



**CCA**

Common Component Architecture

## Welcome to the Common Component Architecture Tutorial

ACTS Collection Workshop  
27 August 2004

**CCA Forum Tutorial Working Group**  
<http://www.cca-forum.org/tutorials/>  
[tutorial-wg@cca-forum.org](mailto:tutorial-wg@cca-forum.org)

1



**CCA**  
Common Component Architecture

## Agenda & Table of Contents

Time	Title	Slide No.	Presenter		
11:30-11:35am	Welcome	1	David Bernholdt, ORNL		
11:35am-12:30pm	A Pictorial Introduction to Components in Scientific Computing	6	David Bernholdt, ORNL		
	An Introduction to Components & the CCA	26	David Bernholdt, ORNL		
12:30-1:30pm	<i>Lunch</i>				
1:30pm-1:50pm	Distributed Computing with the CCA	67	David Bernholdt, ORNL		
1:50-2:50pm	CCA Applications	87	Jaideep Ray, SNL		
2:50-3:20pm	Language Interoperable CCA Components with Babel	136	Tom Epperly, LLNL		
3:20-3:30pm	Questions/Discussion	The Team			
3:30-4:00pm	<i>Break/Relocate to Tolman Hall</i>				
4:00-4:45pm	<i>TAU Hands-On</i>				
4:45-6:30pm	CCA Hands-On	Hand-out	Rob Armstrong, SNL & the Team		

2



## The Common Component Architecture (CCA) Forum

- Combination of standards body and user group for the CCA
- Define Specifications for **High-Performance** Scientific Components & Frameworks
- Promote and Facilitate Development of Domain-Specific **"Standard" Interfaces**
- Goal: **Interoperability** between components developed by different expert teams across different institutions
- Quarterly Meetings, Open membership...

Mailing List: [cca-forum@cca-forum.org](mailto:cca-forum@cca-forum.org)

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

3



## Acknowledgements: Tutorial Working Group

- **People:** Rob Armstrong, David Bernholdt, Randy Bramley, Wael Elwasif, Lori Freitag Diachin, Madhusudhan Govindaraju, Ragib Hasan, Dan Katz, Jim Kohl, Gary Kumlert, Lois Curfman McInnes, Boyana Norris, Craig Rasmussen, Jaideep Ray, Sameer Shende, Torsten Wilde, Shujia Zhou
- **Institutions:** ANL, Binghamton U, Indiana U, JPL, LANL, LLNL, NASA/Goddard, ORNL, SNL, U Illinois, U Oregon
- **Computer facilities** provided by the Computer Science Department and University Information Technology Services of Indiana University, supported in part by NSF grants CDA-9601632 and EIA-0202048.

4



Common Component Architecture

## Acknowledgements: The CCA

- **ANL** –Steve Benson, Jay Larson, Ray Loy, Lois Curfman McInnes, Boyana Norris, Everest Ong, Jason Sarich...
- **Binghamton University** - Madhu Govindaraju, Michael Lewis, ...
- **Indiana University** - Randall Bramley, Dennis Gannon, ...
- **JPL** – Dan Katz, ...
- **LANL** - Craig Rasmussen, Matt Sotille, ...
- **LLNL** – Lori Freitag Diachin, Tom Epperly, Scott Kohn, Gary Kumfert, ...
- **NASA/Goddard** – Shujia Zhou
- **ORNL** - David Bernholdt, Wael Elwasif, Jim Kohl, Torsten Wilde, ...
- **PNNL** - Jarek Nieplocha, Theresa Windus, ...
- **SNL** - Rob Armstrong, Ben Allan, Lori Freitag Diachin, Curt Janssen, Jaideep Ray, ...
- **University of Oregon** – Allen Malony, Sameer Shende, ...
- **University of Utah** - Steve Parker, ...

and many more... without whom we wouldn't have much to talk about!

5



**CCA**

Common Component Architecture

## A Pictorial Introduction to Components in Scientific Computing

**CCA Forum Tutorial Working Group**  
<http://www.cca-forum.org/tutorials/>  
*tutorial-wg@cca-forum.org*

6

CCA  
Common Component Architecture

## Once upon a time...

A diagram illustrating a simple input-output process. At the top, a white document icon is labeled "Input". A red arrow points down to a green 3D block labeled "Program". From the right side of the "Program" block, another red arrow points to a white document icon labeled "Output".

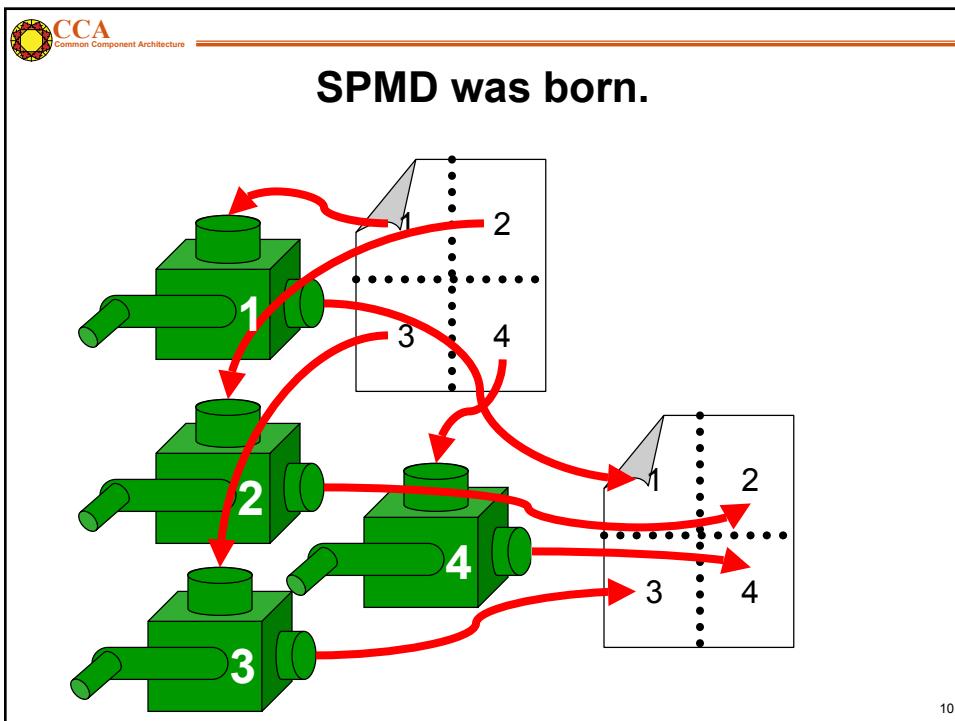
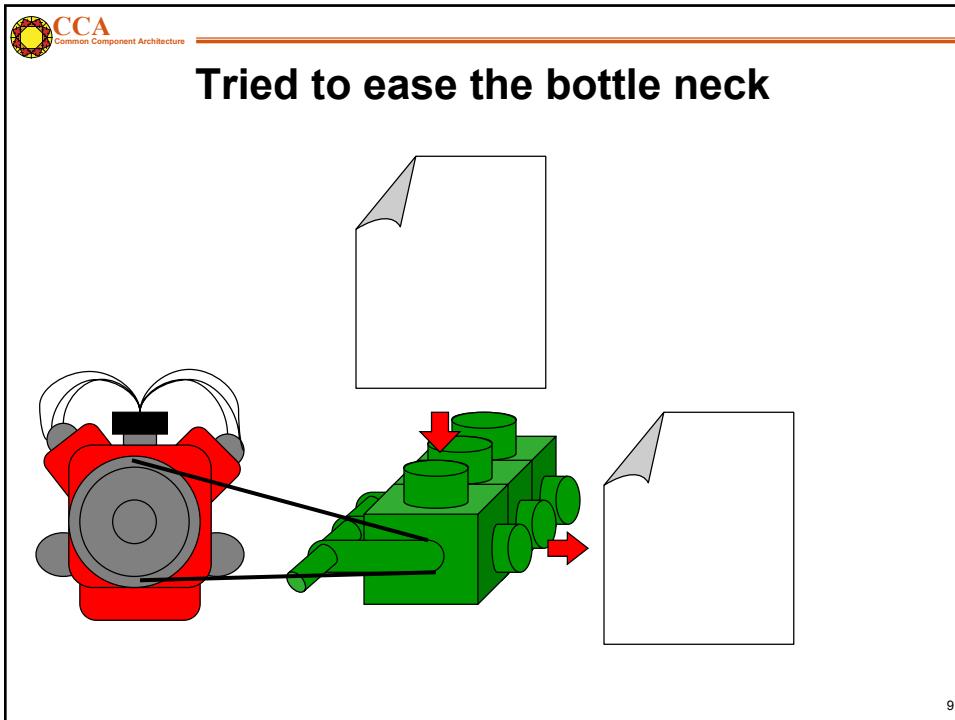
7

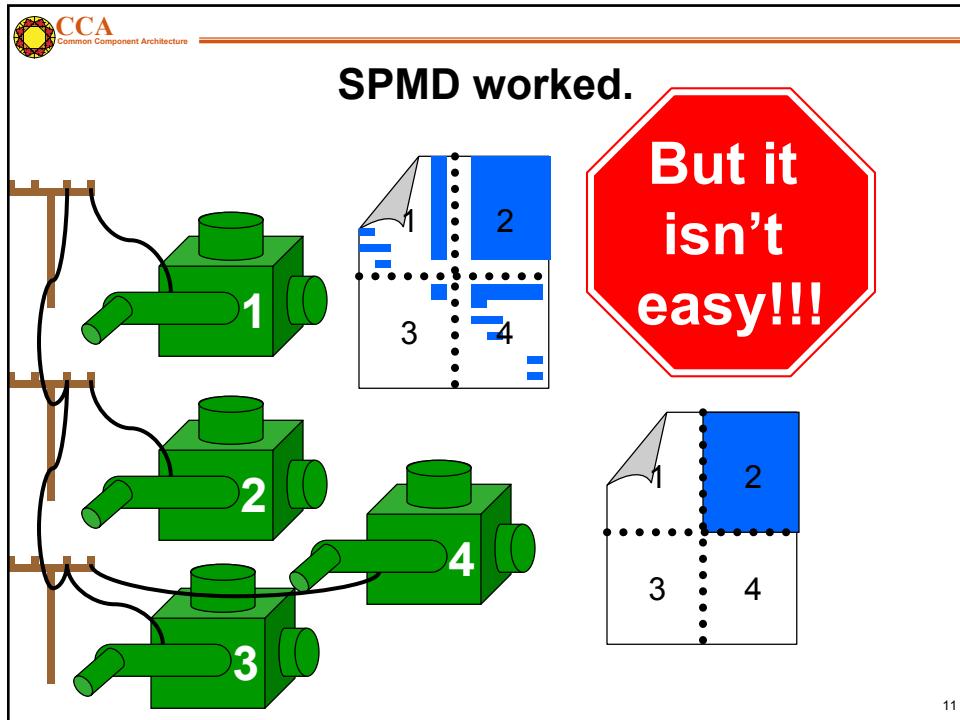
CCA  
Common Component Architecture

## As Scientific Computing grew...

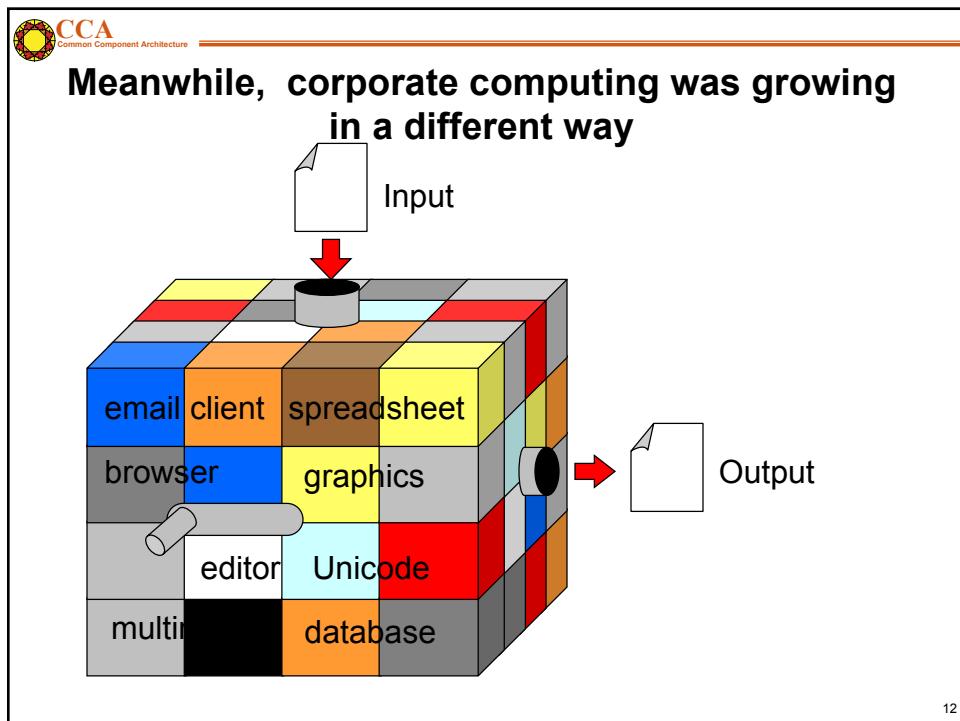
A diagram illustrating a bottleneck in the processing flow. On the left, a large white document icon represents a large input file. A blue arrow points towards a green 3D block labeled "Program". A red arrow points from the "Program" block to a smaller white document icon representing the output. The word "Bottle Neck" is written in large blue letters across the blue arrow.

8





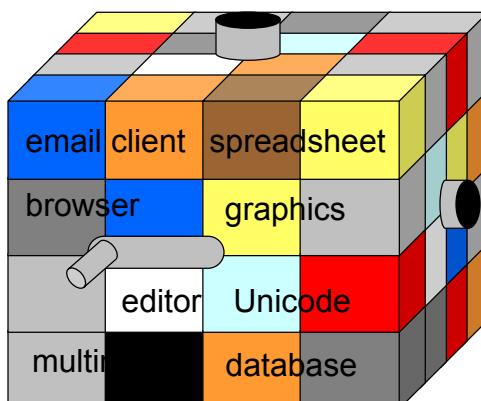
11



12

 CCA  
Common Component Architecture

**This created a whole new set of problems**  
→ complexity



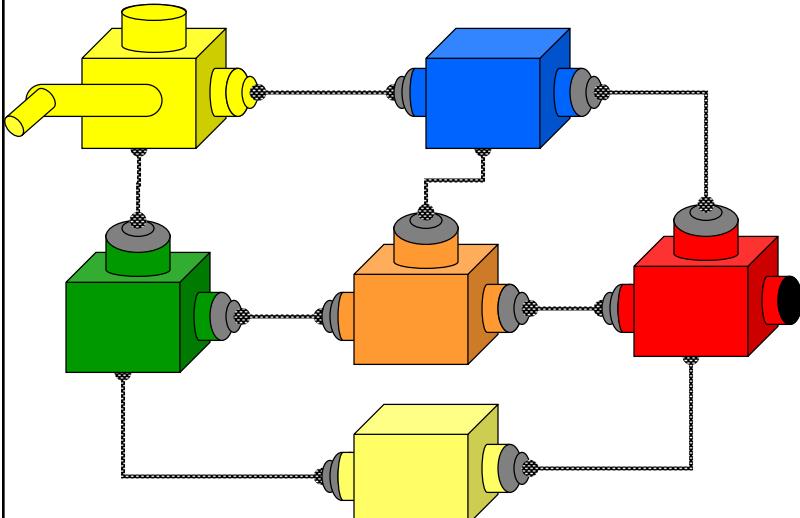
A 3D Rubik's cube is shown, divided into a grid of colored faces. The faces contain labels representing different software components: 'email client', 'spreadsheet', 'browser', 'graphics', 'editor', 'Unicode', 'multi...', and 'database'. The cube is oriented with its top face showing a mix of colors and labels.

- Interoperability across multiple languages
- Interoperability across multiple platforms
- Incremental evolution of large legacy systems (esp. w/ multiple 3rd party software)

13

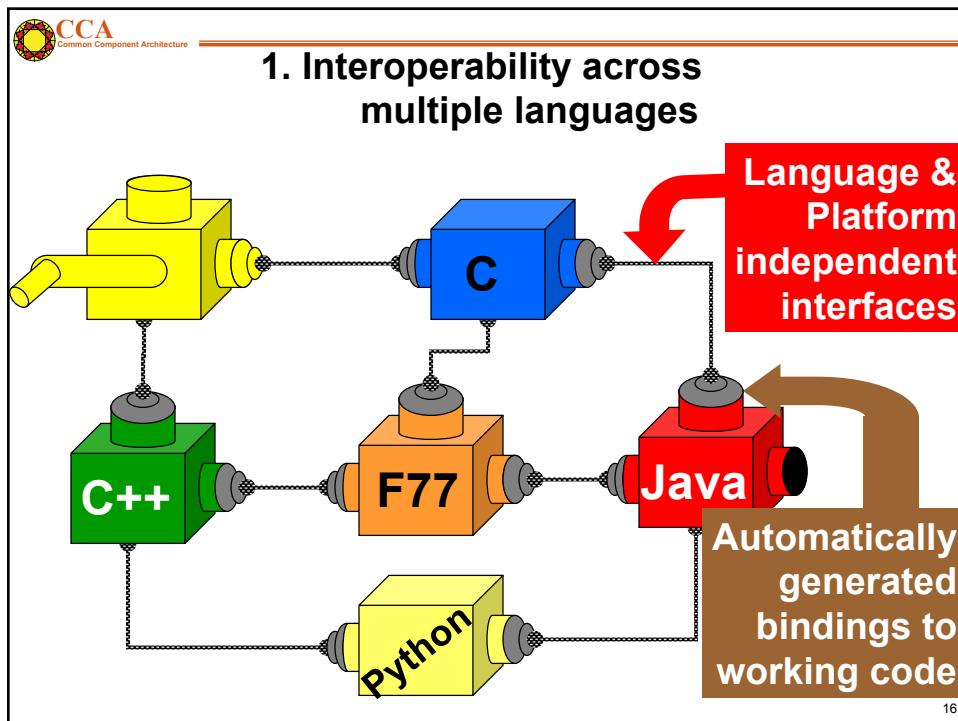
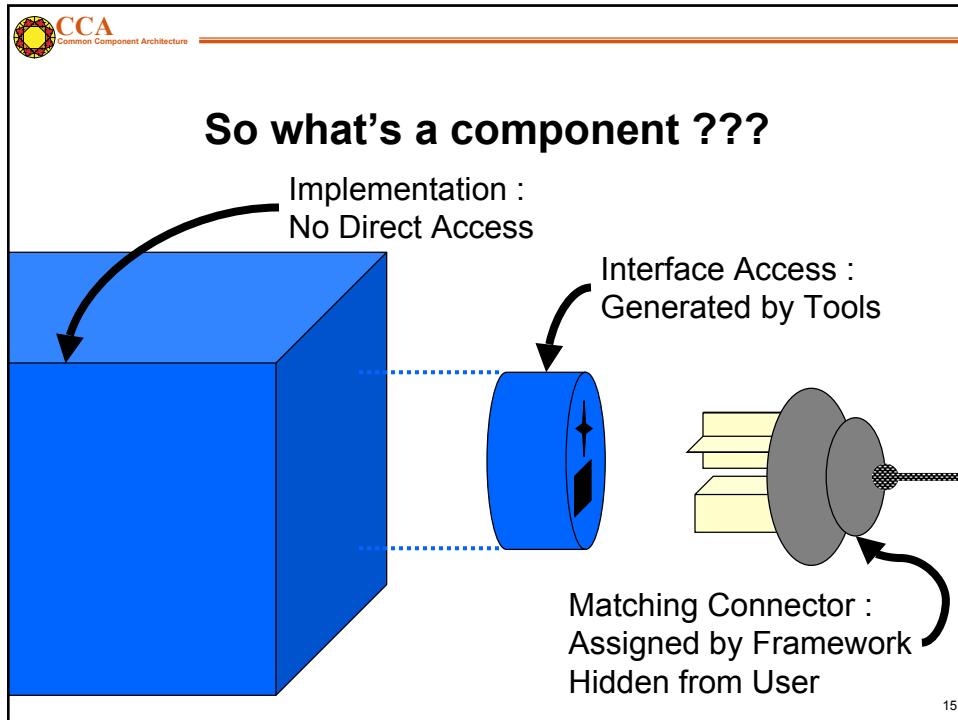
 CCA  
Common Component Architecture

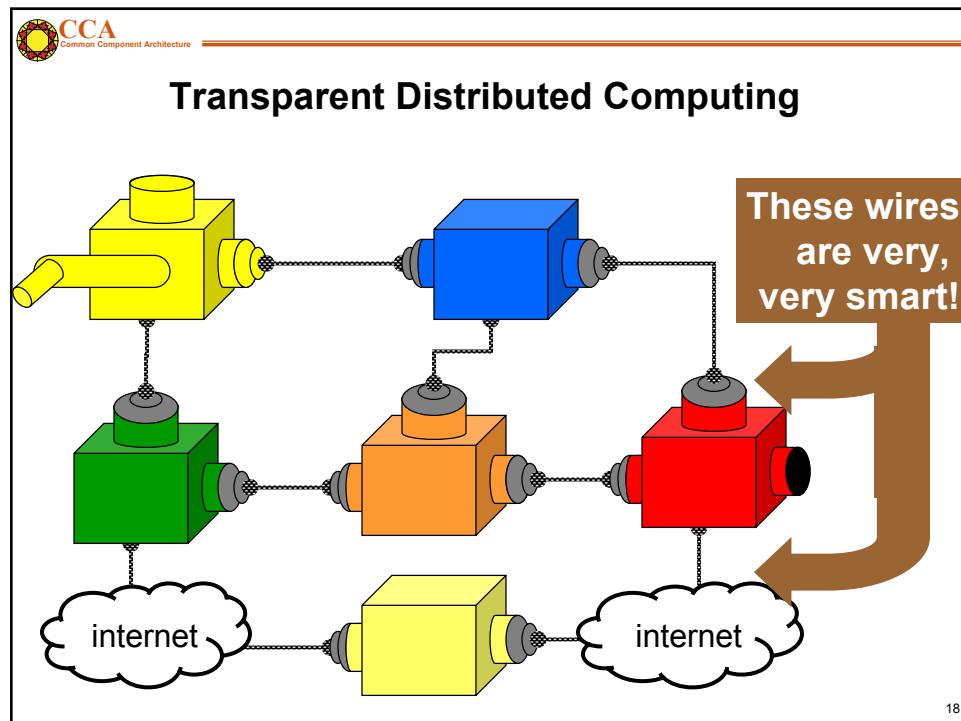
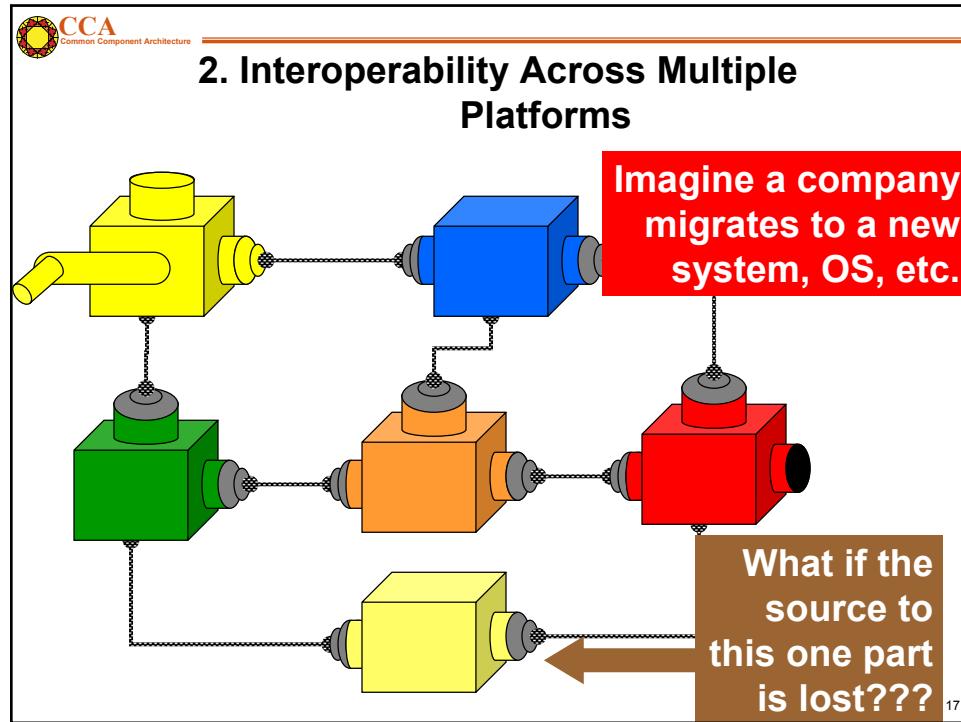
**Component Technology addresses these problems**

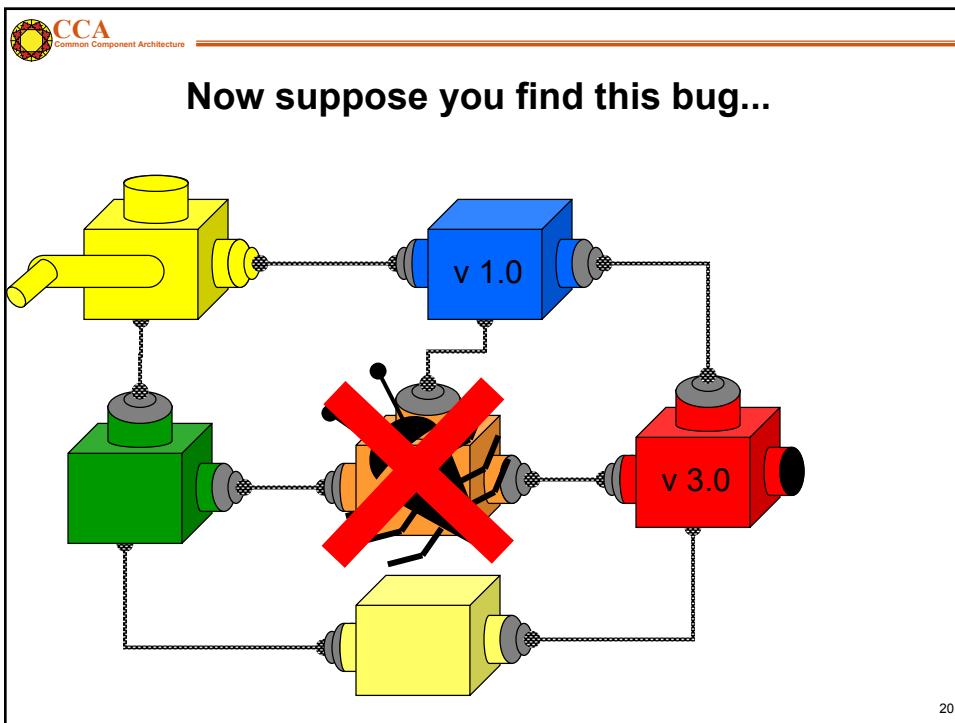
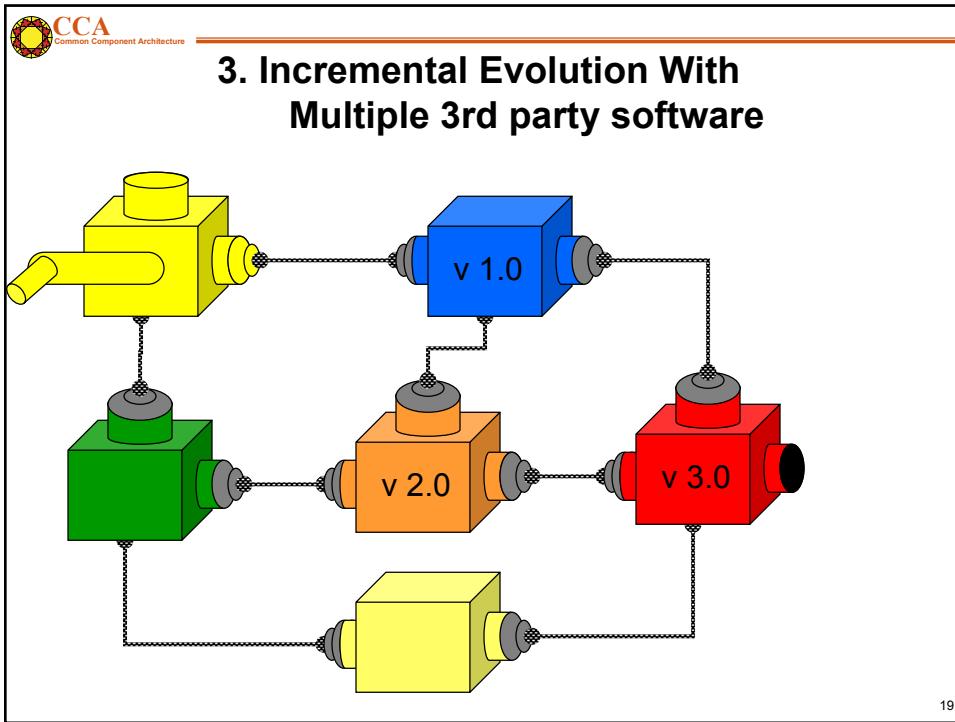


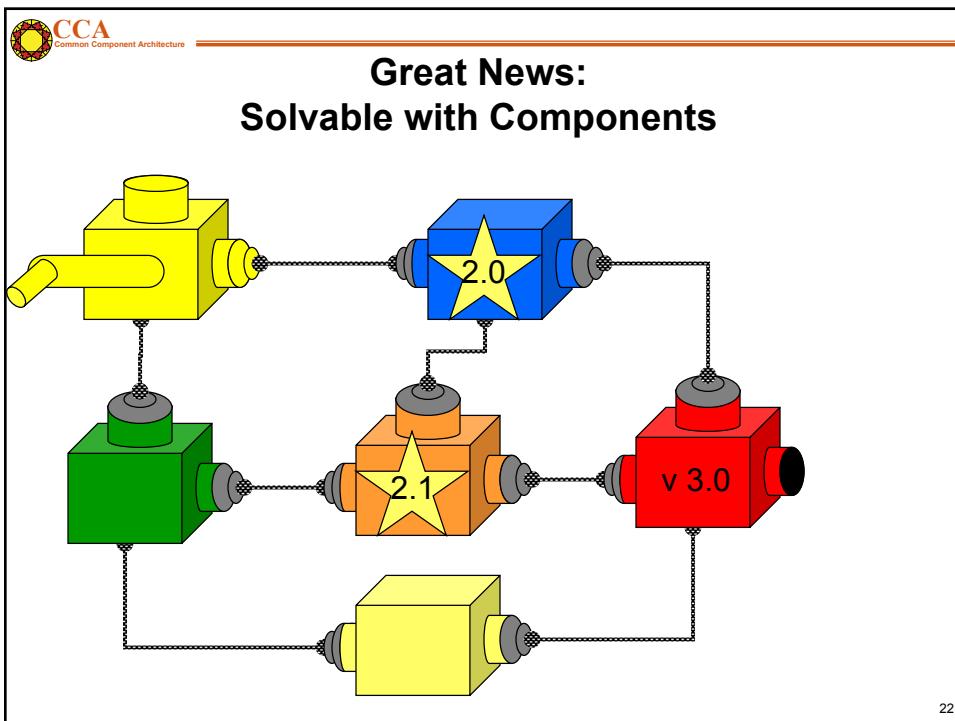
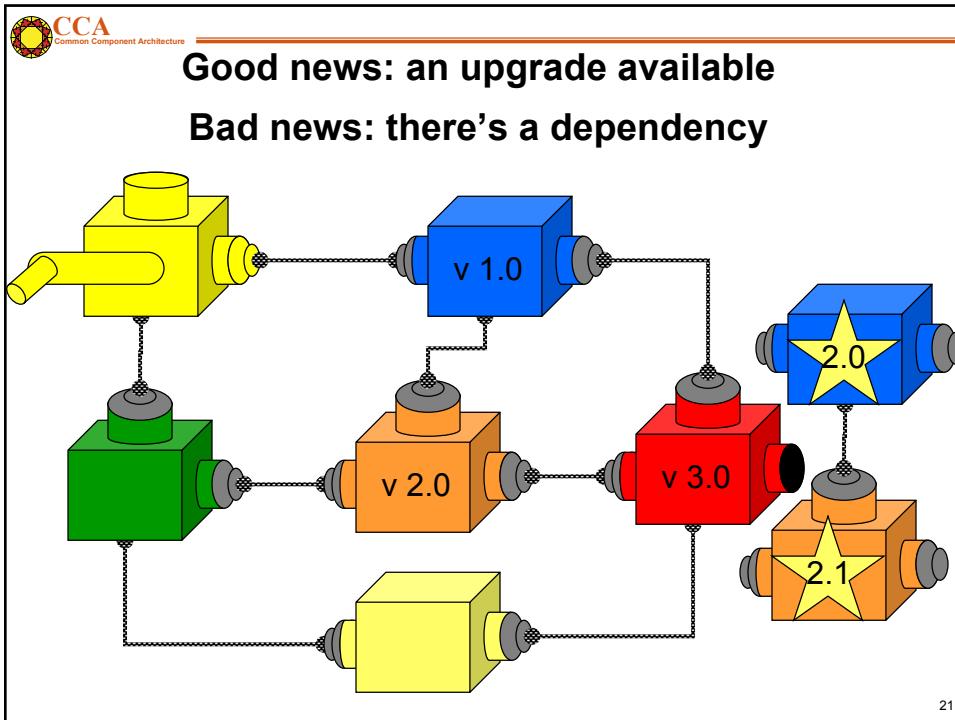
A diagram showing a network of interconnected components. Six colored cubes (yellow, blue, green, orange, red, and light yellow) are arranged in a loop, each with a circular port on one side. Dotted lines connect the ports between adjacent cubes, forming a closed loop. This visualizes how component technology provides a modular and interconnected architecture.

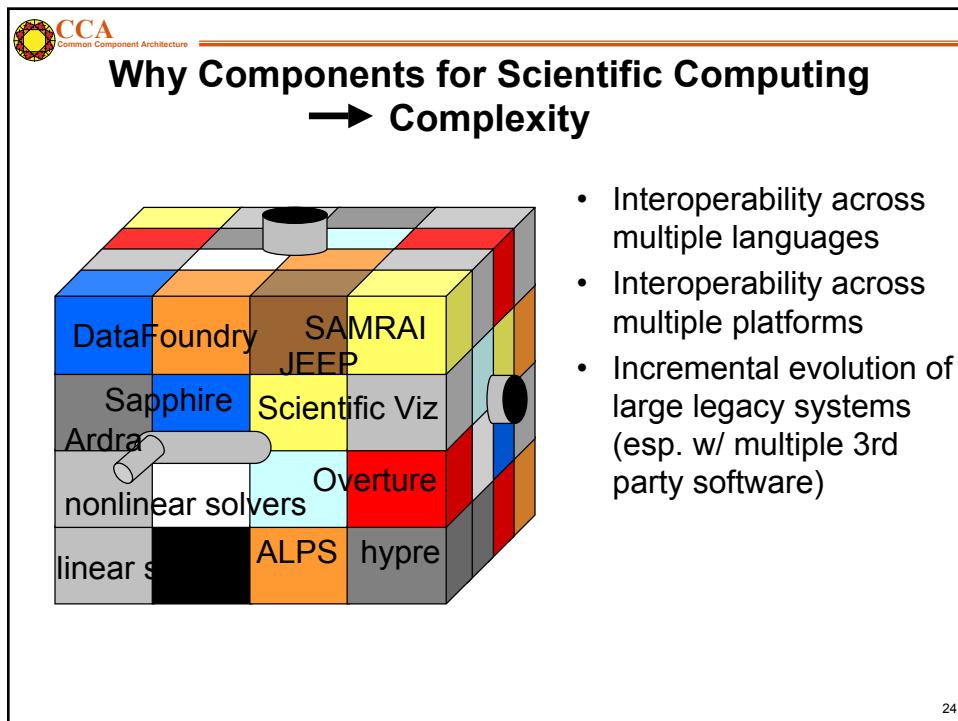
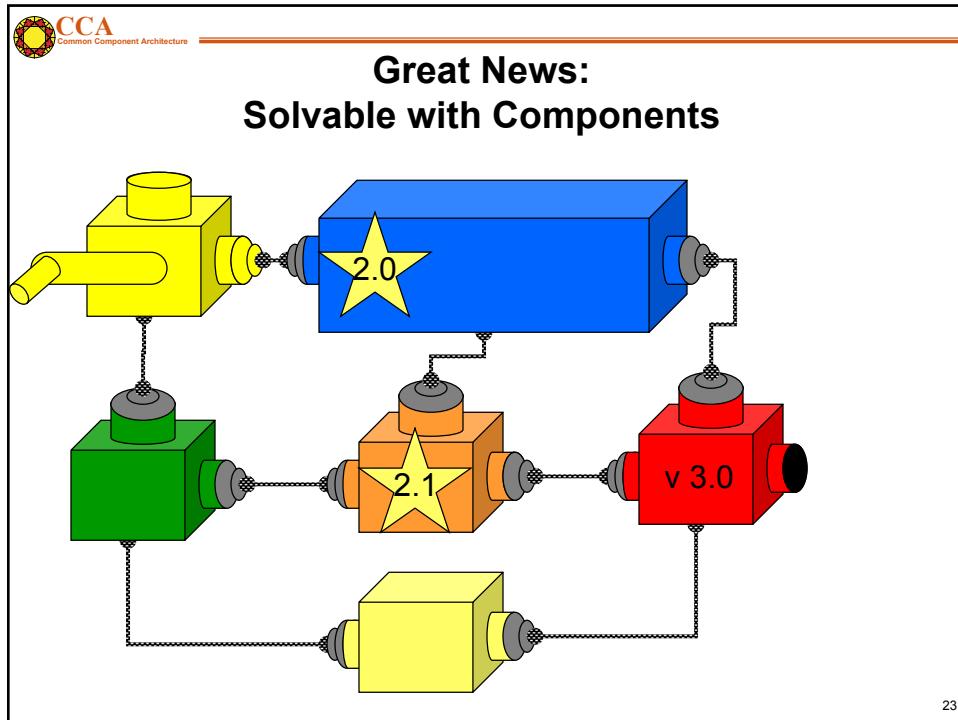
14

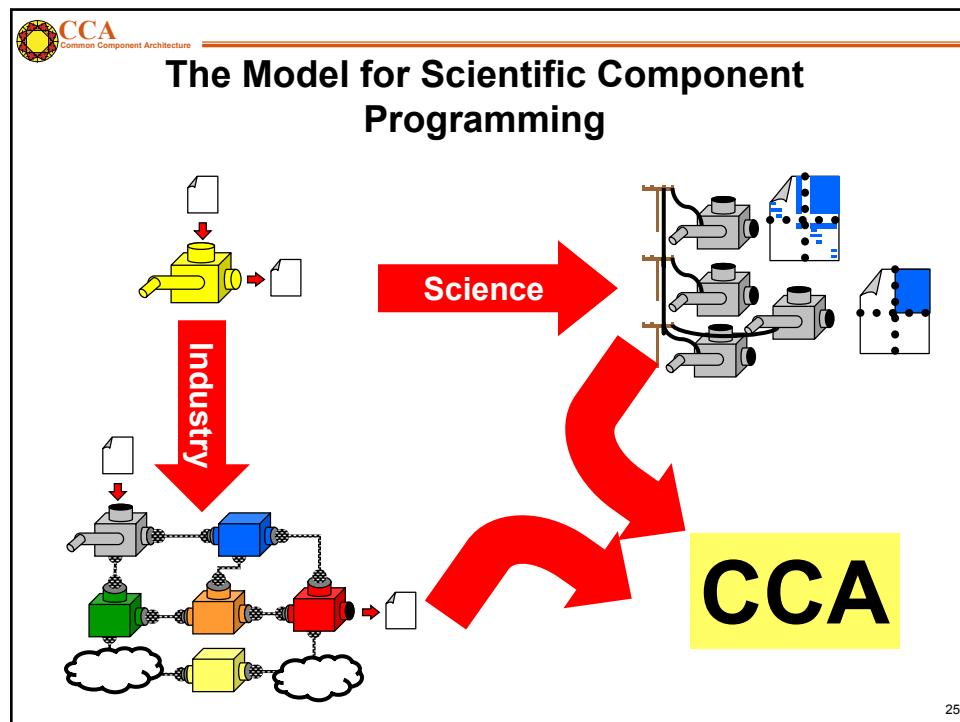












**CCA**  
Common Component Architecture

### An Introduction to Components and the Common Component Architecture

**CCA Forum Tutorial Working Group**  
<http://www.cca-forum.org/tutorials/>  
*tutorial-wg@cca-forum.org*

26



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

27



Common Component Architecture

## Component-Based Software Engineering

- CBSE methodology is an emerging approach to software development
  - Both in research and in practical application
  - Especially popular in business and internet areas
- Addresses software **complexity** issues
- Increases software **productivity**

28



## 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

29



## 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

30



## What are Components?

- No universally accepted definition in computer science research ...yet
- 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**

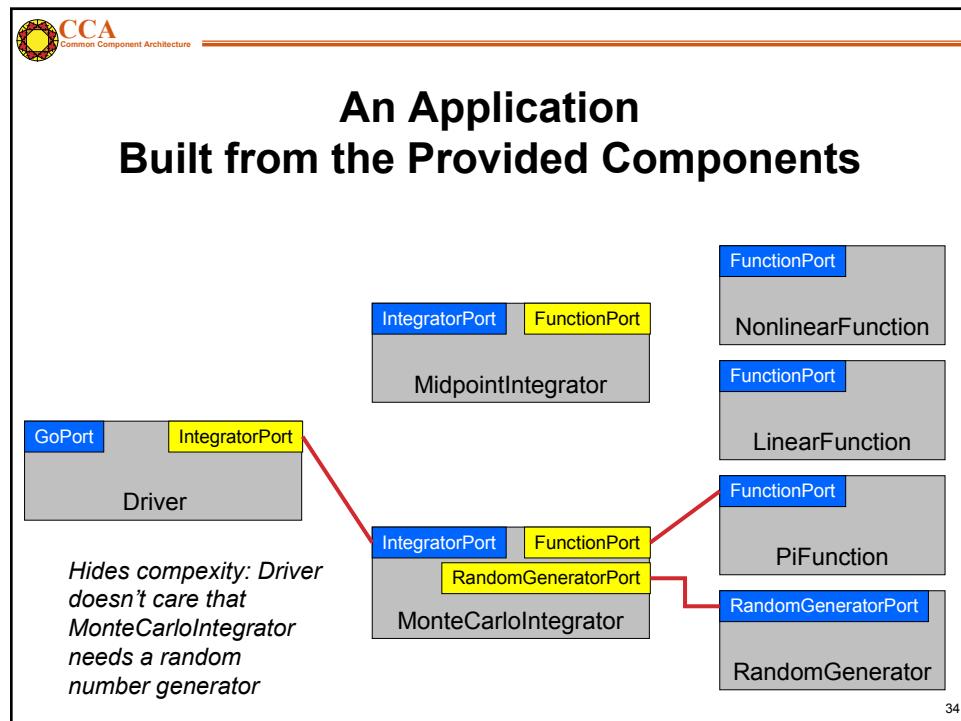
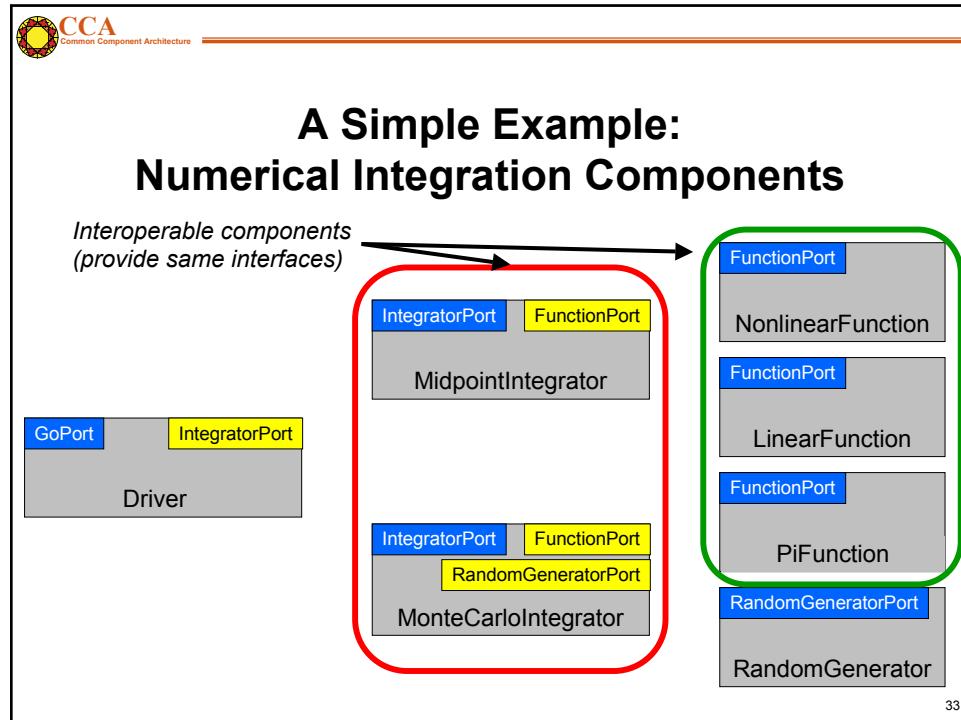
31

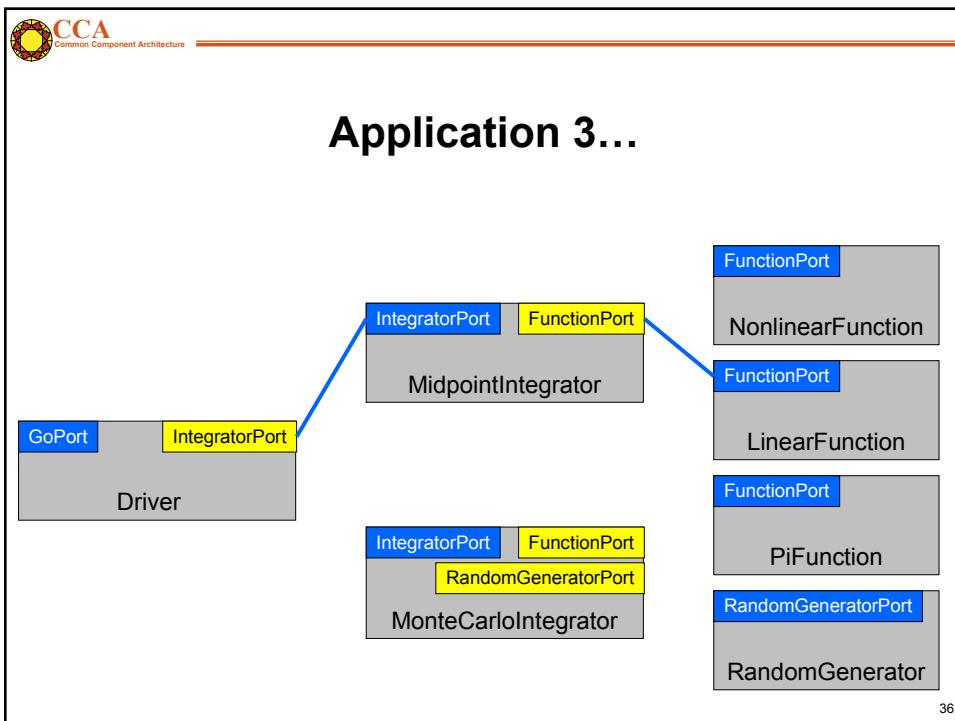
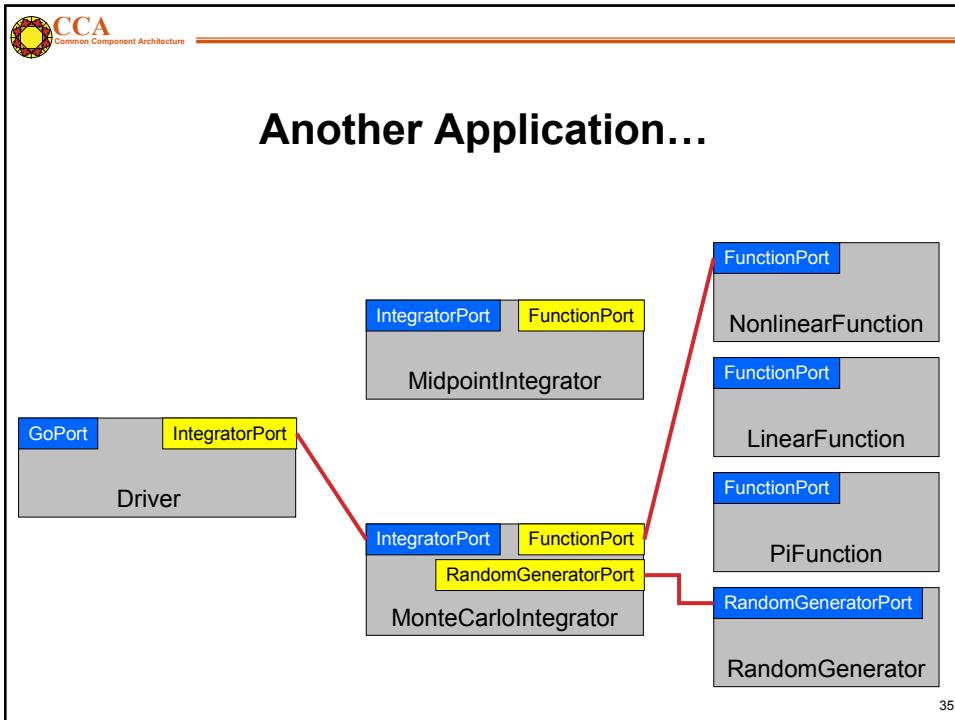


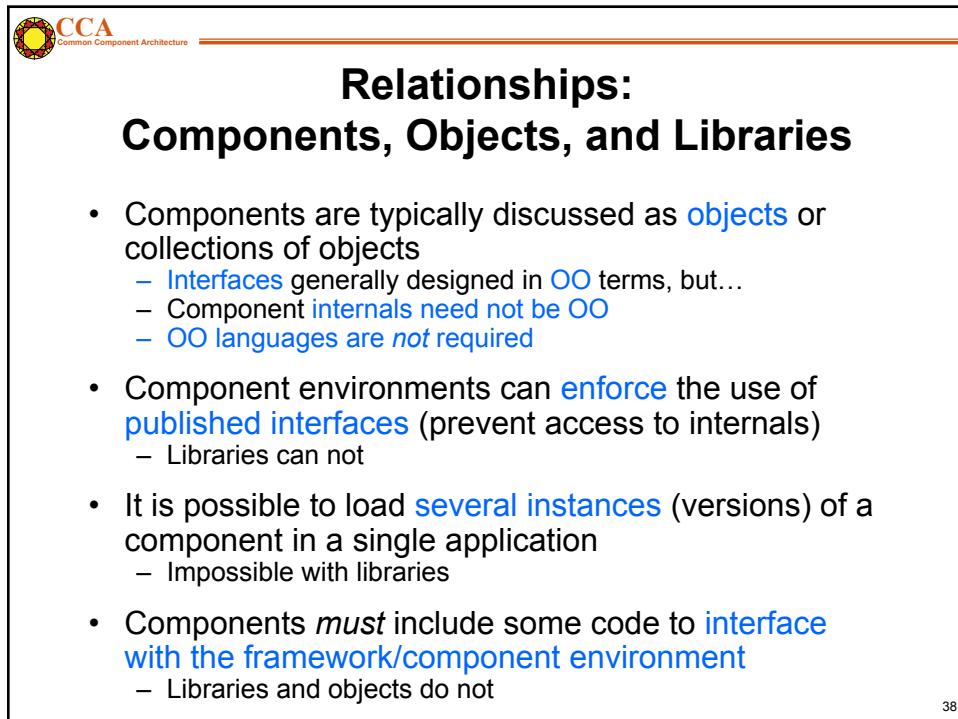
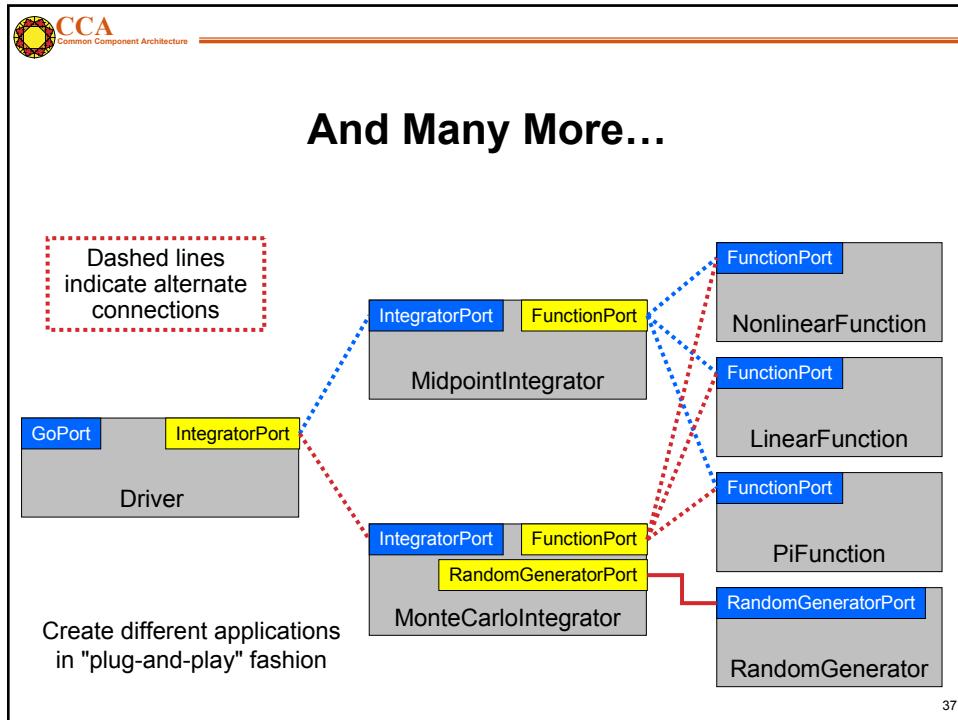
## What is a Component Architecture?

- 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

32









## Domain-Specific Frameworks vs Generic 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**
  - E.g. Cactus, ESMF, PRISM
  - Hypre, Overture, PETSc, POOMA

### Generic

- Provide the infrastructure to **hook components** together
  - Domain-specific infrastructure can be built as 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
  - e.g. CCA

39



## 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...
- Achieving reuse across multiple applications requires agreement on the same interface for all of them
- “**Standard**” or “**community**” interfaces facilitate reuse and interoperability
  - Typically domain specific
  - Formality of “standards” process varies
  - Significant initial investment for long-term payback

More about community interface development efforts in “Applications” module

40



## 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?

41



## Commodity Component Models

- CORBA Component Model (CCM), COM, Enterprise JavaBeans
  - Arise from business/internet software world
- Componentization **requirements** can be **high**
- Can impose significant **performance overheads**
- No recognition of **tightly-coupled parallelism**
- May be **platform specific**
- May have **language constraints**
- May not support common scientific **data types**

42



## What is the CCA?

- CCA is a *specification* of a component environment designed for **high performance scientific computing**
  - Specification is decided by the [CCA Forum](#)
    - CCA Forum membership open to all
  - “CCA-compliant” just means conforming to the specification
    - Doesn’t require using any of our code!
- 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](#)

43



## CCA Philosophy and Objectives

- **Local and remote components**
  - Support local, HPC parallel, and distributed computing
- **High Performance**
  - Design should support high-performance mechanisms wherever possible (i.e. minimize copies, extra communications, extra synchronization)
  - Support SPMD and MPMD parallelism
  - Allow user to chose parallel programming models
- **Heterogeneity**
  - Multiple architectures, languages, run-time systems used simultaneously in an application
- **Integration**
  - Components should be easy to make and easy to use
- **Openness and simplicity**
  - CCA spec should be open & usable with open software

44

 CCA  
Common Component Architecture

## CCA Concepts: Components

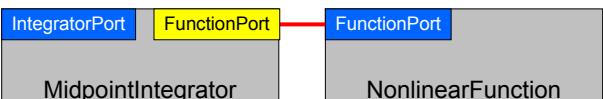


- Components provide/use one or more **ports**
  - A component with no ports isn't very interesting
- Components include some **code which interacts with a CCA framework**

45

 CCA  
Common Component Architecture

## CCA Concepts: Ports



- Components interact through well-defined **interfaces**, or **ports**
  - In OO languages, a port is a **class** or **interface**
  - In Fortran, a port is a bunch of subroutines or a **module**
- Components may **provide** ports – **implement** the class or subroutines of the port (**"Provides" Port**)
- Components may **use** ports – **call** methods or subroutines in the port (**"Uses" Port**)
- Links between ports denote a procedural (caller/callee) relationship, **not dataflow!**
  - e.g., FunctionPort could contain: *evaluate(in Arg, out Result)*

46



## 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
- *Currently:* specific frameworks support specific computing models (parallel, distributed, etc.)
- *Future:* full flexibility through integration or interoperation

47



## Writing Components

- Components...
  - Inherit from **gov.cca.Component**
    - Implement **setServices** method to register ports this component will **provide** and **use**
  - Implement the ports they provide
  - Use ports on other components
    - **getPort/releasePort** from framework **Services** object
- Interfaces (ports) extend **gov.cca.Port**

**Full details in the hands-on!**

48



## Adapting Existing Code into Components

**Example in the hands-on!**

Suitably structured code (programs, libraries) should be relatively easy to adapt to the CCA. Here's how:

1. Decide **level of componentization**
  - Can evolve with time (start with coarse components, later refine into smaller ones)
2. Define **interfaces** and write wrappers between them and existing code
3. Add **framework interaction code** for each component
  - `setServices`
4. Modify component internals to **use other components** as appropriate
  - `getPort`, `releasePort` and method invocations

49



## Writing Frameworks

- ***There is no reason for most people to write frameworks – just use the existing ones!***
- Frameworks must provide certain ports...
  - **ConnectionEventService**
    - Informs the component of connections
  - **AbstractFramework**
    - Allows the component to *behave as a framework*
  - **BuilderService**
    - Instantiate components & connect ports
  - **ComponentRepository**
    - A default place where components are found
- Frameworks must be able to load components
  - Typically shared object libraries, can be statically linked
- Frameworks must provide a way to compose applications from components

50

**CCA** Common Component Architecture

## 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

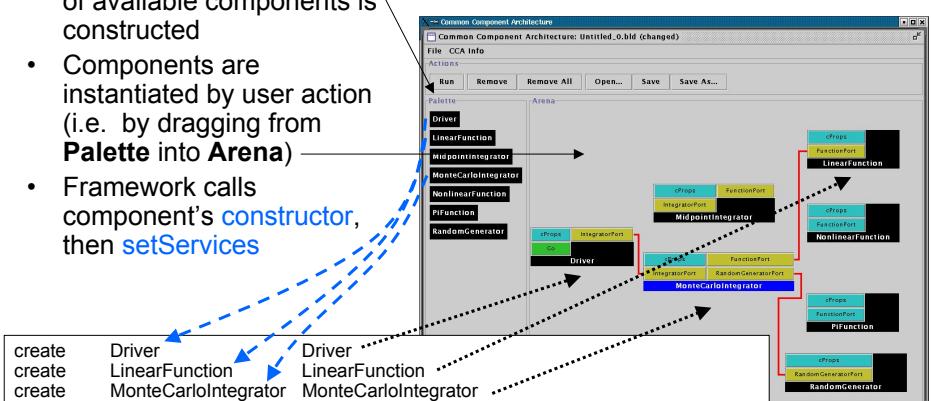
51

**CCA** Common Component Architecture

## 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



create Driver  
 create LinearFunction  
 create MonteCarloIntegrator

create Driver  
 create LinearFunction  
 create MonteCarloIntegrator

create Driver  
 create LinearFunction  
 create MonteCarloIntegrator

52

**CCA** Common Component Architecture

## 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

connect	Driver	IntegratorPort	MonteCarloIntegrator	IntegratorPort	FunctionPort
connect	MonteCarloIntegrator	FunctionPort	LinearFunction	FunctionPort	
...					

53

**CCA** Common Component Architecture

## Component's View of Instantiation

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

Framework interaction code  
**constructor** **setServices** **destructor**

**CCA.Services**  
 provides IntegratorPort  
 uses FunctionPort,  
 RandomGeneratorPort

Integrator code

MonteCarloIntegrator

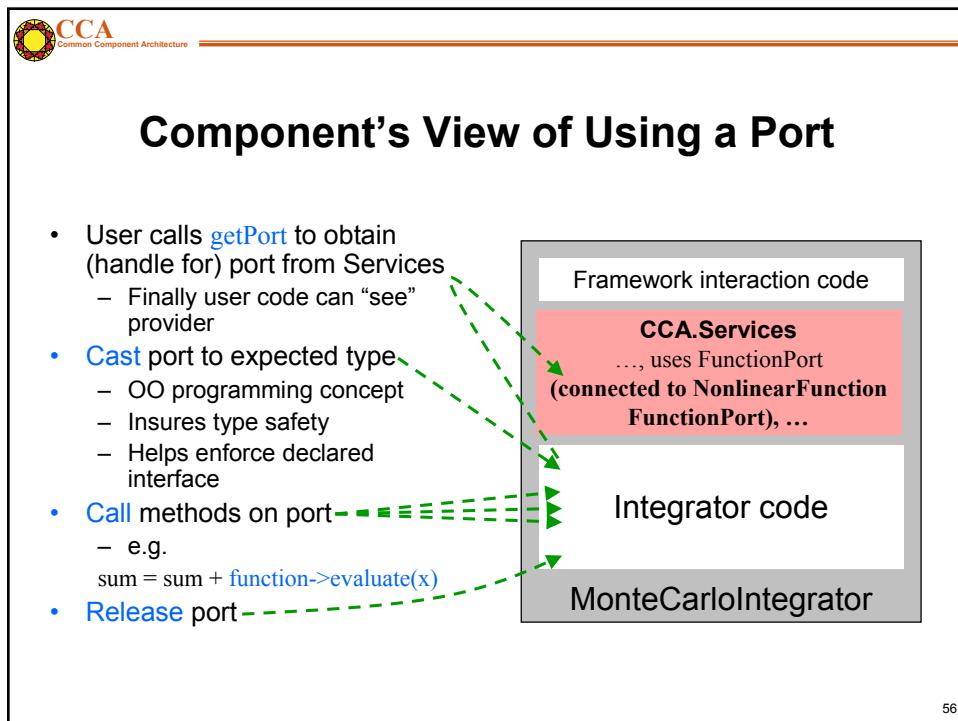
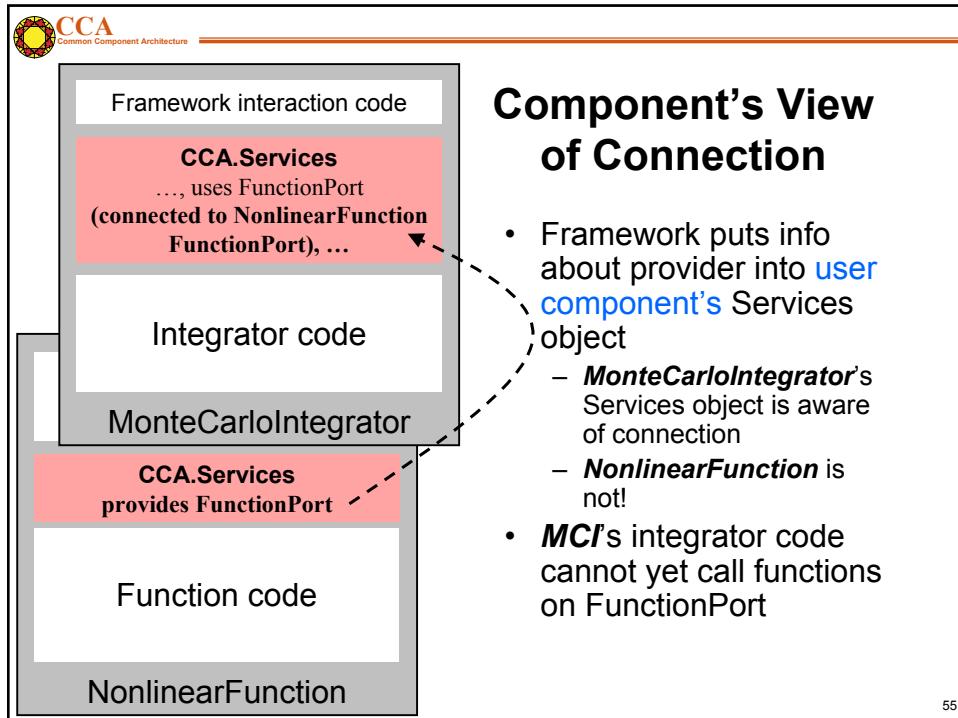
- Framework calls component's **setServices**
  - Passes setServices an object representing everything "outside"
  - setServices declares ports component *uses* and *provides*
- Component *still* knows nothing outside itself
  - But Services object provides the means of communication w/ framework
- Framework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

IntegratorPort      FunctionPort

RandomGeneratorPort

MonteCarloIntegrator

54



**CCA Supports Local, Parallel and Distributed Computing**

- “**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 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

57

**CCA Concepts: “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

More about performance in the “Applications” module

58

**CCA Concepts: 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

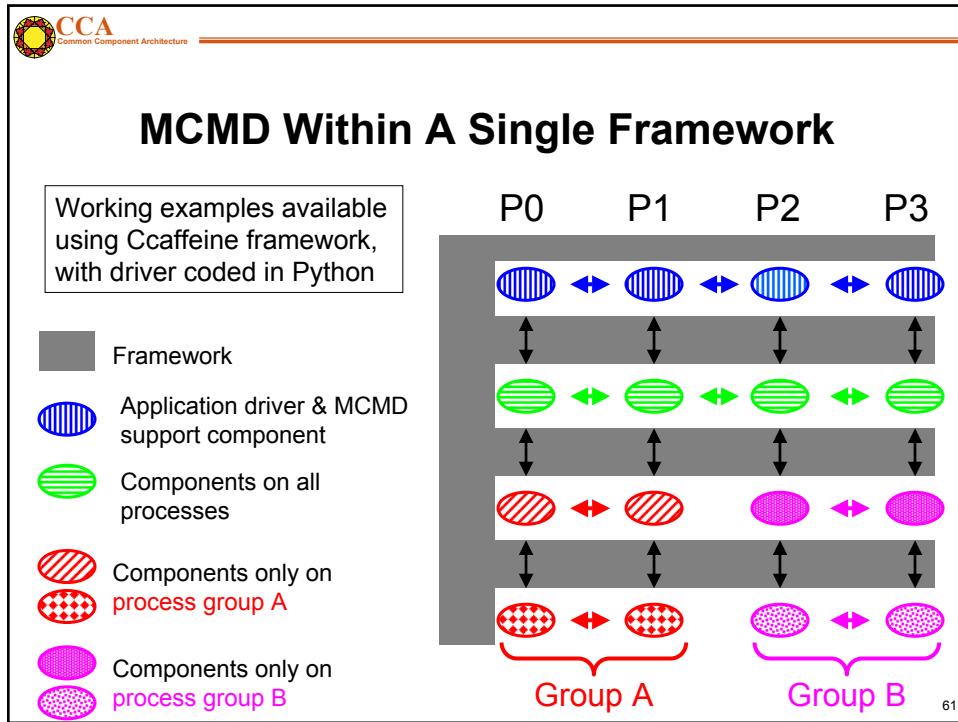
59

**“Multiple-Component Multiple-Data” Applications in CCA**

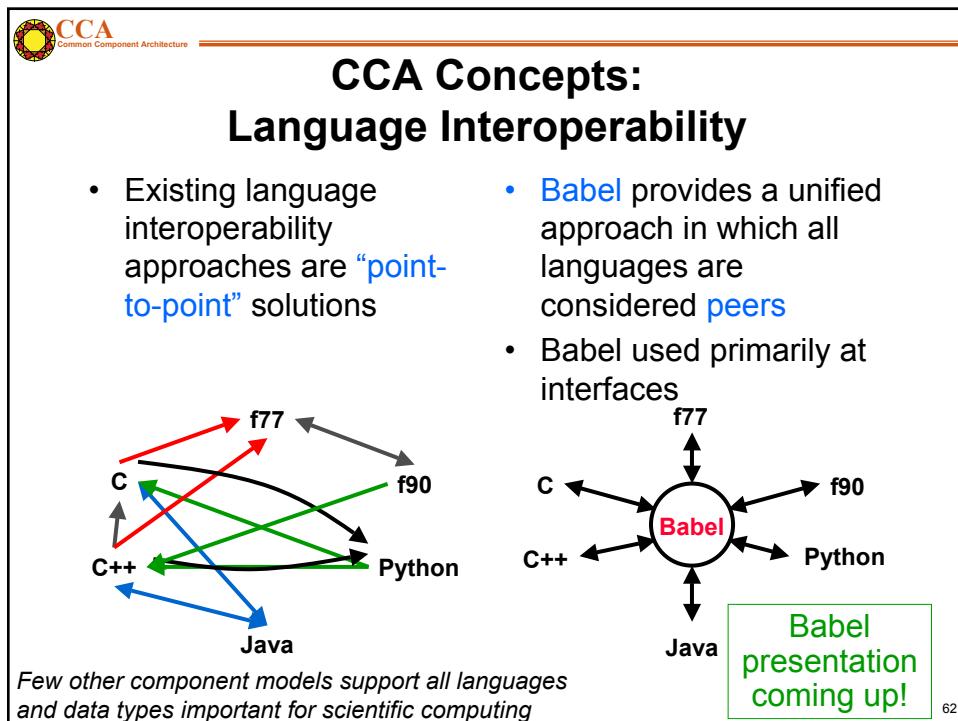
- Simulation composed of multiple SCMD sub-tasks
- Usage Scenarios:
  - Model coupling (e.g. Atmosphere/Ocean)
  - General multi-physics applications
  - Software licensing issues
- Approaches
  - Run single parallel framework
    - Driver component that partitions processes and builds rest of application as appropriate (through BuilderService)
  - Run multiple parallel frameworks
    - Link through specialized communications components (e.g. MxN)
    - Link as components (through AbstractFramework service; highly experimental at present)

Driver		
Atmosphere	Ocean	Land
Coupler (MxN)		

60



61



62



## Advanced CCA Concepts

- Components are **peers**
  - Application architecture determines relationships, not CCA specification
- Frameworks provide a **BuilderService** which allows **programmatic composition** of components
- Frameworks may **present themselves as components** to other frameworks
- A “traditional” application can treat a CCA framework as a **library**
- **Meta-component models** enable bridging between CCA components and other component(-like) environments
  - e.g. SCIRun Dataflow, Visualization Toolkit (VTK), ...

No time to go into detail on these, but ask us for more info after the tutorial

63



## 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
  - Generic programming

64



## What the CCA is...

- CCA is a *specification* for a component environment
  - Fundamentally, a design pattern
  - Multiple “reference” implementations exist
  - Being used by applications
- CCA is designed for interoperability
  - Components within a CCA environment
  - CCA environment with other tools, libraries, and frameworks
- CCA provides an environment in which domain-specific application frameworks can be built
  - While retaining opportunities for software reuse at multiple levels

65



## Concept Review

- **Ports**
  - Interfaces between components
  - Uses/provides model
- **Framework**
  - Allows assembly of components into applications
- **Direct Connection**
  - Maintain performance of local inter-component calls
- **Parallelism**
  - Framework stays out of the way of parallel components
- **Language Interoperability**
  - Babel, Scientific Interface Definition Language ([SIDL](#))

66

 **CCA**  
Common Component Architecture

---

## Distributed Computing with the CCA

**CCA Forum Tutorial Working Group**  
<http://www.cca-forum.org/tutorials/>  
[tutorial-wg@cca-forum.org](mailto:tutorial-wg@cca-forum.org)

67

 **CCA**  
Common Component Architecture

---

## Component Composition

- Components can be linked along shared interfaces (ports) where one component invokes the services of another
  - Two types of **Ports**
    - Provides Ports – implements a remote interface
    - Uses Ports – uses a remote interface
  - A user and a provider of the same type can be linked
  - Details of run-time substrate shielded in **stubs and skeletons**
    - Similar in concept to the files generated by Babel

**Uses port** -  
a call site for  
a remote function  
invocation

**Provides Port** -  
A set of functions  
which can be  
invoked remotely

68



## How Distributed Frameworks are Different

### Remote Creation

- Launch components in remote address spaces
- Heterogeneity management
- Use resource managers to service requests on each remote resource
- Store, move and replicate component binaries

### Remote Invocation

- Need **global pointers** and not local pointers
- Invoke methods across machine boundaries
- Need **global namespace** for names of components and services
- Mechanism for implementing **remote method invocation (RMI)**
- Introspection mechanisms to allow ports and services to be discovered and accessed

69

## CCA Concepts that Influence Design of Distributed Frameworks (1)

- Ports
  - References to *provides* ports can move across address spaces
  - *Uses* ports are local to each component
- Services Object is present in each component
  - Manages all the ports
  - Hides details of framework-specific bindings for ports
- ComponentID: opaque handle to the component
  - Should be **serializable** and **deserializable**
  - Usually points to the **services** object

70



## CCA Concepts that Influence Design of Distributed Frameworks (2)

- Builder Service: charged with following operations
  - Create Components in remote address spaces
    - Return a **ComponentID** of instantiated components
    - Hide details of heterogeneous remote environments
  - Connect ports of components
    - Facilitate connection between uses and provides ports
      - Only if they are of the same SIDL type
    - Place provides port reference in the uses port table
- Introspection
  - Allow remote querying of a component
    - How many and what type of ports does the component have?

71



## Key Design Choices for Distributed CCA Frameworks (1)

- How is the CCA **ComponentID** represented in a distributed environment?
  - Handle that can be passed to remote components
  - Serialize and deserialize ComponentID
  - Belong to a namespace understood in the entire framework
  - Should enable optimized communication for co-located components
- How is the **PortType** represented?
  - *Provides* port should be designed as a remote service
  - *Uses* port should be a local object

72



## Key Design Choices for Distributed CCA Frameworks (2)

- Where can the key CCA functions be called from?  
Are they remote or local?
  - getPort() call on the services object
    - Should return a remote reference for provides ports
    - Note that the same call in the [Ccaffeine](#) framework returns a local object
  - Details of remote and local calls should be hidden
    - Framework should internally distinguish local and remote calls
- How can components be connected?
  - Need internal mechanism for uses port to obtain remote reference of the provides port
    - Information can be stored in a central table, facilitate development of GUIs to show component assembly
    - Distributed across components so they are aware of who they are connected to

73



## Key Design Choices for Distributed CCA Frameworks (3)

- Should Builder Service be centralized or distributed?
  - A component can have its own builder service if
    - The builder service is lightweight
    - The components has special create/connect requirements

74



## Current CCA Distributed Frameworks

- SCIRun2
  - University of Utah
- LegionCCA
  - Binghamton University - State University of New York (SUNY)
- XCAT (Java and C++)
  - Indiana University and Binghamton University
- DCA
  - Indiana University
  - A research framework for MXN
- Frameworks address the design questions in different ways
  - Each has a different set of capabilities
  - Specialized for different kinds of applications

75



## SCIRun2

- Remote Method Invocation (RMI)
  - Allows distributed components to interact through normal mechanisms
  - Components in the same address space incur no additional overhead
  - Based on a C++ in-house SIDL compiler
    - Currently not based on Babel
- Remote creation of distributed components
  - A distributed CCA framework uses RMI to coordinate components
  - A [slave framework](#) resides on each remote address space
  - Uses [ssh](#) to start the slave framework
  - CCA BuilderService communicates with [master framework](#) which coordinates [slave frameworks](#)

76



Common Component Architecture

## SCIRun2

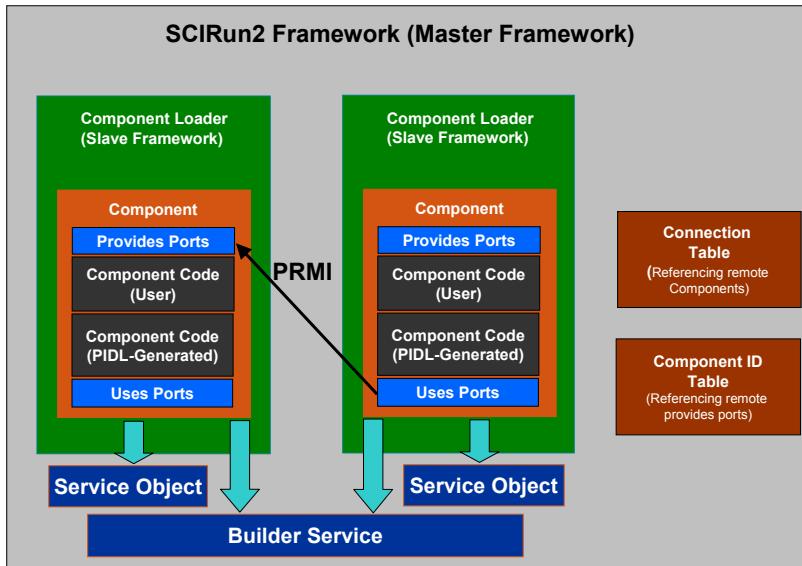
- Support for distributed and parallel components
  - Able to launch MPI-parallel components
    - Parallel components interact with other parallel components (on different machines) through [Parallel Remote Method Invocation \(PRMI\)](#)
  - Each MPI process may contain multiple threads
    - Increases concurrency and efficiency in the face of a large parallel invocation load

77



Common Component Architecture

## Architecture of Distributed SCIRun2

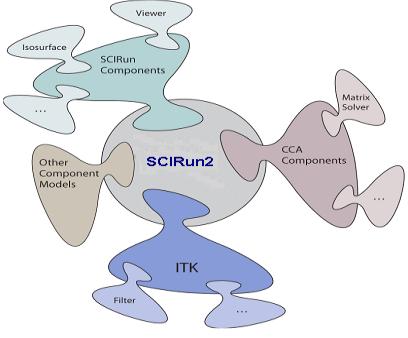


78

 CCA  
Common Component Architecture

## SCIRun2 Meta-Component Model

- In the same way that components plug into a CCA framework, component models (such as CCA) plug into SCIRun2
- Allows components of several different families to be used together
- Currently supports: CCA (Babel), SCIRun Dataflow, Visualization Toolkit (Vtk); others coming...
- Bridging between components of different models is semi-automatic; current research is defining a more automatic form of bridging



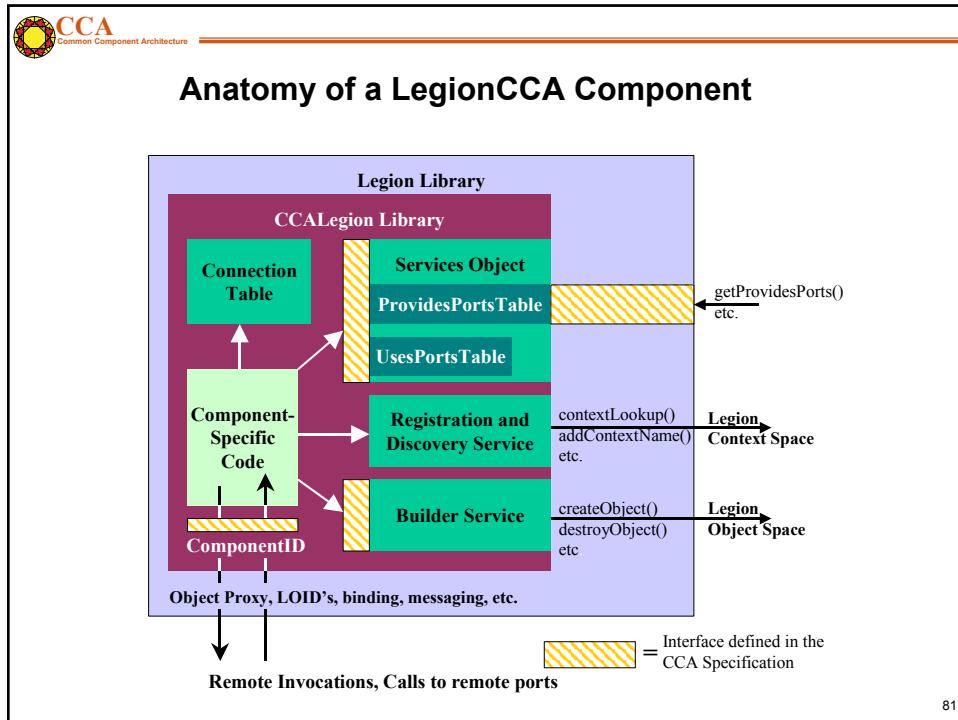
79

 CCA  
Common Component Architecture

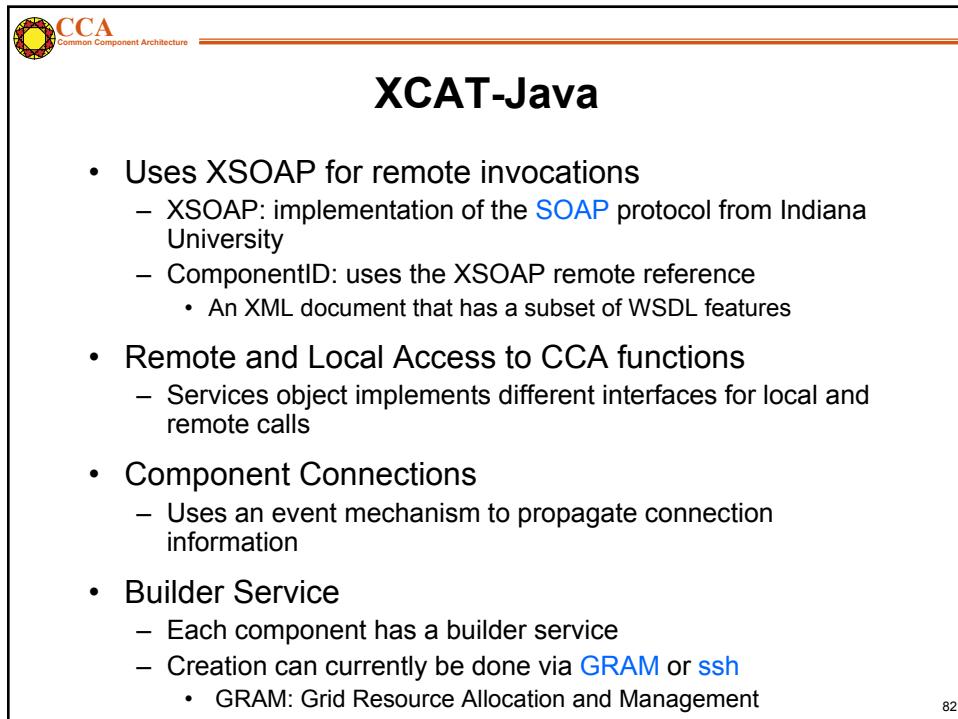
## LegionCCA

- Legion is a collection of software services for the Grid
  - Provides illusion of a virtual machine for geographically-distributed resources
- LegionCCA: models CCA components as Legion objects
- Component Communication
  - Uses Legion's built-in RPC mechanisms, based on Unix sockets
- ComponentID: based on Legion LOID
  - LOID: globally unique object id
- Component Connections:
  - Information distributed across components
  - Tables can be dynamically updated as connections are made and broken

80



81



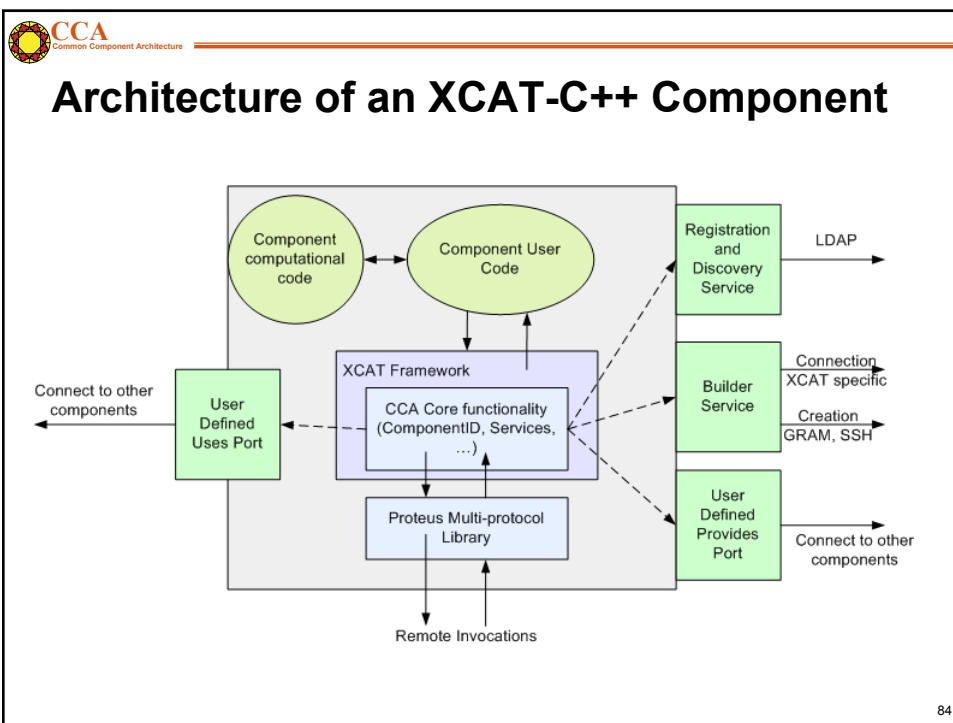
82

**CCA** Common Component Architecture

## XCAT-C++

- Remote Method Invocation
  - Uses the [Proteus](#) multi-protocol library for remote communication
    - Proteus supports both messaging and RMI models
    - Currently supports two protocols: binary and SOAP
  - Stub-Skeleton generation is based on WSDLPull
    - A toolkit for parsing WSDL (Web Service Description Language)
  - Support for SIDL will be provided via BabelRMI
    - BabelRMI: Currently in the research phase
- Remote creation of distributed components
  - Each component has a BuilderService
  - Creation can be based on GRAM or ssh

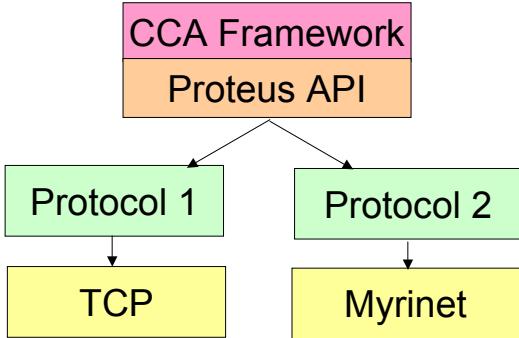
83



 CCA  
Common Component Architecture

## Proteus: Multi-Protocol Library

- One protocol does not suit all applications
- Proteus provides single-protocol abstraction to components
  - Allows users to dynamically switch between protocols
    - Example: RMI A and RMI B, in the picture
  - Facilitates use of specialized implementations of [serialization](#) and [deserialization](#)



```
graph TD; Framework["CCA Framework<br/>Proteus API"] --> Protocol1["Protocol 1<br/>TCP"]; Framework --> Protocol2["Protocol 2<br/>Myrinet"]
```

85

 CCA  
Common Component Architecture

## Babel RMI

[Research!](#)

- Allows Babel objects to be accessed through remote Babel stubs.
- Underlying RMI uses Proteus.
- Objects that can be transmitted (serializable) inherent from Serializable.
- Actual implementation of serialization functions is by users, since only they know what needs to be serialized.

86



**CCA**

Common Component Architecture

## CCA Applications

### CCA Forum Tutorial Working Group

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

*tutorial-wg@cca-forum.org*

87

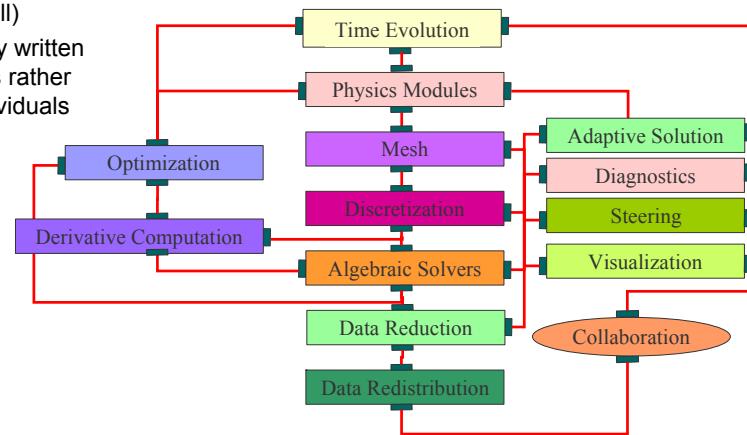


**CCA**

Common Component Architecture

## 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



88

 CCA  
Common Component Architecture

## Overview

- • Examples (scientific) of increasing complexity
  - Laplace equation
  - Time-dependent heat equation
  - Nonlinear reaction-diffusion system
  - Quantum chemistry
  - Climate simulation
- Tools
  - MxN parallel data redistribution
  - Performance measurement, modeling and scalability studies
- Community efforts & interface development
  - TSTT Mesh Interface effort
  - CCTTSS's Data Object Interface effort

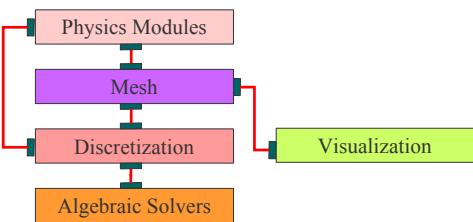
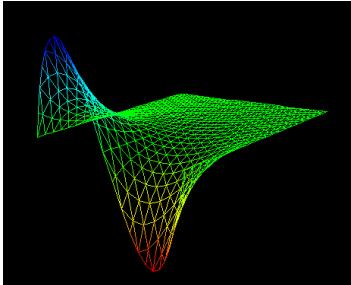
89

 CCA  
Common Component Architecture

## Laplace Equation

$$\nabla^2 \varphi(x,y) = 0 \in [0,1] \times [0,1]$$

$$\varphi(0,y)=0 \quad \varphi(1,y)=\sin(2\pi y)$$

$$\delta\varphi/\delta y(x,0) = \delta\varphi/\delta y(x,1) = 0$$



90

**Laplace Equation with Components**

- The Driver Component
  - Responsible for the overall application flow
  - Initializes the mesh, discretization, solver and visualization components
  - Sets the physics parameters and boundary condition information

91

**Laplace Equation with Components**

- The Driver
- The Mesh Component
  - Provides geometry, topology, and boundary information
  - Provides the ability to attach user defined data as tags to mesh entities
  - Is used by the driver, discretization and visualization components

92

**CCA** Common Component Architecture

## Laplace Equation with Components

- The Driver
- The Mesh
- The Discretization Component
  - Provides a finite element discretization of basic operators (gradient, Laplacian, scalar terms)
  - Driver determines which terms are included and their coefficients
  - BC, Assembly etc

Common Component Architecture: steady.blr

File CCA Info Actions Run Remove Remove All Open... Save Save As...

Palette

Arena

driver

mesh

disc

solver

vizfile

93

**CCA** Common Component Architecture

## Laplace Equation with Components

- The Driver
- The Mesh
- The Discretization
  - The Solver Component
    - Provides access to vector and matrix operations (e.g., create, destroy, get, set)
    - Provides a “solve” functionality for a linear operator

Common Component Architecture: steady.blr

File CCA Info Actions Run Remove Remove All Open... Save Save As...

Palette

Arena

driver

mesh

disc

solver

vizfile

94

The screenshot shows the CCA interface with a tree view on the left and a component graph on the right.

**Tree View (Left):**

- The Driver
  - The Mesh
  - The Discretization
  - The Solver
    - The Visualization Component
      - Uses the mesh component to print a vtk file of  $\varphi$  on the unstructured triangular mesh
      - Assumes user data is attached to mesh vertex entities

**Component Graph (Right):**

The graph shows components and their connections:

- driver**: Contains cProps, go, Mesh, Disc, Solver, CONFIG, Params, and Vizfile.
- mesh**: Contains cProps and Mesh.
- disc**: Contains cProps, Mesh, Solver, CONFIG, Params, and Vizfile.
- solver**: Contains cProps and Mesh.
- vizfile**: Contains cProps and Vizfile.

Connections include: driver → mesh, driver → disc, driver → solver, driver → vizfile, mesh → disc, mesh → solver, disc → solver, and disc → vizfile.

95

The screenshot shows the CCA interface with a tree view on the left and a component graph on the right.

**Tree View (Left):**

- Time Evolution
- Physics Modules
- Mesh
- Discretization
- Algebraic Solvers
- Distributed Arrays
- Data Redistribution

**Component Graph (Right):**

The graph shows components and their connections:

- Time Evolution**, **Physics Modules**, **Mesh**, **Discretization**, **Algebraic Solvers**, **Distributed Arrays**, and **Data Redistribution** are connected in a vertical chain.
- Visualization** is connected to **Data Redistribution**.
- A 3D surface plot of a heat distribution is shown on the right.

96



## Some things change...

- Requires a time integration component
  - Based on the LSODE library
- Uses a new visualization component
  - Based on AVS
  - Requires an MxN data redistribution component
- The MxN redistribution component requires a Distributed Array Descriptor component
  - Similar to HPF arrays
- The driver component changes to accommodate the new physics

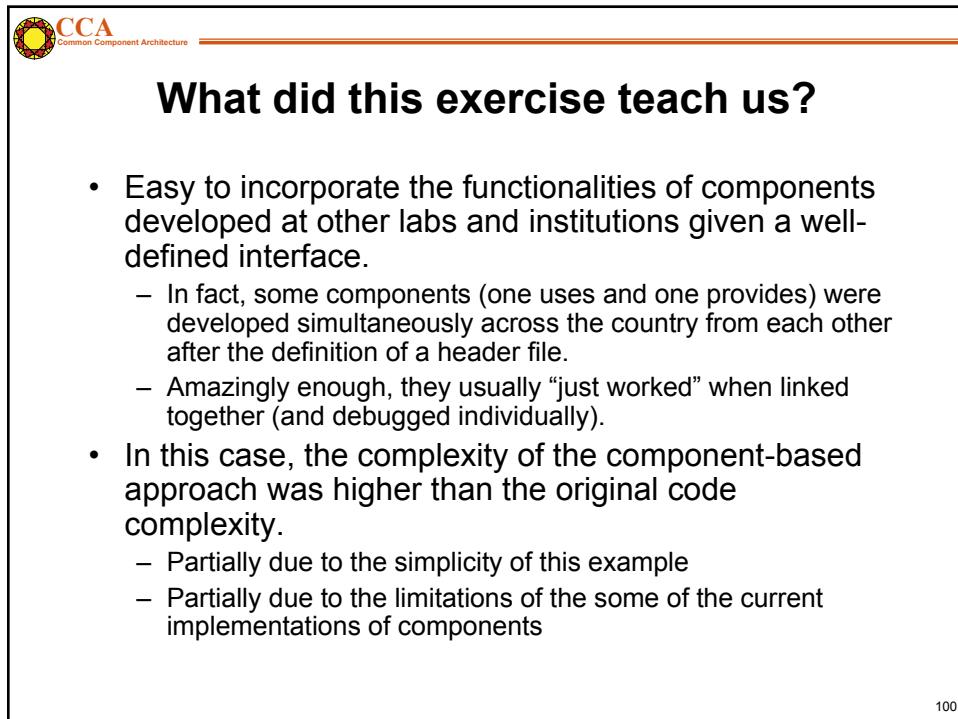
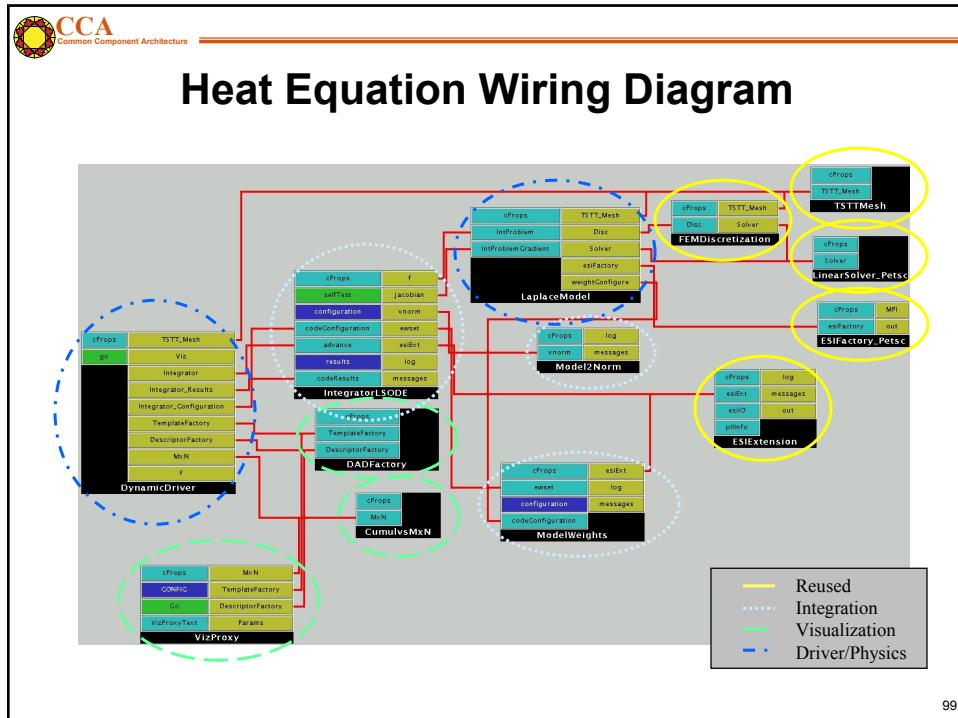
97



## ... and some things stay the same

- The mesh component doesn't change
- The discretization component doesn't change
- The solver component doesn't change
  - What we use from the solver component changes
  - Only vectors are needed

98



**CCA** Common Component Architecture

## Nonlinear Reaction-Diffusion Equation

**Temperature (K)**

- Flame Approximation
  - H<sub>2</sub>-Air mixture; ignition via 3 hot-spots
  - 9-species, 19 reactions, stiff chemistry
- Governing equation

$$\frac{\partial Y_i}{\partial t} = \nabla \cdot \alpha \nabla Y_i + \dot{w}_i$$

- Domain
  - 1cm X 1cm domain
  - 100x100 coarse mesh
  - finest mesh = 12.5 micron.
- Timescales
  - O(10ns) to O(10 microseconds)

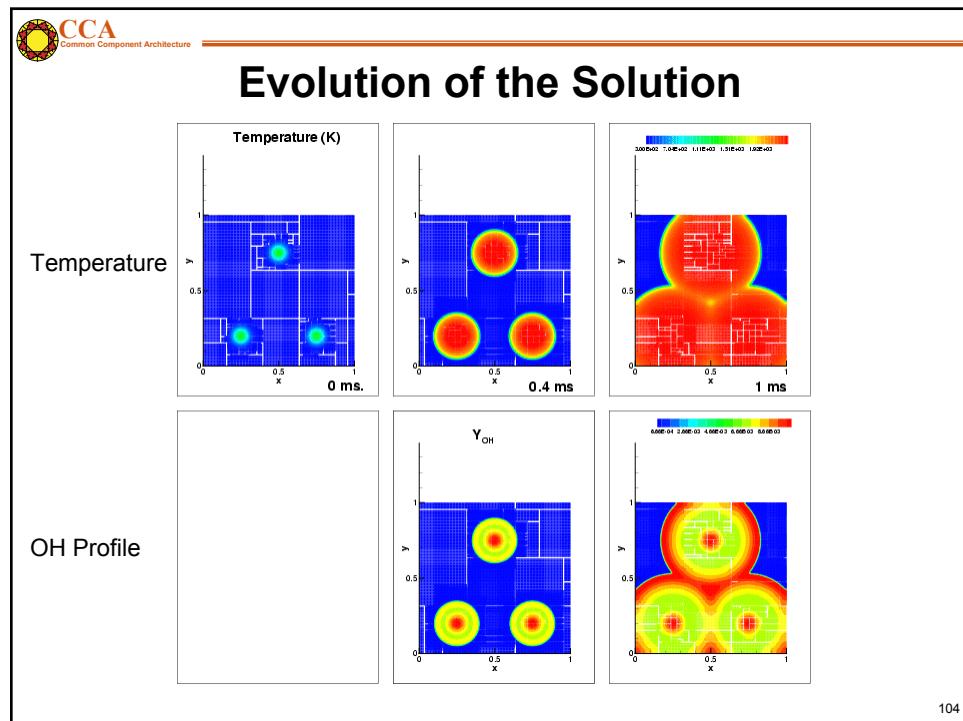
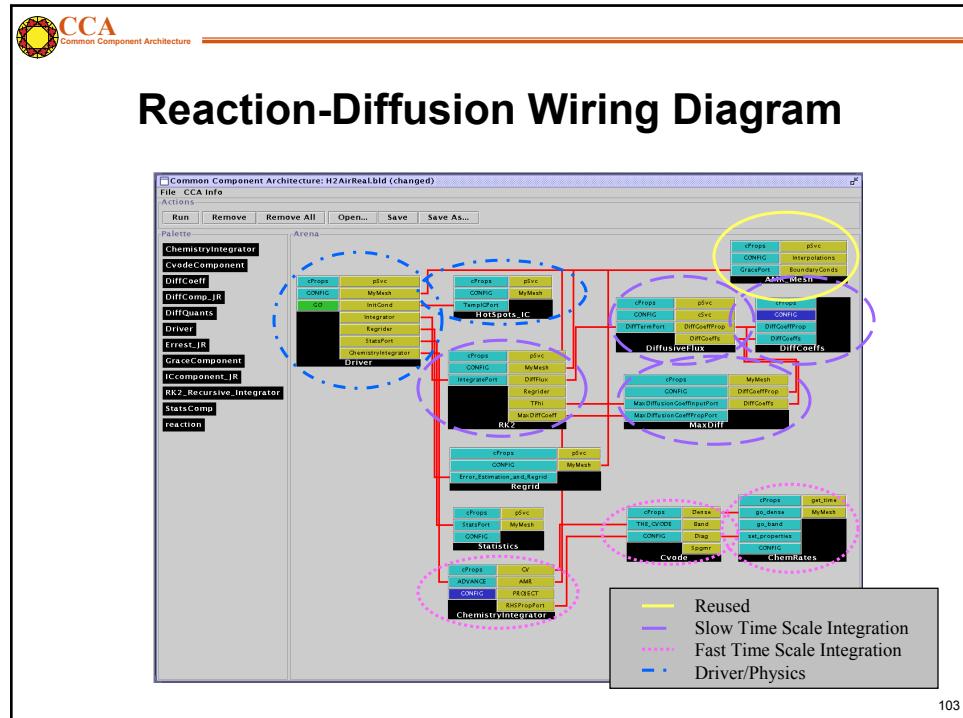
0 ms. 101

**CCA** Common Component Architecture

## Numerical Solution

- Adaptive Mesh Refinement: GrACE
- Stiff integrator: CVODE
- Diffusive integrator: 2<sup>nd</sup> Order Runge Kutta
- Chemical Rates: legacy f77 code
- Diffusion Coefficients: legacy f77 code
- New code less than 10%

102



**CCA**  
Common Component Architecture

## The need for AMR

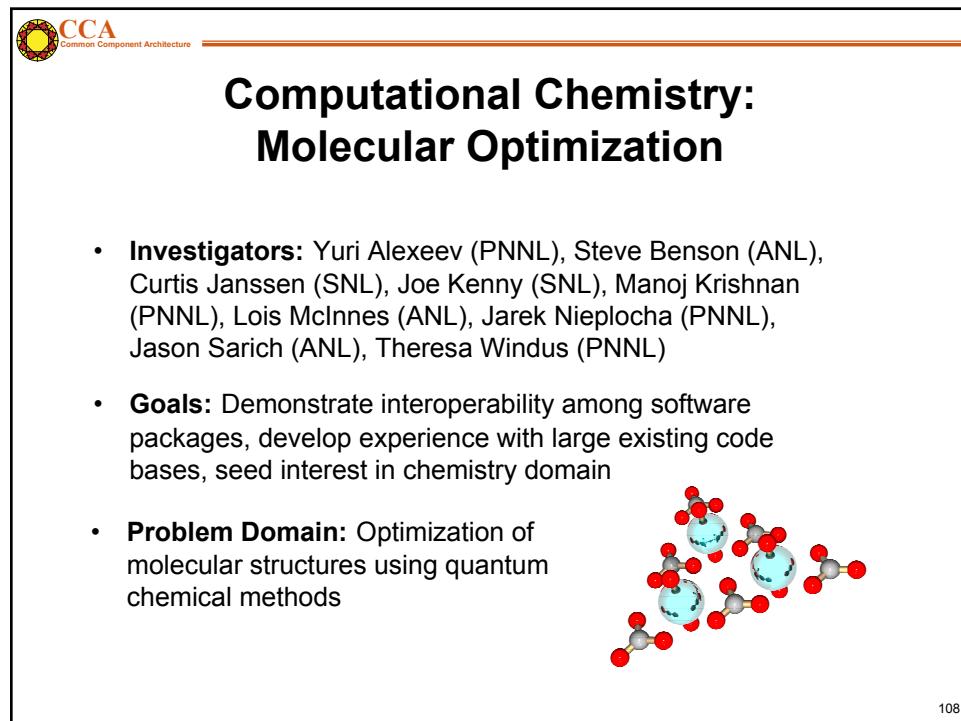
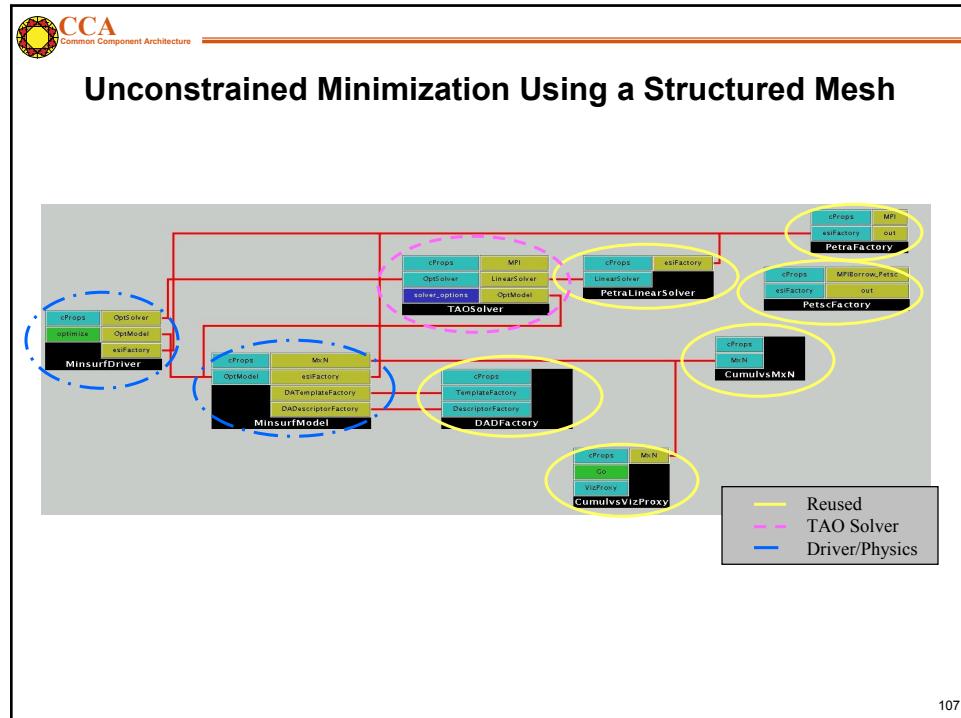
- $\text{H}_2\text{O}_2$  chemical subspecies profile
  - Only 100 microns thick (about 10 fine level cells)
  - Not resolvable on coarsest mesh

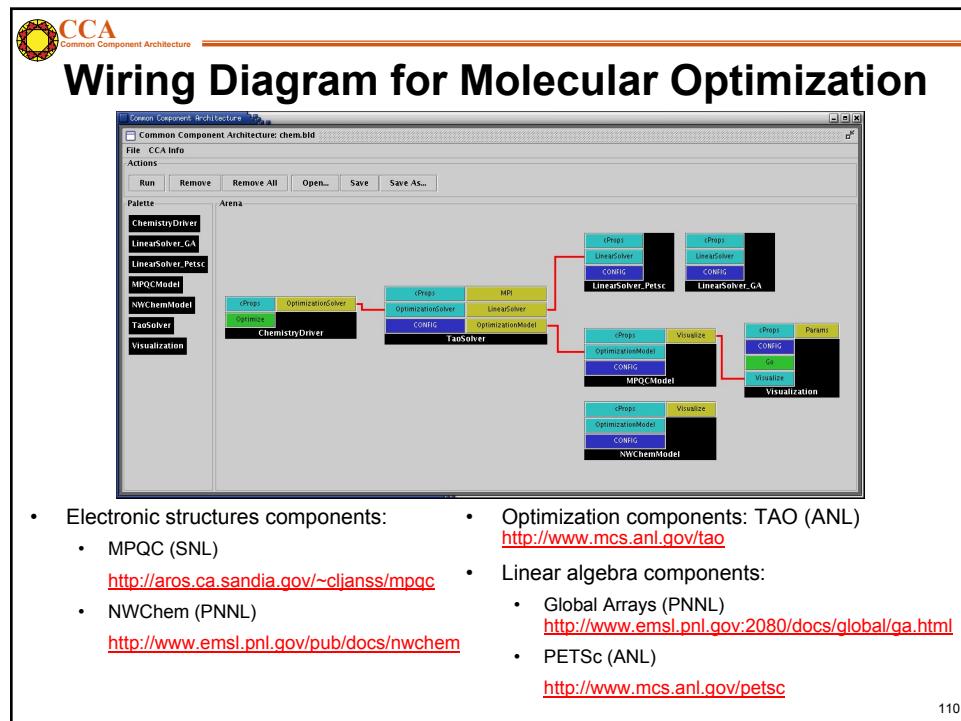
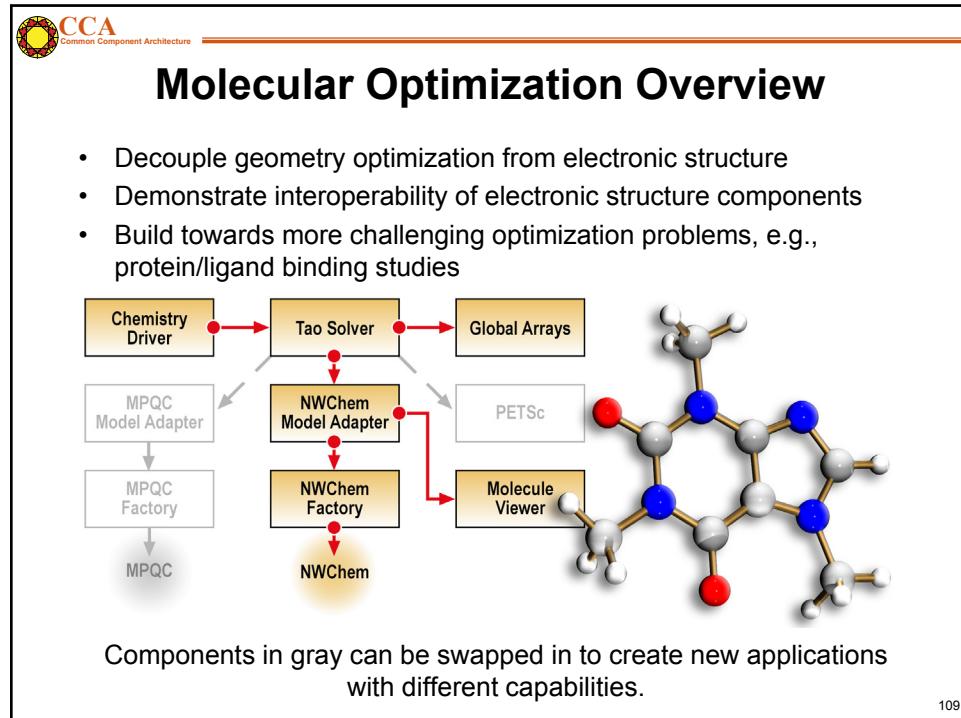
**CCA**  
Common Component Architecture

## Unconstrained Minimization Problem

- Given a rectangular 2-dimensional domain and boundary values along the edges of the domain
- Find the surface with minimal area that satisfies the boundary conditions, i.e., compute  

$$\min f(x), \text{ where } f: \mathbb{R} \rightarrow \mathbb{R}$$
- Solve using optimization components based on TAO (ANL)







## Actual Improvements

Molecule	NWChem	NWChem/TAO	MPQC	MPQC/TAO
Glycine	33	19	26	19
Isoprene	56	45	75	43
Phosphoserine	79	67	85	62
Aspirin	43	51	54	48
Cholesterol	33	30	27	30

Function and gradient evaluations

111



## Componentized Climate Simulations

- NASA's ESMF project has a component-based design for Earth system simulations
  - ESMF components can be assembled and run in CCA compliant frameworks such as Ccaffeine.
- Zhou et al (NASA Goddard) has integrated a simple coupled Atmosphere-Ocean model into Ccaffeine and is working on the Cane-Zebiak model, well-known for predicting *El Niño* events.
- Different PDEs for ocean and atmosphere, different grids and time-stepped at different rates.
  - Synchronization at ocean-atmosphere interface; essentially, interpolations between meshes
  - Ocean & atmosphere advanced in sequence
- Intuitively : Ocean, Atmosphere and 2 coupler components
  - 2 couplers : atm-ocean coupler and ocean-atm coupler.
  - Also a Driver / orchestrator component.

112

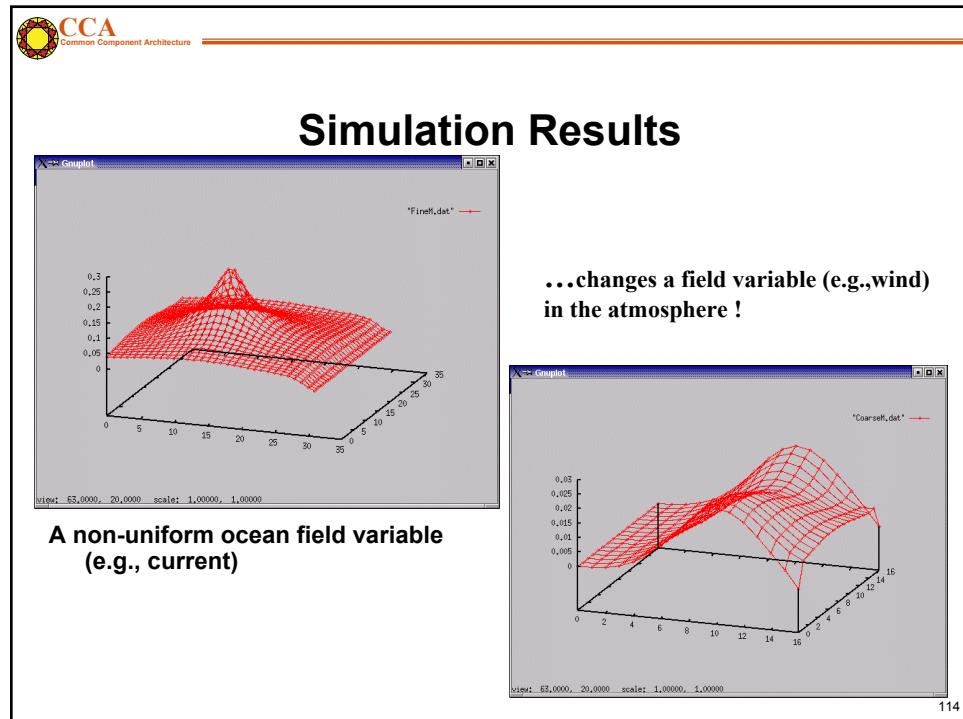
**CCA** Common Component Architecture

## Coupled Atmosphere-Ocean Model Assembly

**• Climate Component :**

- Schedule component coupling
- **Data flow is via pointer NOT data copy.**
  - All components in C++; run in CCAFEINE.
- **Multiple ocean models with the same interface**
  - Can be selected by a user at runtime

113





## Concurrency At Multiple Granularities

- Certain simulations need multi-granular concurrency
  - Multiple Component Multiple Data, multi-model runs
- Usage Scenarios:
  - Model coupling (e.g. Atmosphere/Ocean)
  - General multi-physics applications
  - Software licensing issues
- 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 (e.g. MxN)
    - Link as components (through AbstractFramework service; highly experimental at present)



115



## Componentizing your own application

- The key step: think about the decomposition strategy
  - By physics module?
  - Along numerical solver functionality?
  - Are there tools that already exist for certain pieces? (solvers, integrators, meshes?)
  - Are there common interfaces that already exist for certain pieces?
  - Be mindful of the level of granularity
- Decouple the application into pieces
  - Can be a painful, time-consuming process
- Incorporate CCA-compliance
- Compose your new component application
- Enjoy!

116



## Overview

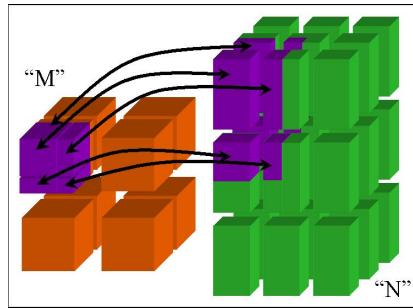
- Examples (scientific) of increasing complexity
  - Laplace equation
  - Time-dependent heat equation
  - Nonlinear reaction-diffusion system
  - Quantum chemistry
  - Climate simulation
- Tools
  - MxN parallel data redistribution
  - Performance measurement, modeling and scalability studies
- Community efforts & interface development
  - TSTT Mesh Interface effort
  - CCTTSS's Data Object Interface effort

117



## CCA Concepts: MxN Parallel Data Redistribution

- Share Data Among Coupled Parallel Models
  - Disparate Parallel Topologies (M processes vs. N)
  - e.g. Ocean & Atmosphere, Solver & Optimizer...
  - e.g. Visualization (Mx1, increasingly, MxN)



*Research area -- tools under development*

118

 CCA  
Common Component Architecture

## “MxN” Parallel Data Redistribution: The Problem...

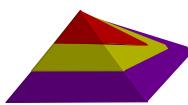
- Create complex scientific simulations by coupling together multiple parallel component models
  - Share data on “M” processors with data on “N”
    - $M \neq N \sim$  Distinct Resources (Pronounced “M by N”)
  - Model coupling, e.g., climate, solver / optimizer
  - Collecting data for visualization
    - $M \times 1$ ; increasingly  $M \times N$  (parallel rendering clusters)
- Define “standard” interface
  - Fundamental operations for any parallel data coupler
    - Full range of synchronization and communication options

119

 CCA  
Common Component Architecture

## Hierarchical MxN Approach

- Basic MxN Parallel Data Exchange
  - Component implementation
  - Initial prototypes based on CUMULVS & PAWS
    - Interface generalizes features of both
- Higher-Level Coupling Functions
  - Time & grid (spatial) interpolation, flux conservation
  - Units conversions...
- “Automatic” MxN Service via Framework
  - Implicit in method invocations, “parallel RMI”



<http://www.csm.ornl.gov/cca/mxn/>

120

**CCA Delivers Performance**

**Local**

- No CCA overhead **within** components
- Small overhead **between** components
- Small overhead for **language interoperability**
- Be aware of costs & design with them in mind
  - Small costs, easily amortized

**Parallel**

- No CCA overhead on **parallel computing**
- Use your favorite parallel programming model
- Supports SPMD and MPMD approaches

**Distributed (remote)**

- No CCA overhead – performance depends on networks, protocols
- CCA frameworks support OGSA/Grid Services/Web Services and other approaches

# of procs	% overhead
1	~0.05%
4	~0.12%
16	~0.15%
32	~0.18%
64	~0.20%
128	~0.20%
170	~0.12%

Maximum 0.2% overhead for CCA vs native C++ code for parallel molecular dynamics up to 170 CPUs

Problem Size	Linear Solver Component (sec)	PETSc SLES (sec)
100x100	~5	~5
150x150	~25	~25
200x200	~55	~55
250x250	~95	~95

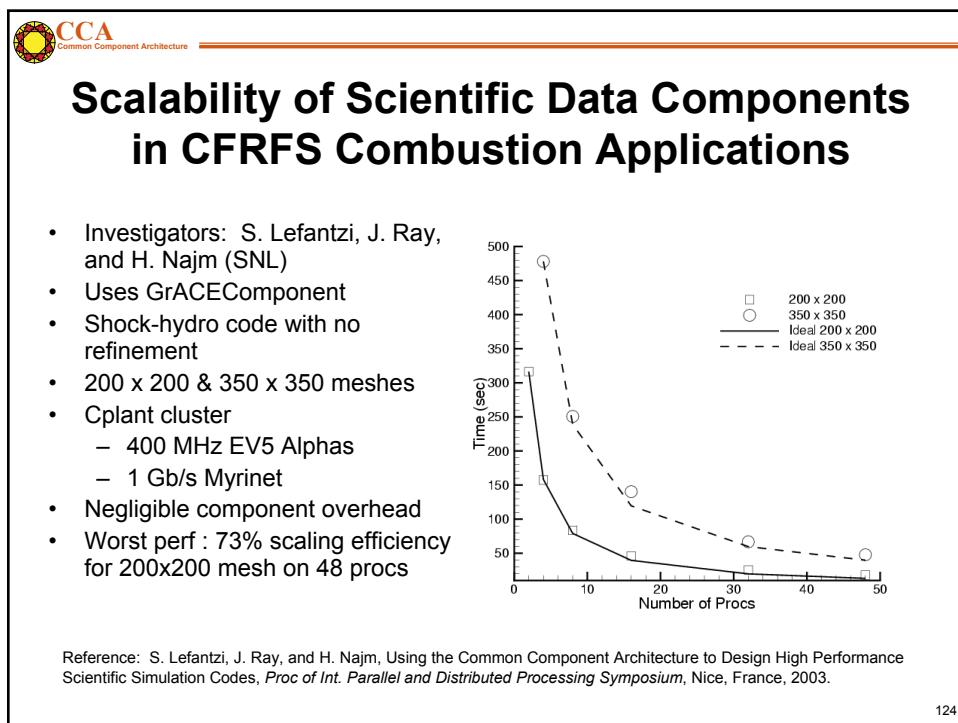
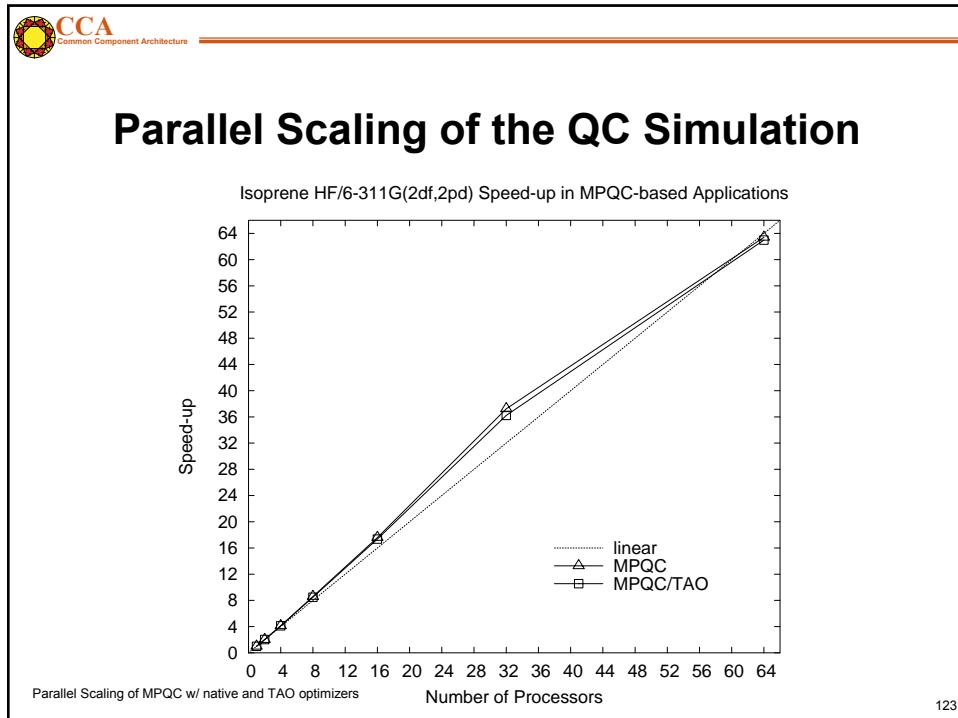
121

**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

Function arg type	f77	Component
Array	80 ns	224ns
Complex	75ns	209ns
Double complex	86ns	241ns

122



**CCA** Common Component Architecture

## Performance Measurement In A Component World

- CCA provides a novel means of profiling & modeling **component** performance
- Need to collect incoming inputs and match them up with the corresponding performance but how ?
  - Need to “instrument” the code
    - But has to be non-intrusive, since we may not “own” component code
- What kind of performance infrastructure can achieve this?
  - Previous research suggests proxies
    - Proxies serve to intercept and forward method calls

125

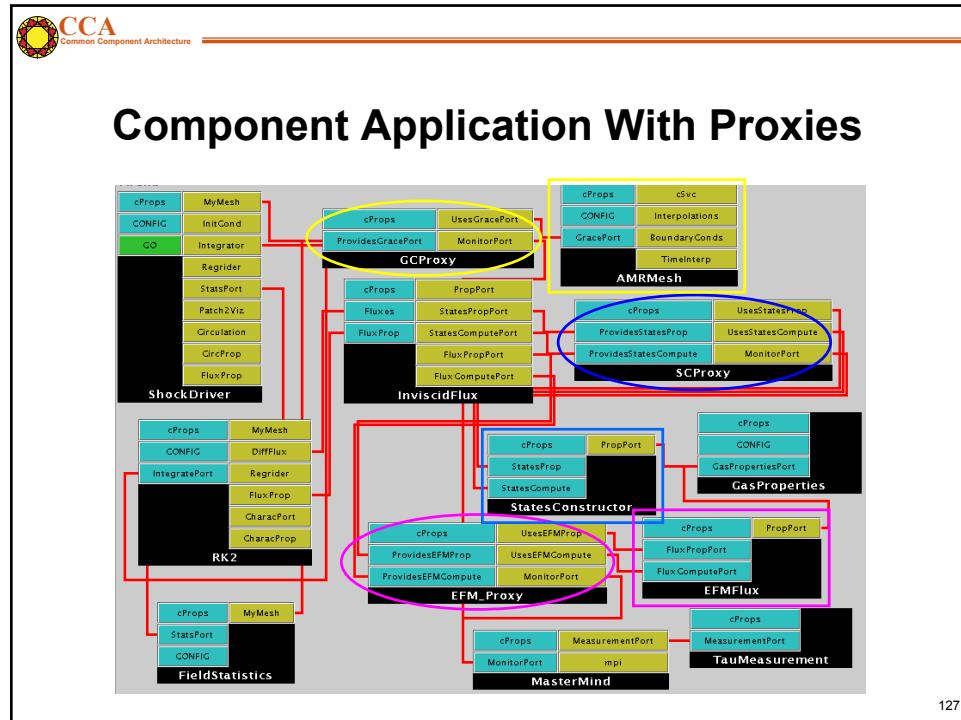
**CCA** Common Component Architecture

## “Integrated” Performance Measurement Capability

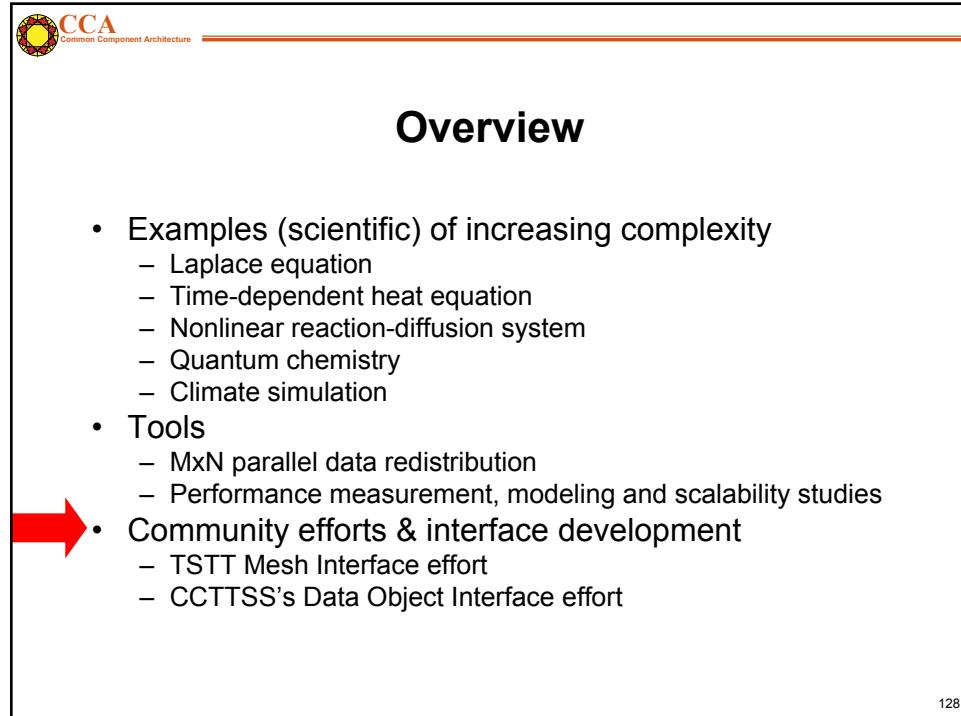
**Measurement infrastructure:**

- **Proxy**
  - Notifies MasterMind of all method invocations of a given component, along with performance dependent inputs
  - Generated automatically using PDT
- **MasterMind**
  - Collects and stores all measurement data
- **TAU**
  - Makes all performance measurements

126



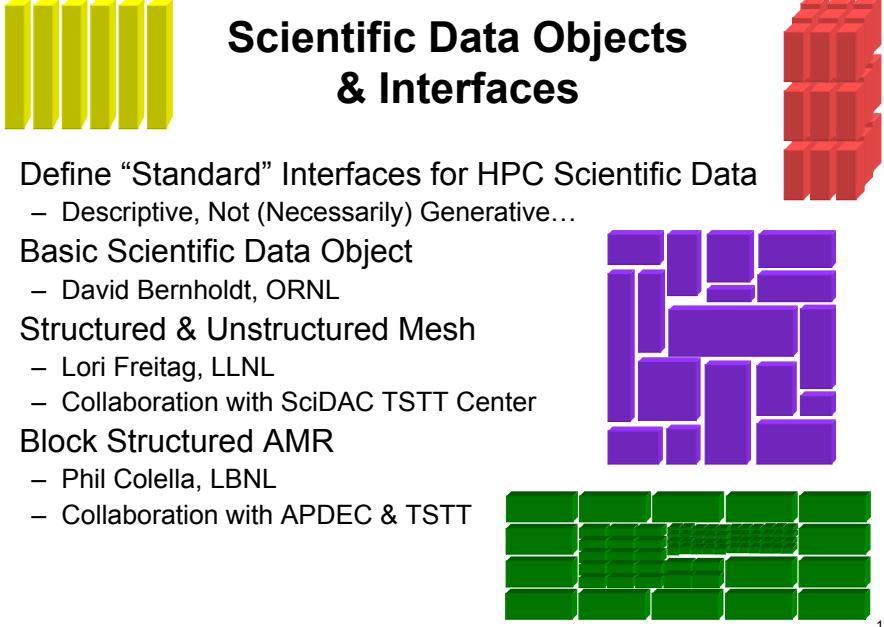
127



128

**CCA**  
Common Component Architecture

## Scientific Data Objects & Interfaces



- Define “Standard” Interfaces for HPC Scientific Data
  - Descriptive, Not (Necessarily) Generative...
- Basic Scientific Data Object
  - David Bernholdt, ORNL
- Structured & Unstructured Mesh
  - Lori Freitag, LLNL
  - Collaboration with SciDAC TSTT Center
- Block Structured AMR
  - Phil Colella, LBNL
  - Collaboration with APDEC & TSTT

129

**CCA**  
Common Component Architecture

## The Next Level

**TSTT**

- Common Interface Specification
  - Provides plug-and-play interchangeability
  - Requires domain specific experts
  - Typically a difficult, time-consuming task
  - A success story: MPI
- A case study... the TSTT/CCA mesh interface
  - TSTT = Terascale Simulation Tools and Technologies ([www.tstt-scidac.org](http://www.tstt-scidac.org))
  - A DOE SciDAC ISIC focusing on meshes and discretization
  - Goal is to enable
    - hybrid solution strategies
    - high order discretization
    - Adaptive techniques

130

**CCA** Common Component Architecture

## Proliferations of interfaces – the N<sup>2</sup> problem

Current Situation

- Public interfaces for numerical libraries are unique
- *Many-to-Many* couplings require *Many<sup>2</sup>* interfaces
  - Often a heroic effort to understand the inner workings of both codes
  - Not a scalable solution

```

graph LR
    DA[Dist. Array] <-->|3| ISIS[ISIS++]
    DA <-->|3| PETSc[PETSc]
    DA <-->|3| Trilinos[Trilinos]
    O[Overture] <-->|3| ISIS
    O <-->|3| PETSc
    O <-->|3| Trilinos
    PA[PAOMD] <-->|3| ISIS
    PA <-->|3| PETSc
    PA <-->|3| Trilinos
    S[SUMAA3d] <-->|3| ISIS
    S <-->|3| PETSc
    S <-->|3| Trilinos
  
```

131

**CCA** Common Component Architecture

## 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

```

graph LR
    DA[Dist. Array] <--> TS1[T, S, T, T]
    O[Overture] <--> TS1
    PA[PAOMD] <--> TS1
    S[SUMAA3d] <--> TS1
    TS1 <-->|2| ESI[E, S, I]
    ESI <-->|3| ISIS[ISIS++]
    ESI <-->|3| PETSc[PETSc]
    ESI <-->|3| Trilinos[Trilinos]
  
```

132



Common Component Architecture

## 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

133



Common Component Architecture



## CCTTSS Research Thrust Areas and Main Working Groups

- Scientific Components
  - Scientific Data Objects  
Lois Curfman McInnes, ANL (curfman@mcs.anl.gov)
- “MxN” Parallel Data Redistribution
  - Jim Kohl, ORNL (kohlija@ornl.gov)
- Frameworks
  - Language Interoperability / Babel / SIDL
  - Component Deployment / Repository  
Gary Kumfert, LLNL (kumfert@llnl.gov)
- User Outreach
  - David Bernholdt, ORNL (bernholdtde@ornl.gov)

134



Common Component Architecture

## Summary

- Complex applications that use components are possible
  - Combustion
  - Chemistry applications
  - Optimization problems
  - Climate simulations
- Component reuse is significant
  - Adaptive Meshes
  - Linear Solvers (PETSc, Trilinos)
  - Distributed Arrays and MxN Redistribution
  - Time Integrators
  - Visualization
- Examples shown here leverage and extend parallel software and interfaces developed at different institutions
  - Including CUMULVS, ESI, GrACE, LSODE, MPICH, PAWS, PETSc, PVM, TAO, Trilinos, TSTT.
- Performance is not significantly affected by component use
- Definition of domain-specific common interfaces is key

135



Common Component Architecture

## Language Interoperable CCA Components via



**CCA Forum Tutorial Working Group**  
<http://www.cca-forum.org/tutorials/>  
*tutorial-wg@cca-forum.org*

136



## Goal of This Module

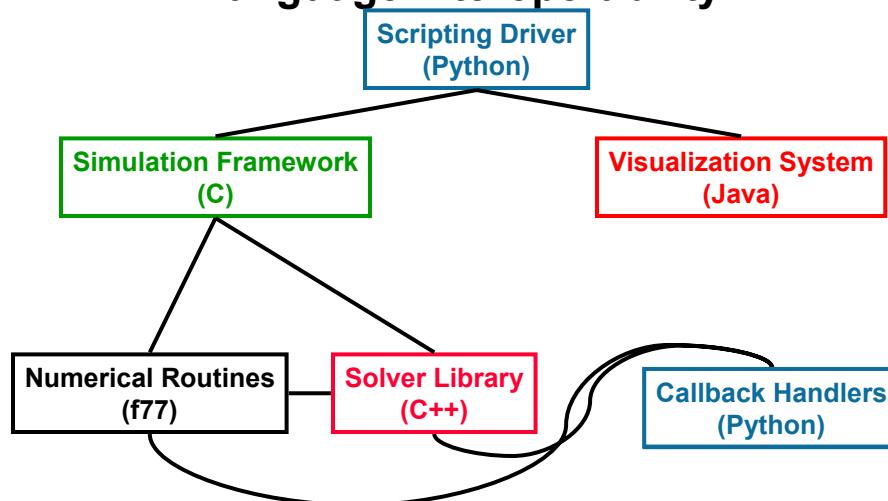
**Legacy codes → Babelized CCA Components**

- Introduction To:
  - Babel
  - SIDL
- See Babel in use
  - “Hello World” example
- Babel aspects of writing a CCA component

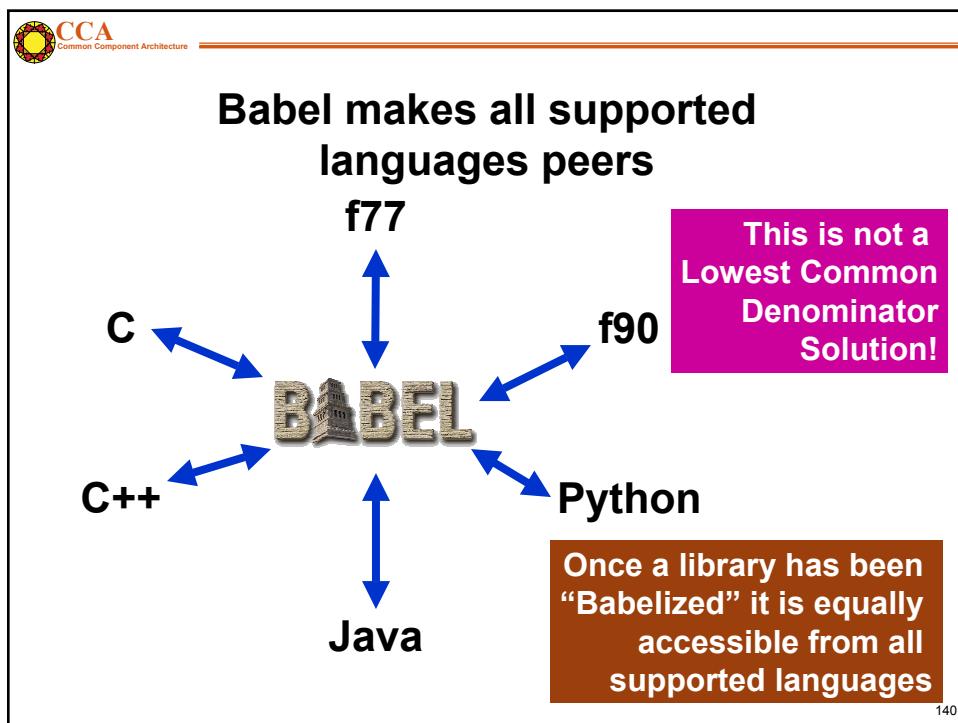
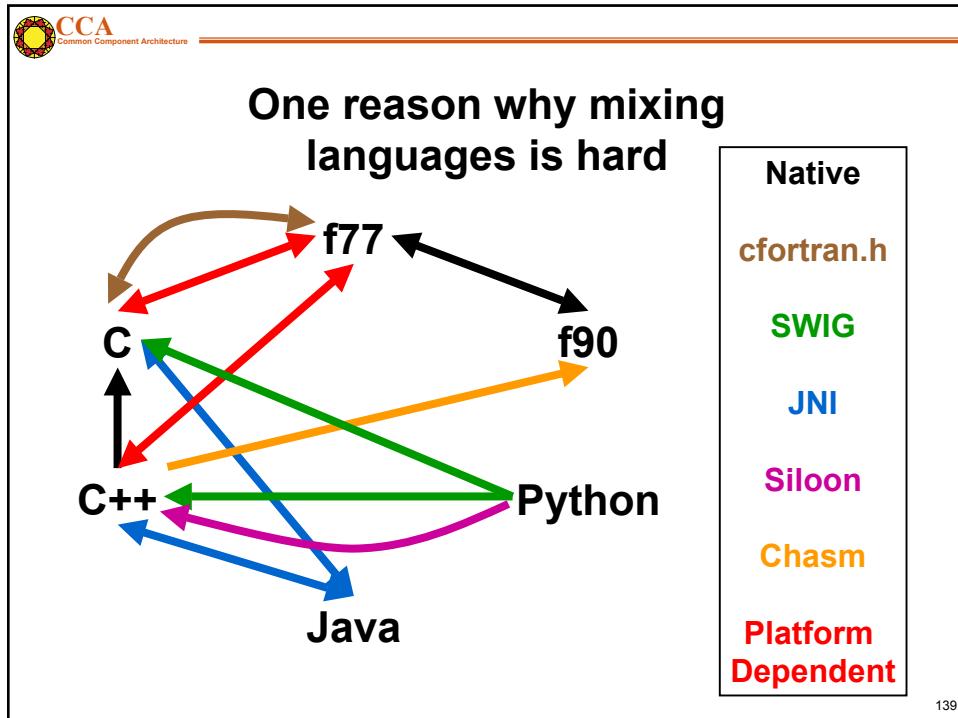
137



## What I mean by “Language Interoperability”



138

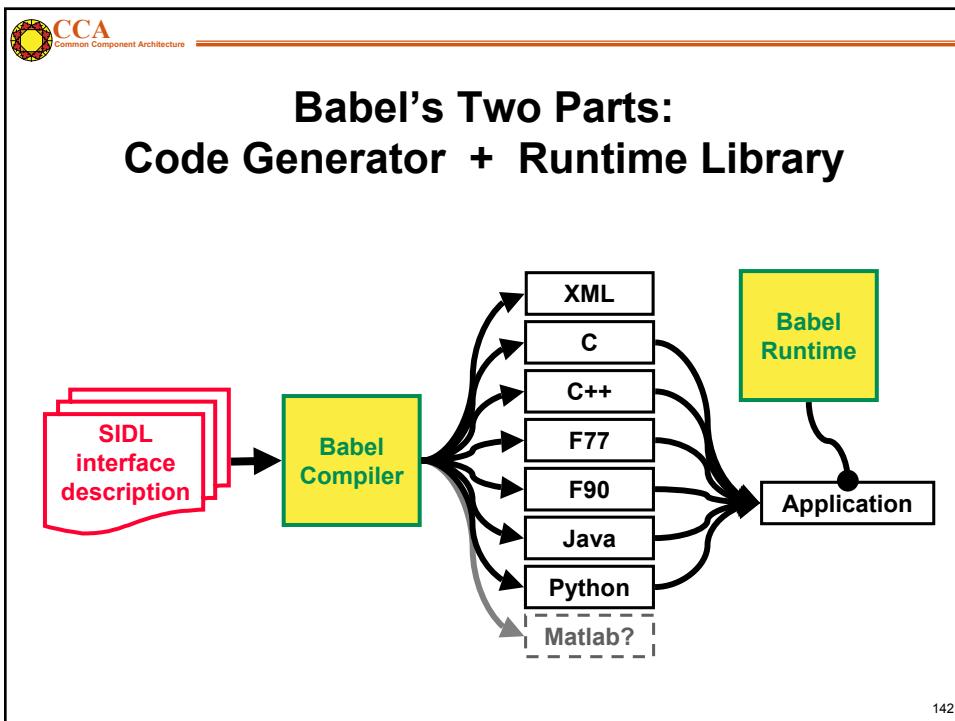


 CCA  
Common Component Architecture

## Babel Module's Outline

- Introduction
- **Babel Basics**
  - How to use Babel in a “Hello World” Example
  - SIDL Grammar
  - Wrapping legacy code
- Babel aspects of writing a CCA component

141





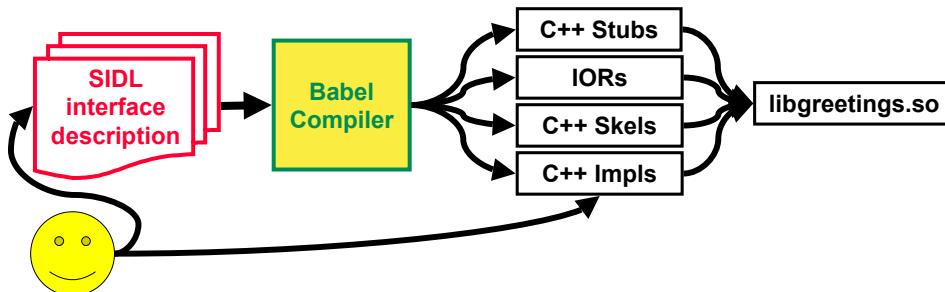
## greetings.sidl: A Sample SIDL File

```
package greetings version 1.0 {
    interface Hello {
        void setName( in string name );
        string sayIt( );
    }
    class English implements-all Hello { }
}
```

143



## Library Developer Does This...



1. `babel --server=C++ greetings.sidl`
2. Add implementation details
3. Compile & Link into Library/DLL

144

**CCA** Common Component Architecture

## Adding the Implementation

```

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)
}

```

45

**CCA** Common Component Architecture

## Library User Does This...

The diagram illustrates the workflow for a library user:

- A red box labeled "SIDL interface description" points to a yellow box labeled "Babel Compiler".
- The "Babel Compiler" box has two arrows pointing to a box containing "F90 Stubs" and "IOR Headers".
- An arrow points from the "F90 Stubs" and "IOR Headers" box to a box labeled "Application".
- The "Application" box has a connection to a yellow box labeled "Babel Runtime".
- A black box labeled "libgreetings.so" is shown below the "Application" box.

1. `babel --client=F90 greetings.sidl`
2. Compile & Link generated Code & Runtime
3. Place DLL in suitable location

146

 CCA  
Common Component Architecture

## F90/Babel “Hello World” Application

```

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
    
```

These subroutines come from directly from the SIDL

Some other subroutines are “built in” to every SIDL class/interface

147

 CCA  
Common Component Architecture

## SIDL Grammar (1/3): Packages and Versions

- Packages can be nested
 

```
package foo version 0.1 { package bar { ... } }
```
- Versioned Packages
  - defined as packages with explicit version number OR packages enclosed by a versioned package
  - Reentrant by default, but can be declared final
  - May contain interfaces, classes, or enums
- Unversioned Packages
  - Can only enclose more packages, not types
  - Must be re-entrant. Cannot be declared final

You'll use SIDL in the hands-on

148



## SIDL Grammar (2/3): Classes & Interfaces

- SIDL has 3 user-defined objects
  - **Interfaces** – APIs only, no implementation
  - **Abstract Classes** – 1 or more methods unimplemented
  - **Concrete Classes** – All methods are implemented
- Inheritance (like Java/Objective C)
  - Interfaces may **extend** Interfaces
  - Classes **extend** no more than one Class
  - Classes can **implement** multiple Interfaces
- Only concrete classes can be instantiated

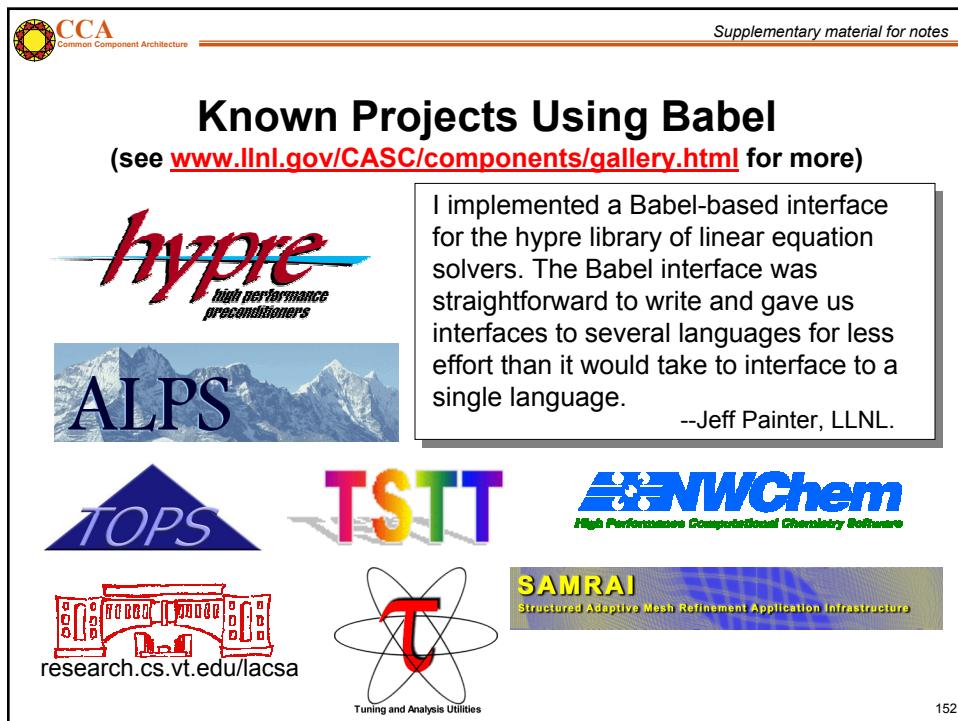
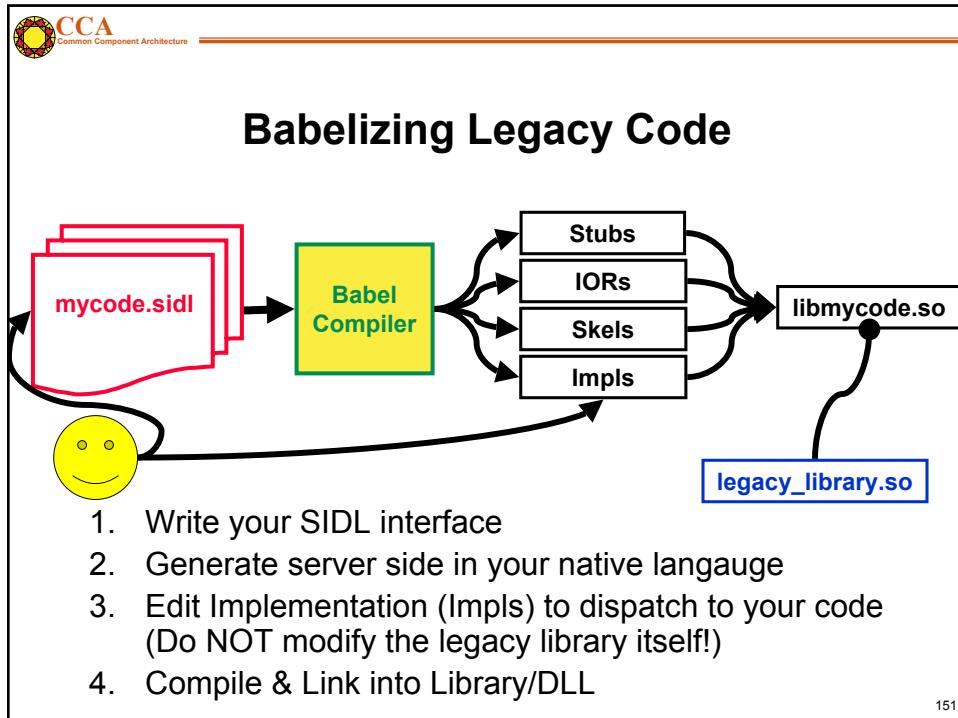
149



## SIDL Grammar (3/3): Methods and Arguments

- Methods are **public virtual** by default
  - **static** methods are not associated with an object instance
  - **final** methods can not be overridden
- Arguments have 3 parts
  - Mode: can be **in**, **out**, or **inout** (like CORBA, but semantically different than F90)
  - Type: one of (bool, char, int, long, float, double, fcomplex, dcomplex, array<*Type, Dimension*>, enum, interface, class )
  - Name

150





## Investing in Babelization can **improve** the interface to the code.

*"When Babelizing LEOS [an equation of state library at LLNL], I completely ignored the legacy interface and wrote the SIDL the way I thought the interface should be. After running Babel to generate the code, I found all the hooks I needed to connect LEOS without changing any of it. Now I've got a clean, new, object-oriented python interface to legacy code. Babel is doing much more than just wrapping here."*

-- Charlie Crabb, LLNL  
(conversation)

153



## Babel Module's Outline

- Introduction
- Babel Basics
  - How to use Babel in a “Hello World” Example
  - SIDL Grammar
- Babel aspects of writing a CCA component

154



## How to Write and Use Babelized CCA Components

1. Define “Ports” in SIDL
2. Define “Components” that implement those Ports, again in SIDL
3. Use Babel to generate the glue-code
4. Write the guts of your component(s)

155



## How to Write A Babelized CCA Component (1/3)

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

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

156

**CCA** Common Component Architecture

## How to Write A Babelized CCA Component (2/3)

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

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

```
class LinearFunction implements-all
    functions.Function, gov.cca.Component { }
```

157

**CCA** Common Component Architecture Supplementary material for notes

### Tip: Use Babel's XML output like precompiled headers in C++

```

graph LR
    subgraph Top [ ]
        direction TB
        ccasidl[ccा.sidl] --> babel1[Babel Compiler]
        functionsidl[functions.sidl] --> babel2[Babel Compiler]
        babel1 --> xml1[XML]
        babel2 --> xml2[XML]
        xml1 --> typeRepo[Type Repository]
        xml2 --> typeRepo
    end
    subgraph Bottom [ ]
        direction TB
        typeRepo --> stubs[Stubs]
        typeRepo --> iors[IORs]
        typeRepo --> skels[Skels]
        typeRepo --> impls[Impls]
    end

```

1. precompile SIDL into XML  
--text=xml
2. store XML in a directory
3. Use Babel's -R option to specify search directories

158

**How to Write A  
Babelized CCA Component (3/3)**

```

graph LR
    Repo["Repo (XML)"] --> Babel[Babel Compiler]
    SIDL["SIDL interface description"] --> Babel
    Babel --> CStubs["C Stubs"]
    Babel --> IORs["IORs"]
    Babel --> CSkels["C Skels"]
    Babel --> CImpls["C Impls"]
    CStubs --> Lib["libfunction.so"]
    IORs --> Lib
    CSkels --> Lib
    CImpls --> Lib
    
```

3. Use Babel to generate the glue code

- `babel --server=C --repo function.sidl`

4. Add implementation details

159

**Limitations of Babel's Approach  
to Language Interoperability**

Supplementary material for notes

- Babel is a code generator
  - Do obscure tricks no one would do by hand
  - Don't go beyond published language standards
- Customized compilers / linkers / loaders beyond our scope
  - E.g. icc and gcc currently don't mix on Linux
  - E.g. No C++-style templates in SIDL. (Would require special linkers/loaders to generate code for template instantiation, like C++ does.)
- Babel makes language interoperability feasible, but not trivial
  - Build tools severely underpowered for portable multi-language codes

160



## Contact Info

- Project: <http://www.llnl.gov/CASC/components>
  - Babel: language interoperability tool
  - Alexandria: component repository
  - Quorum: web-based parliamentary system
  - Gauntlet (coming soon): testing framework
- Bug Tracking: <http://www-casc.llnl.gov/bugs>
- Project Team Email: [components@llnl.gov](mailto:components@llnl.gov)
- Mailing Lists:  
[majordomo@lists.llnl.gov](mailto:majordomo@lists.llnl.gov)  
subscribe babel-users [*email address*]  
subscribe babel-announce [*email address*]