Interfacing Chapel with traditional HPC programming languages [*]

Adrian Prantl, Thomas Epperly
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
adrian@llnl.gov, epperly2@llnl.gov

Shams Imam, Vivek Sarkar
Rice University
shams@rice.edu, vsarkar@rice.edu

Abstract

Chapel is a high-level parallel programming language that implements a partitioned global address space model (PGAS). Programs written in this programming model have traditionally been self-contained entities written entirely in one language. While this approach enables the compiler to produce better performing code by doing whole program optimization, it also carries a risk of positioning PGAS languages as "island" programming languages.

In this paper we present a tool that lets Chapel programs call functions and instantiate objects written in C, C++, Fortran 77–2008, Java and Python. Our tool creates language bindings that are binary-compatible with those generated by the Babel language interoperability tool. The scientific community maintains a large amount of code (mathematical libraries, solvers and numerical models) written in legacy languages. With the help of our tool, users will gain access to their existing codebase with minimal effort and through a well-defined interface.

Knowing the demands of the target audience, we support the full Babel array API. A particular contribution of this paper is that we expose Chapel's distributed data types through our interface and make them accessible to external functions implemented in traditional serial programming languages. We anticipate applying similar concepts to other PGAS languages in the future.

Categories and Subject Descriptors D [*12*]: 2; D [*3*]: 4

General Terms Chapel, language interoperability, PGAS, code generator, compiler

Keywords Chapel, language interoperability, PGAS, compiler, SIDL, Babel, BRAID, C, C++, Fortran, Java, Python

1. Introduction

Getting developers to adopt a new programming language is a delicate process. Not only is it necessary to provide *convincing new features* that make adopting a new programming model worthwhile; it is also instrumental to make it easy to *integrate existing code* into new programs. Having good support for interoperability significantly lowers the hurdle of having to switch, by making it stepwise progression instead of an all-or-nothing move.

The Partitioned Global Address Space (PGAS) programming model is tailored towards high performance computing systems. It combines the performance and data locality (partitioning) features of distributed memory with the "programmability" and data referencing simplicity of a shared-memory (global address space) model. In PGAS languages, there are multiple execution contexts (usually one per core[1]) with separate address spaces, and performance is gained by exploiting data locality. Transparent access to memory locations on other execution contexts is usually supported by one-sided communication libraries. Languages implementing the PGAS approach (such as UPC [5], Chapel [7] and X10 [10]) thus offer a simple/familiar *global-view* programming model that helps to reduce development time. The scientific community maintains a large amount of code written in legacy languages such a C, Fortran, etc. With the help of our tool, users will be able to gain access to their existing codebase with minimal effort in these languages. Consequently, basic interoperability with the host languages of these applications is one of the keys to widespread acceptance of PGAS languages.

In this paper we present a tool that adds interoperability with C, C++, Fortran,[2] Python and Java to the Chapel programming language. Our Chapel language interoperability is the first language binding to use a new *term-based* approach to language interoperability called BRAID [4]. BRAID, which is described in detail in Section 3, is aimed at producing language interoperability code from a wide variety of interface descriptions.

1.1 Related work

This work builds on previous work done at Lawrence Livermore National Laboratory by generating language bindings that are binary-compatible to those generated by the Babel language interoperability tool [15]. Babel is aimed primarily at the high-performance computing (HPC) community and thus addresses the need for mixing programming languages in order to leverage the specific benefits of those languages. Babel allows all of the supported languages to operate in a single address space. Babel is essentially a compiler that generates glue code from a specification in a scientific interface definition language (SIDL) [2]. Prior to this work, there was no sup-

[*] This work performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was supported in full by the DOE Office of Science, Advanced Scientific Computing Research, program managers Sonia Sachs and Lucy Nowell. LLNL-CONF-491213

[1] In Chapel, it is one per *node*.

[2] In many respects the various incarnations of Fortran are so different that we actually treat FORTRAN 77, Fortran 90/95 and Fortran 2003/2008 as separate languages.

PGAS 2011 October 15–18, 2011, Galveston Island, Texas, USA

port in Babel to support new PGAS languages, like Chapel and X10, as clients that can interoperate with the traditional HPC languages.

Currently most PGAS languages already offer some form of *one-to-one* interoperability with their respective host languages, such as C for UPC [5] and Fortran for Co-Array Fortran [21]. X10 has support for calling into C++ or Java, using the *native* interface [24]. The goal of this work is, however, to connect a PGAS language to *all* the languages that are currently used in the HPC community.

1.2 The Challenges

Chapel is a modern high-level parallel programming language originally developed by Cray Inc. as part of the DARPA HPCS program [7]. In contrast to traditional programming languages such as C++, the runtime system of the language takes care of executing the code in parallel. The language still offers the user fine-grained control over things such as the data layout and the allocation of distributed data, but it renders the tedious explicit encoding of communication through, *e. g.*, a message-passing interface, obsolete.

Interestingly, the main selling points of the language also contribute some of the main challenges for achieving interoperability with other (PGAS and non-PGAS) programming languages:

Distributed data types. What may seem like a simple variable in a Chapel source code might actually be a reference to a value that is stored within an entirely different process running potentially on a different node. It might even be an array whose elements are spread out across multiple nodes.

Parallel execution model. In Chapel, parallelism is integrated into the language (*e. g.*, through `forall` statements) and the decision when to off-load work into a separate thread is mostly performed by the Chapel runtime.

Lack of support for calling into Chapel. Although the Chapel language has a fully-fledged module system and also some support for calling functions implemented in C, there are no provisions for compiling Chapel code into an externally callable library yet.[3] We are currently working with the Chapel developers to standardize name mangling and other features required for bi-directional interoperability.

In addition to the aforementioned points, our interoperability approach must deal with the usual challenges of doing cross-language development, such as the impedance mismatch of function arguments caused by different representations of scalar data types and storage formats for compound data types.

1.3 Design goals

Our design goals for the Chapel binding were guided by the needs of the high-performance computing community. The top priority here is maximum performance. For users from the HPC community, it is essential to keep the cost of interoperability low; otherwise, our solution will not be used. For instance, this implies that serialization of arguments for external calls is ruled out – when possible – we should even avoid any copying of function arguments.

We also need to interact correctly with the Chapel runtime. Language interoperability must not break the expected behavior of a Chapel program. Overall, our approach should be minimally invasive. To maximize the acceptance of our solution it is important that we do not require huge patches to be made to the Chapel compiler and runtime. As we will explain in Section 4.2.1, we found it necessary to extend the Chapel runtime to support borrowed arrays to reduce the overhead associated with passing array arguments back and forth to external functions.

[3] The Chapel team is currently working on supporting an `export` keyword to support callbacks from C into Chapel.

2. The Babel interoperability architecture

Babel is a widely-used tool for high-performance language interoperability developed at Lawrence Livermore National Laboratory [13, 15]. Our new tool builds on this work by generating Babel-compatible language bindings for Chapel.

Babel is instrumental in making large amounts of valuable legacy code accessible to the developers of newer generations of scientific codes. To this end, existing code is packaged into components with a well defined interface encapsulating the legacy code. Interfaces are described in a scientific interface definition language (SIDL), which specifies objects, methods and interfaces provided by the components. The interface specification may also include contracts for methods and types: pre-conditions, post-conditions, and data type invariants. The use of Babel is not limited to legacy codes: another typical use-case would be the combination of multiple mathematical models, implemented in different languages, to form a larger multi-model simulation.

The component-based design of Babel is inspired by technologies such as CORBA [22], COM [25], or more recently XPCOM [26]. However, Babel's main focus is the scientific computing community, which manifests in Babel's efficient support for array data types and interoperability with C and Fortran. Babel also supports remote procedure calls and object serialization [18]; features that are present in increasingly popular technologies such as Google Protocol Buffers [17] and Apache Thrift [1].

SIDL provides a language-independent object-oriented programming model and type system. This allows components to share complicated data structures such as multidimensional arrays, interfaces, and structures across various languages. Babel also provides consistent exception handling semantics across all supported languages. Babel generates the necessary glue code that maps these high-level interfaces to a particular language ecosystem. As such, it can be used stand-alone or as part of the full Common Component Architecture [2], which provides additional capabilities such as dynamic composition of applications.

Out of the box, Babel supports the languages C, C++, Java, FORTRAN 77, Fortran 90/95, Fortran 2003/2008 and Python. Babel acts as a compiler that takes SIDL files as input and generates glue code in the respective languages as output. Babel enables any supported language to call any other supported language. In contrast, SWIG [3], another popular language interoperability tool, provides glue code to make C or C++ callable from a variety of supported programming languages. SWIG uses an augmented C or C++ header file as its interface definition.

2.1 Anatomy of a Babel call

A foreign call with Babel starts with the client (known in Babel as the caller) invoking a *stub*, which accepts all the arguments of the method in the native data format (such as value representation or memory layout) of the client language. The stub is automatically generated by Babel and is usually output in the client's language, or in C, if the native language is not expressive enough to perform the argument conversion, which often involves byte-level memory manipulation. Stubs are very small and the compiler can sometimes inline them. The stub converts the arguments into the intermediate object representation (IOR) which is Babel's native data format. It then calls the server implementation, which also has to accept the IOR. On the server side resides a *skeleton*, which does the reverse operation of converting the IOR into the native representation of the server (known in Babel as the callee). The skeleton invokes the *implementation* which is the actual method to be called. Upon return, it converts all outgoing arguments and the return value into the IOR and returns to the Stub, which performs the translation back to the client's data format.

2.2 Intermediate Object Representation (IOR)

The IOR is the native format used to define data types specified in SIDL. Babel uses C to implement the IOR using corresponding equivalent types when possible, *e. g.*, `int32_t` for SIDL `int`s which are defined to be 32 bits in length. Complex numbers are mapped to a pair of numbers of the appropriate type using C `struct`s. Support for arrays is provided using a `struct` to store the array's metadata (such as rank, lower/upper bounds, array ordering, etc.) and a pointer to the block of memory representing the elements of the array. The array API then provides methods to access or mutate the elements of the array and access the various metadata.

Most of the heavy lifting necessary to support classes and interfaces is also done in the IOR. The IOR representation of an object contains a virtual function table (called the Entry-Point Vector or EPV in Babel) which is used to resolve method invocations at runtime. The IOR also contains pointers to the object's base class and to all the implemented interfaces. Moreover, there are EPV-like entries used by Babel's instrumentation mechanism (dubbed *hooks*) and for contract enforcement [15]. Since the IOR is implemented in C, Babel requires that all languages it supports to be able to call C functions and be callable from C.[4]

Babel's architecture provides high-performance, bi-directional language interoperability between multiple languages in a single executable. It reduces the $O(n^2)$ binary mappings between pairs of programming languages to $O(n)$ mappings between C and each of the n supported languages. This approach completely hides the language of implementation from the client code, and server-side implementations do not need to know what languages will be calling them. By going through the IOR, it is possible to switch between different server implementations (in different languages) without having to recompile the client. It also enabled us to implement support for Chapel without making modifications to Babel.

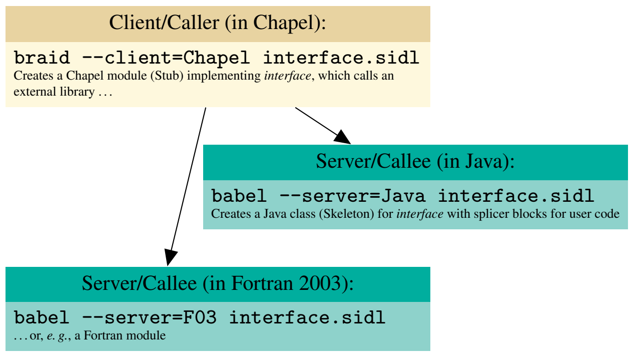
3. BRAID: the next-generation interoperability tool

We implemented all the glue code generation using the **B**RAID system for **r**ewriting **a**bstract **i**ntermediate **d**escriptions; thus creating a new tool that has a command line interface similar to that of Babel. The new tool is implemented in Python, making it very portable itself. BRAID is a multi-faceted, term-based system for generating language interoperability glue code designed and developed as part of the COMPOSE-HPC project [4] to be a reusable component of software composability tools.

From a user's perspective, BRAID is the tool that generates glue code for parallel PGAS languages, while Babel handles traditional HPC languages. Eventually, we intend to make this distinction invisible to the end user by launching both through the same front end. Figure 1 shows an example invocation for a program written in Chapel that wants to make calls to an interface that is implemented in Fortran or Java. Our Chapel language interoperability tool is the first of several applications envisioned for BRAID including optimized language bindings for a reduced set of languages and building multi-language interfaces to code without SIDL interface descriptions.

The most important difference between BRAID and Babel is, however, not in the choice of Python as the implementation language; it is how the language backends are designed: In Babel each code generator is a fixed-function Java class that builds all the glue code out of strings. BRAID, on the other hand, creates glue code in a high-level *language-independent* intermediate representation

[4] Fortran 90/95 breaks this requirement. Babel uses `libChasm` [23] which actually reverse engineered a large number Fortran compilers to achieve the interoperability.



```
braid --client=Chapel interface.sidl
```
Creates a Chapel module (Stub) implementing *interface*, which calls an external library …

```
babel --server=Java interface.sidl
```
Creates a Java class (Skeleton) for *interface* with splicer blocks for user code

```
babel --server=F03 interface.sidl
```
… or, *e. g.*, a Fortran module

Figure 1. Usage of BRAID and Babel to generate interface code

(IR). This intermediate representation is then passed to a code generator which translates it into actual high-level code. At the moment there are code generators for C and Chapel, and also initial versions for Fortran, Java and Python. This architecture offers a higher flexibility than the static approach of Babel: For example, (object-)method calls in Babel need to be resolved by looking up the address of the method in a virtual function pointer table. Since Chapel has no means of dealing with function pointers (it implements its own object system instead), BRAID's Chapel code generator will generate a piece of C code to do the virtual function call *on the fly*, and place a static call to this helper function in lieu of the virtual function call. Using this system we can reduce the number of times the language barrier is crossed to the minimum, leading to more code generated in the higher-level language, which again enables the compiler to do a better job at optimizing the program.

Similar to Babel, BRAID can also be instructed to generate a *Makefile* that is used to compile both program and glue code and link them with any server libraries. The Chapel compiler works by first translating the complete program into C and then invoking the system C compiler to create an executable binary. The Makefile created by BRAID intercepts this process after the C files have been generated and builds a *libtool* [16] library instead. Libtool libraries contain both regular (`.o`) and position-independent (`.so`) versions of all the object files, which can be used for static and dynamic linking, respectively.

The Chapel language already has basic support for interfacing with C code via the `extern` keyword [12]. BRAID uses this interface as an entry point to open up the language for all the other languages supported by Babel.

4. The interface

In this section we describe how the Babel IOR is mapped onto the Chapel data types and what code BRAID generates to translate between the two representations. The generated glue code must translate each method argument in both directions, *in* (going from client to server) and *out* (server to back to client). An argument's intent (*in*, *out*, *inout*) is a mandatory part of the SIDL interface specification for each method.

4.1 Scalar datatypes

Table 1 lists the scalar types supported by SIDL and the corresponding Chapel types used by the skeleton or stub while converting Chapel code from or into the IOR. The SIDL scalar types are (with the exception of strings) of fixed length and were easy to support especially since Chapel has parametric support for the number of bits in the integral and floating point types which map to the same representation as used by the IOR. It also has native types for both single-precision and double-precision complex numbers and sup-

SIDL type	Size (in bits)	Corresponding Chapel type
bool	1	bool
char	8	string (length=1)
int	32	int(32)
long	64	int(64)
float	32	real(32)
double	64	real(64)
fcomplex	64	complex(64)
dcomplex	128	complex(128)
opaque	64	int(64)
string	varies	string
enum	32	enum

Table 1. Scalar Data Types in SIDL and their Chapel equivalents on a 64-bit machine

```
void Args_Basic_testChar_stub(
    struct Args_Basic__object* self,
    /* inout */ const char** c,
    struct sidl_BaseInterface__object** ex) {
  // In Chapel, a char is a string of length 1
  char _babel_c;
  _babel_c = (int)*c[0];
  (*self->d_epv->f_testChar)(self, &_babel_c, ex);
  // Sync back using lookup table
  *c = (const char*)
    &chpl_char_lut[2*(unsigned char)_babel_c];
}
```

Figure 2. Stub code generated by BRAID for Chapel `complex` and `char` types

ports *opaque* types that allow data to be passed around through Babel/BRAID back into the original address space. Chapel also supports enumerated type to defines a set of named constants. On the other hand, the Babel IOR and the Chapel compiler use different representations for complex numbers, hence BRAID generates glue code to pass around copies. Since Chapel does not have a `char` type, BRAID needs to generate code to convert Chapel unit-length strings into chars using a statically allocated lookup table. An example for passing an *inout* argument from Chapel to an external function is shown in Figure 2. Supporting the string type itself was straightforward because the Chapel representation of strings is similar to the Babel IOR representation (both use a character array).

4.2 Array implementation

There are two major variations of arrays in Chapel with regards to how the data is distributed. Chapel arrays can be entirely local, *i. e.*, all the array data are allocated at the same logical locale,[5] or they can be distributed over multiple places. BRAID provides support for both local and distributed arrays.

4.2.1 Local Arrays

We will first deal with Chapel's local arrays. One of the key features of SIDL is the support for multi-dimensional arrays. SIDL arrays come in two flavors: normal and raw [13, Chapter 6.4]. Normal SIDL arrays provide all the features of a normal SIDL type, while raw SIDL arrays, called r-arrays, exist to provide a more native, lower level way to access numeric arrays. SIDL also defines an array API which the client code uses to prepare the argument passed to a SIDL method. The implementation code uses the API to retrieve data and metadata of the incoming array argument.

[5] *Locale*s are Chapel's abstraction for an entity with data processing and storage capabilities.

Chapel supports generic arrays using the concept of domains [9, 11]. Domains are used to specify the index set including the number and size of the dimensions. In Chapel, an array maps indices from a domain to variables of a homogeneous type. This allows Chapel to easily implement associative arrays, as the indices of a domain may be of any type. In this work, we concentrate on arrays defined by unstrided rectangular domains. These are domains where each dimension is defined by an integral range with unit increments. Rectangular domains are relatively cheap to manage as they require only constant space. In local arrays, both the domain and the values mapped by the domain are kept in local memory. These are similar to arrays in traditional sequential languages where metadata and all the array elements are present in the same physical address space.

As SIDL Arrays. In Chapel, we implement normal SIDL arrays by first converting Chapel arrays into rectangular row-major ordered arrays and then wrapping this array in a generic custom array type implementing the SIDL interface. The upper and lower bounds for each dimension can be obtained from the domain. The bounds can in turn be used to compute the strides with the knowledge that rectangular arrays are always stored in row-major order. In addition, local arrays store data in a contiguous block similar to C-arrays. The pointer reference to this block of data can be obtained by invoking an externally defined function and defining the array argument to have an *inout* intent. The main challenge in implementing normal SIDL arrays was to convert the generic Chapel arrays to their corresponding specific SIDL versions without duplicating the BRAID library code while generating the IORs.

We use the BRAID code generator to determine array element types at compile-time and generate the appropriate function calls to create the SIDL arrays. The SIDL array API allows arbitrary accesses to the array and respects the row-major ordering of the Chapel array. Creating the wrapped representation is fairly cheap, taking $O(1)$ space, because the Babel representation involves generating the metadata from the Chapel domain information, and no copying of array elements is involved. In addition, since the wrapper is defined and managed by the Chapel runtime there is no additional responsibility for garbage collection. BRAID also supports rectangular slices on Chapel arrays and generates appropriate code to compute the metadata required to access the array elements.

As R-Arrays. Unlike SIDL arrays, the SIDL interface enforces certain constraints on r-arrays. One such constraint requires the r-arrays to have *column-major* order. Exposing Chapel arrays as r-arrays requires transparent conversion of the array to column-major order when passed over to the server implementation and, ironically, adds an additional overhead of the data copying. The calls for these conversions are inserted by the BRAID code generator. Since SIDL allows r-arrays to have the *inout* intent, the BRAID code generator needs to also insert function calls to sync back data from the column-major ordered r-array into the Chapel array for arguments with the inout intent. To minimize the amount of copy operations, thus making the program efficient, we created column-major ordered and *borrowed* rectangular arrays in Chapel.

Maximum speed: Borrowed Arrays. Both column-major and borrowed arrays implement the standard Chapel array interface and inherit all the syntactic sugar support Chapel provides for natively defined arrays. Hence there is no change in the Chapel code while using these arrays except during array creation. We require a small extension to the Chapel compiler to support borrowed arrays. Borrowed arrays have data blocks allocated external to the Chapel runtime unlike traditional Chapel arrays where each array uses a data block managed by the Chapel runtime. This avoids superfluous allocation and deallocation of array data blocks while passing the array reference between Babel/BRAID calls. It becomes the user's responsibility to manage the memory while using borrowed arrays.

```
// The domain of the distributed array
var overallDomain = [1..8, 1..8];
// Map the domain using a block cyclic domain map
var blockCyclicDomain = overallDomain dmapped
      BlockCyclic(startIdx=(1, 1), blocksize=(2, 3));
// Create the distributed array
var blockCyclicArray: [blockCyclicDomain] int;
```

```
// Distribution of the block—
// cyclic array on six locales
/***************
 0 0 0 1 1 1 0 0
 0 0 0 1 1 1 0 0
 2 2 2 3 3 3 2 2
 2 2 2 3 3 3 2 2
 4 4 4 5 5 5 4 4
 4 4 4 5 5 5 4 4
 0 0 0 1 1 1 0 0
 0 0 0 1 1 1 0 0
***************/
```

Figure 3. Chapel code snippet displaying use of domain maps to create a block-cyclic distributed array

4.2.2 Distributed Arrays

Chapel supports global-view (distributed) array implementations using domain maps [8, 9]. Domain maps are an additional layer above domains that map indices to locales allowing the user to define their own, possibly distributed, data distributions. As we did for local arrays, we concentrate our work only on distributed rectangular arrays. It is important to note that this places a restriction only on the index set and not on how the array data is actually distributed. Hence, these distributed arrays could be using any user-defined data distribution. Figure 3 shows the creation of a distributed array using a block-cyclic domain map. Each of the blocks can be considered as contiguous arrays on a single locale, *i. e.*, a local array. The Chapel runtime takes responsibility for handling any communication while accessing non-local elements of the distributed array.

As R-Arrays. Babel requires r-arrays to represent a contiguous block of local memory when they are passed across language boundaries. Since distributed arrays are not expected to refer to a single contiguous block of local memory, one way to enforce this constraint is to create a local copy of the distributed array before passing the array to an external function. When the parameter representing the distributed array is labeled with *inout* or *out* intents, the contents of the local array needs to be synced back into the distributed array. Accessing elements of the distributed array is always done via the Chapel runtime which transparently manages local and non-local accesses.

When a distributed array is used as an argument to a function defined in the SIDL interface expecting an r-array, BRAID generates code to convert the distributed array into r-arrays before passing the array on to the host language. As mentioned earlier, r-arrays in SIDL are required to refer to a contiguous block of memory in column-major order. Since distributed arrays are not required to refer to single contiguous local block of memory, a local contiguous block of memory is allocated to store the entire distributed array. The elements of the distributed array are then copied into this array in column-major order (using our column-major Chapel arrays) before being passed on to the target function via the IOR. The conversion of the distributed arrays into local arrays and the corresponding syncing of local array back into the distributed arrays is done by BRAID and is completely transparent to the user.

As SIDL Arrays. SIDL arrays are also required to be local arrays although they do not necessarily need to be contiguous; they may have a non-zero *stride*. Since we already support the r-array view of distributed arrays via copying, we have not implemented explicit support for the SIDL array view of distributed arrays. Instead we expose distributed arrays as their own SIDL type.

Direct access: The SIDL DistributedArray type. SIDL and raw arrays are assumed to be local, hence interoperating with distributed arrays that are transparently converted into SIDL or raw arrays requires:

SIDL definition

```
...
static void matrixMultiply(in rarray<int,2> a(n,m),
      in rarray<int,2> b(m,o),
      inout rarray<int,2> res(n,o),
      in int n, in int m, in int o);
...
```

Generated Chapel client code for R-arrays

```
// The generated stub method
_extern proc ArrayTest_ArrayOps_matrixMultiply_stub(
  in a: sidl_int__array, in b: sidl_int__array,
  inout res: sidl_int__array,
  inout ex: sidl_BaseInterface__object);

// The Chapel client accepts any Chapel array
proc matrixMultiply(
    in a: [?_babel_dom_a] int(32),
    in b: [?_babel_dom_b] int(32),
    inout res: [?_babel_dom_res] int(32),
    in n: int(32), in m: int(32), in o: int(32)) {

  var ex:sidl_BaseInterface__object;
  ...
  // Check for rectangular domains, unstrided, etc.
  sidl_perform_sanity_check(_babel_dom_res, "res");
  // Glue code to be generated to ensure/create a local Chapel array
  var _babel_data_res = sidl_get_opaque_data(res(_babel_dom_res.low));
  var _babel_local_res = sidl_ensure_local_array(res,_babel_data_res);
  var _babel_res_rank = _babel_dom_res.rank;

  ...
  // Create the IOR representation of the array
  var _babel_wrapped_local_res: sidl_int__array =
    sidl_int__array_borrow(
      int_ptr(_babel_local_res(_babel_local_res.domain.low)),
      _babel_res_rank, _babel_res_lower[1], _babel_res_upper[1],
      _babel_res_stride[1]);
  ...
  // Invoke the stub method to get to the server implementation
  ArrayTest_ArrayOps_matrixMultiply_stub(
    _babel_wrapped_local_a, _babel_wrapped_local_b,
    _babel_wrapped_local_res, ex);

  // Creating a borrowed array is a constant time operation
  var _babel_wrapped_local_res_sarray =
    new Array(res.eltType, sidl_int__array, _babel_wrapped_local_res);
  var _babel_wrapped_local_res_barray =
    sidl_create_borrowed_array2d(_babel_wrapped_local_res_sarray);

  // Sync back the data from the local array
  sidl_sync_nonlocal_array(_babel_wrapped_local_res_barray, res);
...
```

Figure 4. Glue code generated for r-array interoperability

- glue code to be generated to create a local array,
- copying all the data from the distributed array into the local array before the server method call,
- for *inout* and *out* arguments, syncing back the data from the local array into the distributed array after the method returns.

This copying/syncing of data is expensive and adversely affects the program's performance. Figure 4 shows an example of the glue code generated by BRAID to allow interoperability of rectangular Chapel arrays as r-arrays. Note that the generated client code accepts any Chapel array as an argument and then performs validity checks at compile time and runtime using utility methods. The same code is generated for all arrays but copying and syncing are effectively no-ops for local arrays.

To avoid the overhead of copying, we chose to create and expose distributed arrays as their own SIDL type as shown in Figure 5. Users can create specific instances of these distributed arrays using

```
package Arrays version 1.3 {
  class DistributedArray2dDouble {
    void initData(in opaque data);

    double get(in int idx1, in int idx2);

    void set(in double newVal, in int idx1, in int idx2);
  }
}
```

Figure 5. SIDL definition for a Distributed array

Chapel as a server language and enjoy the benefits of distributed computation from the traditional HPC languages which act as client languages. In addition, the elements of the distributed arrays can be accessed from the client languages similar to how SIDL array elements are accessed. The communication required to retrieve and work with remote elements is handled by the Chapel runtime and is abstracted away from the client language via the glue code generated by BRAID. We believe this to be a unique solution to handle interoperability of distributed arrays not only in Chapel but also in other PGAS languages using arbitrary communication libraries.

Figure 6 displays a modified version of the HPCC *ptrans* [19] benchmark which uses a server side implementation that works on the distributed arrays. In the example, the server side implementation accesses and mutates potentially remote elements of the distributed array transparently. Detailed timing results from running this program on a cluster can be found in Section 5.2.

4.3 Objects layout and the handling of foreign objects

SIDL specifies an object-oriented programming model that features single inheritance, abstract base classes, virtual function calls and multiple inheritance of interfaces. Class methods may be declared as virtual, static and final.

The Chapel language has native support for object-oriented programming which maps nicely onto SIDL and the Babel IOR. Figure 7 shows an example SIDL definition and the corresponding code generated by BRAID. In SIDL, multiple classes can be grouped into packages. With BRAID, we map those to Chapel modules (*cf.* line 1). Chapel follows the convention of using the file name to denote an implicit module. Since Chapel does not know class methods (*static methods* in C++ nomenclature) we create an additional module to serve as a namespace for static methods (*cf.* line 3).

The Chapel class holds a member variable *self*, which holds a reference to the Babel IOR data structure for that object (*cf.* line 7). The default constructor automatically calls the appropriate SIDL function to initialize this variable with a reference to a new object (*cf.* line 8). In addition to that, a second *copy*-constructor is created that can be used to wrap an existing IOR-object in a Chapel class (*cf.* line 14). The constructor calls the object's `addRef()` method which triggers Babel's reference counting scheme. Finally, a destructor is generated, which releases the reference to the IOR object and invokes the destructor of the IOR (*cf.* line 20). To bridge the time until Chapel supports a distributed garbage collector, there is the `delete` keyword to explicitly invoke a destructor and free the memory allocated by an object [11].

The Chapel language in general supports inheritance, however, it currently[6] does not support inheriting from classes that provide custom constructors. The reason for this is that there is no syntax yet to invoke parent constructors. We therefore resort to creating

[6] We refer to the Chapel Language Specification Version 0.8 [11], see also [6].

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

Table 2. 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

independent versions of each of the classes in an inheritance hierarchy, with each of the child classes containing definitions for all the inherited functions. Invocation of virtual methods will still work as expected, since the function definitions in Chapel are merely stubs that invoke the actual implementations through a virtual function table (the EPV, *cf.* Section 2.2) that is part of the IOR. Using this mechanism it is possible to write a Chapel class that "inherits" from a class that was originally implemented in, *e. g.*, C++.

Although the Babel inheritance is not really mapped to the Chapel type system it is still possible to perform operations such a typecasting on such objects. The price to pay is a slightly more verbose syntax. For instance, if an object c of type C implements an interface I, the syntax to up-cast c to an object i of type I would be:

```
var i = new I(c.cast_I());
```

This essentially invokes the copy constructor of I to wrap an up-casted IOR object returned by the C`.cast_I()` method. A similar method is automatically generated for each base class and each interface implemented by a class.

Manual down-casting is also possible with an automatically generated function. Let's assume we have an object b of type B and we want to cast that to the more specific object c of type C, which inherits from base class B, we would write:

```
var c = new C(B_static.cast_C(b.self));
```

The down-cast function is naturally not part of any class (`B_static` is a module), since there are no static member functions in Chapel.

4.4 Using Chapel as a library

So far we only discussed the possibility of calling an external function from Chapel code. BRAID also allows users to do the opposite thing, which was necessary to make the `DistributedArray` interface work. Instead of a stub, BRAID generates a skeleton which calls the implementation and performs the dual conversion operations on the arguments (a skeleton treats an *in*-argument the same way a stub treats an *out*-argument). In this case the *main* program is still a Chapel program that makes calls to external functions, which then again call back into Chapel.

We also have a working experimental version that allows the main program to be written in a language other than Chapel,[7] but this currently relies very much on implementation-specific details of how the Chapel compiler generates C code to make it work. In the future, we plan to make this feature more robust by working with the Chapel team on a standard to create Chapel libraries.

[7] In Babel terminology, this means using Chapel as a server implementation language (invoked via `braid --server=Chapel`).

SIDL definition for methods operating distributed arrays

```
import Arrays version 1.3;

package hpcc version 0.1 {
  class ParallelTranspose {
    // Utility function that
    // works on distributed arrays
    static void ptransCompute(
           in BlockCyclicDistArray2dDouble a,
           in BlockCyclicDistArray2dDouble c,
           in double beta,
           in int i,
           in int j);
  }
}
```

Sample Fortran 2003 server that uses the Chapel distributed array

```
! hpcc_ParallelTranspose_Impl.F03
module hpcc_ParallelTranspose_Impl
...
  subroutine ptransCompute_impl(a, c, beta, i, j, exception)
    ...
    implicit none
    type(hplsupport_BlockCyclicDistArray2dDouble_t), intent(in) :: a
    type(hplsupport_BlockCyclicDistArray2dDouble_t), intent(in) :: c
    real (kind=sidl_double), intent(in) :: beta
    integer (kind=sidl_int), intent(in) :: i
    integer (kind=sidl_int), intent(in) :: j
    type(sidl_BaseInterface_t), intent(out) :: exception

    ! DO−NOT−DELETE splicer.begin(hpcc.ParallelTranspose.ptransCompute)
    real (kind=sidl_double) :: a_ji
    real (kind=sidl_double) :: c_ij
    real (kind=sidl_double) :: new_val

    c_ij = get(c, i, j, exception);
    a_ji = get(a, j, i, exception);
    new_val = beta ∗ c_ij + a_ji;

    call set(c, new_val, i, j, exception);
    ! DO−NOT−DELETE splicer.end(hpcc.ParallelTranspose.ptransCompute)
  end subroutine ptransCompute_impl
...
end module hpcc_ParallelTranspose_Impl
```

Client code (in Chapel) that invokes a server code, *e. g.* in C, Fortran (above), Java, etc., that works with Chapel distributed arrays

```
// ptransHybrid.chpl [Modified version of ptrans.chpl from the standard Chapel distribution]
use Arrays;
use hpcc.ParallelTranspose_static;
...
// Utility function provided by BRAID to obtain a reference to the Chapel array
_extern proc GET_CHPL_REF(inData): int(32);
...
proc main() {
  // Create Block−Cyclic distributions
  const MatrixDist = new BlockCyclic(startIdx=(1,1), blocksize=(rowBlkSize, colBlkSize));
  const TransposeDist = new BlockCyclic(startIdx=(1,1), blocksize=(colBlkSize, rowBlkSize));
  // Declare domains (index sets) for the Matrix and its transpose
  const MatrixDom: domain(2, idxType) dmapped new dmap(MatrixDist) = [1..numrows, 1..numcols];
  const TransposeDom: domain(2, idxType) dmapped new dmap(TransposeDist) = [1..numcols, 1..numrows];
  // Declare the matrices themselves (distributed arrays)
  var A: [MatrixDom ] eltType;
  var C: [TransposeDom] eltType;
  ...
  forall (i,j) in TransposeDom do {
    // Creating wrappers are constant time operations
    var aWrapper = new Arrays.DistributedArray2dDouble();
    aWrapper.initData(GET_CHPL_REF(A));
    var cWrapper = new Arrays.DistributedArray2dDouble();
    cWrapper.initData(GET_CHPL_REF(C));

    // Call the server function via the ParallelTranspose_static stub
    ParallelTranspose_static.ptransCompute(aWrapper, cWrapper, beta, i, j);
  ...
  }
}
...
```

Figure 6. Interoperability of Chapel Distributed arrays

SIDL definition	Generated Chapel code (abbreviated)

```
package Inherit version 1.1 {
  class E2 extends C {
    string c();
    string e();
    static string m();
  };
}
```

```
// Inherit.chpl [implicit module Inherit]
use sidl;
module E2_static { // All the static methods of class E2
  proc m(): string { ... }
}
class E2 {
  var self: Inherit_E2__object;
  proc E2() { // Constructor
    var ex: sidl_BaseInterface__object;
    this.self = Inherit_E2__createObject(0, ex);
    Inherit_E2_addRef_stub( this.self, ex); ...
  }
  proc E2( in obj: Inherit_E2__object) { // Constructor for wrapping an existing object
    this.self = obj;
    ...
    Inherit_E2_addRef_stub( this.self, ex); ...
  }
  proc ~E2() { // Destructor
    var ex: sidl_BaseInterface__object;
    Inherit_E2_deleteRef_stub( this.self, ex);
    ...
    Inherit_E2__dtor_stub( this.self, ex); ...
  }
  proc c(out ex: BaseException): string { // Method c
    var ior_ex:sidl_BaseInterface__object;
    string r = Inherit_E2_c_stub( self, ior_ex);
    if (!IS_NULL(ior_ex)) ex = new BaseException(ior_ex);
    return r;
  }
  ...
```

Figure 7. Mapping SIDL classes into Chapel

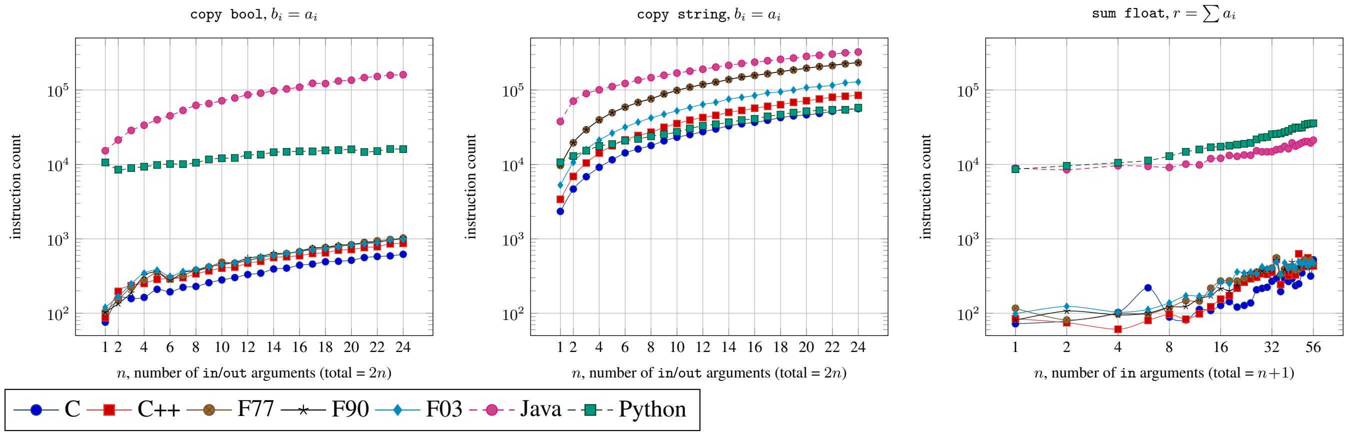


Figure 8. The cost of calling a function that copies or calculates the sum of n arguments, respectively

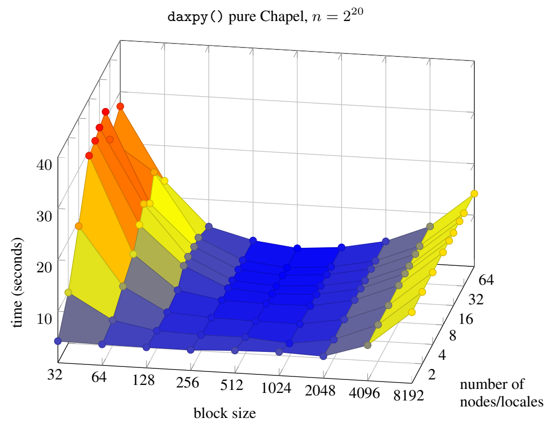
5. Experimental Results

5.1 Call overhead—Local/single-node performance

The first benchmark measures the overhead of converting function arguments from/to Chapel's native format into the native format of all the other supported languages. Figure 8 shows the number of instructions executed on an *x86-64* machine[8] to invoke a function in each of the supported languages from Chapel. This number was measured with the *instructions* performance counter provided by the *perf* [14] interface of Linux 2.6.32. To eliminate the instructions used for start-up and initialization, the instruction count of one execution of the benchmark program with one iteration was subtracted from that of the median of ten runs with 10^6+1 iterations each. The result was divided by 10^6 and plotted into the graph. The plots are logarithmic in the y-axis. The x-axis denotes the number n of *in*-(and *out*)-arguments passed to the function, so the total number of arguments was $2 \cdot n$ for the *copy* and $n + 1$ for the *sum* benchmark. arguments. The y-axis shows the number of instructions executed by the benchmark (lower values are better).

In `copy`, the server functions simply copy over all the ingoing arguments to the outgoing arguments, to show the overhead incurred for accessing scalar arguments. In the benchmarks we can see that Python adds a considerable overhead for the interpreter, which is comparable to that of Java for a small number of arguments. The

[8] The test machine was an Intel Xeon E5540 running at 2.53GHz, with 8 threads and 6GiB of main memory running Ubuntu 10.04. We used Chapel 1.3.0 for the client. The servers were compiled with the C, C++ and Fortran compilers of GCC 4.6.1 using standard optimization settings (`-O2`). The Python version was 2.6.5 and we used the SUN HotSpot 64-Bit Server version 1.6.0.22.



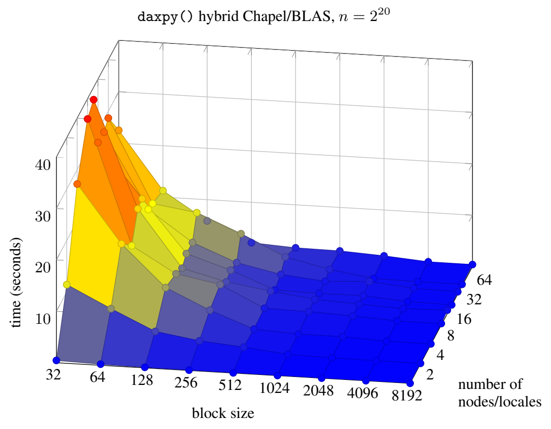


Figure 9. The `daxpy` Benchmark, hybrid and pure Chapel versions compared

performance of Java does not scale as well; this is because of the way *out*-parameters are handled. Since Java does not have pointers, Babel creates a wrapper object to hold each outgoing argument. This additional overhead shows especially in the `float` benchmark (keep in mind that the function body is empty apart from the copy operations), but becomes negligible as more data is being moved, such as in the `string` case.

In `sum`, the sum of all the input arguments is calculated on the server side. This benchmark is interesting because it shows that Java outperforms Python even on moderate workloads.

5.2 Local and Distributed Array benchmarks

We would like to mention that the execution times reported in these benchmarks are to quantify the overhead introduced by BRAID and not to be used to evaluate the quality code generated by the Chapel compiler. There is active work being done on the Chapel compiler to generate optimised code for the combinations of different language constructs and hardware architectures. We tried to keep our server implementations as close to the original Chapel implementations as possible. In addition, all the server code was implemented in C and compiled using the same flags as the C code produced by the Chapel compiler. The Chapel implementations to test array overheads ran on Linux clusters.[9] The software versions were identical to those used for the local tests (Section 5.1). To quantify the overhead of using BRAID-generated code with Chapel arrays, we implemented benchmarks with two main variants. The two variants were as follows:

Using local fragments of SIDL Distributed Arrays. To showcase our integration of distributed arrays we implemented the `daxpy()` function from BLAS [20] in Chapel. This was surprisingly easy to implement due to advanced support for working with arrays in Chapel. The implementation was just a single line:

$Y = a * X + Y;$

The hybrid implementation was about fifteen lines of user code (this excludes code generated by BRAID). We implemented the hybrid version by exposing the BLAS package via SIDL and using BRAID to generate the Chapel glue code. Since BLAS' `daxpy()` expects local arrays as input the hybrid Chapel version converted local fragments of the distributed arrays into SIDL arrays before invoking `daxpy()`. This benchmark displays successful use of local fragments of distributed arrays as SIDL arrays while invoking existing third party libraries. Figure 9 shows the variation in execution times of the hybrid version versus an optimized Chapel version. The hybrid version is up to ten times as fast as the corresponding pure Chapel version. This serves as an example of how BRAID can also be used by developers for rapid prototyping — first write prototype implementations of their problem with the simpler Chapel syntax and later optimize code using existing third party libraries when the need arises.

[9] The machine was a cluster with 324 nodes and InfiniBand interconnects. Each node was a 12-core Intel Xeon 5660 running a customized version of RedHat Linux. Each node had 24 GiB of memory. The Chapel compiler was configured to use GASnet's *ibv-conduit* with the MPI-based spawner.

Parallel calls with SIDL distributed arrays. We implemented the hybrid (Chapel as client and C as server) version of the HPCC *ptrans* benchmark which works with distributed arrays and is used to test the data transfer capacity of the network where remotely located data is frequently accessed by a node. Our *ptrans* implementation, shown in Figure 6, is a modified version of the one available in the standard Chapel distribution. We exposed the Chapel distributed arrays as SIDL objects and allowed an external function, called `ptransCompute()`, to be invoked in parallel from the Chapel program. Note that the body of this external function is not known to the Chapel compiler and hence not optimised to overlap communication with computation. As Table 2 shows, the BRAID-generated code introduces less than four percent overhead despite lacking this optimization. The overhead is attributed to the additional function calls required to invoke the server implementation and also for the callbacks from the server implementation back into the Chapel runtime to access and mutate the, possibly remote, elements of the distributed array.

6. Feature list

The complete list of features supported by the Chapel binding can be seen in Table 3. Generally speaking, there are no technical reasons blocking the support of the remaining Babel features and we expect to implement them in the near future. One interesting perspective is that BRAID will enable us to generate the code for contract enforcement directly in Chapel, which should yield better performance than the route over C that is used by all the Babel backends.

7. Conclusions and future work

In this paper we showed how to achieve interoperability between the Chapel (PGAS) language and a diverse array of traditional HPC languages. Our generated language bindings work both on local and

	Babel/SIDL Features	Status	
	Scalar data types	all	
	SIDL arrays	all*	(*no arrays of objects yet)
	Raw arrays	yes	
	Generic arrays	no	
	Objects	yes	
	Inheritance	yes	
	static calls	yes	
	virtual method calls	yes	
	structs	no	
	Babel RMI	no	
	Contracts	no	
	Exception handling	partial	
	Enumerations	yes	
+	Distributed arrays	yes*	(*not a Babel feature yet)

Table 3. Features supported by BRAID's Chapel $\leftrightarrow$ Babel binding

distributed Chapel programs and support all basic data types and we provide several options for dealing with distributed data types.

Some of the insights we gained during the design of the Chapel binding will be valuable to generalize this work for other PGAS languages: The first lesson learned is that some *modifications to the compiler and runtime system are necessary*, such as the introduction of *borrowed arrays* to avoid an otherwise costly copy operation, when passing large amounts of data from an external function to a Chapel program. The other major insight is that it is even feasible to *support distributed data types* without copying, by exposing them through an interface that is accessible from all languages. In fact, we defined the `DistributedArray` interface in SIDL, reusing our own infrastructure to make it completely portable.

One of the major advantages of the BRAID infrastructure is the separation of the interface code generation and the actual compilation of high-level intermediate code to source code. This separation will enable us to generalize the Chapel backend for other PGAS languages in the future. Backends for X10 and UPC are already being planned. While some parts, such as the argument conversion rules, will have to be rewritten, much of the Babel interface code generation will be reusable — by plugging in new code generator definitions for those languages. The ultimate challenge will then be to apply our experience to achieve interoperability between several PGAS languages, and enable them to share distributed data.

Acknowledgments. We would like to thank Brad Chamberlain and the rest of the Chapel team for many helpful discussions on interfacing with Chapel. We would also like to thank Philip Miller for helpful comments on early versions of the paper.

References

[1] Agarwal, A., Slee, M., and Kwiatkowski, M. Thrift: Scalable cross-language services implementation. Tech. rep., Facebook, April 2007.

[2] Armstrong, R., Kumfert, G., McInnes, L. C., Parker, S., Allan, B., Sottile, M., Epperly, T., and Dahlgren, T. The CCA component model for high-performance computing. *Intl. J. of Concurrency and Comput.: Practice and Experience 18*, 2 (2006).

[3] Beazley, D. SWIG homepage. `http://www.swig.org/`, 2011.

[4] BRAID website. `http://compose-hpc.sourceforge.net/`.

[5] Carlson, W. W., Draper, J. M., Culler, D., Yelick, K., Brooks, E., and Warren, K. Introduction to UPC and Language Specification. Tech. Rep. CCS-TR-99-157, Bowie, MD, May 1999.

[6] Chamberlain, B. Re: Calling the parent's constructor. `http://sourceforge.net/mailarchive/message.php?msg_id=27589055`, June 2011.

[7] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[8] Chamberlain, B. L., Choi, S.-E., Deitz, S. J., Iten, D., and Litvinov, V. Authoring user-defined domain maps in chapel. In *CUG 2011* (2011).

[9] Chamberlain, B. L., Deitz, S. J., Iten, D., and Choi, S.-E. User-defined distributions and layouts in chapel: Philosophy and framework. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2010), HotPar'10, USENIX Association, pp. 12–12.

[10] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not. 40* (Oct. 2005), 519–538.

[11] Cray Inc. Chapel language specification version 0.8. `http://chapel.cray.com/spec/spec-0.8.pdf`, April 2011.

[12] Cray, Inc. *Initial support for calling C routines from Chapel*, 1.3 ed., 2011. see doc/technotes/README.extern in the Chapel distribution.

[13] Dahlgren, T., Epperly, T., Kumfert, G., and Leek, J. *Babel User's Guide*. Lawrence Livermore National Laboratory, July 2006. version 1.0.0.

[14] de Melo, A. C. The new linux 'perf' tools. Slides from Linux Kongress, 2010. `http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf`.

[15] Epperly, T. G. W., Kumfert, G., Dahlgren, T., Ebner, D., Leek, J., Prantl, A., and Kohn, S. High-performance language interoperability for scientific computing through Babel. *IJHPCA*, 1094342011414036 (2011).

[16] Free Software Foundation. *Shared library support for GNU*, 2.4 ed., 2010. `http://www.gnu.org/software/libtool/manual/libtool.html`.

[17] Google. Protocol Buffers – Google's Data Interchange Format. Documentation and open source release. `http://code.google.com/apis/protocolbuffers/`.

[18] Kumfert, G., Leek, J., and Epperly, T. Babel remote method invocation. In *IPDPS* (2007), IEEE, pp. 1–10.

[19] Luszczek, P. R., Bailey, D. H., Dongarra, J. J., Kepner, J., Lucas, R. F., Rabenseifner, R., and Takahashi, D. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[20] National Science Foundation, and Department of Energy. BLAS. http://www.netlib.org/blas/, 2011.

[21] Numerich, R. W., and Reid, J. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum 24*, 2 (2005), 4–17.

[22] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998.

[23] Rasmussen, C. E., Sottile, M. J., Shende, S., and Malony, A. D. Bridging the language gap in scientific computing: the Chasm approach. *Concurrency and Computation: Practice and Experience 18*, 2 (2006), 151–162.

[24] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O., and Grove, D. *X10 Language Specification*, version 2.2 ed. IBM, May 31 2011.

[25] Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*. Wiley and Sons, 1998.

[26] XPCOM Website. `https://developer.mozilla.org/en/XPCOM`.