Some coding guidelines for writing high performance scientific applications

Alex Pletzer RSIS-GFDL

with

Balaji AOS-GFDL

Peter J Phillips GFDL

Alistar Adcroft AOS-GFDL

Koushik Ghosh SGI-GFDL

Giang Nong RSIS-GFDL

Alexander.Pletzer@noaa.gov

PPPL, March 11 2005

Motivation

- Carefully design your code now. Retrofitting later can be hell.
- Strive for:
  - robustness
  - performance
  - scalability in complexity
  - portability
  - code sharing

The hope

- Some of these features are antagonist. One aim of this talk is to stimulate a discussion about the costs and benefits of different programming styles.
- Hopefully, a “good” design will emerge, which we can apply to new code. A good design will be vindicated by the long life span of a program; a bad design by the need to consistently modify or rewrite the code.
- Caveat: At present there no consensus about what a good design is. I will have to be biased.

Background: FMS

The Flexible Modeling Infrastructure developed at GFDL [http://www.gfdl.noaa.gov/∼vb/FMSManual/]:

- Written in Fortran 90
- Support for several parallelisms: MPI-1, MPI-2, and SHMEM
- 2-D domain decomposition with explicit time stepping
- Collection of generic utilities (read/write NetCDF files, time arithmetic, diagnostics, profiling, etc.)
- Generous use of cpp directives to overload routines
- Each file is a Fortran MODULE with a CONTAINS statement.

Skeleton of a FMS module

```
module foo
  use bar, only : bsub1, bsub2,...
  private :: fsub1, fsub2,...
  public  :: foo_init, foo_exit,...
  ! some globals
  real, dimension(:), allocatable :: a, ...
  namelist / foo_nml / c, d, e...
  contains
subroutine foo_init(...)
...
! all routines follow here
end module foo
```

Many of us may have written codes similar to the above

Component/module based design

- Pros:
  - Safe. Routine interfaces will be checked at compilation.
  - Fast. Data communication to routines through globals
- Cons:
  - Complex module dependency must be solved to determine the compilation order.
  - Only one instance allowed.
  - Nearly impossible to invoke from C. Module declaration will mangle the routine names.

The pervasiveness of frameworks

The crisis:

- No or immature community-wide standard
- Hard to share modules between frameworks.
- Need to separate algorithms from frameworks (share algorithms not framework)
- Apply kernel-driver programming model. No framework dependency in algorithm.

Single instance? big deal!

- In general, only one customer allowed. Risk of global space pollution if more than one customer uses the module. Unsafe.
- “I don’t need more than one instance”. Never say never. There are very few cases where one would want to enforce a single instance for ever. Better leave this choice to the user.
- Multi-threaded code requires multiple instances.

What are threads?

- Like processes, threads are a sequences of instructions that can be executed in parallel. Contrary to processes, however, global data are shared among threads. Data in the local scope remain however private to a thread.
- Think of processes as MPI, think of threads as OpenMP
- The 2 great dangers lurking in multi-threaded programming are:
  - race conditions. Two or more threads attempt to write to the same memory address.
  - deadlocks. Excessive use of synchronization can make program “hang”.

Why is thread safety important?

- OpenMP allows you to apply another level of parallelism on top of MPI, say.
- OpenMP is ideally suited on shared memory machines.
- OpenMP is more flexible than MPI; threads can be forked and joined anywhere in your program.
- Your code may be used inside an OpenMP region.
- Multithreading will become the dominant programming paradigm in the coming decade according to Herb Sutter.

See “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software” [http://www.gotw.ca/publications/concurrency-ddj.htm].

Example of a race condition

Why are globals dangerous?

```
call foo(b)
print *,'b=',b
...
subroutine foo(b)
  real, save :: a=0
  b = a
  a = a + 1
end subroutine foo
```

Run serially or as an MPI process, the above code will print "b=0". Run with 2 threads, the above program will print "b=0; b=1" or "b=0;b=0".

Why SAVE is dangerous

Consider

```
subroutine sub(n)
        real, save, allocatable :: a(:)
        logical, save :: first_call=.TRUE.
        if(first_call) then
           first_call = .FALSE.
           allocate(a(n))
           ! preset a here...
        endif
        ! use ``a'' below
end subroutine sub
```

Problem: Thread 2 may attempt to use “a” before “a” was allocated by Thread 1. No easy fix.

Race condition cont'd

- Avoid using SAVE, COMMON, DATA or module globals
- Avoid procedures that have side-effects (a state)
- If you need globals, practice extreme care
  - Flag name of global variables (e.g. foo_a instead of a)
  - Ensure that each thread will write to a different array section (array indices must be thread private)
  - Make sure globals are allocated/deallocated outside threaded region.

If you use globals, the burden of proving that your routine is thread safe will fall on you.

Object oriented (OO) style

- Pack data in a derived type
- No globals above CONTAINS statement (PARAMETERs are OK)
- Methods may be defined under the CONTAINS statement
  - Init/Exit methods common to all objects (need not be threadsafe)
  - New/Del methods to construct/destroy objects
  - Setter/Getter methods to set/get data
  - Save/Load methods for persistence
  - Kernel routines

Note: the term OO used here is much looser than that defined elsewhere. No inheritance/dynamic polymorphism.

OO style

```
module foo
use bar, only : ....
   ! define compound type below
   type foo_obj
....
   end foo_obj
   PUBLIC :: foo_init, foo_exit,   &
  &          foo_new, foo_del, &
  &          foo_set_A,... foo_get_B,... &
  &          foo_save, foo_load, ...
  &          foo_kernel1,...
CONTAINS
   ! implementation
   ...
end module foo
```

Static and class methods

```
type(foo_obj) :: self
```

- Static methods don't take the self argument (foo_init and foo_exit). Static methods will not be required to be threadsafe.
- Class methods take self as first argument (all other methods). Class methods will be required to be thread-safe.

Each object (instance) shares read-only global data (e.g. module version, verbose switch, etc.). All allocation occurs in the foo_new method, memory is reclaimed in foo_del.

OO style in other languages

- derived type (F90) $\equiv$ struct (C) $\equiv$ class (C++ and Python)
- module (F90) $\equiv$ header (C and C++) $\equiv$ file (Python)
- ctor foo_new(self,..) (F90 and C) $\equiv$ foo(...) (C++) $\equiv$ __init__(self,...) (Python)
- dtor foo_del(self,..) (F90 and C) $\equiv$ foo(...) (C++)
- methods foo_meth(self,...) (F90 and C) $\equiv$ self.meth(...) (C++ and Python)
- static methods foo_init(...) (F90 and C) $\equiv$ foo.init(...) (C++ and Python)

The OO style presented here maps in straightforward fashion to C, C++, Python and other languages.

Pros and cons of OO style

- Pros:
  - Can have state and be threadsafe (one instance per thread).
  - Except for foo_init/foo_exit all other routines are threadsafe
  - Allows the module to evolve without breaking the API. (Add more stuff to self).
  - Safe. Interface checking at compilation.
- Cons:
  - May need POINTERs as arrays in compound types. Will this slow down my code?
  - Not multi-language friendly. How can I call this from C?

The problem with POINTERs

- Can lead to spurious copying. Arrays with the POINTER attribute must sometimes be copied in/out to prevent the dangling POINTER problem upon exit of a subroutine. Used to be a major problem.
- May inhibit optimization due to aliasing. Some loop re-arrangement can only be performed if all array elements point to a different address.
- May leak memory through the creation of temps. Some temps cannot be destroyed until the compiler can be sure that no other POINTER is sharing the same address.
- Performance degradation due to memory layout.

Aliasing in other languages

- C introduced the “restrict” qualifier
- C++ introduced valarray

Allocatable member arrays

- The problem of POINTERs is widely acknowledged.
- Fortran 2003 supports ALLOCATABLE members
- This feature is supported by many compilers (ifort, lf95, xlf95, nf95, ...)

```
#ifdef _F2000
#define _ALLOCATABLE allocatable
#define _ALLOCATED allocated
#define _NULL
#else
#define _ALLOCATABLE pointer
#define _ALLOCATED associated
#define _NULL =>NULL()
#endif
```

Allocatable members (2)

```
type foo_obj
 real, _ALLOCATABLE :: a(:)_NULL
end type
if(.NOT. _ALLOCATED(a)) ...
```

Will make your code portable until all compilers support allocatable components.

Pointer pitfalls

```
if(associated(ptr)) nullify(ptr)
```

Nullify will not deallocate $\rightarrow$ memory leak. Use instead

```
if(associated(ptr)) deallocate(ptr, stat=ier)
```

This will succeed regardless whether ptr was allocated or not. Deallocate is more generic, will work also for allocatable arrays.

Pointer pitfalls (2)

```
real, pointer :: ptr(:)
real, pointer :: trg(:)
...
allocate(trg(...))
...
ptr => trg          ! no need to allocate ptr
deallocate(ptr)     ! same as deallocate(trg)
trg = ...           ! Memory error
```

Can deallocate target memory through associated pointer because ptr points to the entire of trg.

Other performance issues

Overall performance of a code can be affected by:

- whether or not assumed-shape arrays are used in subroutine calls
- whether or not arrays are allocated dynamically or statically
- auto vs allocatable arrays

We're moving away from assumed-shape arrays.

```
subroutine foo(a)
    real :: a(:) ! assumed shape
```

vs

```
subroutine foo(is, ie, a)
    real :: a(is:ie) ! pass shape explicitly
```

To copy or not to copy

Ambiguous case:

```
call sub(object(:) % member)
```

Equivalent pseudo-code:

```
temp(:) = object(:) % member
call sub(temp)
object(:) % member = temp(:)
```

Avoid striding in subroutine calls. Copy explicitly when you want a copy. Copying sometimes is OK and may be more performant. May need to change the memory layout of data structure to improve performance.

Summary

Recommended practices

- Separate framework from algorithms (kernels)
- Keep it simple. Simple routine interfaces. Communicate via primitive data structures when possible.
- Beware of SAVE, DATA, COMMON.
- Use module globals sparingly.
- Beware of POINTERs
- Be explicit when copying is desired
- Write unit tests
- Carefully choose your memory layout
- Favour procedural over OO for kernels

Summary (2)

Contentious recommendations

- Favour OO over side-effects in procedures (incl. kernels).
- Consider implementation in C to facilitate interlanguage interoperability.

Open issues are how large a list of subroutine arguments can grow (10, 50 or 500?) before considering packaging in derived types. What to do with constants? Should optional arguments be allowed? Which is better: arrays allocated on the stack (auto) vs arrays on the heap (allocatable)? Pass temporary arrays for storage or not?

Word of wisdom

Don't fall victim to the tyranny of a compiler. Many performance issues are with specific compilers and in specific situations.

The last word

Although highly desirable in many cases, the procedural style does not scale well in complexity. In this case OO should be warranted.

Further pointers

- A tool to generate a Makefile given a set of Fortran modules: http://www.gfdl.noaa.gov/∼vb/mkmf.html
- About thread safety: http://www-106.ibm.com/developerworks/java/library/j-jtp09263.html
- About copying in/out POINTER arrays: http://w3.pppl.gov/∼pletzer/performance/f90pointer.html
- About multi-language friendly programming: http://w3.pppl.gov/∼pletzer/programs/fortran-c-c++/opaque_handles.html
- Earth System Modeling Framework: addresses many multi-language issues. http://www.esmf.ucar.edu/main_site/esmf_home.html