Lecture 19: Tuesday, Apr 8
High-level languages and Julia

High productivity vs. high performance

Scientific computing is about getting answers that are right enough and fast enough for some task. Often, that task is advancing our understanding; or, as Hamming put it in his book on numerical computing:

> The purpose of computing is insight, not numbers.

It's useful to keep this perspective when thinking about performance. Getting useful insights generally involves thought and experimentation, with humans participating in the process. High-performance codes are useful to the extent that they allow us to meet hard or soft time budgets imposed by interactions with the real world or by human impatience. But there is often a tension between performance, development speed, generality, and ease of understanding. While it's incredibly useful to learn a mental model that lets us write code that *can* be optimized, that does not mean that we always *should* optimize. Quoting again – this time from Donald Knuth:

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Part of the reason we no longer require all students to study C/C++ or Fortran programming is that other, higher-level languages are fast enough for many purposes, and certainly for the purpose of setting up an initial prototype or a numerical experiment. For matrix-intensive codes, or for other codes that achieve high performance based on common computational kernels, a prototype written in MATLAB or Python that implicitly uses fast libraries can even be faster than a prototype written in a low-level language that does not use such libraries. And when a library bindings are available in Python or MATLAB, they are often easier to use than the corresponding C bindings. After all, which would you rather write to solve a sparse system of equations:

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

or

```
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);
```

High-level languages have many attractive features: concise code, expressive notation, interactive environments, and rich sets of libraries. Refusing to use such tools because they don't immediately provide the raw speed of carefully-tuned C/C++ or Fortran is misguided at best, and snobbery at worse. Of course, efficiency *is* an important consideration some of the time, and refusing to use a low-level language when appropriate can be just as misguided. Much of the time, the right approach is to use more than one tool. In particular, typical simulations might involve:

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

While the solvers and visualization may be performance critical, it's often fine to emphasize ease of use when it comes to problem descriptions, specification of tolerances, or high-level logic gluing everything together.

Mixed-language programming

Often, it's helpful to consider more than one language. A scripting interface is particularly helpful 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.

And there are many scripting languages from which you can choose! Think MATLAB, LISPs, Lua, Ruby, Python, Perl, etc. There are also many choices of low-level languages, of course, though C/C++, Fortran, and Java are perhaps the most obvious ones at this time.

Language choices

Apart from ease of use or performance, there are several factors worth considering when choosing a mix of languages for a given project:

- What languages are preferred by the relevant communities? This matters both because it affects both how much impact your code will have if you distribute it and because it affects how likely you are to find a sympathetic soul to answer questions when you run into trouble. For scientific computing work, Python and MATLAB are clear favorites at the high level, C/C++ and Fortran at the low level.

- What libraries are available? Are there libraries that make the task at hand easy? Often, library interfaces are themselves really small domain-specific languages; as the old Bell Labs proverb goes, "library design is language design."[1] In some cases, the question is not really "what languages should I use?", but "what libraries should I use?"

- If you use more than one language, how well do the different languages inter-operate? Is it easy to move data or control from one to the other?

- How easy is it to build and deploy libraries written in your languages?

Logistics of mixed-language programming

There are two main barriers to mixed-language programming: issues in cross-language communication, and issues in building and deployment. These barriers go together: less powerful interface techniques are often easier to build and deploy. We will broadly map the landscape in terms of several common patterns.

[1] I've seen this in many places, but looked it up in Stroustroup.

File communication

Perhaps the simplest way for mixed-language programs to communicate is through a shared file system. For example, many scientific computations can be separated into pre-processing, analysis, and post-processing phases. The pre-processor generates input files used by the analysis code; the post-processor takes output from the analysis code and generates useful statistics or visualizations. Often, it makes sense to use different languages for the pre-processor, post-processor, and analysis engines.

The file interface is portable, and it provides a clean separation between the different processes. It is also rather limited: bulk communication via the file system is non-interactive and can be quite slow.

Inter-process communication

Programs written in different languages can use inter-process communication protocols to provide a more dynamic interface. Perhaps the simplest case is UNIX pipes, which tie the standard input or output of one program to the standard input or output of another. Two-way interactions are often better handled by message passing protocols that sit on top of a low-level socket (TCP sockets are great for portability; Unix-domain sockets are better for speed). This is how many GUIs are build: a front-end written in some scripting language uses sockets to communicate with a back-end that does the actual computation.

General inter-process communication mechanisms provide a clean separation between the communicating processes, and are less limited than file system communication. However, two-way inter-process communication can potentially bring in the same complexities as any other message-passing system, like deadlock or failure of one of the communicating processes. There are also portability issues with many IPC mechanisms, particularly if one wants to run in both Windows and Unix variants, though mechanisms built on TCP sockets are available everywhere.

Remote procedure calls

Remote procedure call (RPC) libraries reduce the complexity of IPC somewhat by hiding it behind a procedure call abstraction. In a typical use case, a client (which might be written in a high-level language) would make requests of a computational server (possibly written in a lower-level language). Usually, some mini-language is used to specify the function call interface,

and a specialized compiler generates code at the client and server that manages the behind-the-scenes message passing protocol. This is how a lot of web services work (JavaScript in the browser invokes remote procedure calls on the server via SOAP). This is also the idea behind CORBA, COM, and various other acronym-laden systems. There has been some work on variants of RPC for scientific computing.

It's important to realize that RPC is *not* identical to ordinary function calls, for several reasons:

- Communication involves message passing behind the scense. The server cannot read the client's data structures unless they were sent across the wire.
- Remote procedure calls have much higher overhead than local procedure calls.
- The failure scenarios for RPC are much more complex than those for a local procedure call. Either the client or the server could potentially fail in the middle of a call, and the reason might not be obvious at the other end.

Cross-language calls

When most people think of mixed-language programming, they think of this mechanism: the different interpreters and application libraries are all linked together into the same executable, and can communicate using "ordinary" function calls and shared data structures. The easiest case is typically when a scripting language calls into a library in a compiled language such as C/C++ or Fortran.

A lot of book-keeping goes into these cross-language calls:

- Marshaling of argument data (translation+packaging);
- Function lookup, including any name-mangling;
- Function invocation;
- Translation of return data;
- Translation of exceptional conditions;
- Possibly some consistency checks, other book keeping.

For some types of calls (e.g. from a scripting language into C/C++/Fortran), there are specialized tools that automate this book-keeping. *Wrapper generators* are specialized compilers that generate binding code from a formal interface description (e.g. SWIG). Alternatively, there are sometimes *language extensions* (e.g. Cython) that allow a scripting language to call into compiled code in a more organic way.

Often, the hard part of mixing languages in this manner involves actually building the code. For special cases, there are tools like Python's `distutils` that simplify the process of figuring out what libraries need to be included. However, even with a system like `distutils`, and even with standard choices of languages, it may be necessary to understand what support libraries are used by all the languages involved. For less standard language combinations, there is less support for keeping track of all the behind-the-scenes book-keeping that goes into the programming environment provided by each of the languages individually.

If you are inclined to write mixed-language code in this way, my advice is to give everything a C interface. Most languages provide interfaces to C if they interface to anything.

Julia

Mixed-language programming has been one of my favorite hammers for a long time. Unfortunately, I've written some codes in this style that resemble Frankenstein's monster: I shout "it's alive!", everything works great for a while, and then at some point it goes berserk and starts destroying things. Or I move on and nobody can get it to build again – either way is sub-optimal.

Julia is a new language for technical computing that is meant for both high-level, productivity-oriented computing and for high performance. It combines nice features from some of my favorite languages: MATLAB, Python, Common Lisp, C++, etc. It has garbage collection, macros, dynamic typing (with type specialization for performance), and multiple dispatch. It is interactive, but provides a JIT for performance. It has a liberal license (unlike MATLAB). And it was designed, in part, to avoid the problem of Frankenstein programs.

The Python ecosystem is more mature, and is likely to remain my "home base" scripting language for the immediate future (along with MATLAB). But I find a lot in Julia to be excited about. The social dynamics are right: the language has a distributed group of core developers who really care

about the system, rather than a single corporation or a researcher who might develop the code for a while and then move on or be bought out (the fate of the Star-P system which was one of the direct predecessors of Julia). And there is already a very active user community that is both building packages and asking and answering lots of questions. The core developers keep up-to-date builds for several platforms, and at this point the packages are available for several standard distribution systems (e.g. Homebrew, MacPorts, and Ubuntu). I don't think this will be a flash in the pan, and it has been designed from the outset with performance in mind. Also, the language features a built-in mechanism for calling C functions, so I can always fall back to a mixed-language approach as needed.

Julia resources

For an overview:

https://github.com/ViralBShah/julia-presentations/raw/master/Fifth-Elephant-2013/Fifth-Elephant-2013.pdf

This is the slide set that I used in class. For tutorials and teaching resources:

http://julialang.org/teaching/

And in general, the documentation is pretty good; see

http://julialang.org

For profiling, there are two options: the sampling profiler in the standard library, and an instrumenting profiler. See

- http://julia.readthedocs.org/en/latest/stdlib/profile/
- https://github.com/timholy/IProfile.jl

For performance tips, see

- http://docs.julialang.org/en/latest/manual/performance-tips/
- http://julialang.org/blog/2013/09/fast-numeric/