PSBLAS-KINSOL interface

Istituto per le Applicazioni del Calcolo "M. Picone",
Consiglio Nazionale delle Ricerche
Pasqua D'Ambra
Fabio Durastante
Salvatore Filippone
PSBLAS 3.6.1 — Interface Version 1

April 16, 2020

Contents

1 The logic behind a PSBLAS instrumented application

The main aim of the PSBLAS library is the parallel implementation of iterative solvers for sparse linear systems,

$$A\mathbf{x} = \mathbf{b}, \qquad A \in \mathbb{K}^{n\times n}, \qquad \mathbb{K} = \mathbb{R}, \mathbb{C}, \tag{1}$$

through the distributed memory paradigm operating with message passing. The library includes all the needed routines for this task, e.g, functions for multiplying sparse matrices by dense matrices, for solving block diagonal systems with triangular diagonal entries or for preprocessing sparse matrices.

Owner computes rule

The pivotal choice to be made in this setting regards the distribution of the coefficient matrix A for the linear system (1). In PSBLAS this choice is based on the **owner computes rule** : each unknown is assigned to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations.

If A is obtained from the discretization of a Partial Differential Equation (PDE), this allocation strategy is equivalent to the choice of a partition of the mesh into *sub-domains*.

! $\rightarrow$ PSBLAS supports *any* distribution that keeps together the coefficients of each matrix row there are **no other** constraints on the variable assignment.

Any PSBLAS application will always start with the construction of the parallel environment, i.e., of an MPI (virtual) parallel machine, that we call here context by means of the psb_c_init function as

```
psb_i_t ictxt, iam, np;
ictxt = psb_c_init();
psb_c_info(ictxt,&iam,&np);
```

that creates a parallel environment on np processors $0, \ldots, \text{np}-1$, of which we are process iam.

The next step is represented by the need of subdividing the index space among processes, and this creates a mapping from the "global" numbering $1, \ldots, n$ to a "local" numbering in each process. This means that each process i will own a certain subset $1, \ldots, n_{\text{row}_i}$, each element of which corresponds to a certain element of $1 \ldots n$.

Therefore, after the initialization the first step is to establish an index space, and this is done with a call to one of the variants of the psb_cdall function to allocate a descriptor object psb_c_descriptor:

`psb_c_cdall_vg` : the association between an index and a process is specified via an integer vector vg[], each index $i \in \{1, \ldots, \text{ng}\}$ is assigned to process vg[i]. The vector vg[] must be identical on all calling processes, and its entries have the ranges $(0, \ldots, \text{np}-1)$ or $(1, \ldots, \text{np})$ according to the fact that psb_c_set_index_base(0) or psb_c_set_index_base(1) has been called at the beginning. The size ng is specified one can chose to use the entire vector vg[], thus having vg[ng].

Prototype

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

`psb_c_cdall_vl` : the association is done by specifying the list of indices vl[nl] assigned to the current process; thus, the global problem size nl is given by the range of the aggregate of the individual vectors vl[] specified in the calling processes. The subroutine will check how many times

each entry in the global index space $(1, \ldots, \text{nl})$ is specified in the input lists vl[], therefore it allows for the presence of overlap in the input, and checks for the "orphan" indices.

Prototype

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

`psb_c_cdall_nl` : produces a generalized block-row distribution of the number of indices belonging to the current process in which each process i gets assigned a consecutive chunk of nl global indices,

Prototype

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

for the case of a simple minded block distribution, i.e., the index space is first numbered sequentially in a standard way, then the corresponding vector is distributed according to a block distribution directive.

We consider as example the finite difference discretization of the following boundary value problem

$$-\frac{b_1\partial^2 u}{\partial x^2} - \frac{b_2\partial^2 u}{\partial y^2} - \frac{b_3\partial^2 u}{\partial z^2} + \frac{a_1\partial u}{\partial x} + \frac{a_2\partial u}{\partial y} + a_3\frac{a_3\partial u}{\partial z} = 0, \qquad (2)$$

for $(x, y, z) \in [0, 1]^3$, with Dirichlet boundary conditions, on a uniform grid with idim node per size. All the allocation procedure can be expressed as

```
psb_c_descriptor *cdh;
psb_i_t idim,nb,nlr, nl;
psb_l_t i,ng, *vl, k;

cdh=psb_c_new_descriptor();
psb_c_set_index_base(0);

/* Simple minded BLOCK data distribution */
ng = ((psb_l_t) idim)*idim*idim;
nb = (ng+np-1)/np;
nl = nb;
if ( (ng -iam*nb) < nl) nl = ng -iam*nb;
fprintf(stderr,"%d: Input data %d %ld %d %d\n",iam,idim,ng,nb
    , nl);
if ((vl=malloc(nb*sizeof(psb_l_t)))==NULL) {
  fprintf(stderr,"On %d: malloc failure\n",iam);
  psb_c_abort(ictxt);
}
i = ((psb_l_t)iam) * nb;
for (k=0; k<nl; k++)
  vl[k] = i+k;
psb_c_cdall_vl(nl,vl,ictxt,cdh);
```

Listing 1: "Example of allocation procedure for a 3D block data distribution"

Now that the initial distribution of the index space has been performed, we need to allocate dense vectors and sparse matrices on such index space, and thus we define the complete topology of our computational problem. Since our task is to use the capabilities of SUNDIALS-KINSOL to solve for nonlinear problems, here lies the core of the interfacing between the two codes. In the next two Sections 2, and Section 3 we describe such encapsulation for dense vector, and sparse matrices. Then in Section 4 we describe the interfacing with the **linear solvers** and **preconditioners**

Complete information on the PSBLAS data structures, and functions that are mentioned along the text can be found in [2].

2 The NVECTOR_PSBLAS implementation

The NVECTOR_PSBLAS implementation of the SUNDIALS NVECTOR module provides an interface to the PSBLAS code for handling distributed dense vectors. It defines the *content* field of `N_Vector` to be a structure containing the PSBLAS descriptor for the data distribution, a PSBLAS vector of double, and the PSBLAS communicator (context).

```
struct _N_VectorContent_PSBLAS {
  booleantype own_data;  /*ownership of data*/
  psb_c_descriptor *cdh; /*descriptor for data distribution*/
  psb_c_dvector *pvec;   /*PSBLAS vector*/
  int ictxt;             /*PSBLAS communicator*/
};
```

! → All the vectors that have to interact needs to be instantiated on the same parallel context ictxt, and on the same data distribution cdh.

The header file to include when using this module is `nvector_psblas.h`. The installed module library to link to is `sundials_nvecpsblas.lib` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

2.1 NVECTOR_PSBLAS accessor macros

The following macros are provided to access the content of a NVECTOR_PSBLAS vector. The suffix `_P` in the names denotes the fact that the data are in distributed memory.

```
#define NV_CONTENT_P(v)     ((N_VectorContent_PSBLAS)(v->
    content))
#define NV_DESCRIPTOR_P(v)(NV_CONTENT_P(v)->cdh)
#define NV_OWN_DATA_P(v)   (NV_CONTENT_P(v)->own_data)
#define NV_PVEC_P(v)       (NV_CONTENT_P(v)->pvec)
#define NV_ICTXT_P(v)      (NV_CONTENT_P(v)->ictxt)
```

NV_CONTENT_P(v) — this macro gives access to the contents of the PSBLAS vector `N_Vector`.

NV_DESCRIPTOR_P(v), NV_OWN_DATA_P(v), NV_PVEC_P(v) — these macros give instead individual access to the parts of the content of a PSBLAS parallel `N_Vector`.

NV_ICTXT_P(v) — this macro provides the PSBLAS context used by the NVECTOR_PSBLAS vectors.

2.2 NVECTOR_PSBLAS functions

The NVECTOR_PSBLAS implementation provides PSBLAS implementations of all the vectors operations listed in Tables 6.2, 6.3, and 6.4 of the original KINSOL library [1]. Following the standard nomenclature of the SUNDIALS library, their names are obtained from the ones listed in Tables 6.2, 6.3, and 6.4 by appending the suffix `_PSBLAS`. The NVECTOR_PSBLAS implementation provides the following additional user–callable routine:

`N_VAsb_PSBLAS` : This routine assemble the NVector after that all the elements have been inserted into it, i.e., after that all the calls to the `N_VMake_PSBLAS` routine have been completed. This is substantially a wrapper for the PSBLAS function psb_c_dgeasb.

Prototype — `void N_VAsb_PSBLAS(N_Vector v)`

2.2.1 Description of the NVECTOR_PSBLAS functions

`N_VNew_PSBLAS` : This function creates and allocates memory for a parallel vector on the PSBLAS context ictxt with the communicator cdh

Prototype N_Vector N_VNew_PSBLAS(int ictxt, psb_c_descriptor *cdh);

`N_VNewEmpty_PSBLAS` : This function creates a new PSBLAS vector with empty data array.

Prototype N_Vector N_VNewEmpty_PSBLAS(int ictxt, psb_c_descriptor *cdh);

`N_VMake_PSBLAS` : Function to create a PSBLAS `N_Vector` with user data component. This function is substantially a wrapper for the PSBLAS function psb_c_dgeins.

Prototype N_Vector N_VMake_PSBLAS(int ictxt, psb_c_descriptor *cdh,psb_i_t m, psb_i_t *irow,double *val);

The PSBLAS context ictxt with the communicator cdh are the one defined for the whole programs, the integer m is the number of rows in val [] to be inserted, the array of integers irow is the indices of the rows to be inserted. Specifically, row i of val will be inserted into the local row corresponding to the global index row index row[i].

! → This routine does not assemble the final vector. After the insertion of all the elements has been completed then the vector should be assembled by means of the `N_VAsb_PSBLAS` routine.

`N_VCloneVectorArray_PSBLAS` : This function creates an array of new parallel vectors (by cloning) an array of count parallel vectors v.

Prototype N_Vector *N_VCloneVectorArray_PSBLAS(int count, N_Vector w)

`N_VCloneVectorArrayEmpty_PSBLAS` : This function creates an array of count new parallel vectors with empty data array on the same communicator and context of the vector w.

Prototype N_Vector *N_VCloneVectorArrayEmpty_PSBLAS(int count, N_Vector w)

`N_VDestroyVectorArray_PSBLAS` : This function to frees an array of count `N_Vectors` created with `N_VCloneVectorArray_PSBLAS`

Prototype void N_VDestroyVectorArray_PSBLAS(N_Vector *vs, int count)

`N_VGetLength_PSBLAS` : This function returns the *global* vector length, this is substantially a wrapper for the PSBLAS function psb_c_cd_get_global_rows.

Prototype sunindextype N_VGetLength_PSBLAS(N_Vector v)

`N_VGetLocalLength_PSBLAS` : This function returns the *local* vector length, this is substantially a wrapper for the PSBLAS function psb_c_cd_get_local_rows.

Prototype sunindextype N_VGetLocalLength_PSBLAS(N_Vector v)

`N_VPrint_PSBLAS` : This function prints the local data in a parallel vector to stdout.

Prototype void N_VPrint_PSBLAS(N_Vector x)

`N_VPrintFile_PSBLAS` : This function prints the local data in a parallel vector to outfile.

Prototype void N_VPrintFile_PSBLAS(N_Vector x, FILE* outfile)

`N_VGetVectorID_PSBLAS` : This function returns the SUNDIALS identificative for the PSBLAS vector, since this is a custom implementation it returns the integer constant SUNDIALS_NVEC_CUSTOM.

Prototype N_Vector_ID N_VGetVectorID_PSBLAS(N_Vector v)

`N_VCloneEmpty_PSBLAS` : Clones a NVECTOR_PSBLAS with a NULL pvec field, and with value SUNFALSE in the own_data field.

Prototype N_Vector N_VCloneEmpty_PSBLAS(N_Vector w)

`N_VClone_PSBLAS` : Clones a NVECTOR_PSBLAS allocating its memory following the same communicator of the cloned one.

Prototype N_Vector N_VClone_PSBLAS(N_Vector w)

`N_VDestroy_PSBLAS` : Destroys a NVECTOR_PSBLAS freeing both the memory allocated for the corresponding PSBLAS vector, and the memory allocated for the NVECTOR_PSBLAS structure.

Prototype void N_VDestroy_PSBLAS(N_Vector v)

`N_VSpace_PSBLAS` : Returns storage requirements for one NVECTOR_PSBLAS. lrw contains the number of realtype words and liw contains the number of integer words.

Prototype void N_VSpace_PSBLAS(N_Vector v, sunindextype *lrw,sunindextype *liw)

`N_VGetArrayPointer_PSBLAS` : Returns a pointer to a realtype array from the NVECTOR_PSBLAS, this is the local portion of the distributed PSBLAS vector encapsulated in the `N_Vector` object.

Prototype realtype *N_VGetArrayPointer_PSBLAS(N_Vector v)

`N_VSetArrayPointer_PSBLAS` : This function is a dummy function, in PSBLAS we use **allocatable** objects for the local part of the distributed vector, and, moreover, we assume having an arbitrary distribution of the indexes.

Prototype void N_VSetArrayPointer_PSBLAS(psb_c_dvector *v_data, N_Vector v)

`N_VLinearSum_PSBLAS` : Performs the AXPBY BLAS operation between to NVECTOR_PSBLAS, the result can be both out-of- and in-place.

Prototype void N_VLinearSum_PSBLAS(realtype a, N_Vector x, realtype b, N_Vector y, N_Vector z)

`N_VConst_PSBLAS` : Sets all components of the NVECTOR_PSBLAS to a constant value and assembles it, the user **does not need** to assembly it.

Prototype void N_VConst_PSBLAS(realtype c, N_Vector z)

`N_VProd_PSBLAS` : Performs the entry-wise multiplication of two NVECTOR_PSBLAS, the result can be both out-of- and in-place.

Prototype void N_VProd_PSBLAS(N_Vector x, N_Vector y, N_Vector z)

`N_VDiv_PSBLAS` : Performs the entry-wise division of two NVECTOR_PSBLAS, the result can be both out-of- and in-place, it does not check for possible zero entries in the denumerator. It is up to the user to guarantee that this does not happens.

Prototype void N_VDiv_PSBLAS(N_Vector x, N_Vector y, N_Vector z)

`N_VScale_PSBLAS` : Scales an NVECTOR_PSBLAS by a scalar c, the result can be both out-of- and in-place.

Prototype void N_VScale_PSBLAS(realtype c, N_Vector x, N_Vector z)

`N_VAbs_PSBLAS` : Sets the entries of an NVECTOR_PSBLAS to the absolute values of the entries of the input.

Prototype void N_VAbs_PSBLAS(N_Vector x, N_Vector z)

`N_VInv_PSBLAS` : Sets the entries of an NVECTOR_PSBLAS to the inverse of the entries of the input. It does not check for possible zero entries in the vector. It is up to the user to guarantee that this does not happens.

Prototype `void N_VInv_PSBLAS(N_Vector x, N_Vector z)`

`N_VAddConst_PSBLAS` : Adds a scalar to all components of an NVECTOR_PSBLAS and returns the result in another NVECTOR_PSBLAS object.

Prototype `void N_VAddConst_PSBLAS(N_Vector x, realtype b, N_Vector z)`

`N_VDotProd_PSBLAS` : Compute the dot product of two NVECTOR_PSBLAS objects.

Prototype `realtype N_VDotProd_PSBLAS(N_Vector x, N_Vector y)`

`N_VMaxNorm_PSBLAS` : Compute the max norm of an NVECTOR_PSBLAS object.

Prototype `realtype N_VMaxNorm_PSBLAS(N_Vector x)`

`N_VWrmsNorm_PSBLAS` : Compute the weighted (by the size) 2-norm of an NVECTOR_PSBLAS object.

Prototype `realtype N_VWrmsNorm_PSBLAS(N_Vector x, N_Vector w)`

`N_VWrmsNormMask_PSBLAS` : Returns the weighted root mean square norm of the NVECTOR_PSBLAS `x` with realtype weight vector `w` built using only the elements of `x` corresponding to positive elements of the NVECTOR_PSBLAS `id`.

Prototype `realtype N_VWrmsNormMask_PSBLAS(N_Vector x,N_Vector w,N_Vector id)`

`N_VMin_PSBLAS` : Gives back the minimum entry of an NVECTOR_PSBLAS object.

Prototype `realtype N_VMin_PSBLAS(N_Vector x)`

`N_VWL2Norm_PSBLAS` : Returns the weighted Euclidean 2-norm of the NVECTOR_PSBLAS `x` with realtype weight vector `w`.

Prototype `realtype N_VWL2Norm_PSBLAS(N_Vector x, N_Vector w)`

`N_VL1Norm_PSBLAS` : Computes the 1-norm of an NVECTOR_PSBLAS object.

Prototype `realtype N_VL1Norm_PSBLAS(N_Vector x)`

`N_VCompare_PSBLAS` : Compares the components of the NVECTOR_PSBLAS `x` to the realtype scalar `c` and returns an NVECTOR_PSBLAS `z` such that

$$z_i = \begin{cases} 1.0, & |x_i| \geq c, \\ 0.0, & |x_i| \leq c. \end{cases}$$

Prototype `void N_VCompare_PSBLAS(realtype c, N_Vector x, N_Vector z)`

`N_VInvTest_PSBLAS` : Sets the entries of an NVECTOR_PSBLAS to the inverse of the entries of the input. It checks for possible zero entries in the vector. This routine returns a boolean assigned to SUNTRUE if all components of the vector are nonzero (successful inversion) and returns SUNFALSE otherwise.

Prototype `booleantype N_VInvTest_PSBLAS(N_Vector x, N_Vector z)`

`N_VConstrMask_PSBLAS` : Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to SUNFALSE if any element failed the constraint test and assigned to SUNTRUE if all passed. It also sets a mask vector `m`, with elements equal to

1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.

Prototype booleantype N_VConstrMask_PSBLAS(N_Vector c, N_Vector x, N_Vector m)

`N_VMinQuotient_PSBLAS` : This routine returns the minimum of the quotients obtained by term-wise dividing num[i] by denom[i] . A zero element in denom will be skipped. If no such quotients are found, then the large value BIG REAL (defined in the header file sundials `types.h`) is returned.

Prototype realtype N_VMinQuotient_PSBLAS(N_Vector num, N_Vector denom)

2.2.2 Fused operation

`N_VLinearCombination_PSBLAS` : This routine computes the linear combination of nv vectors with n elements

$$z_i = \sum_{j=0}^{n_v-1} c_j x_{j,i}, \qquad i = 0, \ldots, n-1$$

where c is an array of nv scalars, X is an array of NVECTOR_PSBLAS. When z is one of the vectors in X, then it is assumed to be the first vector in the vector array.

Prototype int N_VLinearCombination_PSBLAS(int nvec, realtype* c, N_Vector* V,N_Vector z)

`N_VScaleAddMulti_PSBLAS` : This routine scales and adds one vector to nv vectors with n elements:

$$z_{j,i} = c_j x_i + y_{j,i}, \qquad j = 0, \ldots, n_v - 1, \qquad i = 0, \ldots, n-1,$$

where c is an array of nv scalars.

Prototype int N_VScaleAddMulti_PSBLAS(int nvec, realtype* a, N_Vector x, N_Vector* Y, N_Vector* Z)

`N_VDotProdMulti_PSBLAS` : This routine computes the dot product of a vector with nv other vectors.

Prototype int N_VDotProdMulti_PSBLAS(int nvec, N_Vector x,N_Vector *Y, realtype* dotprods)

2.2.3 Vector array operations

`N_VLinearSumVectorArray_PSBLAS` : This routine computes the linear sum of two vector arrays containing nv vectors of n elements:

$$z_{j,i} = a x_{j,i} + b y_{j,i}, \qquad i = 0, \ldots, n-1, \qquad j = 0, \ldots, n_v - 1.$$

Prototype int N_VLinearSumVectorArray_PSBLAS(int nvec, realtype a, N_Vector* X, realtype b, N_Vector* Y, N_Vector* Z)

`N_VScaleVectorArray_PSBLAS` : This routine scales each vector of n elements in a vector array of n_v vectors by a (potentially different) constant:

$$z_{j,i} = c_j x_{j,i}, \qquad i = 0, \ldots, n-1, \qquad j = 0, \ldots, n_v - 1.$$

Prototype int N_VScaleVectorArray_PSBLAS(int nvec, realtype* c,N_Vector* X, N_Vector* Z)

`N_VConstVectorArray_PSBLAS` : This routine sets each element in a vector of n elements in a vector array of nv vectors to the same value.

Prototype `int N_VConstVectorArray_PSBLAS(int nvecs, realtype c,N_Vector* Z)`

`N_VWrmsNormVectorArray_PSBLAS` : This routine computes the weighted root mean square norm of nv vectors with n elements.

Prototype `int N_VWrmsNormVectorArray_PSBLAS(int nvecs, N_Vector* X,N_Vector* W, realtype* nrm)`

`N_VWrmsNormMaskVectorArray_PSBLAS` : This routine computes the masked weighted root mean square norm of nv vectors with n elements.

Prototype `int N_VWrmsNormMaskVectorArray_PSBLAS(int nvec, N_Vector* X,N_Vector* W, N_Vector id,realtype* nrm)`

`N_VScaleAddMultiVectorArray_PSBLAS` : This routine scales and adds a vector in a vector array of nv vectors to the corresponding vector in ns vector arrays.

Prototype `int N_VScaleAddMultiVectorArray_PSBLAS(int nvec, int nsum, realtype* a, N_Vector* X, N_Vector** Y, N_Vector** Z)`

`N_VLinearCombinationVectorArray_PSBLAS` : This routine computes the linear combination of ns vector arrays containing nv vectors with n elements.

Prototype `int N_VLinearCombinationVectorArray_PSBLAS(int nvec, int nsum, realtype* c, N_Vector** X, N_Vector* Z)`

By default all fused and vector array operations are disabled in the nvector PSBLAS module. The following additional user-callable routines are provided to enable or disable fused and vector array operations for a specific vector.

```
int N_VEnableFusedOps_PSBLAS(N_Vector v, booleantype tf);
int N_VEnableLinearCombination_PSBLAS(N_Vector v, booleantype
    tf);
int N_VEnableScaleAddMulti_PSBLAS(N_Vector v, booleantype tf);
int N_VEnableDotProdMulti_PSBLAS(N_Vector v, booleantype tf);
int N_VEnableLinearSumVectorArray_PSBLAS(N_Vector v,
    booleantype tf);
int N_VEnableScaleVectorArray_PSBLAS(N_Vector v, booleantype
    tf);
int N_VEnableConstVectorArray_PSBLAS(N_Vector v, booleantype
    tf);
int N_VEnableWrmsNormVectorArray_PSBLAS(N_Vector v,
    booleantype tf);
int N_VEnableWrmsNormMaskVectorArray_PSBLAS(N_Vector v,
    booleantype tf);
int N_VEnableScaleAddMultiVectorArray_PSBLAS(N_Vector v,
    booleantype tf);
int N_VEnableLinearCombinationVectorArray_PSBLAS(N_Vector v,
    booleantype tf);
```

3 The SUNMATRIX_PSBLAS implementation

The SUNMATRIX_PSBLAS implementation of the SUNDIALS SUNMATRIX module provides an interface to the PSBLAS code for handling distributed sparse matrices. It defines the *content* field of `SUNMATRIX` to be a structure containing the PSBLAS descriptor for the data distribution, a PSBLAS sparse matrix object of double, and the PSBLAS communicator (context).

```
struct _SUNMatrixContent_PSBLAS {
 psb_c_descriptor *cdh;        /* descriptor for data
    distribution */
 psb_c_dspmat   *ah;           /* PSBLAS sparse matrix
        */
 int ictxt;                    /* PSBLAS communicator
        */
};
```

! → All the matrices that have to interact needs to be instantiated on the same parallel context ictxt, and on the same data distribution cdh, and this holds also for the vectors for the mixed type operation.

The header file to include when using this module is `sunmatrix_psblas.h`. The installed module library to link to is `libsundials_sunmatrixpsblas.a` where `.lib` is typically `.so` for shared libraries and `.a` for static libraries.

3.1 SUNMATRIX_PSBLAS accessor macros

The following macros are provided to access the content of a SUNMATRIX_PSBLAS matrix. The suffix `_P` in the names denotes the fact that the data are in distributed memory.

```
#define SM_CONTENT_P(A)      ( (SUNMatrixContent_PSBLAS)(A->
    content) )
#define SM_DESCRIPTOR_P(A)   ( SM_CONTENT_P(A)->cdh )
#define SM_PMAT_P(A)         ( SM_CONTENT_P(A)->ah )
#define SM_ICTXT_P(A)        ( SM_CONTENT_P(A)->ictxt )
```

SM_CONTENT_P(A) this macro gives access to the contents of the PSBLAS matrix `SUNMATRIX`.

SM_DESCRIPTOR_P(A), SM_PMAT_P(A) these macros give instead individual access to the parts of the content of a PSBLAS parallel `SUNMATRIX`.

SM_CONTENT_P(A) this macro provides the PSBLAS context used by the SUNMATRIX_PSBLAS sparse matrices.

In PSBLAS every sparse matrix has an associated state, which can take one of the following values:

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

ASSEMBLED : is the 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.*

! →

3.2 SUNMATRIX_PSBLAS functions

The SUNMATRIX_PSBLAS implementation provides PSBLAS implementations of all the sparse matrix operations listed in Table 7.2 of the original KINSOL library [1]. Following the standard nomenclature of the SUNDIALS library, their names are obtained from the ones listed in Table 7.2 by appending the suffix `_PSBLAS`. The SUNMATRIX_PSBLAS implementation provides the following additional user–callable routines

`SUNMatAsb_PSBLAS` : This routine assemble the SUNMATRIX after that all the elements have been inserted into it, i.e., after that all the calls to the `SUNMatIns_PSBLAS` routine have been completed. This is substantially a wrapper for the PSBLAS function psb_c_dspasb.

Prototype `int SUNMatAsb_PSBLAS(SUNMatrix A);`

`SUNMatIns_PSBLAS` : This routine inserts a set of coefficients into a sparse matrix. On entry to this routine the descriptor may be in either the BUILD or ASSEMBLED state, while the sparse matrix may be in either the BUILD or UPDATE state. We stress that it mandatory that if the descriptor is in the BUILD state, then also the sparse matrix *must* be in the BUILD state, since adding entries to the matrix causes internal calls altering the structure of the communication pattern for the distributed object. The insert of the element in the matrix si assumed to be in the COO format, thus the coefficients to be inserted are represented by the ordered triples irw[i], icl[i], val[i], for $i = 1, \ldots, nz$; these triples should belong to the current process, i.e., the index irw[i] should be one of the local indices, but are otherwise arbitrary, it this is not the case any coefficients from matrix rows not owned by the calling process are silently ignored.

Prototype `int SUNMatIns_PSBLAS(psb_i_t nz, const psb_l_t *irw, const psb_l_t *icl, const psb_d_t *val,SUNMatrix A);`

3.2.1 Description of the SUNMATRIX_PSBLAS functions

`SUNPSBLASMatrix` : Define and implement user-callable constructor routines to create a SUNMatrix on the PSBLAS context ictxt with the communicator cdh, and with the content field and ops pointing to the matrix operations defined in the following. The matrix is initialized in the BUILD state.

Prototype `SUNMatrix SUNPSBLASMatrix(int ictxt, psb_c_descriptor *cdh)`

`SUNPSBLASMatrix_Print` : This function prints the content of a dense SUNMatrix in MatrixMarket format to file specified by `filename`, the routine takes care of creating/opening the file, a separated one for each process in which the local part (without any eventual overlap) is printed.

Prototype `void SUNPSBLASMatrix_Print(SUNMatrix A,char *matrixtitle,char *filename )`

`SUNPSBLASMatrix_Rows` : This function returns the number of (global) rows in the PSBLAS SUNMatrix.

Prototype `sunindextype SUNPSBLASMatrix_Rows(SUNMatrix A)`

`SUNPSBLASMatrix_Columns` : This function returns the number of (global) columns in the PSBLAS SUNMatrix.

Prototype `sunindextype SUNPSBLASMatrix_Columns(SUNMatrix A)`

`SUNPSBLASMatrix_NNZ` : This function returns the (global) number of non-zero entries in the PSBLAS SUNMatrix.

Prototype `sunindextype SUNPSBLASMatrix_NNZ(SUNMatrix A)`

`SUNMatGetID_PSBLAS` : Returns the type identifier for the matrix `A`, since this is a custom implementation it returns the value `SUNMATRIX_CUSTOM`.

Prototype `SUNMatrix_ID SUNMatGetID_PSBLAS(SUNMatrix A)`

`SUNMatClone_PSBLAS` : Creates a new SUNMatrix on the same descriptor of an existing matrix `A` and sets the `ops` field. It **does not copy** the matrix, but rather allocates storage for the new matrix leaving the new matrix in the BUILD state.

Prototype: SUNMatrix SUNMatClone_PSBLAS(SUNMatrix A)

`SUNMatDestroy_PSBLAS` : Destroys the SUNMatrix `A` and frees memory allocated for its internal data. This routine **does not free** the communicator, in general there are many objects insisting on the same communicator, therefore it should be destroyed/freed on its own.

Prototype: void SUNMatDestroy_PSBLAS(SUNMatrix A)

`SUNMatZero_PSBLAS` : Performs the operation $(A)_{i,j} = 0$ for all entries of the matrix `A`. The return value is an integer flag denoting success/failure of the operation. The PSBLAS at the end of this operation is left in the UPDATE state.

Prototype: int SUNMatZero_PSBLAS(SUNMatrix A)

`SUNMatCopy_PSBLAS` : Performs the operation $(B)_{i,j} = (A)_{i,j}$ for all entries of the matrices `A` and `B`. The return value is an integer flag denoting success/failure of the operation. The matrix `B` inherits the state of the matrix `A`.

Prototype: int SUNMatCopy_PSBLAS(SUNMatrix A, SUNMatrix B)

`SUNMatScaleAdd_PSBLAS` : Performs the operation $A = cA + B$. The return value is an integer flag denoting success/failure of the operation. This function assumes that both the matrices are defined on the same communicator.

Prototype: int SUNMatScaleAdd_PSBLAS(realtype c, SUNMatrix A, SUNMatrix B)

`SUNMatScaleAddI_PSBLAS` : Performs the operation $A = cA + I$. The return value is an integer flag denoting success/failure of the operation.

Prototype: int SUNMatScaleAddI_PSBLAS(realtype c, SUNMatrix A)

`SUNMatMatvec_PSBLAS` : Performs the matrix-vector product operation, $\mathbf{y} = A\mathbf{x}$. It should only be called with vectors $\mathbf{x}$ and $\mathbf{y}$ that are compatible with the matrix A, i.e., they should be both defined on the same communicator and have the same dimensions. The return value is an integer flag denoting success/failure of the operation.

Prototype: int SUNMatMatvec_PSBLAS(SUNMatrix A, N_Vector x, N_Vector y)

`SUNMatSpace_PSBLAS` : This function is advisory only, for use in determining a user's total space requirements, for this module it is a dummy function always returning true.

Prototype: int SUNMatSpace_PSBLAS(SUNMatrix A, long int *lenrw, long int *leniw)

3.2.2 An example of matrix assembly

As an example of usage of this matrix routines we consider the same boundary value problem in (2), to build the matrix A associated to the centered finite difference discretization

$$\ldots \tag{3}$$

To this end we use the same communicator `cdh` and context `ictxt` we have allocated in Listing 1, by simply doing:

```
A = SUNPSBLASMatrix(ictxt, cdh);
matgen(ictxt, nl, idim, vl,A);
psb_c_cdasb(cdh);
SUNMatAsb_PSBLAS(A);
```

We have firs initialized the sparse PSBLAS matrix inside the SUNMATRIX container, then by using the `matgen` allocation routine in Listing 2 we have populate the sparse matrix, and finally assembled both the descriptor, and the sparse matrix.

The complete example can be found in `examples/sunmatrix/psblas`.

4 The SUNLINSOL_PSBLAS implementation

4.1 Algebraic Multigrid Preconditioners

A Matrix assembly routine

The following routine loops through the local entries of the communicator allocate in Listing 1, and uses the `SUNMatIns_PSBLAS` routine to insert the entries in the sparse matrix. The auxiliary functions $\{a_i, b_i, g\}_{i=1}^{3}$ take care of the coefficient functions in (2).

```
psb_i_t matgen(psb_i_t ictxt, psb_i_t nl, psb_i_t idim,
    psb_l_t vl[],SUNMatrix A)
{
psb_i_t iam, np;
psb_l_t ix, iy, iz, el,glob_row;
psb_i_t i, k, info;
double x, y, z, deltah, sqdeltah, deltah2;
double val[10*NBMAX], zt[NBMAX];
psb_l_t irow[10*NBMAX], icol[10*NBMAX];

info = 0;
psb_c_info(ictxt,&iam,&np);
deltah = (double) 1.0/(idim+1);
sqdeltah = deltah*deltah;
deltah2  = 2.0* deltah;
psb_c_set_index_base(0);
for (i=0; i<nl;  i++) {
glob_row=vl[i];
el=0;
ix = glob_row/(idim*idim);
iy = (glob_row-ix*idim*idim)/idim;
iz = glob_row-ix*idim*idim-iy*idim;
x=(ix+1)*deltah;
y=(iy+1)*deltah;
z=(iz+1)*deltah;
zt[0] = 0.0;
/*  internal point: build discretization */
/*  term depending on   (x-1,y,z)        */
val[el] = -a1(x,y,z)/sqdeltah-b1(x,y,z)/deltah2;
if (ix==0) {
zt[0] += g(0.0,y,z)*(-val[el]);
} else {
icol[el]=(ix-1)*idim*idim+(iy)*idim+(iz);
el=el+1;
}
/*  term depending on     (x,y-1,z) */
val[el]  = -a2(x,y,z)/sqdeltah-b2(x,y,z)/deltah2;
if (iy==0) {
zt[0] += g(x,0.0,z)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy-1)*idim+(iz);
el=el+1;
}
/* term depending on     (x,y,z-1)*/
```

```
val[el]=-a3(x,y,z)/sqdeltah-b3(x,y,z)/deltah2;
if (iz==0) {
zt[0] += g(x,y,0.0)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy)*idim+(iz-1);
el=el+1;
}
/* term depending on     (x,y,z)*/
val[el]=2.0*(a1(x,y,z)+a2(x,y,z)+a3(x,y,z))/sqdeltah + c(x,y,
    z);
icol[el]=(ix)*idim*idim+(iy)*idim+(iz);
el=el+1;
/*  term depending on     (x,y,z+1) */
val[el] = -a3(x,y,z)/sqdeltah+b3(x,y,z)/deltah2;
if (iz==idim-1) {
zt[0] += g(x,y,1.0)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy)*idim+(iz+1);
el=el+1;
}
/* term depending on     (x,y+1,z) */
val[el] = -a2(x,y,z)/sqdeltah+b2(x,y,z)/deltah2;
if (iy==idim-1) {
zt[0] += g(x,1.0,z)*(-val[el]);
} else {
icol[el]=(ix)*idim*idim+(iy+1)*idim+(iz);
el=el+1;
}
/*  term depending on     (x+1,y,z) */
val[el] = -a1(x,y,z)/sqdeltah+b1(x,y,z)/deltah2;
if (ix==idim-1) {
zt[0] += g(1.0,y,z)*(-val[el]);
} else {
icol[el]=(ix+1)*idim*idim+(iy)*idim+(iz);
el=el+1;
}
for (k=0; k<el; k++) irow[k]=glob_row;
if ((info=SUNMatIns_PSBLAS(el,irow,icol,val,A))!=0)
fprintf(stderr,"From psb_c_dspins: %d\n",info);
}

return(info);
}
```

Listing 2: "Allocation routine for the discrete boundary value problem (2)."

Observe that to speed-up the insertion procedure we are collecting together a certain number of rows to be inserted, specifically `10*NBMAX` for a defined value of NBMAX. It is clearly possible to execute one call for each nonzero coefficient, however this would have a substantial computational overhead. Therefore packing a "certain amount of data" (`10*NBMAX`) into each call to the insertion routine is advisable. The best performing value of `NBMAX` depends on both the architecture of the computer being used and on the problem structure.

References

[1] Aaron M. Collier et al. *User Documentation for kinsol v4.1.0*. Center for Applied Scientific Computing Lawrence Livermore National Laboratory. URL: `https://computing.llnl.gov/sites/default/files/public/kin_guide.pdf`.

[2] S. Filippone and A. Buttari. *PSBLAS 3.6.1 User's guide. A reference guide for the Parallel Sparse BLAS library*. Cranfield University, Centre for Computational Engineering Sciences. URL: `https://github.com/sfilippone/psblas3/blob/development/docs/psblas-3.6.pdf`.