PSBLAS 3.0-beta User's guide

A reference guide for the Parallel Sparse BLAS library

by Salvatore Filippone
and Alfredo Buttari
University of Rome "Tor Vergata".

December 15th, 2011.

Contents

1 Introduction

The PSBLAS library, developed with the aim to facilitate the parallelization of computationally intensive scientific applications, is designed to address parallel implementation of iterative solvers for sparse linear systems through the distributed memory paradigm. It includes routines for multiplying sparse matrices by dense matrices, solving block diagonal systems with triangular diagonal entries, preprocessing sparse matrices, and contains additional routines for dense matrix operations. The current implementation of PSBLAS addresses a distributed memory execution model operating with message passing.

The PSBLAS library version 3 is implemented in the Fortran 2003 [17] programming language, with reuse and/or adaptation of existing Fortran 77 and Fortran 95 software, plus a handful of C routines.

The use of Fortran 2003 offers a number of advantages over Fortran 95, mostly in the handling of requirements for evolution and adaptation of the library to new computing architectures and integration of new algorithms. For a detailed discussion of our design see [11]; other works discussing advanced programming in Fortran 2003 include [1, 18]; sufficient support for Fortran 2003 is now available from many compilers, including the GNU Fortran compiler from the Free Software Foundation (as of version 4.6).

Previous approaches have been based on mixing Fortran 95, with its support for object-based design, with other languages; these have been advocated by a number of authors, e.g. [16]. Moreover, the Fortran 95 facilities for dynamic memory management and interface overloading greatly enhance the usability of the PSBLAS subroutines. In this way, the library can take care of runtime memory requirements that are quite difficult or even impossible to predict at implementation or compilation time.

The presentation of the PSBLAS library follows the general structure of the proposal for serial Sparse BLAS [8, 9], which in its turn is based on the proposal for BLAS on dense matrices [15, 5, 6].

The applicability of sparse iterative solvers to many different areas causes some terminology problems because the same concept may be denoted through different names depending on the application area. The PSBLAS features presented in this document will be discussed referring to a finite difference discretization of a Partial Differential Equation (PDE). However, the scope of the library is wider than that: for example, it can be applied to finite element discretizations of PDEs, and even to different classes of problems such as nonlinear optimization, for example in optimal control problems.

The design of a solver for sparse linear systems is driven by many conflicting objectives, such as limiting occupation of storage resources, exploiting regularities in the input data, exploiting hardware characteristics of the parallel platform. To achieve an optimal communication to computation ratio on distributed memory machines it is essential to keep the *data locality* as high as possible; this can be done through an appropriate data allocation strategy. The choice of the preconditioner is another very important factor that affects efficiency of the implemented application. Optimal data distribution requirements for a given preconditioner may conflict with distribution requirements of the rest of the solver. Finding the optimal trade-off may be very difficult because it is application dependent. Possible solutions to these problems and other important inputs to the development of the PSBLAS software package

have come from an established experience in applying the PSBLAS solvers to computational fluid dynamics applications.

2 General overview

The PSBLAS library is designed to handle the implementation of iterative solvers for sparse linear systems on distributed memory parallel computers. The system coefficient matrix A must be square; it may be real or complex, nonsymmetric, and its sparsity pattern needs not to be symmetric. The serial computation parts are based on the serial sparse BLAS, so that any extension made to the data structures of the serial kernels is available to the parallel version. The overall design and parallelization strategy have been influenced by the structure of the ScaLAPACK parallel library. The layered structure of the PSBLAS library is shown in figure 1; lower layers of the library indicate an encapsulation relationship with upper layers. The ongoing discussion focuses on the Fortran 2003 layer immediately below the application layer. The serial parts of the computation on each process are executed through calls to the serial sparse BLAS subroutines. In a similar way, the inter-process message exchanges are encapsulated in an applicaiton layer that has been strongly inspired by the Basic Linear Algebra Communication Subroutines (BLACS) library [7]. Usually there is no need to deal directly with MPI; however, in some cases, MPI routines are used directly to improve efficiency. For further details on our communication layer see Sec. 7.

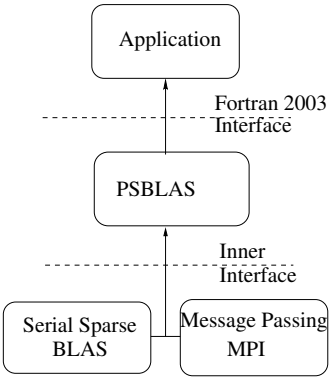

Figure 1: PSBLAS library components hierarchy.

The type of linear system matrices that we address typically arise in the numerical solution of PDEs; in such a context, it is necessary to pay special attention to the structure of the problem from which the application originates. The nonzero pattern of a matrix arising from the discretization of a PDE is influenced by various factors, such as the shape of the domain, the discretization strategy, and the equation/unknown ordering. The matrix itself can be interpreted as the adjacency matrix of the graph associated with the discretization mesh.

The distribution of the coefficient matrix for the linear system is based on the "owner computes" rule: the variable associated to each mesh point is assigned

to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations. This allocation strategy is equivalent to a partition of the discretization mesh into *sub-domains*. Our library supports any distribution that keeps together the coefficients of each matrix row; there are no other constraints on the variable assignment. This choice is consistent with simple data distributions such as `CYCLIC(N)` and `BLOCK`, as well as completely arbitrary assignments of equation indices to processes. In particular it is consistent with the usage of graph partitioning tools commonly available in the literature, e.g. METIS [14]. Dense vectors conform to sparse matrices, that is, the entries of a vector follow the same distribution of the matrix rows.

We assume that the sparse matrix is built in parallel, where each process generates its own portion. We never require that the entire matrix be available on a single node. However, it is possible to hold the entire matrix in one process and distribute it explicitly[1], even though the resulting memory bottleneck would make this option unattractive in most cases.

2.1 Basic Nomenclature

Our computational model implies that the data allocation on the parallel distributed memory machine is guided by the structure of the physical model, and specifically by the discretization mesh of the PDE.

Each point of the discretization mesh will have (at least) one associated equation/variable, and therefore one index. We say that point i *depends* on point j if the equation for a variable associated with i contains a term in j, or equivalently if $a_{ij} \neq 0$. After the partition of the discretization mesh into *sub-domains* assigned to the parallel processes, we classify the points of a given sub-domain as following.

Internal. An internal point of a given domain *depends* only on points of the same domain. If all points of a domain are assigned to one process, then a computational step (e.g., a matrix-vector product) of the equations associated with the internal points requires no data items from other domains and no communications.

Boundary. A point of a given domain is a boundary point if it *depends* on points belonging to other domains.

Halo. A halo point for a given domain is a point belonging to another domain such that there is a boundary point which *depends* on it. Whenever performing a computational step, such as a matrix-vector product, the values associated with halo points are requested from other domains. A boundary point of a given domain is usually a halo point for some other domain[2]; therefore the cardinality of the boundary points set denotes the amount of data sent to other domains.

[1] In our prototype implementation we provide sample scatter/gather routines.

[2] This is the normal situation when the pattern of the sparse matrix is symmetric, which is equivalent to say that the interaction between two variables is reciprocal. If the matrix pattern is non-symmetric we may have one-way interactions, and these could cause a situation in which a boundary point is not a halo point for its neighbour.

Overlap. An overlap point is a boundary point assigned to multiple domains. Any operation that involves an overlap point has to be replicated for each assignment.

Overlap points do not usually exist in the basic data distributions; however they are a feature of Domain Decomposition Schwarz preconditioners which are the subject of related research work [4, 3].

We denote the sets of internal, boundary and halo points for a given subdomain by $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{H}$. Each subdomain is assigned to one process; each process usually owns one subdomain, although the user may choose to assign more than one subdomain to a process. If each process i owns one subdomain, the number of rows in the local sparse matrix is $|\mathcal{I}_i| + |\mathcal{B}_i|$, and the number of local columns (i.e. those for which there exists at least one non-zero entry in the local rows) is $|\mathcal{I}_i| + |\mathcal{B}_i| + |\mathcal{H}_i|$.

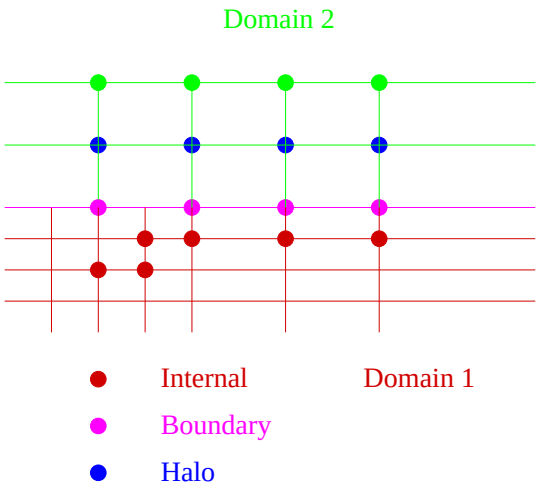

Figure 2: Point classfication.

This classification of mesh points guides the naming scheme that we adopted in the library internals and in the data structures. We explicitly note that "Halo" points are also often called "ghost" points in the literature.

2.2 Library contents

The PSBLAS library consists of various classes of subroutines:

Computational routines comprising:

- Sparse matrix by dense matrix product;
- Sparse triangular systems solution for block diagonal matrices;
- Vector and matrix norms;
- Dense matrix sums;
- Dot products.

Communication routines handling halo and overlap communications;

Data management and auxiliary routines including:

- Parallel environment management
- Communication descriptors allocation;
- Dense and sparse matrix allocation;
- Dense and sparse matrix build and update;
- Sparse matrix and data distribution preprocessing.

Preconditioner routines

Iterative methods a subset of Krylov subspace iterative methods

The following naming scheme has been adopted for all the symbols internally defined in the PSBLAS software package:

- all symbols (i.e. subroutine names, data types...) are prefixed by `psb_`
- all data type names are suffixed by `_type`
- all constants are suffixed by `_`
- all top-level subroutine names follow the rule `psb_xxname` where `xx` can be either:
  - `ge`: the routine is related to dense data,
  - `sp`: the routine is related to sparse data,
  - `cd`: the routine is related to communication descriptor (see 3).

  For example the `psb_geins`, `psb_spins` and `psb_cdins` perform the same action (see 6) on dense matrices, sparse matrices and communication descriptors respectively. Interface overloading allows the usage of the same subroutine names for both real and complex data.

In the description of the subroutines, arguments or argument entries are classified as:

global For input arguments, the value must be the same on all processes participating in the subroutine call; for output arguments the value is guaranteed to be the same.

local Each process has its own value(s) independently.

To finish our general description, we define a version string with the constant

`psb_version_string_`

whose current value is `3.0.0`

2.3 Application structure

The main underlying principle of the PSBLAS library is that the library objects are created and exist with reference to a discretized space to which there corresponds an index space and a matrix sparsity pattern. As an example, consider a cell-centered finite-volume discretization of the Navier-Stokes equations on a simulation domain; the index space $1 \ldots n$ is isomorphic to the set of cell centers, whereas the pattern of the associated linear system matrix is isomorphic to the adjacency graph imposed on the discretization mesh by the discretization stencil.

Thus the first order of business is to establish an index space, and this is done with a call to `psb_cdall` in which we specify the size of the index space n and the allocation of the elements of the index space to the various processes making up the MPI (virtual) parallel machine.

The index space is partitioned among processes, and this creates a mapping from the "global" numbering $1 \ldots n$ to a numbering "local" to each process; each process i will own a certain subset $1 \ldots n_{\mathrm{row}_i}$, each element of which corresponds to a certain element of $1 \ldots n$. The user does not set explicitly this mapping; when the application needs to indicate to which element of the index space a certain item is related, such as the row and column index of a matrix coefficient, it does so in the "global" numbering, and the library will translate into the appropriate "local" numbering.

For a given index space $1 \ldots n$ there are many possible associated topologies, i.e. many different discretization stencils; thus the description of the index space is not completed until the user has defined a sparsity pattern, either explicitly through `psb_cdins` or implicitly through `psb_spins`. The descriptor is finalized with a call to `psb_cdasb` and a sparse matrix with a call to `psb_spasb`. After `psb_cdasb` each process i will have defined a set of "halo" (or "ghost") indices $n_{\mathrm{row}_i} + 1 \ldots n_{\mathrm{col}_i}$, denoting elements of the index space that are *not* assigned to process i; however the variables associated with them are needed to complete computations associated with the sparse matrix A, and thus they have to be fetched from (neighbouring) processes. The descriptor of the index space is built exactly for the purpose of properly sequencing the communication steps required to achieve this objective.

A simple application structure will walk through the index space allocation, matrix/vector creation and linear system solution as follows:

1. Initialize parallel environment with `psb_init`
2. Initialize index space with `psb_cdall`
3. Allocate sparse matrix and dense vectors with `psb_spall` and `psb_geall`
4. Loop over all local rows, generate matrix and vector entries, and insert them with `psb_spins` and `psb_geins`
5. Assemble the various entities:
   (a) `psb_cdasb`
   (b) `psb_spasb`
   (c) `psb_geasb`

6. Choose the preconditioner to be used with `psb_precset` and build it with `psb_precbld`

7. Call the iterative method of choice, e.g. `psb_bicgstab`

This is the structure of the sample program `test/pargen/ppde.f90`.

For a simulation in which the same discretization mesh is used over multiple time steps, the following structure may be more appropriate:

1. Initialize parallel environment with `psb_init`

2. Initialize index space with `psb_cdall`

3. Loop over the topology of the discretization mesh and build the descriptor with `psb_cdins`

4. Assemble the descriptor with `psb_cdasb`

5. Allocate the sparse matrices and dense vectors with `psb_spall` and `psb_geall`

6. Loop over the time steps:

   (a) If after first time step, reinitialize the sparse matrix with `psb_sprn`; also zero out the dense vectors;

   (b) Loop over the mesh, generate the coefficients and insert/update them with `psb_spins` and `psb_geins`

   (c) Assemble with `psb_spasb` and `psb_geasb`

   (d) Choose and build preconditioner with `psb_precset` and `psb_precbld`

   (e) Call the iterative method of choice, e.g. `psb_bicgstab`

The insertion routines will be called as many times as needed; they only need to be called on the data that is actually allocated to the current process, i.e. each process generates its own data.

In principle there is no specific order in the calls to `psb_spins`, nor is there a requirement to build a matrix row in its entirety before calling the routine; this allows the application programmer to walk through the discretization mesh element by element, generating the main part of a given matrix row but also contributions to the rows corresponding to neighbouring elements.

From a functional point of view it is even possible to execute one call for each nonzero coefficient; however this would have a substantial computational overhead. It is therefore advisable to pack a certain amount of data into each call to the insertion routine, say touching on a few tens of rows; the best performng value would depend on both the architecture of the computer being used and on the problem structure. At the opposite extreme, it would be possible to generate the entire part of a coefficient matrix residing on a process and pass it in a single call to `psb_spins`; this, however, would entail a doubling of memory occupation, and thus would be almost always far from optimal.

2.4 Programming model

The PSBLAS librarary is based on the Single Program Multiple Data (SPMD) programming model: each process participating in the computation performs the same actions on a chunk of data. Parallelism is thus data-driven.

Because of this structure, many subroutines coordinate their action across the various processes, thus providing an implicit synchronization point, and therefore *must* be called simultaneously by all processes participating in the computation. This is certainly true for the data allocation and assembly routines, for all the computational routines and for some of the tools routines.

However there are many cases where no synchronization, and indeed no communication among processes, is implied; for instance, all the routines in sec. 3.5 are only acting on the local data structures, and thus may be called independently. The most important case is that of the coefficient insertion routines: since the number of coefficients in the sparse and dense matrices varies among the processors, and since the user is free to choose an arbitrary order in builiding the matrix entries, these routines cannot imply a synchronization.

Throughout this user's guide each subroutine will be clearly indicated as:

Synchronous: must be called simultaneously by all the processes in the relevant communication context;

Asynchronous: may be called in a totally independent manner.

3 Data Structures and Classes

In this chapter we illustrate the data structures used for definition of routines interfaces. They include data structures for sparse matrices, communication descriptors and preconditioners.

All the data types and the basic subroutine interfaces related to descriptors and sparse matrices are defined in the module `psb_base_mod`; this will have to be included by every user subroutine that makes use of the library. The preconditioners are defined in the module `psb_prec_mod`

Real and complex data types are parametrized with a kind type defined in the library as follows:

psb_spk_ Kind parameter for short precision real and complex data; corresponds to a `REAL` declaration and is normally 4 bytes.

psb_dpk_ Kind parameter for long precision real and complex data; corresponds to a `DOUBLE PRECISION` declaration and is normally 8 bytes.

psb_long_int_k_ Kind parameter for long integers, used by the `psb_sizeof` utility.

Together with the classes attributes we also discuss their methods. Most methods detailed here only act on the local variable, i.e. their action is purely local and asynchronous unless otherwise stated. The list of methods here is not completely exhaustive; many methods, especially those that alter the contents of the various objects, are usually not needed by the end-user, and therefore are described in the developer's documentation.

3.1 Descriptor data structure

All the general matrix informations and elements to be exchanged among processes are stored within a data structure of the type `psb_desc_type`. Every structure of this type is associated with a discretization pattern and enables data communications and other operations that are necessary for implementing the various algorithms of interest to us.

The data structure itself `psb_desc_type` can be treated as an opaque object handled via the tools routines of Sec. 6 and 3.5; nevertheless we include here a description for the curious reader.

First we describe the `psb_indx_map` type. This is a data structure that keeps track of a certain number of basic issues such as:

- The value of the communication/MPI context;
- The number of indices in the index space, i.e. global number of rows and columns of a sparse matrix;
- The local set of indices, including:
  - The number of local indices (and local rows);
  - The number of halo indices (and therefore local columns);
  - The global indices corresponding to the local ones.

There are many different schemes for storing these data; therefore there are a number of types extending the base one, and the descriptor structure holds a polymorphic object whose dynamic type can be any of the extended types. The methods associated with this data type answer the following queries:

- For a given set of local indices, find the corresponding indices in the global numbering;
- For a given set of global indices, find the corresponding indices in the local numbering, if any, or return an invalid
- Add a global index to the set of halo indices;
- Find the process owner of each member of a set of global indices.

All methods but the last are purely local; the last method potentially requires communication among processes, and thus is a synchronous method. The choice of a specific dynamic type for the index map is made at the time the descriptor is initially allocated, according to the mode of initialization (see also 6).

The descriptor contents are as follows:

indxmap A polymorphic variable of a type that is any extension of the indx_map type described above.

halo_index A list of the halo and boundary elements for the current process to be exchanged with other processes; for each processes with which it is necessary to communicate:

1. Process identifier;
2. Number of points to be received;
3. Indices of points to be received;
4. Number of points to be sent;
5. Indices of points to be sent;

The list may contain an arbitrary number of groups; its end is marked by a -1.
Specified as: an allocatable integer array of rank one.

ext_index A list of element indices to be exchanged to implement the mapping between a base descriptor and a descriptor with overlap.

ovrlap_index A list of the overlap elements for the current process, organized in groups like the previous vector:

1. Process identifier;
2. Number of points to be received;
3. Indices of points to be received;
4. Number of points to be sent;
5. Indices of points to be sent;

The list may contain an arbitrary number of groups; its end is marked by a -1.
Specified as: an allocatable integer array of rank one.

ovr_mst_idx A list to retrieve the value of each overlap element from the respective master process.
Specified as: an allocatable integer array of rank one.

ovrlap_elem For all overlap points belonging to th ecurrent process:

1. Overlap point index;
2. Number of processes sharing that overlap points;
3. Index of a "master" process:

Specified as: an allocatable integer array of rank two.

bnd_elem A list of all boundary points, i.e. points that have a connection with other processes.

The Fortran 2003 declaration for `psb_desc_type` structures is as follows:

```
type psb_desc_type
    class(psb_indx_map), allocatable :: indxmap
    integer, allocatable  :: halo_index(:)
    integer, allocatable  :: ext_index(:)
    integer, allocatable  :: ovrlap_index(:)
    integer, allocatable  :: ovrlap_elem(:,:)
    integer, allocatable  :: ovr_mst_idx(:)
    integer, allocatable  :: bnd_elem(:)
end type psb_desc_type
```

Figure 3: The PSBLAS defined data type that contains the communication descriptor.

A communication descriptor associated with a sparse matrix has a state, which can take the following values:

Build: State entered after the first allocation, and before the first assembly; in this state it is possible to add communication requirements among different processes.

Assembled: State entered after the assembly; computations using the associated sparse matrix, such as matrix-vector products, are only possible in this state.

3.1.1 Methods

get_local_rows — Get number of local rows

```
nr = desc%get_local_rows()
```

Type: Asynchronous.

On Entry

desc the communication descriptor.
Scope: **local**.

On Return

Function value The number of local rows, i.e. the number of rows owned by the current process; as explained in 1, it is equal to $|\mathcal{I}_i| + |\mathcal{B}_i|$. The returned value is specific to the calling process.

get_local_cols — Get number of local cols

```
nc = desc%get_local_cols()
```

On Entry

Type: Asynchronous.

desc the communication descriptor.
Scope: **local**.

On Return

Function value The number of local cols, i.e. the number of indices used by the current process, including both local and halo indices; as explained in 1, it is equal to $|\mathcal{I}_i| + |\mathcal{B}_i| + |\mathcal{H}_i|$. The returned value is specific to the calling process.

get_global_rows — Get number of global rows

```
nr = desc%get_global_rows()
```

On Entry

Type: Asynchronous.

desc the communication descriptor.
Scope: **local**.

On Return

Function value The number of global rows, i.e. the size of the global index space.

get_global_cols — Get number of global cols

```
nr = desc%get_global_cols()
```

Type: Asynchronous.

On Entry

desc the communication descriptor.
Scope: **local**.

On Return

Function value The number of global cols; usually this is equal to the number of global rows.

get_context—Get communication context

```
ictxt = desc%get_context()
```

Type: Asynchronous.

On Entry

desc the communication descriptor.
Scope: **local**.

On Return

Function value The communication context.

psb_cd_get_large_threshold — Get threshold for index mapping switch

```
ith = psb_cd_get_large_threshold()
```

Type: Asynchronous.

On Return

Function value The current value for the size threshold.

psb_cd_set_large_threshold — Set threshold for index mapping switch

```
call psb_cd_set_large_threshold(ith)
```

Type: Synchronous.

On Entry

ith the new threshold for communication descriptors.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer value greater than zero.

Note: the threshold value is only queried by the library at the time a call to `psb_cdall` is executed, therefore changing the threshold has no effect on communication descriptors that have already been initialized. Moreover the threshold must have the same value on all processes.

3.1.2 Named Constants

psb_none_ Generic no-op;

psb_nohalo_ Do not fetch halo elements;

psb_halo_ Fetch halo elements from neighbouring processes;

psb_sum_ Sum overlapped elements

psb_avg_ Average overlapped elements

psb_comm_halo_ Exchange data based on the `halo_index` list;

psb_comm_ext_ Exchange data based on the `ext_index` list;

psb_comm_ovr_ Exchange data based on the `ovrlap_index` list;

psb_comm_mov_ Exchange data based on the `ovr_mst_idx` list;

3.2 Sparse Matrix class

The `psb_spmat_type` class contains all information about the local portion of the sparse matrix and its storage mode. Its design is based on the STATE design pattern [13] as detailed in [11]; the type declaration is shown in figure 4 where `T` is a placeholder for the data type and precision variants

S Single precision real;

D Double precision real;

C Single precision complex;

Z Double precision complex.

The actual data is contained in the polymorphic component `a%a`; its specific layout can be chosen dynamically among the predefined types, or an entirely new storage layout can be implemented and passed to the library at runtime via the `psb_spasb` routine. The following very common formats are precompiled in PSBLAS and thus are always available:

psb_T_coo_sparse_mat Coordinate storage;

psb_T_csr_sparse_mat Compressed storage by rows;

```
type :: psb_Tspmat_type

  class(psb_T_base_sparse_mat), allocatable  :: a

end type  psb_Tspmat_type
```

Figure 4: The PSBLAS defined data type that contains a sparse matrix.

psb_T_csc_sparse_mat Compressed storage by columns;

The inner sparse matrix has an associated state, which can take the following values:

Build: State entered after the first allocation, and before the first assembly; in this state it is possible to add nonzero entries.

Assembled: State entered after the assembly; computations using the sparse matrix, such as matrix-vector products, are only possible in this state;

Update: State entered after a reinitalization; this is used to handle applications in which the same sparsity pattern is used multiple times with different coefficients. In this state it is only possible to enter coefficients for already existing nonzero entries.

The only storage variant supporting the build state is COO; all other variants are obtained by conversion to/from it.

3.2.1 Methods

get_nrows — Get number of rows in a sparse matrix

```
nr = a%get_nrows()
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value The number of rows of sparse matrix `a`.

get_ncols — Get number of columns in a sparse matrix

```
nc = a%get_ncols()
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value The number of columns of sparse matrix `a`.

get_nnzeros — Get number of nonzero elements in a sparse matrix

```
nz = a%get_nnzeros()
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value The number of nonzero elements stored in sparse matrix `a`.

Notes

1. The function value is specific to the storage format of matrix `a`; some storage formats employ padding, thus the returned value for the same matrix may be different for different storage choices.

get_size — Get maximum number of nonzero elements in a sparse matrix

```
maxnz = a%get_size()
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value The maximum number of nonzero elements that can be stored in sparse matrix `a` using its current memory allocation.

sizeof — Get memory occupation in bytes of a sparse matrix

```
memory_size = a%sizeof()
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value The memory occupation in bytes.

get_fmt — Short description of the dynamic type

```
write(*,*) a%get_fmt()
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value A short string describing the dynamic type of the matrix. Predefined values include `NULL`, `COO`, `CSR` and `CSC`.

is_bld, is_upd, is_asb — Status check

```
if (a%is_bld()) then
if (a%is_upd()) then
if (a%is_asb()) then
```

Type: Asynchronous.

On Entry

a the sparse matrix
Scope: **local**

On Return

Function value A `logical` value indicating whether the matrix is in the Build, Update or Assembled state, respectively.

3.2.2 Named Constants

psb_dupl_ovwrt_ Duplicate coefficients should be overwritten (i.e. ignore duplications)

psb_dupl_add_ Duplicate coefficients should be added;

psb_dupl_err_ Duplicate coefficients should trigger an error conditino

psb_upd_dflt_ Default update strategy for matrix coefficients;

psb_upd_srch_ Update strategy based on search into the data structure;

psb_upd_perm_ Update strategy based on additional permutation data (see tools routine description).

3.3 Dense Vector Data Structure

The `psb_vect_type` data structure encapsulates the dense vectors in a way similar to sparse matrices. The user will not, in general, access the vector components directly, but rather via the routines of sec. 6. Among other simple things, we define here an extraction method that can be used to get a full copy of the part of the vector stored on the local process.

The type declaration is shown in figure 5 where `T` is a placeholder for the data type and precision variants

I Integer;

S Single precision real;

D Double precision real;

C Single precision complex;

Z Double precision complex.

The actual data is contained in the polymorphic component `a%a`; its specific layout can be chosen dynamically among the predefined types, or an entirely new storage layout can be implemented and passed to the library at runtime via the `psb_spasb` routine.

```
type psb_T_base_vect_type
  TYPE(KIND_), allocatable :: v(:)
end type psb_T_base_vect_type

type psb_T_vect_type
  class(psb_T_base_vect_type), allocatable :: v
end type  psb_T_vect_type
```

Figure 5: The PSBLAS defined data type that contains a dense vector..

3.3.1 Methods

get_nrows — Get number of rows in a dense vector

```
nr = v%get_nrows()
```

Type: Asynchronous.

On Entry

v the dense vector
Scope: **local**

On Return

Function value The number of rows of sparse matrix `a`.

get_ncols — Get number of columns in a sparse matrix

sizeof — Get memory occupation in bytes of a dense vector matrix

```
memory_size = v%sizeof()
```

Type: Asynchronous.

On Entry

v the dense vector
Scope: **local**

On Return

Function value The memory occupation in bytes.

get_vect — Get a copy of the vector contents

```
extv = v%get_vect()
```

Type: Asynchronous.

On Entry

v the dense vector
Scope: **local**

On Return

Function value An allocatable array holding a copy of the dense vector contents.

3.4 Preconditioner data structure

Our base library offers support for simple well known preconditioners like Diagonal Scaling or Block Jacobi with incomplete factorization ILU(0).

A preconditioner is held in the `psb_prec_type` data structure reported in figure 6. The `psb_prec_type` data type may contain a simple preconditioning matrix with the associated communication descriptor.The values contained in the `iprcparm` and `rprcparm` define tha type of preconditioner along with all the parameters related to it; thus, `iprcparm` and `rprcparm` define how the other records have to be interpreted. This data structure is the basis of more complex preconditioning strategies, which are the subject of further research.

3.5 Data structure Methods

```
type psb_sprec_type
  type(psb_sspmat_type), allocatable :: av(:)
  real(psb_spk_), allocatable        :: d(:)
  type(psb_desc_type)                :: desc_data
  integer, allocatable               :: iprcparm(:)
  real(psb_spk_), allocatable        :: rprcparm(:)
  integer, allocatable               :: perm(:),  invperm(:)
  integer                            :: prec, base_prec
end type psb_sprec_type

type psb_dprec_type
  type(psb_dspmat_type), allocatable :: av(:)
  real(psb_dpk_), allocatable        :: d(:)
  type(psb_desc_type)                :: desc_data
  integer, allocatable               :: iprcparm(:)
  real(psb_dpk_), allocatable        :: rprcparm(:)
  integer, allocatable               :: perm(:),  invperm(:)
  integer                            :: prec, base_prec
end type psb_dprec_type

type psb_cprec_type
  type(psb_cspmat_type), allocatable :: av(:)
  complex(psb_spk_), allocatable     :: d(:)
  type(psb_desc_type)                :: desc_data
  integer, allocatable               :: iprcparm(:)
  real(psb_spk_), allocatable        :: rprcparm(:)
  integer, allocatable               :: perm(:),  invperm(:)
  integer                            :: prec, base_prec
end type psb_cprec_type

type psb_zprec_type
  type(psb_zspmat_type), allocatable :: av(:)
  complex(psb_dpk_), allocatable     :: d(:)
  type(psb_desc_type)                :: desc_data
  integer, allocatable               :: iprcparm(:)
  real(psb_dpk_), allocatable        :: rprcparm(:)
  integer, allocatable               :: perm(:),  invperm(:)
  integer                            :: prec, base_prec
end type psb_zprec_type
```

Figure 6: The PSBLAS defined data type that contains a preconditioner.

4 Computational routines

psb_geaxpby — General Dense Matrix Sum

This subroutine is an interface to the computational kernel for dense matrix sum:

$$y \leftarrow \alpha\, x + \beta y$$

```
call psb_geaxpby(alpha, x, beta, y, desc_a, info)
```

x, y, α, β	**Subroutine**
Short Precision Real	psb_geaxpby
Long Precision Real	psb_geaxpby
Short Precision Complex	psb_geaxpby
Long Precision Complex	psb_geaxpby

Table 1: Data types

Type: Synchronous.

On Entry

alpha the scalar α.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a number of the data type indicated in Table 1.

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 1. The rank of x must be the same of y.

beta the scalar β.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a number of the data type indicated in Table 1.

y the local portion of the global dense matrix y.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array containing numbers of the type indicated in Table 1. The rank of y must be the same of x.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

y the local portion of result submatrix y.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array containing numbers of the type indicated in Table 1.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_gedot — Dot Product

This function computes dot product between two vectors x and y.
If x and y are real vectors it computes dot-product as:

$$dot \leftarrow x^T y$$

Else if x and y are complex vectors then it computes dot-product as:

$$dot \leftarrow x^H y$$

```
psb_gedot(x, y, desc_a, info)
```

dot, *x*, *y*	**Function**
Short Precision Real	psb_gedot
Long Precision Real	psb_gedot
Short Precision Complex	psb_gedot
Long Precision Complex	psb_gedot

Table 2: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: an array of rank one or two containing numbers of type specified in Table 2. The rank of x must be the same of y.

y the local portion of global dense matrix y.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: an array of rank one or two containing numbers of type specified in Table 2. The rank of y must be the same of x.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

Function value is the dot product of subvectors x and y.
Scope: **global**
Specified as: a number of the data type indicated in Table 2.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_gedots — Generalized Dot Product

This subroutine computes a series of dot products among the columns of two dense matrices x and y:

$$res(i) \leftarrow x(:,i)^T y(:,i)$$

If the matrices are complex, then the usual convention applies, i.e. the conjugate transpose of x is used. If x and y are of rank one, then res is a scalar, else it is a rank one array.

```
call psb_gedots(res, x, y, desc_a, info)
```

res, *x*, *y*	**Subroutine**
Short Precision Real	psb_gedots
Long Precision Real	psb_gedots
Short Precision Complex	psb_gedots
Long Precision Complex	psb_gedots

Table 3: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: an array of rank one or two containing numbers of type specified in Table 3. The rank of x must be the same of y.

y the local portion of global dense matrix y.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: an array of rank one or two containing numbers of type specified in Table 3. The rank of y must be the same of x.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

On Return

res is the dot product of subvectors x and y.
Scope: **global**
Intent: **out**.
Specified as: a number or a rank-one array of the data type indicated in Table 2.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_geamax — Infinity-Norm of Vector

This function computes the infinity-norm of a vector x.
If x is a real vector it computes infinity norm as:

$$amax \leftarrow \max_i |x_i|$$

else if x is a complex vector then it computes infinity-norm as:

$$amax \leftarrow \max_i (|re(x_i)| + |im(x_i)|)$$

```
psb_geamax(x, desc_a, info)
```

amax	*x*	**Function**
Short Precision Real	Short Precision Real	psb_geamax
Long Precision Real	Long Precision Real	psb_geamax
Short Precision Real	Short Precision Complex	psb_geamax
Long Precision Real	Long Precision Complex	psb_geamax

Table 4: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 4.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

On Return

Function value is the infinity norm of subvector x.
Scope: **global**
Specified as: a long precision real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_geamaxs — Generalized Infinity Norm

This subroutine computes a series of infinity norms on the columns of a dense matrix x:

$$res(i) \leftarrow \max_k |x(k,i)|$$

```
call psb_geamaxs(res, x, desc_a, info)
```

res	*x*	**Subroutine**
Short Precision Real	Short Precision Real	psb_geamaxs
Long Precision Real	Long Precision Real	psb_geamaxs
Short Precision Real	Short Precision Complex	psb_geamaxs
Long Precision Real	Long Precision Complex	psb_geamaxs

Table 5: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 5.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

On Return

res is the infinity norm of the columns of x.
Scope: **global**
Intent: **out**.
Specified as: a number or a rank-one array of long precision real numbers.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_geasum — 1-Norm of Vector

This function computes the 1-norm of a vector x.
If x is a real vector it computes 1-norm as:

$$asum \leftarrow \|x_i\|$$

else if x is a vector then it computes 1-norm as:

$$asum \leftarrow \|re(x)\|_1 + \|im(x)\|_1$$

```
psb_geasum(x, desc_a, info)
```

asum	*x*	**Function**
Short Precision Real	Short Precision Real	psb_geasum
Long Precision Real	Long Precision Real	psb_geasum
Short Precision Real	Short Precision Complex	psb_geasum
Long Precision Real	Long Precision Complex	psb_geasum

Table 6: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 6.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

Function value is the 1-norm of vector x.
Scope: **global**
Specified as: a long precision real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_geasums — Generalized 1-Norm of Vector

This subroutine computes a series of 1-norms on the columns of a dense matrix x:

$$res(i) \leftarrow \max_k |x(k,i)|$$

This function computes the 1-norm of a vector x.
If x is a real vector it computes 1-norm as:

$$res(i) \leftarrow \|x_i\|$$

else if x is a complex vector then it computes 1-norm as:

$$res(i) \leftarrow \|re(x)\|_1 + \|im(x)\|_1$$

```
call psb_geasums(res, x, desc_a, info)
```

res	*x*	**Subroutine**
Short Precision Real	Short Precision Real	psb_geasums
Long Precision Real	Long Precision Real	psb_geasums
Short Precision Real	Short Precision Complex	psb_geasums
Long Precision Real	Long Precision Complex	psb_geasums

Table 7: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 7.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

On Return

res contains the 1-norm of (the columns of) x.
Scope: **global**
Intent: **out**.
Short as: a long precision real number. Specified as: a long precision real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_genrm2 — 2-Norm of Vector

This function computes the 2-norm of a vector x.
If x is a double precision real vector it computes 2-norm as:

$$nrm2 \leftarrow \sqrt{x^T x}$$

else if x is double precision complex vector then it computes 2-norm as:

$$nrm2 \leftarrow \sqrt{x^H x}$$

$nrm2$	x	**Function**
Short Precision Real	Short Precision Real	psb_genrm2
Long Precision Real	Long Precision Real	psb_genrm2
Short Precision Real	Short Precision Complex	psb_genrm2
Long Precision Real	Long Precision Complex	psb_genrm2

Table 8: Data types

```
psb_genrm2(x, desc_a, info)
```

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 8.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

Function Value is the 2-norm of subvector x.
Scope: **global**
Type: **required**
Specified as: a long precision real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_genrm2s — Generalized 2-Norm of Vector

This subroutine computes a series of 2-norms on the columns of a dense matrix x:

$$res(i) \leftarrow \|x(:,i)\|_2$$

```
call psb_genrm2s(res, x, desc_a, info)
```

res	*x*	**Subroutine**
Short Precision Real	Short Precision Real	psb_genrm2s
Long Precision Real	Long Precision Real	psb_genrm2s
Short Precision Real	Short Precision Complex	psb_genrm2s
Long Precision Real	Long Precision Complex	psb_genrm2s

Table 9: Data types

Type: Synchronous.

On Entry

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 9.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

On Return

res contains the 1-norm of (the columns of) x.
Scope: **global**
Intent: **out**.
Specified as: a long precision real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_spnrmi — Infinity Norm of Sparse Matrix

This function computes the infinity-norm of a matrix A:

$$nrmi \leftarrow \|A\|_\infty$$

where:

A represents the global matrix A

A	**Function**
Short Precision Real	psb_spnrmi
Long Precision Real	psb_spnrmi
Short Precision Complex	psb_spnrmi
Long Precision Complex	psb_spnrmi

Table 10: Data types

```
psb_spnrmi(A, desc_a, info)
```

Type: Synchronous.

On Entry

a the local portion of the global sparse matrix A.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_spmat_type.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

Function value is the infinity-norm of sparse submatrix A.
Scope: **global**
Specified as: a long precision real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_spmm — Sparse Matrix by Dense Matrix Product

This subroutine computes the Sparse Matrix by Dense Matrix Product:

$$y \leftarrow \alpha P_r A P_c x + \beta y \tag{1}$$
$$y \leftarrow \alpha P_r A^T P_c x + \beta y \tag{2}$$
$$y \leftarrow \alpha P_r A^H P_c x + \beta y \tag{3}$$

where:

x is the global dense submatrix $x_{:,:}$

y is the global dense submatrix $y_{:,:}$

A is the global sparse submatrix A

P_r, P_c are the permutation matrices.

A, x, y, α, β	**Subroutine**
Short Precision Real	psb_spmm
Long Precision Real	psb_spmm
Short Precision Complex	psb_spmm
Long Precision Complex	psb_spmm

Table 11: Data types

```
call psb_spmm(alpha, a, x, beta, y, desc_a, info)
call psb_spmm(alpha, a, x, beta, y,desc_a, info, &
             & trans, work)
```

Type: Synchronous.

On Entry

alpha the scalar α.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a number of the data type indicated in Table 11.

a the local portion of the sparse matrix A.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_spmat_type.

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 11. The rank of x must be the same of y.

beta the scalar β.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a number of the data type indicated in Table 11.

y the local portion of global dense matrix y.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array containing numbers of type specified in Table 11. The rank of y must be the same of x.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

trans indicate what kind of operation to perform.

trans = N the operation is specified by equation 1

trans = T the operation is specified by equation 2

trans = C the operation is specified by equation 3

Scope: **global**
Type: **optional**
Intent: **in**.
Default: $trans = N$
Specified as: a character variable.

work work array.
Scope: **local**
Type: **optional**
Intent: **inout**.
Specified as: a rank one array of the same type of x and y with the TARGET attribute.

On Return

y the local portion of result submatrix y.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: an array of rank one or two containing numbers of type specified in Table 11.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_spsm — Triangular System Solve

This subroutine computes the Triangular System Solve:

$$
\begin{aligned}
y &\leftarrow \alpha P_r T^{-1} P_c x + \beta y \\
y &\leftarrow \alpha D P_r T^{-1} P_c x + \beta y \\
y &\leftarrow \alpha P_r T^{-1} P_c D x + \beta y \\
y &\leftarrow \alpha P_r T^{-T} P_c x + \beta y \\
y &\leftarrow \alpha D P_r T^{-T} P_c x + \beta y \\
y &\leftarrow \alpha P_r T^{-T} P_c D x + \beta y \\
y &\leftarrow \alpha P_r T^{-H} P_c x + \beta y \\
y &\leftarrow \alpha D P_r T^{-H} P_c x + \beta y \\
y &\leftarrow \alpha P_r T^{-H} P_c D x + \beta y
\end{aligned}
$$

where:

x is the global dense submatrix $x_{:,:}$

y is the global dense submatrix $y_{:,:}$

T is the global sparse block triangular submatrix T

D is the scaling diagonal matrix.

P_r, P_c are the permutation matrices.

```
call psb_spsm(alpha, t, x, beta, y, desc_a, info)
call psb_spsm(alpha, t, x, beta, y, desc_a, info,&
             & trans, unit, choice, diag, work)
```

T, x, y, D, α, β	**Subroutine**
Short Precision Real	psb_spsm
Long Precision Real	psb_spsm
Short Precision Complex	psb_spsm
Long Precision Complex	psb_spsm

Table 12: Data types

Type: Synchronous.

On Entry

alpha the scalar α.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a number of the data type indicated in Table 12.

t the global portion of the sparse matrix T.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data type specified in § 3.

x the local portion of global dense matrix x.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of type specified in Table 12. The rank of x must be the same of y.

beta the scalar β.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a number of the data type indicated in Table 12.

y the local portion of global dense matrix y.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array containing numbers of type specified in Table 12. The rank of y must be the same of x.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type `psb_desc_type`.

trans specify with *unitd* the operation to perform.

trans = 'N' the operation is with no transposed matrix

trans = 'T' the operation is with transposed matrix.

trans = 'C' the operation is with conjugate transposed matrix.

Scope: **global**
Type: **optional**
Intent: **in**.
Default: $trans = N$
Specified as: a character variable.

unitd specify with *trans* the operation to perform.

unitd = 'U' the operation is with no scaling

unitd = 'L' the operation is with left scaling

unitd = 'R' the operation is with right scaling.

Scope: **global**
Type: **optional**
Intent: **in**.
Default: $unitd = U$
Specified as: a character variable.

choice specifies the update of overlap elements to be performed on exit:

`psb_none_`

`psb_sum_`

`psb_avg_`

`psb_square_root_`

Scope: **global**
Type: **optional**
Intent: **in**.
Default: `psb_avg_`
Specified as: an integer variable.

diag the diagonal scaling matrix.
Scope: **local**
Type: **optional**
Intent: **in**.
Default: $diag(1) = 1(noscaling)$
Specified as: a rank one array containing numbers of the type indicated in Table 12.

work a work array.
Scope: **local**
Type: **optional**
Intent: **inout**.
Specified as: a rank one array of the same type of x with the TARGET attribute.

On Return

y the local portion of global dense matrix y.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: an array of rank one or two containing numbers of type specified in Table 12.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

5 Communication routines

The routines in this chapter implement various global communication operators on vectors associated with a discretization mesh. For auxiliary communication routines not tied to a discretization space see 6.

psb_halo — Halo Data Communication

These subroutines gathers the values of the halo elements, and (optionally) scale the result:

$$x \leftarrow \alpha x$$

where:

x is a global dense submatrix.

α, x	**Subroutine**
Integer	psb_halo
Short Precision Real	psb_halo
Long Precision Real	psb_halo
Short Precision Complex	psb_halo
Long Precision Complex	psb_halo

Table 13: Data types

```
call psb_halo(x, desc_a, info)
call psb_halo(x, desc_a, info, alpha, work, data)
```

Type: Synchronous.

On Entry

x global dense matrix x.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array with the TARGET attribute containing numbers of type specified in Table 13.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

alpha the scalar α.
Scope: **global**
Type: **optional**
Intent: **in**.
Default: $alpha = 1$
Specified as: a number of the data type indicated in Table 13.

work the work array.
Scope: **local**
Type: **optional**
Intent: **inout**.
Specified as: a rank one array of the same type of x with the POINTER attribute.

data index list selector.
Scope: **global**
Type: **optional**
Specified as: an integer. Values:`psb_comm_halo_`,`psb_comm_mov_`, `psb_comm_ext_`, default: `psb_comm_halo_`. Chooses the index list on which to base the data exchange.

On Return

x global dense result matrix x.
Scope: **local**
Type: **required**
Intent: **inout**.
Returned as: a rank one or two array containing numbers of type specified in Table 13.

info the local portion of result submatrix y.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value that contains an error code.

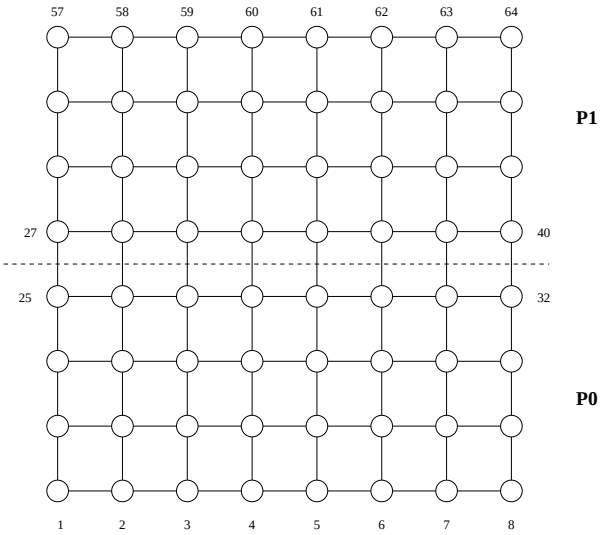

Figure 7: Sample discretization mesh.

Usage Example Consider the discretization mesh depicted in fig. 7, partitioned among two processes as shown by the dashed line; the data distribution is such that each process will own 32 entries in the index space, with a halo made of 8 entries placed at local indices 33 through 40. If process 0 assigns an initial value of 1 to its entries in the x vector, and process 1 assigns a value of 2, then after a call to `psb_halo` the contents of the local vectors will be the following:

	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
9	9	1.0	9	41	2.0
10	10	1.0	10	42	2.0
11	11	1.0	11	43	2.0
12	12	1.0	12	44	2.0
13	13	1.0	13	45	2.0
14	14	1.0	14	46	2.0
15	15	1.0	15	47	2.0
16	16	1.0	16	48	2.0
17	17	1.0	17	49	2.0
18	18	1.0	18	50	2.0
19	19	1.0	19	51	2.0
20	20	1.0	20	52	2.0
21	21	1.0	21	53	2.0
22	22	1.0	22	54	2.0
23	23	1.0	23	55	2.0
24	24	1.0	24	56	2.0
25	25	1.0	25	57	2.0
26	26	1.0	26	58	2.0
27	27	1.0	27	59	2.0
28	28	1.0	28	60	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

psb_ovrl — Overlap Update

These subroutines applies an overlap operator to the input vector:

$$x \leftarrow Qx$$

where:

x is the global dense submatrix x

Q is the overlap operator; it is the composition of two operators P_a and P^T.

x	**Subroutine**
Short Precision Real	psb_ovrl
Long Precision Real	psb_ovrl
Short Precision Complex	psb_ovrl
Long Precision Complex	psb_ovrl

Table 14: Data types

```
call psb_ovrl(x, desc_a, info)
call psb_ovrl(x, desc_a, info, update=update_type, work=work)
```

Type: Synchronous.

On Entry

x global dense matrix x.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array containing numbers of type specified in Table 14.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

update Update operator.

update = psb_none_ Do nothing;
update = psb_add_ Sum overlap entries, i.e. apply P^T;
update = psb_avg_ Average overlap entries, i.e. apply $P_a P^T$;

Scope: **global**
Intent: **in**.
Default: *update_type* = *psb_avg_*
Scope: **global**
Specified as: a integer variable.

work the work array.
Scope: **local**
Type: **optional**
Intent: **inout**.
Specified as: a one dimensional array of the same type of x.

On Return

x global dense result matrix x.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: an array of rank one or two containing numbers of type specified in Table 14.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. If there is no overlap in the data distribution associated with the descriptor, no operations are performed;

2. The operator P^T performs the reduction sum of overlap elements; it is a "prolongation" operator P^T that replicates overlap elements, accounting for the physical replication of data;

3. The operator P_a performs a scaling on the overlap elements by the amount of replication; thus, when combined with the reduction operator, it implements the average of replicated elements over all of their instances.

Example of use Consider the discretization mesh depicted in fig. 8, partitioned among two processes as shown by the dashed lines, with an overlap of 1 extra layer with respect to the partition of fig. 7; the data distribution is such that each process will own 40 entries in the index space, with an overlap of 16 entries placed at local indices 25 through 40; the halo will run from local index 41 through local index 48.. If process 0 assigns an initial value of 1 to its entries in the x vector, and process 1 assigns a value of 2, then after a call to `psb_ovrl` with `psb_avg_` and a call to `psb_halo_` the contents of the local vectors will be the following (showing a transition among the two subdomains)

	Process 0			Process 1	
I	GLOB(I)	X(I)	I	GLOB(I)	X(I)
1	1	1.0	1	33	1.5
2	2	1.0	2	34	1.5
3	3	1.0	3	35	1.5
4	4	1.0	4	36	1.5
5	5	1.0	5	37	1.5
6	6	1.0	6	38	1.5
7	7	1.0	7	39	1.5
8	8	1.0	8	40	1.5
9	9	1.0	9	41	2.0
10	10	1.0	10	42	2.0
11	11	1.0	11	43	2.0
12	12	1.0	12	44	2.0
13	13	1.0	13	45	2.0
14	14	1.0	14	46	2.0
15	15	1.0	15	47	2.0
16	16	1.0	16	48	2.0
17	17	1.0	17	49	2.0
18	18	1.0	18	50	2.0
19	19	1.0	19	51	2.0
20	20	1.0	20	52	2.0
21	21	1.0	21	53	2.0
22	22	1.0	22	54	2.0
23	23	1.0	23	55	2.0
24	24	1.0	24	56	2.0
25	25	1.5	25	57	2.0
26	26	1.5	26	58	2.0
27	27	1.5	27	59	2.0
28	28	1.5	28	60	2.0
29	29	1.5	29	61	2.0
30	30	1.5	30	62	2.0
31	31	1.5	31	63	2.0
32	32	1.5	32	64	2.0
33	33	1.5	33	25	1.5
34	34	1.5	34	26	1.5
35	35	1.5	35	27	1.5
36	36	1.5	36	28	1.5
37	37	1.5	37	29	1.5
38	38	1.5	38	30	1.5
39	39	1.5	39	31	1.5
40	40	1.5	40	32	1.5
41	41	2.0	41	17	1.0
42	42	2.0	42	18	1.0
43	43	2.0	43	19	1.0
44	44	2.0	44	20	1.0
45	45	2.0	45	21	1.0
46	46	2.0	46	22	1.0
47	47	2.0	47	23	1.0
48	48	2.0	48	24	1.0

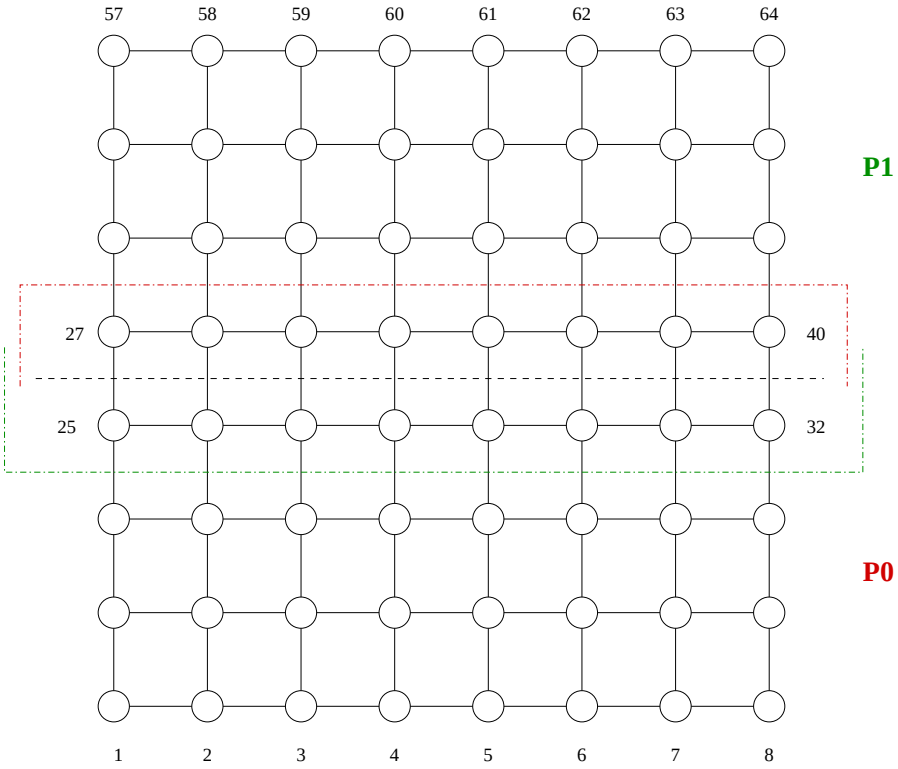

Figure 8: Sample discretization mesh.

psb_gather — Gather Global Dense Matrix

These subroutines collect the portions of global dense matrix distributed over all process into one single array stored on one process.

$$glob\_x \leftarrow collect(loc\_x_i)$$

where:

glob_x is the global submatrix $glob\_x_{1:m,1:n}$

loc_x_i is the local portion of global dense matrix on process i.

collect is the collect function.

x_i, y	**Subroutine**
Integer	psb_gather
Short Precision Real	psb_gather
Long Precision Real	psb_gather
Short Precision Complex	psb_gather
Long Precision Complex	psb_gather

Table 15: Data types

```
call psb_gather(glob_x, loc_x, desc_a, info, root)
call psb_gather(glob_x, loc_x, desc_a, info, root)
```

Type: Synchronous.

On Entry

loc_x the local portion of global dense matrix *glob_x*.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array containing numbers of the type indicated in Table 15.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

root The process that holds the global copy. If $root = -1$ all the processes will have a copy of the global vector.
Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: an integer variable $-1 \leq root \leq np - 1$, default -1.

On Return

glob_x The array where the local parts must be gathered.
Scope: **global**
Type: **required**
Intent: **out**.
Specified as: a rank one or two array.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_scatter — Scatter Global Dense Matrix

These subroutines scatters the portions of global dense matrix owned by a process to all the processes in the processes grid.

$$loc\_x_i \leftarrow scatter(glob\_x)$$

where:

glob_x is the global matrix $glob\_x_{1:m,1:n}$

loc_x_i is the local portion of global dense matrix on process i.

scatter is the scatter function.

x_i, y	**Subroutine**
Integer	psb_scatter
Short Precision Real	psb_scatter
Long Precision Real	psb_scatter
Short Precision Complex	psb_scatter
Long Precision Complex	psb_scatter

Table 16: Data types

```
call psb_scatter(glob_x, loc_x, desc_a, info, root)
call psb_scatter(glob_x, loc_x, desc_a, info, root)
```

Type: Synchronous.

On Entry

glob_x The array that must be scattered into local pieces.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a rank one or two array.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

root The process that holds the global copy. If $root = -1$ all the processes have a copy of the global vector.
Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: an integer variable $-1 \leq root \leq np - 1$, default -1.

On Return

loc_x the local portion of global dense matrix *glob_x*.
Scope: **local**
Type: **required**
Intent: **out**.
Specified as: a rank one or two array containing numbers of the type indicated in Table 16.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

6 Data management routines

psb_cdall — Allocates a communication descriptor

```
call psb_cdall(icontxt, desc_a, info,mg=mg,parts=parts)
call psb_cdall(icontxt, desc_a, info,vg=vg,[mg=mg,flag=flag])
call psb_cdall(icontxt, desc_a, info,vl=vl,[nl=nl,globalcheck=.true.])
call psb_cdall(icontxt, desc_a, info,nl=nl)
call psb_cdall(icontxt, desc_a, info,mg=mg,repl=.true.)
```

This subroutine initializes the communication descriptor associated with an index space. One of the optional arguments `parts`, `vg`, `vl`, `nl` or `repl` must be specified, thereby choosing the specific initialization strategy.

On Entry

Type: Synchronous.

icontxt the communication context.
Scope:**global**.
Type:**required**.
Intent: **in**.
Specified as: an integer value.

vg Data allocation: each index $i \in \{1 \dots mg\}$ is allocated to process $vg(i)$.
Scope:**global**.
Type:**optional**.
Intent: **in**.
Specified as: an integer array.

flag Specifies whether entries in vg are zero- or one-based.
Scope:**global**.
Type:**optional**.
Intent: **in**.
Specified as: an integer value $0, 1$, default 0.

mg the (global) number of rows of the problem.
Scope:**global**.
Type:**optional**.
Intent: **in**.
Specified as: an integer value. It is required if `parts` or `repl` is specified, it is optional if `vg` is specified.

parts the subroutine that defines the partitioning scheme.
Scope:**global**.
Type:**required**.
Specified as: a subroutine.

vl Data allocation: the set of global indices $vl(1 : nl)$ belonging to the calling process.
Scope:**local**.
Type:**optional**.
Intent: **in**.
Specified as: an integer array.

nl Data allocation: in a generalized block-row distribution the number of indices belonging to the current process.
Scope:**local**.
Type:**optional**.
Intent: **in**.
Specified as: an integer value. May be specified together with `vl`.

repl Data allocation: build a replicated index space (i.e. all processes own all indices).
Scope:**global**.
Type:**optional**.
Intent: **in**.
Specified as: the logical value `.true.`

globalcheck Data allocation: do global checks on the local index lists `vl`
Scope:**global**.
Type:**optional**.
Intent: **in**.
Specified as: a logical value, default: `.true.`

On Return

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **out**.
Specified as: a structured data of type psb_desc_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. One of the optional arguments `parts`, `vg`, `vl`, `nl` or `repl` must be specified, thereby choosing the initialization strategy as follows:

   parts In this case we have a subroutine specifying the mapping between global indices and process/local index pairs. If this optional argument is specified, then it is mandatory to specify the argument `mg` as well. The subroutine must conform to the following interface:

   ```
   interface
      subroutine psb_parts(glob_index,mg,np,pv,nv)
        integer, intent (in)  :: glob_index,np,mg
        integer, intent (out) :: nv, pv(*)
      end subroutine psb_parts
   end interface
   ```

   The input arguments are:

   glob_index The global index to be mapped;

np The number of processes in the mapping;

mg The total number of global rows in the mapping;

The output arguments are:

nv The number of entries in `pv`;

pv A vector containing the indices of the processes to which the global index should be assigend; each entry must satisfy $0 \leq pv(i) < np$; if $nv > 1$ we have an index assigned to multiple processes, i.e. we have an overlap among the subdomains.

vg In this case the association between an index and a process is specified via an integer vector `vg(1:mg)`; each index $i \in \{1 \ldots mg\}$ is assigned to process $vg(i)$. The vector `vg` must be identical on all calling processes; its entries may have the ranges $(0 \ldots np-1)$ or $(1 \ldots np)$ according to the value of `flag`. The size mg may be specified via the optional argument `mg`; the default is to use the entire vector `vg`, thus having `mg=size(vg)`.

vl In this case we are specifying the list of indices `vl(1:nl)` assigned to the current process; thus, the global problem size mg is given by the range of the aggregate of the individual vectors `vl` specified in the calling processes. The size may be specified via the optional argument `nl`; the default is to use the entire vector `vl`, thus having `nl=size(vl)`. If `globalcheck=.true.` the subroutine will check how many times each entry in the global index space $(1 \ldots mg)$ is specified in the input lists `vl`, thus allowing for the presence of overlap in the input, and checking for "orphan" indices. If `globalcheck=.false.`, the subroutine will not check for overlap, and may be significantly faster, but the user is implicitly guaranteeing that there are neither orphan nor overlap indices.

nl If this argument is specified alone (i.e. without `vl`) the result is a generalized row-block distribution in which each process I gets assigned a consecutive chunk of $N_I = nl$ global indices.

repl This arguments specifies to replicate all indices on all processes. This is a special purpose data allocation that is useful in the construction of some multilevel preconditioners.

2. On exit from this routine the descriptor is in the build state.

3. Calling the routine with `vg` or `parts` implies that every process will scan the entire index space to figure out the local indices.

4. Overlapped indices are possible with both `parts` and `vl` invocations.

5. When the subroutine is invoked with `vl` in conjunction with `globalcheck=.true.`, it will perform a scan of the index space to search for overlap or orphan indices.

6. When the subroutine is invoked with `vl` in conjunction with `globalcheck=.false.`, no index space scan will take place. Thus it is the responsibility of the user to make sure that the indices specified in `vl` have neither orphans nor overlaps; if this assumption fails, results will be unpredictable.

7. Orphan and overlap indices are impossible by construction when the subroutine is invoked with `nl` (alone), or `vg`.

psb_cdins — Communication descriptor insert routine

```
call psb_cdins(nz, ia, ja, desc_a, info)
```

This subroutine examines the edges of the graph associated with the discretization mesh (and isomorphic to the sparsity pattern of a linear system coefficient matrix), storing them as necessary into the communication descriptor.

Type: Asynchronous.

On Entry

nz the number of points being inserted.
Scope: **local**.
Type: **required**.
Intent: **in**.
Specified as: an integer value.

ia the indices of the starting vertex of the edges being inserted.
Scope: **local**.
Type: **required**.
Intent: **in**.
Specified as: an integer array of length nz.

ja the indices of the end vertex of the edges being inserted.
Scope: **local**.
Type: **required**.
Intent: **in**.
Specified as: an integer array of length nz.

On Return

desc_a the updated communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: a structured data of type psb_desc_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. This routine may only be called if the descriptor is in the build state;
2. This routine automatically ignores edges that do not insist on the current process, i.e. edges for which neither the starting nor the end vertex belong to the current process.

psb_cdasb — Communication descriptor assembly routine

```
call psb_cdasb(desc_a, info)
```

Type: Synchronous.

On Entry

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: a structured data of type psb_desc_type.

On Return

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: a structured data of type psb_desc_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. On exit from this routine the descriptor is in the assembled state.

psb_cdcpy — Copies a communication descriptor

```
call psb_cdcpy(desc_in, desc_out, info)
```

Type: Asynchronous.

On Entry

desc_in the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

desc_out the communication descriptor copy.
Scope:**local**.
Type:**required**.
Intent: **out**.
Specified as: a structured data of type psb_desc_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_cdfree — Frees a communication descriptor

```
call psb_cdfree(desc_a, info)
```

Type: Synchronous.

On Entry

desc_a the communication descriptor to be freed.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: a structured data of type psb_desc_type.

On Return

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_cdbldext — Build an extended communication descriptor

```
call psb_cdbldext(a,desc_a,nl,desc_out, info, extype)
```

This subroutine builds an extended communication descriptor, based on the input descriptor `desc_a` and on the stencil specified through the input sparse matrix `a`.

Type: Synchronous.

On Entry

a A sparse matrix Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data type.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_spmat_type.

nl the number of additional layers desired.
Scope:**global**.
Type:**required**.
Intent: **in**.
Specified as: an integer value $nl \geq 0$.

extype the kind of estension required.
Scope:**global**.
Type:**optional** .
Intent: **in**.
Specified as: an integer value `psb_ovt_xhal_`, `psb_ovt_asov_`, default: `psb_ovt_xhal_`

On Return

desc_out the extended communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: a structured data of type psb_desc_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. Specifying `psb_ovt_xhal_` for the `extype` argument the user will obtain a descriptor for a domain partition in which the additional layers are fetched as part of an (extended) halo; however the index-to-process mapping is identical to that of the base descriptor;

2. Specifying `psb_ovt_asov_` for the `extype` argument the user will obtain a descriptor with an overlapped decomposition: the additional layer is aggregated to the local subdomain (and thus is an overlap), and a new halo extending beyond the last additional layer is formed.

psb_spall — Allocates a sparse matrix

```
call psb_spall(a, desc_a, info, nnz)
```

Type: Synchronous.

On Entry

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

nnz An estimate of the number of nonzeroes in the local part of the assembled matrix.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value.

On Return

a the matrix to be allocated.
Scope:**local**
Type:**required**
Intent: **out**.
Specified as: a structured data of type psb_spmat_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. On exit from this routine the sparse matrix is in the build state.
2. The descriptor may be in either the build or assembled state.
3. Providing a good estimate for the number of nonzeroes nnz in the assembled matrix may substantially improve performance in the matrix build phase, as it will reduce or eliminate the need for (potentially multiple) data reallocations.

psb_spins — Insert a cloud of elements into a sparse matrix

```
call psb_spins(nz, ia, ja, val, a, desc_a, info)
```

Type: Asynchronous.

On Entry

nz the number of elements to be inserted.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: an integer scalar.

ia the row indices of the elements to be inserted.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: an integer array of size nz.

ja the column indices of the elements to be inserted.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: an integer array of size nz.

val the elements to be inserted.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: an array of size nz. Must be of the same type and kind of the `aspk` component of the sparse matrix a.

desc_a The communication descriptor.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: a variable of type `psb_desc_type`.

On Return

a the matrix into which elements will be inserted.
Scope:**local**
Type:**required**
Intent: **inout**.
Specified as: a structured data of type `psb_spmat_type`.

desc_a The communication descriptor.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: a variable of type `psb_desc_type`.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. On entry to this routine the descriptor may be in either the build or assembled state.

2. On entry to this routine the sparse matrix may be in either the build or update state.

3. If the descriptor is in the build state, then the sparse matrix must also be in the build state; the action of the routine is to (implicitly) call `psb_cdins` to add entries to the sparsity pattern; each sparse matrix entry implicitly defines a graph edge, that is passed to the descriptor routine for the appropriate processing.

4. Any coefficients from matrix rows not assigned to the calling process are silently ignored;

5. If the descriptor is in the assembled state, then any entries in the sparse matrix that would generate additional communication requirements will be ignored;

6. If the matrix is in the update state, any entries in positions that were not present in the original matrix will be ignored.

psb_spasb — Sparse matrix assembly routine

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

Type: Synchronous.

On Entry

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

afmt the storage format for the sparse matrix.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an array of characters. Defalt: 'CSR'.

upd Provide for updates to the matrix coefficients.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: integer, possible values: psb_upd_srch_, psb_upd_perm_

dupl How to handle duplicate coefficients.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: integer, possible values: psb_dupl_ovwrt_, psb_dupl_add_, psb_dupl_err_.

On Return

a the matrix to be assembled.
Scope:**local**
Type:**required**
Intent: **inout**.
Specified as: a structured data of type psb_spmat_type.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. On entry to this routine the descriptor must be in the assembled state, i.e. psb_cdasb must already have been called.

2. The sparse matrix may be in either the build or update state;

3. Duplicate entries are detected and handled in both build and update state, with the exception of the error action that is only taken in the build state, i.e. on the first assembly;

4. If the update choice is `psb_upd_perm_`, then subsequent calls to `psb_spins` to update the matrix must be arranged in such a way as to produce exactly the same sequence of coefficient values as encountered at the first assembly;

5. On exit from this routine the matrix is in the assembled state, and thus is suitable for the computational routines.

psb_spfree — Frees a sparse matrix

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

Type: Synchronous.

On Entry

a the matrix to be freed.
Scope:**local**
Type:**required**
Intent: **inout**.
Specified as: a structured data of type psb_spmat_type.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_sprn — Reinit sparse matrix structure for psblas routines.

```
call psb_sprn(a, decsc_a, info, clear)
```

Type: Synchronous.

On Entry

a the matrix to be reinitialized.
Scope:**local**
Type:**required**
Intent: **inout**.
Specified as: a structured data of type psb_spmat_type.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

clear Choose whether to zero out matrix coefficients
Scope:**local**.
Type:**optional**.
Intent: **in**.
Default: true.

On Return

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. On exit from this routine the sparse matrix is in the update state.

psb_geall — Allocates a dense matrix

```
call psb_geall(x, desc_a, info, n, lb)
```

Type: Synchronous.

On Entry

desc_a The communication descriptor.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a variable of type psb_desc_type.

n The number of columns of the dense matrix to be allocated.
Scope: **local**
Type: **optional**
Intent: **in**.
Specified as: Integer scalar, default 1. It is not a valid argument if x is a rank-1 array.

lb The lower bound for the column index range of the dense matrix to be allocated.
Scope: **local**
Type: **optional**
Intent: **in**.
Specified as: Integer scalar, default 1. It is not a valid argument if x is a rank-1 array.

On Return

x The dense matrix to be allocated.
Scope: **local**
Type: **required**
Intent: **out**.
Specified as: a rank one or two array with the ALLOCATABLE attribute, of type real, complex or integer.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_geins — Dense matrix insertion routine

```
call psb_geins(m, irw, val, x, desc_a, info,dupl)
```

Type: Asynchronous.

On Entry

m Number of rows in *val* to be inserted.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: an integer value.

irw Indices of the rows to be inserted. Specifically, row i of *val* will be inserted into the local row corresponding to the global row index $irw(i)$.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: an integer array.

val the dense submatrix to be inserted.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a rank 1 or 2 array. Specified as: an integer value.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

dupl How to handle duplicate coefficients.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: integer, possible values: psb_dupl_ovwrt_, psb_dupl_add_.

On Return

x the output dense matrix.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array with the ALLOCATABLE attribute, of type real, complex or integer.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. Dense vectors/matrices do not have an associated state;
2. Duplicate entries are either overwritten or added, there is no provision for raising an error condition.

psb_geasb — Assembly a dense matrix

```
call psb_geasb(x, desc_a, info)
```

Type: Synchronous.

On Entry

desc_a The communication descriptor.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a variable of type psb_desc_type.

On Return

x The dense matrix to be assembled.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array with the ALLOCATABLE attribute, of type real, complex or integer.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_gefree — Frees a dense matrix

```
call psb_gefree(x, desc_a, info)
```

Type: Synchronous.

On Entry

x The dense matrix to be freed.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one or two array with the ALLOCATABLE attribute, of type real, complex or integer.

desc_a The communication descriptor.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a variable of type psb_desc_type.

On Return

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_gelp — Applies a left permutation to a dense matrix

```
call psb_gelp(trans, iperm, x, info)
```

Type: Asynchronous.

On Entry

trans A character that specifies whether to permute A or A^T.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a single character with value 'N' for A or 'T' for A^T.

iperm An integer array containing permutation information.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: an integer one-dimensional array.

x The dense matrix to be permuted.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a one or two dimensional array.

On Return

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_glob_to_loc — Global to local indices convertion

```
call psb_glob_to_loc(x, y, desc_a, info, iact,owned)
call psb_glob_to_loc(x, desc_a, info, iact,owned)
```

Type: Asynchronous.

On Entry

x An integer vector of indices to be converted.
Scope: **local**
Type: **required**
Intent: **in, inout**.
Specified as: a rank one integer array.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

iact specifies action to be taken in case of range errors. Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: a character variable Ignore, Warning or Abort, default Ignore.

owned Specfies valid range of input Scope: **global**
Type: **optional**
Intent: **in**.
If true, then only indices strictly owned by the current process are considered valid, if false then halo indices are also accepted. Default: false.

On Return

x If y is not present, then x is overwritten with the translated integer indices.
Scope: **global**
Type: **required**
Intent: **inout**.
Specified as: a rank one integer array.

y If y is present, then y is overwritten with the translated integer indices, and x is left unchanged. Scope: **global**
Type: **optional**
Intent: **out**.
Specified as: a rank one integer array.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. If an input index is out of range, then the corresponding output index is set to a negative number;

2. The default `Ignore` means that the negative output is the only action taken on an out-of-range input.

psb_loc_to_glob — Local to global indices conversion

```
call psb_loc_to_glob(x, y, desc_a, info, iact)
call psb_loc_to_glob(x, desc_a, info, iact)
```

Type: Asynchronous.

On Entry

x An integer vector of indices to be converted.
Scope: **local**
Type: **required**
Intent: **in, inout**.
Specified as: a rank one integer array.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

iact specifies action to be taken in case of range errors. Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: a character variable Ignore, Warning or Abort, default Ignore.

On Return

x If y is not present, then x is overwritten with the translated integer indices.
Scope: **global**
Type: **required**
Intent: **inout**.
Specified as: a rank one integer array.

y If y is not present, then y is overwritten with the translated integer indices, and x is left unchanged. Scope: **global**
Type: **optional**
Intent: **out**.
Specified as: a rank one integer array.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_is_owned

```
call psb_is_owned(x, desc_a)
```

Type: Asynchronous.

On Entry

x Integer index.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a scalar integer.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

Function value A logical mask which is true if x is owned by the current process Scope: **local**
Type: **required**
Intent: **out**.

Notes

1. This routine returns a `.true.` value for an index that is strictly owned by the current process, excluding the halo indices

psb_owned_index

```
call psb_owned_index(y, x, desc_a, info)
```

Type: Asynchronous.

On Entry

x Integer indices.
Scope: **local**
Type: **required**
Intent: **in, inout**.
Specified as: a scalar or a rank one integer array.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

iact specifies action to be taken in case of range errors. Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: a character variable Ignore, Warning or Abort, default Ignore.

On Return

y A logical mask which is true for all corresponding entries of x that are owned by the current process Scope: **local**
Type: **required**
Intent: **out**.
Specified as: a scalar or rank one logical array.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. This routine returns a `.true.` value for those indices that are strictly owned by the current process, excluding the halo indices

psb_is_local

```
call psb_is_local(x, desc_a)
```

Type: Asynchronous.

On Entry

x Integer index.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a scalar integer.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

Function value A logical mask which is true if x is local to the current process
Scope: **local**
Type: **required**
Intent: **out**.

Notes

1. This routine returns a `.true.` value for an index that is local to the current process, including the halo indices

psb_local_index

```
call psb_local_index(y, x, desc_a, info)
```

Type: Asynchronous.

On Entry

x Integer indices.
Scope: **local**
Type: **required**
Intent: **in, inout**.
Specified as: a scalar or a rank one integer array.

desc_a the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

iact specifies action to be taken in case of range errors. Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: a character variable Ignore, Warning or Abort, default Ignore.

On Return

y A logical mask which is true for all corresponding entries of x that are local to the current process Scope: **local**
Type: **required**
Intent: **out**.
Specified as: a scalar or rank one logical array.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. This routine returns a `.true.` value for those indices that are local to the current process, including the halo indices.

psb_get_boundary — Extract list of boundary elements

```
call psb_get_boundary(bndel, desc, info)
```

Type: Asynchronous.

On Entry

desc the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

bndel The list of boundary elements on the calling process, in local numbering.
Scope: **local**
Type: **required**
Intent: **out**.
Specified as: a rank one array with the ALLOCATABLE attribute, of type integer.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. If there are no boundary elements (i.e., if the local part of the connectivity graph is self-contained) the output vector is set to the "not allocated" state.

2. Otherwise the size of `bndel` will be exactly equal to the number of boundary elements.

psb_get_overlap — Extract list of overlap elements

```
call psb_get_overlap(ovrel, desc, info)
```

Type: Asynchronous.

On Entry

desc the communication descriptor.
Scope:**local**.
Type:**required**.
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

On Return

ovrel The list of overlap elements on the calling process, in local numbering.
Scope: **local**
Type: **required**
Intent: **out**.
Specified as: a rank one array with the ALLOCATABLE attribute, of type integer.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. If there are no overlap elements the output vector is set to the "not allocated" state.
2. Otherwise the size of `ovrel` will be exactly equal to the number of overlap elements.

psb_sp_getrow — Extract row(s) from a sparse matrix

```
call psb_sp_getrow(row, a, nz, ia, ja, val, info, &
             & append, nzin, lrw)
```

Type: Asynchronous.

On Entry

row The (first) row to be extracted.
Scope:**local**
Type:**required**
Intent: **in**.
Specified as: an integer > 0.

a the matrix from which to get rows.
Scope:**local**
Type:**required**
Intent: **in**.
Specified as: a structured data of type psb_spmat_type.

append Whether to append or overwrite existing output.
Scope:**local**
Type:**optional**
Intent: **in**.
Specified as: a logical value default: false (overwrite).

nzin Input size to be appended to.
Scope:**local**
Type:**optional**
Intent: **in**.
Specified as: an integer > 0. When append is true, specifies how many entries in the output vectors are already filled.

lrw The last row to be extracted.
Scope:**local**
Type:**optional**
Intent: **in**.
Specified as: an integer > 0, default: *row*.

On Return

nz the number of elements returned by this call.
Scope:**local**.
Type:**required**.
Intent: **out**.
Returned as: an integer scalar.

ia the row indices.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: an integer array with the ALLOCATABLE attribute.

ja the column indices of the elements to be inserted.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: an integer array with the `ALLOCATABLE` attribute.

val the elements to be inserted.
Scope:**local**.
Type:**required**.
Intent: **inout**.
Specified as: a real array with the `ALLOCATABLE` attribute.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

Notes

1. The output nz is always the size of the output generated by the current call; thus, if `append=.true.`, the total output size will be $nzin + nz$, with the newly extracted coefficients stored in entries `nzin+1:nzin+nz` of the array arguments;

2. When `append=.true.` the output arrays are reallocated as necessary;

3. The row and column indices are returned in the local numbering scheme; if the global numbering is desired, the user may employ the `psb_loc_to_glob` routine on the output.

psb_sizeof — Memory occupation

This function computes the memory occupation of a PSBLAS object.

```
isz = psb_sizeof(a)
isz = psb_sizeof(desc_a)
isz = psb_sizeof(prec)
```

Type: Asynchronous.

On Entry

a A sparse matrix A.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_spmat_type.

desc_a Communication descriptor.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

prec Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a preconditioner data structure psb_prec_type.

On Return

Function value The memory occupation of the object specified in the calling sequence, in bytes.
Scope: **local**
Returned as: an `integer(psb_long_int_k_)` number.

Sorting utilities

psb_msort — Sorting by the Merge-sort algorithm
psb_qsort — Sorting by the Quicksort algorithm
psb_hsort — Sorting by the Heapsort algorithm

```
call psb_msort(x,ix,dir,flag)
call psb_qsort(x,ix,dir,flag)
call psb_hsort(x,ix,dir,flag)
```

These serial routines sort a sequence X into ascending or descending order. The argument meaning is identical for the three calls; the only difference is the algorithm used to accomplish the task (see Usage Notes below).

Type: Asynchronous.

On Entry

x The sequence to be sorted.
Type:**required**.
Specified as: an integer, real or complex array of rank 1.

ix A vector of indices.
Type:**optional**.
Specified as: an integer array of (at least) the same size as X.

dir The desired ordering.
Type:**optional**.
Specified as: an integer value:

Integer and real data: `psb_sort_up_`, `psb_sort_down_`, `psb_asort_up_`, `psb_asort_down_`; default `psb_sort_up_`.

Complex data: `psb_lsort_up_`, `psb_lsort_down_`, `psb_asort_up_`, `psb_asort_down_`; default `psb_lsort_up_`.

flag Whether to keep the original values in IX.
Type:**optional**.
Specified as: an integer value `psb_sort_ovw_idx_` or `psb_sort_keep_idx_`; default `psb_sort_ovw_idx_`.

On Return

x The sequence of values, in the chosen ordering.
Type:**required**.
Specified as: an integer, real or complex array of rank 1.

ix A vector of indices.
Type: **Optional**
An integer array of rank 1, whose entries are moved to the same position as the corresponding entries in x.

Notes

1. For integer or real data the sorting can be performed in the up/down direction, on the natural or absolute values;
2. For complex data the sorting can be done in a lexicographic order (i.e.: sort on the real part with ties broken according to the imaginary part) or on the absolute values;
3. The routines return the items in the chosen ordering; the output difference is the handling of ties (i.e. items with an equal value) in the original input. With the merge-sort algorithm ties are preserved in the same relative order as they had in the original sequence, while this is not guaranteed for quicksort or heapsort;
4. If $flag = psb\_sort\_ovw\_idx\_$ then the entries in $ix(1:n)$ where n is the size of x are initialized to $ix(i) \leftarrow i$; thus, upon return from the subroutine, for each index i we have in $ix(i)$ the position that the item $x(i)$ occupied in the original data sequence;
5. If $flag = psb\_sort\_keep\_idx\_$ the routine will assume that the entries in $ix(:)$ have already been initialized by the user;
6. The three sorting algorithms have a similar $O(n \log n)$ expected running time; in the average case quicksort will be the fastest and merge-sort the slowest. However note that:
   (a) The worst case running time for quicksort is $O(n^2)$; the algorithm implemented here follows the well-known median-of-three heuristics, but the worst case may still apply;
   (b) The worst case running time for merge-sort and heap-sort is $O(n \log n)$ as the average case;
   (c) The merge-sort algorithm is implemented to take advantage of subsequences that may be already in the desired ordering prior to the subroutine call; this situation is relatively common when dealing with groups of indices of sparse matrix entries, thus merge-sort is often the preferred choice when a sorting is needed by other routines in the library.

7 Parallel environment routines

psb_init — Initializes PSBLAS parallel environment

```
call psb_init(icontxt, np, basectxt, ids)
```

This subroutine initializes the PSBLAS parallel environment, defining a virtual parallel machine.

Type: Synchronous.

On Entry

np Number of processes in the PSBLAS virtual parallel machine.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value. Default: use all available processes.

basectxt the initial communication context. The new context will be defined from the processes participating in the initial one.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value. Default: use MPI_COMM_WORLD.

ids Identities of the processes to use for the new context; the argument is ignored when `np` is not specified. This allows the processes in the new environment to be in an order different from the original one.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer array. Default: use the indices $(0 \ldots np-1)$.

On Return

icontxt the communication context identifying the virtual parallel machine. Note that this is always a duplicate of `basectxt`, so that library communications are completely separated from other communication operations.
Scope: **global**.
Type: **required**.
Intent: **out**.
Specified as: an integer variable.

Notes

1. A call to this routine must precede any other PSBLAS call.
2. It is an error to specify a value for *np* greater than the number of processes available in the underlying base parallel environment.

psb_info — Return information about PSBLAS parallel environment

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

This subroutine returns information about the PSBLAS parallel environment, defining a virtual parallel machine.

Type: Asynchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

On Return

iam Identifier of current process in the PSBLAS virtual parallel machine.
Scope: **local**.
Type: **required**.
Intent: **out**.
Specified as: an integer value. $-1 \leq iam \leq np-1$

np Number of processes in the PSBLAS virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **out**.
Specified as: an integer variable.

Notes

1. For processes in the virtual parallel machine the identifier will satisfy $0 \leq iam \leq np-1$;
2. If the user has requested on `psb_init` a number of processes less than the total available in the parallel execution environment, the remaining processes will have on return $iam = -1$; the only call involving `icontxt` that any such process may execute is to `psb_exit`.

psb_exit — Exit from PSBLAS parallel environment

```
call psb_exit(icontxt)
call psb_exit(icontxt,close)
```

This subroutine exits from the PSBLAS parallel virtual machine.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

close Whether to close all data structures related to the virtual parallel machine, besides those associated with icontxt.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: a logical variable, default value: true.

Notes

1. This routine may be called even if a previous call to `psb_info` has returned with $iam = -1$; indeed, it it is the only routine that may be called with argument `icontxt` in this situation.

2. A call to this routine with `close=.true.` implies a call to `MPI_Finalize`, after which no parallel routine may be called.

3. If the user whishes to use multiple communication contexts in the same program, or to enter and exit multiple times into the parallel environment, this routine may be called to selectively close the contexts with `close=.false.`, while on the last call it should be called with `close=.true.` to shutdown in a clean way the entire parallel environment.

psb_get_mpicomm — Get the MPI communicator

```
call psb_get_mpicomm(icontxt, icomm)
```

This subroutine returns the MPI communicator associated with a PSBLAS context

Type: Asynchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

On Return

icomm The MPI communicator associated with the PSBLAS virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **out**.

psb_get_rank — Get the MPI rank

```
call psb_get_rank(rank, icontxt, id)
```

This subroutine returns the MPI rank of the PSBLAS process *id*

Type: Asynchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

id Identifier of a process in the PSBLAS virtual parallel machine.
Scope: **local**.
Type: **required**.
Intent: **in**.
Specified as: an integer value. $0 \leq id \leq np - 1$

On Return

rank The MPI rank associated with the PSBLAS process *id*.
Scope: **local**.
Type: **required**.
Intent: **out**.

psb_wtime — Wall clock timing

```
time = psb_wtime()
```

This function returns a wall clock timer. The resolution of the timer is dependent on the underlying parallel environment implementation.

Type: Asynchronous.

On Exit

Function value the elapsed time in seconds.
Returned as: a `real(psb_dpk_)` variable.

psb_barrier — Sinchronization point parallel environment

```
call psb_barrier(icontxt)
```

This subroutine acts as an explicit synchronization point for the PSBLAS parallel virtual machine.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

psb_abort — Abort a computation

```
call psb_abort(icontxt)
```

This subroutine aborts computation on the parallel virtual machine.

Type: Asynchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

psb_bcast — Broadcast data

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

This subroutine implements a broadcast operation based on the underlying communication library.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat On the root process, the data to be broadcast.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array, or a character or logical variable, which may be a scalar or rank 1 array. Type, kind, rank and size must agree on all processes.

root Root process holding data to be broadcast.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value $0 <= root <= np - 1$, default 0

On Return

dat On processes other than root, the data to be broadcast.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array, or a character or logical scalar. Type, kind, rank and size must agree on all processes.

psb_sum — Global sum

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

This subroutine implements a sum reduction operation based on the underlying communication library.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat The local contribution to the global sum.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

root Process to hold the final sum, or -1 to make it available on all processes.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value $-1 <= root <= np - 1$, default -1.

On Return

dat On destination process(es), the result of the sum operation.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array.
Type, kind, rank and size must agree on all processes.

Notes

1. The `dat` argument is both input and output, and its value may be changed even on processes different from the final result destination.

2. The `dat` argument may also be a long integer scalar.

psb_max — Global maximum

```
call psb_max(icontxt, dat, root)
```

This subroutine implements a maximum valuereduction operation based on the underlying communication library.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat The local contribution to the global maximum.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: an integer or real variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

root Process to hold the final maximum, or -1 to make it available on all processes.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value $-1 <= root <= np - 1$, default -1.

On Return

dat On destination process(es), the result of the maximum operation.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer or real variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

Notes

1. The `dat` argument is both input and output, and its value may be changed even on processes different from the final result destination.

2. The `dat` argument may also be a long integer scalar.

psb_min — Global minimum

```
call psb_min(icontxt, dat, root)
```

This subroutine implements a minimum value reduction operation based on the underlying communication library.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat The local contribution to the global minimum.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: an integer or real variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

root Process to hold the final value, or -1 to make it available on all processes.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value $-1 <= root <= np - 1$, default -1.

On Return

dat On destination process(es), the result of the minimum operation.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer or real variable, which may be a scalar, or a rank 1 or 2 array.
Type, kind, rank and size must agree on all processes.

Notes

1. The `dat` argument is both input and output, and its value may be changed even on processes different from the final result destination.

2. The `dat` argument may also be a long integer scalar.

psb_amx — Global maximum absolute value

```
call psb_amx(icontxt, dat, root)
```

This subroutine implements a maximum absolute value reduction operation based on the underlying communication library.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat The local contribution to the global maximum.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

root Process to hold the final value, or -1 to make it available on all processes.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value $-1 <= root <= np - 1$, default -1.

On Return

dat On destination process(es), the result of the maximum operation.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

Notes

1. The `dat` argument is both input and output, and its value may be changed even on processes different from the final result destination.

2. The `dat` argument may also be a long integer scalar.

psb_amn — Global minimum absolute value

```
call psb_amn(icontxt, dat, root)
```

This subroutine implements a minimum absolute value reduction operation based on the underlying communication library.

Type: Synchronous.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat The local contribution to the global minimum.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array. Type, kind, rank and size must agree on all processes.

root Process to hold the final value, or -1 to make it available on all processes.
Scope: **global**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value $-1 <= root <= np - 1$, default -1.

On Return

dat On destination process(es), the result of the minimum operation.
Scope: **global**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array.
Type, kind, rank and size must agree on all processes.

Notes

1. The `dat` argument is both input and output, and its value may be changed even on processes different from the final result destination.

2. The `dat` argument may also be a long integer scalar.

psb_snd — Send data

```
call psb_snd(icontxt, dat, dst, m)
```

This subroutine sends a packet of data to a destination.

Type: Synchronous: see usage notes.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

dat The data to be sent.
Scope: **local**.
Type: **required**.
Intent: **in**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array, or a character or logical scalar. Type, kind and rank must agree on sender and receiver process; if m is not specified, size must agree as well.

dst Destination process.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer value $0 <= dst <= np - 1$.

m Number of rows.
Scope: **global**.
Type: **Optional**.
Intent: **in**.
Specified as: an integer value $0 <= m <= size(dat, 1)$.
When *dat* is a rank 2 array, specifies the number of rows to be sent independently of the leading dimension $size(dat, 1)$; must have the same value on sending and receiving processes.

On Return

Notes

1. This subroutine implies a synchronization, but only between the calling process and the destination process *dst*.

psb_rcv — Receive data

```
call psb_rcv(icontxt, dat, src, m)
```

This subroutine receives a packet of data to a destination.

Type: Synchronous: see usage notes.

On Entry

icontxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

src Source process.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer value $0 <= src <= np - 1$.

m Number of rows.
Scope: **global**.
Type: **Optional**.
Intent: **in**.
Specified as: an integer value $0 <= m <= size(dat, 1)$.
When *dat* is a rank 2 array, specifies the number of rows to be sent independently of the leading dimension $size(dat, 1)$; must have the same value on sending and receiving processes.

On Return

dat The data to be received.
Scope: **local**.
Type: **required**.
Intent: **inout**.
Specified as: an integer, real or complex variable, which may be a scalar, or a rank 1 or 2 array, or a character or logical scalar. Type, kind and rank must agree on sender and receiver process; if m is not specified, size must agree as well.

Notes

1. This subroutine implies a synchronization, but only between the calling process and the source process *src*.

8 Error handling

The PSBLAS library error handling policy has been completely rewritten in version 2.0. The idea behind the design of this new error handling strategy is to keep error messages on a stack allowing the user to trace back up to the point where the first error message has been generated. Every routine in the PSBLAS-2.0 library has, as last non-optional argument, an integer `info` variable; whenever, inside the routine, an error is detected, this variable is set to a value corresponding to a specific error code. Then this error code is also pushed on the error stack and then either control is returned to the caller routine or the execution is aborted, depending on the users choice. At the time when the execution is aborted, an error message is printed on standard output with a level of verbosity than can be chosen by the user. If the execution is not aborted, then, the caller routine checks the value returned in the `info` variable and, if not zero, an error condition is raised. This process continues on all the levels of nested calls until the level where the user decides to abort the program execution.

Figure 9 shows the layout of a generic `psb_foo` routine with respect to the PSBLAS-2.0 error handling policy. It is possible to see how, whenever an error condition is detected, the `info` variable is set to the corresponding error code which is, then, pushed on top of the stack by means of the `psb_errpush`. An error condition may be directly detected inside a routine or indirectly checking the error code returned returned by a called routine. Whenever an error is encountered, after it has been pushed on stack, the program execution skips to a point where the error condition is handled; the error condition is handled either by returning control to the caller routine or by calling the `psb\_error` routine which prints the content of the error stack and aborts the program execution, according to the choice made by the user with `psb_set_erraction`. The default is to print the error and terminate the program, but the user may choose to handle the error explicitly.

Figure 10 reports a sample error message generated by the PSBLAS-2.0 library. This error has been generated by the fact that the user has chosen the invalid "FOO" storage format to represent the sparse matrix. From this error message it is possible to see that the error has been detected inside the `psb_cest` subroutine called by `psb_spasb` ... by process 0 (i.e. the root process).

```
subroutine psb_foo(some args, info)
   ...
   if(error detected) then
      info=errcode1
      call psb_errpush('psb_foo', errcode1)
      goto 9999
   end if
   ...
   call psb_bar(some args, info)
   if(info .ne. zero) then
      info=errcode2
      call psb_errpush('psb_foo', errcode2)
      goto 9999
   end if
   ...
9999 continue
   if (err_act .eq. act_abort) then
     call psb_error(icontxt)
     return
   else
     return
   end if

end subroutine psb_foo
```

Figure 9: The layout of a generic `psb_foo` routine with respect to PSBLAS-2.0 error handling policy.

```
============================================================
Process: 0.  PSBLAS Error (4010) in subroutine: df_sample
Error from call to subroutine mat dist
============================================================
Process: 0.  PSBLAS Error (4010) in subroutine: mat_distv
Error from call to subroutine psb_spasb
============================================================
Process: 0.  PSBLAS Error (4010) in subroutine: psb_spasb
Error from call to subroutine psb_cest
============================================================
Process: 0.  PSBLAS Error (136) in subroutine: psb_cest
Format FOO is unknown
============================================================
Aborting...
```

Figure 10: A sample PSBLAS-2.0 error message. Process 0 detected an error condition inside the psb_cest subroutine

psb_errpush — Pushes an error code onto the error stack

```
call psb_errpush(err_c, r_name, i_err, a_err)
```

Type: Asynchronous.

On Entry

err_c the error code
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: an integer.

r_name the soutine where the error has been caught.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a string.

i_err addional info for error code
Scope: **local**
Type: **optional**
Specified as: an integer array

a_err addional info for error code
Scope: **local**
Type: **optional**
Specified as: a string.

psb_error — Prints the error stack content and aborts execution

```
call psb_error(icontxt)
```

Type: Asynchronous.

On Entry

icontxt the communication context.
Scope: **global**
Type: **optional**
Intent: **in**.
Specified as: an integer.

psb_set_errverbosity — Sets the verbosity of error messages.

```
call psb_set_errverbosity(v)
```

Type: Asynchronous.

On Entry

v the verbosity level
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: an integer.

psb_set_erraction — Set the type of action to be taken upon error condition.

```
call psb_set_erraction(err_act)
```

Type: Asynchronous.

On Entry

err_act the type of action.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: an integer. Possible values: `psb_act_ret`, `psb_act_abort`.

```
call psb_errcomm(icontxt, err)
```

9 Utilities

We have some utitlities available for input and output of sparsematrices; the interfaces to these routines are available in the module `psb_util_mod`.

hb_read — Read a sparse matrix from a file in the Harwell–Boeing format

```
call hb_read(a, iret, iunit, filename, b, mtitle)
```

Type: Asynchronous.

On Entry

filename The name of the file to be read.
Type:**optional**.
Specified as: a character variable containing a valid file name, or `-`, in which case the default input unit 5 (i.e. standard input in Unix jargon) is used. Default: `-`.

iunit The Fortran file unit number.
Type:**optional**.
Specified as: an integer value. Only meaningful if filename is not `-`.

On Return

a the sparse matrix read from file.
Type:**required**.
Specified as: a structured data of type `psb_spmat_type`.

b Rigth hand side(s).
Type: **Optional**
An array of type real or complex, rank 2 and having the ALLOCATABLE attribute; will be allocated and filled in if the input file contains a right hand side, otherwise will be left in the UNALLOCATED state.

mtitle Matrix title.
Type: **Optional**
A charachter variable of length 72 holding a copy of the matrix title as specified by the Harwell-Boeing format and contained in the input file.

iret Error code.
Type: **required**
An integer value; 0 means no error has been detected.

hb_write — Write a sparse matrix to a file in the Harwell–Boeing format

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

Type: Asynchronous.

On Entry

a the sparse matrix to be written.
Type:**required**.
Specified as: a structured data of type psb_spmat_type.

b Rigth hand side.
Type: **Optional**
An array of type real or complex, rank 1 and having the ALLOCATABLE attribute; will be allocated and filled in if the input file contains a right hand side.

filename The name of the file to be written to.
Type:**optional**.
Specified as: a character variable containing a valid file name, or `-`, in which case the default output unit 6 (i.e. standard output in Unix jargon) is used. Default: `-`.

iunit The Fortran file unit number.
Type:**optional**.
Specified as: an integer value. Only meaningful if filename is not `-`.

key Matrix key.
Type: **Optional**
A charachter variable of length 8 holding the matrix key as specified by the Harwell-Boeing format and to be written to file.

mtitle Matrix title.
Type: **Optional**
A charachter variable of length 72 holding the matrix title as specified by the Harwell-Boeing format and to be written to file.

On Return

iret Error code.
Type: **required**
An integer value; 0 means no error has been detected.

mm_mat_read — Read a sparse matrix from a file in the MatrixMarket format

```
call mm_mat_read(a, iret, iunit, filename)
```

Type: Asynchronous.

On Entry

filename The name of the file to be read.
Type:**optional**.
Specified as: a character variable containing a valid file name, or `-`, in which case the default input unit 5 (i.e. standard input in Unix jargon) is used. Default: `-`.

iunit The Fortran file unit number.
Type:**optional**.
Specified as: an integer value. Only meaningful if filename is not `-`.

On Return

a the sparse matrix read from file.
Type:**required**.
Specified as: a structured data of type `psb_spmat_type`.

iret Error code.
Type: **required**
An integer value; 0 means no error has been detected.

mm_vet_read — Read a dense vector from a file in the MatrixMarket format

```
call mm_vet_read(b, iret, iunit, filename)
```

Type: Asynchronous.

On Entry

filename The name of the file to be read.
Type:**optional**.
Specified as: a character variable containing a valid file name, or `-`, in which case the default input unit 5 (i.e. standard input in Unix jargon) is used. Default: `-`.

iunit The Fortran file unit number.
Type:**optional**.
Specified as: an integer value. Only meaningful if filename is not `-`.

On Return

b Rigth hand side(s).
Type: **required**
An array of type real or complex, rank 2 and having the ALLOCATABLE attribute; will be allocated and filled in if the input file contains a right hand side, otherwise will be left in the UNALLOCATED state.

iret Error code.
Type: **required**
An integer value; 0 means no error has been detected.

mm_mat_write — Write a sparse matrix to a file in the MatrixMarket format

```
call mm_mat_write(a, mtitle, iret, iunit, filename)
```

Type: Asynchronous.

On Entry

a the sparse matrix to be written.
Type:**required**.
Specified as: a structured data of type psb_spmat_type.

mtitle Matrix title.
Type: **required**
A charachter variable holding a descriptive title for the matrix to be written to file.

filename The name of the file to be written to.
Type:**optional**.
Specified as: a character variable containing a valid file name, or -, in which case the default output unit 6 (i.e. standard output in Unix jargon) is used. Default: -.

iunit The Fortran file unit number.
Type:**optional**.
Specified as: an integer value. Only meaningful if filename is not -.

On Return

iret Error code.
Type: **required**
An integer value; 0 means no error has been detected.

10 Preconditioner routines

The base PSBLAS library contains the implementation of two simple preconditioning techniques:

- Diagonal Scaling
- Block Jacobi with ILU(0) factorization

The supporting data type and subroutine interfaces are defined in the module `psb_prec_mod`.

psb_precinit — Initialize a preconditioner

```
call psb_precinit(prec, ptype, info)
```

Type: Asynchronous.

On Entry

ptype the type of preconditioner. Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a character string, see usage notes.

On Exit

prec Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a preconditioner data structure psb_prec_type.

info Scope: **global**
Type: **required**
Intent: **out**.
Error code: if no error, 0 is returned.

Notes Legal inputs to this subroutine are interpreted depending on the *ptype* string as follows[3]:

NONE No preconditioning, i.e. the preconditioner is just a copy operator.

DIAG Diagonal scaling; each entry of the input vector is multiplied by the reciprocal of the sum of the absolute values of the coefficients in the corresponding row of matrix A;

BJAC Precondition by a factorization of the block-diagonal of matrix A, where block boundaries are determined by the data allocation boundaries for each process; requires no communication. Only the incomplete factorization $ILU(0)$ is currently implemented.

[3]The string is case-insensitive

psb_precbld — Builds a preconditioner

```
call psb_precbld(a, desc_a, prec, info)
```

Type: Synchronous.

On Entry

a the system sparse matrix. Scope: **local**
Type: **required**
Intent: **in**, target.
Specified as: a sparse matrix data structure psb_spmat_type.

prec the preconditioner.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: an already initialized precondtioner data structure psb_prec_type

desc_a the problem communication descriptor. Scope: **local**
Type: **required**
Intent: **in**, target.
Specified as: a communication descriptor data structure psb_desc_type.

On Return

prec the preconditioner.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a precondtioner data structure psb_prec_type

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_precaply — Preconditioner application routine

```
call psb_precaply(prec,x,y,desc_a,info,trans,work)
call psb_precaply(prec,x,desc_a,info,trans)
```

Type: Synchronous.

On Entry

prec the preconditioner. Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a preconditioner data structure psb_prec_type.

x the source vector. Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a double precision array.

desc_a the problem communication descriptor. Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a communication data structure psb_desc_type.

trans Scope:
Type: **optional**
Intent: **in**.
Specified as: a character.

work an optional work space Scope: **local**
Type: **optional**
Intent: **inout**.
Specified as: a double precision array.

On Return

y the destination vector. Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a double precision array.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

psb_precdescr — Prints a description of current preconditioner

```
call psb_precdescr(prec)
call psb_precdescr(prec, iout)
```

Type: Asynchronous.

On Entry

prec the preconditioner. Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a preconditioner data structure psb_prec_type.

iout output unit. Scope: **local**
Type: **optiona**
Intent: **in**.
Specified as: an integer number.

11 Iterative Methods

In this chapter we provide routines for preconditioners and iterative methods. The interfaces for Krylov subspace methods are available in the module `psb_krylov_mod`.

psb_krylov — Krylov Methods Driver Routine

This subroutine is a driver that provides a general interface for all the Krylov-Subspace family methods implemented in PSBLAS version 2.

The stopping criterion is the normwise backward error, in the infinity norm, i.e. the iteration is stopped when

$$err = \frac{\|r_i\|}{(\|A\|\|x_i\| + \|b\|)} < eps$$

or the 2-norm residual reduction

$$err = \frac{\|r_i\|}{\|b\|_2} < eps$$

according to the value passed through the istop argument (see later). In the above formulae, x_i is the tentative solution and $r_i = b - Ax_i$ the corresponding residual at the i-th iteration.

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

Type: Synchronous.

On Entry

method a string that defines the iterative method to be used. Supported values are:

CG: the Conjugate Gradient method;

CGS: the Conjugate Gradient Stabilized method;

BICG: the Bi-Conjugate Gradient method;

BICGSTAB: the Bi-Conjugate Gradient Stabilized method;

BICGSTABL: the Bi-Conjugate Gradient Stabilized method with restarting;

RGMRES: the Generalized Minimal Residual method with restarting.

a the local portion of global sparse matrix A.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_spmat_type.

prec The data structure containing the preconditioner.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_prec_type.

b The RHS vector.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a rank one array.

x The initial guess.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one array.

eps The stopping tolerance.
Scope: **global**
Type: **required**
Intent: **in**.
Specified as: a real number.

desc_a contains data structures for communications.
Scope: **local**
Type: **required**
Intent: **in**.
Specified as: a structured data of type psb_desc_type.

itmax The maximum number of iterations to perform.
Scope: **global**
Type: **optional**
Intent: **in**.
Default: $itmax = 1000$.
Specified as: an integer variable $itmax \geq 1$.

itrace If > 0 print out an informational message about convergence every $itrace$ iterations.
Scope: **global**
Type: **optional**
Intent: **in**.

irst An integer specifying the restart parameter.
Scope: **global**
Type: **optional**.
Intent: **in**.
Values: $irst > 0$. This is employed for the BiCGSTABL or RGMRES methods, otherwise it is ignored.

istop An integer specifying the stopping criterion.
Scope: **global**
Type: **optional**.
Intent: **in**.
Values: 1: use the normwise backward error, 2: use the scaled 2-norm of the residual. Default: 2.

On Return

x The computed solution.
Scope: **local**
Type: **required**
Intent: **inout**.
Specified as: a rank one array.

iter The number of iterations performed.
Scope: **global**
Type: **optional**
Intent: **out**.
Returned as: an integer variable.

err The convergence estimate on exit.
Scope: **global**
Type: **optional**
Intent: **out**.
Returned as: a real number.

cond An estimate of the condition number of matrix A; only available with the CG method.
Scope: **global**
Type: **optional**
Intent: **out**.
Returned as: a real number.

info Error code.
Scope: **local**
Type: **required**
Intent: **out**.
An integer value; 0 means no error has been detected.

References

[1] D. Barbieri, V. Cardellini, S. Filippone and D. Rouson *Design Patterns for Scientific Computations on Sparse Matrices*, HPSS 2011, Algorithms and Programming Tools for Next-Generation High-Performance Scientific Software, Bordeaux, Sep. 2011

[2] G. Bella, S. Filippone, A. De Maio and M. Testa, *A Simulation Model for Forest Fires*, in J. Dongarra, K. Madsen, J. Wasniewski, editors, Proceedings of PARA 04 Workshop on State of the Art in Scientific Computing, pp. 546–553, Lecture Notes in Computer Science, Springer, 2005.

[3] A. Buttari, D. di Serafino, P. D'Ambra, S. Filippone, 2LEV-D2P4: a package of high-performance preconditioners, Applicable Algebra in Engineering, Communications and Computing, Volume 18, Number 3, May, 2007, pp. 223-239

[4] P. D'Ambra, S. Filippone, D. Di Serafino On the Development of PSBLAS-based Parallel Two-level Schwarz Preconditioners Applied Numerical Mathematics, Elsevier Science, Volume 57, Issues 11-12, November-December 2007, Pages 1181-1196.

[5] Dongarra, J. J., DuCroz, J., Hammarling, S. and Hanson, R., An Extended Set of Fortran Basic Linear Algebra Subprograms, ACM Trans. Math. Softw. vol. 14, 1–17, 1988.

[6] Dongarra, J., DuCroz, J., Hammarling, S. and Duff, I., A Set of level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Softw. vol. 16, 1–17, 1990.

[7] J. J. Dongarra and R. C. Whaley, *A User's Guide to the BLACS v. 1.1*, Lapack Working Note 94, Tech. Rep. UT-CS-95-281, University of Tennessee, March 1995 (updated May 1997).

[8] I. Duff, M. Marrone, G. Radicati and C. Vittoli, *Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User Level Interface*, ACM Transactions on Mathematical Software, 23(3), pp. 379–401, 1997.

[9] I. Duff, M. Heroux and R. Pozo, *An Overview of the Sparse Basic Linear Algebra Subprograms: the New Standard from the BLAS Technical Forum*, ACM Transactions on Mathematical Software, 28(2), pp. 239–267, 2002.

[10] S. Filippone and M. Colajanni, *PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices*, ACM Transactions on Mathematical Software, 26(4), pp. 527–550, 2000.

[11] S. Filippone and A. Buttari, *Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003*, ACM Transactions on Mathematical Software, to appear.

[12] S. Filippone, P. D'Ambra, M. Colajanni, *Using a Parallel Library of Sparse Linear Algebra in a Fluid Dynamics Applications Code on Linux Clusters*, in G. Joubert, A. Murli, F. Peters, M. Vanneschi, editors, Parallel Computing - Advances & Current Issues, pp. 441–448, Imperial College Press, 2002.

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[14] Karypis, G. and Kumar, V., *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System*. Minneapolis, MN 55455: University of Minnesota, Department of Computer Science, 1995. Internet Address: `http://www.cs.umn.edu/~karypis`.

[15] Lawson, C., Hanson, R., Kincaid, D. and Krogh, F., Basic Linear Algebra Subprograms for Fortran usage, ACM Trans. Math. Softw. vol. 5, 38–329, 1979.

[16] Machiels, L. and Deville, M. *Fortran 90: An entry to object-oriented programming for the solution of partial differential equations.* ACM Trans. Math. Softw. vol. 23, 32–49.

[17] Metcalf, M., Reid, J. and Cohen, M. *Fortran 95/2003 explained.* Oxford University Press, 2004.

[18] Rouson, D.W.I., Xia, J., Xu, X.: Scientific Software Design: The Object-Oriented Way. Cambridge University Press (2011)

[19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference. Volume 1 - The MPI Core*, second edition, MIT Press, 1998.