D R A F T

Document for the Basic Linear Algebra Subprograms (BLAS) Standard

Basic Linear Algebra Subprograms Technical (BLAST) Forum

August 11, 1997

Contents

The discussion in each chapter is not limited only to the points listed, any related topics can be included at the discretion of the chapter authors. Final acceptance of each section will be decided at the next meeting using *Robert's Rules*. See BLAST Forum homepage for more detailed rules.

Chapter 1

Sparse BLAS

1.1 Overview

This chapter describes a proposal for the Sparse Basic Linear Algebra Subprograms (BLAS). A three layer design is described, consisting of (1) a high-level, data-neutral "*User*" interface, which specifies one routine name per mathematical operation, (2) a mid-level "*Toolkit*" layer, with one routine name for each operation/data-structure combination, and (3) low-level "*development*" layer, whose design and implementation is up to the discretion of the implementor. The current document focuses on the "Toolkit" interfaces, with brief descriptions of two reference implementations as a reference to library writers.

The functionality proposed for the Sparse BLAS comprises the following common operations: (i) sparse matrix times dense matrix product, (ii) the solution of a sparse triangular system with multiple right-hand-sides, (iii) the right permutation of a sparse matrix, and (iv) a check for the integrity of a sparse matrix representation. The interfaces for these four operations are defined for a variety of common data structures and a set of guidelines is given to define interfaces for new data structures. The proposal focuses on Level 2 (matrix/vector) and Level 3 (matrix/matrix) operations, since these are common and the most likely to benefit from optimized implementations. A Level 1 Sparse BLAS interface has been proposed by Dodson, Grimes, and Lewis [3], and we leave it for discussion which if these should be included in the final proposal.

The current goals of this proposal are to (i) define an initial set of data-structure-dependent kernel routines, and (ii) to establish a framework for extending this initial set as new data structures and operations are added. The purpose of the Sparse BLAS Toolkit interface and kernel routines is to provide the primary foundation for higher-level application libraries, for example the User Level Sparse BLAS described in [5].

In Section 1.3 we define the basic operations, independent of data structures. We also discuss our naming and argument conventions. Thus, this section defines the interfaces for the current set of routines in the Sparse BLAS and also provides a framework for future extensions.

In Section 1.4 we define the supported data structures in detail. The precise definition of data structures is very important for this proposal since many varieties of the basic data structures exist. In effect, this proposal is attempting to standardize the definition of these data structures.

In Section 1.5, we present some issues which are open for further discussion. Among these are the relevance of a development interface layer, what functionality to support from BLAST specification, using structure and pointer based arguments, and C/C++ binding issues.

Section 1.6 discusses practical issues regarding the implementation and performance of the Sparse BLAS. One current kernel implementation results in over 10,000 routines, generated automatically from C source code, while an alternate kernel implementation relies on standard Fortran library design. Each implementation style has its strengths and weaknesses, which we will attempt to categorize for the benefit of future implementors. Although the kernel implementation is not a formal part of the specification, these issues are directly related to adoptability of the Sparse BLAS in practice.

1.2 Sparse BLAS Functionality

1.2.1 Operations

The present scope of the Sparse BLAS covers matrix-matrix product operations, the solution of triangular systems with multiple right-hand-sides, permutations of sparse matrices, and the integrity check of a sparse matrix representation. More precisely we have

1. Matrix-matrix products:
$$C \leftarrow \alpha AB + \beta C$$
$$C \leftarrow \alpha A^T B + \beta C$$

2. Solution of triangular systems:
$$C \leftarrow \alpha DA^{-1} B + \beta C$$
$$C \leftarrow \alpha DA^{-T} B + \beta C$$
$$C \leftarrow \alpha A^{-1} DB + \beta C$$
$$C \leftarrow \alpha A^{-T} DB + \beta C$$

3. Right permutation of a sparse matrix in JAD format:
$$A \leftarrow AP$$
$$A \leftarrow AP^T$$

4. Integrity check of a sparse matrix representation:
$$(NERR) \leftarrow A$$

where α and β are scalars, B and C are rectangular matrices, D is a (block) diagonal matrix, and A is sparse matrix. The sparse matrix A can be stored in one of a variety of sparse data structures as described in Section 1.4. Each of the above operations is proposed for single and double precision, real and complex data, as appropriate.

1.2.2 Rationale for Functionality

Below we discuss our motivation for each basic operation we have included.

Matrix-Matrix Product and Solution of Sparse Triangular System

We believe there is little need to justify the first two operations which are supported in the Sparse BLAS Toolkit Ṡparse matrix-dense matrix product and the solution of a sparse triangular system are the primary computational kernels in many sparse linear equation and eigensystem solvers. *Can/should we say something here to justify the compound operations, and the presence of the left and right scaling matrices in the triangular solves?*

Right Permutation of a Sparse Matrix in JAD Format

The right permutation of a sparse matrix is necessary in cases where the rows of A were already permuted in the process of storing A in the sparse data structure. For example, if A is stored in JAD format, then the rows of A are ordered by decreasing number of entries. If the corresponding right permutation is not applied, then the result matrix, C, from a call to DJADMM will not be ordered consistently with the input matrix B. By permuting the columns of A and the rows of B *outside* the iteration loop, we can eliminate this inconsistency. For example, consider the simple iteration loop:

do i = 1, ...
  $y \leftarrow Ax$
  $x \leftarrow x + ay$
end do

If the matrix H is such that $A = PH$ for some permutation matrix P then, in terms of the matrix H, the loop becomes:

do i = 1, ...
  $y \leftarrow Hx$
  $y \leftarrow Py$
  $x \leftarrow x + ay$
end do

The additional permutation Py in each iteration could significantly affect efficiency. By performing an explicit right permutation, we can eliminate permutations from the iteration loop as follows:

$x \leftarrow P^T x$
$H \leftarrow HP$
do i = 1, ...
  $y \leftarrow Hx$
  $x \leftarrow x + ay$
end do
$x \leftarrow Px$

Check for Validity of Sparse Matrix Representation

Because of the potential large size of sparse problems and the complexity of sparse matrix data structures, it is very useful to have the capability to perform some kind of check on the sparse matrix representation. The checking routines presented here make no rigorous attempt to determine the numerical stability and, thus, are not fail-safe. However, they do improve the probability of finding and correcting most programming errors. As the use of object-oriented programming techniques increases, the need for these routines should diminish, since their functionality would be included in the sparse matrix encapsulation.

GDMM(D, A, B, C) Algorithm.

Setup Phase (Performed only once)
Do **Parallel All** $D_i : i = 0, \ldots, \#PEs$
Distribute A_i B_i^l and C_i.
Transform A_i into A_i^l and A_i^r.
End Parallel Do

Compute Phase
Do **Parallel All** $D_i : i = 0, \ldots, \#PEs$
Obtain B_i^r from remote processors.
NO BARRIER
Compute $C_i = \beta C_i + \alpha A_i^l B_i^l$.
BARRIER
Compute $C_i = C_i + \alpha A_i^r B_i^r$
End Parallel Do

Figure 1.1: General Distributed Sparse Matrix-Dense Matrix Product Algorithm

1.2.3 Use on a Distributed Memory Machine

Clearly the proposal presented here is intended for a shared memory programming environment. However, one can easily use these kernels on distributed memory machines in a natural way. Consider the MM operation and partition A, B, and C so that each processor owns the same subset of rows of each array. Let A_i, B_i^l, and C_i denote the portion of A, B, and C, respectively, on the i^{th} processor. Let D_i denote the set of row indices that reside on processor i. Let B_i^r denote elements of B which are not on processor i but are reached to by A_i. Finally, further decompose each A_i into A_i^l, the portion of A_i that reaches to B_i^l, and A_i^r, the portion of A_i that reaches to B_i^r. Then a general distributed MM algorithm is as shown in Figure 1.1. Note that both computation steps can be performed using the MM kernels presented in this proposal. These kernels are also very useful for domain decomposition algorithms where all work in the inner iterations is done locally.

1.2.4 Targeted Data Structures

The following sparse matrix formats are identified for Sparse BLAS support. This list is open for discussion; see Section 1.4 for details.

- Point entry:
  - COO - Coordinate
  - CSC - Compressed sparse column
  - CSR - Compressed sparse row
  - DIA - Sparse diagonal
  - ELL - Ellpack/Itpack
  - JAD - Jagged diagonal
  - SKY - Skyline

- Block entry:
  - BCO - Block coordinate
  - BSC - Block compressed sparse column
  - BSR - Block compressed sparse row
  - BDI - Block sparse diagonal
  - BEL - Block Ellpack/Itpack
  - VBR - Variable block compressed sparse row

Each of the data structures we propose is intended to support an important class of problems or algorithms. Below we list each point entry data structure and one or more of its intended uses. The intended uses for block entry data structures are analogous with added property that we exploit the property of multiple unknowns per gridpoint or related properties as mentioned in Section 1.4.4.

- COO - Coordinate: Most flexible data structure when constructing or modifying a sparse matrix.
- CSC - Compressed sparse column: Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
- CSR - Compressed sparse row: Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
- DIA - Sparse diagonal: Particularly useful for matrices coming from finite difference approximations to partial differential equations on uniform grids.
- ELL - Ellpack/Itpack: Appropriate for finite element or finite volume approximations to partial differential equations where elements are of the same type, but the gridding is irregular.
- JAD - Jagged diagonal: Appropriate for matrices which are highly irregular or for a general-purpose matrix multiplication where the properties of the matrix are not known *a priori*.
- SKY - Skyline: Particularly well suited for Cholesky or LU decomposition when no pivoting is required. In this case, all fill will occur within the existing non-zero structure.

1.3 Sparse BLAS Level 2 and Level 3 Toolkit Level Interface

1.3.1 Naming Conventions for the Sparse BLAS Toolkit Interface

Each Sparse BLAS Toolkit subroutine has a six-character name that is a function of the data type, the data structure of the sparse matrix and the type of operation. More precisely, each name is of the form XYYYZZ where

- X indicates the data type:
  - S - Single-precision real
  - C - Single-precision complex
  - D - Double-precision real
  - Z - Double-precision complex

- YYY indicates the data structure of the sparse matrix: (Note: This list may change. See Section 1.4.)
  - Point entry:
    - COO - Coordinate
    - CSC - Compressed sparse column
    - CSR - Compressed sparse row
    - DIA - Sparse diagonal
    - ELL - Ellpack/Itpack
    - JAD - Jagged diagonal
    - SKY - Skyline
  - Block entry:
    - BCO - Block coordinate
    - BSC - Block compressed sparse column
    - BSR - Block compressed sparse row
    - BDI - Block sparse diagonal
    - BEL - Block Ellpack/Itpack
    - VBR - Variable block compressed sparse row
- ZZ indicates the type of operation:
  - MM - Matrix-matrix product
  - MV - Matrix-vector product (Level 2)
  - SM - Solution of a triangular system with multiple right-hand-sides
  - SV - Solution of a single triangular system (Level 2)
  - RP - Right permutation of a sparse matrix (presently for JAD data structure only)
  - CK - Integrity check of a sparse matrix representation.

1.3.2 Argument Conventions for Sparse BLAS Toolkit Interface

The argument conventions follow the spirit of the dense BLAS. The general order of arguments is:

1. Arguments specifying options.
2. Arguments specifying problem dimensions.
3. Input scalar associated with input matrices.
4. Description of sparse input matrices (*args(A)*, described below).
5. Description of dense input matrices.
6. Input scalar associated with input-output matrix.
7. Description of input-output matrix.
8. Error processing information.
9. Workspace.

10. Length of workspace.

In particular, we have

- Point entry matrix-matrix product:

  XYYYMM(TRANSA, M, N, K, ALPHA, *args(A)*, B, LDB, BETA, C, LDC, WORK, LWORK)

- Block entry matrix-matrix product:

  XYYYMM(TRANSA, MB, N, KB, ALPHA, *args(A)*, B, BLDB, BETA, C, BLDC, WORK, LWORK)

- Point entry solution of triangular system:

  XYYYSM(TRANSA, M, N, UNITD, DV, ALPHA, *args(A)*, B, LDB, BETA, C, LDC, WORK, LWORK)

- Block entry solution of triangular system:

  XYYYSM(TRANSA, MB, N, UNITD, DV, ALPHA, *args(A)*, B, BLDB, BETA, C, BLDC, WORK, LWORK)

- Right permutation of a sparse matrix:

  XYYYRP(TRANSP, M, K, *args(A)*,[IPERM,] WORK, LWORK)

- Point entry integrity check of a sparse matrix representation:

  XYYYCK(TRANSA, M, K, *args(A)*, NERR, WORK, LWORK)

- Block entry integrity check of a sparse matrix representation:

  XYYYCK(TRANSA, MB, KB, *args(A)*, NERR, WORK, LWORK)

Common Arguments The argument TRANSA is an integer argument. The possible values are

- TRANSA - Indicates how to operate with the sparse matrix.
  - 0 - Operate with the matrix (No-transpose).
  - 1 - Operate with the transpose of the matrix.
  - 2 - Operate with the conjugate transpose of the matrix.

  2 is equivalent to 1 if the matrix is real.

The argument M is the number of rows in the matrix C, N is the number of columns in C, and K is the number of rows in B. M and K are the row and column dimensions of A, respectively, if TRANSA = 0 or the row and column dimensions of A^T if TRANSA = 1 or 2.

The argument MB is the number of block rows in the matrix C, and KB is the number of block rows in B. MB and KB are the block row and block column dimensions of A, respectively, if TRANSA = 0 or the block row and block column dimensions of A^T if TRANSA = 1 or 2.

Negative or zero dimensions are allowed. However, if M, N, K, MB, or KB is less than or equal to zero, then no operations are performed.

The arguments ALPHA and BETA are both scalar inputs of the same data type as the matrices. These operands correspond to α and β, respectively, as defined above.

The arguments B and C are rectangular arrays with first dimension LDB and LDC, respectively. If A is a constant block entry matrix, then BLDB and BLDC are the *block* first dimensions of B and C, respectively. This implies for a block entry dimension of LB that LDB = BLDB*LB and LDC = BLDC*LB.

The argument WORK is an array of the same data type as A and of length LWORK. It is used as scratch space for optimizing performance. The minimal value for LWORK is $\max(\mathrm{M}, \mathrm{N})$ for point entry matrices and $\max(\mathrm{MB} * \mathrm{LB} * \mathrm{LB}, \mathrm{N})$ in the block entry case. The optimal value is returned in WORK(1) and can be obtained by reading INT(WORK(1)).

SM Arguments The argument UNITD is an integer argument indicating whether or not the diagonal matrix D is unitary. The possible values for UNITD are

- 1 - Unitary diagonal matrix, i.e., D is the identity. In this case the argument DV is ignored.
- 2 - Scale on the left (row scaling).
- 3 - Scale on the right (column scaling).

The argument DV is an array containing the diagonal entries of the (block) diagonal matrix D.

RP Arguments A permutation P is represented by an integer vector IPERM such that $IPERM(i)$ is equal to the position of the only nonzero element in row i of P. For example, if

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

then IPERM = (3, 1, 2). Note that, since IPERM is part of *args(A)* for the JAD data structure, there is no need to repeat it as an argument. However, for future extensions, we indicate where IPERM would be in the argument list.

The argument TRANSP is an integer argument. The possible values are

- TRANSP - Indicates how to operate with the permutation matrix.
  - 0 - Operate with the matrix (No-transpose).
  - 1 - Operate with the transpose of the matrix.

CK Arguments The argument NERR returns the number of errors detected in integrity check of a sparse matrix representation. Errors are processed by an error handling routine which prints the error information to standard error. The following checks are performed on the sparse matrix representation:

- Valid arguments.
  - Descriptors have valid values.
  - Pointers have non-decreasing order.
  - Indices are within problem dimensions.
  - Floating-point values are valid, i.e., values represent finite floating-point number on the given architecture.

- Structural integrity.
  - Detects empty rows/columns or missing crucial entries, e.g., missing diagonal entry for a triangular matrix with non-unit diagonal.
  - Mismatch of descriptor and matrix graph, e.g., entries from the upper triangle are stored in for a matrix which is declared to be lower triangular.
- Numerical integrity.
  - Detects inappropriate zero values.
  - Detects extremely small values.
- Reducibility check. Checks to see if the matrix is structurally reducible, i.e., the matrix can be separated into two or more disjoint matrices. This step requires a workspace of size two times the minimal value for LWORK. If LWORK is too small, the reducibility check will not be performed.

Order of Arguments for *args(A)* The list of arguments describing the sparse matrix A (denoted by *args(A)*) is specific to each storage format. These arguments appear in a the same order for both level interfaces. The general order of *args(A)*is

1. Descriptor array.
2. Array containing the non-zero values (entries) of input matrix.
3. First dimension of values array (if shape of this array differs from index arrays).
4. Array(s) containing indices corresponding to the entries of input matrix. If more than one index array:
   (a) Arrays ordered in decreasing length.
   (b) Arrays of same length are ordered with row indices first.
5. Length or first dimension of value/index arrays.
6. Array(s) containing pointers. If more than one pointer array:
   (a) Arrays ordered in decreasing length.
   (b) Arrays of same length are ordered with row pointers first, column pointers next and diagonal pointers last.
7. Length or first dimension of pointer arrays.
8. Array(s) representing left and/or right permutations of A. If more than one permutation array order left permutation array first.
9. Block entry dimension.

The exact specification of *args(A)* for each supported data structure is in the following Section 1.4.

Data Structure	*args(A)*
COO	DESCRA, VAL, INDX, JNDX, NNZ
CSC	DESCRA, VAL, INDX, PNTRB, PNTRE
CSR	DESCRA, VAL, INDX, PNTRB, PNTRE
DIA	DESCRA, VAL, LDA, IDIAG, NDIAG
ELL	DESCRA, VAL, INDX, LDA, MAXNZ
JAD	DESCRA, VAL, INDX, PNTR, MAXNZ, IPERM
SKY	DESCRA, VAL, PNTR
BCO	DESCRA, VAL, BINDX, BJNDX, BNNZ, LB
BSC	DESCRA, VAL, BINDX, BPNTRB, BPNTRE, LB
BSR	DESCRA, VAL, BINDX, BPNTRB, BPNTRE, LB
BDI	DESCRA, VAL, BLDA, IBDIAG, NBDIAG, LB
BEL	DESCRA, VAL, BINDX, BLDA, MAXBNZ, LB
VBR	DESCRA, VAL, INDX, BINDX, RPNTR, CPNTR, BPNTRB, BPNTRE

Table 1.1: *args(A)* for each data structure.

1.4 Storage Conventions for Sparse Matrices

In this section, we will describe the data structures and corresponding *args(A)*. Before continuing, one should note that the list of data structures for A presented here is not necessarily complete. We intend to add (or delete) items from this list if a justifiable argument can be presented. In particular, we will add a data structure if it can be shown that it offers a significant performance or algorithmic gains over the existing data structures for an important architecture or class of problems. We would rather not add data structures that are similar to the ones listed.

For example, there are several varieties of the compressed sparse row data structure (CSR) with little or no difference in performance between the variations and the CSR data structure presented here. Thus, we choose to avoid supporting all the variations.

Table 1.1 lists *args(A)* for all data structures. The following sections describe the exact meaning of the arguments for each data structure.

1.4.1 Definitions and Notation

For the following discussion of data structures, we define some terms and notation for an m by k matrix A:

Entry - Any matrix coefficient which is handled explicitly (see [4]). There are two basic types of entries discussed in this proposal:

- *point* - Each entry is a single scalar value. Typically, the point entries of A are simply the non-zero elements of A.

- *block* - Each entry is a dense rectangular matrix stored column by column. Block entries are derived from an assumed row and column partition of the matrix A.

$NNZ(A)$ - The number of point entries of a matrix A.

$NZE(A)$ - The set of point entries of a matrix A.

$NZI(A)$ - The set of row indices corresponding to the point entries of a matrix A. The ordering of elements of $NZI(A)$ should be the same as $NZE(A)$.

$NZJ(A)$ - The set of column indices corresponding to the point entries of a matrix A. The ordering of elements of $NZJ(A)$ should be the same as $NZE(A)$.

A_{i*} - The vector consisting of the elements from row i of the matrix A.

A_{*j} - The vector consisting of the elements from column j of the matrix A.

$diag_i(A)$ - The vector consisting of the elements from a diagonal of A.

- $i < 0$ implies that $diag_i(A)$ is a vector with the first $|i|$ elements being unspecified or containing boundary information. The next q values of $diag_i(A)$ are from the $|i|^{th}$ diagonal below the main diagonal where $q = \min(m - |i|, k)$

- $i > 0$ implies that the first q elements, $q = \min(m, k - i)$, of $diag_i(A)$ are from the i^{th} diagonal above the main diagonal of A and the remaining elements are unspecified or contain boundary information.

- $i = 0$ implies $diag_i(A)$ is the main diagonal of A.

$PRF(A_{i*})$ - The set of elements from row i of A starting with the first point entry in row i up to and including the last point entry. Zero elements are included.

$PRF(A_{*j})$ - The set of elements from column j of A starting with the first point entry in column j down to and including the last point entry. Zero elements are included.

$\hat{A}$ - The matrix A with an assumed row and column partitioning.

$BNNZ(\hat{A})$ - The number of block entries of a matrix A where $\hat{A}$ denotes A with an assumed partition.

$BNZE(\hat{A})$ - The set of block entries of a matrix A. Each block entry is a dense rectangular matrix stored column by column.

$BNZJ(\hat{A})$	-	The set of block column indices corresponding to the block entries of a matrix A. The ordering of elements of $BNZJ(\hat{A})$ should be the same as $BNZE(\hat{A})$.
$\hat{A}_{i*}$	-	The block vector consisting of the elements from block row i of the partitioned matrix $\hat{A}$.
$\hat{A}_{*j}$	-	The block vector consisting of the elements from block column j of the partitioned matrix $\hat{A}$.
$\bar{A}$	-	The matrix PA where P is a permutation matrix.

Note:

1. The data structures presented here handle square or rectangular matrices, and allow for zero rows and columns.
2. No order is assumed for storing the entries of the matrices except that for triangular matrices, we assume that the main diagonal elements are stored in proper relation to lower or upper triangle. For example, if a non-unit diagonal, upper triangular matrix is stored in CSR format, then we assume that the diagonal is the first entry in each row. This is necessary for efficient solution of triangular systems.

1.4.2 Descriptor Arguments

The first argument in *args(A)* is DESCRA. It is a nine-element integer array which describes the relevant characteristics of the matrix. The first three elements of DESCRA describe the matrix structure and type of main diagonal. The fourth through ninth elements of DESCRA can be used for data-structure-specific information. Presently DESCRA(5) indicates the ordering of block entry elements for constant block entry matrices (see Section 1.4.4) and DESCRA(6) is used in the case of the sparse diagonal data structures (DIA and BDI), where it indicates whether or not the matrix is a stencil-based operator. DESCRA(6) is also used by the COO, ELL, JAD, BCO and BEL data structures to assert that there are no repeated indices in the index arrays. The possible values of DESCRA(1)–DESCRA(3) are:

- DESCRA(1) - Matrix structure
  - 0 - General
  - 1 - Symmetric
  - 2 - Hermitian
  - 3 - Triangular
  - 4 - Anti-symmetric (Skew Symmetric/Hermitian)
  - 5 - Diagonal
- DESCRA(2) - Upper/Lower triangular indicator
  - 1 - Lower
  - 2 - Upper
- DESCRA(3) - Main diagonal type
  - 0 - Non-unit
  - 1 - Unit (diagonal elements are not stored)
- DESCRA(4) - Array base

  - 0 - Zero array base (typical for C/C++)
  - 1 - One array base (typical for Fortran)

Note:

1. For the solution of triangular systems, we only support DESCRA(1) = 3. We support all combinations of data and matrix structures for matrix multiplication.

2. The diagonal type for block entry matrices refers to the block diagonal. Currently we only support unit block diagonal for triangular solve routines. See Section 1.5.4 for a discussion of non-unit block diagonal entries.

3. If a matrix is declared to be anti-symmetric, we assume that if the diagonal entries are stored then they are explicitly zero.

4. The SKY data structure is not supported for a general matrix structure (DESCRA(1) = 0).

5. If DESCRA(4) = 1 then addresses based index vector values, e.g., INDX, would be assumed to start at one. Thus, a reference using INDX(i) = 1 would point to the first element of an array. An array base of one is the default for Fortran programs. DESCRA(4) = 0 would indicate that a reference using INDX(i) = 1 points to the second element. An array base of zero is assumed for C and C++ programs.

The remaining arguments in *args(A)* vary and are described below. For precision and conciseness, these arguments are described in terms of the notation defined above. Examples are given for each data structure to illustrate the definition. Section 1.4.3 describes each point entry data structure. Section 1.4.4 describes the BSR data structure and its relationship to CSR. Other constant block size data structures are not discussed since they are straight forward extensions of the point entry equivalents. In Section 1.4.5 we describe the VBR data structure.

1.4.3 Point Entry Data Structures

COO - Coordinate Three arrays are required for the COO format:

- VAL - A scalar array of length $NNZ(A)$ consisting of $NZE(A)$, the entries of A, in any order.

- INDX - An integer array of length $NNZ(A)$ consisting of $NZI(A)$, the corresponding row indices of the entries of A.

- JNDX - An integer array of length $NNZ(A)$ consisting of $NZJ(A)$, the corresponding column indices of the entries of A.

We also need the argument NNZ = $NNZ(A)$.

For example, let

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 \\ 0 & 0 & 23 & 24 & 0 \\ 31 & 32 & 33 & 34 & 0 \\ 0 & 42 & 0 & 44 & 0 \\ 51 & 52 & 0 & 0 & 55 \end{pmatrix}, \tag{1.1}$$

then one representation of A in COO format (assuming DESCRA(4) = 1) would be as follows.

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

If A is symmetric (or Hermitian, triangular, or anti-symmetric) then we only need to store the lower (or upper) triangle. In this case we have (for the lower triangle)

$$\begin{array}{rcccccccccccc}
\mathrm{VAL} & = & (& 11 & 51 & 31 & 32 & 52 & 33 & 42 & 55 & 44 &), \\
\mathrm{INDX} & = & (& 1 & 5 & 3 & 3 & 5 & 3 & 4 & 5 & 4 &), \\
\mathrm{JNDX} & = & (& 1 & 1 & 1 & 2 & 2 & 3 & 2 & 5 & 4 &).
\end{array}$$

Note: If A is Hermitian and the diagonal of A is stored, then we will assume that the imaginary part of the diagonal is zero. Similarly, if A is anti-symmetric and the diagonal is stored, we will assume it is zero.

For the COO data structure, DESCRA(6) is used to indicate whether or not INDX has repeated values. In this case, if it is known that all columns of INDX have no repeated values, then there is no danger of overwriting values when executed in parallel. Performance improvements can be substantial if the user can guarantee there are no repeated indices and indicates so by setting DESCRA(6) = 1. The possible values for DESCRA(6) are:

- 0 - Unknown.
- 1 - No repeated indices.

CSC - Compressed Sparse Column A matrix A is stored in the CSC format using four arrays.

- VAL - A scalar array of length $NNZ(A)$ consisting of the entries of A:

$$\mathrm{VAL} = (NZE(A_{*1}), NZE(A_{*2}), \ldots, NZE(A_{*k})).$$

- INDX - An integer array of length $NNZ(A)$ consisting of the row indices of entries of A:

$$\mathrm{INDX} = (NZI(A_{*1}), NZI(A_{*2}), \ldots, NZI(A_{*k})).$$

- PNTRB - An integer array of length k such that $\mathrm{PNTRB}(j) - \mathrm{PNTRB}(1) + 1$ points to the location in VAL of the first element of $NZE(A_{*j})$. If $NZE(A_{*j})$ is empty then set $\mathrm{PNTRB}(j) = \mathrm{PNTRB}(j+1)$.

- PNTRE - An integer array of length k such that $\mathrm{PNTRE}(j) - \mathrm{PNTRB}(1)$ points to the location in VAL of the last element of $NZE(A_{*j})$. If $NZE(A_{*j})$ is empty then set $\mathrm{PNTRE}(j) = \mathrm{PNTRB}(j)$.

For example, the CSC representation of the matrix in Equation 1.1 would be:

$$\begin{array}{rccccccccccccccccc}
\mathrm{VAL} & = & (& 11 & 31 & 51 & 32 & 42 & 52 & 13 & 23 & 33 & 14 & 24 & 34 & 44 & 55 &), \\
\mathrm{INDX} & = & (& 1 & 3 & 5 & 3 & 4 & 5 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 5 &), \\
\mathrm{PNTRB} & = & (& 1 & 4 & 7 & 10 & 14 &), \\
\mathrm{PNTRE} & = & (& 4 & 7 & 10 & 14 & 15 &),
\end{array}$$

Note: 1. The actual values in PNTRB and PNTRE are not important, only their relative value from PNTRB(1). This allows greater flexibility for the user. In particular, it is common to construct pointer arrays starting at 0. For Equation 1.1, we would then define PNTRB and PNTRE as

$$\begin{array}{rcccccccc}
\mathrm{PNTRB} & = & (& 0 & 3 & 6 & 9 & 13 &), \\
\mathrm{PNTRE} & = & (& 3 & 6 & 9 & 13 & 14 &).
\end{array}$$

For the rest of the examples in this proposal, we use PNTRB(1) = 1, but this is only for convenience.

2. One can represent PNTRB and PNTRE for this example by using a single array PN-TRwhere
$$\text{PNTR} = (\ 1\ \ 4\ \ 7\ \ 10\ \ 14\ \ 15\ ),$$
in which case PNTRB(1:5) = PNTR(1:5) and PNTRE(1:5) = PNTR(2:6). In this way, the change from a single PNTR array in previous proposals to the two arrays PNTRB and PNTRE is compatible for user having only the PNTR array.
3. The two array approach to pointers offers much more flexibility than the previous one array approach. For example, if we define the array
$$PNTRD = (\ 1\ \ 4\ \ 9\ \ 12\ \ 13\ ),$$
to point to the diagonal elements, then letting PNTRB= PNTRD we can use just the lower triangular part of the general CSC representation without modifying or copying the other data structures.
4. A second example of increased flexibility is that there is no longer an implicit storage association between contiguous columns. The columns can be specified in any order. In particular, for the example above it is possible to put column one as the last column stored:
$$\begin{array}{rcl} \text{VAL} & = & (\ 32\ \ 42\ \ 52\ \ 13\ \ 23\ \ 33\ \ 14\ \ 24\ \ 34\ \ 44\ \ 55\ \ 11\ \ 31\ \ 51\ ), \\ \text{INDX} & = & (\ 3\ \ 4\ \ 5\ \ 1\ \ 2\ \ 3\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 1\ \ 3\ \ 5\ ), \\ \text{PNTRB} & = & (\ 12\ \ 1\ \ 4\ \ 7\ \ 11\ ), \\ \text{PNTRE} & = & (\ 15\ \ 4\ \ 7\ \ 11\ \ 12\ ), \end{array}$$

If A is symmetric then we only need to store the lower (or upper) triangle. In this case we have (for the lower triangle)
$$\begin{array}{rcl} \text{VAL} & = & (\ 11\ \ 31\ \ 51\ \ 32\ \ 42\ \ 52\ \ 33\ \ 44\ \ 55\ ), \\ \text{INDX} & = & (\ 1\ \ 3\ \ 5\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\ ), \\ \text{PNTRB} & = & (\ 1\ \ 4\ \ 7\ \ 8\ \ 9\ ). \\ \text{PNTRE} & = & (\ 4\ \ 7\ \ 8\ \ 9\ \ 10\ ). \end{array}$$

CSR - Compressed Sparse Row (Point entry form) A matrix A is stored in the CSR data structure using four arrays.

- VAL - A scalar array of length $NNZ(A)$ consisting of the entries of A:
$$\text{VAL} = (NZE(A_{1*}), NZE(A_{2*}), \ldots, NZE(A_{m*})).$$
- INDX - An integer array of length $NNZ(A)$ consisting of the column indices of entries of A:
$$\text{INDX} = (NZJ(A_{1*}), NZJ(A_{2*}), \ldots, NZJ(A_{m*})).$$
- PNTRB - An integer array of length m such that PNTRB(i) − PNTRB(1) + 1 points to the location in VAL of the first element of $NZE(A_{i*})$. If $NZE(A_{i*})$ is empty then set PNTRB(i) = PNTRB($i+1$).
- PNTRE - An integer array of length m such that PNTRE(i) − PNTRB(1) points to the location in VAL of the last element of $NZE(A_{i*})$. If $NZE(A_{i*})$ is empty then set PNTRE(i) = PNTRB(i).

For example, the CSR representation of the matrix in Equation 1.1 would be:

$$\begin{array}{rcl}
\text{VAL} & = & (\ 11\ \ 13\ \ 14\ \ 23\ \ 24\ \ 31\ \ 32\ \ 33\ \ 34\ \ 42\ \ 44\ \ 51\ \ 52\ \ 55\ ), \\
\text{INDX} & = & (\ 1\ \ 3\ \ 4\ \ 3\ \ 4\ \ 1\ \ 2\ \ 3\ \ 4\ \ 2\ \ 4\ \ 1\ \ 2\ \ 5\ ), \\
\text{PNTRB} & = & (\ 1\ \ 4\ \ 6\ \ 10\ \ 12\ ), \\
\text{PNTRE} & = & (\ 4\ \ 6\ \ 10\ \ 12\ \ 15\ ).
\end{array}$$

If A is symmetric then we only need to store the lower (or upper) triangle. Considering only the lower triangle of A, we have

$$\begin{array}{rcl}
\text{VAL} & = & (\ 11\ \ 31\ \ 32\ \ 33\ \ 42\ \ 44\ \ 51\ \ 52\ \ 55\ ), \\
\text{INDX} & = & (\ 1\ \ 1\ \ 2\ \ 3\ \ 2\ \ 4\ \ 1\ \ 2\ \ 5\ ), \\
\text{PNTRB} & = & (\ 1\ \ 2\ \ 2\ \ 5\ \ 7\ ). \\
\text{PNTRE} & = & (\ 2\ \ 2\ \ 5\ \ 7\ \ 10\ ).
\end{array}$$

Note: See the discussion on the use of PNTRB and PNTRE in the previous discussion of the CSC data structure.

DIA - Sparse Diagonal (Point entry form) Let LDA $\geq \min(m, k)$, and let NDIAG denote the number of non-zero diagonals of A. Two arrays are required for the DIA format:

- VAL - A two-dimensional LDA-by-NDIAG scalar array consisting of the NDIAG non-zero diagonals of A in any order.
- IDIAG - An integer array of length NDIAG consisting of the corresponding indices of the non-zero diagonals of A in VAL. Thus, if IDIAG$(j) = i$ then the j^{th} column of VAL contains $diag_i(A)$.

For example, let

$$A = \begin{pmatrix} 11 & 0 & 13 & 0 & 0 \\ 21 & 0 & 0 & 24 & 0 \\ 31 & 32 & 33 & 0 & 35 \\ 0 & 42 & 0 & 44 & 0 \\ 0 & 0 & 53 & 0 & 55 \end{pmatrix},$$

then A would be stored in DIA format as follows.

$$\text{VAL} = \begin{pmatrix} * & * & 11 & 13 \\ * & 21 & 0 & 24 \\ 31 & 32 & 33 & 35 \\ 42 & 0 & 44 & * \\ 53 & 0 & 55 & * \end{pmatrix},$$

$$\text{IDIAG} = \begin{pmatrix} -2 & -1 & 0 & 2 \end{pmatrix}.$$

For the DIA data structure, DESCRA(6) is used to determine how we should treat the '*' elements. This is useful if the operator is truly grid-based and boundary values are stored in the matrix data structure. If DESCRA(6) = 1, then the '*' elements are accessed and assumed to hold boundary value information from a stencil operator. Note also that in this case elements of B are accessed which fall outside the normal domain. For instance, in the above example, the normal domain of each column, B(*, j), of B is 1 through 5, but the first '*' element of VAL will be multiplied by B(-1, j) and added to C(1, j). The last '*' element will be multiplied by B(7, j) and added to C(5, j). If DESCRA(6) = 0, then '*' elements are unspecified and unused. If A is symmetric then we only need to store diagonals from the lower (or upper) triangle of A.

ELL - Ellpack/Itpack (Point entry form) Let MAXNZ denote the maximum of the number of entries in each row of A, i.e.,

$$\mathrm{MAXNZ} = \mathrm{MAXNZ}(A) = \max_{1 \leq i \leq m} NNZ(A_{i*}),$$

and let LDA $\geq m$. A matrix A is stored in the ELL format using two arrays.

- VAL - A two-dimensional LDA-by-MAXNZ scalar array such that row i of VAL consists of $NZE(A_{i*})$ padded by zero values if the length of $NZE(A_{i*})$ is less than MAXNZ.
- INDX - A two-dimensional LDA-by-MAXNZ integer array such that row i of INDX consists of $NZJ(A_{i*})$ padded by the integer value i if $NNZ(A_{i*})$ is less than MAXNZ.

For example, the ELL representation of the matrix in Equation 1.1 would be:

$$\mathrm{VAL} = \begin{pmatrix} 11 & 13 & 14 & 0 \\ 23 & 24 & 0 & 0 \\ 31 & 32 & 33 & 34 \\ 42 & 44 & 0 & 0 \\ 51 & 52 & 55 & 0 \end{pmatrix},$$

$$\mathrm{INDX} = \begin{pmatrix} 1 & 3 & 4 & 1 \\ 3 & 4 & 2 & 2 \\ 1 & 2 & 3 & 4 \\ 2 & 4 & 4 & 4 \\ 1 & 2 & 5 & 5 \end{pmatrix}.$$

If A is symmetric then we only need to store entries from the lower (or upper) triangle of A.

Note: If A is lower triangular and non-unit, then all diagonal entries must be in the last column of value and all padding entries must precede the diagonal.

For the ELL data structure, DESCRA(6) is used to indicate whether or not columns of INDX have repeated values. This can be useful information when computing with the transpose or symmetric form. In this case, if it is known that all columns of INDX have no repeated values, then there is no danger of overwriting values when executed in parallel. Performance improvements can be substantial if the user can guarantee there are no repeated indices and indicates so by setting DESCRA(6) = 1. The possible values for DESCRA(6) are:

- 0 - Unknown.
- 1 - No repeated indices.

JAD - Jagged Diagonal (Point entry form) The JAD format requires the specification of a (non-unique) permutation matrix P which permutes the rows of A in descending order of $NNZ(A_{i*})$. As mentioned in Section 1.3.2, a permutation is represented by an integer vector IPERM. Let $\bar{A} = PA$ be the permuted matrix and let i' denote the row of $\bar{A}$ corresponding to row i of A. Four arrays are needed:

- VAL - A scalar array of length $NNZ(A)$ consisting of the entries of A. The first element of VAL is the first element of $NZE(\bar{A}_{1*})$ followed by the first element of $NZE(\bar{A}_{2*})$ and so on through the first value of $NZE(\bar{A}_{m'*})$ where m' is the last row of $\bar{A}$ that has non-zero entries (usually $m' = m$). The $m' + 1^{st}$ element of VAL is the second element of $NZE(\bar{A}_{1*})$ and so on.

- INDX - An integer array of length $NNZ(A)$ consisting of the corresponding column indices of the entries of A.
- PNTR - An integer array of length MAXNZ + 1, where MAXNZis as defined in Section 1.4.3, such that PNTR(j) $-$ PNTR(1) + 1 points to the location in VAL of the j^{th} element of $NZE(\bar{A}_{1*})$. PNTR(MAXNZ + 1) is set to the value $NNZ(A)$ + PNTR(1).
- IPERM - An integer array of length m such that $i = IPERM(i')$. IPERM is used to determine the order in which rows of C are updated. If IPERM(1) = 0, then we assume by convention that $P = I$. One should note that IPERM represents P^T.

For example, an appropriate permutation (and its inverse) for the matrix in equation 1.1 would be:

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}, P^T = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix},$$

and $\bar{A}$ would be:

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

The elements of the JAD arrays would be:

$$\begin{array}{rcl} \text{VAL} & = & (\ 31\ \ 11\ \ 51\ \ 23\ \ 42\ \ 32\ \ 13\ \ 52\ \ 24\ \ 44\ \ 33\ \ 14\ \ 55\ \ 34\ ), \\ \text{INDX} & = & (\ 1\ \ 1\ \ 1\ \ 3\ \ 2\ \ 2\ \ 3\ \ 2\ \ 4\ \ 4\ \ 3\ \ 4\ \ 5\ \ 4\ ), \\ \text{PNTR} & = & (\ 1\ \ 6\ \ 11\ \ 14\ \ 15\ ), \\ IPERM & = & (\ 3\ \ 1\ \ 5\ \ 2\ \ 4\ ). \end{array}$$

If A is symmetric then we only need to store the lower (or upper) triangle. In this case, P is computed for the triangular part and will typically be different than P in the general case.

For the JAD data structure, DESCRA(6) is used to indicate whether or not there are repeated values in INDX in the section of INDX starting at PNTR(i) and ending at PNTR(i+1)-1, for each i = 1, MAXNZ. This can be useful information when computing with the transpose or symmetric form. In this case, if it is known that each of these sections of INDX have no repeated values, then there is no danger of overwriting values when executed in parallel. Performance improvements can be substantial if the user can guarantee there are no repeated indices and indicates so by setting DESCRA(6) = 1. The possible values for DESCRA(6) are:

- 0 - Unknown.
- 1 - No repeated indices.

Symmetrically Permuted JAD To avoid permuting the rows of C each time a JAD kernel is called, one can explicitly permute the columns of $\bar{A}$ by calling
DJADRP(0, 5, 5, DESCRA, VAL, INDX, PNTR, MAXNZ, IPERM, WORK, LWORK).

This call will compute

$$\bar{A}P^T = \begin{pmatrix} 33 & 31 & 0 & 32 & 34 \\ 13 & 11 & 0 & 0 & 14 \\ 0 & 51 & 55 & 52 & 0 \\ 23 & 0 & 0 & 0 & 24 \\ 0 & 0 & 0 & 42 & 44 \end{pmatrix}.$$

The elements of the JAD arrays for this fully permuted system would be:

$$\begin{array}{rcl} \text{VAL} & = & (\ 33\ \ 13\ \ 51\ \ 23\ \ 42\ \ 31\ \ 11\ \ 55\ \ 24\ \ 44\ \ 32\ \ 14\ \ 52\ \ 34\ ), \\ \text{INDX} & = & (\ 1\ \ 1\ \ 2\ \ 1\ \ 4\ \ 2\ \ 2\ \ 3\ \ 5\ \ 5\ \ 4\ \ 5\ \ 4\ \ 5\ ), \\ \text{PNTR} & = & (\ 1\ \ 6\ \ 11\ \ 14\ \ 15\ ). \end{array}$$

IPERM is no longer needed and an explicit zero value can be put in its place. Given the fully permuted matrix A and replacing C with PC and B with PB, we have an equivalent system to the original A, B and C except in a permuted coordinate system.

SKY - Skyline (Point entry form) A matrix A is stored in the SKY data structure using two arrays.

- If DESCRA(2) = 1, then we have:
  - VAL - A scalar array consisting of:

    $$\text{VAL} = (PRF(A_{1*}), PRF(A_{2*}), \ldots, PRF(A_{m*}))$$

  - PNTR - An integer array of length $m+1$ such that PNTR(i) − PNTR(1) + 1 points to the location in VAL of the first element of $PRF(A_{i*})$. PNTR($m+1$) is set to the value $NNZ(A)$ + PNTR(1). If $PRF(A_{i*})$ is empty, then set PNTR(i) = PNTR($i+1$).
- If DESCRA(2) = 2, then:
  - VAL - A scalar array consisting of:

    $$\text{VAL} = (PRF(A_{*1}), PRF(A_{*2}), \ldots, PRF(A_{*k}))$$

  - PNTR - An integer array of length $k+1$ such that PNTR(j) − PNTR(1) + 1 points to the location in VAL of the first element of $PRF(A_{*j})$. PNTR($k+1$) is set to the value $NNZ(A)$ + PNTR(1). If $PRF(A_{*j})$ is empty, then set PNTR(j) = PNTR($j+1$).

Note: This data structure is not supported for a general matrix structure (DESCRA(1) = 0).

For example, let

$$A = \begin{pmatrix} 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 31 & 32 & 33 & 0 & 0 \\ 0 & 42 & 0 & 44 & 0 \\ 51 & 52 & 0 & 0 & 55 \end{pmatrix},$$

then A would be stored in SKY format as follows (assuming DESCRA(2) = 1).

$$\begin{array}{rcl} \text{VAL} & = & (\ 11\ \ 31\ \ 32\ \ 33\ \ 42\ \ 0\ \ 44\ \ 51\ \ 52\ \ 0\ \ 0\ \ 55\ ), \\ \text{PNTR} & = & (\ 1\ \ 2\ \ 2\ \ 5\ \ 8\ \ 13\ ). \end{array}$$

1.4.4 Constant Block Entry Data Structures

Each of the above data structures allows the matrix to have block entries where each entry of the matrix is an LB-by-LB dense block. Systems of this form typically arise when there are multiple unknowns per gridpoint of a discretized partial differential equation. By exploiting this property, we can reduce the amount of integer storage, increase the performance of the kernels and allow numerical algorithms to perform analysis on the block entries. Typically LB is a small number, less than ten, determined by the number of quantities measured at each gridpoint, e.g., velocity, pressure, viscosity, etc. However, there are cases where LB can be on the order of 100 as it is in the case of complex chemically reacting flows.

There are two common ways to order the elements of block entries in memory. One is to order the elements of each block entry contiguously, e.g., if LB = 2 then, for each block entry, we store element (1,1) followed by element (2,1) followed by (1,2) and (2,2). Thus, each block entry is stored in LB^2 contiguous memory locations. Another way is to store the (1,1) elements of all block entries contiguously followed by all the (2,1) elements and so on.

The first ordering is particularly suitable for cache-based microprocessors. It is also the natural data structure for performing analysis on the block entries since each entry is a dense LB-by-LB matrix which could be passed, for example, to an LAPACK routine. However, this ordering is typically very bad for vector computers, especially if LB is small, since it forces vectorization with non-unit stride memory access. The second ordering allows vectorization with unit-stride memory access, but loses the cache performance and the ability to pass block entries to other subroutines as standard dense arrays.

Since both orderings of block entry elements appear to be of importance, we provide a way for the user to choose the ordering. If DESCRA(5) = 0, then the Sparse BLAS Toolkit routines assume the first ordering, i.e., the elements of each block entry are stored contiguously. If DESCRA(5) = 1, then the second ordering is assumed.

Note that, for the sparse triangular solve routines, if the argument UNITD = 0(non-unit diagonal), then the argument DV contains the block entries of the main diagonal of a block diagonal matrix. The dimensions of DV are determined by DESCRA(5). If DESCRA(5) = 0, then the dimensions of DV are (LB,MB). Otherwise they are (MB,LB). Also, the dimensions of B and C are (LB,BLDB,NB) and (LB,BLDC,NB), respectively, if DESCRA(5) = 0, or (BLDB,LB,NB) and (BLDC,LB,NB) if DESCRA(5) = 1.

Although all of the point entry data structures discussed above have a constant block entry analogue, we only choose to support some of them. Below we present the BSR data structure which is the block entry form of the CSR data structure. The block entry forms of the other data structures are analogous.

BSR - Constant Block Compressed Sparse Row Data Structure. Let l = LB denote the dimension of the block entries of A, $\hat{A}$ denote A along with its row and column partition and m_b and k_b denote the block row and column dimensions, respectively. $\hat{A}$ is stored in BSR format using three arrays. The order of elements depends on DESCRA(5).

- If DESCRA(5) = 0, we have:
  - VAL - An array of dimension $(l, l, BNNZ(\hat{A}))$ consisting of the block entries of $\hat{A}$:

$$\mathrm{VAL} = (BNZE(\hat{A}_{1*}), BNZE(\hat{A}_{2*}), \ldots, BNZE(\hat{A}_{m_b*})).$$

- BINDX - An integer array of length $BNNZ(\hat{A})$ consisting of the block column indices of the block entries of $\hat{A}$:

$$\text{BINDX} = (BNZJ(\hat{A}_{1*}), BNZJ(\hat{A}_{2*}), \ldots, BNZJ(\hat{A}_{m_b*})).$$

- BPNTRB - An integer array of length m_b such that BPNTRB$(i)-$BPNTRB$(1)+1$ points to the location in BINDX of the the first block entry of $BNZE(\hat{A}_{i*})$. If $BNZE(\hat{A}_{i*})$ is empty then set BPNTRB$(i) =$ BPNTRB$(i+1)$.
- BPNTRE - An integer array of length m_b such that BPNTRE$(i) -$ BPNTRB(1) points to the location in BINDX of the the last block entry of $BNZE(\hat{A}_{i*})$. If $BNZE(\hat{A}_{i*})$ is empty then set BPNTRE$(i) =$ BPNTRB(i).

For example, let

$$\hat{A} = \left(\begin{array}{rr|rr|rr} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ \hline 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \\ \hline 51 & 52 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 \end{array} \right), \tag{1.2}$$

then, since $l = 2$, $\hat{A}$ would be stored in BSR format as follows.

$$\text{VAL}(1:2,1:2,1) = \begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}, \quad \text{VAL}(1:2,1:2,2) = \begin{pmatrix} 15 & 16 \\ 25 & 26 \end{pmatrix},$$

$$\text{VAL}(1:2,1:2,3) = \begin{pmatrix} 33 & 0 \\ 43 & 44 \end{pmatrix}, \quad \text{VAL}(1:2,1:2,4) = \begin{pmatrix} 35 & 36 \\ 45 & 46 \end{pmatrix},$$

$$\text{VAL}(1:2,1:2,5) = \begin{pmatrix} 51 & 52 \\ 61 & 62 \end{pmatrix},$$

$$\begin{array}{rcl} \text{BINDX} & = & (\; 1 \;\; 3 \;\; 2 \;\; 3 \;\; 1 \;), \\ \text{BPNTRB} & = & (\; 1 \;\; 3 \;\; 5 \;). \\ \text{BPNTRE} & = & (\; 3 \;\; 5 \;\; 6 \;). \end{array}$$

The order of elements of VAL in memory would be:

$$(11, 21, 12, 22, 15, 25, 16, 26, 33, 43, 0, 44, 35, 45, 36, 46, 51, 61, 52, 62)$$

- If DESCRA(5) = 1, we have:
  - VAL - An array of dimension $(BNNZ(\hat{A}), l, l)$ consisting of the entries of $\hat{A}$:

$$\begin{aligned} \text{VAL} = (&(BNZE(A_{1*})(1,1)), (BNZE(\hat{A}_{2*})(1,1)), \ldots, (BNZE(\hat{A}_{m_b*})(1,1)), \\ &(BNZE(\hat{A}_{1*})(2,1)), (BNZE(\hat{A}_{2*})(2,1)), \ldots, (BNZE(\hat{A}_{m_b*})(2,1)), \\ &\vdots \\ &(BNZE(\hat{A}_{1*})(l,l)), (BNZE(\hat{A}_{2*})(l,l)), \ldots, (BNZE(\hat{A}_{m_b*})(l,l))), \end{aligned}$$

  where $BNZE(\hat{A}_{1*})(i,j)$ denotes the (i,j) elements of the block entries.

– BINDX - An integer array of length $BNNZ(\hat{A})$ consisting of the block column indices of the block entries of A:

$$\text{BINDX} = (BNZJ(\hat{A}_{1*}), BNZJ(\hat{A}_{2*}), \ldots, BNZJ(\hat{A}_{m_b*})).$$

- BPNTRB - An integer array of length m_b such that BPNTRB(i) − BPNTRB(1) + 1 points to the location in BINDX of the the first block entry of $BNZE(\hat{A}_{i*})$. If $BNZE(\hat{A}_{i*})$ is empty then set BPNTRB(i) = BPNTRB($i+1$).

- BPNTRE - An integer array of length m_b such that BPNTRE(i) − BPNTRB(1) points to the location in BINDX of the the last block entry of $BNZE(\hat{A}_{i*})$. If $BNZE(\hat{A}_{i*})$ is empty then set BPNTRE(i) = BPNTRB(i).

The matrix $\hat{A}$ in Equation 1.2 would be stored in BSR format as follows.

$$\text{VAL}(1, 1:2, 1:2) = \begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}, \quad \text{VAL}(2, 1:2, 1:2) = \begin{pmatrix} 15 & 16 \\ 25 & 26 \end{pmatrix},$$

$$\text{VAL}(3, 1:2, 1:2) = \begin{pmatrix} 33 & 0 \\ 43 & 44 \end{pmatrix}, \quad \text{VAL}(4, 1:2, 1:2) = \begin{pmatrix} 35 & 36 \\ 45 & 46 \end{pmatrix},$$

$$\text{VAL}(5, 1:2, 1:2) = \begin{pmatrix} 51 & 52 \\ 61 & 62 \end{pmatrix},$$

$$\begin{aligned} \text{BINDX} &= (\ 1\ \ 3\ \ 2\ \ 3\ \ 1\ ), \\ \text{BPNTRB} &= (\ 1\ \ 3\ \ 5\ ). \\ \text{BPNTRE} &= (\ 3\ \ 5\ \ 6\ ). \end{aligned}$$

The order of elements of VAL in memory would be:

$$(11, 15, 33, 35, 51, 21, 25, 43, 45, 61, 12, 16, 0, 36, 52, 22, 26, 44, 46, 62)$$

If A is symmetric then we only need to store the lower (or upper) triangle. If the diagonal block entries are stored, we assume that the *entire* square block entry is stored.

1.4.5 Variable Block Entry Data Structures

It is often the case for problems with multiple unknowns per node, that the number of unknowns per node will vary, generating a matrix with a block structure where the size of the blocks varies correspondingly. If the variation in size is small, it may be advantageous to make all block rows the same size by adding identity equations to smaller blocks. However, often the the variation in size is large. For this case, we provide the variable block compressed sparse row (VBR) data structure.

Variable Block Compressed Sparse Row Data Structure. We could consider a variety of data structures to support problems with variable block matrices based on the point entry data structures discussed in Section 1.4.3. However, since to our knowledge there are few existing variable block data structures, we intend to restrict ourselves just in one data structure: the variable block compressed sparse row (VBR) which is a generalization of the supernodal data structures presented in [7, 8].

Formally we define the VBR data structure as follows. Consider an m-by-k sparse matrix A along with a row partition $P_r = \{i_1, i_2, \ldots, i_{m_b+1}\}$ and column partition $P_c = \{j_1, j_2, \ldots, j_{k_b+1}\}$ such that $i_1 = j_1 = 1$, $i_{m_b+1} = m+1$, $j_{k_b+1} = k+1$, $i_p < i_{p+1}$, $\forall p$, and $j_q < j_{q+1}$, $\forall q$. We denote A, along with its assumed row and column partitions P_r and P_c, by $\hat{A}$. Then we have the following equality

$$\hat{A} = \begin{pmatrix} \hat{A}_{11} & \hat{A}_{12} & \ldots & \hat{A}_{1k_b} \\ \hat{A}_{21} & \hat{A}_{22} & & \vdots \\ \vdots & & \ddots & \vdots \\ \hat{A}_{m_b1} & \ldots & \ldots & \hat{A}_{m_bk_b} \end{pmatrix},$$

where $\hat{A}_{pq}$ is of dimension $(i_{p+1} - i_p)$-by-$(j_{q+1} - j_q)$.

If $\hat{A}_{ij} \neq 0$ then $\hat{A}_{ij}$ is a block entry of $\hat{A}$. In the VBR format, the block entries of $\hat{A}$ are stored block row by block row. Each block entry is stored as a dense matrix in standard column major form. One floating point array and three to five integer arrays are used to store the matrix:

- VAL - A scalar array of length $NNZ(A)$ consisting of the block entries of A:

  $$\text{VAL} = (BNZE(\hat{A}_{1*}), BNZE(\hat{A}_{2*}), \ldots, BNZE(\hat{A}_{m_b*})),$$

  where each block entry is a dense rectangular matrix stored column by column.

- INDX - An integer array of length $BNNZ(\hat{A}) + 1$ such that the i-th element of INDX points to the location in VAL of the the (1,1) element of the i-th block entry. INDX$(BNNZ(\hat{A}) + 1)$ is set to the value $NNZ(A) + 1$, the ending number of nonzero point entries of A plus one.

- BINDX - An integer array of length $BNNZ(\hat{A})$ consisting of the block column indices of entries of $\hat{A}$:

  $$\text{BINDX} = (BNZJ(\hat{A}_{1*}), BNZJ(\hat{A}_{2*}), \ldots, BNZJ(\hat{A}_{m_b*})).$$

- RPNTR - An integer array of length $m_b + 1$ such that RPNTR$(i) -$ RPNTR$(1) + 1$ is the row index of the first point row in the i-th block row. RPNTR$(m_b + 1)$ is set to $m +$ RPNTR(1). Thus, the number of point rows in the i-th block row is RPNTR$(i + 1) -$ RPNTR(i).

- CPNTR - An integer array of length $k_b + 1$ such that CPNTR$(j) -$ CPNTR$(1) + 1$ is the column index of the first point column in the j-th block column. CPNTR$(k_b + 1)$ is set to $k +$ CPNTR(1). Thus, the number of point columns in the j-th block column is CPNTR$(j + 1) -$ CPNTR(j).

- BPNTRB - An integer array of length m_b such that BPNTRB$(i) -$ BPNTRB$(1) + 1$ points to the location in BINDX of the first block entry of $BNZE(\hat{A}_{i*})$. If $BNZE(\hat{A}_{i*})$ is empty then set BPNTRB$(i) =$ BPNTRB$(i + 1)$.

- BPNTRE - An integer array of length m_b such that BPNTRE$(i) -$ BPNTRB(1) points to the location in BINDX of the last block entry of $BNZE(\hat{A}_{i*})$. If $BNZE(\hat{A}_{i*})$ is empty then set BPNTRE$(i) =$ BPNTRB(i).

Note: 1. For a general matrix (DESCRA(1) = 0), CPNTR can be different from RPNTR. However, for a typical square matrix, CPNTR = RPNTR so only one copy needs to be kept. For all other matrix types, RPNTR must equal CPNTR and a single array can be passed for both arguments.

2. The array INDX is not essential for reconstructing the matrix. However, it is essential for good performance of most matrix operations and should be computed one time and kept for later use.

To illustrate the VBR data structure, consider the following matrix with the indicated row and column partitioning:

$$\hat{A} = \left(\begin{array}{rr|rrr|r|rrr|rr}
4 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 \\
1 & 5 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\
\hline
0 & 0 & 6 & 1 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 2 & 9 & 3 & 0 & 0 & 0 & 0 & 0 \\
\hline
2 & 1 & 3 & 4 & 5 & 10 & 4 & 3 & 2 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 4 & 13 & 4 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 3 & 3 & 11 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 7 & 0 & 0 \\
\hline
8 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 & 3 \\
-2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 12
\end{array}\right).$$

The sparsity pattern of $\hat{A}$ is

$$\hat{A} = \begin{pmatrix}
* & 0 & * & 0 & * \\
0 & * & * & 0 & 0 \\
* & * & * & * & 0 \\
0 & 0 & * & * & 0 \\
* & 0 & 0 & 0 & *
\end{pmatrix}.$$

It is stored in VBR format as follows:

$$\begin{array}{lcl}
\text{VAL} & = & (\ 4,\ 1,\ 2,\ 5;\ 1,\ 2;\ -1,\ 0,\ 1,\ -1;\ 6,\ 2,\ -1,\ 1, \\
 & & \ 7,\ 2,\ 2,\ 1,\ 9;\ 2,\ 0,\ 3;\ 2,\ 1;\ 3,\ 4,\ 5;\ 10; \\
 & & \ 4,\ 3,\ 2;\ 4,\ 3,\ 0;\ 13,\ 3,\ 2,\ 4,\ 11,\ 0,\ 2,\ 3, \\
 & & \ 7;\ 8,\ -2,\ 4,\ 3;\ 25,\ 8,\ 3,\ 12\ ), \\
 & & \\
\text{INDX} & = & (\ 1,\ 5,\ 7,\ 11,\ 20,\ 23,\ 25,\ 28,\ 29,\ 32,\ 35,\ 44,\ 48,\ 52\ ), \\
 & & \\
\text{BINDX} & = & (\ 1,\ 3,\ 5,\ 2,\ 3,\ 1,\ 2,\ 3,\ 4,\ 3,\ 4,\ 1,\ 5\ ), \\
 & & \\
\text{RPNTR} & = & (\ 1,\ 3,\ 6,\ 7,\ 10,\ 12\ ), \\
 & & \\
\text{CPNTR} & = & (\ 1,\ 3,\ 6,\ 7,\ 10,\ 12\ ), \\
 & & \\
\text{BPNTRB} & = & (\ 1,\ 4,\ 6,\ 10,\ 12,\ ). \\
\text{BPNTRE} & = & (\ 4,\ 6,\ 10,\ 12,\ 14\ ).
\end{array}$$

If A is symmetric (or Hermitian, triangular, or anti-symmetric), then we only need to store the block diagonal and block lower (or upper) triangle. If the diagonal block entries are stored, we assume that the *entire* square block entry is stored.

1.5 Issues for Discussion

1.5.1 Language Compatibility

We believe that the current proposal addresses all major issues related to ease-of-use from Fortran, C and C++. The latest changes of replacing PNTR (BPNTR) with PNTRB and PNTRE (BPNTRB and BPNTRE) and allowing for zero or one based arrays eliminated the known complaints.

Fortran Character Variables

The original version of the Toolkit interface made extensive use of character variables (see [10]). This was motivated by the precedent established by the dense BLAS and because of the improved readability of the Fortran calls to the toolkit kernels. However, in the interest of making the interface more accessible from other languages, especially C, we have eliminated character variables and replaced them with less intuitive, but more portable integer variables.

Support for pointers and structures

The current proposal utilizes sparse matrix data representations based on contiguous arrays (see Section 1.4), thus allowing representations in Fortran 77/90 and C/C++.

Other sparse matrix data structures (e.g. true linked-list implementations) could be proposed which utilize C structures and pointers. Furthermore, in languages like C++, it is possible to define sparse matrices so that a have a single BLAS (templated) algorithm can work for sparse matrices of various types[11], thus greatly reducing the number of that functions that need to be implemented.

The disadvantages to these sparse matrix data types are (1) they language are language-specific and thus limit their benefits to only a subset of Sparse BLAS users, (2) they would increase the number of routines in the Sparse BLAS specification, (3) compatibility of structure pointers, even among C implementations, is difficult to maintain since compilers may pad or align structure members differently.

1.5.2 Constant Block Entry Details

The block data structures presented here are only one version and they differ in detail from those presented in [12]. Several questions arise:

1. Do we need to support both orderings of block entry elements? Presently we allow A, B and C to have dimensions of A(LB,LB,NNZ), B(LB,BLDB,*) and C(LB,BLDC,*) (contiguous) and A(NNZ,LB,LB), B(BLDB,LB,*) and C(BLDC,LB,*) (non-contiguous). Can we get by with only one?

2. Presently the BINDX , BPNTRB and BPNTRE arrays correspond to the *block* entries of the matrix. However, they could also correspond to the *scalar* elements by pointing to the (1,1) scalar element of each block. Compared with the present ordering scheme, this would not affect BPNTRB and BPNTRE but, for the matrix in Equation 1.2 with DESCRA(5) = 0, BINDX as presented in Section 1.4.4 would become

$$\mathrm{BINDX} = \begin{pmatrix} 1 & 5 & 3 & 5 & 1 \end{pmatrix}.$$

1.5.3 Variable Block Column Data Structure

Following the definition of the VBR structure, it is easy to define the variable compressed block column (VBC) data structure, i.e., blocks are stored block-column-by-block-column, and the point entries are stored column-by-column within one block. One possible advantage of the VBC structure is that it could accelerate the matrix-vector (matrix-matrix) multiply operations on vector machines since the columns are longer in the VBC format. However, one needs to compute the leading dimensions of the columns. Our experience tells us that the drawback may overshadow the advantage in many cases. Therefore, we are not considering the VBC data structure in the current proposal, and will leave it as an open issue in the future.

1.5.4 Non-unit Block Diagonal Entries for Triangular Matrices

Currently we do not allow non-unit block diagonal entries for triangular matrices. This is because we are not sure how to invert these blocks when performing a block triangular solve. We see several possibilities: each diagonal block has the LU factorization of the block, or each has the explicit inverse, or each has some other factorization. Furthermore, we believe it is more convenient to keep the block diagonal separate, e.g., instead of using incomplete block LU we can use incomplete block LDU. Because of the possibilities and because of the support for block diagonal matrices we do not see the value in supporting non-unit block diagonal entries for triangular matrices.

1.5.5 Work Arrays and Dimensions

To facilitate optimization, we return in WORK(1) an integer in floating point form which is the optimal value for LWORK. Thus, setting LWORK = INT(WORK(1)) will allow optimal performance from the kernel. This convention is used in LAPACK but we are open to other techniques. Our primary doubt about this approach is that the range of integers that we can express in floating point form is smaller than the standard range of integers if WORK and LWORK have the same element size.

1.5.6 Structurally-symmetric Nonsymmetric Matrices

Some nonsymmetric problems have a symmetric nonzero pattern which could be exploited in some of the routines presented here, e.g., matrix-matrix multiplication for the CSR data structure. In this case, it would be possible to store the structure for only the lower triangle of this type of matrix and the values for the entire matrix. It is possible to do this even under the constraints of the existing routines by making two calls, one for the lower triangle (in CSR format) and one for the upper (in CSC format) using the same integer vectors in both cases. There would be some improvement in performance by processing both the lower and upper triangle simultaneously, but it brings with it some complications to the user interface. At this time, we are not planning to support this capability.

1.5.7 Sparse Matrix Update Kernel

A preliminary version of this proposal included a kernel for updating a sparse matrix. This operation is of the form

$$A \leftarrow A + \alpha A'$$

where A is a sparse matrix stored in any of the supported data structures and A' is another sparse matrix stored in one of a subset of the supported data structures. We consider this kernel to be very useful in the assembly of a sparse matrix and in converting from one data structure to another. However, at this point, we are not certain about the feasibility of the implementation of this kernel and must study the problem further before proposing it.

1.5.8 General Sparse Permutation Kernels

Presently we support only one sparse permutation kernel, the right permutation of the JAD data structure. This is necessary for the reasons presented in Section 1.2.2. However, left and right permutations of sparse matrices can be useful in a more general context (see [9]).

1.5.9 Sparse Matrix-Sparse Matrix Multiplication

Sparse matrix times sparse matrix multiplication is a basic kernel for constructing incomplete Cholesky and ILU preconditioners. We are considering kernels for this operation for a restricted set of data structures. See [1] for details.

1.6 Implementation Notes

Two different reference implementations of the Sparse BLAS Toolkit have been developed to explore practical issues related to the design of a Sparse BLAS library. One current kernel implementation (The NIST Sparse BLAS: `http://math.nist.gov/spblastk`) contains in over 10,000 routines, generated automatically from C source code, while an alternate kernel implementation (The NIST Fortran Sparse BLAS: `http://math.nist.gov/mcsd/Staff/KRemington/fspblas/`) relies on standard Fortran library design. Each implementation style has its strengths and weaknesses, which we will attempt to categorize briefly here for the benefit of future implementors. Although the implementation at the kernel level is not a formal part of the specification, these issues are directly related to adoptability of the Sparse BLAS in practice.

1.6.1 The NIST Sparse BLAS

Even for the dense case, matrix-matrix (Level 3) linear algebra routines can be highly complex when, for the sake of efficiency, various special cases are taken into account. For sparse matrices, an additional level of complexity is added by allowing various storage formats, yielding an easily overwhelming explosion of source code. To provide the broad support required for the Toolkit interface, it was vital for us to strategically plan the design of the library so that various special cases could be handled consistently and with a minimal amount of hand-coding, which is not only tedious, but also error prone.

The approach we developed introduces an additional interface layer, which provides a unique function for each of the combinatorial selections of special case parameters. All of the error checking and decision trees are programmed into the Toolkit layer, while the actual numerical computations are pushed down into the kernel layer. Routines at this lowest level are designed for minimal overhead; they have no case statements, nor any error-detection overhead. This layer was originally termed the “Lite Interface”, to indicate the absence of overhead, and was expected to provide an additional entry point for the library user. However, as the experimental results have demonstrated, (see `http://math.nist.gov/spblas/performance.html`) these entry points rarely provide any significant savings to the user and their use is not required to achieve high performance. Still, they remain important in the implementation design as efficient and convenient building blocks for higher-level routines. We refer to these low level computational kernels as the “development layer”.

Advantages of Design

The NIST Sparse BLAS includes approximately 1,300 double-precision routines (over 100K lines of code) generated *automatically* from only 49 template routines. (Support here is currently limited to the "Basic Subset" of 7 important storage formats.) The development layer routines are generated from a small number of source lines (at present about 5,000) by defining and expanding macros for successively restrictive sets of calling sequence parameters. This allows optimization and debugging changes to the core source code to be *quickly and automatically propagated* to all affected kernel routines. This is a tremendous boon to the library developer. It also *yields a relatively efficient kernel for most important special cases*, which provides a definite payoff for the library user.

Disadvantages of Design

The formalization of the development layer interface for each important parameter combination results in an explosion of subroutines at the development layer. Though the user need not be concerned with this layer, the volume of source code may seem overwhelming. Further, the developer must invest effort in understanding the source generation tools and possibly adapting them for their own needs. This is perhaps not as straightforward as locating a particular routine and optimizing, as in the traditional Fortran library approach.

1.6.2 The NIST Fortran Sparse BLAS

The NIST Fortran Sparse BLAS implements the Sparse BLAS Toolkit interface in a traditional Fortran library style. The reference implementation is not currently optimized for special cases; such optimization would be achieved by placing switches on important special cases in the Toolkit interface layer, and providing tuned Fortran either inline or in ad-hoc kernel routines.

Advantages of Design

This is the approach familiar to most scientific library implementors, and there is therefore no special "start-up" costs associated with becoming familiar with the library design. Optimizations can be easily made by library implementors for at least a limited number of special cases on perhaps the most used storage formats. Users may be able to selectively hack at the library source to satisfy their own special requirements. Finally, Fortran is probably the language most familiar to users/developers, making the implementation details more widely accessible.

Disadvantages of Design

Hand-tuning of the library source is costly and error-prone, since any implemented special cases would be intertwined in the same source routine. For this reason, it is likely that only relatively small number of special cases will be optimized.

Bibliography

[1] Randolph E. Bank and Craig C. Douglas. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics*, 1, 1993.

[2] Sandra Carney, Micheal A. Heroux, Guagye Li, Roldan Pozo, Karin A. Remington, and Kesheng Wu A Revised Proposal for a Sparse BLAS Toolkit SPARKER Working Note # 2, January 1996.

[3] David S. Dodson, Roger G. Grimes, and John G. Lewis. Sparse extensions to the FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 17(2):253–263, June 1991.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Monographs on Numerical Analysis. Oxford University Press, New York, 1986.

[5] Iain Duff, Michele Marrone, and Giuseppe Radicati. A proposal for user level sparse BLAS. Technical Report TR/PA/92/85, CERFACS, December 1992.

[6] Iain Duff, and Michele Marrone. Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User Level Interface ACM Transactions on Mathematical Software, 23(3), to appear.

[7] Elegant Mathematics, Inc., Bothell, Washington. *The EM Symmetric Sparse Supernodal Format*, 1993.

[8] Elegant Mathematics, Inc., Bothell, Washington. *The EM Unsymmetric Sparse Supernodal Format*, 1993.

[9] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[10] Michael A. Heroux. A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90, CERFACS, December 1992.

[11] Roldan Pozo Template Numerical Toolkit (TNT) `http://math.nist.gov/tnt`, 1995.

[12] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Preliminary Version.