Component Technology for High-Performance Scientific Simulation Software

Scott Kohn

with

Tom Epperly and Gary Kumfert

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

October 3, 2000

Presentation outline

- Motivation
- DOE community activities (CCA)
- Language interoperability technology (*Babel*)
- Component type and software repository (*Alexandria*)
- Research issues in parallel component communication
- Deep thoughts...

Goal: Provide an overview of the approach, techniques, and tools we are exploring in adopting software component technology for scientific computing

Numerical simulation software is becoming increasingly complex and interdisciplinary

- **Scientists are asked to develop 3d, massively parallel, high-fidelity, full-physics simulations; *and do it quickly***
- **This requires the integration of software libraries developed by other teams**
  - local resources are limited and expertise may not exist
  - loss of local control over software development decisions
  - language interoperability issues (f77, C, C++, Python, Java, f90)
- **Techniques for small codes do not scale to 500K lines**

What are the barriers to software re-use, interoperability, and integration?

- **Technological barriers**
  - incompatible programming languages (f90 calling C++)
  - incompatibilities in C and C++ header files (poor physical design)
  - conflicting low-level run-time support (e.g., reference counting)

- **Sociological barriers**
  - trust (“how do I know you know what you’re doing?”)
  - “I could re-write it in less time than it would take to learn it…”

- **Domain understanding barriers (the interesting one!)**
  - understand interactions of the math and physics packages
  - write software that reflects that understanding
  - this is where we gain insights and make scientific progress

Component technologies address issues of software complexity and interoperability

- **Industry created component technology to address...**
  - interoperability problems due to languages
  - complexity of large applications with third-party software
  - incremental evolution of large legacy software

Observation: The laboratory must address similar problems but in a different applications space (parallel high-performance scientific simulation, not business).

Current industry solutions will not work in a scientific computing environment

- **Three competing industry component approaches**
  - Microsoft COM
  - Sun JavaBeans and Enterprise JavaBeans
  - OMG CORBA

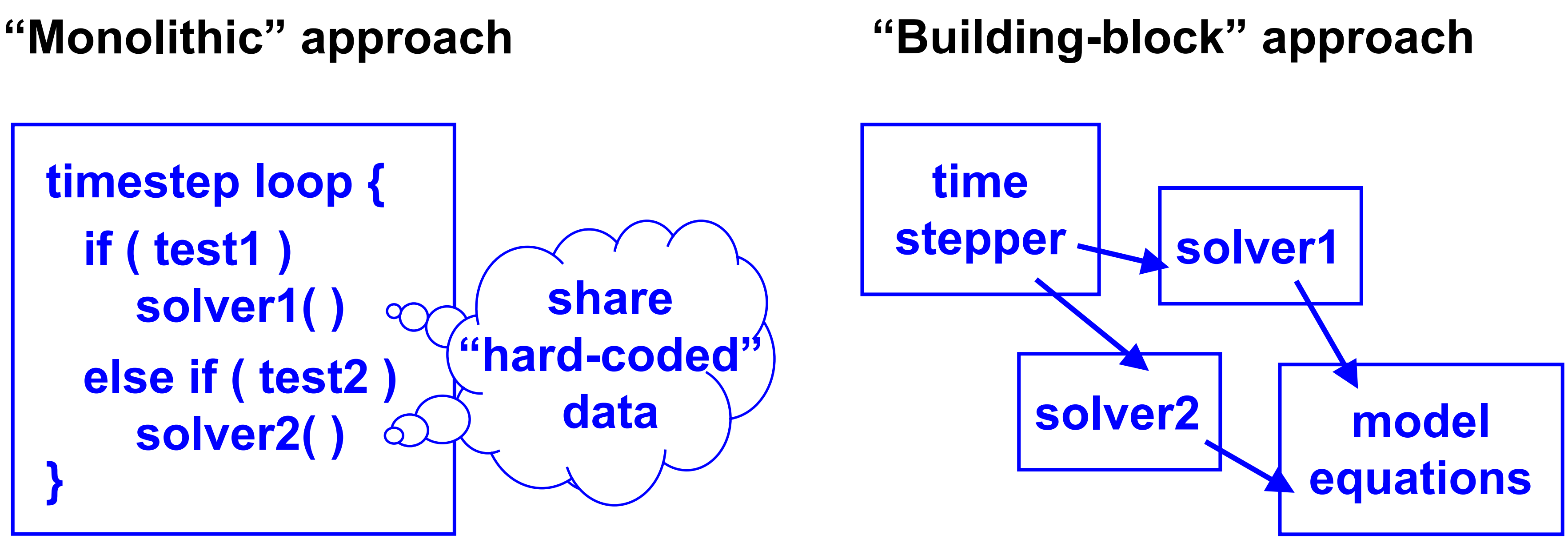
- **Limitations for high-performance scientific computing**
  - do not address issues of massively parallel components
  - industry focuses on abstractions for business (not scientific) data
  - typically unavailable on our parallel research platforms
  - lack of support for Fortran 77 and Fortran 90

- **However, we can leverage techniques and software**

Component technology extends OO with interoperability and common interfaces

- **Start with object-oriented technology**
- **Add language interoperability**
  - describe object calling interfaces independent of language
  - add "glue" software to support cross-language calls
- **Add common behavior, packaging, and descriptions**
  - all components must support some common interfaces
  - common tools (e.g., repositories, builders, …)
- **Component technology is *not*…**
  - object-oriented design, scripting, or frameworks
  - structured programming (e.g., modules)
  - the solution for all of your problems (just some of them)

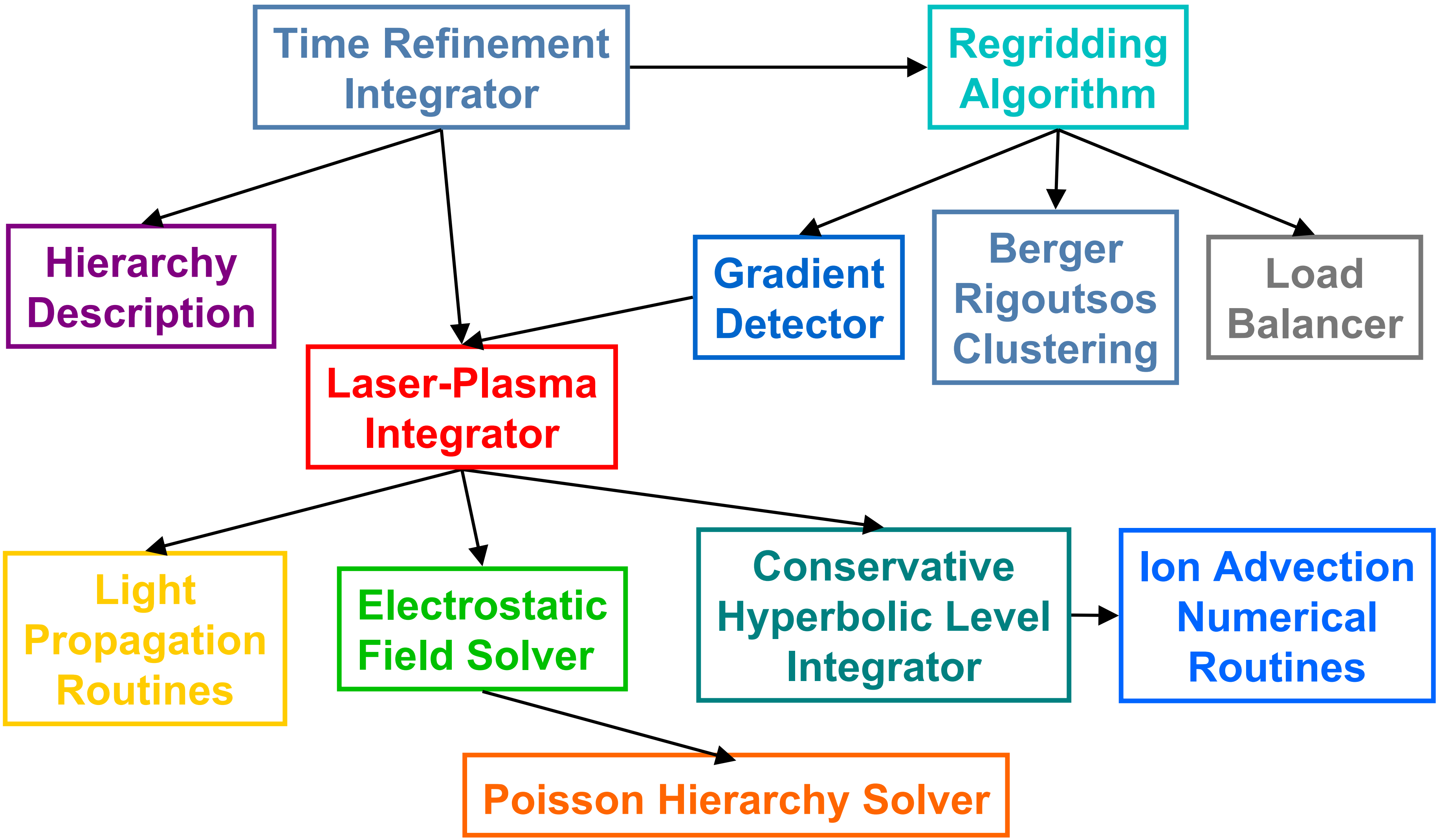
Component technology approaches help to manage application software complexity



"Monolithic" approach:
- tightly-coupled code
- less flexible, extensible
- re-use is difficult
- well-understood by community

"Building-block" approach:
- loosely-coupled code
- more flexible, extensible
- high re-use potential
- new to community

SAMRAI ALPS application combines three physics packages with local time refinement



CCA is investigating high-performance component technology for the DOE

- **Common Component Architecture (CCA) forum**
  - regular workshops and meetings since January, 1998
  - ANL, LANL, LBNL, LLNL, ORNL, SNL, Indiana, and Utah
  - `http://z.ca.sandia.gov/~cca-forum`
- **Goal: interoperability for high-performance software**
  - focus on massively parallel SPMD applications
  - modify industry approaches for the scientific domain
- **Writing specifications and reference implementation**
  - leverage technology developed by CCA participants
  - plan to develop a joint reference implementation by FY02

The CCA is researching a variety of component issues in scientific computing

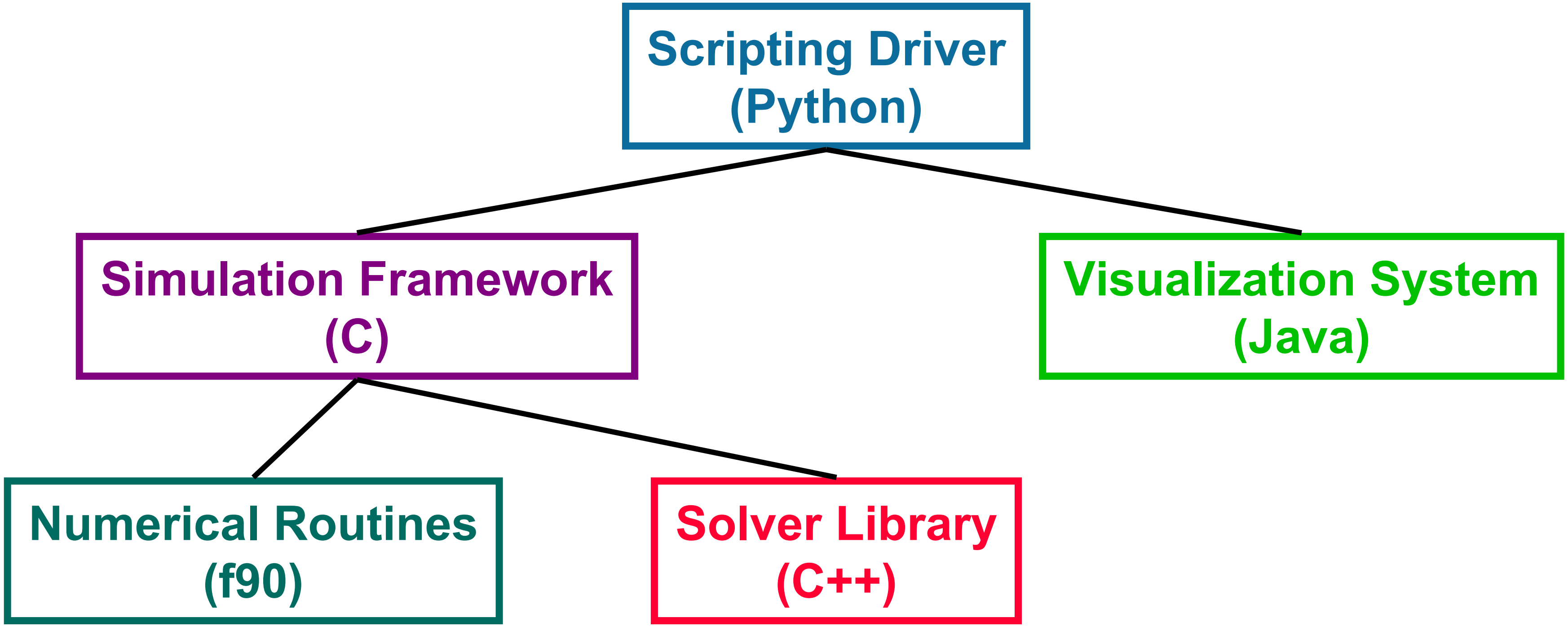
- Communication between components via ports
- Standard component repository formats and tools
- Composition GUIs
- Language interoperability technology
- Dynamic component loading
- Distributed components
- Parallel data redistribution between SPMD components
- Domain interface standards (e.g., solvers, meshes, …)
- Efficient low-level parallel communication libraries

Presentation outline

- Motivation
- DOE community activities (CCA)
- **Language interoperability technology (*Babel*)**
- Component type and software repository (*Alexandria*)
- Research issues in parallel component communication
- Conclusions

Motivation #1: Language interoperability

- **Motivated by Common Component Architecture (CCA)**
  - cross-lab interoperability of DOE numerical software
  - DOE labs use many languages (f77, f90, C, C++, Java, Python)
  - primary focus is on tightly-coupled same-address space codes

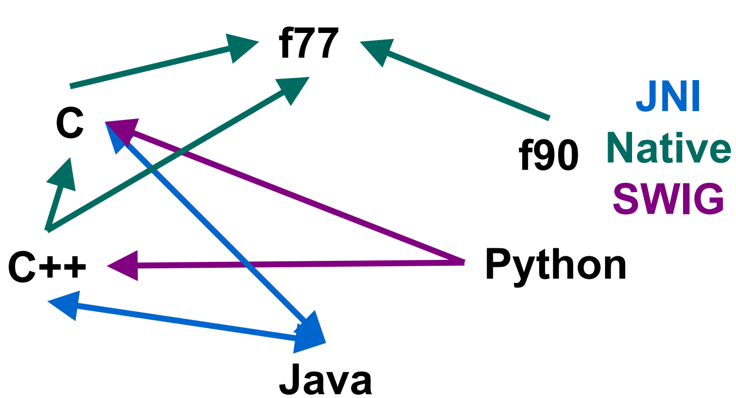


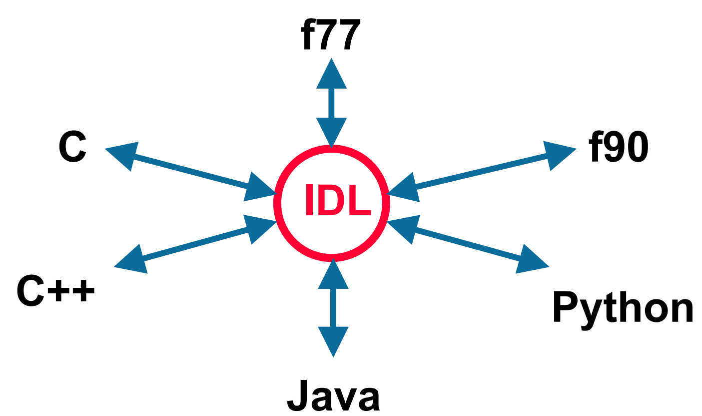
Motivation #2: Object support for non-object languages

- **Want object implementations in non-object languages**
  - object-oriented techniques useful for software architecture
  - but … many scientists are uncomfortable with C++
  - e.g., PETSc and *hypre* implement object-oriented features in C
- **Object support is tedious and difficult if done by hand**
  - inheritance and polymorphism require function lookup tables
  - support infrastructure must be built into each new class
- **IDL approach provides "automatic" object support**
  - IDL compiler automates generation of object "glue" code
  - polymorphism, multiple inheritance, reference counting, RTTI, ...

There are many tradeoffs when choosing a language interoperability approach

- **Hand generation, wrapper tools (e.g., SWIG), IDLs**
- **We chose the IDL approach to language interoperability**
  - goal: any language can call and use *any* other language
  - component tools need a common interface description method
  - sufficient information for automatic generation of distributed calls
  - examples: CORBA, DCOM, ILU, RPC, microkernel OSes





An IDL for scientific computing requires capabilities not present in industry IDLs

- **Industry standard IDLs: CORBA, COM, RPC, …**

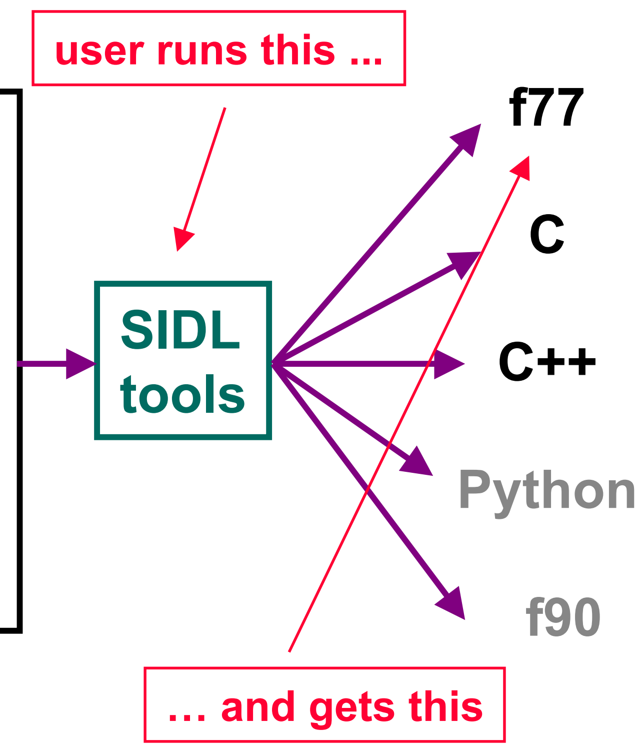
- **Desired capabilities for a scientific computing IDL**
  - attributes for parallel semantics
  - dense dynamic multidimensional arrays and complex numbers
  - bindings for f77/f90 and “special” languages (e.g., Yorick)
  - small and easy-to-modify IDL for research purposes
  - rich inheritance model (Java-like interfaces and classes)
  - high performance for same address-space method invocations

SIDL provides language interoperability for scientific components

- **SIDL is a "scientific" interface definition language**
  - we modified industry IDL technology for the scientific domain
  - SIDL describes calling interfaces (e.g., CCA specification)
  - our tools automatically generate code to "glue languages"

```
package ESI {
   interface Vector {
      void axpy(in Vector x, in double a);
      double dot(in Vector x);
      …
   };
   interface Matrix {
      …
   };
};
```

library writer develops this



SIDL incorporates ideas from Java and CORBA to describe scientific interfaces

```
version Hypre 0.5;
version ESI 1.0;

import ESI;

package Hypre {
   interface Vector extends ESI.Vector {
      double dot(in Vector y);
      void axpy(in double a, in Vector y);
   };
   interface Matrix {
      void apply(out Vector Ax, in Vector x);
   };
   class SparseMatrix implements Matrix, RowAddressible {
      void apply(out Vector Ax, in Vector x);
   };
};
```

class
enumeration
exception
interface
package

Users call automatically generated interface code completely unaware of SIDL tools

C++ Test Code

```
hypre::vector b, x;
hypre::matrix A;
hypre::smg_solver smg_solver;

b = hypre::vector::NewVector(com, grid, stencil);
...
x = hypre::vector::NewVector(com, grid, stencil);
...
A = hypre::matrix::NewMatrix(com, grid, stencil);
...

smg_solver = hypre::smg_solver::New();
smg_solver.SetMaxIter(10);
smg_solver.Solve(A, b, x);
smg_solver.Finalize();
```

Fortran 77 Test Code

```
integer b, x
integer A
integer smg_solver

b = hypre_vector_NewVector(com, grid, stencil)
...
x = hypre_vector_NewVector(com, grid, stencil)
...
A = hypre_matrix_NewMatrix(com, grid, stencil)
...

smg_solver = hypre_smg_solver_new()
call hypre_smg_solver_SetMaxIter(smg_solver, 10)
call hypre_smg_solver_Solve(smg_solver, A, b, x)
call hypre_smg_solver_Finalize(smg_solver)
```

SIDL version management

- **Simple version management scheme for SIDL types**
  - all symbols are assigned a fixed version number
  - SIDL `version` keyword requests specified version (or latest)
  - supports multiple versions of specs (e.g., ESI 0.5, ESI 0.5.1)

```
version ESI 0.5.1;   // access ESI spec v0.5.1
version HYPRE 0.7;   // define HYPRE spec v0.7

package HYPRE {
   // define v0.7 of HYPRE.Vector using v0.5.1
   // of the ESI.Vector interface
   interface Vector extends ESI.Vector {
      ...
   }
}
```

Language support in the *Babel* compiler

- **C, f77, C++ mostly finished using old SIDL grammar**
  - approximately 500 test cases probe implementation
  - used by *hypre* team for exploratory development
- **Currently migrating system to use new grammar**
- **Java, Python, and Yorick support next**
  - Python and Yorick are scripting languages (Yorick from LLNL)
  - hope to begin development in October timeframe
  - "should be quick" because of C interface support in languages
- **f90 and MATLAB will (hopefully) begin early next year**

We are collaborating with *hypre* to explore SIDL technology in a scientific library

- **Collaborators: Andy Cleary, Jeff Painter, Cal Ribbens**

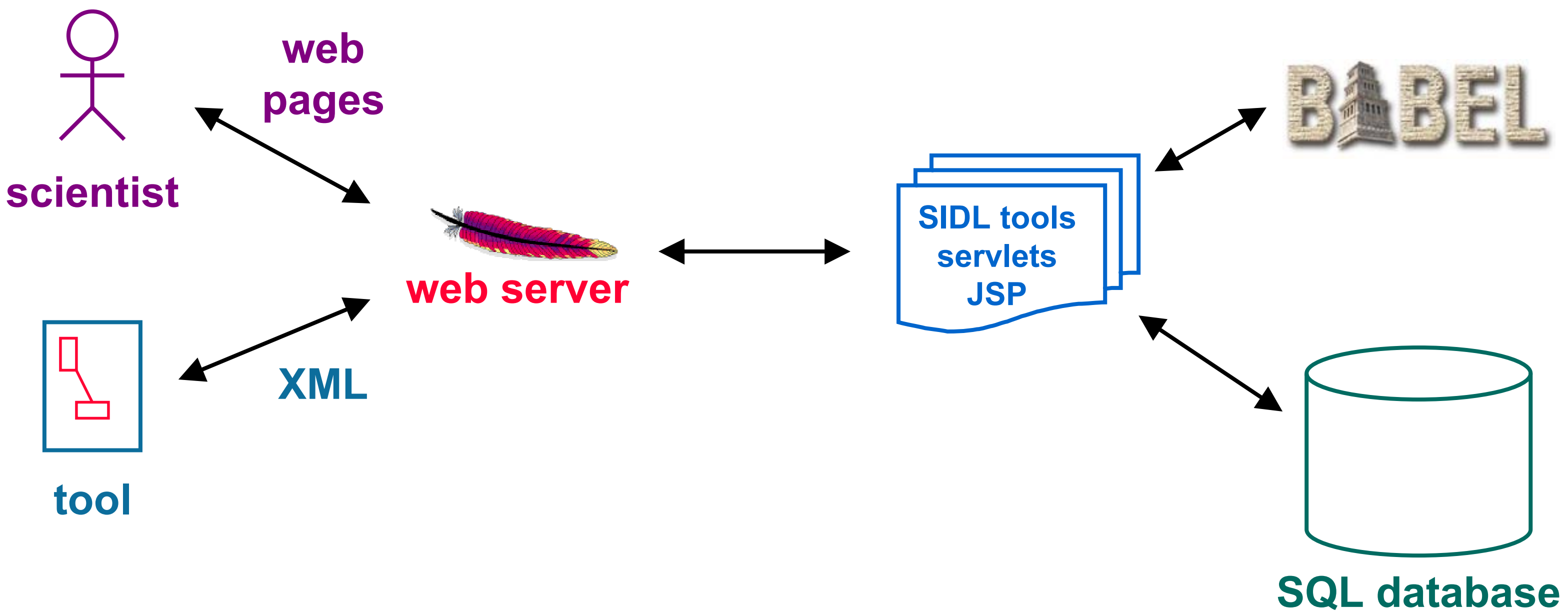
- **SIDL interface description file generated for *hypre***
  - approximately 30 interfaces and classes for *hypre* subset
  - use *Babel* tools to generate glue code for object support

- **Benefits of SIDL use in the *hypre* project**
  - automatic support for object-oriented features in C
  - Fortran capabilities through SIDL in upcoming version
  - plan to integrate existing C, Fortran, and C++ in one library
  - SIDL is a useful language for discussing software design
  - creating better *hypre* design based on SIDL OO support
  - cost overhead in same-address space too small to measure

Presentation outline

- Motivation
- DOE community activities (CCA)
- Language interoperability technology (*Babel*)
- **Component type and software repository (*Alexandria*)**
- Research issues in parallel component communication
- Conclusions

We are developing a web-based architecture to simplify access by scientists and tools

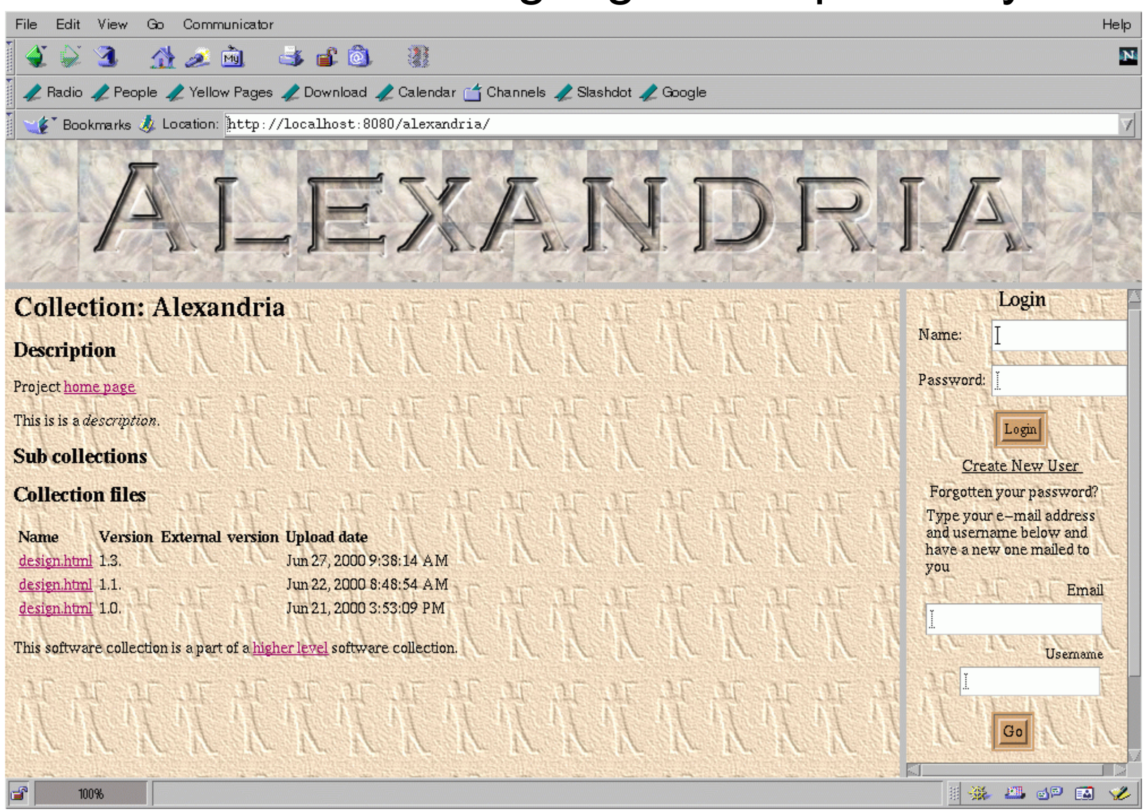


- Scientists and library developers must have easy access to our technology; otherwise, they simply will not use it

- Our web-based deployment lowers the “threshold of pain” to adopting component technology

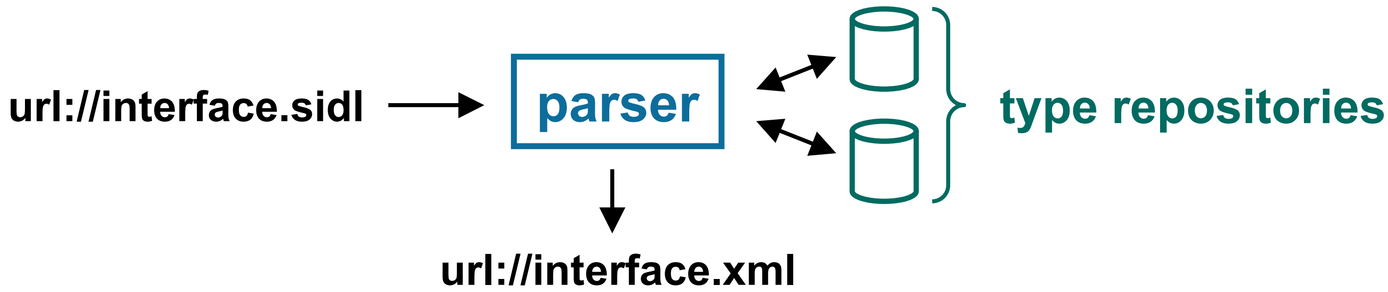
Alexandria is a web-based repository for component software and type descriptions

- **The Alexandria repository supports...**
  - SIDL type descriptions for libraries and components
  - library and component implementations
  - an interface to the Babel language interoperability tools

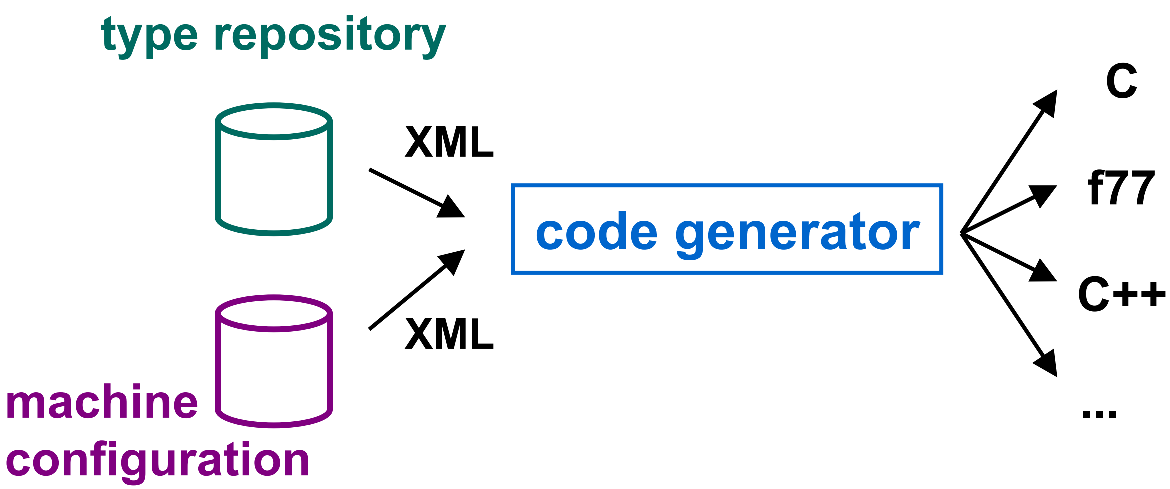


The *Babel* parser converts SIDL to XML that is stored in the *Alexandria* repository

- SIDL is used to generate XML interface information



- XML type description used to generate glue code



Sample XML file for *Hypre.Vector*

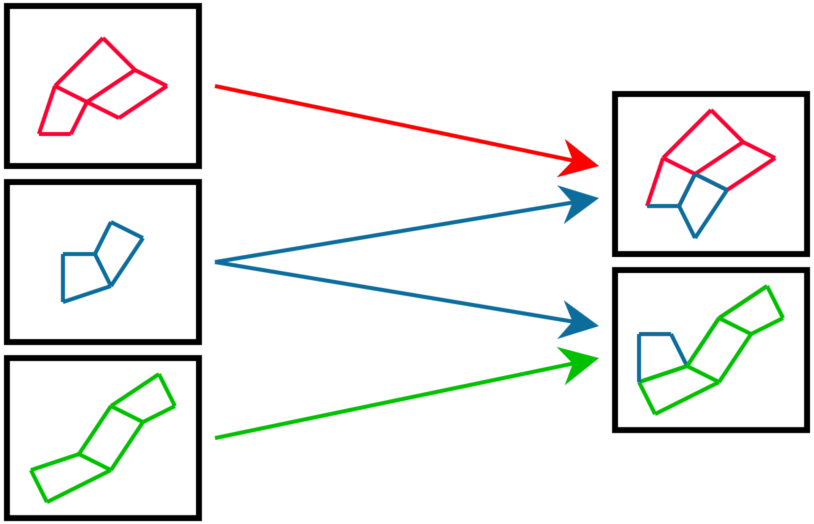
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Symbol PUBLIC "-//CCA//SIDL Symbol DTD v1.0//EN" "SIDL.dtd">
<Symbol>
   <SymbolName name="Hypre.Vector" version="1.0" />
   <Metadata date="20000816 08:47:22 PDT">
      <MetadataEntry key="source-url" value="file:/home/skohn/hypre.sidl" />
      ...
   </Metadata>
   <Comment />
   <Interface>
      <ExtendsBlock>
         <SymbolName name="Hypre.Object" version="1.0" />
      </ExtendsBlock>
      <AllParentInterfaces>
         <SymbolName name="SIDL.Interface" version="0.5" />
         <SymbolName name="Hypre.Object" version="1.0" />
      </AllParentInterfaces>
      <MethodsBlock>
         <Method communication="normal" copy="false" definition="abstract" name="Axpy">
            ...
         </Method>
      </MethodsBlock>
   </Interface>
</Symbol>
```

Presentation outline

- Motivation
- DOE community activities (CCA)
- Language interoperability technology (*Babel*)
- Component type and software repository (*Alexandria*)
- **Research issues in parallel component communication**
- Conclusions

Parallel redistribution of complex data structures between components

- **Parallel data redistribution for non-local connections**
  - example: connect parallel application to visualization server
  - cannot automatically redistribute complex data structures
  - *must support redistribution of arbitrary data structures*



- **Approach - modify SIDL and special interface support**

Parallel components will require special additions to SIDL interface descriptions

- **Special RMI semantics for parallel components**
  - provide a *local* attribute for methods
  - will also need a *copy* attribute for pass-by-value, etc.
  - *however, no data distribution directives - must be done dynamically*

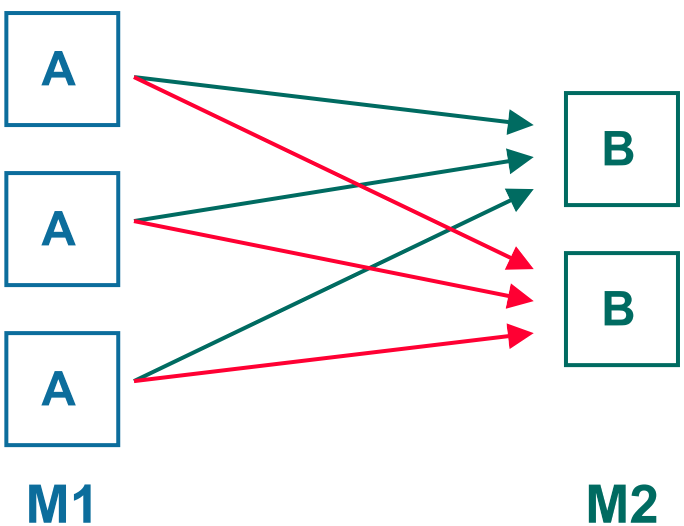
```
package ESI {
    interface Vector {
        double dot(copy in Vector v);
        int getGlobalSize();
        int getLocalSize() local;
    }
}
```

Dynamic redistribution of arbitrary data: Ask the object to do it for you!

- **Irregular data too complex to represent in IDL**
- **Basic approach:**
  - objects implement one of a set of redistribution interfaces
  - library queries object at run-time for supported method

```
interface ComplexDistributed {
   void serialize(in Array<Stream> s);
   void deserialize(in Array<Stream> s);
}
...
interface ArrayDistributed {
   // use existing array description
   // from PAWS or CUMULVS
}
```

two streams on M1
three streams on M2



Will component technology be part of the future of scientific computing?

- **Well, maybe - or maybe not**

- **Component technology does offer new capabilities**
  - techniques to manage large-scale application complexity
  - language interoperability and easier plug-and-play
  - leverage technology, not re-invent the wheel
  - bridges to interoperate with industry software (e.g., SOAP)

- **However, capabilities come at a price**
  - ties scientists to using component technology tools
  - steep learning curve (needs to become part of culture)
  - different paradigm for developing scientific applications

Acknowledgements

- Work performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.
- Document UCRL-VG-140549.