PSCTOOLKIT 1.0: Sparse Computations and Iterative Solvers on Parallel Computers

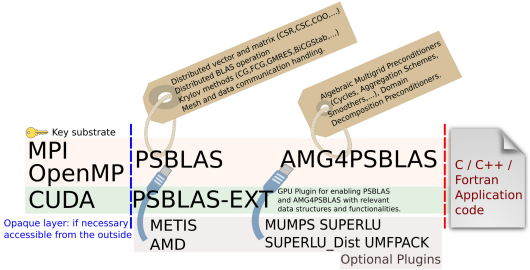
Salvatore Filippone

Università di Roma Tor Vergata
salvatore.filippone@uniroma2.it

http://psctoolkit.github.io

PSCToolKit

- Collection of tools
  1. PSBLAS
  2. PSBLAS-EXT
  3. AMG4PSBLAS
- Freely available `https://psctoolkit.github.io`
- Highly scalable



The PSBLAS Team

Project lead:

- Salvatore Filippone

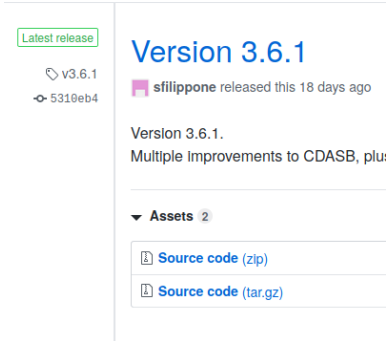
Contributors:

- Fabio Durastante
- Soren Rasmussen
- Zaak Beekman
- Ambra Abdullahi Hassan
- Pasqua D'Ambra
- Alfredo Buttari
- Daniela di Serafino
- Michele Martone
- Michele Colajanni
- Fabio Cerioni
- Stefano Maiolatesi
- Dario Pascucci

PSBLAS Parallel Sparse BLAS

Main features:

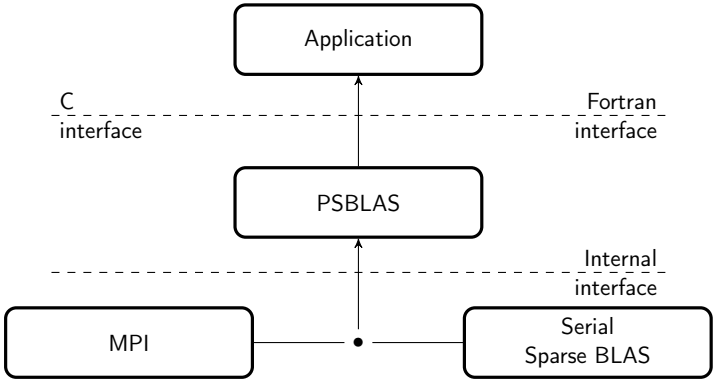
- Designed for iterative solvers; but, support for mesh handling;
- Main application: differential problems;
- Data allocation through graph partitioning;
- Support for overlap;



Prompted by work of Duff et al. [1997], Blackford and et al. [2002]; see Filippone and Colajanni [2000], Filippone and Buttari [2012]; now at version 3.8:

```
git clone https://github.com/sfilippone/psblas3.git
git branch origin/psblas-3.6-maint
git pull
```

Library Structure




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

Why PSBLAS?

- Existing software standards:
  - MPI,
  - serial sparse BLAS,
  - Metis/Parmetis,
  - AMD
- Attention to performance issues (modern Fortran features);
- Research on new preconditioners (next tutorial on MLD2P4);
- Minimal set of requirements on the user (no need to delve in the data structures);
- Support tools for mesh handling beyond simple algebraic operations;
- Error handling;

When dealing with parallel sparse matrices, we end up handling distributed discretization meshes!

PSBLAS Contents

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

Table of Contents

Parallel Environment

We defined our own parallel environment:

- Implemented in pure MPI;
- Convenient subset of MPI communication modes;
- MPI is directly available when needed;
- Fortran generic interfaces (no type mismatch!)

Basic operations:

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

Parallel Environment

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

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

Rules:

- `psb_init` *must* be called before anything else;
- 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(icontxt)
mpirank = psb_get_mpi_rank(icontxt, id)
```

Parallel Environment: Hello World!

We write the file `helloworld.f90`:

```
program hello_world
  use psb_base_mod
  implicit none
  integer(psb_ipk_) :: ictxt, iam, np
  character(len=20)  :: name
  call psb_init(ictxt)
  call psb_info(ictxt,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(ictxt)
  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=..
include $(INSTALLDIR)/Make.inc
INCDIR =$(INSTALLDIR)/include/
MODDIR =$(INSTALLDIR)/modules/
LIBDIR =$(INSTALLDIR)/lib/
PSBLAS_LIB= -L/$(LIBDIR) -lpsb_util -lpsb_krylov \
        -lpsb_prec -lpsb_base
LDLIBS    = $(PSBLDLIBS)
FINCLUDES = $(FMFLAG)$(MODDIR) $(FMFLAG).
EXEDIR=./runs
all: helloworld
helloworld: helloworld.o
        $(FLINK) $(LOPT) helloworld.o -o helloworld \
                $(PSBLAS_LIB) $(LDLIBS)
        /bin/mv helloworld $(EXEDIR)
```

The C Interface

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

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

The routines defining the parallel environment are now:

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

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

The C Interface version of the Hello World!

We write the file `c_helloworld.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 ictxt, iam, np;
char name[]="c_helloworld";

ictxt = psb_c_init();
psb_c_info(ictxt,&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(ictxt);
}
```

The C Interface version of the Hello World!

We can modify the Makefile to compile also this version

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

Point-to-point communications

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

icontxt the communication context

dat The data item: an integer, real, complex variable, scalar or array; also character or logical scalar. Type and rank 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(icontxt, dat [, root])
call psb_sum(icontxt, dat [, root])
call psb_amx(icontxt, dat [, root])
call psb_amn(icontxt, dat [, root])
call psb_max(icontxt, dat [, root])
call psb_min(icontxt, dat [, root])
```

icontxt 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 (only broadcast).

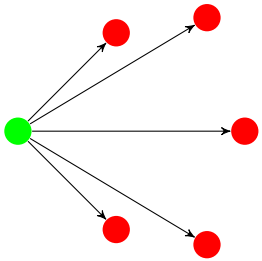
root Root of the collective operation. Default: 0 for broadcast, -1 (i.e.: all processes get the result) for reductions.

Collective operations: broadcast

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

The semantics is equivalent to:

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



The implementation can be *much* 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 ictxt, psb_i_t n, psb_i_t *v, \
         psb_i_t root);
void    psb_c_sbcast(psb_i_t ictxt, psb_i_t n, psb_s_t *v, \
         psb_i_t root);
void    psb_c_dbcast(psb_i_t ictxt, psb_i_t n, psb_d_t *v, \
         psb_i_t root);
void    psb_c_cbcast(psb_i_t ictxt, psb_i_t n, psb_c_t *v, \
         psb_i_t root);
void    psb_c_zbcast(psb_i_t ictxt, psb_i_t n, psb_z_t *v, \
         psb_i_t root);
void    psb_c_hbcast(psb_i_t ictxt, const char *v, psb_i_t root);
```

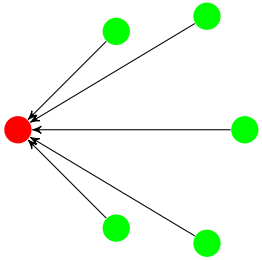
Note: In Fortran 90, we can overload functions with an interface, even with the same argument names, it is sufficient that they can be distinguished by the TYPE of the arguments. For the C interfaces we need to be specific.

Collective operations: sum-reduce

```
call psb_sum(icontxt, dat)
```

The semantics is equivalent to:

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



The implementation can be *much* more efficient.

Parallel Environment

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(icontxt)
call psb_info(icontxt,iam,np)

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

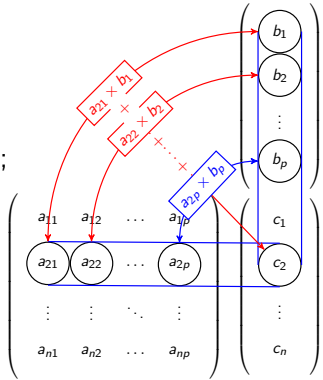
There is a mismatch in this slide, find it!

Table of Contents

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

- (Parallel) Sparse matrix by Vector product;

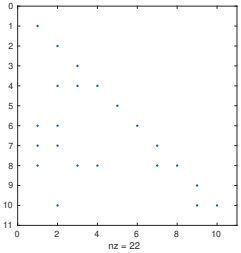


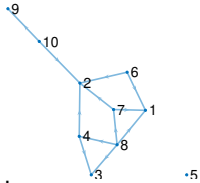
Note: we also need boundary data exchange and mesh management.

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

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





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

Computational kernels: a toolkit for iterative solvers

Necessary ingredients:

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

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

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;

$$\|x\|_1 = \sum_i |x_i|$$

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

$$\|x\|_\infty = \max_i |x_i|$$

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

$x^T y$ ($x^H y$): `dot = psb_gedot(x,y,desc_a)`

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

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

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

$\|x\|_2$: `nrm2 = psb_genrm2(x,desc_a)`

$\|x\|_\infty$ `nrmi = psb_spnrmi(x,desc_a)`

$y \leftarrow \alpha A x + \beta y$: `call psb_spmm(alpha,A,x,beta,y,desc_a)`

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

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

Computational kernels

$x^T y$ ($x^H y$): `dot = psb_gedot(x,y,desc_a)`

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

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

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

$\|x\|_2$: `nrm2 = psb_genrm2(x,desc_a)`

$\|x\|_\infty$ `nrmi = psb_spnrmi(x,desc_a)`

$y \leftarrow \alpha A^T x + \beta y$: `call psb_spmm(alpha,A,x,beta,y,desc_a[,trans])`

$y \leftarrow \alpha D T^{-1} x + \beta y$: `call psb_spsm(alpha,T,x,beta,y,desc_a[,trans,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

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

$y \leftarrow \alpha x + \beta y$: `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);`

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

$\|x\|_\infty$ `psb_d_t psb_c_dspnrmi(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$y \leftarrow \alpha Ax + \beta y$: `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

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

$y \leftarrow \alpha x + \beta y$: `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);`

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

$\|x\|_\infty$ `psb_d_t psb_c_dspnrmi(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$y \leftarrow \alpha A^T x + \beta y$: `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

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

$y \leftarrow \alpha x + \beta y$: `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);`

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

$\|x\|_\infty$ `psb_d_t psb_c_dspnrmi(psb_c_dvector *x, psb_c_descriptor *desc_a);`

$y \leftarrow \alpha D T^{-1} x + \beta y$: `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`.

An example Conjugate Gradient method

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

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

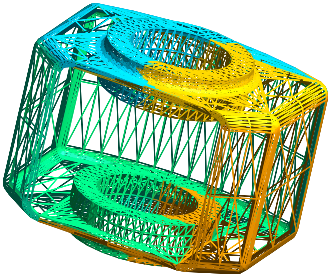
Table of Contents

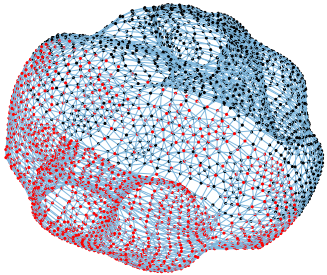
Data Distribution

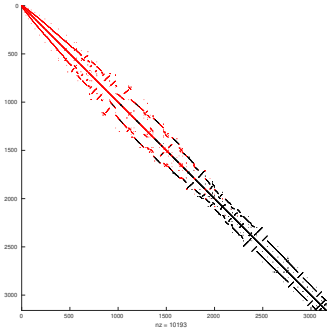
Guiding principle: "Owner computes" paradigm. Given an index space $1 \ldots N$ (and vectors defined on this index space):

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







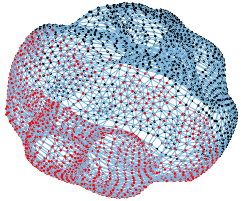
Mesh partition $\Leftrightarrow$ Graph partition $\Leftrightarrow$ Matrix row partition

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

Data Distribution

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

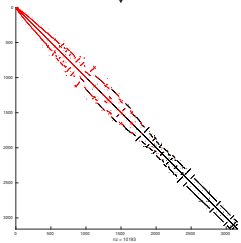
$$V = \{v_1, \ldots, v_n\}$$
$$E \subseteq V \times V$$



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.



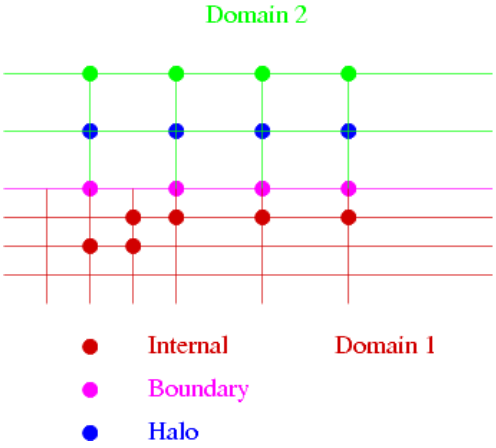
Note: numbering of vertices induces a different pattern (symmetric permutation)

Data Distribution

Mesh point classification:

- **Internal:** Points belonging to a given domain and only interacting with points belonging to the same domain
- **Boundary:** Points interacting with points from other domains;
- **Halo:** Points belonging to other domains whose value is needed;
- **Overlap:** Points replicated across multiple domains (used in some preconditioners);

Data Distribution



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

Communication descriptors

What is a communication descriptor?
An opaque object that:

- Keeps track of the parallel machine (icontxt);
- 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)

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

Communication descriptors

What is a communication descriptor?
An opaque object that:

- Keeps track of the parallel machine (icontxt);
- 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)

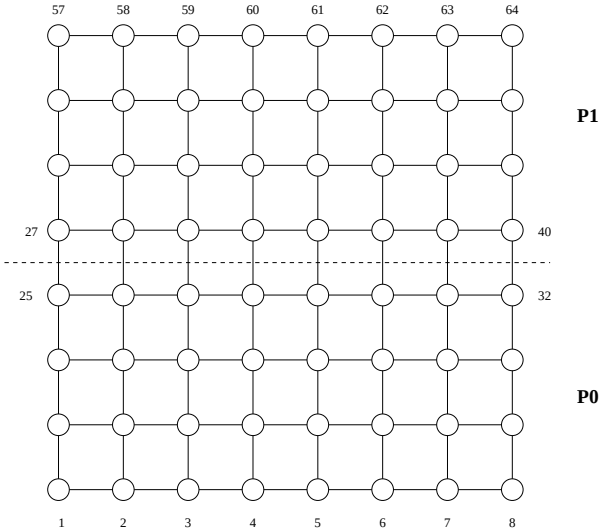
```
psb_i_t psb_c_dhalo(psb_c_dvector *x,
        psb_c_descriptor *desc_a);
```

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



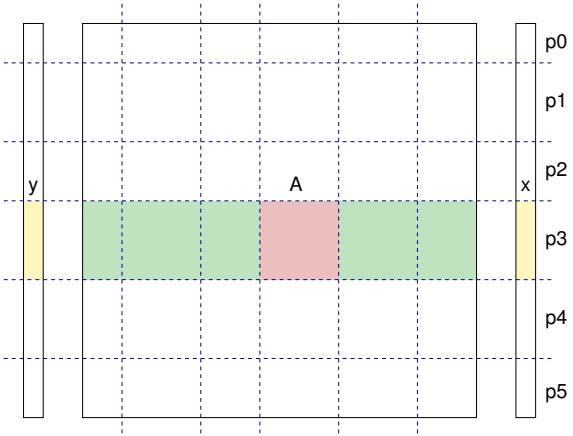
Halo Exchange

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

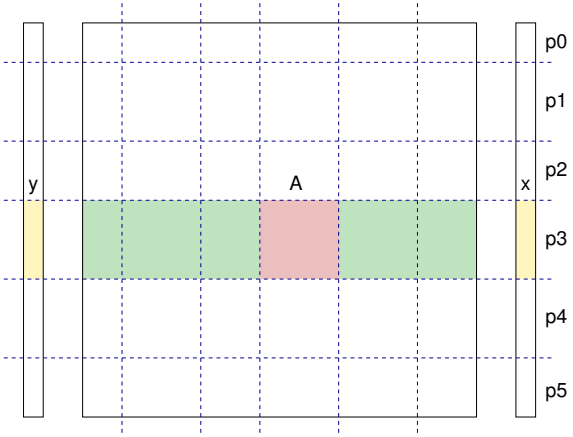
Black values have been conserved; red values have been freshly updated, and are guaranteed to match the "home" value

Distributed Matrix-Vector Product



The pink area corresponds to the local indices, the green area corresponds to the halo. *Surface to Volume effect*: for "sensible" data distributions, most of the nonzeros are in the pink area, the green area is almost empty.

Distributed Matrix-Vector Product



To compute the SpMV, we have to retrieve values of X corresponding to entries in the green area with a *halo data exchange*. This is a *sparse persistent all-to-all-V collective communication*

The halo data exchange

We can implement the halo data exchange in multiple ways:

- Use `MPI_Alltoallv`;
- Build and use a list of messages, then use `MPI_Send` and `MPI_Recv`;
- same as above, but with `MPI_Isend` and `MPI_Irecv`;
- Use the new MPI 4.0 Neighborhood Collective Communications on Process Topologies (not yet widely available)

The halo data exchange

Use `MPI_Alltoallv`

The main issue is that the MPI collective is used in a situation in which most pairs of processes do NOT actually exchange data

Use `MPI_Send` and `MPI_Recv`

Beware of scheduling: should avoid unsafe communications

Use `MPI_Isend` and `MPI_Irecv`

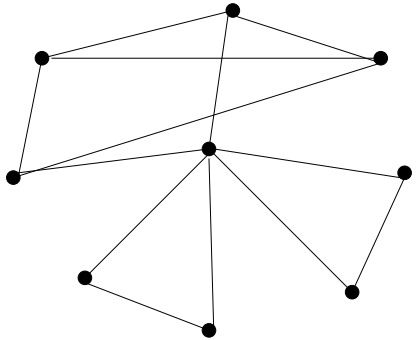
Currently, default (also, uses scheduling)

Use Neighborhood Collectives

Should actually be *persistent* collectives (reuse multiple times)

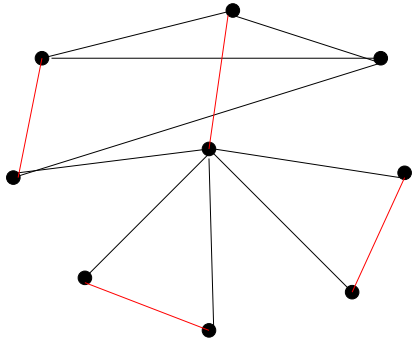
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



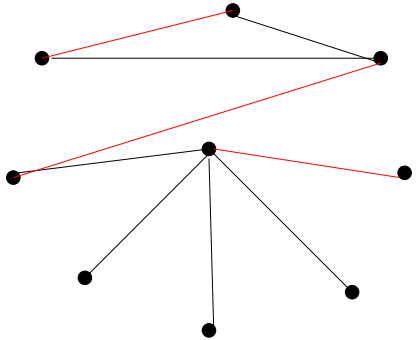
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



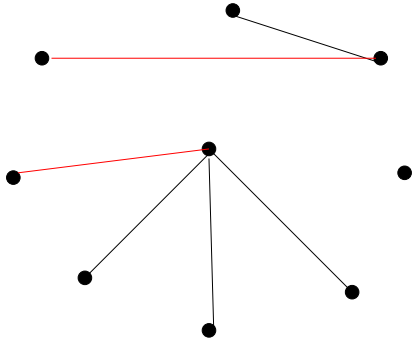
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



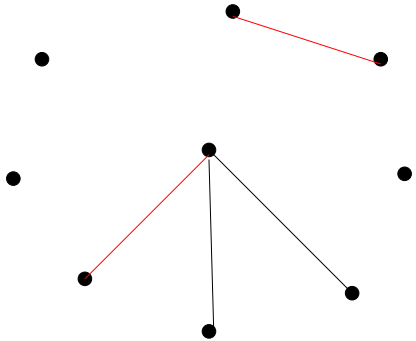
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



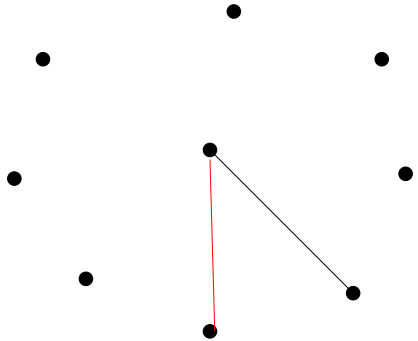
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



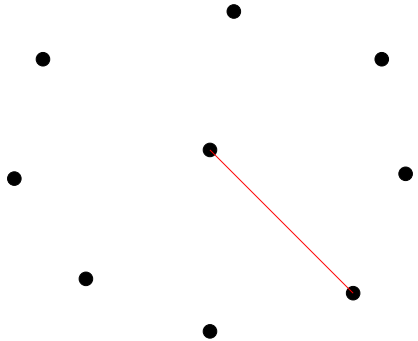
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



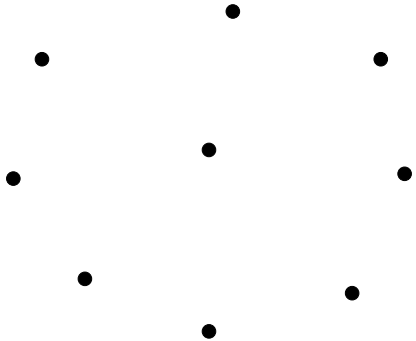
The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?



Scheduling communications

The problem

If you have a number of messages to exchange, how do you schedule them to minimize the time to completion?

Recipe:

Apply a series of maximal matchings.
Heuristics: scan the nodes in order of descending degree

Table of Contents

Storage schemes

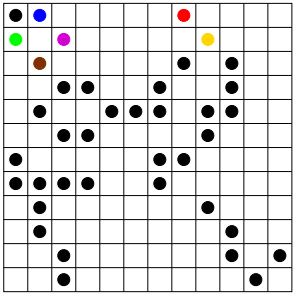
Coordinate storage:

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

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

Storage schemes

COO



Elements Array	●	●	●	●	●	●	●	●	...
Col idx array	1	2	8	1	3	9	2	8	...
Row idx array	1	1	1	2	2	2	3	3	...

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

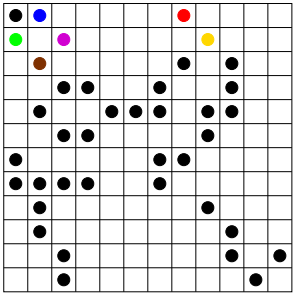
Storage schemes

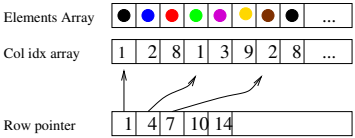
Compressed Storage by Rows:

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

Storage schemes

CSR






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

Are you unsure what to use?

Well, so are we.

Facts:

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

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

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

We want maximum freedom, flexibility, maintainability and performance (i.e. we like to have our cake and eat it too)

We have also GPUs!

We can manage transparently also data on GPUs Cardellini et al. [2014], Filippone et al. [2017]:

https://github.com/davidebarbieri/spgpu

and will be discussed in detail in a following dedicated tutorial.

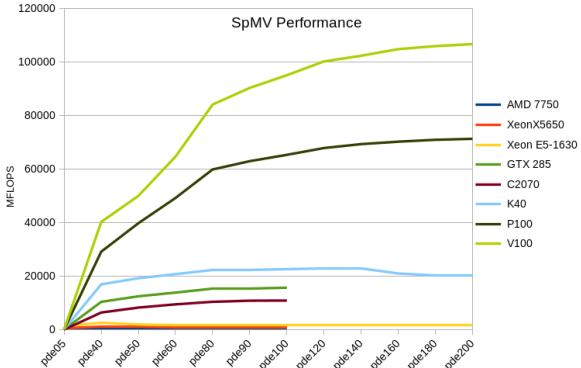


Table of Contents

Data Management

How do we set up a descriptor/sparse matrix?

Data Management

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

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

This is done with the initialization routine `psb_cdall`

Data Management

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

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

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

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

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)`

Metis

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

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)`

The corresponding C style allocation is obtained using

```
psb_i_t psb_c_cdall_vg(psb_i_t ng, psb_i_t *vg,
        psb_i_t ictxt,
        psb_c_descriptor *desc_a);
```

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

```
! 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))`
The corresponding C style allocation is obtained using

```
psb_i_t psb_c_cdall_vl(psb_i_t nl, psb_i_t *vl,
        psb_i_t ictxt,
        psb_c_descriptor *desc_a);
```

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

```
! 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)`
The corresponding C style allocation is obtained using

```
psb_i_t psb_c_cdall_nl(psb_i_t nl, psb_i_t ictxt,
        psb_c_descriptor *desc_a);
```

Descriptor Allocation

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

call psb_cdall(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.

Descriptor Allocation

If you have some information on the topology of the processes you can suggest it to the construction routine:

```
integer(psb_mpk_), allocatable :: neighbours(:)
integer(psb_mpk_) :: cnt
allocate(neighbours(np))
''put in neighbours(1:cnt) the list of its cnt
     neighbours processes''
call psb_realloc(cnt, neighbours,info)
call desc_a%set_p_adjcncy(neighbours)
```

If the information are correct this procedure speeds up the construction of the communicator, if it is not, then slows it down. Nevertheless, it is not compulsory, but if the information is available why not use it?

Data Management

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

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

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

```
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: if $IA(k) \in P$ and $JA(k) \notin P$ then we record the linkage to another (possibly as yet unknown) process Q

Data Management - C Interface

The procedure with the C interface is completely analogous:

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

End of build stage:

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

or, in the C interface,

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

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

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

Sparse Matrix Allocation

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

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

or, in the C interface,

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

Note:

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

Sparse Matrix Allocation

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

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

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
```

Sparse Matrix Allocation - C Interface

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

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

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

The procedures for the other data types are completely analogous.

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 $IA(k) \in P$ and $JA(k) \notin P$ then we have a communication requirement that has to be coherent with $DESC$;
2. 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).
3. It is convenient for performance to group multiple rows into a single call;

Data Management

End of build stage:

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

or, in the C interface,

```
psb_i_t psb_c_dspasb(psb_c_dspmat *a,
        psb_c_descriptor *desc_a);
```

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.

Data Management

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).

Data Management - C Interface

It is possible to put the matrix in the UPDATE state also in the C interface, with

```
psb_i_t psb_c_dsprn(psb_c_dspmat *a,
        psb_c_descriptor *desc_a, _Bool clear);
```

and using the analogous routines for the other data types.

Data Management

Same overall code structure with dense vectors

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

Data Management

Same overall code structure with dense vectors

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

That can be easily translate into the C interface by means of

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

that are available for all the types.

Data Management

Rules of precedence:

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

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 I

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 II

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;

Index mapping

```
call psb_glob_to_loc(iv(:),outv(:),desc,info &
&  [,iact, owned])
call psb_glob_to_loc(v(:),desc,info &
&  [,iact, owned])

call psb_loc_to_glob(iv(:),outv(:),desc,info &
& [,iact])
call psb_loc_to_glob(v(:),desc,info &
& [,iact])
```

Indices not belonging to the calling process are marked with negative numbers on output. `owned=.true.` means treat halo indices as invalid.

Data Query

```
m  = desc%get_global_rows()
n  = desc%get_global_cols()
nr = desc%get_local_rows()
nc = desc%get_local_cols()
ith = psb_cd_get_large_threshold()
call psb_cd_set_large_threshold(ith)

nr = a%get_nrows()
nc = a%get_ncols()
nz = a%get_nzeros()
call a%csget(imin,imax,nz,ia,ja,val,info,&
& [jmin,jmax,iren,append,nzin,rscale,cscale])
asize  =  a%sizeof()
cdsize =  desc%sizeof()
```

I/O from File

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

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

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

Matrix Market different functions for matrices and vectors

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

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

Table of Contents

Preconditioned iterations

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

Mandatory arguments:

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

Preconditioned iterations

Optional arguments:

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

Preconditioned iterations - C Interfaces

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

```
int psb_c_skrylov(const char *method, psb_c_sspmat *ah,
        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_i_t ictxt, 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`.

Advanced Preconditioners

More advanced multilevel preconditioners

- Jacobi, hybrid forward/backward, Gauss-Seidel, block-Jacobi, and additive Schwarz methods
- Algebraic multigrid with smoothed and un-smoothed aggregation
- V-, W-, and K-cycles with exact or approximate coarse level solvers

are available in the library

MLD2P4
MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS
`https://github.com/sfilippone/mld2p4-2`

and will be discussed in a following tutorial.

How to play around

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

How to play around

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

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

Table of Contents

References I

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

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

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

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

References II

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

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

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

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