A Parallel Davidson-Type Algorithm for Several Eigenvalues[†]

Leonardo Borges and Suely Oliveira

Computer Science Department

Texas A&M University

College Station, TX 77802, USA

Email: suely@cs.tamu.edu

April 23, 1998

Mathematics Subject Classification: Primary: 65F15; Secondary: 81-08

Key words and phrases: Eigenvalues, Davidson method, parallel algorithm.

*This research is supported by NSF grant ASC-9528912 and a Texas A&M University Interdisciplinary Research Initiative Award.

Running title: **Parallel Algorithm for Several Eigenvalues**

Send proofs to:

Dr. Suely Oliveira

Department of Computer Science

Texas A&M University

College Station, TX 77802

USA

Email: suely@cs.tamu.edu

Fax: (409)847-8578

A Parallel Davidson-Type Algorithm for Several Eigenvalues

Abstract

In this paper we propose a new parallelization of the Davidson algorithm adapted for many eigenvalues. In our parallelization we use a relationship between two consecutive subspaces which allows us to calculate eigenvalues in the subspace through an arrowhead matrix. Theoretical timing estimates for the parallel algorithm are developed and compared against our numerical results on the Paragon. Finally our algorithm is compared against another recent parallel algorithm for multiple eigenvalues, but based on Arnoldi: PARPACK.

1 Introduction

Many iterative methods for eigenvalue problems proceed in a similar way to obtain a solution for the eigenproblem $Au = \lambda u$. They construct an orthonormal basis V and approximate the exact eigenvector u by a vector y in the subspace spanned by V [33]. In other words the original problem is projected onto the subspace which reduces the problem to a smaller eigenproblem $S_k y = \tilde{\lambda} y$, where $S_k = V^T A V$. Then the eigenpair $(\tilde{\lambda}, y)$ can be obtained with much less computational effort. The approximated eigenvector y is mapped back to the eigenvector of the original problem as $u = Vy$.

The Arnoldi algorithm starts from a given vector $v_1 = x_1/||x_1||$ and successively generates orthogonal bases $Q_k$for Krylov subspaces $K_k(A, v_1) = span\{\, v_1, Av_1, \ldots, A^{k-1}v_1 \,\}$. For a general matrix the projected matrix S_k is a Hessenberg matrix. For a symmetric matrix the projected matrix is a tridiagonal matrix and the Arnoldi algorithm is equivalent to the Lanczos algorithm [17]. On the other hand, Davidson-type methods build an orthonormal basis $\{ v_1, v_2, \ldots, v_k \}$ depending on the initial vector x_1, the matrix A, and a preconditioner M_λ. Specifically, the preconditioner M_λ is applied to the current residual, $r_k = Au_k - \tilde{\lambda}_k u_k$, and the preconditioned residual $t_k = M_\lambda r_k$ is orthonormalized against the previous columns of $V_k = [v_1, v_2, \ldots, v_k]$. The new orthonormal basis $V_{k+1} = [V_k, v_{k+1}]$ defines the new projected matrix $S_{k+1} = V_{k+1}^T A V_{k+1}$ and the process is repeated iteratively. In this paper we present a parallel algorithm for Davidson type methods.

Recently, the Generalized Davidson (GD) algorithm has been modified to calculate several eigenvalues [30]. We use restarts as suggested in [10, 27] since it keeps the size of the subspace small, while preserving important information from previous iterations. This means that the cost of an iteration is kept small, hopefully without significantly increasing the number of iterations, and making the overall algorithm more efficient.

Some previous implementations for the Davidson algorithm solve the eigenvalue problem in subspace S_k by using algorithms for dense matrices: early works [8, 39] adopt EISPACK [34] routines or faster implementations of Householder QR [9]. Later implementations [36, 37] use LAPACK [1].

Partial parallelization is obtained by addressing the matrix-vector operations and sparse format storage for matrix A [36, 35, 37]. The parallelization of [35] aims the reduction of synchronization needed by inner products and has fine granularity. In this paper we present a relationship between two successive subspaces (S_{k-1} and S_k) which allows us to calculate the eigenvalues in the subspace S_k through an arrowhead matrix representation. The arrowhead structure is extremely sparse and the associated eigenvalue problem can be solved in a highly parallel way . We address the data storage for distributed memory architectures. Matrices are partitioned along distinct processors so that the final distribution is well balanced and most of the computational work can be performed in place.

In this paper, we present and compare the Davidson for Several Eigenvalues (DSE) algorithm in two different parallel architectures: the nCUBE and the Paragon. The words **processor** and **node** are used interchangeably. To achieve portability we used MPI, and level-2 and level-3 BLAS whenever possible, in our implementations. Theoretical timing estimates for the new parallel algorithms are developed and compared against our numerical results. Much of the parallel techniques can be used for parallelization of GD in addition of the parallelization of the new variant: DSE.

The remainder of the paper is as follows. In Section 2 we present the DSE algorithm. In Section 3 we design parallel steps for DSE. In Section 4 we develop timings bounds for the parallel algorithm. In Section 5 we show numerical results and finally in Section 6 we present our conclusions.

2 Davidson for Several Eigenvalues

Although in the original formulation M_λ is the diagonal matrix $(diag(A) - \lambda I)^{-1})$[10], GD algorithms adopt different operators for M_λ. For example in [30], a multigrid preconditioner was used. The Davidson for Several Eigenvalues (DSE) algorithm is a new version of GD, adapted for calculating several eigenvalues [30], and implemented in a restarted manner along the lines of [7]. A nice review of iterative methods for finding a few eigenvalues of large matrices is given in [11]. DSE computes the first p eigenpairs of A in order, and uses implicit restarts to limit the increasing computational work from successive larger

V_k and S_k subspaces. The DSE algorithm is given as

Algorithm 1 – **Restarted Davidson for Several Eigenvalues (DSE)**

Given a symmetric matrix A, an initial vector x_1, number of eigenpairs p, restart index q, minimal dimension m for the projected matrix S ($m > p$), and convergence tolerance ϵ, compute approximations λ and u for the p smallest eigenpairs of A.

1. Set $v_1 \leftarrow x_1/\|x_1\|_2$. (initial guess)
2. $V_1 \leftarrow [v_1]$.
3. $S_0 \leftarrow [\ ]$; $dim\ S \leftarrow 0$; $W_0 \leftarrow [\ ]$; $k = 1$;
4. For $j = 1, \ldots, p$ do (approximation for j-th eigenpair)

   While $k = 1$ or $\|r_{k-1}\| < \epsilon$ do

   (a) $w_k \leftarrow Av_k$

   (b) $W_k \leftarrow [W_{k-1}, w_k]$

   (c) Compute $V_k^T w_k$ and make it the last column and row of $S_k = V_k^T A V_k$. Update $dim\ S \leftarrow dim\ S + 1$.

   (d) If $(m + q) \leq dim\ S$ (restart S_k)

   Compute $S_k = Y_k \Lambda_k Y_k^T$, the complete symmetric eigendecomposition of S_k, where Λ_k is a diagonal matrix with its entries ordered in increasing order.

   Reduce $S_k \leftarrow (\Lambda_k)_{(m \times m)}$ to its m smaller eigenvectors, and update V_k and W_k . Update $dim\ S \leftarrow m$.

   (e) Compute the l^{th} smallest eigenpair λ_k, y_k of S_k, where $l = \min(dim\ S, j)$.

   (f) $u_k \leftarrow V_k y_k$.

(g) $r_k \leftarrow Au_k - \lambda_k u_k = W_k y_k - \lambda_k V_k y_k$. (residual)

(h) If $\|r_k\|_2 < \epsilon$ then exit inner loop.

(i) $t_k \leftarrow Mr_k$. (preconditioning)

(j) $v_{k+1} \leftarrow \text{mgs}(V_k, t_k)$. (apply modified Gram Schmidt)

(k) $V_{k+1} \leftarrow [V_k, v_{k+1}]$.

End while

End do

5. Stop.

In the original Davidson algorithm the initial guess is a set of unit vectors corresponding to p diagonal elements (which is a good choice for finding the p biggest or smallest eigenvalues of a diagonally dominant matrix). In our new version (DSE), we were not necessarily treating diagonally dominant matrices. This makes the choice of the initial guesses for the eigenvectors harder. For simplicity our initial guess is a vector which is generated randomly. The best random vector choice has independent components taken from a normal distribution. In contrast, the original Davidson algorithm is deterministic but more susceptible to hidden symmetries in the matrices. Another aspect of our algorithm is that, when coupled with a multigrid or ADI preconditioner, it is often able to identify the multiplicity of multiple eigenvalues. More numerical results about the behavior of the DSE algorithm are shown in [4, 30].

In the next section we detail the parallel steps of the DSE algorithm: data distribution, parallel orthonormalization, parallel calculation of eigenvalues on subspace S_k, and parallel implicit restarting; these parallel steps appearing in any variation of GD. In addition to the Diagonal precondioner we will present numerical results with Multigrid and ADI preconditioners. In this paper we do not intend to address parallelization issues of the multilevel or ADI preconditioners: For multigrid preconditioners several implementations are available in the literature [3, 16, 25]. The same is true about the parallelization of the ADI preconditioner [13, 21, 24].

3 Parallel steps of DSE

A parallel implementation for the restarted Davidson for several eigenvalues (DSE) should exploit the data parallelism inherent in the algorithm. A message-passing computer allows good scalability. If different workloads must be managed, as happens in the arrowhead matrix eigenvalue calculation (step **4.e**), a particular processor can be selected to coordinate the task.

3.1 Data representation and distribution

Data storage is an important aspect when parallelizing the algorithm described above. Matrices are partitioned along distinct processors so that the program exploits all the best possible data parallelism. At each iteration, a new vector v_k is added to the basis $\{v_0, \ldots, v_{k-1}\}$. This is reflected by the addition of a new column to matrices V_k and W_k (steps **4.b** and **4.k**). This makes row-wise storage the best partitioning scheme, since those matrices always have a constant number of rows and the use of restarts keeps a bound on the number of columns; this way data distribution is uniform among the processors for these matrices. Moreover, computations such as matrix-vector multiplies and modified Gram Schmidt orthogonalization can be performed in place. In the following description a subscript stands for the iteration number in DSE (algorithm 1), and superscripts indicate which processor allocates the data. Let A

be a matrix of order n and N be the number of processors available. Matrix A can be split into row blocks A^i, $i = 1, \ldots, N$, each one containing $\leq \lceil n/N \rceil$ rows of A. For simplicity we will assume that n is a multiple of N:

$$A = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^N \end{bmatrix} .$$

Thus processor i, $i = 1, \ldots, N$ stores A^i, the i^{th} row block of A. Matrices V_k and W_k are stored in the same fashion

$$V_k = \begin{bmatrix} V_k^1 \\ \vdots \\ V_k^N \end{bmatrix} \text{ and } W_k = \begin{bmatrix} W_k^1 \\ \vdots \\ W_k^N \end{bmatrix} .$$

Figure 1 illustrates the data allocation. This implies that $V_1 = v_1$ must be initialized in the same way; that is, the first n/N rows of v_1 (named v_1^1) are stored in processor 1, the i^{th} row-block of v_1 (v_1^i) is stored in processor i, and so on. In the following discussion we present the parallelization steps of the DSE algorithm when adopting the just described data structure, in a MIMD architecture.

Even though at first glance we may think $w = Av$ cannot be computed in place, this can be overcome by sending vector v_k to all processors so that each one computes its respective slice of w_k, namely $w_k^i = A^i v_k$. Thus matrix W_k^i can be updated in processor i as $W_k^i = [W_k^i, w_k^i]$.

Residual computations (steps **4.f, 4.g** and **4.h**) are also performed in place: $r_k^i = W_k^i y_k - \lambda V_k^i y_k$. Specifically, vector y_k is broadcast to all processors, each processor computes its slice of r_k and obtains the partial value $||r_k^i||_2^2$. The

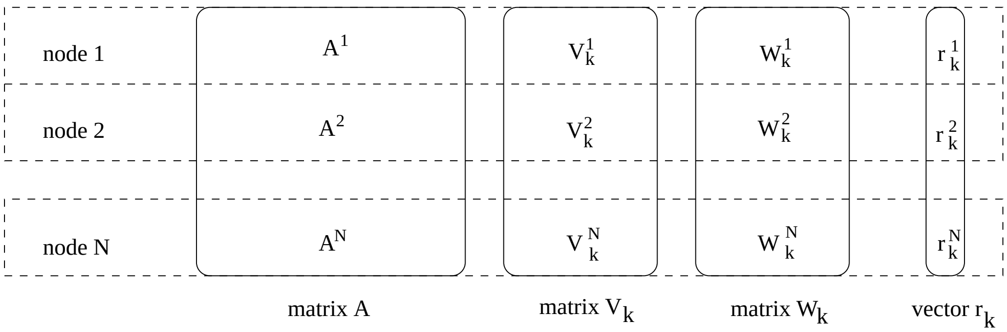


Figure 1: Matrices are distributed row-wise through the processors.

residual vector r_k can be evaluated by concatenating the slices

$$r_k = \begin{bmatrix} r_k^1 \\ \vdots \\ r_k^N \end{bmatrix}$$

and the respective norm is given by $||r_k||_2 = (\sum_{i=1}^{N} ||r_k^i||_2^2)^{1/2}$.

3.2 Parallel orthonormalization (steps 4.j and 4.k)

Steps **4.j** and **4.k** correspond to applying the column version [1] of modified Gram-Schmidt (*mgs*) to the extended matrix $[V_k, t_k]$. The data distribution implies that the *mgs* algorithm will be implemented with a row-wise partitioning scheme similar to the one of [29]. In the mgs algorithm we do not have significant loss of orthogonality; thus only the new column t_k needs to be modified since the current basis V_k has been previously obtained by mgs algorithm in previous iteration steps. Summarizing, t_k is orthonormalized in relation to the current

[1] The reader should not confuse the implementation of mgs (in this case column version and is determined by the ordering of loops in the mgs algorithm) with the storage scheme chosen: row-wise.

basis V_k by

For $j = 1$ to $dimV_k$

$$
\begin{aligned}
c_j &\leftarrow \langle v_j, t_k \rangle \\
t_k &\leftarrow t_k - c_j v_j
\end{aligned}
\tag{1}
$$

$$v_{k+1} \leftarrow t_k / ||t_k||$$

Figure 2 represents data distribution among N processors for steps **4.j** and **4.k**.. Each processor i computes the partial inner products $c_j^i = \langle v_j^i, t_k^i \rangle$, $j = 1, \ldots, dimV_k$, where v_j^i and t_k^i represents the rows of v_j and t_k stored in processor i, respectively. The partial results c_j^i are added in a *host* processor to evaluate the coefficient $c_j = \sum_{i=1}^{N} c_j^i$ which is then broadcasted and used to perform the related projection on each processor $t_k^i \leftarrow t_k^i - c_j v_j^i$, $i = 1, \ldots, N$. The new

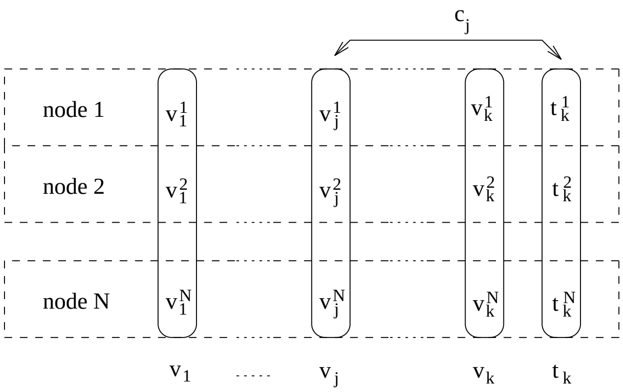


Figure 2: Data distribution for the modified Gram-Schmidt algorithm.

column v_{k+1} is obtained by the normalization of t_k, $v_{k+1}^i = t_k^i / ||t_k||$.

The orthonormalization process is accomplished using $\mathcal{O}(\bar{k})$ broadcast-type communications where $\bar{k} = dim\ S_k$, the number of columns for V at iteration k, which depends on the restart indexes q and m in the DSE algorithm. Notice that a column-wise storage might have a smaller upper bound included for its communication time: groups of consecutive columns v_j would be stored in the same processor and the new column t_k broadcasted to all processors so that the coefficients c_j in the code loop (1) would be performed locally. Thus, the resulting communication time should be $\mathcal{O}(N)$ point-to-point communications. However, those coefficients must be evaluated *in order* to guarantee the stability provided by the modified Gram-Schmidt algorithm. So a column-wise implementation would result in a sequential algorithm: only one processor at time would be calculating c_j. The parallel implementation of the *mgs* presented here differs from the pipelined algorithm of [29], partially because *mgs* does not implement the outer loop of the full version of the modified Gram-Schmidt algorithm. The Davidson algorithm itself implicitly supplies the outer loop of modified Gram-Schmidt since each iteration creates a new t_k which is orthogonalized against the previous basis vectors.

3.3 Parallel calculation of eigenvalues of S_k (step 4.e)

Since the complete eigendecomposition of $S_k = V_k^T A V_k$ must be found for every k, the solution method used for the eigenproblem $S_k y_k = \lambda_k y_k$ is important. Although the eigenvalues calculation in the subspace is $O(k^3)$ and the MGS step cost is $O(kn)$, if $O(k^2)$ approaches n, the eigenvalue calculation of S_k is a significant step of the Davidson algorithm. Restarts can keep k small and also avoid this problem. It should be noted however that the constant in $O(k^3)$ may be large. The relationship between S_k and S_{k-1} [30] provides an alternative way to parallelize this step. Let $S_{k-1} = Y_{k-1}\Lambda_{k-1}Y_{k-1}^T$ be the orthonormal decomposition of the symmetric matrix S_{k-1}, Notice that

$$S_k = [V_{k-1}, v_k]^T \, A \, [V_{k-1}, v_k] = \begin{bmatrix} S_{k-1} & s_k \\ s_k^T & s_{kk} \end{bmatrix} \tag{2}$$

where $s_{kk} = v_k^T w_k$ and $s_k = V_{k-1}^T A v_k = V_{k-1}^T w_k$. Let

$$\tilde{S}_k = \begin{bmatrix} Y_{k-1}^T & 0 \\ 0 & 1 \end{bmatrix} S_k \begin{bmatrix} Y_{k-1} & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \Lambda_{k-1} & Y_{k-1}^T s_k \\ s_k^T Y_{k-1} & s_{kk} \end{bmatrix} . \tag{3}$$

We can see that matrix $\tilde{S}_k$ is similar to matrix S_k and is in the form of an arrowhead matrix

$$\tilde{S}_k = \begin{bmatrix} \Lambda_{k-1} & \tilde{s}_k \\ \tilde{s}_k^T & s_{kk} \end{bmatrix} ,$$

where $\tilde{s}_k$ is the column vector $Y_{k-1}^T s_k$. Thus, the eigenvalues of S_k can be obtained by evaluating the eigenvalues of the arrowhead matrix $\tilde{S}_k$. Given the orthogonal eigendecomposition of $\tilde{S}_k$

$$\tilde{S}_k = Q_k \Lambda_k Q_k^T \tag{4}$$

identity (3) is equivalent to

$$S_k = \begin{bmatrix} Y_{k-1} & 0 \\ 0 & 1 \end{bmatrix} \tilde{S}_k \begin{bmatrix} Y_{k-1}^T & 0 \\ 0 & 1 \end{bmatrix} .$$

Since S_k is symmetric it also has an orthonormal decomposition $S_k = Y_k \Lambda_k Y_k^T$, the previous equation establishes the recursive relation

$$Y_k = \begin{bmatrix} Y_{k-1} & 0 \\ 0 & 1 \end{bmatrix} Q_k \ , \tag{5}$$

for the eigenvectors of S_k. Given the eigenvalues Λ_{k-1} and eigenvectors Y_{k-1} of S_{k-1}, after using (3) for construction of $\tilde{S}_k$, decomposition (4) can be used to find the eigenvalues Λ_k and respective eigenvectors Y_k of S_k by (5). Oliveira improved the parallelization of eigenvalue algorithms for arrowhead matrices [31]. Here we choose to use the O'Leary and Stewart algorithm presented in [28]. Their algorithm is for symmetric arrowhead matrices and can be applied to find Λ_k and Q_k with $\mathcal{O}(k^2)$ computational effort. Consider the following representation for $\tilde{S}_k$

$$\tilde{S}_k = \begin{pmatrix} d_1 & 0 & 0 & \cdots & 0 & \tilde{s}_k(1) \\ 0 & d_2 & 0 & \cdots & 0 & \tilde{s}_k(2) \\ 0 & 0 & d_3 & \cdots & 0 & \tilde{s}_k(3) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & d_{\bar{k}-1} & \tilde{s}_k(\bar{k}-1) \\ \tilde{s}_k(1) & \tilde{s}_k(2) & \tilde{s}_k(3) & \cdots & \tilde{s}_k(\bar{k}-1) & s_{kk} \end{pmatrix} \tag{6}$$

where $\bar{k}$ stands for the order of $\tilde{S}_k$ and d_i are the eigenvalues λ_i of Λ_{k-1}, or, eigenvalues of S_{k-1}. Assume all reducible[2] eigenvalues d_i have already been removed from $\tilde{S}_k$.

The interlacing theorem [32] (or bracketing theorem for quantum chemists) is the basis for the method. This theorem has roots which go back to [2] and more recently has become known as Cauchy's interlacing theorem; it states that if Λ_{k-1} has its entries ordered in increasing order $d_1 \leq d_2 \leq \ldots \leq d_{\bar{k}-1}$; then the sequence of diagonal elements in (6) define a set of intervals $\{(d_{j-1}, d_j) : j = 1, \ldots, \bar{k}\}$ each one containing one and only one eigenvalue of $\tilde{S}_k$, unless $d_{j-1} = d_j$. Numerically, d_{j-1} and d_j are considered to be equal if their relative difference is less than machine epsilon (2×10^{-16} for IEEE double precision).

[2]For a reducible eigenvalue d_i we have $\tilde{s}_k(i) = 0$. In other words, the corresponding eigenvector is the i^{th} unit vector.

Bounds d_0 and $d_{\bar{k}}$ are derived from Gershgorin's localization theorem [22]:

$$d_0 = \min\{ \ d_1 - |\tilde{s}_k(1)|, \dots, \ d_{\bar{k}-1} - |\tilde{s}_k(\bar{k}-1)|, \ s_{kk} - \textstyle\sum_{i=1}^{\bar{k}-1} |\tilde{s}_k(i)| \ \},$$
$$d_{\bar{k}} = \min\{ \ d_1 + |\tilde{s}_k(1)|, \dots, \ d_{\bar{k}-1} + |\tilde{s}_k(\bar{k}-1)|, \ s_{kk} + \textstyle\sum_{i=1}^{\bar{k}-1} |\tilde{s}_k(i)| \ \} \ .$$

If $d_{j-1} < d_j$ then the associated eigenvalue λ_j satisfies $d_{j-1} < \lambda_j < d_j$ and can be found as a root of $det(\tilde{S}_k - \lambda I) = 0$. Since $d_j - \lambda \neq 0$, $j = 0, \dots, \bar{k}$, for any eigenvalue λ of $\tilde{S}_k$, the j-th entry in the last row of $\tilde{S}_k$ can be eliminated by using the diagonal values $d_j - \lambda$ as pivot elements. That is, Gaussian elimination is applied on the last row of $\tilde{S}_k$. The resulting matrix is

$$\begin{pmatrix} d_1 - \lambda & 0 & \cdots & \tilde{s}_k(1) \\ 0 & d_2 - \lambda & \cdots & \tilde{s}_k(2) \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \varphi(\lambda) \end{pmatrix} \tag{7}$$

where

$$\varphi(\lambda) = s_{kk} - \lambda - \sum_{l=1}^{\bar{k}-1} \frac{[\tilde{s}_k(l)]^2}{d_l - \lambda} \ . \tag{8}$$

Consequently, condition $det(\tilde{S}_k - \lambda I) = 0$ is equivalent to have the determinant of (7) equal to zero, and λ is an eigenvalue of $\tilde{S}_k$ iff $\varphi(\lambda) = 0$. A root-finding method can be applied to evaluate λ_j and the associated eigenvector $q_j = (q_j(1), \cdots, q_j(k))^T$, where

$$q_j(l) = \begin{cases} \dfrac{\tilde{s}_k(l)}{\lambda_j - d_l}, & l = 1, \dots, \bar{k} - 1 \\ 1, & l = \bar{k} \end{cases} \ . \tag{9}$$

Notice that if a diagonal entry d_l is already close to an actual eigenvalue λ, then the norm of the associated term $\tilde{s}_k(l)$ may be small so that some terms in equation (9) may generate significant roundoff errors. We address this problem in the appendix of this paper.

Repeated values $d_{j-1} < d_j = \cdots = d_{j+\mu} < d_{j+\mu+1}$ indicate multiplicity μ for the eigenvalue d_{j+1}. In this case, the μ associated eigenvectors $q_{j+i},\ i = 1, \ldots, \mu$ are obtained by

$$q_{j+i}(l) = \begin{cases} 0, & l = 1, \ldots, j-1 \\ \dfrac{\tilde{s}_k(l)\, \tilde{s}_k(j+i)}{\tilde{s}_k(j)^2 + \cdots + \tilde{s}_k(j+i-1)^2}, & l = j,\ \ldots, j+i-1 \\ -1, & l = j+i \\ 0, & l = j+i+1, \ldots, \bar{k} \end{cases} . \tag{10}$$

Eigenvalues are grouped by increasing order so that Λ_k will define a new ordered set of intervals $\{(d_{j-1}, d_j)\}$ in the next iteration. Moreover, the l^{th} eigenpair (λ_k, y_k) of S_k can be retrieved from Λ_k and Y_k, as required in the DSE algorithm.

The arrowhead algorithm of [29] for decomposition $\tilde{S}_k = Q_k \Lambda_k Q_k^T$ is highly parallelizable. Distinct eigenvalues can be processed simultaneously. In this case, a unit of work consists of computing the eigenvalue λ_j and related eigenvector(s) q_{j+i}, $i = 1, \ldots, \mu$ residing in the given interval (d_{j-1}, d_j). Notice that different intervals may generate distinct amounts of work due to different steps for calculation of roots of $\varphi(\lambda)$, and if $d_{j-2} < d_{j-1} = d_j < d_{j+1}$, $\lambda_j = d_j$ and the root-finding algorithm does not even need to be applied. Thus, it is difficult to predict the amount of work performed by each processor. An efficient synchronization among these concurrent computations must resolve the scheduling problem [19]. A particular processor is selected as a *host* to coordinate the work. It removes the set $I = \{i : d_i \text{ is reducible}\}$ of reducible eigenpairs from matrix $\tilde{S}_k$ and begins by sending one or a group of intervals $\{(d_{j-1}, d_j) : j \in J\}$ from the irreducible set $J = \{1, \ldots, \bar{k} = dim S_k\} - I$ to the processors. Results are sent back to the host, indicating that the respective processor is ready to receive another group of intervals. Simultaneously, the host groups all eigenvalues λ_i,

$i \in I$, and λ_j, $j \in J$, in the vector Λ_k and respective eigenvectors q_i and q_j as columns of Q_k.

Recall from equation (2) that S_k is obtained by adding $s_k = V_{k-1}^T w_k$ as a new row and column for S_{k-1}. Moreover, each processor computes locally its respective slice of w_k as $w_k^i = A^i v_k$. This data distribution allows the host to obtain s_k as a sum of partial terms $\sigma_k^i = (V_k^i)^T w_k^i$ which can be computed simultaneously. In other words, $s_k = \sum_{i=1}^N \sigma_k^i$ is given by

$$s_k = \left[\sum_{i=1}^N \sigma_k^i \right] = \left[(V_k^1)^T \,|\, \cdots \,|\, (V_k^i)^T \,|\, \cdots \,|\, (V_k^N)^T \right] \begin{bmatrix} w_k^1 \\ \vdots \\ w_k^i \\ \vdots \\ w_k^N \end{bmatrix}$$

In practice, the matrix S_k will not be stored: only a vector for Λ_k and a matrix for Y_k are required from one iteration to the next.

The recursive relation (5) for calculation of eigenvectors is based on the updating of Y_k at each iteration and can also be performed in parallel. We rewrite the eigenvectors calculated from decomposition $\tilde{S}_k = Q_k \Lambda_k Q_k^T$ as

$$Q_k = \begin{bmatrix} \tilde{Q}_k \\ q_k^T \end{bmatrix} \tag{11}$$

where q_k^T is the last row of Q_k. Under this notation, the new set of eigenvectors Y_k can be obtained by computing $\tilde{Y}_k = Y_{k-1} \tilde{Q}_k$ (which adds one more column into Y_{k-1}), and by appending the new row q_k^T into matrix $\tilde{Y}_k$. At first look, this update could be performed at the host after gathering the individual eigenvectors (columns of Q_k) from the processors. Further insight reveals that the update (5) can be performed in a distributed way, provided that each processor contains a copy of Y_{k-1}: as soon as a given processor obtains a new eigenvector $\vec{q}$ of $\tilde{S}_k$ (i.e., a column of Q_k), it uses the splitting $\vec{q} = [\tilde{q}^T, q]^T$ similarly to (11), so that the corresponding column of Y_k is given by $y = [(Y_{k-1}\tilde{q})^T, q]^T$. After that, the processors send the new columns of Y_k back to the host, instead of a column of Q_k.

Notice that each processor contains a copy of matrix Y_k. It may look redundant, however this data structure allows the parallelization for update (5) as described above. Moreover, restarting imposes an upper-bound for the size of matrix Y_k independent of the number of iterations. It guarantees a modest storage requirement for Y_k. In the remaining sections of this paper, we will be addressing the implementations of DSE which maintains a copy of Y_k on each processor. Combining the above considerations we obtain

Algorithm 2 – Parallel Arrowhead-Decomposition

Given the previous set of eigenvalues Λ_{k-1} and the new row s_k for S_k, compute the new set of eigenpairs Λ_k and Y_k.

1. $\Lambda_k \leftarrow [\,]$; $Q_k \leftarrow [\,]$.
2. Compute $\tilde{s}_k \leftarrow (Y_k)^T s_k$.
3. $d \leftarrow \Lambda_{k-1}$.
4. Remove all reducible eigenvalues on d and compute multiplicity for the remaining intervals (d_{j-1}, d_j) on d.
5. Broadcast d, $\tilde{s}_k$ and s_{kk}.
6. Dispatch a interval (or a group) to each node.
7. host:

   While there are eigenpairs to be received back

Receive a eigenpair (or a group) from any node i.

If there is a interval (or group) to be sent

Send it to node i.

else

Node i is done.

Add the new eigenpair (or group) to Λ_k and Y_k.

other nodes:

While node is not done

Receive a interval (or group) from host.

Compute the eigenpair (or group).

Compute update (5) for the new eigenvector(s).

Send the associated eigenpair (or group) to the host.

8. Broadcast Y_k.
9. Stop.

If no restarting is applied or if a large number of eigenvalues is desired, the dimension of Y_k will increase considerably. Then, storage may turn out to be a dominant issue. To overcome this problem, we would scatter Y_k throughout the processors. Matrix Y_k can be partitioned by rows, as were V_k and W_k, but in a wraparound manner: since it is a square matrix that increases after each iteration, a row-wise cyclic striping may be applied to provide a well balanced distribution of data between processors. Thus, the 1×1 matrix Y_1 will be stored in processor 1; in the next iteration, it will be updated as a 2×2 matrix Y_2 which will be stored in processors 1 and 2, and so on. Figure 3 shows a general situation where Y_k^i denotes the rows of Y_k stored in processor i. Under this storage scheme, $\tilde{s}_k = Y_{k-1}^T s_k$ is obtained by sending the respective rows of s_k

to their related processors. That is, since processor i contains rows $i, i+N, i+2N, \ldots, i+N\lfloor \bar{k}/N \rfloor - N$ of Y_{k-1}, it receives the same subset s_k^i of rows from s_k, and perform

$$\tilde{\sigma}_k^i = (Y_{k-1}^i)^T s_k^i \ .$$

This information is sent to the host which computes $\tilde{s}_k$ as $\tilde{s}_k = \sum_{i=1}^N \tilde{\sigma}_k^i$, and proceeds with the eigendecomposition for the arrowhead matrix of shape shown in (6).

For this case, the host splits matrix Q_k as in (11). Each component Y_{k-1}^i is update by sending $\tilde{Q}_k$ to processor i which performs

$$Y_k^i = Y_{k-1}^i \tilde{Q}_k \ .$$

This multiplication adds one more column to Y_k^i. Finally, the host sends row q_k^T to the next unbalanced processor in the cyclic process (processor i, in the example shown in Figure 3). It is important to notice that this choice contributes for storage-savings but it increases communication overhead.

3.4 Parallel implicit restart (step 4.d)

After each iteration, the matrix S_k increases its dimension by one row and column, causing the computation of eigenvalues in the subspace to increase its cost. An implicit restart scheme is used to periodically reduce the dimension of S_k. Recall the orthogonal decomposition of S_k:

$$\Lambda_k = Y_k^T S_k Y_k. \tag{12}$$

Let q be a fixed restart index, and m the minimal dimension for the projected matrix S_k. Since $m > p$, the desired eigenvalue approximations are contained on the first m entries of Λ_k. If dimension of S_k exceeds the value $m+q$, one

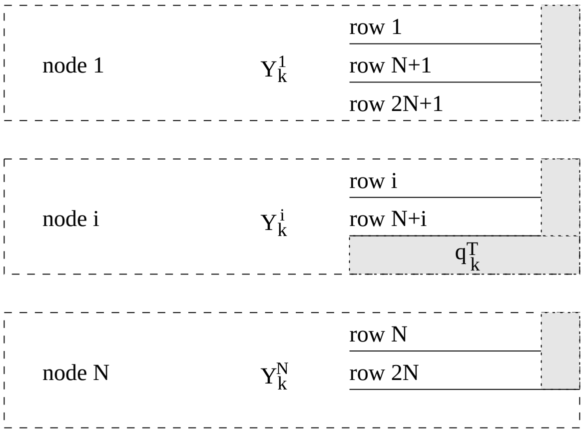


Figure 3: Rows of Y_k are distributed in a cyclic order through the processors. The shadowed area indicates the new entries for Y_k^i, $i = 1 \ldots N$, and the new row q_k^T in the wraparound process.

can drop the q largest eigenvalues. This can be done by splitting

$$\Lambda_k = \begin{bmatrix} \Lambda_{k,1} & 0 \\ 0 & \Lambda_{k,2} \end{bmatrix}$$

and $Y_k = [Y_{k,1}, Y_{k,2}]$ where $\Lambda_{k,1}$ and $Y_{k,1}$ correspond to the m smallest eigenpairs. (Remember that Λ_k and Y_k are already sorted, as specified in the previous paragraph.) Projecting S_k over the space spanned by its m smallest eigenvectors, we obtain the $m \times m$ reduced matrix

$$S_k^{(+)} = Y_{k,1}^T S_k Y_{k,1} = \Lambda_{k,1} \ . \tag{13}$$

Thus, a particular computation occurs during a restarted iteration. Since relation $S_k = V_k^T A V_k$ must be preserved for $S_k^{(+)}$, decomposition (12) imposes

$$\Lambda_{k,1} = (V_k Y_{k,1})^T A V_k Y_{k,1} \ ,$$

which corresponds to update $V_k^{(+)} = V_k Y_{k,1}$ and consequently $W_k^{(+)} = W_k Y_{k,1}$.

Also, equation (13) claims that the new eigenvectors of $S_k^{(+)}$ are canonical vectors. It reduces relation (5) to $Y_k = Q_k$ when performed right after a restarted iteration. It is important to note that if $Y_{k,1}$ is not orthonormal the update $V_k^{(+)} = V_k Y_{k,1}$ may compromise the condition that $V_k^{(+)}$ must be an orthonormal basis. Loss of orthogonality may happen due to roundoff errors in equations (9) and (10). To avoid such problem the modified Gram-Schmidt method is applied to $Y_{k,1}$ before computing $V_k^{(+)}$ and $W_k^{(+)}$.

A restarted iteration requires one more group of updates: $V_k \leftarrow V_k Y_{k,1}$ and $W_k \leftarrow W_k Y_{k,1}$. Thus, each processor i may discard the last q columns of Y_k because the number of columns is reduced from $m+q$ to m, and compute the updates $V_k^i \leftarrow V_k^i Y_{k,1}$ and $W_k^i \leftarrow W_k^i Y_{k,1}$.

Algorithm 3 – Parallel Implicit Restart

Given the set of eigenvalues Λ_k and eigenvectors Y_k and the associated matrices V_k and W_k, check for restart.

1. If $(m+q) \leq dim\ S$ (dimension of Λ_k)

   $\Lambda_k \leftarrow (\Lambda_k)_{(m\times m)}$.

   Orthonormalize $Y_{k,1}$ = first m columns of Y_k.

   For all nodes i, $i \in \{1,\ldots,N\}$

   $V_k^i \leftarrow V_k^i Y_{k,1}$.

   $W_k^i \leftarrow W_k^i Y_{k,1}$.

   $dim\ S \leftarrow m$.

2. Stop.

4 Timing analysis for the parallel algorithm

All the pieces presented above can be integrated to form the complete parallel version for the DSE algorithm (Algorithm 4 shown below). Next estimates for computational and communication timings are presented.

Algorithm 4 – Parallel Restarted Davidson for Several Eigenvalues

Given a matrix A, an initial vector x_1, iteration limit η, number of eigenpairs p, restart indexes q and m, and convergence tolerance ϵ, compute approximations λ and u for the p smallest eigenpairs of A. N indicates the number of processors being used.

1. Set $v_1 \leftarrow x_1/\|x_1\|_2$. (initial guess)
2. For all nodes i, $i \in \{1, \ldots, N\}$

   $V_1^i \leftarrow [v_1^i]$; $W_0^i \leftarrow [\,]$; $Y_0 \leftarrow [\,]$.
3. $\Lambda_0 \leftarrow [\,]$; $dim\ S \leftarrow 0$.
4. For $j = 1, \ldots, p$ do (approximation for j-th eigenpair)

   For $k = 1, \ldots, \eta$ do

   (a) Broadcast v_k.

   (b) For all nodes i, $i \in \{1, \ldots, N\}$

      Receive v_k.

      Compute $w_k^i = A^i v_k$ and update $W_k^i = [W_{k-1}^i, w_k^i]$.

      Compute $s_k^i = (V_{k-1}^i)^T w_k^i$ and send to the host.

   (c) Compute $s_k = \sum_{i=1}^{N} s_k^i$ in the host and use algorithm 2 to obtain the eigendecomposition Λ_k and Y_k. Update $dim\ S \leftarrow dim\ S{+}1$.

   (d) Use algorithm 3 to check and apply restart when it is necessary.

(e) Broadcast the l^{th} smallest eigenpair λ_k, y_k from Λ_k and Y_k, where $l = \min(dim\ S, j)$.

(f) For all nodes i, $i \in \{1, \ldots, N\}$

Receive y_k, and λ_k.

Compute $r_k^i = W_k^i y_k - \lambda V_k^i y_k$.

(g) Broadcast r_k by appending all r_k^i. (residual vector)

(h) If $\|r_k\|_2 < \epsilon$ then exit inner loop.

(i) $t_k \leftarrow M r_k$. (preconditioning)

(j) Send slice t_k^i to node i, $i \in \{1, \ldots, N\}$.

(k) $v_{k+1} \leftarrow \text{pmgs}(V_k, t_k)$. (parallel modified Gram Schmidt)

5. Stop.

The DSE algorithm performs matrix multiplies at various times, e.g. the computation of $w_k = A\, v_k$ and $t_k = M\, r_k$; this is the most time consuming step of the algorithm. However, we do not include these operations in our theoretical operation count because the structure of matrix A changes for different problems. Nevertheless, in our numerical experiments the timings shown include these matrix vector multiplications. Also, in our implementations we resort to BLAS, which is an efficient matrix library and is optimized for each computer.

For the operation count, one iteration of the algorithm was divided into four major tasks: *1. residual calculation*; *2. arrowhead-eigendecomposition*; *3. implicit restart* (when applicable); and *4. modified Gram-Schmidt orthonormalization*. The residual calculation $r_k^i = W_k^i y_k - \lambda_k V_k^i y_k$ depends on the order k of V_k and W_k. Each r_k^i is obtained with $4kn/N$ flops. The arrowhead-eigendecomposition employs three sub-steps, namely, the matrix multiplication $s_k^i = (V_{k-1}^i)^T w_k^i$, eigenvalue-eigenvector computations, and updating $Y_k =$

$Y_{k-1}\tilde{Q}_k$. The first two sub-steps are accomplished in $2kn/N$ and ck^2 flops, respectively for some constant c. The last sub-step can be understood as k matrix-vector multiplications distributed across N processors, which leads to $2k^3/N$ flops. Implicit restart occurs sporadically and proceeds the updating $V_k^i = V_k^i Y_{k,1}$ and $W_k^i = W_k^i Y_{k,1}$ in $4kmn/N$ flops. Finally, the mgs orthonormalization requires $2kn/N$ flops. The resulting lower-bound for the computational time $T^{cp}(k)$ spent in one iteration, excluding the restart operations, is determined by adding all the above float point operations:

$$T^{cp}(k) = \frac{2}{N}k^3 + ck^2 + 8\frac{n}{N}k \ .$$

Such estimate reflects time dependencies on the order of the problem (n), the number of processors used (N), and the current dimension (k) of the subspace S_k. Given the total number of iterations (s) taken by the DSE algorithm to converge, the global computational time can be estimated, for both non-restarted and restarted cases. First consider the case where no restart (nr) was applied. In this case, the dimension k of the subspace successively increases and the resulting time $T_{nr}^{cp}(s)$ is a straightforward summation of the times for each iteration

$$\begin{aligned} T_{nr}^{cp}(s) &= \sum_{k=1}^{s} T^{cp}(k) = \frac{2}{N}\sum_{k=1}^{s} k^3 + c\sum_{k=1}^{s} k^2 + 8\frac{n}{N}\sum_{k=1}^{s} k \\ &= \frac{s^4}{2N} + \big(\frac{c}{3} + \frac{1}{N}\big)s^3 + \big(4\frac{n}{N} + \frac{c}{2} + \frac{1}{2N}\big)s^2 + \big(4\frac{n}{N} + \frac{c}{6}\big)s \ . \end{aligned}$$

Notice that updates $Y_k = Y_{k-1}\tilde{Q}_k$ produce the higher order term in s. It demonstrates the overall effect of growing matrices Y_k. The restart parameter q imposes the upper bound $(m+q)$ for the order of Y_k. To develop an estimate for the restarted case, we need to understand the behavior of dimension k. At the beginning, dimension k grows from 1 to $m+q$. After the first restart, k is reduced to m and begins to grow again, ranging from m to $m+q$, until the next restart is applied. So, the number of restarts, excluding the first, is an integer division by q. More precisely, the total number of restarts for s iterations is given by $1+\lfloor\frac{s-(m+q)}{q}\rfloor = \lfloor\frac{s-m}{q}\rfloor$. And the number of remaining iterations after the last restart is $\delta = (s-m-q)\ mod\ q$. Notice that the case with restarts can be written in terms of times without restarts, T_{nr}^{cp}, by adding the time spent before the first restart, the time from the first restart until the last restart, the remaining time after the last restart, and all the extra times due to re-starts. Thus the total computational time $T_r^{cp}(s)$ using restarts is given by

$$\begin{aligned}
T_r^{cp}(s) &= T_{nr}^{cp}(m+q) \ + \ \left(\lfloor\frac{s-m}{q}\rfloor - 1\right)\left(T_{nr}^{cp}(m+q) - T_{nr}^{cp}(m-1)\right) \\
&\quad + \left(T_{nr}^{cp}(m-1+\delta) - T_{nr}^{cp}(m-1)\right) \ + \ \lfloor\frac{s-m}{q}\rfloor \ \tau_r^{cp} \\
&= \lfloor\frac{s-m}{q}\rfloor \left(T_{nr}^{cp}(m+q) - T_{nr}^{cp}(m-1) + \tau_r^{cp}\right) \\
&\quad + T_{nr}^{cp}(m+\delta-1) \ , \qquad (14)
\end{aligned}$$

where τ_r^{cp} represents the extra computational work performed on each restart step. We can derive an upper-bound by noticing that $\delta < q$

$$T_r^{cp}(s) \ \leq \ \frac{s-m}{q}\left(T_{nr}^{cp}(m+q) - T_{nr}^{cp}(m-1) + \tau_r^{cp}\right) + \ T_{nr}^{cp}(m+q) \ .$$

A restarted iteration applies modified Gram-Schmidt to $Y_{k,1}$ in $(m+q)(2m^2+m)-\frac{1}{2}(m^2-m)$ flops, and performs the updates $V_k^i = V_k^i Y_{k,1}$ and $W_k^i = W_k^i Y_{k,1}$ in $4m\frac{n}{N}(m+q)$ flops. By recalling that the update $Y_k = Y_{k-1}\tilde{Q}_k$ in (5) does not occur after a restarted iteration, we obtain

$$\begin{aligned}
\tau_r^{cp} &= 4m\frac{n}{N}(m+q) + (m+q)(2m^2+m) - \frac{1}{2}(m^2-m) - 2\frac{m^3}{N} \\
&= 2(m+q)(4m\frac{n}{N} + 2m^2 + m) - 2\frac{m^3}{N} - \frac{1}{2}(m^2-m) \ .
\end{aligned}$$

For the communication timing, consider t_s as the message startup time and t_w the per-word transfer time. We assume three distinct communication operations and their theoretical time estimates in a mesh architecture [23]. *One-to-one* communication is performed by sending a message from one processor to another; it takes time $t_s + \ell\, t_w$ for a message of size ℓ. This operation is presented in the arrowhead solver where the host sends specific messages to each processor. For an iteration step of DSE, it takes time $k(t_s + 2t_w)$ for the host to distribute the k intervals (d_{j-1}, d_j) across all processors and time $k(t_s + k\, t_w)$ to receive the k associated eigenvectors, each one of size k. *One-to-all* broadcast is characterized by a single processor sending the same message to all other processors. It takes time $(t_s + \ell\, t_w)\log N$ and is also related with the arrowhead solver: function $\varphi(\cdot)$ is transfered from host to all processors as a vector of size $2k$ which takes time $(t_s + 2k\, t_w)\log N$; Matrix Y_k has order k and costs $(t_s + k^2 t_w)\log N$ to be broadcasted. *All-to-all* broadcast, *reduction* and *data gathering* belong to the most expensive class of communication taking time $(t_s + \ell\, t_w)(N-1)$ for a message of size ℓ. These operations employ communications where all processors broadcast different data to be added (reduction operation) or grouped (gathering operation) into one single processor. In one iteration of the algorithm reduction is performed when computing $s_k = \sum_{i=1}^{N} s_k^i$ by using a reduction operation for N arrays of size k each; when computing the norm $||r_k||^2 = \sum_{i=1}^{N} ||r_k^i||^2$ by a reduction of N arrays of size 1 each; and when applying *mgs* which employs k reductions for N arrays of size 1 each for the inner products. The above operations are accomplished with times $(t_s + kt_w)(N-1)$, $(t_s + t_w)(N-1)$, and $k(t_s + t_w)(N-1)$, respectively. Data gathering is applied to concatenate the eigenvector approximation y_k and the new vector v_{k+1}. The associated timings are based on the respective sizes: $(t_s + k\, t_w)(N-1)$ and $(t_s + \frac{n}{N} t_w)(N-1)$. Notice that there is no particular communication during a restarted iteration. Combining the estimates given above, we obtain the communication time $T^{cm}(k)$ for one iteration

$$\begin{aligned} T^{cm}(k) &= [k(1+N) + 2(\log N + 2N - 2)]\, t_s + \\ &\quad +[k^2(1+\log N) + k(3N + 2\log N - 1) + (\frac{n}{N}+1)\,(N-1)]\, t_w \end{aligned}$$

Similarly to the computational timing analysis, we deduce the time $T^{cm}_{nr}(s)$ spent on s iterations of the non-restarted algorithm as

$$\begin{aligned} T^{cm}_{nr}(s) &= \sum_{k=1}^{s} T^{cm}(k) = [(1+N)\sum_{k=1}^{s} k + 2(\log N + 2N - 2)\sum_{k=1}^{s} 1]\, t_s + \\ &\quad +[(1+\log N)\sum_{k=1}^{s} k^2 + (3N + 2\log N - 1)\sum_{k=1}^{s} k + (\frac{n}{N}+1)\,(N-1)\sum_{k=1}^{s} 1]\, t_w \\ &= \frac{1}{3}(1+\log N)\, t_w\, s^3 + [\,\frac{1}{2}(1+N)\, t_s + \frac{3}{2}(\log N + N)\, t_w\,]\, s^2 + \\ &\quad +[\,(2\log N + \frac{9}{2}N - \frac{7}{2})\, t_s + (\frac{5}{2}N + \frac{7}{6}\log N - \frac{4}{3} + n(1-\frac{1}{N}))\, t_w\,]\, s\ , \end{aligned}$$

and the communication time $T^{cm}_{r}(s)$ for the restarted version as

$$T^{cm}_{r}(s) = \lfloor\frac{s-m}{q}\rfloor \left(T^{cm}_{nr}(m+q) - T^{cm}_{nr}(m-1)\right) + T^{cm}_{nr}(m+\delta-1). \quad (15)$$

When $\delta < q$ we can also derive an upper bound for the total restarted time as

$$T^{cm}_{r}(s) \leq \frac{s-m}{q}\left(T^{cm}_{nr}(m+q) - T^{cm}_{nr}(m-1)\right) + \ T^{cm}_{nr}(m+q)\ .$$

5 Numerical results

The parallel code for DSE was written in C. To achieve portability, we used MPI [15, 18] for the communication standard. The Message Passing Interface (MPI) is a set of standards which describes the user interface for a message passing library. It provides a large set of functions which controls point to point and collective communication. The application can either migrate to a different implementation of MPI or a different computer just by recompiling the code. Also, level 1, 2 and 3 BLAS routines [14] were employed. Currently many computer vendors provide optimized BLAS libraries which achieve a good performance for vector and matrix computations.

Our computational results were obtained for the coefficient matrix which arises from using a finite difference (based on a 5 point stencil) method for solving the following problem:

$$-\Delta u + gu = f \tag{16}$$

on a unitary square domain. Consider g to be null inside a 0.2×0.2 square on the center of the rectangle and $g = 100$ for the rest of the domain. Algorithm 4 was used for finding the ten smallest eigenpairs ($p = 10$) and assumed convergence for residual norms less than or equal to 10^{-7} ($\epsilon = 10^{-7}$). The restart indexes were $q = 10$ and $m = 15$. (This corresponds to apply restarting every time that the projected matrix S_k achieves order 25, reducing its order to 15.)

For the numerical results, we used a diagonal preconditioner as in the original Davidson algorithm, an ADI preconditioner, and a multigrid preconditioner [30]. The ADI preconditioner can be derived by considering a splitting of the matrix into two parts. In terms of differential equations this corresponds to solving the linear system, first in relation to one direction and then in the other direction. An acceleration parameter in a fashion similar to the SOR method can be incorporated. Details of these methods are shown in [38, 40, 41].

The multigrid solver is a well established iterative method for solving the systems arising from discretizations of partial differential equations. In contrast with classical iterative methods (Jacobi, Gauss-Seidel, SOR, or SSOR with Chebyshev acceleration) the convergence rate of multigrid methods does not slow down as the size of the problems increase [12]; in other words, for a given accuracy, it can solve problems in n unknowns in $O(n)$ time. A good

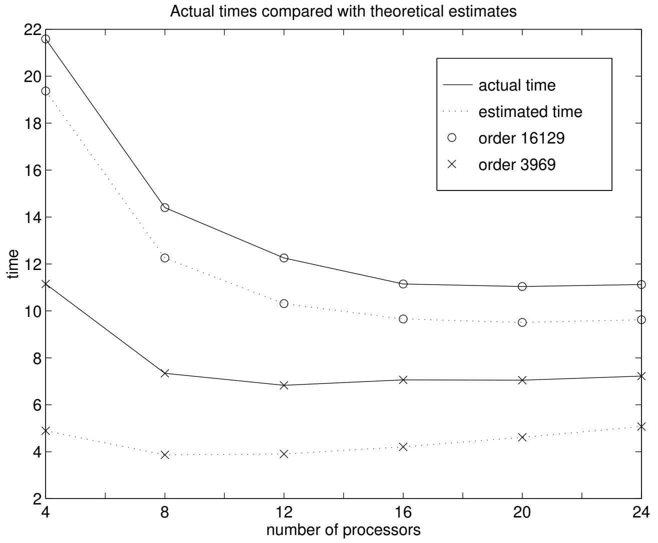


Figure 4: Actual and estimated times for equation (16). Results for two different matrix sizes performance on the Paragon are shown.

introduction for multigrid algorithms is given in [6].

For the calculation of eigenvalues of S_k, through equation (8), we followed [28] but used Brent's root finder [5] instead of bisection and secant methods.

Figure 4 compares estimates developed in Section 4 with the actual running timings for two distinct grid sizes: 63×63 and 127×127 (matrices of orders 3969 and 16129, respectively.). Each problem was solved using 4, 8, 12, 16, 20, and 24 processors on the Intel Paragon computer and the observed running timings[3] were plotted. Thus, the two values of s (number of iterations), for the two different matrices sizes were used in equations (14) and (15) and then the estimated timings in Figure 4 were derived.

Since the timing analysis only considers the major communication and com-

[3]Running times refer to the DSE algorithm excluding the time spent applying the preconditioner M.

putational efforts, the timing estimates are expected to be smaller than the actual algorithm timings. This can be demonstrated in Figure 4. Nevertheless it can describe the behavior and performance as the number of processors increases. The model proved to be very satisfactory in detecting potential bottlenecks prior to coding the algorithm.

As MPI and BLAS libraries were available in both platforms, migration between the Intel Paragon and nCUBE is straightforward. Optimized BLAS is available on Paragon's operating system [20]. A Fortran implementation of BLAS from Netlib [14] was compiled on the nCUBE. Table 1 compares actual running times in seconds for our test case on both computers. The different curves are due to different granularities. Also after a certain number of processors is reached, each curve rises away from its minimum point; this is due to increased interprocessor communication overhead. Our theoretical model predicts the optimal number of processors associated for each granularity.

processors	matrix order 961 (31×31)		matrix order 3969 (63×63)		order 16129
	Paragon	nCUBE	Paragon	nCUBE	Paragon
4	6.4	13	11	52	22
8	4.5	11	7.3	33	14
16	4.5	11	7.1	27	11
20	4.7	$*^1$	7.1	$*^1$	11
24	4.9	$*^1$	7.2	$*^1$	11

$*^1$ not a subcube size.

Table 1: Timings in seconds for problem (16) in the nCUBE and Paragon computers.

The number N of processors for which the total parallel running time $T_r =$

$T_r^{cp} + T_r^{cm}$ achieves its minimum is determined by $\frac{\partial T_r(N)}{\partial N} = 0$. This equation provides an upper-bound for the number of processors to be allocated on a given problem. As the number of processors exceeds the upper-bound, either the parallel run time stays asymptotically at minimal value, or rises away from the it. Equations (14) and (15) were differentiated[4] in relation to N resulting in a function depending on $1/N^2$. The positive root, which indicates that the maximum number of processors to be allocated is $N = 5$ for a matrix of order 961, $N = 10$ for order 3969, and $N = 20$ for a matrix order of 16129. The data in Figure 4 and Table 1 show that our timings for the mesh architecture (Paragon in our case) followed the theoretical estimates.

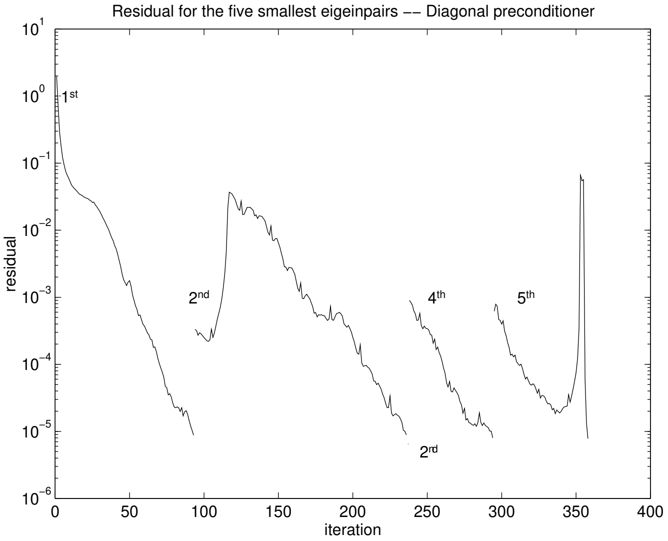
Preconditioning is another relevant issue concerning performance [30]. Figure 5 compares the convergence history of the residual norm for the first five eigenpairs for a matrix of order 3969. Here convergence was estimated using $\epsilon = 10^{-5}$. The restart indexes were $q = 10$ and $m = 15$. For the numerical results in Figure 5(a), a diagonal preconditioner was applied, that is, $M = (D - \lambda I)^{-1}$ where D is the main diagonal of A, the matrix discretization for (16). At first, one may think that the points of sudden increase in the residual are a consequence of restart steps. However, the multigrid preconditioner do not present such problem (Figure 5(b)). Moreover, with the multigrid preconditioner DSE finds all five eigenvalues after 39 iterations while the diagonal version requires 358 iterations. The variations in Figure 5(a) are due to the poor performance provided by a diagonal preconditioner. Indeed, each eigenvector approximation is obtained by preconditioning the residual of the

[4]Notice the above calculations still depend on the parameter s, the number of iterations performed by the DSE algorithm. Further analysis may provide a prior upper bound for s dependent on the kind of matrix studied, so that the timing model can be applied as a tool when deciding the number of processors to be allocated.

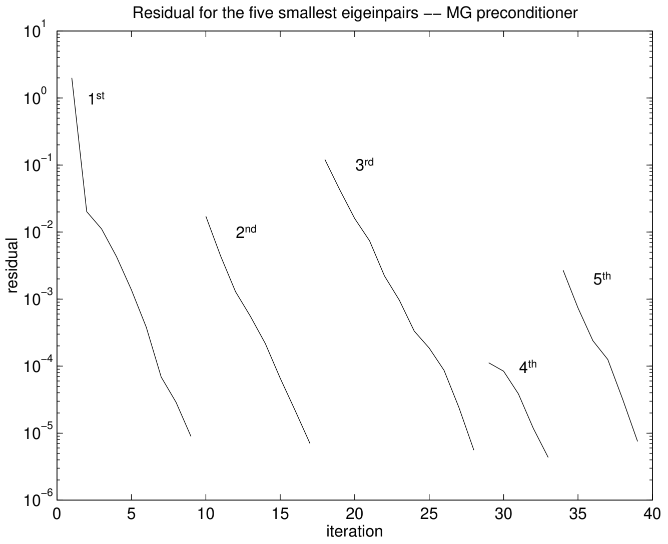
current approximate solution. Although the eigenvalue approximation λ_k converges monotonically to the actual eigenvalue, the preconditioner may worsen the angle between successive eigenvectors v_k, and $span(V_k)$. It can also worsen the convergence rate by reducing the accuracy of the approximate eigenvectors. Moreover, the diagonal preconditioner may induce the lock in phenomena; that is, the algorithm will converge to eigenvalue λ defined by the current matrix $(D - \lambda I)^{-1}$ instead of the actual p^{th} eigenvalue. Consequently, the algorithm can attribute wrong multiplicity to the eigenvalues. In our example, the second eigenvalue is repeated, but the DSE algorithm with preconditioner $(D - \lambda I)^{-1}$ does not demonstrate the eigenvalue multiplicity of two. Our last Figure illustrates numerical results with an ADI preconditioner. For our test cases, we obtained the best results with this preconditioner. Figure 6 compares our timings against PARPACK [26]. PARPACK is a parallel package available for the calculation of several eigenvalues based on the Arnoldi method. For the cases studied, our timings are always competitive with PARPACK and significantly better for increased problem size.

6 Conclusions

Recently, progress has been made in the use of Generalized Davidson algorithms for finding several eigenvalues [30]. In this paper we describe the a Davidson for Several Eigenvalues (DSE) algorithm, and develop its parallelization. The main improvement of this paper is the way the eigenvalues are calculated for the subspaces. The relationship between the consecutive subspaces allowed us to develop a highly parallel step for this part of Davidson type algorithms, particularly DSE. The greatest part of the computational work consists of matrix vector multiplies. For the parallel implementation, row partitioning of data was



(a)



(b)

Figure 5: Residual norms for the first five eigenpairs for an 63×63 discretization of equation (16). Figure (a): Diagonal Preconditioner. Figure (b): Multigrid Preconditioner.

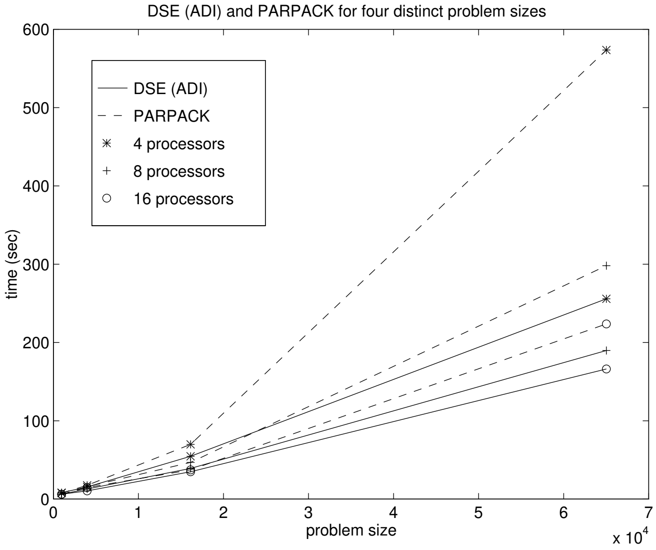


Figure 6: Comparison with PARPACK.

used; BLAS and the MPI library were incorporated. Other parts of the code, additional to matrix vector multiplies, generated the timing model and predicted the general behavior well. From the numerical results, when multigrid is used instead of the diagonal preconditioner (as in the Davidson algorithm) we showed that the number of iterations is significantly decreased. Even better results were obtained with an ADI preconditioner. Our implementation is portable, and numerical results were compared between the the nCUBE and Paragon, with better results on the latter. Finally, our code is competitive with PARPACK and better in some cases.

Acknowledgements

The authors would like to thank the reviewers and editors of this journal for a number of suggestions which have improved the presentation of this work.

Appendix: Stabilizing the eigenvector calculation

The DSE algorithm employs the arrowhead-decomposition iteratively. Using the arrowhead representation, the first p columns of V_{k-1} approach eigenvectors of A. Thus the corresponding entries of $\tilde{s}_k$ approach zero. Such behavior may produce roundoff errors on the eigenvector calculation: as the j^{th} diagonal entry d_j converges to the j^{th} eigenvalue $\bar{\lambda}_j$ of the original problem, the eigenvalue approximation λ_j also converges to $\bar{\lambda}_j$. Consequently, the ratio $u = \tilde{s}_k(j)/(\lambda_j - d_j)$ presented in the eigenvector calculation (9), will operate on $\tilde{s}_k(j) \to 0$ and $\lambda_j - d_j \to 0$ which may result in a wrong estimate for one entry of the eigenvector q_l. To overcome this we avoid evaluating ratio u explicitly. Instead, the identity $\varphi(\lambda_j) = 0$ can be used to obtain u in (9). From (8) we have

$$-d_j = \varphi(\lambda_j) - d_j \Rightarrow \lambda_j - d_j = s_{kk} - d_j + \frac{[\tilde{s}_k(j)]^2}{\lambda_j - d_j} + \sum_{\substack{l=1 \\ l \neq j}}^{\bar{k}-1} \frac{[\tilde{s}_k(l)]^2}{\lambda_j - d_l} ,$$

which is divided by $\tilde{s}_k(j) \neq 0$, leading to

$$u^2 + \alpha u - 1 = 0 , \qquad \alpha = \frac{s_{kk} - d_j + \sum_{\substack{l=1 \\ l \neq j}}^{\bar{k}-1} \frac{[\tilde{s}_k(l)]^2}{\lambda_j - d_l}}{\tilde{s}_k(j)} .$$

Thus $u = (-\alpha \pm \sqrt{\alpha^2 + 4})/2$, where the choice of the positive or negative root is made according to the sign of $\tilde{s}_k(j)/(\lambda_j - d_j)$.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philadelphia, 1992.

[2] H. BATEMAN, *On a set of kernels whose determinants form a sturmian sequence*, Bull. of American Mathematical Society, 18 (1912), pp. 179–182.

[3] N. BECKER, *Note on the parallel efficiency of the Frederickson-McBryan multigrid algorithm*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 208–220.

[4] L. BORGES, *New parallel algorithms for eigenvalue problems*, PhD thesis, Texas A&M University, 1998.

[5] R. P. BRENT, *Algorithms for minimization without derivatives*, Prentice-Hall, 1973.

[6] W. L. BRIGGS, *A Multigrid Tutorial*, SIAM Publ., Philadelphia, PA, 1987.

[7] D. CALVETTI, L. REICHEL, AND D. SORENSON, *An implicitly restarted Lanczos method for large symmetric eigenvalue problems*, Electronic Trans. Numer. Anal., 2 (1994), pp. 1–21.

[8] G. CISNEROS, M. BERRONDO, AND C. F. BRUNGE, *DVDSON: A subroutine to evaluate selected sets of eigenvalues and eigenvectors of large symmetric matrices*, Compu. Chem., 10 (1986), pp. 281–291.

[9] G. CISNEROS AND C. F. BRUNGE, *An improved computer program for eigenvector and eigenvalues of large configuration iteraction matrices using the algorithm of Davidson*, Compu. Chem., 8 (1984), pp. 157–160.

[10] E. R. DAVIDSON, *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices*, J. Comp. Phys., 17 (1975), pp. 87–94.

[11] E. R. DAVIDSON, *Super matrix methods*, J. Comp. Phys. Communications, 53 (1989), pp. 49–60.

[12] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM Publ., Philadelphia, PA, 1997.

[13] R. V. der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, in Supercomputing '93, IEEE Computer Society Press, Los Almitos, CA, 1993, pp. 102–111.

[14] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM trans. Math. Soft., 16 (1990), pp. 1–17.

[15] J. J. Dongarra, S. W. Otto, and D. W. M. Snir, *A message passing standard for MPP and workstations*, Comm. ACM, 39 (1996), pp. 84–90.

[16] P. Frederickson and O. McBryan, *Normalized convergence rates for the PSMG method*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 221–229.

[17] G. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins Press, 2nd ed., 1989. 1st edition published 1983 by North Oxford Academic.

[18] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, 1994.

[19] K. Hwang, *Advanced Computer Architecture: parallelism, scalability, programmability*, McGraw-Hill, 1993.

[20] Intel Corporation, *Paragon System User's Guide*, 1995.

[21] H. Jiang and Y. Wong, *A parallel alternating direction implicit preconditioning method*, J. Comput. Appl. Math., 36 (1991), pp. 209–226.

[22] D. Kincaid and W. Cheney, *Numerical Analysis*, Brooks/Cole, 1990.

[23] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, 1994.

[24] P. Leca and L. Mane, *A 3-D ADI algorithm on distributed memory multiprocessors implementations and results*, in Parallel Computational Fluid Dynamics, H. Simon, ed., MIT Press, 1992, pp. 149–165.

[25] T. Manteuffel, S. McCormick, J. Morel, S. Oliveira, and G. Yang, *A parallel version of a multigrid algorithm for isotropic transport equations*, SIAM J. Sci. Comput., 15 (1994), pp. 474–493.

[26] K. Maschhoff and D. Sorensen, *A portable implementation of ARPACK for distributed memory parallel architectures*, in Proceedings of Copper Mountain Conference on Iterative Methods, April 9–13 1996.

[27] C. W. Murray, S. C. Racine, and E. R. Davidson, *Improved algorithms for the lowest few eigenvalues and associated eigenvectors of large matrices*, J. Comp. Phys., 103 (1991), pp. 382–389.

[28] D. P. O'Leary and G. W. Stewart, *Computing the eigenvalues and eigenvectors of symmetric arrowhead matrices*, J. Comp. Phys., 90 (1990), pp. 497–505.

[29] D. P. O'Leary and P. Whitman, *Parallel QR factorization by Householder and modified Gram-Schmidt algorithms*, Parallel Computing, 16 (1990), pp. 99–112.

[30] S. Oliveira, *A convergence proof of an iterative subspace method for eigenvalues problem*, in Foundations of Computational Mathematics Selected Papers, F. Cucker and M. Shub, eds., Springer, January 1997, pp. 316–325.

[31] ———, *A new parallel chasing algorithm for transforming arrowhead matrices to tridiagonal form*, Mathematics of Computation, 67 (1998), pp. 221–235.

[32] B. N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice-Hall, 1980.

[33] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, Halsted Press, 1992.

[34] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix eigensystem routines: EISPACK guide*, no. 6 in Lecture Notes Comput. Sci., Springer–Verlag, Berlin, Heidelberg, New York, second ed., 1976.

[35] A. Stathopoulos and C. F. Fischer, *Reducing synchronization on the parallel davidson method for the large, sparse, eigenvalue problem*, in Proceedings of Supercomputing Conference, 1993, pp. 172–180.

[36] ———, *A Davidson program for finding a few selected extreme eigeinpairs of a large, sparse, real, symmetric matrix*, Comp. Phys. Comm., 79 (1994), pp. 268–290.

[37] V. M. Umar and C. F. Fischer, *Multitasking the Davidson algorithm for the large, sparse eigenvalue problem*, Int. J. Supercomput. Appl., 3 (1989), pp. 28–53.

[38] R. S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, 1963.

[39] J. Weber, R. Lacroix, and G. Wanner, *The eigenvalue problem in configuration iteration calculations: A computer program based on a new derivation of the algorithm of Davidson*, Compu. Chem., 4 (1980), pp. 55–60.

[40] J. R. Westlake, *A handbook of numerical matrix inversion and solution of linear equations*, Wiley, 1968.

[41] D. M. Young, *Iterative Solution of Large Linear Systems*, Academic Press, 1971.