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 § Weinan E ¶

Abstract

We present an efficient parallel algorithm and its implementation for computing the diagonal of H^{-1} where H is a 2D Kohn-Sham Hamiltonian discretized on a rectangular domain using a standard second order finite difference scheme. This type of calculation can be used to obtain an accurate approximation to the diagonal of a Fermi-Dirac function of H through a recently developed pole-expansion technique. [29]. The diagonal elements are needed in electronic structure calculations for quantum mechanical systems. [10, 23, 25]. We show how elimination tree is used to organize the parallel computation and how synchronization overhead is reduced by passing data level by level along this tree using the technique of local buffers and relative indices. We analyze the performance of our implementation by examining its load balance and communication overhead. We show that our implementation exhibits an excellent weak scaling on a large-scale high performance distributed parallel machine. When compared with standard approach for evaluating the diagonal a Fermi-Dirac function of a Kohn-Sham Hamiltonian associated 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.

1 Introduction

Electronic structure calculation based on density functional theory [10, 23, 25] has been popular in the field of chemistry and material science in recent years. In the framework of density functional theory, the electron density is represented as the diagonal of Fermi-Dirac function $f_{\beta,\mu}(t) = 2/(1 + e^{\beta(t-\mu)})$ evaluated at a Kohn-Sham Hamiltonian H, where the parameter β is proportional to the reciprocal of the temperature, μ is the chemical potential, and the number 2 comes from spin. The standard algorithm for evaluating the diagonal of $f_{\beta,\mu}(H)$ involves computing the invariant subspace associated with k smallest eigenvalues of H. These eigenvalues are known as the occupied states. The complexity of this approach is proportional to k^3. When the value of k, which is proportional to the number of electrons in the system, becomes large, this approach becomes prohibitively expensive. A lot of attention has been attracted in the quest for algorithms with better scaling. One type of such algorithms is Fermi operator expansion.

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

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

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

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

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

The Fermi operator expansion technique expands Fermi-Dirac function using polynomials [16,17] or rational functions[4]. A review article can be found in [18]. From the viewpoint of efficiency, a major concern is the representation cost of Fermi-Dirac function as a function of $\beta\Delta E$, where ΔE is the spectral width of the Hamiltonian matrix. The cost of polynomial based Fermi operator expansion scheme proposed by Goedecker *et al.* is proportional to $\beta\Delta E$. When $\beta\Delta E$ is large, rational approximation becomes preferable. Various schemes have been proposed to reduce the representation cost [7, 38, 44], among which [7] achieves the best scaling as $(\beta\Delta E)^{1/3}$.

Recently a pole expansion technique [27,29] has been developed to further reduce the representation cost to $\log(\beta\Delta E)$. This technique allows the computation of electronic structure of metallic system at room temperature with fine discretization using only a small number of poles. The pole-expansion entails expressing electron density as a linear combination of the diagonal of $(H - z_i I)^{-1}$ as

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

Here Im (H) stands for the imaginary part of H. The parameters z_i and ω_i are the complex shift and weight associated to the i-th pole respectively. z_i and ω_i can be chosen such that the total number of poles P is minimized for a given accuracy requirement.

In this paper, we consider a Kohn-Sham Hamiltonian

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

defined on a two dimensional space. Here Δ is the Laplacian operator, V_{ext} is the external potential, V_H is the Hartree potential, V_{xc} is a 2D exchange-correlation potential. For simplicity the domain is assumed to be rectangular and the boundary condition is zero Dirichlet boundary condition. Such a Hamiltonian can be used to describe, for example, the electronic properties of 2D quantum dots [41]. All the discussions can can also be generalized to quasi-2D systems [39], where the Hamiltonian is defined on a three dimensional space but with a third direction much smaller than the other two directions. We also assume here that all potentials are local, *i.e.* the potentials are functions of position $\mathbf{r}$ only. But the algorithm can also be generalized to Hamiltonians with non-local potentials. In the case of local potential, after H is discretized by a finite difference scheme, the sparsity structure of the matrix Hamiltonian A is completely determined by that associated with the discretized Laplacian operator.

Generally speaking, pole expansion requires the computation of the diagonal of A^{-1}, where A is a (real or complex) sparse symmetric matrix of extremely large dimension. Direct extraction of the diagonal of from the full matrix A^{-1} is impractical. Sequential algorithm for computing the diagonal of A^{-1} for a sparse symmetric matrix A derived from discretization of the Hamiltonian on a rectangular 2D grid has been proposed in [28]. It has been shown that the complexity of these algorithms is $\mathcal{O}(n^{3/2})$ where n is the dimension of the matrix. In this paper we describe an efficient parallel implementation of a diagonal extraction algorithm that can be used to compute the diagonal of A^{-1} for discretized 2D Kohn-Sham Hamiltonian on a large-scale distributed parallel computer with thousands of processors. We also remark that the similar type of computation also appears in other contexts. One such example is electronic transport, where the diagonals of retarded Green's function and less-than Green's function are to be calculated. Fast algorithms have also been recently developed for this purpose [26, 40].

The paper is organized as follows. We start in section 2 with main ideas of our algorithm, and explain why it is possible to extract the diagonal of A^{-1} without computing entire A^{-1}. The algorithmic and implementation details of a parallel procedure we developed for computing the diagonal of A^{-1} is 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 billions of degrees of freedom on 4,096 processor in less than 25 minutes. We show 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 [6].

Standard linear algebra notation is used for vectors and matrices throughout the paper. We use $A_{i,j}$ to denote the (i,j)-th element of A. When describing the algorithms and their implementations, we often use letters in typewriter fonts such as `A`, `L` and `D` etc. to denote matrices stored in an appropriate format using an appropriate data structure. The letters `I`, `J` and `K` often serve the dual purposes of being an integer index as well as representing a set that contains a number of integers as its members. Their meaning should be clear from the context. 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 An Algorithm for Computing the Diagonal of A^{-1}

The algorithm consists of two major steps. 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 diagonal elements of A^{-1} with a computational complexity comparable to that of the sparse block LDL^T factorization and no additional storage requirement. The main idea used here is very general. It dates back to [13], and is shared in other recently developed algorithms [26, 28, 40].

2.1 An Algorithm for Computing the Inverse of a Dense Matrix

Before we describe the algorithm for computing the diagonal of A^{-1} using the L and D matrices produced from an LDL^T factorization of A, it will be helpful to first review the major operations involved in the LDL^T factorization. Let

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

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

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

where $\ell = a/\alpha$ and $S = \hat{A} - aa^T/\alpha$ 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, which all have the form

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

where $\ell^{(1)} = \ell = a/\alpha$, yields the final L factor. The $(1,1)$ entry of each Schur complement together with α become the diagonal entries of the D matrix.

The key observation made in [26] and [28] is that A^{-1} can be expressed by

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

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

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

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

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

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

2.2 Computing Diagonal of the Inverse 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, no computation cost can be saved if we just want to extract the diagonal elements of A^{-1}. All elements of A^{-1} are needed and computed.

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

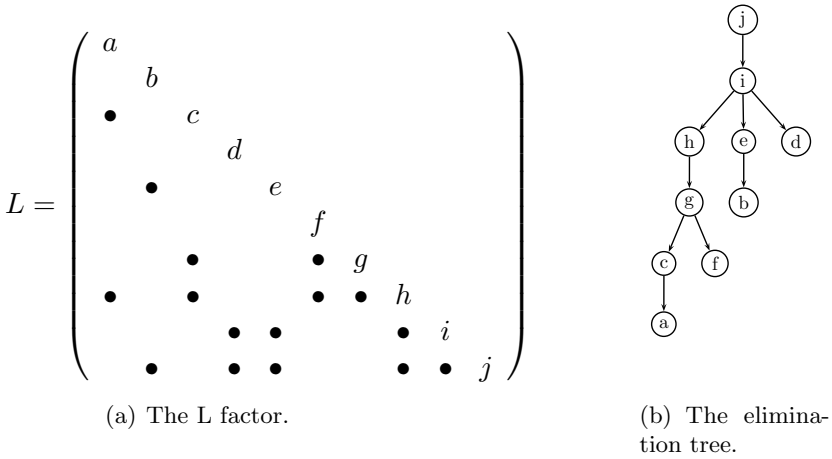
Therefore, to extract the diagonal of A^{-1}, we can simply modify the procedure shown in Algorithm 1 so that at each iteration we only compute selected elements of A^{-1} that will be needed by subsequent iterations of this procedure. We will call the second step of the diagonal extraction computation *selected inversion*. It turns out that the elements that need to be computed is 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 only need to examine the nonzero structure of the kth column of L for all $k < j$ since such a nonzero structure tells us which rows and columns of the trailing sub-block of A^{-1} are needed to complete the calculation of the (k, k) entry of A^{-1}. In particular, we would like to find out which elements in the jth column of A^{-1} are required for computing $A^{-1}_{i,k}$ for any $k < j$ and $i \geq j$.

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

These observations are well known in the sparse matrix factorization literature [12, 14]. They can be made more precise by using the notion of *elimination tree* [30]. In such a tree, each node or vertex of the tree corresponds to a column (or row) of A. Assuming A can be factored as $A = LDL^T$, a node p is the parent of a node j if and only if

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



(a) The L factor.

(b) The elimination tree.

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

If $L_{j,k} \neq 0$ and $k < j$, then the node k is a descendant of j in the elimination tree. An example of the elimination tree of a matrix A and its L factor are shown in Figure 2.2. Such a tree can be used to clearly describe the dependency among different columns in a sparse LDL^T factorization of A. In particular, it is not too difficult to show that constructing the j column of L requires contributions from all descendants of j, if the descendant corresponds to a nonzero matrix element in the jth row of L.

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

2.3 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 a block factorization of A:

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

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

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

Two advantages of using a block algorithm are:

1. It allows us to use level 3 `BLAS` (Basic Linear Algebra Subroutine) to develop an efficient implementation by exploit memory hierarchy in modern microprocessors;
2. It reduces indirect addressing overhead in sparse matrix computation.

If A is sparse, blocks are defined in terms of what is called *supernodes*. A supernode is a set of nodes whose corresponding columns (of its L factor) share the same nonzero lower diagonal structure. The definition of a supernode can be relaxed to include nodes whose corresponding columns of L are nearly identical. In this paper, we will use the relaxed definition of supernodes, which is more natural for a block row-based LDL^T factorization algorithm we choose to implement.

3 Computing the Diagonal of the Inverse of a 2D Kohn-Sham Hamiltonian

In this section, we present the algorithmic and implementation details of a parallel procedure we have developed for computing the diagonal of A^{-1} for a discretized 2D Kohn-Sham Hamiltonian defined on a rectangular domain with a Dirichlet boundary condition. We assume that a standard 5-point stencil is used to discretize the Laplacian operator. However, many of the techniques we describe here can be easily extended to other higher order stencils for both 2D and 3D Hamiltonians.

We will begin with some standard notations in section 3.1.1. A description of a sequential recursive implementation of a block row-based LDL^T factorization follows in section 3.1.2. We introduce a sequential selected inversion algorithm in section 3.1.3. These two sections serves as the building blocks for our parallel factorization algorithm. Implementation details are given in section 3.1.4. In particular, we will show how the elimination tree is used to organize the computation in an efficient manner in the factorization. We will also discuss how sparse matrix updates are managed through the use of relative indices. We will show that the elimination tree can also be used to guide the selected inversion process.

Our basic strategy for parallelization of the block row-based LDL^T factorization and the selected inversion procedures is to divide the computational work among different branches of the elimination tree, which also serves as a parallel task tree. We will describe how this division of work is defined in section 3.2.1 and discuss how data is distributed among different processors to achieve a good load balance in section 3.2.2. One of the key factors that affect the scalability of parallel computation is the synchronization cost. We will describe a technique for reducing the synchronization cost in section 3.2.3.

A unique feature of our implementation is that the symbolic analysis of the nonzero structure of L is extremely efficient and performed in parallel. We will briefly describe the major components of this procedure in section 3.2.4.

We remark that synchronization bottleneck is a particularly prominent problem in the parallel block LDL^T factorization and selected inversion algorithm. In other words, we may easily arrive at a sequential algorithm if the synchronization process is not carefully designed. Therefore many of the efforts are spent for this purpose in section 3.1.4 and 3.2.3, including recursive implementation, level-by-level updating scheme and relative indices. We recommend readers who are not interested in these technical details to skip section 3.1.4 and 3.2.3, and only read the rest parts to obtain the main idea of the parallel algorithm.

3.1 Sequential Algorithm

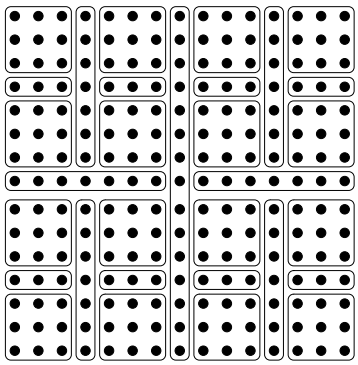
Before we present the sequential algorithms for performing an LDL^T factorization of the discretized 2D Hamiltonian and the selected inversion process, we need to introduce a number of notation and terminologies commonly used in the sparse matrix literature.

3.1.1 Nested Dissection and Elimination Tree

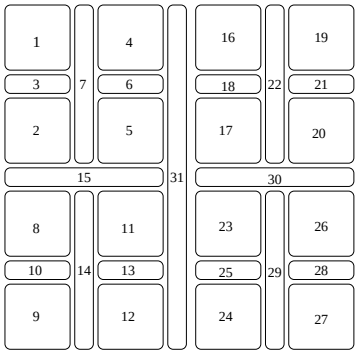
We use the technique of *nested dissection* [15] to reorder and partition the sparse 2D Hamiltonian matrix A into blocks of submatrices as shown in Figure 3(b). Because there is a one to one mapping between each row or column of the matrix and a grid point of the computational domain on which the Hamiltonian is defined, such a partition of the matrix corresponds to a recursive partition of the 2D grid into a number of subdomains with a predefined minimal size. In the example shown in Figure 2(a), this minimal size is 3×3. 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 oval boxes in Figure 2(a). For the sake of convenience, we will call a set of row or columns indices associated with either a minimal subdomain or a separator a *supernode*. We will use uppercase typewriter typeface letters such as `I` to denote such a supernode. We should note that this definition of supernode is different from the conventional definition of supernodes used in the sparse matrix literature [37], where the columns of L corresponding to a supernode are required to take exactly the same lower diagonal sparsity pattern. Our definition is more relaxed to include nodes whose corresponding columns of L are nearly identical in order to take full advantage of `BLAS3`. If the supernodes are numbered in a way as shown in Figure 2(b), the corresponding matrix exhibit a sparsity pattern shown in Figure 3(b).

The hierarchical relationship among different separators can be organized in a tree structure shown in Figure 3(a). The largest separator (supernode `31`) that divides the entire grid in half form the root the tree. The separators that divide each subgrid produced from the first division are the children of the root. The minimal subdomains that are connected by the lowest-level separators form the leaves of the tree. It turns out that this separator tree is identical to the elimination tree associated with the block LDL^T factorization of the reordered and portioned matrix A shown in Figure 3(b). Each node of the tree corresponds to a supernode. The ordering of supernodes follows a postorder traversal of the tree, i.e., a node is traversed only after all its children have been traversed.

The standard matrix-matrix multiplication is denoted by $*$. We also denote a *restricted multiplication* by $\otimes$. The meaning of restricted multiplication will become clear in section 3.1.4.



(a) Nested dissection partition and separators



(b) The ordering of the supernodes.

Figure 2: The nested dissection of a 15×15 grid and the ordering of the supernodes associated with this partition.

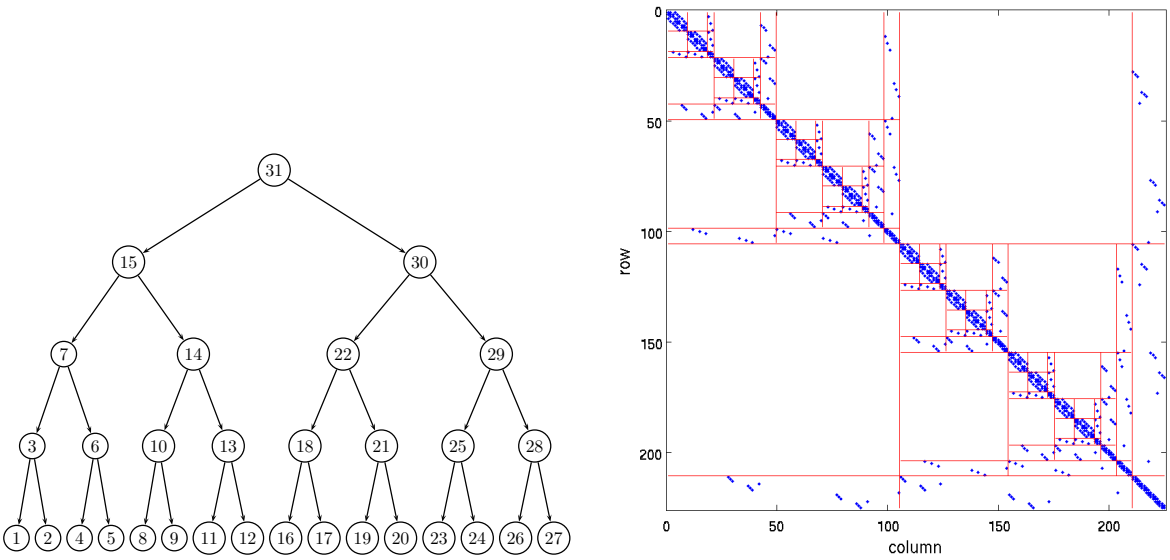
3.1.2 Block LDL^T Factorization

There are a number of ways to implement the block LDL^T factorization (5) of a sparse matrix A. They differ by the order in which rows and columns of a Schur complement are updated. The review papers [22, 36, 37, 43] contain thorough discussions on the *left-looking*, *right-looking* and *multifrontal* algorithms in which Schur complements are updated column by column. The factorization phase in the recently developed hierarchical Schur complement algorithm [28] can be regarded as an efficient right-looking implementation on a 2D rectangular grid. The LDL^T factorization we choose to implement follows a row-based approach presented in [9]. This approach is also known as the *bordering* algorithm in [31, 37]. If we ignore the sparsity structure of A and L for the moment and assume that A and L have been partitioned into `nblocks` block rows and columns indexed by `J`, a block row-based factorization algorithm can be succinctly described by the pseudocode shown in Algorithm 2. Note that in the pseudocode, the `L` matrix is constructed one block row at a time by solving a set of triangular systems (steps 1 and 2 of the algorithm) at each iteration. The index `J` should be interpreted as both a block row or column index and a set of rows and columns within the block indexed by `J`. This dual-purpose notation will be used throughout this paper.

Algorithm 2 A row-based LDL^T factorization of a symmetric matrix A.

for `J = 1` to `nblocks` **do**
  Solve `Y*L(1:J-1,1:J-1)`^{T} `= A(J,1:J-1)` for `Y`;
  Solve `L(J,1:J-1)*D(1:J-1,1:J-1) = Y` for `L(J,1:J-1)`;
  `L(J,J)` $\leftarrow$ identity matrix;
  `D(J,J)` $\leftarrow$ `A(J,J) - L(J,1:J-1)*Y`^{T};
end for

When A is sparse, we should exploit the sparsity structure of L in order to minimize the number of operations. It is well known in the sparse matrix literature that, at each iteration `J`, the sparsity pattern of `Y` and `L(J,1:J-1)` is completely determined by the nonzero structure of the descendants of node `J` in the elimination tree. To be specific, `Y(I)` (or `L(I,J)`) is nonzero if and only if node `I` is a descendent of



(a) The separator (or elimination) tree in postorder

(b) The reordered and partitioned matrix A

Figure 3: The separator tree associated with the nested dissection of the 15×15 grid shown in Figure 2(a) can also be viewed as the supernode elimination tree associated with the LDL^T factorization of the 2D Laplacian defined on that grid.

J in the elimination tree. Furthermore, the way in which Y(I) is computed can be expressed in terms of a traversal of the sub-tree rooted at I [30]. As a result, step 1 of the factorization procedure given in Algorithm 2 can be carried out efficiently by following the pseudocode given in Algorithm 3.

Algorithm 3 A triangular solve algorithm guided by an elimination tree.

```
Y ← A(1:J-1,J);
for I ∈ {Descendants of J} do
   for K ∈ {Descendants of I} do
      Y(I) ← Y(I) − L(I,K) ∗ Y(K);
   end for
end for
```

As we can see from Algorithm 3, Y(I) can be computed only if Y(K) have been calculated for all K that are descendants of I. Hence, to compute Y(I), the descendants of node I must be traversed in a proper order. The postorder [8] of the nodes in the elimination tree shown in Figure 2.2 satisfies this ordering requirement. Moreover, such an order allows us to implement the pseudocode in Algorithm 3 in a recursive fashion, as we will show in section 3.1.4.

3.1.3 Block Selected Inversion

The dependency among different supernodes exhibited by the elimination tree can also be exploited to perform the selected inversion procedure (in Algorithm 1) efficiently for a sparse matrix. A pseudocode for computing selected blocks of A^{-1} required for extracting the diagonal of A^{-1} is shown in Algorithm 4. As we can see from this pseudocode that Ainv(J,K) is calculated if and only if L(J,K) is a non-zero block. Such a calculation makes use of previously calculated blocks Ainv(J,I), where both J and I are

ancestors of the supernode K. However, unlike the LDL^T factorization procedure in which descendants of a supernode must be traversed in a proper order, ancestors of the supernode K can be accessed in any order. This feature makes the implementation of the selected inversion process somewhat easier as we will see in section 3.1.4.

Algorithm 4 A selected inversion algorithm for a sparse symmetric matrix A given its block LDL^T factorization $A = LDL^T$.

```
for K = nblocks to 1 step -1 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)^T;
  for J ∈ {ancestors of K} do
    Ainv(K,K) ← Ainv(K,K) − L(J,K)^T * Ainv(J,K);
  end for
end for
```

3.1.4 Implementation Details of the Sequential Algorithm

We now discuss several implementation details that are key to achieving high performance in the LDL^T factorization and selected inversion computation.

We will first show that the row-based LDL^T factorization we presented in section 3.1.2 can be implemented by using recursion. The use of recursion simplifies the coding effort significantly without sacrificing any performance, as we will see in section 4. In our recursive implementation, two recursive subroutines are created to handle the inner and outer loops of the pseudocode shown in Algorithm 3 respectively. To compute the Jth block row of L, we first call the recursive subroutine `SeqAssemble` to obtain Y such that `Y*L(1:J-1,1:J-1)^T = A(J,1:J-1)`. To distinguish sequential algorithms from the parallel algorithms to be introduced later, we use the prefix `Seq` in subroutine names associated with sequential algorithms and the prefix `Par` in those that are associated with parallel algorithms. Algorithm 5 shows how `SeqAssemble` is used in the main LDL^T factorization program that computes one block row of L and D at a time. To illustrate the recursive nature of our algorithm, all pseudocodes described in the next few sections are written in the form of `subroutines`. The matrix inputs to these subroutines are usually clear from the context. Therefore, we do not list them explicitly to save space. In our implementation, all matrices and work arrays are passed in by reference (or by address) according to the `FORTRAN` convention so that the stack overhead associated with recursion is relatively small. We list supernode indices as the input arguments to some of the subroutines to illustrate the path of recursion along the elimination tree.

The pseudocode given in the left column of Algorithm 6 shows that, at the Jth iteration, the `SeqAssemble` subroutine traverses down the subtree rooted at J in postorder until the first argument I becomes a leaf node at which `Y(I)` is simply `A(I,J)`. Once all descendants of the node I have been traversed, indicating that `Y(K)` is available for all descendants K of I, we use a second subroutine `SeqSum` (shown in the right box in Algorithm 6) to perform the summation required in the inner loop of the triangular solve shown in Algorithm 3. For each descendent I of J, the `SeqSum` subroutine

Algorithm 5 The major steps of a recursive implementation of the block row-based LDL^T factorization algorithm for a symmetric matrix `A`.

```
subroutine SeqBlockLDLT
for J = 1 to nblocks do
  Y ← A(J,1:J-1);
  Update Y by calling SeqAssemble(J,J);
  D(J,J) ← Y(J);
  Update L(J,1:J) and D(J,J) by calling SeqInv(J);
end for
return L, D;
end subroutine
```

traverses the subtree rooted at `I` again to collect the matrices `Y(K)` that have already been computed. It accumulates the product of `Y(K)` and `L(I,K)` in a hierarchical fashion in a workspace denoted by `Buffer` in Algorithm 6. The reason why we use a workspace to accumulate all contributions from the descendants of `I` instead of applying updates to `Y(I)` directly will become clear later in this section.

Algorithm 6 Recursive implementation of the triangular solve shown in Algorithm 3.

```
subroutine SeqAssemble(I,J)
if (I is a leaf node) return;
for K = {children of I} do
  Update Y by calling SeqAssemble(K,J) recursively;
end for
Buffer = 0;
Update Buffer by calling SeqSum(Buffer,I,I,J);
Y(I) ← Y(I) − transpose(Buffer);
return Y;
end subroutine

subroutine SeqInv(J)
D(J,J) ← D(J,J)^{-1};
for I ∈ {descendants of J} do
  L(J,I) ← Y(I)*D(J,J);
end for
return L, D;
end subroutine

subroutine SeqSum(Buffer,K,I,J)
if (K is not a leaf node) then
  for C = {children of K} do
    [RelI,RelJ] =
    GetRelIndex(C,K,I,J);
    Update part of Buffer by calling
    SeqSum(Buffer(RelI,RelJ),C,I,J)
    recursively;
  end for
end if
if ( K ≠ I ) then
  Buffer = Buffer + L(I,K) ⊗ Y(K);
end if
return Buffer;
end subroutine
```

To complete the computation of the `J`th block row of L and D, we need to invert the diagonal block `D(J,J)`, and multiply $\mathtt{D(J,J)}^{-1}$ with `Y(I)` for all descendants `I` of `J` as shown in the `SeqInv` subroutine in Algorithm 6.

Up to this point, we have treated the matrix blocks `L(J,I)` as if they are dense matrices. For the 2D Hamiltonian we consider in this paper, a large number of the off-diagonal matrix blocks in L are quite sparse. For example, in Figure 4, which shows the sparsity pattern of the L factor associated with a 2D Hamiltonian defined on a 7×7 grid as well as the block structure obtained from a 2-level nested dissection reordering, the `L(7,1)` block has dimension 7×9, but contains only 6 nonzero entries at its upper right corner. In order to carry out the summation process in the `SeqSum` subroutine efficiently, we must take advantage of these sparsity structures, which can be predetermined by a symbolic analysis procedure to be discussed in section 3.2.4. In particular, we should not store the zero rows and columns

in L(I,K) and Y(K), and we should exclude these rows and columns when these two matrices are multiplied. As a result, the product of the nonzero rows and columns of these matrices will have a smaller dimension, and it needs to be placed at a proper location in the destination workspace. We will call the multiplication of the nonzero rows and columns of L(I,K) and Y(K) a *restricted* matrix-matrix multiplication, denoted by $\otimes$.

To implement such a restricted matrix-matrix multiplication, we need to keep track of the size and the starting location of nonzero row and column indices of L(J,I) for all J and I. To reduce the bookkeeping overhead, we treat some of the zero entries in L(J,I) as nonzeros and store them explicitly. For example, in Figure 4, the nonzero entries appear within the 3×3 submatrix at the upper right corner of the L(7,1) block. Even though the lower triangular part of this 3×3 submatrix is completely zero, we store the entire submatrix and record its starting location within L(7,1).

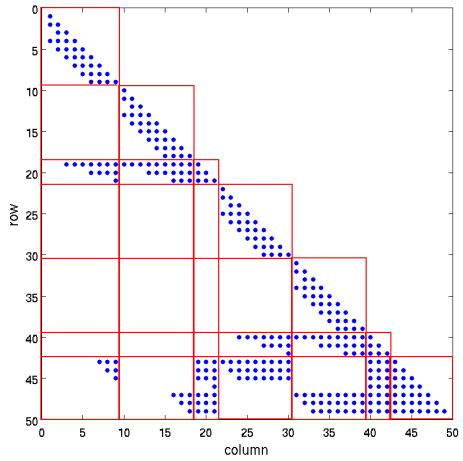


Figure 4: Sparsity pattern of the L factor associated with a Hamiltonian discretized on a 7×7 grid.

Once the positions and sizes of the non-zero submatrices within L(I,K) and Y(K) are known, the size of the product of these submatrices and its location within Y(I) can be determined for all ancestors I of K. For example, it is easy to see from Figure 4 that the product of the nonzero submatrices of L(7,1) and Y(1) is a 3×3 matrix, and it should be accumulated at the upper left corner of Y(1) with the starting row and column positions of $(1, 1)$. We call such type of row and column position indices *absolute indices*. Using these absolute indices, we can in fact perform the summation in the inner loop of Algorithm 3 by adding the product of the nonzero submatrices of L(I,K) and Y(K) directly to the destination location in Y(I). However, such a strategy creates a synchronization bottleneck in the parallel LDL^T factorization algorithm to be discussed in section 3.2. To overcome this problem, we developed a hierarchical scheme for accumulating the product of L(I,K) and Y(K) as the descendants of node I are traversed by the SeqSum subroutine in Algorithm 6. We sketch the basic ideas of this scheme here and will explain how it helps to eliminate the synchronization bottleneck in a parallel LDL^T factorization in the next section.

As illustrated in Algorithm 6, a work array Buffer, which has the same size as the number of nonzero rows and columns in Y(I), is created when node I is traversed by the SeqAssemble subroutine. This array is used to accumulate all L(I,K)*Y(K) contributions from the descendants of I. Because the multiplication of the nonzero submatrices of L(I,K) and Y(K) often produces a matrix with a much

smaller dimension (than that of the `Buffer` array), a subblock of `Buffer` is passed down from a parent to its child recursively as descendants of `I` are traversed by the `SeqSum` subroutine. The reason that it is sufficient to pass a subblock of the `Buffer` array from a parent to its child (instead of passing a subblock of `Buffer` directly from node `I` to its descendants) is that the set of nonzero rows and columns associated with `L(I,R)*Y(R)` is a subset of the nonzero rows and columns that are associated with `L(I,K)*Y(K)` if `R` is a child of `K`. Hence, when the product of the nonzero rows and columns of `L(I,R)` and `Y(R)` is accumulated in the `Buffer` array, we only need to know the relative position of this product in the subblock of the `Buffer` array owned by its parent `K`. The row and column indices that define such a position are called *relative indices*. The use of relative indices is not new. Similar ideas date back to [46], and they are also used in both left-looking and multifront algorithms [11, 32, 37].

Restricted matrix-matrix multiplication must also be used to achieve high performance in the selected inversion process. 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. These rows and columns are placed in a `Buffer` array in Algorithm 7. The `Buffer` array is then multiplied with the corresponding nonzero columns of `L(I,K)`. However, because the ancestors of the node `K` in Algorithm 4 do not have to be visited in a hierarchical order, the use of relative indices is not necessary. Absolute indices, which are obtained by the `GetAbsIndex` function in Algorithm 7, can be used to retrieve the desire submatrix block from `Ainv(J,I)` in a sequential algorithm. This kind of restricted retrieval of `Ainv(J,I)` will become more complicated when the selected inversion of A is carried out in parallel. In that case, both absolute and relative indices need to be used.

Algorithm 7 Selected inversion of `A` with restricted matrix-matrix multiplication given its block LDL^T factorization.

```
subroutine SeqSelInverse
for K = nblocks to 1 step -1 do
  for J ∈ {ancestors of K} do
    Ainv(J,K) ← 0;
    for I ∈ {ancestors of K} do
      [IA,JA] ← GetAbsIndex(L,K,I,J);
      Buffer ← selected rows and columns of Ainv(J,I) starting at (IA, JA)
      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
```

It should be noted that we use `A, L, D, Ainv, Y` as separate variables in Algorithms 5 and 7 only for clarity. In our implementation, `A, L, D, Y` share the same storage space allocated in advance. Moreover, `L` is incrementally overwritten by `Ainv`. Therefore, the memory requirement of our implementation is roughly what is needed to store `L` in a block compressed column format plus a few extra arrays of limited size that are used as workspace (such as `Buffer`).

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 2047×2047 grid. The entire computation, which involves more than 4 million degrees, took less than 2 minutes on an AMD Opteron processor.

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

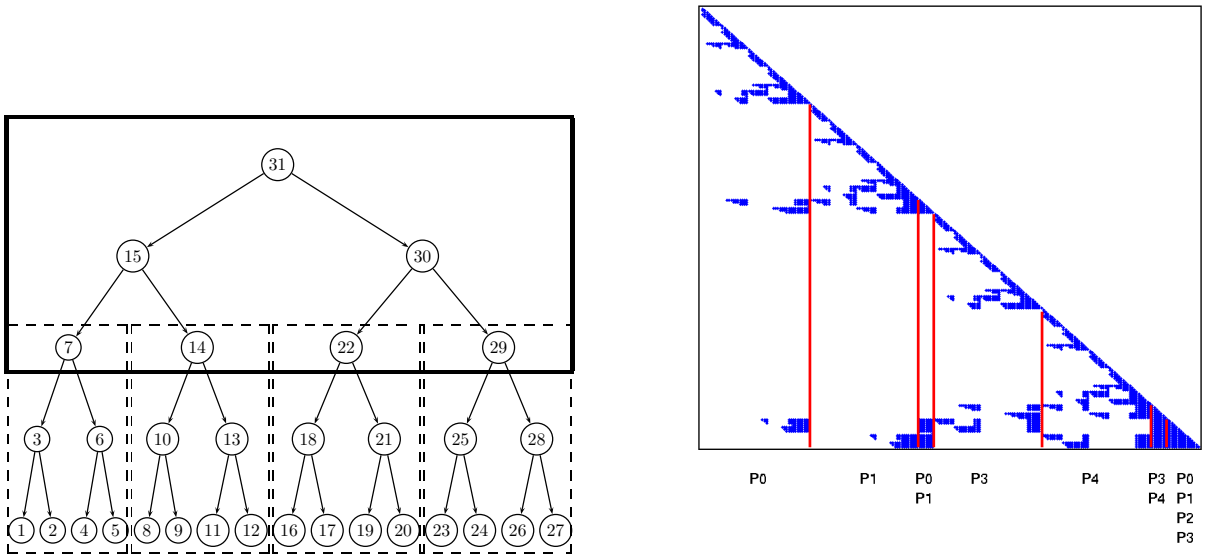
3.2.1 Task Parallelism

The elimination tree associated with the 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*. 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. Our parallel algorithm requires the number of processors p to be a power of two, and ℓ is set to $\log_2(p) + 1$.

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. To be specific, 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`.

Because each processor works on a particular branch of the tree, the recursive procedure shown in Algorithm 6 must be modified so that the `SeqAssemble` and `SeqSum` subroutines do not traverse the entire tree on each processor. In terms of parallelism, it will be convenient to think of the subtree rooted at a level-ℓ node as an *aggregated* leaf node. Figure 5(a) shows how different nodes in an elimination tree are mapped to four processors used in a parallel calculation. The subtree rooted at the 3rd level nodes are enclosed by dashed line boxes to indicate that they are aggregated leaf nodes of the parallel task tree which is enclosed by the solid line box.

The main structures of the parallel block LDL^T factorization and selected inversion procedures are shown in Algorithm 8. Both parallel procedures make use of the sequential algorithms presented earlier to perform necessary calculations that are limited to an aggregated leaf node. In the case of LDL^T factorization, each processor first uses Algorithm 5 to factor a diagonal block of A associated with an aggregate leaf node `J`. This computation is completely independent from similar calculations performed simultaneously by other processors. As each processor moves from an aggregated leaf node towards the root, which is indicated by the `while` loop in the left column of Algorithm 8, it uses the subroutine `ParAssemble` (which is the parallel analog to `SeqAssemble`) to solve `Y*L(1:J-1,1:J-1)=A(J,1:J-1)` for `Y` in parallel. As we can see from the left column of Algorithm 9, the execution of `ParAssemble` by each processor is recursive also. Each processor simply uses `mypid` and `procmap` to select an appropriate branch to move down the parallel task tree. For each node `I` along that branch, processors that belong to `procmap(I)` all contribute to the parallel computation of `Y(I)`, which requires communications among



(a) Parallel task tree

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

Figure 5: Task parallelism expressed in terms of parallel task tree and correponding matrix to processor mapping.

these processors. Eventually, each processor will reach a leaf node `I`. When that happens, `ParAssemble` will call `SeqAssemble` to traverse into the subtree rooted at `I` in postorder to compute `Y(K)` for all `K` that are descendants of `I`, just like how it is done in the sequential algorithm. No inter-processor communication is needed in `SeqAssemble`.

Once all descendants of `I` along a particular branch of the parallel task tree have been traversed, including those that are within the aggregated leaf, we use a new subroutine `ParSum` in Algorithm 9 to add up the `L(I,K)*Y(K)` contributions from all descendants `K` of `I`. The `ParSum` subroutine also moves down a branch of the parallel task tree on each processor until it reaches an aggregated leaf node where the `SeqSum` subroutine is called to sum up `L(I,K)*Y(K)` for all `K` that are in the aggregated leaf. As the `ParSum` recursion backtracks towards node `I`, it merges contributions from different branches of the parallel task tree in a hierarchical fashion. Communication is required in the merging process which must be implemented with care to minimizing the synchronization cost. We will discuss the implementation detail in section 3.2.3.

The parallelization of the selected inversion process can also 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 10, each processor moves from the root of the task tree down towards to 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 $\mathtt{L}(\mathtt{J},\mathtt{K}) \neq 0$. 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 10. Before moving one step further along `mybranch`, all processors belonging to `procmap(K)` performs some additional data redistribution by calling the subroutine `ParRestrict` as in the right column of Algorithm 10, so that selected components of `Ainv(J,K)` that will be needed in the calculation of `Ainv(J,I)` for all descendents `I` of `K` are placed at appropriate locations in the `Buffer` arrays created for each child of `K`. This step is essential for reducing

synchronization overhead and will be disucssed in detail in section 3.2.3. 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 7) to compute `Ainv(J,I)` for all descendents `I` of `K`. No inter-processor communication is required from this point on.

Algorithm 8 Parallel algorithm from extracting selected elements of the inverse of a symmetric matrix A.

subroutine `ParBlockLDLT`
Set `J` to the aggregated leaf node such that `procmap(J) = mypid`;
Call sequential algorithm for the LDL^T factorization of the subtree with root node `J`;
`J = parent(J);`
while (`J` $\neq$ parent(`root`)) **do**
  Update `Y` by calling `ParAssemble(Y,J,J);`
  `D(J,J) = Y(J);`
  Update `L(J,1:J)` and `D(J,J)` by calling `ParInv(D(J,J));`
  `J` = parent(`J`);
end while
return `L, D`;
end subroutine

subroutine `ParSelInverse`
`K` $\leftarrow$ `root`;
while (`K` is not an aggregated leaf node) **do**
  Update `Ainv(K,K:nblocks)` by calling `ParExtract(K);`
  Update Buffer by calling `ParRestrict(K);`
  `K` $\leftarrow$ `child(K)` along `mybranch`;
end while
Call Sequential algorithm to obtain `Ainv` at the leaf node;
return `Ainv`;
end subroutine

3.2.2 Data Distribution

Although it is not mentioned explicitly, one can see from Figure 5(b) that the `L` and `D` factors in the parallel algorithms we presented in Algorithm 9 and Algorithm 10 are distributed among different processors.

Because the computation required to obtain `L(I,J)` and `D(J,J)` for any `J` in an aggregated leaf node can be performed independently by the processor to which `J` is assigned, it is clear how these blocks should be distributed. Again, we do not store the entire submatrix, but only the nonzero subblock within this submatrix as well as the starting location of the nonzero subblock.

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 to this node, i.e. `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)`. 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, 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 gigabytes (GB). As we will see in section 4, the distribution scheme we use in our parallel algorithm leads to only a mild increase of memory usage per processor as we increase the problem size and the number of processors in proportion.

One of the key factors that affect the performance of parallel computing is load balance. In order to achieve that, we use a 2D block cyclic mapping consistent with that used by `ScaLAPACK` to distribute

Algorithm 9 Parallel recursive implementation of the triangular solve shown in Algorithm 3.

```
subroutine ParAssemble(I,J)
if (mypid ∈ procmap(I)) then
  if (I is not an aggregated leaf node) then
    for K ∈ {children of I} do
      Update Y in parallel by calling
      ParAssemble(K,J) recursively;
    end for
    Update Buffer in parallel by calling
    ParSum(Buffer,I,I,J);
  else
    Update Y by calling SeqAssemble(I, J);
    Update Buffer by calling
    SeqSum(Buffer,I,I,J);
  end if
  Y(I) ← Y(I) - transpose(Buffer);
end if
return Y;
end subroutine


subroutine ParInv(J)
D(J,J) ← D(J,J)^{-1};
for I ∈ {Descendants of J} do
  L(J,I) ← Y(I) * D(J,J);
end for
return L, D;
end subroutine
```

```
subroutine ParSum(Buffer,K,I,J)
if (mypid ∈ procmap(K)) then
  if (K is not an aggregated leaf node) then
    allocate CBuffer{C} for each child C of K;
    for C ∈ {children of K} do
      Update CBuffer{C} by calling
      ParSum(CBuffer{C},C,I,J) recursively.
    end for
    for C ∈ {Children of K} do
      [IR,JR] ← GetRelIndex(C,K,I,J);
      Merge CBuffer{C} into Buffer(IR,JR) us-
      ing PDGEMR2D;
    end for
  else
    for C ∈ {children of K} do
      [IR,JR] ← GetRelIndex(C,K,I,J);
      Update sequential part of Buffer by call-
      ing SeqSum(Buffer(IR,JR),C,I,J) recur-
      sively;
    end for
  end if
  if (K ≠ I) then
    Buffer = Buffer + L(I, K) ⊗ Y(K);
  end if
end if
return Buffer;
end subroutine
```

Algorithm 10 Parallel implementation of selected inversion of A given its block LDL^T factorization.

```
subroutine ParExtract(K)
for J ∈ {ancestors of K} do
  Ainv(J,K) ← 0;
  for I ∈ {ancestors of K} do
    Ainv(J,K) ← Ainv(J,K) − Buffer(J,I) ⊗ L(I,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);
end for
return Ainv;
end subroutine
```

```
subroutine ParRestrict(K)
if (K is the root) then
  Buffer ← D(K,K);
end if
for C ∈ {children of K} do
  for all I,J ∈ {ancestors of K} do
    if L(J,C) ≠ 0 and L(I,C) ≠ 0) then
      [IR,JR] ∈ GetRelIndex(C,K,I,J);
      Restrict Buffer(J,I) to a submatrix start-
      ing at (IR, JR).
    end if
  end for
end for
return Buffer;
end subroutine
```

the nonzero blocks of L(I,J) and D(J,J) for any J that is an ancestor of an aggregated leaf node. For example, if the dense submatrix D(7,7) is distributed among four processors arranged as a 2×2 processor grid labeled by $P_{0,0}$, $P_{0,1}$, $P_{1,0}$ and $P_{1,1}$, and if D(7,7) is partitioned into many $m_b \times n_b$ blocks (assuming m_b and n_b divide the number of rows and columns of D(7,7) respectively), then the (i,j)-th block of D(7,7) is stored on processor $P_{i \bmod 2,\ j \bmod 2}$.

The use of 2D block cyclic distribution is key to achieving scalable performance in ScaLAPACK [5]. The purpose of dividing D(J,J) into many $m_b \times n_b$ blocks is to make the BLAS operations required in the summation and inversion processes more efficient by taking advantage of the memory hierarchy of modern processors. However, we should mention that there is a trade-off between the efficiency of BLAS operation and load balance of the computation. Although choosing a larger m_b or n_b value may improve the performance of BLAS operations on some processors, it may cause load imbalance and degrade the overall performance of the computation.

3.2.3 Implementation Details of the Parallel Algorithm

We left out a number of details in section 3.2.1 in order to simplify the description of our parallel algorithm. One such detail is how the restricted matrix-matrix multiplication L(I,K)$\otimes$ Y(K) is carried out in parallel, and how this product is accumulated in Y(I).

Because the nonzero subblocks of Y(I,K) and Y(K) are distributed among the same set of processors (within a processor group), the multiplication of these subblocks can be easily carried out by calling the ScaLAPACK subroutine pdgemm.

However, since Y(I) (where products of Y(I,K) and Y(K) are to be accumulated for all descendants of I) is distributed among a larger number of processors, we need to redistribute Y(I,K)*Y(K) among these processors before it can be added to Y(I). We use the ScaLAPACK utility subroutine PDGEMR2D to accomplish this redistribution task. When PDGEMR2D is called to redistribute data from a processor group A to a larger processor group B that contains all processors in A, all processors in B are *blocked*, meaning that no processor in B can proceed with its own computational work until the data redistribution initiated by processors in A is completed.

This blocking feature of PDGEMR2D, while necessary in ensuring data redistribution is done in a coherent fashion, creates a potential synchronization bottleneck. To be specific, when the L(I,K)*Y(K) contributions from several branches of the parallel task tree are redistributed and accumulated at Y(I), only one branch can proceed at a time while others must wait. Hence, this direct updating scheme makes the inner loop of triangular solve in Algorithm 3 a more or less sequential process.

To reduce the synchronization cost, we make use of the Buffer array already discussed in section 3.1.4 to hold the distributed product of L(I,K) and Y(K) and pass it one level up at a time from a group of processors assigned to a child node K to a larger group of processors assigned to its parent, say R, so that a sufficient amount of parallelism is maintained at levels close to the aggregated leafs of the parallel task tree. The recursive nature of the ParSum and SeqSum subroutines makes the implementation of such an *indirect* level-by-level updating scheme quite natural. The redistributed product L(I,K)*Y(K) is added to the work array used to hold L(I,R)*Y(R) before the sum is redistributed and passed one level further towards the root of the parallel task tree.

Again, because the nonzero rows and columns in L(I,K) and Y(K) are a subset of the nonzero rows and columns in L(I,R) and Y(R) if R is a child of K, we only need to know the relative indices of the nonzero subblock of L(I,K)*Y(K) within that of L(I,R)*Y(R) when L(I,K)*Y(K) is redistributed in a work array that can be added directly to the Buffer array that holds the distributed nonzero subblock of L(I,R)*Y(R).

A similar synchronization issue arises in the selected inversion process when the selected non-zero

rows and columns in `Ainv(J,I)` (Algorithm 10) are extracted from a large number of processors in `procmap(I)` and redistributed among a subset of processors in `procmap(K)`. If `K` is several levels away from `I`, a direct extraction and redistribution process will block all processors in `procmap(I)` that are not in `procmap(K)`, thereby making the computation of `Ainv(J,K)` a sequential process for all descents (`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. Selected subblocks of the `Buffer` array will be passed further to the descendents of `I` 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 10. At a particular node `I`, the `ParRestrict` call is made simultaneously by all processors in `procmap(I)`, and the `Buffer` array is distributed among processors assigned to each child of `I` so that the multiplication of the nonzero subblocks of `Ainv(J,I)` and `L(J,K)` can be carried out in parallel (by `pdgemm`). Because this distributed `Buffer` array contains all information that would be needed by descendents of `K`, no more direct reference to `Ainv(J,I)` is required for any ancestor `I` of `K` from this point on. As a result, no communication is performed between processors that are assigned to different children of `I` once `ParRestrict` is called at node `I`.

As each processor moves down the parallel task tree within the `while` loop of the subroutine `ParSelInverse` in Algorithm 9, the amount of data extracted from the `Buffer` array by the `ParRestrict` subroutine becomes smaller and smaller. The newly extracted data is distributed among a smaller number of processors also. Each call to `ParRestrict(I)` requires a synchronization of all processors in `procmap(I)`, hence incurs 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.

Since the desired data in the `Buffer` array is passed level by level from a parent to its children, we only need to know the relative positions of the subblocks needed by a child within the `Buffer` array owned by its parent. Hence, relative indices which are obtained by the subroutine `GetRelIndex` in Algorithm 10, are used for data extraction in `ParRestrict`. As we mentioned earlier, the use of relative indices is not necessary when each process reaches a leaf node at which the sequential selected inversion subroutine `SeqSelInverse` is called.

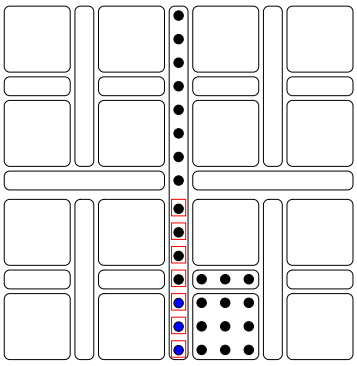
3.2.4 Parallel Symbolic Analysis

An efficient implementation of the parallel diagonal extraction algorithm we presented in section 3.1 relies heavily on a number of nonzero structure properties of the L factor that must be determined in advance. In particular, we need to identify all nonzero blocks of L, where blocks are defined in terms of "relaxed" supernodes introduced in section 3.1.1. Within each nonzero block, we need to identify the location and size of the nonzero subblock that contains nonzero rows and columns of `L(I,J)` only. The process of determining these structure properties of L is often referred to as symbolic analysis or factorization.

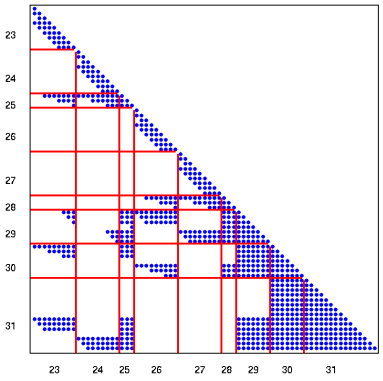
The symbolic analysis of L is usually done by examining the adjacency graph associated with the original sparse matrix A and the corresponding elimination tree. In particular, a nonzero fill in L can be viewed as the formation of a new edge after some vertex of the adjacency graph is removed [14, 42] (as a result of performing one step of elimination).

To give a flavor of how symbolic analysis is performed in our implementation. Let us look at

the nonzero structure of the L(31,24) block which has a dimension of 15×9. (The dimension can be determined from the sizes of supernodes 31 and 24 respectively.) It is clear from Figure 6(a) that supernode 24 is adjacent to the last three nodes (marked in blue) of supernode 31. By treating all nodes within supernode 24 as a single entity, we do not consider the order in which these nodes are eliminated in the factorization process. As a result, all nine nodes within supernode 24 are viewed as being adjacent to the last three nodes of supernode 31 when supernode 24 is being eliminated. Consequently, we can deduce that the last three rows of L(31,24) forms a nonzero block with a dimension of 3×9. This can be clearly seen from Figure 6(b) in which the supernode partition and the nonzero structure of L(23:31,23:31) are shown.



(a) The adjacency relationships among supernodes 24, 25 and 31 on the 2D grid. The supernode index follows that in Figure 2(b).



(b) The nonzero structure of L(23:31,23:31)

Figure 6: The nonzero structure of the L(31,24) block in (b) as well as the absolute and relative positions of the nonzero contribution $\mathtt{L}(31,24) \otimes \mathtt{Y}(24)$ in the work space created for $\mathtt{L}(31,25) \otimes \mathtt{Y}(25)$ and Y(31) can be determined by examing the geometry relationship among supernodes 24, 25 and 31.

The nonzero block in L(31,24) is used to update Y(31) in the inner loop of Algorithm 3 when the loop (or supernode) index K becomes 24. Because Y(24) has the same nonzero structure as $\mathtt{L}(31,24)^T$, the product of L(31,24) and Y(24), which has a dimension of 15×15, contains a 3×3 nonzero block starting from the 12th row and column. The absolute indices $(12, 12)$ can be easily determined from the position of the first node (within supernode 31) that is adjacent to supernode 24, as we can clearly see in Figure 6(a). (We assume the nodes with supernode 31 are numbered in an ascending order from the top to the bottom.)

However, for reasons that we have explained earlier, the product of L(31,24) and Y(24) is not added directly to the work space allocated for Y(31). Instead, it is accumulated in a buffer array allocated for its parent supernode 25, which is also used to hold the product of L(31,25) and Y(25). The dimension of this buffer is 7×7 because supernode 25 becomes adjacent to the last 7 nodes (shown as nodes enclosed by small red square boxes) within supernode 31 after both supernode 23 and 24 are eliminated. The relative position of $\mathtt{L}(31,24) \otimes \mathtt{Y}(24)$ in the buffer space allocated for $\mathtt{L}(31,25) \otimes \mathtt{Y}(25)$ is $(4, 4)$. The relative row and column index 4 can be calculated from the difference between the absolute positions of the nonzero blocks $\mathtt{L}(31,25) \otimes \mathtt{Y}(25)$ and $\mathtt{L}(31,24) \otimes \mathtt{Y}(24)$ within Y(31), i.e., $4 = (12 - 9) + 1$.

In many direct sparse matrix solvers, the symbolic analysis of A is often performed on a single

processor. However, for the size of the problem we would like to work with, a sequential symbolic analysis will be too slow and too memory consuming. Parallel symbolic factorization based on a parallel generation of the elimination tree is discussed in [48]. The explict use of grid geometry allows us to simplify the parallelization of symbolic factorization. In our implementation, each processor analyzes a portion of the grid domain to which it is assigned as well as at most four separators it borders with.

4 Performance

In this section, we report the performance of our implementation of the algorithm developed to compute the diagonal of A^{-1} for a discretized 2D Kohn-Sham Hamiltonian H with zero complex shift. 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 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 per second (Gflops) per core. Each compute node has 8 gigabyte (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 MPI. The floating point calculation is done in 64-bit double precision. We use 32-bit integers to keep index and size information.

Our objective for developing a parallel algorithm and its implementation to compute the diagonal of A^{-1} is to enable us and other researchers to study the electronic structure of large quantum mechanical systems when a vast amount of computational resource is 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 [21]. 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 $\mathcal{O}(n^{3/2})$ or $\mathcal{O}(m^3)$, where n is the matrix dimension and m is the number of grids in one dimension. We will simply call m the *grid size* in the following. Clearly $n = m^2$. 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 [47], the `CrayPat` performance analysis tool [24] as well as `PAPI` [35] to measure various performance characteristics of our implementation.

4.1 Single Processor Performance

In this section, we first report the performance of the computation used to obtained the diagonal of A^{-1} when it is executed on a single processor. The single processor performance is measured in terms of the CPU time and the floating point operations performed per second (flops). Table 1 lists the performance characteristic of single processor calculations for Hamiltonians defined on square grids with different sizes. We choose the grid size m to be $m = 2^\ell - 1$ for some integer $\ell > 1$ so that a perfectly balanced elimination tree is produced from a nested dissection of the computational domain.

The largest problem we can solve on a single processor contains $2,047 \times 2,047$ grid points. The dimension of the corresponding matrix is over 4 million. The memory requirement for solving problems

grid size	matrix dimension	symbolic	factorization	inversion	total	Gflops
127	16,129	0.01	0.04	0.03	0.08	1.29
255	65,025	0.05	0.21	0.17	0.43	2.57
511	261,121	0.22	1.18	1.03	2.43	3.89
1023	1,046,529	0.93	7.29	6.76	15.0	5.12
2047	4,190,209	4.21	48.8	47.3	100.3	6.15

Table 1: Single processor performance

defined on a larger grid (with $\ell > 11$) exceeds what is available on a single node of the Franklin system. Thus they can only be solved in parallel using multiple processors.

We can clearly see from Table 1 that the symbolic analysis of the LDL^T factorization takes a small fraction of the total time, especially when the problem is sufficiently large. The selected inversion calculation (after a block LDL^T factorization has been performed) takes slightly less time to complete than that required by the factorization. The total CPU time listed in the 6th column of the table confirms the $\mathcal{O}(n^{3/2})$ complexity presented in [28].

We also observe that a high flops rate is achieved for larger problems. In particular, when the grid size reaches $2,047$, we achieve 67% (6.16/9.2) of peak performance of the machine. This is due to the fact that as the problem size increases, the overall computation is dominated by computation performed on the dense matrix blocks associated with large supernodes. Therefore the performance approaches that of dense matrix-matrix multiplications.

4.2 Parallel Scalability

In this section, we report the performance of our implementation when it is executed on multiple processors. Although 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, we will first show how well the computation performs on a problem of fixed size with respect to the number of processors. In Table 2, we report the wall clock time (in seconds) used to compute the diagonal of A^{-1} defined a $2,047 \times 2,047$ grid. In the third column of the table, we report also the speedup factor defined as $\tau = t_1/t_{n_p}$, where t_{n_p} is the wall clock time recorded for an n_p-processor run.

n_p	wall clock time	speedup factor	Gflops
1	100.1	1.0	6.2
2	52.2	1.9	11.8
4	30.2	3.3	20.2
8	16.8	6.0	33.5
16	9.5	10.5	55.9
32	5.7	17.6	90.0
64	3.3	30.3	156.2
128	2.3	42.3	226.4
256	1.8	55.6	281.7
512	1.7	58.9	294.2

Table 2: The scalability of parallel computation used to obtain A^{-1} for A of a fixed size ($n = 2047 \times 2047$.)

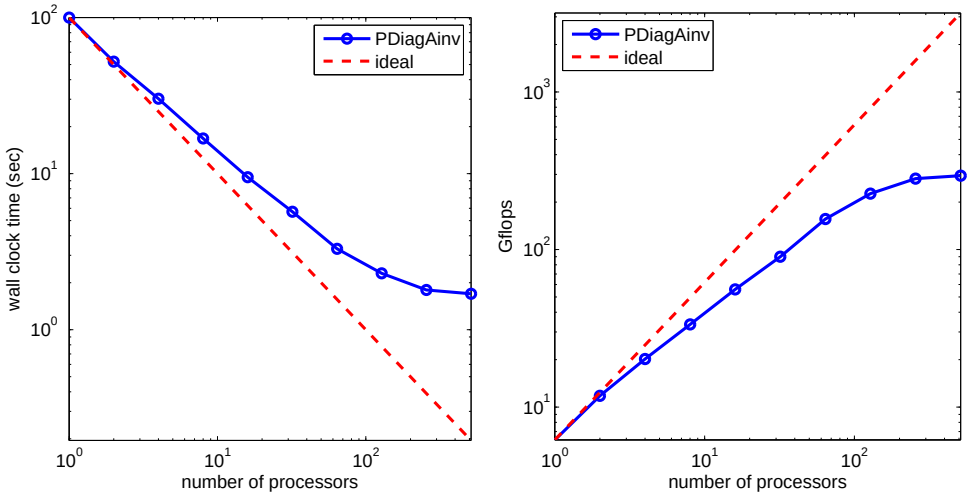


Figure 7: Log-log plot of total wall clock time and total Gflops with respect to number of processors, compared with ideal scaling. The grid size is fixed to be 2047×2047.

Figure 7 compares the performance of our algorithm, called PDiagAinv, with ideal scaling in terms of total wall clock time and total Gflops. As we can clearly see from Table 2 and Figure 7, for problem of this fixed size, deviation from the ideal speedup begins to show up when the computation is performed in parallel on 4 processors. The performance barely improves in a 512-processor run compared to the 256-processor run. Beyond that point, communication overhead starts to dominate. We will discuss the sources of communication overhead in the next few subsections.

In terms of weak scaling, PDiagAinv 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 3, 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 a 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 last column of Table 3, 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 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 3 shows that the performance of our block LDL^T factorization deviates slightly from the ideal

scaling, while the selected inversion process achieves nearly ideal scaling up to $4,096$ processors. The scaling of flops and wall clock time can be better viewed in Figure 8, in which the code performance is compared to ideal performance using log-log plot. From Table 3, we can also see that that the selected inversion time is significantly less than that associated with factorization when the problem size becomes sufficiently large. This is not because the selected inversion process performs fewer floating point operations. On the contrary, our direct measurements of flops by `PAPI` indicates that the total number of floating point operations performed in selected inversion is slightly more than that performed in the factorization. However, the flop rate associated with the selected inversion process is much higher because selected inversion does not need to traverse the elimination tree in postorder and it does not call ScaLAPACK subroutines `pdgetrf` and `pdgetri` that are used in the factorization to invert the diagonal blocks of D. The efficiency of these calculations are typically lower than the simple dense matrix-matrix multiplications used exclusively in the selected inversion process.

grid size	n_p	symbolic time	factorization time	inversion time	total time	weak scaling ratio
1,023	1	0.92	7.29	6.77	14.99	–
2,047	4	1.77	14.44	13.82	30.04	2.00
4,095	16	1.82	34.26	25.39	61.82	2.05
8,191	64	1.91	86.35	47.07	135.34	2.18
16,383	256	1.98	207.51	89.91	299.41	2.21
32,767	1024	2.08	474.94	174.57	651.59	2.17
65,535	4096	2.40	1109.09	348.13	1459.62	2.24

Table 3: The scalability of parallel computation used to obtain A^{-1} for A for increasing system sizes. The largest grid size is $65,535 \times 65,535$ and corresponding matrix size is approximately 4.3 billion.

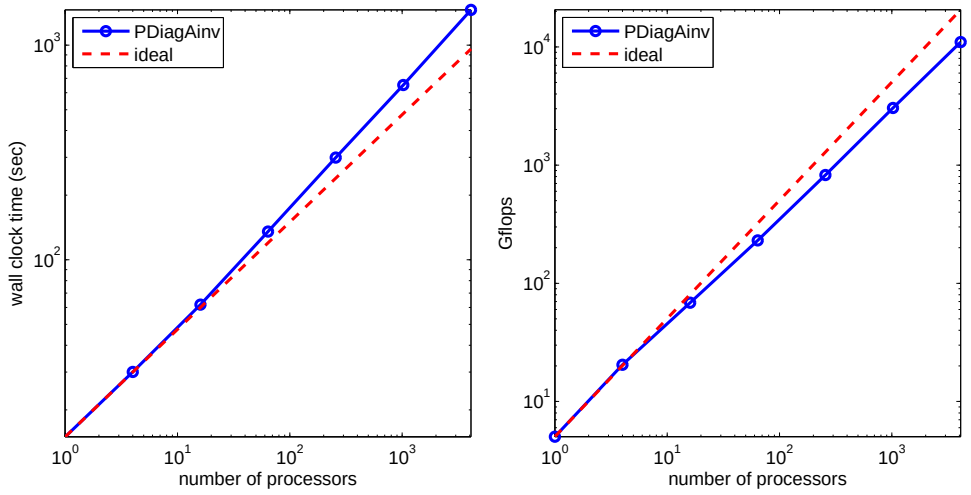


Figure 8: Log-log plot of total wall clock time and total Gflops with respect to number of processors, compared with ideal scaling. The grid size starts from 1023×1023, and is proportional to the number of processors.

4.3 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 9 shows the overall flop counts measured on each processor for a 16-processor run used to compute the diagonal of A^{-1} 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. Using the flop count given in Figure 9, we can divide these 16 processors into four groups. Processors within each group perform roughly the same number of flops. The presence of these four groups can be explained by examining the relative positions of each aggregated supernode with respect to ancestors of these supernodes on the computational grid. In Figure 2(b), each one of the four inner supernodes, i.e., supernodes `5, 11, 17, 23`, borders with four separators. Hence, the four processors assigned to these supernodes perform the largest number of operations. Supernodes `4, 12, 16, 25` each only border with three separators. They perform fewer number of flops, hence form another group. Supernodes `2, 8, 20, 26` also each border with three separators. However, because they do not border with the largest separator `31`, the number of operations performed by the processors assigned to these supernodes is slightly less than those that assigned to supernodes `4, 12, 16, 25`. Hence these processors form the third group of processors in the bar chart shown in Figure 9. Finally, supernodes `1,9,19,27` each interact with only two lower level separators, the processors assigned to these supernodes perform the least number of flops. We should note that the load balance pattern exhibited by Figure 9 will become more complicated when number of processors increases.

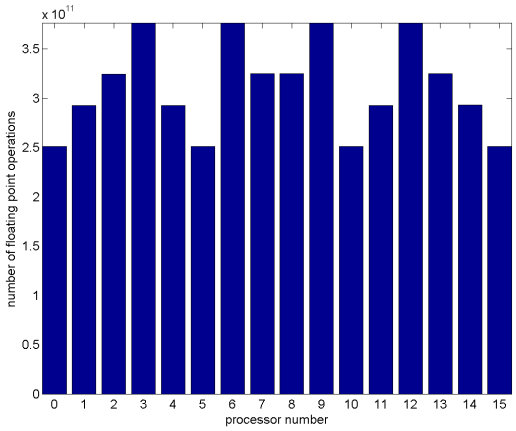


Figure 9: The number of flops performed on each processor for computing the diagonal of A^{-1} defined on a $4,095 \times 4,095$ grid.

4.4 Communication Overhead

A comprehensive measurement of the communication cost can be collected using the `IPM` tool. Table 4 shows the overall communication cost increases moderately as we double the problem size and quadruple the number of processors at the same time.

grid size	n_p	communication (%)
1,023	1	0
2,047	4	2.46
4,095	16	11.14
8,191	64	20.41
16,383	256	28.43
32,767	1024	34.46
65,535	4096	40.80

Table 4: Communication cost as a percentage of the total wall clock 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 point.

2. Communication volume. This is the amount of data transfered 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 is 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 10. In this particular case, `MPI_Barrier`, which is used to synchronize all processors, 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` function is used in several places in our code. In particular, it is used in the `ParSum` subroutine to ensure that all contributions from the children of a supernode `I` are available before these contributions are accumulated and redistributed among `procmap(I)`. The barrier function is also 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 resulting from these barrier function calls results from the variation of computational loads discussed in section 4.3. 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`, dense matrix factorization and inversion routines `pdgetrf` and `pdgetri` respectively.

Figure 11 shows that the percentage of wall clock time spent in `MPI_Barrier` increases moderately as more processors are used solve larger problems. Such 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.

[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

Figure 10: Communication profile for a 16-processor run on a $4,095 \times 4,095$ grid.

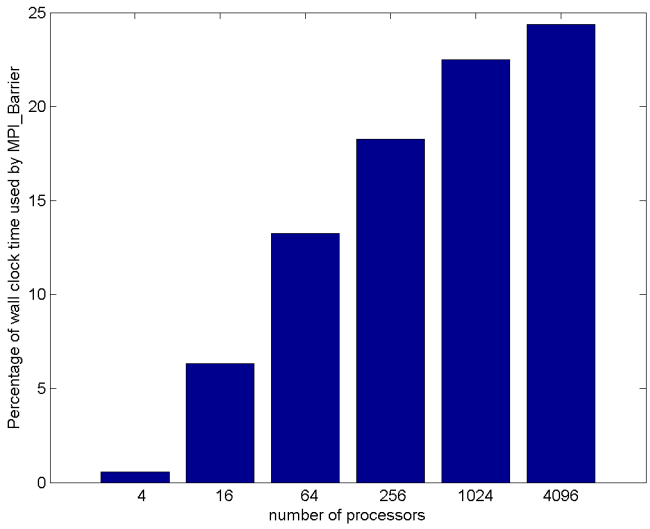


Figure 11: The percentage of time spent in `MPI_Barrier` as a function of the number of processors used and the size of the grid.

In addition to the idle wait time spent in `MPI_Barrier`, the communication overhead is also affected by the volume of data transfered among different processors and how frequent these transfer occurs. 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. Although we do not have a direct way to measure communication volume, Figure 10 shows that the total amount of 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 $4,095 \times 4,095$ 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 are 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 10 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.

4.5 Memory Consumption

In addition to maintaining good load balance among different processors, the data-to-processor mapping scheme discussed in section 3.2.2 also ensures that the memory usage per core only increases logarithmically with respect to matrix dimension in the context of weak scaling. This estimation is based on the observation that when the grid size by a factor of two, the dimension of the extra blocks associated to `L` and `D` to 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 current grid size. This logarithmic dependence is clear from Figure 12, where 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 for 2D rectangular quantum dots. We consider a 2D electron quantum dot confined in a rectangular domain, a model investigated in [41]. This model is also provided in the test suite of the Octopus software [6] which we use for comparison. In this model, the Kohn-Sham Hamiltonian has the form

$$H = -\frac{1}{2m}\Delta + V_{\text{ext}}(\mathbf{r}) + V_H(\mathbf{r}) + V_{\text{xc}}(\mathbf{r}), \tag{6}$$

where V_H is the Hartree potential, V_{xc} is a 2D exchange-correlation potential given in [3] and V_{ext} is an external potential that describes an infinite hard-wall confinement in the xy plane, *i.e.*,

$$V_{\text{ext}}(x,y) = \begin{cases} 0, & 0 \le x \le L, 0 \le y \le L; \\ \infty, & \text{elsewhere.} \end{cases} \tag{7}$$

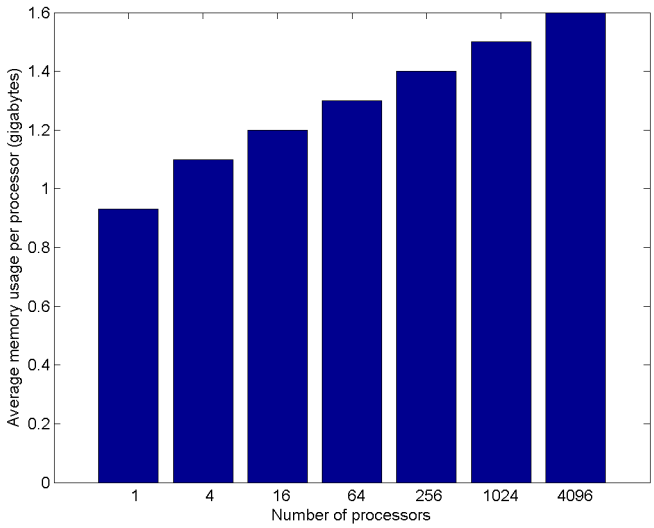


Figure 12: The average memory usage per processor as a function of the number of processors used and the size of the grid. The memory cost per core depends on the matrix dimension logarithmically.

To simplify our experiment, we do not consider spin-polarization. The Laplacian operator Δ is discretized using a five-point stencil. A room temperature of 300K 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 an 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 13. In this case, the quantum dot behaves like a metallic system with a tiny energy gap around 0.08eV.

In Octopus, the electron density associated with the minimum total energy of the quantum dot is obtained by using the self-consistent field (SCF) iteration to solve the Kohn-Sham equations [33]. In the kth SCF iteration, Octopus uses a conjugate gradient (CG) like algorithm to compute the $n_e/2 + n_{\text{ext}}$ smallest eigenvalues of H where n_e is the number of electrons in the quantum dot and n_{ext} is the number of extra states for finite temperature calculation. The extra number depend on the system size. For example, in the case of 32 electrons 4 extra states are necessary for the electronic structure calculation at 300K. The corresponding eigenvectors are used to obtain a new approximation to the electron density $\rho^{(k)} = \text{diag}\,[f_{\beta,\mu}(H(\rho^{(k-1)}))]$, where $f_{\beta,\mu}$ is the Fermi-Dirac distribution defined in section 1.

On the other hand, pole expansion (1) allows the calculation of electron density at room temperature with a small number of poles. The numerical results in [29] shows that in most cases, at most 80 poles are sufficient to achieve a relative accuracy in the order of 10^{-7} in L^1 norm. Pole expansion also requires an explicit knowledge of chemical potential μ. The chemical potential is chosen such that the condition

$$\text{Tr}[\rho] = n_e \tag{8}$$

is satisfied using Newton-Raphson iteration. In order to use (1), we need to compute the diagonal of the inverse of a number of complex symmetric (non-Hermitian) matrices $H - (z_i + \mu)I$ $(i = 1, 2, ..., P)$. In addition to using the parallel algorithm presented in section 3 to evaluate each term in (1), an extra level of coarse grained parallelism can be achieved by assigning each pole to a different group of processors.

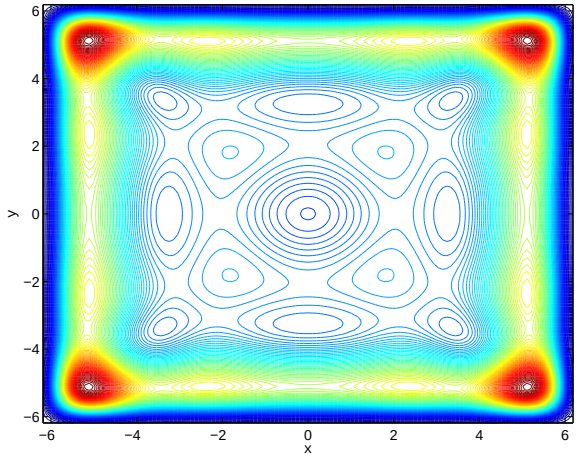


Figure 13: A contour plot of the density profile of a quantum dot with 32 electrons.

In Table 5, we compare the efficiency of the pole expansion technique for the quantum dot density calculation performed with the standard eigenvalue calculation based approach implemented in Octopus using the averaged time spent in each SCF iteration. The maximum number of CG iterations for computing each eigenvalue in Octopus is set to the default value 25. We label the pole expansion based approach that uses the algorithm and implementation discussed in section 3 as PCDiagAinv, where the letter C stands for complex. The factor 80 in the last column of Table 5 comes from 80 poles. 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 PCDiagAinv calculation, whereas such a perfect reduction in wall clock time cannot be easily 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 diag $[H - (z_i + \mu)I]^{-1}$ becomes quite evident. This is because when the number of electrons in the quantum dot is large, the computation cost associated with the eigenvalue calculation in Octopus is dominated by the computation performed to maintain mutual orthogonality among different eigenvectors. 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 2048 electrons, PCDiagAinv is eight times faster than Octopus.

6 Concluding Remarks

We have presented an efficient algorithm and its implementation for computing the diagonal of the inverse of a discretized 2D Kohn-Sham Hamiltonian H. Such a computation allows us to evaluate $f_{\beta,\mu}(H)$ through a recently developed pole-expansion techniques [27, 29], where $f_{\beta,\mu}$ is the Fermi-Dirac distribution function.

Our algorithm consists of two major steps. In the first step, we perform a block LDL^T factorization of the discretized Hamiltonian. Although this type of computation has been investigated extensively in the direct sparse matrix computation community, our algorithm and implementation are different

n_e(#Electrons)	Grid	#proc	Octopus time(s)	PCDiagAinv 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

Table 5: Averaged time spent on each SCF iteration for Octopus and PCDiagAinv with systems of various sizes. The time for PCDiagAinv in the last column assumes using 80 poles. The time corresponding to 2048 electron system using Octopus with 1 and 4 processors is not recorded, since the memory cost exceeds the constraint of 2GB per core on Franklin machine.

from those used in most of the existing software packages [1, 2, 19, 20, 37, 45]. We use a block row-based factorization algorithm that is a block extension of the algorithm presented in [9]. The block row-based algorithm is implemented via recursion. 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. We also use the techniques of local buffer and relative indices to ensure the synchronization cost associated with the parallel update of L and D is minimized. In the second step a block selected inversion algorithm is proposed to compute selected elements of A^{-1}. Besides the techniques in block LDL^T algorithm, level-by-level restriction is utilized to overcome 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 billions degrees of freedom on 4096 processors. The code exhibits satisfactory weak scaling property.

When compared with standard approach for evaluating the diagonal a Fermi-Dirac function of a Kohn-Sham Hamiltonian associated 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's is much faster, especially when the quantum dot contains many electrons.

One of the key components that enable us to achieve high performance in our implementation is the parallel symbolic analysis procedure that makes explicit use of the grid and stencil geometry as well as the nested dissection partition and ordering technique. This type of analysis can be easily extended to regular 3D grids and so is our algorithm and implementation for selected inversion calculation.

To compute the diagonal of A^{-1} for problems that are defined on irregular grids (e.g., problems that are discretized by finite elements or some other techniques), a more general approach based on a general left-looking algorithm [37], a block fan-out algorithm [43] or a multifrontal algorithm is more appropriate. In particular, a sequential algorithm based on the general left-looking LDL^T factorization has been developed in [`?diagex`]. In terms of parallel implementation, these approaches would also require a more general parallel symbolic analysis similar to that developed in [48]. This is an area of

research we are currently pursuing.

Acknowledgement.

This work was partially supported by DOE under Contract No. DE-FG02-03ER25587 and by ONR under Contract No. N00014-01-1-0674 (L. L., J. L. and W. E), by an Alfred P. Sloan fellowship and a startup grant from the University of Texas at Austin (L. Y.), and by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231 (C. Y.). The computational results presented were obtained at the National Energy Research Scientific Computing Center (NERSC), which is supported by the Director, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under contract number DE-AC02-05CH11232. We would like to thank Esmond Ng and Sherry Li for helpful discussion.

References

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

[2] C. Ashcraft and R. Grimes, *SPOOLES: An object-oriented sparse matrix library*, Ninth SIAM conference on parallel processing, 1999.

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

[4] S. Baroni and P. Giannozzi, *Towards very large-scale electronic-structure calculations*, Europhys. Lett. **17** (1992), 547–552.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK users' guide* (1997).

[6] 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), 2465–2488.

[7] M. Ceriotti, T. Kühne, and M. Parrinello, *An efficient and accurate decomposition of the Fermi operator.*, J. Chem. Phys **129** (2008), 024707.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed., MIT Press, 2009.

[9] T. Davis, *Algorithm 849: A concise sparse Cholesky factorization package*, ACM Trans. Math. Software **31** (2005), 587–591.

[10] R. Dreizler and E. Gross, *Density functional theory: an approach to the quantum many-body problem*, Springer-Verlag Berlin, 1990.

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

[12] ———, *Direct methods for sparse matrices*, Oxford University, London, 1987.

[13] A. Erisman and W. Tinney, *On computing certain elements of the inverse of a sparse matrix*, Numer. Math. **18** (1975), 177.

[14] J. George and J. Liu, *Computer solution of large sparse positive definite systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[15] J. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal. **10** (1973), 345–363.

[16] S. Goedecker and L. Colombo, *Efficient linear scaling algorithm for tight-binding molecular dynamics*, Phys. Rev. Lett. **73** (1994), 122.

[17] S. Goedecker and M. Teter, *Tight-binding electronic-structure calculations and tight-binding molecular dynamics with localized orbitals*, Phys. Rev. B **51** (1995), 9455–9464.

[18] S. Goedecker, *Linear scaling electronic structure methods*, Rev. Mod. Phys. **71** (1999), no. 4, 1085–1123.

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

[20] A. Gupta, *WSMP: the Watson Sparse Matrix Package*, IBM Research Report (1997), no. RC 21886(98462).

[21] J. Gustafson, *Reevaluating Amdahl's law*, Comm. ACM **31** (1988), 532–533.

[22] M. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Rev. **33** (1991), 420–460.

[23] P. Hohenberg and W. Kohn, *Inhomogeneous electron gas*, Phys. Rev. **136** (1964), B864.

[24] Cray Inc., *Using cray performance analysis tools*, Cray Inc., 2009.

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

[26] 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), 9408 –9427.

[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 (2009), 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*, Comm. Math. Sci. (accepted).

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

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

[31] ———, *A generalized envelope method for sparse factorization by rows*, ACM Trans. Math. Software **17** (1991), 112–129.

[32] ———, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev. **34** (1992), 82–109.

[33] R. Martin, *Electronic structure – basic theory and practical methods*, Cambridge University Press, 2004.

[34] C. Moler, *Numerical computing with MATLAB*, SIAM, 2004.

[35] S. Moore, P. Mucci, J. Dongarra, S. Shende, and A. Malony, *Performance instrumentation and measurement for terascale systems*, Lecture notes in computer science, 2003, pp. 53–62.

[36] E. Ng and B. Peyton, *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors*, SIAM J. Sci. Comput. **14** (1993), 761.

[37] ———, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput. **14** (1993), 1034.

[38] T. Ozaki, *Continued fraction representation of the fermi-dirac function for large-scale electronic structure calculations*, Phy. Rev. B **75** (2007), 035123.

[39] N. Pelekanos, J. Ding, M. Hagerott, A. Nurmikko, H. Luo, N. Samarth, and J. Furdyna, *Quasi-two-dimensional excitons in (zn,cd)se/znse quantum wells: Reduced exciton˘lo-phonon coupling due to confinement effects*, Phys. Rev. B **45** (1992), 6037–6042.

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

[41] E. Räsänen, H. Saarikoski, V. Stavrou, A. Harju, M. Puska, and R. Nieminen, *Electronic structure of rectangular quantum dots*, Phys. Rev. B **67** (2003), 235307.

[42] D. Rose and R. Tarjan, *Algorithmic aspects of vertex elimination on directed graphs*, SIAM J. Appl. Math. **34** (1978), 176–196.

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

[44] Y. Saad and R. Sidje, *Rational approximation to the Fermi-Dirac function with applications in Density Functional Theory*, Technical Report umsi-2008-279, Minnesota Supercomputer Institute, 2008.

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

[46] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software **8** (1982), 256–276.

[47] D. Skinner and W. Kramer, *Understanding the causes of performance variability in HPC workloads*, IEEE international symposium on workload characterization, IISWC05, 2005.

[48] E. Zmijewski and J. R. Gilbert, *A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor*, Parallel Comput. **7** (1988), 199–210.