HPSC – Lecture 22

Outline:

- Python plus Fortran: f2py
- LAPACK and the BLAS
- Python compilers
- File writing – ASCII vs binary
- Summary

Sample codes:

- `/codes/f2py`

f2py — combining Fortran and Python

Often want to use

Fortran for intensive computations,

Python to provide nice user interface, plot results,

automate a series of runs with different parameters,

do convergence tests as grid size is refined, etc.

Can write data files to disk from Fortran, read into Python,

This is what we've done for plotting in homeworks.

Sometimes nice to call Fortran directly from Python.

e.g. LAPACK is used under the hood in NumPy.

f2py provides a wrapper for Fortran code.

f2py — combining Fortran and Python

Basic idea:

`fortrancode.f90` contains a function or subroutine, e.g. function `f1(x)` that returns a single value.

```
$ f2py -m mymodulename -c fortrancode.f90
```

This creates a binary file `mymodulename.so` that can used as a Python module.

```
>>> from mymodulename import f1
>>> y = f1(3.)
```

f2py — function example

/codes/f2py/fcn1.f90

```
function f1(x)
    real(kind=8), intent(in) :: x
    real(kind=8) :: f1
    f1 = exp(x)
end function f1
```

Then we can do...

```
$ f2py -m fcn1 -c fcn1.f90
$ python
>>> import fcn1
>>> fcn1.f1(1.)
2.7182818284590451
```

f2py — subroutine example

/codes/f2py/sub1.f90

```
subroutine mysub(a,b,c,d)
    real (kind=8), intent(in) :: a,b
    real (kind=8), intent(out) :: c,d
    c = a+b
    d = a-b
end subroutine mysub
```

Then we can do...

```
$ f2py -m sub1 -c sub1.f90
$ python
>>> import sub1
>>> y = sub1.mysub(3., 5.)
>>> print y
(8.0, -2.0)
```

Note: Tuple `(c, d)` is returned by the Python function.

f2py — Jacobi iteration

/codes/f2py/jacobi1.f90

```
    subroutine iterate(u0,iters,f,u,n)
```

Takes input array `u0` of length `n` and right hand side array `f` and produces `u` by taking `iters` iterations of Jacobi.

/codes/f2py/plot_jacobi_iterates.py

```
    # Set u = initial guess;  f = rhs
    for nn in range(nplots):
        u = jacobi1.iterate(u, iters_per_plot, f)
        plt.plot(x, u, 'o-')
        plt.draw()
        time.sleep(.5)
```

Other wrappers...

- Cython: Allows writing C code embedded in Python.
  `http://www.cython.org/`
- Jython: For Java.
  `http://www.jython.org/`
- swig: Connects C and C++ to many other languages
  `http://www.swig.org/`

Mathematical Software

It is best to use high-quality software as much as possible, for several reasons:

- It will take less time to figure out how to use the software than to write your own version. (Assuming it's well documented!)
- Good general software has been extensively tested on a wide variety of problems.
- Often general software is much more sophisticated that what you might write yourself, for example it may provide error estimates automatically, or it may be optimized to run fast.

Software sources

- Netlib: http://www.netlib.org
- NIST Guide to Available Mathematical Software: http://gams.nist.gov/
- Trilinos: http://trilinos.sandia.gov/
- DOE ACTS: http://acts.nersc.gov/
- PETSc nonlinear solvers: http://www.mcs.anl.gov/petsc/petsc-as/
- Many others!

LAPACK — www.netlib.org/lapack/

Many routines for linear algebra

using non-iterative methods.

Typical name: XYYZZZ

X is precision

YY is type of matrix, e.g. GE (general), BD (bidiagonal),

ZZZ is type of operation, e.g. SV (solve system),
EV (eigenvalues, vectors), SVD (singular values, vectors)

LAPACK — www.netlib.org/lapack/

Many routines for linear algebra.

Typical name: XYYZZZ

X is precision

YY is type of matrix, e.g. GE (general), BD (bidiagonal),

ZZZ is type of operation, e.g. SV (solve system),
EV (eigenvalues, vectors), SVD (singular values, vectors)

Examples:

`DGESV` can be used to solve a general $n \times n$ linear system in double precision.

`DGTSV` can be used to solve a general $n \times n$ tridiagonal linear system in double precision.

Installing LAPACK

On Virtual Machine or other Debian or Ubuntu Linux:

```
$ sudo apt-get install liblapack-dev
```

This will include BLAS (but not optimized for your system).

Alternatively can download tar files and compile.

See complete documentation at
http://www.netlib.org/lapack/

The BLAS

Basic Linear Algebra Subroutines

Core routines used by LAPACK (Linear Algebra Package)
and elsewhere.

Generally optimized for particular machine architectures, cache hierarchy.

Can create optimized BLAS using
ATLAS (Automatically Tuned Linear Algebra Software)

See notes and `http://www.netlib.org/blas/faq.html`

- Level 1: Scalar and vector operations
- Level 2: Matrix-vector operations
- Level 3: Matrix-matrix operations

The BLAS

Subroutine names start with:

- S: single precision
- D: double precision
- C: single precision complex
- Z: double precision complex

Examples:

- SAXPY: single precision replacement of $y$by $ax + y$.
- DDOT: dot product of two vectors
- DGEMV: matrix-vector multiply, general matrices
- DGEMM: matrix-matrix multiply, general matrices
- DSYMM: matrix-matrix multiply, symmetric matrices

Using libraries

If `program.f90` uses BLAS routines...

```
$ gfortran -c program.f90
$ gfortran program.o -lblas
```

or can combine as

```
$ gfortran program.f90 -lblas
```

When linking together `.o` files, will look for a file called `libblas.a` (probably in `/usr/lib`).

This is a archived static library.

Using libraries

If `program.f90` uses BLAS routines...

```
$ gfortran -c program.f90
$ gfortran program.o -lblas
```

or can combine as

```
$ gfortran program.f90 -lblas
```

When linking together `.o` files, will look for a file called `libblas.a` (probably in `/usr/lib`).

This is a archived static library.

Can specify different library location using `-L/path/to/library`.

Making blas library

Download **http://www.netlib.org/blas/blas.tgz**.

Put this in desired location, e.g. `$HOME/lapack/blas.tgz`

```
$ cd $HOME/lapack
$ tar -zxf blas.tgz     # creates BLAS subdirect
$ cd BLAS
$ gfortran -O3 -c *.f
$ ar cr libblas.a *.o   # creates libblas.a
```

To use this library:

```
$ gfortran program.f90 -lblas \
           -L$HOME/lapack/BLAS
```

Note: Non-optimized Fortran 77 versions.

Better approach would be to use ATLAS.

Creating LAPACK library

Can be done from source at

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

but somewhat more difficult.

Individual routines and dependencies can be obtained from netlib, e.g. the double precision versions from:

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

Download `.tgz` file and untar into directory where you want to use them, or make a library of just these files.

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

DGESV — Solves a general linear system

http://www.netlib.org/lapack/double/dgesv.f

```
      SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV,
     &                  B, LDB, INFO )
```

`N` = size of system (square $N \times N$)

`A` = matrix on input, L,U factors on output,
`dimension(LDA,K)` with `LDA, K >= N`

`LDA` = leading dimension of A
(number of rows in declaration of A)

Example:

```
real(kind=8) dimension(100,500) :: a
! fill a(1:20, 1:20) with 20x20 matrix
n = 20
lda = 100
```

DGESV — Solves a general linear system

Example:

```
real(kind=8), dimension(100,500) :: a
real(kind=8), dimension(200,400) :: b
integer, dimension(600) :: ipiv
! fill a(1:20, 1:20) with 20x20 matrix
! b(1:20, 1:3) with 3 right hand sides

n = 20; nrhs = 3; lda = 100; ldb = 200

call dgesv(n, nrhs, a, lda, ipiv, b, ldb, info)
```

What is passed to `dgesv` is `start_address`, the address of first element of `a`. (Matrix is stored by columns)

Whenever `a(i,j)` appears in code, address is:

```
address = start_address + (j-1)*lda + (i-1)
```

DGESV — Solves a general linear system

```
      SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV,
     &                  B, LDB, INFO )
```

`NRHS` = number of right hand sides

`B` = matrix whose columns are right hand side(s) on input
solution vector(s) on output.

`LDB` = leading dimension of `B`.

`INFO` = integer returning 0 if successful.

`A` = matrix on input, L,U factors on output,

`IPIV` = Returns pivot vector (permutation of rows)
`integer, dimension(N)`
Row `I` was interchanged with row `IPIV(I)`.

Gaussian elimination as factorization

If A is nonsingular it can be factored as

$$PA = LU$$

where

P is a permutation matrix (rows of identity permuted),

L is lower triangular with 1's on diagonal,

U is upper triangular.

After returning from `dgesv`:

`A` contains L and U (without the diagonal of L),
`IPIV` gives ordering of rows in P.

Gaussian elimination as factorization

Example:

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 3 & 6 \\ 2 & 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 3 \\ 4 & 3 & 6 \\ 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/2 & -1/3 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & 6 \\ 0 & 1.5 & 1 \\ 0 & 0 & 1/3 \end{bmatrix}$$

`IPIV` = (2,3,1)

and `A` ends up as

$$\begin{bmatrix} 4 & 3 & 6 \\ 1/2 & 1.5 & 1 \\ 1/2 & -1/3 & 1/3 \end{bmatrix}$$

`dgesv` examples

See `/codes/lapack/random`.

Sample codes that solve the linear system $Ax = b$ with a random $n \times n$ matrix A, where the value n is run-time input.

`randomsys1.f90` is with static array allocation.

`randomsys2.f90` is with dynamic array allocation.

`dgesv` examples

See `/codes/lapack/random`.

Sample codes that solve the linear system $Ax = b$ with a random $n \times n$ matrix A, where the value n is run-time input.

`randomsys1.f90` is with static array allocation.

`randomsys2.f90` is with dynamic array allocation.

`randomsys3.f90` also estimates condition number of A.

$$\kappa(A) = \|A\| \, \|A^{-1}\|$$

Can bound relative error in solution in terms of relative error in data using this:

$$Ax^* = b^* \text{ and } A\tilde{x} = \tilde{b} \Rightarrow \frac{\|\tilde{x} - x^*\|}{\|x^*\|} \leq \kappa(A) \frac{\|\tilde{b} - b^*\|}{\|b^*\|}$$

Just-in-time compilers for Python

Standard implementation of Python as interpreted language.

Importing `mymodule.py` creates `mymodule.pyc`, which is Bytecode (portable code or pcode):

One-byte operators with operands,
Interpreted by software at runtime.

Runs much slower than compiled code that is machine-specific instructions.

Just-in-time compilers for Python

Standard implementation of Python as interpreted language.

Importing `mymodule.py` creates `mymodule.pyc`, which is Bytecode (portable code or pcode):

One-byte operators with operands,
Interpreted by software at runtime.

Runs much slower than compiled code that is machine-specific instructions.

Just-in -time (JIT) compilation: Converts bytecode at runtime into native machine code.

Can sometimes run faster than pre-compiled code.

Just-in-time compilers for Python

Examples:

- PyPy — alternative implementation of Python
- numba — compiles decorated code to LLVM (formerly Low Level Virtual Machine, compiler infrastructure)

Included in the Anaconda Python distribution

Numba — autojit decorator

In [1]:
```
def loopsum(n):
    x = 0
    for i in range(n):
        x = x + i
```

In [2]:
```
%timeit loopsum(10000)
```

```
1000 loops, best of 3: 495 us per loop
```

Numba — autojit decorator

In [1]:
```
def loopsum(n):
    x = 0
    for i in range(n):
        x = x + i
```

In [2]:
```
%timeit loopsum(10000)
```

```
1000 loops, best of 3: 495 us per loop
```

In [3]:
```
from numba import autojit
```

In [4]:
```
@autojit
def loopsum2(n):
    x = 0
    for i in range(n):
        x = x + i
```

In [5]:
```
%timeit loopsum2(10000)
```

```
1000000 loops, best of 3: 1.5 us per loop
```

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n
    do j=1,n
        do k=1,n
            write(21,'(e24.16)') u(i,j,k)
        enddo; enddo; enddo
```

How much disk space does this take?

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n
    do j=1,n
        do k=1,n
            write(21,'(e24.16)') u(i,j,k)
        enddo; enddo; enddo
```

How much disk space does this take?

A single number such as `0.4000000000000000E+01`
has 24 ASCII characters $\Rightarrow$ 24 bytes per value.

Total $24n^3$ bytes. E.g. $100 \times 100 \times 100$ grid: $n = 100 \Rightarrow$ 24 MB.

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n
    do j=1,n
        do k=1,n
            write(21,'(e24.16)') u(i,j,k)
        enddo; enddo; enddo
```

How much disk space does this take?

A single number such as `0.4000000000000000E+01`
has 24 ASCII characters $\Rightarrow$ 24 bytes per value.

Total $24n^3$ bytes. E.g. $100 \times 100 \times 100$ grid: $n = 100 \Rightarrow$ 24 MB.

Note: In memory storing one 8-byte float takes only 8 bytes.
($n = 100 \Rightarrow$ 8MB.) ASCII takes 3$\times$ the space.

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n
    do j=1,n
        do k=1,n
            write(21,'(e24.16)') u(i,j,k)
        enddo; enddo; enddo
```

How much disk space does this take?

A single number such as `0.4000000000000000E+01`
has 24 ASCII characters $\Rightarrow$ 24 bytes per value.

Total $24n^3$ bytes. E.g. $100 \times 100 \times 100$ grid: $n = 100 \Rightarrow$ 24 MB.

Note: In memory storing one 8-byte float takes only 8 bytes.

($n = 100 \Rightarrow$ 8MB.) ASCII takes $3\times$ the space.

Also takes additional time to convert to ASCII,

$\approx 10\times$ slower to write ASCII than dumping binary.

Binary output in Fortran

Can use unformatted write in Fortran:

```
! /codes/io/binwrite.f90

open(unit=20, file="u.bin", form="unformatted", &
     status="unknown", access="stream")
do j=1,100
    do i=1,500
        u(i,j) = real(m*(j-1) + i, kind=8)
    enddo
enddo

write(20) u   ! writes entire array in binary
close(20)
```

Ascii is 3x larger in size

The resulting binary file `u.bin` cannot be edited directly. But we can read it into Python...

Reading binary data files in Python

To recover `U` array of dimension $m \times n$ in Python:

```
# /codes/io/binread.py

import numpy as np

file = open('u.bin', 'rb')
uvec = np.fromfile(file, dtype=np.float64)

m,n = np.loadtxt('mn.txt',dtype=int)

# now use Fortran ordering to fill u by columns:
u = uvec.reshape((m,n),order='F')
```

Other options for binary data

Binary formats that contain a lot of metadata...

Hierarchical Data Format: HDF, HDF4, HDF5

HDF5 file structure includes two major types of object:

- Datasets: multidimensional arrays of a homogenous type
- Groups: container structures for datasets and other groups

See also: h5py, PyTables

Other options for binary data

Binary formats that contain a lot of metadata...

Hierarchical Data Format: HDF, HDF4, HDF5

HDF5 file structure includes two major types of object:

- Datasets: multidimensional arrays of a homogenous type
- Groups: container structures for datasets and other groups

See also: h5py, PyTables

NetCDF (Network Common Data Form): Built on top of HDF5.

See also ncdump, netcdf4-python

Summary, take away messages...

- Version control — git
  Use for all your projects, collaborations, ...
  Consider contributing to open source projects
    Submit a pull request

Summary, take away messages...

- Version control — git
  Use for all your projects, collaborations, ...
  Consider contributing to open source projects
    Submit a pull request

- Python, NumPy, SciPy, matplotlib, IPython
  Quickly trying out new ideas, optimize later
  Graphics and visualization
  Scripting to guide big computations
  Combining codes from different languages
  Many capabilities not seen in class, e.g.
    Manipulating text files, regular expressions, building web interfaces

Summary, take away messages...

- Fortran 90

  Compiled language

  Tightly constrained but can run very fast

  Native multi-dimensional arrays

Summary, take away messages...

- Fortran 90
  Compiled language
  Tightly constrained but can run very fast
  Native multi-dimensional arrays

- Makefiles
  Dependency checking
  Often used for building software

Summary, take away messages...

- Fortran 90
  Compiled language
  Tightly constrained but can run very fast
  Native multi-dimensional arrays

- Makefiles
  Dependency checking
  Often used for building software

- Debugging code
  Unit tests, pytest
  Print statements, pdb, gdb

Summary, take away messages...

- Fortran 90
  Compiled language
  Tightly constrained but can run very fast
  Native multi-dimensional arrays

- Makefiles
  Dependency checking
  Often used for building software

- Debugging code
  Unit tests, pytest
  Print statements, pdb, gdb

- Memory hierachy, cache considerations
  Consider layout of arrays in memory
  Aim for spatial and temporal locality

Summary, take away messages...

- Parallel computing

  Increasingly necessary for all computing

  Amdahl's law —

  inherently sequential code limits parallelization

  Weak vs. strong scaling

  Fine grain vs. coarse grain parallelism

  Load balancing

Summary, take away messages...

- Parallel computing
  Increasingly necessary for all computing
  Amdahl's law —
    inherently sequential code limits parallelization
  Weak vs. strong scaling
  Fine grain vs. coarse grain parallelism
  Load balancing

- OpenMP
  Assumes shared memory
  Often very easy to add to existing codes
  Need to worry about shared/private variables,
    race conditions

Summary, take away messages...

- MPI — Message Passing Interface

  Always assumes distributed memory

  Sharing data requires message passing

  SPMD: Single Program Multiple Data

  Entire program run by each process

  But different processes may take different branches

Summary, take away messages...

- MPI — Message Passing Interface
  Always assumes distributed memory
  Sharing data requires message passing
  SPMD: Single Program Multiple Data
  Entire program run by each process
    But different processes may take different branches

- Computer arithmetic
  Floating point number representation, 4 byte vs. 8 byte
  IEEE standards
  Reproducibility still difficult in parallel
  Relative error and precision possible

Summary, take away messages...

- Linear algebra
  LAPACK, BLAS — optimized code
  Iterative methods for large sparse system
  Poisson problems: $u_{xx} = f\ (x)$
  Two-dimensional Poisson problem $u_{xx} + u_{yy} = f(x, y)$

Summary, take away messages...

- Linear algebra
  LAPACK, BLAS — optimized code
  Iterative methods for large sparse system
  Poisson problems: $u_{xx} = f(x) \Rightarrow$ tridiagonal
  Two-dimensional Poisson problem $u_{xx} + u_{yy} = f(x, y)$

- Quadrature methods / numerical integration
  Midpoint, Trapezoid, Simpson Rules
  Monte Carlo methods in high dimensions

Summary, take away messages...

- Linear algebra
  Matrix norms and condition number of $Ax = b$
  LAPACK, BLAS — optimized code
  Iterative methods for large sparse system
  Poisson problems: $u_{xx} = f(x) \Rightarrow$ tridiagonal
  Two-dimensional Poisson problem $u_{xx} + u_{yy} = f(x, y)$

- Quadrature methods / numerical integration
  Midpoint, Trapezoid, Simpson Rules
  Monte Carlo methods in high dimensions

- Monte Carlo methods
  Pseudo Random Number Generation
  Use of seed for reproducibility
  Random walks

Happy Computing!