A software framework for Equipping Sparse Solvers for the EXa-scale

Jonas Thies* Dominik Ernst† Faisal Shahzad†

A. Alvermann M. Galg.on A. Basermann H. Fehske G. Hager B. Lang G. Wellein

* Inst. of Simulation and Software Technology, DLR
† RRZE, Erlangen University
Univ. of Wuppertal
Univ. of Greifswald

project ESSEX

Software challenges addressed in ESSEX

Exascale challenges:

1. heterogeneous hardware
2. energy efficiency
3. frequent hardware failures

Main tool to address

- (1) and (2): node-level Performance Engineering
- (3) low overhead Checkpoint/Restart

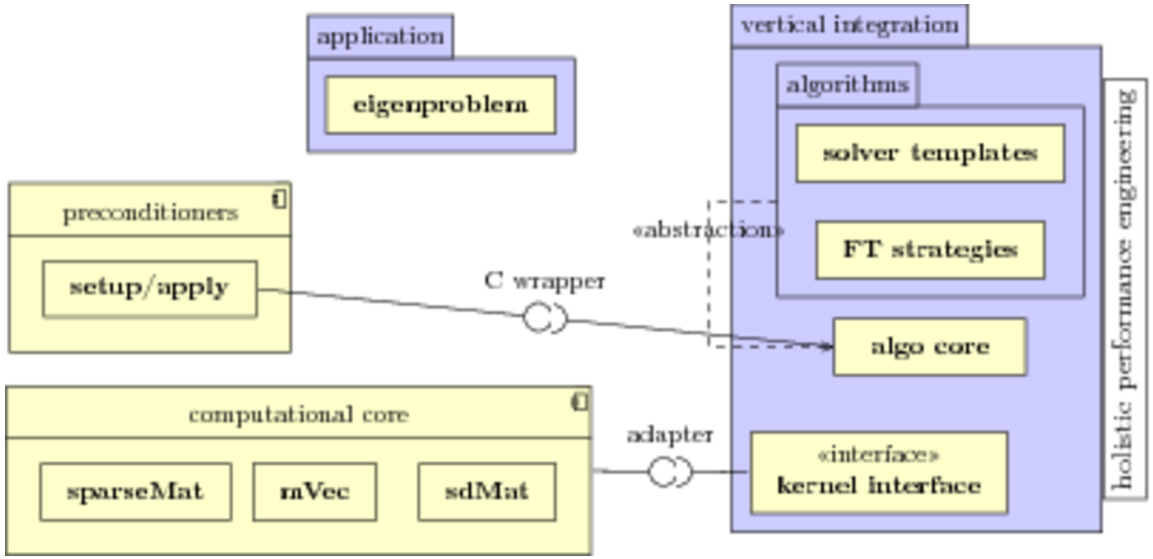
Other practical challenges:

- Portability
- Different communities speak different languages
  - HPC, US/Europe: C, Fortran, C++, C++, C++, ...
  - HPC, Japan: Fortran
  - Data Science: Python+backends (TensorFlow, PyTorch, ...)

Iterative sparse matrix solvers

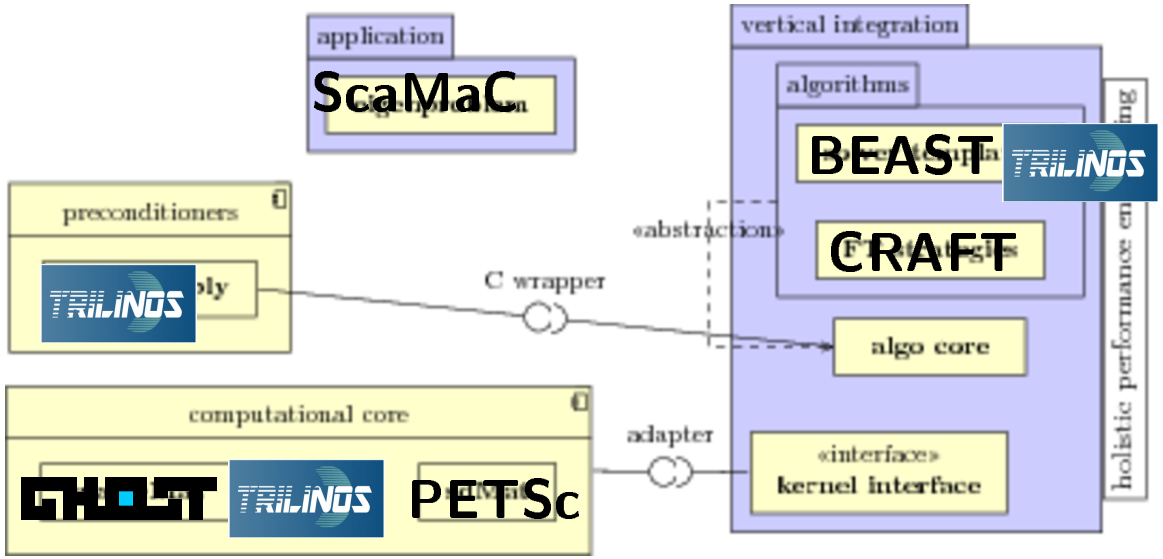
library	solver	target purpose
PHIST	block Jacobi-Davidson QR	10-100 exterior eigenpairs generalized and/or non-Hermitian preconditioning and basis reuse
	linear (Krylov, CGMN) block ILU preconditioning	linear solvers used in BJDQR and BEAST-(C/M)
BEAST	BEAST-C (contour integration, aka FEAST)	many exterior or interior EV, moderate problem size (generally requirees sparse direct solver)
	BEAST-M (moment-based, aka Sakurai-Sugiura)	$\sim$ FEAST with reduced #rhs
	BEAST-P (Chebyshev Filter Diagonalization)	for larger problems (mostly sp-MVMs)

Software architecture



complete implementation in PHIST

Software architecture



complete implementation in PHIST
extensible by other libraries

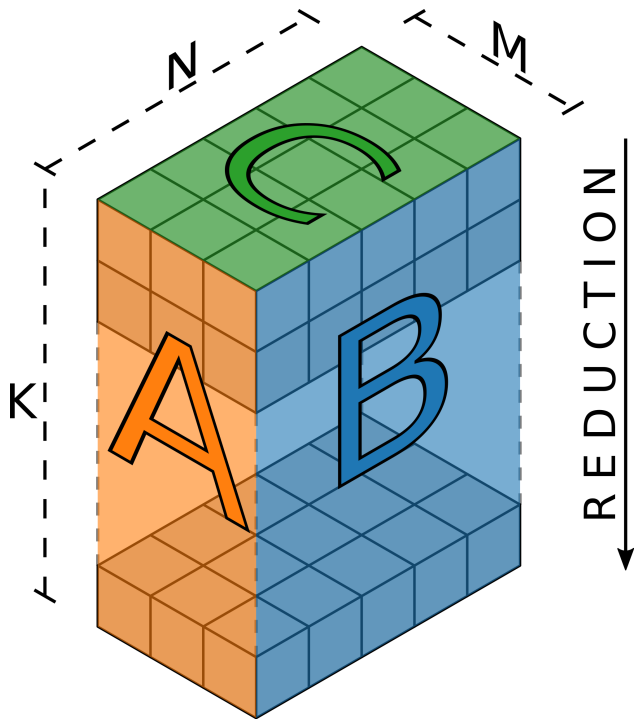
GHOST: custom kernels for sparse solvers

Cannot afford to waste 10 – 90% CPU time at the extreme scale.

- **Example:** 'tall and skinny' matrix-matrix product

$$C \leftarrow A^T B,\ A \in \mathbb{R}^{K \times M}, B \in \mathbb{R}^{K \times N},$$
$$\mathrm{K} \gg \mathrm{M}, \mathrm{N}$$

- typically computed using CUBLAS (DGEMM) on GPUs
- problem: optimized for 'squarish' matrices (compute bound case)
- use case: orthogonal projection ('CGS'): $W \leftarrow (I - VV^T)W$



Iteration space: $K \times N \times M$ elements to be computed

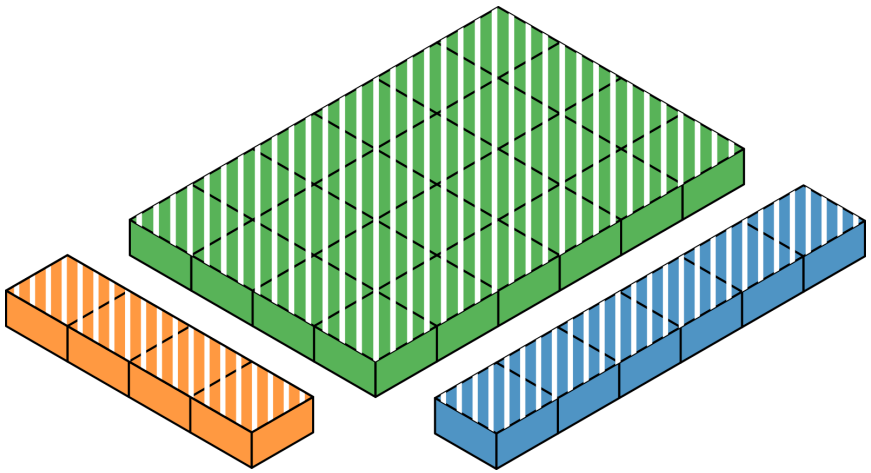
GHOST: custom kernels for sparse solvers

Cannot afford to waste 10 – 90% CPU time at the extreme scale.

- **Example:** 'tall and skinny' matrix-matrix product

$$C \leftarrow A^T B,\ A \in \mathbb{R}^{K \times M}, B \in \mathbb{R}^{K \times N},$$
$$\mathrm{K} \gg \mathrm{M}, \mathrm{N}$$

- typically computed using CUBLAS (DGEMM) on GPUs
- problem: optimized for 'squarish' matrices (compute bound case)
- use case: orthogonal projection ('CGS'): $W \leftarrow (I - VV^T)W$



Variant 1: parallelize reduction (K) only

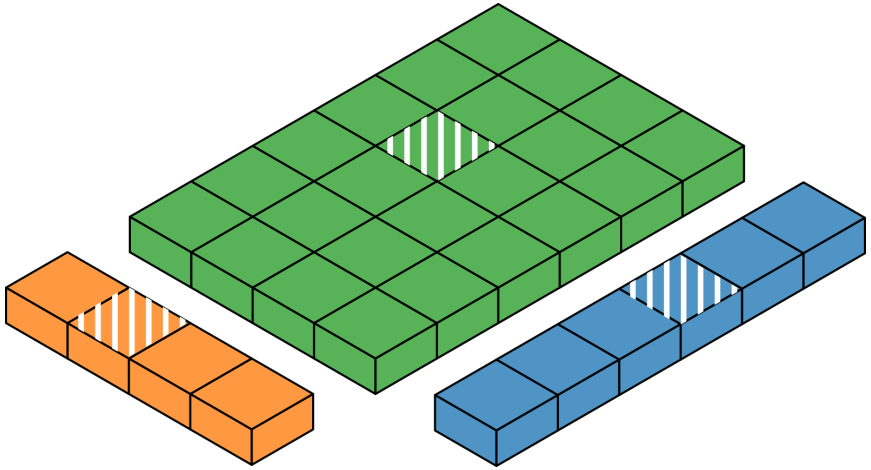
GHOST: custom kernels for sparse solvers

Cannot afford to waste 10 – 90% CPU time at the extreme scale.

- **Example:** 'tall and skinny' matrix-matrix product

$$C \leftarrow A^T B,\ A \in \mathbb{R}^{K \times M}, B \in \mathbb{R}^{K \times N},$$
$$\mathrm{K} \gg \mathrm{M}, \mathrm{N}$$

- typically computed using CUBLAS (DGEMM) on GPUs
- problem: optimized for 'squarish' matrices (compute bound case)
- use case: orthogonal projection ('CGS'): $W \leftarrow (I - VV^T)W$



Variant 2: parallelize K, N, M

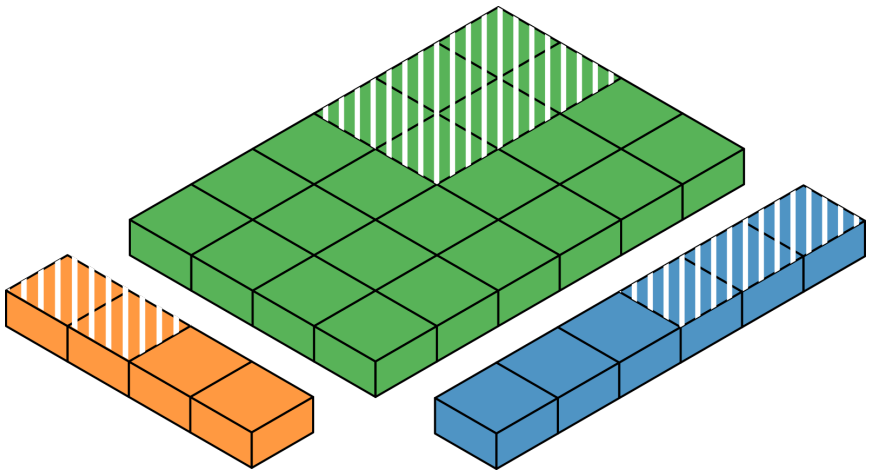
GHOST: custom kernels for sparse solvers

Cannot afford to waste 10 – 90% CPU time at the extreme scale.

- **Example:** 'tall and skinny' matrix-matrix product

$$C \leftarrow A^T B,\ A \in \mathbb{R}^{K \times M}, B \in \mathbb{R}^{K \times N},$$
$$\mathrm{K} \gg \mathrm{M}, \mathrm{N}$$

- typically computed using CUBLAS (DGEMM) on GPUs
- problem: optimized for 'squarish' matrices (compute bound case)
- use case: orthogonal projection ('CGS'): $W \leftarrow (I - VV^T)W$



Variant 3: parallize K, M, N in $m \times n$-tiles

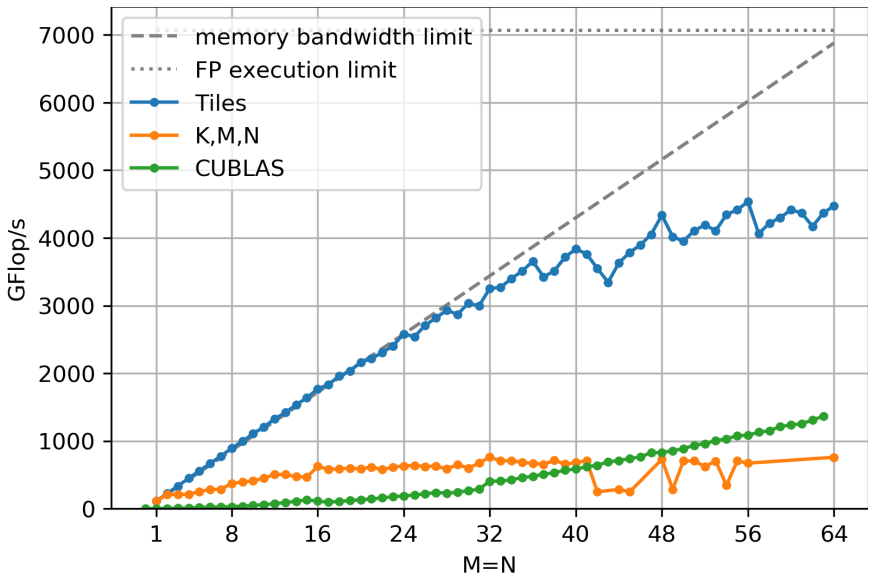
GHOST: custom kernels for sparse solvers

Cannot afford to waste 10 – 90% CPU time at the extreme scale.

- **Example:** 'tall and skinny' matrix-matrix product

$$C \leftarrow A^T B,\ A \in \mathbb{R}^{K \times M}, B \in \mathbb{R}^{K \times N},$$
$$\mathrm{K} \gg \mathrm{M}, \mathrm{N}$$

- typically computed using CUBLAS (DGEMM) on GPUs
- problem: optimized for 'squarish' matrices (compute bound case)
- use case: orthogonal projection ('CGS'): $W \leftarrow (I - VV^T)W$



Performance on V100

CRAFT for fault-tolerance in MPI applications

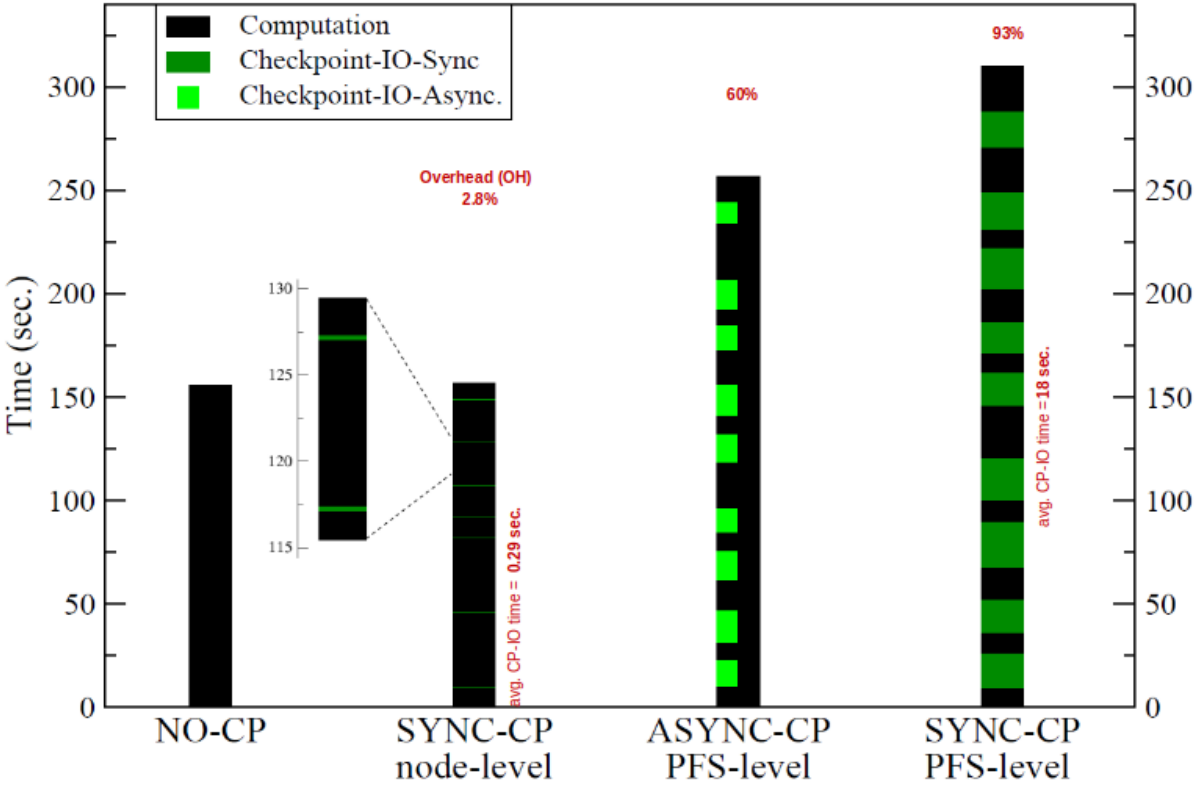
Application-level Checkpoint/Restart:
C++ interface for ALCR with minimal code changes.
checkpointable data-types: PODs, 1D-, 2D-POD arrays, std::vector<> etc.
Extensible for user-defined data-types.
Checkpoint management (e.g. restart checkpoint version, nested checkpoints, etc.).
Optimizations: node-level CR using the SCR library, asynchronous checkpointing.

Automatic Fault Tolerance (AFT):
"Dynamic Process Recovery": Simplified interface for ULFM-MPI.
Recovery models: shrinking, non-shrinking (spawn+merge).

```
#include <mpi.h>
#include <craft.h>
int main(int argc, char* argv[]){
  ...
  int myrank, iteration = 0, cpFreq = 10;
  MPI_Comm FT_Comm;
  MPI_Comm_dup(MPI_COMM_WORLD, &FT_Comm);

  AFT_BEGIN(FT_Comm, &myrank, argv);

  double data = 0;
  Checkpoint myCP( "myCP", FT_Comm);
  myCP.add("data", &data);
  myCP.add("iteration", &iteration);
  myCP.commit();
  if (myCP.needRestart()) {myCP.read();}
  for(; iteration <= n; iteration++){
    /* Computation-communication */
    myCP.updateAndWrite(iteration, cpFreq);
  }
  ...

  AFT_END();

}
```

CRAFT benchmark: Block Jacobi-Davidson QR

FT Scope: application-level CR on the full Meggie-cluster
Num. of nodes 725, num. of MPI-procs 1450
Global CP-size = 135 GB (28 vectors), num. of CP = 9



Finding a common language for sparse linear algebra: the PHIST interface layer

MPI-like:

- objects available only via 'handles' (e.g. `phist_Dmvec_ptr`)
- implementation left to 'kernel library'
- key objects: `mvec`, `sdMat`, `sparseMat`
- `comm` and `map` (very limited, inspired by Petra object model from Trilinos)

graph-based algorithms (ILU, AMG): use underlying data structures

Multi-lingual interface – C++

```
void phist_Dmvec_times_sdMat(double alpha, Dconst_mvec_ptr V,
        Dconst_sdMat_ptr C, double beta,  Dmvec_ptr W, int* iflag);
```

(computes $W \leftarrow \alpha V \cdot C + \beta W$)

... using the auto-generated **C++** bindings:

```
    using PT=phist::types<double>;
    using PK=phist::kernels<double>;
    PT::mvec_ptr V;
    try {
    PK::mvec_create(&V,map,ncols);
    [...]
    PK::mvec_times_sdMat(-2,V,C, 1,W);
    [...]
    PK::mvec_delete(V);
    } catch (phist::Exception e) {...}
```

Multi-lingual interface – Fortran'03

```
void phist_Dmvec_times_sdMat(double alpha, Dconst_mvec_ptr V,
        Dconst_sdMat_ptr C, double beta,  Dmvec_ptr W, int* iflag);
```

(computes $W \leftarrow \alpha V \cdot C + \beta W$)

... using the auto-generated **Fortran 2003** bindings:

```
    use phist_kernels_d
    TYPE(phist_Dmvec_ptr) V
    integer(c_int) :: iflag, ncols
    call phist_Dmvec_create(&V,map,ncols,iflag)
    [...]
    call phist_Dmvec_times_sdMat(-2,V,C, 1,W,iflag)
    [...]
    call phist_Dmvec_delete(V,iflag)
```

(requires manual checking of 'iflag')

Multi-lingual interface – Python

```
void phist_Dmvec_times_sdMat(double alpha, Dconst_mvec_ptr V,
        Dconst_sdMat_ptr C, double beta,  Dmvec_ptr W, int* iflag);
```

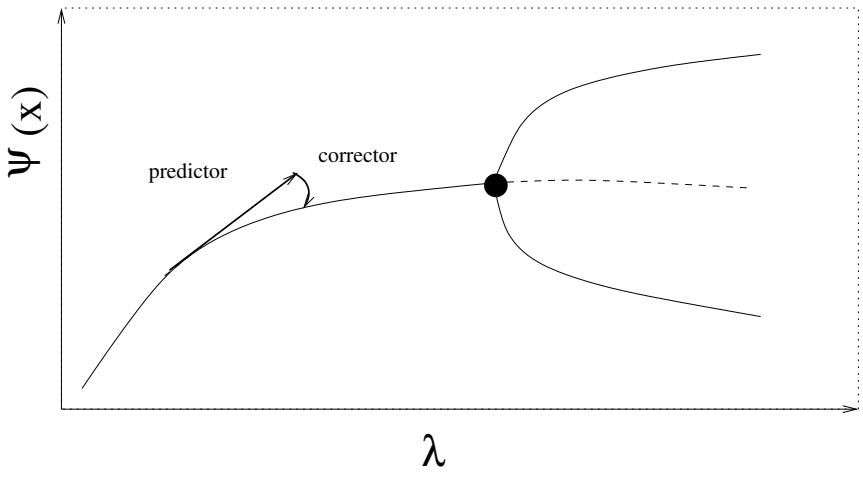
(computes $W \leftarrow \alpha V \cdot C + \beta W$)

... using the auto-generated **Python** bindings:

```
    from phist_kernels import *
    V=phist_Dmvec_ptr()
    try:
        PYST_CHK_IERR(Dmvec_create,V,map,ncols)
        [...]
        PYST_CHK_IERR(Dmvec_times_sdMat,-2,V,C, 1,W)
        [...]
        PYST_CHK_IERR(Dmvec_delete,V)
    except:
        [...]
```

Use case: Python Numerical Continuation Toolkit

- We seek steady states of some discretized PDE
- $\Longrightarrow$ nonlinear system $F(x, \lambda) = 0, F : \mathbb{R}^N \to \mathbb{R}^N$
- Jacobian $J(x, \lambda) = \frac{\partial F_i(x,\lambda)}{\partial x_j}$
  - large
  - sparse
  - non-Hermitian
  - possibly singular



Numerical methods:

1. Newton-Raphson
2. Krylov (GMRES)
3. preconditioning
4. bordering/deflation
5. sparse eigensolver (e.g. JDQR)

PyNCT with object-oriented PHIST-wrapper

Example: MGS orthogonalization in GMRES

original (NumPy):

```
for j in range(it+1):
    alpha=ip(V[:,j],w)
    H[j,it]+=alpha
    w=w-alpha*V[:,j]
```

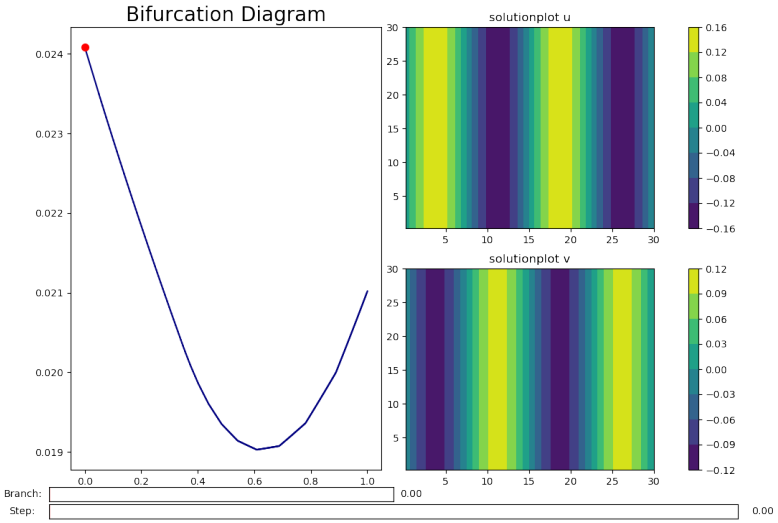
with `linalg` (NumPy or PHIST):

```
for j in range(it+1):
    alpha=ip(V.view(0,n,j,j+1),w)
    H[j,it]+=alpha
    Vj=V.view(0,n,j,j+1)
    w.axpby(-alpha,Vj,1)
```

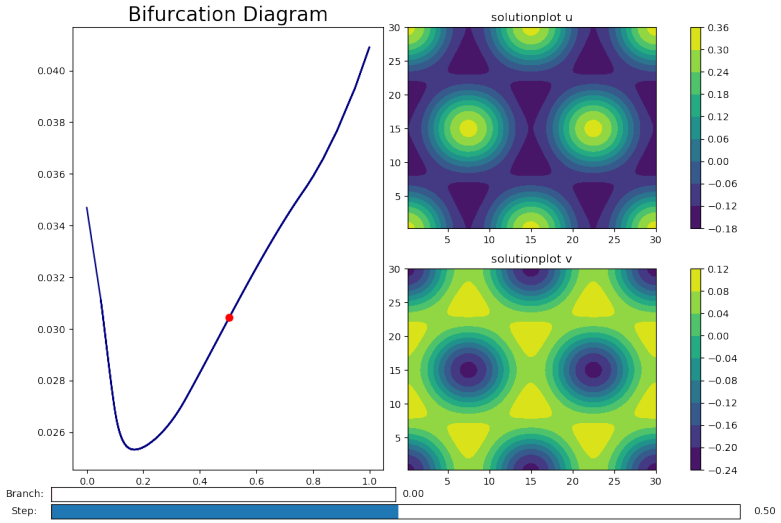
Example: PyNCT for a Turing system

Reaction-Diffusion model for pattern formation in biological systems:

$$\frac{\partial u}{\partial t} = D\delta\nabla^2 u + \alpha u(1 - r_1 v^2) + v(1 - r_2 u),$$
$$\frac{\partial v}{\partial t} = \delta\nabla^2 v + v(\beta + \alpha r_1 uv) + u(\gamma + r_2 v),$$



Branch 2, $n_x = 128$, solution at $r_2 = 0.0$



Branch 4, $n_x = 128$, solution at $r_2 = 0.5$

Comparison: PyNCT/Epetra $\Longleftrightarrow$ LOCA/Epetra

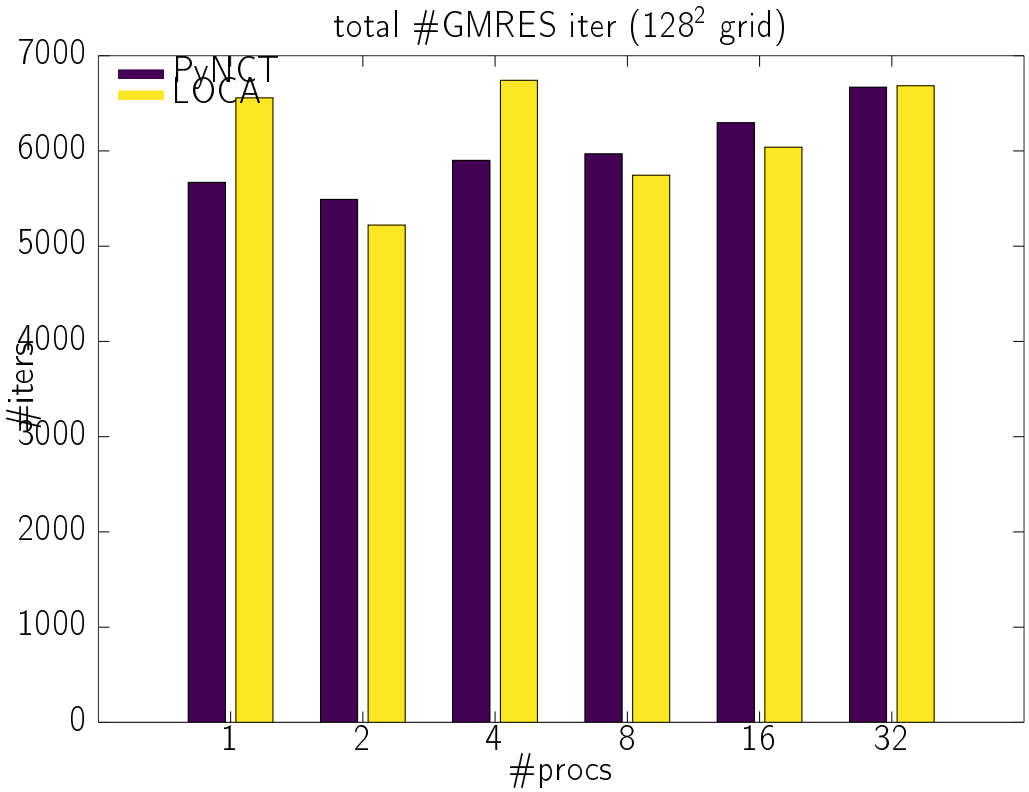
- Experiment: follow branch 4 for $r_2 = 0 \ldots 1$
- 10 continuation steps
- compare operation counts, run time, scalability...

Caveat: algorithmic details are significantly different

Comparison: PyNCT/Epetra $\Longleftrightarrow$ LOCA/Epetra

- Experiment: follow branch 4 for $r_2 = 0 \ldots 1$
- 10 continuation steps
- compare operation counts, run time, scalability...

Caveat: algorithmic details are significantly different

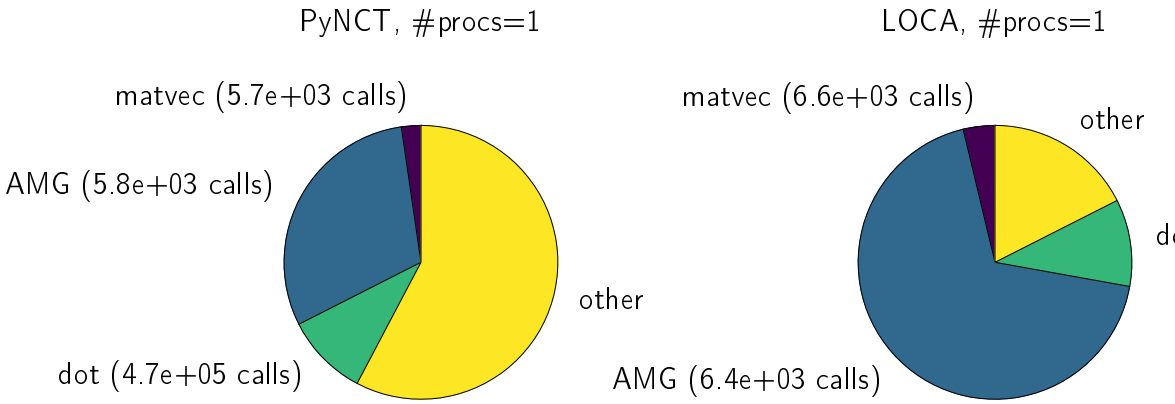


total number of GMRES iterations is similar

Comparison: PyNCT/Epetra $\Longleftrightarrow$ LOCA/Epetra

- Experiment: follow branch 4 for $r_2 = 0 \ldots 1$
- 10 continuation steps
- compare operation counts, run time, scalability...

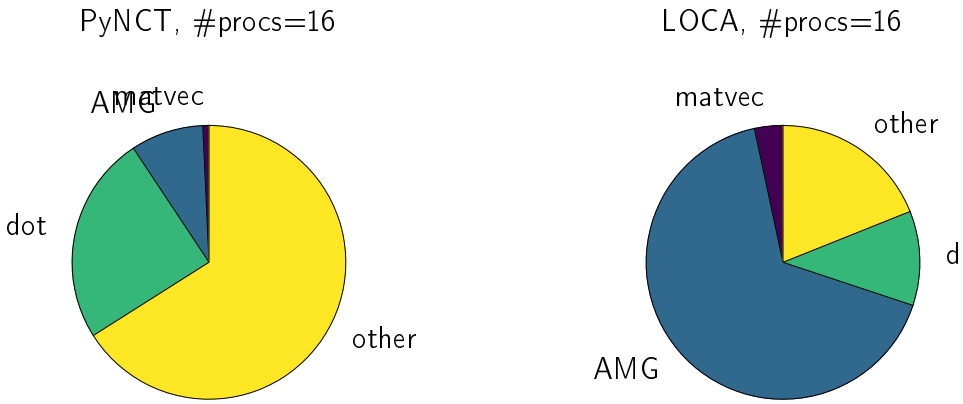
Caveat: algorithmic details are significantly different



Comparison: PyNCT/Epetra $\Longleftrightarrow$ LOCA/Epetra

- Experiment: follow branch 4 for $r_2 = 0 \ldots 1$
- 10 continuation steps
- compare operation counts, run time, scalability...

Caveat: algorithmic details are significantly different

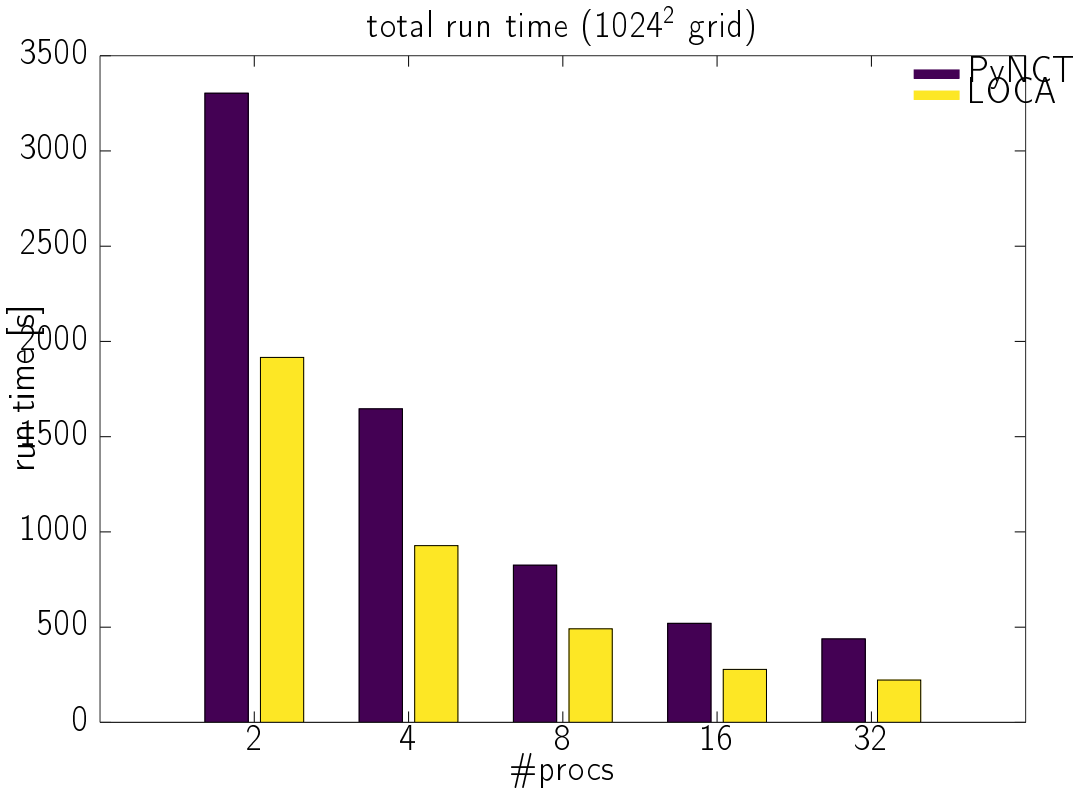


... dot products and Python overhead bad

Comparison: PyNCT/Epetra $\Longleftrightarrow$ LOCA/Epetra

- Experiment: follow branch 4 for $r_2 = 0 \ldots 1$
- 10 continuation steps
- compare operation counts, run time, scalability...

Caveat: algorithmic details are significantly different



Summary: status of the ESSEX software

- 3-clause BSD license
- `https://bitbucket.org/essex/ [ghost|phist|beast|... craft|matrixcollection]/`

GHOST

- **experimental** kernel library
- Intel, NVidia
- S/D/C/Z data types

PHIST

- available in spack and xSDK
- includes own kernels and versions of CRAFT and ScaMaC

BEAST

- uses PHIST
- e.g. Trilinos for direct solvers

CRAFT: can use

- SCR
- MPI-ULFM

ScaMaC stand-alone version with various interfaces