How can application developers respond to advanced architectures?

Timothy Kelley

Applied Computer Science Group (CCS-7)/LANL

Salishan 2012

Many worthy efforts are underway to address the challenges ahead for scientific simulation…

- How do we build and run an exascale machine?
- What chips will we run on?
- What interconnect?
- How will the OS stack change?
- How will we think of data persistence?
- What programming languages? APIs? models?
- How do we evolve software?
- ... and many more!

Social challenges are important, too

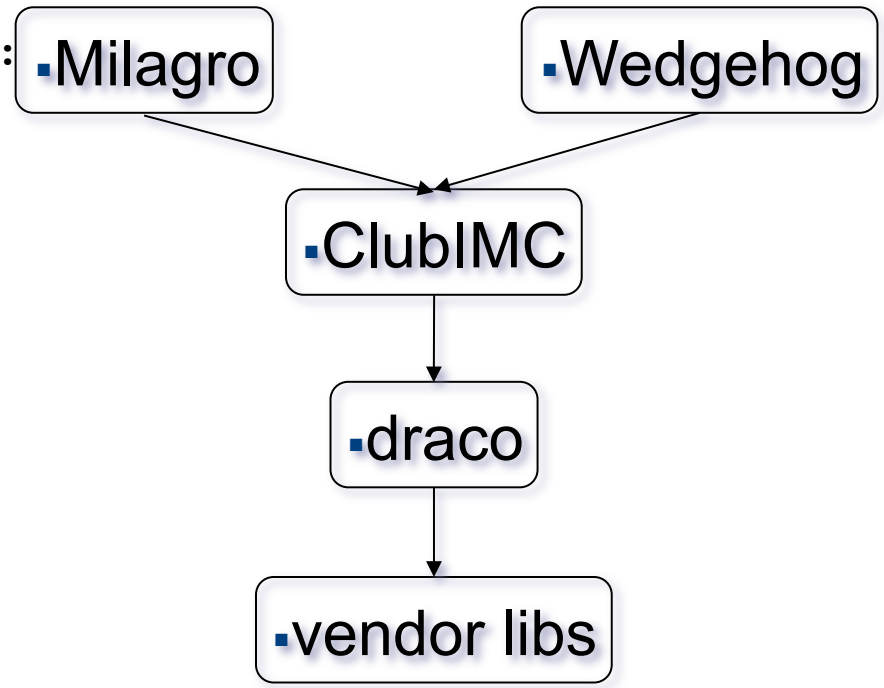
- **Who will program this system?**
  - Better yet, who will debug it?
  - And how do we build this programmer?
  - What about as we move away from the heart of HPC?

Learning about where we need to go as scientific simulation programmers

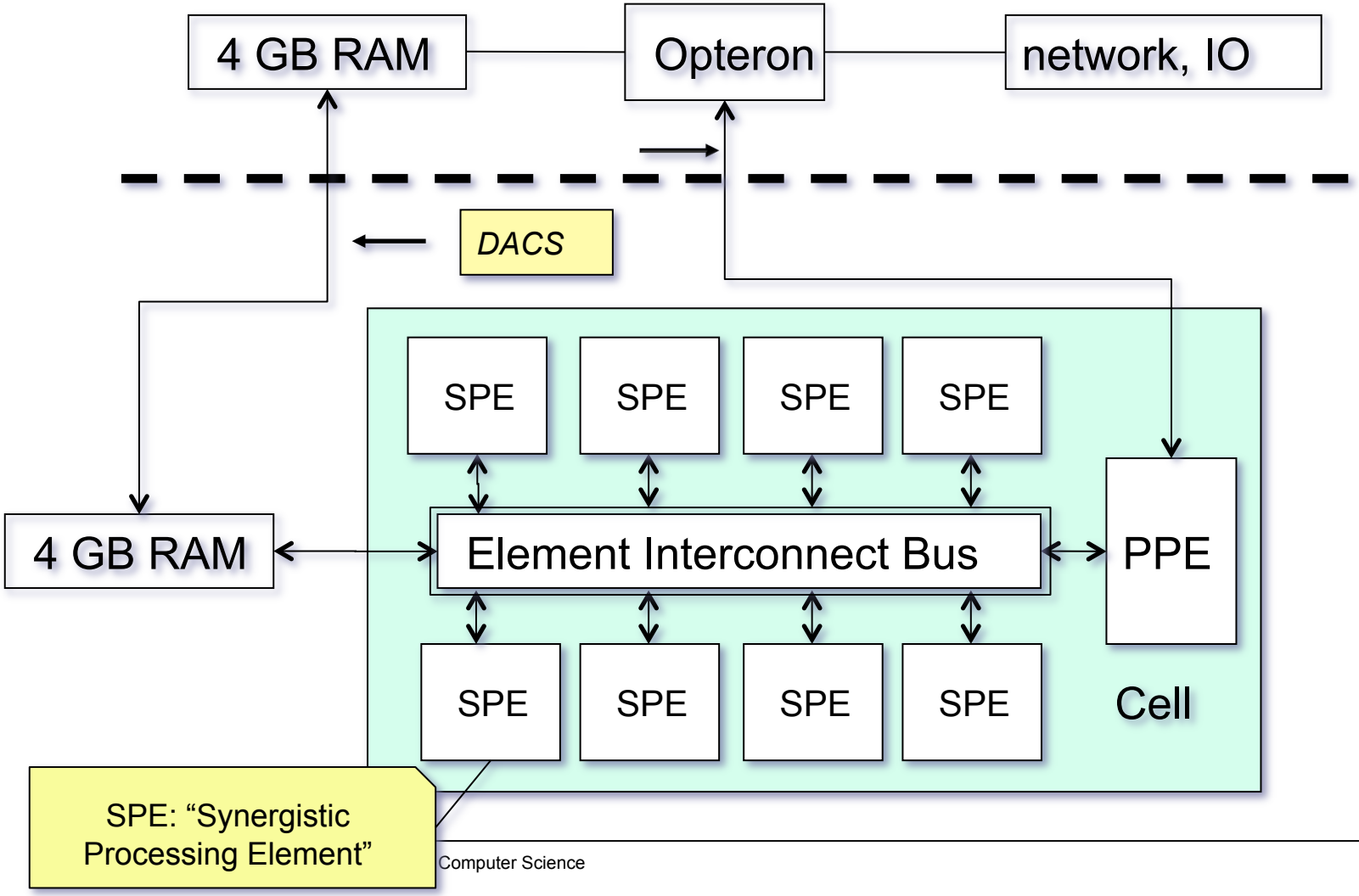
- Rewriting, re-thinking Implicit Monte Carlo transport for Roadrunner
- The advanced architecture tutorial project
- SWIFT
- Language as a means to develop developers

Implicit Monte Carlo simulates thermal X-ray transport for time-dependent, nonlinear problems

- **Fleck & Cummings time discretization**
- **object-oriented, generic C++:**
  - templated on mesh type, freq type, particle type
- **transports particles 3D, meshes articulated in 1,2,3D**
- **multigroup frequency treatment**
- **supports AMR**
- **two distributed parallel modes: mesh replicated, decomposed**
- **Wedgehog: Fortran callable interface library**



An app programmer's view of Roadrunner hybrid node: one Opteron + one Cell



IMC on Roadrunner timeline, 2006-11

- **Summer, 2006: multiple efforts undertaken to port codes to possible Roadrunner architectures.**
- **2006-7: Two efforts (Henning, Kelley) to port Implicit Monte Carlo transport to Roadrunner.**
  - Top-down: free reign with data structures, algorithms.
  - Bottom-up: migrate from existing code base.
  - Both approaches showed similar speedups after 8-9 months work; bottom-up approach chosen.
- **2008-9: Additional IMC physics ported, one major sync with trunk. Much help from IMC code team! (Urbatsch, Hungerford, Rockefeller)**
- **2010-11: RR branch merged with trunk, IMC team takes control.**

We were successful on the IMC/Roadrunner project.

- **Working, accelerated code**
- **Changed the MC transport algorithm**
  - hierarchical concurrency
  - model expressed as set of C++ classes
  - model can be implemented for multiple machine architectures
- **Decoupled particle generation from particle transport**
- **Introduced streams between particle generator, particle transporter, and particle disposer**
  - streams enable physical decoupling
- **See [1] for much more detail**

We learned some interesting things.

- **Advanced architectures create project management problems**
  - the Roadrunner code was a major redesign & rewrite
    - introduced/rewrote ~10 kloc++
  - code was forked for several years (now merged!)
- **Architecture-specific coding wasn't the hard part**
- **Code not properly vectorized (ongoing)**
- **Tally strategy worked for ~10 threads (ongoing)**
  - …probably won't scale to 100+
- **Debugging was painful**
  - We found...gaps...between the machine and C

IMC was one of many efforts. Detailed presentations at

http://www.lanl.gov/roadrunner/rrseminars.shtml

Roadrunner is the foundation of our efforts to use advanced architectures

- **Formed Applied Computer Science Group (CCS-7)**
  - Unite application developers, computer scientists
  - Build on Roadrunner experience to continue moving forward
- **Roadrunner technical seminar series (2008) [1]**
- **Roadrunner programming classes (2008-10)**
- **OpenCL programming classes (2011-2)**
- **XCP-CCS advanced architectures tutorial project (2011-2)**
- **The SWIFT project (2012->2015)**

We tried a new idea: ‘the advanced architecture tutorial’

- **First iteration, FY 11; second iteration, FY 12**
- **Six participants from XCP & CCS divisions**
- **Format:**
  - small (mini-app) code project.
  - sustained involvement—1/4 time for one year
- **Goals:**
  - communicate ideas about abstraction; improve software engineering skills; learn to program advanced architectures.
- **Some lessons learned:**
  - C++ is hard to learn, harder to use well
  - programming advanced architectures is not the hard part
  - the hard part is developing the model of the computation
- **Rolled those lessons into the 2012 iteration (Lally)**

SWIFT

- **New multiphysics code project (started 2012)**
- **Goals:**
  - Develop more flexible approach to writing codes;
  - Incorporate newer programming techniques;
  - reduce time, cost to develop codes.
- **50-50 mix of Roadrunner veterans and physics experts**
  - including people from the advanced architecture tutorial
- **Two week iterations; collocated; pair programming; …**

SWIFT: still early, but we're seeing encouraging signs.

- **Data-centric view of multiphysics code**
  - "You'll develop a database, whether you intend to or not."
- **Grappling with C++, OOP, generic programming**
  - what's the right mix?
- **Experimenting with different code approaches**
- **STL/Thrust-style loops versus traditional loops**
  - reified loops promising for portability (cf. [2])
  - need to be sure they can be optimized
- **Thinking of how data & algorithms will decompose**
  - for parallelism
  - for resilience

The fundamentals still apply.

- **Develop a mental model of the computation before coding.**
- **Write to a model of the computation, not to a machine.**
  - ahem, OpenCL
  - re-implement the model for different architectures
- **Communicate the model to the maintainer**
- **Architecture breaks iteration**
  - different vector sizes, memory characteristics...
  - reify iteration!
- **Beware of shared mutable state (e.g. IMC tallies)**
  - shared concurrently between threads
  - shared sequentially between functions/modules/packages
- **Restrict context of code**
  - greater composability, modularity

Domain scientists are not trained in the fundamentals of computing

- **Typical training: Fortran book, mentor's code**
- **Advanced training: C++ book, mentor's code**
- **Mental horizon restricted to code artifacts**
  - namely: doubles, ints, arrays, loops
- **Computation appears to be a purely phenomenological undertaking**

"Computer science is no more about computers than astronomy is about telescopes"

Languages give us great leverage on how we think

- Languages and programming methods exert enormous influence on our thinking
- Scientific simulation has a de facto language monoculture

Languages give us great leverage to improve

- Languages and programming methods exert enormous influence on our thinking
- Scientific simulation has a de facto language monoculture

Languages give us great leverage to improve

- **Languages and programming methods exert enormous influence on our thinking**
- **Scientific simulation has a de facto language monoculture**
- **We want languages/methods that encourage:**
  - forming a clear model of the computation
  - expressing the model in the code
  - demonstrating correctness before running (less debugging)
    - testing shows lack of failure detection, not correctness
  - careful control of state mutation
  - reification of control flow
  - reusability & composability
- **NB Not trying to find THE language**

What about Object Oriented Programming?

- **May be a good place to end up, but it's hard to get there**
  - OOP suffers from (at least) poor presentation
  - Emphasizes metaphor over math
- **OOP is "close to home"**
  - easy to bring bad habits along
- **OTOH:**
  - thinking in design patterns a step up
- **Distinguish C++ from OOP**

Best OOP advice ever:

- **"Model the computation, not the domain." [3]**
  - typical presentation other way around

Maybe we need to look farther afield...

Suggestion: try Haskell (or any functional programming language)

- **Many FP languages today—not just LISP**
  - ML, OCaml, Haskell; Erlang; JVM: Scala, Clojure; .Net: F#
- **FP emphasizes thinking in expressions, not machines**
- **Type systems are a great tool for expressing abstractions**
- **FP culture puts high premium on correctness**
- **FP compilers are getting good at performance**
  - fast, declarative stencil codes in Haskell [5]
  - SIMD support now going into Glasgow Haskell Compiler [6]
  - our own evaluation confirms this [8]
- **FP is hard**
  - because you're learning something new
  - some ideas introduced too soon

Summary

- Programming advanced architectures is quite plausible
- Roadrunner is the foundation on which we've built our advanced architectures efforts
- We're putting that experience to work in projects like SWIFT
- Haskell will change how you think about programming.
- Hope we'll see more experimentation with languages/ methods. Chapel? Go? D? Scala?

Thank you

references

[1] Roadrunner seminar series: http://www.lanl.gov/roadrunner/rrseminars.shtml

[2] Li-ta Lo et al, "PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators" http://viz.lanl.gov/projects/PISTONPaper.pdf

[3] Robert Webster, private comm.

[4] An interesting perspective on Haskell: "Escape from the Ivory Tower" http://yow.eventer.com/events/1004/talks/1054

[5] Lippmeier et al. "Efficient parallel stencil convolution in Haskell", http://www.cse.unsw.edu.au/~benl/papers/stencil/stencil-icfp2011-sub.pdf

[6] SIMD support in Glasgow Haskell Compiler: http://ghc-simd.blogspot.com/

[7] Using QuickCheck in the automotive industry: http://cufp.org/videos/model-based-testing-autosar-automotive-components

[8] https://github.com/losalamos/McPhD

additional slides

McPhD: initial effort to evaluate FP for simulations

- **McPhD: neutrino Monte Carlo transport [8]**
  - 1D, spherical, analytic cross sections
- **Clear separation in code between event generation (particle tracking) and event consumption (tallying)**
  - compiles to tightly coupled loop
  - key insight for moving MC to GPU, vectorization
- **Example of reified iteration**
  - simple approach to SMP parallelism
- **Good performance: matches a C++ analogue**
- **No show stoppers as far as we've gone**
- **Still much to learn**

What about functional programming?

Take Haskell (it's one limit) [4]

- **Equational definition of functions => simpler reasoning**
- **Pure functions: no (shared) mutable state!**
- **Side effects only where allowed**
- **Composability: build from small, correct pieces**
- **QuickCheck: sophisticated testing [7]**
- **Type system: lightweight formal methods**
  - prove, then check

Would domain scientists learn FP?

- **Definitely not all, probably not most**
  - enough to influence the culture?
- **Consider that programming, like physics, mixes mathematical and empirical aspects.**
  - Of course the mixes are different
  - but the same elements should prove appealing to some
  - FP exposes that mathematical side of programming