Solving Linear Systems using **SPOOLES 2.2**

C. C. Ashcraft, R. G. Grimes, D. J. Pierce, D. K. Wah
Boeing Phantom Works*

April 29, 2002

*P. O. Box 24346, Mail Stop 7L-22, 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.

Contents

1 Overview

The **SPOOLES** software library is designed to solve sparse systems of linear equations $AX = Y$ for X, where A is full rank and X and Y are dense matrices. The matrix A can be either real or complex, symmetric, Hermitian, square nonsymmetric, or overdetermined. When A is square, there are four steps in the process of solving $AX = Y$.

- **communicate** the data for the problem as A, X and Y.
- **reorder** as $\widetilde{A}\widetilde{X} = \widetilde{Y}$, where $\widetilde{A} = P_1 A P_1^T$, $\widetilde{X} = P_1 X$ and $\widetilde{Y} = P_1 Y$, and P_1 is a permutation matrix.
- **factor** $\widetilde{A} = P_2 L D U Q_2^T$, where P_2 and Q_2 are permutation matrices.
- **solve** $LDU(Q_2^T P_1 X) = (P_2^T P_1 Y)$.

When A is symmetric or Hermitian, $L = U^T$ or $L = U^H$, respectively, and $P_2 = Q_2$. For a QR factorization of A, there are also four steps in the process of solving $AX = Y$.

- **communicate** the data for the problem as A, X and Y.
- **reorder** as $\widetilde{A}\widetilde{X} = Y$, where $\widetilde{A} = AP^T$ and $\widetilde{X} = PX$. and P is a permutation matrix.
- **factor** $\widetilde{A} = QR$, where Q is orthogonal and R is upper triangular.
- **solve** $R^T R(PX) = A^T Y$ (if real) or **solve** $R^H R(PX) = A^H Y$ (if complex).

The purpose of this manual is to describe the solution process in a step by step manner for several different scenarios — serial, multithreaded and MPI environments, and LU and QR factorizations. The following sections describe driver programs to solve linear systems of these types.

This document largely replaces the *"User Manual"* from the **SPOOLES 1.0** and **SPOOLES 2.0** releases. The material of those documents that deals with ordering sparse matrices has been removed and can be found in an improved and extended format as the *"Ordering Sparse Matrices and Transforming Front Trees"* user document.

The four simple steps described above to solve a linear system translate into a fair amount of code, as we shall see in the following sections. The user who is looking for a more gentle introduction to solving square linear systems using an LU factorization, might first take a look at `LinSol` directory and the user document *"Wrapper Objects for Solving a Linear System of Equations using* **SPOOLES** *2.2"*. The **SPOOLES** library has been integrated into the CSAR-Nastran finite element package, in the serial, multithreaded and MPI environments. To make the process easier, we wrote "wrappers" around the code found in the three driver programs. These wrapper objects insulate the user from much of the complexity of using the package. There is a cost, functionality is somewhat restricted. They are meant as learning examples rather than final code.

In a similar way, the **SPOOLES** library has been integrated into a Block-Shifted Lanczos eigensolver for symmetric eigensystems. See the `Eigen` subdirectory and the user document *"Integrating the* **SPOOLES** *2.2 Sparse Linear Algebra Library into the* **LANCZOS** *Block-shifted Lanczos Eigensolver"*. This set of wrapper objects has also served as a model to integrate **SPOOLES** into the **PLANSO** eigensystem package.

The **SPOOLES** library is based on an object oriented design philosophy. There are several data structures or objects that the user must interact with. These interactions are performed with a set of methods for each object. Every object has some standard methods, such as initializing the object, placing data into the object, extracting data out of the object, writing and reading the object to a input/output file, printing the contents of the object to a specified file, and freeing the object.

For example, consider the `DenseMtx` object that models a dense matrix. The `DenseMtx/DenseMtx.h` header file defines the object's C struct and has prototypes (with extensive comments) of the object's methods. The source files are found in the `DenseMtx/src` directory. The LaTeX documentation files are found in the `DenseMtx/doc` directory. The files can be used to create the `DenseMtx` object's chapter in the Reference Manual, or in a standalone manner to generate the object's documentation. The `DenseMtx/drivers` directory contains driver programs that exercise and validate the object's functionality.

Almost all the methods in the library are associated with a particular object. There are some exceptions, mostly found in the `misc/src` directory. The `misc/drivers` directory contains the serial LU and QR driver programs. The `MT/drivers` and `MPI/drivers` directories contain the multithreaded and MPI LU driver programs.

2 Serial Solution of $AX = Y$ using an LU factorization

The user has some representation of the data which represents the linear system, $AX = Y$. The user wants the solution X. The **SPOOLES** library will use A and Y and provide X back to the user.

The **SPOOLES** library is based on an object oriented design philosophy. The first object that the user must interact with is `InpMtx`[1]. The `InpMtx` object is where the **SPOOLES** representation of A is assembled. The user can input the representation of A into the `InpMtx` object with methods for single matrix entry (consisting of the row index, the column index, and the value), for an array of entries, for a set of entries in a specified row or column, and for a dense sub-matrices (useful for finite element applications). All of these methods can be used interchangeably with each other.

A complete listing of a sample program is found in Section A. We will now begin to work our way through the program to illustrate the use of **SPOOLES** to solve a system of linear equations.

2.1 Reading the input parameters

The program starts by declaring a variety of variables and pointers for the program. It then reads the following parameters from standard input.

- The variable `msglvl` controls the level of output generated by the program and by **SPOOLES**.
- The printed output is sent to `messageFile`.
- Whether the matrix is real or complex is controled by `type` (1 for real, 2 for complex).
- Similarly, `symmetryflag` controls whether the matrix is symmetric (0), Hermitian (1), or nonsymmetric (2).
- The matrix data will be read from the file `matrixFileName`. The matrix data has a simple format with the first line containing the number of rows (`nrow`), the number of columns (`ncol`), and the number of entries (`nent`). The remaining `nent` lines on the file contain the row number, the column number, and value for each nonzero in the sparse matrix. In our sample case, the matrix is symmetric so only the entries in the upper triangle are given on the file. If the matrix is complex, there would be 2 values, one for the real part and one for the imaginary part. **SPOOLES** follows the C language convention for indexing all arrays starting with 0. So the row and column labels for a matrix of order `neqns` range from 0 to `neqns-1`.
- The right hand side matrix Y will be read from the file `rhsFileName`. The first line of this file has two numbers: `nrow`, the number of rows of Y that are present in the file, followed by `nrhs`, the number of columns of Y. (The number of rows of Y in the file may be different from the number of rows in Y, since often right hand side matrices are sparse. This allows us the option of only reading in nonzero rows of Y.) The remaining lines of the file have the following format: the row id, followed by either `nrhs` floating point numbers if the system is real, or `2*nrhs` numbers if the system is complex.
- The `seed` parameter is a random number seed used in the ordering process.

2.2 Communicating the data for the problem

The following code segment from the full sample program opens the file `matrixFileName`, reads the first line of the file, and then initializes the `InpMtx` object. The program continues by reading each line of the input matrix data and uses either the method `InpMtx_inputRealEntry()` or `InpMtx_inputComplexEntry()`

[1] `InpMtx` stands for `Input Matrix`, for it is the object into which the user inputs the matrix entries.

to place that entry into the `InpMtx` object. Finally this code segment closes the file. finalizes the input to `InpMtx` by converting the internal storage of the matrix entries to a vector form. (This is necessary for later steps.)

```
inputFile = fopen(matrixFileName, "r") ;
fscanf(inputFile, "%d %d %d", &nrow, &ncol, &nent) ;
neqns = nrow ;
mtxA = InpMtx_new() ;
InpMtx_init(mtxA, INPMTX_BY_ROWS, type, nent, neqns) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le", &irow, &jcol, &value) ;
      InpMtx_inputRealEntry(mtxA, irow, jcol, value) ;
   }
} else {
   double   imag, real ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le %le", &irow, &jcol, &real, &imag) ;
      InpMtx_inputComplexEntry(mtxA, irow, jcol, real, imag) ;
   }
}
fclose(inputFile) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n input matrix") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
```

The `InpMtx` object is created via a call to `InpMtx_new()` and initialized via a call to `InpMtx_init()`. The arguments to `InpMtx_init()` are the pointer to the `InpMtx` object created by `InpMtx_new()` followed by four integers, `coordType`, `inputMode`, `maxnent`, and `maxnvector`.

- The second argument `coordType` = `INPMTX_BY_ROWS` represent a general purpose mode that is well suited for most users.[2] Some users may want to use other settings for `coordType` whose complete descriptions are found in the reference manual.
- The third argument `inputMode` controls whether the matrix is real or complex. One use of **SPOOLES** not illustrated here is that the `InpMtx` object can have no values. This allows **SPOOLES** to be used to generate an ordering for use by another package.
- The fourth argument `maxnent` is an estimate of the number of nonzero entries in the matrix.
- The fifth argument `maxnvector` is an estimate of the number of number of vectors that will be used, e.g., number of rows or numbers of columns.

The `maxnent` and `maxnvector` arguments only have to be estimates as they are used in the initial sizing of the object. Either can be 0. The `InpMtx` object resizes itself as required to handle the linear system.

[2]Note that **SPOOLES** has some pre-defined parameters such as `INPMTX_BY_ROWS` for some objects. These parameters are always uppercase and either begin with the name of the object which they apply to, or the library name, e.g., `SPOOLES_REAL`. They are described in the reference manual in the section for the particular object.

Every object in **SPOOLES** has print methods to output the contents of that object. This is illustrated in this code segment by printing the input matrix as contained in the `InpMtx` object, `mtxA`. To shorten this chapter we will from now on omit the part of the code that prints debug output to `msgFile` for the various code segments. The complete sample program in Section A contains all of the debug print statements.

After the matrix A has been read in from the file and placed in an `InpMtx` object, the right hand matrix Y is read in from a file and placed in a `DenseMtx` object. The following code fragment does this operation.

```
inputFile = fopen(rhsFileName, "r") ;
fscanf(inputFile, "%d %d", &nrow, &nrhs) ;
mtxB = DenseMtx_new() ;
DenseMtx_init(mtxB, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxB) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le", &value) ;
         DenseMtx_setRealEntry(mtxB, jrow, jrhs, value) ;
      }
   }
} else {
   double   imag, real ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le %le", &real, &imag) ;
         DenseMtx_setComplexEntry(mtxB, jrow, jrhs, real, imag) ;
      }
   }
}
fclose(inputFile) ;
```

The dense matrix object is created by a call to `DenseMtx_new()` and initialized via a call to `DenseMtx_init()`. There are seven arguments to `DenseMtx_init()`, not counting the initial pointer argument.

- The second argument specifies the type of the matrix, real or complex.
- The third and fourth arguments specify row and column ids of the matrix. This is useful when the dense matrix is a submatrix of a larger block matrix, but this feature is not used in the present context.
- The fifth and sixth arguments are the number of rows and columns in the matrix, here equal to `neqns` and `nrhs`.
- The seventh and eighth arguments are the row stride and column stride for the matrix entries. For our application we require a column major matrix, and so the row stride is `1` and the column stride is the number of rows, or `neqns`.

The initialization step allocates storage for the matrix entries, but it does not fill them with any values. This is done explicitly via the `DenseMtx_zero()` method, which places zeroes in all the entries. This is necessary since the right hand side matrix Y may be sparse, and so the number of rows in the file may not equal the number of equations.

The right hand side entries are then in, row by row, and placed into their locations via one of the two "set entries" methods. Note, the nonzero rows can be read from the file in any order.

2.3 Reordering the linear system

The first step is to find the permutation matrix P, and then permute $AX = Y$ into $(PAP^T)(PX) = PY$. The result of the **SPOOLES** ordering step is not just P or its permutation vector, it is a *front tree* that defines not just the permutation, but the blocking of the factor matrices, which in turn specifies the data structures and the computations that are performed during the factor and solves. To determine this ETree *front tree* object takes three step, as seen in the code fragment below.

```
adjIVL = InpMtx_fullAdjacency(mtxA) ;
nedges = IVL_tsize(adjIVL) ;
graph = Graph_new() ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL,
            NULL, NULL) ;
frontETree = orderViaMMD(graph, seed, msglvl, msgFile) ;
```

The ordering modules requires a graph of $A + A^T$. (The **SPOOLES** *LU* factorization works with matrices of symmetric structure.) The Graph object represents the graph of the matrix. Its internal representation uses adjacency lists, one for each vertex, which in turn are stored in an IVL object. The Graph and InpMtx objects are at a high level in the object hierarchy. To promote independence of the objects, the two do not know about each other, so we cannot create one from the other. Instead, the InpMtx object creates the lower level IVL object[3], which is then used in the initialization step for the Graph object. The Graph object is quite general, and can be used to describe a graph with unit or non-unit vertices and edges. We refrain from describing all the input parameters to initialize the Graph object and instead refer the reader to the reference manual.

Once a Graph object has been created, it is ordered via the multiple minimum degree method, whose return value is a front tree object. The minimum degree method is the simplest of the ordering methods provided in the **SPOOLES** library. For more information on ordering, please see the user document *"Ordering Sparse Matrices and Transforming Front Trees"*.

2.4 Non-numeric work

The next phase is to obtain the permutation matrix P, (stored implicitly in a permutation vector), and apply it to the front tree, the matrix A and the right hand side Y. This is done by the following code fragment.

```
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
oldToNew = IV_entries(oldToNewIV) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
newToOld   = IV_entries(newToOldIV) ;
ETree_permuteVertices(frontETree, oldToNewIV) ;
InpMtx_permute(mtxA, oldToNew, oldToNew) ;
if (  symmetryflag == SPOOLES_SYMMETRIC
   || symmetryflag == SPOOLES_HERMITIAN ) {
   InpMtx_mapToUpperTriangle(mtxA) ;
}
InpMtx_changeCoordType(mtxA, INPMTX_BY_CHEVRONS) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
DenseMtx_permuteRows(mtxB, oldToNewIV) ;
```

The oldToNewIV and newToOldIV variables are IV objects that represent an integer vector. The oldToNew and newToOld variables are pointers to int, which point to the base address of the int vector in an IV object.

[3] IVL stands for Integer Vector List, i.e., a list of integer vectors.

Once we have the permutation vector, we apply it to the front tree, by the `ETree_permuteVertices()` method, and then to the matrix with the `InpMtx_permute()` method. If the matrix A is symmetric or Hermitian, we expect all nonzero entries to be in the upper triangle. Permuting the matrix yields PAP^T, which may not have all of its entries in the upper triangle. If A is symmetric or Hermitian, the call to `InpMtx_mapToUpperTriangle()` ensures that all entries of PAP^T are in its upper triangle. Permuting the matrix destroys the internal vector structure, which has to be restored. But first we need to change the coordinate type of the `InpMtx` object, from rows into *chevrons*.[4] This is necessary in order to assemble entries of PAP^T during the numerical factorization. At this point the `InpMtx` object holds PAP^T in the form required by the factorization. What remains is to transform Y into PY, which is done via a call to `DenseMtx_permuteRows()`.

The final step is to compute the symbolic factorization, which is stored in an `IVL` object.

```
symbfacIVL = SymbFac_initFromInpMtx(frontETree, mtxA) ;
```

2.5 The Matrix Factorization

The numeric factorization step begins by initializing the `FrontMtx` object with the `frontETree` and `symbacIVL` objects created in early steps. The `FrontMtx` object holds the actual factorization. The code segment for the initialization is found below.

```
frontmtx = FrontMtx_new() ;
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, NO_LOCK, 0) ;
FrontMtx_init(frontmtx, frontETree, symbfacIVL, type, symmetryflag,
              FRONTMTX_DENSE_FRONTS, pivotingflag, NO_LOCK, 0, NULL,
              mtxmanager, msglvl, msgFile) ;
```

Here is a brief description of the initialization method and its input parameters.

- The fourth parameter is the matrix type, real or complex.
- The fifth parameter specifies whether the matrix is symmetric, Hermitian or nonsymmetric.
- The sixth parameter defines whether the fronts in the factor matrix are stored as dense or sparse matrices. The latter is necessary for an approximate factorization.
- The seventh parameter says whether pivoting is enabled for numerical stability.
- The eighth, nine and ten parameters are used during a multithreaded or MPI factorization. Their present values are for a serial factorization.
- The eleventh parameter, `mtxmanager` is a `SubMtxManager` object, an object used to manage instances of submatrices that form the factor matrices. The `FrontMtx` object does not concern itself with finding storage for the factor matrices, instead it asks the `SubMtxManager` object for submatrices. While this seems awkward, it allows the `FrontMtx` to operate in serial, multithreaded and MPI environments with little internal code differences, and it is the hook we have left in the library to extend its capabilities to out-of-core factors and solves.
- The twelveth and thirteenth parameters define the message level and message file for the factorization.

The numeric factorization is performed by the `FrontMtx_factorInpMtx()` method. The code segment from the sample program for the numerical factorization step is found below.

[4] The i-th chevron of A consists of the diagonal entry $A_{i,i}$, the i-th row of the upper triangle of A, and the i-th column of the lower triangle of A.

```
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, NO_LOCK, 1) ;
DVfill(10, cpus, 0.0) ;
IVfill(20, stats, 0) ;
rootchv = FrontMtx_factorInpMtx(frontmtx, mtxA, tau, droptol,
             chvmanager, &error, cpus, stats, msglvl, msgFile) ;
ChvManager_free(chvmanager) ;
```

Working storage used during the factorization is found in the form of block *chevrons*, in a `Chv` object, which hold the partial frontal matrix for a front. Much as with the `SubMtx` object, the `FrontMtx` object does not concern itself with managing working storage, instead it relies on a `ChvManager` object to manage the `Chv` objects. We now discuss the arguments to the factor method.

- The third argument is used when pivoting for numerical stability is enabled. Each entry in L and U is bounded above in magnitude by `tau`. We recommend a value of 100 for this parameter.
- The fourth argument is a drop tolerance that is not relevant for this case. When used with approximate factorizations, this argument is a lower bound on the magnitude of the entries that are stored in the front matrices.
- The sixth argument is an error flag. After a successful factorization, `error < 0` implies that the factorization finished. If `error > 0`, then the factorization failed at front `error`.
- The seventh and eighth arguments are vectors to be filled with statistics and breakdown of cpu times.

The return value of the factorization is a `Chv` object, which will be `NULL` if the factorization succeeded. We have left this as a hook for future extensions where only portions of the factor matrices are created.

The factorization is performed using a one-dimensional decomposition of the factor matrices. Keeping the factor matrices in this form severely limits the amount of parallelism for the forward and backsolves. We perform a post-processing step to convert the one-dimensional data structures to submatrices of a two-dimensional block decomposition of the factor matrices. The following code fragment performs this operation.

```
FrontMtx_postProcess(frontmtx, msglvl, msgFile) ;
```

2.6 The Forward and Backsolves

The following code fragment solves the linear system $(PAP^T)(PX) = PY$, and permutes the solution PX back into the original ordering, yielding X.

```
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
FrontMtx_solve(frontmtx, mtxX, mtxB, mtxmanager,
               cpus, msglvl, msgFile) ;
DenseMtx_permuteRows(mtxX, newToOldIV) ;
```

First we initialize a new `DenseMtx` object to hold X (and also PX). (Note, in all cases *but* a nonsymmetric matrix with pivoting enabled in an MPI environment, X may overwrite Y, and so we can use the same `DenseMtx` object for X and Y.) We then solve the linear system with a call to `FrontMtx_solve()`. Note that one of the arguments is the `mtxmanager` object, first created for the numerical factorization. The solve requires working submatrices, and so we continue the convention of having the `FrontMtx` ask the manager object for working storage. The last step is to permute the rows of the `DenseMtx` from the new ordering into the old ordering.

2.7 Sample Matrix and Right Hand Side Files

Immediately below are two sample files: `matrix.input` holds the matrix input and `rhs.input` holds the right hand side. This example is for a symmetric Laplacian operator on a 3×3 grid. Only entries in the upper triangle are stored. The right hand side is the 9×9 identity matrix. Note how the indices are zero-based as for C, instead of one-based as for Fortran.

`matrix.input`

```
9 9 21
0 0  4.0
1 1  4.0
2 2  4.0
3 3  4.0
4 4  4.0
5 5  4.0
6 6  4.0
7 7  4.0
8 8  4.0
0 1 -1.0
1 2 -1.0
3 4 -1.0
4 5 -1.0
6 7 -1.0
7 8 -1.0
0 3 -1.0
1 4 -1.0
2 5 -1.0
3 6 -1.0
4 7 -1.0
5 8 -1.0
```

`rhs.input`

```
9 9
0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
1 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
2 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
3 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
4 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
5 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
6 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
7 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
8 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

3 Multithreaded Solution of $AX = Y$ using an LU factorization

The only computations that are multithreaded are the factorization and forward and backsolves. Therefore, this section will describe only the differences between the serial driver in Section A and the multithreaded driver whose complete listing is found in Section B. This section will refer the reader to subsections in Section 2 for the parts of the code where the two drivers are identical.

The shared memory parallel version of **SPOOLES** is implemented using thread based parallelism. The multi-threaded code uses much of the serial code — the basic steps are the same and use the serial methods. The usage of **SPOOLES** for communicating the data for the problem and reordering the linear system is identical in the serial and multi-threaded versions. Only the numeric factorization and solve steps are parallelized using threads. What is different between the serial and threaded versions of the numeric computations is how the computations are scheduled.

While the storage for the factor matrices lies in one global `FrontMtx` object, all processes access the data in a disjoint way. During the factorization, front J is *owned* by one process that is responsible for factoring the front and computing its updates to all other fronts. In other words, only the process that owns front J performs computations with that data. During the solve, all $L_{J,I}$, $D_{I,I}$ and $U_{I,J}$ submatrices are stored in the front matrix object, but the computations with them are mapped to different threads, i.e., each thread *owns* a subset of the factor submatrices, and performs computations with it. We will now begin to work our way through the program found in Section B to illustrate the use of **SPOOLES** to solve a system of linear equations using multithreaded factor and solves.

3.1 Reading the input parameters

This step is identical to the serial code, as described in Section 2.1, with the exception that `nthread`, the number of threads, is also input.

3.2 Communicating the data for the problem

This step is identical to the serial code, as described in Section 2.2

3.3 Reordering the linear system

This step is identical to the serial code, as described in Section 2.3

3.4 Non-numeric work

This step is identical to the serial code, as described in Section 2.4, with one addition. We need to map factor computations to threads. The `ownersIV` vector object specifies which thread *owns* a front. The **SPOOLES** library has four ways to do this. Two are of academic interest — the wrap map and the balanced map – for these maps yield too much interaction between the threads. The subtree-subset map is suited for extremely well balanced front trees from nested dissection orderings. The domain decomposition map is more robust over a range of orderings, and this is what we recommend, as we see in the code fragment below.

```
if ( nthread > (nfront = FrontMtx_nfront(frontmtx)) ) {
   nthread = nfront ;
}
cumopsDV = DV_new() ;
DV_init(cumopsDV, nthread, NULL) ;
ownersIV = ETree_ddMap(frontETree, type, symmetryflag, cumopsDV, 1./(2.*nthread)) ;
```

The first step is to ensure that each thread has a front to own, decreasing the number of threads if necessary. We then construct the owners map using the front tree object. The `cumopsDV` object is a double precision vector object whose length is the number of threads. On return from the map call, it contains the number of factor operations that will be performed by each thread when pivoting for stability is not enabled.

3.5 The Matrix Factorization

During the factorization and solves, the threads access data and modify the state of the `FrontMtx` and `SubMtxManager` objects in a concurrent fashion, so there must be some way to control this access for critical sections of code. Inside each of the two objects we have placed a `Lock` object. The **SPOOLES** `Lock` object is little more than a wrapper around a mutual exclusion lock. It provides a simple abstract interface so that other objects which contain locks need not know about the particular thread package we use, be it Solaris threads, or POSIX threads, or another.

To notify the `FrontMtx` and `SubMtxManager` objects that they must have a lock, their initialization method calls differ slightly from the serial version. See Section 2.5 for a discussion of the similar features. The code fragment below shows their initialization calls.

```
frontmtx   = FrontMtx_new() ;
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, LOCK_IN_PROCESS, 0) ;
FrontMtx_init(frontmtx, frontETree, symbfacIVL, type, symmetryflag,
              FRONTMTX_DENSE_FRONTS, pivotingflag, LOCK_IN_PROCESS,
              0, NULL, mtxmanager, msglvl, msgFile) ;
```

The difference is that the `SubMtxManager` and `FrontMtx` objects are initialized with a `LOCK_IN_PROCESS` flag instead of a `NO_LOCK` flag. The scope of the mutual exclusion lock is for threads within the same process, not across the system.

The numeric factorization is performed by the `FrontMtx_factorInpMtx()` method. The code segment from the sample program for the numerical factorization step is found below.

```
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, LOCK_IN_PROCESS, 1) ;
DVfill(10, cpus, 0.0) ;
IVfill(20, stats, 0) ;
rootchv = FrontMtx_MT_factorInpMtx(frontmtx, mtxA, tau, droptol, chvmanager, ownersIV,
                                   lookahead, &error, cpus, stats, msglvl, msgFile) ;
ChvManager_free(chvmanager) ;
```

Note that the `ChvManager` is also locked. There are two additional parameters in the calling sequence of the multithreaded factorization.

- The `ownersIV` object maps the fronts to threads.
- The `lookahead` parameter controls the flow of execution during the factorization. Since the threads work cooperatively to compute the factor matrices, there is idle time while one thread waits on another. The `lookahead` parameter controls the ability of the thread to look past the present idle point and perform work that is not so immediate. Unfortunately, while a thread is off doing this work, it may block a thread at a more crucial point. When `lookahead = 0`, each processor tries to do only "immediate" work. Moderate speedups in the factorization have been for values of `lookahead` up to the number of threads. For nonzero `lookahead` values, the amount of working storage can increase, sometimes appreciably.

The post-processing of the factorization is exactly the same as the serial code. Note, this step can be trivially parallelized, but is not done at present.

After the post-processing step, the `FrontMtx` object contains the $L_{J,I}$, $D_{I,I}$ and $U_{I,J}$ submatrices. What remains to be done is to specify which threads own which submatrices, and thus perform computations with them. This is done by constructing a *"solve–map"* object, as we see below.

```
solvemap = SolveMap_new() ;
SolveMap_ddMap(solvemap, symmetryflag, FrontMtx_upperBlockIVL(frontmtx),
               FrontMtx_lowerBlockIVL(frontmtx), nthread, ownersIV,
               FrontMtx_frontTree(frontmtx), seed, msglvl, msgFile) ;
```

This object also uses a domain decomposition map, the only solve map that presently found in the **SPOOLES** library.

3.6 The Forward and Backsolves

The parallel solve is remarkably similar to the serial solve, as we see with the code fragment below.

```
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
FrontMtx_MT_solve(frontmtx, mtxX, mtxY, mtxmanager, solvemap, cpus, msglvl, msgFile) ;
DenseMtx_permuteRows(mtxX, newToOldIV) ;
```

The only difference between the serial and multithreaded solve methods is the presence of the solve–map object in the latter.

3.7 Sample Matrix and Right Hand Side Files

The multithreaded driver uses the same input files as found in Section 2.7.

4 MPI Solution of $AX = Y$ using an LU factorization

Unlike the serial and multithreaded environments where the data structures are global, existing under one address space, in the MPI environment, data is local, each process or processor has its own distinct address space. The MPI step-by-step process to solve a linear system is exactly the same as the multithreaded case, with the additional trouble that the data structures are distributed and need to be re-distributed as needed.

The ownership of the factor matrices during the factorization and solves is exactly the same as for the multithreaded case – the map from fronts to processors and map from submatrices to processors are identical to their counterparts in the multithreaded program. What is different is the explicit message passing of data structures between processors. Luckily, most of this is hidden to the user code.

We will now begin to work our way through the program found in Section C to illustrate the use of **SPOOLES** to solve a system of linear equations in the MPI environment.

4.1 Reading the input parameters

This step is identical to the serial code, as described in Section 2.1, with the exception that the file names for A and Y are hardcoded in the driver, and so are not part of the input parameters.

4.2 Communicating the data for the problem

This step is identical to the serial code, as described in Section 2.2 In the serial and multithreaded codes, the entire matrix A was read in from one file and placed into one `InpMtx` object. In the MPI environment, this need not be the case that one processor holds the entire matrix A. (In fact, A must be distributed across processors during the factorization.)

Each processor opens a matrix file, (possibly) reads in matrix entries, and creates its *local* `InpMtx` object that holds the matrix entries it has read in. We have hardcoded the file names: processor q reads its matrix entries from file `matrix.`q`.input` and its right hand side entries from file `rhs.`q`.input`. The file formats are the same as for the serial and multithreaded drivers.

The entries needed not be partitioned over the files. For example, each processor could read in entries for disjoint sets of finite elements. Naturally some degrees of freedom will have support on elements that are found on different processors. When the entries in A and Y are mapped to processors, an assembly of the matrix entries will be done automatically.

It could be the case that the matrix A and right hand side Y are read in by one processor. (This was the approach we took with the `LinSol` wrapper objects.) There still need to be input files for the other processors with zeroes on their first (and only) line, to specify that no entries are to be read.

4.3 Reordering the linear system

The first part is very similar to the serial code, as described in Section 2.3.

```
graph = Graph_new() ;
adjIVL = InpMtx_MPI_fullAdjacency(mtxA, stats, msglvl, msgFile, MPI_COMM_WORLD) ;
nedges = IVL_tsize(adjIVL) ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL, NULL, NULL) ;
frontETree = orderViaMMD(graph, seed + myid, msglvl, msgFile) ;
```

While the data and computations are distributed across the processors, the ordering process is not. Therefore we need a global graph on each processor. Since the matrix A is distributed across the processors, we use the distributed `InpMtx_MPI_fullAdjacency()` method to construct the `IVL` object of the graph of $A + A^T$.

At this point, each processor has computed its own minimum degree ordering and created a front tree object. The orderings will likely be different, because each processors input a different random number seed to the ordering method. Only one ordering can be used for the factorization, so the processors collectively determine which of the orderings is best, which is then broadcast to all the processors, as the code fragment below illustrates.

```
opcounts = DVinit(nproc, 0.0) ;
opcounts[myid] = ETree_nFactorOps(frontETree, type, symmetryflag) ;
MPI_Allgather((void *) &opcounts[myid], 1, MPI_DOUBLE,
              (void *) opcounts, 1, MPI_DOUBLE, MPI_COMM_WORLD) ;
minops = DVmin(nproc, opcounts, &root) ;
DVfree(opcounts) ;
frontETree = ETree_MPI_Bcast(frontETree, root, msglvl, msgFile, MPI_COMM_WORLD) ;
```

4.4 Non-numeric work

Once the front tree is replicated across the processors, we obtain the permutation vectors and permute the vertices in the front tree. The local matrices for A and Y are also permuted. These steps are identical to the serial and multithreaded drivers, except the fact local instead of global A and Y matrices are permuted.

```
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
ETree_permuteVertices(frontETree, oldToNewIV) ;
InpMtx_permute(mtxA, IV_entries(oldToNewIV),
IV_entries(oldToNewIV)) ;
if (  symmetryflag == SPOOLES_SYMMETRIC || symmetryflag == SPOOLES_HERMITIAN ) {
   InpMtx_mapToUpperTriangle(mtxA) ;
}
InpMtx_changeCoordType(mtxA, INPMTX_BY_CHEVRONS) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
DenseMtx_permuteRows(mtxY, oldToNewIV) ;
```

The next step is to obtain the map from fronts to processors, just as was done in the multithreaded driver. In addition, we need a map from vertices to processors to be able to distribute the matrix A and right hand side Y as necessary. Since we have the map from vertices to fronts inside the front tree object, the vertex map is easy to determine.

```
cutoff   = 1./(2*nproc) ;
cumopsDV = DV_new() ;
DV_init(cumopsDV, nproc, NULL) ;
ownersIV = ETree_ddMap(frontETree, type, symmetryflag, cumopsDV, cutoff) ;
DV_free(cumopsDV) ;
vtxmapIV = IV_new() ;
IV_init(vtxmapIV, neqns, NULL) ;
IVgather(neqns, IV_entries(vtxmapIV), IV_entries(ownersIV), ETree_vtxToFront(frontETree)) ;
```

At this point we are ready to assemble and distribute the entries of A and Y.

```
firsttag = 0 ;
newA = InpMtx_MPI_split(mtxA, vtxmapIV, stats, msglvl, msgFile, firsttag,
MPI_COMM_WORLD) ;
InpMtx_free(mtxA) ;
```

```
mtxA = newA ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
newY = DenseMtx_MPI_splitByRows(mtxY, vtxmapIV, stats, msglvl,
                                msgFile, firsttag, MPI_COMM_WORLD) ;
DenseMtx_free(mtxY) ;
mtxY = newY ;
```

The `InpMtx_MPI_split()` method assembles and redistributes the matrix entries by the vectors of the local matrix. Recall above that the coordinate type was set to chevrons, as is needed for the assembly of the entries into the front matrices. The method returns a new `InpMtx` object that contains the part of A that is needed by the processor. The old `InpMtx` object is free'd and the new one takes its place.

Now we are ready to compute the symbolic factorization, but it too much be done in a distributed manner.

```
symbfacIVL = SymbFac_MPI_initFromInpMtx(frontETree, ownersIV, mtxA,
                    stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
```

The `symbfacIVL` object on a particular processor is only a subset of the global symbolic factorization, containing only what it needs to know for it to compute its part of the factorization.

4.5 The Matrix Factorization

In contrast the the multithreaded environment, data structures are local to a processor, and so locks are not needed to manage access to critical regions of code. The initialization of the front matrix and submatrix manager objects is much like the serial case, with one exception.

```
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, NO_LOCK, 0) ;
frontmtx = FrontMtx_new() ;
FrontMtx_init(frontmtx, frontETree, symbfacIVL, type, symmetryflag,
              FRONTMTX_DENSE_FRONTS, pivotingflag, NO_LOCK, myid,
              ownersIV, mtxmanager, msglvl, msgFile) ;
```

Note that the nineth and tenth arguments are `myid` and `ownersIV`, not `0` and `NULL` as for the serial and multithreaded drivers. These arguments tell the front matrix object that it needs to initialize only those parts of the factor matrices that it "owns", which are given by the map from fronts to processors and the processor id.

The numeric factorization is performed by the `FrontMtx_MPI_factorInpMtx()` method. The code segment from the sample program for the numerical factorization step is found below.

```
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, NO_LOCK, 0) ;
rootchv = FrontMtx_MPI_factorInpMtx(frontmtx, mtxA, tau, droptol,
                     chvmanager, ownersIV, lookahead, &error, cpus,
                     stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
ChvManager_free(chvmanager) ;
```

Note that the `ChvManager` is not locked. The calling sequence is identical to that of the multithreaded factorization except for the addition of the `firsttag` and MPI communicator at the end.

The post-processing of the factorization is the same in principle as in the serial code but differs in that is uses the distributed data structures.

```
FrontMtx_MPI_postProcess(frontmtx, ownersIV, stats, msglvl,
                         msgFile, firsttag, MPI_COMM_WORLD) ;
```

After the post-processing step, each local `FrontMtx` object contains the $L_{J,I}$, $D_{I,I}$ and $U_{I,J}$ submatrices for the fronts that were owned by the particular processor. However, the parallel solve is based on the submatrices being distributed across the processors, not just the fronts.

We must specify which threads own which submatrices, and so perform computations with them. This is done by constructing a *"solve–map"* object, as we see below.

```
solvemap = SolveMap_new() ;
SolveMap_ddMap(solvemap, symmetryflag, FrontMtx_upperBlockIVL(frontmtx),
               FrontMtx_lowerBlockIVL(frontmtx), nproc, ownersIV,
               FrontMtx_frontTree(frontmtx), seed, msglvl, msgFile) ;
```

This object also uses a domain decomposition map, the only solve map that presently found in the **SPOOLES** library.

Once the solve map has been created, (and note that it is identical across all the processors), we redistribute the submatrices with the following code fragment.

```
FrontMtx_MPI_split(frontmtx, solvemap, stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
```

At this point in time, the submatrices that a processor owns are local to that processor.

4.6 The Forward and Backsolves

If pivoting has been performed for numerical stability, then the rows of PY may not be located on the processor that needs them. We must perform an additional redistribution of the local `DenseMtx` objects that hold PY, as the code fragment below illustrates.

```
if ( FRONTMTX_IS_PIVOTING(frontmtx) ) {
   IV   *rowmapIV ;
/*
   ----------------------------------------------------------
   pivoting has taken place, redistribute the right hand side
   to match the final rows and columns in the fronts
   ----------------------------------------------------------
*/
   rowmapIV = FrontMtx_MPI_rowmapIV(frontmtx, ownersIV, msglvl,
                                    msgFile, MPI_COMM_WORLD) ;
   newY = DenseMtx_MPI_splitByRows(mtxY, rowmapIV, stats, msglvl,
                                   msgFile, firsttag, MPI_COMM_WORLD) ;
   DenseMtx_free(mtxY) ;
   mtxY = newY ;
   IV_free(rowmapIV) ;
}
```

Each processor now must create a local `DenseMtx` object to hold the rows of PX that it owns.

```
ownedColumnsIV = FrontMtx_ownedColumnsIV(frontmtx, myid, ownersIV,
                                         msglvl, msgFile) ;
nmycol = IV_size(ownedColumnsIV) ;
```

```
mtxX = DenseMtx_new() ;
if ( nmycol > 0 ) {
   DenseMtx_init(mtxX, type, 0, 0, nmycol, nrhs, 1, nmycol) ;
   DenseMtx_rowIndices(mtxX, &nrow, &rowind) ;
   IVcopy(nmycol, rowind, IV_entries(ownedColumnsIV)) ;
}
```

If A is symmetric, or if pivoting for stability was not used, then `mtxX` can just be a pointer to `mtxY`, i.e., PX could overwrite PY.

The parallel solve is remarkably similar to the serial solve, as we see with the code fragment below.

```
solvemanager = SubMtxManager_new() ;
SubMtxManager_init(solvemanager, NO_LOCK, 0) ;
FrontMtx_MPI_solve(frontmtx, mtxX, mtxY, solvemanager, solvemap, cpus,
                   stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
SubMtxManager_free(solvemanager) ;
```

The only difference between the multithreaded and MPI solve methods is the presence of the first tag and MPI communicator in the latter.

The last step is to permute the rows of the local solution matrix into the original matrix ordering. We also gather all the solution entries into one `DenseMtx` object on processor zero.

```
DenseMtx_permuteRows(mtxX, newToOldIV) ;
IV_fill(vtxmapIV, 0) ;
firsttag++ ;
mtxX = DenseMtx_MPI_splitByRows(mtxX, vtxmapIV, stats, msglvl, msgFile,
                                firsttag, MPI_COMM_WORLD) ;
```

4.7 Sample Matrix and Right Hand Side Files

matrix.0.input	matrix.1.input	matrix.2.input	matrix.3.input
9 9 6 0 0 4.0 0 1 -1.0 0 3 -1.0 1 1 4.0 1 2 -1.0 1 4 -1.0	9 9 5 2 2 4.0 2 5 -1.0 3 3 4.0 3 4 -1.0 3 6 -1.0	9 9 7 4 4 4.0 4 5 -1.0 4 7 -1.0 5 5 4.0 5 8 -1.0 6 6 4.0 6 7 -1.0	9 9 3 7 7 4.0 7 8 -1.0 8 8 4.0
rhs.0.input	rhs.1.input	rhs.2.input	rhs.3.input
2 1 0 0.0 1 0.0	2 1 2 0.0 3 0.0	2 1 4 1.0 5 0.0	3 1 6 0.0 7 0.0 8 0.0

5 Serial Solution of $AX = Y$ using an QR factorization

Let us review the steps is solving $AX = Y$ using a QR factorization.

- **communicate** the data for the problem as A, X and Y.
- **reorder** as $\widetilde{A}\widetilde{X} = Y$, where $\widetilde{A} = AP^T$ and $\widetilde{X} = PX$. and P is a permutation matrix.
- **factor** $\widetilde{A} = QR$, where Q is orthogonal and R is upper triangular.
- **solve** $R^T R(PX) = A^T Y$ (if real) or **solve** $R^H R(PX) = A^H Y$ (if complex).

A complete listing of a sample program is found in Section D. We will now begin to work our way through the program to illustrate the use of **SPOOLES** to solve a system of linear equations.

5.1 Reading the input parameters

The input parameters are identical to those of the serial LU driver program described in Section 2.1 with the exception that the `symmetryflag` is not present.

5.2 Communicating the data for the problem

This step is identical to the serial code, as described in Section 2.2

5.3 Reordering the linear system

For the LU factorization of A, we used the graph of $A + A^T$. For the QR factorization of A, we need the graph of $A^T A$. The only difference between the two orderings is how we create the `IVL` object for the graph. For the QR factorization, we use `InpMtx_adjForATA()`, as we see below.

```
adjIVL = InpMtx_adjForATA(mtxA) ;
nedges = IVL_tsize(adjIVL) ;
graph = Graph_new() ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL,
            NULL, NULL) ;
frontETree = orderViaMMD(graph, seed, msglvl, msgFile) ;
```

The minimum degree method is the simplest of the ordering methods provided in the **SPOOLES** library. For more information on ordering, please see the user document *"Ordering Sparse Matrices and Transforming Front Trees"*.

5.4 Non-numeric work

The next phase is to obtain the permutation matrix P, (stored implicitly in a permutation vector), and apply it to the matrix A. This is done by the following code fragment.

```
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
oldToNew   = IV_entries(oldToNewIV) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
newToOld   = IV_entries(newToOldIV) ;
InpMtx_permute(mtxA, NULL, oldToNew)) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
```

The oldToNewIV and newToOldIV variables are IV objects that represent an integer vector. The oldToNew and newToOld variables are pointers to int, which point to the base address of the int vector in an IV object. Once we have the permutation vector, we apply it to the front tree, by the ETree_permuteVertices() method. We need AP^T, so we permute the InpMtx object using a NULL pointer for the row permutation (which means do not permute the rows) and the oldToNew vector for the column permutation. At this point the InpMtx object holds AP^T in the form required by the factorization.

The final steps are to compute the symbolic factorization, which is stored in an IVL object, and to permute the vertices in the front tree. The symbolic factorization differs slightly from the LU case.

```
symbfacIVL = SymbFac_initFromGraph(frontETree, graph) ;
IVL_overwrite(symbfacIVL, oldToNewIV) ;
IVL_sortUp(symbfacIVL) ;
ETree_permuteVertices(frontETree, oldToNewIV) ;
```

We do not have the A^TA matrix object, so we constuct the symbolic factorization using the front tree and the Graph object. Note, at this point in time, both the graph and front tree are in terms of the original ordering, so after the IVL object is created, its vertices must be mapped into the new permutation and sorted into ascending order. Then the vertices in the front tree are mapped into the new ordering.

5.5 The Matrix Factorization

The numeric factorization step begins by initializing the FrontMtx object with the frontETree and symbacIVL objects created in early steps. The FrontMtx object holds the actual factorization. The code segment for the initialization is found below.

```
frontmtx = FrontMtx_new() ;
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, NO_LOCK, 0) ;
if ( type == SPOOLES_REAL ) {
   FrontMtx_init(frontmtx, frontETree, symbfacIVL, type,
                 SPOOLES_SYMMETRIC, FRONTMTX_DENSE_FRONTS,
                 SPOOLES_NO_PIVOTING, NO_LOCK, 0, NULL,
                 mtxmanager, msglvl, msgFile) ;
} else {
   FrontMtx_init(frontmtx, frontETree, symbfacIVL, type,
                 SPOOLES_HERMITIAN, FRONTMTX_DENSE_FRONTS,
                 SPOOLES_NO_PIVOTING, NO_LOCK, 0, NULL,
                 mtxmanager, msglvl, msgFile) ;
}
```

This differs little from the initialization in Section 2.5, except that the matrix type is symmetric or Hermitian, and no pivoting is used for stability.

The numeric factorization is performed by the FrontMtx_QR_factor() method. The code segment from the sample program for the numerical factorization step is found below.

```
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, NO_LOCK, 1) ;
DVzero(10, cpus) ;
facops = 0.0 ;
FrontMtx_QR_factor(frontmtx, mtxA, chvmanager, cpus, &facops, msglvl, msgFile) ;
ChvManager_free(chvmanager) ;
```

Working storage used during the factorization is found in the form of block *chevrons*, in a `Chv` object, which hold the partial frontal matrix for a front. Much as with the `SubMtx` object, the `FrontMtx` object does not concern itself with managing working storage, instead it relies on a `ChvManager` object to manage the `Chv` objects. On return `facops` contains the number of floating point operations performed during the factorization.

The factorization is performed using a one-dimensional decomposition of the factor matrices. Keeping the factor matrices in this form severely limits the amount of parallelism for the forward and backsolves. We perform a post-processing step to convert the one-dimensional data structures to submatrices of a two-dimensional block decomposition of the factor matrices. The following code fragment performs this operation.

```
FrontMtx_postProcess(frontmtx, msglvl, msgFile) ;
```

5.6 Solving the linear system

The following code fragment solves the linear system $R^T R\widehat{X} = \widehat{A}^T Y$ if real or $R^H R\widehat{X} = \widehat{A}^H Y$ if complex.

```
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
FrontMtx_QR_solve(frontmtx, mtxA, mtxX, mtxB, mtxmanager,
                  cpus, msglvl, msgFile) ;
```

Last, we permute the rows of `widehatX` back into X.

```
DenseMtx_permuteRows(mtxX, newToOldIV) ;
```

5.7 Sample Matrix and Right Hand Side Files

Immediately below are two sample files: `qr.matrix.input` holds the matrix input and `qr.rhs.input` holds the right hand side. This simple example is an 8×6 matrix A and a single right hand side. The solution is the vector of all ones. Note how the indices are zero-based as for C, instead of one-based as for Fortran.

matrix.input

```
8 6 18
0 1 1.0
0 3 2.0
1 2 3.0
1 3 1.0
1 5 1.0
2 0 1.0
2 2 2.0
3 0 3.0
3 2 4.0
3 4 2.0
4 3 1.0
5 1 2.0
5 4 3.0
5 5 1.0
6 0 2.0
6 3 3.0
7 1 1.0
7 4 3.0
```

rhs.input

```
8 1
0 3.0
1 5.0
2 3.0
3 9.0
4 1.0
5 6.0
6 5.0
7 4.0
```

A `allInOne.c` – A Serial *LU* Driver Program

```
/*  allInOne.c  */

#include "../../misc.h"
#include "../../FrontMtx.h"
#include "../../SymbFac.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   --------------------------------------------------------------------
   all-in-one program to solve A X = Y

   (1) read in matrix entries for A and form InpMtx object
   (2) read in right hand side for Y entries and form DenseMtx object
   (3) form Graph object, order matrix and form front tree
   (4) get the permutation, permute the front tree, matrix
       and right hand side and get the symbolic factorization
   (5) initialize the front matrix object to hold the factor matrices
   (6) compute the numeric factorization
   (7) post-process the factor matrices
   (8) compute the solution
   (9) permute the solution into the original ordering

   created -- 98jun04, cca
   --------------------------------------------------------------------
*/
/*--------------------------------------------------------------------*/
char            *matrixFileName, *rhsFileName ;
DenseMtx        *mtxY, *mtxX ;
Chv             *rootchv ;
ChvManager      *chvmanager  ;
SubMtxManager   *mtxmanager  ;
FrontMtx        *frontmtx ;
InpMtx          *mtxA ;
double          droptol = 0.0, tau = 100. ;
double          cpus[10] ;
ETree           *frontETree ;
FILE            *inputFile, *msgFile ;
Graph           *graph ;
int             error, ient, irow, jcol, jrhs, jrow, msglvl, ncol,
                nedges, nent, neqns, nrhs, nrow, pivotingflag, seed,
                symmetryflag, type ;
int             *newToOld, *oldToNew ;
int             stats[20] ;
IV              *newToOldIV, *oldToNewIV ;
IVL             *adjIVL, *symbfacIVL ;
/*--------------------------------------------------------------------*/
/*
```

```
   --------------------
   get input parameters
   --------------------
*/
if ( argc != 9 ) {
   fprintf(stdout, "\n"
      "\n usage: %s msglvl msgFile type symmetryflag pivotingflag"
      "\n        matrixFileName rhsFileName seed"
      "\n    msglvl -- message level"
      "\n    msgFile -- message file"
      "\n    type    -- type of entries"
      "\n      1 (SPOOLES_REAL)    -- real entries"
      "\n      2 (SPOOLES_COMPLEX) -- complex entries"
      "\n    symmetryflag -- type of matrix"
      "\n      0 (SPOOLES_SYMMETRIC)    -- symmetric entries"
      "\n      1 (SPOOLES_HERMITIAN)    -- Hermitian entries"
      "\n      2 (SPOOLES_NONSYMMETRIC) -- nonsymmetric entries"
      "\n    pivotingflag -- type of pivoting"
      "\n      0 (SPOOLES_NO_PIVOTING) -- no pivoting used"
      "\n      1 (SPOOLES_PIVOTING)    -- pivoting used"
      "\n    matrixFileName -- matrix file name, format"
      "\n       nrow ncol nent"
      "\n       irow jcol entry"
      "\n        ..."
      "\n        note: indices are zero based"
      "\n    rhsFileName -- right hand side file name, format"
      "\n       nrow nrhs "
      "\n       ..."
      "\n       jrow entry(jrow,0) ... entry(jrow,nrhs-1)"
      "\n       ..."
      "\n    seed -- random number seed, used for ordering"
      "\n", argv[0]) ;
   return(0) ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else if ( (msgFile = fopen(argv[2], "a")) == NULL ) {
   fprintf(stderr, "\n fatal error in %s"
           "\n unable to open file %s\n",
           argv[0], argv[2]) ;
   return(-1) ;
}
type           = atoi(argv[3]) ;
symmetryflag   = atoi(argv[4]) ;
pivotingflag   = atoi(argv[5]) ;
matrixFileName = argv[6] ;
rhsFileName    = argv[7] ;
seed           = atoi(argv[8]) ;
/*--------------------------------------------------------------------*/
/*
```

```
   --------------------------------------------
   STEP 1: read the entries from the input file
           and create the InpMtx object
   --------------------------------------------
*/
inputFile = fopen(matrixFileName, "r") ;
fscanf(inputFile, "%d %d %d", &nrow, &ncol, &nent) ;
neqns = nrow ;
mtxA = InpMtx_new() ;
InpMtx_init(mtxA, INPMTX_BY_ROWS, type, nent, neqns) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le", &irow, &jcol, &value) ;
      InpMtx_inputRealEntry(mtxA, irow, jcol, value) ;
   }
} else {
   double   imag, real ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le %le", &irow, &jcol, &real, &imag) ;
      InpMtx_inputComplexEntry(mtxA, irow, jcol, real, imag) ;
   }
}
fclose(inputFile) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n input matrix") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------
   STEP 2: read the right hand side matrix Y
   -----------------------------------------
*/
inputFile = fopen(rhsFileName, "r") ;
fscanf(inputFile, "%d %d", &nrow, &nrhs) ;
mtxY = DenseMtx_new() ;
DenseMtx_init(mtxY, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxY) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le", &value) ;
         DenseMtx_setRealEntry(mtxY, jrow, jrhs, value) ;
      }
   }
} else {
```

```
   double   imag, real ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le %le", &real, &imag) ;
         DenseMtx_setComplexEntry(mtxY, jrow, jrhs, real, imag) ;
      }
   }
}
fclose(inputFile) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n rhs matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------------
   STEP 3 : find a low-fill ordering
   (1) create the Graph object
   (2) order the graph using multiple minimum degree
   -------------------------------------------------
*/
graph = Graph_new() ;
adjIVL = InpMtx_fullAdjacency(mtxA) ;
nedges = IVL_tsize(adjIVL) ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL,
            NULL, NULL) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n graph of the input matrix") ;
   Graph_writeForHumanEye(graph, msgFile) ;
   fflush(msgFile) ;
}
frontETree = orderViaMMD(graph, seed, msglvl, msgFile) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n front tree from ordering") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------------
   STEP 4: get the permutation, permute the front tree,
           permute the matrix and right hand side, and
           get the symbolic factorization
   ----------------------------------------------------
*/
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
oldToNew   = IV_entries(oldToNewIV) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
newToOld   = IV_entries(newToOldIV) ;
```

```
ETree_permuteVertices(frontETree, oldToNewIV) ;
InpMtx_permute(mtxA, oldToNew, oldToNew) ;
if (  symmetryflag == SPOOLES_SYMMETRIC
   || symmetryflag == SPOOLES_HERMITIAN ) {
   InpMtx_mapToUpperTriangle(mtxA) ;
}
InpMtx_changeCoordType(mtxA, INPMTX_BY_CHEVRONS) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
DenseMtx_permuteRows(mtxY, oldToNewIV) ;
symbfacIVL = SymbFac_initFromInpMtx(frontETree, mtxA) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n old-to-new permutation vector") ;
   IV_writeForHumanEye(oldToNewIV, msgFile) ;
   fprintf(msgFile, "\n\n new-to-old permutation vector") ;
   IV_writeForHumanEye(newToOldIV, msgFile) ;
   fprintf(msgFile, "\n\n front tree after permutation") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fprintf(msgFile, "\n\n input matrix after permutation") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fprintf(msgFile, "\n\n right hand side matrix after permutation") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fprintf(msgFile, "\n\n symbolic factorization") ;
   IVL_writeForHumanEye(symbfacIVL, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ------------------------------------------
   STEP 5: initialize the front matrix object
   ------------------------------------------
*/
frontmtx   = FrontMtx_new() ;
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, NO_LOCK, 0) ;
FrontMtx_init(frontmtx, frontETree, symbfacIVL, type, symmetryflag,
              FRONTMTX_DENSE_FRONTS, pivotingflag, NO_LOCK, 0, NULL,
              mtxmanager, msglvl, msgFile) ;
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------
   STEP 6: compute the numeric factorization
   -----------------------------------------
*/
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, NO_LOCK, 1) ;
DVfill(10, cpus, 0.0) ;
IVfill(20, stats, 0) ;
rootchv = FrontMtx_factorInpMtx(frontmtx, mtxA, tau, droptol,
             chvmanager, &error, cpus, stats, msglvl, msgFile) ;
ChvManager_free(chvmanager) ;
if ( msglvl > 2 ) {
```

```
   fprintf(msgFile, "\n\n factor matrix") ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
if ( rootchv != NULL ) {
   fprintf(msgFile, "\n\n matrix found to be singular\n") ;
   exit(-1) ;
}
if ( error >= 0 ) {
   fprintf(msgFile, "\n\n error encountered at front %d", error) ;
   exit(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   --------------------------------------
   STEP 7: post-process the factorization
   --------------------------------------
*/
FrontMtx_postProcess(frontmtx, msglvl, msgFile) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n factor matrix after post-processing") ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------
   STEP 8: solve the linear system
   -------------------------------
*/
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
FrontMtx_solve(frontmtx, mtxX, mtxY, mtxmanager,
               cpus, msglvl, msgFile) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n solution matrix in new ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------------------
   STEP 9: permute the solution into the original ordering
   -----------------------------------------------------
*/
DenseMtx_permuteRows(mtxX, newToOldIV) ;
if ( msglvl > 0 ) {
   fprintf(msgFile, "\n\n solution matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
```

```
}
/*--------------------------------------------------------------------*/
/*
   -----------
   free memory
   -----------
*/
FrontMtx_free(frontmtx) ;
DenseMtx_free(mtxX) ;
DenseMtx_free(mtxY) ;
IV_free(newToOldIV) ;
IV_free(oldToNewIV) ;
InpMtx_free(mtxA) ;
ETree_free(frontETree) ;
IVL_free(symbfacIVL) ;
SubMtxManager_free(mtxmanager) ;
Graph_free(graph) ;
/*--------------------------------------------------------------------*/
return(1) ; }
/*--------------------------------------------------------------------*/
```

B allInOne.c – A Serial *LU* Driver Program

```
/*  allInOneMT.c  */

#include "../spoolesMT.h"
#include "../../misc.h"
#include "../../FrontMtx.h"
#include "../../SymbFac.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   --------------------------------------------------------------------
   all-in-one program to solve A X = Y
   using a multithreaded factorization and solve

   (1) read in matrix entries for A and form InpMtx object
   (2) read in right hand side for Y entries and form DenseMtx object
   (3) form Graph object, order matrix and form front tree
   (4) get the permutation, permute the front tree, matrix
       and right hand side and get the symbolic factorization
   (5) initialize the front matrix object to hold the factor matrices
   (6) get the domain-decomposition map from fronts to threads
   (7) compute the numeric factorization
   (8) post-process the factor matrices
   (9) get the map for the parallel solve
   (10) compute the solution
   (11) permute the solution into the original ordering

   created -- 98jun04, cca
   --------------------------------------------------------------------
*/
/*--------------------------------------------------------------------*/
char            *matrixFileName, *rhsFileName ;
DenseMtx        *mtxY, *mtxX ;
Chv             *rootchv ;
ChvManager      *chvmanager ;
double          droptol = 0.0, tau = 100. ;
double          cpus[10] ;
DV              *cumopsDV ;
ETree           *frontETree ;
FrontMtx        *frontmtx ;
FILE            *inputFile, *msgFile ;
Graph           *graph ;
InpMtx          *mtxA ;
int             error, ient, irow, jcol, jrhs, jrow, lookahead, msglvl,
                ncol, nedges, nent, neqns, nfront, nrhs, nrow,
                nthread, pivotingflag, seed, symmetryflag, type ;
int             *newToOld, *oldToNew ;
int             stats[20] ;
```

```
IV              *newToOldIV, *oldToNewIV, *ownersIV ;
IVL             *adjIVL, *symbfacIVL ;
SolveMap        *solvemap ;
SubMtxManager   *mtxmanager  ;
/*--------------------------------------------------------------------*/
/*
   --------------------
   get input parameters
   --------------------
*/
if ( argc != 10 ) {
   fprintf(stdout, "\n"
      "\n usage: %s msglvl msgFile type symmetryflag pivotingflag"
      "\n        matrixFileName rhsFileName seed"
      "\n    msglvl -- message level"
      "\n    msgFile -- message file"
      "\n    type    -- type of entries"
      "\n      1 (SPOOLES_REAL)    -- real entries"
      "\n      2 (SPOOLES_COMPLEX) -- complex entries"
      "\n    symmetryflag -- type of matrix"
      "\n      0 (SPOOLES_SYMMETRIC)    -- symmetric entries"
      "\n      1 (SPOOLES_HERMITIAN)    -- Hermitian entries"
      "\n      2 (SPOOLES_NONSYMMETRIC) -- nonsymmetric entries"
      "\n    pivotingflag -- type of pivoting"
      "\n      0 (SPOOLES_NO_PIVOTING) -- no pivoting used"
      "\n      1 (SPOOLES_PIVOTING)    -- pivoting used"
      "\n    matrixFileName -- matrix file name, format"
      "\n       nrow ncol nent"
      "\n       irow jcol entry"
      "\n        ..."
      "\n        note: indices are zero based"
      "\n    rhsFileName -- right hand side file name, format"
      "\n       nrow nrhs "
      "\n       ..."
      "\n       jrow entry(jrow,0) ... entry(jrow,nrhs-1)"
      "\n       ..."
      "\n    seed    -- random number seed, used for ordering"
      "\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], "a")) == NULL ) {
   fprintf(stderr, "\n fatal error in %s"
           "\n unable to open file %s\n",
           argv[0], argv[2]) ;
   return(-1) ;
}
type           = atoi(argv[3]) ;
```

```
symmetryflag   = atoi(argv[4]) ;
pivotingflag   = atoi(argv[5]) ;
matrixFileName = argv[6] ;
rhsFileName    = argv[7] ;
seed           = atoi(argv[8]) ;
nthread        = atoi(argv[9]) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------------------
   STEP 1: read the entries from the input file
           and create the InpMtx object
   --------------------------------------------
*/
inputFile = fopen(matrixFileName, "r") ;
fscanf(inputFile, "%d %d %d", &nrow, &ncol, &nent) ;
neqns = nrow ;
mtxA = InpMtx_new() ;
InpMtx_init(mtxA, INPMTX_BY_ROWS, type, nent, 0) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le", &irow, &jcol, &value) ;
      InpMtx_inputRealEntry(mtxA, irow, jcol, value) ;
   }
} else {
   double   imag, real ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le %le", &irow, &jcol, &real, &imag) ;
      InpMtx_inputComplexEntry(mtxA, irow, jcol, real, imag) ;
   }
}
fclose(inputFile) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n input matrix") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------
   STEP 2: read the right hand side matrix Y
   -----------------------------------------
*/
inputFile = fopen(rhsFileName, "r") ;
fscanf(inputFile, "%d %d", &nrow, &nrhs) ;
mtxY = DenseMtx_new() ;
DenseMtx_init(mtxY, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxY) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
```

```
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le", &value) ;
         DenseMtx_setRealEntry(mtxY, jrow, jrhs, value) ;
      }
   }
} else {
   double   imag, real ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le %le", &real, &imag) ;
         DenseMtx_setComplexEntry(mtxY, jrow, jrhs, real, imag) ;
      }
   }
}
fclose(inputFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n rhs matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------------
   STEP 3 : find a low-fill ordering
   (1) create the Graph object
   (2) order the graph using multiple minimum degree
   -------------------------------------------------
*/
graph = Graph_new() ;
adjIVL = InpMtx_fullAdjacency(mtxA) ;
nedges = IVL_tsize(adjIVL) ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL,
            NULL, NULL) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n graph of the input matrix") ;
   Graph_writeForHumanEye(graph, msgFile) ;
   fflush(msgFile) ;
}
frontETree = orderViaMMD(graph, seed, msglvl, msgFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n front tree from ordering") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ---------------------------------------------------
   STEP 4: get the permutation, permute the front tree,
```

```
           permute the matrix and right hand side, and
           get the symbolic factorization
   ---------------------------------------------------
*/
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
oldToNew   = IV_entries(oldToNewIV) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
newToOld   = IV_entries(newToOldIV) ;
ETree_permuteVertices(frontETree, oldToNewIV) ;
InpMtx_permute(mtxA, oldToNew, oldToNew) ;
if (  symmetryflag == SPOOLES_SYMMETRIC
   || symmetryflag == SPOOLES_HERMITIAN ) {
   InpMtx_mapToUpperTriangle(mtxA) ;
}
InpMtx_changeCoordType(mtxA, INPMTX_BY_CHEVRONS) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
DenseMtx_permuteRows(mtxY, oldToNewIV) ;
symbfacIVL = SymbFac_initFromInpMtx(frontETree, mtxA) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n old-to-new permutation vector") ;
   IV_writeForHumanEye(oldToNewIV, msgFile) ;
   fprintf(msgFile, "\n\n new-to-old permutation vector") ;
   IV_writeForHumanEye(newToOldIV, msgFile) ;
   fprintf(msgFile, "\n\n front tree after permutation") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fprintf(msgFile, "\n\n input matrix after permutation") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fprintf(msgFile, "\n\n right hand side matrix after permutation") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fprintf(msgFile, "\n\n symbolic factorization") ;
   IVL_writeForHumanEye(symbfacIVL, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ------------------------------------------
   STEP 5: setup the domain decomposition map
   ------------------------------------------
*/
if ( nthread > (nfront = FrontMtx_nfront(frontmtx)) ) {
   nthread = nfront ;
}
cumopsDV = DV_new() ;
DV_init(cumopsDV, nthread, NULL) ;
ownersIV = ETree_ddMap(frontETree, type, symmetryflag,
                       cumopsDV, 1./(2.*nthread)) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n map from fronts to threads") ;
   IV_writeForHumanEye(ownersIV, msgFile) ;
   fprintf(msgFile, "\n\n factor operations for each front") ;
   DV_writeForHumanEye(cumopsDV, msgFile) ;
```

```
   fflush(msgFile) ;
}
DV_free(cumopsDV) ;
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------
   STEP 6: initialize the front matrix object
   -------------------------------------------
*/
frontmtx   = FrontMtx_new() ;
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, LOCK_IN_PROCESS, 0) ;
FrontMtx_init(frontmtx, frontETree, symbfacIVL, type, symmetryflag,
              FRONTMTX_DENSE_FRONTS, pivotingflag, LOCK_IN_PROCESS,
              0, NULL, mtxmanager, msglvl, msgFile) ;
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------------------
   STEP 7: compute the numeric factorization in parallel
   -----------------------------------------------------
*/
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, LOCK_IN_PROCESS, 1) ;
DVfill(10, cpus, 0.0) ;
IVfill(20, stats, 0) ;
rootchv = FrontMtx_MT_factorInpMtx(frontmtx, mtxA, tau, droptol,
                                 chvmanager, ownersIV, lookahead,
                                 &error, cpus, stats, msglvl, msgFile) ;
ChvManager_free(chvmanager) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n factor matrix") ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
if ( rootchv != NULL ) {
   fprintf(msgFile, "\n\n matrix found to be singular\n") ;
   exit(-1) ;
}
if ( error >= 0 ) {
   fprintf(msgFile, "\n\n fatal error at front %d", error) ;
   exit(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   --------------------------------------
   STEP 8: post-process the factorization
   --------------------------------------
*/
FrontMtx_postProcess(frontmtx, msglvl, msgFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n factor matrix after post-processing") ;
```

```
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------------------
   STEP 9: get the solve map object for the parallel solve
   -----------------------------------------------------
*/
solvemap = SolveMap_new() ;
SolveMap_ddMap(solvemap, symmetryflag, FrontMtx_upperBlockIVL(frontmtx),
               FrontMtx_lowerBlockIVL(frontmtx), nthread, ownersIV,
               FrontMtx_frontTree(frontmtx), seed, msglvl, msgFile) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------------------
   STEP 10: solve the linear system in parallel
   --------------------------------------------
*/
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
DenseMtx_zero(mtxX) ;
FrontMtx_MT_solve(frontmtx, mtxX, mtxY, mtxmanager, solvemap,
                  cpus, msglvl, msgFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n solution matrix in new ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------------------
   STEP 11: permute the solution into the original ordering
   -----------------------------------------------------
*/
DenseMtx_permuteRows(mtxX, newToOldIV) ;
if ( msglvl > 0 ) {
   fprintf(msgFile, "\n\n solution matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------
   free memory
   -----------
*/
FrontMtx_free(frontmtx) ;
DenseMtx_free(mtxX) ;
DenseMtx_free(mtxY) ;
IV_free(newToOldIV) ;
```

```
IV_free(oldToNewIV) ;
InpMtx_free(mtxA) ;
ETree_free(frontETree) ;
IVL_free(symbfacIVL) ;
SubMtxManager_free(mtxmanager) ;
Graph_free(graph) ;
SolveMap_free(solvemap) ;
IV_free(ownersIV) ;
/*--------------------------------------------------------------------*/
return(1) ; }
/*--------------------------------------------------------------------*/
```

C `allInOne.c` – A Serial *LU* Driver Program

```
/*  allInOneMPI.c  */

#include "../spoolesMPI.h"
#include "../../timings.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   ------------------------------------------------------------
   all-in-one MPI program for each process

   order, factor and solve A X = Y

   ( 1) read in matrix entries and form InpMtx object for A
   ( 2) order the system using minimum degree
   ( 3) permute the front tree
   ( 4) create the owners map IV object
   ( 5) permute the matrix A and redistribute
   ( 6) compute the symbolic factorization
   ( 7) compute the numeric factorization
   ( 8) split the factors into submatrices
   ( 9) create the submatrix map and redistribute
   (10) read in right hand side entries
        and form dense matrix DenseMtx object for Y
   (11) permute and redistribute Y
   (12) solve the linear system
   (13) gather X on processor 0

   created -- 98jun13, cca
   ------------------------------------------------------------
*/
/*--------------------------------------------------------------------*/
char            buffer[20] ;
Chv             *rootchv ;
ChvManager      *chvmanager ;
DenseMtx        *mtxX, *mtxY, *newY ;
SubMtxManager   *mtxmanager, *solvemanager ;
FrontMtx        *frontmtx ;
InpMtx          *mtxA, *newA ;
double          cutoff, droptol = 0.0, minops, tau = 100. ;
double          cpus[20] ;
double          *opcounts ;
DV              *cumopsDV ;
ETree           *frontETree ;
FILE            *inputFile, *msgFile ;
Graph           *graph ;
int             error, firsttag, ient, irow, jcol, lookahead = 0,
                msglvl, myid, nedges, nent, neqns, nmycol, nproc, nrhs,
```

```
                nrow, pivotingflag, root, seed, symmetryflag, type ;
int             stats[20] ;
int             *rowind ;
IV              *oldToNewIV, *ownedColumnsIV, *ownersIV,
                *newToOldIV, *vtxmapIV ;
IVL             *adjIVL, *symbfacIVL ;
SolveMap        *solvemap ;
/*--------------------------------------------------------------------*/
/*
   ---------------------------------------------------------------
   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 != 7 ) {
   fprintf(stdout,
      "\n usage: %s msglvl msgFile type symmetryflag pivotingflag seed"
      "\n    msglvl -- message level"
      "\n    msgFile -- message file"
      "\n    type    -- type of entries"
      "\n      1 (SPOOLES_REAL)    -- real entries"
      "\n      2 (SPOOLES_COMPLEX) -- complex entries"
      "\n    symmetryflag -- type of matrix"
      "\n      0 (SPOOLES_SYMMETRIC)    -- symmetric entries"
      "\n      1 (SPOOLES_HERMITIAN)    -- Hermitian entries"
      "\n      2 (SPOOLES_NONSYMMETRIC) -- nonsymmetric entries"
      "\n    pivotingflag -- type of pivoting"
      "\n      0 (SPOOLES_NO_PIVOTING) -- no pivoting used"
      "\n      1 (SPOOLES_PIVOTING)    -- pivoting used"
      "\n    seed -- random number seed"
      "\n    "
      "\n   note: matrix entries are read in from matrix.k.input"
      "\n         where k is the process number"
      "\n   note: rhs entries are read in from rhs.k.input"
      "\n         where k is the process number"
      "\n", argv[0]) ;
   return(0) ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else {
   sprintf(buffer, "res.%d", myid) ;
```

```
   if ( (msgFile = fopen(buffer, "w")) == NULL ) {
      fprintf(stderr, "\n fatal error in %s"
              "\n unable to open file %s\n",
              argv[0], buffer) ;
      return(-1) ;
   }
}
type         = atoi(argv[3]) ;
symmetryflag = atoi(argv[4]) ;
pivotingflag = atoi(argv[5]) ;
seed         = atoi(argv[6]) ;
IVzero(20, stats) ;
DVzero(20, cpus) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------------------
   STEP 1: read the entries from the input file
           and create the InpMtx object
   --------------------------------------------
*/
sprintf(buffer, "matrix.%d.input", myid) ;
inputFile = fopen(buffer, "r") ;
fscanf(inputFile, "%d %d %d", &neqns, &neqns, &nent) ;
mtxA = InpMtx_new() ;
InpMtx_init(mtxA, INPMTX_BY_ROWS, type, nent, 0) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le", &irow, &jcol, &value) ;
      InpMtx_inputRealEntry(mtxA, irow, jcol, value) ;
   }
} else if ( type == SPOOLES_COMPLEX ) {
   double   imag, real ;
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le %le", &irow, &jcol, &real, &imag) ;
      InpMtx_inputComplexEntry(mtxA, irow, jcol, real, imag) ;
   }
}
fclose(inputFile) ;
InpMtx_sortAndCompress(mtxA) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n input matrix") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------------
   STEP 2: read the rhs entries from the rhs input file
   and create the DenseMtx object for Y
```

```
   ----------------------------------------------------
*/
sprintf(buffer, "rhs.%d.input", myid) ;
inputFile = fopen(buffer, "r") ;
fscanf(inputFile, "%d %d", &nrow, &nrhs) ;
mtxY = DenseMtx_new() ;
DenseMtx_init(mtxY, type, 0, 0, nrow, nrhs, 1, nrow) ;
DenseMtx_rowIndices(mtxY, &nrow, &rowind) ;
if ( type == SPOOLES_REAL ) {
   double   value ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", rowind + irow) ;
      for ( jcol = 0 ; jcol < nrhs ; jcol++ ) {
         fscanf(inputFile, "%le", &value) ;
         DenseMtx_setRealEntry(mtxY, irow, jcol, value) ;
      }
   }
} if ( type == SPOOLES_COMPLEX ) {
   double   imag, real ;
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", rowind + irow) ;
      for ( jcol = 0 ; jcol < nrhs ; jcol++ ) {
         fscanf(inputFile, "%le %le", &real, &imag) ;
         DenseMtx_setComplexEntry(mtxY, irow, jcol, real, imag) ;
      }
   }
}
fclose(inputFile) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n rhs matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------------------
   STEP 2 : find a low-fill ordering
   (1) create the Graph object
   (2) order the graph using multiple minimum degree
   (3) find out who has the best ordering w.r.t. op count,
       and broadcast that front tree object
   -------------------------------------------------------
*/
graph = Graph_new() ;
adjIVL = InpMtx_MPI_fullAdjacency(mtxA, stats,
                                  msglvl, msgFile, MPI_COMM_WORLD) ;
nedges = IVL_tsize(adjIVL) ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL,
            NULL, NULL) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n graph of the input matrix") ;
```

```
   Graph_writeForHumanEye(graph, msgFile) ;
   fflush(msgFile) ;
}
frontETree = orderViaMMD(graph, seed + myid, msglvl, msgFile) ;
Graph_free(graph) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n front tree from ordering") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fflush(msgFile) ;
}
opcounts = DVinit(nproc, 0.0) ;
opcounts[myid] = ETree_nFactorOps(frontETree, type, symmetryflag) ;
MPI_Allgather((void *) &opcounts[myid], 1, MPI_DOUBLE,
              (void *) opcounts, 1, MPI_DOUBLE, MPI_COMM_WORLD) ;
minops = DVmin(nproc, opcounts, &root) ;
DVfree(opcounts) ;
frontETree = ETree_MPI_Bcast(frontETree, root,
                             msglvl, msgFile, MPI_COMM_WORLD) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n best front tree") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------------------
   STEP 3: get the permutations, permute the front tree,
           permute the matrix and right hand side.
   -------------------------------------------------------
*/
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
ETree_permuteVertices(frontETree, oldToNewIV) ;
InpMtx_permute(mtxA, IV_entries(oldToNewIV), IV_entries(oldToNewIV)) ;
if (  symmetryflag == SPOOLES_SYMMETRIC
   || symmetryflag == SPOOLES_HERMITIAN ) {
   InpMtx_mapToUpperTriangle(mtxA) ;
}
InpMtx_changeCoordType(mtxA, INPMTX_BY_CHEVRONS) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
DenseMtx_permuteRows(mtxY, oldToNewIV) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n rhs matrix in new ordering") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------
   STEP 4: generate the owners map IV object
           and the map from vertices to owners
```

```
   -------------------------------------------
*/
cutoff   = 1./(2*nproc) ;
cumopsDV = DV_new() ;
DV_init(cumopsDV, nproc, NULL) ;
ownersIV = ETree_ddMap(frontETree,
                       type, symmetryflag, cumopsDV, cutoff) ;
DV_free(cumopsDV) ;
vtxmapIV = IV_new() ;
IV_init(vtxmapIV, neqns, NULL) ;
IVgather(neqns, IV_entries(vtxmapIV),
         IV_entries(ownersIV), ETree_vtxToFront(frontETree)) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n map from fronts to owning processes") ;
   IV_writeForHumanEye(ownersIV, msgFile) ;
   fprintf(msgFile, "\n\n map from vertices to owning processes") ;
   IV_writeForHumanEye(vtxmapIV, msgFile) ;
   fflush(msgFile) ;
}
/*------------------------------------------------------------------*/
/*
   ---------------------------------------------------
   STEP 5: redistribute the matrix and right hand side
   ---------------------------------------------------
*/
firsttag = 0 ;
newA = InpMtx_MPI_split(mtxA, vtxmapIV, stats,
                        msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
firsttag++ ;
InpMtx_free(mtxA) ;
mtxA = newA ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n split InpMtx") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
newY = DenseMtx_MPI_splitByRows(mtxY, vtxmapIV, stats, msglvl,
                                msgFile, firsttag, MPI_COMM_WORLD) ;
DenseMtx_free(mtxY) ;
mtxY = newY ;
firsttag += nproc ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n split DenseMtx Y") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
/*------------------------------------------------------------------*/
/*
   ------------------------------------------
   STEP 6: compute the symbolic factorization
```

```
   ------------------------------------------
*/
symbfacIVL = SymbFac_MPI_initFromInpMtx(frontETree, ownersIV, mtxA,
                     stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
firsttag += frontETree->nfront ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n local symbolic factorization") ;
   IVL_writeForHumanEye(symbfacIVL, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------
   STEP 7: initialize the front matrix
   -----------------------------------
*/
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, NO_LOCK, 0) ;
frontmtx = FrontMtx_new() ;
FrontMtx_init(frontmtx, frontETree, symbfacIVL, type, symmetryflag,
              FRONTMTX_DENSE_FRONTS, pivotingflag, NO_LOCK, myid,
              ownersIV, mtxmanager, msglvl, msgFile) ;
/*--------------------------------------------------------------------*/
/*
   ---------------------------------
   STEP 8: compute the factorization
   ---------------------------------
*/
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, NO_LOCK, 0) ;
rootchv = FrontMtx_MPI_factorInpMtx(frontmtx, mtxA, tau, droptol,
                     chvmanager, ownersIV, lookahead, &error, cpus,
                     stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
ChvManager_free(chvmanager) ;
firsttag += 3*frontETree->nfront + 2 ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n numeric factorization") ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
if ( error >= 0 ) {
   fprintf(stderr,
          "\n proc %d : factorization error at front %d", myid, error) ;
   MPI_Finalize() ;
   exit(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   ---------------------------------------------------
   STEP 9: post-process the factorization and split
   the factor matrices into submatrices
```

```
   ----------------------------------------------
*/
FrontMtx_MPI_postProcess(frontmtx, ownersIV, stats, msglvl,
                         msgFile, firsttag, MPI_COMM_WORLD) ;
firsttag += 5*nproc ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n numeric factorization after post-processing");
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------
   STEP 10: create the solve map object
   -----------------------------------
*/
solvemap = SolveMap_new() ;
SolveMap_ddMap(solvemap, symmetryflag,
               FrontMtx_upperBlockIVL(frontmtx),
               FrontMtx_lowerBlockIVL(frontmtx),
               nproc, ownersIV, FrontMtx_frontTree(frontmtx),
               seed, msglvl, msgFile);
if ( msglvl > 3 ) {
   SolveMap_writeForHumanEye(solvemap, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ---------------------------------------------------
   STEP 11: redistribute the submatrices of the factors
   ---------------------------------------------------
*/
FrontMtx_MPI_split(frontmtx, solvemap,
                   stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n numeric factorization after split") ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------------
   STEP 13: permute and redistribute Y if necessary
   -----------------------------------------------
*/
if ( FRONTMTX_IS_PIVOTING(frontmtx) ) {
   IV   *rowmapIV ;
/*
   ---------------------------------------------------------
   pivoting has taken place, redistribute the right hand side
   to match the final rows and columns in the fronts
```

```
   ----------------------------------------------------------
*/
   rowmapIV = FrontMtx_MPI_rowmapIV(frontmtx, ownersIV, msglvl,
                                    msgFile, MPI_COMM_WORLD) ;
   newY = DenseMtx_MPI_splitByRows(mtxY, rowmapIV, stats, msglvl,
                                   msgFile, firsttag, MPI_COMM_WORLD) ;
   DenseMtx_free(mtxY) ;
   mtxY = newY ;
   IV_free(rowmapIV) ;
}
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n rhs matrix after split") ;
   DenseMtx_writeForHumanEye(mtxY, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ------------------------------------------
   STEP 14: create a solution DenseMtx object
   ------------------------------------------
*/
ownedColumnsIV = FrontMtx_ownedColumnsIV(frontmtx, myid, ownersIV,
                                         msglvl, msgFile) ;
nmycol = IV_size(ownedColumnsIV) ;
mtxX = DenseMtx_new() ;
if ( nmycol > 0 ) {
   DenseMtx_init(mtxX, type, 0, 0, nmycol, nrhs, 1, nmycol) ;
   DenseMtx_rowIndices(mtxX, &nrow, &rowind) ;
   IVcopy(nmycol, rowind, IV_entries(ownedColumnsIV)) ;
}
/*--------------------------------------------------------------------*/
/*
   --------------------------------
   STEP 15: solve the linear system
   --------------------------------
*/
solvemanager = SubMtxManager_new() ;
SubMtxManager_init(solvemanager, NO_LOCK, 0) ;
FrontMtx_MPI_solve(frontmtx, mtxX, mtxY, solvemanager, solvemap, cpus,
                   stats, msglvl, msgFile, firsttag, MPI_COMM_WORLD) ;
SubMtxManager_free(solvemanager) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n solution in new ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   --------------------------------------------------------
   STEP 15: permute the solution into the original ordering
            and assemble the solution onto processor zero
   --------------------------------------------------------
```

```
*/
DenseMtx_permuteRows(mtxX, newToOldIV) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n solution in old ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
IV_fill(vtxmapIV, 0) ;
firsttag++ ;
mtxX = DenseMtx_MPI_splitByRows(mtxX, vtxmapIV, stats, msglvl, msgFile,
                                firsttag, MPI_COMM_WORLD) ;
if ( myid == 0 && msglvl > 0 ) {
   fprintf(msgFile, "\n\n complete solution in old ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
MPI_Finalize() ;

return(1) ; }
/*--------------------------------------------------------------------*/
```

D allInOne.c – A Serial QR Driver Program

```
/*  QRallInOne.c  */

#include "../../misc.h"
#include "../../FrontMtx.h"
#include "../../SymbFac.h"

/*--------------------------------------------------------------------*/
int
main ( int argc, char *argv[] ) {
/*
   --------------------------------------------------
   QR all-in-one program
   (1) read in matrix entries and form InpMtx object
       of A and A^TA
   (2) form Graph object of A^TA
   (3) order matrix and form front tree
   (4) get the permutation, permute the matrix and
       front tree and get the symbolic factorization
   (5) compute the numeric factorization
   (6) read in right hand side entries
   (7) compute the solution

   created -- 98jun11, cca
   --------------------------------------------------
*/
/*--------------------------------------------------------------------*/
char            *matrixFileName, *rhsFileName ;
ChvManager      *chvmanager ;
DenseMtx        *mtxB, *mtxX ;
double          facops, imag, real, value ;
double          cpus[10] ;
ETree           *frontETree ;
FILE            *inputFile, *msgFile ;
FrontMtx        *frontmtx ;
Graph           *graph ;
int             ient, irow, jcol, jrhs, jrow, msglvl, neqns,
                nedges, nent, nrhs, nrow, seed, type ;
InpMtx          *mtxA ;
IV              *newToOldIV, *oldToNewIV ;
IVL             *adjIVL, *symbfacIVL ;
SubMtxManager   *mtxmanager ;
/*--------------------------------------------------------------------*/
/*
   --------------------
   get input parameters
   --------------------
*/
if ( argc != 7 ) {
   fprintf(stdout,
```

```
      "\n usage: %s msglvl msgFile type matrixFileName rhsFileName seed"
      "\n    msglvl -- message level"
      "\n    msgFile -- message file"
      "\n    type    -- type of entries"
      "\n      1 (SPOOLES_REAL)    -- real entries"
      "\n      2 (SPOOLES_COMPLEX) -- complex entries"
      "\n    matrixFileName -- matrix file name, format"
      "\n       nrow ncol nent"
      "\n       irow jcol entry"
      "\n        ..."
      "\n        note: indices are zero based"
      "\n    rhsFileName -- right hand side file name, format"
      "\n       nrow "
      "\n       entry[0]"
      "\n       ..."
      "\n       entry[nrow-1]"
      "\n    seed -- random number seed, used for ordering"
      "\n", argv[0]) ;
   return(0) ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else if ( (msgFile = fopen(argv[2], "a")) == NULL ) {
   fprintf(stderr, "\n fatal error in %s"
           "\n unable to open file %s\n",
           argv[0], argv[2]) ;
   return(-1) ;
}
type           = atoi(argv[3]) ;
matrixFileName = argv[4] ;
rhsFileName    = argv[5] ;
seed           = atoi(argv[6]) ;
/*--------------------------------------------------------------------*/
/*
   --------------------------------------------
   STEP 1: read the entries from the input file
   and create the InpMtx object of A
   --------------------------------------------
*/
inputFile = fopen(matrixFileName, "r") ;
fscanf(inputFile, "%d %d %d", &nrow, &neqns, &nent) ;
mtxA = InpMtx_new() ;
InpMtx_init(mtxA, INPMTX_BY_ROWS, type, nent, 0) ;
if ( type == SPOOLES_REAL ) {
   for ( ient = 0 ; ient < nent ; ient++ ) {
      fscanf(inputFile, "%d %d %le", &irow, &jcol, &value) ;
      InpMtx_inputRealEntry(mtxA, irow, jcol, value) ;
   }
} else {
   for ( ient = 0 ; ient < nent ; ient++ ) {
```

```
      fscanf(inputFile, "%d %d %le %le", &irow, &jcol, &real, &imag) ;
      InpMtx_inputComplexEntry(mtxA, irow, jcol, real, imag) ;
   }
}
fclose(inputFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n input matrix") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------
   STEP 2: read the right hand side entries
   ----------------------------------------
*/
inputFile = fopen(rhsFileName, "r") ;
fscanf(inputFile, "%d %d", &nrow, &nrhs) ;
mtxB = DenseMtx_new() ;
DenseMtx_init(mtxB, type, 0, 0, nrow, nrhs, 1, nrow) ;
DenseMtx_zero(mtxB) ;
if ( type == SPOOLES_REAL ) {
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le", &value) ;
         DenseMtx_setRealEntry(mtxB, jrow, jrhs, value) ;
      }
   }
} else {
   for ( irow = 0 ; irow < nrow ; irow++ ) {
      fscanf(inputFile, "%d", &jrow) ;
      for ( jrhs = 0 ; jrhs < nrhs ; jrhs++ ) {
         fscanf(inputFile, "%le %le", &real, &imag) ;
         DenseMtx_setComplexEntry(mtxB, jrow, jrhs, real, imag) ;
      }
   }
}
fclose(inputFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n rhs matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxB, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------------
   STEP 3 : find a low-fill ordering
   (1) create the Graph object for A^TA or A^HA
   (2) order the graph using multiple minimum degree
   -------------------------------------------------
```

```
*/
graph = Graph_new() ;
adjIVL = InpMtx_adjForATA(mtxA) ;
nedges = IVL_tsize(adjIVL) ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL,
            NULL, NULL) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n graph of A^T A") ;
   Graph_writeForHumanEye(graph, msgFile) ;
   fflush(msgFile) ;
}
frontETree = orderViaMMD(graph, seed, msglvl, msgFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n front tree from ordering") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------------
   STEP 4: get the permutation, permute the matrix and
           front tree and get the symbolic factorization
   ----------------------------------------------------
*/
oldToNewIV = ETree_oldToNewVtxPerm(frontETree) ;
newToOldIV = ETree_newToOldVtxPerm(frontETree) ;
InpMtx_permute(mtxA, NULL, IV_entries(oldToNewIV)) ;
InpMtx_changeStorageMode(mtxA, INPMTX_BY_VECTORS) ;
symbfacIVL = SymbFac_initFromGraph(frontETree, graph) ;
IVL_overwrite(symbfacIVL, oldToNewIV) ;
IVL_sortUp(symbfacIVL) ;
ETree_permuteVertices(frontETree, oldToNewIV) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n old-to-new permutation vector") ;
   IV_writeForHumanEye(oldToNewIV, msgFile) ;
   fprintf(msgFile, "\n\n new-to-old permutation vector") ;
   IV_writeForHumanEye(newToOldIV, msgFile) ;
   fprintf(msgFile, "\n\n front tree after permutation") ;
   ETree_writeForHumanEye(frontETree, msgFile) ;
   fprintf(msgFile, "\n\n input matrix after permutation") ;
   InpMtx_writeForHumanEye(mtxA, msgFile) ;
   fprintf(msgFile, "\n\n symbolic factorization") ;
   IVL_writeForHumanEye(symbfacIVL, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ------------------------------------------
   STEP 5: initialize the front matrix object
   ------------------------------------------
*/
```

```
frontmtx = FrontMtx_new() ;
mtxmanager = SubMtxManager_new() ;
SubMtxManager_init(mtxmanager, NO_LOCK, 0) ;
if ( type == SPOOLES_REAL ) {
   FrontMtx_init(frontmtx, frontETree, symbfacIVL, type,
                 SPOOLES_SYMMETRIC, FRONTMTX_DENSE_FRONTS,
                 SPOOLES_NO_PIVOTING, NO_LOCK, 0, NULL,
                 mtxmanager, msglvl, msgFile) ;
} else {
   FrontMtx_init(frontmtx, frontETree, symbfacIVL, type,
                 SPOOLES_HERMITIAN, FRONTMTX_DENSE_FRONTS,
                 SPOOLES_NO_PIVOTING, NO_LOCK, 0, NULL,
                 mtxmanager, msglvl, msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -----------------------------------------
   STEP 6: compute the numeric factorization
   -----------------------------------------
*/
chvmanager = ChvManager_new() ;
ChvManager_init(chvmanager, NO_LOCK, 1) ;
DVzero(10, cpus) ;
facops = 0.0 ;
FrontMtx_QR_factor(frontmtx, mtxA, chvmanager,
                   cpus, &facops, msglvl, msgFile) ;
ChvManager_free(chvmanager) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n factor matrix") ;
   fprintf(msgFile, "\n facops = %9.2f", facops) ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   --------------------------------------
   STEP 7: post-process the factorization
   --------------------------------------
*/
FrontMtx_postProcess(frontmtx, msglvl, msgFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n factor matrix after post-processing") ;
   FrontMtx_writeForHumanEye(frontmtx, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------
   STEP 8: solve the linear system
   -------------------------------
*/
```

```
mtxX = DenseMtx_new() ;
DenseMtx_init(mtxX, type, 0, 0, neqns, nrhs, 1, neqns) ;
FrontMtx_QR_solve(frontmtx, mtxA, mtxX, mtxB, mtxmanager,
                  cpus, msglvl, msgFile) ;
if ( msglvl > 1 ) {
   fprintf(msgFile, "\n\n solution matrix in new ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------------------
   STEP 9: permute the solution into the original ordering
   -------------------------------------------------------
*/
DenseMtx_permuteRows(mtxX, newToOldIV) ;
if ( msglvl > 0 ) {
   fprintf(msgFile, "\n\n solution matrix in original ordering") ;
   DenseMtx_writeForHumanEye(mtxX, msgFile) ;
   fflush(msgFile) ;
}
/*--------------------------------------------------------------------*/
/*
   ------------------------
   free the working storage
   ------------------------
*/
InpMtx_free(mtxA) ;
FrontMtx_free(frontmtx) ;
Graph_free(graph) ;
DenseMtx_free(mtxX) ;
DenseMtx_free(mtxB) ;
ETree_free(frontETree) ;
IV_free(newToOldIV) ;
IV_free(oldToNewIV) ;
IVL_free(symbfacIVL) ;
SubMtxManager_free(mtxmanager) ;
/*--------------------------------------------------------------------*/
return(1) ; }
/*--------------------------------------------------------------------*/
```