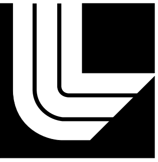

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-AR-499171

Composite Parallelism:Creating Interoperability between PGAS Languages, HPCS Languages and Message Passing Libraries

SCW0839 Progress Report

T. Epperly, A. Prantl, B. Chamberlain

September 14, 2011

Contents

Abstract **3**

1 Introduction **4**
1.1 Background and Related Work . . . 4
1.1.1 Message Passing Libraries . . . 5
1.1.2 Thread Parallelism . . . 5
1.1.3 PGAS languages . . . 5
1.1.4 HPCS languages . . . 6
1.1.5 Mixing Parallel Libraries . . . 7
1.1.6 SIDL . . . 8
1.1.7 Babel Intermediate Object Representation (IOR) . . . 8
1.2 Challenges . . . 9

2 Research Objectives **10**
2.1 Accomplishments . . . 10
2.2 Implementation environment . . . 11
2.3 The interface . . . 12
2.3.1 Scalar datatypes . . . 12
2.3.2 Array implementation: Local Arrays . . . 13
2.3.2.1 Optimization: Borrowed Arrays . . . 14
2.3.3 Array implementation: Distributed Arrays . . . 14
2.3.3.1 Raw Arrays . . . 14
2.3.3.2 SIDL Arrays . . . 15
2.3.3.3 The SIDL DistributedArray type . . . 15
2.3.4 Objects layout and the handling of foreign objects . . . 15
2.4 Alterations of Chapel compiler and runtime . . . 19

3 Ongoing Research Objectives **20**

4 Results **21**
4.1 Call overhead—Local/single-node performance . . . 21
4.2 Local and Distributed Array benchmarks . . . 21
4.2.1 Sequential calls on Local Arrays . . . 22
4.2.2 Using local fragments of SIDL Distributed Arrays . . . 22
4.2.3 Parallel calls with SIDL distributed arrays . . . 22
4.3 Feature list . . . 24

5 Products **25**
5.1 Publications and presentations . . . 25
5.2 Project Website . . . 25
5.3 Software . . . 25
5.4 Collaborations . . . 26

6 Conclusions **27**

7 Bibliography **28**

Abstract

We are creating a mixed-language environment that supports the construction of petascale applications from arbitrary combinations of software written in PGAS languages (Co-Array Fortran, UPC, and Titanium), HPCS languages (Chapel, X10, and Fortress), and message passing libraries called from sequential languages (e. g., Fortran, C, C++, and Python). Such an environment is useful to improve scientific exchange of codes and ideas in general, but particularly important now because the computational science community has entered the age of petascale computing without a clear understanding of how best to program the millions of processors required at this extreme. Our research will benefit multiple communities. It will facilitate experimentation with the newer parallel programming languages in production settings. It will increase the utility of the existing investment in parallel scientific codes, by providing a migration path to be useful in new programming paradigms. Our work will also promote compositions of parallelism; so researchers can interchangeably target different models and/or languages to specific levels of parallelism in these massively hierarchical machines.

Our SIDL and Babel technologies have proven effective for sequential languages, such as Fortran, C, C++, Python, and Java, but are insufficient for parallel languages. Setting aside the numerous technical details, the distinct programming, execution, and synchronization models presented by the PGAS and HPCS languages raise fundamental questions about the composability of parallel systems. For example, it is not clear what conditions are necessary to guarantee correctness of an interlanguage call embedded in a parallel loop. Interoperability must be defined on compositions of parallel programming and execution models instead of a single thread as it is now. Exchange of global addressable and distributed data structures must be supported across languages in addition to simple local data structures.

Our plan is to answer these questions and develop a new interoperability tool as both testbed and proof-of-concept. We will also leverage all the serial language interoperability features developed over the years with Babel. The research methodology and techniques that we developed in creating Babel will be employed and extended on this more ambitious mixed-parallelism challenge. Several illustrative examples of the research methodology are provided in the proposal text.

This project will lay the theoretical and technical foundations for this next-generation mixed-language programming environment. It will also produce a proof-of-principle code sufficient for other PGAS and HPCS language developers to assess the merits of our approach. We hope to persuade and partner with each of these groups in the development of production-quality language bindings in subsequent projects.

1. Introduction

COMPOSITE-PARALLELISM is a project to research into general parallel coupling models as a first step to enable the arbitrary mixing of petascale software written in PGAS languages, HPCS languages, and sequential languages employing message passing libraries. The goal is to resolve enough of the hard interoperability questions to generate a working proof-of-principle that supports high performance interoperability between codes written in C/C++ with MPI and the HPCS language, Chapel. Upon completion of this project, our discoveries, design, and working prototype will provide the PGAS and HPCS communities the means to assess the merits of our approach. We anticipate this research will lead to deeper collaborations with the PGAS and HPCS language communities and follow-on efforts to create production-quality bindings to mix these languages at petascale.

1.1 Background and Related Work

In recent years, performance and power considerations have forced chip designers to embrace multi-core processors instead of higher frequency single-core processors. As such, most current high performance computing systems are distributed collections of multi-core nodes. As the number of cores in high-performance computers scale up dramatically with increasingly complex memory hierarchies, developers of scientific software are increasingly challenged to express complex parallel simulations effectively and efficiently using current mainstream programming models.

It is yet to be shown whether the current mainstream approach of combining a serial programming language, such as C++ or Fortran, with a message passing interface (MPI) will scale past petascale machines. Even now, the pervasive use of *multi-*[1] or *many-core*[2] [6] processing nodes makes it appealing to combine an inter-node communication method like MPI with second layer of thread-level parallelism, such as OpenMP. With the increasing gap between communication latency and CPU power, it is anticipated that the key to performance on future systems will be the control over the locality of the data.

The Partitioned Global Address Space (PGAS) programming model offers a new, alternative paradigm that may well turn out to be a more natural fit to current and future high performance systems. The PGAS model 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 available core) with separate address spaces. Performance is gained by exploiting data locality. Transparent access to memory locations on other execution contexts is usually supported by one-sided communication semantics.

Languages like Co-Array Fortran [54], UPC [31], and Titanium [67] were the first to implement this model. The PGAS model was also adopted by the DARPA-funded HPCS languages, Chapel [17], and X10 [20]. By implementing a partitioned global address space they offer a simpler and more familiar "global-view" programming model, which makes the development much easier. However, it would be impractical (and very costly) to rewrite existing applications entirely in these languages. Consequently, basic interoperability with the host languages of existing applications (C++, Fortran, etc.) is one of the keys to widespread acceptance of the PGAS languages.

In the remainder of this section, we explain the background and the related work in detail and show how the results of our previous efforts can be applied to solve the problems at hand.

[1] Incremental doubling of very large cores on a die (e. g., IBM Power, Intel Core, etc.)

[2] Extreme numbers of very simple, low-power cores on a die as is commonplace in embedded systems.

1.1.1 Message Passing Libraries

Message passing libraries are relatively easy to implement, labor intensive to use, and dominate the high-end computing (HEC) community. This model typically supports data parallelism between multiple cooperating instances of the same program (SPMD) and requires the programmer create all the local and remote accessors, synchronization primitives, and algorithms necessary to realize distributed data structures. Generally, these libraries implement message passing in either two-sided or one-sided form. The main trade-off between the two is that two-sided is more restrictive to program since both sides of the communication must be properly orchestrated whereas the one-sided is more vulnerable to non-deterministic faults such as race conditions and synchronization problems.

The *de facto* standard for distributed memory parallel programming in HEC is the Message Passing Interface (MPI) [60]. MPI was developed by a consortium of industry, academia, and laboratories to reduce the costs associated with each supercomputer vendor releasing its own proprietary message passing library. MPI-1 supports two-sided communications, broadcasts, reductions, and complete data exchanges ("all-to-all"). MPI-2 added a form of one-sided communication, dynamic task creation, and inter-communicators that enable distinct programs to share a communication link [36]. Parallel Virtual Machine (PVM) [35] is another two-sided message passing library from the same era as MPI-1. It is noteworthy for having advanced capabilities — such as spawning new parallel tasks — long before the advent of MPI-2.

One-sided communication protocols enable processors to `put()` and `get()` data from remote memory, without engaging the corresponding remote processor. One of the earliest one-sided protocols is SHMEM [7] developed at CRAY. The Aggregate Remote Memory Copy Interface (ARMCI) [52] from Pacific Northwest National Laboratory supports one-sided communication more portably across architectures without sacrificing performance by calling directly to native network primitives. ARMCI has been used in implementing the Global Array Toolkit [53], Co-array Fortran [30], and a Generalized Portable SHMEM (GPSHMEM) [56]. GASNet [11] is a one-sided messaging library developed at Berkeley. It is used in the PGAS languages Co-array Fortran, UPC [21, 22], and Titanium [62], which are discussed in more detail in Section 1.1.3. The HPCS language Chapel (Section 1.1.4) also supports a GASnet back end. GASNet's core is heavily based on Active Messages [65] and, like ARMCI, implemented to interface directly with the native network.

1.1.2 Thread Parallelism

POSIX Threads, also known as *Pthreads*, is an API for managing threads at a very low level. It is not uncommon to mix Pthreads into MPI to add some thread parallelism within a node. Although the MPI standard did not originally specify anything about threads, implementations gradually adopted some form of thread safety. Four levels of thread support became codified in the MPI-2 standard.

OpenMP is a shared memory programming model specified as extensions to C, C++, and Fortran languages [19]. OpenMP is generally acknowledged an easier model to program than MPI, but — being a shared memory model — less scalable. It is commonly used to add loop-level parallelism to otherwise sequential code by added structured pragmas. Support for task parallelism was recently added [2], but there is still no provision for OpenMP to interoperate with other thread libraries such as Pthreads.

1.1.3 PGAS languages

Partitioned Global Address Space (PGAS) languages support a programming model where the distributed memory hardware is programmed *as if* it were globally shared memory. The compiler and runtime systems employ a one-sided message passing library such as ARMCI or GASNet to implement the virtual global address space. This programming model facilitates building data structures and communication between cooperating program instances. However, by reducing the distributed memory model to a multi-threaded one, all the typical challenges of minimizing synchronization to increase performance without opening up

opportunities for race conditions or deadlock remains on the developer. The three most prominent PGAS languages are Co-array Fortran, UPC, and Titanium which are Fortran-, C-, and Java-based, respectively. Developed and maintained by separate communities, the PGAS languages have distinct interpretations of how to best support a virtual global address model.

Co-array Fortran (formerly known as F−−) [54] started as an extension to Fortran 95 and is now in large parts included in the Fortran 2008 standard [55, 33]. It introduces a *co-array*: an extra dimension that spans the cooperating instances of a SPMD program (called *images*). Images access remote instances of a variable by merely indexing the co-array of their local analog. Synchronization routines are also introduced to keep the images logically consistent. Though the model relieves the programmer of explicit one-sided or two-sided message passing, there can still be a great deal of data structure fragmentation. For example, distributed arrays are implemented by hand where the programmer typically allocates co-arrays of length $n/\#\text{images}$ on each image and explicitly manages the mapping between global and co-array indexing. The co-array index is visually distinguished with square instead of rounded brackets.

Unified Parallel C (UPC) [14, 31, 64] is the synthesis of three independent efforts to parallelize ANSI C: AC [13], CC++ [43] and Split-C [45]. This community refers to separate instances as *threads* and introduces a `shared` keyword that modifies arrays to be distributed between threads in cyclic or block-cyclic manner. Although it provides a more global view of arrays (without an explicit co-array index), it also inherits C's deficiencies with supporting true multidimensional arrays. UPC also introduces a parallel looping structure, `upc_forall` in which global iterations of a C for loop are assigned to threads using an *affinity expression*. This is not wholly consistent with the SPMD model, but useful nonetheless. UPC also ties pointer affinity into the type system of the language. Data as well as pointers to data can be declared `private` (meaning local to a thread) or `shared`, which enhances safety and optimization opportunities for shared address space programs.

The Berkeley-led Titanium [41, 67] language is much more aggressive about adding features that are particular to the needs of the HEC community. It is a Java-based dialect, but implemented as a source-to-source conversion to C code having no virtual machine or just in time (JIT) compilation. Titanium also does not support Java-style class loading or thread creation. Allowing the creation of threads within a Titanium process is under investigation [66, §5.4]. *Regions* are introduced as a kind of memory pool to provide a higher performance alternative to garbage collection for memory management. True multidimensional arrays are introduced with iterators and no-copy subarrays. To coordinate between instances of a SPMD run, Titanium supports synchronization and communication primitives, `private` and `shared` modifiers to data and references similar to UPC, and a `single` modifier to methods and variables to facilitate type-safe synchronization.

1.1.4 HPCS languages

DARPA's High Productivity Computer Systems (HPCS) program [42] included new programming languages as part of its broad mission to facilitate a new generation of economically viable *high productivity* computing systems. Encouraged to "think big," vendors are developing very ambitious languages with sometimes exotic capabilities.

Chapel [23] is Cray's contribution to HPCS and has intellectual heritage from ZPL [16, 28, 61] and HPF [39, 40, 44]. *Domains* are first class types that defines size, shape, and indices of an array and supports parallel iteration. A generalization of ZPL's region concept, Chapel's domains are very expressive, supporting sets, graphs, and associative arrays. Setting aside the more exotic *indefinite domains* and *opaque domains*, the third major type of domain, Chapel's *rectangular domain*, is what supports multidimensional rectilinear index sets found in traditional array languages. A Chapel *locale* is a generalization to what is commonly thought of as a SMP node in a cluster. Chapel's domains distributed among locales form *domain distributions* which can be shared by arrays and determine their size, shape, and data distribution. All of this terminology,

abstraction, and infrastructure makes for a system that can support arbitrarily complex data distributions [29]. Chapel supports multiple communication mechanisms, called *conduits*, including a GASnet back end.

IBM's X10 language is an "extended subset of Java" — meaning that a few Java constructs such as its concurrency models and array type were removed and significant new features (a different concurrency model, multidimensional arrays with distributions and many others) were added. The new concurrency model is carefully defined with conditions under which safety from deadlock and race conditions is guaranteed. X10's *place* is similar to Chapel's locale, but with a less strict mapping of places to actual compute nodes. Distributions, though similar in spirit to Chapel's distributions are not as expressive. X10 communication style for point-to-point (or using X10 vernacular *inter-place*) communications trends to the obtuse, suggesting the use of either nested `async` clauses or a `future-force` and `async` pair for best performance.

Fortress [4] started as Sun's HPCS language and is now developed as an open source project. With the motto "Do for Fortran what Java did for C" [3], this language is the most radical design of the HPCS group. It is programmed in Unicode and is designed to support mathematical notation with Greek letters, subscripts, dozens of mathematical operators, and *italicized* variable names. It supports object-oriented, procedural, and functional programming styles as well as thread, task, and data parallelism. The Fortress mindset is that everything is parallel. Developers have to add extra declarations to restrict it to serial execution when needed. In a case of unfortunate overloading of terms Fortress regions describe machine resources similar to Chapel locales and X10 places — but completely unrelated to a Titanium region. Unlike the Chapel or X10 analogs, Fortress' regions are hierarchical in nature, permitting the user to refer to architectural resources at multiple levels of detail. Fortress also includes distributions and allows user to define their own. However distributions are only advisory to Fortress' managed runtime environment, which is likely one of those performance-enhancing choices that complicates interoperability.

In summary, the HPCS languages are very experimental and young. It is not uncommon for serial programming languages to incubate a decade or more before they are ready for production use. HPCS languages, with their exceptionally sophisticated parallel inter-process optimization needs are expected to require even more time to reach full maturity. This works to our advantage because it gives us opportunity to influence the interoperability these projects in their formative years. It should also be mentioned that the line between is HPCS and PGAS languages is blurred, as most HPCS languages support more and more features of the PGAS model.

1.1.5 Mixing Parallel Libraries

Achieving performance at petascale, regardless of the architecture, will hinge on maximizing locality and parallelism at all levels of the hardware's hierarchy. Starting from message-passing and thread libraries which focused on a single form of parallelism, the increasing feature sets of PGAS/HPCS languages reflects the recognition that tying in multiple levels of parallelism is critical.

An intuitive combination for distributed memory clusters of symmetric multiprocessor (SMP) nodes is to use MPI for inter-node communication and OpenMP for each group of processors within a node. While the pros and cons of this approach have been hashed out in literature for almost a decade [12] enthusiasm for this combination in the long term is mixed. Depending on the application and architecture, a majority of HEC gets better performance by simply using an MPI process on every processor, instead of the MPI and OpenMP combination. There is some concern that MPI itself will fail to scale beyond some number of processors, which would be a tipping point in favor of OpenMP and MPI.

Efforts to employ multiple parallel models is not relegated to the purview of language design. Computational science offers several examples where production codes mix additional forms of parallelism with MPI. Two examples of networking multiple MPI jobs — a Multiple Programs Multiple Data (MPMD) model — are from the climate and space weather communities. The Community Climate System Model (CCSM) [37] uses the Model Coupling Toolkit (MCT) [48] to couple separate SPMD codes representing ocean, atmosphere,

sea ice, etc. CCSM also falls back to a co-existing SPMD version for machines that cannot support the MPMD model [38]. The Center for Integrated Space Weather Modeling (CISM) [1] uses InterComm [63] to exchange distributed arrays between distinct MPI codes. Computational chemists use a combination of Global Arrays and MPI groups to increase performance by focusing on multiple levels of parallelism within a SPMD arrangement [46].

We have made contributions to supporting multiple levels of parallelism with the introduction of BabelRMI [47]. Babel started as a strictly a language interoperability tool, but has since matured to support distributed computing via remote method invocation (RMI). In support of the Cooperative Parallelism project [50], we used BabelRMI to combine MPI parallel, multi-threaded, and serial executables in a single federated application. This collaboration led to advances in materials science [10] and multiscale algorithms in particular with their cached adaptive sampling [9, 8].

1.1.6 SIDL

SIDL is the Scientific Interface Definition Language used be Babel. 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 structs across various languages. Babel also provides a consistent exception handling 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 CCA [5] component architecture, which provides additional capabilities such as dynamic composition of applications.

1.1.7 Babel Intermediate Object Representation (IOR)

The connection between all the different languages supported in Babel is the intermediate object representation (IOR). Babel wants to own the type system and is responsible to convert the between IOR and each language's native data format.

The IOR is the native format used to define all data types that can be 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 (EPV) in the Babel jargon) 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 [32]. 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.[3]

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

[3] Fortran 90/95 breaks this requirement. Babel uses `libChasm` [59] which actually reverse engineered a large number Fortran compilers to still achieve the interoperability.

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.

1.2 Challenges

Interestingly, the main selling points of PGAS/HPCS languages also contribute some of the main challenges for achieving interoperability with other (parallel or serial) programming languages:

Distributed data types. What may seem like a simple variable in the source code might actually be a reference to a value that is stored on 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 PGAS/HPCS languages, parallelism is built into the language (e. g., through `forall` statements) and the decision when to off-load work into a separate thread or process is mostly performed by the runtime system.

Closed ecosystem. The way that most PGAS/HPCS languages are designed, there is an implicit assumption that programs will be written entirely in one language. While this approach enables the compiler to produce better performing code by doing whole program optimization, it significantly hampers the adoption of PGAS languages by a broader audience, as they become “island” programming languages. From our experiences with the Chapel team, however, the community seems to be very interested in changing this situation.

In addition to the above mentioned 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 slightly different representation of data types and storage formats for compound data types.

Our interoperability tool provides developers of scientific code an upgrade path that allows them to combine components modern parallel programming languages with existing packages written in Fortran or C++ (or Java, or Python).

2. Research Objectives

This section describes the research carried out by the end of Year two of the project. The planned research work for Year 3 (2011–2012) will be sketched in section 3. The original project proposal had work on an MPI runtime and RMI support planned for Year 2 and the creation of a Chapel binding planned for Year 3. However, it became clear to us that it was instrumental to first gain the experience of connecting one language to the set of traditional HPC languages before we were in the position to design a message-passing data exchange between those languages. For this reason we decided to first work on the Chapel binding and defer the design of the message-passing runtime to Year 3.

2.1 Accomplishments

The major accomplishment at this point in the project (the end of Year 2) is our client-side Chapel binding for Babel. With this we were able to demonstrate interoperability for basic data types, arrays and SIDL objects between Chapel and any of Babel's supported languages. Moreover, we were able to demonstrate two ways of accessing Chapel's *distributed data types* from other programming languages:

- The first option we implemented is *transparent* support for distributed arrays. This means that all data that is stored on a remote locale (another node) is copied into a local data structure prior to a function call. After the function returned, the modified data is distributed back onto the different locales. While this method impedes an obvious performance hit, it is portable to the level that external functions do not even need to know about the existence of different locales. This makes it trivial to reuse existing code. For loosely coupled components the drop in performance may well be negligible.
- The second option exposes the distributed array through a SIDL interface (more details on this in Section 2.3.3.3). From a usability perspective, this is similar to the existing SIDL Array API, which exposes the array's data through a get/set method. Whenever an element of the distributed array is accessed through the interface, the Chapel runtime is invoked and will perform the low-level remote access through GASnet. This way, external functions that access only part of the distributed array will get significantly better performance than if they would have to copy all of the array's data.

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 2.3.2.1, 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.

In the subsequent sections, we will explain how we realized these goals by building upon and extending the Babel infrastructure.

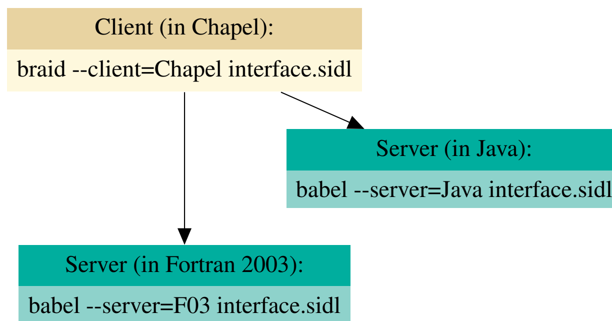

Figure 2.1: Combining BRAID (Chapel, ...) and Babel (C++, Fortran, ...) to generate interface code

2.2 Implementation environment

For the implementation of our prototype we were able to exploit synergies with the related COMPOSE-HPC project[1] by reusing the BRAID system for Rewriting Abstract Intermediate Descriptions. BRAID is a multi-faceted, term-based system for generating language interoperability glue code designed and developed as part of the COMPOSE-HPC project to be a reusable component of software composability tools. We implemented all the glue code generation using BRAID's facilities; 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.

From a user's perspective, BRAID is the tool that generates glue code for parallel PGAS/HPCS 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 2.1 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 the implementation language; it is the design of the language backends. 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 (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), the 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.

[1]COMPOSE-HPC: *Software Composition for Extreme Scale Computational Science and Engineering* is also funded by DOE Office of Science, Advanced Scientific Computing Research, program manager Lucy Nowell.

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 2.1: Scalar Data Types in SIDL on a 64-bit machine

The Makefile created by BRAID intercepts this process after the C files have been generated and builds a *libtool* [34] 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 limited support for interfacing with C code via the `extern` keyword [25]. BRAID uses this interface as an entry point to open up the language for all the other languages supported by Babel.

2.3 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.

2.3.1 Scalar datatypes

Table 2.1 lists the scalar types supported by SIDL and the corresponding Chapel types used by the the skeleton or stub while converting Chapel code from or into the IOR respectively.

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
- native types for both single-precision and double-precision complex numbers
- native support for opaque types: that allow data to be passed around through Babel/BRAID back into the original address space
- native support for enumerated type to defines a set of named constants

The Babel IOR and the Chapel compiler generate different representations for complex numbers, hence BRAID generates glue code to pass around copies as shown in Figure 2.2. 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. 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).

```
void Args_Basic_testFcomplex_stub(
    struct Args_Basic__object* self,
    /* inout */ _complex64* c,
    struct sidl_BaseInterface__object** ex) {
  // Convert to Babel IOR complex
  struct sidl_fcomplex _babel_c;
  _babel_c.real = c−>re;
  _babel_c.imaginary = c−>im;
  (*self−>d_epv−>f_testFcomplex)(self, &_babel_c, ex);
  // Sync data back as c is defined to be inout
  c−>re = _babel_c.real;
  c−>im = _babel_c.imaginary;
  return;
}
```

```
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 2.2: Stub code generated by BRAID for Chapel `complex` and `char` types

2.3.2 Array implementation: 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 (*cf.* [26], 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.

Chapel supports generic arrays using the concept of domains [24]. Domains are used to specify the index set including the number and size of the dimensions. In Chapel, arrays map indices from a domain to variables of homogeneous types. 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 integral ranges with unit increments. Rectangular domains are relatively cheap to manage as they only require 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.

In Chapel, we implement normal SIDL arrays by wrapping the local arrays 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.

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 presenting the Chapel domain information in the metadata 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.

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

```
// 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 2.3: Chapel code snippet displaying use of domain maps to create a block-cyclic distributed array

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.

2.3.2.1 Optimization: Borrowed Arrays

To minimize the amount of copy operations, thus making the program efficient, we introduced column-major ordered and *borrowed* rectangular arrays in Chapel.

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.

2.3.3 Array implementation: Distributed Arrays

Chapel supports global-view (distributed) array implementations using domain maps [18]. Domain maps are an additional layer above domains that map indices to locales allowing the user to define their own 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 2.3 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.

2.3.3.1 Raw Arrays

Babel requires r-arrays and SIDL arrays to represent a contiguous block of local memory whenever arrays 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 on 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.

2.3.3.2 SIDL Arrays

SIDL arrays are also required to be local arrays. 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.

2.3.3.3 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:

- 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. 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 2.4. Users can create specific instances of these distributed arrays using 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.

Figure 2.5 displays a modified version of the HPCC *ptrans* [49] 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 4.2.3.

2.3.4 Objects layout and the handling of foreign objects

The SIDL languages 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 2.6 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

```
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 2.4: SIDL definition for a Distributed array

modules. 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.

The Chapel class holds a member variable *ior*, which holds a reference to the Babel IOR data structure for that object. The default constructor automatically calls the appropriate SIDL function to initialize this variable with a reference to a new object. In addition to that, a second *copy*-constructor is created that can be used to wrap an existing IOR-object in a Chapel class. 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. Chapel will eventually support a distributed garbage collector; in the meantime there is the `delete` keyword to explicitly invoke a destructor and free the memory allocated by an object [24].

The Chapel language in general supports inheritance, however, it currently[2] 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 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 1.1.7) 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 a 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.

[2] We refer to the Chapel Language Specification Version 0.8 [24], see also [15].

SIDL definition for methods operating distributed arrays	Sample Fortran 2003 Server implementation that uses distributed arrays

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 implementation that uses distributed arrays:

```
! 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 (potentially in one of C, Fortran, Java, etc.) that works with the 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 2.5: Interoperability of Chapel Distributed arrays

SIDL definition	Generated Chapel code (abbreviated)

<pre>package Inherit version 1.1 {
  class E2 extends C {
    string c();
    string e();
    static string m();
  };
}</pre>
	<pre>// 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;
	  }
	  ...</pre>

Figure 2.6: Mapping SIDL classes into Chapel

2.4 Alterations of Chapel compiler and runtime

On two occasions we found it necessary to make changes to the Chapel compiler and runtime system. We are working together with our co-investigator Brad Chamberlain at Cray to integrate those changes into a future release of Chapel:

- Borrowed arrays (*cf.* Section 2.3.2.1) are a performance optimization for passing large amounts of data between Chapel code and external functions without copying the payload between the systems.
- Right now, we only have experimental support for a Chapel server binding (*i. e.*, other languages calling into Chapel). This is because Chapel right now has no way of compiling a library. We are currently working to flesh out a standard for Chapel libraries together with the Chapel team. This will deal with issues such as name mangling, deferred runtime initialization and generally separating runtime support functions (that is shared between several compilation units) from the generated code.

3. Ongoing Research Objectives

Our existing prototype allows Chapel programs to call functions and instantiate objects written in C, C++, Fortran 77–2008, Java or Python. For Year 3, we have the following milestones planned:

Server-side Chapel interoperability. This is part of an ongoing collaboration with our co-investigator Brad Chamberlain at Cray. We will need to standardize the name-mangling of identifiers in Chapel to make calls into Chapel code work reliably. We need to develop a standard for how to initialize the Chapel runtime from an external non-Chapel function. The Chapel compiler needs to be extended to generate libraries instead of stand-alone programs. We will also research how to map SIDL generic arrays and structs onto Chapel's type system (*cf.* Section 4.3).

We will continue to work on standardizing our extensions (*cf.* Section 2.4) to the Chapel compiler and runtime. Ideally, our modifications will be accepted into a future release of the Chapel language.

Researching interoperable distributed data types. We will investigate a possible integration of C+MPI code with Chapel code to create a prototype for future interoperability between distributed data types of two PGAS/HPCS languages.

Given the funding situation and the reduced budget, we will not be able to implement the originally proposed remote method invocation (RMI) work (building a stateless RMI protocol, support parallel RMI calls).

Generalize our results. As a stepping stone for related future projects we will explore how we can apply our approach to other PGAS/HPCS languages, such as UPC, X10 and Co-Array Fortran. This will lay the foundation for a future interoperability standard that targets not only PGAS/HPCS languages and traditional languages, but ideally interoperability between multiple PGAS/HPCS languages.

We intend to submit a follow-up proposal for a project that applies what we did for Chapel in this project to other PGAS/HPCS languages, such as UPC and X10.

4. Results

In this section we present detailed benchmarks that analyze the performance overhead incurred by our interoperability approach. It starts in section 4.1 with local benchmarks that show the performance of method invocations for Chapel calling each of the 7 languages. In section 4.2.1 we show data for for our local array implementation. Finally, section 4.2.3 shows measurements of the parallel performance for our distributed array implementation — this was done on a cluster of Linux machines.

4.1 Call overhead—Local/single-node performance

Figure 4.1 shows the number of instructions executed on an *x86-64* machine[1] 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* [27] 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 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.

4.2 Local and Distributed Array benchmarks

To quantify the overhead of using BRAID-generated code with Chapel arrays, we implemented benchmarks with three main variants. 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 first variant was run on a shared memory machine.[2] For the remaining two variants, the Chapel implementations to test array overheads ran on Linux clusters.[3] The software versions were identical to those used for the local tests (Section 4.1). The three

[1]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.

[2]The machine was a quad-core iMac with an Intel Core i7 processor with a total of 8GB memory.

[3]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.

variants were as follows:

4.2.1 Sequential calls on Local Arrays

In this variant we use only local arrays inside sequentially executing functions. To test this we modified the HPL implementation from the HPCC benchmark [49] that is provided in the standard Chapel distribution. HPL is used to solve dense linear equations and involves *LU*-factorization steps that execute in parallel. One such step called *panelSolve* is present in the aforementioned Chapel program which works sequentially on local arrays. We exposed this function via SIDL and the same algorithm was implemented in the server C implementation. We then used BRAID-generated code to invoke the external function from Chapel.

The results in Table 4.1 show that working with local arrays introduces an overhead of around fifteen percent on average. This can be attributed to relatively frequent function calls to the server implementation due to smaller input array and block size. Each function calls also includes the overhead of converting the input to the IOR representation. With larger inputs, the constant overhead will progressively diminish with respect to the larger amount of work done inside the function.

4.2.2 Using local fragments of SIDL Distributed Arrays

To showcase our integration of distributed arrays we implemented the `daxpy()` function from BLAS [51] 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 4.2 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 on 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.

4.2.3 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 2.5, 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 `ptransCompute()` to be invoked in parallel from the Chapel program. The successful implementation of this test confirmed that the BRAID runtime allows functions to be invoked in parallel without introducing data races. As Table 4.2 shows, the BRAID-generated code introduces less than four percent overhead in the worst case. This 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 elements of the distributed array.

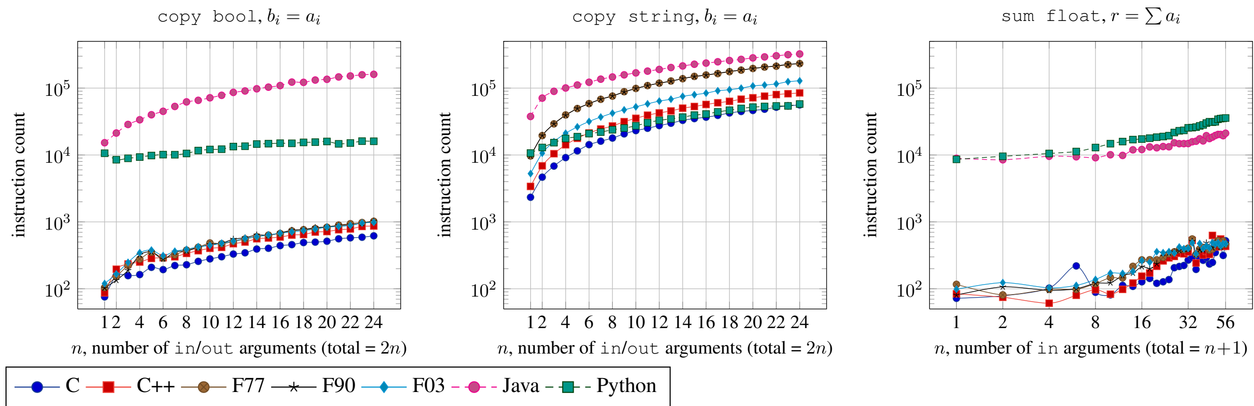

Figure 4.1: The cost of calling a function that copies or calculates the sum of n arguments, respectively

Nodes/locales	Pure execution time	Hybrid execution time	Overhead (in %)
2	62.22	74.67	20.01
4	208.20	229.05	10.01
6	332.94	390.82	17.38
8	473.10	526.08	11.20

Table 4.1: The `hpl` Benchmark, hybrid and pure Chapel versions execution times (in seconds) compared, input matrix is of size 50×50 with a block size of 8

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 4.2: 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

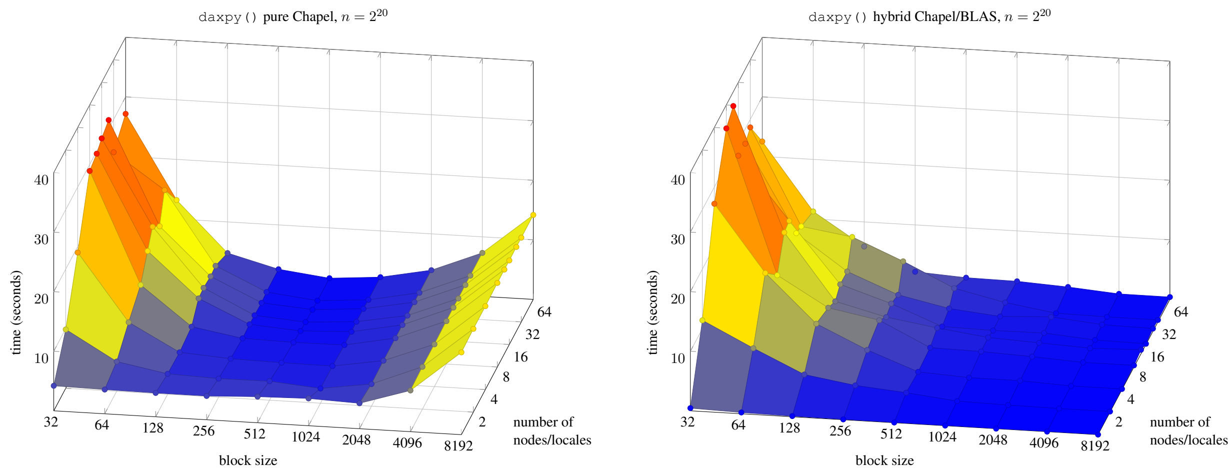

Figure 4.2: The `daxpy` Benchmark, hybrid and pure Chapel versions compared

	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 4.3: Features supported by BRAID's Chapel $\leftrightarrow$ Babel binding

4.3 Feature list

The complete list of features supported by the Chapel binding can be seen in Table 4.3. 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 a better performance than the route over C that is used by all the Babel backends.

5. Products

5.1 Publications and presentations

Results will be presented at the Fifth Partitioned Global Address Space Conference (PGAS 2011) [58] and in the poster session of SC'11 [57]. The following lists all project-related publications and presentations:

1. Adrian Prantl, Thomas G. W. Epperly, and Shams Imam
   Connecting PGAS and traditional HPC languages.
   Poster at the *SC'11 poster session*,
   Nov. 2011, Seattle, WA.
2. Adrian Prantl, Thomas G. W. Epperly, Shams Imam, and Vivek Sarkar
   Interfacing Chapel with traditional HPC languages.
   In *Fifth Partitioned Global Address Space Conference (PGAS 2011)*,
   Oct. 2011, Galveston Island, TX.
3. Shams Imam, Adrian Prantl, Thomas G. W. Epperly
   Connecting Chapel with traditional HPC languages.
   Poster at *LLNL Student Poster Symposium*,
   Aug. 2011, Livermore, CA.
4. Adrian Prantl, Thomas G. W. Epperly
   Reinventing Language Interoperability.
   Poster at *LLNL Postdoc Poster Symposium*,
   Jun. 2011, Livermore, CA.

5.2 Project Website

The project's official website is located at:

`https://computation.llnl.gov/casc/components/`

The source code to all of our interoperability work is kept in a repository together with the BRAID code generator. The repository can be accessed on SourceForge at:

`http://compose-hpc.sourceforge.net/`.

5.3 Software

BRAID is a library that was envisioned to be used to create interoperability applications. To streamline the distribution process, we decided to package the interoperability tool together with the library under the BRAID umbrella. BRAID's source code is available under on the COMPOSE-HPC SourceForge website listed above. A preview release of BRAID will be published as part of the forthcoming Babel 2.0 release.

5.4 Collaborations

We have had a summer intern, Shams Imam, who is a student of Vivek Sarkar at Rice University working on the project for two months in 2010 and returning again in 2011. In his first term he was creating hand-crafted examples for interoperability between Chapel, X10, UPC and C. In his second term he assisted with writing the Chapel glue code generator with BRAID, where he contributed a large part of the array implementation. Our collaboration with Rice University was very successful so far, and we intend to continue this in the future.

6. Conclusions

Given the uncertainty of how to program (post-)petascale machines and the diversity of programming models and languages that are exploring this problem, a general mixed parallel-language programming environment to bridge between them will be essential. Such an environment serves the immediate need of facilitating comparative studies to help resolve uncertainty and a long-term need of providing a migration path for legacy codes to be usable in new parallel paradigms. Our environment will also facilitate mixed parallelism models using conventional tools such as MPI and OpenMP, where different levels of parallelism can be targeted at different levels of the machine's hierarchy.

So for we successfully demonstrated the capabilities of our approach by creating Chapel binding for Babel. We did this be leveraging the BRAID code generator. Our language binding showcases high-performance interoperability between a PGAS/HPCS language and traditional HPC languages. It also provides two ways of integrating distributed (non-local) data types into the other languages: transparent, with no porting effort on the server side at all, or, through a SIDL interface that wraps the Chapel runtime for accessing remote array elements via GASnet in a way similar to the existing SIDL Array API.

This work lays the foundations for greater interoperability collaborations between the multiple communities represented by various PGAS/HPCS languages. It will be an important step in making new, powerful programming models practicable on tomorrow's massively parallel machines.

7. Bibliography

[1] Center for integrated space weather modeling (CISM) [online]. URL: http://www.bu.edu/cism.

[2] *OpenMP API Specification*, version 3.0 edition, May 2008. URL: http://openmp.org/wp/openmp-specifications.

[3] Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr. Project Fortress: A multicore language for multicore processors. *Linux Magazine*, September 2007.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc., March 2008.

[5] Rob Armstrong, Gary Kumfert, Lois Curfman McInnes, Steven Parker, Ben Allan, Matt Sottile, Thomas Epperly, and Tamara Dahlgren. The CCA component model for high-performance computing. *Intl. J. of Concurrency and Comput.: Practice and Experience*, 18(2), 2006.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yellick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, December 2006.

[7] Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., June 1994.

[8] Nathan Barton. Novel algorithms in computational materials science: Enabling adaptive sampling. invited talk, April 2008. URL: http://www.hpcsw.org/pi_meeting/presentations.

[9] Nathan R. Barton, Jaroslaw Knap, Athanasios Arsenlis, Richard Becker, Richard D. Hornung, and David R. Jefferson. Embedded polycrystal plasticity and adaptive sampling. *Int. J. Plasticity*, 24(2):242–266, February 2008. doi:10.1016/j.ijplas.2007.03.004.

[10] Joel V. Bernier, Nathan R. Barton, and Jaroslaw Knap. Polycrystal plasticity based predictions of strain localization in metal forming. *J. Eng. Mater. Technol*, 130(2), April 2008. doi:10.1115/1.2884331.

[11] Dan Bonachea. GASNet specification v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002. (newer versions also available at http://gasnet.cs.berkeley.edu).

[12] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Proc. of ACM/IEEE Supercomputing Conference*, page 12, Dallas, TX, 2000. ACM/IEEE Supercomputing Conference.

[13] William W. Carlson and Jesse M. Draper. Distributed data access in AC. *SIGPLAN Notices*, 30(8):39–47, August 1995. doi:http://doi.acm.org/10.1145/209937.209942.

[14] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

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

[16] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[17] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. doi:10.1177/1094342007078442.

[18] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in Chapel: Philosophy and framework. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association. URL: http://portal.acm.org/citation.cfm?id=1863086.1863098.

[19] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, October 2005. URL: http://dx.doi.org/10.1145/1094811.1094852, doi:10.1145/1094811.1094852.

[21] Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the International Conference of Supercomputing (ICS)*, 2003.

[22] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, St. Louis, MO, September 2005.

[23] Cray Inc., Seattle, WA. *Chapel Specification*, 0.4 edition, February 2005. URL: `http://chapel.cs.washington.edu`.

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

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

[26] Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel User's Guide*. Lawrence Livermore National Laboratory, July 2006. version 1.0.0.

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

[28] Steven J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, 2005.

[29] Roxana Diaconescu and Hans P. Zima. An approach to data distributions in Chapel. *to appear in this same special issue on High Productivity Programming Languages and Models of the International Journal of High Performance Computing Applications*, 2007.

[30] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform Co-Array Fortran compiler. In *Proceedings of the 13th Internaional Conference of Parallel Architectures and Compilation Techniques (PACT 2004)*, Antibes Juan-les-Pins, France, September 2004.

[31] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[32] Thomas G. W. Epperly, Gary Kumfert, Tamara Dahlgren, Dietmar Ebner, Jim Leek, Adrian Prantl, and Scott Kohn. High-performance language interoperability for scientific computing through babel. *IJHPCA*, (1094342011414036), 2011.

[33] Fortran 2008 working draft. Working Document of the J3 Standard Committee, June 2010. ISO/IEC JTC 1/SC 22/WG 5/N1830.

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

[35] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, November 1994.

[36] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference, volume 2*. Scientific and Engineering Computation. MIT Press, September 1998.

[37] Lydia Harper and Brian Kauffman. Community Climate System Model [online]. 2004. URL: `http://www.ccsm.ucar.edu`.

[38] Yun He and Chris H. Q. Ding. Coupling multicomponent models with mph on distributed memory computer architectures. *Int. J. High Perf. Comput. Appl.*, 19(3):329–240, 2005. `doi:10.1177/1094342005056118`.

[39] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, Spring–Summer 1993.

[40] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 2.0*, January 1997.

[41] Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15, Electrical Engineering and Computer Sciences, University of California at Berkeley, November 2005.

[42] HPCS: High productivity computer systems [online]. URL: `http://www.highproductivity.org`.

[43] Carl Kesselman. High performance parallel and distributed computation in compositional CC++. *SIGAPP Appl. Comput. Rev.*, 4(1):24–26, 1996. `doi:http://doi.acm.org/10.1145/240732.240741`.

[44] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, September 1996.

[45] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, New York, NY, USA, 1993. ACM. `doi:http://doi.acm.org/10.1145/169627.169724`.

[46] Manojkumar Krishnan, Yuri Alexeev, Theresa L Windus, and Jarek Nieplocha. Multilevel parallelism in computational chemistry using Common Component Architecture and Global Arrays. In *Proceedings of SuperComputing*. ACM and IEEE, 2005.

[47] Gary Kumfert, James Leek, and Thomas Epperly. Babel remote method invocation. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10, Long Beach, CA, March 2007. IEEE Computer Society. doi:10.1109/IPDPS.2007.370222.

[48] Jay Larson, Robert Jacob, and Everest Ong. The Model Coupling Toolkit: A new Fortran90 toolkit for building multi-physics parallel coupled models. *Int. J. High Perf. Comp. App.*, 19:277–292, 2005.

[49] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1188455.1188677, doi:http://doi.acm.org/10.1145/1188455.1188677.

[50] John May and David Jefferson. The cooperative parallelism project [online]. 2008. URL: https://computation.llnl.gov/casc/coopParallelism.

[51] National Science Foundation and Department of Energy. BLAS [online]. 2011. URL: http://www.netlib.org/blas/ [cited 2010-06-20 22:47:52].

[52] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In J. Rolim *et al.*, editor, *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, LNCS 1586, pages 533–546, San Juan, Puerto Rico, April 1999. Springer Verlag.

[53] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–291, 2006.

[54] Robert W. Numerich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[55] Robert W. Numerich and John Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[56] Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A. Kendall. A generalized portable SHMEM library for high performance computing. In M. Guizani and Z. Shen, editors, *the Twelfth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 401–406, Las Vegas, Nevada, November 2000.

[57] Adrian Prantl, Thomas G. W. Epperly, and Shams Imam. Connecting PGAS and traditional HPC languages.

[58] Adrian Prantl, Thomas G. W. Epperly, Shams Imam, and Vivek Sarkar. Interfacing Chapel with traditional HPC languages. In *Fifth Partitioned Global Address Space Conference (PGAS 2011)*.

[59] Craig Edward Rasmussen, Matthew J. Sottile, Sameer Shende, and Allen D. Malony. Bridging the language gap in scientific computing: the chasm approach. *Concurrency and Computation: Practice and Experience*, 18(2):151–162, 2006.

[60] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, volume 1*. Scientific and Engineering Computation. MIT Press, 2nd edition, September 1998.

[61] Lawrence Snyder. *The ZPL Programmer's Guide*. Scientific and Engineering Computation. MIT Press, March 1999.

[62] Jimmy Su and Katherine Yelick. Automatic support for irregular computations in a high-level language. In *19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[63] Alan Sussman. Building complex coupled physical simulations on the grid with InterComm. *Engineering with Computers*, 22:311–323, 2006.

[64] The UPC Consortium. *UPC Language Specification (v 1.2)*, June 2005. (available at http://upc.gwu.edu).

[65] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *19th Internaional Symposium on Computer Architecture*, pages 256–266, 1992.

[66] Katherine Yelick, Paul Hilfinger, Susan Graham, Dan Bonacha, Jimmy Su, Amir Kamil, Kaushik Datta, Phillip Colella, and Tong Wen. Parallel Languages and Compilers: Perspective from the Titanium Experience. *International Journal of High Performance Computing Applications*, 21(3):266–290, 2007. doi:10.1177/1094342007078449.

[67] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamato, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998.