High Performance Linear Algebra

Lecture 2.6: Putting it all Together: PSCToolkit

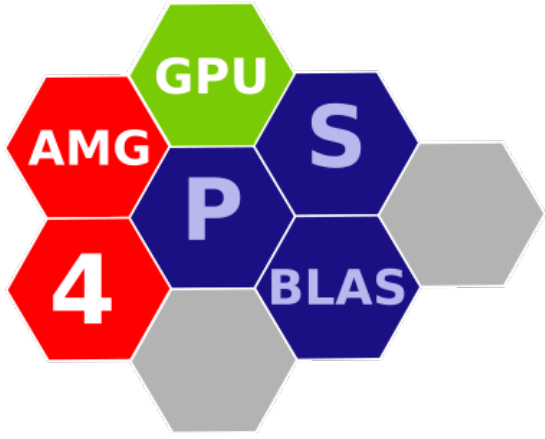
Ph.D. program in High Performance Scientific Computing

Salvatore Filippone **Pasqua D'Ambra** **Fabio Durastante**

March 27 2026 — 09.00:13.00

PSCToolkit

o



Outline

o

- Introduction: the PSBLAS library
- Serial kernels;
- Design patterns and object-oriented techniques;
- Using GPUs;
- Heterogeneous platforms;
- Preconditioners: approximate inverses;

Table of Contents

1 PSCToolkit

Main features:

- Designed for Krylov iterative linear system solvers;
- Main application: PDEs;
- Support for graph partitioning with common graph partitioning tools;
- Preconditioners: Simple preconditioners, plud AMG4PSBLAS: Agebraic Multigrid/Domain Decomposition framework;
- OOP design, easy to use (and extend);
- Transparent MPI/CUDA via PSBLAS (with some caveats);

Freely available from `https://psctoolkit.github.io`
See references [2, 3, 4, 9, 10]

The PSBLAS Team

Lead:

- Salvatore Filippone

Contributors to PSBLAS:

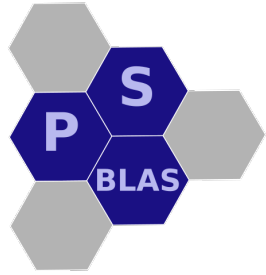
- Fabio Durastante
- Pasqua D'Ambra
- Marco Feder
- Simone Staccone
- Luca Pepè Sciarria
- Soren Rasmussen
- Zaak Beekman
- Theophane Loloum
- Ambra Abdullahi Hassan
- Thomas Amestoy
- Alfredo Buttari
- Michele Martone
- Michele Colajanni
- Fabio Cerioni
- Stefano Maiolatesi
- Dario Pascucci

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

PSCToolkit: PSBLAS Parallel Sparse BLAS

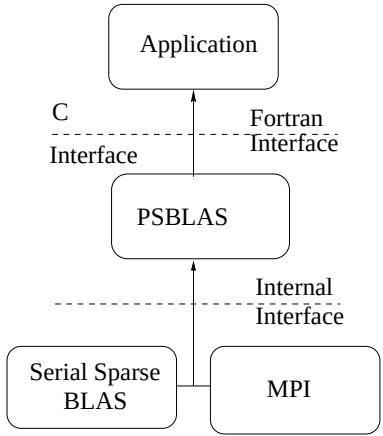
Main features:

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



Lots of previous work in standards for sparse and dense linear algebra (see [1, 8]); described in [9, 10]; version 3.9.0 released at the end of 2025.
Available from `https://psctoolkit.github.io/products/psblas`
RPMs available for Fedora and CentOS; SPACK builds under development, will be available with the next formal release.

Library Structure



autotools

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

CMake

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

Why PSBLAS?

Sparse Matrices and Krylov methods

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

Why PSBLAS?

Sparse Matrices and Krylov methods

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

Methods of choice: Search for a solution by projection

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

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

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

Why PSBLAS?

Sparse Matrices and Krylov methods

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

Methods of choice: Search for a solution by projection

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

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

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

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

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

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

An example Conjugate Gradient method

Compute $r^{(0)} = b - Ax^{(0)}$

for $i = 1, 2, \ldots$

solve $Mz^{(i-1)} = r^{(i-1)}$

$\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$

if $i = 1$

$p^{(1)} = z^{(0)}$

else

$\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$

$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$

endif

$q^{(i)} = Ap^{(i)}$

$\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$

$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$

$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$

Check convergence: $\|r^{(i)}\|_2 \leq \epsilon \|b\|_2$

end

```
call psb_geaxpby(one,b,zero,r,desc_a,info)
call psb_spmm(-one,A,x,one,r,desc_a,info)
rho = zero
iterate: do it = 1, itmax
   call psb_spsm(one,L,r,zero,w,desc_a,info)
   call psb_spsm(one,U,w,zero,z,desc_a,info)
   rho_old = rho; rho = psb_gedot(r,z,desc_a,info)
   if (it == 1) then
      call psb_geaxpby(one,z,zero,p,desc_a,info)
   else
      beta = rho/rho_old
      call psb_geaxpby(one,z,beta,p,desc_a,info)
   endif
   call psb_spmm(one,A,p,zero,q,desc_a,info)
   sigma = psb_gedot(p,q,desc_a,info); alpha = rho/sigma
   call psb_geaxpby(alpha,p,one,x,desc_a,info)
   call psb_geaxpby(-alpha,q,one,r,desc_a,info)
   rn2 = psb_genrm2(r,desc_a,info)
   bn2 = psb_genrm2(b,desc_a,info)
   err = rn2/bn2
   if (err.lt.eps) exit iterate
enddo iterate
```

The code in the previous slide is:

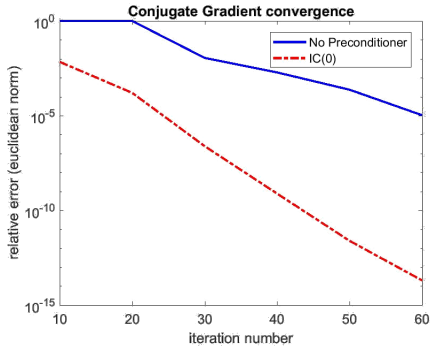
- Object-oriented (even though methods don't always appear as such, but that's syntactic sugar);
- Evolvable to new machines;
- Parallel: uses MPI (even if you don't see it);
- Parallel: uses OpenMP and CUDA (even if you don't see it);
- Handles heterogeneous and hybrid nodes transparently.

To achieve this, we had to develop a (substantial) support infrastructure.

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

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

Solving 2D Poisson eq.
(2500 dofs, $\mu(A) \approx 1.5 \times 10^3$)



IC(0): $B = LL^T$ with L incompl. Cholesky factor,
$\mu(B^{-1}A) \approx 2.2 \times 10^2$

PSBLAS Contents

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

Parallel Environment

1 PSCToolkit

We defined our parallel environment:

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

Basic operations:

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

Parallel Environment

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

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

Rules:

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

MPI interoperability

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

Parallel Environment: Hello World!

Writing `helloworld.f90`:

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

Parallel Environment: Hello World! (Makefile)

To compile and link we can use the information from the library installation, and write a simple Makefile

```
# Set in INSTALLDIR the install location of the PSBLAS  library
INSTALLDIR=/path/to/psblas
include $(INSTALLDIR)/include/Make.inc.psblas
INCDIR =$(INSTALLDIR)/include/
MODDIR =$(INSTALLDIR)/modules/
LIBDIR =$(INSTALLDIR)/lib/
PSBLAS_LIB= -L/$(LIBDIR) -lpsb_util -lpsb_linsolve \
            -lpsb_prec -lpsb_base
LDLIBS    = $(PSBLDLIBS)
FINCLUDES = $(FMFLAG)$(MODDIR) $(FMFLAG).
EXEDIR=./runs
all: helloworld
helloworld: helloworld.o
^^I$(FLINK) $(LOPT) helloworld.o -o helloworld \
             $(PSBLAS_LIB) $(LDLIBS)
^^I/bin/mv helloworld $(EXEDIR)
```

The C Interface

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

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

The routines defining the parallel environment are now:

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

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

The C version of the Hello World!


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

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

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

 psb_c_exit(ctxt);
}
```

The C Interface version of the Hello World! (Makefile)


```
INSTALLDIR=/path/to/psblas
include $(INSTALLDIR)/include/Make.inc.psblas
INCDIR=$(INSTALLDIR)/include/
MODDIR=$(INSTALLDIR)/modules/
LIBDIR=$(INSTALLDIR)/lib/
PSBLAS_LIB= -L/$(LIBDIR) -lpsb_util -lpsb_linsolve \
^^I-lpsb_prec -lpsb_base
PSBC_LIBS = -L/$(LIBDIR) -lpsb_cbind
LDLIBS    = $(PSBLDLIBS)
FINCLUDES = $(FMFLAG)$(MODDIR) $(FMFLAG).
CINCLUDES = -I$(INCDIR) -I$(LIBDIR) $(FIFLAG)$(INCLUDEDIR) \
^^I$(FIFLAG)$(PSBLAS_INCDIR)
EXEDIR=./runs
all: helloworld c_helloworld
helloworld: helloworld.o
  $(FLINK) $(LOPT) helloworld.o -o helloworld \
  ^^I$(PSBLAS_LIB) $(LDLIBS)
  /bin/mv helloworld $(EXEDIR)
c_helloworld: c_helloworld.o
  $(MPFC) c_helloworld.o -o c_helloworld $(PSBC_LIBS) \
^^I$(PSBLAS_LIB) $(LDLIBS) $(PSBLDLIBS) -lm -lgfortran
  /bin/mv c_helloworld $(EXEDIR)
```

Point-to-point communications


```
call psb_snd(ctxt, dat, dst [, m])
call psb_rcv(ctxt, dat, src [, m])
```

ctxt the communication context

dat The data item: an integer, real, complex variable, scalar or array; also character or logical scalar. Type and rank (and m) must agree on sender and receiver process; if m is not specified, size must agree as well.

dst/src Destination/source process.

m Optional number of rows when dat is a rank 2 array.

Semantics: “locally blocking” sends (i.e.: data buffer may be reused, but delivery may not have happened yet), blocking receives.

Collective operations


```
call psb_bcast(ctxt, dat [, root,mode,request])
call psb_sum(ctxt, dat [, root,mode,request])
call psb_amx(ctxt, dat [, root,mode,request])
call psb_amn(ctxt, dat [, root,mode,request])
call psb_max(ctxt, dat [, root,mode,request])
call psb_min(ctxt, dat [, root,mode,request])
call psb_scan_sum(ctxt, dat [,mode,request])
call psb_exscan_sum(ctxt, dat [,mode,request])
```

ctxt the communication context

dat The data item: an integer or real (everywhere), or complex variable (no max/min), scalar, or a rank 1 or 2 array; or a character or logical scalar (broadcast);

root Root of the collective operation. Default: 0 for broadcast, -1 (i.e.: all processes get the result) for reductions;

Collective operations


```
call psb_bcast(ctxt, dat [, root,mode,request])
call psb_sum(ctxt, dat [, root,mode,request])
call psb_amx(ctxt, dat [, root,mode,request])
call psb_amn(ctxt, dat [, root,mode,request])
call psb_max(ctxt, dat [, root,mode,request])
call psb_min(ctxt, dat [, root,mode,request])
call psb_scan_sum(ctxt, dat [,mode,request])
call psb_exscan_sum(ctxt, dat [,mode,request])
```

mode Whether to split the call, `psb_collective_start_`, `psb_collective_end_`, or `ior(psb_collective_start_,psb_collective_end_)` to execute immediately (default);

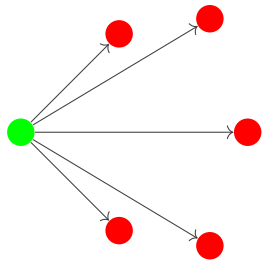
request A request handle, to be used in a completion call with `mode=psb_collective_end_` when a split has been requested.

Collective operations: broadcast


```
call psb_bcast(ctxt, dat [, root])
```

The semantics is equivalent to:

```
if (iam == root) then
  do i=0,np-1
   if (i /= iam) then
     call psb_snd(ctxt,a,i)
   end if
  end do
else
  call psb_rcv(ctxt,a,root)
end if
```



The implementation relies on MPI collectives, so it will be more efficient.

Collective operations: broadcast - C Interface

For the C interface the broadcast operation is divided by data type

```
void    psb_c_ibcast(psb_i_t ctxt, psb_i_t n, psb_i_t *v, psb_i_t root);
void    psb_c_sbcast(psb_i_t ctxt, psb_i_t n, psb_s_t *v, psb_i_t root);
void    psb_c_dbcast(psb_i_t ctxt, psb_i_t n, psb_d_t *v, psb_i_t root);
void    psb_c_cbcast(psb_i_t ctxt, psb_i_t n, psb_c_t *v, psb_i_t root);
void    psb_c_zbcast(psb_i_t ctxt, psb_i_t n, psb_z_t *v, psb_i_t root);
void    psb_c_hbcast(psb_i_t ctxt, const char *v, psb_i_t root);
```

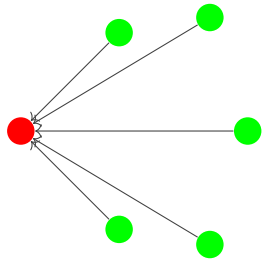
Note: In Fortran, we can overload functions. For the C interfaces we need to be specific.

Collective operations: sum-reduce


```
call psb_sum(ctxt, dat)
```

The semantics is equivalent to:

```
root = 0
if (iam == root) then
  do i=0,np-1
  if (i /= iam) then
    call psb_rcv(ctxt,tmp,i)
    a = a + tmp
  end if
  end do
else
  call psb_snd(ctxt,a,root)
end if
call psb_bcast(ctxt,a,root)
```



The implementation relies on MPI collectives, so it will be more efficient.

An extravagantly expensive way to compute:

$$\sum_{k=1}^{n} k^2 = \frac{n(2n+1)(n+1)}{6}.$$

Parallel Environment

An extravagantly expensive way to compute:

$$\sum_{k=1}^{n} k^2 = \frac{n(2n+1)(n+1)}{6}.$$

```
call psb_init(ctxt)
call psb_info(ctxt,iam,np)

temp = dble(iam) * dble(iam)
call psb_sum(ctxt,temp)
if (iam == 0) then
write(6,*) 'total sum for n = ',np,' is ',temp
endif
```

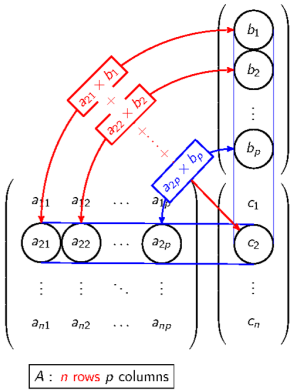
There is a mismatch in this slide, find it!

Exercises:

1. Build a parallel dot product;
2. Build a parallel dense matrix by vector product;
3. Solve a triangular system;
4. Define a mapping between global and local indices for an index space $1 \ldots N$: figure out who owns what, and try to achieve load balance;

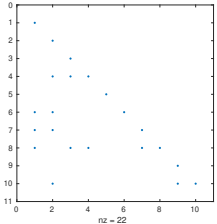
Necessary ingredients:

- (Parallel) Sparse matrix by Vector product;

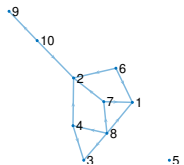


Note: we also need boundary data exchange and mesh management.

Necessary ingredients:



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



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

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

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

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

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

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

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

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

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

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

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

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

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

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

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

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

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

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

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

Computational kernels

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

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

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

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

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

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

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

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

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

Computational kernels

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

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

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

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

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

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

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

$\mathbf{y} \leftarrow \alpha DT^T\mathbf{x} + \beta\mathbf{y}$:

```
call psb_spsm(alpha,T,x,beta,y,desc_a,info,trans='T'[,unitd])
```

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

Computational kernels - C Interfaces

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

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

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

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

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

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

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

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

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

Computational kernels - C Interfaces

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

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

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

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

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

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

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

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

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

Computational kernels - C Interfaces

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

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

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

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

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

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

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

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

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

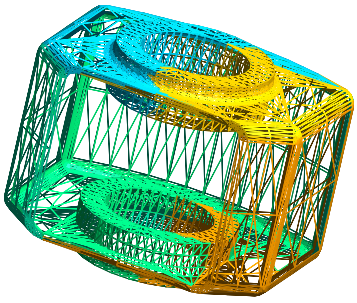
Data Distribution

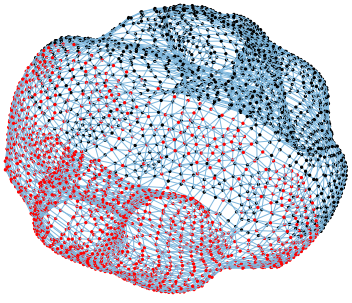
Guiding principle: "Owner computes" paradigm. Given an index space $1 \ldots N$ (and vectors defined on this index space):

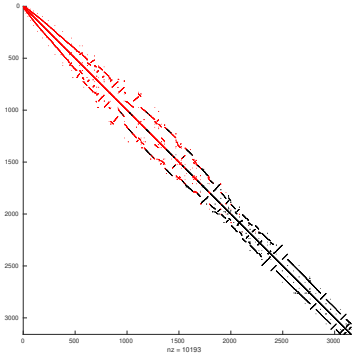
1. The index space is partitioned among processes;
2. Each index has a "home" process;
3. The "home" process holds the authoritative value of the corresponding vector entry;
4. The "home" process performs the arithmetic operations needed to set the value of a vector entry;
5. On each process, the set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

Data Distribution

1 PSCToolkit





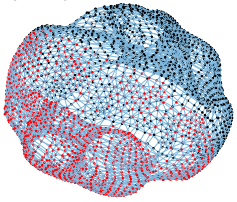


Mesh partition $\Leftrightarrow$ Graph partition $\Leftrightarrow$ Matrix row partition

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

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

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



From a sparse matrix to a graph:

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

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

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

Communication descriptors

What is a communication descriptor?
An opaque object that:

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

```
call psb_halo(x,desc)
```

Communication descriptors

What is a communication descriptor?
An opaque object that:

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

```
psb_i_t psb_c_dhalo(psb_c_dvector *x, psb_c_descriptor *desc_a);
```

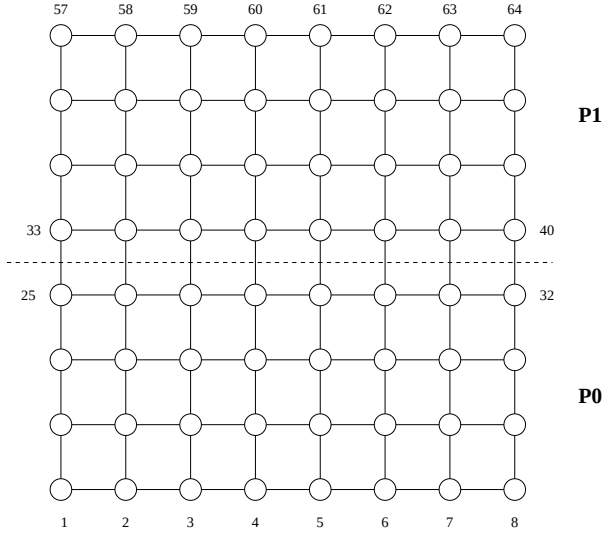
Descriptor: Fortran Interface


```
type psb_desc_type
class(psb_indx_map), allocatable :: indxmap
type(psb_i_vect_type)            :: v_halo_index
type(psb_i_vect_type)            :: v_ext_index
type(psb_i_vect_type)            :: v_ovrlap_index
type(psb_i_vect_type)            :: v_ovr_mst_idx
end type psb_desc_type
```

Support for various data management operations.

Halo Exchange

1 PSCToolkit



Halo Exchange

After a `call` to `psb_halo`:

	Process 0			Process 1	
I	GLOB(I)	X(I)	I	GLOB(I)	X(I)
1	1	1.0	1	33	2.0
2	2	1.0	2	34	2.0
3	3	1.0	3	35	2.0
4	4	1.0	4	36	2.0
5	5	1.0	5	37	2.0
6	6	1.0	6	38	2.0
7	7	1.0	7	39	2.0
8	8	1.0	8	40	2.0
		.....			
29	29	1.0	29	61	2.0
30	30	1.0	30	62	2.0
31	31	1.0	31	63	2.0
32	32	1.0	32	64	2.0
33	33	2.0	33	25	1.0
34	34	2.0	34	26	1.0
35	35	2.0	35	27	1.0
36	36	2.0	36	28	1.0
37	37	2.0	37	29	1.0
38	38	2.0	38	30	1.0
39	39	2.0	39	31	1.0
40	40	2.0	40	32	1.0

Black values have been conserved; red values have been freshly updated, and are guaranteed to match the “home” value

How do we set up a descriptor/sparse matrix?

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

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

This is done with the initialization routine `PSB_CDALL`

Descriptor Allocation


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

Global size: `m = size(v)`

Descriptor Allocation


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

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

Descriptor Allocation


```
! Assign a bunch of contiguous indices
call psb_cdall(ictxt,desc,info,nl=nl)
```

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

Descriptor Allocation


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

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

Here we may even assign an index to multiple processes! Global size: `m = mg`

Descriptor Allocation

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

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

where

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

The mapping as such is complete (On each process P we can now answer whether I belongs here)
BUT
there is no description (yet) of the connections/interactions among subdomains.

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

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

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


```
do i=1, n
  if ( <this index belongs to me> ) then
     nz        = <number of neighbours of i>
     ia(1:nz) = i
     ja(1:nz) = <list of neighbours of i>
     call psb_cdins(nz,ia,ja,desc,info)
  endif
enddo
```

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

1. If $IA(k) \notin P$ then both $IA(k)$ and $JA(k)$ are handled separately, depending on user choice;
2. If $IA(k) \in P$ and $JA(k) \notin P$ then we record the linkage to another (possibly as yet unknown) process Q

End of build stage:

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

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

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

Sparse Matrix Allocation

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

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

Note:

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


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

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

1. If $IA(k) \notin P$ then $IA(k)$, $JA(k)$ and $VAL(k)$ are handled separately (depending on user choices);
2. If $IA(k) \in P$ and $JA(k) \notin P$ then we have a communication requirement that has to be coherent with $DESC$;
3. There actually is no need to process (entire) row by (entire) row; the order may be arbitrary (e.g.: all the coefficients associated with an element, coefficient by coefficient, etc).
4. It is convenient for performance to group multiple rows into a single call;

End of build stage:

```
call psb_spasb(a,desc_a,info [, afmt, upd, dupl, mold])
```

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

- With *DUPL* we may handle duplicated coefficients (i.e. multiple entries with identical row and column indices) as:
  - `psb_dupl_ovwrt_` Keep one of them;
  - `psb_dupl_add_` Sum the coefficients;
  - `psb_dupl_err_` Raise an error.

Often the same matrix pattern (i.e.: mesh and stencil) is reused multiple times with different coefficients; we can put the sparse matrix in the UPDATE state with

```
call psb_sprn(a,desc_a,info)
```

then go through exactly the same insertion loop phase and assembly as before; this time any entries that were NOT in the first build stage will be ignored.
We may also clone a sparse matrix

```
call psb_sp_clone(a,b,info)
```

and then reinit/overwrite it to obtain an independent matrix object with the same pattern (i.e. the two matrices share the same descriptor).

Same overall code structure with dense vectors

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

Rules of precedence:

- A call to `PSB_CDALL` must precede any calls to either `psb_spall` or `psb_geall` using the same descriptor
- A call to `PSB_CDASB` must precede any calls to either `psb_spasb` or `psb_geasb` using the same descriptor

Most routines in PSBLAS must be called by all processes participating in a context: all the computational, allocation, assembly.
The insertion routines `PSB_xxINS` are the main exception, and are called independently; a subsequent call to `PSB_xxASB` is required for synchronization.

General Application Structure 1

1. Initialize communication descriptor `PSB_CDALL`;
2. Loop on mesh with `PSB_CDINS` and finish with `PSB_CDASB`;
3. Initialize sparse matrix `PSB_SPALL`.
4. Loop on all mesh points, build the equations `PSB_SPINS` and `PSB_GEINS`.
5. Assemble matrix `PSB_SPASB`, `PSB_GEASB`;

If the same discretization mesh is reused, it is possible to repeat steps 3 and 4 by using `PSB_SPRN`.

General Application Structure 2

However it is also legal to call PSB_SPINS when the descriptor is in the BUILD state; in this case the library is implicitly adding a call to PSB_CDINS.

1. Initialize communication descriptor PSB_CDALL;
2. Initialize sparse matrix PSB_SPALL.
3. Loop on mesh, build the equations PSB_SPINS and PSB_GEINS.
4. Assemble descriptor and matrix PSB_CDASB, PSB_SPASB, PSB_GEASB;

Preconditioned iterations


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

Mandatory arguments:

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

Preconditioned iterations

Optional arguments:

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

Preconditioned iterations - C Interfaces

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

```
int psb_c_skrylov(const char *method, psb_c_sspmat *ah,
        psb_c_sprec *ph, psb_c_svector *bh, psb_c_svector *xh,
        psb_c_descriptor *cdh, psb_c_SolverOptions *opt);
int psb_c_dkrylov(const char *method, psb_c_dspmat *ah,
        psb_c_dprec *ph,  psb_c_dvector *bh, psb_c_dvector *xh,
        psb_c_descriptor *cdh, psb_c_SolverOptions *opt);
int psb_c_ckrylov(const char *method, psb_c_cspmat *ah,
        psb_c_cprec *ph, psb_c_cvector *bh, psb_c_cvector *xh,
        psb_c_descriptor *cdh, psb_c_SolverOptions *opt);
int psb_c_zkrylov(const char *method, psb_c_zspmat *ah,
        psb_c_zprec *ph, psb_c_zvector *bh, psb_c_zvector *xh,
        psb_c_descriptor *cdh, psb_c_SolverOptions *opt);
```

Preconditioned iterations - C Interfaces

The solver options are contained into a structure

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

that can be initialized to the default values with the routine

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

Preconditioners

Simple preconditioners:

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

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

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

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

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

Table of Contents

2 AMG4PSBLAS

AMG4PSBLAS: Advanced Preconditioners

A package of preconditioners for PSBLAS in PSCToolkit:

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

version 1.2 released at the end of 2025.

AMG4PSBLAS: Advanced Preconditioners

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

Algebraic Multigrid with AMG4PSBLAS

- Jacobi, hybrid forward/backward Gauss-Seidel, block-Jacobi, additive Schwarz, Chebychev polynomials;
- Local solvers: Incomplete Factorizations, Approximate Inverses
- Algebraic multigrid with smoothed and un-smoothed aggregation
- V-, W-, and K-cycles
- Coarse level solvers: sparse direct solvers (SuperLU_Dist, MUMPS, UMFPACK), (recursive) Krylov solvers

The AMG4PSBLAS Team

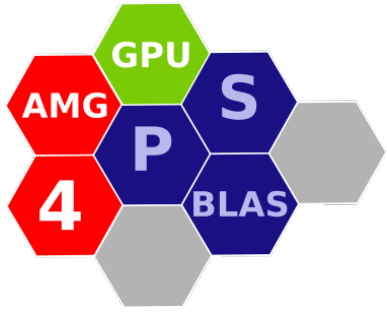
Developers:

- Salvatore Filippone
- Pasqua D'Ambra
- Fabio Durastante

Contributors:

- Marco Feder
- Luca Pepé Sciarria
- Ambra Abdullahi Hassan
- Daniela di Serafino
- Alfredo Buttari

[5, 6]



Freely available from:
psctoolkit.github.io

Scalable (optimal) preconditioners

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

Scalable (optimal) preconditioners

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

MultiGrid (MG) Preconditioners

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

(but optimal preconditioner is not always the fastest preconditioner)

AMG (Brandt, McCormick and Ruge, 1984)

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

AMG (Brandt, McCormick and Ruge, 1984)

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

Key issue in effective AMG for general matrices

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

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

AMG (Brandt, McCormick and Ruge, 1984)

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

Key issue in effective AMG for general matrices

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

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

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

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

Wanted Iterative method B to precondition the CG method:

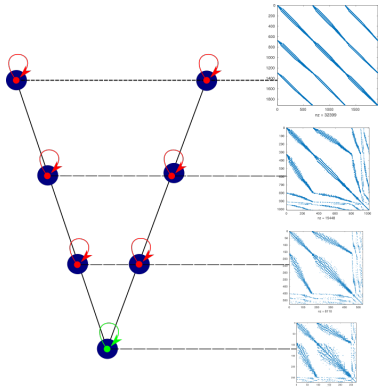
- Hierarchy of systems

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

- Transfer operators:

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

Missing Structural/geometric infos



Smoother

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

Prolongator

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

Complementarity of Smoother and Prolongator

Algebraic MultiGrid (AMG) Setup

Recursive application of a two-grid scheme

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

Recursive application of a two-grid scheme

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

AMG based on Aggregation of dofs

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

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

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

or smoothed version of P (Vaněk 1996)

Parallel AMG Setup: decoupled aggregation

Given a user-defined threshold ε

Repeat

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

Until all points are marked

For all leftover points

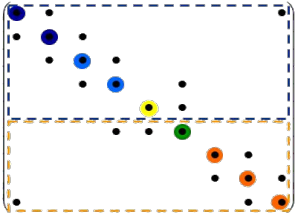
- Add to an aggregated neighbour over threshold; if multiple ones, choose

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

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

Endfor

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



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

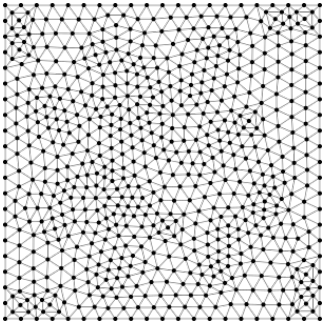
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

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

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

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



P. D'Ambra, S. Filippone and P. S. Vassilevski, BootCMatch: a software package for bootstrap AMG based on graph weighted matching, ACM Trans. Math. Software **44** (2018), no. 4, Art. 39, 25 pp., [7]

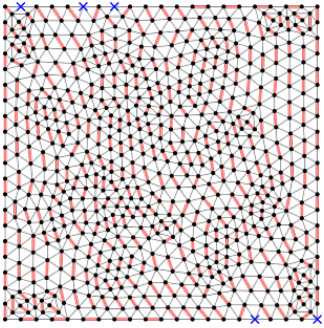
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

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

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

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



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

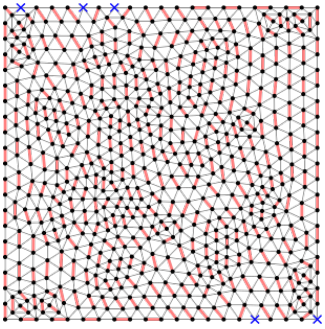
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

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

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

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



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

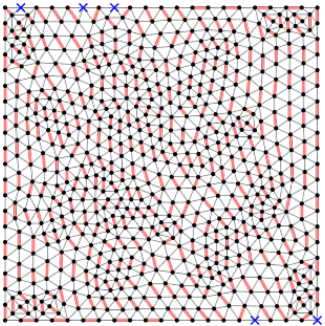
Parallel AMG Setup: Matching based aggregation

AMG based on weighted graph matching

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

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

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



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

Comparison of the Coarsening Strategy

VBM Decoupled aggregation

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

Matching-based aggregation

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

Table of Contents

3 User's Interface

User's interface for preconditioner setup

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

User's interface for preconditioner apply

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

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

Parameter Setting for Preconditioner Setup


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

Parameter Setting for Preconditioner Setup (cont'd)


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

Parameter Setting for Preconditioner Setup (cont'd)


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

Current version of AMG4PSBLAS preconditioners

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

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

solve phase: **GPU application implemented**

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

How to play around

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

How to play around

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

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

Example of use for CPU/GPU


```
! sparse matrix
type(psb_dspmat_type) :: A
! variable declaration needed for GPU running
type(psb_d_hlg_sparse_mat), target   :: ahlg
type(psb_d_vect_gpu)  :: vgm
type(psb_i_vect_gpu)  :: igm
! sparse matrix descriptor
type(psb_desc_type)   :: DESC_A
! preconditioner data
type(amg_dprec_type)  :: P
...
! inizialize parallel environment
call psb_init(ctxt)
call psb_info(ctxt,iam,np)
! read and assemble matrix A and rhs b using
! PSBLAS facilities
```

Example of Use for CPU/GPU (cont'd)


```
! setup AMG preconditioner
call P%init('ML', info)
! Setting up the options (more of this later...)
call P%set(attribute, value, info)
! build preconditioner
call P%hierarchy_build(A,DESCA,info)
! last three optional parameters for GPU running
call P%smoothers_build(A,DESCA,info,am=ahlg, &
       & vm=vgm, im=igm)
! print description of the built preconditioner
call P%descr(info)
! conversions & vector assembly for GPU running
call DESCA%cnv(mold=igm)
call A%cscnv(info,mold=ahlg)
call psb_geasb(x,DESC_A,info,mold=vgm)
call psb_geasb(b,DESC_A,info,mold=vgm)
```

Example of Use for CPU/GPU (cont'd)


```
! set solver parameters and initial guess
...
! solve Ax=b with precond CG
call psb_krylov('CG',A,P,b,x,tol,DESC_A,info,&
       & further options)
...
! cleanup storage
call P%free(info)
...
!
! leave PSBLAS
call psb_exit(ctxt)
```

And that's all there is to it!

See also [11]

References

4 Bibliography

[1] L. S. Blackford and et al. “An updated set of basic linear algebra subprograms (BLAS)”. In: *ACM Trans. Math. Software* 28.2 (2002), pp. 135–151. ISSN: 0098-3500. DOI: 10.1145/567806.567807. URL: https://doi.org/10.1145/567806.567807.

[2] 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. eprint: https://doi.org/10.1137/20M134914X. URL: https://doi.org/10.1137/20M134914X.

[3] P. D’Ambra, D. di Serafino, and S. Filippone. “MLD2P4: a Package of Parallel Algebraic Multilevel Domain Decomposition Preconditioners in Fortran 95”. In: *ACM Trans. Math. Softw.* 37.3 (2010), pp. 7–23.

References

4 Bibliography

[4] P. D'Ambra, D. di Serafino, and S. Filippone. "On the development of PSBLAS-based parallel two-level Schwarz preconditioners". In: *Appl. Numer. Math.* 57.11-12 (2007), pp. 1181–1196. ISSN: 0168-9274.

[5] P. D'Ambra et al. "Optimal polynomial smoothers for parallel AMG". In: *Numerical Algorithms* 100.4 (Dec. 2025), pp. 1783–1812. ISSN: 1572-9265. DOI: 10.1007/s11075-025-02117-6. URL: https://doi.org/10.1007/s11075-025-02117-6.

[6] P. D'Ambra, F. Durastante, and S. Filippone. "Parallel Sparse Computation Toolkit". In: *Software Impacts* 15 (2023), p. 100463. ISSN: 2665-9638. DOI: https://doi.org/10.1016/j.simpa.2022.100463. URL: https://www.sciencedirect.com/science/article/pii/S2665963822001476.

References

4 Bibliography

[7] P. D'Ambra, S. Filippone, and P. S. Vassilevski. "BootCMatch: A Software Package for Bootstrap AMG Based on Graph Weighted Matching". In: *ACM Trans. Math. Softw.* 44.4 (June 2018). ISSN: 0098-3500. DOI: 10.1145/3190647. URL: https://doi.org/10.1145/3190647.

[8] I. S. Duff et al. "Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface". In: *ACM Trans. Math. Software* 23.3 (1997), pp. 379–401. ISSN: 0098-3500. DOI: 10.1145/275323.275327. URL: https://doi.org/10.1145/275323.275327.

[9] S. Filippone and A. Buttari. "Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003". In: *ACM Trans. Math. Softw.* 38.4 (2012), 23:1–23:20.

References

4 Bibliography

[10] S. Filippone and M. Colajanni. “PSBLAS: a library for parallel linear algebra computations on sparse matrices”. In: *ACM Trans. Math. Softw.* 26.4 (Dec. 2000), pp. 527–550.

[11] S. Filippone et al. “Sparse matrix-vector multiplication on GPGPUs”. In: *ACM Trans. Math. Software* 43.4 (2017), Art. 30, 49. ISSN: 0098-3500.