

The implementation of the Sparse BLAS in Fortran 95¹

Iain S. Duff²
Christof Vömel³

Technical Report TR/PA/01/27
September 10, 2001

CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

ABSTRACT

The Basic Linear Algebra Subprograms for sparse matrices (Sparse BLAS) as defined by the Blas Technical Forum are a set of routines providing basic operations for sparse matrices and vectors. A principal goal of the Sparse BLAS standard is to aid in the development of iterative solvers for large sparse linear systems by specifying on the one hand interfaces for a high-level description of vector and matrix operations for the algorithm developer and on the other hand leaving enough freedom for vendors to provide the most efficient implementation of the underlying algorithms for their specific architectures.

The Sparse BLAS standard defines interfaces and bindings for the three target languages: C, Fortran 77 and Fortran 95. We describe here our Fortran 95 implementation intended as a reference model for the Sparse BLAS. We identify the underlying complex issues of the representation and the handling of sparse matrices and give suggestions to other implementors of how to address them.

Keywords: unstructured sparse matrices, sparse data structures, programming standard, iterative linear solvers, BLAS, Sparse BLAS

AMS(MOS) subject classifications: 65F05, 65F50.

¹Current reports available at http://www.cerfacs.fr/algor/algo_reports.html.

²duff@cerfacs.fr. Also at Atlas Centre, RAL, Oxon OX11 0QX, England.

³voemel@cerfacs.fr.

Contents

1	Introduction	1
2	The Sparse BLAS functionalities	2
2.1	Level 1 Sparse BLAS functionalities	2
2.2	Level 2 Sparse BLAS functionalities	2
2.3	Level 3 Sparse BLAS functionalities	3
2.4	Routines for the creation of sparse matrices	3
3	Handle creation and matrix data initialisation	3
4	Level 1 subroutines	10
5	Level 2 and Level 3 subroutines	11
5.1	Product of a sparse matrix with one or many dense vectors	11
5.2	Solution of a sparse triangular system with one or many dense right-hand sides	12
6	Releasing matrix handles	13
7	Some remarks on using Fortran 95	14
8	A sample program	14
9	Software availability and final remarks	17
9.1	Download location	17
9.2	Code generation	17
9.3	Final remarks	17

1 Introduction

The Basic Linear Algebra Subprograms (BLAS), which provide essential functionalities for dense matrix and vector operations, are a milestone in the history of numerical software. BLAS have been proposed for operations on dense matrices for some time, with the original paper for vector operations (Level 1 BLAS) appearing in 1979 [11]. This was followed by the design of kernels for matrix-vector operations (Level 2 BLAS) [4] and matrix-matrix operations (Level 3 BLAS) [3]. The Level 3 BLAS have proved to be particularly powerful for obtaining close to peak performance on many modern architectures since they amortize the cost of obtaining data from main memory by reusing data in the cache or high level memory.

For some years it has been realized that the BLAS standard needed updating and a BLAS Technical Forum was coordinated and has recently published a new standard [1]. Some of the main issues included in the new standard are added functionality, extended and mixed precision, and basic subprograms for sparse matrices (Sparse BLAS). The need for the latter is particularly important for the iterative solution of large sparse systems of linear equations.

As in the dense case, the Sparse BLAS enables the algorithm developer to rely on a standardized library of frequently occurring linear algebra operations and allows code to be written in a meta-language that uses these operations as building blocks. Additionally, vendors can provide implementations that are specifically tuned and tested for individual machines to promote the use of efficient and robust codes. The development of the Sparse BLAS standard has its roots in [2] and [8], the first proposals for Level 1 and Level 3 kernels for the Sparse BLAS. While the final standard [7] has evolved from these proposals, these papers are not only of historical interest but also contain suggestions for the implementor which are deliberately omitted in the final standard.

Similarly to the BLAS, the Sparse BLAS provides operations at three levels, although it includes only a small subset of the BLAS functionality. Level 1 covers basic operations on sparse and dense vectors, Level 2 and Level 3 provide sparse matrix multiply and sparse triangular solve on dense systems that may be vectors (Level 2) or matrices (Level 3). We emphasize that the standard is mainly intended for sparse matrices without a special structure. This has a significant influence on the complexity of the internal routines for data handling. Depending on the matrix, the algorithm used, and the underlying computing architecture, an implementor has to choose carefully an internal representation of the sparse matrix.

The standard defines the following procedure for the use of the Sparse BLAS. First, the given matrix data has to be passed to an initialisation routine that creates a handle referencing the matrix (in Fortran this handle is just an `integer` variable). Afterwards, the user can call the necessary Sparse BLAS routines with the handle as a means to reference the data. The implementation chooses the data-dependent algorithms internally, without the user being involved. When the matrix is no longer

needed the matrix handle can be released and a cleanup routine is called to free any internal storage resources associated with that handle. In the following sections, we describe each step of the above procedure and its implementation in Fortran 95. We describe the functionalities in Section 2. The main part of the paper is Section 3 where we discuss how we organize, create, and use the data structures in Fortran for the sparse data. In Sections 4 and 5, we discuss the Fortran interface for the Level 1 and higher level BLAS respectively and consider the release of the matrix handles in Section 6. We make some general comments on our Fortran 95 interface in Section 7 and illustrate our implementation with a sample program in Section 8. Finally we discuss the availability of our software in Section 9. For details concerning the implementation of interfaces in C, we refer to [12].

2 The Sparse BLAS functionalities

In this section, we briefly review the functionalities provided by the Sparse BLAS so that we can reference them in later sections where they are described in more detail.

2.1 Level 1 Sparse BLAS functionalities

In Table 2.1, we list the operations belonging to the Level 1 Sparse BLAS. The following notation is used: r and α are scalars, x is a compressed sparse vector, y is a dense vector, and $y|_x$ refers to the entries of y that have the same indices as the stored nonzero components of the sparse vector x .

USDOT	sparse dot product	$r \leftarrow x^T y,$ $r \leftarrow x^H y$
USAXPY	sparse vector update	$y \leftarrow \alpha x + y$
USGA	sparse gather	$x \leftarrow y _x$
USGZ	sparse gather and zero	$x \leftarrow y _x; y _x \leftarrow 0$
USSC	sparse scatter	$y _x \leftarrow x$

Table 2.1: Level 1 Sparse BLAS: sparse vector operations.

2.2 Level 2 Sparse BLAS functionalities

Table 2.2 lists the Level 2 operations on sparse matrices and a dense vector. Here, A represents a general sparse matrix and T a sparse triangular matrix, x and y are dense vectors, and α is a scalar.

USMV	sparse matrix-vector multiply	$y \leftarrow \alpha Ax + y$ $y \leftarrow \alpha A^T x + y$ $y \leftarrow \alpha A^H x + y$
USSV	sparse triangular solve	$x \leftarrow \alpha T^{-1} x$ $x \leftarrow \alpha T^{-T} x$ $x \leftarrow \alpha T^{-H} x$

Table 2.2: Level 2 Sparse BLAS: sparse matrix-vector operations.

2.3 Level 3 Sparse BLAS functionalities

The Level 3 operations are presented in Table 2.3. As before, A denotes a general sparse matrix and T a sparse triangular matrix. B and C are dense matrices, and α is a scalar,

USMM	sparse matrix-matrix multiply	$C \leftarrow \alpha AB + C$ $C \leftarrow \alpha A^T B + C$ $C \leftarrow \alpha A^H B + C$
USSM	sparse triangular solve	$B \leftarrow \alpha T^{-1} B$ $B \leftarrow \alpha T^{-T} B$ $B \leftarrow \alpha T^{-H} B$

Table 2.3: Level 3 Sparse BLAS: sparse matrix-matrix operations.

2.4 Routines for the creation of sparse matrices

The routines for the creation of a sparse matrix and its associated handle are listed in Table 2.4. The Sparse BLAS can deal with general sparse matrices and with sparse block matrices with a fixed or variable block size. After the creation of the corresponding handle, the entries must be input using the appropriate insertion routines. The construction is finished by calling `USCR_END`.

Furthermore, we can specify various properties of the matrix in order to assist possible optimization of storage and computation. This is done by calls to `USSP`; possible parameters are listed in Tables 2.5 and 2.6. Calls to `USSP` should be made after a call to the `BEGIN` routine but before the first call to an `INSERT` routine for the same handle. In order to obtain information on the properties of a sparse matrix, the routine `USGP` can be called with parameters as given in Tables 2.6 and 2.7.

3 Handle creation and matrix data initialisation

In this section, we discuss the internal data structures and manipulation related to the creation of a matrix handle that will be used to represent the sparse matrix

USCR_BEGIN	begin point-entry construction
USCR_BLOCK_BEGIN	begin block-entry construction
USCR_VARIABLE_BLOCK_BEGIN	begin variable block-entry construction
USCR_INSERT_ENTRY	add point-entry
USCR_INSERT_ENTRIES	add list of point-entries
USCR_INSERT_COL	add a compressed column
USCR_INSERT_ROW	add a compressed row
USCR_INSERT_CLIQUE	add a dense matrix clique
USCR_INSERT_BLOCK	add a block entry
USCR_END	end construction
USSP	set matrix property
USGP	get/test for matrix property
USDS	release matrix handle

Table 2.4: Sparse BLAS: operations for the handling of sparse matrices.

in the later calls to the Sparse BLAS. From the implementor’s point of view, the choice of the internal data structures is perhaps the most important part of the implementation as it will influence the design, implementation and performance of all subsequent operations.

Conceptually, the Sparse BLAS distinguishes between three different types of sparse matrices; (1) ordinary sparse matrices, (2) sparse matrices with a regular block structure, and (3) sparse matrices with a variable block structure. This distinction will allow a vendor to provide optimised algorithms for blocked data.

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 \\ 0 & 0 & 23 & 24 & 0 \\ 31 & 32 & 33 & 34 & 0 \\ 0 & 42 & 0 & 44 & 0 \\ 51 & 52 & 0 & 0 & 55 \end{pmatrix} \quad (1)$$

$$B = \left(\begin{array}{cc|cc|cc} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ \hline 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \\ \hline 51 & 52 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 \end{array} \right), \quad (2)$$

blas_unit_diag	diagonal entries not stored, assumed to be 1.0
blas_non_unit_diag	diagonal entries stored (default)
blas_no_repeated_indices	input indices are unique (default)
blas_repeated_indices	input indices can be repeated
blas_lower_symmetric	only lower half of symmetric matrix is stored
blas_upper_symmetric	only upper half of symmetric matrix is stored
blas_lower_hermitian	only lower half of Hermitian matrix is stored
blas_upper_hermitian	only upper half of Hermitian matrix is stored
blas_irregular	unstructured matrix
blas_regular	structured matrix
blas_block_irregular	unstructured block matrix
blas_block_regular	structured block matrix
blas_unassembled	unassembled elemental matrix

Table 2.5: Matrix properties (can be set by USSP)

blas_zero_base	indices are 0-based
blas_one_base	indices are 1-based (default for Fortran)
blas_lower_triangular	sparse matrix is lower triangular
blas_upper_triangular	sparse matrix is upper triangular
blas_rowmajor	(block only) dense block stored row major order
blas_colmajor	(block only) dense block stored col major order (default for Fortran)

Table 2.6: Matrix properties (can be set by USSP / read by USGP)

blas_new_handle	handle created but no entries inserted so far
blas_open_handle	already an entry inserted, creation not finished
blas_valid_handle	handle refers to matrix with completed creation
blas_invalid_handle	handle not currently in use
blas_general	sparse matrix is nonsymmetric
blas_symmetric	sparse matrix is symmetric
blas_hermitian	(complex) sparse matrix is Hermitian
blas_complex	matrix values are <code>complex</code>
blas_real	matrix values are <code>real</code>
blas_integer	matrix values are <code>integer</code>
blas_double_precision	matrix values are <code>double precision</code>
blas_single_precision	matrix values are <code>single precision</code>
blas_num_rows	the number of rows of the matrix
blas_num_cols	the number of columns of the matrix
blas_num_nonzeros	the number of nonzero entries of the matrix

Table 2.7: Matrix properties (can be read by USGP)

$$C = \left(\begin{array}{cc|ccc|c|ccc|cc} 4 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 \\ 1 & 5 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\ \hline 0 & 0 & 6 & 1 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 9 & 3 & 0 & 0 & 0 & 0 & 0 \\ \hline 2 & 1 & 3 & 4 & 5 & 10 & 4 & 3 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 13 & 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 3 & 11 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 7 & 0 & 0 \\ \hline 8 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 & 3 \\ -2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 12 \end{array} \right). \quad (3)$$

The actual creation of a sparse matrix with its handle consists of three steps and involves the routines listed in Table 2.4.

1. An internal data structure is initialized by calling one of the `USCR_BEGIN` routines.
2. The matrix data is passed to the internal data structure by one or more calls to `USCR_INSERT` routines.
3. The construction is completed by calling the `USCR_END` routine.

This last step of the creation procedure needs more elaboration. Intentionally, the user need not know how many matrix entries will be input but can simply pass the data to the Sparse BLAS using the insert routines. Consequently, the Sparse BLAS must use dynamic memory allocation and a dynamic data structure for the construction phase of the matrix. We use linked lists which are augmented dynamically when new matrix entries are added. The nodes of the lists contain the matrix entries together with their indices and a pointer to the next list node. In order to limit the size of the list, we keep the matrix entries grouped together in the same way and in the same order as the user passes them to the Sparse BLAS. If a single matrix entry is inserted, the list node contains only this single entry; if a row, a column, or a block is inserted, a different list is used and the list node contains all entries of the row, the column, or the block, respectively. In order to identify which kind of data is associated with each node, we use different pointers for row, column, and block data, respectively. However, using this structure for the Level 2 and

Level 3 algorithms would imply a serious performance loss because of the amount of indirect addressing involved. At the call to `USCR_END`, all the data for the matrix is known. We can now allocate one contiguous block of memory with appropriate size and copy the data into a static data structure for better performance. Additionally, it is possible to sort the matrix entries during this copying process so that the matrix data is held by rows or columns if this is beneficial for the performance of the Level 2 and Level 3 algorithms.

Before we describe the layout of the static internal storage schemes, we discuss alternative approaches to handling the dynamic memory allocation. Generally, it is important to limit memory fragmentation by allocating storage by blocks. We respect this by allocating the space for matrix rows, columns or blocks ‘in one shot’. Another possibility is to preallocate a certain amount of memory and add the matrix entries as long as space is available; more space can be allocated when it is needed. This preallocation allows us to allocate contiguous memory blocks independently from the way a user inserts the matrix data. However, it is not possible *a priori* to predict how much memory should be allocated.

We now present the details of our internal storage schemes:

1. Ordinary sparse matrices consisting of point entries are stored in coordinate format (COO), that is the entries of the matrix are stored along with their corresponding row and column indices. This requires the three arrays:
 - *VAL* - a real or complex array containing the entries of A , in any order.
 - *INDX* - an integer array containing the corresponding row indices of the entries of A .
 - *JNDX* - an integer array containing the corresponding column indices of the entries of A .

For example, a representation of the matrix A in equation (1) in COO format could be:

$$\begin{aligned} VAL &= (11 \ 51 \ 31 \ 32 \ 34 \ 52 \ 13 \ 23 \ 33 \ 14 \ 24 \ 42 \ 55 \ 44 \), \\ INDX &= (1 \ 5 \ 3 \ 3 \ 3 \ 5 \ 1 \ 2 \ 3 \ 1 \ 2 \ 4 \ 5 \ 4 \), \\ JNDX &= (1 \ 1 \ 1 \ 2 \ 4 \ 2 \ 3 \ 3 \ 3 \ 4 \ 4 \ 2 \ 5 \ 4 \). \end{aligned}$$

2. Systems with a regular block structure, where each entry is an LB -by- LB dense block, are stored internally in block coordinate format (BCO). Systems of this form typically arise, for example, when there are multiple unknowns per grid point of a discretized partial differential equation. Typically LB is a small number, less than twenty, determined by the number of quantities measured at each grid point, for example velocity, pressure, temperature, etc. The BCO format is defined similarly to the COO format. Entries are stored block-wise together with their block row and block column indices. This again requires three arrays.

- *VAL* - a real or complex array containing the entries of the matrix, grouped and stored as dense blocks.
- *BINDX* - an integer array containing the block row indices.
- *BJNDX* - an integer array containing the block column indices.

For example, a representation of the matrix B in equation (2) in BCO format could be:

$$\begin{aligned} VAL &= (\begin{array}{cccccccccc} 11, & 21, & 12, & 22, & 15, & 25, & 16, & 26, & 33, & 43, \\ & & 0, & 44, & 35, & 45, & 36, & 46, & 51, & 61, & 52, & 62 \end{array}), \\ BINDX &= (\begin{array}{cccccc} 1, & 1, & 2, & 2, & 3 & \end{array}), \\ BJNDX &= (\begin{array}{ccccc} 1, & 3, & 2, & 3, & 1 \end{array}). \end{aligned}$$

Note that we choose the block internal storage to be in ‘Fortran style’, that is in column major order.

3. Systems with an irregular block structure are stored internally in the Variable Block Row format (VBR). VBR stores the nonzero block entries in each of the block rows as a sparse vector of dense matrices. The matrix is *not* assumed to have uniform block partitioning, that is, the blocks may vary in size. The VBR data structure is defined as follows. Consider an m -by- k sparse matrix along with a row partition $P_r = \{i_1, i_2, \dots, i_{m_b+1}\}$ and column partition $P_c = \{j_1, j_2, \dots, j_{k_b+1}\}$ such that $i_1 = j_1 = 1$, $i_{m_b+1} = m + 1$, $j_{k_b+1} = k + 1$, $i_p < i_{p+1}$ for $p = 1, \dots, m_b$, and $j_q < j_{q+1}$ for $q = 1, \dots, k_b$. The matrix C in equation (3) is an example of a block matrix where the blocks C_{ij} are defined according to the row and column partition shown. The block entries are stored block row by block row and each block entry is stored as a dense matrix in standard column major form. Six arrays are associated with this form of storage.

- *VAL* - a real or complex array containing the block entries of C . Each block entry is a dense rectangular matrix stored column by column.
- *INDPTR* - an integer array, the i -th entry of *INDPTR* points to the location in *VAL* of the (1,1) entry of the i -th block entry.
- *BINDX* - An integer array containing the block column indices of the nonzero blocks of C .
- *RPNTR* - An integer array of length $m_b + 1$ containing the row partition of C . *RPNTR*(i) is set to the row index of the first row in the i -th block row.
- *CPNTR* - An integer array of length $k_b + 1$ containing the column partition of C . *CPNTR*(j) is set to the column index of the first column in the j -th block column.

- *BPNTR* - An integer array of length m_b such that $BPNTRB(i)$ points to the location in *BINDX* of the first nonzero block entry of block row i . If the i -th block row contains only zeros then set $BPNTRB(i + 1) = BPNTRB(i)$.

For example, the matrix C in equation (3) is stored in VBR format as follows:

$$\begin{aligned}
 VAL &= (4, 1, 2, 5, 1, 2, -1, 0, 1, -1, 6, 2, \\
 &\quad -1, 1, 7, 2, 2, 1, 9, 2, 0, 3, 2, 1, \\
 &\quad 3, 4, 5, 10, 4, 3, 2, 4, 3, 0, 13, 3, \\
 &\quad 2, 4, 11, 0, 2, 3, 7, 8, -2, 4, 3, \\
 &\quad 25, 8, 3, 12 \quad), \\
 INDPTR &= (1, 5, 7, 11, 20, 23, 25, 28, 29, 32, 35, 44, \\
 &\quad 48, 52 \quad), \\
 BINDX &= (1, 3, 5, 2, 3, 1, 2, 3, 4, 3, 4, 1, \\
 &\quad 5 \quad), \\
 RPNTR &= (1, 3, 6, 7, 10, 12 \quad), \\
 CPNTR &= (1, 3, 6, 7, 10, 12 \quad), \\
 BPNTR &= (1, 4, 6, 10, 12, 15 \quad),
 \end{aligned}$$

We emphasize that our choice of the internal storage schemes is only one among several possibilities. The coordinate representation is very simple and is, for example, used as the basis for the *Matrix Market* sparse matrix storage format. However, the drawback of both the COO and the BCO storage format consists of the fact that the matrix entries are not necessarily ordered, which can degrade the efficiency of the Level 2 and Level 3 algorithms. Alternative matrix formats include the storage of the matrix entries by compressed columns or rows. Specifically, the Compressed Sparse Column (CSC) storage scheme is used for the matrices of the *Harwell-Boeing* collection [5] and forms also the basis of the *Rutherford-Boeing* format [6]. It is up to the vendor to choose the most appropriate representation.

We present now the fundamental datatype that accommodates all the data belonging to a sparse matrix. Its design is derived from [8]. When `USCR_END` is called, an object of this type is created that will then be referenced by its handle in the calls to the Level 2 and Level 3 routines.

TYPE DSPMAT

INTEGER :: M,K

CHARACTER*5 :: FIDA

CHARACTER*11 :: DESCRA

INTEGER, DIMENSION(10) :: INFOA

DOUBLE PRECISION, POINTER, DIMENSION(:) :: VALUES

INTEGER, POINTER, DIMENSION(:) :: IA1,IA2,PB,PE,BP1,BP2

END TYPE DSPMAT

(This is the datatype for a matrix with real entries in `double precision`. The other datatype formats are analogous.) Since the meaning of most of the components is already obvious from the above discussion of the internal storage formats, we give only short general remarks on them.

- The integers M and K represent the dimensions of the sparse matrix.
- FIDA holds a string representation of the matrix format, for example, ‘COO’.
- DESCRA stores possible matrix properties such as symmetry that appear in the Tables 2.5, 2.6, and 2.7.
- INFOA holds complementary information on the matrix such as the number of nonzero entries.
- The array `VALUES` keeps the values of the matrix entries. The way in which these entries are stored can be deduced from the following character and integer arrays.
- The arrays `IA1, IA2, PB, PE, BP1, BP2` are used to provide the necessary information on the sparsity structure of the matrix. The pointer arrays `PB, PE, BP1, BP2` are only used for block matrices. Note that we use generic array names, since their use depends on the matrix format. For example, in COO format, the arrays `IA1` and `IA2` represent *INDX* and *JNDX*, while in VBR format, they represent *BINDX* and *INDPTR*.

We decided to group the administration of all handle-matrix pairs according to their floating-point data type, that is, we keep a separate list of all valid matrix handles for each of the five floating-point data types supported by the Sparse BLAS.

4 Level 1 subroutines

The storage of sparse vectors is much less complicated than that for sparse matrices and greatly facilitates the implementation of the Level 1 routines.

Generally, only the nonzero entries of a sparse vector x will be stored which leads to a representation of x by a pair of one-dimensional arrays, one for the entries and the other one for their indices. For example, the sparse vector

$$x = (1.0, 0, 3.0, 4.0, 0)^H$$

can be represented as

$$\begin{aligned} VAL &= (1.0, 3.0, 4.0) \\ INDX &= (1, 3, 4). \end{aligned} \tag{4}$$

For the implementation of the Level 1 Sparse BLAS functionalities as listed in Table 2.1, we generally do not assume that the entries of sparse vectors are ordered.

By ordering and grouping the sparse vector according to its indices, it can be ensured that the dense vector involved in the Level 1 operations is accessed in blocks and cache reuse is enhanced.

One peculiarity of the Level 1 routines, in contrast to the sparse matrix operations of Level 2 and Level 3, is that the sparse vector operations do not return an error flag. Level 2 and Level 3 routines have to provide some checking of the input arguments, for example matching matrix dimensions, and can detect at least some of these errors and signal them to the user by setting an error flag. Because of the simplicity, the representation of sparse vectors is left to the user who is thus responsible for ensuring the correctness of the data. Furthermore, the overhead for checking in the Level 2 and Level 3 operations is less important because of their greater granularity.

5 Level 2 and Level 3 subroutines

The discussion in Section 3 on the different internal storage schemes shows that a different Level 2 and Level 3 routine for the Sparse BLAS must be implemented for each scheme. Hidden from the user who uses the matrix handle in a generic subroutine call, the software chooses the appropriate routine according to the type of data. We discuss these issues in the following sections.

5.1 Product of a sparse matrix with one or many dense vectors

In this section, we discuss the implementation of the multiplication of a sparse matrix with a dense vector or a dense matrix. We concentrate on the matrix-vector multiplication, since in our code, the multiplication of a sparse matrix and a dense matrix is performed columnwise. Thus, we discuss the realization of

$$y \leftarrow \alpha Ax + y,$$

and

$$y \leftarrow \alpha A^T x + y.$$

Our implementation uses the generic functions of Fortran 95 extensively. We provide the following interface for the support of the different types

```
interface usmv
  module procedure susmv
  module procedure dusmv
  module procedure cusmv
  module procedure zusmv
  module procedure iusmv
end interface
```

with an implementation for matrices in double precision as follows:

```
subroutine dusmv(a, x, y, ierr, transa, alpha)
integer, intent(in) :: a
double precision, dimension(:), intent(in) :: x
double precision, dimension(:), intent(inout) :: y
integer, intent(out) :: ierr
integer, intent(in), optional :: transa
double precision, intent(in), optional :: alpha
```

where

- a denotes the matrix handle.
- x, y denote the dense vectors.
- $ierr$ is used as an error flag.
- $transa$ allows optionally the use of the transposed sparse matrix.
- $alpha$ is an optional scalar factor.

In the case of 'COO' storage, we perform the multiplication entry by entry, whereas for both regular block matrices in 'BCO' and irregular block matrices in 'VBR' format, we perform a dense matrix-vector multiplication with the subblocks.

We remark that, even if we perform the multiplication of a sparse matrix and a dense matrix column by column, there can be more efficient ways of doing this. Depending on the size of the dense matrix, a vendor could use blocking also on the dense matrix to gain performance.

5.2 Solution of a sparse triangular system with one or many dense right-hand sides

In this section, we show the implementation of the solution of a sparse triangular system with a dense vector or a dense matrix. As in Section 5.1, we focus on the case of a single right-hand side:

$$\begin{aligned}x &\leftarrow \alpha T^{-1}x, \\ &\text{and} \\x &\leftarrow \alpha T^{-T}x.\end{aligned}$$

Similarly to the multiplication routines, we provide a generic interface for the support of the different types in a similar way to that discussed for `usmv` in the previous section. The implementation of the different floating-point data types, for example `dussv`, is given by the following header:

```

subroutine dussv(a,x,ierr,transa,alpha)
integer, intent(in) :: a
double precision, intent(inout) :: x(:)
integer, intent(out) :: ierr
integer, intent(in), optional :: transa
double precision, intent(in), optional :: alpha

```

where

- a denotes the matrix handle.
- x denotes the dense vector.
- $ierr$ is used as an error flag.
- $transa$ allows optionally the use of the transposed sparse matrix.
- $alpha$ is an optional scalar factor.

For block matrices in either 'BCO' or 'VBR' format, the triangular solve is blocked and uses dense matrix kernels for matrix-vector multiplication and triangular solves on the subblocks.

In the case of a simultaneous triangular solve for more than one right-hand side, we have chosen internally to perform the solve separately on each of them. However, the remark given at the end of Section 5.1 applies here, too. Blocking should be applied to the right-hand side if the matrix is big enough and offers enough potential for efficient dense matrix kernels.

6 Releasing matrix handles

Since the matrix handles are created dynamically, we have to be careful with our memory management. In particular, we will want to return allocated memory to the system when we do not need it any more. The Sparse BLAS provides a routine for releasing a created matrix handle and freeing all associated memory.

The Fortran 95 binding of the handle release routine is:

```

subroutine usds(a,ierr)
integer, intent(in) :: a
integer, intent(out) :: ierr

```

Here, a denotes the matrix handle to be released and $ierr$ a variable to signal possible internal errors occurring on the attempt to release the handle.

We have already remarked in Section 3 that we use different internal data structures for the handle initialisation and the Level 2 and Level 3 routines. The linked lists used in the handle initialisation procedure can already be deallocated

when `USCR_END` is called. The call to `USDS` will then result in a deallocation of the memory associated with the fundamental internal datatype shown at the end of Section 3.

When a matrix handle is released, one can either re-use the handle, that is, its `integer` value, for the next matrix that will be created, or prevent it from being used again. We decided to assign a new handle to each created matrix and not to re-use the released handles. This ensures that matrices are not confused by accident, as no matrix handle can represent more than one matrix simultaneously in the context of the program.

7 Some remarks on using Fortran 95

Fortran is widely recognized as very suitable for numerical computation. Fortran 95 is fully compatible with Fortran 77 but also includes some useful features of other modern programming languages.

In our implementation, we benefit from the following features of Fortran 95:

1. Modules allow the code to be structured by grouping together related data and algorithms. An example is given by the Sparse BLAS module itself as it is used by the test program described in Section 8.
2. Generic interfaces as shown in Sections 5.1 and 5.2 allow the *same* subroutine call to be used for each of the 5 floating-point data types supported.
3. The linked lists for matrix and matrix entry management, as described in Sections 3 and 6, depend on dynamic memory allocation.
4. The Level 2 and Level 3 algorithms for block matrices from Sections 5.1 and 5.2 use the new vector operation facilities instead of loops wherever possible.

8 A sample program

In this section, we give an example of how to use the Sparse BLAS and illustrate all steps from the creation of the matrix handle for the sample matrix T and the calls to Level 2 and Level 3 routines up to the release of the handle. It is worth mentioning that the whole Sparse BLAS is available as a Fortran 95 module which has to be included by the statement `use blas_sparse` as shown in the fourth line of the source code. This module contains all Sparse BLAS routines and predefined named constants like the matrix properties defined in the Tables 2.5, 2.6, and 2.7.

```

program test
!
!----- Use the Sparse BLAS module -----
use blas_sparse
!
!----- The test matrix data -----
!
!   / 1  1  1  1  1\
!   |   1  1  1  1|
! T= |       1  1  1|
!   |           1  |
!   \               1/
!
double precision,dimension(14):: T_VAL=1.
integer,dimension(14):: T_indx=(/1,1,2,1,2,3,1,2,3,4,1,2,3,5/)
integer,dimension(14):: T_jndx=(/1,2,2,3,3,3,4,4,4,4,5,5,5,5/)
integer,parameter:: T_m=5, T_n=5, T_nz=14
double precision:: Tx(5) =(/15.,14.,12.,4.,5./)
!
!----- Declaration of variables -----
double precision,dimension(:),allocatable:: x, y, z
double precision,dimension(:,:),allocatable:: dense_B,dense_C,dense_D
integer:: i,prpty,a,ierr
!
      prpty = blas_upper_triangular + blas_one_base
      allocate(x(5), y(5))
      y=0.
      open(UNIT=5,FILE='output',STATUS='new')
!
!----- Begin point entry construction -----
      call duscr_begin(T_m, T_n, a, istat)
!
!----- Insert all entries -----
      call uscr_insert_entries(a, T_VAL, T_indx, T_jndx, istat)
!
!----- Set matrix properties -----
      call ussp(a, prpty,istat)
!
!----- End of construction -----
      call uscr_end(a, istat)
!
      do i=1, size(x)

```

```

        x(i) = dble(i)
    end do
    allocate(z(size(x)))
    z = x
    allocate(dense_B(size(y),3),dense_C(size(x),3),&
            dense_D(size(x),3))
    do i = 1, 3
        dense_B(:, i) = x
        dense_C(:, i) = 0.
        dense_D(:, i) = Tx
    end do

!
!----- Matrix-Vector product -----
    write(UNIT=5, FMT='(A)') '* Test of MV multiply *'
    call usmv(a, x, y, istat)
    write(UNIT=5, FMT='(A)') 'Error : '
    write(UNIT=5, FMT='(D12.5)') maxval(abs(y-Tx))
!
!----- Matrix-Matrix product -----
    write(UNIT=5, FMT='(A)') '* Test of MM multiply *'
    call usmm(a, dense_B, dense_C, istat)
    write(UNIT=5, FMT='(A)') 'Error: '
    write(UNIT=5, FMT='(D12.5)') maxval(abs(dense_C-dense_D))
!
!----- Triangular Vector solve -----
    write(UNIT=5, FMT='(A)') '* Test of tri. vec. solver *'
    call ussv(a, y, istat)
    write(UNIT=5, FMT='(A)') 'Error : '
    write(UNIT=5, FMT='(D12.5)') maxval(abs(y-x))
!
!----- Triangular Matrix solve -----
    write(UNIT=5, FMT='(A)') '* Test of tri. mat. solver *'
    call ussm(a, dense_C, istat)
    write(UNIT=5, FMT='(A)') 'Error : '
    write(UNIT=5, FMT='(D12.5)') maxval(abs(dense_C-dense_B))
!
!----- Deallocation -----
    deallocate(x,y,z,dense_B,dense_C,dense_D)
    call usds(a,istat)
    close(UNIT=5)
!
end program test

```

9 Software availability and final remarks

9.1 Download location

The software that we have described in this report conforms with the Sparse BLAS standard defined by the BLAS Technical forum [1, 7]. The code is available from <http://www.cerfacs.fr/~voemel/SparseBLAS/SparseBLAS.html>.

9.2 Code generation

The Sparse BLAS provides routines for integer computations and **real** and **complex** variables in single and double precision. Because of their similarity, and also to allow easier maintenance of the code, the different instantiations of a routine are generated from a single source by shell preprocessing. Further instructions on the code generation are enclosed with the software.

9.3 Final remarks

Aspects of our Fortran 95 implementation have also been described in the two Technical Reports [9] and [10].

Our software is a reference implementation. Vendors may supply optimized versions which exploit special features of high performance computing architectures. Suggestions for such optimizations have been given in the relevant sections of this paper.

Acknowledgments

We would like to thank Jennifer Scott and John Reid for their comments on an earlier version of this paper.

References

- [1] BLAS Technical Forum Standard. *The International Journal of High Performance Computing Applications*, 15(3–4), 2001.
- [2] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 17:253–263, 1991.
- [3] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

- [4] J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–17, 1988.
- [5] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.
- [6] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997.
- [7] I. S. Duff, M. Heroux, and R. Pozo. The Sparse BLAS. *ACM Trans. Math. Software*, this volume, 2001.
- [8] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 Basic Linear Algebra Subprograms for sparse matrices: a user level interface. *ACM Trans. Math. Software*, 23(3):379–401, 1997.
- [9] I. S. Duff and C. Vömel. Level 2 and Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: A Fortran 95 instantiation. Technical Report TR/PA/00/18, CERFACS, Toulouse, France, 2000.
- [10] I. S. Duff, C. Vömel, and M. Youan. Implementing the Sparse BLAS in Fortran 95. Technical Report TR/PA/00/82, CERFACS, Toulouse, France, 2000.
- [11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.
- [12] K. A. Remington and R. Pozo. NIST Sparse BLAS user’s guide. Internal Report NISTIR 6744, National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2001.