CCA
Common Component Architecture

Common Component Architecture Concepts

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








Goals

- Introduce the motivation and essential features of the Common Component Architecture
- Provide common vocabulary for remainder of tutorial
- What distinguishes CCA from other component environments?

Historical Perspective

Scientific Computing

- Has focused on extracting performance, parallelism, etc.
- Code complexity has increased due to both performance and scientific issues
- Little has been done to address the complexity

Component Models

- Have focused on managing complexity
- Commodity component models are used primarily in the business and internet software arenas
- Performance (at HPC level) not addressed

CCA is intended to bring the benefits of components to scientific computing

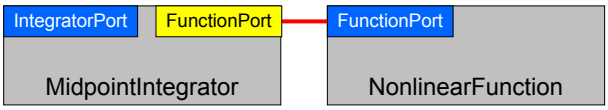
What is the CCA? (User View)

- A component model specifically designed for high-performance scientific computing
- Supports both parallel and distributed applications
- Designed to be implementable without sacrificing performance
- Minimalist approach makes it easier to componentize existing software

What is the CCA? (2)

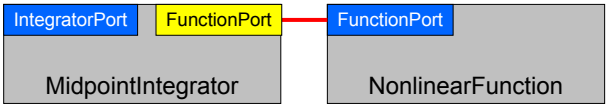
- Components are peers
- *Not* just a dataflow model
- A *tool* to enhance the productivity of scientific programmers
  - Make the hard things easier, make some intractable things tractable
  - Support & promote reuse & interoperability
  - Not a magic bullet

CCA Concepts: Ports



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

CCA Concepts: Components

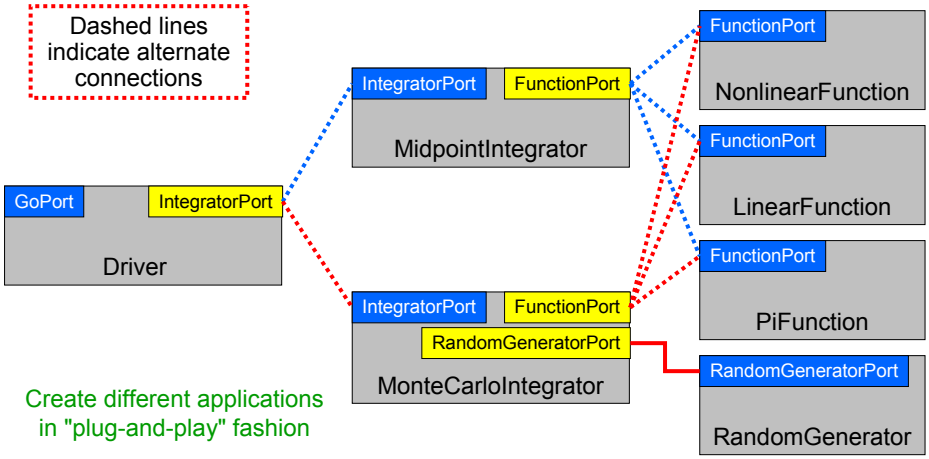


- Components are a unit of software composition
- Components provide/use one or more ports
  - A component with no ports isn't very interesting
- Components include some code which interacts with the CCA framework
  - Implement **setServices** method, constructor, destructor
  - Use **getPort/releasePort** to access ports on other components
- The granularity of components is dictated by the application architecture and by performance considerations
- Components are peers
  - Application architecture determines relationships

CCA Concepts: Frameworks

- The framework provides the means to "hold" components and compose them into applications
  - The framework is often application's "main" or "program"
- Frameworks allow exchange of ports among components without exposing implementation details
- Frameworks provide a small set of standard services to components
  - BuilderService allow programs to compose CCA apps
- Frameworks may make themselves appear as components in order to connect to components in other frameworks
- *Currently:* specific frameworks support specific computing models (parallel, distributed, etc.). *Future:* full flexibility through integration or interoperation

Components and Ports in the Integrator Example



Writing Components

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

***Much* more detail later in the tutorial!**

Adapting Existing Code into Components

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

Writing Frameworks

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

Typical Component Lifecycle

We'll look at actual code in next tutorial module

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

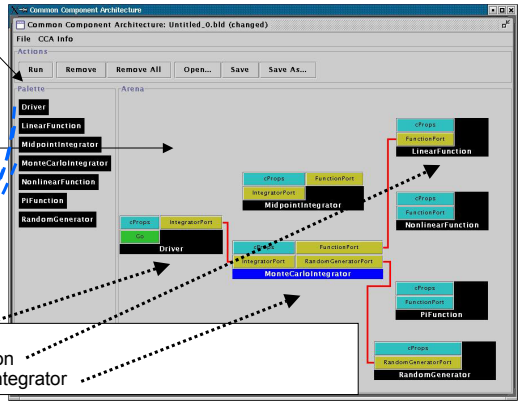
In an application, individual components may be in different phases at different times

Steps may be under human or software control

User Viewpoint: Loading and Instantiating Components

- Components are code (usu. library or shared object) + 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

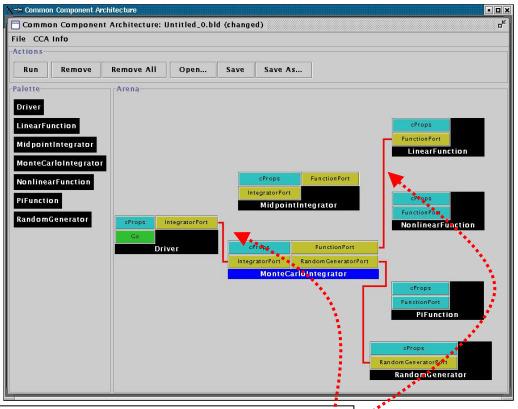
- Details are framework-specific!
- Ccaffeine currently provides both command line and GUI approaches



create	Driver	Driver
create	LinearFunction	LinearFunction
create	MonteCarloIntegrator	MonteCarloIntegrator

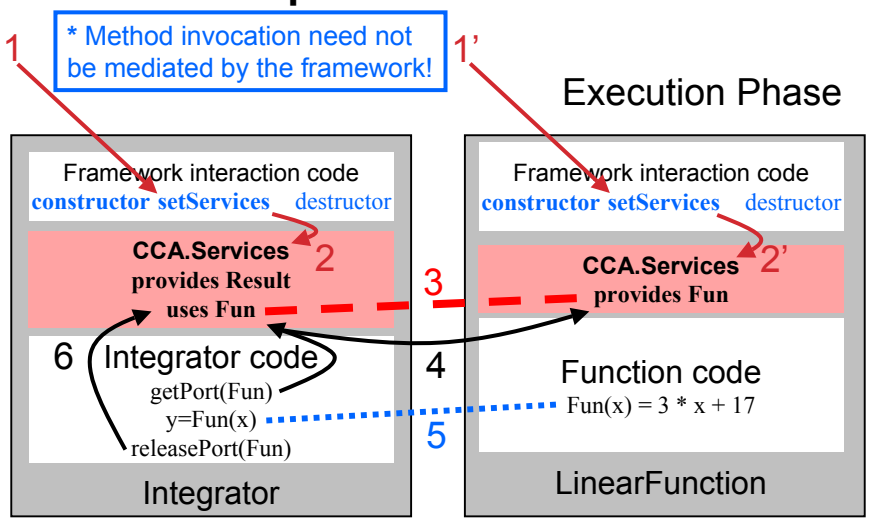
User Connects Ports

- 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



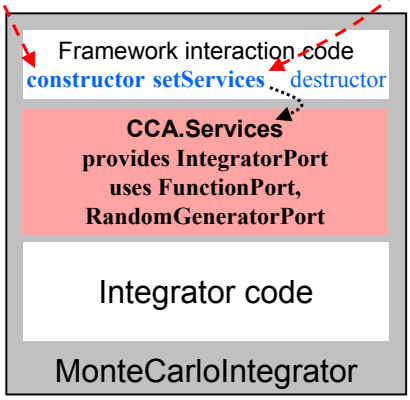
connect	Driver	IntegratorPort	MonteCarloIntegrator	IntegratorPort
connect	MonteCarloIntegrator	FunctionPort	LinearFunction	FunctionPort
…				

Framework Mediates *Most** Component Interactions

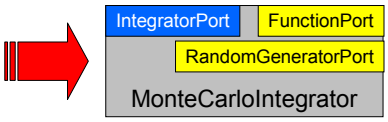


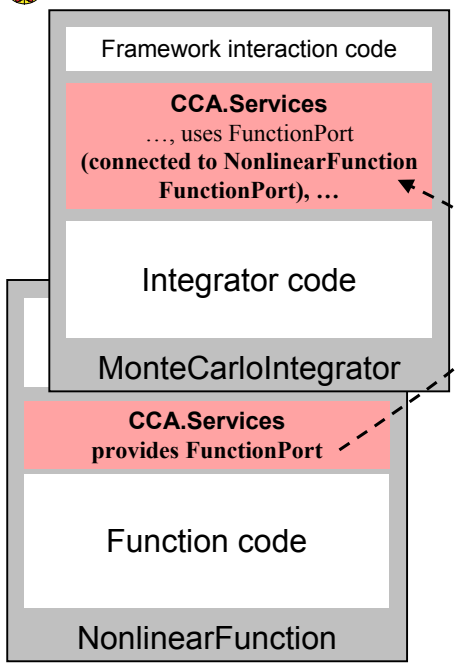
Component's View of Instantiation

- 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





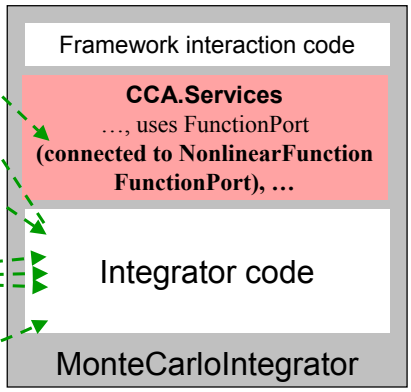
Component's View of Connection

- 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

Component's View of Using a Port

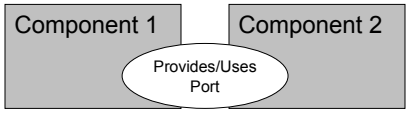
- User calls getPort to obtain (handle for) port from Services
  - Finally user code can "see" provider
- Cast port to expected type
  - OO programming concept
  - Insures type safety
  - Helps enforce declared interface
- Call methods on port
  - e.g.

  sum = sum + function->evaluate(x)
- Release port

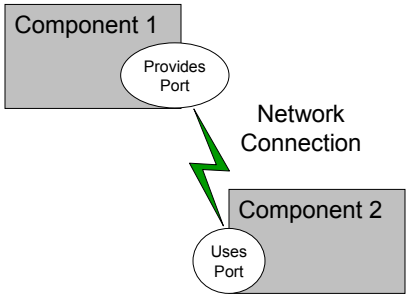


Importance of Provides/Uses Pattern for Ports

- Fences between components
  - Components must declare both what they provide and what they use
  - Components cannot interact until ports are connected
  - No mechanism to call anything not part of a port
- Ports preserve high performance direct connection semantics…
- …While also allowing distributed computing



Direct Connection



CCA Concepts: "Direct Connection" Maintains Local Performance

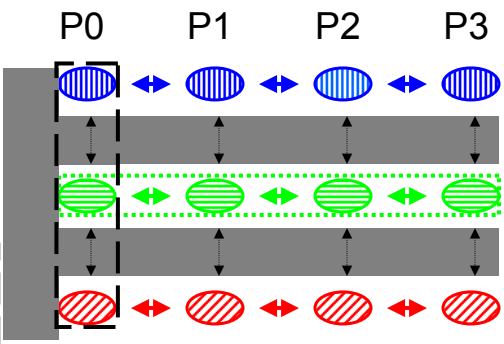
- Calls *between* components equivalent to a C++ virtual function call: lookup function location, invoke it
  - Cost equivalent of ~2.8 F77 or C function calls
  - ~48 ns vs 17 ns on 500 MHz Pentium III Linux box
- Language interoperability can impose additional overheads
  - Some arguments require conversion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

How Does Direct Connection Work?

- Components loaded into separate namespaces in the same address space (process) from shared libraries
- getPort call returns a pointer to the port's function table
- All this happens "automatically" – user just sees high performance
  - *Description reflects Ccaffeine implementation, but similar or identical mechanisms are in other direct connect fwks*
- *Many CORBA implementations offer a similar approach to improve performance, but using it violates the CORBA standards!*

CCA Concepts: Framework Stays "Out of the Way" of Component Parallelism

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



Components: Blue, Green, Red

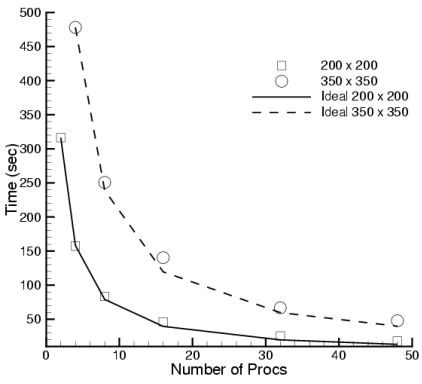
Framework: Gray

MCMD/MPMD also supported

Other component models ignore parallelism entirely

Scalability of Scientific Data Components in CFRFS Combustion Applications

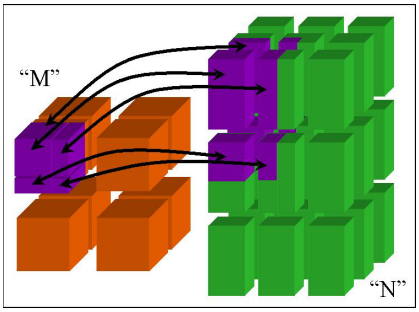
- Investigators: S. Lefantzi, J. Ray, and H. Najm (SNL)
- Uses GrACEComponent, CvodesComponent, etc.
- Shock-hydro code with no refinement
- 200 x 200 & 350 x 350 meshes
- Cplant cluster
  - 400 MHz EV5 Alphas
  - 1 Gb/s Myrinet
- Negligible component overhead
- Worst perf : 73% scaling efficiency for 200x200 mesh on 48 procs



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

CCA Concepts: MxN Parallel Data Redistribution

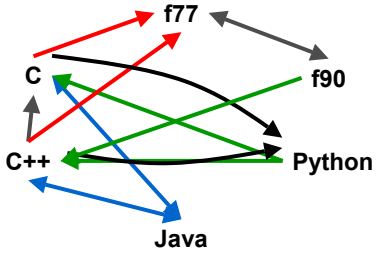
- Share Data Among Coupled Parallel Models
  - Disparate Parallel Topologies (M processes vs. N)
  - e.g. Ocean & Atmosphere, Solver & Optimizer…
  - e.g. Visualization (Mx1, increasingly, MxN)



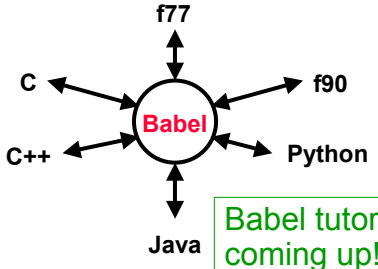
Research area -- tools under development

CCA Concepts: Language Interoperability

- Existing language interoperability approaches are "point-to-point" solutions



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



Babel tutorial coming up!

Few other component models support all languages and data types important for scientific computing

What the CCA isn't…

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

What the CCA *is…*

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

Concept Review

- **Ports**
  - Interfaces between components
  - Uses/provides model
- **Framework**
  - Allows assembly of components into applications
- **Direct Connection**
  - Maintain performance of local inter-component calls
- **Parallelism**
  - Framework stays out of the way of parallel components
- **MxN Parallel Data Redistribution**
  - Model coupling, visualization, etc.
- **Language Interoperability**
  - Babel, Scientific Interface Definition Language (SIDL)