The Parallel Sparse Computation Toolkit

Bringing linear algebra to the extreme scale

Seminar of Numerical Mathematics, Univerzita Karlova, Prague, CZ

Fabio Durastante

April 23, 2024

Dipartimento
di Matematica
Università di Pisa

With a little help from my friends...

1 Collaborators and funding

Pasqua **D**'Ambra

IAC-CNR, Italy

Salvatore **F**ilippone

University of Rome "Tor Vergata", Italy

Marco **F**eder

University of Rome "Tor Vergata", Italy

dealii

Exascale Framework for Digital Twins of the Human Body

E·C·E

Fostering the European Energy Transition with Exascale

Table of Contents

2 What, where, and why?

What: solving linear systems at the extreme scale

Target problem:

$$A\mathbf{x} = \mathbf{b}, \qquad A \in \mathbb{R}^{n \times n},\ n \gg 10^9,\ \mathrm{nnz}(A) \in \mathcal{O}(n).$$

Direct factorizations become expensive:

- memory growth due to fill-in,
- poor scalability at extreme size.

Iterative methods are preferred (CG, GMRES, BiCGStab), convergence depends on:

- conditioning of A,
- quality of the preconditioner M^{-1},
- communication costs in parallel environments.

Main objective:

*Design scalable, robust, and communication-aware solvers for **HPC architectures**.*

Where: the machines from the top500 list

	Machine	Cores	Power (kW)	Accelerator
	El Capitan	11,340,000	29,685	AMD Instinct MI300A
	Frontier	9,066,176	24,607	AMD Instinct MI250X
	Aurora	9,264,128	38,698	Intel Data Center GPU Max
	JUPITER Booster	4,801,344	15,794	NVIDIA GH200 Superchip
	Eagle	2,073,600	N/A	NVIDIA H100
	HPC6	3,143,520	8,461	AMD Instinct MI250X
	Fugaku	7,630,848	29,899	None (CPU-only)
	Alps	2,121,600	7,124	NVIDIA GH200 Superchip
	LUMI	2,752,704	7,107	AMD Instinct MI250X
	Leonardo	1,824,768	7,494	NVIDIA A100 SXM4 64 GB

Where: the machines from the top500 list

	Machine	Cores	Power (kW)	Accelerator
	El Capitan	11,340,000	29,685	AMD Instinct MI300A
	Frontier	9,066,176	24,607	AMD Instinct MI250X
	Aurora	9,264,128	38,698	Intel Data Center GPU Max
	JUPITER Booster	4,801,344	15,794	NVIDIA GH200 Superchip
	Eagle	2,073,600	N/A	NVIDIA H100
	HPC6	3,143,520	8,461	AMD Instinct MI250X
	Fugaku	7,630,848	29,899	None (CPU-only)
	Alps	2,121,600	7,124	NVIDIA GH200 Superchip
	LUMI	2,752,704	7,107	AMD Instinct MI250X
	Leonardo	1,824,768	7,494	NVIDIA A100 SXM4 64 GB

Heterogeneous architectures with *millions of cores* and *accelerators*.

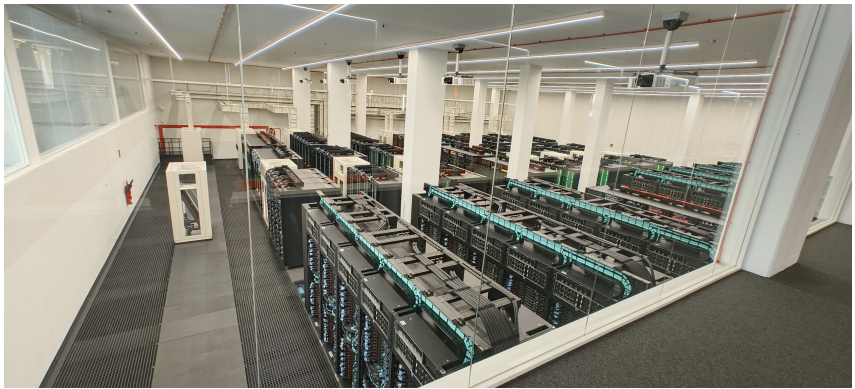
Where: the machines from the top500 list

	Machine	Cores	Power (kW)	Accelerator
	El Capitan	11,340,000	29,685	AMD Instinct MI300A
	Frontier	9,066,176	24,607	AMD Instinct MI250X
	Aurora	9,264,128	38,698	Intel Data Center GPU Max
	JUPITER Booster	4,801,344	15,794	NVIDIA GH200 Superchip
	Eagle	2,073,600	N/A	NVIDIA H100
	HPC6	3,143,520	8,461	AMD Instinct MI250X
	Fugaku	7,630,848	29,899	None (CPU-only)
	Alps	2,121,600	7,124	NVIDIA GH200 Superchip
	LUMI	2,752,704	7,107	AMD Instinct MI250X
	Leonardo	1,824,768	7,494	NVIDIA A100 SXM4 64 GB

Heterogeneous architectures with *millions of cores* and *accelerators*.

We need algorithms that can be implemented efficiently on these platforms.

Where: the machines from the top500 list



Heterogeneous architectures with *millions of cores* and *accelerators*.

We need algorithms that can be implemented efficiently on these platforms.

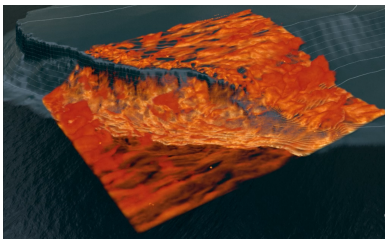
Why?

Why do we need to solve such large linear systems (or such large linear algebra problems)?

First of all because it's a **fun challenge**!

These problems arise in **many applications**:

- Computational Fluid Dynamics for modeling wind and wind turbines,



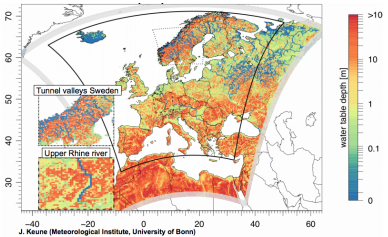
Why?

Why do we need to solve such large linear systems (or such large linear algebra problems)?

First of all because it's a **fun challenge**!

These problems arise in **many applications**:

- Computational Fluid Dynamics for modeling wind and wind turbines,
- Fluid-flow simulation for water in porous media,



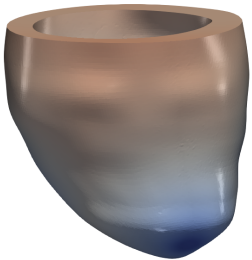
Why?

Why do we need to solve such large linear systems (or such large linear algebra problems)?

First of all because it's a **fun challenge**!

These problems arise in **many applications**:

- Computational Fluid Dynamics for modeling wind and wind turbines,
- Fluid-flow simulation for water in porous media,
- Digital-twins of the human body for personalized medicine,
- ...



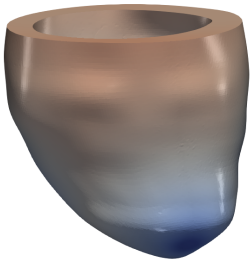
Why?

Why do we need to solve such large linear systems (or such large linear algebra problems)?

First of all because it's a **fun challenge**!

These problems arise in **many applications**:

- Computational Fluid Dynamics for modeling wind and wind turbines,
- Fluid-flow simulation for water in porous media,
- Digital-twins of the human body for personalized medicine,
- …



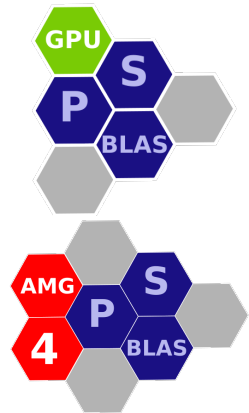
The **large size** comes from accuracy requirements, complex geometries, and the need to capture fine details in simulations.

Table of Contents

Parallel Sparse Computation Toolkit – psctoolkit.github.io

Two central libraries PSBLAS and AMG4PSBLAS:

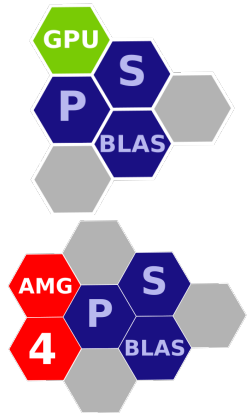
- Existing software standards:
  - MPI, OpenMP, CUDA
  - Serial sparse BLAS,
  - (Par)Metis,
  - AMD
- Attention to performance using **modern Fortran**;
- Research on new preconditioners;
- No need to delve in the data structures for the user;
- Tools for error and mesh handling beyond simple algebraic operations;
- **D**istributed **Sparse BLAS**;
- Standard **K**rylov **s**olvers: CG, FCG, (R)GMRES, BiCGStab, CGS, . . .



Parallel Sparse Computation Toolkit – psctoolkit.github.io

Two central libraries PSBLAS and AMG4PSBLAS:

- Domain decomposition preconditioners
- **A**lgebraic **M**ulti**G**rid with aggregation schemes
  - Vaněk, Mandel, Brezina Aggregation
  - Matching Based
  - Smoothed Aggregation
- Parallel Smoothers (Block-Jacobi, Hybrid-GS/SGS/FBGS, ℓ_1 variants) that can be coupled with specialized block (approximate) solvers MUMPS, SuperLU, Incomplete Factorizations (AINV, INVK/L, ILU-type), and with Polynomial Accelerators (Chebyshev 1st-kind, Chebyshev 4th-kind)
- V-Cycle, W-Cycle, K-Cycle



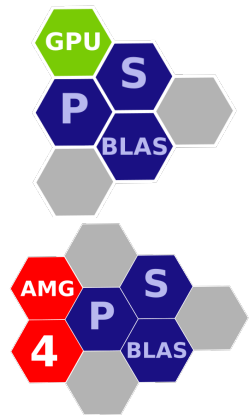
Parallel Sparse Computation Toolkit – psctoolkit.github.io

Two central libraries PSBLAS and AMG4PSBLAS.

- Freely available from: `https://psctoolkit.github.io`,
- Open Source with BSD 3 Clause License,
- Soon to be released/interfaced with the Alya multi-physics solver, and the ParFlow solver, KINSOL non-linear solvers, Deal.II FEM library.

  These are collaborations with:

  Barcelona Supercomputing Center, Centro Nacional de Supercomputación; JÜLICH Forschungszentrum; dealii
- Can be compiled/installed with either *Automake*/*CMake* or *Spack.io*: "`spack install psblas amg4psblas`".



But how does it work?

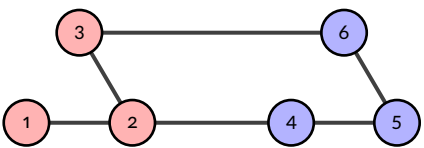
- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,

```
type(psb_ctxt_type) :: ctxt
integer(psb_ipk_) :: iam, np, nth
call psb_init(ctxt)
call psb_info(ctxt,iam,np)
```

But how does it work?

3 The Parallel Sparse Computation Toolkit

- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,
- Build a **partitioned index space**, each **process** has an **arbitrary subset** of the **global index space**:



Build a **descriptor** `type(psb_desc_type):: desc` and init it with global indexes.

On **process 0**:

```
vl = [1,2,3]
call psb_cdall(ctxt, desc, info, vl=vl)
```

On **process 1**:

```
vl = [4,5,6]
call psb_cdall(ctxt, desc, info, vl=vl)
```

- You can do this with any **graph partitioner**: Metis, ParMetis, Zoltan, . . .

But how does it work?

- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,
- Build a **partitioned index space**, each **process** has an **arbitrary subset** of the **global index space**,
- Allocate a **sparse matrix** to be filled with entries computed from your **favorite discretization scheme**

```
type(psb_dspmat_type)  :: a
call psb_spall(a,desc,info,nnz=nnz)
```

Fill the matrix with the entries using *only global indexes* in coordinate format:

```
call psb_spins(num_of_coeffs,irow,icol,val,a,desc,info)
```

The procedure for **vectors** is analogous `psb_geall()`/`psb_geins()`.

But how does it work?

3 The Parallel Sparse Computation Toolkit

- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,
- Build a **partitioned index space**, each **process** has an **arbitrary subset** of the **global index space**,
- Allocate a **sparse matrix** to be filled with entries computed from your **favorite discretization scheme** The procedure for **vectors** is analogous `psb_geall()/psb_geins()`.
- Assemble everything:

```
call psb_cdasb(desc,info)
call psb_spasb(a,desc,info,afmt='CSR') ! or many other formats
```

and you are **ready to perform your solution tasks**.

But how does it work?

3 The Parallel Sparse Computation Toolkit

- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,
- Build a **partitioned index space**, each **process** has an **arbitrary subset** of the **global index space**,
- Allocate a **sparse matrix** to be filled with entries computed from your **favorite discretization scheme** The procedure for **vectors** is analogous `psb_geall()/psb_geins()`.
- Assemble everything:

```
type(psb_d_hlg_sparse_mat) :: gpu_mold
call psb_cdasb(desc,info)
call psb_spasb(a,desc,info,mold=gpu_mold) ! even on the GPU
```

and you are **ready to perform your solution tasks**.

But how does it work?

- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,
- Build a **partitioned index space**, each **process** has an **arbitrary subset** of the **global index space**,
- Allocate a **sparse matrix** to be filled with entries computed from your **favorite discretization scheme** The procedure for **vectors** is analogous `psb_geall()/psb_geins()`.
- Assemble everything:

```
type(psb_d_hlg_sparse_mat) :: gpu_mold
call psb_cdasb(desc,info)
call psb_spasb(a,desc,info,mold=gpu_mold) ! even on the GPU
```

  and you are **ready to perform your solution tasks**.
- Solve a *linear system*, use *Distributed BLAS operations*, *etc.*

Table of Contents

4 Implementing a Krylov method

Implementing a Krylov method

- The whole infrastructure allows to **implement Krylov methods** in a simple way:
  - Write down the implementation in terms of BLAS-like operations,
  - Transform them into the corresponding PSBLAS calls.

Implementing a Krylov method

- The whole infrastructure allows to **implement Krylov methods** in a simple way:
  - Write down the implementation in terms of BLAS-like operations,
  - Transform them into the corresponding PSBLAS calls.

To look at a **few examples**:

Op	BLAS	PSBLAS

Implementing a Krylov method

- The whole infrastructure allows to **implement Krylov methods** in a simple way:
  - Write down the implementation in terms of BLAS-like operations,
  - Transform them into the corresponding PSBLAS calls.

To look at a **few examples**:

Op	BLAS	PSBLAS
$\alpha = \mathbf{x}^\top \mathbf{y}$	`alpha = ddot(n,x,1,y,1)`	`alpha = psb_gedot(x,y,desc_,info)`

Implementing a Krylov method

The whole infrastructure allows to **implement Krylov methods** in a simple way:

- Write down the implementation in terms of BLAS-like operations,
- Transform them into the corresponding PSBLAS calls.

To look at a **few examples**:

Op	BLAS	PSBLAS
$\alpha = \mathbf{x}^\top \mathbf{y}$	`alpha = ddot(n,x,1,y,1)`	`alpha = psb_gedot(x,y,desc␣ ,info)`
$\mathbf{y} = \alpha \boldsymbol{A}\mathbf{x} + \beta\mathbf{y}$	`dgemv('N',n,n,alpha,A,n,x,␣ 1,beta,y,1)`	`psb_spmm(alpha,A,x,beta,y,␣ desc,info)`

Implementing a Krylov method

The whole infrastructure allows to **implement Krylov methods** in a simple way:

- Write down the implementation in terms of BLAS-like operations,
- Transform them into the corresponding PSBLAS calls.

To look at a **few examples**:

Op	BLAS	PSBLAS
$\alpha = \mathbf{x}^\top \mathbf{y}$	`alpha = ddot(n,x,1,y,1)`	`alpha = psb_gedot(x,y,desc␣ ,info)`
$\mathbf{y} = \alpha \boldsymbol{A}\mathbf{x} + \beta\mathbf{y}$	`dgemv('N',n,n,alpha,A,n,x,␣ 1,beta,y,1)`	`psb_spmm(alpha,A,x,beta,y,␣ desc,info)`
$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}$	`y = beta*y` `daxpy(n, alpha, x, 1, ␣ y, 1)`	`psb_geaxpby(alpha,x,beta,y␣ ,desc,info)`

Implementing a Krylov method

The whole infrastructure allows to **implement Krylov methods** in a simple way:

- Write down the implementation in terms of BLAS-like operations,
- Transform them into the corresponding PSBLAS calls.

To look at a **few examples**:

Op	BLAS	PSBLAS
$\alpha = \mathbf{x}^\top \mathbf{y}$	`alpha = ddot(n,x,1,y,1)`	`alpha = psb_gedot(x,y,desc␣ ,info)`
$\mathbf{y} = \alpha \boldsymbol{A}\mathbf{x} + \beta\mathbf{y}$	`dgemv('N',n,n,alpha,A,n,x,␣ 1,beta,y,1)`	`psb_spmm(alpha,A,x,beta,y,␣ desc,info)`
$\mathbf{y} = \alpha\mathbf{x} + \beta\mathbf{y}$	`y = beta*y` `daxpy(n, alpha, x, 1, ␣ y, 1)`	`psb_geaxpby(alpha,x,beta,y␣ ,desc,info)`
$\|\mathbf{x}\|_2$	`dnrm2(n,x,1)`	`psb_genrm2(x,desc,info)`

An example Conjugate Gradient method

4 Implementing a Krylov 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 prec%apply(r,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	`end do iterate`

- Since all operations are library operations these are **offloaded to GPU if the format is right**.
- The library uses a state-pattern design, if you implement a different sparse matrix format, this code remains identical.
- You should implement it also with some **error check** using the content of the `info` variable.

Preconditioners

4 Implementing a Krylov method

To reach convergence Krylov methods also need **preconditioners**, i,e., we want to solve the **preconditioned system**:

$$B^{-1}Ax = B^{-1}b,$$

with matrix $B^{-1} \approx A^{-1}$ (left preconditioner) such that:

Algorithmic scalability $\max_i \lambda_i(B^{-1}A) \approx 1$ being independent of n,

Linear complexity the action of B^{-1} costs as little as possible, the best being $\mathcal{O}(n)$ flops,

Implementation scalability in a massively parallel computer, B^{-1} should be composed of local actions, performance should depend linearly on the number of processors employed.

Algebraic Multigrid Algorithms

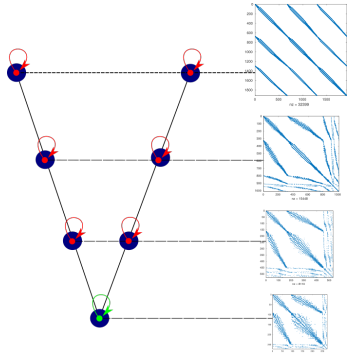
4 Implementing a Krylov method

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

Wanted Iterative method B to precondition the CG/FCG method:

- Hierarchy of systems
$$A_l \mathbf{x}_= \mathbf{b}_l, l = 0, \ldots, \text{nlev}$$
- Transfer operators:
$$P_{l+1}^l : \mathbb{R}^{n_{l+1}} \to \mathbb{R}^{n_l}$$

Missing Structural/geometric infos



Smoother: “High frequencies”

$$M_l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$$

Prolongator: “Low frequencies”

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

Complementarity of Smoother and Prolongator

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is a standard iterative solver with good parallel properties, e.g., ℓ_1–Jacobi, Hybrid-FBGS, Hybrid-SGS, CG method, *etc.*, possibly with a **polynomial accelerator**,

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is a standard iterative solver with good parallel properties, e.g., ℓ_1–Jacobi, Hybrid-FBGS, Hybrid-SGS, CG method, *etc.*, possibly with a **polynomial accelerator**,
- The prolongator P is built by dofs *aggregation based on matching* in the weighted (adjacency) graph of A or by *decoupled* Vaněk, Mandel and Brezina smoothed aggregation.

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is a standard iterative solver with good parallel properties, e.g., ℓ_1–Jacobi, Hybrid-FBGS, Hybrid-SGS, CG method, *etc.*, possibly with a **polynomial accelerator**,
- The prolongator P is built by dofs *aggregation based on matching* in the weighted (adjacency) graph of A or by *decoupled* Vaněk, Mandel and Brezina smoothed aggregation.
- The coarse solver when a large number of processes is used is again a **preconditioned Krylov method**, otherwise a **distributed direct solver** (e.g., MUMPS, SuperLU_dist).

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is an iterative solver with good parallel properties:

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is an iterative solver with good parallel properties:
  - GS $A = M - N$, with $M = L + D$ and $N = -L^T$, where $D = \mathrm{diag}(A)$ and $L = \mathrm{tril}(A)$ is intrinsically sequential!

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is an iterative solver with good parallel properties:
  - GS $A = M - N$, with $M = L + D$ and $N = -L^T$, where $D = \mathrm{diag}(A)$ and $L = \mathrm{tril}(A)$ is intrinsically sequential!
  - HGS Inexact block-Jacobi version of GS, in the portion of the row-block local to each process the method acts as the GS method.

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is an iterative solver with good parallel properties:
  - GS $A = M - N$, with $M = L + D$ and $N = -L^T$, where $D = \mathrm{diag}(A)$ and $L = \mathrm{tril}(A)$ is intrinsically sequential!
  - HGS Inexact block-Jacobi version of GS, in the portion of the row-block local to each process the method acts as the GS method.
  - ℓ_1-HGS On process $p = 1, \ldots, n_p$ relative to the index set Ω_p we factorize $A_{pp} = L_{pp} + D_{pp} + L_{pp}^T$ for $D_{pp} = \mathrm{diag}(A_{pp})$ and $L_{pp} = \mathrm{trilu}(A_{pp})$ and select:

$$
\begin{aligned}
M_{\ell_1-HGS} &= \mathrm{diag}((M_{\ell_1-HGS})_p)_{p=1,\ldots n_p}, \\
(M_{\ell_1-HGS})_p &= L_{pp} + D_{pp} + D_{\ell_1 p}, \\
(d_{\ell_1})_{i=1}^{nb} &= \sum_{j \in \Omega_p^{nb}} |a_{ij}|.
\end{aligned}
$$

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is an iterative solver with good parallel properties:

  GS $A = M - N$, with $M = L + D$ and $N = -L^T$, where $D = \mathrm{diag}(A)$ and $L = \mathrm{tril}(A)$ is intrinsically sequential!

  HGS Inexact block-Jacobi version of GS, in the portion of the row-block local to each process the method acts as the GS method.

  ℓ_1-HGS On process $p = 1, \ldots, n_p$ relative to the index set Ω_p we factorize $A_{pp} = L_{pp} + D_{pp} + L_{pp}^T$ for $D_{pp} = \mathrm{diag}(A_{pp})$ and $L_{pp} = \mathrm{trilu}(A_{pp})$ and select:

$$M_{\ell_1-\mathsf{HGS}} = \mathrm{diag}((M_{\ell_1-\mathsf{HGS}})_p)_{p=1,\ldots n_p},$$

  AINV Block-Jacobi with an approximate inverse factorization on the block $\Rightarrow$ suitable for GPU application!

What is our *recipe*?

4 Implementing a Krylov method

- The smoother M is an iterative solver with good parallel properties:

  GS $A = M - N$, with $M = L + D$ and $N = -L^T$, where $D = \mathrm{diag}(A)$ and $L = \mathrm{tril}(A)$ is intrinsically sequential!

  HGS Inexact block-Jacobi version of GS, in the portion of the row-block local to each process the method acts as the GS method.

  ℓ_1-HGS On process $p = 1, \ldots, n_p$ relative to the index set Ω_p we factorize $A_{pp} = L_{pp} + D_{pp} + L_{pp}^T$ for $D_{pp} = \mathrm{diag}(A_{pp})$ and $L_{pp} = \mathrm{trilu}(A_{pp})$ and select:

$$M_{\ell_1-\mathsf{HGS}} = \mathrm{diag}((M_{\ell_1-\mathsf{HGS}})_p)_{p=1,\ldots n_p},$$

  AINV Block-Jacobi with an approximate inverse factorization on the block $\Rightarrow$ suitable for GPU application!

  POLY Polynomial accelerators, classical and modified polynomial acceleration for stationary iterative methods to accelerate convergence $\Rightarrow$ suitable for GPU application!

What is our *recipe*?

4 Implementing a Krylov method

- The prolongator P is built by dofs *aggregation based on matching* in the weighted (adjacency) graph of A.

What is our *recipe*?

4 Implementing a Krylov method

- The prolongator P is built by dofs *aggregation based on matching* in the weighted (adjacency) graph of A.

Given $\mathbf{w} \in \mathbb{R}^n$, let $P \in \mathbb{R}^{n \times n_c}$ and $P_f \in \mathbb{R}^{n \times n_f}$ be a prolongator and a complementary prolongator, such that:

$$\mathbb{R}^n = \mathrm{Range}(P) \oplus^{\perp} \mathrm{Range}(P_f), \quad n = n_c + n_f$$

$\mathbf{w} \in \mathrm{Range}(P)$: coarse space $\qquad$ $\mathrm{Range}(P_f)$: complementary space

$$[P, P_f]^T A [P, P_f] = \begin{pmatrix} P^T A P & P^T A P_f \\ P_f^T A P & P_f^T A P_f \end{pmatrix} = \begin{pmatrix} A_c & A_{cf} \\ A_{fc} & A_f \end{pmatrix}$$

A_c: coarse matrix $\qquad$ A_f: hierarchical complement

Sufficient condition for efficient coarsening

$A_f = P_f^T A P_f$ as well conditioned as possible, i.e.,

Convergence rate of *compatible relaxation*: $\rho_f = \|I - M_f^{-1} A_f\|_{A_f} \ll 1$

But *how* we achieve it?

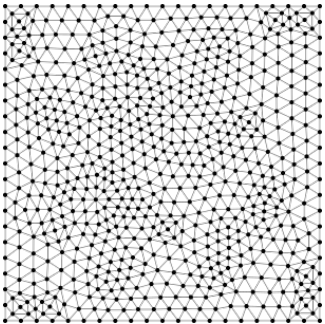
4 Implementing a Krylov method

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.



But *how* we achieve it?

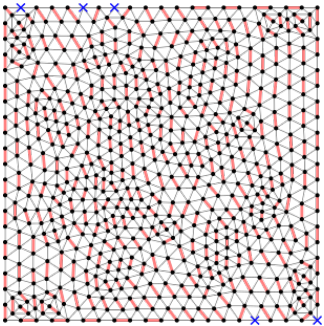
4 Implementing a Krylov method

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 $\mathcal{G}_i$.

From the matching to the prolongator

4 Implementing a Krylov method

We can formally define a *prolongator*:

$$P = \begin{bmatrix} \tilde{P} & O \\ O & W \end{bmatrix} = [\mathbf{p}_1, \ldots, \mathbf{p}_J].$$

Then the preconditioner is the linear operator corresponding to the multiplicative composition of

$$I - B_l A_l = (I - (M_l)^{-T} A_l)(I - P_l B_{l+1} (P_l)^T A_l)(I - M_l^{-1} A_l) \;\; \forall l < nl,$$

where $A_{l+1} = (P_l)^T A_l P_l$ for $l = 0, \ldots, nl - 1$.

From the matching to the prolongator

4 Implementing a Krylov method

We can formally define a *prolongator*:

$$P = \begin{bmatrix} \tilde{P} & O \\ O & W \end{bmatrix} = [\mathbf{p}_1, \ldots, \mathbf{p}_J].$$

Then the preconditioner is the linear operator corresponding to the multiplicative composition of

$$I - B_l A_l = (I - (M_l)^{-T} A_l)(I - P_l B_{l+1} (P_l)^T A_l)(I - M_l^{-1} A_l) \;\; \forall l < nl,$$

where $A_{l+1} = (P_l)^T A_l P_l$ for $l = 0, \ldots, nl - 1$.

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

From the matching to the prolongator

4 Implementing a Krylov method

We can formally define a *prolongator*:

$$P = \begin{bmatrix} \tilde{P} & O \\ O & W \end{bmatrix} = [\mathbf{p}_1, \ldots, \mathbf{p}_J].$$

Then the preconditioner is the linear operator corresponding to the multiplicative composition of

$$I - B_l A_l = (I - (M_l)^{-T} A_l)(I - P_l B_{l+1} (P_l)^T A_l)(I - M_l^{-1} A_l) \;\; \forall l < nl,$$

where $A_{l+1} = (P_l)^T A_l P_l$ for $l = 0, \ldots, nl - 1$.

- To increase dimension reduction we can perform more than one sweep of matching per step,
- To increase regularity of P_l we can consider a smoothed prolongator by applying a Jacobi smoother,

$$P_l^s = (I - \omega D_l^{-1} A_l) P_l, \text{ for } D_l = \mathrm{diag}(A_l).$$

From the matching to the prolongator

4 Implementing a Krylov method

We can formally define a *prolongator*:

$$P = \begin{bmatrix} \tilde{P} & O \\ O & W \end{bmatrix} = [\mathbf{p}_1, \ldots, \mathbf{p}_J].$$

Then the preconditioner is the linear operator corresponding to the multiplicative composition of

$$I - B_l A_l = (I - (M_l)^{-T} A_l)(I - P_l B_{l+1} (P_l)^T A_l)(I - M_l^{-1} A_l) \;\; \forall l < nl,$$

where $A_{l+1} = (P_l)^T A_l P_l$ for $l = 0, \ldots, nl - 1$.

- To increase dimension reduction we can perform more than one sweep of matching per step,
- To increase regularity of P_l we can consider a smoothed prolongator by applying a Jacobi smoother,
- To increase the robustness we can use a non stationary solver as smoother.

From the matching to the prolongator

4 Implementing a Krylov method

We can formally define a *prolongator*:

$$P = \begin{bmatrix} \tilde{P} & O \\ O & W \end{bmatrix} = [\mathbf{p}_1, \ldots, \mathbf{p}_J].$$

Then the preconditioner is the linear operator corresponding to the multiplicative composition of

$$I - B_l A_l = (I - (M_l)^{-T} A_l)(I - P_l B_{l+1} (P_l)^T A_l)(I - M_l^{-1} A_l) \;\; \forall l < nl,$$

where $A_{l+1} = (P_l)^T A_l P_l$ for $l = 0, \ldots, nl - 1$.

- To increase dimension reduction we can perform more than one sweep of matching per step,
- To increase regularity of P_l we can consider a smoothed prolongator by applying a Jacobi smoother,
- To increase the robustness we can use a non stationary solver as smoother.

We employ **distributed half-approximate matching algorithms** to do the construction.

How to use them

4 Implementing a Krylov method

Using these preconditioners is *very simple*!

1. You declare the preconditioner: `type(amg_dprec_type)  :: prec`
2. Initialize it, e.g., as a **Multigrid Preconditioner**:
   `call prec%init(ctxt,'ML',info)`

How to use them

4 Implementing a Krylov method

Using these preconditioners is *very simple*!

1. You declare the preconditioner: `type(amg_dprec_type) :: prec`
2. Initialize it, e.g., as a **Multigrid Preconditioner**:
   `call prec%init(ctxt,'ML',info)`
3. Set all the ingredients you want to use

```
call prec%set('ml_cycle','VCYCLE',info)
call prec%set('outer_sweeps',1,info)
call prec%set('par_aggr_alg','COUPLED',info)
call prec%set('aggr_type','MATCHBOXP',info)
call prec%set('aggr_prol','SMOOTHED',info)
call prec%set('aggr_size',8, info)
```

How to use them

4 Implementing a Krylov method

Using these preconditioners is *very simple*!

1. You declare the preconditioner: `type(amg_dprec_type) :: prec`
2. Initialize it, e.g., as a **Multigrid Preconditioner**:
   `call prec%init(ctxt,'ML',info)`
3. Set all the ingredients you want to use

```
call prec%set('smoother_type','L1-JACOBI',info)
call prec%set('smoother_sweeps',4,info)
call prec%set('coarse_solve','MUMPS',info)
call prec%set('coarse_mat','DIST',info)
```

How to use them

4 Implementing a Krylov method

Using these preconditioners is *very simple*!

1. You declare the preconditioner: `type(amg_dprec_type)  :: prec`
2. Initialize it, e.g., as a **Multigrid Preconditioner**:
   `call prec%init(ctxt,'ML',info)`
3. Set all the ingredients you want to use `call prec%set(...)`
4. Build the **MultiGrid Hierarchy** (aggregation, matching, smoothing, coarse matrices, ...) and **Smoothers** (eventual matrix factorizations)

   ```
   call prec%hierarchy_build(a,desc,info)
   call prec%smoothers_build(a,desc,info)
   ```

How to use them

4 Implementing a Krylov method

Using these preconditioners is *very simple*!

1. You declare the preconditioner: `type(amg_dprec_type) :: prec`
2. Initialize it, e.g., as a **Multigrid Preconditioner**:
   `call prec%init(ctxt,'ML',info)`
3. Set all the ingredients you want to use `call prec%set(...)`
4. Build the **MultiGrid Hierarchy** (aggregation, matching, smoothing, coarse matrices, ...) and **Smoothers** (eventual matrix factorizations)
5. Solve the linear system:
   `call psb_krylov('CG',a,prec,b,x,1.0d-6,desc,info,itmax=30)`

How to use them

4 Implementing a Krylov method

Using these preconditioners is *very simple*!

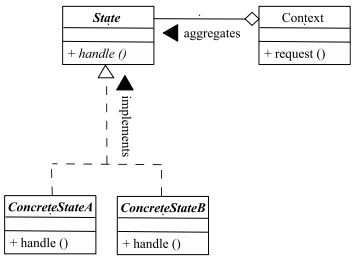
1. You declare the preconditioner: `type(amg_dprec_type) :: prec`
2. Initialize it, e.g., as a **Multigrid Preconditioner**:
   `call prec%init(ctxt,'ML',info)`
3. Set all the ingredients you want to use `call prec%set(...)`
4. Build the **MultiGrid Hierarchy** (aggregation, matching, smoothing, coarse matrices, ...) and **Smoothers** (eventual matrix factorizations)
5. Solve the linear system:
   `call psb_krylov('CG',a,prec,b,x,1.0d-6,desc,info,itmax=30)`

If the **data structures** are GPU data structures everything will **run on the GPU**.

GPU support: the state pattern design

4 Implementing a Krylov method

The GPU support is implemented by a **state pattern design**, i.e., the library contains a **generic interface** for each operation, and the actual implementation is selected at runtime based on the **format** of the data structure.



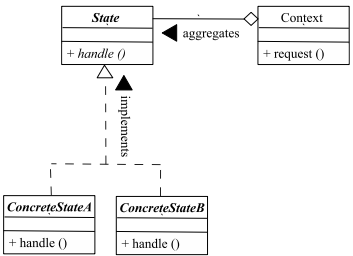
The **mold types** are needed for the sparse matrix, dense arrays, and indexes:

```
type(psb_d_hlg_sparse_mat) :: amold
type(psb_d_vect_cuda) :: vmold
type(psb_i_vect_cuda) :: imold
```

GPU support: the state pattern design

4 Implementing a Krylov method

The GPU support is implemented by a **state pattern design**, i.e., the library contains a **generic interface** for each operation, and the actual implementation is selected at runtime based on the **format** of the data structure.



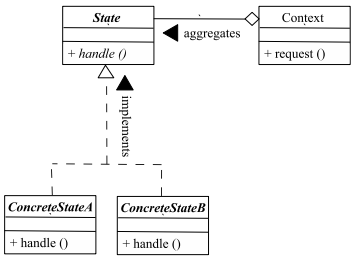
They are used for the **descriptor**, **matrix**, and **vectors**:

```
call psb_cdasb(desc_a,info,mold=imold)
call psb_spasb(a,desc_a,info,mold=amold)
call psb_geasb(x,desc_a,info,mold=vmold)
call psb_geasb(b,desc_a,info,mold=vmold)
```

GPU support: the state pattern design

4 Implementing a Krylov method

The GPU support is implemented by a **state pattern design**, i.e., the library contains a **generic interface** for each operation, and the actual implementation is selected at runtime based on the **format** of the data structure.



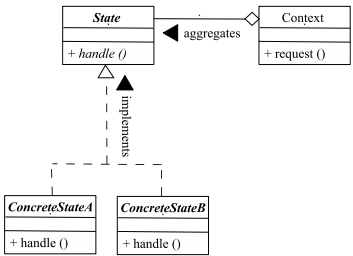
And when **building the preconditioner**:

```
call prec%hierarchy_build(A,desc_A,info)
call prec%smoothers_build(A,desc_A,info,
 ↪  amold=amold, vmold=vmold,
 ↪  imold=imold)
call prec%allocate_wrk(info,vmold=vmold)
```

GPU support: the state pattern design

4 Implementing a Krylov method

The GPU support is implemented by a **state pattern design**, i.e., the library contains a **generic interface** for each operation, and the actual implementation is selected at runtime based on the **format** of the data structure.



And when **building the preconditioner**:

```
call prec%hierarchy_build(A,desc_A,info)
call prec%smoothers_build(A,desc_A,info,
 ↪  amold=amold, vmold=vmold,
 ↪  imold=imold)
call prec%allocate_wrk(info,vmold=vmold)
```

This permits to have a **single code base** for the CPU and GPU implementations, and to **easily add new formats** by just implementing the operations for the new format and adding the corresponding mold type.

Moving from the NVIDIA GPU to a generic GPU

4 Implementing a Krylov method

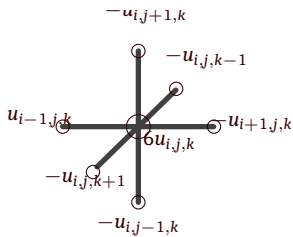
- OpenACC provides a **portable, directive-based** approach to GPU programming,
  - Write once, run on any accelerator: NVIDIA, AMD, Intel, etc.,
  - Pragmatic **incremental parallelization** without rewriting the entire code.
- We are **experimentally supporting** OpenACC in PSBLAS:
  - Target **hybrid CPU-GPU execution** with data movement on demand,
  - Exploit **loop-level parallelism** on GPU while preserving portable code.
- Key implementation aspects:
  - Use `!$acc parallel loop` and related directives for kernels,
  - Leverage **managed memory** or explicit `!$acc data` regions for data movement,
  - Maintain **fallback paths** for CPU execution and non-OpenACC compilers.
- Status: **Experimental** — subject to change based on compiler maturity and performance validation.
- Works by **changing the mold type** to an OpenACC-compatible one.

3D Poisson Problem

4 Implementing a Krylov method

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}.

Weak Scaling

- 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 of the **Leonardo supercomputer** at CINECA:

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

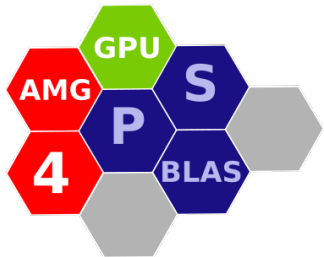
Within the software framework:

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

Algorithms

4 Implementing a Krylov method

- **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

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**:

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

"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
1	1,575	1,894	1,142	4,45456	1,27979
2	1,578	1,905	1,142	4,43576	1,31187
4	1,58	1,915	1,143	4,51377	1,33117
8	1,583	1,917	1,142	4,52376	1,33162
16	1,584	1,925	1,143	4,51239	1,32133

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**:

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

"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

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**:

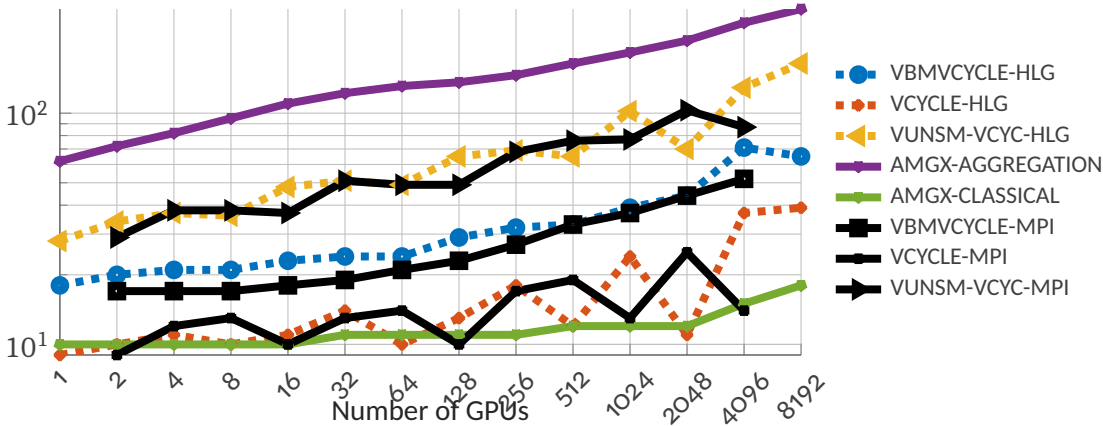
$$\mathrm{opc} = \frac{\sum_{l=0}^{n_{\mathrm{lev}}} \mathrm{nnz}(A_l)}{\mathrm{nnz}(A)} =$$

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

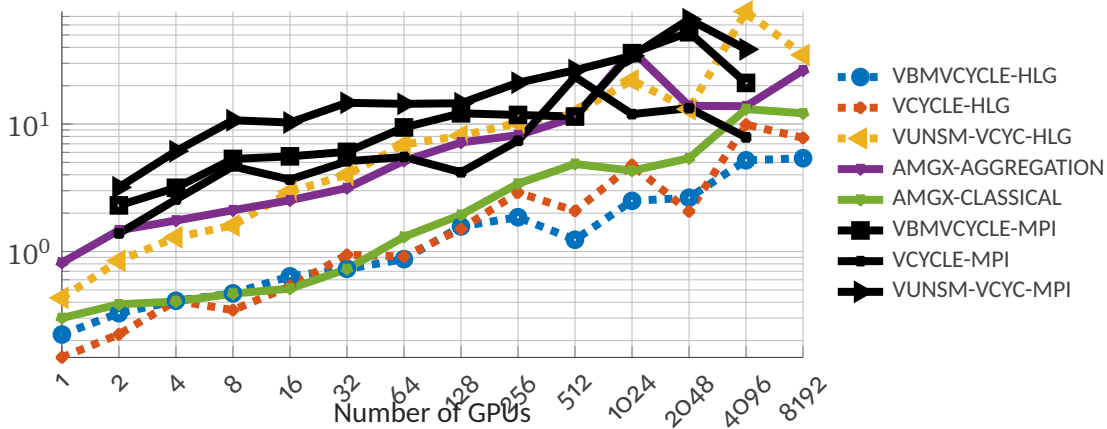
Computing Units	VBM	Matching Smoothed	Matching Unsmoothed	AMGX Classical	AMGX Matching
2048	1,59	1,939	1,143	4,4921	1,31041
4096	1,588	1,906	1,144	4,49354	1,31049
8192			1,144	4,49371	1,30932

Algorithmic Scalability: Iteration Count

4 Implementing a Krylov method

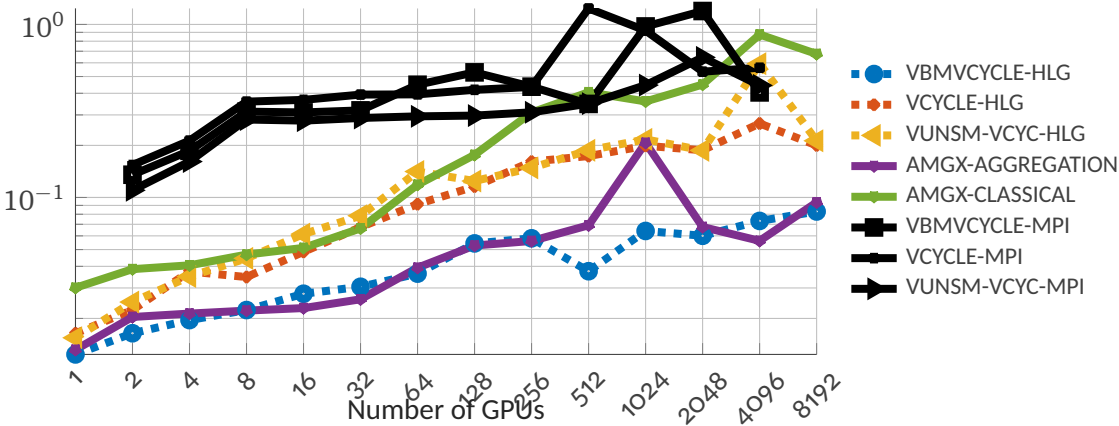


Implementation Scalability: Solve Time (s)



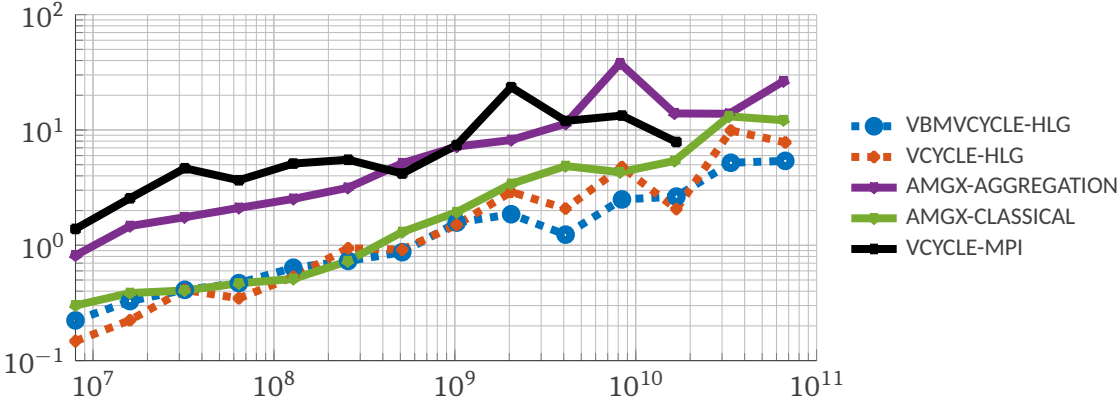
Implementation Scalability: Time $\times$ Iteration (s)

4 Implementing a Krylov method



Best Solve Time vs Global System Size

4 Implementing a Krylov method



Largest System Size is: $67121414144 \approx 6.7 \times 10^{10}$.

Table of Contents

5 The Deal.II interface

The Deal.II library

5 The Deal.II interface

- **deal.II** is an open-source **finite element library** for large-scale PDE simulations.
- Core capabilities:
  - Structured and unstructured meshes in 2D/3D,
  - High-order elements, hp-adaptivity, and *matrix-free operators*,
  - Native support for distributed-memory parallelism (MPI + threads).
- Ecosystem integration:
  - Interfaces to PETSc, Trilinos, and external mesh/IO tools,
  - Suitable for multiphysics and production-grade scientific software.
- Why this matters for us:
  - deal.II provides **robust FE discretizations**,
  - PSBLAS/AMG4PSBLAS provide **scalable linear algebra** and **preconditioning**,
  - Combined workflow: **physics in deal.II**, **solvers in PSCToolkit**.

Integrating PSBLAS and AMG4PSBLAS with deal.II

5 The Deal.II interface

You can compile deal.II with PSBLAS and AMG4PSBLAS support by enabling the corresponding CMake options:

```
cmake /path/to/dealii-source -DDEAL_II_WITH_PSBLAS=ON \
  -DPSBLAS_DIR=/path/to/psblas \
  -DDEAL_II_WITH_AMG4PSBLAS=ON \
  -DAMG4PSBLAS_DIR=/path/to/amg4psblas
```

Then, in your deal.II code, you can **include the headers** to use the library's data structures and functions:

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

System setup with PSBLAS

5 The Deal.II interface

The **descriptor** and **sparse matrix** are initialized from deal.II:

```
PSCToolkit::SparsityPattern sparsity_pattern(locally_owned_dofs,
                                             mpi_communicator);
system_matrix.reinit(sparsity_pattern, mpi_communicator);
```

Key objects in PSBLAS parlance:

- `PSCToolkit::SparseMatrix` — distributed sparse matrix A,
- `PSCToolkit::Vector` — distributed dense vectors $\mathbf{x}$, $\mathbf{b}$,
- `PSCToolkit::SparsityPattern` — encodes the communication pattern for the descriptor.

All objects maintain an internal PSBLAS descriptor, the **distributed index space**, transparent to the user.

deal.II handles **local-to-global indexing**; PSBLAS handles **distributed storage and MPI communication**.

Assembly: constraints and local-to-global

5 The Deal.II interface

- FE assembly computes local element matrices and RHS:

```
constraints.distribute_local_to_global(
  cell_matrix, cell_rhs, local_dof_indices,
  system_matrix, system_rhs);
```

- Constraints (Dirichlet BCs, hanging nodes) are applied during assembly:
  - Global indices are automatically handled by deal.II,
  - PSBLAS matrices accumulate contributions via `compress()`.

- After assembly:

```
// Finalize PSBLAS sparse matrix and RHS vector after assembly
system_matrix.compress(VectorOperation::add);
system_rhs.compress(VectorOperation::add);
```

$\Rightarrow$ Result: a properly **globally consistent** system $A\mathbf{x} = \mathbf{b}$ distributed across MPI ranks.

Solving with AMG4PSBLAS preconditioner

5 The Deal.II interface

Setup the AMG preconditioner via **data structure**:

```
typename PSCToolkit::PreconditionAMG::AdditionalData prec_data;
prec_data.cycle_type = "VCYCLE";
prec_data.smoother_type = "FBGS";
prec_data.aggregation_type = "MATCHBOXP";
prec_data.aggregation_size = 4;
PSCToolkit::PreconditionAMG preconditioner;
preconditioner.initialize(system_matrix, prec_data);
```

Key AMG parameters being set:

- Cycle shape (V-cycle),
- Smoother (Hybrid Forward-Backward Gauss-Seidel),
- Aggregation scheme (Matching-based),
- Coarse solver (MUMPS).

Solving with AMG4PSBLAS preconditioner

5 The Deal.II interface

Solve with CG + preconditioner:

```
SolverCG<PSCToolkit::Vector> solver(solver_parameters);
solver.solve(system_matrix, solution, system_rhs, preconditioner);
```

✓ deal.II CG interface, PSCToolkit backend — all **distributed MPI transparently**.

You can also *use Krylov solvers from PSCToolkit directly* (or if you want to implement something there which is not available in deal.II).

You can ask for **GPU formats** for the **matrix** and **vectors**, since deal.II, is written in C++, this works through the usage of hard-coded format strings.

Preliminary results on the Marenostrum5 supercomputer

5 The Deal.II interface

The machine:

- **MareNostrum 5 ACC** (accelerated partition).
- ♯14 on the Top500 list (November 2025).
- **1120 nodes** based on Intel Sapphire Rapids CPUs and NVIDIA Hopper GPUs.
- Each node is configured with:
  - 2× Intel Sapphire Rapids 8460Y+ @ 2.3 GHz, 40 cores each (**80 cores/node**),
  - **512 GB** DDR5 main memory,
  - 4× NVIDIA Hopper GPUs with **64 GB HBM2** each,
  - **480 GB** NVMe local storage (`/scratch`),
  - 4× NDR200 links (aggregate bandwidth per node: **800 Gb/s**).
- Full-system peak performance: **260 PFLOP/s**.
- Network topology: **fat-tree**, with 160-node non-blocking islands and inter-island contention of **2:1**.

Preliminary results on the Marenostrum5 supercomputer

5 The Deal.II interface

The machine:

- **MareNostrum 5 ACC** (accelerated partition).

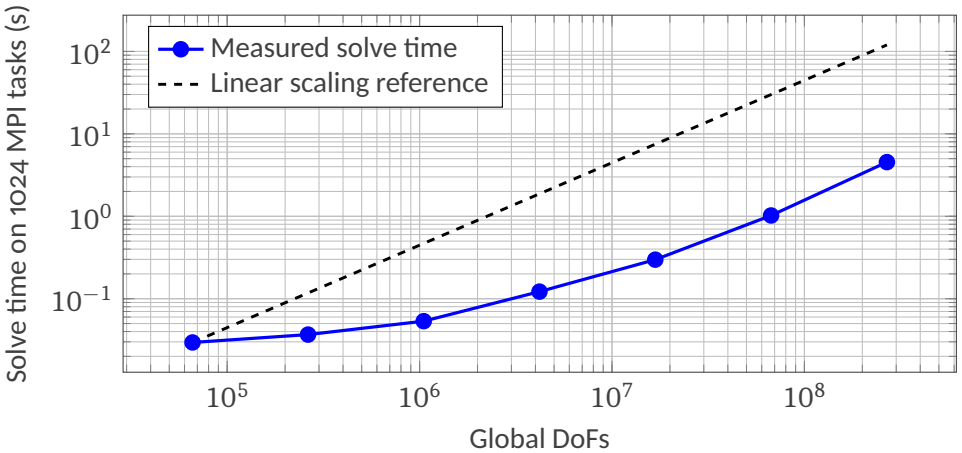
The preconditioner:

- AMG4PSBLAS with **V-cycle**, **FBGS** smoother (sweeps 2),
- Smoothed **matching-based aggregation** with an aggregation size of 8,
- Coarse solver: distributed MUMPS.

The problem:

- 3D Poisson problem on a unit cube, Q_1 FEM on a uniform mesh,
- 2D Poisson problem on a unit square, Q_1 FEM on an adaptively refined mesh.

Example: 3D Poisson problem — uniform refinement



Example: 2D Poisson problem — adaptive refinement

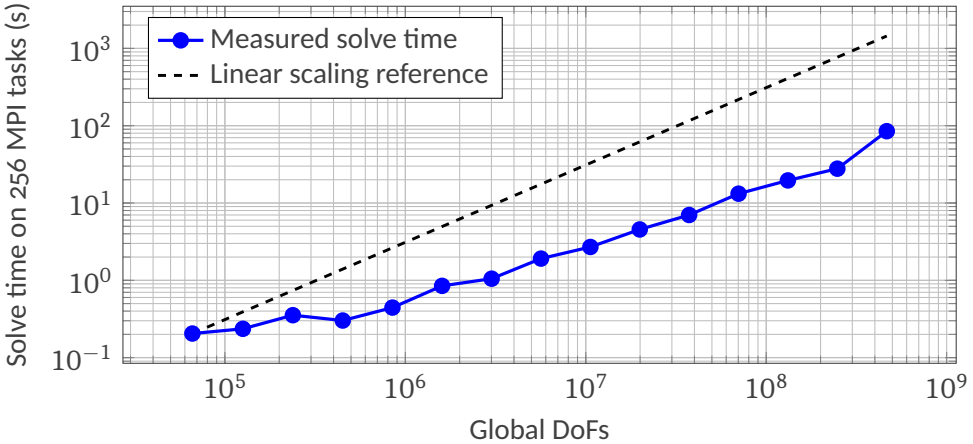


Table of Contents

6 What are working on these days

Conclusions

6 What are working on these days

- PSBLAS and AMG4PSBLAS provide **scalable linear algebra** and **AMG preconditioning**.
- Interface with the deal.II library allows to solve large-scale PDEs with robust FE discretizations.
- Battle-tested on several **European supercomputers** (including *Karolina*).
- If you have in mind linear-algebra algorithms that can be **written in terms of Parallel BLAS and linear solves**, *you* can implement them in PSCToolkit.

</> Getting the code:

```
git clone git@github.com:sfilippone/psblas3.git
git clone git@github.com:sfilippone/amg4psblas.git
```

or

```
spack install psblas amg4psblas
```

Getting these slides:

Scan Me

What are working on these days

6 What are working on these days

We have currently a few **ongoing developments** in PSBLAS and AMG4PSBLAS to further enhance their capabilities and performance on modern architectures:

- **OpenACC support** in PSBLAS and AMG4PSBLAS for portable GPU execution,
- **Matrix-nest support** to support block-matrix structures in PSBLAS,
  - J. Almerol (@ IAC-CNR)
- **Communication-avoiding Krylov methods** to reduce global synchronizations in Krylov basis orthogonalization,
  - I. De Biasi (@ UNIPI)
- **Global-MUMPS** with Block-Low-Rank compression as a scalable smoother inside AMG4PSBLAS
  - P. Amestoy, A. Buttari, B. Rath, T. Mary (MUMPS team)

What are working on these days

My personal to-do-list ("*vaste programme*" – C. De Gaulle) would like to include also:

- **Matrix-equation solvers:** solvers for matrix equations of the form

$$AXB + CXD = FG^\top, \quad A, B, C, D \in \mathbb{R}^{n\times n}, \quad F, G \in \mathbb{R}^{n\times r},\ r \ll, n$$

- **Matrix-function evaluations:** efficient methods to compute $f(A)\mathbf{b}$ for a given function f and matrix A,
- **(Block) Rational Krylov methods:** construction of rational Krylov subspaces for improved convergence in the previous two problems.

Bibliography

9 Bibliography

References for PSBLAS and AMG4PSBLAS:

[2] P. D'Ambra, F. Durastante, and S. Filippone. "Parallel Sparse Computation Toolkit[Formula presented]". In: *Software Impacts* 15 (2023). DOI: 10.1016/j.simpa.2022.100463.

[3] P. D'Ambra, F. Durastante, and S. Filippone. "AMG preconditioners for linear solvers towards extreme scale". In: *SIAM Journal on Scientific Computing* 43.5 (2021), S679–S703. DOI: 10.1137/20M134914X.

[4] P. D'Ambra, F. Durastante, and S. Filippone. "PSCToolkit: Solving Sparse Linear Systems with a Large Number of GPUs". In: *Lecture Notes in Computational Science and Engineering* 146 LNCSE (2025), pp. 271–285. DOI: 10.1007/978-3-031-95709-3_17.

References for the Deal.II library:

[1] D. Arndt, W. Bangerth, M. Bergbauer, B. Blais, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, S. Scheuerman, B. Turcksin, S. Uzunbajakau, D. Wells, and M. Wichrowski. "The deal.II library, version 9.7". In: *Journal of Numerical Mathematics* 33.4 (2025), pp. 403–415. DOI: doi:10.1515/jnma-2025-0115.

Ph.D. in High Performance Scientific Computing @ UniPi

10 Advertisement

- **Interdisciplinary** programme at the University of Pisa, covering:
  - Computational Mathematics & Algorithm Development,
  - HPC Software and Systems,
  - Data Science, AI, Computational Engineering, Life Sciences, …
- Network of **8 university departments**, national research centers, and industry partners.
- Fellowships and projects involve **academic and industrial collaboration**:
  - $\times 2$ funded by S.I.T. – Sordina IORT Technologies S.p.A.,
  - $\times 1$ on the project “Optimal TranspOrt and large Deviations for Deep LEarning theory,”
  - $\times 1$ at the Department of Computer Engineering,
  - $\times 1$ on the project “Open Frameworks for Advanced Finite Element Analysis in Multiphysics and Multiscale Systems”
- **Call opens: May 5, 2026** visit www.dm.unipi.it/phd-hpsc or E-mail me for details.
  - Application deadline: **June 8, 2026**
  - Interviews: **June 22 – July 6, 2026**

The Parallel Sparse Computation Toolkit

Thank you for listening!
Any questions?