PHIST: a Pipelined, Hybrid-parallel Iterative Solver Toolkit

Melven Röhrig-Zöllner and Jonas Thies

German Aerospace Center (DLR)
Simulation and Software Technology
Melven.Roehrig.Zoellner@DLR.de
Equipping Sparse Solvers for the EXa-scale
Sparse Eigenvalue Problems

**Formulation** Find some Eigenpairs \((\lambda_j, v_j)\) of a large and sparse matrix (pair) in a target region of the spectrum

\[
A v_j = \lambda_j B v_j
\]

- **A** Hermitian or general, real or complex
- **B** may be identity matrix (or not)
- ‘some’ may mean ‘quite a few’, 100-1000 or so

**Applications**

- Quantum and Fluid Mechanics
- Graphene
- Hubbard model
- Driven cavity
- Rayleigh-Benard convection
- DLR applications
ESSEX targets various sparse eigensolvers

- **Subspace methods:** Krylov and Jacobi-Davidson
- Polynomial Eigensolvers (KPM, ChebFD, ChebTP)
- Contour integration methods: FEAST, Sakurai-Sugiura
- (block/blocked) **Krylov** subspace linear solvers
- **Preconditioning:** IC/ILU variants + support TPLs

**Productivity of algorithm developers** is crucial to the project
ESSEX: Other project activities

- Implementation of Quantum Physics applications
- Algorithm research (e.g. adaptivity in the FEAST algorithm)
- Kernel optimization for heterogenous multi-/manycore/GPU systems using OpenMP, SIMD and CUDA
- Dealing with Hardware failures
- Research into mixed precision, fault tolerance and performance modelling

PHIST aims to distill these efforts into portable, usable and efficient software.
Our Mission

... in general is to

provide a software framework
for implementing sparse linear and eigenvalue solvers
that run efficiently on present and future supercomputers

... and in particular to

implement Jacobi-Davidson type eigensolvers, and
make ESSEX technology available to algorithm developers and end users
Principles of PHIST

- Useful abstraction
- interoperability
- ‘pipelining’ as algorithm-level performance optimization
- holistic performance engineering
- pragmatic programming approach
  - C skin with ‘whatever’ underneath
  - C++, Python and F’03 bindings are available
  - MPI assumed for distributed memory
- correctness tests
Software architecture

Stand-alone implementation of the ‘Exa-scale Sparse Solver Repo’ (ESSR) that can be enhanced by the other ESSEX toolkits and third-party libs
Useful Abstraction: Kernel Interface

Choose from several ‘backends’ at compile time, to

- easily use PHIST in existing applications
- perform the same run with different kernel libraries
- compare numerical accuracy and performance
- use stable kernel libraries to verify correctness/feasibility of tests
- exploit unique features of a kernel library (e.g. preconditioners)
Kernel Interface (2)

Three linear algebra objects (passed around as void*):
- `sparseMat`, distributed sparse matrix in arbitrary storage format
- `mvec`, multi- (or block-)vector, stored either row or column major (determined at compile time)
- `sdMat`, small dense matrix local to the MPI process

A **view** of certain contiguous columns (rows and cols) of an `mvec` (`sdMat`) can be used exactly the same way as the original object.

**Macros** to
- handle return codes, e.g. `PHIST_ICHK_IERR`
- allow type-generic programming, e.g `SUBR(foo) ⇒ phist_[D/Z/...]foo, _ST_ ⇒ double/complex<double>/...`
- expose cool ESSEX features (e.g. tasking, perfcheck, fault tolerance)
Kernel Interface (3)

Available functions:

- create/view/delete/fill objects
- standard mathematical operations, e.g.
  - $Y \leftarrow \alpha AX + \beta Y$
  - $S \leftarrow \alpha X^H Y + \beta S$
- non-standard kernels, e.g. $Y_{:,1:k} \leftarrow Y_{:,1:m} \cdot S$, $S$ $m \times k$, $k \leq m$
- ‘fused’ kernels with more than one output, e.g. $Y \leftarrow AX$, $S \leftarrow X^H Y$, $T \leftarrow Y^H Y$

For the latter two categories, (suboptimal) default implementations exist if the kernel library does not support some operations.
Pipelining as algorithm-level performance optimization

Idea: algorithms must be reformulated to

- exploit SIMD operations
- reduce the number of global reductions
- hide the remaining ones behind computation

‘Pipelining’ is a broad term here:

- multiple Eigenpairs are improved at once (subspacejada)
- pipelined solution of many linear systems (BEAST-C)
- hide reductions behind spMVM in Krylov solvers (e.g. pipelined GMRES)
- pipelining of basic kernels allows using fused ones (e.g. orthog routine)
Exposing ESSEX: Task Macros

**Requires:** GHOST, C++11

**Example:** overlap dot product with axpy

```c
// declare task dot product
PHIST_TASK_DECLARE(dotTask);

// start a dot product s=x^Ty
PHIST_TASK_BEGIN(dotTask)
SUBR(mvec_dot_mvec)(x,y,&s,&iflag);
PHIST_TASK_END_NOWAIT

// wait for the local dot product computations
PHIST_TASK_WAIT_STEP(dotTask);

// perform some other operation v=v+alpha*w
SUBR(mvec_add_mvec)(alpha,w,ONE,v,iflag);

// wait for the dot product reduction
PHIST_TASK_WAIT(dotTask);
```

Macros to
- wrap code segment up as GHOST task
- wait for completion
- step through a kernel (if kernel lib supports it)

not available? ⇒ in-order execution
Holistic Performance Checks

- entire run of phist_Dsubspacejada (computing 10 eigenvalues of a spin-chain matrix)
- roofline model built into kernel interface
- `grep mvecT_times_mvec` to investigate specific kernel:

```
<table>
<thead>
<tr>
<th>function(dim) / (formula)</th>
<th>mtot.exp</th>
<th>%peak-perf</th>
<th>count</th>
<th>max.%peak</th>
<th>min.%peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>[...] (nV=1,nW=1,(V!=W)=1,ldV=60,ldW=1)</td>
<td>3.2e-03</td>
<td>1.24</td>
<td>3</td>
<td>61.4</td>
<td>0.42</td>
</tr>
<tr>
<td>[...] (nV=32,nW=4,(V!=W)=1,ldV=60,ldW=4)</td>
<td>6.4e-02</td>
<td>50.6</td>
<td>15</td>
<td>52.7</td>
<td>47.7</td>
</tr>
<tr>
<td>[...] (nV=36,nW=4,(V!=W)=1,ldV=60,ldW=4)</td>
<td>6.2e-02</td>
<td>61</td>
<td>12</td>
<td>63.5</td>
<td>57.9</td>
</tr>
<tr>
<td>[...] (nV=19,nW=4,(V!=W)=1,ldV=24,ldW=4)</td>
<td>1.3e-01</td>
<td>88.2</td>
<td>40</td>
<td>89.4</td>
<td>85</td>
</tr>
</tbody>
</table>
```

**Variants:** realistic perf check incorporating data layout (strides)
include line numbers of calls or full stack trace in labels
The Test-Driven HPC Development Process

Nightly **PHIST** runs with thousands of unit tests for various

- #MPI procs, #threads
- data types (S/D/C/Z)
- block sizes and memory alignment
- vectorization (SSE, AVX, CUDA)

![Diagram of the Test-Driven HPC Development Process]

- **Algorithms**
  - implement template
  - missing kernels
  - add unit tests
  - optimize numerics

- **Comp. Core**
  - add robust kernels
  - implement optimized version
  - evaluate overall performance

- **Flow**
  - new algorithm
  - established kernel library
  - optimized kernel library
  - application
Integration of PHIST into the software landscape

- Installation of PHIST and GHOST via the SPACK package manager (https://github.com/spack/spack)
- Use different kernel libraries depending on application and hardware
- Available interfaces: C, C++, Fortran’03 and Python
- Can use Trilinos solvers with GHOST or other kernels (e.g. block CG/GMRES, Krylov-Schur, LOBPCG)
- will join the extreme-Scale Development Kit (https://xSDK.info)