Lecture 12: Tools

David Bindel

6 Oct 2011

Logistics

- `crocus`
  - Martin is installing stuff – but not yet complete
  - Batch queue is acting funny – looking into it
- HW 2
  - Harder for many of you than expected!
  - And timings will be off if things get loaded...
  - Revised due date: next Wednesday at 11:59
- Project 2
  - Parallel smoothed particle hydrodynamics code
  - *Will* be posted by Tuesday

Today: Tools

- Timing and profiling
- Code generation (in passing)
- Scripting and steering

Timing tools

Manual methods are often useful (and portable):

- Manually insert timers (what we've done so far)
- Manually access performance counters

Tools can help!

- Automatic instrumentation
- Sampling profilers
- Processor simulation

Timing troubles

- Manual instrumentation is a pain
- *Any* instrumentation may disrupt optimization
- Frequent sampling disrupts performance
- Infrequent sampling misses details (without lots of data)
- Hard to attribute hot spots in optimized code
- Big runs may generate a *lot* of timing data

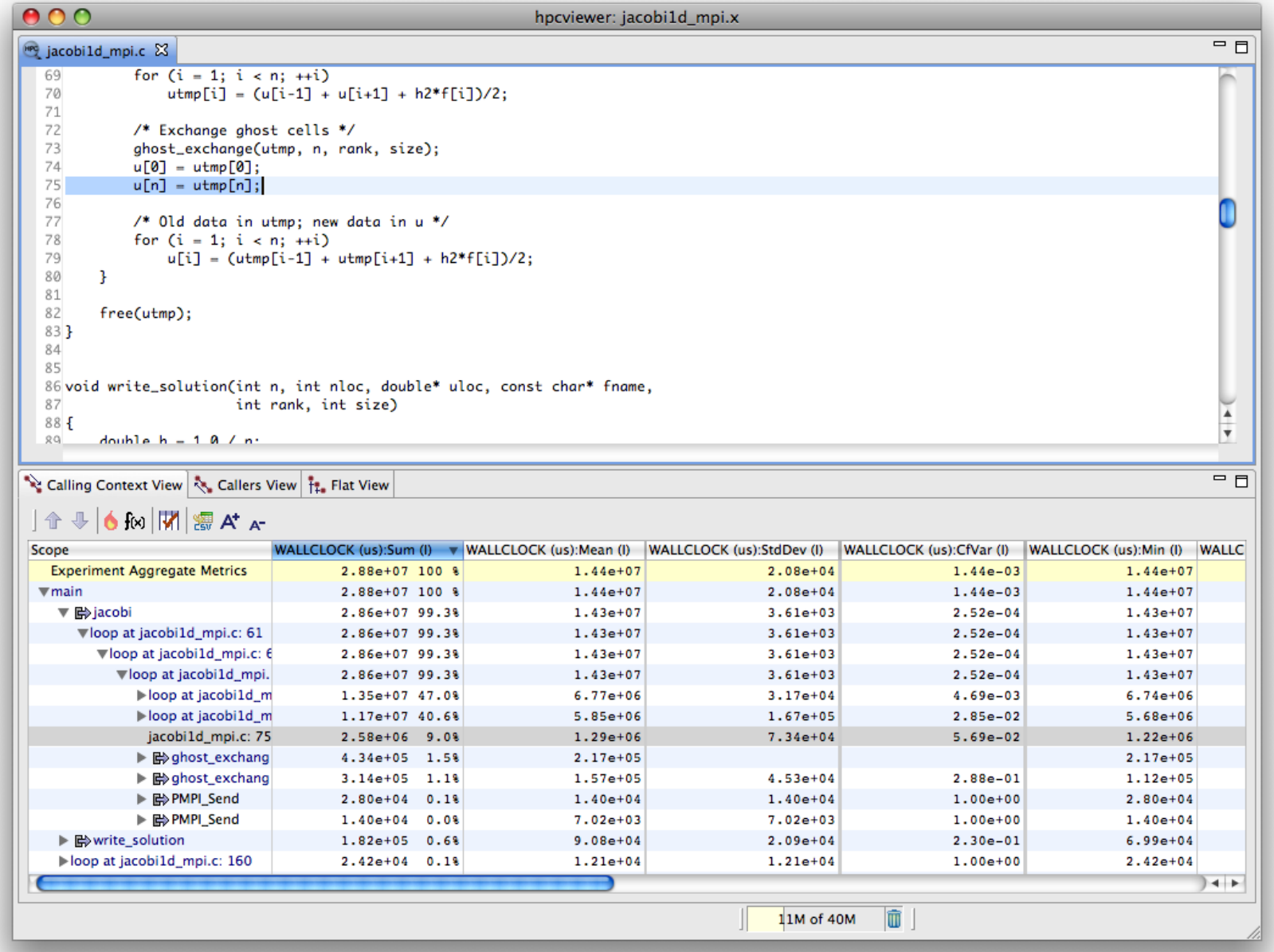
Profilers

I've asked for two profilers on the cluster: HPCToolkit and TAU.

- `http://hpctoolkit.org/`
- `http://www.cs.uoregon.edu/Research/tau/home.php`

And on my laptop, I use Shark.

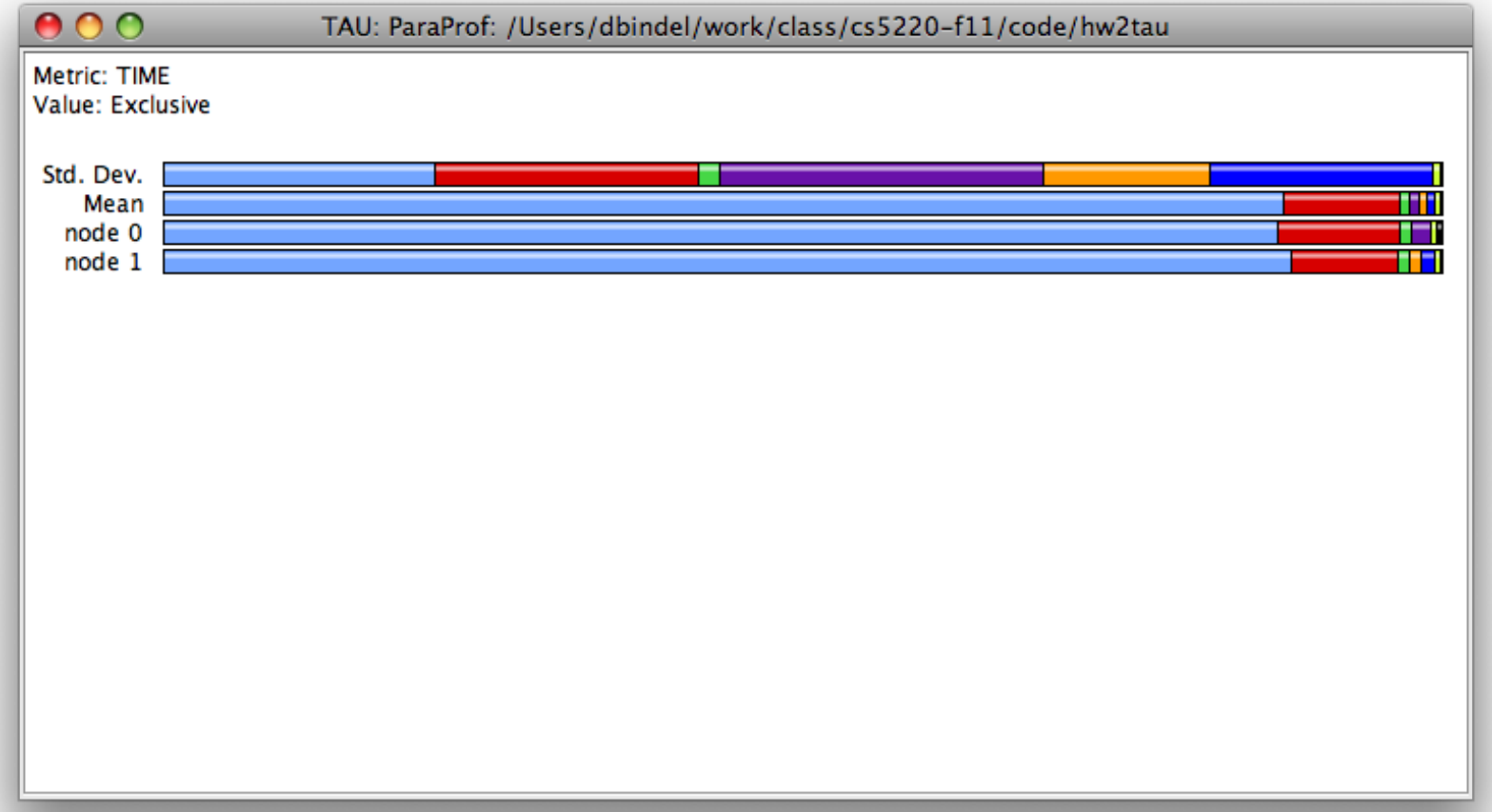
HPCToolkit


hpcviewer: jacobi1d_mpi.x
jacobi1d_mpi.c
69 for (i = 1; i < n; ++i)
70 utmp[i] = (u[i-1] + u[i+1] + h2*f[i])/2;
71
72 /* Exchange ghost cells */
73 ghost_exchange(utmp, n, rank, size);
74 u[0] = utmp[0];
75 u[n] = utmp[n];
76
77 /* Old data in utmp; new data in u */
78 for (i = 1; i < n; ++i)
79 u[i] = (utmp[i-1] + utmp[i+1] + h2*f[i])/2;
80 }
81
82 free(utmp);
83 }
84
85
86 void write_solution(int n, int nloc, double* uloc, const char* fname,
87 int rank, int size)
88 {
Calling Context View
Callers View
Flat View
Scope
WALLCLOCK (us):Sum (I)
WALLCLOCK (us):Mean (I)
WALLCLOCK (us):StdDev (I)
WALLCLOCK (us):CfVar (I)
WALLCLOCK (us):Min (I)
Experiment Aggregate Metrics 2.88e+07 100 % 1.44e+07 2.08e+04 1.44e-03 1.44e+07
main 2.88e+07 100 % 1.44e+07 2.08e+04 1.44e-03 1.44e+07
jacobi 2.86e+07 99.3% 1.43e+07 3.61e+03 2.52e-04 1.43e+07
loop at jacobi1d_mpi.c: 61 2.86e+07 99.3% 1.43e+07 3.61e+03 2.52e-04 1.43e+07
loop at jacobi1d_mpi.c: 6 2.86e+07 99.3% 1.43e+07 3.61e+03 2.52e-04 1.43e+07
loop at jacobi1d_mpi. 2.86e+07 99.3% 1.43e+07 3.61e+03 2.52e-04 1.43e+07
loop at jacobi1d_m 1.35e+07 47.0% 6.77e+06 3.17e+04 4.69e-03 6.74e+06
loop at jacobi1d_m 1.17e+07 40.6% 5.85e+06 1.67e+05 2.85e-02 5.68e+06
jacobi1d_mpi.c: 75 2.58e+06 9.0% 1.29e+06 7.34e+04 5.69e-02 1.22e+06
ghost_exchang 4.34e+05 1.5% 2.17e+05 2.17e+05
ghost_exchang 3.14e+05 1.1% 1.57e+05 4.53e+04 2.88e-01 1.12e+05
PMPI_Send 2.80e+04 0.1% 1.40e+04 1.40e+04 1.00e+00 2.80e+04
PMPI_Send 1.40e+04 0.0% 7.02e+03 7.02e+03 1.00e+00 1.40e+04
write_solution 1.82e+05 0.6% 9.08e+04 2.09e+04 2.30e-01 6.99e+04
loop at jacobi1d_mpi.c: 160 2.42e+04 0.1% 1.21e+04 1.21e+04 1.00e+00 2.42e+04
11M of 40M

HPCToolkit

- Sampling-based profiler (performance counters via PAPI)
- Profiler only runs on Linux; viewer on Linux, Mac, Windows
- Basic steps (see QuickStart section of manual)
  - Compile with symbol information `-g`
  - Analyze code structure with `hpcstruct`
  - Run the code with `hpcrun`
  - Analyze the database(s) with `hpcprof`
  - View the results with `hpcviewer`

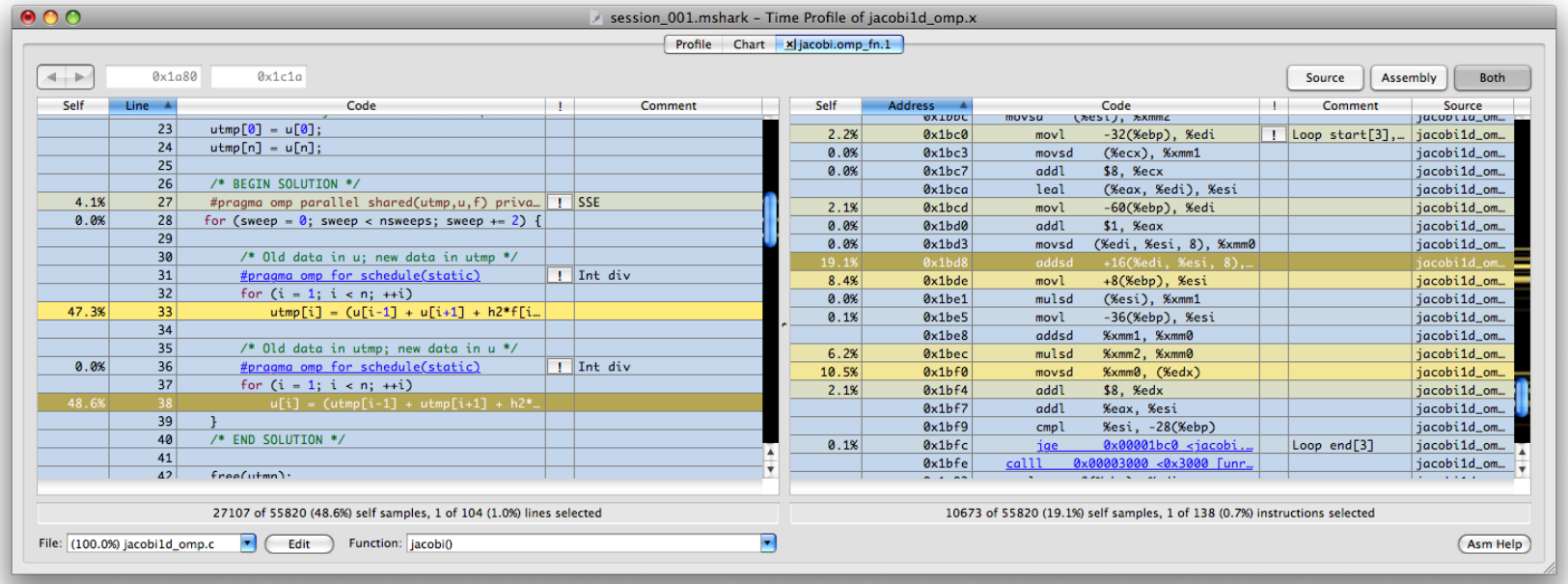
TAU


TAU: ParaProf: /Users/dbindel/work/class/cs5220-f11/code/hw2tau
Metric: TIME
Value: Exclusive
Std. Dev.
Mean
node 0
node 1

TAU

- Instrument code (static or dynamic) for
  - Profiling
  - Profiling with hardware counters
  - Tracing
- Basic steps (see tutorial slides on TAU page)
  - Set up some environment variables
  - Compile (`tau_cc.sh` for static instrumentation)
  - Run (`tau_exec` for dynamic instrumentation)
  - View results with `pprof` (text) or `paraprof` (GUI)

Shark


session_001.mshark – Time Profile of jacobi1d_omp.x
Profile
Chart
jacobi.omp_fn.1
0x1a80
0x1c1a
Source
Assembly
Both
Self
Line
Code
!
Comment
23 utmp[0] = u[0];
24 utmp[n] = u[n];
25
26 /* BEGIN SOLUTION */
4.1% 27 #pragma omp parallel shared(utmp,u,f) priva… ! SSE
0.0% 28 for (sweep = 0; sweep < nsweeps; sweep += 2) {
29
30 /* Old data in u; new data in utmp */
31 #pragma omp for schedule(static) ! Int div
32 for (i = 1; i < n; ++i)
47.3% 33 utmp[i] = (u[i-1] + u[i+1] + h2*f[i…
34
35 /* Old data in utmp; new data in u */
0.0% 36 #pragma omp for schedule(static) ! Int div
37 for (i = 1; i < n; ++i)
48.6% 38 u[i] = (utmp[i-1] + utmp[i+1] + h2*…
39 }
40 /* END SOLUTION */
41
Self
Address
Code
!
Comment
Source
2.2% 0x1bc0 movl -32(%ebp), %edi ! Loop start[3],… jacobi1d_om…
0.0% 0x1bc3 movsd (%ecx), %xmm1 jacobi1d_om…
0.0% 0x1bc7 addl $8, %ecx jacobi1d_om…
0x1bca leal (%eax, %edi), %esi jacobi1d_om…
2.1% 0x1bcd movl -60(%ebp), %edi jacobi1d_om…
0.0% 0x1bd0 addl $1, %eax jacobi1d_om…
0.0% 0x1bd3 movsd (%edi, %esi, 8), %xmm0 jacobi1d_om…
19.1% 0x1bd8 addsd +16(%edi, %esi, 8),… jacobi1d_om…
8.4% 0x1bde movl +8(%ebp), %esi jacobi1d_om…
0.0% 0x1be1 mulsd (%esi), %xmm1 jacobi1d_om…
0.1% 0x1be5 movl -36(%ebp), %esi jacobi1d_om…
0x1be8 addsd %xmm1, %xmm0 jacobi1d_om…
6.2% 0x1bec mulsd %xmm2, %xmm0 jacobi1d_om…
10.5% 0x1bf0 movsd %xmm0, (%edx) jacobi1d_om…
2.1% 0x1bf4 addl $8, %edx jacobi1d_om…
0x1bf7 addl %eax, %esi jacobi1d_om…
0x1bf9 cmpl %esi, -28(%ebp) jacobi1d_om…
0.1% 0x1bfc jge 0x00001bc0 <jacobi… Loop end[3] jacobi1d_om…
0x1bfe calll 0x00003000 <0x3000 [unr… jacobi1d_om…
27107 of 55820 (48.6%) self samples, 1 of 104 (1.0%) lines selected
10673 of 55820 (19.1%) self samples, 1 of 138 (0.7%) instructions selected
File: (100.0%) jacobi1d_omp.c
Edit
Function: jacobi()
Asm Help

Shark

- Sampling-based profiler (with perf counters?)
- Profiler only runs on Mac
- Basic steps
  - Compile with symbol information `-g`
  - Run code with shark `shark -i ./a.out`
  - Analyze the results with the Shark GUI

Debugging tools

This is not so easy:

- What if the code is non-interactive (batch queueing)?
- How can we make tools implementation-neutral?

I'm still a cave man: `printf` and `gdb`.
Or I debug in a scripting language interface.

Code generation tools

Some tools will help write specialized code:

- Single-purpose auto-tuners (ATLAS)
  - Tries many alternate organizations fast
  - You can write these yourself, too!
- Mathematical generators (Mathematica, matexpr, ADIC)
  - Automatically translate matrix expressions into C
  - Automatic differentiation and symbolic manipulation
  - Warning: the computer doesn't do error analysis!
- Wrapper generators
  - Automate cross-language bindings
  - More about these shortly

Scripting tools

Outline:

- Scripting sales pitch + typical uses in scientific code
- Truth in advertising
- Cross-language communication mechanisms
- Tool support
- Some simple examples

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

UMFPACK solve in C:

```
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);
```

UMFPACK solve in MATLAB:

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

Which would *you* rather write?

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. This is effectively what we're doing with our qsub scripts and Makefiles.

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?

```
#! /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
  - 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.

Some simple examples

- nbody example
- Monte Carlo example with MPI
- CG example using Pysparse

Now, to the terminal!