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 9, 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

$$-\frac{b_1\partial^2 u}{\partial x^2} - \frac{b_2\partial^2 u}{\partial y^2} + \frac{a_1\partial u}{\partial x} + \frac{a_2\partial u}{\partial y} + a_3 u = 0, \qquad (x, y, z) \in [0, 1]^3,$$

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;
```

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 routines.

`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_VAss_PSBLAS` 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)

`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_Vector`s 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)

3 The SUNMATRIX_PSBLAS implementation

4 The SUNLINSOL_PSBLAS implementation

4.1 Algebraic Multigrid Preconditioners

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`.