Software Architecture for High-Performance Scientific Computing: A Pragmatic Approach

David E. Bernholdt
Oak Ridge National Laboratory

bernholtdde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
Licensing Information

• These lecture notes are distributed under the Creative Commons Attribution 2.5 License
  – http://creativecommons.org/licenses/by/2.5/

• In summary, you are free:
  – to copy, distribute, display, and perform the work
  – to make derivative works
  – to make commercial use of the work

• Under the following conditions:
  – Attribution. You must attribute the work in the manner specified by the author or licensor.

• For any reuse or distribution, you must make clear to others the license terms of this work.

• Any of these conditions can be waived if you get permission from the copyright holder.

• Your fair use and other rights are in no way affected by the above.

• Requested citation:
### Outline and Table of Contents (1)

<table>
<thead>
<tr>
<th>Topic</th>
<th>Slide No.</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Software Architecture for HPC Scientific Applications</td>
<td>8</td>
<td>Monday</td>
</tr>
<tr>
<td>Introduction to Component-Based Software Engineering</td>
<td>36</td>
<td>Monday-Tuesday</td>
</tr>
<tr>
<td>• Motivation</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>• Component Concepts and the Common Component Architecture</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>• The CCA Tools</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>• Language Interoperable CCA Components with Babel</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>• Strategies for Developing with Components</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>Lab: Hands-On CCA Components</td>
<td></td>
<td>Tuesday afternoon</td>
</tr>
<tr>
<td>Case Studies</td>
<td>181</td>
<td>Wednesday</td>
</tr>
<tr>
<td>• The NWChem Parallel Computational Chemistry Package (Monolithic)</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>• The Earth System Modeling Framework (Framework)</td>
<td>211</td>
<td></td>
</tr>
</tbody>
</table>
# Outline and Table of Contents (2)

<table>
<thead>
<tr>
<th>Topic</th>
<th>Slide No.</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case Studies: Component-Based Applications</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Towards “Community” Codes in Quantum Chemistry</td>
<td>233</td>
<td>Thursday</td>
</tr>
<tr>
<td>• Experimenting with New Parallel Algorithms in Quantum Chemistry</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>• A Toolkit for Combustion Modeling</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>• Community Interfaces and Software Reuse (TSTT Mesh Interface)</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>• Integrated (Multi-Physics) Fusion Simulations</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td><strong>Lab: Hands-On CCA Components in Depth</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hands-On Guide</td>
<td></td>
<td>Thursday afternoon</td>
</tr>
<tr>
<td><strong>Generated Applications</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Overview of Code Generation Approaches</td>
<td>291</td>
<td>Friday</td>
</tr>
<tr>
<td>• Case Study: Automatically Tuned Linear Algebra Software (ATLAS)</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>• Case Study: The Tensor Contraction Engine (TCE)</td>
<td>320</td>
<td></td>
</tr>
</tbody>
</table>
About the Instructor

My approach to software architecture draws heavily on my experience in multiple areas of computational science…

• BS and PhD in chemistry

• Ten years of research in computational chemistry, as an active architect and developer of several large-scale HPC application packages

• Ten years of research in computer science, specializing in HPC software environments, with an application focus

• Also familiar with software architecture issues in materials science, fusion, climate modeling, nuclear physics, and other areas

• Active in the Common Component Architecture effort, work on domain-specific languages, parallel programming models and languages
About this Course

• As the course title says, I take a **pragmatic** view of software architecture
  – Not a lot of formal software engineering
  – Very personal view

• (In my opinion) the biggest problem with software architecture is that there isn’t enough of it!
  – Many scientific software systems just “grow” without any concept of an architecture or design

• My primary goal is to get you **thinking about software architecture**
  – Examples of what considerations have gone into the architectures of packages I’m familiar with
  – Examples of various architectural paradigms used in actual codes

• This is my first time teaching such a course
  – I had to prepare this material without knowing a lot about your backgrounds, experience, and interests
  – I need your help to make this course as **useful to you** as possible
  – Questions and discussion are encouraged!
Where to Find More Information

• Most of the projects mentioned have URLs in the slides for more information about the project, including code, publications, etc.

• Course bibliography lists the primary literature publications on which this course is based
  – Follow citations in those papers for even more details
Introduction to Software Architecture for HPC Scientific Applications

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
What is Software Architecture?

• “Software architecture… can best be thought of as a representation of an engineered… software system, and the process and discipline for effectively implementing the design(s) for such a system.”
  
  -- Wikipedia.org

• “As the size of software systems increases, the algorithms and data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organization of the overall system—the software architecture—presents a new set of design problems.”

  -- Garlan and Shaw (1994)
Architecture vs Design

- Software **architecture** is an overall property of a software system
  - However a single software system might be broken down into multiple subsystems, with different architectural styles
  - Multiple architectural styles may be combined, or one style might be implemented using another style

- Software **design** is localized and more detailed

- For our purposes, we don’t need to make a rigorous distinction between the two
  - Research software tends to suffer from a lack of attention to both architecture and design!
Software Architecture and Software Engineering

• The study of software architecture is considered part of software engineering

• The good news…
  – The software engineering community has published much on architecture
  – Broad experience from many different types of applications

• The bad news…
  – Research software is not typically the subject of software engineering studies
  – Research software often has some special considerations
Architectures in Software Engineering (1)

From Garlan and Shaw (1994):
• Pipes and filters
  – Unix is classic example

• Data abstraction and object-oriented organization
  – Now popular, “built in” to languages like C++, Java, but can be implemented in others too

• Event-based, implicit invocation
  – Discrete event simulation, packet switching
Architectures in Software Engineering (2)

• Layered systems
  – Extremely common style today

• Repositories
  – Central data store, manipulated by multiple “knowledge sources”
  – Example: enterprise management system – many business processes interacting with a (logically) central database

• Table driven interpreters
  – Virtual machines
Architectures in Software Engineering (3)

- **Other Styles**
  - Distributed processes
    - Characterized by topologies, communication protocols, etc.
  - Domain-specific software architectures
    - Attempts to create “reference” architectures for individual application domains
  - State transition systems
  - Process control systems

- **Heterogeneous architectures**
  - A single software system might be broken down into multiple subsystems, with different architectural styles
  - Multiple architectural styles may be combined, or one style might be implemented using another style
What is “Research Software”? 

• Software created and used as a tool in the pursuit of scientific or engineering research ideas 
  – i.e. a simulation of some physical phenomena (“computational science”) 

• Usually not created with the intent of being a commercial software product 
  – Though research software is sometimes commercialized 

• Most often associated with an individual, research group/institute, or a scientific community/discipline 

• Software targeting high-end (parallel) computers is a subset of research software 
  – But arguably the most important, and typically the most complex subset
Why HPC, Scientific, and Research Software is Special

• Developers of research software are not usually trained in software engineering
  – Often not trained in computer science at all

• Focus is on the science
  – Software is merely a means to an end
  – Software engineering is often viewed as a hindrance

• A long-range vision for scientific research isn’t the same as a long-range vision for the software system
  – In many cases, researchers don’t have a long-range vision for the software!

• Software is long-lived and evolves slowly
  – But must run on HPC hardware which changes rapidly
Common HPC/Scientific Architectural Patterns

• Monolithic
  – Case Study: NWChem (computational chemistry)

• Framework
  – Case Study: Earth System Modeling Framework

• Components
  – Case Studies: Common Component Architecture-based applications

• Generated
  – Case Studies: ATLAS (linear algebra), Tensor Contraction Engine (chemistry)

• Distributed

  • These are my personal terms – I know of no generally agreed terminology or definitions. The distinctions among these patterns can be quite fuzzy
  • The goal is not to label things, but to look at important characteristics of each architecture and understand how you might use various concepts in your own projects
Monolithic Applications

• Application is logically a single (large) executable
  – May be configurable at build-time
  – No run-time “plug-in” capabilities
  – May make (extensive) use of third-party libraries

• Term “monolithic” is not pejorative
  – Many monolithic apps are well structured
  – But others are more organic

• Applications developed without (much) planning will typically be monolithic in style

• Longevity of monolithic applications tends to require one or more gurus who understand the overall application structure in some depth
Application Frameworks

- Provides the computational infrastructure to support a class of applications
  - Typically focused on a single scientific domain
- Generally defines an overall workflow and provide (domain-specific) utilities
  - Specific applications created by writing high-level modules which fit into workflow scheme and call utilities extensively
- High-level simulation modules often written, maintained, and distributed by groups other than the developers of the core framework
- Often provide dynamic run-time plug-in capabilities for high-level modules
- Framework itself requires gurus to support, but specific application modules often do not
Examples of Framework Architectures

- Earth System Modeling Framework
  - http://www.esmf.ucar.edu

- CACTUS (numerical relativity & more)
  - http://www.cactuscode.org

- SALOME (mesh-based multiphysics simulations)
  - http://www.salome-platform.org

- Overture (partial differentials equations)
  - http://www.llnl.gov/casc/Overture/
Component Environments

• Focus on opaque components and the interfaces through which they interact

• Plug-and-play paradigm for application creation
  – Often dynamic at run-time

• Easily mimics monolithic and framework architectural styles
  – The line between framework and component architecture can be very fuzzy

• Better supports separation of concerns among developers than other approaches
Examples of Component Environments

- **Common Component Architecture**
  - http://www.cca-forum.org

- **CORBA Component Model**

- **COM/DCOM**
  - http://www.microsoft.com/com

- **Enterprise JavaBeans**
Generated Applications

• Focus on providing higher-level ways for application developers to express their computations
  – Code generation
  – Very high-level languages
  – Address performance and/or productivity issues

• Each individual tool restricted to a narrow domain

• Created by experts for use by non-experts
Examples of Generated Applications

- **Tensor Contraction Engine (quantum chemistry)**
  - [http://www.cse.ohio-state.edu/~gb/TCE/](http://www.cse.ohio-state.edu/~gb/TCE/)
- **Automatically Tuned Linear Algebra Software**
  - [http://math-atlas.sourceforge.net](http://math-atlas.sourceforge.net)
- **FFTW (fast Fourier transforms)**
  - [http://www.fftw.org](http://www.fftw.org)
- **SPIRAL (signal processing)**
  - [http://www.spiral.net](http://www.spiral.net)
- **Broadway compiler (linear algebra library annotations)**
  - [http://www.cs.utexas.edu/users/less/broadway.html](http://www.cs.utexas.edu/users/less/broadway.html)
- **Telescoping Languages (general library annotations)**
  - [http://telescoping.rice.edu](http://telescoping.rice.edu)
Distributed Architectures

• Common in “grid” and related technical applications
  – More loosely coupled than typical HPC
  – Typically factors into HPC simulation only at very high levels or as remote data/computation service where performance is not needed

• Architecturally, closely allied with non-technical distributed computing
  – Strongly driven by communication links
  – Vast literature

• I will not attempt to cover distributed architectures in this class
  – Not enough time to do them justice
  – I don’t have enough experience with them
  – Concerns are (largely) orthogonal to those of tightly-coupled HPC applications
Which Approach is Best?

• It depends on your developers
• It depends on where your software base comes from
• It depends on your users
• It depends on the domain of the project
• It depends on your sponsors
• It depends where you are in your project’s lifecycle, and your long-term plans for it
• It depends on your timelines
• It depends on the priorities of the project
Why Developers Matter

• They’re writing the code!

• Are they familiar with the required languages, tools, and approaches?
  – If not, how quickly can they learn them?

• Are they focused on the science or the code?
  – If they’re not hired programmers, it may be hard to impose software engineering discipline on them

• What are their technical backgrounds?
  – Do you have the cross-disciplinary support you need within the team (i.e. computer scientists, applied mathematicians)? Or must you go outside the project for help of this sort?
  – Some architectural styles make it easier to divide up coding by discipline

• Where are they located (geographically)?
  – Level of communication between team members is an important consideration
  – Some architectural styles make it easier to divide up coding to distinct teams
Why the Software Base Matters

• Where will most of your software come from?

• If you will be writing your code from scratch, you have great architectural flexibility

• Code reuse may result in constraints
  – Existing code may be less flexible
  – Reuse of code from the same development team may be easier than reuse of code from others

• Some libraries may impose structural or architectural requirements

• Intellectual property considerations…
  – For “outside” software you use (i.e. is it redistributable?)
  – In distribution of your application (how much flexibility do you give users?)
Why Users Matter

• They’re running the code!

• Different user groups will have different tolerance for the apparent complexity of the application
  – Some will need a very simple model for deployment & use
    • i.e. those who want a “black box” code, and don’t do development
  – Others can tolerate more complexity
    • i.e. users who are also developers
  – Different architectures have natural levels of complexity
    • Monolithic & Generated < Framework < Components
  – Complex architectures can usually be simplified by good packaging
    • But it requires additional effort
    • i.e. package component application as a single static binary
Why the Scientific Domain Matters

• “This is the way we’ve always done it”
  – Different domains have different histories and practices

• Domains have different traditions for the size and complexity of individual codes

• For example (in my personal experience)…
  – Climate scientists are conservative about their software because it is so widely and prominently used (simulations influence government policies)
  – Plasma physics codes have very long lifetimes and are all written in Fortran90 – large inertia to do anything differently
  – Most users of quantum chemistry codes do not do any development. But many of those who do development are very interested in modern software development techniques, such as components

• May or may not be reusable code available to help
Why Sponsors Matter

• They’re paying for the code!

• But what is the sponsor actually paying for?
  – Software development, or
  – Scientific advances resulting from software development?
  – “I never had a proposal funded to write software.”
    – John Pople, creator of the Gaussian quantum chemistry package, and Nobel Laureate

• The sponsor may have expectations as to the level of engineering and longevity of the code

• Can you implement the architectural style you want in the budget and other constraints of the sponsor?
Why the Project’s Lifecycle Matters

• It is easier to impose good software engineering (architecture, design, etc.) if the project is just starting.

• It is usually harder to impose an architecture on code that already exists.

• It is even harder to impose an architecture on “mature” code that’s been around a long time:
  – Original developers may have moved on.
  – Knowledge of existing code structure and how to adapt may have been lost.

• Having an old, unmaintainable code makes it easier to argue for starting over from scratch:
  – “Do it right the second time.”
Why Timelines Matter

• Both project timelines and staffing timelines are important
• Depending on the experience and abilities of the developers, different approaches will require different amounts of time to implement
• Project timeline: Can you get the software written by the deadline?
• Staff timeline: Do the developers have time to…
  – Learn the software development techniques required by your architectural design,
  – Write and test the software, and
  – Use it for the scientific research that’s important to them
  – Before the funding ends or they find another job?
• Example: You’re relying on postdocs for most of your development
  – In your field, they don’t normally learn anything about generic programming in school. But it is critical to your software design
  – They’ll be around only 1-3 years
  – Professionally, they get credit for scientific advances, not learning generic programming
Why Project Priorities Matter

• Just as sponsors may have expectations, project leaders may have expectations and priorities

• If the priority is scientific results at any cost, it will be hard to convince them to pay attention to software engineering

• If the software is recognized as a tool to obtain scientific results, it may be possible to persuade them of the value of having a good tool
Which Approach is Best (Redux)?

• We can’t definitively say that one approach is better than all the others
  – Components are perhaps the most general, but are just beginning to be mature enough (in an HPC context), and may be overkill in some cases

• Analyze the needs of your project and match your architectural approach to those needs
  – Software engineering for research software is as much a social issue as a technical one!

• During the remainder of this course, I will present a number of case studies to examine why various projects chose different architectural approaches

• Special emphasis on presenting “new” architectural approaches
  – Components
  – Generated applications
Introduction to Component-Based Software Engineering

From the Common Component Architecture Tutorial written by the CCA Forum Tutorial Working Group
http://www.cca-forum.org/tutorials/
What Can Component Technology do for Scientific Computing?
Managing Code Complexity

Some Common Situations:

- Your code is so large and complex it has become fragile and hard to keep running
- You have a simple code, and you want to extend its capabilities – rationally
- You want to develop a computational “toolkit”
  - Many modules that can be assembled in different ways to perform different scientific calculations
  - Gives users w/o programming experience access to a flexible tool for simulation
  - Gives users w/o HPC experience access to HPC-ready software

How CCA Can Help:

- Components help you think about software in manageable chunks that interact only in well-defined ways
- Components provide a “plug-and-play” environment that allows easy, flexible application assembly
Example: Computational Facility for Reacting Flow Science (CFRFS)

• A toolkit to perform simulations of unsteady flames
  • Solve the Navier-Stokes with detailed chemistry
    – Various mechanisms up to ~50 species, 300 reactions
    – Structured adaptive mesh refinement
  • CFRFS today:
    – 61 components
    – 7 external libraries
    – 9 contributors

“Wiring diagram” for a typical CFRFS simulation, utilizing 12 components.

CCA tools used: Ccaffeine, and ccafe-gui
Languages: C, C++, F77
Helping Groups Work with Software

Some Common Situations:

• Many (geographically distributed) developers creating a large software system
  – Hard to coordinate, different parts of the software don’t work together as required
• Groups of developers with different specialties
• Forming communities to standardize interfaces or share code

How CCA Can Help:

• Components are natural units for
  – Expressing software architecture
  – Individuals or small groups to develop
  – Encapsulating particular expertise
• Some component models (including CCA) provide tools to help you think about the interface separately from the implementation
Example: Quantum Chemistry

- Integrated state-of-the-art optimization technology into two quantum chemistry packages to explore effectiveness in chemistry applications.
- Geographically distributed expertise:
  - California - chemistry
  - Illinois - optimization
  - Washington - chemistry, parallel data management
- Effective collaboration with minimal face-to-face interaction

Schematic of CCA-based molecular structure determination quantum chemistry application.

Components based on: MPQC, NWChem (quantum chem.), TAO (optimization), Global Arrays, PETSc (parallel linear algebra)

CCA tools used: Babel, Ccaffeine, and ccafe-gui

Languages: C, C++, F77, Python
Example: TSTT Unstructured Mesh Tool Interoperability

- Common interface for unstructured mesh geometry and topology
  - 7 libraries: FMDB, Frontier, GRUMMP, Mesquite, MOAB, NWGrid, Overture
  - 6 institutions: ANL, BNL/SUNY-Stony Brook, LLNL, PNNL, RPI, SNL
- Reduces need for $N^2$ pairwise interfaces to just $N$

Illustration of geometry domain hierarchy used in TSTT mesh interface.

CCA tools used: Babel (SIDL), Chasm
Library languages: C, C++, F77, F90
Language Interoperability

Some Common Situations:
• Need to use existing code or libraries written in multiple languages in the same application?
• Want to allow others to access your library from multiple languages?
• Technical or sociological reasons for wanting to use multiple languages in your application?

How CCA Can Help:
• Some component models (including CCA) allow transparent mixing of languages
• Babel (CCA’s language interop. tool) can be used separately from other component concepts
Examples

**hypre**

- High performance preconditioners and linear solvers
- Library written in C
- Babel-generated object-oriented interfaces provided in C, C++, Fortran

**LAPACK07**

- Update to LAPACK linear algebra library
  - To be released 2007
  - Library written in F77, F95
- Will use Babel-generated interfaces for: C, C++, F77, F95, Java, Python
- Possibly also ScaLAPACK (distributed version)

“I implemented a Babel-based interface for the hypre library of linear equation solvers. The Babel interface was straightforward to write and gave us interfaces to several languages for less effort than it would take to interface to a single language.”

-- Jeff Painter, LLNL. 2 June 2003

**CCA tools used:** Babel, Chasm
Coupling Codes

Some Common Situations:

• Your application makes use of numerous third-party libraries
  – Some of which interact (version dependencies)
• You want to develop a simulation in which your code is coupled with others
  – They weren’t designed with this coupling in mind
  – They must remain usable separately too
  – They are all under continual development, individually
  – They’re all parallel and need to exchange data frequently

How CCA Can Help:

• Components are isolated from one another
  – Interactions via well-defined interfaces
  – An application can include multiple versions of a component
• Components can be composed flexibly, hierarchically
  – Standalone application as one assembly, coupled simulation as another
• CCA can be used in SPMD, MPMD, and distributed styles of parallel computing
• CCA is developing technology to facilitate data and functional coupling of parallel applications
Example: Global Climate Modeling and the Model Coupling Toolkit (MCT)

- MCT is the basis for Community Climate System Model (CCSM3.0) coupler (cpl6)
- Computes interfacial fluxes and manages redistribution of data among parallel processes
- Written in F90, Babel-generated bindings for C++, Python
- **CCA tools used:** Babel, Chasm

Schematic of CCSM showing coupler managing data exchanges between atmosphere, sea ice, ocean, and land models.

(From http://www.ccsm.ucar.edu/models/ccsm3.0/cpl6/)
Example: Integrated Fusion Simulation

- Proof-of-principle of using CCA for integrated whole-device modeling needed for the ITER fusion reactor
- Couples radio frequency (RF) heating of plasma with transport modeling
- Coarse-grain encapsulation of pre-existing programs
- Follow-on plans for RF, transport, and magneto-hydrodynamics

“Wiring diagram” for integrated fusion simulation.

Components based on: AORSA, Houlberg’s transport library
New components: Driver, State
CCA tools used: Babel, Chasm, Ccaffeine, ccafe-gui
Languages: C++, F90, Python
An Introduction to Components and the Common Component Architecture
Goals of This Module

• Introduce basic concepts and vocabulary of component-based software engineering and the CCA

• Highlight the special demands of high-performance scientific computing on component environments

• Give you sufficient understanding of the CCA to begin evaluating whether it would be useful to you
What are Components?

• No universally accepted definition in computer science research, but key features include…

• A unit of software development/deployment/reuse
  – i.e. has interesting functionality
  – Ideally, functionality someone else might be able to (re)use
  – Can be developed independently of other components

• Interacts with the outside world only through well-defined interfaces
  – Implementation is opaque to the outside world

• Can be composed with other components
  – “Plug and play” model to build applications
  – Composition based on interfaces
What is a Component Architecture?

• A set of standards that allows:
  – Multiple groups to write units of software (components)…
  – And have confidence that their components will work with other components written in the same architecture

• These standards define…
  – The rights and responsibilities of a component
  – How components express their interfaces
  – The environment in which components are composed to form an application and executed (framework)
  – The rights and responsibilities of the framework
A Simple Example: Numerical Integration Components

Interoperable components (provide same interfaces)

Driver

GoPort

IntegratorPort

MonteCarloIntegrator

RandomGeneratorPort

PiFunction

RandomGenerator

NonlinearFunction

LinearFunction

FunctionPort

FunctionPort

FunctionPort

FunctionPort

FunctionPort

FunctionPort

FunctionPort

FunctionPort
An Application
Built from the Provided Components

Hides complexity: Driver doesn’t care that MonteCarloIntegrator needs a random number generator
Another Application…
Application 3…

Diagram showing the relationships between different components:
- Driver
  - GoPort
  - IntegratorPort
  - NonlinearFunction
    - FunctionPort
  - LinearFunction
    - FunctionPort
  - PiFunction
    - FunctionPort
  - RandomGenerator
    - RandomGeneratorPort
  - MonteCarloIntegrator
    - IntegratorPort
    - FunctionPort
    - RandomGeneratorPort
And Many More…

Dashed lines indicate alternate connections

Create different applications in "plug-and-play" fashion
Comparison of Application Development Approaches

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Monolithic Simulation Code</th>
<th>Simulation Frameworks</th>
<th>Library-Based</th>
<th>Component-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for specific workflows and information flows</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Flexibility w.r.t. workflow and information flow</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>User-level extensibility</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Ease of incorporation of outside code (code reuse)</td>
<td>Low</td>
<td>Low-Medium</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Ease of experimentation</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Amount of new code required to create a complete simulation</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High (reuse can reduce)</td>
</tr>
<tr>
<td>Breadth of current “ecosystem” for “plugins”</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low (but growing)</td>
</tr>
<tr>
<td>Ease of coupling simulations</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>
Be Aware: “Framework” Describes Many Things

- Currently in scientific computing, this term means different things to different people

- **Basic software composition environment**
  - Examples: CCA, CORBA Component Model, ...

- An environment facilitating development of applications in a particular scientific domain (i.e. fusion, computational chemistry, …)

- An environment for managing complex workflows needed to carry out calculations
  - Example: Kepler: http://kepler-project.org

- **Integrated data analysis and visualization environments (IDAVEs)**

- Lines are often fuzzy
  - Example: Cactus, http://www.cactuscode.org

- Others types of frameworks *could* be built based on a basic software composition environment
Relationships: Components, Objects, and Libraries

- Components are typically discussed as objects or collections of objects
  - Interfaces generally designed in OO terms, but…
  - Component internals need not be OO
  - OO languages are not required

- Component environments can enforce the use of published interfaces (prevent access to internals)
  - Libraries can not

- It is possible to load several instances (versions) of a component in a single application
  - Impossible with libraries

- Components must include some code to interface with the framework/component environment
  - Libraries and objects do not
What is the CCA?

• Component-based software engineering has been developed in other areas of computing
  – Especially business and internet
  – Examples: CORBA Component Model, COM, Enterprise JavaBeans

• Many of the needs are similar to those in HPC scientific computing

• But scientific computing imposes special requirements not common elsewhere

• CCA is a component environment specially designed to meet the needs of HPC scientific computing
Special Needs of Scientific HPC

• Support for legacy software
  – How much change required for component environment?

• Performance is important
  – What overheads are imposed by the component environment?

• Both parallel and distributed computing are important
  – What approaches does the component model support?
  – What constraints are imposed?
  – What are the performance costs?

• Support for languages, data types, and platforms
  – Fortran?
  – Complex numbers? Arrays? (as first-class objects)
  – Is it available on my parallel computer?
CCA: Concept and Practice

- In the following slides, we explain important concepts of component-based software from the CCA perspective.
- We also sketch how these concepts are manifested in code (full details in the Hands-On).
- The **CCA Specification** is the mapping between concept and code:
  - A standard established by the CCA Forum.
  - Expressed in the Scientific Interface Definition Language (SIDL) for language neutrality (syntax similar to Java).
  - SIDL can be translated into bindings for specific programming languages using, e.g., the Babel language interoperability tool.
CCA Concepts: Components

- A component encapsulates some computational functionality
- Components provide/use one or more interfaces
  - A component with no interfaces is formally okay, but isn’t very interesting or useful
- In SIDL, a component is a class that implements (inherits from) `gov.cca.Component`
  - This means it must implement the `setServices` method to tell framework what ports this component will provide and use
  - `gov.cca.Component` is defined in the CCA specification
CCA Concepts: Ports

- Components interact through well-defined interfaces, or ports
  - A port expresses some computational functionality
  - In Fortran, a port is a bunch of subroutines or a module
  - In OO languages, a port is an abstract class or interface

- Ports and connections between them are a procedural (caller/callee) relationship, not dataflow!
  - e.g., `FunctionPort` could contain a method like `evaluate(in Arg, out Result)` with data flowing both ways
CCA Concepts: *Provides* and *Uses* Ports

- Components may **provide** ports – **implement** the class or subroutines of the port ("
  **Provides” Port"
) – *Providing* a port implies certain inheritance relationships between the component and the abstract definition of the interface *(more details shortly)* – A component can *provide* multiple ports
  - Different “views” of the same functionality, or
  - Related pieces of functionality

- Components may **use** ports – **call** methods or subroutines in the port ("
  **Uses” Port"
) – *Use* of ports is just like calling a method normally except for a little additional work due to the “componentness” *(more details shortly)* – No inheritance relationship implied between caller and callee – A component can *use* multiple ports
Components and Ports (in UML)

Note that only the **provides** ports appear in the component’s inheritance hierarchy. **Uses** ports do not.

A component must **implement** the CCA spec’s component interface.

- **setServices**
  - `gov.cca.Component`
  ```java
  setServices(services: gov.cca.Services)
  ```

A port must **extend** the CCA spec’s port interface.

- **IntegratorPort**
  ```java
  <<interface>>
  gov.cca.Port
  ```
  ```java
  integrate(lowBound: double, upBound: double, count: int): double
  ```

A class for Midpoint Integrator component

- **MidpointIntegrator**

Class for Midpoint Integrator component

Key: `= Inheritance`

**SIDL keywords**
Components and Ports (in SIDL)

package \texttt{gov.cca} {
    interface \texttt{Component} {
        void setServices(...);
    }
}

package \texttt{gov.cca} {
    interface \texttt{Port} {
    }
}

package \texttt{integrators} {
    interface \texttt{IntegratorPort} {
        double integrate(...);
    }
}

package \texttt{integrators} {
    class \texttt{Midpoint} extends \texttt{gov.cca.Port} {
        double integrate(...);
    }
}

package \texttt{integrators} {
    class \texttt{MidpointIntegrator} implements \texttt{gov.cca.Component}, \texttt{interator IntegratorPort} {
        double integrate(...);
        void setServices(...);
    }
}

Key:
\begin{itemize}
\item \textup{\textgreater} = Inheritance
\item SIDL inheritance keywords
\end{itemize}
Using Ports

- Calling methods on a port you use requires that you first obtain a “handle” for the port
  - Done by invoking `getPort()` on the user’s `gov.cca.Services` object
  - Free up handle by invoking `releasePort()` when done with port

- Best practice is to bracket actual port usage as closely as possible without using `getPort()`, `releasePort()` too frequently
  - Can be expensive operations, especially in distributed computing contexts
  - Performance is in tension with dynamism
    - can’t “re-wire” a ports that is “in use”
Where Do Ports Come From?

• Most ports are designed and implemented by users of CCA
  – May be specific to an application or used more broadly (i.e. community-wide)

• The CCA specification defines a small number of ports
  – Most are services CCA frameworks must provide for use by components
  – Some are intended for users to implement in their components, and have a special meaning recognized by the framework
    • E.g. \texttt{gov.cca.ports.GoPort} provides a very simple protocol to start execution of component-based applications
Interfaces are Key to Reuse and Interoperability of Code

- **Interoperability** -- multiple implementations conforming to the same interface
- **Reuse** -- ability to use a component in many applications
- The larger the community that agrees to the interface, the greater the opportunity for interoperability and reuse
Interfaces are an Investment

• The larger the community, the greater the time & effort required to obtain agreement
  – Equally true in component and non-component environments
    • MPI 1.0 (well understood at the start) took 8 months, meeting every six weeks
    • MPI 2.0 (not well understood at the start) took 1.5 years, meeting every six weeks
  – Convenient communities are often “project” and “scientific domain”

• Formality of “standards” process varies

• Biggerstaff’s Rule of Threes
  – Must look at at least three systems to understand what is common (reusable)
  – Reusable software requires three times the effort of usable software
  – Payback only after third release
CCA Concepts: Frameworks

- The framework provides the means to “hold” components and compose them into applications
- Frameworks allow connection of ports without exposing component implementation details
- Frameworks provide a small set of standard services to components
  - Framework services are CCA ports, just like on components
  - Additional (non-standard) services can also be offered
  - Components can register ports as services using the ServiceProvider port

- **Currently**: specific frameworks are specialized for specific computing models (parallel, distributed, etc.)
- **Future**: better integration and interoperability of frameworks
Components Must Keep Frameworks Informed

- Components must tell the framework about the ports they are providing and using
  - Framework will not allow connections to ports it isn’t aware of
- Register them using methods on the component’s `gov.cca.Services` object
  - `addProvidesPort()` and `removeProvidesPort()`
  - `registerUsesPort()` and `unregisterUsesPort()`
  - All are defined in the CCA specification
- Ports are usually registered in the component’s `setServices()` method
  - Can also be added/removed dynamically during execution

Diagram:

```
<table>
<thead>
<tr>
<th>MidpointIntegrator</th>
<th>FunctionPort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FunctionPort</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NonlinearFunction</td>
</tr>
</tbody>
</table>
```
CCA Concepts: Language Interoperability

- Scientific software is increasingly diverse in use of programming languages
- In a component environment, users should not care what language a component is implemented in
- “Point-to-point” solutions to language interoperability are not suitable for a component environment
- The Babel language interoperability tool provides a common solution for all supported languages
- Scientific Interface Definition Language provides language-neutral way of expressing interfaces

More on Babel later!
Coding in a CCA Environment

Port Definitions (SIDL)

Babel compiler (SIDL→language)

Component Definition (SIDL)

CCA Framework

Compiled Components (object libraries)

Application (component assembly)

Language compiler & linker

Generated language code

Babel runtime library & Chasm F90 array library

More details in the CCA Tools module

Key:

User code
CCA Tools
Generated code
Standard Tools
Object libraries
CCA Supports Parallelism -- by “Staying Out of the Way” of it

- Single component multiple data (SCMD) model is component analog of widely used SPMD model
- Each process loaded with the same set of components wired the same way
- Different components in same process “talk to each” other via ports and the framework
- Same component in different processes talk to each other through their favorite communications layer (i.e. MPI, PVM, GA)

Components: Blue, Green, Red
Framework: Gray

Any parallel programming environments that can be mixed outside of CCA can be mixed inside
“Multiple-Component Multiple-Data” Applications in CCA

• Simulation composed of multiple SCMD sub-tasks

• Usage Scenarios:
  – Model coupling (e.g. Atmosphere/Ocean)
  – General multi-physics applications
  – Software licensing issues
    • i.e. limited number of instances

• Approaches
  – Run single parallel framework
    • Driver component that partitions processes and builds rest of application as appropriate (through BuilderService)
  – Run multiple parallel frameworks
    • Link through specialized communications components
    • Link as components (through AbstractFramework service)
MCMD Within A Single Framework

See example in the *Using CCA* module (multilevel parallelism in quantum chemistry)

- Framework
- Application driver & MCMD support component
- Components on all processes
- Components only on process group A
- Components only on process group B

Group A

Group B
CCA Supports High-Performance Local and Distributed Computing

• “Direct connection” preserves high performance of local (“in-process”) and parallel components
  • Framework makes connection
  • But is not involved in invocation

• Distributed computing has same uses/provides pattern, but framework intervenes between user and provider
  • Framework provides a proxy provides port local to the uses port
  • Framework conveys invocation from proxy to actual provides port
“Direct Connection” Details

• Directly connected components are in the same address space
  – Data can be passed by reference instead of copying
  – Just like “traditional” programs
  – Framework involved in connecting components, but not invocations on ports

• Cost of “CCAness” in a direct connect environment is a level of indirection on calls between components
  – Equivalent to a C++ virtual function call: lookup function location, invoke it
  – Overhead is on the invocation only (i.e. latency), not the total execution time
  – Cost equivalent of ~2.8 F77 or C function calls
  – ~48 ns vs 17 ns on 500 MHz Pentium III Linux box
Performance, the Big Picture

Direct-Connect, Parallel
• No CCA overhead on…
  – calls within component
  – parallel communications across components
• Small overheads on invocations on ports
  – Virtual function call (CCAness)
  – Language Interoperability (some data types)

Distributed
• No CCA overhead on calls within component
• Overheads on invocations on ports
  – Language interoperability (some data types)
  – Framework
  – (Wide area) network
Maintaining HPC Performance

• The performance of your application is as important to us as it is to you

• The CCA is designed to provide maximum performance
  – But the best we can do is to make your code perform no worse

• Facts:
  – Measured overheads per function call are low
  – Most overheads easily amortized by doing enough work per call
  – Other changes made during componentization may also have performance impacts
  – Awareness of costs of abstraction and language interoperability facilitates design for high performance

More about performance in notes
Some Performance Results and References


Maximum 0.2% overhead for CCA vs native C++ code for parallel molecular dynamics up to 170 CPUs.
Overhead from Component Invocation

- Invoke a component with different arguments
  - Array
  - Complex
  - Double Complex
- Compare with f77 method invocation
- Environment
  - 500 MHz Pentium III
  - Linux 2.4.18
  - GCC 2.95.4-15
- Components took 3X longer
- Ensure granularity is appropriate!
- Paper by Bernholdt, Elwasif, Kohl and Epperly

<table>
<thead>
<tr>
<th>Function arg type</th>
<th>f77</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>80 ns</td>
<td>224ns</td>
</tr>
<tr>
<td>Complex</td>
<td>75ns</td>
<td>209ns</td>
</tr>
<tr>
<td>Double complex</td>
<td>86ns</td>
<td>241ns</td>
</tr>
</tbody>
</table>
Scalability: Component versus Non-component. I

- Quantum chemistry simulation
- Sandia’s MPQC code
  - Both componentized and non-componentized versions
- Componentized version used TAO’s optimization algorithms
- Problem: Structure of isoprene HF/6-311G(2df,2pd)

Parallel Scaling of MPQC w/ native and TAO optimizers
Scalability: Component versus Non-component. II

- Hydrodynamics; uses CFRFS set of components
- Uses GrACEComponent
- Shock-hydro code with no refinement
- 200 x 200 & 350 x 350 meshes
- Cplant cluster
  - 400 MHz EV5 Alphas
  - 1 Gb/s Myrinet
- Negligible component overhead
- Worst perf: 73% scaling efficiency for 200x200 mesh on 48 procs

Advanced CCA Concepts

- The Proxy Component pattern *(Hands-On Ch. 6, papers)*

- Component lifecycle *(tutorial notes, Hands-On)*

- Components can be dynamic *(papers)*

- Frameworks can provide a specialized programming environment *(papers)*

Brief introductions only, but more info is available – just ask us!
The Proxy Component Pattern

- Component interfaces offer an obvious place to collect information about method invocations for performance, debugging, or other purposes
  - No intrusion on component internals
- A "proxy" component can be inserted between the user and provider of a port without either being aware of it
- Proxies can often be generated automatically from SIDL definition of the port

Sample uses for proxy components:
- Performance: instrumentation of method calls
- Debugging: execution tracing, watching data values
- Testing: Capture/replay

Performance Monitoring with TAU

Before:

After:
Component Lifecycle

• Composition Phase (assembling application)
  – Component is instantiated in framework
  – Component interfaces are connected appropriately

• Execution Phase (running application)
  – Code in components uses functions provided by another component

• Decomposition Phase (termination of application)
  – Connections between component interfaces may be broken
  – Component may be destroyed

In an application, individual components may be in different phases at different times
Steps may be under human or software control
User Viewpoint: Loading and Instantiating Components

- Components are code + metadata
- Using metadata, a **Palette** of available components is constructed
- Components are instantiated by user action (i.e. by dragging from Palette into Arena)
- Framework calls component’s constructor, then `setServices`

- Details are **framework-specific**!
- **Caffeine** currently provides both command line and GUI approaches

Supplementary material for handouts
User Connects Ports

- Can only connect uses & provides
  - Not uses/uses or provides/provides
- Ports connected by type, not name
  - Port names must be unique within component
  - Types must match across components
- Framework puts info about provider of port into using component’s Services object
Component’s View of Instantiation

- Framework calls component’s constructor
- Component initializes internal data, etc.
  - Knows *nothing* outside itself

Framework interaction code

```java
CCA.Services
  provides IntegratorPort
  uses FunctionPort,
  RandomGeneratorPort
```

- Framework calls component’s `setServices`
  - Passes `setServices` an object representing everything “outside”
  - `setServices` declares ports component *uses* and *provides*
- Component *still* knows nothing outside itself
  - But Services object provides the means of communication w/ framework
- Framework now knows how to “decorate” component and how it might connect with others

```
MonteCarloIntegrator
  IntegratorPort
  FunctionPort
  RandomGeneratorPort

MonteCarloIntegrator
```

Supplementary material for handouts
Component’s View of Connection

- Framework puts info about provider into user component’s Services object
  - MonteCarloIntegrator’s Services object is aware of connection
  - NonlinearFunction is not!
- MCI’s integrator code cannot yet call functions on FunctionPort
Component’s View of Using a Port

- User calls `getPort` to obtain (handle for) port from `Services`
  - Finally user code can “see” provider
- **Cast** port to expected type
  - OO programming concept
  - Insures type safety
  - Helps enforce declared interface
- **Call** methods on port
  - e.g.
    ```
    sum = sum + function->evaluate(x)
    ```
- Call `releasePort`
Components can be Dynamic

- **gov.cca.BuilderService** allows programmatic composition of components
  - Components can be instantiated/destroyed, and connected/disconnected under program control

Sample uses of *BuilderService*:

- Python “driver” script which can assemble and control an application
  - i.e. MCMD climate model

- Adaptation to changing conditions
  - Swap components in and out to give better performance, numerical accuracy, convergence rates, etc.

- Encapsulation of reusable complex component assemblies
  - Create a “container component” which exposes selected ports from the enclosed components
Frameworks can Provide Specialized Parallel Programming Environments

- By definition, all execution of components takes place within a framework
- CCA does not dictate a particular approach to parallelism
- Therefore, a specialized parallel programming environment can be made part of a CCA framework
  - May simplify design
  - Components depending on it won’t be useable in other frameworks, even if they are also CCA-compliant

Example:
- Uintah Computational Framework, based on SCIRun2 (Utah) provides a multi-threaded parallel execution environment based on task graphs
  - Graphs express interdependencies of each task’s inputs and outputs
  - Specialized to a class of problems using structured adaptive mesh refinement
Is CCA for You?

• Much of what CCA does can be done without such tools if you have sufficient discipline
  – The larger a group, the harder it becomes to impose the necessary discipline

• Projects may use different aspects of the CCA
  – CCA is not monolithic – use what you need
  – Few projects use all features of the CCA… initially

• Evaluate what your project needs against CCA’s capabilities
  – Other groups’ criteria probably differ from yours
  – CCA continues to evolve, so earlier evaluations may be out of date

• Evaluate CCA against other ways of obtaining the desired capabilities

• Suggested starting point:
  – CCA tutorial “hands-on” exercises
Take an Evolutionary Approach

- The CCA is designed to allow selective use and incremental adoption

- “SIDLize” interfaces incrementally
  - Start with essential interfaces
  - Remember, only externally exposed interfaces need to be Babelized

- Componentize at successively finer granularities
  - Start with whole application as one component
    - Basic feel for components without “ripping apart” your app.
  - Subdivide into finer-grain components as appropriate
    - Code reuse opportunities
    - Plans for code evolution
View it as an Investment

• CCA is a long-term investment in your software
  – Like most software engineering approaches

• There is a cost to adopt

• The payback is longer term

• Remember Biggerstaff’s Rule of Threes
  – Look at three systems, requires three times the effort, payback after third release
CCA is Still Under Development

• We’ve got…
  – A stable component model
  – Working tools
  – Active users

• But…
  – We know it’s not perfect
  – We’re not “done” by any stretch

• Talk to us…
  – If you’re evaluating CCA and need help or have questions
  – If you don’t think CCA meets your needs
  – If you’ve got suggestions for things we might do better
What Can CCA Do Today?

• Ccaffeine framework for HPC/parallel
  – XCAT and other options for distributed computing

• Language interoperability
  – Fortran 77/90/95, C, C++, Java, Python
  – Support for Fortran/C user-defined data structures under development

• CCA Tools working on a variety of platforms
  – Linux most widely used
  – Mac OS X second
  – Some IBM AIX users
  – Ports in progress for Cray X1 and XT3
  – Porting is driven by user needs
CCA Tools – Language Interoperability and Frameworks
Goal of This Module

• Describe tools for multi-lingual, scientific component ‘plug-and-play’
CCA adds value to component development

- **Port Definitions (SIDL)**
- **Component Definition (SIDL)**
- **Component source code**
- **Language compiler & linker**
- **Compiled Components (object libraries)**
- **Application (component assembly)**
  - **CCA Framework**
  - **CCA IDE**

Key:
- **User code**
- **CCA Tools**
- **Standard Tools**
- **Generated code**
- **Object libraries**
Tools Module Overview

- Language interoperability tools
- Frameworks
- CCA Interactive Development Environment
Babel Facilitates Scientific Programming Language Interoperability

- Programming language-neutral interface descriptions
- Native support for basic scientific data types
  - Complex numbers
  - Multi-dimensional, multi-strided arrays
- Automatic object-oriented wrapper generation

Supported on Linux, AIX, and Solaris 2.7, works on OSX;
C (ANSI C), C++ (GCC), F77 (g77, Sun f77), F90 (Intel, Lahey, GNU, Absoft), Java (1.4)
Babel Generates Object-Oriented Language Interoperability Middleware

1. Write your SIDL file to define public methods
2. Generate server side in your native language using Babel
3. Edit Implementations as appropriate
4. Compile and link into library/DLL

IOR = Intermediate Object Representation  
SIDL = Scientific Interface Definition Language
Clients in any supported language can access components in any other language.

IOR = Intermediate Object Representation

IORs

Skeletons

Implementations

Component
(any supported language)
Recent and Upcoming Features

• Remote Method Invocation (BabelRMI) \textit{ALPHA}
• Design-by-Contract \textit{ALPHA}
• Pre- and post-method invocation hooks \textit{ALPHA}
Chasm Provides Language Interoperability for Fortran, C, and C++

- Extracts interfaces from C/C++ and Fortran90 codes
- Uses library of XSLT stylesheets for language transformations → easily extended
  - Generates XML and SIDL representations
  - Generates Fortran90 Babel implementation glue
- Provides Fortran array descriptor library used by Babel

Supported on Linux, AIX, and Solaris 2.7, works on OSX; C (ANSI C), C++ (GCC), F90 (Intel, Lahey, GNU, Absoft)
The Entire **Chasm** Process Involves Three Basic Steps

1. Start with a Fortran (or C/C++) source file
2. Create an XML description of the component (or port)
3. Generate the SIDL specification and glue code files

**XML** = Extensible Markup Language  
**PTD** = Parse Tree Dump
Chasm-Assisted Glue Code Generation

1. Create functions_LinearFunction.impl.xml
2. Create xml description of source procedures
   `gfortran -fdump-parse-tree LinearFunction.f90 > LinearFunction.ptd`
   `gfortran2xml < LinearFunction.ptd > LinearFunction.xml`
3. Create .sidl, _Impl.F90, and _Mod.F90
   `xalan -o functions_LinearFunction.sidl functions_LinearFunction.impl.xml cca-f90-comp.sidl.xsl`
   `xalan -o functions_LinearFunction_Impl.F90 functions_LinearFunction.impl.xml cca-f90.impl.xsl`
   `xalan -o functions_LinearFunction_Mod.F90 functions_LinearFunction.impl.xml cca-f90.mod.xsl`
4. Run Babel...
User-Created XML
Component Description File

```
<componentImplementation name="LinearFunction" package="functions">
    <language name="F90">
        <property name="impl-scope" value="LinearFunction"/>
        <property name="impl-xml" value="/home/cca/LinearFunction.xml"/>
        
        <ports>
            <provides name="FunctionPort" package="function">
                <MethodsBlock>
                    <Method name="evaluate" impl="evaluate_lf"/>
                </MethodsBlock>
            </provides>
        </ports>
    </language>
</componentImplementation>
```
Recent and Upcoming Features

• Generate Fortran 2003 MPI Bindings 1Q 2006

• Update XML processor and generator to new PDToolkit releases 1Q 2006
Tools Module Overview

- Language interoperability tools
- Frameworks
- CCA Interactive Development Environment
Frameworks are Specialized to Different Computing Environments

• "Direct connection" preserves high performance of local ("in-process") and parallel components
  • Framework makes connection
  • Framework not involved in invocation

• Distributed computing has same uses/provides pattern, but the framework intervenes between user and provider
  • Framework provides a proxy port local to the user’s uses port
  • Framework conveys invocation from proxy to actual provides port
Graphical User Interfaces (GUIs) Deliver Plug-and-Play Experience

• Plug & play for:
  – Application software assembly
  – Visualization pipelines
  – Workflow management

• Assembling “wiring” diagrams is almost universal.
  – Software assembly: Ccaffeine, XCAT, SciRUN
  – Workflow: XCAT, SciRUN
  – Visualization: SciRUN

None of these (yet) plug into your favorite Integrated Development Environment (e.g., Eclipse, MS Dev. Studio, Java Studio, …).
Ccaffeine is a Direct-Connect, Parallel-Friendly Framework

- Supports SIDL/Babel components
  - Conforms to latest CCA specification (0.7)
  - Also supports legacy CCA specification (0.5)
    - Any C++ allowed with C and Fortran by C++ wrappers

- Provides command-line and GUI for composition
  - Scripting supports batch mode for SPMD
  - MPMD/SPMD custom drivers in any Babel language

Supported on Linux, AIX, OSX and is portable to modern UNIXes.
Ccaffeine Involves a Minimum of Three Steps

• As easy as 1-2-3:
  – Write your component.
  – Describe it with an XML file.
  – Add a line to your Ccaffeine input file to load the component at runtime.

• Proceed to plug & play...

There are many details and automated tools that help manage components.
**Ccaffeine GUI**

- **Process**
  - User input is broadcast SPMD-wise from Java.
  - Changes occur in GUI *after* the C++ framework replies.
  - If your components are computing, GUI changes are blocked.

- **Components interact through port connections**
  - *provide* ports implement class or subroutines
  - *use* ports *call* methods or subroutines in the port.
  - Links denote caller/callee relationship *not* data flow

![Diagram of port connections between MidpointIntegrator, FunctionPort, and NonlinearFunction]
User Connects Ports

- Can only connect *uses* & *provides*
  - *Not* uses/uses
  - *Not* provides/provides
- Ports connected by type not name
  - Port names must be unique within component
  - Types must match across components
- Framework puts info about *provider* of port into *using* component’s Services object
Building an Application (1 of 2)

- Components are code + XML metadata
- Using metadata, a **Palette** of available components is constructed.
- Components are instantiated by user action (i.e. by dragging from **Palette** into **Arena**).
- Framework calls component’s **constructor**, then **setServices**
Building an Application (2 of 2)

1. Click *Configure* port to start parameter input dialogue.

2. For each connection: click a *uses* port then click a *provides* port to establish a connection.

3. Click *Go* port to start the application.

Right-clicking a connection line breaks the connection -- enabling component substitution.
Application Configurations can be Re-used

1. Click **Save** or **Save As…** to save actions.

2. Click **Open** to replay actions.

   - **Script optimization**
     
     ```bash
     % simplify-bld saved_file.bld > faster_file.bld
     ```

   - **Batch conversion**
     
     ```bash
     % bld2rc faster_file.bld > faster_file.batch
     ```

   - **C++ stand-alone execution**
     
     ```bash
     % bld2babel-cpp faster_file.bld faster_file_babel outdir
     or % bld2neo faster_file.bld faster_file.batch outdir
     ```
Recent and Upcoming Features

- Interoperate with other CCA frameworks
  - Via Babel RMI 2H 2006
**XCAT-C++ Framework**

**XCAT is a Web-services based Distributed Component Framework**

- Remote references
  - Port types described in C++ header files or in WSDL documents

- User Interface
  - C++ and Python interface to CCA BuilderService
  - Uses SWIG for Python-C++ translations

- Component creation
  - Remote creation via SSH

- Component communication
  - Proteus multi-protocol library
  - Communication libraries can be loaded at run-time

Tested on Linux.

WSDL = Web Service Definition Language
XCAT is Designed for High-Performance Scientific Applications
Basic How-To

1. Define port interfaces as C++ header files or WSDL docs
2. Indicate ports used by each component in a config file
3. Run scripts to generate code for stub-skeletons (for ports)
   • Can also generate component-templates for new components
   • Use component-templates to convert a scientific library into a CCA component
4. Build components using XCAT-C++ make scripts
5. Deploy component executables on the target remote hosts
   • Also set up SSH access to remote hosts
6. Write python scripts (edit examples) to use CCA API to connect components and invoke a Go port
   • Alternatively, can use a C++ front-end
7. Execute the python script (or C++ front-end)
Recent and Upcoming Features

- Support GRAM for component creation 1H 2006
  - Allow use of grid resources

- Automated component registration and discovery 2H 2006

- Support new protocols such as UDT (in Proteus) 1H 2006

- Support Babel’s Remote Method Invocation 2H 2006
  - Allows access to Babel objects through remote Babel stubs
  - Provides direct support for SIDL in distributed applications
  - Leverages Proteus

GRAM = Grid Resource Allocation Management  
UDT = UDP-based Data Transfer protocol
**SCIRun2 is a Cross-Component Model, Distributed Component Framework**

- Semi-automatically bridges component models
  - Templated components connected at run-time generate bridges
- Parallel Interface Definition Language (PIDL) – a SIDL variant
- User interface – GUI and textual
  - Dynamic composition
- Component and framework creation
  - Remote via SSH
- Component communication
  - C++ RMI with co-location optimization
  - MPI/ Parallel Remote Method Invocation (PRMI)

Supported on Linux.
Master Framework Coordinates Slave Frameworks in Each Remote Address Space
Basic How-To

1. Add component source files and makefile to SCIRun2 sources
   - May need to define ports in SIDL
2. Add component information to the component model xml file
3. Build component using SCIRun2 make scripts
   - Alternatively, build component using Babel
4. Start the framework and graphical (default) or text builder
5. Graphically connect component to other CCA-based or non CCA-based components
   - May need to create bridge components to go between models
6. Press the “Go” button on the driver component
Simple SCIRun2 CCA (PIDL) and Babel Bridge
Recent and Upcoming Features

• Merge PIDL with SIDL/Babel 1H 2005

• Support additional component models
  – Kepler workflows 1H 2006

• Support Babel’s Remote Method Invocation PRMI 2H 2006

• Automate bridging On-going
## Experimental Frameworks

<table>
<thead>
<tr>
<th>Framework</th>
<th>Purpose</th>
<th>Summary</th>
</tr>
</thead>
</table>
| Distributed CCA Framework (DCA) | MxN research | • **Goal:** explore MxN Parallel-Remote Method Invocation (PRMI) using MPI  
  • Parallel data transfer and redistribution integrated into port invocation mechanism |
| LegionCCA | Grid-based research | • **Goal:** allow component-based CCA applications to run in Grid-scale environments using Legion  
  • Supports creation, scheduling, persistence, migration, and fault notification; relies on Legion’s built-in RPC mechanism (~Unix sockets) |
| XCAT-Java | Globus-based Grid research | • **Goal:** explore web interface for launching distributed applications  
  • This (alpha) version compatible with latest CCA specification and provides built-in seamless compatibility with OGSI. |

OGSI = Open Grid Services Infrastructure
Tools Module Overview

- Language interoperability tools
- Frameworks

• CCA Interactive Development Environment
Component Development Environment
Provided via Eclipse Plug-ins

• Provides a high-level graphical environment
  – Creating new SIDL-based components
  – Componentizing legacy codes
    • C, C++, Java and Fortran

• Automatic code generation

Supported on Linux, Windows, MacOS.
Component Development Environment Starts at the Eclipse Platform Level

Plug-ins for:
- SIDL Editor
- Wizards
- Preliminary automated build support

Imperative that you start by creating a new project!
Wizards are Available for Adding Packages and Classes or Generating SIDL from Legacy Codes

- Intuitive interfaces to port and component definition
- Helper wizards for setting port, component and (in the future) application properties
A Wizard is also Available for Adding Methods

CCA IDE
Supplementary material for notes
Recent and Upcoming Features

- Provide automated build support  
  
- Launch application via GUI  

CCA IDE
Supplementary material for notes
## CCA Tools Contacts (1 of 2)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Purpose</th>
<th>More information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Babel</td>
<td><strong>Scientific</strong> language interoperability tool kit</td>
<td>URL: <a href="http://www.llnl.gov/CASC/components">www.llnl.gov/CASC/components</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: <a href="mailto:components@llnl.gov">components@llnl.gov</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or <a href="mailto:babel-users@lists.llnl.gov">babel-users@lists.llnl.gov</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: Ben Allan, <a href="mailto:ccafe-help@z.ca.sandia.gov">ccafe-help@z.ca.sandia.gov</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wiki: <a href="https://www.cca-forum.org/wiki">https://www.cca-forum.org/wiki</a></td>
</tr>
<tr>
<td>Chasm</td>
<td>Fortran90 interoperability wrapper</td>
<td>URL: <a href="http://chasm-interop.sourceforge.net">chasm-interop.sourceforge.net</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples: chasm/example/cca-tutorial.</td>
</tr>
<tr>
<td>DCA</td>
<td>MxN <strong>research</strong> framework</td>
<td>URL: <a href="http://www.cs.indiana.edu/~febertra/mxn">www.cs.indiana.edu/~febertra/mxn</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: Felipe Bertrand, <a href="mailto:febertra@cs.indiana.edu">febertra@cs.indiana.edu</a></td>
</tr>
<tr>
<td>CCA IDE</td>
<td>CCA development environment</td>
<td>Email: <a href="mailto:usability@cca-forum.org">usability@cca-forum.org</a></td>
</tr>
</tbody>
</table>
# CCA Tools Contacts (2 of 2)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Purpose</th>
<th>More information</th>
</tr>
</thead>
<tbody>
<tr>
<td>LegionCCA</td>
<td>Grid-based research framework</td>
<td>URL: grid.cs.binghamton.edu/projects/legioncca.html</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: Michael J. Lewis, <a href="mailto:mlewis@binghamton.edu">mlewis@binghamton.edu</a></td>
</tr>
<tr>
<td>SCIRun2</td>
<td>Cross-component model framework</td>
<td>URL: <a href="http://www.sci.utah.edu/">www.sci.utah.edu/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: Steve Parker, <a href="mailto:sparker@cs.utah.edu">sparker@cs.utah.edu</a></td>
</tr>
<tr>
<td>XCAT-C++</td>
<td>Globus-based GRID framework</td>
<td>URL: grid.cs.binghamton.edu/projects/xcat/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: Madhu Govindaraju, <a href="mailto:mgovinda@cs.binghamton.edu">mgovinda@cs.binghamton.edu</a></td>
</tr>
<tr>
<td>XCAT-Java</td>
<td>Grid research framework</td>
<td>URL: <a href="http://www.extreme.indiana.edu/xcat/">www.extreme.indiana.edu/xcat/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Email: Dennis Gannon, <a href="mailto:gannon@cs.indiana.edu">gannon@cs.indiana.edu</a></td>
</tr>
</tbody>
</table>
Module Summary

- Described tools for multi-lingual, scientific component ‘plug-and-play’
  - Language interoperability through Babel and Chasm
  - CCA Frameworks provide mechanisms for composition
  - CCA Interactive Development Environment via Eclipse plug-in
Goal of This Module

Learn how existing code is

• Wrapped into Babel objects, and
• Promoted to CCA components

In the process, also need to learn about

• Scientific Interface Definition Language (SIDL)
• Using the Babel tool
• Characteristics of Babelized software
Working Code: “Hello World” in F90
Using a Babel Type

```fortran
program helloclient
    use greetings_English
    implicit none
    type(greetings_English_t) :: obj
    character (len=80) :: msg
    character (len=20) :: name

    name='World'
    call new( obj )
    call setName( obj, name )
    call sayIt( obj, msg )
    call deleteRef( obj )
    print *, msg
end program helloclient
```
Handout Material: Code Notes

1. Use statement for the greetings. English type
2. Obj is a F90 derived type we get from the using statement, note the “_t” extension that prevents it from colliding with the using statement.
3. In C/C++ examples, this variable would be initialized by a the command-line variable “argv[1]”, but its trickier to do portably in F90 and too long, so I just initialize the name to “World”.
4. Obj is not yet initialized. The Babel idiom in F90 is to call new() to initialize the Babel type. In other languages its _create(). NOTE: good code would add error checking.
5. setName() puts data into the obj. It sets its state.
6. sayIt() returns the entire greeting including the aforementioned name.
7. deleteRef() is a subroutine that all Babel types inherit from a parent class. All Babel objects are reference counted. When there are no more outstanding references, the object is told to clean up after itself.
Working Code: “Hello World” in F90 Using a Babel Type

```fortran
program helloclient
    use greetings_English
    implicit none
    type(greetings_English_t) :: obj
    character (len=80) :: msg
    character (len=20) :: name
    name='World'
    call new(obj)
    call setName(obj, name)
    call sayIt(obj, msg)
    call deleteRef(obj)
    print *, msg
end program helloclient
```

- Looks like a native F90 derived type
- These subroutines were specified in the SIDL.
- Other basic subroutines are “built in” to all Babel types.
The SIDL File that defines the “greetings.English” type

```idl
package greetings version 1.0 {
    interface Hello {
        void setName( in string name );
        string sayIt();
    }
    class English implements-all Hello {
    }
}
```
Packages contain user-defined types and are used to reduce naming collisions. Packages can be nested.

Packages can be versioned. User defined types must be nested inside a versioned package and gain the same version number as the innermost versioned package.

SIDL has a inheritance model similar to Java and Objective C. Classes can inherit multiple interfaces, but at most one implementation (other class).

An interface describes an API, but doesn’t name the implementation.

Note that arguments have mode, type, and name. Mode can be one of “in”, “out”, and “inout”. These SIDL modes have slightly different semantics than Fortran90 “intents”.

This class generates English greetings. One could imagine a strategy for internationalization that uses the Hello interface everywhere, but loads in English, French, or whatever classes based on user’s preference.
Question: What language is “obj” really implemented in?

```fortran
program helloclient
  use greetings_English
  implicit none
  type(greetings_English_t) :: obj
  character (len=80) :: msg
  character (len=20) :: name
  
  name='World'
  call new(obj)
  call setName(obj, name)
  call sayIt(obj, msg)
  call deleteRef(obj)
  print *, msg
end program helloclient
```

Answer: Can't Know!

With Babel, it could be C, C++, Python, Java, Fortran77, or Fortran90/95

In fact, it could change on different runs without recompiling this code!
CCA uses Babel for high-performance n-way language interoperability

Each one of these red lines, is potentially a bridge between two languages. No telling which language your component will be connected to when you write it.
CCA uses SIDL to specify APIs and Type Hierarchy for Frameworks, Services, Components, & Ports

- A CCA framework must
  - implement `gov.cca.AbstractFramework`,
  - provide a `gov.cca.ports.BuilderService`,
  - etc.

- A CCA port must
  - be a SIDL interface extending `gov.cca.Port`

- A CCA component must
  - be a SIDL class implementing
    `gov.cca.Component`

The CCA Specification is a SIDL file.
Babel Consists of Two Parts: Code Generator + Runtime Library

Code generator reads SIDL, and generates wrapper code...

... very sophisticated wrapper code.
Typical Workflow to **Use** a Babel type (static linkage)

1. `babel --client=F90 greetings.sidl`
2. Compile generated Code
3. Link driver, generated code ("Stubs"), Babel Runtime, and library containing Babel type
1. `babel --client=F90 greetings.sidl`
2. Compile same generated code with different flags
3. Link driver, and stubs only (both generated code and F90 stubs to Babel Runtime library)
4. Set SIDL_DLL_PATH environment variable to include relevant *.scl (or *.cca) files.
5. Actual implementations are linked in at runtime
Static vs. Dynamic Linkage

• Static
  – Least runtime overhead
  – Easiest to get right, debug, etc.

• Dynamic
  – Allows new types to “plug-in” without relinking driver
  – Necessary for Java or Python calling to other languages (unless you relink their virtual machine)
  – Induces very nondeterministic behavior if done incorrectly
Workflow for a Developer Wrapping Their Code into Babel Objects

1. `babel --server=C++ greetings.sidl`
2. Fill in the implementation details (see next slide)
3. Compile and link into a library/DLL
namespace greetings {
    class English_impl {
        private:
            // DO-NOT-DELETE splicer.begin(greetings.English._impl)
            string d_name;
            // DO-NOT-DELETE splicer.end(greetings.English._impl)

        string greetings::English_impl::sayIt() throw () {
            // DO-NOT-DELETE splicer.begin(greetings.English.sayIt)
            string msg("Hello ");
            return msg + d_name + "!";
            // DO-NOT-DELETE splicer.end(greetings.English.sayIt)
        }
    }
}
Quick Review of Babel in general before proceeding to CCA specifics

• Babel can be used as a standalone tool
• Each language binding strikes a balance
  – support the SIDL type system (OO, exceptions, etc.)
  – provide it in a manner “natural” to experienced programmers in the target language
• For more details about Babel and SIDL
  – SC|04 tutorial slides for Babel
    http://www.llnl.gov/CASC/components/docs/sc04.html
  – Babel User’s Guide (aka. the BUG)
How to write a Babelized CCA Component (1/3)

1. Define “Ports” in SIDL
   
   - CCA Port =
     
     • a SIDL Interface
     • extends gov.cca.Port

```java
package functions version 1.0 {
    interface Function extends gov.cca.Port {
        double evaluate( in double x );
    }
}
```
How to write a Babelized CCA Component (2/3)

2. Define “Components” that implement those Ports
   – CCA Component =
     • SIDL Class
     • implements gov.cca.Component (and any provided ports)

```java
class LinearFunction implements functions.Function, gov.cca.Component {
    double evaluate( in double x );
    void setServices( in cca.Services svcs );
}
```

```java
class LinearFunction implements-all
    functions.Function, gov.cca.Component {
}
```
Tip: Use Babel’s XML output like precompiled headers in C++

1. Precompile SIDL into XML using ‘--text=xml’
2. Store XML in a directory
3. Use Babel’s –R option to specify search directories
How to write a Babelized CCA Component (3/3)

3. Use Babel to generate the interoperability glue
   – Execute `babel --server=C --R Repo functions.sidl`

4. Fill in Implementations as needed
Review: Goal of This Module

Learn how existing code is

- Wrapped into Babel objects, and
- Promoted to CCA components

In the process, also need to learn about

- Scientific Interface Definition Language (SIDL)
- Using the Babel tool
- Characteristics of Babelized software
Contact Information

• Project: http://www.llnl.gov/CASC/components

• Project Team Email: components@llnl.gov

• Mailing Lists: majordomo@lists.llnl.gov
  subscribe babel-users [email address]
  subscribe babel-announce [email address]

• Bug Tracking: https://www.cca-forum.org/bugs/babel/
  or email to babel-bugs@cca-forum.org
Strategies for Developing with Components
Modern Scientific Software Development

• Complex codes, often **coupling** multiple types of physics, time or length scales, involving a broad range of computational and numerical techniques
• Different parts of the code require significantly **different expertise** to write (well)
• Generally written by **teams** rather than individuals
Developing Components
(Both New Codes and Wrappers to Existing Codes)

• Productivity Benefits
• Application Decomposition Strategies
• Interface Design Issues
  – Social factors
  – Technical factors
• Implementation Issues and Patterns
CCA Productivity Benefits

- Fast algorithmic experiments and benchmarks by substituting components
- Once ports are defined, domain-expert component implementers can work separately in their own favorite languages
- Work of transient contributors remains as well-defined, lasting components
- Wrapped legacy portions need not be reimplemented or reverified
Components in the Small: Impacts within a Project

Benefits include:

• Rapid testing, debugging, and benchmarking
• Support for implementation-hiding discipline
• Coordination of independent workers
• Interface change effects across components are clear and usually automatically found by compilers if overlooked
• Object-orientation made simpler for C and Fortran
Components in the Large: Connecting Multiple Projects

Benefits include:

• SIDL can be used to facilitate the interface consensus processes
• Different sub-projects do not have to agree on one implementation language
• Developers who never meet in person have an excellent chance of code integration working on the first try

Costs include:

• Consensus can be expensive to obtain
• Writing code for others to use is more difficult than writing it just for yourself
Application Decomposition Strategies

- Conceptually decompose the application into
  - cutting-edge areas (less stable)
  - areas that can employ existing component-based libraries (more stable)
- Decompose each area into components for
  - physics
  - mathematics
  - data management
  as dictated by the application; sketch a typical component layout
- Many components will encapsulate algorithmic logic only, with little or no private data
- Most HPC applications will have a central data abstraction that provides data memory management and parallel communication
- In a multilanguage application, all I/O may need to be isolated into components written in a single common language (file based I/O should not be affected)
- Component boundaries (and port interfaces) may be set to isolate proprietary code or difficult contributors
Interface Design: Social Factors
(Defining Ports to Wrap Existing Code)

• Will the port hide just one implementation, or will there need to be plug compatibility with other implementations? From other teams?

• Who defines the interface and maintains it?
  1. Project dictator? (Fast)
  2. The owner of the legacy functionality? (Slow, if not you)
  3. A standards committee? (Really slow)

• How many iterations of redefining the ports will the customers tolerate?
Interface Design: Technical Factors

- Do we make a single large port look like the underlying library or divide functions into groups on separate ports?
- Should a function with many optional arguments be split into several alternative functions with simpler usage?
- Do we make the ports more general than the existing code?
- Do we require the ports to work across languages? Across networks?
  - If not, gains in efficiency or coding ease might be had
  - If so, memory management and I/O challenges may arise
Implementation Issues in Wrapping

• Do we split large libraries into several components?
  – Splitting is difficult to do if global variables or common blocks are widely used.

• Do we expect more than one implementation instance of a port in a single run-time?
  – If not, interface contracts may include global side effects

• Do we integrate the wrapper code in the existing code’s development and build processes?
  – If not, how do we ensure build consistency and on-going wrapper support?

• Code bases with large interfaces need automated wrapping tools
  – E.g., see Chasm info in the Tools module of the tutorial
Benefits of Wrapping Code Using CCA

- Setting a language-neutral interface definition (SIDL) can greatly clarify design discussions
- Provides a chance to reorganize the interface and hide globals
- Allows testing of alternate versions if doing performance studies
- Allows easy “experimentation” with new algorithms
- Software discipline is enforced, not optional
- Implementation decisions (to split libraries, etc) can be easily revised over time if interfaces remain constant (possibly with the addition of new interfaces)
Interface Design for New Code

- Write SIDL for each connection (port) in the sketched component layout
- If two ports must always be used together, consider merging them
- Review SIDL drafts for near-duplication of ports
- Avoid creating interface contracts that require using hidden global data
- Consider exporting tuning and/or configuration parameter inputs as a port
- All the design issues from wrapping existing code apply, also
- *Interfaces will change.*
Recommended Implementation Patterns

• Expect to decompose initial components further as work progresses and requirements expand

• Build systems (make) should be kept as simple as possible
  – Keep a subdirectory for port definitions and any implementation-independent glue code derived from the ports
  – Keep each component (and any wrapped code) in its own subdirectory
  – Keep application-wide flags in a configure script or an include file common to all components and ports
  – Consistency is key. Extract build flags from cca-specbabel-config and if possible compile & link with babel-libtool
Case Study:
The NWChem Parallel Computational Chemistry Package

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
How I Know About this Project

• I was an architect and developer of NWChem from the beginning of the project in 1993 until about 2000

• I continue to interact with the NWChem developers and have been involved in discussions of architectural issues around a successor to NWChem
Background: Quantum Chemistry

• A relatively small number of very large, long-lived packages
  – Originate in a single research group
  – Collaborators, former students/postdocs at other sites add to developer base

• Packages tend to be large (100k-1M+ lines) and provide a wide range of quantum chemistry methods
  – Substantially the same core capabilities implemented in each
  – Cutting edge capabilities vary by research focus of group

• Many packages have been under continual development since the 1970s or late 1960s

• Typical architecture is either monolithic or a collection of multiple monolithic executables
  – Somewhat like filter-and-pipe architecture
Common Computational Chemistry Methods

- **Molecular dynamics**
  - Ball and spring model for atomic interactions (no quantum mechanics)
  - Parameterized from expt or high-quality theoretical data
  - Cost scales as $N^2$
  - Useful for large molecules (proteins, drug design, etc.)

- **Hartree-Fock, Self-Consistent Field, Density Functional Theory**
  - Simplest model including quantum mechanics
  - Results are often qualitative, occasionally quantitative
  - Cost scales as $N^4$
  - DFT popular, HF/SCF used less often

- **Perturbation theory (MP2, MP4)**
  - Increasingly sophisticated quantum mechanical models
  - Results often quantitative
  - Costs scale as $N^5$ to $N^7$

- **Coupled cluster theory, configuration interaction**
  - Most sophisticated quantum mechanical models
  - Results can be compared to experiment
  - Costs scale as $N^6$ to $N!$
Background: Parallel Computational Chemistry (in 1993)

- Parallel computing was highly experimental in chemistry
  - A few practitioners, but not a primary research focus
  - Some publications
  - No mainstream codes

- A variety of parallel algorithms had been published
  - Primarily the most basic methods. Important but more complex methods not yet explored
  - Mostly replicated data, few fully distributed
  - None truly scalable
The Task

• The US Dept. of Energy was establishing a basic research facility to develop new technologies for environmental clean up
  – Located at the Pacific Northwest National Laboratory (PNNL)
  – Both experimental and theoretical work recognized as important

• Needed a leading edge computational chemistry capability
  – Would run on a series of large parallel computers at PNNL and elsewhere
  – Specified by methods and capability, not science results per se

• Must provide the “standard” methods of quantum chemistry
  – Also molecular dynamics

• Must allow larger calculations, not just faster calculations
  – 10-100x capabilities of the time
  – Replicated data approaches insufficient

• Initial timeline: 4 years
  – A short time frame for the capability required, considering need to develop new parallel algorithms
The Staff

• ~6 staff chemists
• ~8-12 chemistry postdocs at a time
  – ~2 year terms

• ~2 staff computer scientists
• ~1 staff applied mathematician
• Unusual at that time to “embed” CS/math people in an application project
  – Still relatively unusual
Considerations Guiding the Architectural Design (1)

- **Parallelism** was of paramount importance
  - Infrastructure must make everything else easy
  - Needed suitable *programming model* and tools
    - Strong evidence that *message passing* not adequate
  - No clear understanding what HPC architectures would look like in the future
    - Fully distributed
    - Distributed shared memory
    - True shared memory

- Architecture and overall software development approach must *scale with scope and pace* of development
Considerations Guiding the Architectural Design (2)

• From experience with other large chemistry codes, we know rapid development of so much capability would not work unless we were very careful about software engineering

• Interest in using object-oriented approach, but no significant experience
  – Fortran and C de facto languages for chemistry
  – C++ not well standardized, compilers immature, performance unacceptable
Considerations Guiding the Architectural Design (3)

- Design must support rapid startup of new developers
  - Development team dominated by postdocs
  - Professionally, need to publish science results
  - Probably only know Fortran, nothing about object-oriented programming
  - Probably don’t have parallel programming experience
  - Only around for 1-3 years

- Not enough time to learn a new language and meet project schedules
  - Developers know Fortran; possibly C. Nothing else
  - Writing good code requires experience

- Little existing code to reuse
  - Parallel stuff not suitable
  - Intellectual property issues complicated other potential reuse
Considerations Guiding the Architectural Design (4)

- Might High-Performance Fortran (HPF) be a suitable implementation language?
  - HPF 1.0 was just being standardized in 1993
  - Strictly data parallel – inadequate for quantum chemistry
  - Tools very immature

- Should we develop our own preprocessor or language extension?
  - Simpler for programmers than library-based approach
  - Support problems much more complex than library-based approach
  - Only one CS person on project with compiler expertise

- Should we develop our own parallel programming library?
  - Had experience among chemists as well as CS support
  - Prototypes used in previous small projects gave us some idea of what was needed
  - Library approach seemed most portable and most easily supported
  - Must be compatible with message passing
Considerations Guiding the Architectural Design (5)

- Domain has a tradition of large, monolithic codes
  - Or collection of executables communicating via files
  - Not acceptable to rely on files in HPC/parallel environment
- Framework approaches (with dynamic high-level modules) not common in 1993
  - Ultimate design is very close to modern “framework”, and some design discussions reflected very similar ideas
- Component technology not on the radar screen in 1993
The Resulting Design (1)

• Highly modularized, layered design
  – Strong separation of CS/math/chemistry functionality
  – Strong separation of different levels of chemistry functionality
  – Single executable

• Object-oriented design implemented in Fortran and C
  – Not all benefits of OO available
  – But enough benefits to make it worthwhile
  – OO programming in non-OO languages requires strong discipline of programmers
The Resulting Design (2)

• Develop **Global Array (GA) Toolkit**
  – Library-based parallel programming model
  – Global view of distributed data
  – One-sided access to distributed data
  – Aggregate Remote Memory Copy Interface (ARMCI) library as back-end

• Try to **influence HPF evolution** to better support quantum chemistry needs
  – Could systematically migrate to a future HPF version
  – Task parallelism introduced into HPF 2.0 (1996)
  – HPF never took off (compilers not adequate)

• Try to **influence MPI evolution** to better support quantum chemistry needs (as GA back-end)
  – MPI 2.0 includes one-sided communication
  – Too restricted to provide high performance for GA
The Resulting Design (3)

• Parallel programming with GA
  – **Global view** of data
  – Must explicitly recognize **NUMA environment**
  – **Get/(local) compute/put** pattern typical
  – **Synchronization** is explicit; use only where necessary

• **Code reuse greater** than originally anticipated
  – Primarily occurred **after initial 4-year timeline**
    • As parallel computational chemistry research matured
  – Research community produced other parallel codes
  – Incorporated into NWChem through collaboration or hiring student who had done the original work
  – Degree of modification/adaptation to fit into NWChem varies
Overall Structure of NWChem

- Layered
- Modular
- Separated concerns
- OO design

Following slides provide more details about each layer.
Generic Tasks

- In quantum chemistry, there are only a handful of basic computational tasks
  - Energy
  - Structure (optimize energy; involves evaluation of energies, derivatives, and sometimes second derivatives)
  - Properties (given a particular structure, evaluate observable properties of molecule)
  - ...

- Each task can be done with many different methods
  - Same workflow, but different way of evaluating energy, derivatives, etc.

- Simple interface:
  ```
  logical function task_energy(rtdb)
  integer rtdb

  ! rtdb entry task:theory identifies method to be used
  ```

- Call Molecular Calculation Modules and some lower-level (primarily initialization/finalization)
Molecular Calculation Modules

- Implement the various methods and related functionality (i.e. property evaluation)
  - (Relatively) high-level chemistry concepts
  - Anything that is common to more than one Molecular Calculation Module should be in Molecular Modeling Toolkit

- Called by generic tasks, calls lower-level utilities

- More complex interface, but primarily through run-time database parameters rather than method arguments
Molecular Modeling Toolkit

• Chemistry-related infrastructure to facilitate implementation of many different methods
  – Lower-level chemistry concepts, typically specific to simulation
• In principle, anything which is needed by more than one Molecular Calculation Module (MCM) belongs in this layer
  – Toolbox for MCM implementers
  – As new MCMs are implemented, functionality might migrate (usually requires generalization)
• Complex interfaces, data exchange primarily through method arguments
• Primary focus of OO design efforts
  – Needed to allow generality/flexibility of upper level computations
  – Unique feature – most other quantum chemistry packages have problems with this
  – Example: Originally, methods used only one basis set. So codes were written to support only one basis set. New methods evolved requiring multiple basis sets. Existing codes had to be severely hacked, or rewritten to support. NWChem design makes this trivial.
Molecular Software Development Toolkit

• General computational tools
  – Not chemistry-specific

• Many developed by NWChem team
  – Couldn’t find satisfactory tools outside
  – Designed to satisfy needs of NWChem project…
  – But extremely careful not to make them chemistry-specific

• Example: Parallel I/O
  – Portable abstraction layer over different parallel filesystems/environments
  – Provides shared file, exclusive access file, and out-of-core Global Array abstractions (Disk-Resident Arrays)

• Example: Memory Allocator
  – Designed before Fortran90 compilers were available
  – Works for both Fortran and C
  – Provides safety features to help detect writing outside of array bounds
Parallel Programming Models in Software Architectures

• Early on, we recognized that a new parallel programming model was required for NWChem

• Is a particular or a “special” parallel programming model part of the software’s architecture?

• Software Engineer might say no, but…

• In NWChem it is a central and pervasive feature that influences code design more than anything else I’ve discussed
Global Arrays: Basic Concepts

• Provides global-view distributed multi-dimensional arrays
  – Array indices are global, not local (global view)

• Provides one-sided access to data regardless of location
  – Get, put, accumulate operations
  – No coordination required with “owning” process

• Provides access to local data region
  – Allows straightforward implementation of data-parallel algorithms

• Remote memory references are more expensive than local ones (NUMA). Write code to take NUMA into consideration
  – Approach yields portable high performance
Global Array Model of Computations

Shared Object

copy to local memory

get

local memory

compute/update

put

local memory

Shared Object

copy to shared object

ornl
Oak Ridge National Laboratory
Interoperability of GA

Global Arrays and MPI are completely interoperable. Code can contain calls to both libraries.

- **Global Arrays**
  - Application programming language interface
  - Fortran 77, C, C++, Python, Babel

- **Message Passing**
  - Global operations
  - distributed arrays layer
  - Memory management, index translation

- **ARMCI**
  - Portable 1-sided communication
  - put, get, locks, etc

- **System specific interfaces**
  - LAPI, GM/Myrinet, threads, VIA,..
Effectiveness of GA Model in Chemistry Calculations: Fock Matrix Construction

\[ F_{\mu \nu} \leftarrow D_{\lambda \sigma} \{ 2 (\mu \nu|\lambda \sigma) - (\mu \lambda|\nu \sigma) \} \]

- Indices \( \mu, \nu, \lambda, \sigma \) represent basis functions (\( N \))
- \( F \) is Fock matrix, \( D \) is density matrix (\( N^2 \))
- \( (\mu \nu|\lambda \sigma) \) are “two-electron repulsion integrals” (\( N^4 \))
  - Average 500 FLOPs per integral, but wide variation
  - Computed in blocks; block sizes vary from 1 to 10,000+ integrals
  - Each integral contracts with 6 different \( D \) elements and contributes to 6 different \( F \) elements
Other (Partitioned) Global Address Space Languages/Models

• Library-based
  – Global Array Toolkit

• Complete languages
  – Co-Array Fortran
  – Unified Parallel C (UPC)
  – Titanium (Java dialect)
  – DARPA HPCS Program languages
    • Chapel (Cray)
    • Fortress (Sun)
    • X10 (IBM)
Many would classify NWChem as a framework architecture.

I do not because the high-level simulation components are generally not developed and distributed separately from the rest of the package.
NWChem Today and Tomorrow

• 13 years of continuous development
• Version 4.7
• 58 contributors in official citation
• ~1M lines of hand-written code
• ~2M lines of generated code
  – Code generation approaches will be discussed later in this course

• Still the state of the art for parallel algorithms for the methods provided

• Leading developers interested in updating to latest software development technology, theoretical advances
  – Components, code generation
  – New parallel programming models and algorithms for O(10,000-100,000) CPUs
  – “Linear scaling” methods (extract more sparsity from large problems)
Lessons Learned

• Up-front design pays off
  – Hard work, time-consuming
  – Need to acknowledge extra effort

• OO ideas can be implemented in non-OO languages
  – Requires additional care and discipline

• “Co-location” of CS and math expertise with application scientists can be extremely productive
  – Global Array Toolkit
  – PeIGS eigensolver

• Be judicious in developing specialized tools
  – Do it if there is no other way
  – Be aware of the costs, including on-going support
  – Strive for generality

• Consider developers, users, and other factors in architecture and design
Further Information

http://www.emsl.pnl.gov/docs/nwchem/
Case Study:
The Earth System Modeling Framework

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
How I Know About this Project

• The ESMF project has been collaborating with the Common Component Architecture effort since its inception
  – Bi-directional interoperability between ESMF and CCA has been demonstrated

• I and other CCA members participated in early ESMF community meetings on architecture and requirements gathering
Background: Climate Modeling

• A modest number of carefully tested codes
  – Some “community” codes
  – Others institutional
  – Academic researchers play a role in long-range method development, less in production code development

• Most codes considered treated as “operational” or “production”
  – Carefully tested against historical observations, and each other
  – Used in weather forecasting, policy-making, etc.

• Packages generally target different high-level capabilities, but require the same infrastructure

• Coupled parallel simulation architecture is common
  – Close to framework architecture
A Generic Coupled Climate Model

Modules representing different components of ecosystem, coupled at boundaries

Resolution currently used in most *global* modeling: 280 or 140 km/cell (T42 or T85)
Background: Community Codes

• A research community agrees to contribute to a common code base
  – Cooperation instead of competition on code
  – Reduce duplication of effort
  – More easily leverage work of others

• Amount of **structure and control** over the “official” code base varies by community
  – Uncontrolled environment results in chaos
  – Common to control actual code
  – New approach is to **define framework** for compatibility (interfaces, functionality)
Community Code Example: Community Climate System Model (CCSM)

- Sponsored by US National Science Foundation (NSF) and Dept. of Energy (DOE) in collaboration with NASA
- Begun in 1994 to integrate models for atmosphere, land surface, ocean, and sea ice into a comprehensive coupled simulation
- Managed by National Center for Atmospheric Research (NCAR)
- Oversight by Scientific Steering Committee
- Working Groups on software engineering and various scientific topics
- Official code is carefully controlled and tested (“production” quality)
- Researchers work on code branches
  - May or may not be adopted back into trunk
  - Evaluated based on both scientific and software engineering criteria
- CCSM consumes a significant fraction of supercomputer cycles at NCAR, ORNL, and other sites
The Earth System Modeling Framework (ESMF)

• A set of linked proposals to NASA
  – Develop a common infrastructure of libraries and utilities for use within modeling codes
  – Develop a standard superstructure to allow composition of modeling modules into applications

• Take “community code” idea to the next level

• Initial timeline: 2002-2005
  – Specific milestones along the way
ESMF Target Modeling & Assimilation Codes

- Goddard Earth Modeling System (GEMS)
  - NASA/Goddard

- Forecast Modeling System (FMS)
  - Geophysical Fluid Dynamic Laboratory (GFDL)

- MITgcm
  - MIT

- Community Climate System Model (CCSM)
  - National Center for Atmospheric Research (NCAR)

- Global Data Assimilation System
  - National Centers for Environmental Prediction (NCEP)

- Physical-Space Statistical Analysis System (PSAS)
  - NASA/Goddard Data Assimilation Office
The Staff

(This is what was requested. I’m not sure what was actually funded.)

• 3 separate proposals

• 18 Principal and co-investigators

• 19 software engineers

• 1 postdoc

• 1 grad student
Considerations Guiding the Architectural Design (1)

- Provide an infrastructure suitable for use in a broad spectrum of (~15) applications in climate and weather modeling
- Must support reuse of existing high-level components
  - With a reasonable amount of adaptation
- Must provide common infrastructure capable of replacing that used in existing codes
- Selective adoption
  - Users must be able to decide which parts and how much of ESMF to use
- “Natural” interface (for the domain)
Considerations Guiding the Architectural Design (2)

- Extensive grid support
  - Rectangular, reduced, unstructured, nested, adaptive, icosohedral

- Extensive I/O support
  - netCDF, binary, GRIB, BUFR, EOS HDF
  - Parallel I/O

- Support for applications in F77, F90, C, C++

- Performance portability (16-500 CPUs)
Considerations Guiding the Architectural Design (3)

• Software engineering at Capability Maturity Model (CMM) Level II
  – Repeatable -- basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

• Short timeline
  – 3 yrs to provide significant functionality
  – Intermediate deliverables
  – Can’t afford to experiment with immature technologies
The Resulting Design

ESMF Superstructure
AppDriver
Component Classes: GridComp, CplComp, State

User Code

ESMF Infrastructure
Data Classes: Bundle, Field, Grid, Array
Utility Classes: Clock, Log, Prof, DELayout, Machine
ESMF Superstructure

- **AppDriver** is ESMF “main” program
  - Standard implementation provided
  - Users can write their own

- **State** encapsulates simulation data (ESMF data classes)

- **Gridded components** (GridComp) import a state and export a state
  - Associated with a specific grid
  - Annotated as a specific type of model (i.e. atmosphere, ocean, etc.; “other” is an option)

- **Coupler components** (CplComp) transforms an import state to an export state (i.e. regridding)
  - Associated with one or multiple pairs of GridComps

- Components may be assembled hierarchically
ESMF Execution Model

• An ESMF application runs in a set of parallel **persistent execution threads** (PETs)

• Each component is associated with a set of PETs
  – Can be on a subset or across all PETs
  – Couplers should be allocated on the union of the PETs of the gridded components they couple
  – Components can be invoked on a subset of their PETs

• All interprocessor communications take place **within components**

• Object creation is **collective** across all PETs on which a component exists
ESMF State

• Data and metadata to be transferred between ESMF components

• Can contain: bundles, fields, arrays, states
  – Add, get, and set operations

• States and all data objects they can contain have names associated with them (strings)

• Can add objects or placeholders (name only) to state

• Can set/query (by name) that a particular object is needed by a component

• Can set/get attributes for states
  – Logical, integer, real, string
ESMF Component Model

- All user components must provide methods with initialize, run, and finalize functionality.

- Each component must provide a
  \[\text{ESMF}\_\{\text{Grid}|\text{Cpl}\}\_\text{CompSetServices}()\] method
  - Called by framework to allow component to specify methods that provide initialize/run/finalize
  - Within \[\text{ESMF}\_\{\text{Grid}|\text{Cpl}\}\_\text{CompSetServices}()\], use
    \[\text{ESMF}\_\{\text{Grid}|\text{Cpl}\}\_\text{CompSetEntryPoint}()\] to specify callbacks.

- Component interface also includes methods for...
  - create, destroy, get, print, set, validate, wait, getInternalState, setInternalState.

- Components can be assembled hierarchically.

- Recommended practice is to have a single top-level gridded component to orchestrate the rest of the user components.
ESMF Infrastructure (1)

Data Classes

- Array (data + ghost/halo region + attributes)
- Grid (computational domain and distribution)
- Field (array + grid)
- Bundle (collection of fields)
- I/O capability (currently binary and netCDF only)
- Ghost/halo region operations
- Regrid operations (many adapted from SCRIP package)
- Redistribution operations
- Collective operations (like MPI collectives)
ESMF Infrastructure (2)

- **Decomposition Elements (DELayout)**
  - Fundamental unit of data distribution
  - Mapped to PETs elsewhere
  - Current limitations: rectangular layouts only, no weights, no load balancing functionality

- **Virtual Machine (VM)**
  - Really, an abstraction of interprocessor communications, not a virtual machine in the Java sense
  - Generic representation of hardware and software resources
  - Basic unit is PET (equivalent to MPI process)
  - Provides MPI-like communication operations
ESMF Infrastructure (3)

Time Manager Utilities

• Calendar class
  – Different simulations use different calendars (i.e. Gregorian, 360-day, etc.)
  – Provides queries about calendars (i.e. number days per month)

• Time class
  – Represents a time instant in a particular calendar

• TimeInterval class

• Clock class
  – Functions for model time advancement

• Alarm
  – Provides time-based event triggers (i.e. the day the leaves start falling from the trees)
ESMF Infrastructure (4)

- **Configuration class** (*Config*)
  - Read and parse plain text configuration files

- **Logging and Error handling class** (*Log*)
  - Each run has one (default) or more logs
  - Most methods take a Log as an optional argument
  - Can have all PETs log to a single file, or to individual files
  - Messages can be tagged with severity info (info/warning/error)
  - Log entries are stamped with time, PET number, severity
  - Can automatically halt on warning or error
Lessons Learned

• Component model too limited
  – Need more than Initialize/run/finalize
  – Extended model to allow multiple “run” methods

• Implementing your own programming model is challenging
  – The higher the level of abstraction you provide, the harder it is for users to get the best performance

• Data model
  – Too complex
  – Too limited

Further Information
http://www.esmf.ucar.edu
Case Studies:
Component-Based Applications

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
Outline

• Towards “Community” Codes in Quantum Chemistry
• Experimenting with New Parallel Algorithms in Quantum Chemistry
• A Toolkit for Combustion Modeling
• Community Interfaces and Software Reuse (TSTT Mesh Interface)
• Integrated Multi-Physics Fusion Simulations
How I Know About these Projects

• All of these efforts involve the use of Common Component Architecture (CCA) tools, technology, and ideas

• I have been the CCA project’s liaison to our user community for the past five years, and am close to the work described

• I am a participant in the fusion project (SWIM)
CCA-Chemistry Project Participants

Pacific Northwest National Laboratory
Theresa L. Windus
Yuri Alexeev
Manojkumar Krishnan
Jarek Nieplocha
Carl Fahlstrom
Elizabeth Jurrus

Sandia National Laboratory
Curtis L. Janssen
Joseph P. Kenney

Argonne National Laboratory
Steve Benson
Lois Curfman McInnes
Jason Sarich
Towards Community Codes in Quantum Chemistry

• Historically, quantum chemistry has had a number (6-12) of large packages (100k-1M+ lines) with significant functional overlap
  – But not identical capabilities

• These codes do not interoperate

• Developers must choose a package to work with, and are then “stuck” with their choice
  – Changing packages typically requires substantial code changes

• Developers and users suffer because no single package has exactly the set of features and capabilities they need
What is a Community Code?

• A set of code for solving problems of interest to a scientific community which is open to contributions from researchers throughout the community
  – Bigger than just one or a few researchers

• A community code may or may not dominate the software in the domain
  – Example: US climate simulation is dominated by the Community Climate system Model (CCSM)
  – Example: NWChem is sometimes described as a community code, even though there are other major packages

• May be multiple implementations of various functional components, or community may agree on a single implementation
  – If multiple implementations, community needs to agree on interfaces for each component
Why Promote Community Codes?

• Software is too hard to develop
  – Especially high quality, high performance

• Allow researchers to focus on the areas of interest to them and leverage the expertise of others in other areas

• Share and broaden functionality

• Makes the community more productive overall
  – In software and in science
Challenges in Building Community Codes

- **Standardization is hard work**
  - Pay-off comes later

- **Buy-in**
  - (Ideally) need contributors covering all functional aspects of the domain that you need in your code

- **Requires a degree of collaboration and cooperation**
  - Natural tendencies vary by domain

- **Competitiveness and egos**
  - Some tend to perceive even those with different focus areas as competitors rather than collaborators

- **Software architecture**
  - Until recently, it has been hard to define an architecture that was simultaneously open to contributions and manageable to implement and use. Component ideas have changed that
Architectural Approaches to Community Codes

• Adopt a component-based approach
  – Need not be based around specific tools, like CCA
  – Example: Community Climate System Model (CCSM)

• Focus on defining the functional breakdown and the interfaces between function units

• Think about the right levels of abstraction for each functional component
  – Want to be able to utilize existing capability from existing codes wherever possible

• Hierarchical or layered approaches are usually helpful
Towards Community Codes in Quantum Chemistry (Redux)

- Goal: demonstrate feasibility of cross-package interoperability and extensibility using CCA tools

- Example 1: Molecular geometry optimization (high-level interfaces)
  - Also demonstrates reuse of state-of-the-art optimization library developed outside of chemistry community

- Example 2: Integral evaluation (low-level interfaces)
What is Molecular Geometry Optimization?

- Calculate energy of a molecule at a given structure (geometry), and gradient with respect to displacement of atoms
- Use energy and gradient in standard optimization algorithm to find lowest energy structure
- Very high-level capability in a package like NWChem
Example 1: Molecular Geometry Optimization Application

Compute the molecular geometry with minimum energy, i.e. solve $\min f(u)$, where $f: \mathbb{R}^n \to \mathbb{R}$.

**Builder**
Construct application using framework builder services

**Optimization**
$u_{i+1} = u_i + \alpha s \ldots$

**Coordinate Model**
(perform transformations)

**Model Factory**
(instantiate model)

**Linear Algebra**
PETSc
GA

**GUI**

**User Input**

- $f(u)$ energy
- $u$ Cartesian coordinates
- $u$ internal coordinates
- $g$ gradient (in Cartesians)
- $g$ gradient (in internals)
- $H$ Hessian (in Cartesians)
- $H$ Hessian (in internals)
- $s$ update (in internals)

**Relatively high-level interfaces**

$\mathbf{\text{Example 1: Molecular Geometry Optimization Application}}$

**Compute the molecular geometry with minimum energy, i.e. solve $\min f(u)$, where $f: \mathbb{R}^n \to \mathbb{R}$.

$u_{i+1} = u_i + \alpha s \ldots$
CCA quantum chemistry application using components based on:
- MPQC, NWChem (chemistry – energy evaluation)
- GA, PETSc (parallel data management and linear algebra)
- TAO (numerical optimization)
CCA-based integration of state of the art optimization algorithms from the TAO toolkit provides up to 40% improvement in the time required to optimize molecular structures with MPQC and NWChem.¹

What are Molecular Integrals?

• Matrix elements of physical operators in the basis of Gaussian functions used to expand the electronic wavefunction

• Fundamental quantities used by all quantum methods

• Packages can differ significantly in capability to compute specialized types of integrals

• Low-level capability in a package like NWChem
Molecular Integrals: Deeper Levels of Interoperability among Chemistry Packages

- As a first step toward low-level integration of chemistry models, capabilities to provide and utilize molecular integral evaluator components are being added to MPQC and NWChem (also working with GAMESS developers).
- Specialized integral capabilities are provided to CCA chemistry models:
  - Relativistic integrals in NWChem
  - Non-standard 2-electron integrals for linear-R12 theories in MPQC

\[
\int \phi_p(1) \nabla^2_{\frac{1}{2}} \phi_s(1) dr_1 \\
\int \phi_p(1) \phi_q(2) f(r_{12}) \phi_s(1) \phi_t(2) dr_1 dr_2 \\
\int \frac{Z}{r_1} \phi_s(1) dr_1
\]
Next Steps…

- We have demonstrated basic effectiveness of CCA in defining common interfaces and facilitating interoperability between quantum chemistry packages
- Key players in the community are now convinced to pursue component-based (and interface-based) architecture for next-generation community parallel quantum chemistry package
- Serious design and implementation await serious funding
Experimenting with New Parallel Algorithms in Quantum Chemistry

- Sometimes it is necessary to compute the second derivative (Hessian) of the energy of a molecule
- Two approaches
  - Analytical differentiation (major implementation effort, not available for many methods)
  - Numerical differentiation of energy calculations (simple implementation, works for all methods)
- Energy evaluations needed for numerical Hessian are independent, and known in advance
  - Could all be run in parallel
  - NWChem architecture doesn’t make this easy
What’s Wrong with NWChem’s Architecture?

• NWChem is designed for scalable parallelism of each scientific task
• An energy evaluation is a scientific task
• NWChem was designed to carry out each energy evaluation as a parallel computation
• It was not designed to many such parallel computations simultaneously (multi-level parallelism)
Using Component Architectures to Fix the Original Architecture’s Limitations

• By design, components provide an extra level of “insulation” from other components

• Allows multiple instances of identical code to run without interfering with each other

• Encapsulate each energy evaluation as a (parallel) component

• Driver component orchestrates many energy evaluation components in parallel
Multilevel Parallelism - MCMD

NWChem Scalability of Numerical Hessian Computation on 11.8 TFlop/s HP Cluster at PNNL

- Application efficiency improved 10x times on 256 CPUs

Lessons Learned

- Architectural designs can be limiting
  - The use cases you didn’t think of are often the problem
  - Corollary: If you think a use case is too far fetched, think again!
  - Hopefully you won’t hit the limitations for a long time (NWChem is > 13 years old)

- New technologies or architectural revisions may be able to overcome limitations
  - Architectural revisions tend to be pervasive and expensive

- Component technology provides great flexibility, even to “legacy” code
A Toolkit for Combustion Modeling

- Computational Facility for Reacting Flow Science (CFRFS)
  - Funded via SciDAC initiative (PI: H. Najm)
- Focus: A toolkit to perform simulations of lab-sized unsteady flames
  - Solve the Navier-Stokes with detailed chemistry
  - Various mechanisms up to ~50 species, 300 reactions
- Consequently:
  - Disparity of length scales:
    - use structured adaptively refined meshes
  - Disparity of time scales (transport versus chemistry):
    - use an operator-split construction and solve chemistry implicitly
    - adaptive chemistry: use computational singular perturbation to find and follow low dimensional chemical manifolds
- Contributions to research and codebase:
Why Use CCA in the CFRFS Toolkit?

- Separate clearly the physics models, numerical algorithms, and the “CS” parts of the toolkit
  - Strictly functional!

- Realize the separation in software

- Tame *software* complexity

- Separate contributions by transient contributors
  - Form the bulk of the developers

- Create “chunks” of well-defined functionality that can be developed by experts (usually numerical analysts and combustion researchers)
Design Principles of the Toolkit - 1

• **Principal Aim: Reduce software complexity**
  - We can deal with the rest

• **Functional decomposition into components**
  - “Data Object” and Mesh components
  - (Large) set of numerical algorithmic components (integrators, linear/nonlinear solvers, etc.)
  - (Large) set of physical models components (gas-phase combustion chemistry, thermodynamics, fluid dynamic quantities, e.g. viscous stress tensor)
  - Handful of adaptors
Design Principles of the Toolkit - 2

• Decomposition reflected in the port design and implementation
  - Most re-implemented port is the one that exchanges a rectangular sub-domain’s data for processing by components

• Sparse connectivity between components
  - i.e., components communicate with a few others
  - Large apps (component assemblies) are composed by assembling smaller, largely independent sub-assemblies
    • Sub-assemblies usually deal with a certain physics
  - Intuitive way to assemble a multiphysics code
The Code

Separate component subsystems for transport (dark blue) and for reaction (orange) in a reaction-diffusion code. They two are coupled at a relatively high level.
Has the Toolkit Approach Helped Tame Software Complexity?

Questions to consider:

• How has the code evolved?
  – How often have new ports been added?
  – How many rewrites have been done?

• How large are the components?

• How many ports do they have?
  – How large are the ports?

• How many ports exist?
  – i.e., Is the design general enough to support many implementations?

• What is the connectivity of components in application codes?
CFRFS Toolkit Status

• Started in 2001
• 61 components today, all peers, independent, mixed and matched for combustion and shock hydrodynamics
• 7 external libraries
• Contributors: 9 in all, including 3 summer students
• Only 2 of the 9 contributors are at SNL today

A Fitzhugh-Nagumo equation being solved on a block-structured adaptively refined mesh. The top image illustrates Runge phenomena at coarse-fine interfaces (dashed ovals) when using high-order schemes (6th order interpolations with 4th order discretizations). Filtering them with an 8th order filter removes them (bottom).
Scalability: Capability Growth without Rewrites

- Port designs typically occur in spurts followed by long component development times.
- Ports may have multiple implementations; hence the number of ports is typically less than the number of components.
- As the toolkit has matured, the number of ports is seen to be asymptoting to a slow growth rate.
Taming Complexity: Lines of Code

- Most components are < 1000 lines, i.e., they are easily maintainable.
- Components based on GrACE (M. Parashar, Rutgers) and Chombo (P. Colella, LBNL) are the largest in size: parallel mesh libraries with load-balancers.
Taming Complexity: Code Size

- Most components are < 250 kB
- The larger the binary, the more complexity is being hidden in underlying (externally contributed) libraries
Taming Complexity: Interface Size

- A CCA port is a unit of task exchange and generally also a unit of thought
- In CFRFS code, this is typically in the range of 5-10 functions
- Exception: SAMR mesh data port
Taming Complexity: Implementations

- CFRFS ports may have just one or many implementations, as needed, but ...
- Most ports have 1 or 2 implementations
- High-utility ports exist, e.g., for exchanging a patch’s worth of data
Taming Complexity: Callers

• Most CFRFS ports are used by only a few clients, but …

• Key ports are used by many components
Scientific Productivity

- **Conventional Measures**
  - 4 journal papers in CFD/Numerics
  - 4 software-oriented journal papers, 1 book chapter
  - ~11 conference papers, including best paper award
  - Over 60 presentations
  - 1 MS and 1 PhD theses
  - 6 test applications
  - See papers at: http://cfrfs.ca.sandia.gov

- **Unconventional Measures**
  - Did the toolkit spawn new research in app-focused CS (e.g., performance evaluation/enhancement/modeling?)
  - Can the design accommodate software which were themselves designed to be *frameworks* and *not components*?
Lessons Learned

• Toolkit approach is a very effective paradigm for some classes of problems/applications/domains
  – Helps manage software complexity

• Toolkit architecture is very good for allowing expert developers to make a broad range of scientific capability available to less sophisticated users
  – Facilitates scientific productivity

• Component environment is a good match for toolkit building
Community Interfaces and Software Reuse
(TSTT Mesh Interfaces)

Terascale Simulation Tools and Technologies (TSTT)

• http://www.tstt-scidac.org

• Large-scale DOE-funded project focusing on interoperable technology for meshing, discretization, and adaptivity

• Participants: Argonne, Brookhaven, Lawrence Livermore, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Rensselaer Polytechnic Institute and SUNY Stony Brook
Proliferations of interfaces – the N² problem

Current Situation

- Public interfaces for numerical libraries are unique
- Many-to-Many couplings require Many² interfaces
  - Often a heroic effort to understand the inner workings of both codes
  - Not a scalable solution

Diagram:
- Dist. Array
- Overture
- PAOMD
- SUMAA3d
- ISIS++
- PETSc
- Trilinos
Common Interface Specification

Reduces the *Many-to-Many* problem to a *Many-to-One* problem

– Allows interchangeability and experimentation

– Challenges
  • Interface agreement
  • Functionality limitations
  • Maintaining performance
Common Interfaces and Code Reuse

• Typically, libraries offering equivalent functionality have different interfaces

• Users get “locked in” to a particular library because it becomes too hard to change their codes (correctly) to a new one
  – Hard to take advantage of new features of another

• Standardizing interfaces to the functionality reduces lock-in
  – Libraries become easier to reuse across more applications
  – Users have more choices, more functionality, better software
Examples of Successful Common Interface Efforts

- MPI
  - Replaced vendor-specific parallel communication libraries

- BLAS
  - Replaced user-written numerical routines

- LAPACK
  - Replaced LINPACK and EISPACK, which in turn replaced many user-written and other routines

- VSIPL
  - Industry standard for signal and image processing (especially defense-related)
Challenges of Common Interfaces

• They require **work**
  – Typically months to years of regular (not continuous) activity
  – Pay-off only happens later

• They require **buy-in**
  – The impact of a standard is closely related to the number of users
  – Some industries/scientific communities are more agreeable to standardization at different times

• They require **consensus and compromise**
  – If everyone insists that they “win” you won’t get very far
  – If you make decisions that exclude important use cases or users, they will not accept it

• They benefit from **working examples**
  – Standardization informed by (prototype) implementations is most effective
TSTT Philosophy

• Create a **small set of interfaces** that existing packages can support
  – AOMD, CUBIT, Overture, GrACE, …
  – Enable both interchangeability and interoperability

• **Balance performance and flexibility**

• Work with a large tool provider and application community to **ensure applicability**
  – Tool providers: TSTT and CCA SciDAC centers
  – Application community: SciDAC and other DOE applications
TSTT Interface Structure

- TSTTB (Base)
  - Common definitions, utilities, exceptions

- TSTTG (Geometry)
  - Access geometrical information as single entities or arrays

- TSTTM (Mesh)
  - Access information about mesh as single entities or arrays

- TSTTF (Field)
  - Access data on meshes

- TSTTR (Relations)
  - Establish relationships among entities
Lessons Learned

• Even agreeing on the terminology to be used can be a major issue

• Designing a common interface for libraries which don’t exactly overlap in approach and functionality can be challenging
  – Minimalist approach may not be sufficiently useful to the community
  – Requiring adding to library to support common interface may discourage participation

• Interface design needs to support reasonable performance from all implementations
  – Implementations may be tuned for different use cases
  – Interface layer itself may have some performance overhead
  – In TSTT, provide both single entity iterators and block (array) iterators, independent of library’s underlying data structures
Integrated Multi-Physics Fusion Simulation

(The big picture)

- ITER experimental fusion reactor to be built in Caderache, France by an international consortium ($10B)
- Experiment time on ITER will be precious, and must be planned well in advance
- Need a strong “whole device” simulation capability to model, analyze, and eventually adapt (in near real time) experiments
- The fusion simulation community is far this capability. Problems of physics, mathematics, and computer science to be overcome
The Current State of the Art in Plasma Physics

• Many of the individual physical phenomena occurring during a fusion experiment can be simulated with reasonable fidelity

• Little has been done so far with the interactions of multiple physical phenomena (coupled simulation)
  – Coupling tends to reveal new aspects of the physics, which had not previously been evident
  – Coupling (especially across disparate time- and length-scales) offers mathematical challenges to solve the coupled equations
  – How to write and manage couple codes effectively is a computer science challenge
The Landscape of Plasma Physics Software

Nonlinear Gyrokinetic

- GYRO
- GS2
- GRFFIN
- GTC
- SUMMIT
- CQL3D

Vacuum & Conductors

- VACUUM
- VALEN
- EFIT
- TEQ
- VMEC
- TSSEQ

Free Boundary Equilibrium

- low-n
- high-n
- ideal
- non-ideal

Linear Stability

- PEST-I,II
- BALLOON
- NOVA
- CAMINO
- DCON
- GATO
- intermediate-n
- ELITE

MHD- + particles

- low-n
- high-n
- linear
- high-n gyrokinetic

FP-Code

- CQ3D

2D transport

- TSC
- TRANSP
- WHIST
- ONETWO
- CORSICA
- BALDUR

Inverse Equilibrium

- 3D Nonlinear MHD

- static
- time-dependent

- PIES
- TOQ
- ESC
- VMEC2D
- POLAR2D

Plasma Edge

- 2D plasma
- neutrals

- B2
- DEGAS
- UEDGE
- EIRENE
- BOUT

Antenna

- RANT3D

RF Heating & CD

- AORSA
- TORCH
- ORION
- LSC
- TORAY
- METS
- CURRAY

denotes parallel MPI code

Courtesy of Steve Jardin, PPPL
The SWIM Project

Center for Simulation of RF Waves Interactions with Magnetohydrodynamics

- http://www.cswim.org
- Design and develop an “Integrated Plasma Simulator” coupling
  - Radio frequency energy, used to heat and control the plasma
  - Magnetohydrodynamics (MHD) – the dynamics of electrically conducting fluids (such as a plasma)
  - Transport of plasma particles
- One of several prototype projects for integrated fusion simulation in the US
- Participants
  - Columbia University, CompX, General Atomics, Indiana University, Lawrence Berkeley National Lab, Massachusetts Institute of Technology, New York University, Oak Ridge National Lab, Princeton Plasma Physics Lab, Science Applications International Corporation, University of Wisconsin
Two Regimes of Scientific Interest

- **“Fast” MHD**
  - Plasma response to RF heating is much slower than MHD motion
  - Separation of time scales between RF and MHD simplifies integration (physics, math, and CS)

- **“Slow” MHD**
  - RF and MHD effects interact directly on much closer time scales
  - Much more challenging for integration
Framework Design Guidelines (1)

• “Don’t startle the physicists”
  – Most of the physicists involved in this projects are interested in the physics, not in fancy new software development techniques
  – Framework and integration approaches must be incremental and not suddenly require major changes in the way they’re used to developing software

• Must not fork code base for components to create integrated versions
  – Codes must continue to work and evolve for stand alone use as well as integrated use
Framework Design Guidelines (2)

• Must be able to work with multiple implementations of each component
  – Interoperability
  – Beginnings of “community code” initiative

• Must work with existing codes “as is” for as long as possible
  – Explicit recognition that eventually, scientific drivers will require tearing apart and reconstructing component codes (slow MHD case)
Fast MHD Architecture

Common superstructure: Component orchestration and workflow tools, job launch, and job management and monitoring tools.

**Integrated Plasma Simulator**

- **Initialization**
- **Time update**
- **Apply “reduced” instability model**
  - stable
  - unstable
- **Test Linear Stability** (needs same data as xMHD code suite)
- **Evolve Equilibrium Magnetic Configuration** (map flux surfaces, solve Grad-Shafranov, advance vessel currents)
- **Evolve Profiles** (includes sources and distribution function parts)
- **Compute Self-consistent Electric Field** \( E(\rho) \)

**sources**
- RF:
  - Wave solution and deposition given
  - \( f(x,v) + f_{\text{run}} + f_{\text{source}} + p_x(x), J_x(x) \)
  - compute \( \frac{\partial f}{\partial t} = c(f) + Q(f) + S(x,v) + E_i \)

- NBI & \( \alpha \):
  - Injection, ionization, slowing down, birth
- Fueling:
  - Gas Puffing, recycling, pellets

**Shared infrastructure components:** high level numerical libraries, job monitoring and tracking, file I/O and staging, metadata management, collaboration, graphics

**“Extended MHD” Code Suite**

- Initialize 3D calculation to marginally unstable state
- \( t = t + \Delta t \)

**Evolve 3D fields, velocities, and pressures in presence of energetic non-Maxwellian particles.**

**Evolve 5D energetic particles component in prescribed \( E \) and \( B \) fields.**

**2D State Data:**
- Equilibrium, Profiles, distribution function

**3D State Data:**
- Fields, velocities, particles

---

ORNL: Oak Ridge National Laboratory
Slow MHD Architecture

**Common superstructure:** Component orchestration and workflow tools, job launch, and job management and monitoring tools, initialization.

- **“RF Distribution Function” Code Module**
  - 3D RF:
    - Wave solution and deposition
  - Quasilinear Diffusion Tensor:
    - 5D calculation from RF fields
    - [bounce average to 4D]
  - Distribution Function f:
    - Fokker-Planck and/or Monte Carlo calculations of [4D] 5D distribution function with RF and collisions
  - Closure Moments
    - Velocity moments of f
    - (in terms of XMHD unknowns)

- **“Extended MHD” Code Module**
  - Evolve 3D fields, pressures, and velocities
  - Closure Moments
    - assemble analytic, numerical, J, H, Q
  - Closure Moments
    - analytic forms of f

**3D RF Fields;**

**[4D]5D distribution function**

**3D State Data:**

**Fields, velocities, particles**

**Shared infrastructure components:** high level numerical libraries, job monitoring and tracking, file I/O and staging, metadata management, collaboration, graphics
CS Approach in SWIM Project (1)

• Focus on **interfaces** first
  – Interchangability of physics components is a key element of the SWIM project plan
  – Also a critical technical need if the plasma physicists hope to come together to work as a community (as sponsor desires)

• Focus on **testing** next
  – Need to be able to insure that modifications to codes do not impact their correctness (standalone or coupled)
  – Need to be able to distinguish physics/math problems with coupling from simple code bugs introduce during coupling

• Focus on **community code** ideas
  – CS ideas that will help plasma physics community work together around code
CS Approach in SWIM Project (2)

- Focus on **performance** where it is needed
  - Fast MHD components don’t exchange much data, and file-based coupling is sufficient. But individual components need performance
  - In slow MHD, coupling will tighter and more performance critical

- **Be conservative** introducing CCA and other CS tools
  - Let the physics take the forefront
  - Don’t distract with “good for you” ideas that don’t advance the physics
Lessons Learned

- Design influenced by sociological issues at least as much as technical ones
- Consideration given to…
  - Developers
  - Software base
  - Users
  - Norms of the scientific domain
  - Sponsor expectations
- Sometimes the most effective architectural solution is not the most sophisticated or the most high-tech!
Generated Applications

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
What are Generated Applications?

• Traditionally, we write the code for the application by hand

• Generated applications (my term) use tools to produce the application-level code from a higher-level description of the problem/application

• Why bother?
  – Developers/user lack HPC software expertise
  – Families of related applications which can be generalized at a higher level
    • High-level domain-specific languages (HLLs or DSLs)
  – Algorithms which require different implementations on different hardware platforms for performance reasons (common in HPC)
    • Parameterized algorithms

• Special challenge: requires working at a higher level of abstraction
Common Approaches to Code Generation

While depicted as separate paths here, HLLs and parameterized algorithms are often used together.

Key:
- Application generation tools shown in green and red
- User code shown in yellow
- Generated code shown with diagonal stripes
Why Use Code Generation?

- **Performance**
  - Automate tuning implementations for diverse hardware platforms

- **Productivity**
  - Provides a higher level of abstraction for the programmer (closer to the way they think about their science)
  - Less *complex* user code required to produce an application
  - *Less* user code required to produce an application

- **Friendlier environment** for domain scientists
  - “Little” languages can be easier to learn and use effectively for domain scientists with little HPC programming experience
Why *Not* Use Code Generation?

- Tool may end up being **too narrow** in scope
  - How much effort is required to develop a tool with the breadth of capabilities your users will want?
  - If the tool doesn’t fit the user’s problem, it is of little use (to them)

- Can be harder to **debug and maintain**
  - May have a “different” application for every platform/problem
  - Often, must debug generated code and then figure out why it was generated wrong
  - You may have to develop additional tools & tests

- Harder to **verify** generated codes
  - How will you satisfy users that the generated code is correct?

- Requires a higher level of **abstraction**
  - May require a deeper understanding of the problem domain

- Often require close **collaboration** of domain scientists and computer scientists
  - May not be feasible, often challenging
Don’t Let the Negatives Scare You – Too Much

• The preceding slide was not meant to scare you away from code generation approaches
• It was meant to set expectations appropriately
• Code generation typically requires a major investment in the beginning
• Can yield major pay-offs later on
• Need to seriously consider the balance before pursuing code generation

• Note: all of the negatives on the preceding slide assume you want a robust, general, high-quality tool. Reduced expectations reduce the concerns
Basic Code Generation Approaches (1)

High-level domain-specific languages (DSLs)

• A (small) programming language
• Expresses problems in the target domain more easily than traditional general-purpose languages (GPLs)
• Usually not as broadly expressive as a general-purpose language
• DSL code gets expanded into a traditional language, compiled and run
• High-level class libraries in OO languages can look a lot like DSLs (especially with overloading)
  – Direct access to a complete language may be good or bad
• High-level virtual machine architectures are close cousins
Basic Code Generation Approaches (2)

Parameterized adaptive algorithms

• General algorithm is defined in a general-purpose language

• Implementation is tuned by substitution of low-level code, and definition of parameters defining high-level algorithm

• Tuning input may come from empirical tests, performance models, or other sources

• Some tuning may be done at compile time, some at run time (choosing alternate implementations based on runtime information)
Case Study:
Automatically Tuned Linear Algebra Software (ATLAS)

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
How I Know About this Project

• ATLAS is familiar to many researchers in the computational science/high-performance computing community as a high-performance implementation of the BLAS library
  – Few know (or need to know) the details of how it was designed or how it works

• http://math-atlas.sourceforge.net
  – Code downloads
  – References

• This case study is drawn primarily from
Introduction to ATLAS

• An implementation of the Basic Linear Algebra Subroutine (BLAS) library

• Motivation: BLAS performance on modern CPUs is highly dependent on making optimal use of the memory hierarchy (caches, registers)
  – Compilers don’t do this well (except Level 1 BLAS)
  – Producing and maintaining hand-tuned implementations for all platforms is extremely labor-intensive.
    • Some hardware vendors do it for their platforms, but not all

• Justification: BLAS is probably the most widely used numerical library in scientific computing, and is performance-critical for many applications
  – Other linear algebra libraries rely on BLAS for performance
General Requirements for Automated Empirical Optimization of Software (AEOS)

- Isolation of performance-critical routines
- Method of adapting software to different environments
  - Parameterized algorithms
  - Parameterized low-level code generation/substitution
- Robust, context-sensitive timers
  - Accurate timings across platforms
  - Must reflect real-world use cases (cold/warm cache, etc.)
- Appropriate search heuristics
  - Must find a reasonable solution
  - Must find a solution quickly (must prune search space quickly)
  - How long will users wait to build the code/library?
Overview of the BLAS

- **Level 1** (vector-vector operations, 15 basic routines)
  - $O(N)$ operations, $O(N)$ memory references
  - Optimizations: FP unit usage, loop unrolling, etc. (0-15%)
  - Current compilers optimize well
  - ATLAS provides hand-tuned implementations

- **Level 2** (matrix-vector operations, 24 basic routines)
  - $O(N^2)$ operations, $O(N^2)$ memory references
  - Optimizations: reuse of vector operand, memory blocking, loop optimizations (10-300%)
  - Some optimizations too complex for current compilers
  - ATLAS uses two optimized low-level kernels to implement

- **Level 3** (matrix-matrix operations, 9 basic routines)
  - $O(N^3)$ operations, $O(N^2)$ memory references
  - Optimizations: memory blocking, loop permutation and unrolling (orders of magnitude speedup)
  - Generally too complex for current compilers
  - ATLAS uses one optimized low-level kernel to implement
Level 3 BLAS in ATLAS

• All other BLAS3 operations can be implemented using an efficient matrix multiply ("GEMM") routine

\[ C \leftarrow \alpha \, \text{op}(A) \, \text{op}(B) + \beta \, C \]

where \( \text{op}(X) = X \) or \( X^T \),

\( C \) is \( M \times N \), \( \text{op}(A) \) is \( M \times K \), \( \text{op}(B) \) is \( K \times N \)

• GEMM for matrices of arbitrary size can be implemented using an efficient L1 cache-contained matrix multiply ("L1 matmul")
Building GEMM from L1 matmul (1)

- Parameterized algorithm (not generated)
- L1 matmul works best if blocks are contiguous in memory, therefore…
- Is GEMM large enough to tolerate copying inputs?
  - Copy into “block-major” order, $O(N^2)$ overhead
  - Eliminates TLB problems, minimizes cache thrashing, maximizes cache line reuse
  - Apply a to small of A or B while copying
  - Apply transpose operators as appropriate while copying
- For small cases, copying may dominate $O(N^3)$ compute cost
  - In this case, use non-optimal L1 matmul with strided access to A, B in place
- Crossover points depend on architecture, shape of operands
  - Parameters determined empirically at install time
  - Parameters used at runtime to select between alternatives
Building GEMM from L1 matmul (2)

• Two different loop orderings considered
  – $M$ outer loop (over rows of $A$)
  – $N$ outer loop (over columns of $B$)
  – $K$ always inner loop (common dimension to $A$ and $B$)
  – If a matrix will fit in L2 cache, use it as outer loop
  – If neither fits in L2 cache, use largest as outer loop
  – Effective size of L2 determined by empirical $\text{CacheEdge}$ parameter

• Result may be written directly into $C$ or into a temporary $\hat{C}$
  – $\hat{C}$ advantages: control over address alignment, data is contiguous
  – $\hat{C}$ disadvantages: additional write
  – Additional write not a problem if L1 matmul is called many times
  – In GEMM-based rank-$K$ update, using $\hat{C}$ can double write cost
  – Uses heuristic based on size of $K$-loop
Building GEMM from L1 matmul (3)

• Blocking for higher-level (above L1) cache utilization
  – Algorithm supports two levels of explicit blocking (i.e. L1+L2 or L1+L3). Not recursive
  – Four cases, depending on sizes of matrices and L1 block size
L1 matmul Design Considerations (1)

• Inputs are expected to be $N_B \times N_B$ (block size)
  – Except for edges

• Fit into cache:
  – All of $A$
  – Two columns of $B$
    • Assumes LRU cache eviction policy. Last column of $B$ is LRU, prevents $A$ being evicted
    • Could have swapped roles of $A$ and $B$
  – One cache line of $C$
  – Requirement to have all of $A$ and $B$, or all of $A$, $B$, $C$ in cache is a common misconception

• Code for L1 matmul should fit in instruction cache
  – Limits unrolling and other code-bloating optimization
L1 matmul Design Considerations (2)

• Floating point instruction ordering
  – If processor lacks fused multiply-add operation, skew loop to separate multiply and add to accommodate FP pipeline duration
  – Some hardware and compilers can perform this optimization, but not all, so do it explicitly

• Reducing loop overhead
  – Unroll $K$ loop (while fitting code in instruction cache)
  – Other loops will change memory access patterns

• Expose instruction-level parallelism (multiple FP units)
  – Unroll $M$ and/or $N$ loops
  – Coupled to register blocking

• Find the right number of outstanding cache misses
  – Too few will not maximize memory bandwidth
  – Too many will block execution until some are satisfied
L1 matmul Search Approach (1)

• Find size of L1 cache
  – Timing of memory access
  – Searches powers of 2 only

• Determine FP unit characteristics
  – Presence of fused multiply-add
  – Length of pipeline
  – Number of FP registers

• These results define a range of possible blockings for the registers and unrollings of M, N
  – $a_r b_r + a_r + b_r \leq N_r$, $a_r = m_u$, $b_r = n_u$
  – Empirical search in (reduced) parameter space for best performance

• Search for best $K$ unrolling
  – In practice either 1 or $K$ are best, so try only those two
L1 matmul Search Approach (2)

• Save all search results in a file
  – Avoids need to repeat if search is interrupted before completion

• Typical installation (i.e. parameter search) requires 1-2 hours per precision (2001 hardware)
ATLAS DGEMM Performance (2001)

[Bar chart showing performance comparison between different architectures and libraries (Vendor BLAS, ATLAS BLAS, F77 BLAS).]
GEMM-Based Level 3 BLAS

- \{TR, SY, HE\}MM
  - Triangular, symmetric, or hermitian matrix multiply
- \{SY, HE\}\{RK, R2K\}
  - Symmetric or hermitian rank-\(k\) or rank-\(2k\) update
- Recursively partition matrices until blocks are smaller than \(N_B \times N_B\) and use L1 matmul
- Trade-off between simplicity of implementation and absolute maximum performance
  - In practice, performance penalty is small
Level 2 BLAS in ATLAS

• Each routine has…
  – One matrix operand
  – One or two operands

• Without any blocking, \(O(N^2)\) memory references would be required on each operand
  – Naïve \(y \leftarrow Ax + y\) requires \(3N^2\) writes + \(N^2\) reads
  – Minimum is \(N^2 + N\) reads + \(N\) writes

• Primary goal is to minimize memory access
Register Blocking in Level 2 BLAS

- Register set is essentially a very fast “Level 0” cache
  - Typically only 8-32 registers

- Register blocking can reduce memory access for only one of the vector operands to $O(N)$

**Inner product formulation**

```plaintext
do I = 1, N
    r = y(I)
    do J = 1, N
        r += A(I,J) * x(J)
    end do
end do
```

Unroll I loop to use $R_y$ registers
Reduce $N^2$ access to $x \rightarrow N^2/R_y$
$O(N^2)+O(N)$ reads, $O(N)$ writes

**Outer product formulation**

```plaintext
do J = 1, N
    r = x(J)
    do I = 1, N
        y(I) += A(I,J) * r
    end do
end do
```

Unroll J loop to use $R_x$ registers
Reduce $y$ access $\rightarrow N^2/R_x$ reads and writes
$O(N^2)+O(N)$ reads, $O(N^2)$ writes
Cache Blocking in Level 2 BLAS

- Insure that $O(N^2)$ accesses to $x$ are satisfied from L1 cache
  - Main memory access only $O(N)$, others satisfied from cache

- Compute partitioning as

$$N_p = \frac{S_1 - R_y}{R_y + 1}$$

  - Where $N_p$ is block size of $x$, $S_1$ is size of L1 cache
ATLAS Level 2 BLAS

- Two compute kernels required
  - L1 matvec
  - L1 update1 (rank-1 update)

- Level 2 BLAS built after Level 3
  - L1 cache size already determined

- Use multiple implementation (not code generation)
  - Try: matvec, transpose matvec, L1 update1 to find best performance for each operation
  - Conjugate matvecs perform same as non-conjugate

- Use best algorithm to search for optimal fraction of L1 cache to use
A Note on Empirical vs Model-Based Optimization

• Empirical optimization runs test codes and searches based on actual performance
  – Easier to create, more time to run
  – Can be fooled if search is not sufficiently careful

• Model-based optimization relies on a mathematical performance model
  – Harder to create, less time to “run”
  – Often simplified/idealized
  – May not fully capture interactions of parameters
  – Results only as good as the performance model
Lessons Learned

• Low-level optimization across diverse hardware platforms is challenging
  – Many variables to consider, with complex interdependencies
  – Excellent reasons for automatic code generation approaches!

• Think about where performance is paramount and where you can sacrifice a little for understandability, portability, maintainability, etc.
  – “The perfect is the enemy of the good”

• Strongly hierarchical architecture of library helps reduce the amount of code at the lowest level
  – Lowest levels are hardest to write and maintain

• Think about size of search space and cost of tests
  – Even highly constrained ATLAS design takes $O(\text{hours})$ to search
Case Study: The Tensor Contraction Engine (TCE)

David E. Bernholdt
Oak Ridge National Laboratory

bernholdtde@ornl.gov
http://www.csm.ornl.gov/~bernhold/
How I Know About This Project

• I have been involved in the project since its inception

• Collaboration of researchers at
  – Louisiana State University (G. Baumgartner, J. Ramanujam)
  – Ohio State University (R. Pitzer, P. Sadayappan)
  – ORNL (D. E. Bernholdt, R. J. Harrison)
  – University of Florida (S. Hirata)
  – University of Waterloo (M. Nooijen)

• http://www.csc.lsu.edu/~gb/TCE/
  – Publications
Background: Quantum Chemistry (Redux)

- A relatively **small number** of very large, long-lived packages
  - Originate in a single research group
  - Collaborators, former students/postdocs at other sites add to developer base

- Packages tend to be large (**100k-1M+ lines**) and provide a wide range of quantum chemistry methods
  - Substantially the same core capabilities implemented in each
  - Cutting edge capabilities vary by research focus of group

- Many packages have been under **continual development** since the 1970s or late 1960s

- Typical architecture is either **monolithic** or a collection of multiple monolithic executables
  - Somewhat like filter-and-pipe architecture
Common Computational Chemistry Methods (Redux)

• Molecular dynamics
  – Ball and spring model for atomic interactions (no quantum mechanics)
  – Parameterized from expt or high-quality theoretical data
  – Cost scales as $N^2$
  – Useful for large molecules (proteins, drug design, etc.)

• Hartree-Fock, Self-Consistent Field, Density Functional Theory
  – Simplest model including quantum mechanics
  – Results are often qualitative, occasionally quantitative
  – Cost scales as $N^4$
  – DFT popular, HF/SCF used less often

• Perturbation theory (MP2, MP4)
  – Increasingly sophisticated quantum mechanical models
  – Results often quantitative
  – Costs scale as $N^5$ to $N^7$

• Coupled cluster theory, configuration interaction
  – Most sophisticated quantum mechanical models
  – Results can be compared to experiment
  – Costs scale as $N^6$ to $N!$
Two Modes of Research involving High-End Methods

1. Researchers use codes written by others to solve chemical problems
   - Computational throughput paces progress
     • Most studies involve many calculations
   - Need to be able to run efficiently on whatever computers they have access to
   - Can push codes and computers to their limits
   - May not have strong computer/software knowledge to deal with problems in performance or scale
Two Modes of Research involving High-End Methods

2. Researchers develop and implement new methods in the many-body family and test on challenging molecules
   - Time to implement new methods paces progress (3-6 months typical). Want to try several variants of each
   - Tendency to produce “toy” implementations, may not be capable of examining enough challenging molecules
   - Level of effort and knowledge required to produce high-performance implementations is very high
   - Performance across diverse hardware platforms is a significant challenge
Why are Many-Body Methods So Challenging?

- Iterative non-linear equations involving large tensors with complex symmetry and sparsity properties
- Primary data are tensors of rank-2, -4, -6, …
- \( N \sim 1000-3000 \) is of interest
  - Not yet possible in practice

<table>
<thead>
<tr>
<th>Method</th>
<th>Terms</th>
<th>Cost</th>
<th>Data</th>
<th>F77 Lines</th>
<th>Lines/Term</th>
<th>First Impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCD</td>
<td>11</td>
<td>( N^6 )</td>
<td>( N^4 )</td>
<td>3,209</td>
<td>292</td>
<td>1978</td>
</tr>
<tr>
<td>CCSD</td>
<td>48</td>
<td>( N^6 )</td>
<td>( N^4 )</td>
<td>13,213</td>
<td>275</td>
<td>1982</td>
</tr>
<tr>
<td>CCSDT</td>
<td>102</td>
<td>( N^8 )</td>
<td>( N^6 )</td>
<td>33,932</td>
<td>333</td>
<td>1988</td>
</tr>
<tr>
<td>CCSDTQ</td>
<td>183</td>
<td>( N^{10} )</td>
<td>( N^8 )</td>
<td>79,901</td>
<td>438</td>
<td>1992</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>up to ( N! )</td>
<td>up to ( N_{\text{elec}}^2 )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
In the coupled cluster method with single and double excitations (CCSD) the "singles" and "doubles" equations are iterated until convergence and that solution is used to evaluate the molecular energy.
Why are Many-Body Methods So Challenging? (2)

• Only a few basic tensors, but they are contracted with each other in many different ways
  – “Simple” index mistakes can be catastrophic, but hard to debug
  – Diversity of tensor access patterns make optimization extremely challenging

• Must use permutational symmetries, sparsity to achieve maximum computational efficiency
  – Details depend on specific method, molecule, computational basis, representation

• Intensive use of CPU and memory and disk resources
  – Most calculations don’t fit in memory
  – Some calculations don’t fit on disk

• Each hardware platform has a unique balance of CPU, memory, interconnect, I/O
Is There Any Good News?

• Tensor contraction equations are quite simple
  – The *implementation* in a program is hard

• All tensor contractions follow the *same basic structure*
  – Difference is in details of input & output shapes, which indices are contracted, etc.

• A *broad spectrum of many-body methods* are of interest to chemists
  – More opportunity for “reuse” of ideas and code
  – *Amortize cost* of developing a special tool

• Field is *relatively mature*
  – Much of the design space has been *explored manually*
  – Hand-coded *implementations exist* for many methods
  – Experience with manual optimization
Potential Solution: A Domain-Specific Language and Compiler

• Design a high-level language to allow chemists to provide tensor equations in a form very similar to what they would publish in a paper
  – Highly productive for application developers
  – “Minimal” description of what the resulting application must do, so minimal constraints on how “implementer” makes it work

• Use an “optimizing compiler” to translate from DSL into code in a general-purpose language
  – Productive for application developers
  – High performance for end-users

• Link with existing chemistry packages (i.e. NWChem) to provide chemistry-related infrastructure
  – Complete language not required
Design Requirements: Language

• Language must be **natural** (to chemists), **high-level, compact, and unambiguous**
  – Chemists have a tendency to use slightly different notational conventions in different contexts
  – Interpretation of language should *not* rely on chemistry-specific ideas

• Language must be able to express tensor contraction expressions, as used in many-body methods

• Being able to express a **complete** (iterative) calculation in the language is desirable, but *not required*
  – Produce subroutines for tensor contractions which can be called by a hand-coded driver
  – Language must accommodate calling out to hand-coded routines
Design Requirements: Compiler

- Compiler must generate code in a general-purpose language
  - Take advantage of GP language compiler optimizations
  - Simplify integration with NWChem infrastructure
- Compiler must perform the same kinds of optimizations a chemist would do by hand
  - But do them more generally, more rigorously
  - Otherwise, chemists wouldn’t take it seriously (wouldn’t perform well enough)
- Compiler must also accommodate additional “novel” optimizations
- Compiler must take into account system (hardware) characteristics when generating/optimizing code
- Assume new code will be generated for every calculation
  - Generated code tailored to specific hardware and molecule
  - Don’t worry about range of applicability for a given generated result
  - Not ideal for chemists – concession to computer science challenges!
TCE Language (1)

\[ S_{abij} = \sum_{cefk} A_{acik} B_{befl} C_{dfjk} D_{cdeg} \]

```plaintext
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k,l : O;

mlimit = 100GB;

procedure P(in A[V,V,O,O], in B[V,V,V,O],
in C[V,V,O,O], in D[V,V,V,O],
out S[V,V,O,O])=
begin
S[a,b,i,j] == sum[ A[a,c,i,k] * B[b,e,f,l] * C[d,f,j,k] * D[c,d,e,l],
{} c,e,f,k,l ];
end
```

Compiler uses index range values to determine operation and storage costs

Memory limit is a hardware parameter that will be moved to a separate “system profile” input file

Define a procedure to generate

Procedure body can contain many tensor contraction statements
TCE Language (2)

- Not shown in example:
  - Call-outs to hand-coded routines
  - Specification of tensor permutational symmetries

- Current language is based on tensors and their mathematical properties
  - No chemistry/physics specific constructs
  - Maps obviously and directly into GP language code (for naïve implementation)

- Chemists typically derive TCE-level equations from even higher-level operator equations
  - Largely index-free representation
  - More closely related to the physics (strictly, still mathematical, but doesn’t map directly into GP language)
  - Separate “Operator Contraction Engine” (OCE) tool to expand operator equations into TCE equations
TCE Compiler Design: Structure

Structurally, looks mostly like a compiler for a GP language, or any other language.

Currently where most of the optimizations based on domain knowledge take place.

These components would not be part of a traditional compiler.

Traditional compiler optimizations are mostly based on the AST. But the TCE AST is not the same as a GP language AST, so optimizations are different.
TCE Compiler Design: Function

- **Algebraic Transformations**
  - Minimize operation count (ICCS’05, ICCS’06)

- **Memory Minimization**
  - Reduce intermediate storage via loop fusion (LCPC’03)

- **Space-Time Transformation**
  - Trade-offs between storage and recomputation (PLDI’02)

- **Data Locality Optimization**
  - Optimize use of storage hierarchy via tiling (ICS’01, HiPC’03, IPDPS’04)

- **Data Dist./Comm. Optimization**
  - Optimize parallel data layout (IPDPS’03)

- **Integrated System**
  - (SC’02, Proc. IEEE 05)
TCE Structure vs. Function

**Tensor Expressions**
- TCE Language Parser
- Simple Expression Tree Optimizations
- Loop Fuser
- Abstract Syntax Tree Generator
  - Abstract Syntax Tree Optimizations
  - Code Generator
    - **Generated Code**

**Tensor Expressions**
- Algebraic Transformations
  - Memory Minimization
    - No sol'n fits disk
    - Sol'n fits disk, not mem.
    - Sol'n fits mem.
      - Space-Time Trade-Offs
      - Storage and Data Locality Management
        - Data Distribution and Partitioning
          - Parallel Code Fortran/C/…
          - OpenMP/MPI/Global Arrays
          - No sol'n fits disk, sol'n fits mem.
TCE Implementation

- **Prototype TCE** written in Python
  - Only does (partial) expression tree optimization
  - No AST, nor optimization on AST
  - Generated ~2.5 M lines of code now shipped with NWChem

- **“Optimizing” TCE** written in Java
  - JavaCC parser generator
  - Prototypes for some optimizations/components were written in ML, Haskell, and other languages

- **Generated code:** Fortran + Global Arrays + NWChem
  - Arbitrary choice. I thought chemists would take Fortran code more seriously
  - Size of TCE-specific “runtime” library still under discussion
  - “Work pool” programming model under development
A Brief Sketch of the TCE in Action

- Put user input into operation-minimal form through algebraic transformations on the expression tree (math)

- “Factorize” expression tree to identify common sub-expressions that can be evaluated once and reused (domain-specific)

- Reduce memory usage due to temporaries introduced above via loop fusion (general)

- If necessary, trade memory usage for extra operations by selectively recomputing quantities (general)

- Tile (block) calculation to make efficient use of disk and memory bandwidth (general)
Operation-Minimal and Memory-Minimal Forms

for a, e, c, f

for i, j

\[ X_{aef} += T_{iaef} T_{ijcf} \]

for c, e, b, k

\[ T_{1cebk} = f_1(c, e, b, k) \]

for a, f, b, k

\[ T_{2afbk} = f_2(a, f, b, k) \]

for c, e, a, f

for b, k

\[ Y_{ceaf} += T_{1cebk} T_{2afbk} \]

for c, e, a, f

\[ E += X_{aef} Y_{ceaf} \]

\[ A3A = \frac{1}{2} X_{ce,af} Y_{ae,cf} + \frac{1}{2} \sum_{i} \frac{1}{2} Y_{ae,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} \]

\[ \frac{1}{2} Y_{ae,cf} = \langle ab | ek \rangle \langle cb | fk \rangle \]

for a, f, b, k

\[ T_{2afbk} = f_2(a, f, b, k) \]

for c, e

\[ T_{1bk} = f_1(c, e, b, k) \]

for a, f

\[ X += T_{iaej} T_{ijcf} \]

for b, k

\[ Y += T_{1bk} T_{2afbk} \]

\[ E += X Y \]

\[ a .. f: \text{ range } V = 1000 .. 3000 \]
\[ i .. k: \text{ range } O = 30 .. 100 \]
Tiling to Reduce Recomputation

for \( a^t, e^t, c^t, f^t \)

for \( a, e, c, f \)

for \( i, j \)

\[
X_{aeef} += T_{iafe} T_{ijcf}
\]

for \( b, k \)

for \( c, e \)

\[
T_{1ce} = f_1(c, e, b, k)
\]

for \( a, f \)

\[
T_{2af} = f_2(a, f, b, k)
\]

for \( c, e, a, f \)

\[
Y_{ceaf} += T_{1ce} T_{2af}
\]

for \( c, e, a, f \)

\[
E += X_{aeef} Y_{ceaf}
\]

<table>
<thead>
<tr>
<th>array</th>
<th>space</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X )</td>
<td>( B^4 )</td>
<td>( V^4O^2 )</td>
</tr>
<tr>
<td>( T1 )</td>
<td>( B^2 )</td>
<td>( C_{f1}(V/B)^2V^3O )</td>
</tr>
<tr>
<td>( T2 )</td>
<td>( B^2 )</td>
<td>( C_{f2}(V/B)^2V^3O )</td>
</tr>
<tr>
<td>( Y )</td>
<td>( B^4 )</td>
<td>( V^5O )</td>
</tr>
<tr>
<td>( E )</td>
<td>1</td>
<td>( V^4 )</td>
</tr>
</tbody>
</table>

Tiling provides a controlled compromise between minimal operations and minimal memory (full fusion).
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k : O;

mlimit = 1000000000000;

function F1(V,V,V,O);
function F2(V,V,V,O);

procedure P(in T1[O,O,V,V], in T2[O,O,V,V], out X)=

begin
  X == sum[ sum[F1(a,b,f,k) * F2(c,e,b,k), {b,k}]
    * sum[T1[i,j,a,e] * T2[i,j,c,f], {i,j}],
    {a,e,c,f}];
end

The Value of Optimization (Space-Time Trade-Offs)

TCE explores many algorithms, selects best for each memory size

Hand-coded solution (single algorithm)
Effect of Multidimensional Optimization: AO-to-MO Transformation

\[ B(a,b,c,d) = \sum_{p,q,r,s} C1(s,d) \times C2(r,c) \times C3(q,b) \times C4(p,a) \times A(p,q,r,s) \]

Range \( p,q,r,s = 150 \), Range \( a,b,c,d = 140 \), Memory = 2GB

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Total Disk I/O Time (s)</th>
<th>Total Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Fusion, Fixed Tiling ((^{\sqrt{M/3}}))</td>
<td>1240.85</td>
<td>1957.18</td>
</tr>
<tr>
<td>No Fusion, Optimized Tiling</td>
<td>747.83</td>
<td>1261.95</td>
</tr>
<tr>
<td>Fusion + Optimized Tiling</td>
<td>248.43</td>
<td>954.87</td>
</tr>
</tbody>
</table>

Measurements were taken on an Itanium 2 System
TCE Productivity

• Prototype TCE has been used to generate implementations of > 25 different many-body methods

• TCE implementation took a few weeks
  – Conservative estimate to hand code all methods » 5 years!

• First-ever parallel implementation for many of the methods!

• ~2.5 M lines of generated code
  – Distributed as part of NWChem
TCE Language: Future Directions

- **Extend language** to include control structures, more general data types, etc.
  - Want to be able to express a “complete” application in TCE language
  - Would require addition of chemistry-specific primitives

- **Embed TCE syntax in a scripting language**
  - Avoid having to maintain the “uninteresting” parts of the language (from a research standpoint)
  - Could also use e.g. C++ or a functional language

- **Something to think about:** Design of DSL environments must be a *compromise between generality and simplicity*
  - Simple (constrained) language = fewer user errors
  - Simplicity may prevent a user from doing what they really want to do
TCE Compiler: Future Directions

• More and better **optimizations**
  – Still many opportunities for AST optimization
  – Coupled/interacting optimizations

• **Formalize APIs** for generated code, drivers

• Better/more efficient **data structures** for tensors
  – Incorporate permutational symmetry, sparsity

• Better **abstraction** of code generation
  – Make it easier to plug in different back-ends
  – Other chemistry packages would like to be able to integrate TCE-generated code

• More sophisticated **performance models**
  – Currently just operation counts and data volumes
  – Memory access patterns could significantly influence performance (each tensor is accessed in many different ways)
Aside: The TCE as a Computer Science Research Project

- This project differs from most past DSL efforts in chemistry because of the strong involvement of computer scientists
  - DSLs typically used for throwaway prototypes, not production codes
- Computer scientists eventually want to generalize results to other domains
- Requires careful understanding of origin of potential optimizations, algorithms and their generality
  - Specific to the chemistry/physics?
  - General mathematics/computer science?
- Examples:
  - Operation minimization heuristics are strongly guided by experience of chemists with hand implementations
  - Data locality optimization (disk/memory/cache tiling) completely general
Lessons Learned

• With care, DSLs can be both highly productive and high performance
  – Potential for a revolutionary impact on computational science in many domains

• Production-quality DSLs require lots of work
  – We knew it was a big project, but it turned out to be even bigger!

• Production-quality DSLs benefit from close collaboration of computer and domain scientists

• The greater the generality, the greater the challenge
  – May or may not be better for users
  – Requires serious thought

• If you’re inexperienced developing DSLs, do a lot of prototyping
  – Opportunity to test out ideas on users (domain scientists) and see if they’re practical

• Think about how you will test generated code