Integrating the **SPOOLES** 2.2 Sparse Linear Algebra Library into the **LANCZOS** Block-shifted Lanczos Eigensolver

Cleve Ashcraft
Boeing Phantom Works[1]

Jim Patterson
Boeing Phantom Works[2]

January 2, 1999

[1]P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124, `cleve.ashcraft@boeing.com`. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

[2]P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124, `pattersn@redwood.rt.cs.boeing.com`. 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

Chapter 1

Introduction

The Lanczos eigensolver finds selected eigenvalues and eigenvectors of $AX = BX\Lambda$, where X are eigenvectors and Λ is a diagonal matrix whose elements are eigenvalues. Three types of eigenproblems are supported.

- An "ordinary" eigenvalue problem where A is symmetric and $B = I$.
- An "vibration" eigenvalue problem where A is symmetric and B is symmetric positive semidefinite.
- A "buckling" eigenvalue problem A is symmetric positive semidefinite and B is symmetric.

For the vibration and buckling problems, there must exist a σ that is not an eigenvalue such that $A - \sigma B$ is nonsingular, i.e., A and B cannot share the same null space.

During the computations, the eigensolver requires the following sparse linear algebra computations.

- Sparse factorizations of the form $A - \sigma B$.
- Solves of the form $(A - \sigma B)Z = Y$.
- Multiplies of the form $Z = BY$ (for the vibration problem) or $Z = AY$ (for the buckling problem).

The Lanczos eigensolver has defined a specific interface with an external linear algebra package to perform these three operations. The eigensolver currently interfaces with the **BCSLIB-EXT** linear solver in a serial environment and the **SPOOLES** linear solver in serial, multithreaded and MPI environments.

This paper documents the **SPOOLES** objects and functions that interface with the eigensolver. The three following chapters describe the serial, multithreaded and MPI objects, their data structures, and their methods. The appendix contains listings of three driver programs to exercise the eigensolver using the **SPOOLES** library.

Symmetric permutations of the eigensystem do not change the eigenvalues, and the eigenvectors can be easily constructed using the permutation matrix.

$$AX = BX\Lambda \longrightarrow \widehat{A}\widehat{X} = \widehat{B}\widehat{X}\Lambda \quad \text{where} \quad \widehat{A} = PAP^T, \quad \widehat{B} = PBP^T, \quad \text{and} \quad \widehat{X} = PX$$

The linear algebra package is free to use any permutation matrix P to most efficiently perform the factorizations and solves involving $\widehat{A}$ and $\widehat{B}$. This permutation matrix P is typically found by ordering the graph of $A + B$ using a variant of minimum degree or nested dissection. The ordering is performed prior to any action by the eigensolver. This "setup phase" includes more than just finding the permutation matrix, e.g., various data structures must be initialized. In a parallel environment, there is even more setup work to do, analyzing the factorization and solves and specifying which threads or processors perform what computations and store what data. In a distributed environment, the entries of A and B must also be distributed among the processors in preparation for the factors and multiplies.

For each of the three environments — serial, multithreaded and MPI — the **SPOOLES** solver has constructed a "bridge" object to span the interface between the linear system solver and the eigensolver. Each of the `Bridge`, `BridgeMT` and `BridgeMPI` objects have five methods: set-up, factor, solve, matrix-multiply and cleanup. The factor, solve and matrix-multiply methods follow the calling sequence convention imposed by the eigensolver, and are passed to the eigensolver at the beginning of the Lanczos run. The set-up method is called prior to the eigensolver, and the cleanup method is called after the eigenvalues and eigenvectors have been determined.

Chapter 2

The Serial Bridge Object and Driver

2.1 The Bridge Data Structure

The Bridge structure has the following fields.

- int prbtype : problem type
  - 1 — vibration, a multiply with B is required.
  - 2 — buckling, a multiply with A is required.
  - 3 — simple, no multiply is required.
- int neqns : number of equations, i.e., number of vertices in the graph.
- int mxbsz : block size for the Lanczos process.
- int seed : random number seed used in the ordering.
- InpMtx *A : matrix object for A
- InpMtx *B : matrix object for B
- Pencil *pencil : object to hold linear combination of A and B.
- ETree *frontETree : object that defines the factorizations, e.g., the number of fronts, the tree they form, the number of internal and external rows for each front, and the map from vertices to the front where it is contained.
- IVL *symbfacIVL : object that contains the symbolic factorization of the matrix.
- SubMtxManager *mtxmanager : object that manages the SubMtx objects that store the factor entries and are used in the solves.
- FrontMtx *frontmtx : object that stores the L, D and U factor matrices.
- IV *oldToNewIV : object that stores old-to-new permutation vector.
- IV *newToOldIV : object that stores new-to-old permutation vector.
- DenseMtx *X : dense matrix object that is used during the matrix multiples and solves.
- DenseMtx *Y : dense matrix object that is used during the matrix multiples and solves.

- `int msglvl` : message level for output. When 0, no output, When 1, just statistics and cpu times. When greater than 1, more and more output.
- `FILE *msgFile` : message file for output. When `msglvl` > 0, `msgFile` must not be `NULL`.

2.2 Prototypes and descriptions of `Bridge` methods

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

1. ```
   int Setup ( void *data, int *pprbtype, int *pneqns, int *pmxbsz,
               InpMtx *A, InpMtx *B, int *pseed, int *pmsglvl, FILE *msgFile ) ;
   ```
   All calling sequence parameters are pointers to more easily allow an interface with Fortran.
   - `void *data` — a pointer to the `Bridge` object.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - `1` — vibration, a multiply with B is required.
     - `2` — buckling, a multiply with A is required.
     - `3` — simple, no multiply is required.
   - `int *pneqns` — `*pneqns` is the number of equations.
   - `int *pmxbsz` — `*pmxbsz` is an upper bound on the block size.
   - `InpMtx *A` — `A` is a **SPOOLES** object that holds the matrix A.
   - `InpMtx *B` — `B` is a **SPOOLES** object that holds the matrix B. For an ordinary eigenproblem, B is the identity and `B` is `NULL`.
   - `int *pseed` — `*pseed` is a random number seed.
   - `int *pmsglvl` — `*pmsglvl` is a message level for the bridge methods and the **SPOOLES** methods they call.
   - `FILE *pmsglvl` — `msgFile` is the message file for the bridge methods and the **SPOOLES** methods they call.

   This method must be called in the driver program prior to invoking the eigensolver via a call to `lanczos_run()`. It then follows this sequence of action.
   - The method begins by checking all the input data, and setting the appropriate fields of the `Bridge` object.
   - The `pencil` object is initialized with `A` and `B`.
   - `A` and `B` are converted to storage by rows and vector mode.
   - A `Graph` object is created that contains the sparsity pattern of the union of `A` and `B`.
   - The graph is ordered by first finding a recursive dissection partition, and then evaluating the orderings produced by nested dissection and multisection, and choosing the better of the two. The `frontETree` object is produced and placed into the `bridge` object.
   - Old-to-new and new-to-old permutations are extracted from the front tree and loaded into the `Bridge` object.
   - The vertices in the front tree are permuted, as well as the entries in `A` and `B`. Entries in the lower triangle of `A` and `B` are mapped into the upper triangle, and the storage modes of `A` and `B` are changed to chevrons and vectors, in preparation for the first factorization.
   - The symbolic factorization is then computed and loaded in the `Bridge` object.
   ```

- A `FrontMtx` object is created to hold the factorization and loaded into the `Bridge` object.
- A `SubMtxManager` object is created to hold the factor's submatrices and loaded into the `Bridge` object.
- Two `DenseMtx` objects are created to be used during the matrix multiplies and solves.

The `A` and `B` matrices are now in their permuted ordering, i.e., PAP^T and PBP^T, and all data structures are with respect to this ordering. After the Lanczos run completes, any generated eigenvectors must be permuted back into their original ordering using the `oldToNewIV` and `newToOldIV` objects.

Return value:

1	normal return	-6	`pmxbsz` is `NULL`
-1	`data` is `NULL`	-7	`*pmxbsz` is invalid
-2	`pprbtype` is `NULL`	-8	`A` and `B` are `NULL`
-3	`*pprbtype` is invalid	-9	`seed` is `NULL`
-4	`pneqns` is `NULL`	-10	`msglvl` is `NULL`
-5	`*pneqns` is invalid	-11	`msglvl` >0 and `msgFile` is `NULL`

2. `void Factor ( double *psigma, double *ppvttol, void *data, int *pinertia, int *perror ) ;`

   This method computes the factorization of $A - \sigma B$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `double *psigma` — the shift parameter σ is found in `*psigma`.
   - `double *ppvttol` — the pivot tolerance is found in `*ppvttol`. When $*$`ppvttol` $= 0.0$, the factorization is computed without pivoting for stability. When $*$`ppvttol` > 0.0, the factorization is computed with pivoting for stability, and all offdiagonal entries have magnitudes bounded above by $1/(*$`ppvttol`$)$.
   - `void *data` — a pointer to the `Bridge` object.
   - `int *pinertia` — on return, `*pinertia` holds the number of negative eigenvalues.
   - `int *perror` — on return, `*perror` holds an error code.

1	error in the factorization	-2	`ppvttol` is `NULL`
0	normal return	-3	`data` is `NULL`
-1	`psigma` is `NULL`	-4	`pinertia` is `NULL`

3. `void MatMul ( int *pnrows, int *pncols, double X[], double Y[],`
   `int *pprbtype, void *data ) ;`

   This method computes a multiply of the form $Y = IX$, $Y = AX$ or $Y = BX$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of columns in X and Y.
   - `double X[]` — this is the X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the Y matrix, stored column major with leading dimension `*pnrows`.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - `1` — vibration, a multiply with B is required.
     - `2` — buckling, a multiply with A is required.
     - `3` — simple, no multiply is required.
   - `void *data` — a pointer to the `Bridge` object.

4. `void Solve ( int *pnrows, int *pncols, double X[], double Y[], void *data, int *perror ) ;`

   This method solves $(A-\sigma B)X = Y$, where $(A-\sigma B)$ has been factored by a previous call to `Factor()`. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of columns in X and Y.
   - `double X[]` — this is the X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the Y matrix, stored column major with leading dimension `*pnrows`.
   - `void *data` — a pointer to the `Bridge` object.
   - `int *perror` — on return, `*perror` holds an error code.

1	normal return	-3	`X` is `NULL`
-1	`pnrows` is `NULL`	-4	`Y` is `NULL`
-2	`pncols` is `NULL`	-5	`data` is `NULL`

5. `int Cleanup ( void *data ) ;`

   This method releases all the storage used by the **SPOOLES** library functions.

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

2.3 The `testSerial` Driver Program

A complete listing of the serial driver program is found in chapter A. The program is invoked by this command sequence.

```
testSerial msglvl msgFile parmFile seed inFileA inFileB
```

where

- `msglvl` is the message level for the `Bridge` methods and the **SPOOLES** software.
- `msgFile` is the message file for the `Bridge` methods and the **SPOOLES** software.
- `parmFile` is the input file for the parameters of the eigensystem to be solved.
- `seed` is a random number seed used by the **SPOOLES** software.
- `inFileA` is the Harwell-Boeing file for the matrix A.
- `inFileB` is the Harwell-Boeing file for the matrix B.

This program is executed for some sample matrices by the `do_ST_*` shell scripts in the `drivers` directory.

Here is a short description of the steps in the driver program. See Chapter A for the listing.

1. The command line inputs are decoded.
2. The header of the Harwell-Boeing file for A is read. This yields the number of equations.
3. The parameters that define the eigensystem to be solved are read in from the `parmFile` file.
4. The Lanczos eigensolver workspace is initialized.

5. The Lanczos communication structure is filled with some parameters.
6. The A and possibly B matrices are read in from the Harwell-Boeing files and converted into `InpMtx` objects from the **SPOOLES** library.
7. The linear solver environment is then initialized via a call to `Setup()`.
8. The eigensolver is invoked via a call to `lanczos_run()`. The `FactorMT()`, `SolveMT()` and `MatMulMT()` methods are passed to this routine.
9. The eigenvalues are extracted and printed via a call to `lanczos_eigenvalues()`.
10. The eigenvectors are extracted and printed via calls to `lanczos_eigenvector()`.
11. The eigensolver working storage is free'd via a call to `lanczos_free()`.
12. The linear solver working storage is free'd via a call to `Cleanup()`.

Chapter 3

The Multithreaded Bridge Object and Driver

3.1 The BridgeMT Data Structure

The `BridgeMT` structure has the following fields.

- `int prbtype` : problem type
  - `1` — vibration, a multiply with B is required.
  - `2` — buckling, a multiply with A is required.
  - `3` — simple, no multiply is required.
- `int neqns` : number of equations, i.e., number of vertices in the graph.
- `int mxbsz` : block size for the Lanczos process.
- `int nthread` : number of threads to use.
- `int seed` : random number seed used in the ordering.
- `InpMtx *A` : matrix object for A
- `InpMtx *B` : matrix object for B
- `Pencil *pencil` : object to hold linear combination of A and B.
- `ETree *frontETree` : object that defines the factorizations, e.g., the number of fronts, the tree they form, the number of internal and external rows for each front, and the map from vertices to the front where it is contained.
- `IVL *symbfacIVL` : object that contains the symbolic factorization of the matrix.
- `SubMtxManager *mtxmanager` : object that manages the `SubMtx` objects that store the factor entries and are used in the solves.
- `FrontMtx *frontmtx` : object that stores the L, D and U factor matrices.
- `IV *oldToNewIV` : object that stores old-to-new permutation vector.

- `IV *newToOldIV` : object that stores new-to-old permutation vector.
- `DenseMtx *X` : dense matrix object that is used during the matrix multiples and solves.
- `DenseMtx *Y` : dense matrix object that is used during the matrix multiples and solves.
- `IV *ownersIV` : object that maps fronts to owning threads for the factorization and matrix-multiplies.
- `SolveMap *solvemap` : object that maps factor submatrices to owning threads for the solve.
- `int msglvl` : message level for output. When 0, no output, When 1, just statistics and cpu times. When greater than 1, more and more output.
- `FILE *msgFile` : message file for output. When `msglvl` $>$ 0, `msgFile` must not be `NULL`.

3.2 Prototypes and descriptions of `BridgeMT` methods

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

1.
```
int SetupMT ( void *data, int *pprbtype, int *pneqns,
              int *pmxbsz, InpMtx *A, InpMtx *B, int *pseed,
              int *pnthread, int *pmsglvl, FILE *msgFile ) ;
```

   All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `void *data` — a pointer to the `BridgeMT` object.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - `1` — vibration, a multiply with B is required.
     - `2` — buckling, a multiply with A is required.
     - `3` — simple, no multiply is required.
   - `int *pneqns` — `*pneqns` is the number of equations.
   - `int *pmxbsz` — `*pmxbsz` is an upper bound on the block size.
   - `InpMtx *A` — `A` is a **SPOOLES** object that holds the matrix A.
   - `InpMtx *B` — `B` is a **SPOOLES** object that holds the matrix B. For an ordinary eigenproblem, B is the identity and `B` is `NULL`.
   - `int *pseed` — `*pseed` is a random number seed.
   - `int *pnthread` — `*pnthread` is the number of threads to use during the factorizations, solves and matrix-multiplies.
   - `int *pmsglvl` — `*pmsglvl` is a message level for the bridge methods and the **SPOOLES** methods they call.
   - `FILE *pmsglvl` — `msgFile` is the message file for the bridge methods and the **SPOOLES** methods they call.

   This method must be called in the driver program prior to invoking the eigensolver via a call to `lanczos_run()`. It then follows this sequence of action.

   - The method begins by checking all the input data, and setting the appropriate fields of the `BridgeMT` object.
   - The `pencil` object is initialized with `A` and `B`.

- A and B are converted to storage by rows and vector mode.
- A Graph object is created that contains the sparsity pattern of the union of A and B.
- The graph is ordered by first finding a recursive dissection partition, and then evaluating the orderings produced by nested dissection and multisection, and choosing the better of the two. The frontETree object is produced and placed into the bridge object.
- Old-to-new and new-to-old permutations are extracted from the front tree and loaded into the BridgeMT object.
- The vertices in the front tree are permuted, as well as the entries in A and B. Entries in the lower triangle of A and B are mapped into the upper triangle, and the storage modes of A and B are changed to chevrons and vectors, in preparation for the first factorization.
- The symbolic factorization is then computed and loaded in the BridgeMT object.
- A FrontMtx object is created to hold the factorization and loaded into the BridgeMT object.
- A SubMtxManager object is created to hold the factor's submatrices and loaded into the BridgeMT object.
- Two DenseMtx objects are created to be used during the matrix multiplies and solves.
- The map from fronts to their owning threads is computed and stored in the ownersIV object.
- The map from factor submatrices to their owning threads is computed and stored in the solvemap object.

The A and B matrices are now in their permuted ordering, i.e., PAP^T and PBP^T, and all data structures are with respect to this ordering. After the Lanczos run completes, any generated eigenvectors must be permuted back into their original ordering using the oldToNewIV and newToOldIV objects.

Return value:

1	normal return	-7	*pmxbsz is invalid
-1	data is NULL	-8	A and B are NULL
-2	pprbtype is NULL	-9	seed is NULL
-3	*pprbtype is invalid	-10	msglvl is NULL
-4	pneqns is NULL	-11	msglvl >0 and msgFile is NULL
-5	*pneqns is invalid	-12	pnthread is NULL
-6	pmxbsz is NULL	-13	*pnthread is invalid

2.
```
void FactorMT ( double *psigma, double *ppvttol, void *data,
                int *pinertia, int *perror ) ;
```

This method computes the factorization of $A - \sigma B$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

- double *psigma — the shift parameter σ is found in *psigma.
- double *ppvttol — the pivot tolerance is found in *ppvttol. When *ppvttol $= 0.0$, the factorization is computed without pivoting for stability. When *ppvttol > 0.0, the factorization is computed with pivoting for stability, and all offdiagonal entries have magnitudes bounded above by 1/(*ppvttol).
- void *data — a pointer to the BridgeMT object.
- int *pinertia — on return, *pinertia holds the number of negative eigenvalues.
- int *perror — on return, *perror holds an error code.

1	error in the factorization	-2	ppvttol is NULL
0	normal return	-3	data is NULL
-1	psigma is NULL	-4	pinertia is NULL

3. `void MatMulMT ( int *pnrows, int *pncols, double X[], double Y[],`
   `int *pprbtype, void *data ) ;`

   This method computes a multiply of the form $Y = IX$, $Y = AX$ or $Y = BX$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of columns in X and Y.
   - `double X[]` — this is the X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the Y matrix, stored column major with leading dimension `*pnrows`.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - `1` — vibration, a multiply with B is required.
     - `2` — buckling, a multiply with A is required.
     - `3` — simple, no multiply is required.
   - `void *data` — a pointer to the `BridgeMT` object.

4. `void SolveMT ( int *pnrows, int *pncols, double X[], double Y[],`
   `void *data, int *perror ) ;`

   This method solves $(A-\sigma B)X = Y$, where $(A-\sigma B)$ has been factored by a previous call to `Factor()`. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of columns in X and Y.
   - `double X[]` — this is the X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the Y matrix, stored column major with leading dimension `*pnrows`.
   - `void *data` — a pointer to the `BridgeMT` object.
   - `int *perror` — on return, `*perror` holds an error code.

1	normal return	-3	`X` is `NULL`
-1	`pnrows` is `NULL`	-4	`Y` is `NULL`
-2	`pncols` is `NULL`	-5	`data` is `NULL`

5. `int CleanupMT ( void *data ) ;`

   This method releases all the storage used by the **SPOOLES** library functions.

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

3.3 The `testMT` Driver Program

A complete listing of the multithreaded driver program is found in chapter B. The program is invoked by this command sequence.

```
testMT msglvl msgFile parmFile seed nthread inFileA inFileB
```

where

- `msglvl` is the message level for the `BridgeMT` methods and the **SPOOLES** software.
- `msgFile` is the message file for the `BridgeMT` methods and the **SPOOLES** software.

- `parmFile` is the input file for the parameters of the eigensystem to be solved.
- `seed` is a random number seed used by the **SPOOLES** software.
- `nthread` is the number of threads to use in the factors, solves and matrix-multiplies.
- `inFileA` is the Harwell-Boeing file for the matrix A.
- `inFileB` is the Harwell-Boeing file for the matrix B.

This program is executed for some sample matrices by the `do_ST_*` shell scripts in the `drivers` directory.

Here is a short description of the steps in the driver program. See Chapter A for the listing.

1. The command line inputs are decoded.
2. The header of the Harwell-Boeing file for A is read. This yields the number of equations.
3. The parameters that define the eigensystem to be solved are read in from the `parmFile` file.
4. The Lanczos eigensolver workspace is initialized.
5. The Lanczos communication structure is filled with some parameters.
6. The A and possibly B matrices are read in from the Harwell-Boeing files and converted into `InpMtx` objects from the **SPOOLES** library.
7. The linear solver environment is then initialized via a call to `SetupMT()`.
8. The eigensolver is invoked via a call to `lanczos_run()`. The `FactorMT()`, `SolveMT()` and `MatMulMT()` methods are passed to this routine.
9. The eigenvalues are extracted and printed via a call to `lanczos_eigenvalues()`.
10. The eigenvectors are extracted and printed via calls to `lanczos_eigenvector()`.
11. The eigensolver working storage is free'd via a call to `lanczos_free()`.
12. The linear solver working storage is free'd via a call to `CleanupMT()`.

Chapter 4

The MPI Bridge Object and Driver

4.1 The BridgeMPI Data Structure

The `BridgeMPI` structure has the following fields.

- `int prbtype` : problem type
  - `1` — vibration, a multiply with B is required.
  - `2` — buckling, a multiply with A is required.
  - `3` — simple, no multiply is required.
- `int neqns` : number of equations, i.e., number of vertices in the graph.
- `int mxbsz` : block size for the Lanczos process.
- `int nproc` : number of processors.
- `int myid` : id (rank) of this processor.
- `int seed` : random number seed used in the ordering.
- `int coordFlag` : coordinate flag for local A and B matrices.
  - 1 (`LOCAL`) for local indices, needed for matrix-multiplies.
  - 2 (`GLOBAL`) for global indices, needed for factorizations.
- `InpMtx *A` : matrix object for A
- `InpMtx *B` : matrix object for B
- `Pencil *pencil` : object to hold linear combination of A and B.
- `ETree *frontETree` : object that defines the factorizations, e.g., the number of fronts, the tree they form, the number of internal and external rows for each front, and the map from vertices to the front where it is contained.
- `IVL *symbfacIVL` : object that contains the symbolic factorization of the matrix.
- `SubMtxManager *mtxmanager` : object that manages the `SubMtx` objects that store the factor entries and are used in the solves.

- `FrontMtx *frontmtx` : object that stores the L, D and U factor matrices.
- `IV *oldToNewIV` : object that stores old-to-new permutation vector.
- `IV *newToOldIV` : object that stores new-to-old permutation vector.
- `DenseMtx *Xloc` : dense *local* matrix object that is used during the matrix multiples and solves.
- `DenseMtx *Yloc` : dense *local* matrix object that is used during the matrix multiples and solves.
- `IV *vtxmapIV` : object that maps vertices to owning processors for the factorization and matrix-multiplies.
- `IV *myownedIV` : object that contains a list of all vertices owned by this processor.
- `IV *ownersIV` : object that maps fronts to owning processors for the factorization and matrix-multiplies.
- `IV *rowmapIV` : if pivoting was performed for numerical stability, this object maps rows of the factor to processors.
- `SolveMap *solvemap` : object that maps factor submatrices to owning threads for the solve.
- `MatMulInfo *info` : object that holds all the communication information for a distributed matrix-multiply.
- `int msglvl` : message level for output. When 0, no output, When 1, just statistics and cpu times. When greater than 1, more and more output.
- `FILE *msgFile` : message file for output. When `msglvl` > 0, `msgFile` must not be `NULL`.
- `MPI_Comm comm` : MPI communicator.

4.2 Prototypes and descriptions of `BridgeMPI` methods

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

In contrast to the serial and MT bridge objects, there are seven methods instead of five. In a distributed environment, data structures should be partitioned across processors. On the **SPOOLES** side, the factor entries, and the X and Y matrices that take part in the solves and matrix-multiplies, are partitioned among the processors according to the "front structure" and vertex map of the factor matrices. The **SPOOLES** solve and matrix-multiply bridge methods expect the *local* X and Y matrices. On the **LANCZOS** side, the Krylov blocks and eigenvectors are partitioned across processors in a simple block manner. (The first of p processors has the first n/p rows, etc.)

At the present time, the **SPOOLES** and **LANCZOS** software have no agreement on how the data should be partitioned. (For example, **SPOOLES** could tell **LANCZOS** how it wants the data to be partitioned, or **LANCZOS** could tell **SPOOLES** how it wants the data to be partitioned.) Therefore, inside the **LANCZOS** software a global Krylov block is assembled on each processor prior to calling the solve or matrix-multiply methods. To "translate" between the global blocks to local blocks, and then back to global blocks, we have written two wrapper methods, `JimMatMulMPI()` and `JimSolveMPI()`. Each takes the global input block, compresses it into a local block, call the bridge matrix-multiply or solve method, then takes the local output blocks and gathers them on all the processors into each of their global output blocks. These operations add a considerable cost to the solve and matrix-multiplies, but the next release of the **LANCZOS** software will remove these steps.

1.
```
int SetupMPI ( void *data, int *pprbtype, int *pneqns,
               int *pmxbsz, InpMtx *A, InpMtx *B, int *pseed,
               int *pmsglvl, FILE *msgFile, MPI_Comm comm ) ;
```

   All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `void *data` — a pointer to the `BridgeMPI` object.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - `1` — vibration, a multiply with B is required.
     - `2` — buckling, a multiply with A is required.
     - `3` — simple, no multiply is required.
   - `int *pneqns` — `*pneqns` is the number of equations.
   - `int *pmxbsz` — `*pmxbsz` is an upper bound on the block size.
   - `InpMtx *A` — `A` is a **SPOOLES** object that holds the matrix A.
   - `InpMtx *B` — `B` is a **SPOOLES** object that holds the matrix B. For an ordinary eigenproblem, B is the identity and `B` is `NULL`.
   - `int *pseed` — `*pseed` is a random number seed.
   - `int *pmsglvl` — `*pmsglvl` is a message level for the bridge methods and the **SPOOLES** methods they call.
   - `FILE *pmsglvl` — `msgFile` is the message file for the bridge methods and the **SPOOLES** methods they call.
   - `MPI_Comm comm` — MPI communicator. matrix-multiplies.

   This method must be called in the driver program prior to invoking the eigensolver via a call to `lanczos_run()`. It then follows this sequence of action.

   - The method begins by checking all the input data, and setting the appropriate fields of the `BridgeMPI` object.
   - The `pencil` object is initialized with `A` and `B`.
   - `A` and `B` are converted to storage by rows and vector mode.
   - A `Graph` object is created that contains the sparsity pattern of the union of `A` and `B`.
   - The graph is ordered by first finding a recursive dissection partition, and then evaluating the orderings produced by nested dissection and multisection, and choosing the better of the two. The `frontETree` object is produced and placed into the `bridge` object.
   - Old-to-new and new-to-old permutations are extracted from the front tree and loaded into the `BridgeMPI` object.
   - The vertices in the front tree are permuted, as well as the entries in `A` and `B`. Entries in the lower triangle of `A` and `B` are mapped into the upper triangle, and the storage modes of `A` and `B` are changed to chevrons and vectors, in preparation for the first factorization.
   - The `ownersIV`, `vtxmapIV` and `myownedIV` objects are created, that map fronts and vertices to processors.
   - The entries in `A` and `B` are permuted. Entries in the permuted lower triangle are mapped into the upper triangle. The storage modes of `A` and `B` are changed to chevrons and vectors, and the entries of `A` and `B` are redistributed to the processors that own them.
   - The symbolic factorization is then computed and loaded in the `BridgeMPI` object.
   - A `FrontMtx` object is created to hold the factorization and loaded into the `BridgeMPI` object.

- A `SubMtxManager` object is created to hold the factor's submatrices and loaded into the `BridgeMPI` object.
- The map from factor submatrices to their owning threads is computed and stored in the `solvemap` object.
- The distributed matrix-multiplies are set up.

The `A` and `B` matrices are now in their permuted ordering, i.e., PAP^T and PBP^T, and all data structures are with respect to this ordering. After the Lanczos run completes, any generated eigenvectors must be permuted back into their original ordering using the `oldToNewIV` and `newToOldIV` objects.

Return value:

1	normal return	-7	`*pmxbsz` is invalid
-1	`data` is `NULL`	-8	`A` and `B` are `NULL`
-2	`pprbtype` is `NULL`	-9	`seed` is `NULL`
-3	`*pprbtype` is invalid	-10	`msglvl` is `NULL`
-4	`pneqns` is `NULL`	-11	`msglvl` > 0 and `msgFile` is `NULL`
-5	`*pneqns` is invalid	-12	`comm` is `NULL`
-6	`pmxbsz` is `NULL`		

2. ```
   void FactorMPI ( double *psigma, double *ppvttol, void *data,
                    int *pinertia, int *perror ) ;
   ```

   This method computes the factorization of $A - \sigma B$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `double *psigma` — the shift parameter σ is found in `*psigma`.
   - `double *ppvttol` — the pivot tolerance is found in `*ppvttol`. When $*$`ppvttol` = 0.0, the factorization is computed without pivoting for stability. When $*$`ppvttol` > 0.0, the factorization is computed with pivoting for stability, and all offdiagonal entries have magnitudes bounded above by 1/($*$`ppvttol`).
   - `void *data` — a pointer to the `BridgeMPI` object.
   - `int *pinertia` — on return, `*pinertia` holds the number of negative eigenvalues.
   - `int *perror` — on return, `*perror` holds an error code.

1	error in the factorization	-2	`ppvttol` is `NULL`
0	normal return	-3	`data` is `NULL`
-1	`psigma` is `NULL`	-4	`pinertia` is `NULL`

3. ```
   void JimMatMulMPI ( int *pnrows, int *pncols, double X[], double Y[],
                       int *pprbtype, void *data ) ;
   ```

   This method computes a multiply of the form $Y = IX$, $Y = AX$ or $Y = BX$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of *global* rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of *global* columns in X and Y.
   - `double X[]` — this is the *global* X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the *global* Y matrix, stored column major with leading dimension `*pnrows`.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - `1` — vibration, a multiply with B is required.
     - `2` — buckling, a multiply with A is required.

  - 3 — simple, no multiply is required.
- `void *data` — a pointer to the `BridgeMPI` object.

4.
```
void MatMulMPI ( int *pnrows, int *pncols, double X[], double Y[],
                 int *pprbtype, void *data ) ;
```

   This method computes a multiply of the form $Y = IX$, $Y = AX$ or $Y = BX$. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of *local* rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of *local* columns in X and Y.
   - `double X[]` — this is the *local* X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the *local* Y matrix, stored column major with leading dimension `*pnrows`.
   - `int *pprbtype` — `*pprbtype` holds the problem type.
     - 1 — vibration, a multiply with B is required.
     - 2 — buckling, a multiply with A is required.
     - 3 — simple, no multiply is required.
   - `void *data` — a pointer to the `BridgeMPI` object.

5.
```
void JimSolveMPI ( int *pnrows, int *pncols, double X[], double Y[],
                   void *data, int *perror ) ;
```

   This method solves $(A-\sigma B)X = Y$, where $(A-\sigma B)$ has been factored by a previous call to `Factor()`. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of *global* rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of *global* columns in X and Y.
   - `double X[]` — this is the *global* X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the *global* Y matrix, stored column major with leading dimension `*pnrows`.
   - `void *data` — a pointer to the `BridgeMPI` object.
   - `int *perror` — on return, `*perror` holds an error code.

1	normal return	-3	`X` is `NULL`
-1	`pnrows` is `NULL`	-4	`Y` is `NULL`
-2	`pncols` is `NULL`	-5	`data` is `NULL`

6.
```
void SolveMPI ( int *pnrows, int *pncols, double X[], double Y[],
                void *data, int *perror ) ;
```

   This method solves $(A-\sigma B)X = Y$, where $(A-\sigma B)$ has been factored by a previous call to `Factor()`. All calling sequence parameters are pointers to more easily allow an interface with Fortran.

   - `int *pnrows` — `*pnrows` contains the number of *local* rows in X and Y.
   - `int *pncols` — `*pncols` contains the number of *local* columns in X and Y.
   - `double X[]` — this is the *local* X matrix, stored column major with leading dimension `*pnrows`.
   - `double Y[]` — this is the *local* Y matrix, stored column major with leading dimension `*pnrows`.
   - `void *data` — a pointer to the `BridgeMPI` object.
   - `int *perror` — on return, `*perror` holds an error code.

1	normal return	-3	`X` is `NULL`
-1	`pnrows` is `NULL`	-4	`Y` is `NULL`
-2	`pncols` is `NULL`	-5	`data` is `NULL`

7. `int CleanupMPI ( void *data ) ;`

   This method releases all the storage used by the **SPOOLES** library functions.

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

4.3 The `testMPI` Driver Program

A complete listing of the multithreaded driver program is found in chapter C. The program is invoked by this command sequence.

`testMPI msglvl msgFile parmFile seed inFileA inFileB`

where

- `msglvl` is the message level for the `BridgeMPI` methods and the **SPOOLES** software.
- `msgFile` is the message file for the `BridgeMPI` methods and the **SPOOLES** software.
- `parmFile` is the input file for the parameters of the eigensystem to be solved.
- `seed` is a random number seed used by the **SPOOLES** software.
- `inFileA` is the Harwell-Boeing file for the matrix A.
- `inFileB` is the Harwell-Boeing file for the matrix B.

This program is executed for some sample matrices by the `do_ST_*` shell scripts in the `drivers` directory.

Here is a short description of the steps in the driver program. See Chapter A for the listing.

1. Each processor determines the number of processors and its rank.
2. Each processor decodes the command line inputs.
3. Processor `0` reads the header of the Harwell-Boeing file for A and broadcasts the number of equations to all processors.
4. Each processor reads from the `parmFile` file the parameters that define the eigensystem to be solved.
5. Each processor initializes its Lanczos eigensolver workspace.
6. Each processor fills its Lanczos communication structure with some parameters.
7. Processor `0` reads in the A and possibly B matrices from the Harwell-Boeing files and converts them into `InpMtx` objects from the **SPOOLES** library. The other processors initialize their local `InpMtx` objects.
8. Each processor initializes its linear solver environment via a call to `SetupMPI()`.
9. Each processor invokes the eigensolver via a call to `lanczos_run()`. The `FactorMPI()`, `JimSolveMPI()` and `JimMatMulMPI()` methods are passed to this routine.
10. Processor zero extracts the eigenvalues via a call to `lanczos_eigenvalues()` and prints them out.
11. Processor zero extracts the eigenvectors via a call to `lanczos_eigenvectors()` and prints them out.
12. Each processor free's the eigensolver working storage via a call to `lanczos_free()`.
13. Each processor free's the linear solver working storage via a call to `CleanupMPI()`.

Appendix A

`testSerial.c` — A Serial Driver Program

```
/*  testSerial.c  */

#include "../Bridge.h"

void Factor ( ) ;
void MatMul ( ) ;
void Solve ( ) ;

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

void main ( int argc, char *argv[] )
/*
   ----------------------------------------------------------
   read in Harwell-Boeing matrices, use serial factor, solve,
   and multiply routines based on spooles, invoke eigensolver

   created  -- 98mar31 jcp
   modified -- 98dec18, cca
   ----------------------------------------------------------
*/
{
Bridge    bridge ;
char      *inFileName_A, *inFileName_B, *outFileName,
          *parmFileName, *type ;
char      buffer[20], pbtype[4], which[4] ;
double    lftend, rhtend, center, shfscl, t1, t2 ;
double    c__1 = 1.0, c__4 = 4.0, tolact = 2.309970868130169e-11 ;
double    eigval[1000], sigma[2];
double    *evec;
int       error, fstevl, lfinit, lstevl, mxbksz, msglvl, ncol, ndiscd,
          neig, neigvl, nfound, nnonzeros, nrhs, nrow, prbtyp, rc,
          retc, rfinit, seed, warnng ;
int       c__5 = 5, output = 6 ;
```

```
int       *lanczos_wksp;
InpMtx    *inpmtxA, *inpmtxB ;
FILE      *msgFile, *parmFile;

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

if ( argc != 7 ) {
   fprintf(stdout,
  "\n\n usage : %s msglvl msgFile parmFile seed inFileA inFileB"
  "\n    msglvl   -- message level"
  "\n    msgFile  -- message file"
  "\n    parmFile -- input parameters file"
  "\n    seed     -- random number seed, used for ordering"
  "\n    inFileA -- stiffness matrix in Harwell-Boeing format"
  "\n    inFileB -- mass matrix in Harwell-Boeing format"
  "\n               used for prbtyp = 1 or 2"
  "\n", argv[0]) ;
   return ;
}
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]) ;
   exit(-1) ;
}
parmFileName = argv[3] ;
seed         = atoi(argv[4]) ;
inFileName_A = argv[5] ;
inFileName_B = argv[6] ;
fprintf(msgFile,
        "\n %s "
        "\n msglvl         -- %d"
        "\n msgFile        -- %s"
        "\n parmFile       -- %s"
        "\n seed           -- %d"
        "\n stiffness file -- %s"
        "\n mass file      -- %s"
        "\n",
        argv[0], msglvl, argv[2], parmFileName, seed,
        inFileName_A, inFileName_B) ;
fflush(msgFile) ;
/*
   ---------------------------------------------
   read in the Harwell-Boeing matrix information
   ---------------------------------------------
*/
if ( strcmp(inFileName_A, "none") == 0 ) {
   fprintf(msgFile, "\n no file to read from") ;
```

```
   exit(0) ;
}
MARKTIME(t1) ;
readHB_info (inFileName_A, &nrow, &ncol, &nnonzeros, &type, &nrhs) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in header information for A",
        t2 - t1) ;
/*--------------------------------------------------------------------*/
/*
   ---------------------------------------------------------------
   read in eigenvalue problem data
   neigvl -- # of desired eigenvalues
   which  -- which eigenvalues to compute
     'l' or 'L' lowest (smallest magnitude)
     'h' or 'H' highest (largest magnitude)
     'n' or 'N' nearest to central value
     'c' or 'C' nearest to central value
     'a' or 'A' all eigenvalues in interval
   pbtype -- type of problem
     'v' or 'V' generalized symmetric problem (K,M)
                with M positive semidefinite (vibration problem)
     'b' or 'B' generalized symmetric problem (K,K_s)
                with K positive semidefinite
                with K_s posibly indefinite (buckling problem)
     'o' or 'O' ordinary symmetric eigenproblem
   lfinit -- if true, lftend is restriction on lower bound of
             eigenvalues. if false, no restriction on lower bound
   lftend -- left endpoint of interval
   rfinit -- if true, rhtend is restriction on upper bound of
             eigenvalues.  if false, no restriction on upper bound
   rhtend -- right endpoint of interval
   center -- center of interval
   mxbksz -- upper bound on block size for Lanczos recurrence
   shfscl -- shift scaling parameter, an estimate on the magnitude
             of the smallest nonzero eigenvalues
   ---------------------------------------------------------------
*/
MARKTIME(t1) ;
parmFile = fopen(parmFileName, "r");
fscanf(parmFile, "%d %s %s %d %le %d %le %le %d %le",
       &neigvl, which, pbtype, &lfinit, &lftend,
       &rfinit, &rhtend, &center, &mxbksz, &shfscl) ;
fclose(parmFile);
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in eigenvalue problem data",
        t2 - t1) ;
/*
   ----------------------------------------
   check and set the problem type parameter
   ----------------------------------------
*/
```

```
switch ( pbtype[1] ) {
case 'v' : case 'V' : prbtyp = 1 ; break ;
case 'b' : case 'B' : prbtyp = 2 ; break ;
case 'o' : case 'O' : prbtyp = 3 ; break ;
default :
   fprintf(stderr, "\n invalid problem type %s", pbtype) ;
   exit(-1) ;
}
/*
   ----------------------------
   Initialize Lanczos workspace
   ----------------------------
*/
MARKTIME(t1) ;
lanczos_init_ ( &lanczos_wksp ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : initialize lanczos workspace",
        t2 - t1) ;
/*
   ----------------------------------
   initialize communication structure
   ----------------------------------
*/
MARKTIME(t1) ;
lanczos_set_parm( &lanczos_wksp, "order-of-problem",   &nrow,   &retc );
lanczos_set_parm( &lanczos_wksp, "accuracy-tolerance", &tolact, &retc );
lanczos_set_parm( &lanczos_wksp, "max-block-size",     &mxbksz, &retc );
lanczos_set_parm( &lanczos_wksp, "shift-scale",        &shfscl, &retc );
lanczos_set_parm( &lanczos_wksp, "message_level",      &msglvl, &retc );
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : init lanczos communication structure",
        t2 - t1) ;
/*----------------------------------------------------------------*/
/*
   --------------------------------------------
   create the InpMtx objects for matrix A and B
   --------------------------------------------
*/
if ( strcmp(inFileName_A, "none") == 0 ) {
   fprintf(msgFile, "\n no file to read from") ;
   exit(0) ;
}
MARKTIME(t1) ;
inpmtxA = InpMtx_new() ;
InpMtx_readFromHBfile ( inpmtxA, inFileName_A ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in A", t2 - t1) ;
if ( msglvl > 2 ) {
   fprintf(msgFile, "\n\n InpMtx A object after loading") ;
   InpMtx_writeForHumanEye(inpmtxA, msgFile) ;
   fflush(msgFile) ;
```

```
}
MARKTIME(t1) ;
lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__1, &retc );
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : set A's parameters", t2 - t1) ;
if ( prbtyp != 3 ) {
   if ( strcmp(inFileName_B, "none") == 0 ) {
      fprintf(msgFile, "\n no file to read from") ;
      exit(0) ;
   }
   MARKTIME(t1) ;
   inpmtxB = InpMtx_new() ;
   InpMtx_readFromHBfile ( inpmtxB, inFileName_B ) ;
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : read in B", t2 - t1) ;
} else {
   MARKTIME(t1) ;
   inpmtxB = NULL ;
   lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__4, &retc );
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : set B's parameters", t2 - t1) ;
}
if ( msglvl > 2  && prbtyp != 3 ) {
   fprintf(msgFile, "\n\n InpMtx B object after loading") ;
   InpMtx_writeForHumanEye(inpmtxB, msgFile) ;
   fflush(msgFile) ;
 }
/*
   -----------------------------
   set up the solver environment
   -----------------------------
*/
MARKTIME(t1) ;
rc = Setup((void *) &bridge, &prbtyp, &nrow, &mxbksz, inpmtxA, inpmtxB,
           &seed, &msglvl, msgFile) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : set up solver environment", t2 - t1) ;
if ( rc != 1 ) {
   fprintf(stderr, "\n fatal error %d from Setup()", rc) ;
   exit(-1) ;
}
/*----------------------------------------------------------------------*/
/*
   ----------------------------------------------
   invoke eigensolver
   nfound -- # of eigenvalues found and kept
   ndisc  -- # of additional eigenvalues discarded
   ----------------------------------------------
*/
MARKTIME(t1) ;
lanczos_run(&neigvl, &which[1] , &pbtype[1], &lfinit, &lftend,
```

```
    &rfinit, &rhtend, &center, &lanczos_wksp, &bridge, &nfound,
    &ndiscd, &warnng, &error, Factor, MatMul, Solve ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : time for lanczos run", t2 - t1) ;
/*
   -------------------------
   get eigenvalues and print
   -------------------------
*/
MARKTIME(t1) ;
neig   = nfound + ndiscd ;
lstevl = nfound ;
lanczos_eigenvalues (&lanczos_wksp, eigval, &neig, &retc);
fstevl = 1 ;
if ( nfound == 0 ) fstevl = -1 ;
if ( ndiscd > 0 ) lstevl = -ndiscd ;
hdslp5_ ("computed eigenvalues returned by hdserl",
         &neig, eigval, &output, 39L ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : get and print eigenvalues ", t2 - t1) ;
/*
   -------------------------
   get eigenvectors and print
   -------------------------
*/
/*
MARKTIME(t1) ;
neig = min ( 50, nrow );
Lncz_ALLOCATE(evec, double, nrow, retc);

for ( i = 1 ; i <= nfound ; i++ ) {
   lanczos_eigenvector ( &lanczos_wksp, &i, &i, newToOld,
                         evec, &nrow, &retc ) ;
   hdslp5_ ( "computed eigenvector returned by hdserc",
             &neig, evec, &output, 39L ) ;
}
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : get and print eigenvectors ", t2 - t1) ;
*/
/*
   ------------------------
   free the working storage
   ------------------------
*/
MARKTIME(t1) ;
lanczos_free( &lanczos_wksp ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : free lanczos workspace ", t2 - t1) ;
MARKTIME(t1) ;
rc = Cleanup(&bridge) ;
MARKTIME(t2) ;
```

```
fprintf(msgFile, "\n CPU %8.3f : free solver workspace ", t2 - t1) ;
if ( rc != 1 ) {
   fprintf(stderr, "\n error return %d from Cleanup()", rc) ;
   exit(-1) ;
}
fprintf(msgFile, "\n") ;
fclose(msgFile) ;

return ; }
```

Appendix B

testMT.c — A Multithreaded Driver Program

```
/*  testMT.c  */

#include "../BridgeMT.h"

void FactorMT ( ) ;
void MatMulMT ( ) ;
void SolveMT ( ) ;

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

void main ( int argc, char *argv[] )
/*
   ------------------------------------------------------------------
   read in Harwell-Boeing matrices, using multithreaded factor,
   solve, and multiply routines based on spooles, invoke eigensolver

   created  -- 98mar31, jcp
   modified -- 98dec18, cca
   ------------------------------------------------------------------
*/
{
BridgeMT  bridge ;
char      *inFileName_A, *inFileName_B, *parmFileName, *type ;
char      buffer[20], pbtype[4], which[4] ;
double    lftend, rhtend, center, shfscl, t1, t2 ;
double    c__1 = 1.0, c__4 = 4.0, tolact =  2.309970868130169e-11 ;
double    eigval[1000], sigma[2];
double    *evec;
int       error, fstevl, lfinit, lstevl, msglvl, mxbksz, ncol, ndiscd,
          neig, neigvl, nfound, nnonzeros, nrhs, nrow, nthreads,
          prbtyp, rc, retc, rfinit, seed, warnng ;
int       c__5 = 5, output = 6 ;
int       *lanczos_wksp;
```

```
InpMtx    *inpmtxA, *inpmtxB ;
FILE      *msgFile, *parmFile ;
/*--------------------------------------------------------------------*/
if ( argc != 8 ) {
   fprintf(stdout,
"\n\n usage : %s msglvl msgFile parmFile seed nthread inFileA inFileB"
"\n    msglvl   -- message level"
"\n    msgFile  -- message file"
"\n    parmFile -- input parameters file"
"\n    seed     -- random number seed, used for ordering"
"\n    nthreads -- number of threads "
"\n    inFileA  -- stiffness matrix, in Harwell-Boeing format"
"\n    inFileB  -- mass matrix, in Harwell-Boeing format"
"\n                used for prbtype = 1 or 2"
"\n", argv[0]) ;
   return ;
}
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 able to open file %s\n", argv[0], argv[2]) ;
   exit(-1) ;
}
parmFileName = argv[3] ;
seed         = atoi(argv[4]) ;
nthreads     = atoi(argv[5]) ;
inFileName_A = argv[6] ;
inFileName_B = argv[7] ;
fprintf(msgFile,
        "\n %s "
        "\n msglvl                -- %d"
        "\n message file          -- %s"
        "\n parameter file        -- %s"
        "\n stiffness matrix file -- %s"
        "\n mass matrix file      -- %s"
        "\n random number seed    -- %d"
        "\n number of threads     -- %d"
        "\n",
        argv[0], msglvl, argv[2], parmFileName, inFileName_A,
        inFileName_B, seed, nthreads) ;
fflush(msgFile) ;
/*
   --------------------------------------------
   read in the Harwell-Boeing matrix information
   --------------------------------------------
*/
if ( strcmp(inFileName_A, "none") == 0 ) {
   fprintf(msgFile, "\n no file to read from") ;
   exit(0) ;
```

```
}
MARKTIME(t1) ;
readHB_info (inFileName_A, &nrow, &ncol, &nnonzeros, &type, &nrhs) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in harwell-boeing header info",
        t2 - t1) ;
fflush(msgFile) ;
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------------------------
   read in eigenvalue problem data
   neigvl -- # of desired eigenvalues
   which  -- which eigenvalues to compute
     'l' or 'L' lowest (smallest magnitude)
     'h' or 'H' highest (largest magnitude)
     'n' or 'N' nearest to central value
     'c' or 'C' nearest to central value
     'a' or 'A' all eigenvalues in interval
   pbtype -- type of problem
     'v' or 'V' generalized symmetric problem (K,M)
                with M positive semidefinite (vibration problem)
     'b' or 'B' generalized symmetric problem (K,K_s)
                with K positive semidefinite
                with K_s posibly indefinite (buckling problem)
     'o' or 'O' ordinary symmetric eigenproblem
   lfinit -- if true, lftend is restriction on lower bound of
             eigenvalues. if false, no restriction on lower bound
   lftend -- left endpoint of interval
   rfinit -- if true, rhtend is restriction on upper bound of
             eigenvalues.  if false, no restriction on upper bound
   rhtend -- right endpoint of interval
   center -- center of interval
   mxbksz -- upper bound on block size for Lanczos recurrence
   shfscl -- shift scaling parameter, an estimate on the magnitude
             of the smallest nonzero eigenvalues
   ----------------------------------------------------------------
*/
MARKTIME(t1) ;
parmFile = fopen(parmFileName, "r");
fscanf(parmFile, "%d %s %s %d %le %d %le %le %d %le",
       &neigvl, which, pbtype, &lfinit, &lftend,
       &rfinit, &rhtend, &center, &mxbksz, &shfscl) ;
fclose(parmFile);
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in eigenvalue problem data",
        t2 - t1) ;
fflush(msgFile) ;
/*
   ----------------------------------------
   check and set the problem type parameter
   ----------------------------------------
```

```
*/
switch ( pbtype[1] ) {
case 'v' :
case 'V' :
   prbtyp = 1 ;
   break ;
case 'b' :
case 'B' :
   prbtyp = 2 ;
   break ;
case 'o' :
case 'O' :
   prbtyp = 3 ;
   break ;
default :
   fprintf(stderr, "\n invalid problem type %s", pbtype) ;
   exit(-1) ;
}
/*
   ----------------------------
   Initialize Lanczos workspace
   ----------------------------
*/
MARKTIME(t1) ;
lanczos_init_ ( &lanczos_wksp ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : initialize Lanczos workspace",
        t2 - t1) ;
fflush(msgFile) ;
/*
   ----------------------------------
   initialize communication structure
   ----------------------------------
*/
MARKTIME(t1) ;
lanczos_set_parm( &lanczos_wksp, "order-of-problem", &nrow, &retc );
lanczos_set_parm( &lanczos_wksp, "accuracy-tolerance", &tolact, &retc);
lanczos_set_parm( &lanczos_wksp, "max-block-size", &mxbksz, &retc );
lanczos_set_parm( &lanczos_wksp, "shift-scale", &shfscl, &retc );
lanczos_set_parm( &lanczos_wksp, "message_level", &msglvl, &retc );
lanczos_set_parm( &lanczos_wksp, "number-of-threads", &nthreads, &retc);
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : init lanczos communication structure",
        t2 - t1) ;
/*--------------------------------------------------------------------*/
/*
   -------------------------------------------
   create the InpMtx objects for matrix A and B
   -------------------------------------------
*/
if ( strcmp(inFileName_A, "none") == 0 ) {
```

```
   fprintf(msgFile, "\n no file to read A from") ;
   exit(-1) ;
}
MARKTIME(t1) ;
inpmtxA = InpMtx_new() ;
InpMtx_readFromHBfile ( inpmtxA, inFileName_A ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in A", t2 - t1) ;
fflush(msgFile) ;
if ( msglvl > 2 ) {
  fprintf(msgFile, "\n\n InpMtx A object after loading") ;
  InpMtx_writeForHumanEye(inpmtxA, msgFile) ;
  fflush(msgFile) ;
}
lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__1, &retc );
if ( prbtyp != 3 ) {
   if ( strcmp(inFileName_B, "none") == 0 ) {
      fprintf(msgFile, "\n no file to read from") ;
      exit(0) ;
   }
   MARKTIME(t1) ;
   inpmtxB = InpMtx_new() ;
   InpMtx_readFromHBfile ( inpmtxB, inFileName_B ) ;
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : read in B", t2 - t1) ;
   fflush(msgFile) ;
   if ( msglvl > 2 ) {
      fprintf(msgFile, "\n\n InpMtx B object after loading") ;
      InpMtx_writeForHumanEye(inpmtxB, msgFile) ;
      fflush(msgFile) ;
    }
} else {
  inpmtxB = NULL ;
  lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__4, &retc );
}
/*
   -----------------------------
   set up the solver environment
   -----------------------------
*/
MARKTIME(t1) ;
rc = SetupMT((void *) &bridge, &prbtyp, &nrow, &mxbksz, inpmtxA,
             inpmtxB, &seed, &nthreads, &msglvl, msgFile) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : set up the solver environment",
        t2 - t1) ;
fflush(msgFile) ;
if ( rc != 1 ) {
   fprintf(stderr, "\n error return %d from SetupMT()", rc) ;
   exit(-1) ;
}
```

```
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------
   invoke eigensolver
   nfound -- # of eigenvalues found and kept
   ndisc  -- # of additional eigenvalues discarded
   ----------------------------------------------
*/
MARKTIME(t1) ;
lanczos_run ( &neigvl, &which[1] , &pbtype[1], &lfinit, &lftend,
  &rfinit, &rhtend, &center, &lanczos_wksp, &bridge, &nfound,
  &ndiscd, &warnng, &error, FactorMT, MatMulMT, SolveMT ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : time for lanczos_run", t2 - t1) ;
fflush(msgFile) ;
/*
   -------------------------
   get eigenvalues and print
   -------------------------
*/
MARKTIME(t1) ;
neig   = nfound + ndiscd ;
lstevl = nfound ;
lanczos_eigenvalues (&lanczos_wksp, eigval, &neig, &retc);
fstevl = 1 ;
if ( nfound == 0 ) fstevl = -1 ;
if ( ndiscd > 0 ) lstevl = -ndiscd ;
hdslp5_ ("computed eigenvalues returned by hdserl",
         &neig, eigval, &output, 39L ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : get and print eigenvalues", t2 - t1) ;
fflush(msgFile) ;
/*
   -------------------------
   get eigenvectors and print
   -------------------------
*/
/*
MARKTIME(t1) ;
neig = min ( 50, nrow );
Lncz_ALLOCATE(evec, double, nrow, retc);
for (i = 1; i<= nfound; i++) {d
   lanczos_eigenvector ( &lanczos_wksp, &i, &i, newToOld,
                         evec, &nrow, &retc ) ;
   hdslp5_ ( "computed eigenvector returned by hdserc",
             &neig, evec, &output, 39L ) ;
}
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : get and print eigenvectors", t2 - t1) ;
fflush(msgFile) ;
*/
```

```
/*
   ------------------------
   free the working storage
   ------------------------
*/
MARKTIME(t1) ;
lanczos_free( &lanczos_wksp ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : free lanczos workspace", t2 - t1) ;
fflush(msgFile) ;
MARKTIME(t1) ;
CleanupMT(&bridge) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : free solver workspace", t2 - t1) ;
fflush(msgFile) ;

fprintf(msgFile, "\n") ;
fclose(msgFile) ;

return ; }
```

Appendix C

testMPI.c — A MPI Driver Program

```
/*  testMPI.c  */

#include "../BridgeMPI.h"

void JimMatMulMPI ( ) ;
void JimSolveMPI ( ) ;

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

void main ( int argc, char *argv[] )
/*
   -----------------------------------------------------------------
   MPI environment: read in Harwell-Boeing matrices, using factor,
   solve, and multiply routines based on spooles, invoke eigensolver

   created  -- 98mar31, jcp
   modified -- 98dec18, cca
   -----------------------------------------------------------------
*/
{
BridgeMPI   bridge ;
MPI_Comm    comm ;

char      *inFileName_A, *inFileName_B, *parmFileName, *type ;
char      buffer[20], pbtype[4], which[4] ;
int       error, fstevl, lfinit, lstevl, msglvl, myid, mxbksz, ncol,
          ndiscd, neig, neigvl, nfound, nnonzeros, nproc, nrhs, nrow,
          prbtyp, rc, retc, rfinit, seed, warnng ;
int       c__5 = 5, output = 6 ;
int       *lanczos_wksp ;
InpMtx    *inpmtxA, *inpmtxB ;
FILE      *msgFile, *parmFile ;
double    lftend, rhtend, center, shfscl, t1, t2 ;
double    c__1 = 1.0, c__4 = 4.0, tolact = 2.309970868130169e-11 ;
double    eigval[1000], sigma[2] ;
double    *evec;
```

```
/*
   --------------------------------------------------------------
   find out the identity of this process and the number of process
   --------------------------------------------------------------
*/
MPI_Init(&argc, &argv) ;
MPI_Comm_dup(MPI_COMM_WORLD, &comm) ;
MPI_Comm_rank(comm, &myid) ;
MPI_Comm_size(comm, &nproc) ;
fprintf(stdout, "\n myid = %d", myid) ;
fflush(stdout) ;
/*----------------------------------------------------------------*/
/*
   -----------------------------
   decode the command line input
   -----------------------------
*/
if ( argc != 7 ) {
   fprintf(stdout,
      "\n\n usage : %s msglvl msgFile parmFile seed inFileA inFileB"
      "\n    msglvl   -- message level"
      "\n    msgFile  -- message file"
      "\n    parmFile -- input parameters file"
      "\n    seed     -- random number seed, used for ordering"
      "\n    inFileA  -- stiffness matrix, in Harwell-Boeing format"
      "\n    inFileB  -- mass matrix, in Harwell-Boeing format"
      "\n                used for prbtyp = 1 or 2"
      "\n", argv[0]) ;
   return ;
}
msglvl = atoi(argv[1]) ;
if ( strcmp(argv[2], "stdout") == 0 ) {
   msgFile = stdout ;
} else {
   int    length = strlen(argv[2]) + 1 + 4 ;
   char   *buffer = CVinit(length, '\0') ;
   sprintf(buffer, "%s.%d", argv[2], myid) ;
   if ( (msgFile = fopen(buffer, "w")) == NULL ) {
      fprintf(stderr, "\n fatal error in %s"
              "\n unable to open file %s\n",
              argv[0], buffer) ;
      return ;
   }
   CVfree(buffer) ;
}
parmFileName = argv[3] ;
seed         = atoi(argv[4]) ;
inFileName_A = argv[5] ;
inFileName_B = argv[6] ;
fprintf(msgFile,
        "\n %s "
```

```
        "\n msglvl               -- %d"
        "\n message file         -- %s"
        "\n parameter file       -- %s"
        "\n stiffness matrix file -- %s"
        "\n mass matrix file     -- %s"
        "\n random number seed   -- %d"
        "\n",
        argv[0], msglvl, argv[2], parmFileName, inFileName_A,
        inFileName_B, seed) ;
fflush(msgFile) ;
if ( strcmp(inFileName_A, "none") == 0 ) {
   fprintf(msgFile, "\n no file to read from") ;
   exit(0) ;
}
/*--------------------------------------------------------------------*/
if ( myid == 0 ) {
/*
   ----------------------------------------------
   processor zero reads in the matrix header info
   ----------------------------------------------
*/
   MARKTIME(t1) ;
   readHB_info(inFileName_A, &nrow, &ncol, &nnonzeros, &type, &nrhs) ;
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : read in harwell-boeing header info",
           t2 - t1) ;
   fflush(msgFile) ;
}
MPI_Bcast((void *) &nrow, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------------------------
   read in eigenvalue problem data
   neigvl -- # of desired eigenvalues
   which  -- which eigenvalues to compute
     'l' or 'L' lowest (smallest magnitude)
     'h' or 'H' highest (largest magnitude)
     'n' or 'N' nearest to central value
     'c' or 'C' nearest to central value
     'a' or 'A' all eigenvalues in interval
   pbtype -- type of problem
     'v' or 'V' generalized symmetric problem (K,M)
                with M positive semidefinite (vibration problem)
     'b' or 'B' generalized symmetric problem (K,K_s)
                with K positive semidefinite
                with K_s posibly indefinite (buckling problem)
     'o' or 'O' ordinary symmetric eigenproblem
   lfinit -- if true, lftend is restriction on lower bound of
             eigenvalues. if false, no restriction on lower bound
   lftend -- left endpoint of interval
   rfinit -- if true, rhtend is restriction on upper bound of
```

```
            eigenvalues.  if false, no restriction on upper bound
   rhtend -- right endpoint of interval
   center -- center of interval
   mxbksz -- upper bound on block size for Lanczos recurrence
   shfscl -- shift scaling parameter, an estimate on the magnitude
            of the smallest nonzero eigenvalues
   ---------------------------------------------------------------
*/
MARKTIME(t1) ;
parmFile = fopen(parmFileName, "r");
fscanf(parmFile, "%d %s %s %d %le %d %le %le %d %le",
       &neigvl, which, pbtype, &lfinit, &lftend,
       &rfinit, &rhtend, &center, &mxbksz, &shfscl) ;
fclose(parmFile);
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : read in eigenvalue problem data",
        t2 - t1) ;
fflush(msgFile) ;
/*
   ----------------------------------------
   check and set the problem type parameter
   ----------------------------------------
*/
switch ( pbtype[1] ) {
case 'v' : case 'V' : prbtyp = 1 ; break ;
case 'b' : case 'B' : prbtyp = 2 ; break ;
case 'o' : case 'O' : prbtyp = 3 ; break ;
default :
   fprintf(stderr, "\n invalid problem type %s", pbtype) ;
   exit(-1) ;
}
/*
   ----------------------------
   Initialize Lanczos workspace
   ----------------------------
*/
MARKTIME(t1) ;
lanczos_init_ ( &lanczos_wksp ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : initialize Lanczos workspace",
        t2 - t1) ;
fflush(msgFile) ;
/*
   ----------------------------------
   initialize communication structure
   ----------------------------------
*/
MARKTIME(t1) ;
lanczos_set_parm( &lanczos_wksp, "order-of-problem", &nrow, &retc );
lanczos_set_parm( &lanczos_wksp, "accuracy-tolerance", &tolact, &retc );
lanczos_set_parm( &lanczos_wksp, "max-block-size", &mxbksz, &retc );
```

```
lanczos_set_parm( &lanczos_wksp, "shift-scale", &shfscl, &retc );
lanczos_set_parm( &lanczos_wksp, "message_level", &msglvl, &retc );
lanczos_set_parm( &lanczos_wksp, "mpi-communicator", &comm, &retc );
lanczos_set_parm( &lanczos_wksp, "qfile-pathname", "lqfil", &retc );
lanczos_set_parm( &lanczos_wksp, "mqfil-pathname", "lmqfil", &retc );
lanczos_set_parm( &lanczos_wksp, "evfil-pathname", "evcfil", &retc );
MARKTIME(t2) ;
fprintf(msgFile,
        "\n CPU %8.3f : init the lanczos communication structure",
        t2 - t1) ;
fflush(msgFile) ;
/*----------------------------------------------------------------*/
if ( myid == 0 ) {
/*
   ------------------------------------
   processor zero reads in the matrices
   ------------------------------------
*/
   MARKTIME(t1) ;
   inpmtxA = InpMtx_new() ;
   InpMtx_readFromHBfile ( inpmtxA, inFileName_A ) ;
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : read in first matrix", t2 - t1) ;
   fflush(msgFile) ;
   if ( msglvl > 2 ) {
      fprintf(msgFile, "\n\n InpMtx A object after loading") ;
      InpMtx_writeForHumanEye(inpmtxA, msgFile) ;
      fflush(msgFile) ;
   }
   lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__1, &retc );
   if ( prbtyp != 3 ) {
      if ( strcmp(inFileName_B, "none") == 0 ) {
         fprintf(msgFile, "\n no file to read from") ;
         exit(0) ;
      }
      MARKTIME(t1) ;
      inpmtxB = InpMtx_new() ;
      InpMtx_readFromHBfile ( inpmtxB, inFileName_B ) ;
      MARKTIME(t2) ;
      fprintf(msgFile, "\n CPU %8.3f : read in first matrix", t2 - t1) ;
      fflush(msgFile) ;
      if ( msglvl > 2 ) {
         fprintf(msgFile, "\n\n InpMtx B object after loading") ;
         InpMtx_writeForHumanEye(inpmtxB, msgFile) ;
         fflush(msgFile) ;
      }
   } else {
      inpmtxB = NULL ;
      lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__4, &retc );
   }
} else {
```

```
/*
   -----------------------------------------------
   other processors initialize their local matrices
   -----------------------------------------------
*/
   inpmtxA = InpMtx_new() ;
   InpMtx_init(inpmtxA, INPMTX_BY_CHEVRONS, SPOOLES_REAL, 0, 0) ;
   lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__1, &retc );
   if ( prbtyp == 1 || prbtyp == 2 ) {
      inpmtxB = InpMtx_new() ;
      InpMtx_init(inpmtxB, INPMTX_BY_CHEVRONS, SPOOLES_REAL, 0, 0) ;
   } else {
      inpmtxB = NULL ;
      lanczos_set_parm( &lanczos_wksp, "matrix-type", &c__4, &retc );
   }
}
/*
   -----------------------------
   set up the solver environment
   -----------------------------
*/
MARKTIME(t1) ;
rc = SetupMPI((void *) &bridge, &prbtyp, &nrow, &mxbksz, inpmtxA,
              inpmtxB, &seed, &msglvl, msgFile, MPI_COMM_WORLD) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : set up solver environment", t2 - t1) ;
fflush(msgFile) ;
if ( rc != 1 ) {
   fprintf(stderr, "\n fatal error return %d from SetupMPI()", rc) ;
   MPI_Finalize() ;
   exit(-1) ;
}
/*--------------------------------------------------------------------*/
/*
   ----------------------------------------------
   invoke eigensolver
   nfound -- # of eigenvalues found and kept
   ndisc  -- # of additional eigenvalues discarded
   ----------------------------------------------
*/
MARKTIME(t1) ;
lanczos_run ( &neigvl, &which[1] , &pbtype[1], &lfinit, &lftend,
  &rfinit, &rhtend, &center, &lanczos_wksp, &bridge, &nfound,
  &ndiscd, &warnng, &error, FactorMPI, JimMatMulMPI,
  JimSolveMPI ) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : time for lanczos run", t2 - t1) ;
fflush(msgFile) ;
if ( myid == 0 ) {
/*
   ---------------------------------------------
```

```
   processor 0 deals with eigenvalues and vectors
   ----------------------------------------------
*/
   MARKTIME(t1) ;
   neig   = nfound + ndiscd ;
   lstevl = nfound ;
   lanczos_eigenvalues (&lanczos_wksp, eigval, &neig, &retc);
   fstevl = 1 ;
   if ( nfound == 0 ) fstevl = -1 ;
   if ( ndiscd > 0 ) lstevl = -ndiscd ;
   hdslp5_ ("computed eigenvalues returned by hdserl",
            &neig, eigval, &output, 39L ) ;
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : get and print eigenvalues",
           t2 - t1) ;
   fflush(msgFile) ;
/*
   -------------------------
   get eigenvectors and print
   -------------------------
*/
/*
   MARKTIME(t1) ;
   neig = min ( 50, nrow );
   Lncz_ALLOCATE(evec, double, nrow, retc);
   for (i = 1; i<= nfound; i++) {
      lanczos_eigenvector(&lanczos_wksp, &i, &i, newToOld,
                          evec, &nrow, &retc) ;
      hdslp5_("computed eigenvector returned by hdserc",
              &neig, evec, &output, 39L ) ;
   }
   MARKTIME(t2) ;
   fprintf(msgFile, "\n CPU %8.3f : get and print eigenvectors",
           t2 - t1) ;
   fflush(msgFile) ;
*/
}
/*
   ------------------------
   free the working storage
   ------------------------
*/
MARKTIME(t1) ;
lanczos_free(&lanczos_wksp) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : free lanczos workspace", t2 - t1) ;
fflush(msgFile) ;
MARKTIME(t1) ;
CleanupMPI(&bridge) ;
MARKTIME(t2) ;
fprintf(msgFile, "\n CPU %8.3f : free solver workspace", t2 - t1) ;
```

```
fflush(msgFile) ;

MPI_Finalize() ;

fprintf(msgFile, "\n") ;
fclose(msgFile) ;

return ; }
```

Index