Lecture 14: Thursday, Mar 13
Tools

Logistics

- Given the problems with C4, we will allow submissions until midnight Friday without penalty. Unless you have made prior arrangements with me, you should finish by then. It's fine if your timings are imperfect or incomplete (though not if it's because your code always crashes!)
- At present, we appear to be able to submit jobs on C4 on a single machine (using `ompsub`), but parallel universe jobs that can span machines are just sitting in the queue. You may want to use the `ompsub` script if you have not yet finished your timing runs; see Piazza for details.
- You can check job status with `condor_q`, and see why a job is not getting scheduled with `condor_q -analyze XXXX`. If you have a job that cannot possibly be scheduled, or a job that ran and crashed without clearing the queue, please remove it (`condor_rm XXXX`).
- There is now a CMS slot ready for project proposals. You should submit a project proposal by March 28. At this point, a brief description and a list of project partners will suffice. You may choose different partners, of course. We're looking for groups of 1–5, with 2–3 being an ideal size. If you work in a group of four or five, your project goals should be more ambitious than those of a group of one or two.

Tool support for HPC programming

In broad generalities, what makes a good program? Ideally, we like codes that are:

- Readable, documented, and easy to maintain;
- Flexible and modular
- Correct
- Efficient in terms of time and resource utilization

These are attractive qualities not only in high-performance scientific code, but in any other code as well. Hence, it's not too surprising that there are mature tool ecosystems that address each of these points, and these tools are our main topic for today. While some of the discussion in this lecture is specific to HPC development, most of the high-level points apply to other types of development work, too.

Our plan for the remainder of this lecture is to discuss several different classes of tools, focusing most of our energy on tools that allow us to reason about performance and correctness of parallel codes.

Documentation

The simplest form of documentation is simply comments that go with the code (along with separate manuals and design documents). For those who want something fancier than plain text comments, there are several language-specific (or language-family-specific) documentation tools that are more-or-less standard. For C/C++ and related languages, the standard choice is Doxygen; for Python, the standard choice is Sphinx; and for Java, the standard choice is Javadoc. Each of these tools extract documentation from structured comments or docstrings written with the code, and use that as the basis of nicely formatted and hyperlinked documents.

The task of extracting comments from a file is not that difficult, and so there are also a plethora of smaller tools based on the same idea; I'm guilty of home-brewing a couple of these myself. But writing readable, well-documented code remains difficult. The hard part is not typesetting the documentation or drawing dependency diagrams, but in writing clear and useful text and code. Alas, our tools are not so good at writing text for us, at least so far.

Though *literate programming* has never become really widespread, it's useful to mention the concept in the context of documentation tools. As imagined by Knuth, a literate program is a document designed to be read by people, and secondarily executed by a computer. A source "web" file containing both code and text with typesetting directives can thus be run through a postprocessor to produce either a typeset document or a regular code file. One of my favorite old C books, *C Interfaces and Implementations* by Hansen, is a nice illustration of this approach. But while I have tried this approach in my own projects in the past, but ultimately decided that I prefer the document-in-comments approach. When I want more significant documentation than the normal interface descriptions, I can incorporate it with comment processing tools, and the commented files play nicely with

my normal code development environments.

Flexibility and modularity

Just as writing well-documented code is mostly difficult because writing is hard, writing flexible and modular codes is mostly difficult because good design is hard. That said, there are some ways that good tools can support a modular design.

Scripting is one of my favorite approaches to tying together small modules in a flexible way. Environments like MATLAB and Python are significantly terser and more user-friendly than C; and if used properly, they can be used in very high-performance systems without compromising performance. Little languages are also helpful for configuration and parameter setting, as with the Lua interpreter incorporated into our HW 2 code. Integrating scripting languages with low-level C routines is made somewhat easier by systems like Cython and SWIG (both of which are available on C4).

There is a serious, though often overlooked, cost to writing codes that depend on a variety of libraries and scripting languages. Building on the work of others ultimately saves a lot of time, but it can produce real pain if the build system doesn't provide adequate support. For resolving build dependencies in a reasonably platform-independent way, I strongly recommend looking at tools like CMake. Though CMake is a bit quirky, it is starting to see broad adoption in the HPC community, and it is arguably far easier to read and maintain CMake build systems than systems based on autoconf.

There are other ways that good tools support modularity, including automatic generation of communication protocols between interacting system components. This is, however, a topic for another time.

Correctness

Broadly speaking, there are two classes of analysis tools to check correctness of a piece of code. Static analysis tools attempt to reason about all possible execution paths through a program in order to check correctness properties. Such tools have a long academic history, and basic static analysis is a mainstay of compiler technology. In fact, perhaps the simplest way to use static analysis to help catch errors is to just turn on all compiler warning (`-Wall`)! More recently, however, more serious static analysis methods have more recently been incorporated into some more widely-available tools.

On the C4 cluster, we have both the Clang static analyzer and Intel Inspector XE for static analysis (though, as has been mentioned before, Clang does not yet support OpenMP in the main branch). There are also static analysis tools to check MPI communication, such as MPI-Spin, but these are mostly still research projects.

Dynamic analysis tools are more widely used. Unlike static analysis tools that consider all possible execution paths, dynamic analysis tools consider an execution path that is actually taken. These types of analysis tools may rely on passive monitoring of the code (e.g. by intercepting system calls); or on static (compile-time) or dynamic (run-time) instrumentation; or on simulation of program behavior in a virtualized environment. Dynamic analysis tools can be particularly useful for testing errors related to memory access and for checking for possible deadlocks or data races. On C4, Intel Inspector supports some dynamic analysis along with the static analysis; or you can try out the Valgrind tool suite, including a memory checker (memcheck) and several data race detection tools (`helgrind, DRD, ThreadSanitizer`). Note that the data race detectors are generally familiar with pthreads, but may not know how to interpret calls to other synchronization libraries. For example, the default GNU OpenMP library on Linux uses a kernel-supported fast user-space mutex that helgrind doesn't realize is a synchronization primitive. If needed, one can recompile GCC to make the OpenMP library more valgrind-friendly.

Though there are several dynamic analysis tools to check the correctness of threaded codes, there are fewer tools that focus on MPI. One example of a dynamic MPI analysis code is the Marmot tool. As with MPI-Spin, though, this is primarily a research system.

Performance

There are two main classes of performance measurement experiments:

- *Profiling* involves collecting aggregated statistics: number of times each routine is called, total time spent in each routine, total cache misses in a routine, etc.
- *Tracing* involves logging individual program events and the time at which they occurred.

Tracing is potentially more informative than profiling, but it also can involve a lot more data.

Performance measurement tools can get information about programs by instrumentation (static or dynamic), sampling, or reading hardware performance counters. There are standard hooks for profiling instrumentation for MPI (the PMPI interface) and for OpenMP (POMP). There is also a widely used API for accessing performance counters, called PAPI.

Even for profiling, instrumentation is often turned on only for specific parts of the program execution, specified either by a part of the code or by a time interval during the execution. Many performance measurement tools also have facilities that allow the user to define named regions in order to produce more informative reports.

We have several profiling tools installed on the C4 cluster:

- Intel Analyzer XE / VTune is a very nice profiling and performance analysis system
- `gprof` is an old standby, but it does not work well with parallel code.
- `gperftools` is a lightweight sampling profiler that came out of Google. To use it, load the `gperftools` module.
- IPM (Integrated Performance Monitoring) is a portable low-overhead profiling infrastructure for parallel codes.
- TAU (Tuning and Analysis Utilities) is a long-running project out of U. Oregon that provides a very sophisticated framework for tracing and profiling. It supports static and dynamic instrumentation, sampling, and use of performance counters. It also is associated with at least three separate visualization tools (Paraprof, PerfExplorer, and Jumpshot).
- mpiP is a lightweight MPI profiling tool.

In the class subdirectory, under the basic examples, the `mpi_ping_tau` and `mpi_ping_ipm` subdirectories include Makefiles and documentation that illustrate how to use the TAU and IPM profilers.

Beyond performance monitoring tools, there are tools for simulating the behavior of the processor (the cachegrind tool in valgrind is an example), tools for auto-tuning, and several other tools that I tend to think of as useful for optimizing the performance of small kernels.

1 Representative use cases

It usually makes sense to maintain two distinct sets of use cases for parallel codes. You want test cases that check the proper behavior of your numerical methods in a variety of scenarios; and you want timing cases that serve as a representative computation for doing profiling, tracing, and timing studies. These are frequently *not* the same thing. You want test cases that probe the corners of your code that you don't always get to; it's less interesting to time how long it takes to deal with a peculiar corner case that almost never happens. Slow performance in rare cases can be amortized over job runs; loss of correctness in these rare cases cannot be amortized.

For testing, it's useful to have some automated tests so that you can catch issues in your code incrementally. I like to write my Makefiles to include a phony `test` target, so that typing `make test` produces an error message if I've broken something (and no output otherwise). I sometimes also automatically run very small test cases with `valgrind` to catch memory errors or other correctness issues that might not be obvious from the output alone. You may also want to consider frameworks like `CUnit`.

For timing cases, you may want to consider benchmark cases that are meant to illuminate some particular aspect of performance; and realistic use cases that allow you to look at performance more holistically. For either type of use case, it's helpful to find problems where it makes sense to scale the size up or down, so that you can do weak scaling studies.

2 Caveman tools

In addition to the more sophisticated tools mentioned above, it's worth keeping in mind that there are some simple tools that you can access from about anywhere without installing anything:

- Compile with `-Wall` (all warnings on) to get the benefits of at least a little static analysis. This won't give you the same level of analysis as Intel Inspector and company, but it's certainly a starting point.
- Use `assert` commands to check preservation of invariants at run time.
- Use `printf` for debugging (even in MPI).
- Use `omp_get_wtime()` or `MPI_Wtime()` for timing. Note that these routines return wallclock time at one processor; in general, clock skew

across processors makes it a bad idea to compare too closely the wall-clock times retrieved on different threads or MPI processes. It's also worth knowing that computer systems have different notions of the time taken by a process: wallclock time is the total elapsed time, but there is also processor time (returned by `clock`) which typically does not include time when the job is waiting on I/O or is not resident on the CPU for some other reason.