CCA
Common Component Architecture

Welcome to the Common Component Architecture Tutorial

SciDAC Conference 2007

29 June 2007

CCA Forum Tutorial Working Group

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

tutorial-wg@cca-forum.org

CCA
Common Component Architecture

Licensing Information

- This tutorial is distributed under the Creative Commons Attribution 2.5 License
  - http://creativecommons.org/licenses/by/2.5/
- In summary, **you are free:**
  - to copy, distribute, display, and perform the work
  - to make derivative works
  - to make commercial use of the work
- **Under the following conditions:**
  - **Attribution**. You must attribute the work in the manner specified by the author or licensor.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.
- **Your fair use and other rights are in no way affected by the above.**
- **Requested reference:**
  - **CCA Forum Tutorial Working Group, Common Component Architecture Tutorial, 2007, http://www.cca-forum.org/tutorials/**

CCA Common Component Architecture

Agenda & Table of Contents

No time for a hands-on session today, but the materials and codes are available for self-study at http://www.cca-forum.org/tutorials/

CCA Common Component Architecture

Network Access

- http://web.mit.edu/ist/topics/network/netguests.html
- Authorization may take up to 10 minutes
- We have no experience with this network, but if you have problems, we'll try to get you some help

CCA
Common Component Architecture

Who We Are: 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 tools for component-based software development, components, and component applications
- Open membership, quarterly meetings…

General mailing list: *cca-forum@cca-forum.org*

Web: http://www.cca-forum.org/

- Center for Technology for Advanced Scientific Component Software (TASCS)
  - Funded by the US DOE SciDAC program
  - Core development team for CCA technologies

CCA
Common Component Architecture

What Can Component Technology do for Scientific Computing?

CCA Forum Tutorial Working Group

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

tutorial-wg@cca-forum.org

Managing Code Complexity

Some Common Situations:

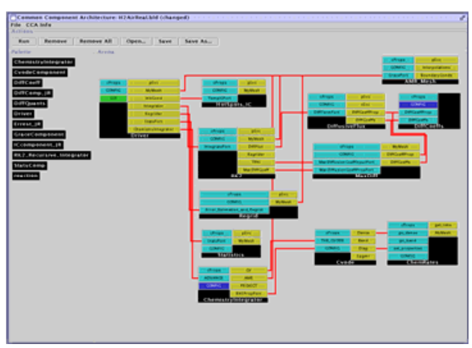
- Your code is so large and complex it has become fragile and hard to keep running
- You have a simple code, and you want to extend its capabilities – rationally
- You want to develop a computational "toolkit"
  - Many modules that can be assembled in different ways to perform different scientific calculations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

How CCA Can Help:

- Components help you think about software in manageable chunks that interact only in well-defined ways
- Components provide a "plug-and-play" environment that allows easy, flexible application assembly

Example: Computational Facility for Reacting Flow Science (CFRFS)

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



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

CCA tools used: Ccaffeine, and ccafe-gui

Languages: C, C++, F77

Helping Groups Work with Software

Some Common Situations:

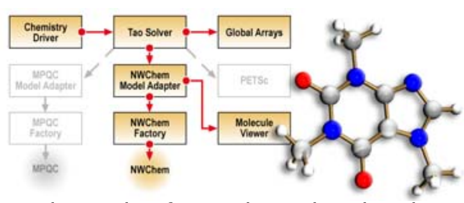
- Many (geographically distributed) developers creating a large software system
  - Hard to coordinate, different parts of the software don't work together as required
- Groups of developers with different specialties
- Forming communities to standardize interfaces or share code

How CCA Can Help:

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

Example: Quantum Chemistry

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



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

Components based on: MPQC, NWChem (quantum chem.), TAO (optimization), Global Arrays, PETSc (parallel linear algebra)
CCA tools used: Babel, Ccaffeine, and ccafe-gui
Languages: C, C++, F77, Python

Example: TSTT Unstructured Mesh Tool Interoperability

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

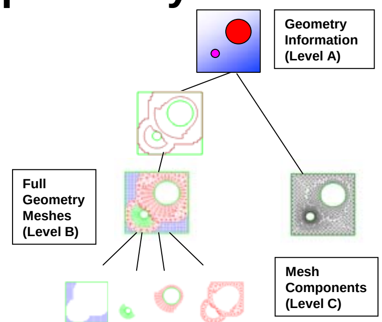


Illustration of geometry domain hierarchy used in TSTT mesh interface.

CCA tools used: Babel (SIDL), Chasm

Library languages: C, C++, F77, F90

Language Interoperability

Some Common Situations:

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

How CCA Can Help:

- Some component models (including CCA) allow transparent mixing of languages
- Babel (CCA's language interop. tool) can be used separately from other component concepts

CCA
Common Component Architecture

Examples

hypre

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

LAPACK07

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

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

-- Jeff Painter, LLNL. 2 June 2003

CCA tools used: Babel, Chasm

CCA
Common Component Architecture

Coupling Codes

Some Common Situations:

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

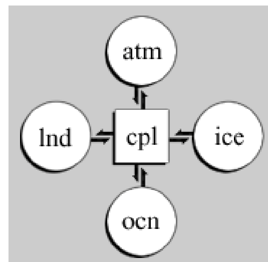
How CCA Can Help:

- Components are isolated from one another
  - Interactions via well-defined interfaces
  - An application can include multiple versions of a component
- Components can be composed flexibly, hierarchically
  - Standalone application as one assembly, coupled simulation as another
- CCA can be used in SPMD, MPMD, and distributed styles of parallel computing
- CCA is developing technology to facilitate data and functional coupling of parallel applications

CCA
Common Component Architecture

Example: Global Climate Modeling and the Model Coupling Toolkit (MCT)

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

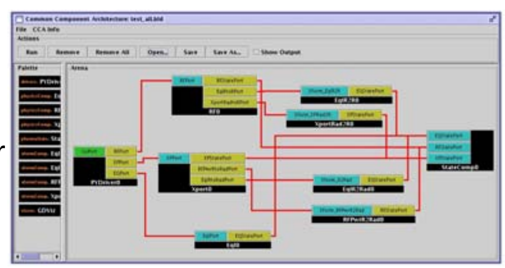


Schematic of CCSM showing coupler managing data exchanges between atmosphere, sea ice, ocean, and land models.
(From http://www.ccsm.ucar.edu/models/ccsm3.0/cpl6/)

CCA
Common Component Architecture

Example: Integrated Fusion Simulation

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



"Wiring diagram" for integrated fusion simulation.

Components based on: AORSA, Houlberg's transport library
New components: Driver, State
CCA tools used: Babel, Chasm, Ccaffeine, ccafe-gui
Languages: C++, F90, Python

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

CCA
Common Component Architecture

Goals of This Module

- Introduce basic concepts and vocabulary of component-based software engineering and the CCA
- Highlight the special demands of high-performance scientific computing on component environments
- Give you sufficient understanding of the CCA to begin evaluating whether it would be useful to you

CCA
Common Component Architecture

What are Components?

- No universally accepted definition in computer science research, but key features include…
- A unit of software development/deployment/reuse
  - i.e. has interesting functionality
  - Ideally, functionality someone else might be able to (re)use
  - Can be developed independently of other components
- Interacts with the outside world only through well-defined interfaces
  - Implementation is opaque to the outside world
- Can be composed with other components
  - "Plug and play" model to build applications
  - Composition based on interfaces

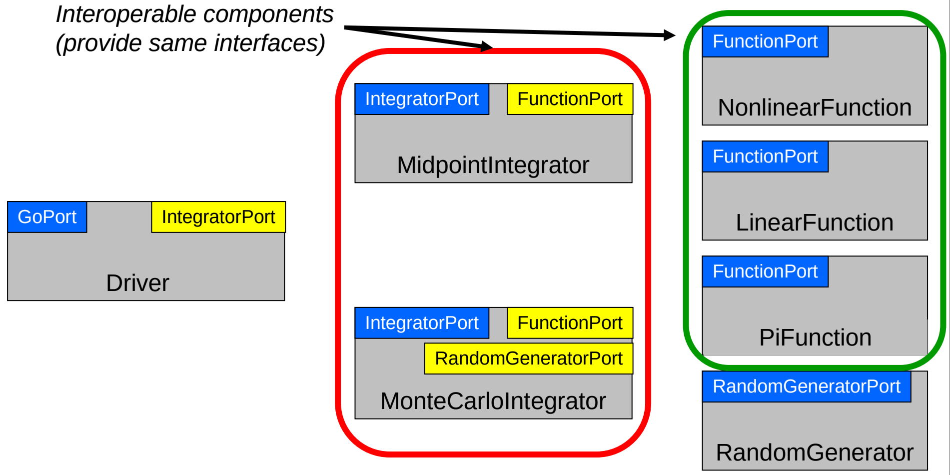
CCA
Common Component Architecture

What is a Component Architecture?

- A set of standards that allows:
  - Multiple groups to write units of software (components)…
  - And have confidence that their components will work with other components written in the same architecture
- These standards define…
  - The rights and responsibilities of a component
  - How components express their interfaces
  - The environment in which components are composed to form an application and executed (framework)
  - The rights and responsibilities of the framework

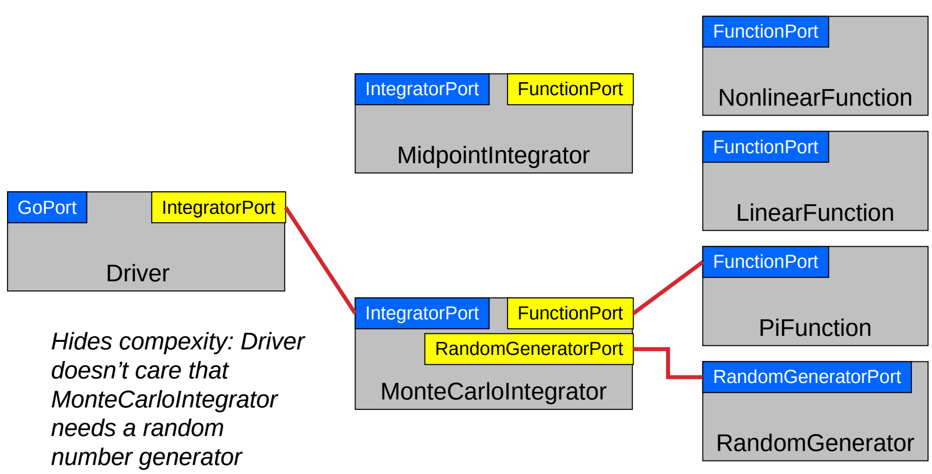
CCA Common Component Architecture

A Simple Example: Numerical Integration Components

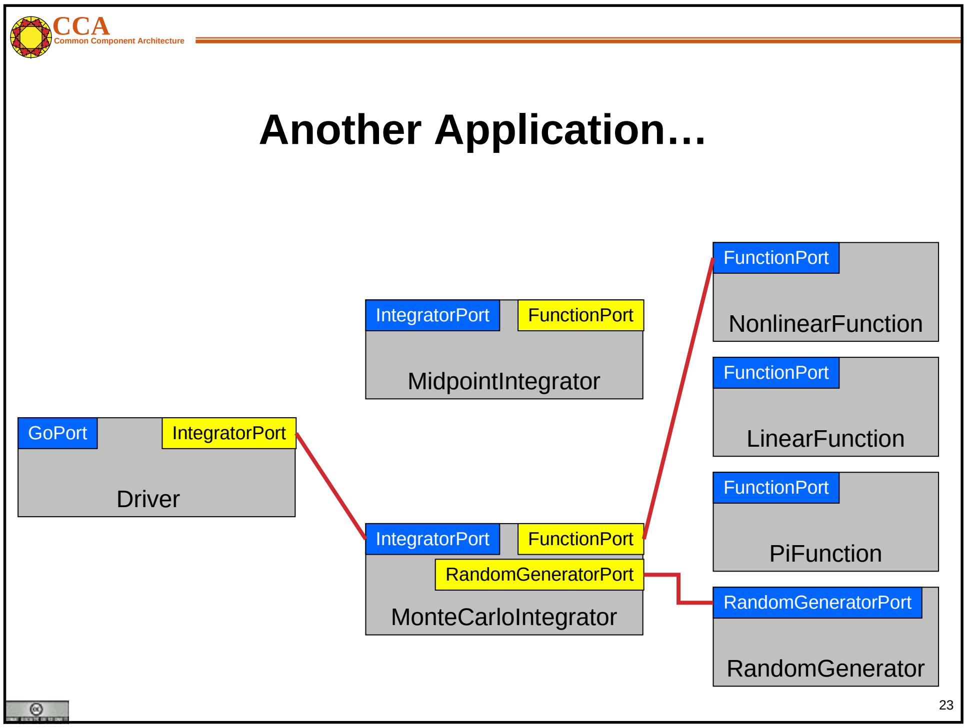


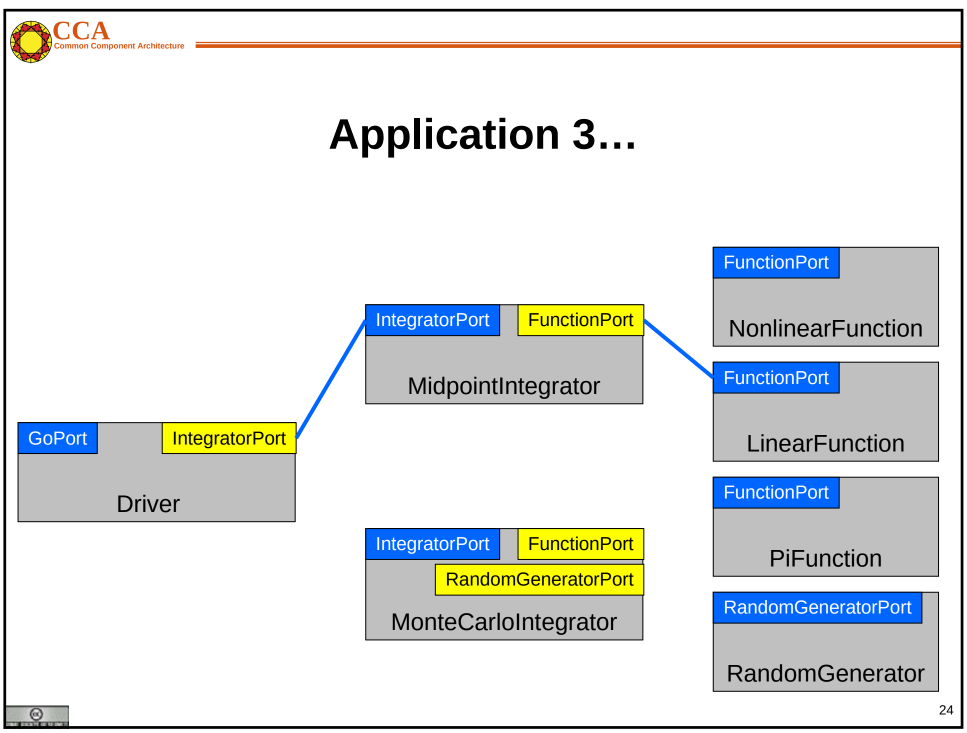
CCA Common Component Architecture

An Application Built from the Provided Components

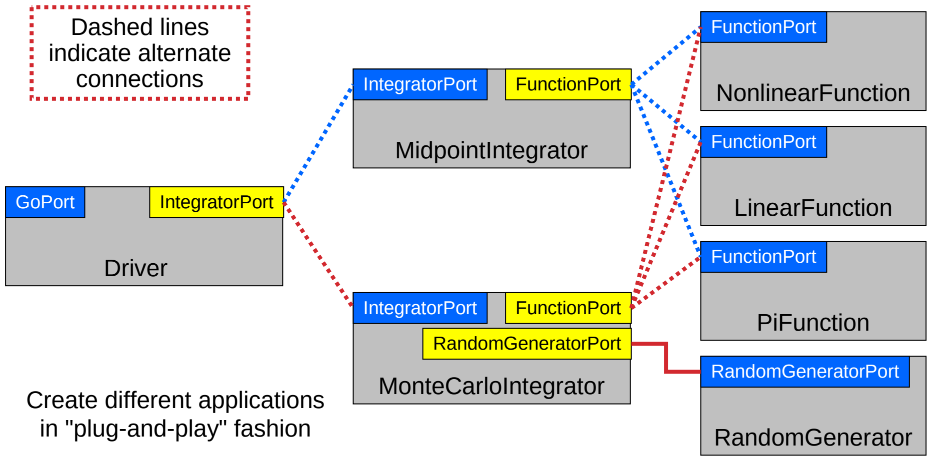


Hides compexity: Driver doesn't care that MonteCarloIntegrator needs a random number generator


CCA
Common Component Architecture
Another Application…
FunctionPort
NonlinearFunction
IntegratorPort
FunctionPort
MidpointIntegrator
FunctionPort
LinearFunction
GoPort
IntegratorPort
Driver
FunctionPort
PiFunction
IntegratorPort
FunctionPort
RandomGeneratorPort
MonteCarloIntegrator
RandomGeneratorPort
RandomGenerator
23


CCA
Common Component Architecture
Application 3…
FunctionPort
NonlinearFunction
IntegratorPort
FunctionPort
MidpointIntegrator
FunctionPort
LinearFunction
GoPort
IntegratorPort
Driver
FunctionPort
PiFunction
IntegratorPort
FunctionPort
RandomGeneratorPort
MonteCarloIntegrator
RandomGeneratorPort
RandomGenerator
24

And Many More…



Create different applications in "plug-and-play" fashion

Comparison of Application Development Approaches

Characteristics	Monolithic Simulation Code	Simulation Frameworks	Library -Based	Component -Based
Support for specific workflows and information flows	High	High	Low	Low
Flexibility w.r.t. workflow and information flow	Low	Medium	High	High
User-level extensibility	Low	Medium	High	High
Ease of incorporation of outside code (code reuse)	Low	Low-Medium	Medium	High
Ease of experimentation	Low	Medium	Medium	High
Amount of new code required to create a complete simulation	Low	Medium	High	High (reuse can reduce)
Breadth of current “ecosystem” for “plugins”	Low	Medium	High	Low (but growing)
Ease of coupling simulations	Low	Low	Medium	High

CCA Common Component Architecture

Be Aware: “Framework” Describes Many Things

- Currently in scientific computing, this term means different things to different people
- Basic software composition environment
  - Examples: CCA, CORBA Component Model, …
- An environment facilitating development of applications in a particular scientific domain (i.e. fusion, computational chemistry, …)
  - Example: Earth System Modeling Framework, http://www.esmf.ucar.edu
  - Example: Computational Facility for Reacting Flow Science, http://cfrfs.ca.sandia.gov
- An environment for managing complex workflows needed to carry out calculations
  - Example: Kepler: http://kepler-project.org
- Integrated data analysis and visualization environments (IDAVEs)
- Lines are often fuzzy
  - Example: Cactus, http://www.cactuscode.org
- Others types of frameworks *could* be built based on a basic software composition environment

CCA Common Component Architecture

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

CCA
Common Component Architecture

What is the CCA?

- Component-based software engineering has been developed in other areas of computing
  - Especially business and internet
  - Examples: CORBA Component Model, COM, Enterprise JavaBeans
- Many of the needs are similar to those in HPC scientific computing
- But scientific computing imposes special requirements not common elsewhere
- CCA is a component environment specially designed to meet the needs of HPC scientific computing

CCA
Common Component Architecture

Special Needs of Scientific HPC

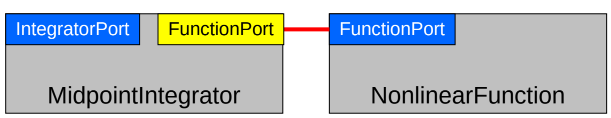
- Support for legacy software
  - How much change required for component environment?
- Performance is important
  - What overheads are imposed by the component environment?
- Both parallel and distributed computing are important
  - What approaches does the component model support?
  - What constraints are imposed?
  - What are the performance costs?
- Support for languages, data types, and platforms
  - Fortran?
  - Complex numbers? Arrays? (as first-class objects)
  - Is it available on my parallel computer?

CCA: Concept and Practice

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

31

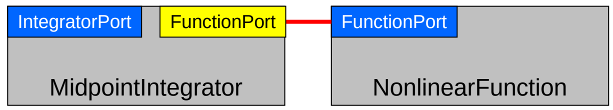
CCA Concepts: Components



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

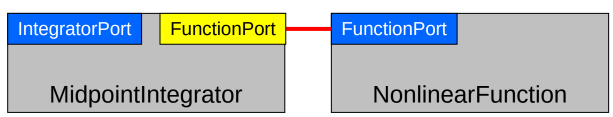
32

CCA Concepts: Ports

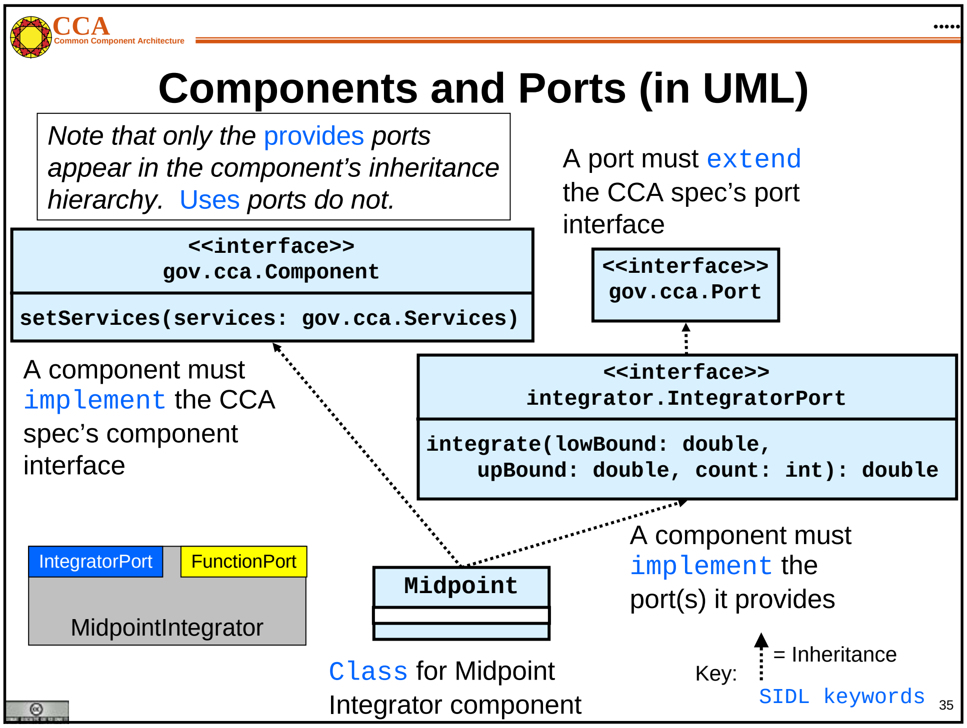


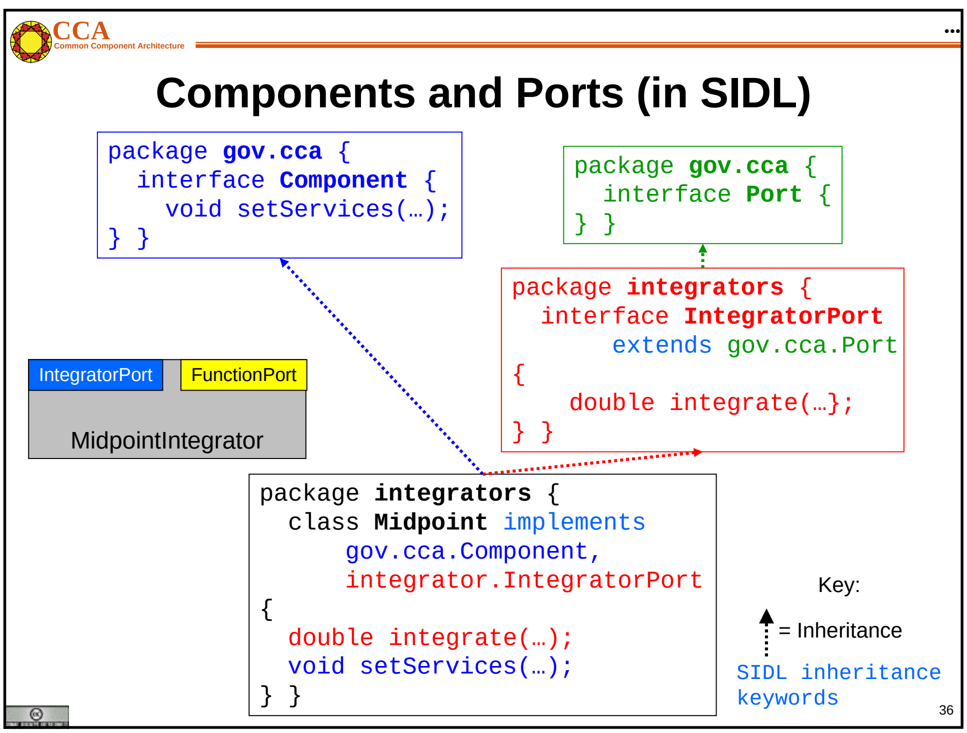
- Components interact through well-defined interfaces, or *ports*
  - A port expresses some computational functionality
  - In Fortran, a port is a bunch of subroutines or a module
  - In OO languages, a port is an abstract class or interface
- Ports and connections between them are a procedural (caller/callee) relationship, not dataflow!
  - e.g., *`FunctionPort`* could contain a method like `evaluate(in Arg, out Result)` with data flowing both ways

CCA Concepts: *Provides* and *Uses* Ports

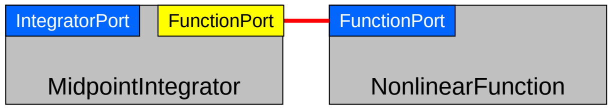


- Components may *provide* ports – implement the class or subroutines of the port (“Provides” Port)
  - *Providing* a port implies certain inheritance relationships between the component and the abstract definition of the interface (more details shortly)
  - A component can *provide* multiple ports
    - Different “views” of the same functionality, or
    - Related pieces of functionality
- Components may *use* ports – call methods or subroutines in the port (“Uses” Port)
  - *Use* of ports is just like calling a method normally except for a little additional work due to the “componentness” (more details shortly)
  - No inheritance relationship implied between caller and callee
  - A component can *use* multiple ports


CCA
Common Component Architecture
Components and Ports (in UML)
Note that only the provides ports appear in the component's inheritance hierarchy. Uses ports do not.
A port must extend the CCA spec's port interface
<<interface>>
gov.cca.Component
setServices(services: gov.cca.Services)
<<interface>>
gov.cca.Port
A component must implement the CCA spec's component interface
<<interface>>
integrator.IntegratorPort
integrate(lowBound: double,
upBound: double, count: int): double
IntegratorPort
FunctionPort
MidpointIntegrator
Midpoint
A component must implement the port(s) it provides
Class for Midpoint Integrator component
Key:
= Inheritance
SIDL keywords
35


CCA
Common Component Architecture
Components and Ports (in SIDL)
package gov.cca {
interface Component {
void setServices(…);
} }
package gov.cca {
interface Port {
} }
package integrators {
interface IntegratorPort
extends gov.cca.Port
{
double integrate(…};
} }
IntegratorPort
FunctionPort
MidpointIntegrator
package integrators {
class Midpoint implements
gov.cca.Component,
integrator.IntegratorPort
{
double integrate(…);
void setServices(…);
} }
Key:
= Inheritance
SIDL inheritance keywords
36

Using Ports



- Calling methods on a port you use requires that you first obtain a “handle” for the port
  - Done by invoking **`getPort()`** on the user’s **`gov.cca.Services`** object
  - Free up handle by invoking **`releasePort()`** when done with port
- Best practice is to bracket actual port usage as closely as possible without using **`getPort()`**, **`releasePort()`** too frequently
  - Can be expensive operations, especially in distributed computing contexts
  - Performance is in tension with dynamism
    - can’t “re-wire” a ports that is “in use”

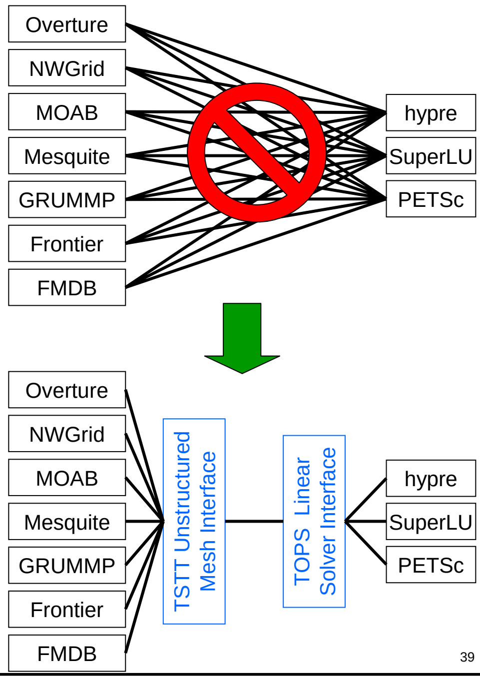
Where Do Ports Come From?

- Most ports are designed and implemented by users of CCA
  - May be specific to an application or used more broadly (i.e. community-wide)
- The CCA specification defines a *small* number of ports
  - Most are services CCA frameworks must provide for use by components
  - Some are intended for users to implement in their components, and have a special meaning recognized by the framework
    - *E.g.* *`gov.cca.ports.GoPort`* provides a very simple protocol to start execution of component-based applications

CCA Common Component Architecture

Interfaces are Key to Reuse and Interoperability of Code

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



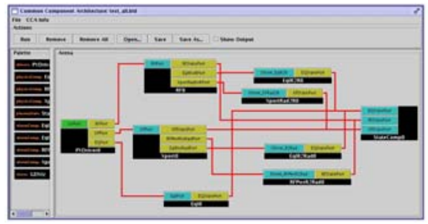
CCA Common Component Architecture

Interfaces are an Investment

- The larger the community, the greater the time & effort required to obtain agreement
  - Equally true in component and non-component environments
    - MPI 1.0 (well understood at the start) took 8 months, meeting every six weeks
    - MPI 2.0 (not well understood at the start) took 1.5 years, meeting every six weeks
  - Convenient communities are often “project” and “scientific domain”
- Formality of “standards” process varies
- Biggerstaff’s Rule of Threes
  - Must look at at least three systems to understand what is common (reusable)
  - Reusable software requires three times the effort of usable software
  - Payback only after third release

CCA Common Component Architecture

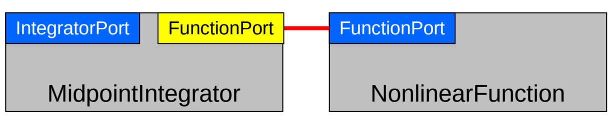
CCA Concepts: Frameworks



- The framework provides the means to “hold” components and compose them into applications
- Frameworks allow connection of ports without exposing component implementation details
- Frameworks provide a small set of standard services to components
  - Framework services are CCA ports, just like on components
  - Additional (non-standard) services can also be offered
  - Components can register ports as services using the *`ServiceProvider`* port
- *Currently:* specific frameworks are specialized for specific computing models (parallel, distributed, etc.)
- *Future:* better integration and interoperability of frameworks

CCA Common Component Architecture

Components Must Keep Frameworks Informed

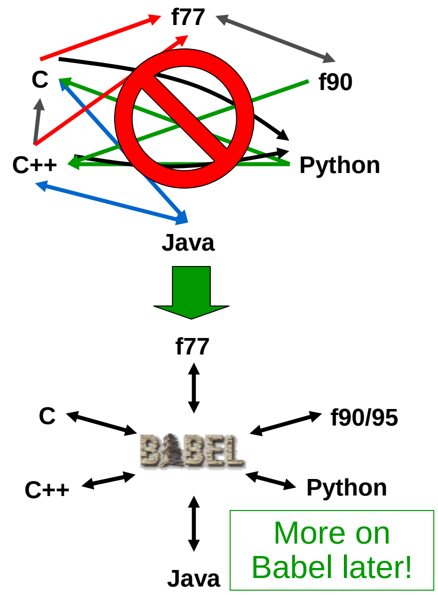


- Components must tell the framework about the ports they are providing and using
  - Framework will not allow connections to ports it isn’t aware of
- Register them using methods on the component’s **`gov.cca.Services`** object
  - **`addProvidesPort()`** and **`removeProvidesPort()`**
  - **`registerUsesPort()`**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

CCA Common Component Architecture

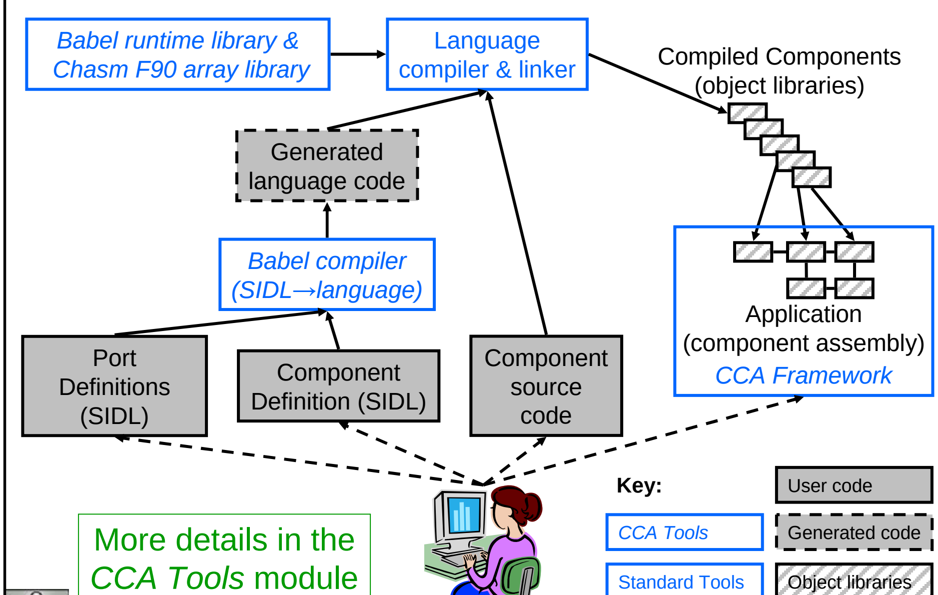
CCA Concepts: Language Interoperability

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



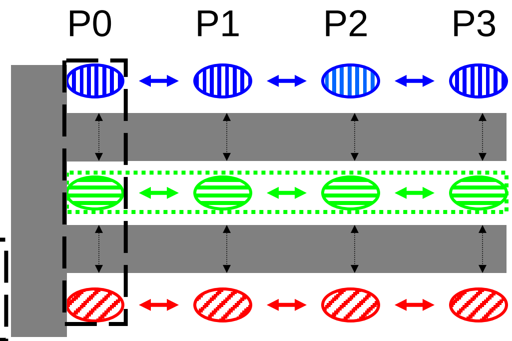
CCA Common Component Architecture

Coding in a CCA Environment



CCA Supports Parallelism -- by "Staying Out of the Way" of it

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



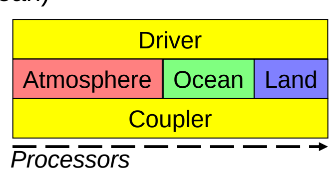
Components: Blue, Green, Red

Framework: Gray

Any parallel programming environments that can be mixed outside of CCA can be mixed inside

"Multiple-Component Multiple-Data" Applications in CCA

- Simulation composed of multiple SCMD sub-tasks
- Usage Scenarios:
  - Model coupling (e.g. Atmosphere/Ocean)
  - General multi-physics applications
  - Software licensing issues
    - i.e. limited number of instances

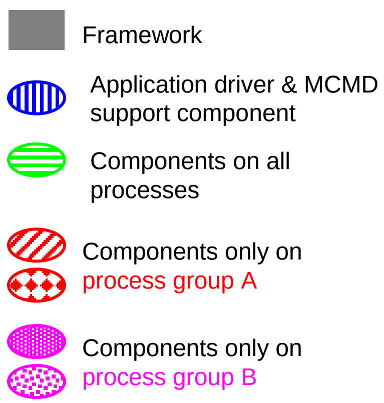


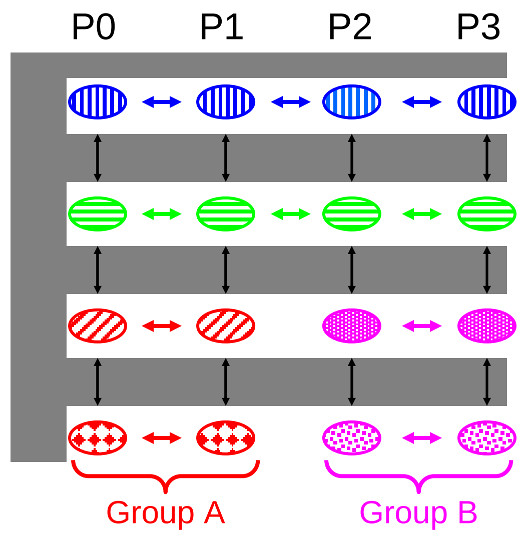
- Approaches
  - Run single parallel framework
    - Driver component that partitions processes and builds rest of application as appropriate (through BuilderService)
  - Run multiple parallel frameworks
    - Link through specialized communications components
    - Link as components (through AbstractFramework service)

CCA Common Component Architecture

MCMD Within A Single Framework

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

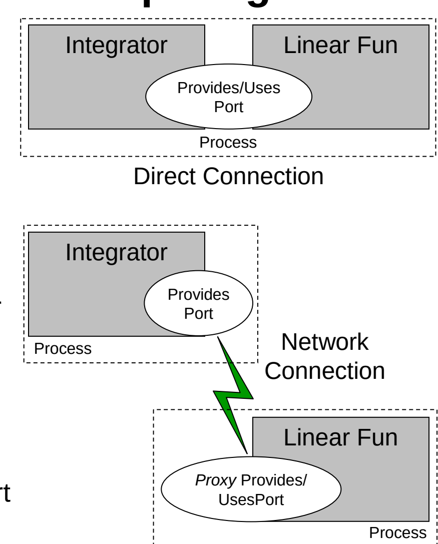




CCA Common Component Architecture

CCA Supports High-Performance Local and Distributed Computing

- "Direct connection" preserves high performance of local ("in-process") and parallel components
  - Framework makes *connection*
  - But is not involved in *invocation*
- Distributed computing has same uses/provides pattern, but framework intervenes between user and provider
  - Framework provides a *proxy* provides port local to the *uses* port
  - Framework conveys invocation from proxy to actual provides port



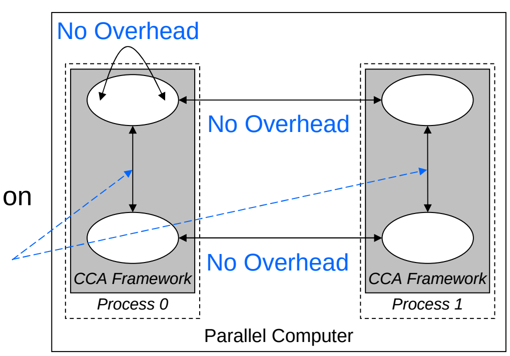
"Direct Connection" Details

- Directly connected components are in the same address space
  - Data can be passed by reference instead of copying
  - Just like "traditional" programs
  - Framework involved in connecting components, but not invocations on ports
- Cost of "CCAness" in a direct connect environment is one level of indirection on calls *between* components
  - Equivalent to a C++ virtual function call: lookup function location, invoke it
  - Overhead is on the invocation only (i.e. latency), not the total execution time
  - Cost equivalent of ~2.8 F77 or C function calls
  - ~48 ns vs 17 ns on 500 MHz Pentium III Linux box

Performance, the Big Picture

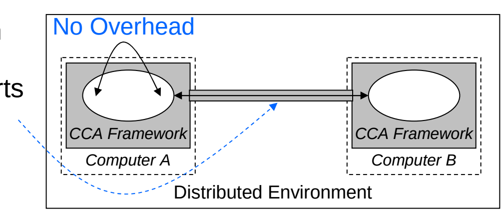
Direct-Connect, Parallel

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



Distributed

- No CCA overhead on calls within component
- Overheads on invocations on ports
  - Language interoperability (some data types)
  - Framework
  - (Wide area) network



CCA
Common Component Architecture

Maintaining HPC Performance

- The performance of your application is as important to us as it is to you
- The CCA is designed to provide maximum performance
  - But the best we can do is to make your code perform no worse
- Facts:
  - Measured overheads per function call are low
  - Most overheads easily amortized by doing enough work per call
  - Other changes made during componentization may also have performance impacts
  - *Awareness* of costs of abstraction and language interoperability facilitates design for high performance

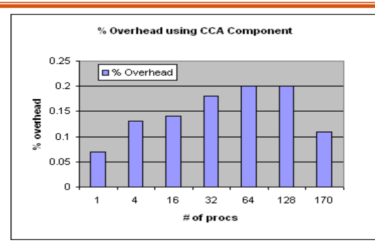
More about performance in notes

CCA
Common Component Architecture

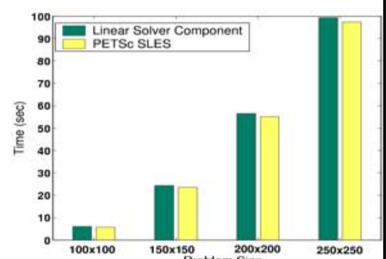
Supplementary material for handouts

Some Performance Results and References

- Lois Curfman McInnes,et al. **Parallel PDE-Based Simulations Using the Common Component Architecture**. In Are Magnus Bruaset, Petter Bjorstad, and Aslak Tveito, editors, *Numerical Solution of PDEs on Parallel Computers*. Springer-Verlag, 2005. Invited chapter, in press.
- S. Benson, et al. **Using the GA and TAO Toolkits for Solving Large-Scale Optimization Problems on Parallel Computers**. Technical report ANL/MCS-P1084-0903, Argonne National Laboratory, September 2003.
- Boyana Norris, et al. **Parallel Components for PDEs and Optimization: Some Issues and Experiences**. *Parallel Computing*, 28:1811--1831, 2002.
- David E. Bernholdt, et al. **A Component Architecture for High-Performance Computing**. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, 2002.



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



Aggregate time for linear solver component in unconstrained minimization problem w/ PETSc

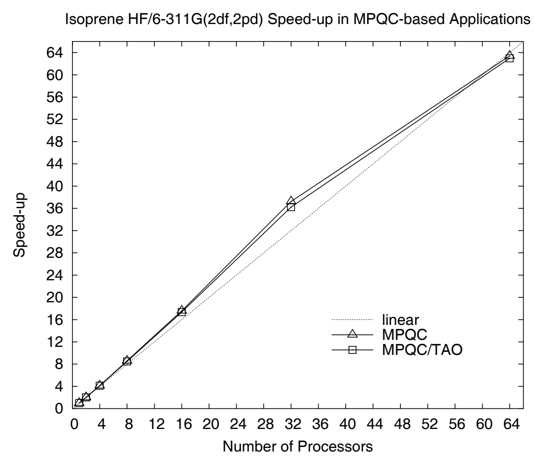
Overhead from Component Invocation

- 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

Scalability : Component versus Non-component. I

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



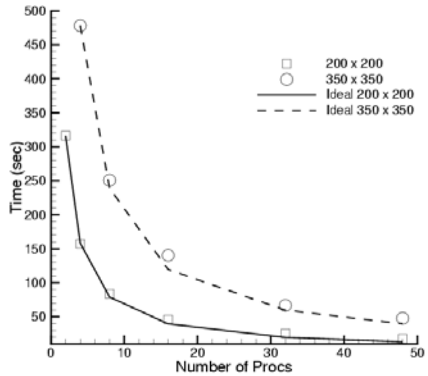
Parallel Scaling of MPQC w/ native and TAO optimizers

CCA
Common Component Architecture

Supplementary material for handouts

Scalability : Component versus Non-component. II

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



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.

CCA
Common Component Architecture

Advanced CCA Concepts

Brief introductions only, but more info is available – just ask us!

- Leveraging the component environment to provide additional capabilities to software developers
- The Proxy Component pattern (Hands-On Ch. 6, papers)
- Component lifecycle (tutorial notes, Hands-On)
- Components can be dynamic (papers)
- Improving the quality of component software (papers)
- Support for advanced parallel/high-performance computing (papers)

CCA
Common Component Architecture

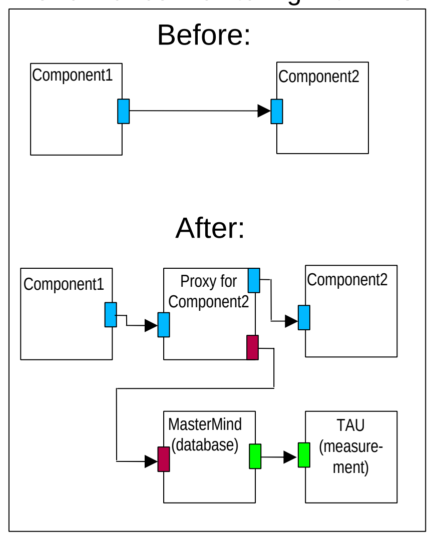
The Proxy Component Pattern

- A "proxy" component can be inserted between the user and provider of a port without either being aware of it (non-invasive)
- Proxy can observe or act on all invocations of the interface
- Similar to aspect-oriented programming
- For many purposes, proxies can be generated automatically from SIDL definition of the port

Sample uses for proxy components:

- Performance: instrumentation of method calls
- Debugging: execution tracing, watching data values
- Testing: Capture/replay

Performance Monitoring with TAU



CCA
Common Component Architecture

Component Lifecycle

Additional material in notes

- **Composition Phase (assembling application)**
  - Component is instantiated in framework
  - Component interfaces are connected appropriately
- **Execution Phase (running application)**
  - Code in components uses functions provided by another component
- **Decomposition Phase (termination of application)**
  - Connections between component interfaces may be broken
  - Component may be destroyed

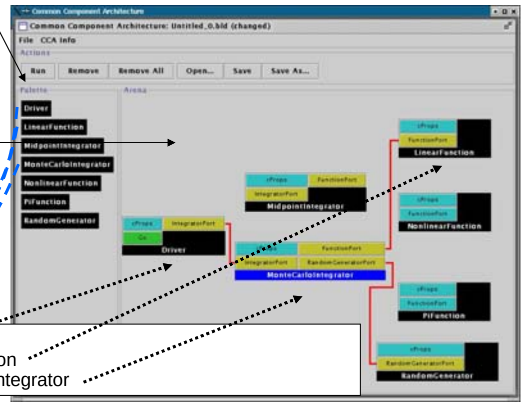
In an application, individual components may be in different phases at different times

Steps may be under human or software control

User Viewpoint: Loading and Instantiating Components

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

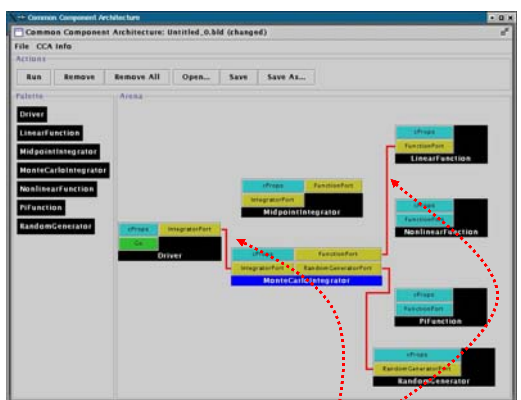
- Details are framework-specific!
- Ccaffeine currently provides both command line and GUI approaches



```
create   Driver                Driver
create   LinearFunction        LinearFunction
create   MonteCarloIntegrator  MonteCarloIntegrator
```


User Connects Ports

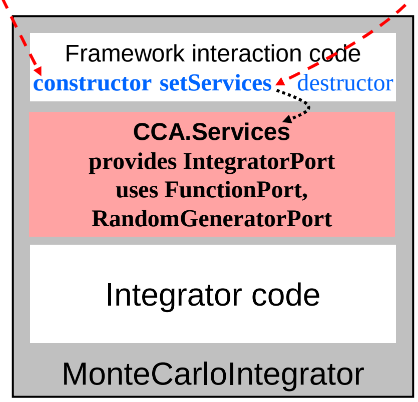
- Can only connect uses & provides
  - Not uses/uses or provides/provides
- Ports connected by type, not name
  - Port names must be unique within component
  - Types must match across components
- Framework puts info about *provider* of port into *using component's* Services object



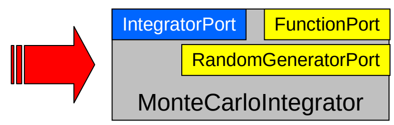
```
connect  Driver                IntegratorPort  MonteCarloIntegrator  IntegratorPort
connect  MonteCarloIntegrator  FunctionPort    LinearFunction        FunctionPort
…
```

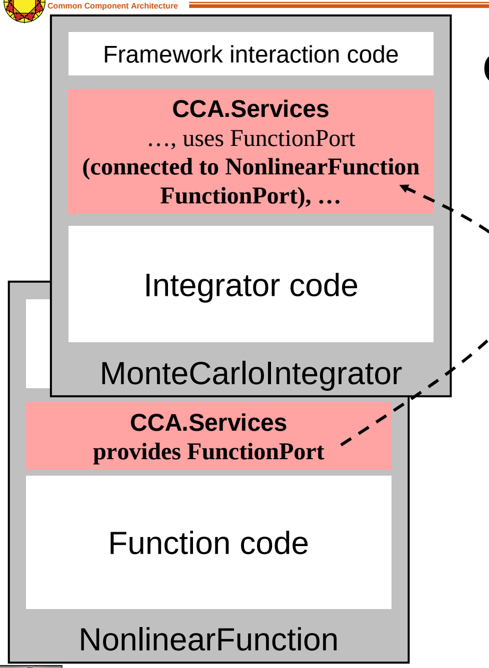
Component's View of Instantiation

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



- 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





Component's View of Connection

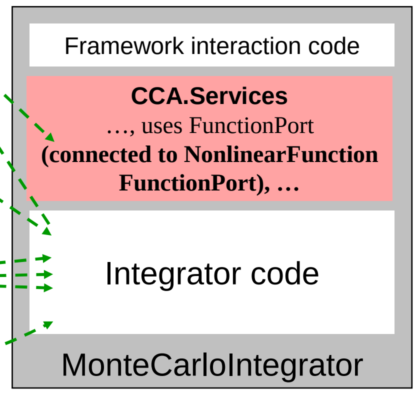
- Framework puts info about provider into user component's Services object
  - ***MonteCarloIntegrator***'s Services object is aware of connection
  - ***NonlinearFunction*** is not!
- ***MCI***'s integrator code cannot yet call functions on FunctionPort

Supplementary material for handouts

Component's View of Using a Port

- User calls getPort to obtain (handle for) port from Services
  - Finally user code can "see" provider
- Cast port to expected type
  - OO programming concept
  - Insures type safety
  - Helps enforce declared interface
- Call methods on port
  - e.g.

    sum = sum + function->evaluate(x)
- Call releasePort



Dynamic Component Assemblies

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

Sample uses of *BuilderService*:

- Python "driver" script which can assemble and control an application
  - i.e. MCMD climate model
- Adaptation to changing conditions
  - Swap components in and out to give better performance, numerical accuracy, convergence rates, etc.
  - TASCS project "Computational Quality of Service" activity

CCA Common Component Architecture

Enhancing Software Quality

- Current component architectures define syntax of interfaces
- Extend interface to include semantics (behavior) for more complete definition
  - "Design by contract"
  - Help ensure component performs correctly
  - Help ensure component is used correctly
- Selective enforcement to control impact
- TASCS project "Software Quality and Verification" activity

```
package vector version 1.0 {
  class Utils { …
    static double norm(in array<double> u,
                       in double tol,
                       in int badLevel)
      require    /* Preconditions */
        not_null : u != null;
        u_is_1d : dimen(u) == 1;
        non_negative_tolerance : tol >= 0.0;
      ensure    /* Postconditions */
        no_side_effects : is pure;
        non_negative_result : result >= 0.0;
        nearEqual(result, 0.0, tol)
          iff isZero(u, tol);
  … }
}
```


CCA Common Component Architecture

Supporting Emerging HPC Hardware Environments

- CCA does not dictate a specific approach to parallelism
- Different approaches and tools can be provided via components and custom frameworks

Examples…

- Uintah Computational Framework (Utah) provides a multi-threaded parallel execution environment based on task graphs
  - Specialized to certain structured adaptive mesh refinement problems
- TASCS developing services to manage groups of parallel components/tasks (MCMD)

Also…

- TASCS developing support for heterogeneous processor environments
  - FPGAs, GP-GPUs, accelerators, and other co-processors
  - Accelerator code encapsulated as components, interacting w/ components on primary processors
- Integration of fault tolerance capabilities with CCA under development (CIFTS-TASCS collaboration)

CCA
Common Component Architecture

Is CCA for You?

- Much of what CCA does can be done without such tools *if* you have sufficient discipline
  - The larger a group, the harder it becomes to impose the necessary discipline
- Projects may use different aspects of the CCA
  - CCA is *not* monolithic – use what *you* need
  - Few projects use all features of the CCA… initially
- Evaluate what *your* project needs against CCA's capabilities
  - Other groups' criteria probably differ from yours
  - CCA continues to evolve, so earlier evaluations may be out of date
- Evaluate CCA against other ways of obtaining the desired capabilities
- Suggested starting point:
  - CCA tutorial "hands-on" exercises

CCA
Common Component Architecture

Take an Evolutionary Approach

- The CCA is designed to allow selective use and incremental adoption
- "SIDLize" interfaces incrementally
  - Start with essential interfaces
  - Remember, only externally exposed interfaces need to be Babelized
- Componentize at successively finer granularities
  - Start with whole application as one component
    - Basic feel for components without "ripping apart" your app.
  - Subdivide into finer-grain components as appropriate
    - Code reuse opportunities
    - Plans for code evolution

CCA
Common Component Architecture

View it as an Investment

- CCA is a long-term investment in your software
  - Like most software engineering approaches
- There is a cost to adopt
- The payback is longer term
- Remember Biggerstaff's Rule of Threes
  - Look at three systems, requires three times the effort, payback after third release

CCA
Common Component Architecture

CCA is Still Under Development

- We've got…
  - A stable component model
  - Working tools
  - Active users
- But…
  - We know its not perfect
  - We're not "done" by any stretch
- Talk to us…
  - If you're evaluating CCA and and need help or have questions
  - If you don't think CCA meets your needs
  - If you've got suggestions for things we might do better

CCA
Common Component Architecture

What Can CCA Do Today?

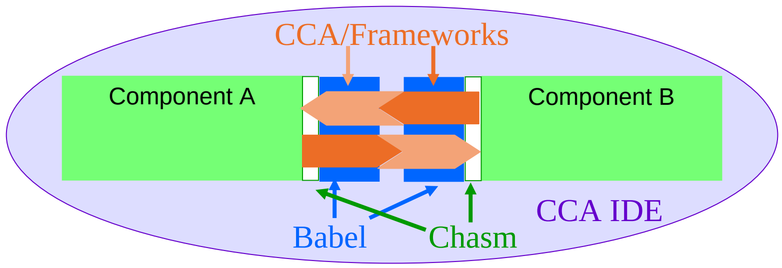
- Ccaffeine framework for HPC/parallel
  - XCAT and other options for distributed computing
- Language interoperability
  - Fortran 77/90/95, C, C++, Java, Python
  - Support for Fortran/C user-defined data structures under development
- CCA Tools working on a variety of platforms
  - Linux most widely used
  - Mac OS X second
  - Some IBM AIX users
  - Works on Cray XT series, port to IBM BlueGene in progress
  - Porting is driven by user needs, so let us know!

CCA
Common Component Architecture

CCA Tools – Language Interoperability and Frameworks

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

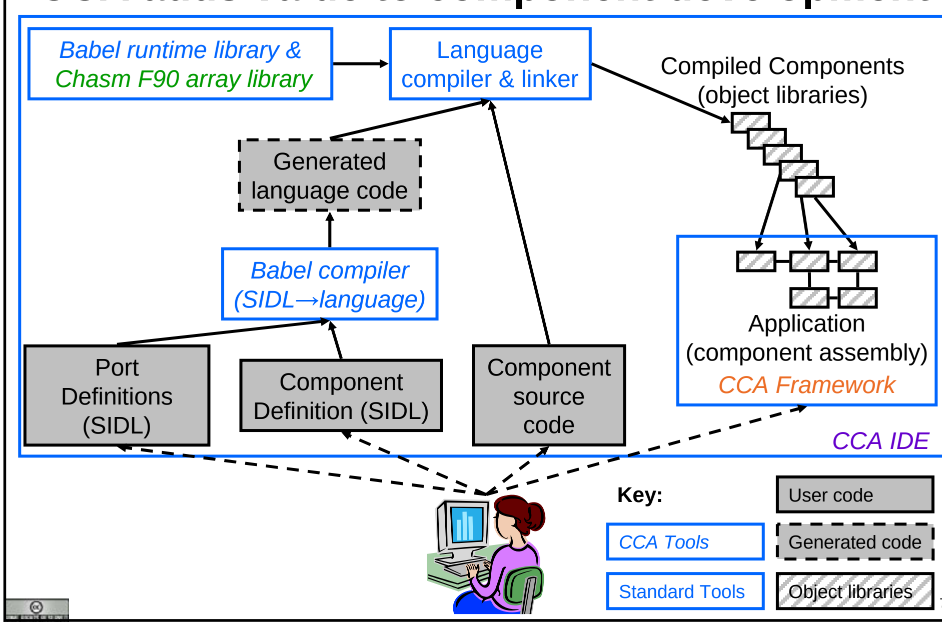
Goal of This Module



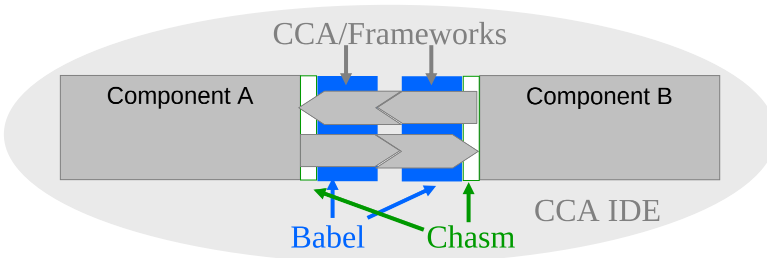
- Describe tools for multi-lingual, scientific component ‘plug-and-play’

IDE = Interactive Development Environment

CCA adds value to component development



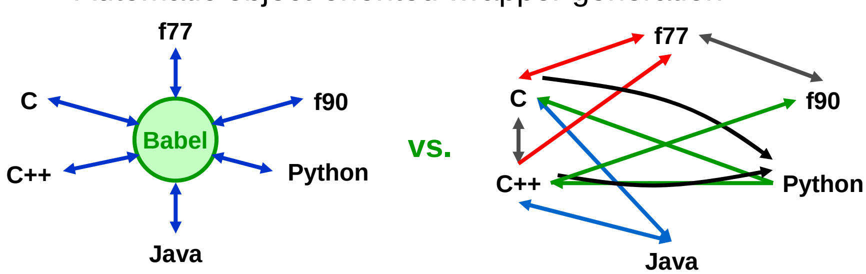
Tools Module Overview



- Language interoperability tools
- Frameworks
- CCA Interactive Development Environment

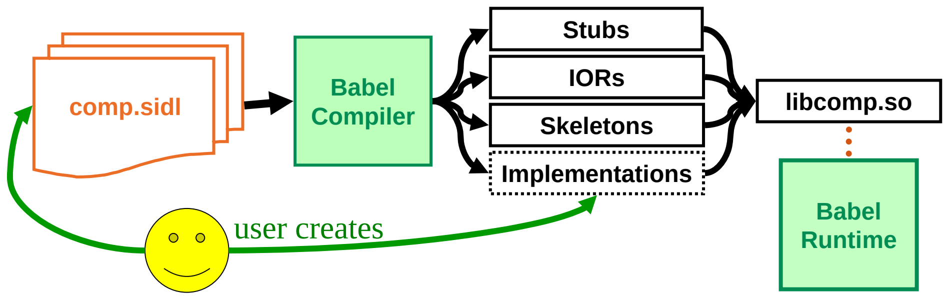
Babel Facilitates *Scientific* Programming Language Interoperability

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



Supported on Linux, AIX, and Solaris 2.7, works on OSX;
C (ANSI C), C++ (GCC), F77 (g77, Sun f77), F90 (Intel, Lahey, GNU, Absoft), Java (1.4)

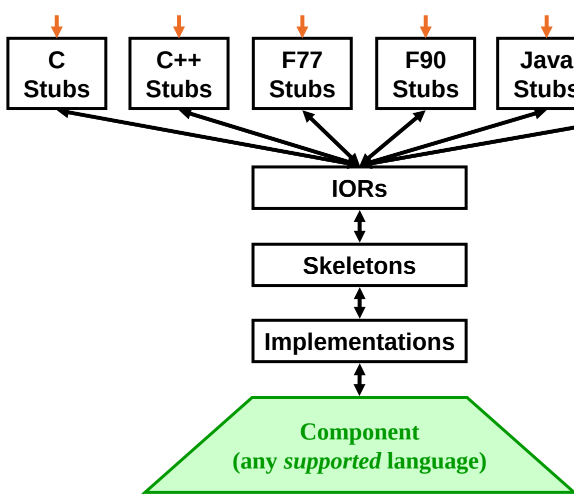
Babel Generates Object-Oriented Language Interoperability Middleware



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

IOR = Intermediate Object Representation SIDL = Scientific Interface Definition Language

Clients in any supported language can access components in any other language



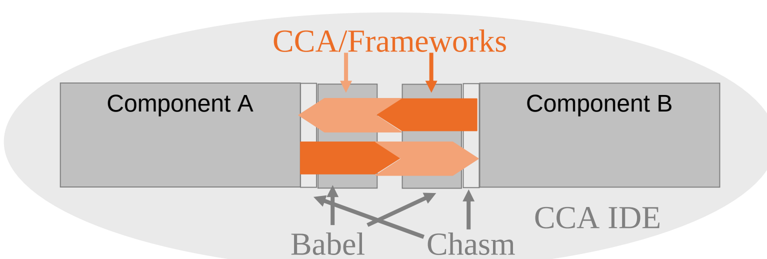
Babel presentation coming up!

IOR = Intermediate Object Representation

Recent and Upcoming Features

- Remote Method Invocation (Babel RMI)
  Kumfert, Leek, & Epperly IPDPS'07
  ALPHA – Babel 1.0. (aka "stable branch")*
  *BETA – Babel 1.1.**
  Final – Future Stable Branch (Babel 1.2?)
- Design-by-Contract
  Dahlgren CBSE'07 (to appear)
  ALPHA – Internal Babel branch
  BETA – Babel 1.1.2 (expected)
- Feature requests are accepted and tracked in the open using Roundup
  http://www.cca-forum.org/bugs/babel

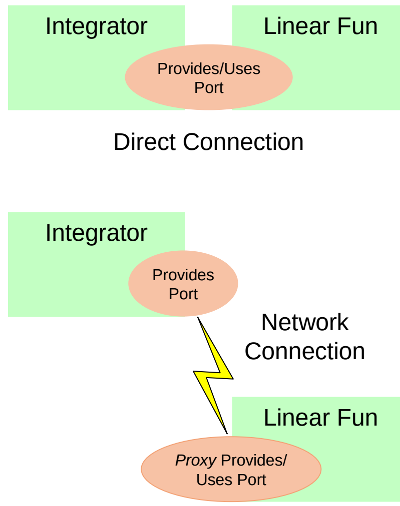
Tools Module Overview



- Language interoperability tools
- Frameworks
- CCA Interactive Development Environment

Frameworks are Specialized to Different Computing Environments

- “Direct connection” preserves high performance of local (“in-process”) and parallel components
  - Framework makes *connection*
  - Framework *not* involved in *invocation*
- Distributed computing has same uses/provides pattern, but the framework intervenes between user and provider
  - Framework provides a *proxy* port local to the user’s *uses* port
  - Framework conveys invocation from proxy to actual *provides* port



Graphical User Interfaces (GUIs) Deliver Plug-and-Play Experience

- Plug & play for:
  - Application software assembly
  - Visualization pipelines
  - Workflow management
- Assembling “wiring” diagrams is almost universal.
  - Software assembly: Ccaffeine, XCAT, SciRUN
  - Workflow: XCAT, SciRUN
  - Visualization: SciRUN

None of these (yet) plug into your favorite Integrated Development Environment (e.g., Eclipse, MS Dev. Studio, Java Studio, …).

Ccaffeine is a *Direct-Connect*, Parallel-Friendly Framework

- Supports SIDL/Babel components
  - Conforms to latest CCA specification (0.7)
  - Also supports legacy CCA specification (0.5)
    - Any C++ allowed with C and Fortran by C++ wrappers
- Provides command-line and GUI for composition
  - Scripting supports batch mode for SPMD
  - MPMD/SPMD custom drivers in any Babel language

Supported on Linux, AIX, OSX and is portable to modern UNIXes.

Ccaffeine Involves a Minimum of Three Steps

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

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

Optional

Ccaffeine GUI

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

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

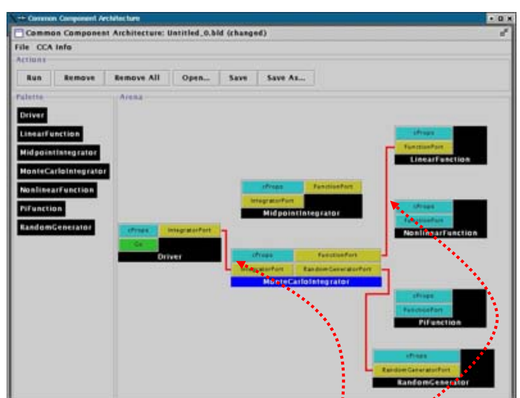
IntegratorPort | FunctionPort — FunctionPort

MidpointIntegrator — NonlinearFunction

Java is required.

User Connects Ports

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



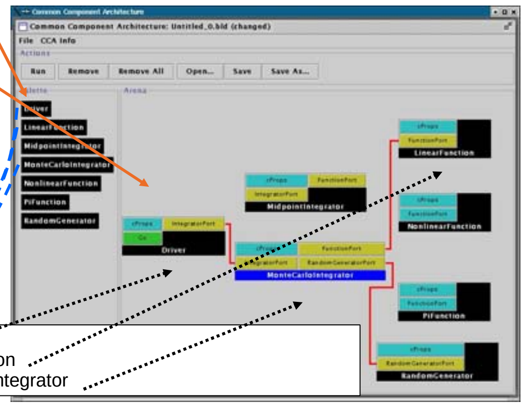
connect	Driver	IntegratorPort	MonteCarloIntegrator	IntegratorPort
connect	MonteCarloIntegrator	FunctionPort	LinearFunction	FunctionPort
…				

CCA
Common Component Architecture

Ccaffeine Framework (ccafe-gui)

Building an Application (1 of 2)

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



create	Driver	Driver
create	LinearFunction	LinearFunction
create	MonteCarloIntegrator	MonteCarloIntegrator

CCA
Common Component Architecture

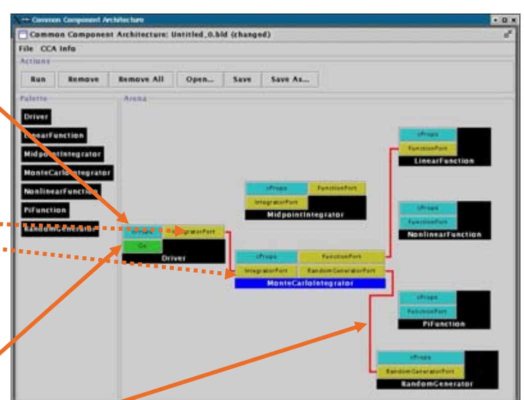
Ccaffeine Framework (ccafe-gui)

Building an Application (2 of 2)

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

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

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



Right-clicking a connection line breaks the connection -- enabling component substitution.

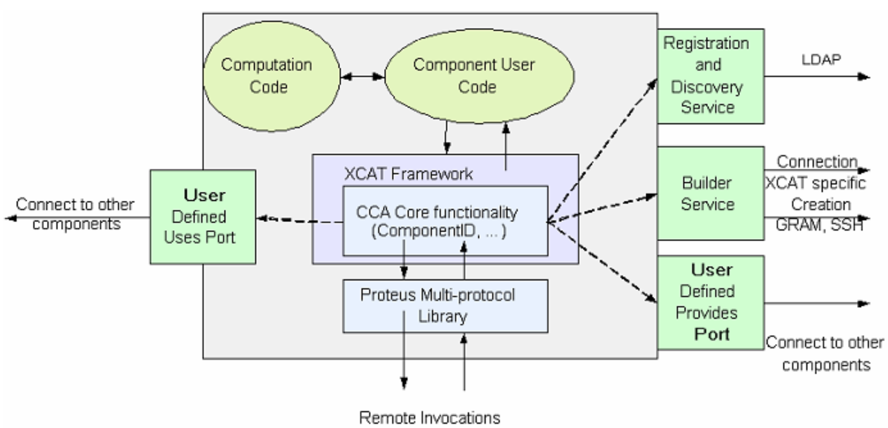
XCAT is a Web-services based *Distributed* Component Framework

- Remote references
  - Port types described in C++ header files or in WSDL documents
- User Interface
  - C++ and Python interface to CCA *BuilderService*
  - Uses SWIG for Python-C++ translations
- Component creation
  - Remote creation via SSH
- Component communication
  - Proteus multi-protocol library
  - Communication libraries can be loaded at run-time

Tested on Linux.

WSDL = Web Service Definition Language

XCAT is Designed for High-Performance Scientific Applications



Basic How-To

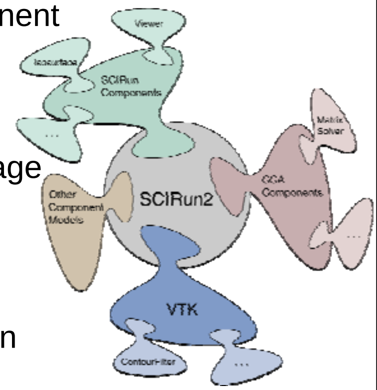
1. Define port interfaces as C++ header files or WSDL docs
2. Indicate ports used by each component in a config file
3. Run scripts to generate code for stub-skeletons (for ports)
   - Can also generate component-templates for new components
   - Use component-templates to convert a scientific library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

Recent and Upcoming Features

- Incorporate Babel's Remote Method Invocation
  - Allows access to Babel objects through remote Babel stubs
  - Provides direct support for SIDL in distributed applications
- Incorporate Web-services based protocols in BabelRMI
- Performance analysis of Babel-RMI based distributed communication
- Design Distributed Framework Interoperability Specification

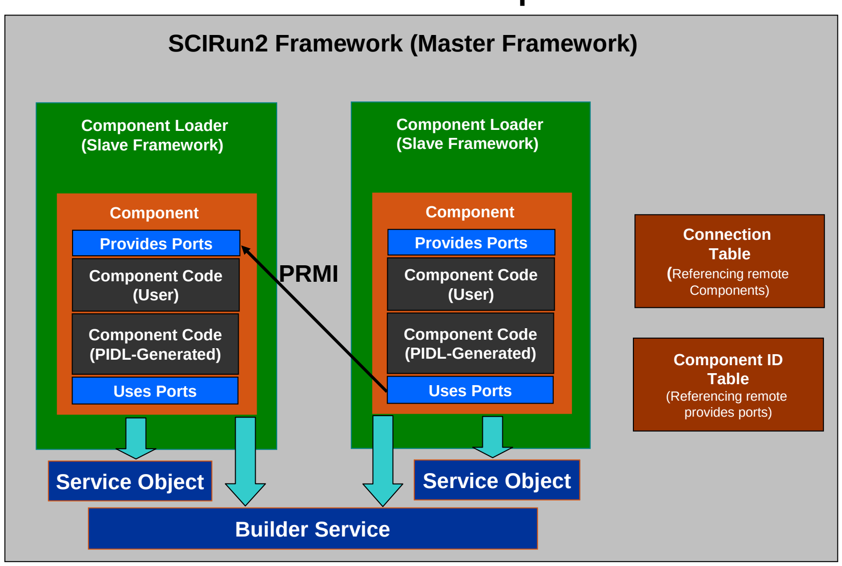
SCIRun2 is a Cross-Component Model, *Distributed* Component Framework

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



Supported on Linux.

Master Framework Coordinates *Slave Frameworks* in *Each* Remote Address Space



CCA
Common Component Architecture

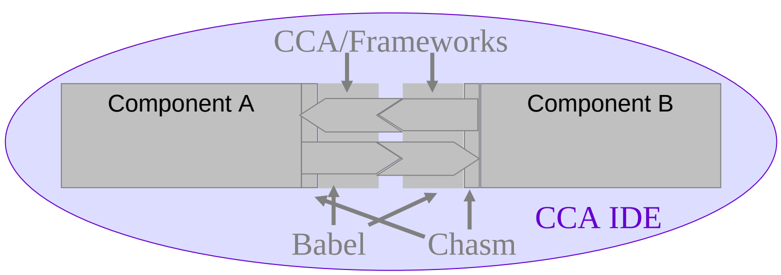
SCIRun2 Framework

Basic How-To

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

CCA
Common Component Architecture

Tools Module Overview



- Language interoperability tools
- Frameworks
- CCA Development Environment

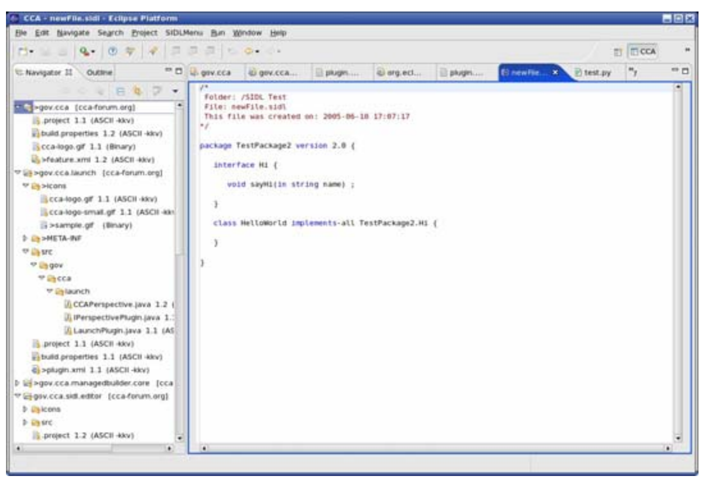
Component Development Environment Provided via Eclipse Plug-ins

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

Supported on Linux, Windows, MacOS.

Component Development Environment Starts at the Eclipse Platform Level

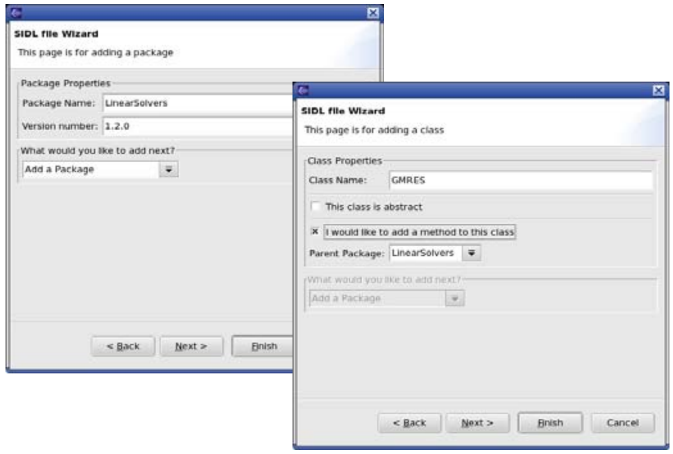
Plug-ins for:
- SIDL Editor
- Wizards
- Preliminary automated build support



Imperative that you start by creating a new project!

Wizards are Available for Adding Packages and Classes or Generating SIDL from Legacy Codes

- Intuitive interfaces to port and component definition
- Helper wizards for setting port, component and (in the future) application properties



Bocca Creates “Scaffolding” for an entire CCA application

- Including ports
  - Give the typename and an empty port is created
- Including components
  - Give the name and an empty component is created
  - With extra parameters the component uses/provides a port
- Including build system
  - For all ports/components in the project
  - Implemented in any CCA supported language
- Including application composition
  - Link components to create application with Ccaffeine GUI

CCA
Common Component Architecture

Bocca is an Application Generator

- Purely command-line driven
  - Generates the glue code that enables components and language interoperability
  - Enables rapid generation of an entire application and build system in minutes
  - Good for rapid prototyping and novice users…
    - But is meant to carry through to full project life-cycle
- Different and complementary to an IDE
  - Does not even have an editor for creating code (choose your own)

CCA
Common Component Architecture

Bocca Demo

CCA Tools Contacts (1 of 2)

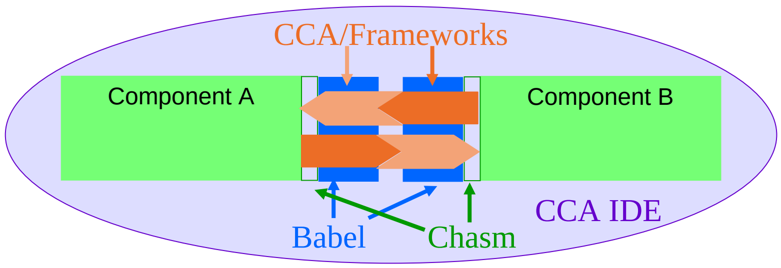
Tool	Purpose	More information
Babel	Scientific language interoperability tool kit	URL: www.llnl.gov/CASC/components Email: components@llnl.gov or babel-users@lists.llnl.gov
Ccaffeine	Direct-connect, parallel-friendly framework	URL: www.cca-forum.org/software/ Email: Ben Allan, ccafe-help@z.ca.sandia.gov Wiki: https://www.cca-forum.org/wiki
CCA IDE	CCA development environment	Email: usability@cca-forum.org

CCA Tools Contacts (2 of 2)

Tool	Purpose	More information
SCIRun2	Cross-component model framework	URL: www.sci.utah.edu/ Email: Steve Parker, sparker@cs.utah.edu
XCAT-C++	Globus-based GRID framework	URL: grid.cs.binghamton.edu/projects/xcat/ Email: Madhu Govindaraju, mgovinda@cs.binghamton.edu
Bocca	Development environment	Email: bocca-dev@cca-forum.org

CCA
Common Component Architecture

Module Summary



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

CCA
Common Component Architecture

Boneyard Slides

Experimental Frameworks

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

OGSI = Open Grid Services Infrastructure

Chasm-Assisted Glue Code Generation

1. Create functions_LinearFunction.impl.xml
2. Create xml description of source procedures

```
% gfortran -fdump-parse-tree LinearFunction.f90 >
  LinearFunction.ptd
% gfortran2xml < LinearFunction.ptd >
  LinearFunction.xml
```

3. Create .sidl, _Impl.F90, and _Mod.F90

```
% xalan –o functions_LinearFunction.sidl
  functions_LinearFunction.impl.xml
  cca-f90-comp.sidl.xsl
% xalan –o functions_LinearFunction_Impl.F90
  functions_LinearFunction.impl.xml
  cca-f90.impl.xsl
% xalan –o functions_LinearFunction_Mod.F90
  functions_LinearFunction.impl.xml
  cca-f90.mod.xsl
```

4. Run Babel…

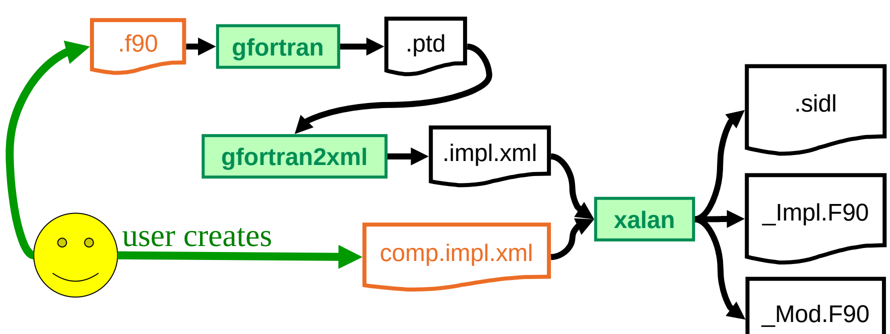
User-Created XML Component Description File

```
<componentImplementation name="LinearFunction” package="functions”>
 <language name=“F90”>
  <property name="impl-scope" value="LinearFunction"/>
  <property name="impl-xml” value=”/home/cca/LinearFunction.xml"/>

  <ports>
   <provides name="FunctionPort" package="function">
   <MethodsBlock>
    <Method name="evaluate" impl="evaluate_lf"/>
   </MethodsBlock>
   </provides>
  </ports>

 </language>
</componentImplementation>
```


The Entire Chasm Process Involves Three Basic Steps



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

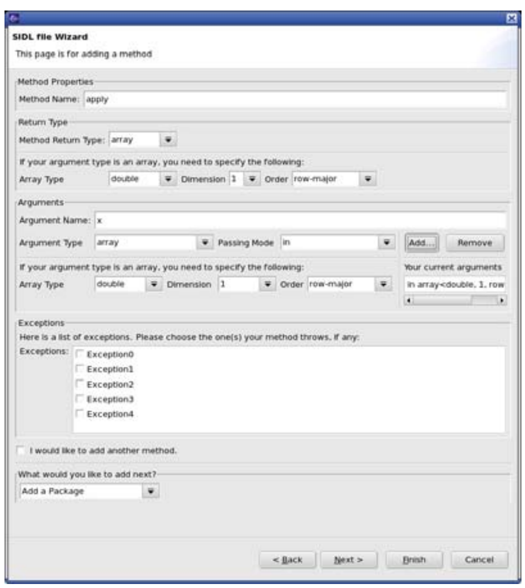
XML = Extensible Markup Language PTD = Parse Tree Dump

Chasm Provides Language Interoperability for Fortran, C, and C++

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

Supported on Linux, AIX, and Solaris 2.7, works on OSX;
C (ANSI C), C++ (GCC), F90 (Intel, Lahey, GNU, Absoft)

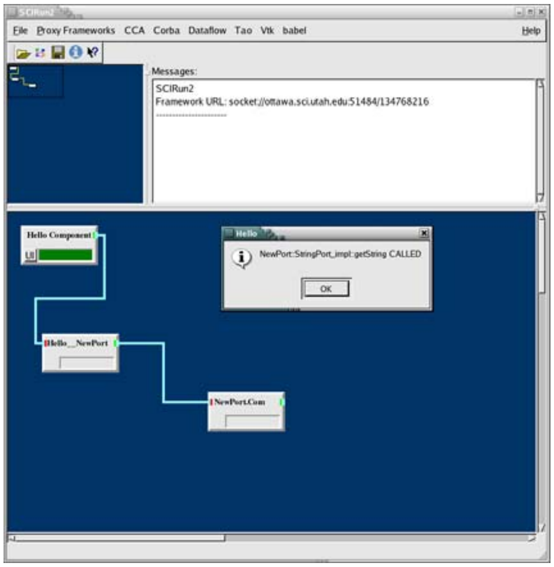
A Wizard is also Available for Adding Methods



Recent and Upcoming Features

- Provide automated build support *1H 2005*
- Launch application via GUI *1H 2006*

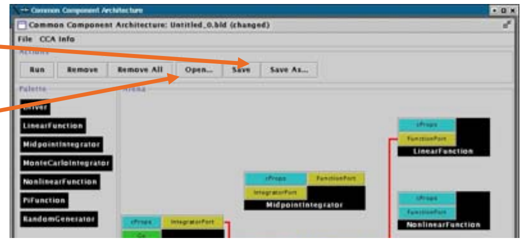
Simple SCIRun2 CCA (PIDL) and Babel Bridge



Application Configurations can be Re-used

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

2. Click *Open* to replay actions.



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

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

- C++ stand-alone execution
  % **bld2babel-cpp** *faster_file.bld faster_file_babel outdir*

  or % **bld2neo** *faster_file.bld* faster_file.batch *outdir*

BABEL

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

CCA
Common Component Architecture

Goal of This Module

Expose Attendees to How Existing Code is

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

In the process, will also cover

- **Babel Basics**
- **Scientific Interface Definition Language (SIDL)**
- **Code sample ("Hello World!")**

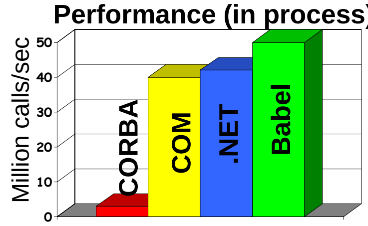
CCA
Common Component Architecture

BABEL is Middleware for HPC

R&D 100
2006

"The world's most rapid communication among many programming languages in a single application."

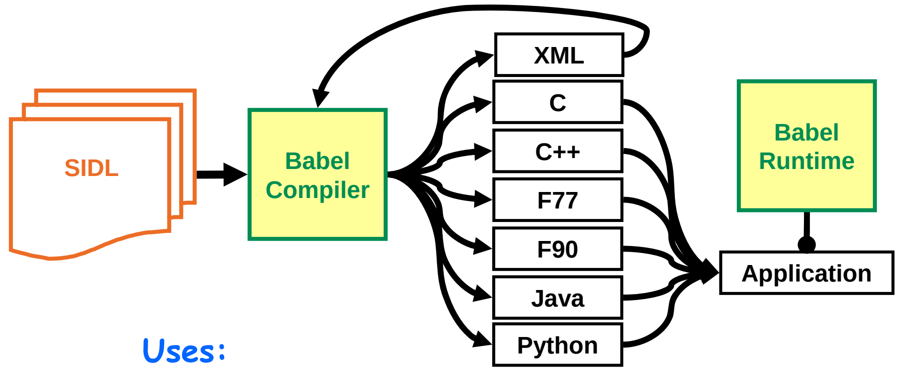
Performance (in process)



	CORBA	COM	.NET	Babel
BlueGene, Cray, Linux, AIX, & OS X	No	No	No	Yes*
Fortran	No	Limited	Limited	Yes
Multi-Dim Arrays	No	No	No	Yes
Complex Numbers	No	No	No	Yes
Licensing	Vendor Specific	Closed Source	Closed Source	Open Source

* BlueGene port expected mid 2007

Babel Supports a Unified Object-Oriented Type System Spanning Many Languages



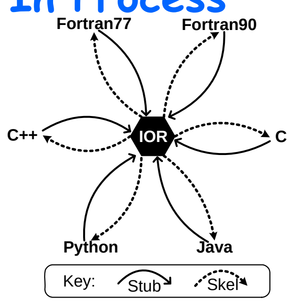
Uses:

- Stop hand-writing language wrappers!
- Basis for component frameworks
- Add polymorphism in single languages
- Generate language-neutral API specifications
- Turn Python to an interactive debugger

Recently, Babel Added Distributed Capabilities (Remote Method Invocation)

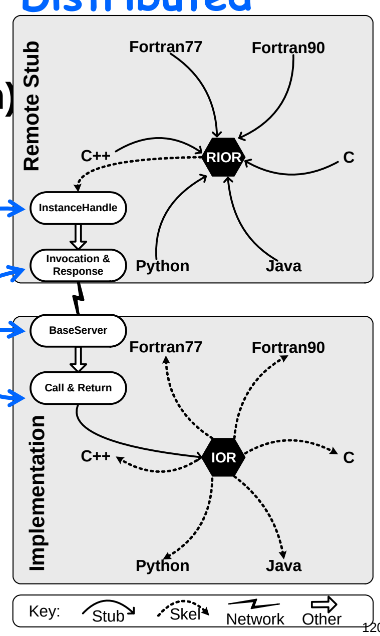
CCA is still experimenting with RMI, so I'll focus my talk on the in-process case.

In Process



Network Layer defined in SIDL for 3rd party plug-ins

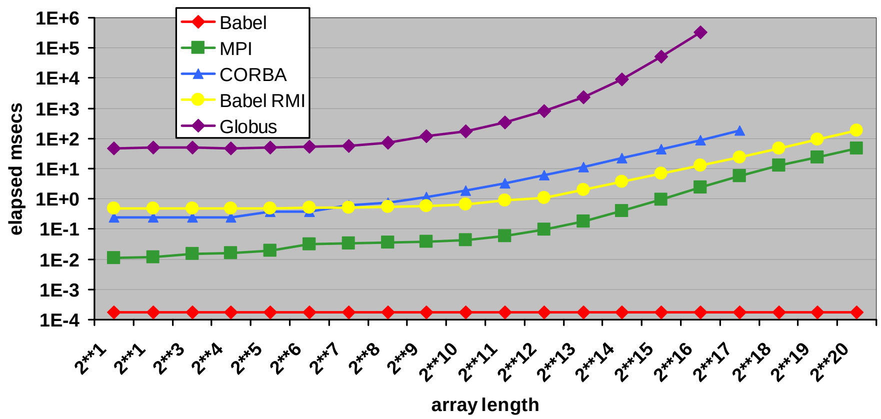
Distributed



CCA Common Component Architecture

Simple Scaling Study using Babel RMI's Reference Impl.

inout array<double>



CCA Common Component Architecture

Working Code: "Hello World" in F90 Using a Babel Type

```
  program helloclient
①   use greetings_English
②   use sidl_BaseInterface
    implicit none
③   type(greetings_English_t) :: obj
④   type(sidl_BaseInterface_t):: exc
    character (len=80)        :: msg
    character (len=20)        :: name
⑤   name='World'
⑥   call new( obj, exc )
⑦   call setName( obj, name, exc )
⑧   call sayIt( obj, msg, exc )
⑨   call deleteRef( obj, exc )
    print *, msg
  end program helloclient
```

CCA
Common Component Architecture

Handout Material: Code Notes

① Use statement for the greetings.English type
② Use statement for the sidl.BaseInterface type
③ Obj is a F90 derived type we get from the using statement, note the "_t" extension that prevents it from colliding with the using statement.
④ Exc is used to hold exceptions thrown by methods
⑤ In C/C++ examples, this variable would be initialized by a the command-line variable "argv[1]", but its trickier to do portably in F90 and too long, so I just initialize the name to "World".
⑥ Obj is not yet initialized. The Babel idiom in F90 is to call new() to initialize the Babel type. In other languages its _create(). NOTE: good code would add error checking.
⑦ setName() puts data into the obj. It sets its state.
⑧ sayIt() returns the entire greeting including the aforementioned name.
⑨ deleteRef() is a subroutine that all Babel types inherit from a parent class. All Babel objects are reference counted. When there are no more outstanding references, the object is told to clean up after itself.

CCA
Common Component Architecture

Working Code: "Hello World" in F90 Using a Babel Type

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

Looks like a native F90 derived type

These subroutines were specified in the SIDL.

Other basic subroutines are "built in" to all Babel types.

CCA
Common Component Architecture

Question: What language is "obj" really implemented in?

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

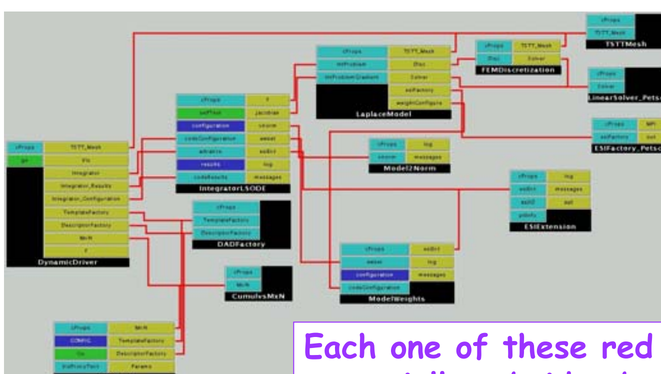
Answer: Can't Know!

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

In fact, it could change on different runs without recompiling this code!

CCA
Common Component Architecture

CCA uses Babel for high-performance n-way language interoperabilty



Each one of these red lines, is potentially a bridge between two languages. No telling which language your component will be connected to when you write it.

CCA
Common Component Architecture

The SIDL File that defines the “greetings.English” type

```
①② package greetings version 1.0 {
③④     interface Hello {
⑤         void setName( in string name );
          string sayIt ( );
      }
⑥     class English implements-all Hello {  }
   }
```


CCA
Common Component Architecture

Handout Material: Code Notes

① Packages contain user-defined types and are used to reduce naming collisions. Packages can be nested.

② Packages can be versioned. User defined types must be nested inside a versioned package and gain the same version number as the innermost versioned package

③ SIDL has a inheritance model similar to Java and Objective C. Classes can inherit multiple interfaces, but at most one implementation (other class).

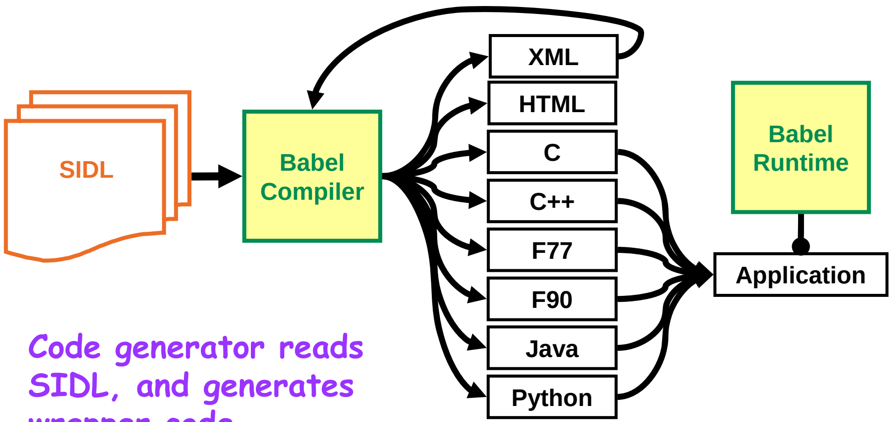
④ An interface describes an API, but doesn’t name the implementation.

⑤ Note that arguments have mode, type, and name. Mode can be one of “in”, “out”, and “inout”. These SIDL modes have slightly different semantics than Fortran90 “intents”.

⑥ This class generates English greetings. One could imagine a strategy for internationalization that uses the Hello interface everywhere, but loads in English, French, or whatever classes based on user’s preference.

CCA
Common Component Architecture

Babel Consists of Two Parts: Code Generator + Runtime Library

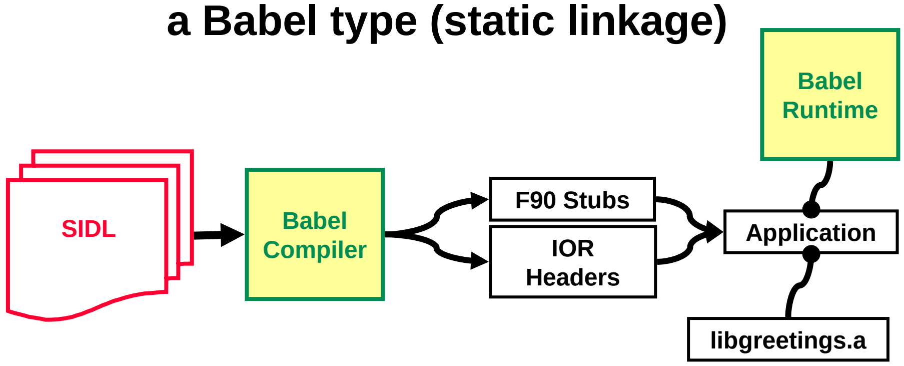


Code generator reads SIDL, and generates wrapper code…

… very sophisticated wrapper code.

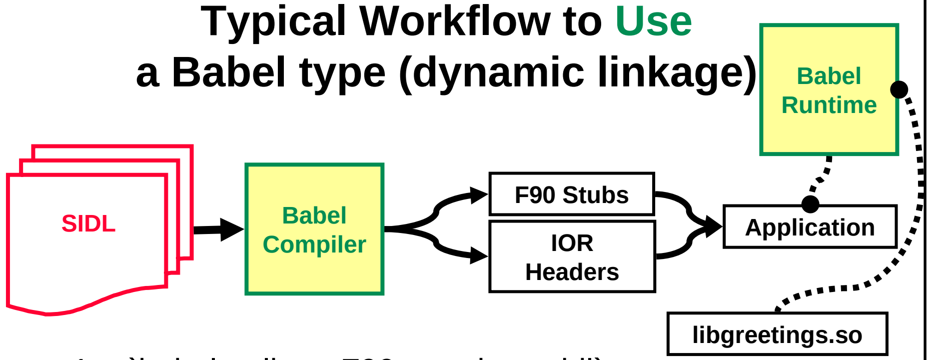
CCA
Common Component Architecture

Typical Workflow to Use a Babel type (static linkage)



1. `babel --client=F90 greetings.sidl`
2. Compile generated Code
3. Link driver, generated code ("Stubs"), Babel Runtime, and library containing Babel type

Typical Workflow to Use a Babel type (dynamic linkage)

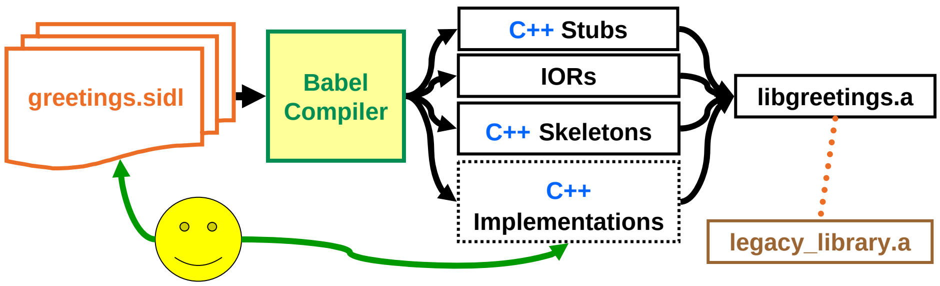


1. \`babel --client=F90 greetings.sidl\`
2. Compile same generated code with different flags
3. Link driver, and stubs only (both generated code and F90 stubs to Babel Runtime library)
4. Set SIDL_DLL_PATH environment variable to include relevant *.scl (or *.cca) files.
5. Actual implementations are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

Static vs. Dynamic Linkage

- Static
  - Least runtime overhead
  - Easiest to get right, debug, etc.
- Dynamic
  - Allows new types to “plug-in” without relinking driver
  - Necessary for Java or Python calling to other languages (unless you relink their virtual machine)
  - Induces very nondeterministic behavior if done incorrectly

Workflow for a Developer Wrapping Their Code into Babel Objects



1. `babel --server=C++ greetings.sidl`
2. Fill in the implementation details (see next slide)
3. Compile and link into a library/DLL

Implementation Details Must be Filled in Between Splicer Blocks

```
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)
}
```

CCA Common Component Architecture

Quick Review of Babel in general before proceeding to CCA specifics

- Babel can be used as a standalone tool
- Each language binding strikes a balance
  - support the SIDL type system (OO, exceptions, etc.)
  - provide it in a manner "natural" to experienced programmers in the target language
- For more details about Babel and SIDL
  - SC|04 tutorial slides for Babel http://www.llnl.gov/CASC/components/docs/sc04.html
  - Babel User's Guide (aka. the BUG) http://www.llnl.gov/CASC/components/docs/users_guide/

135

CCA Common Component Architecture

CCA uses SIDL to specify APIs and Type Hierarchy for Frameworks, Services, Components, & Ports

- **A CCA framework must**
  - **implement `gov.cca.AbstractFramework`,**
  - **provide a `gov.cca.ports.BuilderService`,**
  - **etc.**
- **A CCA port must**
  - **be a SIDL interface extending `gov.cca.Port`**
- **A CCA component must**
  - **be a SIDL class implementing `gov.cca.Component`**

The CCA Specification is a SIDL file.

136

CCA
Common Component Architecture

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 );
    }
}
```


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 (and any provided ports)

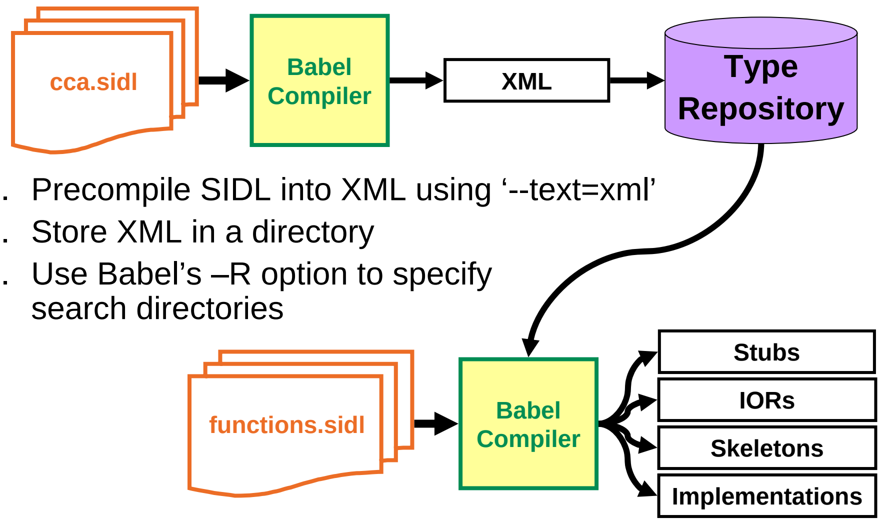
```
class LinearFunction implements functions.Function,
                                gov.cca.Component {
      double evaluate( in double x );
      void setServices( in cca.Services svcs );
}
```

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

CCA
Common Component Architecture

Supplementary material for notes

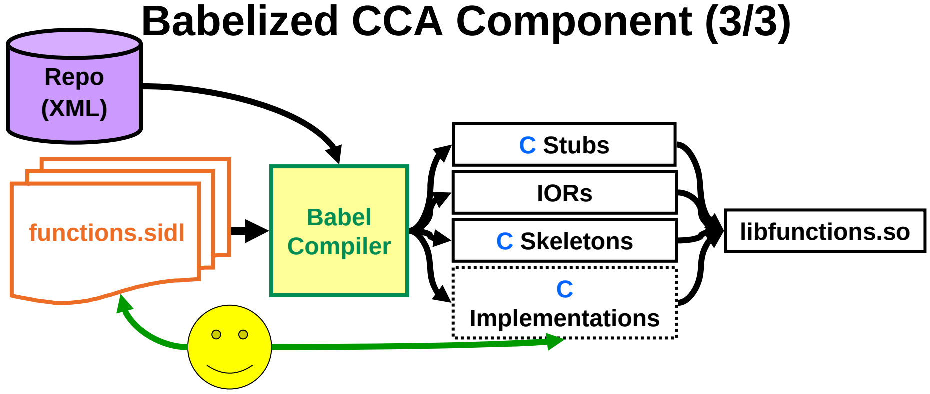
Tip: Use Babel's XML output like precompiled headers in C++



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

CCA
Common Component Architecture

How to write a Babelized CCA Component (3/3)



3. Use Babel to generate the interoperability glue
   – Execute `babel --server=C –R*Repo functions.sidl*`
4. Fill in Implementations as needed

CCA Common Component Architecture

Review: Goal of This Module

Learn how existing code is

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

In the process, will also covered

- **Babel Basics**
- **Scientific Interface Definition Language (SIDL)**
- **Code sample ("Hello World!")**

CCA Common Component Architecture

Contact Information

- Project: http://www.llnl.gov/CASC/components
- Project Team Email: components@llnl.gov
- Mailing Lists: majordomo@lists.llnl.gov
  subscribe babel-users *[email address]*
  subscribe babel-announce *[email address]*
- Bug Tracking: https://www.cca-forum.org/bugs/babel/
  or email to babel-bugs@cca-forum.org

CCA
Common Component Architecture

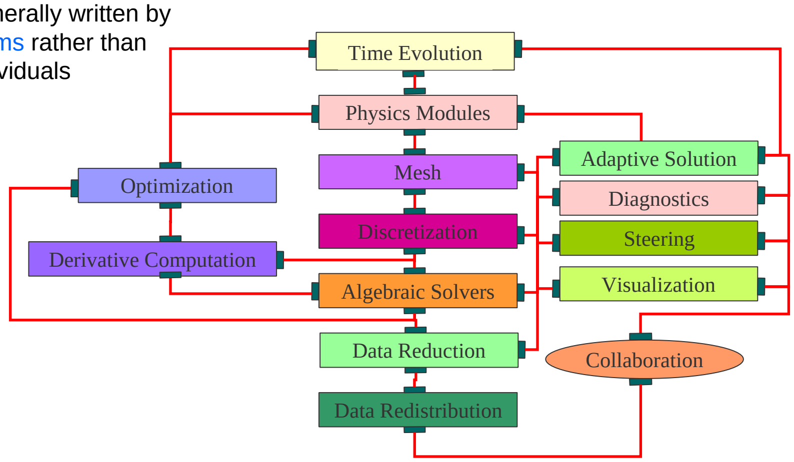
Using CCA: Approaches & Experience

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

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



Using CCA to Help Manage Complexity

- Application areas participating in the CCA:
  astronomy, astrophysics, biological and medical simulations, chemically reacting flow, climate and weather modelling, combustion, computational chemistry, data management, fusion and plasma physics modelling, linear algebra, materials science, molecular electronics, nanoscience, nuclear power plant simulations, structured adaptive meshes, unstructured meshes, and visualization
- Research agencies sponsoring software development using the CCA:
  DOE (SciDAC, Office of Science, NNSA/ASC), NASA, NIH, NSF, DoD, European Union

Outline

- Developing Components
  - Strategies for both developing new codes and wrapping existing codes
- Case Studies
  - Chemistry project
    - Moderate-sized multi-disciplinary, multi-institutional team
    - Using Ccaffeine framework, SIDL components
  - Combustion toolkit
    - Small team, both new and wrapped codes
    - Using Ccaffeine framework, C++ components

CCA
Common Component Architecture

Developing Components
(Both New Codes and Wrappers to Existing Codes)

- Productivity Benefits
- Application Decomposition Strategies
- Interface Design Issues
  - Social factors
  - Technical factors
- Implementation Issues and Patterns

CCA
Common Component Architecture

CCA Productivity Benefits

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

CCA
Common Component Architecture

Components in the Small: Impacts within a Project

Benefits include:

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

CCA
Common Component Architecture

Components in the Large: Connecting Multiple Projects

Benefits include:

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

Costs include:

- Consensus can be expensive to obtain
- Writing code for others to use is more difficult than writing it just for yourself

CCA Common Component Architecture

Application Decomposition Strategies

- Conceptually decompose the application into
  - cutting-edge areas (less stable) *and*
  - areas that can employ existing component-based libraries (more stable)
- Decompose each area into components for
  - physics
  - mathematics
  - data management

  as dictated by the application; sketch a typical component layout
- Many components will encapsulate algorithmic logic only, with little or no private data
- Most HPC applications will have a central data abstraction that provides data memory management and parallel communication
- In a multilanguage application, all I/O *may* need to be isolated into components written in a single common language (file based I/O should not be affected)
- Component boundaries (and port interfaces) may be set to isolate proprietary code or difficult contributors

CCA Common Component Architecture

Interface Design: Social Factors
(Defining Ports to Wrap Existing Code)

- Will the port hide just one implementation, or will there need to be plug compatibility with other implementations? From other teams?
- Who defines the interface and maintains it?
  1. Project dictator? (*Fast…*)
  2. The owner of the legacy functionality? (*Slow, if not you…*)
  3. A standards committee? (*Really slow…*)
- How many iterations of redefining the ports will the customers tolerate?

CCA
Common Component Architecture

Interface Design: Technical Factors

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

CCA
Common Component Architecture

Implementation Issues in Wrapping

- Do we split large libraries into several components?
  - Splitting is difficult to do if global variables or common blocks are widely used.
- Do we expect more than one implementation instance of a port in a single run-time?
  - If not, interface contracts may include global side effects
- Do we integrate the wrapper code in the existing code's development and build processes?
  - If not, how do we ensure build consistency and on-going wrapper support?
- Code bases with large interfaces need automated wrapping tools
  - E.g., see Chasm info in the Tools module of the tutorial

CCA
Common Component Architecture

Benefits of Wrapping Code Using CCA

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

CCA
Common Component Architecture

Interface Design for *New* Code

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

CCA Common Component Architecture

Recommended Implementation Patterns

- Expect to decompose initial components further as work progresses and requirements expand
- Build systems (i.e. `make`) should be kept as simple as possible
  - Keep a subdirectory for port definitions and any implementation-independent glue code derived from the ports
  - Keep each component (and any wrapped code) in its own subdirectory
  - Keep application-wide flags in a configure script or an include file common to all components and ports
  - Consistency is key. Extract build flags from `cca-spec-babel-config` and if possible compile & link with `babel-libtool`

CCA Common Component Architecture

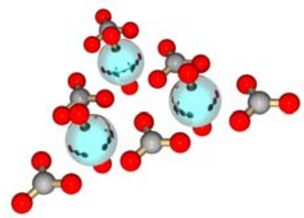
Outline

- Developing Components
  - Strategies for both developing new codes and wrapping existing codes
- Case Studies
  - Chemistry project
    - Moderate-sized multi-disciplinary, multi-institutional team
    - Using Ccaffeine framework, SIDL components
  - Combustion toolkit
    - Small team, both new and wrapped codes
    - Using Ccaffeine framework, C++ components

CCA
Common Component Architecture

Case Study 1: Chemistry Project

- Funded via SciDAC initiative
- Initial focus: Full-featured components for structure optimization
  - **Chemistry models** provided by MPQC (SNL) & NWChem (PNNL)
  - **Numerical optimization** provided by TAO (ANL) solvers
  - **Linear algebra** provided by GA (PNNL) and PETSc (ANL)
- Recent work:
  - Multi-level parallelism
  - Low-level chemistry model integration (e.g., molecular integrals)



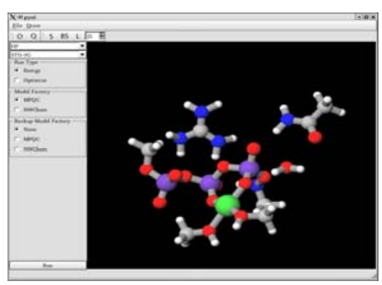
CCA
Common Component Architecture

CCA-Chemistry Project Participants

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

Sandia National Laboratory
- Curtis L. Janssen
- Joseph P. Kenney

Argonne National Laboratory
- Steve Benson
- Lois Curfman McInnes
- Jason Sarich



Pacific Northwest National Laboratory
Operated by Battelle for the U.S. Department of Energy

Argonne National Laboratory, University of Chicago

Sandia National Laboratories

CCA
Common Component Architecture

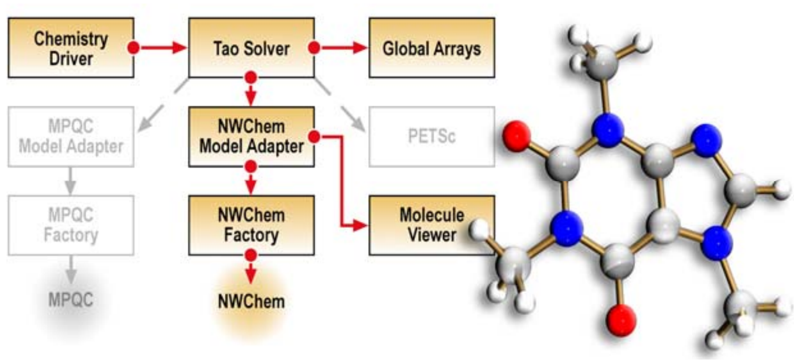
CCA Impacts in Computational Chemistry

Through 4 chemistry applications we consider different facets of CCA functionality:

- **Molecular Geometry Optimization**
  - Combining diverse expertise of 5 different research groups
- **Lennard-Jones Molecular Dynamics**
  - Achieving good scalability and low CCA overhead
- **Multi-level Parallelism in Computational Chemistry**
  - Combining SPMD and MPMD parallelism
- **Molecular Integral Evaluation**
  - Component interoperability at deeper levels within chemistry libraries

CCA
Common Component Architecture

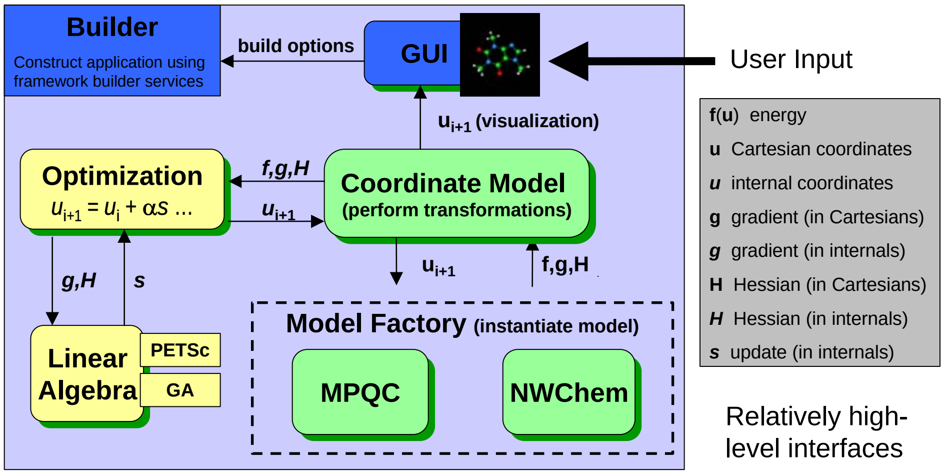
App 1: Molecular Geometry Optimization:
Combining Diverse Expertise



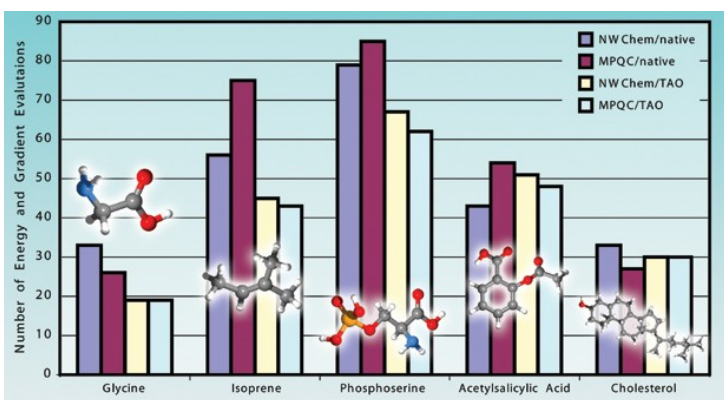
CCA quantum chemistry application using components based on:
- MPQC, NWChem (chemistry – energy evaluation)
- GA, PETSc (parallel data management and linear algebra)
- TAO (numerical optimization)

Molecular Geometry Optimization

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



Performance Gains with Better Optimization Algorithms

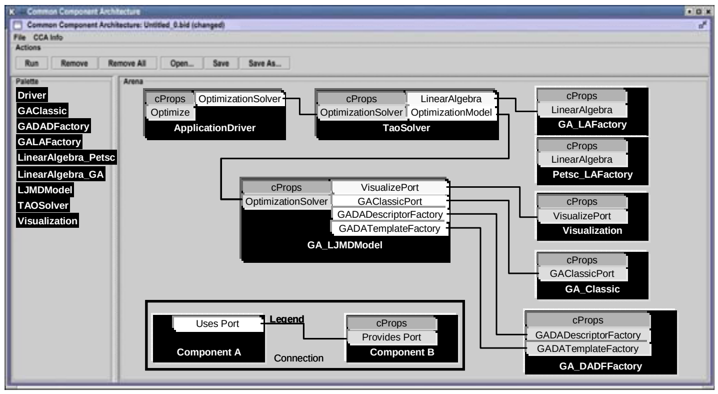


CCA-based integration of state of the art optimization algorithms from the TAO toolkit provides up to 40% improvement in the time required to optimize molecular structures with MPQC and NWChem.[1]

[1]Journal of Computational Chemistry 25: 1717-1725, 2004.

CCA
Common Component Architecture

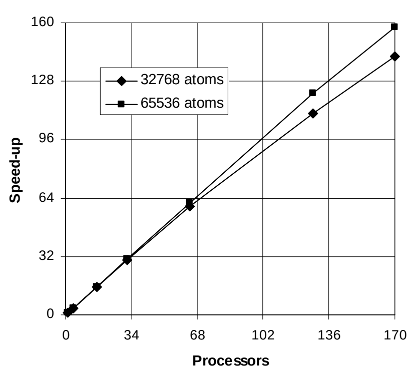
App 2: Molecular Dynamics



CCA component wiring diagram for Lennard-Jones energy optimization, which integrates TAO, GA and Lennard-Jones molecular dynamics components

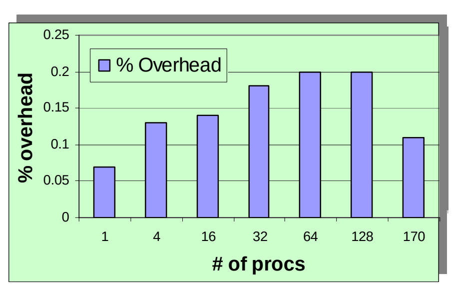
CCA
Common Component Architecture

Good CCA Scalability, Negligible Overhead



Speedup in the Lennard-Jones potential energy optimization for 32,768 and 65,536 particles

Overhead of CCA component (using SIDL) vs. “native” C++ implementation of a parallel Lennard-Jones molecular dynamics simulation

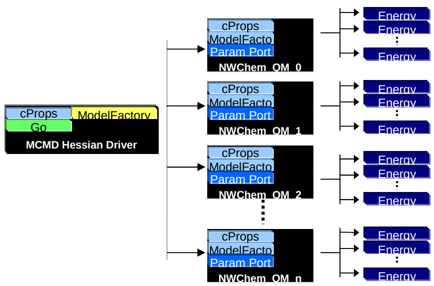


% Overhead using CCA Components

CCA Common Component Architecture

App 3: Multilevel Parallelism Using CCA

- Massive numbers of CPUs in future systems require leading edge tools to exploit all available parallelism
- GA helps exploit multi-level parallelism
- Multi-level parallelism using CCA & GA processor groups: Combining SCMD (Single Component Multiple Data) and MCMD (Multiple Component Multiple Data) paradigms

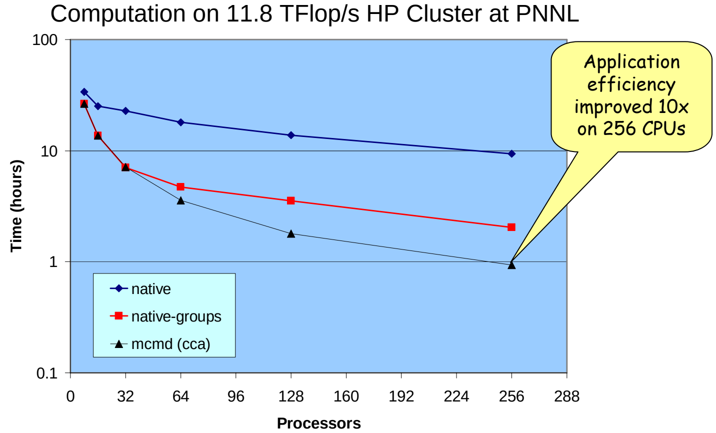


The MCMD Driver launches multiple instances of NWChem components on subsets of processors (also assigns a GA communicator for every instance). Each NWChem QM component does multiple energy computations on subgroups.

CCA Common Component Architecture

Multilevel Parallelism - MCMD

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



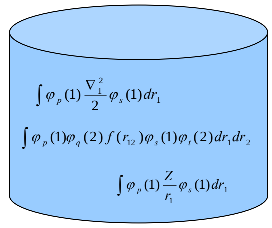
M. Krishnan, Y. Alexeev, T. Windus, and J. Nieplocha, Multilevel Parallelism in Computational Chemistry using the Common Component Architecture and Global Arrays, Proceedings of the IEEE/ACM SC2005 Conference.

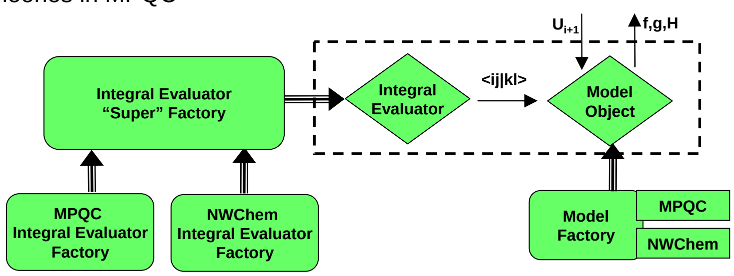
CCA
Common Component Architecture

App 4: Molecular Integrals:

Deeper Levels of Interoperability among Chemistry Packages

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





CCA
Common Component Architecture

CCA Impacts in Computational Chemistry: Review

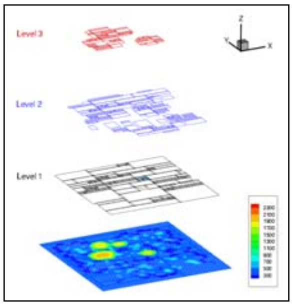
Through 4 chemistry applications we considered different facets of CCA functionality:

- Combining diverse expertise of 5 different research groups
- Achieving good scalability and low CCA overhead
- Implementing multi-level parallelism by combining SPMD and MPMD paradigms
- Addressing component interoperability at deeper levels within chemistry libraries

CCA Common Component Architecture

Case Study 2: Combustion Project

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



J. Ray, S. Lefantzi, J. Lee, C. Kennedy, W. Ashurst, K. Smith, M. Liu, N. Trebon

CCA Common Component Architecture

Why Use CCA in the CFRFS Toolkit?

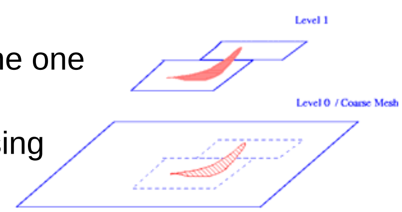
- Separate clearly the physics models, numerical algorithms, and the “CS” parts of the toolkit
  - Strictly functional!
- Realize the separation in software
- Tame *software* complexity
- Separate contributions by transient contributors
  - Form the bulk of the developers
- Create “chunks” of well-defined functionality that can be developed by experts (usually numerical analysts and combustion researchers)

Design Principles of the Toolkit / 1

- **Principal Aim: Reduce software complexity**
  – We can deal with the rest
- Functional decomposition into components
  – “Data Object” and Mesh components
  – (Large) set of numerical algorithmic components (integrators, linear/nonlinear solvers, etc.)
  – (Large) set of physical models components (gas-phase combustion chemistry, thermodynamics, fluid dynamic quantities, e.g. viscous stress tensor)
  – Handful of adaptors

Design Principles of the Toolkit / 2

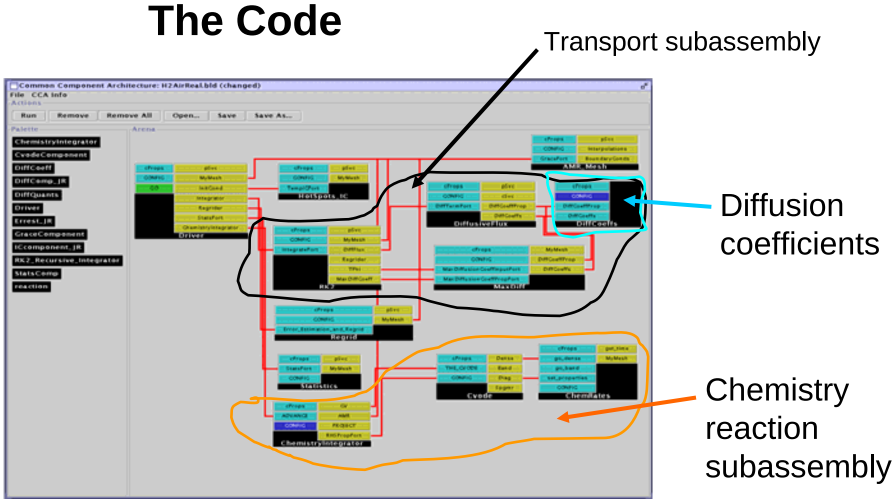
- Decomposition reflected in the port design and implementation
  – Most re-implemented port is the one that exchanges a rectangular sub-domain’s data for processing by components



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

CCA Common Component Architecture

The Code



Separate component subsystems for transport (black) and for reaction (orange) in a reaction-diffusion code. They two are coupled at a relatively high level.

CCA Common Component Architecture

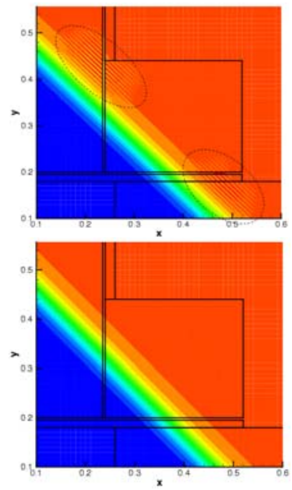
Has the Toolkit Approach Helped Tame Software Complexity?

- How has the code evolved?
  - How often have new ports been added?
  - How many rewrites have been done?
- How large are the components?
- How many ports do they have?
  - How large are the ports?
- How many ports exist?
  - i.e., Is the design general enough to support many implementations?
- What is the connectivity of components in application codes?

CCA Common Component Architecture

CFRFS Toolkit Status

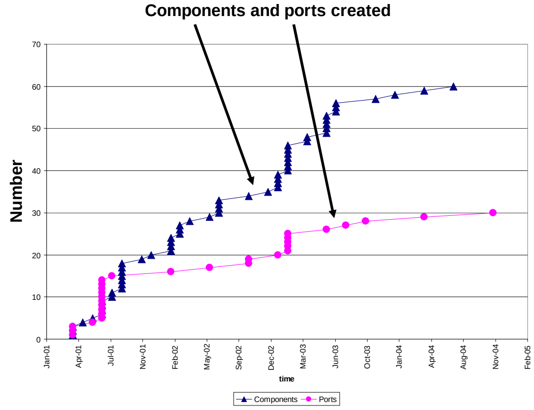
- Started in 2001
- 61 components today, all peers, independent, mixed and matched for combustion and shock hydrodynamics
- 7 external libraries
- Contributors: 9 in all, including 3 summer students
- Only 2 of the 9 contributors are at SNL today



A Fitzhugh-Nagumo equation being solved on a block-structured adaptively refined mesh. The top image illustrates Runge phenomena at coarse-fine interfaces (dashed ovals) when using high-order schemes (6^{th} order interpolations with 4^{th} order discretizations). Filtering them with an 8^{th} order filter removes them (bottom).

CCA Common Component Architecture

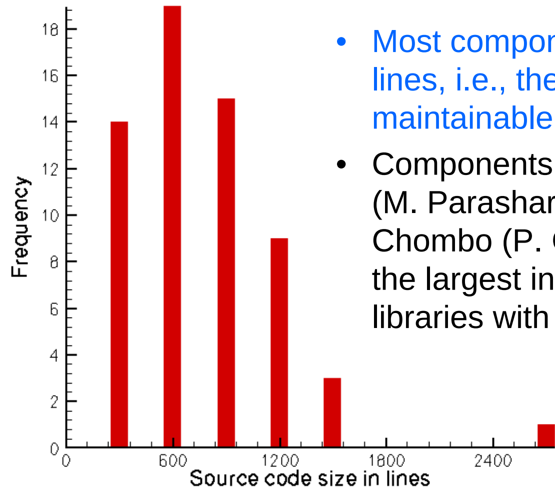
Scalability: Capability Growth without Rewrites



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

CCA Common Component Architecture

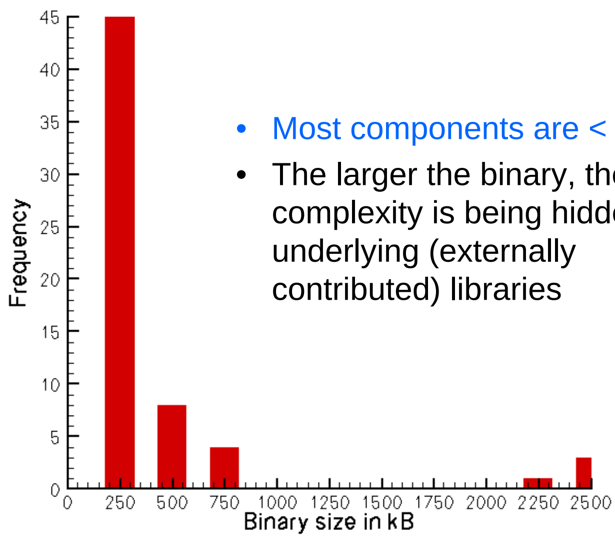
Taming Complexity: Lines of Code



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

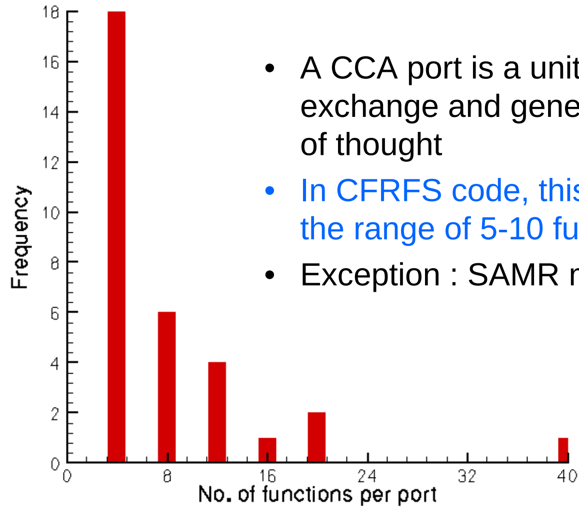
CCA Common Component Architecture

Taming Complexity: Code Size



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

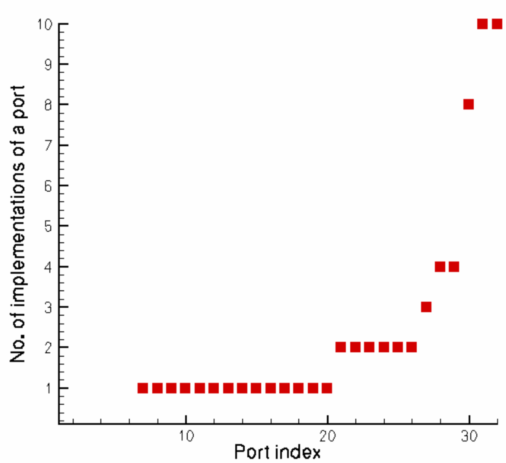
Taming Complexity: Interface Size



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

181

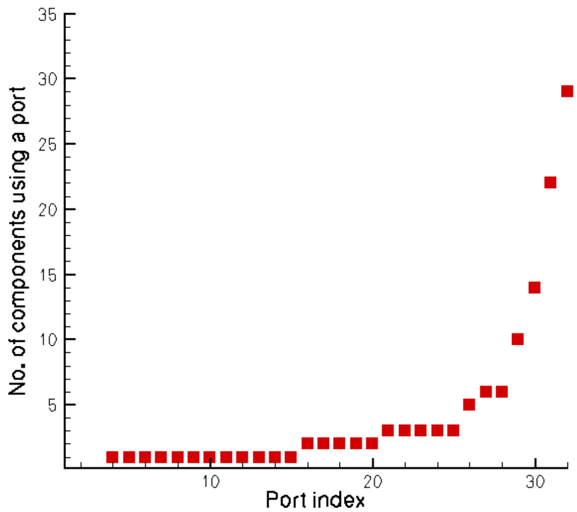
Taming Complexity: Implementations



- CFRFS ports may have just one or many imple-mentations, as needed, but ...
- Most ports have 1 or 2 implementations
- High-utility ports exist, e.g., for exchanging a patch's worth of data

182

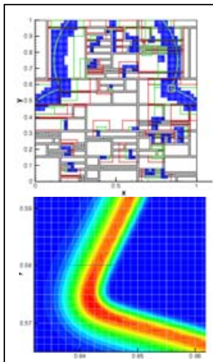
Taming Complexity: Callers



- Most CFRFS ports are used by only a few clients, but …
- Key ports are used by many components

Scientific Productivity

- Conventional Measures
  - 4 journal papers in CFD/Numerics
  - 4 software-oriented journal papers, 1 book chapter
  - ~11 conference papers, including best paper award
  - Over 60 presentations
  - 1 MS and 1 PhD theses
  - 6 test applications
  - See papers at: http://cfrfs.ca.sandia.gov
- Unconventional Measures
  - Did the toolkit spawn new research in app-focused CS (e.g., performance evaluation/enhancement/modeling?)
  - Can the design accommodate software which were themselves designed to be *frameworks* and *not components* ?



H_2O_2 chemical subspecies profile and associated AMR mesh

CCA
Common Component Architecture

Adaptive Codes

- CCA code is an assembly of components
  - Components are shared libraries
  - Component applications can be non-optimal because
    - The mathematical characteristics of the simulation are different from those of the component
    - The component is badly implemented
- So, can one dynamically “re-arrange” an assembly to improve performance?
- Simplistically, 2 approaches:
  - Create performance model per component, use best one
  - Create expert system that analyzes problem and picks good solution strategy

CCA
Common Component Architecture

Performance Measurement in a Component World

- Different from traditional software
  - The researcher may not be the author of the component
  - Will need performance info at component granularity
  - Will need a non-intrusive way of getting it
- Proxies: Simple components to be inserted between a caller and a callee component
  - Intercepts and forwards method calls
  - Use to log size of arrays, calling parameters, counters, etc.
  - Can be used to turn on a “clock”;
    turned off when returning to caller
- Proxy’s ports are the same as the callee’s
  - Functionality is pretty mundane ~ mostly pass through…
    - Can it be automatically generated?

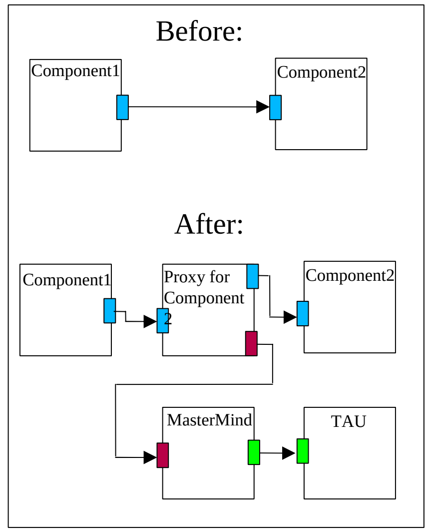
CCA
Common Component Architecture

Integrated Performance Measurement

Measurement infrastructure:

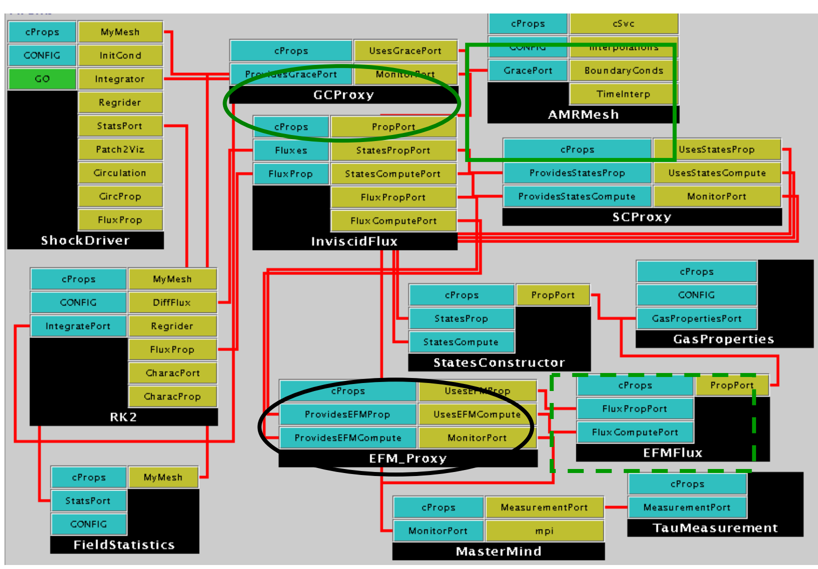
- Proxy
  - Generated automatically using PDT
  - Logs parameters to MasterMind
- MasterMind
  - Collects and stores all measurement data
- TAU
  - Makes all performance measurements

Work done at University of Oregon by A. Malony and team



CCA
Common Component Architecture

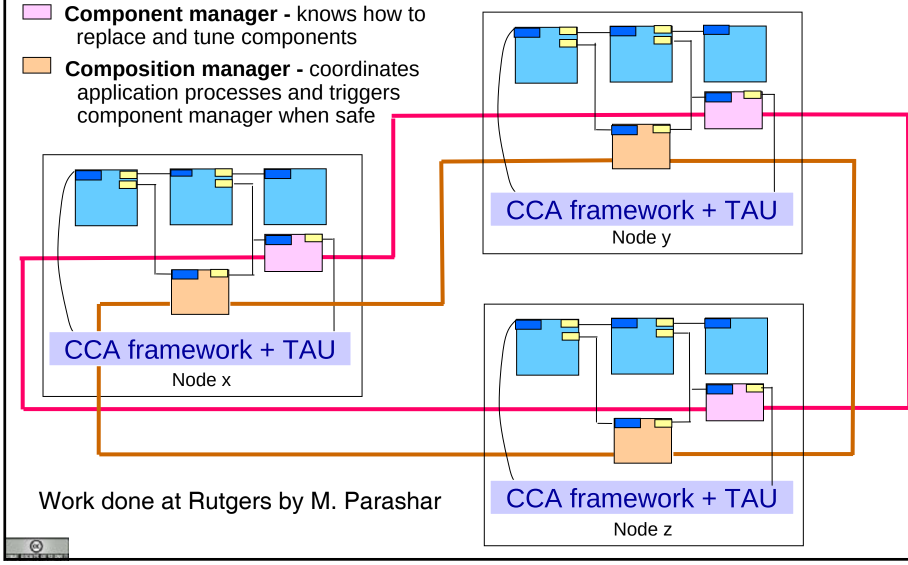
Component Application with Proxies



CCA Common Component Architecture

Extending Ccaffeine to Enable Autonomic Adaptation

- **User component**
- **Component manager -** knows how to replace and tune components
- **Composition manager -** coordinates application processes and triggers component manager when safe

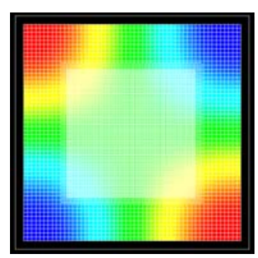


Work done at Rutgers by M. Parashar

CCA Common Component Architecture

Incorporating Other Frameworks

- Chombo (by P. Colella, LBNL) has solvers (AMRGodunov, AMRElliptic, etc.) that:
  - Work on block structured adaptive meshes
  - Accept Chombo-specific data structures
  - But fundamentally require:
    - A double pointer, an array, where variables on a patch (box) are stored in blocked format
    - Bunch of integer arrays that describe the array



Using Chombo to solve the Poisson equation (needed for CFRFS flame simulations).

- Challenge: Can Chombo be used within CCA?
- Need a standardized way of getting data into Chombo
  - Pointer aliasing, not data copy
- Implementation of this “standard” interface

CCA
Common Component Architecture

Using CCA: Summary

- Review of guidelines for developing high-performance scientific components (both new code and wrappers for existing code)
- CCA is an enabling technology for scientific applications
  - Has enabled mathematicians, chemists, combustion scientists, and computer scientists to contribute new strategies that are shrink-wrapped for easy re-use
  - Apart from ***science research***, also spawned ***new research directions in CS***
  - Has enabled research scientists to investigate unconventional approaches, for example multilevel parallelism and dynamic adaptivity
- For more info on the CCA applications/case studies, see:
  - Chemistry: http://www.cca-forum.org/~cca-chem
  - Combustion: http://cfrfs.ca.sandia.gov
- **Different facets of CCA components may be useful within different projects … What are your needs and priorities?**

CCA
Common Component Architecture

A Few Notes in Closing

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

CCA
Common Component Architecture

Resources: Its All Online

- Information about all CCA tutorials, past, present, and future:
  http://www.cca-forum.org/tutorials/
- Specifically…
  - Latest versions of hands-on materials and code:
    http://www.cca-forum.org/tutorials/#sources
    - Hands-On designed for self-study as well as use in an organized tutorial
    - Should work on most Linux distributions, less tested on other unixen
    - Still evolving, so please contact us if you have questions or problems
  - Archives of all tutorial presentations:
    http://www.cca-forum.org/tutorials/archives/
- Questions…
  help@cca-forum.org

CCA
Common Component Architecture

Getting Help

- We want to help insure you have a good experience with CCA, so let us know if you're having problems!
- Tutorial or "start-up" questions
  - help@cca-forum.org
- Problems with specific tools
  - *check documentation for updated contact info*
  - cca-tools bundle (includes Chasm, Babel, Ccaffeine): Rob Armstrong, rob@sandia.gov
  - Chasm: Craig Rasmussen, crasmussen@lanl.gov
  - Babel: babel-users@llnl.gov
  - Ccaffeine: ccafe-users@cca-forum.org
- General questions, or not sure who to ask?
  - help@cca-forum.org

CCA
Common Component Architecture

CCA is Interactive

- Collectively, CCA developers and users span a broad range of scientific interests.
  - There's a good chance we can put you in touch with others with relevant experience with CCA
- CCA Forum Quarterly Meetings
  - Meet many CCA developers and users
  - http://www.cca-forum.org/meetings/
- "Coding Camps"
  - Bring together CCA users & developers for a concentrated session of coding
  - Held as needed, typically 3-5 days
  - May focus on a particular theme, but generally open to all interested participants
  - If you're interested in having one, *speak up* (to individuals or cca-forum@cca-forum.org)
- Visits, Internships, etc.

CCA
Common Component Architecture

Acknowledgements: Tutorial Working Group

- **People:** Benjamin A. Allan, Rob Armstrong, David E. Bernholdt, Randy Bramley, Tamara L. Dahlgren, Lori Freitag Diachin, Wael Elwasif, Tom Epperly, Madhusudhan Govindaraju, Ragib Hasan, Dan Katz, Jim Kohl, Gary Kumfert, Lois Curfman McInnes, Alan Morris, 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.

CCA
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** – Tammy Dahlgren, 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, …
- **Tech-X Corp. –** Johan Carlsson, Svetlana Shasharina, Ovsei Volberg, Nanbor Wang
- **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!

CCA
Common Component Architecture

Thank You!

Thanks for attending this tutorial

We welcome feedback and questions