Interfacing Chapel with traditional HPC programming languages[1]

Adrian Prantl, Tom Epperly, Shams Imam, Vivek Sarkar

Center for Applied Scientific Computing (CASC)
Lawrence Livermore National Laboratory

October 17, 2011
Fifth Conference on Partitioned Global Address Space Programing Models (PGAS 2011)

[1] This work performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-PRES-505696

Interoperability with other programming languages. . .

- is **not optional**
- essential for the acceptance of a new language

Realistically, nobody will rewrite their entire multi-million line codebase in the language *du jour*.

Interoperability with other programming languages. . .

- is **not optional**
- essential for the acceptance of a new language

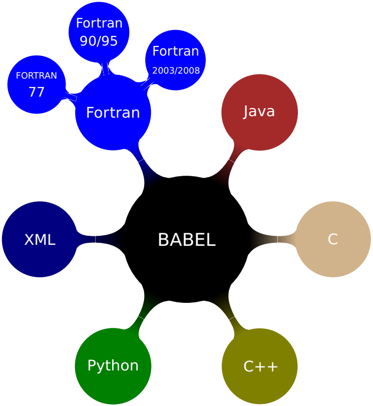
Realistically, nobody will rewrite their entire multi-million line codebase in the language *du jour*.

BRAID

a tool that provides interoperability for PGAS languages

➡Chapel first language to be supported

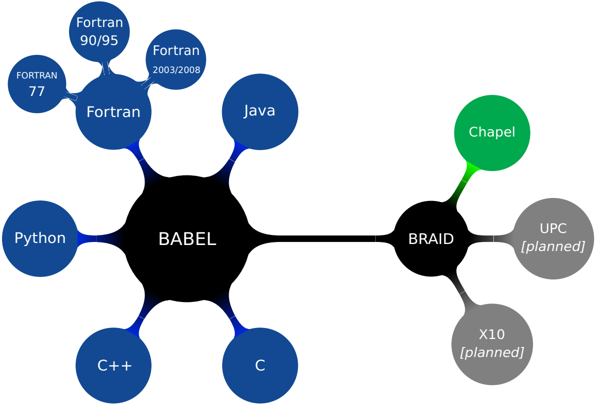
Related work



Babel

- LLNL's language interoperability toolkit for high-performance computing
- Designed for fast in-process communication
- Handles generation of all glue-code
- Features multi-dim. arrays, OOP, RMI, . . .

BRAID connects Babel with PGAS languages



Design goals

- be minimally invasive
  - minimal changes to the Chapel compiler
  - user shouldn't have to write *special* code
- play well with the Chapel runtime
  - expected behavior of programs remains unchanged
  - support distributed data types
- achieve maximum performance
  - avoid copying of arguments (when possible)
  - introduce minimal overhead

How does it work

Programming-language-neutral **interface specification**

Scientific Interface Definition Language (SIDL)

SIDL supporting

- fundamental data types
- object-oriented programming (user-defined types)
- interface inheritance
- exception handling
- dynamic multi-dimensional arrays

Using Chapel with BRAID — I

first, define the interface in SIDL

Example

```
import hplsupport;
package hpcc version 1.0 {
  class ParallelTranspose {
    // C[i,j] = A[j,i] + beta * C[i,j]
    static void ptransCompute(
    in hplsupport.Array2dDouble a,
    in hplsupport.Array2dDouble c,
    in double beta,
    in int i,
    in int j);
  }
}
```

- no data members are defined in the SIDL file
- all methods are public and virtual methods can be defined to be **final** or **static**

Using Chapel with BRAID — II

- next, use the Babel compiler to generate the server (callee) glue code:

  ```
  ~/cxxLib> babel --server=cxx hpcc.sidl
  ```

  - generates code for skeleton and Intermediate Object Representation (IOR)
  - generates empty blocks expecting user code
- user fills in empty blocks as implementation code
- user compiles code into shared libraries
  - Babel provides support for generating makefiles

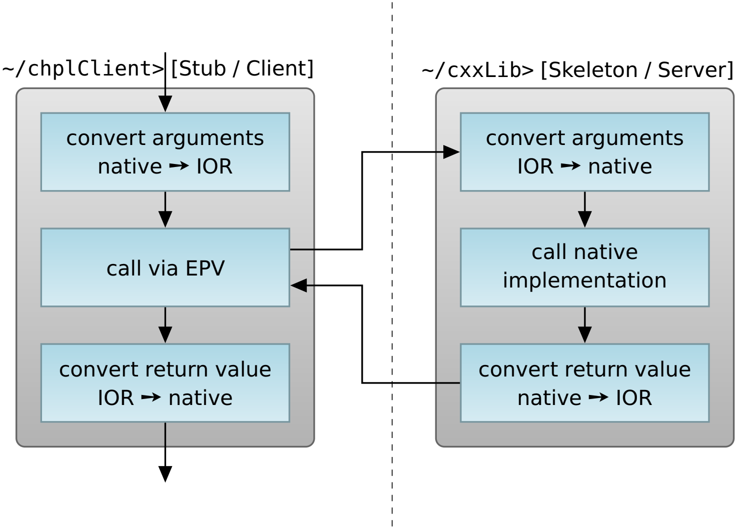
Using Chapel with BRAID — III

- next, use the BRAID compiler to generate the client (caller) glue code:

  ```
  ~/chplClient> braid --client=chapel hpcc.sidl
  ```

  - generates code for stub and IOR
  - user code uses the stub to make method calls
  - user code unaware of implementation
  - link to server code and SIDL runtime library during compilation and run the executable
- Babel/BRAID bindings take care of interoperability!

Control flow for crossing the language boundary


~/chplClient> [Stub / Client]
convert arguments native ➙ IOR
call via EPV
convert return value IOR ➙ native
~/cxxLib> [Skeleton / Server]
convert arguments IOR ➙ native
call native implementation
convert return value native ➙ IOR

IOR ....intermediate object representation

EPV ....entry point vector (vtable)

Chapel as client — challenges

convert Chapel data types to the IOR

add support for

- fundamental (primitive) types
- local arrays
- distributed arrays
- object-oriented programming
- exception handling

Local Arrays

SIDL arrays represent rectangular regions

normal SIDL arrays

- general interface for arrays
- can be used as parameters/return types
- row-major or column-major order
- support arbitrary strides

➥ access via interface

raw arrays (r-arrays)

- not as return type or *out* args
- must be contiguous in memory with column-major order

➥ presented as *native* array type

Local Arrays: Raw Array Example

Example

SIDL File (interface of external function)

```
class ArrayOps {
  static void matrixMultiply(in rarray<int,2> aArr(n,m),
  in rarray<int,2> bArr(m,o), inout rarray<int,2> res(n,o),
  in int n, in int m, in int o);
}
```

User writes Chapel code:

```
var sidl_ex: BaseException = nil;
var n = 3, m = 3, o = 2;
var a: [0.. #n, 0.. #m] int(32); // a 2D Chapel local array
var b: [0.. #m, 0.. #o] int(32);
var x: [0.. #n, 0.. #o] int(32);
// initialize the input matrices
[(i) in [0..8]] a[i / m, i % m] = i;
[(i) in [0..5]] b[i / o, i % o] = i;
// call the implementation of matrix multiply
ArrayOps_static.matrixMultiply(a, b, x, n, m, o, sidl_ex);
```

Local Arrays cont'd.

user can use *any* Chapel rectangular array as raw array

➥ includes support for distributed arrays!

Local Arrays cont'd.

user can use *any* Chapel rectangular array as raw array

➥ includes support for distributed arrays!

BRAID client code automatically

converts input arrays to required SIDL type

- copying involved when input arrays are
  1. not contiguous (e.g. distributed)
  2. not in column-major order for raw-arrays
- custom Chapel library extensions for column-major ordered arrays and **borrowed arrays** for extra speed

Distributed Arrays

Copying everything is too inefficient?

Distributed Arrays

Copying everything is too inefficient?

Custom type: `SIDL.DistributedArray`

- no contiguous or ordering requirements
- use Chapel runtime to access elements, server language (C, Java, etc.) unaware of communication
- minimal overhead, data transferred on access!

Object-oriented programming — I

SIDL supports packages, abstract classes, static and virtual methods

Chapel OOP support still in flux

- cannot inherit from classes with custom constructors

BRAID support for packages and static methods

- packages mapped to Chapel modules
- multiple Chapel classes can reside in a single module
- static methods mapped to additional Chapel modules

Object-oriented programming — II

Chapel classes allocate IOR via calls to SIDL runtime

- reference counting used to keep track of references to this newly allocated object
- Chapel class destructors decrement reference count to the IOR object

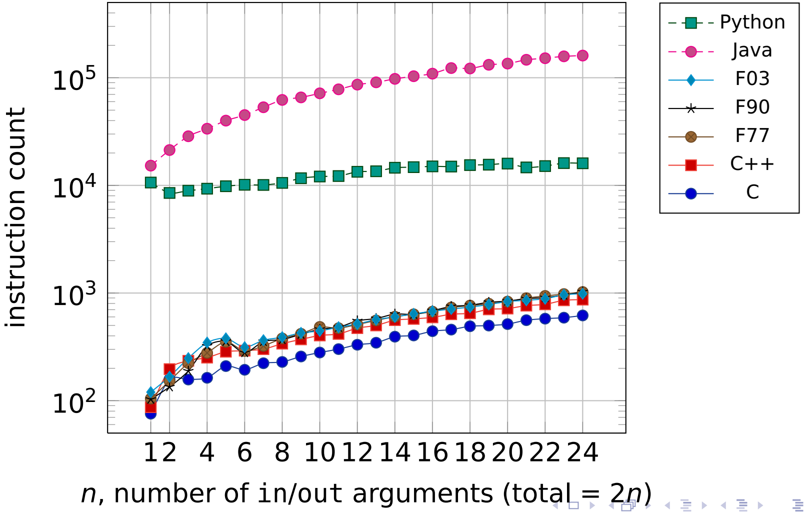
Chapel types delegate calls to IOR

- virtual function calls are handled by SIDL runtime
- type-casting supported by explicit cast calls

Benchmark

Calling a function that copies n arguments

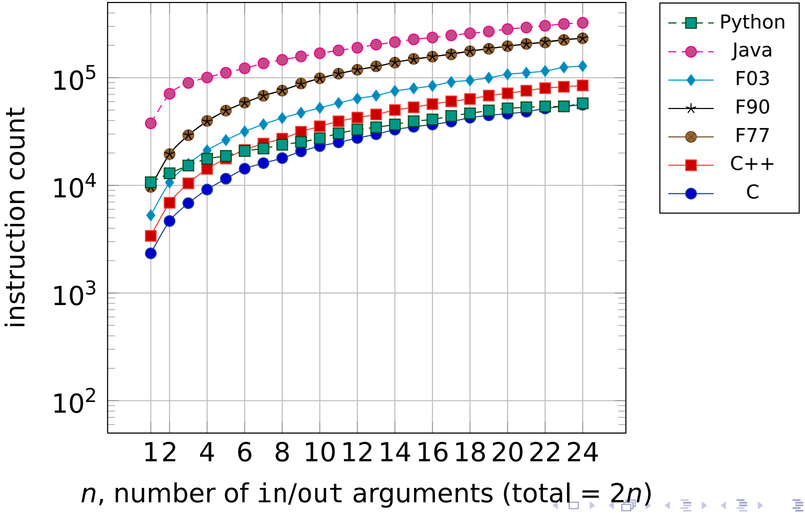
`copy bool`, $b_i = a_i$



Benchmark

Calling a function that copies *n* arguments

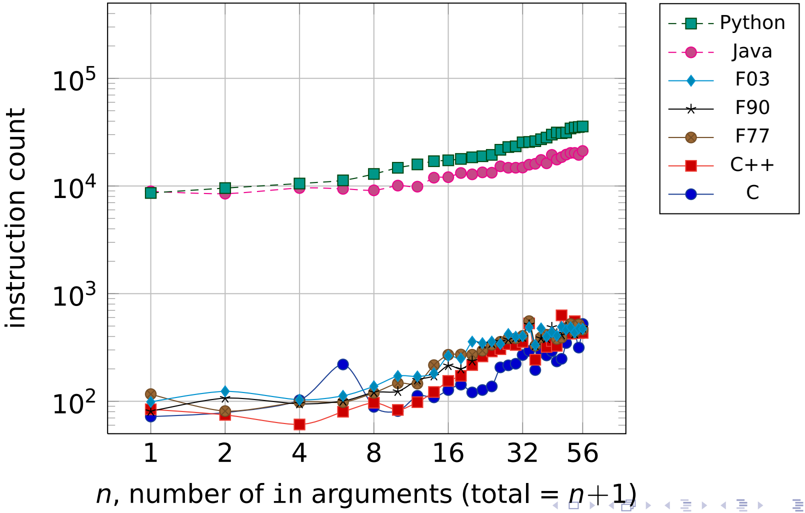
`copy string`, $b_i = a_i$



Benchmark

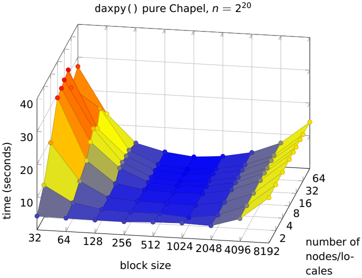
Calling a function that calculates the sum of *n* arguments

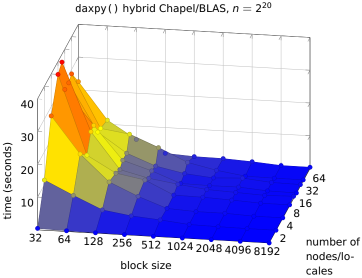
`sum float`, $r = \sum a_i$



Benchmark (distributed)

daxpy Benchmark





pure Chapel

hybrid Chapel/BLAS

Summary and Future Work

- Achieved interoperability between Chapel and
  1. C
  2. C++
  3. FORTRAN 77
  4. Fortran 90/95
  5. Fortran 2003/2008
  6. Java
  7. Python

➡including support distributed arrays

Future work

- add support for Chapel as server language
- use similar concepts to add support for UPC and X10

Thank you!

Thank you!

http://compose-hpc.sourceforge.net (BSD licensed)

Thank you!

http://compose-hpc.sourceforge.net (BSD licensed)

Are there any Questions?