SIAM J. SCI. COMPUT.
Vol. 46, No. 2, pp. S324–S351

A PARALLEL ALGORITHM FOR COMPUTING PARTIAL SPECTRAL FACTORIZATIONS OF MATRIX PENCILS VIA CHEBYSHEV APPROXIMATION*

TIANSHI XU†, ANTHONY AUSTIN‡, VASILEIOS KALANTZIS§, AND YOUSEF SAAD†

Abstract. We propose a distributed-memory parallel algorithm for computing some of the algebraically smallest eigenvalues (and corresponding eigenvectors) of a large, sparse, real symmetric positive definite matrix pencil that lie within a target interval. The algorithm is based on Chebyshev interpolation of the eigenvalues of the Schur complement (over the interface variables) of a domain decomposition reordering of the pencil and accordingly exposes two dimensions of parallelism: one derived from the reordering and one from the independence of the interpolation nodes. The new method demonstrates excellent parallel scalability, comparing favorably with `PARPACK`, and does not require factorization of the mass matrix, which significantly reduces memory consumption, especially for 3D problems. Our implementation is publicly available on GitHub.

Key words. symmetric generalized eigenvalue problem, spectral Schur complements, Chebyshev approximation, parallel computing

MSC codes. 15A18, 65D15, 65F15, 65N55, 65Y05, 68W10

DOI. 10.1137/22M1501155

1. Introduction.

Several applications in science and engineering require the computation of a handful of the algebraically smallest eigenvalues and associated eigenvectors of a large, sparse matrix pencil (A, M), where the $n \times n$ matrices A and M are real symmetric and M is positive-definite. Often, one is provided bounds α and β on the eigenvalues of interest, and the goal is then to compute all n_{ev} eigenpairs of (A, M) that lie within $[\alpha, \beta]$. That is, one seeks nontrivial solutions to

$$Ax = \lambda Mx, \qquad \lambda \in [\alpha, \beta].$$

Problems of this sort arise, for instance, in spectral clustering [41] and low-frequency response analysis [6, 15].

Due to the size of modern matrix problems, parallel computing has become an integral part of software libraries targeting large-scale eigenvalue computations. In many packages (e.g., `PARPACK` [30, 34], `PRIMME` [37], `BLOPEX` [28]), linear algebra kernels are the main source of parallelism, with operations such as matrix-vector and dot products performed in parallel by distributing the data across multiple

*Received by the editors June 7, 2022; accepted for publication (in revised form) February 7, 2023; published electronically September 20, 2023.
https://doi.org/10.1137/22M1501155
Funding: The work of the first and fourth authors was supported by National Science Foundation (NSF) grant DMS-1912048. The work of the second author was supported by the Research Initiation Program at the Naval Postgraduate School. The work of the third author was supported by the Mathematical Sciences Council of IBM Research through its Exploratory Science initiative.
†Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA (xuxx1180@umn.edu, saad@umn.edu).
‡Department of Applied Mathematics, Naval Postgraduate School, Monterey, CA 93940 USA (anthony.austin@nps.edu).
§IBM Research, Thomas J Watson Research Center, Yorktown Heights, NY 10598 USA (vkal@ibm.com).

processors. Several recent packages improve scalability by exploiting additional levels of parallelism via techniques such as spectrum slicing (`pEVSL` [31]), rational filtering (`FEAST/PFEAST` [20, 27, 35] and `z-Pares` [36]), and parallel shift-and-invert methods [42, 46]. The `SLEPc` collection of distributed-memory eigenvalue algorithms [14] contains implementations of several of these methods.

Another class of distributed-memory eigenvalue solvers is based on algebraic domain decomposition, also known as algebraic substructuring. In domain decomposition, the adjacency graph associated with the pencil (A, M) is partitioned into several nonoverlapping subgraphs. The eigenvalue problem then decouples into two separate tasks: first, one determines the eigenvector components associated with the interface variables of the partitioned graph; then, one finds the components associated with the interior variables. The second task parallelizes naturally over the subgraphs. For more information, see [6, 12, 17, 29, 45] and the references therein.

1.1. A new parallel algorithm. In this article, we combine the domain decomposition approach with Chebyshev function approximation to design a new distributed-memory parallel eigensolver. The contributions of our work are the following:

1. The algorithm parameterizes the eigenvector components associated with the interior and interface variables as univariate, analytic, vector-valued functions. It then uses the fact that Chebyshev interpolation of these functions yields good approximations to the eigenvectors to construct a subspace for use with a Rayleigh–Ritz projection scheme. We present theoretical and practical details when the interpolation points are Chebyshev nodes of the second kind.

2. The proposed algorithm leverages multidimensional parallelism by assigning computations associated with different Chebyshev nodes to different processor groups and assigning computations associated with different subdomains to different processors within each group. Our numerical experiments demonstrate that the algorithm achieves higher parallel efficiency than `PARPACK` on distributed-memory systems communicating via the Message Passing Interface (MPI) [13]. A C++/MPI implementation of the proposed algorithm is available publicly at https://github.com/Hitenze/Schurcheb.

3. In contrast to previous work on domain decomposition eigensolvers, the proposed algorithm requires the computation of neither derivatives of eigenvectors [18] nor a large number of eigenvectors of linearized spectral Schur complements [5, 6]. Moreover, unlike branch-hopping domain decomposition algorithms, which compute eigenvalues one at a time [19, 21], the proposed algorithm introduces model parallelism in addition to data parallelism by approximating all sought eigenvalues simultaneously via Rayleigh–Ritz projection. Unlike approaches based on the Lanczos algorithm, the proposed algorithm does not require a distributed-memory factorization of A or M; therefore, it is not limited by the efficiency of distributed-memory triangular solves. Finally, in contrast to most rational filtering techniques, especially those based on discretizations of complex contour integrals [22, 23], the proposed algorithm does not evaluate functions at complex values and therefore does not require complex arithmetic.

1.2. Notation and roadmap. Throughout the paper, we denote the set of eigenvalues of a general pencil (K, F) by $\Lambda(K, F)$ and the eigenpairs of the specific pencil (A, M) by $\left(\lambda_i, x^{(i)}\right)$, $i = 1, \ldots, n$, ordered algebraically: $\lambda_1 \leq \cdots \leq \lambda_n$. Given bounds α and β such that $\alpha < \lambda_1$, our aim is to compute all n_{ev} eigenpairs of (A, M) that lie in $[\alpha, \beta]$, i.e., the n_{ev} algebraically smallest eigenvalues of A and their corresponding eigenvectors. Finally, we denote by $\mathrm{Ran}(K)$ and $\mathrm{Ker}(K)$ the range and kernel of a matrix K and by $\mathrm{span}\{v_1, \ldots, v_k\}$ the linear span of vectors $v_1, \ldots, v_k$.

This paper is organized as follows. Section 2 presents background on algebraic graph partitioning and domain decomposition. Section 3 shows how the eigenvectors of (A, M) can be identified as values of certain univariate, vector-valued functions and discusses how they can be approximated by Rayleigh–Ritz projection onto a subspace formed via Chebyshev approximation. Section 4 discusses the distributed-memory implementation of the proposed algorithm on 2D grids of MPI processes. Section 5 showcases the performance of the proposed algorithm using numerical experiments performed in both sequential and distributed-memory computing environments. Finally, section 6 presents our concluding remarks.

2. Domain decomposition variable ordering. Let $\mathcal{G} = (\mathcal{V}, \mathcal{I})$ be a simple undirected graph with vertex set $\mathcal{V}$ and edge set $\mathcal{I}$. A *p-way edge separator* is a subset $\mathcal{I}_s \subseteq \mathcal{I}$ whose removal from $\mathcal{I}$ divides the vertices of the graph $\mathcal{G}$ into $p \in \mathbb{N}$ nonoverlapping sets $\mathcal{V}_1, \ldots, \mathcal{V}_p$ such that the induced subgraphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{I}_1), \ldots, \mathcal{G}_p = (\mathcal{V}_p, \mathcal{I}_p)$ are disjoint. We refer to the induced subgraphs variously as *subdomains, substructures,* or *partitions.* A vertex is called an *interface vertex* if it is incident to an edge in $\mathcal{I}_s$ and an *interior vertex* otherwise.

Applied to graphs derived from matrices, edge separators are commonly used in parallel computing to achieve load balancing during the execution of distributed-memory linear algebra kernels. In this context, the induced subgraphs ideally have similar numbers of vertices and edges, while the size (cardinality) of the separator set is kept to a minimum. Finding the "best" edge separator is an NP-hard problem. In practice, one relies on heuristics, such as the algebraic partitioning strategies implemented in the popular `METIS` and `ParMETIS` packages [24, 25].

To a symmetric matrix pencil (A, M) of dimension n, we associate a graph $\mathcal{G}_{A,M}$ in the usual way, taking $\mathcal{V} = \{1, \ldots, n\}$ for the vertex set and $\mathcal{I} = \{(i, j) \mid A_{i,j} \neq 0 \text{ or } M_{i,j} \neq 0\}$ for the edge set. Thinking of the eigenvalue equation $Ax = \lambda Mx$ as a set of n linear equations in the components of x (one for each row of the system), the vertices correspond to the n unknown variables in the vector x, and the graph $\mathcal{G}_{A,M}$ has an edge connecting vertices i and j if the variable x_j appears in the ith equation. A p-way edge separator for $\mathcal{G}_{A,M}$ groups the variables into p disjoint sets or subdomains. Interface vertices correspond to variables that are coupled (via equations) with variables from multiple subdomains, while interior vertices correspond to variables that are coupled only with other variables from the same subdomain. Figure 2.1 illustrates this for a 4-way partitioning of a graph that models a 6×6 regular grid.

Having partitioned $\mathcal{G}_{A,M}$, we reorder the variables, listing all interior variables first, grouped in order by subdomain, followed by the interface variables, also grouped by subdomain. Let P be the permutation matrix that effects this reordering. Under P, the matrices A and M are reordered into a pair of structured block matrices:

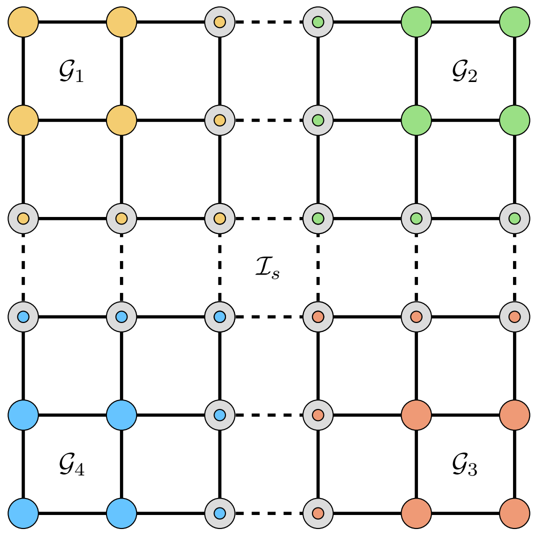

FIG. 2.1. *A* 4*-way partitioning of a* 6×6 *discretized domain obtained from an edge separator. The four colors distinguish the four different subdomains. Solid-colored nodes correspond to interior variables. Nodes with a gray background correspond to interface variables. Solid lines correspond to edges between vertices of the same partition. Dashed lines correspond to edges between vertices of neighboring partitions.*

$$
\begin{aligned}
P^T A P &= \begin{bmatrix}
B_1 & & & & E_1 & & & \\
& B_2 & & & & E_2 & & \\
& & \ddots & & & & \ddots & \\
& & & B_p & & & & E_p \\
E_1^T & & & & C_{1,1} & C_{1,2} & \cdots & C_{1,p} \\
& E_2^T & & & C_{2,1} & C_{2,2} & \cdots & C_{2,p} \\
& & \ddots & & \vdots & \vdots & \ddots & \vdots \\
& & & E_p^T & C_{p,1} & C_{p,2} & \cdots & C_{p,p}
\end{bmatrix} \\
P^T M P &= \begin{bmatrix}
M_{B_1} & & & & M_{E_1} & & & \\
& M_{B_2} & & & & M_{E_2} & & \\
& & \ddots & & & & \ddots & \\
& & & M_{B_p} & & & & M_{E_p} \\
M_{E_1}^T & & & & M_{C_{1,1}} & M_{C_{1,2}} & \cdots & M_{C_{1,p}} \\
& M_{E_2}^T & & & M_{C_{2,1}} & M_{C_{2,2}} & \cdots & M_{C_{2,p}} \\
& & \ddots & & \vdots & \vdots & \ddots & \vdots \\
& & & M_{E_p}^T & M_{C_{p,1}} & M_{C_{p,2}} & \cdots & M_{C_{p,p}}
\end{bmatrix}.
\end{aligned}
\tag{2.1}
$$

To provide more detail, let d_i and s_i denote, respectively, the numbers of interior and interface variables belonging to the ith domain. The matrices B_i and M_{B_i} are of size $d_i \times d_i$ and represent the coupling between the interior variables within the ith subdomain. The matrices E_i and M_{E_i} are of size $d_i \times s_i$ and represent the coupling between the interior and interface variables of the ith subdomain. Finally, the matrices $C_{i,j}$ and $M_{C_{i,j}}$ are of size $s_i \times s_j$ and represent the coupling between the interface variables of the ith subdomain and those of the jth subdomain. If the ith and jth subdomains do not neighbor one another, $C_{i,j} = M_{C_{i,j}} = 0$. Since A and M are symmetric, $C_{j,i} = C_{i,j}^T$ and $M_{C_{j,i}} = M_{C_{i,j}}^T$.

Our algorithm makes essential use of the structure of this reordering of A and M. For the remainder of the paper, we assume that A and M have been so reordered and suppress mention of the permutation P. We write A and M in 2×2 block form as

$$A = \begin{bmatrix} B & E \\ E^T & C \end{bmatrix}, \qquad M = \begin{bmatrix} M_B & M_E \\ M_E^T & M_C \end{bmatrix}, \tag{2.2}$$

with the blocks being defined in the obvious way to conform to the structure just described. Finally, we define $d = d_1 + \cdots + d_p$ and $s = s_1 + \cdots + s_p$, the total numbers of interior and interface variables, respectively. Thus, the matrices B and M_B are $d \times d$, E and M_E are $d \times s$, and C and M_C are $s \times s$. Of course, $d + s = n$.

3. A parallel algorithm based on Chebyshev approximation. Our algorithm is based on the fact that the eigenvalues and eigenvectors of the matrix $A - \zeta M$ are analytic functions of $\zeta \in \mathbb{C}$ (vector-valued in the case of the latter). By definition, if $\zeta = \lambda_i$ is an eigenvalue of the pencil (A, M), then $A - \zeta M$ is singular, and its null vectors are the eigenvectors for (A, M) corresponding to λ_i. By continuity, if ζ is close (but not equal) to λ_i, then $A - \zeta M$ will be "nearly singular" in the sense that it will have one or more eigenvalues that are small in magnitude, and the eigenvectors of $A - \zeta M$ corresponding to these eigenvalues will be good approximations to null vectors of $A - \lambda_i M$. On this basis, our algorithm approximates the eigenvectors corresponding to the smallest eigenvalues of $A - \zeta_i M$ at several points ζ_i within the search interval $[\alpha, \beta]$ using a Schur complement technique. By choosing the ζ_i well, we can guarantee that the subspace spanned by these "near-null" vectors contains good approximations to the eigenvectors of (A, M). The algorithm extracts such approximations from this subspace via Rayleigh–Ritz projection.

3.1. Spectral Schur complements. To make this process efficient and parallelizable, we exploit the block structure of A and M induced by the variable reordering discussed in the previous section. Partition the eigenvector $x^{(i)}$ associated with the eigenvalue λ_i of (A, M) as

$$x^{(i)} = \begin{bmatrix} u^{(i)} \\ y^{(i)} \end{bmatrix},$$

where $u^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}^s$, conforming to the partitioning of A and M in (2.2), and define

$$B(\zeta) = B - \zeta M_B, \qquad E(\zeta) = E - \zeta M_E, \qquad C(\zeta) = C - \zeta M_C \tag{3.1}$$

for $\zeta \in \mathbb{C}$. In this notation, the eigenvector equation $(A - \lambda_i M)x^{(i)} = 0$ becomes

$$\begin{bmatrix} B(\lambda_i) & E(\lambda_i) \\ E^T(\lambda_i) & C(\lambda_i) \end{bmatrix} \begin{bmatrix} u^{(i)} \\ y^{(i)} \end{bmatrix} = 0. \tag{3.2}$$

Under the mild assumption that $B(\lambda_i)$ is invertible, i.e., that $\lambda_i \notin \Lambda(B, M_B)$, we can eliminate the $E^T(\lambda_i)$ block in the second row, yielding

$$\left[C(\lambda_i) - E^T(\lambda_i) B(\lambda_i)^{-1} E(\lambda_i)\right] y^{(i)} = 0. \tag{3.3}$$

That is, the $s \times 1$ bottom part $y^{(i)}$ of the eigenvector $x^{(i)}$ is a null vector of the Schur complement $C(\lambda_i) - E^T(\lambda_i) B(\lambda_i)^{-1} E(\lambda_i)$. Having found $y^{(i)}$, one can recover the corresponding top part $u^{(i)}$ via

$$u^{(i)} = -B(\lambda_i)^{-1}E(\lambda_i)y^{(i)}, \tag{3.4}$$

which requires the solution of a $d \times d$ block diagonal linear system.

What if $\lambda_i \in \Lambda(B, M_B)$? This case would seldom occur in practice, but we can come to understand it by writing $u^{(i)} = u_P^{(i)} + u_N^{(i)}$, where $u_P^{(i)} \in \operatorname{Ran}\big(B(\lambda_i)\big)$ and $u_N^{(i)} \in \operatorname{Ker}\big(B(\lambda_i)\big)$. In place of (3.4), the first block equation in (3.2) yields

$$u_P^{(i)} = -B(\lambda_i)^{+}E(\lambda_i)y^{(i)}, \tag{3.5}$$

where $B^{+}(\lambda_i)$ is the (Moore–Penrose) pseudoinverse of $B(\lambda_i)$. From this and the second block equation in (3.2), we obtain

$$E(\lambda_i)^T u_N^{(i)} + \big[C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^{+}E(\lambda_i)\big]y^{(i)} = 0 \tag{3.6}$$

instead of (3.3).

If it happens that $\operatorname{Ran}\big(E(\lambda_i)\big) \perp \operatorname{Ker}\big(B(\lambda_i)\big)$, so that the first term in (3.6) vanishes, then the eigenvectors can be found in a manner analogous to the case when $\lambda_i \notin \Lambda(B, M_B)$ but with $B(\lambda_i)^{-1}$ replaced by $B(\lambda_i)^{+}$. Specifically, one can take $y^{(i)}$ from among the null vectors of the Schur-complement-like matrix $C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^{+}E(\lambda_i)$ and then recover $u_P^{(i)}$ from (3.5). The component $u_N^{(i)}$ can be taken arbitrarily from $\operatorname{Ker}\big(B(\lambda_i)\big)$ (i.e., from among the eigenvectors of (B, M_B) corresponding to the eigenvalue λ_i). We thus obtain an eigenspace of dimension $\dim \operatorname{Ker}\big(C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^{+}E(\lambda_i)\big) + \dim \operatorname{Ker}\big(B(\lambda_i)\big)$. More generally, given $u_N^{(i)}$, one can solve (3.6) for $y^{(i)}$ and then leverage (3.5) to find $u_P^{(i)}$. Unfortunately, an easy way to compute $u_N^{(i)}$ does not appear to exist, and even if one did, forming and factoring $C(\lambda_i) - E^T(\lambda_i)B(\lambda_i)^{+}E(\lambda_i)$ would still be prohibitively expensive.

It is better simply to avoid the case $\lambda_i \in \Lambda(B, M_B)$ to begin with. This can be done by adjusting the partitioning until no eigenvalues of (B, M_B) lie within the search interval $[\alpha, \beta]$. As the likelihood of this being necessary is already small—in particular, we did not need to do this in any of the numerical experiments reported below—we will not attempt to develop a comprehensive strategy here, leaving this as a potential matter for future work.

3.2. Chebyshev approximation of eigenvector components. We have thus reduced the problem to that of finding those values ζ in $[\alpha, \beta]$ for which the *parameterized spectral Schur complement* [5, 19],

$$S(\zeta) = C(\zeta) - E^T(\zeta)B(\zeta)^{-1}E(\zeta), \tag{3.7}$$

is singular, assuming that no eigenvalue of (A, M) within $[\alpha, \beta]$ is also an eigenvalue of (B, M_B). For $\zeta \notin \Lambda(B, M_B)$, let $\mu_1(\zeta), \ldots, \mu_s(\zeta)$ and $y_1(\zeta), \ldots, y_s(\zeta)$ denote the eigenvalues and corresponding eigenvectors of $S(\zeta)$, respectively:

$$S(\zeta)y_i(\zeta) = \mu_i(\zeta)y_i(\zeta), \qquad i = 1, \ldots, s.$$

The μ_i and y_i can be defined such that they are analytic functions of $\zeta \in \mathbb{C}$ away from $\Lambda(B, M_B)$. At each point of $\Lambda(B, M_B)$, they have at most a pole singularity [21, 26, 33, 39]. We refer to the μ_i as the *eigencurves* of S. We also define

$$u_i(\zeta) = -B(\zeta)^{-1}E(\zeta)y_i(\zeta), \qquad i = 1, \ldots, s,$$

which is also analytic in ζ away from $\Lambda(B, M_B)$.

The matrix $S(\zeta)$ is singular precisely when one of its eigenvalues is zero: $\mu_i(\zeta)=0$ for some i. The following result asserts that each of the $n_{\rm ev} \le s$ eigenvalues of (A,M) in $[\alpha,\beta]$, counted according to multiplicity, occurs as a zero of one and only one μ_i.[1] Moreover, the top and bottom parts of the corresponding eigenvectors are given by the values of u_i and y_i at that zero. The assumption that $\beta < \min\bigl(\Lambda(B,M_B)\bigr)$ ensures that $[\alpha,\beta]$ is free of any poles of S and that the eigencurves are strictly decreasing [21]. The assumption that $n_{\rm ev} \le s$ ensures that the dimension of the space in which we plan to search is large enough to contain all the eigenvectors we seek.

PROPOSITION 3.1. *Assume that $\beta < \min\bigl(\Lambda(B,M_B)\bigr)$ and that $n_{\rm ev} \le s$. Then, there exist $n_{\rm ev}$ distinct integers $\kappa_1,\ldots,\kappa_{n_{\rm ev}} \in \{1,2,\ldots,s\}$ such that*

$$\mu_{\kappa_i}(\lambda_i)=0, \qquad y^{(i)}=y_{\kappa_i}(\lambda_i), \qquad u^{(i)}=u_{\kappa_i}(\lambda_i). \tag{3.8}$$

Proof. First, consider the case in which the λ_i are all simple eigenvalues. Following (3.3), we have $S(\lambda_i)y^{(i)}=0$ for some $y^{(i)} \ne 0$. The matrix $S(\lambda_i)$ is singular and has exactly one zero eigenvalue, denoted by $\mu_{\kappa_i}(\lambda_i)$, for some $1 \le \kappa_i \le s$. The expressions in (3.8) follow directly. It remains to show that $\kappa_i \ne \kappa_j$ when $i \ne j$.

By (3.7), the function S—and, by extension, each eigencurve μ_{κ_i}—has a singularity (a pole) at each eigenvalue of (B,M_B) and nowhere else. Since $\beta<\min\bigl(\Lambda(B,M_B)\bigr)$, it follows that the μ_{κ_i} are free of singularities on $[\alpha,\beta]$. Differentiating the Rayleigh quotient $\mu_{\kappa_i}(\zeta)=y_{\kappa_i}^T(\zeta)S(\zeta)y_{\kappa_i}(\zeta)/\|y_{\kappa_i}(\zeta)\|^2$, we find that $\mu'_{\kappa_i}(\zeta)<0$ on $[\alpha,\beta]$ [21, Proposition 3.1]. Hence, the μ_{κ_i} are strictly decreasing on $[\alpha,\beta]$, which implies that λ_i is the only root of μ_{κ_i} in $[\alpha,\beta]$.

That the result also holds in the case where one or more of the λ_i have nonunit multiplicity can be seen by considering arbitrarily small perturbations of (A,M) that have all simple eigenvalues and appealing to continuity. ∎

We lose no generality in assuming that $\kappa_i=i$, and we will do so throughout the rest of the paper: from this point forward, μ_i will denote the eigencurve of S that crosses the real axis at λ_i.

Proposition 3.1 tells us that the components $u^{(i)}$ and $y^{(i)}$ of a sought eigenvector $x^{(i)}$ are equal to $y_i(\lambda_i)$ and $u_i(\lambda_i)$, respectively. Since both $y_i(\zeta)$ and $u_i(\zeta)$ are analytic on $[\alpha,\beta]$, they can be approximated accurately by interpolation at Chebyshev nodes. Specifically, for an integer $N \ge 1$, let

$$\chi_j=\frac{\alpha+\beta}{2}+\cos\left(\frac{j\pi}{N-1}\right)\frac{\beta-\alpha}{2}, \quad j=0,\ldots,N-1, \tag{3.9}$$

be the N Chebyshev nodes of the second kind in $[\alpha,\beta]$,[2] and let ℓ_j denote the jth Lagrange basis function for polynomial interpolation in these nodes. That is, ℓ_j is the unique polynomial of degree $N-1$ such that $\ell_j(\chi_k)$ is 1 if $k=j$ and 0 if $k\ne j$. Finally, let $\mathcal{E}_\rho$ be the *Bernstein ellipse* centered on $[\alpha,\beta]$ with parameter ρ; that is, $\mathcal{E}_\rho$ is the open subset of $\mathbb{C}$ bounded by the ellipse with foci at α and β and the sum of the lengths of its semimajor and semiminor axes equal to ρ. Since $y_i(\zeta)$ and $u_i(\zeta)$ are analytic on $[\alpha,\beta]$, they can be analytically continued to $\mathcal{E}_\rho$ for some $\rho>0$. We have the following proposition.

[1] For eigenvalues of nonunit multiplicity, this statement is to be interpreted as saying that there is a distinct μ_i associated with each copy of the eigenvalue.

[2] For $N=1$, we take $\chi_0=(\alpha+\beta)/2$.

PROPOSITION 3.2. *Assume that $\beta < \min\big(\Lambda(B, M_B)\big)$, that $n_{\mathrm{ev}} \le s$, and that u_i and y_i are analytic in $\mathcal{E}_\rho$ for all $i = 1, \ldots, n_{\mathrm{ev}}$ and some $\rho > 0$. For each i, there exists $w^{(i)} \in \mathbb{R}^N$ such that*

$$x^{(i)} = \begin{bmatrix} u^{(i)} \\ y^{(i)} \end{bmatrix} = \begin{bmatrix} u_i(\chi_0) & \cdots & u_i(\chi_{N-1}) \\ y_i(\chi_0) & \cdots & y_i(\chi_{N-1}) \end{bmatrix} w^{(i)} + O(\rho^{-N}).$$

Proof. Let $w_j^{(i)} = \ell_j(\lambda_i)$ for $j = 0, \ldots, N-1$. Then, the top d (respectively, bottom s) components of the matrix-vector product give the value at λ_i of the polynomial interpolant to $u^{(i)}$ (respectively, $y^{(i)}$) in the Chebyshev nodes χ_j. The result now follows from a standard theorem on the convergence of Chebyshev interpolants to analytic functions [40, Theorem 8.2]. ∎

Instead of interpolating u_i and y_i directly, we use their samples at the Chebyshev nodes to generate a subspace in which to look for approximations to the $x^{(i)}$. This approach eliminates the need to keep track of the association between the samples and the eigencurves, which may be difficult if the eigencurves cross.[3] Proposition 3.2 ensures that this subspace contains good approximations to the $x^{(i)}$ for large enough N. We can express this fact as a statement about the angle between this subspace and the sought eigenspace.

COROLLARY 3.3. *Let $\mathcal{X} = \operatorname{span}\{x^{(1)}, \ldots, x^{(n_{\mathrm{ev}})}\}$, and let*

$$\mathcal{R} = \operatorname{span}\left\{ \begin{bmatrix} u_1(\chi_0) \\ y_1(\chi_0) \end{bmatrix}, \ldots, \begin{bmatrix} u_1(\chi_{N-1}) \\ y_1(\chi_{N-1}) \end{bmatrix}, \ldots, \begin{bmatrix} u_{n_{\mathrm{ev}}}(\chi_0) \\ y_{n_{\mathrm{ev}}}(\chi_0) \end{bmatrix}, \ldots, \begin{bmatrix} u_{n_{\mathrm{ev}}}(\chi_{N-1}) \\ y_{n_{\mathrm{ev}}}(\chi_{N-1}) \end{bmatrix} \right\}.$$

Then,

$$\sin\theta(\mathcal{X}, \mathcal{R}) = O(\rho^{-N}),$$

where $\theta(\mathcal{X}, \mathcal{R})$ is the largest principal angle between $\mathcal{X}$ and the closest subspace of $\mathcal{R}$ to $\mathcal{X}$ with the same dimension as $\mathcal{X}$.

Proof. The quantity $\sin\theta(\mathcal{X}, \mathcal{R})$ is known as the *gap* between $\mathcal{X}$ and $\mathcal{R}$ and can be expressed as [3], [26, sect. IV.2.1] [38, sect. II.4]:

$$\sin\theta(\mathcal{X}, \mathcal{R}) = \max_{x \in \mathcal{X}} \min_{r \in \mathcal{R}} \frac{\|x - r\|}{\|x\|}.$$

The result follows immediately from this formula and Proposition 3.2. ∎

3.3. A parallel algorithm. Our algorithm builds the subspace $\mathcal{R}$ of Corollary 3.3 and then uses Rayleigh–Ritz projection to extract approximations to the $x^{(i)}$ from $\mathcal{R}$. The procedure is summarized in Algorithm 3.1.

For each Chebyshev node χ_j, Algorithm 3.1 computes the eigenvectors associated with the n_{ev} algebraically smallest eigenvalues of $S(\chi_j)$. These eigenvectors form the $s \times n_{\mathrm{ev}}$ matrix Y_j (step 9). Then, the algorithm computes the matrix V_j, which requires the solution of a linear system with the coefficient matrix $B(\chi_j)$ and n_{ev} right-hand sides (step 10). Finally, the algorithm uses Rayleigh–Ritz projection (steps 15–16) to approximate the sought eigenpairs of (A, M). The dimension of the projected pencil is

[3]For example, it can happen that $\mu_2(\chi_j) < \mu_1(\chi_j) < \mu_3(\chi_j) < \cdots < \mu_s(\chi_j)$ for some j. If so, the eigenvector of $S(\chi_j)$ corresponding to its smallest eigenvalue is a sample of $y_2(\chi_j)$, not $y_1(\chi_j)$, even though μ_1 is the eigencurve for the smallest eigenvalue of (A, M).

Algorithm 3.1 The proposed algorithm.

1: **Input:** $A\in\mathbb{R}^{n\times n},\ M\in\mathbb{R}^{n\times n},\ N\in\mathbb{N},\ \alpha\in\mathbb{R},\ \beta\in\mathbb{R},\ n_{\mathrm{ev}}\in\mathbb{Z},\ Y=0,\ V=0$
2: **Output:** approximations of eigenpairs $\left(\lambda_i, x^{(i)}\right),\ i=1,\dots,n_{\mathrm{ev}}$

3: /* `Preprocessing: reorder matrices` A `and` M */
4: $\triangleright$ Call a p-way edge separator to partition the graph $\mathcal{G}_{A,M}$.
5: $\triangleright$ If $\beta<\min\left(\Lambda(B,M_B)\right)$ continue, else set $p:=2p$ and repeat step 4.

6: /* `Main loop; embarrassingly parallel over the` N `Chebyshev nodes` */
7: **for** $j=0,\dots,N-1$ **do**
8: $\triangleright$ Set $\chi_j=\dfrac{\alpha+\beta}{2}+\cos\left(\dfrac{j\pi}{N-1}\right)\dfrac{\beta-\alpha}{2}$.
9: $\triangleright$ Set $Y_j=[y_1(\chi_j),\dots,y_{n_{\mathrm{ev}}}(\chi_j)]$.
10: $\triangleright$ Solve $B(\chi_j)V_j=-E(\chi_j)Y_j$.
11: **end for**

12: /* `Rayleigh-Ritz projection phase` */
13: $\triangleright$ Set $R=\begin{bmatrix} V_0 & \cdots & V_{N-1}\\ Y_0 & \cdots & Y_{N-1}\end{bmatrix}$.
14: $\triangleright$ Optionally, orthonormalize the columns of R.
15: $\triangleright$ Compute the n_{ev} algebraically smallest eigenvalues and associated eigenvectors of the eigenvalue problem $(R^TAR)f=\theta(R^TMR)f$.
16: $\triangleright$ Return $(\theta_i, PRf^{(i)})\approx\left(\lambda_i,x^{(i)}\right),\ i=1,\dots,n_{\mathrm{ev}}$.

at most Nn_{ev}, and the associated eigenvalue problem is solved by a dense, symmetric eigenvalue solver.

The `for` loop in steps 7–11 is embarrassingly parallel: each matrix pair (Y_j, V_j) can be computed independently of the other pairs. The computation of V_j can be further decomposed into the solution of p independent linear systems. Partition V_j and Y_j by rows as

$$V_j=\begin{bmatrix} V_{1,j}\\ \vdots\\ V_{p,j}\end{bmatrix},\qquad Y_j=\begin{bmatrix} Y_{1,j}\\ \vdots\\ Y_{p,j}\end{bmatrix},$$

where $V_{k,j}$ and $Y_{k,j}$ are associated with the kth subdomain. Then,

$$\begin{bmatrix} B_1(\chi_j) & & \\ & \ddots & \\ & & B_p(\chi_j)\end{bmatrix}\begin{bmatrix} V_{1,j}\\ \vdots\\ V_{p,j}\end{bmatrix}=\begin{bmatrix} E_1(\chi_j)Y_{1,j}\\ \vdots\\ E_p(\chi_j)Y_{p,j}\end{bmatrix}$$

(where we have extended the notation (3.1) to the blocks comprising B, M_B, E, and M_E in the obvious way), and so the $V_{k,j}$ can be computed by solving

$$B_k(\chi_j)V_{k,j}=-E_k(\chi_j)Y_{k,j},\qquad k=1,\dots,p.$$

These p linear systems can be solved in parallel.

3.4. Practical details. A practical implementation of Algorithm 3.1 will need to account for certain details, some of which may include the following:

- If the desired number $n_{\rm ev}$ of eigenvalues is not known a priori, it can be computed directly by decomposing $A-\alpha M$ and $A-\beta M$ in LDL^T factorizations and using Sylvester's law of inertia [7]. Alternatively, if this is too expensive, one can estimate $n_{\rm ev}$ using a spectral density profile of (A,M) [44]. To reduce the chance of the algorithm missing eigenvalues, we recommend taking $n_{\rm ev}$ slightly larger than estimated or required. To further reduce this chance, one can apply a few steps of subspace iteration or Lanczos with polynomial filtering and deflation as postprocessing after step 16. Since the number of iterations needed should not be large, one can use iterative methods to approximate M^{-1} instead of exact factorizations.
- The results of section 3.2 relied on the hypothesis $\beta < \min\bigl(\Lambda(B,M_B)\bigr)$. How can we enforce this requirement in practice? This is difficult and may even be impossible for certain special classes of matrices. Nevertheless, we find empirically that, for a general problem, this is not likely to be an issue provided β is not excessively large. Should it happen that this condition is violated, we also find empirically that the situation frequently can be repaired simply by increasing p, i.e., by further partitioning the graph into a greater number of subdomains. Algorithm 3.1 therefore adopts the practical strategy of doubling p until $\beta < \min\bigl(\Lambda(B,M_B)\bigr)$ is satisfied (step 5). But we observe that this was not required in any of the many tests described in section 5.
- Algorithm 3.1 is a "one-shot" method in the sense that if the accuracy of the approximate eigenpairs is not satisfactory, then the whole process must be repeated with a higher value of N. We find that in practice, $N=8$ reaches nearly the maximum attainable accuracy on a wide range of problems; see section 5. If one wishes to apply Algorithm 3.1 for several values of N, it is beneficial to take these N to have the form $N(k)=2^k+1$ for integers k. Having run the algorithm with $N=N(k)$, one can reduce the computational cost of running the algorithm with $N=N(k+1)$ by exploiting the fact that the nodes (3.9) for $N(k)$ are a subset of those for $N(k+1)$ and reusing the samples taken during the $N=N(k)$ run.
- Besides increasing N, one can also improve the accuracy of one or more of the eigenpairs by using the approximate eigenvectors obtained from Algorithm 3.1 as the initial subspace for an implicitly restarted (or thick-restarted) Lanczos method [8, 43] applied to (A,M). This technique can also be used to ensure that all $n_{\rm ev}$ eigenpairs of (A,M) have been computed (i.e., none have been missed) by checking to see if the algebraically smallest eigenvalue returned by the restarted Lanczos method is smaller than β.

4. A distributed-memory implementation. We now describe our parallel implementation of Algorithm 3.1 based on the MPI standard. Throughout this discussion, we assume a distributed-memory computing environment with $N_p=p_rp_c$ MPI processes organized in a $p_r\times p_c$ 2D MPI grid. In addition to the default communicator `MPI_COMM_WORLD`, we denote by G_i^r, $i=0,\ldots,p_r-1$, and G_j^c, $j=0,\ldots,p_c-1$, the MPI communicators associated with the ith row and jth column of the grid, respectively.

Our parallel implementation utilizes the row dimension of the grid for domain decomposition data parallelism (i.e., distributed storage of A and M) and the column dimension of the grid for model parallelism (i.e., distribution over the N

Chebyshev nodes). Therefore, the row and column dimensions of the grid satisfy the inequalities $p_r \le p$ and $p_c \le N$, respectively.

4.1. Data distribution on 2D MPI grids. First, we consider the data distribution along the row dimension of the grid. For each communicator G_j^c, $j = 0, \ldots, p_c - 1$, we distribute A and M such that the p_r MPI processes associated with G_j^c hold a unique subset of the partitions of the graph $\mathcal{G}_{A,M}$. In particular, let p be a scalar multiple of p_r, and set $\tau = p/p_r$. Then, the ith process is assigned data associated with partitions $i\tau + 1, i\tau + 2, \ldots, (i+1)\tau$, i.e.,

$$\texttt{Data held by process } i \texttt{ of } G_j^c : \begin{cases} B_{i\tau+1}, \ldots, B_{(i+1)\tau}, M_{B_{i\tau+1}}, \ldots, M_{B_{(i+1)\tau}} \\ E_{i\tau+1}, \ldots, E_{(i+1)\tau}, M_{E_{it+1}}, \ldots, M_{E_{(i+1)\tau}} \\ C_{i\tau+1,:}, \ldots, C_{(i+1)\tau,:}, M_{C_{i\tau+1,:}}, \ldots, M_{C_{(i+1)\tau,:}} \end{cases},$$

where the subscript ":" represents all column indices of matrices C and M_C. Ordering the unknowns/equations by increasing MPI rank leads to the following global representation of A (and similarly for M):

$$A = \begin{bmatrix} B_1 & E_1 & & & & & \\ E_1^T & C_{1,1} & & C_{1,2} & & & C_{1,p_r} \\ & & B_2 & E_2 & & & \\ & C_{2,1} & E_2^T & C_{2,2} & & & C_{2,p_r} \\ & & & & \ddots & & \\ & & & & & B_{p_r} & E_{p_r} \\ & C_{p_r,1} & & C_{p_r,2} & & E_{p_r}^T & C_{p_r,p_r} \end{bmatrix}. \tag{4.1}$$

The ordering in (4.1) is more natural from the perspective of parallel computing than that in (2.1), which is more natural for discussing the linear algebra.

We now focus on the column dimension of the grid. Let N be a scalar multiple of p_c, and set $\eta = N/p_c$. We distribute the N Chebyshev nodes across the p_c MPI processes of each row communicator G_i^r, $i = 0, \ldots, p_r - 1$, such that each process receives exactly η unique Chebyshev nodes. In particular, the jth process associated is assigned the Chebyshev node(s) $\chi_{j\eta+1}, \ldots, \chi_{(j+1)\eta}$ $j = 0, \ldots, p_c - 1$. From a parallel efficiency perspective, it is advisable to exhaust parallelism across the N Chebyshev nodes first, by setting $p_c = N$, since this level of parallelism involves no communication among groups of processes assigned different Chebyshev nodes.

An illustration of the data distribution on a 2D MPI grid with $N_p = 16$ processes and $N = 8$ Chebyshev nodes is shown in Figures 4.1 and 4.2, where the dimensions of the grid are $(p_r, p_c) = (4, 4)$ and $(p_r, p_c) = (2, 8)$, respectively. For the $(4, 4)$ case, we have $p_c < N$, and each column subgrid is responsible for processing $h = 8/4 = 2$ Chebyshev nodes, while the computation of each matrix pair (Y_j, V_j) exploits four MPI processes. Contrast this with the $(2, 8)$ case, in which each separate column subgrid handles exactly one Chebyshev node ($\eta = 1$), leading to trivial parallelism with respect to the N Chebyshev nodes, but the computation of each matrix pair (Y_j, V_j) utilizes just two processes.

4.2. Computation of Y_j via `PARPACK`. Our implementation computes the eigenvectors of the Schur complement matrices $S(\chi_j)$, $j = 0, \ldots, N - 1$, via the `PARPACK`[4] software library, a distributed-memory implementation of `ARPACK` [30]. The main

[4]https://github.com/opencollab/arpack-ng

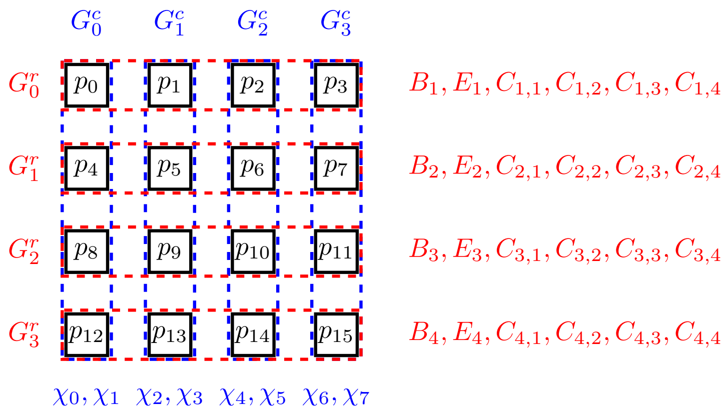

FIG. 4.1. *Distribution of blocks of A and Chebyshev nodes over a 2D MPI grid with $N_p = 16$, $N = 8$, and $(p_r, p_c) = (4, 4)$. The distribution of M is identical to that of A.*

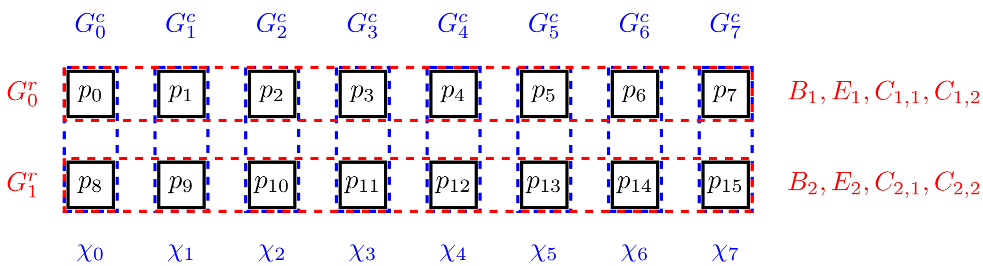

FIG. 4.2. *Distribution of blocks of A and Chebyshev nodes over a 2D MPI grid with $N_p = 16$, $N = 8$, and $(p_r, p_c) = (2, 8)$. The distribution of M is identical to that of A.*

distributed-memory kernels of `PARPACK` are (a) orthogonalization of the Krylov basis and (b) a user-defined routine that performs distributed matrix-vector multiplication with $S(\chi_j)$.

Regarding (a), consider first the case $p_c = N$. Orthonormalizing the basis vectors computed on each m-step cycle of the implicitly restarted Arnoldi method via Gram–Schmidt costs $O(sm^2)$ floating-point operations and $O\big(\log(p_r)m^2\big)$ point-to-point MPI messages. This communication cost increases proportionally with the number of Chebyshev nodes processed by each column subgrid. In particular, when $p_c = 1$, i.e., all available N_p MPI processes are assigned to the default communicator, `PARPACK` requires $O\big(N \log(N_p)m^2\big)$ MPI messages just for Gram–Schmidt.

As for (b), note that the product between the distributed matrix $S(\chi_j)$ and a distributed vector $f = \begin{bmatrix} f_1^T & \cdots & f_p^T \end{bmatrix}^T \in \mathbb{R}^s$ can be written as

$$S(\chi_j)f = \begin{bmatrix} \sum_{k \in \mathcal{N}_1} C_{1,k}(\chi_j) f_k \\ \vdots \\ \sum_{k \in \mathcal{N}_p} C_{p,k}(\chi_j) f_k \end{bmatrix} - \begin{bmatrix} E_1(\chi_j)^T B_1(\chi_j)^{-1} E_1(\chi_j) f_1 \\ \vdots \\ E_p(\chi_j)^T B_p(\chi_j)^{-1} E_p(\chi_j) f_p \end{bmatrix}, \tag{4.2}$$

where $\mathcal{N}_i$ denotes the list of partitions adjacent to partition i (and where we have extended the notation (3.1) to the blocks of $A - \zeta M$ defined by (4.1) in the obvious way). Due to the partitioning, the second term on the right-hand side of (4.2) can be computed in an embarrassingly parallel manner. On the other hand, the first term of the right-hand side of (4.2) requires point-to-point communication between processes handling neighboring partitions.

4.3. Orthonormalization of the Rayleigh–Ritz basis. Our implementation orthonormalizes the columns of the Rayleigh–Ritz projection matrix R via Gram–Schmidt. To take advantage of all N_p MPI processes, we exploit the default communicator `MPI_COMM_WORLD`.

The (i,j) process of the $p_r \times p_c$ 2D MPI grid holds the submatrices $V_{i,j}$ and $Y_{i,j}$, leading to the following representation of R as a 2D logical array:

$$\widehat{R}_{2\mathrm{D}} = \begin{array}{c c} \begin{array}{cccc} G_0^c & G_1^c & \cdots & G_{p_c-1}^c \end{array} & \\ \begin{bmatrix} \begin{bmatrix} V_{0,0} \\ Y_{0,0} \end{bmatrix} & \begin{bmatrix} V_{0,1} \\ Y_{0,1} \end{bmatrix} & \cdots & \begin{bmatrix} V_{0,p_c-1} \\ Y_{0,p_c-1} \end{bmatrix} \\ \vdots & \vdots & \vdots & \vdots \\ \begin{bmatrix} V_{p_r-1,0} \\ Y_{p_r-1,0} \end{bmatrix} & \begin{bmatrix} V_{p_r-1,1} \\ Y_{p_r-1,1} \end{bmatrix} & \cdots & \begin{bmatrix} V_{p_r-1,p_c-1} \\ Y_{p_r-1,p_c-1} \end{bmatrix} \end{bmatrix} & \begin{array}{c} G_0^r \\ \vdots \\ G_{p_r-1}^r \end{array} \end{array}.$$

The goal is to transform $\widehat{R}_{2\mathrm{D}}$ into an $n \times Nn_{\mathrm{ev}}$ matrix $R_{1\mathrm{D}}$ such that each one of the N_p processes holds a submatrix that has roughly n/N_p rows and Nn_{ev} columns. This can be achieved by the following two-step procedure. First, we perform a gather reduction on the submatrices $\begin{bmatrix} V_{i,j}^T & Y_{i,j}^T \end{bmatrix}^T$, $j = 0, \ldots, p_c - 1$. This reduction is performed independently within each communicator G_i^r, $i = 0, \ldots, p_r - 1$. Second, each process associated with G_i^r discards all rows of the previously reduced matrix except for a unique, contiguous set of rows. We can then write

$$R_{1\mathrm{D}} = \begin{bmatrix} V_{0,0} & \cdots & V_{0,p_c-1} \\ Y_{0,0} & \cdots & Y_{0,p_c-1} \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ V_{p_r-1,0} & \cdots & V_{p_r-1,p_c-1} \\ Y_{p_r-1,0} & \cdots & Y_{p_r-1,p_c-1} \end{bmatrix} = \begin{bmatrix} R_{0,0} \\ \vdots \\ R_{0,p_c-1} \\ \vdots \\ R_{p_r-1,0} \\ \vdots \\ R_{p_r-1,p_c-1} \end{bmatrix}, \tag{4.3}$$

where $R_{i,j}$ is held by the MPI process of rank ip_c+j associated with `MPI_COMM_WORLD`, i.e., the jth process associated with the row communicator G_i^r. This can be done efficiently in a single line of code by calling `MPI_Alltoall` independently within each communicator G_i^r, $i = 0, \ldots, p_r-1$. A graphical illustration of this 2D-to-1D grid remapping is shown in Figure 4.3.

Once the remapping is complete, we apply distributed block Gram–Schmidt to the columns of $R_{1\mathrm{D}}$ using `MPI_COMM_WORLD` and a block size equal to n_{ev}. Then, we map $R_{1\mathrm{D}}$ back to the 2D layout by reversing the above procedure. For further details on parallel Gram–Schmidt, including a discussion of numerical stability, see [4, 9].

4.4. Formation and solution of the projected eigenvalue problem. Finally, we form the projected pencil (R^TAR, R^TMR) and find its eigenvalues. As the projected pencil is small, once it is formed, we compute its eigenvalues serially using the `DSYGVX` routine from `LAPACK` [2]. The remainder of this section is devoted to discussing our approach to forming R^TAR within the 2D distributed-memory data layout described above. The procedure for forming R^TMR is identical.

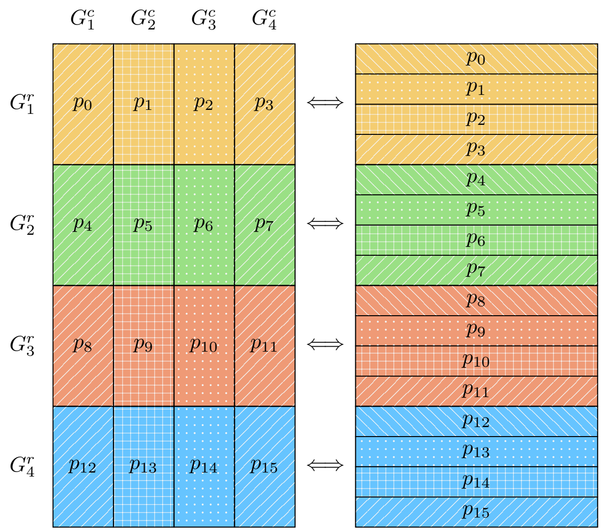

FIG. 4.3. *2D-to-1D (and vice versa) MPI grid mapping. Left: color/pattern layout of a 2D grid of MPI processes with $p_c = p_r = 4$. Right: color/pattern layout of the same grid collapsed into a 1D MPI grid topology.*

We form R^TAR in two phases. Let $R_j = \begin{bmatrix} V_j^T & Y_j^T \end{bmatrix}^T$. In the first phase, we compute $AR = \begin{bmatrix} AR_0 & AR_1 & \cdots & AR_{N-1} \end{bmatrix}$. When $p_c = N$, this operation is embarrassingly parallel since each of the products AR_j, $j = 0, \ldots, N-1$, can be computed independently. Using the rank-based representation of A from (4.1), we write

$$AR_j = \begin{bmatrix} B_1 & E_1 & & & & & \\ E_1^T & C_{1,1} & & C_{1,2} & & & C_{1,p_r} \\ & & B_2 & E_2 & & & \\ & C_{2,1} & E_2^T & C_{2,2} & & & C_{2,p_r} \\ & & & & \ddots & & \\ & & & & & B_{p_r} & E_{p_r} \\ & C_{p_r,1} & & C_{p_r,2} & & E_{p_r}^T & C_{p_r,p_r} \end{bmatrix} \begin{bmatrix} V_{0,j} \\ Y_{0,j} \\ V_{1,j} \\ Y_{1,j} \\ \vdots \\ V_{p_r-1,j} \\ Y_{p_r-1,j} \end{bmatrix}. \tag{4.4}$$

Communication between different MPI processes of G_j^c is point-to-point, and the ith process needs to send $Y_{i,j}$ to the kth process if and only if $C_{k,i} \neq 0$.

The second phase multiplies R^T and AR and stores the matrix product in the root process of `MPI_COMM_WORLD`. To achieve this, we apply the following procedure, which is illustrated in Figure 4.4:

1. Apply `MPI_Allgather` on the submatrices $[AR_j]_i$, $j = 0, \ldots, p_c - 1$, across the row communicator G_i^r, where $[AR_j]_i$ denotes the submatrix of AR_j held by the ith process. Each process associated with G_i^r then has its own copy of the matrix $\begin{bmatrix} [AR_0]_i & [AR_2]_i & \cdots & [AR_{p_c-1}]_i \end{bmatrix}$.
2. The ith process associated with the column communicator G_j^c then computes $Z_{i,j} = R_{i,j}^T \begin{bmatrix} [AR_0]_i & [AR_2]_i & \ldots & [AR_{p_c-1}]_i \end{bmatrix}$ and calls `MPI_Reduce` on the data $Z_{i,j}$ associated with the processes in G_j^c.
3. At the end of the previous step, the kth MPI process associated with G_0^r holds the kth block of rows of the matrix R^TAR. Finally, all processes in G_0^r call `MPI_Gather`, creating R^TAR in the root process.

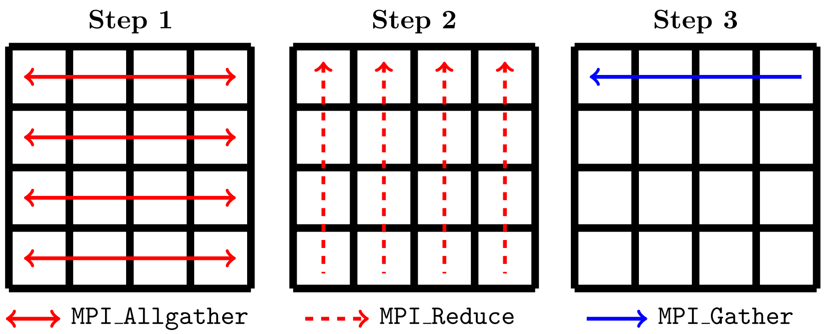

FIG. 4.4. *Communication pattern for the distributed-memory computation of R^TAR and R^TMR using our 2D MPI data layout ($p_r = p_c = 4$). The root process of MPI_COMM_WORLD is located in the upper-left corner.*

5. Numerical experiments. We now illustrate the performance of Algorithm 3.1 in both sequential and distributed-memory computing environments. We performed our experiments on the Minnesota Supercomputing Institute's `Mesabi` cluster. Each node of `Mesabi` is equipped with 64 GB of system memory and two 12-core 2.5 GHz Intel Xeon E5-2680v3 (Haswell) CPUs. We built our code with the Intel ICC 18.0.0 compiler. We used the Intel Math Kernel Library (MKL) for basic matrix operations, including its sparse matrix routines and its implementation of the standard `BLAS` and `LAPACK` libraries for sequential dense matrix operations. While it is possible to exploit shared-memory parallelism, the experiments described below use just one thread per MPI process.

To compute the n_{ev} sought eigenvectors of the spectral Schur complements $S(\chi_j)$, we used `PARPACK` with full orthogonalization and restart dimension $m = 2n_{\mathrm{ev}}$. The linear systems involving the block-diagonal matrix $B(\chi_j)$ were solved with the Intel MKL implementation of the `PARDISO` solver. For the search interval $[\alpha, \beta]$, we set $\alpha = 0$, $\beta = (\lambda_{n_{\mathrm{ev}}} + \lambda_{n_{\mathrm{ev}}+1})/2$ in all experiments.

5.1. Numerical illustration. We first demonstrate the qualitative performance of Algorithm 3.1 on a set of four small problems:

- "APF4686," a standard eigenvalue problem of dimension $n = 4{,}686$ generated by the `ELSES` quantum mechanical nanomaterial simulator[5] [16],
- "Kuu/Muu," a generalized eigenvalue problem of dimension $n = 7{,}102$ from the SuiteSparse matrix collection[6] [10],
- "FDmesh," a standard eigenvalue problem generated by a regular 5-point finite difference discretization of the Laplacian on a square, and
- "FEmesh," a generalized eigenvalue problem obtained by discretizing the Laplacian on a square with linear finite elements.

For the latter two, the discretization fineness was chosen to yield matrices of dimension $n \approx 20{,}000$, and the associated pencils have several eigenvalues of multiplicity 2.

Figure 5.1 plots the relative errors in the eigenvalues returned by Algorithm 3.1 and the corresponding residual norms for the problems "APF4686" (left, $n_{\mathrm{ev}} = 30$) and "Kuu/Muu" (right, $n_{\mathrm{ev}} = 100$) for $N = 2, 4, 6, 8$. Figure 5.2 plots the same quantities for "FDmesh" (left) and "FEmesh" (right), where $n_{\mathrm{ev}} = 100$ in both cases.

[5] http://www.elses.jp
[6] https://sparse.tamu.edu/

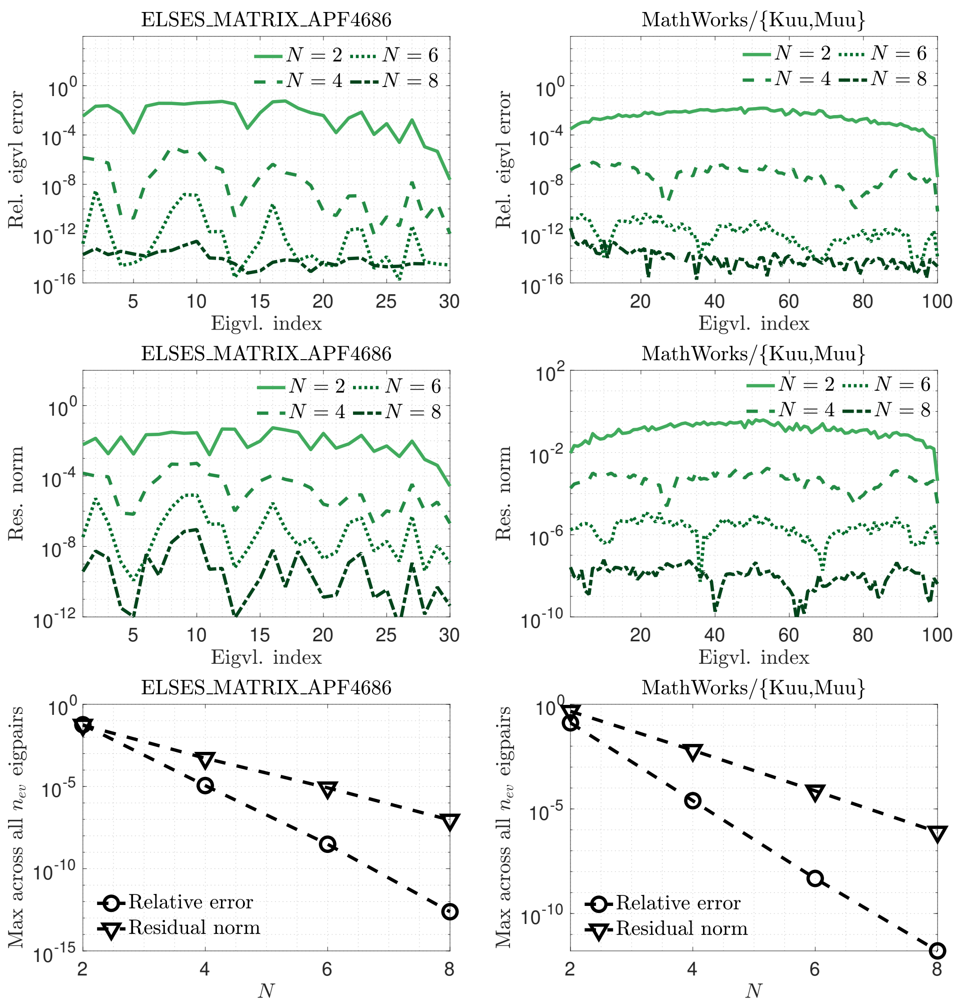

Fig. 5.1. *Relative errors in the eigenvalues returned by Algorithm* 3.1 *(top) and corresponding residual norms (center) for various values of N for the problems "APF*4686*" (left, $n_{\rm ev} = 30$) and "Kuu/Muu" (right, $n_{\rm ev} = 100$). The bottom two figures plot the maximum relative error in the eigenvalues and the maximum residual norm across all $n_{\rm ev}$ eigenpairs.*

In agreement with the discussion in section 3, increasing N leads to greater accuracy in the approximation of the sought eigenpairs. Moreover, all eigenpairs are approximated to comparable accuracies for a given value of N; i.e., the accuracy of an eigenpair is relatively insensitive to the location of the eigenvalue inside $[\alpha, \beta]$.

5.2. Distributed-memory performance. We now illustrate the distributed-memory efficiency of Algorithm 3.1 on a variety of larger problems coming from discretizations of the Laplacian as well as general symmetric matrices and pencils from the SuiteSparse collection. Unless otherwise indicated, throughout the rest of this section, we take $n_{\rm ev} = 100$, and we set the second dimension of the 2D MPI grid to be $p_c = N$. In most of the tests, we report the results with $N = 8$ or $N = 4$. The parallel efficiency of a program executing on $\phi \in \mathbb{N}$ processes is $P(\phi) = T_1/(\phi T_\phi)$, where T_ϕ denotes the wall-clock time for execution on ϕ processes.

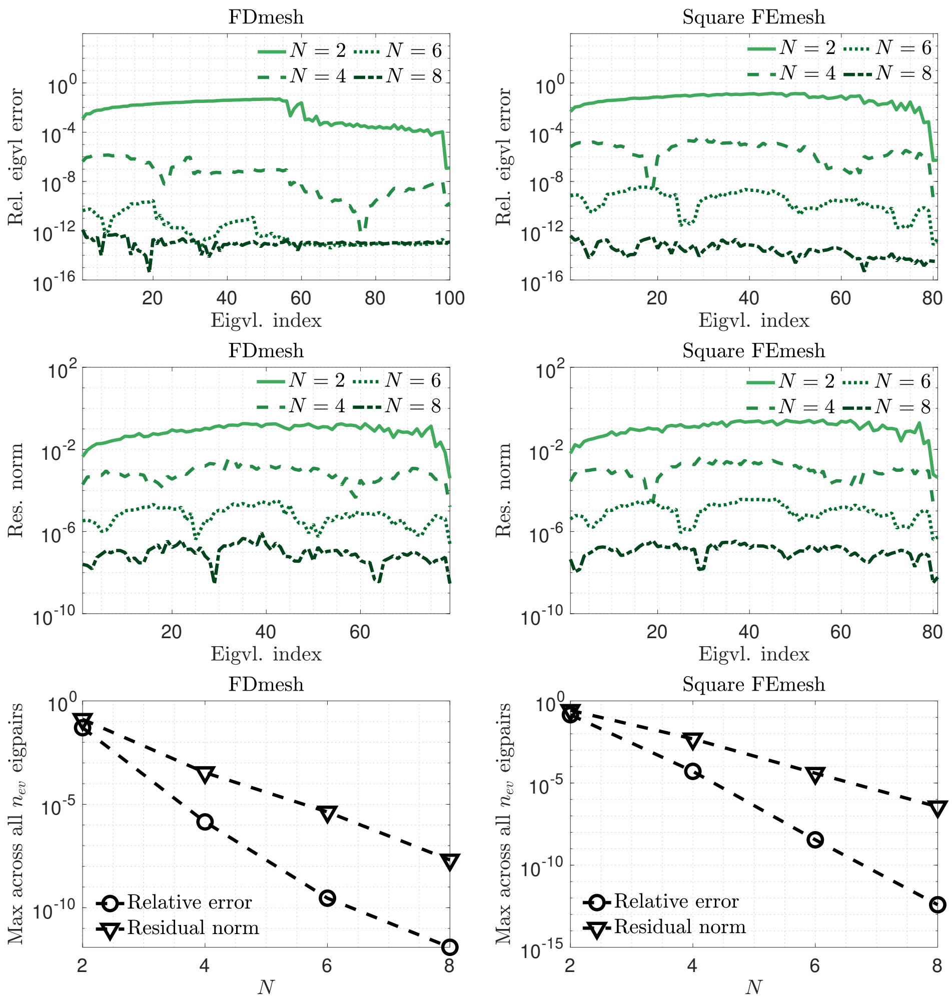

FIG. 5.2. *Relative errors in the eigenvalues returned by Algorithm* 3.1 *(top) and corresponding residual norms (center) for various values of N for the problems "FDmesh" (left) and "FEmesh" (right). The bottom two figures plot the maximum relative error in the eigenvalues and the maximum residual norm across all* n_{ev} *eigenpairs.*

We benchmark Algorithm 3.1 against `PARPACK` applied directly to the pencil (A, M) both with and without shift-and-invert. `PARPACK` requires the application of either M^{-1} (without shift-and-invert) or A^{-1} (with shift-and-invert), and since A and M are distributed, we used a distributed direct solver for these operations. The results reported here were generated using the `MUMPS` package [1], but our code also provides interfaces for `SuperLU_Dist` [32] and the Intel Cluster Sparse Solver (provided in the MKL). For `PARPACK`, we report the wall-clock time and parallel efficiency for a restart length equal to $m = 2n_{\mathrm{ev}}$ with all MPI processes bundled in the default communicator `MPI_COMM_WORLD`. To keep the comparisons fair, the convergence tolerance passed to `PARPACK` for each problem is set to the maximum residual norm returned by Algorithm 3.1.

5.2.1. Eigenvalue problems from finite difference discretizations. First, we apply Algorithm 3.1 to matrices arising from finite difference discretizations of the Dirichlet eigenvalue problem,

Table 5.1
Maximum relative error in the eigenvalues returned by Algorithm 3.1 for the finite difference problems.

	$n = 257 \times 256$	$n = 513 \times 512$	$n = 1025 \times 1024$	$n = 65 \times 64 \times 63$
`SchurCheb(4)`	5.1×10^{-4}	8.2×10^{-5}	1.4×10^{-4}	9.1×10^{-5}
`SchurCheb(8)`	2.3×10^{-9}	2.9×10^{-11}	2.5×10^{-7}	1.9×10^{-10}

$$\begin{aligned} -\Delta u &= \lambda u \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{5.1}$$

where Δ denotes the Laplacian and Ω is either the square $(0,1)^2$ in two dimensions or the cube $(0,1)^3$ in three dimensions. We use the standard 5- and 7-point stencils in two and three dimensions, respectively. All these eigenvalue problems are standard ones, with M equal to the identity matrix.

Our first set of experiments focuses on the strong scaling of Algorithm 3.1. We take $n_{\text{ev}} = 100$ and use $N = 4, 8$ Chebyshev nodes. In our results, we refer to Algorithm 3.1 with $N = 4$ as `SchurCheb(4)` and with $N = 8$ as `SchurCheb(8)`. We first consider three different 2D discretizations with matrix sizes $n = 257 \times 256$, $n = 513 \times 512$, and $n = 1025 \times 1024$, respectively. Table 5.1 lists the maximum relative error in the eigenvalues returned by Algorithm 3.1. Figure 5.3 (left) plots the parallel efficiency of Algorithm 3.1 for $N = 8$, where we report separately the parallel efficiencies associated with (a) computation of the eigenvector matrices Y_j, $j = 0, \ldots, N-1$, (b) orthonormalization of the projection matrix R, and (c) everything else. Since $p_c = N$, the computation of the Y_j is embarrassingly parallel, leading to nearly perfect efficiency for this step. On the other hand, both the orthonormalization of R and the formation of $R^T AR$ require communication among the N_p processes, and their efficiency can deteriorate for larger values of N_p. Note also that the parallel granularity of Algorithm 3.1 is lower for smaller problem sizes, leading to lower efficiencies compared to larger problems.

Figure 5.3 (right) plots the wall-clock time achieved by Algorithm 3.1 for $N = 4, 8$, `PARPACK` with and without shift-and-invert, and the Locally Optimal Block Preconditioned Conjugate Gradient (`LOBPCG`) method as implemented in the `BLOPEX` package of `hypre` [11]. The wall-clock times of `LOBPCG` were obtained with AMG preconditioning, and we present the best (lowest) times after performing extensive tests involving various choices for the hyperparameters and preconditioners. Regarding the performance of `PARPACK`, note that due to the fact that A comes from a 2D discretization, shift-and-invert is generally very fast when the direct solver scales satisfactorily; however, the efficiency of `MUMPS` falls off faster than that of Algorithm 3.1 as N_p increases, and for larger values of N_p, Algorithm 3.1 becomes the fastest and most scalable approach. Similarly, `LOBCPG` is competitive with Algorithm 3.1 for smaller values of N_p but becomes comparatively slower as N_p increases.

Figure 5.4 plots the same quantities for a 3D discretization matrix of size $n = 65 \times 64 \times 63$. The main difference between the 2D and 3D cases is that `PARPACK` without shift-and-invert now converges much faster, leading to lower orthogonalization costs. Moreover, because A is banded, the parallel efficiency of distributed-memory sparse matrix-vector products with A remains high even when $N_p = 256$. Nonetheless, Algorithm 3.1 still attains greater strong scaling efficiency than `PARPACK` (with or without shift-and-invert) and hence will outperform it given enough parallel resources.

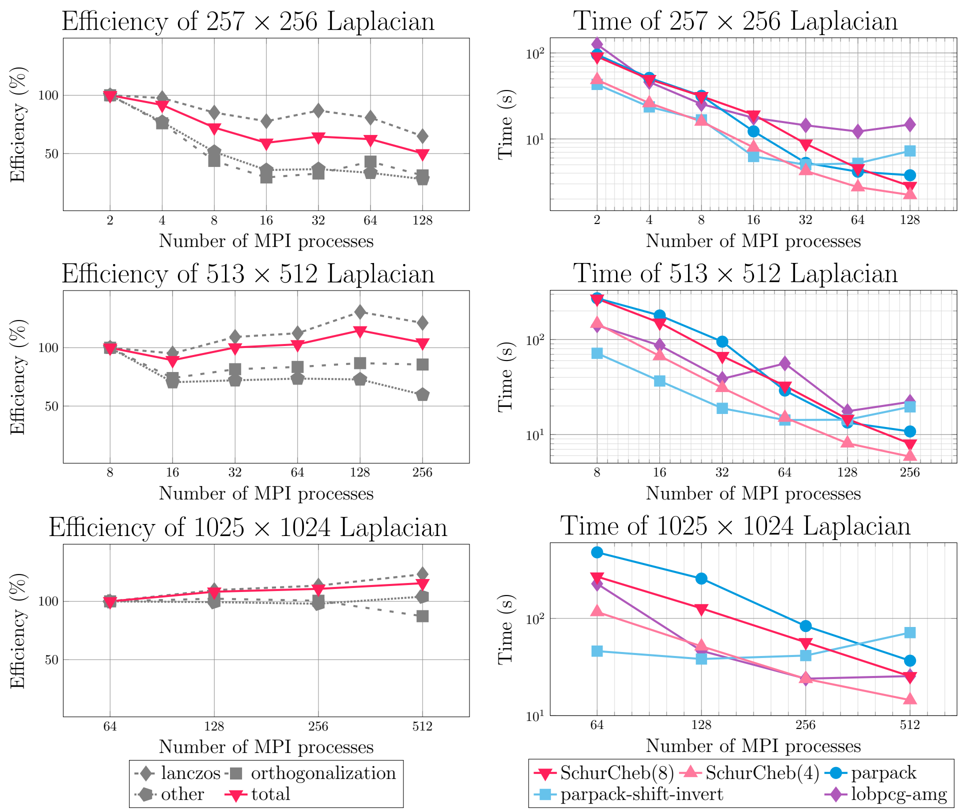

FIG. 5.3. *Left: parallel efficiency of Algorithm* 3.1 *with* $n_{\mathrm{ev}} = 100$ *and* $p_c = N = 8$. *Right: wall-clock time comparison between Algorithm* 3.1 *with* $N = 4, 8$ *and* PARPACK *with and without shift-and-invert. The number of MPI processes ranges from* $N_p = 2$ *to* $N_p = 512$. *The number of partitions is set equal to* $p = 32$ *(*$n = 257 \times 256$*),* $p = 64$ *(*$n = 513 \times 512$*), and* $p = 128$ *(*$n = 1025 \times 1024$*) when* $N = 8$. *The value of* p *is doubled when* $N = 4$ *since each column communicator now has twice as many processes.*

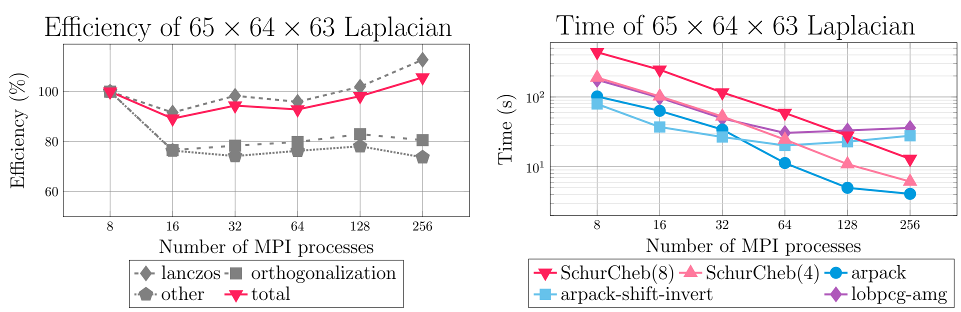

FIG. 5.4. *Left: parallel efficiency of Algorithm* 3.1 *with* $n_{\mathrm{ev}} = 100$ *and* $p_c = N = 8$. *Right: wall-clock time comparison between Algorithm* 3.1 *with* $N = 4, 8$ *and* PARPACK *with and without shift-and-invert. The number of MPI processes ranges from* $N_p = 8$ *to* $N_p = 256$. *The number of partitions is set to* $p = 64$ *(*$N = 8$*) and* $p = 128$ *(*$N = 4$*).*

As Algorithm 3.1 does not need to factor A, it requires considerably less storage than PARPACK with shift-and-invert. Table 5.2 lists the global peak memory consumption for both of these algorithms for the finite difference discretization problems

TABLE 5.2
Peak memory consumption of Algorithm 3.1 and of `PARPACK` with shift-and-invert for the finite difference problems.

	$n = 257 \times 256$ $N_p = 128$	$n = 513 \times 512$ $N_p = 256$	$n = 1025 \times 1024$ $N_p = 512$	$n = 65 \times 64 \times 63$ $N_p = 256$
`SchurCheb(4)`	1.2 GB	2.4 GB	9.3 GB	2.3 GB
`SchurCheb(8)`	2.2 GB	4.6 GB	18.8 GB	4.6 GB
`PARPACK`	21.4 GB	45.0 GB	106.4 GB	46.6 GB

TABLE 5.3
Partitioning information for the test matrices arising from regular finite difference discretizations of the Laplacian in two and three dimensions.

Size	N	p	d	s
1025×1024	8	128	1,002,735	46,865
	4	256	982,871	66,729
513×512	8	64	247,046	15,610
	4	128	240,021	22,635
257×256	8	32	60,722	5,070
	4	64	58,315	7,477
$65 \times 64 \times 63$	8	64	193,420	68,660
	4	128	171,288	90,792

just described. Even with $N = 8$ Chebyshev nodes, Algorithm 3.1 uses 5 to 10 times less memory than shift-and-invert `PARPACK` across all problems.

Table 5.3 presents statistics on the partitioning of the matrices used in the experiments of Figures 5.3 and 5.4. When the number N of Chebyshev nodes is cut from $N = 8$ to $N = 4$, the number p of subdomains is doubled to keep the total number of MPI processes constant. The dimension s of the Schur complement ranges from about 5,000 for the 257×256 2D Laplacian with $p = 8$ up to just over 90,000 for the $65 \times 64 \times 63$ 3D Laplacian with $p = 4$. In all cases, the value of s is considerably (roughly 2 to 10 times) smaller than the dimension d of the corresponding B block.

We now focus on the performance of Algorithm 3.1 when the problem size n and number of partitions p are fixed and N_p varies proportionally to N. We set $p = p_r = 8$ and $p_c = N$, where $N = 2, 4, \ldots, 16$. For this experiment, we consider the 2D discretizations of sizes $n = 257 \times 256$ and $n = 513 \times 512$ and report the wall-clock times for each major operation of Algorithm 3.1 in Figure 5.5. The amount of time spent computing the matrices Y_j and V_j is nearly constant since the maximum number of matrix-vector products (iterations) required by `PARPACK` to compute each Y_j is more or less the same for each N_p (see the solid lines). On the other hand, the amount of time required for orthonormalization and the Rayleigh–Ritz projection both increase due to (a) higher computational complexity and (b) a higher volume of communication among the increasing number of MPI processes.

Next, we evaluate the performance of Algorithm 3.1 when computing different numbers of eigenvalues (different n_{ev}) for the same matrix. We consider the 2D discretizations of sizes $n = 257 \times 256$ and $n = 513 \times 512$. In each group of tests, we fix p, p_r, p_c, and N_p and then vary n_{ev}. For the $n = 257 \times 256$ problem, we take $N_p = 128$ and $p_r = N$ and then set $p = 16$ when $N = 8$ and $p = 32$ when $N = 4$. For the $n = 512 \times 512$ problem, we double p and N_p. Figure 5.6 reports the total wall-clock times for Algorithm 3.1 under these configurations, taking $n_{\text{ev}} = 50, 100, 150, 200,$

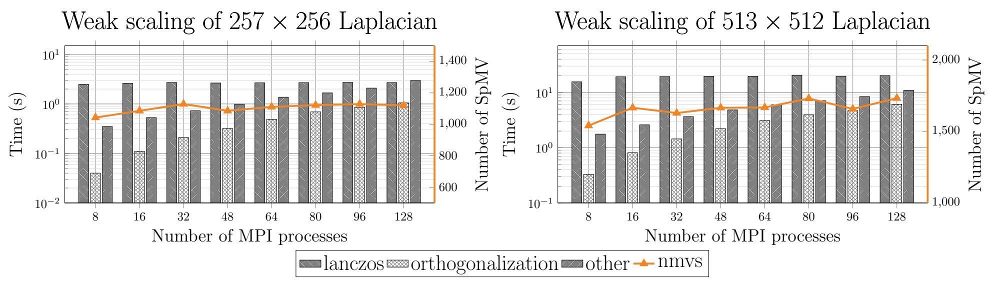

FIG. 5.5. *Weak scaling with respect to* N *(*$p_r = 8$, $p_c = N$*) for two 2D finite difference discretization problems. The number of MPI processes ranges from* $N_p = 8$ *to* $N_p = 128$*. The solid orange lines denote the maximum number of iterations required by* PARPACK *to compute the matrices* Y_j, $j = 0, \ldots, N-1$.

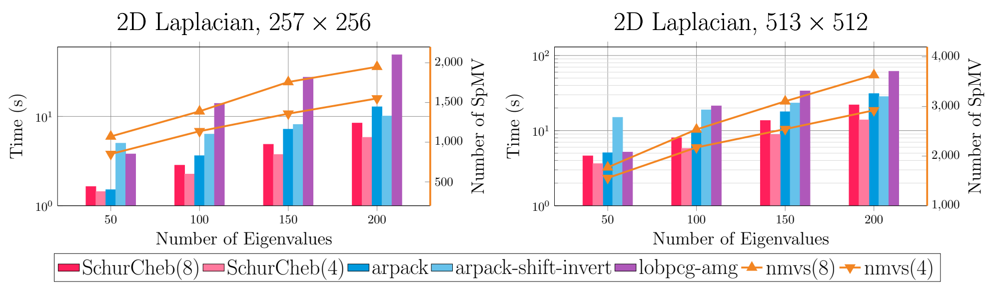

FIG. 5.6. *Scaling with respect to* n_{ev} *for two 2D finite difference discretization problems. The number of MPI processes are* $N_p = 128$ *and* $N_p = 256$*, respectively. The solid orange lines denotes the maximum number of iterations required by* PARPACK *to compute the matrices* Y_j, $j = 0, \ldots, N-1$*, in Algorithm* 3.1.

TABLE 5.4
Wall-clock time breakdown of Algorithm 3.1 *for various 2D MPI grid topologies (RR: Rayleigh–Ritz, GS: Gram–Schmidt).*

(p_r, p_c)	Setup	$Y_{0,\ldots,N-1}$	$V_{0,\ldots,N-1}$	GS	RR	DSYGVX	Total
(128,1)	1.42	26.08	0.35	1.41	1.76	0.14	31.17
(64,2)	0.68	18.06	0.36	1.94	1.81	0.14	23.15
(32,4)	0.32	13.95	0.35	1.71	1.91	0.14	18.41
(16,8)	0.18	13.21	0.35	1.65	2.03	0.14	17.61

as well as those for PARPACK (with and without shift-and-invert) and LOBPCG. The cost of solving the Schur complement eigenvalue problems in Algorithm 3.1 at each Chebyshev node increases as n_{ev} increases. Nonetheless, Algorithm 3.1 still attains wall-clock times that are competitive with PARPACK and LOBPCG.

In the preceding experiments, we took $p_c = N$. As our final experiment in this section, we consider the effect of varying the 2D MPI grid topology. We consider the 2D discretizations of sizes $n = 513 \times 512$. We take $N = 8$, $N_p = p = 128$, $n_{\text{ev}} = 100$ and vary the topology as $(p_r, p_c) = (128,1), (64,2), (32,4), (16,8)$. Table 5.4 lists a breakdown of the wall-clock times for the various parts of Algorithm 3.1 for each topology. The topology $(p_r, p_c) = (128,1)$ processes the N Chebyshev nodes sequentially, one after the other, but uses all N_p MPI processes during the computation of each matrix pair (Y_j, V_j), $j = 0, \ldots, N-1$, taking on average $(26.08 + 0.35)/8 \approx 3.3$ seconds for each. At the other extreme, the topology $(p_r, p_c) = (16,8)$

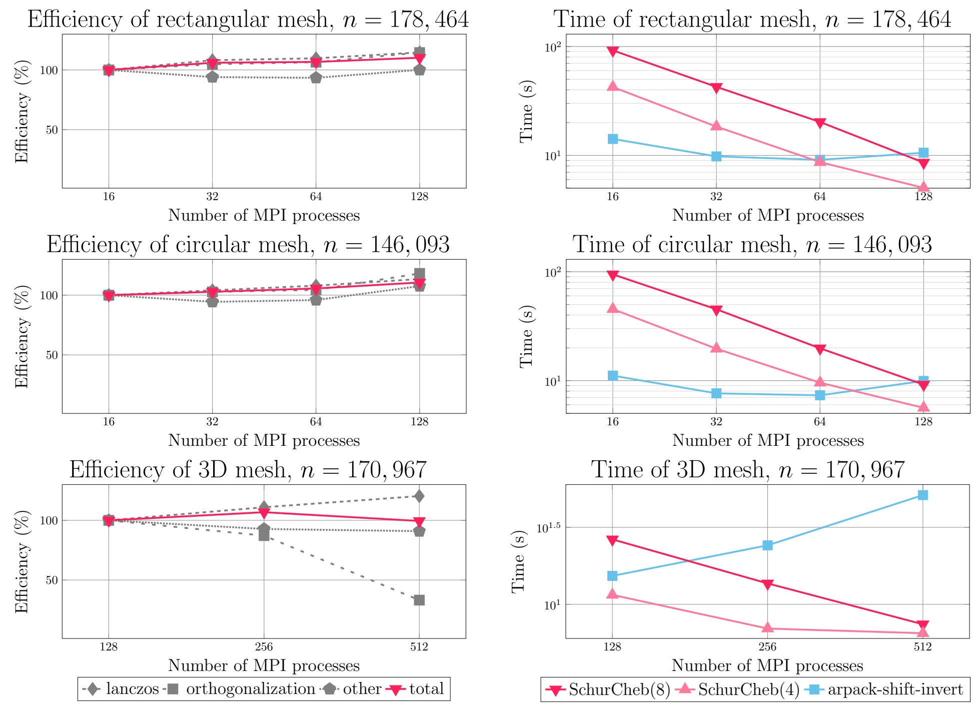

Fig. 5.7. *Left: parallel efficiency of Algorithm* 3.1 *applied to the finite element problems with* $n_{\mathrm{ev}} = 100$ *and* $p_c = N = 8$. *Right: wall-clock time comparison between Algorithm* 3.1 *with* $N = 4$ *and* $N = 8$ *and* `PARPACK` *with shift-and-invert. The number of MPI processes ranges from* $N_p = 8$ *to* $N_p = 512$. *The number of partitions is set equal to* $p = 16$ *for the* 2D *meshes and* $p = 64$ *for the* 3D *mesh.*

processes the N Chebyshev nodes completely in parallel, but now computing each (Y_j, V_j) requires more time—in the worst case, approximately four times as much $(13.21 + 0.35 = 13.56$ seconds)—since only $p_r = 16$ processes are available for parallelization of those computations. Nevertheless, the total time to solution is nearly halved with $(p_r, p_c) = (16, 8)$ versus $(p_r, p_c) = (128, 1)$. Thus, in agreement with our previous results, setting $p_c = N$ is best unless the smaller value of p_r creates a memory bottleneck.

5.2.2. Eigenvalue problems from finite element discretizations. To illustrate the performance of Algorithm 3.1 for generalized eigenvalue problems, we again consider matrices arising from discretizations of (5.1) but with linear finite elements instead of finite differences. In two dimensions, we consider the square $\Omega = (0, 1)^2$ and the disc $\Omega = \{(x, y) \,:\, x^2 + y^2 \leq 1\}$, both meshed with unstructured triangular elements. In three dimensions, we consider the cube $\Omega = (0, 1)^3$, meshed with unstructured tetrahedra.

Figure 5.7 plots the parallel efficiency of Algorithm 3.1 (left) and associated wall-clock times as N_p varies. We also plot the wall-clock time of `PARPACK` with shift-and-invert but omit results for `PARPACK` without shift-and-invert, which required an excessive amount of time to converge for these problems. The small sizes of the problems ($n \approx 150{,}000$) have been chosen intentionally in order to simulate an environment with an abundance of parallel resources. As in the experiments of the previous

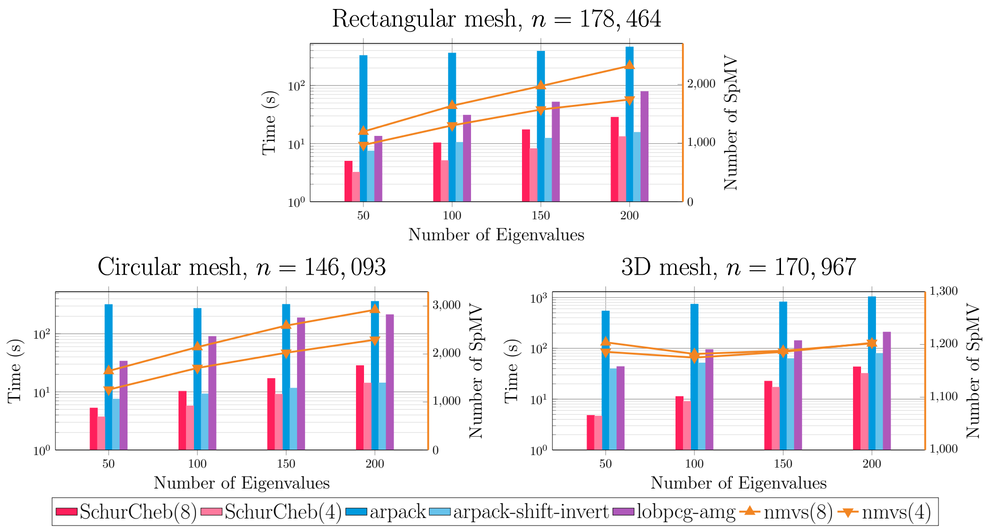

FIG. 5.8. *Scaling with respect to n_{ev} for three finite element problems. The numbers of MPI processes are $N_p = 128$ for the 2D domains and $N_p = 512$ for the 3D domain. The solid red lines denote the maximum number of iterations required by* `PARPACK` *to compute the matrices Y_j, $j = 0, \ldots, N-1$, in Algorithm* 3.1.

TABLE 5.5
Total wall-clock time for Algorithm 3.1 *and* `PARPACK` *with shift-and-invert for the finite element problems with $N_p = 512$, $p = 128$, and $p_c = N$.*

	2D square $n = 1,086,615$	2D disc $n = 845,397$	3D cube $n = 1,351,083$
`SchurCheb(4)`	17.2 s	18.3 s	90.1 s
`PARPACK`	33.6 s	25.9 s	90.3 s

section, Algorithm 3.1 attains high parallel efficiency and scales better than `PARPACK`. The efficiency of the orthogonalization step in Algorithm 3.1 dropped below 50% for the 3D case when $N_p = 512$ due to a large communication-to-computation ratio for the Gram–Schmidt process; nevertheless, the overall efficiency is still close to 100%.

Next, we show the results of a test similar to one in the previous section, wherein Algorithm 3.1 is applied to a given problem for increasing values of n_{ev}. As before, we fix p, p_r, p_c, and N_p for each group of tests and vary n_{ev} as $n_{\text{ev}} = 50, 100, 150, 200$. We use the same finite element problems of the previous experiment set $p_c = N$. When $N = 8$, we use $N_p = 128$ and $p = 16$ for the 2D domains and $N_p = 512$ and $p = 64$ for the 3D domains. When $N = 4$, we double p. The results are reported in Figure 5.8. Again, Algorithm 3.1 attains times to solution that are competitive with `PARPACK`, even though the cost of solving the local eigenvalue problems at each Chebyshev node increases with n_{ev}.

Finally, Table 5.5 lists the wall-clock times for Algorithm 3.1 and `PARPACK` with shift-and-invert on a set of larger finite element problems. For Algorithm 3.1 we report the wall-clock times for the case $N_p = 512$ and $p_c = N = 4$; for `PARPACK`, we report the best (lowest) wall-clock time obtained over several runs with different N_p. Algorithm 3.1 was twice as fast for the 2D problems and about as fast as `PARPACK`

TABLE 5.6
Problems from the SuiteSparse matrix collection. Here, n denotes the size of the pencil (A, M); $\mathrm{nnz}(\cdot)$ counts the number of nonzero entries in its argument; and p denotes the number of partitions for the case $N = 8$.

Dataset	n	p	$\mathrm{nnz}(A)/n$	$\mathrm{nnz}(M)/n$	Application
qa8fk/qa8fm	66,172	16	25.10	25.1	3D acoustics
af_shell3	504,855	64	34.80	1.0	structural problem
tmt_sym	726,713	64	6.99	1.0	electromagnetics
ecology2	999,999	64	5.00	1.0	2D/3D problem
thermal2	1,228,045	64	6.99	1.0	thermal problem

for the 3D problem. Note, though, that in addition to having superior[7] scalability, Algorithm 3.1 also uses much less memory.

5.2.3. Eigenvalue problems from the SuiteSparse collection. Finally, to demonstrate the performance of Algorithm 3.1 for more general matrices, we apply it to several problems taken from the SuiteSparse matrix collection with sizes ranging from $n = 66{,}172$ to $n = 1{,}222{,}045$. Additional details are given in Table 5.6. The "qa8fk/qa8fm" problem is a generalized eigenvalue problem; the other four are standard problems (M is the identity matrix).

Figure 5.9 plots the parallel efficiency (left) and wall-clock time (right) for Algorithm 3.1 on each of these problems. For comparison, we also plot the wall-clock time of `PARPACK` with and without shift-and-invert. As in the previous experiments, Algorithm 3.1 maintains high parallel efficiency up to 512 MPI processes and, provided enough parallel resources, outperforms `PARPACK`. Additionally, Algorithm 3.1 is more memory efficient than shift-and-invert `PARPACK` as N_p increases; Table 5.7 lists the peak memory consumption for both algorithms for the maximum N_p used in each group of tests for each problem. Finally, Table 5.8 lists the maximum error in the eigenvalues returned by Algorithm 3.1 for $N = 4$ and $N = 8$.

6. Conclusion. We presented a distributed-memory Rayleigh–Ritz projection algorithm to compute a few of the smallest eigenvalues and associated eigenvectors of a sparse, symmetric matrix pencil. The algorithm introduces embarrassing parallelism by recasting the problem as one of approximating univariate, vector-valued functions via Chebyshev approximation. The computational work associated with each Chebyshev node can be assigned to a different group of processors, and we described a scheme for doing this using a 2D grid of MPI processes. We discussed several theoretical aspects and implementation details, including how to orthonormalize the Rayleigh–Ritz basis and form the projected eigenvalue problem. Our experiments demonstrated that the proposed algorithm attains good parallel efficiency, superior to `PARPACK`.

While we have focused on computing the smallest eigenvalues of (A, M), our technique can be extended to find eigenvalues in other regions of the spectrum. We leave the details of this extension as a matter for future work. Additionally, we plan to develop a version of this algorithm based on *generalized spectral Schur complements,* in which the matrix Y_j is formed by computing a few eigenvectors of the pencil $\big(S(\chi_j), -S'(\chi_j)\big)$ instead of $S(\chi_j)$ alone. This may allow one to reduce the value of N,

[7]The best wall-clock time of `PARPACK` for the 3D mesh problem was achieved for $N_p = 128$.

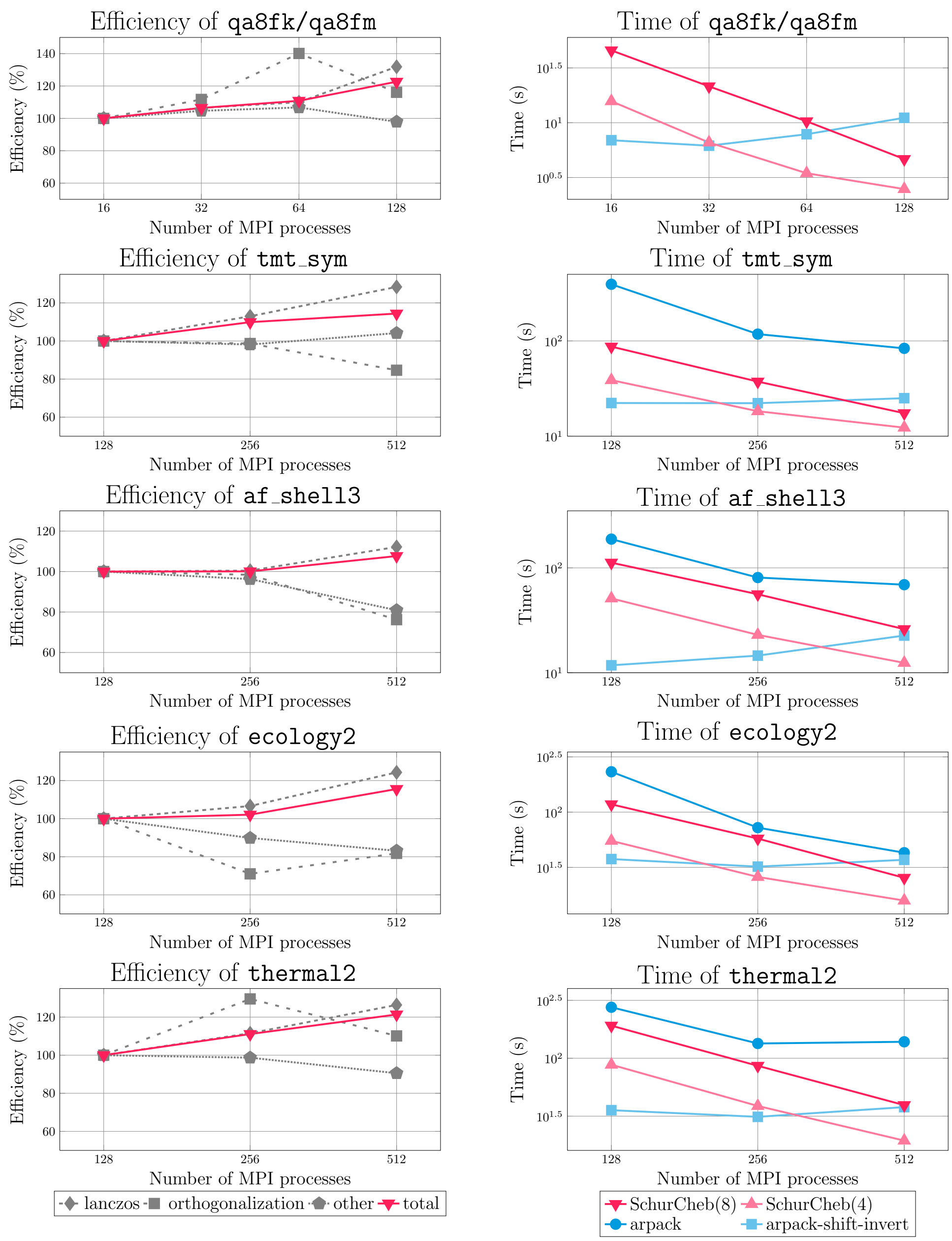

FIG. 5.9. *Left: parallel efficiency of Algorithm* 3.1 *with* $n_{\mathrm{ev}} = 100$ *and* $p_c = N = 8$. *Right: wall-clock time comparison between Algorithm* 3.1 *with* $N = 4$ *and* $N = 8$ *and* ***PARPACK*** *with and without shift-and-invert. The number of MPI processes ranges from* $N_p = 16$ *to* $N_p = 512$.

permitting the use of more parallel resources within each column MPI communicator. We also plan on extending the implementation of our current algorithm so that the computations local to each MPI process are performed using graphics processing units. Finally, we plan on applying our software to problems from real-world applications, e.g., frequency response analysis.

Table 5.7

Peak memory consumption of Algorithm 3.1 *and of* `PARPACK` *with shift-and-invert for the SuiteSparse problems.*

	qa8 $N_p = 128$	af_shell3 $N_p = 512$	tmt_sym $N_p = 512$	ecology2 $N_p = 512$	thermal2 $N_p = 512$
`SchurCheb(4)`	0.7 GB	5.9 GB	6.7 GB	8.9 GB	11.2 GB
`SchurCheb(8)`	1.4 GB	11.9 GB	13.2 GB	17.5 GB	22.2 GB
`PARPACK`	21.7 GB	47.7 GB	50.8 GB	58.7 GB	56.5 GB

Table 5.8

Maximum relative error in the eigenvalues returned by Algorithm 3.1 *for the SuiteSparse problems.*

	qa8	af_shell3	tmt_sym	ecology2	thermal2
`SchurCheb(4)`	3.2×10^{-4}	2.1×10^{-4}	1.6×10^{-4}	1.8×10^{-5}	9.1×10^{-5}
`SchurCheb(8)`	1.0×10^{-8}	3.8×10^{-10}	6.5×10^{-8}	8.9×10^{-9}	1.9×10^{-10}

Acknowledgments. The authors acknowledge the Minnesota Supercomputing Institute (MSI) at the University of Minnesota for providing resources that contributed to the research results reported within this paper (http://www.msi.umn.edu). They also thank the referees for several suggestions that have improved the presentation of this article.

REFERENCES

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *MUMPS: A general purpose distributed memory sparse solver*, in International Workshop on Applied Parallel Computing, Springer, Berlin, Heidelberg, 2000, pp. 121–130, https://doi.org/10.1007/3-540-70734-4_16.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1999, https://doi.org/10.1137/1.9780898719604.

[3] C. A. Beattie, M. Embree, and D. C. Sorensen, *Convergence of polynomial restart Krylov methods for eigenvalue computations*, SIAM Rev., 47 (2005), pp. 492–515, https://doi.org/10.1137/S0036144503433077.

[4] C. Bekas and A. Curioni, *Very large scale wavefunction orthogonalization in density functional theory electronic structure calculations*, Comput. Phys. Commun., 181 (2010), pp. 1057–1068, https://doi.org/10.1016/j.cpc.2010.02.013.

[5] C. Bekas and Y. Saad, *Computation of smallest eigenvalues using spectral Schur complements*, SIAM J. Sci. Comput., 27 (2005), pp. 458–481, https://doi.org/10.1137/040603528.

[6] J. K. Bennighof and R. B. Lehoucq, *An automated multilevel substructuring method for eigenspace computation in linear elastodynamics*, SIAM J. Sci. Comput., 25 (2004), pp. 2084–2106, https://doi.org/10.1137/S1064827502400650.

[7] J. R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comp., (1977), pp. 163–179, https://doi.org/10.1090/S0025-5718-1977-0428694-0.

[8] D. Calvetti, L. Reichel, and D. C. Sorensen, *An implicitly restarted Lanczos method for large symmetric eigenvalue problems*, Electron. Trans. Numer. Anal., 2 (1994), pp. 1–21.

[9] E. Carson, K. Lund, M. Rozložník, and S. Thomas, *Block Gram-Schmidt algorithms and their stability properties*, Linear Algebra Appl., 638 (2022), pp. 150–195, https://doi.org/10.1016/j.laa.2021.12.017.

[10] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), 1, https://doi.org/10.1145/2049662.2049663.

[11] R. D. Falgout and U. M. Yang, *hypre: A library of high performance preconditioners*, in International Conference on Computational Science, Springer, Berlin, Heidelberg, 2002, pp. 632–641, https://doi.org/10.1007/3-540-47789-6_66.

[12] W. Gao, X. S. Li, C. Yang, and Z. Bai, *An implementation and evaluation of the AMLS method for sparse eigenvalue problems*, ACM Trans. Math. Software, 34 (2008), 20, https://doi.org/10.1145/1377596.1377600.

[13] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, 1999.

[14] V. Hernandez, J. E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Trans. Math. Software, 31 (2005), pp. 351–362, https://doi.org/10.1145/1089014.1089019.

[15] W. Heylen, S. Lammens, and P. Sas, *Modal Analysis Theory and Testing*, KU Leuven, Leuven, Belgium, 1997.

[16] T. Hoshi, H. Imachi, A. Kuwata, K. Kakuda, T. Fujita, and H. Matsui, *Numerical aspect of large-scale electronic state calculation for flexible device material*, Jpn. J. Ind. Appl. Math., 36 (2019), pp. 685–698, https://doi.org/10.1007/s13160-019-00358-2.

[17] V. Kalantzis, *Domain Decomposition Algorithms for the Solution of Sparse Symmetric Generalized Eigenvalue Problems*, Ph.D. thesis, University of Minnesota, Minneapolis, MN, 2018, https://doi.org/11299/201170.

[18] V. Kalantzis, *A domain decomposition Rayleigh–Ritz algorithm for symmetric generalized eigenvalue problems*, SIAM J. Sci. Comput., 42 (2020), pp. C410–C435, https://doi.org/10.1137/19M1280004.

[19] V. Kalantzis, *A spectral Newton-Schur algorithm for the solution of symmetric generalized eigenvalue problems*, Electron. Trans. Numer. Anal., 52 (2020), pp. 132–153, https://doi.org/10.1553/etna_vol52s132.

[20] V. Kalantzis, J. Kestyn, E. Polizzi, and Y. Saad, *Domain decomposition approaches for accelerating contour integration eigenvalue solvers for symmetric eigenvalue problems*, Numer. Linear Algebra Appl., 25 (2018), e2154, https://doi.org/10.1002/nla.2154.

[21] V. Kalantzis, R. Li, and Y. Saad, *Spectral Schur complement techniques for symmetric eigenvalue problems*, Electron. Trans. Numer. Anal., 45 (2016), pp. 305–329.

[22] V. Kalantzis, Y. Xi, and L. Horesh, *Fast randomized non-Hermitian eigensolvers based on rational filtering and matrix partitioning*, SIAM J. Sci. Comput., 43 (2021), pp. S791–S815, https://doi.org/10.1137/20M1349217.

[23] V. Kalantzis, Y. Xi, and Y. Saad, *Beyond automated multilevel substructuring: Domain decomposition with rational filtering*, SIAM J. Sci. Comput., 40 (2018), pp. C477–C502, https://doi.org/10.1137/17M1154527.

[24] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.

[25] G. Karypis, K. Schloegel, and V. Kumar, *PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library*, Technical Report TR 97-060, University of Minnesota, Minneapolis, MN, 1997, https://doi.org/11299/215345.

[26] T. Kato, *Perturbation Theory for Linear Operators*, Springer-Verlag, New York, 1966.

[27] J. Kestyn, V. Kalantzis, E. Polizzi, and Y. Saad, *PFEAST: A high performance sparse eigenvalue solver using distributed-memory linear solvers*, in SC'16: Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 178–189, https://doi.org/10.1109/SC.2016.15.

[28] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov, *Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in hypre and PETSc*, SIAM J. Sci. Comput., 29 (2007), pp. 2224–2239, https://doi.org/10.1137/060661624.

[29] J. H. Ko and Z. Bai, *High-frequency response analysis via algebraic substructuring*, Internat. J. Numer. Methods Engrg., 76 (2008), pp. 295–313, https://doi.org/10.1002/nme.2326.

[30] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM, Philadelphia, 1998, https://doi.org/10.1137/1.9780898719628.

[31] R. Li, Y. Xi, L. Erlandson, and Y. Saad, *The eigenvalues slicing library (EVSL): Algorithms, implementation, and software*, SIAM J. Sci. Comput., 41 (2019), pp. C393–C415, https://doi.org/10.1137/18M1170935.

[32] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Math. Software, 29 (2003), pp. 110–140, https://doi.org/10.1145/779359.779361.

[33] S. Lui, *Kron's method for symmetric eigenvalue problems*, J. Comput. Appl. Math., 98 (1998), pp. 35–48, https://doi.org/10.1016/S0377-0427(98)00110-1.

[34] K. J. Maschhoff and D. Sorensen, *A portable implementation of ARPACK for distributed memory parallel architectures*, in Proceedings of the Copper Mountain Conference on Iterative Methods, Vol. 1, 1996.
[35] E. Polizzi, *Density-matrix-based algorithm for solving eigenvalue problems*, Phys. Rev. B, 79 (2009), 115112, https://doi.org/10.1103/PhysRevB.79.115112.
[36] T. Sakurai, Y. Futamura, A. Imakura, and T. Imamura, *Scalable eigen-analysis engine for large-scale eigenvalue problems*, in Advanced Software Technologies for Post-Peta Scale Computing, Springer, Singapore, 2019, pp. 37–57, https://doi.org/10.1007/978-981-13-1924-2_3.
[37] A. Stathopoulos and J. R. McCombs, *PRIMME: PReconditioned Iterative MultiMethod Eigensolver—methods and software description*, ACM Trans. Math. Software, 37 (2010), 21, https://doi.org/10.1145/1731022.1731031.
[38] G. W. Stewart and J.-G. Sun, *Matrix Perturbation Theory*, Academic Press, Boston, MA, 1990.
[39] Y. Su, T. Lu, and Z. Bai, 2*D Eigenvalue Problems* I*: Existence and Number of Solutions*, preprint, https://arxiv.org/abs/1911.08109v1, 2019.
[40] L. N. Trefethen, *Approximation Theory and Approximation Practice*, SIAM, Philadelphia, 2013.
[41] U. von Luxburg, *A tutorial on spectral clustering*, Stat. Comput., 17 (2007), pp. 395–416, https://doi.org/10.1007/s11222-007-9033-z.
[42] D. B. Williams-Young, P. G. Beckman, and C. Yang, *A shift selection strategy for parallel shift-invert spectrum slicing in symmetric self-consistent eigenvalue computation*, ACM Trans. Math. Software, 46 (2020), 35, https://doi.org/10.1145/3409571.
[43] K. Wu and H. Simon, *Thick-restart Lanczos method for large symmetric eigenvalue problems*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 602–616, https://doi.org/10.1137/S0895479898334605.
[44] Y. Xi, R. Li, and Y. Saad, *Fast computation of spectral densities for generalized eigenvalue problems*, SIAM J. Sci. Comput., 40 (2018), pp. A2749–A2773, https://doi.org/10.1137/17M1135542.
[45] C. Yang, W. Gao, Z. Bai, X. S. Li, L.-Q. Lee, P. Husbands, and E. Ng, *An algebraic substructuring method for large-scale eigenvalue calculation*, SIAM J. Sci. Comput., 27 (2005), pp. 873–892, https://doi.org/10.1137/040613767.
[46] H. Zhang, B. Smith, M. Sternberg, and P. Zapol, *SIPs: Shift-and-Invert parallel spectral transformations*, ACM Trans. Math. Software, 33 (2007), 9, https://doi.org/10.1145/1236463.1236464.