Language Interoperability Mechanisms for High-Performance Scientific Applications

Scott Kohn

with

Andrew Cleary, Steven G. Smith, and Brent Smolinski

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

October 21, 1998

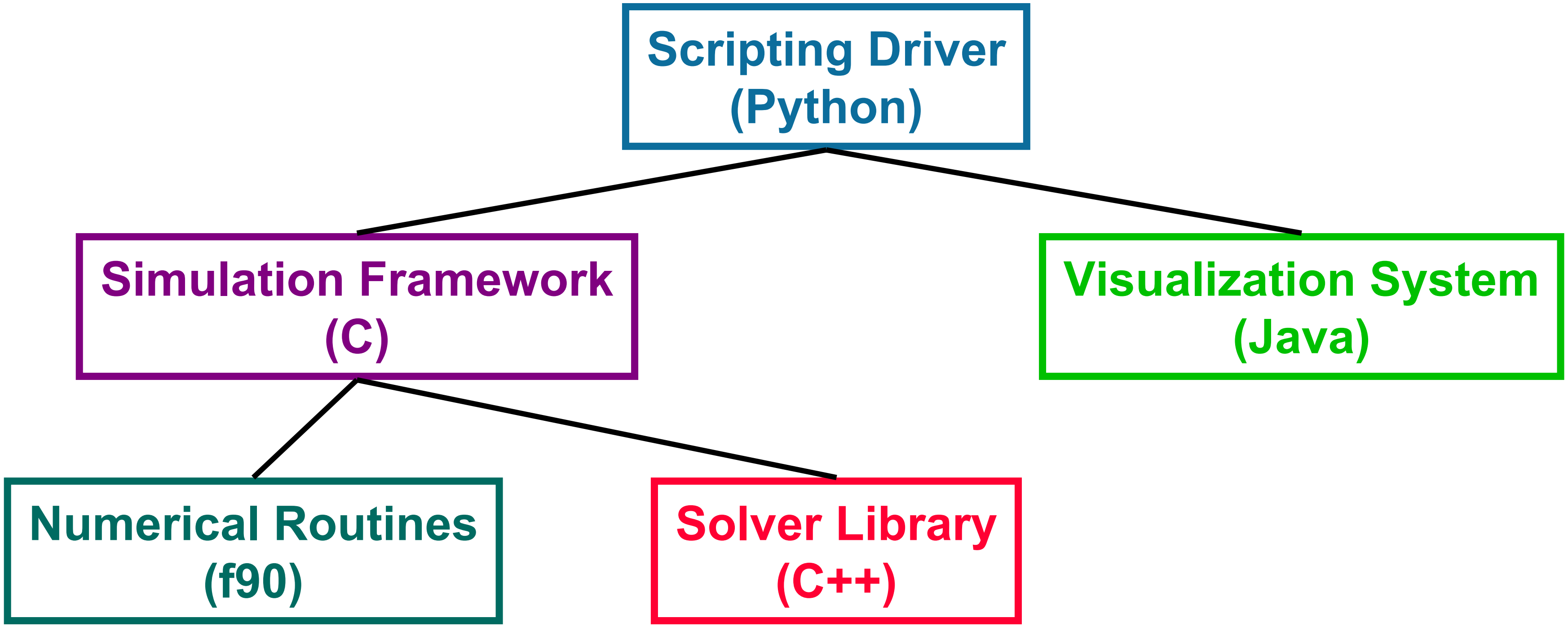
Overview

<u>Goal</u>: Apply IDL interoperability technology to problems in high-performance parallel scientific computing

- Motivation
- Traditional interoperability mechanisms
- Review of IDL technology
- Using an IDL for interoperability in scientific computing
  - what is a “Scientific IDL”
  - Fortran issues
  - performance considerations
- Analysis and 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)
  - language should not be a barrier to software reuse

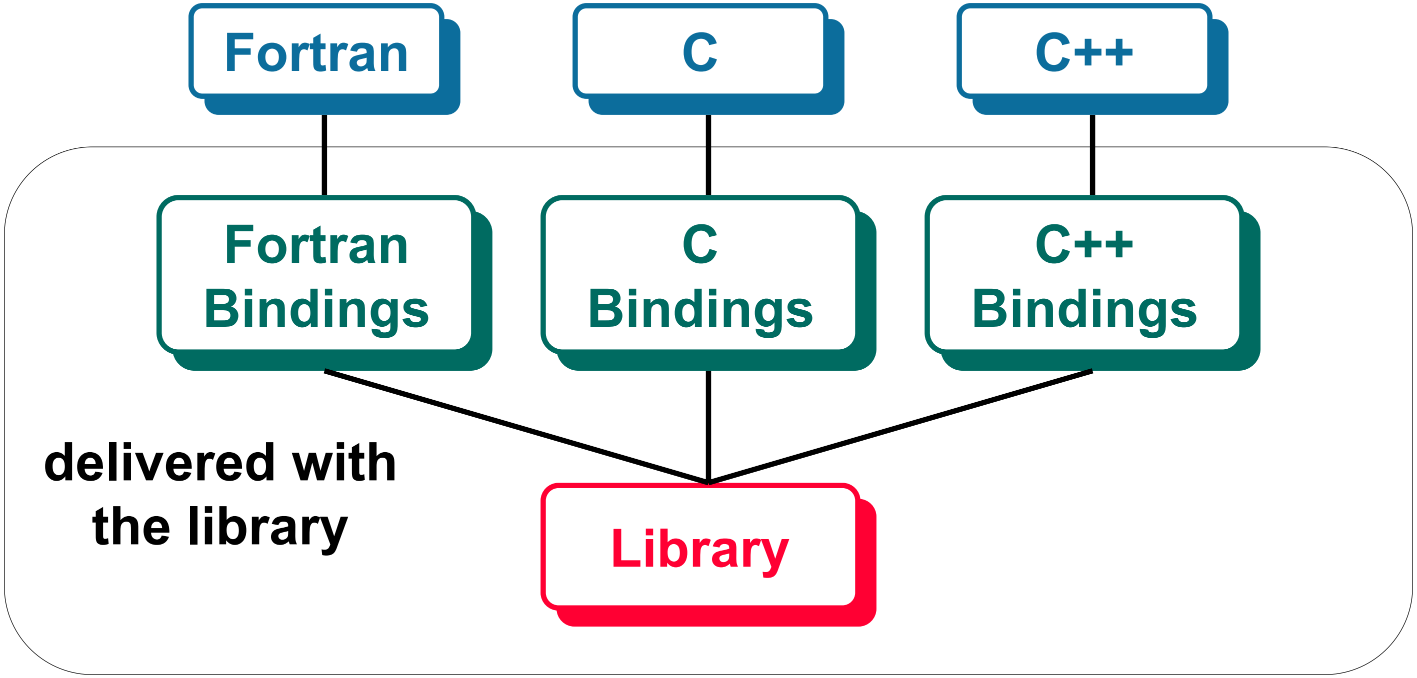


Motivation #2: Object support for non-object languages

- Want object implementations in non-object languages
  - object-oriented techniques useful for software architecture
  - but … many scientists are uncomfortable with C++
  - e.g., PETSc library implements 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
  - introspection, RTTI, simple exception mechanism

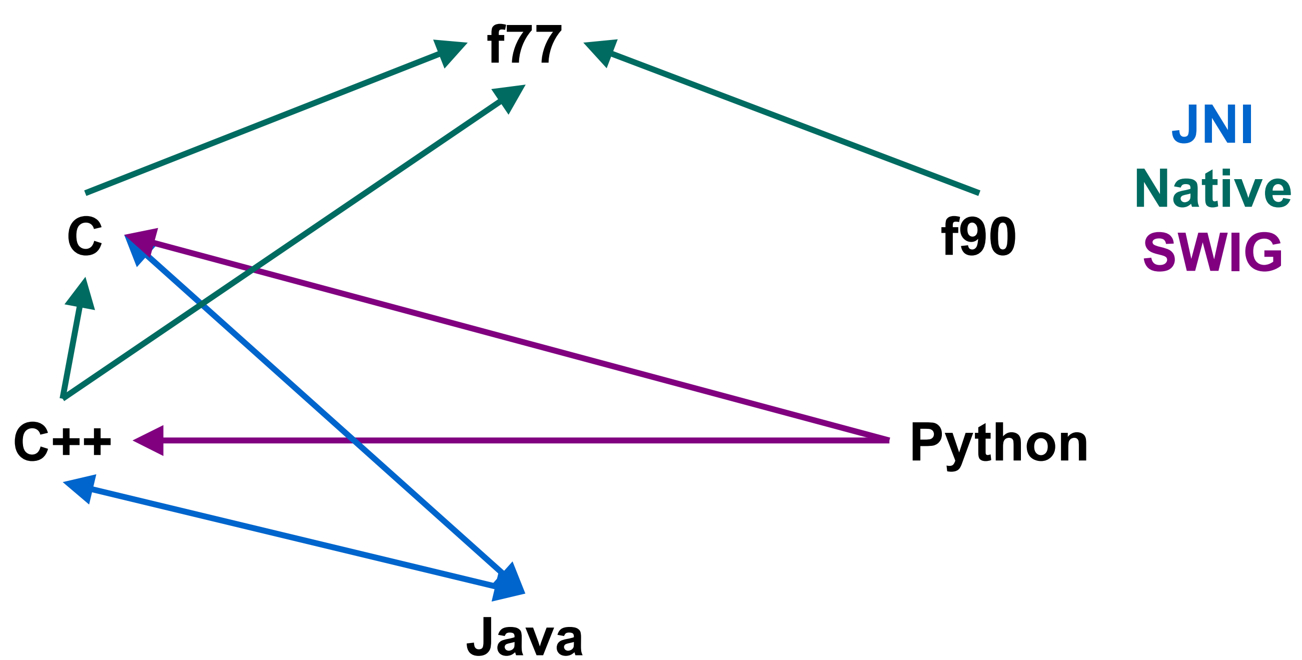
Interoperability through language bindings

- Generate custom bindings for each language
  - labor-intensive to generate bindings for supported languages
  - can tailor binding to style and conventions of language
  - approach taken in the MPI standardization effort



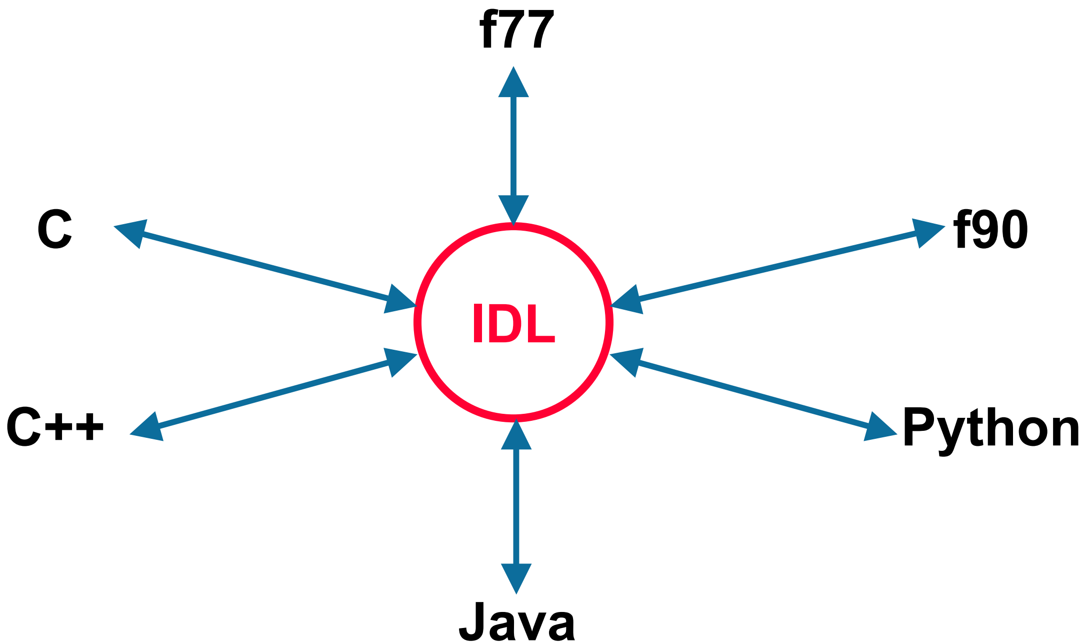
Interoperability through custom solutions

- Techniques that target small collections of languages
  - foreign call interfaces or automatic glue code generators
  - e.g., SWIG wraps C and C++ for calls from scripting languages
  - not very general - limited only to subsets of languages



Interoperability through an IDL

- Describe objects in an "interface definition language"
  - each language interoperates with the "IDL language"
  - IDL compiler generates "glue" that wraps components
  - examples: CORBA, DCOM, ILU, RPC, microkernel OSes

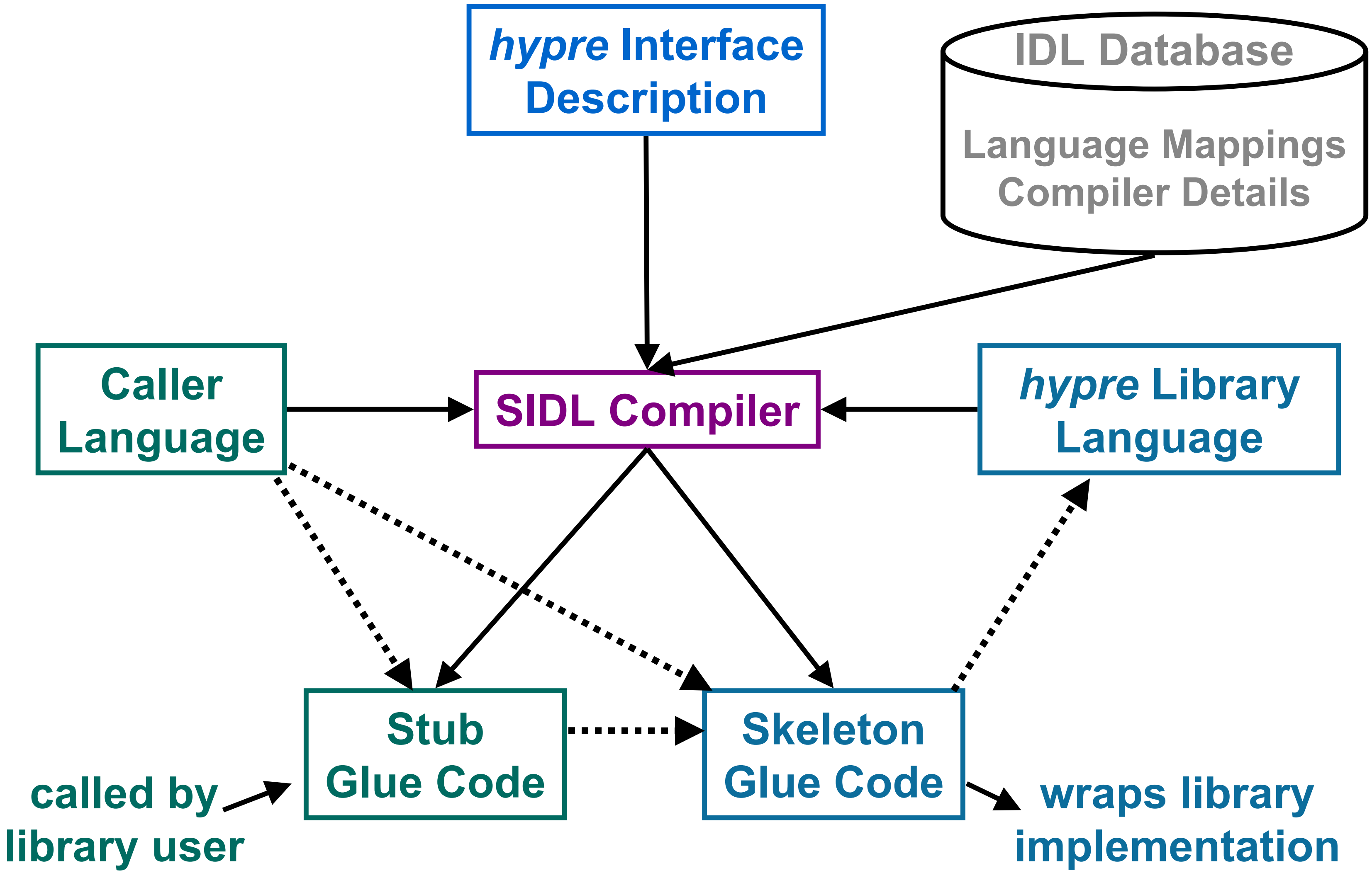


IDL: Interface Definition Language

- **Declarative interface description language**
  - describes interface only (no implementation details)
  - language-independent way to describe object APIs

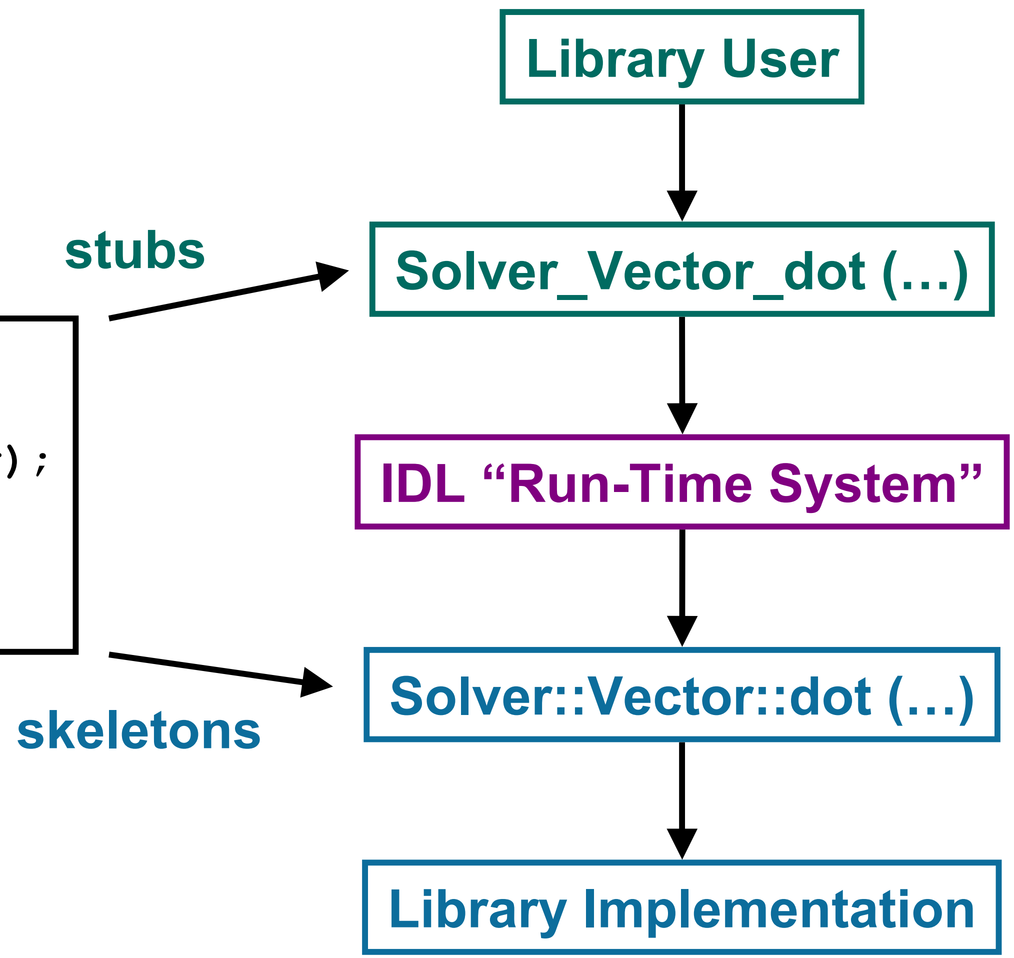
```
package Solver {
   interface Vector {
      Vector clone();
      double dot(in Vector y);
      void axpy(in double a, in Vector y);
      …
   };
   interface Matrix {
      void apply(out Vector Ax, in Vector x);
   };
   ...
   class SparseMatrix implements Matrix {
      void apply(out Vector Ax, in Vector x);
   };
);
```

Compiler generates “glue” code from IDL



IDL glue code bridges languages

```
package Solver {
   interface Vector {
      double dot(in Vector y);
      ...
   };
};
```



Calling the IDL stubs from user code

C++

```
double error(Solver::Matrix A, Solver::Vector x)
{
   Solver::Vector r = x.clone();
   A.apply(r, x);
   return(r.dot(r));
}
```

C

```
double error(Solver_Matrix A, Solver_Vector x)
{
   Solver_Vector r = Solver_Vector_clone(x);
   Solver_Matrix_apply(A, r, x);
   double result = Solver_Vector_dot(r, r);
   Solver_Vector_delete(r);
   return(result);
}
```

f77

```
double precision function error(A, x)
integer A, x, r
r = Solver_Vector_clone(x)
call Solver_Matrix_apply(A, r, x)
error = Solver_Vector_dot(r, r)
call Solver_Vector_delete(r)
return
end
```

types
methods

Design goals for the interoperability of high-performance scientific software

- **High-performance**
  - no data copies (means single address-space)
  - overhead of a C++ virtual function call (or maybe a few)
  - works in either a threaded or MPI environment
- **Language support**
  - C, C++, Fortran 77/90 in “high-performance” mode
  - Java and Python with maybe a little more overhead
  - investigate support for prototyping tools like MatLab
- **Expressibility for IDL**
  - sufficiently general to express most scientific interfaces
  - must be object oriented and support error mechanisms

Research issues for scientific interoperability

- Leverage existing technology where appropriate
  - CORBA and ILU IDLs and language mappings
  - Java inheritance and introspection ideas
- Research issues
  - what features are needed in a “Scientific IDL”
  - mapping the “Scientific IDL” onto Fortran
  - performance and overheads in the run-time system
- Basic research approach in our project
  - prove a simple prototype can work in a scientific environment
  - then … add new features (e.g., distributed computation)

What our “Scientific IDL” looks like

- Start with the CORBA IDL
  - object oriented, exceptions, namespaces
  - syntax similar to Java and C++
  - mappings to all languages of interest but Fortran
- Eliminate (for now, anyway) unnecessary features
  - one-way qualifier
  - struct and union (performance and Fortran considerations)
  - CORBA arrays and sequences (replaced - see below)
- Add new data types and fix stupid things
  - CORBA inheritance model broken - adopt Java
  - add complex type and dynamic multidimensional arrays
  - add static and final qualifiers for methods
- The new IDL looks a lot like Java

IDL specification for a linear solver package

```
package Solver {
   interface Vector {
      Vector clone();
      double dot(in Vector y);
      void axpy(in double a, in Vector y);
      void initialize(in array<double,1> data);
   };
   interface Matrix {
      void apply(out Vector Ax, in Vector x);
   };
   class SparseMatrix implements Matrix, RowAddressible {
      void apply(out Vector Ax, in Vector x);
   };
   class CG extends Krylov implements AbstractSolver {
      void solve(in Matrix A,...) throws ConvergenceException;
   };
);
```

class
exception
interface
package

Evaluation of IDL expressibility

- **Sufficiently expressive for some numerical packages**
  - *hypre*, KINSOL, structured AMR linear solvers interface
  - looked at some parts of the PETSc package
- **Type system limited to types expressible in the IDL**
  - no pointers and currently no templates
  - problems with external opaque objects (e.g., `MPI_Comm`)
  - but … only high-level interfaces will be expressed in the IDL
- **More than just the intersection of language capabilities**
  - features can be supported through the run-time system
  - e.g., object-oriented support in C or f77
  - e.g., introspection, RTTI, reference counting, exceptions …

Language mapping issues

- Fortran 77/90 are the only real problems
  - C, C++, Java mappings are defined by CORBA
  - Python mappings are defined by ILU
  - new constructs do not add to complexity of language mapping
- Fortran 77 should be OK
  - follow C bindings for function names
  - opaque object references become integers (as in MPI)
  - no structs or unions needed (or can be mapped to objects)
- Fortran 90 will be major pain …
  - can use f77 mapping and not exploit expanded type system
  - want IDL arrays to map onto Fortran 90 arrays
  - calling sequences and array descriptors compiler-dependent
  - but … doing this by hand would be almost impossible

Performance and the run-time system

- Planned run-time system is very simple
  - single address space (perhaps distributed later)
  - IDL design means no data copies (perhaps array transpose)
  - polymorphism requires function tables as in C++ or PETSc
  - overhead is a few function calls (could be inlined away)
- So what do we need in a run-time system?
  - implementation of vtables for Java inheritance model
  - up/down casting and introspection queries
  - object reference counting
  - exception support as in CORBA (really just error return codes)
  - other miscellaneous stuff (complex numbers in C, etc.)

Project status and future development

- **Current implementation status**
  - parser/analyzer written in Java with JavaCC
  - prototype for polymorphism/multiple inheritance support in C
  - integrating prototype with parser stub generator

- **Future development plans**
  - prove the technology can work in C
  - demonstrate interoperability for C, C++, and f77
  - develop language mappings for f90
  - add language mappings for Java, Python, and MatLab
  - add distributed capabilities (borrow existing technology)

Analysis and conclusions

- Advantages of the IDL approach
  - language interoperability (even for Fortran 90!)
  - automatic support for object oriented features in C
  - expressive enough for many numerical libraries (?)
  - IDL provides a nice description language for interface
  - automatically generates type information for introspection
- Disadvantages and potential problems
  - library designers write the IDL description (but … it's simple)
  - language mapping may not be as "natural" as if by hand
  - types are limited to the IDL type system
- What are the alternatives?

Acknowledgements

- Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.
- Document UCRL-MI-131823