Title
UPC++: A PGAS Extension for C++

ISBN
9780769552071

Authors
Zheng, Yili
Kamil, Amir
Driscoll, Michael B
et al.

Publication Date
2014-05-01

DOI
10.1109/ipdps.2014.115

Peer reviewed

UPC++: A PGAS Extension for C++

Yili Zheng*, Amir Kamil*, Michael B. Driscoll*†, Hongzhang Shan*, Katherine Yelick*†
*Lawrence Berkeley National Laboratory
†Department of EECS, University of California, Berkeley

***Abstract*—Partitioned Global Address Space (PGAS) languages are convenient for expressing algorithms with large, random-access data, and they have proven to provide high performance and scalability through lightweight one-sided communication and locality control. While very convenient for moving data around the system, PGAS languages have taken different views on the model of computation, with the static Single Program Multiple Data (SPMD) model providing the best scalability. In this paper we present UPC++, a PGAS extension for C++ that has three main objectives: 1) to provide an object-oriented PGAS programming model in the context of the popular C++ language; 2) to add useful parallel programming idioms unavailable in UPC, such as asynchronous remote function invocation and multidimensional arrays, to support complex scientific applications; 3) to offer an easy on-ramp to PGAS programming through interoperability with other existing parallel programming systems (e.g., MPI, OpenMP, CUDA).**

We implement UPC++ with a "compiler-free" approach using C++ templates and runtime libraries. We borrow heavily from previous PGAS languages and describe the design decisions that led to this particular set of language features, providing significantly more expressiveness than UPC with very similar performance characteristics. We evaluate the programmability and performance of UPC++ using five benchmarks on two representative supercomputers, demonstrating that UPC++ can deliver excellent performance at large scale up to 32K cores while offering PGAS productivity features to C++ applications.

I. Introduction

Many recent parallel programming languages adopt a partitioned global address space (PGAS) abstraction for managing data and communication on distributed-memory systems. Popular examples include Chapel, Co-Array Fortran, X10, and UPC. Despite the prosperity of the PGAS language family, prior work does not include a PGAS framework based on the C++ language. One of the hurdles to providing a PGAS extension for C++ is the significant cost required to craft a robust C++ compiler front-end with PGAS support, due to the enormous complexity of parsing C++. The other big challenge is determining the interplay between PGAS and C++ semantics. At the same time, there is an increasing demand for C++ in computational science and engineering applications. To fill this void, we design and implement UPC++, a library-based PGAS extension for C++.

Learning from many years of experience as both implementors and users of UPC, Titanium, and other PGAS languages, we distill the most important programming constructs from these languages and incorporate them into UPC++. We start with a basic design that allows existing UPC applications to be ported to UPC++ with minimal syntactic changes. In addition, we augment the parallel programming features in UPC with asynchronous remote function invocation and multidimensional domains and arrays, both of which are important for writing productive and efficient code for emerging computer architectures.

While there may be potential advantages to implementing a UPC++ compiler, such as static type checking and program analysis, we choose to implement UPC++ with a "compiler-free" approach, using C++ templates and runtime libraries to provide PGAS features. We use the library approach for several reasons. First, recent studies [1], [2], [3] show that major performance gains in UPC applications mostly result from runtime or manual optimizations rather than automatic compiler optimizations; insufficient data dependency information at compile-time often prohibits the compiler from performing important optimizations. In these common cases, the library approach provides similar performance as a compiler. Second, C++ generic programming and template specialization features provide sufficient mechanisms to implement syntactic sugar for PGAS programming idioms. Modern C++ compilers such as Clang are also capable of providing clear diagnosis information for the template types used in UPC++. In addition, a "compiler-free" approach enjoys many other benefits including better portability across platforms, more flexible interoperability with other parallel language extensions, such as OpenMP and CUDA, and significant savings in development and maintenance costs.

The main contributions of our work are:

- the design and a prototype implementation of UPC++
- case studies of UPC++ benchmarks demonstrating that the "compiler-free" PGAS approach can deliver the same scalable performance as PGAS programming languages, such as UPC and Titanium

In the rest of this paper, we start by reviewing the background of UPC++ and related work. We then describe the syntax and semantics of UPC++ and briefly discuss the implementation strategy. We present five case studies of writing UPC++ benchmarks and applications, demonstrating the productivity and performance of UPC++. Finally, we conclude with lessons learned from our experience implementing and using UPC++.

II. Background and Related work

UPC [4] is a parallel programming language extension for C that provides a global address space abstraction for memory management and data communication on shared-memory and distributed-memory systems. In UPC, all variables are in the private address space by default; a programmer must explicitly

declare a variable as `shared`. UPC has been shown to scale well on large systems ([5], [3]), but some of its limitations have also been revealed over the years. Programming language problems in UPC are hard to fix without breaking backwards compatibility. As a result, we have decided to start UPC++ from a clean slate, enabling us to freely enhance UPC while keeping the most popular features.

The execution model of UPC is SPMD and each independent execution unit is called a *thread*, which can be implemented as an OS process or a Pthread. Because the number of UPC threads is fixed during program execution, it is inconvenient to express the dynamic and irregular parallelism required by emerging data-intensive applications and runtime hardware changes.

Both MPI-3 RMA and Global Arrays [6] provide comprehensive support for one-sided communication functions, but they are not specialized for C++. In contrast, UPC++ takes advantage of unique C++ language features, such as templates, object-oriented design, operator overloading, and lambda functions (in C++ 11) to provide advanced PGAS features not feasible in C. For example, UPC++ programs can use assignment statements to express reads and writes to `shared` variables or use `async` to execute a code block (expressed as a lambda function) on a remote node.

Many recent PGAS languages, such as Chapel [7] and X10 [8], support asynchronous task execution using a fork-join execution model. UPC++ brings the `async` feature to the SPMD execution model on distributed-memory systems in the same way as the C++11 standard async library for shared-memory systems. Intranode programming models such as Cilk [9], OpenMP, and Threading Building Blocks [10] also have programming idioms for dynamic tasking, but they require hardware shared-memory support. As with other PGAS languages, UPC++ works on both shared-memory and distributed-memory systems.

There exist extensive research studies about using C++ libraries for parallel programming, including but not limited to Hierarchically Tiled Arrays [11], C++ Coarrays [12], [2], STAPL [13], Charm++ [14], HPX [15], and Phalanx [16]. UPC++ uniquely combines and adapts popular parallel programming constructs from several successful programming systems (UPC, Titanium [17], Phalanx and X10). For example, UPC++ includes a high-level multidimensional array package for regular numerical applications but also allows users to build irregular data structures using low-level mechanisms such as global pointers and dynamic remote memory allocation.

III. Programming Constructs

In this section, we describe the parallel programming constructs provided by UPC++, which include shared objects, dynamic global memory management, multidimensional domains and arrays, and asynchronous remote function invocation.

Table I summarizes the main UPC programming idioms and their UPC++ equivalents. All UPC++ extensions are packaged in the `upcxx` namespace to avoid naming conflicts with other libraries. For brevity, the code examples in this paper assume that the `upcxx` namespace is being used.

A. Shared Objects

In UPC, the `shared` type qualifier is used to declare shared objects in the global address space. There are two types of shared objects: shared scalars and shared arrays. A shared scalar is a single memory location, generally stored on thread 0 but accessible by all threads. A shared array is an array distributed across all threads in a one dimensional block-cyclic layout. Shared scalars and arrays are implemented in UPC++ using two different parametric types (template classes), `shared_var` and `shared_array`, since they require different implementation strategies.

Shared scalar variables in UPC++ can be declared by using the `shared_var` template:

```
shared_var<Type> s;
```

Any thread can read or write a shared variable directly, as in the following example:

```
s = 1; // using shared var as an lvalue
int a = s; // using shared var as a rvalue
```

Shared arrays in UPC++ are declared by using the `shared_array` template:

```
shared_array<Type, BS> sa(Size);
```

where `T` is the element type and `BS` is the block size for distributing the array, with a default value of 1 as in UPC (cyclic distribution). As with shared scalars, the user may directly access shared array elements. The subscript operator `[]` is overloaded to provide the same interface as non-shared arrays:

```
sa[0] = 1;
cout << sa[0];
```

A UPC++ `shared_array` can also be initialized with a dynamic size at runtime (e.g., `sa.init(THREADS)`), which allocates block-cyclically distributed global storage space similar to `upc_all_alloc`.

B. Global Pointers

UPC++ provides the PGAS model using the generic `global_ptr` type to reference shared objects in the global address space. A global pointer encapsulates both the thread ID and the local address of the shared object referenced by the pointer. Operator overloading is used to provide the semantics of pointers in the global address space. The global pointer template has the following form:

```
global_ptr<Type> sp;
```

Template specialization for the global void pointer type (`global_ptr<void>`) allows type-casting to and from other global pointer types. In addition, casting a global pointer to a regular C++ pointer results in the local address of a shared object, as in the following example:

```
global_ptr<void> sp;
void *lp = (void *)sp;
```

Programming Idioms	UPC	UPC++
Number of execution units	`THREADS`	`THREADS` or `ranks()`
My ID	`MYTHREAD`	`MYTHREAD` or `myrank()`
Shared variable	`shared Type v`	`shared_var<Type> v`
Shared array	`shared [BS] Type A[size]`	`shared_array<Type, BS>` A(size)
Global pointer	`shared Type *p`	`global_ptr<Type> p`
Memory allocation	`upc_alloc(...)`	`allocate<Type>(...)`
Data movement	`upc_mempcy(...)`	`copy<Type>(...)`
Synchronization	`upc_barrier` & `upc_fence`	`barrier()` & `fence()`
Forall loops	`upc_forall(...; affinity_cond) { stmts; }`	`for(...) { if (affinity_cond){ stmts } }`

TABLE I
SUMMARY OF UPC++ EQUIVALENTS FOR UPC

The `where()` member function of a global pointer queries the location of the object referenced by the global pointer (the thread affinity in UPC terminology).

One intentional difference between UPC and UPC++ global pointers is that the latter do not contain the block offset (or *phase*) when referencing an element in a shared array. As a result, pointer arithmetic with global pointers in UPC++ works the same way as arithmetic on regular C++ pointers. This simplification has two advantages: 1) it results in a more compact global pointer representation; 2) it simplifies pointer arithmetic and avoids common confusion about the pointer phase in UPC. Users who prefer the UPC pointer semantics for shared arrays can use UPC++ `shared_array` index arithmetic (e.g. `A[i+j]`) instead, which achieves the same goal with less ambiguity.

C. Dynamic Memory Management

Memory can be allocated in the global address space by

```
global_ptr<T> allocate<T>(int rank, size_t sz);
```

where `rank` is the thread id on which to allocate memory and `sz` is the number of elements for which to allocate memory. The following example allocates space for 64 integers on thread 2:

```
global_ptr<int> sp = allocate<int>(2, 64);
```

With `allocate`, UPC++ supports allocating memory on local or remote threads, a feature that is not available in either UPC or MPI but is very useful in creating distributed data structures. For example, when inserting an element into a distributed linked list, it may be necessary to allocate memory for the new element at the thread that owns the insertion point of the list. Dynamically allocated memory can be freed by calling the `deallocate` function from any UPC++ thread.

Note that the `allocate` function doesn't call the object's constructor implicitly. If object initialization is necessary, the user may invoke the object constructor by using *placement new* with the memory region returned by `allocate`.

In addition to the global memory allocator, UPC++ also allows users to construct a `global_ptr` from a regular C++ pointer to a local heap or stack object, which semantically *escalates* a private object into a shared object. This feature requires the underlying communication runtime to support network access to the whole memory space, as in the *segment everything* configuration of GASNet.

D. Bulk Data Transfer Functions

Modern networks usually move a large chunk of data more efficiently than smaller pieces of data. To take advantage of this, UPC++ provides a `copy` function for bulk data transfers,

```
copy(global_ptr<T> src, global_ptr<T> dst,
     size_t count);
```

where the `src` and `dst` buffers are assumed to be contiguous.

A non-blocking copy function is also provided to enable overlapping communication with computation or other communication, along with a synchronization function to wait for completion of all non-blocking copies:

```
async_copy(global_ptr<T> src,
           global_ptr<T> dst,
           size_t count);

// Synchronize all previous async_copy's
async_copy_fence();
```

Finally, the user may register an `async_copy` operation with an `event` (similar to a `MPI_Request`) for synchronizing individual operations later.

E. Multidimensional Domains and Arrays

A major limitation of UPC shared arrays is that they can only be distributed in a single dimension. Users that require multidimensional arrays distributed across more than one dimension must build them on top of shared arrays, either by manually linearizing the multidimensional array into a 1D shared array or by building a directory structure with global pointers to the individual pieces on each thread. Furthermore, support for multidimensional arrays is limited in C and C++, as the sizes of all but the first dimension must be compile-time constants. These sizes are part of the array type, which makes it difficult to write generic code using multidimensional arrays.

In addition to manually building a distributed multidimensional array, the user must translate array indices from the logical index space to the physical index space and vice versa. Operations that require logically but not physically contiguous data, such as slicing a 3D array to obtain a 2D ghost zone,

must be manually handled by the user. Copying a ghost zone from one thread to another requires a complicated pack operation on the source, a bulk transfer, and a complicated unpack operation on the destination. This results in a two-sided operation, negating the benefits of the one-sided PGAS model.

In order to address these limitations, UPC++ includes a multidimensional domain and array library based on that of Titanium. Titanium's library is similar to Chapel's dense and strided domains, as both were inspired by the dense and strided regions and arrays in ZPL [18]. The UPC++ domain and array library includes the following components:

- *points* are coordinates in N-dimensional space
- *rectangular domains* consist of a lower-bound point, an upper-bound point, and a stride point
- arrays are constructed over a rectangular domain and indexed by points

Templates are used to provide classes for arbitrary N, and in the case of arrays, for arbitrary element type.

Since Titanium has its own compiler, it can provide its own syntax for points, rectangular domains, and arrays. UPC++, on the other hand cannot do so. This makes constructing such objects very verbose in UPC++, such as the following expression for a three dimensional rectangular domain:

```
rectdomain<3>((point<3>) {1, 2, 3},
              (point<3>) {5, 6, 7},
              (point<3>) {1, 1, 2})
```

However, using macros, UPC++ can also provide convenient shorthand for creating points, domains, and arrays. For example, the following macro expression is equivalent to the full expression above:

```
RECTDOMAIN((1, 2, 3), (5, 6, 7), (1, 1, 2))
```

Table II compares Titanium syntax with the macros provided by UPC++, which result in expressions nearly as compact as in Titanium[1]. A user can further compact expressions by defining shorter macros for `POINT`, `RECTDOMAIN`, and `ARRAY`. Operator overloading is used to provide arithmetic operations over points and domains and to index arrays with points.

As part of the language, Titanium provides unordered iteration over any domain in the form of a `foreach` loop. The user specifies a variable name and a domain, and in each iteration of the loop, the name is bound to a different point in the domain. Unlike a `upc_forall` loop, the iterations occur sequentially on the thread that executes the loop, allowing a user to iterate over a multidimensional domain with only a single loop. UPC++ provides a `foreach` macro with similar syntax as Titanium, as shown in Table II.

Arrays may be constructed over any rectangular domain, so that an array's index space matches the logical domain. As in Titanium, UPC++ arrays also allow different views to be created of the same underlying data. For example, an array may be restricted to a smaller domain, such as to obtain a view of the interior of a grid that has ghost zones. An array may also be sliced to obtain an $(N-1)$-dimensional view of an N-dimensional array. Many other reinterpreting operations are provided, such as permuting dimensions and translating the domain of an array.

The elements of an array must be located on a single thread, which may be in a remote memory location. The array index operator is overloaded to retrieve data from a remote location if necessary. Furthermore, the library provides a copy operation, invoked as `A.copy(B)`, which copies data from array `B` to array `A`. The two arrays need not have affinity to the same thread, and their underlying domains need not be equal. The library automatically computes the intersection of their domains, obtains the subset of the source array restricted to that intersection, packs elements if necessary, sends the data to the processor that owns the destination, and copies the data to the destination array, unpacking if necessary. The entire operation is one-sided, with active messages performing remote operations. Copying a ghost zone requires the single statement

```
A.constrict(ghost_domain_i).copy(B);
```

This is a vast improvement over the UPC model, which requires the user to perform all the work.

The current multidimensional array library is limited in that an array cannot be distributed over multiple threads. The user must use a directory structure to store pointers to the individual pieces on each thread. In fact, multidimensional arrays can be composed with shared arrays to build such a directory, as in the following:

```
shared_array< ndarray<int, 3> > dir(THREADS);
void init() {
  dir[MYTHREAD] = ARRAY(int, ...);
  ...
}
```

Unlike in UPC, the UPC++ multidimensional array library allows the individual pieces to be multidimensional with runtime-specified sizes and index spaces corresponding to the logical domains.

An advantage of the library approach to UPC++ is that the array library can be extended and new array libraries can be added at any time to provide new features or meet specialized needs. For example, we have implemented template specializations for arrays that have a matching logical and physical stride, avoiding expensive stride calculations when indexing into such an array. The combination of macros, templates, and operator overloading allows array libraries to provide the same generality and a similar interface as arrays built into a language. In the future, we plan to take further advantage of this capability by building true distributed multidimensional arrays on top of the current non-distributed library.

[1]For UPC++, we chose to use an exclusive upper bound rather than the inclusive upper bound used in Titanium. Thus, the upper bounds in the UPC++ examples are one greater in each dimension than those in the Titanium examples.

Domain/Array Syntax	Titanium	UPC++
Point literals	`[1, 2]`, `[1, 2, 3]`	`POINT(1, 2)`, `POINT(1, 2, 3)`
Rectangular domains[1]	`[[1, 2] : [8, 8] : [1, 3]]`	`RECTDOMAIN((1, 2), (9, 9), (1, 3))`
Domain arithmetic	`rd1 + rd2`, `rd1 * rd2`, etc.	`rd1 + rd2`, `rd1 * rd2`, etc.
Array literals[1]	`new int[[1, 2] : [8, 8] : [1, 3]]`	`ARRAY(int, ((1, 2), (9, 9), (1, 3)))`
Array indexing	`array[pt]`	`array[pt]` or `array[pt[1]]...[pt[N]]`
Iteration	`foreach (p in dom) ...`	`foreach (p, dom) ...`

TABLE II
COMPARISON OF TITANIUM AND UPC++ DOMAIN AND ARRAY SYNTAX

F. Memory Consistency Model and Synchronization

UPC++ uses a relaxed memory consistency model similar to that of UPC [4]. Remote memory operations can be reordered by the underlying runtime or hardware as long as there are no data dependencies. Only memory operations *issued from the same thread and accessing the same memory location* are required to execute in program order. Memory operations issued from different threads can be executed in arbitrary order unless explicit synchronization is specified by the programmer.

Both UPC++ and UPC provide synchronization primitives, such as barriers, fences, and locks, to help programmers write correct parallel programs. The only syntactic distinction is that these primitives are language keywords in UPC while they are functions or macros in UPC++ (see Table I). However, because the underlying implementations of these synchronization primitives are usually the same for both UPC and UPC++, there is no observable performance difference between UPC and UPC++ synchronization operations.

G. Remote Function Invocation

Another feature provided by UPC++ but not by UPC is remote function invocation. UPC++ remote function invocation was inspired by Phalanx, X10, and the async library in C++11. The user may start an asynchronous remote function invocation with the following syntax:

```
future<T> f = async(place)(function, args...);
```

where `place` can be a single thread ID or a group of threads. A UPC++ async call optionally returns a `future` object (requires C++11), which can be used to retrieve the return value of the remote function call by `future.get()`.

```
// Example: call a lambda function on Thread 2
async(2)([] (int n) {printf("n: %d", n);}, 5);
```

In addition, UPC++ provides two programming constructs for specifying dynamic dependencies among tasks: 1) *event-driven* execution as in Phalanx; 2) *finish-async* as in X10. In the event-driven execution model, events are used to specify dependencies between tasks. The user may register `async` operations with an `event` to be signaled after task completion. An `event` can be signaled by one or more async operations, and used as a precondition to launch later `async` operations.

```
// signal event ''ack'' after task1 completes
async(place, event *ack)(task1, args);

// launch task2 after event ''after''
async_after(place, event *after)(task2, args);
```

Listing 1. An Example of Building a Task Dependency Graph

```
event e1, e2, e3;
async(p1, &e1)(t1);
async(p2, &e1)(t2);
async_after(p3, &e1, &e2)(t3);
async(p4, &e2)(t4);
async_after(p5, &e2, &e3)(t5);
async_after(p6, &e2, &e3)(t6);
e3.wait();
```

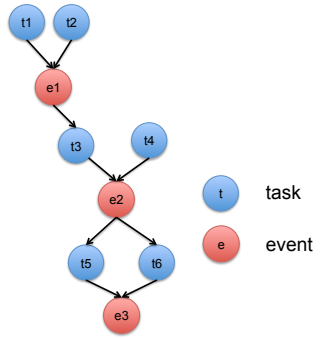

Fig. 1. Task dependency graph created by Listing 1

In X10, the `finish` construct defines a scope that prevents the program from proceeding past the construct until all asyncs spawned within the construct complete. UPC++ provides a similar `finish` construct, as in the following example:

```
finish {
  async(p1)(task1);
  async(p2)(task2);
}
```

Both `task1` and `task2` must complete before the program can exit the `finish` block. Because UPC++ is implemented as a library, we leverage C++ macros and the *Resource Allocation is Initialization (RAII)* pattern to implement the `finish` construct. The following is a simplified implementation of `finish` that illustrates the mechanism we use:

```
#define finish for (finish_scope _fs; \
                    _fs.done == 0; _fs.done = 1)
#define async(p) _async(p, _fs)
```

The C++ preprocessor translates a `finish` code block into code similar to the following:

```
for (finish_scope _fs; _fs.done==0; _fs.done=1)
{
```

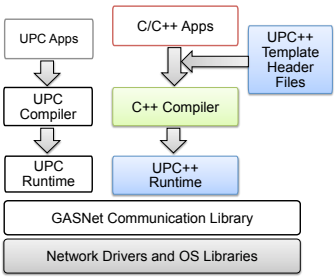

Fig. 2. UPC++ vs. UPC implementation stack

```
  _async(p1, _fs)(task1);
  _async(p2, _fs)(task2);
} // _fs.~finish_scope() is called
```

In the translated code, a `finish_scope` object is created at the beginning the `finish` construct. Async tasks spawned inside the finish block register themselves with the `finish_scope` object `_fs`, which tracks the number of outstanding tasks. This counter is decremented each time an async task spawned in the finish block completes. Since the `finish_scope` object is created with automatic storage duration, the C++ specification requires that its destructor be automatically called when it goes out of scope at the end of the finish block. The destructor is defined to wait for the completion all async activities started in the current finish scope, ensuring that all such activities complete before proceeding from the finish block.

There are a handful of differences between asyncs in X10 and UPC++. The X10 compiler determines which objects are reachable from the async task, arranging for all such objects to be serialized and transferred to the remote node, which may be an expensive operation. In contrast, UPC++ does not capture the closure of an async task, serializing and transferring only the function arguments. Another difference between X10 and UPC++ is that finish in X10 waits for all activities transitively spawned by the finish block to complete before proceeding, whereas UPC++ only waits for those spawned in the dynamic scope of the finish block. We made this design decision because termination detection for unbounded async tasks is a very expensive operation on distributed-memory systems.

IV. Implementation

For the reasons mentioned earlier in Section I, we take a "compiler-free" approach to implement UPC++, using a collection of C++ templates and runtime libraries without modifying the C++ compiler. In this section, we provide an overview of our implementation techniques for UPC++.

Figure 2 shows the software layers for UPC++ in comparison to the Berkeley UPC compiler. The UPC++ front-end consists of a set of C++ header files that enables the C++ compiler to "translate" UPC++ features in a user program to runtime library calls. In particular, we use *templates* and *operator overloading* to customize the behavior of UPC++ types. Figure 3 illustrates how UPC++ converts an assignment statement on a shared array element to runtime function calls,

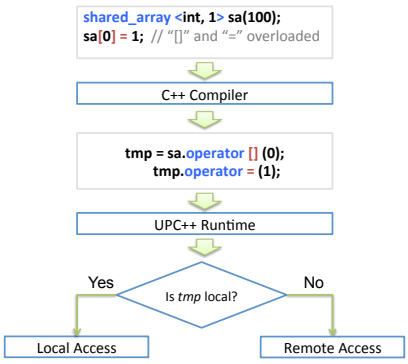

Fig. 3. UPC++ translation and execution flow example

as well as how the runtime handles data access based on the location of the element.

A UPC++ shared object has a local proxy object on each thread that references it. An access to a shared object is compiled to an access to the corresponding local proxy. At runtime, a proxy object access results in communication with the owner of the shared object.

Async remote function invocation is implemented by active messages [19] and a task queue, with syntactic sugar provided by C++ function templates and functor objects. An async call `async(place)(function, args...)` is handled in two steps: `async(place)` first constructs an internal async object that has a customized "`()`" operator for handling the arguments in the second pair of parentheses. Next, UPC++ uses helper function templates to pack the task function pointer and its arguments into a contiguous buffer and then sends it to the target node with an active message. Upon receiving the message, the remote node unpacks the async task object and inserts it into a task queue for later execution. Finally, enqueued async tasks are processed when the `advance()` function in the UPC++ runtime is called by the user program or by a worker Pthread. After an async invocation is completed, the runtime sends a reply message to the requester with the return value of the function. We assume that the function entry points on all processes are either all identical or have an offset that can be collected at program loading time.

Each UPC++ rank is implemented as an OS process and can interoperate with MPI with a straightforward one-to-one mapping between UPC++ and MPI ranks. The UPC++ runtime has two thread-support modes: 1) serialized mode, in which the application is responsible for serializing the calls to UPC++ functions (similar to `MPI_THREAD_SERIALIZED`). 2) concurrent mode, in which the UPC++ runtime provides the thread-safety of UPC++ functions (similar to `MPI_THREAD_MULTIPLE`). The serialized mode is sufficient for many common cases of multi-threading, such as using OpenMP to parallelize `for` loops or calling multi-threaded math libraries for computation-intensive regions. Because UPC++ doesn't employ thread-safety protection mechanisms (e.g., mutex and atomic operations) in the serialized mode, it has less performance overheads than the concurrent mode.

Benchmark	Description	Computation	Communication
Random Access	Measure throughput of random memory accesses	bit-xor operations	global fine-grained random accesses
Stencil	3D 7-point stencil modeling heat equation	nearest-neighbor computation	bulk ghost zone copies
Sample Sort	Sort a distributed array of integers	local quick sort	irregular one-sided communication
Embree	Ray-tracing code	Monte Carlo integration	single gatherv
LULESH [20]	Shock hydrodynamics simulation	Lagrange leapfrog algorithm	nearest-neighbor communication

TABLE III
SUMMARY OF BENCHMARK CHARACTERISTICS

V. CASE STUDIES

To better understand the usability and obtainable performance of UPC++, we studied five common cases from anticipated usage scenarios:

- port the UPC Random Access benchmark to UPC++
- port the Titanium Stencil benchmark to UPC++
- implement Sample Sort in UPC++ and compare to the same algorithm implemented in UPC
- parallelize a large existing C++ application (Embree) for distributed-memory systems
- port an MPI C++ proxy application (LULESH) to UPC++

Table III summarizes the computation and communication characteristics of the five benchmarks we used. For each case, we discuss the programming effort and present the performance results.

We evaluated our benchmarks on two supercomputer platforms, Edison and Vesta. Edison at NERSC is a Cray Cascade system (XC-30) with Intel Ivy Bridge CPUs and an Aries interconnect with DragonFly topology. Vesta at ALCF is an IBM BG/Q system with 16-core customized PowerPC A2 chips and a 5D torus network. In all our performance experiments, each UPC or UPC++ thread is mapped to an OS process. We used Berkeley UPC 2.18 and vendor C/C++ compilers to build our benchmarks, with the exception of Stencil, where we used GCC to build both the Titanium and UPC++ versions.

A. Random Access (GUPS)

The Random Access benchmark measures the throughput of random updates to global memory by the giga-updates-per-second metric. Since the Random Access code can be concisely expressed using a globally shared array, it is a classical PGAS example and is often used to predict the performance of applications with irregular data accesses. The main update loop is as follows:

```
// shared uint64_t Table[TableSize]; in UPC
shared_array<uint64_t> Table(TableSize);

void RandomAccessUpdate() {
  ...
  for(i=MYTHREAD; i<NUPDATE; i+=THREADS) {
    ran = (ran<<1)^((int64_t)ran<0?POLY:0);
    Table[ran & (TableSize-1)] ^= ran;
  }
}
```

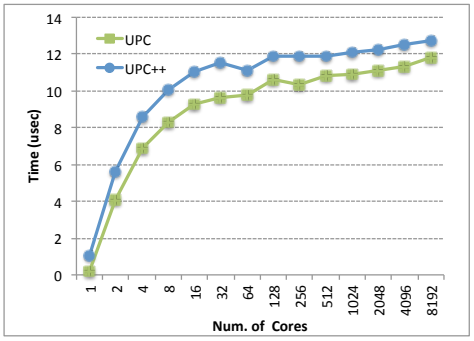

Fig. 4. Random Access latency per update on IBM BlueGene/Q

THREADS	16	128	1024	8192
UPC	0.0017	0.012	0.094	0.69
UPC++	0.0014	0.0108	0.084	0.64

TABLE IV
RANDOM ACCESS GIGA-UPDATES-PER-SECOND

Porting the UPC version of GUPS to UPC++ is straightforward as it only requires minimal syntactic changes (from `shared []` in UPC to `shared_array` in UPC++).

The UPC version of GUPS outperforms the UPC++ version at small scales (10% better at 128 cores) as the Berkeley UPC compiler and runtime are heavily optimized for shared array accesses. However, when scaling to a large number of processors, the network access latency and throughput start to dominate the runtime, so that the performance gap between UPC and UPC++ decreases to a very small percentage of the total runtime, as shown in Figure 4 and Table IV.

The Random Access benchmark represents the worst case data movement scenario, in which there is no data locality at all. In our experience, most scalable parallel applications fetch data in large chunks, avoiding random fine-grained access when possible. Since bulk data transfer functions (e.g., `upc_memcpy`) are completely implemented in libraries, we expect the performance overhead of UPC++ to be negligible compared to UPC in more realistic cases.

B. Stencil

The Stencil benchmark performs a nearest-neighbor computation over a regular three-dimensional grid. In each iteration, the new value of a grid point is computed from its old value and those of its neighboring points. In this benchmark, we use

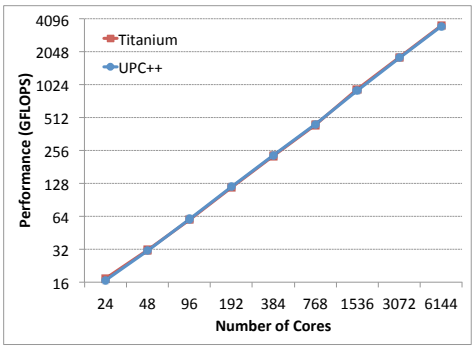

Fig. 5. Stencil weak scaling performance (GFLOPS) on Cray XC30

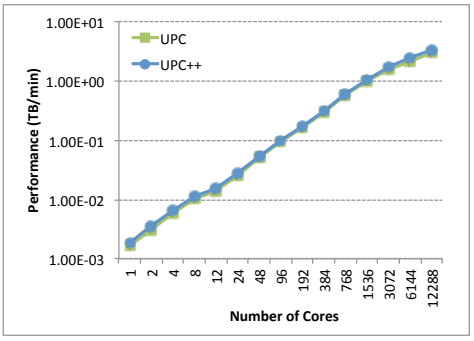

Fig. 6. Sample Sort weak scaling performance (TB/min) on Cray XC30

one neighboring point in each direction, resulting in a seven-point stencil, and the Jacobi (out-of-place) iteration strategy. The overall grid is distributed in all three dimensions such that each thread has a fixed 256^3 local portion of the grid, or 258^3 including ghost zones and boundary conditions. Each local grid is represented by a multidimensional array, which allows views of the interior of the local grid and its ghost cells to be created without any copying.

Porting the Titanium implementation to UPC++ requires few changes outside of basic syntax, since UPC++ multidimensional domains and arrays are based on Titanium's domains and arrays. We declare the grid arrays to have a matching physical and logical stride, bypassing expensive stride calculations. In addition, we index into the arrays one dimension at a time rather than with points, allowing the C++ compiler to lift most of the indexing logic out of the inner loop. The resulting computation is as follows:

```
ndarray<double, 3, unstrided> A, B;
...
// Local stencil computation
foreach3 (i, j, k, interiorDomain) {
  B[i][j][k] = c * A[i][j][k] +
    A[i][j][k+1] + A[i][j][k-1] +
    A[i][j+1][k] + A[i][j-1][k] +
    A[i+1][j][k] + A[i-1][j][k];
}
```

Copying a ghost zone requires only a single call, as described in §III-E. Thus, UPC++ multidimensional arrays provide a high level of productivity for both local computation and communication in this benchmark.

Figure 5 shows the performance of Stencil on the Cray XC30 in billions of floating-point operations per second (GFLOPS). Performance scales very well from 24 cores (1 node) to 6144 cores (512 nodes). UPC++ performance is nearly equivalent to Titanium performance, demonstrating that we can provide the same performance in a library as in a full language.

C. Sample Sort

The Sample Sort benchmark sorts a large array of 64-bit integer keys using a variant of the sample sort algorithm [21]. The keys are generated by the Mersenne Twister random number generator and stored in a shared array. The algorithm samples the keys to compute splitters, redistributes the keys based on those splitters, and then does a final local sort. The UPC++ Sample Sort code leverages the PGAS abstraction to communicate the key samples and uses non-blocking one-sided communication to redistribute the keys after partitioning them. The following code demonstrates the sampling portion of the algorithm:

```
shared_array<uint64_t> keys(key_count);
...
// Sample the key space to find the splitters
for (i = 0; i < candidate_count; i++) {
  uint64_t s = genrand_uint64() % key_count;
  candidates[i] = keys[s]; // global accesses
}
```

Figure 6 shows the performance of Sample Sort in terabytes sorted per minute (TB/min). On 12288 cores, UPC++ Sample Sort achieves 3.39 TB/min. Even though the benchmark is communication-bound, its performance scales reasonably well on Edison as a result of the RDMA support in the network hardware and the Dragonfly network topology. The performance of UPC++ is nearly identical to the UPC version of the benchmark.

D. Embree (Ray Tracing)

Embree is an open-source collection of vectorized kernels for ray tracing on multi-core and many-core architectures. The distribution includes a sample renderer, written in C++, that solves the rendering equation via Monte Carlo integration. The sample renderer supports multi-bounce light paths, area lights, realistic materials, and high-dynamic range imaging.

We extend the sample renderer to target distributed-memory machines. The original code divides the image plane into tiles that are processed on demand by worker threads. We remove the custom load balancer in favor of a static, cyclic tile distribution among UPC++ threads. Within each UPC++ thread, we use OpenMP with dynamic scheduling to balance the evaluation of the tiles. A final gather operation combines the tiles into the final image, but our implementation uses a simpler reduction to add the partial images. (The performance cost of this compromise is negligible at small scale.) Our initial implementation assumes that scene geometry is small enough to be replicated on every UPC++ thread.

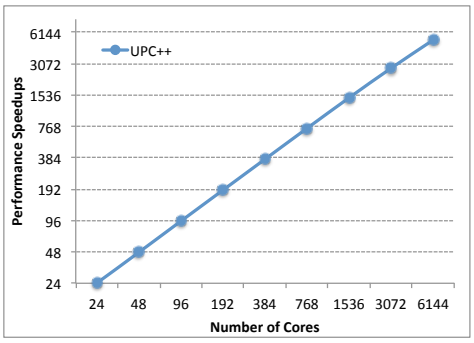

Fig. 7. Embree ray tracing strong scaling performance on Cray XC30

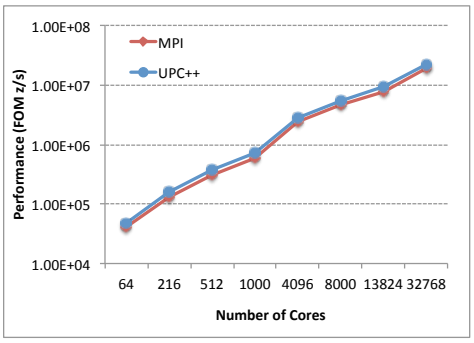

Fig. 8. LULESH weak scaling performance on Cray XC30; the number of processes is required to be a perfect cube by the application

Figure 7 shows that our renderer achieves nearly perfect strong scaling on Edison. This is of little surprise since the application is mostly embarrassingly parallel. However, this case study illustrates the ease with which an existing C++ application can be ported to distributed-memory platforms with UPC++. The performance advantage readily translates into higher-fidelity images for scientists and artists. Our implementation also demonstrates the natural composability of "compiler-free" PGAS constructs with existing frameworks for parallelism like OpenMP. A previous attempt at writing a communication library in UPC (to be linked to Embree) was abandoned because of the difficulty of composition at the object-code level.

In the future, we hope to improve performance by implementing global load balancing via distributed work queues and work stealing. Others have found PGAS a natural paradigm for implementing such schemes [22], so we expect this to be straightforward. In addition, for scales where the sum-reduction is too costly, we plan to overlap the computation and gathering of the tiles using a combination of one-sided writes and gather operations.

E. LULESH

LULESH [20] is a shock hydrodynamics proxy application written in C++ with support of both MPI and OpenMP. Each processor needs to communicate with 26 of its neighbors in the 3D domain. Since array elements on each processor are laid out in row-major order, the data that need to be transferred are not contiguous in two of the dimensions. The LULESH code uses a packing and unpacking strategy and overlaps computation with communication. The MPI version of LULESH uses non-blocking two-sided message passing (`MPI_Isend` and `MPI_Irecv`) for communication, while the UPC++ version replaces the two-sided communication operations with one-sided communication.

To explore the communication effects at large scale, which we expect will be a critical performance factor in the near future, we use MPI and UPC++ without multi-threading. Since the LULESH code restricts the number of processes to a perfect cube (i.e. n^3 for integer n) for equal partitioning of the 3-D data domain, which isn't always a multiple of the number of cores per node (24), we attempt to lay out processes onto different nodes as evenly as possible.

UPC++ and the underlying GASNet runtime support "handle-less" non-blocking communication, which frees the user from explicitly managing the communication handles (or the equivalent `MPI_Request`) for each individual communication operation. Instead, the user can initiate multiple non-blocking communication calls and then ensure their completion with just a single `async_copy_fence` call. This mechanism is especially convenient in programs where the start and completion of non-blocking communication are separated in different functions or files.

LULESH is designed to have good weak scaling performance, and both the UPC++ and the MPI version of LULESH scale well up to at least 32K processes using 32K cores. In the largest experiment we ran (32K processes), the UPC++ version of LULESH is about 10% faster than its MPI counterpart. Our results also demonstrate that applications can now take advantage of PGAS runtimes without the expensive effort of adopting a new language.

In our experience, it is more natural to use UPC++ than UPC for porting LULESH because of two main reasons: 1) UPC++ speaks the same language, C++, as LULESH; 2) the library approach of UPC++ requires less restructuring effort of existing applications than a language approach would require. Since the UPC++ version of LULESH starts from the MPI version, it retains much of its original structure and communication patterns. In the future, we plan to explore the possibility of rewriting the code in a more "PGAS fashion" and using multidimensional arrays to avoid explicitly packing and unpacking non-contiguous ghost regions.

VI. Conclusion

We have presented a PGAS extension for C++ that incorporates popular features from established languages and libraries such as UPC, Titanium and Phalanx. Our "compiler-free" implementation ensures interoperability with other parallel programming frameworks like OpenMP and CUDA, yet it preserves most of the productive syntax supported by UPC. The library-based approach, together with features in C++11, enables the development of features that would be impossible or difficult to achieve in UPC. Such features include support

for task-based programming via `asyncs` and multidimensional arrays like those in Titanium.

Our results indicate that UPC++ applications match the performance of their UPC, Titanium, or MPI equivalents. In choosing the library-based approach, we effectively traded marginally better syntax, static optimizations, and static correctness guarantees for better portability, interoperability, and maintainability. We think this trade-off is justified for three reasons. First, UPC++'s syntax is slightly more noisy than UPC's, but it is well within the tolerance of human programmers. Second, our performance results indicate that static PGAS optimizations are largely ineffective on our applications of interest. Lastly, in our experience, the C++ compiler reports approximately the same amount of information as the UPC compiler for static type checking and error diagnosis.

In contrast, the library-based approach offers important technical and economic advantages. It gives us a more flexible platform for exploring and composing different parallel programming idioms, and it encourages the development of libraries written in UPC++. From an economic perspective, UPC++ doesn't require the maintenance of a compiler, and it makes hybrid programs easier to write, since it doesn't force interoperability with other languages at the object-code level. For these reasons, and based on our experience with the case study applications, we plan to continue evolving UPC++ to meet our long-term application needs.

ACKNOWLEDGMENTS

Authors from Lawrence Berkeley National Laboratory were supported by DOE's Advanced Scientific Computing Research under contract DE-AC02-05CH11231. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory and the National Energy Research Scientific Computing Facility (NERSC) at Lawrence Berkeley National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357 and DE-AC02-05CH11231, respectively.

REFERENCES

[1] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick, "Accelerating applications at scale using one-sided communication," in *International Conference on PGAS Programming Models, PGAS'12*, 2012.

[2] T. A. Johnson, "Coarray C++," in *International Conference on PGAS Programming Models, PGAS'13*, 2013.

[3] R. Nishtala, Y. Zheng, P. Hargrove, and K. A. Yelick, "Tuning collective communication for partitioned global address space programming models," *Parallel Computing*, vol. 37, no. 9, pp. 576–591, 2011.

[4] "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[5] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral, "Shared memory programming for large scale machines," *SIGPLAN Not.*, vol. 41, no. 6, pp. 108–117, 2006.

[6] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apr, "Advances, applications and performance of the Global Arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, Summer 2006. [Online]. Available: http://hpc.sagepub.com/content/20/2/203.abstract

[7] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05, 2005.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98, 1998, pp. 212–223.

[10] W. Kim and M. Voss, "Multicore desktop programming with Intel Threading Building Blocks," *IEEE Software*, vol. 28, no. 1, pp. 23–31, 2011.

[11] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almási, B. B. Fraguela, M. J. Garzarán, D. A. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays." in *PPOPP*, 2006, pp. 48–57.

[12] M. Eleftheriou, S. Chatterjee, and J. E. Moreira, "A C++ implementation of the Co-Array programming model for Blue Gene/L," in *IPDPS'02*, 2002.

[13] "STAPL: Standard Template Adaptive Parallel Library," https://parasol.tamu.edu/stapl.

[14] L. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[15] "HPX: High Performance ParalleX," https://github.com/STEllAR-GROUP/hpx.

[16] M. Garland, M. Kudlur, and Y. Zheng, "Designing a unified programming model for heterogeneous machines," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.

[17] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," in *Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.

[18] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby, "ZPL: A machine independent programming language for parallel computers," *Software Engineering*, vol. 26, no. 3, pp. 197–211, 2000. [Online]. Available: citeseer.ist.psu.edu/article/chamberlain00zpl.html

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th annual international symposium on Computer architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266.

[20] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[21] W. D. Frazer and A. C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *J. ACM*, vol. 17, no. 3, pp. 496–507, Jul. 1970.

[22] S. Olivier and J. Prins, "Scalable dynamic load balancing using UPC," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, 2008, pp. 123–131.