Interfacing Chapel with traditional HPC programming languages

Shams Imam, Vivek Sarkar
Rice University

Adrian Prantl, Tom Epperly
LLNL
Introducing

- new programming language developed by Cray Inc. as part of DARPA High Productivity Computing Systems program
- provides a parallel programming model for use in HPC systems
- supports “global-view” abstractions allowing operations on distributed data to be expressed naturally
  - no explicit communications like MPI programs
Language Interoperability

• providing *new features* isn't enough to attract developers to adopt a new programming language

• should be easy to **integrate existing code** into new programs

• good support for interoperability lowers hurdle of accepting a new language
Babel – language interoperability tool

- LLNL's language interoperability toolkit for high-performance computing
- designed for fast, in-process communication
- handles generation of all glue-code
Babel – relevant features

• programming language-neutral interface specification language – Scientific Interface Definition Language (SIDL)

• SIDL supports
  – fundamental data types
  – object-oriented programming (user-defined types)
  – interface inheritance
  – exception handling
  – dynamic multi-dimensional arrays
Chapel: Language Interoperability

BRAID is the first PGAS language to be supported by Babel/BRAID.
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
Using Chapel with BRAID - I

• first, define the interface in SIDL

```chapel
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!
Babel/Braid – method invocation scheme

- example flow while calling from Chapel into C++

Chapel

Stub (Client)

- user chapel code
- convert arguments native $\leftrightarrow$ IOR
- call via EPV
- convert return value IOR $\leftrightarrow$ native

Skeleton (Server)

- convert arguments IOR $\leftrightarrow$ native
- call native implementation
- convert return value native $\leftrightarrow$ IOR

C++
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
## Supporting scalar data types

<table>
<thead>
<tr>
<th>SIDL Type</th>
<th>Size (in bits)</th>
<th>Corresponding Chapel Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1</td>
<td>bool</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>string (length=1)</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>int(32)</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>int(64)</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>real(32)</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>real(64)</td>
</tr>
<tr>
<td>fcomplex</td>
<td>64</td>
<td>complex(64)</td>
</tr>
<tr>
<td>dcomplex</td>
<td>128</td>
<td>complex(128)</td>
</tr>
<tr>
<td>opaque</td>
<td>64</td>
<td>int(64)</td>
</tr>
<tr>
<td>string</td>
<td>varies</td>
<td>string</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>enum</td>
</tr>
</tbody>
</table>
Local Arrays

- SIDL arrays represent rectangular regions
- two flavors of SIDL arrays
  - normal SIDL arrays
    - general interface for arrays
    - can be used as parameters/return types
    - row-major or column-major order
  - raw arrays (r-arrays)
    - can be used only as parameters
    - must be contiguous in memory with column-major order
Local Arrays contd.

• 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
    • not contiguous (e.g. distributed)
    • not in column-major order for raw-arrays
  – uses custom Chapel library extensions for column-major ordered arrays and borrowed-arrays to allow ease of using raw-arrays
Local Arrays: Raw Array Example

SIDL File:

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: SIDL Array Example

SIDL File:

```sidl
class ArrayOps {
    static bool reverseDouble(inout array<double,1> a);
}
```

User writes Chapel code:

```chapel
var sidl_ex: BaseException = nil;
// create a sidl array using SIDL runtime
var darray: sidl.Array(real(64), sidelined_double__array) = ...;
...
// call the implementation method
ArrayOps_static.reverseDouble(darray, sidl_ex)
```
Distributed Arrays

- one of the most challenging to support since Chapel allow user-defined data distributions

- Chapel runtime handles communication transparently, user uses these arrays just as local arrays

- BRAID requires users to distinguish between distributed arrays and SIDL arrays
  - BRAID provides library support for distributed arrays
Distributed Arrays: SIDL.DistributedArray

- copying/syncing of data is expensive
- SIDL arrays are not sufficient
  - meant for traditional languages like C, C++, ...
- create our 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, no copying!
Distributed Arrays: Example

SIDL File:

```chapel
class ParallelTranspose {
    static void ptransCompute(
in  hplsupport.Array2dDouble a,
    in  hplsupport.Array2dDouble c, in double beta,
    in int i, in int j);
}
```

User Chapel Code:

```chapel
... 
var A: [MatrixDom   ] real(64), // Chapel Distributed Array
    C: [TransposeDom] real(64);
forall (i,j) in TransposeDom do { // parallel loop
    var aWrapper = new hplsupport.BlockCyclicDistArray2dDouble();
    aWrapper.initData(GET_CHPL_REF(A));
    var cWrapper = new hplsupport.BlockCyclicDistArray2dDouble();
    cWrapper.initData(GET_CHPL_REF(C));
    // C[i,j] = beta * C[i,j] + A[j,i];
    ParallelTranspose_static.ptransCompute( 
        aWrapper, cWrapper, beta, i, j, sidl_ex);
}
```
Object-oriented programming

- SIDL supports packages, abstract classes, static and virtual methods
- Chapel doesn't yet fully support OOP, minimal support for classes
  - cannot inherit from classes with custom constructors
- 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 data structure which maintains virtual function table

- inheritance simulated via the IOR object, SIDL runtime manage the IOR representation
  - type-casting supported by explicit cast calls
Object-oriented programming: Example

**SIDL File:**
```plaintext
interface A { string a(); };  
interface B { int b(); };  
class C { string c(); };  
class D extends implements-all A, B { string d(); };  
```

**User Chapel Code:**
```plaintext
// var a: A = new A(); disallowed as A is an interface

var d: D = new D(sidl_ex);
var v1 = d.a(sidl_ex);
var v2 = d.c(sidl_ex);

var a: A = d.asA(); // Explicitly cast d as an instance of A
var v3 = a.a(sidl_ex);
assertEquals(v1, v3);

var c: C = d.asC(); // Explicitly cast d as an instance of C
var v4 = c.c(sidl_ex);
assertEquals(v2, v4);
```
Exception Handling

- Chapel supports \texttt{inout} arguments
- SIDL exposed functions require an exception object as argument
- BRAID generated code fills in exception object to notify calling code of exceptions
Exception Handling: Example

• User Chapel code for handling exceptions

```chapel
var sidl_ex: BaseException = nil;
// create a sidl array using SIDL runtime
var darray: sidl.Array(real(64), sidl_double__array) = ...;
...
// call the implementation method
ArrayOps_static.reverseDouble(darray, sidl_ex)

if (sidl_ex != nil) {
    // exception occurred while invoking reverseDouble()
    // user handles exception how she wishes
    halt(sidl_ex.getMessage());
}
```
Performance results - I

copy bool, \( b_i = a_i \)

\[ \begin{array}{|c|c|c|c|c|}
\hline
n, number of in/out arguments (total = 2n) & C & C++ & F77 & F90 & F03 & Java & Python \\
\hline
1 & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} \\
2 & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} \\
4 & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} \\
8 & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} \\
16 & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} \\
32 & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} & \text{instruction count} \\
\hline
\end{array} \]
Performance results - II

\[ \text{sum float, } r = \sum a_i \]

![Graph showing instruction count vs. number of arguments (total = n+1)](image)

- **C**
- **C++**
- **F77**
- **F90**
- **F03**
- **Java**
- **Python**
### Performance results - III

The ptrans Benchmark, hybrid and pure Chapel versions execution times (in seconds) compared, input matrix is of size 2048 × 2048 with a block size of 128 DistributedArray interface in SIDL, reusing our own infrastructure to make it completely portable.

<table>
<thead>
<tr>
<th>Nodes/locales</th>
<th>Pure execution time</th>
<th>Hybrid execution time</th>
<th>Overhead (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>898.26</td>
<td>893.08</td>
<td>-0.58</td>
</tr>
<tr>
<td>6</td>
<td>520.51</td>
<td>540.88</td>
<td>3.91</td>
</tr>
<tr>
<td>8</td>
<td>443.74</td>
<td>457.59</td>
<td>3.12</td>
</tr>
<tr>
<td>12</td>
<td>343.90</td>
<td>339.42</td>
<td>-1.30</td>
</tr>
<tr>
<td>16</td>
<td>221.93</td>
<td>226.60</td>
<td>2.11</td>
</tr>
<tr>
<td>24</td>
<td>163.17</td>
<td>169.04</td>
<td>3.60</td>
</tr>
<tr>
<td>32</td>
<td>112.11</td>
<td>114.30</td>
<td>1.95</td>
</tr>
<tr>
<td>48</td>
<td>112.55</td>
<td>114.77</td>
<td>1.97</td>
</tr>
<tr>
<td>64</td>
<td>59.45</td>
<td>60.59</td>
<td>1.91</td>
</tr>
</tbody>
</table>
Performance results - IV

Comparing pure and hybrid performance of daxpy() functionality

array sizes are $2^{20}$, programs ran on 64 nodes

pure: Chapel implementation of $C = a \times X + Y$ where $X$ and $Y$ are distributed arrays
hybrid: same example implemented by calling the blas daxpy() function using SIDL.DistributedArray
Summary and Future Work

- achieved interoperability between Chapel and traditional HPC languages
  - support all basic data types
  - support distributed arrays

- future work:
  - add support for Chapel as server language
  - use similar concepts to add support for UPC and X10
Questions