Sparse BLAS Library: Lite and Toolkit Level Specifications

BLAS Technical Forum *

January 1997

*Edited by Roldan Pozo, Micheal A. Heroux, and Karin A. Remington

Contents

NOTE: This writeup is based on the papers, ''A Revised Proposal for a Sparse BLAS Toolkit'', (Carney, Heroux, Li, Pozo, Remington, and Wu [4]), and the "NIST Sparse BLAS Implementation Notes" (Remington, Pozo [20]).

1 Introduction

This document describes a proposal for the Sparse Basic Linear Algebra Subprograms (BLAS). A three layer interface is described, consisting of (1) a high-level, data-neutral *User* interface which specfies one routine name per mathematical operation, (2) a mid-level *Toolkit* layer, with one routine name for each operation/data-structure combination, and (3) a low-level *Lite* interface, with one routine name for each parameter combination (i.e. no imbedded case statements on arguments). The current document focuses on the Toolkit and Lite interfaces as a starting point, with future revisions to contain more specifications of the User Level Sparse BLAS [10].

The ultimate goal of this effort is to eventually have the Sparse BLAS component of the BLAS Technical Form to become a *de facto* standard. We envision that hardware vendors and software developers will provide optimized versions for some subset of the kernel routines presented here, as appropriate for the targeted scientific applications and hardware platforms. Application developers can then, via calls to higher level interfaces or application libraries, attain high performance without code modifications on a wide variety of architectures.

In particular, we highlight interfaces for routines which perform the commont 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.

For the moment, we focus 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 [5], portions of which will be incorporated in future revisions of this document.

The current goals of this proposal are to define an initial set of data-structure-dependent kernel routines and to establish a framework for extending this initial set as new data structures and operations are added. The purpose of the Toolkit and Lite kernel routines is to provide the primary foundation for higher-level application libraries, primarily the User Level Sparse BLAS described in [10].

1.1 General Overview

In the remainder of this proposal we define the details of the Sparse BLAS Toolkit and Lite specifications. In Section 1.2 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 3 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.

Section 4 discusses our rationale for the design. In this section we attempt to justify what is, and is not, included in our proposal. We also illustrate some of the intended uses of these kernels.

In Section 5, we present some issues which are open for further discussion. Among these are the relevance of a Lite interface, what functionality to support from BLAST specification, using structure and pointer based arguments, and C/C++ binding issues.

Appendices A and B discuss practical issues in the implementation and performance of the Sparse BLAS. The current specification results in over 10,000 routines and poses several challenges for development, documentation, and optimization. Although the implementation is not a formal part of the specification, these issues are directly related to adoptability of the Sparse BLAS in practice.

Finally, in Appendix C we present some sample man pages for the Toolkit interface.

1.2 Sparse BLAS Functionality (Level 2 and Level 3)

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:

$$(\text{NERR}) \leftarrow A$$

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 3. NERR is the integer value which returns information about the integrity of a sparse matrix representation. Each of the above operations is proposed for single and double precision, real and complex data, as appropriate.

1.3 Targeted Data Structures

The following sparse matrix formats are identified for Sparse BLAS support. This list may change; see Section 3 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

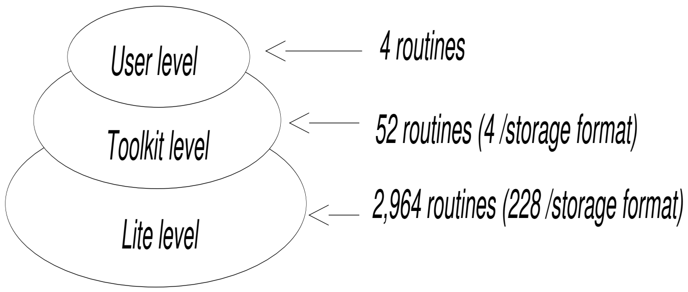


Figure 1: Layered design for the Sparse BLAS. (Numbers are for double-precision routines only.

1.4 Interface Hierarchy

One approach to standardized software for sparse linear algebra software is to provide interfaces at various levels (Figure 1), each tailored for different situtations. For example, User Level BLAS routine may utilize a single function DCSMM to perform matrix multiplication with various types of sparse matrices. The argument sequence would look the same, regardless of the particular flavor of matrix, or variation of the general matrix multiply operation. In this way, the underlying data layout may be unknown. In fact, it could even be suggested by the underlying architecture for efficiency.

While such an interface may be quite useful at the user level, its overhead may be prohibitive for use as low-level building blocks or its scope may be too encompassing for an fully optimized implementation.

The Toolkit interface, on the other hand, is specific to each matrix storage type. Thus, there is one matrix multiply routine for compressed column (DCSCMM), another for compressed row DCSRMM, and so on. Each Toolkit routine still contains many variations, based on attributes such as special values of α and β, and symmetric properties of the sparse matrix.

The Lite interface has a separate function signature for every possible set of parameter combinations, including special cases of α and β and subtle variations of the sparse matrix attributes (for example: lower symmetric, unit diagonal A in the triangular solve). The current specification produces over 10,000 Lite routines. (See Section A implementation details and Section 5.1 for relevance issues.)

2 Interface Specifications

2.1 Naming Conventions for 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 3.)
  - 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.

2.2 Argument Conventions for 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)

2.2.1 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. See Sections 3.4 and 3.5 for a full discussion of block entry matrices.

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. See Section 3.4 for details.

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)).

2.2.2 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. It is described in more detail in Section 3.4.

2.2.3 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.

2.2.4 CK Arguments

The arguments 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.

2.3 Order of Arguments for *args(A)*

For both Toolkit and Lite kernels, 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 3.

2.4 Naming Conventions for Lite Interface

The Lite routines are unique to each parameter combination of the Toolkit interface. They are designed for minimal overhead, with no case statements or elaborate error-detection overhead. Thus, they are ideal for use on small matrices or as efficient building blocks in higher-level routines. Some typical examples of the Lite routines for compressed-row (CSR) matrix:

$C \leftarrow A^T B$	`CSR_MatMult_CATB_double()`
$C \leftarrow AB + C$	`CSR_MatMult_CABC_double()`
$C \leftarrow \alpha AB + \beta C$	`CSR_MatMult_CaBbC_double()`
$C \leftarrow DA^{-1}B + C$	`CSR_MatTriangSlvLD_CDABC_double()`

The substrings '`CATB`', '`CaBbC`', and so on may seem strange, but actually follow a convenient rule: these letters are simply the arguments of algebraic expression without the operators. (The symbols α and β, if present become lowercase '`a`' and '`b`'. The argument list is then unique for each Lite function. This allows one to easily reconstruct the function name of any low-level kernel operation. (Notice that these routines are difficult to name with the 8-character limit of Fortran 77. At the moment, the Lite interface is specified using a C language binding. Nevertheless, all modern Fortran compilers and, in fact, the current international Fortran standard, Fortran 90, support long names.)

NOTE: an alternate scheme would simply extend the Toolkit name, e.g. `DCSRMM` *could become* `DCSRMM_CaABbC`. *Since such a scheme already violates the 8-character F77 restriction, the more descriptive naming convention above was proposed. This is still open to discussion.)*

Each Lite routine has a variable-length character name `XY_ZF_KT`, where

- X indicates the data structure of the Sparse Matrix: (Note: This list may change. See Section 3.)

- 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

- Y denotes optional symmetric properties of the sparse matrix:
  - symm - Symmetric
  - skew - Skew-symmetric ($A = -A^T$)
  - herm - Hermitian (available only for complex types)

- Z denotes whether it is a Level 2 or Level 3 operation:
  - Mat - matrix/matrix (Level 3)
  - Vec - matrix/vector (Level 2)

- F denotes the high-level basic functionality
  - Mult - matrix multiply
  - TriangSlv - triangular solve, with optional parameters MN, where M denotes
    - L - lower triangular matrix
    - U - upper triangular matrix

    and N denotes
    - U - unit diagonal entries are assumed
    - D - no assumption on diagonal entries

- K denotes the particular kernel arguments (see Table 1 for a listing)

- T denotes the data element type

CAB	CADBbC	CDADBC	CaADB	CaDABbC
CABC	CDAB	CDADBbC	CaADBC	CaDADB
CABbC	CDABC	CaAB	CaADBbC	CaDADBC
CADB	CDABbC	CaABC	CaDAB	CaDADBbC
CADBC	CDADB	CaABbC	CaDABC	

Table 1: Enumeration of valid kernel strings for the Lite interface.

- float - single-precision real
- double - double-precision real
- complex - single-precision complex
- dcomplex - double-precision complex

$$
\begin{array}{cccccc}
\mathrm{X} & \mathrm{Y} & \mathrm{Z} & \mathrm{F} & \mathrm{K} & \mathrm{T} \\
\left\{\begin{array}{c}\mathrm{CSR}\\ \mathrm{CSC}\\ \mathrm{COO}\\ \mathrm{BSR}\\ \mathrm{BSC}\\ \mathrm{BCO}\\ \mathrm{VBR}\end{array}\right\}
&
\left\{\begin{array}{c}-\\ \mathrm{symm}\\ \mathrm{skew}\\ \mathrm{herm}^{\dagger}\end{array}\right\}
&
\left\{\begin{array}{c}\mathrm{Mat}\\ \mathrm{Vec}\end{array}\right\}
&
\left\{\begin{array}{lcc}\mathrm{Mult} & - & -\\ \mathrm{TriangSlv} & \left\{\begin{array}{c}\mathrm{L}\\ \mathrm{U}\end{array}\right\} & \left\{\begin{array}{c}\mathrm{U}\\ \mathrm{D}\end{array}\right\}\end{array}\right\}
&
\mathrm{kernel}^{\ddagger}
&
\left\{\begin{array}{l}\mathrm{float}\\ \mathrm{double}\\ \mathrm{complex}\\ \mathrm{dcomplex}\end{array}\right\}
\end{array}
$$

Figure 2: Naming convention for Lite interface.
† – herm routines only for complex data types.
‡ – kernel operations are listed in next table.

A comprehensive chart of the Lite naming convention appears in Figure 2.

2.5 Argument Conventions for Lite Interface

The arguments to a Lite routine are typically a specializaiton of the the Toolkit interface for that function. For example, the interface to a particular variant of matrix/matrix multiply with a variable block compressed row (VBR) matrix, A, and a dense rectangular matrix B:

$$C \leftarrow AB + C$$

would have the following Toolkit interface:

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

and the following Lite interface:

```
void VBR_MatMult_CABC_double(m, n, k, \argsa, b, ldb, c, ldc
           work, ind_base,);
```

where *args(A)*is described in Section 2.3.

Due to the large number of routines in the Lite Interface (over 10,000) we describe the details only for a particular example, while illustrating the framework for extending this convention to the other cases.

Since there are several hundred functions for the VBR storage format alone, it is important that these follow a predictable scheme: start with the prototype for the most generic function (corresponding to the matrix matrix multiplication kernel CaABbC and the matrix triangular solve kernel CaDADBbC), then remove any corresponding unnecessary arguments from the argument list. For example, the vector version of the general routine

```
void VBR_MatTriangSlvLU_CaDADBbC_double}(
               const int mb, const int n,
               const double *dvl, const double *dvr,
               const double alpha,
               const double *val, const int *indx,
               const int *bindx, const int *rpntr,
               const int *cpntr, const int *bpntrb, const int *bpntre,
               const double *b, const int ldb, const double beta,
               double *c, const int ldc,
               double *work, const int ind_base);
}
```

can be obtained by changing the function name to VBR_VecTriangSlvLU_CaDADBbC_double, and eliminating the arguments {n, ldb, ldc}, to arrive at the prototype:

```
void VBR_VecTriangSlvLU_CaDADBbC_double} (
               const int mb,
               const double *dvl, const double *dvr,
               const double alpha,
               const double *val, const int *indx,
               const int *bindx, const int *rpntr,
```

```
               const int *cpntr, const int *bpntrb, const int *bpntre,
               const double *b,  const double beta,
               double *c,
               double *work, const int ind_base);
}
```

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 2: *args(A)* for each data structure.

3 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 2 lists *args(A)* for all data structures. The following subsections describe the exact meaning of the arguments for each data structure.

3.1 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 [9]). 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.

3.2 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 3.4) 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 5.9 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 3.3 describes each point entry data structure. Section 3.4 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 3.5 we describe the VBR data structure.

3.3 Point Entry Data Structures

3.3.1 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

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

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

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

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)

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

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.

3.3.2 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:

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

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

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

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

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

For example, the CSC representation of the matrix in Equation 1 would be:

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

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, we would then define PNTRB and PNTRE as

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

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

2. One can represent PNTRB and PNTRE for this example by using a single array PNTRwhere

$$\text{PNTR} = (\ 1\ 4\ 7\ 10\ 14\ 15\ ),$$

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:

$$\begin{aligned}
\text{VAL} &= (\ 32\ 42\ 52\ 13\ 23\ 33\ 14\ 24\ 34\ 44\ 55\ 11\ 31\ 51\ ),\\
\text{INDX} &= (\ \ 3\ \ 4\ \ 5\ \ 1\ \ 2\ \ 3\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 1\ \ 3\ \ 5\ ),\\
\text{PNTRB} &= (\ 12\ \ 1\ \ 4\ \ 7\ 11\ \ ),\\
\text{PNTRE} &= (\ 15\ \ 4\ \ 7\ 11\ 12\ \ ),
\end{aligned}$$

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

$$\begin{aligned}
\text{VAL} &= (\ 11\ 31\ 51\ 32\ 42\ 52\ 33\ 44\ 55\ ),\\
\text{INDX} &= (\ \ 1\ \ 3\ \ 5\ \ 3\ \ 4\ \ 5\ \ 3\ \ 4\ \ 5\ ),\\
\text{PNTRB} &= (\ \ 1\ \ 4\ \ 7\ \ 8\ \ 9\ \ ).\\
\text{PNTRE} &= (\ \ 4\ \ 7\ \ 8\ \ 9\ 10\ \ ).
\end{aligned}$$

3.3.3 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 would be:

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

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

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

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

3.3.4 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.

3.3.5 ELL - Ellpack/Itpack (Point entry form)

Let MAXNZ denote the maximum of the number of entries in each row of A, i.e.,

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

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 would be:

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

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

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.

3.3.6 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 2.2.3, 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 3.3.5, 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 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:

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

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:

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

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.

3.3.7 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).

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

3.4 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 $\hat{A}$:
  $$\text{VAL} = (BNZE(\hat{A}_{1*}), BNZE(\hat{A}_{2*}), \ldots, BNZE(\hat{A}_{m_b*})).$$
  - 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 $\text{BPNTRB}(i) - \text{BPNTRB}(1) + 1$ 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 $\text{BPNTRB}(i) = \text{BPNTRB}(i+1)$.
  - BPNTRE - An integer array of length m_b such that $\text{BPNTRE}(i) - \text{BPNTRB}(1)$ 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 $\text{BPNTRE}(i) = \text{BPNTRB}(i)$.

For example, let

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

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

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

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

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

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

The order of elements of VAL in memory would be:

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

- 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 2 would be stored in BSR format as follows.

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

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

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

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

  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.

3.5 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 3.3. 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 [11, 12].

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:

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

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:

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

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.

4 Rationale for the Library Design

In this section we discuss the reasons for our design. We believe there is little argument about the need for what we propose here. However, one may reasonably argue that there are other important kernels omitted. One may also argue with the details of our sparse matrix data structures and which data structures are supported or unsupported. We are aware of these issues and are willing to make changes if necessary. In Section 5, we list and discuss issues that we believe to be open for further review.

4.1 Language Independent Specifications

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.

4.2 Choice of Data Structures

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 3.4.

- 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.

4.3 Choice of Functionality

Below we discuss our motivation for each basic operation we chose. We realize that there are many other important operations and plan to implement more in the future. See Section 5.12 for specific future plans.

4.3.1 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 since sparse 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.

4.3.2 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
$x \leftarrow Px$

4.3.3 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.

4.4 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 3. 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.

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 3: General Distributed Sparse Matrix-Dense Matrix Product Algorithm

5 Open Issues

In this section we present a list of open issues. Certainly there are others which we have omitted, so this list will change with each revision.

5.1 Relevance of the Lite Interface

There has been some concern over the large number of routines needed for the Lite interface specification. The original justification for this was that these routines would have minimal overhead (no complicated case statements or logic based on parameter values) and execute more efficiently on target architectures. However, initial performance measurements have shown no measurable improvement, even for small sparse matrices B. In practice, it seems the computational granularity of the Sparse BLAS kernels, even for tiny point-entry matrices, is sufficiently large to hide the overhead of the Toolkit interface.

If so, is a Lite interface for the Sparse BLAS still necessary? Its most obvious drawback is the programming and documentation effort required to support over 10,000 routines that could be reduced to just a few hundred in the Toolkit interface. On the other hand, one argument for its inclusion would be that Lite routines retain simpler argument lists, since only relevant parameters are passed. (See example in Section 2.5.) Also, it is easier for vendors to optimize routines via a Lite interface, since each routine is atomic and independent.

5.2 Language Incompatibilities

5.2.1 Fortran Character Variables

The original version of the Toolkit interface made extensive use of character variables (see [15]). 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.

5.2.2 Support for pointers and structures

The current proposal utilizes sparse matrix data representations based on contiguous arrays (see Section 3), 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[19], 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.

5.3 Omission of Rank Updates

The dense BLAS support rank updates of a matrix, e.g., $A \leftarrow A + \alpha x y^T$. This operation is not useful if A is sparse since the update, xy^T, is dense. It would be possible to do an implicit rank update in the sense that we never really form the updated matrix A, but instead keep A, x and y. However, we have not seen any reason for doing this since it is possible to use a combination of sparse and dense BLAS to perform this operation. For example, to compute $w = (A + xy^T)z$ we first compute $w = Az$ using a sparse BLAS kernel, then we compute $\gamma = y^T z$ using the level-1 BLAS routine DDOT. Then we finally update $w = w + \gamma x$ using the level-1 BLAS routine DAXPY.

5.4 Omission of SIDE

In the specification of the dense BLAS, the SIDE argument is used to indicate whether the operand matrix A is to the left or right of its associated operand. We assume that the sparse operand A always appears on the left of its associated operand. Thus, the SIDE argument is unnecessary.

5.5 Minimal support for pre-processing

In many cases, high-performance implementations of the Sparse BLAS Toolkit kernels will require some pre-processing that should be done only once. Repeated calls to a Sparse BLAS Toolkit kernel for a fixed matrix A would then use this pre-processed information to achieve better performance. In this proposal, there is minimal support for pre-processing. We include only one work array. However, because we believe that addressing these issues is not within the scope of this proposal, we hesitate to add much more support for pre-processing. Also, users should not need to be concerned about these implementation-specific details. The drawback is that some types of pre-processing may be difficult to implement.

5.6 Omission of MSR, MSC and Other Data Structures

The modified sparse row (MSR) and modified sparse column (MSC) data structures (see SPARSKIT User Guide [22]) are popular data structures and were originally supported in this proposal. Also, other data structures might be very useful. However, in the interest of having a small set of supported data structures, we have chosen to avoid data structures with similar performance characteristics.

Note: Matrix multiplication and the solution of triangular systems with MSR (and MSC) can be expressed using a combination of the DIA and with CSR (and CSC) with modification of the user's data structure.

5.7 Details of Constant Block Data Structures

The block data structures presented here are only one version and they differ in detail from those presented in [22]. 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 2 with DESCRA(5) = 0, BINDX as presented in Section 3.4 would become

$$\text{BINDX} = (\ 1\ 5\ 3\ 5\ 1\ ).$$

5.8 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.

5.9 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.

5.10 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.

5.11 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.

5.12 Future Work

During the development of this proposal several other basic operations and data structures were considered for inclusion. In particular, there are three topics that are not discussed here which we believe are important and should be presented in later working notes. They are as follows.

5.12.1 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.

5.12.2 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 4.3.2. However, left and right permutations of sparse matrices can be useful in a more general context (see [14]).

5.12.3 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 [2] for details.

A Implementation Issues

While a discussion of implementation issues may seem premature at this stage of the standardization process, it is a critical component that can help expose the amount of effort required by vendors and software developers to optimize such libraries for their particular platforms. This has a direct impact on the practical success of the Sparse BLAS standardization effort. If the specification requires too much development effort to implement, vendors will be reluctant to adopt it.

For the Sparse BLAS specification, this is especially critical, since the large number of data structures supported and variations of kernel algorithms can quickly lead to unmanagable numbers of routines for documentation and optimization.

It is unlikely that vendors will optimize all of these, but rather an particular subset that occur often in their targeted applications. One possible scenario is to provide a general public-domain version of all routines (written in C for Fortran) and allow vendors to substitute optimized versions of routines for the subset they are interested in. (This is similar to the approach taken with the existing BLAS [7],[8].) This ensures portability for users and provides a common platform for optimization.

Nevertheless, developing and documenting a library with over 10,000 routines, even if they are all closely related, poses several challenges. Foremost, it is clear that each kernel routine will not be written by hand (too tedious and error prone) but rather generated automatically or (psuedo-automatically) by some coding mechanism. Identifying which code segements, program structures, and processing tools are best suited for this is critical to developing a real implementation.

A.1 A Model Implementation

To investigate these issues, we have developed a model implementation of the Sparse BLAS Toolkit and Lite routines (`http://math.nist.gov/spblas`) [21]. This is not intended to be part of the Sparse BLAS specification, but rather a vehicle for making discussions related to this subject more concrete. We are also interested in experimenting with techiniques for code distribution and making cursory performance studies. Our intent is to make the source code publicly available to get it in the hands for users, vendors, and developers for comments and feedback.

This implementation has been developed in C and is not optimized for any particular platform. Nevertheless, in Section B, we provide some preliminary performance results for various architectures.

A.1.1 Source Code Generation

The Lite kernel routines are generated from a small number of source lines (less than 5,000 for the storage formats currently supported) by defining and expanding macros for successively restrictive sets of calling sequence parameters. This allows changes to the core source code, made for optimization or debugging, to be rapidly and automatically propagated to all affected kernel routines (approximately 130,000 lines of code).

A directory contains the following *template* files

```
coomm.c cscmm.c  csrmm.c  bcomm.c  bscmm.c  bsrmm.c  vbrmm.c
        cscmts.c csrmts.c bscmts.c bsrmts.c vbrmts.c
```

These source files are used as "master files", and are written in such a way that special case routines can be generated by relatively simple shell scripts which use "sed" and "awk" for text replacement. The approach saves considerable programming effort by generating most source files automatically, and reduces errors by insuring that any changes are propagated throughout all of the related source code.

Here is an example section of a *master* file for the block coordinate (BCO) matrix multiply Lite routines:

```
#include "nsbbco.h"

/*--------------------------------------------------------------------*/
/*   BLOCK COORDINATE MATRIX MULTIPLY:                                */
/*   BCO_MatMult_kernel                                               */
/*--------------------------------------------------------------------*/
/*                                                                    */
/*   Arguments:                                                       */
/*                                                                    */
/*   (NOTE: This describes all arguments in the most general kernel   */
/*          interface.  Arguments listed here may not appear in the   */
/*          interface for the specific routines below.  Use only those */
/*          arguments which appear in the function prototype.)        */
/*                                                                    */
/*   int mb       Number of block rows in matrix A                    */
/*   int n        Number of columns in matrix c                       */
/*   int kb       Number of block columns in matrix A                 */
/*   double alpha Scalar parameter                                    */
/*   double *val  Array of length nnz containing nonzero entries of A */
/*   int *bindx   integer array of length bnnz consisting of the block */
/*                row indices of the entries of A.                    */
/*   int *bjndx   integer array of length bnnz consisting of the block */
```

```
/*               column indices of the entries of A.                     */
/*   int bnnz     number of block entries in A                           */
/*   int lb       dimension of blocks in A                               */
/*   double *b    rectangular array with first dimension ldb             */
/*   int ldb      dimension of b                                         */
/*   double beta  Scalar parameter                                       */
/*   double *c    rectangular array with first dimension ldc             */
/*   int ldc      dimension of c                                         */
/*   int ind_base indexing offset (1 for Fortran based arrays, 0 for C)  */
/*                                                                       */
/*-----------------------------------------------------------------------*/
void BCO_MatMult_CaABbC_double(
                 const int mb, const int n, const int kb, const double alpha,
                 const double *val, const int *bindx, const int *bjndx,
                 const int bnnz, const int lb,
                 const double *b, const int ldb, const double beta,
                 double *c, const int ldc,
                 const int ind_base)
{
  double t;
  const double *pb;
  double *pc=c;
  const double *pval;
  const double *ptmp;                                    /* VECDEL */
  int i,j,jb,je,block;
  int cs,bs,br;
  int index;
  int l;                                                 /* VECDEL */
  int ii,jj;
  int m=mb*lb;
  int k=kb*lb;
  int mm=lb*lb;

  for (l=0;l!=n;l++)                              /* VECDEL MMBETADEL */
    for (i=0;i!=m;i++) *pc++ *= beta;                    /* MMBETADEL */

  for (i=0;i!=bnnz;i++) {      /* For each block                        */
      cs = (bindx[i]-1)*lb;     /* start of c for this block operation */
      bs = (bjndx[i]-1)*lb;        /* start of b                        */
      pval = &val[i*mm];           /* pointer to start of block         */
      ptmp = pval;                 /* copy for later use     VECDEL     */
                                   /* GEMM call inlined here:           */
      pb = &b[bs];                 /* pointer to start of b             */
      pc = &c[cs];                 /* pointer to start of c             */
      for (l=0;l!=n;l++) {         /* For each column of c   VECDEL     */
       for (jj=0;jj!=lb;jj++) {       /* For each column of block      */
         if( pb[jj] != 0.0 ) {           /* If non-zero multiplier     */
           t = alpha * pb[jj];
```

```
          for (ii=0;ii!=lb;ii++) {       /* For each element in column */
            pc[ii] += t* (*pval++);         /* update element of c     */
          }
        } else {
          pval+=lb;                      /* Skip this column of block  */
        }
      }
      pb+=ldb; pc+=ldc; pval=ptmp;       /* to next column of c;VECDEL */
     }                                             /* VECDEL */
  }
}
```

The master files provide working source code for the most general version of the kernel routine. This is where real programming effort should be expended to optimize the library. The code is commented with tags which can be used to selectively delete code for special case routines. The "rules" for creating each special case file are defined in a separate subdirectory. The kernels subdirectory contains the files

```
CAB    CADBbC CDADBC  CaADB   CaDABbC
CABC   CDAB   CDADBbC CaADBC  CaDADB
CABbC  CDABC  CaAB    CaADBbC CaDADBC
CADB   CDABbC CaABC   CaDAB   CaDADBbC
CADBC  CDADB  CaABbC  CaDABC
```

one for each of the specializations from the generic master code, along with kernel files for the master codes. Each of these kernel files contains pointers to appropriate "Definition" files, in a subdirectory, which are used to build up the **sed** script for the text replacement to generate the kernel routines.

B Initial performance of model Sparse BLAS implementation

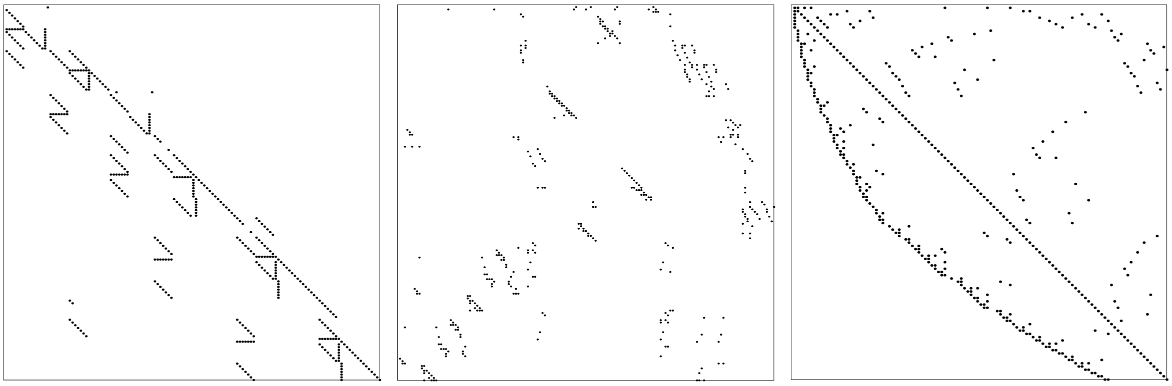
B.1 Test Problem Generation

The test matrices used in the following tests were generated by reading sparsity patterns from Harwell-Boeing Collection, and using these patterns as the block structure for a matrix of given blocksize. The results shown are for matrix-vector and matrix-matrix multiplications only, rather than a full DAXPY, since we are interested in the efficiency of the *sparse* code. Source code for the performance testers is available [21].

The test results presented in Sections B.3 and B.4 are for the matrix patterns shown in Figure 4. The three Harwell-Boeing matrices used in this study are IMPCOL C, WEST0156, and GRE 115, and were chosen to give a variety of different memory access patterns. Size information for these matrices is summarized in Table 3.

NAME	size	nonzeros
IMPCOL C	137 by 137	411
WEST0156	156 by 156	371
GRE 115	115 by 115	421

Table 3: Test matrix size information



IMPCOL C WEST0156 GRE 115

Figure 4: Sparsity patterns of test matrices.

B.2 Performance measurements

Figures 5–10 show test measurements taken on a SUN Sparc 20, and Figures 11–16 represent the same measurements taken on an IBM RS6000 Model 590, and Figures 17–22 represent those taken on a 200 MHz Intel Pentium Pro. The experimental parameters are blocksize and number of right-hand-sides. The machines were used under light load conditions, but not in dedicated mode. Each data point represents the average result of 4 runs with the same parameters.

Each graph plots three pairs of test results, one for each of the storage formats tested: Compressed Sparse Row (CSR), Block Compressed Row (BSR), and Variable Block Row (VBR). The pair of results represents the performance of the Toolkit interface and the Lite interface. Although the Lite interface was expected to improve performance by eliminating a layer of error checks and decision trees, the results indicate that the savings is minimal for problems of any substantial size. Further note that the point-wise oriented routines (for CSR) are more efficient than the block-oriented routines (for BSR and VBR) for problems with small blocksize. It is not until the blocksize reaches 5 to 10 that the block-oriented format begins to pay off. The performance of the block routines may improve if library calls to optimized dense level 3 BLAS are used. The present library uses fixed inlined versions of the dense BLAS to facilitate the automatic code generation scheme, and the possibility modularizing this part of the computation is being examined.

As expected, performance improves for block-wise Level 3 operations (compare, for example, Figure 5 and 6). However, an improvement was not seen when the number of right-hand-sides was raised from 10 to 20.

B.3 Sparse BLAS performance on SUN Sparc 20

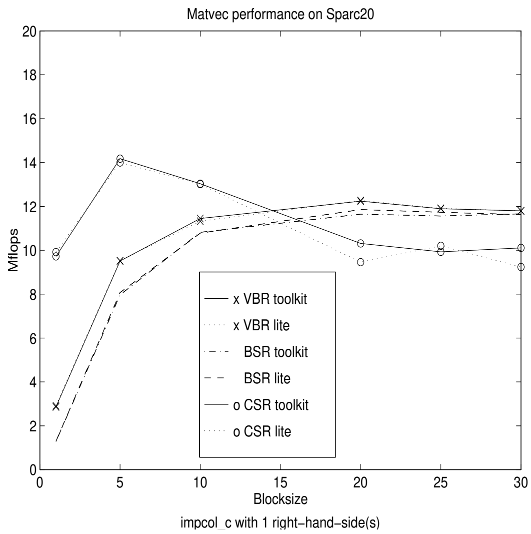


Figure 5: Sparc 20 results for IMPCOL C with 1 right-hand-side

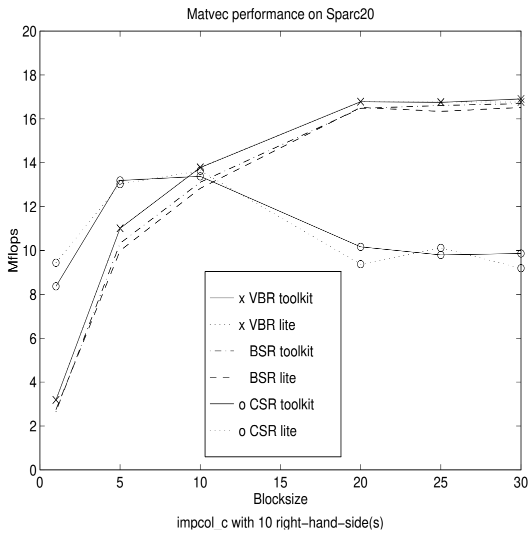


Figure 6: Sparc 20 results for IMPCOL C with 10 right-hand-sides

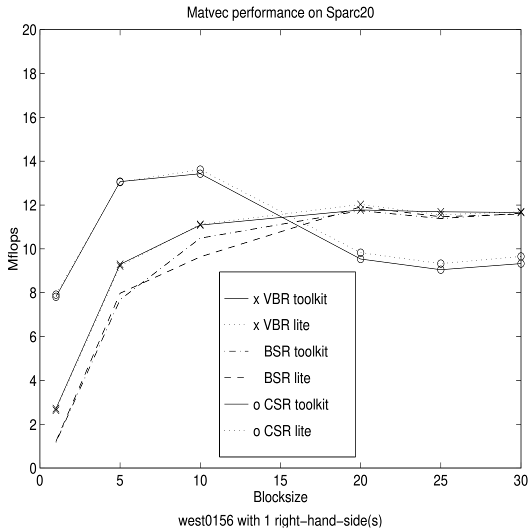


Figure 7: Sparc 20 results for WEST0156 with 1 right-hand-side

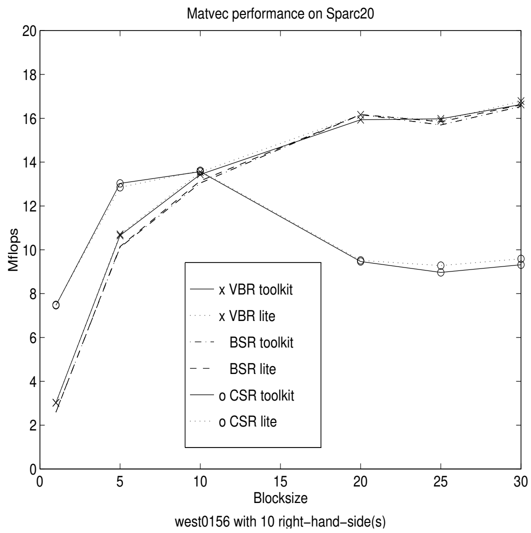


Figure 8: Sparc 20 results for WEST0156 with 10 right-hand-sides

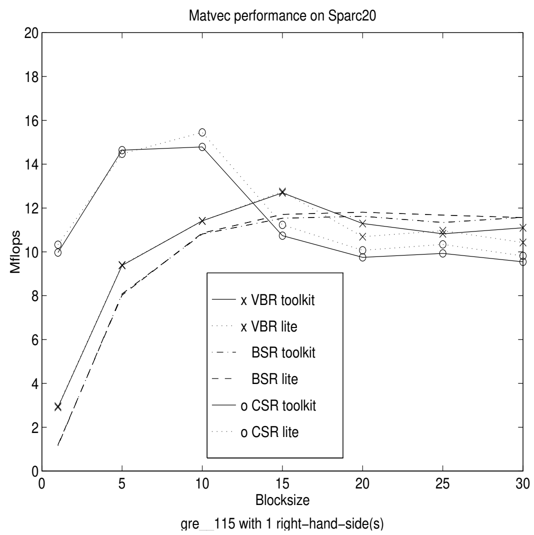


Figure 9: Sparc 20 results for GRE 115 with 1 right-hand-side

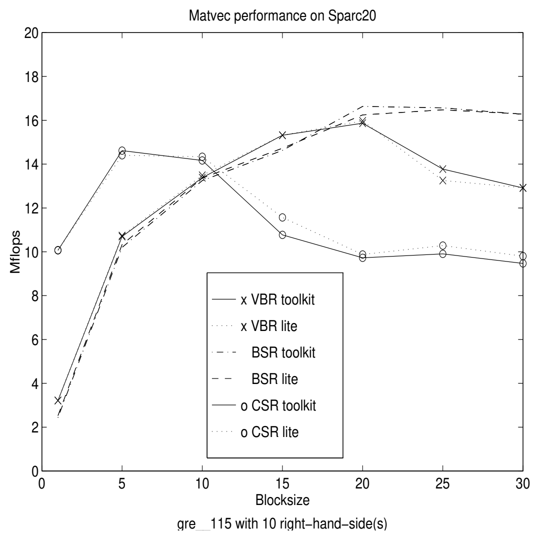


Figure 10: Sparc 20 results for GRE 115 with 10 right-hand-sides

B.4 Sparse BLAS performance on IBM RS6000 Model 590

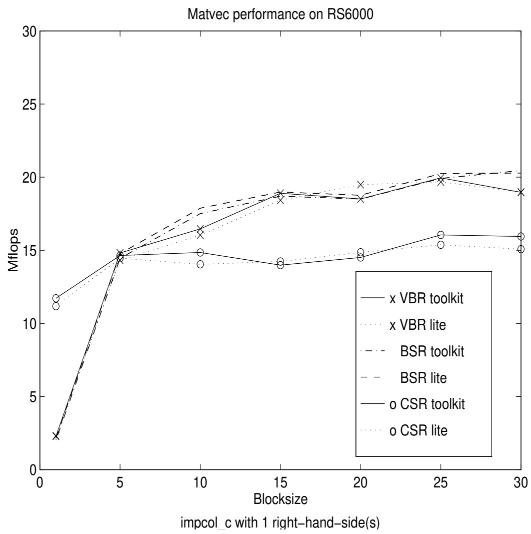


Figure 11: RS6000 results for IMPCOL C with 1 right-hand-side

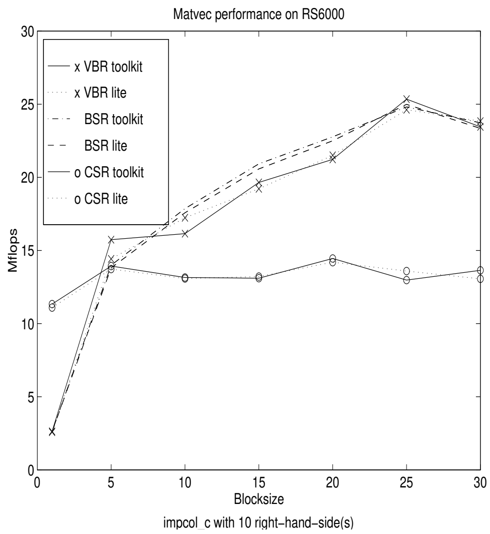


Figure 12: RS6000 results for IMPCOL C with 10 right-hand-sides

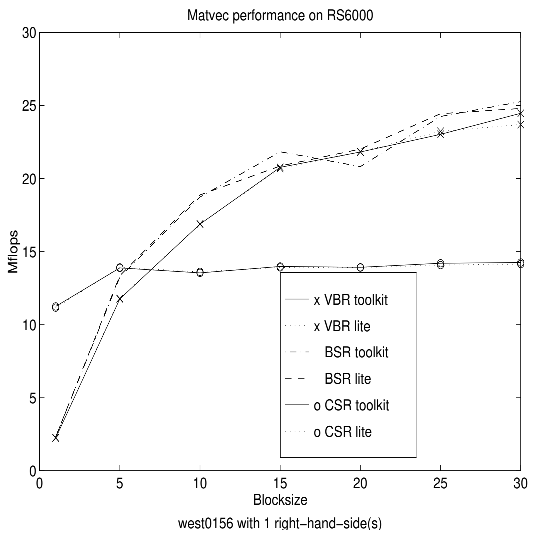


Figure 13: RS6000 results for WEST0156 with 1 right-hand-side

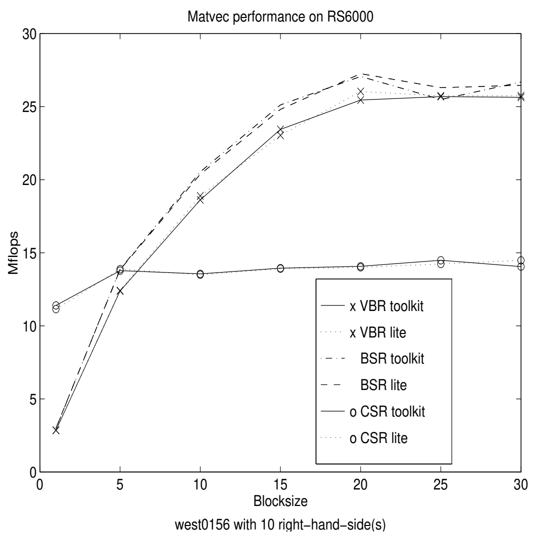


Figure 14: RS6000 results for WEST0156 with 10 right-hand-sides

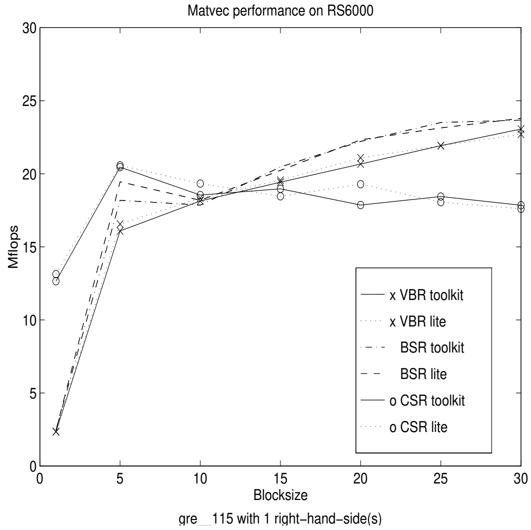


Figure 15: RS6000 results for GRE 115 with 1 right-hand-side

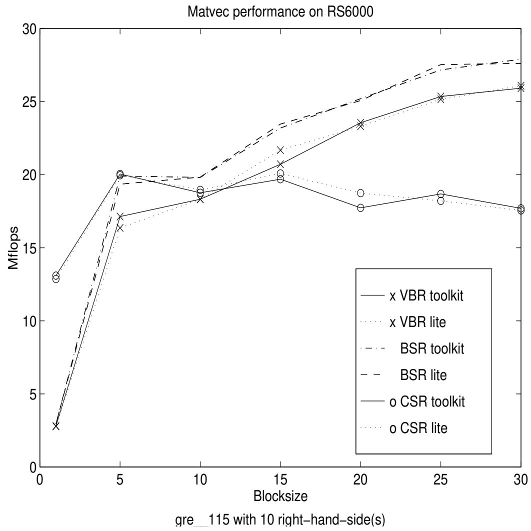


Figure 16: RS6000 results for GRE 115 with 10 right-hand-sides

B.5 Sparse BLAS performance on 200 MHz Intel Pentium Pro

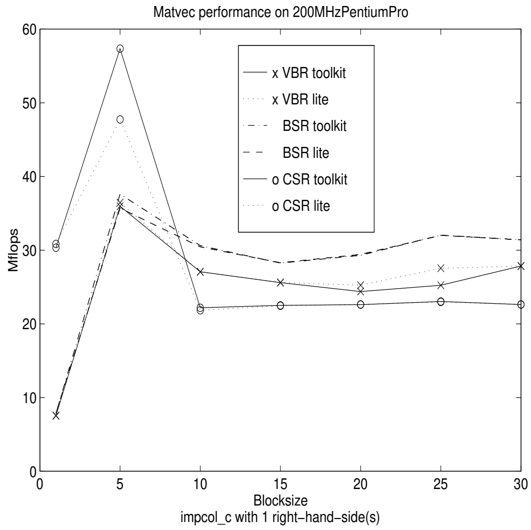


Figure 17: Pentium Pro results for IMPCOL C with 1 right-hand-side

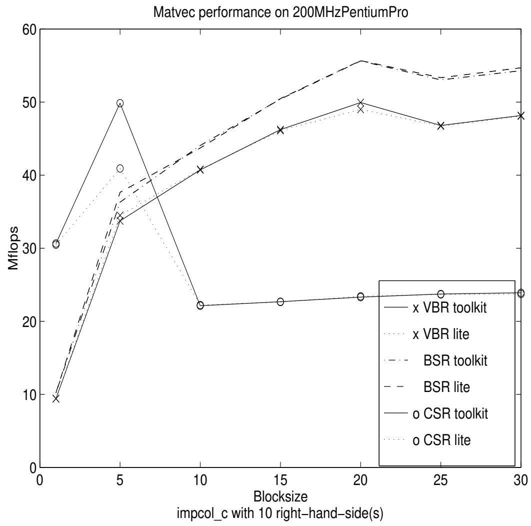


Figure 18: Pentium Pro results for IMPCOL C with 10 right-hand-sides

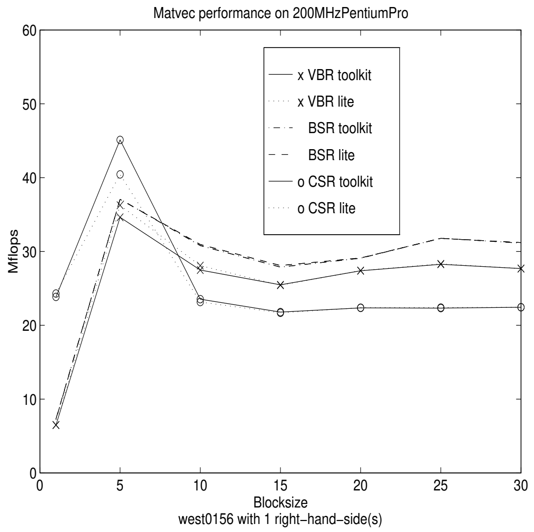


Figure 19: Pentium Pro results for WEST0156 with 1 right-hand-side

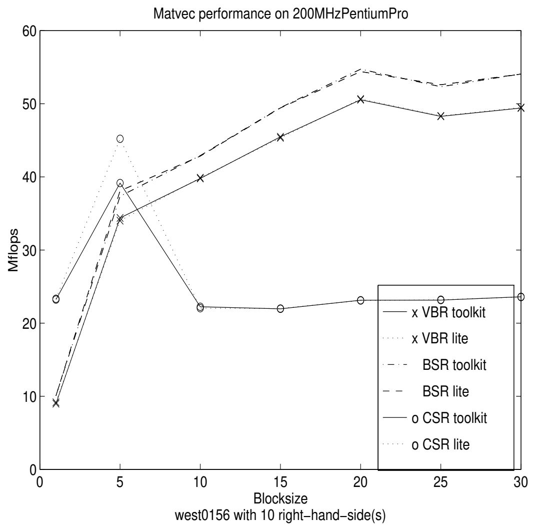


Figure 20: Pentium Pro results for WEST0156 with 10 right-hand-sides

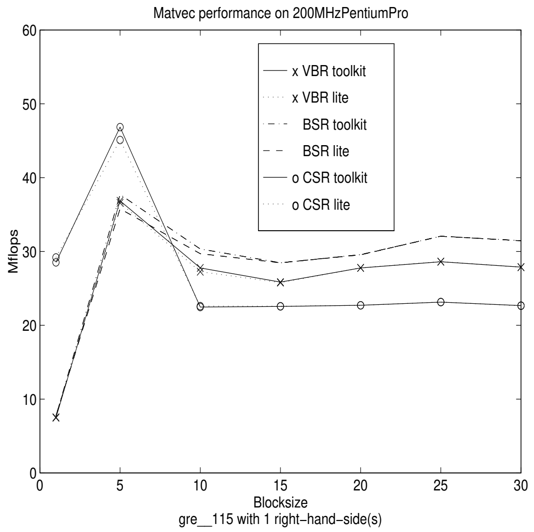


Figure 21: Pentium Pro results for GRE 115 with 1 right-hand-side

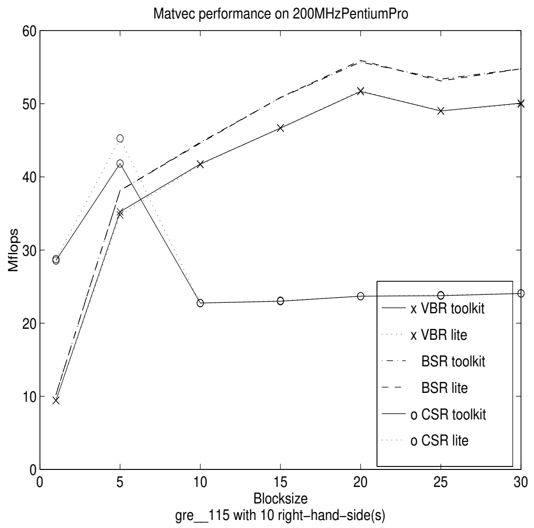


Figure 22: Pentium Pro results for GRE 115 with 10 right-hand-sides

C Sample Sparse BLAS Manual Pages

Name dbcomm

Calling Sequence

```
void  dbcomm( const int transa, const int mb, const int n, const int kb,
              const double alpha, const int descra[], const double val[],
              const int bindx[], const int bjndx[], const int bnnz,
              const int lb, const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Block coordinate format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int mb	Number of block rows in matrix A
int n	Number of columns in matrix c
int kb	Number of block columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base

	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *bindx	integer array of length bnnz consisting of the block row indices of the entries of A.
int *bjndx	integer array of length bnnz consisting of the block column indices of the entries of A.
int bnnz	number of block entries
int lb	dimension of blocks
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dbscmm

Calling Sequence

```
void  dbscmm( const int transa, const int mb, const int n, const int kb,
              const double alpha, const int descra[], const double val[],
              const int bindx[], const int bpntrb[], const int bpntre[],
              const int lb, const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Block sparse column format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int mb	Number of block rows in matrix A
int n	Number of columns in matrix c
int kb	Number of block columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base

	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *bindx	integer array of length bnnz consisting of the block row indices of the entries of A.
int *bpntrb	integer array of length mb such that bpntrb(i)-bpntrb(1) points to location in bindx of the first block entry of the j-th column of A.
int *bpntre	integer array of length mb such that bpntre(i)-bpntrb(1) points to location in bindx of the last block entry of the j-th column of A.
int bnnz	number of block entries
int lb	dimension of blocks
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dbscsm

Calling Sequence

```
void  dbscsm( const int transa, const int mb, const int n,
              const int unitd, const double dv[],
              const double alpha, const int descra[], const double val[],
              const int bindx[], const int bpntrb[], const int bpntre[],
              const int lb, const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Block sparse column format triangular solve.

$$C \leftarrow \alpha D A^{-1} B + \beta C \qquad C \leftarrow \alpha D A^{-T} B + \beta C$$

$$C \leftarrow \alpha A^{-1} D B + \beta C \qquad C \leftarrow \alpha A^{-T} D B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int mb	Number of block rows in matrix A
int n	Number of columns in matrix c
int unitd	Type of scaling: 1 : Identity matrix (argument dv[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type

	0 : non-unit
	1 : unit
	descra[3] Array base
	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *bindx	integer array of length bnnz consisting of the block row indices of the entries of A.
int *bpntrb	integer array of length mb such that bpntrb(i)-bpntrb(1) points to location in bindx of the first block entry of the j-th column of A.
int *bpntre	integer array of length mb such that bpntre(i)-bpntrb(1) points to location in bindx of the last block entry of

¿the j-th column of A.

int bnnz	number of block entries
int lb	dimension of blocks
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dbsrmm

Calling Sequence

```
void  dbsrmm( const int transa, const int mb, const int n, const int kb,
              const double alpha, const int descra[], const double val[],
              const int bindx[], const int bpntrb[], const int bpntre[],
              const int lb, const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Block sparse row format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int mb	Number of block rows in matrix A
int n	Number of columns in matrix c
int kb	Number of block columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base

	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *bindx	integer array of length bnnz consisting of the block column indices of the entries of A.
int *bpntrb	integer array of length mb such that bpntrb(i)-bpntrb(1) points to location in bindx of the first block entry of the j-th row of A.
int *bpntre	integer array of length mb such that bpntre(i)-bpntrb(1) points to location in bindx of the last block entry of the j-th row of A.
int bnnz	number of block entries
int lb	dimension of blocks
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dbsrsm

Calling Sequence

```
void  dbsrsm( const int transa, const int mb, const int n,
              const int unitd, const double dv[],
              const double alpha, const int descra[], const double val[],
              const int bindx[], const int bpntrb[], const int bpntre[],
              const int lb, const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Block sparse row format triangular solve.

$$C \leftarrow \alpha D A^{-1} B + \beta C \qquad C \leftarrow \alpha D A^{-T} B + \beta C$$

$$C \leftarrow \alpha A^{-1} D B + \beta C \qquad C \leftarrow \alpha A^{-T} D B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int mb	Number of block rows in matrix A
int n	Number of columns in matrix c
int unitd	Type of scaling: 1 : Identity matrix (argument dv[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type

	0 : non-unit
	1 : unit
	descra[3] Array base
	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *bindx	integer array of length bnnz consisting of the block column indices of the entries of A.
int *bpntrb	integer array of length mb such that bpntrb(i)-bpntrb(1) points to location in bindx of the first block entry of the j-th row of A.
int *bpntre	integer array of length mb such that bpntre(i)-bpntrb(1) points to location in bindx of the last block entry of the j-th row of A.
int bnnz	number of block entries
int lb	dimension of blocks
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dcoomm

Calling Sequence

```
void  dcoomm( const int transa, const int m, const int n, const int k,
              const double alpha, const int descra[], const double val[],
              const int indx[], const int jndx[], const int nnz,
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Coordinate format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int m	Number of rows in matrix A
int n	Number of columns in matrix c
int k	Number of columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base

	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length nnz containing row indices
int *jndx	integer array of length nnz containing column indices
int nnz	Number of nonzero matrix entries
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)

Name dcscmm

Calling Sequence

```
void  dcscmm( const int transa, const int m, const int n, const int k,
              const double alpha, const int descra[], const double val[],
              const int indx[], const int pntrb[], const int pntre[],
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Compressed sparse column format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int m	Number of rows in matrix A
int n	Number of columns in matrix c
int k	Number of columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base

	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length nnz containing row indices
int *pntrb	integer array of length k such that pntrb(j)-pntrb(1) points to location in val of the first nonzero element in column j
int *pntre	integer array of length k such that pntre(j)-pntrb(1) points to location in val of the last nonzero element in column j
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)

Name dcscsm

Calling Sequence

```
void  dcscsm( const int transa, const int m, const int n,
              const int unitd, const double dv[],
              const double alpha, const int descra[], const double val[],
              const int indx[], const int pntrb[], const int pntre[],
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Compressed sparse column format triangular solve.

$$C \leftarrow \alpha D A^{-1} B + \beta C \qquad C \leftarrow \alpha D A^{-T} B + \beta C$$

$$C \leftarrow \alpha A^{-1} D B + \beta C \qquad C \leftarrow \alpha A^{-T} D B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int m	Number of rows in matrix A
int n	Number of columns in matrix c
int unitd	Type of scaling: 1 : Identity matrix (argument dv[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type

	0 : non-unit
	1 : unit
	descra[3] Array base
	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length nnz containing row indices
int *pntrb	integer array of length k such that pntrb(j)-pntrb(1) points to location in val of the first nonzero element in column j
int *pntre	integer array of length k such that pntre(j)-pntrb(1) points to location in val of the last nonzero element in column j
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dcsrmm

Calling Sequence

```
void  dcsrmm( const int transa, const int m, const int n, const int k,
              const double alpha, const int descra[], const double val[],
              const int indx[], const int pntrb[], const int pntre[],
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Compressed sparse row format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int m	Number of rows in matrix A
int n	Number of columns in matrix c
int k	Number of columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base

	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length nnz containing column indices
int *pntrb	integer array of length k such that pntrb(j)-pntrb(1) points to location in val of the first nonzero element in row j
int *pntre	integer array of length k such that pntre(j)-pntrb(1) points to location in val of the last nonzero element in row j
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)

Name dcsrsm

Calling Sequence

```
void  dcsrsm( const int transa, const int m, const int n,
              const int unitd, const double dv[],
              const double alpha, const int descra[], const double val[],
              const int indx[], const int pntrb[], const int pntre[],
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Compressed sparse row format triangular solve.

$$C \leftarrow \alpha D A^{-1} B + \beta C \qquad C \leftarrow \alpha D A^{-T} B + \beta C$$

$$C \leftarrow \alpha A^{-1} D B + \beta C \qquad C \leftarrow \alpha A^{-T} D B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int m	Number of rows in matrix A
int n	Number of columns in matrix c
int unitd	Type of scaling: 1 : Identity matrix (argument dv[] is ignored) 2 : Scale on left (row scaling) 3 : Scale on right (column scaling)
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type

	0 : non-unit
	1 : unit
	descra[3] Array base
	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indices (not currently supported)
	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length nnz containing column indices
int *pntrb	integer array of length k such that pntrb(j)-pntrb(1) points to location in val of the first nonzero element in row j
int *pntre	integer array of length k such that pntre(j)-pntrb(1) points to location in val of the last nonzero element in row j
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least max(m,n)
int lwork	length of work array

Name dvbrmm

Calling Sequence

```
void  dvbrmm( const int transa, const int mb, const int n, const int kb,
              const double alpha, const int descra[], const double val[],
              const int indx[], const int bindx[],
              const int rpntr[], const int cpntr[],
              const int bpntrb[], const int bpntre[],
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Variable block row format matrix-matrix multiply.

$$C \leftarrow \alpha AB + \beta C$$

$$C \leftarrow \alpha A^T B + \beta C$$

Arguments

int transa	Indicates how to operate with the sparse matrix 0 : operate with matrix 1 : operate with transpose matrix
int mb	Number of block rows in matrix A
int n	Number of columns in matrix c
int kb	Number of block columns in matrix A
double alpha	Scalar parameter
double beta	Scalar parameter
int descra[]	Descriptor argument. Nine element integer array: descra[0] matrix structure 0 : general 1 : symmetric 2 : Hermitian 3 : Triangular 4 : Skew(Anti)-Symmetric 5 : Diagonal descra[1] upper/lower triangular indicator 1 : lower 2 : upper descra[2] main diagonal type 0 : non-unit

	1 : unit
	descra[3] Array base
	0 : C/C++ compatible
	1 : Fortran compatible
	descra[4] repeated indice (not currently supported)
s	0 : unknown
	1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length bnnz+1 such that the i-th element of indx[] points to the location in val of the (1,1) element of the i-th block entry.
int *bindx	integer array of length bnnz consisting of the block column indices of the entries of A.
int *rpntr	integer array of length mb+1 such that rpntr(i)-rpntr(1) is the row index of the first point row in the i-th block row. rpntr(mb+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).
int *cpntr	integer array of length kb+1 such that cpntr(j)-cpntr(1) is the column index of the first point column in the j-th block column. cpntr(kb+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).
int *bpntrb	integer array of length mb such that bpntrb(i)-bpntrb(1) points to location in bindx of the first block entry of the j-th row of A.
int *bpntre	integer array of length mb such that bpntre(i)-bpntrb(1) points to location in bindx of the last block entry of the j-th row of A.
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least m*n + max(blocksize)
int lwork	length of work array

Name dvbrsm

Calling Sequence

```
void  dvbrsm( const int transa, const int mb, const int n,
              const int unitd, const double dv[],
              const double alpha, const int descra[], const double val[],
              const int indx[], const int bindx[], const int rpntr[],
              const int cpntr[], const int bpntrb[], const int bpntre[],
              const double b[], const int ldb,
              const double beta, double c[], const int ldc,
              double work[], const int lwork);
```

Functionality Variable block row format triangular solve.

$$C \leftarrow \alpha D A^{-1} B + \beta C \qquad C \leftarrow \alpha D A^{-T} B + \beta C$$

$$C \leftarrow \alpha A^{-1} D B + \beta C \qquad C \leftarrow \alpha A^{-T} D B + \beta C$$

Arguments

int transa — Indicates how to operate with the sparse matrix
0 : operate with matrix
1 : operate with transpose matrix

int mb — Number of block rows in matrix A

int n — Number of columns in matrix c

int unitd — Type of scaling:
- 1 : Identity matrix (argument dv[] is ignored)
- 2 : Scale on left (row scaling)
- 3 : Scale on right (column scaling)

double alpha — Scalar parameter

double beta — Scalar parameter

int descra[] — Descriptor argument. Nine element integer array:
descra[0] matrix structure
- 0 : general
- 1 : symmetric
- 2 : Hermitian
- 3 : Triangular
- 4 : Skew(Anti)-Symmetric
- 5 : Diagonal

descra[1] upper/lower triangular indicator
- 1 : lower
- 2 : upper

	descra[2] main diagonal type 0 : non-unit 1 : unit descra[3] Array base 0 : C/C++ compatible 1 : Fortran compatible descra[4] repeated indices (not currently supported) 0 : unknown 1 : no repeated indices
double *val	scalar array of length nnz containing matrix entries
int *indx	integer array of length bnnz+1 such that the i-th element of indx[] points to the location in val of the (1,1) element of the i-th block entry.
int *bindx	integer array of length bnnz consisting of the block column indices of the entries of A.
int *rpntr	integer array of length mb+1 such that rpntr(i)-rpntr(1) is the row index of the first point row in the i-th block row. rpntr(mb+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).
int *cpntr	integer array of length kb+1 such that cpntr(j)-cpntr(1) is the column index of the first point column in the j-th block column. cpntr(kb+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).
int *bpntrb	integer array of length mb such that bpntrb(i)-bpntrb(1) points to location in bindx of the first block entry of the j-th row of A.
int *bpntre	integer array of length mb such that bpntre(i)-bpntrb(1) points to location in bindx of the last block entry of the j-th row of A.
double *b	rectangular array with leading dimension ldb
int ldb	leading dimension of b
double *c	rectangular array with leading dimension ldc
int ldc	leading dimension of c
double *work	scratch array of length lwork. lwork should be at least m*n + max(blocksize)
int lwork	length of work array

D References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM Pub., 1992.

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

[3] Richard Barrett, Michael Berry, Tony F. Cahn, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Rharles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[4] 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.

[5] 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.

[6] J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide.* SIAM Pub., 1979.

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[8] J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14, 1988.

[9] 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.

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

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

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

[13] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM J. Mat. Anal.*, 13(1):333–356, January 1992.

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

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

[16] Mark T. Jones and Paul E. Plassmann. BlockSolve v1.1: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-92/46, Mathematics and Computer Science Division, Argonne National Laboratory, December 1992.

[17] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5, 1979.

[18] Thomas C. Oppe and David R. Kincaid. Are there iterative BLAS? Technical report, Center for Numerical Analysis, The University of Texas at Austin, February 1990.

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

[20] Roldan Pozo and Karin Remington. NIST Sparse BLAS Implementation Nodes National Institute of Standards and Technology, Technical Report, `math.nist.gov/spblas/`, 1995.

[21] Roldan Pozo and Karin Remington. NIST Sparse BLAS (source code) National Institute of Standards and Technology, `http://math.nist.gov/spblas/`, 1995.

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

[23] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer–Verlag, New York, second edition, 1976.

[24] Barry F. Smith and William D. Gropp. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems, 1993. Submitted to SIAM J. Sci Comput.