Parallel Iterative Refinement in Polynomial Eigenvalue Problems*

Carmen Campos Jose E. Roman

July 18, 2016

Abstract

Methods for the polynomial eigenvalue problem sometimes need to be followed by an iterative refinement process to improve the accuracy of the computed solutions. This can be accomplished by means of a Newton iteration tailored to matrix polynomials. The computational cost of this step is usually higher than the cost of computing the initial approximations, due to the need of solving multiple linear systems of equations with a bordered coefficient matrix. An effective parallelization is thus important, and we propose different approaches for the message-passing scenario. Some schemes use a subcommunicator strategy in order to improve the scalability whenever direct linear solvers are used. We show performance results for the various alternatives implemented in the context of SLEPc, the Scalable Library for Eigenvalue Problem Computations.

1 Introduction

We are interested in the accurate computation of a few eigenpairs (x, λ) of the polynomial eigenvalue problem, defined as

$$P(\lambda)x = 0, \qquad x \neq 0, \tag{1}$$

where $\lambda \in \mathbb{C}$ is the eigenvalue, $x \in \mathbb{C}^n$ is the eigenvector, and $P(\cdot)$ is an $n \times n$ matrix polynomial of degree d. This problem appears in many practical applications, for instance when discretizing a second (or higher) order partial differential equation, and also as an intermediate tool for solving general nonlinear eigenproblems, e.g., via interpolation. Many of the examples described in the NLEVP collection [5] are polynomial eigenproblems. Throughout the paper we will assume that the matrix polynomial is regular, that is, $\det P(\lambda)$ is not identically zero.

Instead of the usual monomial form, in this paper we express the matrix polynomial in terms of a more general polynomial basis,

$$P(\lambda) = \Phi_0(\lambda)A_0 + \cdots + \Phi_d(\lambda)A_d, \tag{2}$$

*This work was partially supported by the Spanish Ministry of Economy and Competitiveness under grant TIN2013-41049-P. Carmen Campos was supported by the Spanish Ministry of Education, Culture and Sport through an FPU grant with reference AP2012-0608. The computational experiments of section 5 were carried out on the supercomputer Tirant at Universitat de València.

where $\{\Phi_j(\lambda)\}_{j=0}^{\infty}$ is a sequence of real polynomials with $\Phi_j(\lambda)$ of degree j satisfying a 3-term recurrence

$$\lambda\,\Phi_j(\lambda) = \alpha_j\,\Phi_{j+1}(\lambda) + \beta_j\,\Phi_j(\lambda) + \gamma_j\,\Phi_{j-1}(\lambda), \quad \text{for } j = 1, 2, \ldots \tag{3}$$

where $\Phi_{-1} \equiv 0$, $\Phi_0 \equiv 1$, and for $j = 0, 1, \ldots$, α_j, β_j and γ_j are real, $\alpha_j > 0$ and $\alpha_j = \frac{c_j}{c_{j+1}}$, with c_j being the leading coefficient of $\Phi_j(\lambda)$ [2, 7]. These include Chebyshev polynomials among others. This approach may be more reliable numerically than the monomial basis in the case of high degree polynomials, especially when the eigenvalues are located on (or close to) an interval of the real axis.

We focus on the particular case of large-scale problems, where the polynomial coefficients A_i are sparse matrices, and only a few eigensolutions are required. The most common approach for this scenario is to apply a projection method on a certain linearization of the matrix polynomial: first, build matrices L_0 and L_1 of order dn such that the eigenvalues of the pencil $L_0 - \lambda L_1$ coincide with those of (1), then get approximate solutions by projecting this linear problem onto a subspace built, e.g., with a Krylov iteration. Krylov methods for the linearized eigenproblem have been addressed in, e.g., [18]. It is possible to formulate variants of well-known Krylov methods that are able to exploit the structure of L_0 and L_1 from the linearization, resulting in very efficient algorithms in terms of memory as well as computational cost [7].

For problems of large dimension, parallel computing is required. We have implemented solvers based on Krylov iterations on the linearization, as sketched above, where the problem matrices A_i as well as the associated vectors are distributed across available processes and message-passing (with MPI) is employed to coordinate the required computations. Our solvers, that are described in detail in [7], have been implemented in SLEPc, the Scalable Library for Eigenvalue Problem Computations [10, 15], which is an extension of PETSc (Portable, Extensible Toolkit for Scientific Computation [3]). In the context of this kind of computations, it is often necessary to perform linear system solves, and this can be done with iterative methods provided by PETSc or, alternatively, with direct methods from a third-party solver such as MUMPS [1].

The overall solution process based on linearizing the polynomial is not guaranteed to be backward stable, even if a backward stable method is employed for the linear eigenproblem [12]. Hence, robust polynomial eigensolvers such as those included in SLEPc must provide an effective way of improving the accuracy of the computed solution. This is done with iterative refinement, where the computed solution is fed as the starting guess for one (or more) Newton iteration. We remark that scaling the coefficient matrices can sometimes improve the conditioning of the linearized eigenproblem, hence improving accuracy, but it is not effective in some problems with degree larger than 2. In our codes, users can choose to perform scaling as described in [4] (see implementation details in [7]), but we do not consider it in this paper since operation of the refinement algorithm is the same regardless of whether the matrices have been scaled or not.

Iterative refinement for the linear eigenvalue problem has been addressed by Tisseur [17]. For the polynomial eigenproblem, iterative refinement can be formulated in terms of a single eigenpair (x, λ) or, more generally, in terms of invariant pairs (X, H). Invariant pairs [14] are a generalization of invariant subspaces for nonlinear eigenvalue problems, and they will be defined in §2 for matrix polynomials. Kressner [14] formulates a Newton iteration that operates on invariant pairs, aiming at refining solutions of nonlinear eigenvalue problems. This method was later particularized to the case of polynomial eigenproblems [6].

In this paper, we provide all the details regarding the implementation of Newton iterative refinement for polynomial eigenproblems in SLEPc, as a way to complement the description of the solvers in [7]. Our implementation is based on [14] and is therefore more general than [6], since we do not restrict ourselves to polynomials expressed in the monomial basis and formulate the methods assuming polynomial bases of the form (3). We focus particularly on the aspects of parallel computing, since the Newton step can be very costly as it usually involves many linear system solves. We propose several alternatives to organize this computation, and analyze how all these solutions scale when the number of processes increase. For this, it will sometimes be useful to organize the participating processes in several subcommunicators, especially if direct linear solvers are to be used.

The rest of the paper is organized as follows. Section 2 provides the formal definition of invariant pair and summarizes the essentials of Krylov methods available in SLEPc to solve (1). Section 3 describes the Newton method for polynomial eigenproblems [6], adapting it to the non-monomial form of the polynomial, (2). Section 4 gives a description of the different variants proposed to solve the linear systems, together with details of the parallel implementation. Computational results are provided in §5. Finally, we wrap up with some concluding remarks.

2 Computing invariant pairs

When solving a linear eigenvalue problem, it is known that computing invariant subspaces instead of several eigenvectors may have better numerical behaviour, e.g., when the matrix of computed eigenvectors has a large condition number. This is also the case when computing a few eigenpairs associated with a polynomial eigenproblem (1), although in this latter case, as explained in [6, 14], the concept of invariant subspace should be substituted with the concept of invariant pair.

Definition 1. Given $(X, H) \in \mathbb{C}^{n\times k} \times \mathbb{C}^{k\times k}$, it is said to be an invariant pair for a regular matrix polynomial defined as in (2) if

$$\mathbb{P}(X, H) := A_0 X\, \Phi_0(H) + A_1 X\, \Phi_1(H) + \cdots + A_d X\, \Phi_d(H) = 0, \tag{4}$$

where $\Phi_i(H)$ stands for the matrix function defined by the polynomial Φ_i, see [11].

For linear eigenproblems, eigenvalues of the H matrix from an invariant pair, (X, H), are also eigenvalues of the linear eigenproblem provided that X has full column rank. For nonlinear eigenproblems, this assumption has to be replaced by minimality [14, lemma 4]. In the case of matrix polynomials expressed in the form (2), we will use a similar concept to guarantee that eigenvalues of H are indeed eigenvalues for the eigenproblem (1).

2.1 Linearization

We focus on methods that approximate the solution of a polynomial eigenproblem of dimension n and degree d via linearization. In these methods the involved vectors have length dn. We will consider that vectors $v \in \mathbb{C}^{dn}$ and tall-skinny matrices $V \in \mathbb{C}^{dn\times k}$ are divided in d blocks of n rows,

$$v = \begin{bmatrix} v^0 \\ \vdots \\ v^{d-1} \end{bmatrix}, \qquad V = \begin{bmatrix} V^0 \\ \vdots \\ V^{d-1} \end{bmatrix}, \tag{5}$$

where $v^i \in \mathbb{C}^n$ and $V^i \in \mathbb{C}^{n\times k}$ for $i = 0, \ldots, d-1$. Throughout the text, we will use superindices to denote each of the blocks of the split form (5).

We will suppose that polynomial eigensolvers used for (1) are based on the following linearization (details can be found in [7]):

$$L(\lambda) = L_0 - \lambda L_1, \tag{6}$$

$$L_0 = \begin{bmatrix} \beta_0 I & \alpha_0 I & & & & \\ \gamma_1 I & \beta_1 I & \alpha_1 I & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \gamma_{d-2} I & \beta_{d-2} I & \alpha_{d-2} I \\ \tilde{A}_0 & \tilde{A}_1 & \tilde{A}_2 & \cdots & \tilde{A}_{d-3} & \tilde{A}_{d-2} & \tilde{A}_{d-1} \end{bmatrix}, \quad L_1 = \begin{bmatrix} I & & & \\ & \ddots & & \\ & & I & \\ & & & c_d A_d \end{bmatrix},$$

with $\tilde{A}_j = -c_{d-1}A_j$ $(j = 0, \ldots, d-3)$, $\tilde{A}_{d-2} = -c_{d-1}A_{d-2} + c_d\gamma_{d-1}A_d$ and $\tilde{A}_{d-1} = -c_{d-1}A_{d-1} + c_d\beta_{d-1}A_d$. This is a strong linearization and therefore $L(\lambda)z = 0$ and (1) share the same eigenvalues with the same algebraic and geometric multiplicities (details in [2] and references therein). Furthermore, the eigenvector z has the structure

$$z = \begin{bmatrix} x \\ \Phi_1(\lambda)x \\ \vdots \\ \Phi_{d-1}(\lambda)x \end{bmatrix}, \tag{7}$$

with x being the corresponding eigenvector of the polynomial eigenproblem (1). Proposition 1 shows that it is possible to extend this expression to invariant pairs.

Proposition 1. *Let (Z, H) be an invariant pair of the linearized problem (6) in which Z has full rank, then the following holds:*

1. *$Z = V_d(Z^0, H)$, where $V_m(X, H)$ is defined for $m \in \mathbb{N}$, $X \in \mathbb{C}^{n\times k}$ and $H \in \mathbb{C}^{k\times k}$ as*

$$V_m(X, H) := \begin{bmatrix} X \\ X\,\Phi_1(H) \\ \vdots \\ X\,\Phi_{m-1}(H) \end{bmatrix}. \tag{8}$$

2. *(Z^0, H) is an invariant pair of the polynomial eigenproblem (1) satisfying that each eigenvalue of H is also an eigenvalue of (1).*

Proof. Since (Z, H) is an invariant pair for the linearization, we have that $L_0Z - L_1ZH = 0$ for L_0 and L_1 given in (6). By equating the first $d-1$ block rows of this equation, we obtain

$$\begin{cases} Z^1 = \alpha_0^{-1}(Z^0H - \beta_0 Z^0), \\ Z^i = \alpha_{i-1}^{-1}(Z^{i-1}H - \beta_{i-1}Z^{i-1} - \gamma_{i-1}Z^{i-2}), \quad i = 2, \ldots, d-1. \end{cases} \tag{9}$$

These equations allow us to prove easily by induction that

$$Z^i = Z^0\,\Phi_i(H), \quad i = 0, 1, \ldots, d-1. \tag{10}$$

To prove the second statement, we equate the last block row of the equation $L_0Z - L_1ZH = 0$, and we obtain that

$$\begin{aligned}
0 &= -c_{d-1}\sum_{i=0}^{d-1} A_iV^i + c_d\gamma_{d-1}A_dV^{d-2} + c_d\beta_{d-1}A_dV^{d-1} - c_dA_dV^{d-1}H = \\
&= -c_{d-1}\sum_{i=0}^{d-1} A_iV^0\,\Phi_i(H) - c_dA_dV^0(\Phi_{d-1}(H)H - \beta_{d-1}\,\Phi_{d-1}(H) - \gamma_{d-1}\,\Phi_{d-2}(H)) = \\
&= -c_{d-1}\sum_{i=0}^{d-1} A_iV^0\,\Phi_i(H) - c_d\alpha_{d-1}A_dV^0\,\Phi_d(H) = -c_{d-1}\sum_{i=0}^{d} A_iV^0\,\Phi_i(H),
\end{aligned}$$

from where we conclude that (V^0, H) is an invariant pair of (1). On the other hand, for an eigenpair of H, (y, λ), we have that (Zy, λ) is an eigenpair of the linearized problem (Z has full rank), so it has form (7) and we conclude that there exists some x eigenvector of (1) associated with λ. □

Proposition 1 states that it is always possible to extract an invariant pair of the polynomial eigenvalue problem (1) from one for the linearized problem, by taking the first block Z^0. Other extraction alternatives that make use of other blocks Z^i have been proposed in [6] for polynomial eigenproblems defined in terms of the monomial basis, and their counterparts for non-monomial bases are implemented in SLEPc.

The converse of Proposition 1 is also true:

Proposition 2. *Let (X, H) be an invariant pair of the polynomial eigenproblem* (1)*, then $Z := V_d(X, H)$ is an invariant pair for the linearized problem* (6)*.*

Proof. The proof is a simple verification of the equality $L_0Z = L_1ZH$. As in Proposition 1, the first $d-1$ row blocks of this equality are checked using the recurrence (3), and the last one using the condition of (X, H) satisfying (4). □

Remark. As a consequence of Propositions 1 and 2 we have that, for an invariant pair of (1), (X, H), such that $V_d(X, H)$ has full column rank, every eigenvalue of H is also an eigenvalue of (1). In this case, taking as reference the definition given in [6], we say that (X, H) is a *minimal* invariant pair of (1).

A property needed to ensure convergence in the Newton process described in §3 is the concept of simple invariant pair. A minimal invariant pair of (1) is said to be simple if the algebraic multiplicity of each eigenvalue of H matches the algebraic multiplicity of these same eigenvalues in (1) (that is, the multiplicity as a root of the characteristic polynomial $\det P(\lambda)$). Since the linearization (6) is a strong linearization for (1), if the computed invariant pair for the linearized eigenproblem is simple, then the corresponding invariant pair for the polynomial eigenproblem (1) will also be simple.

2.2 Krylov methods for the linearized polynomial eigenproblem

The Newton process described in §3 starts from an approximate simple invariant pair for the polynomial eigenproblem, and it improves its accuracy iteratively. In this section, we briefly review the methods we use to compute the initial approximate invariant pair. These methods are described in [7], and all of them are based on solving the linearized problem (6) using the Krylov-Schur method and extracting a minimal invariant pair for (1) from the one computed for (6).

Krylov-Schur [16] is an implicitly restarted variant of the Arnoldi method. Starting from an initial vector $v \in \mathbb{C}^n$, Arnoldi generates an orthogonal basis $\{v_1, \ldots, v_{k+1}\}$ of the Krylov subspace $\mathcal{K}_{k+1}(M, v) := \text{span}\{v, Mv, \ldots, M^k v\}$, and the projected matrix $H_k = V_k^* M V_k$ verifying,

$$MV_k = V_k H_k + \beta_k v_{k+1} e_k^*, \tag{11}$$

where $\beta_k \in \mathbb{R}$, $M = L_1^{-1} L_0$ (or $M = (L_0 - \sigma L_1)^{-1} L_1$ if a shift-and-invert transformation is used), $V_k := [v_1, \ldots, v_k]$, and $e_k^* = [0, 0, \ldots, 0, 1]$.

When, for a particular value of k, the norm of the residual $MV_k - V_k H_k = \beta_k v_{k+1} e_k^*$ is small enough, the Krylov-Schur process is considered to be converged and then (V_k, H_k) is an approximate minimal invariant pair for (6) in which the columns of V_k are orthonormal.

The main two variants of the SLEPc polynomial solvers described in [7] differ in the way that the v_j vectors are stored. These variants are, on one hand, Plain Arnoldi that stores full-sized Arnoldi vectors of dimension dn, and, on the other hand, the TOAR variant that generates an orthonormal set, $U_{k+d} \in \mathbb{C}^{n \times (k+d)}$ and matrices $\{G_{k+1}^i\}_{i=0}^{d-1} \subset \mathbb{C}^{(k+d) \times (k+1)}$, from which it reconstructs each block of the Krylov basis V_{k+1} as

$$V_{k+1}^i = U_{k+d} G_{k+1}^i, \quad i = 0, \ldots, d-1. \tag{12}$$

In a more compact form, the Krylov vectors are expressed as

$$V_{k+1} = (I_d \otimes U_{k+d}) G_{k+1}. \tag{13}$$

We are interested in showing this notation here because it will be referenced later. Other details of the methods can be found in [7].

3 Newton refinement for invariant pairs

In this section, we explain in detail how the iterative refinement of invariant pairs is carried out in SLEPc's Krylov-based polynomial eigensolvers. We use the Newton iteration for nonlinear eigenproblems described in [14] assuming that the functions f_i defining the nonlinear eigenproblem $(f_0(\lambda)A_0 + \cdots + f_d(\lambda)A_d)x = 0$ are the polynomials Φ_i that define the polynomial eigenproblem (1). When it is possible, we simplify some of the associated computations giving expressions similar to those in [6].

Computing a minimal invariant pair of (1) is equivalent to obtaining $(X, H) \in \mathbb{C}^{n \times k} \times \mathbb{C}^{k \times k}$ such that

$$\mathbb{P}(X, H) = 0, \qquad \text{and} \tag{14a}$$

$$\mathbb{V}(X, H) := W^* V_d(X, H) - I_k = 0, \tag{14b}$$

for some matrix $W \in \mathbb{C}^{dn \times k}$ with full column rank, and $\mathbb{P}$ defined in (4).

Applying results in Kressner [14] we obtain that once an approximation $(\tilde{X}, \tilde{H})$ of a simple invariant pair (X, H) of (1) has been computed, it can be used as the starting guess for the Newton method applied to the system of nonlinear equations (14), obtaining in this way a refined solution of (1) closer to (X, H). Provided that the initial approximation $(\tilde{X}, \tilde{H})$ is sufficiently close to (X, H), this method generates a sequence of iterates, $\{(X_i, H_i)\}_{i \in \mathbb{N}}$ that converges quadratically to (X, H). These iterates are given by

$$(X_{i+1}, H_{i+1}) = (X_i, H_i) - (\mathbb{L}(X_i, H_i))^{-1}(\mathbb{P}(X_i, H_i), \mathbb{V}(X_i, H_i)), \tag{15}$$

where $\mathbb{L}(X,H) := (\mathrm{D}\,\mathbb{P}(X,H), \mathrm{D}\,\mathbb{V}(X,H))$ being $\mathrm{D}\,\mathbb{P}(X,H)$ and $\mathrm{D}\,\mathbb{V}(X,H)$ the Fréchet derivatives, in (X,H), of $\mathbb{P}$ and $\mathbb{V}$, respectively. The (local) quadratic convergence of the Newton method is a consequence of [14, Theorem 10] which proves that the linear operator $\mathbb{L}(X,H)$ is invertible for simple invariant pairs.

Algorithm 1 Newton method for refining invariant pairs

```
Input: Initial pair $(X_0,H_0) \in \mathbb{C}^{n\times k} \times \mathbb{C}^{k\times k}$ such that $V_d(X_0,H_0)^* V_d(X_0,H_0) = I_k$
Output: Approximate solution $(X_{i+1},H_{i+1})$ to (14)
1: $W \leftarrow V_d(X_0,H_0)$
2: for $i = 0,1,\ldots,$ maxit do
3:     Compute residual $R \leftarrow \mathbb{P}(X_i,H_i)$
4:     Compute $(\Delta X,\Delta H)$ such that $\mathbb{L}(X_i,H_i)(\Delta X,\Delta H) = (R,0)$
5:     $\tilde{X}_{i+1} \leftarrow X_i - \Delta X, \quad \tilde{H}_{i+1} \leftarrow H_i - \Delta H$
6:     Compute compact QR decomposition $V_d(\tilde{X}_{i+1},\tilde{H}_{i+1}) = WT$
7:     $X_{i+1} \leftarrow \tilde{X}_{i+1}T^{-1}, \quad H_{i+1} \leftarrow T\tilde{H}_{i+1}T^{-1}$
8:     Check convergence, exit if satisfied
9: end for
```

Algorithm 1 shows the procedure described in [14] to iteratively compute an invariant pair of (1), starting from an approximate simple invariant pair. This algorithm works with orthonormal $V_d(X_i,H_i)$ and takes $W := V_d(X_i,H_i)$ so that $\mathbb{V}(X_i,H_i) = 0$ at each iteration. For this, it computes a compact QR decomposition of $V_d(\tilde{X}_{i+1},\tilde{H}_{i+1})$ and updates the approximate solution $(\tilde{X}_{i+1},\tilde{H}_{i+1})$ accordingly. Note that for doing this it is not necessary to explicitly compute the QR factorization of $V_d(X_i,H_i)$ of size $(dn \times k)$. The matrix T can be computed in a cheaper way if we decompose $X_i = UG_i$ being $U \in \mathbb{C}^{n\times k}$ with orthonormal columns and $G_i \in \mathbb{C}^{k\times k}$. In this case we have that $V_d(X_i,H_i) = (I_d \otimes U)V_d(G_i,H_i)$ for $V_d(G_i,H_i) := \begin{bmatrix} G_i\,\Phi_0(H_i) \\ \vdots \\ G_i\,\Phi_{d-1}(H_i) \end{bmatrix}$ and T can be obtained from the QR factorization of $V_d(G_i,H_i) = \tilde{U}T$.

The most expensive step in Algorithm 1 (step 4) involves the solution of a system of linear matrix equations

$$\begin{cases} \mathrm{D}\,\mathbb{P}(X_i,H_i)(\Delta X,\Delta H) = \mathbb{P}(X_i,H_i) \\ \mathrm{D}\,\mathbb{V}(X_i,H_i)(\Delta X,\Delta H) = 0. \end{cases} \tag{16}$$

More explicitly, for $\mathbb{P}$ defined in (4), the system to solve at each refinement iteration is

$$\begin{cases} \mathbb{P}(\Delta X,H_i) + \displaystyle\sum_{j=0}^{d} A_j X_i\,\mathrm{D}\,\Phi_j(H_i)(\Delta H) = \mathbb{P}(X_i,H_i) \\ (W^0)^*\Delta X + \displaystyle\sum_{j=1}^{d-1}(W^j)^*\left(\Delta X\,\Phi_j(H_i) + X_i\,\mathrm{D}\,\Phi_j(H_i)(\Delta H)\right) = 0, \end{cases} \tag{17}$$

where, for $j > 0$, $\mathrm{D}\,\Phi_j(H_i)$ represents the Fréchet derivative of Φ_j in H_i, which can be obtained recursively from (3) as:

$$\mathrm{D}\,\Phi_j(H_i)(\Delta H) = \begin{cases} 0, & j=0 \\ \alpha_0^{-1}\Delta H, & j=1 \\ \alpha_{j-1}^{-1}\big(\mathrm{D}\,\Phi_{j-1}(H_i)(\Delta H)(H_i-\beta_{j-1}I)+\Phi_{j-1}(H_i)\Delta H- & \\ \qquad -\gamma_{j-1}\,\mathrm{D}\,\Phi_{j-2}(H_i)(\Delta H)\big), & j>1. \end{cases} \tag{18}$$

To solve (17), we use the *forward substitution* technique described in [14, 6]. It requires matrix H_i being triangular, which is always possible by computing the complex Schur form of H_i and updating X_i properly. In this case, the columns of ΔX and ΔH are successively computed as solutions of k linear systems of dimension $n+k$, updating the equation right-hand side at each step. For example, post-multiplying (17) by e_1 produces the linear system

$$\begin{cases} P(h_{11})\Delta x_1 + \displaystyle\sum_{j=0}^{d} A_j X_i\,\mathrm{D}\,\Phi_j(H_i)(\Delta H)e_1 = r_1 \\ (W^0)^*\Delta x_1 + \displaystyle\sum_{j=1}^{d-1}(W^j)^*\left(\Delta x_1\,[\Phi_j(H_i)]_{11} + X_i\,\mathrm{D}\,\Phi_j(H_i)(\Delta H)e_1\right) = f_1, \end{cases} \tag{19}$$

where Δx_1 and Δh_1 are the first columns of ΔX and ΔH, respectively, h_{11} and $[\Phi_j(H_i)]_{11}$ denote the $(1,1)$ element of H_i and $\Phi_j(H_i)$ (both of them upper triangular matrices), r_1 is the first column of the residual $\mathbb{P}(X_i,H_i)$, and $f_1=0$. Defining $\{\mathrm{D}\,\Phi_j(H_i)\}_{11}$, for $j=1,\ldots,$ as the triangular matrix verifying

$$\mathrm{D}\,\Phi_j(H_i)(C)e_1 = \{\mathrm{D}\,\Phi_j(H_i)\}_{11}\,Ce_1, \quad \forall C\in\mathbb{C}^{k\times k}, \tag{20}$$

results in a linear system from where it is possible to obtain the first columns of ΔX and ΔH,

$$\begin{bmatrix} P(h_{11}) & \sum_{j=0}^{d} A_j X_i\,\{\mathrm{D}\,\Phi_j(H_i)\}_{11} \\ \sum_{j=0}^{d-1}\Phi_j(h_{11})(W^j)^* & \sum_{j=1}^{d-1}(W^j)^* X_i\,\{\mathrm{D}\,\Phi_j(H_i)\}_{11} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta h_1 \end{bmatrix} = \begin{bmatrix} r_1 \\ f_1 \end{bmatrix}. \tag{21}$$

Matrices (20) can be obtained recursively from (18):

$$\{\mathrm{D}\,\Phi_0(H_i)\}_{11} = 0, \quad \{\mathrm{D}\,\Phi_1(H_i)\}_{11} = \alpha_0^{-1}I_k, \tag{22}$$
$$\{\mathrm{D}\,\Phi_{j+1}(H_i)\}_{11} = \alpha_j^{-1}\left((h_{11}-\beta_j)\{\mathrm{D}\,\Phi_j(H_i)\}_{11} + \Phi_j(H_i) - \gamma_j\{\mathrm{D}\,\Phi_{j-1}(H_i)\}_{11}\right), \quad j>0.$$

After computing Δx_1 and Δh_1 the right-hand side of (17) is updated before proceeding with the second column of ΔX_i and ΔH_i. To compute the successive columns Δx_p and Δh_p, $p=2,\ldots,k$, we form the corresponding systems analog to (21) by substituting h_{11} by h_{pp} (also in (22)), and replacing the right-hand side $\left[\begin{smallmatrix} r_1 \\ f_1 \end{smallmatrix}\right]$ by the one computed applying consecutive updates according to

$$r_p = \mathbb{P}(X_i,H_i)e_p - \sum_{q=1}^{p-1}\left(\sum_{j=0}^{d} A_j\left(\Delta x_q\,[\Phi_j(H_i)]_{qp} + X_i\,[\mathrm{D}\,\Phi_j(H_i)(Z_q)]_p\right)\right), \tag{23}$$

$$f_p = -\sum_{q=1}^{p-1}\left(\sum_{j=1}^{d-1}(W^j)^*\left(\Delta x_q\,[\Phi_j(H_i)]_{qp} + X_i\,[\mathrm{D}\,\Phi_j(H_i)(Z_q)]_p\right)\right),$$

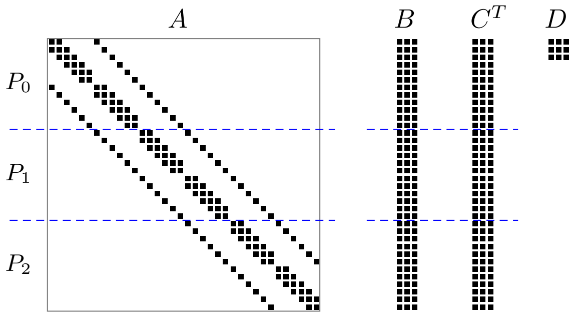

Figure 1: Parallel distribution of matrices and vectors.

where $[\Phi_j(H_i)]_{qp}$ denotes the (q, p) element of $\Phi_j(H_i)$, and $[\mathrm{D}\,\Phi_j(H_i)(Z_q)]_p$ the pth column of $\mathrm{D}\,\Phi_j(H_i)(Z_q)$, being $Z_q := \Delta x_q e_q^T$.

In the case of having a simple eigenpair, (x_i, λ_i), of (1), it can also be seen as a simple invariant pair of size $k = 1$ (clearly $V_d(x_i, \lambda_i)$ has full column rank). That motivates two refining variants when computing a set of q eigenpairs which are included in our solver. The first one is the multiple variant, which refines the invariant pair of dimension $k = q$, yielding an invariant pair of the same dimension. On the other hand, the simple variant, which refines each computed simple eigenpair individually by solving q systems in the form (21) with dimension $k = 1$. Both options require the solution of bordered linear systems in which the leading block of order n is a nearly singular matrix. Aiming to minimize the time required by the solution of these systems, in this work we evaluate several forms to carry out these solves.

4 Solving the correction equation

In this section, we describe several methods that we have considered for the linear systems arising in the forward substitution method used to solve the correction equation (17), explained in §3. We will also discuss several alternatives relative to their parallel implementation.

The linear systems to be solved, (21), have the form

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \tag{24}$$

where the coefficient matrix is a bordered matrix with blocks $A \in \mathbb{C}^{n \times n}$, $B \in \mathbb{C}^{n \times k}$, $C \in \mathbb{C}^{k \times n}$ and $D \in \mathbb{C}^{k \times k}$, with $k \ll n$. The leading block A is nearly singular. The parallel distribution of these four submatrices is depicted in Fig. 1. We store D as a sequential matrix (every process owns a copy), and B, C are stored as an array of k parallel vectors. In PETSc, parallel vectors are distributed by blocks, with each process owning a contiguous range of indices. Regarding the sparse block A, it is stored as a standard PETSc matrix, with every process owning a contiguous range of rows.

The first alternative, that will be referred to as the **explicit matrix** approach, corresponds to explicitly building the whole matrix involved in (21) as a PETSc matrix. This allows using any of the PETSc linear solvers and preconditioners, including direct solvers, which may be seen as an

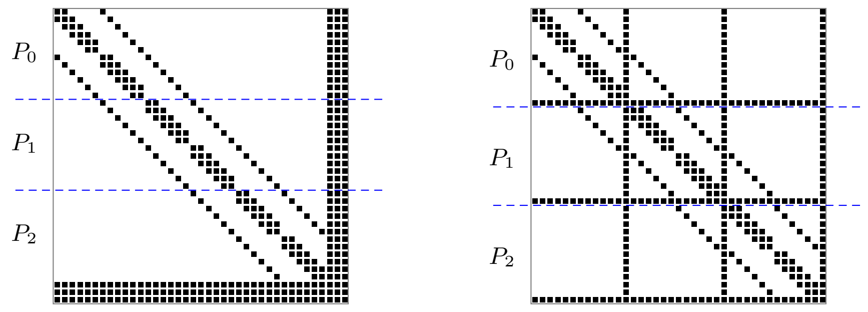

Figure 2: Illustration of the parallel distribution of the matrix created in the explicit matrix approach before (left) and after (right) applying the symmetric permutation.

advantage. However, this approach presents several drawbacks. On one hand, the coefficient matrix of the linear system to be solved is not very sparse, compared to the sparsity of $P(\cdot)$, since it has been bordered with dense stripes. This affects the fill-in that is produced in the factorization when a direct method is used, and it also has an impact on the performance of parallel matrix-vector multiplication since processors owning any of the fully populated rows require the whole distributed vector to carry out its computational part in this operation. On the other hand, when the created matrix is distributed by blocks of consecutive rows across the involved processes (as mentioned above) then it will produce load imbalance that can seriously penalize the overall performance.

Aiming to reduce the load imbalance in this first approach, an appropriate symmetric permutation of the matrix is distributed among the processes. The permutation is chosen in such a way that the fully populated rows are evenly distributed across the available processes, by placing them right after the local rows of the leading block A assigned to them. Fig. 2 shows the matrix before and after applying the permutation. With this approach, all involved vectors are subject to the same permutation.

The second option we have evaluated uses the **Schur complement** of the trailing diagonal block D in (24),

$$S := A - BD^{-1}C, \tag{25}$$

to compute x_1 and x_2 from

$$Sx_1 = y_1 - BD^{-1}y_2, \tag{26}$$

$$x_2 = D^{-1}(y_2 - Cx_1). \tag{27}$$

Note that a similar scheme using the Schur complement of the leading diagonal block A in (24) is not appropriate since it implies linear solves with the nearly singular matrix A.

The matrix S (25) is dense so it should not be explicitly computed. This fact limits the methods available to solve the linear system involved in (26) with the matrix S, which cannot be solved via a direct solver. Instead, iterative methods such as GMRES or any of the Krylov methods provided by PETSc are adequate for the solves. To build the preconditioner needed for the iterative methods we have used the approximation to S given by $P := A - \mathrm{diag}(BD^{-1}C)$, where the operator $\mathrm{diag}(M)$ represents a matrix whose diagonal elements are the same as the matrix M, and have zeros outside the diagonal.

A third alternative that enables the use of direct methods when solving (21), is the **mixed block elimination** (MBE) method described in [8, 9]. This method solves bordered linear systems in the form (24) making solves with the nearly singular leading block matrix and its transpose. Algorithm 2 shows the process followed for this method to solve a linear system with a one-dimensional border,

$$\begin{bmatrix} A & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}. \tag{28}$$

Algorithm 2 Mixed block elimination (MBE) method

```
Input: $A \in \mathbb{C}^{n\times n}$, $b, y_1 \in \mathbb{C}^{n\times 1}$, $c \in \mathbb{C}^{1\times n}$ and $d, y_2 \in \mathbb{C}$ defining the linear system (28)
Output: $x \in \mathbb{C}^{n+1}$ solution of (28) ($x = \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]$)
 1: Solve $A^T v = c^T$
 2: $\delta \leftarrow d - v^T b$
 3: Solve $Aw = b$
 4: $\rho \leftarrow d - cw$
 5: $p_2 \leftarrow (y_2 - v^T y_1)/\delta$
 6: $g_1 \leftarrow y_1 - bx_2$
 7: $g_2 \leftarrow y_2 - dp_2$
 8: Solve $Az = g_1$
 9: $q_2 \leftarrow (g_2 - cz)/\rho$
10: $x_1 \leftarrow z - wq_2$
11: $x_2 \leftarrow p_2 + q_2$
```

To solve a linear system with a wider border, this method works recursively decreasing the dimension of the border in the linear systems that it generates at each step. For example, to solve a linear system (24) for a border of dimension 2,

$$\begin{array}{c} \\ {\scriptstyle n} \\ {\scriptstyle 1} \\ {\scriptstyle 1} \end{array} \begin{array}{c} \begin{array}{ccc} {\scriptstyle n} & {\scriptstyle 1} & {\scriptstyle 1} \end{array} \\ \left[\begin{array}{cc|c} A & b_1 & b_2 \\ c_1 & d_{11} & d_{12} \\ \hline c_2 & d_{21} & d_{22} \end{array}\right] \end{array} \left[\begin{array}{c} x_{1:n} \\ x_{n+1} \\ \hline x_{n+2} \end{array}\right] = \left[\begin{array}{c} y_{1:n} \\ y_{n+1} \\ \hline y_{n+2} \end{array}\right], \tag{29}$$

using Algorithm 2, the method performs 3 solves with the submatrix

$$\begin{bmatrix} A & b_1 \\ c_1 & d_{11} \end{bmatrix} \tag{30}$$

of dimension $n+1$, the vectors $b := \left[\begin{smallmatrix} b_2 \\ d_{12} \end{smallmatrix}\right]$, $c := \left[\begin{smallmatrix} c_2 & d_{21} \end{smallmatrix}\right]$ and the right-hand side vector $g \in \mathbb{C}^{n+1}$ computed from y with several updates (steps 1, 3 and 8 of Algorithm 2). These linear systems are one-dimensional bordered systems as in (28) and they can directly be solved using Algorithm 2. Each solve with matrix (30) makes two solves (with A and A^T) that are independent of the right-hand side, thus, the three required solves with matrix (30) and right-hand sides b, c and g have common computations that can be shared.

In the case of avoiding repeated solves by storing intermediate calculations, the MBE method requires $2k+1$ solves of dimension n, to solve a bordered linear system of dimension $n+k$ with a border of dimension k. When using this method in the forward substitution process, the number of overall linear solves required by this method could be seen as a strong limitation to the dimension of the invariant pair to refine. However, in the results of §5 we will see that for moderate values of k the mixed block elimination shows a good behaviour, compared to the other methods. Moreover, most computations in the MBE method can be done independently of the right-hand side, since $2k$ out of $2k+1$ solves required for this method only involve data from the system matrix. This fact represents better opportunities of parallelism, as it alleviates inherent sequentiality of other approaches. In particular, the use of subcommunicators will be especially beneficial in this case, as discussed below.

The forward substitution process used to solve the correction equation implies solving k linear systems of size $n+k$ that, in some cases, need to be solved using a direct method. In this case, the overall parallel performance can be seriously penalized due to the limited scalability of factorizations such as LU, as well as the associated triangular solves. To reduce such limitation we have studied the possibility of adding a second level of parallelism by splitting the set of MPI processes into several subgroups, so that each new subgroup will be responsible for solving some of the linear systems required by the forward substitution procedure. For this, the matrices defining the polynomial eigenproblem are redundantly replicated in each subgroup, and so are the set of distributed vectors X (the small matrix H is fully stored by each process).

This subcommunicator approach is applicable in both the simple and multiple refinement schemes. In the case of the simple refinement, when refining k individual eigenpairs instead of an invariant pair of dimension k, the linear systems come from separate refinement processes and there is complete independence between the resolution of the linear systems, which can be freely distributed and solved among the subgroups of processes. However, this is not the case for multiple refinement, due to the forward substitution procedure, where each computed solution updates the right-hand sides of the subsequent linear systems. When forming the right-hand side, $\left[\begin{smallmatrix} r_p \\ f_p \end{smallmatrix}\right]$, of the pth system to be solved, update (23) is required, involving the previously computed $(\Delta x_q, \Delta h_q)$. This forces the solves to be carried out in a sequential way. Despite that, the building and solving of these systems also entails several time consuming operations that can be carried out in parallel by the different subgroups, as described in Algorithm 3.

The copy of the parallel matrices and vectors in step 2 of Algorithm 3 represents a data redistribution requiring communication involving all processes. For example, Fig. 3 shows how a matrix distributed in 5 processes is duplicated redundantly into two subcommunicators with 3 and 2 processes, respectively.

In steps 4–11 of Algorithm 3, each subgroup is assigned a different column of ΔX and ΔH to be computed, calculates the associated bordered matrix and performs a matrix factorization required for using a direct method. The MBE method factorizes the leading submatrix A of (24), whereas the explicit matrix approach factorizes the full bordered matrix. After that, the dependence between the right-hand sides in the involved linear systems forces a sequential stage for the solves (steps 12–17 in Algorithm 3). Despite that, in the case of using MBE, the solution of the bordered systems (24) entails the solution of $2k$ linear systems with the leading block A which are independent of the right-hand side $\left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right]$. Therefore, these solves can be moved to the concurrent phase (step 9 of Algorithm 3), and only one right-hand side dependent solve is left in the sequential stage (step 15 of Algorithm 3). After a column of ΔX and ΔH has been computed, it is redistributed from the corresponding subcommunicator to the original global communicator.

The gain when activating this second level of parallelism is limited by the size of the invariant pair to refine (or the number of eigenpairs in the case of simple refinement) but, as will be shown in §5, it relieves the limited scalability of solves based on direct methods.

Algorithm 3 Subcommunicators splitting in the forward substitution procedure

```
Input: Number of MPI subcommunicators $n_g$
1: Create $n_g$ subgroups identified with $id_g = 0, \dots, n_g - 1$
2: Duplicate matrices $\{A_j\}_{j=0}^{d}$ and vectors $X_i$ for each subgroup
3: for $l = 0, \dots, (k-1)/n_g$ do
4:   for all subgroups in parallel do
5:     $p \leftarrow l * n_g + id_g$
6:     if $p < k$ then
7:       Compute matrix blocks corresponding to (21) for the $p$th column of $X_i$ and $H_i$
8:       Factorize the leading block $P(h_{pp})$
9:       [Only in MBE] Solve $2k$ right-hand side independent linear systems
10:    end if
11:  end for
12:  for $g = 0, 1, \dots, n_g - 1$ do
13:    $q \leftarrow l * n_g + g$
14:    All processes in original communicator compute right-hand side $\left[\begin{smallmatrix} r_q \\ f_q \end{smallmatrix}\right]$ and send it to subgroup $g$
15:    Subgroup $g$ performs triangular solve with the updated right-hand side received
16:    Subgroup $g$ redistributes computed column $(\Delta x_q, \Delta h_q)$ among the original communicator
17:  end for
18: end for
```

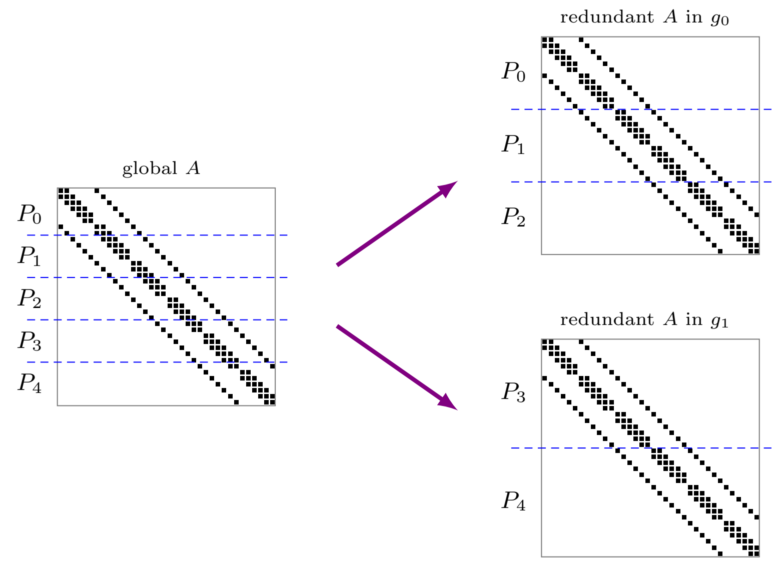

Figure 3: Parallel redistribution of matrices in the case of two subcomunicators g_0 and g_1, where the parent communicator has 5 processes.

Table 1: Description of the test problems used for the performance analysis, indicating the degree (*deg*) of the matrix polynomial, the dimension (*dim*) of the coefficient matrices, the requested number of eigenvalues (*nev*) eigenvalues selected from different parts of the spectrum (closest to target σ).

name	deg	dim	nev	σ
qd_cylinder	3	751,900	6	0.1
qd_pyramid-186k	5	186,543	8	0.4
qd_pyramid-1.5m	5	1.5 mill	8	0.4
sleeper	2	1 mill	8	-0.9
pdde_stability	2	640,000	32	-1
acoustic_wave_2d	2	999,000	16	0
loaded_string	10	1 mill	8	0

5 Computational results

In this section we present the results of several computational experiments, comparing the various methods for iterative refinement. We focus especially on the scalability of the different approaches. The computer used for the executions was Tirant, an IBM cluster consisting of 512 JS21 blade computing nodes, each of them with two 64-bit PowerPC 970MP dual core processors running at 2.2 GHz with 4 GB of memory, interconnected with a low latency Myrinet network. All runs used a single MPI process per node. Our implementations have been developed on top of SLEPc 3.6. Apart from SLEPc 3.6 and PETSc 3.6, we also used MUMPS 5.0 whenever an LU or Cholesky factorization was required. All software has been compiled with gcc-4.6.1 and MPICH2.

We have used several test problems to assess the robustness and performance of our solvers. Table 1 summarizes the test cases, providing information about the degree of the polynomial, the matrix size, the number of requested eigenpairs and the target value around which eigenvalues are sought. The first problems arise in the computation of the electronic structure of quantum dots via discretization of the Schrödinger equation [13]. The rest belong to the NLEVP collection [5]. All problems use the monomial basis for the matrix polynomial except the last one (loaded_string) which is expressed in the Chebyshev basis since it is obtained from a nonlinear eigenproblem via polynomial interpolation. All computations have been carried out in complex arithmetic (although our code supports real arithmetic provided that the eigenvalues to be refined are real).

For measuring the quality of the computed eigenpairs we use the relative backward error, which is defined for an approximate right eigenpair (x, λ) of P as in (2) by

$$\eta(x,\lambda) = \frac{\|P(\lambda)x\|_2}{\left(\sum_{i=0}^{d} |\,\Phi_i(\lambda)|\|A_i\|_2\right)\|x\|_2}. \tag{31}$$

In practical computations, we replace the matrix 2-norm by the ∞-norm. In Table 2 we show the maximum backward error before and after a single step of iterative refinement is performed on the problems of Table 1. In all cases, iterative refinement provides a significant improvement in accuracy with respect to the initial approximations.

Table 2 also shows some sample timing results with various refinement methods, when computing

Table 2: Computational results for iterative refinement. Initial approximations are computed with tolerance *tol*. The maximum backward error is shown for eigenpair approximations before (η_{KS}) and after (η_{NR}) refinement, together with the running time (in seconds) for both the computation of initial approximations (t_{KS}) and refinement (t_{NR}). The number of subcommunicators is *sub*.

name	tol	t_{KS}	η_{KS}	ref. method	sub	t_{NR}	η_{NR}
qd_cylinder	10^{-8}	1837	9×10^{-12}	none	1	-	-
qd_cylinder	10^{-4}	1033	2×10^{-6}	multiple-schur	1	169	2×10^{-13}
qd_pyramid-186k	10^{-4}	50	3×10^{-8}	simple-schur	1	25	8×10^{-14}
qd_pyramid-1.5m	10^{-4}	591	4×10^{-6}	multiple-schur	1	152	7×10^{-12}
sleeper	10^{-6}	42	8×10^{-10}	multiple-mbe	8	29	5×10^{-17}
pdde_stability	10^{-4}	185	1×10^{-6}	simple-mbe	1	277	3×10^{-13}
acoustic_wave_2d	10^{-6}	111	1×10^{-8}	multiple-exp-lu	1	837	3×10^{-14}
loaded_string	10^{-4}	40	2×10^{-6}	simple-mbe	8	36	6×10^{-17}

a single step of refinement starting from initial approximations computed with the Krylov solver (TOAR) with a tolerance *tol* using 8 MPI processes (arranged in *sub* subcommunicators). We can see that there are cases where the computation of the initial approximations takes much more time than the refinement, while in other problems the situation is the opposite. In general, refining is computationally demanding, so it is recommended only if initial approximations have a bad accuracy. In our experiments, we usually set a large tolerance (10^{-4}) for the Krylov solver so that it provides unusually inaccurate approximations and hence improvement of the refinement is more apparent.

Next we provide results of parallel scalability for several test cases. In all cases, we analyze strong scaling, i.e., the problem size is the same for any number of processes.

The two representative test cases from the quantum dot simulation are: qd_cylinder (cubic polynomial from a cylinder quantum dot discretized with finite differences on a uniform mesh) and qd_pyramid (quintic polynomial from a pyramid quantum dot discretized with finite volumes). In these problems, preconditioned iterative solvers for linear systems perform quite well. In all results shown below, we use Bi-CGStab with block Jacobi preconditioner (using an incomplete LU factorization with zero fill-in in each subdomain).

Figure 4 shows the parallel execution time with increasing number of processes for the qd_cylinder test case. The figure compares the situation where no iterative refinement is carried out (eigenvalue approximations are computed with a tolerance of 10^{-8}) and the case where one step of multiple iterative refinement is done (using the Schur complement approach) on initial approximations computed with $tol = 10^{-4}$. As already seen in Table 2, in this case it pays off to refine, since the extra iterations required by TOAR to reach 10^{-8} are expensive. What we are interested now is to see that both alternatives scale similarly in this case.

We now compare several methods when solving the qd_pyramid problem. Figure 5 shows execution times for two problem sizes. In the left panel, we can appreciate that in this problem using a direct method for the linear solves is counterproductive, because factorization time dominates and that is why the corresponding lines overlap. Regarding the alternatives based on iterative linear solves, the ones based on the Schur complement scale much better than the ones that build the explicit matrix (which require an increasing number of iterations in the linear solver with increas-

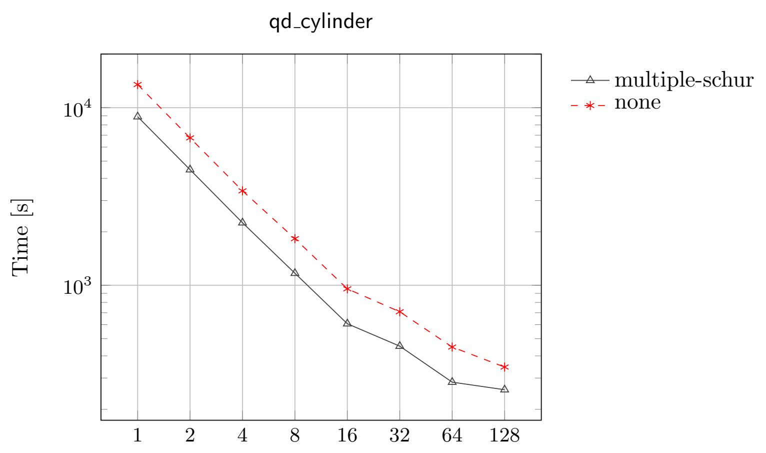

Figure 4: Parallel scaling (up to 128 processes) of the computation of 6 eigenpairs of the qd_cylinder problem with and without refinement.

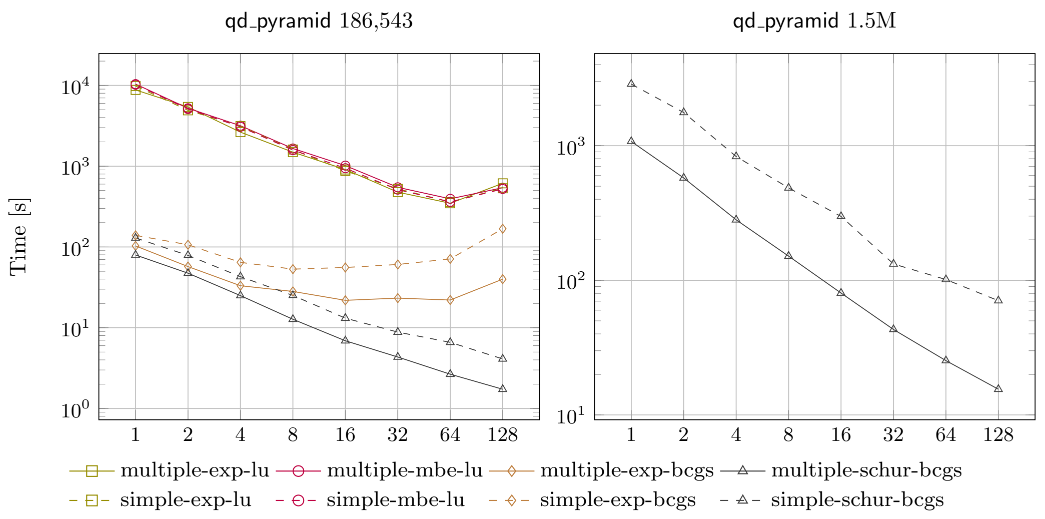

Figure 5: Parallel scaling (up to 128 processes) of different iterative refinement methods working on the qd_pyramid problem of dimension 186,543 (left) and 1.5 million (right).

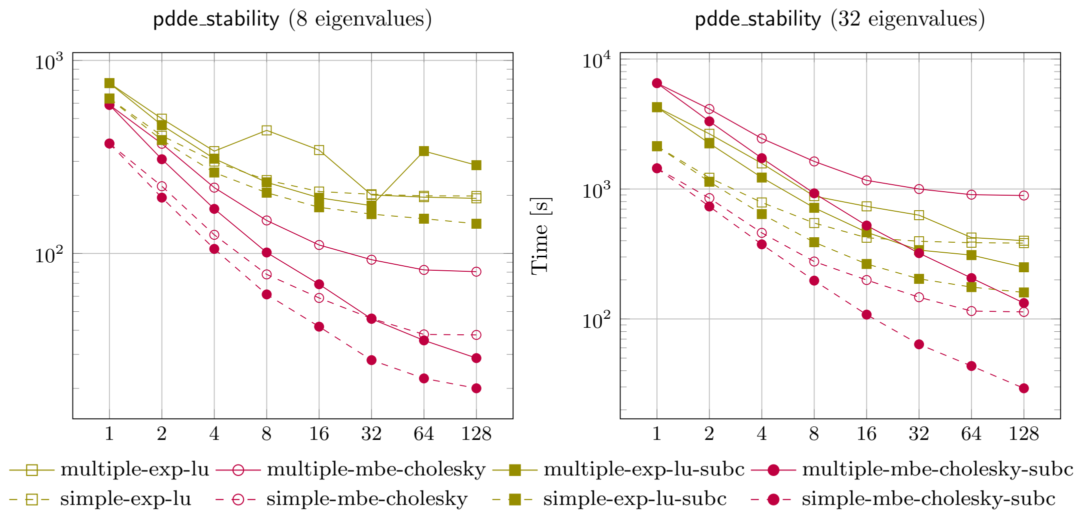

Figure 6: Parallel scaling (up to 128 processes) of different iterative refinement methods when refining 8 (left) or 32 (right) eigenvalues of the pdde_stability problem. Legend: *simple*/*multiple* refers to refinement of single eigenpairs/invariant pair; linear systems via explicit matrix (*exp*) or mixed block elimination (*mbe*), with LU or Cholesky decomposition; *subc* indicates that more than one subcommunicator is being used.

ing number of processes). Scalability of the Schur complement versions is best displayed in the 1.5 million problem in Figure 5 (right). We see that multiple refinement scales linearly, and is faster than simple refinement (as we will see below, it is more often the other way round). In this problem size, direct linear solvers were not viable.

We now focus on the performance of the mixed block elimination method. Figure 6 shows a comparison of this strategy with respect to the explicit matrix approach in the pdde_stability problem (we do not show results of the Schur complement variant because we could not make any iterative method converge in this problem). When refining 8 eigenvalues, MBE is always faster than the explicit matrix approach, for any number of processes. As pointed out in §4, the MBE scheme (with multiple refinement) is penalized when the number of eigenvalues to refine increases, so we also show on Figure 6 (right) the results corresponding to refinement of 32 eigenvalues. In this latter case, multiple refinement with MBE is slower than the explicit matrix method, as expected. Note that simple refinement with MBE (dashed lines) does not have this drawback and continues to be below the explicit matrix lines.

Even though MBE may be slower, it turns out that its scalability can be better provided that subcommunicators are employed. This was the goal of Algorithm 3. Figure 6 illustrates this with the lines whose name ends with "subc". For drawing these lines, we execute with p processes and a number of subcommunicators equal to $min(nev, p)$ (that is, subcommunicators composed of just 1 process until the number of processes is larger that the number of eigenvalues to refine). In both panels of Figure 6 we see that subcommunicators significantly improve the scalability of MBE variants, with a reduction of time almost proportional to the number of processes. In the case of refining 32 eigenvalues, even multiple MBE refinement beats the explicit matrix counterparts when

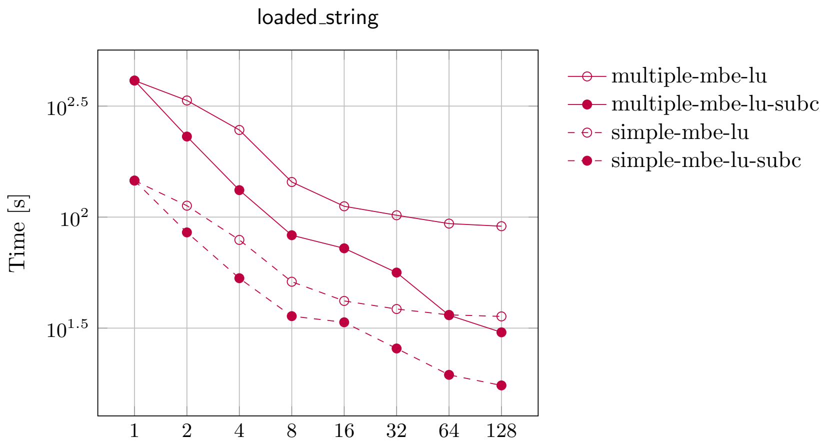

Figure 7: Parallel scaling (up to 128 processes) of different iterative refinement methods operating on the loaded_string problem. Legend: *simple/multiple* refers to refinement of single eigenpairs/invariant pair; linear systems are solved via mixed block elimination (*mbe*) with LU decomposition; *subc* indicates that more than one subcommunicator is being used.

a sufficient number of processes are employed.

We finish this section with the analysis of the loaded_string problem, whose associated polynomial has degree 10 and is represented with a non-monomial basis. As in the previous case, this problem also requires using a direct linear solver (since the iterative methods and preconditioners that we tried had convergence difficulties), so scalability will be limited and it will be beneficial to split the processes into subcommunicators. Figure 7 shows execution times for the MBE strategy, for both simple and multiple refinement, with and without subcommunicators. It is evident that using subcommunicators confers a higher degree of scalability, both for simple and multiple refinement. One could expect an improved scalability for even more processes if the number of eigenvalues to refine was larger (it is 8 in this case).

6 Conclusions

We have implemented Newton-based iterative refinement in the context of polynomial eigenvalue problems, with a number of alternative schemes for the most computationally expensive part, namely the solution of a sequence of linear systems. This method represents a valuable addition to the Krylov solvers for polynomial eigenvalue problems that we have implemented in SLEPc, presented in [7], making it possible to attain very good accuracy in cases where the Krylov method itself can have difficulties. The refinement step can sometimes be very costly, but we have proposed several ways of arranging the computation to exploit parallelism. The scheme based on the Schur complement scales very well. In the case of requiring direct linear solvers, the mixed block elimination (MBE) strategy with the MPI processes arranged in subcommunicators can scale with good performance up to 128 processes or even more, for both the simple and multiple refinement strategies. In this way, we are able to perform iterative refinement of very large scale problems, possibly with large degree polynomials, even in the case that the computed solution consists of tens of eigenpairs.

The developed codes allow solving a general nonlinear eigenvalue problem by first building a polynomial eigenproblem (e.g., via Chebyshev interpolation in a prescribed interval), possibly of high degree, that is then solved and its solution refined iteratively with the methods presented in this paper. We remark that in this particular case it would be better to perform iterative refinement from the perspective of the original nonlinear problem, rather than the polynomial problem. Although not discussed here, we have already implemented the simple refinement strategy for nonlinear problems, leaving multiple refinement as a topic for future research.

References

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

[2] A. Amiraslani, R. M. Corless, and P. Lancaster. Linearization of matrix polynomials expressed in polynomial bases. *IMA Journal of Numerical Analysis*, 29(1):141–157, 2009.

[3] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, K. Rupp, B. Smith, S. Zampini, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

[4] T. Betcke. Optimal scaling of generalized and polynomial eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 30(4):1320–1338, 2008.

[5] T. Betcke, N. J. Higham, V. Mehrmann, C. Schröder, and F. Tisseur. NLEVP: a collection of nonlinear eigenvalue problems. *ACM Transactions on Mathematical Software*, 39(2):7:1–7:28, 2013.

[6] T. Betcke and D. Kressner. Perturbation, extraction and refinement of invariant pairs for matrix polynomials. *Linear Algebra and its Applications*, 435(3):514–536, 2011.

[7] C. Campos and J. E. Roman. Parallel Krylov solvers for the polynomial eigenvalue problem in SLEPc. Submitted, 2015.

[8] W. Govaerts. Stable solvers and block elimination for bordered systems. *SIAM Journal on Matrix Analysis and Applications*, 12(3):469–483, 1991.

[9] W. Govaerts and J. D. Pryce. Mixed block elimination for linear systems with wider borders. *IMA Journal of Numerical Analysis*, 13(2):161–180, 1993.

[10] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, 2005.

[11] N. J. Higham and A. H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159–208, 2010.

[12] N. J. Higham, R.-C. Li, and F. Tisseur. Backward error of polynomial eigenproblems solved by linearization. *SIAM Journal on Matrix Analysis and Applications*, 29(4):1218–1241, 2007.

[13] Feng-Nan Hwang, Zih-Hao Wei, Tsung-Ming Huang, and Weichung Wang. A parallel additive Schwarz preconditioned Jacobi-Davidson algorithm for polynomial eigenvalue problems in quantum dot simulation. *Journal of Computational Physics*, 229(8):2932–2947, 2010.

[14] D. Kressner. A block Newton method for nonlinear eigenvalue problems. *Numerische Mathematik*, 114:355–372, 2009.

[15] J. E. Roman, C. Campos, E. Romero, and A. Tomas. SLEPc users manual. Technical Report DSIC-II/24/02–Revision 3.6, D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2015.

[16] G. W. Stewart. A Krylov–Schur algorithm for large eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 23(3):601–614, 2001.

[17] F. Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1038–1057, 2001.

[18] F. Tisseur and K. Meerbergen. The quadratic eigenvalue problem. *SIAM Review*, 43(2):235–286, 2001.