Final thoughts

David Bindel

23 Nov 2015

Logistics

Reminder: My last lecture!

Dec 3 (or 5): GPU lecture

Dec 16: Project reports due

Dec 23: Fall 15 grade deadline

Project Report

Goal: understand performance!

- *Do* give a description of your problem.
- *Do* describe performance analysis, which might include
  - Serial tuning and reorganizations
  - Strong and weak scaling experiment (speedup plots!)
  - Profiling of communication and computation
  - Tuning of parallelism (communication, synchronization, etc)
  - Comparison to analytical models
  - Comparisons between alternate organizations
- *Do* tell me how this work might continue given more time.
- *Don't* make me read a ton of code.
- *Don't* ask for an extension. This is due 12/16.

Recap and Overview

Goals for Scientific Codes

Right enough, fast enough.

Recall: Goals for the Class (from Lecture 1)

- Reason about code performance
  - Many factors: hardware, software, algorithms
  - Want simple, “good enough” models
- Read/judge HPC literature
- Apply model numerical HPC patterns
- Tune existing codes for modern HW
- Apply good software practices

Hardware ideas

These things matter:

- ILP: Pipelining, concurrent execution, and vectorization
- Memory heirarchy and the cost of cache misses
- Communication costs (latency and bandwidth)
- Synchronization overheads

Model ideas

Essentially, all models are wrong, but some are useful.
– George E. P. Box

- Use simple performance models for guidance
- Fit the parameters to empirical experiment

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, 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
  - 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
  - Does keep evolving (e.g. RDMA support)
- OpenMP
  - Parallelize C, Fortran codes with simple changes
  - ... but may need more invasive changes to go fast
- Cilk Plus, Intel Thread Building Blocks, ...
  - Threading alternatives to OpenMP
- CUDA, OpenCL, etc
  - Highly data-parallel kernels (e.g. for GPU)
- GAS systems: Fortran co-arrays, UPC, Titanium, Chapel
  - 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: Elemental, 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, Anaconda Python distro, etc. add value in part by selecting and pre-building interoperable libraries

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, Julia, R, …
- 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 ideas

Read! Among other things:

- "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.
- Best Practices for Scientific Computing (Wilson *et al*)
- Follow-up: Good Enough Practices for Scientific Computing

Looking back, looking forward

Today: Hardware

- My phone is a multicore machine
- Shared memory programming hasn't disappeared
- 128 processors + a terabyte of memory = 1 beefy box
- Accelerators are everywhere
- Caches keep getting more important
- A modest class cluster has nearly 1000 processors
- Getting a significant fraction of peak is hard
- Statistical computations (machine learning) burn lots of cycles

Today: Software

- Lots is still C/C++/Fortran
  - These are evolving languages!
  - Most new languages don't go far...
- Increased emphasis on high-level (e.g. Python)
  - High performance in specific domains
  - Domain-specific specializations
  - JIT and on-the-fly optimization are commonplace
- High productivity matters along with high performances
- We still suffer some "accidental complexities"
  - Think struct-of-arrays vs array-of-structs transformation

Today: Applications

- Still lots of "traditional" HPC computations
  - Large-scale optimization
  - PDE solves
  - Engineering simulation
- Graph applications?
  - Different properties from PDEs
  - Similar applications
- Also lots of stats / ML computations
  - Often more opportunities for parallelism
  - Often more data, less accuracy – I/O becomes the key
  - Lots of work on frameworks for these problems
  - Closer to traditional HPC over time...
- "Big data" and DB ideas
  - Lots of relatively modest computations over lots of data
  - Still rather different community from lots of HPC

Where we're heading

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

- Done with scaling up frequency, pipeline length
- Current hardware: multicore and manycore (GPU and Phi)
  - Often specialized parallelism — go, chickens!
  - We're back to not-so-short vectors
- Where current hardware lives
  - Often in clusters, maybe "in the cloud"
  - More embedded computing, too!
- Straight line prediction: double core counts every 18 months
- Real question is still how we'll use these cores!
- Ever-worse issues: deep memory, communication costs

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
- Other measures of performance also have influence

Where we're heading

- Top 500 has stayed much the same for several years!
- DOE still says "exascale" pretty often
  - And nobody knows how to use it
- Next Xeon Phi: independent board (vs co-processor)
  - How long with the co-processors?
- Cloud vendors still care more about high throughput, but...
  - Accelerated cloud instances a viable path to some HPC
- Languages advance slowly, but
  - New LLVM Fortran is exciting
  - Multidimensional array functionality being considered by ISO/C++ standard committee
  - Other goodies planned for C++17 (better atomics)

Next steps

- Next offering: likely not S18 – S19? S20?
- Between now and then: how to keep the ball rolling?
  - Keep totient a useful *educational* resource?
  - Continue building relevant skills?
- One idea: two (largely student-guided) activities
  - *Software carpentry workshops* (per semester)
  - *Scientific software meetup* (biweekly)
  - Drop me a line if you're interested in either...

Given enough time

- Serious parallel programming in Cilk++, UPC, etc
- Parallel I/O issues
- Code generation and specialization
- Visualization
- "Big data" processing and frameworks
- Kokkos, TBB, other frameworks
- Reproducibility
- Multigrid
- Tree codes
- Particle codes

Your Turn