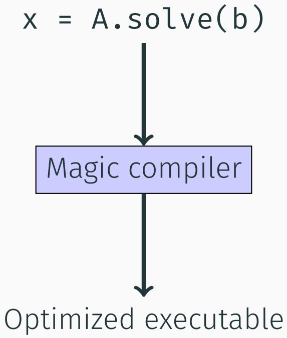
CS 5220: Mixed languages, libraries, and frameworks

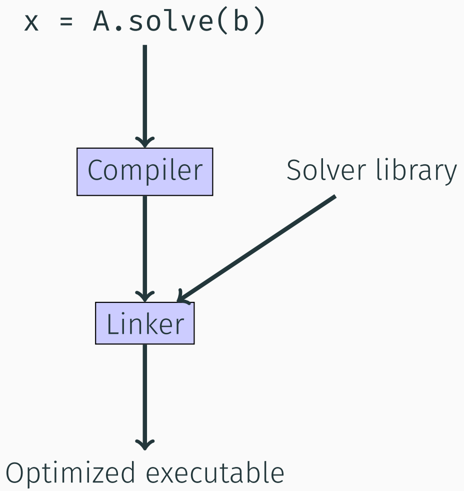
David Bindel

2017-11-21

Nerdvana?



Nerdvana?


x = A.solve(b)
Compiler
Solver library
Linker
Optimized executable

Wait, what?

- *Compiler* maps source code to assembly
- *Assembler* maps to object files
- *Librarian* produces
  - Static *archives* (`.a`)
  - Dynamic libraries or *shared objects* (`.so`)
- *Linker* combines objects and resolves symbols
- *Loader* brings executable into memory

Usually worry about compile and link.

Wait, what?

Consider calling LAPACK from C++.

```
extern "C"
int dgesv_(const int& n, const int& nrhs,
           double* A, const int& ldA, int* ipiv,
           double* B, const int& ldB, int& info);

...
dgesv_(n,1, A,n,ipiv, b,n, info);
if (info != 0)
   complain_bitterly();
```

Need to understand LAPACK conventions, name mangling (C++ and Fortran), calling conventions (by value/reference), one-vs-zero-base indexing (`ipiv`), ...

NB: Can circumvent some with `iso_c_binding` wrapper

Wait, what?

Consider calling LAPACK from C++.

```
 mycode.x: mycode.o
  g++ -o mycode.x mycode.o \
    -llapack -lopenblas -lgfortran -lm

mycode.o: mycode.cc
  g++ -c -O2 mycode.cc
```

Need to understand inter-library dependencies, role of order on link line, role of front-end in bringing in language support libraries, ...

NB: Using wrapper libraries may not make this part simpler...

Nerdvana?

```
x = A\b;
```

The mortal realm

```
umfpack_di_symbolic(n, n, Ap, Ai, Ax, &Symbolic,
                    NULL, NULL);
umfpack_di_numeric(Ap, Ai, Ax, Symbolic, &Numeric,
                    NULL, NULL);
umfpack_di_free_symbolic(&Symbolic);
umfpack_di_solve(UMFPACK_A, Ap, Ai, Ax, x, b, Numeric,
                 NULL, NULL);
umfpack_di_free_numeric(&Numeric);
```

Hell

Write and tune a parallel sparse direct solver (in assembly?)

The mountain of abstraction

Consider the trajectory of the class:

- Started very low-level (caches, vector units, etc)
- Up to general ideas/kernels (tiling, matrix multiply)
- Up to parallel concepts, application ideas
- Nirvana: high-level code, performance “just happens?”

Maybe the grass is greener across the C...

Low-level frameworks and languages

- OpenMP and MPI (of course)
- Intel Thread Building Blocks (TBB)
- Global arrays
- Newer (?) parallel languages and extensions
    - Cilk++
    - UPC++
    - Titanium
    - Chapel

Libraries

One thing (or a few) done fast:

- BLAS (MKL, OpenBLAS, ATLAS, etc)
- LAPACK and successors
- FFTW
- Sparse direct solvers

Key challenge: linking (esp across languages)

Framework libraries

- Many in PDE land
    - PETSc, SLEPc, TAO, etc
    - Trilinos
    - Overture
    - deal.ii
- Interface more complicated than procedure call
- Effectively defines embedded solver language

Key challenge: learning framework + build/link

Runtime frameworks

- Lots of trendy examples
  - MapReduce / Hadoop
  - Pregel, GraphLab, PowerGraph, Ligra, etc
  - Spark
- Write code to match interface desired by framework
- Promise: “Code like this, we’ll make it go fast”
  - Great when it works!
  - Sometimes not as fast as you’d hope

Key challenge: map your problem to desired form

Scripting languages and PSEs

- Matlab, Octave, R, Python, Julia
- “High productivity” vs “high performance”?
- Not necessarily slow!
    - Speed via extensions (Cython, MWrap, etc)
    - Speed via Jit (Matlab, Julia, Python Numba)
    - Speed via BLAS3 calls (all of the above)
    - Often some parallel support as well
- Performance strategies transfer
    - Model and understand data access
    - Profile and tune
- Bottlenecks may not be where you expect

Key challenge: map your problem to fit language strengths

Domain specific languages

- Classic example: SQL
- PDE domain: finite element compilers
    - Dolfin framework
    - Sundance
- Embedded languages/specializers (PyCUDA, SEJITS)

Key challenge: great opportunities from limited scope

Simulation codes

- ANSYS, ABAQUS, LS-DYNA, OpenSEES, FEAP, COMSOL, FLUENT, OpenFOAM, SPICE, Cadence, BioSPICE, ...
- Typical pattern
    - Custom language (or preprocessor) for problem input
    - Scripting language to describe analysis
    - User-defined elements/modules in compiled language
- Great for some classes of problems
- Can often be tortured into covering other types!

Key challenge: limited scope

High performance + high productivity?

Warning: Strong opinion ahead!

Scripting is one of my favorite hammers!

- Used in my high school programming job
- And in my undergrad research project (tkbtg)
- And in early grad school (SUGAR)
- And later (FEAPMEX, HiQLab, BoneFEA)

I think this is the Right Way to do a lot of things.
But the details have changed over time.

The rationale

Why is MATLAB nice?

- Conciseness of codes
- Expressive notation for matrix operations
- Interactive environment
- Rich set of numerical libraries

… and codes rich in matrix operations are still fast!

The rationale

Typical simulations involve:

- Description of the problem parameters
- Description of solver parameters (tolerances, etc)
- Actual solution
- Postprocessing, visualization, etc

What needs to be fast?

- Probably the solvers
- Probably the visualization
- Maybe not reading the parameters, problem setup?

So save the C/Fortran coding for the solvers, visualization, etc.

Scripting uses

Use a mix of languages, with scripting languages to

- Automate processes involving multiple programs
- Provide more pleasant interfaces to legacy codes
- Provide simple ways to put together library codes
- Provide an interactive environment to play
- Set up problem and solver parameters
- Set up concise test cases

Other stuff can go into the compiled code.

Smorgasbord of scripting

There are *lots* of languages to choose from.

- MATLAB, LISPs, Lua, Ruby, Python, Perl, …

For purpose of discussion, we'll use Python:

- Concise, easy to read
- Fun language features (classes, lambdas, keyword args)
- Freely available with a flexible license
- Large user community (including at national labs)
- "Batteries included" (including SciPy, matplotlib, Vtk, …)

Truth in advertising

Why haven't we been doing this in class so far? There are some not-always-simple issues:

- How do the languages communicate?
- How are extension modules compiled and linked?
- What support libraries are needed?
- Who owns the main loop?
- Who owns program objects?
- How are exceptions handled?
- How are semantic mismatches resolved?
- Does the interpreter have global state?

Still worth the effort!

Simplest scripting usage

- Script to prepare input files
- Run main program on input files
- Script for postprocessing output files
- And maybe some control logic

This is portable, provides clean separation, but limited.

Scripting with IPC

- Front-end written in a scripting language
- Back-end does actual computation
- Two communicate using some simple protocol via inter-process communication (e.g. UNIX pipes)

This is the way many GUIs are built. Again, clean separation; somewhat less limited than communication via filesystem. Works great for Unix variants (including OS X), but there are issues with IPC mechanism portability, particularly to Windows.

Scripting with RPC

- Front-end client written in a scripting language
- Back-end server does actual computation
- Communicate via *remote procedure calls*

This is how lots of web services work now (JavaScript in browser invoking remote procedure calls on server via SOAP). Also idea behind CORBA, COM, etc. There has been some work on variants for scientific computing.

Cross-language calls

- Interpreter and application libraries in same executable
- Communication is via “ordinary” function calls
- Calls can go either way, either extending the interpreter or extending the application driver. Former is usually easier.

This has become the way a lot of scientific software is built — including parallel software. We’ll focus here.

Concerning cross-language calls

What goes on when crossing language boundaries?

- Marshaling of argument data (translation+packaging)
- Function lookup
- Function invocation
- Translation of return data
- Translation of exceptional conditions
- Possibly some consistency checks, book keeping

For some types of calls (to C/C++/Fortran), automate this with *wrapper generators* and related tools.

Wrapper generators

Usual method: process interface specs

- Examples: SWIG, luabind, f2py, …
- Input: an interface specification (e.g. cleaned-up header)
- Output: C code for gateway functions to call the interface

Alternate method: language extensions

- Examples: weave, cython/pyrex, mwrap
- Input: script augmented with cross-language calls
- Output: normal script + compiled code (maybe just-in-time)

Example: `mwrap` interface files

Lines starting with # are translated to C calls.

```
function [qobj] = eventq();
  qobj = [];
  # EventQueue* q = new EventQueue();
  qobj.q = q;
  qobj = class(qobj, 'eventq');

function [e] = empty(qobj)
  q = qobj.q;
  # int e = q->EventQueue.empty();
```

Example: SWIG interface file

The SWIG input:

```
%module ccube
%{
extern int cube( int n );
%}
int cube(int n);
```

Example usage from Python:

```
import ccube
print "Output (10^3): ", ccube.cube(10)
```

Is that it?

```
INC= /Library/Frameworks/Python.framework/Headers

example.o: example.c
	gcc -c $<

example_wrap.c: example.i
	swig -python example.i

example_wrap.o: example_wrap.c
	gcc -c -I$(INC) $<

_example.so: example.o example_wrap.o
	ld -bundle -flat_namespace \
	   -undefined suppress -o $@ $^
```

This is a Makefile from my laptop. Must be a better way!

A better build?

```python
#! /usr/bin/env python
# setup.py

from distutils.core import *
from distutils      import sysconfig

_example = Extension( "_example",
                      ["example.i","example.c"])

setup( name        = "cube function",
       description = "cubes an integer",
       author      = "David Bindel",
       version     = "1.0",
       ext_modules = [_example] )
```

Run `python setup.py build` to build.

The build problem

Actually figuring out how to build the code is hard!

- Hard to figure out libraries, link lines
- Gets harder when multiple machines are involved
- Several partial solutions
  - `pkg-config` is great (where it applies)
  - CMake looks promising
  - SCons was a contender
  - Python distutils helps
  - autotools are the old chestnut
  - RTFM (when all else fails?)
- Getting things to play nice is basis of some businesses!

This is another reason to seek a large user community.

Brute-forcing the build problem

Idea:

- Figure out how to build on one Linux setup
- Capture key dependencies
- Encode in a virtual machine or Docker container

Pro: Everyone uses the same libraries, etc.
Con: How does a generic build use specialized HW?

I suspect this will be how I teach this stuff in a couple years...

Mixed languages: tech issues

Primary technical pain points for mixed language code:

- Cross-language communication (wrapper generators help)
- Building and deployment (CMake and Docker help)
- Debugging and run-time issues (knowing what you’re doing helps)

Mixed languages: social issues

Primary sociological issues for (mixed) language adoption:

- Availability of learning resources and documentation
- Availability of support tools and libraries
- Active user base
- Open-ness of code base?
- Longevity

A concluding thought

I don't know what the programming language of the year 2000 will look like, but I know it will be called FORTRAN.

— C.A.R. Hoare