Jül-4359

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Performance analysis and comparison of parallel eigensolvers on Blue Gene architectures

Jan Felix Münchhalfen

Berichte des Forschungszentrums Jülich 4359

Performance analysis and comparison of parallel eigensolvers on Blue Gene architectures

Jan Felix Münchhalfen

Berichte des Forschungszentrums Jülich; 4359
ISSN 0944-2952
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Jül-4359

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Abstract

The solution of eigenproblems with dense, symmetric system matrices is a core task in many fields of computational science and engineering. As the problem complexity and thus the size of the matrices involved increases, the application of distributed memory supercomputer architectures and parallel algorithms becomes inevitable. Nearly all modern algorithms for eigensolving implement a tridiagonal reduction of the eigenproblem system matrix and a subsequent solution of the tridigonalized eigenproblem. Additionally, back transformation of the eigenvectors is required if these are of interest. In the context of this thesis, implementations of two basically different approaches to the parallel solution of eigenproblems were benchmarked, reviewed and compared with particular regard to their performance on the Blue Gene/P and Blue Gene/Q supercomputers JUGENE and JUQUEEN at the Forschungszentrum Jülich: ELPA, which implements an optimized version of the divide and conquer algorithm and Elemental which utilizes the PMRRR implementation of the MR^3 algorithm. ELPA features two fundamentally different kinds of tridiagonalization, the standard one-stage and a two-stage approach. The comparision of the two-stage to the direct reduction was a primary concern in the performance analysis.

Zusammenfassung

Viele Anwendungen aus den Natur- und Ingenieurswissenschaften erfordern die Berechnung der Eigenwerte und -vektoren symmetrischer und dichtbesetzter Matrizen. Mit steigender Komplexität der Problemstellungen, d.h. mit steigender Matrixgröße, wird der Einsatz paralleler Algorithmen auf Supercomputern mit verteiltem Speicher unabdingbar. Nahezu alle modernen Algorithmen zur Lösung dieser Eigenwertprobleme implementieren zunächst eine Reduktion der Systemmatrix auf Tridiagonalgestalt und im Folgenden die Lösung des tridiagonalen Eigenwertproblems. Zusätzlich ist eine Rücktransformation der Eigenvektoren vonnöten, wenn die Anwendung diese benötigt.

In dieser Masterarbeit werden zwei neue parallele Programme zur Lösung symmetrischer Eigenwertprobleme aus den Bibliotheken Elemental und ELPA vorgestellt. In ELPA basiert die Implementierung auf dem Divide-and-Conquer Verfahren, während in Elemental der MRRR-Algorithmus (Multiple Relatively Robust Representations) genutzt wird. ELPA bietet zwei verschiedene Methoden zur Reduktion auf Tridiagonalgestalt - eine direkte und eine zweistufige. Die zweistufige Reduktion führt zu einer aufwändigeren Rücktransformation. Der Vergleich der direkten mit der zweistufigen Methode war ein vorrangiges Anliegen der Performance-Analyse. Die Performancemessungen wurden auf den Blue Gene/P- und Blue Gene/Q-Systemen JUGENE und JUQUEEN durchgeführt.

Inhaltsverzeichnis

1 Introduction

In many fields of computational sciences and engineering, numerical simulation is an important tool towards the solution of a vast variety of different problems. Examples for this range from computational problems in natural sciences as chemistry or biology like the simulation of protein folding to the very practical field of crash test simulations in modern car manufacturing companies. Frequently, the solution of eigenvalue problems is an important part of such simulations.

As the reality is mapped to a mathematical problem through simplification of a continuous problem to a discrete problem, every simulation is only an approximation to the real world. The more coarse a mathematical model maps the continuous problem, the more distant the solution of that model will be from reality. A most accurate mathematical approach is therefore desirable but also increases the complexity of the model. With increasing complexity of the mathematical models, the matrices that arise throughout the computations increase in their dimensions and frequently reach huge dimensions that only can be addressed by modern supercomputer architectures. The solution of eigenproblems is a core task of many applications that require numerical simulation. Consequently the development of efficient numerical eigensolvers is of increasing importance.

Modern supercomputers usually have a distributed memory design and consist of many thousand different and mostly homogenous individual computers to provide the needed performance and memory amount. These indivual components are interconnected through some kind of specialized network and cooperate in the solution of computational problems. With the advent of distributed memory architectures, the task of developing methods to divide the mathematical problems into smaller subproblems and to solve these subproblems efficiently arose.

The size of the matrices that arise during the computations often reflect the fineness or accuracy of the mathematical approximation to the real world problem. Often these matrices are dense, symmetric and of such large dimension that it may become impossible to store them in the memory of a single computation unit. Additionaly, computations with dense matrices often involve problems with a computational complexity of $\mathcal{O}(n^3)$. With mathematical problems of such complexity, both in memory usage and computational expenses, the development of parallel algorithms on distributed memory supercomputer architectures is required. There exist several algorithms that were specially designed for the parallel solution of eigenproblems with dense system matrices of huge dimension. Two of them are discussed in this thesis and implementations of them are then further benchmarked and compared in the following: the divide and conquer algorithm introduced by Cuppen in 1981 [10] - and the MR^3 algorithm introduced by Dhillon and Parlett in 2004[13]. Both algorithms are designed for the solution of the tridiagonal eigenproblem as tridiagonal matrices have special properties that can be exploited in the process of eigenvalue and eigenvector computation. Because the system matrices in eigenproblems are rarely tridiagonal right from the start, tridiagonalization is an essential part of the solution of eigenproblems and should not be neglected in its complexity. The implementations of the previously named algorithms both include tridiagonalization algorithms. The Elemental library[22] only features direct tridiagonalization at this point(version 0.75). It utilizes the PMRRR[2] implementation of the MR^3 algorithm developed by Matthias Petschow. The divide and conquer algorithm is implemented in the ELPA library[15]. This library is the result of a collaborative effort of several German research institutes. It features two different kinds of tridiagonalization: A direct approach as it is implemented in Elemental and a two-stage approach as it was introduced by Bischof et al.[6]. Since the back transformation of the eigenvectors is of increased computational complexity when the two-stage approach is utilized, the crossline in runtime of the eigensolvers that feature direct and two-stage tridiagonalization was an important aspect of this thesis.

Both libraries were benchmarked for test matrices of varying sizes and percentages of the eigenspectrum on two different supercomputers: The Blue Gene/P installation JUGENE, and the Blue Gene/Q installation JUQUEEN at the Forschungszentrum Jülich. The results of these benchmarks are visualized and discussed in chapter 5.

2 Mathematical principals

This chapter introduces the basic mathematical definitions and terms in context to the eigenvalue problems that will be discussed in the following chapters.

2.1 Basic definitions

Matrices and vectors are commonly used terms in linear algebra. A system of linear equations

$$\begin{array}{ccccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + & \dots & + & a_{1,n}x_n & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + & \dots & + & a_{2,n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m,1}x_1 & + & a_{m,2}x_2 & + & \dots & + & a_{m,n}x_n & = & b_m \end{array}$$

can be written as $Ax = b$, where A is a matrix representing the two-dimensional grid of equation coefficients $a_{i,j}$, and x and b are the vectors with elements $x_1, \dots, x_n$ and $b_1, \dots, b_m$ respectively.

Depending on the elements type we either say $A \in \mathbb{R}^{m \times n}, x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ for real numbers or $A \in \mathbb{C}^{n \times m}, x \in \mathbb{C}^n$ and $b \in \mathbb{C}^m$ for complex numbers. Multiplication of matrices and vectors is defined as seen in the equation system above. The multiplication of two vectors is defined by the scalar product.

Definition 2.1.1. Let $u, v \in \mathbb{C}^n$ be two complex column vectors, then

$$< u, v > := \sum_{j=1}^{n} \bar{u}_j v_j \tag{2.1.1}$$

is the **scalar product** of both vectors. Here, $\bar{u}_j$ is the **complex conjugate** value of u_j.

Remark 2.1.2. If for two vectors $u, v \in \mathbb{C}$ the scalar product condition

$$< u, v > = 0 \tag{2.1.2}$$

is true, these vectors are called **orthogonal**. This relationship is denoted as $u \perp v$.

Definition 2.1.3. Given a vector $x \in \mathbb{C}^n$,

$$\|x\|_2 := \sqrt{< x, x >} = \sqrt{\sum_{j=1}^{n} \bar{x}_j x_j} \tag{2.1.3}$$

defines the euclidean norm. This norm represents the length of a vector in the euclidean space. If not stated otherwise, $\|\cdot\|$ refers to the euclidean norm $\|\cdot\|_2$.

Definition 2.1.4. A matrix $B \in \mathbb{C}^{m\times n}$ is called the **adjoint matrix** of a matrix $A \in \mathbb{C}^{n\times m}$ if the equation

$$b_{j,i} = \bar{a}_{i,j} \tag{2.1.4}$$

is true for each of its elements. The adjoint matrix is written as $\bar{A}^T$ or A^*. For a real matrix, B is called the **transposed matrix** A^T of A.

Definition 2.1.5. For two complex matrices $A \in \mathbb{C}^{m\times k}$ and $B \in \mathbb{C}^{k\times n}$, the product $AB = C, C \in \mathbb{C}^{m\times n}$ is defined via the matrix-matrix multiplication:

$$\underset{\substack{1\le i\le m\\ 1\le j\le n}}{\forall} : c_{i,j} = \sum_{l=1}^{k} a_{i,l} b_{l,j} \tag{2.1.5}$$

Remark 2.1.6. Note that the scalar product of two complex vectors $u \in \mathbb{C}^n$ and $v \in \mathbb{C}^n$ can be expressed via the matrix multiplication if the first vector is conjugated and transposed as follows:

$$< u, v >= u^* v$$

Definition 2.1.7. A square matrix $A \in \mathbb{C}^{n\times n}$ is called **hermitian**, if it is **self-adjoint**, that means $A = A^*$. If A is a real matrix, it is called **symmetric** and $A = A^T$.

Definition 2.1.8. A matrix $A \in \mathbb{C}^{n\times n}$ is called **regular**, **nonsingular** or **invertible** if there exists a matrix $A^{-1} \in \mathbb{C}^{n\times n}$ so that

$$AA^{-1} = I,$$

where I stands for the identity matrix. The matrix A^{-1} is called the **inverse matrix** of A.

Definition 2.1.9. Let $A \in \mathbb{C}^{n\times n}$ be a complex square matrix. If the equation

$$A^*A = I, \tag{2.1.6}$$

is true for A, it is called **unitary**. In this case, the matrix has the inverse $A^{-1} = A^*$. Each row of the matrix is orthogonal to every other row and has the norm 1. Together they form an **orthogonal basis** in $\mathbb{C}^{n\times n}$. If for a matrix $A \in \mathbb{R}^{n\times n}$

$$A^{-1} = A^T,$$

is true, A is called an **orthogonal matrix** or **orthonormal matrix**.

Definition 2.1.10. Two matrices $A, \tilde{A} \in \mathbb{C}^{n\times n}$ are said to be **similar** if they are related by

$$\tilde{A} = C^{-1}AC, \tag{2.1.7}$$

where $C \in \mathbb{C}^{n\times n}$ is an arbitrary nonsingular matrix. The transformation through C (2.1.7), is called a **similarity transformation**

Remark 2.1.11. For an **orthogonal similarity transformation**,

$$\tilde{A} = Q^*AQ$$

with Q unitary, the similar matrix $\tilde{A}$ of a hermitian matrix A is hermitian again:

$$\tilde{A}^* = (Q^*AQ)^* = Q^*A^*Q^{**} = Q^*AQ = \tilde{A}$$

2.2 Eigenvalue problem

Definition 2.2.1. Let A be a complex matrix $A \in \mathbb{C}^{n\times n}$. The problem of finding the solutions for the following equation

$$Av = \lambda v, \text{w.l.o.g. } \|v\| = 1 \tag{2.2.1}$$

is known as the eigenvalue problem. The scalar $\lambda \in \mathbb{C}$ is called the **eigenvalue** and the vector $v \in \mathbb{C}^n$ is called **eigenvector**. The pair of both is called **eigenpair** and is denoted as follows: (λ, v).

Theorem 2.2.2. The equation (2.2.1) is equivalent to the problem

$$Av - \lambda v = 0 \Leftrightarrow (A - \lambda I)v = 0$$

and therefore is a homogeneous linear equation system.
Furthermore, if the determinant of the equation system differs from 0, then it only has the trivial solution $v = \vec{0}$.

If the determinant equals zero,

$$det(A - \lambda I) = 0 \tag{2.2.2}$$

there exists an infinite number of solutions for v, depending on λ. The roots of the **characteristic polynomial** (2.2.2) thus are the **eigenvalues** of A.

In real-world applications, frequently hermitian or symmetric matrices are encountered. This special type of matrix has certain properties which allows to develop efficient algorithms for eigenvalue computation. The following theorems turn out to be very useful in that context.

Theorem 2.2.3. All the eigenvalues of a hermitian matrix are real.

Proof: Let $\lambda \in \mathbb{C}$ be an eigenvalue of the hermitian matrix $A \in \mathbb{C}^{n\times n}$ with corresponding eigenvector $z \in \mathbb{C}^n$ and $\|z\| = 1$. Because z has the norm 1, the scalar product with itself $< z, z >= 1^2$ equals 1 as well. Therefore it follows

$$\lambda = z^*\lambda z = z^*Az = z^*A^*z = (Az)^*z = (\lambda z)^*z = z^*\bar{\lambda}z = \bar{\lambda}$$

and thus $\lambda = \bar{\lambda}$. From this follows that $\lambda \in \mathbb{R}$.

Theorem 2.2.4. The eigenvectors corresponding to different eigenvalues of a hermitian matrix are linearly independent and pairwise orthogonal.

Proof: Let $\lambda_1, \lambda_2 \in \mathbb{R}, \lambda_1 \neq \lambda_2$ be some arbitrarily chosen eigenvalues of a hermitian matrix $A \in \mathbb{C}^{n\times n}$. Furthermore let $v_1, v_2 \in \mathbb{C}^{n\times n}$ be the corresponding eigenvectors. Then the following holds:

$$\begin{array}{ccccccc} & \lambda_2 < v_1, v_2 > & = & \lambda_2(v_1^* v_2) & = & v_1^*\lambda_2 v_2 & = & v_1^* A v_2 \\ = & (A^* v_1)^* v_2 & = & (Av_1)^* v_2 & = & (\lambda_1 v_1)^* v_2 & = & v_1^*\bar{\lambda}_1 v_2 \\ = & \bar{\lambda}_1(v_1^* v_2) & = & \bar{\lambda}_1 < v_1, v_2 > & & & & \end{array}$$

As proven in 2.2.3, all eigenvalues of the hermitian matrix are real. From this it follows that $\lambda_2 < v_1, v_2 >= \lambda_1 < v_1, v_2 >$, which either can be true if $\lambda_1 = \lambda_2$ or $< v_1, v_2 >= 0$. Because $\lambda_1 \neq \lambda_2$ was the initial assumption, it is necessary for the scalar product that $< v_1, v_2 >= 0$.
Therefore the vectors are orthogonal.

□

Theorem 2.2.5. For a given matrix $A \in \mathbb{C}^{n\times n}$ a unitary matrix $Q \in \mathbb{C}^{n\times n}$ can be found so that

$$R = Q^*AQ, \tag{2.2.3}$$

where $R \in \mathbb{C}^{n\times n}$ is an upper triangular matrix.

Proof: For the decomposition of A at least one eigenvector v_1 of the matrix A is needed. This vector can be extended to an orthonormal basis $v_1, y_2, \ldots, y_n$ in $\mathbb{C}^n$ using the **Gram-Schmidt process**. The generated vectors are further combined into the columns of an unitary matrix Q_1 which fulfills the equation

$$Q_1^*AQ_1 = \left[\begin{array}{c|ccc} \lambda_1 & * & \ldots & * \\ \hline 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right]$$

where $A_1 \in \mathbb{C}^{(n-1)\times(n-1)}$ is a modified submatrix of A.

The next step is to calculate another eigenpair of the submatrix A_1 and repeat the above process to determine another unitary matrix $Q_2 \in \mathbb{C}^{(n-1)\times(n-1)}$. This matrix Q_2 is extended to a matrix $\tilde{Q}_2$ as follows:

$$\tilde{Q}_2 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & Q_2 & \\ 0 & & & \end{array}\right]$$

so that the unitarity is preserved.

If both of these matrices Q_1 and $\tilde{Q}_2$ are applied to A as denoted in the following equation

$$\tilde{Q}_2^*Q_1^*AQ_1\tilde{Q}_2 = \left[\begin{array}{cc|ccc} \lambda_1 & * & * & \cdots & * \\ 0 & \lambda_2 & * & \cdots & * \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & A_2 & \\ 0 & 0 & & & \end{array}\right],$$

then the eigenvalues appear on the diagonal of the resulting matrix, where $A_2 \in \mathbb{C}^{(n-2)\times(n-2)}$. This can be inductively repeated for matrices $Q_i, i = 1\ldots(n-1)$, to determine a unitary matrix Q as their product:

$$Q = Q_1 \tilde{Q}_2 \dots \tilde{Q}_{n-1}$$

In fact, the matrix Q is the very matrix that fulfills (2.2.3).

□

Definition 2.2.6. The decomposition in equation (2.2.3) is known as the **Schur decomposition** of a given matrix $A \in \mathbb{C}^{n \times n}$.

It is an orthogonal similarity transformation as introduced in definition 2.1.10. For hermitian matrices $A \in \mathbb{C}^{n \times n}$, the matrix $R \in \mathbb{C}^{n \times n}$ is hermitian, too because of remark 2.1.11. Therefore R is a diagonal matrix with the eigenvalues of A on its main diagonal. If A is hermitian, the matrix $R \in \mathbb{R}^{n \times n}$ is denoted as Λ.

The decomposition

$$\Lambda = Q^* A Q$$

is called the **spectral decomposition** of the hermitian matrix A.

Concerning eigenvalue computations, similarity transformations have some special properties that numerical algorithms can benefit from. The following two theorems prove to be extremely useful in this context.

Theorem 2.2.7. Consider a similarity transformation $\tilde{A} = C^{-1}AC$ of $A \in \mathbb{C}^{n \times n}$ with the transformation matrix C. If (λ_i, v_i) are the eigenpairs of A, the eigenpairs of $\tilde{A}$ can be expressed as

$$(\lambda_i, C^{-1}v_i), i = 1 \dots n$$

While the eigenvectors change, the eigenvalues are invariant to the similarity transformation.

Proof:

Let (λ, v) be an eigenpair of an arbitrary matrix $A \in \mathbb{C}^{n \times n}$ and $C \in \mathbb{C}^{n \times n}$ a transformation matrix. The following equation has to be proven:

$$\tilde{A}(C^{-1}v) = \lambda(C^{-1}v) \tag{2.2.4}$$

Substituting $\tilde{A}$ it follows that

$$\begin{array}{rccc} & (C^{-1}AC)(C^{-1}v) & = & \lambda(C^{-1}v) \\ \Leftrightarrow & C^{-1}(Av) & = & C^{-1}(\lambda v) \\ \Leftrightarrow & Av & = & \lambda v \end{array}$$

While (λ, v) is an eigenpair of the matrix A, the last equation is always true. Because all transformations in the proof were equivalent transformations, equation (2.2.4) is true, too. Therefore theorem 2.2.7 is proven.

□

Remark 2.2.8. The spectral decomposition of A as introducuded in definition 2.2.6 is equivalent to $AQ = Q\Lambda$, which is the matrix representation of the eigenvalue problem with Λ containing the eigenvalues of A on its main diagonal and the columns of Q representing the eigenvectors of A.

Theorem 2.2.9. The eigenvectors of an arbitrary matrix $A \in \mathbb{C}^{n\times n}$ are invariant to shifts. For a matrix $(A + \mu I)$ with shift $\mu \in \mathbb{C}$ the eigenpairs conform to

$$(\lambda_i + \mu, v_i),$$

if (λ_i, v_i) is an eigenpair of A.

Proof:
Let (λ, v) be an eigenpair of an arbitrary matrix A. For A it is true that:

$$\begin{array}{rccc} & Av & = & \lambda v \\ \Leftrightarrow & Av + \mu v & = & \lambda v + \mu v \\ \Leftrightarrow & (A + \mu I)v & = & (\lambda + \mu)v \end{array}$$

And therefore, as the last equation equals the definition of the eigenproblem itself (2.2.1), $(\lambda + \mu, v)$ is an eigenpair of $(A + \mu I)$.

□

3 Algorithms

This chapter will describe the general approach towards the numerical solution of eigenvalue problems. All following approaches expect a matrix $A \in \mathbb{R}^{n \times n}$, which is symmetric and dense. At first the general approach for symmetric, dense matrices will be explained. In the following, the numerical approaches towards the solution of the eigenvalue problem will be discussed in full detail.

3.1 General approach

The general approach for the numerical solution of an eigenvalue problem with a symmetric dense system matrix $A \in \mathbb{R}^{n \times n}$ consists of 3 steps.

1. The dense, symmetric matrix $A \in \mathbb{R}^{n \times n}$ is reduced to a matrix $T \in \mathbb{R}^{n \times n}$ of tridiagonal, symmetric form. The reduction is accomplished using only orthogonal similarity transformations as introduced in remark 2.1.11. A very common approach utilizes Householder transformation matrices and will be subsequently discussed (another well-known class of orthogonal similarity transformations are the Givens rotations).

2. In the next step the tridiagonal eigenproblem

   $$Tz = \lambda z, z \in \mathbb{R}^n \tag{3.1.1}$$

   has to be solved. Several algorithms exist for the solution of this problem, two of them will be discussed in more detail in the context of this thesis. Because only similarity transformations were used to reduce the matrix A to T, the eigenvalues of both are identical as proven in theorem 2.2.7.

3. In contrast to the eigenvalues, the eigenvectors of A and T are not the same. As proven in theorem 2.2.7, if w is an eigenvector of T, then the corresponding eigenvector v of A can be calculated as $v = Q^{-1}w$, where Q was the orthogonal similarity transformation matrix that reduced A to T. This back transformation will be discussed in detail later on. Because the eigenvalues of T and A are the same, the last step is only necessary if the eigenvectors are of interest.

3.2 Tridiagonalization

As shown in definition 2.2.6, for a symmetric matrix $A \in \mathbb{R}^{n\times n}$, an orthogonal matrix $Q \in \mathbb{R}^{n\times n}$ can be found so that

$$Q^T AQ = T, \tag{3.2.1}$$

T is a symmetric tridiagonal matrix. The construction of the orthogonal matrix can be achieved using either Housholder or Givens transformations. Speaking of either one the process is called **Householder reduction**, or **Givens reduction**. Givens transformations are linear transformations which represent a rotation in the euclidean space. The matrix Q is constructed via the product of several of Givens rotations. One matrix has to be calculated for each element that is not on the main diagonal or the first minor diagonal and differs from zero. The Householder transformation on the other hand, represents a reflection in the euclidean space. If the vector that identifies the reflection hyperplane is chosen optimal, the matrix Q can be constructed via the product of much less Householder matrices than Givens matrices would be needed in the equivalent reduction method. Because the complexity of the Householder reduction is, in most cases, much lower than the complexity of the Givens reduction, the latter will not be discussed any further.

A Householder transformation matrix $H \in \mathbb{R}^{n\times n}$ for the vectors $x, z \in \mathbb{R}^n$ with $\|x\| = \|z\|$ is defined as

$$H = I - 2yy^T, y = \frac{x - z}{\|x - z\|}$$

The matrix H then is a symmetric orthogonal matrix and suffices

$$z = Hx$$

For each $k \in \{1, \dots, n-2\}$ vectors can be chosen so that the constructed Householder matrices H_k fulfill:

$$H_k x = H_k \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ x_{k+2} \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

If now y_k is chosen as

$$\tilde{a}_k = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_{k,k+1} \\ a_{k,k+2} \\ \vdots \\ a_{k,n} \end{pmatrix} \text{ and } y_k = \frac{\tilde{a}_k - \|\tilde{a}_k\| e_{k+1}}{\|\tilde{a}_k - \|\tilde{a}_k\| e_{k+1}\|},$$

the corresponding Householder matrix $H_k = I - 2y_k y_k^T$ exists and fulfills

$$H_k a_k = H_k \begin{pmatrix} a_{k,1} \\ \vdots \\ a_{k,k} \\ a_{k,k+1} \\ a_{k,k+2} \\ \vdots \\ a_{k,n} \end{pmatrix} = \begin{pmatrix} a_{k,1} \\ \vdots \\ a_{k,k} \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

In the following, the k-dimensional submatrix of A, with the first $(n-k)$ columns and rows truncated, will be denoted as $A^{(k)}$.

Applying the first orthogonal similarity transformation via multiplication of the Householder matrix H_1 onto the matrix A results in:

$$H_1^T A H_1 = \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & H_1^{(n-1)} & \\ 0 & & & \end{array}\right) \left(\begin{array}{c|ccc} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \hline a_{2,1} & & & \\ \vdots & & A^{(n-1)} & \\ a_{n,1} & & & \end{array}\right) \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & H_1^{(n-1)} & \\ 0 & & & \end{array}\right)$$

$$= \left(\begin{array}{c|cccc} a_{1,1} & \beta_2 & 0 & \cdots & 0 \\ \hline \beta_2 & & & & \\ 0 & & \tilde{A}_1^{(n-1)} & & \\ \vdots & & & & \\ 0 & & & & \end{array}\right) = \tilde{A}_1$$

As can be seen in the above equation, the multiplication of $H_1^T = H_1$ from the left side eliminates elements below the first minor diagonal of the first column, while the multiplication of H_1 from the right side eliminates elements next to the the right of the first minor diagonal of the first row of A. In the next step, the newly constructed Householder matrix H_2 will be applied in the same way to the matrix $\tilde{A}_1$ to achieve the same effect for the second column and row:

$$H_2^T \tilde{A}_1 H_2 = \left(\begin{array}{cc|cccc} a_{1,1} & \beta_2 & 0 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & 0 & \cdots & 0 \\ \hline 0 & \beta_3 & & & & \\ 0 & 0 & & \tilde{A}_2^{(n-2)} & & \\ \vdots & \vdots & & & & \\ 0 & 0 & & & & \end{array}\right) = \tilde{A}_2$$

Using these similarity transformations, the matrix A is subsequently reduced to a tridiagonal matrix T that is *similiar* to A in $(n-2)$ steps. The complete reduction can be expressed as a product of Householder matrices:

$$\begin{aligned} Q &= H_1 \dots H_{n-2} \\ Q^T &= H_{n-2} \dots H_1 \end{aligned}$$

and

$$Q^T A Q = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha n-1 & \beta_n \\ & & & \beta_n & \alpha_n \end{pmatrix}$$

As the product of symmetric matrices is not necessarily symmetric, Q^T is not equal to Q while this is true for the individual Householder matrices H_i. The product of the orthonormal matrices is itself orthonormal and in fact the very matrix that satisfies (3.2.1).

For numerical benefits, the Householder transformation can be split up into

$$\begin{aligned}
H^TAH &= (I-2yy^T)A(I-2yy^T)\\
&= (A-2yy^TA)(I-2yy^T)\\
&= A-2yy^TA-2Ayy^T+4yy^TAyy^T\\
&= A-2yy^TA+2yy^TAyy^T-2Ayy^T+2yy^TAyy^T\\
&= A-y\underbrace{(2y^TA-2y^TAyy^T)}_{:=v^T}-\underbrace{(2Ay-2yy^TAy)}_{:=v}y^T
\end{aligned}$$

As v^T is the transposed of the vector v, the numerical benefit is that the Householder transformation can now be expressed via:

$$\begin{aligned}
H^TAH &= A-yv^T-vy^T\\
v &= 2Ay-2yy^TAy
\end{aligned}$$

which, with $y, v \in \mathbb{R}^n$, is a rank 1-update and therefore a level-2 BLAS operation.

```
Input : A
for ($k = 1, \ldots, (n-2)$) do
    $\mu = \|\tilde{a}_k\|$
    $y = \frac{\tilde{a}_k - \mu e_{k+1}}{\|\tilde{a}_k - \mu e_{k+1}\|}$
    $v = 2Ay - 2yy^TAy$
    $A_{k+1,k} = \mu$
    $A_{k,k+1} = \mu$
    $A_{k+2:n,k} = \vec{0}$
    $A_{k,k+2:n} = \vec{0}^T$
    $A_{k+1:n,k+1:n} = A_{k+1:n,k+1:n} - (yv^T)_{k+1:n,k+1:n} - (vy^T)_{k+1:n,k+1:n}$
end
return A
```

Algorithm 3.2.1: Householder reduction of symmetric matrix $A \in \mathbb{R}^{n\times n}$. The matrix A is overridden by the tridiagonalization.

Reducing a symmetric matrix via the Householder reduction requires approximately $\frac{4n^3}{3}$ Flops[21]. Most operations involved are Level-2 BLAS(i.e. matrix-vector operations[1]). If the eigenvectors of A are of interest, the matrix Q is needed for the back transformation step. While algorithm 3.2.1 does not provide the similarity transformation matrix Q, the computational costs in this

[1]in this case rank-1 updates

case increase to $\frac{8n^3}{3}$ Flops[18]. As will be seen later in the discussion of the back transformation step, Q itself will never be accumulated, but the Householder matrices will again be applied subsequently.

Because Level-3 BLAS(i.e. matrix-matrix operations) are far more efficient than Level-2 BLAS[21], Bischof and Van Loan[7] developed the WY-representation for a sequence of Householder matrix products whichs allows to exploit the computational benefits of Level-3-BLAS routines.

Let Q_k be the product of k Householder matrices, then the following applies:

$$Q_k = H_1 \dots H_k = I - W_k Y_k^T \quad \text{where} \quad W_k, Y_k \in \mathbb{R}^{n \times k}$$

The construction of the matrices W_k and Y_k can be shown inductively. Beginning with $k = 1$,

$$Q_1 = H_1 = I - 2y_1 y_1^T = I - W_1 Y_1^T$$

applies.

Let $(A|b)$ be a matrix that consists of A concatenated with b as the last column vector. In the step from k to $k+1$ the matrices W_k and Y_k are known and it applies that:

$$\begin{aligned} Q_{k+1} = H_1 \dots H_k H_{k+1} &= (I - W_k Y_k^T)(I - 2y_{k+1} y_{k+1}^T) \\ &= I - W_k Y_k^T - Q_k 2 y_{k+1} y_{k+1}^T \\ &= I - \underbrace{(W_k | Q_k 2 y_{k+1})}_{W_{k+1}} \underbrace{(Y_k | y_{k+1})^T}_{Y_{k+1}^T} \end{aligned}$$

This representation is used in blocked Householder transformation algorithms. The matrix A can not be directly reduced to tridiagonal form implying this representation and is first reduced to a band matrix of bandwidth k and then, in a second step, further reduced to tridiagonal form using single orthogonal transformations[7]. The WY-representation allows to use the more efficient level-3 BLAS as

$$Q_k^T A Q_k = A - 2W_k Y_k^T - 2Y_k^T W_k \tag{3.2.2}$$

with $W, Y \in \mathbb{R}^{n \times k}$, is a rank k-update and therefore can be implemented utilizing level-3 BLAS.

If the resulting band matrix is sparse, Givens transformations can be computationally more efficient than the Householder approach.

The WY-representation comes to use in the ELPA two-stage solver.

3.3 Solution of the tridiagonal eigenvalue problem

This section will introduce two different algorithms for the solution of the tridiagonal eigenvalue problem (3.1.1) as they are used in the libraries that will be compared in the context of this thesis.

3.3.1 Divide and conquer

The divide and conquer algorithm was introduced by Cuppen in 1981 [10]. It is an algorithm which calculates the spectral decomposition of a given symmetric tridiagonal matrix T.
The matrix T is decomposed into two submatrices, for each of which the spectral decomposition is calculated independently. These individual spectral decompositions are then merged into the spectral decomposition of T. The algorithm is recursive, because the spectral decompositon of the two submatrices is again accomplished using the divide and conquer approach.

If the matrix T has no 0-elements on the off-diagonal it can be decomposed to:

$$T = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_{n-1} & \beta_n \\ & & & \beta_n & \alpha_n \end{pmatrix} = \left(\begin{array}{c|c} T_1 & \\ & \beta_{m+1} \\ \hline \beta_{m+1} & \\ & T_2 \end{array}\right) \tag{3.3.1}$$

$$= \left(\begin{array}{c|c} \hat{T}_1 & \\ \hline & \hat{T}_2 \end{array}\right) + \left(\begin{array}{c|c} \beta_{m+1} & \beta_{m+1} \\ \hline \beta_{m+1} & \beta_{m+1} \end{array}\right) \tag{3.3.2}$$

So for $m < n$, T can be decomposed into two symmetric tridiagonal submatrices $\hat{T}_1 \in \mathbb{R}^{m \times m}$, $\hat{T}_2 \in \mathbb{R}^{(n-m)\times(n-m)}$ and a one-rank correction matrix K_{m+1} which is defined as

$$K_{m+1} = \beta_{m+1} v v^T = \beta_{m+1} \begin{pmatrix} e_m \\ e_1 \end{pmatrix} \begin{pmatrix} e_m^T & e_1^T \end{pmatrix}$$

Suppose the spectral decompositions of the matrices $\hat{T}_1$ and $\hat{T}_2$ are given by

$$\begin{aligned}\hat{T}_1 &= Q_1\Lambda_1 Q_1^T\\ \hat{T}_2 &= Q_2\Lambda_2 Q_2^T\end{aligned}$$

where Q_1, Q_2 are orthonormal matrices and Λ_1, Λ_2 are diagonal, the spectral decomposition of T can be calculated out of these two smaller decompositions as follows:

$$\begin{aligned}T &= \begin{pmatrix} Q_1\Lambda_1Q_1^T & \\ & Q_2\Lambda_2Q_2^T\end{pmatrix} + K_{m+1}\\ &= \begin{pmatrix} Q_1 & \\ & Q_2\end{pmatrix}\begin{pmatrix} \Lambda_1 & \\ & \Lambda_2\end{pmatrix}\begin{pmatrix} Q_1^T & \\ & Q_2^T\end{pmatrix} + \beta_{m+1}vv^T\\ &= \underbrace{\begin{pmatrix} Q_1 & \\ & Q_2\end{pmatrix}}_{Q_{1,2}}\left(\underbrace{\begin{pmatrix} \Lambda_1 & \\ & \Lambda_2\end{pmatrix}}_{\Lambda_{1,2}} + \beta_{m+1}\hat{v}\hat{v}^T\right)\begin{pmatrix} Q_1^T & \\ & Q_2^T\end{pmatrix}\end{aligned}$$

with

$$\hat{v} = \begin{pmatrix} Q_1^T & 0\\ 0 & Q_2^T\end{pmatrix} v = \begin{pmatrix} Q_1^Te_m\\ Q_2^Te_1\end{pmatrix}$$

In the following process the spectral decomposition of $\Lambda_{1,2} + \beta_{m+1}\hat{v}\hat{v}^T$

$$\Lambda_{1,2} + \beta_{m+1}\hat{v}\hat{v}^T = Z\Lambda Z^T \tag{3.3.3}$$

has to be calculated, so that the spectral decompositions of the submatrices can be further reduced into the spectral decomposition of T, as shown in the following equation:

$$T = Q_{1,2}(\Lambda_{1,2} + \beta_{m+1}\hat{v}\hat{v}^T)Q_{1,2}^T = Q_{1,2}(Z\Lambda Z^T)Q_{1,2}^T = (Q_{1,2}Z)\Lambda(Q_{1,2}Z)^T$$

The last part of this equation is the full spectral decomposition of T, reduced out of the partial spectral decompositions of $\hat{T}_1$ and $\hat{T}_2$. Here the diagonal matrix $\Lambda \in \mathbb{R}^{n\times n}$ contains the eigenvalues of T while the columns of $Q_{1,2}Z$ represent the eigenvectors.

The decomposition in equation (3.3.3) can be achieved using a theorem that restates the results of Golub[17] as well as Bunch, Nielsen and Sorensen[9] who added the explicit formula for the calculation of the eigenvectors.

Theorem 3.3.1. [10] If D is a diagonal matrix D=diag$(d_1, \dots, d_n), n \geq 2$, with $d_1 < d_2 < \cdots < d_n$, $z \in \mathbb{R}^n$ is a vector with $z_i \neq 0$ for $i \in \{1, \dots, n\}$ and $\rho > 0$ a scalar, then the eigenvalues of the matrix $D + \rho z z^T$ are equal to the n roots $\lambda_1 < \cdots < \lambda_n$ of the rational function

$$\begin{aligned} w(\lambda) &= 1 + \rho z^T (D - \lambda I)^{-1} z \qquad (3.3.4) \\ &= 1 + \rho \sum_{j=1}^{n} \frac{z_j^2}{d_j - \lambda} \end{aligned}$$

The corresponding eigenvectors $p^1, \dots, p^n$ of $D + \rho z z^T$ are given by

$$p^i = (D - \lambda_i I)^{-1} z / \|(D - \lambda_i I)^{-1} z\|_2 \qquad (3.3.5)$$

and the d_i strictly seperate the eigenvalues as follows:

$$d_1 < \lambda_1 < d_2 < \lambda_2 < \cdots < d_n < \lambda_n < d_n + \rho z^T z \qquad (3.3.6)$$

Proof: An eigenpair (λ, p) of $D + \rho z z^T$ satifies

$$(D + \rho z z^T) p = \lambda p$$

or with little transformation

$$(D - \lambda I) p = -\rho z^T p z$$

We now show that $D - \lambda I$ is nonsingular. Assuming $D - \lambda I$ is singular, we have $\lambda = d_i$ for some i which results in $((D - \lambda I)p)_i = 0 = -\rho z^T p z_i$.
Because $z_i \neq 0$ by initial assumption, we have $z^T p = 0$ which again results in $(D - \lambda I)p = -\rho z^T p z = 0$ and therefore $(d_j - \lambda) p_j = 0$ for all j. This again means that $p_j = 0$ for $j \neq i$. So $0 = z^T p = z_i p_i$ must be fulfilled which, as p is an eigenvector and has at least one element differing from 0, contradicts the initial assumption $z \neq 0$. Therefore $D - \lambda I$ is nonsingular and we have:

$$p = -\rho z^T p (D - \lambda I)^{-1} z$$

which consequently satifies equation (3.3.5) and further yields that

$$\begin{aligned} & p &=& \quad -\rho z^T p (D - \lambda I)^{-1} z \\ \Leftrightarrow\ & z^T p &=& \quad -\rho z^T z^T p (D - \lambda I)^{-1} z \\ \Leftrightarrow\ & z^T p &=& \quad -\rho z^T (D - \lambda I)^{-1} z z^T p \\ \Leftrightarrow\ & 0 &=& \quad z^T p + \rho z^T (D - \lambda I)^{-1} z z^T p \\ \Leftrightarrow\ & 0 &=& \quad (1 + \rho z^T (D - \lambda I)^{-1} z) z^T p \end{aligned}$$

Because $z^T p \neq 0$ as proven above, the last equation requires that

$$1 + \rho z^T (D - \lambda I)^{-1} z = 0$$

which proves that λ is a root of equation (3.3.4).
(3.3.6) easily follows from the behavior of $w(\lambda)$.

□

Remark 3.3.2. If D contains mutually equal elements and/or some components of z are zero, deflation is possible and the eigenproblem can be reduced to two problems of smaller size. Without loss of generality it can be assumed that $z_i \neq 0 \, \forall i \in \{1 \dots n\}$ and $d_i \neq d_j \, \forall i \neq j$[10].

The matrices $\hat{T}_1$ and $\hat{T}_2$ are consecutively split using the divide and conquer method until the partial eigenproblems reach the dimension 1, in which case the solution is trivial, or the individual problems of smaller dimensions are solved using different algorithms. The pseudocode for the divide and conquer method is given below.

```
Input : T
if ( $T \in \mathbb{R}^{1\times 1}$ ) then
    $\Lambda = T$
    $Q = 1$
else
    [$\hat{T}_1, \hat{T}_2, \beta_{m+1}, v$] = divide($T$)
    [$\Lambda_1, Q_1$] = divide_conquer($\hat{T}_1$)
    [$\Lambda_2, Q_2$] = divide_conquer($\hat{T}_2$)
    $\hat{v} = Q_{1,2}^T v$
    [$\Lambda, Z$] = solve_secular($\beta_{m+1}, \Lambda_{1,2}, \hat{v}$)
    $Q = Q_{1,2} Z$
end
return [$\Lambda, Q$]
```

Algorithm 3.3.1: The recursive function divide_conquer. It is initially called with a tridiagonal matrix T as the argument. In the further process it splits the matrix into smaller sub-matrices until they reach the dimension 1. Then it reduces the individual results of the sub-problems into the spectral decomposition of the original matrices the sub-problems were split from. The method divide accomplishes a decomposition as described in (3.3.1) and (3.3.2). The method solve_secular calculates the solution to the secular spectral decomposition as introduced in (3.3.3)

The efficiency of the divide and conquer algorithm depends on the relative placement of the eigenvalues. As the roots of the secular equation (3.3.3) are mostly calculated using iterative methods, the average computational effort amounts to $\mathcal{O}(n^{2.3})$ and varies between $\mathcal{O}(n^2)$ in best case and $\mathcal{O}(n^3)$ in worst case scenarios.[11] With this algorithm all eigenvalues and eigenvectors are calculated.

3.3.2 MRRR

The Algorithm of Multiple Relatively Robust Representations, shortly denoted as MRRR or MR^3, was introduced by Dhillon and Parlett in 2004[13]. It is an algorithm that can either calculate the full spectrum of eigenvalues, or only parts of it. It uses relatively robust representations as introduced by Dhillon in his Ph.D. Thesis in 1997[12]. The relatively robust representation of a matrix A is a representation that aims to produce minimal errors in the eigenvalues and eigenvectors under a certain amount of disturbance. To define a RRR, another definition in context to the eigenvalues has to be made first.

Definition 3.3.3. The function relgap calculates the minimal relative deviation of a scalar $y \in \mathbb{R}$ from a finite set of scalars $\mathbb{X} = \{x_i\}, \mathbb{X} \subset \mathbb{R}$[12].

$$relgap(y, \mathbb{X}) = \min_{x \in \mathbb{X}} \frac{\|y - x\|}{\|y\|}$$

Definition 3.3.4. A **relatively robust representation** is a finite field of real numbers $\mathbb{X} = \{x_i\}, x_i \in \mathbb{R}$ defining a matrix A in such a way that if x_i is disturbed by ϵ_i with $x_i(1 + \epsilon_i)$, the following equations apply for all $1 \leq j \leq n$:

$$\frac{|\delta\lambda_j|}{|\lambda_j|} = \mathcal{O}\left(\sum_i \epsilon_i\right)$$

$$|\sin \angle(z_j, z_j + \delta z_j)| = \mathcal{O}\left(\frac{\sum\limits_i \epsilon_i}{relgap(\lambda_j, \{\lambda_k \mid k \neq j\})}\right)$$

Here λ_j denotes the j-th eigenvalue whereas $\lambda_j + \delta\lambda_j$ represents the corresponding disturbed eigenvalue. The eigenvectors are denoted as z_j or $z_j + \delta z_j$ respectively.

A relatively robust representation of a matrix A therefore is a representation that results in minimal errors in the eigenvectors and eigenvalues when disturbed within certain boundaries. When this is only true for a part of the eigenvectors or eigenvalues, the representation is called a **partial relatively robust representation**.

The MR^3 algorithm does not exactly use fields of real numbers $\{x_i\}$ as introduced in the above definition, but shifted representations of the tridiagonal matrix T that differs by a scalar $\mu \in \mathbb{R}$ as denoted in the following:

$$L_c D_c L_c^T = T - \mu I$$

The decomposition $L_c D_c L_c^T$ is a lower triangular decomposition, such as the Cholesky decomposition. In the special case of tridiagonal matrices T, the decomposition results in a bidiagonal $L \in \mathbb{R}^{n \times n}$ with $l_{ii} = 1, i = 1 \ldots n$ and a diagonal $D \in \mathbb{R}^{n \times n}$.

As it is not always possible to calculate a full RRR, the MR^3 algorithm concentractes on partial RRR's near individual clusters of eigenvalues and tries to find such representations using a trial and error approach. If $T - \mu I$ is positive definite,

$$x^T(T - \mu I)x > 0 : \underset{x \in \mathbb{R}^n \setminus \{0\}}{\forall}$$

the decomposition $L_c D_c L_c^T$ is always a RRR for all of its eigenvalues[14]. The crucial part of the algorithm is to distinguish the individual clusters of eigenvalues. Therefore the algorithm, in a first step, calculates all of the requested eigenvalues using bisection. Let $\hat{\mathbb{L}} = \{\hat{\lambda}_i\}$ be the set of calculated eigenvalues of the RRR. A single eigenvalue $\hat{\lambda}_k$ is considered **isolated** if,

$$\underset{1 \leq i \leq n}{\forall} : relgap(\hat{\lambda}_k, \hat{\mathbb{L}} \setminus \{\hat{\lambda}_k\}) > tol \tag{3.3.7}$$

is satified. Here *tol* is a certain predefined tolerance (e.g. 10^{-3} [13]) that has to be exceeded in order for the *getvec* routine[13] utilized by the MR^3 algorithm to calculate numerically orthogonal eigenvectors without using an explicit approach such as the Gram-Schmidt orthogonalization. This method implements QD-transformations and the twisted factorization to calculate a relatively highly accurate eigenvector corresponding to a certain eigenvalue $\hat{\lambda}_k$. If equation (3.3.7) is not fulfilled for the eigenvalues of $L_c D_c L_c^T$, they are called **clustered** and the getvec routine can not calculate numerically orthogonal eigenvectors for $L_c D_c L_c^T$. It is not able to calculate numerically orthogonal eigenvectors for eigenvalues of $L_c D_c L_c^T$ meaning equation (3.3.7) is not satisfied on them.

The algorithm constructs *clusters* of eigenvalue Γ_c: The resulting clusters are denoted as $m \leq n$ finite sets. Isolated eigenvalues build up their own cluster, whereas eigenvalues that lie close to each other and do not satisfy equation 3.3.7 are merged into clusters with more than one eigenvalue Γ_c. Then the MR^3-algoritm processes each of these clusters independently. If a cluster identifies an isolated eigenvalue, which means that

$$\|\Gamma_c\| = 1$$

holds, then the *getvec* routine is called directly to calculate the corresponding eigenvector. If the cluster consists of more than one eigenvalue, the current RRR can not be used to calculate their eigenvectors using the *getvec* routine and a new RRR has to be calculated. The MR^3-algorithm therefore picks a $\tau_c \in \mathbb{R}$ near one of the eigenvalues in the cluster and computes

$$L_c D_c L_c^T = LDL^T - \tau_c I$$

It then checks if $L_c D_c L_c^T$ meets the requirements that were made in definition 3.3.4 of a RRR and if not, repeats the step until an adequate RRR is found.

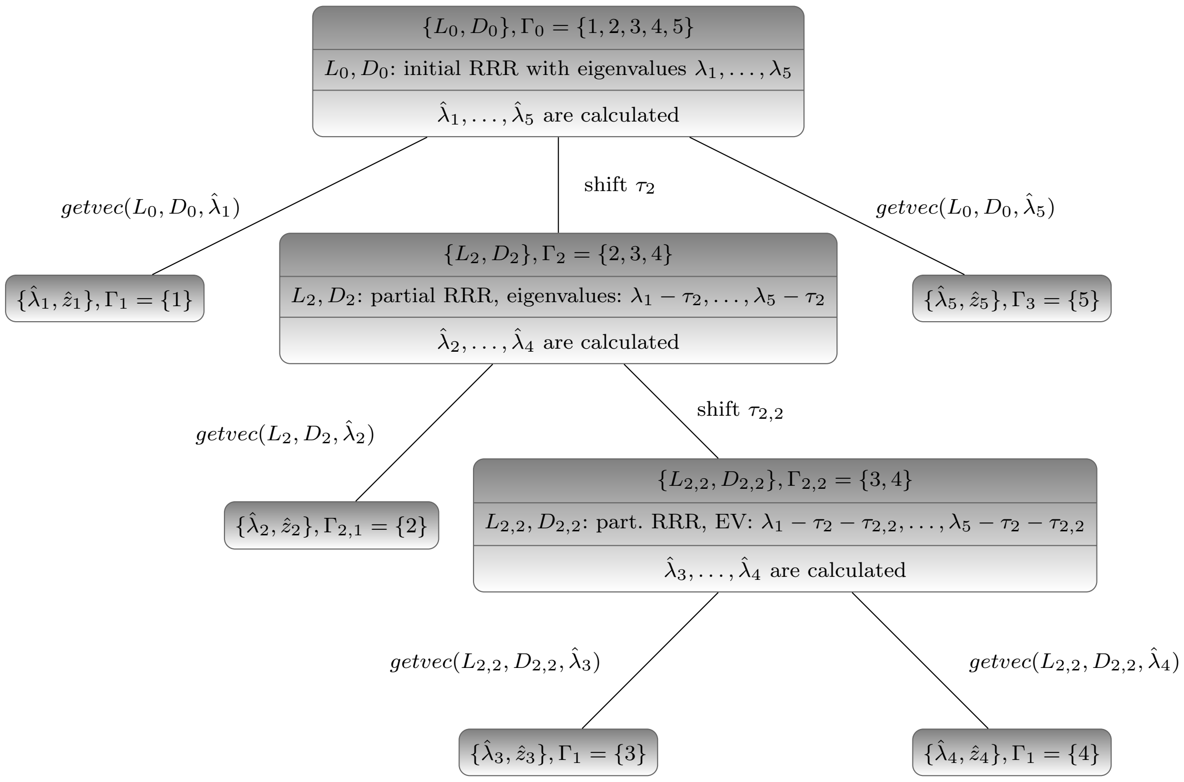

$\{L_0, D_0\}, \Gamma_0 = \{1,2,3,4,5\}$
L_0, D_0: initial RRR with eigenvalues $\lambda_1, \dots, \lambda_5$
$\hat{\lambda}_1, \dots, \hat{\lambda}_5$ are calculated
$getvec(L_0, D_0, \hat{\lambda}_1)$
shift τ_2
$getvec(L_0, D_0, \hat{\lambda}_5)$
$\{\hat{\lambda}_1, \hat{z}_1\}, \Gamma_1 = \{1\}$
$\{L_2, D_2\}, \Gamma_2 = \{2,3,4\}$
L_2, D_2: partial RRR, eigenvalues: $\lambda_1 - \tau_2, \dots, \lambda_5 - \tau_2$
$\hat{\lambda}_2, \dots, \hat{\lambda}_4$ are calculated
$\{\hat{\lambda}_5, \hat{z}_5\}, \Gamma_3 = \{5\}$
$getvec(L_2, D_2, \hat{\lambda}_2)$
shift $\tau_{2,2}$
$\{\hat{\lambda}_2, \hat{z}_2\}, \Gamma_{2,1} = \{2\}$
$\{L_{2,2}, D_{2,2}\}, \Gamma_{2,2} = \{3,4\}$
$L_{2,2}, D_{2,2}$: part. RRR, EV: $\lambda_1 - \tau_2 - \tau_{2,2}, \dots, \lambda_5 - \tau_2 - \tau_{2,2}$
$\hat{\lambda}_3, \dots, \hat{\lambda}_4$ are calculated
$getvec(L_{2,2}, D_{2,2}, \hat{\lambda}_3)$
$getvec(L_{2,2}, D_{2,2}, \hat{\lambda}_4)$
$\{\hat{\lambda}_3, \hat{z}_3\}, \Gamma_1 = \{3\}$
$\{\hat{\lambda}_4, \hat{z}_4\}, \Gamma_1 = \{4\}$

Abbildung 3.1: Example of a representation tree: the tridiagonal matrix $T \in \mathbb{R}^{5\times 5}$ has 3 eigenvalues in a cluster in the initial representation. After step 1 of the algorithm, λ_3 and λ_4 still are clustered.

Once a RRR for the cluster has been found, it repeats the clustering process for the new RRR, whose eigenvalues are now shifted by τ_c, (see theorem 2.2.9), and show different relative gaps than in the precedent RRR. Therefore some of the eigenvalues that were clustered in a previous step of the algorithm, possibly exceed the condition in equation (3.3.7), and thus being considered *isolated* permit the calculation of their corresponding eigenvectors utilizing the *getvec* routine now. As also proven in theorem 2.2.9, the eigenvectors are invariant to shifts like those that were used to calculate $L_c D_c L_c^T$, and resemble the eigenvectors of the initial representation of T under the aspects of floating point accuracy.

The clustering process in the MR^3-algorithm can be visualized using a *representation tree* as can be seen in figure 3.1. In the representation tree, Γ_c contains the indices of the eigenvalues that are to be calculated during a specific recursion step of the algorithm. A matrix for the example shown in figure

3.1 could have the following eigenvalues with a tolerance limit of 10^{-3}.

$$\begin{aligned}
\lambda_1 &= 0 \\
\lambda_2 &= 2 \\
\lambda_3 &= 2 + 10^{-10} \\
\lambda_4 &= 2 + 10^{-14} \\
\lambda_5 &= 3
\end{aligned}$$

With the chosen tolerance, λ_2, λ_3 and λ_4 would form a cluster in the first algorithm step while λ_1 and λ_5 would be considered isolated. Therefore λ_1, λ_5 would be calculated using the *getvec* routine, while the algorithm would search a new partial RRR for $\lambda_2, \dots, \lambda_4$. After a shift with e.g. $\tau_1 = 2$, the eigenvalues of the partial RRR now resemble $\hat{\lambda_i} = \lambda_i - \tau_1$ and show larger relative gaps than in the previous representation. In the clustering process, $\hat{\lambda}_3$ and $\hat{\lambda}_4$ form a new cluster $\Gamma_{2,2}$ while $\hat{\lambda}_2$ shows a relative gap above the chosen tolerance limit and is considered isolated.

The following pseudocode describes the MR^3-algorithm:

```
Input : $T, \Gamma, tol$
1 Choose $\mu$ such that $T - \mu I$ is a RRR that determines the
      desired eigenvalues and eigenvectors to high relative
      accuracy. Chosing $T - \mu I$ as positive definite is a safe
      choice.
2 $L, D$ = Cholesky( $T - \mu I$ )
3 Calculate the eigenvalues $\hat{\lambda}_i, i \in \Gamma$ of $LDL^T$ using bisection or
      the dqds-algorithm and add them to $\hat{\mathbb{L}}$
4 $[\Lambda, Z]$ = MRRR_recursion( $L, D, \Gamma, \hat{\mathbb{L}}, tol$ )
5 $\Lambda = \Lambda + \mu I$
6 return $[\Lambda, Z]$
```

Algorithm 3.3.2: MR^3-algorithm: It starts with a tridiagonal matrix $T \in \mathbb{R}^{n \times n}$, a set of indices of desired eigenvalues Γ and a tolerance limit *tol*. The algorithm calculates the desired eigenvalues and eigenvectors utilizing a recursive routine *MRRR_recursion*. The algorithm returns $\Lambda \in \mathbb{R}^{k \times n}$ with the eigenvalues on the main diagonal and $Z \in \mathbb{R}^{n \times k}$ with the eigenvectors on the columns.

The MR^3-algorithm takes $\mathcal{O}(nk)$ FLOPS for the calculation of k eigenpairs of a tridiagonal matrix $T \in \mathbb{R}^{n \times n}$. For a full spectrum calculation it would therefore need $\mathcal{O}(n^2)$ FLOPS just as the divide and conquer algorithm in the best case. Unlike the divide and conquer algorithm, the worst-case scenario takes $\mathcal{O}(nk)$ FLOPS, too, so the computational effort stays the same independent of the type of matrix.[14] The MR^3-algorithms delivers eigenvectors that are guaranteed to be orthogonal. In the contrary - other algorithms do not necessarily ensure the orthogonality of eigenvectors.

```
Input : $L, D, \Gamma, \hat{\mathbb{L}}, tol$
1 Group the eigenvalues $\hat{\lambda}_x, \dots, \hat{\lambda}_{x+\|\hat{\mathbb{L}}\|-1} \in \hat{\mathbb{L}}$ in $m$ clusters. Fill $\Gamma_c$
     with the indices of the eigenvalues in the $c$-th cluster.
2 for ($c$ = $1, \dots, m$) do
3  if ($\|\Gamma_c\| = 1$) then
4   j = element in $\Gamma_c$
5   $z_j$ = getvec( $L, D, \hat{\lambda}_j$ )
6  else
7   Chose $\tau_c$ near the eigenvalues in the cluster.
8   $[L_c, D_c]$ = dstqds( $LDL^T - \tau_c I$ )
9   Recalculate the eigenvalues $\hat{\lambda}_i$ of the new RRR $L_c D_c L_c^T$ for
       all $i \in \Gamma_c$ and add them to $\hat{\mathbb{L}}_c$.
10  $[\hat{\Lambda}_c, Z_c]$ = MRRR_recursion( $L_c, D_c, \Gamma_c, \hat{\mathbb{L}}_c, tol$ )
11  $\hat{\lambda}_i$ = $\hat{\lambda}_i + \tau_c$ for all $i \in \Gamma_c$
12  end
13 end
14 return $[\hat{\Lambda}, Z]$
```

Algorithm 3.3.3: MRRR_recursion: The function calculates the eigenvectors to all eigenvalues in the set $\hat{\mathbb{L}}$. It first distinguishes between *isolated* and *clustered* eigenvalues. The eigenvectors of *isolated* eigenvalues can be determined directly, using the *getvec* routine. Clustered eigenvalues are grouped, then a new partial RRR is calculated and the recursion starts again. The return values contain the eigenvalues and eigenvectors calculated in the substeps of this function.

3.4 Back transformation

After the eigenvalues and eigenvectors of the matrix $T \in \mathbb{R}^{n \times n}$ are calculated the eigenvectors need to be transformed back to those of the matrix $A \in \mathbb{R}^{n \times n}$ from which it originated.

The orthogonality transformation used to accomplish the reduction to tridiagonal form was introduced in (3.2.1):

$$Q^T A Q = T$$

As proven in theorem (2.2.7), the eigenvalues of a matrix are invariant to similarity transformations and therefore do not need back transformation. The eigenvectors however are affected by both shifts and similarity transformations and require further processing to match the eigenvectors of A.

The back transformation of a eigenvector is accomplished using the transformation

$$v = Qz$$

which basically follows from theorem (2.2.7). It can also be done for all eigenvectors simultaniously using matrix-matrix multiplication.

$$\Upsilon = QZ$$

where $Z \in \mathbb{R}^{n \times n}$ contains the eigenvectors of T on its columns, $\Upsilon \in \mathbb{R}^{n \times n}$ contains the eigenvectors of A on its columns and Q is the transformation matrix that was used to reduce A to T. In real world applications, the matrix Q is not explicitly formed but the individual Householder matrices are applied in reverse order to the eigenvectors.

```
Input : $Z, H_1, \dots, H_{n-2}$
for ($i = (n-2), \dots, 1$) do
    $Z = H_i Z$
end
return $Z$
```

Algorithm 3.4.1: Back transformation of the eigenvectors: The matrix Z is multiplicated in the reverse order by the Householder matrices that were used to reduce A to T

Transforming the eigenvectors back so that they correspond to the original matrix A has a computational effort of about $\mathcal{O}(2n^2m)$ FLOPS when transforming m eigenvectors. The back transformation can be realized very efficiently using almost only Level-3-BLAS.

As the ELPA library includes a two-stage tridiagonalization approach that uses specialized algorithms for the banded to tridiagonal transformation and only utilizes Householder matrices for the full to banded reduction, the costs of back transformation increase by $2kn^2$ FLOPS compared to this direct tridiagonalization approach. While the efficiency of the reduction may achieve a benefit in runtime for the tridiagonalization itself, the back transformation step strikes back with this increased computational expenses. The intensity of this - and whether the two-stage approach is better than the direct approach or not - primarily depends on k, the number of eigenvectors that are to be back transformed.

4 Architectures and libraries

This chapter will deal with the architectures and libraries that were used throughout this work. Two different architectures of the *Blue Gene* family were used to compare and analyze the eigensolver libraries *Elemental* and *EL-PA*, which will be discussed in chapter 4.2. In the following the *Blue Gene* family of supercomputers will be introduced, beginning with the *Blue Gene/P* architecture in chapter 4.1.1, followed by *Blue Gene/Q*, the contemporary successor in chapter 4.1.2.

4.1 Blue Gene

The Blue Gene project was initiated in December 1999 as a five-year effort to build a massively parallel supercomputer for simulating phenomena such as protein folding[3]. The project had two main goals: to advance the understanding of the mechanics behind protein folding and to explore novel ideas in the field of massivly parallel computer architectures. A major effort of the project was to identify the requirements the machine should suffice to meet the scientific goals. The first system to emanate this project was the Blue Gene/L that was set up in the Lawrence Livermore National Laboratory (LLNL) in November 2004. It was a 16-rack system with 1024 compute nodes per rack and each compute node holding two PowerPC 440 cores at a clock rate of 700 MHZ, resulting in 32.768 processor cores total. The system achieved the first place in the TOP500 list [24] with a LINPACK performance of 70.72 TFLOPS and held this position for 3.5 years. From 2004 to 2008 it was enlarged to a final installation of 104 racks which corresponds to 212.992 processor cores in total with a peak performance of 596 TFLOPS and a LINPACK performance of 478 TFLOPS. While this first Blue Gene/L installation was the largest, many smaller Blue Gene/L installations followed. In November 2006 there were 27 systems in the TOP500 list using the Blue Gene/L architecture[25].

The second generation of the Blue Gene family of supercomputers that was delivered was the Blue Gene/P architecture. A Blue Gene/P installation was assembled at the Forschungszentrum Jülich in November 2007 under the name JUGENE. It will be further described in section 4.1.1. The third and latest generation of the Blue Gene family is the Blue Gene/Q architecture. One installation of the Blue Gene/Q architecture that was set up in the LLNL has reached a peak performance of 20 PFLOPS in 2012 and continues to expand. In June 2012 a Blue Gene/Q installation was set up in Jülich under the name JUQUEEN. The Blue Gene/Q architecture will be discussed in detail in section 4.1.2.

4.1.1 Blue Gene/P (JUGENE)

The Blue Gene/P architecture consists of various hardware components of which the smallest is considered to be the chip. The chip has 4 processors (and there is one chip per compute card). Each the four PowerPC 450 cores that reside on a compute card are clocked at 850 MHZ. There are 32 compute cards on a node card and 32 node cards per rack (4096 processors per rack). Figure 4.1 shows the individual hardware components of the Blue Gene/P architecture.

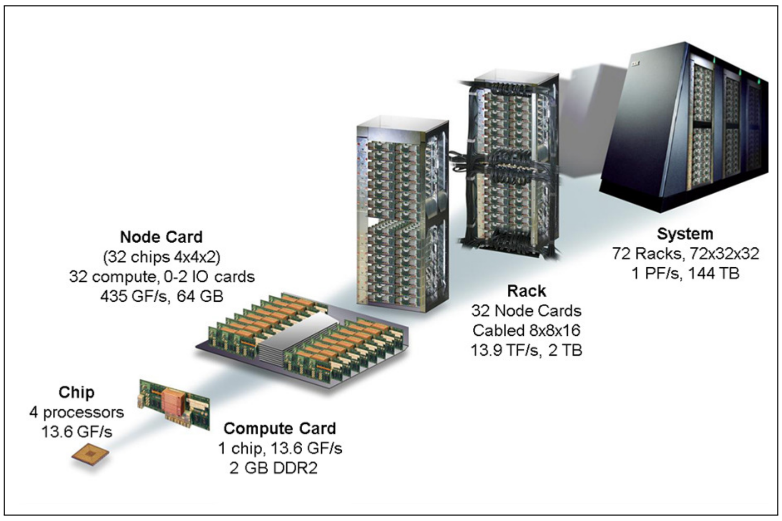


Abbildung 4.1: Step by step composition of the Blue Gene/P architecture

The JUGENE was a Blue Gene-P installation at the Jülich Supercomputing Centre (JSC) of the Forschungszentrum Jülich. It was in service from November 2007 until the end of July 2012 and at the time it was introduced the second fastest supercomputer in the world according to the TOP500 list[26]. The first installation consisted of 16 racks each hosting 1024 compute nodes with a peak performance of 167.3 TFLOPS.[1] Two years after the inauguration of JUGENE, from May to June 2009, it was expanded to 72 racks (294.912 cores) and reached the PFLOPS mark as the first European supercomputer. At this point it had a theoretical peak performance of 1002.70 TFLOPS. In the LINPACK benchmarks it reached a R_{max} of 825.50 TFLOPS.

The following table explains the detailed structure of the Blue Gene/P machine JUGENE as it was in July 2012.

Basic Structure	• 72 Racks - 73.728 nodes (294.912 cores) • 2 midplanes a 16 node cards per rack (4096 cores) • 32 compute cards per node card (128 cores) • 4 processor cores per compute card/per chip
Processor type	• Power PC 450, 32-bit, 850 MHz, 4-way SMP • L3 Cache: shared, 8 MB • 2 GB main memory, 13.6 GB/s bandwidth • 13.6 GFLOPS R_{max}
Network	• three dimensional torus for communication between compute cards (bandwidth per link: 425 MB/s, hardware latency: 100ns - 800ns) • global tree for collective operations (bandwidth per link: 850 MB/s) • barrier low latency and interrupt network • External: 10 GigE / Functional network (for I/O)
Performance	• 1 PFLOPS peak performance • 825.5 TFLOPS LINPACK performance

The peak performance is a purely theoretical value that represents the performance under optimal cooperation of the different hardware components. It represents the performance that could be achieved if all components of the machine were perfectly utilized. The LINPACK performance is the actual performance that was achieved during the LINPACK benchmark. The value of 825.5 TFLOPS represents an 80% effiency for this test.

With 4 cores per compute card, the Blue Gene/P has twice as many cores as the Blue Gene/L. The clock rate was increased from 700 MHZ to 850 MHZ while the number of compute cards per rack remained the same.

4.1.2 Blue Gene/Q (JUQUEEN)

The Blue Gene/Q is the most recent architecture of the Blue Gene series. It was designed to reach 20 PFLOPS until 2011. The first installation went in production at the Lawrence Livermore National Laboratory(LLNL) in 2011 as part of the *Advanced Simulation and Computing Program* under the name IBM Sequoia. The supercomputer architecture consists of compute cards, node cards, midplanes and racks as the predecessors but features 16 cores per compute card, a doubled clock rate compared to Blue Gene/P of 1.6 GHz and 16 GBytes of main memory per compute card. A node card includes 32 compute cards and 16 node cards form a midplane. This results in a processor count of 8192 cores per midplane. A single rack contains two midplanes. As the cores of the Blue Gene/Q implement 4-way hyperthreading, they can execute between 1 and 4 threads simultaneously. To exploit the 4-way hyperthreading to the limit, at least 2 threads have to be executed per core. This results in the theoretical maximum of 32,768 threads executed in parallel per rack.

By 2012 this first installation was expanded to 96 racks(1572864 cores) and

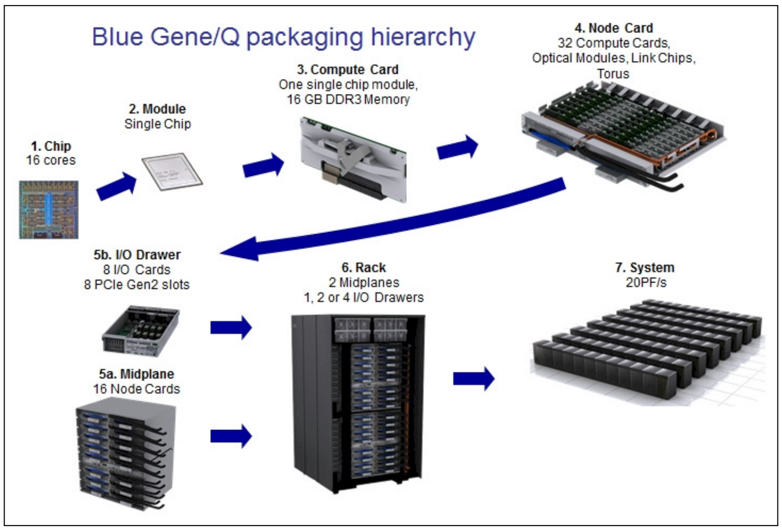


Abbildung 4.2: Step by step composition of the Blue Gene/Q architecture

reached a R_{peak} of 20.13 PFLOPS and a Linpack R_{max} of 16.32 PFLOPS as originally intended by IBM[23]. According to the TOP500 it ranked in position 1. This effectively made it the fastest supercomputer in the world as of July 2012. The LLNL installation held position 1 in the Green500 list, with a performance of 2100.88 MFLOPSW, which also made it the most energy efficient supercomputer in the world as of June 2012[19]. Compared to the previously fastest and now second fastest supercomputer, the K computer, which resides at RIKEN Advanced Institute for Computational Science and has a result of 830.18 MFLOPS/W, it is more than twice as energy efficient.

In contrast to the Blue Gene/P design, the network for communcation between the nodes was realized using not a 3- but a 5-dimensional torus with increased bandwidth compared to the predecessor.

The following table explains the detailed structure of the Blue Gene/Q installation JUQUEEN as it was at the time of this work.

Basic Structure	• 8 racks - 8.192 nodes (131.072 cores) • 2 midplanes a 16 node cards per rack (16384 cores) • 32 compute cards per node card (128 cores) • 16 processor cores per compute card
Processor type	• IBM PowerPC©A2 1.6 GHz, 4-way SMP • 16 GB SDRAM-DDR3 (per compute card)
Network	• five dimensional torus for communication between compute cards (bandwidth: 5 GB/s, hardware latency: 2.5 μsec worst case) • collective network as part of the 5D torus • global barrier/interrupt network as part of the 5D torus • 1 GBit control network for System Boot, Debug, Monitoring, not part of the torus, no interference
Performance	• 1.67 PFLOPS peak performance • 1.38 TFLOPS LINPACK performance

4.2 Libraries

The two different libraries that were used throughout this work are developed for the use on massively parallel distributed memory architectures. There will be a general focus on data distribution aspects as this is one of the fundamental differences between ELPA and Elemental.

4.2.1 ELPA

ELPA (Eigenvalue SoLvers for Petaflop Architectures) was a project initiated with the aim to develop and implement an efficient eigenvalue solver for petaflop architectures. It was founded by the German Federal Ministry of Education and Research. The task was adressed by a collaborative consortium consisting of several German research institutes[15].

Throughout this work, ELPA Version 1.1.2 was used for all test cases.

4.2.1.1 Structure

ELPA is implemented as a set of Fortran subroutines that can be either compiled as a seperate library or as part of a program. The compilation process is relatively trivial using standard compilers. The compiled modules can be linked against any Fortran or C, C++ program respectively very easily. It is meant as a drop-in replacement for certain ScaLAPACK calls and is not autonomous in that context. In fact it relies on certain aspects of data distribution that are specific to ScaLAPACK. The communication in ELPA itself is realized using only MPI[1]-calls. As ELPA is dependant on ScaLAPACK, BLACS[2] is also a necessary prerequisite. It is a communication library that builds on top of MPI to fit the communcation aspects to the needs of linear algebra. To achieve that goal it arranges the processors in a two dimensional processor-grid over which the matrices and vectors are distributed. As ScaLAPACK does, ELPA makes heavy use of the BLAS[3], a standardized set of routines that are frequently used in linear algebra. There exist machine specific versions of the BLAS, which are highly optimized and sometimes implemented using assembly language or assembly intrinsics for Fortran. On the Blue Gene/P and Blue Gene/Q the ESSL[4] was used as the BLAS implementation. The BLAS standard implements 3 Levels of linear algebra subroutines. At the lowest level, level-1, it consists of vector-vector operations. At level-2 are the matrix-vector operations and at level-3 the matrix-matrix operations. The higher the level of

[1]Message Passing Interface, a standard for message passing on distributed-memory architectures

[2]Basic Linear Algebra Communication Subprograms

[3]Basic Linear Algebra Subroutines

[4]Engineering and Scientific Subroutine Library

the operations, the more efficient they are. This property is exploited in ELPA in the two-stage solver.

4.2.1.2 Eigensolver

ELPA implements two different approaches for the solution of the eigenproblem. They both are replacements for the ScaLAPACK call *pdsyevd* which internally calls *pdsytrd*, *pdstedc* and *pdormtr* for tridiagonalization, solution of the tridiagonal eigenproblem and back transformation respectively [4]. The two routines introduced by ELPA are *solve_evp_real* and *solve_evp_real_2stage*. Both implement a specialized version of Cuppen's Divide and Conquer method. While the *solve_evp_real* routine implements a direct tridiagonalization, the *solve_evp_real_2stage* routine utilizises a two-stage approach introduced by Bischof et al. [6]. In the two-stage approach the matrix is first reduced to banded form with a certain bandwidth b using Householder transformations and then in a second step the banded form is transformed to tridiagonal form using specialized algorithms [4]. The advantage of that approach is that the vast majority of operations can be implemented using level-3 BLAS which is highly efficient. While the costs of the banded-to-tridiagonal reduction step are relatively cheap ($6bn^2$ FLOPS), the back transformation step is more expensive and takes $2kn^2$ FLOPS more than the one-stage solver. As a consequence, the tridiagonalization may be faster than in the one-stage approach, but the back transformation of the eigenvectors from those of the tridiagonal matrix to those of the banded matrix takes so much additional costs that for a large amount of eigenvectors to be computed the two-stage solver will supersede the one-stage solver in runtime. The two-stage solver is therefore especially promising if only a smaller subset of $k < n$ eigenvectors are of interest.

4.2.1.3 Data distribution

As ELPA is developed as a drop-in replacement for ScaLAPACK, it has certain requirements towards the distribution of data. The data has to be distributed before any processor issues a call to one of the eigensolver routines. In distributed-memory systems, every compute node has its own memory that can not be accessed by other compute nodes. There exist several libraries that aim to ease the programming of distributed-memory architectures; especialliy concerning the aspects of data distribution and transfer between different nodes. ScaLAPACK builds up on BLACS which internally make calls to MPI routines. In a MPI setting, the arrangement of processors is considered to be an one-dimensional array with processor indexes ranging from $0 \ldots (P-1)$, where P is the number of processors involved.

P_0	$\dots$	P_{N-1}

Abbildung 4.3: view on the processor arrangement in MPI

As linear algebra deals with matrices which are known to be arranged in a two-dimensional fashion, a two-dimensional arrangement of processors seems very appropiate for these purposes. In ScaLAPACK and ELPA settings respectively, this is done using BLACS as the communcation library. The two-dimensional arrangement of processors, called the *processor grid*, can be seen in figure 4.4.

A matrix $A \in \mathbb{R}^{n \times n}$ is distributed over the processor grid in a two-dimensional block cyclic fashion. Therefore it is divided into blocks of size $nb_{row} \times nb_{col}$ which are then cyclicly distributed over the processors in the grid. Two blocking factors can be specified at runtime by the user: nb_{row}[5] and nb_{col}[6].

$P_{0,0}$	$\dots$	$P_{0,NC-1}$
$\vdots$	$\ddots$	$\vdots$
$P_{NR-1,0}$	$\dots$	$P_{NR-1,NC-1}$

Abbildung 4.4: view on the two-dimensional processor arrangement as it is used in BLACS

As ScaLAPACK mostly deals with matrices of quadratic dimension, a distinction between nb_{row} and nb_{col} is rendered unnecessary. For the eigensolvers implemented in ScaLAPACK it it necessary that[8]

$$nb_{row} = nb_{col} \tag{4.2.1}$$

The optimal value of this blocking factor can vary between different architectures. As there is no distinction between these factors, the *distribution block size* will be called nb_{dist} in the following. In computations on the local array (see figure 4.5) another blocking factor comes to use. This factor is commonly named the *algorithmic block size* and will further be denoted nb_{algo}. It determines how many matrix elements are processed at once in a local algorithmic computation, e.g. a BLAS call. The optimal value for this factor is strongly constrained by the system's cache size as the cache represents the fastest memory and smallest memory that resides in a system and is the very bottleneck to the speed at which the data can be processed in local computations that

[5]rows per block blocking-factor
[6]columns per block blocking-factor

don't rely on interprocess communication. If the factor nb_{algo} is chosen optimal, the amount of load operations to the cache is reduced to a minimum and the computation is most efficient concerning only the local node.

In ScaLAPACK the *algorithmic block size* is dependent on the *distribution*

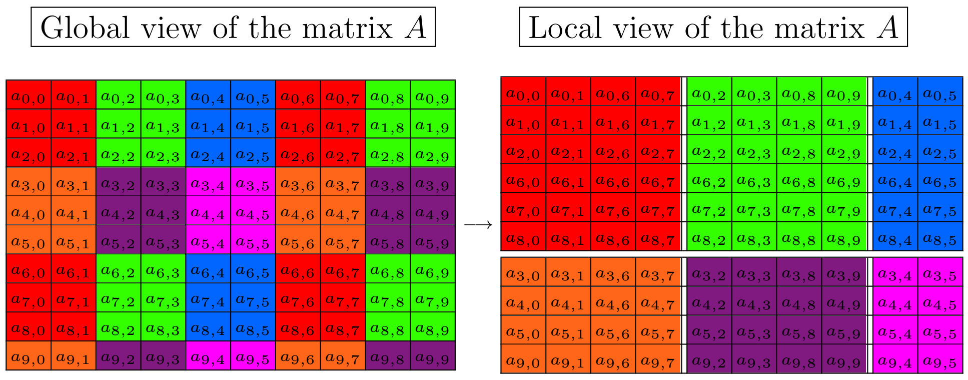


Abbildung 4.5: view on the two-dimensional block cyclic data distribution of a matrix $A \in \mathbb{R}^{10\times 10}$ as it is implemented in ScaLAPACK. The processor grid is of size 2×3, the blocking factors $nb_{col} = 2$ and $nb_{row} = 3$.

block size nb_{dist}:

$$nb_{dist} = nb_{algo} \tag{4.2.2}$$

The chosen blocksize determines the size of the blocks that are processed in evel-3 BLAS calls. As the physical and algorithmic blocksize is equal in ScaLAPACK, the choice also determines how much communication is needed between the different nodes. A bigger blocksize may lead to load imbalance, as can be seen in figure 4.5 while a smaller blocksize decreases the BLAS efficiency and increases the expenses on communication.

4.2.2 Elemental

Elemental is a framework for distributed-memory linear algebra that provides several routines for the parallel solution of numerical problems. It is designed as a C++ library that makes extensive use of the C++ template feature. The library is still undergoing development and continually improving. Throughout this work different versions of Elemental were tested while the performance tests and measurements were done with Elemental version 0.75.

4.2.2.1 Structure

Elemental makes heavy use of object-orientation as this is the main concept of the C++ language. There are several different classes that describe matrices (or vectors as $n \times 1$ matrices), e.g. classes that represent local matrices or distributed matrices. Another key concept of the C++ language are namespaces, all classes and routines are organized in such way. The namespaces in Elemental 0.74 were labeled clearly according to their purpose while future versions of the library will use a common namespace *elem* to ease overviewing and using the many different routines and classes. The object orientated design greatly simplifies the programming of distributed-memory architectures as many details of the data distribution and logical arrangement of processors are implicitly initialized with default values of Elemental if not explicitly stated otherwise. To illustrate the simplicity of programming with Elemental, the initialization and distribution of a distributed matrix will be further illuminated: An instatiation of the class DistMatrix, e.g. **DistMatrix <double, MC, MR>(** *n*,*m*, **grid)** constructs a distributed matrix $A \in \mathbb{R}^{n \times m}$ with a column-major cyclic distribution of the matrix elements as illustrated in figure 4.5, but with a fixed, not adjustable blocksize of 1. This means that the individual matrix elements are cyclicly distributed over the two-dimensional process arrangement **grid**. The kind of distribution over the process grid is specified via [**MC, MR**]. While [**MC, MR**] is the far most common distribution used in Elemental, there are several others that could be used. As the eigenvalue solvers that are of interest to this work expect the data to be distributed using the [**MC, MR**] distribution, none of the others will be discussed in detail.

4.2.2.2 Eigensolver

The library provides several routines to the solution of the Hermitian eigenproblem all of which utilize the MRRR algorithm as introduced in 3.3.2. The different routines that come to use during the solution of the eigenproblem will be described in the following:

internal routines:

- **HermitianTridiag** - Reduction from real symmetric or complex Hermitian to tridiagonal form
- **PMRRR** - Calculation of the solution to the tridiagonalized eigenproblem $T\Lambda = \Lambda Q$ utilizing the MR^3-algorithm (3.3.2). The PMRRR library[2] was developed by Matthias Petschow and Paolo Bientinesi from the RWTH Aachen as a hybrid distributed-memory implementation for the MR^3-algorithm. Elemental uses it as an external library that is linked statically.

interface routines to the solution of the eigenproblem

- **HermitianEig** - Computes the full set of eigenvalues and optionally eigenvectors. Elemental exports a few different *overloaded* function prototypes to this routine. Depending on which function prototype is called, Elemental decides whether to calculate all eigenvalues, all eigenvalues and eigenvectors, only a subset of the eigenvalues or a subset of eigenvalues and eigenvectors. Those function prototypes that were used in the eigensolver benchmarks of this work are explained in the following:

– **HermitianEig (UpperOrLower uplo, DistMatrix<double>& A, DistMatrix<double,VR,STAR>& w)** - calculate the full set of eigenvalues w to the real, symmetric matrix $A \in \mathbb{R}^{n\times n}$.

– **HermitianEig (UpperOrLower uplo, DistMatrix<double>& A, DistMatrix<double, VR, STAR>& w, DistMatrix<double>&Z)** - calculate the full set of eigenvalues w and eigenvectors Z to the real, symmetric matrix $A \in \mathbb{R}^{n\times n}$.

– **HermitianEig (UpperOrLower uplo, DistMatrix<double>& A, DistMatrix<double, VR, STAR>& w, DistMatrix<double>&Z, int a, int b)** - calculate the set of eigenvalues w and eigenvectors Z with indexes in the range $a, a+1, \ldots, b$ to the real, symmetric matrix $A \in \mathbb{R}^{n\times n}$.

4.2.2.3 Data distribution

In contrast to ScaLAPACK, where distribution and algorithmic blocksize are linked, Elemental is designed as a modern extension of the communication insights of PLAPACK. [22] The simplification Elemental uses is that the distribution block size is fixed at one. The approach is not new and is justified in [20]:

> “In principle, the concepts of storage blocking and algorithmic blocking are completely independent. But as a practical matter, a code that completely decoupled them would be painfully complex. Our code and ScaLAPACK avoid this complexity in different ways. Both codes allow any algorithmic blocking, but ScaLAPACK requires that the storage blocking factor be equal to the algorithmic blocking factor. We instead restrict storage blocking to be equal to 1.”

An argument towards a distribution block size of 1 is the obvious benefit to load balancing. Another notable quote from [20] states this:

> “Block storage is not necessary for block algorithms and level-3 [BLAS] performance. Indeed, the use of block storage leads to a significant load imbalance when the block size is large. This is not a concern on the Paragon, but may be problematic for machines requiring larger block sizes for optimal BLAS performance.”

With the fixed minimal distribution size as Elemental's basic approach, the data is then cyclicly distributed over a specified two-dimensional processor grid. The same distribution can be achieved in ScaLAPACK if $nb_{dist} = 1$.

Because in ScaLAPACK the distribution block size is always linked to the algorithmic blocksize with the constraint $nb_{dist} = nb_{algo}$ this would lead to a significant inefficiency when using the BLAS. In Elemental on the contrary, the distribution blocksize is always 1 and the algorithmic blocksize is fully independent of that value:

$$1 = nb_{dist} \neq nb_{algo} \tag{4.2.3}$$

An example for an Elemental distribution can be seen in figure 4.6.

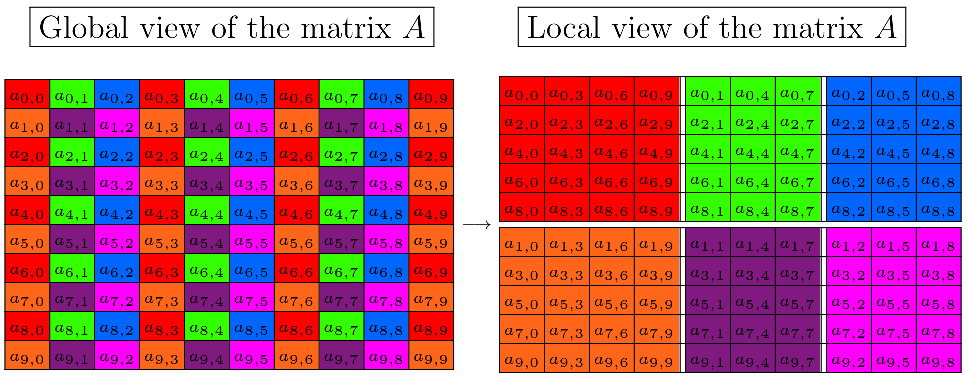

Global view of the matrix A
Local view of the matrix A
$a_{i,j}$ entries $a_{0,0}$ … $a_{9,9}$

Abbildung 4.6: view on the two-dimensional cyclic data distribution of a matrix $A \in \mathbb{R}^{10\times10}$ as it is implemented in Elemental. The processor grid is of size 2×3.

An elemental distribution as implemented in the Elemental library results in better load-balancing over the processor grid as each processor receives a nearly equal sized local matrix compared to a block cyclic distribution. Because the loadbalancing in concerns of data is better in this case, the algorithmic blocksize may be chosen bigger as it is usual in ScaLAPACK. In Elemental, the algorithmic blocksize can be changed, via a call to the *SetBlocksize* routine. It can be changed at any time to the current requirements of the application.

As mentioned in 4.2.2.1, the kind of data distribution is specified via template parameters to the DistMatrix class. It requires three template parameters that determine the data type and the kind of distribution. The constructor of the class itself is overloaded and takes up to 8 parameters. The constructor prototype that was used during this work only takes 3: the horizontal and vertical dimensions of the matrix to be constructed and the processor grid over which it is to be distributed. If the processor grid is omitted, it defaults to the return value of *DefaultGrid* routine. The most commonly used distribution is called the [MC,MR] distribution. It implements a element-wise cyclic distribution where the assignment of elements is done over the processor grid in a column-major way. To precise this, the [MC,MR] distribution of a 7×7 matrix is illustrated below.

The processor grid that will be used to illustrate the differences between the distributions is of the following two-dimensional arrangement:

$$\begin{pmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix}$$

[MC,MR] distribution:
A matrix $A \in \mathbb{R}^{7\times7}$ distributed over the processor grid using the [MC,MR] distribution. Each element corresponds to the processor number it is owned by.

$$\begin{pmatrix} 0 & 2 & 4 & 0 & 2 & 4 & 0 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 0 & 2 & 4 & 0 & 2 & 4 & 0 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 0 & 2 & 4 & 0 & 2 & 4 & 0 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 0 & 2 & 4 & 0 & 2 & 4 & 0 \end{pmatrix}$$

[MR,MC] distribution:
Another possibility of data distribution over the processor grid would be the [MR,MC] distribution. This distribution assigns the elements over the processor grid in a row-major way. In this case, the matrix would be distributed as if it were transposed:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 \\ 4 & 5 & 4 & 5 & 4 & 5 & 4 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 \\ 4 & 5 & 4 & 5 & 4 & 5 & 4 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

[VR,*] distribution:
When calculating eigenvalues, another type of distribution comes to use in elemental. It is a one-dimensional distribution, commonly used for distributing vectors. In elemental's eigensolver, the vector of eigenvalues that will be calculated is distributed using the [VR,*] distribution. The individual elements of the vector are assigned in a row-major fashion over the processor grid:

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

5 Performance analysis, comparision and optimization

The first section in this chapter introduces the test matrices that were used during the performance benchmarks of the individual eigensolvers. In the following sections, the libraries will be compared and discussed in terms of performance-analysis.

5.1 Test Matrices

Test applications were written for both Elemental and ELPA. Both applications use the same set of test matrices to guarantee comparability of the results. The test matrices are randomly chosen dense symmetric matrices $A \in \mathbb{R}^{n \times n}$ that are constructed with the following scheme: At first a random vector of eigenvalues $d \in \mathbb{R}^n$ is generated. Subquently another random vector $v \in \mathbb{R}^n$ is generated to construct a Householder matrix Q as follows

$$Q = I - \frac{2}{v^T v} v v^T \tag{5.1.1}$$

The matrix Q is an orthogonal matrix with the eigenvectors that will later be determined by the eigensolvers on its column.

Appropiate test matrices then can be calculated using an eigendecomposition approach:

$$A = Q \begin{pmatrix} d_1 & & & & \\ & d_2 & & & \\ & & \ddots & & \\ & & & d_{n-1} & \\ & & & & d_n \end{pmatrix} Q^T \tag{5.1.2}$$

with a square matrix containing only the random eigenvalue elements $d_i, i \in \{1, \ldots, n\}$ on its main diagonal. Both the calculation of eigenvectors and eigenvalues of these test matrices A utilize the same random number generator in Fortran and C++. As the speedup of the ELPA one- and two-stage solver computing different amounts of eigenvalues and eigenvectors was a main concern, there were no tests done in which only the eigenvalues were calculated.

5.2 Correctness of results

To verify the numerical correctness of the results, all benchmarks computed different residual and error norms on the results.

With $T_{orig} \in \mathbb{R}^{n\times n}$ the input matrix, $Z \in \mathbb{R}^{n\times n}$ the calculated eigenvector matrix, $\Lambda_{orig} \in \mathbb{R}^{n\times n}$ the matrix with the input eigenvalues on it's main diagonal and $\Lambda \in \mathbb{R}^{n\times n}$ the matrix with the calculated eigenvalues on it's main diagonal the following residual norms were computed:

$$\|\Lambda_{orig} - \Lambda\|_\infty \tag{5.2.1}$$

$$\|T_{orig}Z - Z\Lambda\|_\infty \tag{5.2.2}$$

$$\|ZZ^T - I\|_\infty \tag{5.2.3}$$

The first residual (5.2.1) only verifies the numerical correctness of the calculated eigenvalues. The second residual (5.2.2) evaluates the spectral decomposition property for the calculated matrix Z and eigenvalue matrix Λ and the third norm (5.2.3) tests the matrix Z with the calculated eigenvectors on its columns for numerical orthogonality.

5.3 Tuning parameters

The choice for an optimal algorithmic blocksize was well known for Elemental to be the library's default value of 128 on the Blue Gene/P and Blue Gene/Q architectures[5]. For ELPA an optimal algorithmic blocksize was determined over benchmarks with matrix sizes $T_1 \in \mathbb{R}^{5000\times 5000}$ and $T_2 \in \mathbb{R}^{10000\times 10000}$ and varying blocksizes out of the interval $\{10, 12, \ldots, 38, 40\}$.

As each invidiual processor has to compute several decompositions on its local parts of the matrix, a blocksize where $\frac{N}{nb_{algo}\max\{np_{row}, np_{col}\}}$ is integer seems most appropiate, as this is the number of decompositions that a processor which is working on the diagonal of the distribution grid has to compute. The surrounding processors have to update their local parts respectively often. If $np_{row} \times np_{col} = 8 \times 4$ is the processor grid and the test matrix has dimension

5000×5000, then the blocksize that would meet this requirements approximately is

$$nb_{algo} \mid \left\lfloor \frac{N}{\max\{np_{row}, np_{col}\}} \right\rfloor = \left\lfloor \frac{5000}{8} \right\rfloor = \lfloor 625 \rfloor = 625$$

For a parallel algorithm there is a trade-off between both load balancing and BLAS efficiency in terms of the algorithmic blocksize. Facing this fact it is rendered impossible to determine the optimal blocksize for ELPA theoretically. The optimal values that were determined are therefore purely empirically and may vary for a different system configuration.

Because the minimum node amount for a JUQUEEN batch system call is fixed to 32, the blocksize benchmark results on JUQUEEN and JUGENE are not comparable to this case, but show a tendency for each of both machines. While a JUGENE job ran with 32 MPI processes (1 per node), a JUQUEEN job enfolded 512 MPI processes (16 per node). The calculation for each blocksize was repeated three times. The diagrams in the following sections show the arithmetical mean of all three timings.

The benchmarks in section 5.3.1 on JUGENE and section 5.3.2 on JUQUEEN lead to the parameter choices for the final benchmarks. All runtime benchmarks that will be discussed in the following were done accordingly.

5.3.1 ELPA - JUGENE

As mentioned in 4.2.1, in ScaLAPACK and ELPA the algorithmic blocksize nb_{algo} has to be the same as the distribution blocksize nb_{dist} and has a significant effect on both local performance and data distribution effiency. In the beginning of this section, the influence of the algorithmic blocksize on the runtime of certain components of ELPA will be further delighted. As figure 5.1 reveals, the overall optimal blocksize on JUGENE is either 18 or 12 - in the benchmarks 18 was used. Larger blocksizes seem to slow down the performance.

While the tridiagonalization and eigensolver step of the ELPA one-stage solver seems relatively unaffected by changes to the blocksize, the influence on the runtime of the back transformation step is of more fluctuating nature. There are strong local minima to the runtimes at both values that were considered to be optimal for the Blue Gene/P system JUGENE. With increased matrix size and therefore increased runtimes, as can be seen in the right half of figure 5.2, the effect of the blocksize on the runtimes of tridiagonalization, solution

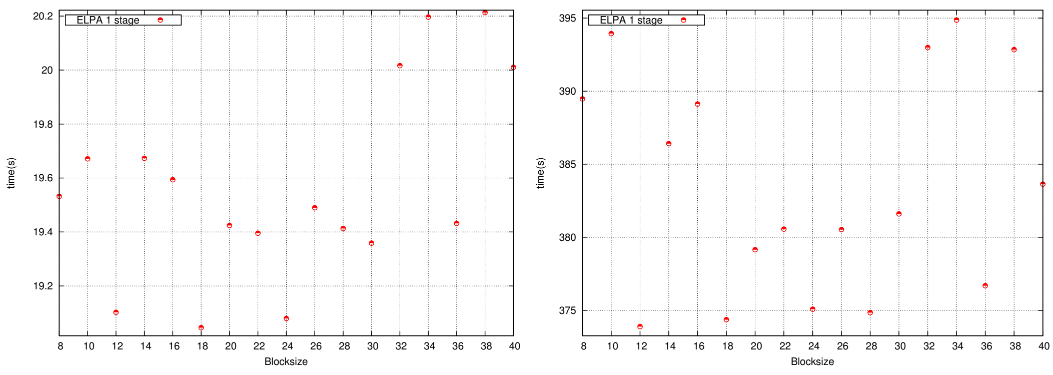


Abbildung 5.1: Benchmark of the ELPA one-stage solver on JUGENE using a processor grid of 8×4 MPI processes with a fixed matrix size of $T_1 \in \mathbb{R}^{5000\times5000}$ on the left and $T_2 \in \mathbb{R}^{15000\times15000}$ on the right. The blocksizes are plotted on the x-axis and range from 8 to 40.

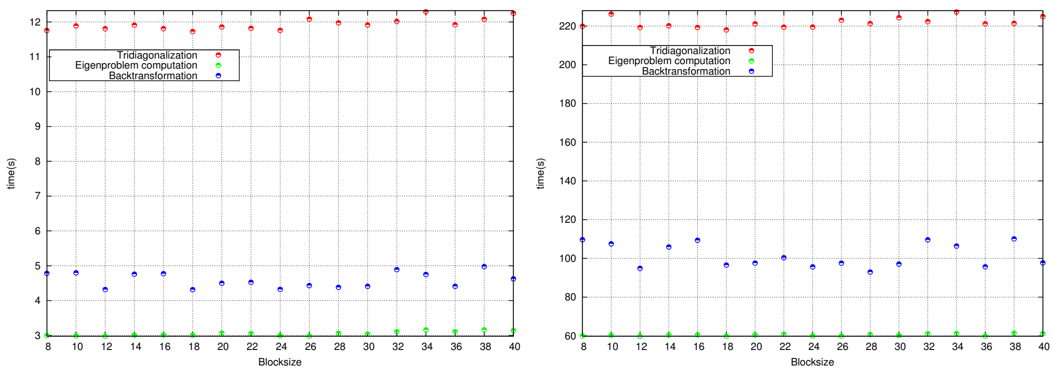


Abbildung 5.2: The individual components of the ELPA one-stage solver on JUGENE using a processor grid of 8×4 MPI processes with a fixed matrix size of $T_1 \in \mathbb{R}^{5000\times5000}$ on the left and $T_2 \in \mathbb{R}^{15000\times15000}$ on the right. The blocksizes are plotted on the x-axis and range from 8 to 40.

and back transformation is apparantly intensifying.

5.3.2 ELPA - JUQUEEN

Results on JUQUEEN show a similar behaviour of runtimes compared to the JUGENE results. The effect of the blocksize on the runtime, depicted in figure 5.4, seems limited but with local deviations at 16, 24 and 32. Similar to the benchmarks on JUGENE, the effect of the nb_{algo} parameter seems to increase with larger matrix sizes. The right half of figure 5.4 shows benchmarks with a test matrix size of 15000.

The Blue Gene/Q architecture greatly exceeds its predecessor in both memo-

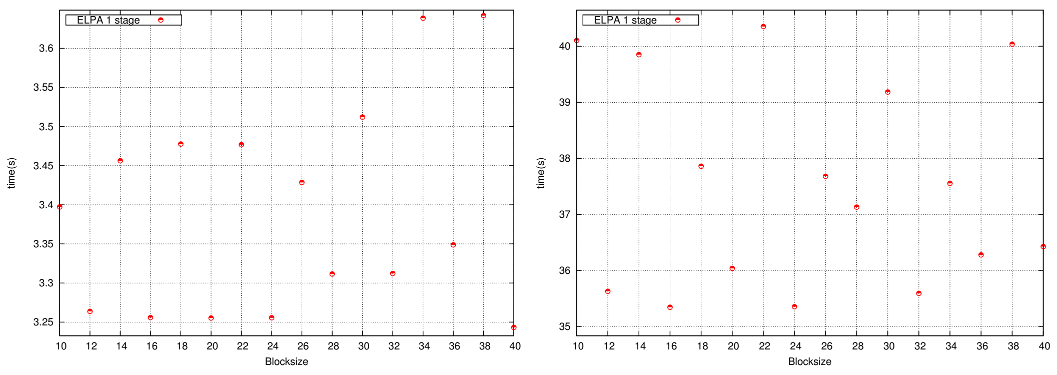


Abbildung 5.3: Benchmark of the ELPA one-stage solver on JUQUEEN using a processor grid of 32 × 16 MPI processes with a fixed matrix size of $T_1 \in \mathbb{R}^{5000\times5000}$ on the left and $T_2 \in \mathbb{R}^{15000\times15000}$ on the right. The blocksizes are plotted on the x-axis and range from 10 to 40.

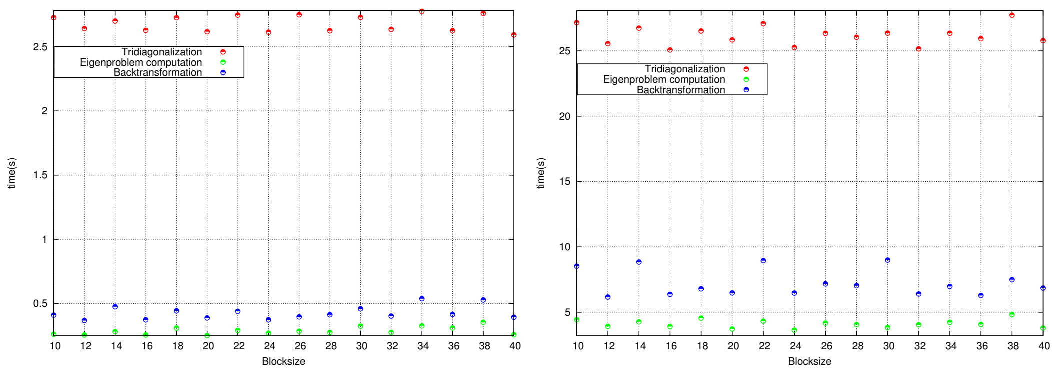


Abbildung 5.4: The individual components of the ELPA one-stage solver on JUQUEEN using a processor grid of 32 × 16 MPI processes with a fixed matrix size of $T_1 \in \mathbb{R}^{5000\times5000}$ on the left and $T_2 \in \mathbb{R}^{15000\times15000}$ on the right. The blocksizes are plotted on the x-axis and range from 10 to 40.

ry bandwith and clockrate. A different result seems not unexpected. On the contrary, previously to the blocksize benchmarks, based on private communications an optimal value of 16 was expected for the Blue Gene/Q system. As the results greatly meet these expectations, a value for nb_{algo} of 16 was consequently used for the ELPA benchmarks on JUQUEEN.

For the benefit of the overall efficiency, small blocksizes of about 10-40 prove to be a good choice for ELPA to achieve a compromise between load balance and communication minimization.

5.4 Runtime Analysis

5.4.1 JUGENE - ELPA one-stage, two-stage and Elemental

The benchmarks on JUGENE were done for three different node amounts that will be further specified in the following tables:

Partial eigenspectrum benchmarks

- 256 nodes, 1 MPI process per node, logical processor grid 16×16
- 1024 nodes, 1 MPI process per node, logical processor grid 32×32
- 4096 nodes, 1 MPI process per node, logical processor grid 64×64

Each of these benchmarks incremented its test matrix sizes from $6000 \ldots 50000$ with step sizes of 4000. The percentage of eigenvalues that was calculated in each of these benchmarks ranged from $5\% - 45\%$ with step sizes of 5 to investigate whether the ELPA two-stage solver is really faster than its one-stage equivalent when only some of the eigenvectors are needed.

Benchmarks in which a full eigenspectrum was calculated were done on JUGENE, too. Their details in terms of MPI processes and nodes involved, are summarized in the following listing:

Full eigenspectrum benchmarks

- 1024 nodes, 1 MPI process per node, logical processor grid 32×32
- 4096 nodes, 1 MPI process per node, logical processor grid 64×64
- 16384 nodes, 1 MPI process per node, logical processor grid 128×128

Each figure in this section depicts the runtimes of the ELPA one-stage, two-stage and the PMRRR eigensolver in Elemental at the same matrix and processor settings.

Discussion of partial Eigenspectrum benchmarks
As figure 5.5 depicts, at a low level of eigenvalues and vectors to be calculated, as 5% in the underlying benchmark, the ELPA 2 outspeeds both Elemental and the ELPA one-stage solver in its runtime consequently over different matrix sizes. The runtimes of Elemental and the ELPA one-stage solver in some cases nearly match, but those of Elemental are more erratic and in most cases exceed the ELPA one-stage solver.

As previously mentioned in 4.2.1 the increasing influence of the amount of

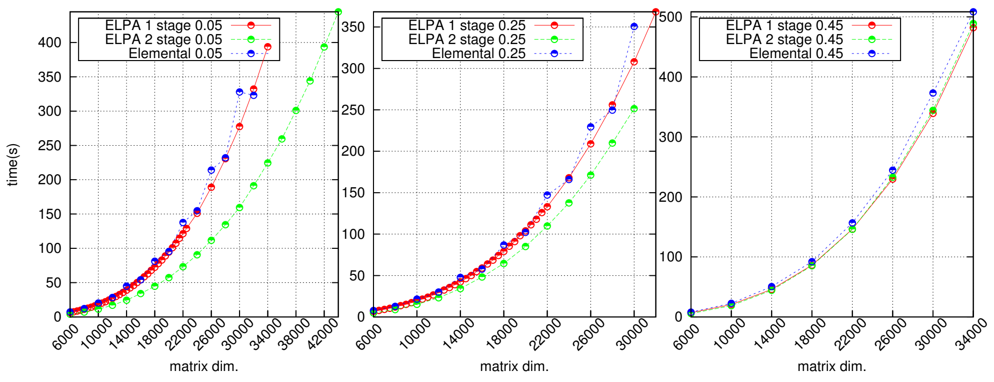


Abbildung 5.5: Benchmark of ELPA one-stage, two-stage and Elemental, 5%, 25% and 45% of eigenvalues and vectors computed, 256 MPI processes

eigenvalues and vectors to be computed on the runtime of the back transformation step in the ELPA two-stage solver can be seen in the second and third from left figures of 5.5. With an increasing set of eigenvalues and vectors to

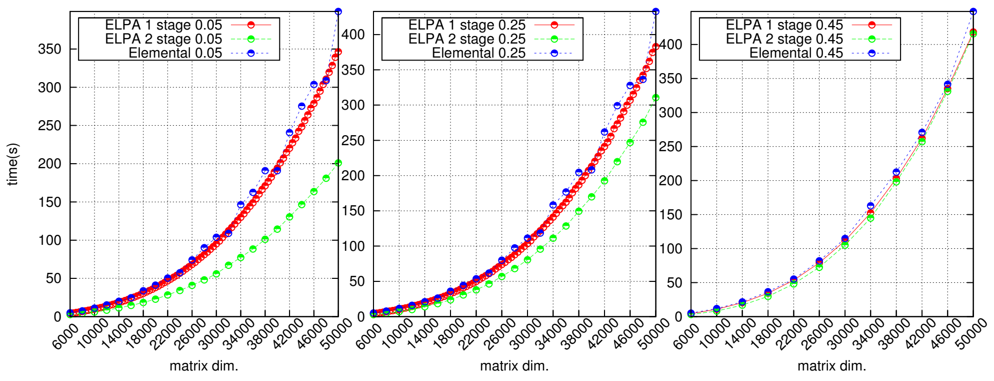


Abbildung 5.6: Benchmark of ELPA one-stage, two-stage and Elemental, 5%, 25% and 45% of eigenvalues and vectors computed, 1024 MPI processes

compute, 25% in the underlying figures 5.5, 5.6 and 5.7, the runtimes of all three solvers increase, too, but while Elemental and the ELPA one-stage solver strongly match the profile of the 5% situation, the ELPA two-stage solver takes significantly longer.

The increasing number of processors involved apparently has an effect on the relative gap between the ELPA 1 and two-stage solver. With a larger number of nodes participating in the calculation, the ELPA two-stage solver seems to become more and more efficient as can be seen in figure 5.7. The runtime profiles of the ELPA one-stage and ELPA two-stage solver do not cross anymore as it was the case in the benchmarks with 256 and 1024 nodes involved.
If the amount of processors involved is increased, the runtimes of all three solvers expectedly decrease. The impact of the back tranformation step in the

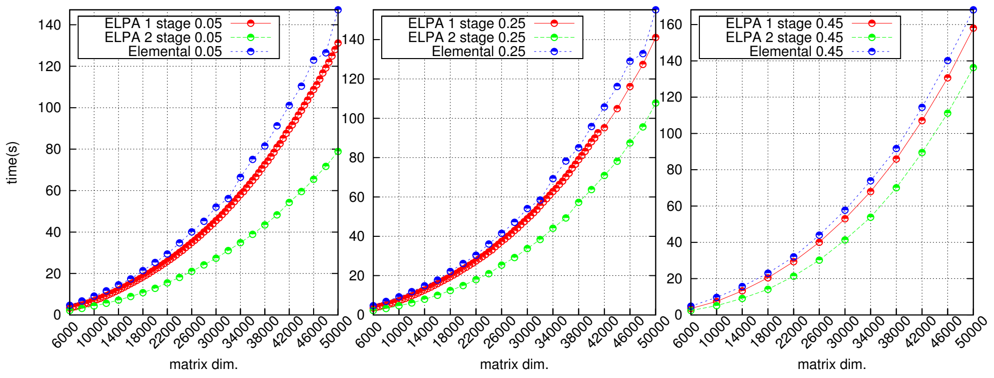


Abbildung 5.7: Benchmark of ELPA one-stage, two-stage and Elemental, 5%, 25% and 45% of eigenvalues and vectors computed, 4096 MPI processes

ELPA two-stage solver however, still seems distinctive with an increasing number of eigenvalues to be computed as can be seen in the comparision of figures 5.6 and 5.7.

In the presented benchmarks with 256, 1024 and 4096 MPI processes on 256, 1024 and 4096 nodes of JUGENE, the crossline of the ELPA one- and two-stage solver can be seen in the right third of the figures. On JUGENE they intersect at 45% of eigenvalues and vectors to compute. At this percentage the expensive back transformation of eigenvectors from tridiagonal to banded form, as mentioned in 4.2.1, of the ELPA two-stage solver strikes and the one-stage approach seems more attractive if a larger spectrum of eigenvalues and vectors is needed.

Discussion of full eigenspectrum benchmarks

In the full eigenspectrum benchmarks, the processor count of 256 was omitted because the maximum runtime for a JUGENE batch system call of this size was 30 minutes and none of the jobs were able to complete their tasks in this time slice. The benchmarks were done with processor counts of 1024, 4096 and 16384. Their runtime profiles are depicted below. As the intersection of the runtime profiles of the ELPA two-stage and one-stage solvers has been determined to be at about 45% of eigenpairs to compute, benchmarks for the ELPA two-stage solver were not done when a full eigenspectrum was wanted. As figure 5.8 clearly depicts, the runtime profiles of Elemental and ELPA diverge with an increasing number of processors involved. While the test matrix dimensions in the above figures steadily increase, the runtimes slowly diverge with a overall tendency of the ELPA one-stage solver to be faster than Elemental. The case of 16384 MPI processes, in the rightmost third of figure 5.8 visualizes this difference in runtime very obviously.

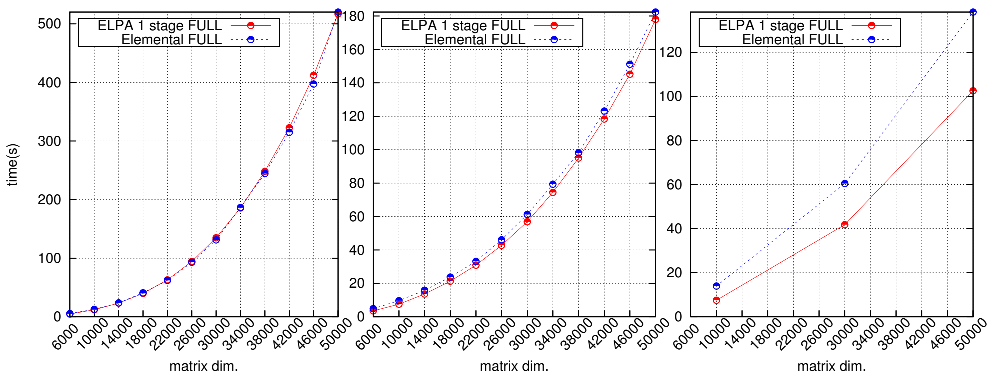


Abbildung 5.8: Benchmark of ELPA one-stage and Elemental, full eigenspectrum computed, 1024, 4096 and 16384 MPI processes

5.4.2 JUQUEEN - ELPA one-stage, two-stage and Elemental

Benchmarks on JUQUEEN were done for the constant node count of 32 but with a differing partial spectrum of eigenvalues and vectors to compute. In these benchmarks each node was executing 16 MPI processes in parallel. The partial spectrum benchmarks that were done are denoted in the following table. The calculations included the ELPA one-stage, two-stage and the Elemental eigensolver.

Partial eigenspectrum benchmarks

- 32 nodes, 16 MPI processes per node, logical processor grid 32×16, 5%-45% of eigenvalues and vectors computed with steps of 10%

Of the partial eigenspectrum benchmarks, only the cases 5%, 25% and 45% are presented in this work as they are representative enough to visualize the impact of the back transformation step of the ELPA two-stage solver.

Full spectrum benchmarks were done on JUQUEEN, too, with 32 nodes but varying amount of MPI proccesses per node. These benchmarks only included the ELPA one-stage and the Elemental eigensolver. The ELPA two-stage solver disqualified for these benchmarks because of too high runtimes when calculating the full spectrum of eigenvalues and vectors.

Full eigenspectrum benchmarks

- 32 nodes, 16 MPI processes per node, logical processor grid 32×16
- 32 nodes, 32 MPI processes per node, logical processor grid 32×32
- 32 nodes, 64 MPI processes per node, logical processor grid 64×32

Discussion of partial eigenspectrum benchmarks

As can be seen in figure 5.9, the Elemental eigensolver has a less erratic profile than it was the case on JUGENE. The profile is much smoother and closely fits the one of the ELPA one-stage solver. While the ELPA two-stage solver is clearly faster at the level of 5% of eigenvalues and vectors to compute, the impact of an increasing eigenspectrum to calculate seems to be more intensive than on JUQUEEN's predecessor. This issue is addressed by ELPA developers and already fixed in a newer version of the library. It can be reviewed more clearly in the runtime profiles depicted in the second and third subfigure of figure 5.9.

With 25% of eigenvalues to compute, as figure 5.9 depicts, all three runtime

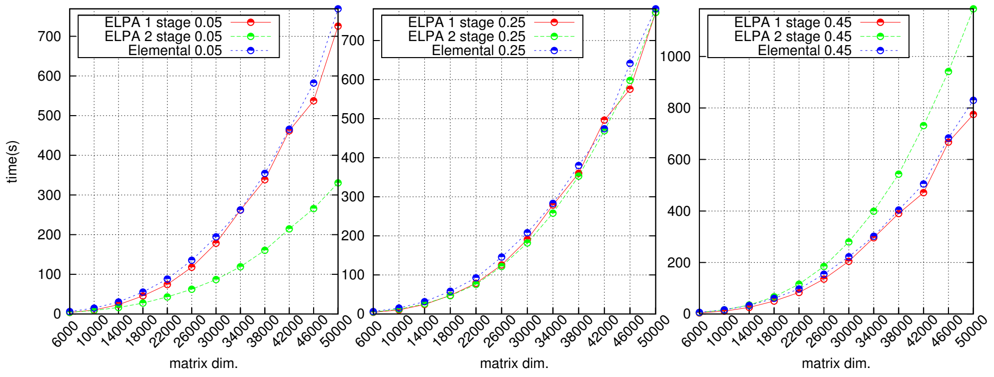


Abbildung 5.9: Benchmark of ELPA one-stage, two-stage and Elemental, 5%, 25% and 45% of eigenvalues and vectors computed, 512 MPI processes

profiles closely match. For a benchmark with 32 nodes and 16 MPI proccesses per node, this seems to be the practical limit where the ELPA two-stage solver outspeeds both the ELPA one-stage and Elementals eigensolver routines on the JUQUEEN system. In the 25% case in figure 5.9, the runtime profile of the ELPA two-stage eigensolver does not drastically deviate from the other two competitors. With 45% of the eigenspectrum to compute, the runtimes of Elemental and ELPA one-stage do not increase significantly, while the ELPA two-stage sets apart far above the other two runtime profiles and therefore currently does not qualify as an option for situations where more than approximately 25% of the eigenspectrum is needed.

Discussion of full eigenspectrum benchmarks

The full eigenspectrum benchmarks on JUQUEEN were done for MPI process counts of 512, 1024 and 2048 respectively. Each of these benchmarks were done on 32 nodes of the JUQUEEN system, which means that they executed 16, 32 and 64 threads per node in parallel. Their runtime profiles are depicted in figure 5.10.

As mentioned before, the ELPA two-stage solver can not compete with the

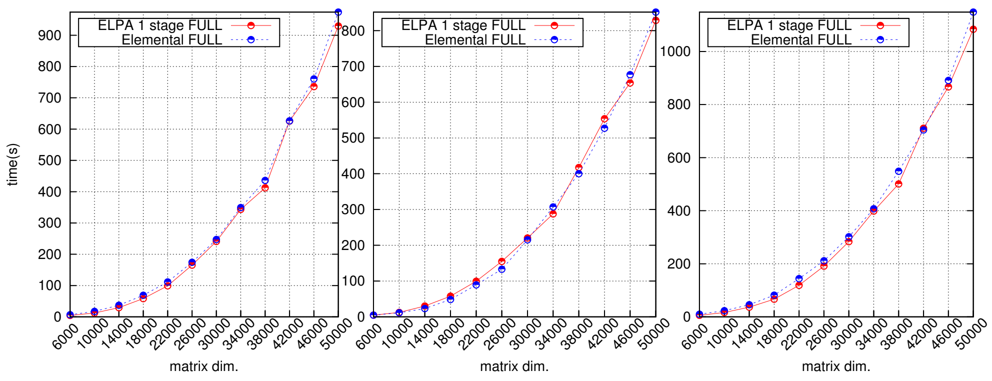


Abbildung 5.10: Benchmark of ELPA one-stage and Elemental full eigenspectrum computation, 512, 1024 and 2048 MPI processes

other two solvers when a full eigenspectrum is to be computed. Its runtimes drastically diverge from the other two solvers with an increasing test matrix dimension as it can be reviewed in the partial eigenspectrum computations depicted in figure 5.9. In computations where the testmatrix dimension was above 22000, it required more than double the time that ELPA one-stage and Elemental took to complete when a full eigenspectrum was to be computed. If 32 threads are executed in parallel on each JUQUEEN node, the runtimes of all three solvers expectedly drop as depicted in figure 5.10 for the Elemental and ELPA one-stage solver. This can be justified with the properties of the Blue Gene/Q architecture as they were described in section 4.1.2. As each of the 16 processors on a Blue Gene/Q node can execute 2 threads in parallel (and 4 with hyperthreading, see section 4.1.2), the machine's peak performance is practically maxed out if 1024 MPI processes are executed in parallel in a benchmark with 32 Blue Gene/Q nodes involved.

Though the effect on ELPA one-stage and Elemental seems minimal, the ELPA two-stage runtime drew significantly nearer to the profiles of the other two solvers. While the results of the ELPA two-stage solver are not depicted, the back transformation of tridiagonal to banded eigenvectors obviously appears to execute more effectively with an increasing number of threads involved as the benchmark results indicate. The minimal benefit in runtime that ELPA one-stage and Elemental experiences seems unexpected as the machine's theoretical peak performance should be reached when 2 threads are executed in parallel on each core as mentioned before in this section.

5.4.3 Runtime analysis conclusions

In all benchmarks that were described in the previous two sections, three different solvers were reviewed for their performance over different matrix sizes which usually varied from 6000 to 50000. The first two solvers, namely ELPA one-stage and ELPA two-stage implement an optimized version of Cuppen's divide-and-conquer algorithm which is introduced in chapter 3.3.1. The third eigensolver that is part of the Elemental library and which implements the MR^3 algorithm is described in chapter 3.3.2.

Each of these eigensolvers first implement a matrix transformation from dense to tridiagonal form. While ELPA one-stage and Elemental utilize Householder transformations to directly reduce the problem's system matrix A to its tridiagonal counterpart T, ELPA two-stage implements a new approach that was introduced by Bischof et al.[6]. The reduction in this case is done in two steps, first from full to banded and then from banded to tridiagonal form. The advantages of this approach are described in section 4.2.1.

These basically different approaches to the reduction of the full eigenproblem additionally were benchmarked for different percentages of the eigenspectrum ranging from 5% to 45%. In each scenario both eigenvalues and eigenvectors were computed. Each of the three solvers have to accomplish a more or less computationally expensive back transformation of eigenvectors, depending on how much of the eigenspectrum is needed for a specific scenario. The benefit that ELPA two-stage experiences in the tridiagonalization process later has an impact on the back transformation step - but this effect can be minimized if only a small amount of the spectrum is required. The intersection of the runtime profiles of the other two solvers and this two-stage approach was determined to lie at about 45% on the JUGENE Blue Gene/P system and at about 25% on the JUQUEEN Blue Gene/Q system at the Forschungszentrum Jülich. The poor performance of the two-stage approach on the newer Blue Gene/Q system is now fixed by an optimized version of the first step of the back transformation phase.

Concluding from these results, both the divide-and-conquer algorithm implemented in the ELPA solvers and the MR^3 algorithm utilized by Elemental are adequate solvers for the solution of large scale eigenproblems on distributed memory architectures and are closely competing in runtime. Both algorithms do not significantly differ in their runtime on the Blue Gene/P and Blue Gene/Q architectures, over differing matrix sizes and percentages of the eigenspectrum to be compute. The two-stage eigensolver implemented in ELPA is a promising candidate on the Blue Gene architectures if only a small amount of eigenvalues and vectors is required.

6 Scalasca instrumentation

6.1 Scalasca

This section presents the Scalasca[16] instrumentations of the test applications that were used throughout the benchmarks. Scalasca is an instrumentation toolset designed for the analysis of parallel applications that use MPI, OpenMP or hybrids of these. It can be used to instrument applications that were written in C, C++ and Fortran and is specifically intended for HPC architectures such as the family of Blue Gene supercomputers.

The Scalasca project is developed actively and has its origins at the Forschungszentrum Jülich. Therefore, it is compatible with the most recent supercomputers in the facility such as JUGENE, JUQUEEN and several others.

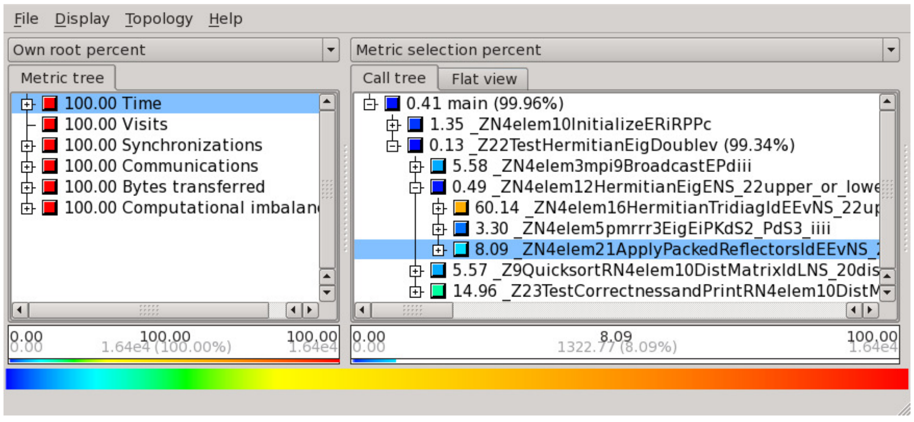
Before any instrumentation measurements can be done, the application needs to be compiled with a special Scalasca preposition command:
scalasca -instrument <compiler><sourcecode>. The analysis of the instrumented executable then can be started using the Scalasca command **scalasca -analysis** followed by the usual MPI initialization command such as *mpirun* on JUGENE or *runjob* on JUQUEEN. During the analysis a subfolder will be created in which the results of the instrumentation run are subsequently stored.

To examine the results of a complete instrumentation run, the command **scalasca -examine <subfolder>** postprocesses the raw instrumentation results and then displays them to the user. Depending on which level the Scalasca instrumentation happended at compile time, either only the MPI calls are recorded by the instrumentation or even function calls at the lowest level. The test applications in this thesis were fully instrumented so the individual steps of the eigensolvers, namely tridiagonalization, solving and back transformation can be reviewed in the examination.

The instrumentations of the test applications that will be introduced in the following were done for a single test matrix of size 10000×10000 each. All instrumentation runs were done on JUQUEEN.

6.2 Elemental instrumentation

The instrumentation of the Elemental test application is depicted in figure 6.1. Routines with a runtime percentage below 1% are filtered out, so that only the initialization and actual eigensolving steps are depicted in the figure.



```
100.00 Time
100.00 Visits
100.00 Synchronizations
100.00 Communications
100.00 Bytes transferred
100.00 Computational imbalan

0.00        100.00              100.00
0.00   1.64e4 (100.00%)         1.64e4
```

```
0.41 main (99.96%)
  1.35 _ZN4elem10InitializeERiRPPc
  0.13 _Z22TestHermitianEigDoublev (99.34%)
    5.58 _ZN4elem3mpi9BroadcastEPdiii
    0.49 _ZN4elem12HermitianEigENS_22upper_or_lowe
      60.14 _ZN4elem16HermitianTridiagIdEEvNS_22up
      3.30 _ZN4elem5pmrrr3EigEiPKdS2_PdS3_iiii
      8.09 _ZN4elem21ApplyPackedReflectorsIdEEvNS_
    5.57 _Z9QuicksortRN4elem10DistMatrixIdLNS_20dis
    14.96 _Z23TestCorrectnessandPrintRN4elem10DistM

0.00             8.09              100.00
0.00      1322.77 (8.09%)          1.64e4
```

Abbildung 6.1: Scalasca instrumentation of the Elemental test application

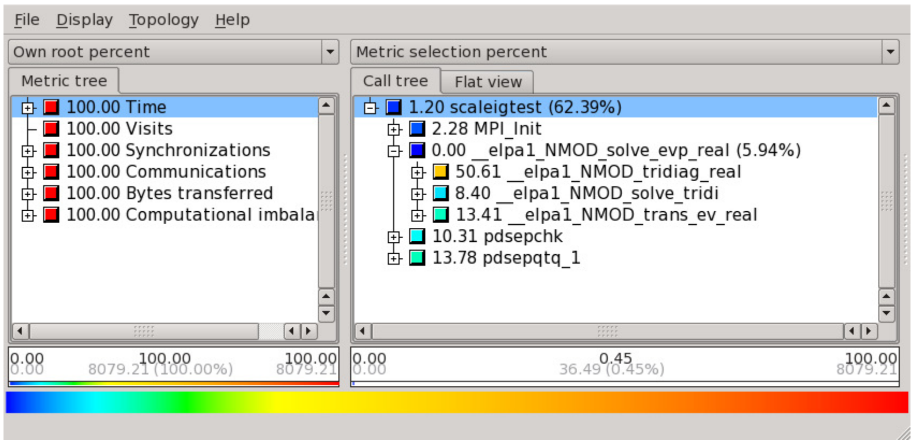
As can be seen in the above figure, the largest parts in runtime of the routine *HermitianEig* consist of the 3 steps: tridiagonalization (*HermitianTridiag* in the figure), solving (*PMRRR* in the figure), and back transformation (*ApplyPackedReflectors* in the figure).

The tridiagonalization takes a great percentage of 60.14%, while the eigensolver routine PMRRR only needs 3.30%. The back transformation is relatively quick compared to ELPA (see figures 6.2 and 6.3) with only 8.09% of the overall runtime. Deriving from these results, there is an obvious potential for optimization in the tridiagonalization routine of Elemental.

Most MPI operations involved were collective (67.74%). Only 32.26% of the MPI operations included point-to-point communication.

6.3 ELPA one-stage instrumentation

In the instrumentation of the ELPA one-stage solver the routine *solve_evp_real* was of primary interest. As with Elemental's eigensolver routine, ELPA includes the process of tridiagonalization, solving of the eigenproblem and back transformation of the eigenvectors. Figure 6.2 depicts the individual percentages of the overall runtime of these operations. The fractions of the overall runtime that these three procedures hold tend to diverge less than it is the case with Elemental (see figure 6.1). While this one-stage tridiagonalization routine (*elpa_NMOD_tridiag_real* in the above figure) claims a major percentage of



```
Metric tree
100.00 Time
100.00 Visits
100.00 Synchronizations
100.00 Communications
100.00 Bytes transferred
100.00 Computational imbala
0.00   100.00   100.00
0.00   8079.21 (100.00%)   8079.21

Call tree
1.20 scaleigtest (62.39%)
  2.28 MPI_Init
  0.00 __elpa1_NMOD_solve_evp_real (5.94%)
    50.61 __elpa1_NMOD_tridiag_real
    8.40 __elpa1_NMOD_solve_tridi
    13.41 __elpa1_NMOD_trans_ev_real
  10.31 pdsepchk
  13.78 pdsepqtq_1
0.00   0.45   100.00
0.00   36.49 (0.45%)   8079.21
```

Abbildung 6.2: Scalasca instrumentation of the ELPA one-stage test application

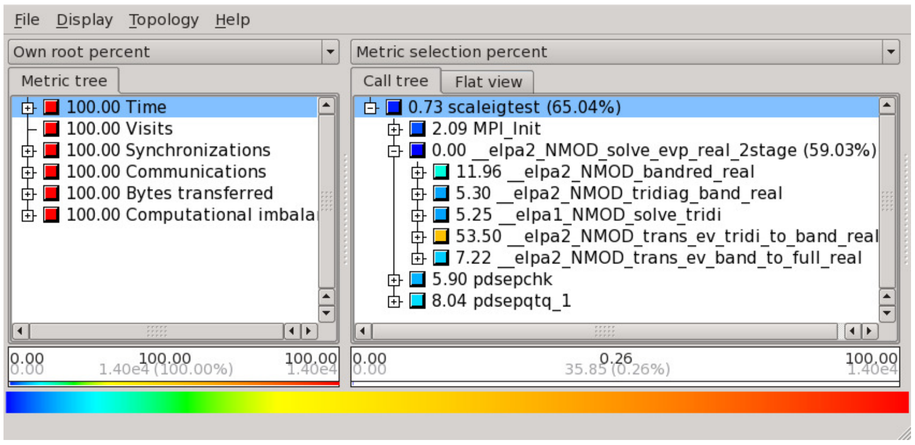
50.61% as expected, its implementation is more effective than the tridiagonalization routine implemented in Elemental. The eigensolver itself acquires a percentage of 8.40% of the overall runtime. As the percentages of the eigensolver and back transformation increase with a more efficient tridiagonalization routine, this does not necessarily mean that it is faster than the MR^3 algorithm implemented in the PMRRR library that is utilized by Elemental. The results of the benchmarks that were presented in chapter 5 however resemble to a general tendency of the PMRRR library to be faster than the divide and conquer algorithm.

The eigenvector back transformation routine holds a share of 13.41% of the overall runtime. Compared to Elemental an even larger percentage of the MPI operations in the process were collective (97.03%) and only a small percentage of 2.97% included point-to-point communications.

6.4 ELPA two-stage instrumentation

While a full eigenspectrum computation with the ELPA two-stage solver is little reasonable as justified in section 5.4.2, an instrumentation run was tasked to visualize the individual routines that are involved in the two-stage tridiagonalization and back transformation process and their shares of the overall runtime.

Figure 6.3 illustrates the instrumentation of the ELPA two-stage test application. As can be seen, the eigensolving process includes several more routines than it was the case with the ELPA one-stage solver (see figure 6.2).
As depicted in the above figure, the tridiagonalization step includes a full to banded and a following banded to tridiagonal reduction. The full to banded reduction (*elpa2_NMOD_bandred_real* in the figure) has a share of 11.96% of



```
Own root percent                         Metric selection percent

100.00 Time                              0.73 scaleigtest (65.04%)
100.00 Visits                              2.09 MPI_Init
100.00 Synchronizations                    0.00 __elpa2_NMOD_solve_evp_real_2stage (59.03%)
100.00 Communications                        11.96 __elpa2_NMOD_bandred_real
100.00 Bytes transferred                     5.30 __elpa2_NMOD_tridiag_band_real
100.00 Computational imbala                  5.25 __elpa1_NMOD_solve_tridi
                                             53.50 __elpa2_NMOD_trans_ev_tridi_to_band_real
                                             7.22 __elpa2_NMOD_trans_ev_band_to_full_real
                                           5.90 pdsepchk
                                           8.04 pdsepqtq_1

0.00        100.00          100.00       0.00            0.26             100.00
0.00   1.40e4 (100.00%)     1.40e4       0.00       35.85 (0.26%)         1.40e4
```

Abbildung 6.3: Scalasca instrumentation of the ELPA two-stage test application

the overall runtime while the banded to tridiagonal reduction (*elpa2_NMOD_tridiag_band_real* in the figure) requires less time with only 5.30%. This means that the reduction phase in the ELPA two-stage solver takes only about 17% of the whole solver's runtime.

ELPA's two-stage approach utilizes the same eigensolver routine as the ELPA one-stage solver, which in the underlying instrumentation run acquires a percentage of 5.25% of the overall runtime.

The performance issues that were mentioned in section 5.4.2 can be reviewed in the figure as the tridiagonal to banded back transformation step holds a share of 53.50% of the overall runtime. This issue is meanwhile fixed in a newer version of the ELPA library. The banded to full back transformation requires only 7.22% as it includes merely the muliplication of the individual Householder matrices to the eigenvectors. In contrast to the ELPA one-stage solver and Elemental, most MPI operations in this instrumentation run included point-to-point communication (75.53%). A relatively small share of 24.47% of all MPI operations were of collective nature.

Because the runtime of the eigensolver routine can be expected to be nearly equal in both the instrumentation of ELPA one-stage and ELPA two-stage, a factor of $\frac{8.4}{5.25} = 1.6$ can be derived from that and applied to the percentage of the reduction phase in ELPA two-stage. If the runtimes of both solvers were equal, the two-stage reduction phase would then only require a percentage of $(11.96\% + 5.30\%) \cdot 1.6 = 27.61\%$ compared to the direct approach with 50.61%.

7 Outlook

The performance analysis in chapter 5 and instrumentations of chapter 6 prove that the ELPA two-stage approach is a promising candidate for eigenproblem computations on large scale distributed memory supercomputer architectures of the Blue Gene family. As the performance issues with the ELPA two-stage solver that occured on JUQUEEN are meanwhile fixed in a newer version of the ELPA library, benchmarks regarding the improved efficiency of the new ELPA version are a compulsive assignment for further analysis. With the promising speedup of the new ELPA version, it is very likely that the crossline of the ELPA one- and two-stage solvers lies at about 45% of the eigenspectrum as it was the case on JUGENE.

An original intend was to benchmark the hybrid versions of both ELPA and Elemental on the Blue Gene systems, too. Unfortunately the hybrid versions of both libraries are up to now malfunctioning on JUQUEEN. A bug in the implementation of the MPI routine *Allreduce* in the Blue Gene/Q MPI implementation additionally delayed the benchmarks of Elemental on JUQUEEN. These benchmarks were conducted after this issue was resolved in July 2012. As the compilers for the Blue Gene/Q architecture steadily improve, the analysis and comparision of the hybrid and non-hybrid versions of Elemental and ELPA is another possible starting point for future exploration.

Literaturverzeichnis

[1] FZJ-JSC System Configuration - IBM Blue Gene/P. http://www2.fz-juelich.de/jsc/service/sco_ibmBGP, September 2012.

[2] PMRRR - Parallel Multiple Relatively Robust Representations. http://code.google.com/p/pmrrr/, September 2012.

[3] F. Allen and G. Almasi et al. Bluegene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40:310–328, 2001.

[4] T. Auckenthaler and V. Blum et al. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing*, 37:783–794, 1997.

[5] Tommy Berg. Performance-Analyse und -Optimierung paralleler Eigenwertlöser auf Bluegene Architekturen. Master's thesis, Fachhochschule Aachen Abt. Jülich, 2011.

[6] C. Bischof, B. Lang, and X. Sun. Parallel tridiagonalization through two-step band reduction. *Proc. Scalable High-Performance Computing Conf., IEEE Computer Society Press, Los Alamitos, CA*, pages 23–27, 1994.

[7] C. Bischof and C. Van Loan. The wy representation for products of householder matrices. *SIAM*, 8:2–13, 1987.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[9] J.R. Bunch, C.P. Nielsen, and D.C. Sorensen. Rank one modification of the symmetric eigenproblem. *Numer. Math.*, 31:31–48, 1978.

[10] J.J.M. Cuppen. *A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem*. PhD thesis, Universiteit of Amsterdam, 1981.

[11] J.W. Demmel. *Applied numerical linear algebra*, pages 5, 12. SIAM, 1997.

[12] I.S. Dhillon. *New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue Eigenvector Problem*. PhD thesis, University of California, 1997. 3.3.1, 3.3.4.

[13] I.S. Dhillon and B.N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear algebra and its applications*, 387:1–28, 2004.

[14] I.S. Dhillon, B.N. Parlett, and C. Vömel. The design and implementation of the mrrr algorithm. *ACM Transactions on Mathematical Software*, pages 533 – 560, 2006. 4.2.1.2.

[15] ELPA Authors. Eigenvalue SoLvers for Petaflop Architectures Website. http://elpa.rzg.mpg.de/, August 2012.

[16] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[17] G.H. Golub. Some modified matrix eigenvalue problems. *SIAM*, 15:318–334, 1973.

[18] G.H. Golub and C.F. Van Loan. *Matrix computations*, pages 2–5. Johns Hopkins, 1996. 3.2,3.3.4.

[19] Green500 Supercomputing Sites. June 2012 Green500 List. http://www.green500.org/lists/green201206, June 2012.

[20] B. Hendrikson, E. Jessup, and C. Smith. Toward an efficient parallel eigensolver for dense symmetric matrices. *SIAM J. Sci. Comput.*, 20:1132–1154, 1999.

[21] B. Lang. Direct solvers for symmetric eigenvalue problems. In J. Grotendorst, editor, *Modern Methods and Algorithms of Quantum Chemistry*, number 2, pages 231, 259. NIC Series, 2000.

[22] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2).

[23] TOP500 Supercomputing Sites. June 2012 Top500 List. http://www.top500.org/list/2012/06/100, August 2012.

[24] TOP500 Supercomputing Sites. November 2004 Top500 List. http://www.top500.org/list/2004/11/100, November 2004.

[25] TOP500 Supercomputing Sites. November 2006 Top500 List. http://www.top500.org/list/2006/11/100, November 2006.

[26] TOP500 Supercomputing Sites. November 2007 Top500 List. http://www.top500.org/list/2007/11/100, November 2007.

Jül-4359
Januar 2013
ISSN 0944-2952

JÜLICH
FORSCHUNGSZENTRUM