SIAM J. SCI. COMPUT.
Vol. 37, No. 6, pp. C697–C722

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 when it comes to computing multiple or clustered eigenvalues. In practice, however, they are typically avoided because the total number of matrix-vector 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 through better cache usage on modern CPUs, resulting in a method that is both more efficient and robust than its single vector counterpart. The steps to be taken to achieve a block speedup involve both kernel optimizations for sparse matrix and block vector operations, and algorithmic choices to allow using blocked operations in most parts of the computation. We discuss the aspect of avoiding synchronization in the algorithm and show by numerical experiments with our hybrid parallel implementation that a significant speedup through blocking can be achieved for a variety of matrices on up to 5 120 CPU cores as long as at least about 20 eigenpairs are sought.

Key words. sparse eigenvalue problems, Jacobi–Davidson, block methods, performance engineering, high performance computing, multicore processors, hybrid parallel implementation

AMS subject classifications. 65F15, 65F50, 65Y05, 65Y20

DOI. 10.1137/140976017

1. Introduction.

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

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

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

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

with $B = B^T \in \mathbb{R}^{n\times n}$ symmetric 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).

[*]Submitted to the journal's Software and High-Performance Computing section July 7, 2014; accepted for publication (in revised form) September 29, 2015; published electronically December 15, 2015. This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) under project ESSEX.
http://www.siam.org/journals/sisc/37-6/97601.html

[†]German Aerospace Center (DLR), Simulation and Software Technology, Cologne 51147, Germany (Melven.Roehrig-Zoellner@DLR.de, Jonas.Thies@DLR.de, Achim.Basermann@DLR.de).

[‡]Erlangen Regional Computing Center, Friedrich-Alexander-University of Erlangen-Nuremberg, Erlangen 91058, Germany (moritz.kreutzer@fau.de, georg.hager@fau.de, gerhard.wellein@rrze.uni-erlangen.de).

[§]Institute of Physics, Ernst Moritz Arndt University of Greifswald, Greifswald 12487, Germany (Alvermann@physik.uni-greifswald.de, Pieper@physik.uni-greifswald.de, fehske@physik.uni-greifswald.de).

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 here is quantum mechanics, where A is a sparse matrix representation of the Hamiltonian in the Schrödinger equation. We choose a number of examples from solid state physics (see section 5), supplemented by some matrices from different application fields to show the generality of the results.

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. The novelty of this work is that block operations are typically not investigated in the context of a particular algorithm. We show that the Jacobi–Davidson (JD) method must be reviewed and implemented with optimal block usage in mind in order to exploit the performance gains of low-level optimizations and parallel synchronization avoidance. A topic which is not addressed in detail in this paper (but briefly touched on in section 4.1) is preconditioning of the inner iteration. While this is obviously an important issue for the effectiveness of JD methods, it is too broad and problem specific to be covered in the scope of this work.

Related work. In this paper we investigate a block JD method that performs matrix-vector products and vector-vector operations with several vectors at once. JD methods for the calculation of several eigenvalues were originally proposed in [17]. Since then, many authors have worked on the algorithm, its theoretical properties, and efficient implementation. For a review article, see [21]. Stathopoulos and McCombs investigated JD and generalized Davidson methods for symmetric (respectively, Hermitian) eigenvalue problems in [38] 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 [27] to the case of multiple vectors and a careful grouping 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.

Performance characteristics of the sparse matrix-multiple-vector multiplication (spMMVM), which plays a central role in this paper, were also discussed, though not in the context of a particular algorithm, in [24, 29]. Recently, the sparse matrix-vector multiplication with row-major vector blocks has been used in [2, 30]. While the idea is not new [18], it is typically not employed for more complex algorithms such as JD.

The performance analysis and algorithmic principles presented here are also useful for other algorithms, such as block Krylov methods or eigenvalue solvers that require the solution of linear systems with multiple right-hand sides such as FEAST [40] or TraceMin [26].

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 [19]. 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 multicore 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 main memory bandwidth is insufficient to keep the cores working with code that has a low ratio of floating-point operations to data transfers (which is frequently the case in sparse linear algebra). In some cases, the adverse effects of slow memory can be ameliorated by the cache hierarchy: 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. A typical cache line size on modern CPUs is 64 bytes.

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. This paper clearly focuses on the node-level optimization, though aspects of multinode performance, such as avoiding global synchronization, are addressed as well.

Document structure. In section 2 we derive a block formulation of the JD 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 a similar algorithm (PRIMME [39]).

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 [6]. 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 exterior 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. While this is not a new algorithm (a similar scheme is implemented, e.g., in the PRIMME software [39]), a thorough derivation is not found in the literature. On the other hand, the exact formulation of the method determines the extent to which the hardware performance of the implementation can benefit from the blocking. The formulation presented here allows using block operations as much as possible, which is crucial to avoid strided memory access if vector blocks are stored in row-major ordering (cf. section 3), and we also show clearly where terms are dropped in the nonsymmetric case in order to allow blocked (rather than sequential) solution of the correction equations for different shifts. In the appendix, an algorithm template summarizes the method described here.

The starting point for the derivation is the invariant subspace $\mathcal{V} = \operatorname{span}\{v_1, \ldots, v_l\}$ 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 [43] 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 BJD-style QR algorithm (see [17]). 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:

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

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:

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

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 standard JDQR method [17] uses the following correction equation for a single eigenvalue (with deflation of an already converged eigenbasis Q_k and $\bar{Q} = \begin{pmatrix}Q_k & \tilde{q}\end{pmatrix}$):

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

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 yet converged. 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, and 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

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

Inexact solution. If the single-vector JD correction equation is solved exactly in a subspace method, one recovers the directions of the Rayleigh quotient iteration (RQI) as discussed, for example, in [44]. Similarly the block correction equation (2.10) can be related to a block variant of the RQI, which converges cubically to an invariant subspace for the Hermitian case as shown in [1]. For the non-Hermitian case we still expect quadratic convergence to at least single eigenvalues (from the standard RQI).

In [32], Notay shows for a special single-vector 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 $(I - QQ^T)(A - \tilde{\lambda}I)$ 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 (with a different shift $\tilde{\lambda}_j$ for each vector in the block). 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 eigenvectors 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 contrast 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 potential performance gain from combining several BLAS level 1 or level 2 routines into specialized kernels has been observed by, amongst others, Baker, Dennis, and Jessup [5] for GMRES and Howell et al. [23] for Househoulder bidiagonalization. The relevance of this is also shown by the inclusion of so-called *BLAS* 2.5 functions in the updated BLAS standard [7].

In addition, the number of global synchronization points and messages sent for all key operations is reduced by using block vectors, although the amount of communicated data remains the same. Obviously, we can combine this approach with appropriate matrix reorderings (e.g., using reverse Cuthill–McKee (RCM) [12], PT-SCOTCH [11], or ParMETIS [25]) to further reduce the communication effort in the spMMVM. Another idea is to overlap communication and computation during the spMMVM. The price one has to pay is an additional write operation to the result vector, so that it is not immediately clear whether the overall performance will benefit or suffer [35]. We do not investigate this technique here, but note that it may be beneficial if the network is slow or the number of nodes used is very large.

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 nonzero entries) and the sparsity pattern (distribution of nonzero 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 [27] yields high performance on a much wider set of architectures for many matrices in practice. It contains two tuning parameters: the hardware-dependent chunk size C which is closely connected to the width of the single-instruction multiple-data (SIMD) vector registers, and the matrix-dependent sorting scope σ. All rows within a chunk will be stored columnwise, and chunks are stored one after another. All rows in a chunk are padded to have the length of the longest row. In order to avoid excessive zero padding (especially for irregular matrices), local sorting by row length within blocks of $\sigma >$ C rows can be applied, which leads to similarly sized rows in each chunk and less overhead. The sorting is applied to both rows and columns, so that symmetry and spectral properties of the matrix are preserved. We do note that it may have an impact on the implementation and effectiveness of preconditioning techniques for iterative linear solvers. This is briefly addressed at the end of section 4.1.

Extending the results in [27] 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. Row-major storage of the block vectors was also mentioned to be beneficial in [29] but was abandoned because of the inconvenience it imposes on the user. The GHOST library[1] (General Hybrid Optimized Sparse Toolkit) provides simple mechanisms for changing the data layout of a vector in-place or out-of-place to circumvent this problem.

Concerning the sparse matrix format, we find that for multicore CPUs, C may be reduced if $n_b > 1$; for instance, a reasonable choice was C $= 8$, $\sigma = 32$ for $n_b = 4$, in contrast to C $= 32$, $\sigma = 256$ for single-vector computations. In practice, for BJDQR n_b is comparatively small (sometimes only 2), so introducing the SELL format helps to increase the vectorization potential of the spMMVM. If n_b is equal to (or a multiple of) the SIMD width of the CPU, one can often choose C $= \sigma = 1$—which recovers the CRS format—and still get good performance. For accelerator hardware such as GPGPUs or the Intel(R) Xeon Phi, which have a larger SIMT/SIMD width, the SELL-C-σ format is preferred (cf. [27]).

Assuming 64-bit matrix/vector values and 32-bit indices, no overhead in the matrix storage (which can be achieved by a good choice of σ), and a large enough cache to hold X during the entire operation, the minimal data volume for a single spMMVM

[1]https://bitbucket.org/essex/ghost/

can be computed as (generalizing from, e.g., [18, 27])

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

where n_{nz} denotes the number of nonzero matrix entries. The minimum code balance of the spMMVM kernel for processing a single nonzero 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 nonzero entries per row. Here we employ nontemporal stores for the result vector Y, which eliminates the write-allocate data transfer from memory for this vector. 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 suboptimal).

More details on this topic can be found in, for example, [27, 35]. The actual transferred data volume $V_{\text{meas}} \geq V_{\min}$ can be measured with hardware performance monitoring tools like likwid-perfctr [41], 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 metric for the relative data traffic can be formulated as

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

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. [42] and the references therein) in order to predict the maximum performance:

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

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 are conducted on a single socket Intel(R) 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 32 GB of DDR3-1600 main memory and 25 MB of L3 cache. The maximum memory bandwidth is between 41 GB/s for the STREAM [31] Triad and 47 GB/s for a purely load-dominated microbenchmark. Our implementation uses OpenMP for shared memory parallelization, and all results are obtained with the Intel(R) compiler version 14.0. In the case $n_b = 1$, the matrix is stored in SELL-32-2048 and `"DYNAMIC,250"` OpenMP scheduling is applied to the outer loop over C-blocks (chunks). Otherwise, `"DYNAMIC,` `1000"` OpenMP scheduling is used together with SELL-4-8. The choice of `DYNAMIC` over `STATIC,0` scheduling (which is the default for the Intel(R) compiler in the version used) is motivated by possible load imbalance for static scheduling and irregular matrices. The relatively large block size together with the fact that the parallelization is done over SELL chunks instead of rows leads to having enough work for each OpenMP thread at a time and a negligible scheduling overhead. In this paper we only use OpenMP inside each uniform memory access (UMA) domain, e.g., a single CPU socket. In a NUMA (nonuniform memory access) setting, one MPI process per UMA domain is used to ensure local accesses to the matrix and vector elements.

Reducing Ω towards one is the key to achieving optimal spM(M)VM performance. For a qualitative estimate of Ω the matrix bandwidth ω plays an important role. It is defined as the width of the diagonal band which contains all of the nonzero entries of the matrix. 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.

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.*

L	$\omega/10^3$	$n_b = 1$ V_{min}	V_{meas}	Ω	$n_b = 4$ V_{min}	V_{meas}	Ω	$n_b = 8$ 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 shows relevant information and data traffic measurement results for three test cases. More details can be found in Table 3 in section 5. The traffic overhead Ω increases due to the limited cache size as the problem size n or the vector block size n_b is 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 intrasocket 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 3) 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 $Y \leftarrow (I - QQ^T)Y$, carried out

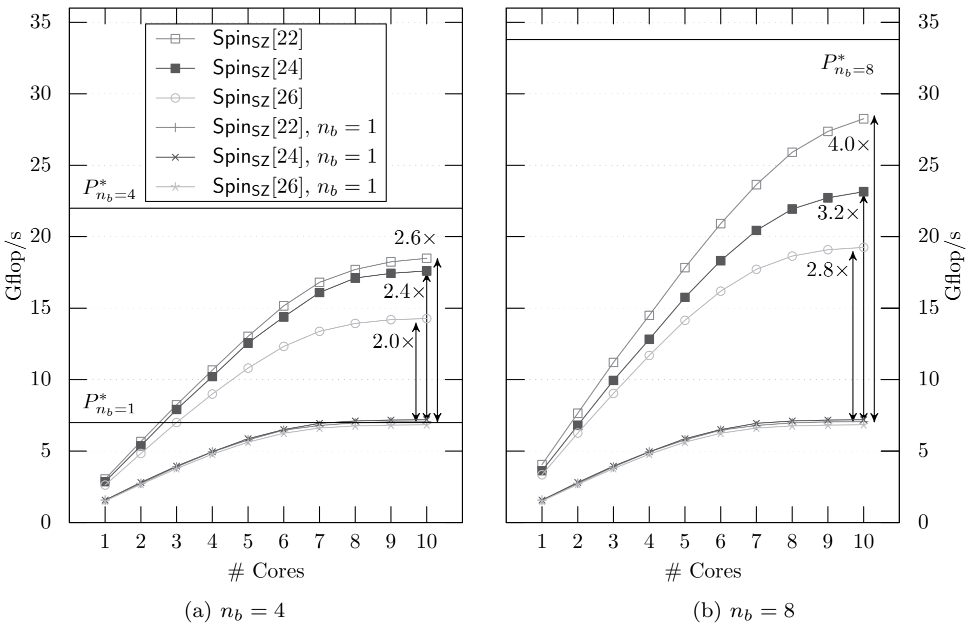


FIG. 1. *Scaling performance for different block and problem sizes on the Ivy Bridge test system. The upper performance bound P^* as computed with* (3.5) *for $n_b = 1$, $n_b = 4$, and $n_b = 8$ using the* Spin$_{SZ}$[$22 - 26$] *matrices is shown (assuming $\Omega = 1$ and $b = 47$ GB/s).*

in each inner iteration of JD. It consists of two dense matrix-matrix multiplications (GEMM operations) and a vector update. The matrices Q and Y 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 efficiently). Thus, we can 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 implemented them by hand and provided automatically generated kernels for useful values of n_b in BJD (e.g., $n_b = 1, 2, 4, 8$) which were selected at runtime. SIMD extensions SSE ($n_b = 2$) and AVX ($n_b = 4, 8$) were used for optimal floating-point performance. The operations we use in BJDQR are, in particular, $C \leftarrow V^T W$, $W \leftarrow V \cdot C$, and $V(:, 1:k) \leftarrow V \cdot C$, with $V \in \mathbb{R}^{n\times m}$, $W \in \mathbb{R}^{n\times k}$, and $C \in \mathbb{R}^{m\times k}$. The latter operation (with $k \le m$) occurs, e.g., when shrinking the subspace in step 29 of Algorithm 1. Due to the simple nature of these operations, we could easily match the measured performance to the maximum performance predicted by the model (see Table 2).

Figure 2a shows the contribution of the different kernels to the overall runtime of the JD operator. It is clear from our experiments that achieving a block speedup is owed to the choice of row-major storage for vector blocks, whereas the sparse matrix format (SELL vs. CRS) may improve the overall performance, depending on the matrix. Due to the characteristics of the (blocked) vector-vector operations, the speedup due to blocking of the entire JD 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 JD operator for $n_b = 8$ and the Spin$_{SZ}$[26] matrix adds up to 34.7 Gflop/s compared to 19 Gflop/s (cf. Figure 1) for the spMMVM operation alone.

TABLE 2

Predicted and measured performance in Gflop/s of the memory-bound GEMM operations for the projection $Y \leftarrow (I - QQ^T)Y$ on the 10-core Ivy Bridge system (assuming a peak bandwidth of $b = 47$ GB/s). We show results for $Q \in \mathbb{R}^{n\times 20}$ and $Y \in \mathbb{R}^{n\times n_b}$ with $n_b = 1$, $n_b = 2$, and $n_b = 4$. The number of rows n in the measurements is in the order of 10^7.

Operation	Code balance	$n_b = 1$		$n_b = 2$		$n_b = 4$	
		P^*	P_{meas}	P^*	P_{meas}	P^*	P_{meas}
$S \leftarrow Q^TY$	$\frac{8(20+n_b)}{2(20n_b)}$ $\frac{\text{bytes}}{\text{flop}}$	11.2	10.9	21.4	20.0	39.2	37.5
$Y \leftarrow Y - QS$	$\frac{8(20+2n_b)}{2(20n_b)}$ $\frac{\text{bytes}}{\text{flop}}$	10.7	10.3	19.6	18.2	33.6	30.1

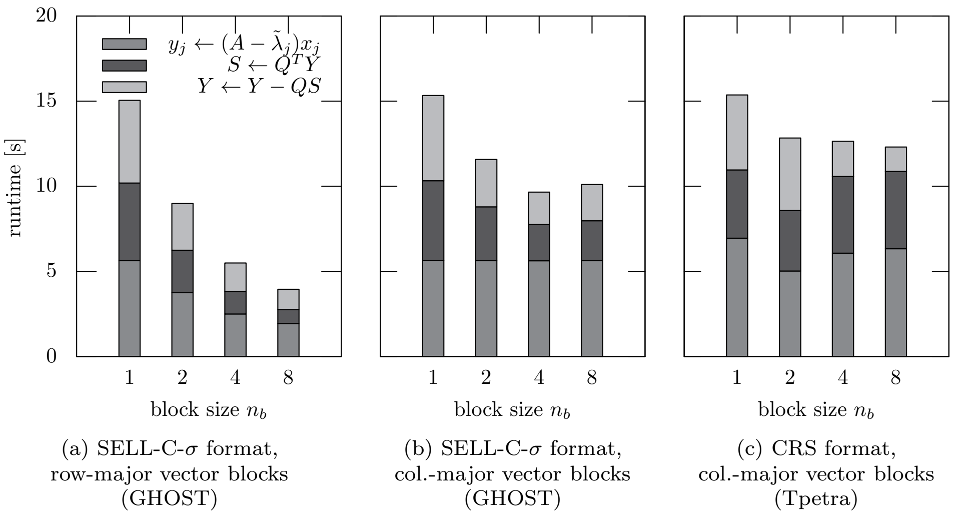


FIG. 2. *Required runtime on the 10-core Ivy Bridge CPU for $120/n_b$ applications of the JD operator ($y_j \leftarrow (I - QQ^T)(A - \tilde{\lambda}_jI)x_j$) with shifts $\tilde{\lambda} \in \mathbb{R}$ and 20 projection vectors $(q_1, \ldots, q_{20}) = Q$ for different block sizes n_b using the bandwidth-reduced* $\mathsf{Spin_{SZ}}$[26] *matrix. For column-major vector blocks SELL-32-2048 was used, whereas for the row-major case SELL-8-32 yields better performance for $n_b > 1$. In all cases dynamic OpenMP scheduling was employed with a chunk size of 1000. The Trilinos [20] package Tpetra does not offer a shifted spMMVM operation, such that the subtraction of $\tilde{\lambda}_jx_j$ is done separately. Shared-memory parallelization in Tpetra was provided by the Trilinos library ThreadPool.*

The CPU socket has a peak performance of 176 Gflop/s, of which we achieve 20%, which is quite satisfying. As we will see in the next section, we can implement the BJD 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 with the column-major storage for the blocks of vectors X and Y (see Figure 2b). For the Trilinos implementation (see Figure 2c), we observe a small block speedup for block size two but no further improvement for $n_b = 4$ and $n_b = 8$.

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 $(A - \tilde{\lambda}_jI)$, $j = 1, \ldots, l$, using a Krylov subspace method. This is required for solving the correction equations in BJD. We chose GMRES here because of its general applicability (see, for example, [33]), but the ideas could be transferred 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 compared to textbook implementations of JDQR.

4.1. Blocked solution of the correction equations. A blocked GMRES solver allows the concurrent solution of l_{gmres} independent linear systems of the form (2.10) using a standard GMRES method, but grouping together similar operations across the systems. For real symmetric (Hermitian) matrices, we use a MINRES variant which is simply obtained by replacing the long recurrence in GMRES by a short one, saving some orthogonalization effort.

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 an spMMVM should be chosen based on sparsity pattern and hardware characteristics such as the SIMD width or the network bandwidth (cf. section 3 and [27]), whereas the JD block size l_{bjdqr} might be chosen to contain the largest multiplicity of the eigenvalues encountered. It is reasonable to choose $l_{gmres} = l_{mach} \leq l_{bjdqr}$.

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_{gmres} 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 the experiments in section 5.

A single (unpreconditioned) GMRES iteration consists of the following steps (cf. [33, 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 $\tilde{v}_{k+1}$ with respect to all previous basis vectors;
3: local operations (compute/apply Givens rotations, check residual).
```

Step 1 is always performed on l_{gmres} 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_{gmres} 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 $l_{bjdqr} = l_{gmres}$ and the iteration is stopped as soon as a system converges, only full blocks are used). In contrast to the outer JD loop, MGS is preferred here because the basis vectors of a particular system are not stored contiguously, so that it is not possible to gain additional performance from block operations in classical Gram–Schmidt.

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 four 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). Such an approach has the restriction $l_{gmres} = l_{bjdqr}$ and therefore

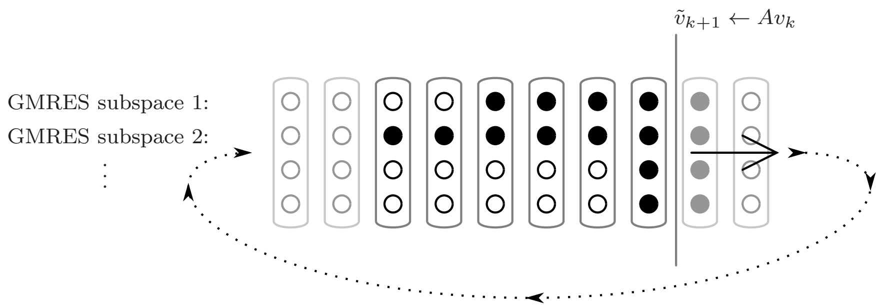


Fig. 3. *Visualization of the ring buffer used to store the Krylov subspace basis vectors of the blocked GMRES algorithm for four linear systems. The vectors of the different subspaces are grouped together in blocks (black dots in one column). The different subspaces can have different dimensions (number of filled black dots in one row, e.g.,* $k_1 = 4, k_2 = 6, k_3 = k_4 = 1$*). The blank circles indicate that some of the preallocated blocks of vectors are currently (partially) unused.*

does not offer the full flexibility presented here. The performance results in section 5, however, indicate that this flexibility is not crucial for achieving high performance with BJDQR.

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; i.e., one may use a fixed (or no) shift to compute the preconditioner and apply it to the individually shifted systems. The MINRES method we use for symmetric matrices requires a positive definite preconditioner, which immediately suggests this approach. Popular preconditioning techniques such as incomplete factorization [10] or multigrid methods [16] should benefit from performance gains due to blocking similar to that of the spMMVM in this paper. If a multigrid method for the shifted systems is used (as in [16]), one could choose a different shift per iterated system in smoothing operations like Jacobi or Gauß–Seidel.

The SELL-C-σ sparse matrix format (cf. section 3) used in this paper does not allow straightforward rowwise access to matrix entries, and the local sorting it employs may additionally complicate constructing an effective preconditioner. For matrices with a relatively simple structure (like the MATPDE example in section 5), most rows have the same length and sorting is only used in a limited area near the boundary (or σ can be set to 1 at the cost of some more zero-padding). If the desired preconditioner cannot be implemented in the SELL format, one can choose $C = \sigma = 1$, which recovers the well-known CRS format. This may incur a slight performance degradation of the spMMVM, but good block speedups can still be expected, and applying the preconditioner may be more costly than the spMMVM anyway.

For the quantum physics examples in this paper, it is an open research issue to find effective preconditioners, as they have little in common with the well-understood matrices stemming from discretized elliptic PDEs. Polynomial preconditioning is one option, as it can be implemented straightforwardly in SELL-C-σ and with different shifts per system solved simultaneously. An alternative is using model specific "physics-based" preconditioners as in [3].

4.2. Improving the communication behavior. With the increasing number of nodes (and cores per node) of HPC systems, it becomes inevitable to ask the question whether we can improve the communication behavior of a parallel algorithm.

The BJDQR method typically requires more matrix-vector and scalar products than single-vector JDQR, which leads to a higher total data volume to be transferred. However, the blocking can lead to a significant reduction in the number of messages sent and thus in the number of global synchronization points. It is clear that this is the case for the spMMVM operation, but for the orthogonalization with respect to the current basis and converged eigenvectors, algorithmic choices need to be made.

Block orthogonalization. An accurate method to create an orthogonal basis for the subspace $\mathcal{W}$ is crucial for the convergence of the JD 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 BLAS2.5 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 [14, 22] 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 optimize both cache usage for intranode 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 [22]). We use TSQR from the Trilinos 11.12 [20] library, with thread-level parallelism provided by Intel(R)TBB. As this implementation does not support row-major storage of the input block vector (yet), we change the memory layout on-the-fly before and after TSQR, which incurs a performance penalty. This is not further discussed in the following, as it influences the total runtime of the complete algorithm (cf. section 5) by less than 4%.

Locking vs. deflation. A deflation approach for JD explicitly orthogonalizes the residual with respect to 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 [36] 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

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

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

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

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 [34, Chapter 5], but we exploit 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 BJDQR method with different symmetric and nonsymmetric real eigenvalue problems, which are summarized in Table 3. Finally, we show some results for larger matrices on up to 256 nodes (5 120 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 SELL-32-256 for block size $n_b = 1$ and SELL-8-32 for $n_b > 1$.

Table 3

Overview of the matrices used in the experiments. All matrices are sparse and have real entries. The symmetric positive definite (spd) matrices come from the University of Florida Sparse Matrix Collection [13]*, the nonsymmetric (nonsymm.) matrices are from the Matrix Market* [9]*, and the matrices in the lower table are used as scalable examples, stemming from quantum physics and a generic PDE problem (see text).*

Name	n	n_{nz}	Eigenvalues sought	Symmetry
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	none
cry10000	$1.0 \cdot 10^4$	$5.0 \cdot 10^4$	”	”
dw8192	8 192	$4.2 \cdot 10^4$	”	”
rdb3200l	3 200	$1.9 \cdot 10^4$	”	”

(a) various small matrices

Name	n	n_{nz}	Eigenvalues sought	Symmetry
$\mathsf{Spin}_{\mathsf{SZ}}[22]$	$7.0 \cdot 10^5$	$8.8 \cdot 10^6$	leftmost	symm.
$\mathsf{Spin}_{\mathsf{SZ}}[24]$	$2.7 \cdot 10^6$	$3.6 \cdot 10^7$	”	”
$\mathsf{Spin}_{\mathsf{SZ}}[26]$	$1.0 \cdot 10^7$	$1.5 \cdot 10^8$	”	”
$\mathsf{Spin}_{\mathsf{SZ}}[28]$	$4.0 \cdot 10^7$	$6.1 \cdot 10^8$	”	”
tV[26]	$1.0 \cdot 10^7$	$1.5 \cdot 10^8$	”	”
tV[28]	$4.0 \cdot 10^7$	$6.2 \cdot 10^8$	”	”
Hubbard[14]	$1.2 \cdot 10^7$	$1.9 \cdot 10^8$	”	”
Hubbard[16]	$1.7 \cdot 10^8$	$3.0 \cdot 10^9$	”	”
BosHub[20]	$2.0 \cdot 10^7$	$3.0 \cdot 10^8$	”	”
BosHub[22]	$1.3 \cdot 10^8$	$2.0 \cdot 10^9$	”	”
MATPDE[$4k$]	$1.7 \cdot 10^7$	$8.4 \cdot 10^7$	”	none
MATPDE[$16k$]	$2.7 \cdot 10^8$	$1.3 \cdot 10^9$	”	”
MATPDE3D[512]	$1.3 \cdot 10^8$	$9.4 \cdot 10^8$	”	”

(b) larger matrices

Test matrices. We choose the test matrices from two sources. On the one hand, we show some numerical studies with a variety of small problems from various applications (Table 3(a)) and a nonsymmetric problem MATPDE[n] representing a variable (but smooth) coefficient convection-diffusion problem on an $n \times n$ grid.[2]

[2]The matrix generator is available online from http://math.nist.gov/MatrixMarket/data/NEP/matpde/matpde.html.

MATPDE3D$[n]$ is a straightforward generalization to a three-dimensional problem on an $n \times n \times n$ grid. These matrices are partitioned using a simple quadtree (octree) algorithm, and no additional graph partitioning or local bandwidth reduction has to be applied.

On the other hand, we want to test the block algorithm on larger matrices from current real applications in quantum physics research. As typical examples we choose the matrices $\mathsf{Spin}_{\mathsf{SZ}}[L]$, $\mathsf{tV}[L]$ (spin chain and spinless fermions [4]), $\mathsf{Hubbard}[L]$ (fermionic Hubbard model [15]), and $\mathsf{BosHub}[L]$ (bosonic Hubbard model [8]) from solid state and ultracold atomic gas physics. These matrices pose "hard" test cases for exterior eigenvalue computations. First, the spectrum of the matrix depends critically on the test problem and the problem parameters, with the possibility of several closely or truly degenerate eigenvalues. Second, the matrix dimension grows exponentially fast with the physical problem size L, such that scalability of the eigensolver becomes an issue early on.

Consider, e.g., the matrices $\mathsf{Spin}_{\mathsf{SZ}}[L]$ that reflect the Hamilton operator for the Heisenberg XXZ spin chain model with S_z-symmetry. For a chain with L spins, in the case of zero total magnetization and without translational symmetry, the matrix $\mathsf{Spin}_{\mathsf{SZ}}[L]$ is a symmetric matrix with dimension $N_L = \frac{L!}{2(L/2)!}$. It contains between 2 and $L+1$ nonzeroes per row, with about $L/2$ nonzeroes 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 ($N_L \leq 10^8$) the (serial) reverse Cuthill-McKee (RCM) algorithm [12] can be used, but for larger matrices parallel strategies are required [11, 25]. Here, an RCM reordering was used for single-node calculations and PT-SCOTCH [11] for the internode tests.

Notice that a reasonably good partitioning and local ordering of the matrix entries is key to achieving good performance of the single-vector spMVM and of the spMMVM independent of the block size. Thus, the results depend to some extent on the availability of such a preordering. We do not report the time for the partitioning here but mention that it is a significant overhead if tools like PT-SCOTCH have to be used. In some of the larger runs the partitioning took longer than finding the first 20 eigenvalues. An efficient numbering of the unknowns—similar to the fast quadtree ordering used for the MATPDE problem—is a research topic of its own for the quantum physics matrices used as examples here [35] and is beyond the scope of this paper.

Comparison with another implementation. We would like to compare our results to a state-of-the-art implementation of JD. Here we use the PRIMME software [39], with sparse matrix-vector products (and spMMVM) provided by the Trilinos library Epetra [20]. Note that the aim here is not to compare the overall runtimes of the two implementations, as they are very different in nature (in particular, PRIMME's JDQMR method does not use blocking in the inner solves). Instead, we want to verify that the solver presented requires a similar number of iterations in total and that the runtimes give a consistent picture. We run PRIMME/Epetra using MPI on the 10 cores of a CPU socket of the above mentioned cluster for block sizes 1, 2, and 4.

The software PHIST[3] (Pipelined Hybrid-parallel Iterative Solver Toolkit) used

[3]https://bitbucket.org/essex/phist

in our work provides an abstract C interface for sparse parallel linear algebra operations and implements the block JD algorithm discussed here, as well as some Krylov-subspace solvers for large linear systems. The block JD implementation is also executed on one CPU socket of the same machine and uses GHOST to provide the OpenMP-parallel building blocks discussed in section 3. We remark that the PRIMME interface only allows us to provide a sparse matrix-vector product on column-major block vectors and uses standard BLAS library calls for the (block) vector operations. In measurements, providing a faster shared memory parallel spMMVM was outweighed by the slower BLAS operations in (multithreaded) Intel MKL, so that the variant shown here was the fastest we found. The very low performance level of the multithreaded MKL performance for tall-skinny shaped block operations is investigated in [28], where it was shown that manually tuned multithreaded implementations can outperform Intel MKL performance for small block dimensions by more than an order of magnitude.

Table 4
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.*

Method	n_b	Matvecs	Walltime [s]	Time/spMVM [ms]	Block speedup
PRIMME (a)	1	1387	326	80 (34%)	1.00
	2	1688	370	80 (37%)	0.82
	4	1811	403	81 (39%)	0.77
PRIMME (b)	1	1395	327	80 (34%)	1.00
	2	1487	326	80 (37%)	0.94
	4	1746	366	81 (39%)	0.80
PHIST	1	1476	343	53 (23%)	1.00
	2	1562	286	37 (20%)	1.20
	4	1774	252	26 (19%)	1.36

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 eigenvalue residual has been reduced by one order of magnitude (see also [37, 38]). The test matrix is $\mathsf{Spin}_{\mathsf{SZ}}$[26] (cf. Table 3(b)) 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 28 vectors if a basis size of 60 is reached. PHIST performs 28 single-vector Arnoldi steps to construct the initial basis.

The results are shown in Table 4. PRIMME is somewhat faster than our code in the single-vector case, even though the matrix-vector product is slower. This is because it requires fewer spMVMs and probably also saves some other operations by smarter implementation details such as delayed locking. The more sophisticated inner stopping criterion can further accelerate the computations, especially in the block variant (cf. the results for "PRIMME (b)"). As the JDQMR method used here does not solve the correction equation in a blocked fashion, the overall "block speedup" simply reflects the increase in the number of spMVMs required.

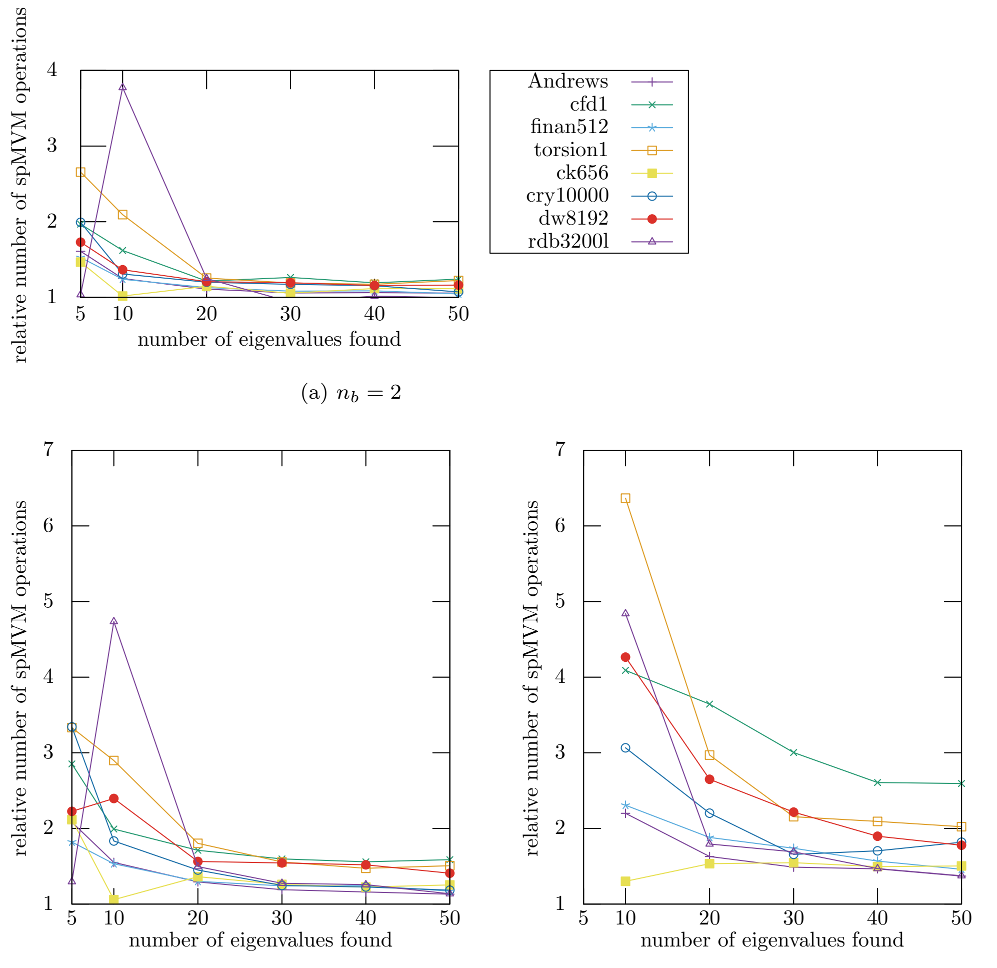
PHIST becomes up to 1.36 times faster due to the techniques discussed in this paper, in particular due to the massive performance gain of the spMMVM and the fast block vector kernels shown in section 3. Noting that the performance of the spMVMs is increased by a factor of 2 by using $n_b = 4$ instead of $n_b = 1$, one might expect an even larger speedup of up to 1.7 here. This is not achieved in practice because other operations in the inner MINRES solver do not benefit likewise from the blocking. For instance, vector AXPY operations achieve the same performance as for a single vector, and scalar products only benefit in the sense that fewer reduction operations are performed, which is not relevant on the single socket CPU used here. Consequently, as the block size is increased, the contribution of the spMMVM decreases and other operations start to dominate the overall runtime. Here more algorithmic optimizations, such as an adaptive inner tolerance, a true block Krylov solver, and a direct usage of TSQR (without intermediate transposition of the memory layout; see section 4.2) may further reduce the overall runtime of PHIST.

We do not claim that the algorithm discussed here is "better" than the certainly more sophisticated implementation in PRIMME. The advantages of the JDQMR method presented in [37] over JDQR are obvious, but they are complementary to the performance advantages of PHIST (i.e., runtime reduction by blocking).

Numerical behavior. To get an idea of the increase in the number of operations due to blocking, we compare the number of spMVMs needed to calculate a given number of eigenvalues with different block sizes. In order to get an indication of the generality of our result—that blocking may reduce the overall runtime of the method—we use a variety of symmetric and nonsymmetric test cases here. These matrices are too small to yield realistic performance data (they typically fit in the cache of the CPU), but from the increase in the number of spMVMs we can estimate under which circumstances it may be beneficial to use a block method for larger problems. Figure 4 shows the relative number of spMVMs compared to the single-vector method.

As soon as more than about 20 eigenpairs are sought (30 for block size 8), the increase in the number of spMVMs is roughly constant, so that a simple benchmark of the spMMVM performance for a realistic matrix on a given machine can be used to choose the block size that is most likely to give good overall performance a priori. Thus, we can expect to improve the performance of the JD method by blocking for a wide range of matrices, symmetric or nonsymmetric. While the spMVM may not be the dominant operation in all applications, the number of other operations—such as preconditioner applications or inner products—typically increases proportionally to this count, which justifies using spMVMs as a proxy here.

Note that in the setup of the numerical tests presented, no prior knowledge of the spectrum of the matrix was assumed. Thus the shifts in the first block JDQR iterations are in some cases far away from the finally calculated set of eigenvalues. This might explain why the block method requires many more spMVM operations for the first eigenvalues found. One might improve the behavior by starting with a single-vector method and switching to the block method only after the first few eigenvalues converge or by using predefined shifts for the first iterations. The latter approach, however, requires some information on the spectrum of the matrix. In the present implementation of the algorithm, we found that increasing the block size to 8 will typically not pay off because there are no sufficiently accurate Ritz values for so many eigenpairs at once, and consequently the overall number of matrix-vector operations increases more than the performance gain can compensate. It may be possible to use a larger block size if one seeks eigenvalues near two targets simultaneously, for instance,



(a) $n_b = 2$

(b) $n_b = 4$

(c) $n_b = 8$

FIG. 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.*

at both ends of the spectrum; however, one should keep in mind that increasing the block size also increases the communication volume of the spMMVM, which is another argument against using $n_b > 4$. In the parallel performance tests below, we therefore do not consider this case.

Parallel performance. Next, we investigate the strong scaling of the code beyond a single node for a spin chain matrix and the MATPDE example (two-dimensional and three-dimensional), which has better scaling properties of the spMVM but is nonsymmetric. Figure 5 shows the runtime results scaled by the number of nodes used, so that a constant bar height would indicate an optimal linear parallel speedup. It is clear from these results that excellent strong scaling can be achieved for both the single-vector and the block methods if the application of the JD operator (i.e., an spM(M)VM followed by an orthogonal projection) scales well. We do not show weak scaling results for the complete algorithm, as the number of iterations depends on the problem size. Hence, weak scaling results would be of very limited

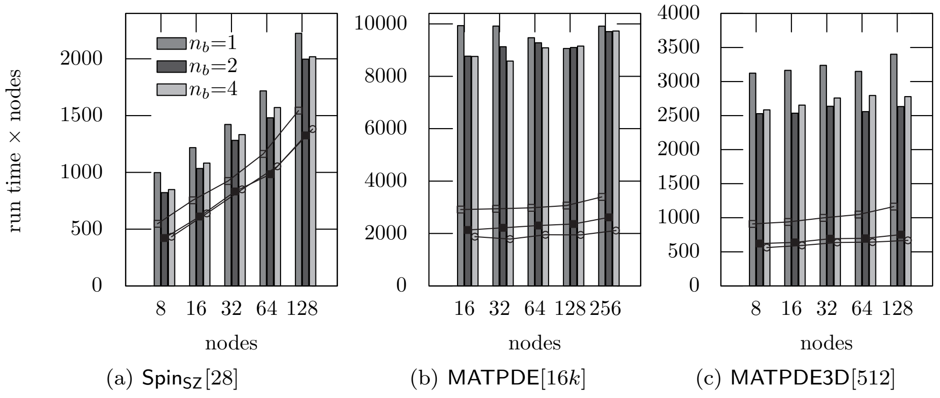


FIG. 5. *Strong scaling of the computation of* 20 *exterior eigenvalues of three matrices on an Intel Ivy Bridge cluster. The different bars represent the block size used, and the lines show the overall contribution of applying the "JD operator"* $(I - QQ^T)(A - \tilde{\lambda}I)$.

significance. The matrices from quantum mechanics show relatively poor strong scaling behavior as more and more communication is required with an increasing node count.

Our final experiments are strong scaling tests for the larger test cases in Table 3(b), where we report the overall speedup achieved by using block sizes 2 or 4, respectively. Figure 6 shows that in most cases, blocking reduces the overall runtime for block sizes 2 and 4 (points above the horizontal line at 1). When too many nodes are used for a given problem size, the behavior changes from memory/network bandwidth bounded to cache/latency bounded. Our analysis and implementation are not intended for this case, which is why we do not report scaling on more than 32 (respectively, 128) nodes here.

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. For the matrices from the Hubbard (and Bose–Hubbard) model, a block speedup is not achieved for the larger node counts because these matrices exhibit a notoriously poor scaling due to their sparsity pattern: In this case the runtime is dominated by the communication bandwidth, such that the higher communication volume of the block method impedes computational gains by blocking.

Summary and conclusions. We have investigated a block formulation of the JD method for general (nonsymmetric) and 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 rowwise storage of blocks of vectors. This design choice is the key to achieving the performance gains we have shown for the sparse matrix-vector products and block vector operations,

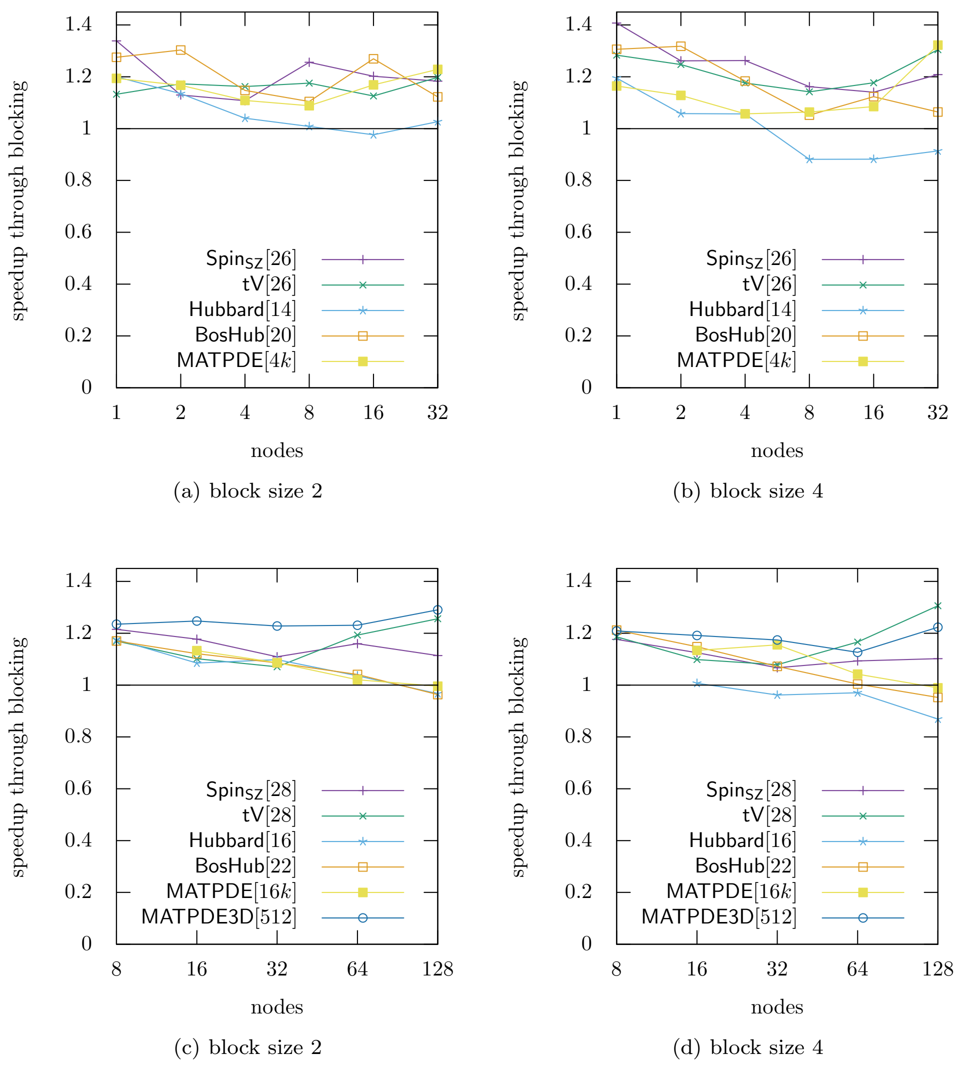


Fig. 6. *Relative performance gains through blocking for the computation of* 20 *exterior eigenvalues of several benchmark matrices on an Intel Ivy Bridge cluster.*

but it has consequences for the formulation of the complete algorithm, as operations on single vectors in a block become much less efficient. We therefore showed how to achieve optimal blocking of operations by collecting the operations from separate inner solves and discussed some ways to further reduce the total number of global synchronization points.

Our numerical results indicate that the block method works well for a wide range of matrices, both symmetric and nonsymmetric. 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 5 120 cores.

In this study the performance and the block speedup on a cluster were mostly determined by the spMMVM performance (cf. Figure 5). The aspect of synchronization reduction by collecting scalar products was discussed but not demonstrated in practice because the test problems from quantum physics lead to a poorly strong-scaling matrix-vector product. We plan to perform a separate study on suitable test cases like MATPDE to demonstrate the benefits of blocking in this respect experimentally.

The direct comparison of eigensolvers and implementations is difficult because there are so many aspects that determine the overall runtime. We chose the JDQMR method in PRIMME. Of the schemes implemented in PRIMME, it is the most similar one to the algorithm discussed, so we could verify that a similar number of matrix-vector operations is needed. It may be interesting to also apply the GD+k block method in PRIMME to the test problems for comparison.

The numerical results confirm that blocking can significantly reduce the time to solution if implemented correctly. Going beyond block sizes of 2 and 4, however, may not pay off because no good directions for those additional eigenmodes are added to the subspace, and the subspace grows very quickly for, e.g., block size $n_b = 8$, increasing the cost of orthogonalization. The communication volume during spMMVM also grows linearly with n_b. With these observations in mind, we advocate the use of "SIMD friendly" matrix storage formats (here SELL-C-σ), as the SIMD width on upcoming CPUs (such as Intel(R) Skylake and Knight's Landing) is already eight doubles.

While we have shown experiments only on a single machine, the performance models can easily be fitted to a variety of architectures, so that we can expect similar overall results.

In the future we will work on including preconditioning techniques in the PHIST library. We also plan to investigate communication hiding techniques to alleviate the increased volume of the transferred data at large node counts and develop fast matrix ordering schemes for matrices from quantum physics. Furthermore, we will support using accelerator hardware such as GPUs, which is already supported by the GHOST library. Both PHIST and GHOST are open source projects and are available under a BSD license online from https://bitbucket.org/essex/.

Appendix. Algorithm template.

Algorithm 1. Complete block Jacobi–Davidson QR (part 1).

Input: $A \in \mathbb{C}^{n\times n}$, hpd $B \in \mathbb{C}^{n\times n}$, $v_0 \in \mathbb{C}^n$, n_{Eig}, n_b, ϵ_{tol}, $maxIter$, m_{min}, m_{max}
Output: approximative partial Schur composition $AQ \approx BQR$

```
    Initialize result:
 1: $Q \leftarrow 0, \quad Q_A \leftarrow 0, \quad Q_B \leftarrow 0, \quad R \leftarrow 0$
 2: $l \leftarrow 0$                                    ▷ number of locked converged eigenvalues

    Initialize search space:
 3: $m \leftarrow m_{min}$                              ▷ current subspace dimension
 4: Compute $m_{min}$ Arnoldi-iterations: $AW_{:,1:m_{min}} = W_{:,1:m_{min}+1}H_{1:m_{min}+1,1:m_{min}}$
    with $w_1 = v_0$, $W_B = BW$ and $W_B^* W = I$
 5: $W \leftarrow W_{:,1:m}, \quad W_B \leftarrow W_{B:,1:m}$
 6: $W_A \leftarrow AW, H \leftarrow (W^* W_A)$          ▷ recalculated to prevent inaccuracies

    Main iteration loop:
 7: for $nIter \leftarrow 1, maxIter$ do

 8:     $\hat{l} \leftarrow \min(l + 2n_b, n_{Eig} + n_b - 1)$      ▷ for multiplicity detection

        Update projected Schur form:
 9:     Calculate Schur decomposition $H_{l:m,l:m}q^H = q^H r^H$
        with the eigenvalues on the diagonal of $r^H$ sorted by modulus
        and locked part in $R_{1:l,1:l}$
10:     $Q^H \leftarrow \begin{pmatrix} I & 0 \\ 0 & q^H \end{pmatrix}, \quad R^H \leftarrow \begin{pmatrix} R_{1:l,1:l} & H_{1:l,l:m}q^H \\ 0 & r^H \end{pmatrix}$

        Update approximate Schur form:
11:     $q_i \leftarrow WQ^H_{:,i}, \quad (q_A)_i = W_A Q^H_{:,i}, \quad (q_B)_i = W_B Q^H_{:,i},$     $i = l, \dots, \hat{l}$
12:     $r_i \leftarrow R^H_{1:i,i},$                                      $i = l, \dots, \hat{l}$
13:     $res_i \leftarrow (q_A)_i - (Q_B)_{:,1:i} r_i,$                    $i = l, \dots, \hat{l}$
14:     $\epsilon_i \leftarrow \|res_i\|_2,$                               $i = l, \dots, \hat{l}$

15:     For Hermitian $A$:    ▷ otherwise omitted due to possibly bad condition number
        Reorder multiple eigenvalues in $(Q^H, R^H)$ by $\epsilon_i$
16:     Update ordering of $Q, Q_B, R, res$
```

Algorithm 2. Complete block Jacobi–Davidson QR (part 2).

```
    Check for converged eigenpairs
17:     $\Delta l \leftarrow \max\{i : \epsilon_i < \epsilon_{tol}, i = 1, \dots, n_{Eig}\} - l$
18:     if $\Delta l > 0$ then
19:         $W \leftarrow W{Q^H}_{:,1:m}, \quad W_A \leftarrow W_A{Q^H}_{:,1:m}, \quad W_B \leftarrow W_B{Q^H}_{:,1:m}$
20:         $H \leftarrow {Q^H}^*_{:,1:m} H {Q^H}_{:,1:m}$
21:         $Q^H \leftarrow I, \quad R^H \leftarrow {R^H}_{1:m,1:m}$
22:         $l \leftarrow l + \Delta l$
23:         if $l = n_{Eig}$ then
24:             return $(Q, R)$
25:         end if
26:         $\hat{n}_b \leftarrow \min(n_b, \hat{l} - l)$                    ▷ effective block size
27:     end if

    Shrink search space:
28:     if $m + \hat{n}_b > m_{max}$ then
29:         $W \leftarrow W{Q^H}_{:,1:m_{min}}, \quad W_A \leftarrow W_A{Q^H}_{:,1:m_{min}}, \quad W_B \leftarrow W_B{Q^H}_{:,1:m_{min}}$
30:         $H \leftarrow {Q^H}^*_{:,1:m_{min}} H {Q^H}_{:,1:m_{min}}$
31:         $m \leftarrow m_{min}$
32:     end if

    Calculate corrections:
33:     Choose $\tilde{l} \geq l + \hat{n}_b$ depending on eigenvalue multiplicity
34:     $\tilde{Q} \leftarrow Q_{1:\tilde{l}}, \quad \tilde{Q}_B \leftarrow (Q_B)_{1:\tilde{l}}$
35:     for $i \leftarrow 1, \hat{n}_b$ do
36:         Solve approximately $(I - \tilde{Q}_B\tilde{Q}^*)(A - r_{l+i,l+i}B)(I - \tilde{Q}\tilde{Q}_B^*)t_i = -res_{l+i}$
37:         $t_i \leftarrow (I - \tilde{Q}\tilde{Q}_B^*)t_i$
38:     end for

    Enlarge search space:
39:     Orthogonalize $t$ w.r.t. $W$ and $\langle\cdot,\cdot\rangle_B$;
        use random orthogonal vector on breakdown
40:     $t_A \leftarrow At, \quad t_B \leftarrow Bt$
41:     $H \leftarrow \begin{pmatrix} H & W^* t_A \\ t^* W_A & t^* t_A \end{pmatrix}$
42:     $W \leftarrow \begin{pmatrix} W & t \end{pmatrix}, \quad W_A \leftarrow \begin{pmatrix} W_A & t_A \end{pmatrix}, \quad W_B \leftarrow \begin{pmatrix} W_B & t_B \end{pmatrix}$

43: end for

44: abort                    ▷ no convergence after maxIter iterations
```

REFERENCES

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

[2] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang, *Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations*, in Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, 2014, pp. 1213–1222.

[3] J. Andrzejewski, *On optimizing Jacobi-Davidson method for calculating eigenvalues in low dimensional structures using eight band $k \cdot p$ model*, J. Comput. Phys., 249 (2013), pp. 22–35.

[4] A. Auerbach, *Interacting Electrons and Quantum Magnetism*, Grad. Texts Contemp. Phys., Springer, New York, 1994.

[5] A. H. Baker, J. M. Dennis, and E. R. Jessup, *On improving linear solver performance: A block variant of GMRES*, SIAM J. Sci. Comput., 27 (2006), pp. 1608–1626.

[6] F. L. Bauer, *Das Verfahren der Treppeniteration und verwandte Verfahren zur Lösung algebraischer Eigenwertprobleme*, Z. Angew. Math. Phys., 8 (1957), pp. 214–235.

[7] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, *An updated set of basic linear algebra subprograms (BLAS)*, ACM Trans. Math. Software, 28 (2002), pp. 135–151.

[8] I. Bloch, J. Dalibard, and W. Zwerger, *Many-body physics with ultracold gases*, Rev. Modern Phys., 80 (2008), pp. 885–964.

[9] 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, London, 1997, pp. 125–137.

[10] M. Bollhöfer and Y. Notay, *JADAMILU: A software code for computing selected eigenvalues of large sparse symmetric matrices*, Comput. Phys. Commun., 177 (2007), pp. 951–964.

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

[12] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th National ACM Conference, 1969, pp. 157–172.

[13] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), 1.

[14] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput., 34 (2012), pp. A206–A239.

[15] F. H. L. Essler, H. Frahm, F. Göhmann, A. Klümper, and V. E. Korepin, *The One-Dimensional Hubbard Model*, Cambridge University Press, Cambridge, UK, 2005.

[16] Y. T. Feng, *An integrated Davidson and multigrid solution approach for very large scale symmetric eigenvalue problems*, Comput. Methods Appl. Mech. Engrg., 190 (2001), pp. 3543–3563.

[17] 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 J. Sci. Comput., 20 (1998), pp. 94–125.

[18] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, *Towards realistic performance bounds for implicit CFD codes*, in Proceedings of Parallel CFD '99, Elsevier, Amsterdam, 1999, pp. 233–240.

[19] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, Chapman & Hall/CRC Comput. Sci. Ser., CRC Press, Boca Raton, FL, 2010.

[20] 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. Software, 31 (2005), pp. 397–423.

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

[22] M. Hoemmen, *Communication-Avoiding Krylov Subspace Methods*, Ph.D. thesis, University of California, Berkeley, Berkeley, CA, 2010.

[23] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol, *Cache efficient bidiagonalization using BLAS 2.5 operators*, ACM Trans. Math. Software, 34 (2008), pp. 14:1–14:33.

[24] E.-J. Im, K. A. Yelick, and R. Vuduc, *SPARSITY: An optimization framework for sparse matrix kernels*, Int. J. High Perform. C., 18 (2004), pp. 135–158.

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

[26] A. Klinvex, F. Saied, and A. Sameh, *Parallel implementations of the trace minimization scheme TraceMIN for the sparse symmetric eigenvalue problem*, Comput. Math. Appl., 65 (2013), pp. 460–468.

[27] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors*

with wide SIMD units, SIAM J. Sci. Comput., 36 (2014), pp. C401–C423.

[28] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein, *GHOST: Building Blocks for High Performance Sparse Linear Algebra on Heterogeneous Systems*, CoRR, abs/1507.08101, 2015.

[29] B. C. Lee, R. W. Vuduc, J. W. Demmel, K. A. Yelick, M. de Lorimier, and L. Zhong, *Performance Optimizations and Bounds for Sparse Symmetric Matrix-Multiple Vector Multiply*, Tech. Report UCB/CSD-03-1297, EECS Department, University of California, Berkeley, Berkeley, CA, 2003.

[30] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, *Improving the performance of dynamical simulations via multiple right-hand sides*, in Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium, 2012, pp. 36–47.

[31] J. D. McCalpin, *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, Tech. Report, University of Virginia, Charlottesville, VA, 1991–2007 (continually updated).

[32] Y. Notay, *Convergence analysis of inexact Rayleigh quotient iteration*, SIAM J. Matrix Anal. Appl., 24 (2003), pp. 627–644.

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

[34] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, revised ed., Classics Appl. Math. 66, SIAM, Philadelphia, 2011.

[35] G. Schubert, H. Fehske, G. Hager, and G. Wellein, *Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems*, Parallel Process. Lett., 21 (2011), pp. 339–358.

[36] A. Stathopoulos, *Locking Issues for Finding a Large Number of Eigenvectors of Hermitian Matrices*, Tech. Report WM-CS-2005-09, Department of Computer Science, College of William and Mary, Williamsburg, VA, 2005; preprint, submitted to Elsevier Science, 2006.

[37] A. Stathopoulos, *Nearly optimal preconditioned methods for Hermitian eigenproblems under limited memory. Part* I*: Seeking one eigenvalue*, SIAM J. Sci. Comput., 29 (2007), pp. 481–514.

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

[39] A. Stathopoulos and J. R. McCombs, *PRIMME: Preconditioned iterative multimethod eigensolver—methods and software description*, ACM Trans. Math. Software, 37 (2010), pp. 1–30.

[40] P. T. P. Tang and E. Polizzi, *FEAST as a subspace iteration eigensolver accelerated by approximate spectral projection*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 354–390.

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

[42] S. W. Williams, A. Waterman, and D. A. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

[43] K. Wu, Y. Saad, and A. Stathopoulos, *Inexact Newton preconditioning techniques for large symmetric eigenvalue problems*, Electron. Trans. Numer. Anal., 7 (1998), pp. 202–214.

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