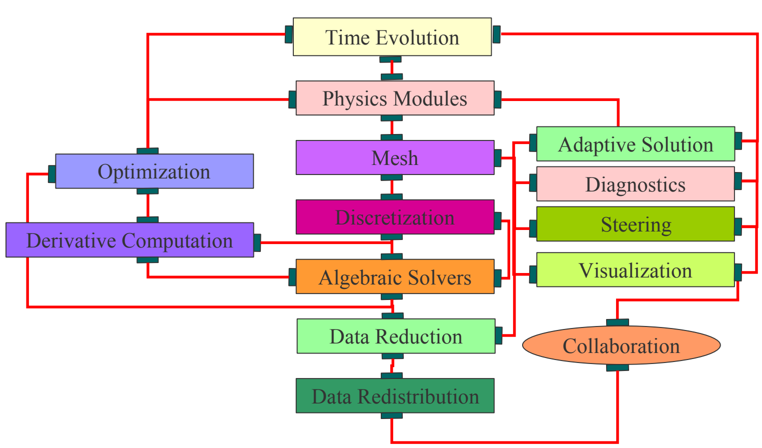
CCA
Common Component Architecture

A Look at More Complex Component-Based Applications

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

Modern Scientific Software Development

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



Overview

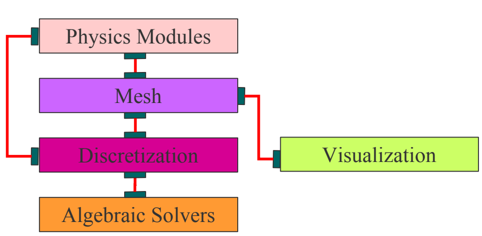
- Using components in high performance simulation codes
  - Examples of increasing complexity
  - Performance
    - Single processor
    - Scalability
- Developing components for high performance simulation codes
  - Strategies for thinking about your own application
  - Developing interoperable and interchangeable components

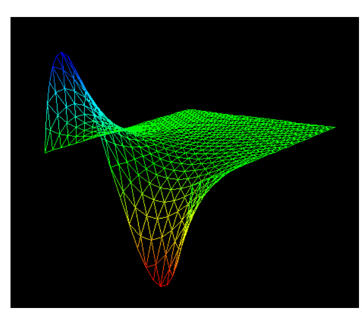
Our Starting Point

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

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

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





Numerical Solution of Example 1

- Physics: Poisson's equation
- Grid: Unstructured triangular mesh
- Discretization: Finite element method
- Algebraic Solvers: PETSc (Portable Extensible Toolkit for Scientific Computation)
- Visualization: VTK tool
- Original Language: C

Creating Components: Step 1

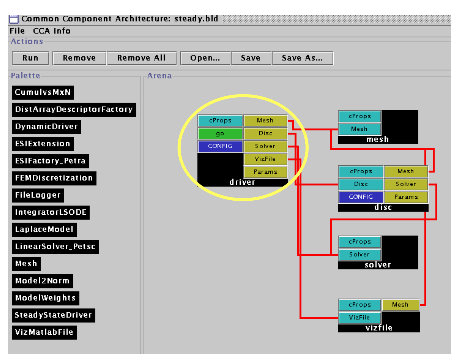
- Separate the application code into well-defined pieces that encapsulate functionalities
  - Decouple code along numerical functionality
    - Mesh, Discretization, Solver, Visualization
    - Physics is kept separate
  - Determine what questions each component can ask of and answer for other components (this determines the ports)
    - Mesh provides geometry and topology (needed by discretization and visualization)
    - Mesh allows user defined data to be attached to its entities (needed by physics and discretization)
    - Mesh *does not* provide access to its data structures
  - If this is not part of the original code design, this is by far the hardest, most time consuming aspect of componentization

Creating the Components: Step 2

- Writing C++ Components
  – Create an abstract base class for each port
  – Create C++ objects that inherit from the abstract base port class and the CCA component class
  – Wrap the existing code as a C++ object
  – Implement the setServices method
- This process was significantly less time consuming (with an expert present) than the decoupling process
  – Lessons learned
    - Definitely look at an existing, working example for the targeted framework
    - Experts are very handy people to have around ;-)

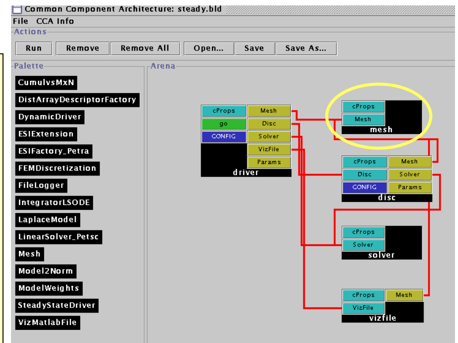
The Componentized Example

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



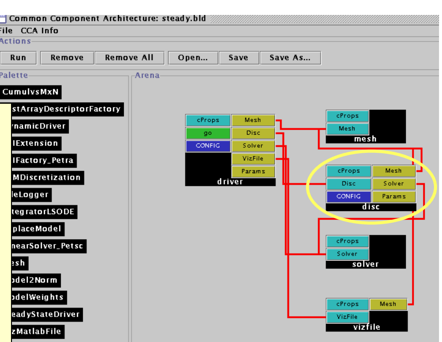
The Componentized Example

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



The Componentized Example

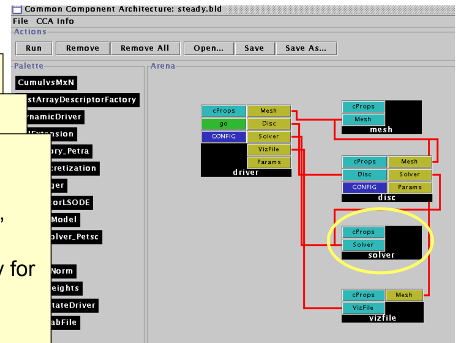
- The Driver Component
- The Mesh Component
- The Discretization Component
  - Provides a finite element discretization of basic operators (gradient, laplacian, scalar terms)
  - Driver determines which terms are included and their coefficients
  - Provides mechanisms for general Dirichlet and Neumann boundary condition matrix manipulations
  - Computes element matrices and assembles them into the global stiffness matrix via set methods on the solver
  - Gathers and scatters vectors to the mesh (in this case φ)



Complex Applications

The Componentized Example

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

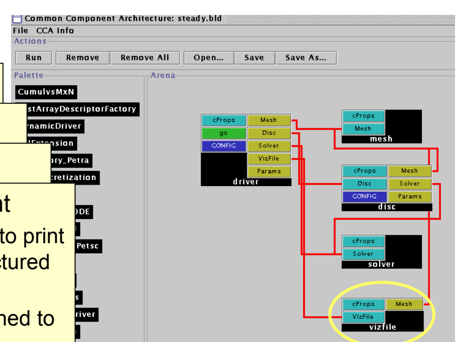


11

Complex Applications

The Componentized Example

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



12

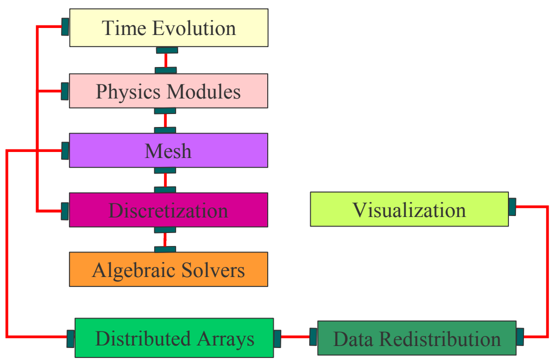
The next step… time dependence

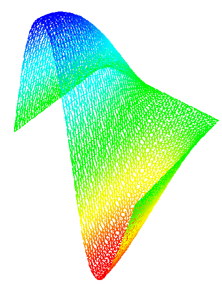
$$\delta\varphi/\delta t = \nabla^2\varphi \;(x,y,t) \in [0,1] \times [0,1]$$

$$\varphi(0,y,t)=0 \qquad \varphi(1,y,t)=.5\sin(2\pi y)\cos(t/2)$$

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

$$\varphi(x,y,0)=\sin(.5\pi x)\ \sin(2\pi y)$$





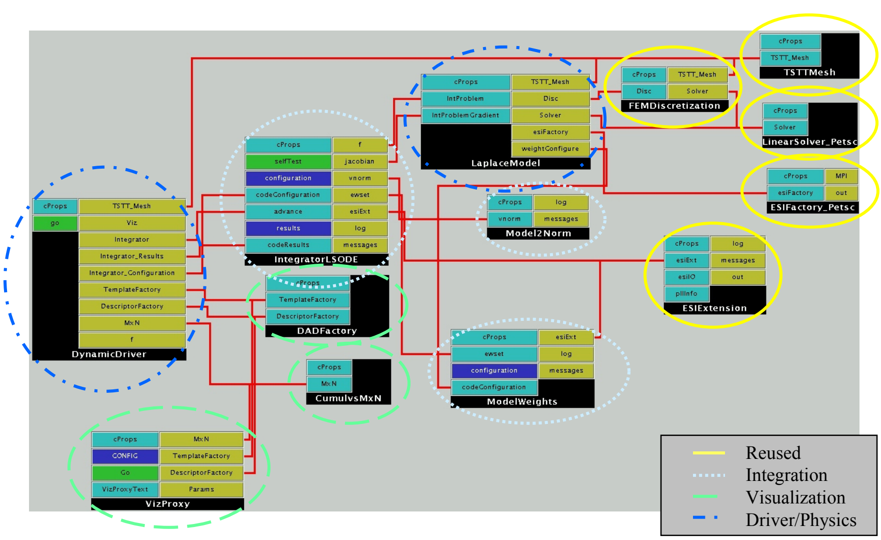
Some things change…

- Requires a time integration component
  - Based on the LSODE library (LLNL)
  - Component implementation developed by Ben Allan (SNL)
- Uses a new visualization component
  - Based on AVS
  - Requires an MxN data redistribution component
  - Developed by Jim Kohl (ORNL)
- The MxN redistribution component requires a Distributed Array component
  - Similar to HPF arrays
  - Developed by David Bernholdt (ORNL)
- The driver component changes to accommodate the new physics

… and some things stay the same

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

The CCA wiring diagram

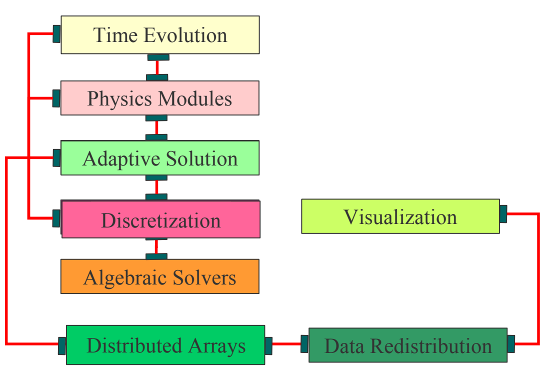


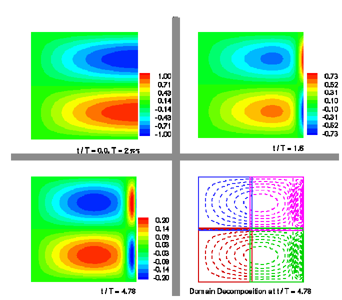
What did this exercise teach us?

- It was easy to incorporate the functionalities of components developed at other labs and institutions given a well-defined interface and header file.
  - In fact, some components (one uses and one provides) were developed simultaneously across the country from each other after the definition of a header file.
  - Amazingly enough, they usually “just worked” when linked together (and debugged individually).
- In this case, the complexity of the component-based approach was higher than the original code complexity.
  - Partially due to the simplicity of this example
  - Partially due to the limitations of the some of the current implementations of components

One more layer of complexity… AMR

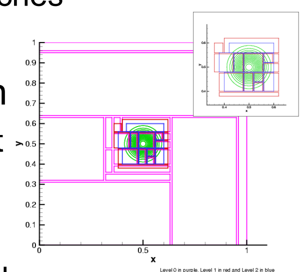
The same physics but use a block structured adaptive mesh





Adaptive Mesh Refinement

- Used to accurately capture a wide spectrum of length scales
- Many different techniques
  - We use structured axis-aligned patches
  - Provided by the GrACE library
- Start with a uniform coarse mesh
  - Identify regions needing refinement
  - Collate into rectangular patches
  - Impose finer mesh in patches
  - Recurse and obtain a mesh hierarchy.



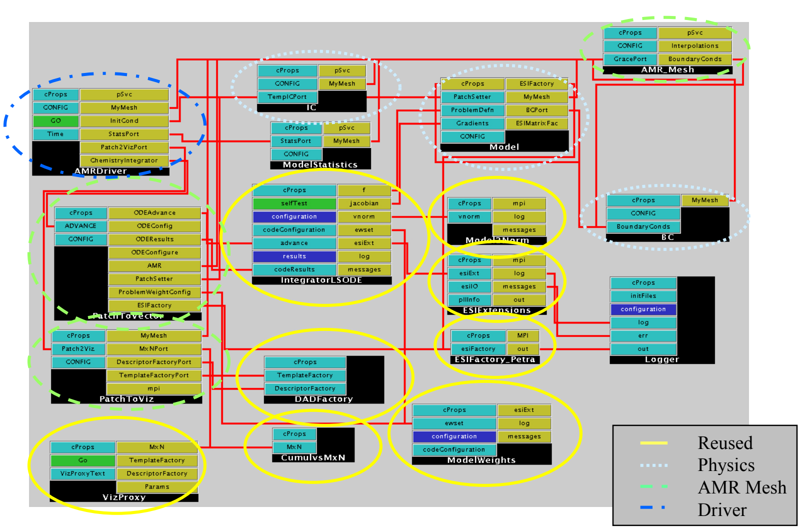
Some things change…

- The mesh component changes
  - Block structured AMR based on GRACE
- The discretization component changes
  - Finite difference on patches
  - BC handled differently
- The driver component changes

… and some things stay the same

- The integration component stays the same
- The solver component stays the same
- The data redistribution component stays the same
- The distributed array component stays the same
- The visualization component stays the same

The component implementation



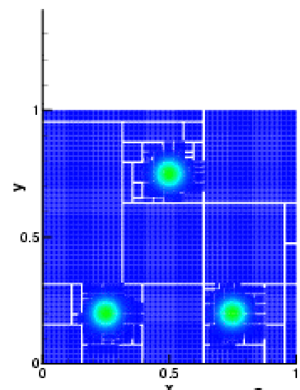
Beyond the heat equation…

- Flame Approximation
  - H_2-Air mixture; ignition via 3 hot-spots
  - 9-species, 19 reactions, stiff chemistry
- Governing equation

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

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

Temperature (K)

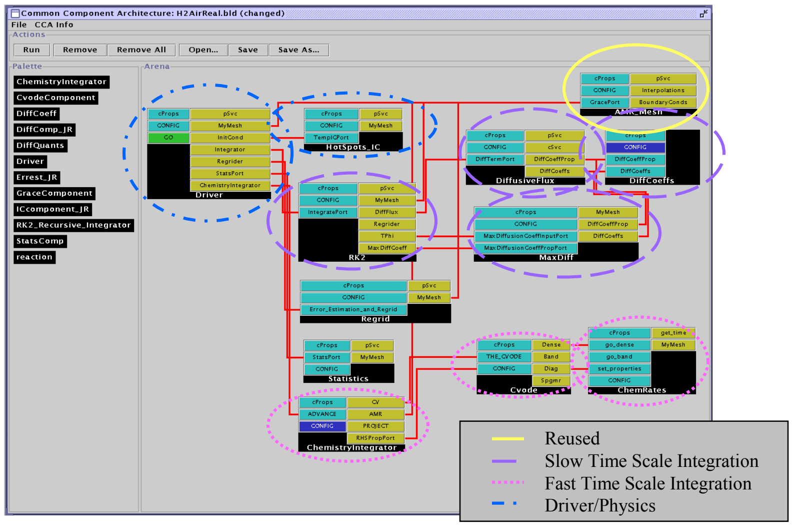


0 ms.

Numerical Solution

- Adaptive Mesh Refinement: GrACE
- Stiff integrator: CVODE (LLNL)
- Diffusive integrator: 2^{nd} Order Runge Kutta
- Chemical Rates: legacy f77 code (SNL)
- Diffusion Coefficients: legacy f77 code (SNL)
- New code less than 10%

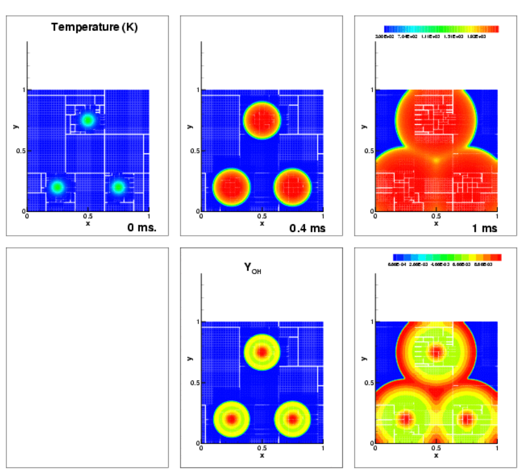
The CCA Wiring Diagram



Evolution of the Solution

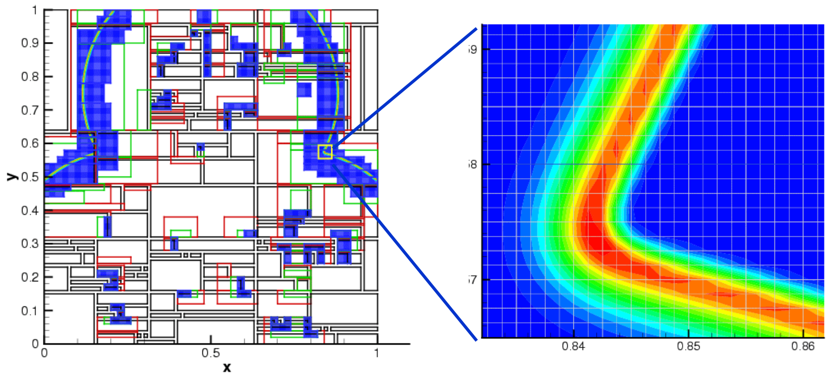
Temperature

OH Profile



The need for AMR

- H_2O_2 chemical subspecies profile
  - Only 100 microns thick (about 10 fine level cells)
  - Not resolvable on coarsest mesh

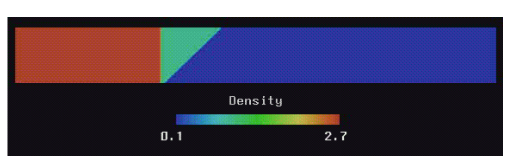


Shock-Hydrodynamics

- Governing equation

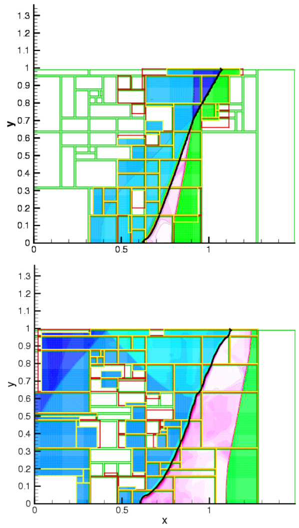
$$U_t = F_x(U) + G_y(U) \quad U = \{\rho, \rho u, \rho v, \rho E, \rho\zeta\}$$

- Domain
  - Square cross section shock-tube
- Experiment
  - Two gases are separated by a clean interface
  - Shock moves from left to right and interacts with the interface
    - Deposits vorticity
    - Reflects
    - Refracts

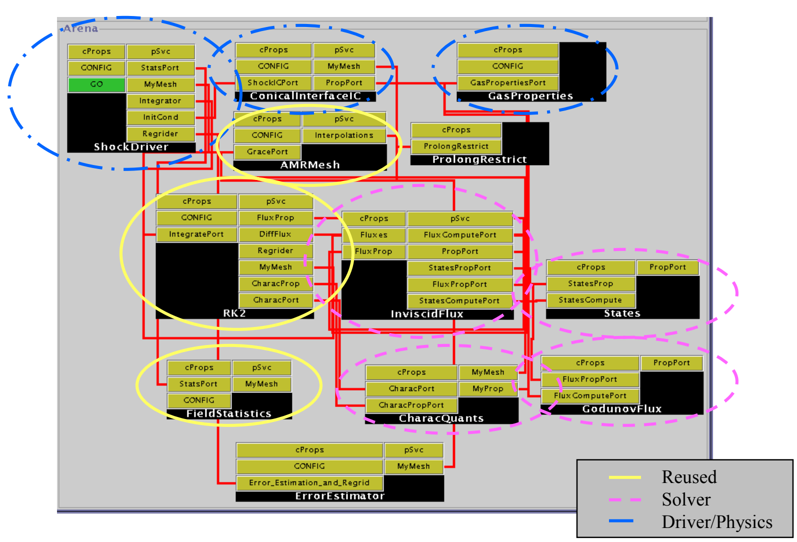


Interesting features

- Shock & interface are sharp discontinuities which need refinement
- Shock deposits vorticity – a governing quantity for turbulence, mixing, …
- If there is insufficient refinement
  - under predict vorticity
  - slower mixing/turbulence.



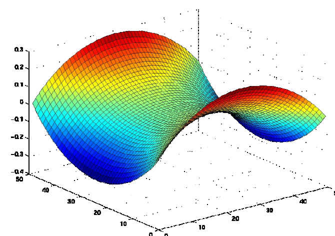
The CCA Wiring Diagram



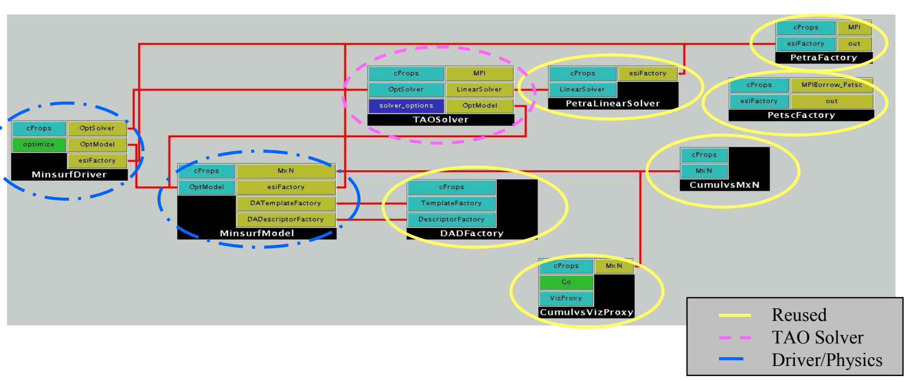
Unconstrained Minimization Problem

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

  min f(x), where f: $R^n \rightarrow R$
- Solve using optimization components based on TAO (ANL)



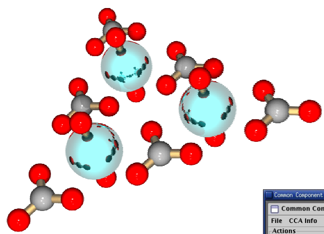
Unconstrained Minimization Using a Structured Mesh



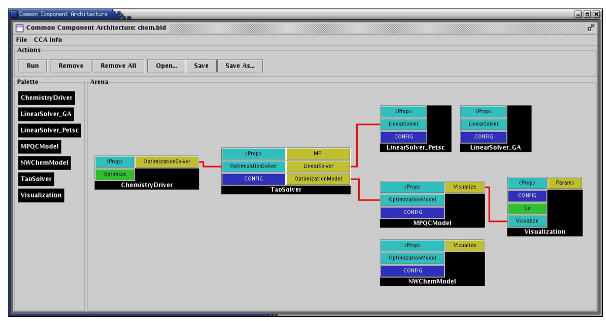
Molecular Geometry Optimization

Wiring diagram using Ccaffeine framework and:

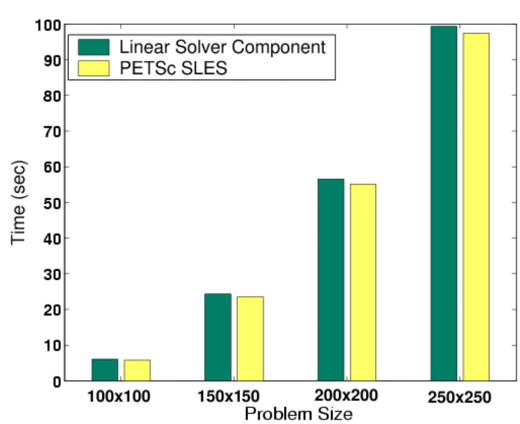
- Electronic structure components based on NWChem (PNNL) and MPQC (SNL)
- Optimization components based on TAO (ANL)
- Linear algebra components based on Global Arrays (PNNL) and PETSc (ANL)



Relativistic quantum chemistry calculation of $(UO_2)_3(CO_3)_6$ using NWChem. Image courtesy of Wibe deJong, PNNL.



Component Overhead



Aggregate time for linear solver component in unconstrained minimization problem.

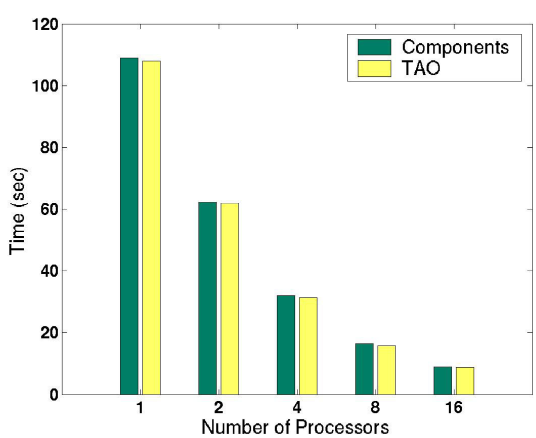
- Negligible overhead for component implementation and abstract interfaces when using appropriate levels of abstraction
- Linear solver component currently supports any methods available via the ESI interfaces to PETSc and Trilinos; plan to support additional interfaces the future, e.g., those under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

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 on a Linux Cluster



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

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

List of Component Re-Use

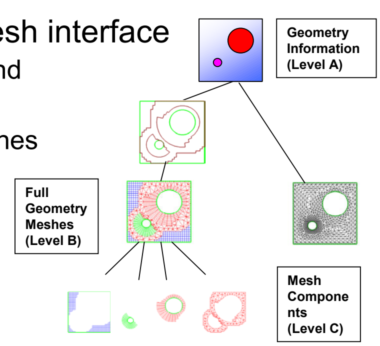
- Various services in CCAFFEINE
- Integrator
  - *IntegratorLSODE (2)*
  - *RK2 (2)*
- Linear solvers
  - *LinearSolver_Petra (4)*
  - *LinearSolver_PETSc (4)*
- AMR
  - *AMRmesh (3)*
- Data description
  - *DADFactory (3)*
- Data redistribution
  - *CumulvsMxN (3)*
- Visualization
  - *CumulvsVizProxy (3)*

Component interfaces to numerical libraries

Component interfaces to parallel data management and visualization tools

The Next Level

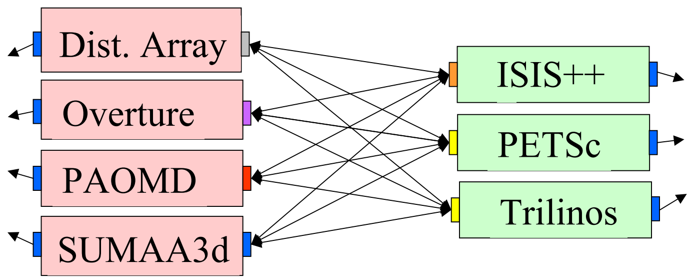
- Common Interface Specification
  - Provides plug-and-play interchangeability
  - Requires domain specific experts
  - Typically a difficult, time-consuming task
  - A success story: MPI
- A case study… the TSTT/CCA mesh interface
  - TSTT = Terascale Simulation Tools and Technologies (www.tstt-scidac.org)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



Current Situation

Current Situation

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



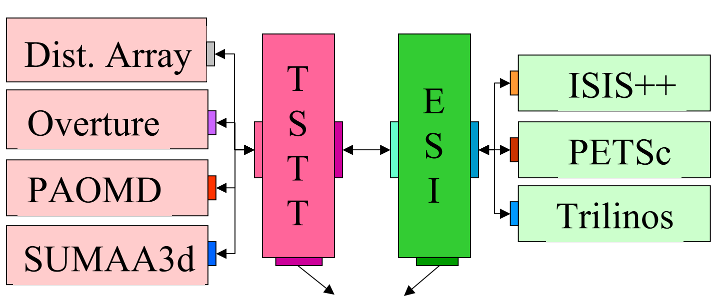
Common Interface Specification

Reduces the *Many-to-Many* problem to a *Many-to-One* problem

– Allows interchangeability and experimentation

– Challenges

- Interface agreement
- Functionality limitations
- Maintaining performance



TSTT Philosophy

- Create a small set of interfaces that existing packages can support
  - AOMD, CUBIT, Overture, GrACE, …
  - Enable both interchangeability and interoperability
- Balance performance and flexibility
- Work with a large tool provider and application community to ensure applicability
  - Tool providers: TSTT and CCA SciDAC centers
  - Application community: SciDAC and other DOE applications

Basic Interface

- Enumerated types
  - Entity Type: VERTEX, EDGE, FACE, REGION
  - Entity Topology: POINT, LINE, POLYGON, TRIANGLE, QUADRILATERAL, POLYHEDRON, TETRAHEDRON, HEXAHEDRON, PRISM, PYRAMID, SEPTAHEDRON
- Opaque Types
  - Mesh, Entity, Workset, Tag
- Required interfaces
  - Entity queries (geometry, adjacencies), Entity iterators, Array-based query, Workset iterators, Mesh/Entity Tags, Mesh Services

Issues that have arisen

- Nomenclature is harder than we first thought
- Cannot achieve the 100 percent solution, so...
  - What level of functionality should be supported?
    - Minimal interfaces only?
    - Interfaces for convenience and performance?
  - What about support of existing packages?
    - Are there atomic operations that all support?
    - What additional functionalities from existing packages should be required?
  - What about additional functionalities such as locking?
- Language interoperability is a problem
  - Most TSTT tools are in C++, most target applications are in Fortran
  - How can we avoid the "least common denominator" solution?
  - Exploring the SIDL/Babel language interoperability tool

Summary

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

Componentizing your own application

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

Next: Status and Plans