D1.6
Tutorials based on external libraries

Project Title	dealii-X: an Exascale Framework for Digital Twins of the Human Body
Project Number	101172493
Funding Program	European High-Performance Computing Joint Undertaking
Project start date	1 October 2024
Duration	27 months

dealii-X has received funding from the European High-Performance Computing
nt Undertaking Programme under grant agreement Nº 101172493

Deliverable title	Tutorials based on external libraries
Deliverable number	D1.6
Deliverable version	0.1
Date of delivery	Feb 28th 2026
Actual date of delivery	Feb 28th 2026
Nature of deliverable	DEM - Demonstrator, pilot, prototype
Dissemination level	Public
Work Package	WP1
Partner responsible	UNITOV

Abstract	This report contains the tutorial on the external library PSCToolkit, delivered to the dealii-X project during the winter school event at SISSA on Dec. 11th and 12th, 2025.
Keywords	Linear algebra solvers. Libraries. Data structures.

Document Control Information

Version	Date	Author	Changes Made
0.1	Feb 15th 2026	Salvatore Filippone	Initial draft
0.2	Feb 27th 2026	Marco Feder	Tutorial program

Approval Details

Approved by: [Name]
Approval Date: [Date]

Distribution List

- Project Coordinators (PCs)
- Work Package Leaders (WPLs)
- Steering Committee (SC)
- European Commission (EC)

Disclaimer: This project has received funding from the European Union. The views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European High-Performance Computing Joint Undertaking (the “granting authority”). Neither the European Union nor the granting authority can be held responsible for them.

Contents

1 Introduction

Work Package 1 is dedicated to the numerical software backbone of dealii-X, enabling the transition to exascale computing. One key aspect of the WP is the interfacing with external libraries, especially PSCToolkit https://psctoolkit.github.io and MUMPS. This interface is mostly transparent to the application end-user; however, the developers need to acquire a working knowledge of the features available and of the performance that can be attained. The integration status of PSCToolkit and MUMPS in dealii-X has been documented in D1.7, which we refer to for details.

1.1 Purpose of the Document

The purpose of this document is to make available in deliverable format the tutorial presentations given at the dealii-X winter school event at SISSA on Dec. 10th and 11th, 2025. The present tutorial program can be found at https://github.com/psctoolkit/dealii/blob/configure_amg4psblas/examples/benchmark. We also report in the Appendix the slides of the presentation given at the winter school event.

2 PSCToolkit: PSBLAS 3.9 & AMG4PSBLAS 1.2 Tutorial

The new interface targets latest release candidates PSBLAS 3.9 and AMG4PSBLAS 1.2. In order to demonstrate the correctness of the new interface, we describe here a tutorial program which shows how to use the wrappers in a simple but illustrative test case, in the spirit of classical tutorials programs of the deal.II library[1].

We consider the following variable-coefficient Poisson problem on a bounded polygonal domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$:

$$-\nabla \cdot (\mathbf{K}\nabla u) = f \quad \text{in } \Omega = [0,1]^d,$$

where $\mathbf{K}$ is a symmetric and uniformly positive definite tensor and u satisfies homogeneous Dirichlet boundary conditions on $\partial\Omega$. We discretize the continuous problem

[1] https://dealii.org/developer/doxygen/deal.II/Tutorial.html

with standard nodal Lagrangian $\mathcal{Q}^1$ elements on quadrilateral meshes.

All the preliminary steps involved in a standard finite element pipeline, i.e. mesh generation, partitioning of the computational grid among MPI processes, construction of finite element spaces and distribution of degrees of freedom are orthogonal to the present interface, which only needs to interact with deal.II exclusively at the linear algebra level, taking as input the assembled distributed sparse matrix and right-hand side vector.

For this reason, we do not report here the details regarding the setup of the problem, which can be found in the tutorial program `step-40` of the deal.II library, and we focus instead only on the minor modifications required to use the PSCToolkit wrappers. Since our problem yields a symmetric positive definite linear system, we use the conjugate gradient method preconditioned by one V-cycle of the algebraic multigrid preconditioner implemented by AMG4PSBLAS. Details about the setup of the preconditioner will be presented later.

2.1 Include files and problem setup

In addition to classical deal.II include files, the tutorial program includes the headers which contains the interface to PSBLAS and AMG4PSBLAS.

Listing 1: New header files

```
1 #include <deal.II/lac/psblas_precondition.h>
2 #include <deal.II/lac/psblas_sparse_matrix.h>
3 #include <deal.II/lac/psblas_vector.h>
```

The tutorial program supports parameter files that can be used at runtime to configure the details of the problem and the solver without recompiling. These parameter files allow the user to control, among other settings:

- the number of refinement cycles and whether adaptive mesh refinement (AMR) is enabled,
- the solver tolerances (relative and absolute) for the iterative linear solver,
- the details of the AMG preconditioner, such as coarsening strategy, smoother type, and number of smoothing steps.

As common in tutorial programs, the triangulation, system matrix, right-hand side

vector, and solution vector are declared as class members of the main class, and the setup of the linear system is performed in a dedicated member function. The main difference with respect to a standard tutorial program is that the system matrix and vectors are declared using the PSBLAS wrappers. Listing 2 shows the code snippet for the class declaration of the main class of the tutorial program, where only the PSCToolkit-specific members are shown.

Listing 2: Class declaration (only PSCToolkit-specific members shown).

```
template <int dim>
class LaplaceProblem
{
public:
  // declaration of constructors and run()

private:
  // data structures for triangulation, degrees of freedom, etc. as in step-40

  PSCToolkit::SparseMatrix<double> system_matrix;
  PSCToolkit::Vector<double>       locally_relevant_solution;
  PSCToolkit::Vector<double>       system_rhs;
};
```

The allocation of matrices and vectors are performed in the `setup_system()` member function, which is called at the end of each refinement cycle. The user interface is identical to the one of the other linear algebra classes: we first need to acquire information about the MPI communicator and the degrees of freedom owned[2] by the current process. Then, the `reinit()` member function performs the allocation of distributed sparse matrices and vectors. Listing 3 shows the code snippet for the setup of the linear system and vectors.

Listing 3: Setup of the linear system and vectors.

```
template <int dim>
void LaplaceProblem<dim>::setup_system()
{

  locally_owned_dofs    = dof_handler.locally_owned_dofs();
  locally_relevant_dofs =
    DoFTools::extract_locally_relevant_dofs(dof_handler);

  locally_relevant_solution.reinit(locally_owned_dofs,
                                   locally_relevant_dofs,
                                   mpi_communicator);
  system_rhs.reinit(locally_owned_dofs, mpi_communicator);

  PSCToolkit::SparsityPattern sparsity_pattern(locally_owned_dofs,
                                               mpi_communicator);
```

[2]Plus some so-called ”ghost” or ”halo” indices, i.e. indices owned by neighboring processes.

```
16    system_matrix.reinit(sparsity_pattern, mpi_communicator);
17    system_matrix.reinit(locally_owned_dofs, locally_owned_dofs, dsp,
18                         mpi_communicator);
19  }
```

The assembly loop is performed as in any standard tutorial program, looping over cells owned by the current process and scattering local contributions to the global system matrix and right-hand side vector. The implementation of the assembly loop can be found in the `assemble_system()` member function, which is to all extends identical to many other tutorial programs.

2.2 Solver and preconditioner setup

We finally describe the `solve()` function, which involves the setup of the linear solver and the preconditioner, as well as the solution of the linear system. As previously mentioned, the positive definiteness of the bilinear form calls for the use of the preconditioned conjugate gradient (PCG) method. At the solver stage, the interface to PSCToolkit becomes essentially transparent to the user: any Krylov subspace solver provided by the native deal.II library can be instantiated with PSCToolkit matrix and vector wrappers, since they comply with the linear-algebra interface expected by deal.II. Hence, we use the native `SolverCG` class, which implements the PCG method, and is templated on the vector type, which is in this case `PSCToolkit::Vector<double>`. The workflow is shown in Listing 4 where, for the sake of clarity, we omit the preconditioner settings, which are reported in Listing 5.

Among the several available configuration options, we highlight the integration with the MUMPS package (cfr. deliverable D1.7), which can be employed as a direct coarse-level solver within the AMG preconditioner. This combination is particularly effective for achieving robust convergence when algebraic multigrid methods benefit from an accurate solution at the coarsest level. Moreover, parallel-efficient coarse solvers can improve the overall scalability of the solver, especially in contexts where the size of coarse-level system is large.

The solver setup thus remains unchanged with respect to standard user codes; only the AMG4PSBLAS preconditioner settings (coarsening strategy, coarse solver, smoothers, tolerances) may require tuning by users.

Listing 4: Native SolverCG class with PSCToolkit data structures.

```
template <int dim>
void LaplaceProblem<dim>::solve()
{

  PSCToolkit::Vector<double> completely_distributed_solution(
    locally_owned_dofs, mpi_communicator);
  SolverCG<PSCToolkit::Vector<double>> solver(solver_parameters);

  typename PSCToolkit::PreconditionAMG::AdditionalData prec_data;
  // fill prec_data with the desired settings for the AMG preconditioner
  PSCToolkit::PreconditionAMG preconditioner;
  preconditioner.initialize(system_matrix, prec_data);
  solver.solve(system_matrix,
               completely_distributed_solution,
               system_rhs,
               preconditioner);
    }
```

Listing 5: Excerpt of parameter file with AMG preconditioner settings.

```
subsection Laplace Problem<2>
  // problem-related parameters (refinement cycles, boundary ids, etc.)

  subsection AMG control
    set Aggregation filter             = FILTER
    set Aggregation threshold          = 1e-2
    set Aggregation type               = SOC1
    set Aggregation size               = 8
    set Coarse matrix type             = DIST
    set Coarse solver                  = MUMPS
    set Cycle type                     = VCYCLE
    set Number of cycles               = 1
    set Parallel aggregation algorithm = DECOUPLED
    set Prolongator aggregation        = SMOOTHED
    set Smoother sweeps                = 3
    set Smoother type                  = FBGS
    set Verbose AMG info               = true
  end
end
```

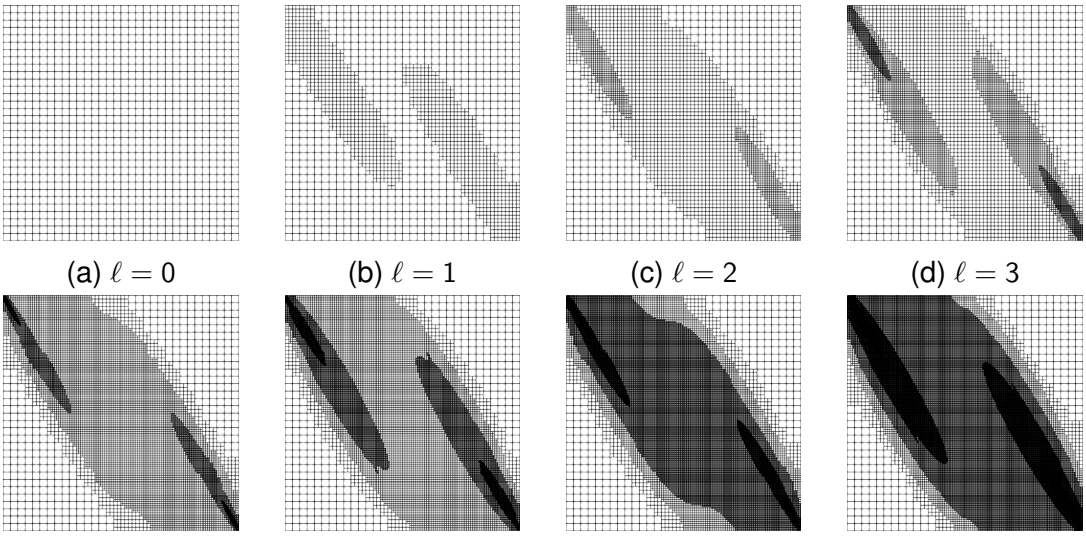
2.3 Numerical experiment

We consider the $\mathcal{Q}^1$ discretization of the anisotropic Poisson problem with diffusion tensor

$$\mathbf{K} = \begin{pmatrix} \cos^2\theta + \varepsilon \sin^2\theta & (1-\varepsilon)\sin\theta\cos\theta \\ (1-\varepsilon)\sin\theta\cos\theta & \sin^2\theta + \varepsilon\cos^2\theta \end{pmatrix}, \tag{1}$$

and right-hand side $f = 1$. The CG solver is stopped when the relative residual is reduced by a factor of 10^{-6}, or when the absolute residual is smaller than 10^{-10}.

Figure 1 shows the solution obtained with several AMR cycles starting from a Cartesian grid, and letting the AMR be driven by a suitable error estimator. The anisotropy parameters are $\varepsilon = 100$ and $\theta = \frac{\pi}{6}$, which yield a challenging test case for algebraic multigrid methods, especially when AMR is employed.



(a) $\ell = 0$ (b) $\ell = 1$ (c) $\ell = 2$ (d) $\ell = 3$

(e) $\ell = 4$ (f) $\ell = 5$ (g) $\ell = 6$ (h) $\ell = 7$

Figure 1: Example of AMR on a quadrilateral grid for an anisotropic Poisson problem with $\ell = 1, \ldots, 7$ levels of refinements from an original tensor product grid $\ell = 0$.

Iteration counts obtained with the setup presented in Listing 5 are reported in Table 1, where we show the results for the two-dimensional case with several AMR cycles, and for different numbers of MPI processes. Iterations are fairly robust with respect to the number of MPI processes and the AMR level.

	AMR level							
n_p	0	1	2	3	4	5	6	7
1	17	20	24	26	31	32	38	39
2	14	19	24	26	30	32	38	40
4	8	14	21	25	30	31	37	39
8	9	10	16	23	29	33	39	39
16	9	10	10	16	23	26	36	39
32	9	10	10	11	17	25	27	36
64	10	10	10	11	11	18	26	27
128	11	12	12	10	10	12	19	26
256	11	14	12	13	11	12	12	13

Table 1: Iteration count for the anisotropic 2D Poisson problem ($\epsilon = 100$, $\theta = \frac{\pi}{6}$) varying number of processes n_p and number of refinement levels. Results obtained on four compute nodes.

Tutorial Presentation Slides

We report in this appendix the slides of the presentation given at the dealii-X winter school event at SISSA on Dec. 10th and 11th, 2025.

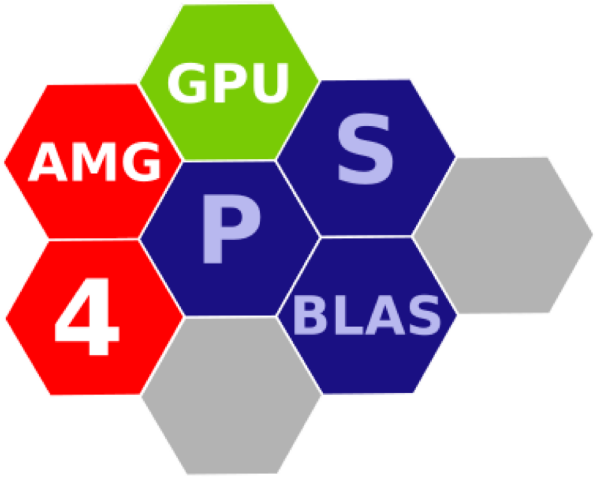
PSCToolkit
PSBLAS 3.9 & AMG4PSBLAS 1.2
Sparse Computation & Iterative Solvers for HPC

Salvatore Filippone — salvatore.filippone@uniroma2.it

dealii X

Exascale Framework for Digital Twins of the Human Body

dealii X PSCToolkit



Collaborators, funding and acknowledgments

Pasqua D'Ambra,
Consiglio Nazionale delle Ricerche
Istituto per le Applicazioni del Calcolo
"M. Picone"

Fabio Durastante,
Università di Pisa
CNR
Istituto per le Applicazioni del Calcolo
"M. Picone"

Salvatore Filippone,
Università di Roma Tor Vergata
CNR — Istituto per le Applicazioni del Calcolo "M. Picone"
Cranfield University (UK)
Lawrence Berkeley Lab

ICSC Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

EoCoE Project ID: 824158 European Commission Horizon 2020 European Union funding for Research & Innovation

The PSCToolkit Team

Main team:

- Salvatore Filippone
- Fabio Durastante
- Pasqua D'Ambra

Contributors:

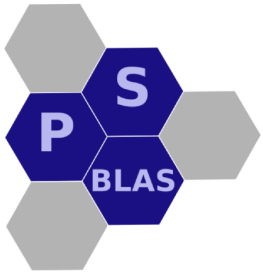
- Marco Feder
- Simone Staccone
- Luca Pepè Sciarria
- Soren Rasmussen
- Zaak Beekman
- Andrea Di Iorio
- Daniela di Serafino
- Theophane Loloum
- Dimitri Walker
- Davide Barbieri
- Alessandro Fanfarillo
- Alfredo Buttari
- Ambra Abdullahi Hassan
- Thomas Amestoy
- Michele Martone
- Michele Colajanni
- Fabio Cerioni
- Stefano Maiolatesi
- Dario Pascucci

Available from `https://psctoolkit.github.io/`

PSCToolkit: PSBLAS Parallel Sparse BLAS

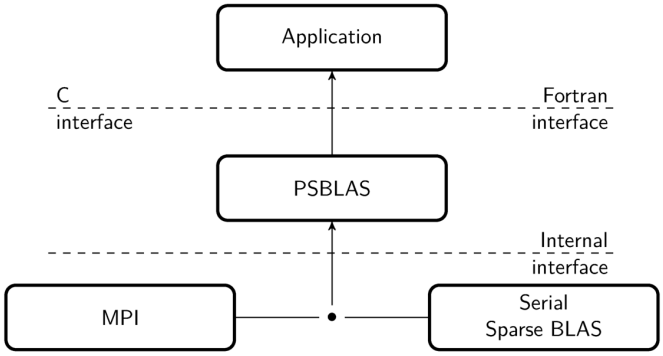
Main features:

- Designed for iterative solvers; but, support for mesh handling;
- Main application: differential problems;
- Data allocation through graph partitioning;
- Support for overlap;



Lots of previous work in standards for sparse and dense linear algebra (see Duff et al. [1997], Blackford and et al. [2002]); described in Filippone and Colajanni [2000], Filippone and Buttari [2012]; version 3.9.0 to be released by year's end.
Available from `https://psctoolkit.github.io/products/psblas`
RPMs available for Fedora and CentOS; SPACK builds under development, will be available with the next formal release.

Library Structure



autotools

```
./configure \
  --with-blas=... \
  --prefix=...
make
make install
```

CMake

```
mkdir build
cd build
cmake ..
make; make install
```

Why PSBLAS?

Sparse Matrices and Krylov methods

A matrix is sparse when there are so many zeros (nonzeros are typically $\mathcal{O}(n)$) that it pays off to take advantage of them in the computer representation. James Wilkinson

Why PSBLAS?

Sparse Matrices and Krylov methods

A matrix is sparse when there are so many zeros (nonzeros are typically $\mathcal{O}(n)$) that it pays off to take advantage of them in the computer representation. James Wilkinson

Methods of choice: Search for a solution by projection

$$\begin{aligned} \mathbf{x}_m &\in \mathcal{K}_m(A, \mathbf{r}_0) \\ \mathbf{r}_m = \mathbf{b} - A\mathbf{x}_m &\perp \mathcal{K}_m(A, \mathbf{r}_0) \\ \mathcal{K}_m(A, \mathbf{r}_0) &= Span\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{m-1}\mathbf{r}_0\} \end{aligned}$$

Krylov subspace (growing with iteration until $\mathbf{x}_m$ is good enough)

Why PSBLAS?

Sparse Matrices and Krylov methods

A matrix is sparse when there are so many zeros (nonzeros are typically $\mathcal{O}(n)$) that it pays off to take advantage of them in the computer representation. James Wilkinson

Methods of choice: Search for a solution by projection

$$\begin{aligned} \mathbf{x}_m &\in \mathcal{K}_m(A, \mathbf{r}_0) \\ \mathbf{r}_m = \mathbf{b} - A\mathbf{x}_m &\perp \mathcal{K}_m(A, \mathbf{r}_0) \end{aligned}$$

$$\mathcal{K}_m(A, \mathbf{r}_0) = Span\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{m-1}\mathbf{r}_0\}$$

Krylov subspace (growing with iteration until $\mathbf{x}_m$ is good enough)

Conjugate Gradient (CG) for s.p.d. matrices (1952). CG Convergence

$$\frac{\|\mathbf{e}_k\|_A}{\|\mathbf{e}_0\|_A} \leq 2\left(\frac{a-1}{a+1}\right), \quad a = \sqrt{\mu(A) = \lambda_{max}/\lambda_{min}}$$

$\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k$ error at iteration k, λ eigenvalue of A

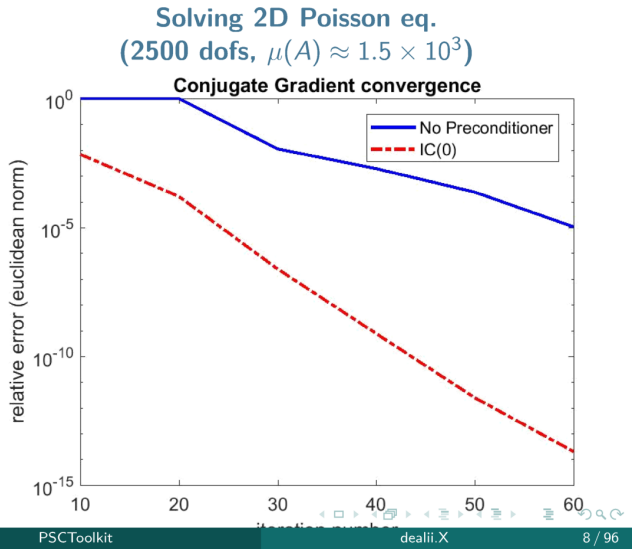
An example Conjugate Gradient method

Compute $r^{(0)} = b - Ax^{(0)}$	`call psb_geaxpby(one,b,zero,r,desc_a,info)`
	`rho = zero`
for $i = 1, 2, \ldots$	`iterate: do it = 1, itmax`
solve $Mz^{(i-1)} = r^{(i-1)}$	`call psb_spsm(one,L,r,zero,w,desc_a,info)`
	`call psb_spsm(one,U,w,zero,z,desc_a,info)`
$\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$	`rho_old = rho; rho = psb_gedot(r,z,desc_a,info)`
if $i = 1$	`if (it == 1) then`
$p^{(1)} = z^{(0)}$	`call psb_geaxpby(one,z,zero,p,desc_a,info)`
else	`else`
$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$	`beta = rho/rho_old`
$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$	`call psb_geaxpby(one,z,beta,p,desc_a,info)`
endif	`endif`
$q^{(i)} = Ap^{(i)}$	`call psb_spmm(one,A,p,zero,q,desc_a,info)`
$\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$	`sigma = psb_gedot(p,q,desc_a,info); alpha = rho/sigma`
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$	`call psb_geaxpby(alpha,p,one,x,desc_a,info)`
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	`call psb_geaxpby(-alpha,q,one,r,desc_a,info)`
Check convergence: $\|r^{(i)}\|_2 \leq \epsilon \|b\|_2$	`rn2 = psb_genrm2(r,desc_a,info)`
	`bn2 = psb_genrm2(b,desc_a,info)`
	`err = rn2/bn2`
	`if (err.lt.eps) exit iterate`
end	`enddo iterate`

Preconditioning

Solve the system $B^{-1}A\mathbf{x} = B^{-1}\mathbf{b}$, with matrix $B \approx A^{-1}$ (left preconditioner) such that:

$$\mu(B^{-1}A) << \mu(A)$$



PSBLAS Contents

- Parallel Environment handling;
- Computational kernels:
  - Sparse matrix by dense vector product;
  - Sparse triangular systems solution;
  - Vector and matrix norm;
  - Dense vector sums;
  - Dot products;
- Data exchange and update;
- Data Management;
- Preconditioner setup;
- Iterative solvers

Table of Contents

Parallel Environment

We defined our parallel environment:

- Implemented in pure MPI;
- Subset of MPI communication modes;
- MPI directly available when/if needed;
- Fortran generic interfaces available (no type mismatch!)

Basic operations:

- Initialize/close a process grid (parallel machine environment handling)
- Point-to-point send/receive
- Collective operations: Broadcasts, Reductions, Scan-sum;

Parallel Environment

Each context (MPI communicator) identifies a virtual parallel machine:

```
call psb_init(ctxt [, np, basectxt, ids])
call psb_info(ctxt,iam,np)
call psb_exit(ctxt [, close=.true.])
```

Rules:

- psb_init *must* be called before anything else;
- Creates a new communicator: the library communication is cleanly separated from the application;
- It is legal to specify a (permuted) subset of the available processes;

MPI interoperability

```
mpicomm = psb_get_mpi_comm(ctxt)
mpirank = psb_get_mpi_rank(ctxt, id)
```


Parallel Environment: Hello World!

Writing helloworld.f90:

```
program hello_world
  use psb_base_mod
  implicit none
  type(psb_ctxt_type) :: ctxt
  integer(psb_ipk_) :: iam, np
  character(len=20)  :: name
  call psb_init(ctxt)
  call psb_info(ctxt,iam,np)
  name='helloworld'
  if (iam == psb_root_) then
    write(*,*)'Welcome to PSBLAS version: ',psb_version_string_
    write(*,*)'This is the ',trim(name),' sample program'
    write(*,*)'I am process ',iam,'of ',np
  else
    write(*,*)'I am process ',iam,'of ',np
  end if
  call psb_exit(ctxt)
  stop
end program hello_world
```

The C Interface

The PSBLAS library comes with a C interface.
The general rule for switching between the Fortran and C variants of the same PSBLAS routine is

```
call psb_<something>(...) ↦ psb_c_[PRECISION]<something>(...);
```

The routines defining the parallel environment are now:

```
psb_i_t psb_c_init();
void psb_c_info(psb_i_t ctxt, \
^^Ipsb_i_t *iam, psb_i_t *np);
void psb_c_exit(psb_i_t ctxt);
```

The headers for these routines are in the `psb_base_cbind.h` file.

The C version of the Hello World!

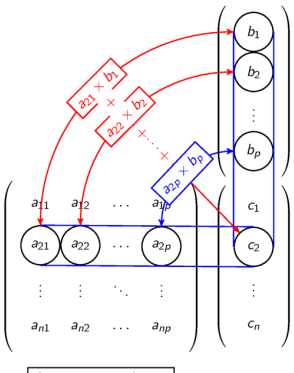
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "psb_base_cbind.h"
int main(int argc, char *argv[])
{
 int ctxt, iam, np;
 char name[]="c_helloworld";

 ctxt = psb_c_init();
 psb_c_info(ctxt,&iam,&np);

 if (iam == 0) {
   printf("This is the %s sample program\n",name);
   printf("I am process %d of %d\n",iam,np);
 } else {
   printf("I am process %d of %d\n",iam,np);
 }

 psb_c_exit(ctxt);
}
```

Table of Contents

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

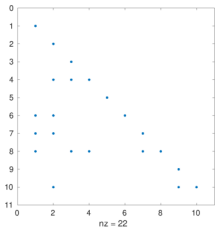
- (Parallel) Sparse matrix by Vector product;



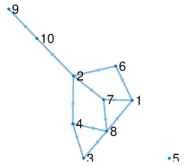
Note: we also need boundary data exchange and mesh management.

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:



- (Parallel) Sparse matrix by Vector product;
- Sparse triangular system solution;



Note: we also need boundary data exchange and mesh management.

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

- (Parallel) Sparse matrix by Vector product;
- Sparse triangular system solution;
- Dot products;

$$< \mathbf{x}, \mathbf{y} >= \sum_{i=1}^{n} x_i y_i$$

Note: we also need boundary data exchange and mesh management.

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

- (Parallel) Sparse matrix by Vector product;
- Sparse triangular system solution;
- Dot products;
- Vector norms;

$$\|\mathbf{x}\|_1 = \sum_i |x_i|$$

$$\|\mathbf{x}\|_2 = \left(\sum_i |x_i|^2 \right)^{\frac{1}{2}}$$

$$\|\mathbf{x}\|_\infty = \max_i |x_i|$$

Note: we also need boundary data exchange and mesh management.

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

- (Parallel) Sparse matrix by Vector product;
- Sparse triangular system solution;
- Dot products;
- Vector norms;
- Matrix norms;

$$\|A\|_1 = \max_j \sum_i |a_{i,j}|$$

$$\|A\|_\infty = \max_i \sum_j |a_{i,j}|$$

Note: we also need boundary data exchange and mesh management.

S. Filippone PSCToolkit dealii.X 18 / 96

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

- (Parallel) Sparse matrix by Vector product;
- Sparse triangular system solution;
- Dot products;
- Vector norms;
- Matrix norms;
- Scaled sums (AXPY-like);

Note: we also need boundary data exchange and mesh management.

Computational kernels

$\mathbf{x}^T\mathbf{y}$ ($\mathbf{x}^H\mathbf{y}$): `dot = psb_gedot(x,y,desc_a,info)`

$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$: `call psb_geaxpby(alpha,x,beta,y,desc_a,info)`

$\max_i |x_i|$: `amax = psb_geamax(x,desc_a,info)`

$\sum_i |x_i|$: `asum = psb_geasum(x,desc_a,info)`

$\|\mathbf{x}\|_2$: `nrm2 = psb_genrm2(x,desc_a,info)`

$\|\mathbf{A}\|_\infty$ `nrmi = psb_spnrmi(A,desc_a,info)`

$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$: `call psb_spmm(alpha,A,x,beta,y,desc_a,info[,trans])`

$\mathbf{y} \leftarrow \alpha DT^{-1}\mathbf{x} + \beta\mathbf{y}$: `call psb_spsm(alpha,T,x,beta,y,desc_a,info[,trans,unitd])`

Note: T is a triangular AND block diagonal matrix (i.e.: Block-Jacobi or Hybrid GS type preconditioners)

Computational kernels

$\mathbf{x}^T\mathbf{y}$ ($\mathbf{x}^H\mathbf{y}$): `dot = psb_gedot(x,y,desc_a,info)`

$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$: `call psb_geaxpby(alpha,x,beta,y,desc_a,info)`

$\max_i |x_i|$: `amax = psb_geamax(x,desc_a,info)`

$\sum_i |x_i|$: `asum = psb_geasum(x,desc_a,info)`

$\|\mathbf{x}\|_2$: `nrm2 = psb_genrm2(x,desc_a,info)`

$\|\mathbf{A}\|_\infty$ `nrmi = psb_spnrmi(A,desc_a,info)`

$\mathbf{y} \leftarrow \alpha A^T\mathbf{x} + \beta\mathbf{y}$: `call psb_spmm(alpha,A,x,beta,y,desc_a,info,trans='T')`

$\mathbf{y} \leftarrow \alpha DT^T\mathbf{x} + \beta\mathbf{y}$: `call psb_spsm(alpha,T,x,beta,y,desc_a,info,trans='T'[,unitd])`

Note: T is a triangular AND block diagonal matrix (i.e.: Block-Jacobi or Hybrid GS type preconditioners)

Computational kernels - C Interfaces

The routines are defined for each data type, e.g., in the `double` case

$\mathbf{x}^T\mathbf{y}$: `psb_d_t psb_c_dgedot(psb_c_dvector *x, psb_c_dvector *y, psb_c_descriptor *desc_a);`

$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$: `psb_i_t psb_c_dgeaxpby(psb_d_t alpha, psb_c_dvector *x, psb_d_t beta, psb_c_dvector *y, psb_c_descriptor *desc_a);`

$\max_i |x_i|$: `psb_d_t psb_c_dgeamax(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\sum_i |x_i|$: `psb_d_t psb_c_dgeasum(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\|\mathbf{x}\|_2$: `psb_d_t psb_c_dgenrm2(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\|\mathbf{A}\|_\infty$ `psb_d_t psb_c_dspnrmi(psb_c_dspmat *A, psb_c_descriptor *desc_a);`

$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$: `psb_i_t psb_c_dspmm(psb_d_t alpha, psb_c_dspmat *A, psb_c_dvector *x, psb_d_t beta, psb_c_dvector *y, psb_c_descriptor *desc_a);`

Note: The headers for these functions are in the file `psb_c_dbase.h`, `psb_c_cbase.h`, `psb_c_sbase.h`, `psb_c_zbase.h`, they can be included all together by including `psb_base_cbind.h`.

Computational kernels - C Interfaces

The routines are defined for each data type, e.g., in the `double` case

$\mathbf{x}^T\mathbf{y}$: `psb_d_t psb_c_dgedot(psb_c_dvector *x, psb_c_dvector *y, psb_c_descriptor *desc_a);`

$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$: `psb_i_t psb_c_dgeaxpby(psb_d_t alpha, psb_c_dvector *x, psb_d_t beta, psb_c_dvector *y, psb_c_descriptor *desc_a);`

$\max_i |x_i|$: `psb_d_t psb_c_dgeamax(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\sum_i |x_i|$: `psb_d_t psb_c_dgeasum(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\|\mathbf{x}\|_2$: `psb_d_t psb_c_dgenrm2(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\|\mathbf{A}\|_\infty$ `psb_d_t psb_c_dspnrmi(psb_c_dspmat *A, psb_c_descriptor *desc_a);`

$\mathbf{y} \leftarrow \alpha A^T\mathbf{x} + \beta\mathbf{y}$: `psb_i_t psb_c_dspmm_opt(psb_d_t alpha, psb_c_dspmat *A, psb_c_dvector *x, psb_d_t beta, psb_c_dvector *y, psb_c_descriptor *desc_a, char *trans, bool doswap);`

Note: The headers for these functions are in the file `psb_c_dbase.h`, `psb_c_cbase.h`, `psb_c_sbase.h`, `psb_c_zbase.h`, they can be included all together by including `psb_base_cbind.h`.

Computational kernels - C Interfaces

The routines are defined for each data type, e.g., in the `double` case

$\mathbf{x}^T\mathbf{y}$: `psb_d_t psb_c_dgedot(psb_c_dvector *x, psb_c_dvector *y, psb_c_descriptor *desc_a);`

$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$: `psb_i_t psb_c_dgeaxpby(psb_d_t alpha, psb_c_dvector *x, psb_d_t beta, psb_c_dvector *y, psb_c_descriptor *desc_a);`

$\max_i |x_i|$: `psb_d_t psb_c_dgeamax(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\sum_i |x_i|$: `psb_d_t psb_c_dgeasum(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\|\mathbf{x}\|_2$: `psb_d_t psb_c_dgenrm2(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$\|\mathbf{A}\|_\infty$ `psb_d_t psb_c_dspnrmi(psb_c_dspmat *A, psb_c_descriptor *desc_a);`

$\mathbf{y} \leftarrow \alpha DT^{-1}\mathbf{x} + \beta\mathbf{y}$: `psb_c_dspsm(psb_d_t alpha, psb_c_dspmat *T, psb_c_dvector *x, psb_d_t beta, psb_c_dvector *y, psb_c_descriptor *desc_a);`

Note: The headers for these functions are in the file `psb_c_dbase.h`, `psb_c_cbase.h`, `psb_c_sbase.h`, `psb_c_zbase.h`, they can be included all together by including `psb_base_cbind.h`.

S. Filippone PSCToolkit dealii.X 20 / 96

An example Conjugate Gradient method

Template CG	PSBLAS Implementation
Compute $r^{(0)} = b - Ax^{(0)}$	`call psb_geaxpby(one,b,zero,r,desc_a,info)`
	`rho = zero`
for $i = 1, 2, \ldots$	`iterate: do it = 1, itmax`
solve $Mz^{(i-1)} = r^{(i-1)}$	`call psb_spsm(one,L,r,zero,w,desc_a,info)`
	`call psb_spsm(one,U,w,zero,z,desc_a,info)`
$\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$	`rho_old = rho`
	`rho = psb_gedot(r,z,desc_a,info)`
if $i = 1$	`if (it == 1) then`
$p^{(1)} = z^{(0)}$	`call psb_geaxpby(one,z,zero,p,desc_a,info)`
else	`else`
$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$	`beta = rho/rho_old`
$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$	`call psb_geaxpby(one,z,beta,p,desc_a,info)`
endif	`endif`
$q^{(i)} = Ap^{(i)}$	`call psb_spmm(one,A,p,zero,q,desc_a,info)`
$\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$	`sigma = psb_gedot(p,q,desc_a,info)`
	`alpha = rho/sigma`
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$	`call psb_geaxpby(alpha,p,one,x,desc_a,info)`
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	`call psb_geaxpby(-alpha,q,one,r,desc_a,info)`
Check convergence:	
$\|r^{(i)}\|_2 \leq \epsilon \|b\|_2$	`rn2 = psb_genrm2(r,desc_a,info)`
	`bn2 = psb_genrm2(b,desc_a,info)`
	`err = rn2/bn2`
	`if (err.lt.eps) exit iterate`
end	`enddo iterate`

Exercise: write the corresponding C version for `double` vectors and matrices

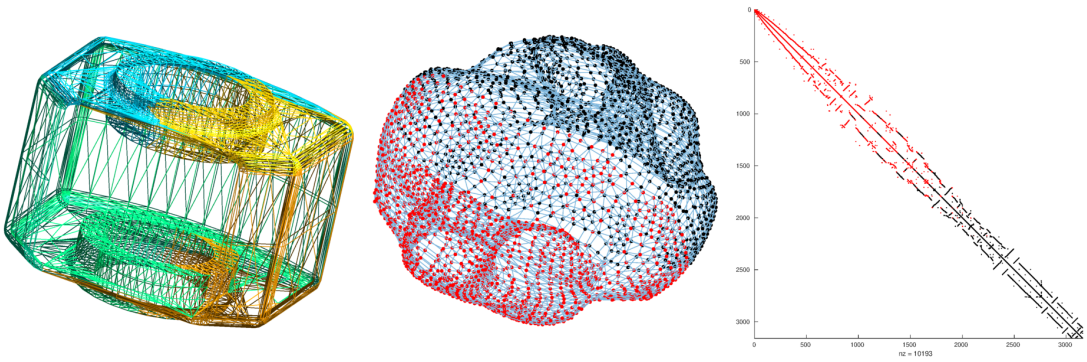
Table of Contents

Data Distribution

Guiding principle: "Owner computes" paradigm. Given an index space $1 \dots N$ (and vectors defined on this index space):

1. The index space is partitioned among processes;
2. Each index has a "home" process;
3. The "home" process holds the authoritative value of the corresponding vector entry;
4. The "home" process performs the arithmetic operations needed to set the value of a vector entry;
5. On each process, the set of "resident" indices will have a local numbering;
6. There is a map between global and local indices; the map is (usually) one-to-one when restricted to "home" processes;
7. There is a certain amount of redundancy due to "halo" indices (see below)

Data Distribution



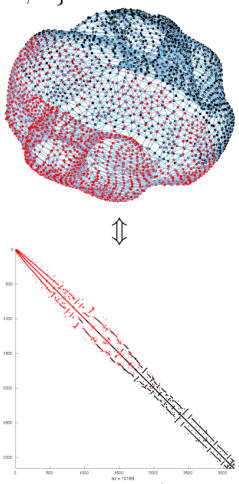
Mesh partition $\Leftrightarrow$ Graph partition $\Leftrightarrow$ Matrix row partition

Finding the optimal decomposition is equivalent to a graph partition problem ($\mathcal{NP}$-complete).

Data Distribution

Isomorphism between sparse matrix (pattern) and a graph: $G = \{V, E\}$ where

$$\begin{aligned} V &= \{v_1, \ldots, v_n\} \\ E &\subseteq V \times V \end{aligned}$$



From a sparse matrix to a graph:

- To each row i there corresponds a vertex v_i;
- To each coefficient a_{ij} there corresponds an edge (v_i, v_j);

From a graph to a sparse matrix (pattern): same as above.

Note: numbering of vertices induces a different pattern (symmetric permutation)

Communication descriptors

What is a communication descriptor?
An opaque object that:

- Keeps track of the parallel machine (ctxt);
- Is associated with a discretization topology (mesh graph plus discretization stencil);
- Stores the mapping of the index space onto the parallel machine;
- Contains all the data necessary to implement a neighbour-to-neighbour data exchange (or: halo data exchange; or: ghost cell update; or: persistent neighborhood all-to-all on a virtual distributed graph topology)

```
^^Icall psb_halo(x,desc)
^^I
```

Communication descriptors

What is a communication descriptor?
An opaque object that:

- Keeps track of the parallel machine (ctxt);
- Is associated with a discretization topology (mesh graph plus discretization stencil);
- Stores the mapping of the index space onto the parallel machine;
- Contains all the data necessary to implement a neighbour-to-neighbour data exchange (or: halo data exchange; or: ghost cell update; or: persistent neighborhood all-to-all on a virtual distributed graph topology)

```
psb_i_t^^Ipsb_c_dhalo(psb_c_dvector *x,
^^Ipsb_c_descriptor *desc_a);
^^I
```


Table of Contents

Storage schemes

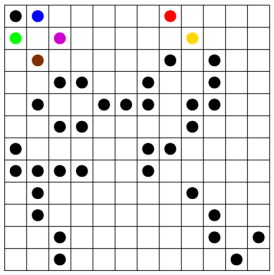
Coordinate storage:

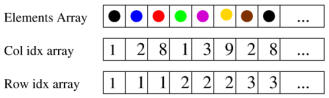
- M Rows;
- N Columns;
- NZ Non zeroes;
- IA(1:NZ) Row indices;
- JA(1:NZ) Column indices;
- AS(1:NZ) Coefficients;

Note: by definition of number of rows we have $1 \leq IA(i) \leq M$, likewise for the columns.

Storage schemes

COO






```
do i=1,nz
 ir = ia(i)
 jc = ja(i)
 y(ir) = y(ir) + as(i)*x(jc)
enddo
```

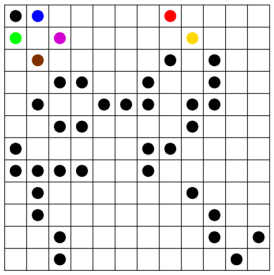
Storage schemes

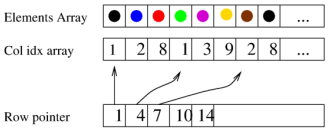
Compressed Storage by Rows:

- M Rows;
- N Columns;
- IA(1:M+1) Pointers to row start;
- JA(1:NZ) Column indices;
- AS(1:NZ) Coefficients;

Storage schemes

CSR






```
do i=1,m
 do j=ia(i), ia(i+1)-1
  y(i) = y(i) + as(j)*x(ja(j))
 enddo
enddo
```

Well, so are we (see also Filippone et al. [2017])

Are you unsure what to use?

Well, so are we (see also Filippone et al. [2017])

Facts:

- Different computer architectures are best exploited by different formats;
- Different formats are suited to different operations (and we need them all);

Are you unsure what to use?

Well, so are we (see also Filippone et al. [2017])

Facts:

- Different computer architectures are best exploited by different formats;
- Different formats are suited to different operations (and we need them all);

Requirements (put your library developer's hat on):

- We want to be able to change in response to machine changes (might be done at compile time);
- We want to be able to change in response to use patterns (need to change at run time)
- We want to switch among formats, some of them unkonwn at compile time;

Essentially, we want objects to switch between different types at runtime

Design Patterns to the Rescue

We want maximum freedom, flexibility, maintainability and performance

we like to have our cake and eat it too

⇒ Need to use **Design Patterns**

- STATE;
- BUILDER;
- MEDIATOR;
- PROTOTYPE.

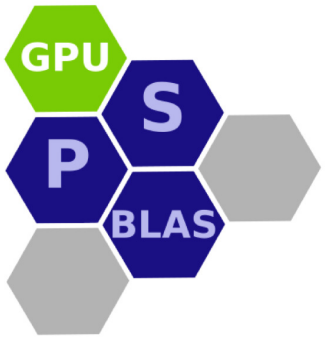
For details see:

- A. Buttari, S. Filippone, ACM TOMS, 2012
- V. Cardellini, S. Filippone and D. Rouson, Scientific Programming, 2014

All problems in computer science can be solved by another level of indirection — Butler Lampson

We have also GPU support!

Our design allows us to manage data on GPUs in a transparent way Cardellini et al. [2014], Filippone et al. [2017]



SpMV performance

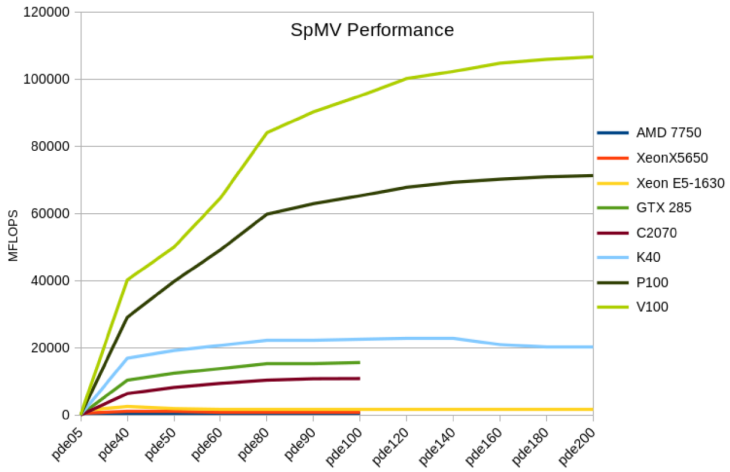


Table of Contents

Data Management

How do we set up a descriptor/sparse matrix?

Data Management

How do we set up a descriptor/sparse matrix?
First step, we have to decide a distribution of the index space of our problem, and how we are going to specify it:

1. Assign a process to each index;
2. Assign a list of indices to each process;
3. Assign a bunch of consecutive indices to each process;
4. Other;

This is done with the initialization routine `psb_cdall`

dealiiX Data Management

How do we set up a descriptor/sparse matrix?
First step, we have to decide a distribution of the index space of our problem, and how we are going to specify it:

1. Assign a process to each index;
2. Assign a list of indices to each process;
3. Assign a bunch of consecutive indices to each process;
4. Other;

For the C interface, we have different allocation routines for the different styles:

```
psb_i_t     psb_c_cdall_vg(psb_l_t ng, psb_i_t *vg,
        psb_c_ctxt ctxt, psb_c_descriptor *desc_a);
psb_i_t     psb_c_cdall_vl(psb_i_t nl, psb_l_t *vl,
        psb_c_ctxt ctxt, psb_c_descriptor *desc_a);
psb_i_t     psb_c_cdall_nl(psb_i_t nl,
        psb_c_ctxt ctxt, psb_c_descriptor *desc_a);
```


dealiiX Descriptor Allocation

```
! Assign a process to each index, e.g. via
!(serial) Metis
if (iam == 0) then
 call bld_mtpart(. ......)
 call getv_mtpart(v)
endif
call psb_bcast(ctxt,v,root=0)
call psb_cdall(ctxt,desc,info,vg=v)
```

Global size: `m = size(v)`

Metis

Information on how to obtain and use Metis and its parallel version can be found at `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

Descriptor Allocation

```
! Assign a process to each index, e.g. via
!(serial) Metis
if (iam == 0) then
 call bld_mtpart(. ......)
 call getv_mtpart(v)
endif
call psb_bcast(ctxt,v,root=0)
call psb_cdall(ctxt,desc,info,vg=v)
```

Global size: `m = size(v)`
The corresponding C style allocation is obtained using

```
psb_i_t psb_c_cdall_vg(psb_l_t ng, psb_i_t *vg,
^^Ipsb_c_ctxt ctxt,
^^Ipsb_c_descriptor *desc_a);
```


Descriptor Allocation

```
! Assign a bunch of contiguous indices to each process
call psb_cdall(ctxt,desc,info,nl=nl)
```

There is **NO** requirement that the NLs be evenly distributed;
Global size: `m = psb_sum(ctxt,nl)`

dealiiX Descriptor Allocation

```
! Assign a bunch of contiguous indices to each process
call psb_cdall(ctxt,desc,info,nl=nl)
```

There is **NO** requirement that the NLs be evenly distributed;
Global size: `m = psb_sum(ctxt,nl)`
The corresponding C style allocation is obtained using

```
psb_i_t psb_c_cdall_nl(psb_i_t nl,
        psb_c_ctxt ctxt, psb_c_descriptor *desc_a);
```


dealiiX Descriptor Allocation

```
! Build a list of locally owned indices
do i=1,nl
 vl(i) = get_ith_index(....)
end do
call psb_cdall(ctxt,desc,info,vl=vl)
```

There is **NO** requirement for the indices to be contiguous, or even ordered.
Global size: `m = psb_sum(ctxt,size(vl))`

Descriptor Allocation

```
! Build a list of locally owned indices
do i=1,nl
 vl(i) = get_ith_index(....)
end do
call psb_cdall(ctxt,desc,info,vl=vl)
```

There is **NO** requirement for the indices to be contiguous, or even ordered.
Global size: `m = psb_sum(ctxt,size(vl))`
The corresponding C style allocation is obtained using

```
psb_i_t psb_c_cdall_vl(psb_i_t nl, psb_l_t *vl,
^^Ipsb_c_ctxt ctxt, psb_c_descriptor *desc_a);
```


Descriptor Allocation

```
! Build an arbitrary strategy
interface
 subroutine parts(glob_index,nrow,np,pv,nv)
  integer, intent (in)  :: glob_index,np,nrow
  integer, intent (out) :: nv, pv(*)
 end subroutine parts
end interface

call psb_cdall(ctxt,desc,info,m=mg,parts=parts)
```

Here we may even assign an index to multiple processes (aka *overlap*)!
Global size: `m = mg`

Descriptor Allocation

At the end of the call to `psb_cdall` the descriptor enters into the BUILD state.
Note: we have just specified (implicitly) a mapping between the GLOBAL numbering into a LOCAL numbering (for the local subdomain)

$$I \mapsto (P, J)$$

where

- I is a global index $1 \leq I \leq M$
- P is a process index $0 \leq P < NP$
- J is a local index $1 \leq J \leq NL$

The mapping is complete (On each process P we can now answer whether I belongs here, and we can retrieve the global I corresponding to local J)
BUT
there is no description (yet) of the connections/interactions among subdomains.

Data Management

Second step, we have to describe the mesh topology. This may be done in two ways:

1. Explicitly, with a list of edges;
2. Implicitly, while building a sparse matrix (whose pattern is isomorphic to the graph).

This works as long as the descriptor stays in the BUILD state.

Data Management - C Interface

The procedure with the C interface:

```
for(int i = 0; i < n; i++){
  if ( 'this index belongs to me' ){
    nz = 'number of neighbours of i';
    ia = 'vector of size nz with all values i';
    ja = 'list of the nz neighbours of i';
    info = psb_c_cdins(nz, ia, ja, desc_a);
  }
}
```


Data Management

End of build stage:

```
call psb_cdasb(desc,info)
```

or, in the C interface,

```
info = psb_c_cdasb(desc);
```

The descriptor has now entered the ASSEMBLED state, and may be used for actual data exchanges.
What happened:

- The mapping now identifies local and HALO indices;
- We have built the lists encoding the data exchange patterns.

Sparse Matrix Allocation

In the same way, we allocate a sparse matrix object through:

```
call psb_spall(a,desc_a [, nnz, dupl, bldmode])
```

or, in the C interface,

```
info = psb_c_dspall(a, desc_a);
```

Note:

- The matrix A enters the BUILD state;
- If an estimate *nnz* of the final number of nonzeros (on the current process P) is available, it speeds up the build phase.

Sparse Matrix Allocation

In the same way, we allocate a sparse matrix object through:

```
call psb_spall(a,desc_a [, nnz, dupl, bldmode])
```

or, in the C interface,

```
info = psb_c_dspall(a, desc_a);
```

Note:

- Since version 3.8.0 you can specify `bldmode=psb_matbld_remote_`, i.e. you can track contributions generated on one process, but whose destination is another process;
- The `dupl` argument handles duplicates; since 3.7 the default is `psb_dupl_add_`, consistent with common finite-element practice;

Sparse Matrix Allocation

In the same way, we allocate a sparse matrix object through:

```
call psb_spall(a,desc_a [, nnz, dupl, bldmode])

do i=1, n
 if ( 'this index belongs to me' ) then
  nz = 'number of entries in equation  i'
  ia(1:nz) = i
  ja(1:nz) = 'list of neighbours of i'
  val(1:nz) = 'coefficients Aij'
  call psb_spins(nz,ia,ja,val,a,desc_a,info)
 endif
enddo
```

Note that remote contributions generate an overhead, hence it you are able to generate locally you'll go faster

Sparse Matrix Allocation - C Interface

In the same way, we allocate a sparse matrix object through:

```
info = psb_c_dspall(a, desc_a);

for(int i = 0; i < n; i++){
 if( 'this index belongs to me' ){
  nz = 'number of entries in equation  i'
  ia = 'vector of nz value i'
  ja = 'list of nz neighbours of i'
  val = 'coefficients Aij'
  info = psb_c_dspins(nz,ia,ja,val,a,desc_a);
 }
}
```

The procedures for the other data types are completely analogous.

Data Management

Note: the values contained in IA, JA are (usually) written in terms of the GLOBAL numbering. As we go through $k = 1 : NZ$ on process P:

1. If $IA(k) \notin P$ then $IA(k)$, $JA(k)$ and $VAL(k)$ are ignored (if `psb_matbld_noremote_`) or stashed (if `psb_matbld_remote_`);
2. If $IA(k) \in P$ and $JA(k) \notin P$ then we have a communication requirement that has to be coherent with $DESC$;
3. There actually is no need to process (entire) row by (entire) row; the order may be arbitrary (e.g.: all the coefficients associated with an element, coefficient by coefficient, etc).
4. It is convenient for performance to group a certain amount of data into a single function call;

Data Management

End of build stage:

```
call psb_spasb(a,desc_a,info [, afmt, upd, &
^^I^^I^^I& mold])
```

or, in the C interface,

```
psb_i_t psb_c_dspasb(psb_c_dspmat *a,
^^Ipsb_c_descriptor *desc_a);
```

After this call the sparse matrix enters the ASSEMBLED state.
Notes:

- With either $AFMT$ or $MOLD$ we may specify the desired internal storage format;

Data Management

Same overall code structure with dense vectors

```
call psb_geall(x,desc,info)
do i=1, n
 if ( 'this index belongs to me' ) then
  val = 'i-th term of X '
  call psb_geins(1,(/i/),(/val/),x,desc,info)
 endif
enddo
call psb_geasb(x,desc,info)
```

The equivalent C interface code makes use of

```
psb_i_t psb_c_dgeall(psb_c_dvector *x, psb_c_descriptor *desc);
psb_i_t psb_c_dgeins(psb_i_t nz, const psb_l_t *irw,
  const psb_d_t *val, psb_c_dvector *x,
  psb_c_descriptor *desc);
psb_i_t psb_c_dgeasb(psb_c_dvector *x, psb_c_descriptor *desc);
```


Data Management

Rules of precedence:

- A call to psb_cdall must precede any calls to either psb_spall or psb_geall using the same descriptor
- A call to psb_cdasb must precede any calls to either psb_spasb or psb_geasb using the same descriptor

Note: Most routines in PSBLAS must be called synchronously by all processes participating in a context; these include all the computational, allocation, and assembly routines.
The insertion routines psb_XXins are the main exception, as are called independently; a subsequent call to psb_XXasb is required for synchronization.

dealiiX I/O from File

For *debug* and *testing* purposes it is possible to read and write matrices/vectors to file

Harwell-Boeing the same file can (optionally) contains also the rhs

```
call hb_read(a, iret, iunit, filename, rhs, mtitle),
call hb_write(a, iret, iunit, filename, key , rhs , mtitle)
```

Matrix Market different functions for matrices and vectors

```
call mm_mat_read(a, iret, iunit, filename)
call mm_array_read(rhs, iret, iunit, filename)
call mm_mat_write(a, mtitle, iret, iunit, filename)
call mm_array_write(rhs, iret, iunit, filename)
```

where `iret` is always an integer error code, and `iunit` the Fortran file unit number.

dealiiX Table of Contents

Preconditioned iterations

```
call psb_krylov(methd,a,prec,b,x,&
& eps,desc_a,info &
& [ itmax, iter, err, itrace, &
&   istop, irst] )
```

Mandatory arguments:

- methd "BiCGSTAB" (default), "BICG", "CGS", "RGMRES", "BiCGSTABL", "CG", "FCG";
- a The sparse matrix (local part);
- prec The preconditioner object;
- b The RHS;
- x The initial guess/final result;
- eps The stopping tolerance;
- desc_a The communication descriptor;
- info Error code.

Preconditioned iterations

Optional arguments:

- itmax Maximum number of iterations (default: 1000);
- iter Actual number of iterations on output;
- err Error estimate on output;
- istop Stopping criterion:
  1. Normwise backward error in the infinity norm (default): $\frac{\|r\|}{\|A\|\|x\|+\|b\|} < \epsilon$
  2. 2-Norm relative residual $\frac{\|r\|}{\|b\|} < \epsilon$
- itrace Print the current value of the error estimator every *itrace* > 0 iterations; default -1 (i.e. no message).
- irst Restart parameter for RGMRES (default: 10) and BiCGSTAB(L) (default: 1).

Preconditioned iterations - C Interfaces

The interfaces to the same routines are contained in the `psb_krylov_cbind.h` header, and are available for the complex/real single and double precision types

```
int psb_c_skrylov(const char *method, psb_c_sspmat *ah,
^^Ipsb_c_sprec *ph, psb_c_svector *bh, psb_c_svector *xh,
^^Ipsb_c_descriptor *cdh, psb_c_SolverOptions *opt);
int psb_c_dkrylov(const char *method, psb_c_dspmat *ah,
^^Ipsb_c_dprec *ph,  psb_c_dvector *bh, psb_c_dvector *xh,
^^Ipsb_c_descriptor *cdh, psb_c_SolverOptions *opt);
int psb_c_ckrylov(const char *method, psb_c_cspmat *ah,
^^Ipsb_c_cprec *ph, psb_c_cvector *bh, psb_c_cvector *xh,
^^Ipsb_c_descriptor *cdh, psb_c_SolverOptions *opt);
int psb_c_zkrylov(const char *method, psb_c_zspmat *ah,
^^Ipsb_c_zprec *ph, psb_c_zvector *bh, psb_c_zvector *xh,
^^Ipsb_c_descriptor *cdh, psb_c_SolverOptions *opt);
```


Preconditioned iterations - C Interfaces

The solver options are contained into a structure

```
typedef struct psb_c_solveroptions {
 int iter;       /* On exit how many iterations were performed */
 int itmax;      /* On entry maximum number of iterations   */
 int itrace;     /* On entry print an info message every itrace
                                                  iterations */
 int irst;       /* Restart depth for RGMRES or BiCGSTAB(L) */
 int istop;      /* Stopping criterion: 1:backward error
                                        2: ||r||_2/||b||_2 */
 double eps;     /* Stopping tolerance */
 double err;     /* Convergence indicator on exit */
} psb_c_SolverOptions;
```

that can be initialized to the default values with the routine

```
int psb_c_DefaultSolverOptions(psb_c_SolverOptions *opt);
```

Preconditioners

Simple preconditioners:

```
type(psb_dprec_type) :: prec
call psb_precinit(prec,precname,info)
call psb_precbld(a,desc_a,prec,info)
```

- NOPREC No preconditioning;
- DIAG Scaling by a diagonal $d(i) = 1/a_{ii}$
- BJAC Block Jacobi with factorization $ILU(0)$.

They are available, in the relevant types, as C interfaces in

```
psb_c_dprec* psb_c_new_dprec();
psb_i_t  psb_c_dprecinit(psb_c_ctxt ctxt, psb_c_dprec *ph,
^^Iconst char *ptype);
psb_i_t  psb_c_dprecbld(psb_c_dspmat *ah,
^^I psb_c_descriptor *cdh, psb_c_dprec *ph);
```

all the prototypes can be included from `psb_prec_cbind.h`.

AMG4PSBLAS: Advanced Preconditioners

A package of preconditioners for PSBLAS in PSCToolkit:

- Domain decomposition methods: block-Jacobi, Additive Schwarz;
- Incomplete Factorizations and Approximate Inverses local solvers;
- Algebraic Multigrid, with multiple variants, and various options for the coarse level solvers.

AMG4PSBLAS
Algebraic Multigrid Preconditioners For PSBLAS
Available from `https://psctoolkit.github.io/products/amg4psblas`

version 1.2 to be released by year's end

Scalable (optimal) preconditioners

- $\mu(B^{-1}A) \approx 1$, being independent of n (algorithmic scalability)
- the action of B^{-1} costs as little as possible, the best being $\mathcal{O}(n)$ flops (linear complexity)
- in a massively parallel computer, B^{-1} should be composed of easily applied local actions, (implementation scalability, i.e., parallel execution time increases linearly with n)

Scalable (optimal) preconditioners

- $\mu(B^{-1}A) \approx 1$, being independent of n (algorithmic scalability)
- the action of B^{-1} costs as little as possible, the best being $\mathcal{O}(n)$ flops (linear complexity)
- in a massively parallel computer, B^{-1} should be composed of easily applied local actions, (implementation scalability, i.e., parallel execution time increases linearly with n)

MultiGrid (MG) Preconditioners

show optimal behaviour for many s.p.d. matrices,
e.g., matrices coming from scalar elliptic PDEs

(but optimal preconditioner is not necessarily the fastest preconditioner)

Algebraic MultiGrid (AMG) Methods

AMG (Brandt, McCormick and Ruge, 1984)

Algebraic MultiGrid methods do not explicitly use the problem geometry but rely only on matrix entries to generate coarse-grids by using characterizations of *algebraic smoothness*

Algebraic MultiGrid (AMG) Methods

AMG (Brandt, McCormick and Ruge, 1984)

Algebraic MultiGrid methods do not explicitly use the problem geometry but rely only on matrix entries to generate coarse-grids by using characterizations of *algebraic smoothness*

Key issue in effective AMG for general matrices

error not reduced by the (chosen) smoother are called algebraic smoothness:

$$(Aw)_i = r_i \approx 0 \Longrightarrow w_{i+1} \approx w_i$$

Algebraic MultiGrid (AMG) Methods

AMG (Brandt, McCormick and Ruge, 1984)

Algebraic MultiGrid methods do not explicitly use the problem geometry but rely only on matrix entries to generate coarse-grids by using characterizations of *algebraic smoothness*

Key issue in effective AMG for general matrices

error not reduced by the (chosen) smoother are called algebraic smoothness:

$$(Aw)_i = r_i \approx 0 \Longrightarrow w_{i+1} \approx w_i$$

effective AMG requires that algebraic smoothness is well represented on the coarse grid and well interpolated back $\mathbf{w} = (w_i) \in \mathcal{R}ange(P)$

Algebraic Multigrid Algorithms

Given Matrix $A \in \mathbb{R}^{n \times n}$ SPD

Wanted Iterative method B to precondition the CG method:

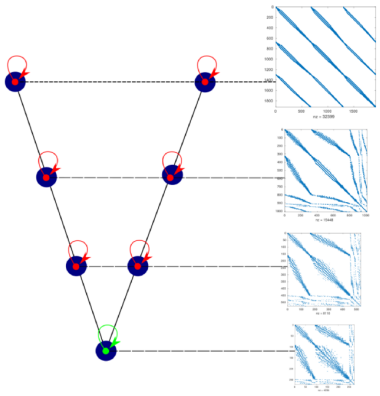
- Hierarchy of systems

$$A_l \mathbf{x}_= \mathbf{b}_l, l = 0, \ldots, \text{nlev}$$

- Transfer operators:

$$P_{l+1}^l : \mathbb{R}^{n_{l+1}} \rightarrow \mathbb{R}^{n_l}$$

Missing Structural/geometric infos



Smoother

$M_l : \mathbb{R}^{n_l} \rightarrow \mathbb{R}^{n_l}$: "High frequencies"

Prolongator

$P_{l+1}^l : \mathbb{R}^{n_l} \rightarrow \mathbb{R}^{n_{l+1}}$: "Low frequencies"

Complementarity of Smoother and Prolongator

Table of Contents

Algebraic MultiGrid (AMG) Setup

Recursive application of a two-grid scheme

- setup of a convergent iterative solver M (the smoother)
- setup of a coarse vector space $\mathcal{R}^{n_c}$ from $\mathcal{R}^n$
- build the prolongation P from A
- compute coarse grid matrix $A_c = P^T AP$

dealii X Algebraic MultiGrid (AMG) Setup

Recursive application of a two-grid scheme

- setup of a convergent iterative solver M (the smoother)
- setup of a coarse vector space $\mathcal{R}^{n_c}$ from $\mathcal{R}^n$
- build the prolongation P from A
- compute coarse grid matrix $A_c = P^T AP$

AMG based on Aggregation of dofs

Group the dofs into disjoint sets of aggregates G_j; each aggregate G_j corresponds to 1 coarse dof
Associated prolongation:

$$P := P_{ij} = \begin{cases} w_i & \text{if } i \in G_j \\ 0 & \text{otherwise} \end{cases}$$

$$i = 1, \ldots, n,\ j = 1, \ldots, n_c,$$

or smoothed version of P (Vaněk 1996)

dealii X Parallel AMG Setup: decoupled aggregation

Given a user-defined threshold ε

Repeat

- Pick a new root point not adjacent to any existing aggregate
- Add neighbours which are strongly connected $\left(\left|a^k_{ij}\right| \geq \varepsilon\sqrt{\left|a^k_{ii}a^k_{jj}\right|}\right)$
- Mark all points adjacent to the aggregate

Until all points are marked

For all leftover points

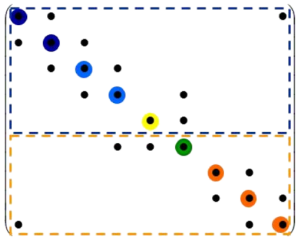
- Add to an aggregated neighbour over threshold; if multiple ones, choose

$$j : \left|a^k_{ij}\right| \geq \left|a^k_{il}\right| \ \forall l$$

- If no neighbour is above threshold, start a new aggregate

Endfor

P. Vaněk, J. Mandel and M. Brezina, Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems, Computing **56** (1996)



- embarrassingly parallel but it may produce non-uniform aggregates
- generally it yields good results in practice on scalar elliptic problems (Tuminaro and Tong, 2000)

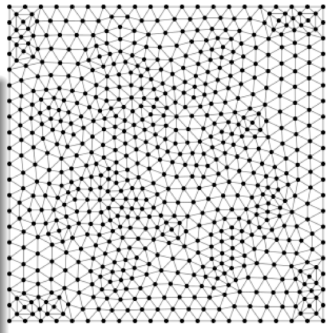
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

Given a graph $G = (\mathcal{V}, \mathcal{E})$ (with adjacency matrix A), and a weight vector $\mathbf{w}$ we consider the weighted version of G obtained by considering the weight matrix $\hat{A}$:

$$(\hat{A})_{i,j} = \hat{a}_{i,j} = 1 - \frac{2a_{i,j}w_i w_j}{a_{i,i}w_i^2 + a_{j,j}w_j^2},$$

- a *matching* $\mathcal{M}$ is a set of pairwise non-adjacent edges, containing no loops;
- a maximum product matching if it maximizes the product of the weights of the edges $e_{i\mapsto j}$ in it.



P. D'Ambra, S. Filippone and P. S. Vassilevski, BootCMatch: a software package for bootstrap AMG based on graph weighted matching, ACM Trans. Math. Software **44** (2018), no. 4, Art. 39, 25 pp.

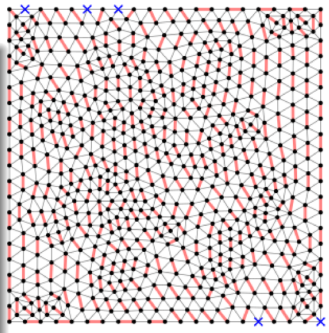
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

Given a graph $G = (\mathcal{V}, \mathcal{E})$ (with adjacency matrix A), and a weight vector $\mathbf{w}$ we consider the weighted version of G obtained by considering the weight matrix $\hat{A}$:

$$(\hat{A})_{i,j} = \hat{a}_{i,j} = 1 - \frac{2a_{i,j}w_i w_j}{a_{i,i}w_i^2 + a_{j,j}w_j^2},$$

- a *matching* $\mathcal{M}$ is a set of pairwise non-adjacent edges, containing no loops;
- a maximum product matching if it maximizes the product of the weights of the edges $e_{i\mapsto j}$ in it.



We divide the index set into matched vertexes $\mathcal{I} = \bigcup_{i=1}^{n_p} \mathcal{G}_i$, with $\mathcal{G}_i \cap \mathcal{G}_j = \emptyset$ if $i \neq j$, and unmatched vertexes, i.e., n_s singletons G_i.

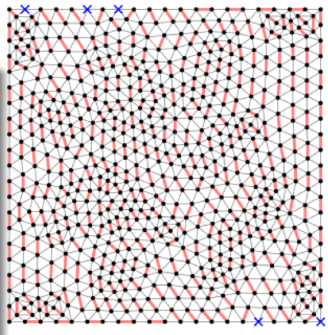
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

Given a graph $G = (\mathcal{V}, \mathcal{E})$ (with adjacency matrix A), and a weight vector $\mathbf{w}$ we consider the weighted version of G obtained by considering the weight matrix $\hat{A}$:

$$(\hat{A})_{i,j} = \hat{a}_{i,j} = 1 - \frac{2a_{i,j}w_i w_j}{a_{i,i}w_i^2 + a_{j,j}w_j^2},$$

- a *matching* $\mathcal{M}$ is a set of pairwise non-adjacent edges, containing no loops;
- a maximum product matching if it maximizes the product of the weights of the edges $e_{i \mapsto j}$ in it.



To increase dimension reduction we can perform more than one sweep of matching per step.

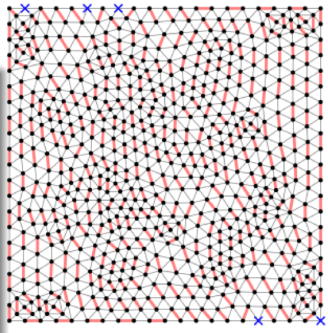
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

Given a graph $G = (\mathcal{V}, \mathcal{E})$ (with adjacency matrix A), and a weight vector $\mathbf{w}$ we consider the weighted version of G obtained by considering the weight matrix $\hat{A}$:

$$(\hat{A})_{i,j} = \hat{a}_{i,j} = 1 - \frac{2a_{i,j}w_i w_j}{a_{i,i}w_i^2 + a_{j,j}w_j^2},$$

- a *matching* $\mathcal{M}$ is a set of pairwise non-adjacent edges, containing no loops;
- a maximum product matching if it maximizes the product of the weights of the edges $e_{i \mapsto j}$ in it.



To increase regularity of P_l we can consider a smoothed prolongator by applying a Jacobi step.

dealiiX Comparison of the Coarsening Strategy

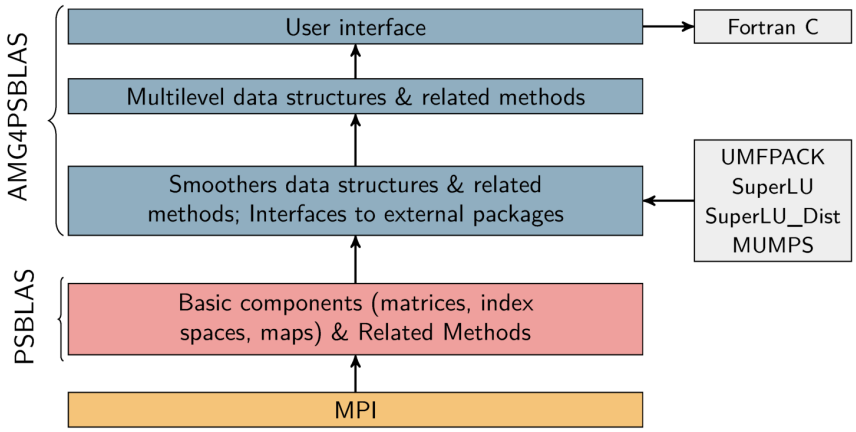
VBM Decoupled aggregation

- ✓ Embarrassingly parallel,
- ✓ Good results with discretized scalar PDEs on a limited number of cores,
- ✗ May produce non-uniform aggregates,
- ✗ Needs user inputted parameters for strength of connection,
- ✗ Issues with anisotropic problems.

Matching-based aggregation

- ✓ Independent of any heuristics or a priori information on the *near kernel* of A,
- ✓ Builds coarse matrices which are well-balanced among parallel processes,
- ✓ No need for special treatment of process-boundary dofs,
- ✓ Works with discretized system of PDEs with arbitrary ordering,
- ✗ May have problems with *highly anisotropic* problems.

dealiiX AMG4PSBLAS Software Architecture



dealiiX Current version of AMG4PSBLAS preconditioners

setup phase: **GPU implementation is work in progress (as far as possible)**

- decoupled smoothed aggregation
- parallel coupled matching-based aggregation
- distributed or replicated coarsest matrix

solve phase: **GPU application implemented**

- cycles: V, W, K
- smoothers: l_1-Jacobi, hybrid (F/B) Gauss-Seidel, Chebychev polynomials, block-Jacobi / additive Schwarz with LU, ILU factorizations or sparse approximate inverses for the blocks
- coarsest-matrix solvers: sparse LU, l_1-Jacobi, hybrid (F/B) Gauss-Seidel, block-Jacobi with LU, ILU factorizations or sparse approximate inverses of the blocks, iterative PCG
- LU factorizations for smoothers & coarsest-level solvers: UMFPACK, MUMPS, SuperLU, SuperLU_Dist

dealiiX User's interface for preconditioner setup

- `p%init(contx,ptype,info)`: allocates and initializes the preconditioner `p`, according to the preconditioner type chosen by the user
- `p%set(what,val,info [,ilev, ilmax, pos, idx])`: sets the parameters defining the preconditioner `p`, i.e., the value contained in `val` is assigned to the parameter identified by `what`
- `p%hierarchy_build(a,desc_a,info)`: builds the hierarchy of matrices and restriction/prolongation operators for the multilevel preconditioner `p`
- `p%smoothers_build(a,desc_a,p,info[,am,vm,im])`: builds the smoothers and the coarsest-level solvers for the multilevel preconditioner `p`
- `p%build(a,desc_a,info[,am,vm,im])`: builds the preconditioner `p` (it is internally implemented by invoking the two previous methods)

dealiiX User's interface for preconditioner apply

- `p%apply(x,y,desc_a,info [,trans,work])`: computes $y = op(B^{-1})\,x$, where B is a previously built preconditioner, stored into `p`, and op denotes the preconditioner itself or its transpose, according to the value of `trans`.
  `p%apply` is called within the PSBLAS method `psb_krylov` and hence it is completely transparent to the user.

- `call p%free(p,info)`: deallocates the preconditioner data structure `p`
- `call p%descr(info, [iout])`: prints a description of the preconditioner `p`

dealiiX Table of Contents

Parameter Setting for Preconditioner Setup

```
...
! build a V-cycle preconditioner with 1
! block-Jacobi sweep (with ILU(0) on the
! blocks) as pre- and post-smoother, and
! 8 block-Jacobi sweeps (with ILU(0)
! on the blocks) as coarsest solver
call P%init('ML',info)
call P%set('SMOOTHER_TYPE','BJAC',info)
call P%set('COARSE_SOLVE','BJAC',info)
call P%set('COARSE_SWEEPS',8,info)
call P%hierarchy_build(A,desc_A,info)
call P%smoothers_build(A,desc_A,info)
...
```


Parameter Setting for Preconditioner Setup (cont'd)

```
...
! build a W-cycle preconditioner with 2
! hybrid Gauss-Seidel sweeps as pre- and
! post-smoother, a distributed coarsest
! matrix, and MUMPS as coarsest-level solver
call P%init('ML',info)
call P%set('ML_CYCLE','WCYCLE',info)
call P%set('SMOOTHER_TYPE','FBGS',info)
call P%set('SMOOTHER_SWEEPS',2,info)
call P%set('COARSE_SOLVE','MUMPS',info)
call P%set('COARSE_MAT','DIST',info)
call P%hierarchy_build(A,desc_A,info)
call P%smoothers_build(A,desc_A,info)
...
```

dealiiX Parameter Setting for Preconditioner Setup (cont'd)

```
...
! build a V-cycle preconditioner with the L1-Jacobi
! variant of a Chebychev Polynomial  of degree 6
call P%init('ML',info)
call P%set('ML_CYCLE','VCYCLE',info)
call P%set('SMOOTHER_TYPE','POLY',info)
call P%set('POLY_DEGREE',6,info)
call P%set('POLY_VARIANT','CHEB_4',info)
call P%set('POLY_RHO_ESTIMATE','POLY_RHO_POWER',info)
call P%set('POLY_RHO_ESTIMATE_ITERATIONS',20,info)
call P%set('SUB_SOLVE','L1-JACOBI',info)
call P%set('COARSE_MAT','DIST',info)
call P%set('COARSE_SOLVE','L1-JACOBI',info)
call P%set('COARSE_SWEEPS',30,info)
call P%hierarchy_build(A,desc_A,info)
call P%smoothers_build(A,desc_A,info)
```


dealiiX How to play around

- If you want to test some of the library capabilities on your problem without jumping in and implementing everything from scratch, then you can use in the `test` directory the examples in the `fileread` folder to try it,

How to play around

- If you want to test some of the library capabilities on your problem without jumping in and implementing everything from scratch, then you can use in the `test` directory the examples in the `fileread` folder to try it,
- The test in `pargen` folder shows how the various part discussed here can be used to solve for a second order equation in 3D with Dirichlet boundary conditions

$$\begin{cases} -\frac{a_1\partial^2 u}{\partial x^2} - \frac{a_2\partial^2 u}{\partial y^2} - \frac{a_3\partial^2 u}{\partial z^2} + b_1\frac{\partial u}{\partial x} + b_2\frac{\partial u}{\partial y} + b_3\frac{\partial u}{\partial z} + cu = f, \\ \qquad \text{for } (x,y,z) \in [0,1]^3, \\ u = g, \\ \qquad \text{for } (x,y,z) \in \partial[0,1]^3. \end{cases}$$

Table of Contents

Interfacing in dealii

The interface in dealii is to a large extent transparent to the user:

- Interfacing from dealii classes (just like PETSc);
- Interfacing through SUNDIALS/Kinsol:
- PSCToolkit can also be interfaced at a "lower" level:
  - Basic operators;
  - Preconditioner application (after setup);

⇒ Therefore usable with the dealii native solvers.

The SUNDIALS/KINSOL Software Framework

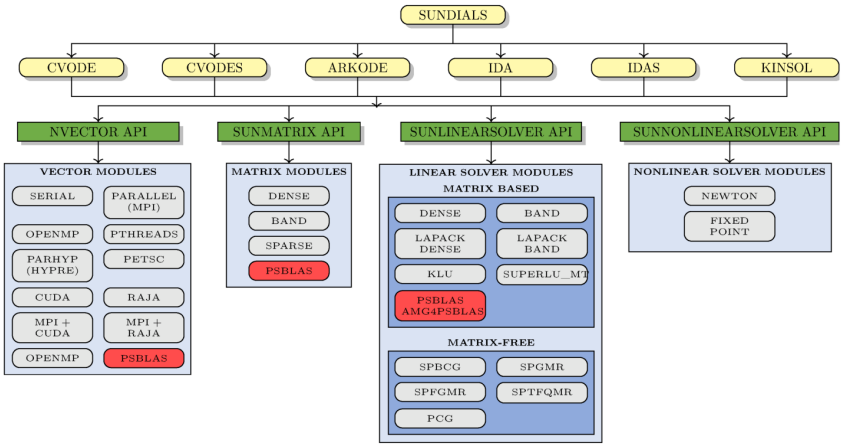


Table of Contents

Weak Scaling on Leonardo

In **weak scaling**, both the **number of computing units** and the **problem size** are **increased**: *constant workload per computing unit*.

We use 8×10^6 unknowns per GPU, *i.e.*, 3.2×10^7 unknowns per node.

We use the following resources:

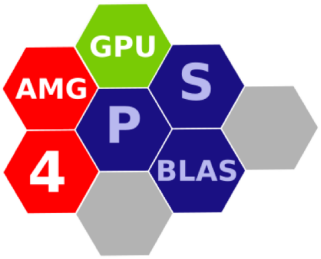
- Number of GPUs from 1 to 8192,
- GPUs x Node 4 (1 MPI Task x GPU, 8 CPUs per Task)
- Pure MPI: 32 MPI Tasks per Node

Within the software framework:

- Compilers: `gcc/11.3.0`
- MPI: `openmpi/4.1.4`
- CUDA `compilation tools, release 11.8, V11.8.89`

dealii-X Algorithms

- **Aggregation:** VBM, **Cycle:** V, **Smoother:** ℓ_1-Jacobi, **Coarse Solver:** PCG + ℓ_1-Jacobi,
- **Aggregation:** Smoothed Matching, **Cycle:** V, **Smoother:** ℓ_1-Jacobi, **Coarse Solver:** PCG + ℓ_1-Jacobi,
- **Aggregation:** Matching, **Cycle:** Variable V, **Smoother:** ℓ_1-Jacobi, **Coarse Solver:** PCG + ℓ_1-Jacobi,
- **Coarsening:** Classical Algebraic Multigrid, **Cycle:** V, **Smoother:** ℓ_1-Jacobi, **Coarse Solver**: ℓ_1-Jacobi, 40 sweeps
- **Aggregation:** (Iterative) Parallel Graph Matching, **Cycle:** V, **Smoother:** ℓ_1-Jacobi, **Coarse Solver**: ℓ_1-Jacobi, 40 sweeps



NVIDIA/**AMGX**

Distributed multigrid linear solver library on GPU

NVIDIA

dealii-X Operator Complexity

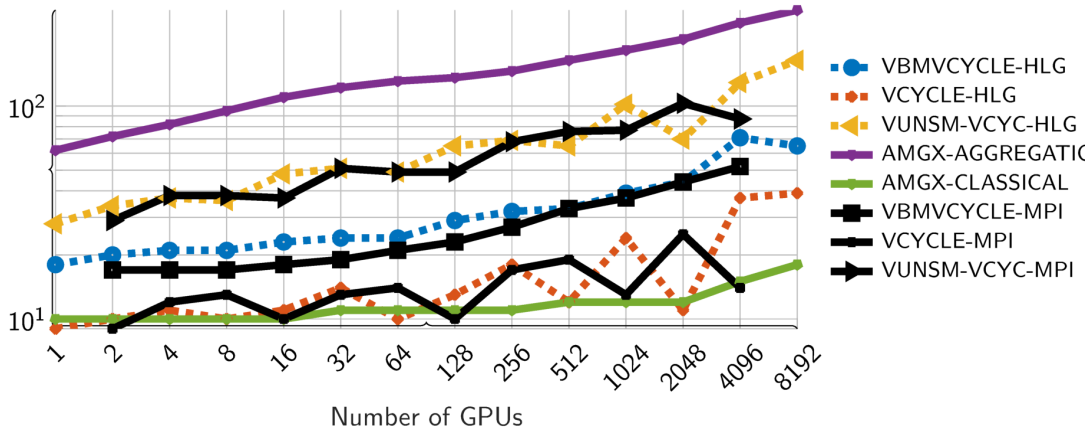
A first measure of the **theoretical computational cost** and of the **memory footprint** of the different algorithms is given by the **operator complexity**:

$$\text{opc} = \frac{\sum_{l=0}^{n_{\text{lev}}} \text{nnz}(A_l)}{\text{nnz}(A)} =$$

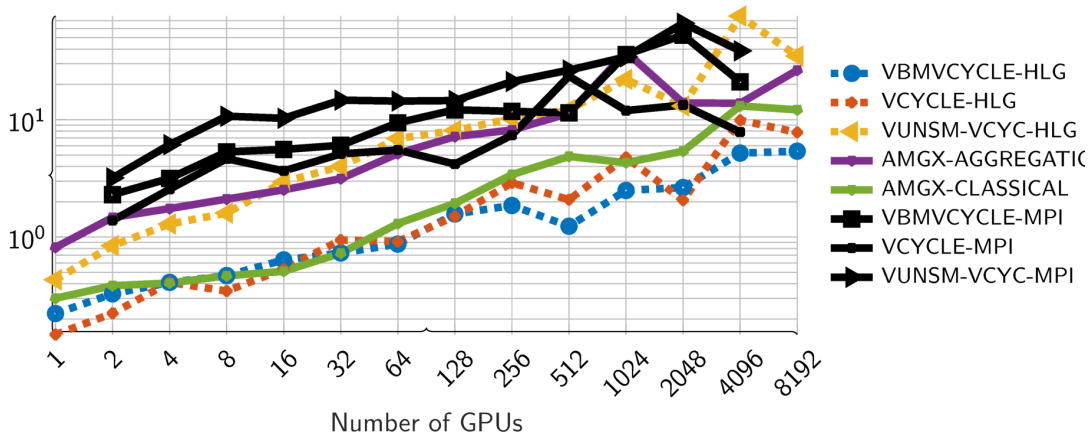
"the total number of nonzeroes in the linear operators on all grids divided by the number of nonzeroes in the fine grid operator"

Computing Units	VBM	Matching Smoothed	Matching Unsmoothed	AMGX Classical	AMGX Matching
32	1,584	1,93	1,143	4,49595	1,31887
64	1,587	1,93	1,143	4,50135	1,31914
128	1,588	1,936	1,143	4,49925	1,31421
256	1,587	1,905	1,144	4,49252	1,31314
512	1,589	1,937	1,143	4,4952	1,31329
1024	1,588	1,942	1,144	4,49503	1,31091

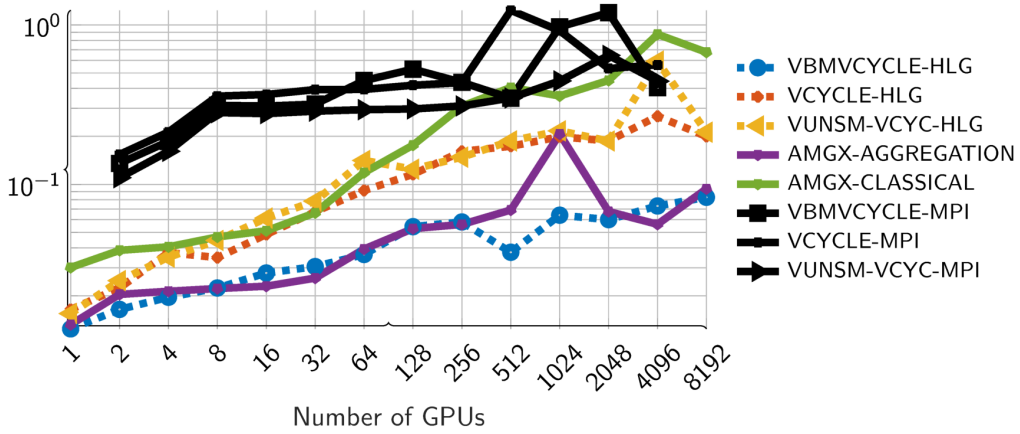
dealiiX Algorithmic Scalability: Iteration Count



dealiiX Implementation Scalability: Solve Time (s)



dealii-X Implementation Scalability: Time × Iteration (s)

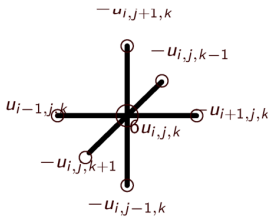


Largest System Size is: 67121414144 $\approx 7 \times 10^{10}$.

dealii-X 3D Poisson Problem

Finite **D**ifferences discretization of

$$\begin{cases} -\nabla^2 u = 1, & \mathbf{x} \in [0,1]^3 \\ u(\mathbf{x}) = 0, & \mathbf{x} \in \partial[0,1]^3. \end{cases}$$



Data distribution:

- For PSCToolkit we use a block 3D Distribution,
- For AMGX we use the `amgx_mpi_poisson7` tester.

Solver is **F**lexible **C**onjugate **G**radient and **CG** for PSCToolkit and AMGX respectively, tolerance 10^{-6}.

dealiiX Weak Scaling on Leonardo

- In **weak scaling**, both the **number of computing units** and the **problem size** are **increased**: *constant workload per computing unit*.
- We use 8×10^6 unknowns per GPU, *i.e.*, 3.2×10^7 unknowns per node.

We use the following resources:

- Number of GPUs from 1 to 8192,
- GPUs x Node 4 (1 MPI Task x GPU, 8 CPUs per Task)
- Pure MPI: 32 MPI Tasks per Node

Within the software framework:

- Compilers: `gcc/11.3.0`
- MPI: `openmpi/4.1.4`
- CUDA `compilation tools, release 11.8, V11.8.89`

dealiiX Test case: Poisson equation

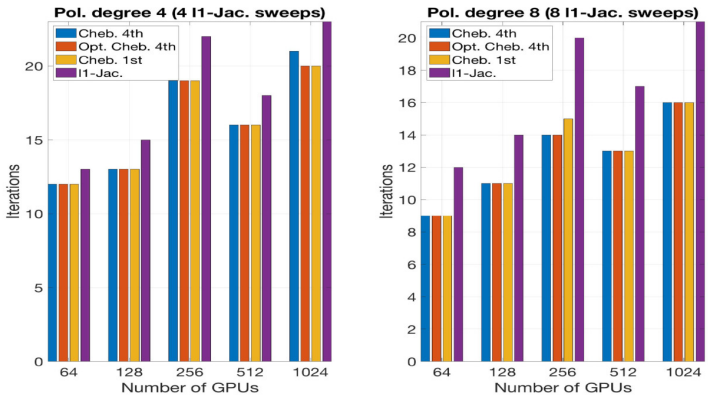
$$-\Delta u = 1 \quad \text{on unit cube, with DBC}$$

Solver/preconditioner settings

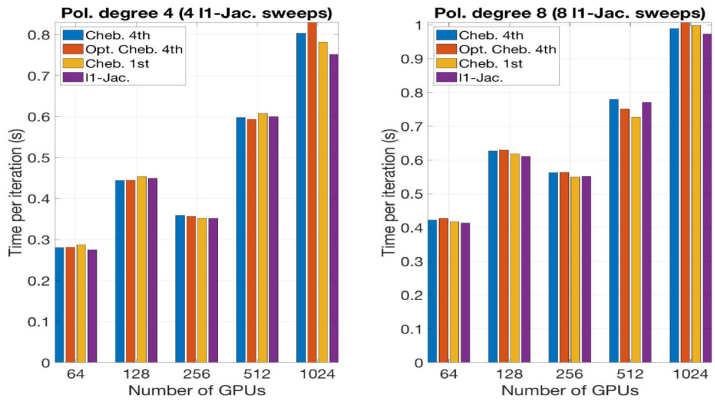
- AMG as preconditioner of CG, stopped when $\|\mathbf{r}^k\|_2/\|\mathbf{b}\|_2 \leq 10^{-7}$, or $itmax = 500$
  VSMATCH V-cycle for matching-based coarsening with aggregates of max size 8, smoothed prolongators
- coarsest matrix size $n_c \leq 200np$, with np number of tasks (GPUs)
- ℓ_1-Jacobi iterations, quasi-opt. 4th-kind Cheb., approximate opt. 4th-kind Chebyshev and quasi opt. 1st-kind Cheb. accelerations; 30 iterations of ℓ_1-Jacobi at the coarsest level.

Platform: Leonardo booster, ranked 6th in the last Top500 list (BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband)

Results:Iterations for Solve



Results:Time per Iteration



Results: Solve Time

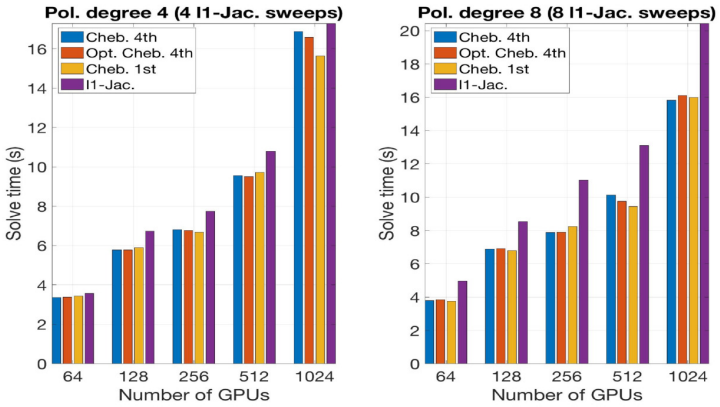


Table of Contents

References I

Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. *ACM Trans. Math. Software*, 23(3):379–401, 1997. ISSN 0098-3500. doi: 10.1145/275323.275327. URL https://doi.org/10.1145/275323.275327.

L. Susan Blackford and et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software*, 28(2):135–151, 2002. ISSN 0098-3500. doi: 10.1145/567806.567807. URL https://doi.org/10.1145/567806.567807.

Salvatore Filippone and Michele Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Software*, 26(4):527–550, 2000.

Salvatore Filippone and Alfredo Buttari. Object-oriented techniques for sparse matrix computations in Fortran 2003. *ACM Trans. Math. Software*, 38(4):23, 2012.

Valeria Cardellini, Salvatore Filippone, and Damian WI Rouson. Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms. *Scientific Programming*, 22(1): 1–19, 2014.

References II

Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw.*, 43(4):30:1–30:49, January 2017. ISSN 0098-3500. doi: 10.1145/3017994. URL http://doi.acm.org/10.1145/3017994.

Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Trans. Math. Software*, 37(3):Art. 30, 23, 2010a. ISSN 0098-3500. doi: 10.1145/1824801.1824808. URL https://doi.org/10.1145/1824801.1824808.

Pasqua DAmbra, Fabio Durastante, and Salvatore Filippone. Parallel sparse computation toolkit. *Software Impacts*, 15:100463, 2023. ISSN 2665-9638. doi: https://doi.org/10.1016/j.simpa.2022.100463. URL https://www.sciencedirect.com/science/article/pii/S2665963822001476.

References III

Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Trans. Math. Software*, 37(3):Art. 30, 23, 2010b. ISSN 0098-3500. doi: 10.1145/1824801.1824808. URL https://doi.org/10.1145/1824801.1824808.

Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. AMG preconditioners for linear solvers towards extreme scale. *SIAM Journal on Scientific Computing*, 43(5):S679–S703, 2021. doi: 10.1137/20M134914X. URL https://doi.org/10.1137/20M134914X.

Pasqua D'Ambra, Fabio Durastante, Salvatore Filippone, and Ludmil Zikatanov. Automatic coarsening in algebraic multigrid utilizing quality measures for matching-based aggregations. *Computers and Mathematics with Applications*, 144:290–305, 2023. ISSN 0898-1221. doi: https://doi.org/10.1016/j.camwa.2023.06.026. URL https://www.sciencedirect.com/science/article/pii/S089812212300278X.

References IV

Daniele Bertaccini, Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. Why diffusion-based preconditioning of richards equation works: Spectral analysis and computational experiments at very large scale. *Numerical Linear Algebra with Applications*, 31(1):e2523, 2024. doi: https://doi.org/10.1002/nla.2523. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.2523.

Pasqua D'Ambra, Fabio Durastante, Salvatore Filippone, Stefano Massei, and Stephen Thomas. Optimal polynomial smoothers for parallel AMG. *Numerical Algorithms*, 100(4): 1783–1812, Dec 2025. ISSN 1572-9265. doi: 10.1007/s11075-025-02117-6. URL https://doi.org/10.1007/s11075-025-02117-6.

Herbert Owen, Oriol Lehmkuhl, Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. Alya toward exascale: algorithmic scalability using psctoolkit. *The Journal of Supercomputing*, 80(10):13533–13556, Jul 2024. ISSN 1573-0484. doi: 10.1007/s11227-024-05989-y. URL https://doi.org/10.1007/s11227-024-05989-y.