# Wrapper Objects for Solving a Linear System of Equations using **SPOOLES** 2.2

Cleve Ashcraft
Boeing Shared Services Group[1]

Peter Schartz
CSAR Corporation[2]

January 2, 1999

[1]P. O. Box 24346, Mail Stop 7L-21, Seattle, Washington 98124. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

[2]28035 Dorothy Drive, Agoura Hills, CA 91301. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

**Abstract**

The **SPOOLES** library stands for **SP**arse **O**bject **O**riented **L**inear **E**quation **S**olver. It is written in the C language using object oriented design and can solve real or complex linear systems in serial, multithreaded and MPI environments. It contains three options to order the matrices: minimum degree, generalized nested dissection and multisection. The matrices may be symmetric, Hermitian or nonsymmetric. Pivoting for numerical stability is supported.

While the functionality of the library is broad, the learning curve can be steep for the initial user. We present in this paper some "wrapper" objects in the serial, multithreaded and MPI environments that ease the transition. They were originally written to integrate the **SPOOLES** library into CSAR's CSAR-Nastran library.

The wrapper objects are presented as a learning device; anything that reduces the interface between the user and the library also restricts the ability to tune the library to a particular need. This drawback is ameliorated by a number of wrapper methods that allow the user to change default parameters that govern the ordering, factorization and solve.

# Contents

# Chapter 1

# Introduction

One common task for the **SPOOLES** library is to solve a linear system of equations $AX = Y$. The matrix $A$ is large and sparse, and the right hand side $Y$ and solution $X$ will have one or more columns. The matrices may be real or complex. We will consider the case where $A$ is square, and could be symmetric, Hermitian, or nonsymmetric.

The first step is the find a permutation matrix $P$ such that $\widehat{A} = PAP^T$ has a *low-fill* factorization, i.e., after $\widehat{A}$ has been factored into $(U^T + I)D(I + U)$ (if $\widehat{A}$ is symmetric), $(U^H + I)D(I + U)$ (if $\widehat{A}$ is Hermitian), or $(L + I)D(I + U)$ (if $\widehat{A}$ is nonsymmetric), the factor matrices $L$ and $U$ have relatively few entries. The **SPOOLES** library can compute three types of low-fill orderings: minimum degree, generalized nested dissection, and multisection.

The second step is to permute $A$ into $\widehat{A}$ and compute the factorization. The **SPOOLES** library has a great deal of flexiblity in this step. The *choreography*, (what data structures exist, what block computations take place), can be specified and tuned by the knowledgeable user. Pivoting for numerical stability is supported, i.e., the user can specify an upper bound on the magnitudes of the entries in $L$ and $U$. The factorization can be exact (up to roundoff) or approximate, where entries that are small in magnitude are dropped, neither stored nor used in computations.

The last step is to solve $\widehat{A}\widehat{X} = \widehat{Y}$, where $\widehat{Y} = PY$, and $X = P\widehat{X}$. The matrix $Y$ needs to be permuted to form $\widehat{Y}$, and $X$ is obtained from $\widehat{X}$ by a permutation.

Needless to say, the complex process outlined above can be intimidating to the first time user. The complete step-by-step process for serial, multithreaded and MPI environments is described at length in the **SPOOLES** User's Manual. The purpose of this document is to present a *vastly* simplified approach for the first-time user. We describe three *wrapper* objects that we wrote for the integration of the **SPOOLES** library into CSAR-Nastran. Bewarned, while the wrapper objects insulate the user from many of the details, they also restrict the ability of the user to tune the code to the particular linear system. We hope that these wrapper methods will provide a gentle introduction to the library, and be a good example from which the user can tune as necessary.

The user's application program must interface with the **SPOOLES** library in some manner. The serial, multithreaded and MPI wrapper objects we describe in sections 3, 4 and 5. But first the user must communicate the matrix $A$ and right hand side $Y$ to the library, and receive back the solution $X$. To do this the user must generate two **SPOOLES** objects — a `InpMtx` object for $A$ and `DenseMtx` objects for $Y$ and $X$. This process is described in section 2.

Serial code has one process and one address space. Multithreaded code can have multiple threads sharing one address space. The **SPOOLES** library utilizes multiple threads only in the factorization and solve steps. All other operations act on the global data structures using serial methods. In the MPI environment, the data structures for $A$, $X$ and $Y$ may be distributed, and all working data structures that contain the factor matrices and their supporting information are distributed. The MPI code is much more complex than the serial or multithreaded codes, for not only are the factor and solves parallel and distributed (as is the symbolic factorization), but there is a great deal of support code necessary because of the distributed data structures.

The wrapper methods described in this paper do not exercise all the functionality of the MPI environment. This is due to the present state of the CSAR-Nastran code from CSAR, where the matrix $A$ and right hand side $Y$ are generated on one processor. We chose to do all the serial preprocessing

- generate a graph of the matrix,
- order the graph,
- compute the symbolic factorization,
- and construct the permutations

on processor 0 that reads in $A$ and $Y$ from the CSAR-Nastran files. Since the bulk of the overall time for a CSAR-Nastran run is dominated by the factor and solves, this approach was considered acceptable. For the user who is interested in using the MPI environment for the entire process, e.g., when $A$ and $Y$ cannot fit on one processor, see the **SPOOLES** User Manual for driver programs.

# Chapter 2

# Setting up the linear system

Our typical user is interested in solving $AX = Y$, where $A$ is square, large and sparse, and $X$ and $Y$ are dense matrices with one or more columns. **SPOOLES** is a very large sophisticated library with a commensurate learning curve to master its functionality. But what is the bare minimum a user has to know to obtain a solution to their linear system?

- They need to construct an `InpMtx` object that holds the entries of $A$. (`InpMtx` stands for `Input matrix`, for it is an easy to use object that one uses to input, assemble, sort and manipulate entries in a sparse matrix.)
- They need to construct a `DenseMtx` object that holds the entries of $Y$.
- They need to construct a `DenseMtx` object to hold the entries of $X$.

These two objects encapsulate the minimal interface to the **SPOOLES** library. the application program needs to know how to construct the `InpMtx` and `DenseMtx` objects, either directly inside an application program, or by reading in a custom matrix file. This is what we now describe.

## 2.1 Constructing an `InpMtx` object

The `InpMtx` object is more of an "Input" object than a "Matrix" object. It descended from an out-of-core assembly code that assembled and sorted entries of a sparse matrix. Simplicity and functionality are its goals, at some expense of efficiency in storage and computation. *Note: all indices are zero-based as in C, not 1-based as in FORTRAN.*

The `InpMtx` object is simplest understood as a "bag" of triples $\langle r(i,j), c(i,j), a_{i,j}\rangle$, where $r()$ and $c()$ are some functions that define the first and second coordinates. Each `InpMtx` object has a "coordinate type", one of

- `INPMTX_BY_ROWS`, where $r(i,j) = i$, $c(i,j) = j$.
- `INPMTX_BY_COLUMNS`, where $r(i,j) = j$, $c(i,j) = i$.
- `INPMTX_BY_CHEVRONS`, where $r(i,j) = \min(i,j)$, $c(i,j) = j - i$.

Rows and columns are self-explanatory, the first coordinate $r(i,j)$ is either the row or column of $a_{i,j}$. The $j$-th "chevron" is composed of the diagonal entry $a_{j,j}$, entries in the $j$-th row of the upper triangle, and entries in the $j$-th column of the lower triangle. It is the natural data structure for the assembly of the matrix entries into the "fronts" used to factor the matrix.

The `InpMtx` object can hold one of three types of entries as "indices only" (no entries are present), real entries, or complex entries. The type is specified by the `inputMode` parameter to the `InpMtx_init()` method.

- `INPMTX_INDICES_ONLY` where the triples $langler(i,j),c(i,j),-\rangle$ are really only pairs, i.e., no numerical values are present. This mode is useful for assembling graphs.
- `SPOOLES_REAL` where $a_{i,j}$ is a real number, a `double` value.
- `SPOOLES_COMPLEX` where $a_{i,j}$ is a complex number, really two consecutive `double` values.

"Coodinate type" and "input mode" (equivalently, the type of entries) are the two parameters that must be specified when initializing an `InpMtx` object.

```
InpMtx   *mtxA = InpMtx_new() ;
InpMtx_init(mtxA, coordType, inputMode, 0, 0) ;
```

Every object in the **SPOOLES** library is initialized via an *ObjectName*`_new()` method, which allocates space for the object and sets its fields to default values. If you wish to use an *automatic* variable, then one must explicitly set the default fields, as follows.

```
InpMtx   mtxA ;
InpMtx_setDefaultFields(&mtxA) ;
InpMtx_init(&mtxA, coordType, inputMode, 0, 0) ;
```

Only the coordinate type and input mode are necessary. The fourth and fifth arguments are upper bounds on the number of entries and vectors for the object. (More on vectors in just a moment.) The user does not need to know values for the number of entries or vectors, for the object resizes itself as necessary as information is placed into it.

"Vectors" is one way that the entries can be stored. There are actually three ways, specified by the `storageMode` field of the `InpMtx` object.

- `INPMTX_RAW_DATA`, where the pairs or triples are stored in unordered form.
- `INPMTX_SORTED`, where the pairs or triples are stored in ascending lexicographic order of the first two coordinates.
- `INPMTX_BY_VECTORS`, where the pairs or triples are sorted and stored in vectors defined by their first coordinate.

The storage mode can be changed via a call to `InpMtx_changeStorageMode()`.

The user does not really need to know about this "storage mode". Fill the `InpMtx` object with data in any way at all (we will describe this shortly). The wrapper method will check that the data is in the form it needs. If is isn't, the object will be transformed as necessary. The "sort" operation is really "sort-and-compress", the pairs or triples are sorted into ascending order, and then the list is scanned duplicates are "merged" together, i.e., if real or complex entries are present, they are added together. (This allows us to assemble finite element matrices.) The knowledgeable user can change the storage mode as necessary, and thus avoiding expensive sorts when possible. For example, after reading in the matrix data from the CSAR-Nastran file, the entries are already in sorted form, and the explicit sort can be avoided.

Now let us see how we "input" information into the `InpMtx` object. There are several input methods, e.g., single entries, rows, columns, and submatrices, and each input method has three types of input, e.g, indices only, real entries, or complex entries. Here are the prototypes below.

- Input methods for "indices only" mode.

```
void InpMtx_inputEntry ( InpMtx *mtxA, int row, int col ) ;
void InpMtx_inputRow ( InpMtx *mtxA, int row, int rowsize, int rowind[] ) ;
void InpMtx_inputColumn ( InpMtx *mtxA, int col, int colsize, int colind[] ) ;
void InpMtx_inputMatrix ( InpMtx *mtxA, int nrow, int ncol, int rowstride,
                          int colstride, int rowind[], colind[] ) ;
```

- Input methods for real entries.

```
void InpMtx_inputRealEntry ( InpMtx *mtxA, int row, int col, double value ) ;
void InpMtx_inputRealRow ( InpMtx *mtxA, int row, int rowsize,
                           int rowind[], double rowent[] ) ;
void InpMtx_inputRealColumn ( InpMtx *mtxA, int col, int colsize,
                              int colind[], double colent[] ) ;
void InpMtx_inputRealMatrix ( InpMtx *mtxA, int nrow, int ncol, int rowstride,
                    int colstride, int rowind[], colind[], double mtxent[] ) ;
```

- Input methods for complex entries.

```
void InpMtx_inputComplexEntry ( InpMtx *mtxA, int row, int col,
                                double real, double imag ) ;
void InpMtx_inputComplexRow ( InpMtx *mtxA, int row, int rowsize,
                              int rowind[], double rowent[] ) ;
void InpMtx_inputComplexColumn ( InpMtx *mtxA, int col, int colsize,
                                 int colind[], double colent[] ) ;
void InpMtx_inputComplexMatrix ( InpMtx *mtxA, int nrow, int ncol, int rowstride,
              int colstride, int rowind[], colind[], double mtxent[] ) ;
```

The `rowind[]` row indices and `colind[]` column indices are precisely that. Don't worry about what coordinate type the `InpMtx` object has, the translation from row and column indices into the particular coordinate is done inside the input methods.

Let us look at a particular example, where we have a `n1`×`n2` grid and we want to have a $\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$ 5-point operator at each grid point. Note, this matrix is symmetric, so we need input only the upper triangle (or the lower triangle) of the matrix.

```
mtxA = InpMtx_new() ;
InpMtx_init(mtxA, INPMTX_BY_ROWS, SPOOLES_REAL, 0, 0) ;
for ( ii = 0 ; ii < n1 ; ii++ ) {
   for ( jj = 0 ; jj < n2 ; jj++ ) {
      ij = ii + jj*n1 ;
      indices[0] = ij ;
      entries[0] = 4.0 ;
      count = 1 ;
      if ( ii < n1 ) {
         indices[count] = ij + 1 ;
         entries[count] = -1.0 ;
         count++ ;
      }
      if ( jj < n2 ) {
         indices[count] = ij + n1 ;
```

```
            entries[count] = -1.0 ;
            count++ ;
         }
         InpMtx_inputRealRow(mtxA, ij, count, indices, entries) ;
      }
   }
   InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
```

The process begins by allocating an `InpMtx` object `mtxA` using the `InpMtx_new()` method, initializing it with the `InpMtx_init()` method, and filling it with matrix entries with the `InpMtx_inputRealRow()` method. The last method, `InpMtx_changeStorageMode()`, "assembles" the data (not really necessary because the entries are disjoint, "sorts" the data (again not necessary since the entries were input in ascending order, and creates a vector structure inside the `InpMtx` object that allows easy access to each individual row.

We could have input all the entries and treated it as a nonsymmetric matrix, but that would not be efficient with respect to storage or factorization cost. Alternatively, we could have input all the entries and called the `InpMtx_dropLowerTriangle()` method to drop the lower triangular entries.

## 2.2 Constructing an `DenseMtx` object

The `DenseMtx` stores a real or complex dense matrix. It is not just an array of numbers, it also has row indices and column indices. This allows it to exist in a distributed MPI environment where each processors has only a submatrix of the matrix. Here is how to initialize a `DenseMtx` object.

```
int        type, rowid, colid, nrow, ncol, inc1, inc2 ;
DenseMtx   *mtx = DenseMtx_new() ;
DenseMtx_init(mtx, type, rowid, colid, nrow, ncol, inc1, inc2) ;
```

- The `type` is either `SPOOLES_REAL` or `SPOOLES_COMPLEX`.
- The `rowid` and `colid` values are used to identify a `DenseMtx` as a submatrix of a larger matrix. Any values are suitable.
- `nrow` and `ncol` are the number of rows and columns in the matrix, respectively.
- The entries of the matrix can be stored in either row major or column major form. For row major, use `inc1 = ncol` and `inc2 = 1`. For column major, use `inc1 = 1` and `inc2 = nrow`. Note, all solve and matrix-matrix multiply methods require that the `DenseMtx` object be column major.

For example, here is the call to initialize a `DenseMtx` object to have real entries, 100 rows and 5 columns, entries column major.

```
DenseMtx_init(mtx, SPOOLES_REAL, 0, 0, 100, 5, 1, 100) ;
```

During the initialization, the row indices are set to $0, 1, \ldots, \texttt{nrow} - 1$ and the column indices are set to $0, 1, \ldots, \texttt{ncol} - 1$. The entries are **not** initialized. Zero the entries with a call to `DenseMtx_zero()`. (This is crucial when loading a sparse right hand side into the `DenseMtx` object.)

Once we have the `DenseMtx` object initialized, we want to be able to access the row indices, the column indices and the entries. We do this through instance methods.

```
void DenseMtx_rowIndices ( DenseMtx *mtx, int *pnrow, int *prowind ) ;
void DenseMtx_columnIndices ( DenseMtx *mtx, int *pncol, int *pcolind ) ;
double * DenseMtx_entries ( DenseMtx *mtx ) ;
```

We would use them as follows.

```
double   *entries ;
int      ncol, nrow, *colind, *rowind ;

DenseMtx_rowIndices(mtx, &nrow, &rowind) ;
DenseMtx_columnIndices(mtx, &ncol, &colind) ;
entries = DenseMtx_entries(mtx) ;
```

We can now fill the indices or the entries. The location of the `(irow,jcol)` entry is found at `offset = irow*inc1 + jcol*inc2`. The row and column increments can be found as follows.

```
int inc1 = DenseMtx_rowIncrement(mtx) ;
int inc2 = DenseMtx_columnIncrement(mtx) ;
```

To avoid dealing with row and column increments, we can retrieve and set values of a particular entry.

```
double   value, real, imag ;
int      irow, jcol ;

DenseMtx_realEntry(mtx, irow, jcol, &value) ;
DenseMtx_complexEntry(mtx, irow, jcol, &real, &imag) ;
DenseMtx_setRealEntry(mtx, irow, jcol, value + 10.) ;
DenseMtx_setComplexEntry(mtx, irow, jcol, real + 1., imag + 2.) ;
```

As a real example, consider the `n1` $\times$ `n2` grid from the previous subsection, where we assembled a finite difference matrix. Assume that the right hand side is zero except for points where `(n1-1,0:n2-1)`, where a unit load is applied. Here is the code to generate the `DenseMtx` object.

```
mtxY = DenseMtx_new();
DenseMtx_init(mtxY, SPOOLES_REAL, 0, 0, n1*n2, 1, 1, n1*n2) ;
DenseMtx_zero(mtxY) ;
ii = n1 - 1 ;
for ( jj = 0 ; jj < n2 ; jj++ ) {
   ij = ii + jj*n1 ;
   DenseMtx_setRealEntry(mtxY, ij, 1, 1.0) ;
}
```

Do not forget to zero the entries in `mtxY` before setting any entries.

## 2.3 IO for the `InpMtx` and `DenseMtx` objects

The three driver programs that we describe in the next sections read $A$ and $Y$ from files and write $X$ to a file. So the first thing we know is that the `InpMtx` and `DenseMtx` objects can read and write themselves from and to files. This convention is supported by most of the objects in the **SPOOLES** library. In fact, there is a common *protocol* that is followed. Let us take a look at the common IO methods for the `InpMtx`.

- `int InpMtx_readFromFile ( InpMtx *obj, char *filename ) ;`
- `int InpMtx_readFromFormattedFile ( InpMtx *obj, FILE *fp ) ;`
- `int InpMtx_readFromBinaryFile ( InpMtx *obj, FILE *fp ) ;`

- `int InpMtx_writeToFile ( InpMtx *obj, char *filename ) ;`
- `int InpMtx_writeToFormattedFile ( InpMtx *obj, FILE *fp ) ;`
- `int InpMtx_writeToBinaryFile ( InpMtx *obj, FILE *fp ) ;`
- `int InpMtx_writeForHumanEye ( InpMtx *obj, FILE *fp ) ;`

There are corresponding methods for the `DenseMtx` object, just replace "`Inp`" by "`Dense`" in the above prototypes.

Two methods take as input `char *` file names. Each object can be archived in its own file with a particular suffix. For example, `InpMtx` objects can be read from and written to files of the form `*.inpmtxf` for a formatted file and `*.inpmtxb` for a binary file. For a `DenseMtx` object, the file names are `*.densemtxf` and `*.densemtxb`. The `InpMtx_readFromFile()` method looks at the `filename` argument, and calls the binary or formatted read methods, depending on the suffix of `filename`. A normal return code is `1`. If the suffix does not match either `*.inpmtxf` or `*.inpmtxb`, an error message is printed and the return code is `0`. Something similar works for writing an `InpMtx` object to a file using `InpMtx_writeToFile()`, except if `filename`'s suffix does not match, the `InpMtx_writeForHumanEye()` method is called.

Here are three approaches to link $A$ and $Y$ from an application code to the `InpMtx` and `DenseMtx` objects demanded by the **SPOOLES** application.

- An application could take the simple approach of creating an `InpMtx` and `DenseMtx` object to hold $A$ and $Y$, write them to a file, and then call a totally separate code that functions much like our drivers, reading in $A$ and $Y$, computing $X$ and writing $X$ to a file, which is then read in by the application code.
- A second approach, one that was taken during the first integration of the **SPOOLES** library into CSAR-Nastran, was to have the CSAR-Nastran code generate two files for $A$ and $Y$ in CSAR-Nastran format. (This way CSAR-Nastran did not need to know any of the **SPOOLES** interface.) Two custom routines were written to read in the entries of $A$ and $Y$ from the CSAR-Nastran files and construct `InpMtx` and `DenseMtx` objects. The wrapper routines we describe in the next three chapters were called to solve for $X$ which was then written to a CSAR-Nastran file.
- A third approach would be to generate the `InpMtx` and `DenseMtx` objects in the application program, and then call the wrapper methods to solve for $X$, i.e., no IO would be necessary.

# Chapter 3

# The Serial Wrapper Object and Driver

The goal is to solve $AX = Y$ in a serial environment. Section 1 of the User's Manual presents a listing of the `AllInOne.c` driver program for solving $AX = Y$. There are nine steps, and each requires "mid-level" knowledge of several objects of the **SPOOLES** library. To reduce the complexity of using the library, (and the complexity rises dramatically in the multithreaded and MPI environments), we created the `Bridge` object. The term "bridge" symbolizes spanning the distance between the **SPOOLES** library and the CSAR-Nastran application code. The nine steps of the `allInOne.c` driver program is reduced to three using the `Bridge` object.

- Initialization and setup step.

  Here the `Bridge` object is allocated via a call to `Bridge_new()`. Parameters are set using `Bridge_set*()` methods. The setup phase orders the matrix and prepares all the necessary **SPOOLES** data structures for the factorization and solve that follows.

- Factorization step.

  The matrix is factored via a call to `Bridge_factor()`.

- Solution step.

  The linear system is solved via a call to `Bridge_solve()`.

The `Bridge` object has many parameters that control the ordering of the matrix, the pivoting tolerance (if pivoting is requested), the drop tolerance (for an approximate factorization), and so on. Rather than burden the user with the knowledge of and setting these parameters, there are decent default values built into the object. There are also methods to set various parameters to allow the user some control over the ordering, factor and solve processes.

Section 3.1 takes a quick look at the `Bridge` driver program (whose complete listing is found in Appendix A). Section 3.2 describes the internal data fields of the `Bridge` object. Section 3.3 contains the prototypes and descriptions of all `Bridge` methods.

## 3.1 A quick look at serial driver program

The entire listing of this serial driver is found in Appendix A. We now extract parts of the code.

- Decode the input.

  ```
  msglvl       = atoi(argv[1]) ;
  msgFileName  = argv[6] ;
  neqns        = atoi(argv[3]) ;
  type         = atoi(argv[4]) ;
  symmetryflag = atoi(argv[5]) ;
  mtxFileName  = argv[6] ;
  rhsFileName  = argv[7] ;
  solFileName  = argv[8] ;
  seed         = atoi(argv[9]) ;
  ```

  Here is a description of the input parameters.

  – `msglvl` is the message level.
  – `msgFile` is the message file name
  – `neqns` is the number of equations.
  – `type` is the type of entries: 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
  – `symmetryflag` is the type of matrix symmetry: 0 (`SPOOLES_SYMMETRIC`), 1 (`SPOOLES_HERMITIAN`) or 2 (`SPOOLES_NONSYMMETRIC`).
  – `mtxFile` is the name of the file from which to read the `InpMtx` object for $A$. The file name must have the form `*.inpmtxb` for a binary file or `*.inpmtxf` for a formatted file.
  – `rhsFile` is the name of the file from which to read the `DenseMtx` object for the right hand side $Y$. The file name must have the form `*.densemtxb` for a binary file or `*.densemtxf` for a formatted file.
  – `solFile` is the name of the file to write the `DenseMtx` object for the solution $X$. The file name must have the form `*.densemtxb` for a binary file or `*.densemtxf` for a formatted file, `"none"` for no output, or any other name for a human-readable listing.
  – `seed` is a random number seed used in the ordering process.

- Read in the `InpMtx` object for $A$.

  ```
  mtxA = InpMtx_new() ;
  rc = InpMtx_readFromFile(mtxA, mtxFileName) ;
  ```

  The `rc` parameter is the error return. In the driver it is tested for an error, but we omit this from the present discussion.

- Read in the `DenseMtx` object for $Y$.

  ```
  mtxY = DenseMtx_new() ;
  rc = DenseMtx_readFromFile(mtxY, mtxFileName) ;
  DenseMtx_dimensions(mtxY, &nrow, &nrhs) ;
  ```

  The `nrhs` parameter contains the number of right hand sides, or equivalently, the number of columns in $Y$.

- Create and setup the `Bridge` object.

```
bridge = Bridge_new() ;
Bridge_setMatrixParams(bridge, neqns, type, symmetryflag) ;
Bridge_setMessageInfo(bridge, msglvl, msgFile) ;
rc = Bridge_setup(bridge, mtxA) ;
```

The `Bridge` object is allocated by `Bridge_new()`, and various parameters are set. The actual ordering of the matrix, symbolic factorization, and permutation creation are performed inside the `Bridge_setup()` method.

- Compute the matrix factorization.

```
permuteflag = 1 ;
rc = Bridge_factor(bridge, mtxA, permuteflag, &error) ;
```

When `permuteflag` is `1`, it means that the matrix in `mtxA` has not yet been permuted into the new ordering and so is done inside the method. The `error` flag is filled with an error code that tells how far the factorization was able to proceed. If `rc = 1`, the factorization completed without any error.

- Solve the linear system.

```
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
rc = Bridge_solve(bridge, permuteflag, mtxX, mtxY) ;
```

The `DenseMtx` object `mtxX` is created and initialized to be the same type and size as `mtxY`. Its entries are explicitly zeroed (this is not necessary but is a good idea in general). The solution is then solved. Again, note the presence of `permuteflag`. When `1`, `mtxY` needs to be permuted into the new ordering, and `mtxX` is returned in the original ordering.

## 3.2 The `Bridge` Data Structure

The `Bridge` structure has the following fields.

- Graph parameters:
  - `int neqns` : number of equations, i.e., number of vertices in the graph.
  - `int nedges` : number of edges (includes $(u, v)$, $(v, u)$ and $(u, u)$).
  - `int Neqns` : number of equations in the compressed graph.
  - `int Nedges` : number of edges in the compressed graph.
- Ordering parameters:
  - `int maxdomainsize` : maximum size of a subgraph to not split any further during the nested dissection process.
  - `int maxnzeros` : maximum number of zeros to allow in a front during the supernode amalgamation process.
  - `int maxsize` : maximum size of a front when the fronts are split.
  - `int seed` : random number seed.

   - `double compressCutoff` : if the `Neqns` $<$ `compressCutoff` $*$ `neqns`, then the compressed graph is formed, ordered and used to create the symbolic factorization.

- Matrix parameters:
   - `int type` : type of entries, `SPOOLES_REAL` or `SPOOLES_COMPLEX`, default value is `SPOOLES_REAL`.
   - `int symmetryflag` : type of symmetry for the matrix, `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`, default value is `SPOOLES_SYMMETRIC`.
- Factorization parameters:
   - `int sparsityflag` : `SPOOLES_DENSE_FRONTS` for a direct factorization, or `SPOOLES_SPARSE_FRONTS` for an approximate factorization, default value is `SPOOLES_DENSE_FRONTS`.
   - `int pivotingflag` : `SPOOLES_PIVOTING` for pivoting enabled, or `SPOOLES_NO_PIVOTING` for no pivoting, default value is `SPOOLES_NO_PIVOTING`.
   - `double tau` : used when pivoting is enabled, all entries in $L$ and $U$ have magnitude less than or equal to `tau`, default value is 100.
   - `double droptol` : used for an approximation, all entries in $L$ and $U$ that are kept have magnitude greater than or equal to `droptol`. default value is 0.001.
   - `PatchAndGoInfo *patchinfo` : pointer to an object that controls special factorizations for optimization matrices and singular matrices from structural analysis, default value is `NULL` which means no special action is taken. See the Reference Manual for more information.
- Pointers to objects:
   - `ETree *frontETree` : object that defines the factorizations, e.g., the number of fronts, the tree they form, the number of internal and external rows for each front, and the map from vertices to the front where it is contained.
   - `IVL *symbfacIVL` : object that contains the symbolic factorization of the matrix.
   - `SubMtxManager *mtxmanager` : object that manages the `SubMtx` objects that store the factor entries and are used in the solves.
   - `FrontMtx *frontmtx` : object that stores the $L$, $D$ and $U$ factor matrices.
   - `IV *oldToNewIV` : object that stores old-to-new permutation vector.
   - `IV *newToOldIV` : object that stores new-to-old permutation vector.
- Message information, statistics and cpu times:
   - `int msglvl` : message level for output. When 0, no output, When 1, just statistics and cpu times. When greater than 1, more and more output.
   - `FILE *msgFile` : message file for output. When `msglvl` $> 0$, `msgFile` must not be `NULL`.
   - `int stats[6]` : statistics for the factorization.

| | | | |
|---|---|---|---|
| `stats[0]` : | # of pivots | `stats[3]` : | # of entries in $D$ |
| `stats[1]` : | # of pivot tests | `stats[4]` : | # of entries in $L$ |
| `stats[2]` : | # of delayed rows and columns | `stats[5]` : | # of entries in $U$ |

   - `double cpus[14]` : cpus for the different functions.

| | | | |
|---|---|---|---|
| `cpus[0]` : | time to construct `Graph` | `cpus[7]` : | time to factor matrix |
| `cpus[1]` : | time to compress `Graph` | `cpus[8]` : | time to post-process matrix |
| `cpus[2]` : | time to order `Graph` | `cpus[9]` : | total factor time |
| `cpus[3]` : | time for symbolic factorization | `cpus[10]` : | time to permute rhs |
| `cpus[4]` : | total setup time | `cpus[11]` : | time to solve |
| `cpus[5]` : | time to permute matrix | `cpus[12]` : | time to permute solution |
| `cpus[6]` : | time to initialize front matrix | `cpus[13]` : | total solve time |

## 3.3 Prototypes and descriptions of `Bridge` methods

This section contains brief descriptions including prototypes of all methods that belong to the `Bridge` object.

### 3.3.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `Bridge * Bridge_new ( void ) ;`

   This method simply allocates storage for the `Bridge` structure and then sets the default fields by a call to `Bridge_setDefaultFields()`.

2. `int Bridge_setDefaultFields ( Bridge *bridge ) ;`

   The structure's fields are set to default values:

   - `neqns` = `nedges` = `Neqns` = `Nedges` = 0.
   - `maxdomainsize` = `maxnzeros` = `maxsize` = `seed` = -1. `compressCutoff` = 0.
   - `type` = `SPOOLES_REAL`.
   - `symmetryflag` = `SPOOLES_SYMMETRIC`.
   - `sparsityflag` = `SPOOLES_DENSE_FRONTS`.
   - `pivotingflag` = `SPOOLES_NO_PIVOTING`.
   - `tau` = 100., `droptol` = 0.001.
   - `patchinfo` = `frontETree` = `symbfacIVL` = `mtxmanager` = `frontmtx` = `oldToNewIV` = `newToOldIV` = `NULL`.

   The `stats[6]` and `cpus[14]` vectors are filled with zeros.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

3. `int Bridge_clearData ( Bridge *bridge ) ;`

   This method clears the object and free's any owned data. It then calls `Bridge_setDefaultFields()`.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

4. `int Bridge_free ( Bridge *bridge ) ;`

   This method releases any storage by a call to `Bridge_clearData()` and then free the space for `bridge`.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

### 3.3.2 Instance methods

1. `int Bridge_oldToNewIV ( Bridge *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `oldToNewIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

2. `int Bridge_newToOldIV ( Bridge *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `newToOldIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

3. `int Bridge_frontETree ( Bridge *bridge, ETree **pobj ) ;`

   This method fills `*pobj` with its `frontETree` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

4. `int Bridge_symbfacIVL ( Bridge *bridge, IVL **pobj ) ;`

   This method fills `*pobj` with its `symbfacIVL` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

5. `int Bridge_mtxmanager ( Bridge *bridge, SubMtxManager **pobj ) ;`

   This method fills `*pobj` with its `mtxmanager` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

6. `int Bridge_frontmtx ( Bridge *bridge, FrontMtx **pobj ) ;`

   This method fills `*pobj` with its `frontmtx` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

### 3.3.3 Parameter methods

1. `int Bridge_setMatrixParams ( Bridge *bridge, int neqns, int type, int symmetryflag ) ;`

   This method sets the number of equations, type of entries, and symmetry type of the matrix.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `type` is invalid |
| -1 | `bridge` is `NULL` | -4 | `symmetryflag` is invalid |
| -2 | `neqns` $\le 0$ | -5 | symmetry flag is Hermitian but type is real |

2. `int Bridge_setOrderingParams ( Bridge *bridge, int maxdomainsize, int maxnzeros,`
   `int maxsize, int seed, double compressCutoff ) ;`

   This method sets parameters needed for the ordering.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `maxsize` $\le 0$ |
| -1 | `bridge` is `NULL` | -4 | `compressCutoff` $> 1$ |
| -2 | `maxdomainsize` $\le 0$ | | |

3. `int Bridge_setFactorParams ( Bridge *bridge, int sparsityflag, int pivotingflag,`
   `double tau, double droptol, PatchAndGoInfo *patchinfo ) ;`

   This method sets parameters needed for the factorization.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `pivotingflag` is invalid |
| -1 | `bridge` is `NULL` | -4 | `tau` $< 2.0$ |
| -2 | `sparsityflag` is invalid | -5 | `droptol` $< 0.0$ |

4. `int Bridge_setMessagesInfo ( Bridge *bridge, int msglvl, FILE *msgFile ) ;`

   This method sets the message level and file.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`, -2 if `msglvl` $> 0$ and `msgFile` is `NULL`.

### 3.3.4 Setup methods

1. `int Bridge_setup ( Bridge *bridge, InpMtx *mtxA ) ;`

   This method orders the graph, generates the front tree, computes the symbolic factorization, and creates the two permutation vectors.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`, -2 if `mtxA` is `NULL`.

2. `int Bridge_factorStats ( Bridge *bridge, int type, int symmetryflag, int *pnfront,`
   `int *pnfactorind, int *pnfactorent, int *pnsolveops, double *pnfactorops ) ;`

   This method takes as input the type and symmetry of the matrix, and fills the pointer fields with the number of fronts, factor indices, factor entries, forward and back solve operations, and factor operations.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -6 | `pnfront` is `NULL` |
| -1 | `bridge` is `NULL` | -7 | `pnfactorind` is `NULL` |
| -2 | `type` is invalid | -8 | `pnfactorent` is `NULL` |
| -3 | `symmetryflag` is invalid | -9 | `pnsolveops` is `NULL` |
| -4 | `type` is real but `symmetryflag` is Hermitian | -10 | `pnfactorops` is `NULL` |
| -5 | front tree is not present | | |

### 3.3.5 Factor method

1. `int Bridge_factor ( Bridge *bridge, InpMtx *mtxA, int permuteflag, int *perror ) ;`

   This method permutes the matrix into the new ordering (if `permuteflag` is 1), factors the matrix, and then post-processes the factors.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return, factorization did complete | -1 | `bridge` is `NULL` |
| 0 | factorization did not complete | -2 | `mtxA` is `NULL` |
| | | -3 | `perror` is `NULL` |

### 3.3.6 Solve method

1. `int Bridge_solve ( Bridge *bridge, int permuteflag, DenseMtx *mtxX, DenseMtx *mtxY ) ;`

   If `permuteflag` is 1, then `mtxY` is permuted into the new ordering. The linear system $AX = Y$ is solved. If `permuteflag` is 1, then `mtxX` is permuted into the old ordering.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -4 | `frontmtx` is `NULL` |
| -1 | `bridge` is `NULL` | -5 | `mtxmanager` is `NULL` |
| -2 | `X` is `NULL` | -6 | `oldToNewIV` needed, but not available |
| -3 | `Y` is `NULL` | -7 | `newToOldIV` needed, but not available |

# Chapter 4

# The Multithreaded Wrapper Object and Driver

The goal is to solve $AX = Y$ in a multithreaded environment. Section 8 of the User's Manual presents a listing of the `AllInOneMT.c` driver program for solving $AX = Y$. There are ten steps, and each requires "mid-level" knowledge of several objects of the **SPOOLES** library. To reduce the complexity of using the library, (and the complexity rises dramatically in the MPI environments), we created the `BridgeMT` object. The term "bridge" symbolizes spanning the distance between the **SPOOLES** library and the CSAR Nastran application code. The ten steps of the `allInOneMT.c` driver program is reduced to five using the `BridgeMT` object.

- Initialization and setup step.

  Here the `BridgeMT` object is allocated via a call to `BridgeMT_new()`. Parameters are set using `BridgeMT_set*()` methods. The setup phase orders the matrix and prepares all the necessary **SPOOLES** data structures for the factorization and solve that follow

- Setup the numerical factorization.

  In this step, `BridgeMT_factorSetup()` is called to define the parallelism for the factorization, and all data structures for the parallel execution are created.

- Factorization step.

  The matrix is factored via a call to `BridgeMT_factor()`.

- Setup the numerical solves.

  `BridgeMT_solveSetup()` is called to set up the parallel solves. This must be called *once* after a factorization, one or more solves may follow.

- Solution step.

  The linear system is solved via a call to `BridgeMT_solve()`.

The `BridgeMT` object has many parameters that control the ordering of the matrix, the pivoting tolerance (if pivoting is requested), the drop tolerance (for an approximate factorization), and so on. Rather than burden the user with the knowledge of and setting these parameters, there are decent default values built into the object.

Section 4.1 takes a quick look at the `BridgeMT` driver program (whose complete listing is found in Appendix B). Section 4.2 describes the internal data fields of the `BridgeMT` object. Section 3.3 contains the prototypes and descriptions of all `Bridge` methods.

## 4.1 A quick look at the multithreaded driver program

The entire listing of this multithreaded driver is found in Appendix B. We now extract parts of the code.

- Decode the input.

  ```
  msglvl       = atoi(argv[1]) ;
  msgFileName  = argv[6] ;
  neqns        = atoi(argv[3]) ;
  type         = atoi(argv[4]) ;
  symmetryflag = atoi(argv[5]) ;
  mtxFileName  = argv[6] ;
  rhsFileName  = argv[7] ;
  solFileName  = argv[8] ;
  seed         = atoi(argv[9]) ;
  nthread      = atoi(argv[10]) ;
  ```

  Here is a description of the input parameters.

  - `msglvl` is the message level.
  - `msgFile` is the message file name
  - `neqns` is the number of equations.
  - `type` is the type of entries: 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
  - `symmetryflag` is the type of matrix symmetry: 0 (`SPOOLES_SYMMETRIC`), 1 (`SPOOLES_HERMITIAN`) or 2 (`SPOOLES_NONSYMMETRIC`).
  - `mtxFile` is the name of the file from which to read the `InpMtx` object for $A$. The file name must have the form `*.inpmtxb` for a binary file or `*.inpmtxf` for a formatted file.
  - `rhsFile` is the name of the file from which to read the `DenseMtx` object for the right hand side $Y$. The file name must have the form `*.densemtxb` for a binary file or `*.densemtxf` for a formatted file.
  - `solFile` is the name of the file to write the `DenseMtx` object for the solution $X$. The file name must have the form `*.densemtxb` for a binary file or `*.densemtxf` for a formatted file, `"none"` for no output, or any other name for a human-readable listing.
  - `seed` is a random number seed used in the ordering process.
  - `nthread` is the number of threads to be used in the factorization and solve.

- Read in the `InpMtx` object for $A$.

  ```
  mtxA = InpMtx_new() ;
  rc = InpMtx_readFromFile(mtxA, mtxFileName) ;
  ```

  The `rc` parameter is the error return. In the driver it is tested for an error, but we omit this from the present discussion.

- Read in the `DenseMtx` object for $Y$.

  ```
  mtxY = DenseMtx_new() ;
  rc = DenseMtx_readFromFile(mtxY, mtxFileName) ;
  DenseMtx_dimensions(mtxY, &nrow, &nrhs) ;
  ```

The `nrhs` parameter contains the number of right hand sides, or equivalently, the number of columns in $Y$.

- Create and setup the `BridgeMT` object.

  ```
  bridge = BridgeMT_new() ;
  BridgeMT_setMatrixParams(bridge, neqns, type, symmetryflag) ;
  BridgeMT_setMessageInfo(bridge, msglvl, msgFile) ;
  rc = BridgeMT_setup(bridge, mtxA) ;
  ```

  The `BridgeMT` object is allocated by `BridgeMT_new()`, and various parameters are set. The actual ordering of the matrix, symbolic factorization, and permutation creation are performed inside the `BridgeMT_setup()` method.

- Setup the numerical factorization.

  ```
  rc = BridgeMT_factorSetup(bridge, nthread, 0, 0.0) ;
  ```

  This step tells the `BridgeMT` object the number of threads to be used in the factorization and solve. The third and fourth parameters define the particular type of map of the computations to processors. When the third parameter is zero, the defaults map is used. If `rc = 1`, the setup completed without any error.

- Compute the matrix factorization.

  ```
  permuteflag = 1 ;
  rc = BridgeMT_factor(bridge, mtxA, permuteflag, &error) ;
  ```

  When `permuteflag` is `1`, it means that the matrix in `mtxA` has not yet been permuted into the new ordering and so is done inside the method. The `error` flag is filled with an error code that tells how far the factorization was able to proceed. If `rc = 1`, the factorization completed without any error.

- Setup the solve.

  ```
  rc = BridgeMT_solveSetup(bridge) ;
  ```

  This method sets up the environment for a parallel solve. If `rc = 1`, the setup completed without any error.

- Solve the linear system.

  ```
  mtxX = DenseMtx_new() ;
  DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
  DenseMtx_zero(mtxX) ;
  rc = BridgeMT_solve(bridge, permuteflag, mtxX, mtxY) ;
  ```

  The `DenseMtx` object `mtxX` is created and initialized to be the same type and size as `mtxY`. Its entries are explicitly zeroed (this is not necessary but is a good idea in general). The solution is then solved. Again, note the presence of `permuteflag`. When `1`, `mtxY` needs to be permuted into the new ordering, and `mtxX` is returned in the original ordering.

## 4.2 The `BridgeMT` Data Structure

The `BridgeMT` structure has the following fields.

- Graph parameters:
  - `int neqns` : number of equations, i.e., number of vertices in the graph.
  - `int nedges` : number of edges (includes $(u, v)$, $(v, u)$ and $(u, u)$).
  - `int Neqns` : number of equations in the compressed graph.
  - `int Nedges` : number of edges in the compressed graph.
- Ordering parameters:
  - `int maxdomainsize` : maximum size of a subgraph to not split any further during the nested dissection process.
  - `int maxnzeros` : maximum number of zeros to allow in a front during the supernode amalgamation process.
  - `int maxsize` : maximum size of a front when the fronts are split.
  - `int seed` : random number seed.
  - `double compressCutoff` : if the `Neqns` < `compressCutoff` * `neqns`, then the compressed graph is formed, ordered and used to create the symbolic factorization.
- Matrix parameters:
  - `int type` : type of entries, `SPOOLES_REAL` or `SPOOLES_COMPLEX`, default value is `SPOOLES_REAL`.
  - `int symmetryflag` : type of symmetry for the matrix, `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`, default value is `SPOOLES_SYMMETRIC`.
- Factorization parameters:
  - `int sparsityflag` : `SPOOLES_DENSE_FRONTS` for a direct factorization, or `SPOOLES_SPARSE_FRONTS` for an approximate factorization, default value is `SPOOLES_DENSE_FRONTS`.
  - `int pivotingflag` : `SPOOLES_PIVOTING` for pivoting enabled, or `SPOOLES_NO_PIVOTING` for no pivoting, default value is `SPOOLES_NO_PIVOTING`.
  - `double tau` : used when pivoting is enabled, all entries in $L$ and $U$ have magnitude less than or equal to `tau`, default value is 100.
  - `double droptol` : used for an approximation, all entries in $L$ and $U$ that are kept have magnitude greater than or equal to `droptol`. default value is 0.001.
  - `PatchAndGoInfo *patchinfo` : pointer to an object that controls special factorizations for optimization matrices and singular matrices from structural analysis, default value is `NULL` which means no special action is taken. See the Reference Manual for more information.
- Pointers to objects:
  - `ETree *frontETree` : object that defines the factorizations, e.g., the number of fronts, the tree they form, the number of internal and external rows for each front, and the map from vertices to the front where it is contained.
  - `IVL *symbfacIVL` : object that contains the symbolic factorization of the matrix.
  - `SubMtxManager *mtxmanager` : object that manages the `SubMtx` objects that store the factor entries and are used in the solves.

  - `FrontMtx *frontmtx` : object that stores the $L$, $D$ and $U$ factor matrices.
  - `IV *oldToNewIV` : object that stores old-to-new permutation vector.
  - `IV *newToOldIV` : object that stores new-to-old permutation vector.

- Multithreaded information:
  - `int nthread` : number of threads to be used during the factor and solve.
  - `int lookahead` : this parameter is used to possibly reduce the idle time of threads during the factorization. When `lookahead` is 0, the factorization uses the least amount of working storage but threads can be idle. Larger values of `lookahead` tend to increase the working storage but may decrease the execution time. Values of `lookahead` greater than `nthread` are not useful.
  - `IV *ownersIV` : this object contains the map from fronts to their owning processors.
  - `SolveMap *solvemap` : this object contains the map from factor submatrices to their owning processors.
  - `DV *cumopsDV` : this object is formed when the map from fronts to owning processors is created. Its size is `nthread` and contains the operations that each thread will perform during a direct factorization without pivoting.

- Message information, statistics and cpu times:
  - `int msglvl` : message level for output. When 0, no output, When 1, just statistics and cpu times. When greater than 1, more and more output.
  - `FILE *msgFile` : message file for output. When `msglvl` $> 0$, `msgFile` must not be `NULL`.
  - `int stats[6]` : statistics for the factorization.

| | | | |
|---|---|---|---|
| `stats[0]` : | # of pivots | `stats[3]` : | # of entries in $D$ |
| `stats[1]` : | # of pivot tests | `stats[4]` : | # of entries in $L$ |
| `stats[2]` : | # of delayed rows and columns | `stats[5]` : | # of entries in $U$ |

  - `double cpus[16]` : cpus for the different functions.

| | | | |
|---|---|---|---|
| `cpus[0]` : | time to construct `Graph` | `cpus[8]` : | time to factor matrix |
| `cpus[1]` : | time to compress `Graph` | `cpus[9]` : | time to post-process matrix |
| `cpus[2]` : | time to order `Graph` | `cpus[10]` : | total factor time |
| `cpus[3]` : | time for symbolic factorization | `cpus[11]` : | time to setup the parallel solve |
| `cpus[4]` : | total setup time | `cpus[12]` : | time to permute rhs |
| `cpus[5]` : | time to setup the factorization | `cpus[13]` : | time to solve |
| `cpus[6]` : | time to permute matrix | `cpus[14]` : | time to permute solution |
| `cpus[7]` : | time to initialize front matrix | `cpus[15]` : | total solve time |

## 4.3 Prototypes and descriptions of `BridgeMT` methods

This section contains brief descriptions including prototypes of all methods that belong to the `BridgeMT` object.

### 4.3.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `BridgeMT * BridgeMT_new ( void ) ;`

   This method simply allocates storage for the `BridgeMT` structure and then sets the default fields by a call to `BridgeMT_setDefaultFields()`.

2. `int BridgeMT_setDefaultFields ( BridgeMT *bridge ) ;`

   The structure's fields are set to default values:

   - `neqns` = `nedges` = `Neqns` = `Nedges` = 0.
   - `maxdomainsize` = `maxnzeros` = `maxsize` = `seed` = -1. `compressCutoff` = 0.
   - `type` = `SPOOLES_REAL`.
   - `symmetryflag` = `SPOOLES_SYMMETRIC`.
   - `sparsityflag` = `SPOOLES_DENSE_FRONTS`.
   - `pivotingflag` = `SPOOLES_NO_PIVOTING`.
   - `tau` = 100., `droptol` = 0.001.
   - `lookahead` = `nthread` = 0.
   - `patchinfo`, `frontETree`, `symbfacIVL`, `mtxmanager`, `frontmtx`, `oldToNewIV`, `newToOldIV`, `ownersIV`, `solvemap` and `cumopsDV` are all set to `NULL`.

   The `stats[6]` and `cpus[16]` vectors are filled with zeros.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

3. `int BridgeMT_clearData ( BridgeMT *bridge ) ;`

   This method clears the object and free's any owned data. It then calls `BridgeMT_setDefaultFields()`.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

4. `int BridgeMT_free ( BridgeMT *bridge ) ;`

   This method releases any storage by a call to `BridgeMT_clearData()` and then free the space for `bridge`.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

### 4.3.2 Instance methods

1. `int BridgeMT_oldToNewIV ( BridgeMT *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `oldToNewIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

2. `int BridgeMT_newToOldIV ( BridgeMT *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `newToOldIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

3. `int BridgeMT_frontETree ( BridgeMT *bridge, ETree **pobj ) ;`

   This method fills `*pobj` with its `frontETree` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

4. `int BridgeMT_symbfacIVL ( BridgeMT *bridge, IVL **pobj ) ;`

   This method fills `*pobj` with its `symbfacIVL` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

5. `int BridgeMT_mtxmanager ( BridgeMT *bridge, SubMtxManager **pobj ) ;`

   This method fills `*pobj` with its `mtxmanager` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

6. `int BridgeMT_frontmtx ( BridgeMT *bridge, FrontMtx **pobj ) ;`

   This method fills `*pobj` with its `frontmtx` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

7. `int BridgeMT_ownersIV ( BridgeMT *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `ownersIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

8. `int BridgeMT_solvemap ( BridgeMT *bridge, SolveMap **pobj ) ;`

   This method fills `*pobj` with its `solvemap` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

9. `int BridgeMT_nthread ( BridgeMT *bridge, int *pnthread ) ;`

   This method fills `*pobj` with the number of threads.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pnthread` is `NULL`.

10. `int BridgeMT_lookahead ( BridgeMT *bridge, int *plookahead ) ;`

    This method fills `*pobj` with the lookahead parameter.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `plookahead` is `NULL`.

### 4.3.3 Parameter methods

1. `int BridgeMT_setMatrixParams ( BridgeMT *bridge, int neqns, int type, int symmetryflag ) ;`

   This method sets the number of equations, type of entries, and symmetry type of the matrix.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `type` is invalid |
| -1 | `bridge` is `NULL` | -4 | `symmetryflag` is invalid |
| -2 | `neqns` $\le 0$ | -5 | symmetry flag is Hermitian but type is real |

2. `int BridgeMT_setOrderingParams ( BridgeMT *bridge, int maxdomainsize, int maxnzeros,`
   `int maxsize, int seed, double compressCutoff ) ;`

   This method sets parameters needed for the ordering.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `maxsize` $\le 0$ |
| -1 | `bridge` is `NULL` | -4 | `compressCutoff` $> 1$ |
| -2 | `maxdomainsize` $\le 0$ | | |

3. `int BridgeMT_setFactorParams ( BridgeMT *bridge, int sparsityflag, int pivotingflag,`
   `double tau, double droptol, int lookahead, PatchAndGoInfo *patchinfo ) ;`

   This method sets parameters needed for the factorization.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -4 | `tau` $< 2.0$ |
| -1 | `bridge` is `NULL` | -5 | `droptol` $< 0.0$ |
| -2 | `sparsityflag` is invalid | -6 | `lookahead` $< 0$ |
| -3 | `pivotingflag` is invalid | | |

4. `int BridgeMT_setMessagesInfo ( BridgeMT *bridge, int msglvl, FILE *msgFile ) ;`

   This method sets the message level and file.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`, -2 if `msglvl` $> 0$ and `msgFile` is `NULL`.

### 4.3.4 Setup methods

1. `int BridgeMT_setup ( BridgeMT *bridge, InpMtx *mtxA ) ;`

   This method orders the graph, generates the front tree, computes the symbolic factorization, and creates the two permutation vectors.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`, -2 if `mtxA` is `NULL`.

2. `int BridgeMT_factorStats ( BridgeMT *bridge, int type, int symmetryflag, int *pnfront,`
   `int *pnfactorind, int *pnfactorent, int *pnsolveops, double *pnfactorops ) ;`

   This method takes as input the type and symmetry of the matrix, and fills the pointer fields with the number of fronts, factor indices, factor entries, forward and back solve operations, and factor operations.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -6 | `pnfront` is `NULL` |
| -1 | `bridge` is `NULL` | -7 | `pnfactorind` is `NULL` |
| -2 | `type` is invalid | -8 | `pnfactorent` is `NULL` |
| -3 | `symmetryflag` is invalid | -9 | `pnsolveops` is `NULL` |
| -4 | `type` is real but `symmetryflag` is Hermitian | -10 | `pnfactorops` is `NULL` |
| -5 | front tree is not present | | |

### 4.3.5 Factor methods

1. `int BridgeMT_factorSetup ( BridgeMT *bridge, int nthread, int maptype, double cutoff ) ;`

   This method constructs the map from fronts to owning threads, and computes the number of factor operations that each thread will execute. The `maptype` parameter can be one of four values:

   - 1 — wrap map
   - 2 — balanced map
   - 3 — subtree-subset map
   - 4 — domain decomposition map

   The wrap map and balanced map are not recommended. The subtree-subset map is a good map with a very well balanced nested dissection ordering. The domain decomposition map is recommended when the nested dissection tree is imbalanced or for the multisection ordering. The domain decomposition map requires a `cutoff` parameter in $[0,1]$ which specifies the relative size of a subtree that forms a domain. If `maptype` is not one of 1, 2, 3 or 4, the default map is used: domain decomposition with `cutoff` = 1/(2*`nthread`).

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return, factorization did complete | -2 | `nthread` $< 1$ |
| -1 | `bridge` is `NULL` | -5 | `frontETree` is not present |

2. `int BridgeMT_factor ( BridgeMT *bridge, InpMtx *mtxA, int permuteflag, int *perror ) ;`

   This method permutes the matrix into the new ordering (if `permuteflag` is 1), factors the matrix, and then post-processes the factors.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return, factorization did complete | -1 | `bridge` is `NULL` |
| 0 | factorization did not complete | -2 | `mtxA` is `NULL` |
| | | -3 | `perror` is `NULL` |

### 4.3.6 Solve methods

1. `int BridgeMT_solveSetup ( BridgeMT *bridge ) ;`

   This method creates the `SolveMap` object that governs the parallel solve.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -2 | `frontMtx` is `NULL` |
| -1 | `bridge` is `NULL` | -3 | `frontMtx` needs to be postprocessed |

2. `int BridgeMT_solve ( BridgeMT *bridge, int permuteflag, DenseMtx *mtxX, DenseMtx *mtxY ) ;`

   If `permuteflag` is 1, then `mtxY` is permuted into the new ordering. The linear system $AX = Y$ is solved. If `permuteflag` is 1, then `mtxX` is permuted into the old ordering.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -4 | `frontmtx` is `NULL` |
| -1 | `bridge` is `NULL` | -5 | `mtxmanager` is `NULL` |
| -2 | `X` is `NULL` | -6 | `oldToNewIV` needed, but not available |
| -3 | `Y` is `NULL` | -7 | `newToOldIV` needed, but not available |

# Chapter 5

# The MPI Wrapper Object and Driver

The goal is to solve $AX = Y$ in a distributed environment using MPI. Section 9 of the User's Manual presents a listing of the `AllInOneMPI` driver program for solving $AX = Y$. There are thirteen steps, and each requires "mid-level" knowledge of several objects of the **SPOOLES** library. To reduce the complexity of using the library, we created the `BridgeMPI` object. The term "bridge" symbolizes spanning the distance between the **SPOOLES** library and the CSAR Nastran application code. The ten steps of the `allInOneMPI` driver program is reduced to five using the `BridgeMPI` object.

- Initialization and setup step.

  Here the `BridgeMPI` object is allocated via a call to `BridgeMPI_new()`. Parameters are set using `BridgeMPI_set*()` methods. The setup phase orders the matrix and prepares all the necessary **SPOOLES** data structures for the factorization and solve that follows.

- Setup the numerical factorization.

  In this step, `BridgeMPI_factorSetup()` is called to define the parallelism for the factorization, and all data structures for the parallel execution are created.

- Factorization step.

  The matrix is factored via a call to `BridgeMPI_factor()`.

- Setup the numerical solves.

  `BridgeMPI_solveSetup()` is called to set up the parallel solves. This must be called *once* after a factorization, one or more solves may follow.

- Solution step.

  The linear system is solved via a call to `BridgeMPI_solve()`.

The `BridgeMPI` object has many parameters that control the ordering of the matrix, the pivoting tolerance (if pivoting is requested), the drop tolerance (for an approximate factorization), and so on. Rather than burden the user with the knowledge of and setting these parameters, there are decent default values built into the object. Using the `BridgeMPI` object to solve a linear system of equations can be broken down into three steps.

Section 5.1 takes a quick look at the `BridgeMPI` driver program (whose complete listing is found in Appendix C). Section 5.2 describes the internal data fields of the `BridgeMPI` object. Section 3.3 contains the prototypes and descriptions of all `Bridge` methods.

## 5.1 A quick look at the MPI driver program

The entire listing of this MPI driver is found in Appendix C. We now extract parts of the code.

- Decode the input.

  ```
  msglvl       = atoi(argv[1]) ;
  msgFileName  = argv[6] ;
  neqns        = atoi(argv[3]) ;
  type         = atoi(argv[4]) ;
  symmetryflag = atoi(argv[5]) ;
  mtxFileName  = argv[6] ;
  rhsFileName  = argv[7] ;
  solFileName  = argv[8] ;
  seed         = atoi(argv[9]) ;
  ```

  Here is a description of the input parameters.

  – `msglvl` is the message level.
  – `msgFile` is the message file name
  – `neqns` is the number of equations.
  – `type` is the type of entries: 1 (`SPOOLES_REAL`) or 2 (`SPOOLES_COMPLEX`).
  – `symmetryflag` is the type of matrix symmetry: 0 (`SPOOLES_SYMMETRIC`), 1 (`SPOOLES_HERMITIAN`) or 2 (`SPOOLES_NONSYMMETRIC`).
  – `mtxFile` is the name of the file from which to read the `InpMtx` object for $A$. The file name must have the form `*.inpmtxb` for a binary file or `*.inpmtxf` for a formatted file.
  – `rhsFile` is the name of the file from which to read the `DenseMtx` object for the right hand side $Y$. The file name must have the form `*.densemtxb` for a binary file or `*.densemtxf` for a formatted file.
  – `solFile` is the name of the file to write the `DenseMtx` object for the solution $X$. The file name must have the form `*.densemtxb` for a binary file or `*.densemtxf` for a formatted file, `"none"` for no output, or any other name for a human-readable listing.
  – `seed` is a random number seed used in the ordering process.

- Processor 0 reads in the `InpMtx` object for $A$.

  ```
  mtxA = InpMtx_new() ;
  rc = InpMtx_readFromFile(mtxA, mtxFileName) ;
  ```

  The `rc` parameter is the error return. Processor 0 then broadcasts the error return to the other processors. If an error occured reading in the matrix, all processors call `MPI_Finalize()` and exit.

- Processor 0 reads in the `DenseMtx` object for $Y$.

  ```
  mtxY = DenseMtx_new() ;
  rc = DenseMtx_readFromFile(mtxY, mtxFileName) ;
  DenseMtx_dimensions(mtxY, &nrow, &nrhs) ;
  ```

  The `nrhs` parameter contains the number of right hand sides, or equivalently, the number of columns in $Y$. Processor 0 then broadcasts the error return to the other processors. If an error occured reading in the matrix, all processors call `MPI_Finalize()` and exit.

- Create and setup the `BridgeMPI` object.

  ```
  bridge = BridgeMPI_new() ;
  BridgeMPI_setMPIparams(bridge, nproc, myid, MPI_COMM_WORLD) ;
  BridgeMPI_setMatrixParams(bridge, neqns, type, symmetryflag) ;
  BridgeMPI_setMessageInfo(bridge, msglvl, msgFile) ;
  rc = BridgeMPI_setup(bridge, mtxA) ;
  ```

  The `BridgeMPI` object is allocated by `BridgeMPI_new()`, and various parameters are set. The actual ordering of the matrix, symbolic factorization, and permutation creation are performed inside the `BridgeMPI_setup()` method.

- Setup the numerical factorization.

  ```
  rc = BridgeMPI_factorSetup(bridge, 0, 0.0) ;
  ```

  This step tells the `BridgeMPI` object the number of threads to be used in the factorization and solve. The second and third parameters define the particular type of map of the computations to processors. When the second parameter is zero, the defaults map is used. If `rc = 1`, the setup completed without any error.

- Compute the matrix factorization.

  ```
  permuteflag = 1 ;
  rc = BridgeMPI_factor(bridge, mtxA, permuteflag, &error) ;
  ```

  When `permuteflag` is `1`, it means that the matrix in `mtxA` has not yet been permuted into the new ordering and so is done inside the method. The `error` flag is filled with an error code that tells how far the factorization was able to proceed. If `rc = 1`, the factorization completed without any error.

- Setup the solve.

  ```
  rc = BridgeMPI_solveSetup(bridge) ;
  ```

  This method sets up the environment for a parallel solve. It is called once per factorization, not once per solve. If `rc = 1`, the setup completed without any error.

- Solve the linear system. Processor 0 initializes the `DenseMtx` object `mtxX` to hold the global solution $X$. Its entries are explicitly zeroed (this is not necessary but is a good idea in general). The solution is then solved.

  ```
  mtxX = DenseMtx_new() ;
  DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
  DenseMtx_zero(mtxX) ;
  ```

  All processors then cooperate to compute the solution $X$.

  ```
  rc = BridgeMPI_solve(bridge, permuteflag, mtxX, mtxY) ;
  ```

  Again, note the presence of `permuteflag`. When `1`, `mtxY` needs to be permuted into the new ordering, and `mtxX` is returned in the original ordering.

## 5.2 The `BridgeMPI` Data Structure

The `BridgeMPI` structure has the following fields.

- Graph parameters:
  - `int neqns` : number of equations, i.e., number of vertices in the graph.
  - `int nedges` : number of edges (includes $(u, v)$, $(v, u)$ and $(u, u)$).
  - `int Neqns` : number of equations in the compressed graph.
  - `int Nedges` : number of edges in the compressed graph.
- Ordering parameters:
  - `int maxdomainsize` : maximum size of a subgraph to not split any further during the nested dissection process.
  - `int maxnzeros` : maximum number of zeros to allow in a front during the supernode amalgamation process.
  - `int maxsize` : maximum size of a front when the fronts are split.
  - `int seed` : random number seed.
  - `double compressCutoff` : if the `Neqns` $<$ `compressCutoff` $*$ `neqns`, then the compressed graph is formed, ordered and used to create the symbolic factorization.
- Matrix parameters:
  - `int type` : type of entries, `SPOOLES_REAL` or `SPOOLES_COMPLEX`, default value is `SPOOLES_REAL`.
  - `int symmetryflag` : type of symmetry for the matrix, `SPOOLES_SYMMETRIC`, `SPOOLES_HERMITIAN` or `SPOOLES_NONSYMMETRIC`, default value is `SPOOLES_SYMMETRIC`.
- Factorization parameters:
  - `int sparsityflag` : `SPOOLES_DENSE_FRONTS` for a direct factorization, or `SPOOLES_SPARSE_FRONTS` for an approximate factorization, default value is `SPOOLES_DENSE_FRONTS`.
  - `int pivotingflag` : `SPOOLES_PIVOTING` for pivoting enabled, or `SPOOLES_NO_PIVOTING` for no pivoting, default value is `SPOOLES_NO_PIVOTING`.
  - `double tau` : used when pivoting is enabled, all entries in $L$ and $U$ have magnitude less than or equal to `tau`, default value is 100.
  - `double droptol` : used for an approximation, all entries in $L$ and $U$ that are kept have magnitude greater than or equal to `droptol`. default value is 0.001.
  - `PatchAndGoInfo *patchinfo` : pointer to an object that controls special factorizations for optimization matrices and singular matrices from structural analysis, default value is `NULL` which means no special action is taken. See the Reference Manual for more information.
  - `int lookahead` : this parameter is used to possibly reduce the idle time of threads during the factorization. When `lookahead` is 0, the factorization uses the least amount of working storage but threads can be idle. Larger values of `lookahead` tend to increase the working storage but may decrease the execution time. Values of `lookahead` greater than `nthread` are not useful.
- Pointers to objects:
  - `ETree *frontETree` : object that defines the factorizations, e.g., the number of fronts, the tree they form, the number of internal and external rows for each front, and the map from vertices to the front where it is contained.

  - `IVL *symbfacIVL` : object that contains the symbolic factorization of the matrix.
  - `SubMtxManager *mtxmanager` : object that manages the `SubMtx` objects that store the factor entries and are used in the solves.
  - `FrontMtx *frontmtx` : object that stores the $L$, $D$ and $U$ factor matrices.
  - `IV *oldToNewIV` : object that stores old-to-new permutation vector.
  - `IV *newToOldIV` : object that stores new-to-old permutation vector.

- MPI information:
  - `int nproc` : number of processors.
  - `int myid` : id of this processor.
  - `MPI_Comm` : MPI communicator.
  - `IV *ownersIV` : this object contains the map from fronts to their owning processors.
  - `SolveMap *solvemap` : this object contains the map from factor submatrices to their owning processors.
  - `DV *cumopsDV` : this object is formed when the map from fronts to owning processors is created. Its size is `nthread` and contains the operations that each thread will perform during a direct factorization without pivoting.
  - `IV *vtxmapIV` : this object contains the map from vertices to their owning processors.
  - `IV *rowmapIV` : this object contains the map from rows to their owning processors during the solve. This may be different from `vtxmapIV` if pivoting is enabled.
  - `IV *ownedColumnsIV` : this object contains the columns of the matrix that are owned by this processor during the solve.
  - `InpMtx *Aloc` : this object contains the entries of $A$ that are local to this processor during the factorization.
  - `DenseMtx *Xloc` : this object contains the local solution during the solve.
  - `DenseMtx *Yloc` : this object contains the local right hand side during the solve.

- Message information, statistics and cpu times:
  - `int msglvl` : message level for output. When 0, no output, When 1, just statistics and cpu times. When greater than 1, more and more output.
  - `FILE *msgFile` : message file for output. When `msglvl` $> 0$, `msgFile` must not be `NULL`.
  - `int stats[6]` : statistics for the factorization.

| | | | |
|---|---|---|---|
| `stats[0]` : | # of pivots | `stats[3]` : | # of entries in $D$ |
| `stats[1]` : | # of pivot tests | `stats[4]` : | # of entries in $L$ |
| `stats[2]` : | # of delayed rows and columns | `stats[5]` : | # of entries in $U$ |

  - `double cpus[22]` : cpus for the different functions.

| | | | |
|---|---|---|---|
| `cpus[0]` : | construct `Graph` | `cpus[11]` : | factor matrix |
| `cpus[1]` : | compress `Graph` | `cpus[12]` : | post-process matrix |
| `cpus[2]` : | order `Graph` | `cpus[13]` : | total factor time |
| `cpus[3]` : | symbolic factorization | `cpus[14]` : | setup the parallel solve |
| `cpus[4]` : | broadcast the front tree | `cpus[15]` : | permute rhs |
| `cpus[5]` : | broadcast symbolic factor | `cpus[16]` : | distribute rhs |
| `cpus[6]` : | total setup time | `cpus[17]` : | create solution matrix |
| `cpus[7]` : | setup the factorization | `cpus[18]` : | solve |
| `cpus[8]` : | permute matrix | `cpus[19]` : | gather solution |
| `cpus[9]` : | distribute matrix | `cpus[20]` : | permute solution |
| `cpus[10]` : | initialize front matrix | `cpus[21]` : | total solve time |

## 5.3 Prototypes and descriptions of `BridgeMPI` methods

This section contains brief descriptions including prototypes of all methods that belong to the `BridgeMPI` object.

### 5.3.1 Basic methods

As usual, there are four basic methods to support object creation, setting default fields, clearing any allocated data, and free'ing the object.

1. `BridgeMPI * BridgeMPI_new ( void ) ;`

   This method simply allocates storage for the `BridgeMPI` structure and then sets the default fields by a call to `BridgeMPI_setDefaultFields()`.

2. `int BridgeMPI_setDefaultFields ( BridgeMPI *bridge ) ;`

   The structure's fields are set to default values:

   - `neqns` = `nedges` = `Neqns` = `Nedges` = 0.
   - `maxdomainsize` = `maxnzeros` = `maxsize` = `seed` = -1. `compressCutoff` = 0.
   - `type` = `SPOOLES_REAL`.
   - `symmetryflag` = `SPOOLES_SYMMETRIC`.
   - `sparsityflag` = `SPOOLES_DENSE_FRONTS`.
   - `pivotingflag` = `SPOOLES_NO_PIVOTING`.
   - `tau` = 100., `droptol` = 0.001.
   - `lookahead` = `nproc` = 0.
   - `myid` = -1.
   - `patchinfo`, `frontETree`, `symbfacIVL`, `mtxmanager`, `frontmtx`, `oldToNewIV`, `newToOldIV`, `ownersIV`, `solvemap`, `cumopsDV`, `vtxmapIV`, `rowmapIV`, `ownedColumnsIV`, `Aloc`, `Xloc`, `Yloc` and `comm` are all set to `NULL`.

   The `stats[6]` and `cpus[22]` vectors are filled with zeros.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

3. `int BridgeMPI_clearData ( BridgeMPI *bridge ) ;`

   This method clears the object and free's any owned data. It then calls `BridgeMPI_setDefaultFields()`.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

4. `int BridgeMPI_free ( BridgeMPI *bridge ) ;`

   This method releases any storage by a call to `BridgeMPI_clearData()` and then free the space for `bridge`.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`.

### 5.3.2 Instance methods

1. `int BridgeMPI_oldToNewIV ( BridgeMPI *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `oldToNewIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

2. `int BridgeMPI_newToOldIV ( BridgeMPI *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `newToOldIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

3. `int BridgeMPI_frontETree ( BridgeMPI *bridge, ETree **pobj ) ;`

   This method fills `*pobj` with its `frontETree` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

4. `int BridgeMPI_symbfacIVL ( BridgeMPI *bridge, IVL **pobj ) ;`

   This method fills `*pobj` with its `symbfacIVL` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

5. `int BridgeMPI_mtxmanager ( BridgeMPI *bridge, SubMtxManager **pobj ) ;`

   This method fills `*pobj` with its `mtxmanager` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

6. `int BridgeMPI_frontmtx ( BridgeMPI *bridge, FrontMtx **pobj ) ;`

   This method fills `*pobj` with its `frontmtx` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

7. `int BridgeMPI_ownersIV ( BridgeMPI *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `ownersIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

8. `int BridgeMPI_solvemap ( BridgeMPI *bridge, SolveMap **pobj ) ;`

   This method fills `*pobj` with its `solvemap` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

9. `int BridgeMPI_vtxmapIV ( BridgeMPI *bridge, IV **pobj ) ;`

   This method fills `*pobj` with its `vtxmapIV` pointer.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

10. `int BridgeMPI_rowmapIV ( BridgeMPI *bridge, IV **pobj ) ;`

    This method fills `*pobj` with its `rowmapIV` pointer.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

11. `int BridgeMPI_ownedColumns ( BridgeMPI *bridge, IV **pobj ) ;`

    This method fills `*pobj` with its `ownedColumnsIV` pointer.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

12. `int BridgeMPI_Xloc ( BridgeMPI *bridge, DenseMtx **pobj ) ;`

    This method fills `*pobj` with its `Xloc` pointer.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

13. `int BridgeMPI_Yloc ( BridgeMPI *bridge, DenseMtx **pobj ) ;`

    This method fills `*pobj` with its `Yloc` pointer.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pobj` is `NULL`.

14. `int BridgeMPI_nproc ( BridgeMPI *bridge, int *pnproc ) ;`

    This method fills `*pobj` with the number of processors.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pnproc` is `NULL`.

15. `int BridgeMPI_myid ( BridgeMPI *bridge, int *pmyid ) ;`

    This method fills `*pobj` with the id of this processor.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `pmyid` is `NULL`.

16. `int BridgeMPI_lookahead ( BridgeMPI *bridge, int *plookahead ) ;`

    This method fills `*pobj` with the lookahead parameter.

    *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`. -2 if `plookahead` is `NULL`.

### 5.3.3 Parameter methods

1. `int BridgeMPI_setMatrixParams ( BridgeMPI *bridge, int neqns, int type, int symmetryflag ) ;`

   This method sets the number of equations, type of entries, and symmetry type of the matrix.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `type` is invalid |
| -1 | `bridge` is `NULL` | -4 | `symmetryflag` is invalid |
| -2 | `neqns` $\leq 0$ | -5 | symmetry flag is Hermitian but type is real |

2. `int BridgeMPI_setMPIparams ( BridgeMPI *bridge, int nproc, int myid, MPI_Comm comm ) ;`

   This method sets the MPI environment parameters.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -2 | `nproc` $\leq 0$ |
| -1 | `bridge` is `NULL` | -3 | `myid` $< 0$ or $>=$ `nproc` |

3. `int BridgeMPI_setOrderingParams ( BridgeMPI *bridge, int maxdomainsize, int maxnzeros,`
   `                                int maxsize, int seed, double compressCutoff ) ;`

   This method sets parameters needed for the ordering.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -3 | `maxsize` $\leq 0$ |
| -1 | `bridge` is `NULL` | -4 | `compressCutoff` $> 1$ |
| -2 | `maxdomainsize` $\leq 0$ | | |

4. `int BridgeMPI_setFactorParams ( BridgeMPI *bridge, int sparsityflag, int pivotingflag, double tau, double droptol, int lookahead, PatchAndGoInfo *patchinfo ) ;`

   This method sets parameters needed for the factorization.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -4 | `tau` $< 2.0$ |
| -1 | `bridge` is `NULL` | -5 | `droptol` $< 0.0$ |
| -2 | `sparsityflag` is invalid | -6 | `lookahead` $< 0$ |
| -3 | `pivotingflag` is invalid | | |

5. `int BridgeMPI_setMessagesInfo ( BridgeMPI *bridge, int msglvl, FILE *msgFile ) ;`

   This method sets the message level and file.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`, -2 if `msglvl` $> 0$ and `msgFile` is `NULL`.

### 5.3.4 Setup methods

1. `int BridgeMPI_setup ( BridgeMPI *bridge, InpMtx *mtxA ) ;`

   This method orders the graph, generates the front tree, computes the symbolic factorization, and creates the two permutation vectors.

   *Return value:* 1 for a normal return, -1 if `bridge` is `NULL`, -2 if `mtxA` is `NULL`.

2. `int BridgeMPI_factorStats ( BridgeMPI *bridge, int type, int symmetryflag, int *pnfront, int *pnfactorind, int *pnfactorent, int *pnsolveops, double *pnfactorops ) ;`

   This method takes as input the type and symmetry of the matrix, and fills the pointer fields with the number of fronts, factor indices, factor entries, forward and back solve operations, and factor operations.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -6 | `pnfront` is `NULL` |
| -1 | `bridge` is `NULL` | -7 | `pnfactorind` is `NULL` |
| -2 | `type` is invalid | -8 | `pnfactorent` is `NULL` |
| -3 | `symmetryflag` is invalid | -9 | `pnsolveops` is `NULL` |
| -4 | `type` is real but `symmetryflag` is Hermitian | -10 | `pnfactorops` is `NULL` |
| -5 | front tree is not present | | |

### 5.3.5 Factor methods

1. `int BridgeMPI_factorSetup ( BridgeMPI *bridge, int maptype, double cutoff ) ;`

   This method constructs the map from fronts to owning processors, and computes the number of factor operations that each thread will execute. The `maptype` parameter can be one of four values:

   - 1 — wrap map
   - 2 — balanced map
   - 3 — subtree-subset map
   - 4 — domain decomposition map

The wrap map and balanced map are not recommended. The subtree-subset map is a good map with a very well balanced nested dissection ordering. The domain decomposition map is recommended when the nested dissection tree is imbalanced or for the multisection ordering. The domain decomposition map requires a `cutoff` parameter in $[0, 1]$ which specifies the relative size of a subtree that forms a domain. If `maptype` is not one of 1, 2, 3 or 4, the default map is used: domain decomposition with `cutoff` = 1/(2*`nthread`).

*Return value:* 1 normal return, factorization did complete, -1 `bridge` is `NULL`, -2 `frontETree` is not present.

2. `int BridgeMPI_factor ( BridgeMPI *bridge, InpMtx *mtxA, int permuteflag, int *perror ) ;`

   This method permutes the matrix into the new ordering (if `permuteflag` is 1), factors the matrix, and then post-processes the factors.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return, factorization did complete | -1 | `bridge` is `NULL` |
| 0 | factorization did not complete | -2 | `mtxA` is `NULL` |
| | | -3 | `perror` is `NULL` |

### 5.3.6 Solve methods

1. `int BridgeMPI_solveSetup ( BridgeMPI *bridge ) ;`

   This method creates the `SolveMap` object that governs the parallel solve.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -2 | `frontMtx` is `NULL` |
| -1 | `bridge` is `NULL` | -3 | `frontMtx` needs to be postprocessed |

2. `int BridgeMPI_solve ( BridgeMPI *bridge, int permuteflag, DenseMtx *mtxX, DenseMtx *mtxY ) ;`

   If `permuteflag` is 1, then `mtxY` is permuted into the new ordering. The linear system $AX = Y$ is solved. If `permuteflag` is 1, then `mtxX` is permuted into the old ordering.

   *Return value:*

| | | | |
|---|---|---|---|
| 1 | normal return | -4 | `frontmtx` is `NULL` |
| -1 | `bridge` is `NULL` | -5 | `mtxmanager` is `NULL` |
| -2 | `X` is `NULL` | -6 | `oldToNewIV` needed, but not available |
| -3 | `Y` is `NULL` | -7 | `newToOldIV` needed, but not available |

# Appendix A

# testWrapper.c — A Serial Driver Program

```
/*  testWrapper.c  */

#include "../Bridge.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   -----------------------------------------------------------
   purpose -- main driver program to solve a linear system
      where the matrix and rhs are read in from files
      and the solution is written to a file.

   created -- 98oct31, cca
   -----------------------------------------------------------
*/
Bridge     *bridge ;
char       *mtxFileName, *rhsFileName, *solFileName ;
double     nfactorops ;
FILE       *msgFile ;
InpMtx     *mtxA ;
int        error, msglvl, neqns, nfent, nfind, nfront, nrhs, nrow,
           nsolveops, permuteflag, rc, seed, symmetryflag, type ;
DenseMtx   *mtxX, *mtxY ;
/*--------------------------------------------------------------------*/
/*
    --------------------
    get input parameters
    --------------------
*/
if ( argc != 10 ) {
   fprintf(stdout,
           "\n\n usage : %s msglvl msgFile neqns type symmetryflag"
           "\n        mtxFile rhsFile seed"
           "\n   msglvl  -- message level"
           "\n      0 -- no output"
           "\n      1 -- timings and statistics"
           "\n      2 and greater -- lots of output"
           "\n   msgFile -- message file"
           "\n   neqns   -- # of equations"
           "\n   type    -- type of entries"
           "\n      1 -- real"
           "\n      2 -- complex"
           "\n   symmetryflag -- symmetry flag"
           "\n      0 -- symmetric"
           "\n      1 -- hermitian"
           "\n      2 -- nonsymmetric"
           "\n   neqns   -- # of equations"
           "\n   mtxFile -- input file for A matrix InpMtx object"
           "\n      must be *.inpmtxf or *.inpmtxb"
           "\n   rhsFile -- input file for Y DenseMtx object"
           "\n      must be *.densemtxf or *.densemtxb"
           "\n   solFile -- output file for X DenseMtx object"
           "\n      must be none, *.densemtxf or *.densemtxb"
           "\n   seed -- random number seed"
           "\n",
           argv[0]) ;
   return(0) ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else if ( (msgFile = fopen(argv[2], "w")) == NULL ) {
   fprintf(stderr, "\n fatal error in %s"
           "\n unable to open file %s\n",
           argv[0], argv[2]) ;
   return(-1) ;
}
neqns        = atoi(argv[3]) ;
type         = atoi(argv[4]) ;
symmetryflag = atoi(argv[5]) ;
mtxFileName = argv[6] ;
rhsFileName = argv[7] ;
solFileName = argv[8] ;
seed         = atoi(argv[9]) ;
fprintf(msgFile,
        "\n\n %s input :"
        "\n msglvl       = %d"
        "\n msgFile      = %s"
        "\n neqns        = %d"
        "\n type         = %d"
        "\n symmetryflag = %d"
        "\n mtxFile      = %s"
        "\n rhsFile      = %s"
        "\n solFile      = %s"
        "\n seed         = %d"
        "\n",
        argv[0], msglvl, argv[2], neqns, type, symmetryflag,
        mtxFileName, rhsFileName, solFileName, seed) ;

/*--------------------------------------------------------------------*/
/*
   ------------------
   read in the matrix
   ------------------
*/
mtxA = InpMtx_new() ;
rc = InpMtx_readFromFile(mtxA, mtxFileName) ;
if ( rc != 1 ) {
   fprintf(msgFile, "\n fatal error reading mtxA from file %s, rc = %d",
           mtxFileName, rc) ;
   fflush(msgFile) ;
   exit(-1) ;
}
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n InpMtx object ") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------
   read in the right hand side matrix
   ----------------------------------
*/
mtxY = DenseMtx_new() ;
rc = DenseMtx_readFromFile(mtxY, rhsFileName) ;
if ( rc != 1 ) {
   fprintf(msgFile, "\n fatal error reading mtxY from file %s, rc = %d",
           rhsFileName, rc) ;
   fflush(msgFile) ;
   exit(-1) ;
}
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n DenseMtx object for right hand side") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
DenseMtx_dimensions(mtxY, &nrow, &nrhs) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------
   create and setup a Bridge object
   --------------------------------
*/
```

```
bridge = Bridge_new() ;
Bridge_setMatrixParams(bridge, neqns, type, symmetryflag) ;
Bridge_setMessageInfo(bridge, msglvl, msgFile) ;
rc = Bridge_setup(bridge, mtxA) ;
if ( rc != 1 ) {
   fprintf(stderr, "\n error return %d from Bridge_setup()", rc) ;
   exit(-1) ;
}
fprintf(msgFile, "\n\n ----- SETUP -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to construct Graph"
        "\n    CPU %8.3f : time to compress Graph"
        "\n    CPU %8.3f : time to order Graph"
        "\n    CPU %8.3f : time for symbolic factorization"
        "\n CPU %8.3f : total setup time\n",
        bridge->cpus[0], bridge->cpus[1],
        bridge->cpus[2], bridge->cpus[3], bridge->cpus[4]) ;
rc = Bridge_factorStats(bridge, type, symmetryflag, &nfront,
                        &nfind, &nfent, &nsolveops, &nfactorops) ;
if ( rc != 1 ) {
   fprintf(stderr,
           "\n error return %d from Bridge_factorStats()", rc) ;
   exit(-1) ;
}
fprintf(msgFile,
        "\n\n factor matrix statistics"
        "\n %d fronts, %d indices, %d entries"
        "\n %d solve operations, %12.4e factor operations",
        nfront, nfind, nfent, nsolveops, nfactorops) ;
fflush(msgFile) ;
/*--------------------------------------------------------------------*/
/*
   -----------------
   factor the matrix
   -----------------
*/
permuteflag = 1 ;
rc = Bridge_factor(bridge, mtxA, permuteflag, &error) ;
if ( rc == 1 ) {
   fprintf(msgFile, "\n\n factorization completed successfully\n") ;
} else {
   fprintf(msgFile, "\n return code from factorization = %d"
                    "\n error code                     = %d",
           rc, error) ;
   exit(-1) ;
}
fprintf(msgFile, "\n\n ----- FACTORIZATION -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to permute original matrix"
        "\n    CPU %8.3f : time to initialize factor matrix"
        "\n    CPU %8.3f : time to compute factorization"
        "\n    CPU %8.3f : time to post-process factorization"
        "\n CPU %8.3f : total factorization time\n",
        bridge->cpus[5], bridge->cpus[6], bridge->cpus[7],
        bridge->cpus[8], bridge->cpus[9]) ;
fprintf(msgFile, "\n\n factorization statistics"
        "\n %d pivots, %d pivot tests, %d delayed vertices"
        "\n %d entries in D, %d entries in L, %d entries in U",
        bridge->stats[0], bridge->stats[1], bridge->stats[2],
        bridge->stats[3], bridge->stats[4], bridge->stats[5]) ;
fprintf(msgFile,
        "\n\n factorization: raw mflops %8.3f, overall mflops %8.3f",
        1.e-6*nfactorops/bridge->cpus[7],
        1.e-6*nfactorops/bridge->cpus[9]) ;
fflush(msgFile) ;
/*--------------------------------------------------------------------*/
/*
   ----------------
   solve the system
   ----------------
*/
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
rc = Bridge_solve(bridge, permuteflag, mtxX, mtxY) ;
if ( rc == 1 ) {
   fprintf(msgFile, "\n\n solve completed successfully\n") ;
} else {
   fprintf(msgFile, "\n" " return code from solve = %d\n", rc) ;
   exit(-1) ;
}
fprintf(msgFile, "\n\n ----- SOLVE -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to permute rhs into new ordering"
        "\n    CPU %8.3f : time to solve linear system"
        "\n    CPU %8.3f : time to permute solution into old ordering"
        "\n CPU %8.3f : total solve time\n",
        bridge->cpus[10], bridge->cpus[11],
        bridge->cpus[12], bridge->cpus[13]) ;
fprintf(msgFile, "\n\n solve: raw mflops %8.3f, overall mflops %8.3f",
        1.e-6*nsolveops/bridge->cpus[11],
        1.e-6*nsolveops/bridge->cpus[13]) ;
fflush(msgFile) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n solution matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
if ( strcmp(solFileName, "none") != 0 ) {
/*
   -----------------------------------
   write the solution matrix to a file
   -----------------------------------
*/
   rc = DenseMtx_writeToFile(mtxX, solFileName) ;
   if ( rc != 1 ) {
      fprintf(msgFile,
              "\n fatal error writing mtxX to file %s, rc = %d",
              solFileName, rc) ;
      fflush(msgFile) ;
      exit(-1) ;
   }
}
/*--------------------------------------------------------------------*/
/*
   ---------------------
   free the working data
   ---------------------
*/
InpMtx_free(mtxA) ;
DenseMtx_free(mtxX) ;
DenseMtx_free(mtxY) ;
Bridge_free(bridge) ;

/*--------------------------------------------------------------------*/

return(1) ; }

/*--------------------------------------------------------------------*/
```

# Appendix B

# testWrapperMT.c — A Multithreaded Driver Program

```
/*  testWrapperMT.c  */

#include "../BridgeMT.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   ------------------------------------------------------------
   purpose -- main driver program to solve a linear system
      where the matrix and rhs are read in from files and
      the solution is written to a file.
      NOTE: multithreaded version

   created -- 98sep24, cca
   ------------------------------------------------------------
*/
BridgeMT   *bridge ;
char       *mtxFileName, *rhsFileName, *solFileName ;
double     nfactorops ;
FILE       *msgFile ;
InpMtx     *mtxA ;
int        error, msglvl, neqns, nfent, nfind, nfront, nrhs, nrow,
           nsolveops, nthread, permuteflag, rc, seed, symmetryflag,
           type ;
DenseMtx   *mtxX, *mtxY ;
/*--------------------------------------------------------------------*/
/*
    --------------------
    get input parameters
    --------------------
*/
if ( argc != 11 ) {
   fprintf(stdout,
         "\n\n usage : %s msglvl msgFile neqns type symmetryflag "
         "\n         mtxFile rhsFile solFile seed nthread\n"
         "\n   msglvl  -- message level"
         "\n      0 -- no output"
         "\n      1 -- timings and statistics"
         "\n      2 and greater -- lots of output"
         "\n   msgFile -- message file"
         "\n   neqns   -- # of equations"
         "\n   type    -- type of entries"
         "\n      1 -- real"
         "\n      2 -- complex"
         "\n   symmetryflag -- symmetry flag"
         "\n      0 -- symmetric"
         "\n      1 -- hermitian"
         "\n      2 -- nonsymmetric"
         "\n   neqns   -- # of equations"
         "\n   mtxFile -- input file for A matrix InpMtx object"
         "\n      must be *.inpmtxf or *.inpmtxb"
         "\n   rhsFile -- input file for Y DenseMtx object"
         "\n      must be *.densemtxf or *.densemtxb"
         "\n   solFile -- output file for X DenseMtx object"
         "\n      must be none, *.densemtxf or *.densemtxb"
         "\n   seed -- random number seed"
         "\n   nthread -- number of threads"
         "\n",
         argv[0]) ;
   return(0) ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else if ( (msgFile = fopen(argv[2], "w")) == NULL ) {
   fprintf(stderr, "\n fatal error in %s"
           "\n unable to open file %s\n",
           argv[0], argv[2]) ;
   return(-1) ;
}
neqns        = atoi(argv[3]) ;
type         = atoi(argv[4]) ;
symmetryflag = atoi(argv[5]) ;
mtxFileName  = argv[6] ;
rhsFileName  = argv[7] ;
solFileName  = argv[8] ;
seed         = atoi(argv[9]) ;
nthread      = atoi(argv[10]) ;
fprintf(msgFile,
        "\n\n %s input :"
        "\n msglvl       = %d"
        "\n msgFile      = %s"
        "\n neqns        = %d"
        "\n type         = %d"
        "\n symmetryflag = %d"
        "\n mtxFile      = %s"
        "\n rhsFile      = %s"
        "\n solFile      = %s"
        "\n nthread      = %d"
        "\n",
        argv[0], msglvl, argv[2], neqns, type, symmetryflag,
        mtxFileName, rhsFileName, solFileName, nthread) ;
/*--------------------------------------------------------------------*/
/*
   ------------------
   read in the matrix
   ------------------
*/
mtxA = InpMtx_new() ;
rc = InpMtx_readFromFile(mtxA, mtxFileName) ;
if ( rc != 1 ) {
   fprintf(msgFile, "\n fatal error reading mtxA from file %s, rc = %d",
           mtxFileName, rc) ;
   fflush(msgFile) ;
   exit(-1) ;
}
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n InpMtx object ") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------
   read in the right hand side matrix
   ----------------------------------
*/
mtxY = DenseMtx_new() ;
rc = DenseMtx_readFromFile(mtxY, rhsFileName) ;
if ( rc != 1 ) {
   fprintf(msgFile, "\n fatal error reading mtxY from file %s, rc = %d",
           rhsFileName, rc) ;
   fflush(msgFile) ;
   exit(-1) ;
}
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n DenseMtx object for right hand side") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
DenseMtx_dimensions(mtxY, &nrow, &nrhs) ;
/*--------------------------------------------------------------------*/
/*
   ----------------------------------
   create and setup a BridgeMT object
   ----------------------------------
*/
bridge = BridgeMT_new() ;
BridgeMT_setMatrixParams(bridge, neqns, type, symmetryflag) ;
BridgeMT_setMessageInfo(bridge, msglvl, msgFile) ;
rc = BridgeMT_setup(bridge, mtxA) ;
if ( rc != 1 ) {
   fprintf(stderr, "\n error return %d from BridgeMT_setup()", rc) ;
   exit(-1) ;
}
fprintf(msgFile, "\n\n ----- SETUP -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to construct Graph"
        "\n    CPU %8.3f : time to compress Graph"
        "\n    CPU %8.3f : time to order Graph"
        "\n    CPU %8.3f : time for symbolic factorization"
        "\n CPU %8.3f : total setup time\n",
        bridge->cpus[0],
        bridge->cpus[1],
        bridge->cpus[2],
        bridge->cpus[3],
        bridge->cpus[4]) ;
rc = BridgeMT_factorStats(bridge, type, symmetryflag, &nfront,
                          &nfind, &nfent, &nsolveops, &nfactorops) ;
if ( rc != 1 ) {
   fprintf(stderr,
           "\n error return %d from BridgeMT_factorStats()", rc) ;
   exit(-1) ;
}
fprintf(msgFile,
        "\n\n factor matrix statistics"
        "\n %d fronts, %d indices, %d entries"
        "\n %d solve operations, %12.4e factor operations",
        nfront, nfind, nfent, nsolveops, nfactorops) ;
fflush(msgFile) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------
   setup the parallel factorization
   --------------------------------
*/
rc = BridgeMT_factorSetup(bridge, nthread, 0, 0.0) ;
fprintf(msgFile, "\n\n ----- PARALLEL FACTOR SETUP -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to setup parallel factorization",
        bridge->cpus[5]) ;
if ( msglvl > 0 ) {
   fprintf(msgFile, "\n total factor operations = %.0f",
           DV_sum(bridge->cumopsDV)) ;
   fprintf(msgFile,
           "\n upper bound on speedup due to load balance = %.2f",
           DV_sum(bridge->cumopsDV)/DV_max(bridge->cumopsDV)) ;
   fprintf(msgFile, "\n operations distributions over threads") ;
   DV_writeForHumanEye(bridge->cumopsDV, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------
   factor the matrix
   -----------------
*/
permuteflag  = 1 ;
rc = BridgeMT_factor(bridge, mtxA, permuteflag, &error) ;
if ( rc == 1 ) {
   fprintf(msgFile, "\n\n factorization completed successfully\n") ;
} else {
   fprintf(msgFile,
           "\n return code from factorization = %d\n"
           "\n error code                     = %d\n",
           rc, error) ;
   exit(-1) ;
}
fprintf(msgFile, "\n\n ----- FACTORIZATION -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to permute original matrix"
        "\n    CPU %8.3f : time to initialize factor matrix"
        "\n    CPU %8.3f : time to compute factorization"
        "\n    CPU %8.3f : time to post-process factorization"
        "\n CPU %8.3f : total factorization time\n",
        bridge->cpus[6],
        bridge->cpus[7],
        bridge->cpus[8],
        bridge->cpus[9],
        bridge->cpus[10]) ;
fprintf(msgFile, "\n\n factorization statistics"
        "\n %d pivots, %d pivot tests, %d delayed vertices"
        "\n %d entries in D, %d entries in L, %d entries in U",
        bridge->stats[0], bridge->stats[1], bridge->stats[2],
        bridge->stats[3], bridge->stats[4], bridge->stats[5]) ;
fprintf(msgFile,
        "\n\n factorization: raw mflops %8.3f, overall mflops %8.3f",
        1.e-6*nfactorops/bridge->cpus[8],
        1.e-6*nfactorops/bridge->cpus[10]) ;
fflush(msgFile) ;
/*--------------------------------------------------------------------*/
/*
   ------------------------
   setup the parallel solve
   ------------------------
*/
```

```
rc = BridgeMT_solveSetup(bridge) ;
fprintf(msgFile, "\n\n ----- PARALLEL SOLVE SETUP -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to setup parallel solve",
        bridge->cpus[11]) ;
/*--------------------------------------------------------------------*/
/*
   ----------------
   solve the system
   ----------------
*/
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
rc = BridgeMT_solve(bridge, permuteflag, mtxX, mtxY) ;
if (rc == 1) {
   fprintf(msgFile, "\n\n solve complete successfully\n") ;
} else {
   fprintf(msgFile, "\n" " return code from solve = %d\n", rc) ;
}
fprintf(msgFile, "\n\n ----- SOLVE -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to permute rhs into new ordering"
        "\n    CPU %8.3f : time to solve linear system"
        "\n    CPU %8.3f : time to permute solution into old ordering"
        "\n CPU %8.3f : total solve time\n",
        bridge->cpus[12], bridge->cpus[13],
        bridge->cpus[14], bridge->cpus[15]) ;
fprintf(msgFile,
        "\n\n solve: raw mflops %8.3f, overall mflops %8.3f",
        1.e-6*nsolveops/bridge->cpus[13],
        1.e-6*nsolveops/bridge->cpus[15]) ;
fflush(msgFile) ;
if ( msglvl > 0 ) {
   fprintf(msgFile, "\n\n solution matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
if ( strcmp(solFileName, "none") != 0 ) {
/*
   -----------------------------------
   write the solution matrix to a file
   -----------------------------------
*/
   rc = DenseMtx_writeToFile(mtxX, solFileName) ;
   if ( rc != 1 ) {
      fprintf(msgFile,
              "\n fatal error writing mtxX to file %s, rc = %d",
              solFileName, rc) ;
      fflush(msgFile) ;
      exit(-1) ;
   }
}
/*--------------------------------------------------------------------*/
/*
   ---------------------
   free the working data
   ---------------------
*/
InpMtx_free(mtxA) ;
DenseMtx_free(mtxX) ;
DenseMtx_free(mtxY) ;
BridgeMT_free(bridge) ;

/*--------------------------------------------------------------------*/

return(1) ; }

/*--------------------------------------------------------------------*/
```

# Appendix C

# testWrapperMPI.c — A MPI Driver Program

```
/*  testWrapperMPI.c  */

#include "../BridgeMPI.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   ----------------------------------------------------------
   purpose -- main driver program to solve a linear system
      where the matrix and rhs are read in from files and
      the solution is written to a file.
      NOTE: MPI version

   created -- 98sep25, cca and pjs
   ----------------------------------------------------------
*/
BridgeMPI   *bridge ;
char        *mtxFileName, *rhsFileName, *solFileName ;
double      nfactorops ;
FILE        *msgFile ;
InpMtx      *mtxA ;
int         error, msglvl, myid, neqns, nfent, nfind, nfront,
            nproc, nrhs, nrow, nsolveops, permuteflag, rc, seed,
            symmetryflag, type ;
int         tstats[6] ;
DenseMtx    *mtxX, *mtxY ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------------------------------------
   find out the identity of this process and the number of process
   --------------------------------------------------------------
*/
```

```
MPI_Init(&argc, &argv) ;
MPI_Comm_rank(MPI_COMM_WORLD, &myid) ;
MPI_Comm_size(MPI_COMM_WORLD, &nproc) ;
/*--------------------------------------------------------------------*/
/*
    --------------------
    get input parameters
    --------------------
*/
if ( argc != 10 ) {
   fprintf(stdout,
           "\n\n usage : %s msglvl msgFile neqns type symmetryflag"
           "\n         mtxFile rhsFile solFile seed"
           "\n   msglvl  -- message level"
           "\n      0 -- no output"
           "\n      1 -- timings and statistics"
           "\n      2 and greater -- lots of output"
           "\n   msgFile -- message file"
           "\n   neqns   -- # of equations"
           "\n   type    -- type of entries"
           "\n      1 -- real"
           "\n      2 -- complex"
           "\n   symmetryflag -- symmetry flag"
           "\n      0 -- symmetric"
           "\n      1 -- hermitian"
           "\n      2 -- nonsymmetric"
           "\n   mtxFile -- input file for A matrix InpMtx object"
           "\n      must be *.inpmtxf or *.inpmtxb"
           "\n   rhsFile -- input file for Y DenseMtx object"
           "\n      must be *.densemtxf or *.densemtxb"
           "\n   solFile -- output file for X DenseMtx object"
           "\n      must be none, *.densemtxf or *.densemtxb"
           "\n   seed -- random number seed"
           "\n",
        argv[0]) ;
   return(0) ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else {
   int    length = strlen(argv[2]) + 1 + 4 ;
   char   *buffer = CVinit(length, '\0') ;
   sprintf(buffer, "%s.%d", argv[2], myid) ;
   if ( (msgFile = fopen(buffer, "w")) == NULL ) {
      fprintf(stderr, "\n fatal error in %s"
              "\n unable to open file %s\n",
              argv[0], argv[2]) ;
      MPI_Finalize() ;
      return(0) ;
   }
   CVfree(buffer) ;
}
neqns        = atoi(argv[3]) ;
type         = atoi(argv[4]) ;
symmetryflag = atoi(argv[5]) ;
mtxFileName  = argv[6] ;
rhsFileName  = argv[7] ;
solFileName  = argv[8] ;
seed         = atoi(argv[9]) ;
fprintf(msgFile,
        "\n\n %s input :"
        "\n msglvl       = %d"
        "\n msgFile      = %s"
        "\n neqns        = %d"
        "\n type         = %d"
        "\n symmetryflag = %d"
        "\n mtxFile      = %s"
        "\n rhsFile      = %s"
        "\n solFile      = %s"
        "\n",
        argv[0], msglvl, argv[2], neqns, type, symmetryflag,
        mtxFileName, rhsFileName, solFileName) ;
/*--------------------------------------------------------------------*/
/*
   ----------------------------------
   processor zero reads in the matrix.
   if an error is found,
   all processors exit cleanly
   ----------------------------------
*/
if ( myid != 0 ) {
   mtxA = NULL ;
} else {
/*
   ---------------------------------------------------
   open the file, read in the matrix and close the file
   ---------------------------------------------------
*/
   mtxA = InpMtx_new() ;
   rc = InpMtx_readFromFile(mtxA, mtxFileName) ;
   if ( rc != 1 ) {
      fprintf(msgFile,
              "\n fatal error reading mtxA from file %s, rc = %d",
              mtxFileName, rc) ;
      fflush(msgFile) ;
   }
}
/*
   -------------------------------------------------------------
   processor 0 broadcasts the error return to the other processors
   -------------------------------------------------------------
```

```
*/
MPI_Bcast((void *) &rc, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
if ( rc != 1 ) {
   MPI_Finalize() ;
   return(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   ---------------------------------------------------
   processor zero reads in the right hand side matrix.
   if an error is found, all processors exit cleanly
   ---------------------------------------------------
*/
if ( myid != 0 ) {
   mtxY = NULL ;
} else {
/*
   ----------------------------------
   read in the right hand side matrix
   ----------------------------------
*/
   mtxY = DenseMtx_new() ;
   rc = DenseMtx_readFromFile(mtxY, rhsFileName) ;
   if ( rc != 1 ) {
      fprintf(msgFile,
              "\n fatal error reading mtxY from file %s, rc = %d",
              rhsFileName, rc) ;
      fflush(msgFile) ;
   } else {
      DenseMtx_dimensions(mtxY, &nrow, &nrhs) ;
   }
}
/*
   ---------------------------------------------------------------
   processor 0 broadcasts the error return to the other processors
   ---------------------------------------------------------------
*/
MPI_Bcast((void *) &rc, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
if ( rc != 1 ) {
   MPI_Finalize() ;
   return(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   ------------------------------------------
   create and setup a BridgeMPI object
   set the MPI, matrix and message parameters
   ------------------------------------------
*/
bridge = BridgeMPI_new() ;
BridgeMPI_setMPIparams(bridge, nproc, myid, MPI_COMM_WORLD) ;
```

```
BridgeMPI_setMatrixParams(bridge, neqns, type, symmetryflag) ;
BridgeMPI_setMessageInfo(bridge, msglvl, msgFile) ;
/*
   -----------------
   setup the problem
   -----------------
*/
rc = BridgeMPI_setup(bridge, mtxA) ;
fprintf(msgFile,
        "\n\n ----- SETUP -----\n"
        "\n    CPU %8.3f : time to construct Graph"
        "\n    CPU %8.3f : time to compress Graph"
        "\n    CPU %8.3f : time to order Graph"
        "\n    CPU %8.3f : time for symbolic factorization"
        "\n    CPU %8.3f : time to broadcast front tree"
        "\n    CPU %8.3f : time to broadcast symbolic factorization"
        "\n CPU %8.3f : total setup time\n",
        bridge->cpus[0], bridge->cpus[1], bridge->cpus[2],
        bridge->cpus[3], bridge->cpus[4], bridge->cpus[5],
        bridge->cpus[6]) ;
rc = BridgeMPI_factorStats(bridge, type, symmetryflag, &nfront,
                           &nfind, &nfent, &nsolveops, &nfactorops) ;
if ( rc != 1 ) {
   fprintf(stderr,
           "\n error return %d from BridgeMPI_factorStats()", rc) ;
   MPI_Finalize() ;
   exit(-1) ;
}
fprintf(msgFile,
        "\n\n factor matrix statistics"
        "\n %d fronts, %d indices, %d entries"
        "\n %d solve operations, %12.4e factor operations",
        nfront, nfind, nfent, nsolveops, nfactorops) ;
fflush(msgFile) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------
   setup the parallel factorization
   --------------------------------
*/
rc = BridgeMPI_factorSetup(bridge, 0, 0.0) ;
if ( rc != 1 ) {
   fprintf(stderr,
           "\n error return %d from BridgeMPI_factorSetup()", rc) ;
   MPI_Finalize() ;
   exit(-1) ;
}
fprintf(msgFile, "\n\n ----- PARALLEL FACTOR SETUP -----\n") ;
fprintf(msgFile,
        "\n    CPU %8.3f : time to setup parallel factorization",
        bridge->cpus[7]) ;
```

```
if ( msglvl > 0 ) {
   fprintf(msgFile, "\n total factor operations = %.0f"
           "\n upper bound on speedup due to load balance = %.2f",
           DV_sum(bridge->cumopsDV),
           DV_sum(bridge->cumopsDV)/DV_max(bridge->cumopsDV)) ;
   fprintf(msgFile, "\n operations distributions over processors") ;
   DV_writeForHumanEye(bridge->cumopsDV, msgFile) ;
   fflush(msgFile) ;
}
/*-------------------------------------------------------------------*/
/*
   ----------------------------------------------------
   set the factorization parameters and factor the matrix
   ----------------------------------------------------
*/
permuteflag = 1 ;
rc = BridgeMPI_factor(bridge, mtxA, permuteflag, &error) ;
fprintf(msgFile, "\n\n ----- FACTORIZATION -----\n") ;
if ( rc == 1 ) {
   fprintf(msgFile, "\n\n factorization completed successfully\n") ;
} else {
   fprintf(msgFile, "\n"
           "\n return code from factorization = %d\n"
           "\n error code                     = %d\n",
           rc, error) ;
   MPI_Finalize() ;
   exit(-1) ;
}
fprintf(msgFile,
        "\n    CPU %8.3f : time to permute original matrix"
        "\n    CPU %8.3f : time to distribute original matrix"
        "\n    CPU %8.3f : time to initialize factor matrix"
        "\n    CPU %8.3f : time to compute factorization"
        "\n    CPU %8.3f : time to post-process factorization"
        "\n CPU %8.3f : total factorization time\n",
        bridge->cpus[8], bridge->cpus[9], bridge->cpus[10],
        bridge->cpus[11], bridge->cpus[12], bridge->cpus[13]) ;
IVzero(6, tstats) ;
MPI_Reduce((void *) bridge->stats, (void *) tstats, 6, MPI_INT,
           MPI_SUM, 0, bridge->comm) ;
fprintf(msgFile,
        "\n\n   factorization statistics"
        "\n   %d pivots, %d pivot tests, %d delayed vertices"
        "\n   %d entries in D, %d entries in L, %d entries in U",
        tstats[0], tstats[1], tstats[2],
        tstats[3], tstats[4], tstats[5]) ;
fprintf(msgFile,
        "\n\n   factorization: raw mflops %8.3f, overall mflops %8.3f",
        1.e-6*nfactorops/bridge->cpus[11],
        1.e-6*nfactorops/bridge->cpus[13]) ;
fflush(msgFile) ;
```

```
/*--------------------------------------------------------------------*/
/*
   ------------------------
   setup the parallel solve
   ------------------------
*/
rc = BridgeMPI_solveSetup(bridge) ;
fprintf(msgFile, "\n\n ----- PARALLEL SOLVE SETUP -----\n"
        "\n    CPU %8.3f : time to setup parallel solve",
        bridge->cpus[14]) ;
if ( rc != 1 ) {
   fprintf(stderr,
           "\n error return %d from BridgeMPI_solveSetup()", rc) ;
   MPI_Finalize() ;
   exit(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------
   processor 0 initializes a DenseMtx object
   to hold the global solution matrix
   -----------------------------------------
*/
if ( myid == 0 ) {
   mtxX = DenseMtx_new() ;
   DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
   DenseMtx_zero(mtxX) ;
} else {
   mtxX = NULL ;
}
/*
   ---------------------------------------------
   the processors solve the system cooperatively
   ---------------------------------------------
*/
permuteflag = 1 ;
rc = BridgeMPI_solve(bridge, permuteflag, mtxX, mtxY) ;
if ( rc == 1 ) {
   fprintf(msgFile, "\n\n solve complete successfully\n") ;
} else {
   fprintf(msgFile, "\n" " return code from solve = %d\n", rc) ;
}
fprintf(msgFile, "\n\n ----- SOLVE -----\n"
        "\n    CPU %8.3f : time to permute rhs into new ordering"
        "\n    CPU %8.3f : time to distribute rhs "
        "\n    CPU %8.3f : time to initialize solution matrix "
        "\n    CPU %8.3f : time to solve linear system"
        "\n    CPU %8.3f : time to gather solution "
        "\n    CPU %8.3f : time to permute solution into old ordering"
        "\n CPU %8.3f : total solve time"
        "\n\n solve: raw mflops %8.3f, overall mflops %8.3f",
        bridge->cpus[15], bridge->cpus[16], bridge->cpus[17],
        bridge->cpus[18], bridge->cpus[19], bridge->cpus[20],
        bridge->cpus[21],
        1.e-6*nsolveops/bridge->cpus[18],
        1.e-6*nsolveops/bridge->cpus[21]) ;
fflush(msgFile) ;
if ( myid == 0 ) {
   if ( msglvl > 0 ) {
      fprintf(msgFile, "\n\n solution matrix in original ordering") ;
      DenseMtx_writeForHumanEye(mtxX, msgFile) ;
      fflush(msgFile) ;
   }
}
/*--------------------------------------------------------------------*/
/*
   ---------------------
   free the working data
   ---------------------
*/
if ( myid == 0 ) {
   InpMtx_free(mtxA) ;
   DenseMtx_free(mtxX) ;
   DenseMtx_free(mtxY) ;
}
BridgeMPI_free(bridge) ;

/*--------------------------------------------------------------------*/

MPI_Finalize() ;

return(1) ; }

/*--------------------------------------------------------------------*/
```

# Index