Symbolic Supercomputing

Bringing Scientific Supercomputing out of the Stone Age

A D Kennedy
University of Edinburgh
Abstract

We contemplate strongly typed, categorical, efficient, portable, reusable, modular, robust, architecture-neutral, buzzword-compliant software and languages for high performance parallel scientific computing.
Summary

- **Problem**
  - Programs need to run efficiently on multiple supercomputer architectures
  - It must be easy to make high-level algorithmic changes
  - Programs must adapt themselves around low-level assembly routines and data layouts
  - Algorithms are built out of a few common building blocks

- **Solution**
  - Categorical programming language that allows abstraction of these building blocks
  - Compile-time optimization
  - Flexible memory management

- **Obstacle**
  - **Risk**: application scientist are not willing to take on the risk of using experimental software in addition to the risks of any cutting-edge scientific supercomputing project
  - **Careers**: developing scientific software does not get you a permanent academic job
  - **Expertise**: very few applications scientist are familiar with modern software techniques; most supercomputer codes are written by graduate students who believe that documentation is an unnecessary evil
• Machine code
  • Location dependent
  • Functionality is Architecture dependent
  • Performance is machine dependent

• Assembler
  • Use symbolic names for location independence

• We must still write our time-critical kernels in assembler
  • Especially as we build our own hardware for these kernels
  • QCDSP, QCDOC, Blue Gene, …
  • We mainly optimize data motion (prefetching, etc.)

Real programmers code in binary!
Jurassic

- Macros
  - Textual substitution
  - Poor error handling
    - Errors reported in code that you didn’t write
  - Hard to debug
  - No type checking
    - Maybe at run-time
    - Painful at compile-time

What’s wrong with C++ templates?

Copyright © 1998 Christopher Strickland and Jeff Poling
Cretaceous

- Compilation
  - Fortran
  - Algol
  - C
- Sacrifice performance?
  - But not nearly as much as people thought
- Greatly increase programming productivity

Real programmers write in assembler!
APIs
- Application Programming Interfaces
- Standardised portable low-level components
  - Maths libraries
  - System calls
  - Run-time libraries
  - GMP
  - X windows
  - Microsoft COM

Data layout is fixed
- We lose either portability or efficiency

I think I’ll just reinvent the wheel instead…
Object-Oriented

- C++, Java
- Virtual functions
  - Virtual function tables, thunks
- Function and operator overloading
  - Already found in Algol68
- Inheritance
  - Multiple inheritance (C++)
  - Interfaces (Java)
- Low-level implementation is hidden
  - But too much is done at run-time
  - Memory management is either automatic or non-existent
Neolithic

Serialization
- Portable interchange format for objects
- Usually XML-based
- Grid
  - Exchange everything via serialised data
  - Many↔One instead of many↔many
  - Performance?
- Data layout transformations
  - FFTs
  - Redistribute grid points for different number of processors

I’ll flatten you any way that you like!
Categories

- Strong typing for classes
- A category is a class of classes sharing the same structure
  - But not necessarily having anything else in common
  - SU(2) and SU(3) are both groups, but you can’t multiply their elements together (except in C++ or Java)
- Structure is defined by
  - An explicit set of signatures
  - An implicit set of axioms
- Aspect-Oriented programming

Computer languages can’t evolve: they must be created by a vendor!
Modularity is natural
Follows mathematical structure
  - Why not write linear system solvers the way they are expressed in textbooks?
  - Select appropriate algorithms using type information
    - At compile time
    - E.g., use CG for positive symmetric matrices
  - Mendacity is sometimes useful
    - E.g., use CG for non-positive symmetric matrices
Plug-and-play algorithms
Plug-and-play data layout
Some useful hardware features can be abstracted
  - Vectors
  - Data parallelism
Optimizations

- Specialization
  - Use hand-coded kernels for “hot spots”
  - Make rest of code use data layout required by kernels
- Inlining
  - Remove unnecessary function calls
- Constant folding
  - Evaluate constant expressions at compile time
  - Evaluate constant functions at compile time
  - Partially evaluate functions at compile time

See how fast it goes after I’ve tuned it a little bit!
We need computational software to be…

- Efficient
- Portable
- Reusable
- Modular
- Robust
- Architecture-neutral

…and it can be done

- Must be a cooperative venture of application scientists and computer scientists
- Need to develop real software to get it accepted
- Must be constructed in layers with well-defined categorical interfaces
  - Low-level machine dependent layers
  - Intermediate building blocks (e.g., linear algebra)
  - Top-level application specific
- Categorical interface means that functionality is specified, not implementation or data layout
- Layers can be changed independently
- New memory management model is needed, neither universal garbage collection nor complete user responsibility is good enough
- Needs to be well-documented
- But is it fully buzzword-compliant?
Conclusions

Nevertheless, I’ll still write it in Fortran