Coarray C++

Troy A. Johnson

Cray Inc.
troyj@cray.com

Abstract

Years ago, the C language made strides into the Fortran-dominated field of high-performance computing (HPC). Recently, a similar trend is that C++ is being used instead of C to write HPC applications. Although C and Fortran programmers have the option to use a partitioned global address space (PGAS) model via Unified Parallel C (UPC) and coarrays, respectively, C++ programmers do not have an equivalently good option. They must resort to writing a mixed-language application, usually linking together C++ and UPC object files, or calling a communication library. Either alternative defeats static type checking because a C++ compiler does not understand UPC types and general-purpose libraries deal with raw bytes. The solution is to develop a PGAS model for C++. As others have demonstrated, Fortran's coarray model can be ported directly to C++ using templates; however, this paper shows that a direct port is insufficient because it does not permit static type checking and C++ idioms. Instead, this paper presents the design of a new approach where these problems are addressed. The implementation is released as Coarray C++ in version 8.2 of the Cray Compiling Environment.

1 Introduction

Fortran [1] is the archetypal HPC programming language and C [3] is common, but increasingly often developers consider C++ [2] for their implementation language. Modern compilers make its performance competitive, many universities teach programming using C++, and its object-oriented features help to organize large applications. Similarly, the Message Passing Interface (MPI) library [12, 13] is the typical HPC parallel programming model, but PGAS languages like Fortran and UPC [17] offer an alternative that is often simpler due to one-sided communication requiring less coordination by the programmer. Although one-sided MPI [13] can provide that same feature, language-based approaches allow the compiler to help the programmer via static type checking.

Unfortunately for HPC application developers, no solid option has arisen to combine these trends to allow programming in C++ with a PGAS model. Currently C++ programmers have two options: write a mixed-language application or use a one-sided communication library. An example of the first option is to write most of the application in C++, but keep all communication and shared data in UPC source files. These parts are glued together with a C language interface that uses type punning because C and C++ compilers understand neither UPC `shared` data types nor pointers to them. For example, the UPC code below provides a C interface for obtaining a `shared` struct from another UPC thread. The struct contains a UPC pointer-to-shared, so its type declaration is not syntactically valid on the C++ side of the interface. Worse, the C++ code cannot portably declare a different structure that has the same layout because UPC does not require that a pointer-to-shared have a particular size or alignment. For example, the C++ code below might allow the interface to work using certain compilers. Using pointers to `void` or treating a UPC data structure as an array of `char` are other techniques commonly seen to interface UPC and C++ code. Therefore, static type checking is lost at the interface. An example of the second option for writing C++ PGAS programs is

to write the entire application in C++ and call a library like SHMEM [6] or Global Arrays [14] for communication. Such libraries are general purpose and deal with raw bytes or a limited set of fundamental types.

```
1  /* UPC Code */
2  struct S1 {
3      ...  /* other C data members */
4      shared int* p;
5      ...  /* other C data members */
6  };
7
8  shared struct S1 data[THREADS];
9
10 void get_from_thread( struct S1* local, int thread ) {
11     *local = data[thread];
12 }
13
14 /* C++ Code */
15 struct S2 {
16     ...  /* other C data members */
17     long blob[2];  /* hopefully the same size and alignment as p */
18     ...  /* other C data members */
19 };
20
21 extern "C" { void get_from_thread( struct S2* local, int thread ); };
```

A PGAS model designed for C++ can provide type-checked communication in a way that feels natural to C++ programmers by supporting C++ idioms, but the details of such a model have been an open question without much investigation. Common features of C and C++ kindle speculation about a hypothetical UPC++ language, but even though UPC is a PGAS model for C, it does not follow that it is the most appropriate model for C++. C and C++ are two distinct languages that abide by different language standards [2, 3] and have their own programming idioms. Specifically, the preferred mechanism for introducing new features to C++ is via the template library and not via grammar modification. This strategy allows new features to be prototyped without compiler modification and made available for comment, often via Boost [5], before being considered for the standard. UPC modifies the C grammar and a UPC++ language would need to modify the C++ grammar to maintain consistency for programmers. Although a `shared<T>` template could implement UPC `shared` types in C++, there are many complications. Briefly, an array of `shared<T>` would not allocate the correct amount of memory per UPC thread, plus array elements are not self-aware of their position in the array – knowledge that a UPC compiler uses to determine data locality. Therefore, T would need to be the array type and properties like block size and rank of the `THREADS` dimension would need to be indicated via additional template parameters. These conflicting strategies – grammar modification versus templates – give UPC++ a difficult path to official adoption, compounded by UPC never having been adopted by standard C. Finally, UPC offers a wide variety of data distribution options that intentionally hide the location of a data access, such that an array element can be accessed without knowing which UPC thread owns it. This distribution flexibility and access transparency is sometimes useful, but experience shows that many data distributions are out-performed by a more locality-aware block distribution [4, 10, 11] that resembles coarrays. Fortran coarrays have been shown to compare well to MPI and outperform UPC versions of the same application [7].

The coarray model adopted by Fortran [1] adds an additional dimension, called a codimension, to a normal scalar or array type. The codimension spans instances of a Single-Program Multiple-Data (SPMD) application, called images, such that the scalar or array on each im-

age becomes a coarray. Each image has immediate access via processor loads and stores to its own coarray, which resides in that image's local partition of the global address space. By explicitly specifying an image number in the cosubscript of the codimension, each image may access other images' coarrays. Fortran permits multiple cosubscripts to be mapped to an image number. Although originally a Fortran idea [15], coarrays have been implemented with C++ templates [8] and Python modules [16].

The C++ coarrays implemented by [8] prototyped an early incarnation of the Fortran coarray concept to avoid the cost of modifying a Fortran compiler. Fortran coarray syntax was literally moved into C++ instead of considering approaches that are more idiomatic for C++ and that permit a greater degree of static type checking. Furthermore, C++ has changed since then. C++11 [2] introduced a shared-memory parallel programming model using threads. Coarray images are a broader and typically orthogonal concept to threads, representing cooperating processes in a distributed system where a given image might consist of multiple threads acting within a shared-memory domain. For consistency and ease of parallel programming, it makes sense that idioms developed for C++11 threads should influence how coarrays are implemented in C++. This paper presents the design and implementation of Coarray C++ in version 8.2 of the Cray Compiling Environment, where type safety and maintaining the "look and feel" of standard C++ are of paramount concern. Implementing Coarray C++ did not require modifying the compiler or runtime libraries, which is typical for C++ template-based programming models [8, 9].

2 Coarray C++ "Hello World"

The following program is the Coarray C++ equivalent of the classic "Hello World" program:

```
1 #include <iostream>
2 #include <coarray_cpp.h>
3 using namespace coarray_cpp;
4 int main( int argc, char* argv[] ) {
5     std::cout << "Hello from image " << this_image()
6               << " of " << num_images() << std::endl;
7     return 0;
8 }
```

The header file coarray_cpp.h included by Line 2 provides all Coarray C++ declarations within `namespace coarray_cpp`. Normally a program imports all of the declarations into its namespace with a `using` directive as on Line 3, but having the `coarray_cpp` namespace grants the programmer the flexibility to deal with name conflicts. The functions `this_image()` and `num_images()` called on Lines 5 and 6 return the current image's zero-based rank and the total number of images in the job. Note that in [8], these functions curiously were member functions, such that one needed a coarray object just to find out image information. The `this_image()` function is vital for the SPMD practice of branching on the image number; global scope is important for cases where no coarray is in scope. The program is compiled with the Cray compiler and executed using four images as follows:

```
> CC -o hello hello.cpp
> aprun -n4 ./hello
Hello from image 0 of 4
Hello from image 1 of 4
Hello from image 2 of 4
Hello from image 3 of 4
```

The Cray compiler automatically links the application with the same PGAS language runtime library used for Cray UPC and Cray Fortran, as well as a networking library. It is possible to use other C++ compilers, such as the GNU g++ compiler, on a Cray system to build Coarray C++ applications, but the user must link with the necessary libraries explicitly. For Coarray C++ implemented for a different platform, the types and function signatures within coarray_cpp.h would remain the same, but their implementation, the runtime libraries, and the mechanism for launching the application would be different.

3 Type System

3.1 Coarrays

Coarray C++ has a generic coarray template with several specializations. Their forward declarations are shown below:

```
1 template < typename T >           class coarray;
2 template < typename T >           class coarray<T[]>;
3 template < typename T, size_t S > class coarray<T[S]>;
4 template < typename T >           class coarray< coatomic<T> >;
```

Line 1 is the generic coarray template, Line 2 is specialized for unbounded arrays (an array where the leftmost extent is unspecified at compile time), Line 3 is specialized for bounded arrays, and Line 4 is specialized for coatomic, which serves the same purpose for images as the `std::atomic<T>` template does for C++11 threads (see Section 5.2). These specializations let the coarray template provide type-appropriate constructors, member functions, and operators. Coarrays of pointers are handled sufficiently by the generic template, but have special properties (see Section 3.6). For bounded array types, all array extents appear as part of the template argument; for an unbounded array type, the leftmost extent is specified at runtime via a constructor argument:

```
1 coarray<int> i;
2 coarray<int[10][20]> x;
3 coarray<int[][20]> y(n);
```

An important distinction from [8], where the declaration for x at Line 2 would be `CoArray<int> x(10, 20)`, is that the extents are part of the type of the coarray so that the compiler can help enforce type safety. Note that for unbounded array types, the compiler has partial extent information. The C++ type system permits only the first array extent to be unbounded, thus in Coarray C++ one cannot declare a coarray where multiple extents are specified at runtime. This restriction matches C++ because it does not permit allocating multidimensional arrays where a non-leading dimension is variable (e.g., `new int[10][n]` is a compile-time error whereas `new int[n][20]` is fine).

A coarray declaration creates a coarray object which allocates and manages an object of its template argument type in the current image's partition of the global address space. Both the creation and destruction of a coarray must be executed collectively by all images. This requirement is satisfied automatically for global and static local coarray declarations, but the programmer is responsible for ensuring it for local and dynamically-allocated coarrays. Although image teams would allow this requirement to be waived, Coarray C++ does not yet have teams. The Fortran standard committee is considering how to implement teams and following their lead might make sense for Coarray C++.

The Cray implementation allocates the managed objects at symmetric virtual addresses across all images, so an all-to-all communication of addresses during construction, as done by [8], is unnecessary. A scalar type is assumed to have a default constructor; if it also has a copy constructor, then the `coarray<T>` constructor accepts an initializer of type T which may have a different value on different images. For an array type, the ultimate element type of the array is assumed to have a default constructor and initializers are not supported. Internally, these constructors are called using the C++ language's placement `new` syntax, passing the symmetric memory address.

3.2 Local Data Access

A `coarray<T>` object transparently behaves like the local T object that it manages, in both l-value and r-value contexts, so that an existing variable declaration of type T can be changed to a coarray by modifying its declaration without having to modify all uses of the variable:

```
1 i = 0;
2 x[1][2] = i;
3 y[3][4] = x[1][2];
```

Assuming the same declarations as in Section 3.1, `i`, `x[1][2]`, and `y[3][4]` all act as a reference to an object of type `int` on the current image. The assignments compile to processor loads and stores and permit normal compiler optimizations like forward substitution. Local assignments within loops may be vectorized for targets that support vectorization.

Note that in [8], the access `x[1][2]` would need to be changed to `x(1, 2)`, which is strange for C++ programmers (though familiar to Fortran programmers). Beyond seeming strange, the different syntax prevents the C++ practice of writing generic code, such as a function template that accepts either a normal array or a coarray. Member access is the only exception to local-access syntax transparency due to C++ operator overloading limitations. C++ does not permit the member access (dot) operator to be overloaded, so accessing a member of a coarray of struct type requires switching to the arrow operator:

```
1 struct Point { int x, y; };
2 coarray<Point> pt;
3 pt->x = 1;
4 pt->y = pt->x;
```

This change is familiar to any C or C++ programmer who has changed an existing variable declaration to pointer type.

3.3 Coreferences

Coreferences extend the C++ concept of references to potentially remote objects that reside in coarrays. Similar to the coarray template, there is a generic `coref` template and multiple specializations to provide type-specific behavior. These types are discussed in Sections 5.2, 6.2, and 6.3.

```
1 template < typename T >           class coref;
2 template < typename T >           class coref<T[]>;
3 template < typename T, size_t S > class coref<T[S]>;
4 template < typename T >           class coref<T*>;
5 template < >                      class coref<coevent>;
6 template < >                      class coref<comutex>;
7 template < typename T >           class coref< coatomic<T> >;
```

To access data on other images, the image number is placed in parenthesis immediately after the coarray object. Any square brackets for array subscripts follow these parenthesis, so `x(5)[1][2]` is `x[1][2]` on image 5. In [8], the syntax would be `x[5](1, 2)`, which uses the Fortran syntax of square brackets for the image number. Having already established in Section 3.2 that square brackets must continue in their familiar role for C++ array element access, it makes more sense to use parenthesis for the cosubscript to maintain a visual distinction for subscripts and cosubscripts. The following assignments compile to code capable of one-sided gets and puts across a network:

```
1 i(7) = 0;                  // put
2 x[1][2] = i(6);            // get
3 y(0)[3][4] = x(5)[1][2]; // get then put
```

From a type perspective, `x(5)[1][2]` is a call to `operator()` of coarray x, which returns a coreference of type `coref<int[10][20]>`. The `[1]` bracket calls `operator[]` of `coref<int[10][20]>`, which returns a `coref<int[20]>`. Finally the `[2]` bracket calls `operator[]` of `coref<int[20]>`, which returns a `coref<int>`. This call sequence is inlined. The `coref<int>` behaves as an `int` in l-value and r-value expression contexts, except that reading it will get data from image 5 and writing it will put data to image 5. If x had been `const` in the context of the access, a `const_coref` would be obtained from the `operator()` call instead. A `const_coref` does not permit modification of data. Accessing members on other images is again complicated by C++ limitations – a pointer to the member must be used:

```
1 pt(5).member( &Point::x ) = 1;
```

The above code sets `pt.x` equal to 1 on image 5. Overloading the `->*` operator was investigated, but it has a different precedence than the dot and arrow operators, which caused confusion in some contexts. Calling member functions of objects on other images is not supported.

3.4 Traits

The assignments in Section 3.3 made bitwise copies of fundamental data types, but C++ objects may have non-trivial copy semantics. C++11 has a template, `std::is_pod<T>`, to check if a type is a "plain old data" type that does not require special semantics; however, Coarray C++ does not yet require a C++11 compiler and `std::is_pod<T>` cannot be usefully emulated without compiler support. To solve this problem, Coarray C++ has the following template:

```
1 template < typename T >
2 struct coarray_traits {
3 static const bool is_trivially_gettable = true;
4 static const bool is_trivially_puttable = true;
5 };
```

Various `coref<T>` member functions consult `coarray_traits<T>` to determine if copying an object's bits is sufficient. A user can specialize the template for a non-trivial type of their own creation. Indicating that a type is not trivially gettable enters a contract for the type to have a remote copy constructor and a remote assignment operator that accept a `const_coref<T>`. These functions can use the `const_coref<T>` to read enough information to calculate how much local storage is required to copy the object, allocate sufficient space, then copy the rest of the remote object's data. Indicating that a type is not trivially puttable prohibits the type from being written in a one-sided manner to another image because it would require help from the target image.

3.5 Copointers

Just as any C++ object can have its address taken, a `coref<T>` has an `address()` member function that returns a `coptr<T>`. Likewise, a `const_coptr<T>` is obtained from a `const_coref<T>`. The `&` operator was not overloaded in case the programmer wants a pointer to a coreference, but one can write a standalone `&` operator that calls `address()`. Copointers support pointer arithmetic and can be used as iterators with standard C++ algorithms. Unlike coreferences, they can be null. Unlike (most) UPC pointers, arithmetic on them never changes the target image. The following code fills the array on image 2 with the value 42:

```
1 #include <algorithm>
2 coarray<int[100]> x;
3 coptr<int> begin = x(2)[0].address();
4 coptr<int> end = x(2)[100].address();
5 std::fill( begin, end, 42 );
```

Local pointers are convertible to copointers; going the other direction, copointers have a `to_local()` member function that attempts to return a normal C++ pointer to the same data. If the copointer targets the current image, then this function will succeed. If the copointer targets a different image within the same shared-memory domain, then the function may be able to return a special address that is mapped to the original object. In all other cases, `to_local()` returns NULL.

3.6 Coarrays of Pointers

A coarray allocates the same amount of memory on every image, but this approach sometimes can waste memory. A coarray of pointers mirrors a Fortran feature where one can create a coarray of a derived type containing a pointer component that is then associated with a different amount of memory on each image. Coarray C++ uses the specialization `coref<T*>` to provide the necessary behavior:

```
1  coarray<int*> x;
2  x = new int[this_image() * 10];
3  *x = this_image();
4  x[4] = this_image();
5  sync_all();
6  ...
7  int y = *x(2);
8  y += x(3)[4];
9  ...
10 sync_all();
11 delete [] x;
```

In the above code, x acts like an `int*` and can hold the result of an allocation via `new` on Line 2. Lines 3 and 4 show that x can be dereferenced locally using normal syntax. The `sync_all()` calls (see Section 6.1) on Lines 5 and 10 ensure that other images do not read the current image's data before it is allocated and initialized or after it is deallocated. The remote accesses on Lines 7 and 8 work because the cosubscript returns a `coref<int*>`, which first reads the pointer from the target image before issuing a second read to the pointer's target *on the target image*. For repeated access to the same data, such as within a loop, this extra read can be hoisted manually via a copointer:

```
1 const_coptr<int> p = x(3)[0].address();
2 for ( int i = 0; i < 30; ++i )
3     foo( p[i] );
```

3.7 Coarrays and Functions

A coarray may be passed to a function via a reference or a pointer, but may not be passed by value. If a coarray could be passed by value, the call would have to be collective. There would be a collective allocation of a temporary coarray, the data within the original coarray would need to be copied into the temporary coarray, and eventually the temporary coarray would need to be collectively destroyed. Pass by value is expensive and there are better alternatives, like passing a coarray as a const reference, so it is a compile-time error.

4 Type Checking

4.1 Static Checking

Coarray C++ types whose shapes are completely known at compile time are statically type checked by the C++ compiler. The following example shows a type error detectable by the compiler:

```
1 void foo( coarray<int[10][20]>& x );
2 coarray<int[5][10]> y;
3 foo( y ); // illegal
```

A bounded type is convertible to an unbounded type:

```
1 void foo( coarray<int[][20]>& x );
2 coarray<int[10][20]> y;
3 foo( y ); // legal
```

4.2 Dynamic Checking

An unbounded type is convertible to a bounded type, but may throw a `mismatched_extent_error`:

```
1 void foo( coarray<int[10][20]>& x );
2 coarray<int[][20]> y(n);
3 foo( y ); // throws if n != 10
```

4.3 shape_cast

For instances where a coarray or coreference of one shape needs to be reinterpreted as a different shape, `shape_cast` provides a runtime conversion. The conversion works provided that the ultimate element type matches and the new type does not have more elements than the old type, otherwise it throws `std::bad_cast`. The syntax is modeled after the other C++ casts: `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`. None of those are sufficient to provide the same functionality as `shape_cast`.

```
1 void foo( coarray<int[10][20]>& x );
2 coarray<int[200]> y;
3 coarray<int[50]> z;
4 foo( shape_cast<int[10][20]>(y) ); // legal
5 foo( shape_cast<int[10][20]>(z) ); // throws
```

5 Memory Model

5.1 atomic_image_fence

Coarray C++ follows the host compiler's C++ memory model for local accesses and the Fortran model for accesses to other images. Accesses by a single image appear to execute in program order. Prior to executing an `atomic_image_fence()` call, which is modeled after C++11's `atomic_thread_fence()`, a write to another image need only be visible to the image that wrote the value. After the fence, the write is visible to all images. Therefore, an implementation of Coarray C++ is free to use non-blocking communication for all writes, provided that it can ensure program order when the same image makes multiple accesses to the same data. Reads block so that a coarray used in an expression context can provide a value. Therefore, coreferences provide a `get()` member function to launch a non-blocking read that is not guaranteed to complete until the next fence.

5.2 coatomics

C++11 introduced the `atomic<T>` template for constructing atomic types. All operations on these types are atomic with respect to C++11 threads. Likewise, Coarray C++ has `coatomic<T>` to provide operations that are atomic with respect to images. Similar convenience typedefs are provided, like `coatomic_long` for `coatomic<long>`. A generic coatomic type uses a comutex to provide atomicity (see Section 6.3), but implementations of Coarray C++ may specialize certain types, like `coatomic<long>`, to use lock-free hardware atomics. In distributed systems, `atomic<T>` operations would use processor atomics whereas `coatomic<T>` operations would use network atomics; these two sets of atomic operations might not be memory coherent and atomic with respect to each other. The following code shows an atomic addition:

```
1 coarray<coatomic_long> x(0L);
2 size_t n = num_images();
3 for ( size_t i = 0; i < n; ++i ) {
4     x(i) += this_image(); // atomic add
5 }
6 sync_all();
7 assert( x == ( n * ( n - 1 ) / 2 ) );
```

Atomic operations may be performed on regular types by explicitly creating a coreference to the matching atomic type, but it is left up to the programmer to ensure that non-atomic operations do not simultaneously touch the data.

```
1 coarray<long> x(0L);
2 coref<coatomic_long> ref( x(i) );
3 ref += this_image(); // atomic add
```

6 Image Synchronization

6.1 sync_all

As with Fortran's `sync all` statement, calling `sync_all()` synchronizes control-flow across all images and implies a fence. A direct equivalent to Fortran's `sync images` statement for point-to-point image synchronization is not provided, but see Section 6.2 below. Experience shows that programmers use `sync images` in contexts where `sync all` is more appropriate and expect equivalent performance, which is not realistic.

6.2 coevent

Events are under consideration for inclusion in the Fortran standard. Coevents provide similar behavior, allowing one image to post an event and another image to wait on an event to be posted. A `post()` call is directed at a particular image, but the `wait()` call does not know which image posted the event:

```
1 coarray<coevent> events;
2 if ( this_image() == 0 ) {
3     // write something to image 1, then
4     events(1).post();
5 }
6 else if ( this_image() == 1 ) {
7     events->wait();
8     // then read the data
9 }
```

The specialization `coref<coevent>` provides the `post()` function in the above example. Waiting on non-local events (e.g., `events(1).wait()`) is not supported.

6.3 comutex

A comutex is modeled after C++11's `std::mutex`. A comutex provides mutual exclusion among images. It is up to the programmer to establish a relationship between a comutex and the data that it protects, although most sensible programming styles dictate that acquiring a mutex on an image implies that the mutex guards data on that image:

```
1 coarray<comutex> m;
2 m(i).lock();
3 // access data on image i
4 m(i).unlock();
```

7 Cofutures

The facility for explicit management of non-blocking communication is based on C++11's `std::future<T>` template. In C++11, a `std::future<T>` manages completion of an asynchronous operation that produces a T value. Likewise, in Coarray C++, a `cofuture<T>` manages a non-blocking copy of type T. A `coref<T>` can provide a `cofuture<T>`, either by calling `get_cofuture()` or relying on implicit conversion:

```
1 coarray<int> x;
2 ...
3 cofuture<int> f = x(i);
4 ...
5 int z = f + 1;
```

Using the cofuture in an expression context automatically waits on the data to arrive. If the data is large, existing storage may be preferable to duplicating storage inside a cofuture. In that case, T is `void` because the cofuture does not store data and the `wait()` member function or the destructor ensures completion:

```
1 coarray<int[100]> x;
2 int y[100];
3 ...
```

```
4 cofuture<void> f = x(i).get_cofuture(&y);
5 ...
6 f.wait();
```

Finally, a non-blocking write can be explicitly managed via `put_cofuture()`:

```
1 coarray<int> x;
2 int y;
3 ...
4 cofuture<void> f = x(i).put_cofuture(&y);
5 ...
6 f.wait();
```

8 Collectives

8.1 cobroadcast

Cobroadcast replicates the value of a coarray from a root image across all images. The broadcast does not imply a `sync_all()` because synchronization is not needed when only local values are accessed, as in this example:

```
1 coarray<int> x;
2 if ( this_image() == 0 ) {
3     x = 42;
4 }
5 cobroadcast( x, 0 );
6 assert( x == 42 );
```

8.2 coreduce

Coreduce performs a broadside reduction of coarray images. For example, reducing a `coarray<int[100]>` yields 100 result values instead of one. Like cobroadcast, no `sync_all()` is implied. By default, every image receives the results as part of the original coarray, but there are options to send the result to only one image or to use a different coarray for the results. Coreduce accepts a commutative and associative function, but implementations may provide optimized specializations with alternative names. For example:

```
1 coarray<int[100]> x;
2 cosum( x ); // coreduce( x, std::plus<int> )
3 comin( x ); // coreduce( x, std::less<int> )
4 comax( x ); // coreduce( x, std::greater<int> )
```

An example of using comax:

```
1 coarray<int> x;
2 x = this_image();
3 comax( x );
4 assert( x == num_images() - 1 );
```

9 Performance Considerations

Because C++ compilers are not aware of Coarray C++, performance would appear to be a huge challenge. This misconception stems from a belief that compilers for PGAS languages only

achieve high-performance by exploiting the language's memory consistency model to schedule and cache remote accesses. Although UPC, and to some extent Fortran, were designed with these optimizations in mind, most of the performance of Cray UPC and Fortran comes from other techniques. The primary network latency optimization is to issue non-blocking writes for all writes, relying on the language runtime to efficiently enforce the memory model. Coarray C++ behaves identically. The primary network throughput optimization is to recognize loops that do small, constant-stride remote memory accesses and replace them with bulk copy functions. UPC has a family of functions (e.g., `upc_memput()`, `upc_memget()`) for this purpose; both UPC loops and Fortran array syntax loops can be mapped to similar functions. To provide this behavior in C++, which does not permit array syntax, Coarray C++ allows a coreference to an array to act as the source or target of an assignment. For syntactic convenience, a `make_coref()` function automatically creates a `coref` from a local object:

```
1 coarray<int[10][100]> x;
2 int local[10][100];
3 ...
4 make_coref( local ) = x(2);
5 x(7)[3] = local[3];
```

Line 4 creates a `coref<int[10][100]>` from *local*, then uses it to store the contents of x from image 2. There also is support for copying complete slices of arrays. Line 5 copies just row 3 to image 7. Experiments with these techniques have shown performance equivalent to Cray UPC and Fortran.

10 Conclusions

Coarray C++ provides typical PGAS language features in a manner that is compatible with C++ idioms and that permits static and dynamic type checking. Throughout this paper, its design was contrasted with a more literal approach [8] to moving Fortran features into C++ that did not have the benefit of building on C++11 parallel programming ideas. The initial version of Coarray C++ is released with version 8.2 of the Cray Compiling Environment and uses the same language runtime and networking libraries as Cray UPC and Fortran. Future work is to evolve the language based on user experience, as well as any interesting developments in both the Fortran and C++ standards. Implementations for other platforms are encouraged.

Acknowledgements

The author thanks David Henty for presenting this paper, Bill Long for insight into Fortran's coarray rationale, Steve Vormwald for discussions comparing UPC and coarrays, Kristyn Maschhoff for taking the time to report bugs while undertaking a large project written in Coarray C++, and Harvey Richardson and the anonymous reviewers for their helpful comments.

References

[1] Fortran Standard: ISO/IEC 1539-1:2010. 2010.

[2] C++ Standard: ISO/IEC 14882:2011. 2011.

[3] C Standard: ISO/IEC 9899:2011. 2011.

[4] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, B. Pugh, P. Sadayappan, J. Spacco, and C. wen Tseng. Evaluating the Impact of Programming Language Features on the Performance of

Parallel Applications on Cluster Architectures. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 194–208.

[5] Boost C++ Libraries. www.boost.org.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Models*, 2010.

[7] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47, 2005.

[8] M. Eleftheriou, S. Chatterjee, and J. E. Moreira. A C++ Implementation of the Co-Array Programming Model for Blue Gene/L. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[9] M. Garland, M. Kudlur, and Y. Zheng. Designing a Unified Programming Model for Heterogeneous Machines. In *SC12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[10] N. Jansson. Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-Sided Communication. *VECPAR 2012, Lecture Notes in Computer Science*, 7851:128–139, 2013.

[11] H. Jin, R. Hood, and P. Mehrotra. A Practical Study of UPC with the NAS Parallel Benchmarks. In *Proceedings of the Third Conference on Partitioned Global Address Space Programming Models*, 2009.

[12] MPI Forum. MPI: A Message-Passing Interface Standard, Version 2.2. 2009.

[13] MPI Forum. MPI: A Message-Passing Interface Standard, Version 3.0. 2012.

[14] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[15] R. W. Numrich. A Parallel Extension to Cray Fortran. *Scientific Programming*, 6:275–284, 1997.

[16] C. E. Rasmussen, M. J. Sottile, J. Nieplocha, R. W. Numrich, and E. Jones. Co-array Python: A Parallel Extension to the Python Language. *Euro-Par 2004, Lecture Notes in Computer Science*, 3149:632–637, 2004.

[17] UPC Consortium. UPC Language Specification 1.3 (Draft). September 2013.