# PHIST: a Pipelined, Hybrid-parallel Iterative Solver Toolkit

JONAS THIES, MELVEN RÖHRIG-ZÖLLNER, NIGEL OVERMARS, and ACHIM BASERMANN,
German Aerospace Center (DLR), Simulation and Software Technology
DOMINIK ERNST, GEORG HAGER, and GERHARD WELLEIN,
Erlangen Regional Computing Center (RRZE), University of Erlangen-Nuremberg

The increasing complexity of hardware and software environments in high-performance computing poses big challenges on the development of sustainable and hardware-efficient numerical software. This paper addresses these challenges in the context of sparse solvers. Existing solutions typically target sustainability, flexibility or performance, but rarely all of them.

Our new library PHIST provides implementations of solvers for sparse linear systems and eigenvalue problems. It is a productivity platform for performance-aware developers of algorithms and application software with abstractions that do not obscure the view on hardware-software interaction.

The PHIST software architecture and the PHIST development process were designed to overcome shortcomings of existing packages. An interface layer for basic sparse linear algebra functionality that can be provided by multiple backends ensures sustainability, and PHIST supports common techniques for improving scalability and performance of algorithms such as blocking and kernel fusion.

We showcase these concepts using the PHIST implementation of a block Jacobi-Davidson solver for non-Hermitian and generalized eigenproblems. We study its performance on a multi-core CPU, a GPU and a large-scale many-core system. Furthermore, we show how an existing implementation of a block Krylov-Schur method in the Trilinos package Anasazi can benefit from the performance engineering techniques used in PHIST.

**ACM Reference Format:**
Jonas Thies, Melven Röhrig-Zöllner, Nigel Overmars, Achim Basermann, Dominik Ernst, Georg Hager, and Gerhard Wellein. 2020. PHIST: a Pipelined, Hybrid-parallel Iterative Solver Toolkit. 1, 1 (September 2020), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Iterative solvers for sparse linear systems and eigenvalue problems are common components of many simulations and often take a significant portion of the overall runtime. There exist a variety of libraries providing basic linear algebra data structures and operations (called kernels subsequently), and implementations of iterative solvers. We will list a few efforts in order to motivate the development of a new library below. PHIST originated in an Exa-scale eigensolver project[1] and therefore has a focus on linear eigenproblems up to now, but the close relation between the two classes of linear algebra problems allows us to also address linear systems to some extent. Future work may lead more in

[1]ESSEX, https://blogs.fau.de/essex/

Authors' addresses: Jonas Thies, Jonas.Thies@DLR.de; Melven Röhrig-Zöllner; Nigel Overmars; Achim Basermann,
German Aerospace Center (DLR), Simulation and Software Technology, Linder Höhe
51147 Cologne, Germany; Dominik Ernst; Georg Hager; Gerhard Wellein,
Erlangen Regional Computing Center (RRZE), University of Erlangen-Nuremberg, Martensstraße 6, 91054 Erlangen, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
Manuscript submitted to ACM

the direction of linear solvers and preconditioners as well. An early version of PHIST and some related software was described in [Thies et al. 2016], and algorithmic and performance details on our block Jacobi-Davidson implementation can be found in [Röhrig-Zöllner et al. 2015].

### 1.1 Related software

The two most well-known open source software frameworks in the field of high performance numerical linear algebra are PETSc [Balay et al. 2016] and Trilinos [Heroux et al. 2005]. PETSc provides MPI-only implementations of both the kernel and solver level, focusing on linear systems. This effort is augmented by the SLEPc library [Hernandez et al. 2005], which provides eigensolvers based on PETSc. In this framework it is not straight-forward to integrate faster kernel operations but one has to rely on the PETSc developers to do a good job.

Trilinos is organized in interoperable subpackages. Epetra and Tpetra [Baker and Heroux 2012] provide data structures and kernels using MPI and 'MPI+X' parallelization, respectively, i.e. the aim of Tpetra is to support thread-level parallelism, GPUs and future technology on the node level while employing MPI between the nodes of a cluster. The other two Trilinos packages we should mention are called Anasazi [Baker et al. 2009] and Belos, the former provides iterative solvers for linear eigenvalue problems, the latter Krylov methods for linear systems (with preconditioning provided by other packages). Algorithms in these libraries are implemented using an abstraction layer that can be provided by any linear algebra framework supporting 'multi-vectors', i.e. very tall and skinny matrices. We have adopted a similar but more expressive abstraction layer in PHIST, and since the operations required by Belos and Anasazi are a subset of ours, we can integrate their algorithm implementations in PHIST, as will be shown in Section 5.1.

In the field of sparse eigensolvers, two more libraries should be mentioned. The hugely popular ARpack [Lehoucq et al. 1998] (and its MPI-parallel version PARpack) implements the implicitly restarted Arnoldi method. It uses a reverse communication interface (RCI) so that the user does not need to be aware of the underlying data structures. PRIMME [Stathopoulos and McCombs 2010] implements variants of the Davidson method (Jacobi-Davidson QMR, Generalized Davidson+$k$) but is restricted to symmetric/Hermitian problems. Both ARpack and PRIMME rely on the BLAS library for process-level operations (except for the sparse matrix-vector and preconditioning operations) and expose raw data arrays to the user for applying operators. Unfortunately, BLAS implementations typically perform poorly for tall and skinny matrices because they are optimized for the compute-bounded case. And the way the libraries allocate memory for vectors and expose it to the user makes it very difficult to efficiently use NUMA machines or accelerators like GPUs.

### 1.2 Performance optimization of sparse solvers

For many years, HPC users are experiencing the effects of the impending end of Moore's law. Hardware performance improvements happen mostly on the node level by increasing the complexity of the memory subsystem and parallelizing the low (SIMD/SIMT) and intermediate levels (more cores). There are (at least) three approaches to helping programmers tackle this increased complexity: (i) tasking frameworks that perform runtime scheduling and allow the user to specify his program as a series of code blocks with input and output dependencies; (ii) provide an expressive 'language' that hides the underlying complexity by automatically generating code for different hardware, and (iii) provide highly optimized libraries that use the full expressiveness of programming languages like CUDA. Some examples the approaches are

(i) PLASMA[2] and MAGMA[3],

[2] https://bitbucket.org/icl/plasma
[3] http://icl.cs.utk.edu/magma/

(ii) RAJA[4], Alpaka[5] and the Trilinos library Kokkos (which is used for the node-level parallelization of Tpetra used in some examples below),
(iii) (cu)BLAS and our own GHOST library [Kreutzer et al. 2017], see also Section 2.6.

In addition to the low-level optimization, algorithm researchers are working out ways to increase the performance of iterative schemes on modern hardware. Many efforts are focused on reducing the cost of synchronizations in an algorithm. 'Communication avoiding' Krylov methods (e.g. [Hoemmen 2010], [Mohiyuddin et al. 2009]) are based on the idea of $s-$step methods, which use other than Krylov bases for some steps and then add the generated block to a Krylov subspace. The advantage is that fewer single vectors need to be orthogonalized against the existing basis, which reduces the number of global synchronizations. Another class of methods that receives significant attention in the HPC community are the so-called 'pipelined' Krylov methods (see e.g. [Ghysels et al. 2013]). These techniques rearrange operations in order to be able to overlap the communication/synchronization required for inner products with computations during the sparse matrix-vector operation.

From a mathematical point of view, both approaches change the underlying polynomials of the algorithms and may infringe the numerical robustness. In particular for non-Hermitian eigenvalue problems we therefore did not focus our research so far on such approaches but follow down the numerically robust but fully optimized path here. In particular we want to stress that the term 'pipelined' in the name of our software refers to a wide range of techniques to improve the computational performance, from enabling low-level SIMD operations to block eigensolvers that solve for a number of eigenpairs in a pipelined way. We do have basic support for overlapping e.g. reductions with the sparse matrix-vector product, but are not using them so far in algorithm implementations. Even without sacrificing numerical stability, though, it is possible to significantly improve the performance. PHIST is designed to facilitate various algorithm-level optimizations, which will be described in Section 3.

The remainder of this paper is organized as follows. Section 2 gives an overview of the entire PHIST software, and summarizes some related software that can be used to extend the functionality of PHIST. In Section 3 we discuss possibilities to improve the performance of iterative solvers on HPC systems, and how PHIST supports their implementation. As an example we show how the performance of the block Krylov-Schur method can be improved by a combination of fast kernels and a block orthogonalization scheme. In Section 4 we introduce the software architecture of PHIST and motivate it by a test- and benchmark-driven development process for HPC codes. Section 5 describes some details of the eigensolvers available in PHIST, along with some node-level performance results. Scalability on a many-node/many-core system is investigated in Section 6. Section 7 concludes the paper with a summary and an outlook on future work.

## 2 OVERVIEW OF PHIST

In this section we describe the software architecture and basic interface of PHIST, and give an overview of the functionality currently available at the algorithm level. PHIST was developed alongside several other libraries which are not covered in this paper but can be used to add functionality to the basic package. Some of these libraries are included as subdirectories in the PHIST software, and we will briefly describe them in Section 2.6.

[4] https://github.com/LLNL/RAJA
[5] https://github.com/ComputationalRadiationPhysics/alpaka

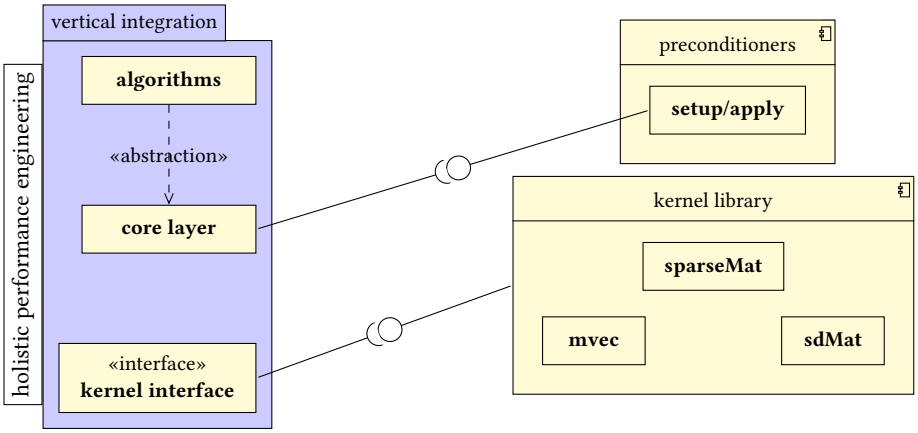


Fig. 1. Software architecture of PHIST. The components on the left can be provided by external libraries, the right-hand side constitutes the PHIST framework, which can also be extended by other libraries.

### 2.1 Software architecture

Our software architecture consists of three layers, as shown in Figure 1: the algorithms layer, the (algorithmic) core and the kernels (or computational core). A kernel interface hides the low-level implementations of basic linear algebra from the higher layers. To maximize portability and performance, the upper two layers can not access objects with significant amounts data, like sparse matrices or (multi-)vectors. For example, a Gram-Schmidt process only requires basic operations like vector additions and inner products, so it will be implemented in the algorithmic core layer. A Gauß-Seidel sweep requires access to individual matrix entries and a hardware-specific implementation, so it would belong to the computational core. The kernel library takes care of all levels of parallelism (e.g. SIMD, multi-threading, accelerator usage and inter-process communication). This means that the core and algorithm layers can be largely oblivious of e.g. whether the present machine uses distributed memory, contains GPUs etc. The kernel interface can be provided by any linear algebra library supporting the required operations, see Section 2.2.

The core layer provides implementations of common building blocks of high-level solvers. Examples are routines for orthogonalizing vector spaces, computing a matrix polynomial or factorizing a small and dense matrix.

At the top level, iterative methods are implemented using the kernel and core layer operations. The aim is to allow algorithm researchers to work only on the top two layers, while low-level HPC experts can implement the required kernels. Collaboration between the two groups follows a test- and benchmark-driven development process, detailed in Section 4.

A fourth component of the software architecture is the preconditioning interface. Typically, there is a close interaction between the kernel library and more advanced preconditioning techniques like multigrid or incomplete factorization methods. We therefore allow preconditioners to be based on a particular kernel library, in which case they can only be used with that kernel library, unless additional ‘glue code’ is used to e.g. convert sparse matrix formats or apply an operator to another multi-vector class.

### 2.2 Basic data structures and interface

Our basic interface is inspired by the Message Passing Interface (MPI), and the Petra object model used in Trilinos [Heroux et al. 2003]. Independent of the underlying kernel library, the primary interface of PHIST is in plain C. From this, C++, Python and Fortran 2003 bindings are generated[6]. The C++ bindings allow code which is templated on the scalar data type to conveniently call the appropriate PHIST functions via a static class template instead of having to generate the C function names using macros. The first and last PHIST functions called by a program should always be `phist_kernels_init` and `phist_kernels_finalize`.

Macros are used to generate function names for different data types, e.g. functions taking real-valued double precision data start with `phist_D`. The data types supported depend on the kernel library, but in principle there are four (`S` and `D` for real single and double precision, and `C` and `Z` for complex single and double precision). An example of this approach is shown in Listing 2. Listing 3 and 4 shows the same code snippet using the C++ and Fortran bindings, respectively (note that these are only code snippets, not complete programs). The basic error handling mechanism is an integer error code returned as the last argument (`iflag`) of each function, and a macro is available for checking this flag and printing an error message. The `iflag` argument serves a dual purpose in PHIST: it can also be used to encode 'hints' for the underlying implementation of the function, e.g. in Example 1, one could set `iflag=PHIST_SPARSEMAT_OPT_BLOCKSPMVM|PHIST_SPARSEMAT_PERM_GLOBAL` on line 9. This will tell the kernel library that we are primarily intending to compute sparse matrix products with multiple vectors at the same time, and it may optimize its storage format for this case. Furthermore, we allow the kernel library to repartition the matrix in case the feature is available.

The Petra object model defines a hierarchy of objects, including high-level linear algebra and low-level communication data structures. We only adopt a subset of these objects, namely:

- `comm` abstracts the MPI_Comm so that it is in principle possible to implement the kernels using some other communication layer;
- `map` defines the ordering and distribution ('index space') of rows of a sparse or dense matrix across processes;
- `sdMat` represents a small and dense matrix that is local to each process. It must be stored in column-major order and use a constant column stride larger than or equal to the number of rows. Operations on `sdMats` are assumed to be cheap. In the Petra model this object is a special case of an `mvec`;
- `mvec`, a multi-column vector or 'tall and skinny matrix', stored either in row- or column major order, depending on the kernel library used. The row resp. column stride must be constant and may be larger than the respective dimension of the object;
- `sparseMat` large and sparse matrix object which is presently mainly accessible via sparse matrix-vector multiplication (spMVM) in PHIST.

The ordering and distribution of the elements of an `mvec` are defined by a `map`. In PHIST this is a much waker construct than in the Petra object model, where maps can represent partial and overlapping index spaces, e.g. to describe the global column indices living on each process ('column map'). For (sparse) matrices, we use an additional object called `context` for this purpose. It provides access to map objects for creating `mvec` objects compatible with the matrix and may store additional information e.g. on communication patterns required for the spMVM. There exists a default implementation that defines the context to consist of four maps, defining the row- and column index spaces of the matrix, and the index spaces of the vectors $X$ and $Y$, respectively, if $X = AY$ is a valid sparse matrix-vector product.

[6] Generating the Fortran 2003 bindings is possible in PHIST 1.7.2 and later

The `context` is also needed if one wants to create a second matrix B such that $X = BY$ is again a valid spMVM. The capabilities of the map object are left mostly to the kernel library, for PHIST a simple object suffices that stores the processor offsets for a distributed global linear index space.

Similar to MPI, the above objects are passed to functions via handles. Their implementation details are left mostly to the kernel library. In addition, PHIST defines a C struct called `linearOp`, which in the simplest case wraps a `sparseMat` or a preconditioner, but also allows to use PHIST solvers in a matrix-free way.

The PHIST kernel interface is more extensive than the interface layers of Belos and Anasazi, which only contain functions needed for implementing iterative solvers for given operators. PHIST's kernel interface is meant to allow complete applications to stay independent of the backend used, so it also contains methods to create and fill sparse matrices, to copy data to and from multi-vectors, or write them to a file. In contrast to Belos/Anasazi, we also have an object for small and dense matrices, which almost doubles the number of functions required.

Another set of arithmetic functions is added to allow specific optimizations for our solvers. For these, a default implementation yielding correct behavior but suboptimal performance is provided. An example is the generalized scaling routine $V \leftarrow V \cdot C, V \in \mathbb{R}^{N \times m}, C \in \mathbb{R}^{m \times m}$, for which the default implementation uses a temporary copy of $V$. Other examples of such functions include fused kernels computing two or more quantities simultatneously (see Section 3 below).

In order to support accelerator hardware that requires explicit data transfers to and from main memory (e.g. GPUs), there are functions like `sdMat_from_device`, which must be called in the appropriate locations in an algorithm in order to be able to run on such hardware. This is different from e.g. the Tpetra library, which keeps track of whether the host- and device-side need synchronization. This approach is certainly more convenient but may also lead to performance bugs which have to be tracked down using e.g. a profiling tool.

In total, Anasazi requires 23 functions to be implemented, most of which are for multi-vector operations. Implementing the full PHIST interface requires about four times as many functions, but the main increase comes from the requirement to implement the `sdMat` interface (which is sequential and not performance critical) and advanced kernels which have a default implementation. If one only wants to use the iterative solvers (and not the full application interface), the functionality required for Anasazi and PHIST is roughly the same. The look and feel of PHIST code is illustrated in Listings 1–4 for different supported programming languages.

```c
#include "phist_kernels.h"


phist_comm_ptr comm=NULL;
phist_DsparseMat_ptr A=NULL;
int iflag=0;
phist_comm_create(&comm,&iflag);
assert(iflag==0);
iflag=0;
phist_DsparseMat_read_mm(&A, "my_matrix.mm", &iflag); assert(iflag==0);
```

Listing 1. Example 1: C code for reading a sparse matrix from a MatrixMarket file (in real double precision arithmetic).

```
#include "phist_kernels.h"
#include "phist_macros.h"
#include "phist_gen_d.h"
phist_comm_ptr comm=NULL;
TYPE(sparseMat_ptr) A=NULL;
int iflag=0;
PHIST_CHK_IERR(phist_comm_create(&comm,&iflag),iflag);
iflag=0;
PHIST_CHK_IERR(SUBR(sparseMat_read_mm)
        (&A, "my_matrix.mm",&iflag),iflag);
```

Listing 2. Example 1, C code using PHIST macros for generating type-specific code and checking return flags.

```
#include "phist_types.hpp"
#include "phist_kernels.hpp"

using namespace phist;
phist::comm_ptr comm = nullptr;
types<double>::sparseMat_ptr A = nullptr;
int iflag = 0;
try {
phist_comm_create(&comm, &iflag);
kernels<double>::sparseMat_read_mm
        (&A, comm, "my_matrix.mm", &iflag);
} catch (Exception const& ex) {...}
```

Listing 3. Example 1 using the C++ bindings.

```
#include "phist_fort.h"

use, intrinsic :: iso_c_binding
use phist_kernels
use phist_kernels_d
implicit none

type(phist_comm_ptr) comm
type(phist_DsparseMat_ptr) A
integer(c_int) iflag

iflag=0
call phist_comm_create(comm,iflag)
iflag=0
call phist_DsparseMat_read_mm &
         (A, C_CHAR_"my_matrix.mm"//C_NULL_CHAR,iflag);
if (iflag/=0) STOP 'error in PHIST'
```

Listing 4. Example 1 using the Fortran 2003 bindings.

*Kernel libraries currently supported.* In order to make PHIST self-contained, there is a reference implementation of the kernel interface using Fortran 2003, OpenMP and MPI. This reference implementation uses the CRS format for sparse matrix storage, row-major multi-vectors (see Section 3) and some pre-compiled kernels for the common block sizes $n_b = 1, 2, 4$ and 8 (e.g., sparse matrix times $n_b$-column `mvecs` and different matrix products involving two or more `mvecs` with $n_b$ columns). It gives decent performance on multi- and manycore machines, as we will show later, and could be fully integrated in Fortran applications. These 'builtin' kernels only support the real double precision data type.

A performance-oriented alternative is GHOST [Kreutzer et al. 2017], which uses MPI, OpenMP and CUDA to support a wider range of hardware. GHOST implements all four data types (i.e. single and double precision, real and complex), and `mvecs` in either row- or column-major storage. While GHOST is not a part of PHIST, we include some performance results in this paper to make the point that especially on GPUs there is much optimization potential in mainstream libraries.

In order to be useful for a wide range of application codes, we also support several popular HPC libraries mentioned in the introduction; PETSc, Trilinos (Epetra and Tpetra), Eigen[7] and MAGMA. For MAGMA, only a subset of the functions are implemented because we realized that using GPUs in this context only makes sense with fully optimized kernels as provided by GHOST, and applications are typically not built directly on top of MAGMA. All interfaces are regularly tested for regressions, see also Section 4.

### 2.3 Core functionality

In this layer we currently have implementations of some factorizations of small and dense matrices (e.g. Cholesky, Schur and singular value decompositions), which mainly make use of the LAPACK library. Furthermore, there are routines for orthogonalizing multi-vectors (see Section 3.1), and a Chebyshev method for counting eigenvalues in an interval (the so-called Kernel Polynomial Method, KPM [Weiße et al. 2006]).

Furthermore, our operator interface (`linearOp`) is implemented in the core layer, along with functions to e.g. wrap a sparse matrix, a pair of matrices or construct a product of several operators.

### 2.4 Preconditioning interface

In principle all that is needed to use a preconditioner in PHIST is the light-weight `linearOp` interface. However, in order to provide a simple interface for using methods available in (or based on) a particular kernel library, we provide an interface for constructing and updating the `linearOp` wrapper given a C enumerated type and a character string, which may e.g. contain the name of an option file or some other specification for the underlying method. The most important function for the user in this interface is `precon_create` (prefixed by e.g. phist_D). In our primary eigenvalue solver (the Jacobi-Davidson method) an approximation of the kernel of the operator to be preconditioned is available, and some preconditioners may be able to exploit such information. The interface therefore also allows providing the approximate null space of the operator as an `mvec` when creating or updating the preconditioner.

Internally, a C++ traits class called `PreconTraits` is used to implement the necessary interfaces. Methods currently supported are the Ifpack and ML packages (for Epetra), and the Ifpack2 library (for Tpetra). Users can specialize the class template for the enum value `USER_PRECON` in order to extend the functionality with their own preconditioner.

[7] https://github.com/eigenteam/eigen-git-mirror

### 2.5 High-level algorithms (solvers)

In this category PHIST provides on the one hand interfaces to the Trilinos packages Belos and Anasazi, so that linear solvers like block CG and GMRES, and eigensolvers like block Krylov-Schur or LOBPCG can be used via the PHIST interface, and with any PHIST kernel library. The central eigensolver in PHIST is the block Jacobi-Davidson QR (BJDQR) method called `subspacejada`, which can be used for solving generalized and non-Hermitian eigenproblems. Some implementation details will be discussed in Section 5.

BJDQR requires the solution of a set of linear systems $(A - \sigma_j B)x_j = -r_j, j = 1 \ldots n_b$. For this purpose we have implemented a number of *blocked* Krylov methods like GMRES, MINRES and BiCGStab. In contrast to the block Krylov methods found in Belos, these solvers build $n_b$ separate Krylov spaces, which reduces the effort for orthogonalization at the cost of some numerical efficiency. Compared to solving a sequence of $n_b$ linear systems with one right-hand side each, we still achieve better performance when applying operators and need fewer reductions. A non-standard algorithm we implemented is the CGMN method [Björck and Elfving 1979]. This algorithm can be seen as a CG-accelerated Kaczmarz method, and is particularly useful for matrices with small diagonal entries and highly indefinite matrices [Gordon and Gordon 2008]. A distributed memory variant called CARP-CG was developed by Gordon and Gordon [Gordon and Gordon 2010], and in [Galgon et al. 2015] we demonstrated its potential for solving linear systems arising when computing interior eigenvalues of some matrices using the FEAST method. We have not yet used it for preconditioning the Jacobi-Davidson method. Applications for which this method may be useful include the Helmholtz equations [Gordon and Gordon 2013].

### 2.6 Related libraries co-developed with PHIST

It is clear that the challenges of extreme-scale computing must be tackled by a collection of software packages that work well together. We have already mentioned the interoperability of PHIST with some other libraries like Trilinos. But there are also some developments that address complementary topics of extreme-scale computing. Two of these efforts have been integrated in PHIST, namely CRAFT (Checkpoint-Restart and Automatic Fault-Tolerance[8]) and SCAMAC (SCAlable MAtrix Collection[9]). The former provides an easy-to-use interface for making a program resilient to hardware faults, the latter allows the scalable and portable construction of benchmark problems for eigensolvers.

A third library that is important for the motivation of PHIST is called GHOST (General, Hybrid and Optimized Sparse Toolkit[10]). It provides highly optimized implementations of the kernel layer required by PHIST for Clusters of CPUs, GPUs and many-core processors. GHOST is to be seen as highly experimental, though, and relies completely on the PHIST test framework for correctness checks. While GHOST is not under discussion in this paper (for a reference see [Kreutzer et al. 2017]), we do include some performance results in Section 5. They show the achievement of PHIST to integrate experimental kernels for new hardware while at the same time maintaining software robustness.

Finally we want to mention the BEAST library (Beyond fEAST[11]), which is based on PHIST and implements several projection-based eigensolvers using contour integration, Chebyshev polynomials and moments.

GHOST and BEAST are available as independent but interoperable software packages. Some of these efforts are described in more detail in [Thies et al. 2016].

[8] https://bitbucket.org/essex/craft
[9] https://bitbucket.org/essex/MatrixCollection
[10] https://bitbucket.org/essex/ghost
[11] https://bitbucket.org/essex/beast/

## 3 ALGORITHM-LEVEL PERFORMANCE OPTIMIZATION USING PHIST

Typically, the performance of sparse matrix solvers is bounded by the memory bandwidth on modern HPC systems. This observation leads to some typical techniques for optimizing the performance of such methods, which must be supported by the lower levels of the software in order to keep the algorithm implementation simple and readable. If the amount of data is small compared to the memory bandwidth, the limiting factor may shift to some latency in the system, e.g. for communication between host and device or via the network, or for launching a kernel on the GPU. In this case reduction operations may become the bottleneck, which occur in inner products.

A central technique in PHIST is the use of block algorithms. For linear systems one can e.g. solve for multiple right-hand sides using a block Krylov method [Gutknecht 2006]. Similarly, eigenvalue solvers can often be straight-forwardly generalized to block variants which are applicable as soon as more than one eigenpair is sought, and may be numerically superior for finding tightly clustered or multiple eigenvalues. Besides this numerical benefit, block methods may be advantageous from a performance point of view [Gropp et al. 1999; Röhrig-Zöllner et al. 2015]. Typically when applying a sparse matrix to a vector, due to the indirect memory access pattern unneeded vector elements are loaded into the cache. When performing the operation on multiple vectors stored as a block (or multi-vector) in row-major order (that is, the elements of the different columns lie adjacent in memory for each row), this unnecessary and often erratic memory traffic can be reduced. Furthermore, BLAS1 operations like scalar products and vector scaling are replaced by slightly more compute intensive inner products with 'tall and skinny' matrices. Performing these operations by BLAS3 (GEMM) function calls though, typically leads to poor performance because implementations are optimized for the compute-bound case of roughly square matrices. PHIST offers a number of kernel functions for these operations, e.g. if $V, W$ are `mvecs` and $C$ is an `sdMat` with appropriate dimensions,

- `mvecT_times_mvec` computes $C \leftarrow \alpha V^T W + \beta C$,
- `mvec_times_sdMat` computes $V \leftarrow \alpha WC + \beta V$, and
- `mvec_times_sdMat_inplace` performs the operation $V_{:,1:k} \leftarrow V \cdot C$ if $V$ is $n \times m$ and $C$ is $m \times k, k \leq m$.

A mechanism to avoid data movements is the use of *views*. A view of an `mvec` in PHIST is a lightweight object that represents (a subset of) the columns of another `mvec`. In contrast to the Trilinos libraries Epetra and Tpetra, a view can only be created of a contiguous range of columns. The reason for this restriction is that we want to support multi-vectors stored in row-major order without too much implementation and performance overhead. A view of an `sdMat` can be created as well. Such an object represents a contiguous subset of rows and columns of the existing object. A view is fully equivalent to an actual object and can be passed to any of the PHIST functions. Deleting a view does not affect the original object, and all views of an object must be deleted before the object itself. From a performance point of view, one has to be careful because operations on one or a few columns of an `mvec` in row-major storage leads to strided data accesses and decreased performance.

A third technique in PHIST is *kernel fusion*. If two or more subsequent operations are performed that involve the same data structure, it may be possible to load it only once into the cache and perform all required operations. For example, the sequence

$$w = \mathrm{A}v$$
$$\alpha = v^T w$$

can be computed in a single loop, requiring only one vector to be loaded instead of up to three. The corresponding PHIST function is called (following a shortened naming convention) `fused_spmv_mvTmv`. There are a few functions like this,

with simple fallback implementations for kernel libraries that are not specifically optimized for PHIST. Note that there is no general mechanism for concatenating operations like this, In our experience with the CUDA implementations in GHOST, achieving reasonable performance on GPUs requires a dedicated effort for each operation implemented, taking into account the details of all memory movements.

The final mechanism we aim to support is thread-level concurrency, meaning that multiple kernels can be 'launched' before querying their results, and they can be executed concurrently on multi-threaded hardware. The GHOST kernel library offers a tasking mechanism for this, which takes care of handling thread affinity and works well together with OpenMP [Kreutzer et al. 2017]. Launching multiple tasks may in general lead to oversubscription of computational cores, and may interfere with the way the kernel library handles thread affinity. For this reason, we currently use the feature only with GHOST and only for the case of overlapping other computations with the reductions in dot products and the 'halo exchange' in sparse matrix-vector multiplication. We offer an interface to execute inner products and sparse matrix-vector products in several stages, so that one can start the operation, perform other work, and then finish the operation by waiting for the result. The code for overlapping the global reduction in a dot product with a vector 'AXPY' is shown in Listing 5. The general technique could be used to implement 'pipelined' Krylov methods and similar algorithms. For other kernel libraries, the macros result in in-order execution of the operations and thus numerically correct behavior. For kernel libraries using non-blocking MPI for these communication tasks, one could easily implement our interface in a non-blocking way.

```
1  // declare a task for the dot product
2  PHIST_TASK_DECLARE(dotTask);
3
4  // start a dot product s=x^Ty
5  PHIST_TASK_BEGIN(dotTask)
6  phist_Dmvec_dot_mvec)(x,y,&s,&iflag);
7  PHIST_TASK_END_NOWAIT
8
9  // wait for the local dot product computations
10 PHIST_TASK_WAIT_STEP(dotTask);
11
12 // perform some other operation v=v+alpha*w
13 phist_Dmvec_add_mvec(alpha,w,1.0,v,iflag);
14
15 // wait for the dot product reduction
16 PHIST_TASK_WAIT(dotTask);
```

Listing 5. Overlapping communication and computation using task macros

In the future we plan to implement the interface in a more general way so that it works with other kernel libraries than GHOST as well, e.g. using the C++ std::future concept.

### 3.1 Example: block orthogonalization

In various sparse iterative solvers an operation is needed which we call 'block orthogonalization'. Given orthonormal vectors $\left(w_1, \ldots, w_k\right) = W$ and a multi-vector $X \in \mathbb{R}^{n \times n_b}$, find orthonormal $Y \in \mathbb{R}^{n \times \tilde{n}_b}$ with

$$YR_1 = X - WR_2, \qquad \text{and} \qquad W^T Y = 0$$

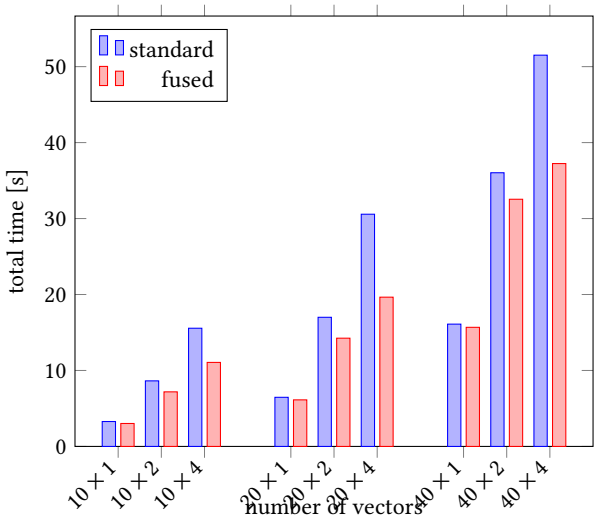


Fig. 2. Runtime reduction for block orthogonalization by kernel fusion. The label $M \times K$ means that $X \in \mathbb{R}^{N \times K}$ is orthogonalized against $W \in \mathbb{R}^{N \times M}$ for a fixed vector length $N = 8 \cdot 10^6$. The experiment was run on a 2-socket Intel Haswell EP CPU (E-2670 v.3) with 12 cores per socket.

This problem can be addressed using a two phase algorithm:

**Phase 1** Project: $\bar{X} \leftarrow (I - WW^T)X$

**Phase 2** Normalize: $Y \leftarrow f(\bar{X})$

Suitable choices for $f$ include SVQB [Stathopoulos and Wu 2002] or TSQR [Demmel et al. 2012]. As each phase may deteriorate the result of the other, one needs to iterate between the phases. It is therefore in our experience not necessary to use the highly accurate TSQR method, and we resort to SVQB, which has simpler performance characteristics. This results in a method of the form $f(\bar{X}) = \bar{X} \cdot g(M)$, where $M = \bar{X}^T\bar{X}$ is called the Gram matrix. The `sdMat` $g$ can be computed using a Cholesky factorization or an eigendecomposition (as in SVQB).

*Kernel fusion.* It is possible to reduce the amount of data traffic in the above iterative procedure by rearranging the operations:

**Phase 3'** $\bar{X} \leftarrow X \cdot g(M)$, $N \leftarrow W^T\bar{X}$

**Phase 1'** $\bar{X} \leftarrow X - WN$, $M \leftarrow \bar{X}^T\bar{X}$

**Phase 2'** $\bar{X} \leftarrow Xg(M)$, $\bar{M} \leftarrow \bar{X}^T\bar{X}$

In this algorithm, the bars above $X$ and $M$ are used only to make clear within a phase whether the 'old' or the 'updated' quantity is used, the next phase will always start with the updated quantity (e.g. $\bar{X}$ from the previous phase as $X$) because all operations are performed in-place. In order to start the iteration, $M$ must be computed once beforehand. The two operations in each phase can be performed using a fused kernel while the required data is available in the cache. The second computation of the Gramian $M$ in Phase 1' can be used to check a stopping criterion and is input for the next step (Phase 3') if needed, so that the overall number of reductions is the same as before. A brief performance study with this approach (using the builtin kernels) is shown in Figure 2.

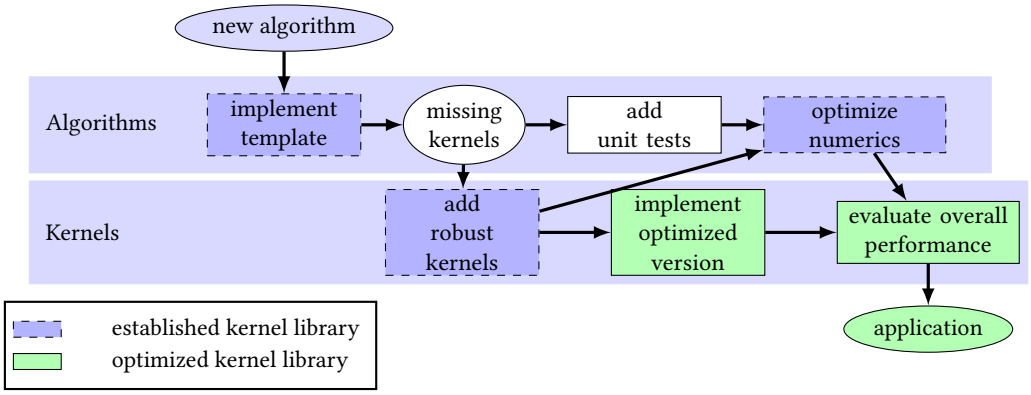


Fig. 3. The test-driven HPC development process

## 4 SOFTWARE AND PERFORMANCE ENGINEERING

It is our proclaimed goal to implement *holistic performance engineering* for sparse iterative solvers, an approach that yields an implementation with well understood and predictable overall hardware performance. We will show in Section 5 that this requires considering all three software layers (kernels, core and algorithms) together. Our software development methodology could be described as 'test- and benchmark-driven co-design' of the three layers. The workflow is depicted in Figure 3. Solvers can be implemented using an established kernel library in the first place. Any functionality useful for different algorithms is moved into the core layer.

Whenever a new kernel is required, tests and a performance model (see 'perfcheck' below) are added, and the operation is implemented using the established kernel library. This is typically easy because there may already be an implementation in one of the supported kernel libraries, or a 'quick-and-dirty' implementation suffices. At this point the optimization of the kernel for different hardware can start, using the performance model and tests to verify the code.

A crucial part of PHIST is the extensive test suite. It is based on the Google test framework[12], with some modifications to support MPI parallel programs (e.g. all assertions are globalized using a reduction). A fully optimized implementation of the kernels we need can take a very large number of code paths, for instance, GHOST calls different functions for a certain operation depending on data layout and alignment and available SIMD features of the CPU. It is therefore difficult to achieve full coverage on this level. We address this problem by trying to anticipate common bugs in the kernel library. Using macros like those described in Section 2.2, we generate from a single code tests for different block sizes, data types, with or without alignment, using both small and larger vector lengths.

The second tool for implementing our workflow is the so-called 'perfcheck' feature. By default, a timer is used to provide some information on where time is spent in a PHIST run. This can be replaced by information on how much of the performance predicted by an appropriate model is achieved by the various kernels. There are currently two simple types of models: either a kernel is 'small', meaning that it should not take a significant amount of time (e.g. operations on `sdMats`), or it is bounded by the memory bandwidth. In this case we select an appropriate benchmark with similar balance of loads and stores (a STREAM benchmark [McCalpin 2007]) and apply the roofline model [Williams et al. 2008]. The assumption that kernels involving sparse matrices and/or multi-vectors are memory bandwidth-bound holds true as long as the number of rows per process is large enough. For sparse matrix-vector products, an optimistic lower

[12] https://github.com/google/googletest

bound for the run time is calculated by assuming that each element of the input vector is loaded exactly once, which may not be true for irregular sparsity patterns (see [Kreutzer et al. 2014] for a more accurate roofline model).

*Example:* when running the Anasazi block Krylov-Schur solver with a block size of 4 (see Section 5), one gets a line like this for each kernel (with some additional columns to show the variation in the timing results, omitted here):

| function(dim) / (formula) | total time | %roofline | count |
|---|---|---|---|
| `phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,*iflag=0)`<br>`STREAM_TRIAD((nV+nW)*n*sizeof(_ST_))` | 6.156e+00 | 11.7 | 174 |

This operation is called 174 times, takes about 6 seconds in total and achieves only 12% of the predicted performance. One can activate in addition the 'realistic' option, meaning that strided memory accesses are taken into account and the model assumes that data is loaded by cache lines:

| function(dim) / (formula) | total time | %roofline | count |
|---|---|---|---|
| `phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,ldV=85,*iflag=0)`<br>`STREAM_TRIAD((nV_+nW_)*n*sizeof(_ST_))` | 6.013e+00 | 23.8 | 174 |

From the above output we can learn two things. On the one hand, many operations are performed using a large stride of $ldV = 85$. This happens when carelessly using views while the `mvecs` are stored in row-major order. To avoid large strides at least at the beginning, our Jacobi-Davidson implementation resizes the searchspace in a step-wise manner. For the blocked GMRES correction solver we use an array of `mvecs` to store the basis. The second observation is that even taking the stride into account, the achieved performance is only about 24%. The reason in this particular case may be that the problem size of $128^3$ is comparatively small and the kernel library is optimized for larger data sets (i.e. the memory-bounded case).

*Documentation.* Here PHIST uses three complementary techniques. The Doxygen software can be used to generate HTML documentation for all relevant functions and data structures from comments in the header files. We make an effort to group the documentation into useful modules, the top level of which correspond roughly to the software layers described above.

A second component of the documentation is a wiki that can be found at https://bitbucket.org/essex/phist/wiki/. It provides a more high-level view of the software, explains some of the underlying principles discussed in this paper and contains information on compilation and usage.

Finally, the software contains a number of *drivers* and *examples*, programs that can be run from the command line for assessing the performance of specific solvers and kernels, and which give an overview of how to use the different layers.

## 5 EIGENSOLVERS AVAILABLE IN PHIST

In this section we describe the implementation of two central solvers in PHIST: the block Krylov-Schur solver implemented in Anasazi, which we enhanced with our own orthogonalization scheme, and the block JDQR method. Some performance results using three different kernel libraries are presented to show the benefits of hand-optimized kernels for these methods. The experiments are performed on a multi-core CPU and a GPU:

- "Skylake": 4× of Intel Xeon Scalable "Skylake" Gold 6132, 2.60 GHz, 14-Core Socket 3647, 384GB DDR4 RAM.
- "Volta": NVidia Tesla V100-SXM2 GPU, 16GB HBM2 memory.

On the CPU, we use one MPI rank per socket, and one OpenMP thread per core for the PHIST builtin, GHOST and Tpetra kernel libraries, and one MPI rank per core for Epetra. In all cases, we fill all of the 56 available cores. In Table 1

(left) we measured the streaming memory bandwidth for the two architectures with some simple benchmarks (a load, a store dominated benchmark and a benchmark with two loads and one store). The right part of the table shows the performance achieved on the GPU in practice for the inner product of two `mvecs` (using the GHOST implementation). The fundamental problem when using GPUs in our context becomes apparent here: the main memory is extremely fast but small, and in order to even get away from the launch latency penalty one needs quite large data sets (i.e. long vectors). A similar benchmark on the Skylake CPU gives more than 90% roofline performance already for $N = 2M, n_b = 2$. We have not tackled this problem, as will be seen later on, but the measurements indicate that despite the high memory bandwidth we should not expect much performance gain here for practical algorithms because we are restricted to relatively small matrices.

| benchmark | Skylake | Volta |
|---|---|---|
| load | 360 | 812 |
| store | 200 | 883 |
| triad | 260 | 843 |

| $n_b$ | N=1M | 2M | 4M | 8M | 16M | 32M |
|---|---|---|---|---|---|---|
| 1 | 12 | 23 | 37 | 58 | 78 | 83 |
| 2 | 31 | 35 | 53 | 68 | 81 | 88 |
| 4 | 34 | 53 | 66 | 83 | 88 | 95 |
| 8 | 51 | 70 | 85 | 87 | 99 | 100 |

Table 1. Left: measurements of the streaming memory bandwidth (in Gb/s) on our test hardware. Right: Percentage of the memory bandwidth achieved by the operation $X^T Y$, $X$, $Y \in \mathbb{R}^{N \times n_b}$ on the Volta GPU. Note that the STREAM benchmarks are run with one billion elements, whereas the problem size on the right is a few million.

### 5.1 Block Krylov-Schur

This eigensolver is available in Anasazi and is essentially a block variant of the Arnoldi method which is restarted using a Schur decomposition [Stewart 2002]. In every iteration, a new block is generated by applying the operator and orthogonalizing the result against the current basis. Anasazi has several options for the block orthogonalization. We will compare the SVQB variant implemented in Anasazi with our own, as described in the previous section.

We consider a non-symmetric standard eigenproblem which stems from the discretization of a 3D PDE using a 7-point stencil[13] and $128^3$ grid points (the matrix is generated in PHIST using the string "BENCH3D-128-B1"). We request the 10 right-most eigenpairs to an accuracy of $10^{-6}$ and allow at most 80 vectors in the basis (10-80 blocks depending on the block size $n_b$). Block orthogonalization is performed here using our own implementation. The reason why we use such a relatively small matrix is that we want to run these experiments also on the GPU, which has very limited memory. In Figure 4 we compare the overall runtime for different kernel libraries and block sizes on the Skylake CPU. At a first glance, the GHOST library performs slightly worse than our own kernels, which can probably be explained by additional overheads in that library for performing relatively small computations (the problem size is so small here that a single vector fits into the cache of the four combined CPU sockets, and GHOST uses dynamic dispatching to automatically generated kernels, which introduces a small constant overhead). Experience shows that GHOST outperforms the builtin PHIST kernels typically when the matrix structure is irregular (due to the more sophisticated SELL-$C-\sigma$ format [Kreutzer et al. 2014]). If the run-time is dominated by block vector operations, the overall performance of both implementations is similar, with variationns in single kernels because not all variants are implemented in both libraries. The builtin kernels achieve a speed-up of 2-3 over Tpetra here. This is most likely caused by missing fused and in-place kernels in Tpetra, so that the block orthogonalization routine needs to allocate temporary vectors. As Tpetra performs

[13] a 3D variant of the matpde generator available at
`https://math.nist.gov/MatrixMarket/data/NEP/matpde/matpde.html`

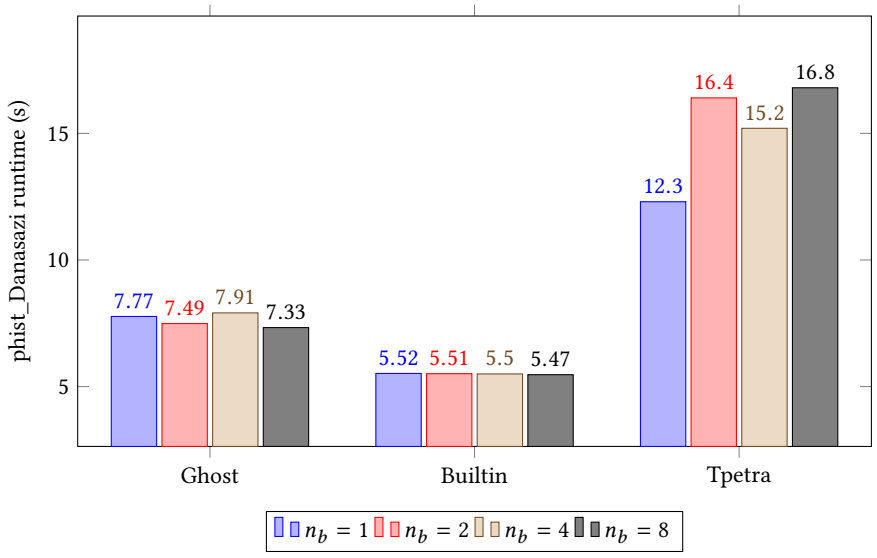


Fig. 4. CPU runtime of Block Krylov-Schur method with different block sizes and kernel libraries.

'first touch' allocation, this leads to significant additional memory traffic. Another point is that the spMVM is faster using row-major storage, as in GHOST and the builtin kernels.

A more thorough analysis using PHIST's perfcheck tool (see Section 4) reveals that with the builtin kernels, more than half of the runtime is spent in operations with a stride of 85. Anasazi was not implemented with row-major `mvecs` in mind, and a significant speed-up could be achieved by rewriting its algorithms to avoid views leading to large strides.

In Figures 5 and 6 we look at the effect of replacing the orthogonalization scheme in Anasazi with our own implementation, on Skylake and Volta, respectively. Both schemes use iterated CGS/SVQB, but obviously implemented differently. The combination of row-major storage (Builtin/GHOST) with our block orthogonalization gives the fastest result, but the Tpetra implementation (with col-major storage) can also benefit, at least on the CPU. On the GPU, the temporary memory allocations (see above) lead to too much overhead. Furthermore, we see the anticipated result that for such a small problem size, the higher memory bandwidth of the GPU does not translate into an actual speed-up.

### 5.2 Block Jacobi-Davidson QR eigensolver

The primary solver in PHIST is a block Jacobi-Davidson method for computing some eigenpairs of large, sparse, Hermitian or non-Hermitian matrix pencils $[A, B]$, where B should be Hermitian and positive definite. Details of the algorithm and implementation can be found in [Röhrig-Zöllner et al. 2015], but we will summarize the main lines here as well and add some extensions that have been made in the mean time.

Let $\mathbb{B}$ denote the set of real or complex numbers. Given $A, B \in \mathbb{B}^{N\times N}$ and an integer $k \ll N$, the method tries to compute $(Q, R)$, $Q \in \mathbb{B}^{N\times k}$, $R \in \mathbb{B}^{k\times k}$ such that $AQ = BQR$, $Q^H BQ = I$, and $R$ a Schur form (block upper triangular with complex pairs of eigenvalues represented by a $2\times 2$ block on the diagonal in the real case). The user can specify whether to compute the smallest ($S$) or largest ($L$) eigenvalues in terms of their real part ($R$) or magnitude ($M$). The primary aim of the implementation is to compute eigenpairs at the lower or upper end of the spectrum ('SR' or 'LR'). The approximate eigenvalues can then be found on the diagonal of $R$. We use standard Ritz values to approximate the eigenvalues, for

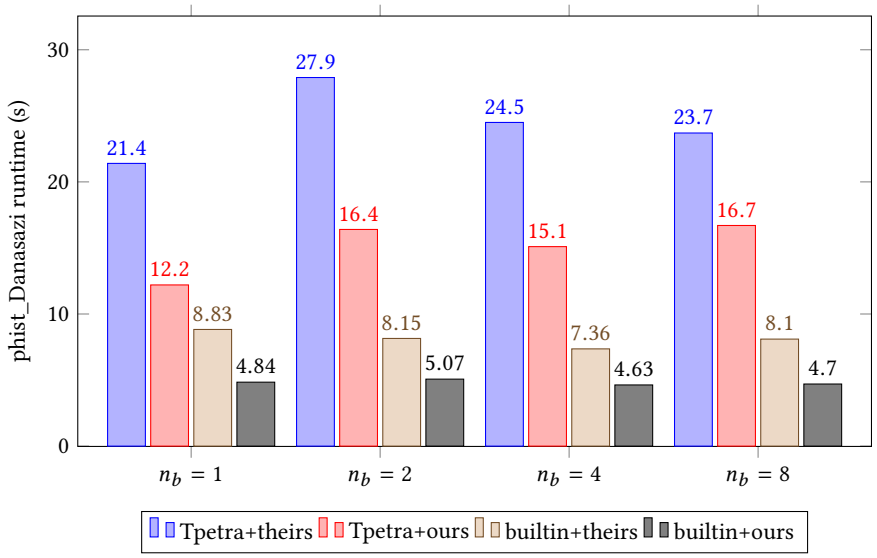


Fig. 5. Runtime of the block Krylov Schur solver on the Skylake CPU with the Anasazi ('theirs') and PHIST ('ours') implementations of SVQB.

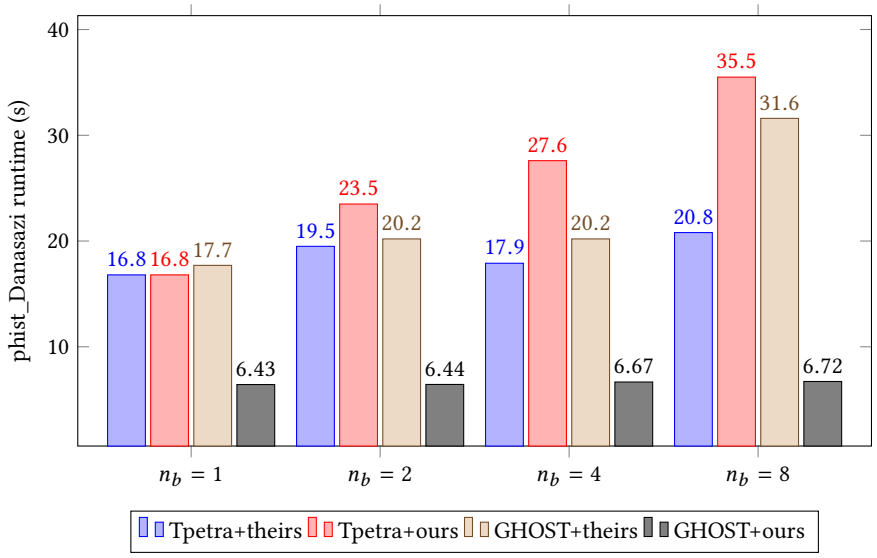


Fig. 6. Runtime of the block Krylov Schur solver on the Volta GPU with the Anasazi ('theirs') and PHIST ('ours') implementations of SVQB.

computing interior ones harmonic Ritz values may be more appropriate (see also the review article [Hochstenbach and Notay 2006] on Jacobi-Davidson methods).

Our implementation takes as parameters the minimum and maximum basis sizes $m_{min}$ and $m_{max}$, and the block size $n_b$. It starts by building a basis of size $m_{min}$ using the Arnoldi method, and then extends this basis by $n_b$ vectors

per iteration. If the basis exceeds $m_{max}$ vectors, a restart is performed by extracting the 'best' $m_{min}$ directions from the larger subspace. The directions to be added are determined by solving the $n_b$ independent *correction equations*

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

where $\tilde{q}_j$ are the current approximations, $\Delta q_j$ the desired (Newton) corrections, and $\tilde{Q} = [Q\ \tilde{q}]$ contains the already converged ('locked') eigenbasis $Q$ and the current approximations. We can write the equation in a more abstract form as

$$\text{precOp} \cdot \text{jdOp} \cdot \Delta q_j = -\text{precOp} \cdot (A\tilde{q}_i - \tilde{Q}\tilde{r}_j), j = 1 \ldots n_b, \tag{2}$$

and solve it approximately using some iterations of a Krylov subspace method. The default choices in PHIST are GMRES and MINRES for general and Hermitian problems, respectively. In the simplest case of a standard eigenvalue problem without preconditioning (discussed in [Röhrig-Zöllner et al. 2015]), precOp $= I$ and jdOp $= (I - \tilde{Q}\tilde{Q}^H)(A - \sigma_i I)(I - \tilde{Q}\tilde{Q}^H)$, where the rightmost (pre-)projection can be omitted in practice.

For generalized eigenvalue problems, the preprojection operator cannot be omitted and we obtain

$$\text{jdOp}_\text{B} = (I - (B\tilde{Q})\tilde{Q}^H)(A - \sigma_i B)(I - \tilde{Q}(B\tilde{Q})^H).$$

Left) preconditioning is implemented by following the jdOp operator by

$$\text{precOp}_\text{B} = (I - (K^{-1}V)((BV)^H K^{-1} V)^{-1}(BV)^H)K^{-1}.$$

We note that different choices are possible. Many variants for the Hermitian standard problem are implemented in PRIMME [Stathopoulos and McCombs 2010], but for non-Hermitian and generalized problems one has to be more careful. In our implementation we decided to preselect the variants above to achieve robust behavior for many practical problems, but in the future we may expose more options to the users if the need in applications arises.

*Benchmark results.* We will evaluate the performance of the `subspacejada` solver for two different matrices, the first is a larger variant of the previous 3D example, where we now compute the left-most eigenvalues (near 0) instead (for Krylov-Schur this would require a shift-invert technique). The second is the original 2D matpde benchmark on a $2048^2$ grid, where we look for the right-most eigenvalues. This leads to slightly longer vectors and less memory consumption for storing the matrix, making it hopefully more suitable for the Volta GPU. To save memory we compute in both cases only 10 eigenpairs and allow at most 40 vectors in the basis (restarting from 20 when needed). The tolerance is $10^{-6}$ as before.

Figure 7 shows some overall timing results for the $256^3$ problem on the Skylake CPU. We see that for large enough problem sizes our optimization techniques (kernel fusion, row major storage etc.) start to pay off and the PHIST builtin kernels are faster than the pure MPI variant in Epetra. Increasing the block size $n_b$ beyond 2 does not pay off when searching for only a few eigenvalues, as was also reported in [Röhrig-Zöllner et al. 2015]. Table 2 shows the results for the 2D problem on the two architectures and using different kernel libraries. The first observation is that for the single-CPU configuration with a relatively small matrix, Epetra (MPI only) performs remarkably well, and the Tpetra (OpenMP) implementation is remarkably slow (possibly due to 'first touch' and temporary objects, see above). With the GHOST CUDA kernels one can match the CPU performance, which agrees with our previous experience for small problem sizes.

The overall number of iterations and runtime for computing the smallest eigenvalues of the BENCH3D matrix is quite high. The convergence is dominated by the Laplace-like component of the equations, and using a preconditioner is

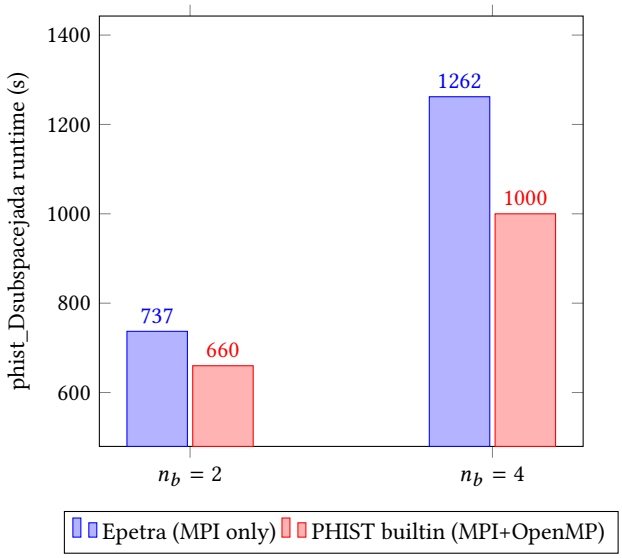


Fig. 7. Runtime of the Jacobi-Davidson solver for computing the smallest 10 eigenpairs of the BENCH3D-256-B1 matrix on the Skylake CPU.

| kernel lib | $t_{tot}[s]$ |
|---|---|
| Tpetra | 80.28 |
| Epetra | 7.21 |
| Builtin | 9.11 |
| Ghost (Volta) | 7.09 |
| Tpetra (Volta) | 46 |

Table 2. Timing results for the matpde2048 problem using subspacejada. The first three cases are run on the Skylake CPU, the last two on the GPU.

the method of choice to alleviate this. Table 3 shows the effect of using the AMG preconditioner ML in addition to doing some inner GMRES iterations. For configuring the ML preconditioner we use the default settings for 'non-symmetric smoothed aggregation' (NSSA) as implemented in Trilinos 12.12.1.

| problem size | preconditioner | iterations | spMVMs | $t_{tot}$ | $t_{gmres}$ |
|---|---|---|---|---|---|
| $128^3$ | GMRES | 471 | 10 403 | 38.5 | 24.7 |
| | GMRES+ML | 31 | 720 | 26.3 | 13.2 |
| $256^3$ | GMRES | 815 | 17 971 | 736 | 496 |
| | GMRES+ML | 29 | 668 | 227 | 116 |

Table 3. Effect of using the AMG solver ML to precondition the PHIST Jacobi-Davidson method (block size 2) for the non-symmetric problem 'B1'.

The number of iterations is reduced to a small constant, as one would hope with a multigrid method. The timing results indicate that there is quite some room for optimization. Our implementation applies the preconditioner repeatedly

to the projection space, for instance. With row-major storage of `mvecs` the preconditioning operation itself would also become faster.

## 6 SCALABILITY BEYOND ONE NODE

So far we have focused on the performance on a single node. This case can be investigated thoroughly using performance models, and we give it special attention because nowadays the performance increase of HPC systems comes almost exclusively from increased parallelism on the nodes. In this section we will show some results of large-scale benchmarks for the block Jacobi-Davidson QR method. The eigenproblems are scaled-up versions of the 7-point Laplace problem ('A0') and the non-symmetric 'B1' PDE problem used before. As the convergence for these matrices (without additional preconditioning) depends strongly on the grid size, we fix the number of outer JDQR iterations to 240 in order to compare runs for different matrix sizes, and report the performance in TFlop/s. The correction equations are solved approximately by 10 steps of MINRES or GMRES, respectively, where in the GMRES algorithm we use a robust iterated modified Gram-Schmidt (IMGS) orthogonalization, which requires a relatively large number of global reductions. The matrix is partitioned linearly after sorting the indices using an octree algorithm (also known as the Morton space-filling curve).

The machine we use is the Oakforest-PACS supercomputer (OFP) at the Japanese joint center for advanced high-performance computing (JCAHPC)[14]. It consists of 8 208 nodes of Intel Xeon Phi 7250 processors with 68 cores each, and an Intel Omnipath interconnect. OFP achieved a performance of about 385 TFlop/s in the HPCG benchmark, which is the appropriate measure for our memory bounded sparse matrix algorithms. We chose the GHOST kernel library because it has dedicated AVX512 code, and compiled everything using the Intel compiler and Intel MPI (version 2018.1.163). Benchmark sizes are chosen to fit in the 16GB high bandwidth memory on each node.

Figure 8 shows the weak scaling for the non-symmetric (B1) and symmetric (A0) case. The largest problem size is $N = 4\,096^3$ on 8 192 and 4 096 nodes, respectively. The percentage in the figure shows the parallel efficiency compared to the first measurement, e.g. if $P_k$ is the performnce on $k$ nodes, $\frac{k_{min} \cdot P_k}{k \cdot P_{k_{min}}} \cdot 100\%$ is shown. We observe that in both cases we achieve a parallel efficiency of more than 50%, but the overall performance of 148 TFlop/s is clearly below the HPCG value, despite the block size of 4 used. This can be explained by the fact that we do not exploit the stencil structure of the test matrix, whereas the HPCG code is optimized to solve this particular problem. The average roofline performance achieved over the run with block size $n_b = 4$ is estimated by PHIST between 21% (16 nodes) and 12% (8 192 nodes) for the symmetric problem, and between 27% (8 nodes) and 17% (4 096 nodes) for the non-symmetric problem, which requires significantly more global reductions due to the IMGS scheme in GMRES. The detailed profiling output shows the expected behavior that the relative cost of vector updates and other trivially parallel kernels decreases at scale compared to sparse matrix-vector multiplication (spMVM). Kernels involving an 'allreduce' consume about 3.3 times as much of the runtime as the spMVM for B1 on 4 096 nodes. These results indicate that – at least for the case of weak scaling – the price for the orthogonalizations in BJDQR is tolerable.

Fig. 9 shows the performance achieved for the symmetric problem when keeping the problem size fixed to $N = 1\,024^3$ and increasing the number of nodes. The right-hand figure shows the block speed-up defined by $\frac{P_{n_b=k}}{P_{n_b=1}}$. This shows that the strong scaling behavior of BJDQR can indeed improved by using larger block sizes (by collecting scalar products into a single reduction). However, one has to keep in mind that this also increases the number of iterations to convergence.

[14] http://jcahpc.jp/ofp/ofp_intro.html

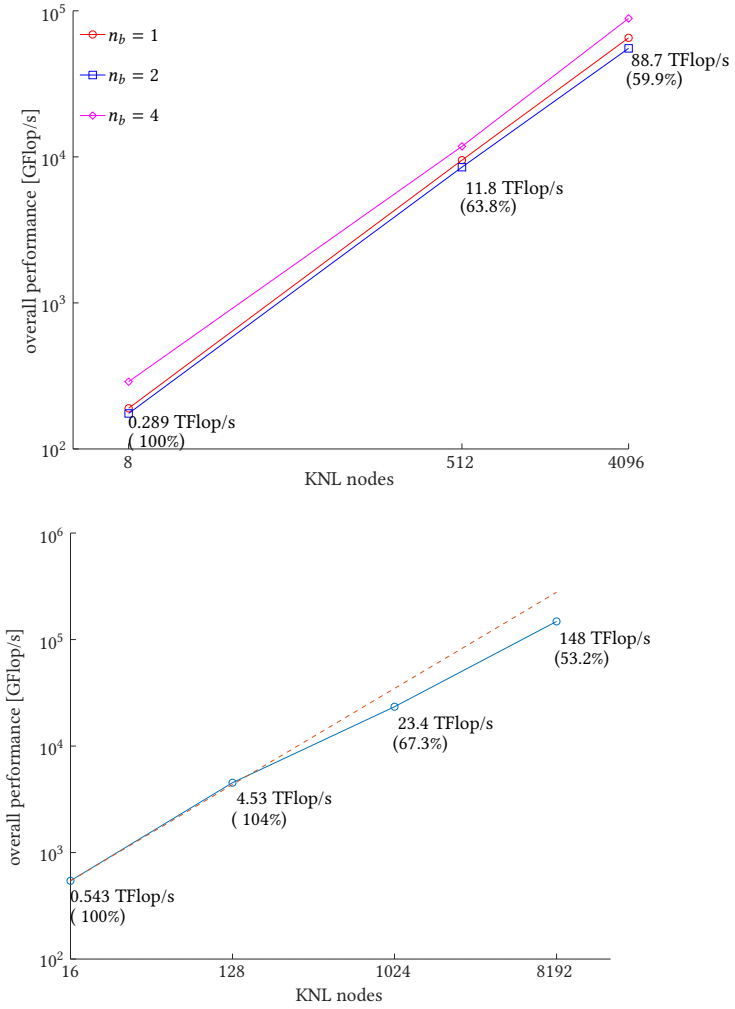


Fig. 8. Weak scaling results for BJDQR. Top: non-symmetric 7-point PDE matrix using GMRES+IMGS and approx. 16.7 million unknowns/node (different block sizes). Bottom: symmetric 7-point Laplace problem using MINRES as correction solver and approx. 8.4 million unknowns/node (block size 4).

## 7 SUMMARY AND FUTURE WORK

We hope to have shown in this paper how a combination of classical software engineering and HPC performance engineering allows to make reliable statements about the performance of sparse iterative solvers. The PHIST software offers a framework for implementing new algorithmic ideas in a portable way, putting emphasis on different aspects (e.g. performance, supported hardware or data types) by choosing an appropriate back-end. It allows developers of algorithms and applications to stay independent of a concrete implementation for a long time, making a thorough performance assessment before deciding on a back-end to run their simulation.

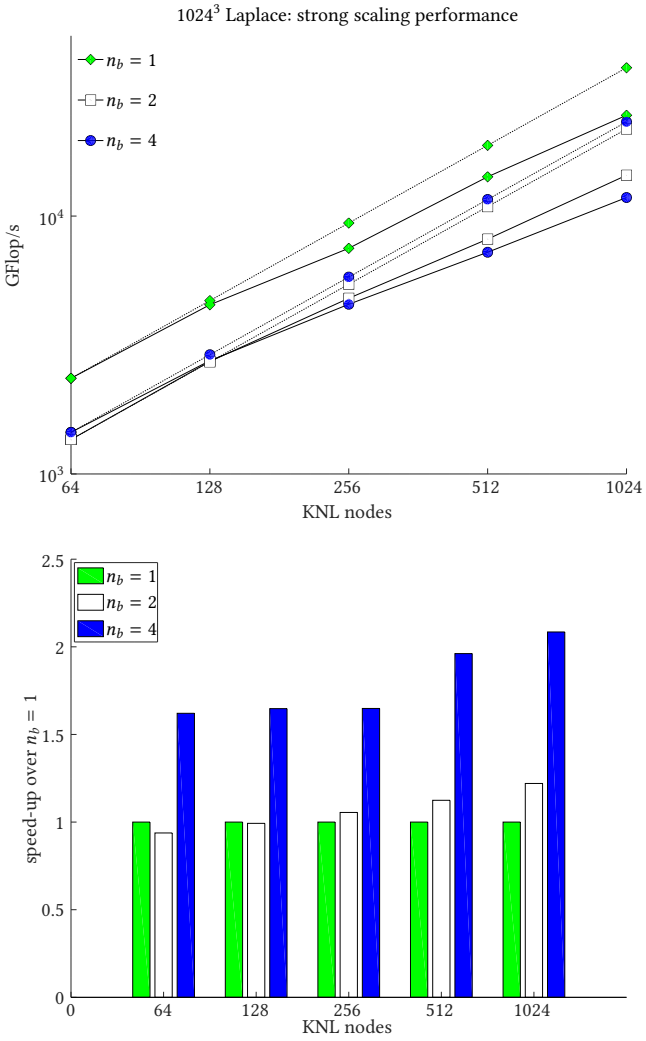


Fig. 9. strong scaling (top) and corresponding 'block speed-up' (bottom) for the symmetric $N = 1\,024^3$ problem.

We discussed and demonstrated various performance optimizations for iterative methods, from multi-vectors in row-major storage to kernel fusion. The algorithms we presented (block orthogonalization, Krylov-Schur and Jacobi-Davidson) are to be understood as blueprints for implementing other methods.

The Krylov-Schur implementation in Anasazi is currently geared towards column-major storage. We have made a first step towards higher block performance by introducing our orthogonalization scheme, but the use of views in larger blocks still costs quite some performance.

On GPUs we showed that good performance can be achieved with GHOST as long as the data sets are large enough. The comparatively small GPU memory is a problem here and we will investigate the use of unified virtual memory

(UVM) to be able to solve larger problems. Another idea is to reduce the memory footprint of algorithms, for Jacobi-Davidson this could be achieved e.g. by computing $V^T AV$ on-the-fly rather than storing $AV$. Another idea is to store `mvecs` in single precision (but perform calculations in double for robustness) if the required accuracy is not too high. For concrete applications, matrix-free operators and preconditioners may be used.

The builtin PHIST kernels include an experimental 'high precision' feature that has not been discussed in this paper because fast kernels are only available for certain block sizes so far, and we plan to assess its usefulness in concrete case studies first. The feature allows storing small and dense matrices in quadruple precision and using more accurate reductions to e.g. increase the effect of an orthogonalization step.

We demonstrated weak and strong scaling efficiency of BJDQR on a Peta-scale machine, and the advantage of the block variant as it reduces the number of global reductions. These results are, however, to be taken with some caution as the machine's behavior at large scale is much more difficult to understand than on the node level. Performance modelling of large distributed memory systems remains a topic for future work.

Many-node Performance could be further improved by reducing the number of reductions, or hiding them behind useful computation. However, this would likely infringe the numerical robustness, especially for non-Hermitian eigenvalue problems. We therefore focus on the approach to reduce the number of iterations by using good preconditioning techniques. We showed by an example how a multigrid preconditioner could reduce the total number of matrix-vector products by a large factor, but in our implementation there is quite some room for improvement in terms of runtime. We believe that the development of efficient preconditioners for eigenvalue problems (especially when looking for interior eigenvalues) is a major challenge in numerical linear algebra to date and offers interesting opportunities both on the mathematical and computational side. Future work in PHIST will address this challenge.

## ACKNOWLEDGMENTS.

This work was funded by the German Research Council (DFG) under priority program 1648 (SPPEXA, "Software for Exa-Scale"), project ESSEX. The computational resource of the Oakforest-PACS system was awarded by the "Large-scale HPC Challenge" project, JCAHPC (Joint Center for Advanced High Performance Computing).

We would like to thank Rebekka-Sarah Hennig (University of Bonn), who worked on the code and documentation as a student assistant at DLR.

## A NOTES ON REPRODUCING THE EXPERIMENTS IN THIS PAPER

It is our goal to make high performance available via PHIST, but unfortunately the complexity of the entire software stack may still make it difficult for readers to reproduce similar performance results. As an effort towards reproducibility we here specify the versions of software used, and the process of installing and running PHIST used in the paper.

- Spack (https://github.com/spack/spack), branch develop at commit `c1e3e5de5c`)
- OpenMPI and Trilinos installation without CUDA:

```
> spack find -v -d trilinos

-- linux-ubuntu16.04-x86_64 / gcc@7.3.0 -------------------------
    trilinos@12.12.1~alloptpkgs+amesos+amesos2+anasazi+aztec+belos~boost build_type=Release
    ~cgns~dtk+epetra+epetraext+exodus+fortran~fortrilinos+gtest+hdf5~hypre+ifpack+ifpack2
    +instantiate+instantiate_cmplx~intrepid~intrepid2+metis+ml+muelu~mumps+nox+openmp~pnetcdf
    ~python~rol+sacado~shards+shared~stk+suite-sparse~superlu~superlu-dist+teuchos+tpetra~x11
    ~xsdkflags~zlib+zoltan+zoltan2
        ^glm@0.9.7.1 build_type=Release
        ^hdf5@1.10.1~cxx~debug~fortran+hl+mpi+pic+shared~szip~threadsafe
            ^openmpi@3.0.1~cuda fabrics=verbs ~java~memchecker+pmi schedulers=slurm
```

```
        ~sqlite3+thread_multiple~ucx+vt
            ^hwloc@1.11.9~cairo~cuda+libxml2+pci+shared
                ^libpciaccess@0.13.5
                ^libxml2@2.9.4~python
                    ^xz@5.2.3
                    ^zlib@1.2.11+optimize+pic+shared
                ^numactl@2.0.11
    ^intel-mkl@2018.1.163~ilp64+shared threads=none
    ^matio@1.5.9+hdf5+shared+zlib
    ^metis@5.1.0 build_type=Release ~gdb~int64 patches=[omitted] ~real64+shared
    ^netcdf@4.4.1.1~dap~hdf4 maxdims=1024 maxvars=8192 +mpi~parallel-netcdf+shared
    ^parmetis@4.0.3 build_type=Release ~gdb patches=[omitted] +shared
    ^suite-sparse@5.2.0~cuda~openmp+pic~tbb
```

- OpenMPI installation with CUDA:

```
> spack find -v -d openmpi+cuda

-- linux-ubuntu16.04-x86_64 / gcc@5.4.0 -------------------------
    openmpi@3.0.1+cuda fabrics=verbs ~java~memchecker+pmi schedulers=slurm
    ~sqlite3+thread_multiple~ucx+vt
        ^hwloc@1.11.9~cairo+cuda+libxml2+pci+shared
            ^cuda@9.0.176
            ^libpciaccess@0.13.5
            ^libxml2@2.9.4~python
                ^xz@5.2.3
                ^zlib@1.2.11+optimize+pic+shared
            ^numactl@2.0.11
```

- Trilinos (master branch at commit `52db64a86f`) with CUDA: see `phist/buildScripts/build-trilinos-gpu.sh` included in PHIST 1.6.x (note that we used our own adaptation of the `nvcc_wrapper` script, which is also included in phist).
- PHIST v1.6.1 with Tpetra and CUDA: `phist/buildScripts/script_sc-hpc_tpetra_cuda.sh` in PHIST 1.6.x.
- GHOST (devel branch at commit `a3b75fc52c7ea`)
- Scripts for running the examples are found in `phist/exampleRuns/solvers/` in PHIST 1.6.x.

Finally we want to mention two particular performance hazards the user should be aware of when trying to achieve good performance with PHIST. The first is that a *sequential* BLAS library should be used, e.g. the reference implementation on `http://www.netlib.org/` or Intel MKL with the appropriate `sequential` flag (PHIST CMake option `-DBLA_VENDOR="Intel10_64lp_seq"`). Second, the binding of processes and threads to physical cores is very important on NUMA systems like our Skylake node. OpenMPI in the current version binds processes to cores by default. PHIST also tries to bind processes and threads to cores, and the two do not necessarily work well together. You can either disable the PHIST feature using the CMake flag `-DPHIST_TRY_TO_PIN_THREADS=OFF` and attempt to bind the threads using e.g. OpenMP environment variables, or choose the appropriate flags for mpirun (we use `-np 4 --bind-to numa` with builtin, Tpetra and ghost, and `-np 56 --bind-to core` with Epetra. Here 4 is the number of available CPU sockets and 56 the total number of CPU cores on the node. Using `-np 1 --bind-to none` gave a performance degradation in the eigensolver runs of about 10% in our experiments. We assume that here irregular accesses to other NUMA domains during the spMVM are more expensive than copying large chunks into MPI communication buffers.

## REFERENCES


Baker, C. G. and Heroux, M. A. 2012. Tpetra, and the use of generic programming in scientific computing. *Sci. Program. 20,* 2 (Apr.), 115–128.

Baker, C. G., Hetmaniuk, U. L., Lehoucq, R. B., and Thornquist, H. K. 2009. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Trans. Math. Softw. 36,* 3 (July), 13:1–13:23.

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., and Zhang, H. 2016. PETSc Web page.

Björck, Å. and Elfving, T. 1979. Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations. *BIT 19,* 2, 145–163.

Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comp. 34,* 1 (Jan), A206–A239.

Galgon, M., Krämer, L., Thies, J., Basermann, A., and Lang, B. 2015. On the parallel iterative solution of linear systems arising in the FEAST algorithm for computing inner eigenvalues. *J. Par. Comp. 49*, 153–163.

Ghysels, P., Ashby, T. J., Meerbergen, K., and Vanroose, W. 2013. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM J. Sci. Comp. 35,* 1, C48–C71.

Gordon, D. and Gordon, R. 2008. CGMN revisited: Robust and efficient solution of stiff linear systems derived from elliptic partial differential equations. *ACM Trans. Math. Softw. 35,* 3, 18:1–18:27.

Gordon, D. and Gordon, R. 2010. CARP-CG: A robust and efficient parallel solver for linear systems, applied to strongly convection dominated PDEs. *Parallel Comput. 36,* 9, 495–515.

Gordon, D. and Gordon, R. 2013. Robust and highly scalable parallel solution of the Helmholtz equation with large wave numbers. *J. Comput. Appl. Math. 237,* 1, 182–196.

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

Gutknecht, M. H. 2006. Block krylov space methods for linear systems with multiple right-hand sides: An introduction.

Hernandez, V., Roman, J. E., and Vidal, V. 2005. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software 31,* 3, 351–362.

Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A. 2003. An overview of Trilinos. Tech. Rep. SAND2003-2927, Sandia National Laboratories.

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., and Stanley, K. S. 2005. An overview of the Trilinos project. *ACM Trans. Math. Softw. 31,* 3, 397–423.

Hochstenbach, M. E. and Notay, Y. 2006. The Jacobi-Davidson method. *GAMM-Mitteilungen 29,* 2, 368–382.

Hoemmen, M. 2010. Communication-avoiding Krylov subspace methods. Ph.D. thesis, University of California, Berkeley.

Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM J. Sci. Comp. 36,* 5, C401–C423.

Kreutzer, M., Thies, J., Röhrig-Zöllner, M., Pieper, A., Shahzad, F., Galgon, M., Basermann, A., Fehske, H., Hager, G., and Wellein, G. 2017. GHOST: building blocks for high performance sparse linear algebra on heterogeneous systems. *Int. J. Parallel Program. 45,* 5 (Oct.), 1046–1072.

Lehoucq, R., Sorensen, D., and Yang, C. 1998. *ARPACK Users' Guide.* Society for Industrial and Applied Mathematics.

McCalpin, J. D. 1991-2007. STREAM: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, VA. A continually updated technical report.

Mohiyuddin, M., Hoemmen, M., Demmel, J., and Yelick, K. 2009. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* SC '09. ACM, New York, NY, USA, 36:1–36:12.

Röhrig-Zöllner, M., Thies, J., Kreutzer, M., Alvermann, A., Pieper, A., Basermann, A., Hager, G., Wellein, G., and Fehske, H. 2015. Increasing the performance of the Jacobi–Davidson method by blocking. *SIAM Journal on Scientific Computing 37,* 6, C697–C722.

Stathopoulos, A. and McCombs, J. R. 2010. PRIMME: preconditioned iterative multimethod eigensolver–methods and software description. *ACM Trans. Math. Softw. 37,* 2 (Apr), 1–30.

Stathopoulos, A. and Wu, K. 2002. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comp. 23,* 6, 2165–2182.

Stewart, G. W. 2002. A Krylov–Schur algorithm for large eigenproblems. *SIAM Journal on Matrix Analysis and Applications 23,* 3, 601–614.

Thies, J., Galgon, M., Shahzad, F., Alvermann, A., Kreutzer, M., Pieper, A., Röhrig-Zöllner, M., Basermann, A., Fehske, H., Hager, G., Lang, B., and Wellein, G. 2016. Towards an exascale enabled sparse solver repository. In *Software for Exascale Computing - SPPEXA 2013-2015*, N. W. Bungartz H.-J., Neumann P., Ed. Vol. 113. Springer.

Weisse, A., Wellein, G., Alvermann, A., and Fehske, H. 2006. The kernel polynomial method. *Rev. Mod. Phys. 78*, 275–306.

Williams, S. W., Waterman, A., and Patterson, D. A. 2008. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Tech. Rep. UCB/EECS-2008-134, EECS Department, University of California, Berkeley. Oct.