Krylov: better, faster, parallel

The 26th Conference of the International Linear Algebra Society

MS23: Advances in Krylov subspace methods and their application

Fabio Durastante (fabio.durastante@unipi.it)

June 24, Tuesday – 17:00–17:30 @ SC4011

Dipartimento
di Matematica
Università di Pisa

Collaborators & Funding

1 With a Little Help from My Friends

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

Salvatore Filippone,
Università degli Studi di Roma "Tor Vergata"
Dipartimento di Ingegneria Civile e
Ingegneria Informatica
IAC-CNR

dealii X
HORIZON-EUROHPC-JU-2023-COE-03-01
Agreement ID: 101172493

E·CoE
HORIZON-EUROPEHPC-JU-2023-COE-01
Agreement N.101144014

INdAM
PASTRAMI - sPline And Solver innovaTions foR
Adaptive isogeoMetric analysIs

Table of Contents

2 Large-Scale Numerical Linear Algebra

Large-Scale Numerical Linear Algebra

PDE discretization yields extremely large sparse linear systems that are central to scientific simulation, hence we target $10^9 \sim 10^{12}$ dofs.

- **Scalability:** Solvers must handle millions–billions of unknowns; algorithms need to scale efficiently on HPC architectures (multi-core CPUs, GPUs, clusters), i.e., thousands/hundred of thousands computing units
- **Computational Cost:** Direct solvers have prohibitive time/memory costs at large scales; iterative methods (***Krylov***, multigrid, domain decomposition) are used to reduce cost and exploit sparsity; robust preconditioners and error-control techniques are needed to ensure convergence and accuracy.
- **Parallelism:** Efficient parallel implementations demand managing communication, load balancing, and *heterogeneous resources* (MPI, hybrid CPU/GPU).

The TOP500 and EuroHPC machines

2 Large-Scale Numerical Linear Algebra

To solve such **large problems** we need to employ machines from the **TOP500 list**

	System Description	Cores	Power (kW)
1	El Capitan - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8 GHz, AMD Instinct MI300A, Slingshot-11, TOSS	11,039,616	29,581
2	Frontier - HPE Cray EX235a, AMD Optimized 3rd Gen EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS	9,066,176	24,607
3	Aurora - HPE Cray EX – Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4 GHz, Intel Data Center GPU Max, Slingshot-11	9,264,128	38,698
	⋮		
9	LUMI - HPE Cray EX235a, AMD Optimized 3rd Gen EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot-11	2,752,704	7,107
10	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6 GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband	1,824,768	7,494

The TOP500 and EuroHPC machines

2 Large-Scale Numerical Linear Algebra

To solve such **large problems** we need to employ machines from the **TOP500 list**

	System Description	Cores	Power (kW)
1	El Capitan - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8 GHz, AMD Instinct MI300A, Slingshot-11, TOSS	11,039,616	29,581
2	Frontier - HPE Cray EX235a, AMD Optimized 3rd Gen EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS	9,066,176	24,607
3	Aurora - HPE Cray EX – Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4 GHz, Intel Data Center GPU Max, Slingshot-11	9,264,128	38,698
	⋮		
9	LUMI - HPE Cray EX235a, AMD Optimized 3rd Gen EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot-11	2,752,704	7,107
10	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6 GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband	1,824,768	7,494

The TOP500 and EuroHPC machines

To solve such **large problems** we need to employ machines from the **TOP500 list**, and for **EU researchers** the ones that are accessible through the **EuroHPC consortium**.

	System Description	Cores	Power (kW)
4	JUPITER Booster - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200	4,801,344	13,088
9	LUMI - HPE Cray EX235a, AMD Optimized 3rd Gen EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot-11	2,752,704	7,107
10	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6 GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband	1,824,768	7,494
14	MareNostrum 5 ACC - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR	663,040	4,158.90
45	MareNostrum 5 GPP - ThinkSystem SD650 v3, Xeon Platinum 8480+ 56C 2GHz, Infiniband NDR200	725,760	5,752.90

The TOP500 and EuroHPC machines

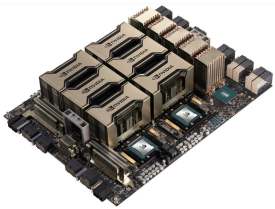
2 Large-Scale Numerical Linear Algebra

To solve such **large problems** we need to employ machines from the **TOP500 list**, and for **EU researchers** the ones that are accessible through the **EuroHPC consortium**.

	System Description	Cores	Power (kW)
136	MeluXina - Accelerator Module - BullSequana XH2000, AMD EPYC 7452 32C 2.35GHz, NVIDIA A100 40GB, Mellanox HDR InfiniBand	99,200	N/A
195	Karolina, GPU partition - Apollo 6500, AMD EPYC 7452 32C 2.35GHz, NVIDIA A100 SXM4 40 GB, Infiniband HDR200	64,960	297.26
258	Discoverer - BullSequana XH2000, AMD EPYC 7H12 64C 2.6GHz, Mellanox HDR InfiniBand	144,384	N/A
259	JEDI - BullSequana XH3000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, ParTec/EVIDEN	19,584	67
297	Deucalion - PRIMEHPC FX700, Fujitsu A64FX 48C 2GHz, InfiniBand HDR100	78,336	365.21
305	VEGA HPC CPU - BullSequana XH2000, AMD EPYC 7H12 64C 2.6GHz, Mellanox InfiniBand HDR100	122,880	N/A

The TOP500 and EuroHPC machines

To solve such **large problems** we need to employ machines from the **TOP500 list**, and for **EU researchers** the ones that are accessible through the **EuroHPC consortium**.



The Accelerators

The Leonardo Machine

MPI

OpenACC OpenMP

More Science, Less Programming

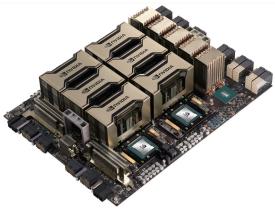
SYCL™ NVIDIA CUDA

The Programming Tools

The TOP500 and EuroHPC machines

2 Large-Scale Numerical Linear Algebra

To solve such **large problems** we need to employ machines from the **TOP500 list**, and for **EU researchers** the ones that are accessible through the **EuroHPC consortium**.



The Accelerators

The Leonardo Machine

MPI

OpenACC OpenMP

More Science, Less Programming

SYCL™ NVIDIA CUDA

The Programming Tools

(Probably) you want to **focus more on the problem** you wish to solve and on the **algorithmic aspects**.

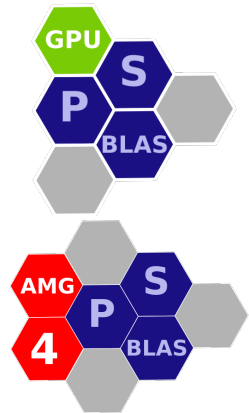
Table of Contents

3 The Parallel Sparse Computation Toolkit

Parallel Sparse Computation Toolkit – psctoolkit.github.io

Two central libraries PSBLAS and AMG4PSBLAS:

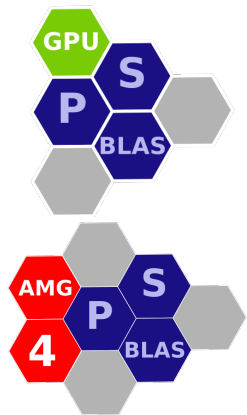
- Existing software standards:
  - MPI, OpenMP, CUDA
  - Serial sparse BLAS,
  - (Par)Metis,
  - AMD
- Attention to performance using **modern Fortran**;
- Research on new preconditioners;
- No need to delve in the data structures for the user;
- Tools for error and mesh handling beyond simple algebraic 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



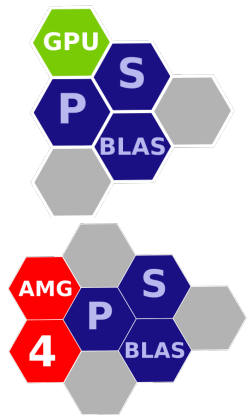
Parallel Sparse Computation Toolkit – psctoolkit.github.io

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`”.



But how does it work?

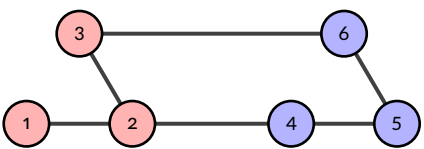
- You start a **parallel environment**—if you are familiar with MPI, an MPI *communicator*,

```
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?

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**

```
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?

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**.
- 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

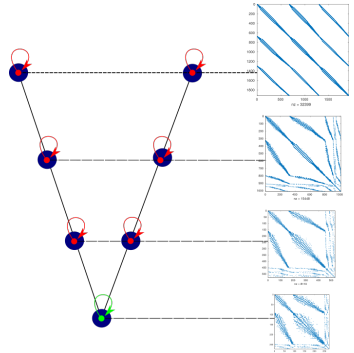
4 Implementing a Krylov method

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?

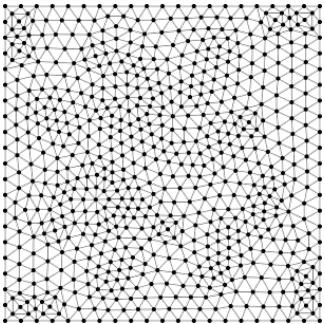
4 Implementing a Krylov method

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?

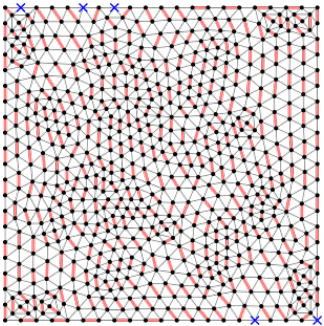
4 Implementing a Krylov method

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)`

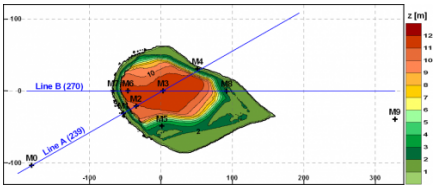
💡 If the **data structures** are GPU data structures everything will **run on the GPU**.

Table of Contents

5 An example at scale

Large Eddy Simulation: Wind Simulation

5 An example at scale



Bolund is an isolated hill situated in Roskilde Fjord, Denmark. An almost vertical escarpment in the prevailing W-SW sector ensures flow separation in the windward edge resulting in a complex flow field.

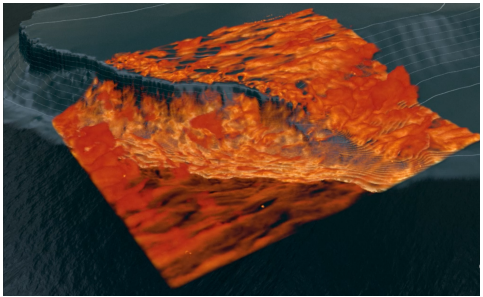
- **Model**: 3D incompressible unsteady Navier-Stokes equations for the Large Eddy Simulations of turbulent flows,
- **Discretization**: low-dissipation *mixed FEM* (linear FEM both for velocity and pressure) on an *hybrid unstructured meshes*, which can include tetrahedra, prisms, hexahedra, and pyramids,
- **Time-Stepping**: non-incremental fractional-step for pressure, explicit fourth order Runge-Kutta method for velocity.

Full details are available in the paper:

Owen, H., Lehmkuhl, O., D'Ambra, P., D., F., & Filippone, S. (2024). *Alya toward exascale: algorithmic scalability using PSCToolkit*. J. Supercomput., 80(10), 13533–13556.

Large Eddy Simulation: Wind Simulation

5 An example at scale



An **example of solution** we obtain with this configuration.

- **Model**: 3D incompressible unsteady Navier-Stokes equations for the Large Eddy Simulations of turbulent flows,
- **Discretization**: low-dissipation *mixed FEM* (linear FEM both for velocity and pressure) on an *hybrid unstructured meshes*, which can include tetrahedra, prisms, hexahedra, and pyramids,
- **Time-Stepping**: non-incremental fractional-step for pressure, explicit fourth order Runge-Kutta method for velocity.

Full details are available in the paper:

Owen, H., Lehmkuhl, O., D'Ambra, P., D., F., & Filippone, S. (2024). *Alya toward exascale: algorithmic scalability using PSCToolkit*. J. Supercomput., 80(10), 13533–13556.

Preconditioner and solver setup

5 An example at scale

Pre-smoother	4 iterations of **h**ybrid **f**orward **G**auss–**S**eidel		
Post-smoother	4 iterations of **h**ybrid **b**ackward **G**auss–**S**eidel		
Coarsest solver	**FCG** preconditioned by block-Jacobi with ILU(1) block solvers		
Cycle	V-cycle		
Aggregation	Coupled smoothed based on matching		Decoupled classic smoothed
	$\lvert\mathcal{G}\rvert \leq 8$	$\lvert\mathcal{G}\rvert \leq 16$	
Label	*MLVSMATCH3*	*MLVSMATCH4*	*MLVSBM*

- FCG with $\varepsilon = 10^{-3}$,
- Initial guess for pressure from the previous time step,
- Reynolds Number: $RE_\tau = Uh/\nu \approx 10^7$ with $U = 10\,\mathrm{m\,s^{-1}}$,
- Rossby number $R_O = 667 \gg 1$ (i.e., no Coriolis force in the horizontal direction).

Strong scaling results

5 An example at scale

We consider **strong scaling** performance on **three grids**:

Small $n_1 = 5570786 \approx 6 \times 10^6$ dofs with $\min_p = 48$ to $\max_p = 192$ cores

Medium $n_2 = 43619693 \approx 4.4 \times 10^7$ dofs with $\min_p = 384$ to $\max_p = 1536$ cores

Large $n_3 = 345276325 \approx 0.35 \times 10^9$ dofs with $\min_p = 3072$ to $\max_p = 12288$ cores

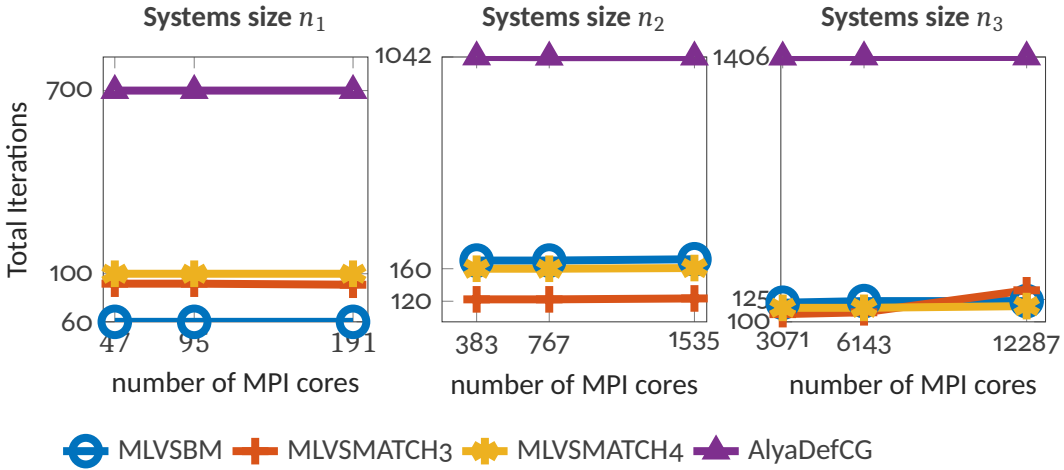
Run where performed on the **Marenostrum-4 supercomputer** (machine with 3456 nodes with 2 Intel Xeon Platinum 8160 CPUs with 24 cores per CPU) now superseded by the **Marenostrum-5 supercomputer**.

Strong Scaling

In case of **strong scaling**, *the number of processors is increased while the problem size remains constant*. This also results in a reduced workload per processor. Strong scaling is mostly used for long-running CPU-bound applications to find a setup which results in a reasonable runtime with moderate resource costs.

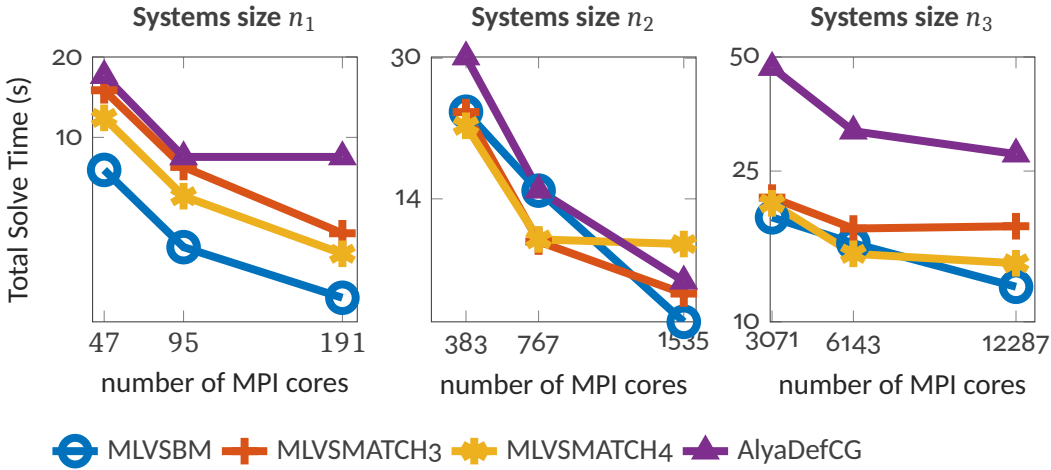
Strong scaling results: iteration count

5 An example at scale



Strong scaling results: total solve time

5 An example at scale



Weak scaling results

5 An example at scale

We consider the **same three meshes** plus a **fourth one** with $n_4 \approx 2.9 \times 10^9$

- The **number of dofs per core** we consider is: $\text{nxcore}_1 = 1.1 \times 10^5$ dofs.
- We run at 45, 367, 2943, **23551 cores** with the four meshes.

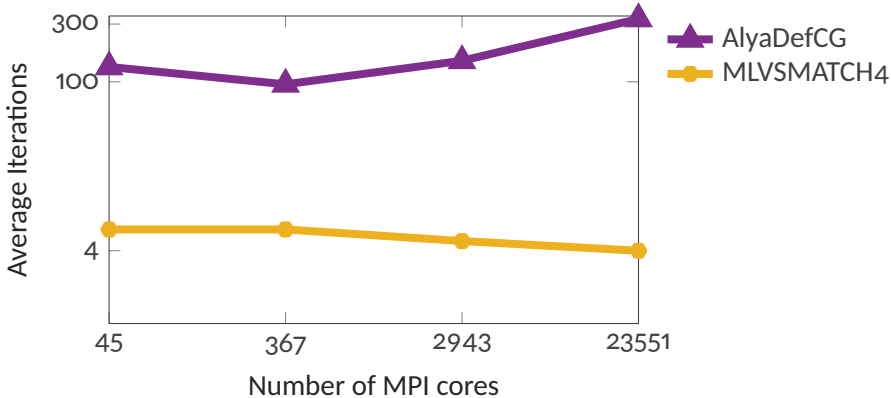
Results are obtained on the **Juwels supercomputer** (2271 compute nodes with 2 Intel Xeon Platinum 8168 CPUs, of 24 cores each)

Weak scaling

In case of **weak scaling**, *both the number of processors and the problem size are increased*. This also results in a constant workload per processor. Weak scaling is mostly used for large memory-bound applications where the required memory cannot be satisfied by a single node.

Weak scaling: average iterations

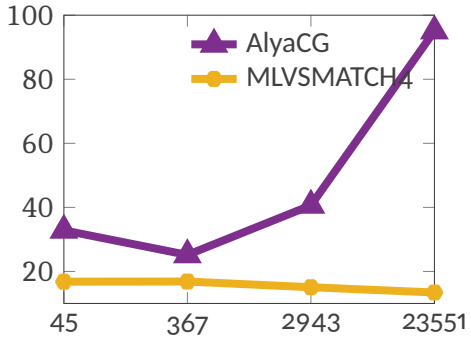
5 An example at scale



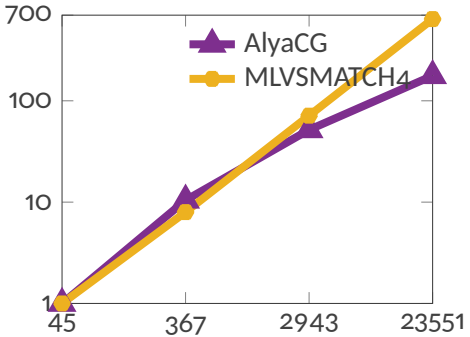
⚠ To **preserve** computational time, we **run** only **20 time steps** and **average the measures**.

Weak scaling: total solve time and speedup

5 An example at scale



Total solve time



Speedup

Table of Contents

Where we would like to go

6 The way forward

We started investigations on

- Distributed implementation of **S**ketched-**GMRES**,
- **C**ommunication-**A**voiding **K**rylov methods.

Always on the lookout for collaborations in

- Block-Krylov methods: $AX = B$;
- Linear matrix-equations: $AX + XB = UV^\top$;
- Matrix-function vector products: $\mathbf{y} = f(A)\mathbf{v}$.

Krylov: better, faster, parallel

Thank you for listening!

Any questions?

3D Poisson benchmark - solvers

7 GPU Example

Solvers from PSCToolkit

VBM decoupled Vaněk, Mandel, Brezina aggregation, V-cycle, ℓ_1-Jacobi smoother (4 sweeps), at most 40 iterations of the Preconditioned CG coupled to ℓ_1-Jacobi preconditioner as coarsest solver;

SMATCH matching-based aggregation with aggregates of maximum size equal to 8, smoothing of prolongators, further algorithmic choices as in VBM;

VMATCH matching-based aggregation as in SMATCH, un-smoothed prolongators, Variable V-cycle[1], further algorithmic choices as in VBM.

[1] 2 smoother iteration at the first level, and doubled at each following level.

3D Poisson benchmark - solvers

7 GPU Example

Solvers from NVIDIA

AMGX CLASSICAL coarsening done by classical, also known as Ruge-Stüben, AMG approach, where the coarse nodes are a subset of the fine nodes and a distance-2 interpolation is applied, V-cycle, smoother ℓ_1-Jacobi (4 sweeps), ℓ_1-Jacobi (40 sweeps) coarsest solver;

AMGX AGGREGATION aggregation by iterative parallel graph matching, aggregates of maximum size equal to 8, further algorithmic choices as in AMGX CLASSICAL.

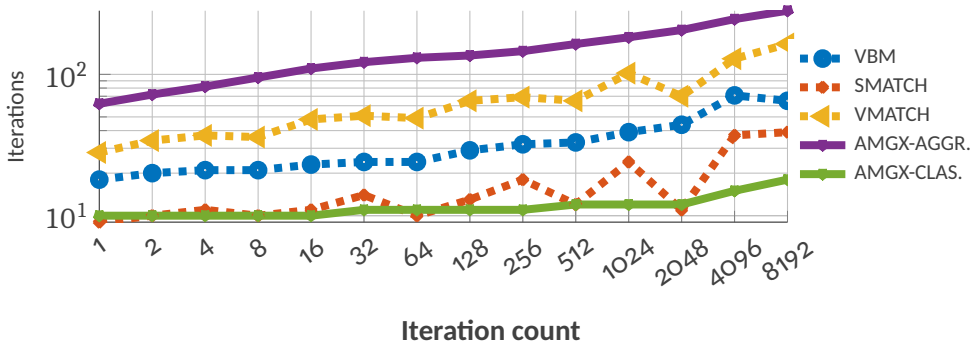
Solver details:

- FCG with relative tolerance on the residual of $\tau = 10^{-6}$,
- Weak-scaling with 8×10^6 unknowns per GPU, *i.e.*, 3.2×10^7 unknowns per node

The **largest system** we consider has $\approx 6.5 \times 10^{10}$ degrees of freedom.

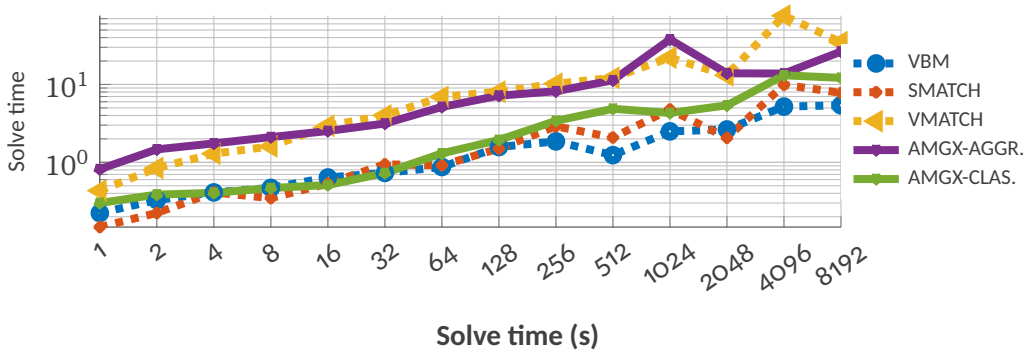
3D Poisson benchmark - Comparison with AMGx

7 GPU Example



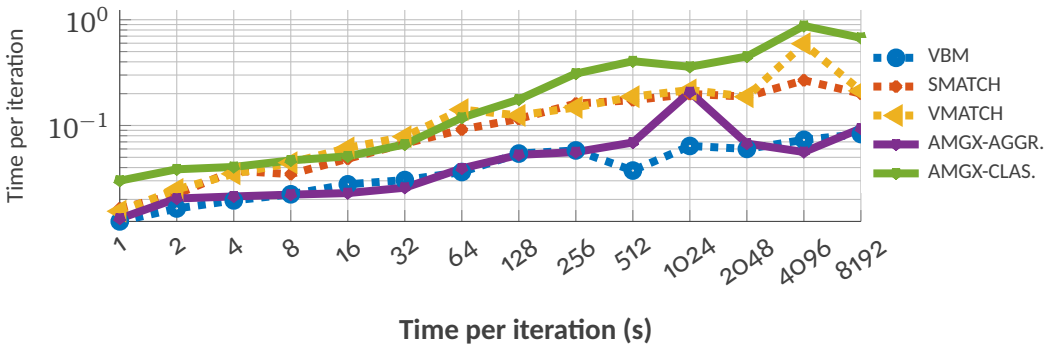
3D Poisson benchmark - Comparison with AMGx

7 GPU Example



3D Poisson benchmark - Comparison with AMGx

7 GPU Example



High Performance Scientific Computing @ UNIPI

8 New Ph.D. Program in HPSC

Call for **4 Ph.D. position** @ UNIPI

1. Position financed by IAC-CNR on **Parallel Linear Algebra**,
2. Positions financed by S.I.T. — Sordina IORT Technologies S.p.A on **Computational Methods and Models for Flash Radiotherapy**,
1. Position on the development and use of **HPC for electronic devices** based on advanced and innovative materials.

Call closes on July 18th - 13:00 CEST

www.dm.unipi.it/phd-hpsc/