Lecture 27: Tools, trends, and concluding thoughts

David Bindel

3 May 2010

Some take-aways

- Knowledge of some programming models
  (message passing, threads)
- A little computer architecture
  (memory and communication costs)
- Some back-of-the-envelope performance modeling
- A few numerical organizational ideas
  (sparsity, blocking, multilevel)
- Appreciation for a few tools and libraries!

Numerical ideas

... thinking about high-performance numerics often involves:

- Tiling and blocking algorithms; building atop the BLAS
- Ideas of sparsity and locality
- Graph partitioning and communication / computation ratios
- Information propagation, deferred communication, ghost cells
- Big picture view of sparse and direct iterative solvers
- Some multilevel ideas
- And a few other numerical methods (FMM, FFT, MC, MD) and associated programming patterns

Improving performance

- Zeroth steps
  - Working code (and test cases) first
  - Be smart about trading your time for CPU time!
- First steps
  - Use good compilers (if you have access – Intel is good)
  - Use flags intelligently (`-O3`, maybe others)
  - Use libraries someone else has tuned!
- Second steps
  - Use a profiler (Shark, gprof, Google profiling library)
  - Learn some timing routines (system-dependent)
  - Find the bottleneck!
- Third steps
  - Tune the data layout (and algorithms) for cache locality
  - Put in context of computer architecture
  - *Now* tune
    - Maybe with some automation (Spiral, FLAME, ATLAS, OSKI)

Parallel environments

- MPI
  - Portable to many implementations
  - Giant legacy code base
  - Largely lowest common denominator for mid-80s
- OpenMP
  - Parallelize C, Fortran codes with simple changes
  - ... but may need more invasive changes to go fast
- Cilk++ (now Intel), Intel Thread Building Blocks, ...
  - Threading alternatives to OpenMP
- CUDA, OpenCL, Intel Ct (?), etc
  - Highly data-parallel kernels (e.g. for GPU)
- GAS systems: HPF, UPC, Titanium, X10
  - Shared-memory-like programs
  - Explicitly acknowledge of different types of memory

Libraries and frameworks

- Dense LA: LAPACK and BLAS (ATLAS, Goto, Veclib, MKL, AMD Performance Library)
- Sparse direct: Pardiso (in MKL), UMFPACK (in MATLAB), WSMP, SuperLU, TAUCS, DSCPACK, MUMPS, ...
- FFTs: FFTW
- Graph partitioning: METIS, ParMETIS, SCOTCH, Zoltan, ...
- Other; deal.ii (FEM), SUNDIALS (ODEs/DAEs), SLICOT (control), Triangle (meshing), ...
- Frameworks: PETSc/Trilinos
  - Gigantic, a pain to compile... but does a lot
  - Good starting places for ideas, library bindings!
- Collections: Netlib (classic numerical software), ACTS (reviews of parallel code)
- MATLAB, Enthought's Python distro, Star-P, etc. add value in part by selecting and pre-building interoperable libraries

UNIX programming

... because we're still using UNIX (Linux, OS X, etc), it's helpful to know about:

- Make and successors (autoconf, CMake)
- A little shell (see Advanced Bash Programming Guide)
- A few tools (cat/grep/find/which/...)
- A few little languages (Perl, awk, ...)

Scripting

... because we don't want to spend all our lives debugging C memory errors, it helps to make judicious use of other languages:

- Many options: Python, Ruby, Lua, ...
- Wrappers help: SWIG, tolua, Boost/Python, Cython, etc.
- Scripts are great for
  - Prototyping
  - Problem setup
  - High-level logic
  - User interfaces
  - Testing frameworks
  - Program generation tasks
  - ...
- Worry about performance at the bottlenecks!

Development environments

Whether in Unix or Windows, it helps to know how to use...

- An editor or IDE (emacs or vi? or something more modern?)
- A compiler (i.e. know what stages you actually go through)
- A debugger (gdb, ddd, Xcode debugger, MSVC debugger)
- Valgrind, Electric Fence, gaurd malloc, or other memory debugging tools
- The C `assert` macros
- Source control (git, mercurial, subversion, CVS)
- Documentation tools (Doxygen, Javadoc, some web variant?)

Development ideas

Read! See lecture 9 notes. A few other things to check out:

- "Five recommended practices for computational scientists who write software" (Kelley, Hook, and Sanders in *Computing in Science and Engineering*, 9/09)
- "Barely sufficient software engineering: 10 practices to improve your CSE software" (Heroux and Willenbring)
- "15 years of reproducible research in computational harmonic analysis" (Donoho et al)
  - Daniel Lemire has an interesting rebuttal.

Where we're heading

> *"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"*
>
> *– Seymour Cray*

- Mostly done with scaling up frequency, ILP
- Current hardware: multicore, some manycore (e.g. GPU)
  - Often specialized parallelism — go, chickens!
- Where current hardware lives
  - Often in clusters, maybe "in the cloud"
  - More embedded computing, too!
    - I'm still waiting for MATLAB for the iPhone
- Straight line prediction: double core counts every 18 months
- Real question is still how we'll use these cores!
  - There's a reason why Intel is associated with at least four parallel language technology projects...

Where we're heading

- Many dimensions of "performance"
  1. Time to execute a program or routine
  2. Energy to execute a program or routine (esp. on battery)
  3. Total cost of ownership / computation?
  4. Time to write and debug programs
- Scientific computing has been driven by speed
- Maybe other measures of performance will gain influence?

Concluding thoughts

- Our technology may be very different in the S12 offering!
- Basic principles remain
  - Same numerical ideas (FFT, FMM, Krylov subspaces, etc)
  - Overheads limit parallel performance
  - Communication (with memory or others) has a cost
  - Back-of-the-envelope models can help
  - Timing comes before tuning
  - Basic algorithmic ideas (sparsity, locality) are key

Your turn!

Reminder:

- Wednesday (5/5): *brief* project presentations
  - Tell me (and your fellow students) what you're up to
  - Keep to about 5 minutes – slides or board
  - This is largely for *your* benefit – so don't panic!
- Project reports due by 5/20 at latest
  - *Don't* make me read a ton of code
  - *Don't* ask for an extension (pretty please!)
  - *Do* show speedup plots, timing tables, profile results, models, and anything else that shows you're thinking about performance
  - *Do* tell me how this work might continue given more time