An Overview of Components for Scientific Computing and Introduction to the Common Component Architecture

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

Goals of This Module

- Introduce basic concepts and vocabulary of component-based software engineering
- Highlight the special demands of high-performance scientific computing on component environments
- Introduce some terminology and concepts from the Common Component Architecture
- Provide a unifying context for the remaining talks
  - For those attending the extended CCA tutorial

Motivation: Modern Scientific Software Engineering Challenges

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

Motivation: For Library Developers

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

Motivation: For Application Developers and Users

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

Some Observations About Software…

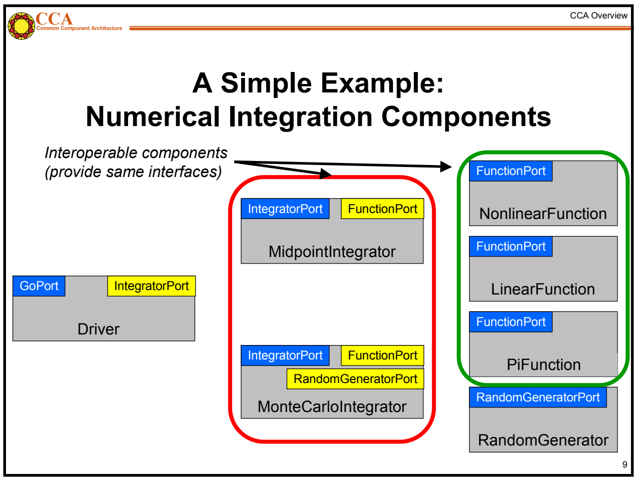
- "The complexity of software is an essential property, not an accidental one." *[Brooks]*
  - We can't get rid of complexity
- "Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements." *[Booch]*
  - We must find ways to manage it

More Observations…

- "A complex system that works is invariably found to have evolved from a simple system that worked… A complex system designed from scratch never works and cannot be patched up to make it work." *[Gall]*
  - Build up from simpler pieces
- "The best software is code you don't have to write" *[Jobs]*
  - Reuse code wherever possible

Component-Based Software Engineering

- CBSE methodology is emerging, especially from business and internet areas
- **Software productivity**
  - Provides a "plug and play" application development environment
  - Many components available "off the shelf"
  - Abstract interfaces facilitate reuse and interoperability of software
- **Software complexity**
  - Components encapsulate much complexity into "black boxes"
  - Plug and play approach simplifies applications
  - Model coupling is natural in component-based approach
- **Software performance** (indirect)
  - Plug and play approach and rich "off the shelf" component library simplify changes to accommodate different platforms



A Simple Example: Numerical Integration Components

Interoperable components (provide same interfaces)

GoPort IntegratorPort
Driver

IntegratorPort FunctionPort
MidpointIntegrator

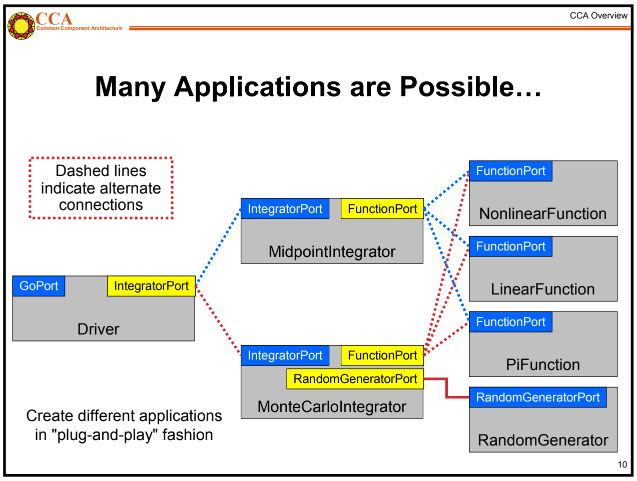
IntegratorPort FunctionPort RandomGeneratorPort
MonteCarloIntegrator

FunctionPort
NonlinearFunction

FunctionPort
LinearFunction

FunctionPort
PiFunction

RandomGeneratorPort
RandomGenerator



Many Applications are Possible…

Dashed lines indicate alternate connections

GoPort IntegratorPort
Driver

IntegratorPort FunctionPort
MidpointIntegrator

IntegratorPort FunctionPort RandomGeneratorPort
MonteCarloIntegrator

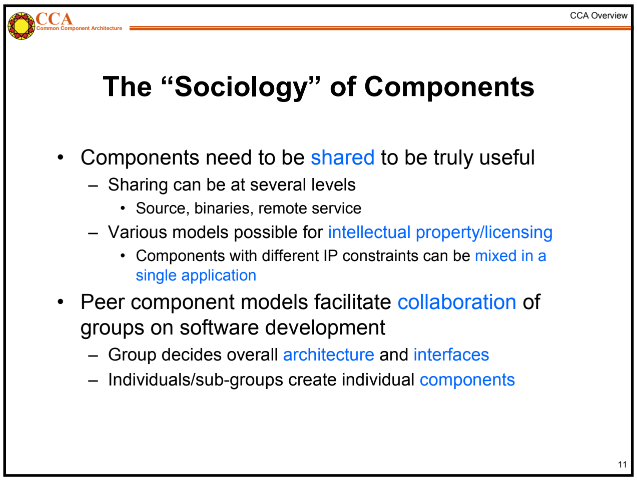
FunctionPort
NonlinearFunction

FunctionPort
LinearFunction

FunctionPort
PiFunction

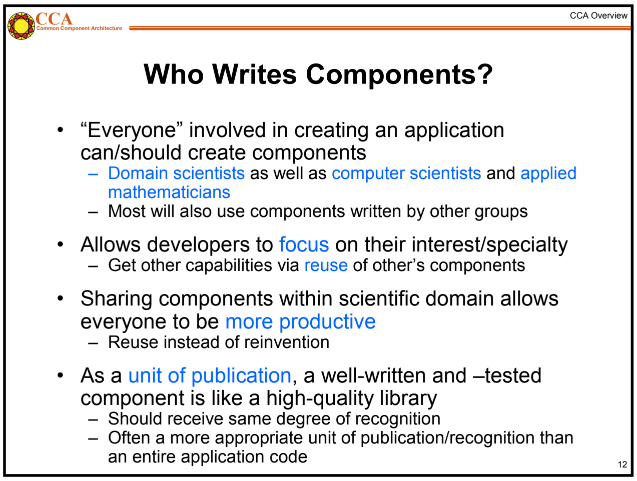
RandomGeneratorPort
RandomGenerator

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



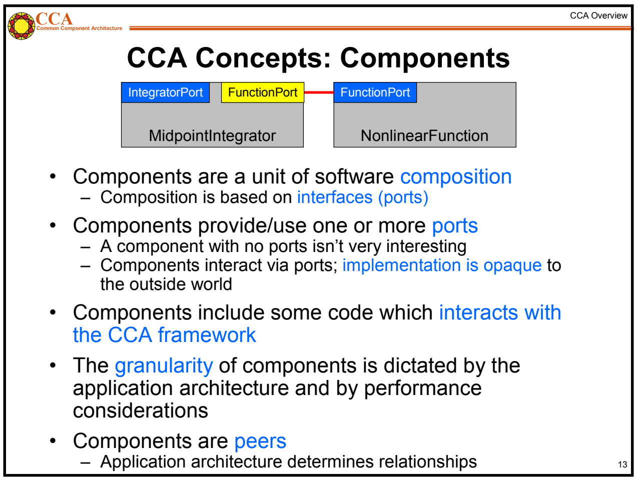
The “Sociology” of Components

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



Who Writes Components?

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

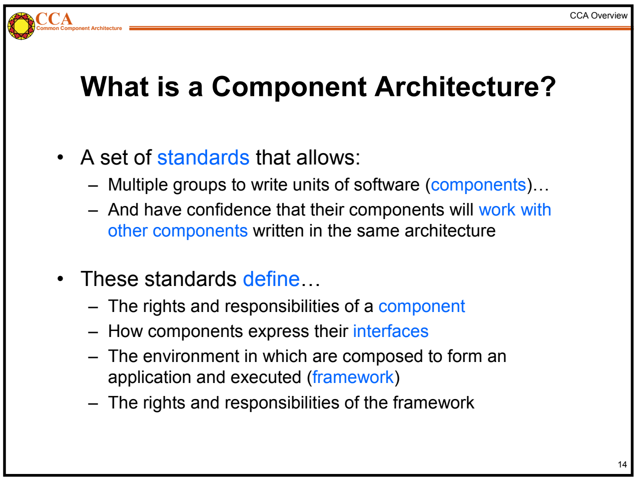


CCA Concepts: Components

IntegratorPort | FunctionPort — FunctionPort

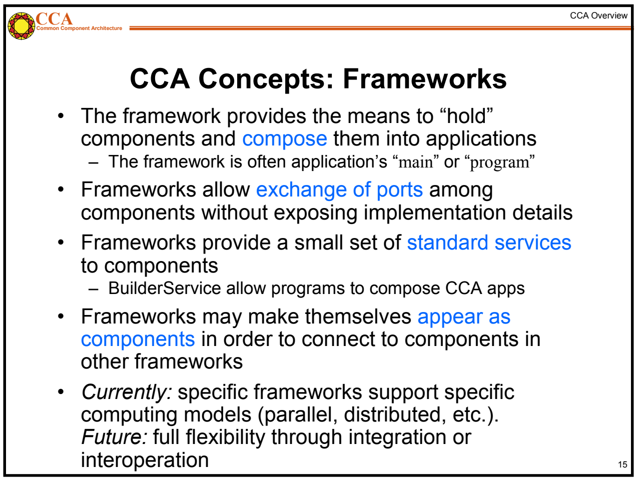
MidpointIntegrator | NonlinearFunction

- Components are a unit of software composition
  - Composition is based on interfaces (ports)
- Components provide/use one or more ports
  - A component with no ports isn't very interesting
  - Components interact via ports; implementation is opaque to the outside world
- Components include some code which interacts with the CCA framework
- The granularity of components is dictated by the application architecture and by performance considerations
- Components are peers
  - Application architecture determines relationships



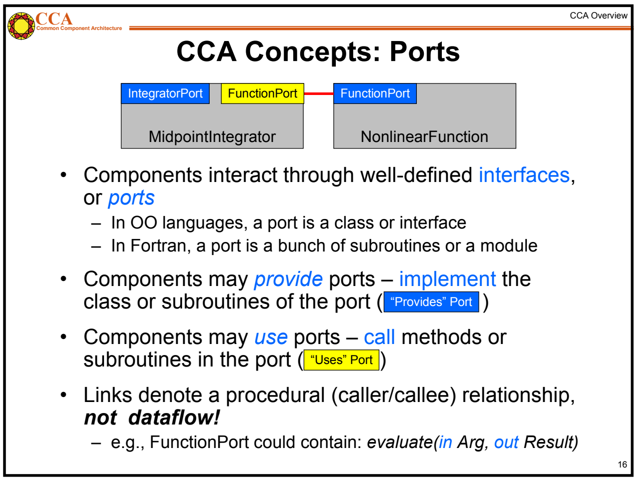
What is a Component Architecture?

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



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



CCA Concepts: Ports

IntegratorPort | FunctionPort — FunctionPort

MidpointIntegrator | NonlinearFunction

- 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)*

Interfaces, Interoperability, and Reuse

- Interfaces define how components interact…
- Therefore interfaces are key to interoperability and reuse of components

- In many cases, "any old interface" will do, but…
- General plug and play interoperability requires multiple implementations providing the same interface

- Reuse of components occurs when they provide interfaces (functionality) needed in multiple applications

Designing for Reuse, Implications

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

Relationships: Components, Objects, and Libraries

- Components are typically discussed as objects or collections of objects
  - Interfaces generally designed in OO terms, but…
  - Component internals need not be OO
  - OO languages are not required
- Component environments can enforce the use of published interfaces (prevent access to internals)
  - Libraries can not
- It is possible to load several instances (versions) of a component in a single application
  - Impossible with libraries
- Components *must* include some code to interface with the framework/component environment
  - Libraries and objects do not

Domain-Specific Frameworks vs Generic Component Architectures

Domain-Specific

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

Generic

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

Special Needs of Scientific HPC

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

Commodity Component Models

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

What is the CCA? (User View)

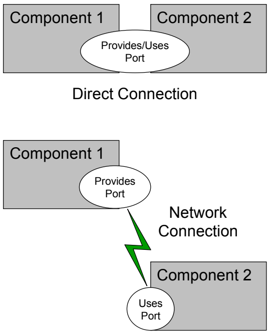
- A component model specifically designed for high-performance scientific computing
- Supports both parallel and distributed applications
- Designed to be implementable without sacrificing performance
- Minimalist approach makes it easier to componentize existing software

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

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

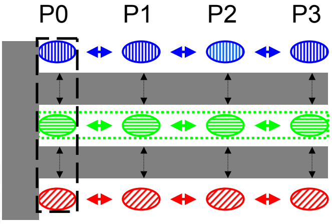


CCA Concepts: "Direct Connection" Maintains Local Performance

- 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

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

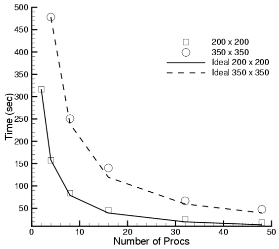
Framework: Gray

MCMD/MPMD also supported

Other component models ignore parallelism entirely

Scalability of Scientific Data Components in CFRFS Combustion Applications

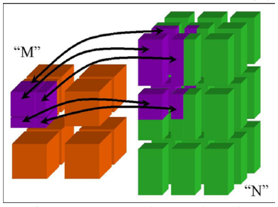
- Investigators: S. Lefantzi, J. Ray, and H. Najm (SNL)
- Uses GrACEComponent, CvodesComponent, etc.
- Shock-hydro code with no refinement
- 200 x 200 & 350 x 350 meshes
- Cplant cluster
  - 400 MHz EV5 Alphas
  - 1 Gb/s Myrinet
- Negligible component overhead
- Worst perf : 73% scaling efficiency for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

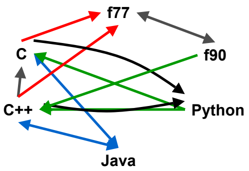


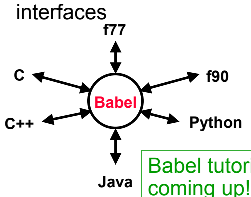
Research area -- tools under development

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

Review of CCA Terms & Concepts

- **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)

Summary

- Components are a software engineering tool to help address software productivity and complexity
- Important concepts: components, interfaces, frameworks, composability, reuse
- Scientific component environments come in “domain specific” and “generic” flavors
- Scientific HPC imposes special demands on component environments
  - Which commodity tools may have trouble with
- The Common Component Architecture is specially designed for the needs of HPC