SPOOLES 2.2 Installation Manual

Cleve Ashcraft, Boeing Shared Services Group*

January 25, 1999

1 Overview

The **SPOOLES** library is *object-oriented*. Just about everything is an object, data structures and algorithms. The directory structure reflects this design philosophy. First, make a directory (say called `spooles`) and place the tar file `spooles.2.2.tar.gz` in this directory. unzip the file and then extract the files. For example, here are the Unix commands.

```
% mkdir spooles
% mv spooles.2.2.tar.gz spooles
% cd spooles
% gzip -d spooles.2.2.tar.gz
% tar -xvf spooles.2.2.tar.gz
```

The top level directory has many subdirectories and many header files. Most subdirectories deal with a single object.

Let's look at the first object, the `A2` dense matrix object. (`A2` stands for `A`rray, `2`-dimensional.) The `A2` directory has two files and three subdirectories.

- `A2.h` — This is the header file that defines the `struct` that holds the `A2` object and has prototypes of all of the `A2` methods. If you don't have printed documentation in front of you, this is a handy place to look. This file contains comments from the source code telling what the method does and describes the calling sequence parameters.
- `makefile` — This makefile is called by the top level makefile to compile source, drivers, and clean up the subdirectories.
- `doc/` — This subdirectory contains LaTeX files that document the `A2` object. These files also form a chapter of the **SPOOLES** Reference Manual.
- `src/` — This subdirectory contains all the source code for the `A2` object.
- `drivers/` — This subdirectory contains all the driver programs that exercise and validate the behavior of the `A2` object.

Each object's directory contains a header file, a make file, and a source and document subdirectories. Most but not all contain driver directories.

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

SPOOLES is written in the C language, which does not directly support object-oriented programming. There is no inheritance in C, and so there is no nesting of our object directories. For example, there are three tree objects — `Tree` for a simple tree, `ETree` for a front tree (which contains a `Tree` object and so could be descended from it), and `DSTree` for a "domain/separator" tree (which also contains a `Tree` object) — but they are separate objects whose directories are at the same level.

There are some directories that are peripheral to the main library.

- The `LinSol` directory contains some "bridge" or "wrapper" methods used to incorporate **SPOOLES** into CSAR-Nastran, a finite element program, to solve linear systems.
- The `Eigen` directory contains some "bridge" or "wrapper" methods used to incorporate **SPOOLES** into a block-shifted Lanczos eigensolver.
- The `FanBothMap` directory is an experimental object used to investigate the feasibility of replacing the fan-in method with the fan-both method for a distributed factorization.
- The `ReferenceManual` directory contains LaTeX files to construct the reference manual (currently 400+ pages).
- The `UserManual` directory contains LaTeX files to construct the various user manuals. (This document is one.)
- The `Matrices` directory contains some matrix files that can be used to run the example programs.

The goal is to get work done. The **SPOOLES** library can be used in several ways. Here are three scenarios.

- I want a global library to link with other application programs.
- I want to exercise the objects in the library to get a feel for performance, e.g., test out the matrix orderings, or the factors and solves, or the multithreaded and MPI programs.
- I want to do development work, modify existing objects or create new objects.

We will discuss each in turn. But first, let's talk about makefiles.

2 Makefiles

Each object's `src/` and (possibly) `drivers/` directories have their own makefile. Each of these makefiles "includes" (via an include statement) the file `Make.inc` from the top level directory.[1] This is the file where one sets the following parameters.

- `CC` is your favorite C compiler.
- `OPTLEVEL` is the compiler's optimization level.
- `CFLAGS` is the compiler's compilation flags, which normally includes the `OPTLEVEL` data.
- `LDFLAGS` are load flags for executables.
- `THREAD_LIBS` are system dependent libraries for multithreaded programs.

[1] Many thanks go to Clay Breshears from CEWES, `clay@turing.wes.hpc.mil`, for the prototype `Make.inc` file.

- PURIFY and PURIFY_GCC_VERSION deal with the Purify program from PureAtria. This is a handy program that detects memory leaks, overwriting of storage, and the like. Most of the code at one point or another has been run through Purify.
- AR and ARFLAGS are the loader and its flags.
- RANLIB is used when the library must be postprocessed to get a symbol table. (Some systems still use ranlib.) If ranlib is not needed (as on most systems), RANLIB is set to echo to echo out the library name during the make process.
- The .c.o and .c.a suffix rules are defined.
- MPI_INSTALL_DIR is the installation directory for MPI.
- MPI_LIB_PATH is the library path for MPI.
- MPI_LIBS are the libraries needed for MPI programs.
- MPI_INCLUDE_DIR is the include directory for MPI.

The Make.inc file contains several different values for each of the parameters along with some comments about which lines are for which system. We have found most of this to be fairly straightforward, but getting the thread libraries and MPI libraries correct may take a bit of doing. Contact your local system administrator for help.

3 Setting the thread type

Once the Make.inc file is modified for your system, it's time to think about what functionality you want to use. The **SPOOLES** library operates in serial, multithreaded and MPI environments. The code for these three environments is fairly segregated. The MPI directory contains all source and driver code for MPI programs. The MT directory contains all source and driver code for multithreaded programs. All other directories contain serial code.[2] The MPI source code is compiled into a spoolesMPI.a library. The multithreaded source code is compiled into a spoolesMT.a library. The serial code is compiled into a spooles.a library.

If you are going to operate only in serial mode, then you should edit one file, Lock/Lock.h. Here is where we hide the operating system specific details about *mutual exclusion* locks. Presently only POSIX threads are actively supported by the library. The original development work was done using Solaris threads, and there is some Solaris thread code still in the Lock and MT src/ directories. (The library has also been ported to use the NeXT and WindowsNT thread packages, but this code is not included in this release.) The decision to use threads or not, or which thread library to use, requires the Lock/Lock.h file to be modified. Line 9 defines the thread type. It is presently set to POSIX threads.

```
#define THREAD_TYPE TT_POSIX
```

To use no threads, one would replace this line with the one below.

```
#define THREAD_TYPE TT_NONE
```

TT_POSIX and TT_NONE are defined in the Lock/Lock.h file. There is also a line to define Solaris threads. If you are planning to port the **SPOOLES** library to another thread library, add a THREAD_TYPE define statement and modify the definition of the lock in the Lock structure. The files in Lock/src and MT/src will also have to be modified to use different subroutine calls to create and join threads, etc.

[2] This is not quite true. The LinSol directory contains serial, MT and MPI subdirectories, but the LinSol package is *adjunct* to the **SPOOLES** library, not part of it.

4 Building global libraries

To build the global serial library `spooles.a`, type the following.

```
% cd spooles
% make lib
```

This visits each of the serial objects' `src/` directories, compiles the source code and loads it into the `spooles.a` library. Note, this option requires the Perl language installed on the system, for we use a Perl script to generate a temporary makefile for each object. If your computer system does *not* have Perl installed, this option will have the same result.

```
% cd spooles
% make global
```

This approach uses hard-coded makefile inside each `src/` directory, and is less reliable than the first way.

The result of either of these two commands is a `spooles.a` library that is present in the top level `spooles` directory. Recall, this only contains the serial code. This process may take a long time, for there are about 140,000 lines of source code in the library.

To build a multithreaded library, one must go to the `MT/src` directory. Now we have two choices — to build a separate `spoolesMT.a` library, or two merge the multithreaded code into the `spooles.a` library. We recommend the first, via typing `make spoolesMT.a`, though the second is perfectly fine by typing `make makeLib`. If Perl is not installed, type `make -f makeGlobalLib` for the latter behavior.

Much the same applies to the MPI library. To build a MPI library, one must go to the `MPI/src` directory. Now we have two choices — to build a separate `spoolesMPI.a` library, or two merge the MPI code into the `spooles.a` library. We recommend the first, via typing `make spoolesMPI.a`, though the second is perfectly fine by typing `make makeLib`. If Perl is not installed, type `make -f makeGlobalLib` for the latter behavior.

Note, in the top level `spooles/makefile` there are two lines commented out under the `lib:` target which will compile the multithreaded and MPI source code into `spooles.a`. If you uncomment these lines, (replace the `#` with a tab), and type `make lib`, then the multithreaded and MPI source will be included in `spooles.a`.

5 Exercising the objects' driver programs

Each object has a `src/` subdirectory that contains source code, and a `doc` subdirectory that contains LaTeX files for documentation. Most objects also have a `drivers/` subdirectory that contain driver programs that exercise and verify the objects' behavior.

Once the global `spooles.a` library is built, one can compile, link and load the driver programs. This can be done inside the `drivers/` subdirectory of each individual object, or all the executables for the driver programs can be created by typing `make all_drivers` in the top level directory.

Let us look at one particular example. The `Chv` object is used during the sparse factorization to store a "front", a submatrix of the sparse matrix that is worked on at one time. The most expensive part of the factorization is the dense matrix-matrix operations that are done to a front. The `Chv/drivers/test_update.c` program exercises this functionality. To create the `test_update` executable, type `make test_update` while in the `Chv/drivers/` directory. To execute this program, there is a `do_update` shell script. Here we can test the accuracy and performance of this particular computation.

The shell script has a number of parameters — real or complex entries, symmetric, Hermitian or nonsymmetric matrices, sparse or dense fronts, sizes of the updating and updated fronts, etc. The message level parameter `msglvl` governs the amount of output sent to the message file. When `msglvl` is `1`, only timing information is written. When `msglvl` is `2` or greater, the program dumps output in a form that is readable by Matlab, (and, with some tweaking, by the GNU Octave program that is very similar and free.) This is our usual practice to validate the behavior of small parts of the computations before we link them into larger pieces.

While the source code is very well documented, and the driver programs are fairly well documented, the shell scripts that exercise the driver programs are rarely documented. Consult the source of each driver program to understand their accompanying shell script.

If you want to get a feel for the speed of the **SPOOLES** library on your system, see the `test_update` program in the `Chv/drivers/` directory and the `test_solveupd*` programs in the `SubMtx/drivers/` directory. These are the programs that exercise the BLAS3 computations used during the factors and solves. Remember to set `msglvl` to `1` to get just timings. To test out the numeric factorizations and solves, see the `testGrid` program. in the `FrontMtx/drivers` directory. There are similar programs in the `MT/drivers` and `MPI/drivers` directories. Many shell scripts, particularly those that deal with the ordering and factorization objects, attempt to read in matrix and graph files you will not find on your systems.[3] There is a small `Matrices` directory that contains test matrices, which can be used in some of the shell scripts, once the files are un-zip'ed.

6 Modifying source code and new development

One of the strengths of the **SPOOLES** library is its open design. Objects manage much of the complexity of 140,000 lines of source code. There is a great deal of compartmentalization or encapsulation of data structures that make it relatively easy to replace modules or extend the functionality.

To make a new object, simply copy over the directory structure of a similar object, and then modify the header, source, documentation and drivers files as necessary. Let us call this new object `NewObj`. When in `NewObj/src`, typing `make updateLib` or `make -f makeGlobalLib` will load the new source files into the global `spooles.a` library. Or one can make a local library by typing `make NewObj.a`. In the `NewObj/drivers` directory, modify `makefile` as necessary. If the `NewObj` source code has been loaded into `spooles.a`, then the line

```
LIBS = ../../spooles.a -lm
```

will be fine. (Of course, if `NewObj` has multithreaded or MPI code, see the `makefile`'s in the `MT/drivers` or `MPI/drivers` directories.) If instead you have created the `NewObj/src/NewObj.a` library, modify the library line to

```
LIBS = ../src/NewObj.a ../../spooles.a -lm
```

so the new source code can be linked. As changes are made to the `NewObj` source files, they need to be recompiled via `make updateLib` (for the global `spooles.a` library) or via `make NewObj.a` (for the local `NewObj.a` library).

Any driver programs will have to be re-linked after changes to the `NewObj` source code, whether it lies in the global or local library. There is currently no connection in the `driver/makefile` between the drivers and the accompanying source code. That can be easily added by the user.

The flat directory structure and the relative independence of the objects is not without its faults. It makes keeping libraries current in an efficient matter difficult to do. We now explain some of the details that hopefully will save time and confusion later. We focus on the situation where some change has been made to an objects source file(s), and the new source must be loaded into the global `spooles.a` library. There are three choices.

[3] What you are seeing is a snapshot of my development environment. You should be able to customize your environment.

- If your system does not have Perl, you must call `make -f makeGlobalLib` inside the `src` directory. The `makeGlobalLib` compiles and loads *all* source files in the `src` directory, at least that is what it should do. The `makeGlobalLib` file contains a list of source files, which **should** be the same as in the `makefile` file. It is the user's responsibility to keep these two files current — add or remove a file to/from one, add or remove it to/from the other. This approach can be quite cumbersome if only one file in the `src` directory has been modified, for all the source files will be compiled. Alternatively, you can edit the `SRC =` line (or preferably, add a second line below the first) to reflect only those files that need loading.
- If your system does have Perl, you have more flexibility.
  - To compile all source files into the `spooles.a` library, type `make makeLib` while in the `spooles` directory. Execution proceeds in each of the objects' `src/` directories in turn. The Perl script `makeLib` found in the top level `spooles` directory is executed, which scans the `makefile` for all source code names, compiles them and loads them into the library.
  - Inside an object's `src/` directory, you can compile the newly modified source files into the `spooles.a` library by typing `make updateLib`. The Perl script `updLib` found in the top level `spooles` directory is executed, which scans the `makefile` for *all* source code names, compares their modification time against the `spooles.a` library, compiles the newer ones and loads them into the library. This last step can be somewhat tricky, at least when changes are being made to source files in two or more objects.

    Consider the case where we are running the `testGrid` program in the `FrontMtx/drivers` directory, and we have made some modifications to some source files for the `Chv` and `SubMtx` objects. We change directories to `Chv/src` and type `make updateLib`, and the modified files are compiled and loaded into `spooles.a`. We then change directories to `SubMtx/src` and type `make updateLib`, but nothing happens, our newly modified files are not compiled nor loaded into `spooles.a`. The problem is that the `spooles.a` directory was modified when the new `Chv/src` files were compiled and loaded, and so is newer than the modified `SubMtx` files.

    The solution is to modify files in the `Chv/src` directory, type `make updateLib`, then modify the files in the `SubMtx/src` directory, again type `make updateLib`, and then type `make testgrid` in the `FrontMtx/drivers` directory.

7 Cleaning up and packaging

Let's say we've just created the `spooles.a` library and then typed `make all_drivers` when in the `spooles` directory. All the executable programs are now on disk, and taking up a lot of space. To remove all the executable programs, we don't need to go into each object's `driver` program and remove them one by one. We can do this from the top level directory by typing `make clean`.

This process visits each of the objects' `src/`, `doc/` and `drivers/` directories one by one. In the `src/` directories, any `*.o` and `*.a` files are removed. In the `doc/` directories, any `*.dvi` files are removed. In the `drivers/` directories, any `*.o` and `*.a` files are removed, as well as all driver program executables. (The `makefile` in a drivers directory keeps track of the driver programs in the `DRIVERS` variable. Modify this as necessary if you add a driver program to the directory.)

Let's say you've got **SPOOLES** working on one system and you want to move it to another system. Even after cleanup (via the `make clean` call) there may be a large number of files that don't need to be moved. Or perhaps you'd like to `tar` everything that is necessary without removing your `spooles.a` library and executables that took so long to compile.

The `dotar` shell script is used to package up everything that is necessary into one tar file. You should modify this as necessary. Here are the lines from `dotar` that archive what is necessary from the `A2` object.

```
A2.h \
   A2/{*.h,makefile} \
   A2/src/{makefile,makeGlobalLib,*.c} \
   A2/drivers/{do*,makefile,*.c} \
   A2/doc \
```

This `tar`'s the header file and middle level makefile, then goes into the `src/` directory and `tar`'s the makefiles and source code, then goes into the `drivers/` directory and `tar`'s the makefiles, driver programs and shell scripts, and then `tar`'s everything in the `doc/` directory. This happens for every object, with a few minor exceptions.

8 Afterword

Any questions, comments, and particularly, suggestions, contact Cleve Ashcraft, `cleve.ashcraft@boeing.com`.