TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

Highly Scalable Eigensolvers for Petaflop Applications

Thomas Auckenthaler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 01.10.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.12.2012 angenommen.

Acknowledgments

At this place I want to thank all those people who contributed to this thesis.

First of all, I want to thank my supervisor Prof. Hans-Joachim Bungartz. He gave me a lot of support and motivation and always found an empty slot in his overfull time schedule. Thanks also to Prof. Thomas Huckle who accompanied me during the ELPA project and was like a second supervisor to me. Special thanks go to Prof. Bruno Lang. Due to his respective scientific background he was source of many interesting ideas and approaches. Thank you for the pleasant collaboration and many fruitful discussions.

Moreover, I want to thank all members of the ELPA consortium. Especially Dr. Hermann Lederer, Dr. Rainer Johanni, Dr. Paul Willems, Dr. Volker Blum and Mario Thüne. I enjoyed the collaboration and I am proud to be part of this team.

I want to thank Prof. Michael Bader for convincing me for this step of my life and for helpful comments while writing this thesis. Thanks also to my students Roland Wittmann and Stefan Schulze Frielinghaus. Last, but not least, I want to thank all members of the Chair of Scientific Computing in Computer Science in Munich for a really enjoyable atmosphere.

Abstract

High-performance computing (HPC) is an area of research which is subject to big changes. Fundamental properties of computer architectures such as cache, floating-point, or memory performance, e.g., are progressing at different pace. All this and other trends make it a challenging task for algorithm and software designers to exploit the available power of leading-edge supercomputers in real world applications.

One such important problem, which has been identified as a so-called "Grand Challenge", is the solution of symmetric eigenproblems. In current state-of-the-art implementations there is a lack of both parallel and sequential efficiency on modern computer architectures. From an improvement of parallel eigensolvers many scientific disciplines, such as structural mechanics, fluid mechanics, or quantum chemistry, could benefit.

This thesis presents the development of the parallel eigensolver library ELPA and highlights the challenges of a hardware-aware algorithm design. We have implemented and evaluated existing and developed new algorithms for all stages within a parallel symmetric eigensolver. New algorithms have been designed towards cache efficiency and avoidance or reduction of communication. All implementations have consequently been optimized. Competitiveness and usability of ELPA are demonstrated in two different scientific applications: a software package for ab-initio molecular simulations (FHI-aims) and an application for the inspection of large networks. Efficiency and scalability of the developed eigensolver are unprecedented and result in an up to 10-fold improvement compared to current state-of-the-art libraries.

Contents

1 Introduction

Nowadays, developing software for supercomputers is a complex task. On the one hand we need a profound understanding of current technologies and developments in the field of High Performance Computing (HPC). On the other hand we need knowledge of available algorithms and their behavior to solve specific problems in scientific computing. It is the task of a computer scientist to match those requirements or even develop new algorithms to get optimal outcomes.

1.1 High Performance Computing

The history of High Performance Computing goes back to the 1970s which was mainly marked by the name Seymour Cray building the first supercomputers. Since 1993 the fastest supercomputers in the world are tracked in the so called Top500 list [1]. As a benchmark for this list serves the LINPACK benchmark which solves a huge dense linear system of equations. Since the starting of the Top500 list we observe an exponential growth in computing power, leading to a tenfold increase in LINPACK performance every 3-4 years (see Figure 1.1). The architectures, however, on which this performance was achieved, were subject of some minor and major changes. The 1970s and mid 1980s were dominated by so called vector processors (SIMD, single instruction multiple data) where a single instruction is applied on a vector of data. From the late 1980s on we can observe a trend towards massive parallel systems which were connected over a high speed network. Finally (starting from the current millennium), due to economic reasons, specialized processors for supercomputing were progressively displaced by clusters of commodity hardware, mainly based on Power- and x86-architectures.

Nowadays we are in the middle of a further changeover in processor technology which is caused by a phenomenon called "the power wall". Moore's Law [2] predicts that the number of transistors on a single chip will double every 18 to 24 months. Until the mid 2000s this resulted in an increased processor clock speed which in turn speeded up most applications without any changes on the software level. The exponential increase in power, required to further increase processor cycle times, however, limits

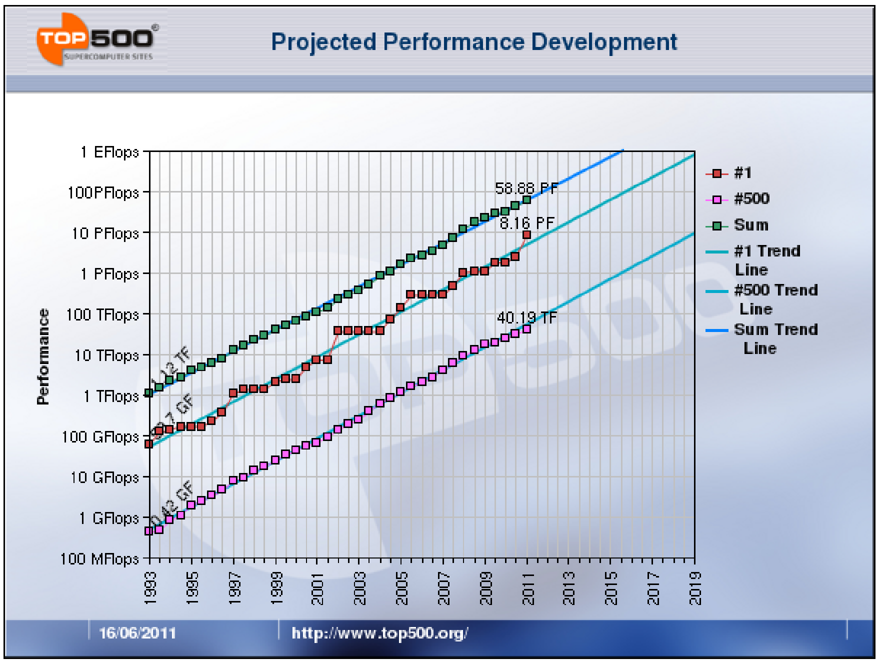


Figure 1.1: Performance development in the Top500 list.

any substantial growth in processor clock speed. Todays performance gains come mostly from parallelism on the chip level. This can result in a widening of the floating point units (instruction level parallelism, ILP) or an increase of the core count on a single chip (thread level parallelism, TLP). Examples for an increased ILP can be found all over the place. Intel introduced the AVX instruction set which supports four double precision floating point operations within one instruction instead of two (SSE). The Power7 and PowerPC A2 (BlueGene/Q) of IBM will have twice as many FPUs compared to its predecessors (Power6 and PowerPC 450 respectively). The trend towards multicore can be observed since a couple of years. Today the core counts range from 2 to 64 on a single chip. The most common programming paradigm on these chips is the shared memory model with two or more levels of cache. The probably most challenging task in designing current and upcoming multicore chips is to provide cache coherence between the individual cores and their caches. This is done with so called cache coherence protocols (e.g., MESI). Thereby each cache has to listen for memory accesses of all other caches on the same cache level (bus snooping). If another cache reads or writes data which exists already in the own cache, the right action has to be initiated (invalidate, write back, etc.). It is obvious that this mechanism will stop scaling at some point. One consequence of this limited scaling is the introduction of shared caches which can be found in most multicore chips with a substantial number of cores. An alternative to the shared memory model is the distributed memory model on

	Annual change [%]	Typical value in 2004	Typical value in 2020
FP-performance [Gflops]	59	2	3300
Memory bandwidth [GB/s]	25	8	216
Memory latency [ns]	(5.5)	70	28
MPI bandwidth [GB/s]	26	0.5	20
MPI latency [ns]	(28)	3000	300
No. of processors	20	4000	74000

Table 1.1: Single chip and parallel hardware trends [6].

a single chip. In [3] and [4] prototypes have been presented where the cores on a chip are organized in a two-dimensional mesh. Each core has a local storage, the cores are connected over a low-latency on-chip network. The corresponding programming model would be some kind of "lightweight MPI". An early example for such an architecture has been the Cell processor [5] where 8 so-called Synergistic Processing Elements have been connected over a ring bus. A third example for massive parallelism on a chip are accelerator technologies, e.g. in the form of GPUs. Right now it is hard to predict which model will come out on top.

Beside these minor and major revolutions we can also observe some continuous trends in the HPC landscape. The performance of HPC systems is characterized by a vast number of different properties, ranging from crucial measures such as network bandwidth or floating point performance to details such as cache replacement strategies or network protocols. Some of these properties are measurable and are tracked since a couple of years or decades.

Table 1.1 shows the average growth rate of important metrics of HPC systems from 1988 to 2004. We can see that the floating point performance increased by an average of 59% over the 16-year period. This trend is expected to continue. However, a substantial fraction of the performance increase will come from any form of on-chip parallelism. At the same time the memory bandwidth has increased by an average of 25% per year. This means that the gap between floating point performance and memory bandwidth has grown by 30% each year (memory wall [7, 8]). The memory latency was decreasing at the slowest pace of 5.5% per year such that more and more data has to be loaded in advance to fully load the memory bandwidth. This will be a big challenge for coming prefetching units. Regarding network bandwidth and network latency the growth rates are similar to those of the memory performance. The decrease of network latency was remarkably fast with 28%. However, this trend is expected to slow down due to physical limitations (e.g., propagation speed of light).

Another trend is the growing size of HPC systems. In the mentioned 16-year period the number of processors in one system increased by an average of 20% per year. The total level of parallelism, however, will grow much faster due to the additional on-chip parallelism. The current number one in the Top500 list (BlueGene/Q Sequoia, June 2012) has 98304 processors. With 16 cores per chip and 256-bit wide SIMD units, the system will be able to execute more than six million floating point operations (double precision fused multiply add operations) in each cycle.

These are trends which have to be taken into account when developing new algorithms for current and upcoming supercomputers. Different growth rates of different parameters lead to a shift of performance bottlenecks in HPC applications. An application which was floating-point-bounded in the past, may be memory-bounded on current architectures, and become even network-latency-bounded in the future. A software engineer has to choose or design algorithms according to these surrounding conditions to get optimal results.

Beside the use of appropriate algorithms, the choice of proper programming model(s) is another important design decision when writing parallel code. As described earlier, recent developments in the architectures of supercomputers have let to a very heterogeneous mix of technologies and a bunch of different levels of parallelism. The only constant in these systems seem to be the most fine grained and coarse grained levels of parallelism which can be found in nearly all systems in the current Top500 list. These are the SIMD parallelism on the instruction level and the distributed memory parallelism on the system level. In between we can find various types of NUMA (non uniform memory access) architectures or accelerator technologies.

Meanwhile, there exist many parallel programming models to tackle these different architectures. Beside the well established programming models for distributed and shared memory systems, with MPI [9] and OpenMP [10] as their most important representatives, there exist so called PGAS (partitioned global address space) languages. PGAS languages, such as Unified Parallel C (UPC) [11], Co-array Fortran (CAF) [12], Chapel [13] or X10 [14], have the aim to hide explicit communication from the programmer and, thus, to simplify the development of parallel codes. Even though these are desirable goals, PGAS languages still don't reach satisfying performance [15, 16, 17, 18] and are, thus, no option for the use in production codes.

Another class of programming models, which became very popular in the last few years, are dataflow approaches. The user of such models defines tasks to express his parallel program. A task is a portion of computational work with a defined set of input and output parameters. The input and output of all tasks represent the data dependencies of an application such that each parallel program can be expressed through a DAG (directed acyclic graph). Finally, at runtime, any form of scheduling system has to assign tasks to the individual processing units. There exist different implementations using this type of programming model. The most important one is the StarSs family, containing implementations for shared memory systems (SMPSs [19]), distributed memory systems (ClusterSs [20]), and other types of systems, such as the Cell processor [21]. PLASMA [22] and DPLASMA [23] are dedicated libraries which use the dataflow approach to express the parallelism of linear algebra routines. While DAG approaches are well suited to distribute work (also in heterogeneous computing environments), in the authors opinion, there are still some unresolved problems regarding communication in distributed memory systems. In message passing programming models, such as MPI, there exist highly optimized collective communication operations which, in some sense, lead to a parallelization of communication load, e.g. by using tree- or pipelining algorithms for a broadcast. All mentioned dataflow approaches which address distributed memory systems lack in such a mechanism to parallelize communication. Especially in dense linear algebra applications, where an efficient communication pattern is essential, this is a serious drawback.

For the mentioned reasons, we use the well established MPI to express distributed memory parallelization. Shared memory models as well as programming models which address GPGPU computing or any form of accelerators will not be considered here.

1.2 Eigensolvers: Fields of application

In this thesis we pick up a problem which is of great interest in many scientific disciplines and is currently tackled by many research groups all over the world. The symmetric eigenproblem is one of few problems in dense linear algebra which was not satisfactorily solved at the beginning of our research. On the basis of the symmetric eigenproblem we will demonstrate the hardware aware design of a new library routine for the use on massively parallel systems. Although our algorithms were implemented as a library routine which can be used wherever a parallel symmetric eigensolver is needed, we wanted to demonstrate the outcome on two scientific applications.

1.2.1 Ab-initio molecular simulations

The invention of accurate models, together with the increasing computing capabilities of modern hardware, made the field of quantum chemistry an ideal candidate for numerical simulations on HPC systems. The prediction of properties of materials at the atomic level requires computations on the quantum mechanical level, based on the Schrödinger equation. An analytical solution of the Schrödinger equation is only possible for the most restricted cases, such as the simulation of a hydrogen atom. For more complex simulations we need appropriate approximations. This leads to so called ab-initio methods. Today's most prevalent models for such approximations are based on Kohn-Sham density-functional theory (DFT) [24]. Thereby, using a set of approximations and discretizations, the time-independent Schrödinger equation,

$$\hat{\mathcal{H}}\Phi_i(\vec{X}) = E_i\Phi_i(\vec{X}), \tag{1.1}$$

is reduced to a non-linear generalized matrix eigenproblem:

$$\hat{h}c_i = \epsilon_i S c_i, \tag{1.2}$$

where $\hat{h}$ is a Hamilton matrix of size $O(N)$ (N refers to the total number of particles in the system). c_i and S result from the discretization of the wave functions Φ_i, with $\Phi_i(r) \approx \sum_j c_{ij}\phi_j(r)$. Our application in mind (`FHI-aims` [25]) uses so-called numeric atom-centered orbitals as basis functions ϕ_j. However, there exist several other discretization approaches (e.g., plane waves, Gaussian-type orbitals) which all result in a (generalized) non-linear symmetric eigenproblem. To get the eigenvectors of the non-linear eigenproblem, a linear eigenproblem is solved in a so-called self-consistent loop where the eigenvectors of the previous iteration are used to compute the next Hamilton matrix. In practice, this requires ten(s) of iterations until convergence is reached. For long ab-initio molecular simulations, in turn, tens of thousands of such self-consistent loops have to be solved successively.

Beside the solution of (generalized) eigenproblems, there are other costly operations within those self-consistent loops. These operations can vary between the different approaches for ab-initio molecular simulations. In `FHI-aims`, for example, all other operations have a runtime behavior of $O(N)$ with a huge constant, whereas the solution of the eigenproblem has a complexity of $O(N^3)$. From [25] can be seen that the solution of the eigenproblem becomes dominating (i) for simulations with more atoms ($O(N)$ vs. $O(N^3)$ runtime behavior) and (ii) for simulations using an increasing number of parallel cores (since the solution of the eigenproblem is the worst scaling substep of the simulation). Performance studies for other types of simulations (e.g. [26], for simulations using plane waves as basis functions) come to similar results and identify the symmetric eigenproblem as the bottleneck of ab-initio molecular simulations. Considering the demands for the simulation of even larger systems and increasing timescales, there is a tremendous need for an efficient, highly scalable eigensolver.

1.2.2 Inspection of large networks

Another important application of symmetric eigensolvers is the analysis of large networks. The study of complex networks has become a major field of interdisciplinary research [27]. Networks can be found all over the place (e.g. in the form of power grids, social networks, etc.) and are modeled with graphs, consisting of vertices and edges. One of the main objectives of related research is to understand the structure of those networks. While in the past the focus was on the analysis of single small graphs and the properties of individual vertices or edges, we, meanwhile, can observe a shift towards larger networks where large-scale statistical properties are of interest [28]. One approach to extract such information from a graph is the analysis of the eigenspectrum, more precisely the eigenspectrum of the Laplacian of the graph.

The Laplacian of a (undirected, unweighted) graph is defined as

$$\mathcal{L}(u,v) = \begin{cases} 1 & \text{if } u = v \text{ and } d_v \neq 0, \\ -\frac{1}{\sqrt{d_u d_v}} & \text{if } u \text{ and } v \text{ are adjacent,} \\ 0 & \text{otherwise,} \end{cases}$$

where d_v is the degree of the vertex v (number of edges incident to the vertex). Obviously $\mathcal{L}$ is symmetric. For many classes of graphs (e.g. stars, cycles, complete graphs) the eigenspectra can be computed analytically [29]. By comparing the eigenspectrum of a graph with these eigenspectra, it is possible to classify real-world graphs or to show similarities with certain classes of graphs. [30] and [31] show related work.

Usually, the matrices, resulting from real-world networks, are very sparse and, in practice, not limited in size. Due to the possible appearance of large eigenvalue-clusters, the eigenspectra of these matrices cannot be computed satisfactorily with iterative solvers [32] and require the use of direct eigensolvers. Moreover, by reordering, some of the matrices can be brought to banded form (e.g. using Cuthill/McKee [33]) which significantly reduces the complexity of direct eigensolvers. Hence, the field of network analysis would profit from the development of more efficient direct eigensolvers in general and especially from the development of banded eigensolvers which exploit the capabilities of modern hardware.

1.3 Overview of this thesis

This thesis is organized as follows. In Ch. 2 we will present the symmetric eigenproblem and give an overview of the available algorithms for symmetric eigensolvers when a large fraction of the eigenspectrum is required. In particular, we will present the algorithmic variants to bring a symmetric matrix to tridiagonal form, the most performance relevant step of a symmetric eigensolver. In Ch. 3 we will present all details of the parallel two-step tridiagonalization which is probably the most promising approach to bring a dense symmetric matrix to tridiagonal form. Beside presenting existing algorithms, we will describe all our new developments and improvements which are crucial for an efficient eigensolver. We will provide a runtime estimation for all existing and newly developed algorithms, according to a defined performance model. We conclude the chapter with a discussion of the scalability of the algorithms and an overview of libraries and research tackling the same problem. In Ch. 4 we will outline all details of our implementation which can't be described by our performance model but are still crucial for the achieved performance. Furthermore, we will provide performance results for all parts of our symmetric eigensolver and give a detailed discussion thereof. Finally in Ch. 5 we conclude our thesis by giving an overview of what we reached and by giving an outlook on upcoming challenges.

2 Symmetric eigensolvers

The following chapter will give a brief introduction into eigenproblems and will, then, present the basic techniques for the developed symmetric eigensolvers. The chapter is not intended to give a complete overview but to provide the necessary knowledge for the algorithms presented in Ch. 3. Reference to further literature can be found in [34], [35], or [36]

2.1 The eigenvalue problem

The standard eigenvalue problem is defined by the equation

$$Ax = \lambda x, \quad x \neq 0, \tag{2.1}$$

where $A \in \mathbb{C}^{n \times n}$, $x \in \mathbb{C}^n$ and $\lambda \in \mathbb{C}$. One solution (λ_i, x_i) of Equation (2.1), consisting of the eigenvalue λ_i and the eigenvector x_i, is called eigenpair. Some applications are interested in both, eigenvalues and eigenvectors, whereas in some problems the computation of eigenvalues is sufficient.

For Hermitian or real symmetric matrices some properties can be used which reduce the complexity of the eigenproblem. If A is Hermitian it can be shown that

- A has exactly n real eigenvalues λ_i (not all need to be distinct),
- each λ_i has an associated eigenvector x_i, and
- all eigenvectors can be defined to be mutually orthogonal, i.e. $x_i^* x_j = 0, i \neq j$.

The eigenvectors x_i are real if A is real symmetric. Due to the orthogonality of the eigenvectors, Equation (2.1) can be written as

$$A = X \Lambda X^*, \tag{2.2}$$

where X is the eigenvector matrix $[x_1, x_2, ..., x_n]$ and $\Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_n)$.

The generalized eigenvalue problem is defined as

$$Ax = \lambda Bx, \quad x \neq 0, \tag{2.3}$$

where A and B are two n by n matrices. For the Hermitian or rather real symmetric case A and B are Hermitian and real symmetric respectively and the matrix B has to be positive definite. A generalized Hermitian eigenproblem can always be converted to a standard Hermitian eigenproblem in the following way:

(1) Decompose $B = LL^*$, e.g. using Cholesky-factorization, where L is a lower triangular matrix.

(2) Solve the standard Hermitian eigenproblem for $\hat{A} = L^{-1}A(L^{-1})^*$.

(3) Compute the eigenvectors x of the generalized eigenproblem out of the eigenvectors $\hat{x}$ of $\hat{A}$, i.e. $x_i = (L^{-1})^*\hat{x}_i$.

Hence, it is sufficient to handle the standard case.

Definition 1 (Similarity transformation) *If $A, Q \in \mathbb{C}^{n\times n}$ and Q is nonsingular, then we say that A and $B = Q^{-1}AQ$ are similar. Similar eigenproblems have the same eigenvalues ($\Lambda_A = \Lambda_B$). Q is called similarity transformation.*

A standard Hermitian eigenproblem $A = X_A\Lambda_A X_A^*$, in turn, can be transformed into a similar eigenproblem $T = X_T\Lambda_T X_T^*$ by using a unitary transformation Q with $T = QAQ^*$. For the similar eigenproblem holds

$$\Lambda_A = \Lambda_T \tag{2.4}$$

and

$$X_A = Q^* X_T. \tag{2.5}$$

The matrix Q is called unitary similarity transformation.

The presented properties hold for real symmetric as well as for Hermitian matrices. However, for reasons of simplicity we will limit our further explanations and analysis of the presented algorithms to the real symmetric case.

2.2 Direct and iterative eigensolvers

A common classification of eigensolvers is a distinction between direct and iterative eigensolvers. Direct solvers have in common that the eigenproblem is first transformed to a similar eigenproblem which is easier to solve. Usually, these are transformations to tridiagonal form. A direct transformation to diagonal form is, in general, not possible, for reasons which will become clear during the next sections. In a second step, the transformed eigenproblem is solved. It has to be mentioned that the solution of this transformed eigenproblem in turn is an iterative process. Nevertheless, only iterative algorithms which operate directly on the original matrix are classified as iterative eigensolvers [34].

Iterative eigensolvers are appropriate if only a small fraction of the eigenspectrum is desired. The matrices are usually large and sparse. Common methods are e.g. the Lanczos method, Jacobi-Davidson methods or the Jacobi method. Iterative methods are not discussed in this thesis. Further information, on when and how to use iterative solvers, can be found in [36].

Direct solvers are based on similarity transformations. Due to accuracy issues, orthogonal similarity transformations are used. The initial eigenproblem is successively transformed to an eigenproblem which is easier to solve. The most common proceeding is to bring a matrix A to tridiagonal form. To compute the eigendecomposition of the tridiagonal matrix, in turn, exists a variety of methods (e.g. QR iteration, Divide-and-Conquer, MRRR). Direct solvers are the only feasible choice if the whole or a large fraction of the eigenspectrum is desired [32].

2.3 Tridiagonal based (direct) eigensolvers

According to [36], the most common method to compute the whole eigenspectrum of a matrix are tridiagonal based eigensolvers. As depicted in Figure 2.1, tridiagonal based eigensolvers consist of three phases:

1. Reduce the dense symmetric matrix A to symmetric tridiagonal form.
2. Solve the tridiagonal eigensystem.
3. Transform the eigenvectors of the tridiagonal matrix back to those of the matrix A.

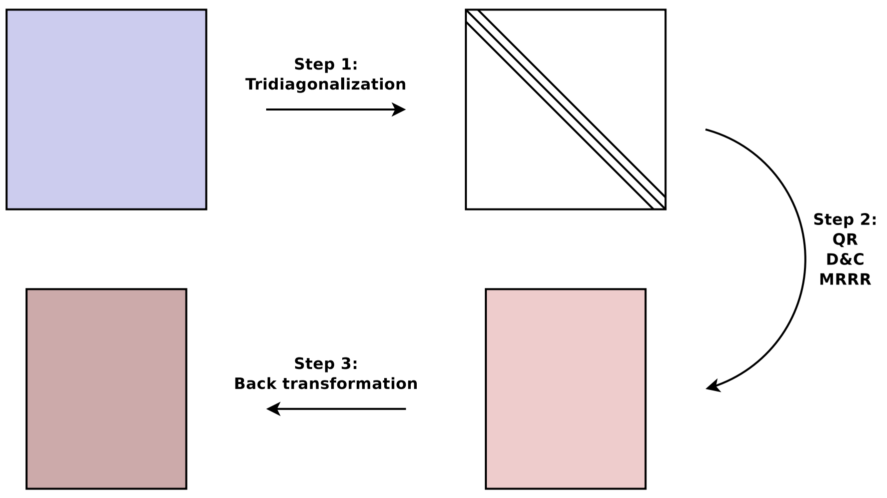


Figure 2.1: Three phases of a tridiagonal based eigensolver. Phase 1: Tridiagonalization. Phase 2: Tridiagonal eigensolver. Phase 3: Back transformation of eigenvectors.

In Sect. 2.3.1, 2.3.2, and 2.3.3 we will briefly describe the three individual stages (tridiagonalization, tridiagonal eigensolver, back transformation) of the eigensolver. Afterwards, in Sect. 2.3.4, we will give a detailed overview on orthogonal similarity transformations, the basic building block of each tridiagonal based eigensolver.

2.3.1 Tridiagonalization

For the reduction of a symmetric matrix A to tridiagonal form T a sequence of orthogonal transformations Q_i has to be found, with

$$T = Q_l \cdot \ldots \cdot Q_2 \cdot Q_1 \cdot A \cdot Q_1^T \cdot Q_2^T \cdot \ldots \cdot Q_l^T. \tag{2.6}$$

The 2-sided application of the transformations preserves the symmetry of the matrix. The sequence of orthogonal transformations may be any combination of Givens- or Householder transformations whereas each transformation introduces zeros into the matrix. Nevertheless, most relevant implementations rely on Householder transformations [37, 38, 39, 40].

The order and form of those Householder transformations is crucial to achieve high performance on modern computer architectures. As we will see, there exist different strategies to achieve this tridiagonal form (1-step tridiagonalization, 2-step tridiagonalization, successive band reduction). These strategies will be presented in Sect. 2.5.

2.3.2 Tridiagonal eigensolvers

The symmetric tridiagonal eigenproblem is a well studied field with lots of different methods for its solution. According to [41], among the most efficient and accurate ones we can mention

- Bisection and Inverse Iteration [42],
- QR algorithm [43],
- Divide & Conquer (D&C) [44], and
- MRRR - Multiple Relatively Robust Representations [45, 46].

All mentioned algorithms are numerically stable. However, they differ regarding performance, achievable accuracy and other properties such as parallelizability or the possibility to compute only a part of the eigenspectrum at reduced cost.

Bisection and Inverse Iteration as well as the QR algorithm and the Divide & Conquer approach have all a worst case complexity of $O(n^3)$ flops. However, the first allows to compute a fraction of k eigenpairs at reduced cost of $O(kn^2)$ flops. D&C, in turn, can be speeded up due to a technique called deflation. The speedup of deflation depends on the content of the tridiagonal matrix but can be significant. In [47] a modification of the D&C algorithm is introduced which can compute partial eigensystems at reduced cost. The savings are not linear in k but lead to speedups of up to 3.

The MRRR algorithm is the only stable algorithm which allows the computation of k eigenpairs with a worst case complexity of $O(kn)$ flops. Due to the reduced number of required flops, MRRR is usually the fastest available algorithm. However, as we said before, the runtime depends on the content of the matrix, such that D&C may outperform MRRR in certain scenarios. Bisection and Inverse Iteration as well as QR are, in general, not competitive.

Regarding accuracy, D&C and the QR algorithm are, in general, preferable over MRRR and Bisection and Inverse Iteration [48]. In [49] we can find research towards improving the robustness of MRRR. In [50] a mixed precision variant of MRRR is presented which reaches the same or even better accuracy results compared to QR and D&C.

Altogether, D&C and MRRR are the by far most promising approaches for the symmetric tridiagonal eigenproblem. Moreover, both algorithms show a good behavior for parallel execution [51].

Since the focus of this thesis lies on the tridiagonalization of symmetric matrices, we gave only a brief overview of available methods and their properties. For further information we refer to the cited literature.

2.3.3 Back transformation of eigenvectors

According to Equation (2.5), the eigenvectors X_T of the tridiagonal matrix have to be transformed back to the eigenvector X_A of the original matrix. This is done by applying all the orthogonal transformations from the tridiagonalization step in reverse order to the eigenvector matrix:

$$X_A = Q_1^T \cdot Q_2^T \ldots \cdot Q_{l-1}^T \cdot Q_l^T \cdot X_T. \tag{2.7}$$

Number, order, and form of the orthogonal transformations Q_i, obviously, depend on the tridiagonalization strategy and will be described more detailed in Sect. 2.5. Independent from this, the back transformation of eigenvectors is much better suited for parallel execution compared to the tridiagonalization. For the back transformation all orthogonal transformations are known in advance which eliminates many data dependencies.

An alternative approach would be to accumulate the Householder transformations during the reduction to tridiagonal form:

$$Q = I \cdot Q_1^T \cdot Q_2^T \ldots \cdot Q_{l-1}^T \cdot Q_l^T. \tag{2.8}$$

The eigenvectors can then be transformed back with one huge matrix matrix multiplication (QX_T). This approach saves memory for the storage of Householder vectors if the reduction is done with more than two steps (see Sect. 2.5). However, the computational complexity increases from $O(kn^2)$ to $O(n^3)$, if k is the number of computed eigenvectors.

2.3.4 Algorithmic kernels: orthogonal similarity transformations

Orthogonal similarity transformations are the basic instrument to bring a dense symmetric matrix to tridiagonal form. A wisely chosen Givens rotation can bring an element of a matrix to zero. An appropriate Householder transformation can zero all but one element of a given vector. The tridiagonalization of a matrix consists of a series of such orthogonal transformations. In the following these two types of transformations are presented.

Givens rotations

Givens rotations [52] are plane rotations represented by the matrix

$$G(i,j,\Theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

where $G(i,j,\Theta)$ is the identity matrix I, overwritten with $g_{i,i} = g_{j,j} = c = \cos\Theta$ and $g_{i,j} = -g_{j,i} = s = \sin\Theta$ for $i > j$. Multiplying a vector with a Givens matrix $G(i,j,\Theta)$ corresponds to a counterclockwise rotation at an angle of Θ in the (i,j) plane.

If we choose c and s such that $c = a_{j,j}/r$, $s = a_{i,j}/r$ and $r = \sqrt{a_{j,j}^2 + a_{i,j}^2}$, we can set the element $a_{i,j}$ to zero by computing $G^T A$. To get better numerical stability we find also the following definition to compute c and s [43]:

$$s = \frac{1}{\sqrt{1+\gamma^2}}, \qquad c = s \cdot \gamma \qquad \text{and} \qquad \gamma = \frac{a_{j,j}}{a_{i,j}} \qquad \text{if } |a_{j,j}| \le |a_{i,j}|, \tag{2.9}$$

$$c = \frac{1}{\sqrt{1+\tau^2}}, \qquad s = c \cdot \tau \qquad \text{and} \qquad \tau = \frac{a_{i,j}}{a_{j,j}} \qquad \text{if } |a_{j,j}| > |a_{i,j}|. \tag{2.10}$$

Applied from the left side, a Givens rotation updates the rows i and j of the matrix, leading to $6n$ floating point operations ($4n$ multiplications, $2n$ additions) for a dense matrix with n columns. For the right-sided application column i and j are updated respectively.

In the literature we also find a variant named Fast Givens Rotation [53, 54], which reduces the number of floating point operations (flops) to $4n$ but suffers from a slightly worse numerical stability and an increased effort to prevent under-/overflow [55].

Givens rotations were not used for the developed algorithms. Although, they shouldn't be missing because they are utilized in other algorithmic approaches.

Householder transformations

Householder transformations [56, 57] are matrices of the form

$$H = I - \tau v v^T, \tag{2.11}$$

where $\tau = \frac{2}{v^T v}$. It can easily be shown that Householder matrices are symmetric and orthogonal.

Householder transformations can be used to zero selected entries of a vector. Let x be a vector of size n. If we set

$$v = x \pm \|x\| e_1, \text{ then} \tag{2.12}$$

$$Hx = (I - 2\frac{vv^T}{v^T v})x = \mp \|x\| e_1,$$

where e_i is the ith unit vector. $\|x\|$ always stands for the Euclidean norm. H is called a Householder transformation of order n.

If we want to zero only selected entries of a vector, we have to introduce zeros in the Householder vector. All entries in x, where the Householder vector v has zero entries, will be unchanged after the application of H.

$$(Hx)_i = x_i \quad \forall i : v_i = 0.$$

The sign in Equation (2.12) should be set to $+\operatorname{sign}(x_{(1)})$. If we set $v = x - \operatorname{sign}(x_{(1)})\|x\|e_1$ and x is close to a multiple of e_1, the norm of v is small, which may lead to a large relative error in τ [43].

A Householder transformation is defined for an arbitrary scaling of the Householder vector v. In the literature, however, we find the following common variants [35]:

(1) $v = x + \operatorname{sign}(x_{(1)})\|x\|e_1$,

(2) $v_{(1)} = 1$,

(3) $\|v\| = 1$.

In our algorithms we use the variant (2). For this type of scaling, the first entry of the Householder vector doesn't have to be stored explicitly. This allows us to store the Householder transformation in the new zero entries of a transformed matrix/vector. Algorithm 1 shows, how to compute such a scaled Householder vector v which introduces zeros in $x_{(2:n)}$. Naming schemes are those of [35]. It can easily be seen that

Algorithm 1 *HouseGen*$(x) \rightarrow v, \tau$

```
1: $\beta \leftarrow \sqrt{x_{(1:n)}^T x_{(1:n)}} \cdot \operatorname{sign}(x_{(1)})$
2: $\tau \leftarrow \frac{x_{(1)}+\beta}{\beta}$
3: $v \leftarrow (1, \frac{x_{(2:n)}}{x_{(1)}+\beta})^T$
4: $x \leftarrow (-\beta, 0_{(2:n)})^T$
```

Algorithm 1 has a complexity of $3n + O(1)$ floating point operations.

Householder transformations can be applied from the left side and from the right side to a matrix A. This corresponds to multiplying the Householder matrix H with the matrix A (HA and AH respectively). Using the structure of the Householder matrix leads to matrix vector operations. Algorithm 2 and Algorithm 3 show the left-sided and right-sided application of a Householder transformation. The number of floating point operations is $4nm$ for a Householder vector of size n and a matrix of size $n \times m$ ($HouseLeft$) and $m \times n$ ($HouseRight$) respectively.

Algorithm 2 $HouseLeft(A, v, \tau)$

```
1: $z^T \leftarrow \tau v^T A$
2: $A \leftarrow A - vz^T$
```

Algorithm 3 $HouseRight(A, v, \tau)$

```
1: $z \leftarrow \tau Av$
2: $A \leftarrow A - zv^T$
```

The 2-sided application of a Householder transformation (i.e., HAH) on a symmetric matrix A is shown in Algorithm 4. Note that, due to the symmetry, only the lower or the upper triangle of A has to be updated. Therefore $4n^2 + O(n)$ floating point operations are required for the symmetric application.

Algorithm 4 $HouseSymm(A, v, \tau)$

```
1: $z \leftarrow \tau Av$
2: $z \leftarrow z - \frac{\tau z^T v}{2} v$
3: $A \leftarrow A - vz^T - zv^T$
```

Blocked Householder transformations

As can be seen from Algorithm 2 to 4, the application of a Householder transformation on a matrix involves matrix vector operations. If we want to apply more than one Householder transformation, there exist so called blocked representations which allow the use of more efficient matrix matrix operations. Thereby, a product of Householder transformations is expressed by one or two matrices instead of a set of vectors. In the following we will present the two most common techniques for a blocked representation of Householder transformations, by name: WY transformations [58] and compact WY transformations [59].

WY transformations Let $H_i = I - \tau_i v_i v_i^T$ be a sequence of n_b Householder transformations of order n and let $Q_{n_b} = H_1 \cdot H_2 \cdot \ldots \cdot H_{n_b}$ be the product thereof. Then Q can be expressed in the form

$$Q_{n_b} = I - W_{n_b} Y_{n_b}^T, \tag{2.13}$$

where W_{n_b} and Y_{n_b} are matrices of size $n \times n_b$. W and Y can be constructed recursively:

$$W_{i+1} = [W_i, \tau_{i+1}(v_{i+1} - W_i(Y_i^T v_{i+1}))], \quad W_0 = [], \tag{2.14}$$

$$Y_{i+1} = [Y_i, v_{i+1}], \quad Y_0 = []. \tag{2.15}$$

The computations in Equation (2.14) and (2.15) require $4in + O(n)$ floating point operations. Thus, the construction of the final matrices W_{n_b} and Y_{n_b} has a complexity of $\sum_{i=0}^{n_b-1}(4in + O(n)) = 2n{n_b}^2 + O(nn_b)$.

The application of a WY transformation is similar to Algorithm 2 ($HouseLeft$), 3 ($HouseRight$) and 4 ($HouseSymm$), except that we use matrix matrix operations instead of matrix vector operations. Algorithm 5 and 6 show the left-sided ($A = Q^T A$) and right-sided ($A = AQ$) application of a blocked Householder transformation on a general matrix A. Algorithm 7 shows the symmetric application ($A = Q^T AQ$) on a symmetric matrix.

Algorithm 5 $WYLeft(A, W, Y)$

```
1: $Z^T \leftarrow W^T A$
2: $A \leftarrow A - YZ^T$
```

Algorithm 6 $WYRight(A, W, Y)$

```
1: $Z \leftarrow AW$
2: $A \leftarrow A - ZY^T$
```

Algorithm 7 $WYSymm(A, W, Y)$

```
1: $Z \leftarrow AW$
2: $Z \leftarrow Z - \frac{1}{2}Y(Z^T W)$
3: $A \leftarrow A - YZ^T - ZY^T$
```

It can be seen that Algorithm 5 and 6 require $4mnn_b$ floating point operations if A is a matrix of size $n \times m$ ($WYLeft$) and $m \times n$ ($WYRight$) respectively. The complexity of Algorithm 7 is $4n^2 n_b + 4n{n_b}^2 + O(nn_b)$ if the symmetry of A is exploited.

Till here, all shown algorithms (1 - 7) originate from [35].

Compact WY transformations Another well-known representation of blocked Householder transformations are compact WY transformations. Thereby a product of transformations $Q_{n_b} = H_1 \cdot H_2 \cdot \ldots \cdot H_{n_b}$ is expressed as

$$Q_{n_b} = I - Y_{n_b} T_{n_b} Y_{n_b}^T, \tag{2.16}$$

where Y_{n_b} is a matrix of size $n \times n_b$ containing the individual Householder vectors

$$Y_{n_b} = [v_1, v_2, \ldots, v_{n_b}], \tag{2.17}$$

and T_{n_b} is a triangular matrix of size n_b which can be constructed recursively:

$$T_{i+1} = \begin{bmatrix} T_i & z \\ 0 & \tau_{i+1} \end{bmatrix}, \quad z = -\tau_{i+1} T_i Y_i^T v_{i+1}, \quad T_0 = []. \tag{2.18}$$

The recursion in Equation (2.18) involves two matrix vector operations. An analysis of data dependencies reveals that each computation of $Y_i^T v_{i+1}$ can be done in advance using one matrix matrix operation $(Y^T Y)$. Note that, due to the symmetry, only the upper or the lower triangle has to be computed.

[60] and [61] come to a similar result. They show that

$$T^{-1}_{(i,i)} = \frac{v_i^T v_i}{2}, \quad T^{-1}_{(i,j)} = v_i^T v_j, \quad i < j. \tag{2.19}$$

Starting from Equation (2.19), T_{n_b} can be computed with the following three steps:

```
1: $S \leftarrow Y^T Y$ (compute only one triangle)
2: $S_{(i,i)} \leftarrow S_{(i,i)}/2$
3: $T \leftarrow S^{-1}$
```

Thereby the inversion of S in step 3 corresponds to the final computations of z in Equation (2.18) ($z_i = -\tau_{i+1} T_i x_i$; $x_i = (Y^T Y)_{(1:i,i+1)}$).

Summing up all the operations from Equation (2.18) leads to $nn_b^2 + n_b^3/3 + O(nn_b)$ flops.

Applying a compact WY transformation to a matrix is similar to the application of WY transformations, except of an additional matrix multiplication (YT). Algorithm 8, 9 and 10 show the left-sided, right-sided and symmetric application respectively.

The additional matrix multiplication leads to a complexity of $4mnn_b + nn_b^2 + O(nn_b)$ for Algorithm 8 and 9. For the 2-sided application on a symmetric matrix (Algorithm 10) we get costs of $4n^2 n_b + 5n{n_b}^2 + O(nn_b)$.

Algorithm 8 $CWYLeft(A, Y, T)$

```
1: $X \leftarrow YT$
2: $Z^T \leftarrow X^T A$
3: $A \leftarrow A - YZ^T$
```

Algorithm 9 $CWYRight(A, Y, T)$

```
1: $X \leftarrow YT$
2: $Z \leftarrow AX$
3: $A \leftarrow A - ZY^T$
```

For all presented algorithms concerning blocked Householder representations we assumed to have Householder transformations of full order. If, however, the Householder vectors have a structure of zero entries (e.g. Householder transformations resulting from a QR-decomposition), this structure may be exploited and the complexity of the algorithms can be adjusted downwards.

Whether to use WY or compact WY representations cannot be answered in general. Compact WY representations require less storage compared to the WY representation (compact WY: $nn_b + n_b^2/2 + O(n_b)$ words, WY: $2nn_b$ words). Furthermore the construction of the matrix W is more costly than the construction of T. However, compact WY representations need an additional matrix multiplication (of lower order) if applied to a matrix.

2.4 1-sided and 2-sided factorizations

Before going into the details of the individual tridiagonalization approaches, we want to give some background information which allows a more profound understanding of the faced problems.

A common classification of linear algebra algorithms is the distinction between 1-sided and 2-sided factorizations. The class of 1-sided factorizations contains algorithms such as the QR-factorization, the LU-factorization or the Cholesky-factorization. They have

Algorithm 10 $CWYSymm(A, Y, T)$

```
1: $X \leftarrow YT$
2: $Z \leftarrow AX$
3: $Z \leftarrow Z - \frac{1}{2}Y(X^T Z)$
4: $A \leftarrow A - YZ^T - ZY^T$
```

in common that the computed transformations are applied only from one side to the matrix. The QR-factorization, for example, decomposes a matrix $A \in \mathbb{R}^{m \times n}$ to

$$A = QR, \tag{2.20}$$

where Q is orthogonal and R is upper triangular. Algorithm 11 shows a QR-decomposition based on Householder transformations. The algorithm transforms the

Algorithm 11 QR-decomposition

```
1: for $i = 1 \to n - 1$ do
2:     $v_i, \tau_i \leftarrow HouseGen(A_{(i:m,i)})$
3:     $HouseLeft(A_{(i:m,i:n)}, v_i, \tau_i)$
4: end for
```

matrix A to R, the orthogonal matrix Q is defined by the product of Householder transformations:

$$Q = \prod_i (I - \tau_i v_i v_i^T). \tag{2.21}$$

Obviously, the tridiagonalization is part of the 2-sided factorizations because the orthogonal transformations have to be applied from both sides to the matrix. Whether an algorithm belongs to the class of 1-sided or 2-sided factorizations has huge impacts on the achievable performance on modern computer architectures. 1-sided factorizations can be formulated in a cache efficient way by doing a simple loop blocking. This is not the case for 2-sided factorizations.

Out of Algorithm 11, for example, we can easily formulate a blocked variant of the QR-decomposition (see Algorithm 12). The same can be done for the LU- and Cholesky-

Algorithm 12 QR-decomposition, blocked

```
1: for $i = 1 \to n - 1 : n_b$ do
2:     $V \leftarrow QR(A_{(i:m,i:i+n_b-1)})$
3:     $T \leftarrow CWYGen(V)$
4:     $CWYLeft(A_{(i:m,i+n_b:n)}, V, T)$
5: end for
```

factorization [37]. Blocking strategies for the tridiagonalization are discussed in the next section.

2.5 Tridiagonalization approaches

2.5.1 1-step Tridiagonalization

The 1-step tridiagonalization is the method which is currently used in most (parallel) linear algebra libraries for the tridiagonalization of symmetric matrices. The proceeding is sketched in Algorithm 13. The algorithm consists of $n-2$ iterations. In each

Algorithm 13 1-step Tridiagonalization

```
1: for $i = 1 \to n-2$ do
2:     $v, \tau \leftarrow HouseGen(A_{(i+1:n,i)})$
3:     $HouseSymm(A_{(i+1:n,i+1:n)}, v, \tau)$
4: end for
```

iteration i a Householder vector is generated which sets all elements from $i+1$ to n of column i to zero. The Householder transformation is then applied from both sides to the matrix. Due to the symmetric application, the symmetry of the matrix is preserved throughout the algorithm.

Each iteration of the algorithm requires $4(n-i)^2$ operations for the symmetric update and $3(n-i)$ flops for the generation of the Householder vector. This results in a complexity of $4/3n^3 + O(n^2)$ for the whole algorithm. All operations are done using memory-bounded matrix vector and vector vector operations and are, thus, very inefficient on modern computer architectures.

Algorithm 13 can also be formulated in a blocked fashion which allows that half of the operations can be done with more efficient matrix matrix operations [62]. Therefore, the operations from Algorithm 4 (*HouseSymm*) have to be split up. The matrix vector multiplication in line 1 and the lower order term in line 2 are computed as for the unblocked algorithm in every iteration. The rank-2 updates in line 3, however, are aggregated to rank-$2n_b$ updates

$$A_{i+n_b} = A_i - V_{n_b} Z_{n_b}^T - Z_{n_b} V_{n_b}^T,$$

where n_b is the blocking factor of the algorithm, A_{i+1} is the matrix A after the ith iteration and $V_{n_b} = [v_1, v_2, \ldots, v_{n_b}]$ and $Z = [z_1, z_2, \ldots, z_{n_b}]$. For the computation of z_i we have to consider that A is not updated in every iteration. Assuming that A_i is the last explicit available update of A, we can write

$$\begin{aligned} A_{i+j} v_{i+j} &= (A_i - V_j Z_j^T - Z_j V_j^T) v_{i+j} \\ &= A_i v_{i+j} - V_j Z_j^T v_{i+j} - Z_j V_j^T v_{i+j} \end{aligned}$$

Algorithm 14 1-step Tridiagonalization, blocked

```
1: for $i = 1 \to n-2 : n_b$ do
2:     $V_0 = [], Z_0 = []$
3:     for $j = 0 \to n_b - 1$ do
4:         $k \leftarrow i + j$
5:         $A_{(k:n,k)} \leftarrow A_{(k:n,k)} - V_{j\ (k:n,:)} Z^T_{j\ (k,:)} - Z_{j\ (k:n,:)} V^T_{j\ (k,:)}$
6:         $v, \tau \leftarrow HouseGen(A_{(k+1:n,k)})$
7:         $z \leftarrow A_{(k+1:n,k+1:n)} v - V_{j\ (k+1:n,:)} Z^T_{j\ (k+1:n,:)} v - Z_{j\ (k+1:n,:)} V^T_{j\ (k+1:n,:)} v$
8:         $z \leftarrow \tau(z - \frac{z^T v}{2} v)$
9:         $V_{j+1} \leftarrow \left[ V_j, \begin{pmatrix} 0_{1:k} \\ v \end{pmatrix} \right]$
10:        $Z_{j+1} \leftarrow \left[ Z_j, \begin{pmatrix} 0_{1:k} \\ z \end{pmatrix} \right]$
11:    end for
12:    $A_{(i+n_b:n,i+n_b:n)} \leftarrow A_{(i+n_b:n,i+n_b:n)} - V_{n_b\ (i+n_b:n,:)} Z^T_{n_b\ (i+n_b:n,:)} - Z_{n_b\ (i+n_b:n,:)} V^T_{n_b\ (i+n_b:n,:)}$
13: end for
```

This leads to Algorithm 14. The blocking allows that half of the $4/3n^3$ operations can be done using matrix matrix operations and increases the number of flops by $2n^2 n_b + O(n^2)$.

For the back transformation of eigenvectors we can make use of blocked Householder transformations. n_b Householder transformations are combined to one WY or compact WY representation and are then applied from the left side to the eigenvector matrix X_T. The back transformation is sketched by Algorithm 15. Assuming X_T is of size $n \times k$

Algorithm 15 1-step back transformation of eigenvectors

```
1: for $i = 1 \to n-2 : n_b$ do
2:     $T \leftarrow CWYGen(V_{(:,n-i-n_b:n-i-1)})$
3:     $CWYLeft(X_T, V, T)$
4: end for
```

(k eigenvectors of size n) we need $2kn^2 + n^2 n_b + O(nn_b^2, kn)$ floating point operations for the back transformation based on compact WY transformations. Beside of lower order terms all operations can be done with cache efficient matrix matrix operations.

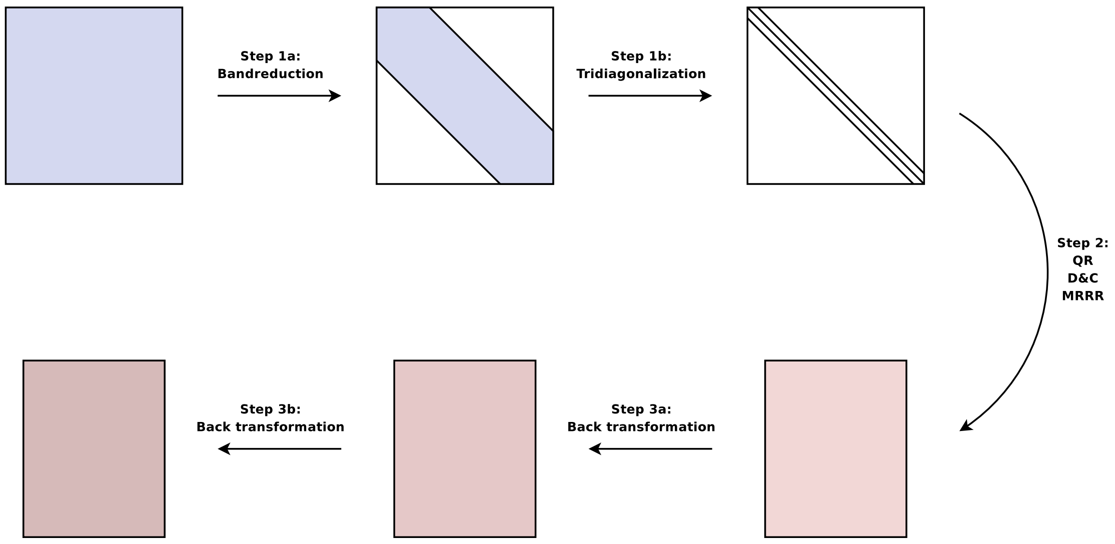


Figure 2.2: Tridiagonalization through the 2-step approach.

2.5.2 2-step Tridiagonalization

The 1-step tridiagonalization contains data dependencies which prevent a fully blocked formulation of the algorithm. The second Householder vector, for example, cannot be computed until the first transformation is applied on the second column/row of the matrix. A closer look at Algorithm 13 and 4, in turn, reveals that the second column/row of the matrix is affected by both, the left-sided and right sided application of the Householder transformation. These data dependencies can be avoided if the matrix is first reduced to banded form.

The 2-step tridiagonalization was originally presented in [63]. In the first step the symmetric matrix A is reduced to banded form. Afterwards the banded matrix B is brought to tridiagonal form (see Figure 2.2). Let b be the (semi)bandwidth of the banded matrix, i.e. $B_{(i,j)} = b_{i,j} = 0$ if $|i-j| > b$. For reasons of symmetry we consider only the lower triangle of the matrix.

Reduction to banded form

The reduction to banded form is similar to the unblocked tridiagonalization in Algorithm 13, except that not the $n-i-1$ but the $n-i-b$ lowermost elements of a column i are made to zero. For the application of the Householder vector we can partition A the following way: (1) $A_{(i+b:n,i:i+b-1)}$ is only affected by the left-sided application of the Householder transformation, (2) $A_{(i+b:n,i+b:n)}$ is updated by both, left- and right-sided application and (3) the rest of the matrix is not affected by the Householder transformation. Based on this partitioning we can formulate the blocking strategy in Algorithm 16. Each block $A_{(i+b:n,i:i+n_b-1)}$ can be decomposed to an orthogonal matrix

Algorithm 16 Reduction to banded form

```
1: for $i = 1 \rightarrow n - b - 1 : n_b$ do
2:     $V \leftarrow QR(A_{(i+b:n,i:i+n_b-1)})$
3:     $T \leftarrow CWYGen(V)$
4:     $CWYLeft(A_{(i+b:n,i+n_b:i+b-1)}, V, T)$
5:     $CWYSymm(A_{(i+b:n,i+b:n)}, V, T)$
6: end for
```

Q and an upper triangular matrix R. Afterwards, the orthogonal transformations can be applied in a blocked fashion from both sides to the rest of the matrix. The blocking factor n_b has to be smaller or equal to the bandwidth b of the resulting banded matrix. If b equals n_b, line 4 of Algorithm 16 can be omitted.

Note that Algorithm 16 is based on the QR-decomposition shown in Algorithm 11. However, different strategies for the parallel QR-decomposition require accordant algorithms for the reduction to banded form. Obviously, also the back transformation of eigenvectors of the banded matrix to those of the full matrix depends on the used strategy for the QR-decomposition. If Algorithm 11 is used for the QR-factorization, the back transformation is similar to Algorithm 15 (1-step back transformation of eigenvectors), unless that the length of each Householder vector is $n-i-b+1$ instead of $n-i$. Except of lower order terms, these operations require $2kn^2$ flops.

A detailed description and analysis of algorithms for the parallel reduction to banded from (and corresponding back transformation) will be given in Ch. 3.

Tridiagonalization of banded matrices

For the tridiagonalization of banded matrices (band reduction) exists a variety of algorithms which exploit the banded structure of the matrix. All algorithms remove all or part of intermediate fill-in to preserve the banded structure of the matrix. The removing of each fill-in generates new fill-in further down the matrix until the end of the matrix is reached and no new fill-in is generated. This proceeding is called chasing the fill-in and can be found in every algorithm mentioned in this section.

In [64] and [65] Schwarz proposes two algorithms based on Givens rotations. [64] successively decreases the bandwidth of the matrix by one. For every column of the

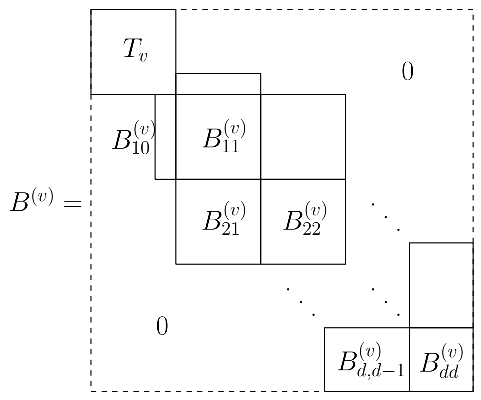


Figure 2.3: Partitioning of the block tridiagonal matrix during the band reduction [47].

matrix, starting from the first, the outermost element is zeroed with an appropriate Givens rotation. Fill-in is consequently removed and chased down the matrix. This proceeding is repeated until the matrix has tridiagonal form (bandwidth equals 1). The algorithm presented in [65] eliminates, column after column, all row entries of a column, starting from the outermost element. Again, all intermediate fill-in is removed as soon as it occurs. In [66] a blocked algorithm, based on the algorithms of Schwarz, has been presented. [64] requires asymptotically more transformations compared to [65] ($O(n^2 log(b))$ vs. $O(n^2)$). The number of required flops, however, is the same for both algorithms ($6n^2b$ with ordinary Givens rotations and $4n^2b$ with fast Givens rotations) because the work for each transformation decreases linearly with the bandwidth of the matrix.

[67] suggests an algorithm based on Householder transformations to introduce zeros in the matrix. The same idea has been used in [68] for a parallel band reduction. The algorithm in [68] builds the basis for our implementation and will be presented more detailed in the following.

The algorithm consists of $n-2$ stages whereas in each stage v the vth column of the matrix B is brought to tridiagonal form. Let $B^{(v)}$ denote the matrix B at stage v of the algorithm. Then, $B^{(v)}$ can be treated as a block tridiagonal matrix and be partitioned the following way (see Figure 2.3 and [47]): T_v is of size $v \times v$ and is already in tridiagonal form. $B^{(v)}_{10} = (b_{v+1,v}, \ldots, b_{v+b,v})^T$ contains, together with $b_{v,v}$, the first column of the remaining banded matrix. The diagonal blocks $B^{(v)}_{\beta\beta}$ and the subdiagonal blocks $B^{(v)}_{\beta+1,\beta}$ are both of size $b \times b$ for $\beta \geq 1$ (except for smaller blocks at the end of the band).

Each stage v is initiated by a length-b Householder transformation which reduces the first remaining column to tridiagonal form:

$$\tilde{B}_{10} = Q_1^{(v)} B_{10}^{(v)} = (*, 0, \ldots, 0)^T. \tag{2.22}$$

This transformation must then be applied to the rest of the matrix, resulting in a symmetric update of $B_{11}^{(v)}$ and a right-sided update of $B_{21}^{(v)}$: $\tilde{B}_{11} = Q_1^{(v)} B_{11}^{(v)} {Q_1^{(v)}}^T$, $\tilde{B}_{21} = B_{21}^{(v)} {Q_1^{(v)}}^T$. After this transformation, the subdiagonal block B_{21} is filled completely. To preserve the banded structure of the matrix it is necessary to recover the zeros in the first column of B_{21} with a second length-b Householder transformation $Q_2^{(v)}$. The application of $Q_2^{(v)}$ in turn creates fill-in in B_{31}. This process is repeated until the end of the band is reached and no further fill-in is generated. More general, the chasing of fill-in can be expressed with the following equations:

$$\tilde{B}_{\beta\beta} = Q_\beta^{(v)} B_{\beta\beta}^{(v)} {Q_\beta^{(v)}}^T, 1 \leq \beta \leq d, \tag{2.23}$$

$$\tilde{B}_{\beta+1,\beta} = Q_{\beta+1}^{(v)} B_{\beta+1,\beta}^{(v)} {Q_\beta^{(v)}}^T, 1 \leq \beta < d, \tag{2.24}$$

where $\tilde{B} := B^{(v+1)}$. The index β of a transformation will be called sweep of the transformation throughout this thesis. In the next stage the partitioning of the matrix is shifted by one column/row.

Due to fill-in, the bandwidth of the matrix grows to two times the initial bandwidth b during the reduction to tridiagonal form. Note that the memory requirements are accordant. For the algorithms [64] and [65] the bandwidth grows only by one. The floating point operation count for the Householder based algorithm is the same as for the algorithms based on Givens rotations ($6n^2b$) if ordinary Givens rotations are used. In [68] the Householder based reduction has shown to outplay band reductions using Givens rotations [65, 69].

For the back transformation of eigenvectors all transformations have to be applied in reverse order to the eigenvector matrix. Just as the banded to full back transformation, the tridiagonal to banded back transformation requires $2kn^2$ flops. A detailed description thereof will be given in Ch. 3.

2.5.3 Successive band reduction

The 2-step band reduction has been generalized to a multi-step reduction in [70, 71]. An initial full or banded matrix is successively transformed to a matrix with smaller bandwidth, until tridiagonal form is reached. The reduction of a banded matrix to narrow banded form is similar to the algorithms presented in [67] and [68]. Except that, instead of simple Householder operations, blocked Householder operations can

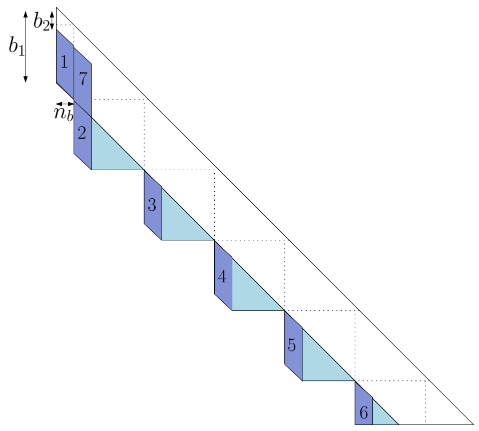


Figure 2.4: First stage of the reduction of a banded matrix with bandwidth b_1 to bandwidth b_2. The blue area represents the generated fill-in during the first stage. The area colored in dark blue is immediately removed with QR-decompositions. The numbers stand for a possible order of execution.

be used. Instead of *HouseGen* a QR-decomposition can be used. The left-sided, right-sided and symmetric application of a Householder transformation can be replaced by their blocked variants. In other words, [67] can be seen as a special case of the band to narrow band reduction. Figure 2.4 sketches the reduction of a banded matrix with bandwidth b_1 to a banded matrix with bandwidth b_2. Algorithm 17 shows the band to narrow band reduction as published in [71]. The blocking factor n_b of the algorithm

Algorithm 17 Band to narrow band reduction

```
 1: for $j = 1 \rightarrow n - b_2 - 1 : n_b$ do
 2:   $j_1 \leftarrow j; j_2 \leftarrow j_1 + n_b - 1; i_1 \leftarrow j + b_2; i_2 \leftarrow min(j + b_1 + n_b - 1, n)$
 3:   while $i_1 < n$ do
 4:     $V \leftarrow QR(B_{(i_1:i_2,j_1:j_2)})$
 5:     $T \leftarrow CWYGen(V)$
 6:     $CWYLeft(B_{(i_1:i_2,j_2+1:i_1-1)}, V, T)$
 7:     $CWYSymm(B_{(i_1:i_2,i_1:i_2)}, V, T)$
 8:     $CWYRight(B_{(i_2+1:min(i_2+b_1,n),i_1:i_2)}, V, T)$
 9:     $j_1 \leftarrow i_1; j_2 \leftarrow j_1 + n_b - 1; i_1 \leftarrow i_1 + b_1; i_2 \leftarrow min(i_2 + b_1, n)$
10:   end while
11: end for
```

has to be smaller or equal to b_2.

The presented algorithms for reducing (1) full matrices to tridiagonal form, (2) banded matrices to tridiagonal form, (3) full matrices to banded form and (4) banded matrices

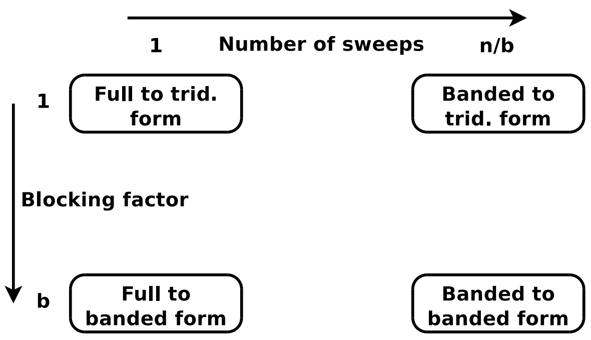


Figure 2.5: Different reduction algorithms for symmetric matrices. The reduction to banded form allows the blocking of operations. Starting from a banded matrix requires more than one sweep to remove intermediate fill-in.

to narrow banded form are very similar to each other. They can be seen as a more general or a more specialized representation of the same algorithm whereas (1) is the most specialized and (4) is the most general case (see Figure 2.5).

As the reduction from full to banded form, the band to band reduction allows a better cache reuse compared to its non-blocked alternatives. All but the last reduction step of the successive band reduction can be done using matrix matrix operations. Although, the multi-step approach requires much more Householder transformations ($\sum_i O(n^2/b_i)$ vs. $O(n)$) and much more elements have to be removed due to intermediate fill-in ($\sum_i O(n^2)$ vs. $O(n^2)$), this proceeding doesn't increase the asymptotic costs compared to the direct tridiagonalization. This is true because each transformation becomes less costly while the bandwidth of the matrix decreases. For the back transformation of eigenvectors, however, the transformations have to be applied to the, in general, full eigenvector matrix. This leads to arithmetic costs of $2kn^2\frac{d_i}{b_i}$ for each step i of the back transformation where d_i stands for the number of eliminated subdiagonals.

2.6 Eigensolvers in the context of HPC

Table 2.1 summarizes the costs for the three presented tridiagonalization approaches. We can see that the 2-step and multi-step approach have a much better cache behavior than the 1-step reduction. However, these algorithms have the drawback of the additional effort for the back transformation of eigenvectors. Issues regarding parallelizability will be addressed in the next section.

	Tridiagonalization		Back transformation	
	flops	words moved	flops	words moved
1-step	$4/3n^3$	$O(n^3)$	$2kn^2$	$O(\frac{n^3}{b})$
2-step	$4/3n^3$	$O(\frac{n^3}{b}+n^2b)$	$4kn^2$	$O(\frac{n^3}{b})$
m-step	$4/3n^3$	$O(\frac{n^3}{b})$	$2mkn^2$	$O(\frac{n^3}{b})$

Table 2.1: Complexity and cache efficiency of different tridiagonalization approaches. Lower order terms are omitted. "words moved" stands for the amount of data, which has to be loaded from slow memory.

Whether the use of a 1-step, 2-step or even multi-step approach is preferable cannot be answered in general, but should become more clearly during this thesis. Considering the trends in hardware architectures (e.g. memory wall), it is only a matter of time till the 2- or multi-step reduction outperform the direct tridiagonalization. We saw that the second step of the 2-step tridiagonalization is still memory-bounded. However, this step is much cheaper compared to the reduction to banded form. Moreover, the working set of the problem should be small enough to fit into the cache of massively parallel systems. In consideration of these facts, we rate the 2-step tridiagonalization to be the most promising approach for current and upcoming architectures.

Figure 2.6 emphasizes this statement by revealing two severe bottlenecks of the ScaLAPACK tridiagonalization routine PDSYTRD which uses a 1-step approach. We can see that the performance of the tridiagonalization is limited in two ways: absolute performance and scalability. The tridiagonalization step is much slower than the back transformation, although the latter requires 1.5 times more flops. Additionally, we can observe a poor strong scaling behavior such that for the examined problem size we get no additional speedup beyond 8 compute nodes (320 cores). We want to overcome these limitations by using the 2-step tridiagonalization approach.

In the next section we will present and analyze existing and newly developed parallel algorithms for the 2-step tridiagonalization of symmetric matrices with the corresponding back transformation of eigenvectors.

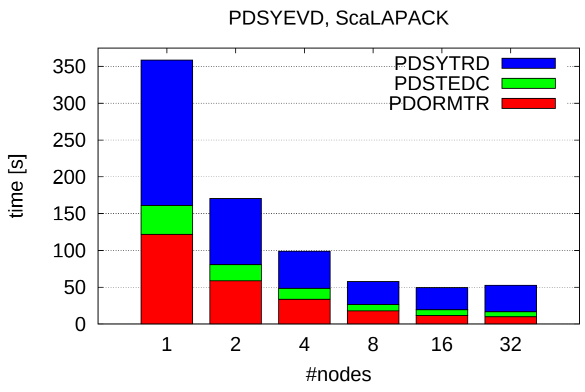


Figure 2.6: Runtime of the ScaLAPACK eigensolver PDSYEVD for symmetric matrices computing all eigenvectors of a matrix of size 20000. The routines PDSYTRD, PDSTEDC and PDORMTR represent the three stages of the eigensolver (tridiagonalization, tridiagonal eigensolver using D&C and back transformation). Measurements were performed on the SuperMIG system of the Leibniz Rechenzentrum (LRZ). 1 node represents four Intel Westmere-EX CPUs with 10 cores each.

3 The parallel 2-step tridiagonalization

In this chapter we present and analyze the algorithms of the ELPA library and, in particular, all aspects of parallelization. Regarding the contribution of the author the presented algorithms can be grouped in three classes: (1) Parallel algorithms which have been newly developed and implemented (blocked QR-decomposition, tridiagonal-to-banded back transformation), (2) parallel algorithms which rely on existing algorithms but have been improved regarding parallelism or cache behavior (tridiagonalization of banded matrices), (3) parallel algorithms which rely on existing algorithms and have been implemented by partners within the ELPA consortium (reduction to banded form using the default QR-decomposition, banded-to-full back transformation).

We will provide a detailed description of parallelization schemes for all parts of the tridiagonalization and back transformation. Furthermore, we will analyze the algorithms according to our model of parallel computation and motivate our design decisions. The model will be introduced in Sect. 3.1. In Sect. 3.2 through 3.6 we will present the parallel algorithms for the individual stages of the tridiagonalization and back transformation. In Sect. 3.2 we describe the reduction of dense symmetric matrices to banded form. The QR-decomposition is a crucial substep of the reduction to banded form for which we have developed fundamentally different parallelization schemes. Since this part is self-contained and very extensive, it has been transfered in its own section (Sect. 3.3). Afterwards in Sect. 3.4 we present the reduction of symmetric banded matrices to tridiagonal form using Householder transformations. In Sect. 3.5 and 3.6 we describe the back transformation of eigenvectors, corresponding to the two reduction steps. In Sect. 3.7 we analyze the algorithms towards weak and strong scaling behavior and, finally, in Sect. 3.8 we give an overview of existing implementations in other libraries.

3.1 Model of parallel computation

The model is not intended to precisely predict the runtime of an algorithm but to highlight the differences between different algorithmic variants. Under these requirements the runtime of the presented algorithms will be analyzed with a simple but effective model of parallel computation. The runtime estimation of network communication is based on a model which has been defined in [72] to model collective communication. It uses the alpha-beta or latency-bandwidth approach to model the cost of sending a message. I.e., under the assumption that no network conflict occurs, the sending of a message of n words is estimated with $\alpha + n\beta$, where α represents the message startup time and β stands for the transmission time per word. In the following we will replace α with t_{msg} and β with t_{word}. Furthermore, we make the following assumptions for network communication and parallel execution in general:

- A total of p processes is involved in the parallel execution of an algorithm. The processes are indexed from 0 to $p - 1$ and are organized in a two-dimensional Cartesian grid with p_r rows and p_c columns.

- A process can directly send a message to any other process through a two-dimensional bidirectional torus network where automatic routing is provided. A two-dimensional torus or mesh as network topology is the minimum requirement for an efficient execution of our algorithms.

- If a link in the network is occupied by two or more messages, a network conflict occurs and the network bandwidth is shared among the messages. This results in costs of $t_{\text{msg}} + k \cdot n \cdot t_{\text{word}}$ if k is the number of messages which have to be sent over the same link.

- Communication and computation cannot be overlapped. This means that at the sender side each communication is blocking. The effect of overlapping communication and computation is very different on various architectures and is thus difficult to model.

- Communication cannot be overlapped with communication. This means that only one message can be sent and received at a time. This assumption simplifies the runtime modelling drastically without provoking significant inaccuracies.

- Collective operations are assumed to take place in a one-dimensional sub-communicator (process row or process column) and are modeled with point-to-point communication. Using the most common algorithms [72] we get the runtime estimations in Table 3.1.

Operation	Runtime estimation
$Broadcast(n)$	$\lceil log(p) \rceil (t_{\text{msg}} + n \cdot t_{\text{word}})$
$Reduce(n)$	$\lceil log(p) \rceil (t_{\text{msg}} + n \cdot t_{\text{word}})$
$Allreduce(n)$	$2\lceil log(p) \rceil (t_{\text{msg}} + n \cdot t_{\text{word}})$

Table 3.1: Runtime estimation of the used collective operations for p processes and a message size of n words. We assume to have a one-dimensional bidirectional network topology. The computational effort for the reduction operations and lower order terms are omitted.

The cache hierarchy is modeled with the external memory model [73], making the following assumptions:

- The memory hierarchy consists of two levels. Fast and slow memory. The fast memory is capable to hold M words, the size of the slow memory is infinite.
- The fast memory is organized in M blocks with the size of one word each. It takes t_{mem} time to move a block from slow to fast memory.
- The execution of an arithmetic operation takes t_{flop} time and can only be done on data residing in fast memory.
- At the beginning of an algorithm all data resides in slow memory.

Altogether, our model consists of the four metrics t_{flop} (floating point performance), t_{mem} (memory bandwidth), t_{word} (network bandwidth), and t_{msg} (message startup time). All quantities are counted on the critical path of the parallel algorithm.

Outgoing from the described model we, first, estimate the runtime of the algorithmic kernels from Sect. 2.3.4. The results are shown in Table 3.2. Later on we will use these results for the analysis of the parallel algorithms.

3.2 Reduction to banded form using QR-decompositions

The sequential reduction to banded form is sketched in Algorithm 16 (Sect. 2.5.2). The algorithm consists of $\left\lceil \frac{n-b-1}{n_b} \right\rceil$ iterations, whereby in each iteration we compute the QR-decomposition of an n_b wide panel of the matrix and apply the resulting orthogonal transformations in a symmetric way to the rest of the matrix. For the sake

Operation	Runtime estimation
$HouseGen()$	$3n \cdot t_{\text{flop}} + 2n \cdot t_{\text{mem}}$
$HouseLeft/Right()$	$4nm \cdot t_{\text{flop}} + 2nm \cdot t_{\text{mem}}$
$HouseSymm()$	$4n^2 \cdot t_{\text{flop}} + n^2 \cdot t_{\text{mem}}$
$WYGen()$	$2nb^2 \cdot t_{\text{flop}} + nb^2 \cdot t_{\text{mem}}$
$WYLeft/Right()$	$4nmb \cdot t_{\text{flop}} + 4nm \cdot t_{\text{mem}}$
$WYSymm()$	$(4n^2b + 4nb^2) \cdot t_{\text{flop}} + 2n^2 \cdot t_{\text{mem}}$
$CWYGen()$	$(nb^2 + \frac{b^3}{3}) \cdot t_{\text{flop}} + 2nb \cdot t_{\text{mem}}$
$CWYLeft/Right()$	$(4nmb + nb^2) \cdot t_{\text{flop}} + 4nm \cdot t_{\text{mem}}$
$CWYSymm()$	$(4n^2b + 5nb^2 + b^3) \cdot t_{\text{flop}} + 2n^2 \cdot t_{\text{mem}}$

Table 3.2: Runtime of the algorithmic kernels according to the described model. The Householder transformations are of order n. The matrices, the Householder transformations are applied on, are of size $n \times m$, $m \times n$ and $n \times n$ for the left-sided, right-sided and symmetric application respectively. b is the blocking factor of the blocked Householder transformations. Furthermore, we assume that $n, m, b^2 < M < n^2, nm$. For the access to memory t_{mem} we omitted lower order terms if the operation is not memory-bounded. As memory-bounded we define operations where the ratio of floating point operations to the number of memory accesses $\frac{x_{\text{flop}}}{x_{\text{mem}}} \notin \Omega(b)$.

of simplicity we assume that the blocking factor n_b of the algorithm is equal to the bandwidth b of the banded matrix.

For a parallel execution of the reduction to banded form on a distributed memory system we have to consider some more issues. Such issues are, e.g., decisions on how to choose the parallel data layout, when and how to use collective communication and last but not least a bunch of intern blocking possibilities. To get a little bit of order into this jungle of options, we first partition our parallel reduction to banded form into a part for the QR-decomposition, a part for the symmetric update, and a part for the generation of the compact WY representation. Similarly we partition the runtime estimation:

$$t_{\text{fullbnd}} = t_{\text{qr}} + t_{\text{symm}} + t_{\text{tgen}}. \tag{3.1}$$

For the parallel QR-decomposition, as it occurs during the reduction to banded form, we have implemented different algorithms:

(i) a standard Householder QR-decomposition as it is defined in [74] and

(ii) a newly developed blocked Householder QR-decomposition [75].

We will present those algorithms for the parallel QR-decomposition in Sect. 3.3.1 and 3.3.2. In the following we will discuss all aspects of the symmetric update and the parallel data layout. For the symmetric update we have to use Householder transformations as they are generated in the QR-decomposition , i.e. for the decomposition of a matrix of size $n \times b$ we get b Householder vectors with descending size from n to $n - b + 1$.

For the parallel data distribution of the input matrix A we use a 2D block-cyclic data layout. This means that a matrix A is partitioned into blocks of size $blk_r \times blk_c$. For our algorithms we assume to have square blocks, i.e. $blk = blk_r = blk_c$. The blocks of the matrix, in turn, are distributed in a cyclic fashion over the 2D Cartesian grid of processes. In Figure 3.1 we see an example of a 2D block-cyclic distribution. Please note that for a symmetric matrix only one triangle is stored explicitly. We introduce the notation $A(p_r, p_c)$ to express that the matrix A is distributed in a 2D block-cyclic way across a process grid of p_r rows and p_c columns. Other matrices are distributed only in one dimension and are replicated in the other dimension. These matrices are usually tall and skinny, i.e. $n \gg m$ for a matrix of size $n \times m$, and are distributed in a block-cyclic fashion along the longer dimension of the matrix. $A(p_r, *)$, thus, denotes that the matrix A is distributed in a 1D block-cyclic fashion across one column of processes and is replicated on all other columns of processes. $A(*, p_c)$, in turn, means that a matrix A is distributed across one row of processes and is replicated on all other

	0	1	2	3	4
0	0,0	0,1	0,2	0,0	0,1
1	1,0	1,1	1,2	1,0	1,1
2	0,0	0,1	0,2	0,0	0,1
3	1,0	1,1	1,2	1,0	1,1
4	0,0	0,1	0,2	0,0	0,1

Figure 3.1: 2D block-cyclic distribution of a matrix with 5×5 blocks over a 2×3 grid of processes.

row communicators. Finally, $A(*,*)$ means that the matrix A is replicated on all $p_r \times p_c$ processes. $A(p_r)$ and $A(p_c)$ stand for a 1D block-cyclic distribution within one column and row communicator respectively with no replication in the second dimension. $A_{[i,j]}$ and $A_{[i]}$ stand for the local matrices on process $p_{i,j}$ and p_i within the 2D and 1D block-cyclic distribution respectively. Furthermore, we introduce $n_{\text{localr}} = \left\lceil \frac{n_l}{p_r \cdot blk} \right\rceil \cdot blk$ and $n_{\text{localc}} = \left\lceil \frac{n_l}{p_c \cdot blk} \right\rceil \cdot blk$ as the local matrix size in the row and in the column dimension.

Using this notation we will formulate parallel algorithms for the computation of *CWYGen* (Algorithm 18) and *CWYSymm* (Algorithm 19) as defined in Equation (2.18) and Algorithm 10 (Sect. 2.3.4). Afterwards we will derivate the runtime estimation for the symmetric update t_{symm} and for the generation of the matrix T t_{tgen}.

We consider the l-th iteration of Algorithm 16 and assume to get the matrix $Y(p_r,*)$ as input from a parallel QR-decomposition as it will be described in Sect. 3.3.1 and 3.3.2. Y is a matrix of size $n_l \times b$ with $n_l = n - l \cdot b - 1$, containing all the Householder vectors from the current QR-decomposition. The upper triangle of Y is filled with zero entries.

The computation of the matrix T is done redundantly on all p_c columns of processes. Algorithm 18 shows the parallel computation of T within one such process column with p_r processes. The τ values of the Householder transformations are stored in the diagonal entries of T. Due to the symmetry of Z only one triangle has to be computed. Accordingly, the allreduce operation is applied only on one triangle of Z. Using Table 3.1 and 3.2 we get the following runtime estimation for Algorithm 18:

$$t_{\text{cwygen}} \approx \left(b^2 n_{\text{local}} + \frac{b^3}{3} \right) t_{\text{flop}} + 2b n_{\text{localr}} t_{\text{mem}} + \lceil log(p_r) \rceil (b^2 t_{\text{word}} + 2t_{\text{msg}}). \qquad (3.2)$$

Algorithm 18 Parallel $CWYGen(Y(p_r,*),T(*,*)) \rightarrow T$

```
1: $Z(*,*)_i \leftarrow Y_{[i]}^T Y_{[i]}$
2: $Z \leftarrow Allreduce(Z_i)$
3: for $k = 2 \rightarrow b$ do
4:     $\tau \leftarrow T_{(k,k)}$
5:     $T_{(1:k-1,k)} \leftarrow -\tau T_{(1:k-1,1:k-1)} Z_{(1:k-1,k)}$
6: end for
```

Summing up all timings for the generation of T within Algorithm 16 leads to t_{tgen}:

$$\begin{aligned} t_{\text{tgen}} \lessapprox & \left(\frac{n^2 b}{2p_r} + \frac{nb^2}{3} + nb \cdot blk\right) t_{\text{flop}} \\ & + \left(\frac{n^2}{p_r} + 2n \cdot blk\right) t_{\text{mem}} \\ & + \lceil log(p_r) \rceil \left(nbt_{\text{word}} + \frac{2n}{b} t_{\text{msg}}\right). \end{aligned} \tag{3.3}$$

The parallel symmetric application of a compact WY transformation is shown in Algorithm 19. Please note that much of the algorithms complexity is hidden within the individual statements, which will be explained in the following. One important fact, for example, is that only one triangle of the symmetric matrix A is stored explicitly.

Algorithm 19 Parallel $CWYSymm(A(p_r,p_c),Y(p_r,*),T(*,*))$

```
1: $Y'(*,p_c)_{[j]} \leftarrow Transpose(Y_{[i]})$
2: $Z(p_r,*)_{[i]\,j} \leftarrow A_{[i,j]} Y'_{[j]}$
3: $Z'(*,p_c)_{[j]\,i}^T \leftarrow Y_{[i]}^T A_{[i,j]}$
4: $Z'_{[j]\,j} \leftarrow Reduce(Z'_{[j]\,i})$
5: if $(i == j)$ then
6:     $Z_{[i]\,j} \leftarrow Z_{[i]\,j} + Z'_{[i]\,j}$
7: end if
8: $Z_{[i]} \leftarrow Allreduce(Z_{[i]\,j})$
9: $Z_{[i]} \leftarrow Z_{[i]} T$
10: $X(*,*)_i \leftarrow Y_{[i]}^T Z_{[i]}$
11: $X_i \leftarrow T^T X_i$
12: $X \leftarrow Allreduce(X_i)$
13: $Z_{[i]} \leftarrow Z_{[i]} - 0.5 Y_{[i]} X$
14: $Z'_{[j]} \leftarrow Transpose(Z_{[i]})$
15: $A_{[i,j]} \leftarrow A_{[i,j]} - Y_{[i]} Z'^T_{[j]} - Z_{[i]} Y'^T_{[j]}$
```

Obviously, the other triangle of A is implicitly available due to the symmetry of A. For a block-cyclic distributed matrix, however, the local matrix blocks are, in general, not symmetric, i.e. $A_{[i,j]} \neq A_{[i,j]}^T$. Thus, the matrix matrix multiplication AY can be decomposed into $A_{[i,j]}Y_{[j]} + (Y_{[i]}^T A_{[i,j]})^T$ if only one triangle of $A_{[i,j]}$ is available. These two matrix matrix products are computed in line (2) and (3) of Algorithm 19.

To execute line (2) and (3), in turn, a process $p_{i,j}$ needs $Y_{[i]}$ and $Y_{[j]}$. $Y_{[i]}$ is input of the algorithm. $Y_{[j]}$ is made available through the $Transpose$ operation in line (1). $Transpose()$ is a complex communication routine which requires one broadcast operation within each column communicator if the grid of processes is a perfect square. For a non-square process grid the total number of broadcasts increases to lcm (= least common multiple of p_r and p_c) whereas always p_c of the broadcasts can be done in parallel (runtime estimation: $\lceil log(p_r) \rceil \left(n_{\text{localc}} b t_{\text{word}} + \frac{lcm}{p_c} t_{\text{msg}}\right)$).

The partial results of line (2) and (3) have to be summed up in line (4) through (8). In a first step the results $U'_{[j]\,i}$ are summed up to one single result $U'_{[j]\,j}$ which is then added to $U_{[i]\,j}$ on the root process of each reduction. Thereby, the communication pattern of the operation in line (4) is the same as for the transpose operation with the difference that the broadcasts are replaced by reductions (runtime estimation: $\lceil log(p_r) \rceil \left(n_{\text{localc}} b t_{\text{word}} + \frac{lcm}{p_c} t_{\text{msg}}\right)$). Finally, in line (8) an allreduce is called on each column communicator to make $U_{[i]}$ available on each process (runtime estimation: $2 \lceil log(p_c) \rceil \left(n_{\text{localr}} b t_{\text{word}} + t_{\text{msg}}\right)$).

The operations in line (9) through (15) correspond more or less to the operations of the sequential algorithm. The matrix matrix multiplication $U_{[i]}T$ in line (9) as well as $T^T Z_i$ in line (11) and $U_{[i]} - 0.5 Y_{[i]} Z$ in line (13) are entirely local. The partial results of $Y_{[i]}^T U_{[i]}$ in line (10) have to be reduced to the final matrix Z in line (12) (runtime estimation: $2 \lceil log(p_r) \rceil \left(b^2 t_{\text{word}} + t_{\text{msg}}\right)$). To finally compute the symmetric update of $A_{[i,j]}$ the matrix $U_{[i]}$ has to be transposed such that $U_{[i]}$ and $U_{[j]}$ are available on process $p_{i,j}$ (runtime estimation: $\lceil log(p_r) \rceil \left(n_{\text{localc}} b t_{\text{word}} + \frac{lcm}{p_c} t_{\text{msg}}\right)$).

The runtime estimation t_{cwysymm} of Algorithm 19 is shown in Equation (3.4):

$$\begin{aligned} t_{\text{cwysymm}} \approx\, & (4 b n_{\text{localr}} n_{\text{localc}} + 5 b^2 n_{\text{localr}} + b^3) t_{\text{flop}} \\ & + 2 n_{\text{localr}} n_{\text{localc}} t_{\text{mem}} \\ & + (2 \lceil log(p_c) \rceil n_{\text{localr}} b + \lceil log(p_r) \rceil (3 n_{\text{localc}} b + 2b^2)) t_{\text{word}} \\ & + \left(2 \lceil log(p_c) \rceil + \lceil log(p_r) \rceil \left(3 \frac{lcm}{p_c} + 2\right)\right) t_{\text{msg}}. \end{aligned} \tag{3.4}$$

The accumulated runtime of t_{cwysymm} over the whole reduction to banded form is shown in Equation (3.5):

$$\begin{aligned}
t_{\text{symm}} \lessapprox{} & \left(\frac{4n^3}{3p} + \frac{5n^2b}{2p_r} + \frac{2n^2 \cdot blk}{p_c} + \frac{2n^2 \cdot blk}{p_r} + nb^2 + 5nb \cdot blk + 4n \cdot blk^2\right) t_{\text{flop}} \\
& + \left(\frac{2n^3}{3pb} + \frac{n^2 \cdot blk}{p_c b} + \frac{n^2 \cdot blk}{p_r b} + \frac{2n \cdot blk^2}{b}\right) t_{\text{mem}} \\
& + \left(\lceil log(p_c)\rceil \frac{n^2}{p_r} + \lceil log(p_r)\rceil \left(\frac{3n^2}{p_c} + 2nb\right)\right) t_{\text{word}} \\
& + \frac{n}{b}\left(2\lceil log(p_c)\rceil + \lceil log(p_r)\rceil \left(3\frac{lcm}{p_c} + 2\right)\right) t_{\text{msg}}. \qquad (3.5)
\end{aligned}$$

In Sect. 3.7 we will simplify the derived runtime estimations for the analysis of the asymptotic behavior during weak and strong scaling.

3.3 QR-decomposition: algorithmic variants

The QR-decomposition is a crucial substep within the reduction to banded form. Especially if high scalability is desired, the QR-decomposition will become the bottleneck of the whole reduction to banded form. This is the case since the QR-decomposition requires an asymptotically higher number of synchronization points, compared to the rest of the algorithm. In this section we give an overview of the faced problem and describe the two algorithmic variants which have been implemented for the ELPA library. The second of this approaches has been developed during this thesis to tackle the scalability problems of the QR-decomposition.

An algorithm for the QR-decomposition of a matrix has already been presented in Sect. 2.4. The task is to decompose a matrix $A \in \mathbb{R}^{n\times m}$ into a product of matrices $Q \in \mathbb{R}^{n\times m}$ and $R \in \mathbb{R}^{m\times m}$, where Q is orthogonal and R is upper triangular. The QR-decomposition is defined for any rectangular matrix $A \in \mathbb{R}^{n\times m}$ with $n \geq m$. In the context of the reduction to banded form we are interested in the decomposition of so called tall and skinny matrices, i.e. $n \gg m$, which represent small panels of the original symmetric matrix. Of course, mathematically, there is no difference between the QR-decomposition of tall and skinny matrices and general rectangular matrices. From the algorithmic point of view, however, it makes sense to distinguish between these two cases. The decomposition of general rectangular matrices, usually, is composed of QR-factorizations of tall and skinny panels of a matrix ($A_i \to Q_iR_i$) followed by an update of the trailing matrix ($A \leftarrow Q_i^T A$) until A has upper triangular form. In Algorithm 12 (Sect. 2.4) we saw an example for such a proceeding if the QR-decomposition is based on Householder transformations.

We are only interested in the factorization of tall and skinny matrices. In the literature we can find several methods to achieve this task. The most common proceeding is a QR-decomposition based on Householder transformations as it has been shown in Algorithm 11 (Sect. 2.4). QR-decompositions based on Householder transformations are numerically stable and are used in most (parallel) linear algebra libraries [37, 38, 39, 40]. The same principle of introducing zeros into A can be achieved with Givens rotations. Such algorithms have comparable numerical properties [43]. Another class of algorithms for the QR-decompositions are methods based on Gram-Schmidt orthogonalization [76]. However, those methods don't have advantages compared to the classic parallel Householder QR-decomposition (neither regarding parallelization nor regarding numerical stability) [77] and will thus be ignored in the following. In [77] the TSQR (Tall Skinny QR) algorithm has been presented. TSQR is organized as a reduction operation with the local QR-decomposition as reduction operation. In a first step, each process computes a QR-decomposition on its local data. Afterwards, the resulting triangular matrices are successively reduced, e.g. using a binary tree, to one single triangular matrix. It can be shown that TSQR is optimal regarding communication. However the method has significant computational overheads during the algorithm itself, as well as for the symmetric update of the trailing matrix if TSQR is used within the reduction to banded form. Finally, there exists an algorithm based on the Cholesky factorization [76]. CholeskyQR computes R with the Cholesky decomposition of A^TA since $A^TA = R^TQ^TQR = R^TR$. Q can then be computed with $Q = AR^{-1}$. CholeskyQR is suited very well for parallel computation since it requires only one synchronization point. Moreover the computations are mostly based on BLAS 3 operations. However, CholeskyQR is not numerically stable if A is ill conditioned [78].

In the following sections we will present two different parallel algorithms for the QR-decomposition of tall and skinny matrices which are all based on Householder transformations. The classic Householder QR-decomposition (Sect. 3.3.1) is the by far most prevalent algorithm and can be found in most linear algebra libraries. In Sect. 3.3.2 we will give a detailed derivation of our newly developed blocked QR-decomposition. The blocked QR-decomposition is an algorithm which generates and applies more Householder transformations at once and should, thus, be better suited for parallel execution. Under certain circumstances the algorithm runs into numerical problems. However, these problems can be recognized and avoided very cheaply.

3.3.1 Classic Householder QR-decomposition

Algorithm 20 shows the parallel version of the classic Householder QR-decomposition. The sequential version has been presented in Algorithm 11. The algorithm requires

Algorithm 20 Parallel Householder QR-decomposition$(A(p_r, p_c)) \rightarrow Y(p_r, *), \tau(*, *)$

```
1: for $k = 1 \rightarrow m - 1$ do
2:     if $(A_{(:,k)}$ in $A_{[i,j]})$ then
3:         $Y(p_r)_{(k:n,k)\,[i]}, \tau_k(*) \leftarrow parallelHouseGen(A_{(k:n,k)\,[i]})$
4:     end if
5:     $Y(p_r, *)_{(k:n,k)\,[i]}, \tau_k(*, *) \leftarrow Broadcast(Y_{(k:n,k)\,[i]}, \tau_k)$
6:     $parallelHouseLeft(A_{[i,j]}, Y_{(k:n,k)\,[i]}, \tau_k)$
7: end for
```

a parallel version of *HouseGen* and *HouseLeft* which are shown in Algorithm 21 and 22. For each column of the matrix *parallelHouseGen* is called by the process column, owning this matrix column. Afterwards, the resulting Householder vector is broadcasted to all other column communicators. Finally, *parallelHouseLeft* is called to update the remaining columns of the matrix.

The parallelization of *HouseGen* and *HouseLeft* is straightforward. Each matrix operation is parallelized according to the parallel data layout. Thereby, the computation of the dot product in line (1) of Algorithm 21 and the matrix vector product in line (1) of Algorithm 22 require a reduction operation to combine the partial results to the final result.

The runtime estimations of Algorithm 21 and 22 follow directly from Table 3.1 and 3.2:

$$t_{\text{pHouseGen}} \approx 3n_{\text{localr}}t_{\text{flop}} + 2n_{\text{localr}}t_{\text{mem}} + 2\lceil log(p_r)\rceil(2t_{\text{word}} + t_{\text{msg}}), \tag{3.6}$$

$$\begin{aligned} t_{\text{pHouseLeft}} \approx\ & 4n_{\text{localr}}m_{\text{localc}}t_{\text{flop}} + 2n_{\text{localr}}m_{\text{localc}}t_{\text{mem}} \\ & + 2\lceil log(p_r)\rceil(m_{\text{localc}}t_{\text{word}} + t_{\text{msg}}), \end{aligned} \tag{3.7}$$

where $n \times m$ is the size of the matrix A.

For the runtime estimation of the whole QR-decomposition we, firstly, make the assumption that A is tall and skinny and, thus,

$$n_{\text{localr}} = \left\lceil \frac{n-k}{p_r \cdot blk} \right\rceil \cdot blk \tag{3.8}$$

Algorithm 21 $parallelHouseGen(x(p_r)) \to v(p_r), \tau(*)$

```
1: $d(*)_i \leftarrow x_{[i]}^T \cdot x_{[i]}$
2: $a(*)_i \leftarrow 0$
3: **if** ($x_{(1)}$ in $x_{[i]}$) **then**
4:     $a_i \leftarrow x_{(1)}$
5: **end if**
6: $d, a \leftarrow Allreduce(d_i, a_i)$
7: $\beta \leftarrow \sqrt{d} \cdot \mathrm{sign}(a)$
8: $\tau \leftarrow \frac{a+\beta}{\beta}$
9: $v_{[i]} \leftarrow \frac{1}{a+\beta} x_{[i]}$
10: $x_{[i]} \leftarrow 0$
11: **if** ($x_{(1)}$ in $x_{[i]}$) **then**
12:     $v_{(1)} \leftarrow 1$
13:     $x_{(1)} \leftarrow -\beta$
14: **end if**
```

Algorithm 22 $parallelHouseLeft(A(p_r, p_c), v(p_r, *), \tau(*, *))$

```
1: $z(*, p_c)_i^T \leftarrow \tau v_{[i]}^T A_{[i,j]}$
2: $z_{[j]} \leftarrow Allreduce(z_{[j]\,i})$
3: $A_{[i,j]} \leftarrow A_{[i,j]} - v_{[i]} z_{[j]}^T$
```

is approximated to be constant for all k. Secondly, we make the approximation that

$$m_{\text{localc}} = \left\lceil \frac{m-k}{p_c \cdot blk} \right\rceil \cdot blk = blk \tag{3.9}$$

for all iterations of the algorithm. Considering also the costs of the broadcasts, this leads to Equation (3.10) for the runtime of the parallel Householder QR-decomposition of a tall and skinny matrix:

$$\begin{aligned} t_{\text{parallelQR}} \lessapprox\ & 4n_{\text{localr}} b \cdot blk \cdot t_{\text{flop}} + 2n_{\text{localr}} b \cdot blk \cdot t_{\text{mem}} \\ & + b\lceil log(p_r)\rceil((2blk + n_{\text{localr}})t_{\text{word}} + 5t_{\text{msg}}). \end{aligned} \tag{3.10}$$

t_{qr} shows the accumulated runtime of Algorithm 20 over the entire reduction to banded form:

$$\begin{aligned} t_{\text{qr}} \lessapprox & \left(\frac{2n^2 \cdot blk}{p_r} + 2n \cdot blk^2\right) t_{\text{flop}} + \left(\frac{n^2 \cdot blk}{p_r} + n \cdot blk^2\right) t_{\text{mem}} \\ + & \left(\frac{n^2}{2p_r} + 2n \cdot blk\right) \lceil log(p_r)\rceil t_{\text{word}} + 5n\lceil log(p_r)\rceil t_{\text{msg}}. \end{aligned} \tag{3.11}$$

Please note that the number of required messages for the classic Householder QR-decomposition dominates the whole reduction to banded form ($O(n)$ vs. $O(n/b)$). The time for memory access t_{mem} may become important if high scalability is desired. The terms for t_{flop} and t_{word} are not critical.

3.3.2 Blocked QR-decomposition

The aim of the blocked QR-decomposition was to make the reduction to banded form less dependent on network latency and memory bandwidth requirements and, thus, to improve the scalability of our algorithms. The basic idea is to generate and apply more than one Householder transformation with a single communication operation. In a first step we will derive an algorithm which computes two Householder transformations at once. Afterwards we will generalize this concept to arbitrary blocking. We will analyze both algorithms regarding numerical stability.

It turned out that the derived algorithms share many commonalities with CholeskyQR. However, our approach is extended by a concept of adaptive blocking to guarantee numerical stability. Furthermore we generate Householder transformations instead of an orthogonal matrix Q such that the QR-decomposition is easily integrable in the existing reduction to banded form. In the following we sketch the original derivation of the algorithms which can also be found in [75] and [79].

At first we introduce the notation to describe the following algorithms. Contrary to the preceding algorithms, we resign to express the parallelization explicitly. This should improve the readability of the algorithms. The concept of parallelization is the same as for the classic Householder QR-decomposition, except that more than one matrix column is handled at once. Furthermore, we define the matrix $A \in \mathbb{R}^{n\times m}$ as A_0. A_k stands for the content of the matrix after applying k Householder transformations. $Y \in \mathbb{R}^{n\times k}$ contains all generated Householder vectors.

Rank-2 Householder QR-decomposition

Starting from the classic Householder QR-decomposition, we derive an algorithm which transforms two columns of the matrix A at once. The outcome, however, should be the same as after applying two iterations of Algorithm 20.

The first Householder vector can be computed as usual with a call of *HouseGen*:

$$\begin{aligned}
\beta_1 =& \sqrt{A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,1)}} \cdot \operatorname{sign}(A_{0\,(1,1)}) \\
\tau_1 =& \frac{A_{0\,(1,1)} + \beta_1}{\beta_1} \\
Y_{(:,1)} =& \left(1, \frac{A_{0\,(2:n,1)}}{A_{0\,(1,1)} + \beta_1}\right)^T \\
R_{(:,1)} =& \left(-\beta_1, 0_{(2:n)}\right)^T
\end{aligned}$$

For the second Householder vector $Y_{(:,2)}$ we try to find a formulation which uses only data from A_0 instead of A_1.

At first we look at the left-sided application of the first Householder transformation on the remaining columns of A_0. Using Algorithm 2 (*HouseLeft*) each column j of A_0 will be updated in the following way:

$$\begin{aligned}
z_{1\,(j)} &= \tau_1 \cdot Y^T_{(:,1)} \cdot A_{0\,(:,j)} \\
A_{1\,(:,j)} &= A_{0\,(:,j)} - z_{1\,(j)} \cdot Y_{(:,1)}
\end{aligned}$$

Moreover, $z_{1\,(j)}$ can be expressed without the knowledge of $Y_{(:,1)}$ using solely the original matrix A_0 , τ_1 and β_1:

$$\begin{aligned}
z_{1\,(j)} &= \tau_1 \cdot (1, \frac{A_{0\,(2:n,1)}}{A_{0\,(1,1)} + \beta_1})^T \cdot A_{0\,(1:n,j)} \\
&= \tau_1 \cdot (A_{0\,(1,j)} + \frac{A^T_{0\,(2:n,1)} \cdot A_{0\,(2:n,j)}}{A_{0\,(1,1)} + \beta_1}) \\
&= \tau_1 \cdot \frac{A_{0\,(1,j)} \cdot \beta_1 + A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,j)}}{A_{0\,(1,1)} + \beta_1}
\end{aligned}$$

Additionally, we are able to replace τ_1 by its computational formula and get

$$\begin{aligned}
z_{1\,(j)} &= \frac{A_{0\,(1,1)} + \beta_1}{\beta_1} \cdot \frac{A_{0\,(1,j)} \cdot \beta_1 + A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,j)}}{A_{0\,(1,1)} + \beta_1} \\
&= \frac{A_{0\,(1,j)} \cdot \beta_1 + A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,j)}}{\beta_1} \\
&= A_{0\,(1,j)} + \frac{A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,j)}}{\beta_1}. \qquad (3.12)
\end{aligned}$$

To calculate the second Householder vector we have to update the second column of A_0 using the *HouseLeft* algorithm. Instead of using the first Householder vector directly, it is possible to replace it by the contents of A_0:

$$A_{1\,(1,2)} = A_{0\,(1,2)} - z_{1\,(2)} \qquad (3.13a)$$

$$A_{1\,(2:n,2)} = A_{0\,(2:n,2)} - z_{1\,(2)} \cdot \frac{A_{0\,(2:n,1)}}{A_{0\,(1,1)} + \beta_1} \qquad (3.13b)$$

This result can now be used to generate β_2 by applying *HouseGen* on $A_{1\,(2:n,2)}$:

$$\beta_2 = \sqrt{A^T_{1\,(2:n,2)} \cdot A_{1\,(2:n,2)}} \cdot \text{sign}(A_{1\,(2,2)})$$

Instead of computing β_2 out of the column $A_{1\,(:,2)}$, it is possible to generate the dot product $A^T_{1\,(2:n,2)} \cdot A_{1\,(2:n,2)}$ out of the contents of A_0. Therefor, we use some properties of the Householder transformation.

Without loss of generality we define the first and the second column of A_0 as

$$a_1 = A_{0\,(1:n,1)} = \alpha_{1,1} u_1 \qquad (3.14a)$$

$$a_2 = A_{0\,(1:n,2)} = \alpha_{1,2} u_1 + \alpha_{2,2} u_2, \qquad (3.14b)$$

where u_1 and u_2 are normalized and mutually orthogonal, i.e. $u_1^T u_2 = 0$. Let H_1 be the Householder matrix which zeros all but the first element of a_1. Obviously,

$$H_1 a_1 = A_{1\,(1:n,1)} = \pm\alpha_{1,1} e_1.$$

Moreover, it can easily be shown that

$$(H_1 a_2)_{(1)} = A_{1\,(1,2)} = \pm\alpha_{1,2}.$$

Since a Householder transformation doesn't change the length of a vector, we further know that

$$A^T_{1\,(2:n,2)} \cdot A_{1\,(2:n,2)} = A^T_{0\,(1:n,2)} \cdot A_{0\,(1:n,2)} - A^2_{1\,(1,2)} = \alpha^2_{2,2}. \tag{3.15}$$

The coefficients $\alpha^2_{1,1}$, $\alpha^2_{1,2}$ and $\alpha^2_{2,2}$ in turn can be computed using the dot products $a_1^T a_1$, $a_1^T a_2$ and $a_2^T a_2$:

$$\begin{aligned}
\alpha^2_{1,1} &= a_1^T a_1 \\
\alpha^2_{1,2} &= \frac{(a_1^T a_2)^2}{\alpha^2_{1,1}} \\
\alpha^2_{2,2} &= a_2^T a_2 - \alpha^2_{1,2}
\end{aligned}$$

Combined with all previous equations, this results offer the possibility to generate β_1 and β_2 on the fly without updating the whole matrix. Only the three dot products $A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,1)}$, $A^T_{0\,(1:n,1)} \cdot A_{0\,(1:n,2)}$ and $A^T_{0\,(1:n,2)} \cdot A_{0\,(1:n,2)}$ as well as $A_{0\,(1,1)}$, $A_{0\,(1,2)}$, and $A_{0\,(2,2)}$ are needed to calculate the scalars β and τ.

After computing these scalars, the Householder vectors Y and the transformed matrix R can be computed in the following way using *HouseGen* as well as Equation (3.13a) and (3.13b):

$$\begin{aligned}
Y_{(:,1)} &= \left(1, \frac{A_{0\,(2:n,1)}}{A_{0\,(1,1)} + \beta_1}\right)^T \\
R_{(:,1)} &= \left(-\beta_1, 0_{(2:n)}\right)^T \\
Y_{(:,2)} &= \left(0, 1, \frac{A_{1\,(3:n,2)}}{A_{1\,(2,2)} + \beta_2}\right)^T \\
R_{(:,2)} &= \left(A_{1\,(1,2)}, -\beta_2, 0_{(3:n)}\right)^T
\end{aligned}$$

We can now formulate a rank-2 version of the *HouseGen*-algorithm which computes two Householder vectors at once (see Algorithm 23).

Algorithm 23 HouseGen, rank-2 version

```
 1: $a_{11} \leftarrow A_{(1,1)}, \quad a_{12} \leftarrow A_{(1,2)}, \quad a_{22} \leftarrow A_{(2,2)}, \quad a_{21} \leftarrow A_{(2,1)}$
 2: $d_{11} \leftarrow A^T_{(1:n,1)} A_{(1:n,1)}, \quad d_{12} \leftarrow A^T_{(1:n,1)} A_{(1:n,2)}, \quad d_{22} \leftarrow A^T_{(1:n,2)} A_{(1:n,2)}$
 3: $\beta_1 \leftarrow \sqrt{d_{11}} \cdot \mathrm{sign}(a_{11})$
 4: $\rho \leftarrow \frac{a_{12} + \frac{d_{12}}{\beta_1}}{a_{11} + \beta_1}$
 5: $d_{22} \leftarrow d_{22} - \frac{d_{12}^2}{d_{11}}$
 6: $a_{22} \leftarrow a_{22} - \rho a_{21}$
 7: $\beta_2 \leftarrow \sqrt{d_{22}} \cdot \mathrm{sign}(a_{22})$
 8: $\tau_1 \leftarrow T_{(1,1)} \leftarrow \frac{a_{11} + \beta_1}{\beta_1}$
 9: $\tau_2 \leftarrow T_{(2,2)} \leftarrow \frac{a_{22} + \beta_2}{\beta_2}$
10: $R \leftarrow 0_{(1:n,1:2)}$
11: $R_{(1,1)} \leftarrow -\beta_1$
12: $R_{(1,2)} \leftarrow \frac{d_{12}}{\beta_1}$
13: $R_{(2,2)} \leftarrow -\beta_2$
14: $y_1 = \left(1, \frac{A_{(2:n,1)}}{a_{11} + \beta_1}\right)^T$
15: $y_2 = \left(0, 1, \frac{A_{(3:n,2)} - \rho A_{(3:n,1)}}{a_{22} + \beta_2}\right)^T$
16: $Y = [y_1, y_2]$
17: $T_{(1,2)} \leftarrow \frac{a_{21}}{\beta_1} - \frac{a_{12} - \frac{d_{12}}{d_{11}} a_{11}}{\beta_2}$
```

After the generation, the two Householder transformations have to be applied to the rest of the matrix. This can be done with blocked Householder transformations. If we use compact WY transformations, T has the following form:

$$T = \begin{bmatrix} \tau_1 & \tau_1 \tau_2 y_1^T y_2 \\ 0 & \tau_2 \end{bmatrix}$$

Thereby, $y_1^T y_2$ can be calculated out of the already computed dot products. In the following we will derive a formula to compute $\tau_1 \tau_2 y_1^T y_2$ out of the previous results. At first we define

$$\begin{aligned} s_1 &= \operatorname{sign}(A_{0\,(1,1)}) \text{ and} \\ s_2 &= \operatorname{sign}(A_{1\,(2,2)}). \end{aligned}$$

The Householder vectors y_1 and y_2 are computed as defined in Algorithm 23 and can be written as

$$\begin{aligned} y_1 &= \frac{1}{s_1 + u_{1\,(1)}} \cdot (u_1 + s_1 e_1), \quad y_{2\,(1)} = 0, \quad \text{and} \\ y_{2\,(2:n)} &= \frac{1}{s_2 + u_{2\,(2)} - \frac{u_{2\,(1)} u_{1\,(2)}}{s_1 + u_{1\,(1)}}} \cdot \left(u_{2\,(2:n)} - \frac{u_{2\,(1)}}{s_1 + u_{1\,(1)}} u_{1\,(2:n)} + s_2 e_{2\,(2:n)} \right). \end{aligned}$$

Finally, out of y_1 and y_2 we can derive formulas for τ_1 and τ_2

$$\begin{aligned} \tau_1 &= \frac{2}{y_1^T y_1} = 1 + s_1 u_{1\,(1)} \\ \tau_2 &= \frac{2}{y_2^T y_2} = 1 + s_2 u_{2\,(2)} - s_2 \frac{u_{2\,(1)} u_{1\,(2)}}{s_1 + u_{1\,(1)}}, \end{aligned}$$

as well as for the dot product $y_1^T y_2$

$$\begin{aligned} y_1^T y_2 =& \frac{1}{s_1 + u_{1\,(1)}} \cdot \frac{1}{s_2 + u_{2\,(2)} - \frac{u_{2\,(1)} u_{1\,(2)}}{s_1 + u_{1\,(1)}}} \\ & \cdot \left(-u_{1\,(1)} u_{2\,(1)} - \frac{u_{2\,(1)} (1 - u_{1\,(1)}^2)}{s_1 + u_{1\,(1)}} + s_2 u_{1\,(2)} \right) \\ =& \frac{1}{\tau_1 \tau_2} (s_1 u_{1\,(2)} - s_2 u_{2\,(1)}). \end{aligned} \tag{3.16}$$

$u_{1\,(2)}$ and $u_{2\,(1)}$, in turn, can be computed out of the scalars $\alpha_{1,1}$, $\alpha_{1,2}$, and $\alpha_{2,2}$ as well as $a_{1\,(1)}$, $a_{1\,(2)}$, and $a_{2\,(1)}$:

$$
\begin{aligned}
u_{1\,(1)} &= \frac{a_{1\,(1)}}{\alpha_{1,1}} \\
u_{1\,(2)} &= \frac{a_{1\,(2)}}{\alpha_{1,1}} \\
u_{2\,(1)} &= \frac{a_{2\,(1)} - \alpha_{1,2} u_{1\,(1)}}{\alpha_{2,2}}
\end{aligned}
$$

The computation of the matrix T can be done within Algorithm 23 since all required values are already available.

Out of Algorithm 23 and the blocked left-sided application we can now formulate the rank-2 QR-decomposition (see Algorithm 24). For uneven matrix widths m we have

Algorithm 24 QR-decomposition, rank-2 version

```
1: for $block\_col = 1 \to m/2$ do
2:     $col \leftarrow 2 * block\_col - 1$
3:     $T, Y, R \leftarrow HouseGen2(A_{(:,col:col+1)})$
4:     $A_{(:,col+2:m)} \leftarrow CWYHouseLeft(T, Y, A_{(:,col+2:m)})$
5:     $A_{(:,col:col+1)} \leftarrow R$
6: end for
```

to perform one iteration with the classic Householder QR-decomposition. Please note that after line (3) of Algorithm 24 the Householder transformations (T and Y) have to be broadcasted to the remaining columns of processes.

Accuracy analysis Under certain conditions the presented algorithm runs into numerical problems. In the following we will analyze the relative error of the rank-2 QR-decomposition and derive a criterion for the stability of the algorithm. Again, we assume to have two vectors a_1 and a_2 as defined in Equation (3.14a) and (3.14b).

According to Algorithm 1 and 23 the Householder vectors are defined as

$$
\begin{aligned}
y_1 =& \frac{a_1 + \|a_1\| e_1}{a_{1\,(1)} + \|a_1\|}, \\
y_2 =& \frac{\hat{a}_{2\,(2:n)} + \|\hat{a}_{2\,(2:n)}\| e_1}{\hat{a}_{2\,(2)} + \|\hat{a}_{2\,(2:n)}\|}, \text{ with} \\
\hat{a}_{2\,(2:n)} =& a_{2\,(2:n)} - \rho a_{1\,(2:n)}, \\
\rho =& \frac{a_{2\,(1)} + \frac{a_1^T a_2}{\|a_1\|}}{a_{1\,(1)} + \|a_1\|}.
\end{aligned}
\tag{3.17}
$$

The blocked and non-blocked variants of *HouseGen* only differ in how to compute $\|\hat{a}_{2\,(2:n)}\|$ which is $\alpha_{2,2}$ in exact numerics. For the non-blocked variant we first do the componentwise subtraction in Equation (3.17) and build the norm afterwards. This leads to a relative error of $O(n\epsilon)$. For the rank-2 variant we first build the dot products $a_1^T a_1$, $a_1^T a_2$ and $a_2^T a_2$. After this we compute

$$\begin{aligned}\|\hat{a}_{2\,(2:n)}\|^2 =& (a_2^T a_2)(1+n\epsilon_1) - \frac{(a_1^T a_2 + \|a_1\|\|a_2\|n\epsilon_2)^2}{a_1^T a_1(1+n\epsilon_3)}\\ =&\alpha_{2,2}^2 + O((\alpha_{1,2}^2+\alpha_{2,2}^2)n\epsilon),\end{aligned}$$

and, thus, we get a relative error of $O\left(\left(1+\frac{\alpha_{1,2}^2}{\alpha_{2,2}^2}\right)n\epsilon\right)$.

To guarantee similar accuracy compared to the non-blocked QR-decomposition, we have to limit the term $\frac{\alpha_{1,2}^2}{\alpha_{2,2}^2}$. This leads to the following criterion:

$$\frac{(a_1^T a_2)^2}{a_1^T a_1 \cdot a_2^T a_2} \leq \frac{\epsilon_{\text{fallback}}}{1+\epsilon_{\text{fallback}}} \tag{3.18}$$

which corresponds to

$$\frac{\alpha_{1,2}^2}{\alpha_{2,2}^2} \leq \epsilon_{\text{fallback}}.$$

If Equation (3.18) doesn't hold, we switch back to the classic non-blocked approach. We set $\epsilon_{\text{fallback}}$ to 1 to avoid any substantial accuracy losses.

Runtime estimation For the runtime estimation of the parallel rank-2 QR-decomposition we ignore any issues regarding accuracy and assume that the fallback to the unblocked algorithm doesn't occur. Results on how often such fallbacks appear and how they affect performance can be found in Ch. 4.

Compared to the classic Householder QR-decomposition, the rank-2 variant halves the number of messages to be sent and improves the cache efficiency such that half as many words have to be read from slow memory. Due to the additional computation of some dot products, the blocked QR-decomposition requires a higher number of flops for the generation of Householder vectors (see $t_{\text{pHouseGen\_2}}$).

$$t_{\text{pHouseGen\_2}} \approx 10 n_{\text{localr}} t_{\text{flop}} + 4 n_{\text{localr}} t_{\text{mem}} + 2\lceil log(p_r)\rceil (7 t_{\text{word}} + t_{\text{msg}}) \tag{3.19}$$

$t_{\text{pHouseLeft\_2}}$ shows the estimated runtime for the left-sided application of two transformations using the compact WY technique:

$$\begin{aligned} t_{\text{pHouseLeft\_2}} &\approx 8n_{\text{localr}}m_{\text{localc}}t_{\text{flop}} + 2n_{\text{localr}}m_{\text{localc}}t_{\text{mem}} \\ &\quad + 2\lceil log(p_r)\rceil(2m_{\text{localc}}t_{\text{word}} + t_{\text{msg}}) \end{aligned} \tag{3.20}$$

The increased lower order terms within $t_{\text{pHouseGen\_2}}$ are ignored for the runtime estimation of the QR-decomposition of a tall and skinny matrix ($t_{\text{parallelQR\_2}}$) and the runtime estimation of all QR-decompositions during the reduction to banded form ($t_{\text{qr\_2}}$). Considering the costs of broadcasting Householder transformations to the remaining process columns and using the approximations from Equation (3.8) and (3.9) we get

$$\begin{aligned} t_{\text{parallelQR\_2}} &\lessapprox 4n_{\text{localr}}b \cdot blk \cdot t_{\text{flop}} + n_{\text{localr}}b \cdot blk \cdot t_{\text{mem}} \\ &\quad + b\lceil log(p_r)\rceil \left((2 \cdot blk + n_{\text{localr}})t_{\text{word}} + \frac{5}{2}t_{\text{msg}}\right), \end{aligned} \tag{3.21}$$

and

$$\begin{aligned} t_{\text{qr\_2}} &\lessapprox \left(\frac{2n^2}{p_r} \cdot blk + 2n \cdot blk^2\right) t_{\text{flop}} + \frac{1}{2}\left(\frac{n^2}{p_r} \cdot blk + n \cdot blk^2\right) t_{\text{mem}} \\ &\quad + \left(\frac{n^2}{2p_r} + 2n \cdot blk\right) \lceil log(p_r)\rceil t_{\text{word}} + \frac{5}{2}n\lceil log(p_r)\rceil t_{\text{msg}}. \end{aligned} \tag{3.22}$$

The rank-2 QR-decomposition tackles those issues which limit the scalability of the classic Householder QR-decomposition (latency and memory access). At next we will extend the algorithm to arbitrary blocking to reduce those terms even further.

Rank-k Householder QR-decomposition

Again, w.l.o.g., we assume to start with a matrix $A_0 \in \mathbb{R}^{n\times m}$ where the first k columns have the following form:

$$A_{0\,(:,1:k)} = (a_1, a_2, \cdots, a_k) = (u_1, u_2, \cdots, u_k) \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,k} \\ & \alpha_{2,2} & \cdots & \alpha_{2,k} \\ & & & \cdots \\ & & & \alpha_{k,k} \end{pmatrix} \tag{3.23}$$

u_i and u_j are normalized and mutually orthogonal for $i \neq j$. Out of these vectors we want to compute the set of Householder vectors $y_1, \ldots, y_k$ and the appropriate scalars $\tau_1, \ldots, \tau_k$ and $\beta_1, \ldots, \beta_k$ as well as the content of R.

We start with the definition of the Householder vectors y_i

$$\begin{aligned} y_{i\,(1:i-1)} &= 0_{(1:i-1)}, \quad y_{i\,(i)} = 1, \\ y_{i\,(i+1:n)} &= \frac{A_{i-1\,(i+1:n,i)}}{A_{i-1\,(i,i)} + \beta_i}, \end{aligned} \tag{3.24}$$

and the scalars τ_i and β_i from Algorithm 1

$$\begin{aligned} \beta_i =& \|A_{i-1\,(i:n,i)}\| \cdot \operatorname{sign}(A_{i-1\,(i,i)}) \\ \tau_i =& \frac{A_{i-1\,(i,i)} + \beta_i}{\beta_i}. \end{aligned}$$

Furthermore, we define $H_1, H_2, \cdots, H_k$ to be the Householder matrices corresponding to $y_1, y_2, \cdots, y_k$.

A possible decomposition into Q and R is shown in Equation (3.23) since $(u_1, u_2, \cdots, u_k)$ is orthogonal and the matrix containing $\alpha_{i,j}$ is upper triangular. If the Householder transformations are defined as it has been done in Algorithm 1, then R has the following form:

$$R = \begin{bmatrix} -s_1\alpha_{1,1} & -s_1\alpha_{1,2} & \cdots & -s_1\alpha_{1,k} \\ & -s_2\alpha_{2,2} & \cdots & -s_2\alpha_{2,k} \\ & & & \cdots \\ & & & -s_k\alpha_{k,k} \end{bmatrix}, \tag{3.25}$$

where s_i is the sign of $A_{i-1\,(i,i)}$.

For the first row of R we can easily show that

$$R_{1,j} = (H_1 \cdot A_0)_{1,j} = -s_1\alpha_{1,j}.$$

After the application of the first Householder transformation $A_1 = H_1 A_0$ has the following form:

$$A_{1\,(:,1:k)} = H_1(a_1, a_2, \cdots, a_k) = (\hat{u}_1, \hat{u}_2, \cdots, \hat{u}_k) \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,k} \\ & \alpha_{2,2} & \cdots & \alpha_{2,k} \\ & & & \cdots \\ & & & \alpha_{k,k} \end{pmatrix},$$

where $\hat{u}_i = H_1 u_i$. Furthermore, we can show that $\hat{u}_1 = -s_1 e_1$ and $\hat{u}_{i\,(1)} = 0$ for $i > 1$.

If we now look at the matrix $A_{1\,(2:n,2:k)}$, we come upon the same pattern as for A_0. But now the QR-decomposition is of size $(n-1) \times (k-1)$ instead of $n \times k$. As for $R_{(1,:)}$ we can now determine $R_{(2,:)}$. By induction Equation (3.25) is true.

To compute the coefficients $\alpha_{i,j}$ of the matrix R we need all combinations of dot products $D = A^TA$ where $D_{(i,j)} = a_i^T a_j$. In addition we need all signs s_i to get the final content of R. In Algorithm 25 we show how to compute all the $\alpha_{i,j}$ out of D. Please note that this algorithm corresponds to a Cholesky-decomposition of A^TA.

Algorithm 25 Computation of $\alpha_{i,j}$ (Cholesky decomposition)

```
1: for $i = 1 \to k$ do
2:     $\alpha_{i,i} \leftarrow \sqrt{D_{(i,i)}}$
3:     for $j = i + 1 \to k$ do
4:         $\alpha_{i,j} \leftarrow \frac{D_{(i,j)}}{\alpha_{i,i}}$
5:         for $l = i + 1 \to j$ do
6:             $D_{(l,j)} \leftarrow D_{(l,j)} - \alpha_{i,j}\alpha_{i,l}$
7:         end for
8:     end for
9: end for
```

After computing the coefficients $\alpha_{i,j}$ we need an update strategy to compute A_i out of A_{i-1} without any further synchronization requirements. Therefor, Equation (3.12) can be generalized to

$$z_{i\,(j)} = A_{i-1\,(i,j)} + \frac{A^T_{i-1\,(i:n,i)} \cdot A_{i-1\,(i:n,j)}}{\beta_i},$$

and, thus, using Equation (3.24),

$$\begin{aligned} A_{i\,(i+1:n,j)} =& A_{i-1\,(i+1:n,j)} - z_{i\,(j)} \cdot \frac{A_{i-1\,(i+1:n,i)}}{A_{i-1\,(i,i)} + \beta_i} \\ =& A_{i-1\,(i+1:n,j)} - \rho_{i,j} A_{i-1\,(i+1:n,i)}, \text{ with} \qquad (3.26) \\ \rho_{i,j} =& \frac{A_{i-1\,(i,j)} + \frac{A^T_{i-1\,(i:n,i)} \cdot A_{i-1\,(i:n,j)}}{\beta_i}}{A_{i-1\,(i,i)} + \beta_i}. \qquad (3.27) \end{aligned}$$

Since

$$A^T_{i-1\,(i:n,i)} A_{i-1\,(i:n,j)} = \underbrace{(\alpha_{i,i}\hat{u}_i)^T}_{A^T_{i-1\,(i:n,i)}} \cdot \underbrace{(\hat{u}_i \; \hat{u}_{i+1} \; \cdots \; \hat{u}_j) \cdot (\alpha_{i,j} \; \alpha_{i+1,j} \; \cdots \; \alpha_{j,j})^T}_{A_{i-1\,(i:n,j)}} = \alpha_{i,i}\alpha_{i,j},$$

Equation (3.27) can be simplified to

$$\rho_{i,j} = \frac{A_{i-1\,(i,j)} + \frac{\alpha_{i,i}\alpha_{i,j}}{\beta_i}}{A_{i-1\,(i,i)} + \beta_i}, \qquad (3.28)$$

Algorithm 26 HouseGen, rank-k version

```
1: $D \leftarrow A^T A$
2: Compute $\alpha_{i,j}$ (Algorithm 25)
3: for $i = 1 \rightarrow k$ do
4:     $\beta_i \leftarrow \alpha_{i,i} \cdot \text{sign}(A_{i-1\,(i,i)})$
5:     $R_{(i,i)} \leftarrow -\beta_i$
6:     $\tau_i \leftarrow T_{(i,i)} \leftarrow \frac{A_{i-1\,(i,i)}+\beta_i}{\beta_i}$
7:     for $j = i+1 \rightarrow k$ do
8:         $\rho_{i,j} \leftarrow \frac{A_{i-1\,(i,j)}+\frac{\alpha_{i,i}\alpha_{i,j}}{\beta_i}}{A_{i-1\,(i,i)}+\beta_i}$
9:         $A_{i\,(i+1:n,j)} \leftarrow A_{i-1\,(i+1:n,j)} - \rho_{i,j} A_{i-1\,(i+1:n,i)}$
10:        $R_{(i,j)} \leftarrow -\,\text{sign}(A_{i-1\,(i,i)}) \cdot \alpha_{i,j}$
11:    end for
12:    $y_i \leftarrow (0_{(1:i-1)}, 1, \frac{A_{i-1\,(i+1:n,i)}}{A_{i-1\,(i,i)}+\beta_i})^T$
13: end for
```

where $\hat{\hat{u}}$ corresponds to u after applying i Householder transformations. Out of Equation (3.26) and Algorithm 25 we can now formulate the rank-k variant of *HouseGen* (see Algorithm 26). The update of the matrix A in line (9) can easily be done in a blocked fashion such that cache efficiency is guaranteed.

Once a Householder vector is generated, there are several blocking possibilities for distributing the vectors and applying the transformations. For a detailed discussion of the different blocking possibilities and their effects we refer to [80]. In the following runtime estimation as well as for the performance measurements in Ch. 4 we use full blocking. This means that the broadcasting of Householder vectors as well as the generation of the compact WY representation T and their left-sided application on the remaining panels of the tall and skinny matrix occurs once if the number of k (maximal blocking factor) Householder transformations is reached. Thus, the proceeding of the rank-k QR-decomposition is the same as for Algorithm 24, except that the rank-2 routines are replaced by their rank-k variants and the number of loop iterations is adjusted accordingly. The rank-k QR-decomposition is shown in Algorithm 27.

Accuracy analysis As for the rank-2 QR-decomposition, the rank-k variant can run into numerical problems. In the following we will generalize the criterion for the stability of the algorithm to arbitrary blocking.

Let $e(x)$ be an upper bound for the numerical error when computing x. For the

Algorithm 27 QR-decomposition, rank-k version

```
1: for $block\_col = 1 \to m/k$ do
2:     $col \leftarrow k * block\_col - 1$
3:     $T, Y, R \leftarrow HouseGenk(A_{(:,col:col+k-1)})$
4:     $T \leftarrow CWYGen(Y, T)$
5:     $A_{(:,col+k:m)} \leftarrow CWYHouseLeft(T, Y, A_{(:,col+k:m)})$
6:     $A_{(:,col:col+k-1)} \leftarrow R$
7: end for
```

computation of dot products we can estimate the error with

$$e(D_{(i,i)}) = \sum_{l=1}^{n} (a_{i\,(l)}^2 \epsilon) \leq (a_i^T a_i) n\epsilon = (\alpha_{1,i}^2 + \alpha_{2,i}^2 + \cdots + \alpha_{i,i}^2) n\epsilon, \text{ and} \tag{3.29}$$

$$\begin{aligned} e(D_{(j,i)}) =& \sum_{l=1}^{n} (a_{i\,(l)} a_{j\,(l)} \epsilon) \leq \|a_i\| \|a_j\| n\epsilon \\ =& \sqrt{(\alpha_{1,j}^2 + \alpha_{2,j}^2 + \cdots + \alpha_{j,j}^2)(\alpha_{1,i}^2 + \alpha_{2,i}^2 + \cdots + \alpha_{i,i}^2)} n\epsilon, \end{aligned} \tag{3.30}$$

for the diagonal and non-diagonal elements of D respectively.

Out of Algorithm 25 and Equation (3.29) and (3.30) we can now derive error bounds for $\alpha_{i,i}$ and $\alpha_{j,i}$ with $j < i$:

$$e(\alpha_{i,i}^2) = e(D_{(i,i)}) + \sum_{j=1}^{i-1} e(\alpha_{j,i}^2) \tag{3.31}$$

$$e(\alpha_{j,i}) = \frac{e(D_{(j,i)}) + \sum_{l=1}^{j-1} (e(\alpha_{l,i}) e(\alpha_{l,j}))}{|\alpha_{j,j}|}. \tag{3.32}$$

For the first column of R we can estimate the error with $e(\alpha_{1,1}^2) = O(\alpha_{i,i}^2 n\epsilon)$ leading to a relative error of

$$\frac{e(\alpha_{1,1}^2)}{\alpha_{1,1}^2} = O(n\epsilon) \ll 1.$$

Considering that the relative error of all preceding diagonal elements $\alpha_{j,j}$ is bounded by $O(n\epsilon)$, we can simplify Equation (3.32) to

$$e(\alpha_{j,i}) = O\left(\sqrt{(\alpha_{1,i}^2 + \alpha_{2,i}^2 + \cdots + \alpha_{i,i}^2) n\epsilon}\right) + \sum_{l=1}^{j-1} e(\alpha_{l,i}). \tag{3.33}$$

This result, in turn, allows us to simplify Equation (3.31):

$$e(\alpha_{i,i}^2) = O((\alpha_{1,i}^2 + \alpha_{2,i}^2 + \cdots + \alpha_{i,i}^2) n\epsilon) \tag{3.34}$$

As for the rank-2 QR-decomposition we limit the relative error by claiming

$$\frac{\alpha_{1,i}^2 + \alpha_{2,i}^2 + \cdots + \alpha_{i-1,i}^2}{\alpha_{i,i}^2} \leq \epsilon_{\text{fallback}}. \tag{3.35}$$

Finally, we can set a criterion for the numerical stability of generating and applying the i-th Householder transformation: the i-th Householder vector is regarded as "stable" if Householder transformation $i-1$ is stable and Equation (3.35) is fulfilled.

Please note that we assumed the maximal blocking factor k to be a constant.

Runtime estimation As already mentioned, the rank-k QR-decomposition leads to a further reduction of synchronization requirements. Again, for the runtime estimation we assume that no fallbacks to lower blocking occur. The maximum blocking factor k is assumed to be the blocksize blk of the block-cyclic distribution.

As for the classic and rank-2 QR-decomposition we break down the estimated execution time into parts for the generation and left-sided application of Householder transformations ($t_{\text{pHouseGen\_k}}$ and $t_{\text{pHouseLeft\_k}}$).

$$\begin{aligned} t_{\text{pHouseGen\_k}} \lessapprox & \left(2n_{\text{localr}} \cdot blk^2 + \frac{blk^3}{3}\right) t_{\text{flop}} + 2n_{\text{localr}} \cdot blk \cdot t_{\text{mem}} \\ & + 2\lceil log(p_r)\rceil (blk^2 \cdot t_{\text{word}} + t_{\text{msg}}) \end{aligned} \tag{3.36}$$

$$\begin{aligned} t_{\text{pHouseLeft\_k}} \lessapprox & 5n_{\text{localr}} \cdot blk^2 \cdot t_{\text{flop}} + 4n_{\text{localr}} \cdot blk \cdot t_{\text{mem}} \\ & + 2\lceil log(p_r)\rceil (blk^2 \cdot t_{\text{word}} + t_{\text{msg}}) \end{aligned} \tag{3.37}$$

Out of Equation 3.36 and 3.37 and the required broadcast to distribute the Householder vectors to the other process columns we get runtime estimations for the whole rank-k QR-decomposition ($t_{\text{parallelQR\_k}}$) and for all QR-decompositions during the reduction to banded form ($t_{\text{qr\_k}}$):

$$\begin{aligned} t_{\text{parallelQR\_k}} \lessapprox & \left(7n_{\text{localr}} b \cdot blk + \frac{b \cdot blk^2}{3}\right) t_{\text{flop}} + 6n_{\text{localr}} b t_{\text{mem}} \\ & + \lceil log(p_r)\rceil \left(n_{\text{localr}} b t_{\text{word}} + \frac{5b}{blk} t_{\text{msg}}\right) \end{aligned} \tag{3.38}$$

$$\begin{aligned} t_{\text{qr\_k}} \lessapprox & \left(\frac{7n^2}{2p_r} \cdot blk + \frac{n \cdot blk^2}{3}\right) t_{\text{flop}} + \frac{3n^2}{p_r} t_{\text{mem}} \\ & + \lceil log(p_r)\rceil \left(\left(\frac{n^2}{2p_r} + 2n \cdot blk\right) t_{\text{word}} + \frac{5n}{blk} t_{\text{msg}}\right) \end{aligned} \tag{3.39}$$

The rank-k QR-decomposition requires a higher number of flops on the critical path since the generation of a set of Householder transformations and its application to the rest of the tall and skinny matrix do not overlap. However, this term should never dominate the runtime of the whole reduction to banded form. The biggest advantage is the reduced number of required messages which should significantly improve the scalability of the algorithm.

Worst case handling For the blocked QR-decomposition we have to consider also the worst case scenario where we fall back to the unblocked case in each column of the matrix. In this case we get a similar behavior as for the classic QR-decomposition but we do additional work for the computation of Cholesky(A^TA). This causes additional costs of $\frac{n^2 \cdot blk^2}{2p_r} t_{\text{flop}} + blk^2 \cdot n \lceil log(p_r) \rceil t_{\text{word}}$ for the whole reduction to banded form. These costs can be reduced to $\frac{n^2 \cdot blk}{2p_r} t_{\text{flop}} + blk \cdot n \lceil log(p_r) \rceil t_{\text{word}}$ with the following approach.

If a fallback to lower blocking occurs, the computed $\alpha_{i,j}$ are not accurate enough to compute Householder vectors thereout. However, the $\alpha_{i,j}$ can still be used to decide, whether a blocked decomposition is numerically stable or not. Out of this idea we formulate Algorithm 28. Initially, we compute once the Cholesky-decomposition of

Algorithm 28 HouseGen, rank-k, framework

```
1: $D \leftarrow Cholesky(A^TA)$
2: $i \leftarrow 0,\ l \leftarrow 1,\ k_0 \leftarrow 0$
3: **while** $i < k$ **do**
4:     $b \leftarrow CheckNumericalStability(D_{i+1:k,i+1:k})$
5:     $k_l \leftarrow k_{l-1} + b,\ l \leftarrow l + 1,\ i \leftarrow i + b$
6: **end while**
7: $i \leftarrow 0$
8: **for** $i < l - 1$ **do**
9:     $HouseGenk(A_{k_i+1:n,k_i+1:k_{i+1}})$
10:    $i \leftarrow i + 1$
11: **end for**
```

A^TA. Afterwards, we use the result D to successively determine the allowed blockings for decomposing A. *CheckNumericalStability* returns the maximal blocking factor for the given matrix according to the defined fallback criterion. Finally, we use the maximal blockings k_l to call the original rank-k Householder vector generation (Algorithm 26). Within these calls, the computation of the coefficients $\alpha_{i,j}$ has to be performed only on a submatrix of A.

Using this approach, the blocked QR-decomposition attains the same worst case run-time behavior as the classic unblocked QR-decomposition.

We can summarize that the blocked QR-decomposition is a very promising approach to resolve the bottlenecks of the classic parallel Householder QR-decomposition of tall and skinny matrices (high synchronization and memory bandwidth requirements). Contrary to TSQR, the algorithm doesn't lead to computational overhead for the application of the Householder transformations if used within the reduction to banded form. The blocked QR-decomposition reduces the number of messages by the defined maximal blocking factor k if the corresponding submatrix is well conditioned. As we will see in Ch. 4, this is the case most of the time. If an accuracy loss is imminent, the algorithm switches back to lower blocking or even to the unblocked case and guarantees, thus, numerical stability.

3.4 Tridiagonalization of banded matrices

After reducing a symmetric matrix to banded form, we have to bring the banded matrix to tridiagonal form. Although, this step is less compute intensive compared to the reduction to banded form ($\frac{4}{3}n^3$ vs. $6n^2b$), it is not less important due to the limited parallelizability of this part of the tridiagonalization. In this section we describe the parallel tridiagonalization based on Householder transformations, as published in [68], and provide a detailed runtime estimation. Furthermore, we will present our enhancements regarding the parallelization of the problem.

3.4.1 Existing parallelization

The parallelization approach in [68] is a pipelining approach which exploits that certain operations from different stages of the algorithm can be computed concurrently. As has been said in Sect. 2.5, during the tridiagonalization process the banded matrix grows to block tridiagonal form with a blocksize of b if b is the bandwidth of the initial banded matrix. In each stage v of the algorithm, the block tridiagonal matrix can be partitioned as depicted in Figure 2.3. Two blocks $B_{i,i}$ and $B_{i+1,i}$ can be combined to a block pair P_i with

$$P_i^{(v)} \leftarrow \begin{bmatrix} B_{i,i}^{(v)} \\ B_{i+1,i}^{(v)} \end{bmatrix}. \tag{3.40}$$

The block pairs, in turn, are distributed in a one-dimensional blocked or block-cyclic fashion across the processes. The detailed distribution has effects on the load balance

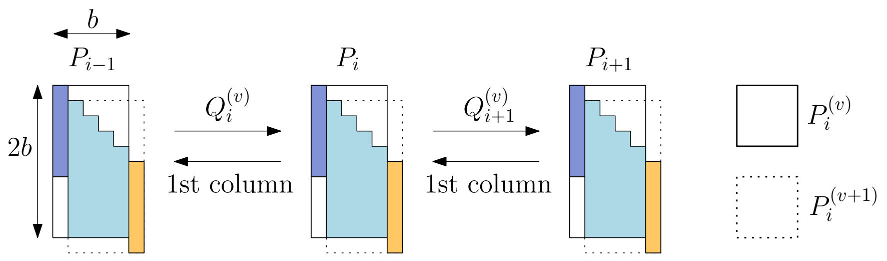


Figure 3.2: Communication pattern of the band reduction. The newly generated Householder transformation $Q_{i+1}^{(v)}$ is sent to the right neighboring block pair (if existing), the transformed first column of $P_i^{(v)}$ is sent to the left neighboring block pair. Accordingly Q_i is received from the left and the first column of P_{i+1} from the right neighboring block pair.

of the problem and the amount of required communication. The details of the data distribution will be discussed later in this section.

For the first block pair $P_0^{(v)}$ we have to determine a Householder transformation $Q_1^{(v)}$ with

$$Q_1^{(v)} B_{10}^{(v)} = (*, 0, \dots, 0)^T. \tag{3.41}$$

For all other block pairs $P_i^{(v)}$, $i > 0$ we have to apply $Q_i^{(v)}$ and determine a new transformation $Q_{i+1}^{(v)}$ which eliminates all but the first element of the first column of $B_{i+1,i}^{(v)}$:

$$transform\ P_i^{(v)} \rightarrow \begin{bmatrix} Q_i^{(v)} B_{i,i}^{(v)} {Q_i^{(v)}}^T \\ Q_{i+1}^{(v)} B_{i+1,i}^{(v)} {Q_i^{(v)}}^T \end{bmatrix}, \quad i > 0. \tag{3.42}$$

In the next stage of the algorithm each block pair $P_i^{(v+1)}$ is shifted by one column to the right and by one row to the bottom.

It is easy to see that transformations from different stages can be executed concurrently. More concrete, the transformation of $P_i^{(v+1)}$ can start if $Q_i^{(v+1)}$ and the first column of the transformed block pair $P_{i+1}^{(v)}$ are available. If P_i and P_{i-1} or P_i and P_{i+1} are not located on the same process, explicit communication is required. The communication between different block pairs is illustrated in Figure 3.2. Algorithm 29 shows the corresponding parallel algorithm as published in [68]. The algorithm consists of two different programs for block pair P_0 and the remaining block pairs and assumes that block pair P_i is handled by process p_i in each case. Algorithm 29 can be generalized such that the block pairs are distributed in any block-cyclic form across the processes. We refer to [68] for this variant. In particular the control flow has to be altered such that a deadlock free execution is guaranteed. Furthermore it allows to bundle communication if the distribution consists of more than one cycle.

Algorithm 29 Parallel band reduction [68]

```
1: $P_0$ :
2: for $v = 1 \to n-1$ do
3:    transform $P_0^{(v)} \to Q_1^{(v)}$
4:    send $Q_1^{(v)}$ to $P_1$
5:    receive 1st column of $P_1$
6: end for
7:
8: $P_\beta$ $(\beta \geq 1)$ :
9: for $v = 1 \to n-1$ do
10:    if $P_\beta$ is not empty then
11:       receive $Q_\beta^{(v)}$ from $P_{\beta-1}$
12:       transform $P_\beta^{(v)} \to Q_\beta^{(v+1)}$
13:       send 1st column of $P_\beta$ to $P_{\beta-1}$
14:       shift $P_\beta$ by one column/row
15:       if $P_{\beta+1}$ is not empty then
16:          send $Q_{\beta+1}^{(v)}$ to $P_{\beta+1}$
17:          receive 1st column of $P_{\beta+1}$
18:       end if
19:    end if
20: end for
```

For the runtime estimation we assume to have a one-dimensional block-cyclic distribution of block pairs where l is the blocksize of the distribution and $c = \lceil \frac{n}{b \cdot l \cdot p} \rceil$ is the number of cycles. As we said before, the data distribution has effects on load balancing and communication requirements. It can be seen that block pairs in the upper part of the band require more work than block pairs further down the matrix. More precise, P_1, the block pair with the highest workload, requires twice as much work than the average workload per block pair ($12nb^2$ vs. $6nb^2$). Thus, with a pure block data layout ($c = 1$), the load imbalance of the parallel algorithm would be 2 ("highest workload / average workload"). A block-cyclic distribution reduces the load imbalance to $1 + 1/c$.

On the other hand, the communication requirements are minimal if $c = 1$. If neighboring block pairs are located on different processes, we need to send/receive one Householder vector and one column of the band in each stage of the algorithm. Thus, the communication requirements grow linear with the number of cycles c.

The third aspect which is influenced by the data distribution is "idle waiting". From Figure 3.2 and Algorithm 29 we can see that neighboring block pairs cannot be transformed at the same time. If $l = 1$, a process will idle for, at least, half of the time while waiting for Householder transformations from the left and matrix columns from the right neighboring process. Hence, the parallelizability of the algorithm is limited to $p = \frac{n}{2b}$. Any further increase of the number of processes doesn't lead to any speedup.

For the expected runtime of the algorithm we distinguish between the two cases $p \leq \frac{n}{2b}, l \geq 2$

$$t_{\text{bndtrd}} \approx n \left\lceil \frac{n}{bp} \right\rceil (1 + 1/c)(6b^2 t_{\text{flop}} + 5/2b^2 t_{\text{mem}}) + (c+1)nbt_{\text{word}} + 2nt_{\text{msg}}, \tag{3.43}$$

and $p > \frac{n}{2b}, l = 2$

$$t_{\text{bndtrd}} \approx 24nb^2 t_{\text{flop}} + 10nb^2 t_{\text{mem}} + 2nbt_{\text{word}} + 2nt_{\text{msg}}. \tag{3.44}$$

A blocksize l of 1 is counterproductive and increases the communication requirements of the algorithm (according to our model).

3.4.2 Fine grained data dependency analysis

The parallelization of the algorithm in [68] can be described with the dependency graph in Figure 3.3 (left). Thereby, a node $Q_i^{(v)}$ of the graph stands for the operation "transform $P_i^{(v)}$". The arrows in the graph represent the data dependencies. Each node of the graph, beside the first, requires a Householder transformation from the

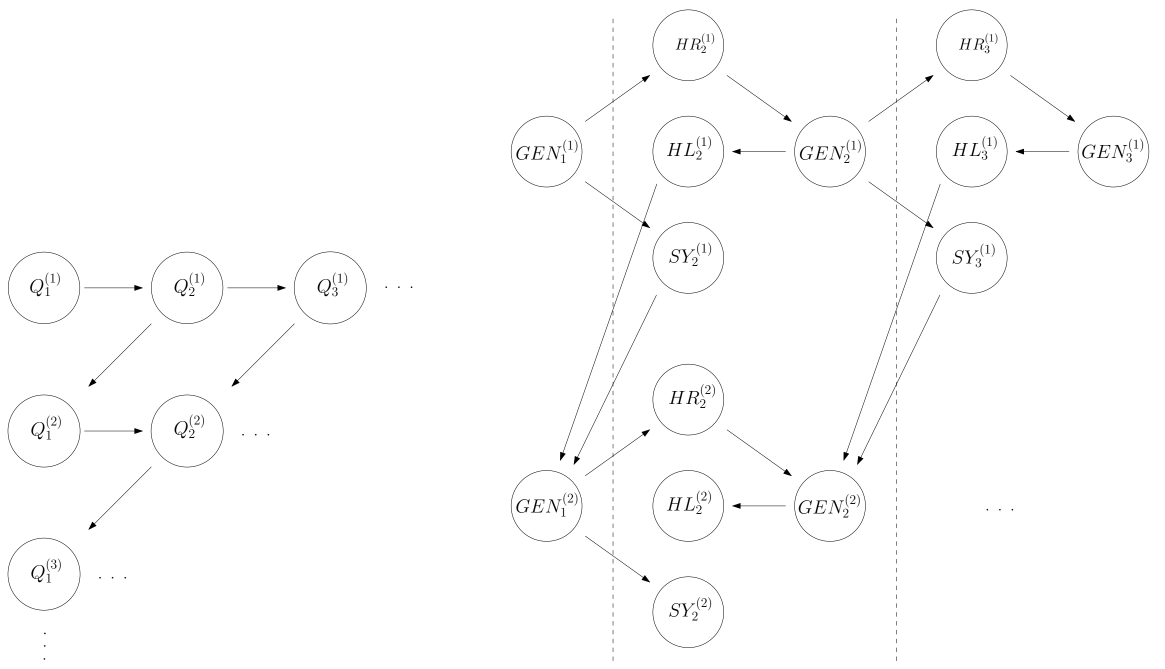


Figure 3.3: Simplified (left) and more detailed (right) variant of the dependency graph for the band reduction based on Householder transformations [81]. One bubble in the graph on the left represents the operations from Equation (3.42). In the graph on the right the operations are divided into four subtasks: Generation of a Householder vector (GEN), left-sided (HL), right-sided (HR) and symmetric (SY) application of a Householder vector.

left neighboring block pair (horizontal arrows) and each node, beside the last, requires a matrix column from the right neighboring block pair (diagonal arrows).

With the aim to increase the parallelizability of the problem we can further divide the operation "transform $P_i^{(v)}$" into four subtasks: (1) apply $Q_i^{(v)}$ from both sides to $B_{i,i}$, (2) apply $Q_i^{(v)}$ from the right side to $B_{i+1,i}$, (3) generate $Q_{i+1}^{(v)}$ and (4) apply $Q_{i+1}^{(v)}$ from the left side to $B_{i+1,i}$. In Figure 3.3 (right) we can see a more detailed variant of the dependency graph. The four subtasks are abbreviated with $SY_i^{(v)}$, $HR_i^{(v)}$, $GEN_i^{(v)}$ and $HL_i^{(v)}$.

Out of the dependency graph in Figure 3.3 (right) we can formulate Algorithm 30. The basic idea is to bring forward the sending of data as far as possible. $Q_{\beta+1}^{(v)}$ can be generated and sent as soon as $B_{\beta+1,\beta}^{(v)} Q_\beta^{(v)^T}$ has been computed. The first column of P_β can be sent to $P_{\beta-1}$ as soon as $Q_\beta^{(v)} B_{\beta,\beta}^{(v)} Q_\beta^{(v)^T}$ has been computed. Algorithm 30 allows to use up to $p = \frac{n}{b}$ processes without provoking idle time. According to our

Algorithm 30 Parallel band reduction (improved variant)

```
1: $P_0$ :
2: for $v = 1 \to n-1$ do
3:     wait for last column of $P_1^{(v)}$
4:     generate $Q_1^{(v)}$
5:     send $Q_1^{(v)}$ to $P_1$
6: end for
7:
8: $P_\beta$ $(\beta \geq 1)$ :
9: for $v = 1 \to n-1$ do
10:    if $P_\beta$ is not empty then
11:        wait for $Q_\beta^{(v)}$ from $P_{\beta-1}$
12:        wait for last column of $P_\beta$
13:        if $P_{\beta+1}$ is not empty then
14:            compute $B_{\beta+1,\beta}^{(v)} {Q_\beta^{(v)}}^T$
15:            generate $Q_{\beta+1}^{(v)}$
16:            send $Q_{\beta+1}^{(v)}$ to $P_{\beta+1}$
17:        end if
18:        compute $Q_\beta^{(v)} B_{\beta,\beta}^{(v)} {Q_\beta^{(v)}}^T$
19:        send 1st column of $P_\beta$ to $P_{\beta-1}$
20:        if $P_{\beta+1}$ is not empty then
21:            compute $Q_{\beta+1}^{(v)} B_{\beta+1,\beta}^{(v)}$
22:        end if
23:        shift $P_\beta$ by one column/row
24:    end if
25: end for
```

model the time on the critical path can be estimated with

$$t_{\text{bndtrd}} \approx 12nb^2 t_{\text{flop}} + 6nb^2 t_{\text{mem}} + 3nbt_{\text{word}} + 3nt_{\text{msg}}, \quad p \geq \frac{n}{b}. \tag{3.45}$$

Thereby, the communication to memory is reduced to $O(nb)$ words if the working set of $\frac{3b^2}{2}$ words fits into the cache.

It has to be mentioned that the operations $SY_i^{(v)}$, $HR_i^{(v)}$ and $HL_i^{(v)}$ can further be sub-divided into subtasks (see Algorithm 2, 3 and 4 in Sect. 2.3.4). The right-sided application of a Householder transformation, for example, can be split up into a matrix vector product and a matrix update. In doing so, we can prepone the sending of data even further. However, this has no significant effects on the estimated runtime.

3.5 Tridiagonal-to-banded back transformation

After computing the eigenpairs of the tridiagonal matrix using one of the mentioned algorithms (see Sect. 2.3.2), the eigenvectors of the tridiagonal matrix have to be transformed back to the eigenvectors of the original matrix. In a first step we transform the eigenvectors of the tridiagonal matrix back to those of the banded matrix.

During the tridiagonal-to-banded back transformation many relatively short Householder transformations have to be applied to the eigenvector matrix. As we introduced in Sect. 2.5, the Householder transformations Q_i^v from the band reduction are numbered with two indices: The index v stands for the stage of the algorithm. i stands for the sweep of the algorithm. All Householder transformations have b nonzero elements (except of the transformation from the last sweep in each stage). Each stage i (with $1 \leq i \leq n-2$) of the algorithm consists of $\lceil \frac{n-i}{b} \rceil$ sweeps. Thus, the total number of Householder transformations is $\frac{n^2}{2b} + O(n)$. All transformations and their order of application are sketched in Figure 3.4. For the band reduction we have the requirements that (1) $Q_{i+1}^{(v)}$ is applied after $Q_i^{(v)}$ and that (2) $Q_i^{(v+1)}$ is applied after $Q_{i+1}^{(v)}$ (see Figure 3.4, left). For the back transformation of eigenvectors the order of application is the reverse order of the reduction step. However, the requirement (1) drops out because all transformations are known in advance. This leads to the dependencies sketched in Figure 3.4, right.

As mentioned in Sect. 2.3.3, the Householder transformations have to be applied from the left side to the eigenvector matrix X_T. The eigenvector matrix is a matrix of size $n \times k$ if k eigenvectors are transformed back. It is obvious that each eigenvector can be transformed independently from each other.

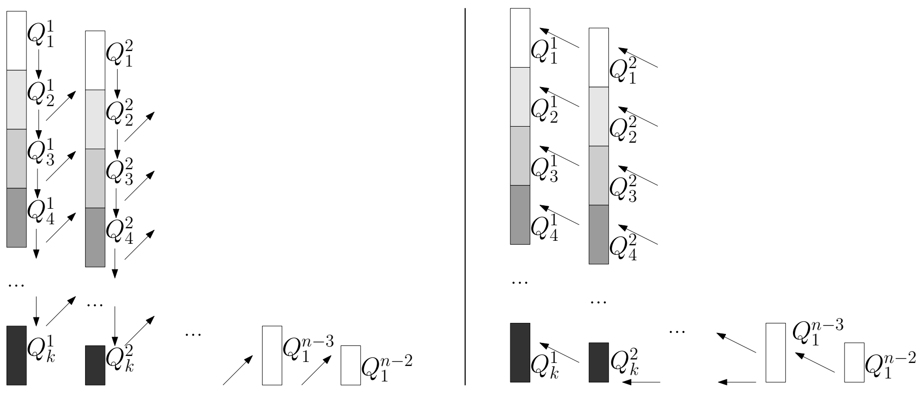


Figure 3.4: Householder vectors from the reduction from banded to tridiagonal form [68]. The arrows indicate the order of execution during reduction (left) and back transformation of eigenvectors (right).

The Householder transformations can be applied in a cache efficient way to the matrix. We have developed different strategies for this task. One is based on WY transformation, the other is based on a explicit loop blocking in combination with compact WY transformations with a small blocking factor. The developed techniques will be presented in Sect. 3.5.1.

For the ELPA library we have developed and implemented two different parallelization schemes. According to the underlying parallel data distribution the algorithms are called 1D and 2D parallelization. Thereby, only the latter is the contribution of the author. The parallelization approaches will be presented in Sect. 3.5.2 and 3.5.3.

3.5.1 High performance kernels

The general construction and application of WY transformations is described in Sect. 2.3.4. For the tridiagonal-to-banded back transformation we have the special case that (1) the Householder vectors are relatively short (b nonzero elements) and (2) Householder vectors which have to be applied one after another are vertically shifted by one element. For example, $Q_1^{(v+1)}$ has nonzero elements from index $v+2$ to $v+b+1$ and $Q_1^{(v)}$, which has to be applied next, has nonzeros from $v+1$ to $v+b$. Due to this shift, W and Y are of size $(b+n_b-1)\times n_b$ instead of $b\times n_b$. The nonzero structure in the matrices W and Y is sketched in Figure 3.5.

The shift of Householder vectors in combination with a small vector length b leads to significant overhead for the application of WY transformation. If zero entries are not exploited, we need $4k(b+n_b-1)n_b$ arithmetic operations instead of $4kbn_b$ to apply

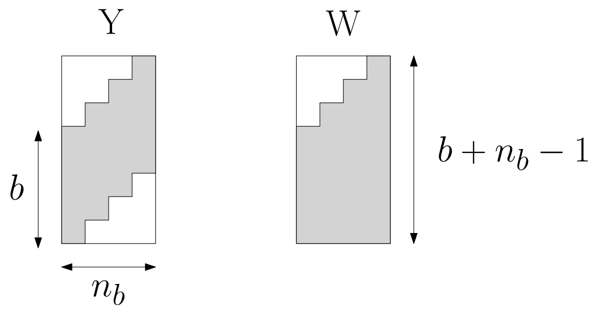


Figure 3.5: Nonzero structure of the matrices W and Y during the tridiagonal-to-banded back transformation.

n_b Householder transformations on k eigenvectors. This is an overhead of $\frac{b+n_b-1}{b}$. According to our model, the runtime for the generation and application of a WY transformation can be estimated with $(4kn_b + 2n_b^2)(b + n_b - 1) \cdot t_{\text{flop}} + (4k + n_b^2)(b + n_b - 1) \cdot t_{\text{mem}}$.

To overcome this overhead we have developed an alternative approach which is based on simple loop blocking. In the following we will call this proceeding non-WY approach. Instead of WY transformations we use simple unblocked Householder transformations (later on we will substitute the unblocked Householder transformations with compact WY representations with a small blocking factor). However, these Householder transformations are not applied to all eigenvectors at a time. The loop over the eigenvectors is blocked such that k_b eigenvectors are transformed simultaneously and the working set of bk_b words fits into the cache. In this way all Householder transformations of one sweep are applied to the k_b eigenvectors. Afterwards, the next set of k_b eigenvectors is transformed. The detailed proceeding is shown in Algorithm 31. Figure 3.6 illustrates the loop blocking. The time to apply

Algorithm 31 Tridiagonal-to-banded back transformation (sequential version, non-WY)

```
1: for $sweep = 1 \to \lceil \frac{n-1}{b} \rceil$ do
2:     for $i = 1 \to k$ step $k_b$ do
3:         for $j = n - 2 - (sweep - 1) \cdot b \to 1$ step $-1$ do
4:             $HouseLeft(X_{(1:n,i:i+k_b-1)}, Q^{(j)}_{sweep})$
5:         end for
6:     end for
7: end for
```

n_b Householder transformations with the non-WY approach can be estimated with $4kbn_b \cdot t_{\text{flop}} + \left(\left\lceil \frac{k}{k_b} \right\rceil bn_b + (b + n_b - 1)k\right) \cdot t_{\text{mem}}$.

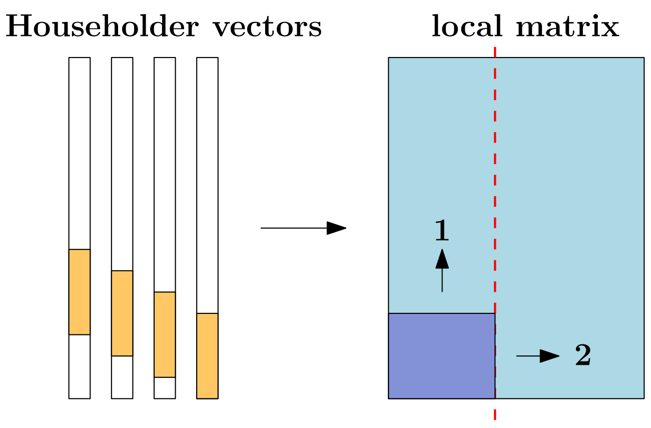


Figure 3.6: Loop blocking during the tridiagonal-to-banded back transformation. The loop over the eigenvectors is blocked (red dashed line) such that the working set (dark blue area) fits into the cache and can be reused for the next Householder transformation. When all Householder transformations of one sweep have been applied to the block of eigenvectors, we continue with the next block.

Some microarchitectures, for example PowerPC450, have a write-through cache as L1-cache. This means that each write to the cache causes also a write to the next cache level or the main memory. In other words, a write-through cache accelerates read operations if the data is already in the cache. Write operations, however, are still limited by the memory bandwidth or the bandwidth of the next cache level. The non-WY approach, based on unblocked Householder transformations, lacks in performance on such systems. The performance bottleneck is the rank-1 update within the $HouseLeft$ operation (Algorithm 2, Sect. 2.3.4). To overcome this problem we have developed kernels which apply two or even four Householder transformations to a set of eigenvectors using the compact WY technique. The improved cache behavior of this proceeding cannot be described by our model and is therefor ignored for the runtime estimation of the algorithms. The detailed implementation and the corresponding results will be presented in Ch. 4.

3.5.2 1D parallelization

The 1D parallelization uses the fact that each eigenvector can be transformed independently. The k eigenvectors are distributed uniformly to the p processes, leading to a 1D block layout. The eigenvectors are then transformed according to the order in Figure 3.4, right. This proceeding requires no synchronization for the application of the Householder transformations. However, the Householder vectors have to be distributed to all processes. Since every process needs the whole set of Householder vectors this is a very costly operation.

The distribution can be done with one huge broadcast operation if enough buffer space is available. In our implementation the distribution is split up such that one broadcast is performed for each sweep. For the model, however, we assume to have one single huge broadcast.

If we use the non-WY approach for the application of Householder transformations, we get the following runtime estimation:

$$\begin{aligned} t_{\text{trdback\_1d}} \approx {} & 2 \left\lceil \frac{k}{p} \right\rceil n^2 t_{\text{flop}} + \left(\left\lceil \frac{k}{pk_b} \right\rceil + \left\lceil \frac{k}{p} \right\rceil \frac{1}{b} \right) \frac{n^2}{2} t_{\text{mem}} \\ & + \lceil log(p) \rceil \left(n^2 t_{\text{word}} + t_{\text{msg}} \right). \end{aligned} \tag{3.46}$$

The estimated runtime for the WY approach is always higher (especially if the number of eigenvectors per process gets small). For the blocksize k_b of the loop blocking holds $k_b \leq \frac{M}{b}$.

From Equation (3.46) we can identify three terms which limit the scalability and the performance of the algorithm:

- $\lceil log(p) \rceil n^2 t_{\text{word}}$ - the broadcast of Householder vectors doesn't scale at all and will become a bottleneck if the number of processes p increases.
- $2 \left\lceil \frac{k}{p} \right\rceil n^2 t_{\text{flop}}$ - the maximum level of parallelism is limited by the number of eigenvectors k.
- $\left\lceil \frac{k}{pk_b} \right\rceil \frac{n^2}{2} t_{\text{mem}}$ - the performance of the kernels will decrease if the number of local eigenvectors gets small. For this scenario the loading of Householder vectors dominates the runtime of the kernels.

3.5.3 2D parallelization

According to the data distribution, the second parallelization scheme is called 2D parallelization. The eigenvector matrix of size $n \times k$ is distributed in a 2D blocked manner across a 2D processor grid with p_r rows and p_c columns.

The 2D approach [47, 81] uses a second level of parallelism which becomes clear after a closer look at the Householder transformations and their effect on the eigenvector matrix. A Householder transformation updates only those rows of the eigenvector matrix where the corresponding Householder vector has nonzero entries. This fact allows us to transform a single eigenvector in parallel. In the following we will describe the detailed algorithm.

The k eigenvectors are distributed uniformly across p_c process columns, similar to the 1D approach. The individual eigenvectors, in turn, are distributed in a blocked manner across the p_r processes of a process column. Except for the distribution of Householder vectors, the individual process columns are independent from each other. For the parallelization within one process column the dependencies in Figure 3.4 (right) have to be preserved. This leads to a pipelining algorithm. The process on the bottom of a process column starts the pipeline and applies the Householder transformations from the first sweep to its local part of the eigenvectors. In the next step the upper neighboring process can apply transformations from sweep one to its local part of the eigenvector matrix while the process on the bottom can start with the transformations from sweep two. Finally, after $p_r - 1$ steps the pipeline is full and all processes are involved in computations.

During this pipelining algorithm it is necessary to exchange data between vertically neighboring processes. Lets assume $Q_i^{(v)}$ is the last Householder transformation which is applied by process p_j of each process column and $Q_i^{(v-1)}$ is the first transformation, applied by process p_{j-1}. It is obvious that $Q_i^{(v)}$ and $Q_i^{(v-1)}$ operate on $b-1$ shared rows of the eigenvector matrix. In a distributed memory environment we have to define a halo region of $b-1$ rows which is duplicated and exists on both vertically neighboring processes. After process p_j has applied $Q_i^{(v)}$, it sends the content of the halo region to process p_{j-1} and process p_{j-1} can continue with the application of $Q_i^{(v-1)}$. Once process p_{j-1} has applied the transformation $Q_i^{(v-b)}$ (this is the last transformation of the current step which updates the halo region), it can send the content of the halo region to process p_j. The halo region and the data exchange are sketched in Figure 3.7.

To avoid idle waiting we have to split up the application of the Householder transformations of the current step. We define m as the local matrix height (according to the 2D block layout) which corresponds to the number of Householder transformations, we have to apply in the current step. Furthermore we define $m \geq 2b$. In phase (1) we apply the first $b-1$ transformations (which update the lower halo region), in phase (2) we apply transformation b through $m-b+1$ and in phase (3) we apply the last $b-1$ transformations (which update the upper halo region). Algorithm 32 sketches the basic cycle of computation and communication during the tridiagonal-to-banded back transformation. The total runtime for the synchronization of halo regions can be estimated with $2n \left\lceil \frac{k}{p_c} \right\rceil t_{\text{word}} + \frac{2n}{b} t_{\text{msg}}$. Please note that we need a non-blocking receive

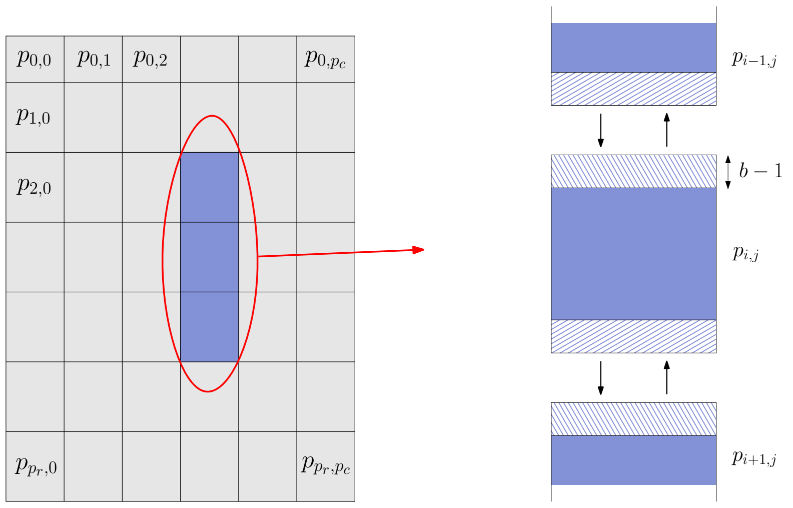


Figure 3.7: 2D data distribution and synchronization between vertically neighboring processes.

to attain this runtime estimation. In Ch. 4 we will see some interesting results where the non-blocking receive somehow doesn't work.

Beside the synchronization between vertically neighboring processes we need communication for the distribution of Householder vectors. For the initial distribution of Householder vectors we assume that each Householder vector resides on the process row where it will be used during the algorithm. An example of the Householder vector distribution is depicted in Figure 3.8. As for the 1D parallelization, the Householder

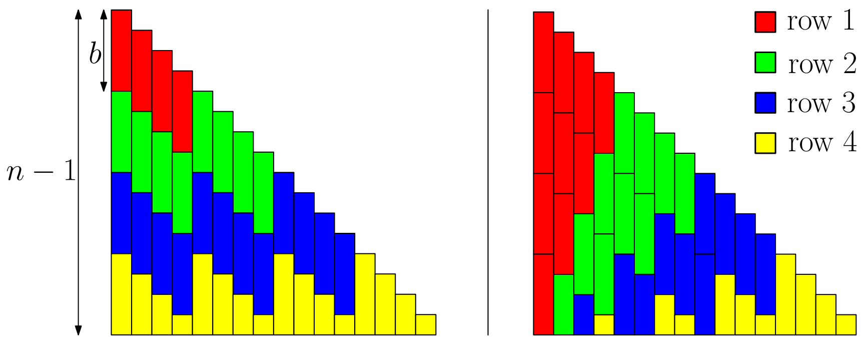


Figure 3.8: Example of the distribution of Householder vectors with static (left) and dynamic (right) data distribution ($n = 17$, $b = 4$). The Householder vectors painted in one color are distributed across one row of the 4×4 process grid.

vectors have to be broadcasted within each process row. Such that each process owns

Algorithm 32 Tridiagonal-to-banded back transformation (2D variant, simplified)

```
 1: for $step = 1 \to \frac{n-1}{b}$ do
 2:     if (not lowermost process) then
 3:         wait for lower halo
 4:         compute phase (1)
 5:         send down lower halo
 6:     else
 7:         compute phase (1)
 8:     end if
 9:     compute phase (2)
10:     if (not topmost busy process) then
11:         wait for upper halo
12:         compute phase (3)
13:         send up upper halo
14:     else
15:         compute phase (3)
16:     end if
17: end for
```

all the transformations which have to be applied to the local part of the eigenvector matrix. However, for the 2D parallelization the communication requirements drop from $O(n^2)$ to $O(n^2/p_r)$ words.

From Figure 3.8 (left) we can see that the Householder vectors are not distributed uniformly across the process grid. The processes in the lowermost process row own twice as many vectors as the average process. This affects load imbalances by a factor of 2 for the distribution and application of Householder transformations.

To overcome these load imbalances we redistribute the eigenvector matrix after each sweep of Householder transformations. When all transformations of one sweep have been applied we remove the b topmost rows from the eigenvector matrix and redistribute the matrix according to their new height in the usual 2D blocked manner. The redistribution of the matrix doesn't generate additional communication. All required data is already available due to the halo synchronization. However, the height of the lower halo has to be set to b instead of $b-1$. The height of the upper halo is computed after every sweep and may be smaller than b. The Householder vector distribution is adapted to the dynamic data distribution (see Figure 3.8, right). Please note that after the whole algorithm the complete eigenvector matrix would reside on the topmost processes. To avoid such memory imbalances we distribute the removed rows uniformly to the whole grid of processes. This results in additional communication costs of kn/p_c words for the topmost processes. For the whole algorithm we get the following runtime estimation:

$$\begin{aligned} t_{\text{trdback}} \approx & \left\lceil \frac{k}{p_c} \right\rceil \frac{2n^2}{p_r} t_{\text{flop}} + \left(\left\lceil \frac{k}{p_c k_b} \right\rceil + \left\lceil \frac{k}{p_c} \right\rceil \frac{1}{b} \right) \frac{n^2}{2p_r} t_{\text{mem}} \\ & + \left(3n \left\lceil \frac{k}{p_c} \right\rceil + \lceil log(p_c) \rceil \frac{n^2}{2p_r} \right) t_{\text{word}} + \left(\frac{3n}{b} + \lceil log(p_c) \rceil \right) t_{\text{msg}}. \end{aligned} \tag{3.47}$$

3.6 Banded-to-full back transformation

The tridiagonal-to-banded back transformations computes the eigenvectors of the banded matrix out of the eigenvectors of the tridiagonal matrix. In a final step we transform the eigenvectors of the banded matrix to those of the original matrix.

The parallel implementation of the banded-to-full back transformation is straightforward. All Householder transformations from the reduction to banded form have to be applied from the left side to the eigenvector matrix. This is done through compact WY transformations consisting of b Householder transformations each. Thus, the back transformation consists of $\frac{n-1}{b}$ steps. Thereby in step l the $n - l \cdot b - 1$ topmost rows of $Y(p_r, *)_l$ are filled with zeros. The matrices $T(*, *)_l$ have already been generated during the reduction and are still available for the back transformation. Contrary to the tridiagonal-to-banded back transformation, the eigenvector matrix $X(p_r, p_c)$ is distributed in a block-cyclic way to guarantee load balancing.

At the beginning of each step the matrices Y and T have to be broadcasted such that each process $p_{i,j}$ has access to $Y_{[i]}$ and T. This is done with one broadcast in each row of processes. Afterwards, the transformations can be applied using Algorithm 33.

Algorithm 33 Parallel $CWYLeft(X(p_r, p_c), Y(p_r, *), T(*, *))$

```
1: $U(*, p_c)_{[j]\,i} \leftarrow Y_{[i]}^T X_{[i,j]}$
2: $U_{[j]} \leftarrow Allreduce(U_{[j]\,i})$
3: $U_{[j]} \leftarrow TU_{[j]}$
4: $X_{[i,j]} \leftarrow X_{[i,j]} - Y_{[i]} U_{[j]}$
```

The runtime of Algorithm 33 can be estimated with

$$\begin{aligned} t_{\text{cwyleft}} \approx & (4bn_{\text{localr}} n_{\text{localc}} + b^2 n_{\text{localc}}) t_{\text{flop}} + 4n_{\text{localr}} n_{\text{localc}} t_{\text{mem}} \\ & + 2 \lceil log(p_r) \rceil (n_{\text{localc}} b t_{\text{word}} + t_{\text{msg}}). \end{aligned} \tag{3.48}$$

Summing up over all $\frac{n-1}{b}$ iterations and adding the costs for distributing all Householder transformations leads to the total estimated runtime of the banded-to-full back transformation:

$$\begin{aligned} t_{\text{bndback}} \lessapprox & \left\lceil \frac{k}{p_c} \right\rceil \left(\frac{2n^2}{p_r} + bn + 4n \cdot blk \right) t_{\text{flop}} + \left\lceil \frac{k}{p_c} \right\rceil \frac{n^2}{p_r b} t_{\text{mem}} \\ & + \left(\lceil log(p_r) \rceil \, n \left\lceil \frac{k}{p_c} \right\rceil + \lceil log(p_c) \rceil \left(\frac{n^2}{2p_r} + nb \right) \right) t_{\text{word}} \\ & + \frac{n}{b} \lceil log(p_r) \rceil \, t_{\text{msg}}. \end{aligned} \tag{3.49}$$

3.7 Runtime analysis

In this section we will put all the pieces from the previous sections together. At first we are going to compare the parallel 2-step tridiagonalization with a runtime estimation of the 1-step approach which can be found in [82]. Afterwards we will analyze our algorithms towards weak and strong scaling.

The presented runtime estimations for the individual stages are very detailed but not suited for a simple comparison. In the following we will try to simplify the formulas as far as possible. At first we assume $p_r = p_c = \sqrt{p}$. To get rid of some terms which result from load imbalances due to the block-cyclic data distribution, we claim $\frac{n}{b} \gg \sqrt{p}$. Furthermore we suppose that $t_{\text{flop}} \ll t_{\text{mem}} \ll t_{\text{word}} \ll t_{\text{msg}}$ such that, e.g. $c_1 t_{\text{flop}} + c_2 t_{\text{mem}}$, will be simplified to $c_2 t_{\text{mem}}$. In doing so we get the following runtime estimation for the tridiagonalization and back transformation using the 2-step approach:

$$\begin{aligned} t_{\text{trd\_2step}} \approx & \frac{4n^3}{3p} t_{\text{flop}} + \left(\frac{2n^3}{3pb} + 6nb^2 \right) t_{\text{mem}} \\ & + \lceil log(\sqrt{p}) \rceil \frac{9n^2}{2\sqrt{p}} t_{\text{word}} \\ & + \left(\lceil log(\sqrt{p}) \rceil \left(\frac{5n}{blk} + \frac{9n}{b} \right) + 3n \right) t_{\text{msg}}, \end{aligned} \tag{3.50}$$

$$\begin{aligned} t_{\text{back\_2step}} \approx & \frac{4kn^2}{p} t_{\text{flop}} + \frac{2kn^2}{pb} t_{\text{mem}} \\ & + \left(\lceil log(\sqrt{p}) \rceil \frac{n^2}{\sqrt{p}} + \frac{kn}{\sqrt{p}} \left(\lceil log(\sqrt{p}) \rceil + 3 \right) \right) t_{\text{word}} \\ & + \frac{n}{b} \left(\lceil log(\sqrt{p}) \rceil + 3 \right) t_{\text{msg}}. \end{aligned} \tag{3.51}$$

For the reduction to banded form we assume to use the blocked QR-decomposition. Furthermore we assume $p \geq \frac{n}{b}$, since this simplifies the runtime estimation of the band reduction.

3.7.1 Comparison with 1-step tridiagonalization

The runtime estimation, presented in [82], describes the execution time of the ScaLAPACK (version 1.5) routines PDSYTRD (tridiagonalization) and PDORMTR (back transformation) which are based on the 1-step approach. It uses a different performance model. The execution time of computations is modeled using the performance of BLAS routines. In doing so, γ_1, γ_2 and γ_3 stand for the time per flop within BLAS1, BLAS2 and BLAS3 routines, δ_1, δ_2 and δ_3 stand for the software overhead to call the corresponding BLAS routines. The performance modeling of communication is the same as in our model.

To transform the runtime estimation of the 1-step tridiagonalization into our performance model, we use the following conversion:

$$\begin{aligned}\gamma_1 &= t_{\text{flop}} + 2t_{\text{mem}},\\ \gamma_2 &= t_{\text{flop}} + t_{\text{mem}},\\ \gamma_3 &= t_{\text{flop}}.\end{aligned}$$

The software overhead δ_1, δ_2 and δ_3 cannot be described with our model and is thus ignored.

Using the same simplifications, we get the following runtime estimation for the 1-step tridiagonalization and back transformation. This time b stands for the intern blocking factor of the algorithms.

$$\begin{aligned}t_{\text{trd\_1step}} \approx\ & \frac{4n^3}{3p} t_{\text{flop}} + \frac{2n^3}{3p} t_{\text{mem}}\\ & + \frac{n^2}{\sqrt{p}} \left(5 \lceil log(\sqrt{p}) \rceil + \frac{5}{2} \right) t_{\text{word}}\\ & + \left(18n \lceil log(\sqrt{p}) \rceil + n \right) t_{\text{msg}},\end{aligned} \tag{3.52}$$

$$\begin{aligned}t_{\text{back\_1step}} \approx\ & \frac{2kn^2}{p} t_{\text{flop}} + \frac{n^2 b}{\sqrt{p}} t_{\text{mem}}\\ & + \left(\frac{2kn}{\sqrt{p}} \lceil log(\sqrt{p}) \rceil + \frac{n^2}{\sqrt{p}} \right) t_{\text{word}}.\end{aligned} \tag{3.53}$$

The runtime comparison between the 1-step and 2-step approach confirms some well understood facts regarding the number of required flops and their memory efficiency and shows some interesting differences in the communication requirements which may explain, to some extent, the different scaling behavior.

As introduced in Ch. 2, the runtime estimations confirm the better cache efficiency of our 2-step implementation during the reduction to tridiagonal form. The number of required flops remains the same (according to the defined assumptions). The $6nb^2t_{\text{mem}}$ term, resulting from the reduction from banded to tridiagonal form, may become a bottleneck if strong scaling is desired. However, for this configuration the working set should be small enough to fit into the first- or second-level cache. Regarding network bandwidth requirements, both implementations are comparable. A big difference can be observed when looking at the number of required messages. Several blocking possibilities allow the 2-step implementation to aggregate data into larger messages. This should especially pay off if the number of processes is high, compared to the problem size.

As expected, the 2-step back transformation requires twice the amount of flops compared to the 1-step approach. Obviously, it is advantageous for the 2-step approach if only a fraction of the eigenvectors is required. Both algorithms are cache efficient with a memory reuse rate of b. The communication requirements are comparable and won't be a problem for both, the 1-step and the 2-step approach.

Altogether, the 2-step tridiagonalization in combination with the blocked QR-decomposition is able to eliminate the bottlenecks of the 1-step approach (memory bandwidth and network latency) and is, hence, well prepared for coming hardware developments. Measurements in Ch. 4 will confirm this statement.

3.7.2 Strong scaling and efficiency

Strong scaling describes how the runtime for a fixed problem size evolves for different numbers of processes. This type of scaling is especially important for the use in quantum chemistry (see Sect. 1.2.1) where relatively small eigenproblems have to be solved thousands of times.

When looking at the runtime estimation of our algorithms we notice that there are terms which scale linear in p, some terms scale with $\sqrt{p}$ (p_c and p_r respectively), and some terms don't scale at all. The factor $\lceil log(\sqrt{p}) \rceil$, which appears in some communication terms, is approximated with the constant log. Moreover, we have to consider that between the constants t_{flop}, t_{mem}, t_{word}, and t_{msg} can be several orders of magnitude. To get a feeling for the ratios between t_{flop}, t_{mem}, t_{word}, and t_{msg}, in Table

	t_{flop}	t_{mem}	t_{word}	t_{msg}	Ratio ($t_{\text{flop}} : t_{\text{mem}} : t_{\text{word}} : t_{\text{msg}}$)
BlueGene/P	75ps	0.6ns	12.5ns	3μs	1 : 8 : 160 : 38400
SuperMUC	46ps	1.25ns	64ns	3μs	1 : 27 : 350 : 16500

Table 3.3: Timings for t_{flop}, t_{mem}, t_{word}, and t_{msg} based on the specification of two current HPC systems.

	t_{flop}	t_{mem}	t_{word}	t_{msg}
x_1	$\mathbf{\frac{4}{3}n^3}$	$\mathbf{\frac{2}{3}\frac{n^3}{b}}$	-	-
x_2	$5n^2b$	$\frac{9}{2}n^2$	$\mathbf{\frac{9}{2} \cdot \log \cdot n^2}$	-
x_3	$15nb^2$	$\mathbf{6nb^2}$	$(\frac{7}{2} \cdot log + 3)nb$	$\mathbf{29 \cdot \log \cdot \frac{n}{b} + 3n}$

Table 3.4: Breakdown of the highest order terms according to their scaling behavior. x_1 comprises all terms which scale linear in p. x_2 and x_3 contain terms which scale with $\sqrt{p}$ and 1 respectively. The terms printed in bold are crucial for scalability and efficiency issues.

3.3 we have estimated those timings for a BlueGene/P system and the SuperMUC system (see Sect. 4.2.1) based on the theoretical peak of those systems. As already mentioned in Sect. 1.1, these ratios are expected to change on future systems according to some long time trends.

From the stated point of view we can split up the runtime into

$$x_1\frac{1}{p} + x_2\frac{1}{\sqrt{p}} + x_3.$$

The following scaling and efficiency considerations will be done solely for the reduction to tridiagonal form. The behavior of the back transformation is always better. In Table 3.4 we have listed the highest order terms in x_1, x_2, and x_3 for t_{flop}, t_{mem}, t_{word}, and t_{msg} respectively. For reasons of simplicity we assumed $b = 4 \cdot blk$.

To evaluate the scaling of the tridiagonalization we have to compare the ratios between $\frac{x_1}{p}$, $\frac{x_2}{\sqrt{p}}$, and x_3 for different numbers of processes. To get a statement on the efficiency of the algorithm, in turn, we have to compare the terms containing t_{flop}, t_{mem}, t_{word}, and t_{msg}. As we will see, both properties (scaling and efficiency) are linked together.

Using the freely selectable intermediate bandwidth b we can regulate the memory efficiency of the algorithm. b should be large enough to compensate for the gap between t_{flop} and t_{mem}. Furthermore, a larger b reduces the total number of messages. On the other hand, a larger b increases the effort for the reduction from banded to tridiagonal form which is responsible for the $6nb^2t_{\text{mem}}$ term. As we will see in the next chapter, a b between 32 and 64 is a good choice for current systems. For a given n and b we, then, can compute the number of processes p where (i) $\frac{4n^3}{3p}t_{\text{flop}} = \frac{9 \cdot log}{2}\frac{n^2}{\sqrt{p}}t_{\text{word}}$, (ii) $\frac{4n^3}{3p}t_{\text{flop}} = 6nb^2 t_{\text{mem}}$, and (iii) $\frac{4n^3}{3p}t_{\text{flop}} = (29 \cdot log \cdot \frac{n}{b} + 3n)t_{\text{msg}}$. (i) is the crossover where network communication becomes dominating, (ii) and (iii) are crossover points where non-scaling terms begin to get important.

Accordingly we get

$$\text{(i)}\quad p_1 \approx \frac{n^2}{11 \cdot log^2} \cdot \frac{t_{\text{flop}}^2}{t_{\text{word}}^2}, \tag{3.54}$$

$$\text{(ii)}\quad p_2 \approx \frac{2n^2}{9b^2} \cdot \frac{t_{\text{flop}}}{t_{\text{mem}}}, \text{ and} \tag{3.55}$$

$$\text{(iii)}\quad p_3 \approx \frac{4n^2}{\frac{87 \cdot log}{b} + 9} \cdot \frac{t_{\text{flop}}}{t_{\text{msg}}}. \tag{3.56}$$

Please note that in the model as well as in Table 3.3 are too many sources of inaccuracy for a precise computation of the defined crossover points. However, this section should help to understand how scalability and efficiency are correlated with the problem size, the intern blocking possibilities, as well as the properties of a supercomputing system.

3.7.3 Weak scaling

The weak scaling describes the runtime behavior for a varying number of processes if the problem size per process is constant. Contrary to the strong scaling this is not a unique definition. I.e., it is not defined whether "problem size" refers to the computational work or the required memory. In the following we will analyze the two cases where $\frac{n^3}{p}$ and $\frac{n^2}{p}$ are constant. Therefor we eliminate p from the formulas and analyze which of the remaining terms dominates the total runtime.

Constant memory If we assume $\frac{n^2}{p}$ to be constant we easily see that all remaining terms are linear in n. To be precise, some terms for communication still contain $\lceil log(\sqrt{p}) \rceil$ which was assumed to be constant. Without this simplification the time for communication is of order $n \cdot log(n)$ and, thus, the weak scaling behavior is not perfect but still good.

Constant work For the stricter definition of weak scaling we assume $\frac{n^3}{p}$ to be constant. We can see that for this type of scaling the time spent for the real computational work ($\frac{4n^3}{3}t_{\text{flop}}$) is constant. Other terms for communication or computational overhead have a runtime behavior of $\sqrt{n}$, n or even $n \cdot log(n)$. Hence, using this definition, weak scaling is not attained. It has to be said that this is also the case for all other 2D-parallelized algorithms in dense linear algebra. Only 3D-parallelizations (which currently exist for the matrix multiplication [83] and the LU-decomposition [84]) can show such a scaling behavior.

3.8 Overview of existing implementations

As already mentioned, the parallel symmetric eigenproblem is a very active field of research, tackled by research groups all over the world. In this section we will give an overview of existing libraries and new developments.

The de-facto standard for parallel dense linear algebra is ScaLAPACK, containing also routines for the symmetric eigenproblem. The ScaLAPACK tridiagonalization routines are based on the 1-step approach and will be used for runtime comparisons throughout the next chapter.

The already mentioned libraries PLASMA and DPLASMA are intended to replace LAPACK and ScaLAPACK over the next couple of years or decades. A tridiagonalization routine in PLASMA was published in [85]. The corresponding back transformation of eigenvectors is not yet available. The development of a symmetric eigensolver for DPLASMA is work in progress. Both libraries will use the 2-step tridiagonalization with the TSQR algorithm as QR-decomposition.

PLAPACK and its successor Elemental [86] are the most notable parallel dense linear algebra libraries beside ScaLAPACK. The biggest difference to ScaLAPACK is the 2D cyclic data distribution, whereas the latter uses a 2D block-cyclic distribution. Both libraries provide a symmetric eigensolver based on the 1-step tridiagonalization. Performance results of the symmetric eigensolver in Elemental on a BlueGene/P system can be found in [86].

Lately a group in Japan presented results on their eigensolver library Eigen-K [87] which was developed for the K-computer [88]. Eigen-K contains an optimized implementation of the 1-step tridiagonalization (*eigen_s*). A second approach (*eigen_sx*) reduces the dense symmetric matrix to pentadiagonal form and uses afterwards a D&C algorithm for symmetric banded matrices to solve the pentadiagonal eigenproblem.

4 Implementation and results

In the previous chapter we presented all those algorithmic details whose effects can be reflected by our performance model. In this chapter we will provide extensive performance results. Furthermore, we will provide all details regarding the implementation which are important for the achieved performance. Such things are, e.g., register efficiency of kernel routines or issues regarding data layouts.

We will begin this chapter with a short description of the functionality of the ELPA library (Sect. 4.1). Afterwards, in Sect. 4.2 we describe the test settings by giving an overview of the hardware and the matrices, we use for our measurements. The main part of this chapter comprises Sect. 4.3 to 4.7, containing the mentioned results for each stage of the algorithm together with some hints on the implementation, if necessary. Beside the individual stages of the symmetric eigensolver (reduction to banded form in Sect. 4.3, reduction from banded to tridiagonal form in Sect. 4.4, tridiagonal-to-banded back transformation in Sect. 4.6, and banded-to-full back transformation in Sect. 4.7) this chapter contains also the reduction of banded matrices to narrow banded form. This functionality has been implemented to efficiently tridiagonalize banded matrices with a substantial number of off-diagonals and will be described in Sect. 4.5. Finally in Sect. 4.8 we provide performance results for the whole eigensolver and we try to valuate the performance of ELPA in relation to its competitors.

4.1 The ELPA library

The ELPA library arose from the identically named BMBF project ELPA and is a joint work of Rechenzentrum Garching, Bergische Universität Wuppertal, Fritz-Haber-Institut, Max-Plack-Institut für Mathematik in den Naturwissenschaften, IBM Deutschland GmbH and Technische Universität München. The library is publicly available through a slightly modified LGPL [89].

ELPA provides both, a symmetric eigensolver based on the 1-step approach (`solve_evp_real`) and an eigensolver based on the presented 2-step tridiagonalization (`solve_evp_real_2stage`). The eigensolver, based on the direct tridiagonalization is

Name	Description
`tridiag_real`	Reduces a symmetric matrix to tridiagonal form using the 1-step approach.
`trans_ev_real`	Back transformation of eigenvectors, corresponding to `tridiag_real`.
`solve_tridi`	Tridiagonal eigensolver based on the D&C method. The solver has been published in [47] and has the ability to compute a fraction of the eigenspectrum at reduced costs.
`bandred_real`	Reduces a symmetric matrix to banded form.
`tridiag_band_real`	Reduces a banded matrix to tridiagonal form.
`trans_ev_tridi_to_band_real`	Back transformation of eigenvectors, corresponding to `tridiag_band_real`.
`trans_ev_band_to_full_real`	Back transformation of eigenvectors, corresponding to `bandred_real`.
`band_band_real`	Reduces a banded matrix to narrow banded form.

Table 4.1: Listing of subroutines within ELPA.

an optimized variant of the symmetric eigensolver PDSYEVD in ScaLAPACK. A description can be found in [90]. The structure of the solvers is modular, with routines for each individual stage. These are listed in Table 4.1. All listed routines, beside the band to narrow band reduction, are also available for complex arithmetics.

To valuate the performance of our 2-step eigensolver we will use both, ScaLAPACK to compare against the de-facto standard for symmetric eigensolvers on distributed memory systems and the 1-step solver of ELPA to compare against an alternative algorithmic approach with an implementation of comparable quality.

The reduction of banded matrices to narrow banded form (`band_band_real`) is not part of the symmetric eigensolvers. This routine can be used to perform a 2-step reduction of banded matrices to tridiagonal form. This proceeding is beneficial if we have eigenproblems where we start with large banded matrices (see Sect. 4.5).

4.2 Test settings

4.2.1 Hardware overview

The following systems are primarily used for our performance measurements. To be able to interpret the performance results, we will briefly describe the architectures.

BlueGene/P For our measurements we use the BlueGene/P at the Rechenzentrum Garching (RZG) for runs up to 16384 cores. All runs with more cores are done on the BlueGene/P at the Forschungszentrum Jülich (JUGENE) consisting of up to 73728 compute nodes and 294912 cores. Each node is based on one PowerPC 450 with four cores running at 850MHz. Each node has a theoretical peak performance of 13.6GFlops and a memory bandwidth of 13.6GB/s. The nodes are connected over a three-dimensional torus network with a bidirectional bandwidth of 425MB/s per link. The additional tree-network for collective communication is only available for global communicators and will, thus, not be used for our algorithms.

Power6 The Power6 system at the RZG consists of 205 compute nodes with 16 Power6 chips each. Each Power6 chip is a dual-core processor and runs at 4.7GHz. This leads to a theoretical peak performance of 18.8GFlops per core. The Power6 cores have a relatively low memory bandwidth of 4GB/s but a big (4MB) and powerful (70GB/s) L2-cache. The nodes are connected over a 8-link DDR-Infiniband interconnect. Within each node we can find a complex hierarchical network, also based on DDR-Infiniband.

Nehalem cluster The Nehalem cluster, provided by the Fritz Haber Institute, is based on dual-socket Nehalem nodes. The nodes are connected over QDR-Infiniband. Each node consists of two Xeon 5570 processors, running at 2.93GHz. The four cores of the Xeon 5570 share 8MB of L2-cache and a memory bandwidth of 32GB/s. Each core is capable to execute two SSE instructions per clock which leads to a theoretical peak of 11.7GFlops.

SuperMUC / SuperMIG The SuperMUC is the new high-end system at the Leibniz-Rechenzentrum and currently the fourth fastest supercomputing system in the world. The system consists of dual-socket Sandy Bridge nodes, connected over FDR10 Infiniband. The topology of the communication network is a fat tree with one link per node on the bottommost level of the tree. In this way, 512 nodes are connected to a so

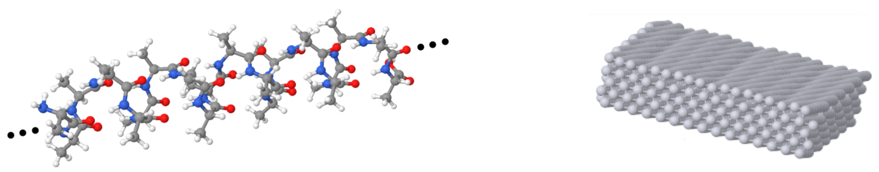


Figure 4.1: α-helical polyalanine molecule [47] (left) and platinum-5x40 surface slab [91] (right).

called island. On the next and last tree level, 18 of such islands are connected among themselves with an average of four nodes per FDR10 Infiniband link. Each node is equipped with 32GB of main memory and two eight-core Xeon E5-2680. Each core is capable to execute eight double precision floating point operations (two AVX instructions) per clock. For the measurements the cores clocked with 2.3GHz, although the Xeon E5-2680 is specified for 2.7GHz. The turbo mode of the CPUs was disabled.

The SuperMIG is the migration system of the SuperMUC, consisting of 205 Westmere-EX nodes with four Xeon E7-4870 each. The nodes are connected over a star topology with one QDR Infiniband link per node.

4.2.2 Test matrices

Beside random matrices of various size, we use matrices from real scientific problems for our performance measurements. The matrices Poly27069 and Pt67990 refer to two problems from quantum chemistry and will be used for evaluations throughout this chapter. Poly27069 is a matrix of size 27069 where the 3410 lowest eigenvectors have to be computed and describes a α-helical polyalanine molecule with 1000 atoms (Figure 4.1, left). The matrix Pt67990, describing a platinum-5x40 surface slab (Figure 4.1, right), is a matrix of size 67990 whose 43409 lowest eigenvectors are needed. These matrices represent typically sized problems which are currently computed with FHIaims. Due to the different fraction of required eigenvectors (12.6% for Poly27069 and 63.8% for Pt67990), these problems are well suited for a comparison with the 1-step tridiagonalization.

For the evaluation of the reduction of banded matrices to narrow banded form (Sect. 4.5) we use huge matrices from the field of network analysis. Table 4.2 shows size and bandwidth of these matrices. The matrices represent the road network of several federal states and have been brought to banded form by a proper reordering. Due to their huge size of more than one million, these matrices are very difficult to tackle.

name	matrix size	bandwidth
rap	79872	576
rp1088	1088092	2946
rt1379	1379917	4157
rc1965	1965206	5427

Table 4.2: Listing of the used banded matrices.

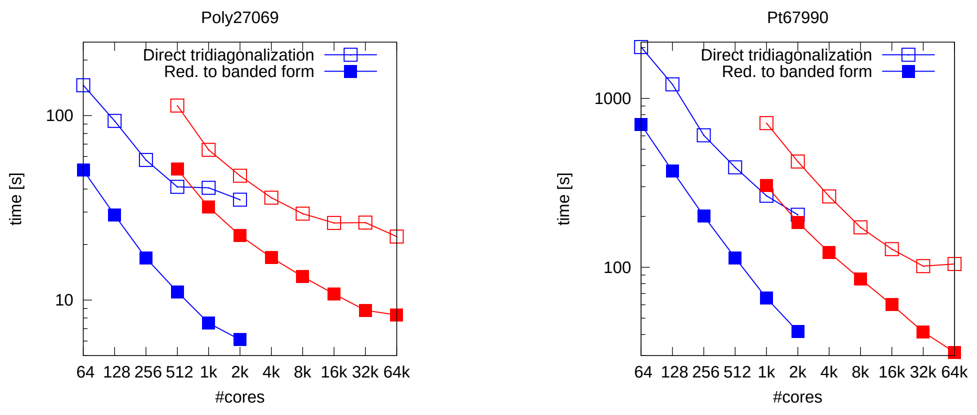


Figure 4.2: Strong scaling of the reduction to banded form for Poly27069 and Pt67990 (intermediate bandwidth $b = 64$). Blue lines: Nehalem cluster, red lines: BlueGene/P.

4.3 Reduction to banded form

In Figure 4.2 we compare the strong scaling behavior of the reduction to banded form with the direct tridiagonalization within the ELPA library. As expected the reduction to banded form shows both, better absolute performance and better scaling. As QR-decomposition the classic unblocked approach is used.

In Table 4.3 we examine the importance of the QR-decomposition during the reduction to banded form of Poly27069. We can see that the QR-decomposition gets increasingly relevant with higher numbers of processes. While this part of the algorithm requires about 20% of the time on 512 cores of the BlueGene/P and on 256 cores of the Power6 system, this ratio grows to 35% on 4096 BlueGene/P cores and to 49% on 2048 cores of the Power6. The results clearly show that the QR-decomposition is the limiting factor of the reduction to banded form if high scalability is required. The bottleneck QR-decomposition is more pronounced on the Power6. This may be explained with the higher single-core performance of the Power6 which implies that network latency issues become even more relevant.

	BlueGene/P [#cores]				Power6 [#cores]			
	512	1024	2048	4096	256	512	1024	2048
Full to band	53.6s	33.0s	22.5s	17.0s	18.0s	12.9s	8.0s	6.1s
thereof QR	10.8s	9.1s	6.3s	5.9s	3.5s	3.2s	3.1s	3.0s
	20.1%	27.6%	28.0%	34.7%	19.4%	24.8%	38.8%	49.2%

Table 4.3: Absolute and relative effort of the QR-decompositions during the reduction to banded form (intermediate bandwidth $b = 32$) of Poly27069.

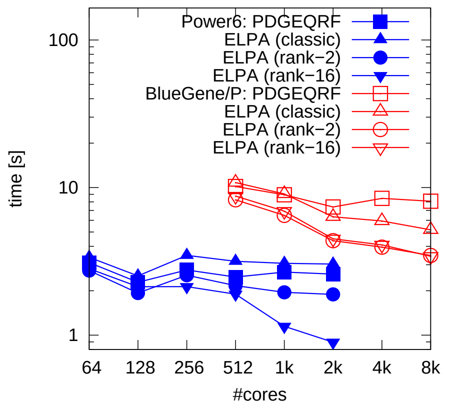
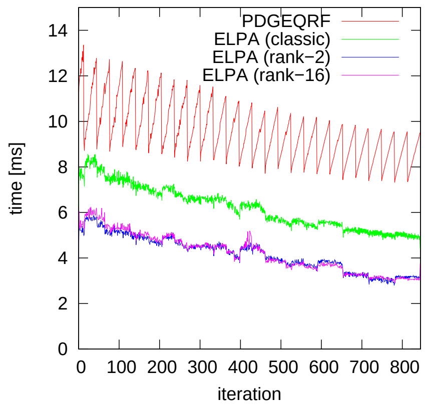


Figure 4.3: Accumulated runtime of the QR-decompositions during the reduction to banded form of Poly27069.

In Figure 4.3 we compare different implementations of the QR-decomposition. The plot shows the accumulated runtime of all QR-decompositions during the reduction to banded form of Poly27069. Due to the tininess of the problems in relation to the number of processes, the scaling is expectably poor. However, the blocked QR-decomposition leads to noticeable speedups, whereas the unblocked algorithms don't scale at all.

In Figure 4.4 we split up the accumulated runtime into timings for each iteration of the algorithm. The reduction to banded form of Poly27069 consists of $\frac{n}{b} - 1 = 845$ iterations, whereas in each iteration i a matrix of size $(n - ib - b) \times 32$ has to be decomposed. The gap between the individual variants seems to be rather constant over the iterations of the algorithm. This means, in other words, that the speedup of the blocked QR-decomposition is higher for smaller matrices. Again, the blocked QR-decomposition seems to be more profitable on the Power6, compared to the Blue-Gene/P.

Up to now we have solely looked at the performance of the blocked QR-decomposition by disabling any fallbacks and ignoring the accuracy of the results. In the next plots



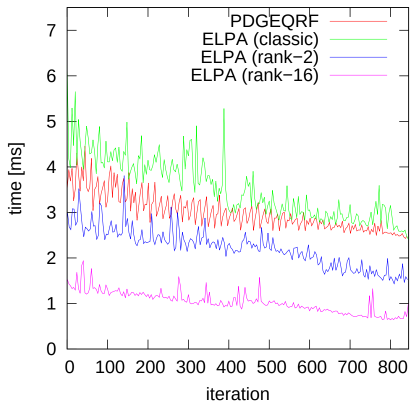


Figure 4.4: Timings of the occurring QR-decompositions during the reduction to banded form of Poly27069. In iteration i a QR-decomposition of a matrix of size $(n - ib - b) \times 32$ is performed. Left: BlueGene/P with 8192 cores, right: Power6 with 2048 cores.

(Figure 4.5 and 4.6) we will analyze accuracy issues and blocking statistics.

For the measurements in Figure 4.5 we construct ill conditioned matrices of the following form. We construct a matrix A by multiplying Q' and R'. Q' is an arbitrary orthogonal matrix. R' is upper triangular and has entries of 1 on the diagonal and entries of odiag/diag above the diagonal. We compare runtime and accuracy of the results for different values of odiag/diag. We use the orthogonality $(I - Q^T Q)$ and the residual error $(A - QR)$ of the result as measures for the accuracy. In Figure 4.5 we compare the blocked QR-decomposition with no fallback (left), the blocked QR-decomposition with $\epsilon_{\text{fallback}} = 1$ (middle), and the ScaLAPACK QR-decomposition PDGEQRF (right). As expected, the ScaLAPACK QR-factorization produces accurate results for all examined matrices. On the other hand the blocked QR-decomposition with no fallback is very fast. However, if odiag/diag is larger than 1, the algorithm gets numerically unstable. With the conservative choice of $\epsilon_{\text{fallback}} = 1$ we profit from both, the efficiency of the blocked execution if the matrix is well conditioned and the numerical stability of the classic Householder QR-decomposition if the matrix is ill conditioned.

In the next plot we investigate how often such fallbacks to lower blockings occur for real matrices. In Figure 4.6 we can see blocking statistics during the reduction to banded form of Poly27069 (right) and a random matrix of size 27069 (left). The maximal blocking is set to 16. For both matrices the algorithm can perform full blocking most of the time. These are very promising results. However, more testing with a broader set of matrices has still to be done.

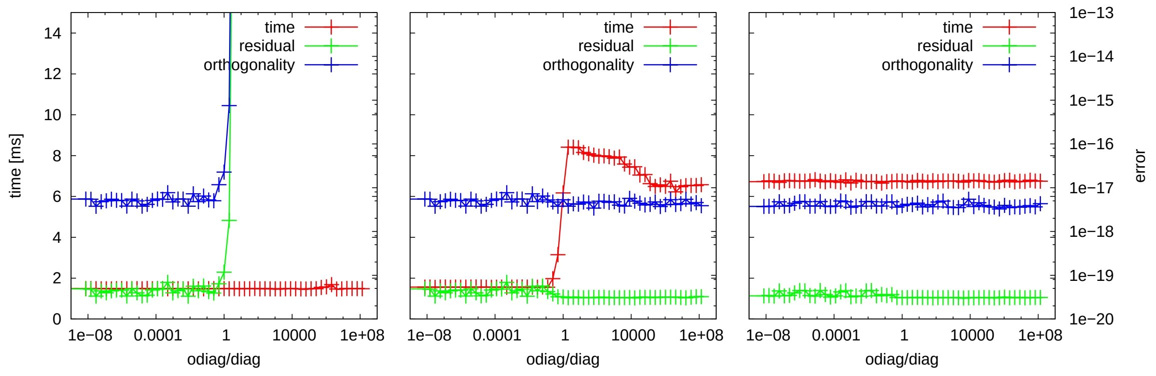


Figure 4.5: Runtime and accuracy while computing the QR-decomposition of $A = QR$, where R has the following structure: values of odiag/diag above the diagonal and values of 1 on the diagonal. Left: blocked QR-decomposition with no fallback, middle: blocked QR-decomposition with $\epsilon_{\text{fallback}} = 1$, right: ScaLAPACK QR-decomposition.

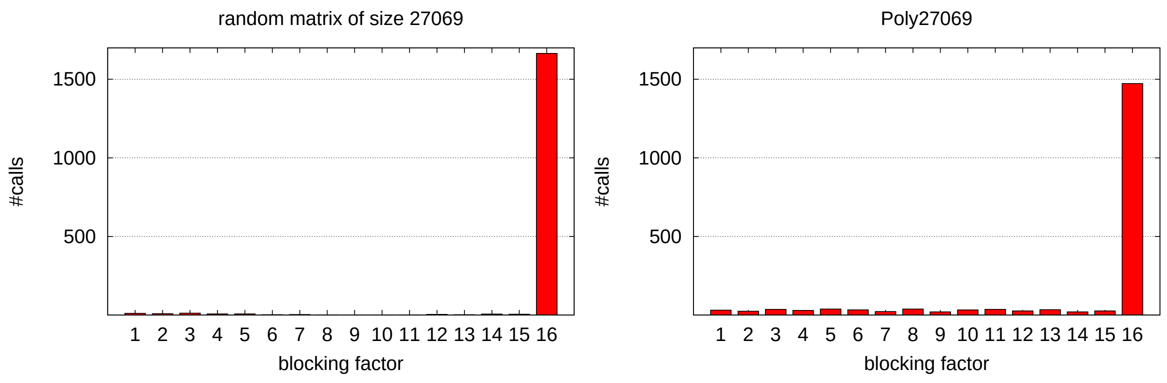


Figure 4.6: Blocking statistics of the blocked QR-decomposition during the reduction to banded form. The maximal blocking factor was set to 16. $\epsilon_{\text{fallback}}$ was set to 1. Left: random matrix of size 27069, right: Poly27069.

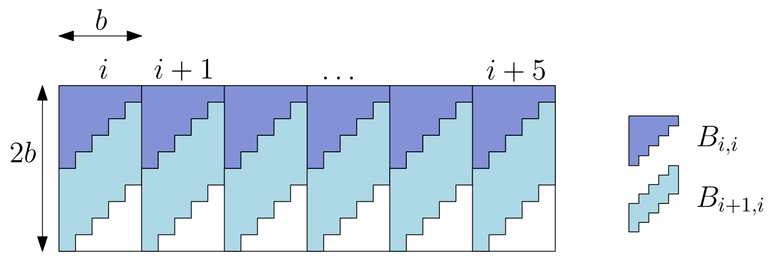


Figure 4.7: Memory layout during the reduction from banded to tridiagonal form.

4.4 Tridiagonalization of banded matrices

Regarding the reduction from banded to tridiagonal form, there are three important things which have not yet been defined:

(i) Memory layout

(ii) Parallel data distribution

(iii) Storage of Householder vectors

Regarding the memory layout we have to consider that the partitioning into block pairs is shifted by one column and one row in each iteration of the algorithm. To avoid unnecessary copying of memory we use the memory layout as depicted in Figure 4.7. The blocks of the banded matrix are stored in a two-dimensional array with column major storage and a leading dimension of $2b$. Thereby the diagonal entries of the matrix are stored at index 1 in each column. At the end of the array has to be reserved some extra space to shift the matrix without the need to copy the whole matrix in each iteration of the algorithm.

The second issue arises from the fact that the band reduction is a 1D parallelization, whereas all other stages of the eigensolver require a two-dimensional parallel data layout with the corresponding 2D Cartesian grid of processes. Thus, we have to map a 1D communicator onto the 2D communicator. Figure 4.8 shows the mapping, we use in our implementation. The mapping fulfills two important things. On the one hand, processes which are neighboring in the 1D communicator are also neighboring in the 2D communicator and should, thus, also be physically neighboring. This is important since the algorithm may become network latency bounded on certain systems. The other thing concerns the distribution and storage of the Householder vectors.

The Householder vectors, arising from the band reduction, cannot reside on the processes where they are generated. We can assume that usually we will have much more available processes than we can use during the band reduction ($p = \frac{n}{b}$). If the

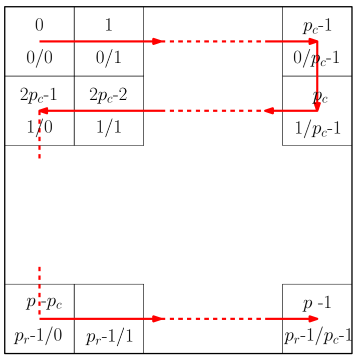


Figure 4.8: Mapping from the 1D communicator to the 2D communicator during the reduction from banded to tridiagonal form. Each rectangle corresponds to a process. The number on the top of each rectangle corresponds to the rank within the 1D communicator, the numbers on the bottom correspond to the Cartesian coordinates within the 2D communicator.

Householder vectors ($O(n^2)$ of data) are not uniformly distributed to the whole grid of processes, we may get a severe memory bottleneck. Additionally, the Householder vectors are, in general, not generated where they will be needed for the back transformation. According to that, we use the following distribution: Let the process p_k in the 1D communicator correspond to process p_i/p_j in the 2D communicator and let process p_k generate all transformations $Q_\beta^{(v)}$ of sweep β. Then, the transformations of sweep β are distributed within the jth column of processes according to the dynamic data layout of the back transformation (see Figure 3.8, Sect. 3.5.3). We can see that the process mapping in Figure 4.8 avoids memory bottlenecks while storing and network bandwidth bottlenecks while distributing the Householder vectors.

Performance measurements

In Figure 4.9 we can see the achieved sequential performance for the reduction from banded to tridiagonal form on the BlueGene/P and the Power6 system. On both systems the performance rises with an increasing bandwidth until saturation is reached. On the Power6 the algorithm definitely profits from the cache. The memory bandwidth of the Power6 ($4GB/s$) allows a theoretical peak of 1GFlops for memory bounded operations, whereas the band reduction reaches up to 4GFlops. For much larger matrix bandwidths we can expect the performance to drop to a lower level. The sequential performance on the BlueGene/P is about one order of magnitude worse.

Figure 4.10 and 4.11 show runtime behavior and speedup for different combinations

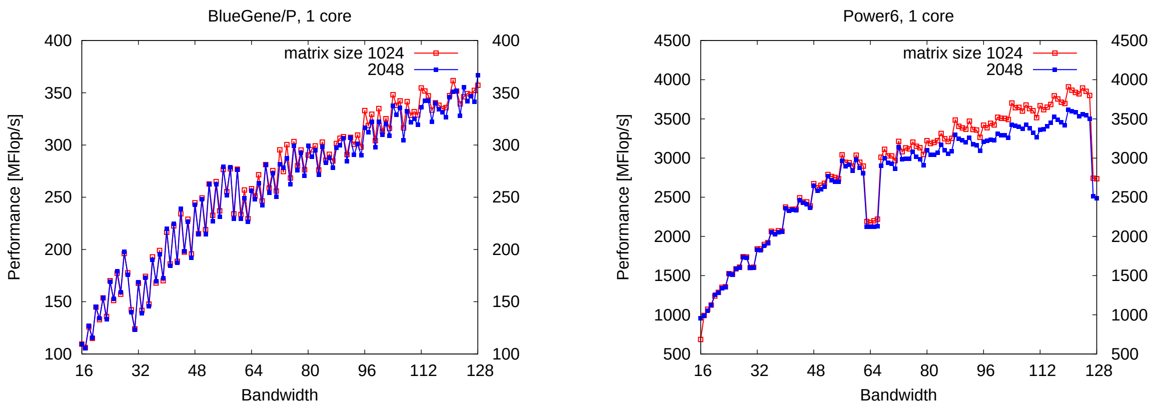


Figure 4.9: Sequential performance of the reduction from banded to tridiagonal form for different bandwidths on BlueGene/P and Power6.

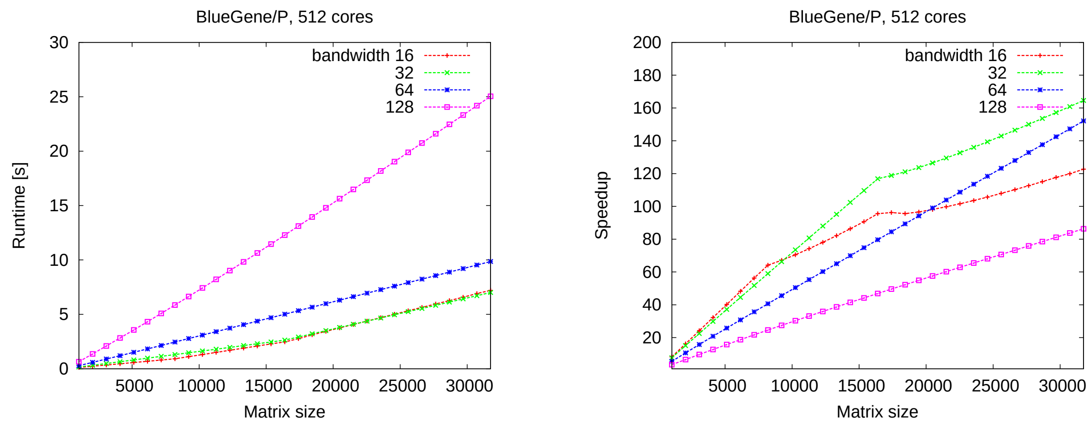


Figure 4.10: Runtime and speedup for different combinations of matrix size and bandwidth on 512 cores of the BlueGene/P.

of matrix size and bandwidth. We use 512 cores of the BlueGene/P (Figure 4.10) and 64 cores of the Power6 system (Figure 4.11). As expected, due to the higher number of flops ($O(n^2b)$) and the limited parallelism of $p = \frac{n}{b}$ processes, a larger matrix bandwidth b leads to a higher runtime. However, on the other side, a smaller bandwidth leads to worse sequential performance. For example, we can see that matrices with a bandwidth of 16 and 32 can be reduced in about the same amount of time, although the latter requires two times as many flops.

When looking at the speedup graph we can observe a linearly increasing speedup up to a matrix size of $n = pb$. Up to this point less and less processes have to idle until at $n = pb$ all processes can be used for the band reduction. For the shown results we fixed the blocksize of the block distribution (see Sect. 3.4) to $l = 1$. Accordingly,

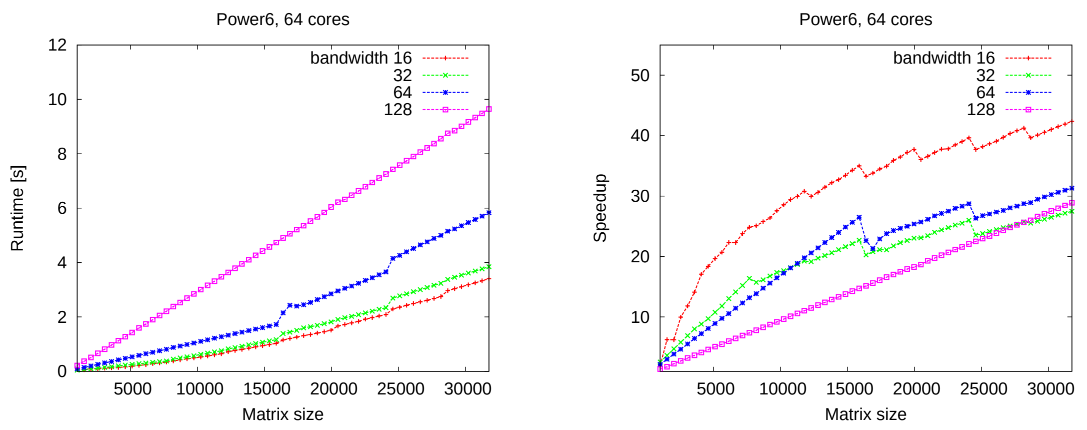


Figure 4.11: Runtime and speedup for different combinations of matrix size and bandwidth on 64 cores of the Power6 system.

the number of cycles c is $\left\lceil \frac{n}{pb} \right\rceil$. Between each salient point in the speedup graph c is increased by one which leads to a further reduction of load imbalance and, thus, also to an increased speedup.

In Figure 4.12 we see strong (left) and weak (right) scaling results on the BlueGene/P. Furthermore we compare between an implementation based on the original algorithm and an implementation where communication is preponed as far as possible.

For the strong scaling we use a matrix size of 30000 and a bandwidth of 30. As predicted, the new variant scales up to twice as many process compared to the variant without early communication. For the weak scaling results we use a bandwidth of 64. The matrix size is set to $n = pb$. Both algorithms show an almost perfect weak scaling behavior. However, the improved variant is nearly twice as fast.

Due to the limited strong scaling, the achievable performance of this stage is limited by the sequential performance (beside of network latency issues). Hence, on systems with a poor single-core performance, such as the BlueGene/P, the band reduction may become the bottleneck for certain problems.

4.5 Reduction of banded matrices to smaller bandwidth

The tridiagonalization of banded matrices is optimized for small bandwidths since for the 2-step tridiagonalization we can use arbitrary intermediate bandwidths and small

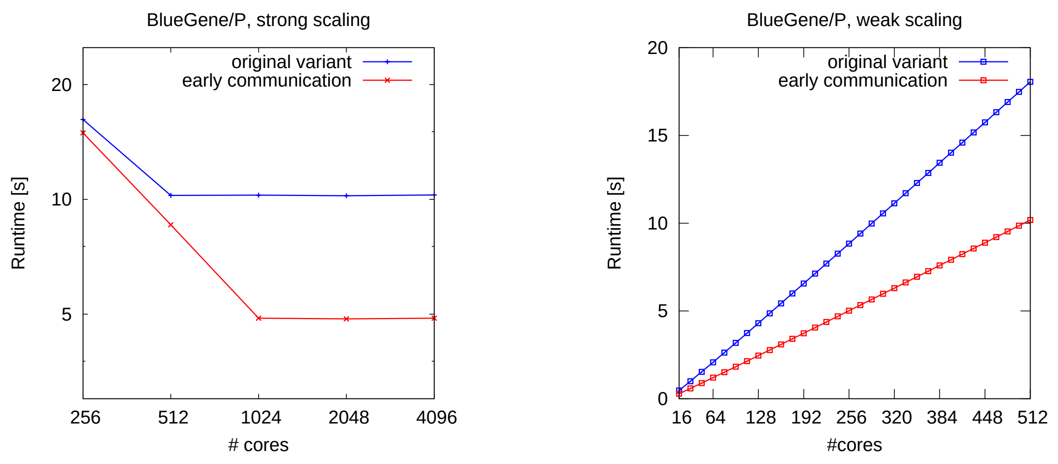


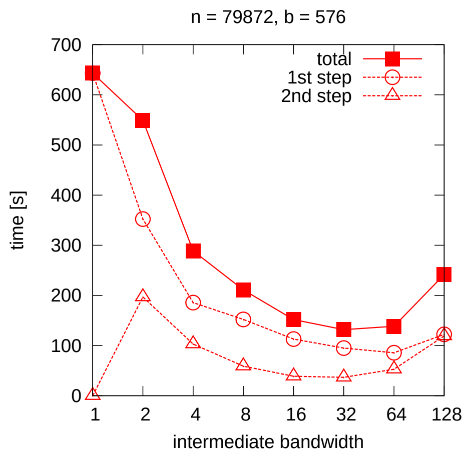
Figure 4.12: Strong and weak scaling of the reduction from banded to tridiagonal form on the BlueGene/P. The strong scaling results were performed on a matrix of size 30000 with a bandwidth of 30. For the weak scaling results we used a bandwidth b of 64. The matrix was of size $\#cores \times b$.

values in the range of 32 to 64 have led to the best results. For these bandwidths the algorithm can profit from cache effects and the parallelizability is quite high. Altogether, this is a satisfying solution for the tridiagonalization of dense symmetric matrices.

However, there exist problems (such as for the examination of large networks, see Sect. 1.2.2), where the initial matrix can be transformed into a banded matrix. The bandwidths of those matrices can be substantial but are still small enough to profit from the reduced computational complexity of $6n^2b$. For this type of problem the direct tridiagonalization is very inefficient, because memory bounded.

To tackle these problems we extended the functionality of ELPA by a reduction of banded matrices to narrow banded form which allows a multi-step band reduction. Please note that this proceeding is applicable if only eigenvalues are needed. The computation of eigenvectors is still very costly ($O(n^3)$) and, thus, a multi-step back transformation is not provided by ELPA.

The concept of the band to narrow band reduction was introduced in Sect. 2.5.3. Algorithm 17 shows the corresponding proceeding. For the respective parallel implementation we modified the parallel band reduction such that Householder operations are replaced by their blocked variants and the resulting Householder vectors are not stored for the back transformation.



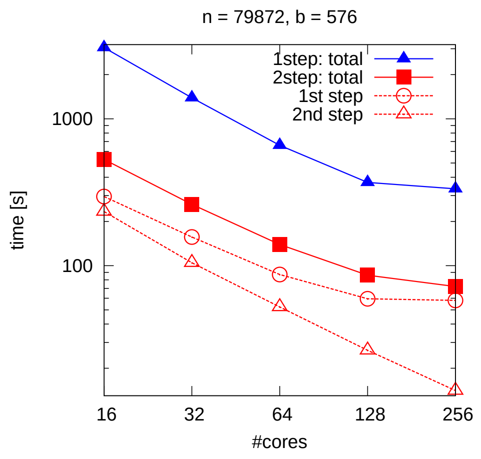


Figure 4.13: Runtimes for the tridiagonalization of banded matrices on the Power6 system. (left) Runtimes depending on the intermediate bandwidth of the 2-step approach. All runs were done using 64 processor cores. An intermediate bandwidth of 1 corresponds to the 1-step approach. (right) Performance comparison of the 1-step and 2-step tridiagonalization. For the 2-step implementation an intermediate bandwidth of 64 was used.

Performance measurements

In Figure 4.13 (left) we can see how the size of the intermediate bandwidth affects the runtime for tridiagonalizing a matrix of size 79872 with a bandwidth of 576. The first step (reduction to narrow banded form) speeds up very quickly if the intermediate bandwidth is increased. The second step was already investigated in the previous section and shows again the best performance for the bandwidths 16 and 32. Altogether we get optimal results for an intermediate bandwidth in the range of 32 to 64.

In Figure 4.13 (right) we see strong scaling results for the same matrix and an intermediate bandwidth set to 64. The direct tridiagonalization as well as the first step of the 2-step band reduction stop scaling at 128 cores, due to the limited parallelizability of the problem. Meanwhile, the second step (reduction from intermediate bandwidth to tridiagonal form) shows nearly perfect scaling. Altogether, the 2-step band reduction is more than four times as fast, compared to the direct tridiagonalization. Although, the 1-step reduction may still profit from the large L2-cache of the Power6.

In Table 4.4 we show runtime results for the larger problems rp1088, rt1379, and rc1965. Thereby rp1088 is computed with 128 processes, rt1379 and rc1965 are computed with 256 processes. One run is performed with one thread per process. A second run is performed with four threads per process (still one thread per core) using

		1-step		2-step	
		1 Thread	4 Threads	1 Thread	4 Threads
rp1088, 128 processes	1st step			9.7h	4.8h*
	2nd step			1.6h	1.6h
	total	174.2h*	132.3h*	11.3h	6.4h*
rt1379, 256 processes	1st step			16.7h	6.1h*
	2nd step			0.7h	0.7h
	total	301.8h*	289.5h*	17.4h	6.8h*
rc1965, 256 processes	1st step			37.8h*	14.8h
	2nd step			2.5h	2.5h
	total	831.6h*	983.5h*	40.3h*	17.3h

Table 4.4: Runtime for the tridiagonalization of the matrices rp1088, rt1379 and rc1965 on the Power6 system. Values denoted with a * are projections based on the first two hours of runtime. All other values come from exact measurements.

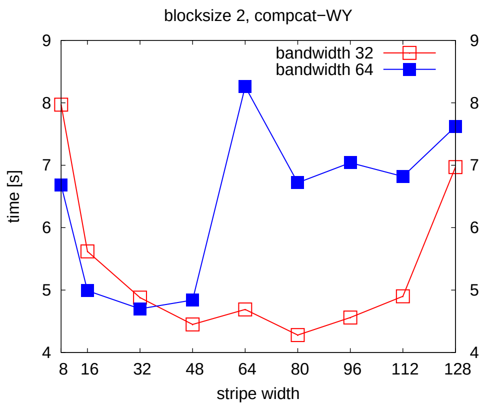
multithreaded BLAS. Due to the immense computing time, the runtime of the direct tridiagonalization could only be projected. We can observe tremendous performance gaps between the 1-step and the 2-step band reduction. The 2-step reduction is 15 to 20 times faster than the direct tridiagonalization. Moreover, the latter doesn't profit from the use of multithreaded BLAS operations, whereas the 2-step approach speeds up by a factor of 1.75 to 2.55 if four threads per process are used. Altogether, the reduction from banded to narrow banded form allows the efficient tridiagonalization of matrices which, otherwise, would require an enormous amount of time and computing resources. To our knowledge, our method is the first distributed memory implementation of a band to narrow band reduction.

4.6 Tridiagonal-to-banded back transformation

4.6.1 High performance kernels

The basic idea of the non-WY approach for the tridiagonal-to-banded back transformation was already presented in Sect. 3.5.1. Here we want to depict the detailed implementation and present and interpret the performance results.

The problem is to apply m Householder transformations to k eigenvectors of size n. The Householder vectors have b nonzero elements (except of the first $b-2$ vectors which have less nonzero elements) and are shifted such that the ith Householder vector has



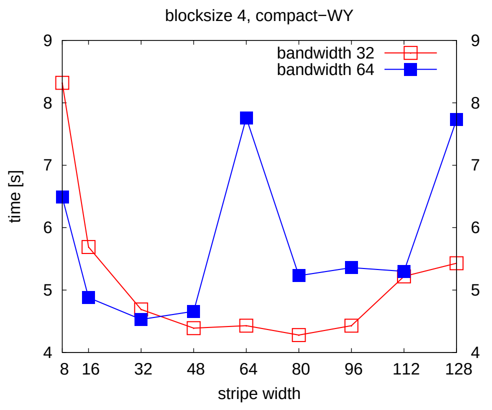


Figure 4.14: Runtime of the tridiagonal-to-banded back transformation depending on the width k_b of the loop blocking and the blocking factor of the compact WY transformations for two different intermediate bandwidths 32 and 64. BlueGene/P with 512 cores. Eigenvector matrix of size 10000×10000.

nonzero elements from $i-1$ to $i+b-2$. The order of application of the Householder transformations is shown in Figure 3.4 (Sect. 3.5), right.

The first technique to achieve high performance is a simple loop blocking over the number of eigenvectors. The loop blocking has been described in Sect. 3.5.1 and can be implemented as shown in Algorithm 31. The blocking size k_b (stripe width) for the loop over the eigenvectors has to be chosen large enough to increase the cache reuse for the access to Householder vectors and small enough such that the working set of bk_b words fits into the L1-cache. The Householder transformations are, finally, applied using compact WY transformations with a blocksize of 2 or 4 (the corresponding kernels will be presented later in this section). In Figure 4.14 we see the runtime of the tridiagonal-to-banded back transformations for blocking sizes k_b from 8 to 128. In the figure on the left we use a blocking factor of 2 for the compact WY representation. In the figure on the right we apply 4 Householder transformations at a time. As expected, up to a certain point, a larger stripe width leads to better performance. Once the stripe width is too large, the performance drops to a lower level. With a bandwidth b of 64 this happens earlier ($k_b > 48$) as for a bandwidth of 32 ($k_b > 96$). This correlates well to the L1-cache size of the PowerPC450 (32KB). We can also observe that the performance drop between "working set fits into the cache" and "working set doesn't fit into the cache" is more pronounced for the 2-fold compact WY transformations than for the 4-fold compact WY transformations. Using compact WY transformations with a blocking factor of 2 or 4 guarantees a cache reuse of 2 and 4 respectively. This means that the variant with 4 Householder transformations is

	4 eigenvectors	8 eigenvectors	12 eigenvectors
2-fold compact WY	14	26	38
4-fold compact WY	24	44	64

Table 4.5: Register requirements for the different kernels.

less dependent on the memory- or higher level cache-bandwidth. The peaks at stripe width 64 and 128 for the measurements with a bandwidth of 64 are reproducible and may be caused by cache conflicts.

The loop blocking guarantees a high cache reuse for the whole algorithm. However, on some architectures the bandwidth to the L1-cache is not sufficient to fully load the floating point units. The PowerPC450 in the BlueGene/P, for example, has a L1-cache bandwidth of 8 Bytes per cycle with 4 cycles of latency. The floating point unit, however, is capable to execute two FMA (fused multiply add) operations in double precision per cycle which requires 32 Bytes of data.

To reduce the bandwidth requirements to the L1-cache we have to keep as many operands as possible in the registers. This technique is called register blocking and is used in our kernel routines. We have implemented several kernels which apply 2 or 4 Householder transformations to 4, 8 or 12 eigenvectors. The number of eigenvectors, which can be transformed with one kernel call, is limited by the number of floating point registers. Table 4.5 shows the minimal number of required registers for the different kernels. For the 4-8 kernel (apply 4 Householder transformations to 8 eigenvectors), for example, this number consists of 4×8 registers to hold the matrix Z (see Algorithm 8, Sect. 2.3.4) and 4 and 8 registers to stream through the Householder vectors and eigenvectors respectively. The kernels differ slightly from the definition of $CWYLeft$ in Algorithm 8. The tridiagonal matrix T is not multiplied with the matrix Y (this would destroy a part of the zero structure) but with the matrix Z. Table 4.6 shows the number of flops, loads, and stores for each different kernel. The costs for the generation of T occur once for every set of Householder vectors and are ignored in Table 4.6.

There exist three variants of the described kernels: a standard implementation in Fortran, an implementation based on the Double Hummer instruction set of the PowerPC 450 and an implementation based on SSE/AVX intrinsics[1]. In Table 4.7, 4.8, and 4.9 we measure the kernel performance on the Power6 and the BlueGene/P as well as on a Intel Westmere and Intel Sandy Bridge system.

[1]Thanks to Alexander Heinecke for implementing the SSE/AVX intrinsic kernels and for providing the results

	2-fold compact WY			4-fold compact WY		
	flops	loads	stores	flops	loads	stores
4 eigenvectors	131	50	17	133	36	9
8 eigenvectors	131	42	17	133	27	9
12 eigenvectors	131	39	17	133	24	9

Table 4.6: Average number of flops, loads and stores to apply a length-32 Householder transformation to one eigenvector using the different kernels. We have counted the number of floating point operations, loads and stores for each kernel and divided these numbers by the number of eigenvectors and the number of Householder transformations.

	4 eigenvectors	8 eigenvectors	12 eigenvectors
2-fold compact WY [MFlops]	4511	5012	4680
4-fold compact WY [MFlops]	5105	-	4509

Table 4.7: Per core kernel performance of the tridiagonal-to-banded back transformation on the Power6 using 4 cores. Eigenvector matrix of size 4096×4096, length-32 Householder vectors.

	4 eigenvectors	8 eigenvectors	10 eigenvectors
2-fold compact WY [MFlops]	958	1137	1162
4-fold compact WY [MFlops]	1131	1190	1030

Table 4.8: Per core kernel performance of the tridiagonal-to-banded back transformation on the BlueGene/P using 512 cores. Eigenvector matrix of size 4096×4096, length-32 Householder vectors.

	Westmere		Sandy Bridge	
	Fortran	SSE Intr.	Fortran	AVX Intr.
2-fold compact WY [MFlops]	5966	6911	10113	12961
4-fold compact WY [MFlops]	-	7201	-	14798
6-fold compact WY [MFlops]	-	7167	-	14313

Table 4.9: Per core kernel performance of the tridiagonal-to-banded back transformation on a Intel Westmere system using 12 cores and a Intel Sandy Bridge system using 4 cores. Eigenvector matrix of size 4096×3192, length-64 Householder vectors.

The 4-fold compact WY kernels show the best performance on all systems if the suitable number of eigenvectors is chosen. The appropriate number of eigenvectors, in turn, correlates with the number of available floating point registers. The Power6, for example, offers 32 double precision floating point registers per core. According to Table 4.5, the best performance is achieved with 8 eigenvectors for the 2-fold compact WY transformations and with 4 eigenvectors for 4-fold compact WY. On the BlueGene/P (32 dual double precision floating point registers), the register requirements are higher than in Table 4.5 because some additional registers are needed for the prefetching of operands. Here, the best performance is achieved with 10 eigenvectors for the 2-fold compact WY transformations and with 8 eigenvectors for the 4-fold compact WY transformations. On the Intel systems we use the kernels which operate on 12 eigenvectors and compare between the Fortran implementation and the implementation based on intrinsics. The intrinsic kernels show significantly better performance, most notably if the 256-bit wide AVX instructions can be used (Intel Sandy Bridge).

Regarding absolute performance, we reach 35% of the peak performance on the BlueGene/P. The properties of the cache hierarchy (write back strategy, low bandwidth) don't allow a better performance. On the Intel systems we reach about 65% of the peak performance.

4.6.2 Parallel performance

In the following we present the parallel performance of the different variants of the tridiagonal-to-banded back transformation (1D/2D, WY/non-WY) and show some interesting results we got with Intel MPI.

In Figure 4.15 we compare the performance of the WY and non-WY approach for different sized Householder vectors. Especially for short vectors, the non-WY approach clearly outperforms the WY technique. This behavior was to be expected because the overhead of the WY approach gets increasingly dominant for short Householder vectors (see Sect. 3.5.1). As we will see later, the optimal intermediate bandwidth b of the whole algorithm is in the range of 32 to 64 on the used architectures. In this range the non-WY approach shows clearly better performance than the WY approach and will be used within the ELPA library. For all following results, where the kernel type is not explicitly defined, we use the non-WY approach.

In Figure 4.16 we compare strong scaling results of the tridiagonal-to-banded back transformation of eigenvectors for the matrices Poly27069 and Pt67990. We can see that the non-WY approach almost always outperforms the WY variant. Moreover, the

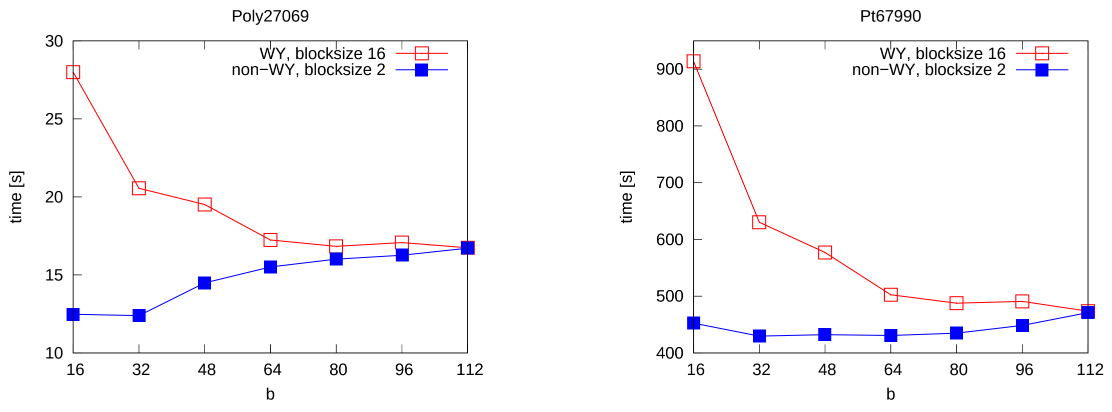


Figure 4.15: Runtime of the tridiagonal-to-banded back transformation with varying intermediate bandwidth b. The WY- and non-WY approach are compared to each other. The problem Poly27069 was computed on a BlueGene/P using 512 cores, Pt67990 was computed on 1024 cores.

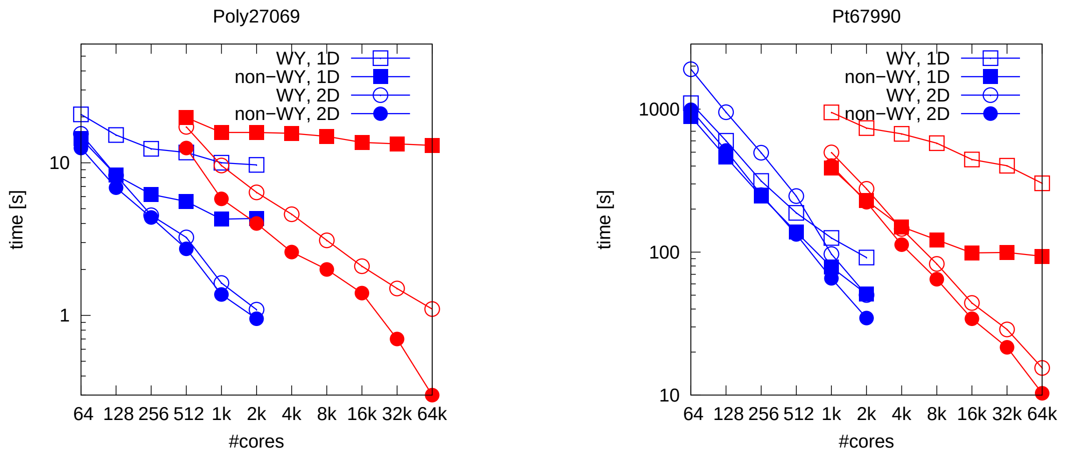
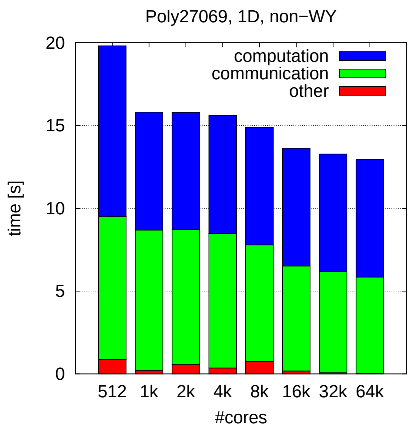


Figure 4.16: Strong scaling of the tridiagonal-to-banded back transformation of eigenvectors for Poly27069 and Pt67990 (intermediate bandwidth $b = 64$) [47]. Blue lines: Nehalem cluster, red lines: BlueGene/P. The WY, 1D times for Poly27069 on BlueGene/P are above the plotting area.



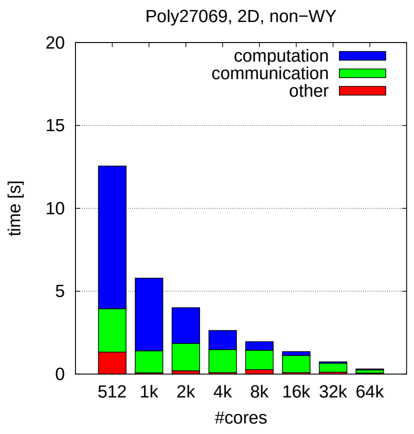


Figure 4.17: Strong scaling of communication and computation during the tridiagonal-to-banded back transformation. Poly27069, BlueGene/P.

gap between WY and non-WY is more pronounced for the 1D parallelization. This is the case because the local matrices are much thinner (and taller) if we use a 1D data distribution and the generation of the matrix W begins to dominate the total runtime. This behavior is also described by our model (see Sect. 3.5.1).

Furthermore, the 2D parallelization scales much better than the 1D approach. For Poly27069 the 1D parallelization stops scaling at 1024 cores on the BlueGene/P and is far from ideal on the Nehalem cluster. As expected, the scaling is much better for the bigger problem size Pt67990 but still not satisfying on the BlueGene/P. The 2D parallelization scales up to the total number of available cores (65536 on the BlueGene/P and 2048 on the Nehalem cluster) for both problems.

In Figure 4.17, 4.18, 4.19 and 4.20 we profile the strong scaling results from Figure 4.16 and distinguish between time for computation, communication and other operations. Computation time contains the pure kernel times, communication time contains all calls to MPI operations including idle waiting and "other" contains all the rest including, e.g. a change of the parallel data layout at the beginning and at the end of the tridiagonal-to-banded back transformation. The time is measured on the lowermost processes of the 2D Cartesian grid of processes which are the first processes to start and the last processes to finish the back transformation.

From Figure 4.17 trough 4.20 we can make several observations. The achievable performance of the 1D parallelization is not only limited by the communication (which doesn't scale at all) but also by the kernel time. For Poly27069 we can see a stagnation of runtime on both systems (BlueGene/P and Nehalem cluster) if we use more than 1024 cores. According to the higher number of eigenvectors (3410 vs. 43409), for the

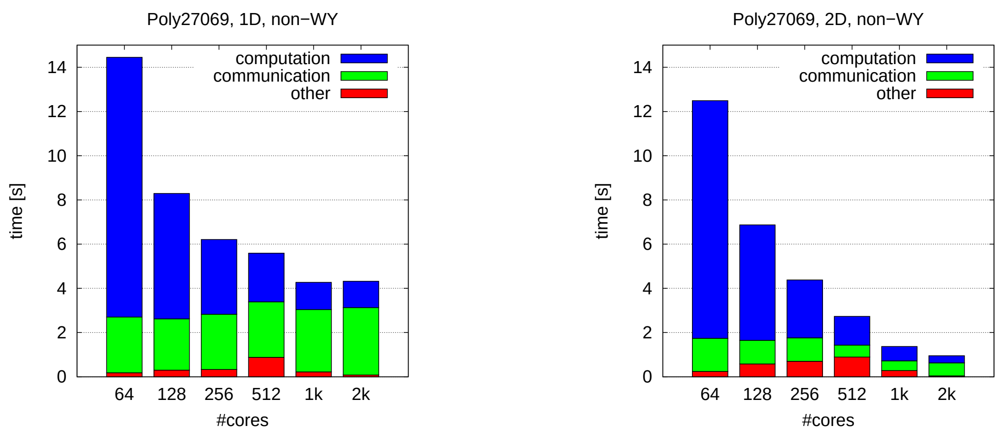


Figure 4.18: Strong scaling of communication and computation during the tridiagonal-to-banded back transformation. Poly27069, Nehalem cluster.

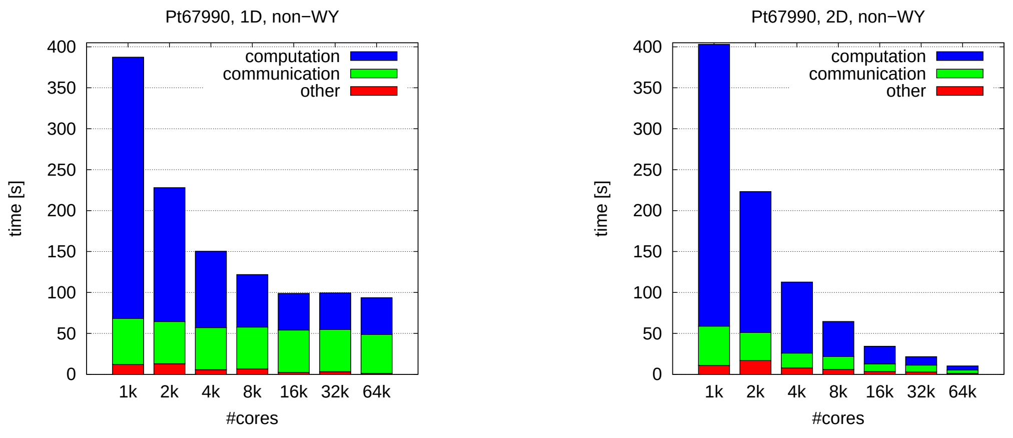


Figure 4.19: Strong scaling of communication and computation during the tridiagonal-to-banded back transformation. Pt67990, BlueGene/P.

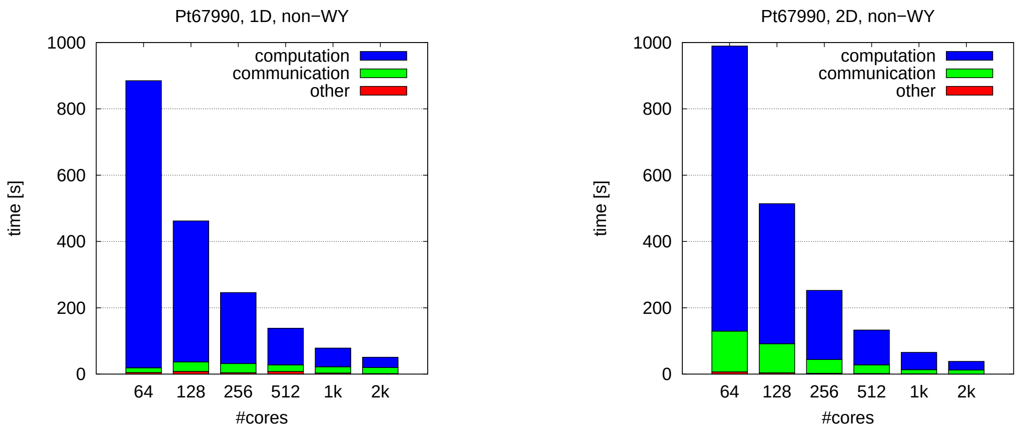


Figure 4.20: Strong scaling of communication and computation during the tridiagonal-to-banded back transformation. Pt67990, Nehalem cluster.

	128 cores	256 cores	512 cores	1024 cores	2048 cores
2D - first results	796s	352s	217s	88s	48s
2D - final results	514s	253s	133s	66s	35s
1D results	457s	245s	142s	78s	50s

Table 4.10: Execution time of the banded-to-tridiagonal back transformation for the Pt67990 problem on the Nehalem cluster.

Pt67990 problem the scaling stops at 16384 cores. In this range the application is communication and memory bounded. The ratio between communication and computation differs from architecture to architecture but is independent from the problem size.

For the 2D parallelization we can observe a scaling, as predicted by our model. Thereby, the computation scales better than the communication such that the overall efficiency drops and the communication begins to dominate the total runtime. However, this behavior is typical for all dense linear algebra applications. Surprising are the communication times of the 2D parallelization in Figure 4.20. Although the communication volume is much smaller than for the 1D parallelization, we spend much more time in MPI operations. These measurements are still not fully understood and may stick together with an unexpected behavior of Intel MPI which will be described in the following.

In our first runs on the Nehalem cluster we got much worse results than those in Figure 4.16. These results are shown in Table 4.10. It turned out that the non-blocking communication of Intel MPI doesn't behave as expected. Depending on the

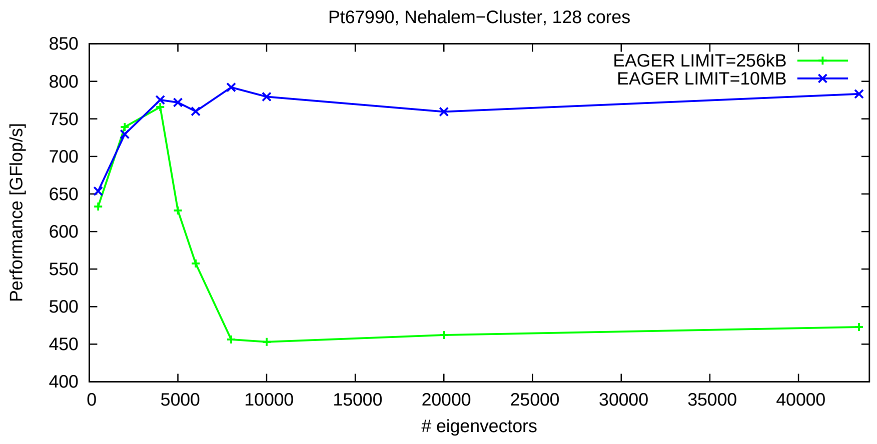


Figure 4.21: Performance of the tridiagonal to banded back transformation for varying numbers of eigenvectors and using different thresholds to switch between eager and rendezvous protocol (Intel MPI). Pt67990, length-64 Householder vectors.

message size, MPI switches between the so-called "eager" and "rendezvous" protocol when sending data from one process to another. The eager-protocol, which is used for small messages, writes data directly into a reserved buffer of the receiving process. The rendezvous protocol, in turn, requires some kind of handshaking between sender and receiver to guarantee that enough buffer space is available when sending larger messages.

In Figure 4.21 we can see the performance of the 2D parallelization on 128 cores of the Nehalem cluster. The back transformation is performed for different numbers of eigenvectors and with two different thresholds for switching between eager and rendezvous protocol. Note that the size of the halo regions and, thus, the size of the exchanged messages is proportional to the number of eigenvectors (kb/p_c). With the default configuration of Intel MPI (EAGER_LIMIT $= 256kB$) we can observe a severe performance drop when increasing the number of eigenvector from 4000 to 5000. A closer look ($b = 64$, $p_c = 8$) reveals that in this region MPI switches from the eager to the rendezvous protocol. After increasing the threshold EAGER_LIMIT, the performance drop disappears.

Although the two protocols can differ in communication performance, this is definitively not the reason for the observed behavior. The absolute time, required to exchange the halo regions, is orders of magnitude smaller than the observed performance loss. Instead, measurements with different kernels have shown that the time loss correlates with the kernel time. The non-blocking communication of Intel MPI seems not

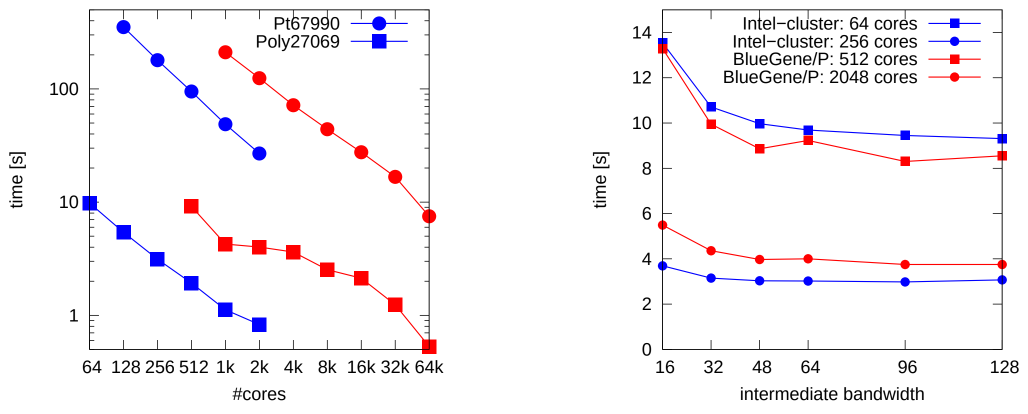


Figure 4.22: Left: strong scaling of the banded-to-full back transformation on the BlueGene/P (red lines) and the Nehalem cluster (blue lines) using an intermediate bandwidth of 64. Right: runtime behavior of the banded-to-full back transformation of Poly27069 using different intermediate bandwidths.

to work as expected which in turn causes idle waiting in our algorithms. Although this is an issue of the MPI implementation, the current ELPA implementation avoids the use of the rendezvous protocol by subdividing the halo regions into different stripes.

4.7 Banded-to-full back transformation

The parallel implementation of the banded-to-full back transformation is entirely based on BLAS3 calls and collective communication operations. Accordingly, the performance results are characterized by the performance of these methods. In Figure 4.22 (left) we analyze the strong scaling behavior for Poly27069 and Pt67990 on the BlueGene/P and the Nehalem cluster. The scaling for the problem Pt67990 is nearly perfect on both systems. We reach 76% and 55% of the theoretical peak on 128 Nehalem cores and 1024 BlueGene/P cores respectively. This values drop to 62% and 24% when going to 2048 Nehalem cores and 65576 cores on the BlueGene/P. For the much smaller problem Poly27069 the stage begins to become network-bandwidth bounded. The performance per core drops from 8.0GFlops (68% of peak) to 2.9GFlops when going from 64 to 2048 Nehalem cores. On the BlueGene/P the performance drops from 1061MFlops (31% of peak) to 144MFlops when going from 512 to 65536 cores. Nevertheless, the scaling is still better than, e.g., the reduction to banded form and won't become a bottleneck for the whole eigensolver.

In Figure 4.22 (right) we examine how the performance of the banded-to-full back transformation depends on the intermediate bandwidth b. We can see that the performance increases for higher intermediate bandwidths. Especially the step from $b = 16$ to $b = 32$ reduces the runtime significantly. For bandwidths greater than 32 we can observe a saturation in the performance graphs.

Altogether, the banded-to-full back transformation is an efficient, well scaling stage, provided that efficient BLAS and MPI implementations are available.

4.8 Overall results

In the previous sections we analyzed the performance and the scaling of the individual stages during the tridiagonalization and back transformation. The results confirm the performance characteristics, predicted by our model. In this section we investigate the overall performance. This includes timings for the reduction to banded form, the reduction from banded to tridiagonal form, the tridiagonal-to-banded back transformation, and the banded-to-full back transformation. Moreover, we will present the performance of existing and planned tridiagonal eigensolvers within ELPA to get a complete picture of the dense symmetric eigensolver.

While looking at the runtime of the whole tridiagonalization and back transformation, we want to resolve a series of questions:

(i) How does the intermediate bandwidth influence the runtime of the algorithm?

(ii) How behave absolute performance and scaling of the individual stages relative to each other?

(iii) How does the two-step tridiagonalization perform, compared to the one-step tridiagonalization in general and to the competing libraries in particular?

As already mentioned, ELPA contains an improved D&C tridiagonal eigensolver which can compute a fraction of the eigenvectors at reduced cost. Table 4.11 shows the runtime of the dense symmetric eigensolver (tridiagonalization + tridiagonal eigensolver + back transformation) and, thereof, the time required for the tridiagonal eigensolver. Our 2-step solver in ELPA is compared to the ScaLAPACK routine PDSYEVD while computing Poly27069 on the Nehalem cluster. For a more detailed performance analysis of this tridiagonal solver we refer to [47]. Table 4.11 shows that the D&C implementation in ELPA scales up to a large number of cores. Although it requires a substantial fraction of the total runtime, usually, it is not the bottleneck of the dense symmetric eigensolver.

cores	64	128	256	512	1k	2k
ELPA, 2-step	92.5s	52.2s	31.3s	19.9s	13.5s	10.4s
thereof D&C	14.6s	8.0s	4.7s	2.5s	2.0s	1.2s
PDSYEVD	239.1s	140.7s	87.7s	74.9s	70.5s	96.6s
thereof D&C	41.1s	28.9s	19.2s	21.8s	25.6s	43.8s

Table 4.11: Total runtime and runtime of the tridiagonal solvers while computing Poly27069 on the Nehalem cluster.

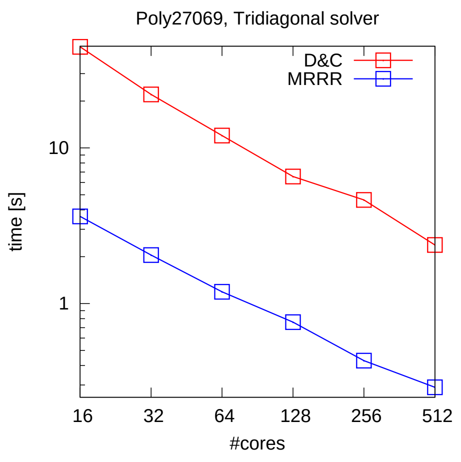


Figure 4.23: Runtime of different tridiagonal eingensolvers while computing Poly27069 on the Power6 system.

Moreover, as alternative to the D&C solver, it is planned to integrate a parallel implementation of the MRRR algorithm into ELPA. Figure 4.23 shows preliminary results on the Power6 system. The implementation is based on the sequential MRRR algorithm of Paul Willems [49]. For the parallelization, the set of desired eigenvectors is split into p equally sized portions. Afterwards, each process calls the sequential MRRR algorithm to compute its part of the eigenspectrum. From Figure 4.23 we can see that this simple parallelization scheme leads to decent speedups up to 512 Power6 cores for Poly27069. The absolute performance is excellent and about one order of magnitude faster than the D&C solver. The achievable accuracy has still to be analyzed.

After giving a short overview on the performance of the tridiagonal solvers, the following results will solely consider the tridiagonalization and back transformation.

In Figure 4.24 and 4.25 we examine how the overall performance and the performance of the individual stages depend on the intermediate bandwidth. The measurements are done on the BlueGene/P and the Nehalem cluster. As we saw in the previous

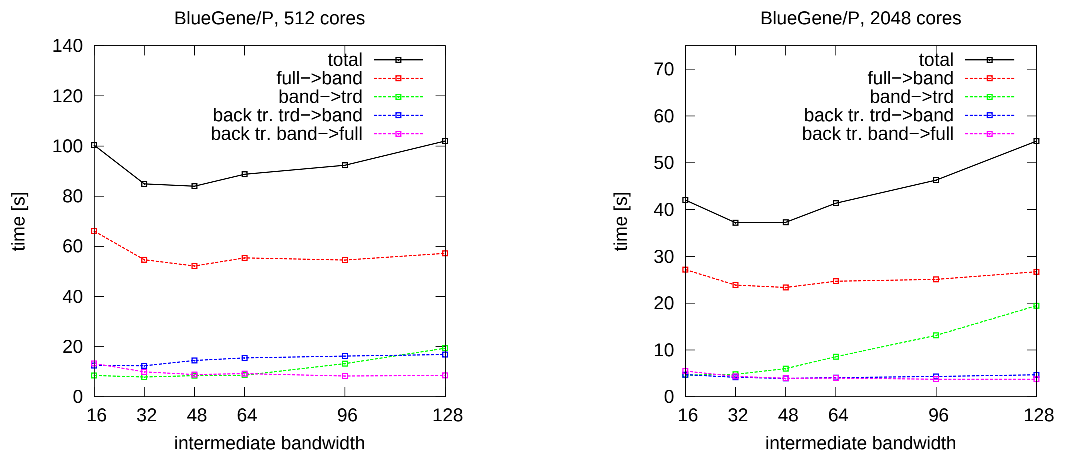


Figure 4.24: Runtime behavior of the 2-step tridiagonalization for Poly27069 on the BlueGene/P using different intermediate bandwidths.

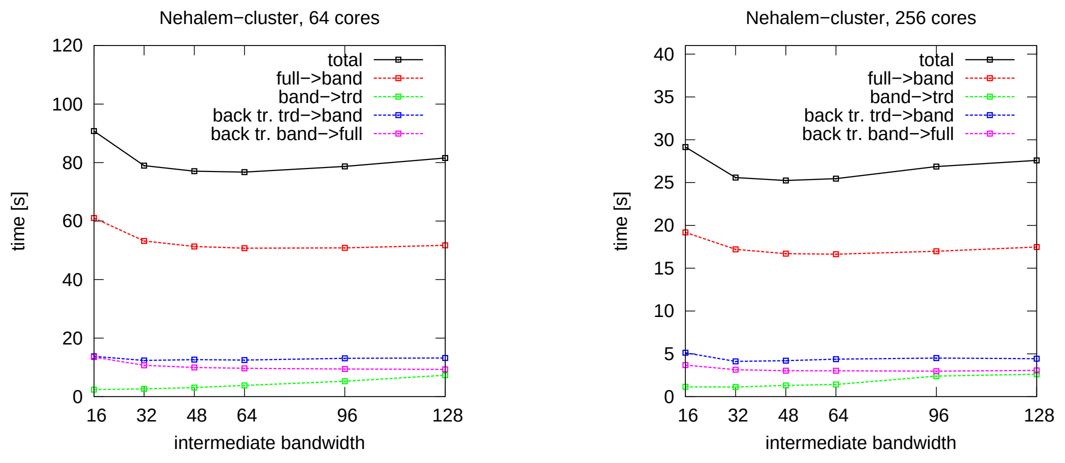


Figure 4.25: Runtime behavior of the 2-step tridiagonalization for Poly27069 on the Nehalem cluster using different intermediate bandwidths.

sections, each stage has its own characteristics. The reduction to banded form as well as both back transformations profit from a higher intermediate bandwidth. However, for bandwidths higher than 48 the performance gain is negligible. On the other side, the reduction from banded to tridiagonal form significantly slows down with higher bandwidths due to the limited parallelizability and the higher flop count. Altogether, we get an optimal runtime for intermediate bandwidths in the range of 32 to 64. On systems with a low single core performance, such as the BlueGene/P, smaller bandwidths are preferable by tendency since the relative effort of the band reduction increases. The ELPA library uses per default an intermediate bandwidth of 32.

In Figure 4.26 we compare, on the one hand, the weak scaling of the 1-step and the 2-step approach within ELPA. On the other hand we analyze the weak scaling behavior of the individual stages to determine the bottlenecks of the algorithm. Therefor we perform weak scaling tests on the SuperMIG with four different problem sizes ranging from a tiny problem size with 1000 elements per process up to problems with 4 million elements per process. We compute the whole eigenspectrum which is the worst case for the 2-step eigensolver. Since both approaches use the same tridiagonal eigensolver, this step is not included in the total runtime. From the plots we can extract a series of insights:

- For all investigated scenarios the 2-step approach is faster than the 1-step solver. The speedup compared to the 1-step approach is higher if (a) more processes are involved and (b) the problem size per process is smaller. This emphasizes the superior weak and strong scaling behavior of the 2-step tridiagonalization.
- The reduction of banded matrices to tridiagonal form shows a nearly perfect weak scaling behavior. Due to the limited strong scaling, this step gets increasingly important if the problem size per process decreases. For larger problems the effort for this stage is negligible.
- Both back transformations show a very good weak and strong scaling behavior. However, these stages require the highest amount of flops if all eigenvectors have to be computed ($2n^3$ flops each) and, thus, begin to dominate the total runtime for large problem sizes per process.
- The reduction to banded form dominates the total runtime for most of the studied cases. However, in prior plots (Figure 4.2) we have seen that, compared to the direct tridiagonalization, there is a significant improvement regarding efficiency (compute bounded instead of memory bounded) and scalability (reduction of synchronization points).

Finally in Figure 4.27 and 4.28 we compare the 2-step tridiagonalization with its competitors ScaLAPACK and Elemental. Figure 4.26 shows strong scaling results for the

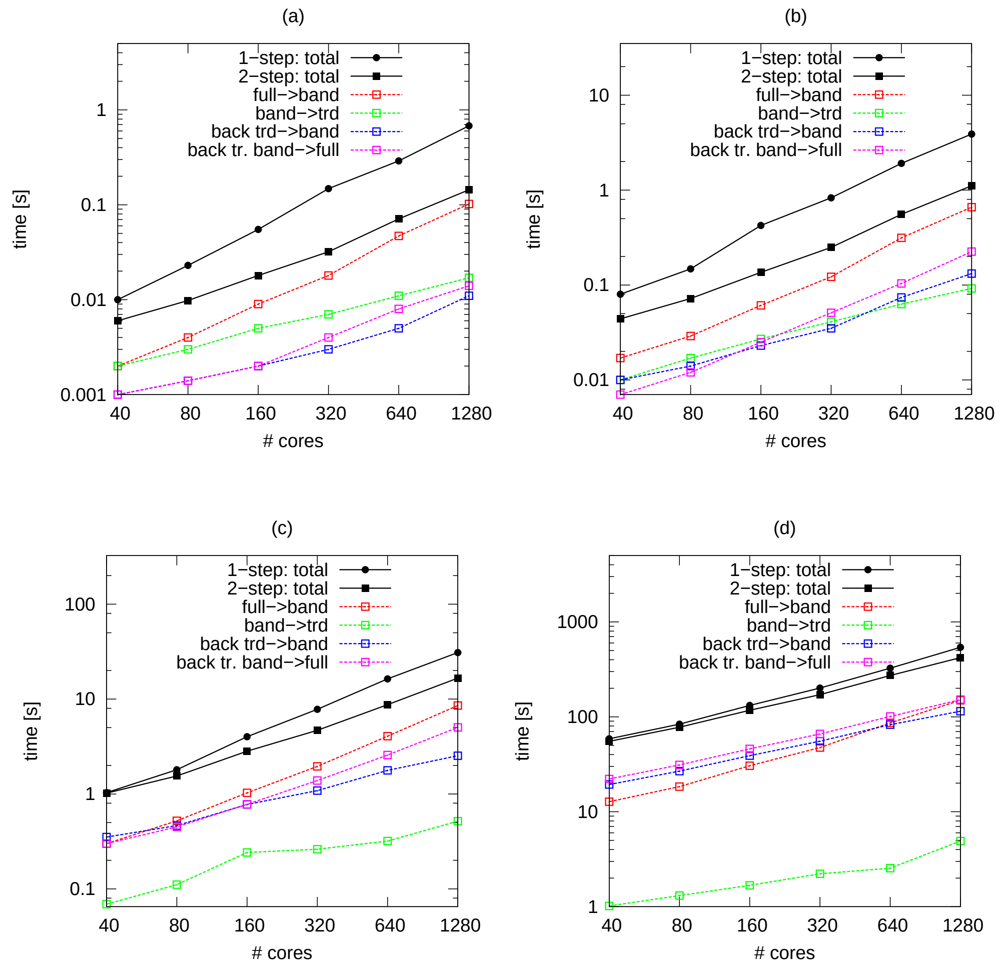


Figure 4.26: Weak scaling results on SuperMIG with four different problem sizes: (a) 1k matrix elements per process, (b) 16k matrix elements per process, (c) 256k matrix elements per process and, (d) 4M matrix elements per process. The intermediate bandwidth was set to 32.

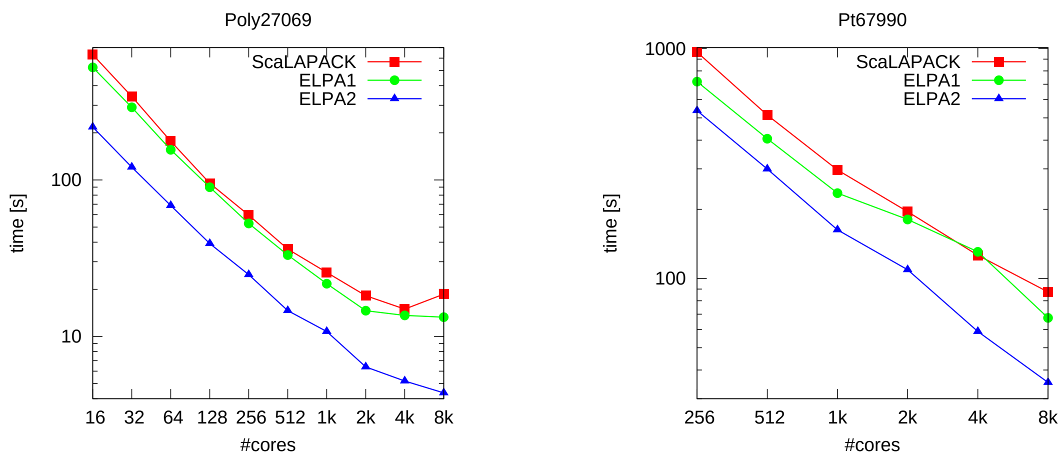
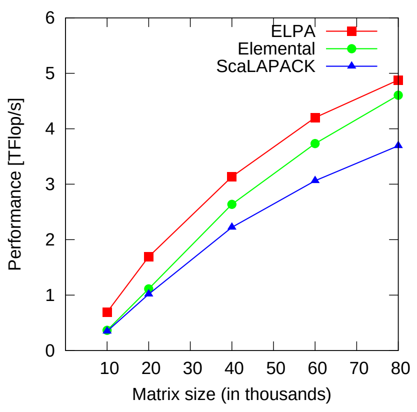


Figure 4.27: Comparison between the strong scaling behavior of ScaLAPACK and the ELPA solvers based on the 1-step and 2-step tridiagonalization on the SuperMUC. For ScaLAPACK the new tridiagonalization routine `pdsyntrd` is used.

well-known problems Poly27069 and Pt67990 on the SuperMUC. The ELPA eigensolver, based on the 2-step tridiagonalization, is compared against the 1-step ELPA solver as well as the ScaLAPACK eigensolver. The runtimes include the time for the tridiagonalization and back transformation. The 2-step tridiagonalization performs best for both problems and all used core counts. We can observe both, a higher efficiency (for small core counts) and a better scalability (for higher core counts). Due to the small fraction of requires eigenvectors, Poly27069 is advantageous for the 2-step solver. ScaLAPACK is used with the new tridiagonalization routine `pdsyntrd` which is significantly faster than `pdsytrd` (see [47]) and comes close to the performance of the 1-step tridiagonalization in ELPA. Compared to ScaLAPACK, the 2-step tridiagonalization leads to speedups in the range of 1.8 to 4.3 over all measurements.

In Figure 4.28 we compare our tridiagonalization and back transformation with the respective implementations in ScaLAPACK and Elemental on the BlueGene/P. For three kinds of reasons the results represent a worst case scenario for our 2-step solver: (i) We compute all eigenvectors. This requires $\frac{16}{3}n^3$ flops for the 2-step approach, whereas the 1-step approach needs only $\frac{10}{3}n^3$ flops. (ii) The measurements for ScaLAPACK and Elemental are out of [86] and represent the best performance using a large set of parameter configurations. ELPA is used with the default settings. (iii) The BlueGene/P is a system which is not advantageous for the 2-step tridiagonalization (poor single-core performance and a relatively small ratio between t_{mem} and t_{flop}). Nevertheless, we outperform the competing libraries ScaLAPACK and Elemental by up to a factor of 2.



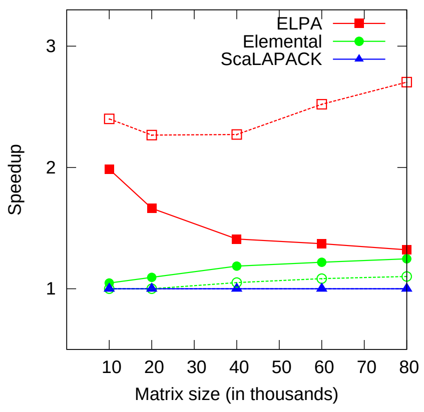


Figure 4.28: Performance comparison of our 2-step solver (tridiagonalization and back transformation) with the respective implementations in ScaLAPACK and Elemental on 8192 cores of the BlueGene/P. The results of ScaLAPACK and Elemental are out of [86]. Left: absolute performance for different matrix sizes (on the basis of $\frac{10}{3}n^3$ flops for each problem). Right: Speedup over ScaLAPACK while computing all eigenvectors (continuous lines) and while computing only eigenvalues (dashed lines).

To sum up, we can say that our implementation of the 2-step tridiagonalization is a promising approach. Our measurements on a wide range of different supercomputing architectures result in unprecedented performance for all examined problems. The 2-step approach is especially advantageous if huge core counts come to use or only a small fraction of the eigenvectors have to be transformed back. Last but not least, our new implementation is well-prepared for current trends in the development of hardware.

5 Conclusion

In this thesis we presented the development of a new library routine for the parallel symmetric eigenproblem. The development was mainly driven by the tremendous need of an efficient eigensolver in the field of quantum chemistry. The requirements for this type of applications are to solve small to medium-sized (in terms of High Performance Computing) eigenproblems. However, tens or or even hundred of thousands of such eigenproblems have to be solved consecutively. Typical dimensions for the corresponding matrices range from a few thousand up to one hundred thousand or more. In terms of efficiency and scalability, theses are the most challenging problems. Existing libraries fail to solve these problems efficiently and in an reasonable amount of time.

The new eigensolver ELPA is based on the 2-step tridiagonalization, a cache-efficient approach to tridiagonalize symmetric matrices which was already developed in the mid 90s. Starting from this work we optimized existing and developed new parallelization schemes for all stages of the eigensolver. The library is a joint work of the ELPA consortium with Bergische Universität Wuppertal, Rechenzentrum Garching and Technische Universität München as main contributors on the development side and two groups within Fritz-Haber-Institut and Max-Planck-Institut as users.

The contribution of the author has been presented in this thesis and comprises important parts of the solver which are crucial for the scalability of the algorithm. Among them are a new parallelization scheme for the tridiagonal-to-banded back transformation with the ability to apply also short Householder transformations in an efficient way, and a new QR-decomposition for the reduction to banded form which asymptotically reduces the synchronization requirements and is, thus, an important step towards weak scaling of the algorithm.

The outcome has been tested on a wide range of different supercomputing architectures, leading to unprecedented performance. Compared to the state-of-the-art library ScaLAPACK, ELPA leads to speedups of up to 10, depending on the problem size and the used system. For typical scenarios, the speedups are in the range of 2 to 4, with the expectation that this ratios will grow on future systems. Applied to the field of quantum chemistry, ELPA allows on the one hand to solve existing problems more efficiently. On the other hand, our new eigensolver makes it possible to simulate larger systems or longer timescales which wouldn't have been possible in a reasonable timeframe with existing eigensolvers.

Nevertheless, the ELPA library is not a finished project. As we already mentioned, our implementation of the 2-step tridiagonalization is well prepared for the observed long-term hardware trends. However, these changes require a steadily adaptation of intern blocking parameters. Currently, ELPA uses default settings which show good performance on a wide range of matrices and architectures. In the long term, an autotuning mechanism would be desirable which automatically choses the best parameter settings for a given problem and supercomputing system. Preliminary work can be found in [92]. Another issue arises from the scaling behavior. What if we want to solve much larger eigenproblems in a comparable amount of time? Although ELPA yields significant improvements regarding weak and strong scaling, the current algorithms won't allow to solve larger eigenproblems in the same time with comparable efficiency. This requires completely new algorithms (if existing) and is currently an active research field in parallel dense linear algebra [93, 94, 95]. So, one thing we know for sure. It will remain challenging.

Bibliography

[1] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Host Simon. Top500 list, November 2011, 2011. `http://www.top500.org`, accessed 01-05-2012.

[2] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[3] J. Howard. A 48-core IA-32 processor with on-die message-passing and DVFS in 45nm CMOS. In *Solid State Circuits Conference (A-SSCC), 2010 IEEE Asian*, pages 1 – 4, 2010.

[4] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98 – 589, 2007.

[5] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation, 2005. `http://www.ibm.com/developerworks/power/library/pa-cellperf/`, accessed 01-30-2012.

[6] Susan L. Graham, Marc Snir, and Cynthia A. Patterson. *Getting Up to Speed: The Future of Supercomputing*. The National Academies Press, 2004.

[7] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[8] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, New York, NY, USA, 2004. ACM.

[9] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[10] OpenMP Architecture Review Board. OpenMP application program interface. Specification, 2008.

[11] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[12] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[13] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[14] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[15] Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, and J. Carlos Mouriño. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 174–184, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Guillermo L. Taboada, Carlos Teijeiro, Juan Tourino, Basilio B. Fraguela, Ramón Doallo, José Carlos Mourino, Damián A. Mallon, and Andrés Gomez. Performance evaluation of Unified Parallel C collective communications. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, HPCC '09, pages 69–78, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Hongzhang Shan, Filip Blagojević, Seung-Jai Min, Paul Hargrove, Haoqiang Jin, Karl Fuerlinger, Alice Koniges, and Nicholas J. Wright. A programming model performance study using the NAS parallel benchmarks. *Sci. Program.*, 18(3-4):153–167, August 2010.

[18] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[19] Barcelona Supercomputing Center. SMP Superscalar (SMPSs) user's manual. Specification, 2011.

[20] Enric Tejedor, Montse Farreras, David Grove, Rosa M. Badia, Gheorghe Almasi, and Jesus Labarta. ClusterSs: a task-based programming model for clusters. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 267–268, New York, NY, USA, 2011. ACM.

[21] Barcelona Supercomputing Center. Cell Superscalar (CellSs) user's manual. Specification, 2007.

[22] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA. Technical report, Innovative Computing Laboratory, University of Tennessee, 2010.

[24] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965.

[25] Volker Blum, Ralf Gehrke, Felix Hanke, Paula Havu, Ville Havu, Xinguo Ren, Karsten Reuter, and Matthias Scheffler. Ab initio molecular simulations with numeric atom-centered orbitals. *Computer Physics Communications*, 180(11):2175 – 2196, 2009.

[26] P. R. C. Kent. Computational challenges of large-scale, long-time, first-principles molecular dynamics. *Journal of Physics: Conference Series*, 125(1):012058, 2008.

[27] Ernesto Estrada. Structural patterns in complex networks through spectral analysis. In *Proceedings of the 2010 joint IAPR international conference on Structural, syntactic, and statistical pattern recognition*, SPR'10, pages 45–59, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256, 2003.

[29] Fan R. K. Chung. *Spectral Graph Theory (CBMS Regional Conference Series in Mathematics, No. 92).* American Mathematical Society, February 1997.

[30] I.J. Farkas, I. Derenyi, A.L. Barabasi, and T. Vicsek. Spectra of "real-world" graphs: Beyond the semicircle law. *Physical Review E*, 64(2):026704, 2001.

[31] Anirban Banerjee and Jürgen Jost. Spectral plot properties: Towards a qualitative classification of networks. In *In European Conference on Complex Systems*, 2007.

[32] Martin Galgon, Lukas Krämer, and Bruno Lang. Schlussbericht zu ELPA, Bergische Universität Wuppertal, 2012.

[33] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[34] Bruno Lang. Direct solvers for symmetric eigenvalue problems in modern methods and algorithms of quantum chemistry. In *J. Grotendorst (Editor), Proceedings, NIC Series Volume*, pages 231–259, 2000.

[35] B. Lang. *Effiziente Orthogonaltransformationen bei der Eigen- und Singulärwertberechnung.* Wuppertal, 1997.

[36] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[37] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[38] 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.

[39] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, and Yuan jye J. Wu. PLAPACK: Parallel Linear Algebra Package, 1997.

[40] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[41] M. Petschow and P. Bientinesi. MR3-SMP: A symmetric tridiagonal eigensolver for multi-core architectures. *Parallel Computing*, 37(12):795 – 805, 2011. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[42] J. H. Wilkinson. The calculation of the eigenvectors of codiagonal matrices. *The Computer Journal*, 1(2):90–96, 1958.

[43] G.H. Golub and C.F.V. Loan. *Matrix computations.* Johns Hopkins series in the mathematical sciences. Johns Hopkins University Press, 1989.

[44] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[45] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* PhD thesis, Univ. of California at Berkeley, 1997.

[46] I. S. Dhillon and Beresford N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and Appl*, 387:1–28, 2004.

[47] T. Auckenthaler, V. Blum, H. J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Comput.*, 37:783–794, December 2011.

[48] James W. Demmel, Osni A. Marques, Beresford N. Parlett, and Christof Vömel. Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers. *SIAM J. Sci. Comput.*, 30(3):1508–1526, March 2008.

[49] P. R. Willems. *On MRRR-type Algorithms for the Tridiagonal Symmetric Eigenproblem and the Bidiagonal SVD.* PhD thesis, Bergische Univ. Wuppertal, 2010.

[50] Matthias Petschow, Enrique S. Quintana-Orti, and Paolo Bientinesi. Improved orthogonality for dense Hermitian eigensolvers based on the MRRR algorithm. In *AICES technical report*, 2012.

[51] Paolo Bientinesi, Inderjit S. Dhillon, and Robert A. van de Geijn. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM J. Sci. Comput.*, 27(1):43–66, July 2005.

[52] Wallace Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):pp. 26–50, 1958.

[53] W. Morven Gentleman. Least squares computations by Givens transformations without square roots. *IMA Journal of Applied Mathematics*, 12(3):329–336, 1973.

[54] Sven Hammarling. A note on modifications to the Givens plane rotation. *IMA Journal of Applied Mathematics*, 13(2):215–218, 1974.

[55] Wolfgang Rath. Fast Givens rotations for orthogonal similarity transformations. *Numerische Mathematik*, 40:47–56, 1982. 10.1007/BF01459074.

[56] Alston S. Householder and Friedrich L. Bauer. On certain methods for expanding the characteristic polynomial. *Numerische Mathematik*, 1:29–37, 1959.

[57] J. H. Wilkinson. Householder's method for the solution of the algebraic eigenproblem. *The Computer Journal*, 3:23–27, 1960.

[58] Christian Bischof and Charles van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8:2–13, January 1987.

[59] Robert Schreiber and Charles van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10:53–57, January 1989.

[60] Chiara Puglisi. Modification of the Householder method based on the compact WY representation. *SIAM J. Sci. Stat. Comput.*, 13:723–726, May 1992.

[61] Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. Accumulating Householder transformations, revisited. *ACM Trans. Math. Softw.*, 32:169–179, June 2006.

[62] Sven J. Hammarling, Danny C. Sorensen, and Jack J. Dongarra. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math*, 27:215–227, 1987.

[63] Christian Bischof, Bruno Lang, and Xiaobai Sun. Parallel tridiagonalization through two-step band reduction. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 23–27. IEEE Computer Society Press, 1994.

[64] H. Schwarz. Algorithm 183: Reduction of a symmetric bandmatrix to triple diagonal form. *Communications of the ACM*, 6:315–316, 1963.

[65] H. Schwarz. Tridiagonalization of a symetric band matrix. *Numerische Mathematik*, 12:231–241, 1968.

[66] Sivasankaran Rajamanickam and Timothy A. Davis. Blocked band reduction for symmetric and unsymmetric matrices, 2010.

[67] Murata Kenro and Horikoshi Kiyomi. A new method for the tridiagonalization of the symmetric band matrix. *Information processing in Japan*, 15:108–112, 1975.

[68] Bruno Lang. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM J. Sci. Comput.*, 14:1320–1338, November 1993.

[69] Linda Kaufman. Banded eigenvalue solvers on vector machines. *ACM Trans. Math. Softw.*, 10:73–85, January 1984.

[70] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. The SBR toolbox - software for successive band reduction, 1996.

[71] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. A framework for symmetric band reduction, 1999.

[72] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Geijn. Collective communication: Theory, practice, and experience, FLAME Working Note #22, 2006.

[73] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[74] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines, 1996.

[75] Thomas Auckenthaler, Thomas Huckle, and Roland Wittmann. A blocked QR-decomposition for the parallel symmetric eigenvalue problem, 2012. in preparation.

[76] W. Gander. *Algorithms for the QR decomposition*. Seminar für Angewandte Mathematik: Research report. 1980.

[77] James Demmel, Laura Grigori, Mark Frederick Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations, 2008.

[78] Andreas Stathopoulos and Kesheng Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput*, 23, 2002.

[79] Roland Wittmann. Blocking strategies for the parallel QR-decomposition. Research project, Institut für Informatik, Technische Universität München, 2011.

[80] Roland Wittmann. Analysis, implementation and evaluation of parallel algorithms for the QR-decomposition. Masters thesis, Institut für Informatik, Technische Universität München, 2012.

[81] T. Auckenthaler, H. J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *J. Comput. Science*, 2(3):272–278, 2011.

[82] Kendall Swenson Stanley. Execution time of symmetric eigensolvers. Technical Report UCB/CSD-99-1039, EECS Department, University of California, Berkeley, 1997.

[83] R.C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39:39–5, 1995.

[84] Dror Irony and Sivan Toledo. Communication-efficient parallel dense LU using a 3-dimensional approach. In *in: Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[85] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 8:1–8:11, New York, NY, USA, 2011. ACM.

[86] 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*, 2012. To appear.

[87] T. Imamura, S. Yamada, and M. Machida. Development of a high performance eigensolver on the petascale next generation supercomputer system. In *Proceedings of Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2010*, 2010.

[88] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, and Tadashi Watanabe. *The K computer*: Japanese next-generation supercomputer development project. In *ISLPED*, pages 371–372, 2011.

[89] ELPA documentation. `http://elpa-lib.fhi-berlin.mpg.de`, accessed 09-14-2012.

[90] R. Johanni, V. Blum, V. Havu, H. Lederer, and M. Scheffler, 2011. in preparation.

[91] Paula Havu, Volker Blum, Ville Havu, Patrick Rinke, and Matthias Scheffler. Large-scale surface reconstruction energetics of Pt(100) and Au(100) by all-electron density functional theory. *Phys. Rev. B*, 82:161418, Oct 2010.

[92] Stefan Schulze Frielinghaus. Parameter optimization for the parallel symmetric eigenvalue problem. Studienarbeit/IDP, Institut für Informatik, Technische Universität München, December 2010.

[93] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. Technical Report UCB/EECS-2011-72, EECS Department, University of California, Berkeley, Jun 2011.

[94] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Sequential communication bounds for fast linear algebra. Technical Report UCB/EECS-2012-36, EECS Department, University of California, Berkeley, Mar 2012.

[95] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. Technical Report UCB/EECS-2012-31, EECS Department, University of California, Berkeley, Mar 2012.