Towards an Exascale Enabled Sparse Solver Repository

Jonas Thies, Martin Galgon, Faisal Shahzad, Andreas Alvermann, Moritz Kreutzer, Andreas Pieper, Melven Röhrig-Zöllner, Achim Basermann, Holger Fehske, Georg Hager, Bruno Lang, and Gerhard Wellein

Abstract As we approach the Exascale computing era, disruptive changes in the software landscape are required to tackle the challenges posed by manycore CPUs and accelerators. We discuss the development of a new 'Exascale enabled' sparse solver repository (the ESSR) that addresses these challenges—from fundamental design considerations and development processes to actual implementations of some prototypical iterative schemes for computing eigenvalues of sparse matrices. Key features of the ESSR include holistic performance engineering, tight integration between software layers and mechanisms to mitigate hardware failures.

1 Introduction

It is widely accepted that the step from Peta- to Exascale is qualitatively different from previous advances in high performance computing and therefore poses urgent questions. Considering applications that need these vast computing resources, which algorithms expose such massive parallelism? What will the next generations of supercomputers look like, and how can we write sustainable yet efficient software for them? The ESSEX project [1] has developed the 'Exascale enabled Sparse Solver Repository' (ESSR) over the past three years,

Jonas Thies, Melven Röhrig-Zöllner and Achim Basermann
German Aerospace Center (DLR), Simulation and Software Technology, Cologne, Germany.

Martin Galgon and Bruno Lang
University of Wuppertal, School of Mathematics and Natural Sciences, Wuppertal, Germany.

Andreas Alvermann, Andreas Pieper and Holger Fehske
University of Greifswald, Institute of Physics, Greifswald, Germany.

Moritz Kreutzer, Faisal Shahzad, Georg Hager and Gerhard Wellein
Erlangen Regional Computing Center, Erlangen, Germany

[1] Equipping Sparse Solvers for the Exascale, `http://blogs.fau.de/essex`, funded by the priority program "Software for Exascale Computing" (SPPEXA) of the German Research Foundation (DFG)

and in this paper we want to share our experiences and summarize our results in order to contribute to answering these questions.

The applications we study come from quantum physics and material science, and are directly or indirectly related to solving the Schrödinger equation. The Hamiltonian of the systems studied can be represented as a (very) large and sparse matrix, and the numerical task is to solve sparse eigenvalue problems in various flavors. The software we develop is intended as a blueprint for other applications of sparse linear algebra.

In the next few years, we expect no radical change in the architecture of supercomputers, so that a scaled up version of current Petascale systems is used as target architecture for the ESSR. That is, a distributed memory cluster of (possibly heterogeneous) nodes. On the other hand, node-level programming will become much more challenging because of the strong increase in node level parallelism and complexity[2]. Due to the increasing node count, we do anticipate a much shorter mean time to failure (MTTF) on the full system scale, which has to be addressed for large simulations using substantial parts of an Exascale system.

A key challenge in the efficient implementation of sparse matrix algorithms is the 'bandwidth bottleneck', the fact that in most modern architectures, the amount of data that can be loaded per floating point operation is continually decreasing. To hide this gap, cache systems of increasing complexity and non-uniform cache/memory hierarchies are used. Another issue is the relative increase of the latency of global reduction/synchronization operations, which are central to many numerical schemes. In the ESSR we address these problems using block algorithms with tailored kernels (see also [1]) and communication hiding.

Three overarching principles guide the design of the ESSR: *disruptive changes of data structures* for node-level efficiency, *holistic performance engineering* to avoid losses on various hardware or software levels to accumulate, and *user-level fault tolerance* schemes to keep the overhead for guaranteeing stable runs as low as possible.

The various layers of the ESSR (application, algorithms and building blocks) were co-developed 'from scratch' within the past three years. This rapid process was only possible with a comprehensive software engineering approach, which we will describe in this paper. We use the term 'repository' rather than 'library' because of the young age of our effort. In the future, the ESSR components will be integrated to form a complete software stack for extreme scale sparse eigenvalue computations and applications.

Related work. A large number of decisions has to be made when designing basic linear algebra data structures such as classes for sparse matrices, (block) vectors or dense matrices. On the other hand, iterative algorithms may remain largely oblivious of these implementation details (e.g. the storage scheme for sparse matrices, the parallelization techniques used). In the past, iterative solver libraries were therefore often based on reverse communication interfaces (RCI, see, e.g., (P)ARPACK [2] or FEAST [3]), or simple callback functions that allowed the user only to provide the result of a matrix-vector product and possibly a preconditioning operation (as in PRIMME [4]). In such approaches, the user is bound to the parallelization technique prescribed by the solver library (i.e. pure MPI in the examples above), and the solver library can not exploit techniques like kernel fusion or overlapping of communication and computation across operations. Another library implementing sparse

[2] see, e.g., `https://www.olcf.ornl.gov/summit/`

eigenvalue solvers is SLEPc [5]. Here the user has to adapt to the data structures of the larger software framework PETSc [6].

A more flexible approach is the concept of an interface layer in the Trilinos library Anasazi [7]. Solvers in this C++ library are templated on scalar data type and the ‘multi-vector’ and operator types. For each kernel library providing these objects, an ‘adapter’ has to be written. Apart from the operator application (which may wrap a sparse matrix-vector product), the kernel library implements a multi-vector class with certain functionality. For an overview of Trilinos, see [8, 9]. Our own approach is to use an interface layer which is slightly more extensive than the one in Anasazi, but puts less constraints on the underlying data structures (see Sect. 3.4).

The predicted range of MTTF for Exascale machines (between hours and minutes [10]) necessitates the inclusion of fault tolerance capabilities in our applications, as they fall in the category of long running large jobs. The program can face various failures during its run, e.g. hardware faults, soft errors, Byzantine failures, software bugs, etc. [11]. According to [12], a large fraction of failures can be attributed to CPU and memory related issues which eventually lead to complete process failures. Such failures define the fault tolerance scope in this work.

Document structure. We start out by describing the basic software architecture of the ESSR in Sect. 2, and a process that allows the concurrent development of sparse solvers and the building blocks they need to achieve optimal performance. Section 3 gives an overview of the software components available in the ESSR. In Sect. 4, three classes of algorithms studied in the ESSEX project are briefly discussed. The objective here is neither to present new algorithmic features or performance results, nor to study any particular application. Instead, we want to summarize the optimization techniques and implementation details we identified while developing these solvers. The fault tolerance capabilities explored in our applications are described in Sect. 5. Section 6 summarizes the paper and gives an outlook on future developments surrounding the ESSR.

2 ESSR Architecture and Development Process

It is a substantial effort to implement a scalable sparse solver library ‘from scratch’. In this section we describe the architecture and development cycle of a set of tightly integrated software layers, that together form the ‘Exascale enabled Sparse Solver Repository’, ESSR. The actual implementation in terms of software packages is detailed further in Sect. 3.

2.1 Software Architecture

The ESSR consists of three main parts, depicted in Fig. 1: an application layer, the computational core and a vertical integration pillar. A fourth part is an extensive test suite, not shown here.

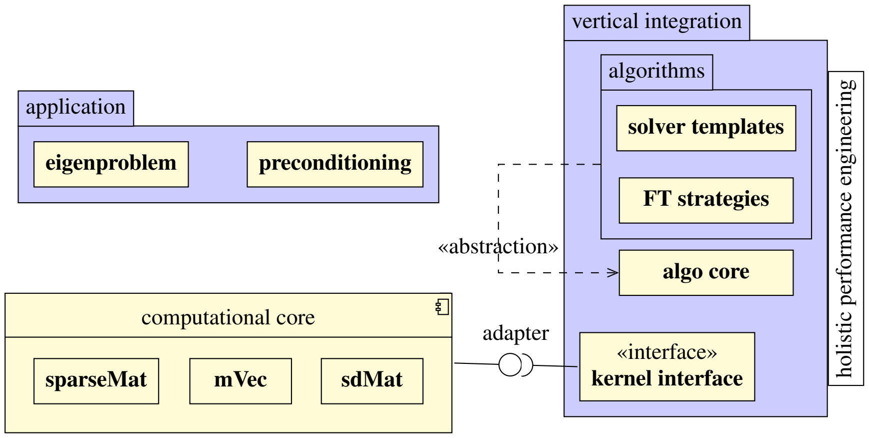

Fig. 1 ESSR software architecture

The computational core (or kernel library) has the task of providing highly optimized implementations of the kernels required by the algorithms and applications we study. It hides implementation details such as SIMD instructions, NUMA aware memory management and MPI communication from the other layers. It is a 'component' in the sense that it could be replaced by another implementation if the software is ported to radically different hardware, or if new applications with different requirements come up. The basic data structures it provides are classes for sparse matrices (`sparseMats`), tall and very skinny matrices (or 'multi-vectors', `mVecs`) and small and dense matrices (`sdMats`).

The vertical integration pillar is based on a clear interface to the computational core, subsequently referred to as 'kernel interface'. It defines the basic data structures and operations that the computational core has to provide. The 'algo core' layer implements common functionality useful for various high level algorithms. Examples include block orthogonalization, evaluating matrix polynomials and extracting Ritz values from a subspace. On top of the kernel interface and core functionality, iterative algorithms are implemented. Fault tolerance strategies are built into algorithms, and common concepts here are again implemented in the algorithmic core layer. The vertical integration pillar is designed to enable holistic performance engineering, as will be discussed below.

The application layer defines an eigenvalue problem and uses an algorithm to solve it. To set up the problem and pre-/postprocess the results, it may either use the simplified kernel interface or the full functionality of the computational core library. The second component of the application layer is an optional preconditioner, i.e. any deterministic linear operator used to accelerate the solution of linear systems arising in an eigenvalue computation. We placed this important building block into the application layer because the construction of a suitable preconditioner may be problem dependent, especially in quantum physics, where 'black box' AMG or ILU methods are generally not applicable. The implementation of a preconditioning technique will typically work directly on the data structures of the computational core. While the vertical pillar is connected to the computational core only via a clear interface, the degree to which an application can use another kernel library depends on its implementation and need for specific preconditioners and pre-/postprocessing. Simple applications that only need matrix construction (or I/O) and standard operations can stay independent of the underlying implementation by using the kernel interface as the lowest level.

Tightly connected to the vertical pillar is an extensive test framework (cf. Sect. 3.6), with a continuous integration process to ensure software quality. The largest number of tests targets the computational core, through the kernel interface. The algorithmic core is tested using synthetic cases (integration tests), and system tests (numerical test cases for the algorithm layer) are provided by matrix collections/generators and the application layer.

2.2 Concurrent Development of all Layers

The introduction of the kernel interface enables the use of established libraries while developing/implementing iterative methods. The core layers can thus be developed in parallel to the algorithms layer. The kernels required are defined dynamically during the development process and implemented in a test-driven process in the computational core, see Fig. 2. In a similar workflow, common functionality used in several solvers is identified and abstracted into the ‘algo core’ layer, where a numerically robust and fully optimized implementation is brought forth while algorithm development continues at a higher level. An example is the development of a communication optimal and robust block orthogonalization scheme while implementing block Jacobi-Davidson (Sect. 4.3) based on a simple yet robust (iterated) modified Gram-Schmidt process.

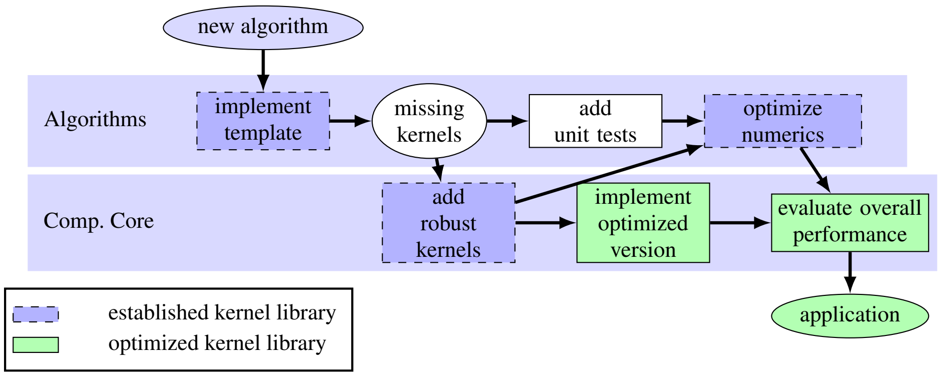

Fig. 2 Test-driven co-development of optimized algorithms in the ESSR

2.3 Integration of Performance Engineering

While developing an iterative solver, all performance critical operations are identified and added to the kernel interface. As the number of relevant kernels is moderate, a combination of performance models and dedicated benchmarks can be used to ensure their near optimal performance. Many of these operations (such as the sparse matrix-vector multiplication, spMVM, or operations on `mVecs`), are bounded by the main memory bandwidth, such that the roofline model [13] gives a good indication of the quality of the implementation. To understand the performance of a complete algorithm, code instrumentation for performance analysis tools is used. This may reveal, e.g., overhead of thread synchronization or effects of non-uniform memory access (NUMA) which may not occur in isolated benchmarks. More details on how this concept is implemented can be found in Sect. 3.6.

Our primary focus here is node-level performance. The changes in CPU architecture are currently more dramatic than those concerning node interconnection, and any losses at the node level scale with the number of nodes in a supercomputer.

2.4 Fault Tolerance Strategy

The strategy followed in the ESSR to achieve fault tolerance w.r.t. hardware failures can be classified as an application-level checkpoint/restart (C/R) method. In this approach, algorithm specific knowledge is exploited to store the minimum amount of data needed for restarting the computation. A highly optimized implementation of this approach (using e.g. asynchronous checkpointing and neighbor-level checkpoints) promises a low overhead for our long running iterative schemes on many nodes.

Due to the early development stage of fault tolerant communication libraries [14], our strategy is to evaluate various technical solutions in simple use cases before condensing them into a common feature of the ESSR solvers and applications in the 'algo core' layer. Section 5 gives an overview of our work in this area.

3 ESSR Software Landscape

The conceptual design discussed in the previous section is implemented in a collection of compatible software packages, which are publicly accessible under a BSD open source license[3]. Before discussing the software structure further, we will comment on the target computer architecture for the software.

[3] see `http://bitbucket.org/essex`

3.1 Hardware and Execution Models Supported

Exascale computers are not available to date, and a competitive ‘race of flops’ is going on to develop this new generation of supercomputers. Based on the developments in the TOP500 list [15] over the past few years, we decided to develop software targeting machines that consist of many nodes with distributed memory. A node features several multi- or manycore CPUs with non uniform access to caches and main memory, and ‘accelerator’ hardware, e.g. multiple GPUs. At the lowest level, data parallelism is exploited by the hardware through SIMD/SIMT like techniques, compelling choices in data structures and low level implementation. Typical sparse matrix algorithms will continue to be memory-bound on these devices

In this environment we employ the following execution model. Numerical algorithms are implemented as a sequence of function calls, executed transparently on a parallel heterogeneous machine (SPMD model). A distributed memory communication protocol (e.g. MPI) is used between processes running on complete nodes or parts of nodes of the cluster. Within a function we allow arbitrary multithreading techniques for flexible node utilization. The execution of functions may be interleaved using ‘tasks’ which use only a part of the resources available to the process. Data transfers between host CPU and accelerator devices must be handled explicitly by the computational algorithm between function calls where necessary (the underlying kernels do not ‘know’ if the CPU or device memory is up to date).

3.2 ESSR Toolkits and Functionality

The ESSR is implemented in a number of co-developed software packages, also called *toolkits*. These toolkits do not necessarily implement one part of the architecture (Fig. 1) each. Rather, each partner in the ESSEX project has the responsibility for one of the toolkits, whereas the responsibility for the conceptual ESSR parts may be shared among several project partners. In the future, the repository will evolve into a set of libraries providing state-of-the-art, highly scalable and fault tolerant eigensolvers. This may lead to a redistribution of functionality according to the architecture depicted in Fig. 1.

The four toolkits are briefly characterized as follows:

- ESSEX-Physics, a quantum physics toolkit defining applications that we want to solve using the ESSR. It provides scalable sparse matrices from real-world applications and polynomial eigensolvers (see Sects. 3.3 and 4.1).
- GHOST (General, Hybrid and Optimized Sparse Toolkit) implements basic building blocks with a focus on optimal performance on heterogeneous supercomputers. This design goal is achieved by consequent application of performance engineering techniques. GHOST implements the ‘computational core’ of the ESSR in single or double precision, and in real or complex arithmetic [1, 16].
- PHIST (Pipelined Hybrid-parallel Iterative Solver Toolkit) implements the vertical integration pillar of Fig. 1, and adapters for several kernel libraries. It also hosts the test framework, and contributes Jacobi-Davidson type eigensolvers and Krylov methods for

linear systems to the algorithms layer. To provide a more diverse spectrum of methods, we also included adapters for GHOST to the Trilinos libraries Anasazi and Belos.

- BEAST (Beyond fEAST) extends the algorithms layer of the ESSR by innovative projection based eigensolvers which take up the idea of the contour integration based FEAST method [3] (see Sect. 4.2).

We will now describe some of the features of the ESSR, with references to the toolkit where they can be found. The eigensolvers are described in more detail in Sect. 4.

3.3 Applications

Following the overall philosophy of the SPPEXA priority program[4], our development of the ESSR components is closely guided by—but not restricted to—the intended application range in quantum physics and chemistry. Three different types of eigenvalue problems arise for the large sparse symmetric (or Hermitian) matrices derived from the Schrödinger equation. The study of equilibrium properties, e.g., of the electronic states in a certain material, requires computation of either a few extremal eigenvalues (of the order 10–100) or many interior eigenvalues (100–1000) with the Jacobi-Davidson algorithm or BEAST, respectively. On the other hand, effectively all the eigenvalues contribute to the dynamic properties of highly excited or driven systems out of equilibrium, and expansion techniques such as the kernel polynomial method (KPM) and Chebyshev time propagation (ChebTP) come into play. These algorithms and their implementation are briefly discussed in Sect. 4. Thus, our target applications require solution of the entire range of large sparse symmetric eigenvalue problems.

Similarly, a variety of matrices occur in the applications: While stencil- and band-like matrices are characteristic for graphene and topological insulators, the tensor structure of quantum mechanical Hilbert space leads to intricate sparsity patterns with long thin subdiagonals or scattered small subblocks for correlated many-particle quantum systems. Also, spectral properties of the matrices differ widely, which allows for algorithmic developments and thorough testing without losing contact to the real application. For example, the appearance of a pseudo-gap in the density of states for topological insulators can be exploited for interior eigenvalue computations with polynomial filter functions [17]. Scalable matrix generation routines are included in the ESSEX-Physics library for correlated many-particle systems and new topological materials, all of which are research problems of current interest.

3.4 Kernel Interface

The algorithms summarized in Sect. 4 can be implemented with the three basic data structures introduced in Sect. 2, `sparseMats`, `mVecs` and `sdMats`. To maintain flexibility, we

[4] see `http://www.sppexa.de/`

added a fourth, an abstract linear operator type (`linearOp`), which may be used to provide, e.g., preconditioning techniques or implement matrix-free methods. Inspired by the Petra object model employed by Trilinos [8], we also abstracted data distribution into a `map` object and inter-process communication into a `comm` object. Another Petra concept that is useful when implementing iterative solvers is a 'view' of (part of) an `mVec` or `sdMat`. A view is a light-weight object that only has meta data and provides (read and/or write) access to the elements of the 'viewed' object without copying them. Thus it is, e.g., possible to apply an operator or sparse matrix to selected columns of an `mVec`.

As mentioned in Sect. 1, the Anasazi interface layer resolves the problems of earlier techniques by allowing the sparse matrix and block vector implementations to be co-designed with matching parallelization techniques and data layouts. We adapted this idea to our needs, in PHIST, with the following main differences:

C interface. Having to provide a C++ adapter may be a hassle for e.g. Fortran programmers. We restrict ourselves to four scalar data types (ST), single or double, real or complex, which can be implemented optionally. For each ST, a set of plain C functions has to be provided, which accept objects as `void` pointers. Errors and flags are passed via the last (`int*`) argument, similar to the MPI interface. This minimalistic interface allows maximum flexibility for users of PHIST and providers of kernel libraries alike. The lack of type safety introduced by passing around objects as `void*` is alleviated by the test framework discussed in Sect. 3.6.

`sdMat`. We require the kernel library to provide this object to increase flexibility. For instance, an `sdMat` may be replicated on host CPU and GPU, or it may be stored in higher precision to increase the numerical stability of reduction operations.

View concepts. Allowing custom `sdMats`, we also require views of contiguous rows and columns in an `sdMat`. On the other hand, we only require views of contiguous and increasing columns of an `mVec`. This makes it easier to implement `mVecs` in row-major ordering for better performance [18]. Strided memory access leads to a significant performance penalty in that case, and restricting the interface therefore gives more uniform performance of the view objects supported.

Explicit data transfers for accelerators. For compute platforms that have both a host processor and one or more accelerators, we support the data parallel execution model implemented in GHOST [16]. At least one MPI process is used for each component of a heterogeneous node, and a 'GPU process' has a management thread running on the host CPU. Special kernel interface functions exist to transfer the data of `sdMats` between host and device.

3.5 Computational Core

The mathematical simplicity of the objects and functions required by the kernel interface is misleading. Let us consider the operation $C = V^T W, C \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{n\times k}$. If this operation is implemented using OpenMP inside each MPI process and Intel(R) AVX SIMD instructions, the data in the objects must be contiguous, correctly aligned and padded, which may not be the case if V, W and/or C are views of some parts of larger objects. The reduction operation must produce consistent results on all MPI processes, and if accelerators like GPUs are involved, data transfers must be managed explicitly. The constraints on data layout also hold for efficient GPU processing. All of these complexities are hidden in the ESSR library GHOST [16]. Automatically generated kernels are selected dynamically depending on data alignment, block size and CPU type. Shared memory parallelism on CPUs and the Intel(R) Xeon Phi is implemented using OpenMP, and Nvidia GPUs are supported by providing optimized CUDA kernels.

Another important component of GHOST is a lightweight, general purpose tasking mechanism that plays well within the standard data parallel execution model of 'MPI+X'. It is used in the ESSR for overlapping communication with computation, asynchronous checkpointing etc. The PHIST library provides macros to simplify the use of this tool when implementing an algorithm.

Apart from GHOST, PHIST currently has adapters for the Trilinos libraries Epetra and Tpetra. Builtin Fortran/C99 kernels make PHIST self-contained in principle and are used for performance engineered prototypes of functionality not yet available in GHOST.

3.6 Verifying Software Correctness and Performance

Correctness tests. The number of possible execution paths in GHOST is huge, because it uses automatically generated high-end kernels for fixed block sizes, allows mixing of row- and column-major dense matrices and real and complex arithmetic, etc. In order to keep the effort of testing the building blocks in ESSEX at a reasonable level, we therefore restrict ourselves to testing via the kernel interface.

The test framework in PHIST is based on Google Test[5], with modifications to ensure correct behavior in a hybrid parallel setting with MPI+X. These modifications include broadcasting test errors to all MPI processes and assertions to verify that certain data is identical on all processes. The main point here is to decide what type of errors the tests should be able to detect, and under which conditions they should work correctly. For example, some communication errors with MPI cannot be detected by the test framework as it relies on MPI itself. Here one may run the tests in simplified settings (single/multiple thread(s), single/multiple MPI rank(s), GPU only etc.) to test each layer of parallelism separately. Various tools can support this kind of testing, e.g., the thread and address sanitizer included in recent versions of GCC[6], the MPI checker MUST [7] or CUDA-MEMCHECK[8].

Tests are automatically generated from single source files for different block sizes, vector lengths, and data types, and for views and standard objects where appropriate. They are executed in nightly builds for different configurations, which leads to a total of currently about 80,000 tests for each kernel library, compiler and MPI version tested. We use the

[5] `https://github.com/google/googletest`
[6] `https://github.org/google/sanitizers`
[7] `https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST`
[8] `http://docs.nvidia.com/cuda/cuda-memcheck/`

continuous integration tool Jenkins[9] to obtain an overview of the results. Comparison with the comparatively stable Epetra and Tpetra implementations increases the confidence in the correctness of the tests themselves.

Performance testing. Our adapters for the kernel interface and the functions of the core and algorithmic layers are instrumented to provide timing information and/or markers for the Likwid performance monitoring tool [19]. Another option that can be turned on at compile time is to include a simple performance model for memory bounded kernels. In this case, a small benchmark of the memory bandwidth is run and the percentage of the roofline [13] performance achieved by each kernel function is printed at the end of a run.

There are two 'modes' of performance testing: one incorporates the actual data layout in memory and thus helps to verify that the underlying kernel library achieves the predicted performance for each operation, whether it involves views or not. The other mode only considers the amount of data. This reveals possible performance flaws in the design or implementation of algorithms. For example, if the main operations are performed with a single column view of a row major multi-vector of block size 2, less than 50% of the roofline performance may be achieved on cache-based architectures.

4 Algorithms Implemented in the ESSR

In this section we want to give a broad overview of the algorithms studied in the ESSEX project, and summarize the lessons learned while developing their highly optimized implementations in the ESSR. For more details, numerical experiments and performance results on massively parallel systems, we refer to the publications cited below.

4.1 Algorithms Based on Chebyshev Polynomials

Algorithms based on the evaluation of polynomial matrix functions are a basic ESSR component. They are represented by the kernel polynomial method (KPM) [20] for spectral functions and eigenvalue densities, Chebyshev time propagation (ChebTP) [21,22] for matrix exponentials $\exp[tA]$, and Chebyshev filter diagonalization (ChebFD) [17] for the computation of interior eigenvalues. The latter is available through the BEAST-P variant, see Sect. 4.2.

In contrast to, e.g., sparse factorizations or preconditioning that require explicit access to the matrix elements, polynomial algorithms address the matrix in question only through spMVM. Therefore, they are well suited for situations where the former techniques do not work, or where the matrix is not stored explicitly but only constructed 'on-the-fly' in the spMVM routine. While from the mathematical point of view polynomial algorithms are inferior to algorithms based on rational matrix functions, they are often the only alternative for extremely large matrices.

[9] `https://jenkins-ci.org`

The common idea behind KPM, ChebTP, and ChebFD is the expansion of a function $f(z) = \sum_{n=0}^{\infty} c_n p_n(z)$ into a series of polynomials $p_n(z)$, especially the Chebyshev polynomials $T_n(z)$ which are often the most favorable choice for numerical algorithms. The algorithms come in two variants: KPM computes the expansion coefficients c_n from scalar products $\langle y, p_n[A]x \rangle$ in order to (re-)construct the function $f(z)$, e.g., the eigenvalue density, while ChebTP and ChebFD use given coefficients c_n to accumulate a result vector $y = \sum_n c_n p_n[A]x$, either for the matrix exponential $y = \exp[tA]x$ (ChebTP) or a subspace projection $y = Px$ (ChebFD). An important idea from approximation theory that features both in KPM and ChebFD is the use of so-called kernel polynomials to improve convergence of the expansion [17,20,23].

Algorithm 1 Polynomial matrix function evaluation

```
 1 for $k = 1$ to $M$ do                                              ▷ First two recurrence steps
 2     $\mathbf{u}_k = \alpha_1(A + \beta_1 \mathbb{1})\mathbf{x}_k$                                   ▷ spmv()
 3     $\mathbf{w}_k = \alpha_2(A + \beta_2 \mathbb{1})\mathbf{u}_k + \gamma_2 \mathbf{x}_k$                          ▷ spmv()
 4     $\mathbf{x}_k = c_0\mathbf{x}_k + c_1\mathbf{u}_k + c_2\mathbf{w}_k$                       ▷ axpy & scal (ChebTP, ChebFD)
 5     $c_0^{(k)} = \langle \mathbf{y}, \mathbf{x}_k \rangle$, $c_1^{(k)} = \langle \mathbf{y}, \mathbf{u}_k \rangle$, $c_1^{(k)} = \langle \mathbf{y}, \mathbf{w}_k \rangle$        ▷ dot or gemm (KPM)
 6 for $n = 3$ to $N$ do                                         ▷ Remaining recurrence steps
 7     for $k = 1$ to $M$ do
 8         swap($\mathbf{w}_k, \mathbf{u}_k$)                                           ▷ swap pointers
 9         $\mathbf{w}_k = \alpha_n(A + \beta_n \mathbb{1})\mathbf{u}_k + \gamma_n \mathbf{w}_k$                      ▷ spmv()
10         $\mathbf{x}_k = \mathbf{x}_k + c_n\mathbf{w}_k$                                  ▷ axpy (ChebTP, ChebFD)
11         $c_n^{(k)} = \langle \mathbf{y}, \mathbf{w}_k \rangle$                                   ▷ dot or gemm (KPM)
```

To achieve high execution speed with minimal memory requirements, the polynomials $p_n(z)$ are computed from a two term recurrence

$$x_{n+1} = \alpha_n(A + \beta_n \mathbb{1})x_n + \gamma_n x_{n-1} \tag{1}$$

for the vectors $x_n = p_n[A]x$, which gives the algorithmic core in Alg. 1 of KPM, ChebTP, and ChebFD. Depending on which operations are used in lines 4/5 and 10/11, it serves two different purposes: replace x_k by $f[A]x_k$ (lines 4,10), or compute moments $\{c_n^{(k)}\}$ (lines 5,11). Algorithm 1 computes the polynomials $p_n[A]x_k$ for several vectors $x_1, \ldots, x_M$ simultaneously, as required in KPM and ChebFD. In addition to spMVM it uses only BLAS-1 vector operations within the two loops over k (vector index) and n (polynomial degree). Owing to this simplicity, the algorithmic core allows for effective performance engineering through straightforward optimizations such as loop-fusion. A particularly rewarding step is the combination of the individual spMVMs for $k = 1, \ldots, M$ into spMMVMs on block vectors, which improves cache utilization due to less erratic memory access patterns. Row-major storage of `mVecs` (as implemented in GHOST) is the key to reaping the full benefits of this optimization [17,24]. With such node-level optimizations one can achieve decoupling of the algorithmic core performance from main memory bandwidth on modern CPU systems. Then, the overall performance depends only on the distributed sp(M)MVMs, i.e., is bounded by the inter node communication bandwidth and latency.

Notice that Alg. 1 has no internal synchronization points, because neither the dot products in lines 5/11 nor the vectors accumulated in lines 4/10 are used in the following iterations steps. Global synchronization can be delayed until after the execution of the entire algorithmic core, and thus does not affect scalability.

Apart from KPM, Algorithm 1 is normally executed repeatedly. In ChebTP intermediate computations between different executions usually consist of a few `xDOT` operations, and can be delegated to separate tasks. The results are not needed in the next iterations, and (global) synchronization still is not required. In ChebFD, however, vectors have to be orthogonalized between subsequent executions of the algorithmic core. We use communication-avoiding techniques such as TSQR [25] or SVQB [26] to mitigate the ensuing adverse effects on performance.

The potential of the ESSR implementations of KPM, ChebTP, and ChebFD was demonstrated in a series of papers [17,24,27]. With the fully heterogeneous CPU-GPU implementation of KPM [24] we computed the density of states of a matrix with dimension $D = 6.5 \times 10^9$ on 1024 hybrid nodes of the Piz Daint supercomputer[10]. Performance engineering resulted in a speed up of 3–5 at the single node level [27]. Recently, these computations were extended to 4096 nodes ($D = 10^{10}$) and achieved 0.5 Pflop/s sustained performance [28], which corresponds to 11% of LINPACK efficiency. With the ChebFD implementation we could compute the 148 innermost eigenvalues of a matrix with dimension $D = 10^9$, using 512 nodes of SuperMUC[11] at 40 Tflop/s sustained performance [17]. With the full SuperMUC phase 2 we will be able to obtain inner eigenvalues for matrix dimensions 10^{10}, at an expected sustained performance level of 250 Tflop/s.

The only remaining bottleneck for our polynomial algorithms is the performance of the distributed sp(M)MVMs. In many quantum physics applications (see Sect. 3.3) the inter node communication volume grows strongly with matrix dimension, and reduction of communication is the most crucial issue for scalability. For stencil type matrices, techniques such as octree ordering are used [18]. For more complex sparsity patterns, GHOST allows sparse matrix repartitioning by PT-Scotch [29]. Future versions of the ESSR will include scalable matrix reordering techniques tailored to the application matrices.

4.2 Beyond FEAST: Projection Based Methods

Consider the (generalized) eigenvalue problem $AX = \Lambda BX$. FEAST [3] is a subspace iteration method to compute all eigenvalues inside a user-defined interval I_λ, and their corresponding eigenvectors. In each step, a size-m search space Y is projected approximately onto the desired invariant subspace, and a Rayleigh-Ritz procedure is used to compute approximate eigenpairs. The computed eigenvectors serve as the new refined search space and the scheme is iterated until convergence. The projection is achieved by (numerical) integration of the resolvent $(zB - A)^{-1}B$ over a contour in the complex plane that encloses I_λ, but no other eigenvalues of (A, B); see [3] for more details and [30] for recent variants. The ESSEX

[10] `http://www.cscs.ch/computers/pizdaint/index.html`

[11] `https://www.lrz.de/services/compute/supermuc/`

project has contributed to improving FEAST in two ways: by proposing techniques for solving or avoiding the linear systems that arise, and by improving robustness and performance of the algorithmic scheme.

Linear systems. Our intended use of the FEAST adaptations in BEAST is computing up to 1 000 interior eigenpairs of very large and sparse Hermitian matrices. This use case is not well-supported by other FEAST implementations as they typically rely on direct sparse solvers for the linear systems that arise. We use two strategies to overcome this problem: (i) a robust and scalable iterative solver for the linear systems in contour integration based BEAST (BEAST-C, [31]), and (ii) use of polynomial approximation as an alternative to contour integration (BEAST-P, [32]). A rough layout of algorithmic key steps in BEAST is presented in Algorithm 2; see [32] for a more detailed formulation.

Algorithm 2 Basic BEAST projection-based eigensolver

```
Input: Interval $I_\lambda$, Matrix pair $A, B \in \mathbb{C}^{N\times N}$
Output: $\hat{m}$ eigenpairs $(X, \Lambda)$ in $I_\lambda$
Estimate $\tilde{m} \approx \hat{m}$, choose random $Y \in \mathbb{C}^{N\times m}$ of rank $m > \tilde{m}$
while not $\tilde{m}$ pairs converged do
    Compute $U = PY$ with suitable projector $P = P_{I_\lambda}(A, B)$
    Compute Rayleigh quotients $A_U = U^*AU$ and $B_U = U^*BU$
    Update estimate $\tilde{m}$ of $\hat{m}$ and adjust $m > \tilde{m}$
    Solve EVP $A_U W = B_U W \Lambda$
    $X \leftarrow UW$
    Orthogonalize $X$ against locked vectors and lock newly converged vectors
    $Y \leftarrow BX$
```

The linear systems arising in BEAST-C have the form $(zB - A)X = F$, with a possibly large number of right-hand sides F. The complex shifts z get very close to the spectrum, making these systems very ill-conditioned. For interior eigenvalue computations, the system matrix also becomes completely indefinite. For these reasons, standard preconditioned iterative solvers typically fail in this context [31,33]. In [31] we demonstrated that an accelerated parallel row-projection method called CARP-CG [34] is well suited for highly indefinite systems arising in this context, and particularly apt at handling small diagonal elements, which are common in our applications. We also proposed a hybrid parallel implementation of the method, which is available as a prototype in the PHIST builtin kernels.

Matrix inversion can be avoided altogether if the projector can be acquired by means other than numerical integration or rational approximation. A common choice is spectral filtering using Chebyshev polynomials via the ChebFD scheme [17], see Sect. 4.1, in particular for the discussion of kernel functions for reducing Gibbs oscillations [20, 33]. This is implemented in the BEAST-P variant, available through PHIST and GHOST.

General improvements. The size of the search space is crucial for the convergence of the method [3,17,33,35]. In BEAST we compute a suitable initial guess of the number of eigenpairs in the target interval by integrating the density of states obtained by the KPM (cf. 4.1). The most recent version of the FEAST library uses a similar approach [36]. As iteration progresses, the search space size is controlled using singular value decomposition [32, 33, 37], that gives a more accurate estimation and consequentially a smaller searchspace. This lowers memory usage, which may be preferable for very large problems. A more generous searchspace size can be chosen to reduce the impact of the polynomial degree on convergence speed. The SVD is also used for other purposes like detecting empty intervals or undersized search spaces [33, 38].

Furthermore, a locking technique is implemented in BEAST. By excluding converged eigenpairs from the search space—at the cost of orthogonalizing the remaining vectors in each iteration—it is possible to reduce the cost of later iterations where only few eigenpairs have not yet converged [32, 33, 38].

The most influential parameters for the cost of an iteration in BEAST are the polynomial degree in BEAST-P and residual accuracy for the iterative linear solver in BEAST-C, respectively. These two parameters have different semantics for the progress of the method, though, and need separate consideration.

To minimize the overall work, BEAST-P finds a (problem-dependent) polynomial degree p that, in one BEAST iteration, achieves comparably large residual drop with respect to the number of spMVMs required to evaluate the polynomial [32]. It is then adjusted dynamically by inspecting the residual reduction versus p. This removes the necessity of an initial guess for a suitable degree and makes early iterations cheap since the optimal degree is approached from below. In BEAST-C, we reduce the target residual of the iterative linear solver [32] in early iterations. In later iterations, a higher accuracy is required to achieve a good overall approximation.

Future releases of BEAST will include extension of the method to multiple adjacent intervals (which requires careful orthogonalization and is currently in the testing stage), and the use of single-precision solves in early iterations. BEAST was successfully tested with matrices from graphene and topological insulator modeling of size up to 10^9, typically computing few hundred interior eigenpairs, using the BEAST-P variant with GHOST back end.

4.3 Block Jacobi-Davidson QR

The Jacobi-Davidson method [39] is a popular algorithm for computing a few eigenpairs of a large sparse matrix. It can be seen as a Rayleigh-Ritz procedure with subspace acceleration and deflation. Depending on some implementation details, such as the inner product used and the way eigenvalue approximations are extracted, it may be used for Hermitian and non-Hermitian, standard or generalized eigenproblems, and to find eigenpairs at the border or inside of the spectrum. The Jacobi-Davidson method has several attractive features: it exhibits locally cubic (quadratic) convergence for Hermitian (general) eigenvalue problems, and is very robust w.r.t. approximate solution of the linear systems that occur in each iteration. It also allows integrating preconditioning techniques, and the deflation of eigenvalues near the shift make the linear systems much more well-behaved than in the case of FEAST. For an overview of the Jacobi-Davidson method, see [40].

In [18, 41] we presented the implementation of a block Jacobi-Davidson QR (BJDQR) method which uses block operations to increase the arithmetic intensity and reduce the number of synchronization points (i.e. mitigate the latency of global reduction operations). Use cases for this ESSR solver include the computation of a moderate number of extremal eigenpairs of large, sparse, symmetric or nonsymmetric matrices. BJDQR is a subspace algorithm: in every iteration the search space V is extended by n_b new vectors, w_j, which are obtained by approximately solving a set of correction equations (2), and orthogonalized against all previous directions. The solution of the sparse linear systems (2) is done iteratively.

$$(I-\tilde{Q}\tilde{Q}^*)(A-\sigma_i I)(I-\tilde{Q}\tilde{Q}^*)\Delta q_i \approx -(A\tilde{q}_i - \tilde{Q}\tilde{r}_i), \qquad i = 1\ldots n_b. \qquad (2)$$

The successful implementation of this method in PHIST goes hand-in-hand with the development of highly optimized building blocks in GHOST. The basic operations required are spMMVM ($Y_j \leftarrow AX_j$) and the dense matrix-matrix products $Y = X \cdot C$ and $C = X^H Y$, where X and Y denote `mVecs` and C an `sdMat`. For the full optimization, we added several custom kernels, including the 'in place' variant $X_{:,1:k} = X \cdot C, X \in \mathbb{C}^{n\times m}, C \in \mathbb{C}^{m\times k}$ and an spMMVM with varying shifts per column, $Y_j = AX_j + \sigma_j X_j$.

Two main observations guided the implementation of this algorithm:

1. row-major storage of `mVecs` leads to much better performance of both the spMMVM, see also [42], and the dense kernels;
2. accessing single columns in an `mVec` in row-major storage is disproportionally more expensive than in column-major storage because unnecessary data is loaded into the cache.

To avoid access to single vectors, 'blocked' implementations of the GMRES and MINRES solvers for the correction equation are used. These schemes solve k linear systems simultaneously with separate Krylov spaces, bundling inner products and spMVMs. The second important phase, orthogonalization of W against V, is performed using communication optimal algorithms like TSQR [25] or SVQB [26].

The final performance critical component for Jacobi-Davidson is a preconditioning step used to accelerate the inner solver. Preconditioning techniques typically depend strongly on details of the sparse matrix storage format. As we do not want to impose a particular format on the kernel library that provides the basic operations, PHIST views the preconditioner as an abstract operator (`linearOp`). This struct contains a pointer to a data object and an `apply` function, which the application can use to implement e.g. a sparse approximate inverse, an incomplete factorization or a multigrid cycle. The only preconditioned iteration implemented directly in PHIST is CARP-CG, used in the BEAST-C algorithm in ESSEX (Sect. 4.2). This method could also be used in the context of BJDQR, but this combination is not yet implemented.

It is well known that the block variant of JDQR increases the total number of operations (measured for instance in the number of spMVMs). The ESSEX results presented in [18] demonstrated for the first time that this increase is more than compensated by the performance gains in the basic operations, so that an overall speedup of about 20% can be expected for a wide range of problems and problem sizes. The paper also shows that the only way to achieve this is by consequent performance engineering on all levels. On upcoming hardware, one can expect the benefits of the block variant over the single vector JDQR to grow because of the increasing gap between memory bandwidth and flop rate. Furthermore, the reduction in the number of synchronization points will increase this advantage on large scale systems. We will present results on the heterogeneous execution of this solver on large CPU/GPU clusters in the near future.

5 Fault Tolerance

This section describes our development and evaluation of strategies for efficient checkpointing and restarting of iterative eigenvalue solvers. The former can be done either by storing critical data on a parallel file system (PFS) or on a neighboring node. The latter depends highly on the availability of a fault tolerant communication library, and two options have been evaluated here.

Asynchronous checkpointing via dedicated threads: We use the term 'asynchronous checkpointing' for application-level checkpointing where a dedicated thread is used to transfer the checkpoint data to the PFS while the application performs its computations. The benefits of this approach over synchronous PFS-level checkpointing have been demonstrated as proof of concept in [43]. In a first step, an asynchronous copy of the critical data is made in an application (or algorithm) specific checkpoint object. The task concept available in GHOST [16] is then used for asynchronously writing the backup file to a global file system. Critical data in the context of eigensolvers may, for instance, be a basis for the (nearly) converged eigenspace. We have implemented and tested this strategy for KPM, ChebTP, ChebFD, and Lanczos solvers. The detailed analysis of this approach for the Lanczos algorithm is presented in [44] where we used dedicated OpenMP-threads for asynchronous writing.

Node-level checkpointing using SCR: A more scalable approach has been evaluated using the Scalable Checkpoint-Restart (SCR) library [45], which provides node-level checkpoint/restart mechanisms. Beside the local node-level checkpoints, SCR also provides the functionality to make partner-level and XOR-encoded checkpoints. In addition, occasional PFS-level checkpoints can be made to enable recovery from any catastrophic failures. This strategy introduces very little overhead to the application and is demonstrated in detail along with its comparison with asynchronous checkpointing in [44,46]. Within the ESSR, we have equipped KPM, ChebTP, and Lanczos algorithms with this checkpointing strategy.

Automatic Fault Recovery: The automatic fault recovery (AFR) concept is to enable the application to 'heal itself' after a failure. The basic building block of the concept is a fault-tolerant (FT) communication library. As an FT MPI implementation was not (yet) available, we used the GASPI communication layer [47] to evaluate the concept in a conjugate gradient (CG) solver [48].

As a next step, we evaluated a recent prototype of FT MPI—'User-Level Failure Mitigation' or ULFM [49]—in the context of the KPM with automatic fault recovery. In this implementation, we combined the AFR technique with node-level checkpointing using SCR. The failed processes are replaced by newly spawned ones which take over the identity (i.e., rank) of the failed processes in a rebuilt communicator. All processes then read a consistent copy of the checkpoint from the local or neighbor's memory and resume the computation. Experimental results on this approach are currently being prepared for publication.

6 Summary and Outlook

We have discussed the development of a new software repository for extreme scale sparse eigenvalue computations on heterogeneous hardware. One key challenge of the project was to co-design several interdependent software layers 'from scratch'. We described a simple layered software architecture and a flexible test-driven development process which enabled this. The scalability challenge is addressed by holistic performance engineering and redesigning algorithms for better data locality and communication avoidance. Techniques for mitigating hardware failure were investigated and implemented in prototypical iterative methods.

While this report focused on the software engineering process and algorithmic advancements, we have submitted a second report which demonstrates the parallelization strategy as well as hardware and energy efficiency of our basic building block library GHOST, see [1].

In order to achieve scalability beyond today's Petascale computers, we are planning to investigate (among other) scalable communication reducing orderings for our application matrices, communication hiding using the tasking mechanism in our GHOST library, and scalable preconditioners in GHOST for accelerating BEAST-C and Jacobi-Davidson, for instance based on the prototype of CARP-CG in the PHIST builtin kernel library. Future applications will include non-Hermitian matrices and generalized eigenproblems, which requires extensions to some of the algorithms. We are also planning to further integrate our efforts and improve the software structure and documentation to bring forth an ESSL (Exascale Sparse Solver Library).

Acknowledgments. This work was supported by the German Research Foundation (DFG) through the Priority Programs 1648 "Software for Exascale Computing" under project ESSEX. We would like to thank Michael Meinel (DLR Simulation and Software Technology, software engineering group) for helpful comments on the manuscript.

References

1. Kreutzer, M., Hager, G., Wellein, G.: Optimal energy efficiency by performance engineering for a kernel polynmial method algorithm on a multicore cluster, submitted to the same journal issue
2. Lehoucq, R.B., Yang, C.C., Sorensen, D.C.: ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods. SIAM, Philadelphia (1998)
3. Polizzi, E.: A density matrix-based algorithm for solving eigenvalue problems. Phys. Rev. B 79, 115112 (2009)
4. Stathopoulos, A., McCombs, J.R.: PRIMME: preconditioned iterative multimethod eigensolver–methods and software description. ACM Trans. Math. Softw. 37, 1–30 (2010)
5. Hernandez, V., Roman, J.E., Vidal, V.: SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Trans. Math. Software 31, 351–362 (2005)
6. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zampini, S., Zhang, H.: PETSc Web page (2015), http://www.mcs.anl.gov/petsc
7. Baker, C.G., Hetmaniuk, U.L., Lehoucq, R.B., Thornquist, H.K.: Anasazi software for the numerical solution of large-scale eigenvalue problems. ACM Trans. Math. Softw. 36, 1–23 (2009)
8. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring,

J.M., Williams, A., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31, 397–423 (2005)
9. Heroux, M.A., Willenbring, J.M.: A new overview of the Trilinos project. Sci. Program. 20 (2012)
10. J. Daly et al.: Inter-Agency Workshop on HPC Resilience at Extreme Scale. Tech. rep. (2012)
11. Hursey, J.: Coordinated Checkpoint/Restart Process Fault Tolerance for MPI Applications on HPC Systems. Ph.D. thesis, Indiana University, Bloomington, IN, USA (2010)
12. El-Sayed, N., Schroeder, B.: Reading between the lines of failure logs: Understanding how HPC systems fail. In: Proc. of the 2013 43rd Annual IEEE-IFIP Int. Conf. on Dependable Systems and Networks (DSN). pp. 1–12. DSN '13, IEEE Computer Society, Washington, DC, USA (2013)
13. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52, 65–76 (2009)
14. I. Laguna et al.: Evaluating User-Level Fault Tolerance for MPI Applications. In: Proc. of the 21st European MPI Users' Group Meeting. pp. 57:57–57:62. EuroMPI/ASIA '14, ACM, New York, NY, USA (2014)
15. TOP500 Supercomputer Sites. `http://www.top500.org`, accessed: June 2015
16. Kreutzer, M., Thies, J., Röhrig-Zöllner, M., Pieper, A., Shahzad, F., Galgon, M., Basermann, A., Fehske, H., Hager, G., Wellein, G.: GHOST: building blocks for high performance sparse linear algebra on heterogeneous systems (2015), preprint (arXiv:1507.08101)
17. Pieper, A., Kreutzer, M., Galgon, M., Alvermann, A., Fehske, H., Hager, G., Lang, B., Wellein, G.: High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations (2015), preprint (arXiv:1510.04895), submitted
18. Röhrig-Zöllner, M., Thies, J., Kreutzer, M., Alvermann, A., Pieper, A., Basermann, A., Hager, G., Wellein, G., Fehske, H.: Increasing the performance of the Jacobi-Davidson method by blocking (2014), accepted for publication in SISC
19. Treibig, J., Hager, G., Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops. pp. 207–216. ICPPW '10, IEEE Computer Society, Washington, DC, USA (2010)
20. Weiße, A., Wellein, G., Alvermann, A., Fehske, H.: The kernel polynomial method. Rev. Mod. Phys. 78, 275–306 (2006)
21. Tal-Ezer, H., Kosloff, R.: An accurate and efficient scheme for propagating the time dependent Schrödinger equation. J. Chem. Phys. 81, 3967 (1984)
22. Weiße, A., Fehske, H.: Chebyshev expansion techniques. In: Fehske, H., Schneider, R., Weiße, A. (eds.) Computational Many-Particle Physics. Lect. Notes Physics, vol. 739, pp. 545–577. Springer (2008)
23. Jackson, D.: On approximation by trigonometric sums and polynomials. Trans. Am. Math. Soc. 13, 491–515 (1912)
24. Kreutzer, M., Hager, G., Wellein, G., Pieper, A., Alvermann, A., Fehske, H.: Performance engineering of the kernel polynomial method on large-scale CPU-GPU systems. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. pp. 417–426 (2015)
25. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. SIAM J. Sci. Comp. 34, A206–A239 (2012)
26. Stathopoulos, A., Wu, K.: A block orthogonalization procedure with constant synchronization requirements. SIAM J. Sci. Comp. 23, 2165–2182 (2002)
27. Alvermann, A., Basermann, A., Fehske, H., Galgon, M., Hager, G., Kreutzer, M., Krämer, L., Lang, B., Pieper, A., Röhrig-Zöllner, M., Shahzad, F., Thies, J., Wellein, G.: ESSEX: Equipping sparse solvers for exascale. In: Lopes, L., et al. (eds.) Euro-Par 2014: Parallel Processing Workshops, Lecture Notes in Computer Science, vol. 8806, pp. 577–588. Springer International Publishing (2014)
28. Kreutzer, M., Pieper, A., Alvermann, A., Fehske, H., Hager, G., Wellein, G., Bishop, A.R.: Efficient large-scale sparse eigenvalue computations on heterogeneous hardware (2015), `http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post205.html`, poster at the 2015 ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis
29. (PT-)SCOTCH project website, `http://www.labri.fr/perso/pelegrin/scotch/`
30. Polizzi, E., Kestyn, J.: High-performance numerical library for solving eigenvalue problems: FEAST eigenvalue solver v3.0 user guide (2015), `http://arxiv.org/abs/1203.4031`

31. Galgon, M., Krämer, L., Thies, J., Basermann, A., Lang, B.: On the parallel iterative solution of linear systems arising in the FEAST algorithm for computing inner eigenvalues. J. Par. Comp. 49, 153–163 (2015)
32. Galgon, M., Krämer, L., Lang, B.: Adaptive choice of projectors in projection based eigensolvers (2015), submitted, available from `http://www.imacm.uni-wuppertal.de/`
33. Krämer, L.: Integration based solvers for standard and generalized Hermitian eigenvalue problems. Ph.D. thesis, Bergische Universität Wuppertal (2014), `http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:468-20140701-112141-6`
34. Gordon, D., Gordon, R.: CARP-CG: A robust and efficient parallel solver for linear systems, applied to strongly convection dominated PDEs. J. Par. Comp. 36, 495–515 (2010)
35. Krämer, L., Di Napoli, E., Galgon, M., Lang, B., Bientinesi, P.: Dissecting the FEAST algorithm for generalized eigenproblems. J. Comp. Appl. Math. 244, 1–9 (2013)
36. Di Napoli, E., Polizzi, E., Saad, Y.: Efficient estimation of eigenvalue counts in an interval (2013), preprint (arXiv:1308.4275)
37. Galgon, M., Krämer, L., Lang, B., Alvermann, A., Fehske, H., Pieper, A.: Improving robustness of the FEAST algorithm and solving eigenvalue problems from graphene nanoribbons. Proc. Appl. Math. Mech. 14, 821–822 (2014)
38. Galgon, M., Krämer, L., Lang, B.: Counting eigenvalues and improving the integration in the FEAST algorithm (2012), preprint BUW-IMACM 12/22, available from `http://www.imacm.uni-wuppertal.de`
39. Fokkema, D.R., Sleijpen, G.L.G., van der Vorst, H.A.: Jacobi–Davidson style QR and QZ algorithms for the reduction of matrix pencils. SIAM J. Sci. Comp. 20, 94–125 (1998)
40. Hochstenbach, M.E., Notay, Y.: The Jacobi-Davidson method. GAMM-Mitteilungen 29, 368–382 (2006)
41. Röhrig-Zöllner, M., Thies, J., Kreutzer, M., Alvermann, A., Pieper, A., Basermann, A., Hager, G., Wellein, G., Fehske, H.: Performance of block jacobi-davidson eigensolvers (2014), poster at 2014 ACM/IEEE Int. Conf. on High Performance Computing Networking, Storage and Analysis
42. Gropp, W.D., Kaushik, D.K., Keyes, D.E., Smith, B.F.: Towards realistic performance bounds for implicit CFD codes. In: Proceedings of Parallel CFD'99. pp. 233–240. Elsevier (1999)
43. Shahzad, F., Wittmann, M., Zeiser, T., Wellein, G.: Asynchronous checkpointing by dedicated checkpoint threads. In: Proc. of the 19th European conf. on Recent Advances in the Message Passing Interface. pp. 289–290. EuroMPI'12, Springer-Verlag, Berlin, Heidelberg (2012)
44. Shahzad, F., Wittmann, M., Kreutzer, M., Zeiser, T., Hager, G., Wellein, G.: A survey of checkpoint/restart techniques on distributed memory systems. Parallel Processing Letters 23, 1340011–1 – 1340011–20 (2013)
45. K. Sato et al.: Design and modeling of a non-blocking checkpointing system. In: Proc. of the Conf. on High Performance Computing, Networking, Storage and Analysis. pp. 19:1–19:10. IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
46. Shahzad, F., Wittmann, M., Zeiser, T., Hager, G., Wellein, G.: An evaluation of different I/O techniques for checkpoint/restart. In: Proc. of the 2013 IEEE Int. Par. and Dist. Processing Symp. (IPDPS). pp. 1708–1716. IEEE Computer Society (2013)
47. GASPI project website: `http://www.gaspi.de/en/project.html`
48. Shahzad, F., Kreutzer, M., Zeiser, T., Machado, R., Pieper, A., Hager, G., Wellein, G.: Building a fault tolerant application using the GASPI communication layer. In: Proceedings of the 1st Int. Workshop on Fault Tolerant Systems (FTS 2015), in conjunction with IEEE Cluster 2015. pp. 580–587 (2015)
49. Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J.: An Evaluation of User-Level Failure Mitigation Support in MPI. In: Jesper, T., Benkner, S., Dongarra, J. (eds.) Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science, vol. 7490, pp. 193–203. Springer Berlin Heidelberg (2012)