**Exercise: Calling LAPACK**

In this exercise, we’ll use the same conventions and commands as in the batch computing exercise. You should refer back to the batch computing exercise description for details on various Unix commands.

You’ll be editing a simple code that will need to use a certain LAPACK routine.

Here are the steps for this exercise:

1. Log in to the Linux cluster supercomputer (sooner.oscer.ou.edu).
2. Conﬁrm that you’re in your home directory:

**pwd**

/home/yourusername

1. Check that you have a NCSIPARII2011\_exercises subdirectory inside your home directory:

**ls**

NCSIPARII2011\_exercises

1. Copy the LAPACK directory into your NCSIIPARII2011\_exercises directory:

**cp -r ~hneeman/NCSIPARII2011\_exercises/LAPACK/ ~/NCSIPARII2011\_exercises/**

1. Go into your NCSIPARII2011\_exercises subdirectory:

**cd NCSIPARII2011\_exercises**

1. Conﬁrm that you’re in your NCSIPARII2011\_exercises subdirectory:

**pwd**

/home/yourusername/NCSIPARII2011\_exercises

1. See what files or subdirectories (if any) are in the current working directory:

**ls**

1. Go into your LAPACK subdirectory:

**cd LAPACK**

1. Conﬁrm that you’re in your LAPACK subdirectory:

**pwd**

/home/yourusername/NCSIPARII2011\_exercises/LAPACK

1. See what files or subdirectories (if any) are in the current working directory:

**ls**

1. Choose which language you want to use (C or Fortran90), and cd into the appropriate directory:

**cd C**

OR:

**cd Fortran90**

1. Conﬁrm that you’re in your C or Fortran90 subdirectory:

**pwd**

/home/yourusername/NCSIPARII2011\_exercises/LAPACK/C

1. See what files or subdirectories (if any) are in the current working directory:

**ls**

1. Edit the batch script lapack\_test.bsub to use your username and e-mail address.
2. If you haven’t already examined lapack\_test.c (or lapack\_test.f90), do so now.
3. Edit the source file with your favorite text editor (for example, nano, vi, emacs etc):

**nano lapack\_test.c**

OR:

**nano lapack\_test.f90**

1. Look for the line in the source file that says:

/\* Call to LAPACK routine goes here. \*/

OR

!... Call to LAPACK routine goes here.

This is where you will write a call to the appropriate LAPACK routine.

1. To find the appropriate LAPACK routine:
   1. Open your favorite web browser (for example, Internet Explorer, Firefox, Safari).
   2. Go to:

**http://www.netlib.org/lapack/**

* 1. When that webpage loads, find:

single precision real

and click on that link.

* 1. When that webpage loads, in the webpage find:

solve

In particular, you’re looking for the solver for a general system of linear equations.

1. When you find that solver, click on the link for its Fortran source code – the file whose extension is .f (dot f). (Ignore the link for the source file plus its dependencies.)
2. Using the Fortran subroutine header

SUBROUTINE subroutinename (argument1, argument2, ...)

and the comments immediately below the header that describe the subroutine’s arguments, determine how to call that subroutine from either lapack\_test.c or lapack\_test.f90.

1. Modify either lapack\_test.c or lapack\_test.f90 to call that LAPACK routine.

**NOTES**

* 1. In this exercise, we’re intentionally using statically allocated arrays. (If you don’t know what that means, then don’t worry about it.)
  2. If you’re calling the subroutine from lapack\_test.c, bear in mind the following:

In Fortran (all versions), **ALL** arguments are *pass-by-reference*, meaning that the routine that you call is using the same memory location for each argument as the calling routine.

For example:

SUBROUTINE caller ()

IMPLICIT NONE

INTEGER :: actual\_argument !! local variable inside caller

actual\_argument = 5

PRINT \*, “caller: before calling, actual\_argument = “, actual\_argument

CALL callee(actual\_argument)

PRINT \*, “caller: after calling, actual\_argument = “, actual\_argument

END SUBROUTINE caller

SUBROUTINE callee (formal\_argument)

IMPLICIT NONE

INTEGER :: formal\_argument !! argument passed to callee from caller

PRINT \*, “callee: before, formal\_argument = “, formal\_argument

formal\_argument = formal\_argument + 1

PRINT \*, “callee: after, formal\_argument = “, formal\_argument

END SUBROUTINE callee

The output of this code fragment is:

caller: before calling, actual\_argument = 5

callee: before, formal\_argument = 5

callee: after, formal\_argument = 6

caller: after calling, actual\_argument = 6 **🡨 NOTICE!!!**

In C, **ALL** arguments are *pass-by-copy*, meaning that the routine that you call is using a different memory location containing the same value for each argument as the calling routine – except for arrays, which are always passed by reference (don’t ask).

void caller ()

{

int actual\_argument; /\* local variable inside caller \*/

actual\_argument = 5;

printf(“caller: before calling, actual\_argument = %d\n“, actual\_argument);

callee(actual\_argument);

printf(“caller: after calling, actual\_argument = %d\n“, actual\_argument);

}

void callee (int formal\_argument)

{

printf(“callee: before, formal\_argument = %d\n“, formal\_argument);

formal\_argument = formal\_argument + 1;

printf(“callee: after, formal\_argument = %d\n“, formal\_argument);

}

The output of this code fragment is:

caller: before calling, actual\_argument = 5

callee: before, formal\_argument = 5

callee: after, formal\_argument = 6

caller: after calling, actual\_argument = 5 **🡨 NOTICE!!!**

Why is this relevant?

As you can see from the LAPACK routine source file, LAPACK is written in Fortran. So if you’re working in Fortran, you’re fine, and you don’t have to worry about this.

But if you’re working in C, then when you call the LAPACK Fortran routine, for your scalar (non-array) arguments, you have to accomplish pass-by-reference via fakery, by passing a pointer to (that is, the address of) each scalar argument.

For example:

void caller2 ()

{

int actual\_argument; /\* local variable inside caller \*/

actual\_argument = 5;

printf(“caller2: before calling, actual\_argument = %d\n“, actual\_argument);

callee2(**&**actual\_argument);

printf(“caller2: after calling, actual\_argument = %d\n“, actual\_argument);

}

void callee2 (int**\*** formal\_argument)

{

printf(“callee2: before, **\***formal\_argument = %d\n“, **\***formal\_argument);

**\***formal\_argument = **\***formal\_argument + 1;

printf(“callee2: after, **\***formal\_argument = %d\n“, **\***formal\_argument);

}

The output of this code fragment is:

caller2: before calling, actual\_argument = 5

callee2: before, formal\_argument = 5

callee2: after, formal\_argument = 6

caller2: after calling, actual\_argument = 6 **🡨 NOTICE!!!**

On the other hand:

* **YOU DON’T HAVE TO DO THIS FOR ARRAY ARGUMENTS;**
* **YOU DON’T HAVE TO WORRY ABOUT THIS IN FORTRAN**.
  1. The LAPACK implementation you’ll be linking to is part of Intel’s Math Kernel Library (MKL), rather than a hand-compiled version of an implementation from Netlib or a similar open source provider. We have no specific bias in favor of or against MKL, but we don’t have time to build the relevant libraries from source code.

1. Save the source file and close your favorite text editor.
2. Compile using the make command:

**make**

1. If the program doesn’t compile, edit the source file and try again. Continue to do this until the compilation succeeds.
2. Submit the batch script file lapack\_test.bsub to the batch scheduler:

**bsub < lapack\_test.bsub**

**NOTICE** the less than symbol < which is **EXTREMELY IMPORTANT**.

You should get back output something like this:

Job <######> is submitted to queue <parii\_q>.

where ###### is replaced by the batch job ID for the batch job that you’ve just submitted.

1. Check the status of your batch job:

**bjobs**

You’ll get one of the following outputs, either:

No unfinished job found

(if you get this right after the bjobs command, try it several more times, because sometimes there’s a pause just before the batch job starts showing up, as shown below):

JOBID USER STAT QUEUE FROM\_HOST EXEC\_HOST JOB\_NAME SUBMIT\_TIME

4081250 yourusername PEND parii\_q sooner1 lapack\_test Oct 17 14:58

where ###### is replaced by a batch job ID number, and yourusername is replaced by your user name, and where PEND is short for “pending,” meaning that your job is waiting to start, OR:

JOBID USER STAT QUEUE FROM\_HOST EXEC\_HOST JOB\_NAME SUBMIT\_TIME

4081250 yourusername RUN parii\_q sooner1 c127 lapack\_test Oct 17 14:58

1. You may need to check the status of your batch job repeatedly, using the bjobs command, until it runs to completion. **This may take several minutes (occasionally much longer).**

You’ll know that the batch job has finished when it no longer appears in the list of your batch jobs:

No unfinished job found

1. Once your job has finished running, ﬁnd the *standard output* and *standard error* files from your job:

**ls -ltr**

Using this command, you should see files named

lapack\_test\_######\_stdout.txt

and

lapack\_test\_######\_stderr.txt

(where ###### is replaced by the batch job ID).

These files should contain the output of lapack\_test. Ideally, the stderr file should have length zero.

1. Look at the contents of the standard output file:

**cat lapack\_test\_######\_stdout.txt**

(where ###### is replaced by the batch job ID). You may also want to look at the stderr file:

**cat lapack\_test\_######\_stdout.txt**

1. If this run had **ANY** problems, then send e-mail to:

[support@oscer.ou.edu](mailto:support@oscer.ou.edu)

which reaches all OSCER staff (including Henry), and attach the following files:

makefile

lapack\_test.c

lapack\_test.bsub

lapack\_test\_######\_stdout.txt

lapack\_test\_######\_stderr.txt