, 45] for improving the performance of the parallel triangular substitution step of solving a large-scale sparse symmetric linear system.

From a computational viewpoint, it is natural to ask whether one can develop algorithms for selected inversion that are faster than inverting the whole matrix. This is possible at least in some cases. Consider, for example, the finite difference discretization of a Hamiltonian operator of the form

$$H = -\frac{1}{2}\Delta + V(\mathbf{r}). \tag{1.1}$$

Here Δ is the Laplacian operator and V is the external or effective potential. The resulting matrix has certain structure, as in lattice models in statistical or quantum mechanics with a local Hamiltonian. For such matrices, a fast but sequential algorithm has been proposed to extract the diagonal or subdiagonal elements of its inverse matrix [28]. The algorithm is based on the LDL^T factorization of H. To extract the diagonal of H^{-1}, the algorithm computes $(H^{-1})_{i,j}$ for all i and j such that $L_{i,j} \neq 0$. The complexity of this algorithm is $\mathcal{O}(n^{1.5})$ for two dimensional (2D) problems and $\mathcal{O}(n^2)$ for three dimensional (3D) problems, with n being the dimension of H. Such complexity is much lower than the $\mathcal{O}(n^3)$ complexity associated with the direct inversion of the full matrix.

The present paper follows the concept in [28], and focuses on the parallel implementation of a block LDL^T factorization based selected inversion algorithm for a certain class of structured matrices. We point out that the use of relative indices and passing a local buffer array level by level along a branch of the parallel task tree is key to reducing communication cost and achieving excellent weak scaling.

This paper is organized as follows. In section 2, we introduce briefly the main concept of selected inversion based on LDL^T factorization. The algorithmic and implementation details of a parallel procedure that we have developed for selected inversion are presented in section 3. The performance of such a procedure is demonstrated and analyzed in section 4. In particular, we show that our parallel implementation can be used to solve problems defined on a $65{,}535 \times 65{,}535$ grid with more than four billion degrees of freedom on 4,096 processors in less than 25 minutes. We present an application of the algorithm to an electronic structure calculation for a quantum dot and compare its performance with a standard approach implemented in the software package Octopus [8].

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. When describing the algorithms and their implementations, we often use letters in a typewriter font such as `A`, `L`, and `D` to denote matrices stored in an appropriate format using an appropriate data structure. The letters `I`, `J`, and `K` are used to represent a column or row index set that contains a number of integers as its members. Occasionally, we use a MATLAB [34] 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.

2. Selected inversion based on LDL^T factorization: The basic idea. There are two main steps in the LDL^T factorization-based selected inversion algorithm. In the first step, we simply compute a sparse block LDL^T factorization of A.

We assume here that A is nonsingular and all pivots produced in the LDL^T factorization are sufficiently large so that no row or column permutation is needed during the factorization. In the second step, L and D are used to retrieve the selected elements of A^{-1} with a computational complexity comparable to that of the sparse block LDL^T factorization. No additional storage is required at this step. The main idea used here is very general. It dates back to [16, 44] and is related to other recent work [7, 28, 25, 36, 43, 2]. To set the notation, it will be helpful to first review the major operations involved including the LDL^T factorization. We will use a dense matrix for illustration first and then discuss the case for sparse matrices.

2.1. An algorithm for computing the inverse of a dense matrix. Let

$$A = \begin{pmatrix} \alpha & b^T \\ b & \hat{A} \end{pmatrix}, \tag{2.1}$$

with $\alpha \neq 0$. The first step of an LDL^T factorization produces a decomposition of A that can be expressed by

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

where $\ell = b/\alpha$ and $S = \hat{A} - bb^T/\alpha$ is known as a *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 yields the final L factor. The $(1,1)$ entry of each Schur complement together with α become the diagonal entries of the D matrix.

The key observation made in [25] and [28] is that A^{-1} can be expressed by

$$A^{-1} = \begin{pmatrix} \alpha^{-1} + \ell^T S^{-1} \ell & -\ell^T S^{-1} \\ -S^{-1}\ell & S^{-1} \end{pmatrix}. \tag{2.2}$$

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 by using the recipe given by (2.2). 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. 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, and D can always be kept as a diagonal matrix. For general symmetric matrices, a block LDL^T factorization that allows 2×2 block pivots [6, 5] or partial pivoting [19] may be used to achieve numerical stability in the factorization. As long as such a factorization is available, the algorithm and implementation discussed in this paper can be used to compute selected elements of A^{-1} that lie in the nonzero positions of $L + L^T$.

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, 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 with the lower triangular part of A^{-1} incrementally.

2.2. Selected inversion for sparse matrices. It is not difficult to observe that 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}, which simply solves the matrix equation $AX = I$, where I is the $n \times n$ identity matrix. Furthermore, all elements of A^{-1} are needed and computed. No computation cost can be saved if we just want to extract selected elements (for example, the diagonal elements) of A^{-1}.

However, when A is sparse, a tremendous amount of saving can be achieved if we are interested only in the diagonal of A^{-1}. If the vector ℓ in (2.2) 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 that correspond to the nonzero rows of ℓ are required.

Therefore, to compute selected elements A^{-1}, and in particular the diagonal elements of A^{-1}, we can simply modify the procedure shown in Algorithm 1 so that at each iteration we compute only selected elements of A^{-1} that will be needed by subsequent iterations of this procedure (see [16], where the same property is obtained using equations in [44]). 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 $L_{i,j} \neq 0$.

To see why this type of selected inversion is sufficient, we need only examine the nonzero structure of the kth column of L for all $k < j$ since the nonzero structure of these columns tells us which rows and columns of the trailing subblock of A^{-1} are needed to complete the calculation of the (k,k)th entry of A^{-1}. It is well known in the sparse matrix factorization literature [16, 14, 18] that the nonzero structures associated with different columns of L have a dependency relationship that can be characterized by the notion of an *elimination tree* [32]. 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 in the elimination tree if and only if

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

If we assume the jth diagonal element of D is not in a 2×2 block, then it follows from (2.2) that the selected elements of A^{-1} required to compute the (j,j)th element of A^{-1} must belong to the set $\{(A)^{-1}_{i,k} | \text{for } i \text{ and } k \text{ such that } L_{i,j} \neq 0 \text{ and } L_{k,j} \neq 0\}$. It is well known in the elimination tree theory (see, for example, Theorem 4.4 in [11])

that $L_{i,j} \neq 0$ implies that i must be an ancestor of j in the elimination tree. Therefore, the selected elements required to compute the (j,j)th element of A^{-1} must belong to rows and columns of A^{-1} that are among the ancestors of j. These elements are precisely those that have a corresponding nonzero entry in $L + L^T$. We remark that although we are motivated by calculating the diagonal and subdiagonal elements of A^{-1}, the selected inversion algorithm calculates A^{-1} restricted to the nonzero pattern of $L + L^T$. This sparse pattern is certainly larger than the set of diagonal and subdiagonal patterns of A^{-1}, but this is necessary in order to maintain the desired accuracy.

2.3. A block algorithm. 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 of a block LDL^T factorization of A:

(2.3)
$$A = \begin{pmatrix} A_{11} & B_{21}^T \\ B_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} A_{11} & \\ & A_{22} - B_{21}A_{11}^{-1}B_{21}^T \end{pmatrix} \begin{pmatrix} I & L_{21}^T \\ & I \end{pmatrix},$$

where $L_{21} = B_{21}A_{11}^{-1}$ and $S = A_{22} - B_{21}A_{11}^{-1}B_{21}^T$ is the Schur complement. In particular, a block version of (2.2) can be expressed by

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

There are at least three advantages of 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. It reduces indirect addressing overhead in sparse matrix computation.
3. It allows 2×2 block pivots that can be used to overcome numerical instabilities that may arise when A is indefinite.

If A is sparse, blocks can be defined in terms of what are called *supernodes* [13]. A supernode is a set of nodes whose corresponding columns (of its L factor) share the same nonzero structure below the diagonal. The definition of a supernode can be relaxed to include nodes whose corresponding columns in L are nearly identical [3].

3. Algorithmic and implementation details. In this section, we present the algorithmic and implementation details of a parallel procedure we have developed for selected inversion. The elements of A^{-1} to be calculated in the selected inversion algorithm coincide with the positions of the nonzero entries of $L + L^T$. No element of A^{-1} outside such pattern is calculated or stored. We assume a block LDL^T factorization is available, and make use of the elimination tree and other structure information that can be generated during a preprocessing step that involves both matrix reordering and symbolic factorization. For illustration purposes, we use a 2D Laplacian with nearest neighbor interaction, where the nearest neighbor is defined in terms of a standard five-point stencil, as an example in this section. However, the techniques we describe here are applicable to other higher order stencils for both 2D and 3D systems and to irregular problems obtained from, e.g., a finite element discretization. Although we have developed an efficient parallel implementation of a supernodal LDL^T factorization for 2D problems, we will focus our discussion on the selected inversion procedure

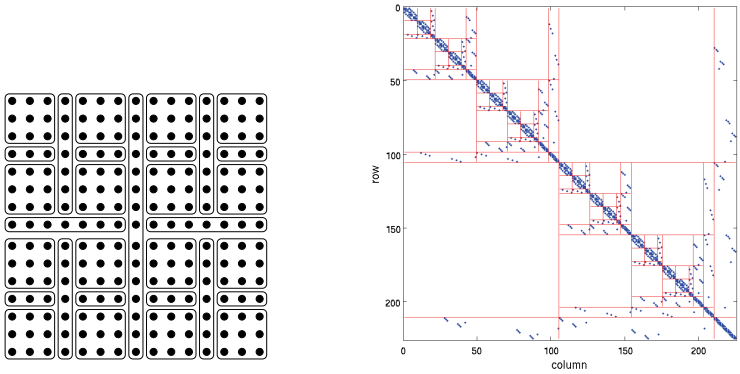
only. We refer readers to [30] for more detailed information about our parallel implementation of a block LDL^T factorization.

We will introduce some standard notation in section 3.1 and describe a sequential block selected inversion algorithm. Our basic strategy for parallelization of the block selected inversion procedure is to divide the computational work among different branches of the block elimination tree, which also serves as a parallel task tree. We will describe how this division of work is defined and how data is distributed among different processors in section 3.2.1. The main parallel selected inversion algorithm is presented in section 3.2.2. One of the key factors that affect the scalability of parallel computation is interprocessor synchronization. We will describe where synchronization is needed and discuss a technique for reducing the synchronization cost in section 3.2.3.

3.1. The sequential algorithm. Before we present the sequential algorithms for the selected inversion process, we need to introduce some notation and terminologies commonly used in the sparse matrix literature. We use the technique of *nested dissection* [17] to reorder and partition the sparse matrix A. For 2D problems defined on a rectangular grid, the nested dissection corresponds to a recursive partitioning of the 2D grid into a number of subdomains with a predefined minimal size. In the example shown in Figure 3.1(a), this minimal size is 3×3. The reordered matrix has a sparsity structure similar to that shown in Figure 3.1(b). Each subdomain is separated from other subdomains by *separators* that are defined in a hierarchical or recursive fashion. The largest separator is defined to be a set of grid points that divides the entire 2D grid into two subgrids of approximately equal sizes. Smaller separators can be constructed recursively within each subgrid. These separators are represented as rectangular oval boxes in Figure 3.1(a) and are labelled in the post order in Figure 3.2(a). The separators and minimal subdomains can be further organized in a tree structure shown in Figure 3.2(b). This tree is sometimes called a separator tree, which is also the elimination tree associated with a block LDL^T factorization of the reordered and partitioned matrix A. Each leaf node of the tree corresponds to a minimal subdomain. Other nodes of the tree correspond to separators defined at different levels of the partition. For general symmetric sparse matrices, separators and leaf nodes can be obtained from the analysis of the adjacency graph associated with the nonzero structure of A [23, 12, 9].

We will denote a set of row or column indices associated with each node in the separator tree by an uppercase typewriter typeface letter such as `I`. Each of these nodes corresponds to a diagonal block in the block diagonal matrix D produced from the block LDL^T factorization. A subset of columns in `I` may have a similar nonzero structure below the diagonal block. These columns can be grouped together to form what is known as a *supernode* or a *relaxed supernode*. (See [13] for a more precise definition of a supernode, and [3] for the definition of a relaxed supernode.) Sparse direct methods often take advantage of the presence of supernodes or relaxed supernodes in the reordered matrix A to reduce the amount of indirect addressing. Because the nonzero matrix element within a supernode can be stored as a dense matrix, we can take full advantage of `BLAS3` when working with supernodes.

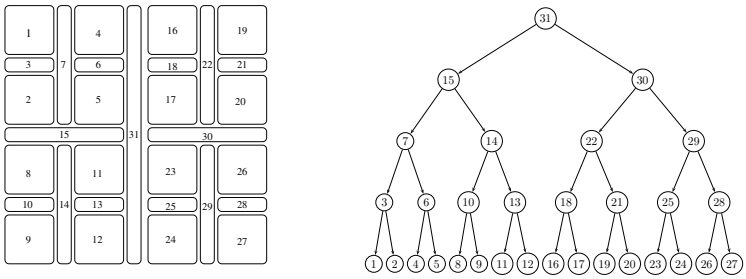
Once the separator tree and the block LDL^T factorization of A become available, we can use the pseudocode shown in Algorithm 2 to perform block selected inversion. As we can see from this pseudocode, `Ainv(J,K)` is calculated if and only if `L(J,K)` is a nonzero block. Such a calculation makes use of previously calculated blocks `Ainv(J,I)`, where both `J` and `I` are ancestors of the node `K`.



(a) Nested dissection partition and separators.

(b) The reordered and partitioned matrix A.

FIG. 3.1. *The nested dissection of a* 15×15 *grid and the elimination tree associated with a block* LDL^T *factorization of the* $2D$ *Laplacian defined on that grid.*



(a) The ordering of separators and subdomains.

(b) The separator (or elimination) tree in postorder.

FIG. 3.2. *The nested dissection ordering of the separators and subdomains and the associated separator tree.*

The pseudocode in Algorithm 2 treats the matrix block `L(J,I)` as if it were a dense matrix. As we can see from Figure 3.1(b), this is clearly not the case. In order to carry out the matrix-matrix multiplication efficiently, we must take advantage of these sparsity structures. In particular, we should not store the zero rows and columns in `L(I,K)`. Moreover, during the calculation of `Ainv(J,K)`, selected rows and columns of `Ainv(J,I)` must be extracted before the submatrix associated with these rows and columns is multiplied with the corresponding nonzero rows and columns of `L(I,K)`. We place the extracted rows and columns of `Ainv(J,I)` in a `Buffer` array in Algorithm 3. The `Buffer` array is then multiplied with the corresponding nonzero columns of `L(I,K)`. As a result, the product of the nonzero rows and columns of these matrices will have a smaller dimension. We will call the multiplication of the nonzero rows and columns of `Buffer` and `L(I,K)` a *restricted* matrix-matrix multiplication, and denote it by $\otimes$. The row and column indices associated with the needed rows and columns of `Ainv(J,I)` are called *absolute indices*. These indices can be predetermined by a *symbolic analysis* procedure described in [30]. This technique is similar to those used in [26] and [41].

Algorithm 2. A selected inversion algorithm for a sparse symmetric matrix A given its block LDL^T factorization $A = LDL^T$. The algorithm returns `Ainv(I,J)` for `I` and `J` such that `L(I,J)` $\neq 0$.

```
for K = {separator tree nodes arranged in reverse post order} do
  for J ∈ {ancestors of K} do
    Ainv(J,K) ← 0;
    for I ∈ {ancestors of K} do
      Ainv(J,K) ← Ainv(J,K) − Ainv(J,I) * L(I,K);
    end for
    Ainv(K,J) ← Ainv(J,K)^T;
  end for
  Ainv(K, K) ← D(K,K)^-1;
  for J ∈ {ancestors of K} do
    Ainv(K,K) ← Ainv(K,K)-L(J,K)^T * Ainv(J,K);
  end for
end for
```

Algorithm 3. Selected inversion of `A` with restricted matrix-matrix multiplication given its block LDL^T factorization.

```
subroutine SeqSelInverse
for K = {separator tree nodes arranged in reverse post order} do
  for J ∈ {ancestors of K} do
    Ainv(J,K) ← 0;
    for I ∈ {ancestors of K} do
      Retrieve absolute indices [JA,IA] of nonzero rows and columns of Ainv(J,I);
      Buffer ← Ainv(JA, IA);
      Ainv(J,K) ← Ainv(J,K) − Buffer ⊗ L(I,K);
    end for
    Ainv(K,J) ← transpose(Ainv(J,K));
  end for
  Ainv(K,K) ← D(K,K)^-1;
  for J ∈ {ancestors of K} do
    Ainv(K,K) ← Ainv(K,K) - L(J,K)^T ⊗ Ainv(J,K);
  end for
end for
return Ainv;
end subroutine
```

3.2. Parallelization. The sequential algorithm described above is very efficient for problems that can be stored on a single processor. For example, we have used the algorithm to compute the diagonal of a discretized Kohn–Sham Hamiltonian defined on a $2{,}047 \times 2{,}047$ grid. The entire computation, which involves more than four million degrees, took less than two minutes on an AMD Opteron processor.

For larger problems that we would like to solve in electronic structure calculation, the limited amount of memory on a single processor makes it difficult to store the L and D factors in-core. Furthermore, because the complexity of the computation is $\mathcal{O}(n^{3/2})$ in 2D [28], the CPU time required to complete a calculation on a single processor will eventually become excessively long.

Thus, it is desirable to modify the sequential algorithm so that the selected inversion process can be performed in parallel on multiple processors. The parallel

algorithm we describe below focuses on distributed memory machines that do not share a common pool of memory.

3.2.1. Task parallelism and data distribution. The elimination tree associated with the block LDL^T factorization of the reordered A (using nested dissection) provides natural guidance for parallelizing the factorization calculation. It can thus be viewed also as a *parallel task tree*. The same task tree can be used for carrying out selected inversion.

We divide the computational work among different branches of the tree. A *branch* of the tree is defined to be a path from the root to a node `K` at a given level ℓ as well as the entire subtree rooted at `K`. The choice of ℓ depends on the number of processors available. For a perfectly balanced tree, our parallel algorithm requires the number of processors p to be a power of two, and ℓ is set to $\log_2(p) + 1$. Figure 3.3(a) illustrates the parallel task tree in the case of four processors.

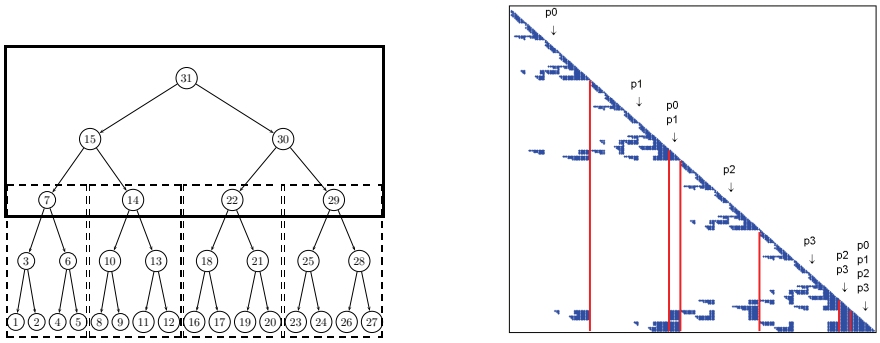
In terms of tree-node-to-processor mapping, each node at level ℓ or below is assigned to a unique processor. Above level ℓ, each node is shared by multiple processors. The amount of sharing is hierarchical, and depends on the level at which the node resides. For a perfectly balanced tree, a level-k node is shared by $2^{\ell-k}$ processors. We will use `procmap(J)` in the following discussion to denote the set of processors assigned to node `J`. Each processor is labeled by an integer processor identification (id) number between 0 and $p-1$. This processor id is known to each processor as `mypid`. In section 4, we show that this simple parallelization strategy leads to good load balance for a 2D Hamiltonian defined on a rectangular domain and discretized with a five-point stencil. For an irregular computational domain or a nonuniform mesh partitioning strategy, more complicated task-to-processor mapping algorithms should be used [37] to take into account the structure of the separator tree. It may also be necessary to perform task scheduling on the fly [1].

The data distribution scheme used for selected inversion is compatible with that used for LDL^T factorization. We should emphasize that the matrix `D(J,J)` in our implementation of the block LDL^T factorization is not necessarily diagonal. Again, we do not store the entire submatrix `L(I,J)`, but only the nonzero subblock within this submatrix as well as the starting location of the nonzero subblock.

In our parallel LDL^T factorization computation, the `L(I,J)` and `D(J,J)` submatrices associated with any `J` in an aggregated leaf node are stored on a single processor to which the aggregated leaf node is assigned. These matrices are computed using a sequential sparse LDL^T factorization algorithm on this processor. Furthermore, this computation is done independently without that of other processors.

When `J` is an ancestor of an aggregated leaf node, computing `L(I,J)` and `D(J,J)` requires the participation of all processors that are assigned to this node `procmap(J)`. As a result, it is natural to divide the nonzero subblock in `L(I,J)` and `D(J,J)` into smaller submatrices, and distribute them among all processors that belong to `procmap(J)`. Figure 3.3(b) illustrates how the columns of the L factor are partitioned and distributed among 4 processors.

Distributing these smaller submatrices among different processors is also necessary for overcoming the memory limitation imposed by a single processor. For example, for a 2D Hamiltonian defined on a $16{,}383 \times 16{,}383$ grid, the dimension of `D(J,J)` is 16,383 for the root node `J`. This matrix is completely dense, and hence contains $16{,}383^2$ matrix elements. If each element is stored in double precision, the total amount of memory required to store `D(J,J)` alone is roughly 2.1 GB. As we will see in section 4, the distribution scheme we use in our parallel algorithm leads to only



(a) Parallel task tree.

(b) Columns of the L factor are partitioned and distributed among different processors.

FIG. 3.3. *Task parallelism expressed in terms of a parallel task tree and corresponding matrix to processor mapping.*

a mild increase of memory usage per processor as we increase the problem size and the number of processors in proportion.

To achieve a good load balanced, we use a 2D block cyclic mapping consistent with that used by `ScaLAPACK` to distribute the nonzero blocks of `L(I,J)` and `D(J,J)` for any `J` that is an ancestor of an aggregated leaf node. In our parallel selected inversion algorithm, the distributed nonzero blocks of `L(I,J)` and `D(J,J)` are overwritten by the corresponding nonzero blocks of `Ainv(I,J)` and `Ainv(J,J)`.

3.2.2. Parallel selected inversion algorithm. Once the task-to-processor mapping and the initial data distribution are established, the parallelization of the selected inversion process can be described in terms of operations performed on different branches of the parallel task tree simultaneously by different processors. As illustrated in the subroutine `ParSelInverse` in Algorithm 4, each processor moves from the root of the task tree down towards an aggregated leaf node along a particular branch identified by `mybranch`. At each node `K`, it first computes `Ainv(J,K)` for ancestors `J` of `K` that satisfy $\texttt{L}(\texttt{J},\texttt{K}) \neq 0$. This calculation is followed by the computation of the diagonal block `Ainv(K,K)`. These two operations are accomplished by the subroutine `ParExtract` shown in the left column of Algorithm 5. All matrix-matrix multiplications in `ParExtract` are restricted matrix-matrix multiplications carried out in parallel among processors belonging to `procmap(K)`. Communication is required in these multiplications, and it is performed by the PBLAS subroutine `pdgemm`. A `Buffer` array is used to hold the distributed nonzero rows and columns of `Ainv(J,I)`. We should note here that a single `Buffer` array is allocated on each processor to hold distributed copies of `Ainv` blocks. The `Buffer` array is updated by the `ParRestrict` subroutine listed in the right column of Algorithm 5 as we move down the parallel task tree in Algorithm 4. Each update involves restricting the previously computed `Ainv` data from `procmap(K)` to `procmap(C)` for all children `C` of `K` so that the already computed `Ainv` blocks required at node `C` are properly redistributed among `procmap(C)`. Since the desired data in the `Buffer` array is passed level by level from a parent to its children, we need only know the relative positions of the `Ainv(J,I)` subblocks needed by a child within the `Buffer` array owned by its parent. These relative positions can be described by a set of *relative indices* `[IR, JR]` precomputed during the symbolic

analysis phase of the computation. This step is essential for reducing synchronization overhead and will be discussed further in section 3.2.3. To simplify our description, we use the notation `Buffer(JR,IR)` to represent the redistributed `Ainv(J,I)` block within the updated `Buffer` array. After `ParRestrict` is called, no communication is required between the processors assigned to different children of `K`. Finally, when each processor reaches an aggregated leaf node `K`, it calls the sequential selected inversion subroutine `SeqSelInverse` (Algorithm 3) to compute `Ainv(J,I)` for all descendants `I` of `K`. No interprocessor communication is required from this point on.

Algorithm 4. Parallel algorithm attained by extracting selected elements of the inverse of a symmetric matrix A.

subroutine `ParSelInverse`
`K` ← `root`;
while (`K` is not an aggregated leaf node) **do**
  Update `Ainv`(`K`, `K`) and `Ainv(J,K)` for all `J` ∈ {ancestors of `K`} such that `L(J,K)` $\neq 0$ by calling `ParExtract(K);`
  Update `Buffer` by calling `ParRestrict(K);`
  `K` ← `child(K)` along `mybranch`;
end while
Call `SeqSelInverse` within the subtree rooted at `K` to obtain `Ainv(J,I)` for all subtree nodes `I` and `J` ∈ {ancestors of `I`} such that `L(J,I)` $\neq 0$.
return `Ainv`;
end subroutine

3.2.3. Avoiding synchronization bottleneck. Avoiding synchronization bottleneck is the key to achieving scalable performance in selected inversion. Synchronization is needed in selected inversion as each processor proceeds from the root of the parallel task tree to an aggregated leaf node because the submatrix `L(I,K)` required at a particular node `K` of the parallel task tree is distributed in a block cyclic fashion among a larger group of processors that are mapped to the ancestors of `K`. Some of these processors will not participate in the computation of `Ainv(K,K)`. Therefore, data redistribution is required to move the required matrix elements from this larger group of processors to the set of processors in `procmap(K)`.

We use the `ScaLAPACK` subroutine `PDGEMR2D` to perform such a data redistribution. When `PDGEMR2D` is called to redistribute data from a larger processor group A to a smaller processor group B that is contained in A, all processors in A are *blocked*, meaning that no processor in A can proceed with its own computational work until the data redistribution initiated by processors in B is completed. This blocking feature of `PDGEMR2D`, while necessary for ensuring data redistribution is done in a coherent fashion, creates a potential synchronization bottleneck.

To be specific, when the selected nonzero rows and columns in `Ainv(J,I)` (Algorithm 5) are to be extracted from a large number of processors in `procmap(I)` and redistributed among a subset of processors in `procmap(K)`, a direct extraction and redistribution via the use of `PDGEMR2D` will block all processors in `procmap(I)`. If `K` is several levels away from `I`, a communication bottleneck that involves all processors in `procmap(I)` is created. This bottleneck makes the computation of `Ainv(J,K)` a sequential process for all descendants `K` of `I` that are at the same level.

The strategy we use to overcome this synchronization bottleneck is to place selected nonzero elements of `Ainv(J,I)` that would be needed for subsequent calculations in a `Buffer` array. Here the indices `I`, `J`, and `K` refer to general block row or

Algorithm 5. Selected inversion calculation subroutine `ParExtract` and data redistribution subroutine `ParRestrict` in the parallel implementation of selected inversion algorithm.

```
subroutine ParExtract(K)
for J ∈ {ancestors of K} do
  Ainv(J,K) ← 0;
  for I ∈ {ancestors of K} do
    • Retrieve the relative indices
    [JR,IR] associated with the
    distributed Ainv(J,I) block in
    Buffer;
    • Ainv(J,K) ← Ainv(J,K)−
      Buffer(JR,IR) ⊗ L(I,K);
    (requires communication within
    procmap(K))
  end for
  Ainv(K,J) ← Ainv(J,K)^T;
end for
Ainv(K,K) ← D(K,K);
for J ∈ {ancestors of K} do
  Ainv(K,K) ← Ainv(K,K)−
    L(J,K)^T ⊗ Ainv(J,K);
  (requires communication within
  procmap(K))
end for
return Ainv(J,K) for all J ∈
{ancestors of K} such that L(J,K) ≠ 0;
end subroutine
```

```
subroutine ParRestrict(K)
for all I ∈ {ancestors of K}∪{K} do
  Update Buffer by including Ainv(I,K)
  calculated from ParExtract(K);
end for
for all I,J ∈ {ancestors of K}∪{K} do
  for C ∈ {children of K} do
    if (L(J,C) ≠ 0 and L(I,C) ≠ 0), then
      • Retrieve the relative indices [JR,IR]
      of the Ainv(J,I) block in Buffer;
      • Update Buffer by redistributing
      Buffer(JR, IR) within procmap(C);
      (requires communication within
      procmap(K))
    end if
  end for
end for
return Buffer;
end subroutine
```

column indices for tree nodes. Selected subblocks of the `Buffer` array will be passed further to the descendants as each processor moves down the parallel task tree. The task of extracting necessary data and placing it in `Buffer` is performed by the subroutine `ParRestrict` shown in Algorithm 5. At a particular node `K`, the subroutine `ParRestrict` is called simultaneously by all processors in `procmap(K)`, and the `Buffer` array is redistributed among processors assigned to each child node `C` of `K` so that the subsequent multiplication of the nonzero blocks of `Ainv(J,I)` and `L(J,C)` can be carried out in parallel (by `pdgemm`). Because this distributed `Buffer` array contains all information that would be needed by descendants of `K`, no more direct reference to `Ainv(J,I)` is required. As a result, no communication is performed between processors that are assigned to different children of `K` once `ParRestrict` is called at node `K`.

As each processor moves down the parallel task tree within the `while` loop of the subroutine `ParSelInverse` in Algorithm 4, the amount of data extracted from the `Buffer` array by the `ParRestrict` subroutine becomes smaller and smaller. The newly extracted data are distributed among a smaller number of processors also. Each call to `ParRestrict(K)` requires a synchronization of all processors in `procmap(K)`, hence incurring some synchronization overhead. This overhead becomes smaller as each processor gets closer to an aggregated leaf node because each `ParRestrict` call is then performed within a small group of processors. When an aggregated leaf node is reached, all selected nonzero rows and columns of `Ainv(J,I)` required in subsequent computation are available in the `Buffer` array allocated on each processor. As a result, no communication is required among different processors from this point on.

The use of a `Buffer` array in our parallel selected inversion algorithm introduces a modest amount of memory overhead. Our selected inversion algorithm computes elements of A^{-1} that lie in the nonzero positions of $L + L^T$, and all matrices are distributed among all processors. The size of the `Buffer` array is the maximum size of distributed `Ainv` restricted to the longest critical path on the parallel task tree. This size is generally smaller than the size of distributed `L`. It does not grow rapidly as the number of processors increases. This is demonstrated in section 4 where we show the total amount of memory usage per processor increases logarithmically with respect to the matrix size in our implementation. We also remark that the size of the buffer array may increase in the more general scenario where the elimination tree is far from a binary tree.

4. Performance. In this section, we report the performance of our implementation of the selected inversion algorithm for a discretized 2D Kohn–Sham Hamiltonian H using a five-point stencil with a zero shift, which we will refer to as **PSelInv** in the following. All elements of H^{-1} restricted to the nonzero pattern of $L + L^T$ are calculated. The nested dissection procedure stops when the dimension of the subdomain is 3×3. We analyze the performance statistics by examining several aspects of the implementation that affect the efficiency of the computation and communication. Our performance analysis is carried out on the Franklin system maintained at the National Energy Research Scientific Computing (NERSC) Center. Franklin is a distributed-memory parallel system with 9,660 compute nodes. 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 gigaflops (Gflops) per second per core. Each compute node has 8 GB of memory (2 GB per core). Each compute node is connected to a dedicated SeaStar2 router through Hypertransport with a 3D torus topology that ensures high performance, low-latency communication for the message passing interface (MPI). The floating point calculation is done in 64-bit double precision. We use 32-bit integers to keep index and size information.

Our implementation of the selected inversion achieves very high single processor performance which we described in detail in [30]. In particular, when the grid size reaches 2,047, we are able to reach 67% (6.16/9.2) of the peak performance of a single Franklin core.

In this section, we will mainly focus on the parallel performance of our algorithm and implementation. Our objective for developing a parallel selected inversion algorithm is to enable us and other researchers to study the electronic structure of large quantum mechanical systems when a vast amount of computational resource are available. Therefore, our parallelization is aimed at achieving a good *weak scaling*. Weak scaling refers to a performance model similar to that used by Gustafson [20]. In such a model, performance is measured by how quickly the wall clock time increases as both the problem size and the number of processors involved in the computation increase. Because the complexity of the factorization and selected inversion procedures is $\mathcal{O}(n^{3/2})$, where $n = m^2$ is the matrix dimension and m is the number of grid points in one dimension, we will simply call m the *grid size* in the following. We also expect that, in an ideal scenario, the wall clock time should increase by a factor of two when the grid size doubles and the number of processor quadruples.

In addition to using `MPI_Wtime()` calls to measure the wall-clock time consumed by different components of our code, we also use the integrated performance monitoring (IPM) tool [42], the `CrayPat` performance analysis tool [22], as well as `PAPI` [35] to measure various performance characteristics of our implementation.

4.1. Parallel scalability. In this section, we report the performance of our implementation when it is executed on multiple processors. Our primary interest is in the weak scaling of the parallel computation with respect to an increasing problem size and an increasing number of processors. The strong scaling of our implementation for a problem of fixed size is described in [30].

In terms of weak scaling, PSelInv performs quite well with up to 4,096 processors for problems defined on a $65{,}535 \times 65{,}535$ grid (with corresponding matrix dimension around 4.3 billion). In Table 4.1, we report the wall clock time recorded for several runs on problems defined on square grids of different sizes. To measure weak scaling, we start with a problem defined on a $1{,}023 \times 1{,}023$ grid, which is solved on a single processor. When we double the grid size, we increase the number of processors by a factor of 4. In an ideal scenario in which communication overhead is small, we should expect to see a factor of two increase in wall clock time every time we double the grid size and quadruple the number of processors used in the computation. Such prediction is based on the $\mathcal{O}(m^3)$ complexity of the computation. In practice, the presence of communication overhead will lead to a larger amount of increase in total wall clock time. Hence, if we use $t(m, n_p)$ to denote the total wall clock time used in an n_p-processor calculation for a problem defined on a square grid with grid size m, we expect the weak scaling ratio defined by $\tau(m, n_p) = t(m/2, n_p/4)/t(m, n_p)$, which we show in the second to the last column of Table 4.1, to be larger than two. However, as we can see from this table, deviation of $\tau(m, n_p)$ from the ideal ratio of two is quite modest even when the number of processors used in the computation reaches 4,096.

A closer examination of the performance associated with different components of our implementation reveals that our parallel symbolic analysis takes a nearly constant amount of time that is a tiny fraction of the overall wall clock time for all configurations of problem size and the number of processors. This highly scalable performance is primarily due to the fact that most of the symbolic analysis performed by each processor is carried out within an aggregated leaf node that is completely independent from other leaf nodes.

Table 4.1 shows that the performance of our block selected inversion subroutine achieves nearly ideal weak scaling up to 4,096 processors. The scaling of flops and wall clock time can be better viewed in Figure 4.1, where the code performance is compared to ideal performance using a log-log plot. We should point out that the performance of our implementation of the parallel LDL^T factorization is comparable to that achieved by state-of-the-art sparse matrix software packages such as MUMPS [1] on a relatively small 2D problem used in our experiment, even though our factorization includes the

TABLE 4.1

The scalability of parallel computation used to obtain A^{-1} for A for increasing system sizes. The largest grid size is 65,535×65,535, and the corresponding matrix size is approximately 4.3 billion.

Grid size	n_p	Symbolic time	Factorization time	Inversion time	Total time	Weak scaling ratio	% comm
1,023	1	0.92	7.29	6.77	14.99	–	0.0%
2,047	4	1.77	14.44	13.82	30.04	2.00	2.5%
4,095	16	1.82	34.26	25.39	61.82	2.05	11.4%
8,191	64	1.91	86.35	47.07	135.34	2.18	20.4%
16,383	256	1.98	207.51	89.91	299.41	2.21	28.4%
32,767	1024	2.08	474.94	174.57	651.59	2.17	34.5%
65,535	4096	2.40	1109.09	348.13	1459.62	2.24	40.8%

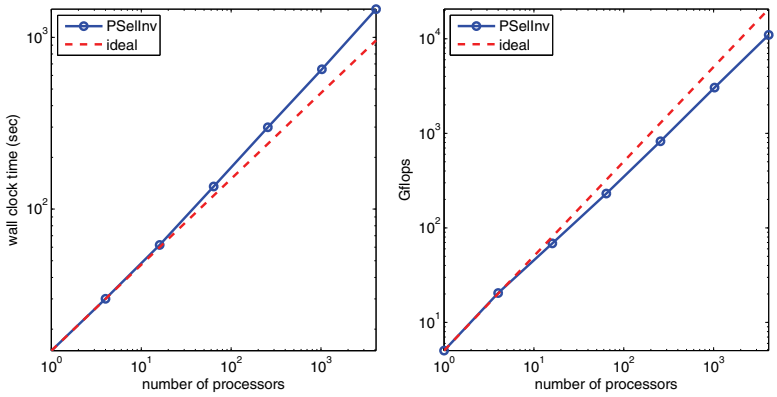


FIG. 4.1. *Log-log plot of total wall clock time and total Gflops with respect to the number of processors, compared with ideal scaling. The grid size starts from* 1023×1023*, and is proportional to the number of processors.*

additional computation of using the ScaLAPACK subroutines `pdgetri` to invert the diagonal blocks of D. (We have not been able to use MUMPS to factor problems that are discretized with $8{,}191 \times 8{,}191$ or more grid points.) From Table 4.1, we can also see that the selected inversion time is significantly less than that associated with factorization when the problem size becomes sufficiently large. This is due primarily to the fact that selected inversion involves a smaller amount of indirect addressing, and almost all float point operations involved in block selected inversion are dense matrix-matrix multiplications.

4.2. Load balance. To have a better understanding of the parallel performance of our code, let us now examine how well the computational load is balanced among different processors. Although we try to maintain a good load balance by distributing the nonzero elements in `L(I,J)` and `D(J,J)` as evenly as possible among processors in `procmap(J)`, such a data distribution strategy alone is not enough to achieve perfect load balance as we will see below.

One way to measure load balance is to examine the flops performed by each processor. We collected such statistics by using PAPI [35]. Figure 4.2 shows the overall flop counts measured on each processor for a 16-processor run of the selected inversion for A defined on a $4{,}095 \times 4{,}095$ grid. There is clearly some variation in operation counts among the 16 processors. Such variation contributes to idle time that shows up in the communication profile of the run, which we will report in the next subsection. Such variation can be explained by the different ways the separator tree nodes can be ordered and the tree's relationship with the 2D grid topology [30]. We should note that, in general, a better matrix ordering scheme can improve the load balance of selected inversion calculation.

4.3. Communication overhead. A comprehensive measurement of the communication cost can be collected using the `IPM` tool. Table 4.1 shows that the overall communication cost increases moderately as we double the problem size and quadruple the number of processors at the same time.

As we discussed earlier, the communication cost can be attributed to the following three factors:

1. Idle wait time. This is the amount of time a processor spends waiting for other processors to complete their work before proceeding beyond a synchronization

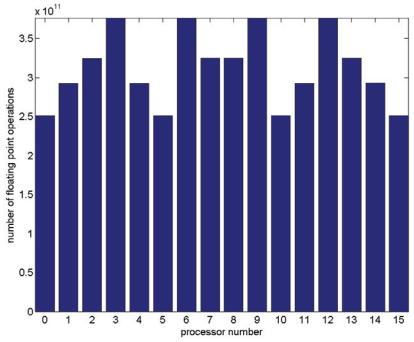


FIG. 4.2. *The number of flops performed on each processor for the selected inversion of* A^{-1} *defined on a* $4{,}095 \times 4{,}095$ *grid.*

point.

2. Communication volume. This is the amount of data transferred among different processors.
3. Communication latency. This factor pertains to the startup cost for sending a single message. The latency cost is proportional to the total number of messages communicated among different processors.

The communication profile provided by `IPM` shows that `MPI_Barrier` calls are the largest contributor to the communication overhead. An example of such a profile obtained from a 16-processor run on a $4{,}095 \times 4{,}095$ grid is shown in Figure 4.3. In this particular case, `MPI_Barrier` represents more than 50% of all communication cost. The amount of idle time the code spent in this MPI function is roughly 6.3% of the overall wall clock time.

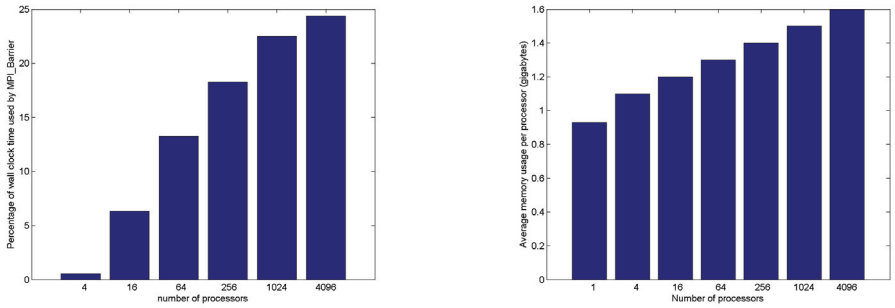
The `MPI_Barrier` and `BLACS_Barrier` (which shows up in the performance profile as `MPI_Barrier`) functions are used in several places in our code. In particular, the barrier functions are used in the selected inversion process to ensure relative indices are properly computed by each processor before selected rows and columns of the matrix block associated with a higher level node are redistributed to its descendants. The idle wait time spent in these barrier function calls is due to the variation of computational loads discussed in section 4.2. Using the call graph provided by `CrayPat`, we examined the total amount of wall clock time spent in these `MPI_Barrier` calls. For the 16-processor run (on the $4{,}095 \times 4{,}095$ grid), this measured time is roughly 2.6 seconds, or 56% of all idle time spent in `MPI_Barrier` calls. The rest of the `MPI_Barrier` calls are made in ScaLAPACK matrix-matrix multiplication routine `pdgemm`, and dense matrix factorization and inversion routines in `pdgetrf` and `pdgetri`, respectively.

Figure 4.4(a) shows that the percentage of wall clock time spent in `MPI_Barrier` increases moderately as more processors are used to solve larger problems. Such an increase is due primarily to the increase in the length of the critical path in both the elimination tree and in the dense linear algebra calculations performed on each separator.

In addition to the idle wait time spent in `MPI_Barrier`, communication overhead is also affected by the volume of data transfered among different processors and how frequent these transfers occur. It is not difficult to show that the total volume of communication should be proportional to the number of nonzeros in L and independent from the number of processors used. Figure 4.3 shows that the total amount of

[name]	[time]	[calls]	<%mpi>	<%wall>
MPI_Barrier	67.7351	960	52.21	6.32
MPI_Recv	30.4719	55599	23.49	2.84
MPI_Reduce	16.6104	18260	12.80	1.55
MPI_Send	7.86273	25865	6.06	0.73
MPI_Bcast	5.86476	100408	4.52	0.55
MPI_Allreduce	0.842473	320	0.65	0.08
MPI_Isend	0.261145	29734	0.20	0.02
MPI_Testall	0.0563367	33515	0.04	0.01
MPI_Sendrecv	0.0225533	1808	0.02	0.00
MPI_Allgather	0.00237397	16	0.00	0.00
MPI_Comm_rank	8.93647e-05	656	0.00	0.00
MPI_Comm_size	1.33585e-05	32	0.00	0.00

FIG. 4.3. *Communication profile for a* 16*-processor run on a* 4,095 × 4,095 *grid.*



(a) The percentage of time spent in `MPI_Barrier` as a function of n_p (and the corresponding grid size m).

(b) The average memory usage per processor as a function of n_p and m.

FIG. 4.4. *Communication overhead and memory usage profile.*

wall clock time spent in MPI data transfer functions `MPI_Send`, `MPI_Recv`, `MPI_ISend`, `MPI_Reduce`, `MPI_Bcast`, and `MPI_Allreduce` etc. is less than 5% of the overall wall clock time for a 16-processor run on a 4,095 × 4,095 grid. Some of the time spent in `MPI_Recv` and collective communication functions such as `MPI_Reduce` and `MPI_Bcast` corresponds to idle wait time that is not accounted for in `MPI_Barrier`. Thus, the actual amount of time spent in data transfer is much less than 5% of the total wall clock time. This observation provides an indirect measurement of the relatively low communication volume produced in our calculation.

In terms of the latency cost, we can see from Figure 4.3 that the total number of MPI related function calls made by all processors is roughly 258,000 (obtained by adding up the call numbers in the third column). Therefore, the total number of messages sent and received per processor is roughly 16,125. The latency for sending one message on Franklin is roughly 8 microsecond. Hence, the total latency cost for this particular run is estimated to be roughly 0.13 seconds, a tiny fraction of the overall wall clock time. Therefore, latency does not contribute much to communication overhead.

4.4. Memory consumption. In addition to maintaining good load balance among different processors, the data-to-processor mapping scheme discussed in section 3.2.1 also ensures that the memory usage per core only increases logarithmically with respect to the matrix dimension in the context of weak scaling. This estimation is based on the observation that when the grid size is increased by a factor of two, the dimensions of the extra blocks associated with `L` and `D` are proportional to the grid size, and the total amount of extra memory requirement is proportional to the square of the grid size. Since the number of processors is increased by a factor of four, the extra memory requirement stays fixed regardless of the grid size. This logarithmic dependence is clear from Figure 4.4(b), where the average memory cost per core with respect to number of processors is shown. The x-axis is plotted in logarithmic scale.

5. Application to electronic structure calculation of 2D rectangular quantum dots. In this section, we show how the algorithm and implementation we described in section 3 can be used to speed up electronic structure calculations within the density functional theory (DFT) framework [21, 24]. The example we use here is a 2D electron quantum dot confined in a rectangular domain, a model investigated in [40]. This model is also provided in the test suite of the Octopus software [8], which we use for comparison.

The most time-consuming part of a DFT electronic structure calculation is the evaluation of the electron density

$$\rho = \mathrm{diag}(f_{\beta,\mu}(H)), \tag{5.1}$$

where $f_{\beta,\mu}(t) = 2/(1 + e^{\beta(t-\mu)})$ is the Fermi–Dirac distribution function with β a parameter that is proportional to the reciprocal of the temperature and μ the chemical potential. The symmetric matrix H in (5.1) is a discretized Kohn–Sham Hamiltonian [33] defined as

$$H = -\frac{1}{2}\Delta + V_H(\mathbf{r}) + V_{\mathrm{xc}}(\mathbf{r}) + V_{\mathrm{ext}}, \tag{5.2}$$

where Δ is the Laplacian, V_H is the Hartree potential, V_{xc} is a 2D exchange-correlation potential constructed via the local density approximation (LDA) theory [33, 4], and V_{ext} is an external potential that describes an infinite hard-wall confinement in the xy plane, i.e.,

$$V_{\mathrm{ext}}(x,y) = \begin{cases} 0, & 0 \le x \le L, 0 \le y \le L, \\ \infty & \text{elsewhere.} \end{cases} \tag{5.3}$$

To simplify our experiment, we do not consider spin-polarization.

The standard approach for evaluating (5.1) is to compute the invariant subspace associated with a few smallest eigenvalues of H. This approach is used in Octopus [8], which is a real space electronic structure calculation software package.

An alternative way to evaluate (5.1) is to use a recently developed pole expansion technique [27, 29] to approximate $f_{\beta,\mu}$. The pole expansion technique expresses the electron density ρ as a linear combination of the diagonal of $(H - (\mu + z_i)I)^{-1}$, i.e.,

$$\rho \approx \sum_{i=1}^{P} \mathrm{Im}\ \left(\mathrm{diag}\, \frac{\omega_i}{H - (\mu + z_i)I}\right). \tag{5.4}$$

Here Im (A) stands for the imaginary part of A. The parameters z_i and ω_i are the complex shift and weight associated with the ith pole, respectively. They can

be chosen so that the total number of poles P is minimized for a given accuracy requirement. At room temperature, the number of poles required in (5.4) is relatively small (less than 100). In addition to the temperature, the pole expansion (5.4) also requires an explicit knowledge of the chemical potential μ, which must be chosen so that the condition

$$\text{trace}(f_{\beta,\mu}(H)) = n_e \tag{5.5}$$

is satisfied. This can be accomplished by solving (5.5) using the standard Newton's method.

In order to use (5.4), we need to compute the diagonal of the inverse of a number of complex symmetric (non-Hermitian) matrices $H - (z_i + \mu)I$ $(i = 1, 2, \ldots, P)$. The implementation of the parallel selected inversion algorithm described in section 3 can be used to perform this calculation efficiently, as the following example shows.

In this example, the Laplacian operator Δ is discretized using a five-point stencil. A room temperature of 300K (which defines the value of β) is used in our calculation. The area of the quantum dot is L^2. In a two-electron dot, setting $L = 1.66$Å and discretizing the 2D domain with 31×31 grid points yields a total energy error that is less than 0.002Ha. When the number of electrons becomes larger, we increase the area of the dot in proportion so that the average electron density is fixed. A typical density profile with 32 electrons is shown in Figure 5.1. In this case, the quantum dot behaves like a metallic system with a tiny energy gap around 0.08eV.

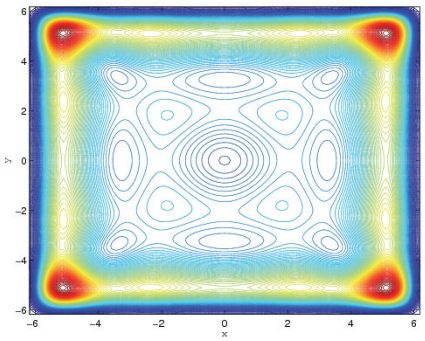


FIG. 5.1. *A contour plot of the density profile of a quantum dot with* 32 *electrons.*

We compare the density evaluation (5.1) performed by both Octopus and the pole expansion technique. In Octopus, the invariant subspace associated with the smallest $n_e/2 + n_{\text{h}}$ eigenvalues of H is computed using a conjugate gradient (CG) like algorithm, where n_e is the number of electrons in the quantum dot and n_{h} is the number of extra states for finite temperature calculation. The value of n_{h} depends on the system size and temperature. For example, in the case of 32 electrons, 4 extra states are necessary for the electronic structure calculation at 300K. In the pole expansion approach, we use 80 poles in (5.4), which, in general, could give a relative error in electron density on the order of 10^{-7} (in L_1 norm) [29].

In addition to using the parallel algorithm presented in section 3 to evaluate each term in (5.4), an extra level of coarse grained parallelism can be achieved by assigning each pole to a different group of processors.

In Table 5.1, we compare the efficiency of the pole expansion technique for the quantum dot density calculation performed with the standard eigenvalue calculation

TABLE 5.1
Timing comparison of electron density evaluation between Octopus and PCSelInv for systems of different sizes. The multiplication by 80 *in the last column accounts for the use of* 80 *pole in* (5.4).

n_e(#Electrons)	Grid	#proc	Octopus time(s)	PCSelInv time(s)
2	31	1	< 0.01	0.01×80
8	63	1	0.03	0.06×80
32	127	1	0.78	0.03×80
128	255	1	26.32	1.72×80
		4	10.79	0.59×80
512	511	1	1091.04	9.76×80
		4	529.30	3.16×80
		16	131.96	1.16×80
2048	1023	1	out of memory	60.08×80
		4	out of memory	19.04×80
		16	7167.98	5.60×80
		64	1819.39	2.84×80

approach implemented in Octopus. The maximum number of CG iterations for computing each eigenvalue in Octopus is set to the default value of 25. We label the pole expansion-based approach that uses the algorithm and implementation discussed in section 3 as PCSelInv, where the letter C stands for complex. The factor 80 in the last column of Table 5.1 accounts for 80 poles used in (5.4). When a massive number of processors are available, this pole number factor will easily result in a factor of 80 reduction in wall clock time for the PCSelInv calculation, whereas such a perfect reduction in wall clock time cannot easily be obtained in Octopus.

We observe that for quantum dots that contain a few electrons, the standard density evaluation approach implemented in Octopus is faster than the pole expansion approach. However, when the number of electrons becomes sufficiently large, the advantage of the pole expansion approach using the algorithms presented in section 3 to compute $\mathrm{diag}[H - (z_i + \mu)I]^{-1}$ becomes quite evident. This is because the computation cost associated with the eigenvalue calculation in Octopus is dominated by the computation performed to maintain mutual orthogonality among different eigenvectors when the number of electrons in the quantum dot is large. The complexity of this computation alone is $\mathcal{O}(n^3)$, whereas the overall complexity of the pole-based approach is $\mathcal{O}(n^{3/2})$. The crossover point in our experiment appears to be 512 electrons. For a quantum dot that contains 2,048 electrons, PCSelInv is eight times faster than Octopus.

6. Concluding remarks. We have presented an efficient parallel selected inversion algorithm for structured sparse matrices. This algorithm can be used in the large scale electronic structure analysis when it is combined with the recently developed pole-expansion techniques [27, 29]. Our algorithm computes the selected components, and, in particular, the diagonal elements of A^{-1} in parallel. Our algorithm requires necessary symbolic information from the elimination tree and the L and D factor from a block LDL^T factorization. The symbolic analysis can be done in parallel relatively easily for 2D Hamiltonians discretized on a rectangular domain by a finite difference. For problems that are defined on irregular grids (e.g., problems that are discretized by finite elements or some other techniques) and for 3D problems, the tasks of parallel symbolic factorization and relative index calculation are generally more complicated. We have developed a sequential selected inversion algorithm `SelInv`, which calculates the elements of A^{-1} corresponding to the nonzero pattern of $L + L^T$ for general sym-

metric matrix A [31]. However, we remark that a parallel implementation for general symmetric matrix is still not straightforward and will be studied in the future.

We use the elimination tree associated with the LDL^T factorization to divide the computational load and distribute the data allocated to hold the L and D factors, these distributed matrices are overwritten by the selected components of A^{-1}.

We also use the techniques of local buffering and relative indexing to ensure that the synchronization cost associated with the parallel selected inversion is minimized. A level-by-level restriction scheme is utilized to overcome the synchronization bottleneck and to achieve high performance.

We have demonstrated that our implementation of the LDL^T factorization and selected inversion calculation is very efficient. We have used our code to solve problems defined on a $65{,}535 \times 65{,}535$ grid with more than four billion degrees of freedom on 4,096 processors. The code exhibits an excellent weak scaling property.

When compared with the standard approach for evaluating the diagonal of a Fermi–Dirac function of a Kohn–Sham Hamiltonian associated with a 2D electron quantum dot, the new pole-expansion technique that uses our algorithm to compute the diagonal of $(H - z_i I)^{-1}$ for a small number of poles z_i is much faster, especially when the quantum dot contains many electrons.

Acknowledgments. We would like to thank Esmond Ng and Sherry Li for helpful discussion. We would also like to thank the anonymous referees for their careful reading and helpful comments.

REFERENCES

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

[2] P. Amestoy, I. Duff, Y. Robert, F. Rouet, and B. Ucar, *On computing inverse entries of a sparse matrix in an out-of-core environment*, SIAM J. Sci. Comput., to appear.

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

[4] C. Attaccalite, S. Moroni, P. Gori-Giorgi, and G. B. Bachelet, *Correlation energy and spin polarization in the* 2*d electron gas*, Phys. Rev. Lett., 88 (2002), p. 256601.

[5] J. R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comp., 31 (1977), pp. 163–179.

[6] J. R. Bunch and B. N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM J. Numer. Anal., 8 (1971), pp. 639–655.

[7] Y. Campbell and T. Davis, *Computing the Sparse Inverse Subset: An Inverse Multifrontal Approach*, Technical report TR-95-021, University of Florida, Gainesville, FL, 1995.

[8] A. Castro, H. Appel, M. Oliveira, C. Rozzi, X. Andrade, F. Lorenzen, M. Marques, E. Gross, and A. Rubio, *Octopus: A tool for the application of time-dependent density functional theory*, Phys. Stat. Sol. B, 243 (2006), pp. 2465–2488.

[9] C. Chevalier and F. Pellegrini, *PT-SCOTCH: A tool for efficient parallel graph ordering*, Parallel Comput., 34 (2008), pp. 318–331.

[10] S. Datta, *Electronic Transport in Mesoscopic Systems*, Cambridge University Press, West Nyack, NY, 1997.

[11] T. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2006.

[12] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Comput. Sci. Engrg., 4 (2002), pp. 90–97.

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

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

[15] E. N. Economou, *Green's Functions in Quantum Physics*, Springer-Verlag, Berlin, 2006.

[16] A. M. Erisman and W. F. Tinney, *On computing certain elements of the inverse of a sparse matrix*, Numer. Math., 18 (1975), pp. 177–179.

[17] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.
[18] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[19] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput., 9 (1986), pp. 862–874.
[20] J. L. Gustafson, *Reevaluating Amdahl's law*, Comm. ACM, 31 (1988), pp. 532–533.
[21] P. Hohenberg and W. Kohn, *Inhomogeneous electron gas*, Phys. Rev. (2), 136 (1964), pp. B864–B871.
[22] Cray Inc., *Using Cray Performance Analysis Tools*, Cray Inc., Seattle, WA, 2009.
[23] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel Distrib. Comput., 48 (1998), pp. 71–85.
[24] W. Kohn and L. J. Sham, *Self-consistent equations including exchange and correlation effects*, Phys. Rev. (2), 140 (1965), pp. A1133–A1138.
[25] 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 (2008), pp. 9408–9427.
[26] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Math. Software, 29 (2003), pp. 110–140.
[27] 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, 79 (2009), p. 115133.
[28] 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*, Commun. Math. Sci., 7 (2009), pp. 755–777.
[29] L. Lin, J. Lu, L. Ying, and W. E, *Pole-based approximation of the Fermi-Dirac function*, Chin. Ann. Math. Ser. B, 30 (2009), pp. 729–742.
[30] 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*, Technical report LBNL-2677E, Lawrence Berkeley National Laboratory, Berkeley, CA, 2009.
[31] L. Lin, C. Yang, J. Meza, J. Lu, L. Ying, and W. E, *SelInv–An algorithm for selected inversion of a sparse symmetric matrix*, ACM Trans. Math. Software, accepted.
[32] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.
[33] R. Martin, *Electronic Structure–Basic Theory and Practical Methods*, Cambridge University Press, West Nyack, NY, 2004.
[34] C. Moler, *Numerical Computing with MATLAB*, SIAM, Philadelphia, PA, 2004.
[35] S. Moore, P. Mucci, J. Dongarra, S. Shende, and A. Malony, *Performance instrumentation and measurement for terascale systems*, Lecture Notes in Comput. Sci., 2723, Springer-Verlag, Heidelberg, 2003, pp. 53–62.
[36] D. E. Petersen, S. Li, K. Stokbro, H. H. Sørensen, P. C. Hansen, S. Skelboe, and E. Darve, *A hybrid method for the parallel computation of Green's functions*, J. Comput. Phys., 228 (2009), pp. 5020–5039.
[37] A. Pothen and C. Sun, *A mapping algorithm for parallel sparse Cholesky factorization*, SIAM J. Sci. Comput., 14 (1993), pp. 1253–1257.
[38] P. Raghavan, *Efficient parallel sparse triangular solution using selective inversion*, Parallel Process. Lett., 8 (1998), pp. 29–40.
[39] P. Raghavan, *DSCPACK: Domain-Separator Codes for the Parallel Solution of Sparse Linear Systems*, Technical report CSE-02-004, The Pennsylvania State University, University Park, PA, 2002.
[40] E. Räsänen, H. Saarikoski, V. Stavrou, A. Harju, M. Puska, and R. Nieminen, *Electronic structure of rectangular quantum dots*, Phys. Rev. B, 67 (2003), p. 235307.
[41] E. Rothberg and A. Gupta, *An efficient block-oriented approach to parallel sparse Cholesky factorization*, SIAM J. Sci. Comput., 15 (1994), pp. 1413–1439.
[42] D. Skinner and W. Kramer, *Understanding the causes of performance variability in HPC workloads*, in IEEE International Symposium on Workload Characterization, IISWC05, IEEE, 2005, pp. 137–149.
[43] T. Slavova, *Parallel Triangular Solution in the Out-of-Core Multifrontal Approach for Solving Large Sparse Linear Systems*, Ph.D. thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2009.

[44] K. Takahashi, J. Fagan, and M. Chin, *Formation of a sparse bus impedance matrix and its application to short circuit study*, in 8th PICA Conference Proceedings, Minneapolis, MN, 1973.

[45] K. Teranishi, P. Raghavan, and E. Ng, *A new data-mapping scheme for latency-tolerant distributed sparse triangular solution*, in Supercomputing, ACM/IEEE 2002 Conference, 2002, paper 27.