Parallel Eigenvalue Calculation based on Multiple Shift-invert Lanczos and Contour Integral based Spectral Projection Method

Hasan Metin Aktulga\textsuperscript{a*}, Lin Lin\textsuperscript{a}, Christopher Haine\textsuperscript{b}, Esmond G. Ng\textsuperscript{a}, Chao Yang\textsuperscript{a}

\textsuperscript{a}Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
\textsuperscript{b}University of Versailles St-Quentin-En-Yvelines, France

Abstract

We discuss the possibility of using multiple shift-invert Lanczos and contour integral based spectral projection method to compute a relatively large number of eigenvalues of a large sparse and symmetric matrix. The key to achieving high parallel efficiency in this type of computation is to divide the spectrum into several intervals in a way that leads to optimal use of computational resources. We discuss strategies for dividing the spectrum. Our strategies make use of an eigenvalue distribution profile that can be estimated through inertia counts and cubic spline fitting. We also make use of a simple cost model that describes the cost of computing \( k \) eigenvalues within a single interval in terms of the asymptotic cost of sparse matrix factorization and triangular substitutions. Several computational experiments are performed to demonstrate the effect of spectrum division on the overall performance of both multiple shift-invert Lanczos and the contour integral based method. We also show parallel scalability of both approaches in the strong and weak sense. Our experiments indicate that multiple shift-invert Lanczos is generally more efficient than the contour integral based spectral projection method when implemented properly.

Key words: parallel eigenvalue computations, spectral transformation, multiple shift-invert Lanczos, contour integral based spectral projection method, strong scaling, weak scaling.

1. Introduction

In a number of applications, we are interested in computing a small percentage of the eigenvalues of a large sparse matrix \( A \) or matrix pencil \((A, B)\) [2, 19]. Often these are eigenvalues at the low end of the spectrum.

When the dimension of the matrix \( n \) is above \( 10^9 \), for example, even one percent of \( n \) amounts to more than 10,000 eigenvalues. When that many eigenpairs are needed, many of the existing software packages such as PARPACK [15], Anasazi [3], BLOPEX [10], PRIMME[23], for solving large-scale eigenvalue problems, which are mostly based on extracting approximate eigenpairs from a single subspace (e.g, a \( k \)-dimensional Krylov subspace \( K(A, v_0) = \{v_0, Av_0, ..., A^{k-1}v_0\} \) associated with some starting vector \( v_0 \)) often do not provide an efficient way to solve the problem. This is true even when a vast amount of computational resource is available. Three major sources of the problems with these solvers are:

1. The amount of parallelism that can be exploited in a standard Krylov subspace method such as the Lanczos algorithm is limited. The main computational tasks involved in such an algorithm include sparse matrix operations, basis orthogonalization, and the Rayleigh-Ritz procedure for extracting desired eigenvalues from a subspace. For a general sparse matrix, it may be difficult to speed up the sparse matrix operations (such as sparse matrix vector multiplication or sparse matrix factorization) on more than hundreds of processors.

*Corresponding author

Email address: HMAktulga@lbl.gov (Hasan Metin Aktulga)
2. As the number of required eigenpairs increases, the dimension of the subspace required to extract approximate spectral information can become quite large. Hence the cost for constructing an orthonormal basis for such a subspace may grow even faster than that associated with performing sparse matrix vector multiplications.

3. The projected eigenvalue problem is often replicated on each processor and solved sequentially in existing solvers. Solving the projected problem can become a bottleneck when the number of desired eigenvalues is over a few thousands. Although this part of the computational can be parallelized through the use of ScaLAPACK[4], it often requires a nontrivial change to the data structure that becomes incompatible with a user defined sparse matrix operations. When the computation is carried out on a large number of processors, a large amount of communication overhead is expected. The overhead may degrade the parallel scalability of the computation.

New strategies and software tools must be developed to handle this type of calculations on the emerging distributed multi/many-core computing platforms. These strategies may depend on a number of factors such as the sparsity of the matrix, the distribution of the eigenvalues, and also the type of computational resources available to the user.

In this paper, we examine one of the strategies and provide some computational examples that illustrate the type of issues we must consider and how they may be resolved. The general idea we pursue is to divide the desired part of the spectrum into several subintervals and to compute eigenvalues within each interval in parallel. By adding a coarse grained level of parallelism at the top, this approach obviously increases the total amount of concurrency in computation. By limiting the size of each interval, it also mitigates the second and the third issues that currently prevent some existing eigensolvers from achieving high efficiency.

We will consider two methods for computing eigenvalues within a prescribed interval:

1. Shift-invert Lanczos (SIL) with implicit restart [12]
2. Subspace iteration applied to a projection operator constructed from an approximation to the following contour integral

\[ P = \frac{1}{2\pi i} \oint_{\Gamma} (A - zI)^{-1} dz, \]  

where \( \Gamma \) is a contour on the complex plane that encloses only the eigenvalues of interest [5]. We will refer to this approach as a contour integral based spectral projection method. Figure 1 (a) shows a typical contour \( \Gamma \) that encloses a number of eigenvalues marked by solid circles. The contour is numerically discretized into several quadrature points. Figure 1 (b) shows the value of the discretized projection operator associated with such a contour, which maps eigenvalues within the contour to 1 and eigenvalues outside of the contour to 0, with certain amount of smearing near the boundary of the circle due to finite discretization. This approach has been discussed in [20, 21, 17, 26].

We will refer to the combination of using spectrum division and SIL as **multiple shift-invert Lanczos** (MSIL) in subsequent discussions. The use of spectrum division and subspace iterations on spectral projectors constructed from the contour integral formulation will be referred to as **multiple contour integral based spectral projection method** (MCISPM). Both algorithms require solving a number of linear systems of equations of the form \( (A - zI)x = b \), where \( z \) is either a complex or a real shift. Although these equations can also be solved by preconditioned iterative methods such as MINRES[16], GMRES[18], BiCGSTAB [25] and QMR [7], the operator \( A - zI \) is in general not positive definite even when \( z \) is chosen to be on the real axis as in MSIL. As a result it can be difficult to keep the number of iterations under control using iterative methods. In this paper, we will focus on direct methods that perform sparse \( LDL^T \) factorizations of \( A - zI \) and sparse back substitutions using \( L \) and \( D \) factors.

Although spectrum division is conceptually simple, implementing this strategy efficiently on a large-scale parallel computer is not straightforward. The key to achieving high efficiency is to develop a way to divide the spectrum so that computational resources can be utilized in an optimal way (i.e., to maximize the overall throughput of the computation.) An optimal spectrum division scheme should place an appropriate number of eigenvalues in each interval to keep the cost per interval relatively low, while making sure the overall
Figure 1: Sketch of the contour and the associated projection operator used by the spectral projection method.

computation is load balanced and can be completed quickly. Developing such a scheme requires a careful analysis of the computational cost associated with each interval, which depends on the cost for performing an $LDL^T$ factorization and a number of triangular substitutions as well as the number of eigenvalues and the convergence rate of these eigenvalues within either an SIL or subspace iteration.

When a large amount of computational resource is available, the optimal strategy may be different from one that is designed for a computational environment with a small limited amount of computing power. In both cases, one has to decide how much computational resource one should allocate to compute eigenvalues within one interval. Using all computational resources on one interval at a time is often not the most efficient strategy because neither a sparse $LDL^T$ factorization nor a triangular substitution can scale beyond a few hundred processors. It is usually much more efficient to allocate fewer processors to one interval and compute eigenvalues in several intervals in parallel.

Although both MISL and the MCISPM require solving linear systems, there are a number of notable differences between these two approaches for computing eigenvalues within each interval. The most important difference is perhaps the fact that linear equations must be solved in sequence in SIL, whereas more parallelism may be exploited in a subspace iteration when multiple equations are solved simultaneously. Thus, it may appear that better scalability can be expected from a subspace iteration method. However, when the lower triangular factor of $A$ must be distributed among different processors, it is actually not easy to take advantage of multiple right hand sides to achieve full concurrency in triangular substitutions. Therefore, the actual parallel efficiency may be far more limited, as we will show in our experiments.

One drawback of a MCISPM is that the construction of a projection operator requires multiple factorizations to be performed in complex arithmetic. Although these factorizations can be performed in an embarrassingly parallel way, each factorization is more expensive than the real arithmetic factorizations required in the SIL approach, and as a whole these factorizations consume more computational resource than what is required in MSIL for computing a comparable number of eigenvalues.

In terms of memory usage, MCISPM is also more costly than MSIL. This is partly because the factor $L$ is a complex matrix in the contour integral based approach. Even though the matrix $A$ is sparse, the space required for storing the factor $L$ can be significant due to fill-ins during factorization. In addition, several (8 to 16) factorizations are required to compute eigenvalues within each interval. To perform them
in parallel, additional (distributed) memory is required. The increased memory requirements of MCISPM can be a limiting factor in terms of performance as well.

The use of MSIL was considered in the work of [27]. The focus of their work is to use shift-invert with implicitly restarted Lanczos (IRL) to probe the spectrum dynamically as eigenvalues are being computed. The idea of using parallel contour integral based spectral projectors has been studied in [17, 20, 21, 26]. However, the authors focused mainly on how to use the contour integral approach to compute eigenvalues within a single interval, and left the task of spectrum division to users. They did not discuss how to organize the entire computation in an efficient manner. In [26], the authors’ comparison of the performance a parallel contour integral based spectral projection method with that of a single shift-invert Lanczos for computing 40 eigenvalues in a single interval is somewhat misleading. We believe that a proper comparison should be made with respect to the use of MSIL.

In this paper, we focus on the issue of spectrum partitioning, which we believe is critical for achieving high efficiency regardless of whether a shift-invert IRL or a contour integral based spectral projection method is used to compute eigenvalues within each interval. We will list a few objectives of spectrum division in section 2, and discuss how the distribution of eigenvalues can be estimated through an inertia count and cubic spline fitting. The eigenvalue distribution profile enables us to obtain an optimal strategy for spectrum division. We present a simple cost model that can be used to develop some general strategies for partitioning the spectrum and allocating computational resources in an efficient way. A few implementation details of MSIL and MCISPM are discussed in section 3. In section 4, we present a number of examples that illustrate the effect of different spectrum division schemes on the overall performance of the eigenvalue computational for a number of examples. We show that MSIL outperforms MCISPM on all our test problems with different sizes and sparsity patterns. We also discuss how spectrum division affects the strong and weak scaling properties of both MSIL and MCISPM.

2. Spectrum division

Although it is clear that dividing the spectrum into a number of smaller intervals will increase concurrency and reduce the cost of both orthogonalization and Rayleigh-Ritz calculation, it is less obvious how the division should be done to achieve optimal overall computational efficiency.

We will first discuss the objectives we would like to achieve by dividing the spectrum in section 2.1. Achieving some of these objectives requires us to have some knowledge about the distribution of eigenvalues. Such a distribution can be described in terms of a histogram of eigenvalues or, borrowing a terminology from physics, the *density of states* (DOS). We discuss how to obtain DOS in section 2.2. If the distribution of eigenvalues within the desired part of the spectrum (with spectral gaps excluded) is nearly uniform, a good strategy for dividing the spectrum may be developed by examining a simpler cost model that depends on the size of the problem, the total number of processors available, and the relative cost of a factorization versus a triangular substitution. We discuss this model and general strategies for dividing the spectrum in section 2.3.

2.1. Objectives

In general, a spectrum division scheme should try to achieve the following objectives:

1. The eigenvalues within each interval should be computed in an efficient manner by a group of processors. This objective suggests that we should exclude spectral gaps from each interval because the presence of such a gap often slows down the convergence of both the shift-invert Lanczos algorithm and the subspace iteration applied to the contour integral representation of the projector. It also suggests that we should limit the number of eigenvalues in each interval to keep the cost of orthogonalization and Rayleigh-Ritz calculation under control.

2. The division should make it easy to group the processors in such a way that allows desired eigenvalues to be computed in a load balanced way.
3. Since factorization is expensive, the spectrum should be divided in a way that minimizes the total number of factorizations.

Some of these objectives may conflict with each other. There may be tradeoffs among different objectives to optimize resource usage. The most important objective we should try to achieve may depend also on the type of available computational resource. For example, dividing the spectrum into fewer intervals and allocating more processors to each interval will reduce the cost of factorizations (objective 3). However, because each interval contains more eigenvalues, the cost of basis orthogonalization and Rayleigh-Ritz calculation may become excessively high. Furthermore, the convergence of some of the eigenvalues within each interval may become slow, for example, in a SIL calculation. As a result, more triangular substitutions may be required in each interval to reach convergence, making it difficult to achieve the first objective.

2.2. Density of States Estimation

It follows from the Sylvester’s inertia theorem [24] that the number of eigenvalues of $A$ that are within an interval $[\alpha, \beta]$ can be determined by performing two factorizations on the shifted matrices: $A - \alpha I = L_\alpha D_\alpha L_\alpha^T$ and $A - \beta I = L_\beta D_\beta L_\beta^T$. The difference between the number of negatives entries in $D_\beta$ and $D_\alpha$ yields the eigenvalue count within $[\alpha, \beta]$.

Consequently, an estimation of the density of states (DOS) can be obtained by partitioning the spectrum into a number of intervals and performing a number of $LDL^T$ factorizations. The partitioning can be done by either estimating the upper and lower bounds of the desired part of the spectrum $\lambda_{\text{ub}}$ and $\lambda_{\text{lb}}$ respectively first and dividing $[\lambda_{\text{lb}}, \lambda_{\text{ub}}]$ evenly into $s$ intervals or running an $s$-step Lanczos iteration first and using $s$ Ritz values $\theta_1, \theta_2, \ldots, \theta_s$ returned from the Lanczos procedure to partition the spectrum and construct a DOS profile. In both cases, $s$ can be chosen to be a relatively small constant, say 100 first. When a large amount of computational resource is available, one can perform multiple parallel factorizations simultaneously.

Furthermore, we may use cubic spline fitting to construct a high resolution cumulative density of state profile from which a refined DOS can be constructed [6]. A cumulative density of states function $c(\lambda)$ gives the total number of eigenvalues less than or equal to $\lambda$. By counting the number of eigenvalues to the left of $\theta_i$, for $i = 1, 2, \ldots, s$, we can construct a crude approximation to the true cumulative density of the spectrum. The crude approximation can be improved by connecting $(\theta_i, c(\theta_i))$ with $(\theta_{i+1}, c(\theta_{i+1}))$ using cubic splines. Figure 2 shows that the approximate cumulative density of states function constructed from 100 inertia counts and cubic spline fitting is nearly indistinguishable from the exact cumulative density of states. We observe from the cubic spline fitting that the matrix used in this example has a spectral gap between -2.0 and -1.7, and the distribution of eigenvalues is nearly uniform below and above the spectral gap.

![Figure 2](image)

Figure 2: The approximate cumulative density function of the first 2048 eigenvalues of a sparse matrix (red) constructed from 100 inertial counts and cubic spline fitting is nearly indistinguishable from the exact one (blue).
The cubic spline fitting of the cumulative density provides a reliable way to evaluate \( c(\lambda) \) and its derivative for any \( \lambda \) in the region where the DOS is relatively smooth. Consequently, we can find the approximate location of the \( j \)th eigenvalue by solving \( c(\lambda_j) = j \) using either Newton’s method or the bisection method. This capability allows us to define the upper and lower bounds of each interval regardless how the spectrum is partitioned.

2.3. Strategy

We now discuss how to achieve some of the objectives listed in section 2.1. To facilitate our discussion, we list some of the notation used throughout this paper to describe various quantities related to the size of the problem as well as the size and configuration of the computing platform in Table 1.

Table 1: Notation used to describe the problems as well as the configuration of the computing platform.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>matrix size</td>
</tr>
<tr>
<td>( nev )</td>
<td>the total number of desired eigenvalues</td>
</tr>
<tr>
<td>( p )</td>
<td>the total number of processors (cores) available</td>
</tr>
<tr>
<td>( l )</td>
<td>the number of intervals into which the spectrum is divided</td>
</tr>
<tr>
<td>( k )</td>
<td>the average number of eigenvalues per interval (( =nev/l ))</td>
</tr>
<tr>
<td>( q )</td>
<td>the number of processors (cores) used per interval (( =p/l ))</td>
</tr>
</tbody>
</table>

We assume that we have access to a relatively large but not infinite number of processors. In the latter case, we may divide the spectrum into \( nev \) intervals, where \( nev \) is the total number of eigenvalues of interest, and assign as many processors to each interval as we possibly can without slowing down the computation. This situation is neither interesting nor practical.

There are a number of ways to achieve the objective of load balance. One way is to keep the granularity of the computation sufficiently fine by partitioning the spectrum into many small intervals and using a relatively large number of processors to compute eigenvalues within each interval. In this case, we may have more intervals than the number of processor groups. Each processor group may be responsible for computing eigenvalues in multiple intervals. If eigenvalues within each interval can be computed quickly, then good load balance can be expected from fine grained parallelism.

However, there are a couple of issues associated with this approach. First of all, because there is a limit on how far a sparse \( LDL^T \) factorization and a triangular substitution can scale with respect to the number of processors, the granularity of each parallel task cannot be made arbitrarily small.

Secondly, it is generally more costly to partition a task into \( m \) subtasks and assign all processors to work on each subtask, one after another, than to divide the processors into \( m \) groups, with each group responsible for each subtask, unless the parallel efficiency of having all processors work on each subtask is perfect. To see why this is the case, let us assume the total amount of work required to compute eigenvalues within an interval is \( C \), and the wall clock time required to compute these eigenvalues using \( p \) processors is proportional to \( C/p^\eta \), where the parameter \( 0 < \eta < 1 \) is used to account for parallel efficiency. If we now divide the interval into \( m \) subintervals and use all \( p \) processors to compute \( 1/m \) of the eigenvalues at a time, and repeat the process \( m \) times, the wall clock time is still proportional to \( C/p^\eta \) if the parallel efficiency does not change with respect to the number of eigenvalues to be computed. However, if we use \( p/m \) processors to compute eigenvalues within each one of the \( m \) subintervals with the same parallel efficiency, the wall clock time is proportional to

\[
\frac{C/m}{(p/m)^\eta} = \frac{C}{p^\eta} m^{\eta - 1} < \frac{C}{p^\eta}.
\]

Therefore, it is generally more effective to partition a group of processors in the same way the interval is partitioned so that eigenvalues within each subinterval are computed by a subgroup of processors.
This observation suggests that a better way to achieve load balance is to partition the spectrum in such a way that the amount of work associated with each interval is roughly the same. In this case, the processors can be divided into the same number of groups, with each group responsible for computing eigenvalues within each subinterval. This approach may not be feasible if the number of available processors is relatively small compared to the number of eigenvalues to be computed. However, this is not the regime that we are primarily concerned with in this paper. Some difficulties may also arise when the eigenvalues of interest are not evenly distributed. In this case, the spectrum may be divided in such a way that the amount of work associated with different intervals is not exactly the same. As a result, the grouping of the processors need to be adjusted accordingly so that it is consistent with the amount of work associated with each interval. Making this adjustment is possible when the DOS profile is available.

Although it is generally not easy to answer the question of how to partition the desired part of the spectrum, and how many processors we should allocate to each interval in order to optimize the use of computational resources, a reasonable strategy may be developed if we can characterize the cost using a simple model. Using such a model, we can then analyze how other objectives listed in section 2.1 may be achieved and how the effectiveness of a partition may be affected by a number of factors such as the size of the problem, the relative cost and efficiency of factorization versus triangular substitution, and the number of processors available.

To simplify our analysis, let us assume eigenvalues are uniformly distributed in the desired part of the spectrum. Using the notation in Table 1, we assume $p$ and $nev$ are relatively large. To be more precise, let us assume that $p = c_p n$ and $nev = c_n n$, where $c_p$ and $c_n$ are constants. We assume that the number of eigenvalues within each interval $k$ is sufficiently small so that the cost of computing eigenvalues by either the shift-invert Lanczos method or the contour integral based projection method is dominated by sparse matrix factorizations and triangular substitutions.

The wall-clock time for performing a factorization on a single processor can be estimated by $c_f n^{\alpha_f}$ for some constant $c_f$ and $\alpha_f$. The wall clock time for a single triangular substitution performed on a single processor can be modeled as $c_s n^{\alpha_s}$ for some constant $c_s$ and $\alpha_s$. The constants $c_f, c_s, \alpha_f, \alpha_s$ are problem and/or machine dependent. The matrices used in the numerical experiments in section 4 mainly originate from the density functional theory. The sparsity of these matrices are similar to that of the discretized Laplacian operator. The discretized operators that share a sparsity pattern similar to the discretized Laplacian operator also appear frequently in other scientific computing problems such as the Poisson equation and the Helmholtz equation. For $d$-dimensional Laplacian type of operators discretized by finite difference and ordered by the nested dissection technique \[8\], the values of $\alpha_f, \alpha_s$ are given in the second and the third columns of Table 2. For other general sparse matrices, the values of $\alpha_f$ and $\alpha_s$ will depend on the number of nonzeros in the factor $L$, which is in turn determined by matrix ordering scheme (e.g. \[9\]) used to permute the rows and columns of $A$ to minimize the amount of nonzero fills in $L$.

Table 2: Asymptotic wall clock time using one processor for the factorization cost $\alpha_f$, the triangular solve cost $\alpha_s$, and the optimal number of eigenvalues $k^{opt}$ for a Laplacian operator discretized by second order finite difference method in dimension $d$.

<table>
<thead>
<tr>
<th>$d$</th>
<th>$\alpha_f$</th>
<th>$\alpha_s$</th>
<th>$k^{opt}$ ($\eta_f = \eta_s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$O(n^1)$</td>
</tr>
<tr>
<td>2</td>
<td>$\frac{3}{2}$</td>
<td>1</td>
<td>$O(n^{\frac{3}{2}})$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$\frac{4}{3}$</td>
<td>$O(n^{\frac{4}{3}})$</td>
</tr>
</tbody>
</table>

We model the parallel efficiency for the factorization and the triangular substitution by two parameters $\eta_f, \eta_s$ respectively. These parameters satisfy $\eta_f, \eta_s \in (0, 1)$, and usually $\eta_f \geq \eta_s$. When $p$ processors are used to perform a factorization and a triangular substitution, the wall clock time required is proportional to $c_f n^{\alpha_f}/p^{\eta_f}$ and $c_s n^{\alpha_s}/p^{\eta_s}$, respectively. To simplify our discussion, we assume that the range of $k$ we consider corresponds to cases in which the cost associated with orthogonalization and Rayleigh Ritz calculation is relatively low compared to that associated with factorization and triangular substitutions. In addition, we
assume that the number of triangular substitutions used per eigenvalue is constant across the range of $k$ under consideration.

Given the assumptions above, the wall clock time $W(k)$ of the calculation can be expressed by

$$W(k) = \frac{c_f n^\alpha_f}{\eta_f} + \frac{c_s n^\alpha_s k}{\eta_s}. \quad (2)$$

Minimizing $W(k)$ with respect to $k$ gives rise to

$$k_{\text{opt}} = \left( \frac{\eta_f c_f}{(1 - \eta_s) c_s} \right)^{1/(\eta_f - \eta_s)} \left( \frac{c_n}{c_p} \right)^{\eta_f - \eta_s} n^{\alpha_f - \alpha_s}. \quad (3)$$

Although it is possible to include extra terms in (2) to account for the cost of orthogonalization and Rayleigh-Ritz calculation, introducing these terms makes it difficult to obtain an analytic expression for $k_{\text{opt}}$. Because we assume the cost of orthogonalization and Rayleigh-Ritz calculation is relatively low in the range of $k$ we consider, the true minimizer for the more sophisticated model is likely to be quite close to the solution given by (3).

The expression given by (3) indicates that choice of an optimal $k$ depends on the problem size $n$ as well as the relative cost and parallel efficiency of factorization and triangular substitution, when $c_n/c_p$ is a constant. When the parallel efficiencies of both the factorization and the triangular substitution are high, i.e., when both $\eta_f$ and $\eta_s$ are close to 1, we should place more eigenvalues in each interval without increasing the cost of orthogonalization and Rayleigh-Ritz calculation too much. When $\eta_f$ and $\eta_s$ are relatively small, $k_{\text{opt}}$ will be smaller as well. A smaller $k_{\text{opt}}$ would suggest dividing the spectrum into more intervals and breaking the processors into smaller groups. However, there is often a lower bound on the number of processors in each group due to the memory requirement for holding the lower triangular factor of $A - \sigma I$ or $A - z_i I$. We should also point out that the parallel efficiency parameters $\eta_f$ and $\eta_s$ depend on a nontrivial way on the actual number processors used in the factorization and triangular substitutions, which is ultimately related to $c_p$. When the number of processors used in each factorization or triangular substitution increases, both $\eta_f$ and $\eta_s$ may decrease. Such loss of parallel efficiency implies that we need to divide the spectrum into more intervals with fewer eigenvalues in each interval as the total number of processors increases.

While in general factorization is expected to scale better than a triangular substitution (e.g., $\eta_f > \eta_s$), our numerical experiments show that when $c_p$ is not too large (e.g., $c_p = 0.1$), $\eta_f \approx \eta_s$, and they do not depend strongly on $c_p$. This can be explained by the additional overheads incurred during the sparse matrix factorization phase, such as the analysis step necessary to determine a good pivoting strategy and the distribution of the sparse matrix over the processor group. By taking $\eta_f \approx \eta_s$, Eq. (3) can be simplified, and the minimizer of (3) has the form

$$k_{\text{opt}} = \left( \frac{\eta_f c_f}{(1 - \eta) c_s} \right)n^{\alpha_f - \alpha_s}, \quad (4)$$

where $\eta = \eta_f = \eta_s$. It is interesting to see that in this case, $k_{\text{opt}}$ is independent of $c_p$ which determines the total number of processors used in the computation, and we should put more eigenvalues in each interval as the problem size $n$ increases. Furthermore, for Laplacian type operators defined in $d$-dimensional space, the optimal number of eigenvalues to place in each interval also depends on $d$ because both $\alpha_f$ and $\alpha_s$ are functions of $d$ in this case. The fourth column of Table 2 gives the dependency of $k_{\text{opt}}$ on $n$ for the Laplacian type of operators in different spatial dimensions.

It is easy to show that when the factorization and triangular substitution have roughly the same parallel efficiency, the wall clock time associated with $k_{\text{opt}}$ is

$$W(k_{\text{opt}}) = \frac{c_f n^\alpha_f}{u} + \frac{\eta}{1 - \eta} \frac{c_f n^\alpha_f}{u} \quad (5)$$
where \( u = (pk_{\text{opt}}/nev)^{\eta} \). The expression given by (5) indicates that the first and second terms of \( W \) are roughly the same at \( k_{\text{opt}} \) unless \( \eta \) is either close to 1 or 0. If \( \eta = 0.5 \), the second term in (5) is identical to the first term. When \( k \) is large, the second term in \( W(k) \) can be much larger than the first term, which is usually an indication that such a \( k \) is not optimal.

The cost model given by (2) allows us to make some prediction on how the MSIL algorithm and MCISPM scale in a weak sense. We now give an example in which both the problem size \( n \) and the number of eigenvalues \( nev \), as well as the total number of processors \( p \) increase by a factor of 4.

If we increase the number of processors per interval \( q \) by a factor of 4 without changing the number of intervals \( l \), the number of eigenvalues per interval \( k \) is also increased by a factor of 4 (due to the increase of the problem size.) It follows from (2) that the total wall clock time for factorization will increase by a factor of \( 4^{\alpha_f-\eta_f} \) and the wall clock time for triangular substitution will increase by a factor of \( 4^{\alpha_s-\eta_s+1} \). If we assume \( \eta_f = \eta_s = 1/2 \) and the costs associated with factorization and triangular substitutions are roughly the same before \( q \) and \( k \) are increased, then for 2D problems, we expect the total wall clock time to increase by a factor

\[
\frac{4^{\alpha_f-\eta_f} + 4^{\alpha_s-\eta_s+1}}{2} = \frac{4^2 + 4^{3/2}}{2} = 12.
\]

If we keep the number of cores per interval \( q \) and the number of eigenvalues per interval \( k \) fixed, and simply increase the number of intervals \( l \) by a factor of 4, then based on the model (2), we expect the total wall clock time to be increased by a factor of

\[
(4^{\alpha_f} + 4^{\alpha_s})/2,
\]

if the same assumptions about the relative cost of factorization and triangular substitutions before \( n \) and \( p \) are increased are made. For 2D problems, this amounts to an increase by a factor of 6.

The same type of calculation shows that, under the same assumption about factorization and triangular substitutions, if we increase both \( k \) and \( q \) by a factor of 2, the total wall clock time is expected to increase by factor of 5.6.

This simple analysis shows that when using a larger number of processors to solve larger problems, it may be beneficial to split the spectrum into more intervals and putting more processors into each interval at the same time. This analysis is confirmed by our computational experiments which we show in section 4.

### 3. Implementation issues

In this section, we discuss a number of implementation issues pertaining to both the MSIL algorithm and the MCISPM.

#### 3.1. Multiple Shift-Invert Lanczos

Shift-invert is a widely used technique for computing interior eigenvalues when using the Lanczos algorithm. Because eigenvalues near a target shift \( \sigma \) are mapped to the extremal eigenvalues of the shifted and inverted matrix \((A - \sigma I)^{-1}\), through the equation

\[
(A - \sigma I)^{-1}x = \frac{1}{\lambda - \sigma}x,
\]

where \((\lambda, x)\) is an eigenpair of \( A \), the eigenvectors associated with these eigenvalues tend to converge rapidly when we apply the Lanczos procedure to \((A - \sigma I)^{-1}\).

When the desired part of the spectrum is divided into several intervals, we can apply the shift-invert Lanczos procedure in each interval simultaneously with an appropriately chosen shift so that eigenvalues within the interval can be computed quickly.

As discussed earlier, the partitioning of the spectrum can be obtained by first estimating the density of state profile using the technique described above, and using the simple model presented in section 2.3 to decide how many intervals we should create with an appropriate number of eigenvalues in each interval.
Such a decision may require us to perform some computational experiments (often referred to as automatic tuning in the high performance computing community), to estimate various parameters in the model. We should also avoid defining an interval that contains a sizable spectral gap even if that means we may end up with some intervals that contain fewer eigenvalues. This approach is very different from the one taken in [27], where each processor determines how many eigenvalues to compute in a dynamic way.

If the total number of processors available \( p \) is sufficiently large, which is the case that we are primarily concerned with, and if the eigenvalues are uniformly distributed, we can allocate \( p/l \) processors to each interval once the total number of intervals \( l \) is chosen.

In most cases, the target shift for each shift-invert Lanczos (SIL) run can be set as the mid-point of each interval. We perform at least one parallel \( LDL^T \) factorization in each interval.

One of the difficulties with using the Lanczos algorithm is that it is difficult to predict a priori how many Lanczos iterations it takes to obtain all eigenvalues within each interval. Even if the number of eigenvalues \( k \) within each interval may be small, the dimension of the Krylov subspace generated by the standard Lanczos algorithm may be much larger.

This problem can be partially solved by using the implicitly restart technique [22] to limit the dimension of the Krylov subspace from which approximate eigenvalues of \( (A - \sigma I)^{-1} \) or \( A \) are extracted. Limiting the dimension of the Krylov subspace reduces the memory footprint of the calculation. It also prevents the cost of Rayleigh-Ritz procedure from becoming too high.

In an implicitly restarted Lanczos (IRL) procedure, we typically construct and update a Krylov subspace whose dimension is slightly larger than the number of desired eigenvalues \( k \). In our experiments, we typically use \( 2k \) as the maximum subspace dimension. We perform \( 2k \) triangular substitutions in the first shift-invert IRL cycle. In each subsequent IRL cycle, we perform \( k \) additional triangular substitutions.

We set a limit on the number of restart cycles in each shift-invert IRL run, so that the total number and cost of triangular substitutions do not become too high. We also set \( k \) to be slightly larger than the actual number of eigenvalues contained in the interval with the hope that we do not miss any eigenvalues within the interval. However, it is possible that some of the desired eigenvalues fail to converge before the limit on the restart cycles is reached. In that case, we may need to perform a post-processing procedure to capture the missing eigenpairs. Such a procedure would require an additional factorization and a few more triangular substitutions. Hence, in that case, the entire computation may not be well load balanced.

### 3.2. Contour integral based projection method

If \( P \) is an orthogonal projection operator that projects onto the invariant subspace \( Z \) associated with a subset of \( k \) eigenvalues of \( A \), we may compute this invariant subspace by applying a subspace iteration to \( P \) (see Algorithm 1).

**Algorithm 1 Subspace iteration**

- **Input:** projector \( P \), random \( k \)-dimensional subspace \( V \);
- **Output:** an invariant subspace \( Z \);
- **while** not converged **do**
  - \( W \leftarrow PV \);
  - Perform a QR factorization of \( W \) to obtain an orthonormal basis \( V \)
- **end while**
- \( Z = V \);

Let \( V \) be a random \( k \)-dimensional subspace that contains contributions from all eigenvectors of \( A \). In exact arithmetic, we need to apply \( P \) to \( V \) only once in order to obtain the desired the invariant subspace \( Z \), if the exact \( P \) is available. In practice, because the projection operator \( P \) given by (1) is approximated
by a numerical quadrature

\[ P \approx \sum_{i=1}^{q} \omega_i (A - z_i I)^{-1}, \]  

for some appropriately chosen poles \( z_i \) and weights \( \omega_i \) \( (i = 1, 2, ..., q) \), more than one iteration may be needed to obtain \( \tilde{Z} \) with sufficient accuracy.

In the numerical experiments that we show in section 4, we use the quadrature rule proposed in [17] to evaluate the contour integral (1). In most cases, sixteen quadrature points (poles) are sufficient. For each pole \( z_i \), we must perform an \( LDL^T \) factorization of \( A - z_i I \) in complex arithmetic. This is roughly four times more costly than an \( LDL^T \) factorization performed in SIL. The higher factorization cost would suggest that a larger number of processors should be allocated to each interval (or contour) to ensure that each factorization can be computed efficiently in parallel. These processors are further partitioned among \( n_{\text{pole}} \) poles. As a result, the number of processor groups and, consequently, the number of intervals \( l \) will decrease, and we need to place more eigenvalues in each interval. However, placing too many eigenvalues within a single interval will lead to a higher cost due to triangular substitutions. Since \( (A - z_i I)^{-1} \) must be applied to all columns of \( V \) at each pole \( z_i \), we must perform \( k \) triangular substitutions at each pole. Such increase must be taken into account when we decide what the optimal \( k \) should be.

Although it may appear that performing multiple triangular substitutions is an inherently parallel task, the fact that the triangular factors of \( A - z_i I \) are distributed makes it difficult to achieve a high level of parallelism during triangular substitutions, even when a large number of right hand sides are available. Some gain in efficiency can be achieved by making use of level-3 BLAS with an appropriate block size when multiple substitutions are performed simultaneously. However, there is a limit on how large this type of gain can be. It typically does not exceed a factor of 3, and is quickly offset by a large increase in the number of right hand sides.

We observe in our experiments that the overall computation becomes less efficient if the cost of triangular substitutions far exceeds that associated with factorization.

Another drawback of having too many eigenvalues within a single interval (or contour) is that more subspace iterations may be required to reach convergence. This is due to the fact that the interval enclosed by the contour may be too large for a 8 or 16-point quadrature to construct an accurate approximation to the projection operator (1). A larger number of subspace iterations will lead to an increase in the cost of triangular substitutions, and hence, the average cost for computing an eigenpair.

4. Computational Experiments

In this section, we present a number of computational experiments that demonstrate the efficiency that can be achieved by both MSIL and MCISPM. These examples also illustrate the importance of the issues we discussed in the previous section and how one may achieve nearly optimal performance by making a proper choice of interval size and processor groups.

A number of medium sized sparse matrices are used in our experiments. The matrices are listed in Table 3. The Graphene\( X \) matrices are obtained from a discretization of the Kohn-Sham equation using an adaptive local basis expansion scheme [14]. The number \( X \) indicates the number of carbon atoms in the graphene. The dimension of these matrices is proportional to the number of atoms in these quasi-2D materials. All other matrices are obtained from a finite difference discretization of the Kohn-Sham equation for 3D materials. They are produced by the PARSEC software package [11], and tend to be sparser than the Graphene matrices. However, as we can see from Table 3, their \( L \)-factors tend to be slightly denser than those associated with the Graphene problems.

We performed our numerical experiments on Hopper, a Cray XE6 supercomputer maintained at the National Energy Research Scientific Computer Center (NERSC) at Lawrence Berkeley National Laboratory. Each node on Hopper has 2 twelve-core AMD Magny Cours 2.1-GHz processors with a total of 32 gigabyte (GB) shared memory. However, memory access bandwidth and latency are nonuniform across all cores. Each core has its own 64 kilobytes (KB) L1 and 512 KB L2 caches. One 6 MB L3 cache is shared among
6 cores on the Magny Cours processor. There are four DDR3 1333-MHz memory channels per twelve-core Magny Cours processor.

The PARPACK software package [13, 15] is used for MSIL experiments. We use the FEAST program developed in [17] to test the performance of MCISPM. We set the convergence tolerance $\text{tol}$ to $10^{-10}$ in both experiments, i.e., we terminate SIL or a subspace iteration when the relative residual norm of an approximate eigenpair $(\theta_i, z_i)$, defined by $\|Az_i - \theta_i z_i\|/\|\theta_i\|$ falls below $\text{tol}$ for all $i = 1, 2, ..., \text{nev}$.

We use the MUMPS (Multifrontal Massively Parallel sparse Direct Solver) [1] to solve the linear system of equations in both MSIL and MCISPM. We should note that the performance of the sparse solver has a direct effect on the absolute performance of the eigenvalue calculation. However, some of the issues we will discuss below is independent of the sparse solver used in the computation.

4.1. MSIL

In this section, we study the parallel performance of MSIL when it is used to compute roughly 10% of the lowest eigenvalues of matrices given in Table 3. We first show that the way the spectrum is partitioned and processors are grouped (i.e. the choice of an appropriate number of eigenvalues $k$ for each interval) has a significant impact on the performance of MSIL. We then present the results of a strong scaling study, where we examine the performance of MSIL as we increase the number of processors $p$ for a given problem. Finally, in the weak scaling study, which is performed using the Graphene matrices only, we investigate the performance characteristics of MSIL when the number of processors is increased at the same rate as the increase in matrix dimensions.

4.1.1. Spectrum division and processor groups

As we discussed earlier, for a given problem and a fixed number of processors $p$, the performance of MSIL depends on how the spectrum is divided and how many eigenvalues are placed in each interval. In Figure 3, we show how the performance of MSIL changes as we change the way the desired part of the spectrum of the Graphene512 problem is partitioned. In this test, we used a total of $p = 512$ cores to compute $\text{nev} = 2048$ eigenvalues. As the spectrum is partitioned into $l$ intervals with roughly $k = \text{nev}/l$ eigenvalues within each interval, the computational cores are divided into $l$ groups with roughly $q = 512/l$ cores in each group.

The height of each color coded bar in Figure 3 gives the total wall clock time used to compute all desired eigenvalues for a particular combination of $l$ and $q$ that satisfy $l \times q \approx 512$. The $l \times q$ combination associated with each bar is labeled on the horizontal axis. The blue portion of each bar shows the portion of the wall-clock time spent in factorization. The red portion represents the average wall clock time spent in triangular substitutions. Because the number of IRL restart cycles used in different intervals may be slightly different, there is some difference in the actual number of triangular substitutions performed in different intervals. The difference between the maximum and average amount of wall clock time spent in triangular substitutions over all intervals is shown by the green portion of each bar. This portion measures how well the computation is load balanced.

Although the cost model and analysis given in section 2.3 provide some guidance on how $k$ should be chosen to minimize the total wall clock time, the actual value of $k_{\text{opt}}$ depends on a number of architecture

<table>
<thead>
<tr>
<th>Name</th>
<th>$n$</th>
<th>$\text{nnz}$</th>
<th>$L_{\text{nnz}}$</th>
<th>factor ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphene128</td>
<td>5,120</td>
<td>1,026,560</td>
<td>3,100,160</td>
<td>$2.56 \times 10^8 (\approx 97n^2)$</td>
</tr>
<tr>
<td>Graphene512</td>
<td>20,480</td>
<td>4,106,240</td>
<td>135,445,760</td>
<td>$1.2 \times 10^{11}(\approx 267n^2)$</td>
</tr>
<tr>
<td>Graphene2048</td>
<td>81,920</td>
<td>16,424,960</td>
<td>5.35 $\times 10^{11}(\approx 79n^2)$</td>
<td></td>
</tr>
<tr>
<td>Graphene8192</td>
<td>327,680</td>
<td>65,699,840</td>
<td>727,511,040</td>
<td>$6.42 \times 10^{12}(\approx 60n^2)$</td>
</tr>
<tr>
<td>C60</td>
<td>17,576</td>
<td>212,390</td>
<td>19,573,813</td>
<td>$3.74 \times 10^{10}(\approx 121n^2)$</td>
</tr>
<tr>
<td>GaAsH6</td>
<td>61,349</td>
<td>3,016,148</td>
<td>247,861,525</td>
<td>$1.73 \times 10^{12}(\approx 460n^2)$</td>
</tr>
<tr>
<td>SiO</td>
<td>33,401</td>
<td>675,528</td>
<td>89,796,599</td>
<td>$4.03 \times 10^{11}(\approx 361n^2)$</td>
</tr>
</tbody>
</table>
and sparse solver dependent parameters such as $c_f$, $c_s$, $\eta_f$ and $\eta_s$. We observe in Figure 3 that by creating $l = 128$ intervals with $k \approx 16$ eigenvalues within each interval, the factorization, which is performed by $q = 4$ cores in parallel, is much more costly than triangular substitutions. As a result, the total wall clock time is far from optimal. On the other hand, when only $l = 8$ intervals are created with $k = 256$ eigenvalues per interval, an excessive amount of time is spent in triangular substitutions. This combination leads to a sharp increase in the total wall clock time also.

The minimum wall-clock time is achieved when $l = 25$ and $q = 20$. In this case, the difference between the maximum and average times spent on triangular substitutions is relatively small indicating that the computation is reasonably well load balanced, and the wall-clock times used in factorization and triangular substitutions are roughly the same also. This observation agrees well with our prediction made in section 2.3.

For other $l \times q$ combinations, the difference between the maximum and average times spent on triangular substitutions can be large. This indicates one of the challenges of MSIL, which is to ensure that eigenvalues in different intervals converge at approximately the same rate.

4.1.2. Strong scaling
In this section, we examine how efficiently MSIL can utilize an increasing number of processors for a given problem. Since our model indicates that the wall clock time of MSIL depends on how the spectrum is partitioned as well as the relative cost and parallel efficiency of factorization and triangular substitutions, which are themselves a function of the number of processors used, performing a strong scaling study is not straightforward.

Figure 4 shows the strong scaling of MSIL for Graphene matrices, which originates from quasi-2D problems. For each problem, we plot the best total wall clock time against the total number of processors used in the computation, where the best wallclock time is chosen from various $l \times q \approx p$ combinations for all $p$’s. We label each data point by the optimal $l \times q$ combination associated with the best measured wall clock time. While the points on the dashed lines indicate the maximum wall-clock time taken by any processor group in the best combination, the points on the solid line indicate the average wall-clock time taken by all processor groups.

The general trend we observe is that as the number of processors increases for a given problem, there is an increase both in the number of intervals $l$ used and in the number of processors $q$ assigned to each interval. Intuitively, this can be interpreted as follows. Increasing the number of processors in an interval decreases factorization time. However, since the parallel efficiency of triangular substitution typically degrades faster
than that associated with factorization, the decrease in the time for triangular substitutions is limited. By partitioning the spectrum into slightly more intervals, additional reduction in the time for triangular substitutions can be achieved, since each interval will have fewer linear equations to solve.

The observation above is also consistent with the cost model presented in section 2.3. Our model indicates that $k_{\text{opt}}$ will decrease as $c_p$ increases, resulting in more intervals, when all other parameters are held fixed and $\eta_f > \eta_s$. However, we notice that the increase of the number of intervals $l$ is slower than the increase of total number of cores. That means more cores are assigned to each interval also, as $p$ increases. This strategy has the combined effect of reducing both the factorization and total triangular substitution times, as $p$ increases.

4.1.3. Weak scaling

For the same reasons discussed in the previous section, the weak-scaling of MSIL is not easy to measure either. We are interested in learning how the wall clock time changes when both the problem size ($n$) and the number of cores ($p$) used in the computation are increased at the same rate. The analysis we presented in section 2.3 indicates that the change in wall clock time depends on how the spectrum is partitioned as $n$ and $p$ increase at the same rate. As we move from Graphene512 ($n = 20, 480$) to Graphene2048 ($n = 81, 920$) in Table 4, the number of intervals associated with the best wall clock time increases from $l = 25$ to $l = 102$. As a result, the number of eigenvalues within each interval and the number of processors used for each interval stays the same. In this case, our model predicts that the wall clock time used for factorization will increase by a factor of $4^{\alpha_f}$, and the triangular substitution time will increase by a factor of $4^{\alpha_s}$, if $c_f$ and $c_s$ do not change as $n$ changes. These changes are independent of $\eta_f$ or $\eta_s$. Since graphene is a 2D problem, $\alpha_f = 3/2$ and $\alpha_s = 1$. Using the factorization and substitution timing measurements recorded for $n = 20, 480$ and $p = 512$, which are $t_f = 4.8$ and $t_s = 11.6$ respectively, we can estimate the expected increase in wall clock time is roughly $(4^{3/2} \times 4.8 + 4 \times 11.6)/(4.8 + 11.6) \approx 3.2$. Table 4 shows that measured increase in wallclock time from $n = 20, 480$, $p = 512$ to $n = 81, 920$, $p = 2048$ is about $50.6/16.4 \approx 3.1$. Therefore the weak scaling from from Graphene512 to Graphene2048 is nearly perfect.
Table 4: Weak scaling of MSIL for the 2D problems.

<table>
<thead>
<tr>
<th>problem</th>
<th>p</th>
<th>q × l</th>
<th>k</th>
<th>$t_f$</th>
<th>mean $t_s$</th>
<th>max $t_s$</th>
<th>$t_{wall}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphene512</td>
<td>512</td>
<td>20 × 25</td>
<td>82</td>
<td>4.8</td>
<td>9.2</td>
<td>11.6</td>
<td>16.4</td>
</tr>
<tr>
<td>Graphene2048</td>
<td>2048</td>
<td>20 × 102</td>
<td>80</td>
<td>14.8</td>
<td>25.6</td>
<td>35.8</td>
<td>50.6</td>
</tr>
<tr>
<td>Graphene8192</td>
<td>8192</td>
<td>24 × 341</td>
<td>96</td>
<td>108</td>
<td>190</td>
<td>245</td>
<td>353</td>
</tr>
</tbody>
</table>

A similar estimate of the increase in wall clock time can be made when $n$ and $p$ increase from 81, 920 and 2, 048 to 327, 680 and 8, 192 respectively. In this case, $k$ increases from 80 to approximately 96 (a factor of as $1.2$) $l$ increases from 102 to 341. These changes yield an increase in factorization time by a factor of $4^{3/2}/1.2^2$, and in the time for triangular substitutions by a factor of $4/1.2^{n_e-1}$. Using the simplified assumption that $\eta_f = \eta_s = 1/2$ and the measured $t_f$ and $t_s$, we predict the total wall clock time to increase by a factor of $(4^{3/2} \times 14.8/\sqrt{1.2} + 4 \times \sqrt{1.2} \times 35.8)/(14.8 + 35.8) \approx 5.2$. Our measured increase is 353/50.6 $\approx 7$, which is slightly worse than our prediction. We believe the deviation is partly due to simplified assumptions we made about $\eta_f$, $\eta_s$, and partly due to the increased cost of orthogonalization and Rayleigh-Ritz calculation, which contributes a gradual degradation in weak scaling as the total number of cores increase.

4.2. MCISPM

In this section, we examine the performance of MCISPM by reporting the wall clock time used by the FEAST program to compute roughly 10% of the lowest eigenvalues of each matrix. We first show that the best way to divide the spectrum to achieve maximum efficiency for a fixed amount of computational resource is a nontrivial problem. We then study how the computation scales with an increasing amount of computational resource in both the strong and weak sense.

We use 16 poles in the numerical quadrature for approximating the contour integral in (6) in all our experiments. If the total number of cores used to compute eigenvalues within a single contour is $q$, then $q/16$ cores are assigned to each pole to perform the factorization of $A - z_i I$ and solve triangular systems required in the subspace iteration. In all experiments below, we apply two subspace iterations to the approximate projection operator constructed from the approximate contour integral of the resolvent. They yield eigenpairs whose relative residual norms are all below the tolerance of $10^{-10}$.

4.2.1. Spectrum division and processor groups

In Table 5, we report the total wall clock time $t_{wall}$ for different $q \times l = p$ combinations. Although it is possible to compute all desired eigenvalues by enclosing them in a single contour, Table 5 shows that this may not be the optimal way of using the contour integral projection method for a fixed amount of computational resource. In this experiment, we use $p = 512$ cores to compute 2,048 (10%) lowest eigenvalues of the Graphene512 matrix. Table 5 shows that the most efficient way to obtain all desired eigenvalues is to break the interval into $l = 8$ subintervals, with each subinterval containing $k = 256$ eigenvalues. We also divide 512 cores into 8 groups with $q = 64$ cores in each group. Each group of 64 cores is responsible for computing 256 eigenvalues within one interval.

Table 5: The performance profile of FEAST for different combinations of $q \times l = 512$ when it is used to compute 2,048 eigenvalues of Graphene512.

<table>
<thead>
<tr>
<th>$q \times l$</th>
<th>$k$</th>
<th>$t_{wall}$</th>
<th>$t_f$</th>
<th>$t_s$</th>
<th>$t_{other}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 × 1</td>
<td>2,048</td>
<td>384</td>
<td>11</td>
<td>248</td>
<td>125</td>
</tr>
<tr>
<td>256 × 2</td>
<td>1,024</td>
<td>224</td>
<td>15</td>
<td>152</td>
<td>57</td>
</tr>
<tr>
<td>128 × 4</td>
<td>512</td>
<td>148</td>
<td>27</td>
<td>94</td>
<td>27</td>
</tr>
<tr>
<td>64 × 8</td>
<td>256</td>
<td>125</td>
<td>47</td>
<td>68</td>
<td>10</td>
</tr>
</tbody>
</table>
In Table 5, we also report the time used for factorization ($t_f$) and triangular substitutions ($t_s$) respectively. The column labeled by $t_{other}$ is simply the difference between total wall clock time and the sum of factorization and total triangular substitution times. This portion of the wall clock time mainly accounts for the cost of performing the Rayleigh-Ritz procedure in the subspace iteration. It includes the time for performing sparse matrix vector multiplications, dense matrix matrix multiplications and computing eigenvalues of the projected matrix. When the number of eigenvalues within each interval is small, e.g. $k = 64$, this portion of the time is relatively small. When the number of eigenvalues within each interval is large, e.g., $k = 2048$, $t_{other}$ can be quite significant (33%). However, even if we exclude this part of the timing measurement, the total wall clock time is still the least for the $q = 64$ and $l = 8$ combination.

The reason that choosing $l = 1$ and $q = 512$ leads to the most time consuming computation can be explained by noticing the significant amount of time spent in triangular substitutions. As we pointed out in section 2.3, the optimal $k$ often makes the wall clock spent in factorization and triangular substitutions roughly equal.

For this test problem, as we increase $l$ and reduce $k$ and $q$ for each interval accordingly, the cost of triangular substitutions decreases by a much larger margin than the increase in the factorization cost. This results in a reduction in total wall clock time. It is interesting to observe that when $p = 64$ and $l = 8$, which yields the best performance, the amount of time spent in factorization and triangular substitutions is comparable.

In Table 6, we report the best combinations of $q$ and $l$ for a given total core count for all test problems. The column labeled by min_q gives the minimum number of cores required to solve the problem due to memory constraints, and the column labeled by $t_{wall}$ gives the total wall clock time used to compute 10% lowest eigenvalues of these test problems. Note that for C60, the best $q$ is larger than the min_q. When 16 cores are used to solve the problem, each core is assigned to a different pole, as a result, the factorization time dominates the calculation even though each interval has fewer eigenvalues, hence requires fewer triangular substitutions. The decrease in the time for triangular substitutions is not enough to offset the increase in factorization time.

One observation that can be made based on Table 6 is that the minimum $q$ for 3D problems is larger than that associated with 2D problems with a similar matrix size. In both cases, the optimal combination is obtained when the time spent in factorization and triangular substitutions are about the same.

The last column in Tab. 6 gives the best total wall-clock times achieved using MSIL to solve the same problem. A comparison between the best wall-clock times for the two methods we studied reveals that MSIL is much faster than MCISP.

### 4.2.2. Strong scaling

We observe from above that there is a tradeoff between using more cores to factor $A - z I$ (which results in having fewer intervals with more eigenvalues to compute in each interval) and splitting the spectrum into more intervals, each with fewer eigenvalues to compute, when the total number of cores is fixed. The optimal choice of $q$ and $l$ combination will depend on the total number of cores ($p$) available. Therefore, the strong scaling of MCISP depends on how $q$ and $l$ are chosen.
Table 7 shows that if we simply increase $l$ and reduce the number of eigenvalues $k$ within each interval as the total number of cores increases, the wall clock time may not decrease at the same rate because the factorization time may start to dominate the computation. Likewise, simply increasing $q$ without changing $l$ may not be the right solution either because the performance of a triangular substitution benefits less from adding more cores. Hence, the cost of triangular substitutions may begin to dominate the computation as $q$ increases.

Table 7: The scaling of wall clock time for computing $n_{\lambda} = 8,192$ lowest eigenvalues of Graphene2048 with respect to the total number of cores $p$ used in the computation. We divide the desired part of the spectrum into $l$ intervals, and use $q = 256$ cores to compute $k = 8,192/l$ eigenvalues within each interval.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q \times l$</th>
<th>$k$</th>
<th>$t_f$</th>
<th>$t_s$</th>
<th>$t_{\text{other}}$</th>
<th>$t_{\text{wall}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,192</td>
<td>$256 \times 32$</td>
<td>256</td>
<td>65</td>
<td>104</td>
<td>37</td>
<td>208</td>
</tr>
<tr>
<td>16,384</td>
<td>$256 \times 64$</td>
<td>128</td>
<td>65</td>
<td>52</td>
<td>18</td>
<td>137</td>
</tr>
<tr>
<td>32,768</td>
<td>$256 \times 128$</td>
<td>64</td>
<td>65</td>
<td>26</td>
<td>18</td>
<td>104</td>
</tr>
<tr>
<td>65,536</td>
<td>$256 \times 256$</td>
<td>32</td>
<td>65</td>
<td>12</td>
<td>27</td>
<td>89</td>
</tr>
<tr>
<td>131,072</td>
<td>$256 \times 512$</td>
<td>16</td>
<td>65</td>
<td>6</td>
<td>12</td>
<td>87</td>
</tr>
<tr>
<td>262,144</td>
<td>$256 \times 1024$</td>
<td>8</td>
<td>65</td>
<td>3.2</td>
<td>10</td>
<td>83</td>
</tr>
</tbody>
</table>

It follows from the model established in section 2.3 that a reasonable strategy for selecting $q$ and $k$ is to increase $q$ without decreasing $k$ when the cost of factorization is relatively high. As $t_f$ is reduced significantly or when factorization cannot benefit much from an increase in $q$ per interval, it then becomes more effective to increase the number of intervals $l$ and reduce $k$ per interval accordingly.

Table 8 shows that by increasing $q$ from 256 to 512 as the total number of cores $p$ is doubled from 8,192 to 16,384, we can reduce $t_f$ by a half and $t_s$ by 40%. This leads to a significant reduction of the total wall clock time for computing 10% of the lowest eigenvalues of Graphene2048. At that point, the time spent in factorization becomes significantly lower than that spent in triangular substitutions. Thus, when the total number of cores is doubled further to $p = 131,072$, it is more effective to increase the number of intervals and reduce the number of eigenvalues within each interval. As a result, the time for triangular substitutions ($t_s = 9$ seconds) is significantly less than the factorization time ($t_f = 33$ seconds). If $p$ is to be doubled again, we should then increase $p$ per interval to reduce the factorization time and, consequently, the total wall clock time significantly.

Table 8: The scaling of wall clock time for computing 8,192 lowest eigenvalues of Graphene2048 with an optimal choice of $p$ and $l$. The solid curves include only the wall clock time used in MSIL. The dashed curves also include postprocessing time for validating the eigenvalue count and computing a few missing eigenvalues.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q \times l$</th>
<th>$k$</th>
<th>$t_f$</th>
<th>$t_s$</th>
<th>$t_{\text{other}}$</th>
<th>$t_{\text{wall}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,192</td>
<td>$256 \times 32$</td>
<td>256</td>
<td>65</td>
<td>104</td>
<td>37</td>
<td>208</td>
</tr>
<tr>
<td>16,384</td>
<td>$512 \times 32$</td>
<td>256</td>
<td>33</td>
<td>68</td>
<td>30</td>
<td>131</td>
</tr>
<tr>
<td>32,768</td>
<td>$512 \times 64$</td>
<td>128</td>
<td>33</td>
<td>36</td>
<td>16</td>
<td>85</td>
</tr>
<tr>
<td>65,536</td>
<td>$512 \times 128$</td>
<td>32</td>
<td>33</td>
<td>9</td>
<td>5</td>
<td>47</td>
</tr>
<tr>
<td>131,072</td>
<td>$512 \times 256$</td>
<td>16</td>
<td>24</td>
<td>9</td>
<td>4</td>
<td>38</td>
</tr>
</tbody>
</table>

4.2.3. Weak scaling

We observe from (3) that the optimal number of eigenvalues within each interval depends on the size of the problem as well as the relative cost and efficiency of the factorization and triangular substitution when $c_n/c_p$ is held constant. When $\eta_f$, $\eta_s$, $c_f$ and $c_s$ do not vary much as $n$ and $p$ are increased, the optimal $k$ should increase with respect to $n$, which is consistent with what we observe in our experiment.
In Table 9, we report the best wall clock time we observed when Graphene problems of increasing sizes are solved by an increasing number of cores. In particular, when the matrix size is increased by a factor of 4, we increase the total number of cores by a factor of 4 also. The best wall clock time measurements are obtained by increasing both \( k \) and \( q \) by a factor of two when both \( n \) and \( p \) are quadrupled.

Table 9: Weak Scaling on Graphene problems

<table>
<thead>
<tr>
<th>problem</th>
<th>( n )</th>
<th>( p )</th>
<th>( t_{\text{wall}} )</th>
<th>( t_{f} )</th>
<th>( t_{s} )</th>
<th>( k )</th>
<th>( q )</th>
<th>( l )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphene128</td>
<td>5,120</td>
<td>512</td>
<td>3.6</td>
<td>1.7</td>
<td>1.2</td>
<td>64</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>Graphene512</td>
<td>20,480</td>
<td>2048</td>
<td>55</td>
<td>27</td>
<td>24</td>
<td>128</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>Graphene2048</td>
<td>89,120</td>
<td>8192</td>
<td>208</td>
<td>65</td>
<td>104</td>
<td>256</td>
<td>256</td>
<td>32</td>
</tr>
</tbody>
</table>

The increase of wall clock time by a factor of 3.8 reported in Table 9 when the matrix size is increased from \( n = 20,480 \) to \( n = 81,920 \) is well within the range predicted by our simple model. Our prediction is based on the analysis presented in section 2.3, which estimates the increase to be by a factor of

\[
\frac{\frac{4^{3/2}}{2^3} t_{f} + \frac{4^2}{2^{n_f}} t_{s}}{t_{f} + t_{s}},
\]  

(7)

where \( t_{f} = 27 \) and \( t_{s} = 24 \) represent the cost of factorization and triangular substitutions before \( n \) and \( p \) are increased. If we use \( \eta_f = \eta_s = 1/2 \), (7) is approximately 5.6.

However, when the matrix size is increased from \( n = 5,120 \) to \( n = 20,480 \), the increase of wall clock time by a factor of 16 is large than we expected. We believe this kind of deviation from our predicted increase of a factor of 5.6 is due to the relative small size of the problem. When the problem size is too small, the prefactors of the estimated cost of factorization and triangular substitutions play a more prominent role, but they were neglected in our previous analysis. This can be see from the factorization time alone, which increases by a factor of more than 15, whereas the standard complexity analysis predicts an increase by a factor of 4 only. Such increase is directly related to the rapid increase in the number of nonzeros in the lower triangular factor, which exceeds a factor of 10 when the number of atoms in the graphene is increased from 128 to 512.

Our main interest in this paper is on larger problems that are solved with a large number of cores. In this regime, it appears that the MCISPM has a good weak scaling property.

5. Concluding remarks

We investigated two parallel methods for computing a relatively large number of eigenpairs of a large sparse matrix. Both of these two methods are based on the general technique of dividing the spectrum of interest into several intervals and computing eigenvalues within each interval simultaneously. One method uses the shift-invert Lanczos procedure with implicitly restarted to compute eigenvalues within each interval. The other method constructs an approximate spectral projection operator \( P \) through a contour integral formulation and applies a few subspace iterations to obtain the desired eigenvalues. Both methods require solving a number of linear systems of equations. We assume in this paper that these equations can be solved by sparse direct methods such as the one implemented in the MUMPS software package [1].

We also assume that a large number of processors or computational cores are available to solve the eigenvalue problem. A practical question we examined is how we should partition the spectrum in a way to make optimal use of a vast amount of computational resource. We established a simple model to study the trade-off between sparse factorization and triangular substitutions and how this trade-off may affect the wall-clock time of the overall computation. Spectrum partitioning strategies based on the analysis of this model appears to yield good performance in our computational experiments.
Our computational experiments indicate that MSIL tends to give better performance when a moderate number of processors are used in the calculation. On the other hand, MCISPM may be able to take advantage of a larger number of processors, if available.

We remark that the optimal spectrum partition strategy and the actual performance of the eigensolver depends on the performance and the scalability of the sparse direct method. Although we only experimented with the use of MUMPS, many of the observations and trends we discussed are applicable to other solvers.

References


