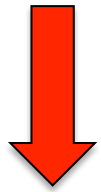
Appraising Your Programming Skills

David McCaughan, *HPC Software Analyst*
SHARCNET, University of Guelph
dbm@sharcnet.ca

Motivation

- Summer school is always fairly programming-oriented
  - use of pre-made packages requires less training
  - domain specificity of packages limits audience

- Typical pre-requisite statement:
  - *"we assume no prior knowledge of parallel programming; however, we assume you have some experience with serial programming"*
  - people's notions of "some experience with serial programming" can differ dramatically; where do you fit?
    - dabbled with scripting languages
    - written or modified some code, but nothing extensive
    - significant programming in some compiled language, and generally comfortable looking at languages never seen before
    - extensive experience with programming in multiple languages

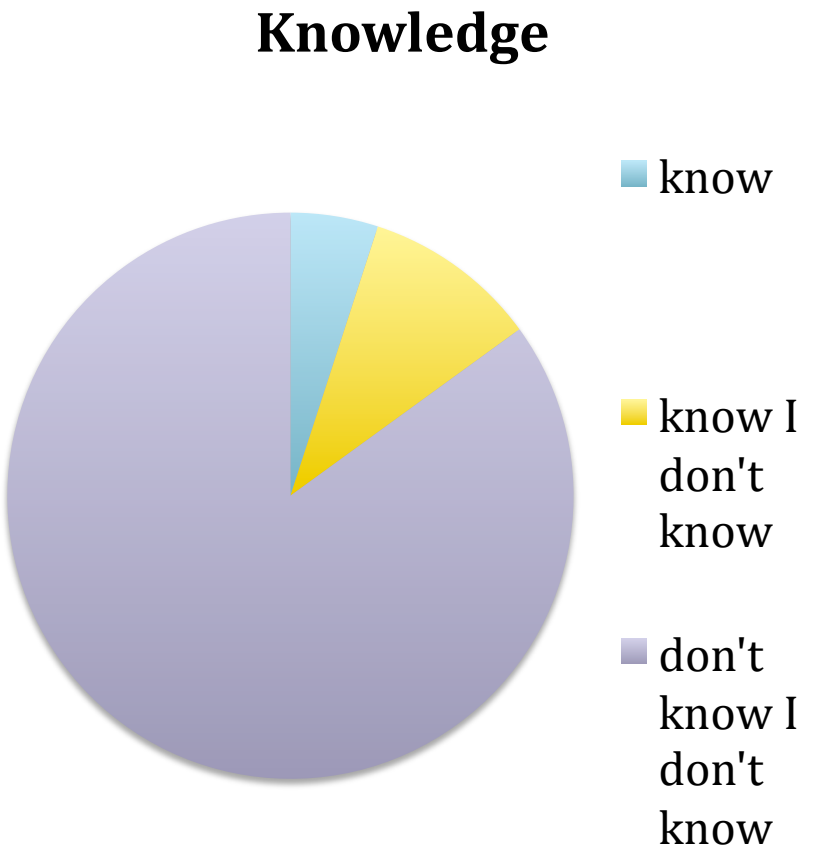


these people tend to be ok, the rest may be underestimating things...

How do I know what I don't know?

- J S Atherton (among others) characterize the universe of our knowledge along axes of "know" and "don't know"

- There are things I:
  - *know I know*
  - *know I don't know*
  - *don't know I don't know*
    - this one is the mind killer
  - (don't know I know is also fair, but misses the point)



A survey of some programming issues…

- The goal in this seminar:
  - illustrate some common, fundamental programming concepts in C, C++, Fortran
  - allow the novice programming to better assess what they don't know they don't know

- The intention is not to demonstrate your code-fu is weak
  - make you aware that there is more to productive programming than loops and statements
  - lack of experience is always the impediment moving to parallel programming—be realistic in appraising your skills

The many faces of Fortran

- Fortran == Formula Translation
  - procedural imperative programming language for math and science applications (as opposed to C which is systems oriented and provides a lower level interface)
- Fortran tends to track C and its derivatives
  - most people don't want to rewrite hereditary code
- That being said: the older standards receive a lot of attention from compiler developers
  - primarily to maintain the hereditary code base and see it move forward with new hardware
  - this can be a catch-22

Fortran77

- This language dates to the punch-card era
  - fixed format source == whitespace matters (or lack of it)
  - no dynamic memory
  - no structured programming
  - poor type support
    - precision depends on architecture, compiler, etc.
  - lots of non-standard extensions in compilers and their runtimes that result in portability issues

```
C F77:
        INTEGER A(200,200,200)
C … INFINITE LOOP – BEWARE OF THE GOTO
20      DO I=1,200
          A(I,I,I)=42
        ENDDO
90      GOTO 20
C…
        PRINT 10, 'TEST PASSED',I,'X',J,'Y',K
        PRINT *, '----FORTRAN RULES AT STRING HANDLING----'
10      FORMAT(A,I4,A,I4,A,I4)
```

Fortran90

- "Fortran with C features"
  - free-form source code
  - dynamic memory management, pointers, recursion
  - modules
    - providing interfaces and grouping together closely related code and data in a modular fashion

```
! f90:
recursive subroutine bisect(x,y)
use blas
    integer(kind=4), allocatable, dimension(:,:,:) :: a

!...

    allocate(a(x,x,x))
    do i=1,x
        a(i,i,i)=42_4

!...
    call spmvd(a,x,y)
    deallocate(a)
end
!etc.
```

Fortran2003

- Object-oriented programming support
  - type extensions and inheritance, polymorphism, dynamic type allocation and type-bound procedures
  - input/output enhancements including the FLUSH statement and access to error messages

- In terms of MPI and conventional parallel processing, most libraries support F77 and F90
  - features from Fortran95, most notably HPF, failed to get widespread adoption so it may not be as portable

Re-inventing the wheel

- It is common to want to code everything from first principles
  - with significant effort and knowledge, you may even produce something near-optimal
  - with no effort you can typically leverage existing libraries that do common tasks for you, which typically have been written by very bright people who have optimized the hell out of it
- For certain problem types common in the physical sciences (linear algebra, etc.), historically optimized code will have been compiled and run faster than any other language bar assembly in most cases
  - if what you're doing is fundamental, look for a library
  - BLAS, LAPACK, MKL, FFTW, etc.

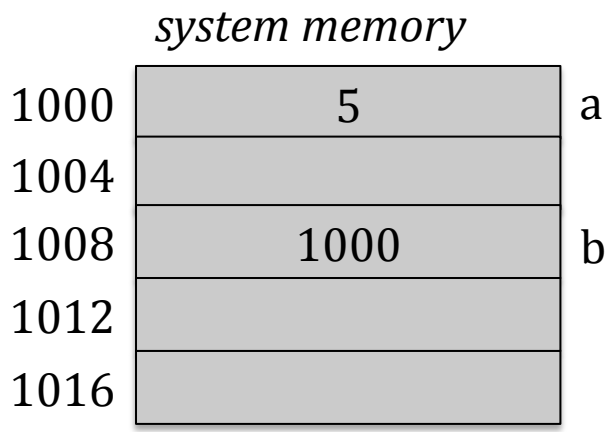
Pointers

- Pointers provide a means of accessing data and code *indirectly* through the address of the item
  - being able to deal with pointers is absolutely fundamental to getting anything done in C/C++ (F90)
  - pointers to functions, pointers to buffers, passing arrays, changing the value of arguments to functions, "generic" function behaviour, etc.

```
int a = 5;
int *b;

…

b = &a;
```



Pointers come with a price

- Easily the #1 problem for the inexperienced
  - no way around it – must understand what is going on
- Pointers are *aliases* for other variables
  - makes it difficult for the compiler to optimize code automatically
  - once you know what you are doing, you can more freely use pointers and maintain optimization options
    - e.g. *restrict* keyword
- Memory management becomes *very* important
  - memory allocated to pointers and not freed properly lead to memory *leaks*
    - easy to ignore them for code that doesn't run for long, quickly gets out of control for long-running processes
  - `valgrind` is a tool you can use to help with this

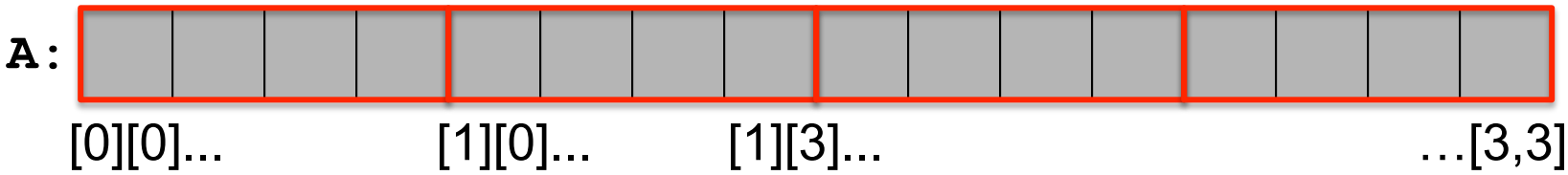
Array/memory organization

- Regardless of abstraction, memory is an inherently linear thing
  - it is important to understand this to maximize efficiency

- Moving data around is typically done in "buffers"
  - a buffer is just a contiguous region of memory
  - learn to take advantage of contiguous storage, whether it occurs that way naturally, or you impose it

```
/* C: row-major ordering for arrays */

double A[4][4];
```



Array/memory organization (cont.)

- Performance warning: the layout of memory is significant
  - ensure you stride through memory in the underlying order to optimize cache-line reuse:

```
/* C: row-major ordering for multi-dimensional arrays */

double a[x][y][z], b[x][y][z];

for (i = 0; i < x; i++)                    /* rows as outermost indices */
    for (j = 0; j < y; j++)
        for (k = 0; k < z; k++)
            a(i,j,k) = a(i,j,k)+b(i,j,k)  /* linear memory access, stride 1 */
```

```
! Fortran: column-major ordering for multi-dimensional arrays

real(kind=8) dimension(x,y,z) :: a,b

do k=1,z                                   ! columns as leading indices
    do j=1,y
        do i=1,x
            a(i,j,k) = a(i,j,k)+b(i,j,k)  ! linear memory access, stride 1
```

Dynamic memory allocation

- What is the difference between the way the following 2-D arrays are allocated?

```
int **2DAlloc1(int M, int N)
{
    int **a;

    a = malloc(sizeof(int *) * M);

    for (i = 0; i < M; i++)
        a[i] = malloc(sizeof(int) * N);

    return(a);
}
```

```
int **2DAlloc2(int M, int N)
{
    int *tmp, **a;

    a = malloc(sizeof(int *) * M);

    tmp = malloc(sizeof(int) * M * N);

    for (i = 0; i < M; i++)
        a[i] = &tmp[M*i]);

    return(a);
}
```

- Does it make a difference?
  - or more appropriately, *will it* make a difference?
- Important that you understand the implications of contiguous vs. non-contiguous allocation
  - what about `int a[M][N];`

Command-line arguments

- Cutting corners: hard coded values
  - input/output files, sizes, parameters, etc.
  - this limits the flexibility of the code in situations you might not anticipate
    - running hundreds of instances program with hard-coded input/output files at the same time

- Get comfortable with handling command-line arguments
  - C example:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    printf("# arguments: %d\n",argc);
    for (i = 0; i < argc; i++)
        printf("arg%d = %s\n",i,argv[i]);

    return(0);
}
```

The Fortran tax

- Fortran remains in use, some may even say commonly… but:
  - increasingly restricted to mathematics, physics and chemistry
  - rarely even mentioned to computer scientists, never mind used
  - some have characterized it as a downstream boutique language at this point

- Cost:
  - Fortran tools are becoming increasingly expensive as the user base becomes more niche (e.g. Intel compilers)
  - Fortran interfaces to “breakthrough” technologies are often unavailable (at best, lagging years behind C/C++ offerings)

- There is nothing wrong with the language
  - but be aware of the potential limitations in planning/expectations

Data representation (C/Fortran)

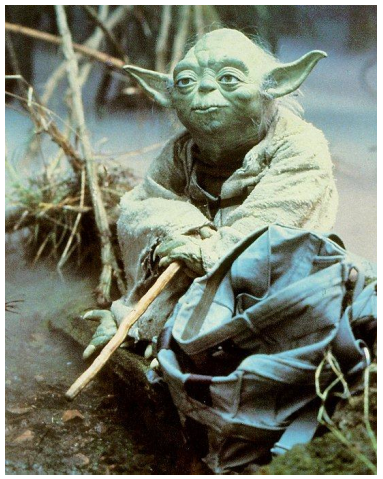
- It isn't uncommon to need to link Fortran and C code
  - this is a whole new world of "understanding" that is required
- Name mangling conventions can differ
  - this is a compiler issue
  - most function symbols have leading underscores added --- 1 or 2?
  - can generate some very interesting link errors
- Endian issues
  - storage of integer values is not always consistent
- Array organization
  - storage of a multi-dimensional array in C is row-major, Fortran assumes column major

Data representation (C++)

- Object-oriented code is all about encapsulating data and operations on that data
  - encourages the programmer to treat the objects as abstract types (which is a good thing from a software engineering perspective)

- The lack of an introspection mechanism in C++ makes it difficult to deal with (automatically) serialized objects
  - the first thing you'll want to do in parallel C++ is send objects around—which you can't do
    - well you can, but it's all manual and very expensive so to be avoided
  - take this into consideration when designing your classes, and planning for communication

Design philosophies



- Common software engineering wisdom:
  - encapsulate data and operations on the type
  - object-orientation takes this as an axiom
  - focus is on modularity and abstraction
- Must unlearn (some of) what you have learned
  - in a parallel world, cost of communication is everything
  - strive to keep data that can be sent/received together
  - consider: a large number of objects or types with a floating point value that must be communicated between processes regularly
    - OO design says bind that value to the object – this can encourage you to send them all in a loop
    - cost of communication says have them all in an array together so you can send them all at once (latency!)

The Deadliest Warrior: interpreted vs. compiled

- Many people have experience programming with an interpreted language such as Python (even Java falls into this category)
  - since it's what you know, you would like to continue just using it
  - understand that the reason scripting languages are relatively "easy" is because of the volume of processing being done behind your back
    - e.g. string slicing: creates a new string ever time!
    - e.g. garbage collected: overhead + leaks
- Python example: string similarity metric
  - naïve Python implementation: ~0.5 strings per second
  - optimized Python implementation: ~4000 strings per second
  - optimized C implementation: ~4 000 000 strings per second
- If performance doesn't matter, the convenience of scripting is worth it; otherwise understand the high price you can pay
  - bottom line: right tool for the right job

Knowledge of standards

- Your knowledge of the language standard will correlate directly with:
  - current understanding
    - correctly predict behaviour (even with compiler flags)
  - future flexibility
    - adhering to the standards maximizes compatibility with future compilers (and we all know, some code will never die)
  - portability
    - avoiding vendor-specific extensions (which requires that you know what are the extensions) make it easier to move the code between systems.

- This can be very subtle: are the following types equivalent in C?
  - `int a[10];`
  - `int *a;`

Knowledge of standards (cont.)

- Example: calling a function carries a cost
  - some languages provide an *inline* keyword – do you know what it does? are there other options with more determinism?

- Recall the interpreted/scripting example:
  - do you fully understand what is going on "under the hood" when you use features of a language?
  - "easy" usually correlates with "computationally expensive"
    - just because you didn't write the code to do it all, doesn't mean something isn't being done

- In general, portability is a major issue in Fortran
  - explosion of vendor-specific intrinsic features/functions
  - avoid using these if you value portability
    - at the very least, *know* that you're using something non-standard

Other issues that have come up…

- File handling
  - opening files
  - traversing directories
- String handling
  - dynamic construction of paths (for files for example)
  - `sprintf(),strcat()`
- Calling external programs
  - `system(),popen()`

Get ready for coding in parallel…

- Computational/communication costs are the altars at which you shall worship
  - function calls aren't free
  - abstract language features aren't free
  - contiguous memory is generally your friend
  - being fast and loose with memory usually isn't a good idea
  - be able to visualize actions over time
    - avoid imposing serial behaviour in what is intended to be parallel code

- Understand the resources available for your language of choice
  - e.g. man pages, good reference for language standard