INCREASING THE PERFORMANCE OF THE JACOBI-DAVIDSON METHOD BY BLOCKING [*]

MELVEN RÖHRIG-ZÖLLNER[†], JONAS THIES[†], MORITZ KREUTZER[‡], ANDREAS ALVERMANN[§], ANDREAS PIEPER[§], ACHIM BASERMANN[†], GEORG HAGER[‡], GERHARD WELLEIN[‡], AND HOLGER FEHSKE[§]

Abstract. Block variants of the Jacobi-Davidson method for computing a few eigenpairs of a large sparse matrix are known to improve the robustness of the standard algorithm, but are generally shunned because the total number of floating-point operations increases. In this paper we present the implementation of a block Jacobi-Davidson solver. By detailed performance engineering and numerical experiments we demonstrate that the increase in operations is typically more than compensated by performance gains on modern architectures, giving a method that is both more efficient and robust than its single vector counterpart.

Key words. sparse eigenvalue problems, Jacobi-Davidson, block methods, performance engineering, high performance computing, multi-core processors, hybrid parallel implementation

[*] This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 “Software for Exascale Computing” (SPPEXA) under project ESSEX.

[†] German Aerospace Center (DLR), Simulation and Software Technology

[‡] Erlangen Regional Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg

[§] Institut für Physik, Ernst-Moritz-Arndt-Universität Greifswald, Germany

1. Introduction. We consider the problem of finding a small number of extremal eigenvalues and corresponding eigenvectors of a large, sparse matrix $A \in \mathbb{R}^{n\times n}$,

$$Av_i = \lambda_i v_i, \qquad i = 1, \ldots, l, \quad l \ll n. \tag{1.1}$$

We will also comment from time to time on the closely related generalized eigenproblem

$$Av_i = \lambda_i B v_i, \qquad i = 1, \ldots, l, \tag{1.2}$$

with $B = B^* \in \mathbb{R}^{n\times n}$ Hermitian positive definite, to which our method can be generalized straightforwardly. Although we present test cases and performance results for real-valued matrices, all results carry over to the complex case (in which the performance models have to be adjusted slightly to account for the additional data transfers and floating-point operations).

Eigenproblems of the form (1.1) or (1.2) arise in many scientific and engineering applications such as structural mechanics and material science, to name just two. The main application of the authors here is quantum mechanics, where A is a sparse matrix representation of the Hamiltonian in the Schrödinger equation. Here, we choose a spin chain model as a typical example from solid state physics (see Section 5). The methods presented can obviously be used in any application that requires finding a few exterior eigenvalues of a large sparse operator and are suitable for both Hermitian and general (non-Hermitian) operators A.

Related work. In this paper we investigate a block Jacobi-Davidson method that performs matrix-vector products and vector-vector operations with several vectors at once. Jacobi-Davidson (JD) methods for the calculation of several eigenvalues were originally proposed in [9]. Since then, many authors have worked on the algorithm, its theoretical properties and efficient implementation. For a review article, see [12]. Stathopoulos and McCombs investigated Jacobi-Davidson and Generalized Davidson methods for symmetric (respectively Hermitian) eigenvalue problems in [23] and also addressed block methods briefly. The general consensus in the literature seems to be that block methods do not "pay off" in the sense that the performance gains do not justify the overall increase in the number of operations. In this paper we seek to demonstrate the opposite by extending the performance model for the sparse matrix-vector product given in [16] to the case of multiple vectors and a careful pipelining of operations to optimally exploit performance gains. We also want to provide a thorough derivation and discussion of a block variant, as we found this to be missing in the literature to date.

The performance analysis and algorithmic principles presented here are also useful for other algorithms like block Krylov methods or eigenvalue solvers that require the solution of linear systems with multiple right-hand sides such as FEAST [26] or TraceMin [15].

Challenges posed by modern computer hardware. In this section we discuss some aspects of present high performance computing (HPC) systems that are crucial for the development of efficient sparse linear algebra algorithms. For an extensive introduction to the topic, see [10]. A simplified model for a present supercomputer is a cluster of compute nodes connected by some network (*distributed memory* architecture), where each compute node consists of several cores - i.e. sequential computing units - that share memory and other resources (*shared memory* architecture). Additionally each node may contain special accelerator devices, but for simplicity we only consider clusters of multi-core nodes here.

Communication between nodes requires sending messages over the network, which is usually slow (in terms of bandwidth and latency) compared to memory accesses. On the node level, the speed of the main memory is already insufficient to keep the cores working. To hide this effect, a hierarchy of caches is available. The usual setup consists of one small and fast cache per core and one or several slower caches that are possibly shared by all or several cores in a node. Data is fetched into the cache from the main memory in fixed-length *cache lines* of consecutive elements.

From this machine model we can see that the overall performance of a computer program is determined by two main aspects: parallelism—the ability to distribute work among the nodes and the cores within a node—and data locality—the ability to reduce data traffic by reusing data in the cache (temporal locality) or using as many elements from each cache line as possible (spatial locality). In our experience, many authors emphasize parallel scalability and neglect the discussion of the node-level performance. We would like to point out that optimal use of all available cores increases the energy efficiency of a program, rather than just reducing the runtime at an increasing energy cost. This paper clearly focuses on the node-level optimization, though aspects of multi-node performance, such as avoiding communication, are addressed as well.

Document structure. In Section 2 we derive a block formulation of the Jacobi-Davidson method and discuss its properties and relation with other methods. In Section 3 we perform a series of benchmark experiments for the computational kernels required. This is intended to motivate the use of block methods without considering the numerical effects that may obscure the pure performance characteristics in the context of a complete eigenvalue solver. Section 4 describes some aspects of the efficient implementation of the proposed algorithm, and the paper is concluded with an experimental investigation of the numerical behavior and computational efficiency in Section 5, where we also compare our results with an existing software package with similar functionality (PRIMME, [25]).

2. The block Jacobi-Davidson QR (BJDQR) method.

For solving the large sparse eigenvalue problems (1.1) or (1.2), the subspace iteration algorithm is one of the simplest methods. The original version was introduced by Bauer under the name of *Treppeniteration* (staircase iteration) in [3]. Practical implementations apply projection and deflation techniques. Krylov subspace methods are based on projection methods, both orthogonal and oblique, onto Krylov subspaces, i.e., subspaces spanned by the iterates of the simple power method. Well-known representatives are the (non-)Hermitian Lanczos algorithm and Arnoldi's method and its variations. Davidson's method, widely used among chemists and physicists, is a generalization of the Lanczos algorithm and can be seen as a preconditioned Lanczos method. A significantly faster method for the determination of several extremal eigenvalues and eigenvectors is the Jacobi-Davidson QR (JDQR) method with efficient preconditioning. It combines an outer iteration of the Davidson type with inner iterations to solve auxiliary linear systems. These inner systems can be solved iteratively using Krylov subspace methods.

In the following, we focus on a block formulation of JD. Starting point for the derivation of this method is the invariant subspace $\mathcal{V} = \operatorname{span}\{v_1, \dots, v_l\}$ spanned by the eigenvectors v_i of (1.1), but for general, non-Hermitian matrices the conditioning of the eigenvector basis of $\mathcal{V}$ may be arbitrarily bad.
If we consider an orthonormal basis of the invariant subspace, we obtain the following

formulation of the problem (1.1):

$$\begin{cases} AQ - QR &= 0, \\ -\frac{1}{2}Q^*Q + \frac{1}{2}I &= 0. \end{cases} \tag{2.1}$$

Here $AQ = QR$ denotes a partial Schur decomposition of A with an orthogonal matrix $Q \in \mathbb{C}^{n\times l}$ and an upper triangular matrix $R \in \mathbb{C}^{l\times l}$ with $r_{i,i} = \lambda_i$. There are also other suitable formulations, see [29] for a discussion of the single vector case.

We can apply a Newton scheme to the nonlinear system of equations (2.1), which yields a block Jacobi-Davidson style QR algorithm (see [9]). First, we write (2.1) as corrections ΔQ and ΔR for existing approximations $\tilde{Q}$ and $\tilde{R}$,

$$\begin{cases} A(\tilde{Q} + \Delta Q) - (\tilde{Q} + \Delta Q)(\tilde{R} + \Delta R) &= 0, \\ -\frac{1}{2}(\tilde{Q} + \Delta Q)^*(\tilde{Q} + \Delta Q) + \frac{1}{2}I &= 0. \end{cases} \tag{2.2}$$

Ignoring quadratic terms and assuming an orthogonal approximation $\tilde{Q}$, we obtain

$$\begin{cases} A\Delta Q - \Delta Q\tilde{R} &\approx -(A\tilde{Q} - \tilde{Q}\tilde{R}) + \tilde{Q}\Delta R, \\ \frac{1}{2}\tilde{Q}^*\Delta Q + \frac{1}{2}\Delta Q^*\tilde{Q} &\approx 0. \end{cases} \tag{2.3}$$

The second equation indicates that the term $\tilde{Q}^*\Delta Q$ must be skew-Hermitian. In a subspace iteration we are only interested in the part of ΔQ perpendicular to $\tilde{Q}$ as only these directions extend the search space. We split the correction into two parts, its projections onto $\tilde{Q}$ and onto the orthogonal complement $\tilde{Q}^\perp$ of $\tilde{Q}$, respectively:

$$\Delta Q = \underbrace{\tilde{Q}\tilde{Q}^*\Delta Q}_{\Delta Q^\parallel} + \underbrace{(I - \tilde{Q}\tilde{Q}^*)\Delta Q}_{\Delta Q^\perp}. \tag{2.4}$$

In order to improve the conditioning of the linear problem, we use the projection of A onto $\tilde{Q}^\perp$:

$$\begin{aligned} & A^\perp := (I - \tilde{Q}\tilde{Q}^*)A(I - \tilde{Q}\tilde{Q}^*) \\ \Leftrightarrow \quad & A = A^\perp + \tilde{Q}\tilde{Q}^*A + A\tilde{Q}\tilde{Q}^* - \tilde{Q}\tilde{Q}^*A\tilde{Q}\tilde{Q}^*. \end{aligned} \tag{2.5}$$

With these expressions for A and ΔQ and noting that $A^\perp\tilde{Q} = 0$, we get

$$\begin{aligned} A^\perp\Delta Q^\perp - \Delta Q^\perp\tilde{R} \approx & -(A\tilde{Q} - \tilde{Q}\tilde{R}) - (I - \tilde{Q}\tilde{Q}^*)A\Delta Q^\parallel \\ & + \tilde{Q}\tilde{Q}^*(A\Delta Q - \Delta Q\tilde{R} + \tilde{Q}\Delta R). \end{aligned} \tag{2.6}$$

The first term on the right-hand side is the current residual. If the current approximation satisfies a Galerkin condition, $(A\tilde{Q} - \tilde{Q}\tilde{R}) \perp \tilde{Q}$, all terms in the first line of the equation are orthogonal to $\tilde{Q}$ and the second line vanishes. In this case we can also express the second term on the right-hand side using the residual:

$$\begin{aligned} (I - \tilde{Q}\tilde{Q}^*)A\Delta Q^\parallel &= (I - \tilde{Q}\tilde{Q}^*)A\tilde{Q}\tilde{Q}^*\Delta Q \\ &= (I - \tilde{Q}\tilde{Q}^*)\Big((A\tilde{Q} - \tilde{Q}\tilde{R}) + \tilde{Q}\tilde{R}\Big)\tilde{Q}^*\Delta Q \\ &= (A\tilde{Q} - \tilde{Q}\tilde{R})(\tilde{Q}^*\Delta Q). \end{aligned} \tag{2.7}$$

Near the solution both the residual and the correction are small (and $\tilde{Q}$ is orthogonal), so this presents a second order term that we neglect in the following. We obtain a JD style correction equation for an approximate Schur form:

$$(I - \tilde{Q}\tilde{Q}^*)A(I - \tilde{Q}\tilde{Q}^*)\Delta Q - (I - \tilde{Q}\tilde{Q}^*)\Delta Q\tilde{R} = -(A\tilde{Q} - \tilde{Q}\tilde{R}). \tag{2.8}$$

This is a Sylvester equation for $\Delta Q^{\perp}$. As $\tilde{R}$ is upper triangular, we can also write (2.8) as a set of correction equations with a modified right-hand side for general matrices:

$$(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)(I - \tilde{Q}\tilde{Q}^*)\Delta q_i = -(A\tilde{q}_i - \tilde{Q}\tilde{r}_i) - \sum_{j=1}^{i-1} \tilde{r}_{j,i}\Delta q_j^{\perp}, \quad i = 1 \dots l. \tag{2.9}$$

With this formulation we need to solve the correction equations successively for $i = 1 \dots l$. This prevents us from exploiting the performance benefits of block methods. So from a computational point of view it would be desirable to ignore the coupling terms $\sum_{j=1}^{i-1} \tilde{r}_{j,i}\Delta q_j$, which yields the uncoupled form

$$(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)(I - \tilde{Q}\tilde{Q}^*)\Delta q_i \approx -(A\tilde{q}_i - \tilde{Q}\tilde{r}_i), \qquad i = 1 \dots l. \tag{2.10}$$

For Hermitian matrices, this is identical to (2.8) as $\tilde{R}$ is diagonal in this case. The following argument shows that even for general A the uncoupled formulation should provide suitable corrections Δq_i for a subspace iteration. The classical JDQR method [9] uses the following correction equation for a single eigenvalue (with deflation of an already converged eigenbasis Q_k and $\bar{Q} = \begin{pmatrix} Q_k & \tilde{q} \end{pmatrix}$):

$$(I - \bar{Q}\bar{Q}^*)(A - \tilde{\lambda} I)(I - \bar{Q}\bar{Q}^*)\Delta q = -(I - Q_k Q_k^*)(A\tilde{q} - \tilde{\lambda}\tilde{q}). \tag{2.11}$$

The individual correction equations from (2.10) are similar to (2.11), with the difference that they include a deflation of eigenvector approximations that have not converged yet. The residuals of the two formulations are related in this way: in (2.11) we need to orthogonalize the residual $A\tilde{q} - \tilde{\lambda}\tilde{q}$ with respect to Q_k, in the formulation $(A\tilde{q}_i - \tilde{Q}\tilde{r}_i)$ from (2.10) we obtain directly the part of the residual of a single eigenvalue orthogonal to the eigenvector approximations due to the Galerkin condition of the surrounding subspace iteration.

Generalized eigenproblems. We can use a similar approach for the generalized eigenvalue problem (1.2) with Hermitian positive definite B if we require Q to be B-orthogonal. The resulting uncoupled block correction equation (corresponding to (2.10)) is:

$$(I - (B\tilde{Q})\tilde{Q}^*)(A - \tilde{\lambda}_i B)(I - \tilde{Q}(B\tilde{Q})^*)\Delta q_i \approx -(A\tilde{q}_i - B\tilde{Q}\tilde{r}_i), \quad i = 1 \dots l. \tag{2.12}$$

Relation with the Rayleigh quotient iteration (RQI). The Newton approach may not explain the convergence behavior of JD methods very well. If the subspace span$\{q_1, \dots, q_l\}$ of a partial Schur decomposition (Q, R) contains only part of the invariant subspace of a multiple eigenvalue, the Jacobian of (2.1) may be singular so that the conditions for local quadratic convergence are not met [29]. Zhou et al. [30] show that more insight into the convergence of JD is gained by considering its relation to the Rayleigh quotient iteration (RQI): in each iteration the JD correction extends the subspace in such a way that it contains the RQI direction $x_{RQI} = (A - \rho_A(\tilde{q})I)^{-1}\tilde{q}$. This relation can be generalized to our block correction equation (2.8). Absil et al. [1]

propose a generalization of the RQI method to invariant subspaces with cubic convergence for Hermitian matrices. The next direction $X_{GRQI} \in \mathbb{C}^{n\times l}$ of the *Grassmann-Rayleigh quotient iteration* (GRQI) is calculated from the following Sylvester equation:

$$AX_{GRQI} - X_{GRQI}R_A(\tilde{Q}) = \tilde{Q}, \tag{2.13}$$

with the *matrix Rayleigh quotient* $R_A(\tilde{Q}) = (\tilde{Q}^*\tilde{Q})^{-1}\tilde{Q}^*A\tilde{Q}$. As our approximate partial Schur decomposition $(\tilde{Q}, \tilde{R})$ satisfies a Galerkin condition, we obtain $\tilde{R} = R_A(\tilde{Q})$. Remembering the *homogeneity* property of GRQI (algorithm independent of the representation of the subspace), we can show that (2.8) leads to corrections satisfying

$$\tilde{Q} + \Delta Q^{\perp} = X_{GRQI}M \tag{2.14}$$

as long as $M = \tilde{Q}^*A\Delta Q^{\perp}$ is invertible. If M is not invertible it is at least nonzero and (2.14) holds for its largest regular submatrix and the corresponding columns of $\Delta Q^{\perp}$.

Inexact solution. So if (2.10) is solved exactly in a subspace method, we may expect cubic convergence to an invariant subspace for the Hermitian case. In the non-Hermitian case we still have quadratic convergence to at least single eigenvalues (from the standard RQI). In [18], Notay shows for a special case that the fast convergence is preserved even with approximate corrections under the condition that we increase the required accuracy of the corrections in the outer iteration fast enough. This obviously holds for the block algorithm as well, and we will discuss the practical implementation of varying the “inner tolerance” for each eigenvalue approximation in Section 4.1.

Computational kernels. Block variants of iterative methods in general aim to achieve higher performance by exploiting faster computational kernels. For the method described in this section, we assume that one of the main contributors to the overall runtime is the application of the operator $(I - QQ^T)(A - \sigma I)$ to a vector in each iteration of an inner iterative solver for (2.10). In Section 4.1 we will show how to implement the algorithm such that this operator is (almost) always applied to a fixed number of vectors at a time. To motivate this effort, we will next quantify the performance advantages using simple qualitative models and a case study. Another important operation is the orthogonalization of a block of vectors against an existing orthogonal basis. This step also benefits from block operations and will be briefly discussed in Section 4.2.

3. Performance engineering for the key operations.

In the field of sparse eigensolvers in general it is often possible to extend existing algorithms that determine one eigenvector at a time to block algorithms that search for a set of eigenvalues and -vectors at once. This is interesting from a numerical point of view since subspace methods usually gather information for several eigenvalues near a specific target automatically. Blocking of operations is also beneficial for the performance of the implementation. By grouping together several matrix-vector multiplications of the same matrix with different right-hand-side vectors (in the following called sparse matrix-multiple-vector multiplication, spMMVM in distinction to spMVM for the single-vector case), we can achieve that the matrix needs to be loaded only once from main memory for several vectors. Additionally, faster dense matrix operations can be employed for the vector-vector calculations when using block vectors. The number of messages sent for all key operations can also be reduced by using block vectors, although the amount of communicated data remains the same. Obviously, we can also combine this approach with appropriate matrix reorderings (e.g. using RCM [6], Scotch [5] or ParMETIS [14]) to further reduce the communication effort in the spMMVM.

SpMMVM performance models and implementation. Already for moderately sized problems the spM(M)VM kernel is memory-bound on all modern computer architectures. This is due to the fact that the kernel's code balance (ratio of accessed data from main memory to executed floating-point operations) is larger than typical values of machine balance (ratio of maximum memory bandwidth to arithmetic peak performance). In addition, matrix properties like the density (share of non-zero entries) and the sparsity pattern (distribution of non-zero entries) can have a large influence on the performance.

A key factor for high spM(M)VM performance is the storage format of the sparse matrix. While the popular compressed row storage (CRS) usually gives good performance on CPUs, the more sophisticated SELL-C-σ format [16] yields high performance on a much wider set of architectures for many matrices in practice. Extending their results to the spMMVM ($Y \leftarrow AX, X, Y \in \mathbb{R}^{n \times n_b}$ with a block size n_b), our experiments have shown that it is important to use row-major storage for the blocks of vectors X, Y, rather than the commonly used column-major ordering. This design choice improves the spatial cache locality during the spMMVM and thus the overall performance, independent of the matrix sparsity pattern or storage scheme. On the other hand, the consecutive storage of many vectors in row-major ordering may lead to strided memory access if single vectors or small blocks need to be accessed, which led us to some of the implementation choices discussed in Section 4.

Assuming 64-bit matrix/vector values and 32-bit indices, no overhead in the matrix storage (which can be achieved by sorting the matrix rows locally for SELL-C-σ) and a large enough cache to hold X during the entire operation, the minimal data volume for an spMMVM can be computed as

$$V_{\min} = 12 n_{nz} + 16 n\, n_b \text{ bytes}, \tag{3.1}$$

where n_{nz} denotes the number of non-zero matrix entries. The minimum code balance of the spMMVM kernel for processing a single non-zero matrix element is then given by

$$B_C = \frac{12 + 16\frac{n_b}{n_{nzr}}}{2n_b} \frac{\text{bytes}}{\text{flop}} = \left(\frac{6}{n_b} + \frac{8}{n_{nzr}}\right) \frac{\text{bytes}}{\text{flop}}, \tag{3.2}$$

with $n_{nzr} = \frac{n_{nz}}{n}$ the average number of non-zero entries per row. Here we employ non-temporal stores for the result vector Y, which saves a load of this vector's data from the memory. In practice we expect a larger data transfer volume due to multiple loads of X elements from the main memory (the cache size is limited and the access pattern to X may be sub-optimal).

More details on this topic can be found for example in [21] and [16]. The actual transferred data volume $V_{\text{meas}} \geq V_{\min}$ can be measured with hardware performance monitoring tools like LIKWID [27], and the traffic overhead due to multiple loads of elements of X can be quantified as

$$V_{\text{extra}} = V_{\text{meas}} - V_{\min}. \tag{3.3}$$

In order to get an idea of the impact of V_{extra} on the performance, a measure for the relative data traffic can be formulated as

$$\Omega = \frac{V_{\text{meas}}}{V_{\text{min}}} \geq 1. \tag{3.4}$$

A value of $\Omega = 1$ indicates the optimal case where each element of X is loaded only once from the main memory in an spMMVM.

Assuming boundedness of the kernel to the machine's maximum memory bandwidth b [bytes/s], one can apply a simple roofline performance model (cf. [28] and the references therein) in order to predict the maximum performance P^* [flops/s]:

$$P^* = \frac{b}{B_C} = \frac{b}{\frac{6}{n_b} + \frac{8}{n_{nzr}}}. \tag{3.5}$$

Using this relation, one can also predict the potential speedup compared to n_b single spMVMs:

$$S^* = \frac{P^*_{n_b}}{P^*_{n_b=1}} = n_b \cdot \frac{6n_{nzr} + 8}{6n_{nzr} + 8n_b}. \tag{3.6}$$

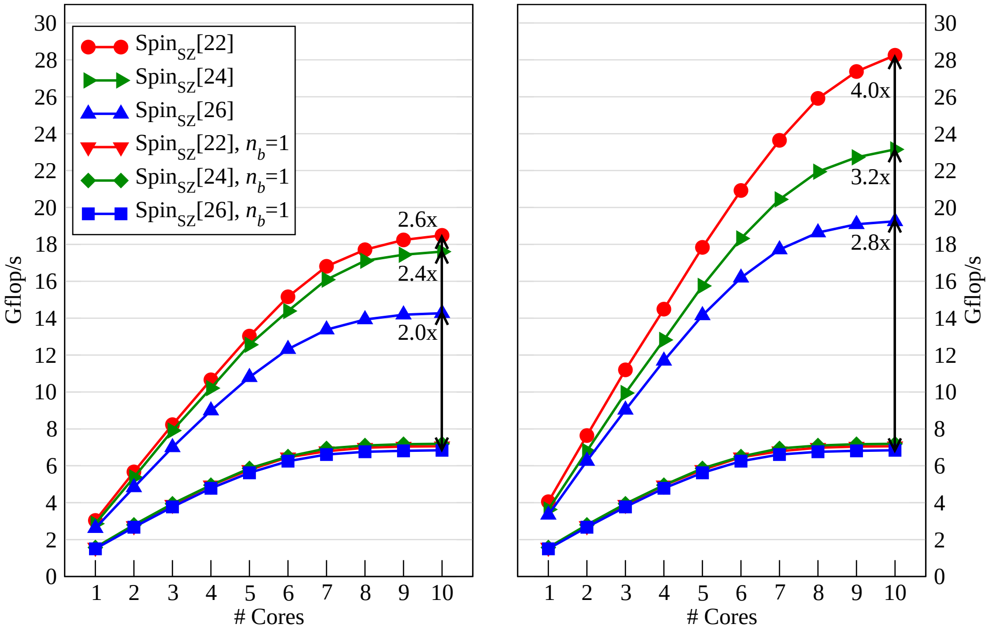
Benchmarking setup and results. The measurements in this paper have been conducted on a single socket Intel Xeon CPU E5-2660 v2 ("Ivy Bridge") running at 2.20 GHz. This processor comes with 10 cores on which SMT has been disabled. It features 64 GB of DDR3-1600 main memory and 25 MB of L3 cache. The maximum memory bandwidth adds up to values between 41 GB/s for the STREAM [17] Triad and 47 GB/s for a purely load-dominated micro-benchmark. Our implementation uses OpenMP for shared memory parallelization and all results are obtained with the Intel compiler version 13.1. In the case $n_b = 1$, the matrix is stored in SELL-32-2048 and `"DYNAMIC,250"` OpenMP scheduling is applied. Otherwise, `"DYNAMIC,1000"` OpenMP scheduling is used together with SELL-4-8.

Reducing Ω towards one is the key to achieving optimal spM(M)VM performance. For a qualitative estimate of Ω the matrix bandwidth ω ($1 \leq \omega \leq 2n + 1$) plays an important role. It is defined as the width of the diagonal band which contains all of the matrix' non-zero entries. Clearly, a small value of ω is favorable in order to have a potentially local access to the block vector X. The bandwidth of a matrix can be reduced by row reordering which is also applied here.

L	$\omega/$	$n_b = 1$			$n_b = 4$			$n_b = 8$		
	1e3	V_{min}	V_{meas}	Ω	V_{min}	V_{meas}	Ω	V_{min}	V_{meas}	Ω
22	76	0.117	0.120	1.02	0.151	0.173	1.15	0.196	0.220	1.12
24	255	0.472	0.484	1.03	0.602	0.741	1.23	0.774	1.098	1.42
26	869	1.979	2.141	1.08	2.478	3.820	1.54	3.143	5.629	1.79

Table 1: Test case properties and data traffic measurements for the matrices $\mathsf{Spin}_{\mathsf{SZ}}[L]$. Data traffic volumes V are given in GB.

Table 1 shows relevant information and data traffic measurement results for three test cases. More details can be found in Table 2 in Section 5. The traffic overhead Ω increases due to the limited cache size as the problem size n or the vector block size



(a) $n_b = 4$

(b) $n_b = 8$

Figure 1: Scaling performance for different block and problem sizes.

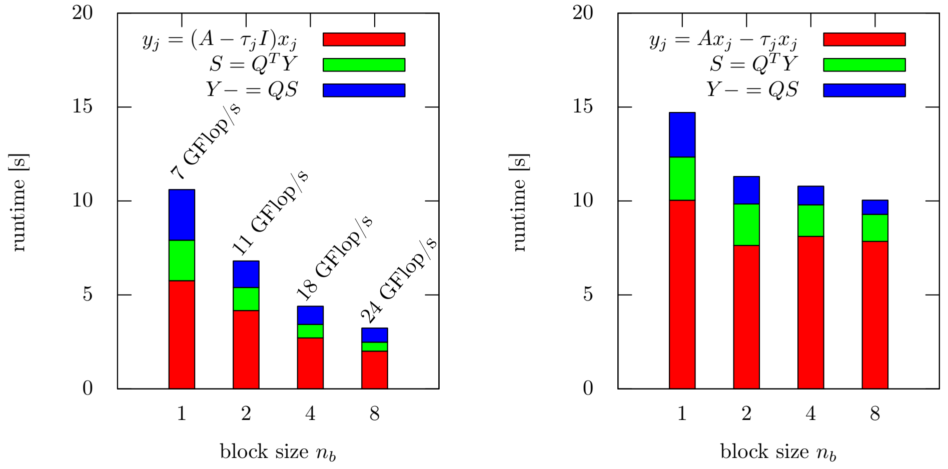
n_b are increased. The row-major storage of the vector blocks helps to keep Ω close to one as compared to column-major storage (see also Figure 2).

Figure 1 shows the intra-socket scaling performance and the speedup through blocking for the same set of test matrices and block sizes $n_b = 4$ and $n_b = 8$. A first observation is that the speedup achieved by using block operations decreases as the problem size increases. On the other hand, higher speedups can be expected for larger values of n_b. These observations can be related to the measured traffic overhead Ω in Table 1. The discrepancy between the actual speedup S and the maximum speedup S^* from (3.6) is rooted in Ω. As an example, we consider $\mathsf{Spin}_{\mathsf{SZ}}[22]$ ($n = 7 \cdot 10^5$ and $n_{nzr} = 12.57$, cf. Table 2) for $n_b = 4$:

$$S^* = n_b \cdot \frac{6n_{nzr} + 8}{6n_{nzr} + 8n_b} = 4 \cdot \frac{6 \cdot 12.57 + 8}{6 \cdot 12.57 + 8 \cdot 4} \approx 3.1 \tag{3.7}$$

However, the actual speedup adds up to $S \approx 2.6$ and $S^*/S \approx 1.19 \approx \Omega\,(= 1.15)$. The rather moderate values of Ω for $\mathsf{Spin}_{\mathsf{SZ}}[22]$ for all values of n_b can be explained by the fact that the outermost cache (25 MB) can easily accommodate n_b vector chunks of length ω. This is no longer true for $\mathsf{Spin}_{\mathsf{SZ}}[26]$ for $n_b = 4$ and higher, resulting in a considerable increase in Ω. Finally, the properties of a block algorithm may limit the meaningful block size by introducing more iterations and memory overhead with increasing n_b (cf. Section 5).

Implementation and performance of the projection operator. Another major contributor to the runtime is the projection $Y \leftarrow (I - QQ^T)X$, carried out in each inner iteration of JD. It consists of two dense matrix-matrix multiplications



(a) SELL-C-σ format, row-major vector blocks

(b) CRS format, column-major (Epetra)

Figure 2: Required runtime for $120/n_b$ applications of the Jacobi-Davidson operator ($y_j \leftarrow (I - QQ^T)(A - \tau_j I)x_j$) with shifts $\tau \in \mathbb{R}$ and 8 projection vectors $(q_1, \ldots, q_8) = Q$ for different block sizes n_b using the bandwidth-reduced $\mathsf{Spin}_{\mathsf{SZ}}$[26] matrix. The Trilinos [11] package Epetra (MPI only) achieves lower overall performance as the vectors x_j are copied before the spMMVM and the subtraction of $\tau_j x_j$ is done separately.

(GEMM operations) and a vector update. The matrices Q and X are very tall and skinny, so that the GEMM operation is memory-bound (as opposed to the case of square matrices where it is typically compute-bound if implemented correctly). We can therefore again apply the roofline model from (3.5) for performance predictions.

Using high-performance BLAS implementations typically does not yield satisfactory performance for the case of tall skinny matrices. We therefore implement them by hand and provide specialized kernels for useful values of n_b in block Jacobi-Davidson (e.g. 1, 2, 4 and 8). Due to the simple nature of these operations, we can easily match the maximum performance predicted by the model.

Figure 2a shows the contribution of the different kernels to the overall runtime of the Jacobi-Davidson operator. Due to the characteristics of the (blocked) vector-vector operations, the speedup due to blocking of the entire Jacobi-Davidson operator is larger than the speedup only for the spMMVM. Taking into account the additional flops of the projection, the total flop rate of the Jacobi-Davidson operator for $n_b = 8$ and the $\mathsf{Spin}_{\mathsf{SZ}}$[26] matrix adds up to 23.9 Gflop/s compared to 19 Gflop/s (cf. Figure 1) for the spMMVM operation alone. The CPU socket has a peak performance of 176 Gflop/s, of which we achieve 13.6%, which is quite satisfying for such a sparse matrix. As we will see in the next section, we can implement the block Jacobi-Davidson method such that it exploits these fast kernels as much as possible. The tall skinny matrix operations discussed here are also used in block orthogonalization steps using iterated classical Gram-Schmidt (in combination with e.g. TSQR, cf. Section 4.2).
In contrast, no block speedup for the spMMVM is achieved in Figure 2b beyond $n_b = 2$ due to the column-major storage of the blocks of vectors X and Y.

4. Algorithmic choices.

In this section we will discuss some aspects of our implementation that are different from what is common practice. The first point concerns the simultaneous solution of several linear systems with l shifted matrices $(A - \tilde{\lambda}_j I), j = 1 \dots l$ using a Krylov subspace method. This is required for solving the correction equations in block JD. We chose GMRES here because of its general applicability (see for example [19]), but the ideas could be transfered to any other Krylov subspace method. The second aspect we investigate is how to formulate the algorithm to reduce the number of global synchronization points and the total amount of communication compared to textbook implementations of JDQR.

4.1. Pipelining for the correction equations.

Our pipelined GMRES (PGMRES) solver allows the concurrent solution of l_{pgmres} independent linear systems of the form (2.10) using a standard GMRES method, but grouping together similar operations across the systems.

The block size l_{mach} that would deliver the optimal performance on the given machine is not always the best from a numerical point of view. For instance, the number of vectors in a spMMVM should be chosen based on the sparsity pattern and hardware characteristics such as the cache line length or the network bandwidth (cf. Section 3 and [16]), whereas the Jacobi-Davidson block size l_{bjdqr} might be chosen to contain the largest multiplicity of the eigenvalues encountered. It is reasonable to choose $l_{pgmres} = l_{mach} \leq l_{bjdqr}$.

A system that has converged (to its individual tolerance) is replaced by another until the number of unconverged systems is smaller than l_{mach}. At this point the iteration is stopped for all systems, or l_{pgmres} is reduced gradually until all systems have converged. In our experience the former approach gives better overall performance while not affecting the robustness of BJDQR, and is therefore chosen in our experiments in Section 5.

A single (unpreconditioned) GMRES iteration consists of the following steps (cf. [19, Section 6.5] for the complete algorithm):

```
1: apply operator to preceding basis vector ($\tilde{v}_{k+1} \leftarrow (I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_j)v_k$),
2: orthogonalize $\tilde{v}_{k+1}$ w.r.t. all previous basis vectors,
3: local operations (compute/apply Givens rotations, check residual).
```

Step 1 is always performed on l_{pgmres} contiguously stored vectors at a time. Step 2 is implemented using a modified Gram-Schmidt (MGS) method. Similar to the spMVM, the vector operations required are combined. If the shortest (longest) Krylov sequence among the l_{pgmres} systems currently iterated is m_{min} (m_{max}), we can perform m_{min} MGS steps with full blocks, and then $m_{max} - m_{min}$ operations with single vectors or parts of blocks (if $l_{bjdqr} = l_{pgmres}$ and the iteration is stopped as soon as a system converges, only full blocks are used).

The basis vectors of the individual systems j are stored as column j of block vectors in a ring buffer. Figure 3 illustrates a partly filled buffer for 4 systems. If all blocks are filled for a particular solver, it is restarted. The next block vector to be used is selected periodically.

Block GMRES. One could use a block Krylov method that constructs a single Krylov space for all systems (the shifts $\tilde{\lambda}_j$ do not change the Krylov space for a given starting vector). However, such an approach has the restriction $l_{pgmres} = l_{bjdqr}$ and therefore does not offer the full flexibility presented here.

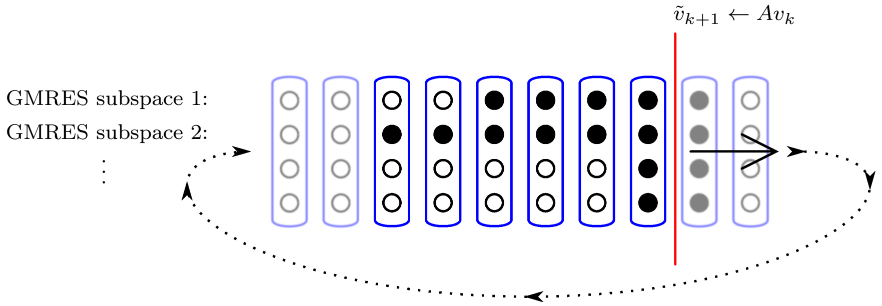


Figure 3: PGMRES ring buffer for the Krylov subspaces of 4 systems (rows) with current subspace dimensions (black dots) $k_1 = 4, k_2 = 6, k_3 = k_4 = 1$.

Preconditioning. In this paper we do not employ additional preconditioning for the linear systems, but the common practice of preconditioning based on a nearby positive definite matrix could be readily implemented here. Popular preconditioning techniques such as incomplete factorization or multigrid methods should benefit from similar performance gains due to blocking as the spMVM in this paper.

4.2. Avoiding communication. With the increasing number of nodes (and cores per node) of HPC systems, it becomes inevitable to ask the question if we can reduce the amount of communication a parallel algorithm requires. We want to briefly address this issue for the BJDQR method as well, although it is not the central topic of this work. Obviously the blocking of operations in BJDQR already reduces the number of data transfers compared to its single vector counterpart.

Block orthogonalization. An accurate method to create an orthogonal basis for the subspace $\mathcal{W}$ is crucial for the convergence of the Jacobi-Davidson algorithm. The efficient (and accurate) parallel orthogonalization of a block of vectors T with respect to previously calculated basis vectors W is considerably more challenging than just orthogonalizing one vector after another. Standard methods for the latter are the iterated classical Gram-Schmidt (ICGS) and the (iterated) modified Gram-Schmidt (IMGS) algorithms. Working with a complete block T, however, allows using faster BLAS3 operations. One problem here lies in the fact that when we first orthogonalize the columns of T internally and then against W, the second step may reduce the accuracy of the first, and vice versa.

In [8, 13] a new algorithm to orthogonalize a small block of vectors (TSQR) is described. It uses Householder-transformations of subblocks with reductions on arbitrary tree structures to both optimize cache usage for intra-node performance and communication between nodes. In combination with a rank revealing technique and block ICGS to orthogonalize the new block T against W one obtains a very fast and robust method (see the discussion about RR-TSQR-BGS in [13]). We have seen very good results with this method in BJDQR, but as the current TSQR implementation in Trilinos 11.6 [11] does not support row-major storage of the input block vector, we use IMGS for the internal orthogonalization of T in this paper.

Locking vs. deflation. A deflation approach for Jacobi-Davidson explicitly orthogonalizes the residual w.r.t. already converged Schur vectors Q, $\tilde{r} = (I - QQ^*)\tilde{r}$, in every iteration. Additionally, explicit orthogonalization is required whenever an eigenvalue converges. We can also achieve orthogonality by keeping the converged vectors in the search space (through the Galerkin condition). Obviously, we still need to orthogonalize the new corrections t with respect to Q, but this is now part of the regular orthogonalization step. As locking Schur vectors of converged eigenvalues improves the stability of the method for multiple or tightly clustered eigenvalues (see [22] and the references therein), it is important to transform the search space such that the locked Schur vectors are listed as the first basis vectors in W. This allows us to lock k_{conv} eigenvalues and corresponding Schur vectors in the left part of the projected Schur decomposition $HQ^H = Q^H R^H$ with

$$Q^H = \begin{pmatrix} I & 0 \\ 0 & q^H \end{pmatrix} ,$$

$$R^H = \begin{pmatrix} R_{1:k_{conv},1:k_{conv}} & H_{1:k_{conv},k_{conv}:k}\, q^H \\ 0 & r^H \end{pmatrix} \qquad \text{and} \tag{4.1}$$

$$H_{k_{conv}:k,k_{conv}:k}\, q^H = q^H r^H \ . \tag{4.2}$$

When the search space grows too large, we shrink it to a fixed size j_{min}. This operation does not require communication, and as long as $j_{min} \geq k_{conv}$ all converged eigenvectors remain locked. Locking is a standard technique in the field of subspace accelerated eigensolvers [20, Chapter 5], but we show this specific formulation to illustrate how the additional communication required for the deflation can be avoided.

5. Numerical and performance studies. In this section, we first want to check our implementation against an existing code. We then test the block JDQR method with different symmetric and non-symmetric real eigenvalue problems, which are summarized in Table 2. Finally, we show some results for larger matrices on up to 64 nodes (1280 cores) of a state-of-the-art cluster consisting of dual socket Intel Ivy Bridge nodes (cf. Section 3 for architectural details). The sparse matrix format used in this section is CRS for all experiments.

name	number of rows	non-zero count	eigenvalues sought	properties
Andrews	$6.0 \cdot 10^4$	$7.6 \cdot 10^5$	smallest	spd
cfd1	$7.1 \cdot 10^4$	$1.8 \cdot 10^6$	"	"
finan512	$7.5 \cdot 10^4$	$6.0 \cdot 10^5$	"	"
torsion1	$1.0 \cdot 10^4$	$2.0 \cdot 10^5$	"	"
ck656	656	3 884	rightmost	non-symm.
cry10000	$1.0 \cdot 10^4$	$5.0 \cdot 10^4$	"	"
dw8192	8 192	$4.2 \cdot 10^4$	"	"
rdb3200l	3 200	$1.9 \cdot 10^4$	"	"
$\mathsf{Spin}_{\mathsf{SZ}}[22]$	$7.0 \cdot 10^5$	$8.8 \cdot 10^6$	leftmost	symm.
$\mathsf{Spin}_{\mathsf{SZ}}[24]$	$2.7 \cdot 10^6$	$3.6 \cdot 10^7$	"	"
$\mathsf{Spin}_{\mathsf{SZ}}[26]$	$1.0 \cdot 10^7$	$1.5 \cdot 10^8$	"	"
$\mathsf{Spin}_{\mathsf{SZ}}[28]$	$4.0 \cdot 10^7$	$6.1 \cdot 10^8$	"	"

Table 2: Overview of the matrices used in the experiments. The symmetric positive definite (spd) matrices come from the University of Florida Sparse Matrix Collection [7], the non-symmetric matrices are from the Matrix Market [4], and the $\mathsf{Spin}_{\mathsf{SZ}}[L]$ matrices are used here as a scalable example from quantum physics (see text).

Spin chain test matrices. As an example for a typical matrix from quantum physics we use the matrices $\mathsf{Spin}_{\mathsf{SZ}}[L]$ that give the Hamilton operator for the Heisenberg XXZ spin chain model with S_z-symmetry (see e.g. [2] for a physical introduction). For a chain with L spins, in the case of zero total magnetization ($\langle S_z \rangle = 0$) and without translational symmetry, the matrix $\mathsf{Spin}_{\mathsf{SZ}}[L]$ is a symmetric matrix whose dimension $N_L = \binom{L}{L/2}$ grows exponentially with L. It contains between $2 \dots L$ (open boundary conditions) or $3 \dots L+1$ (periodic boundary conditions) non-zeroes per row, with about $L/2$ non-zeroes on average. The sparsity pattern is characterized by many thin (one-element wide) outlying diagonals, such that the band width is of the order $N_L/2$. Consequently, a good matrix reordering strategy is required to reduce the communication overhead and achieve reasonable performance on distributed memory machines. For moderately sized matrices ($N_L \lesssim 10^8$) the (serial) reverse Cuthill-McKee (RCM) algorithm [6] can be used, for larger matrices parallel strategies are required [5, 14]. Here, a RCM reordering was used for single-node calculations and ParMETIS for the inter-node tests.

Comparison with another implementation. We would like to compare our results to a state-of-the-art implementation of Jacobi-Davidson. Here we use the PRIMME software, [25], with sparse matrix-vector products (and spMMVM) provided by the Trilinos library Epetra [11]. We run PRIMME/Epetra using MPI on all 20 cores of a node of the above mentioned cluster for block sizes 1 to 8. Our software PHIST (Pipelined Hybrid-parallel Iterative Solver Toolkit) is executed on the same machine using OpenMP for the intra-node parallelism. We also compare two ways of stopping the inner QMR iterations in PRIMME: (a) limiting the number of iterations to 8 or stopping if a decreasing inner tolerance is reached (comparable to our own implementation), and (b) using an adaptive inner tolerance or stopping when the

method	n_b	matvecs	walltime [s]	time/spMVM [ms]	block speedup
PRIMME (a)	1	1374	183	49.7* (37.3%)	1.0
	2	1569	203	50.6 (39.1%)	0.90
	4	1899	236	49.9 (40.2%)	0.78
	8	2323	286	49.7 (40.4%)	0.64
PRIMME (b)	1	1377	180	49.6 (37.9%)	1.0
	2	1553	202	50.8 (39.0%)	0.89
	4	1680	210	50.2 (40.1%)	0.86
	8	1989	260	50.3 (38.4%)	0.69
PHIST	1	1426	228	39.2** (25%)	1.0
	2	≤1591	174	22.1 (20%)	1.31
	4	≤1887	172	15.2 (17%)	1.33
	8	≤2463	219	11.9 (13%)	1.04

Table 3: Comparison of our code PHIST with the PRIMME software. The column 'time/spMVM' shows the average time per single matrix-vector product and the contribution to the overall runtime in percent. The different configurations (a) and (b) are explained in the text.
* As explained in Figure 2 copying data slows down the spMVM in Epetra.
** Times for $y_j = (A - \sigma_j I)x_j$ were measured for PHIST.

eigenvalue residual has been reduced by one order of magnitude (see also [23,24]). The test matrix is $\mathsf{Spin}_{\mathsf{SZ}}$[26] (cf. Table 2) with RCM reordering. We seek 20 eigenpairs at the lower end of the spectrum, the required accuracy is 10^{-8}, and the methods are restarted from 30 vectors if a basis size of 50 is reached (32-64 vectors for $n_b = 8$). PHIST performs 30 single-vector Arnoldi steps to construct the initial basis.

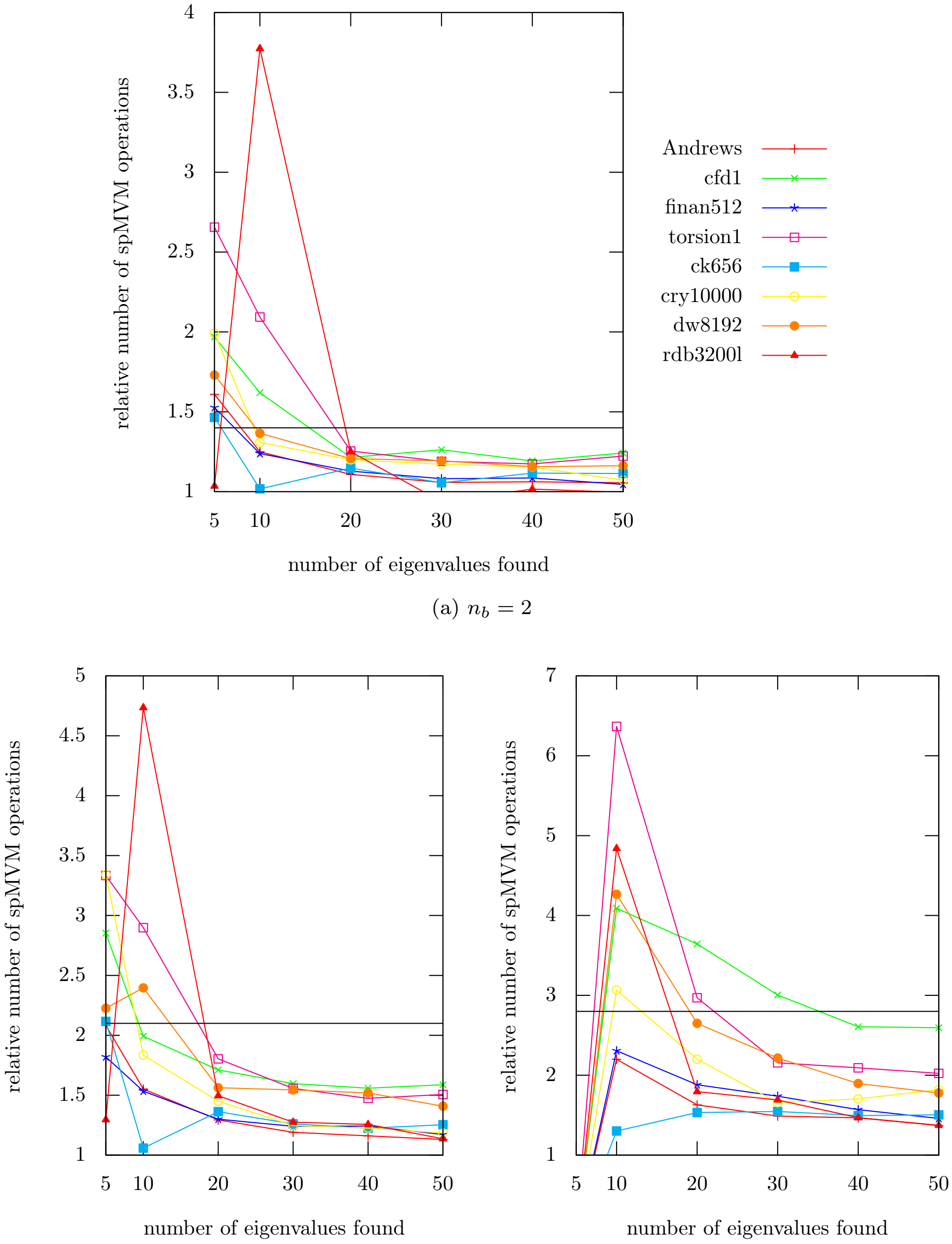
The results are shown in Table 3. PRIMME is somewhat faster than our code in the single-vector case because it requires slightly fewer matrix-vector products and is specialized for the case of symmetric matrices, which saves some vector-vector operations (for instance, a short recurrence in the inner iterations is exploited). The more sophisticated inner stopping criterion can further accelerate the computations, especially in the block variant (cf. the results for 'PRIMME (b)').

PRIMME becomes substantially slower when using the block algorithm and in our experiments only about 15% of the matrix-vector products performed are actually block operations. In contrast, our code becomes about 25% faster due to the techniques discussed in this paper, in particular due to the massive performance gain of the spMMVM shown in Section 3. With block size $n_b = 4$, we can match the overall runtime of PRIMME/Epetra for $n_b = 1$, even though significantly more spMVMs are performed and a method for unsymmetric matrices is used.

The advantages of PRIMME (exploiting symmetry and an improved inner stopping criterion) are obviously complementary to the performance advantages of PHIST (runtime reduction by blocking). When comparing the PHIST results among each other, we see that for an increasing block size the contribution of the spMMVM decreases and other operations start to dominate the overall runtime. This can be explained by the lack of an efficient block orthogonalization (like TSQR, cf. Section 4.2) in our current implementation. Note that in particular the performance of the block spMVM is significantly superior to the performance of the (block) spMVM implementation in PRIMME/Epetra.

Numerical behavior. To get an idea of the increase in the number of operations due to blocking, we compare the number of spMVM needed to calculate a given number of eigenvalues with different block sizes. Figure 4 shows the relative number of spMVMs compared to the single vector method. As long as more than about 20 eigenpairs are sought, the increase in the number of spMVMs is roughly constant, and for all test matrices it is below the typical speedup for the spMMVM we observed in Section 3 (indicated by the horizontal black line in Figure 4). Points below the horizontal line promise a distinct gain in wall clock time in favor of the block method. Thus, we can expect to improve the performance of the Jacobi-Davidson method by blocking for a wide range of matrices, symmetric or unsymmetric.

Next, we investigate the scalability of our code beyond a single node for two of the spin chain matrices. The implementation exploits MPI for the communication between the nodes (cf. also [21]). Figure 5 shows that on up to 64 nodes, blocking reduces the overall runtime for block sizes 2 and 4. Using a block method has two counteracting effects here: On one hand, the total communication volume increases with the number of matrix-vector multiplications; on the other hand, the individual messages become larger through message aggregation. So the pure communication time can increase as well as decrease depending on the sparsity pattern of the matrix and its distribution among the nodes, which explains the deviations from the single-node case.

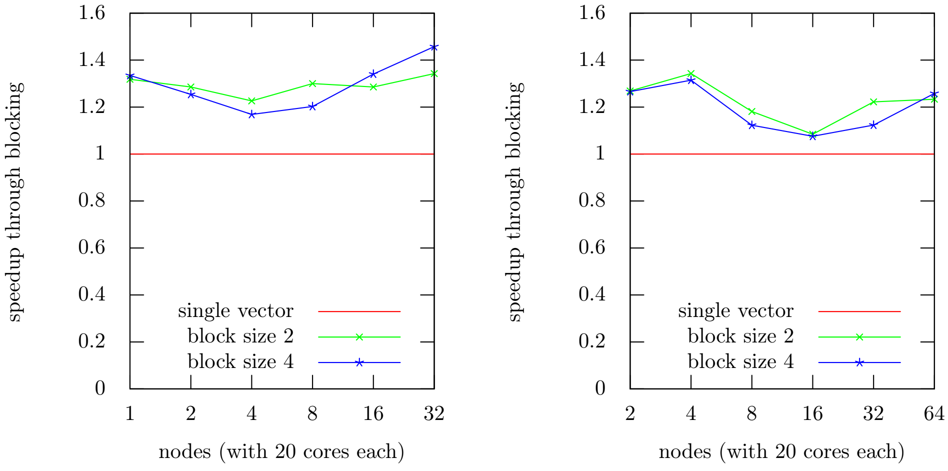


(a) $n_b = 2$

(b) $n_b = 4$

(c) $n_b = 8$

Figure 4: Influence of the block size n_b on the required number of spMVMs for different matrices. The relative increase compared to the single vector computation is shown. The horizontal line indicates the increase in single-socket performance measured for the comparatively large prototype matrix $\mathsf{Spin}_{\mathsf{SZ}}[26]$ in Figure 1.

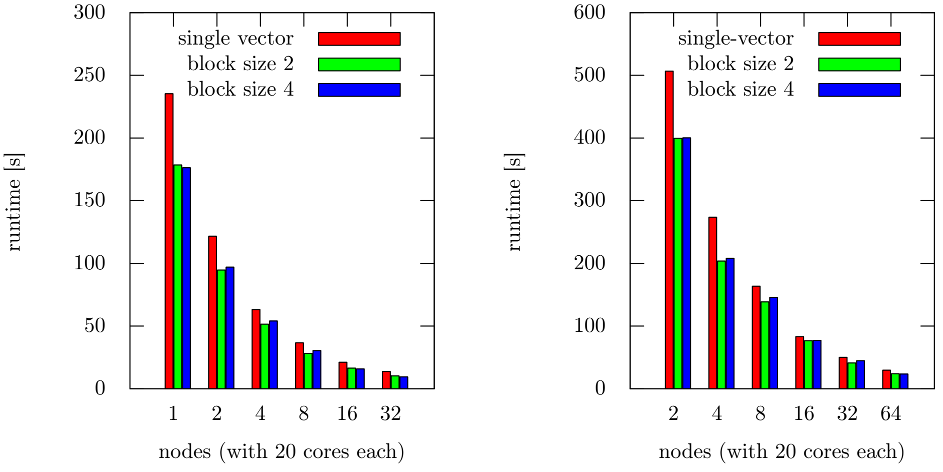


(a) Spin$_{SZ}$[26]

(b) Spin$_{SZ}$[28]

Figure 5: Relative performance gains through blocking for the computation of 20 exterior eigenvalues of two spin chain matrices on an Intel Ivy Bridge cluster.

Figure 6 shows the runtime reduction when increasing the number of nodes used. A significant parallel speedup is achieved in this strong scaling experiment, but on larger numbers of nodes the matrix-vector multiplications start to dominate the overall runtime (about 50% on 32 nodes for both test cases) as they require more and more communication.



(a) Spin$_{SZ}$[26]

(b) Spin$_{SZ}$[28]

Figure 6: Required runtime for the computation of 20 exterior eigenvalues of two spin chain matrices on an Intel Ivy Bridge cluster.

Summary and conclusions. We have derived a block formulation of the Jacobi-Davidson method for general (non-symmetric) eigenvalue problems. The key operation in this method, which is executed many times in the inner loop, consists of a sparse matrix-vector product followed by an orthogonal projection. By performance engineering and benchmarking we have demonstrated that applying this operation to blocks of vectors, as in our proposed algorithm, has significant performance advantages over the single vector case. An important implementation detail is the row-wise storage of blocks of vectors. This design choice, which is hardly ever found in implementations of block algorithms, is the key to achieving the speedup we have shown for the sparse matrix-vector products. We then showed some ways to achieve optimal blocking of operations by pipelining the operations in separate inner solves and discussed some ways to reduce the total amount of communication.

Our numerical results in the final section indicate that the assumptions made in deriving the block method are justified: the method works well for a wide range of matrices, both symmetric and non-symmetric. The performance results show that the hybrid parallel approach we take (MPI+OpenMP) gives good scalability on a modern cluster, and that the block variant outperforms its single-vector counterpart even for fairly large problems on up to 1280 cores. The direct comparison of eigensolvers and implementations is difficult because there are so many aspects that determine the overall runtime. However, all our numerical results confirm that blocking can significantly reduce the time to solution if implemented correctly.

Future work will include algorithmic improvements such as a specialized solver for Hermitian matrices, an implementation of the TSQR orthogonalization for block vectors in row-major storage, and an improved stopping criterion for the inner iterations. We also plan to investigate how to hide the communication of the matrix-vector multiplications behind other operations, and to allow using accelerator hardware such as GPUs.

REFERENCES

[1] P.-A. Absil, R. Mahony, R. Sepulchre, and P. Van Dooren, *A Grassmann–Rayleigh quotient iteration for computing invariant subspaces*, SIAM Review, 44 (2002), pp. 57–73.

[2] A. Auerbach, *Interacting Electrons and Quantum Magnetism*, Graduate Texts in Contemporary Physics, Springer New York, 1994.

[3] F. L. Bauer, *Das Verfahren der Treppeniteration und verwandte Verfahren zur Lösung algebraischer Eigenwertprobleme*, Zeitschrift für angewandte Mathematik und Physik ZAMP, 8 (1957), pp. 214–235.

[4] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, *Matrix market: A web resource for test matrix collections*, in The Quality of Numerical Software: Assessment and Enhancement, Chapman & Hall, 1997, pp. 125–137.

[5] C. Chevalier and F. Pellegrini, *PT-Scotch: A tool for efficient parallel graph ordering*, Parallel Comput., 34 (2008), pp. 318–331.

[6] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 1969 24th National Conference, ACM '69, New York, NY, USA, 1969, ACM, pp. 157–172.

[7] T. A. Davis and Y. Hu, *The university of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1–25.

[8] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM Journal on Scientific Computing, 34 (2012), pp. A206–A239.

[9] D. R. Fokkema, G. L. G. Sleijpen, and H. A. van der Vorst, *Jacobi–Davidson style QR and QZ algorithms for the reduction of matrix pencils*, SIAM Journal on Scientific Computing, 20 (1998), pp. 94–125.

[10] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers (Chapman & all/CRC Computational Science)*, CRC Press, 2010.

[11] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the Trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.

[12] M. E. Hochstenbach and Y. Notay, *The Jacobi-Davidson method*, GAMM-Mitteilungen, 29 (2006), pp. 368–382.

[13] M. Hoemmen, *Communication-avoiding Krylov subspace methods*, PhD thesis, University of California, Berkeley, Apr. 2010.

[14] G. Karypis and K. Schloegel, *ParMETIS. Parallel Graph Partitioning and Sparse Matrix Ordering Library*, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, 4.0 ed., 2013.

[15] A. Klinvex, F. Saied, and A. Sameh, *Parallel implementations of the trace minimization scheme TraceMIN for the sparse symmetric eigenvalue problem*, Computers & Mathematics with Applications, 65 (2013), pp. 460–468.

[16] M. Kreutzer, G. Hager, G. Wellein, G. Fehske, and A. R. Bishop, *A unified sparse matrix data format for modern processors with wide SIMD units*, ArXiv e-prints, (2013).

[17] J. D. McCalpin, *STREAM: Sustainable memory bandwidth in high performance computers*, tech. report, University of Virginia, Charlottesville, VA, 1991-2007. A continually updated technical report.

[18] Y. Notay, *Convergence analysis of inexact Rayleigh quotient iteration*, SIAM Journal on Matrix Analysis and Applications, 24 (2003), pp. 627–644.

[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, 2nd ed., 2003.

[20] ———, *Numerical Methods for Large Eigenvalue Problems*, Classics in Applied Mathematics, SIAM, revised ed., Jan 2011.

[21] G. Schubert, G. Hager, H. Fehske, and G. Wellein, *Parallel Sparse Matrix-Vector Multiplication as a Test Case for Hybrid MPI+OpenMP Programming*, Institute of Electrical and Electronics Engineers, May 2011, pp. 1751–1758.

[22] A. Stathopoulos, *Locking issues for finding a large number of eigenvectors of Hermitian matrices*, Tech. Report WM-CS-2005-09, College of William and Mary, Department of Computer Science, Jul 2005.

[23] ———, *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part I: Seeking one eigenvalue*, SIAM Journal on Scientific Computing, 29 (2007), pp. 481–514.

[24] A. Stathopoulos and J. R. McCombs, *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part II: Seeking many eigenvalues*, SIAM Journal on Scientific Computing, 29 (2007), pp. 2162–2188.

[25] ———, *Primme: preconditioned iterative multimethod eigensolver–methods and software description*, ACM Trans. Math. Softw., 37 (2010), pp. 1–30.

[26] P. T. P. Tang and E. Polizzi, *Subspace iteration with approximate spectral projection*, ArXiv e-prints, (2013).

[27] J. Treibig, G. Hager, and G. Wellein, *Likwid: A lightweight performance-oriented tool suite for x86 multicore environments*, in Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW '10, Washington, DC, USA, 2010, IEEE Computer Society, pp. 207–216.

[28] S. W. Williams, A. Waterman, and D. A. Patterson, *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*, Tech. Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

[29] K. Wu, Y. Saad, and A. Stathopoulos, *Inexact Newton preconditioning techniques for large symmetric eigenvalue problems.*, Electronic Transactions on Numerical Analysis, 7 (1998), pp. 202–214.

[30] Y. Zhou, *Studies on Jacobi-Davidson, Rayleigh quotient iteration, inverse iteration, generalized Davidson and Newton updates*, Numerical Linear Algebra with Applications, 13 (2006), pp. 621–642.