Math3101/Math5303
9. Software Libraries

A/Prof William McLean,
School of Maths and Stats, UNSW
Session 1, 2017

Version: May 2, 2017

Introduction

Previously, we saw how separate compilation facilitates use of the same procedures by different programs. A library is essentially a container of object files that in turn contain compiled procedures. In this lesson, we will to create and use a simple library, and how libraries enable Julia to call procedures written in Fortran or C.

BLAS

Examples using a non-system library

Examples using a system library

Calling a library routine from Julia

BLAS

In 1979, C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh published a paper in the *ACM Transactions on Mathematical Software* describing a Fortran 77 library called BLAS — the Basic Linear Algebra Subroutines. The library provided a collection of low-level operations that could be used as building blocks for more complex algorithms in linear algebra.

The importance of the BLAS stems from the fact that it defines a standard API (Application Programmer Interface). In addition to a basic reference implementation, several high-performance implementations have been developed, optimised for specific hardware. By using the BLAS, one linear algebra code can run efficiently on a variety of different computer architectures simply by linking the appropriate library. In addition, researchers in numerical linear algebra are largely spared the need for detailed technical expertise about low-level programming and computer hardware.

The BLAS was subsequently expanded and extended to include an API for C in addition to Fortran. The current BLAS routines are classified into 3 levels:

- Level 1: vector-vector operations involving $O(N)$ operations and $O(N)$ data.
- Level 2: matrix-vector operations involving $O(N^2)$ operations and $O(N^2)$ data.
- Level 3: matrix-matrix operations involving $O(N^3)$ operations and $O(N^2)$ data.

The favourable ratio of operations to data allow the Level 3 BLAS to take advantage of the cache memory hierarchy found in all modern computers.

Four data types

We will look at some example programs that use a Level 1 BLAS operation called **axpy**, standing for "αx plus y". This operation overwrites the vector **y** with **ax+y** for a given scalar **a** and vector **x**. The Fortran BLAS actually provide 4 routines,

saxpy, **daxpy**, **caxpy**, **zaxpy**,

where the first letter in each name indicates the data type:

- s: single precision real (**Float32**).
- d: double precision real (**Float64**).
- c: single precision complex (**Complex**{**Float32**}).
- z: double precision complex (**Complex**{**Float64**}).

Julia and BLAS

You can see that Julia uses OpenBLAS by typing the command

```
julia> versioninfo()
```

Later, we will see how to call an OpenBLAS routine from Fortran or C.

Examples not using a library

Type

```
$ make
```

to build the Fortran executable **example_my_blas**, and then execute this program in the usual way:

```
$ ./example_my_fblas
```

Check that the output makes sense.

In this example, we use the object file **my_fblas.o** directly when creating the executable.

Similarly for the C version:

```
$ make example2
```

Creating a library

Linux shared libraries have the extension **.so** and traditionally start with **lib**. Type

```
$ make libmyblas.so
```

to generate the commands

```
gfortran -c -fpic my_fblas.f08
gcc -c -fpic my_cblas.c
gfortran -o libmyblas.so -shared my_fblas.o my_cblas.o
```

Here, the -**fpic** option is needed to generate position-independent code, and the -**shared** option causes a shared library to be created. The **file** command produces the output

```
libmyblas.so: ELF 64-bit LSB shared object, x86-64 ...
```

Notice that **libmyblas.so** contains both a Fortran object file and a C object file.

The **readelf** command can be used to display information about object file or shared library:

```
$ readelf -s my_fblas.o
$ readelf -s my_cblas.o
$ readelf -s libmyblas.so | grep daxpy
```

Notice how the Fortran compiler's name-mangling scheme means that the location of the **daxpy** subroutine from **my_fblas.f08** is referenced in **libmyblas.so** by the symbol **__myblas_MOD_daxpy**. Contrast this outcome with the behaviour of the C compiler, which does not mangle the name of **daxpy** from **my_cblas.c**.

The **bind(c)** attribute on **impersonate_c_daxpy** in **my_fblas.f08** tells the Fortran compiler not to mangle the name, as if it were a C function.

Examples using a non-system library

The command

```
$ gfortran -o example3 example_my_fblas.o
```

fails with the error message

```
example_my_fblas.o: In function ‘MAIN__’:
example_my_fblas.f08:(.text+0x44b):
undefined reference to ‘__myblas_MOD_daxpy’
collect2: error: ld returned 1 exit status
```

By including the shared library,

```
$ gfortran -o example3 example_my_fblas.o libmyblas.so
```

we can create an executable **example3**, but attempting to run this program gives another error message

```
./example3: error while loading shared libraries:
libmyblas.so:
cannot open shared object file:
No such file or directory
```

The problem is that at runtime only a list of standard system directories gets searched for libraries. The **ldd** command is used to list the library dependencies of an executable. Doing

```
$ ldd example3
```

lists several libraries but notice the line

```
libmyblas.so => not found
```

To fix this problem we need an extra option **-Wl,-rpath=.** to tell the link loader to add the current directory **.** to the runtime search path. This extra option is provided in the makefile rule for **example3**, so you need only type

```
$ make example3
```

Now, **ldd** will show

```
libmyblas.so => ./libmyblas.so
```

and **example3** will produce the expected output.

The same considerations apply using the C version. Doing

```
$ make clean
$ make example4
```

produces

```
gcc -c example_my_cblas.c
gfortran -c -fpic my_fblas.f08
gcc -c -fpic my_cblas.c
gfortran -o libmyblas.so -shared my_fblas.o my_cblas.o
gcc -o example4 example_my_cblas.o -L. -lmyblas \
        -Wl,-rpath=.
```

and **example4** works as expected. As before, the list of library dependencies

```
$ ldd example4
```

shows

```
libmyblas.so => ./libmyblas.so
```

Examples using a system library

Type

```
$ make clean
$ make example5
```

to produce

```
gfortran -c example_system_fblas.f08
gfortran -o example5 example_system_fblas.o -lblas
```

The option **-lblas** tells the linker to look for a shared library **libblas.so** in library search path. Doing

```
ldd example5
```

shows that

```
libblas.so.3 => /usr/lib/libblas.so.3
```

Similarly,

```
$ make example6
```

produces

```
gcc -c example_system_cblas.c
gcc -o example6 example_system_cblas.o -lblas
```

Note that the line

```
#include<cblas.h>
```

includes the header file **cblas.h** from one of the system directories. You might need to use the **-I** compiler option to specify explicitly a directory to search for the header file.

Libraries and package management

The software that makes up a Linux distribution is organised into packages. If you want to use a particular library, the first step is to find out if it has been packaged by your distribution. End-user applications can usually be installed using a graphical software management tool, but compilers and libraries typically have to be installed by console applications.

The two most widely used package formats are

.rpm Red-Hat package

.deb Debian package

For example, Ubuntu uses **.deb** packages and you can manage these using **apt** (Advance Package Tool) and **dpkg** (Debian Package Manager). Fedora uses **.rpm** packages, managed with **dnf** and **rpm**.

Ubuntu 16.10

To see if openblas is available, we can do

```
$ apt-cache search openblas
```

and get a list showing the packages

```
libopenblas-dev - Optimized BLAS ...
libopenblas-base - Optimized BLAS ...
```

We can install these two packages as follows.

```
$ sudo apt-get libopenblas-dev
```

To see the files installed from the package, do

```
$ dpkg -L libopenblas-dev
```

and for information about the package

```
$ dpkg -s libopenblas-dev
```

In this way we can locate the files needed:

```
/usr/lib/libopenblas.so
/usr/include/openblas/cblas.h
```

Fedora 25

To see if openblas is available, we do

```
$ dnf search openblas
```

and get a list of 8 packages, but the command

```
$ sudo dnf install openblas-devel
```

installs the other 7 as dependencies. To see the files installed from the package, do

```
$ rpm -ql openblas-devel
```

and for information about the package

```
$ rpm -qi openblas-devel
```

In this way we can locate the files needed:

```
/usr/lib64/libopenblas.so
/usr/include/openblas/cblas.h
```

Calling a library routine from Julia

The Julia module **MyBlas.jl** defines four versions of the **daxpy!** function:

1. **daxpy!**: a pure Julia function.
2. **fortran_daxpy!**: a Julia wrapper around the Fortran subroutine **daxpy** from **myfblas.f08**.
3. **c_daxpy!**: a Julia wrapper around the C function **daxpy** from **mycblas.c**.
4. **impersonate_c_daxpy!**: a Julia wrapper around the Fortran subroutine with C linkage **impersonate_c_daxpy** from **myfblas.f08**.

Run the Julia script **julia_example.jl** and observe that all four functions produce the same results.

Each of the wrapper functions uses **ccall** to call the relevant routine from the shared library **libmyblas.so**.

In the case of **fortran_daxpy!**, this call is

```
1                           y::Vector{Float64}, incy::Int32)
2     @assert length(x) <= 1 + (n-1)*incx
3     @assert length(y) <= 1 + (n-1)*incy
4     ccall((:__my_fblas_MOD_daxpy, "./libmyblas.so"),
          Void,
```

The first argument of **ccall** is a pair giving the symbol for the routine and the path of the shared library, in this case

```
1  (:__myblas_MOD_daxpy, "./libmyblas.so")
```

The second argument is the return type, which (for any Fortran subroutine) is Void. The third argument is a tuple specifying the argument types of the Fortran subroutine, and finally the remaining arguments are the Julia variables that we want to pass as those arguments. Notice that Fortran arguments are passed by reference.

In the case of **c_daxpy!**, we have

```
                    y::Vector{Float64}, incy::Int32)
    @assert length(x) <= 1 + (n-1)*incx
    @assert length(y) <= 1 + (n-1)*incy
    ccall((:cblas_daxpy, SYSTEM_BLAS), Void,
```

In C, each scalar argument is passed by value, and each array argument via a pointer to the first entry.
Finally, for **impersonate_c_daxpy!**,

```
                                y::Vector{Float64}, incy::
                                    Int32)
    @assert length(x) <= 1 + (n-1)*incx
    @assert length(y) <= 1 + (n-1)*incy
    ccall((:impersonate_c_daxpy, "./libmyblas.so"),
```

This approach, of wrapping a Fortran procedure in another Fortran procedure with C linkage, and then wrapping the latter in Julia, is often the safest approach, especially if any arguments are strings or characters.

Prominent mathematical software libraries

Lapack is a large Fortran library of linear algebra routines for dense matrices.

Suitesparse is a C library for sparse matrix computations.

Arpack is a Fortran library of iterative methods for eigenvalue computations.

FLAME is an alternative to Lapack, written in C.

PETSc is a library to support parallel PDE applications.