Deutsches Zentrum
für Luft- und Raumfahrt
German Aerospace Center

RWTH AACHEN UNIVERSITY

Parallel solution of large sparse eigenproblems using a Block-Jacobi-Davidson method

Parallele Lösung großer dünnbesetzter Eigenwertprobleme mit einem Block-Jacobi-Davidson Verfahren

Masterarbeit

in der Studienrichtung

Computational Engineering Science

an der

Rheinisch-Westfälischen Technischen Hochschule Aachen

von

Melven Röhrig-Zöllner

Externer Betreuer:
Dr. Jonas Thies (DLR)

Betreuer:
Prof. Paolo Bientinesi (Ph.D.)
Dr. Edoardo Di Napoli

February 25, 2014

Abstract

This thesis deals with the computation of a small set of exterior eigenvalues of a given large sparse matrix on present (and future) supercomputers using a Block-Jacobi-Davidson method. The main idea of the method is to operate on blocks of vectors and to combine several sparse matrix-vector multiplications with different vectors in a single computation. Block vector calculations and in particular sparse matrix-multiple-vector multiplications can be considerably faster than single vector operations if a suitable memory layout is used for the block vectors. The performance of block vector computations is analyzed on the node-level as well as for a cluster of nodes.
The implementation of the method is based on an existing sparse linear algebra framework and exploits several layers of parallelism. Numerical tests show that the block method developed works well for a wide range of matrices and that a small block size can speed up the complete computation of a set of eigenvalues significantly in comparison to a single vector calculation.

Zusammenfassung

Ziel dieser Arbeit ist die Berechnung einer kleinen Anzahl äußerer Eigenwerte großer dünnbesetzter Matrizen auf aktuellen (und zukünftigen) Supercomputern. Dazu wird eine Block-Jacobi-Davidson-Methode entwickelt, in der viele Rechenoperationen auf ganze Blöcke von Vektoren angewendet werden. Insbesondere werden so mehrere Matrix-Vektor-Multiplikationen mit verschiedenen Vektoren zugleich bestimmt. Dadurch steigt die Performance der benötigten Rechenoperationen erheblich, falls man ein geeignetes Speicherlayout für die Blöcke von Vektoren verwendet. Dies zeigt sich in der Untersuchung der Node-Level-Performance der Matrix-Blockvektor-Multiplikation in dieser Arbeit. Auch die parallele Performance auf HPC-Clustern wird so verbessert.
Die Implementierung der Methode basiert auf einem bestehenden Framework für lineare Algebra mit dünnbesetzten Matrizen und nutzt Parallelität auf mehreren Ebenen. In numerischen Experimenten mit Eigenwertproblemen aus verschiedenen Disziplinen erzielt die Block-Methode gute numerische Resultate und in Performance-Tests ergibt sich für kleine Blockgrößen eine deutliche Beschleunigung der kompletten Rechnung.

Eidesstattliche Erklärung

Ich versichere hiermit, die vorgelegte Arbeit in dem gemeldeten Zeitraum ohne unzulässige fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben.
Diese Arbeit wurde bisher weder im In- noch im Ausland als Prüfungsarbeit vorgelegt.

Aachen, den 25. Februar 2014

(Vorname Nachname)

Contents

1. Introduction

This thesis deals with the solution of large sparse eigenvalue problems on present (and future) supercomputers. A particular focus lies on the calculation of a small set of exterior eigenvalues of possibly non-symmetric matrices. I have developed the algorithm presented in this thesis as part of the ESSEX (Equipping Sparse Solvers for Exascale) project[1] from the DFG program SPPEXA[2].
Large linear eigenvalue problems arise in many important applications; a particular application that is addressed in the ESSEX project comes from quantum mechanics:
A simple model in the theoretical condensed matter physics consists of interacting electron spins in a magnetic field (see for example [14]); when we consider a system of L electrons with *spin up* or *spin down*, their configuration can be described by a state vector Ψ of dimension 2^L (due to quantum superposition of the individual spin states). The stationary states are given by the following form of the *Schrödinger* equation

$$\mathbf{H}\Psi = E\Psi \ .$$

The Hamiltonian operator $\mathbf{H}$ results from interactions between the spins of the electrons. Assuming only interactions between a small number of neighboring spins, we obtain a large sparse eigenvalue problem where we are interested in the states (e.g. eigenvectors) with the lowest energy levels E (e.g. eigenvalues).
This short example illustrates that the theoretical analysis of a small quantum system may already require the solution of a large eigenvalue problem, and that the dimension of the problem can grow exponentially with the number of particles considered.

In this thesis I investigate a block Jacobi-Davidson method that performs matrix-vector multiplications and vector-vector calculations with several vectors at once. In order to understand the possible performance gains of block methods I analyze block variants of numerical linear algebra operations, in particular of the sparse matrix-vector multiplication, using simple performance models. This helps to improve the performance of the required operations and to verify that the implementation achieves (approximately) the performance limit of the operation on a specific machine. The performance limit is derived through theoretical considerations and benchmarking results.
Jacobi-Davidson methods for the calculation of several eigenvalues were originally presented in [4]. Stathopoulos and McCombs excellently investigated Jacobi-Davidson and Generalized Davidson methods for symmetric (respectively Hermitian) eigenvalue problems in [21] and also addressed block methods briefly.
An interesting new development in the field of parallel sparse eigensolvers is the FEAST algorithm which is theoretically analyzed in [22]. FEAST is also a research topic of the ESSEX project because it permits the calculation of inner eigenvalues (or even all eigenvalues). As it requires the solution of linear systems with multiple right-hand sides, the performance analysis of block operations in this thesis is interesting for FEAST, too. This also applies to TraceMin-Multisectioning, another parallel approach to calculate all eigenvalues of a symmetric matrix in a specific interval [12].

[1] For information about the ESSEX project, see `http://blogs.fau.de/essex`
[2] The German Priority Programme *Software for Exascale Computing*, see `http://www.sppexa.de`

The outline of this thesis is as follows:
First, I present a short overview of the mathematical theory of sparse eigensolvers that is needed to understand the block Jacobi-Davidson method derived later. Then I analyze the performance of block variants of important linear algebra operations. These include vector operations as well as sparse matrix-vector multiplications. Afterwards, I derive a block Jacobi-Davidson method for the calculation of a set of exterior eigenvalues. Special attention is paid to the case of multiple eigenvalues and to avoiding unnecessary communication in the algorithm. Subsequently, important aspects of my implementation of the method are discussed. Numerical results from experiments with varying block sizes substantiate the assumptions made in the derivation of the method and illustrate its numerical behavior for eigenvalue problems from different applications. Finally, it is demonstrated that the block method can be faster than a single vector algorithm on the basis of a set of eigenvalue problems from quantum mechanics.

2. Sparse eigensolvers

This thesis deals with the problem of finding a set of eigenvalues $\lambda_i \in \mathbb{C}$ with corresponding eigenvectors $v_i \in \mathbb{C}^n$ of a given large sparse matrix $A \in \mathbb{C}^{n\times n}$:

$$Av_i = \lambda_i v_i. \tag{1}$$

See [25] or [18] for a detailed discussion of the topic. Here I want to concentrate on the general case with a non-Hermitian complex matrix A and present some basic ideas that are needed later.
For problems with small to medium dimension n there are standard methods available: These are based on the reduction of A to some simpler form, e.g. the Schur form

$$AQ = QR.$$

Here $Q \in \mathbb{C}^{n\times n}$ denotes an orthogonal matrix and $R \in \mathbb{C}^{n\times n}$ an upper triangular matrix, whose diagonal entries are obviously the eigenvalues of A. This Schur decomposition can be calculated by an iterative process that is based on successive QR decompositions of the form ($Q_1 = I$, $Q_2R_2 = AQ_1$, $Q_3R_3 = AQ_2$, ...). Combined with properly chosen shifts and reductions to Hessenberg form one obtains fast convergence so that the overall costs are of order $O(n^3)$ operations and $O(n^2)$ storage. For this reason these methods are also called *direct methods*.
For large problems with sparse matrices this is not a suitable approach. For this case several methods were developed in the past century; but this is still an active area of research because these methods usually lack the robustness of direct methods for general matrices. In order to develop efficient algorithms for the solution of large sparse eigenproblems on current supercomputers it is also important to consider aspects of high performance computing, especially concerning the parallelization.

2.1. General approach

In order to reduce the dimension of the problem one usually works with a subspace of lower dimension $\mathcal{W} = \operatorname{span}\{w_1, w_2, \ldots, w_k\} \subset \mathbb{C}^{n\times n}$. We can project the problem onto this subspace using one of the following conditions:

- *Ritz-Galerkin* condition: $Av - \lambda v \perp \mathcal{W}, \quad v \in \mathcal{W}$,
- *Petrov-Galerkin* condition: $Av - \lambda v \perp \tilde{\mathcal{W}}, \quad v \in \mathcal{W}$.

Here $\tilde{\mathcal{W}} = \operatorname{span}\{\tilde{w}_1, \tilde{w}_2, \ldots, \tilde{w}_k\} \subset \mathbb{C}^{n\times n}$ denotes another subspace with the same dimension as $\mathcal{W}$.
If we use an orthogonal basis $W = \begin{pmatrix} w_1 & w_2 & \ldots & w_k \end{pmatrix}$ for $\mathcal{W}$ (respectively two biorthogonal bases for $\mathcal{W}, \tilde{\mathcal{W}}$ in the *Petrov-Galerkin* case) we can also formulate the above conditions as

$$\begin{aligned} & W^*(AWs - \lambda Ws) = 0, && s \in \mathbb{C}^k \\ \Leftrightarrow \quad & \underbrace{(W^*AW)}_{M} s - \lambda s = 0 \end{aligned} \tag{2}$$

or as $(\tilde{W}^*AW)s - \lambda s = 0$ for the *Petrov-Galerkin* projection.
So from the *Galerkin* condition we obtain an eigenvalue problem of much lower dimension with a matrix $M \in \mathbb{C}^{k\times k}$ which is essentially a projection of A onto the subspace $\mathcal{W}$. For the projected matrix M we can compute the eigenvalues and eigenvectors using a direct method. The eigenvalues are called *Ritz* values of the original problem and from the eigenvectors s_i of M we obtain the corresponding *Ritz* vectors $\tilde{v}_i = Ws_i$. On certain assumptions these Ritz values and vectors provide approximations for the eigenvalues and eigenvectors of A (for more information please refer to [25, section 15 in chapter V], respectively the references therein).

One possibility to construct the subspace $\mathcal{W}$ consists in a *Krylov* approach where we select $\mathcal{W} = \mathcal{K} = \{x, Ax, A^2x, \dots\}$ for some starting vector $x \in \mathbb{C}^n$. The idea behind this is the power method; for increasing k the vector A^kx tends to point more and more in the direction of the eigenvector of the biggest eigenvalue $|\lambda_i|$. In the Hermitian case one can exploit a short recurrence for the efficient construction of an orthogonal basis for $\mathcal{K}$. This results in the *Lanczos* method.
For general matrices it is necessary to orthogonalize $A^{k+1}v$ in each step to all previous vectors $w_1, w_2, \dots, w_k$. This results in the *Arnoldi* method.
Using a *Petrov-Galerkin* condition one can also efficiently construct two bi-orthogonal bases with a short recurrence for non-Hermitian matrices. This is the *Bi-Lanczos* method. For details, see [25, chapter VIII].
Another possibility is the *Jacobi-Davidson* method where the subspace $\mathcal{W}$ is constructed by an inexact Newton process. A more detailed description of this method will be presented in Section 4.

A completely different approach consists in the spectral projection of the *FEAST* algorithm which essentially combines ideas from complex analysis with linear algebra. It is based on integrals of the matrix function $f(\theta) = (A-\theta I)^{-1}, \theta \in \mathbb{C}$ on closed curves in the complex plane [22].
Yet another approach is used in the TraceMin-Multisectioning algorithm presented in [12]: it transforms the problem of finding a set of eigenvalues into an optimization problem that seeks to minimize the trace of a projected matrix.

2.2. Generalized eigenproblems

The standard eigenvalue problem (1) is a special case of the generalized eigenvalue problem:
Given matrices $A, B \in \mathbb{C}^{n\times n}$, find pairs $\alpha, \beta \in \mathbb{C}$ and corresponding vectors $v \in \mathbb{C}^n$ with

$$\alpha Av = \beta Bv.$$

Here $\lambda = \frac{\alpha}{\beta}$ respectively $\bar{\lambda} = \frac{\beta}{\alpha}$ is called a generalized eigenvalue and v a generalized eigenvector of the matrix stencil (A, B).

In this thesis I only consider the special case where one matrix is Hermitian positive definite:

Given a matrix $A \in \mathbb{C}^{n\times n}$ and a Hermitian positive definite matrix (hpd) $B \in \mathbb{C}^{n\times n}$ we look for generalized eigenpairs (λ_i, v_i), $\lambda_i \in \mathbb{C}$, $v_i \in \mathbb{C}^n$ with

$$Av_i = \lambda_i B v_i. \tag{3}$$

We can treat this problem quite similar to the standard eigenvalue problem in many cases because the Hermitian positive definite matrix B induces the inner product

$$< \cdot, \cdot >_B : x, y \in \mathbb{C}^n \to x^H B y.$$

Considering the *Ritz-Galerkin* condition for the generalized eigenproblem with hpd B, we see

$$\begin{aligned} & Av - \lambda Bv \perp \mathcal{W},\ v \in \mathcal{W} \\ \Leftrightarrow \qquad & B^{-1}Av - \lambda v \perp_B \mathcal{W},\ v \in \mathcal{W}. \end{aligned} \tag{4}$$

That means we can transform it to the standard case just by switching the inner product to $< \cdot, \cdot >_B$. Similar to (2) we obtain a standard eigenvalue problem for the *Galerkin* projection onto $\mathcal{W}$ if we use a B-orthogonal basis W:

$$\begin{aligned} & W^H B(B^{-1}AWs - \lambda Ws) = 0, && s \in \mathbb{C}^k \\ \Leftrightarrow \qquad & \underbrace{(W^H AW)}_{M} s - \lambda s = 0. \end{aligned} \tag{5}$$

In most parts of this thesis I will only discuss the standard eigenvalue problem and point out some necessary adaptations in the algorithms for this special case of the generalized eigenproblem.
Additionally, we can replace the generalized eigenproblem with hpd B by the standard eigenproblem with the matrix $B^{-1}A$ in the Hilbert space $(R^n, < \cdot, \cdot >_B)$. For this we may just redefine the adjoint operator as $(\cdot)^* := ((\cdot)^H B)$. Subsequently, we can directly transfer all results for the standard eigenvalue problem to the special generalized eigenvalue problem from (3).

2.3. Basic Techniques for preconditioning and deflation

The conditioning of a simple isolated eigenvalue is given by

$$\delta\lambda_i = \frac{1}{\bar{v}_i^* v_i} \bar{v}_i^* \delta A v_i \ .$$

Here δA denotes a small deviation in the matrix A, v_i and $\bar{v}_i$ the right and the left eigenvector and $\delta\lambda_i$ the resulting deviation of the eigenvalue λ_i.
We see that for a Hermitian matrix A, where the right and left eigenvectors are identical, the condition number is one. However, for non-Hermitian matrices it depends on the angle between the right and the left eigenvector.

The condition number of an eigenvector depends on the distance of the corresponding eigenvalue to the remaining part of the spectrum, which is

$$\mathrm{cond}(v_i) = \frac{1}{\mathrm{dist}(\lambda_i, \Lambda(A) \setminus \{\lambda_i\})}$$

for an isolated eigenvalue in the Hermitian case. In this case the condition number gives also a nice a posteriori error bound for Ritz values:

$$|\tilde{\lambda}_i - \lambda_i| \leq \frac{\|r\|_2^2}{\mathrm{dist}(\lambda_i, \Lambda(A) \setminus \{\lambda_i\})}.$$

Here $\tilde{\lambda}_i = \frac{\tilde{v}_i^* A \tilde{v}_i}{\tilde{v}_i^* \tilde{v}_i}$ denotes a Ritz-value and $r = A\tilde{\lambda}_i - A\tilde{v}_i$ the corresponding residual. This indicates that in cases of multiple or tightly clustered eigenvalues it is sensible to search for the invariant subspace spanned by the corresponding eigenvectors and that the conditioning of an eigenvector can be improved by separating the wanted eigenvalue from the remaining part of the spectrum.

So in order to compute eigenvalues near a specific target $\theta \in \mathbb{C}$ we can replace the matrix A in the subspace methods mentioned previously by the matrix

$$\bar{A} = (A - \theta I)^{-1}.$$

This is called *shift and invert*. As a result the desired eigenvalues of the shifted and inverted matrix $\bar{\lambda}_i = \frac{1}{\lambda_i - \theta}$ are amplified and hopefully better separated from the rest of the spectrum whereas the eigenvectors remain the same. The term above is usually not computed directly, but rather systems of the form $(A - \theta I)z = b$ are solved.
A more elaborated approach that amplifies the spectrum near a target value θ and damps it near $\tau \in \mathbb{C}$ is the *Cayley* transform (formulated for the generalized eigenproblem (3)):

$$\bar{A} = (A - \theta B)^{-1}(A - \tau B). \tag{6}$$

The resulting eigenvalues of $\bar{A}$ are $\bar{\lambda}_i = \frac{\lambda_i - \tau}{\lambda_i - \theta}$. Additionally, it transforms the generalized eigenproblem with the matrices A, B to a standard eigenvalue problem of $\bar{A}$.
The two approaches presented above can be seen as a special form of polynomial preconditioning where we consider a (rational) matrix polynomial in A which amplifies a specific part in the spectrum of A.

There are also techniques to blend out already computed eigenvalues and -vectors; assuming we have a method that calculates the largest eigenvalue λ_1 and the corresponding eigenvector v_1, and we now want to determine the second largest eigenpair (λ_2, v_2): To achieve this we can try making λ_2 the largest eigenvalue of some modified matrix $\bar{A}$.
In the *Wielandt's* deflation procedure we replace A by

$$\bar{A} := A - Q\Sigma Q^*,$$

where $Q = \begin{pmatrix} q_1 & \dots & q_k \end{pmatrix}$ denotes an orthogonal basis that spans the eigenspace of all already computed eigenvectors $v_1, \dots, v_k$, and $\Sigma = \mathrm{diag}(\sigma_1, \dots, \sigma_k)$ is a set of appropriate shifts. The resulting spectrum of $\bar{A}$ is given by $\Lambda(\bar{A}) = \{\lambda_1 - \sigma_1, \dots, \lambda_k - \sigma_k, \lambda_{k+1}, \dots, \lambda_n\}$.
We can also define a deflation procedure based on orthogonal projections: Granted that we already have a partial Schur form

$$AQ_k = Q_k R_k$$

with an orthogonal matrix $Q_k \in \mathbb{C}^{n \times k}$ and an upper triangular matrix $R_k \in \mathbb{C}^{k \times k}$, and now we want to expand it by another vector $q_{k+1} \in \mathbb{C}^n, Q_k^* q = 0$:

$$A \begin{pmatrix} Q_k & q_{k+1} \end{pmatrix} = \begin{pmatrix} Q_k & q_{k+1} \end{pmatrix} \begin{pmatrix} R_k & * \\ & r_{k+1,k+1} \end{pmatrix}.$$

This is obviously equivalent to determining an eigenpair of the projection of A onto the orthogonal complement $Q_k^{\perp}$:

$$\Rightarrow \qquad (I - Q_k Q_k^*)(A - r_{k+1,k+1} I)(I - Q_k Q_k^*) q = 0.$$

This section should only provide an idea of some basic operations involved in many eigensolvers. For more details see [18, chapter 4].

2.4. Theory of convergence

For non-Hermitian matrices there are few general convergence results. Even for simple cases the Ritz values can converge very slowly or result in bad eigenvalue approximations (see [25, sections 29 and 33 in chapter VII] for some illustrative examples).
An important concept in the theory and also in the practical implementation of eigensolvers is the *Rayleigh quotient* ρ_A of a vector $w \in \mathbb{R}^n$. It is defined as

$$\rho_A(w) = \frac{w^* A w}{w^* w}. \tag{7}$$

For a given vector w it minimizes the residual $\|Aw - \rho w\|_2, \rho \in \mathbb{R}$; so in this sense it is the best estimate for the eigenvalue to a given eigenvector approximation. Obviously Ritz values are Rayleigh quotients of the corresponding Ritz vectors.
In the following I describe the convergence results of two very simple methods that provide some insight into the behavior of more elaborated methods.

As mentioned previously the power method calculates iterates of the form

$$w_{k+1} = \frac{A w_k}{\|A w_k\|_2}.$$

On the assumption $|\lambda_1| > |\lambda_2| \geq \dots$ the Rayleigh quotients $\rho_A(w_k)$ converge to the dominant eigenvalue λ_1 if the starting vector is not orthogonal to v_1. The rate of convergence is linear and depends on the factor $\frac{|\lambda_2|}{|\lambda_1|}$ (see [25, section 8, chapter IV]).

This directly presents a lower bound for the convergence rate of Krylov subspace methods. Since all previous directions are retained in the subspace, they can often provide much better approximations. Additionally, if we seek several eigenvalues, slow convergence to one eigenvalue may come hand in hand with faster convergence to the next eigenvalue [25, section 46 in chapter IX].
Note that preconditioning the problem with a shift and invert approach leads to the factor $\frac{|\lambda_1-\theta|}{|\lambda_2-\theta|}$ which is possibly much smaller for a well-chosen shift θ.

If we update the shift in each iteration using the current eigenvalue approximation $\rho_A(w_k)$, we obtain the following *Rayleigh quotient iteration* (RQI):

$$w_{k+1} = \frac{(A - \rho_A(w_k)I)^{-1} w_k}{\|(A - \rho_A(w_k)I)^{-1} w_k\|_2}. \tag{8}$$

For Hermitian matrices this method converges globally to some eigenvalue and its speed of convergence is essentially cubic in the neighborhood of an eigenpair. For non-Hermitian matrices it still features quadratic local convergence (see [16] for an overview of the convergence properties of RQI methods).
I will refer to these properties in the discussion of Jacobi-Davidson methods in Section 4.

3. Sparse linear algebra algorithms on present HPC systems

In this section I describe some aspects of present supercomputers that are crucial for the development of efficient sparse linear algebra algorithms. For a general introduction to high performance computing the reader may refer to [5].
A simplified model for a present supercomputer is a cluster of compute nodes connected by a network (*distributed memory* architecture) where each compute node consists of several cores, e.g. single sequential computation units that share memory and resources (*shared memory* architecture). Additionally, they could contain special accelerator devices, but for simplicity I only consider clusters of multi-core nodes here.
The performance characteristics of such a supercomputer are complex:
Communication between nodes requires sending messages over the network (for example using MPI), which is usually slow (in terms of bandwidth and latency) compared to memory accesses. Still, the speed of the main memory alone is insufficient to feed the cores with enough data to keep them working. That is why in today computers a hierarchy of faster caches is employed to hide latencies of data transfers from main memory. The usual setup consists of one small fast cache per core and one or several larger slower caches that are possibly shared by all or several cores in a node. The caches are organized in lines that contain consecutive elements of data.

From the model above we can derive two important aspects that influence the performance of a given application: Parallelism and data locality.

Parallelism and concurrency We need to distribute the work among the compute nodes of the cluster and among the cores on each node. So we have to exploit several levels of parallelism to account for the different characteristics of inter-node and intra-node communication.
We can distinguish between two concepts that help to parallelize a given application:

- Distributing the data (*data parallelism*) such that each compute unit does a part of the same global operation on his own chunk of data. Communication is needed to resolve data dependencies. Notably, this allows computations that are not possible on a single node due to memory restrictions.
- Splitting the algorithm into (mostly) independent subtasks (*function parallelism*) such that the compute units each do their own operation on different data items. Synchronization is needed to exchange data between interdependent subtasks.

In the field of sparse eigensolvers we have large data (sparse matrices and several sets of large (dense) vectors), thus we obviously need to distribute these data structures among the compute nodes. On each compute node we can employ both approaches. In particular, it may be useful to execute independent subtasks concurrently in order to hide communication latencies.
I also need to note that it may be cheaper in some cases to do additional work in order to avoid slow communication (*communication avoiding*).

Data locality In order to minimize data transfers from slower cache or main memory one should exploit data locality.
This refers to *temporal locality*; that is reusing data recently accessed, as well as to *spatial locality*; that is accessing data stored in multiple consecutive (or nearby) memory addresses sharing a cache line.
As multiple cores in one node usually share a common cache, it is probably not advisable to execute several independent subtasks concurrently if these access large amounts of (different) data.

So the performance analysis can basically be split into two parts: First, I examine the node-level performance which includes effects of the cache architecture. Then I address the inter-node performance which is influenced by the communication pattern and the load (im-)balance. The following subsection gives a rough overview of the operations to be discussed.

3.1. Basic operations

The basic operations needed by sparse linear algebra are essentially sparse matrix-vector products and vector-vector operations.
Concerning the sparse matrix-vector multiplications ($y \leftarrow Ax$, $A \in \mathbb{C}^{n\times n}$, $x, y \in \mathbb{C}^n$) the performance depends on the sparsity pattern of the matrix A: Already for moderately sized problems the single node performance is limited by the main memory bandwidth because the matrix and vectors do not fit into the cache. And it may be necessary to load the vector x or y (depending on the matrix storage format) multiple times because the accesses to the vector exhibit bad spatial and temporal locality.
This situation can be improved by reordering the rows and columns of the matrix (see for example [13]). Identifying dense blocks (through reordering and storing some additional zero entries) can also greatly improve the performance because this allows more consecutive memory accesses and reduces indexing overhead.
If the sparse matrix A is distributed over the nodes, the inter-node performance is easily dominated by communication. The ideas above also apply for this case; additionally, it may be worth overlapping communication and local calculations in order to hide waiting times (see for example [19]).
All the previous considerations to improve the performance of the sparse matrix-vector multiplication seek to exploit the structure of the matrix A. Another approach consists in 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)) such that the matrix needs only to be loaded once for several vectors. In Figure 2 in Section 3.2 one can see possible improvements of the node-level performance for the matrix-multiple-vector product of a matrix from quantum mechanics with different block sizes.
This approach also reduces the number of messages sent compared to multiple consecutive matrix-single-vector multiplications. Obviously, we can also combine this approach with appropriate matrix reorderings.

So it may be beneficial to modify a given algorithm such that several matrix-vector multiplications with the same matrix are grouped together.

There are essentially two kinds of vector-vector operations: dot-products and operations consisting of simple addition and scaling with a scalar ($y \leftarrow \alpha x + \beta y$, $\alpha, \beta \in \mathbb{C}$). The latter are usually cheap compared to matrix-vector products and perfectly parallelizable and thus quite unproblematic. To calculate the dot-product of distributed vectors we need a reduction operation over all involved processes, which may produce significant communication overhead on larger numbers of nodes (through synchronizing all processes).
The performance can still be improved by grouping together several similar operations: Block vector operations allow to reduce the number of messages sent and also to employ faster dense matrix operations.

Especially in the field of sparse eigensolvers it may be easy to extend existing algorithms that determine one eigenvector after another to block algorithms that search for a set of eigenvalues and -vectors at once. This is also interesting from a numerical point of view since subspace methods usually gather information for eigenvalues near a specific target eigenvalue automatically. So it may be helpful to investigate block methods in order to calculate several close eigenvalues.

3.2. Node-level performance modelling and micro benchmarks

We can set up models for the performance of a given algorithm on a given piece of hardware. These models are useful to obtain a better understanding of the behavior of the algorithm on the system involved, and they allow us to evaluate the performance of the implementation. A very simple model is the *roofline* model. It states that the attainable performance of a given (part of an) algorithm is limited by the peak performance of the system and the required bandwidth:

$$P_{ideal} = \min\left(P_{peak}, \frac{b_{data}}{B_{algorithm}}\right). \tag{9}$$

Here P_{ideal} denotes the attainable performance, P_{peak} the peak performance of the system on the assumption that all data are already available in the fastest cache and b_{data} the bandwidth of the slowest data transfer path utilized. The term $B_{algorithm}$ is the *code balance* of the algorithm. It specifies required data transfers (stores and loads) per floating point operation (flop).
In this model we ignore effects of the caching system and the parallel execution. The real performance can be much better when (most of) the required data is already in the cache or also much worse when the data needs to be loaded multiple times to the cache. As the cache is organized in lines and its size is limited this may happen for irregular data accesses (e.g. only part of a line is used at once). Also possibly required synchronization between several compute units may slow down the real performance (or unintended synchronization due to false sharing in multicore systems). There are also other more sophisticated performance models (see for example [23]), but these are

not considered in this thesis. The roofline model already helps us to understand the effects we are interested in.

3.2.1. Sparse matrix-multiple-vector multiplication

Schubert et al. show in [19] that we can still learn something from this simple model for the sparse matrix-vector multiplication. They introduce a parameter that incorporates additional data transfers due to the limitations of the cache system.
The following analysis is based on [19], but I want to introduce two small modifications: Firstly, we can employ *non-temporal stores* (see [5, section 1.3] for a short explanation) for the left-hand side vector. This means that it is not necessary to load the vector (from the main memory to the cache), but we can directly write the data to the main memory. Please note that this is not possible if one wants to compute for example $y \leftarrow Ax + \beta y, \beta \in \mathbb{R}$ or in the case of calculating the spMVM of a distributed matrix with overlapping calculation and communication.
And secondly, I want to show the effect of performing multiple sparse matrix-vector multiplications at once with different right-hand side vectors. So I obtain the following algorithm:

Algorithm 1 Sparse matrix-multiple-vector multiplication $Y \leftarrow AX$ with a matrix in the CRS format

```
1 do i = 1, nrows
2   tmp(:) = 0
3   do j = row_ptr(i), row_ptr(i+1)-1
4     tmp(:) = tmp(:) + val(j) * X(:,col_idx(j))
5   end do
6   Y(:,i) = tmp(:)
7 end do
```

Here we assume that the matrix is stored in CRS format (for a description of the CRS format see for example [18, chapter 2]). To simplify the analysis, I only consider the representative case of real valued matrices and vectors with calculations in double-precision. The code balance is composed of $2n_b$ flops per inner iteration, 12 bytes to load the matrix (8 for `val` and 4 for `col_idx`) and $16\frac{n_b}{n_{nzr}}$ to load X and store Y. Without non-temporal stores we would obtain $24\frac{n_b}{n_{nzr}}$ here.

$$\begin{aligned} B_{CRS\_NT} &= \frac{12 + 16\frac{n_b}{n_{nzr}} + n_b\kappa(n_b)}{2n_b} \\ &= \frac{6}{n_b} + \frac{8}{n_{nzr}} + \frac{\kappa(n_b)}{2} \qquad \left[\frac{bytes}{flops}\right]. \end{aligned} \tag{10}$$

Here n_b denotes the *block size*, e.g. the number of vectors $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}^n$ in $X = \left(x_1, \ldots, x_{n_b}\right)$ and $Y = \left(y_1, \ldots, y_{n_b}\right)$. The term n_{nzr} is the average number of non-zero entries in each row of the matrix. Additional loads due to irregular accesses

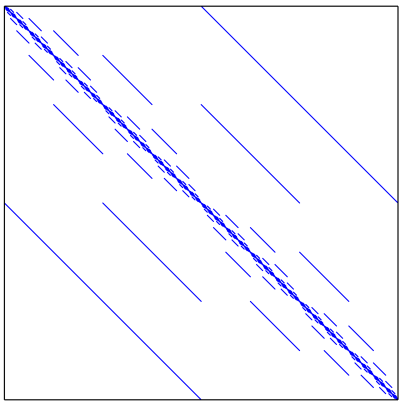


Figure 1: Sparsity pattern of the spin20 matrix. It has $n_r = 10^6$ rows and in each row there are $n_{nzr} \approx 10$ entries.

to X are represented by $n_b\kappa(n_b)$; the parameter κ depends obviously not only on the sparsity pattern of the matrix and the hardware used but also on the block size.

As test case here the matrix spin20 from the ESSEX project is used (see Table 6 in Section 7.1). Figure 1 illustrates the sparsity pattern of the matrix. Its dimension is approximately $10^6 \times 10^6$ with ~ 10 non-zero entries per row. The results in [19] for other matrices on similar hardware show values for $\kappa(1)$ in the order of $2 - -4$. Obviously, the code balance B_{CRS_NT} is greater than one such that the algorithm is inevitably memory bounded on present architectures (see [5, Figure 3.2] for an overview of peak performance vs. peak bandwidth). Increasing the block size n_b (say from 1 to 4) greatly improves the code balance if the parameter $\kappa(n_b)$ remains fixed. In the first row of Table 1 we see the ideal code balance for $\kappa = 0$ for different block sizes n_b. As in [19] I have measured the obtained bandwidth and performance to estimate κ. The results for different numbers of right-hand side vectors and different implementations of the spMMVM are shown in Figure 2: The first two variants use a row-major storage scheme for X and Y, in GHOST and Trilinos (see Section 5) the vectors are stored in column-major order. The STREAM Triad benchmark was used to determine the effective memory bandwidth (for more information, please refer to [5, section 1 of chapter 3]). We can see that the row-major variants achieve about 90% of the STREAM Triad bandwidth. For the column-major implementation in GHOST, the bandwidth drops significantly for $n_b > 1$. I need to mention that in the version of the GHOST library used here the developers have not yet tried to optimize the performance of multiple-vector operations.
For the Trilinos implementation I did not measure the bandwidth, but the performance only increases for $n_b = 2$ and then decreases slightly for bigger block sizes.
We can also see that the achieved performance increases significantly for the row-major variants when we increase the block size. For bigger block sizes we observe

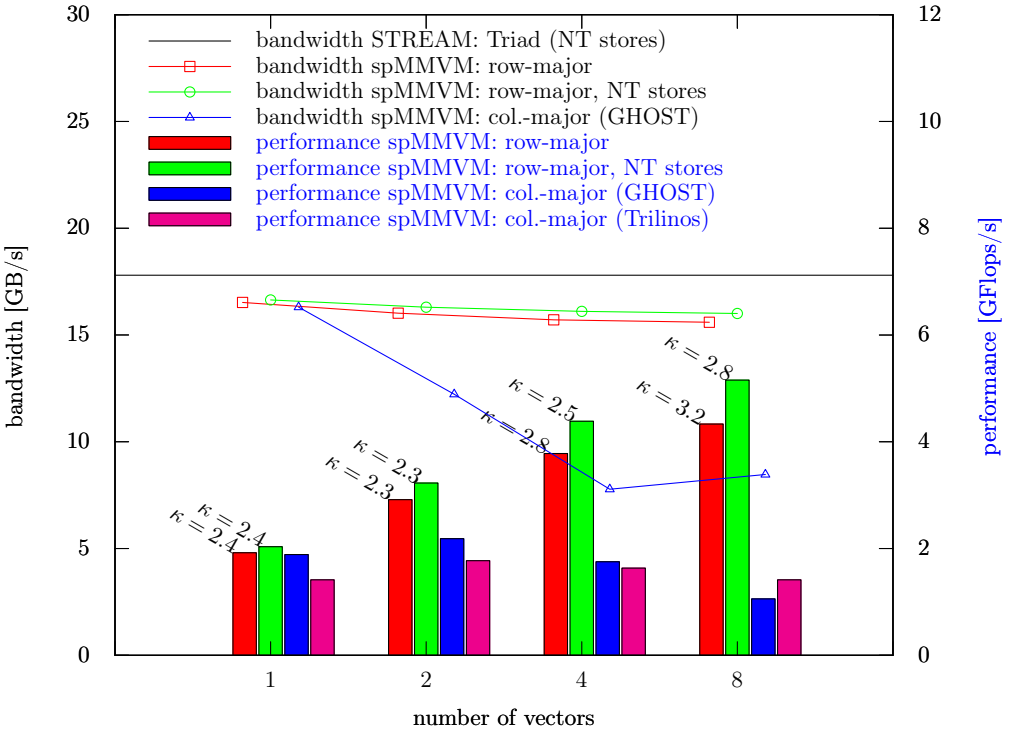


Figure 2: Single-node performance of the sparse matrix-multiple-vector multiplication for different block sizes using the spin20 matrix on an Intel Westmere EP (1×Xeon E5645 with 6 cores at 2.4 GHz). Effective STREAM Triads bandwidth and measured bandwidth of the spMMVM operation is also shown.

block size n_b	1	2	4	8	12	16
ideal code balance $B_{CRS}(\kappa = 0)$	6.8	3.8	2.3	1.55	1.3	1.175
measured bytes per flop B_{CRS}	8.0	5.0	3.6	3.0	2.8	2.8
estimated parameter κ	2.4	2.3	2.5	2.8	3.1	3.2

Table 1: Ideal and measured code balance of the spMMVM with non-temporal stores for the spin20 test case.

also more additional memory traffic per vector due to the non-consecutive memory accesses (parameter κ). The results in Table 1 suggest that we may expect no further performance improvements for $n_b = 12$ and $n_b = 16$. Since the cache line size of the system used is 64 bytes, with $n_b = 8$ one row of X exactly fits into one cache line. This could explain these observations. As the parameter κ strongly depends on the sparsity pattern of the matrix, one could observe different behavior for other matrices.
It is also worth noting that the performance gain through the usage of non-temporal stores grows with the block size, as we would expect it from the code balance. The parameter κ is also smaller in this case; this could indicate that without non-temporal stores cache-lines containing parts of X used again later are discarded to load Y.

From this test case we can draw the following conclusions:
Blocking several vectors together can greatly improve the spMVM performance, mainly due to the better code balance (matrix only loaded once for several vectors). However, the storage scheme is important; for the column-major implementations in this test the performance even decreases. As this seems to go hand in hand with a decrease of the achieved memory bandwidth, a more careful implementation could probably lead to better results.
Additionally, I want to note that the row-major storage scheme obviously implies severe performance penalties in a complete eigensolver algorithm when only one vector from a block of vectors is accessed in some operations. An interesting compromise between both storage schemes could be a scheme where a larger block of multiple vectors is stored as a set of column-major smaller subblocks, but this would need further investigation that is not possible in the scope of this thesis.

3.2.2. Multiple-vector operations

I also want to explain shortly why blocking multiple vectors together may also be advantageous for vector-vector operations. As we will see later, the following operation is needed for the Jacobi-Davidson algorithm as part of orthogonalizations and projections:

Algorithm 2 Projection with multiple vectors $Y \leftarrow (I - QQ^T)Y$

```
1 m(:,:) = matmul(transpose(Q(:,:)),Y(:,:))
2 Y(:,:) = Y(:,:) - matmul(Q(:,:),m(:,:))
```

For simplicity we just consider the hopefully representative case $Q = \left(q_1, \ldots, q_8\right), q_i \in \mathbb{R}^n$ and $Y = \left(y_1, \ldots, y_{n_b}\right), y_i \in \mathbb{R}^n$ for different block sizes n_b. For the case $n_b = 1$ we have two consecutive dense matrix-vector operations (BLAS2), whereas for $n_b > 1$ these are two dense matrix-matrix operations (BLAS3). The ideal code balance for the first part of the projection $m \leftarrow Q^T Y$ in double-precision is given by

$$B_{W^T V} = \frac{(8 + n_b)8}{8n_b \cdot 2} = \frac{4}{n_b} + \frac{1}{2}. \tag{11}$$

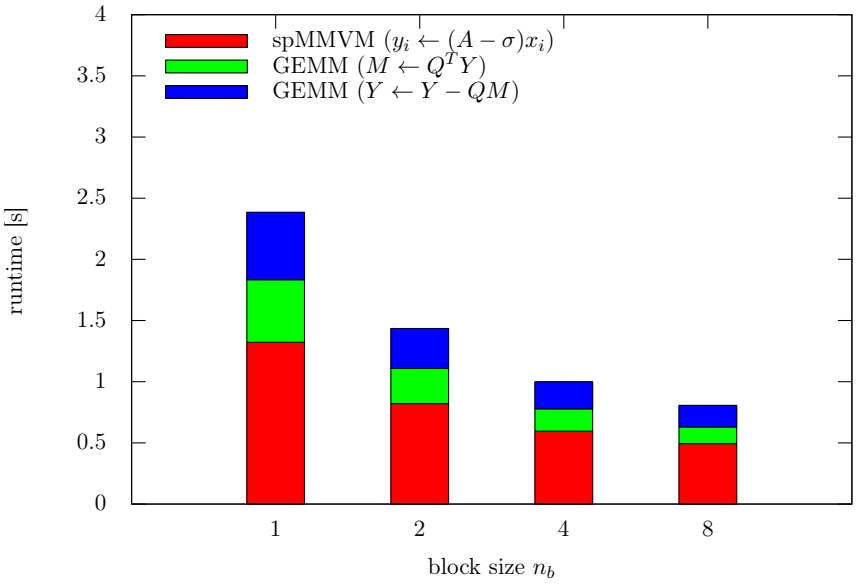


Figure 3: Required runtime for $120/n_b$ applications of the Jacobi-Davidson operator $(y \leftarrow (I - QQ^T)(A - \sigma I)x)$ with shifts $\sigma \in \mathbb{R}$ and 8 projection vectors $(q_1, \ldots, q_8) = Q$ for different block sizes n_b using the spin20 matrix on an Intel Westmere EP.

Here I assume that n is very large while n_b is small; so that only the memory transfers required to load Q and Y are relevant, which gives $(8 + n_b)8$ bytes per row. And we need to multiply each number in a row of Q with each number in the corresponding row of Y, and then add the result to m, such that one obtains $8_n b \cdot 2$ operations per row.
We can directly see that the performance is memory bounded for huge n and that it increases significantly when we choose a block size $n_b > 1$. Compared to multiple consecutive operations with single vectors y_i, we do not have to load Q several times. This is also similar for the second part of the projection $Y \leftarrow Y - Qm$ which has the following ideal code balance in double-precision:

$$B_{Vm} = \frac{(8 + 2n_b)8}{8n_b \cdot 2} = \frac{4}{n_b} + 1. \tag{12}$$

The only difference consists in the fact that we modify Y here, and thus we need to transfer it back to the main memory.
The code balance demonstrates how the performance of both operations increases for a given small block size n_b. This is nicely illustrated in Figure 3. For $n_b = 1$ both operations approximately require the same runtime. The runtime decreases as expected when we increase the block size n_b. We can also see that the performance of the first part of the projection becomes slightly faster then the second part for bigger block sizes due to its better code balance. Moreover, Figure 3 also shows how the performance of the spMMVM changes in relation to the vector operations.

3.3. Discussion of the inter-node performance

In many applications the inter-node performance is dominated by the required communication between the nodes. This term refers to several distinct effects that play an important role when the calculation is distributed among the nodes of a computing cluster:
First, the bandwidth of the network is limited and usually considerably smaller than the memory bandwidth within a node.
Additionally, one may incur a significant latency when messages are sent to another node respectively received from another node. This latency does not only include the latency of the network itself (e.g. the traveling time required for the message and overhead of the hardware used), but in most cases the data needs to be copied to a send buffer first, then the communication partners need to exchange a handshake before the data is actually transferred, and afterwards the receiver possibly needs to copy the data from a receive buffer to the desired destination.
Finally, for increasing numbers of nodes synchronization effects lead to bigger and bigger waiting times even if there are only small imbalances in the distribution of the work. Furthermore, for global communication operations latencies of single messages add up. How to compensate or hide this communication overhead using more asynchronous communication patterns is a main challenge of software for future supercomputers. This includes in particular the usage of non-blocking (collective) communication, but also aggregating several small messages into one bigger message, which is a key idea of block methods as the one developed in Section 4.

In the following I address effects of bandwidth and latency occurring on small to medium numbers of nodes. All tests in this section are performed on the LiMa cluster of the RRZE[3]. The nodes of the LiMa cluster are connected by an InfiniBand network, and each node consists of two Intel Xeon 5650 Westmere processors.
Figure 4 illustrates the effective communication bandwidth between several (pairs of) nodes when they exchange messages of different size using MPI. Here we see that latency effects dominate for small messages. Only for large messages with sizes of at least 0.5 Mb the bandwidth limit of about 5 Gb/s is reached.

3.3.1. Sparse matrix-multiple-vector multiplication

These results helps us to characterize the communication behavior of the sparse matrix-multiple-vector multiplication. Table 2 presents the estimated average size of data required to be communicated on each node for the matrices spinSZ22 and spinSZ26 presented in Section 7.1. For these results a suitable reordering of the matrix was calculated using ParMETIS [11], which seeks to minimize the communication by determining a graph partitioning with small edge-cut. The numbers shown in Table 2 are directly calculated from the global edge-cut (that is the global number of vector elements that must be communicated) and thus only represent estimated averages. Additionally, no information was gathered about the actual communication pattern

[3]The computing center of the university of Erlangen-Nürnberg (`http://www.rrze.uni-erlange.de`)

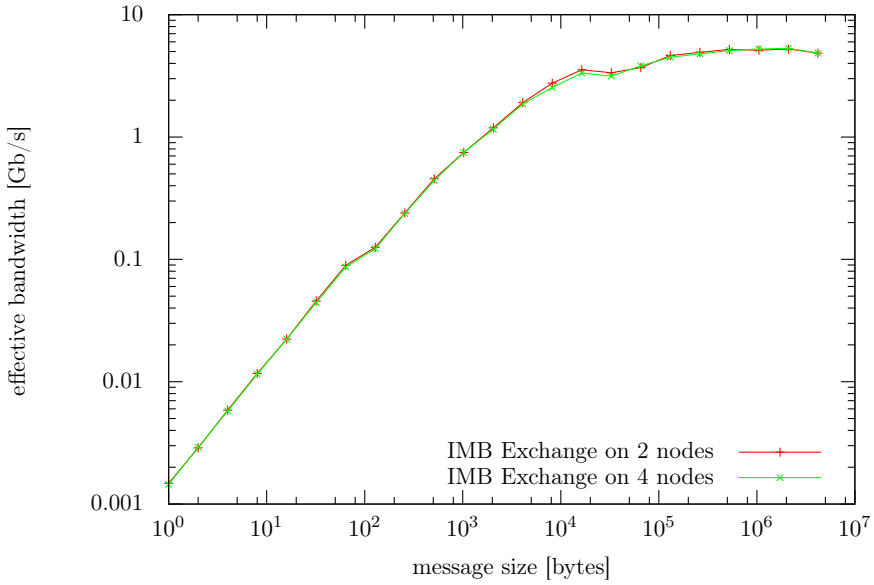


Figure 4: Measured communication bandwidth for different message sizes on two/four nodes of the RRZE's LiMa cluster obtained with the Intel MPI Benchmarking suite [9]. In the Exchange benchmark each process exchanges data with a left and a right neighbor using non-blocking MPI operations. That means the reported bandwidth includes both ingoing and outgoing messages of a node.

between the nodes.

matrix	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
spinSZ22	$\sim$ 1.4Mb	$\sim$ 1.3Mb	$\sim$ 1.0Mb	$\sim$ 0.7Mb	$\sim$ 0.4Mb	
spinSZ26	$\sim$ 19.8Mb	$\sim$ 21.5Mb	$\sim$ 14.5Mb	$\sim$ 9.4Mb	$\sim$ 6.1Mb	$\sim$ 3.7Mb

Table 2: Average data size per node that must be communicated (e.g. sent to other nodes or received from other nodes) for each sparse matrix-vector multiplication for the two matrices spinSZ22 and spinSZ26. This is the size of the buffers required to send and receive the vector elements. To reduce communication overhead a suitable distribution was calculated using ParMetis[11].

Still, we can draw several interesting conclusions from Table 2: For a small number of nodes the message size lies definitely in the range where the full communication bandwidth can be achieved. For the smaller matrix spinSZ22 this is the case for 2 and possibly 4 nodes depending on the actual communication pattern, as the message sizes for different communication partners of one node add up to the value shown in Table 2. For a distribution onto 8 and more nodes the individual messages are most probably too small to achieve the full bandwidth.
For the larger matrix spinSZ26 more vector elements need to be communicated for each

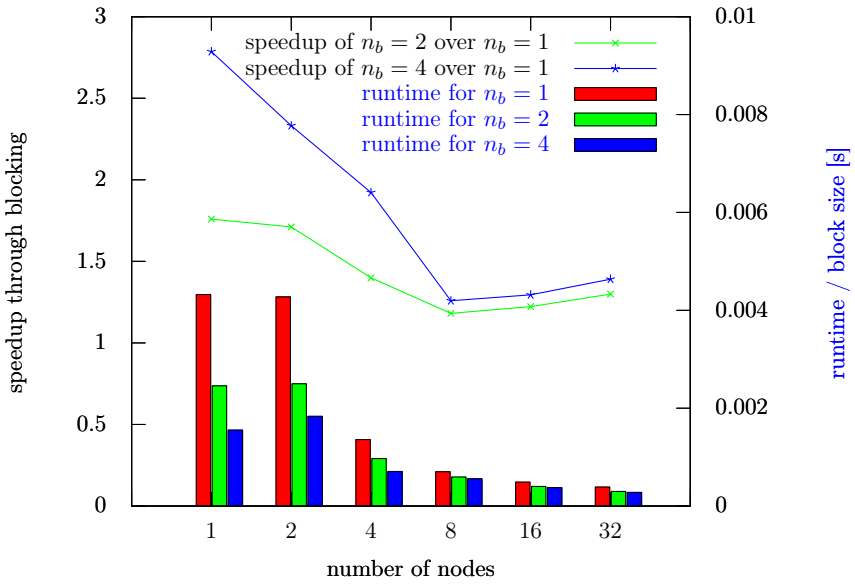


Figure 5: Runtime divided by the block size (1, 2 and 4) of the sparse matrix-multiple-vector multiplication of the matrix spinSZ22 on 1-64 nodes of the LiMa cluster. Additionally, the relative speedup of the block variants in relation to the single vector computation is shown. All calculations were performed using a row-major storage scheme for blocks of vectors.

matrix-vector multiplication. So obviously for at least up to 8 nodes the message sizes are in average large enough to obtain nearly the limit bandwidth. Since the actual communication patterns are not known, the results are not as clear for more than 8 nodes. Here we can only assume that the average message sizes may become smaller than ~ 0.5 Mb depending on the number of communication partners of the individual nodes.

As long as the full network bandwidth can be achieved by a matrix-single-vector multiplication, we cannot improve the performance of the communication through the usage of block vectors. The calculation itself is still faster. However, the amount of local work scales with the number of nodes, that means it decreases by approximately 50 % if the number of nodes is doubled. In contrast, the required amount of communication per node decreases only moderately (if it decreases at all) with the number of nodes. Hence, the speedup that can be obtained by using a bigger block size in the spMMVM decreases with the number of nodes at least until the messages in the single vector calculation become too small to achieve the full bandwidth.

This describes very well what we can observe in Figure 5 and Figure 6: Here the bars show the required runtime for each single vector computation (that is the runtime of the block operation divided by the block size). In all tests the block variants are faster than the respective single vector calculations. Interestingly, in Figure 5 the computation on two nodes is not faster than on one node for the block sizes 1 and 2. Moreover,

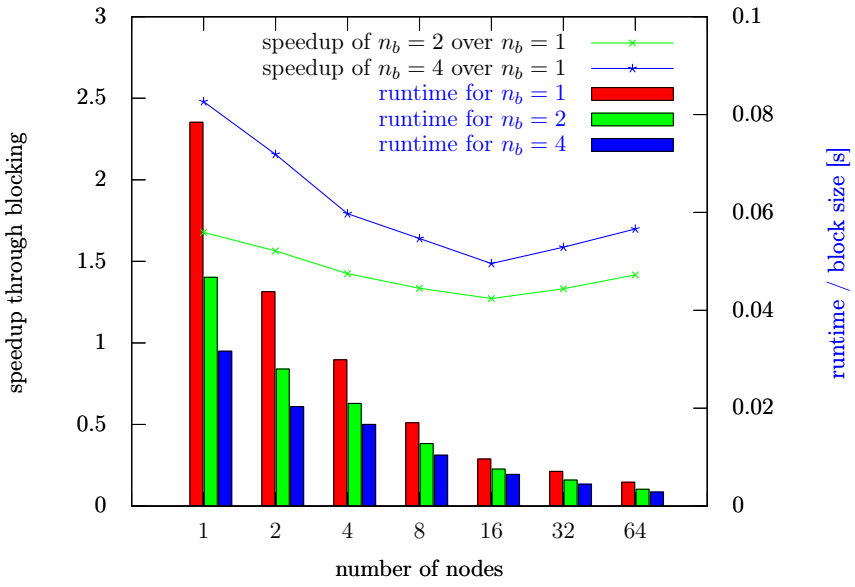


Figure 6: Runtime divided by the block size (1, 2 and 4) of the sparse matrix-multiple-vector multiplication of the matrix spinSZ26 on 1-64 nodes of the LiMa cluster. Additionally, the relative speedup of the block variants in relation to the single vector computation is shown. All calculations were performed using a row-major storage scheme for blocks of vectors.

for a block size of 4 it is slower. Here the communication overhead even outweighs the decreasing amount of local work. However, we can still see that the speedup obtained through a bigger block size decreases significantly until a minimum is reached on 8 nodes; then it increases slowly. In particular, there is no big performance difference between the block sizes 2 and 4 on more than 8 nodes for the matrix spinSZ22.
The tests with the matrix spinSZ26 globally exhibit a similar behavior, but here one gains more performance through the usage of block vectors as the matrix is much larger. The minimum speedup is observed on 16 nodes; however, the calculation is still significantly faster with a block size of 4 than with a block size of 2.

In the tests performed no overlapping of calculation and communication was applied, which is discussed as pure vector-mode in [19] for single vector spMVM. Ideally, one would overlap the communication with other calculations performed before the result of the matrix-vector multiplication is required (but the right-hand size vector must already be available) such that one could possibly profit from the whole speedup of the block operations on the node-level.
Please note that the results shown here represent strong scaling experiments; therefore the parallel speedup is clearly limited due to an increasing amount of communication. Furthermore, simple scalability metrics such as parallel speedup or parallel efficiency are not very helpful here because the block operations are much faster on a single node so that they *scale* worse even though they achieve a higher performance than single vector computations in all cases.

3.3.2. Multiple-vector operations

There is not much to say about multiple-vector operations. Some of the required vector operations are completely local such as $Y \leftarrow Y - Qm$ from Algorithm 2 and thus they are perfectly scalable. Others require all-reductions of a small fixed amount of data. In particular, the message size is independent of the number of nodes here. An example is given by the operation $m \leftarrow Q^T Y$ (also from Algorithm 2). Combining several single vectors in Y increases the message size and reduces the number of global communication operations required. This aggregation of messages should always increase the performance for small message sizes as indicated in Figure 4. Additionally, this may reduce the overhead due to synchronization effects on a large number of nodes.

4. Block Jacobi-Davidson QR method

To return to the topic of sparse eigenproblems, we search for a small set of eigenvalues and eigenvectors at the periphery of the spectrum of a matrix:

$$Av_i = \lambda_i v_i, \qquad i = 1, \ldots, l. \tag{13}$$

The eigenvectors v_i form a basis of the invariant subspace $\mathcal{V} = \text{span}\{v_1, \ldots, v_l\}$, but for general, non-Hermitian matrices the conditioning of this basis may be arbitrarily bad. If we consider an orthonormal basis of the invariant subspace, we obtain the following formulation of the problem:

$$\begin{cases} AQ - QR & = 0, \\ -\frac{1}{2}Q^*Q + \frac{1}{2}I & = 0. \end{cases} \tag{14}$$

Here $AQ = QR$ denotes a partial Schur decomposition of the matrix 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 [26] for a discussion of different approaches for single vectors.

4.1. Derivation of a block correction equation

We can now try applying a Newton scheme to the non-linear system of equations above, which will lead us to a block Jacobi-Davidson style QR algorithm (see [4]):
First, we write the above equations as corrections ΔQ and ΔR for existing approximations $\tilde{Q}$ and $\tilde{R}$:

$$\begin{cases} A(\tilde{Q} + \Delta Q) - (\tilde{Q} + \Delta Q)(\tilde{R} + \Delta R) & = 0, \\ -\frac{1}{2}(\tilde{Q} + \Delta Q)^*(\tilde{Q} + \Delta Q) + \frac{1}{2}I & = 0. \end{cases} \tag{15}$$

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

$$\begin{cases} A\Delta Q - \Delta Q\tilde{R} & \approx -(A\tilde{Q} - \tilde{Q}\tilde{R}) + \tilde{Q}\Delta R, \\ \frac{1}{2}\tilde{Q}^*\Delta Q + \frac{1}{2}\Delta Q^*\tilde{Q} & \approx 0. \end{cases} \tag{16}$$

The second equation indicates that the term $\tilde{Q}^*\Delta Q$ must be skew-Hermitian. However, in a subspace iteration we are only interested in the part of ΔQ perpendicular to $\tilde{Q}$ because only these directions enlarge the subspace. So we can split the correction into two parts: its projection onto $\tilde{Q}$ and its projection onto the orthogonal complement of $\tilde{Q}$:

$$\Delta Q = \underbrace{\tilde{Q}\tilde{Q}^*\Delta Q}_{\Delta Q^{\parallel}} + \underbrace{(I - \tilde{Q}\tilde{Q}^*)\Delta Q}_{\Delta Q^{\perp}} . \tag{17}$$

In order to improve the conditioning of the linear problem and to reduce its dimension we can also use the projection of the matrix A onto the orthogonal complement of the current approximation $\tilde{Q}$:

$$\begin{aligned} & A^{\perp} := (I - \tilde{Q}\tilde{Q}^*)A(I - \tilde{Q}\tilde{Q}^*) \\ \Leftrightarrow \qquad & A = A^{\perp} + \tilde{Q}\tilde{Q}^*A + A\tilde{Q}\tilde{Q}^* - \tilde{Q}\tilde{Q}^*A\tilde{Q}\tilde{Q}^*. \end{aligned} \tag{18}$$

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

$$\begin{aligned} A^{\perp}\Delta Q^{\perp} - \Delta Q^{\perp}\tilde{R} \approx & -(A\tilde{Q} - \tilde{Q}\tilde{R}) - (I - \tilde{Q}\tilde{Q}^*)A\Delta Q^{\parallel} \\ & + \tilde{Q}\tilde{Q}^*(A\Delta Q - \Delta Q\tilde{R} + \tilde{Q}\Delta R). \end{aligned} \tag{19}$$

The first term on the right-hand side is the residual of the current approximation. All terms in the first line of the equation are orthogonal to $\tilde{Q}$ if the current approximation fulfills a Galerkin condition (e.g. the residual is orthogonal to $\tilde{Q}$). So the second line must vanish in this case.
Additionally, on this assumption we can also express the second term on the right-hand side using the residual:

$$\begin{aligned} (I - \tilde{Q}\tilde{Q}^*)A\Delta Q^{\parallel} &= (I - \tilde{Q}\tilde{Q}^*)A\tilde{Q}\tilde{Q}^*\Delta Q \\ &= (I - \tilde{Q}\tilde{Q}^*)\left((A\tilde{Q} - \tilde{Q}\tilde{R}) + \tilde{Q}\tilde{R}\right)\tilde{Q}^*\Delta Q \\ &= (A\tilde{Q} - \tilde{Q}\tilde{R})(\tilde{Q}^*\Delta Q) \end{aligned} \tag{20}$$

Near the solution both the residual as well as the correction are small (and $\tilde{Q}$ is orthogonal), so this presents a second order term that we simply neglect in the following.

The result is a Jacobi-Davidson style correction equation for an approximate Schur form (for simplicity I only write ΔQ for $\Delta Q^{\perp}$ in the following):

$$(I - \tilde{Q}\tilde{Q}^*)A(I - \tilde{Q}\tilde{Q}^*)\Delta Q - (I - \tilde{Q}\tilde{Q}^*)\Delta Q\tilde{R} = -(A\tilde{Q} - \tilde{Q}\tilde{R}). \tag{21}$$

This is essentially a Sylvester equation for $\Delta Q^{\perp}$.
For the Hermitian case and thus diagonal $\tilde{R}$ it simplifies to the following set of independent Jacobi-Davidson correction equations (with $\tilde{\lambda}_i = \tilde{r}_{i,i}$):

$$(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)(I - \tilde{Q}\tilde{Q}^*)\Delta q_i = -(A\tilde{q}_i - \tilde{\lambda}_i\tilde{q}_i), \qquad i = 1, \ldots, l. \tag{22}$$

As $\tilde{R}$ is upper triangular, we can also write (21) as a set of correction equations with a modified residual for general matrices:

$$(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)(I - \tilde{Q}\tilde{Q}^*)\Delta q_i = -(A\tilde{q}_i - \tilde{Q}\tilde{r}_i) - \sum_{j=1}^{i-1}\tilde{r}_{j,i}\Delta q_j, \quad i = 1, \ldots, l. \tag{23}$$

With this formulation we need to solve the correction equations successively for $i = 1, \ldots, l$. 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 $\sum_{j=1}^{i-1} \tilde{r}_{j,i}\Delta q_j$ in the residual:

$$(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)(I - \tilde{Q}\tilde{Q}^*)\Delta q_i = -(A\tilde{q}_i - \tilde{Q}\tilde{r}_i), \qquad i = 1, \ldots, l. \tag{24}$$

As we can see from the following argument, this uncoupled formulation should also provide suitable corrections Δq_i for a subspace iteration:
The classical JDQR from [4] uses the correction equation

$$(I - \bar{Q}\bar{Q}^*)(A - \tilde{\lambda}I)(I - \bar{Q}\bar{Q}^*)\Delta q = -(I - Q_k Q_k^*)(A\tilde{q} - \tilde{\lambda}\tilde{q}). \tag{25}$$

Here $\bar{Q}$ is composed as $\bar{Q} = \begin{pmatrix} Q_k & \tilde{q} \end{pmatrix}$ and $Q_k = \begin{pmatrix} q_1 & \ldots & q_k \end{pmatrix}$ denotes an orthogonal basis of all previously converged eigenvectors. So we can interpret this equation as the Jacobi-Davidson correction equation for a single eigenvalue with a deflation of already computed eigenpairs.
It is quite similar to a single correction Δq_i from the set of equations (24). The difference lies in the fact that we use a deflation of eigenvector approximations that have not converged yet. The residuals of the two formulations are also related in this sense: In the classical JDQR formulation we need to orthogonalize the residual $A\tilde{q} - \tilde{\lambda}\tilde{q}$ with respect to Q_k, in the formulation $A\tilde{q}_i - \tilde{Q}\tilde{r}_i$ 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.

4.1.1. Generalized eigenproblems

We can use a similar approach for the special generalized eigenvalue problem (3) if we require Q to be B-orthogonal (see Section 2.2). The resulting uncoupled block correction equation (corresponding to (24)) is:

$$(I - (B\tilde{Q})\tilde{Q}^*)(A - \tilde{\lambda}_i B)(I - \tilde{Q}(B\tilde{Q})^*)\Delta q_i = -(A\tilde{q}_i - B\tilde{Q}\tilde{r}_i), \quad i = 1, \ldots, l. \tag{26}$$

4.1.2. Relation with RQI methods

The Newton approach presented here may not explain the convergence behavior of Jacobi-Davidson methods very well: in particular for the case of multiple eigenvalues the Jacobian of the system (14) can be singular. More precisely, this is the case for a partial Schur decomposition (Q, R) when the subspace $\text{span}\{q_1, \ldots, q_l\}$ does not contain the complete invariant subspace of a multiple eigenvalue but only part of it. See [26] for a more detailed explanation for the single-vector case. This means that the conditions for the quadratic convergence rate of the Newton method are not always met (Jacobian singular at solution).
Based on theoretical considerations and numerical results [27] illustrates that one can explain the fast convergence of Jacobi-Davidson methods better by their relation to the Rayleigh quotient iteration: in each iteration the Jacobi-Davidson correction expands the subspace in such a way that it contains the RQI direction $x_{RQI} = (A - \rho_A(\tilde{q})I)^{-1}\tilde{q}$.

In the following I want to show how one can transfer this relation to the block correction equation presented here:
Absil et al. propose a generalization of the RQI method to invariant subspaces with cubic convergence for Hermitian matrices in [1]. First, they define the *matrix Rayleigh quotient* $R_A \in \mathbb{C}^{l\times k}$ as

$$R_A(\tilde{Q}) = (\tilde{Q}^*\tilde{Q})^{-1}\tilde{Q}^*A\tilde{Q}. \tag{27}$$

Then 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:

$$AX_{GRQI} - X_{GRQI}R_A(\tilde{Q}) = \tilde{Q}. \tag{28}$$

Obviously, for an approximate partial Schur decomposition $(\tilde{Q}, \tilde{R})$ that fulfills a Galerkin condition we obtain $\tilde{R} = R_A(\tilde{Q})$. On certain assumptions the block correction equation (21) also leads to corrections with $X_{GRQI} \in \text{span}\{q_1, \ldots, q_l, \Delta q_1, \ldots, \Delta q_l\}^l$ (see Section A.1).
So if the block correction equation (24) is solved exactly in a subspace method, we may also 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).
We can gain some insight about the effect of inexact solutions on the convergence from the theoretical analysis in [15] where Notay shows for a special case that the fast convergence is preserved even with approximate corrections on the assumption that we increase the required accuracy of the corrections in the outer iteration fast enough.

4.2. Complete algorithm

In the following I discuss shortly all the computational steps that are required in the complete block Jacobi-Davidson algorithm.

4.2.1. Subspace iteration

As mentioned before we do not employ the block correction equation (24) directly, but rather consider the resulting corrections Δq_i as hopefully useful directions to enlarge the search space $\mathcal{W}$. This yields Algorithm 3.

We start with a set of vectors T and an empty search space. In each iteration of the main loop (line 2 to 24) we need to orthogonalize the new directions in T with respect to the current search space $\mathcal{W}$. For simplicity we assume here that we abort for the case $T \in \mathcal{W}$; in Section 4.2.3 this is discussed in more detail. Then we add the new directions in T to the subspace (line 10).
Additionally, we need to update the projection of the matrix A onto the subspace $\mathcal{W}$ (line 8 and 9). For this we also need AW where W denotes the orthogonal basis of $\mathcal{W}$. Therefore we also need to store $W_A = AW$ in the case of a non-symmetric matrix A. This can be avoided in the symmetric case where we could just update H using W, $\tilde{T}$ and $\tilde{T}_A = A\tilde{T}$.

Algorithm 3 simple block Jacobi-Davidson QR

Input: matrix $A \in \mathbb{C}^{n \times n}$, start block vector $T \in \mathbb{C}^{n \times l}$, desired accuracy eps, $maxIter$
Output: approximative partial Schur decomposition $AQ \approx QR$

$H \leftarrow (\,)$, $W \leftarrow (\,)$, $W_A \leftarrow (\,)$, $k \leftarrow 0$ ▷ initialize search space: $\mathcal{W} \leftarrow \emptyset$

for $nIter \leftarrow 1, maxIter$ **do** ▷ main iteration loop
  Orthogonalize T wrt. $\mathcal{W}$
  $\tilde{l} \leftarrow \text{rank}(T)$, $\tilde{T} \leftarrow \begin{pmatrix} t_1 & \dots & t_{\tilde{l}} \end{pmatrix}$
  if $\tilde{l} = 0$ **then**
    abort ▷ no valid new directions found
  end if
  $\tilde{T}_A \leftarrow A\tilde{T}$
  $H \leftarrow \begin{pmatrix} H & W^*\tilde{T}_A \\ \tilde{T}^*W_A & \tilde{T}^*\tilde{T}_A \end{pmatrix}$ ▷ update projection of A onto $\mathcal{W}$
  $W \leftarrow \begin{pmatrix} W & \tilde{T} \end{pmatrix}$, $W_A \leftarrow \begin{pmatrix} W & \tilde{T}_A \end{pmatrix}$, $k \leftarrow k + \tilde{l}$ ▷ enlarge search space

  Compute Schur form $HQ_H = Q_H R_H$, with the eigenvalues on the diagonal of R_H sorted by modulus

  $R \leftarrow R_{H\,1:l,1:l}$, $Q \leftarrow WQ_{H\,1:l}$ ▷ calculate approximate schur form of A
  if $||Aq_i - Qr_i||_2 < eps,\ i = 1, \dots, l$ **then**
    return (Q, R)
  end if

  for $i \leftarrow 1, l$ **do** ▷ Calculate corrections
    if $||Aq_i - Qr_i||_2 < eps$ **then**
      $t_i \leftarrow 0$
    else
      Solve approximately $(I - QQ^*)(A - r_{i,i}I)(I - QQ^*)t_i = -(Aq_i - Qr_i)$
      $t_i \leftarrow (I - QQ^*)t_i$
    end if
  end for
end for

abort ▷ no convergence after $maxIter$ iterations

The next step (line 12) consists in computing the Schur form of the projection matrix H. To obtain the desired eigenvalues at the periphery of the spectrum, the Schur form must be sorted appropriately. Here we can also choose which eigenvalues we want to compute: possible examples are the largest (smallest) real eigenvalues or largest (smallest) eigenvalues in modulus.
From the sorted Schur form and the basis vectors of the search space we obtain the current approximate partial Schur decomposition of A (line 12). Afterwards, we determine the current residuals $Aq_i - Qr_i$. If their norms are smaller than a certain threshold, the eigensolver algorithm is terminated (line 14). If not, we need to determine new corrections t_i for those eigenvalues that have not yet converged.
This is where we need to compute an approximate solution of the block Jacobi-Davidson correction equation (line 20).

This variant is still quite simple; in particular, we assumed that the block size is equal to the number of desired eigenvalues. A more sophisticated algorithm is presented in Section 4.3, but first I want to discuss two points that were still left open: the block orthogonalization and the approximate solution of the block correction equation.

4.2.2. Approximate solution of the correction equation

As we need only to approximate the solution of the (block) Jacobi-Davidson equation in each outer iteration, it is a promising approach to use an iterative scheme here. For general matrices both BiCGStab- and GMRES-methods have been successfully employed in practical implementations (see for example [27]). For Hermitian matrices one can also use special iterative linear solvers for indefinite Hermitian problems (even if the matrix is hpd, the correction equation is probably indefinite due to the shifts). A sophisticated stopping criterion that allows to abort the inner iteration early and still makes progress in the outer iteration is presented in [7]. Additionally, it may be useful (or even necessary) depending on the conditioning of the matrix A to use a preconditioner for the inner solver. Here I just want to note that this preconditioner must be suited for different shifts $\tilde{\lambda}_i$ and it needs to work on the projected subspace $(I - \tilde{Q}\tilde{Q}^*)$. It is also a common approach to construct a preconditioner for a fixed shift in order to avoid the need of taking care of the possible indefiniteness and (near) singularity of $(A - \tilde{\lambda}_i I)$. Such a technique could also benefit from the better cache performance of the block algorithm.
So we can combine a wide range of different methods for linear systems here; for simplicity I only consider an unpreconditioned GMRES-method.

Obviously, one can avoid the application of the projection at the right side of A in the correction equation here if the Krylov-subspace is kept orthogonal to Q. As the residual vectors $res_i = Aq_i - q_i r_{i,i}$ are already orthogonal to Q because of the Galerkin condition, one only needs to choose a starting vector orthogonal to Q (e.g. by simply starting with the zero vector). So we obtain the following form of the block correction equation:

$$(I - QQ^*)(A - r_{i,i}I)t_i = -res_i \qquad\qquad i = 1, \ldots, l.$$

Please note that this is not possible for the correction equation (26) of the generalized eigenproblem (3).

As we need to approximate the solutions of a set of linear problems at once, we could also use a block-GMRES method (or in general a block Krylov-subspace method, see for example [17, chapter 6, section 12]). It is true that the system matrices of the single linear problems may differ in their corresponding shifts $\tilde{\lambda}_i$. Even so, this does not lead to different Krylov-subspaces (shift invariance). This approach may offer some interesting possibilities: for example one could directly construct a set of orthogonal directions (because all corrections stem from the same subspace) without much additional effort. However, in many cases the number of iterations needed by the single systems in the block correction equation may vary significantly, especially since one wants to choose a different stopping tolerance for each of them. (A system of a badly approximated eigenvalue in the current iteration, e.g. with a big residual norm, does not need to be solved accurately).
So one may probably have more freedom grouping together the operations of several independent — but concurrently executed — GMRES iterations. From the numerical point of view this is the same as if the single systems were solved one after another. I will describe the actual implementation of this approach in Section 5.2.

4.2.3. 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. In [25, section 39 in chapter IX] this is underlined with some nice examples.
The efficient parallel (and accurate!) orthogonalization of a complete 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 and the (iterated) modified Gram-Schmidt algorithms. Working with a complete block T, however, offers the possibility to employ 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 infringe the accuracy of the first step and vice versa.
In [8] 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. This can be used to both optimize cache usage for intra-node performance and communication between nodes.
In combination with a rank revealing technique and an iterated block Gram-Schmidt to orthogonalize the new block T against W one obtains a very fast and robust method (see the discussion about RR-TSQR-BGS in [8]).

4.3. Restart and deflation

In Algorithm 3 I have assumed some unnecessary simplifications: In particular, the block size does not need to be equal to the number of desired eigenvalues. As in the

JDQR algorithm presented in [4] one can use deflation with previously calculated eigenvectors. Additionally, the algorithm can be restarted to avoid calculations with a huge subspace. The resulting algorithm is outlined in Algorithm 4.

Algorithm 4 Main loop of the block JDQR algorithm with restart and explicit deflation.
Local operations are marked in blue and global operations in red.

In-/Output: Search space W, W_A, the projection H and converged partial (Q, R)

1: Compute Schur form $HQ^H = Q^H R^H$
2: Calculate approximate Schur form $(Aq - qr) \perp W$ from Q^H, R^H and its residual
3: Deflation with previously calculated eigenvectors $\widetilde{res} \leftarrow (I - QQ^*)res$
4: Calculate the residual norm $\tilde{\epsilon}_i \leftarrow \|\widetilde{res}_i\|_2,\ i = 1, \ldots, l$
5: **if** some eigenvalues converged $(\tilde{\epsilon}_i < \epsilon_{tol})$ **then**
6: Update (Q, R), W, H and current approximation (q, r)
7: Deflation with previously calculated eigenvectors $\widetilde{res} \leftarrow (I - QQ^*)res$
8: Calculate the residual norm $\tilde{\epsilon}_i \leftarrow \|\widetilde{res}_i\|_2,\ i = 1, \ldots, l$
9: **end if**

10: Shrink search space W if necessary

11: Calculate approximate corrections $(I - \tilde{Q}\tilde{Q}^*)(A - r_{i,i}I)t_i = -\widetilde{res}_i$

12: Orthogonalize t wrt. W

13: Calculate $t_A \leftarrow At$
14: Update projection $H \leftarrow \begin{pmatrix} H & W^*t_A \\ t^*W_A & t^*t_A \end{pmatrix}$
15: Update search space $W \leftarrow \begin{pmatrix} W & t \end{pmatrix}$, W_A

What is new here is the deflation step in lines 3 and 4. After an eigenvalue has converged, the corresponding Schur vector q_i is added as a new column to Q and the upper triangular matrix R is updated using the corresponding column of r and Q^*res (for the upper part of R). If all eigenvalues have been calculated, the algorithm can stop here. If not, it may need to calculate new approximations to the next l eigenvalues and eigenvectors and apply the deflation step again.
Afterwards, we need to shrink the search space $\mathcal{W}$ if its size exceeds a certain threshold. For this we can use the first k_{min} vectors of WQ^H. This can be seen as a *thick restart*. In fact, this step could also be located at other places in the algorithm; when it is performed before new directions are calculated, one does probably not need to allocate additional temporary space for the vectors t_i.
For the Jacobi-Davidson projection the block vector $\tilde{Q}$ in line 11 is built from both Q and q. As explained before we have omitted the right part of the projection here.
The other steps are the same as in Algorithm 3.

4.3.1. Multiple eigenvalues

I want to address some issues concerning multiple eigenvalues because the matrices from the ESSEX project used for the tests in Section 7.1 all exhibit a lot of multiple eigenvalues. In more detail the eigenvalues we are interested in have multiplicities of 1–8.
This may strongly influence the convergence behavior of the Jacobi-Davidson algorithm. A short discussion about the effects of multiple eigenvalues can also be found in [4].
First of all, we can obviously not associate a single specific eigenvector direction with a given eigenvalue. So one cannot expect a Schur vector from the approximated Schur form to converge to a specific Schur vector of an exact solution. Additionally, the method to calculate a sorted Schur form $HQ^H = Q^H R^H$ of the projection H may not produce comparable results in successive iterations. The behavior that I have observed for some applications of my implementation of the classical JDQR method is the following:
The whole invariant subspace of a multiple eigenvalue converged slowly (e.g. $\sum_i ||res_i||_2^2$ decreased), but the norms of single residuals ($||res_i||_2$) sometimes increased or decreased significantly from iteration to iteration.
I want to illustrate why this behavior could strongly delay the convergence with a small example: If we consider a Hermitian matrix with a repeated eigenvalue the algorithm may calculate approximate eigenvectors q_1, q_2 with $||res_1|| >> ||res_2||$ and $\tilde{\lambda}_1 \approx \tilde{\lambda}_2$ in one iteration. Here we would expect to obtain a much better correction from solving the correction equation of q_2 instead that of q_1.

As already suggested in [4] a suitable approach for multiple eigenvalues could be to adjust the block size in accordance to the dimension of the invariant subspace of the desired eigenvalue.
Since we want to choose a specific block size for reasons of performance, in the following I present a simple approach that combines this idea with a fixed block size:
One can try to detect which eigenvalue approximations will probably converge to the same value: in the simplest case one just needs a parameter that specifies a tolerance. If two or several eigenvalue approximations fall inside this tolerance, the Schur form is reordered such that the Schur vectors with the smallest residual norm are listed first inside the corresponding group. This allows the algorithm to lock converged Schur vectors independently of the original ordering. And this helps to set up the correction equations for the approximations with the smallest residual norms even when the current block with a fixed block size only covers part of an invariant subspace corresponding to a multiple eigenvalue.
Additionally, all Schur vectors corresponding to eigenvalues in the given tolerance can be added to $\tilde{Q}$ to project out all (known) approximate directions of the complete subspace. This may also improve the conditioning of the correction equation for several distinct but close eigenvalues.
In practical applications however, it may be difficult to choose a suitable tolerance when one has no previous knowledge about the distribution of the eigenvalues.
Please note that the reordering only works in the Hermitian case where R^H is a diagonal

matrix and thus the residual vectors of the reordered Schur form are just those of the original one after reordering them. For non-Hermitian matrices one could still try to improve the current approximation to the next Schur vector using linear combinations of all Schur vectors of the invariant subspace.

4.4. Avoiding communication

I want to analyze briefly the communication required in the implementation of this algorithm on systems with distributed memory because it may strongly affect the performance. In Algorithm 4 we can see in which operations communication is required (and which are purely local). Here I assume that the sparse matrix and the vectors are distributed onto different nodes whereas the small matrices related to projections are stored redundantly on all nodes.
First of all, we need to calculate the residual, deflate it with the previously determined Schur vectors and calculate its norm (lines 2-4 and 7-8). Transformations of the search space with a small matrix (e.g. $W \leftarrow WQ^H_{:,\dots}$) are local. These are needed to extract a converged eigenvalue (line 6) and to shrink the search space (line 10).
In order to calculate approximate corrections, one obviously needs global communication (line 11), but I want to discuss this in more detail in Section 5.2 and consider only the outer iteration of the Jacobi-Davidson algorithm here.
The RR-TSQR-BGS algorithm used for the orthogonalization (line 12) needs one or several projections of the form W^*T and applications of the TSQR block orthogonalization, which are already essentially designed to avoid communication.
Afterwards, we need to multiply the matrix A with the new directions in t (line 13) and calculate the missing parts of the new projection H (line 14).

If we look carefully, we notice that for several of these steps all required data are also locally available — or can be made available by storing intermediate results of previous operations or regrouping them.
In particular, this is true for the deflation step: We can also make the residual orthogonal to Q if we just keep the converged vectors in the search space (through the Galerkin condition). Then the additional possibly two global operations for the deflation (line 3 and 7) are not needed any more. Obviously, we still need to orthogonalize the new corrections t with respect to Q, but this is simply part of the orthogonalization step in line 12. As locking Schur vectors of converged eigenvalues improves the stability of the method for multiple or tightly clustered eigenvalues (see [20] and the references herein), 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

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

$$R^H = \begin{pmatrix} R_{1:k_{conv},1:k_{conv}} & H_{1:k_{conv},k_{conv}:k}\, q^H \\ 0 & r^H \end{pmatrix} \qquad \text{and} \tag{29}$$

$$H_{k_{conv}:k,k_{conv}:k}\, q^H = q^H r^H \ . \tag{30}$$

Please note that the locking approach described is a standard technique in the field of subspace accelerated eigensolvers, see for example [18, chapter 5]. But I wanted to show this specific formulation to illustrate how one can avoid the additional communication required for the deflation in Algorithm 4.

If an *unpreconditioned* Krylov-subspace method is used to solve the correction equation (line 11), one can possibly also calculate At from intermediate results: For the GMRES method one obtains the solution vector t_i in iteration k from the orthogonal basis V_k of the current Krylov-subspace as:

$$t = V_k y.$$

The basis V_k is calculated using the Arnoldi process

$$\bar{A} V_k = V_{k+1} H_{k+1,k} \ ,$$

where each new vector is orthogonalized to all previous vectors (for more details about the GMRES method please refer to [17, chapter 6, section 5]). Here $\bar{A}$ denotes the Jacobi-Davidson operator $(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)$ for one approximate eigenvalue $\tilde{\lambda}_i$. Obviously, we need to calculate $Av_j, j = 1, \ldots, k$ as part of the application of the operator $\bar{A}$ during the Arnoldi process. If we simply store these intermediate results as $(V_A)_k$, we can directly compute $(t_A)_i := At_i$ locally using

$$(t_A)_i = (V_A)_k y \ . \tag{31}$$

As long as the GMRES subspace is very small, this is probably faster than an additional application of the matrix. And this approach avoids the communication required for the matrix-vector multiplication.
In the Jacobi-Davidson algorithm we still need to orthogonalize the new directions t_i with respect to W:

$$\tilde{t} s_1 = t - W s_2 \ .$$

Here the small matrices $s_1 \in \mathbb{C}^{\tilde{l} \times l}$ and $s_2 \in \mathbb{C}^{k \times l}$ are intermediate results of the orthogonalization. Obviously, we need to apply the same transformations to $t_A := At$:

$$\Rightarrow \qquad \tilde{t}_A s_1 = t_A - W_A s_2 \ . \tag{32}$$

This shows us that we may replace the matrix-vector multiplications for the calculation of $A\tilde{t}$ (line 13) by local calculations.
However, in simple tests with this approach I have encountered some problems with the accuracy of W_A obtained this way, possibly due to a problem in my implementation: The results obtained for t_A from the GMRES iteration were still quite accurate, but $\tilde{t}_A$ showed significant deviations with respect to $A\tilde{t}$ in some cases. There are two possible sources of inaccuracies here: First, we may require two (or several) iterations of the orthogonalization where one needs to sum up s_1 and s_2. Second, one needs to apply the inverse of s_1 which may not be well conditioned.

The columns of W_A together with the orthogonal columns of W, however, are the only representation of the matrix A in the Jacobi-Davidson algorithm (when Aq is directly computed from $W_A Q^H_{:,\dots}$). This means: If we use inaccurate results for $\tilde{t}_A = A\tilde{t}$ to enlarge W_A, this is effectively the same as calculating the eigenvalues of a different matrix $\tilde{A}$. Thus, one may simply observe nice reductions in the residual norms for this case, but the resulting eigenvalue approximations are prone to errors.
So in order to minimize the effects of rounding errors, we should probably apply the additional matrix-vector multiplications for the calculation of $A\tilde{t}$ in each iteration.

With the modifications discussed we obtain Algorithm 5. In line 1 the lower right part

Algorithm 5 Main loop of the communication-optimized block JDQR algorithm. Local operations are marked in blue and global operations in red.

In-/Output: Search space W, W_A and the projection H

1: Update Schur form $HQ^H = Q^H R^H$
2: Update the approximative partial Schur form (Q, R)
and its residual vectors $res = (Q_A - QR) \perp W$
3: Calculate residual norms $\|res_{k_{conv}+i}\|_2, i = 1, \dots, l$

4: Reorder similar eigenvalues in the approximate Schur form by their residual norms
5: Lock converged eigenvalues
6: Shrink search space W if necessary

7: Calculate approximate corrections $(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_{k_{conv}+i} I)t_i = -res_{k_{conv}+i}$
for non-converged directions $\|res_{k_{conv}+i}\|_2 > \epsilon_{tol}$;
may also be used to calculate: $t_A = At$

8: Orthogonalize t wrt. W

9: Update t_A or calculate $t_A \leftarrow At$
10: Update projection $H \leftarrow \begin{pmatrix} H & W^* t_A \\ t^* W_A & t^* t_A \end{pmatrix}$
11: Update search space $W \leftarrow \begin{pmatrix} W & t \end{pmatrix}$, W_A

of the Schur decomposition is calculated according to (30) and the matrices Q^H, R^H are updated with (29).
With the Schur decomposition of the projection and the basis vectors W we obtain the current approximation for the next few eigenvalues and Schur vectors in line 2 with

$$\begin{aligned} q_i &= W Q^H_{:,i}, \\ r_i &= R^H_{1:i,i}, \\ (q_A)_i &= W_A Q^H_{:,i}, \qquad\qquad i = k_{conv}, \dots, k_{conv} + l \\ res_i &= (q_A)_i - Q_{:,1:i} r_i. \end{aligned} \tag{33}$$

As mentioned previously, the residual vectors calculated this way are already orthogonal to previously converged Schur vectors; so no explicit deflation step is needed here.
Afterwards, the residual norms are calculated in line 3, which requires an all-reduce operation.
In line 4 the algorithm tries to detect possibly multiple eigenvalues and sorts these by their respective residual norm. For the symmetric case one does not need to modify q_i and res_i here, but may rather just modify the small matrices R and Q^H and store the new ordering for later.
When the algorithm detects that one or more eigenvalues have converged, it transforms the search space using the following steps:

$$\begin{aligned} W &\leftarrow WQ^H, \\ W_A &\leftarrow W_A Q^H, \\ H &\leftarrow (Q^H)^* H Q^H \qquad \text{and afterwards} \\ Q^H &\leftarrow I. \end{aligned} \tag{34}$$

Then the newly converged Schur vectors and eigenvalues can be marked as locked.
If the search space has reached its maximal dimension in line 6, a similar transformation as the one above is applied in order to obtain a basis of the most useful directions; the difference lies only in the fact that we can discard the right part of W and only keep $(w_1, \ldots, w_{k_{min}})$.
Subsequently, the correction equations are solved approximately in line 7 and the resulting corrections are orthogonalized in line 8 and added to the search space for the next iteration in line 11.
In line 9 we need to multiply the matrix A with the new directions t to obtain t_A if the latter is not computed locally using (31) and (32).
Finally, one needs another all-reduction to calculate the missing parts of the new projected matrix H.

Algorithm 9 in Section B.1 shows the complete block JDQR algorithm for the generalized eigenvalue problem (3).

5. Implementation

I have implemented the block JDQR algorithm as part of a framework currently developed at the DLR for the ESSEX project. As mentioned in the introduction ESSEX is one of the projects of the DFG program SPPEXA, and it seeks to develop concepts and methods for large scale sparse eigenvalue solvers on future exascale computers. The framework has the working title Pipelined Hybrid-parallel Iterative Solver Toolkit, in the following abbreviated with *phist*.
In this section I describe shortly the *phist* framework and address some aspects of my block JDQR implementation.

5.1. The *phist* framework

The core of the *phist* framework is a general linear algebra interface that allows the user to write complex algorithms for distributed memory systems using simple statements. This makes it possible to use the same implementation with different numerical linear algebra libraries; currently one can choose between two kernel libraries: Trilinos and GHOST. Trilinos is a huge object oriented library developed for parallel mathematical and scientific computations at the Sandia National Laboratories [6]. In the *phist* framework one can also choose between the Trilinos packages Tpetra and Epetra; all tests shown in this thesis are performed using the package Tpetra. The *General, Hybrid and Optimized Sparse Toolkit* (GHOST) is a library for large sparse linear algebra computations that is also developed as part of the ESSEX project at the RRZE[4].
Both Trilinos and GHOST allow computations with huge sparse matrices and blocks of huge vectors on distributed systems with multiple threads on each node as well as hybrid calculations using accelerator devices such as GPUs.
Additionally, I have implemented a set of kernel routines for performance tests with the block JDQR algorithm. These achieve a much higher performance in some of my test cases because they use a different memory layout for blocks of vectors (see the previous discussion in Section 3.2).

The *phist* framework itself is written in C (and internally partly in C++) to allow using it from other programming languages as well (respectively make it easy to write bindings for them). And it offers an extensive set of tests that can both be used to check the interface to the underlying kernel library (and the library itself under different parallel configurations) and to check the numerical algorithms implemented in the framework. For a small example of a simple algorithm in the *phist* framework see Algorithm 11 in Section B.2.
In the following I present some aspects of the framework required to understand how the algorithms developed in this thesis are implemented and how these work in a distributed setting.

[4]The computing center of the university of Erlangen-Nürnberg (`http://www.rrze.uni-erlange.de`)

5.1.1. Linear algebra data types

The framework currently supports computations in real and complex arithmetic with single and double precision, but for simplicity one cannot mix these scalar types. In order to allow writing algorithms for different scalar types the macros TYPE and SUBR add prefixes for the currently selected scalar type (e.g. TYPE(mvec_ptr) becomes Dmvec_ptr_t for double precision). Furthermore, there are macros respectively predefined classes in C++ for the allocation and calculation with the currently selected scalar type; here we distinguish between the scalar type (_ST_) and the magnitude type (_MT_), since the norm of a complex vector is a real number. One obtains for example:

```
typedef double complex _ST_;
typedef double _MT_;
```

The following basic linear algebra data types are provided by the *phist* interface:

crsMat specifies a large distributed sparse matrix. It provides a routine to calculate a sparse matrix-multiple-vector multiplication in parallel (crsMat_times_mvec). Despite the name the kernel library employed is free to choose an appropriate distributed data format for the sparse matrix. Therefore we also decided that the construction of preconditioners that need to access individual entries of the matrix (like incomplete factorizations) should be handled by the kernel library itself.

mvec specifies a block of vectors where the individual columns are distributed among the nodes. It offers mainly three different kinds of routines:
AXPY-like routines that add or scale multiple vectors at once possibly with different scaling factors (mvec_vadd_mvec). The scaling factors must be provided on all nodes; so these computations do not involve any communication.
DOT-like routines for multiple vectors that calculate the individual dot products $v_i^* w_i$, $i = 1, \ldots, n_b$ of two blocks of vectors $V = (v_1, \ldots, v_{n_b})$ and $W = (w_1, \ldots, w_{n_b})$. The result is returned on all nodes, which requires obviously an all-reduction.
Two kinds of GEMM-like routines where the first one can be seen as an extension of an AXPY operation to complete blocks of vectors: It computes locally dense matrix-matrix multiplications of the form $W \leftarrow Vm$ with a small matrix $m \in \mathbb{C}^{n_b \times n_b}$ (mvec_times_sdMat). And the second one can be seen as an extension of DOT operations to complete blocks: It provides essentially operations of the form $m \leftarrow V^* W$ (mvecT_times_mvec) and also requires an all-reduction.

sdMat specifies a small dense matrix that is stored redundantly on all nodes. It is needed for the m matrix from the routines mentioned above. Additionally, it provides routines for the addition and multiplication of several sdMats (e.g. sdMat_add_sdMat and sdMat_times_sdMat).

These data types and the corresponding routines provide all operations required in the block JDQR and other linear algebra algorithms such as GMRES. And due to the

multiple vector approach one may also formulate block variants of a given single vector method very easily.

5.1.2. Important concepts

There are mainly two important concepts in the *phist* framework I want to point out: *views* and *operators*.
A *view* allows to access contiguous subblocks of a sdMat or a mvec. It essentially creates a new sdMat respectively mvec that consists of the selected submatrix of the sdMat respectively the selected set of columns of the mvec. Here we do not allow the construction of scattered views (e.g. views to non-contiguous subblocks) in order to avoid performance penalties due to complex memory access patterns. I want to note that in the *phist* interface the user does not know (and should not need to know) if a given sdMat respectively mvec is actually a view of another matrix or multi-vector. So you can also create views of views and the *phist* interface together with the underlying kernel library take care that this works as expected; The user of a function must make sure though that the arguments passed to a function do not point to overlapping regions of data (e.g. aliasing arguments are not allowed).

An *operator* is an abstract function that applies some transformations to a block of vectors of type mvec. In the *phist* framework it is currently only used to represent linear operators (but not restricted to it). So one can replace a crsMat in an algorithm by an operator. This gives two advantages: First of all, the matrix does not need to be explicitly available in a linear solver or an eigensolver from the *phist* framework. And secondly, one can easily replace a matrix by a more complex linear operator such as the Jacobi-Davidson operator $(I - \tilde{Q}\tilde{Q}^*)(A - \tilde{\lambda}_i I)$ or a preconditioned matrix.

5.1.3. Self-implemented kernel routines

As shown in Section 3.2 the kernel libraries Trilinos and GHOST do not achieve a satisfying node-level performance for cases with multiple vectors. This also holds for GEMM-like vector operations (e.g. mvecT_times_mvec and mvec_times_sdMat). In both libraries multiple vectors are stored in column-major order and for both libraries we use the same data type for the *phist* types mvec and sdMat with additional flags to specify that the data in the sdMat should be replicated on all nodes. And both libraries simply call the appropriate GEMM routine from the BLAS, which may be problematic in the context of non-uniform memory accesses (NUMA) (see Section 5.4). Besides, my tests with the Intel MKL[5] and OpenBLAS[6] suggest that the BLAS libraries may not provide optimized algorithms for the relevant cases here. These are in particular cases with tall skinny matrices (e.g. block vectors) with a very small number of entries in one dimension (for example $n_b = 1$, 2, 4 or 10) and $n > 10^5$ in the other dimension.

My self-implemented kernel routines use a row-major storage scheme for the type mvec and use OpenMP for intra-node parallelization and MPI for communication be-

[5]For the Intel MKL see `http://software.intel.com/en-us/intel-mkl`.
[6]For OpenBLAS please refer to `http://www.openblas.net`.

tween nodes. Most of the kernel functions themselves are quite short; but in order to obtain adequate compiler optimizations I have implemented individual routines for different block sizes, e.g. for $n_b = 1$, $n_b = 2$, $n_b = 4$ and $n_b = 8$. Additionally, in most cases there are two variants for each block size, one for completely contiguous data and one with strided memory accesses for views.
In Algorithm 6 we see an example of such a kernel routine.

Algorithm 6 Self-implemented GEMM implementation for the local part of the computation of mvecT_times_mvec for block size $n_b = 4$ and strided memory accesses

```
1  subroutine dgemm_sC_strided_4(nrows, nvecw, v, w, ldw, M)
2    integer,         intent(in)  :: nrows, nvecw, ldw
3    real(kind=8), intent(in)  :: v(4, nrows)
4    real(kind=8), intent(in)  :: w(ldw, nrows)
5    real(kind=8), intent(out) :: M(4 ,nvecw)
6    integer :: i, j
7
8    M = 0.
9  !$omp parallel do reduction(+:M) schedule(static)
10   do i = 1, nrows
11     do j = 1, nvecw
12       M(:,j) = M(:,j) + v(:,i)*w(j,i)
13     end do
14   end do
15
16 end subroutine dgemm_sC_strided_4
```

This specific subroutine performs continuous accesses to v and allows a stride ldw > 4 for w. And the calculation for the individual rows is partitioned in equally sized chunks among the cores of the node using OpenMP with static scheduling. In order to calculate the resulting small dense matrix M an array reduction (OpenMP 3.0 feature) is performed.
I do also assume that matrices of the type sdMat are very small compared to a multivector (mvec), so for operations which involve both types my implementation always creates a contiguous copy for the data from the sdMat. Additionally, all operations involving only sdMat types are performed serially, as it obviously does not make sense to use multiple threads to add for example two 4×4-matrices.

In the wrapper routines for the *phist* interface an appropriate individual implementation is chosen based on the memory layout (e.g. stride and block size). There I also try to detect cases where one could employ non-temporal stores: the previous data in resulting mvec must not be needed in the calculation and the resulting data must be aligned on 16 byte boundaries (single contiguous vector or stride and block size multiples of 2 for double precision). These restrictions are mainly due to my implementation as the SSE-streaming statements employed always work on aligned pairs of double-

precision numbers (see [10, volume 1, chapter 11] for more information). There are also more complex SSE-statements for non-temporal stores that could handle individual double-precision numbers. Furthermore, the Intel compiler offers nice vectorization pragmas but these are not portable.)
The following example in Algorithm 7 shows the implementation for $n_b = 2$ and continuous data accesses. The header file `emmintrin.h` provides data types and functions to

Algorithm 7 Self-implemented vector version of an AXPY operation for the computation of mvec_add_mvec for block size $n_b = 2$ and continuous memory accesses with non-temporal stores

```
1  #include <emmintrin.h>
2  void daxpy_nt_2_c(int nrows, const double *restrict alpha,
3                    const double *restrict x, double *restrict y)
4  {
5    __m128d alpha_ = _mm_set_pd(alpha[1],alpha[0]);
6  #pragma omp parallel for schedule(static)
7    for(int i = 0; i < nrows; i++)
8    {
9      // get x
10     __m128d x_ = _mm_load_pd(x+2*i);
11     // multiply with alpha
12     __m128d y_ = _mm_mul_pd(x_,alpha_);
13     // non-temporal store
14     _mm_stream_pd(y+2*i, y_);
15   }
16 }
```

use SSE-instructions explicitly: the type __m128d stores two double-precision values at once and the function _mm_stream_pd actually performs the non-temporal store (e.g. writes y_ to y[2*i+1] and y[2*i] without polluting the cache).

For all these kernel routines I have checked that they achieve nearly the same main memory bandwidth as the STREAM benchmark using LIKWID[24]. The implementation of the matrix-multiple-vector multiplication with non-temporal stores looks similar to the vector addition above, but the implementation is a bit more complex than Algorithm 1 because of the additional buffer required for non-local vector elements.
Unfortunately, the TSQR implementation in Trilinos can only be used with a column-major storage scheme for blocks of vectors. Therefore I had to implement a simple block IMGS scheme for the orthogonalization that cannot compete with the TSQR algorithm.
This is not discussed further in Section 7, but the effect on the runtime is small in relation to the total computation time for the tests cases considered.

5.2. Linear solver: pipelined GMRES

The idea of the pipelined GMRES solver is to allow the concurrent solution of a fixed number of independent linear systems using a standard GMRES method and to group together similar operations, such that one can work with blocks of vectors instead of single vectors. Additionally, it should be easy to replace a system that has converged (to a given tolerance) by a new linear system. And when there are no new linear systems to solve, it must still be possible to continue the iteration of the remaining unconverged systems.
In the following I describe first the interface available to the user of the PGMRES algorithm in *phist* and then shortly discuss the internal data structures required to perform most operations with blocks of vectors.

5.2.1. Public interface

In order to handle different linear systems in the PGMRES solver individually, we assign a state object to each system. Each state has an id (unique index in the array of states created) and its own status. Additionally, one can adjust the residual tolerance and obtain the total number of iterations performed with a state object. The status of a state allows the user to check whether a system reached its residual tolerance (e.g. converged) or requires a restart.
The following code shows the definition of the state type:

```
struct TYPE(pgmresState)
{
  // public part:
  const int id;       // index of this state in block of states
  _MT_ tol;           // desired residual tolerance
  int status;         // {converged, iterating, needs restart}
  int totalIter;      // total number of iterations performed

  // private part:
  // ... data used internally ...
};
```

Internally, each state also stores all data needed by the GMRES method: the right-hand side vector b, the orthogonal basis of the current Krylov subspace (V_k) with the corresponding matrix ($H_{k+1,k}$) from the Arnoldi process ($AV_k = V_{k+1}H_{k+1,k}$) and orthogonal transformations (parameters of Givens rotations).

There are three operations that can be performed with a state object (except for allocation and deallocation): We can reset a state, perform some GMRES iterations or update the solution vector. Resetting a state means to start with a new empty Krylov subspace in the next iteration for this state object. This can be used to assign a new system to a state or to perform a restart of the GMRES method (which tears down to starting a new GMRES iteration with an empty subspace with the current approximation as starting vector).

The reset function is defined as:

```
void SUBR(pgmresState_reset)(      TYPE(pgmresState_ptr) S,
                                   TYPE(const_mvec_ptr) b,
                                   TYPE(const_mvec_ptr) x0,
                                   int *ierr);
```

It takes a single state object S, the right-hand side vector of the linear system to be solved b, which can be set to NULL to indicate a restart, and the starting vector x0. The latter can also be set to NULL to use the vector b directly as the first residual vector in the Krylov subspace. This indicates a starting vector of zero and allows to omit an additional matrix-vector multiplication.
To start or continue the GMRES iterations of a set of state objects, one needs to call the following function:

```
void SUBR(pgmresStates_iterate)(   TYPE(const_op_ptr) Op,
                                   TYPE(pgmresState_ptr) states[],
                                   int nStates,
                                   int* ierr);
```

The first argument defines the operator that is applied in each GMRES iteration to calculate the next directions of the individual Krylov subspaces. This means that the user must provide an appropriate operator that calculates the desired matrix-vector multiplications of all systems currently iterated. The operator does not necessarily need to apply the same linear operation to all vectors. This allows us to solve several linear systems concurrently that differ not only in the right-hand side vectors but also in the system matrices. The id of the states defines which columns of the multi-vectors in the operator correspond to which system.
This is essentially required for the solution of the correction equations in the block JDQR algorithm because the individual linear systems have different shifts.
One does not need to call this function with a set of states with continuous ids; this allows the user to finish the calculation of some remaining systems that have not converged yet. A drawback of this approach is that one may require computations with non-contiguous blocks of vectors. For example if one continues an iteration with a previous block size of 4 and two states with ids 1 and 3, the operator is applied to a multi-vector with stride 4 (for row-wise storage), but one only needs the results of the 1st and 3rd column.
The iterate-function above returns when at least one system converged or needs to be restarted.

In order to obtain the approximate solution of one or several linear systems one needs to call the following function:

```
void SUBR(pgmresStates_updateSol)(TYPE(pgmresState_ptr) states[],
                                   int nStates,
                                   TYPE(mvec_ptr) x,
                                   _MT_ *resNorm,
                                   int* ierr);
```

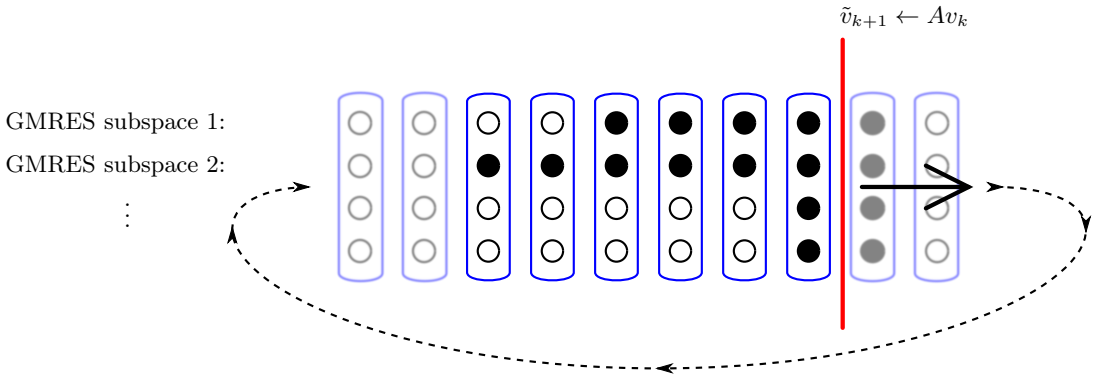


Figure 7: Visualization of the PGMRES ringbuffer for the Krylov subspaces of 4 different linear systems.

For the given state objects it adds the current GMRES correction to x. This means that x must be set to the starting value x0 from the last call to pgmresState_reset for this specific state object. This also allows adding up the individual solution vectors over several restarts of the GMRES iteration. In addition, the current estimate of the relative residual norm is written to the parameter resNorm.

5.2.2. Internal data structures

A single GMRES iteration consists of the following steps: First, one needs to multiply the matrix with the preceeding Krylov subspace basis vector ($\tilde{v}_{k+1} \leftarrow Av_k$). Then the new vector must be orthogonalized with respect to all previous basis vectors. For the PGMRES this is implemented using a modified Gram-Schmidt method. Afterwards, one still needs to apply and calculate some small orthogonal transformations and check the new estimated residual norm (see [17, chapter 6, section 5] for a complete GMRES algorithm). This only requires local operations that are not relevant for the performance of the method (at least for huge systems).
In order to allow the computation of the matrix-vector multiplications for all iterated systems at once, the basis vectors of the subspaces of the individual systems are stored as blocks of vectors in a ring buffer. In Figure 7 we see an illustration of a partly filled buffer from 4 systems. The dimensions of the Krylov subspaces of the individual systems (rows in the figure) may differ; the single basis vectors of the subspaces are visualized as filled black circles: For the first system we have a subspace dimension of $k_1 = 4$, for the second $k_2 = 6$ and for the third and fourth $k_{3,4} = 1$. So the last two systems were just added in the previous iteration using the pgmresState_reset function. The individual basis vectors from the different subspaces are stored in blocks of 4 vectors (columns in the figure). Currently unused blocks are greyed out. In the next iteration a new full block of vectors will be required to store the result of the next matrix-vector multiplication (greyed out block directly right of the red line). This allows us to apply a spMMVM with full blocks of vectors even for subspaces of different dimension. Moreover, we can also group the vector operations required for the orthogonalization

by column. This means that in most cases we may require several operations with full blocks, and then some additional operations with single vectors or parts of blocks for the subspaces of higher dimension. The PGMRES algorithm always tries to perform the vector calculations with the biggest contiguous subblock.
We employ a ring buffer here to allow replacing converged systems by new systems with a fixed amount of preallocated memory for the subspaces. When the subspace of one system fills a complete row in the ring buffer, a restart is required.
The approximate solution of several states objects can be computed in a single call to the function pgmresStates_updateSol. In this case the algorithm also tries to perform the required calculations (vector additions) with contiguous (sub)blocks where possible.

5.3. Complete BJDQR driver

In this section I describe the setup and possible configuration options of my block JDQR eigensolver driver in the *phist* framework.
There are two variants of the BJDQR driver: the first one reads a matrix from a file, the second one obtains the matrix entries from a function, which is currently used for the spin matrices described in Section 7.1. The only advantage of the second variant consists in the fact that one does not need to write the matrix to a file and read it later, which can be quite slow for huge matrices.
Both variants set up the distributed matrix in parallel so that the maximum possible matrix size scales with the number of nodes (e.g. the matrix only needs to fit in the combined memory of all nodes).
For the performance tests with symmetric eigenvalue problems the matrices were reordered (symmetrically) using ParMETIS [11] to reduce the costs of communication. To obtain the resulting eigenvectors one simply needs to apply the inverse permutation in this case. Non-symmetric matrix reorderings, however, lead to different eigenvectors (apart from simply permutated eigenvectors) and thus effectively modify the original eigenvalue problem.
In the current implementation the user can select the number of eigenvalues desired and specify in which eigenvalues he is interested in. Possible options are the largest or smallest eigenvalues in modules respectively the eigenvalues with the largest or smallest real part. The user can also specify the desired residual tolerance of the approximate Schur vectors.
In addition, one can set the maximal number of block JDQR iterations and various block sizes and subspace dimensions. In particular, one can choose the minimal and maximal dimension of the Jacobi-Davidson subspace and the block size used in the BJDQR algorithm. As the converged Schur vectors are still stored in the subspace, its minimal dimension must be larger than the number of desired eigenvalues. For a block size of 1 one obtains a variant of the classical JDQR algorithm with my modifications concerning the deflation and the handling of multiple eigenvalues.
Additionally, one may choose the block size of the pipelined GMRES solver, e.g. the number of systems solved concurrently, and the maximal Krylov subspace dimension. In the configuration used for the test cases in this thesis the latter also specifies the

maximal number of GMRES iterations; this means that the GMRES solver is not restarted because in most cases a few GMRES iterations should be sufficient to obtain approximate solutions of the Jacobi-Davidson correction equations. Furthermore, in the GMRES solver one can either use a modified Gram-Schmidt orthogonalization or an iterated modified Gram-Schmidt orthogonalization. The iterated orthogonalization is more costly as it often requires an additional MGS iteration. It is only employed for the numerical tests in Section 6 as some of the matrices used there are badly conditioned.
And one can choose to abort the iteration of a complete block in the pipelined GMRES iteration when the first system in the block reached its residual norm. This allows to avoid operations with single vectors in the PGMRES algorithm, but also most probably produces less accurate corrections. In Section 6 I will test different combinations of the options mentioned above and discuss the results.

The BJDQR driver first performs a set of Arnoldi iterations to construct an orthogonal basis of an initial subspace of the specified minimal subspace dimension. One can also choose to use a fixed shift (near target eigenvalues) in the first iterations of the (block) JDQR method, but this is not tested in this thesis because I want to focus on the effects of different block sizes.

5.3.1. Unimplemented features

For non-symmetric real matrices one may obtain pairs of complex-conjugated eigenvalues. In real arithmetic one needs to work with 2×2 blocks on the diagonal of the (approximated) Schur form for this case. In the block JDQR algorithm one may need to adjust the block size for this case. This case is currently not fully supported in my block JDQR implementation. So the numerical tests with non-symmetric matrices in Section 6 are simply performed in complex arithmetic. And for the performance tests in Section 7 I will only consider symmetric matrices.

For parallel computations on a large number of nodes the costs of communication increase significantly. Essentially the communication required in the matrix-vector multiplication but also the all-reductions required in the algorithm are concerned. Using block operations may help to decrease latency effects, but the parallel performance of the matrix-vector multiplication may be limited by the bandwidth of the network (as discussed in Section 3.3). One way to hide the communication overhead consists in overlapping the communication with local calculations. This is possible in both kernel libraries GHOST and Trilinos but currently not fully supported in the *phist* framework yet, and hence not implemented in the BJDQR driver.

5.4. NUMA awareness?

On the Lima-cluster at the RRZE each node consists of two Intel Xeon 5650 Westmere processors. Each processor has its own memory attached, but can also access the memory of the other processor (ccNUMA). Thus, the attainable memory bandwidth depends on the correct placement of the data in NUMA domains as illustrated in Table 3 (here

	1 CPU	2 CPUs
wrong placement	12.1 Gb/s	18.5 Gb/s
correct placement	19.8 Gb/s	39.5 Gb/s

Table 3: Measured STREAM bandwidth on a single node of the Lima cluster with different NUMA placements using LIKWID

a NUMA domain corresponds to one CPU). One obtains the worst bandwidth (12.1 Gb/s) when all threads run on one CPU but the data lies in the memory attached to the other CPU. With a correct placement the attainable bandwidth on the node is the sum of the bandwidths of the individual CPUs (39.5 Gb/s $\approx 2 \cdot 19.8$ Gb/s).
As previously discussed in Section 3.2, the memory bandwidth is the limiting factor of the node-level performance of the operations required in the JDQR algorithm. Therefore the correct NUMA placement is crucial to obtain an adequate performance. If it is not handled correctly, the performance may drop significantly (up to a factor of three on the Lima cluster).

In order to obtain the best performance possible in the presence of NUMA one needs to address two points (for more information see [5, chapter 8]):
First, the threads must be pinned to the individual CPU cores (or at least to the NUMA domains) to make sure that a specific thread will always run in the same NUMA domain. For the results shown in this thesis I have used the KMP_AFFINITY environment variable which offers control over the pinning of OpenMP threads when the application is compiled with the Intel compiler. The following command line statement was employed on the Lima cluster to start the BJDQR driver in parallel using Intel MPI with a given number of nodes and 12 threads on each nodes with 1 thread per core:

```
mpirun -np <NNODES> -npernode 1 \
    env OMP_NUM_THREADS=12 \
    env KMP_AFFINITY="granularity=fine,compact,1,0" \
    <BJDQR driver and arguments>
```

There are also other methods to pin the threads of an application (e.g. the linux `taskset` utility, `likwid-pin`), but the Intel OpenMP and MPI create additional threads for management purposes. These must be distinguished from the OpenMP threads.
Second, the data must be distributed correctly among the NUMA domains. One possibility to achieve this is the *first touch* policy. It essentially states that pages of memory previously allocated are stored in the NUMA domain of the thread that first touches them (e.g. writes to them). This way one may allocate a long array shared by all threads, but the memory used by the array is actually distributed over the NUMA domains. In Algorithm 8 you can see in a simple example that shows how this is handled in my self-implemented kernel routines.

The GHOST kernel library takes care of pinning the threads itself; therefore one should not set the `KMP_AFFINITY` variable. GHOST should also place the data in the

Algorithm 8 Example of NUMA placement with OpenMP in Fortran (all threads in one NUMA domain should have consecutive ids)

```
1   ! allocate array A
2   allocate(A(n))
3
4   ! initialize A to 0 with correct NUMA placement
5  !$omp parallel do schedule(static)
6   do i = 1, n
7     A(i) = 0
8   end do
9
10  ! all further accesses should be of the form:
11 !$omp parallel do schedule(static)
12  do i = 1, n
13    ! ... A(i) ...
14  end do
```

correct NUMA domains, but there still seemed to be a problem when using the *phist* interface with GHOST that somehow interfered with the NUMA placement. So I have obtained the best performance results using two MPI processes per node, such that on each node GHOST pins all threads of the first process to the first NUMA domain and all threads of the second process to the second NUMA domain:

```
mpirun -np <2*NNODES> -npernode 2 \
    env OMP_NUM_THREADS=6 \
    <BJDQR driver and arguments>
```

This setup with one MPI process per NUMA domain also allows the correct configuration with additional threaded libraries. In particular, a threaded BLAS library (such as the Intel MKL) may probably not distribute the work, and thus the data accesses, among the threads in the way intended by the user of the library.

6. Numerical results

I have tested the block JDQR method developed in this thesis with different symmetric and non-symmetric (real) matrices from the Matrix Market [2], the University of Florida Sparse Matrix Collection [3] and matrices from the ESSEX project. The latter are used for the performance tests in Section 7. In the following I analyze the numerical behavior of the method, in particular the number of matrix-vector multiplications performed to calculate a given number of eigenvalues. Here, the focus lies on the effect of different block sizes for the matrices considered. I also compare the BJDQR variant derived in this thesis with an implementation of the classical JDQR algorithm in order to show the effects of my modifications for the single vector case. These consist mainly in the following two points: A different approach for the deflation of previously calculated eigenvalues to avoid communication and a modification of the Jacobi-Davidson correction equation for the case when multiple eigenvalues are detected.
The discussion of the numerical experiments is divided into two parts because the setup for the different kinds of matrices varies significantly.

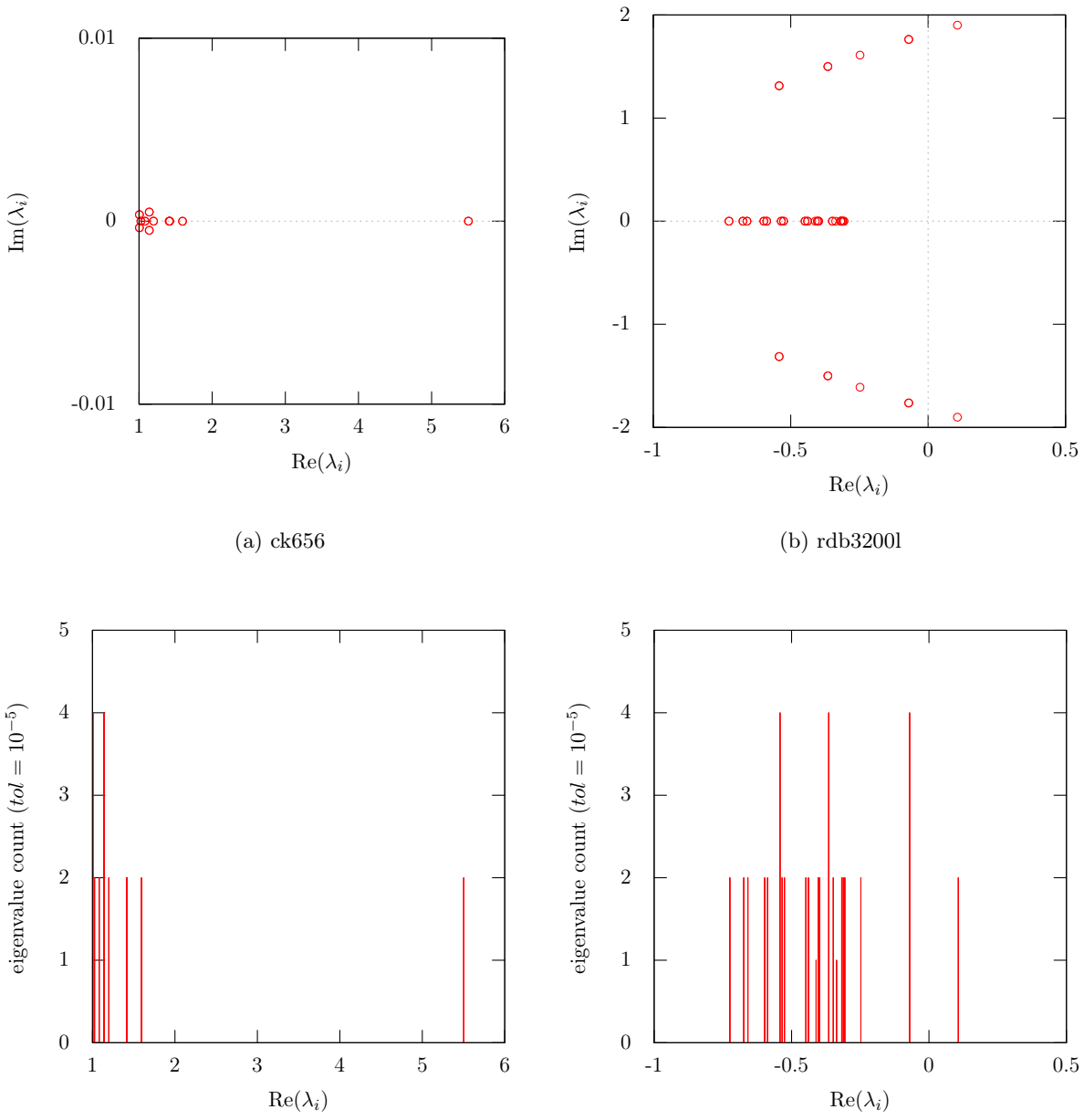
6.1. Non-symmetric matrices

The 5 matrices used for the experiments with non-symmetric eigenproblems are listed in Table 4. For these matrices one is interested in a set of eigenvalues with largest real value. For the ck656 matrix only the eigenvalues with a real value bigger than one are relevant (about 22). I have picked these matrices out of the matrices from the Matrix Market [2] because they depict difficult non-symmetric eigenvalue problems from different fields of applications where one is interested in a set of exterior eigenvalues.

name	number of rows	non-zero count	eigenvalues sought	remarks
ck656	656	3884	rightmost	real, clustered
cry10000	$1 \cdot 10^4$	$\sim 5 \cdot 10^4$	rightmost	real, $\kappa \approx 7 \cdot 10^{20}$
dw8192	$8,192$	$\sim 4.2 \cdot 10^4$	rightmost	real
olm5000	$5 \cdot 10^3$	$\sim 2.0 \cdot 10^4$	rightmost	real
rdb3200l	$3,200$	$\sim 1.9 \cdot 10^4$	rightmost	real

Table 4: Overview of the non-symmetric matrices used in the experiments. They all come from the Matrix Market [2]

To discuss the different behavior of the methods for multiple and tightly clustered eigenvalues we first look at the eigenvalue distribution in the desired part of the spectrum. Figure 8 and Figure 9 show the calculated eigenvalues obtained with the classical JDQR implementation. In particular, one can also see which eigenvalues are tightly clustered (on the real axis), respectively (nearly) multiple. However, in some cases the eigenvalues are not detected in the correct order by the algorithm; that means that first a few smaller eigenvalues are found before parts of eigenvectors of larger eigenvalues enter the subspace. So, obviously, the algorithm may have missed some of the rightmost eigenvalues.



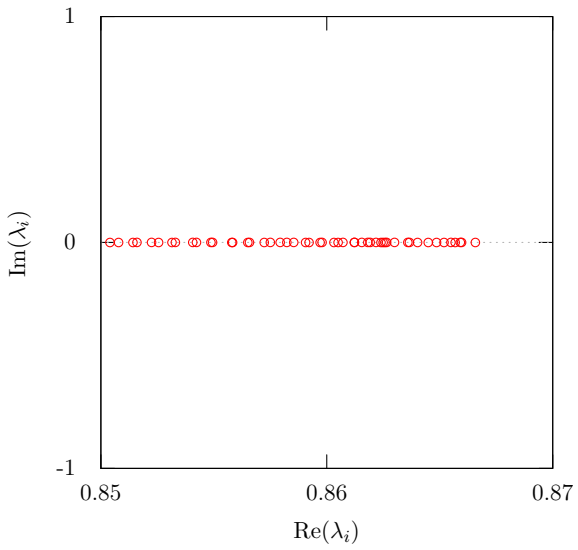
(a) ck656

(b) rdb3200l

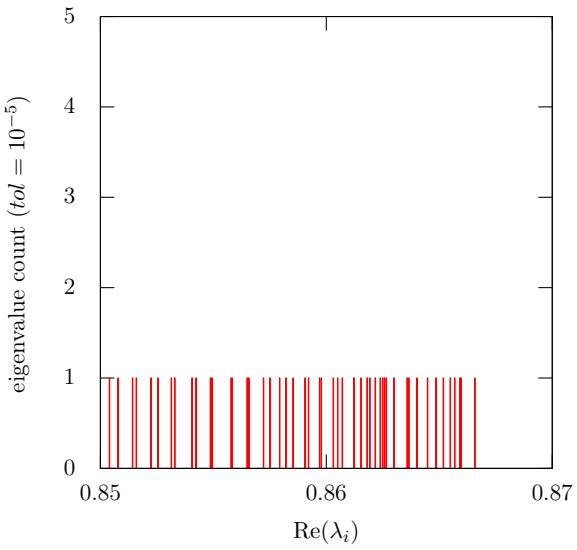
(c) ck656, clustering

(d) rdb3200l, clustering

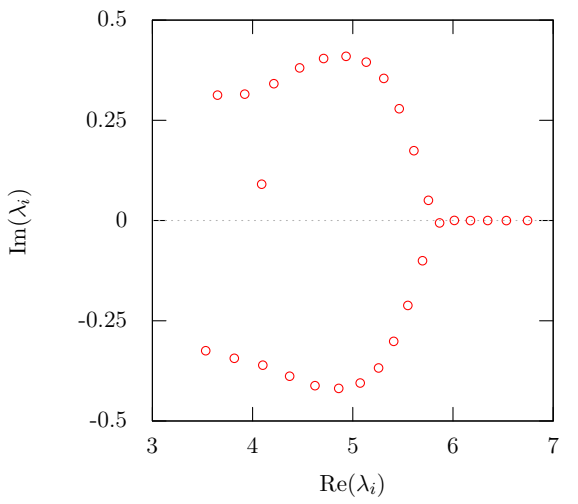
Figure 8: Calculated distribution of the rightmost eigenvalues for the matrices ck656 and rdb3200l. The images on the top show the largest real eigenvalues in the complex plane. The images on the bottom show the corresponding number of eigenvalues within a specific tolerance of their real value in order to visualize (nearly) multiple eigenvalues and pairs of complex conjugate eigenvalues.



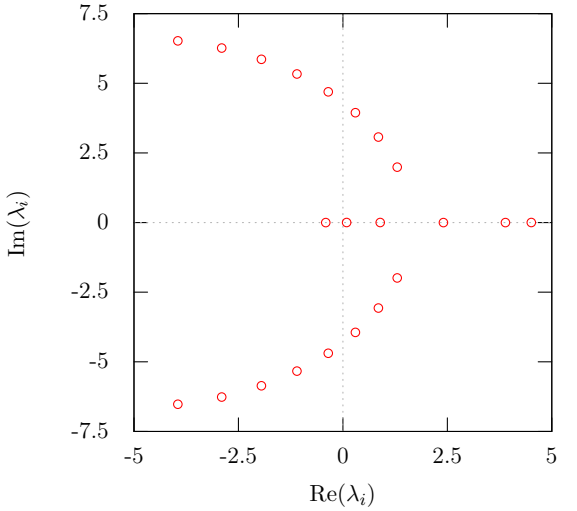
(a) dw8192



(b) dw8192, clustering



(c) cry10000



(d) olm5000

Figure 9: Calculated distribution of the rightmost eigenvalues for the matrices dw8192, cry10000 and olm5000. For dw8192 also the clustering is visualized. No (nearly) multiple eigenvalues where found for these matrices.

The Ritz residual of all approximated eigenvalues is smaller than 10^{-7}. However, as already mentioned previously, this does not ensure that slight deviations may not disturb the results significantly. In the following the terms eigenvalue and spectrum always refer to calculated eigenvalues. Additionally, the multiplicity relates to the number of calculated eigenvalues within a small tolerance.

Looking at Figure 8 we state that the desired part of the spectra of the matrices ck656 and rdb3200l are tightly clustered and feature several multiple eigenvalues with a maximal multiplicity of 4. The desired eigenvalues of the matrix ck656 lie all near the real axis whereas the matrix rdb3200l has several complex-conjugate eigenpairs.
As seen from Figure 9 the desired eigenvalues of the matrix dw8192 are all real and well separated but still clustered loosely. In the rightmost part of the spectrum of the matrices cry10000 and olm5000 there are many complex-conjugate eigenpairs and we observe no tight clusters.

6.1.1. Setup of the experiments

For the matrices described above I have performed tests with different configurations: The number of desired rightmost eigenvalues n_{ev} varies between 1 and 50 and the tested block sizes are 1, 2, 4 and 10 for the block JDQR algorithm (in the following called BJDQR1, BJDQR2, ...). The minimal and maximal Jacobi-Davidson subspace dimensions depend on the number of eigenvalues sought: $n_{ev} + 9 < k < n_{ev+} + 29$. The iteration starts with a subspace of minimal dimension obtained by an Arnoldi iteration with a starting vector of $(1, 1, \ldots, 1)^T$. If the Arnoldi algorithm breaks down (because it found an invariant subspace), the construction of the initial subspace continues with a random vector.
The inner solver used for the approximate solution of the correction equation is an unpreconditioned GMRES iteration. For each correction vector the inner iteration is stopped either when a maximal number of iterations or a sufficient reduction in the relative residual is reached. The residual tolerance is simply set to $0.5^{n_{ev\_iter}}$ where n_{ev_iter} corresponds to the number of iterations already performed for a specific approximate eigenvalue. The maximal number of iterations was set to 25 respectively 8 (GMRES25 and GMRES8). Additionally, for block sizes of $n_b > 1$ we can also stop the iteration when any of the correction vectors in the block reaches a sufficient residual reduction (denoted by GMRES25* and GMRES8*). In the GMRES method an iterated modified Gram-Schmidt orthogonalization (IMGS) was used because with MGS the orthogonality of the basis vectors strongly deteriorates in some cases even after only a few GMRES iterations. This may be caused by the fact that no preconditioner was employed while the conditioning of some of the matrices used here is bad.
For comparison a similar setup with at most 25 inner GMRES iterations was used for the classical JDQR algorithm (in the following simply denoted by JDQR). In all cases the number of matrix-vector multiplications required for the calculation of the given number of eigenvalues is determined irrespective of whether these eigenvalues are actually the rightmost eigenvalues of the matrix.

6.1.2. Comparison with classical JDQR

In order to see how my modifications to the classical JDQR algorithm influence the convergence behavior of the method we can compare the JDQR algorithm with BJDQR1. Figure 10 and Figure 11 show the number of matrix-vector multiplications performed for the calculation of a given number of eigenvalues using JDQR and BJDQR1 with at most 25 GMRES steps per outer iteration. Additionally, also the results of BJDQR with at most 8 GMRES steps are shown.
For the matrices cry10000 and dw8192 we observe similar behavior for a low number of eigenvalues; but when we seek more eigenvalues, the classical JDQR algorithm needs less matrix-vector multiplications (see red and blue lines in Figure 10c and Figure 10d). This may be contributed to the fact that the effectively usable dimension of the Jacobi-Davidson subspace is smaller for BJDQR1 because all previously calculated Schur vectors are locked in the subspace whereas they are stored separately in the JDQR algorithm. However, the BJDQR1 method requires less memory and also needs fewer operations for the deflation of already calculated eigenvalues. So in order to decide which variant is preferable here one needs to look at the obtained runtime performance which is discussed in Section 7.
For the matrix olm5000 we observe similar trends for 1, 5 and 10 eigenvalues, but for 20 eigenvalues the JDQR algorithm needs more iterations than BJDQR1 (see Figure 10b). As there are no (nearly) multiple or tightly clustered eigenvalues in the desired part of the spectrum we cannot explain this behavior by the different handling of multiple eigenvalues. However after eigenvalue 10 was found, the algorithm seems to detect spurious eigenvalues with big real value in some iterations and then switches back to a better eigenvalue approximation with smaller real part. In this case the BJDQR1 algorithm resets the required residual norm used as stopping criterion in the inner GMRES iteration in contrast to the JDQR implementation used. Hence, the JDQR implementation possibly performs many useless GMRES iterations for correction equations from these bad eigenvalue approximations. So this only suggests that there is problem in our implementation of the JDQR algorithm and not in the JDQR method itself. This effect did not occur in any of the other test cases.
The results of the calculations with the matrix ck656 in Figure 10a show that the detection of multiple or clustered eigenvalues in BJDQR1 works well and that one can reduce the number of required matrix-vector multiplications in the JDQR method significantly by the approach for multiple eigenvalues presented in Section 4.3.1.

At first sight the behavior observed for the matrix rdb3200l in Figure 11a looks strange as the algorithms seem to converge much faster for the tests with 50 eigenvalues than for the tests with fewer eigenvalues. However, we can explain this by the fact that the calculations use different subspace dimensions depending on the number of desired eigenvalues. For all three configurations the convergence rate (e.g. the required number of matrix-vector multiplications per eigenvalue) is much slower for 20 eigenvalues than for 10 eigenvalues. Through the usage of a bigger subspace for 50 eigenvalues the convergence rate increases significantly. So in this case the subspace dimension has a great impact on the required number of matrix-vector multiplications.

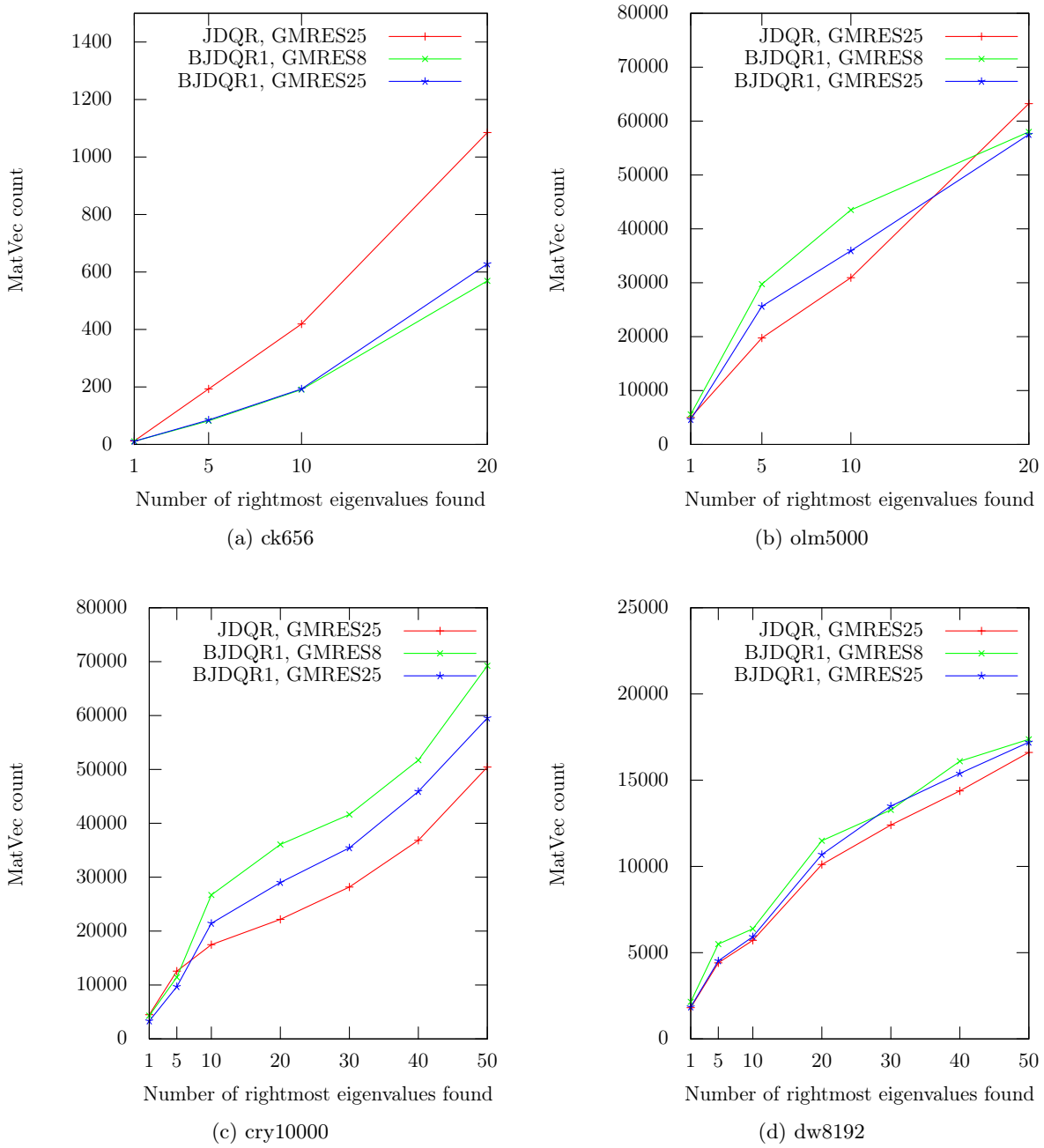
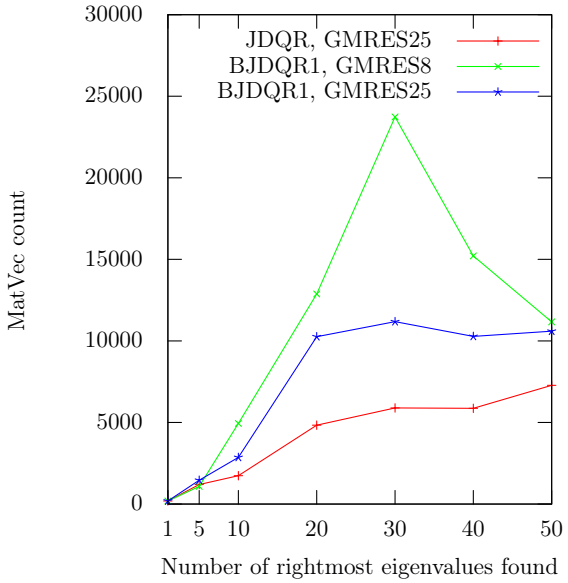


Figure 10: Total number of matrix-vector multiplications required by the JDQR and the BJDQR1 method for the calculation of the rightmost eigenvalues of the matrices ck656, olm5000, cry10000 and dw8192 with a residual norm of 10^{-7}..



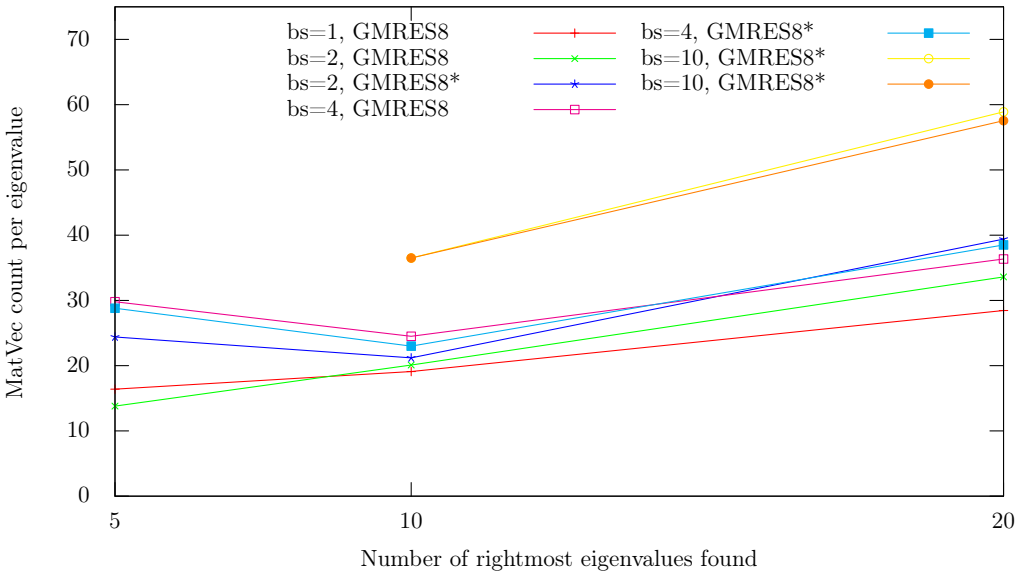
(a) rdb3200l

Figure 11: Total number of matrix-vector multiplications required by the JDQR and the BJDQR1 method for the calculation of the rightmost eigenvalues of the matrix rdb3200l with a residual norm of 10^{-7}..

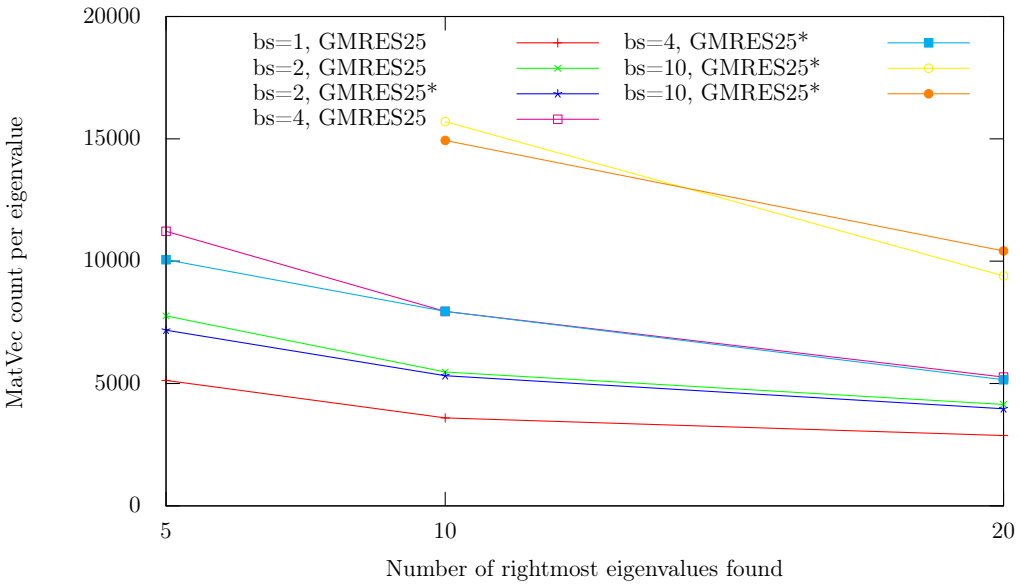
This also explains why the JDQR algorithm achieves much better results than the BJDQR1 algorithm here: As mentioned above, the usable subspace dimension of the BJDQR method decreases through locking of converged Schur vectors.

6.1.3. Influence of the block size

If we increase the block size of the BJDQR method, the individual operations can achieve a much better performance but we may also need more operations. In order to understand in which cases choosing a bigger block size lowers the overall computation time, I want to analyze the influence of the block size on the convergence behavior of the BJDQR algorithm. As before I use the required number of matrix-vector operations to compare the different algorithms. Here a block operation is counted as several single operations, e.g. a spMMVM with two vectors is considered as two matrix-vector multiplications keeping in mind that the block operations are faster. So a small number of additional operations is acceptable for a bigger block size.
In Figure 12, Figure 13 and Figure 14 we see the number of matrix-vector multiplications divided by the desired number of eigenvalues (MatVec ratio) for the test cases considered. Only the results for maximal 8 respectively for maximal 25 inner iterations are shown depending on which configuration was faster for the specific matrices. In almost all most cases the MatVec ratio increases when the block size is increased. However, the effect is much smaller if more than 20 eigenvalues are sought. Additionally, for

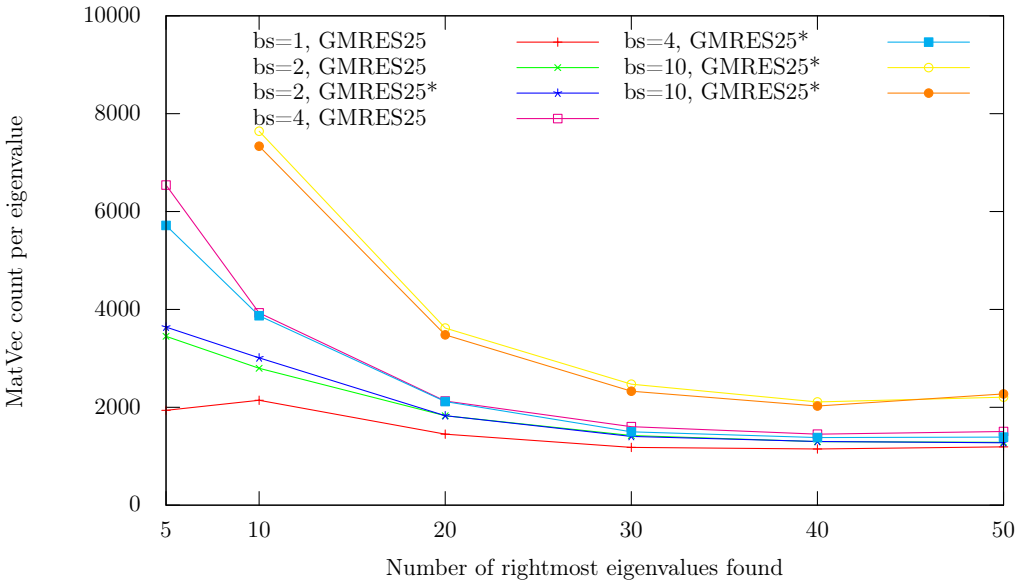


(a) ck656

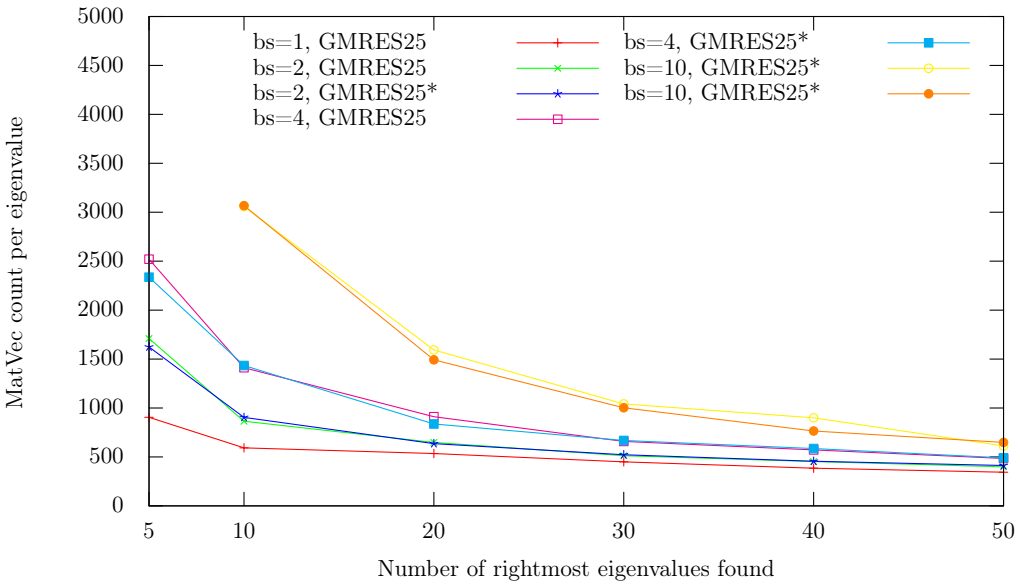


(b) olm5000

Figure 12: Ratio of Matrix-vector multiplications to the number of desired eigenvalues obtained with the BJDQR method with different block sizes. Here the results are shown for the calculation of the rightmost eigenvalues of the matrices ck656 and olm5000 with a residual norm of 10^{-7}.

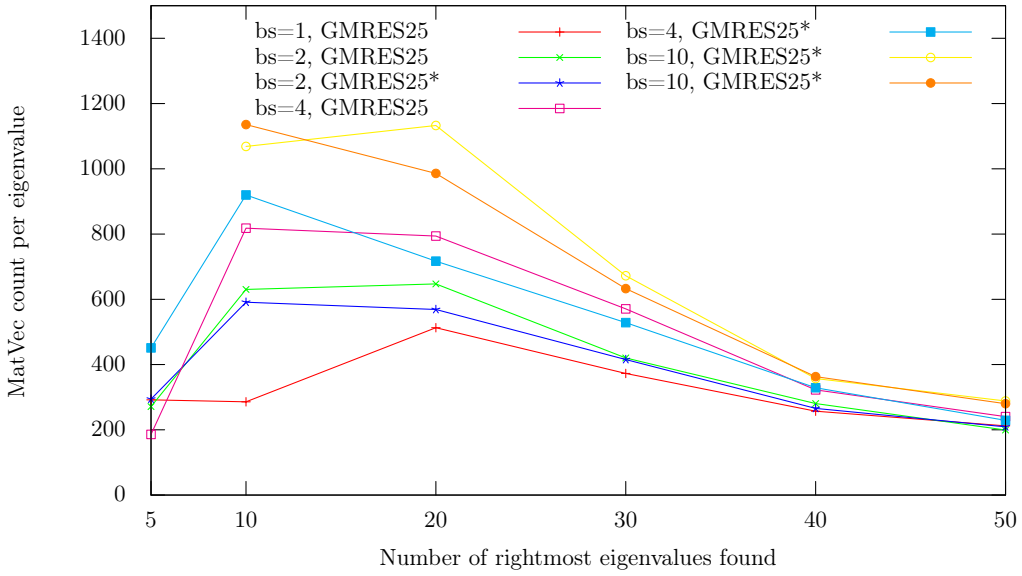


(a) cry10000



(b) dw8192

Figure 13: Ratio of Matrix-vector multiplications to the number of desired eigenvalues obtained with the BJDQR method with different block sizes. Here the results are shown for the calculation of the rightmost eigenvalues of the matrices cry10000 and dw8192 with a residual norm of 10^{-7}.



(a) rdb3200l

Figure 14: Ratio of Matrix-vector multiplications to the number of desired eigenvalues obtained with the BJDQR method with different block sizes. Here the results are shown for the calculation of the rightmost eigenvalues of the matrix rdb3200l with a residual norm of 10^{-7}.

more than 10 eigenvalues the overall number of matrix-vector multiplications changes only slightly when the inner GMRES iteration is aborted after the first vector in a block reached its residual tolerance (GMRES8* and GMRES25*).
The results for the matrices cry10000 and dw8192 shown in Figure 13 look very similar: For the calculation of 5 eigenvalues one needs almost twice as many operations when the block size is increased from 1 to 2. The observed speedup of the Jacobi-Davidson operator in Section 3.2 lies in the range of 1.6 for a block size of 2 and 2.5 for a block size of 4, but there are also other operations that do not benefit as much from the block size (orthogonalizations of the individual Krylov subspaces in the GMRES iterations). So for the calculation of 5 eigenvalues a block method may not be beneficial yet. However, already for 10 eigenvalues the number of additional operations reduces significantly (in a relative sense), such that block methods could become faster. For more eigenvalues (20-50) block sizes of 2 and 4 only lead to a small increase in the number of operations. Furthermore, also with a block size of 10 one needs less than twice the number of operations.
The behavior observed for the matrix rdb3200l in Figure 14 is similar for larger numbers of desired eigenvalues (40 and 50). For 20 and 30 eigenvalues the MatVec ratios spread more widely, but the results obtained with block sizes 2 and 4 still look promising. Surprisingly, for 5 eigenvalues one needs slightly fewer operations with a block size of

2 and significantly fewer operations with a block size of 4 (using GMRES25 and not GMRES25*). This may be due to the repeated complex-conjugate eigenvalues in the right part of the spectrum of this matrix (see Figure 10).
For the matrix olm5000 we observe a significant number of additional operations depending on the block size (see Figure 12b). So here blocking may not be as beneficial as in the other cases.
Similar to the tests with matrix rdb3200l a block size of 2 (with GMRES8) leads to a smaller number of iterations for the matrix ck656 when only 5 eigenvalues are sought (see Figure 12a). For 10 desired eigenvalues the MatVec ratio for the block sizes 1, 2 and 4 is very similar. When more eigenvalues are sought the number of additionally required operations increases slightly for all block sizes. Here a block size up to 4 seems promising whereas a block size of 10 leads to a significant overhead.

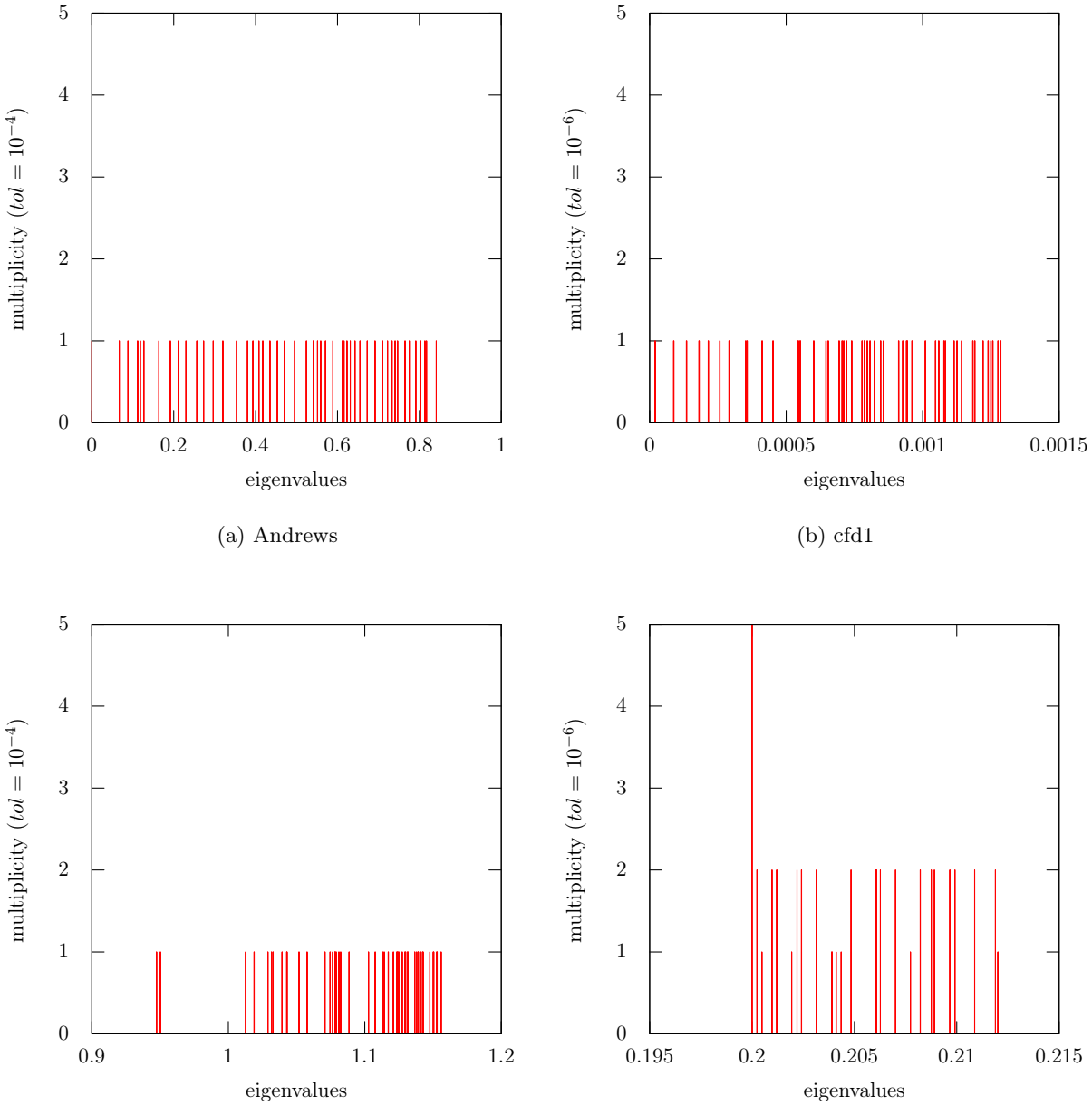
6.2. Symmetric matrices

The second set of test matrices is presented in Table 5 and consists of four real and symmetric positive definite matrices from the University of Florida Sparse Matrix Collection [3]. In contrast to the previous experiments, one is interested in the smallest eigenvalues here. In [21] these matrices are also used amongst others for numerical experiments with several different Davidson and Jacobi-Davidson methods. I want to note that we cannot easily compare the BJDQR algorithm with the elaborated methods presented in [21] (GD+k, JDQMR) because the latter are specifically designed to calculate a larger set of eigenvalues of a real symmetric matrix with a subspace dimension independent of the number of eigenvalues sought, whereas the BJDQR algorithm is based on the assumptions that one only wants to calculate a small set of outer eigenvalues of a distributed huge and possibly non-symmetric matrix. Therefore for the test problems at hand the GD+k and JDQMR methods are most probably much better suited and require much less vector-vector operations for the same number of matrix-vector multiplications performed.

name	number of rows	non-zero count	eigenvalues sought	remarks
Andrews	$6.0 \cdot 10^4$	760,154	smallest	real spd
cfd1	$\sim 7.1 \cdot 10^4$	$\sim 1.8 \cdot 10^6$	smallest	real spd
finan512	$\sim 7.5 \cdot 10^4$	$\sim 6.0 \cdot 10^5$	smallest	real spd
torsion1	$1.0 \cdot 10^4$	$\sim 2.0 \cdot 10^5$	smallest	real spd

Table 5: Overview of the symmetric matrices used in the experiments. They all come from the University of Florida Sparse Matrix Collection[3]

Figure 15 illustrates the distribution and (nearly) multiplicity of the calculated smallest eigenvalues of the matrices obtained with the JDQR algorithm. Only the matrix torsion1 features multiple eigenvalues, mostly of multiplicity 2 with one exception of multiplicity 5. The considered part of the spectrum of the matrix finan512 is clustered loosely. The matrix cfd1 has only several close eigenvalues, and the smallest eigenvalues of the matrix Andrews are more evenly distributed.



(a) Andrews

(b) cfd1

(c) finan512

(d) torsion1

Figure 15: Calculated distribution of the smallest 50 eigenvalues for the spd matrices Andrews, cfd1, finan512 and torsion1. As all eigenvalues are real only their distribution on the real axis and their multiplicity (within a tolerance) is shown.

The setup of the experiments is similar to the one previously used for the non-symmetric matrices; the only difference lies in the fact that here all eigenvalues are positive and real and that we are interested in the smallest eigenvalues, that is the leftmost part of the spectra.

6.2.1. Comparison with classical JDQR

In Figure 16 we can see the number of required matrix-vector multiplications of the algorithms JDQR and BJDQR1 for the different symmetric matrices. We observe essentially the same effects as previously discussed for the non-symmetric matrices. For a small number of eigenvalues the results of JDQR and BJDQR1 with at most 25 GMRES iterations are very similar for the matrices Andrews, cfd1 and finan512. The detection of multiple eigenvalues helps to reduce the number of matrix-vector multiplications for the matrix torsion1 (see Figure 16d). Interestingly, there is almost no difference in the results for 5 eigenvalues but only for 10 eigenvalues. This indicates that the high multiplicity of the smallest eigenvalue of the matrix torsion1 is only detected when more than 5 eigenvalues are sought so that we observe convergence to bigger eigenvalues first before more smaller eigenvalues are discovered.
For all four matrices we can see that the JDQR method requires less matrix-vector multiplications in cases with more than 20 desired eigenvalues. As discussed previously, we can contribute this fact to the smaller usable subspace dimension of the BJDQR method after a set of Schur vectors has been locked.
Additionally, we can note that for the matrices Andrews, finan512 and torsion1 it is favorable to lower the maximal number of GMRES iterations as the results obtained with GMRES8 are slightly faster.

In [21, Figure 5.6] the number of matrix-vector multiplications is shown that is required for the calculation of a set of the smallest eigenvalues of the matrix cfd1 with a tolerance of 10^{-7} using amongst others the GD+k and the JDQMR methods presented there. Thus, the setup of the experiments is similar except for the starting vector. As already discussed the (B)JDQR methods analyzed here probably require many more vector-vector operations per matrix-vector multiplication. Nevertheless, we can compare the required number of matrix-vector multiplications in order to underline that the results shown in this thesis are reasonable. In fact, the GD+k and the JDQMR methods require approximately 2500 matrix-vector multiplication for the calculation of 10 eigenvalues and about $2 \cdot 10^4$ matrix-vector multiplications for 50 eigenvalues. In Figure 16b we see nearly 5000 matrix-vector multiplications for 10 eigenvalues and about $1.8 \cdot 10^4$ for 50. This means that the required number of operations observed here for the matrix cfd1 is of the same order of magnitude as in [21].

6.2.2. Influence of the block size

Concerning the usage of different block sizes in the BJDQR algorithm the results obtained with the symmetric matrices shown in Figure 17 and Figure 18 affirm the trends observed previously for the non-symmetric test cases.

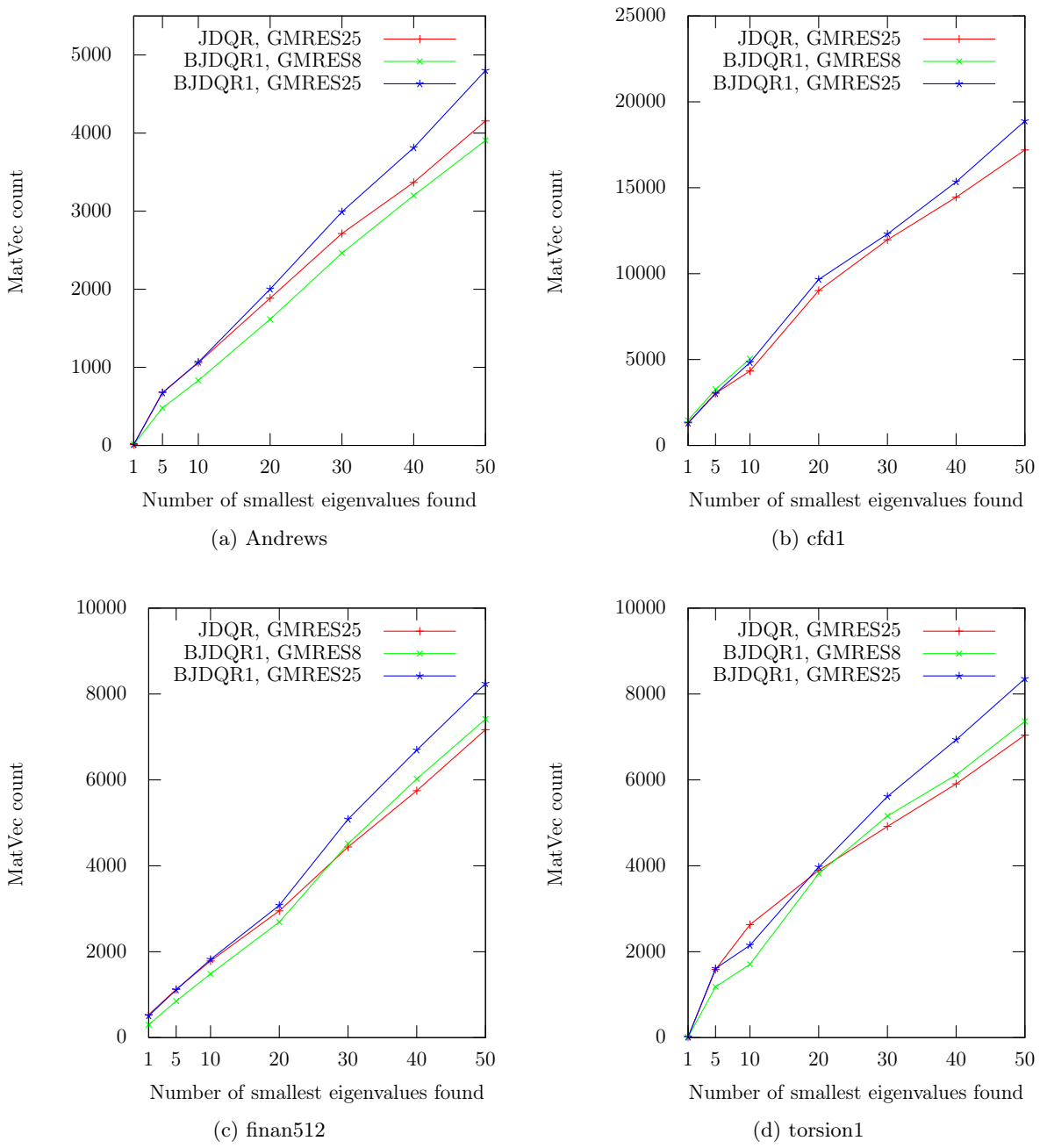
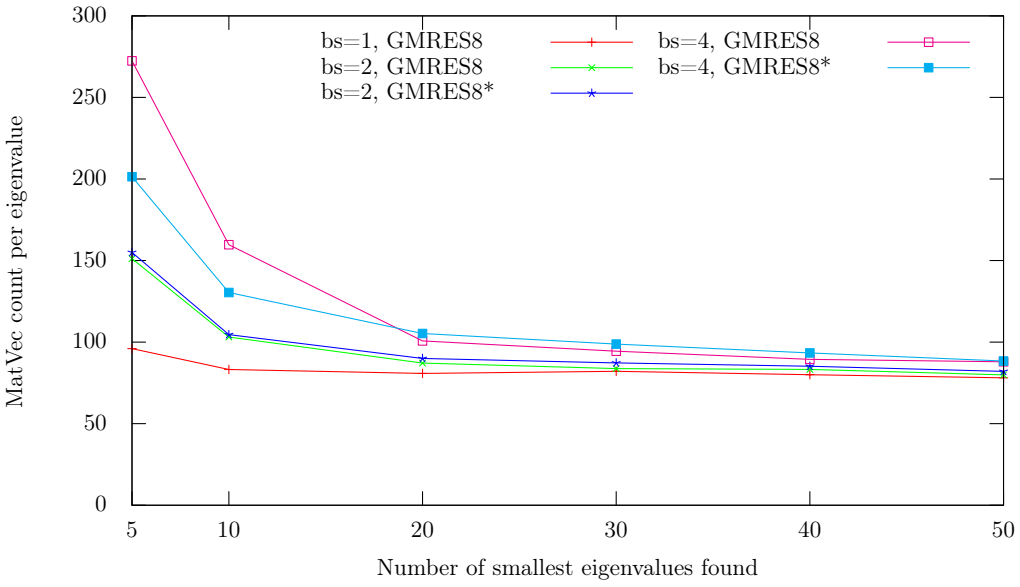
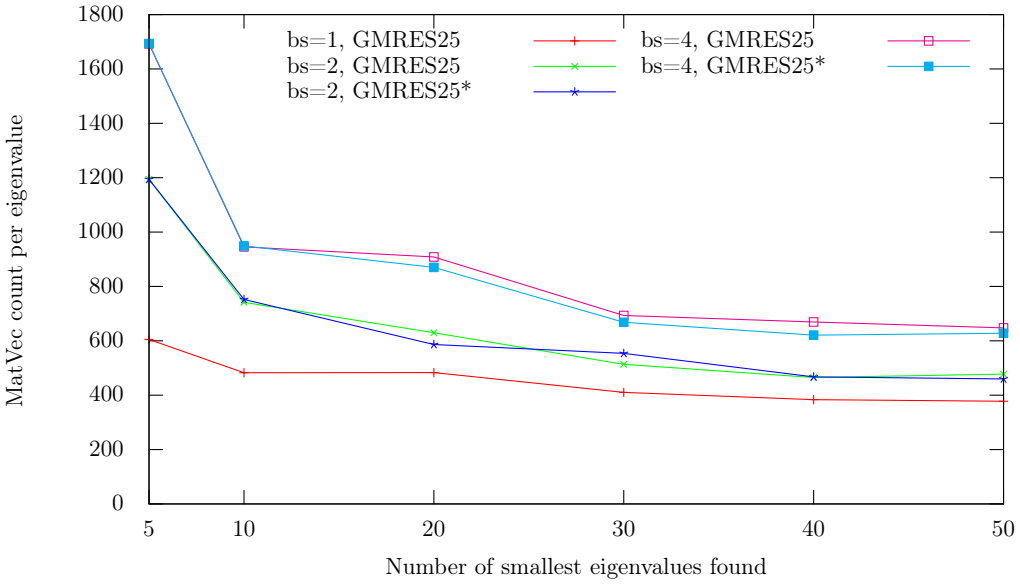


Figure 16: Total number of matrix-vector multiplications required by the JDQR and the BJDQR1 method for the calculation of the smallest eigenvalues of the matrices Andrews, cfd1, finan512 and torsion1 with a residual norm of 10^{-7}..

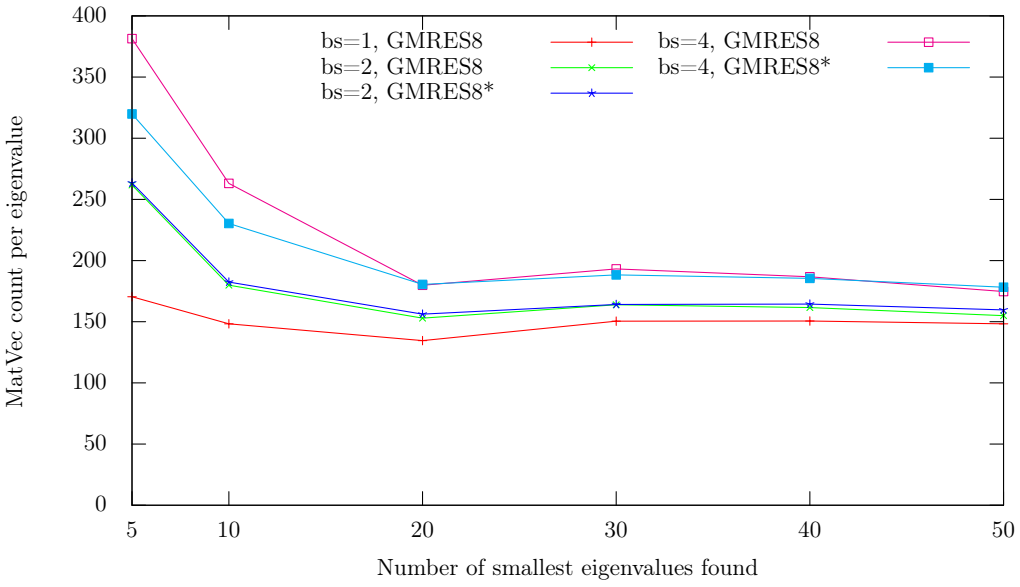


(a) Andrews

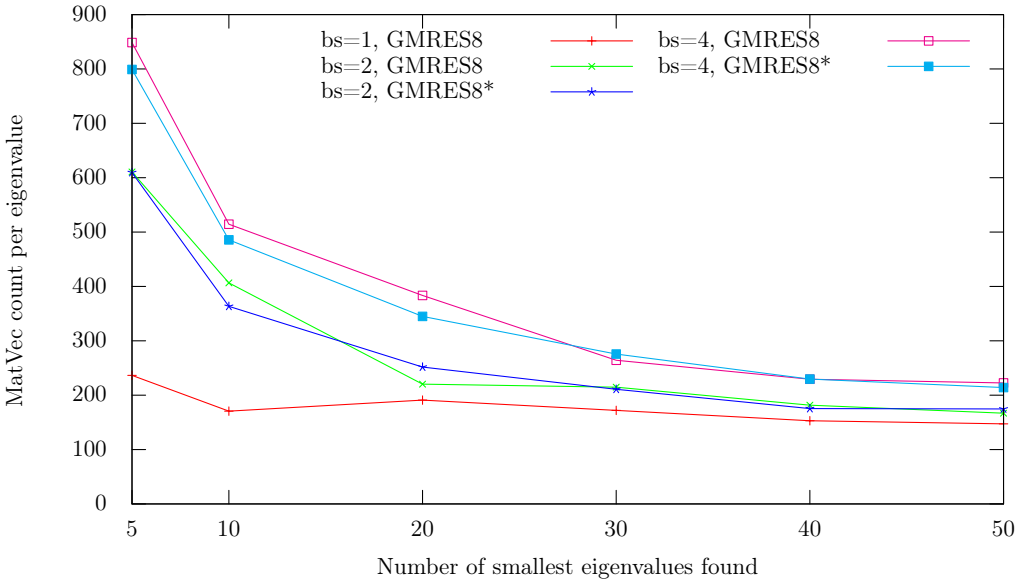


(b) cfd1

Figure 17: Total number of matrix–single-vector multiplications for the calculation of the smallest eigenvalues of the matrices Andrews and cfd1 with a residual norm of 10^{-7} and different block sizes.



(a) finan512



(b) torsion1

Figure 18: Total number of matrix–single-vector multiplications for the calculation of the smallest eigenvalues of the matrices finan512 and torsion1 with a residual norm of 10^{-7} and different block sizes.

For all four matrices one needs considerably more matrix-vector multiplications for a block size of 2 and 4 when only few eigenvalues are sought. For 20 and more eigenvalues the number of additional operations for the block variants may be amortized by the better performance of block operations. However, the relative overhead of the block methods depends on the matrix:
We obtain the best results for the matrix Andrews (see Figure 17a) where for 5 eigenvalues the number of additional matrix-vector multiplications only increases by a factor of 1.5 for a block size of 2. For larger numbers of eigenvalues there is no significant overhead in this case. Even for a block size of 4 the required number of operations increases only slightly.
The results of the matrix finan512 shown in Figure 18a look very similar but feature a slightly higher overhead. For the matrix torsion1 the overhead for smaller numbers of eigenvalues is larger than for the other matrices even though there are a lot of multiple eigenvalues in the desired part of the spectrum.
The largest overhead of block methods (in a relative sense) even for up to 50 eigenvalues occurs in the experiments with the matrix cfd1, but it is still in a range where the blocking could be beneficial.
Furthermore, we can see that aborting the inner iteration after the first vector in the block converged has at most a very small negative effect and in some cases also leads to fewer operations.

6.3. Conclusions from the numerical experiments

From the numerical experiments discussed we can draw several interesting conclusions:
First of all, the block JDQR method works fine for symmetric matrices as well as for general, non-symmetric matrices. Additionally, there is no indication that the block variants lead to significantly slower convergence for non-symmetric matrices compared to the symmetric case. This underlines that the assumptions made in the derivation of the block method in Section 4.1 are reasonable. In particular, one obtains satisfying results even though we ignore the coupling terms that occur in the block-correction equation in the non-symmetric case in Equation 23.
Many of the test cases represent examples of complex eigenvalue problems that require a huge number of matrix-vector multiplications. We could probably speed up the convergence significantly by applying an appropriate preconditioner in the inner GMRES iteration but this is out of the scope of this thesis.
For the case of multiple or tightly clustered eigenvalues the number of matrix-vector multiplications may be reduced significantly by using additional approximate Schur vectors for the projection in the Jacobi-Davidson correction equation. This requires only little computational overhead when several eigenvalues are sought.
If one does only look at the required number of matrix-vector multiplications, the setup of the experiments performed privileges the classical JDQR method for larger amounts of eigenvalues. This can be related to the fact that the usable subspace of the BJDQR1 algorithm becomes smaller when converged Schur vectors are locked. However, we can also argue that the BJDQR1 algorithm allows a better usage of the available mem-

ory resources because the storage reserved for the resulting Schur vectors are not used in the JDQR method at the beginning of the iteration. Furthermore, the number of vector-vector operations required for the deflation in the JDQR method increases with the number of eigenvalues found, which is not the case for the BJDQR1 algorithm.
The results for different block sizes are very promising: For some matrices with several multiple eigenvalues an appropriate block size may lead to fewer operations (in addition to using more projection vectors), but this is not always the case and thus probably difficult to exploit. In most cases the required number of operations increases moderately with the block size depending on the matrix, and it is less significant when larger numbers of eigenvalues are sought. So the additional directions obtained from the approximate solution of the block-correction equation mostly speed up the convergence of eigenvalues detected later in the iteration.
Altogether, the performance speedup of block operations could outweigh the small number of additional operations in many cases. In particular, one may also restrict the inner iteration to perform only block operations by aborting the GMRES method when one vector in the block reached its convergence criterion. This does not seem to influence the convergence rate considerably. The most interesting block sizes for possible gains in the overall performance are 2 and 4. Only for larger numbers of eigenvalues bigger block sizes (e.g. 8 or 10) may become beneficial.

7. Performance tests

Based on the numerical results discussed I demonstrate in the following that the block JDQR algorithm can be significantly faster than a single vector Jacobi-Davidson method. Here, I only consider the total runtime required by the algorithms for the calculation of a given number of exterior eigenvalues, but one has to keep in mind that small numerical deviations in the algorithms tested may already lead to slightly different numbers of inner and outer iterations of the Jacobi-Davidson method. Additionally, the initial subspace varies in the tests performed here, because different random vectors were used for its construction (due to a breakdown of the initial Arnoldi iteration for a vector of ones).

7.1. Matrices from the ESSEX project

The eigenvalue problems used for the performance measurements come from a target application of the ESSEX project. They arise in the simulation of electron spins in a magnetic field as described shortly in the introduction. In Table 6 we see an overview of the matrices used:

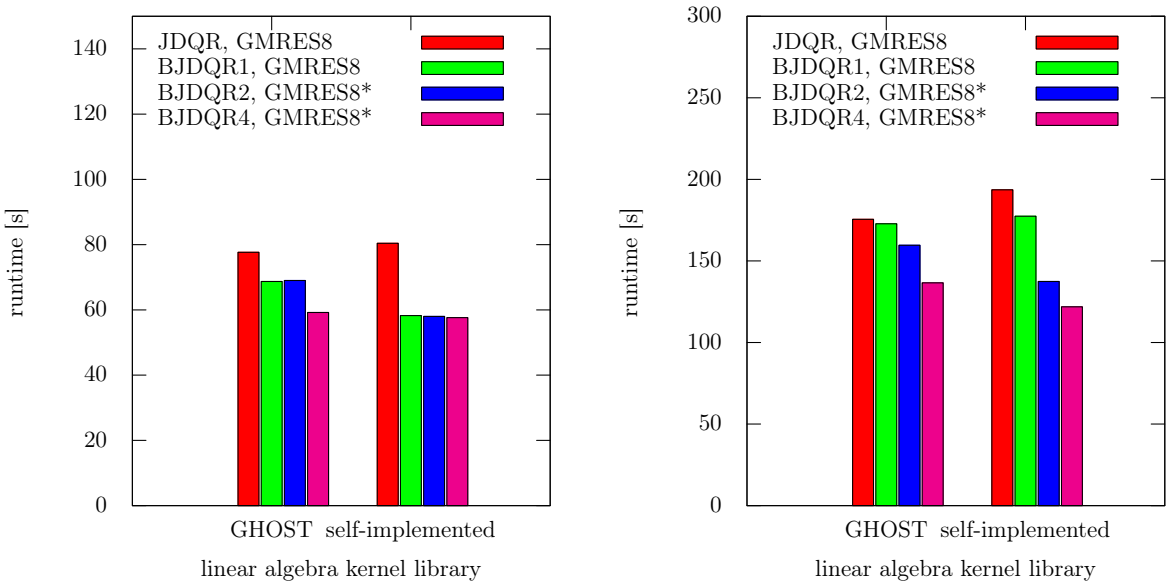
name	number of rows	non-zero count	eigenvalues sought	remarks
spin20	$\sim 10^6$	$\sim 10^7$	leftmost	real symm.
spinSZ22	$\sim 7 \cdot 10^5$	$\sim 8.8 \cdot 10^6$	leftmost	real symm.
spinSZ24	$\sim 2.7 \cdot 10^6$	$\sim 3.6 \cdot 10^7$	leftmost	real symm.
spinSZ26	$\sim 10^7$	$\sim 1.5 \cdot 10^8$	leftmost	real symm.

Table 6: Overview of the test matrices from the ESSEX project.

They all have multiple and tightly clustered eigenvalues in the desired part of the spectrum. In particular one of the leftmost eigenvalues of the matrix spin20 has a multiplicity of 8. The smallest matrix spinSZ22 is only listed here as it was used for benchmarking the sparse matrix-vector multiplication in Section 3.3. However, the numerical behavior observed for the three matrices spinSZ22 - spinSZ24 is very similar. Therefore only a few selected performance results of the matrices spinSZ24 and spinSZ26 are shown to illustrate the performance on the node-level as well as on a cluster of nodes.
The numbers in the names of the matrices (20–26) refer to the number L of electrons considered; in the spinSZ model specific symmetries are assumed such that the resulting matrices spinSZ$<L>$ have basically the same spectra as the spin$<L>$-matrices, but the eigenvalues have lower multiplicities and the matrices are much smaller.

7.2. Intra-node performance

Figure 19 presents the required runtime for the calculation of 20 eigenvalues of the matrices spin20 and spinSZ24 on a single node of the LiMa cluster. We can also compare the performance obtained with the GHOST library and my self-implemented kernel routines.



(a) Seeking 20 eigenvalues of spin20

(b) Seeking 20 eigenvalues of spinSZ24

Figure 19: Runtime required for the calculation of the 20 leftmost eigenvalues of the matrices spinSZ24 and spin20 on a single node of the RRZE's LiMa cluster using the JDQR and the BJDQR algorithm with different block sizes. Results are shown for both the GHOST library that stores blocks of vectors in column-major order as well as for my self-implemented kernel routines based on a row-major storage scheme.

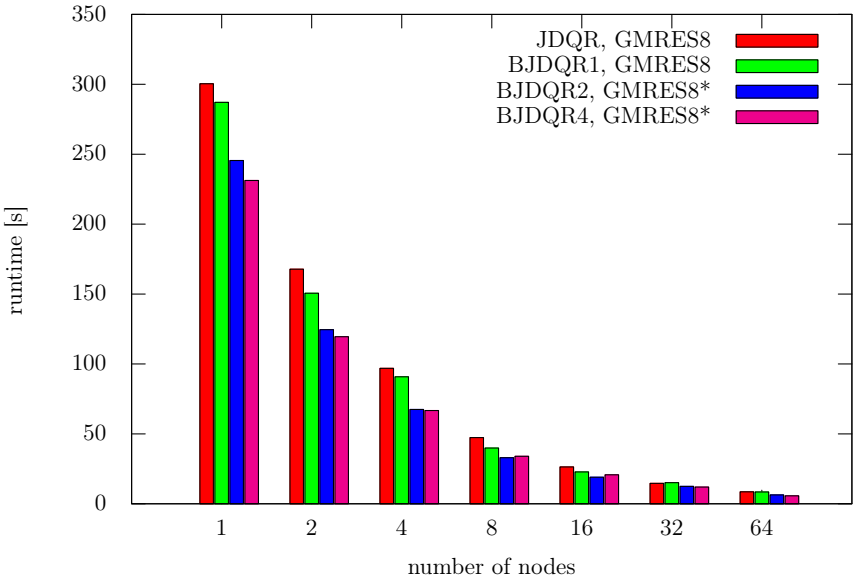
First of all, we notice that the BJDQR1 algorithm is significantly faster than the classical JDQR implementation for the matrix spin20 (with both kernel libraries). This is due to the better handling of multiple eigenvalues in my BJDQR algorithm. A bigger block size however does not speed up the computation in all cases for this matrix.
The results for the matrix spinSZ24 on the right show the typical behavior that I have also observed in other calculations with the spinSZ-matrices: There is only a small improvement from JDQR to BJDQR1 in spite of the fact that the matrices have multiple eigenvalues in the desired part of the spectrum (especially for GHOST). Changing the block size from 1 to 2 or 4 reduces the overall computation time considerably. This effect is much smaller for GHOST because the block operations in GHOST are slower. When we look at the results of the single vector JDQR method for both matrices, we notice that GHOST achieves a slightly better performance in this case. We can explain this by the fact that the row-major storage scheme yields a performance penalty when only single vectors of a complete block are accessed. In the BJDQR1 implementation I have tried to avoid operations that involve such strided accesses, but the basis vectors of the Jacobi-Davidson subspace are still stored as one big block so that almost all calculations with these basis vectors (such as the block orthogonalization) incur this

performance penalty. Nevertheless, the results underline the potential benefits of block Jacobi-Davidson methods on the node-level.

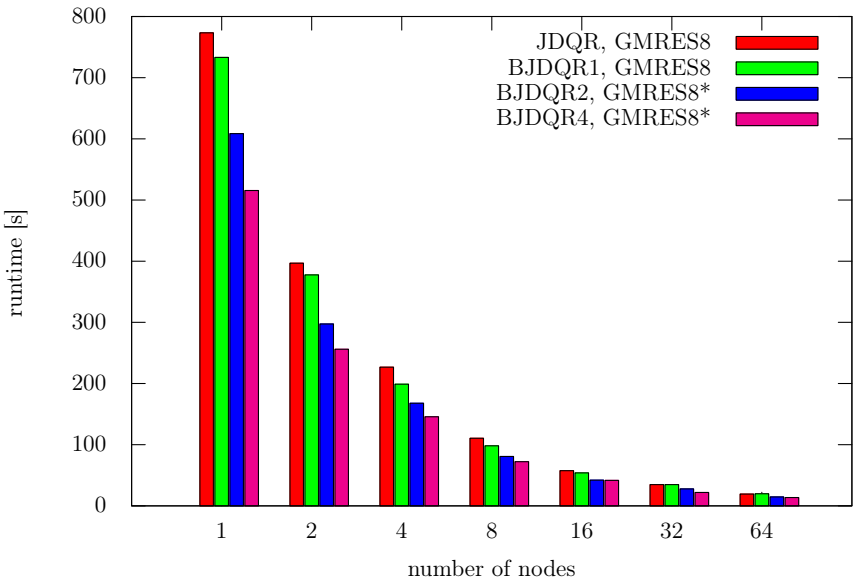
7.3. Inter-node performance

In the following I examine the performance of the BJDQR algorithm on a small to medium number of nodes on the LiMa cluster (up to 64 nodes, respectively 768 cores). I only consider results obtained with my self-implemented kernel routines in order to focus on the effects of varying the block size. It would be very insightful to investigate the behavior of the block Jacobi-Davidson method on an even larger number of nodes, where the aggregation of several global all-reduction operations should decrease the communication overhead significantly. However, this is left to future work because the current implementation lacks the possibility to overlap communication and calculations efficiently.

In Figure 20 we see the required runtime for the calculation of the 10 and 20 leftmost eigenvalues of the matrix spinSZ26 on 1-64 nodes of the LiMa cluster. In all cases BJDQR1 is at least slightly faster than the JDQR implementation.
When only 10 eigenvalues are sought, a block size of 2 improves the performance compared to the single vector computation, but increasing the block size further is not always beneficial (see Figure 20a). This reflects what we observed in Section 6: For a small number of eigenvalues increasing the block size from 2 to 4 raises the required number of operations significantly, which is not always compensated by the better performance of block operations.
When we seek 20 eigenvalues however, we expect that the number of additional operations required by the block method is more or less negligible, at least for block sizes of 2 and 4. The performance results shown in Figure 20b confirm this assumption: Compared to the single vector calculation the total runtime decreases significantly for a block size of 2 and a block size of 4 further speeds up the computation.
We essentially observe this behavior for the whole range of 1–64 nodes. To point out the performance gains achieved by the block algorithm I have plotted the block speedup, that is the quotient of the runtimes of the single vector and block vector computations (see Figure 21). Here we can see that a block size of 2 increases the overall performance by a factor of $1.2 - 1.3$ for both the calculation of 10 and of 20 eigenvalues. When 20 eigenvalues (or more) are sought, a block size of 4 increases the performance on average by a factor of ~ 1.4.

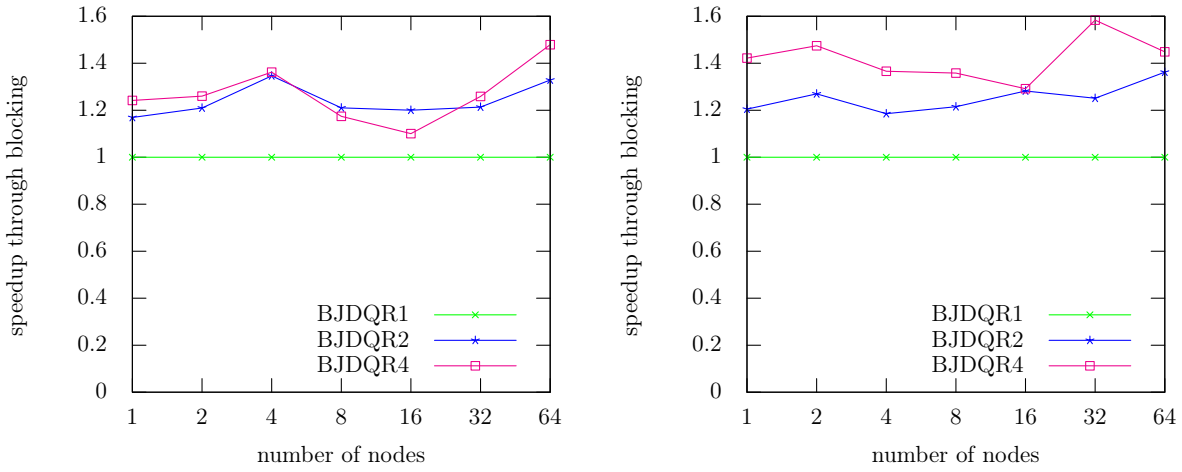


(a) Seeking 10 eigenvalues



(b) Seeking 20 eigenvalues

Figure 20: Runtime required for the calculation of the 10 respectively 20 leftmost eigenvalues of the matrix spinSZ26 on 1-64 nodes of the RRZE's LiMa cluster using the JDQR and the BJDQR algorithm with different block sizes. These results are obtained with my self-implemented kernel routines.



(a) Seeking 10 eigenvalues

(b) Seeking 20 eigenvalues

Figure 21: Speedup over BJDQR1 obtained with the block sizes 2 and 4 for the calculation of 10 respectively 20 eigenvalues of the matrix spinSZ26 on 1-64 nodes of the RRZE's LiMa cluster.

8. Conclusion

In this thesis a block Jacobi-Davidson algorithm was developed suitable for the calculation of a set of exterior eigenvalues of large sparse symmetric or non-symmetric matrices. Specific adjustments were made to improve the handling of multiple eigenvalues and to avoid unnecessary global communication operations for highly parallel calculations.

The method was tested with a set of symmetric and non-symmetric matrices from a wide range of applications, which showed that a bigger block size increases the total operation count in most cases (as already noted in previous work, see [21] respectively the references therein). Nevertheless, in many cases the amount of additional work is in a range where the performance gains through the usage of block operations could prevail. Additionally, the overhead of block methods is less significant when larger numbers of eigenvalues are sought. These conclusions can be drawn for both symmetric and non-symmetric matrices.

An important ingredient for a successful block method could be a sufficiently large subspace so that eigenvalues detected later in the Jacobi-Davidson iteration may profit from the directions calculated. This distinguishes the method analyzed here from the methods discussed for example in [21], where it is stated that block JD methods (with a very small block size) may only be beneficial in special cases for matrices with multiple or highly clustered eigenvalues.

Another interesting result of the numerical experiments performed is the fact that one can in most cases abort the linear solver in the BJDQR algorithm as soon as the first

eigenvector in a complete block reached its convergence criterion. This does not have (strong) negative effects on the convergence rate of the method. Thus, a block GMRES method (or any other approximate linear block solver for multiple right-hand sides) is probably also a good choice for the approximate solution of the correction equation. This would also allow the exploitation of fast block vector operations in the inner iteration.
As inner linear solver an unpreconditioned GMRES iteration was used in this thesis. In the ESSEX project preconditioners for the linear problems suitable for highly parallel computations still have to be investigated. This is perhaps more involved as standard methods based on incomplete factorizations are difficult to parallelize.

The current versions of the linear algebra libraries GHOST and Trilinos analyzed in this thesis did not achieve a satisfactory node-level performance for block variants of the sparse matrix-vector multiplication with multiple vectors. In contrast, it was demonstrated that an implementation using a row-major storage scheme for blocks of vectors is clearly superior because it allows the exploitation of caches. This has drawbacks for other operations however, when only single vectors out of a complete block are accessed. So one has to find a good balance to profit from an appropriate block size without slowing down operations required at other points in the algorithm. There might be storage schemes that may mostly overcome these problems (for example using a small fixed block size and storing larger blocks of vectors as sets of smaller blocks), but the main problem consists in the fact that strided accesses occur in at least one dimension.
Beyond that, the inter-node performance can further be increased by using block operations (as long as the bandwidth is not the limiting factor), which was analyzed in strong scaling experiments with the sparse matrix-vector multiplication. The author also assumes that block operations increase the scalability on highly parallel clusters and future supercomputers significantly. Unfortunately, this could not be analyzed in this thesis because the implementation of the method currently does not yet support overlapping communication and calculation and non-blocking (collective) communication. This would avoid increasing waiting times on a high number of nodes. To obtain reasonable performance on future supercomputers the algorithm also must be modified in such a way that more asynchronous calculations are possible. One way to achieve this could be a communication avoiding inner linear solver (see [8]), but one can also think of modifying the Jacobi-Davidson algorithm itself: Since the subspace iteration is very robust one could for example interleave the inner and outer iterations such that the resulting correction vectors of the inner solver are only available after an additional outer iteration. This would increase possible parallelism in the algorithm. Similarly to block methods, this obviously raises the question if the (possible) performance gains outweigh an increasing number of operations.

In short, the node-level performance of block operations for large problems heavily depends on the storage scheme of the block vectors. A block Jacobi-Davidson method can significantly speed up the calculation for small block sizes in comparison with a single vector Jacobi-Davidson implementation. This is valid for a wide range of sparse eigenvalue problems, as long as the number of desired eigenvalues is not too small.

A. Formula

A.1. From the block correction equation to a GRQI-direction

We can transform (21) to

$$\begin{aligned} & (I - \tilde{Q}\tilde{Q}^*)\left(A\Delta Q^\perp - \Delta Q^\perp \tilde{R}\right) = -(A\tilde{Q} - \tilde{Q}\tilde{R}), \\ \Leftrightarrow \qquad & A\Delta Q^\perp - \Delta Q^\perp \tilde{R} = -(A\tilde{Q} - \tilde{Q}\tilde{R}) + \tilde{Q}\tilde{Q}^* A\Delta Q^\perp \ . \end{aligned} \tag{35}$$

If we define $M \in \mathbb{C}^{l\times l}$ as $M = \tilde{Q}^* A\Delta Q^\perp$, we obtain

$$A\Delta Q^\perp - \Delta Q^\perp \tilde{R} = -(A\tilde{Q} - \tilde{Q}\tilde{R}) + \tilde{Q}M. \tag{36}$$

Assuming that M is invertible, the correction $\Delta Q^\perp$ is given by

$$\tilde{Q} + \Delta Q^\perp = X_{GRQI} M, \tag{37}$$

where X_{GRQI} is the next iterate of the GRQI method from [1]:

$$AX_{GRQI} - X_{GRQI}(\tilde{Q}^*\tilde{Q})^{-1}\tilde{Q}^* A\tilde{Q} = \tilde{Q}.$$

Note the *homogeneity* property of the GRQI algorithm (algorithm independent of representation of the subspace, e.g. invariant wrt. to right-multiplication by an invertible matrix).
The assumption above that M is invertible is probably not valid in all cases, but we have at least $M \neq 0$ because the assumption $M = 0$ leads to the contradiction $\Delta Q^\perp = Q$ since the Sylvester equation above is only singular in the case that A and $\tilde{R}$ have at least one eigenvalue in common.
So we can always show the property (37) for the largest regular submatrix of M and the corresponding columns of ΔQ.

B. Algorithms

B.1. Complete block JDQR algorithm for the generalized eigenproblem

Algorithm 9 Complete block Jacobi-Davidson QR (part 1)

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

Initialize result:

$Q \leftarrow 0, \quad Q_A \leftarrow 0, \quad Q_B \leftarrow 0, \quad R \leftarrow 0$

$l \leftarrow 0$ ▷ number of converged eigenvalues

$\tilde{n}_{Eig} \leftarrow n_{Eig} + n_b - 1$ ▷ for multiplicity detection

Initialize search space:

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 $W_B^* W = I$

$m \leftarrow m_{min}$ ▷ $m = \dim(\mathcal{W})$

$W_A \leftarrow W_{A:,1:m+1}H_{1:m+1,1:m}, \quad W \leftarrow W_{:,1:m}, \quad W_B \leftarrow W_{B:,1:m}$

$H \leftarrow (W^* W_A)$ ▷ for $B = I : H \leftarrow H_{1:m,1:m}$

Main iteration loop:

for $nIter \leftarrow 1, maxIter$ **do**

Update projected Schur form:

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

$$Q^H \leftarrow \begin{pmatrix} I & 0 \\ 0 & q^H \end{pmatrix}, \quad R^H \leftarrow \begin{pmatrix} R_{1:l,1:l} & H_{1:l,l:m}q^H \\ 0 & r^H \end{pmatrix}$$

Update approximate Schur form:

$q_i \leftarrow WQ^H_{:,i}, \quad (q_A)_i = W_A Q^H_{:,i}, \quad (q_B)_i = W_B Q^H_{:,i}, \qquad i = l, \ldots, \tilde{n}_{Eig}$

$r_i \leftarrow R^H_{1:i,i}, \qquad i = l, \ldots, \tilde{n}_{Eig}$

$res_i \leftarrow (q_A)_i - (Q_B)_{:,1:i} r_i, \qquad i = l, \ldots, \tilde{n}_{Eig}$

$\epsilon_i \leftarrow \|res_i\|_2, \qquad i = l, \ldots, \tilde{n}_{Eig}$

For Hermitian A: Reorder multiple eigenvalues in (Q^H, R^H) by ϵ_i

Update ordering of Q, Q_B, R, res

Algorithm 10 Complete block Jacobi-Davidson QR (part 2)

Check for converged eigenpairs

17: $\Delta l \leftarrow \max\{i : \epsilon_i < \epsilon_{tol}, i = 1, \ldots, n_{Eig}\} - l$
18: **if** $\Delta l > 0$ **then**
19: $W \leftarrow WQ^H{}_{:,1:m}, \quad W_A \leftarrow W_A Q^H{}_{:,1:m}, \quad W_B \leftarrow W_B Q^H{}_{:,1:m}$
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: $n_b \leftarrow \min(n_b, n_{Eig} - l)$
27: **end if**

Shrink search space:

28: **if** $m + n_b > m_{max}$ **then**
29: $W \leftarrow WQ^H{}_{:,1:m_{min}}, \quad W_A \leftarrow W_A Q^H{}_{:,1:m_{min}}, \quad W_B \leftarrow W_B Q^H{}_{:,1:m_{min}}$
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 $\tilde{l} \geq l + n_b$ depending on eigenvalue multiplicity
34: $\tilde{Q} \leftarrow Q_{1:\tilde{l}}, \quad \tilde{Q}_B \leftarrow (Q_B)_{1:\tilde{l}}$
35: **for** $i \leftarrow 1, n_b$ **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 wrt. W and $< \cdot, \cdot >_B$;
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

B.2. Example implementations in the *phist*-framework

In order to shorten and simplify the examples I have left out code and macros used for error checking; all statements accessing ierr can be wrapped with the following macro that prints an error message with a backtrace and allows a (hopefully) clean exit: PHIST_CHK_IERR(..., *ierr)).

Algorithm 11 Implementation of the *Arnoldi* process used to create an initial subspace for the BJDQR algorithm

```
1  void SUBR( simple_arnoldi )(TYPE(const_op_ptr)    A_op,
2                              TYPE(const_mvec_ptr) v0,
3                              TYPE(mvec_ptr)       V,
4                              TYPE(sdMat_ptr)      H,
5                              int                  m,
6                              int*                 ierr)
7  {
8    // define views required later
9    TYPE(mvec_ptr)  v  = NULL, Av = NULL, Vprev = NULL;
10   TYPE(sdMat_ptr) R1 = NULL, R2 = NULL;
11   SUBR( mvec_view_block )(V, &v, 0, 0,   ierr);
12   // initialize first column of V with v0 and set H to zero
13   SUBR( mvec_set_block  )(V, v0, 0, 0,   ierr);
14   SUBR( sdMat_put_value )(H, st::zero(), ierr);
15
16   for(int i = 0; i < m; i++)
17   {
18     // move views in V
19     SUBR( mvec_view_block  )(V, &Vprev, 0,   i,          ierr);
20     SUBR( mvec_view_block  )(V, &Av,    i+1, i+1,        ierr);
21     // apply the matrix operator
22     A_op->apply (st::one(), A_op->A, v, st::zero(), Av, ierr);
23
24     // move views to the next column in H
25     SUBR( sdMat_view_block )(H, &R2,    0,   i,   i, i, ierr);
26     SUBR( sdMat_view_block )(H, &R1,    i+1, i+1, i, i, ierr);
27     // orthogonalize, Q*R1 = Av - Vprev*R2; W <- Q
28     // fills in orthog. random vectors when Av is in span(Vprev)
29     SUBR( orthog )(Vprev, Av, R1, R2, 3,                ierr);
30
31     // swap the views v and Av; so Av becomes v
32     // in the next iteration and v is reused for the new Av
33     std::swap(v, Av);
34   }
35
36   // delete all local views
37   SUBR( mvec_delete  )(v,     ierr);
38   SUBR( mvec_delete  )(Av,    ierr);
39   SUBR( mvec_delete  )(Vprev, ierr);
40   SUBR( sdMat_delete )(R1,    ierr);
41   SUBR( sdMat_delete )(R2,    ierr);
42 }
```

References

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

[2] Boisvert, R. F., 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*, pages 125–137. Chapman & Hall, 1997.

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

[4] Fokkema, D. R., 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(1):94–125, January 1998, ISSN 1064-8275.

[5] Hager, G. and G. Wellein: *Introduction to High Performance Computing for Scientists and Engineers (Chapman & Hall/CRC Computational Science).* CRC Press, 2010, ISBN 143981192X.

[6] Heroux, Michael, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams: *An Overview of Trilinos.* Technical Report SAND2003-2927, Sandia National Laboratories, 2003. `http://trilinos.sandia.gov/TrilinosOverview.pdf`.

[7] Hochstenbach, M. E. and Y. Notay: *Controlling Inner Iterations in the Jacobi-Davidson Method.* SIAM Journal on Matrix Analysis and Applications, 31(2):460–477, Jan 2009.

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

[9] Intel Corporation: *Intel MPI Benchmarks. User Guide and Methodology Description*, 3.2.4 edition, 2013. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks`.

[10] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*, February 2014. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[11] Karypis, G. and K. Schloegel: *ParMETIS. Parallel Graph ParttiParti and Sparse Matrix Ordering Library.* University of Minnesota, Department of Computer Science and Engineering, Minneapolis, 4.0 edition, 2013. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis`.

[12] Klinvex, A., F. Saied, and A. Sameh: *Parallel implementations of the trace minimization scheme TraceMIN for the sparse symmetric eigenvalue problem.* Computers & Mathematics with Applications, 65(3):460–468, Feb 2013.

[13] Kreutzer, Moritz, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop: *A unified sparse matrix data format for modern processors with wide SIMD units.* ArXiv e-prints, July 2013. `http://arxiv.org/abs/1307.6209`.

[14] Nolting, Wolfgang: *Viel-Teilchen-Theorie*, volume 7 of *Grundkurs Theoretischer Physik*. Springer, 2005, ISBN 3540241175.

[15] Notay, Y.: *Convergence Analysis of Inexact Rayleigh Quotient Iteration.* SIAM Journal on Matrix Analysis and Applications, 24(3):627–644, Jan 2003.

[16] Parlett, B. N.: *The Rayleigh quotient iteration and some generalizations for nonnormal matrices.* Mathematics of Computation, 28(127):679–679, Sep 1974.

[17] Saad, Y.: *Iterative Methods for Sparse Linear Systems.* SIAM - Society for Industrial & Applied Mathematics, 2nd edition, 2003, ISBN 0898715342.

[18] Saad, Y.: *Numerical Methods for Large Eigenvalue Problems.* Classics in Applied Mathematics. SIAM - Society for Industrial & Applied Mathematics, revised ed edition, January 2011, ISBN 978-1-61197-072-2.

[19] Schubert, G., G. Hager, H. Fehske, and G. Wellein: *Parallel Sparse Matrix-Vector Multiplication as a Test Case for Hybrid MPI+OpenMP Programming*, pages 1751–1758. Institute of Electrical and Electronics Engineers, May 2011, ISBN 978-1-61284-425-1.

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

[21] Stathopoulos, A. 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(5):2162–2188, Jan 2007.

[22] Tang, P. T. P. and E. Polizzi: *Subspace Iteration with Approximate Spectral Projection.* ArXiv e-prints, Feb 2013. `http://arxiv.org/abs/1302.0432v3`.

[23] Treibig, J. and G. Hager: *Introducing a Performance Model for Bandwidth-Limited Loop Kernels.* In Wyrzykowski, Roman, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski (editors): *Parallel Processing and Applied Mathematics*, volume 6067 of *Lecture Notes in Computer Science*, pages 615–624. Springer Berlin Heidelberg, 2010.

[24] Treibig, J., G. Hager, and G. Wellein: *LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments.* In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. Institute of Electrical & Electronics Engineers (IEEE), Sep 2010, ISBN 978-1-4244-7918-4.

[25] Van der Vorst, H. A.: *Computational methods for large eigenvalue problems.* In *Solution of Equations in R^n (Part 4), Techniques of Scientific Computing (Part 4), Numerical Methods for Fluids (Part 2)*, volume 8 of *Handbook of Numerical Analysis*, pages 3–179. Elsevier, 2002.

[26] Wu, K., Y. Saad, and A. Stathopoulos: *Inexact Newton preconditioning techniques for large symmetric eigenvalue problems.* Electronic Transactions on Numerical Analysis, 7:202–214, 1998. `http://eudml.org/doc/119823`.

[27] Zhou, Y.: *Studies on Jacobi-Davidson, Rayleigh quitient iteration, inverse iteration generalized Davidson and Newton updates.* Numerical Linear Algebra with Applications, 13(8):621–642, 2006, ISSN 1099-1506.