DSCPACK: Domain-Separator Codes For The Parallel Solution Of Sparse Linear Systems *

Padma Raghavan †

1 Introduction

DSCPACK can be used to solve sparse linear systems using direct methods on multiprocessors and networks-of-workstations. This package is suitable for systems of the form \( Ax = b \) where the coefficient matrix \( A \) is symmetric and sparse. DSCPACK provides a variety of sparsity preserving (fill-reducing) orderings and computes either an \( L L^T \) (Cholesky) or an \( L D L^T \) factorization of \( A \). An important feature is the capability to perform latency-tolerant triangular solution (using the computed factors) on multiprocessors and networks of workstations. Many applications need a sequence of triangular solutions after a single factorization; such triangular solution using substitution schemes can be inefficient because of large inter-processor communication latencies. This solver uses the selective-inversion method to speed up this step; this solver is based partially on results reported in [4, 6, 7, 8]. This solver is written in C; it uses MPI for inter-processor communication [2, 3] and the BLAS library for improved cache-performance [1, 5].

2 Design Overview

The implementation is based on the idea of partitioning the sparse matrix \( A \) into domains and separators. In the graph representation of \( A \), vertices correspond to columns (or rows) of \( A \) while edges correspond to nonzeroes in the matrix. In the graph model, domains are typically large connected components and the separators are sets of vertices through which pass all paths connecting two or more domains. Such a partitioning allows effective utilization of sparsity and parallelism. Each domain is mapped to a processor and forms a leaf of a certain computational tree; separators are represented by interior nodes in the tree. The separator at the root is mapped to all processors, and the next two separators are

---

*This work was supported in part through grants ACL-97-21361, CCR-00-75792, and ACL-01-02537 from the National Science Foundation and the Maria Goeppert Mayer Award from Argonne National Laboratory and the Department of Energy.

† Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802-6106. E-mail: raghavan@cse.psu.edu.
mapped to two disjoint processor subsets each containing half the total number of processors. This mapping is applied recursively until each leaf (domain) is mapped to a single processor. Such a mapping also serves as data-partitioning of \( A \) among processors; separators and domains contain vertices which correspond to columns which thus get mapped to processors.

Domain related computations are often referred to as \textit{local phase} computations while separator related computations are \textit{distributed phase} computations. The former proceed independently on each processor with no inter-processor communication and are a consequence of functional or task parallelism resulting from the sparsity. The distributed phase represents \textit{data parallel} computation that typically requires substantial inter-processor communication.

The solution process has four major steps (i) ordering to determine a fill-reducing numbering, (ii) symbolic factorization to determine the zero-nonzero structure of the factor, (iii) numeric factorization to compute the factor, and (iv) triangular solution to solve for a given right hand side vector using the factors. When more than one processor is used, each of these steps has both a local and a distributed phase for the underlying computation.

\textbf{Referencing DSCPACK.} Users should cite this solver using references [9, 10] in any resulting publication, presentation or software.

\textbf{Communication and Numerical Libraries.} The solver uses MPI [2, 3] and BLAS [1, 5] libraries. Using BLAS libraries increases the storage requirement during numeric factorization by a factor of \( 1.2—2 \) while improving execution rates. This increase occurs because symmetry cannot be used to store only one half of the many smaller dense matrices that occur within the sparse factor. The interface functions of DSCPACK can accept options that specify the level of BLAS to be used. For example, \texttt{DSCBLAS1} and \texttt{DSCDBLAS1} avoid the storage expense at the cost of using slower level-1 routines for both local and distributed phase computations during numeric factorization. Using options \texttt{DSCBLAS3} and \texttt{DSCDBLAS2} leads to level-3 in the local-phase and level-2 in the distributed phase.

\textbf{Memory Requirements and Performance.} In DSCPACK, the \textit{sparsity structure} of \( A \) is replicated on each processor using a compact representation. The nonzero values are not replicated; instead they are partitioned among the processors. Using \( P \) processors, the solver will typically require, at each processor, memory proportional to \( (1/P) \) times the size of \( L \). The selective inversion feature of the solver requires no more storage than for the traditional form, i.e., selective inversion is done in place. This feature should lead to better performance if a single factorization is to be followed by a sequence of triangular solutions using different right hand side vectors.

The solver can use either single or double precision floating point arithmetic (floats or doubles in C). The choice is made while compiling the library. Within the code, the type \texttt{RealNumberType} stands for the precision of the data used (float or double). On typical 32-bit architectures, single precision will be approximately fifty percent faster than double precision with half the peak memory usage. The compiler flag \texttt{DBL_RNUM} gives a double precision version; without this flag the data type defaults to single precision (float in C).
Memory Allocation and Deallocation Rules. If the application allocates a vector, it should be freed by the application; the solver will use the vector but it will not modify the contents or deallocate memory. Likewise any vectors made available to the application by a solver function should not be freed or modified by the application.

Function Prototypes and Error Codes. The calling application will include “dsmain.h” which contains solver specific definitions and prototypes for interface functions. Each function will be called at every processor involved in the solve unless noted otherwise. Functions returning a single integer data value are returning an error code unless noted otherwise. This returned value should equal DSC_NO_ERROR if there are no errors; errors can be displayed using the DSC_ErrorDisplay function.

Limitations of Version 1.0. There are two main limitations Version 1.0. First, the distributed phase of numeric factorization does not provide the option to use BLAS level 3; currently, BLAS level 2 is the highest level supported for this stage. Second, the $LDL^T$ factorization does not do pivoting; consequently the solution process could fail for some indefinite systems. These two limitations will be addressed in the next release.

2.1 Sparse Matrix Representation

Consider specifying the sparse matrix $A$. Row and column numbers and vector indices start at 0 (C-style). The term column is used interchangeably with equation and denotes a specific column of the matrix (equation in the system). The matrix is stored in a column by column manner. The term structure refers to the connectivity and not actual nonzero values. Furthermore, the connectivity is specified in a very compact form. The motivation for the compact connectivity (structure) representation is from matrices from finite-difference formulations. In a finite-difference mesh, a node is typically connected to several neighbors. This node is associated with a number of solution components (columns/equations). All the components at each node are connected to each other and to all components at neighboring nodes, as given by the mesh connectivity. Now the matrix connectivity can be easily derived from the mesh nodal connectivity with some auxiliary information such as the number of components at each node, and their numbering. Our representation allows the structure to be very compact while specifying the positions of nonzero elements of the actual matrix.

We explain our representation using an example in the next section. Other notation and conventions include the following.

- The term GlobalNS refers to the total number of distinct structures. Structure numbers start at 0 and end at GlobalNS -1.

- Each structure can have several equations (columns) associated with it; this is the replication of the structure. If Replicates[i] = 6 then the $i$-th structure is associated with 6 equations.
• The \textit{equations} (columns) for a given \textit{structure} are numbered consecutively; if the 0-th sparsity structure has $\text{Replicates}[0] = 6$, then it denotes the structure of columns(equations) $0, \ldots, 5$.

• The numeric values needed at a processor (after ordering) correspond to the \textit{symmetric lower half}, including the \textit{diagonal}, of columns assigned to the processor.

• The structure has the \textit{full-connectivity} (not just the symmetric half). It is stored in sparse form using a 3-vector representation $\text{Replicates}$, $\text{AIndex}$, and $\text{AStruct}$.

• The matrix nonzero values can be stored in another vector $\text{ANonz}$. Each element in this vector corresponds to a nonzero value at a specific row and column position in the sparse matrix. The row and column position can be computed using the information in $\text{GlobalNS}$, $\text{Replicates}$, $\text{AIndex}$, and $\text{AStruct}$ as shown in the next section.

2.2 Structure and Matrix Values

We discuss the compact structure representation and its relation to the sparse matrix using a small example shown in Figure 1. This example is also used in the rest of this document to explain various interface function calls.

\[
\begin{array}{ccc}
(12-13) & (14-15) & (16-17) \\
<6> & ----- & <7> & ----- & <8> \\
| & | & | \\
| & | & | \\
|(6-7) & |(8-9) & |(10-11) \\
<3> & ----- & <4> & ----- & <5> \\
| & | & | \\
| & | & | \\
|(0-1) & |(2-3) & |(4-5) \\
<0> & ----- & <1> & ----- & <2>
\end{array}
\]

Figure 1: An example to illustrate the relationship between structure and matrix using a 3x3 grid with 2 replicates per structure. Sparsity structure numbers are enclosed in <> and column (equation) numbers are enclosed in ()

Using our notation, the value of $\text{GlobalNS}$ for the example is 9. The vector $\text{Replicates}$ has 9 elements in positions 0, $\cdots$, 8 each equal to 2. If the
nodal replication (replication at each structure) were different, then this vector would have different values for different structure numbers. The vector AIndex has GlobalNS + 1 elements; the neighbors of structure \( i \) are given in positions AIndex\([i]\), ..., (AIndex\([i+1]\) - 1) of the vector AStruct. The elements of AIndex in positions 0, ..., 9 are: 0, 3, 7, 10, 14, 19, 23, 26, 30, 33. The elements of AStruct in positions 0, ..., 33 are:

\[
0, 1, 3, 0, 1, 2, 4, 1, 2, 5, 0, 3, 4, 6, 1, 3, 4, 5, 7, 2, 4, 5, 8, 3, 6, 7, 4, 6, 7, 8, 5, 7, 8. 
\]

For example, the connectivity of structure numbered 0 is given in AIndex\([0]\) = 0 through AIndex\([1]\) - 1 = 2 of AStruct as 0, 1, 3. For the code to work correctly, a structure must always be connected to itself. Furthermore, the value of Replicates cannot be less than one. DSCPACK also assumes that values in AStruct for a given structure are listed in increasing order. For example, the connectivity of structure numbered 0 given in AIndex\([0]\) = 0 through AIndex\([1]\) - 1 = 2 of AStruct as 3, 1, 0 is incorrect; its should be ordered as 0, 1, 3.

The associated sparse matrix has 18 columns and its sparsity structure can be obtained using AIndex, AStruct, and Replicates. The convention is that all columns (equations) associated with a structure are numbered consecutively. Consider columns associated with structure \( j \); let \( k = \sum_{i=0}^{j-1} \text{Replicates}[i] \). Now columns numbered \( k, k + 1, \ldots, k + \text{Replicates}[j] - 1 \) are associated with structure \( j \). Let ColumnOfStruct be a vector of size GlobalNS such that each element contains the smallest column number associated with the corresponding structure. This vector can be computed as follows using a simple prefix operation.

\[
\text{ColumnOfStruct}[0] = 0; \\
\text{for} (i=1; i < \text{GlobalNS}; i++) \{
\quad \text{ColumnOfStruct}[i] = \text{ColumnOfStruct}[i-1] + \text{Replicates}[i-1];
\}
\]

The subscripts of nonzeroes in a given column of the matrix can be easily ascertaining using the ColumnOfStruct information and the structure in AStruct. The convention is that if structure \( p \) is connected to structure \( q \) then all columns of \( p \) are connected to all columns of \( q \). For the example, the connectivity of structure numbered 0 is given in AIndex\([0]\) = 0 through AIndex\([1]\) - 1 = 2 of AStruct as 0, 1, 3. This means that associated columns 0, 1 have nonzeroes in rows 0, 1, 2, 3, 6, 7.

Consider a representation where the nonzeroes of the entire matrix (upper and lower halves and diagonal) are given in a vector ANonz. Then its initial 12 locations correspond to nonzeroes in columns 0 and 1 and contain:

\[
A_{0,0}, A_{1,0}, A_{2,0}, A_{3,0}, A_{6,0}, A_{7,0}, A_{0,1}, A_{1,1}, A_{2,1}, A_{3,1}, A_{6,1}, A_{7,1}. 
\]

An alternative representation is one such that only nonzeroes in the symmetric lower half (including the diagonal) are given in a vector ANonz. Then its initial
11 locations contain nonzeros in the diagonal and lower half of columns 0 and 1: \( A_{0,0}, A_{1,0}, A_{2,0}, A_{3,0}, A_{6,0}, A_{7,0}, A_{1,1}, A_{2,1}, A_{3,1}, A_{6,1}, A_{7,1} \).

If nonzeros in the entire matrix are stored in a vector \texttt{ANonz}, then the row and column (subscripts) of each nonzero can be computed as follows.

```c
/* this example displays elements of the matrix in
 row, column, nonzero form; ANonz contains
 nonzeros in the entire matrix.
*/
index = 0;
for (i=0; i < GlobalNS; i++) {
    column=ColumnOfStruct[i];
    for (; column <ColumnOfStruct[i]+ Replicates[i]; column++){
        for (j= AIndex[i]; j < AIndex[i+1]; j++) {
            k = AStruct[j];
            row=ColumnOfStruct[k];
            for (; row <ColumnOfStruct[k]+ Replicates[k]; row++){
                printf("row %d, column %d, value %e\n", row, column, ANonz[index]);
                index++;
            }
        }
    }
}
```

The code fragment shown above can be easily modified for the alternative representation when only nonzeros in the \textit{symmetric lower half} (including the diagonal) are given in the vector \texttt{ANonz}. The \texttt{printf} and \texttt{index} increment statements should be executed if \texttt{row} is greater than or equal to \texttt{column}.
3 Installation and Usage Examples

DSC PACK Version 1.0 is distributed as a gzipped tar file that opens into a directory DSCPACK1.0. This directory contains subdirectories Doc containing this report, Data containing matrix data files for use with example drivers, and DSCLIB containing the main solver library. This directory also contains six example codes example1.c through example6.c, and a file named utilities.c. The latter contains utility functions that are not part of the solver but are used in the example drivers (see Section 4). Examples one through four generate test systems associated with the model five-point finite-difference grids; the remaining examples use data provided in the Data directory. A sample Makefile is provided in DSCLIB to compile the library; this file should be edited to provide locations of MPI include files. On successful completion of make for double-precision data, a library named dsclibh1.a is produced in that directory. This library must be linked with the application code. The example codes in main directory also link to this library. A sample Makefile is provided for compiling the examples; this Makefile must be edited to provide paths for MPI and BLAS libraries and include files. The examples compile to executables named Solve1 through Solve6. These executables can be run on the command line using mpi run and some command line parameters as indicated in the Readme file. A brief description of each example code is as follows.

Solve1 (example1.c) It generates a model grid problem, factors and solves for several right hand side vectors. It repeats the numeric factorization and triangular solution steps to simulate solving several systems with the same sparsity structure of the matrix but different numeric values.

Solve2 (example2.c) Similar to Solve1; additionally, it repeats the entire process a user specified number of times. This simulates solving several sparse linear systems; for each system, several matrices with the same structure but different nonzeroes are factored and used for repeated triangular solution.

Solve3 (example3.c) This is very similar to Solve 2. The main difference is in the choice of the DSCopen function type, see Sections 4.4 and 4.5.

Solve4 (example4.c) This is similar to Solve 4 but differs in the use of auxiliary functions (see Section 4).

Solve5 (example5.c) This uses sparse matrix data in files provided in the directory Data. It only uses the structure information and generates the numeric values of a symmetric positive definite matrix to match. It shows how to use auxiliary functions in Section 4.

Solve6 (example6.c) This differs from Solve5 in that it uses both the structure and numeric data from the sparse matrix data files.

We next show two sample calling sequences where the interface function names are provided without full details of associated parameters.
Example 1. Perhaps the most common usage form is to factor a sparse matrix
A and use the factors to solve for a number of different right hand side vectors.
The calling sequence is as follows.
Call sequence for a single factorization followed by many triangular solution steps.
MyDSCObject = DSC_Begin(); get a solver object MyDSCObject
error_code = DSC_Open0 (MyDSCObject, NumberProcessors, ...); set MPI group
error_code = DSC_Order (MyDSCObject, ...); input structure and order
error_code = DSC_SFActor (MyDSCObject, ...); symbolic factor

place code here to use information returned by the last two functions
to provide in MyANonz: the nonzeroes in columns mapped to the processor

error_code = DSC_NFactor (MyDSCObject, MyANonz, ...); numeric factor
for
(as many RHS vectors as needed) {
    error_code = DSC_input...Rhs(MyDSCObject, ...); enter RHS vector
    error_code = DSC_Solve(MyDSCObject); solve
    error_code = DSC_Get...Solution(MyDSCObject, ...); extract solution
}
} end repeated triangular solution

DSC_FreeAll (MyDSCObject); free internal data in solve object
DSC_Close0 (MyDSCObject); close MPI groups for this solver instance
DSC_End (MyDSCObject); release solver object

Example 2. With a slight variation to the last calling sequence, the application
can repeatedly factor many sparse matrices with the same sparsity structure
but different nonzero elements. This allows reuse of the ordering and symbolic
steps for a sequence of numeric factorization steps. The changes to the calling
sequence are given below.

Call sequence for a single symbolic phase followed by many numeric factorization
steps for matrices with the same sparsity structure but different nonzero values.
Each numeric factorization is followed by a number of triangular solution steps.
... as in Example 1
error_code = DSC_SFActor (MyDSCObject, ...); symbolic factor
for
(as many different nonzero values of the matrix as needed) {
    place code here to provide matrix nonzero data
    error_code = DSC_NFactor (MyDSCObject, MyANonz, ...);
    for
        (as many RHS vectors as needed) {
            ... as in Example 1
        }
} end repeated triangular solution
if
    (another factorization is needed)
    DSC_RerFactorInitialize(MyDSCObject);
} end repeated factorization
... as in Example 1
4 Interface and Auxiliary Functions

The interface functions begin with the DSC prefix and described in detail in this section. They are listed in a natural calling sequence order and not in alphabetical order. This calling sequence order allows a simpler description of each function. We also provide a few auxiliary functions that are not part of the main solver package but are provided to help the application developer.

Recall that function prototypes and solver specific definitions are provided in the header file “dscmain.h”. Each function should be called \textit{at every processor involved in the solve} unless noted otherwise. Functions returning a single integer data value are returning an error code unless noted otherwise. This returned value should equal DSC\_NO\_ERROR if there are no errors; errors can be displayed using the DSC\_ErrorDisplay function.

4.1

\textbf{int DSC\_RemoveNullStructs(int \*GlobalNS, int \*AIndex, int \*AStruct, int \*Replicates)}

This function is not an essential part of the solver but it could be of use to the application before it sets up the solver. If the structure generated by the application has zero values in the vector Replicates, then it is not in the correct form for DSCPACK. This function will change the data to conform to DSCPACK conventions. The function changes GlobalNS, and the vectors AIndex, AStruct, and Replicates to remove structures with replication 0. It should be called before any other calls to the solver if the structure has any zero Replicates.

4.2

\textbf{int DSC\_Analyze(int \*GlobalNS, int \*AIndex, int \*AStruct, int \*Replicates)}

The solver can use processor subsets whose sizes are powers of two. Furthermore, the largest number of processors that can be potentially used depends on the sparse matrix. It is based on the largest number of domains (of size greater than zero) into which the matrix can be partitioned. This function returns the largest number of processors possible ( ≤ 512) for the solver. For example, if the value returned is 8, the problem can be solved on 1, 2, 4 and 8 processors. This value can be used to start the actual solver with an appropriate number of processors. If needed, DSC\_RemoveNullStructs must be called before this function.

4.3

\textbf{DSC\_Solver DSC\_Begin(void)}

This is first true call to the solver. It creates and returns a DSC\_Solver object; this object is passed to all subsequent solver function calls. Henceforth, we
refer to the returned object as \texttt{DSC::Solver MyDSCObject}, and use it as the first parameter in subsequent solver interface function calls.

4.4

\texttt{void DSC::Open0} (\texttt{DSC::Solver MyDSCObject}, \texttt{int NumberProcessors}, \texttt{int *MyRank, MPI_Comm Comm})

DSC PACK is set up to use its own MPI context and communicator. The input parameter \texttt{MyDSCObject} is the object returned by the preceding call to \texttt{DSC::Begin}. The parameter \texttt{NumberProcessors} (input) is the number of processors for the solver. The parameter \texttt{Comm} (input) is the communicator being used currently within the application in the group of processors making this call. More than \texttt{NumberProcessors} (processors) can make this call; the processors with ranks 0, \ldots, \texttt{NumberProcessors-1} in the calling group will form a new MPI group with a new context for the solver. Upon return, \texttt{MyRank} (output) will contain the rank of this processor in the solver processor group; it is -1 if the processor is not in solver group. Recall that \texttt{NumberProcessors} must be a power of two and no more than 512 processors. Additionally, it cannot be any larger than the value returned by a preceding call to \texttt{DSC::Analyze}.

4.5

\texttt{void DSC::Open1} (\texttt{DSC::Solver MyDSCObject}, \texttt{int NumberProcessors}, \texttt{int *ProcessorList, int *MyRank, MPI_Comm Comm})

This version of the open call allows the application to select the processors to be used in the solver group; this can be useful because the application may be able to identify lightly loaded processors that it would prefer be used by the solver. Its parameters and requirements are the same as for \texttt{DSC::Open0} except that now a list of processors must be specified. The parameter \texttt{ProcessorList} (input) is a list of processor ranks in the \texttt{calling} group that must be used in the solver group; this list should have \texttt{NumberProcessors} distinct processor ranks. At least all the processors specified in \texttt{ProcessorList} should call this function (otherwise the function will not return). Only processors in the solver group (those with \texttt{MyRank} \neq -1) should continue with the remaining solver steps.

4.6

\texttt{int DSC::Order} (\texttt{DSC::Solver MyDSCObject}, \texttt{int OrderCode, int GlobalNS}, \texttt{int *Index, int *AStruct, int *Replicates, int *NumGlobalCols}, \texttt{int *NumLocalStructs, int *NumLocalCols, int *NumLocalNonz}, \texttt{int **GlobalStructNewColNum, int **GlobalStructNewNum, int **GlobalStructOwner, int **LocalStructOldNum})

As customary, \texttt{MyDSCObject} (input) is the first parameter; recall that it is the object returned by an earlier call to \texttt{DSC::Begin}. The parameter \texttt{OrderCode} can
be one of 1, 2 or 3; 1 specifies a pure nested dissection ordering while 2 (3) is a hybrid ordering with minimum degree (minimum deficiency) for the local domains. The sparse structure is specified (input) using GlobalNS, AIndex, AStruct and Replicates. These input parameters are left unchanged. The function computes an ordering and a data mapping and conveys them through the following output parameters. These output parameters should not be changed by the calling routine nor freed.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>* NumGlobalCols</td>
<td>Number of global columns (equations)</td>
</tr>
<tr>
<td>* NumLocalStructs</td>
<td>Number of structures assigned to this processor.</td>
</tr>
<tr>
<td>* NumLocalCols</td>
<td>Number of columns assigned to this processor</td>
</tr>
<tr>
<td>* NumLocalNonz</td>
<td>Number of nonzeros over all columns assigned to this processor (in lower half of reordered A)</td>
</tr>
<tr>
<td>* GlobalStructNewColNum</td>
<td>Size GlobalNS; if GlobalStructNewColNum[i] is x then the columns of structure i are renumbered x, x + 1, ... , x + Replicates[i] - 1</td>
</tr>
<tr>
<td>* GlobalStructNewNum</td>
<td>Size GlobalNS; the new numbering of structures after the fill-reducing ordering; if element i contains x then the new number of structure i is x</td>
</tr>
<tr>
<td>* GlobalStructOwner</td>
<td>Size GlobalNS; if element i contains x then this structure and all its associated columns are assigned to processor rank x in the solver group</td>
</tr>
<tr>
<td>* LocalStructOldNum</td>
<td>Size NumLocalStructs; if element i contains x then the original structure x is the i-th structure on this processor</td>
</tr>
</tbody>
</table>

The output parameters will be used by the calling code to assemble and provide the nonzeros owned by this processor in the symmetric lower half of the reordered matrix. The input vectors AIndex, AStruct and Replicates can be freed by the calling routine on return. They are not directly needed by the solver after this call, although the application may need them to set up the data for the numeric factorization and triangular solution steps. The DSC_Order call is explained using a a 3x3 grid example with 2 replicates per structure.

Using our notation, the value of GlobalNS for the example is 9. The vector Replicates has 9 elements in positions 0, ..., 8 each equal to 2. The vector AIndex has GlobalNS + 1 elements; the neighbors of structure (node) i are given in positions AIndex[i], ..., AIndex[i+1]-1 of the vector AStruct. The elements of AIndex in positions 0, ..., 9 are: 0, 3, 7, 10, 14, 19, 23, 26, 30, 33. The elements of AStruct in positions 0, ..., 33 are:

0, 1, 3, 0, 1, 2, 4, 1, 2, 5, 0, 3, 4, 6, 1, 3, 4, 5, 7, 2, 4, 5, 8, 3, 6, 7, 4, 6, 7, 8, 5, 7, 8.
<table>
<thead>
<tr>
<th>Original State</th>
<th>Processor Assignment and Renumbering after DSC_Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12-13)</td>
<td>p0(2-3)</td>
</tr>
<tr>
<td>(14-15)</td>
<td>p0(16-17)</td>
</tr>
<tr>
<td>(16-17)</td>
<td>p1(8-9)</td>
</tr>
<tr>
<td>&lt;6&gt;</td>
<td>&lt;7&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(6-7)</td>
<td>(8-9)</td>
</tr>
<tr>
<td>&lt;3&gt;</td>
<td>&lt;4&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(0-1)</td>
<td>(2-3)</td>
</tr>
<tr>
<td>&lt;0&gt;</td>
<td>&lt;1&gt;</td>
</tr>
</tbody>
</table>

Figure 2: An example to illustrate the effect of a call to DSC_Order; sparsity structure numbers are enclosed in $\langle$>, column (equation) numbers are enclosed in ($'), and processors are indicated as p0, p1.

For example, the connectivity of structure numbered 0 is given in AIndex[0] = 0 through AIndex[1]-1 = 2 of AStruct as 0, 1, 3.

The information returned by the ordering step is to be used by the application to set up a vector of nonzeros MyAlNonz, on each processor for the numeric factorization. After ordering, the structures have been given new numbers and assigned to an owner processor. The columns associated with the structures now follow the new numbering of the structures and the replicates for a given structure are numbered consecutively. Furthermore, the structures are arranged in different orders locally on each processor. This arrangement is given in the vector LocalStructOldNum. Nonzeros in associated columns (in the local order or arrangement) must be provided if they occur in the diagonal or sub-diagonal positions of the matrix with the new numbering.

Some of the output variables of DSC_Order will have different values on different processors. We show in detail the output at processor 0 (processor with rank 0 returned by a call to DSC_Open0 or DSC_Open1). Values include: *NumGlobalCols = 18, the total number of columns over all processors and in the entire matrix. The value *NumLocalStructs = 5, the number of structures assigned to this processor. The value *NumLocalCols = 10; it is the number of columns assigned to this processor. The value *NumLocalNonz = 39, indicates the number of nonzeros in columns assigned to this processors (diagonal and below). Recall that we only use nonzeros in the lower half of A after reordering for subsequent numeric factorization. The vector *GlobalStructNewNum contains 0, 6, 3, 2, 7, 5, 1, 8, 4; for example, the original structure 0 is still 0 after ordering but the original structure 1 has been given a new number of 6.

The vector *GlobalStructNewColNum, contains the smallest column number
for a given structure after ordering, i.e., 0, 12, 6, 4, 14, 10, 2, 16, 8 for the example. The vector *GlobalStructOrder* indicates the processor rank to which a given structure (and all its associated columns) have been assigned; it is 0, 0, 1, 0, 1, 1, 0, 0, 1 for the example. The vector *LocalStructOldNum* contains 0, 6, 3, 1, 7; it has as many elements as NumLocalStructs and indicates the original value of each structure prior to the ordering. The value 0 at position 0 indicates that after the ordering and mapping, the zero-th structure on processor 0 corresponds to the original structure 0; likewise the last local structure on processor 0 is the original structure numbered 7.

The application code at processor 0 must construct a vector to hold non-zeroes of reordered A (lower half and diagonal). Let this vector be *MyANonz*. Since the local structure 0 corresponds to the old structure 0, local columns 0 and 1 correspond to old columns 0 and 1. The positions 0, ..., 5 of *MyANonz* should contain in terms of the new numbers:

\[ A_{0,0}, A_{1,0}, A_{4,0}, A_{5,0}, A_{12,0}, A_{13,0}. \]

These values are the same as the following using the original column numbers:

\[ A_{0,0}, A_{1,0}, A_{6,0}, A_{7,0}, A_{2,0}, A_{3,0}. \]

Likewise, local column 1 on processor 0 corresponds to old column 1. Thus positions 6-10 of *MyANonz* should contain (in terms of the new numbers):

\[ A_{1,1}, A_{4,1}, A_{5,1}, A_{12,1}, A_{13,1}. \]

In terms of the original numbers these are the same as:

\[ A_{1,1}, A_{6,1}, A_{7,1}, A_{2,1}, A_{3,1}. \]

To continue with the example, observe that local structure 1 corresponds to old structure 6 and thus local columns 2 and 3 on processor 0 correspond to old columns 12 and 13. Thus, in terms of the new numbers positions 11, ..., 16 (column 2) and 17, ..., 21 (column 3) of *MyANonz* contain:

\[ A_{2,2}, A_{3,2}, A_{4,2}, A_{5,2}, A_{16,2}, A_{17,2}, \text{and} \ A_{3,3}, A_{4,3}, A_{5,3}, A_{16,3}, A_{17,3}. \]

In terms of the original numbers, these correspond to:

\[ A_{12,12}, A_{13,12}, A_{12,6}, A_{12,7}, A_{14,12}, A_{13,13}, \text{and} \ A_{13,13}, A_{13,3}, A_{13,7}, A_{14,13}, A_{13,13}. \]

The remaining elements of *MyANonz* on processor 0 are as follows. Local structure 2 corresponds to old structure 3 and hence columns 4 and 5 correspond to old columns 6 and 7. In terms of the new numbers, positions 22, ..., 25 (column 4) and 26, ..., 28 (column 5) of *MyANonz* contain:

\[ A_{4,4}, A_{5,4}, A_{14,4}, A_{15,4}, \text{and} \ A_{5,5}, A_{14,5}, A_{15,5}. \]
Local structure 3 corresponds to old structure 1 and hence columns 6 and 7 correspond to old columns 12 and 13. In terms of the new numbers, positions 29, ..., 32 (column 6) and 33, ..., 35 (column 7) of \texttt{MyANonz} contain:

\[ A_{12,12}, A_{13,12}, A_{14,12}, A_{15,12}, \text{ and } A_{13,13}, A_{14,13}, A_{15,13}. \]

Finally, local structure 4 corresponds to old structure 7 and hence columns 8 and 9 correspond to old columns 14 and 15. In terms of the new numbers, positions 36, 37 (column 8) and 38 (column 9) of \texttt{MyANonz} contain: \( A_{16,16}, A_{17,16}, \text{ and } A_{17,17}. \)

4.7

\begin{verbatim}
int DSC_SFActor (DSC_Solver MyDSCObject, int *TotalMemory,
int *MaxSingleBlock, int Limit, int LBLASLevel, int DBLASLevel)
\end{verbatim}

This function does symbolic factorization and estimates the amount of memory needed for factorization; it is a fair estimate but not an exact estimate down to the last byte. Set \texttt{Limit} to reflect what is available after application memory demands (excluding solver memory). The parameter \texttt{LBLASLevel} can be set to \texttt{DSC_LBLAS3}, \texttt{DSC_LBLAS2}, or \texttt{DSC_LBLAS1} (least memory required); \texttt{DBLASLevel} can be set of \texttt{DSC_DBLAS2}, or \texttt{DSC_DBLAS1} (least memory). The output parameter \texttt{TotalMemory} contains an estimate of the total memory required for the factorization step as a whole number of Mbytes. The output parameter \texttt{MaxSingleBlock} contains an estimate of the largest single block requested in a single memory allocation call (as a whole number of Mbytes). If the input parameter \texttt{Limit} is smaller than either one of the two memory estimates, error codes are set and the solver cannot proceed to numeric factorization.

4.8

\begin{verbatim}
int DSC_JNFactor (DSC_Solver MyDSCObject, int SchemeCode,
RealNumberType *MyANonz, int FactorType,
int LBLASLevel, int DBLASLevel)
\end{verbatim}

All parameters are input parameters. For \texttt{SchemeCode}, use 1 for standard factorization; with a value of 2 factorization is followed by selective inversion (to speed up repeated triangular solution). The vector \texttt{MyANonz} is of type \texttt{RealNumberType} with \texttt{NumLocalNonz} elements (as returned by \texttt{DSC_Order}). This vector contains nonzero values of \( A \) in the diagonal and subdiagonal positions in columns assigned to this processor.

For the parameter \texttt{FactorType} use \texttt{DSC_JLT} for Cholesky and \texttt{DSC_JLDLT} for the \( LDL^T \) factorization. In \texttt{LBLASLevel} indicate the level of BLAS to be used in the local phase and in \texttt{DBLASLevel} indicate level of BLAS to be used in the distributed phase. \texttt{LBLASLevel} and \texttt{DBLASLevel} must match values given earlier to \texttt{DSC_SFActor}.

The application must set up \texttt{MyANonz} using the information provided by \texttt{DSC_Order}. \texttt{MyANonz} must be freed by calling routine upon return. The data to
be provided in MyANonZ is explained using the 3X3 grid example with 2 replicates per structure. For ease of presentation, Figure 2 is shown below as Figure 3. At all processors, DSC_Order returns the same values for the following:

\[ \text{NumGlobalCols} = 18 \]

\*GlobalStructNewNum = 0, 6, 3, 2, 7, 5, 1, 8, 4

\*GlobalStructNewColNum = 0, 12, 6, 4, 14, 10, 2, 16, 8

\*GlobalStructOwner = 0, 0, 1, 0, 1, 0, 0, 1

<table>
<thead>
<tr>
<th>Original State</th>
<th>Processor Assignment and Renumbering after DSC_Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>(12-13)</td>
<td>(14-15)</td>
</tr>
<tr>
<td>&lt;6&gt; -----</td>
<td>&lt;7&gt; -----</td>
</tr>
<tr>
<td>&lt;1&gt; -----</td>
<td>&lt;8&gt; ------</td>
</tr>
<tr>
<td>&lt;0&gt; -----</td>
<td>&lt;1&gt; -----</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>8</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0(4-5)</td>
<td>p1(14-15)</td>
<td>p1(10-11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p0(0-1)</td>
<td>p0(12-13)</td>
<td>p1(6-7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: An example to illustrate setting up data for DSC_MFactor; sparsity structure numbers are enclosed in (), column (equation) numbers are enclosed in (), and processors are indicated as p0, p1.

At processor 0, DSC_Order returns \*NumLocalStructs = 5, \*NumLocalCols = 10, \*NumLocalNonz = 39, and \*LocalStructOldNum = 0, 6, 3, 1, 7. The application must construct a vector to hold nonzeros of reordered A (lower half) owned by processor 0. This vector should contain nonzeros in new columns 0, 1, 2, 3, 4, 5, 12, 13, 16 and 17 of A at and below the diagonal. It is important to note that the nonzeros must be listed within each column in increasing order of the new row numbers (row numbers after ordering).

At processor 1, DSC_Order returns \*NumLocalStructs = 4, \*NumLocalCols = 8, and \*LocalStructOldNum = 2, 8, 5, 4. The application must construct a vector to hold nonzeros of the reordered A (lower half) owned by processor 1. This vector should contain nonzeros in the new columns 6, 7, 8, 9, 10, 11, 14 and 15 of A at and below the diagonal. Once again, it is important to note that the nonzeros must be listed within each column in increasing order of the new row numbers (row numbers after ordering).
4.9

int DSC_InputRhsGlobalVec(DSC_Solver MyDSCObject, 
RealNumberType *RhsVec, int NumGlobalCols)

The parameter RhsVec (input) of size NumGlobalCols is the global right hand side vector. The function uses the global right hand side vector to pick out elements of the (local) vector needed at a processor. This is not efficient but merely for convenience to the application developer; we suggest using the local version of this function. Note that the components of RhsVec are provided in the original order; i.e., RhsVec[i] corresponds to the i-th equation in the original order. The parameter NumGlobalCols (input) is total number of equations (columns) over all processors.

4.10

int DSC_InputRhsLocalVec(DSC_Solver MyDSCObject, 
RealNumberType *MyRhsVec, int NumLocalCols)

The function sets up the right hand side vector at this processor. Now components of MyRhsVec are specified in the new order using the local numbering at each processor. The vector MyRhsVec must be allocated by the application. The parameter NumLocalCols (input) is the number of equations (columns) assigned to this processor.

For the example we have been using, recall that processor 0 has been assigned reordered columns 0, 1, 2, 3, 4, 5, 12, 13, 16 and 17. The application must construct a vector to hold corresponding values of the right hand size vector (b). In terms of new numbers, MyRhsVec on processor 0 must contain:

\[ b_0, b_1, b_2, b_3, b_4, b_5, b_{12}, b_{13}, b_{16}, b_{17}. \]

In terms of the original numbers, these components are the same as

\[ b_0, b_1, b_{12}, b_{13}, b_6, b_7, b_2, b_3, b_{14}, b_{15}. \]

On processor 1, in terms of new numbers, MyRhsVec must contain:

\[ b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{14}, b_{15}. \]

In terms of the original numbers, these components are the same as

\[ b_4, b_5, b_7, b_{17}, b_{10}, b_{11}, b_{14}, b_9. \]

4.11

int DSC_Solve(DSC_Solver MyDSCObject)

16
This function performs triangular solution using either distributed matrix-vector products or distributed substitution in the distributed phase (as specified by SchemeCode in the call to DSC_JNFactor. The local phase uses substitution. The solution remains in data structures internal to the solver.

4.12

int DSC_GetGlobalSolution(DSC_Solver MyDSCObject,
int *EqNumber, RealNumberType *SolutionVec)

The vectors SolutionVec and EqNumber (of size NumGlobalCols) must be allocated by the application. Upon completion, SolutionVec[i] contains component EqNumber[i] in the original ordering (numbering). This function is meant to be paired up with DSC_InputRhsGlobalVec. Once again, this for convenience at the application end; we recommend using the Local counterparts of these functions.

4.13

int DSC_GetLocalSolution(DSC_Solver MyDSCObject,
RealNumberType *MySolutionVec, int NumLocalCols)

The parameter NumLocalCols (input) is number of equations (columns) assigned to this processor. The application must allocate MySolutionVec (of size NumLocalCols); upon completion it contains the local components of the solution vector.

For the example used so far, MySolutionVec on processor 0 contains solution x components x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_10, x_11, x_12, x_13 using new numbers. These are the same as x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_10, x_11, x_12, x_13 using original numbers. MySolutionVec on processor 1 contains x_14, x_15, x_16, x_17, x_18, x_19, x_20, x_21, x_22, x_23, x_24, x_25, x_26, x_27 using new numbers, or equivalently, x_4, x_5, x_6, x_7, x_8, x_9, x_10, x_11, x_12, x_13, x_14, x_15 using original numbers.

4.14

void DSC_ReFactorInitialize(DSC_Solver MyDSCObject)

A common situation is one which the application needs to factor a matrix whose sparsity structure stays the same but its nonzero values change. Use DSC_ReFactorInitialize to free up nonzero value related storage in the factor before calling DSC_JNFactor. The information from the ordering and symbolic steps is preserved and reused.

4.15

void DSC_FreeAll(DSC_Solver MyDSCObject)
This function should be called once when all work with a matrix (and its factors) are completed. This call frees up the factor and other internal data structures in the solver. At this point, the ordering and other symbolic information is also freed. If the matrix structure stays the same and the nonzero values change, then the code can utilize the ordering and symbolic steps using a call to DSC_ReFactorInitialize described earlier.

4.16

void DSC_DoStats(DSC_Solver MyDSCObject)

Timing and storage statistics are displayed to the standard output.

4.17

void DSC_Close0(DSC_Solver MyDSCObject)

This frees the MPI communicator and group created by the earlier call to DSC_Open0. Use DSC_Close0 to match DSC_Open0.

4.18

void DSC_Close1(DSC_Solver MyDSCObject)

This frees up the MPI communicators and groups created by the earlier call to DSC_Open1. Use DSC_Close1 to match DSC_Open1.

4.19

void DSC_End(DSC_Solver MyDSCObject)

This frees up the solver object created by an earlier call to DSC_Begin. After DSC_End, you can solve another system by using DSC_Begin and the other function calls again.

4.20

void DSC_ErrorDisplay(DSC_Solver MyDSCObject)

This function can be called at any point (prior to DSC_Close) to display a message to stdout describing the error (if any). If it displays nothing, then there is no error with the solver (at that processor). If DSC_Begin returns the solver object MyDSCObject, then you can examine the status of the solver by looking at DSCObject->DSC_STATUS->error_code; this field should equal DSC_NO_ERROR when there is no error.
4.21

```c
int DSC_Test2DNonz(int gridsize1, int gridsize2, int ncopies,
int *PtrGlobalNS, int **PtrAIndex,
int **PtrAStruct, int **PtrReplicates)

int DSC_Test3DNonz(int gridsize1, int gridsize2, int gridsize3,
int ncopies, int *PtrGlobalNS, int **PtrAIndex,
int **PtrAStruct, int **PtrReplicates)
```

These functions can be used to create the structure of a 2(or 3)-dimensional model 5-point finite-difference grid. They allocate and create vectors for the structure in the DSCPACK GlobalNS, AIndex, AStruct, Replicates form (returned as the last four output parameters). The replication of each grid-point is set to the same value, that given by ncopies. These functions are used in example1.c through example3.c to generate test matrices.

4.22

```c
int DSC_Test2D(int gridsize1, int gridsize2, int ncopies,
int *PtrGlobalNS, int **PtrAIndex, int **PtrAStruct,
int **PtrReplicates, RealNumberType **PtrANonz)

int DSC_Test3D(int gridsize1, int gridsize2, int gridsize3,
int ncopies, int *PtrGlobalNS, int **PtrAIndex,
int **PtrAStruct, int **PtrReplicates, RealNumberType **PtrANonz)
```

These functions can be used to create the structure and nonzero values in the matrix of a 2(or 3)-dimensional model 5-point finite-difference grid. They allocate and create vectors for the structure and nonzeroes in the DSCPACK GlobalNS, AIndex, AStruct, Replicates, ANonz form (returned as the last five output parameters). The replication of each grid-point is set to the same value, that given by ncopies. This function is suitable only for testing purposes; the memory requirements are large because the entire matrix (and not just its compact structure) is generated and stored on each processor. The matrix generated is symmetric and positive definite. These functions are used in example4.c to generate test matrices.

4.23

```c
int DSC_FitLocalNonz(DSC_Solver MyDSCObject,
RealNumberType **PtrMyANonz)
```

This is another utility function for easy testing. Assume the ordering step has been performed by DSC_Order using a specific matrix structure on a set of processors. Now DSC_FitLocalNonz can be called just before invoking DSC_NFactor to
create a vector of nonzeros. This vector corresponds to the nonzeros of an artificial symmetric positive definite matrix created to match the structure already input to the solver. The vector will contain nonzeros in columns assigned to this processor and hence can be used directly as input MyANonz into a subsequent DSC_NFactor call. These functions are used in example1.c through example3.c and example5.c to generate MyANonz vectors prior to a DSC_NFactor call.

4.24

int DSC_TestGeneralNoNonz (char *Filename, int ncopies, int *PtrGlobalNS, int **PtrAIndex, int **PtrAStruct, int **PtrReplicates)

int DSC_TestGeneral (char *Filename, int ncopies, int *PtrGlobalNS, int **PtrAIndex, int **PtrAStruct, int **PtrReplicates, RealNumberType **PtrANonz)

These are two utility functions for ease of testing. Consider a sparse matrix given in the simple traditional form in a file (such as the data files in the Data directory). The file format is a simplified version of the Harwell-Boeing format. The first line contains a character comment. The second contains GlobalNS and the total number of subscripts in the structure. This is followed by data for the index and subscript vectors followed by data for the nonzero vector. The data file format assumes traditional Fortran style numbering of columns starting at 1; the data is automatically changed to fit the C-Style used in this package. The Replicates vector is not provided in the data file; the replicates vector will be set to the value supplied in ncopies. The latter MUST be set to 1 for the DSC_TestGeneral call; see example6.c. For the DSC_TestGeneralNoNonz, the values of ncopies can be a suitable small integer; this function does not read the nonzero values of the matrix from the file. With a value of ncopies greater than 1, the structure given the file is treated as a compact representation. Nonzeros to fit this structure can be generated using the function DSC_FitLocalNonz; see example5.c.

4.25 Auxiliary Functions

We provide two sample functions to help the application developer. These functions are not part of the core package and hence do not start with the DSC prefix. These functions are defined in the file utilities.c.

4.25.1

int SetUpANonz( int MyRank, int GlobalNS, int *AIndex, int *AStruct, int *Replicates, RealNumberType *ANonz, int NumGlobalCols, int NumLocalStructs, int NumLocalCols, int NumLocalNonz, int *GlobalStructNewColNum, int *GlobalStructNewNum, int *GlobalStructOwner, int *LocalStructOldNum, RealNumberType **PtrMyANonz)
Assume the structure and nonzeros of the entire matrix are available at each processor. This can be accomplished using calls to \texttt{DSC\_TestGeneral} (with \texttt{ncopies=1}), \texttt{DSC\_Test2D}, \texttt{DSC\_Test2D} or some application specific utility. Prior to \texttt{DSC\_Nfactor}, a call to \texttt{SetUpANonz} can be used to set up the list \texttt{MyANonz} containing nonzeros in columns of the matrix assigned to that processor. The function extracts the latter from the original \texttt{ANonz}; it provides nonzeros in the diagonal and lower half of columns (mapped to that processor) in increasing order of the row subscripts using the ordering (new numbering) computed by \texttt{DSC\_Order}. This function is provided for purposes of testing and as a sample of what the application must provide in a call to \texttt{DSC\_NFactor}. \texttt{PtrMyANonz} is the only output parameter; see \texttt{example6.c}.

4.25.2

\begin{verbatim}
int SetUpLocalRhsVec(int GlobalNS, int NumLocalStructs,
 int NumLocalCols, int *LocalStructOldNum,
 int *GlobalStructNewColNum, int *GlobalStructOwner,
 int *Replicates, RealNumberType *GlobalRhsVec,
 RealNumberType *MyRhsVec)
\end{verbatim}

Assume the elements of the entire right hand side vector are available at each processor, arranged as in the original numbering of the linear system. A call to \texttt{SetUpLocalRhsVec} can be used to set up the list \texttt{MyRhsVec} containing the right hand side vector elements assigned to that processor. The function extracts the latter from the original \texttt{GlobalRhsVec} using the information in the remaining (input) parameters. \texttt{MyRhsVec} is the only output parameter; see \texttt{example6.c}.

References


