SIPs: Shift-and-Invert Parallel Spectral Transformations

HONG ZHANG
Computer Science Department, Illinois Institute of Technology
BARRY SMITH
Mathematics and Computer Science, Argonne National Laboratory
MICHAEL STERNBERG and PETER ZAPOL
Materials Science, Argonne National Laboratory

SIPs is a new efficient and robust software package implementing multiple shift-and-invert spectral transformations on parallel computers. Built on top of SLEPc and PETSc, it can compute very large number of eigenpairs for sparse generalized Hermitian eigenvalue problems.

The development of SIPs is motivated by applications in nanoscale materials modeling, in which the growing size of the matrices and the pathological eigenvalue distribution challenge the efficiency and robustness of the solver. In this paper, we develop a parallel eigenvalue algorithm that is based on the idea of distributed spectrum slicing. We describe SIPs' object-oriented software design and implementation techniques, and demonstrate its numerical performance on an advanced distributed computer.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Eigenvalues and eigenvectors, Sparse, structured, and very large systems (direct and iterative methods)*; G.4 [**Mathematical Software**]: Parallel and vector implementations, Efficiency, Reliability and robustness

General Terms: Algorithm, Performance

Additional Key Words and Phrases: Parallelism, sparse eigenvalue computation, spectral transformation

1. INTRODUCTION

This work is motivated by applications in nanoscale materials modeling. A key problem in this field is to calculate and optimize the configurational energy of a system of atoms. A hierarchy of methods exists, and the mathematical core underlying many of them is a generalized eigenvalue problem of the form

$$Ax_i = \lambda_i B x_i \,, \qquad i = i_{\min}, \cdots, i_{\max} \,, \tag{1}$$

where A and B are $n \times n$ Hermitian matrices, B is positive definite, and $i_{\min}$ and $i_{\max}$ are the index range of requested eigenpairs.

Author's address: H. Zhang, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439-4844

More specifically, we use the density-functional-based tight-binding method (DFTB) [Elstner et al. 1998] for materials modeling applications. The eigenproblems posed by DFTB as studied in this paper are distinguished by several features:

(1) The matrix pencil $(A,\, B)$ is large and sparse. Its size n is proportional to the number of atoms in the model. With an ultimate goal of simulating 50,000 atoms, the matrices are expected to be as large as $n = 200,000$.
(2) A large number of eigensolutions are requested, e.g., 60% eigenvalues and the associated eigenvectors are wanted based on current DFTB applications.
(3) The spectrum is pathologically difficult. It has clusters of hundreds of tightly packed eigenvalues, and very poor average relative eigenvalue separation:

$$\text{ave}\left(\frac{\lambda_{i+1}-\lambda_i}{\lambda_n-\lambda_1}\right) = O(n^{-1}) \quad \text{(typically} \approx 10^{-5} \text{ in our study).}$$

Even worse, some clusters are adjacent to gaps that have lengths far larger than the average eigenvalue separation:

$$\lambda_{j+1}-\lambda_j \gg \text{ave}\,(\lambda_{i+1}-\lambda_i)\,.$$

(4) The global coupling of the nonzero elements in $(A,\, B)$ gives rise to not-very-sparse to dense matrix factorizations. For example, the matrix factors of matrices with $n = 16,000$ under study have sparse densities ranging from 7% to 50%. By conventional sparse matrix standards, 7% is still extremely dense.
(5) The simulation requires a significant number of iterations (possibly 1000's) of Eq. (1) with closely related matrices A and B.[1]

In this paper, we consider the eigenvalue problem Eq. (1) with the features described above. Our discussion focuses on computing eigenvalues ordered increasingly from $i_{\min}$ to $i_{\max}$, and their associated eigenvectors. The discussion can easily be applied to other general forms, e.g., computing all eigenvalues in $[a, b]$ and their eigenvectors.

Existing eigensolvers are customarily developed based on two types of matrix storage: dense and sparse storage. They are typically solved using *direct methods* and *iterative methods*, respectively.

Direct methods compute all or almost all eigensolutions of dense matrices. They exhibit $O(n^3)$ time and $O(n^2)$ memory complexity. Library software based on the direct methods is provided in the LAPACK [Anderson et al. 1999] and ScaLAPACK [Blackford et al. 1997] packages. Both were used in the past for DFTB, but time and memory scaling prevents us from advancing to larger nanoscale systems of current interest. In particular, our experience has shown that extensive communication requirements in ScaLAPACK cause scaling problems on workstation clusters with commodity networking hardware.

Iterative methods such as Lanczos [Lanczos 1950] and Jacobi-Davidson [Sleijpen and van der Vorst 1996] are widely used for extracting a few extreme eigensolutions of sparse matrices. Their time and space complexity are bound above by

[1]The matrix A contains a perturbation dependent upon the solution vectors x, so that a fixpoint solution is sought for a given position of atoms and fixed B. In an outer iteration, both matrices depend on the positions of atoms being optimized or time-stepped.

the complexity of direct methods, yet they are usually much more efficient when the matrices involved are indeed sparse. The available software include ARPACK [Lehoucq et al. 1998] and several others [Hernandez et al. 2004] [Marques 1995] [Wu and Simon 1997]. When interior eigenvalues are requested, a practical approach is to replace (A, B) by a shift-and-invert operator C [Ericsson and Ruhe 1980]:

$$\begin{aligned} Ax = \lambda Bx \quad &\Longleftrightarrow \quad (A-\sigma B)x = (\lambda-\sigma)Bx, \quad \sigma \neq \lambda \\ &\Longleftrightarrow \quad \tfrac{1}{\lambda-\sigma} y = B \underbrace{(A-\sigma B)^{-1}}_{C} y, \quad y = Bx, \end{aligned}$$

which yields

$$Cy = \tilde{\lambda} y, \qquad \tilde{\lambda} = \frac{1}{\lambda - \sigma}\,. \tag{2}$$

Employing Lanczos iterations to (2) leads to eigensolutions of the original equation (1) that are close to the shift σ.

DFTB and related methods for materials modeling require a large fraction of eigensolutions—typically the lower 50% of the spectrum—with accurate accounting for all requested eigenpairs and reliable orthogonalization in degenerate or nearly degenerate subspaces. Without significant customization, none of the available iterative eigenvalue packages are able to provide sufficient efficiency and robustness crucial to the modeling process. In this article we propose SIPs, a new software package implementing shift-and-invert parallel spectral transformations on top of the existing iterative eigensolver. We introduce an eigenvalue algorithm in Section 2, and describe its implementation in Section 3. The implementation includes the object-oriented software design for performance, portability and re-usability, and the techniques that build efficiency and robustness into the proposed eigensolver. In Section 4 we present numerical experiments using SIPs, and comparison with ScaLAPACK. From our tests, three systems of nanoscale materials with diverse characteristics are reported on for their distinct sparse densities of the matrix factorization, a dominating factor for the performance of SIPs. Finally, we give conclusions and impact of this work.

2. MULTIPLE SHIFT-AND-INVERT PARALLEL EIGENVALUE ALGORITHM

Based on the idea of distributed spectrum slicing, we propose concurrently using Lanczos iterations with multiple shift-and-invert spectral transformations on a distributed eigenvalue spectrum. Note, this general approach have been studied by others, e.g., [Teranishi et al. 2003], but our algorithm described here is different, and the software development is entirely new.

Initially, the user provides a requested eigenvalue interval. We divide it into overlapping subintervals and assign each to a process. A process starts from a shift in the middle of its interval, and picks new shifts at the left and right sides of the current shift. The bounds of processed subintervals are exchanged between neighboring processes during the computation of local eigensolutions. Using this information, each process adjusts its assigned subinterval, which redistributes the initially assigned subintervals dynamically and balances the parallel workload.

Within each process, multiple shifts are selected one after the other for computing all eigensolutions in the assigned subinterval. For a single shift, the shift-and-invert

spectral transformation enhances convergence to the eigenpairs that are close to the shift. A well-chosen shift allows us to compute tens to hundreds of eigensolutions with one to several Lanczos runs.

When a particular shift σ is chosen, we apply a matrix factorization

$$A - \sigma B = LDL^T, \tag{3}$$

then feed it into a Lanczos iteration for generating a Krylov subspace. As a byproduct, Eq. (3) also provides $\nu(A - \sigma B)$, the number of eigenvalues of (A, B) that are smaller than σ. For simplicity, we denote $\nu(\sigma) = \nu(A - \sigma B)$ and refer to it as *matrix inertia*. This number is used for validating the eigensolutions computed through the shift σ. Each shift incurs an expensive matrix factorization (3), and two further shifts σ_i, σ_j are needed for checking the validity of eigensolutions in the interval (σ_i, σ_j). To make our solver efficient and robust, we dynamically select a set of shifts which produces multiple sets of eigensolutions at minimum redundancy, and reuse these shifts to validate the eigensolutions.

We describe the Multiple Shift-and-Invert Parallel Eigenvalue Algorithm in Fig. 1.

Input: matrix pencil (A, B);
index range $i_{\min}$, $i_{\max}$ and/or requested eigenvalue spectrum $(\lambda_{\min}, \lambda_{\max})$;
eigenvalue approximation $\{\hat{\lambda}_i\}$ (optional).

Output: k-th process has its local list of eigenpairs indexed from
i_k to $i_{k+1} - 1$ ($i_0 = i_{\min}$, $i_{\text{np}} = i_{\max}$).

Process k (k=0,...,np-1):

(1) Initialize:
  (a) set an initial shift $\sigma_0^{(k)}$; compute matrix inertia $\nu(\sigma_0^{(k)})$;
  (b) get initial local assigned spectrum $(\lambda_{\min}^{(k)}, \lambda_{\max}^{(k)})$ and associated index bound $i_{\min}^{(k)}$, $i_{\max}^{(k)}$;
  (c) initialize local computed spectrum $[\sigma_{\min}^{(k)}, \sigma_{\max}^{(k)}]=\{\sigma_0^{(k)}\}$.

(2) Do until the local list of eigensolutions is full:
  (a) compute eigenpairs that are close to the selected shift $\sigma_i^{(k)}$ by the shift-and-invert Lanczos iteration;
  (b) check validity of the computed eigenvalues against the eigenvalues that are already on the local list; add the new and desired eigensolutions to the list;
  (c) compute new possible shifts at left and right side of $\sigma_i^{(k)}$;
  (d) pick next shift $\sigma_{i+1}^{(k)}$ and compute $\nu(\sigma_{i+1}^{(k)})$; if $\sigma_{i+1}^{(k)} < \sigma_{\min}^{(k)}$ or $\sigma_{i+1}^{(k)} > \sigma_{\max}^{(k)}$, update computed spectrum $[\sigma_{\min}^{(k)}, \sigma_{\max}^{(k)}]$ and send $\sigma_{i+1}^{(k)}$, $\nu(\sigma_{i+1}^{(k)})$ to neighboring processes;
  (e) receive messages from neighboring processes and update its assigned spectrum $(\lambda_{\min}^{(k)}, \lambda_{\max}^{(k)})$ and index bounds $i_{\min}^{(k)}$, $i_{\max}^{(k)}$.

(3) Final check:
  (a) exchange $\lambda_{\min}^{(k)}$, $\lambda_{\max}^{(k)}$ and $i_{\min}^{(k)}$, $i_{\max}^{(k)}$ with neighboring processes;
  (b) delete duplicate eigensolutions.

Fig. 1. Multiple Shift-and-Invert Parallel Eigenvalue Algorithm

3. IMPLEMENTATION OF THE ALGORITHM

Next, we discuss the implementation of the Multiple Shift-and-Invert Parallel Eigenvalue Algorithm. We start from the software design, then proceed to explain the techniques for dynamically selecting shifts, bookkeeping of locally computed eigensolutions, maintaining parallel job balance, ensuring global accuracy of the eigensolutions, and finally, organizing subgroups of MPI communicators for processing large-scale matrix operations.

3.1 Software Structure Design

The design objective of our eigensolver is to deliver high performance with limited effort for development and maintenance, and to enable portability and re-usability. Our design choices are guided by the following three major components of the algorithm:

(1) Sequential and parallel sparse matrix operations, e.g., matrix-vector multiplication, matrix factorization, triangular solve etc. These fundamental operations would dominate the performance of the eigensolver.
(2) Lanczos iterations with a single shift-and-invert spectral transformation.
(3) Sequential selection of multiple shifts, parallel distribution of the eigenvalue spectrum, and bookkeeping computed eigensolutions.

(1) and (2) have been implemented by well-developed software packages, whilst (3) is not available. After examining various existing software packages that implement (1) and (2), we choose PETSc [Balay et al. 2004] and its add-on package, SLEPc [Hernandez et al. 2004]. PETSc provides us with sequential and parallel data structures and basic operations that implement (1). SLEPc offers built-in support for spectral transformation and Lanczos eigensolvers required by (2).

Although PETSc and SLEPc provide adequate data and solver objects for implementing (1) and (2), state-of-the-art special-purpose software packages exist that either outperform or are more reliable than PETSc and SLEPc on specific tasks. Through interfaces provided by PETSc and SLEPc, we can easily make use of these desirable packages. Because the direct sparse matrix factorization and triangular solve dominate computational time, through PETSc interface, we link to the MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [Amestoy et al. 2000] [Amestoy et al. 2001] for such demanding computation. In addition, due to the pathological eigenvalue distribution, very few available iterative eigenvalue packages are able to deliver reliable solutions for our DFTB models. Among them, we pick ARPACK, whose interface is provided by SLEPc. To recapitulate, our task for the proposed algorithm (Fig. 1) is to implement (3) as a new package on top of SLEPc and PETSc. We name this new package *S*hift-and-*I*nvert *P*arallel Spectral Transformation*s* (*SIPs*). Fig. 2 illustrates our overall software design for implementing the Multiple Shift-and-Invert Parallel Eigenvalue Algorithm described in Section 2.

Through the interfaces of PETSc and SLEPc, SIPs easily uses the external eigenvalue package ARPACK and the parallel sparse direct solver MUMPS. The packages can be upgraded or replaced without programming changes to SIPs. SIPs itself implements the following major tasks:

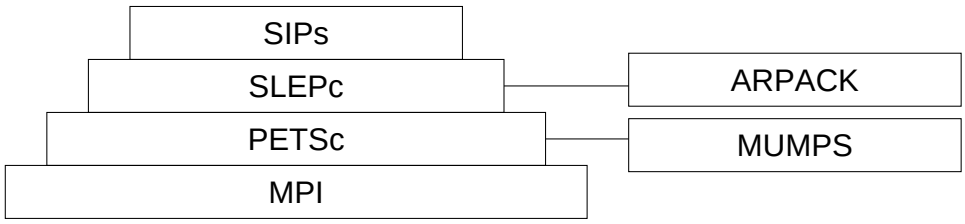

Fig. 2. Software Structure

(1) Select shifts;
(2) Bookkeep and validate eigensolutions;
(3) Balance parallel workload;
(4) Ensure global orthogonality of eigenvectors;
(5) Organize subgroups of MPI communicators.

We discuss technical details of these tasks in the next five subsections.

3.2 Select Shifts

In the beginning, an initial shift σ_0 is chosen at the midpoint of the subinterval $[\lambda_{\min}^{(k)}, \lambda_{\max}^{(k)}]$ which is assigned to the k-th process. Using it, a set of eigenvalues λ_1, λ_2, ..., λ_{nev} (in increasing order) close to σ_0 are computed. To determine the next two possible shifts, extending from the left and right of σ_0, we adopt a similar strategy as proposed by [Grimes et al. 1994], i.e., assuming the *radii of convergence* for neighboring shifts are about the same, the new shift σ_1 extending from the right of σ_0 can be selected as

$$\sigma_1 = \sigma_0 + 2\delta, \qquad \delta = \max(|\lambda_1 - \sigma_0|, |\lambda_{nev} - \sigma_0|). \tag{4}$$

Our actual code uses a slightly more conservative estimate than the above.

We use a queue to store all the selected shifts that are waiting for processing. In this article, a shift is called *active* when it is taken from the head of this queue and currently participates in the eigenvalue computation, *pending* when it waits on the queue, and *used* after its Lanczos iterations are finished.

When a new shift σ_{new} is selected from an active shift, it is appended to the end of the queue with the following data:

—σ_{left}, σ_{right}: neighboring active or used shifts (thus their matrix inertias have been computed). We call $(\sigma_{\text{left}}, \sigma_{\text{right}})$ a *pending interval* in which the eigenvalues are either uncomputed or computed, but have not gone through a validity check.

—$\nu(\sigma_{\text{left}})$, $\nu(\sigma_{\text{right}})$: $\nu_{\text{right}} - \nu_{\text{left}}$ is the number of pending eigenvalues located in $(\sigma_{\text{left}}, \sigma_{\text{right}})$.

—*isLext*, *isRext*: information about from which side of the active shift σ_{new} is selected or extended.

For example, if $\sigma_{\text{new}} = \sigma_1$ is selected from the right of an active σ_0, then it has the attached data: $\sigma_{\text{left}} = \sigma_0$, $\sigma_{\text{right}} = \lambda_{\max}$, $\nu(\sigma_{\text{left}}) = \nu(\sigma_0)$, $\nu(\sigma_{\text{right}}) = i_{\max}$, *isLext* = *false*, and *isRext* = *true*. Similarly, σ_2 is extended from left of σ_0 and attached with the data about its pending interval $(\lambda_{\min}, \sigma_0)$.

After all eigensolutions are computed from the active σ_i, and the possible new shifts at each side of σ_i are selected and appended to the end of the queue, we take σ_{i+1} from the head of the queue and proceed to the next round of Lanczos iterations until the queue becomes empty.

The shift selection process described above works well in normal situations, i.e., under the assumption that neighboring shifts have roughly the same radius of convergence. However, the DFTB eigenvalue spectrum in general has a very large disparity, and sometimes an eigenvalue cluster is adjacent to a gap that is a thousand times larger than the convergence radius of the cluster. When this occurs, the next shift σ_{i+1} likely falls into the adjacent gap and is closer to the just computed cluster than to uncomputed eigenvalues located at the other side of the gap. Using it, the eigensolver would recompute the just obtained eigenvalue cluster at an extremely high number of Lanczos runs or reach the maximum number of Lanczos runs without getting any converged eigensolutions. To deal with this difficulty, we must be able to detect a gap and know how to move the shift over to the side of uncomputed eigenvalues.

We detect the gap and move the shift based on information from the pending interval $(\sigma_{\text{left}}, \sigma_{\text{right}})$. For example, if σ has $isRext = true$ (extended from the right side of its parent shift) and $\nu(\sigma) - \nu(\sigma_{\text{left}}) \approx 0$, i.e., there are none or few eigenvalues in $(\sigma_{\text{left}}, \sigma)$, then σ likely falls into a gap. In this case, we move σ toward the right side of the pending interval $(\sigma_{\text{left}}, \sigma_{\text{right}})$:

$$\sigma = (1 - \tau) \times \sigma + \tau \times \sigma_{\text{right}}, \qquad 0.5 \leq \tau < 1. \tag{5}$$

The move (5) is repeated if a single move is not sufficient. Note that Eq. (5) assigns a new value to σ within its pending interval. While the pending interval remains unchanged, the move requires a matrix factorization (3) which should be applied only when it is necessary.

In DFTB modelings, the eigenvalue problem (1) is solved repeatedly, each with a slightly modified matrix pencil (A,B). The previously computed eigensolutions can be used as approximations to the solutions of the next eigenvalue problem. In SIPs, we use computed eigenvalues or the Rayleigh quotient of computed eigenvectors, denoted as $\{\hat{\lambda}_i\}$, as approximations to the new eigenvalues. When the approximations are available, the new shifts are selected based on its information. For example, the shift σ_1 in Eq. (4) now can be selected as

$$\sigma_1 = \hat{\lambda}_i, \quad i = \nu(\sigma_0) + nev.$$

The eigenvalue approximations $\{\hat{\lambda}_i\}$ also indicate possible gap locations, so that the shifts would be chosen effectively away from gaps.

There are other special cases in which selected shifts need to be adjusted or dumped when their pending intervals become empty due to changes made before becoming active. In this article, however, we concentrate on major techniques used by SIPs and skip technical description of minor cases.

3.3 Bookkeep and Validate Eigensolutions

We use a structure array in C, named *Local Solution List*, to store eigensolutions computed by a process. Each element in the array represents an eigensolution:

```
typedef struct {
PetscReal *val,*cval;      /* eigenvalues, eigenvalue clusters */
PetscInt  *cmap;           /* maps cluster to eigenvalues */
Vec       *vec;            /* eigenvectors */
PetscInt  *status;         /* one of status: UNCOMPUT, COMPUT, or DONE */
PetscInt  *mult,*cmult;    /* multiplicity of eigenvalues and clusters */
PetscInt  *isigma;         /* matrix inertia of the shift by which val is computed */
} EVSOL;
```

The initial length of the list in process k is set as $i_{k+1} - i_k$ (see Fig. 1). When a set of eigensolutions is computed through a shift σ, the eigenvalues are ordered as

$$\lambda_1 \leq \cdots \leq \lambda_i < \sigma < \lambda_{i+1} \leq \cdots \leq \lambda_{\text{nev}}. \tag{6}$$

Using the matrix inertia $\nu(\sigma)$, we can compute absolute indices for these eigenvalues, e.g., λ_i is the $\nu(\sigma)$-th eigenvalue of (A, B). These values are checked or compared with the ones that are already on the Local Solution List. The newly computed eigensolutions assigned to this process are then added onto the list together with relevant data.

Due to existence of multiple and clustering eigenvalues, some eigensolutions with values between λ_1 and λ_{nev} might not be computed. Without further validating the computed eigensolutions, those put on the list cannot be trusted yet. The validity check is implemented by verifying whether a computed eigenvalue is in a *trusted interval*, by which we mean the number of eigenvalues expected in the interval agrees with the number actually computed [Grimes et al. 1994].

Establishing a trust interval (a, b) requires matrix inertias at a and b, as well as the total number of computed eigenvalues in it. Because matrix factorization, Eq. (3), is expensive, we want to limit its invocation to the necessary shift-and-invert Lanczos iterations. The trust intervals are then established by reusing the active and used shifts.

Before any eigensolution is computed, its status on the Local Solution List is initialized as *UNCOMPUT*. When a set of eigensolutions are computed through the initial shift σ_0 and added onto the Local Solution List, their status are upgraded to *COMPUT*. Assume a new shift σ_1 ($> \sigma_0$) becomes active. It will generate a new set of eigensolutions in one of the following two cases:

Case 1. An eigenvalue $\lambda^{(1)}$ ($< \sigma_1$) overlaps a previously computed eigenvalue $\lambda^{(0)}$ ($> \sigma_0$). Using matrix inertia $\nu(\sigma_0)$ and $\nu(\sigma_1)$, the overlapping eigenvalues $\lambda^{(1)} = \lambda^{(0)}$ obtain two eigenvalue indices.

If the indices match, the number of actually computed eigensolutions in (σ_0, σ_1) matches the expected number $\nu(\sigma_1) - \nu(\sigma_0)$. Thus (σ_0, σ_1) is a trusted interval. All the solutions in this interval pass the validity check. Their status are then upgraded to *DONE*. Those not located inside a trusted interval have status *COMPUT*.

When the indices do not match for all the overlapping eigenvalues, the newly computed eigensolutions are put onto the list with status *COMPUT*. A new shift will be selected somewhere between σ_0 and σ_1.

Case 2. None of the newly computed eigenvalues overlap the ones already on the Local Solution List. Then the status *COMPUT* is set for all newly computed

eigensolutions. New shifts will be selected at both sides of the active shift σ_1 by the shift selection process described in Section 3.2.

This bookkeeping process repeats recursively until all the solutions on the Local Solution List have status *DONE*.

3.4 Balance Parallel Workload

The workload for each process is proportional to the assigned number of eigensolutions $i_{k+1} - i_k$ or the length of the assigned eigenvalue spectrum $(\lambda_{\min}^{(k)}, \lambda_{\max}^{(k)})$, see Fig. 1, and can be dramatically affected by the eigenvalue distribution. In general, an accurate *a priori* workload estimate is impossible.

For the k-th process, we name the interval $(\lambda_{\min}^{(k)}, \lambda_{\max}^{(k)})$ assigned to this process for eigenvalue computation the *Assigned Spectrum*, and call $[\sigma_{\min}^{(k)}, \sigma_{\max}^{(k)}]$ the *Computed Spectrum*, where $\sigma_{\min}^{(k)}$ and $\sigma_{\max}^{(k)}$ are the smallest and the largest local active or used shifts.

We balance parallel workload through dynamically updating the Assigned Spectrum during computation. Initially, we distribute overlapping Assigned Spectra into processes, then reduce the overlap through neighboring exchanges of Computed Spectra.

Using np processes, we initially pick np points inside the requested global eigenvalue spectrum $(\lambda_{min}, \lambda_{max})$

$$\lambda_{min} < \sigma_0^{(0)} < \sigma_0^{(1)} < \cdots < \sigma_0^{(np-1)} < \lambda_{max},$$

and form overlapping Assigned Spectra

$$(\sigma_0^{(k-1)}, \sigma_0^{(k+1)}), \quad k = 0, \cdots, np-1, \quad (\sigma_0^{(-1)} = \lambda_{\min}, \sigma_0^{(np)} = \lambda_{\max}).$$

Process k takes $\sigma_0^{(k)}$ as initial shift, expands its initial Computed Spectrum $[\sigma_{\min}^{(k)} = \sigma_0^{(k)}, \sigma_{\max}^{(k)} = \sigma_0^{(k)}]$ outward by selecting new shifts from both sides of $\sigma_0^{(k)}$ as described in the previous sections. When a new $\sigma_{\min}^{(k)}$ or $\sigma_{\max}^{(k)}$ is computed, e.g., a new $\sigma_{\max}^{(k)}$ is computed, it is sent to the neighbor process $k+1$. Upon receiving it, process $k+1$ is informed that $[\sigma_0^{(k)}, \sigma_{\max}^{(k)}]$, a portion of its Assigned Spectrum, has been processed by process k. Then process $k+1$ updates its Assigned Spectrum by moving its $\lambda_{\min}^{(k+1)}$ from $\sigma_0^{(k)}$ inward to $\sigma_{\max}^{(k)}$. Fig. 3 illustrates this scheme.

Assigning overlapping spectra enables some processes to compute more eigensolutions than others during same time period. As Computed Spectra expand from the middle of Assigned Spectra at various rates, each process receives information about its neighbors' Computed Spectra and updates its own Assigned Spectrum. This procedure dynamically reassigns the workload among processes. At the end, np Computed Spectra cover the user requested global eigenvalue spectrum $(\lambda_{min}, \lambda_{max})$ with minimum overlap. Duplicate eigensolutions are dumped at the final phase of the computation, in Step (3) of Fig. 1.

All processes implement this procedure using asynchronous neighboring communication of short messages. We stress that the communication cost incurred is insignificant compared with the computational cost.

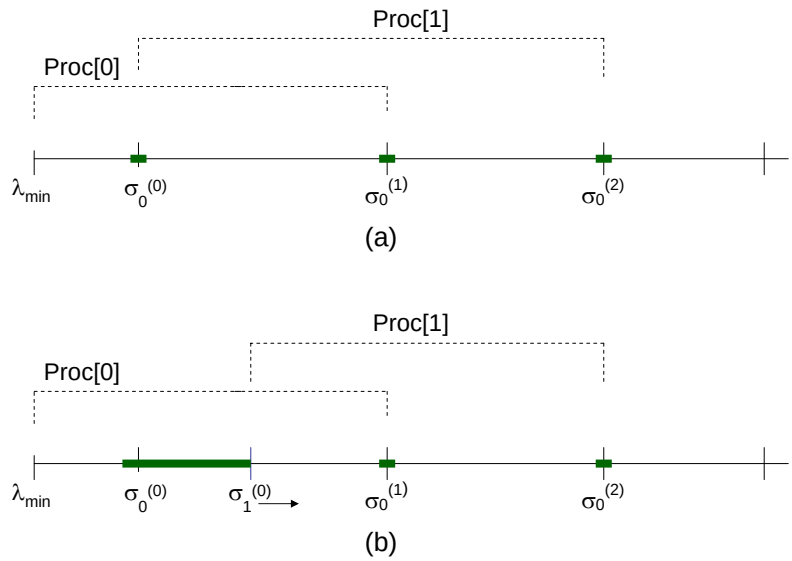

Fig. 3. Assigned Spectrum (dashed line) and Computed Spectrum (bold line)
(a) before updating; (b) after updating

3.5 Ensure External Orthogonality of Eigenvectors

In this section, we discuss how to ensure the accuracy of the proposed eigensolver, by which we mean the relative residual norm

$$r_i = \frac{||Ax_i - \lambda_i Bx_i||_2}{||\lambda_i x_i||_2} \tag{7}$$

and the orthogonality

$$\vartheta_{ij} = x_i^T Bx_j - \delta_{ij} \tag{8}$$

of computed eigenpairs, both desired to fall within user specified tolerances. We distinguish between *internal orthogonality*, i.e., orthogonality among eigenvectors computed through the *same* shift-and-invert transformation, and *external orthogonality*, for all other pairs.

Both the residual norm of all computed eigenpairs and their internal orthogonality are inherited from the external eigenvalue software package that SIPs is built upon, i.e., ARPACK [Lehoucq et al. 1998] in our current implementation. Our numerical experiments show that ARPACK gives satisfying eigensolutions when the user provides sufficiently small error tolerances (see Section 4).

What remains to be addressed regarding the accuracy of eigensolutions is the external orthogonality of eigenvectors, i.e., orthogonality between eigenvectors computed from different shift-and-invert transformations.

Hermitian matrices have orthogonal eigenvectors. They do not pose numerical difficulties as long as the associated eigenvalues are well separated. Occasionally, numerical loss of eigenvector orthogonality occurs, usually for clustered eigenvalues. In the following, our discussion focuses on how to ensure external orthogonality between clustered eigenvectors, i.e., orthogonality between eigenvectors obtained from different shifts, and their associated eigenvalues are clustered. We distinguish two cases:

Case 1. (λ_i, x_i) and (λ_{i+1}, x_{i+1}) are computed from two different shifts in the same process:

Assume λ_i is the largest eigenvalue computed from a shift σ, and the new active shift σ_{new} is close to σ with $\sigma_{\text{new}} > \sigma$. Then from σ_{new}, (λ_{i+1}, x_{i+1}) would be computed and x_{i+1} might fail to be orthogonal to x_i. In this case, we adopt the idea of *external selective orthogonalization* [Parlett and Scott 1979] [Grimes et al. 1994], i.e., computing a new set of eigensolutions by deflating the already computed invariant subspace associated to λ_i.

Our implementation takes advantage of SLEPc's built-in support for subspace deflation. We first determine when the deflation is needed, e.g., when σ_{new} is close to σ, such as $\nu(\sigma_{\text{new}}) - \nu(\sigma) < nev$, where nev is the number of requested eigensolutions for a single shift (an ARPACK parameter). We then select all eigenvectors of λ_i as the invariant subspace and call SLEPc's function EPSAttachDeflationSpace() during the setup phase for the eigenvalue solver.

Case 2. Eigenvectors are computed in two neighboring processes:

We use eigensolution duplication in this case. As discussed in Section 3.4, each process expands its Computed Spectrum until all requested eigensolutions are found collectively. During this procedure, we enforce neighboring Computed Spectra to overlap for at least one eigenvalue at the ends of their subintervals. For example, assuming $\lambda_{i-1} < \lambda_i < \lambda_{i+1}$ are single eigenvalues and λ_i is computed by two neighboring processes. From the previous discussion, the overlapping eigenvector x_i ensures it is orthogonal to the eigenvectors x_{i-1} and x_{i+1} that are associated to λ_{i-1} and λ_{i+1}, because x_{i-1}, x_i and x_i, x_{i+1} are computed by the same process respectively. Consequently, x_{i-1} and x_{i+1} are orthogonal to each other because their eigenvalues are relatively further apart. This discussion is applicable to multiple eigenvalues $\lambda_i < \lambda_j < \lambda_k$.

At the end of the computation all duplicate eigensolutions are dumped, cf. Step (3) in Fig. 1.

3.6 Organize Subgroups of MPI Communicators

Up to this point, the discussion on SIPs has assumed that Step (2) of the Multiple Shift-and-Invert Parallel Eigenvalue Algorithm (see Fig. 1) are processed sequentially, i.e., a single process holds the entire matrix pencil (A,B) and implements shift-and-invert Lanczos iterations at distributed shifts. As the size of the eigenproblems increases, the local memory space becomes a limiting factor for holding the matrix factorization and distributed eigenvectors. In this situation, we must have multiple processes provide adequate storage spaces collectively and execute matrix operations concurrently.

We cope with this local memory limitation by organizing subgroups of MPI communicators [Gropp et al. 1999]. An MPI communicator defines a context or scope for parallel communication. For example, *MPI_COMM_SELF* and *MPI_COMM_WORLD* are two default communicators normally used in MPI-based parallel programs. New communicators can be created by spliting existing ones for restricted communications. In SIPs, we introduce two new communicators, called *commEps* and *commMat*, by spliting *MPI_COMM_WORLD* into a 2D process grid. Each communicator *commEps* and *commMat* has $npEps$ and $npMat$ processes, for which we assume $npEps \times npMat = np$, the total number of processes in *MPI_COMM_WORLD*. Within each *commMat*, $npMat$ processes concurrently implement operations described in the Multiple Shift-and-Invert Parallel Eigenvalue Algorithm (Fig. 1) as sequential operations, which are primarily matrix operations. Every process also belongs to a communicator *commEps*, by which they exchange information about Computed Spectra for balancing workload among *commEps* as discussed in Section 3.4. Fig. 4 illustrates the layout of a $4(npEps) \times 3(npMat)$ process grid.

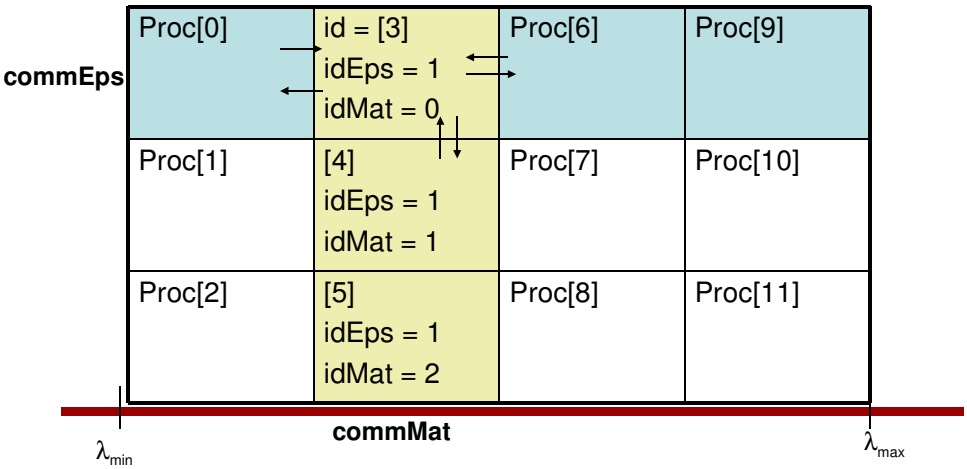

Fig. 4. Communicator Layout

4. NUMERICAL EXPERIMENTS AND COMPARISON

From algorithmic analysis and numerical experiments we find that the performance of our eigensolver will be dominated by the computational cost of matrix factorizations and triangular solves. The communication time is ignorable when $npMat = 1$, and remains insignificant as $npMat$ is increased to meet memory demand. When $npMat = 1$, SIPs only exchanges neighboring short messages which overlap with computation. When the matrix size becomes too large to be held on a single process, $npMat$ is increased to the minimum number of processes for storing the matrix factorization. For our largest tests, $npMat = 4$ was sufficient to satisfy the memory demand. The communications within the small MPI communicators $commMat$ remain insignificant for the total execution time.

We tested SIPs on eigenvalue problems arising from DFTB models of various materials and dimensionality. In order to give a balanced evaluation on SIPs' performance, we present here numerical results for three representative systems:

(1) a single-wall carbon nanotube;[2]

(2) a diamond nanowire;[3]

(3) a diamond crystal.[4]

We built each system at varying physical sizes, resulting in a consistent set of matrix sizes up to $n = 64,000$. For the first two test systems, one-dimensional periodic boundary conditions are applied in the physical model, and three-dimensional ones for the last system. The nature of the test systems and their physical boundary conditions are reflected in distinct occupancy patterns of the matrices A and B, and the resulting sparse densities of their factorizations. For example, when $n = 1,600$, the sparse densities of the matrix factorizations are 7% for the nanotube, i.e., fairly sparse, 15% for the nanowire, and 50% for diamond, i.e., dense.

The numerical experiments were performed on a Linux cluster called *Jazz*, at Argonne National Laboratory. Jazz comprises 350 computing nodes, each with a 2.4 GHz Pentium Xeon processor and a connection to both Myrinet and Ethernet communication networks.

For all the tests except the few cases discussed below, we specify the input error tolerance as $tol = 10^{-8}$. All eigensolutions achieve the relative residual norm $r_i = O(10^{-8})$, Eq. (7), and the numerical orthogonality $|\vartheta_{ij}| < 10^{-8}$, Eq. (8).

Figuress 5–7 show the execution time for computing the lowest 60% of the eigenvalues and the associated eigenvectors for the systems. Timings are collected from the second run of the eigenvalue problem (1), in which the previously computed eigensolutions provide initial eigenvalue approximations $\{\hat{\lambda}_i\}$ as discussed in Section 3.2. This is because the eigenvalue problem (1) is solved many times using initial eigenvalue approximations except for the first run. When the initial approximation $\{\hat{\lambda}_i\}$ is not available, the execution time of the current version of SIPs is sensitive to the user input data ($\lambda_{\min}$, $\lambda_{\max}$) and the initial distribution of Assigned Spectra. With sensible spectral bounds,[5] the initial run typically takes up to twice the time of the successive execution of the eigenvalue problem Eq. (1).

Fig. 5 shows the execution time for a single-wall carbon nanotube system on Myrinet (left) and Ethernet (right). For matrix sizes $n = 6,400$ to $n = 32,000$, we use $npMat = 1$ because the local distributed memory is sufficient for holding the entire matrix factorization. When $n \geq 48,000$, we increase to $npMat = 4$. The sudden increase of execution time from $n = 32,000$ to $n = 48,000$ is caused by the delay of parallel matrix factorizations and triangular solves as implemented

[2]A (10,0) *armchair*-tube with diameter 1.36 nm; 20 atoms on the circumference, at varying length.
[3]Oriented along the (001) crystal direction with (110) faces and a cross section of $(1.14\,\text{nm})^2$, 25 atoms per layer, at varying length.
[4]Same orientation as the nanowire, with periodic boundary conditions in all three dimensions, at varying size of the supercell.
[5]It should be noted that in a materials modeling application like DFTB, the eigenvalue search interval ($\lambda_{\min}$, $\lambda_{\max}$) is known reasonably well, either from general materials properties or from traditional calculations of a smaller system.

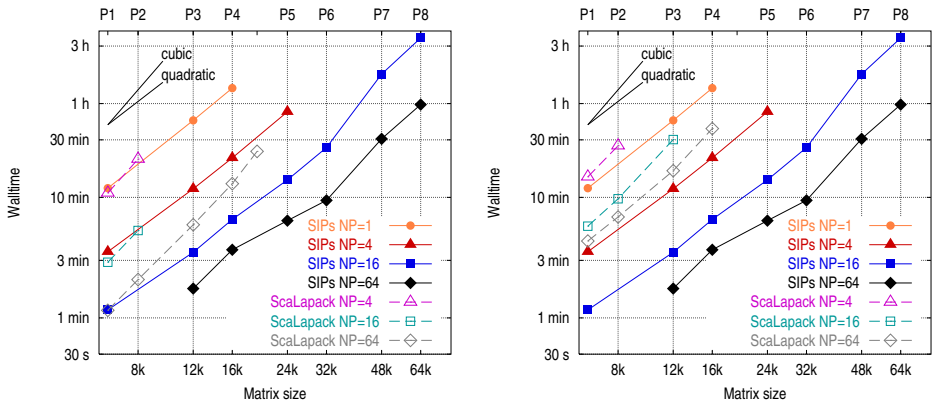

Fig. 5. Execution time for a single-wall carbon nanotube system on Myrinet(left) and Ethernet(right)

by MUMPS. Normally, one should not anticipate an ideal speedup of np when increasing the number of processes from 1 to np due to communication, algorithmic limitations and other overhead in parallel processing. We find that for the same test system on the same number of processes, the execution time with $npMat = 4$ is approximately twice as long as the case with $npMat = 1$. This indicates the speedup of matrix factorization and triangular solve implemented by MUMPS is approximately 2 from 1 process to 4 processes. We also observe much better speedups on our systems by MUMPS when the number of processes is increased from 4 processes to 16 or 64 processes. When the problem size n becomes large, e.g. $n \geq 48,000$, we notice that among more than ten thousands of computed eigensolutions, one or two pairs of eigenvectors lose orthogonality. These eigenvectors are computed from the same shift. Therefore, we tighten the error tolerance to $tol = 10^{-11}$ for the cases of $n = 48,000$ and $n = 64,000$. The resulting execution times are about 3% higher than the cases with $tol = 10^{-8}$. Actually, it is only necessary to apply such strict error tolerance to the few shifts from which highly clustered eigensolutions are computed. What we need here is an *a priori* estimate for eigenvalue clusters, based on which the error tolerances can be adjusted dynamically. This is a subject for future development of SIPs.

Fig. 5 clearly shows that the execution time scales with the problem size n as $O(n^2)$ for SIPs and $O(n^3)$ for ScaLAPACK. For a fixed problem size n, SIPs achieves a speedup of 3 or higher whenever the number of processes is increased 4 times. Because the matrices involved are sparse for this system, SIPs is significantly faster than ScaLAPCK, e.g., for problem size $n = 16,000$ using 64 processes, SIPs takes approximately 4 minutes vs. ScaLAPACK's 13 and 37 minutes on Myrinet and Ethernet respectively. ScaLAPACK fails to compute problems with a size larger than 19,200 due to memory limitations, while SIPs solves problems up to $n = 64,000$. Thus far, we have not seen any indication for SIPs' memory restriction yet.

Fig. 6 shows the execution time for a diamond nanowire system with $npMat = 1$. The matrix factors involved are about twice as dense as in the previous system,

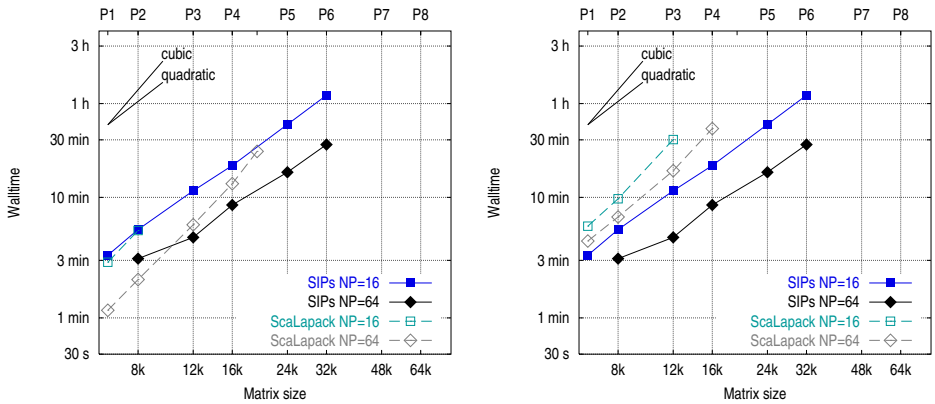

Fig. 6. Execution time for a diamond nanowire system on Myrinet(left) and Ethernet(right)

with a sparse density quantified somewhere between sparse and dense. SIPs takes a much longer execution time on this system as compared to the previous one that has sparser matrix factorizations, e.g., 27 minutes vs. 9 minutes for $n = 32,000$ and $np = 64$. However, the SIPs execution time still scales $O(n^2)$ with the problem size n. As the problem size increases, SIPs becomes faster than ScaLAPACK with increased performance on both Myrinet and Ethernet.

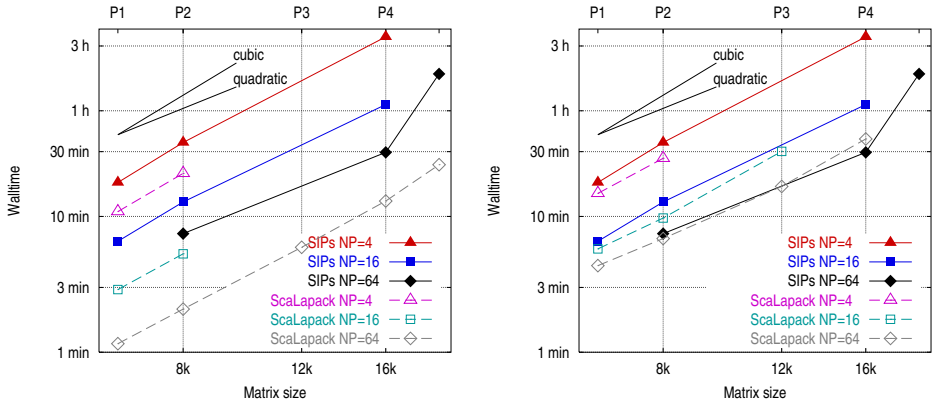

Fig. 7. Execution time for a diamond crystal system on Myrinet(left) and Ethernet(right)

For the last system, a diamond crystal, the matrix factorizations are dense with approximately 50% nonzero entries. Fig. 7 shows that ScaLAPACK is faster than SIPs on Myrinet, but its expensive communication cost on 64 processes with Ethernet pull its performance behind SIPs. Because the matrices involved are dense, SIPs scales with the problem size n worse than $O(n^2)$, but seems still scales better than $O(n^3)$ (note the slopes of the curve). When $n > 16,000$, the local distributed memories become too small for the matrix factorization, so we increase $npMat = 1$ to $npMat = 4$, which explains why the execution time jumps up at $n = 19,200$.

Summarizing from all three systems, we find that first, SIPs requires far less memory than ScaLAPACK, which enables solutions of much larger eigenvalue problems. Secondly, ScaLAPACK requires extensive data communications as the number of processes or the problem size increases. Its performance is heavily affected by the speed of network. SIPs' communication cost is ignorable when $npMat = 1$ and remains insignificant for $npMat = 4$. Its execution time on Myrinet and Ethernet are almost identical. The communication speed in parallel processing does not have a noticeable effect on SIPs' performance. Third, for matrices with sparse factorizations, the computational time for SIPs scales as $O(n^2)$ vs. $O(n^3)$ for ScaLAPACK. SIPs' computational time strongly depends on the sparsity of the matrix factorization Eq. (3). Although the number of collected data points is insufficient to draw a concluding scale, we can concede that, all systems together bear out the theoretical prediction $O(n * nnz)$, where nnz is the number of nonzero entries in the matrix factorization Eq. (3). For matrices with sparse factorizations, the computational time is significantly smaller than the ones with dense factorizations, e.g., 21 minutes for the system of carbon nanotube vs. 3.5 hours on the diamond crystal system with $np = 4$ and $n = 16,000$. Finally, SIPs is robust, giving accurate solutions for all the tested systems that have extremely pathological spectrum. For example, the system of carbon nanotube with $n = 64,000$ has more than 30,000 requested eigenvalues clustered in a relatively tiny interval (-0.9, 0.1). SIPs delivers all eigensolutions with relative residual norm $r_i = O(10^{-8})$, Eq. (7), orthogonality $|\vartheta_{ij}| = O(10^{-10})$, Eq. (8), and satisfying efficiency.

5. CONCLUSIONS

This article describes SIPs, a new efficient and robust software package implementing multiple shift-and-invert spectral transformations on parallel computers. It is developed on top of PETSc, SLEPc, ARPACK and MUMPS for computing a large number of solutions of sparse Hermitian generalized eigenvalue problems.

We presented the algorithm and detailed implementation techniques. We demonstrated parallel numerical experiments on a set of selected eigenvalue problems from nanoscale materials modeling. Comparing SIPs with ScaLAPACK on both fast and slow communication networks, SIPs (1) requires much less memory; (2) scales $O(n^2)$ vs. ScaLAPACK's $O(n^3)$ with the problem size n when the shifted matrix $(A-\sigma B)$ has sparse or not-very-dense matrix factorization; and (3) SIPs is robust, capable of computing large and pathological eigenvalue problems with high accuracy.

Its object-oriented design makes SIPs applicable to most available Lanczos-based eigensolvers, especially the solvers provided or interfaced by SLEPc. Through this design, SIPs immediately inherits the flexibility and portability from PETSc, functionalities of eigenvalue computation from SLEPc, performance and robustness from the state-of-the-art external sparse matrix solvers (MUMPS and ARPACK in our current implementation).

Finally, the work reported here is not restricted to the eigenvalue problems posed by the DFTB method. It is a general robust eigensolver applicable to a wide range of sparse Hermitian generalized eigenvalue problems. SIPs or its design and algorithmic approach can be adopted to leverage existing sparse eigenvalue software packages.

ACKNOWLEDGMENTS

We would like to thank Satish Balay for assisting software installations, and YunKai Zhou for customization of ARPACK.

REFERENCES

Amestoy, P. R., Duff, I. S., Koster, J., and L'Excellent, J.-Y. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal of Matrix Analysis and Applications 23,* 1, 15–41.

Amestoy, P. R., Duff, I. S., and L'Excellent, J.-Y. 2000. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Eng. 184*, 501–520.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., Mckcnncy, A., and Sorensen, D. 1999. *LAPACK User's Guide, third edition.* SIAM, Philadelphia.

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W., Kaushik, D., Knepley, M., McInnes, L. C., Smith, B., and Zhang, H. 2004. PETSc users manual. Tech. Rep. ANL 95/11 - Revision 2.2.1, Argonne National Laboratory.

Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., , Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. 1997. *ScaLAPACK User's Guide.* SIAM, Philadelphia.

Elstner, M., Porezag, D., Jungnickel, G., Elsner, J., Haugk, M., Frauenheim, T., Suhai, S., and Seifert, G. 1998. Self-consistent-charge density-functional tight-binding method for simulations of complex materials properties. *Phys. Rev. B 58,* 11, 7260–7268.

Ericsson, T. and Ruhe, A. 1980. The spectral transformation lanczos method. *Math. Comp 34*, 1251–1268.

Grimes, R. G., Lewis, J. G., and Simon, H. D. 1994. A shifted block lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM J. Matrix Anal. Appl. 15,* 1 (Jan.), 228–272.

Gropp, W., Lusk, E., and Skjellum, A. 1999. *Using MPI, 2nd Edition.* MIT Press.

Hernandez, V., Roman, J. E., Tomas, A., and Vidal, V. 2004. SLEPc users manual. Tech. Rep. DSIC-II/24/02 - Revision 2.2.1, D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.

Lanczos, C. 1950. An interation method for the solution of eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Stand 45*, 255–282.

Lehoucq, R. B., sorensen, D. C., and Yang, C. 1998. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, Philadelphia.

Marques, O. A. 1995. BLZPACK: Description and user's guide. Tech. Rep. TR/PA/95/30, CERFACS, Toulouse, Franc.

Parlett, B. and Scott, D. 1979. The lanczos algorithm with selective orthogonalization. *Math Comp. 33*, 217–238.

Sleijpen, G. L. G. and van der Vorst, H. A. 1996. A jacobi-davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl. 17*, 401–425.

Teranishi, K., Raghavan, P., and Yang, C. 2003. Time-memory trade-offs using sparse matrix methods for large-scale eigenvalue problems. In *Proceedings of the 2003 International Conference on Computational Science and its Applications, Lecture notes in Computer Science 2677, Editors V. Kumar, M. L. Gavrilova C. J. K. Tan, and P. L'Ecuyer.* 840–847.

Wu, K. and Simon, H. 1997. A parallel lanczos method for symmetric generalized eigenvalue problems. Tech. Rep. LBNL-41284, Lawrence Berkeley National Laboratory.