

# Components for Scientific Computing: An Introduction to the Common Component Architecture

**David E. Bernholdt**

Computer Science and Mathematics Division  
Oak Ridge National Laboratory

*In collaboration with the CCA Forum*

***<http://www.cca-forum.org>***

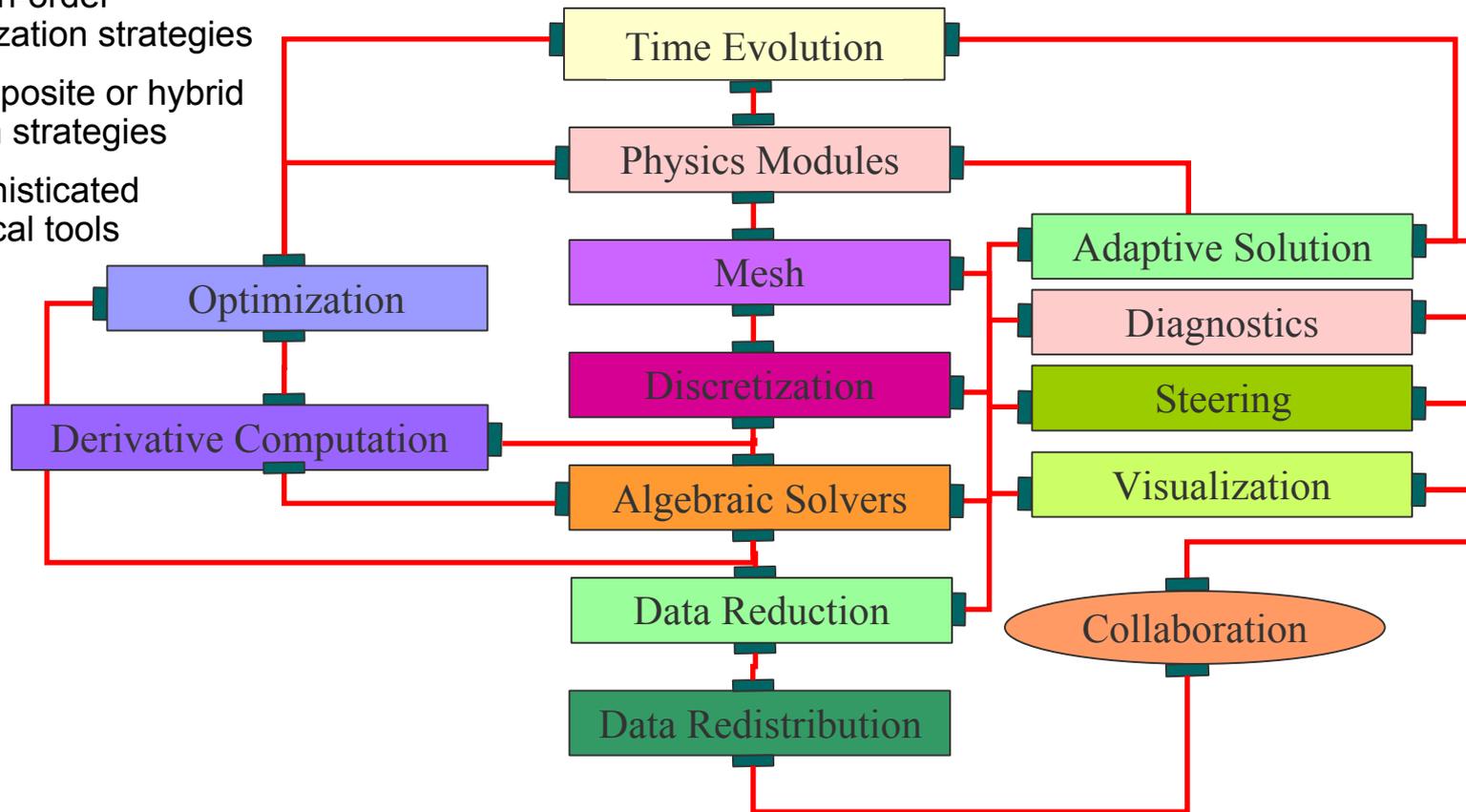
Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, U.S. Dept. of Energy. Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725

# Modern Scientific Software Engineering Challenges

- **Productivity**
  - Time to first solution (prototyping)
  - Time to solution (“production”)
  - Software infrastructure requirements
- **Complexity**
  - Increasingly sophisticated models
  - Model coupling – multi-scale, multi-physics, etc.
  - “Interdisciplinarity”
- **Performance**
  - Increasingly complex algorithms
  - Increasingly complex computers
  - Increasingly demanding applications

# Modern Scientific Software Development

- Terascale computing will enable high-fidelity calculations based on multiple coupled physical processes and multiple physical scales
  - Adaptive algorithms and high-order discretization strategies
  - Composite or hybrid solution strategies
  - Sophisticated numerical tools



# Component-Based Software Engineering

- CBSE methodology is emerging, especially popular in business and internet areas
- **Software productivity**
  - Provides a “plug and play” application development environment
  - Many components available “off the shelf”
  - Abstract interfaces facilitate reuse and interoperability of software
- *“The best software is code you don’t have to write” [Jobs]*
- **Software complexity**
  - Components encapsulate much complexity into “black boxes”
  - Plug and play approach simplifies applications & adaptation
  - Model coupling is natural in component-based approach
- **Software performance (indirect)**
  - Plug and play approach and rich “off the shelf” component library simplify changes to accommodate different platforms

# Motivation: For Library Developers

- People want to use your software, but need wrappers in languages you don't support
  - Many component models provide language interoperability
- Discussions about standardizing interfaces are often sidetracked into implementation issues
  - Components separate interfaces from implementation
- You want users to stick to your published interface and prevent them from stumbling (prying) into the implementation details
  - Most component models actively enforce the separation

# Motivation: For Application Developers and Users

- You have difficulty managing **multiple third-party libraries** in your code
- You (want to) use **more than two languages** in your application
- Your code is **long-lived** and different pieces **evolve** at different rates
- You want to be able to **swap** competing implementations of the same idea and **test** without modifying any of your code
- You want to **compose** your application with some other(s) that weren't originally designed to be combined

# The “Sociology” of Components

- Components need to be **shared** to be truly useful
  - Sharing can be at several levels
    - Source, binaries, remote service
  - Various models possible for **intellectual property/licensing**
    - Components with different IP constraints can be **mixed in a single application**
- Peer component models facilitate **collaboration** of groups on software development
  - Group decides overall **architecture** and **interfaces**
  - Individuals/sub-groups create individual **components**

# Who Writes Components?

- “Everyone” involved in creating an application can/should create components
  - Domain scientists as well as computer scientists and applied mathematicians
  - Most will also use components written by other groups
- Allows developers to focus on their interest/specialty
  - Get other capabilities via reuse of other’s components
- Sharing components within scientific domain allows everyone to be more productive
  - Reuse instead of reinvention
- As a unit of publication, a well-written and –tested component is like a high-quality library
  - Should receive same degree of recognition
  - Often a more appropriate unit of publication/recognition than an entire application code

# What *are* Components?

- No universally accepted definition...yet
- **A unit of software 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
  - Components *may* maintain state information
  - But external access to state info must be through an interface (*not a common block*)
  - File-based interactions can be recast using an “I/O component”
- **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 are composed to form an application and executed (**framework**)
  - The rights and responsibilities of the framework

# 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

# Domain-Specific Frameworks vs General Component Architectures

## Domain-specific

- Often known as “frameworks”
- Provide a significant software infrastructure to support applications in a **given domain**
  - Often attempts to generalize an existing large application
- Often hard to adapt to use outside the original domain
  - Tend to assume a **particular structure/workflow** for application
- Relatively **common**

## General

- Provide the infrastructure to **hook components** together
  - Domain-specific infrastructure can be built as more components
- Usable in **many domains**
  - Few assumptions about application
  - **More opportunities for reuse**
- Better supports **model coupling** across traditional domain boundaries
- Relatively **rare** at present
  - Commodity component models often not so useful in HPC scientific context

# 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?
- “Commodity” component models may have problems
  - CORBA, COM, JavaBeans

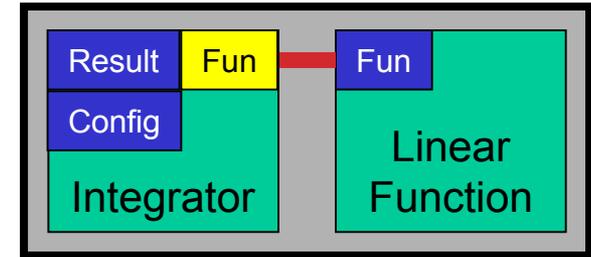
# What is the CCA? (User View)

- A component model specifically designed for **high-performance scientific computing**
- Supports both **parallel and distributed** applications
- Designed to be implementable without sacrificing **performance**
- **Minimalist approach** makes it easier to componentize existing software
- A **tool** to enhance the productivity of scientific programmers
  - Make the hard things easier, make some intractable things tractable
  - Support & promote reuse & interoperability
  - **Not a magic bullet**

# Basic CCA Terminology

- **Port** (aka *interface*)

- Procedural interface (not just dataflow!)
- Like C++ abst. virtual class, Java interface
- Uses/provides design pattern



- **Component**

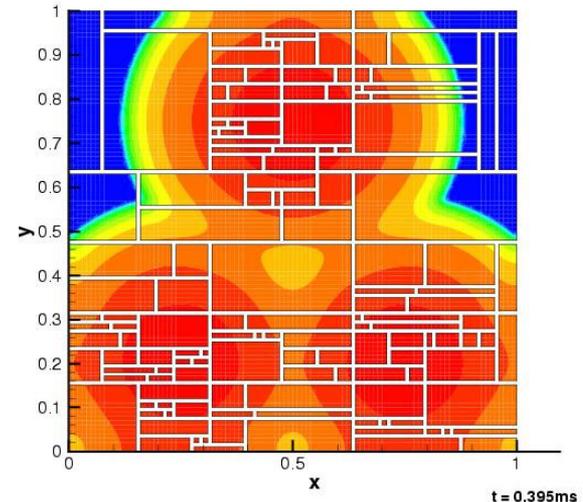
- A unit of software deployment/reuse (i.e. has interesting functionality)
- Interacts with the outside world only through well-defined **interfaces**
- Implementation is opaque to the outside world
- Components are **peers**

- **Framework**

- Holds **components** during application composition and execution
- Controls the “**exchange**” of **interfaces** between components (while ensuring implementations remain hidden)
- Provides a small set of standard, ubiquitous services to components
  - *CCA spec doesn't specify a framework per se, so components can be constructed to provide framework-like services*

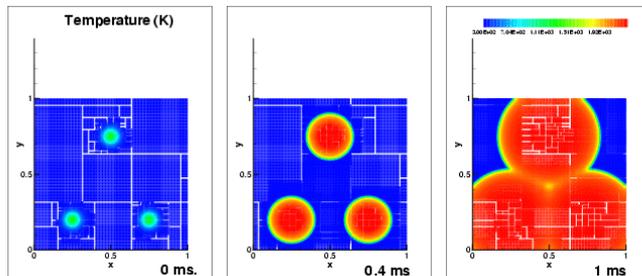
# Computational Facility for Reacting Flow Science (CFRFS)

- SciDAC BES project, H. Najm PI
- **Investigators:** Sofia Lefantzi (SNL), Jaideep Ray (SNL), Sameer Shende (Oregon)
- **Goal:** A “plug-and-play” toolkit environment for flame simulations
- **Problem Domain:** Structured adaptive mesh refinement solutions to reaction-diffusion problems



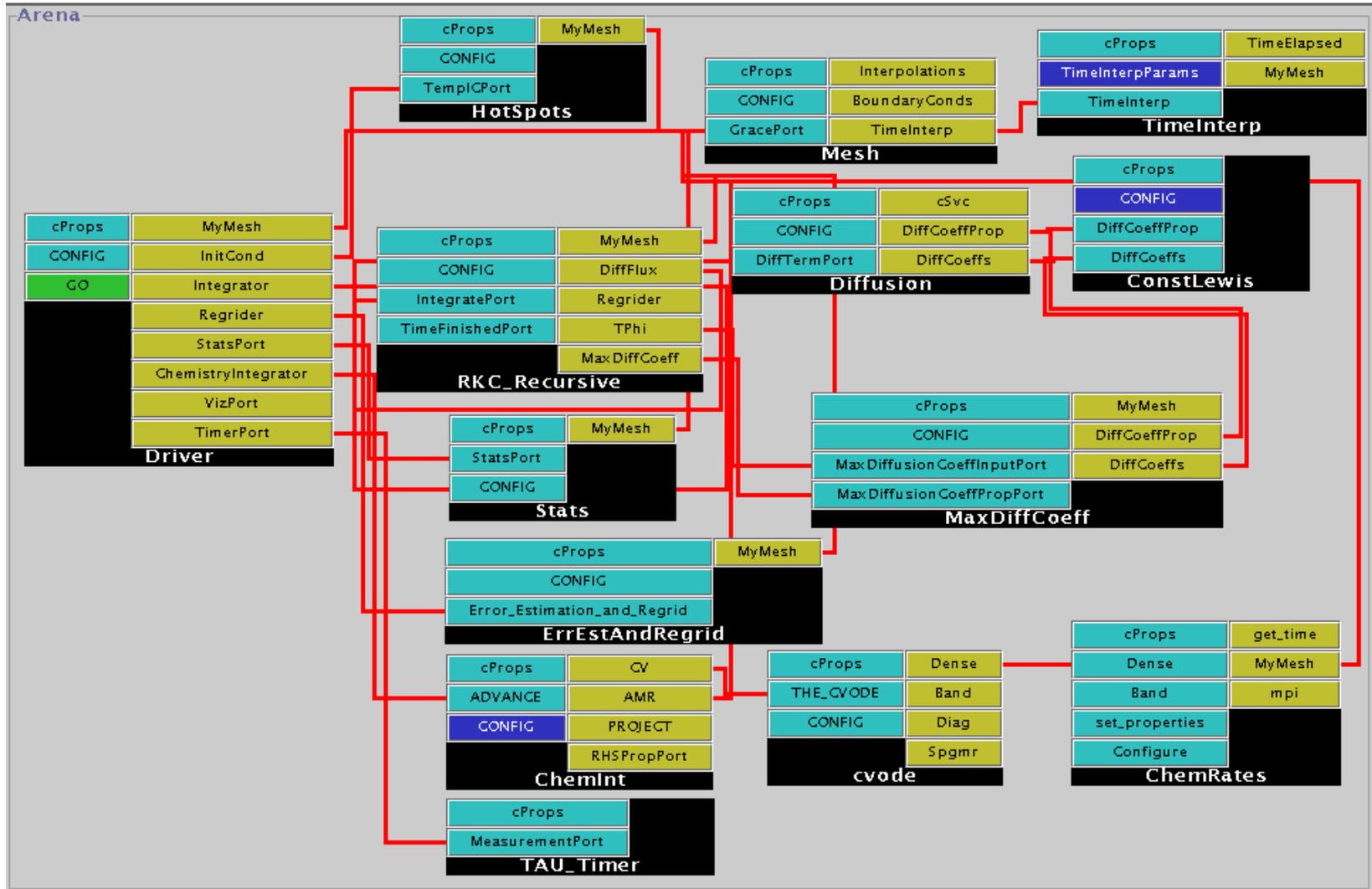
# Scientific and Technical Summary

- H<sub>2</sub>-Air ignition on a structured adaptive mesh, with an operator-split formulation
- RKC for non-stiff terms, BDF for stiff
- 9-species, 19-reactions, stiff mechanism
- 1cm x 1cm domain; max resolution = 12.5 microns
- Kernel for a 3D, adaptive mesh low Mach number flame simulation capability in SNL, Livermore
- Components are usually in C++ or wrappers around old F77 code
- Developed numerous components
  - Integrator, spatial discretizations, chemical rates evaluator, transport property models, timers etc.
  - Structured adaptive mesh, load-balancers, error-estimators (for refining/coarsening)
  - In-core, off-machine, data transfers for post-processing

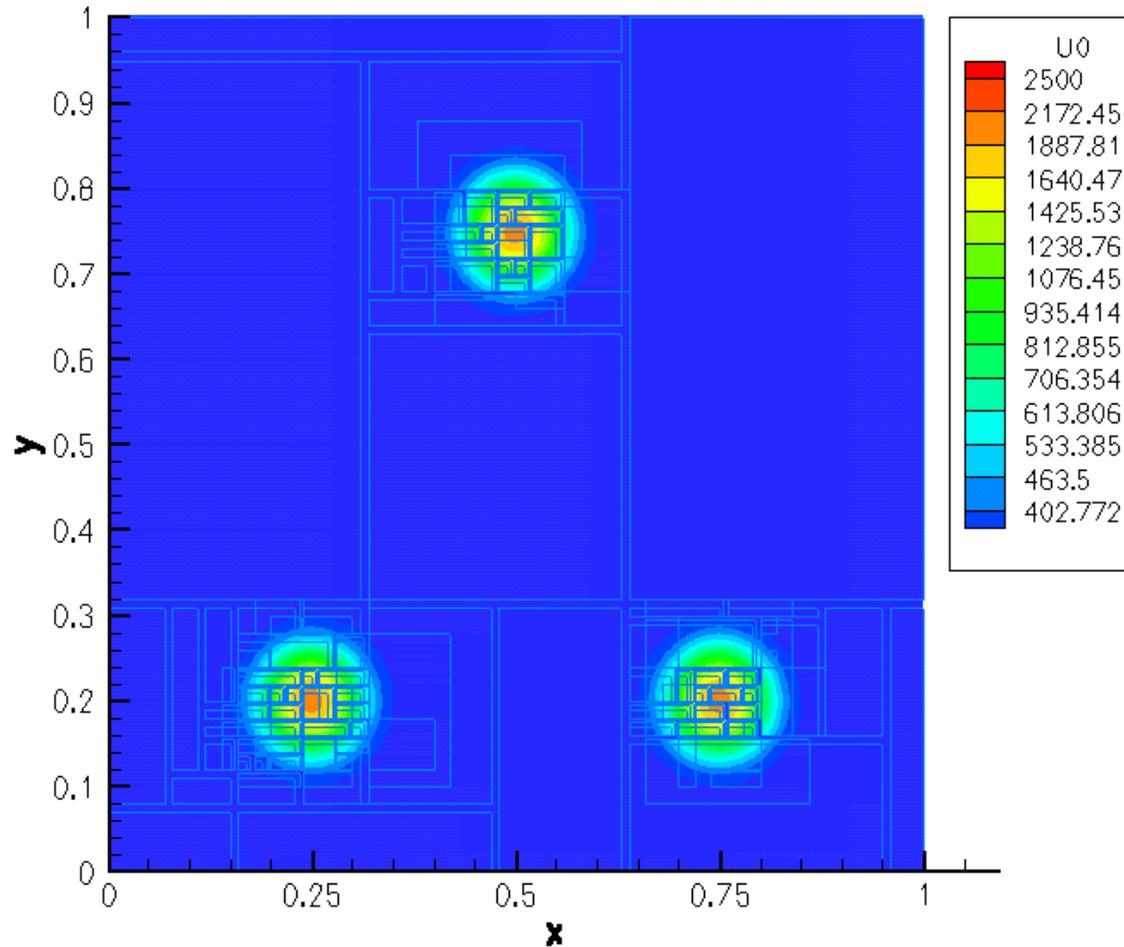


- TAU for timing (Oregon, PERC)
- CVODES integrator (LLNL, TOPS)

# “Wiring Diagram” for CFRFS App.

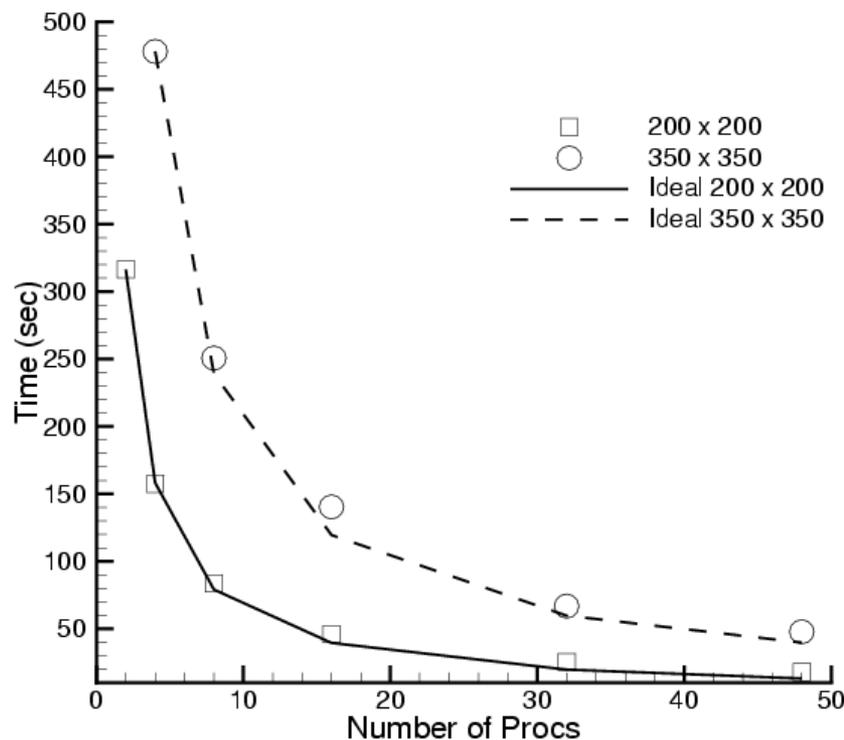


# Animation of the Temperature Field



# Scalability of Scientific Data Components in CFRFS Combustion Applications

- Investigators: S. Lefantzi, J. Ray, and H. Najm (SNL)
- Uses GrACEComponent, CvodesComponent, etc.
- 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



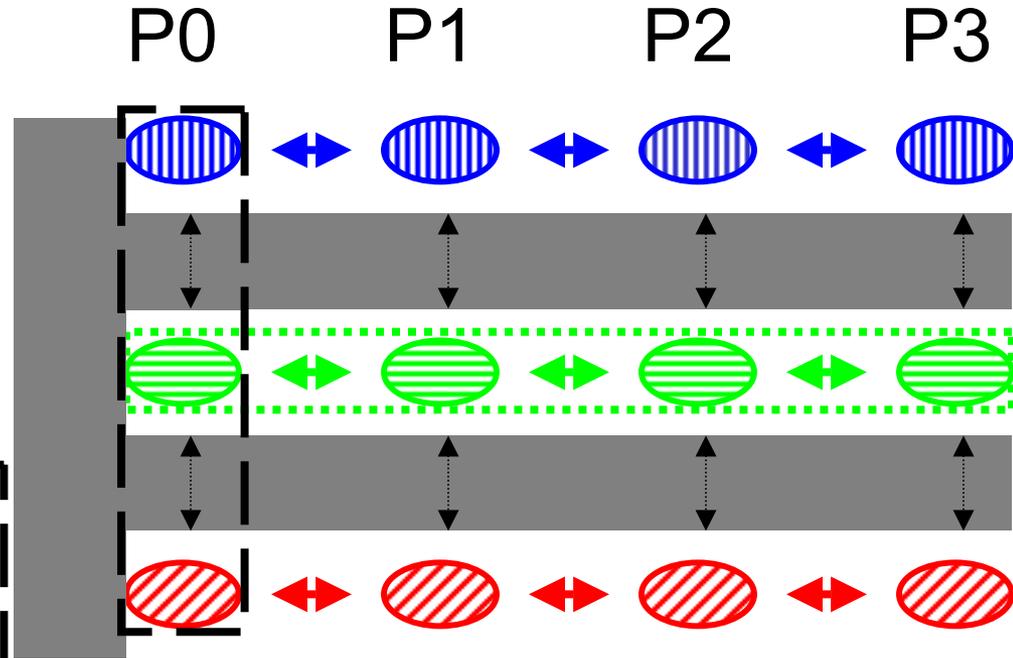
Reference: S. Lefantzi, J. Ray, and H. Najm, Using the Common Component Architecture to Design High Performance Scientific Simulation Codes, *Proc of Int. Parallel and Distributed Processing Symposium*, Nice, France, 2003, accepted.

# CCA Framework Stays “Out of the Way” of Component Parallelism

- 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)**

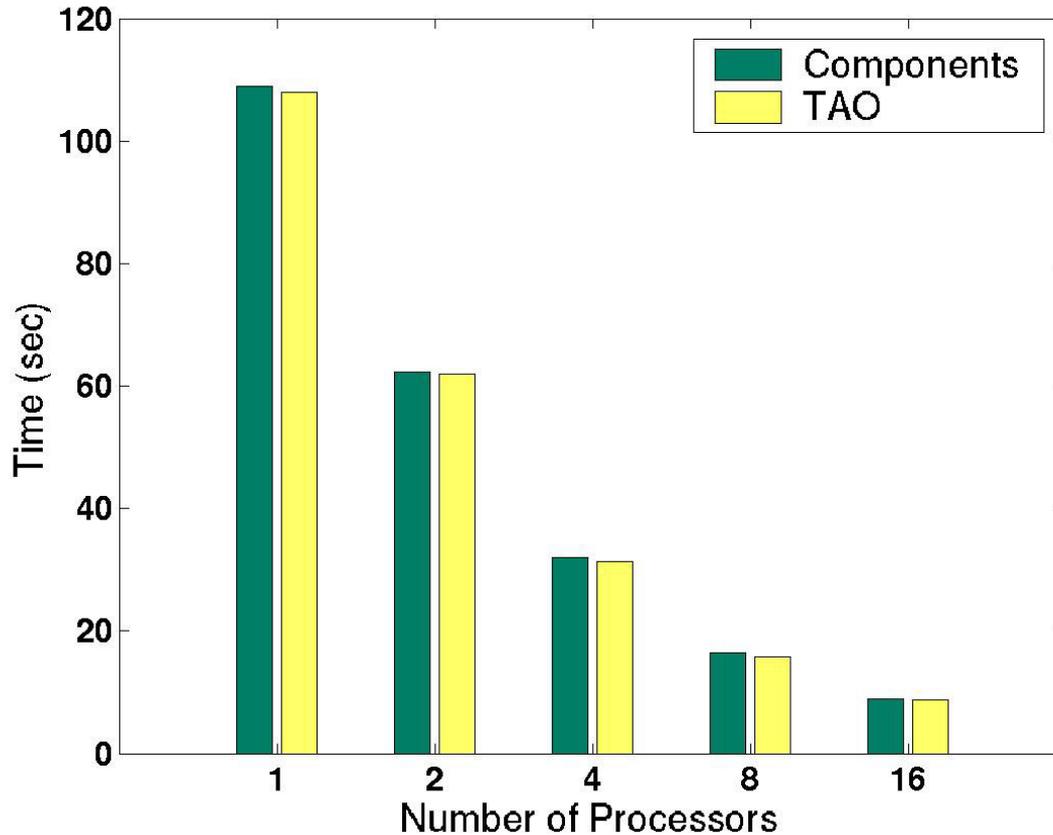


Components: Blue, Green, Red

Framework: Gray

*MCMD/MPMD also supported  
Other component models  
ignore parallelism entirely*

# Scalability on a Linux Cluster



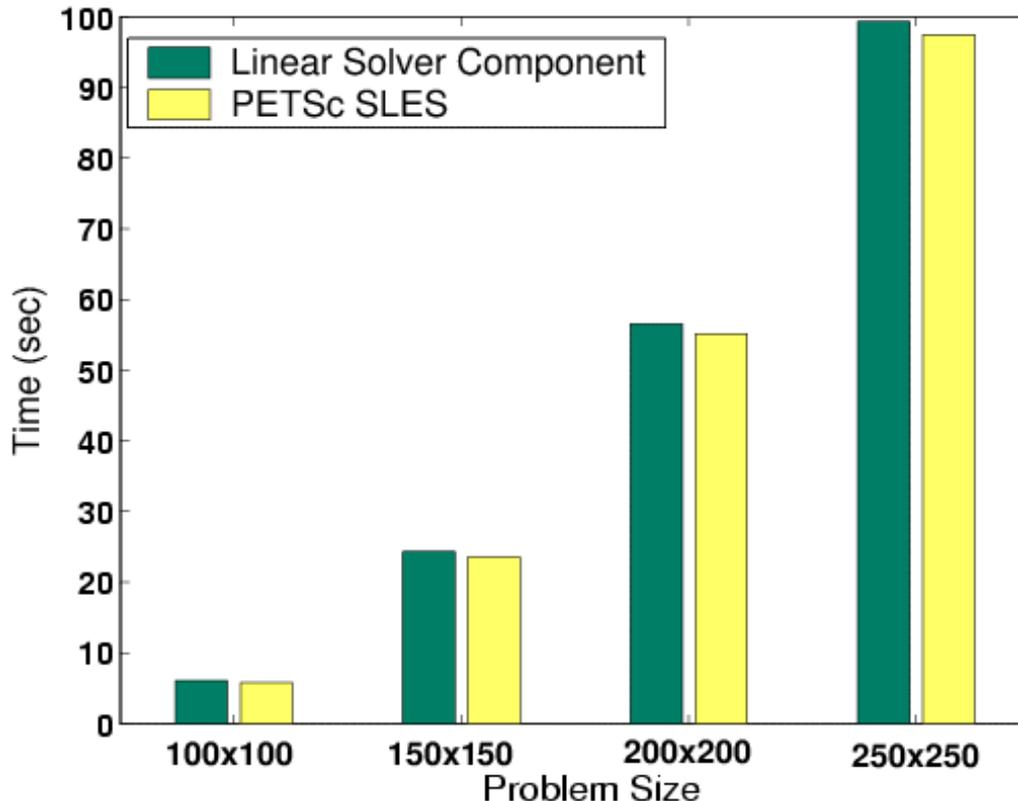
Total execution time for the minimum surface minimization problem using a fixed-sized 250x250 mesh.

- Newton method with line search
- Solve linear systems with the conjugate gradient method and block Jacobi preconditioning (with no-fill incomplete factorization as each block's solver, and 1 block per process)
- Negligible component overhead; good scalability

# “Direct Connection” Maintains Local Performance

- Calls *between* components equivalent to a C++ **virtual function call**: lookup function location, invoke it
  - Cost equivalent of **~2.8 F77 or C function calls**
  - **~48 ns vs 17 ns** on 500 MHz Pentium III Linux box
- **Language interoperability** can impose additional overheads
  - Some arguments require conversion
  - Costs vary, but small for typical scientific computing needs
- Calls **within** components have **no CCA-imposed overhead**
- **Implications**
  - **Be aware of costs**
  - Design so inter-component calls **do enough work** that overhead is negligible

# Component Overhead

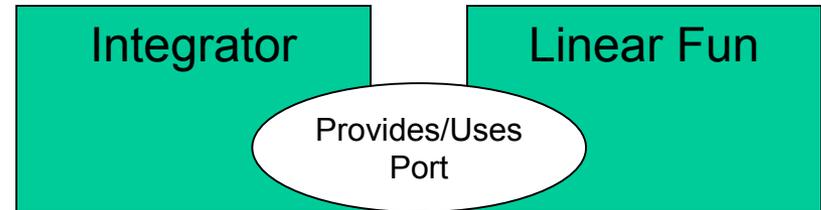


Aggregate time for linear solver component in unconstrained minimization problem.

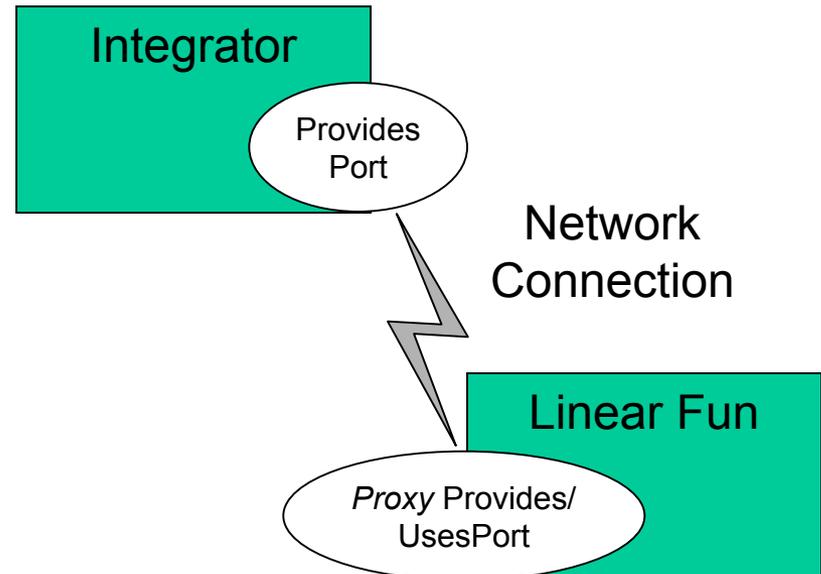
- Negligible overhead for component implementation and abstract interfaces when using appropriate levels of abstraction
- Linear solver component currently supports any methods available via the ESI interfaces to PETSc and Trilinos; plan to support additional interfaces the future, e.g., those under development within the TOPS center
- Here: Use the conjugate gradient method with no-fill incomplete factorization preconditioning

# Distributed Computing Is Also Supported

- “Direct connection” preserves high performance of local (“in-process”) components
  - Framework makes *connection*
  - But is not involved in *invocation*
- Distributed computing has same uses/provides semantics, 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



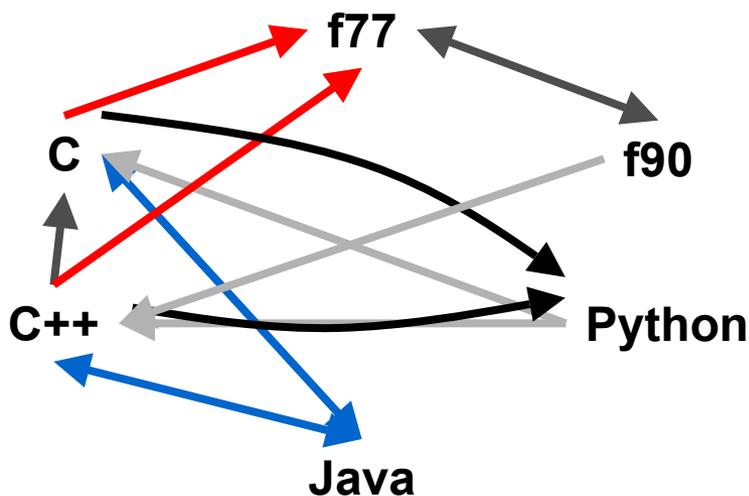
Direct Connection



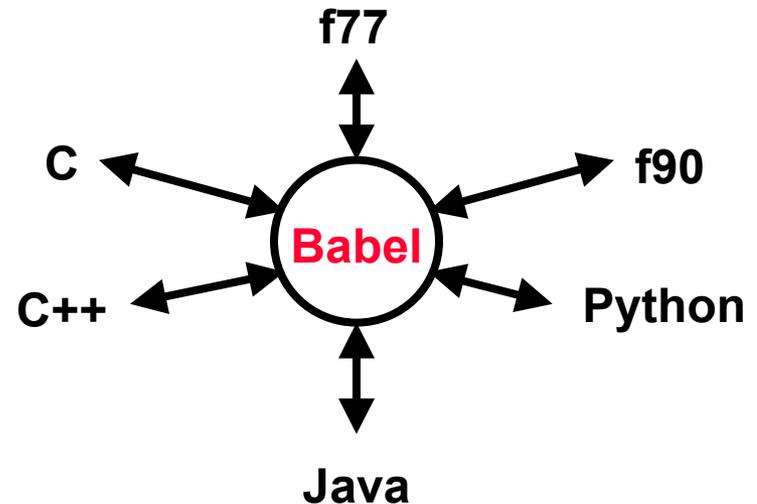
Network Connection

# Language Interoperability

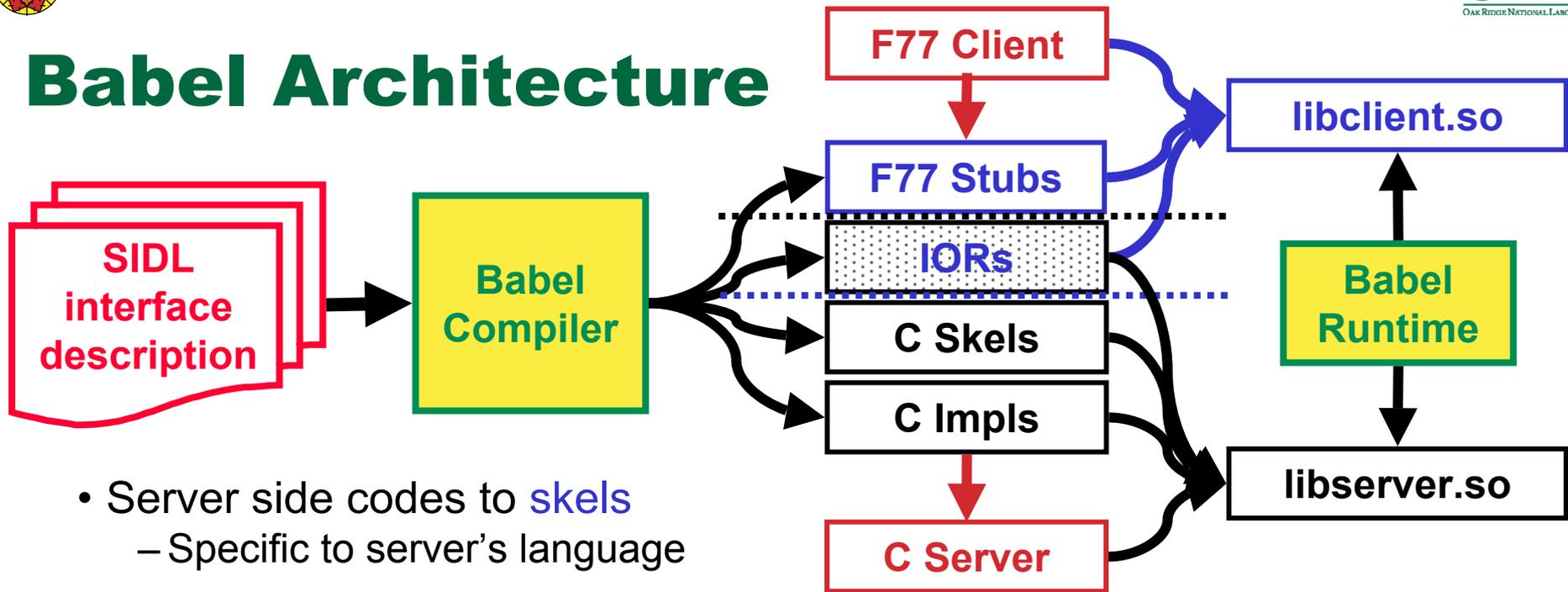
- Existing language interoperability approaches are “point-to-point” solutions



- Babel provides a unified approach in which all languages are considered peers
- Babel used primarily at interfaces

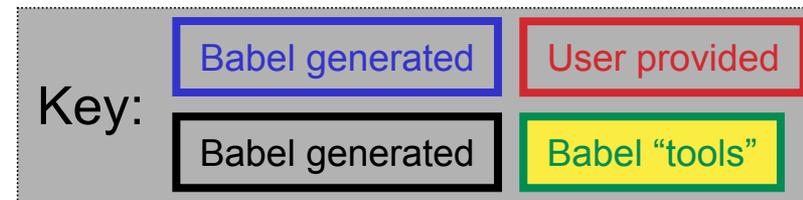


# Babel Architecture



- Server side codes to **skels**
  - Specific to server’s language
- Client side codes to **stubs**
  - Specific to client’s language
- Internal Object Representation (**IOR**) bridges between
  - Implemented in C
- Babel generates glue code from **SIDL** specification of interface

- Babel includes both **code generation** and **runtime** components
- Strives to allow **natural-looking code** in each supported language



# Scientific Interface Definition Language (SIDL)

```
version functions 1.0;  
  
package functions {  
    interface Function extends gov.cca.Port {  
        double evaluate( in double x );  
    }  
}
```

- SIDL expresses only interfaces
  - Not a full-fledged programming language
- Interface definition is basis of code generation for both client and server
- Server side also needs SIDL expression of what's implemented

# Additional SIDL for Server Side

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

- **Objects:** Interfaces, Abstract Classes, Concrete Classes
- **Methods:** *all public*; virtual, static, final
- **Mode:** in, out, inout (like CORBA, not quite like F90)
- **Types:** bool, char, int, long, float, double, fcomplex, dcomplex, array<Type,Dimension>, enum, interface, class

# Interfaces, Interoperability, and Reuse

- Interfaces define how components interact...
- Therefore interfaces are key to interoperability and reuse of components
- In many cases, “any old interface” will do, but...
- General plug and play interoperability requires **multiple implementations** providing the same interface
- Reuse of components occurs when they provide interfaces (functionality) needed in **multiple applications**

# Designing for Reuse

- Designing for interoperability and reuse requires “standard” interfaces
  - Typically domain-specific
  - “Standard” need not imply a formal process, may mean “widely used”
- Generally means collaborating with others
- *Higher* initial development cost (amortized over multiple uses)
- Reuse implies longer-lived code
  - thoroughly tested
  - highly optimized
  - improved support for multiple platforms

# Adapting Existing Code into Components

- Suitably structured code (programs, libraries) should be relatively easy to adapt to CCA
- Decide **level of componentization**
  - Can evolve with time (start with coarse components, later refine into smaller ones)
- Define **interfaces** and write wrappers between them and existing code
- Add **framework interaction code** for each component
  - `setServices`, constructor, destructor
- Modify component internals to **use other components** as appropriate
  - `getPort`, `releasePort` and method invocations

# CCA-Related Organizations

## CCA Forum

- Standards body for CCA
  - CCA Specification
- Promote/facilitate interface development
- Goal: interoperability
- Open membership
- Quarterly meetings
  - Dates set ~1 year ahead
  - Next: 30-31 October, Santa Fe, NM

## CCTTSS

- DOE-funded SciDAC Center
- Develop “prototype” stage to full production environment
- Understand how to use component architectures effectively in HPC environments
- Subset of CCA Forum
  - ANL, LANL, LLNL, PNNL, ORNL, SNL, Indiana, Utah

# CCA Research Thrusts and Application Domains

- Frameworks
  - Framework interoperability
  - Language interoperability
  - Deployment
- Scientific Components
  - Component suite dev.
  - Basic data interfaces
  - Numerical quality of svc.
- MxN Parallel Data Redistribution
  - Component-based
  - Framework-based
- Application Outreach
  - Education
  - Best practices for use
  - Chemistry, Climate

## SciDAC:

- *Combustion (CFRFS)*
- *Climate Modeling (CCSM)*
- *Meshing Tools (TSTT)*
- *(PDE) Solvers (TOPS)*
- *IO, Poisson Solvers (APDEC)*
- *Fusion (CMRS)*
- Supernova simulation (TSI)
- Accelerator simulation (ACCAST)

## DOE Outside of SciDAC:

- *ASCI: C-SAFE, Views, Data Svc's*
- *Quantum Chemistry*

## Outside of DOE:

- *NASA: ESMF, SWMF*
- Etc....

# Component Inventory:

## Data Management, Meshing and Discretization

- **Global Array Component** – M. Krishnan and J. Nieplocha (PNNL) – classic and SIDL – Provides capabilities for manipulating multidimensional dense distributed arrays; supports DADF common interface.
- **TSTTMesh** – L.F. Diachin (SNL, formerly ANL) – classic – Provides prototype capabilities for querying unstructured meshes based on interfaces being designed within the TSTT SciDAC Center.
- **FEMDiscretization** – L.F. Diachin (SNL, formerly ANL) – classic – Provides finite element discretization of diffusion and advection PDE operators and linear system assembly capabilities.
- **GrACEComponent** – J. Ray (SNL) – classic – Provides parallel AMR infrastructure, which follows a hierarchy-of-patches methodology for meshing and includes load-balancing; based on GrACE (Rutgers); being used in combustion applications within the SciDAC CFRFS project.

# Component Inventory:

## Integration, Optimization, and Linear Algebra

- ***CvodesComponent*** – Radu Serban (LLNL, TOPS collaborator) – classic – Provides a generic implicit ODE integrator and an implicit ODE integrator with sensitivity capabilities; based on CVODES; used in combustion applications within the CFRFS.
- ***TaoSolver*** – S. Benson, L.C. McInnes, B. Norris, and J. Sarich (ANL) – SIDL – Provides solvers for unconstrained and bound constrained optimization problems, which build on infrastructure within TAO (ANL); uses external linear algebra capabilities.
- ***LinearSolver*** – B. Norris (ANL) – classic – Provides a prototype linear solver port; in the process of evolving to support common interfaces for linear algebra that are under development within the TOPS SciDAC center.

# Component Inventory:

## Parallel Data Description, Redistribution, and Visualization

- ***DistArrayDescriptorFactory*** – D. Bernholdt and W. Elwasif (ORNL) – classic – Provides a uniform means for applications to describe dense multi-dimensional arrays; based upon emerging interfaces from the CCA Scientific Data Components Working Group.
- ***CumulvsMxN*** – J. Kohl, D. Bernholdt, and T. Wilde (ORNL) – classic – Builds on CUMULVS (ORNL) technology to provide an initial implementation of parallel data redistribution interfaces that are under development by the CCA “MxN” Working Group.
- ***VizProxy*** – J. Kohl and T. Wilde (ORNL) – classic – Provides a companion “MxN” endpoint for extracting parallel data from component-based applications and then passing this data to a separate front-end viewer for graphical rendering and presentation. Variants exist for structured data and unstructured triangular mesh data as well as text-based output.

# Component Inventory:

## Services, Graphical Builders, and Performance

- ***Ccaffeine Services*** – B. Allan, R. Armstrong, M. Govindaraju, S. Lefantzi, and E. Walsh (SNL) – classic and SIDL – Services for parameter ports, connections between SIDL and classic ports, MPI access, connection events, etc.
- ***Graphical Builders*** – B. Norris (ANL) and S. Parker (Univ. of Utah) – Prototype graphical builders that can be used to assemble components, set parameters, execute, and monitor component-based simulations.
- ***Performance Observation*** – S. Shende and A. Malony (Univ. of Oregon), C. Rasmussen and M. Sotille (LANL), and J. Ray (SNL) – classic and SIDL – Provides measurement capabilities to components, thereby aiding in the selection of components and helping to create performance aware intelligent components; based on TAU (Oregon).

# Applications: Computational Chemistry

- **Molecular Optimization**
- **Investigators:** Steve Benson (ANL), Curtis Janssen (SNL), Liz Jurriss (PNNL), Manoj Krishnan (PNNL), Lois McInnes (ANL), Jarek Nieplocha (PNNL), Boyana Norris (ANL), Jason Sarich (ANL), Theresa Windus (PNNL)
- **Goal:** Demonstrate interoperability between packages, develop experience with large existing code bases, seed interest in Chemistry domain
- **Problem Domain:** Optimization of molecular structures using quantum chemical methods
- **Advanced Software for the Calculation of Thermochemistry, Kinetics, and Dynamics**
- SciDAC BES project, Al Wagner PI
- **Investigators:** Ronald Duchovic (Indiana-Purdue Fort Wayne), Wael Elwasif (ORNL), Lois McInnes (ANL), Craig Rasmussen (LANL)
- **Goal:** Develop a standard interface to provide numerical potential energy surface information to reaction dynamics codes
- **Problem Domain:** Reaction dynamics

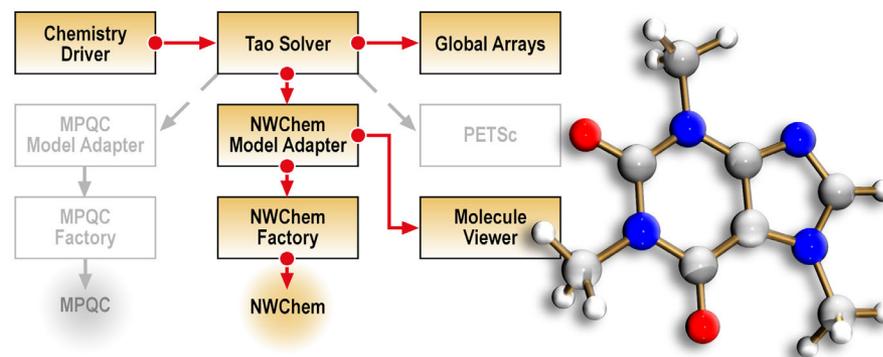
# Scientific and Technical Summary

## Molecular Optimization

- Decouple geometry optimization from electronic structure
- Demonstrate interoperability of electronic structure components
- Build towards more challenging optimization problems, e.g., protein/ligand binding studies
- Software:
  - Electronic structure: MPQC, NWChem,
  - Optimization: TAO
  - Linear algebra: Global Arrays, PETSc

## Reaction Dynamics

- Software: POTLIB
- Developing a component interface for POTLIB
  - Original interface makes extensive use of Fortran common blocks



# Applications: Climate Modeling

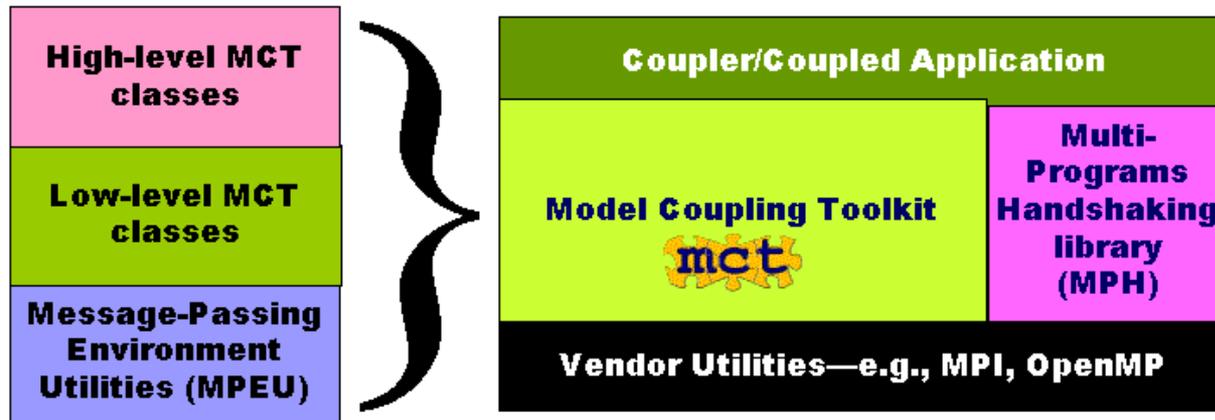
## Community Climate System Model

- SciDAC BER project, John Drake and Robert Malone PIs
- **Goals:** Investigate model coupling and parameterization-level componentization within models
- **Investigators:** John Drake (ORNL), Wael Elwasif (ORNL), Michael Ham (ORNL), Jay Larson (ANL), Everest Ong (ANL)

## Earth System Modeling Framework

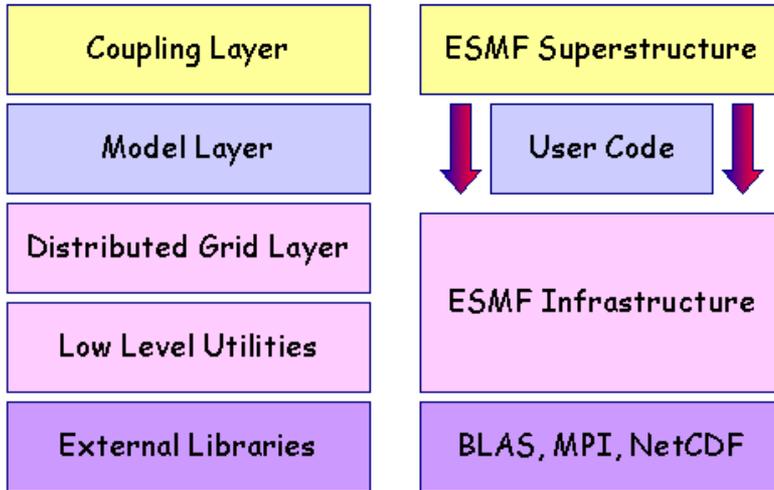
- NASA project, Tim Killeen, John Marshall, and Arlindo da Silva PIs
- **Goal:** Build domain-specific framework for the development of climate models

# Community Climate System Model

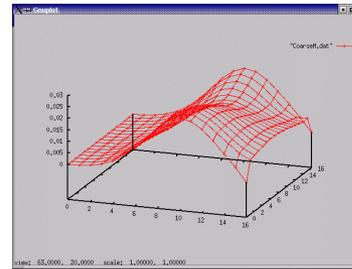


- Model Coupling Toolkit (MCT)
  - Coupler for CCSM
  - Basis for ESMF coupler
  - Contributions to MxN
  - River runoff model
- Community Atmosphere Model (CAM)
  - Componentization at physics/dynamics interface

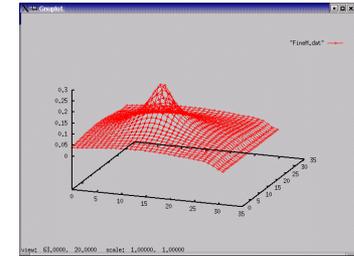
# Earth System Modeling Framework



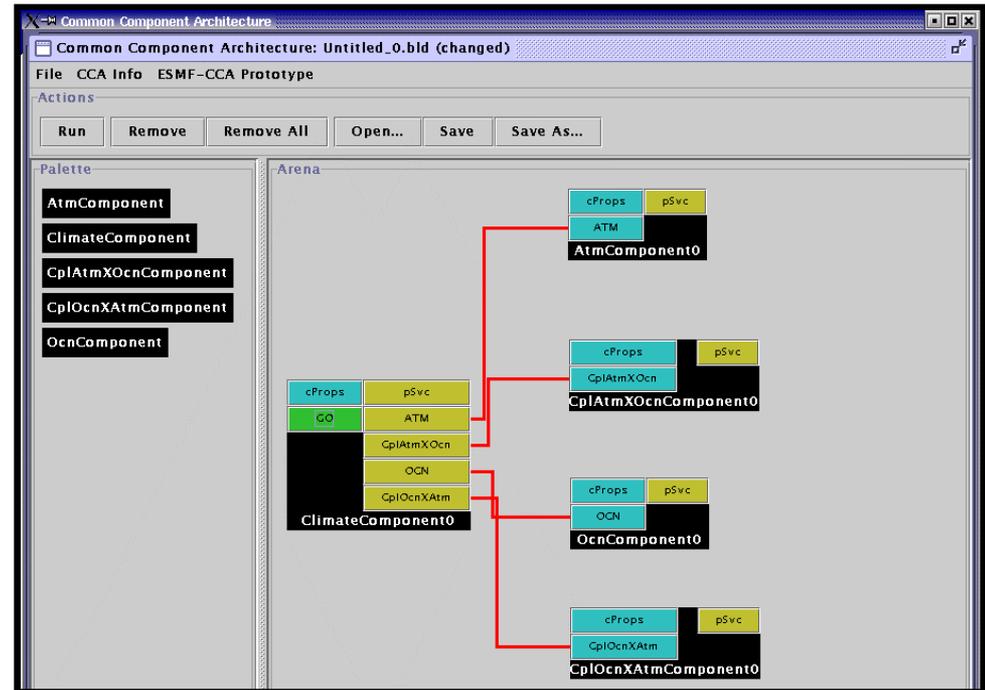
ATM



OCN



- Prototype superstructure
- Investigating grid layer interfaces



Courtesy Shujia Zhou, NASA Goddard

# What the CCA isn't...

- CCA doesn't specify who owns "main"
  - CCA components are peers
  - Up to application to define component relationships
    - "Driver component" is a common design pattern
- CCA doesn't specify a parallel programming environment
  - Choose your favorite
  - Mix multiple tools in a single application
- CCA doesn't specify I/O
  - But it gives you the infrastructure to create I/O components
  - Use of stdio may be problematic in mixed language env.
- CCA doesn't specify interfaces
  - But it gives you the infrastructure to define and enforce them
  - CCA Forum supports & promotes "standard" interface efforts
- CCA doesn't require (but does support) separation of algorithms/physics from data

# What the CCA is...

## (a.k.a. Summary)

- CCA is a *specification* for a component environment
  - Fundamentally, a design pattern
  - Multiple “reference” implementations exist
  - Being used by applications
- CCA increases productivity
  - Supports and promotes software interoperability and reuse
  - Provides “plug-and-play” paradigm for scientific software
- CCA offers the flexibility to architect your application as you think best
  - Doesn’t dictate component relationships, programming models, etc.
  - Minimal performance overhead
  - Minimal cost for incorporation of existing software
- CCA provides an environment in which domain-specific application frameworks can be built
  - While retaining opportunities for software reuse at multiple levels

# Information Pointers & Acknowledgements

- <http://www.cca-forum.org>
  - Tutorial presentations, software, etc. available
- Tutorials
  - SC2003 (November)
  - SIAM Parallel Processing '04 (February)
  - Before most CCA Forum meetings (quarterly)
- Personal contact with me or other CCA people
- Thanks to the *many* people who have contributed to the development and use of the CCA, who's work this talk represents!