



# Tutorial S08:

---

# Introductory for Massive Supercomputing Software Integration.

Gary Kumfert & Thomas Epperly



This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

UCRL-PRES-234812

# Today's Tutorial

---

- **a.m. Lecture**

- ▶ **Babel – Our Software Integration Tool**
- ▶ **SIDL – Our Universal Type System**
- ▶ **RMI – Remote Method Invocation**
- ▶ **Real World Applications**
- ▶ **Open Babel Community**

- **p.m. Hands-On**

- ▶ **On your laptop (Linux, MacOSX)**
- ▶ **Guest Accounts hosted by Indiana Univ.**

# Each Instructor Has >7 Years Experience Designing & Using Babel

---



Gary Kumfert  
<kumfert@llnl.gov>



Thomas Epperly  
<tepperly@llnl.gov>

- **Ph.D. in Computer Science & B.S. Applied Math**
- **Used/Uses Babel in**
  - ▶ Material Science
  - ▶ Graph Partitioning
  - ▶ CCA Frameworks

- **Ph.D. & B.S. in Chemical Engineering**
- **Used/Uses Babel in**
  - ▶ Fusion Research
  - ▶ Source Code Analysis
  - ▶ CCA Frameworks

# General Announcements

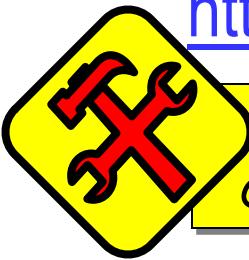
---

- Restrooms
- Break schedule
- You should have:
  - ▶ SC Tutorial Questionnaire
  - ▶ Babel Slides (including this one)
  - ▶ Personal Laptop w/ access to UNIX-like environment (for Part II: Hands-on)
- Please Ask Questions
  - ▶ Lot of arcana goes into interoperability

# Details about Slides

- **Many Slides are labeled**
  - ▶ These are “hidden slides” in the presentation (intended for readers)
  - ▶ Speakers will skip these slides by default.
  - ▶ But, we are happy to visit them if there are any questions.
- **Glossary of Terms in back of Handouts.**
- **We will post Errata at**

<http://www.llnl.gov/CASC/components/docs/sc07.html>



This symbol is used to warn about corrections made after publication of notes.

# a.m. Lecture ~ 3 hours

---

## I. Lecture - 3 hours

**Introduction – Gary Kumfert**

**SIDL Language – Thomas Epperly**

**Babel Tool – Thomas Epperly**

**Using Babel Objects – Thomas Epperly**

**Building Babel Libraries – Gary Kumfert**

**Distributed Computing & RMI – Gary Kumfert**

**Closing – Gary Kumfert**

## II. Hands On – 3 hours

---

---

# I. Introduction

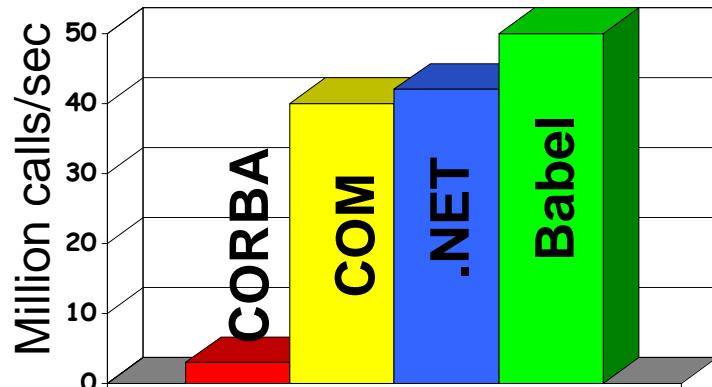
15 minute “Manager Overview”

# Babel is Software Integration Technology for HPC



“The world’s most rapid communication among many programming languages in a single application.”

Performance (in process)



	CORBA	COM	.NET	Babel
BlueGene, Cray, Linux, AIX, & OS X	No	No	No	Yes
Fortran	No	Limited	Limited	Yes
Multi-Dim Arrays	No	No	No	Yes
Complex Numbers	No	No	No	Yes
Licensing	Vendor Specific	Closed Source	Closed Source	Open Source

# Babel Is Used in Many Domains

Some Known Babel Users

>200 downloads  
Babel source  
per month (avg.)

Application	Project	POC
Chemistry	NWChem	Theresa Windus, Iowa State
Quantum Chemistry	MPQC	Curtis Janssen, Sandia
Chemistry	GAMESS-CCA	Masha Sosonkina, Ames Lab
Fusion	FMCFM	Johann Carlsson, Tech-X Corp.
Fusion	DFC	Nanbor Wang, Tech-X Corp.
Fusion	FACTS	Tom Epperly, LLNL
Electron Effects	CMEE	Peter Stoltz, Tech-X Corp.
Component Frameworks	CCA	David Bernholdt, ORNL
Programming Models	Co-Op	John May, LLNL
Performance Monitoring	TAU	Sameer Shende, U Oregon
Meshing	TSTT/ITAPS	Lori Diachin, LLNL
Sparse Linear Algebra	Sparsekit-CCA	Masha Sosonkina, Ames Lab
Solvers	TOPS	Barry Smith, Argonne
Grid Programming	Legion-CCA	Michael J. Lewis, Binghamton University
MOCCA	Harness	Vaiday Sunderam, Emory University
Programming Models	Global Arrays	Jarek Nieplocha, PNNL
Cell Biology	VMCS (using TSTT)	Harold Trease, PNNL
Computational Mechanics	DLSMM	Nathan Barton, LLNL
Nuclear Power Plant Trainer		M. Diaz, U. Malaga, Spain
Subsurface Transport	PSE Compiler	Jan Prins, UNC Chapel Hill
Radio Astronomy	eMiriad	Athol Kemball, UIUC
Solvers	Hypre	Jeff Painter, LLNL
Source Code Transformation	CASC	Dan Quinlan, LLNL
Distributed Component Env.	SCIJump (fka SCIRun2)	Steve Parker, Utah

# **Customers Use Babel To Serve a Variety of Needs**

---

- **Manage community codes**  
[Chemistry, Fusion, Radio Astronomy]
- **Create software interface specifications.**  
[CCA, ITAPS, TOPS]
- **Integrate multiple 3<sup>rd</sup> party libraries into a single scientific application.**  
[Chemistry, Fusion, CMEE]
- **Develop libraries that connect to multiple languages.**  
[hypre, TAU, Sparsekit-CCA]
- **Scientific Distributed Computing**  
[Co-Op, SCIJump, Legion-CCA, Harness, GA]

# Customers Use Babel To Serve a Variety of Needs

- Manage community codes  
[Chemistry, Fusion, Radio Astronomy]

“...the nominal collective commercial replacement cost for this sampling of community codes [AIPS, MIRIAD, AIPS++] is of order ~480 person-years.”

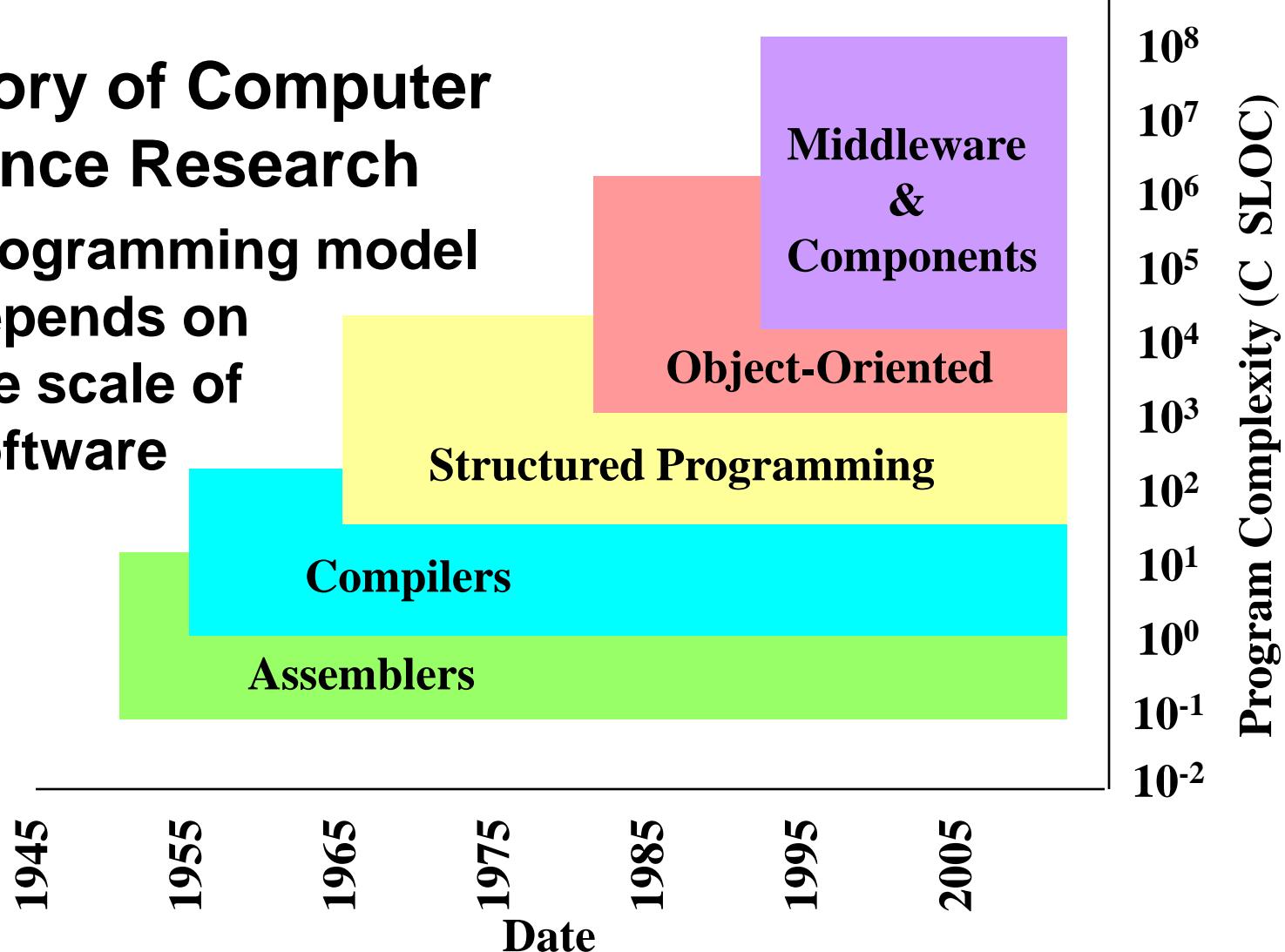
“In this case study, Babel has proven to be a good component middleware choice. It is well suited to scientific problem domains, such as radio astronomy imaging, due to the support for multi-dimensional arrays, FORTRAN bindings, good interoperability with HPC, and peer-to-peer language bindings. The latter property is particularly useful in providing developer choice in component implementation and in providing a general scripting interface using Python.”

A. J. Kemball, R. M. Crutcher, R. Hasan. “*Component-Based Framework for Radio-Astronomical Imaging*,” Software Practice & Experience, Wiley, 2007

# **Creating software is a human activity, not a scalable process.**

- **History of Computer Science Research**

- ▶ Programming model depends on the scale of software



# In Industry, All Enterprise Software uses Middleware

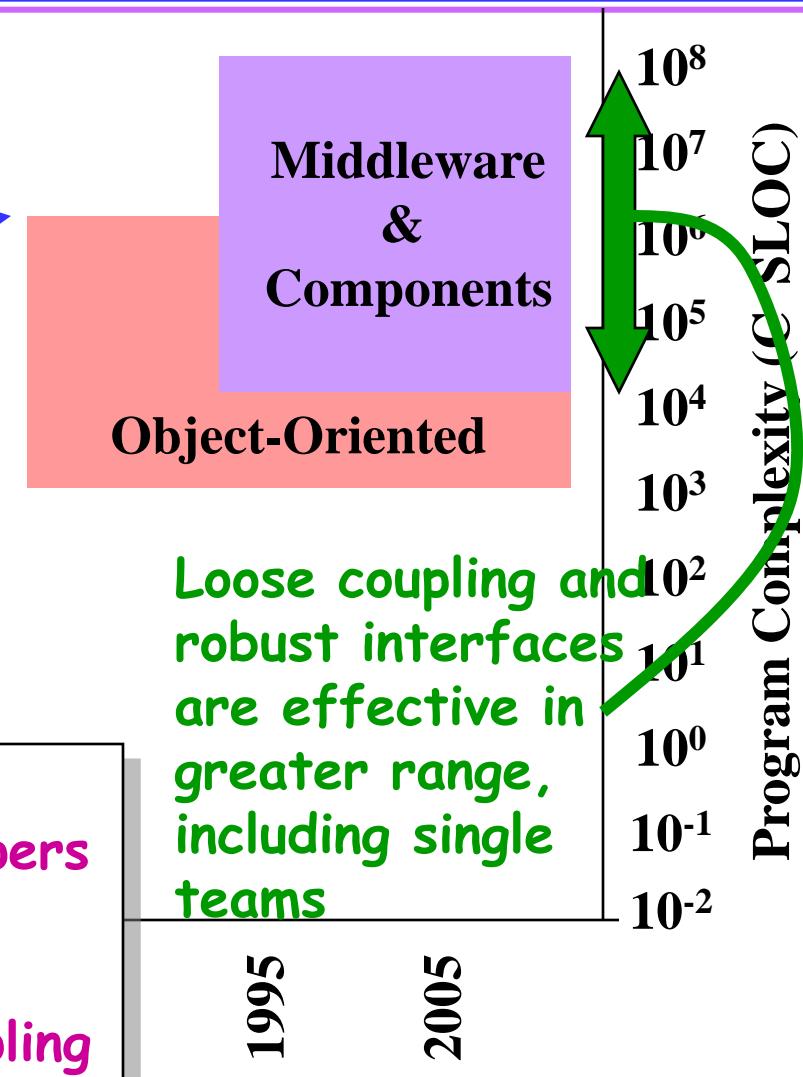
Invented for codes where complexity exceeds the comprehension of a single human mind

OOP falls down because

1. Assumes a single language
2. Implementation details pollute the interfaces

Middleware adds

1. Code generation (language wrappers & stronger interfaces)
2. Additional runtime services to support dynamicism & loose coupling



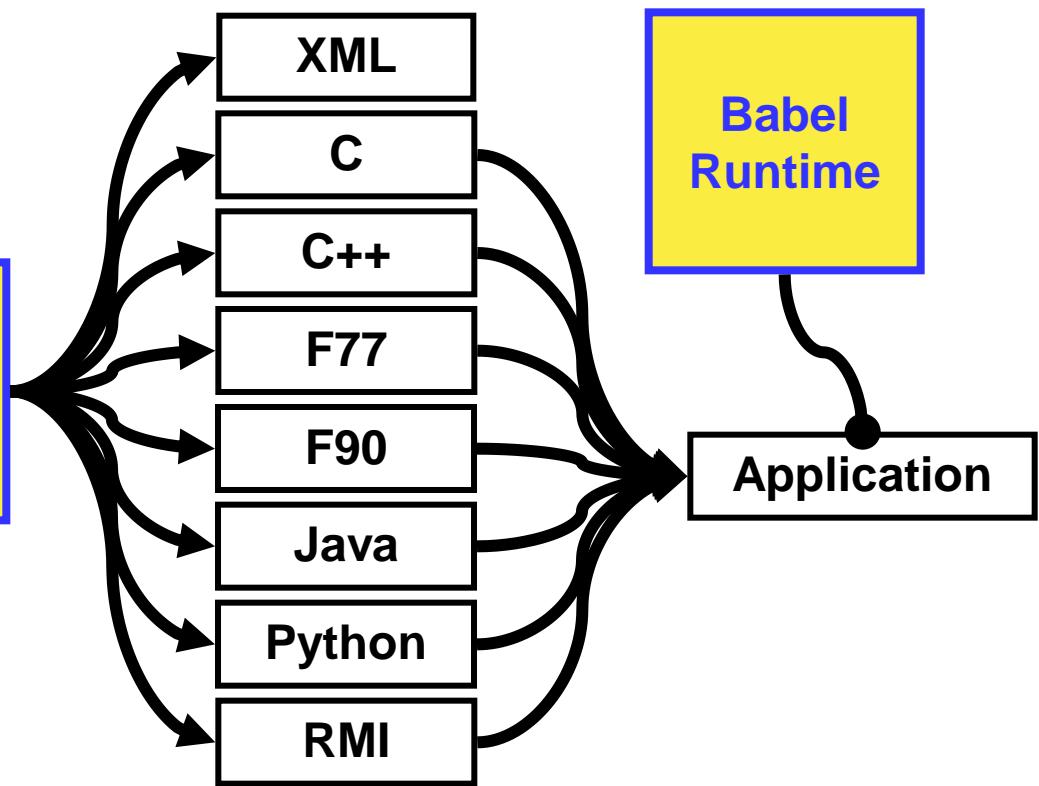
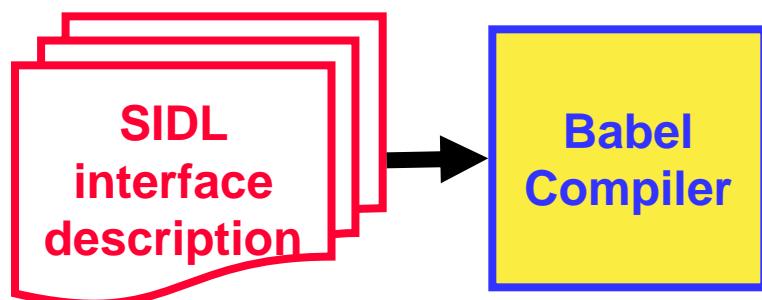
# Customers Use Babel To Serve a Variety of Needs

---

- Manage community codes  
[Chemistry, Fusion, Radio Astronomy]
- **Create software interface specifications.**  
[CCA, ITAPS, TOPS]
- Integrate multiple 3<sup>rd</sup> party libraries into a single scientific application.  
[Chemistry, Fusion, CMEE]
- Develop libraries that connect to multiple languages.  
[hypre, TAU, Sparsekit-CCA]
- Scientific Distributed Computing  
[Co-Op, SCIJump, Legion-CCA, Harness, GA]

# Babel Has Two Parts: Code Generator & Runtime Library

Specifications are  
written in SIDL



# Tutorial Sections Follow this Flow

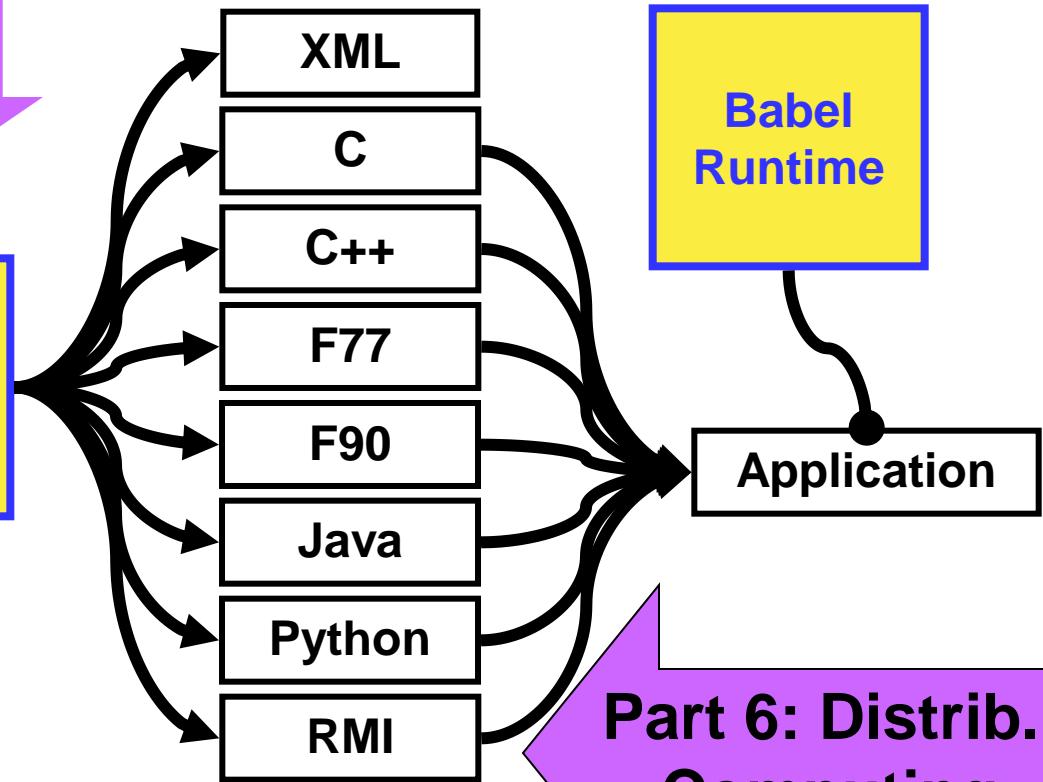
Parts 4 & 5:  
Using Objects  
& Building Libs

Part 2:  
SIDL  
Language

Part 3:  
Babel  
Tool

SIDL  
interface  
description

Babel  
Compiler



Part 6: Distrib.  
Computing

# Customers Use Babel To Serve a Variety of Needs

---

- Manage community codes  
[Chemistry, Fusion, Radio Astronomy]
- Create software interface specifications.  
[CCA, ITAPS, TOPS]
- **Integrate multiple 3<sup>rd</sup> party libraries into a single scientific application.**  
[Chemistry, Fusion, CMEE]
- Develop libraries that connect to multiple languages.  
[hypre, TAU, Sparsekit-CCA]
- Scientific Distributed Computing  
[Co-Op, SCIJump, Legion-CCA, Harness, GA]

# Example: Babel Used To Design Plasma Thrusters

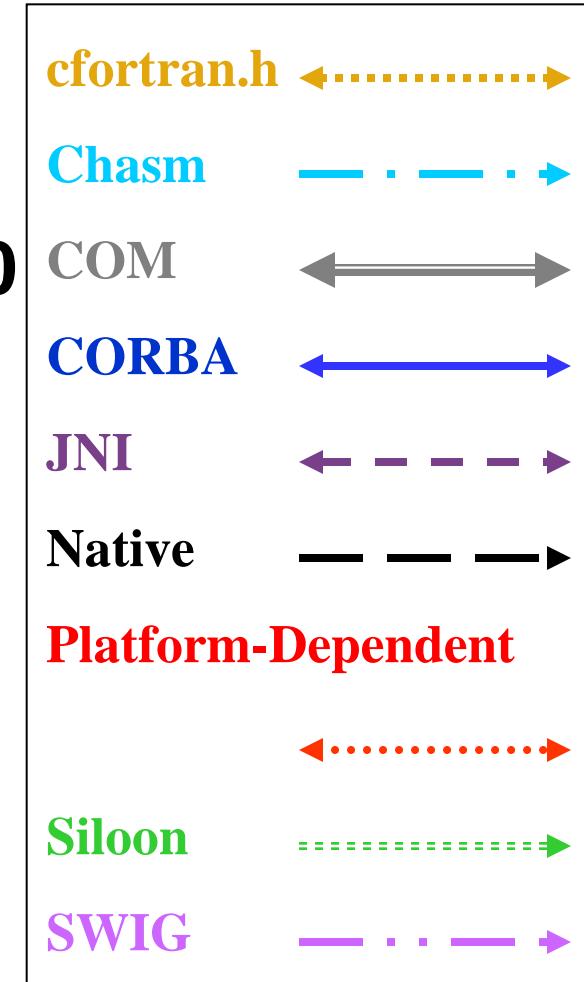
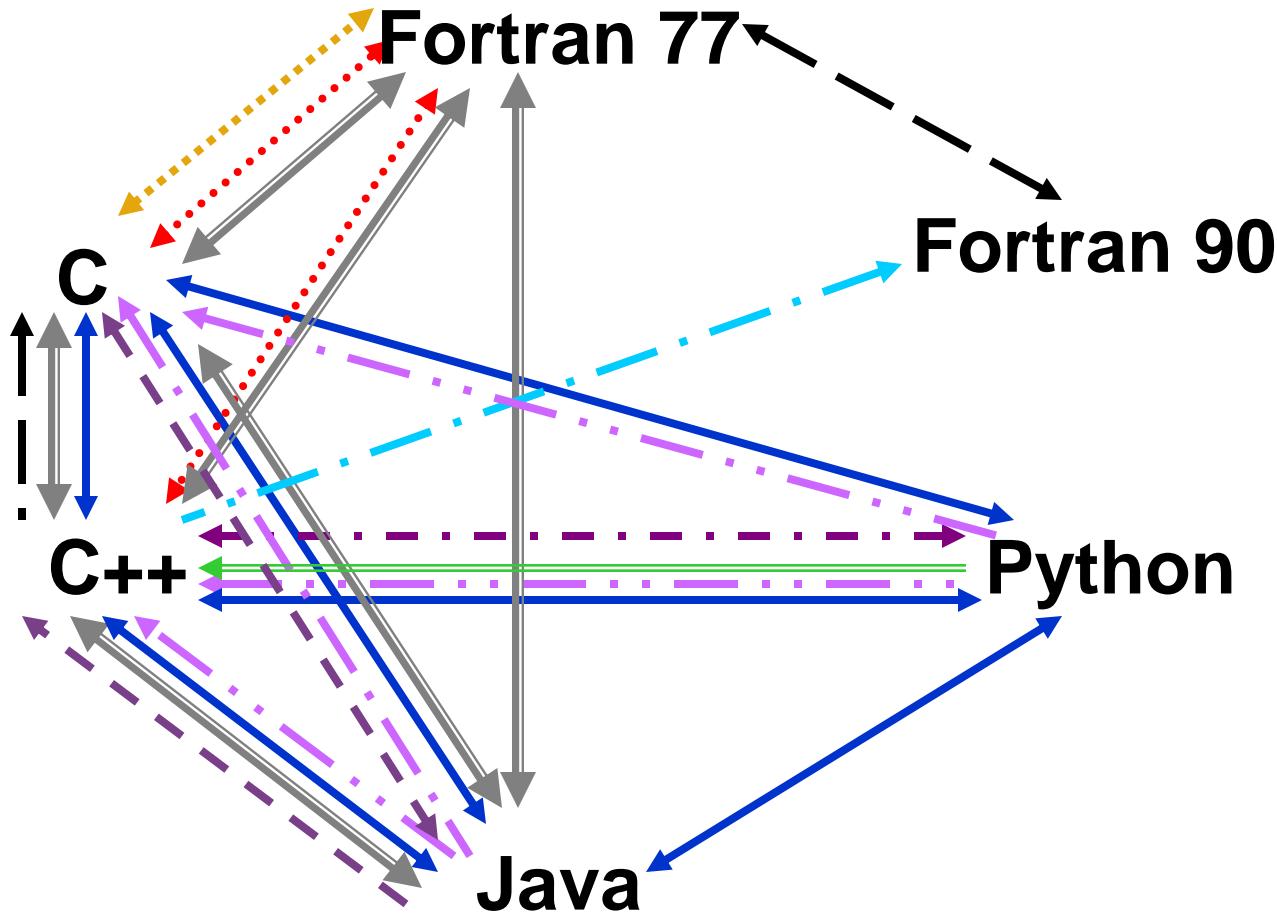
CMEC takes widely-used physics routines for modeling electron effects like gas ionization and secondary electron emission from metals and uses Babel to make them widely available. The resulting code is used in applications such as accelerator physics and plasma drives for satellites.

In addition to having legacy codes in Fortran 77, they also integrate new codes in Fortran 90, C, and Python.

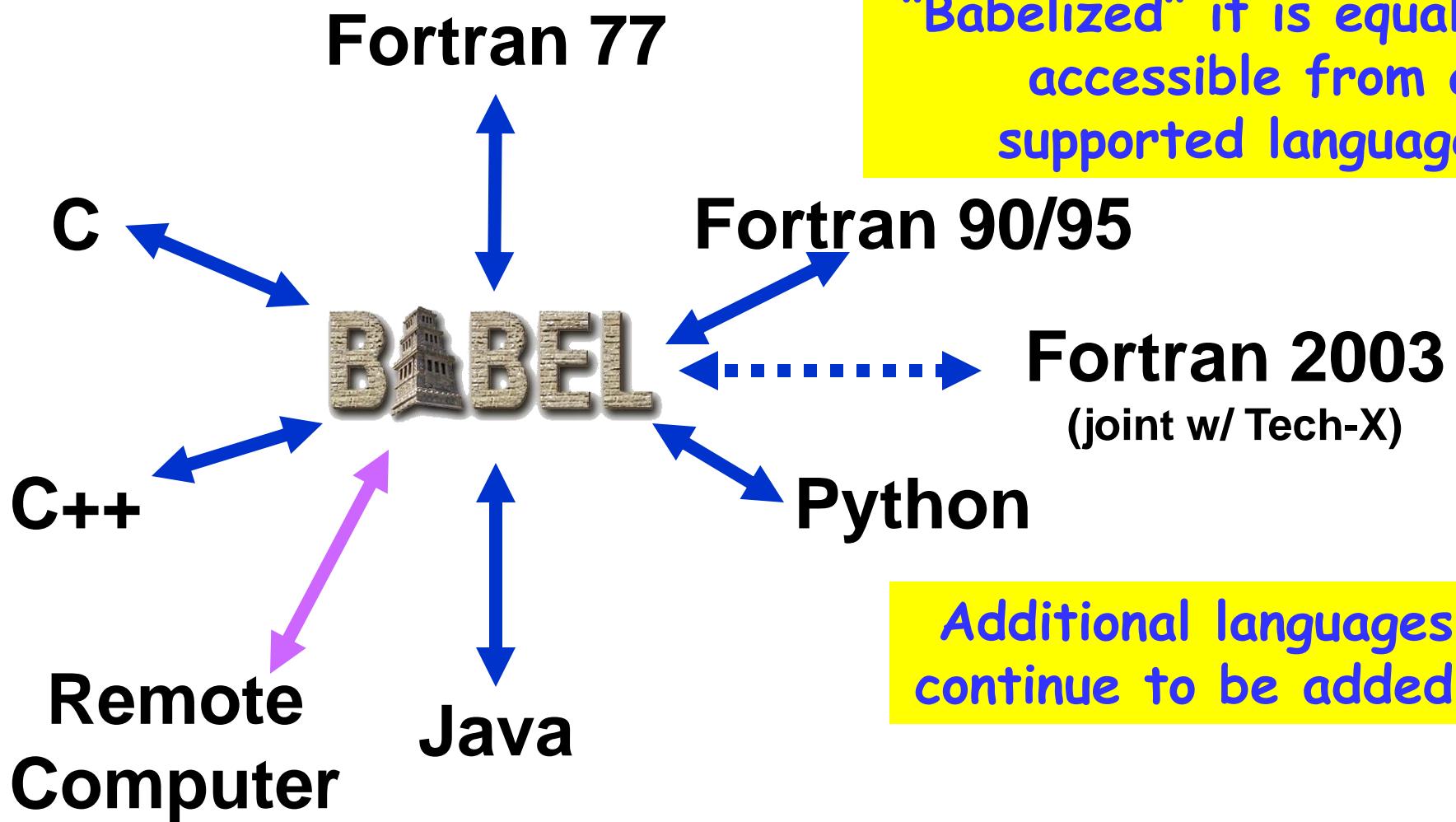
Before incorporating Babel, this project had used combinations of Pyfort and SWIG or f2py and SWIG, which reportedly gave them 90% of what they wanted. However, a new customer (U.S. Air Force) added the requirement of Java interfaces. Rather than discard their substantial investment in Python... they replaced all other point-to-point language tools with SIDL/Babel.

Kumfert et. al. *How the Common Component Architecture Advances Computational Science.* Proc SciDAC 2006 JoP **46**(2006) pp 479-493

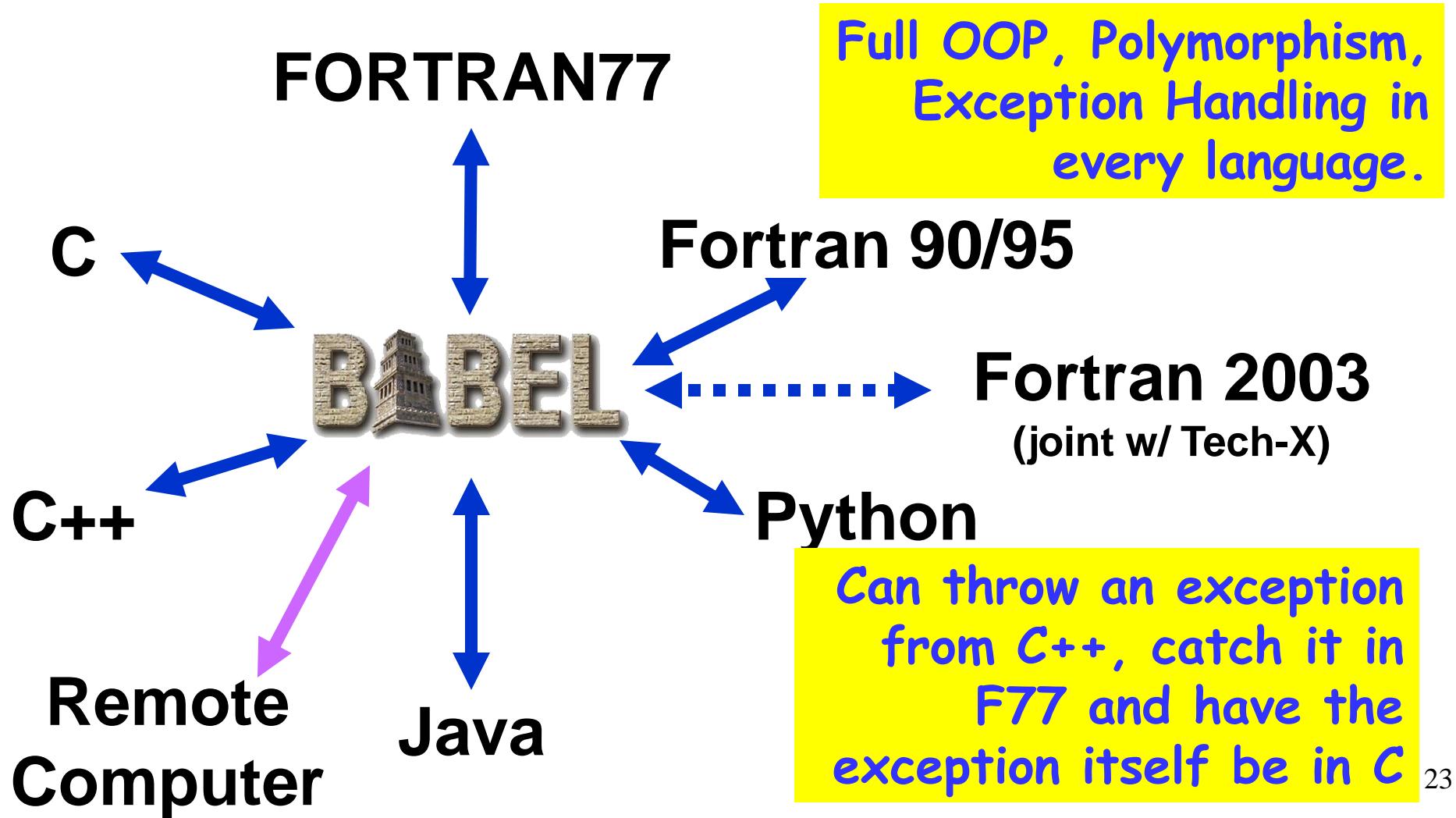
# When Mixing n Languages, Tool usage can grow $O(n^2)$



# Babel is an n-way Language Interoperability Tool



# Babel Supports a Uniform Model Across All Languages



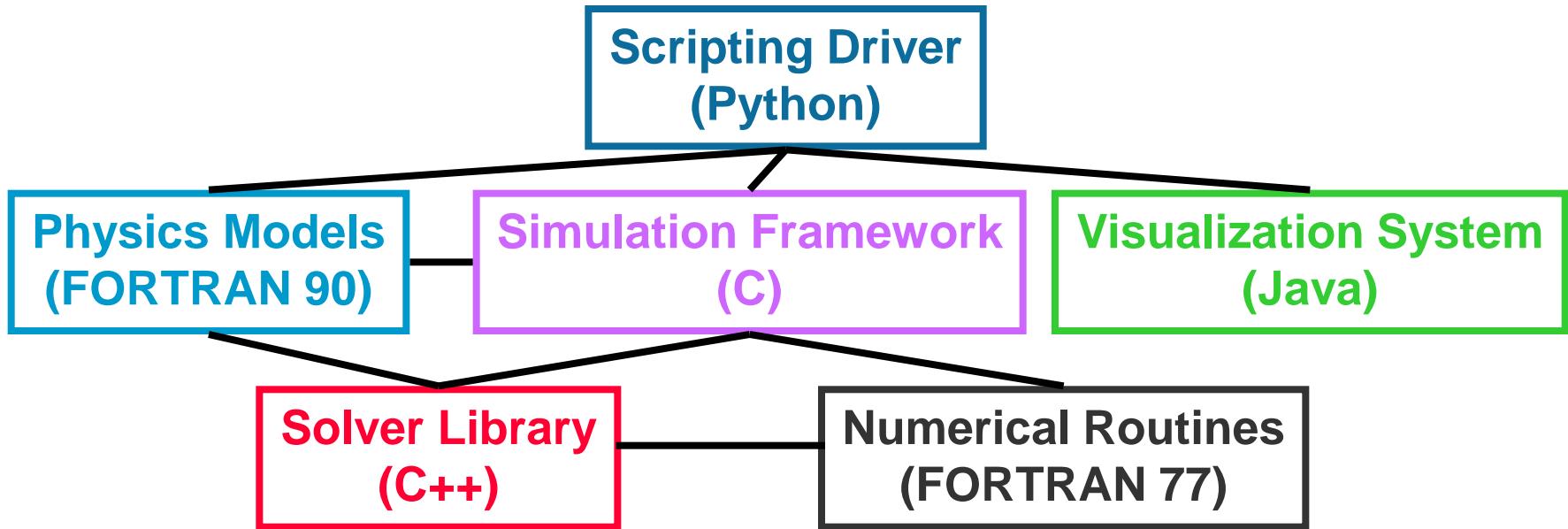
# Customers Use Babel To Serve a Variety of Needs

---

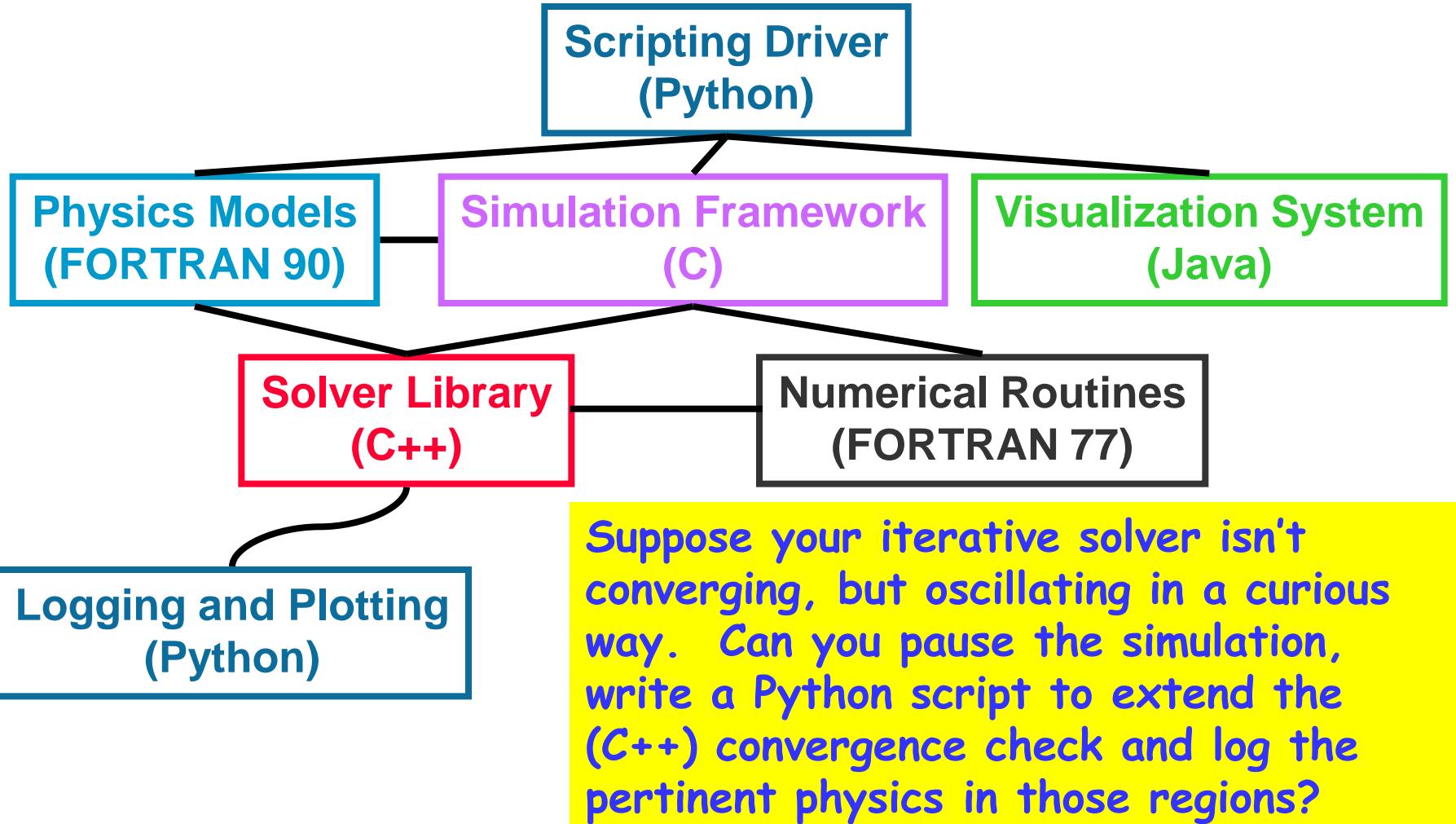
- Manage community codes  
[Chemistry, Fusion, Radio Astronomy]
- Create software interface specifications.  
[CCA, ITAPS, TOPS]
- Integrate multiple 3<sup>rd</sup> party libraries into a single scientific application.  
[Chemistry, Fusion, CMEE]
- **Develop libraries that connect to multiple languages.**  
[hypre, TAU, Sparsekit-CCA]
- Scientific Distributed Computing  
[Co-Op, SCIJump, Legion-CCA, Harness, GA]

# Most Science & Engineering Apps Already Mix Languages

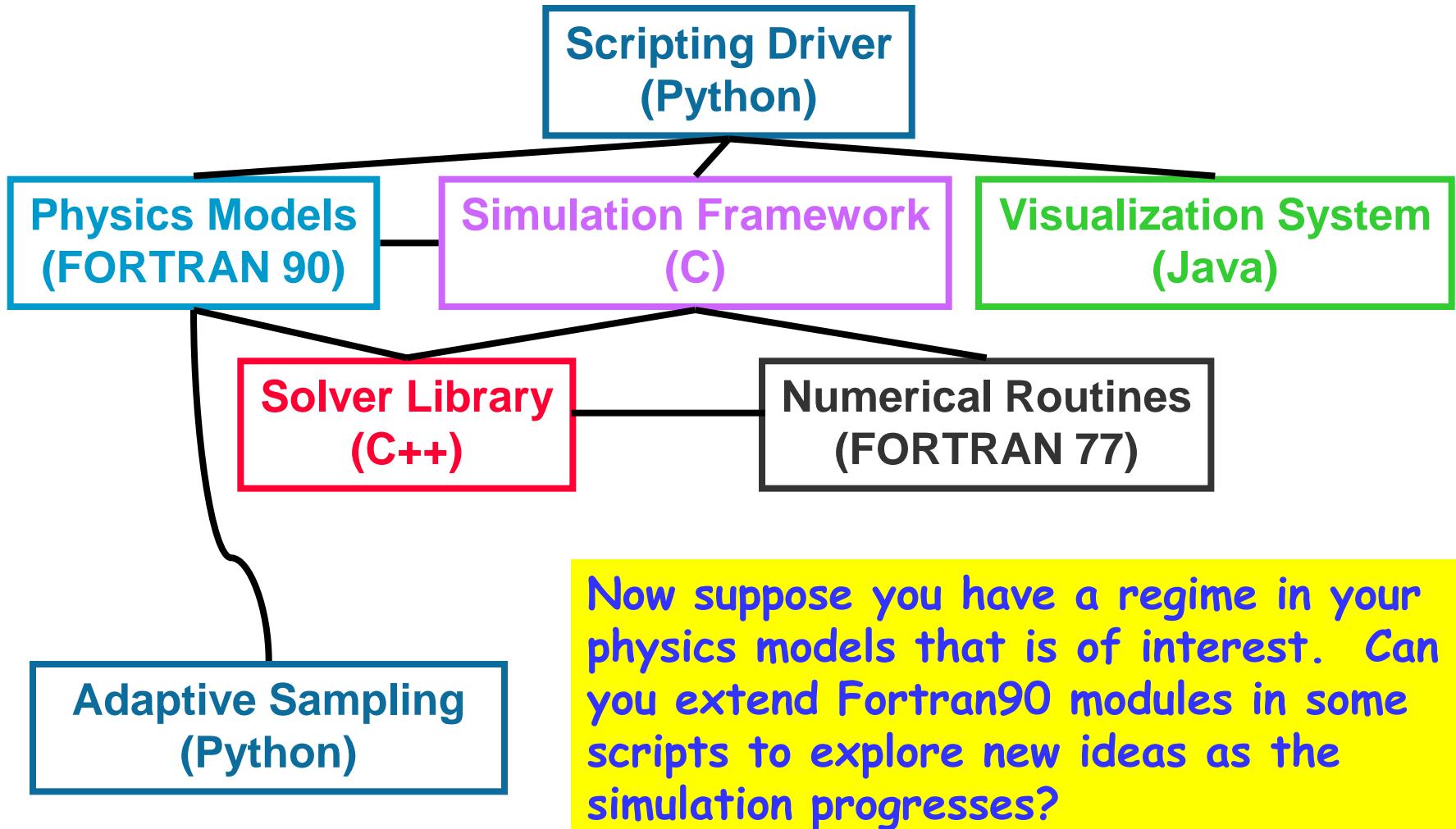
---



# When we say “Language Interoperability” we mean something very different than from what most applications do.



# When we say “Language Interoperability” we mean complete language transparency



# Customers Use Babel To Serve a Variety of Needs

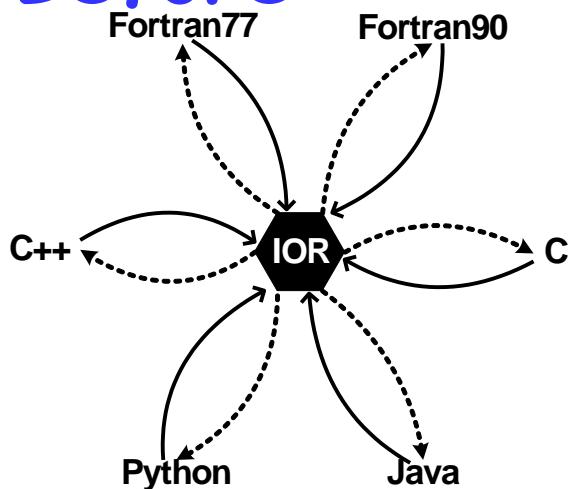
---

- Manage community codes  
[Chemistry, Fusion, Radio Astronomy]
- Create software interface specifications.  
[CCA, ITAPS, TOPS]
- Integrate multiple 3<sup>rd</sup> party libraries into a single scientific application.  
[Chemistry, Fusion, CMEE]
- Develop libraries that connect to multiple languages.  
[hypre, TAU, Sparsekit-CCA]
- **Scientific Distributed Computing**  
[Co-Op, SCIJump, Legion-CCA, Harness, GA]

# Now, Babel is More Than a Language Tool... After

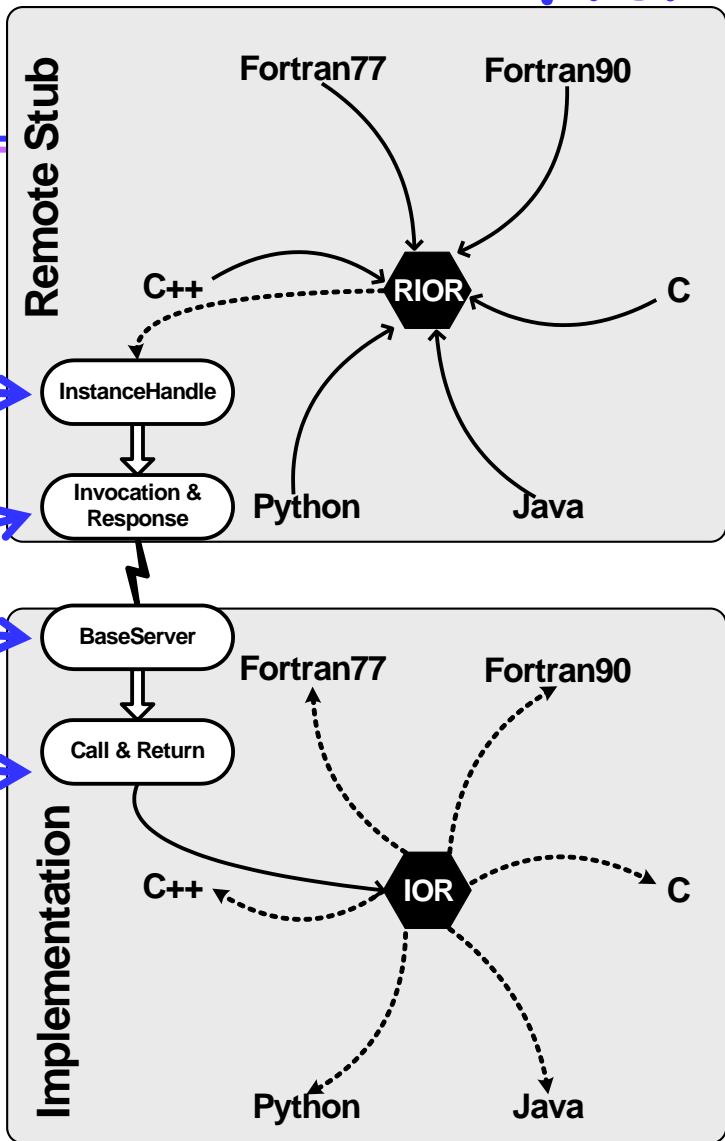
...it's distributed computing infrastructure

Before



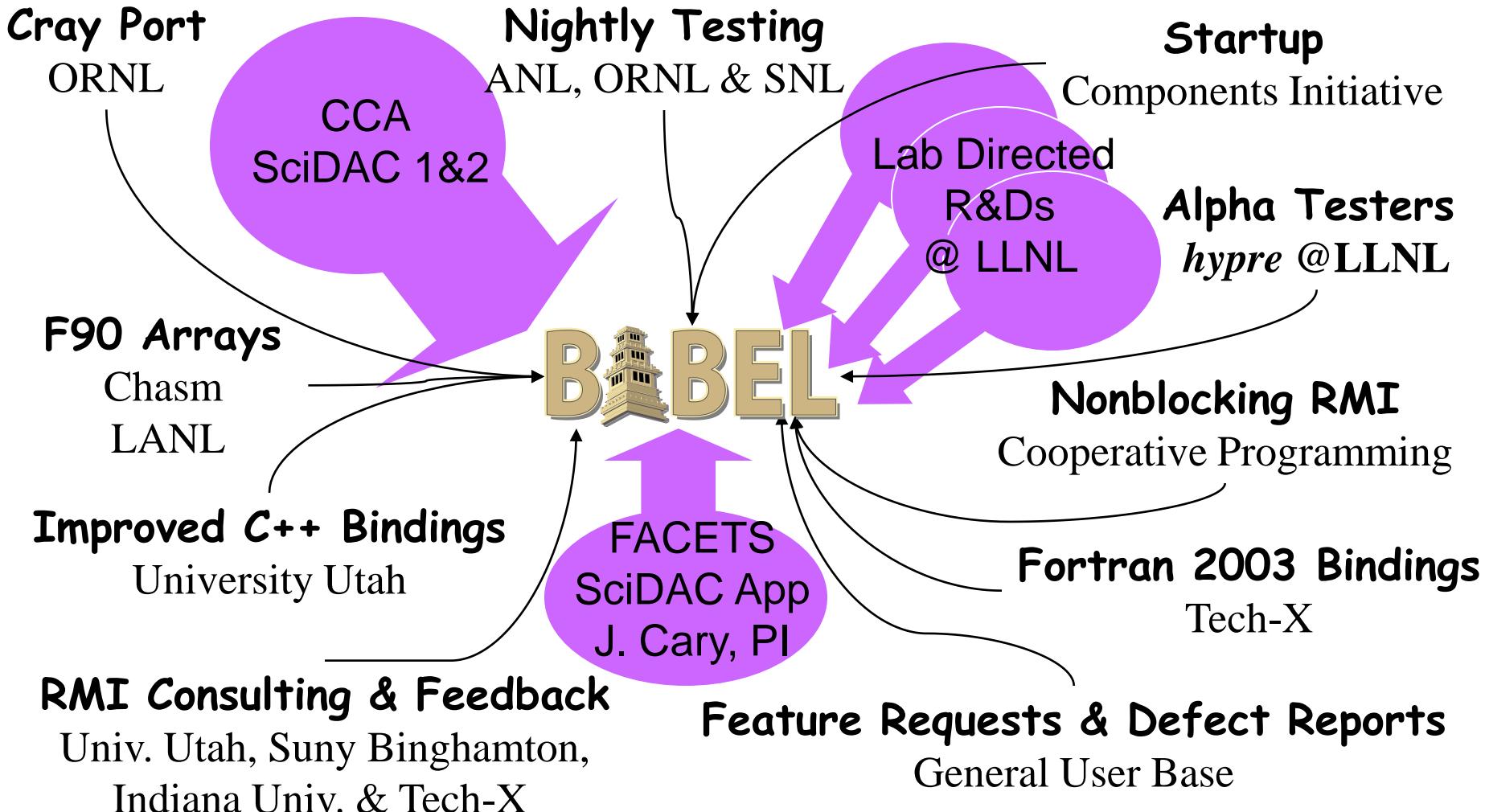
Key: Stub Skel

Network Layer defined in SIDL for 3rd party plug-ins



Key: Stub Skel Network Other

# Babel Thrives on Research Collaborations & Community



# Introduction Summary

---

- **Babel is a Software Integration Tool for big HPC codes**
- **Customers Use Babel to Serve Many Needs**
  - ▶ Manage community codes
  - ▶ Create software interface specifications
  - ▶ Integrate multiple 3<sup>rd</sup> party libraries
  - ▶ Develop libraries that connect to multiple languages
  - ▶ Scientific Distributed Computing
- **Babel itself is supported by research collaborations and community participation**

---

---

## II. SIDL

# Scientific Interface Definition Language

---

- The Smallest, non-degenerate SIDL file
- SIDL Type System
- SIDL Object Model
- Runtime Library of “built-in” objects
- Finer details

NOTE: In this section

- blue text is used to highlight words/concepts being defined
- blue typewriter font for SIDL keywords being defined

# A Simple, Complete SIDL File

---

```
package simple version 1.0 {  
    class HelloWorld {  
        string getMessage();  
    }  
}
```

- Method – only inside objects
- Object – only inside packages
- Package – can be arbitrarily nested

# Examples of Methods in SIDL

```
// ... sidl fragment
void run();

void xEqAxpY( in array<double,2> A,
               inout array<double> x,
               in array<double> y) ;

NetObject connect( in SocketID id )
                  throws NetworkException ;
```

- **Method must have**
  - ▶ return type
  - ▶ name  
(unique to the object)
  - ▶ argument list  
(may be empty)
- **Each Argument must have**
  - ▶ intent (in, out, inout)
  - ▶ type
  - ▶ name (unique to arg. list)

# Methods Only Appear Within Objects

```
// ... sidl fragment
interface Shape {
    void draw( in Brush b );
}

class Brush {
    void setwidth( in int pixels );
}
```

- Every piece of functioning code “wrapped” in Babel appears as either
  - ▶ a SIDL class, or
  - ▶ a SIDL class hierarchy  
(which may include SIDL interfaces)

# Objects only appear within a versioned package

```
package gov {  
    package cca version 0.6 {  
        // ... objects  
        package ports {  
            // ... more objects  
        }  
    }  
}
```

- Cannot have objects in package gov (not versioned)
- Objects in package gov.cca are version 0.6
- Objects in package gov.cca.ports also get version 0.6 (could be versioned separately)

# **SIDL is a Declarative Language Not a Programming Language**

---

```
package simple version 1.0 {  
    class HelloWorld {  
        string getMessage();  
    }  
}
```

- **Common Programming Constructs not in SIDL include:**
  - ▶ Assignment statement
  - ▶ String Literal
  - ▶ Print Statement

# Scientific Interface Definition Language

---

- The Smallest, non-degenerate SIDL file
- Basic SIDL types
- SIDL Object Model
- Runtime Library of “built-in” objects
- Finer details

# Key to Integrating Software is a Unified Type System

---

## ● Fundamental Types

- ▶ **bool**
- ▶ **char (8 bit)**
- ▶ **int (32 bit)**
- ▶ **long (64 bit)**
- ▶ **float (IEEE)**
- ▶ **double (IEEE)**
- ▶ **fcomplex**
- ▶ **dcomplex**
- ▶ **opaque (void \*)**
- ▶ **string**

## ● Aggregate Types

- ▶ **arrays**
- ▶ **enums**
- ▶ **structs** (in progress)

## ● Objects

- ▶ **classes**
- ▶ **interfaces**
- ▶ **abstract classes**

# Use SIDL to Define/Extend these Types

---

- Fundamental Types

- ▶ `bool`
- ▶ `char (8 bit)`
- ▶ `int (32 bit)`
- ▶ `long (64 bit)`
- ▶ `float (IEEE)`
- ▶ `double (IEEE)`
- ▶ `fcomplex`
- ▶ `dcomplex`
- ▶ `opaque (void *)`
- ▶ `string`

- Aggregate Types

- ▶ `arrays`
- ▶ `enums`
- ▶ `structs` (in progress)

- Objects

- ▶ `classes`
- ▶ `interfaces`
- ▶ `abstract classes`

# array - Fortran 95-style multidimensional array (& more)

```
array< int >  
  
array< T >  
  
array< double, 7 >  
  
array< fcomplex, 2, row-major>
```

- Have a stride, lower bound & upper bound in each dimension
- Can be reference counted, borrowed, and sliced
- Looks like C++ templates, but is specific to SIDL arrays
  - ▶ 1<sup>st</sup> argument: type of array (required)
  - ▶ 2<sup>nd</sup> argument: dimension (default=1, max=7)
  - ▶ 3<sup>rd</sup> argument: row-major, column-major, or arbitrary (default)
- No arrays of arrays

# rarray - C/F77-style “raw” array

```
void solve(in      rarray<double,2> A(m,n),  
          inout rarray<double>      x(n),  
          in      rarray<double>      b(m),  
          in      int                  m,  
          in      int                  n);
```

- Many restrictions from full-featured Babel arrays
  - ▶ always column major and packed
  - ▶ index range is always 0 to n-1
  - ▶ only numeric types... etc.
- Benefits: More intuitive in some cases  
(esp. C or F77 based math libraries like BLAS)

# rarray Lengths Are Flexible

```
Shape rotate3d( in Shape s  
                in rarray<double,2> M(3,3) );  
  
double CSRMatNorm( in int n, in int nnz,  
                    in rarray<int> Aptr(n+1),  
                    in rarray<int> Aind(nnz),  
                    in rarray<double> Aval(nnz) );
```

- Can be:
  - ▶ Constants
  - ▶ Variables
  - ▶ Any invertible function with one independent variable
- Variables must appear somewhere in argument list

# **enum** – Mutually exclusive symbolic values

```
enum checking{ aggressive, lazy, never }  
enum errorLevel {  
    warning, error, fatal, none=0  
}
```

- Unique numbers are assigned if not specified (Just like ANSI C enum)
- Not technically exciting (or innovative like rarrays & objects are)
- Important for software engineering

# **struct - Work In Progress**

```
struct Field {  
    array<double,3> data;  
    Units unit;  
    array<double, 3> coords;  
    bool dirty;  
}
```

- Will support structs of any other type
  - ▶ including structs of structs
- No data copy between C, C++, and Fortran 2003 (using BIND(C))
- Fortran 77/90/95, Java, & Python will access structs via get/set methods.

# **Scientific Interface Definition Language**

---

- The Smallest, non-degenerate SIDL file
  - Basic SIDL types
- SIDL Object Model**
- Runtime Library of “built-in” objects
  - Finer details

# Objects Contain User-Defined Methods (so we start here)

---

- Fundamental Types

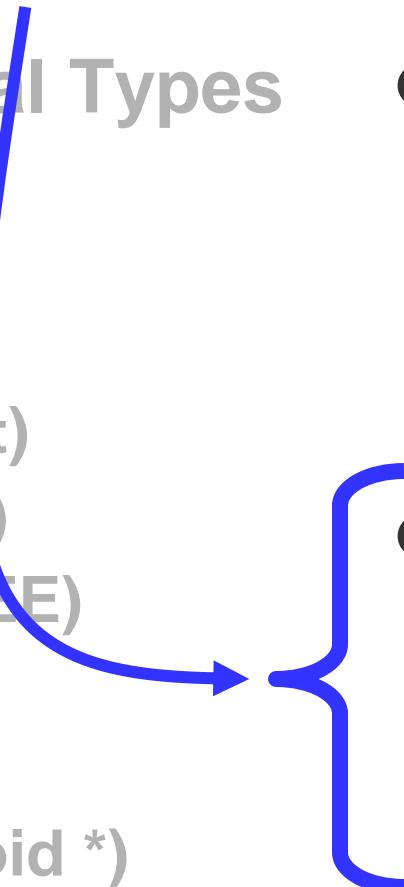
- ▶ `bool`
- ▶ `char (8 bit)`
- ▶ `int (32 bit)`
- ▶ `long (64 bit)`
- ▶ `float (IEEE)`
- ▶ `double (IEEE)`
- ▶ `fcomplex`
- ▶ `dcomplex`
- ▶ `opaque (void *)`
- ▶ `string`

- Aggregate Types

- ▶ `arrays`
- ▶ `enums`
- ▶ `structs` (in progress)

- Objects

- ▶ **classes**
- ▶ **interfaces**
- ▶ **abstract classes**

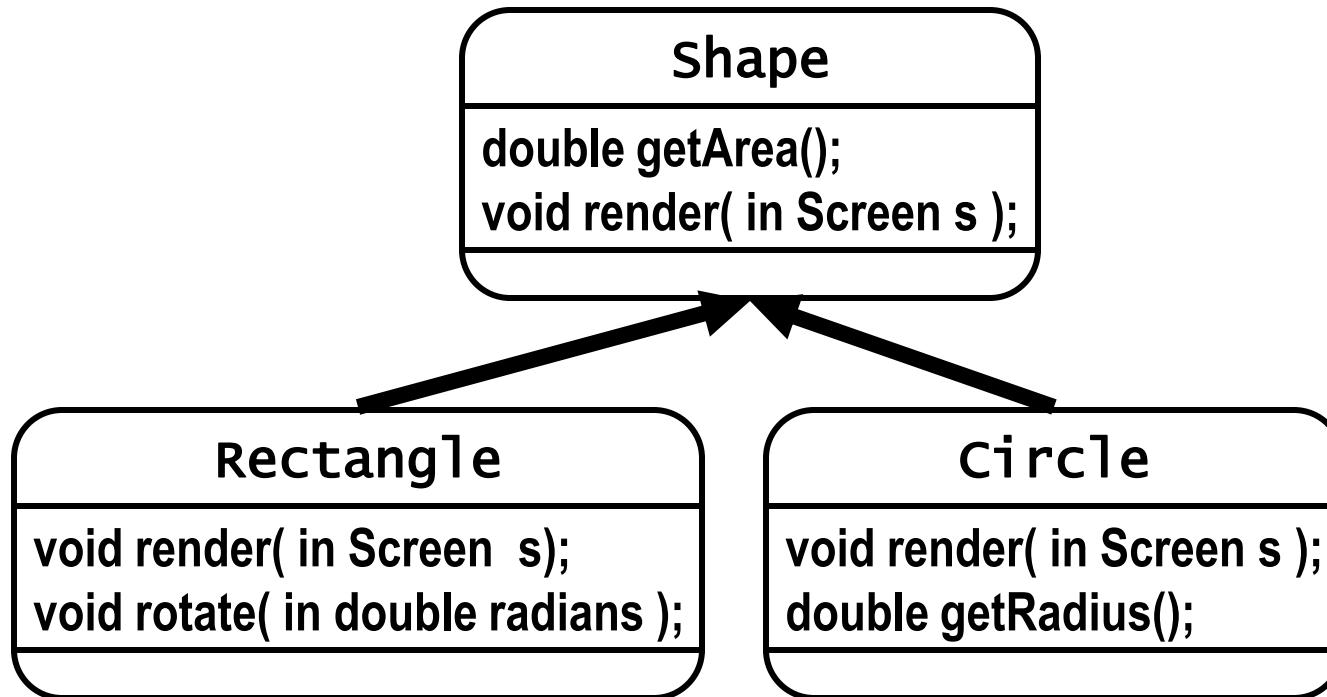


# **Standard Object-Oriented Design Principles Apply to SIDL**

---

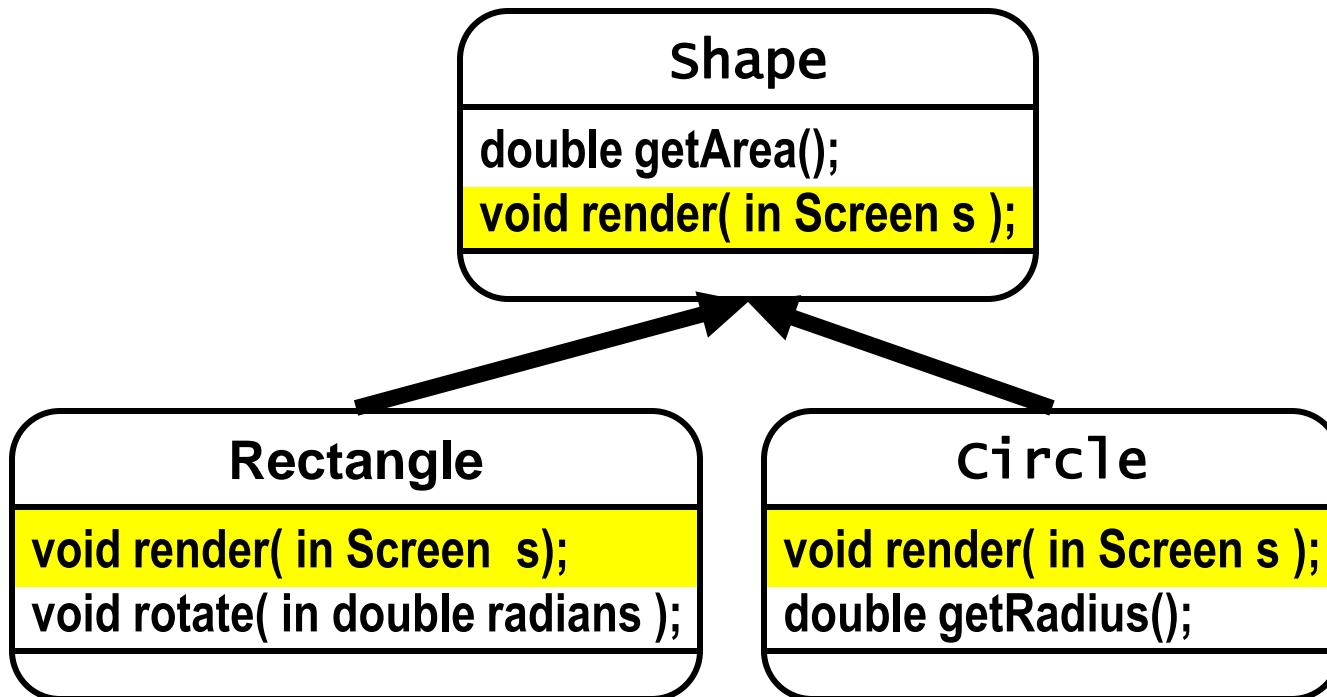
- Inheritance
- Polymorphism
- Method Overriding
- Method Overloading
- abstract vs. concrete methods
- virtual vs. final methods
- Exceptions

# Inheritance - Group related objects into hierarchies



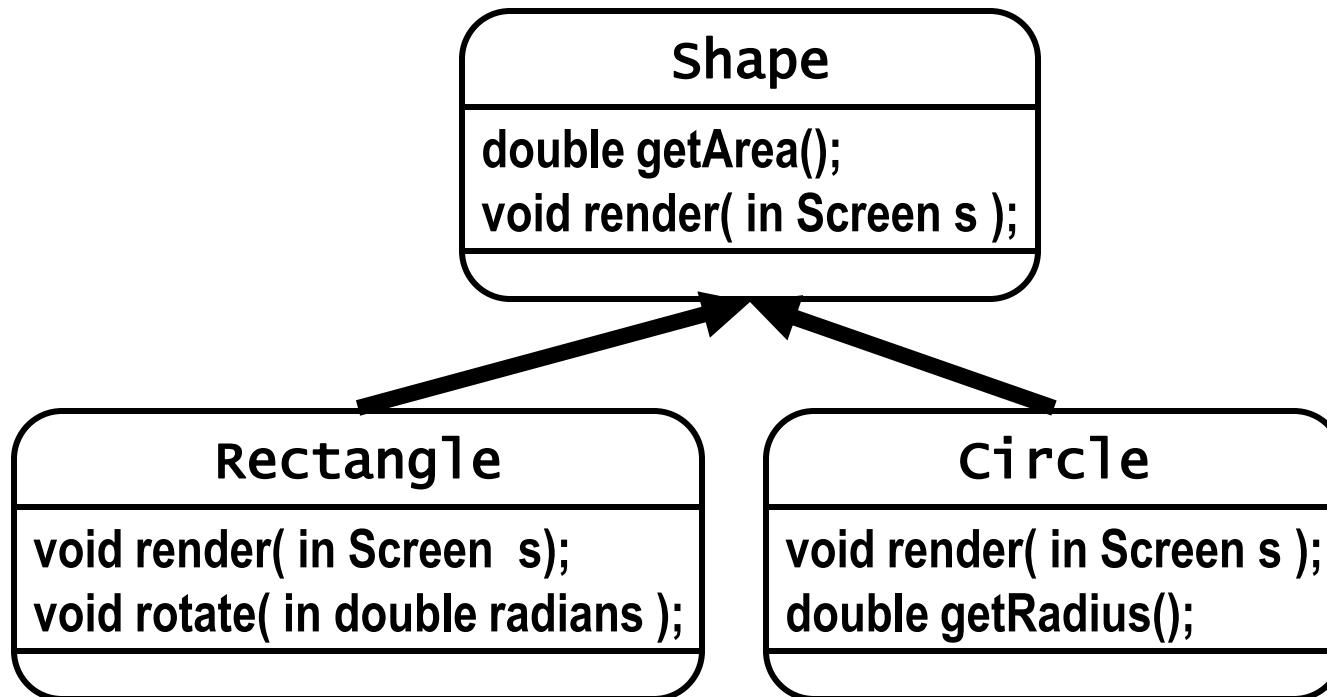
- Promotes code reuse
- Objects can inherit a capability from a common ancestor
- Objects can support the same method interface, but replace or augment with their own capabilities

# Method Overriding



- Promotes code reuse
- Objects can inherit a capability from a common ancestor
- Objects can support the same method interface, but replace or augment with their own capabilities

# Polymorphism - Lets Hierarchy Work Out Details For You



Deal with collections of objects through their common ancestor

- method overloading chooses proper implementation at runtime
- e.g. To draw a bunch of rectangles and circles to a screen,
  - ▶ keep them in a single list of shapes and call render().

# **virtual** vs. **final**

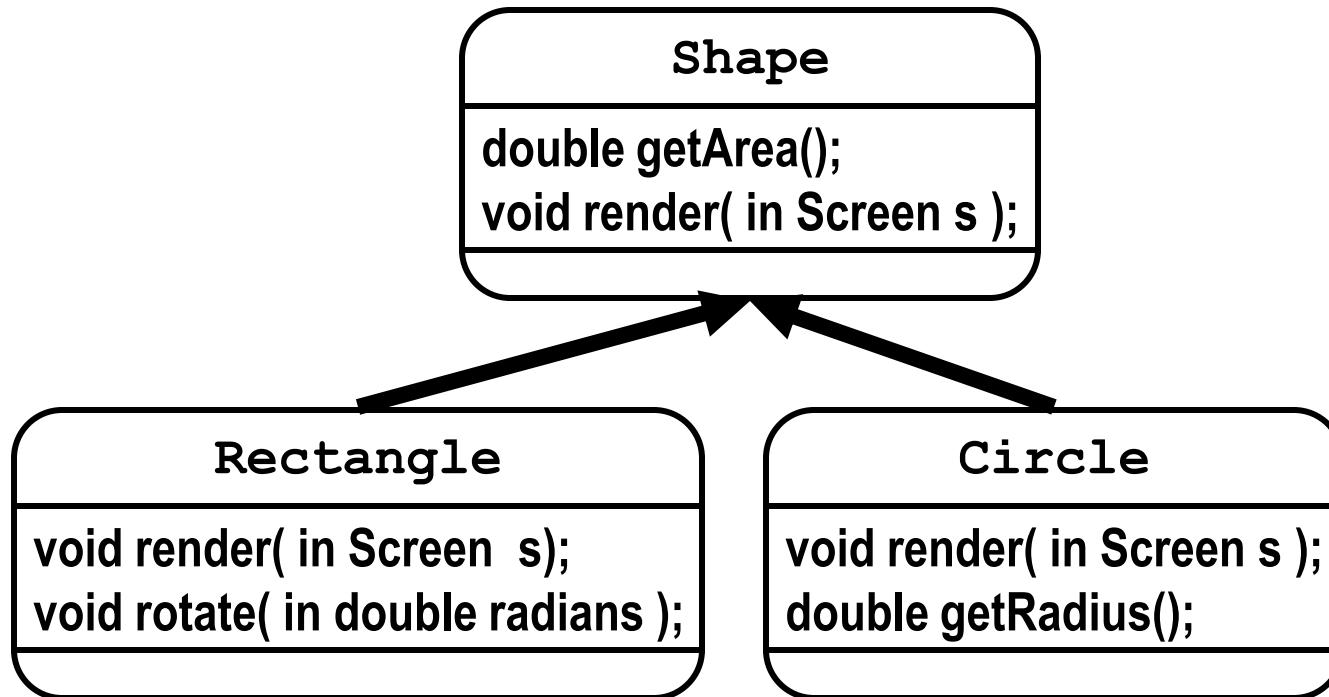
## **Can a method be overridden?**

---

- **SDL default is always “yes” (**virtual**)**
  - ▶ Use **final** keyword to prohibit overriding
  - ▶ Same as Java
  - ▶ Opposite of C++

# abstract vs. concrete

## Is the method implemented?



- **abstract methods**
  - ▶ specify an Method signature
  - ▶ require a derived class to implement
- **e.g. Diagram does not specify implementations, but we can guess**
  - ▶ Shape.getArea() – intuitively abstract, should be concrete in children
  - ▶ Shape.render() – clearly overridden, but ambiguous in figure

# Two Method Characteristics Fine Tune Polymorphism

overridable?	virtual	final
implemented?		
abstract	Mandatory for SIDL interfaces	Impossible
concrete	default for SIDL classes	

# SIDL Object Types

---

- **interface**
  - ▶ All methods are **virtual & abstract**
- **concrete class**
  - ▶ All methods are **concrete**
  - ▶ Is **only object that can be instantiated**
- **abstract class – (less common)**
  - ▶ A **class with at least one abstract method**
  - ▶ Can have **concrete methods too**

# SIDL Inheritance Model

---

- **interfaces**
  - ▶ **extend** multiple interfaces
- **classes**
  - ▶ **extend** at most one class, but
  - ▶ **implement** multiple interfaces
- Same as Java and Objective C

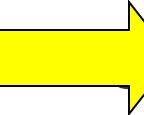
# **SIDL also has Implicit Base Types to Root All Objects**

---

- **sidl.BaseInterface**  
is the root of all SIDL interfaces
- **sidl.BaseClass**  
is the root of all SIDL classes
- **sidl.BaseClass** implements  
**sidl.BaseInterface**

# **Scientific Interface Definition Language**

---

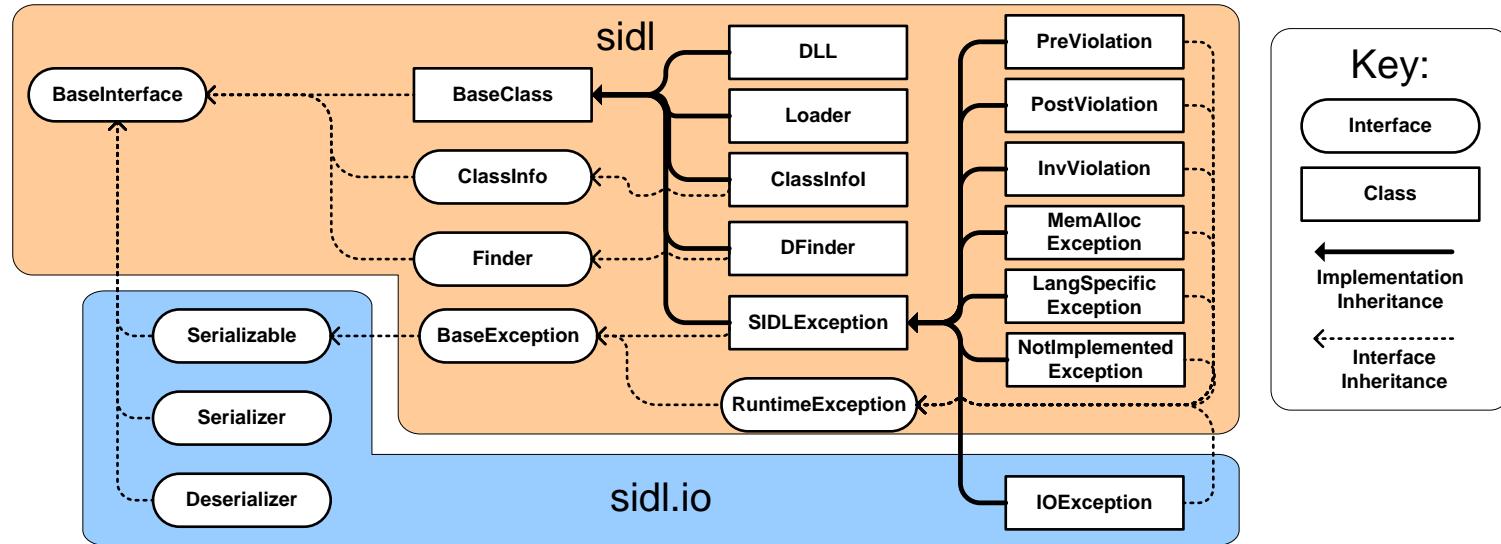
- The Smallest, non-degenerate SIDL file
  - Basic SIDL types
  - SIDL Object Model
-  Runtime Library of “built-in” objects
- Finer details

# SIDL's Type System Presumes Existence of Certain Types

---

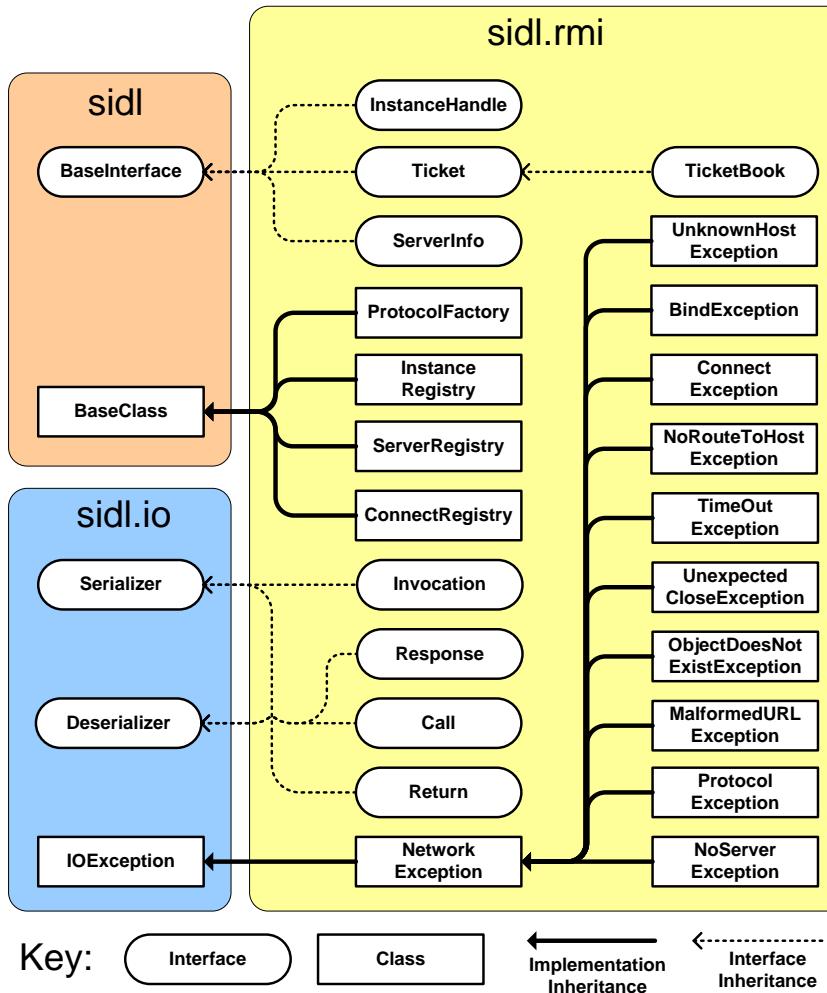
- **package sidl**
  - ▶ Base Classes, Interfaces, Runtime Class Loaders, Limited Introspection and Exception support
- **package sidl.io**
  - ▶ Object serialization (includes exceptions)
- **package sidl.rmi**
  - ▶ Remote Method Invocation

# Hierarchy of Objects in sidl and sidl.io packages



- Added constraints on “special” objects
  - ▶ Exceptions must implement `sidl.BaseException`
  - ▶ All methods implicitly throw `sidl.RuntimeException`
  - ▶ An object copied via RMI must implement `sidl.io.Serializable`
  - ▶ etc.

# Heirarchy of Objects in sidl.rmi package



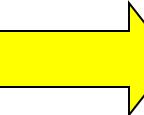
● Most of these are specific to implementing a wire-protocol in Babel RMI

- ▶ TCP/IP – built-in
- ▶ PSP – LLNL
- ▶ IIOP – Tech-X
- ▶ SOAP – SUNY Binghamton
- ▶ RMIX – GA Tech
- ▶ others...

# Scientific Interface Definition Language

---

- The Smallest, non-degenerate SIDL file
- Basic SIDL types
- SIDL Object Model
- Runtime Library of “built-in” objects

 Finer details

# SIDL Supports Comments and Doc-Comments

---

```
/*
 * 1. This is a multi-line comment
 */

// 2. A single line comment

/* 3. Comment is less than a line */

/** 4. A documentation comment */

/**
 * 5. Documentation comments can span
 *     multiple lines without the beginning
 *     space-asterisk-space combinations
 *     getting in the way.
 */
```

Doc-Comments  
will appear in  
generated code.

# SIDL has a unique approach to Method Overloading

```
// ...  
  
class Set {  
  
    void insert[Float] ( in float f );  
  
    void insert[Int] ( in int i );  
  
}
```

- Let the author define the hash for the method name.
  - ▶ The text in the brackets is a “uniquifying suffix”
- Programming languages with support for overloading (F90, Java and C++) use the “overloaded name”: `insert`
- Programming languages without overloading support (F77, C, and Python) use the “unique full-name”: `insertFloat` or `insertInt`

# SIDL supports Eiffel-like Design-by-Contract

```
// ... sidl fragment

double norm( in array<double> u, in double tol)

require /* preconditions */

    not_null : u != null ;
    non_negative_tol : tol >= 0.0 ;

ensure /* postconditions */

    non_negative_result : result >= 0.0 ;
    nearEqual(result, 0.0, tol) iff isZero(u, tol) ;
```

- Tamara Dahlgren's Ph.D. Dissertation
  - Performance-Driven Interface Contract Enforcement for Scientific Components [CBSE '07]

# SIDL Module Review

---

- SIDL is a declarative language, not a programming language
  - SIDL is used to define/extend arrays, enums, structs and especially... objects
  - SIDL is an Object-Oriented type system
  - SIDL has a runtime library of built-in types in sidl, sidl.io, and sidl.rmi packages
- 
- This module is far from exhaustive:
    - ▶ SIDL has 55 reserved words

---

---

# V. Babel Tool

# Outline

---

- **Introduction to the Babel Developers Kit**
- **How to download, build & install it**
- **How to run Babel**
- **What Babel produces**

# The Babel developers kit has three main parts

---

- The Babel tool (implemented in Java) to translate interface definition language into useable glue code
- The Babel runtime library that provides basic services (implemented in C)
- Babel examples and an extensive suite of multi-language tests

# Babel supports common HPC languages

---

- C
- C++
- FORTRAN 77
- Fortran 90/95
- Python 2.x
- Java
- Fortran 2003 (future)

# Getting/installing Babel

---

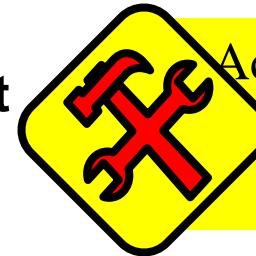
- <http://www.llnl.gov/CASC/components/software.html>
- In an ideal world...

```
% tar --file babel-1.1.2.tar.gz --ungzip --extract
```

```
% cd babel-1.1.2
```

```
% ./configure ; make
```

where make is GNU make



Actual release  
number is  
babel-1.2.0

- Babel configure script may disable some languages if it can't find required features.
  - ▶ F90 needs CHASM (CHASMPREFIX env. variable)
  - ▶ Python needs NumPy or Numeric Python
  - ▶ Java needs approved Java Developer Kit (JDK)
- Configure has lots of settings. See `configure --help`

# Checking a Babel build

---

- It's good idea to check your Babel  
% make check  
If everything goes right, you should see something like:

```
621. wrapper/runSIDL/runSIDL.sh[wrapper.User.XML->XML] ..... PASS
622. wrapper/runSIDL/runSIDL.sh[wrapper.XML->XML] ..... PASS
*****
Tue, 04 Sep 2007 at 23:29:09
by unknown@tux163.llnl.gov
Total      Passed      Xfailed      Failed      Broken
Tests        622          616             6            0
Parts       19074        19062           12            0
*****
Broken|Failed|Warning          Exit   Tot %Fail List of Failed
```

# How to run Babel

---

- In a shell, try typing  
% babel --version  
Babel version 1.1.2  
%  
● If that works, babel is already in your path; otherwise, ask your system administrator or person who installed Babel where it is



# Babel's command line interface

---



- Babel is invoked from a shell command line
- The general pattern is  
% babel <options> <SIDL files|type names>
- For example,  
% babel --client=c matrix.sidl  
This generates a C api for the types found in matrix.sidl

# Babel has three primary capabilities

---

**% babel --client=<lang>**

- ▶ Generate client-side glue code for <lang>

**% babel --server=<lang>**

- ▶ Generate server-side glue code and implementation file

**% babel --text=(sidl|xml)**

- ▶ Generate a textual description of type information

# Babel has three ancillary functions

---

**% babel --parse-check**

- ▶ Check the syntax of a SIDL file

**% babel --version**

- ▶ Show the version of Babel

**% babel --help**

- ▶ Show brief descriptions of command line options

**% babel --client=<lang>  
generates code for using types**

---

- <lang> can be c, c++, f77, f90, python or java
- This generates source code to allow you to use one or more types from C, C++, F77, F90, Python or Java.
- This code must be compiled before you can use the API.



**% babel --server=<lang>**  
**generates code for implementing**

---

- <lang> can be c, c++, f77, f90, python or java
- Generates code for you to implement one or more types in <lang>
- Insert your implementation code in something\_Impl.<lang specific extension>



# **Server=Client+Server**

---

- --server generates everything that  
--client generates plus the glue code to  
link the IOR to your implementation  
code

# Options controlling how Babel generates directories

---

- **--output-directory**

Change the root directory where Babel will generate its files

- **--generate-subdirs**

Build directory tree instead of putting everything in current directory

- **--hide-glue**

Put everything except implementation code in a glue subdirectory (CCA)

# Building/Using an XML repository

---

```
% mkdir repo
```

```
% babel --text=xml --output-directory=repo \  
yourtypes.sidl mytypes.sidl theirs.sidl
```

- Now you can refer to it

```
% babel --repository-path=repo \  
--client=python MyClassDef
```

- Babel fetches MyClassDef and types its references from XML repository

---

---

# IV. Using Babel Objects

# How to use Babel objects that are already implemented

---

- ● Intrinsic capabilities and methods
- Basic reference counting
- Conway's game of life example (C++, C, F90 & Python)
- Dynamic loading example (Python & F77)
- Borrowed array example (C & C++)
- rarray examples (C, F77, and F90)
- Overview of basic rules

# Babel's type system provides intrinsic capabilities

---

- Classes have constructors/destructors
- Concrete classes have a `_create` method
- Up and down casting object/interface references
- Null object reference
- Null reference tests
- No explicit destroy method (destruction managed by reference counting)

# Sources of methods

---

- Explicitly declared in SIDL file
- Inherited from parent class or interfaces
- Intrinsic builtins
  - ▶ `_cast` change the type of an object
  - ▶ `_create` make a new instance
  - ▶ RMI: `_createRemote`, `_getURL`, `_isRemote`, & `_exec`
  - ▶ Binding specific: `_getior()` (C++)

# How to use Babel objects that are already implemented

---

- Intrinsic capabilities and methods
- ● Basic reference counting
- Conway's game of life example (C++, C, F90 & Python)
- Dynamic loading example (Python & F77)
- Borrowed array example (C & C++)
- rarray examples (C, F77, and F90)
- Overview of basic rules

# Creating a Babel object

- Creation is tailored to each language

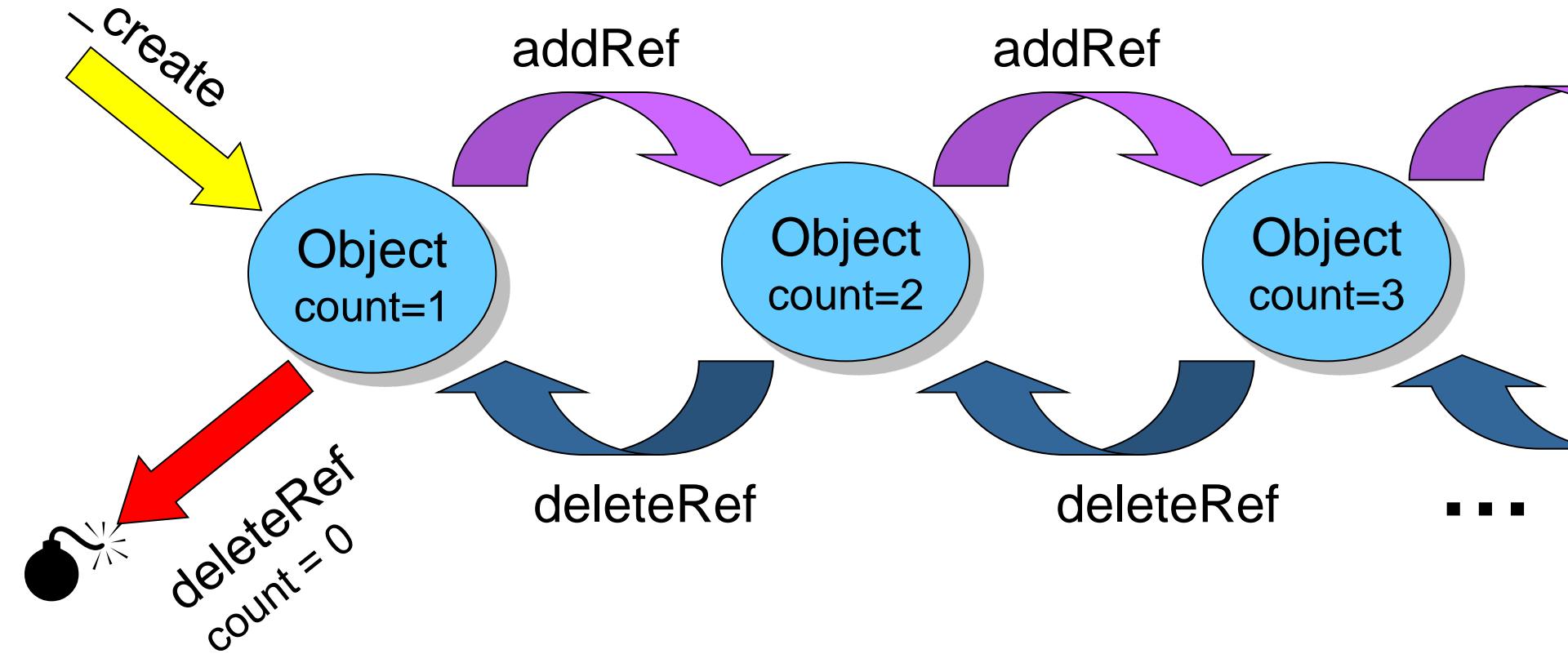
```
package a version 1.0 {  
    class B {}  
}
```

a.sidl

Language	Example
C	<code>a_B a =     a_B__create(&amp;ex);</code>
C++	<code>::a::B a =     ::a::B::__create();</code>
F77	<code>Integer*8 a call a_B__create_f(a,ex)</code>
F90	<code>using a_B type(a_B_t) :: a call new(a, ex)</code>
Java	<code>a.B a = new a.B();</code>
Python	<code>import a.B a = a.B.B()</code>

# Babel Object Lifecycle

---



# **Reference counting responsibilities vary by language**

---

- For C, FORTRAN 77, and Fortran 90
  - ▶ Reference counting is **your responsibility**
  - ▶ Requires explicit addRef/deleteRef calls
  - ▶ Everyone makes mistakes when starting out
  
- For C++, Java & Python
  - ▶ Reference counting is transparent
  - ▶ Avoid explicit calls to addRef/deleteRef



# Owning a reference

---

- Your reference is part of the current count
- Responsibilities that come with ownership
  - ▶ delete the reference when you're done with it or
  - ▶ transfer the reference to another piece of code that will take ownership



# Parameter passing modes and reference counting

---

- **in** parameters are borrowed by the implementation
- Returned references (i.e., return value and **out** parameters) are owned by the caller
- For an **inout** parameter, implementation may delete your reference and give you a different one or Null.

# How to use Babel objects that are already implemented

---

- Intrinsic capabilities and methods
- Basic reference counting
- ● Conway's game of life example (C++, C, F90 & Python)
- Dynamic loading example (Python & F77)
- Borrowed array example (C & C++)
- rarray examples (C, F77, and F90)
- Overview of basic rules

# The conway.BoundsException & conway.Environment interfaces

---

```
package conway version 2.0 {  
interface BoundsException extends sidl.BaseException {}  
  
interface Environment {  
    /** Initialize a grid to a certain height & width */  
    void init( in int height, in int width );  
  
    /** Return true iff a cell is alive */  
    bool isAlive( in int x, in int y ) throws  
    BoundsException ;  
  
    /** Return the number of living adjacent cells */  
    int nNeighbors( in int x, in int y ) throws  
    BoundsException ;  
  
    /** Return the current height & width */  
    void getBounds(out int height, out int width);  
  
    /** Set an entire grid of data */  
    void setGrid( in array<int,2,column-major> grid );  
}
```

# Example using conway.Environment from C++

---

```
// include C++ stub header
#include "conway_Environment.hh"
using sidl;
using ::conway::Environment;
```

- SIDL packages translate into C++ namespaces. Use "using" to avoid using fully qualified names.

# Example: calculating a time step in C++

---

```
int32_t height, width, x, y;
try {
    env.getBounds(height, width); ①
    array<int32_t> grid = array<int32_t>::create2dCol(height,
        width); ②

    for(x = 0, x < width; ++x) {
        for(y = 0; y < height; ++y) {
            int32_t n = env.nNeighbors(x, y); ①
            if ((n == 2 && env.isAlive(x, y) ) || n == 3) ①
                grid.set(y, x, 1); ②
            else
                grid.set(y, x, 0); ②
        }
    }
    env.setGrid(grid); ①
}
catch (BoundsException &be) { /* do something */ }
catch (RuntimeException &re) { /* Babel internal exception */ } ③
```

# Example: calculating a time step in C - part 1

---

```
#include "conway_Environment.h"
#include "sidl_Exception.h"
/* lines skipped */
int32_t height, width, x, y, n;
sidl_bool isAlive;
struct sidl_int_array *grid = NULL; ①
sidl_BaseInterface ex = NULL; ②
conway_Environment_getBounds(env, &height, &width, &ex); ③
SIDL_CHECK(ex); /* Check for a runtime exception */ ④
grid = sidl_int_array_create2dCol(height, width); ⑤
```

# Example: calculating a time step in C - part 2

---

```
for(x = 0, x < width; ++x) {
    for(y = 0; y < height; ++y) {
        n = conway_Environment_nNeighbors(env, x, y, &ex); ①
        SIDL_CHECK(ex); /* check for exception */ ②
        switch(n) {
            case 2:
                isAlive = conway_Environment_isAlive(env, x, y,
                    &ex); ③
                SIDL_CHECK(ex); /* check for exception */
                sidl_int_array_set2(grid, y, x, isAlive ? 1 : 0); ④
                break;
            case 3:
                sidl_int_array_set2(grid, y, x, 1); break;
            default:
                sidl_int_array_set2(grid, y, x, 0); break;
        }
    }
}
```

# Example: calculating a time step in C - part 3

---

```
conway_Environment_setGrid(env, grid, &ex); ①
SIDL_CHECK(ex); /* check for a runtime exception */
EXIT:; ②
/* cleanup extra array reference */
if (grid) sidl_int_array_deleteRef(grid); ③
/* exception handling here */
```

# Example: calculating a time step in F90 - part 1

---

```
#include "sidl_BaseInterface_fAbbrev.h" ①
#include "conway_Environment_fAbbrev.h"
#include "conway_BoundsException_fAbbrev.h"
! skipping to later in the file
use sidl_BaseInterface ②
use conway_Environment
use conway_BoundsException
implicit none
type(sidl_int_2d) :: grid ③
type(sidl_BaseInterface_t)::ex, ex2 ④
logical :: alive
integer(selected_int_kind(9)) :: x, y, height, width, n ⑤
call set_null(ex) ⑥
call getBounds(env, height, width, ex) ⑦
if (not_null(ex)) goto 100 ⑧
call create2dCol(height, width, grid) ⑨
```

# Example: calculating a time step in F90 - part 2

---

```
do x = 0, width - 1
  do y = 0, height - 1
    grid%d_data(y,x) = 0      ! assume that it's dead ①
    call nNeighbors(env, x, y, n, ex) ②
    if (not_null(ex)) go to 100 ③
    if (n .eq. 2) then
      call isAlive(env, x, y, alive, ex) ④
      if (not_null(ex)) go to 100
      if (alive) then
        grid%d_data(y,x) = 1 ! alive ⑤
      endif
    else
      if (n .eq. 3) then
        grid%d_data(y,x) = 1 ! alive
      endif
    endif
  enddo
enddo
```

# Example: calculating a time step in F90 - part 3

---

```
call deleteRef(grid, ex) ! return unneeded reference ①
if (not_null(ex)) go to 100
return
100 call deleteRef(grid, ex2) ②
print *, 'BoundException or RuntimeException'
```

# Example: calculating a time step in Python

---

```
import Numeric ①
import conway.Environment ②
import conway.BoundsException
try:
    (height, width) = env.getBounds() ③
    grid = Numeric.zeros((height, width), Numeric.Int32)

    for x in xrange(width):
        for y in xrange(height):
            n = env.nNeighbors(x, y) ④
            if (n == 2 and env.isAlive(x, y) ) or n == 3:
                grid[y][x] = 1
    env.setGrid(grid)
except conway.BoundsException, be: ⑤
    pass # exception handling code
except sidl.RuntimeException, re: ⑥
    pass # runtime exception handling
```

# How to use Babel objects that are already implemented

---

- Intrinsic capabilities and methods
- Basic reference counting
- Conway's game of life example (C++, C, F90 & Python)
- ● Dynamic loading example (Python & F77)
- Borrowed array example (C & C++)
- rarray examples (C, F77, and F90)
- Overview of basic rules

# Dynamic class loading example: SIDL

---

```
// selected excerpts from sidl.sidl
package sidl version 0.9.0 {

    enum Scope { LOCAL, GLOBAL, SCLSCOPE } ;

    enum Resolve { LAZY, NOW, SCLRESOLVE } ;

    class DLL {
        BaseClass createClass(in string sidl_name);
    }

    class Loader {
        static DLL findLibrary(in string sidl_name,
                               in string target,
                               in Scope lScope,
                               in Resolve lResolve);
    }
}
```

# SIDL Class Loader (.scl) Files

## “sidl.Class” $\longleftrightarrow$ libsomething.so

```
<scl>
  <library uri="/usr/local/lib/libsidl.la" scope="global"
            resolution="now" >
    <class name="sidl.BaseClass" desc="ior/impl"/>
    <class name="sidl.DLL" desc="ior/impl"/>
  </library>
</scl>
```

- Important Environment Variables

- ▶ **\$SIDL\_DLL\_PATH** – Directories to search for .scl files
- ▶ **\$SIDL\_DEBUG\_DLOPEN** – (optional) if defined, sidl.Loader displays debugging information to stdout

# Dynamic class loading example in Python

---

```
from sidl.Scope import * ①
from sidl.Resolve import *
from sidl.Loader import findLibrary
import mouse.Trap # interface ②
dll = findLibrary("better.Trap", "ior/impl",
    SCLSCOPE, SCLRESOLVE) ③
if (dll):
    obj = dll.createClass("better.Trap") ④
    if (obj):
        trap = mouse.Trap.Trap(obj) # cast ⑤
        if (trap): # now we have a trap
            trap.catchMouse()
```

# Dynamic loading example in Fortran 77

---

```
integer*8 dll, obj, trap, ex ①
include 'sidl_Resolve.inc' ②
include 'sidl_Scope.inc'
call sidl_Loader_findLibrary_f('better.Trap', 'impl/ior', SCLSCOPE,
    SCLRESOLVE, dll, ex) ③
if (ex .ne. 0) go to 100
if (dll .ne 0) then ④
    call sidl_DLL_createClass_f('better.Trap', obj, ex)
    if (ex .ne. 0) go to 100
    if (obj .ne. 0) then
        call mouse_Trap_cast_f(obj, trap, ex) ④
        if (ex .ne. 0) go to 100
        if (trap .ne. 0) then
            call mouse_Trap_catchMouse_f(trap, ex)
            if (ex .ne. 0) go to 100
        endif
        call sidl_BaseClass_deleteRef_f(obj, ex) ⑤
    endif
    call sidl_DLL_deleteRef_f(dll, ex)
endif
C exception handling block not shown
```

# How to use Babel objects that are already implemented

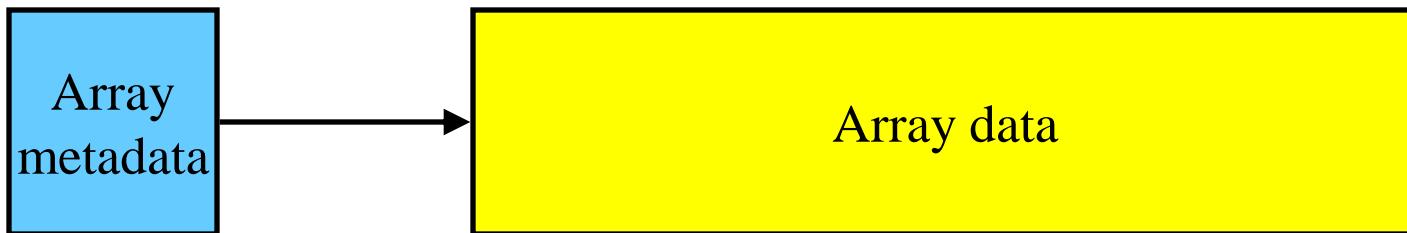
---

- Intrinsic capabilities and methods
  - Basic reference counting
  - Conway's game of life example (C++, C, F90 & Python)
  - Dynamic loading example (Python & F77)
- ● Borrowed array example (C & C++)
- rarray examples (C, F77, and F90)
  - Overview of basic rules

# Normal & borrowed arrays

---

In a normal SIDL array, both parts are allocated on the heap, and Babel frees both parts when the reference count goes to zero.



In a borrowed array, the data is allocated by your program, and Babel will never free it.



# Creating a borrowed array in C

---

```
double A[100][100], x[100], b[100]; ①
const int32_t low[2] = { 0, 0 }; ②
const int32_t up[2] = { 99, 99 };
const int32_t stride[2] = { 100, 1 }, vstride[1] = { 1 };
struct sidl_double_array ③
    *sA = sidl_double_array_borrow(A, 2, low, up, stride),
    *sx = sidl_double_array_borrow(x, 1, low, up, vstride),
    *sb = sidl_double_array_borrow(b, 1, low, up, vstride),
    *extrax = sx;
sidl_double_array_addRef(extrax); ④
loadProblem(A, b); /* initialize A & b */
matrix.Solver.solve(/*in*/ sA, /*inout*/ &sx, /*in*/ sb, &ex); ⑤
if (sx != extrax) sidl_double_array_copy(sx, extrax); ⑥
sidl_double_array_deleteRef(sx);
sidl_double_array_deleteRef(extrax);
sidl_double_array_deleteRef(sA);
sidl_double_array_deleteRef(sb);
```

# Creating a borrowed array in C++

---

```
// assuming using sidl
double A[100][100], x[100], b[100]; ①
const int32_t low[2] = { 0, 0 }; ②
const int32_t up[2] = { 99, 99 };
const int32_t stride[2] = { 100, 1 },
    vstride[1] = { 1 };
array<double> sA, sx, sb, extrax; ③
loadProblem(A, b); // initialize A & b
sA.borrow(A, 2, low, up, stride); ④
sx.borrow(x, 1, low, up, vstride); ④
sb.borrow(b, 1, low, up, vstride); ④
extrax = sx; ⑤
matrix.Solver.solve(/*in*/ sA, /*inout*/ sx, /*in*/
    sb);
if (sx != extrax) extrax.copy(sx); ⑥
```

# Creating a persistent reference to an array

---

- Use smartCopy when creating a persistent reference to an unknown array to avoid a reference to a borrowed array because the array data may unexpectedly disappear

```
struct sidl_double_array *g_array;  
void cache(struct sidl_double_array *src)  
{  
    if (g_array)  
        sidl_double_array_deleteRef(g_array);  
    g_array =  
        sidl_double_array_smartCopy(src);  
}
```

# How to use Babel objects that are already implemented

---

- Intrinsic capabilities and methods
  - Basic reference counting
  - Conway's game of life example (C++, C, F90 & Python)
  - Dynamic loading example (Python & F77)
  - Borrowed array example (C & C++)
- ● rarray examples (C, F77, and F90)
- Overview of basic rules

# rarray's provide more natural method signatures

---

- rarray's were design to provide more natural looking array bindings for C, FORTRAN 77, and Fortran 90/95
- Only for `in` and `inout` passing modes
- No resizing or replacing the array
- Multi-dimensional arrays must be in column-major (FORTRAN 77) order

# Example rarray declarations

---

```
static

void solve(in rarray<double, 2> A(m,n) ,
           inout rarray<double> x(n) ,
           in rarray<double>      b(n) ,
           in int                  m,
           in int                  n) ;

void ex(in      rarray<double> x(n+1) ,
        inout rarray<double> y(n-1) ,
        in      int            n) ;
```

rarray extents  
can be simple  
expressions

# C API for rarrays - part 1

---

## ● Client- and server-side bindings

```
void  
a_B_solve(  
    /* in rarray[m,n] */ double* A,  
    /* inout rarray[n] */ double* x,  
    /* in rarray[m] */ double* b,  
    /* in */ int32_t m, /* in */ int32_t n,  
    /* out */ sidl_BaseInterface *_ex);  
void  
impl_a_B_solve(  
    /* in rarray[m,n] */ double* A,  
    /* inout rarray[n] */ double* x,  
    /* in rarray[m] */ double* b,  
    /* in */ int32_t m, /* in */ int32_t n,  
    /* out */ sidl_BaseInterface *_ex)
```

# C API for rarrays - part 2

---

## ● Client- and server-side bindings

```
void  
impl_a_B_ex(  
    /* in */ a_B self,  
    /* in rarray[n+1] */ double* x,  
    /* inout rarray[n-1] */ double* y,  
    /* in */ int32_t n,  
    /* out */ sidl_BaseInterface *_ex)  
  
void  
impl_a_B_ex(  
    /* in */ a_B self,  
    /* in rarray[n+1] */ double* x,  
    /* inout rarray[n-1] */ double* y,  
    /* in */ int32_t n,  
    /* out */ sidl_BaseInterface *_ex)
```

# **FORTRAN 77 API for rarray's**

## **part 1**

---

### **● Client-side binding**

```
subroutine a_B_solve_f(A, x, b, m, n, ex)
integer*4 m, n
double precision A(0:m-1, 0:n-1), x(0:n-1), b(0:m-1)
integer*8 ex
```

### **● Server-side binding**

```
subroutine a_B_solve_fi(A, x, b, m, n, ex)
integer*4 m, n
double precision A(0:m-1, 0:n-1), x(0:n-1), b(0:m-1)
integer*8 ex
```

# **FORTRAN 77 API for rarray's**

## **part 2**

---

### **● Client-side binding**

```
subroutine a_B_ex_f(self, x, y, n, ex)
integer*8 self, ex
integer*4 n
double precision x(0:n+1-1), y(0:n-1-1)
```

### **● Sever-side binding**

```
subroutine a_B_ex_hi(self, x, y, n, ex)
integer*8 self, ex
integer*4 n
double precision x(0:n+1-1), y(0:n-1-1)
```

# Fortran 90 rarray client-side bindings – Part 1

---

- F90 overloading allows you to pass either all SIDL arrays or all native F90 arrays

```
recursive subroutine solve_1s(A, x, b, exception)
    implicit none
    type(sidl_double_2d) , intent(in) :: A
    type(sidl_double_1d) , intent(inout) :: x
    type(sidl_double_1d) , intent(in) :: b
    type(sidl_BaseInterface_t) , intent(out) :: exception

recursive subroutine solve_2s(A, x, b, exception)
    implicit none
    real (kind=sidl_double) , intent(in), dimension(:, :) :: A
    real (kind=sidl_double) , intent(inout), dimension(:) :: x
    real (kind=sidl_double) , intent(in), dimension(:) :: b
    type(sidl_BaseInterface_t) , intent(out) :: exception
```

# Fortran 90 rarray client-side bindings – Part 2

---

- F90 overloading allows you to pass either all SIDL arrays or all native F90 arrays

```
recursive subroutine ex_1s(self, x, y, exception)
  implicit none
  type(a_B_t) , intent(in) :: self
  type(sidl_double_1d) , intent(in) :: x
  type(sidl_double_1d) , intent(inout) :: y
  type(sidl_BaseInterface_t) , intent(out) :: exception

recursive subroutine ex_2s(self, x, y, exception)
  implicit none
  type(a_B_t) , intent(in) :: self
  real (kind=sidl_double) , intent(in), dimension(:) :: x
  real (kind=sidl_double) , intent(inout), dimension(:) :: y
  type(sidl_BaseInterface_t) , intent(out) :: exception
```

# Fortran 90 rarray server-side bindings

---

```
recursive subroutine a_B_solve_mi(A, x, b, m, n,
    exception)
integer (kind=sidl_int) :: m, n
type(sidl_BaseInterface_t) :: exception
real (kind=sidl_double), dimension(0:m-1, 0:n-1) :: A
real (kind=sidl_double), dimension(0:n-1) :: x
real (kind=sidl_double), dimension(0:m-1) :: b
```

```
recursive subroutine a_B_ex_mi(self, x, y, n, exception)
type(a_B_t) :: self
integer (kind=sidl_int) :: n
type(sidl_BaseInterface_t) :: exception
real (kind=sidl_double), dimension(0:n+1-1) :: x
real (kind=sidl_double), dimension(0:n-1-1) :: y
```

# How to use Babel objects that are already implemented

---

- Intrinsic capabilities and methods
  - Basic reference counting
  - Conway's game of life example (C++, C, F90 & Python)
  - Dynamic loading example (Python & F77)
  - Borrowed array example (C & C++)
  - rarray examples (C, F77, and F90)
  - Access without function calls
- ● Overview of basic rules

# Long and short names

---

- Long name includes packages  
`sidl.BaseClass.addRef`
- Short name is just the last part  
`addRef`
- Often Babel replaces ‘.’ with ‘\_’ to  
create a globally unique name  
`sidl_BaseClass_addRef`

# Overloading

---

- Methods can have overloading extensions, for example  
`double get[Part](in int partNo);`
- All languages except C++ and Java would use “getPart” as the method name

# Fortran 90 name length

---

- Fortran 90 names are limited to 31 characters
- `#include “sidl_BaseClass_fAbbrev.h”`
  - ▶ name mangling for `sidl.BaseClass`
- Preprocess your F90 with a C preprocessor (we use GCC everywhere)

# Special argument handling - C

---

- **in and inout argument should be initialized**
  - ▶ object/interface reference should be initialized to NULL or a valid object
- **inout and out parameters need pass by reference**
  - ▶ pass address of a argument using &

# Special argument handling - Python

---

- inout and out parameters are contained in the returned tuple

- Example:

```
int lookup(in int col, out int row)  
(result, row) = lookup(current)
```

- You can use positional or keyword args in Python

```
(result, row) = lookup(col = current)
```

# Extra arguments

---

- **self object parameter added to object methods for C, F77 & F90**
- **C adds “, out sidl.BaseInterface excpt)” to all object methods because all object methods can throw exceptions**
- **F77 & F90 add return value and exception as extra arguments (in that order)**

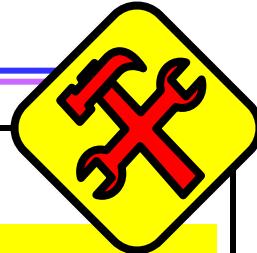
# Method naming for supported languages

C++	<b>Short method name</b>
Java	<b>Short method name</b>
C	<b>Long method name with _</b>
Fortran 77	<b>Long method name with _ and _f appended</b>
Fortran 90	<b>Short method name</b>
Python	<b>Short or long depending on import</b>

# Casting objects

- Failed casts produce a Null object
- Remember cast doesn't increment the reference count!

C++	<b>newt=oldt; // upcast</b>  <b>newt=::sidl::babel_cast&lt;newt&gt;(oldt)</b> <b>// safe downcast</b>
C	<b>new=x_y_z__cast(oldt);</b>
Java	<b>newt=(x.y.z)</b> <b>x.y.z._cast(oldt);</b>
F77	<b>call x_y_z__cast_f(oldt,</b> <b>newt)</b>
F90	<b>call cast(oldt, newt)</b>
Python	<b>newt = x.y.z.z(oldt)</b>



# Checking/initializing Null objects

---

- C++: if (**obj.\_not\_nil()**)  
    // born Nil
- C: if (**obj**)  
    **obj = NULL**; /\* init to Null object \*/
- Fortran 77: if (**obj .ne. 0**)  
    **obj = 0**
- Fortran 90: if (**is\_null(obj)**)  
    call **set\_null(obj)**
- Python: if (**obj**):  
    **obj = None**

---

---

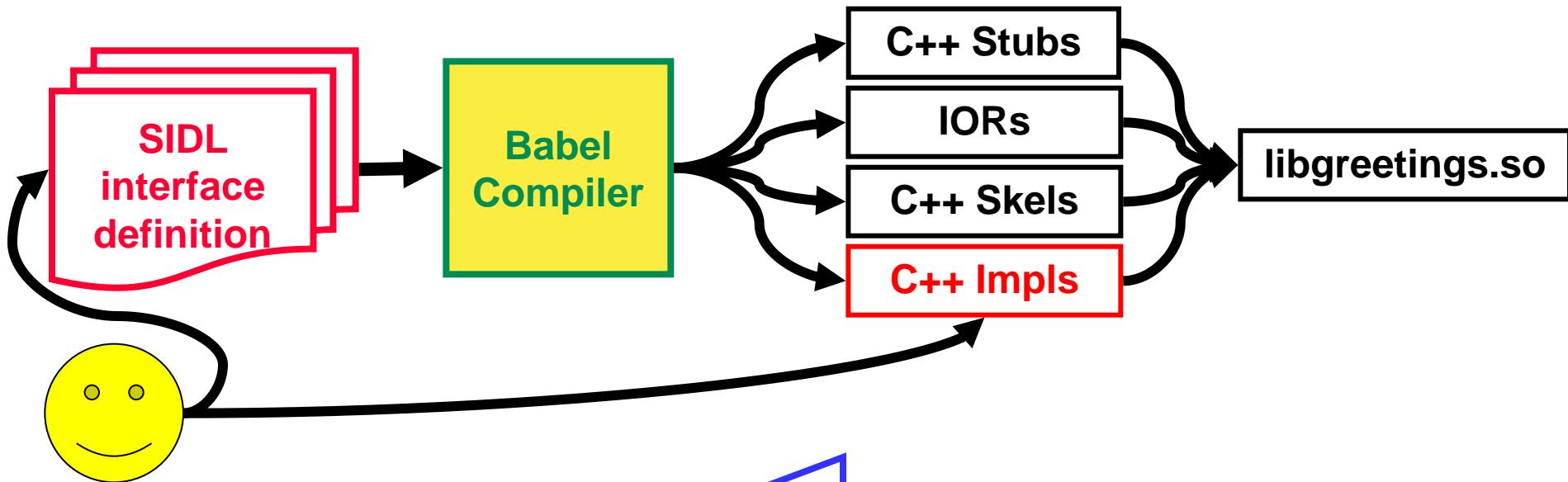
# V. Building Babel Libraries

# Scope of this Module

---

- **Example Implementations**
  - ▶ C++
  - ▶ C
  - ▶ Fortran 90
  - ▶ Python
- **Example of Wrapping Legacy Codes in Babel**
  - ▶ MPI\_send
- **Babel Build Tools & Techniques**
- **Packaging and Distribution**

# This Module for Implementers of a Babelized Library



1. Write SIDL File
2. `babel --server=C++ greetings.sidl`
3. Add implementation details
4. Compile & Link into Library/DLL

Section 2: SIDL Language

Section 3:  
Babel Tool

This Section

# greetings.sidl: A Sample SIDL File

```
① package greetings version 1.0 {  
②     interface Hello {  
        void setName( in string name );  
        string sayIt ( );  
    }  
③     class English implements-all Hello { }  
}
```

- Picked a very small example to show the implementations.
- Next several slides will show implementations of this interface in C, C++, Fortran 90 and Python

# F90/Babel “Hello World” Driver

```
program helloclient
use greetings_English
implicit none
type(greetings_English_t) :: obj
type( sidl_BaseInterface_t) :: ex
character (len=80) :: msg
character (len=20) :: name

name='world'
call new( obj, ex )
call setName( obj, name, ex )
call sayIt( obj, msg, ex )
call deleteRef( obj, ex)
print *, msg
end program helloclient
```

These subroutines  
come directly  
from the SIDL

Some other subroutines  
are “built in” to every  
SIDL class/interface

# A C++ Implementation

```
1 namespace greetings {
2 class English_impl {
3 protected:
4     // DO-NOT-DELETE splicer.begin(greetings.English._implementation)
5     ::std::string d_name;
6     // DO-NOT-DELETE splicer.end(greetings.English._implementation)
```

greetings\_English\_Impl.hxx

```
::std::string
greetings::English_impl::sayIt_Impl()
{
    // DO-NOT-DELETE splicer.begin(greetings.English.sayIt)
    ::std::string msg("Hello ");
    return msg + d_name + "!";
    // DO-NOT-DELETE splicer.end(greetings.English.sayIt)
}
```

greetings\_English\_Impl.cxx

# A C Implementation (1/4): The private data

```
1 struct greetings_English__data {  
2     /* DO-NOT-DELETE splicer.begin(greetings.English._data) */  
3     char * d_name;  
4     /* DO-NOT-DELETE splicer.end(greetings.English._data) */  
5 }
```

greetings\_English\_Impl.h

# A C Implementation (2/4): Allocate data in ctor

```
void  
① impl_greetings_English__ctor(greetings_English self,  
②                                 sidl_BaseInterface *_ex)  
{  
③     *_ex = 0;  
④     /* DO-NOT-DELETE splicer.begin(greetings_English._ctor) */  
⑤     struct greetings_English_data *dptr =  
⑥         malloc(sizeof(struct greetings_English_data));  
⑦     if (dptr) {  
⑧         dptr->d_name = NULL;  
⑨     }  
⑩     greetings_English__set_data(self, dptr);  
⑪     /* DO-NOT-DELETE splicer.end(greetings_English._ctor) */  
⑫ }
```

greetings\_English\_Impl.c

# A C Implementation (3/4): Deallocate Data in dtor

```
void
impl_greetings_English__dtor(greetings_English self,
                               sidl_BaseInterface *_ex)
{
    *_ex = 0;
⑦ /* DO-NOT-DELETE splicer.begin(greetings_English._dtor) */
    struct greetings_English__data *dptr =
        greetings_English__get_data(self);
    if (dptr) {
        if ( dptr->d_name != NULL ) {
            free( (void *) dptr->d_name );
        }
        memset(dptr, 0, sizeof(struct greetings_English__data));
        free( (void *) dptr );
    }
    /* DO-NOT-DELETE splicer.end(greetings_English._dtor) */
}
```

# A C Implementation (4/4): Implement the Method

```
char *
impl_greetings_English_sayIt( greetings_English self,
                               sidl_BaseInterface *_ex )
{ *_ex = 0;
① /* DO-NOT-DELETE splicer.begin(greetings.English.sayIt) */
    struct greetings_English_data dptr =
        greetings_English_get_data( self );
    char[1024] buffer = "Hello ";
    if (dptr->d_name) {
②     strncat( buffer, dptr->cname, 1017 );
        strncat( buffer, "!", 1017 - strlen(dptr->d_name));
    }
④ return sidl_String_strdup( buffer );
/* DO-NOT-DELETE splicer.end(greetings.English.sayIt) */
}
```

# Fortran 90 Impl (1/4): Add state to \*Mod.F90

```
❶ #include "greetings_English_fAbbrev.h"
❷ module greetings_English_impl
    ! DO-NOT-DELETE splicer.begin(greetings.English.private_data)
    character (len=1024) :: d_name
    ! DO-NOT-DELETE splicer.end(greetings.English.private_data)
end type greetings_English_private

❸ type greetings_English_wrap
    sequence
        type( greetings_English_Private ), pointer :: d_private_data
    end type greetings_English_wrap

end module greetings_English_impl
```

greetings\_English\_Mod.F90

# Fortran 90 Impl (2/4): Implement subroutines

```
1 recursive subroutine greetings_English_sayIt_mi(self, retval &
exception )
2   use sidl_BaseInterface
3   use greetings_English
4   use greetings_English_impl
5   ! DO-NOT-DELETE splicer.begin(greetings.English.sayIt.use)
6   ! DO-NOT-DELETE splicer.end(greetings.English.sayIt.use)
7   implicit none
8   type(greetings_English_t) :: self ! in
9   character (len=*) :: retval ! out
10  type(sidl_BaseInterface_t) :: exception ! out
11
12  ! DO-NOT-DELETE splicer.begin(greetings.English.sayIt)
13  type(greetings_English_wrap) :: dp
14  call greetings_English__get_data_m(self, dp)
15  retval = 'Hello ' // dp%d_private_data%d_name // '!'
16  ! DO-NOT-DELETE splicer.end(greetings.English.sayIt)
17 end subroutine greetings_world_sayIt_mi
```

# Fortran 90 Impl (3/4):

## Allocate private\_data in ctor

```
recursive subroutine greetings_English__ctor_mi(self, exception)
use sidl_BaseInterface
use greetings_English
use greetings_English_impl
! DO-NOT-DELETE splicer.begin(greetings.English._ctor.use)
! DO-NOT-DELETE splicer.end(greetings.English._ctor.use)
implicit none
type(greetings_English_t) :: self ! in
type(sidl_BaseInterface_t) :: exception ! out

! DO-NOT-DELETE splicer.begin(greetings.English._ctor)
⑥ type(greetings_English_wrap) :: dp
allocate(dp%d_private_data)
dp%d_private_data%d_name = ''
call greetings_English__set_data_m(self, dp)
! DO-NOT-DELETE splicer.end(greetings.English._ctor)
end subroutine greetings_English__ctor_mi
```

# Fortran 90 Impl (4/4): Release private\_data in dtor

```
recursive subroutine greetings_English_dtor_mi(self, exception)
use sidl_BaseInterface
use greetings_English
use greetings_English_impl
! DO-NOT-DELETE splicer.begin(greetings_English._dtor.use)
! DO-NOT-DELETE splicer.end(greetings_English._dtor.use)
implicit none
type(greetings_English_t) :: self ! in
type(sidl_BaseInterface_t ) :: exception ! out

! DO-NOT-DELETE splicer.begin(greetings_English._dtor)
⑦ type(greetings_English_wrap) :: dp
call greetings_English_get_data_m(self, dp)
deallocate(dp%d_private_data)
! DO-NOT-DELETE splicer.end(greetings_English._dtor)
end subroutine greetings_English_ctor_mi
```

# A Python Implementation

```
❶ class English:

❷     def __init__(self, IORself=None):
        if (IORself == None):
            #handle a rare case
        else:
            self.__IORself = IORself
        # DO-NOT-DELETE splicer.begin(__init__)
❸         self.d_name = ''
        # DO-NOT-DELETE splicer.end(__init__)

    def sayIt(self):
        # DO-NOT-DELETE splicer.begin(sayIt)
❹         return 'Hello ' + self.d_name + '!'
        # DO-NOT-DELETE splicer.end(sayIt)

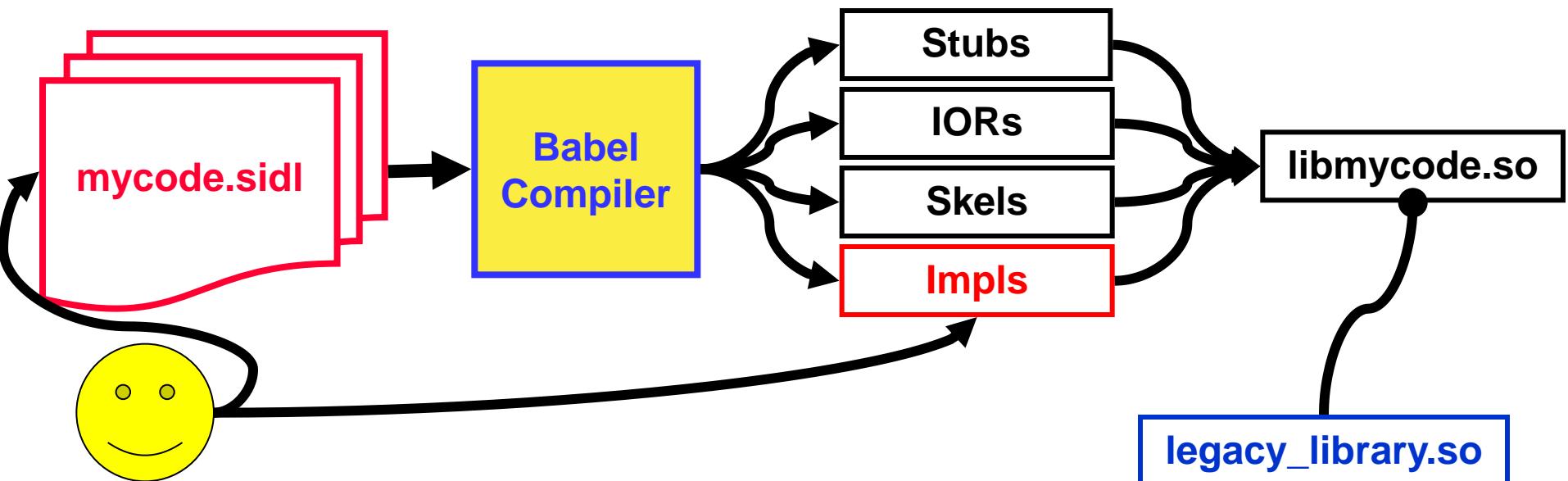
    def setName(self, name):
        # DO-NOT-DELETE splicer.begin(sayIt)
❺         self.d_name = name
        # DO-NOT-DELETE splicer.end(sayIt)
```

# Additional Splicer Blocks

---

- Generic splicer blocks usually appear at the beginning and end of an IMPL file
  - ▶ e.g. `_include` and `_misc`
- `_ctor2` (aka “Back door constructor”)
  - ▶ Builds temporary Babel wrappers around existing instances
- `_load`
  - ▶ Guaranteed to run exactly once and before any other splicer block for that type
  - ▶ Useful for initializing singletons

# Applying Babel to Legacy Code



1. Write your SIDL interface
2. Generate server side in your native language
3. Edit Implementation (Impls) to dispatch to your code  
(Do NOT modify the legacy library itself!)
4. Compile & Link into Library/DLL

# Example of Babelized Legacy Code: MPI

```
package mpi version 2.0 {
    class Comm {
        int send[Int]( in array<int,1,row-major> data,
                      in int dest, in int tag );
        int send[IntR]( in rarray<int,1> data(n),
                        in int n,
                        in int dest, in int tag );
        ...
    }
}
```

mpi.sidl

## API choices made in this example:

- Operations are methods on MPI Comm's
- Overloaded methods based on scalar type
- Use Babel arrays instead of buffer and count
  - ▶ “row-major” (or “column-major”) also guarantees non-strided, even for 1-D arrays.
  - ▶ Added rarray for comparison

# Example of Babelized Legacy Code (MPI): The \*Impl.h

```
/* DO-NOT-DELETE splicer.begin(MPI_Comm._includes) */
#include "mpi.h"
/* DO-NOT-DELETE splicer.end(MPI_Comm._includes) */
...
struct mpi_Comm_data {
    /* DO-NOT-DELETE splicer.begin(MPI_Comm._data) */
    MPI_Comm com;
    /* DO-NOT-DELETE splicer.end(MPI_Comm._data) */
}
```

mpi\_comm\_Impl.h

- **MPI is a C standard, so implement the Babel wrappers in C.**
- **New Communication Objects have state**
  - ▶ For C state is kept in a \*\_data struct.
  - ▶ Remember to observe splicer blocks

# A Babelized MPI Using Packed Arrays

```
int32_t
impl_mpi_Comm_sendInt( mpi_Comm self,
                        struct sidl_int_array* data,
                        int32_t dest, int32_t tag,
                        sidl_BaseInterface *_ex)
{ *_ex = 0;
  /* DO-NOT-DELETE splicer.begin(MPI.Comm.sendInt) */
  struct mpi_Comm_data *dptr = mpi_Comm_get_data( self );
  void * buff = (void*) sidl_int_array_first(data);
  int count = sidl_int_array_length(data, 0);
  return mpi_send( buff, count, MPI_INT, dest, tag, dptr->comm);
  /* DO-NOT-DELETE splicer.end(MPI.Comm.sendInt) */
}
```

- CAUTION: Assumes MPI\_INT corresponds to 32-bit integers! mpi\_comm\_Impl.c
- Since array is 1-D and unstrided, use address of first element as buffer

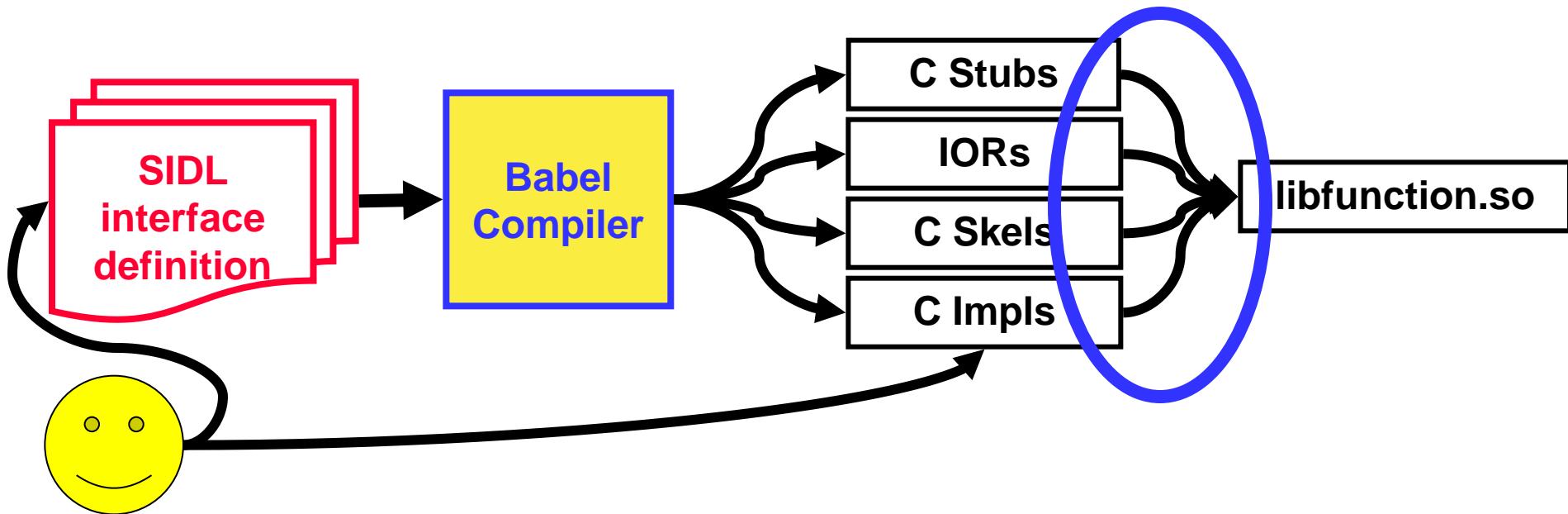
# A Babelized MPI using Raw Arrays

```
int32_t  
impl_mpi_Comm_sendInt( mpi_Comm self, int32_t * data,  
                        int32_t n, int32_t dest, int32_t tag  
                        sidl_BaseInterface *_ex)  
{ *_ex = 0;  
/* DO-NOT-DELETE splicer.begin(MPI_Comm.sendInt) */  
struct mpi_Comm_data *dptr = mpi_Comm_get_data( self );  
return mpi_send( (void*) data, count, MPI_INT, dest,  
                 tag, dptr->comm);  
/* DO-NOT-DELETE splicer.end(MPI_Comm.sendInt) */  
}
```

mpi\_comm\_Impl.c

- **CAUTION: Assumes MPI\_INT corresponds to 32-bit integers!**
- **Raw arrays fit very well with legacy C libraries.**

# What's the Hardest Part of this Process?



- “The Build”

- ▶ Properly compiling and linking the libraries
- ▶ especially dynamically loadable .so files.

# Compile and Link Is Tricky for Several Reasons

---

- Mixed language context is less forgiving
  - ▶ extra diligence needed to resolve linker symbols
  - ▶ After compilation, no guarantee that linker will be launched with same language in mind.
- Poor tools to support and debug dynamic loaded libraries
- Little agreement among UNIX vendors on how to deal with three kinds of linkage
- Babel's own build gets things right, but users need to reproduce this effect in every “Babelized” library

# **Configuration For 6+ Languages On Any Unix Is A Challenge**

---

**Babel's Own Build Has...**

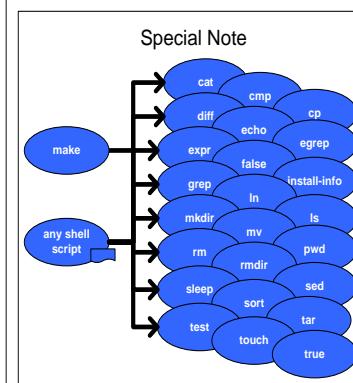
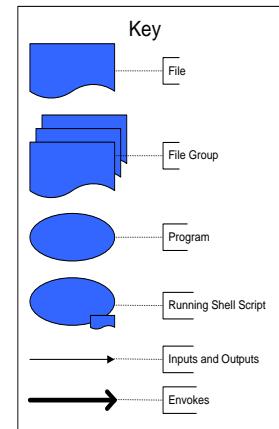
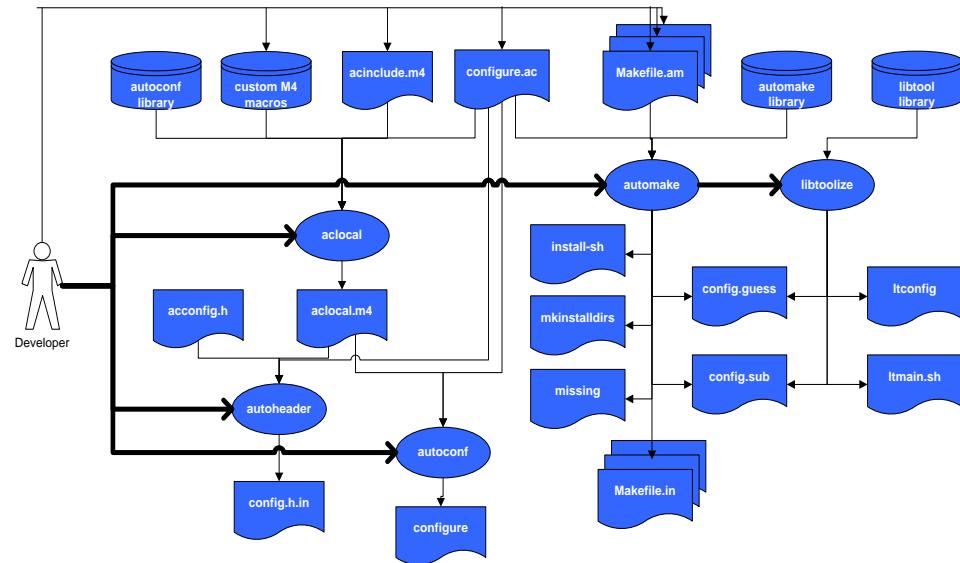
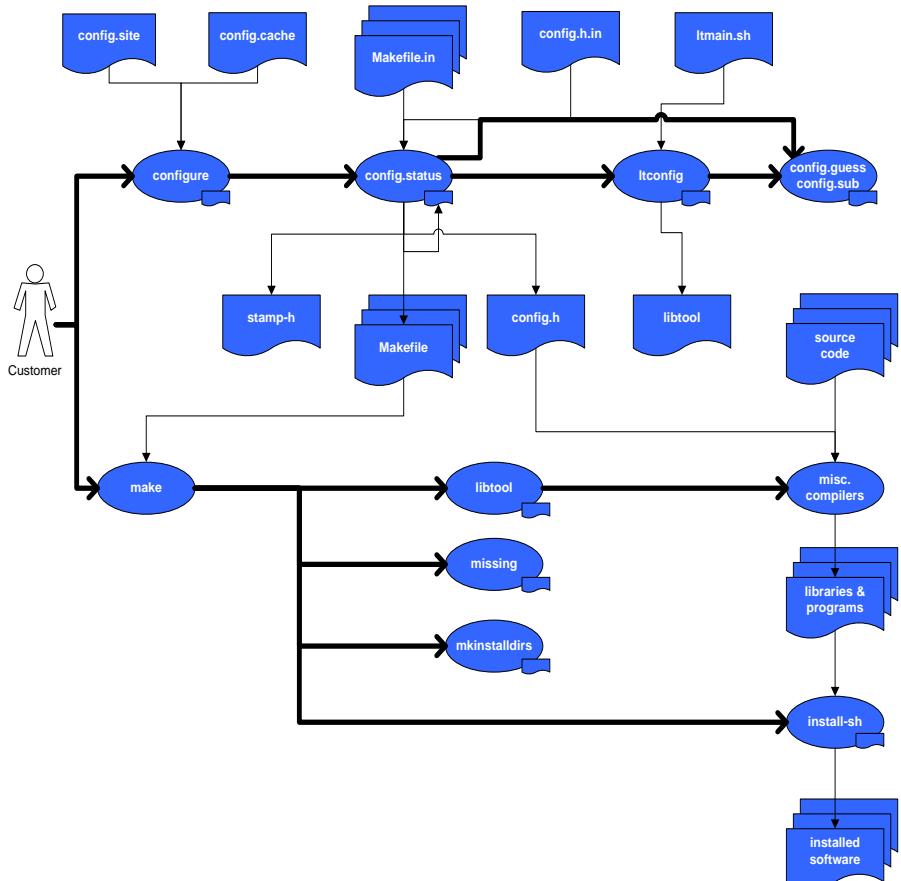
**>48K lines of configure script (x2)**

**>58 custom autoconf macros**

**>120 configuration settings exported  
to users via babel-config**

**>193 #defines in babel\_config.h**

# GNU Autotools Is Too Complex to Impose on Babel Users



# We Provide Multiple Tools To Assist in Your Builds

---

- Specific Tools & Assists

- ▶ **babel.make** files (generated by Babel itself)
- ▶ **babel-config** script (query Babel's configuration)
- ▶ **babel-libtool** script (helps build dynamic libraries)
- ▶ **babel-cc** compiler front-end  
(supercedes mpicc)
- ▶ **LLNL\_PROG\_BABEL** M4 macro for customers using autoconf

- No “one size fits all” solution

- ▶ Customer builds are highly specialized

- Custom combinations of these tools are broadly effective

# Use babel-stamp to workaround a make deficit

- Problem: Make assumes one action produces one file.
  - ▶ Babel generates multiple source files from a single SIDL file
  - ▶ Don't want make running the same Babel command multiple times
- Solution: The following trick using a proxy file and recursion

```
babel-stamp : $(SIDLFILES)
    $(RM) -f babel-temp
    touch babel-temp
    $(BABEL) $(BABEL_ARGS) $(SIDLFILES)
    $(MV) -f babel-temp babel-stamp

$(ALL_SRCS) $(ALL_HDRS) : babel-stamp
    @if test -f $@; then \
        touch $@; \
    else \
        $(RM) -f babel-stamp; \
        $(MAKE) $(MAKEFLAGS) babel-stamp; \
    fi
```

# babel.make: A makefile fragment Babel creates

- **Problem:** Names of Babel-generated source files to compile determined by SIDL file and change often
  - ▶ Make is static and wants the filenames listed
- **Solution:**
  - ▶ Each compiled language binding in Babel will generate a babel.make file along with source code.

```
IMPLHDRS = Hello_World_Impl.h
IMPLSRCs = Hello_World_Impl.c
IORHDRS = Hello_IOR.h Hello_World_IOR.h
IORSRCs = Hello_World_IOR.c
SKELSRCS = Hello_World_Skel.c
STUBHDRS = Hello.h Hello_World.h
STUBSRCs = Hello_World_Stub.c
```

- ▶ Add “include babel.make” in your Makefile
- ▶ Variable names depend on language and whether you’re doing client or server

# Use Macro Renaming and Suffix Rules in Makefiles

- We use these techniques a lot in our Makefiles

```
IMPOBJS = $(IMPLSRCS:.c=.o)
ILOBJS = $(IORSRCS:.c=.o)
SKELOBJS = $(SKELSRCS:.c=.o)
STUBOBJ = $(STUBSRCS:.c=.o)

.SUFFIXES:
.SUFFIXES: .c .o

.c.o:
$(CC) -c $< -o $@
```

- Remember to preprocess F90

```
.SUFFIXES: .F90 .o
.F90.o:
$(CPP) $(INCLUDES) -P -o $(@:.o=.f90) -x c $<
$(F90COMPILE) -c -o $@ $(@:.o=.f90)
rm -f $(@:.o=.f90)
```

# Use “babel-config” to query Babel’s configuration

---

```
% babel-config --jardir  
/you/installed/it/here/jar  
% babel-config --with-f90 && echo $?  
0  
% babel-config --libs-f77-L  
-L /some/wierd/dir/lib -L/other/f77/lib  
% babel-config --dump-vars=X_  
X_PACKAGE=babel  
X_VERSION=1.2.0  
X_CONFIGPATH=/some/path
```

...

# **babel-libtool homogenizes library linking flags**

---

- Too arcane to go into detail here.
- It is a slightly modified version of GNU libtool
  - ▶ Needed more flexibility on AIX
- We “install” the libtool script in Babel’s \$bindir
  - ▶ Also not the GNU intended use, but useful
- Used by the babel-cc scripts following

# **babel-cc, babel-cxx, babel-f90, and babel-f77 scripts**

---

- **Modifies your compiler arguments and adds Babel flags**

- ▶ Inspired by “mpicc” compiler front end
  - ▶ But very different in practice

```
% babel-cc --no-quiet -c test.c  
babel-libtool --tag=CC --mode=compile gcc \  
    -I/path/to/babel/include test.c  
  
% babel-cc --no-quiet-libtool -c test.c  
gcc -c -I/path/to/babel/include test.c -fPIC \  
    -DPIC -o .libs/test.o  
gcc -c -I/path/to/babel/include test.c \  
    -o test.o >/dev/null 2>&1
```

# The babel-cc approach is newer and not for everyone

---

- Pros:

- ▶ Keeps Makefiles small & simple
- ▶ Protects against common mistakes
- ▶ Effective on toy codes

- Cons:

- ▶ Uses Libtool (unusual idioms)
- ▶ User surrenders some control
- ▶ Not “battle hardened” like rest of Babel (yet)

# LLNL\_PROG\_BABEL: For users who want to use autoconf

---

- This is a M4 macro for autoconf
  - ▶ Copy the file into your source tree
  - ▶ List the file to acinclude.m4
  - ▶ Edit your configure.ac file and put the macro before AC\_PROG\_CC.
- Convenient way to pre-initialize all autoconf variables with Babel defaults
- Built on top of babel-config

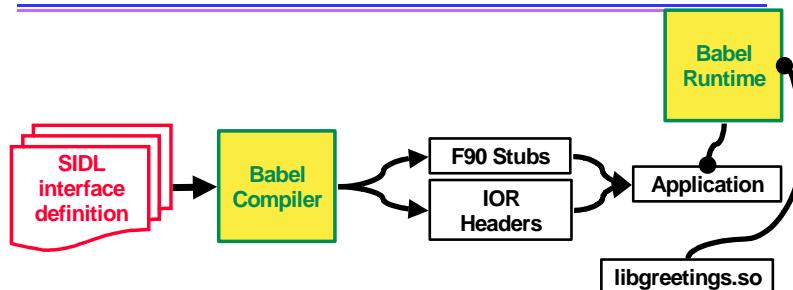
# Multiple approaches to see how Babel built DLL's

---

- 1. Watch our regression tests build**
  - make check, delete one library, make check
- 2. Try using “babel-libtool”**
  - Libtool is an obscure tool and may be more confusing to learn than its worth
- 3. Read the “Advanced Topics” section of Babel User’s Guide**
  - Page on “linkers, loaders, and PIC” is most downloaded page of the manual
- 4. email [babel-users@llnl.gov](mailto:babel-users@llnl.gov)**

# There's more than one way to distribute Babelized Code

Library User Does This...

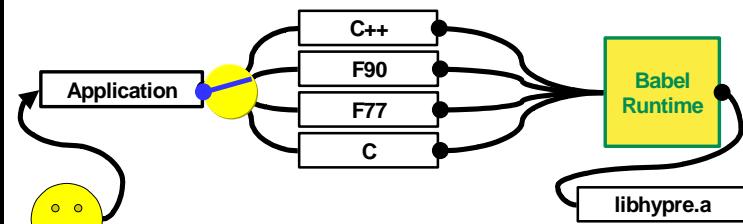


1. `babel --client=F90 greetings.sidl`
2. Compile & Link generated Code & Runtime
3. Place DLL in suitable location

BABEL

4

hypre Users Do This



1. -I directory of your calling language
2. -L directory of your calling language
3. -lhypre -lsidl

BABEL

38

- **hypre wants their customers to be relatively unaware they're using Babel.**
  - They pre-generate compiled language bindings
  - They ship Babel's runtime bundled with hypre

# Babel Distributions: Developer Kit vs. Runtime

---

- Runtime subdirectory in the developer kit has its own configure script.
  - ▶ It gets called by top level configure script and becomes part of babel-x.y.z distro.
  - ▶ Calling it directly from the command line configures it for separate distribution
- Essentially, Babel uses its own runtime subdirectory the same way customers can bundle the Babel Runtime with their distros.

# Lots More Information In Babel Users' Guide (aka BUG)

- Fine tuning your file layout
- Primer Static and Dynamic Linkage
- Platform specific details
- Different strategies for mixing generated and hand-written code in CVS

# Module Review

---

- **Example Implementations**
  - ▶ C++
  - ▶ C
  - ▶ Fortran 90
  - ▶ Python
- **Example of Wrapping Legacy Codes in Babel**
  - ▶ MPI\_send
- **Babel Build Tools & Techniques**
- **Packaging and Distribution**

---

---

# **VI. Remote Method Invocation**

# Scope of This Module

---

- Motivation
- Why RMI is superior to RPC
- Goals for Babel RMI
- Specifics to support RMI in Babel
  - ▶ SIDL keywords,
  - ▶ sidl.rmi package
  - ▶ builtin methods
- Example: Multiscale Material Science
- Protocols

# Motivation for Remote Method Invocation

---

## ● Research

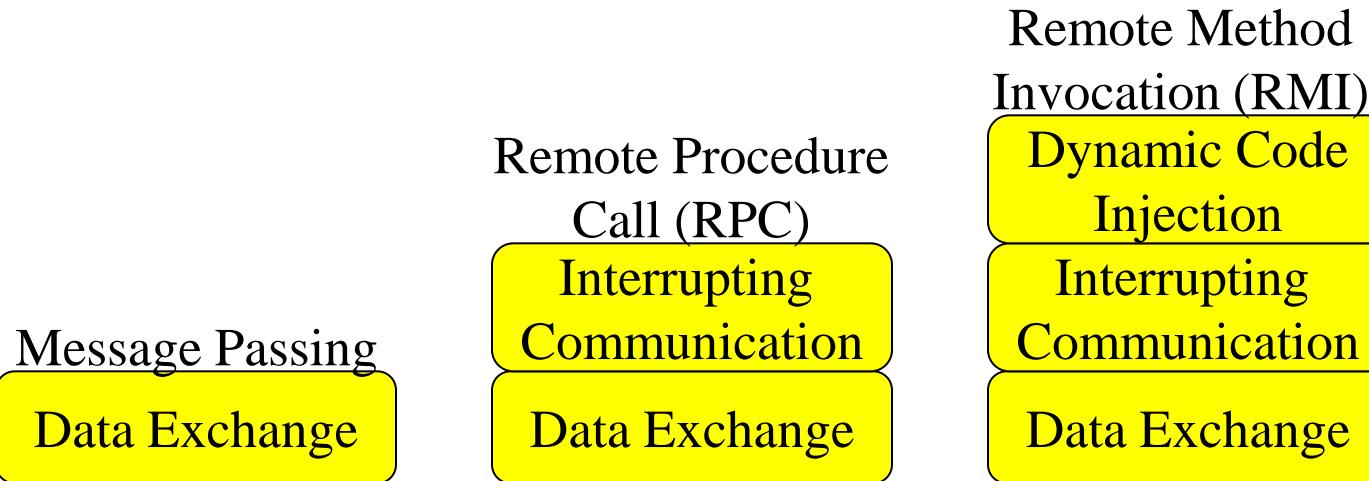
- ▶ CCA's “MxN” problem. (SciDAC 1)
  - Bertrand et. al. Data redistribution and remote method invocation for coupled components. *JPDC*. 66(7). July 2006, pp 931-946.
- ▶ Petascale Simulation Initiative (LLNL R&D)
  - Make whole 1000 processor jobs a single component in a federated simulation.

## ● Practical Benefits

- ▶ Workaround for nearly impossible situations
  - Legacy codes will not port
  - Two codes interfere with each other's linker symbols
- ▶ Provides communication modes not covered by MPI... truly asynchronous and interrupting

# Key Technical Argument: RMI is more than RPC

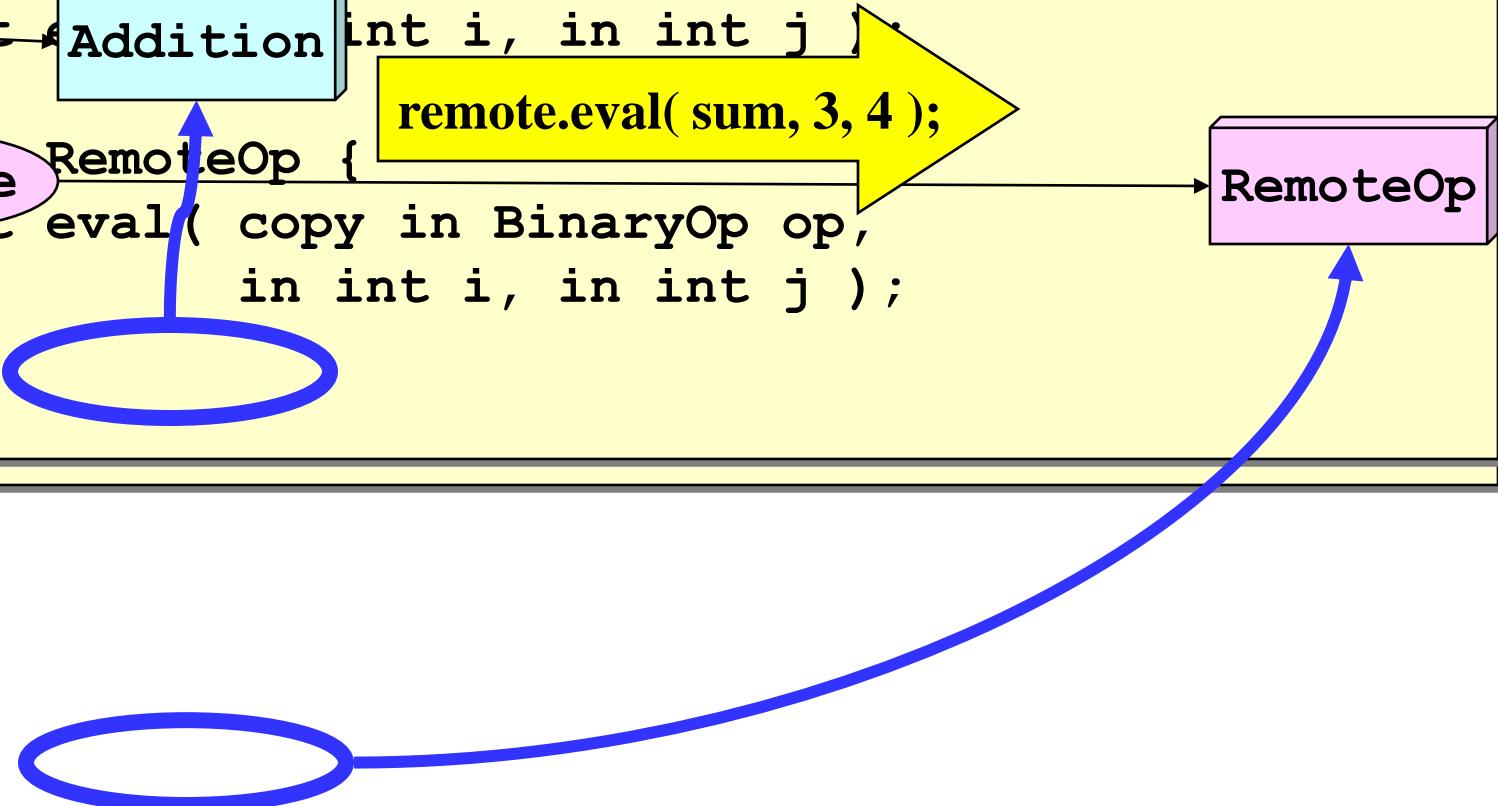
---



- **Consistent OOP Model cannot be built on RPC**
- **Dynamic Code Injection is required to properly support polymorphism**
- **Consequently CORBA & DCOM are object-like**
- **Java RMI and Babel RMI are different**
  - ▶ see also: Jim Waldo. **Remote Procedure Calls and Java Remote Method Invocation.** *IEEE Concurrency*. 6(3). 1998. pp 5-7.

# Do More With RMI Than RPC: A Single, Concrete Example

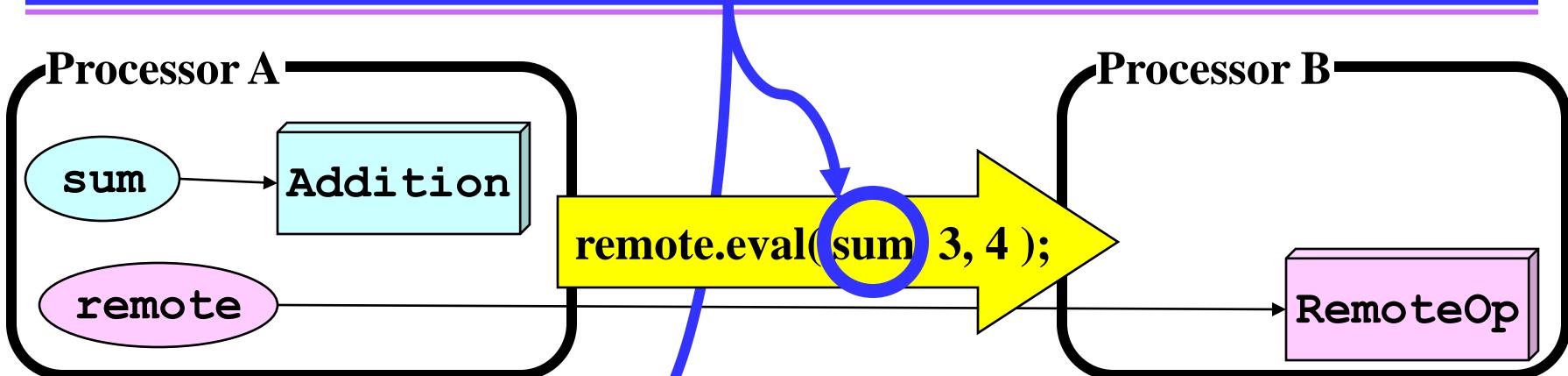
```
package example version 1.0 {  
    interface BinaryOp extends sidl.io.Serializable {  
        int sum( int i, int j );  
    }  
    class Addition implements BinaryOp {  
        public int sum( int i, int j ) {  
            return i + j;  
        }  
    }  
    interface RemoteOp extends sidl.io.Serializable {  
        int eval( copy in BinaryOp op,  
                  in int i, in int j );  
    }  
}
```



# Do More As Less

Though the method is implemented in terms of an abstract interface, it requires copy of the concrete type.

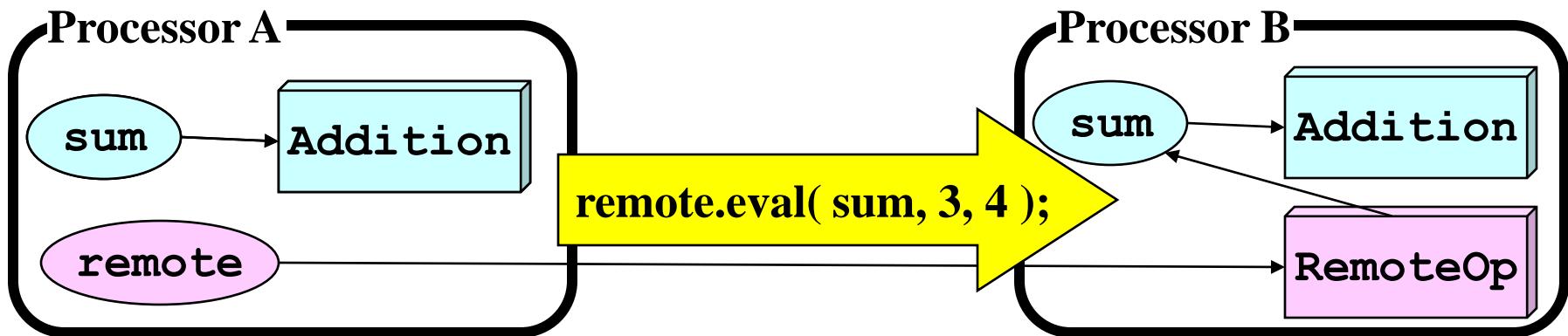
# an RPC: example



```
package mine version 1.0 {  
    class Addition implements-all example.BinaryOp {}  
}
```

```
package example version 1.0 {  
    interface BinaryOp extends sidl.io.Serializable {  
        int eval( in int i, in int j );  
    }  
    class RemoteOp {  
        int eval( copy in BinaryOp op,  
                 in int i, in int j );  
    }  
}
```

# RMI Requires Runtime Code Injection Cannot Mimic This With RPC



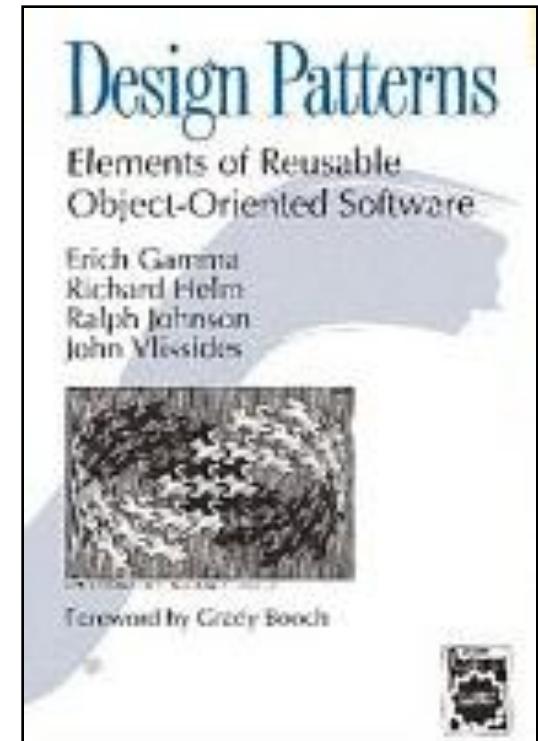
```
package mine version 1.0 {  
    class Addition implements-all example.BinaryOp {}  
}
```

```
package example version 1.0 {  
    interface BinaryOp extends sidl.io.Serializable {  
        int eval( in int i, in int j );  
    }  
    class RemoteOp {  
        int eval( copy in BinaryOp op,  
                 in int i, in int j );  
    }  
}
```

# With RMI, OO Design Patterns Extend to Distributed Regime

---

- Not particularly news to the Java world
- Corba & DCOM really are RPC
  - ▶ C/C++/Fortran programmers have had to do without
- Babel brings RMI to these languages
  - ▶ Relies on DLLs in file system instead of encoding bytecodes on the wire



# Goals for Babel RMI

---

- **Transparency**

- ▶ Same “look and feel” for local and remote objects
- ▶ Easy transition for existing customers

- **Generality**

- ▶ Actively encourage 3<sup>rd</sup> party protocols
  - Defined a Babel RMI API in SIDL
  - Distribute a TCP/IP reference implementation

# Important Design Choices

---

- **Multithreaded**

- ▶ RMI Servers are analogous to Web Servers
- ▶ Implementors are required to make their objects thread-safe
  - workaround: limit the threadpool of the server to one
  - side effect: prone to deadlock with circular RMI
- ▶ ergo: RMI is not an atomic operation

- **No Network Security!**

- ▶ Babel RMI niche is within a single cluster
  - rely on security of batch systems and firewalls
  - allows protocols to be leaner & faster
- ▶ For untrusted networks, Babel RMI needs more
  - Cybersecurity experts welcome

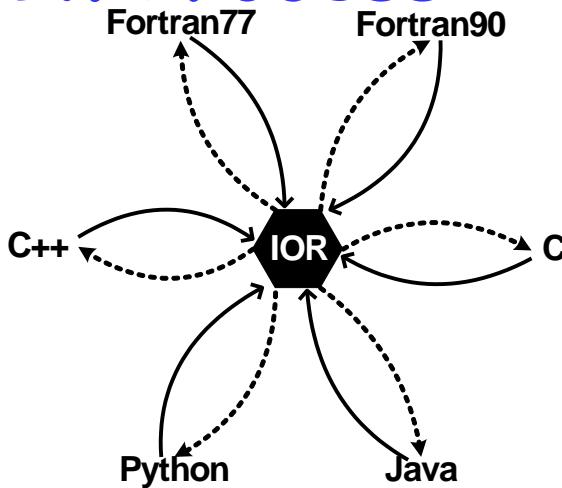
# Parts of Babel System Specific to RMI

---

- **Remote IOR**
- **Builtin Functions (Stubs)**
  - ▶ `_create[Remote]( in string url )`
  - ▶ `_connect( in string url )`
  - ▶ `_isLocal() / _isRemote()`
  - ▶ `_getURL()`
  - ▶ `_exec( in string name, in Deserializer inArgs, in Serializer outArgs )`
- **sidl.rmi package**
- **IDL Keywords**
  - ▶ **nonblocking** – modifies a method
  - ▶ **oneway** – modifies a method
  - ▶ **copy** – modifies an argument

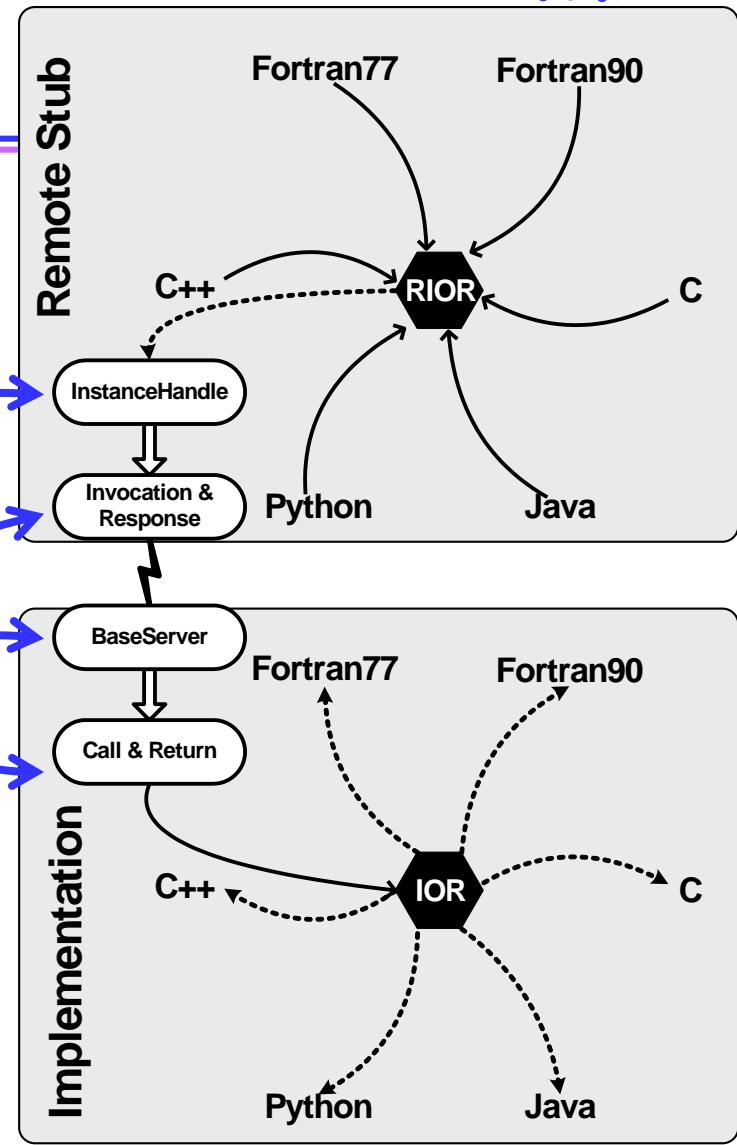
# Remote IOR issues calls to InstanceHandle instead of Skels

## In Process



Key: Stub Skel

Network Layer  
defined in  
SIDL for  
3rd party  
plug-ins



# Wire Protocols Are Registered Using Abstract Factory Pattern

---

```
sidl::rmi::ProtocolFactory::addProtocol(  
    "simhandle", "sidlx.rmi.SimHandle");
```

- This associates URLs beginning with “simhandle” to class that implements the `sidl.rmi.InstanceHandle` interface
- `sidlx` is a package name we use for project experiments
- `sidlx.rmi.SimHandle` is the TCP/IP-based reference implementation

# Need to Also Launch & Register a Server

```
string url = "simhandle://localhost:" +port;
sidlx::rmi::SimpleOrb orb =
    sidlx::rmi::SimpleOrb::_create();
orb.init(url, 1);
int64_t tid = orb.run();
sidl::rmi::ServerRegistry::registerServer( orb );
```

- This Server is multithreaded and uses a thread pool.
- Strongly advise thread-safe implementations
- Can limit thread pool to 1 thread, but runs the risk of cyclic deadlock

`_create[Remote]( in string url )`  
`& _connect( in string url)`

---

- Creates a remote instance or connects to a known instance.
- “url” here is a misnomer
  - ▶ Reference Implementation uses URL format, but Babel is more general
  - ▶ ProtocolFactory lookup key is everything before the first non alphanumeric
  - ▶ Matching protocol interprets rest of string as it sees fit.

# Once Connected, Local & Remote Objects Behave Same

---

- Added `_isLocal()` and `_isRemote()` to distinguish
- `_getURL()`
  - ▶ generates a string for remote handles to `_connect()` to
  - ▶ has a useful side-effect of registering objects in an instance registry

# The `copy` Keyword & `sidl.io.Serializable`

---

- In RMI, Most Types are Pass-By-Copy
  - ▶ Fundamental Types (`int`, `float`, etc.)
  - ▶ Aggregate Types (`array`, `enum`, & `struct`)
  - ▶ Note: opaque not so useful
- In RMI, Objects are Pass-by-Reference
  - ▶ Exception: iff both are met
    - Object implements `sidl.io.Serializable` and
    - `copy` keyword in argument list
- Exceptions are semantically equivalent to “copy out” parameters.

# Asynchrony with **nonblocking** and **oneway** modifiers

---

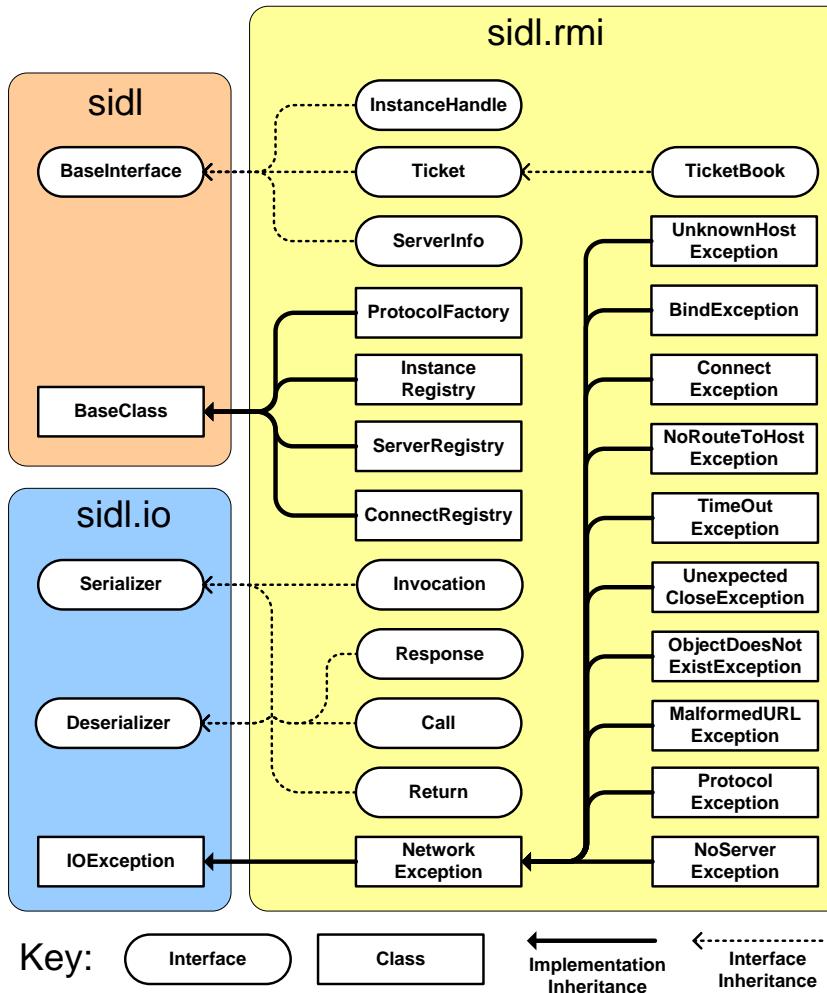
- **nonblocking:**

- ▶ Babel generates 3 stubs for each nonblocking method, Z = foo(in A, inout B, out C ).
  - Z = foo(A, B, C) <- normal blocking form
  - Ticket t = foo\_send(A, B) <- nonblocking send
  - Z = foo\_recv( t, B, C ) <- nonblocking recv
- ▶ Caution: Protocol may support syntax but implement in blocking calls (or vice versa).

- **oneway:**

- ▶ Generates sid1.RuntimeExceptions iff reliable delivery cannot be guaranteed.
- ▶ Restrictions: No out or inout parameters, return void only, no user exceptions.
- ▶ Fire and forget.

# Heirarchy of Objects in sidl.rmi package



- The Interfaces in sidl.rmi are specific to implementing a wire-protocol in Babel RMI

- ▶ TCP/IP – built-in
- ▶ PSP – LLNL
- ▶ IIOP – Tech-X
- ▶ SOAP – SUNY Binghamton
- ▶ RMIX – GA Tech
- ▶ others...

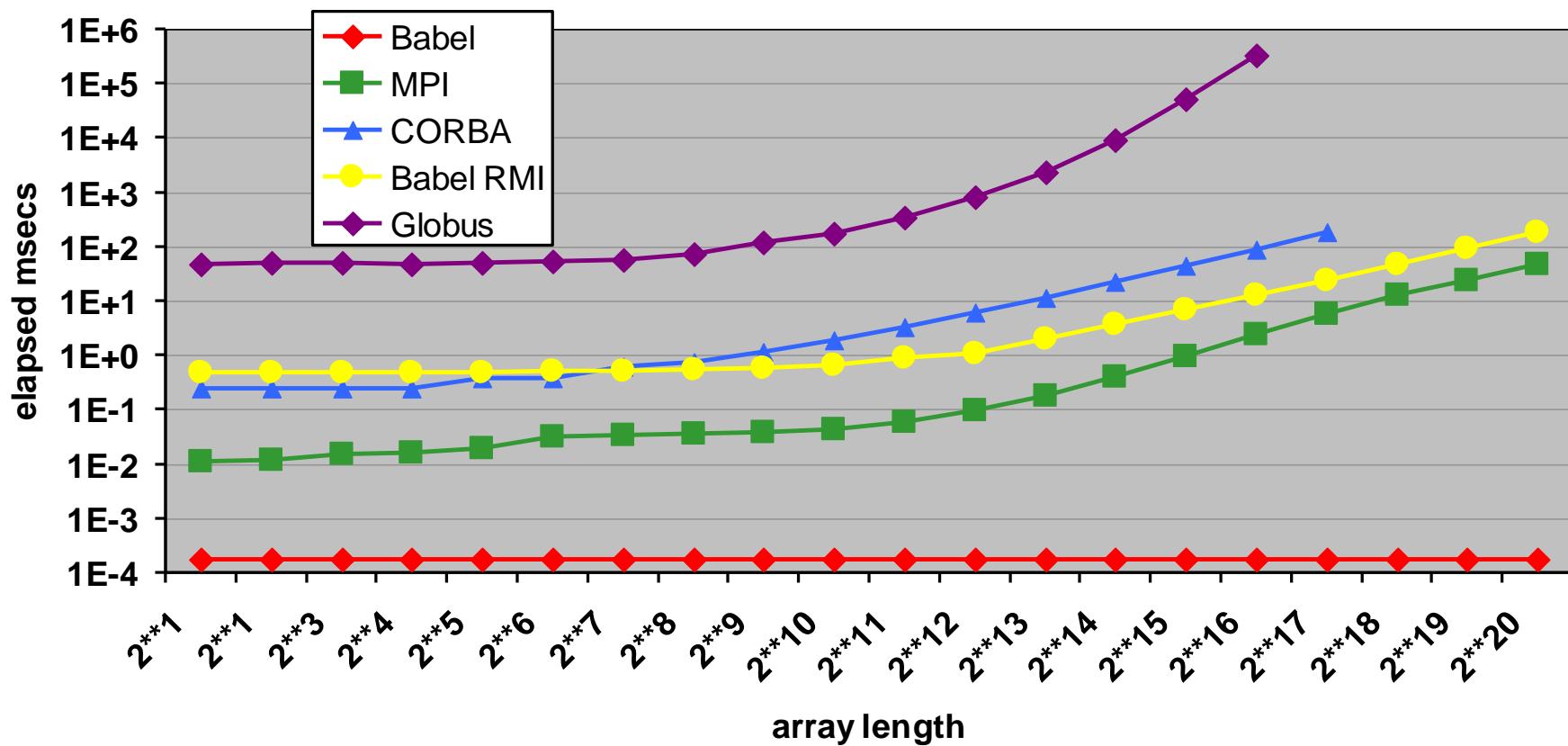
# Babel RMI is Protocol Neutral: Reference Implementation is TCP

Supporting Middleware	μsecs
Babel: in process (C)	0.030
MPI: ping-pong on elan3 (C)	9.43
CORBA: omniORB (C++)	251
Babel RMI: Simple TCP/IP (C++)	609 <b>375 in Babel 1.1</b>
Globus 4.0 WS Core: no security (Java)	28,000

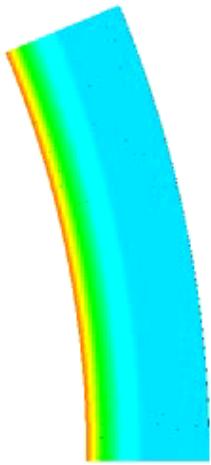
Round trip latency of a no-op on 3.06 GHz Intel P4  
Xeons with Elan3 switch using Intel 9.1 compilers  
and Babel 1.0.1

# Simple Scaling Study using Babel RMI's Reference Impl.

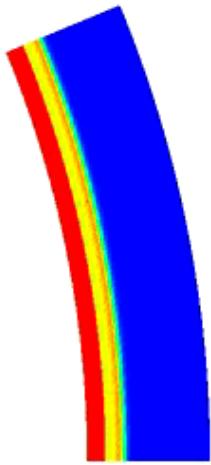
inout array<double>



# 2-D Material Failure of Shock-Driven Cylinder, /w Babel RMI



pressure



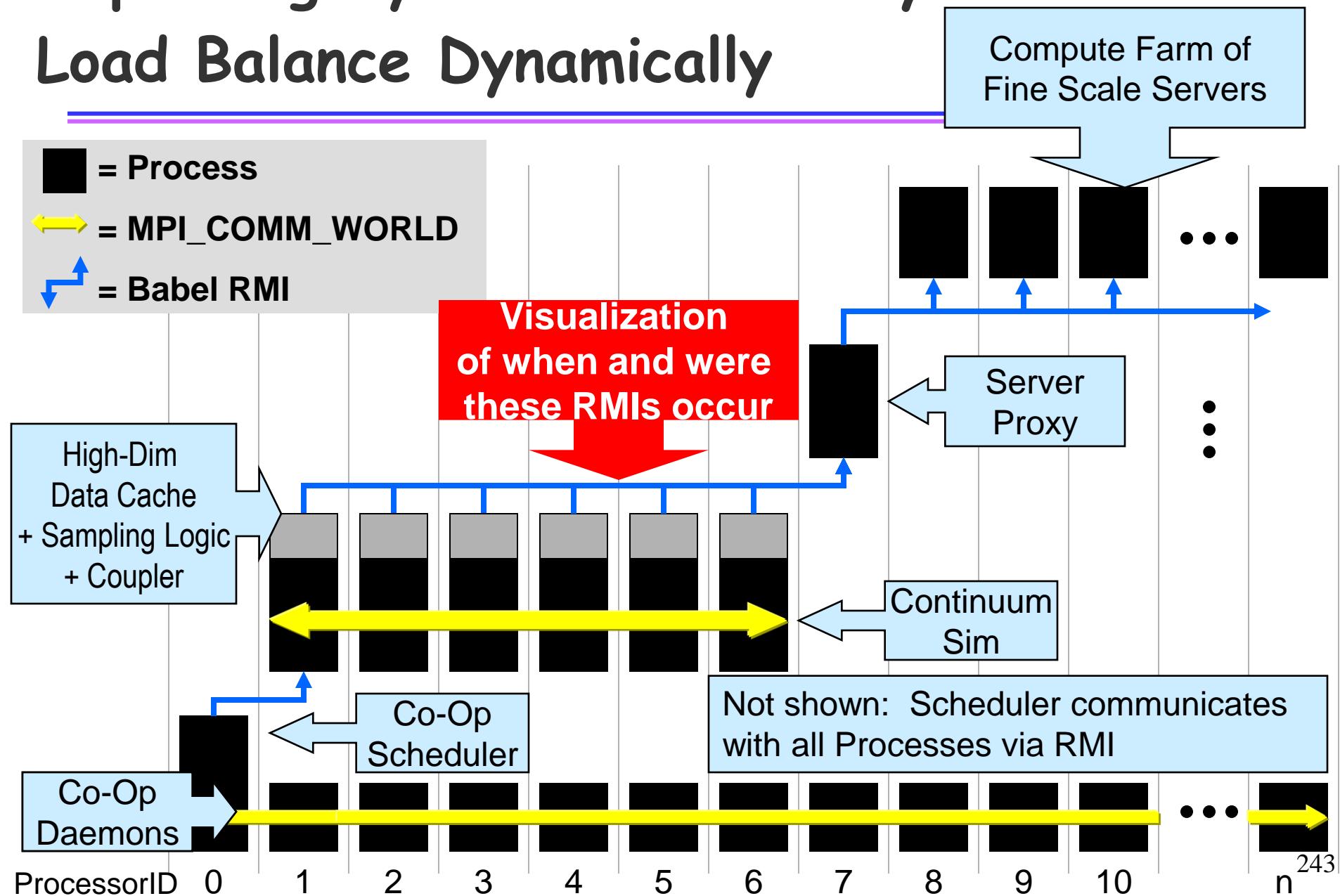
Plastic strain rate



fine-scale evaluations

- Simulation & Animation by Nathan Barton, LLNL (Engr.)
- 140 Processors, 33 wallclock hours
- 1/3 cluster doing macroscale (continuum C code)
- Other 2/3 machine is a fine-scale material compute farm (experimental crystal-plasticity models in Fortran 90)
- Cooperative Parallelism Project (Co-Op), John May, PI

# Expanding Cylinder Uses Proxy Pattern to Load Balance Dynamically



# RMI References

---

- Kumfert, Leek, & Epperly. **Babel Remote Method Invocation.** *IPDPS '07*
- Kumfert & Leek. **How to Implement a Protocol for Babel RMI.** LLNL Tech Report UCRL-TR-220292. March 2006.
- Damevski, Zhang, & Parker. **Practical Parallel Remote Method Invocation for the Babel Compiler.** *CompFrame '07*

# Conclusions

---

- **Babel RMI is new and powerful tool**
  - ▶ Distributed programming model is consistent with serial OOP
  - ▶ Enables C, C++, F77/90/95, Python & Java to cooperate and even migrate objects between them.
- **Babel RMI is currently a niche tool**
  - ▶ We use it inside a single cluster
  - ▶ No facilities for security or authentication (unlike Grid)
- **Babel RMI is designed for collaborative research**
  - ▶ Parallel, SOAP, and IIOP implementations underway
  - ▶ SCI Institute @ Utah -----→
  - ▶ Seeking high-performance / switch-specific implementations... esp. Cray and BlueGene

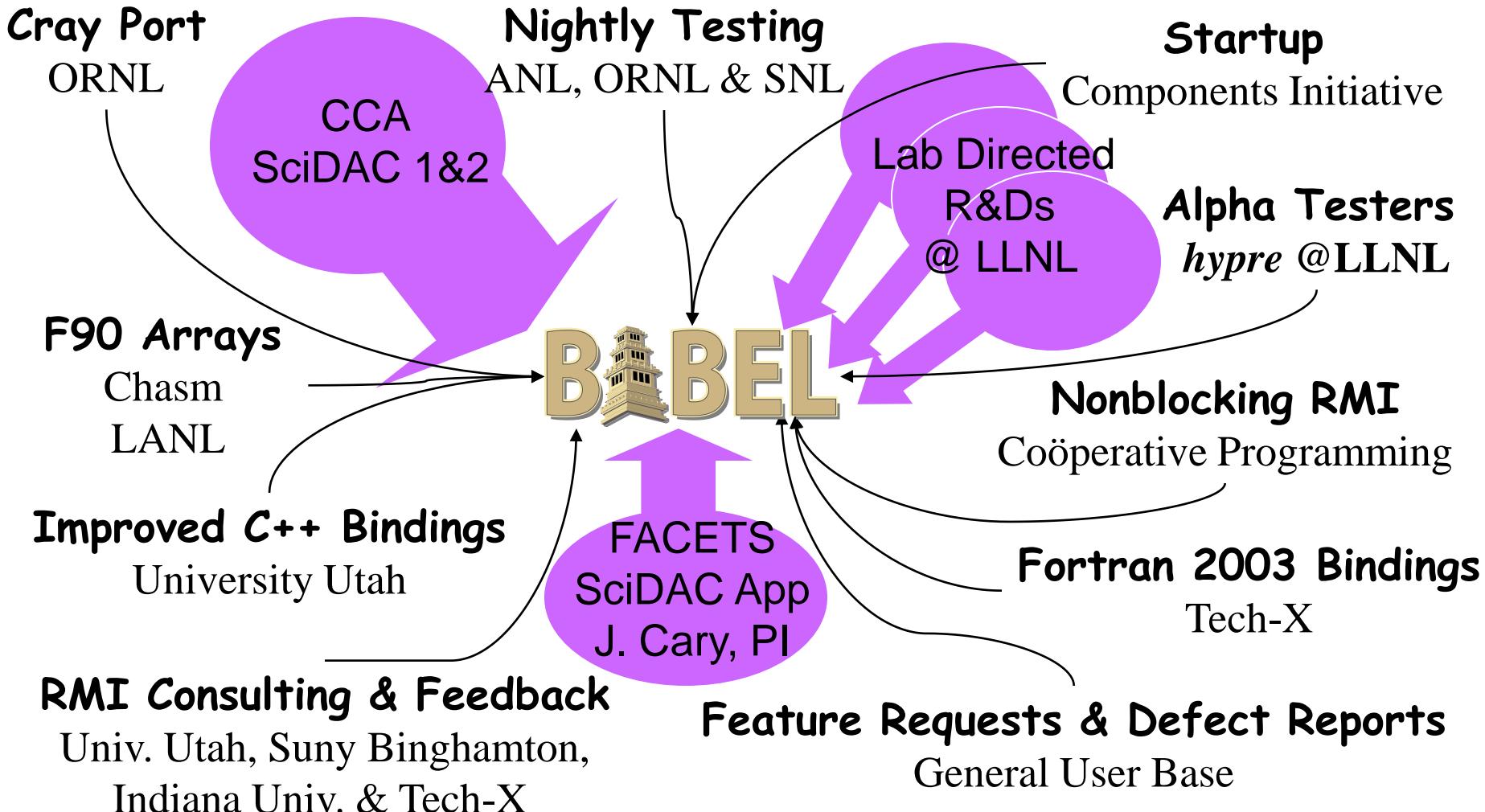


---

---

# VII. Closing

# Babel Thrives on Research Collaborations & Community



# Current R&D Activities in Babel

---

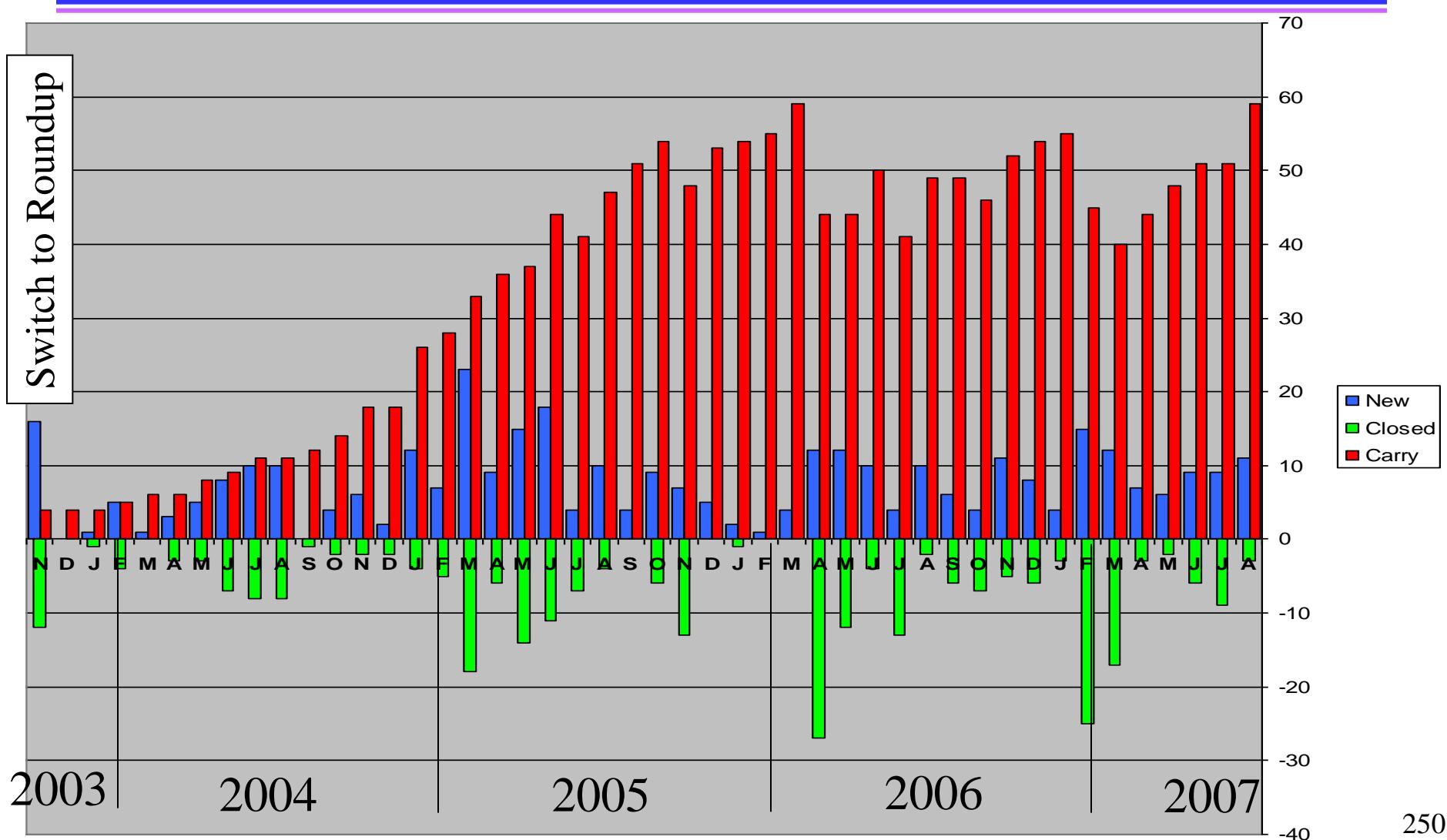
- **Semantics (Design by Contract) in SIDL**
  - ▶ Tamara Dahlgren's Ph.D. thesis
- **Parallel RMI**
  - ▶ Joint work with CCA, Cooperative Programming and Others.
- **CScADS (@ Rice)**
  - ▶ Applying compiler optimizations to Babel's inter-language context (exploratory)
- **Cray Inc.**
  - ▶ Math Libraries Group – Babel Generated Fortran Interfaces and CCA
  - ▶ Chapel Group – Chapel Binding to Babel (exploratory)
- **Tech-X**
  - ▶ Creating Fortran 2003 binding (SBIR)

# Open Source in Licensing, and in Practice

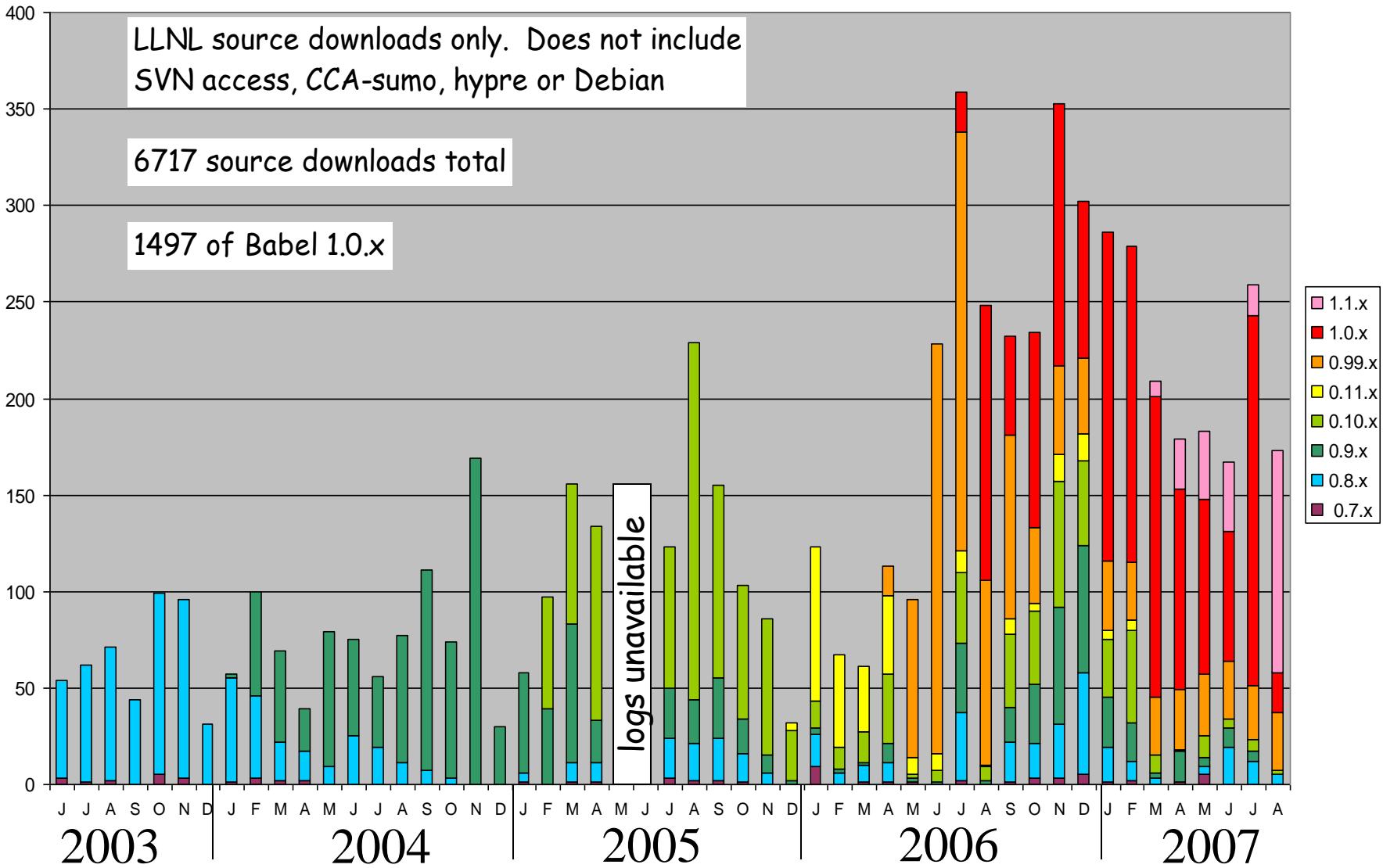
---

- Open Mailing Lists
  - ▶ **babel-announce, babel-users, babel-dev**
- Open Feature Request & Bug Tracking
- Online SVN repository
- Distributed Testing

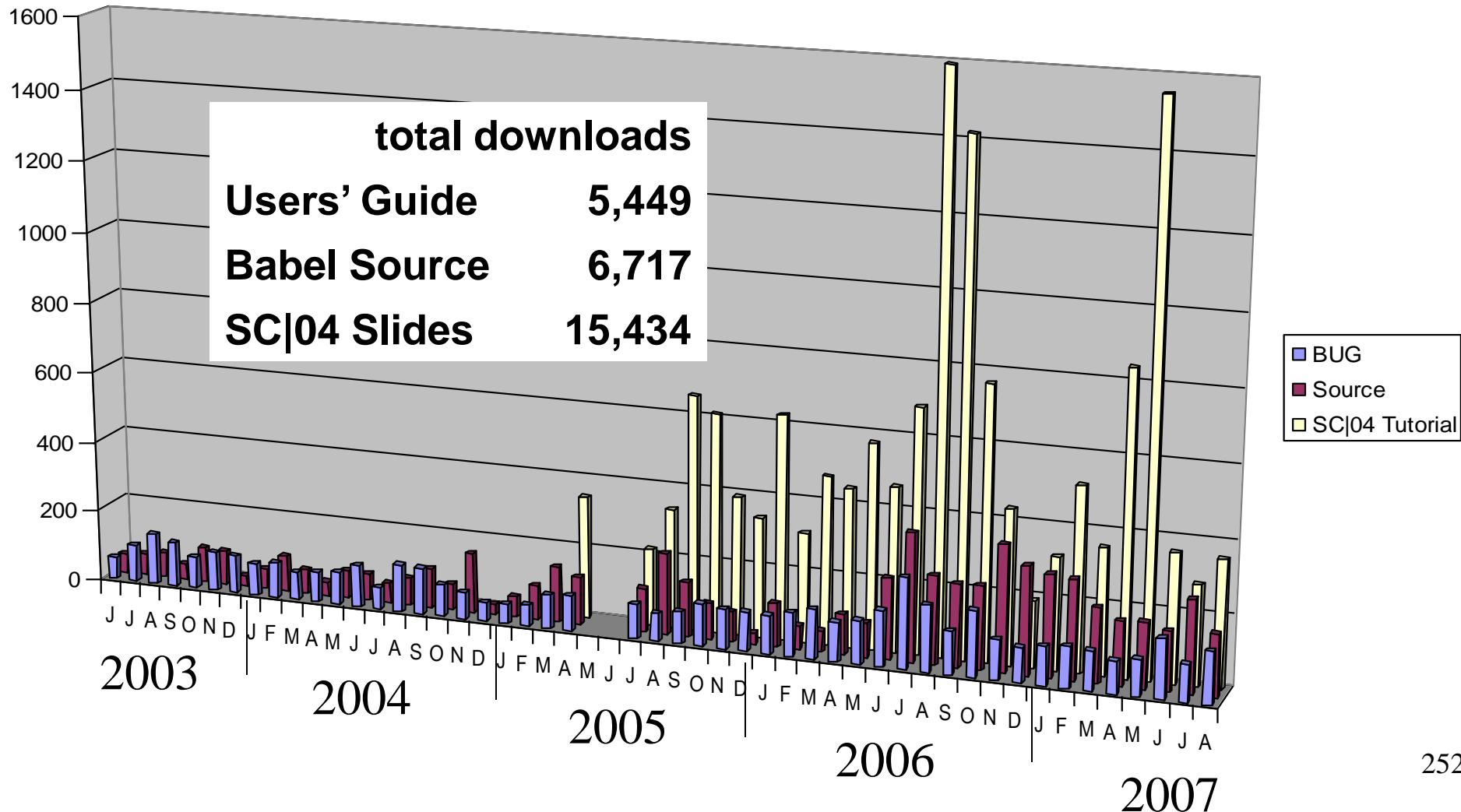
# All Feature Requests and Defect Reports are Public and Tracked



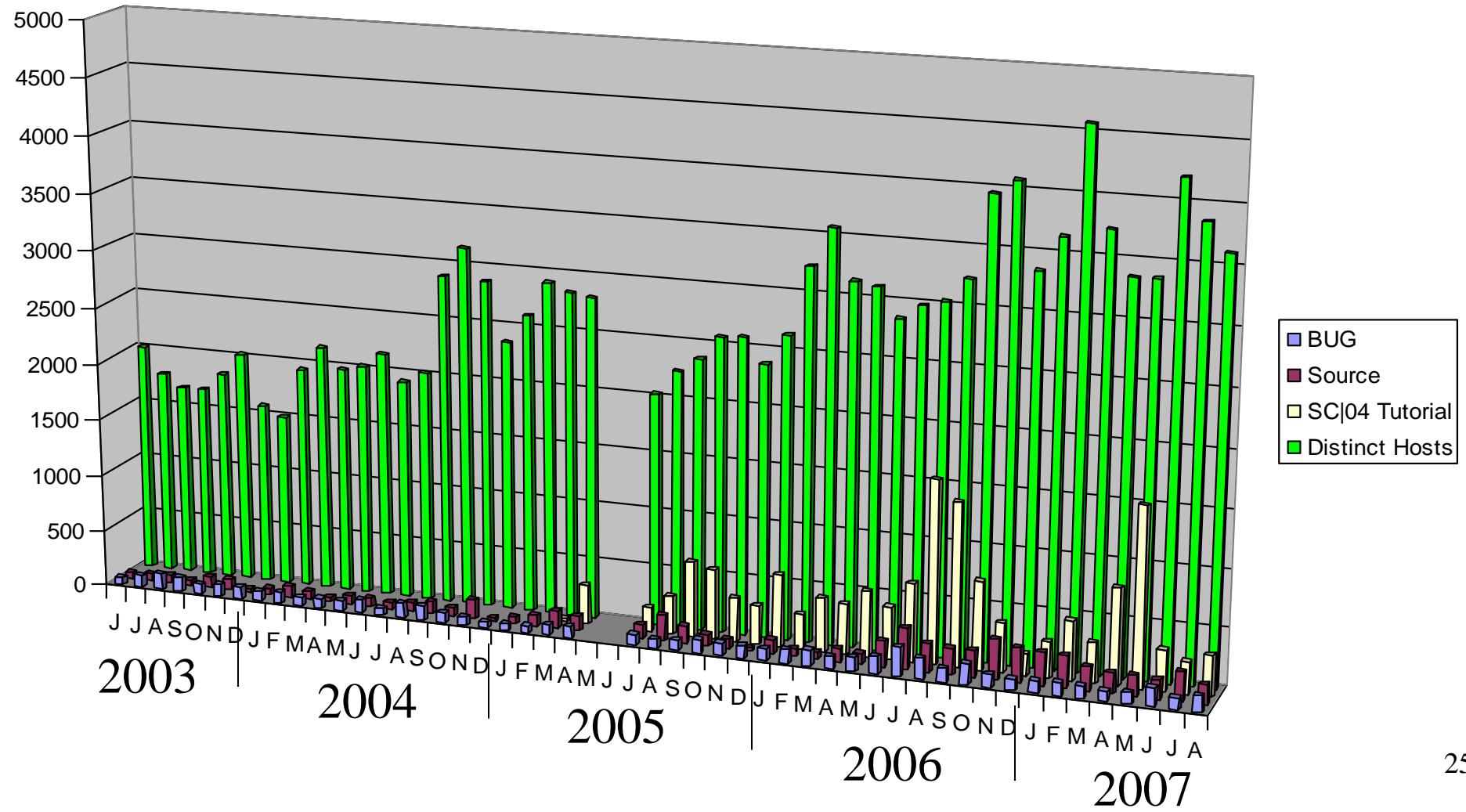
# Downloads Indicate a Sustained Growth Over Years



# Downloads of Our SC|04 Tutorial Were a Surprise



# Distinct Hosts Hitting Babel Website Overall Continues to Grow



# What Next?

## Customers Use Babel To Serve a Variety of Needs

- Manage community codes  
[Chemistry, Fusion, Radio Astronomy]
- Create software interface specifications.  
[CCA, ITAPS, TOPS]
- Integrate multiple 3<sup>rd</sup> party libraries into a single scientific application.  
[Chemistry, Fusion, CMEE]
- Develop libraries that connect to multiple languages.  
[hypre, TAU, Sparsekit-CCA]
- Scientific Distributed Computing  
[Co-Op, SCIJump, Legion-CCA, Harness, GA]

12

- Where does your project fit?
- Do you have different needs or special constraints?
- Talk to us! [components@llnl.gov](mailto:components@llnl.gov)

---

---

## Part II: Hands On

NOTE: Check online for latest changes and corrections to this document.

<http://www.llnl.gov/CASC/components/docs/scO7.html>

# Outline: Babel Acoustic

---

- **Background: Simple 2D Acoustic Wave**
- **Task 0: Setup and Run**
  - ▶ Appreciating the Babel Implementation
- **Task 1: Write a Program Using Babel Objects**
- **Task 2: Modify Existing Impl**
- **Task 3: Re-implement a piece in your language of choice**
- **Task 4: Add new method/feature**
- **Task 5: Create new object**

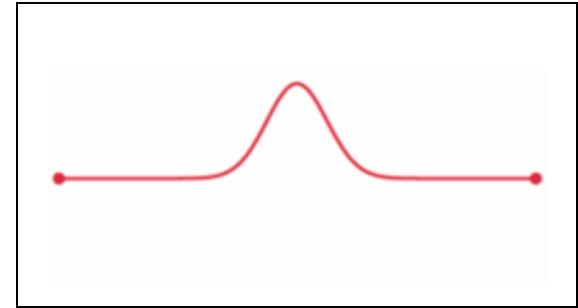
# Simple Wave Equation

---

$$u_{tt} = c^2 \nabla^2 u$$

- Prototypical hyperbolic PDE

- Animation is a 1D pulse through a string with fixed endpoints.



- We will compute 2D waves through isotropic inhomogenous elastic media

- ▶  $c_{\text{air}} \approx 344 \text{ m/s}$  (21 °C, no humidity)
- ▶  $c_{\text{water}} \approx 1497 \text{ m/s}$  (25 °C, no bubbles or sediment)
- ▶  $c_{\text{steel}} \approx 5100 \text{ m/s}$

# Before we start doing tasks, this is not a race!

---

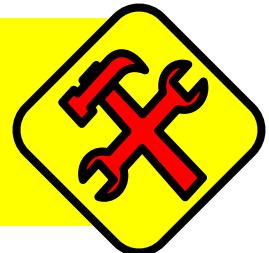
- Goal here is **learning**, not get to the end fastest
- Someone may not get past first three tasks, but still acquire more new information than a seasoned CS pro who races through
- We will go through tasks leisurely
- Happy to stop and discuss interesting developments from the class

# Task 0: Setup and Run

---

- Choose your setup.
- Install Babel (option B only)
- Configure/Build/Run Babel-wave
- Tour of your working Babel-wave installation
  - ▶ Preparation for next tasks.

People are circulating  
to help you



Group projects are  
encouraged

# LLNL ONLY

---

- **Babel is pre-installed on LC machines**
  - ▶ **chaos\_3\_x86\_elan3 (alc & pengra)**
  - ▶ **chaos\_3\_ia64\_elan4 (thunder)**
  - ▶ **chaos\_3\_x86\_64\_ib (atlas & zeus)**
- **Install dot-kits in your directory**

```
% gtar zxvf /usr/gapps/babel/babel.kits.tar.gz $HOME
```

```
% use babel-1.2.0-ic91
```
- **Copy tutorial code**

```
% cp /usr/gapps/babel/tutorial/babel-wave.tar.gz ...
```

# Choose your setup

---

- **Option A: ssh to our server**
  - 😊 Babel is preinstalled
  - 😢 You can't take it home with you
- **Option B: install Babel on your machine**
  - 😊 You can play with it after class
  - 😢 Installing Babel takes more time/expertise

# Option A: Setup

---

- Install ssh client, if needed.
- Get student account for our machine
- Get IP address of our machine
- ssh to our environment
- copy babel-wave.tar.gz to your home directory
- skip next two slides

# Option B: Prerequisites to Installing Babel

---

- **Mandatory**
  - ▶ Java <http://java.sun.com>
- **Required for Python-support**
  - ▶ Python
  - ▶ Numpy extension Module
- **Required for Fortran90-support**
  - ▶ Chasm
  - ▶ Fortran 90 compiler that Chasm supports
- **Required for visualization:**
  - ▶ Python
  - ▶ GNUploat

# Option B: Installing Babel

---

- Get tarball (& dependencies)
  - ▶ website: <http://www.llnl.gov/CASC/components>
  - ▶ CD ROM: (you can keep)
  - ▶ USB drive (please return to us!!!)
- Refer Back to Section IV: Babel tool
- Refer to the B.U.G.  
(Babel Users' Guide)
- Copy babel-1.1.2/contrib/babel-wave.tar.gz  
to your home directory

# Both Options A & B (&LLNL)

- `babel --help` to confirm Babel works
- gtar zxvf babel-wave.tar.gz
- cd babel-wave
- ./mini-configure `where babel-config`
- make
- cd runPython;
- python sanity\_check.py

```
...stuff ...
[[ 0.25207194  0.25207194]
 [ 0.25207194  0.25207194]]
[[ 1.19015414  1.19015414]
 [ 1.19015414  1.19015414]]
```

# The Babel-Wave Implementation

---

- Design
- SIDL File
- Directory Layout
- Implementations
- mini-configure
- Make
- Run

# Three Main Objects and their Roles

---

- **ScalarField – Manages a 2-D state**
  - ▶ Creates 2-D array based on rectangular coordinate space, and a mesh-spacing.
  - ▶ Can render shapes in coordinate space onto its 2-D array
- **WavePropagator – Applies physics**
  - ▶ Takes a Scalar Field as material wave speed
  - ▶ Takes a 2-D array for initial pressure distribution
  - ▶ Steps forward in time and creates new pressure distributions
- **Shape**
  - ▶ Drawing primitive for creating interesting inhomogenous material configurations.
  - ▶ Useful for demonstrating advanced OOP capabilities.

# Basic setup

- Specification is written entirely in SIDL interfaces
- Each language is implemented in a separate (descriptively named) package
- To demonstrate language interoperability, we'll mix-and-match from various implementations

```
package wave2d version 1.0 {  
    interface WavePropagator {...}  
    interface ScalarField {...}  
    interface Shape {...}  
}
```

```
package cxx version 1.0 {  
    class WavePropagator implements-all  
        wave2d.WavePropagator { ... }  
    class ScalarField implements-all  
        wave2d.ScalarField { ... }
```

```
package f90 version 1.0 {  
    class WavePropagator implements-all  
        wave2d.WavePropagator { ... }  
    class ScalarField implements-all  
        wave2d.ScalarField { ... }
```

# Directory Layout

---

- **babel-wave/**

- ▶ libC/ *C implementations*
- ▶ libCxx/ *C++ implementations*
- ▶ libF90/ *F90 implementations*
- ▶ runC/ *incomplete C driver*
- ▶ runCxx/ *C++ drivers*
- ▶ runPython/ *Python drivers*

# wave2d.sidl file details

```
package wave2d version 1.0 {
    interface Shape {
        Shape translate( in double delta_x, in double delta_y );
        Shape scale( in double delta_x, in double delta_y );
        Shape rotate( in double angle );
        Shape unify( in Shape other );
        Shape intersect( in Shape other );
        // . .
    }
    interface ScalarField {
        void render( in Shape shape, in double value );
        array<double, 2> getData();
        void getBounds( out double minX, out double minY,
                        out double maxX, out double maxY,
                        out double spacing );
    }
    interface WavePropagator {
        void step( in int n );
        array<double, 2> getPressure();
    }
}
```

# mini-configure

---

- This is a hand-made script that uses babel-config to build
  - ▶ `settings.make`
  - ▶ `wave2d.scl`
  - ▶ `runPython/babelenv.py`
- The makefiles also use babel-cc scripts

# make

---

- If you are option#2 and some of your languages were disabled at configure time:
  - ▶ make at the babel-wave/ directory may not be enough
  - ▶ cd into each of the relevant subdirectories and run make there too

# A bit more about drivers

---

- in runCxx/

- ▶ **runCxx2Cxx – statically linked, C++ only version of the sanity\_check**
- ▶ **runCxx2F90 – statically linked, C++ to F90 version of the sanity\_check**
- ▶ **runCxx2GNUpot – Generates \*.gif frames for a fixed problem**
  - Use ImageMagik to merge frames  
`% convert –adjoin loop 40 –delay 10 \  
pressure*.gif pressure_anim.gif`
  - copy animation from LC machine to local workstation and view in web browser

# **Task 1: Write a Program That Uses Babel Objects**

---

- Look in `runCxx/` or `runPython/` directories for examples
- There is a `runC/` directory... but its incomplete.
- We'll complete the C driver
- from `babel-wave/`
  - ▶ `cd runC/`
  - ▶ write the Makefile and driver code

# Task 1: Self Check

---

- The output of your new application  
`babel-wave/runC/runC2Cxx`  
should be identical to  
`cd babel-wave/runCxx/runCxx2F90`
- A word about environment variables:
  - ▶ It is critical to have paths set up for appropriate libraries to be found at runtime.
  - ▶ Original version of this tutorial required users use make to pass these settings to python (not elegant)
  - ▶ Now path info is generated by mini-config and imported directly into python via `babelenv.py`

# Losing track what object is implemented in what language?

- **That's the whole point of Babel!!!**
  - ▶ you shouldn't have to care
  - ▶ they're all just Babel objects.
- To change which implementation you use in statically linked C driver, simply change which one gets created and recompile.
- Want to try
  - ▶ a driver in a different language?
  - ▶ dynamic loading so you can change Impls without recompile?
  - ▶ Exception handling?

# Task 1 Recap: What did we learn?

---

- To use Babel objects:
  - ▶ Need the SIDL file
  - ▶ Run Babel to generate language bindings of choice
    - `babel --client=language [SIDL FILES...]`
    - Requires no knowledge of what language the objects were written in
    - (we made the package name indicate the implementation language for educational purposes)
  - ▶ Code the application to the wrappers.

# Task 2: Modify Existing Impl

---

- `cxx.internal.Unification`
  - ▶ has an intentional bug
  - ▶ should be `||` instead of `&&`.

# Task 2: How To's

---

- No need to modify SIDL files yet
- Just edit the `libCxx/cxx_internal_Unification_Impl.cxx` file
- Then
  - ▶ rebuild that library
  - ▶ rebuild in runCxx/ and run “task2”
  - ▶ or cd runPython/ and “python task2.py”

# Task 2: Self Check

---

- Why don't you have to rebuild the runPython/ directory?

# Task 3: Reimplement An Object in Language of Choice

- Not everything in Babel-wave is implemented in every language
- Regardless of what language you write it in, it should load into the python driver by only modifying the `from <pkg>.<class> import <class>`

# Task 3: Details

---

- Now, you'll have to edit (or write) a SIDL file... depending on language of interest
- Ask instructors for Makefile templates for each case

# Task 3: More Details

---

- For dynamic loading, you will also want to edit the `wave2d.scl` file
  - ▶ This is an XML catalog of what types are available in what libraries.
  - ▶ You added a new type, so either add it to the existing SCL file, or create a new one and add it to `SIDL_DLL_PATH`.
- For static linked executables, (e.g. the C driver we wrote) you may need to edit libraries in the final link line.

# Task 3: Self Check

---

- You implemented in a different language, but shouldn't have changed the behavior.

# Task 3 Recap: What did we learn?

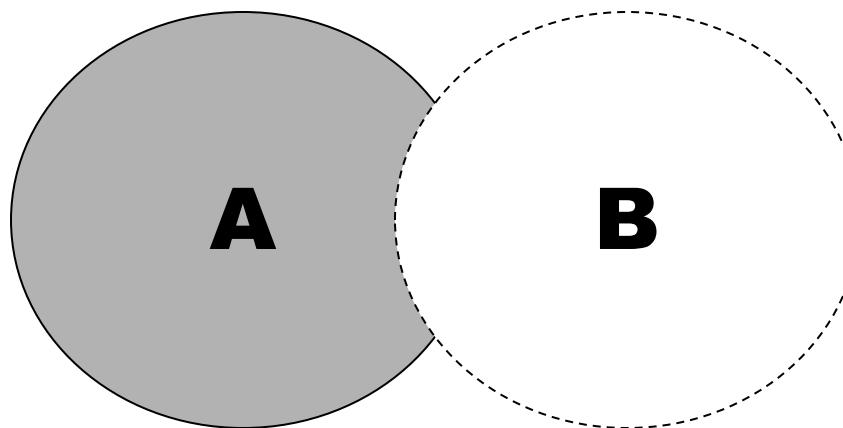
---

- Adding a new type is a lot more involved than modifying the behavior of an existing one.
- Adding one type didn't affect any of the others.
- Didn't affect the driver (when using dynamic loading).

# Task 4: Add a new feature

---

- Add a Subtract method to `wave2d.Shape`



- Since this interface is high in our hierarchy, follow the ramifications and implement what is needed.

# Recap of Entire Hands-on up to Now

---

- **Task 1:** Wrote a driver that used existing Babel Objects
- **Task 2:** Changed behavior of an existing Impl, but not its API.
- **Task 3:** Implemented a new object with behavior and API identical to an existing one.
- **Task 4:** Changed an interface that multiple objects implement.

# Task 5: Create a New Object

---

- **Add a multiline shape**

- ▶ Takes an array of x,y pairs
- ▶ Define `inLocus(x,y)` as follows:
  - Define a point outside the shape
  - Consider how many line segments are crossed between outside point and (x,y).
    - ★ odd implies `inLocus == true`
    - ★ even implies `inLocus == false`

# Please Fill Out Survey Forms

---

- Thank You
- Happy Babeling

---

---

Thank You