Lecture 20: Toolkits

David Bindel

8 Nov 2011

Logistics

- Project 3
  - Scaling experiments should vary number of vertices and number of threads / processors
  - It's okay to just use MPI / OpenMP timer routines if you don't like HPCToolkit
  - Give yourself time to get confused – start now!

Project 3 reminder: communication

On the board.

Lessons from Proj 2: Take 3

- Smart algorithms and tuning trump tuning alone
  - Some of you tried the naive code with $h = 0.01$. The cost really does scale quadratically with particle count!
  - Binning did far more for performance than parallelism.
- There is no point in tuning non-bottlenecks
  - The leapfrog updates took effectively no time
  - Re-binning is also negligible compared to interactions

Lessons from Proj 2: Take 3

- OpenMP is not magic
  - Starting/stopping thread times is expensive
  - Synchronization is expensive
  - Easy to slow things down by parallelizing!
  - Helps to be thoroughly aware of dependency structure
- Race conditions are sometimes soul-suckingly subtle
  - May not show up except by accidents of timing
  - Very difficult to detect (note: `valgrind` DRD tool may help)

Lessons from Proj 2: Take 3

- Test and debug incrementally
  - The code and pray method does not scale
  - Automate the process of sanity checking your computations!
- Pointer chasing (and index chasing) are performance drags
  - Pays to have data locality and simple, regular access patterns
  - Can get significant wins from paying attention to data used in inner loops
- `calloc` and `free` are not free

Performance koans

- Tuning often starts not with code, but with data structures.
- You cannot judge performance without a model.
- You cannot optimize what you cannot measure.
- Eventually, a better algorithm will always beat a tuning trick.
- Your time is worth more than the computer's time.
- The fastest code to write may be someone else's library.

Frameworks

- PETSc: `http://www.mcs.anl.gov/petsc/petsc-as/`
- Trilinos: `http://trilinos.sandia.gov/`

The build problem

Thomas Epperly and Gary Kumfert did a survey of code groups working in a variety of disciplines at Lawrence Livermore National Laboratory. Their results suggested that of the effort spent on modern scientific code, perhaps 10 and up to 30 percent of it (for small and large projects, respectively) went into trying to solve the build problem. My own experience confirms this estimate.

– *P. Dubois, "Why Johnny Can't Build,"*
Computing in Science & Engineering, 5 (2003), pp. 83–88.

The build problem

Things are getting a little better:

- Fewer common platforms
- Helpful tools like `CMake`
- More people working on package systems

But setting things up is still a pain!

Why it's worthwhile

- Someone else did the debugging.
- Someone else did the tuning.

The role of frameworks

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort. PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.

– Barry Smith

PETSc programming basics

- Object-oriented C
- Sits on top of MPI
- Most operations are (logically) collective
- Provides sparse LA, nonlinear solvers, ODE solvers
- Interfaces to many other solvers
- Option database to play with solvers at run time
- Built-in profiling support

PETSc in the developer's words

- Numerous tutorials and examples at the PETSc web site
- Let's look at some!

If you're following along at home:

`http://www.mcs.anl.gov/petsc/petsc-as/documentation/tutorials/UTAustinTutorial2011.pdf`

Slides 58–90; 111–118

(Slides 121–128 are good for your own time, too)

Questions for discussion

- Why is PETSc in object-oriented C?
- What do you think goes into vector assembly? How would you implement it?
- How might matrix assembly differ?
- Why is preallocation strongly recommended for matrices, but not for vectors?