A Local-View Array Library for Partitioned Global Address Space C++ Programs

Amir Kamil
Lawrence Berkeley National Laboratory
akamil@lbl.gov

Yili Zheng
Lawrence Berkeley National Laboratory
yzheng@lbl.gov

Katherine Yelick
University of California, Berkeley and Lawrence Berkeley National Laboratory
yelick@cs.berkeley.edu

Abstract

Multidimensional arrays are an important data structure in many scientific applications. Unfortunately, built-in support for such arrays is inadequate in C++, particularly in the distributed setting where bulk communication operations are required for good performance. In this paper, we present a multidimensional library for partitioned global address space (PGAS) programs, supporting the one-sided remote access and bulk operations of the PGAS model. The library is based on Titanium arrays, which have proven to provide good productivity and performance. These arrays provide a local view of data, where each rank constructs its own portion of a global data structure, matching the local view of execution common to PGAS programs and providing maximum flexibility in structuring global data. Unlike Titanium, which has its own compiler with array-specific analyses, optimizations, and code generation, we implement multidimensional arrays solely through a C++ library. The main goal of this effort is to provide a library-based implementation that can match the productivity and performance of a compiler-based approach. We implement the array library as an extension to UPC++, a C++ library for PGAS programs, and we extend Titanium arrays with specializations to improve performance. We evaluate the array library by porting four Titanium benchmarks to UPC++, demonstrating that it can achieve up to 25% better performance than Titanium without a significant increase in programmer effort.

1. Introduction

One of the primary data structures in many scientific applications is a regular multidimensional array. A multidimensional array is often used to represent a physical domain, such as a discretization of space. Regular arrays are also at the core of many numerical algorithms, including dense linear algebra and Fourier transforms. Unfortunately, support for multidimensional arrays is very limited in C++. Such arrays generally must be manually mapped to one-dimensional arrays, with the user performing all the indexing logic. The problem is even worse in the parallel setting, since efficiently communicating subsets of an array that are not physically contiguous requires significant effort from the programmer.

In this paper, we describe the design and implementation of a C++ multidimensional array library based on that of Titanium [16], an explicitly parallel dialect of Java, since Titanium's array library has proven to provide good productivity and performance in large-scale parallel programs. Whereas Titanium has a compiler with support for array-specific analyses, optimizations, and code generation, we seek to implement an equivalent array abstraction solely through a C++ library. The main goal of this work is to demonstrate that a pure library implementation can match the productivity and performance of a compiler-based approach to multidimensional arrays.

We implement the array library in the context of UPC++ [17], a library for partitioned global address space (PGAS) programs, which allow data to be directly accessed by processors that do not share the physical address space in which the data are located. We also include the ability for the array library to be used separately from UPC++ in standard C++ programs. The array library provides a local view of data, where each rank is responsible for creating its own pieces of a global data structure, which aligns with the local view of execution in explicitly parallel PGAS programs. Finally, the library allows one-sided communication of non-contiguous data between processors with minimal programmer effort.

In order to match the performance of Titanium arrays, we implement a number of specializations to improve the performance of array accesses. We evaluate our library by porting four benchmarks from Titanium to UPC++, demonstrating that the UPC++ versions match the productivity and exceed the performance of Titanium code.

2. Background and Related Work

UPC++ is a library for C++ that implements the main features of UPC [6] in the context of C++, while also providing many features found in other PGAS languages. It includes the global pointers, shared variables, shared arrays, and basic single program, multiple data (SPMD) execution model from UPC, a model of asynchronous task execution inspired by X10 [14], Chapel [8], Habanero [5], and Phalanx [10], and a team model similar to Titanium [11]. The UPC++ library is designed for large-scale scientific applications written in C++ and is interoperable with other parallel libraries such as MPI, OpenMP, and CUDA.

Despite the importance of multidimensional arrays to scientific applications, C++ provides only very limited support for such arrays, and this support is even more inadequate in the PGAS setting. On the other hand, Titanium's multidimensional array library is one of its most productive features [9], allowing one-sided remote access to arrays and one-sided bulk copies of non-contiguous subsets of arrays. These features are especially useful in adaptive block-structured computations; for example, a port of the Chombo adaptive mesh refinement (AMR) application requires an order of

ARRAY'14, June 11 2014, Edinburgh, United Kingdom
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2937-8/14/06$15.00.
http://dx.doi.org/10.1145/2627373.2627378.

magnitude fewer lines in Titanium than in C++/Fortran, largely due to Titanium's array library [15]. Due to the success of Titanium's library, we use it as the basis for an array library for UPC++.

Titanium's multidimensional array library was inspired by the dense and strided regions and arrays in ZPL [7], which also inspired Chapel's domain and array library. Unlike in Chapel, the elements of a Titanium array are located in a single memory space, though they may be accessed from a remote processor. However, distributed array structures can be built from local pieces on different processors, allowing the construction of a wide variety of complex data structures. Since a distributed structure is built from local pieces, Titanium provides a *local view* of data.

Many existing libraries provide the ability to define a *global-view* data structure distributed across multiple memory spaces. While a global-view structure is easier to construct than a local-view one, it often precludes the expression of the irregular structures required in adaptive computations. We have chosen to provide a local-view library as a starting point; in the future, we plan on building a global-view library on top of it that provides the most common global distributions.

Perhaps the most commonly used PGAS array library is Global Arrays (GA) [1]. While the GA toolkit is useful for many applications, it is difficult to express the structures required in AMR. In addition, it does not provide a higher-level C++ interface using templates and operator overloading, and it restricts the element type to basic numerical types. Other library-based approaches include hierarchically tiled arrays [3] and POOMA [13]. However, both libraries are designed for data-parallel applications rather than explicitly parallel PGAS programs.

3. Library Overview

In this section, we provide an overview of the UPC++ multidimensional array library and its features. All classes and functions in the library are located in the `upcxx` namespace. For simplicity, we use unqualified names in this paper, as in a program with a **`using namespace`** `upcxx;` declaration. Aside from syntax, most features are shared between Titanium and UPC++ arrays.

3.1 Domain Library

As in Titanium, the array library contains a domain library as a subset in order to represent indices and index sets. This component consists of points, rectangular domains, and general domains.

Points A *point* is a set of N integer coordinates, representing a location in N-dimensional space. In UPC++, it is represented using the `point<N>` class template, parameterized by dimensionality. A `point<N>` is defined as plain-old data (POD), with an array of size N as its only member, so it may be constructed using an initializer list when declaring a variable:

```
point<3> p = {{ -1, 3, 2 }};
```

In addition, the `PT` function is overloaded to construct a `point<N>` when passed N arguments:

```
point<3> p = PT(-1, 3, 2);
```

Many useful operations are defined on points, including arithmetic operations between points of the same dimensionality or points and integers. Comparison operations are also defined between points. Operator overloading is used to provide simple syntax:

```
point<3> p2 = PT(-1, 3, 2) + PT(3, -2, 4);
```

The indexing operator `[]` is also overloaded on a `point<N>`, where an argument of i retrieves the ith component of the point for $1 \leq i \leq N$.

Rectangular Domains A *rectangular domain* is a regular set of points between a given lower bound and upper bound, with an optional stride in each dimension. For example, the set of points $\{(1,1),(1,3),(3,1),(3,3)\}$ constitutes a rectangular domain with a stride of $(2,2)$.

In UPC++, the `rdomain<N>` class template represents a rectangular domain. An `rdomain<N>` is created by passing in an inclusive lower bound `point<N>`, an exclusive upper bound `point<N>`, and an optional stride `point<N>` to the constructor. The latter defaults to 1 in each dimension. The following produces a representation of the rectangular domain above:

```
rdomain<2> rd(PT(1, 1), PT(4, 4), PT(2, 2));
```

The `RD` function is overloaded to produce an `rdomain<N>` when passed `point<N>`s:

```
rdomain<2> rd2 = RD(PT(-1, 1), PT(3, 2));
```

General Domains A *general domain* is an arbitrary set of points. It is represented by the `domain<N>` class template. A general domain can be created from a set of points or as the result of domain operations, as described below.

Domain Operations Arithmetic operations are defined between both types of domains and points. For example, adding a point to a domain results in a translation of that domain, so that the expression `RD(PT(1, 1), PT(3, 3)) + PT(1, 2)` is equivalent to `RD(PT(2, 3), PT(4, 5))`.

Set operations are defined between both types of domains, with the + operator for union, $*$ for intersection, and $-$ for difference. The type of the resulting object is `domain<N>` unless the result is necessarily rectangular, such as an intersection between two rectangular domains, in which case the result is of type `rdomain<N>`.

Many other transformation operations are defined on an `rdomain<N>`, such as slicing to get a `rdomain<N-1>` and shrinking, accretion, and border operations that are useful in applications that use ghost zones.

Foreach Loops As in Titanium, the UPC++ domain library defines a special type of loop to sequentially iterate over the points in a domain. Given a variable name and a domain, a `foreach` loop declares a point variable with the given name and iterates over the points in the domain, binding a point to that name in each iteration. For example, the following prints out all the points in a domain:

```
foreach (pt, some_domain)
  cout << pt << endl;
```

A `foreach` loop does not require the dimensionality of the given domain or the type of the point variable to be explicitly specified. This allows a user to write generic code that works for any dimensionality.

3.2 Arrays

A multidimensional array is a mapping of points in a rectangular domain to elements. In UPC++, the `ndarray<T, N, F1, F2>` class template represents an array. The first template parameter is the element type, the second is the dimensionality, and the remaining two parameters are optional locality and layout specifiers. The locality specifier may be `local` or `global`; the former specifies that the array's elements are stored in memory that is physically addressable by the current thread, while the latter allows the elements to be located in a remote memory space. The default is `local` if no locality specifier is provided. The layout specifiers will be discussed in §4.2.

An array may be created over a rectangular domain:

```
ndarray<int, 2> A(RD(PT(1, 1), PT(4, 3)));
```

This allocates memory for the array elements in local memory, even if the array is declared as `global`. The given domain is the index space of the newly created array.

The `ndarray` class template also defines a number of operations that produce new views over the elements of an existing array, without copying any data. For example, the call `A.constrict(rd)` produces a view of the array `A` that is restricted to the intersection of its index space and the rectangular domain `rd`. Many other view operations are defined, including translation, injection, slicing, shrinking, and so on.

The call `A.domain()` returns the index space of the array view `A` and is always a rectangular domain. `A.size()` is the number of elements in `A`.

An array is indexed using points, so that `A[p]` produces a reference to the element mapped to index `p`. Indexing a global array produces a global reference (type `global_ref<T>` in UPC++).

While it is possible to iterate over a non-rectangular subset of an array, either manually or with a `foreach` loop, it is not currently possible to create an array or an array view over a general domain. We may add a mechanism for doing so in the future.

In order to facilitate construction of distributed array structures, the `exchange` method is an all-to-all communication operation over all ranks in the current team, sending an element from each rank to every other rank. This operation can be used to construct a directory of the arrays on each rank:

```
ndarray<double, 3> myArray(myDomain);
ndarray<ndarray<double, 3, global>,
        1> dir(RD(PT(0), PT(ranks())));
dir.exchange(myArray);
```

The elements of `dir` must be declared `global`, since their underlying elements may be located on a remote rank. The array library allows implicit promotion of `local` arrays to `global`, so the actual arrays to be exchanged need not be declared `global`.

Array Copies The most significant feature of Titanium and UPC++ arrays is the ability to copy elements from one array to another with a single call, even if one or both arrays are located remotely. The call `A.copy(B)` copies all elements in the intersection of the index spaces of `A` and `B` from `B` to `A`. Semantically, this is a one-sided operation, even if packing or unpacking must be done at the (possibly remote) source and destination. Compare this to the equivalent operation in MPI, which requires the source to explicitly pack its elements and post a send and the destination to post a receive and explicitly unpack the elements.

As an example, consider a grid computation with ghost zones, in which each rank has 6 neighbors with which it must exchange data. In the setup phase, a directory is created using an `exchange` operation:

```
allGrids.exchange(myGrid);
```

Then the following is the code to perform the copies, assuming the array `nb` holds the IDs of a rank's neighbors and `SIZE` is the thickness of the ghost regions:

```
for (int i = 0; i < 6; i++)
  allGrids[nb[i]].copy(myGrid.shrink(SIZE));
```

The `shrink` operation restricts `myGrid` to its interior, so that only interior elements are copied from the source to the destination. A single statement is required for each copy, even though most of these copies require packing at the source and unpacking at the destination. This is much simpler than the equivalent code in MPI with standard C++ arrays.

UPC++ arrays also support asynchronous copies, which integrate with the asynchronous tasking model defined by UPC++. Sparse scatter and gather operations are also defined on arrays.

Bounds Checking One of the most common types of errors in array-based code is accessing an array with an invalid index. Such errors are difficult to debug in C++; C++ provides no bounds checking, and its weak degree of memory safety often results in silent errors. On the other hand, checking every array access for legality can result in overhead that is unacceptable to many scientific programmers.

The UPC++ array library takes an intermediate approach where bounds checking is disabled by default but can be explicitly enabled by the user. In our implementation, defining the `UPCXXA_BOUNDS_CHECKING` macro to 1 before including the array headers turns on bounds checking for the given compilation unit.

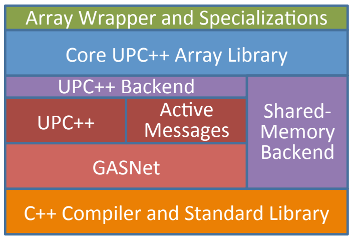

Figure 1. The UPC++ array implementation stack.

3.3 Additional Features

In addition to the features above, which are all available in the Titanium language, we have added several new features that are not in Titanium. Since arrays are provided as a library, new features can be implemented without modifying a compiler.

User-Defined Coordinate Type By default, the coordinate type (i.e. the components of a `point<N>`) used in the domain and array library is **`int`**. However, a program may require an index space that exceeds the bounds of the **`int`** type (generally 32 bits on most modern platforms). The UPC++ library allows an alternate coordinate type, such as `int64_t`, to be specified by defining the `UPCXXA_COORDINATE_TYPE` macro to the desired type before including the array library headers.

Padding Some algorithms benefit from padding arrays with extra elements after each dimension to optimize for caching effects (e.g. pad the last dimension to take up a whole cache line). In many cases, it is insufficient to create an array over a larger physical domain; the size of dimension i of an unpadded, row-major N-dimensional array is always a multiple of the sizes of dimensions $i+1$ through N, which may not divide the size of a cache line. Thus, padding must be allowed through some other means in order to avoid this constraint.

In order to specify padding in UPC++, a `point<N>` may be passed in as the last argument when constructing an N-dimensional array. The ith coordinate of the `point` is the number of elements of padding to use after the ith dimension of the array. It is an error if any coordinate is negative.

Column-Major Arrays By default, UPC++ arrays are laid out in row-major format, with the last dimension contiguous (except in the case of the `simple_column` specialization described in §4.2, which requires column-major format). The user may manually specify whether to use row-major or column-major layout by passing in an optional **`bool`** after the array domain, with **`true`** signifying column-major format and **`false`** row-major format.

4. Implementation

In this section, we describe the implementation of UPC++ arrays, as well as the implementation-driven specializations provided by the array library.

4.1 Software Architecture

The overall software architecture of the UPC++ array library is shown in Figure 1. The implementation takes a multilayered approach, allowing reuse of existing code and enabling the array library to be used with different backends. The top layer is a true C++ interface using templates and template metaprogramming to implement the user-level interface described in §3. Below that is the core implementation ported from Titanium, which itself uses a thin translation layer to support multiple runtime backends. The existing backends are a shared-memory backend over standard C++ and a distributed-memory backend built on the core UPC++ library and active messages, which are in turn based on the GASNet communication layer [4].

The core of the UPC++ domain and array library is a direct port of the Titanium libraries. The Titanium implementation uses a combination of Titanium code, native C code, and compiler-generated code. The UPC++ implementation is entirely in C++.

Domain Library Titanium's domain library is implemented in the Titanium language itself, with the exception of points, which are implemented directly in the Titanium compiler. We implemented points from scratch in C++ and ported rectangular and general domains from Titanium to C++.

Array Library In Titanium, the bulk of the array library is written in C, with macros as placeholders for element type, dimensionality, and operations on local and global pointers. The Titanium compiler generates the appropriate macro definitions when instantiating a particular array type.

We ported the C implementation of Titanium's array library to C++, using template parameters to specify element type and dimensionality. Some care had to be taken to avoid errors in dimensionality-specific code. Titanium avoids any issues by using macros to disable code for the wrong dimensionality. The UPC++ library, on the other hand, has to use template metaprogramming techniques to do so, since the preprocessor does not understand template parameters.

For simplicity of implementation, we found it convenient to retain the use of macros for specifying operations on local and global pointers. As such, we actually generate two internal class templates for arrays from the same source code, one for local arrays and one for global arrays, with macros used to distinguish their implementation.

On top of the two internal array templates, we implemented the `ndarray` class template as a unified wrapper that uses the appropriate internal array template class depending on whether the `ndarray` is local or global. Thus, from the user perspective, local and global arrays appear to be different instantiations of the same class template. This makes it easier to write code that works for both local and global arrays.

Template metaprogramming is used to select the correct internal array template, as well as the appropriate local or global pointer and reference type. Template metaprogramming is also used to allow local arrays to be implicitly promoted to global arrays, and for global arrays to be explicitly cast to local arrays.

Foreach Loops A foreach loop is implemented as a macro, translating to a pair of nested for loops. This allows a foreach header to be used syntactically in the same manner as a for header, while also allowing **`break`** and **`continue`** statements to function as expected. The following is the C++11 implementation of the foreach macro:

```
#define foreach(p, D)                          \
  FOREACH_(p, (D), UNIQUIFYN(fptr_, p))
#define FOREACH_(p, D, F)                      \
  for (auto F=D.iter(); F.val; F.val=0)        \
    for (auto p=F.start(); F.next(p);)
```

Two for loops are needed since the types of the variables declared in each header differ. The outer loop is used solely to declare an iterator over a domain; the `UNIQUIFYN` macro, whose implementation is not shown, is used to obtain a unique temporary name. The inner loop iterates over the actual points in the domain.

The C++11 foreach loop uses type inference to determine the dimensionality of the given domain and its points. We also provide a non-C++11 implementation that infers the dimensionality using the **`sizeof`** operator.

Backend Implementation A major goal of the UPC++ array library was to enable it to be used as a standalone library, without requiring the rest of UPC++, while also allowing it to be well-integrated with UPC++. We accomplish this by defining a thin backend interface using macros, which specify what backend types to use for global pointers and references, how to operate on global pointers, how to perform remote reads and writes, and so on. Currently, the array library comes with a backend implementation over shared memory, while the separate UPC++ library defines its own distributed-memory backend implementation.

The distributed-memory backend implementation uses UPC++ types `global_ptr` and `global_ref` for global pointers and references, respectively, and uses UPC++ remote copying operations to perform bulk reads and writes. In addition, active messages are used for more complicated remote copying operations, such as those that require the receiver to unpack elements into non-contiguous memory. Both UPC++ and the array active messages are built on top of the GASNet communication library.

4.2 Specializations

Since UPC++ is implemented as a library, it does not have static analyses and transformations that would be available to a compiler. Titanium, for example, can perform dimensionality-specific transformations on foreach loops, and a lot of analysis and optimization work was done on array accesses within such loops [12].

Rather than relying on compiler analysis and transformations, the UPC++ array library provides specializations that can be explicitly applied by the user. Currently, these include specializations for the layout of an array as well as dimensionality-specific foreach loops.

Layout Specifiers Since an array can be created over any rectangular domain, the logical, or index, domain of an array may differ from its physical domain. For example, an array that is constructed with a domain that has a stride of 2 in each dimension is laid out contiguously in physical memory, resulting in a physical stride of 1 in each dimension. When indexing into such an array, divisions are required to translate from the logical to the physical domain. These are expensive operations that are useful to avoid if possible.

It is possible for an array with a non-unit logical stride to have a matching physical stride. For example, if an array is created with unit stride, but then that array is restricted to a domain with stride 2, then the physical stride of the restricted array will also be 2. In this case, a division is not required to translate from the logical to the physical domain.

UPC++ defines a set of specializations that restrict the layout of an array, allowing unnecessary operations to be avoided when indexing into an array. A `strided` array may have any logical or physical stride. An `unstrided` array has matching logical and physical strides in each dimension. A `simple` array has matching strides and is in row-major format, with the last dimension contiguous. Finally, a `simple_column` array has matching strides and is in column-major format, with the first dimension contiguous. An array's layout may be specialized as one of these options by passing in the appropriate specifier as the third or fourth template argument to the `ndarray` type. The default layout is `strided`.

Implicit conversions are provided from more-constrained to less-constrained layouts (e.g. `simple_column` to `unstrided`,

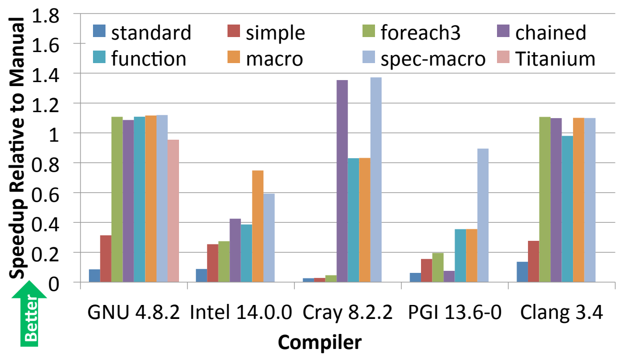

Figure 2. Sequential performance of stencil variants, compared to manual indexing using the same compiler.

but not `simple_column` to `simple`), and explicit conversions otherwise. Thus, in most cases it is sufficient to cast an array to the most constrained layout possible in the computationally intensive parts of a program. Methods such as `is_simple` are provided for querying the layout of an array at runtime, allowing a cast to be done only when legal.

Specialized Foreach Loops As discussed in §4.1, the standard foreach loop is implemented using an iterator over the points of a domain. When iterating over a multidimensional array, this can be much worse than using nested loops, since the compiler cannot lift loop-invariant expressions out of the inner loops.

To address this, UPC++ provides foreach loops that are specialized for a particular dimensionality. For example, the following is a reduction over a 3D array using a specialized `foreach3` loop:

```
foreach3 (i, j, k, A.domain())
  sum += A[PT(i, j, k)];
```

Rather than declaring a `point<3>`, the `foreach3` loop declares an integer index variable for each dimension. The user may explicitly construct a `point<3>` or use one of the alternate indexing variants described below. As with `foreach`, a **`continue`** statement works as expected in a specialized foreach loop; however, a **`goto`** must be used to exit the loop rather than a **`break`**, since a specialized N-dimensional foreach loop translates into N nested for loops.

By default, the array library provides specialized foreach loops up to 9 dimensions.

Indexing Variants Since the UPC++ array library relies on the underlying C++ compiler to optimize array indexing, it provides multiple alternative means to index an array; the fastest mechanism is dependent on the C++ compiler. For example, the body of the `foreach3` loop above may be written as follows:

- using points: `sum += A[PT(i, j, k)];`
- using chained indexing: `sum += A[i][j][k];`
- using function-call syntax: `sum += A(i, j, k);`
- using macros; a preamble must be added before the loop:
  ```
  AINDEX3_SETUP(A);
  foreach3 (i, j, k, A.domain())
    sum += AINDEX3(A, i, j, k);
  ```
- using specialized macros; for example, for a `simple` array, `AINDEX3_simple` may be used instead of `AINDEX3` above.

All of these indexing varieties are semantically equivalent and optionally support bounds checking. The most efficient indexing variant depends on the compiler. Figure 2 shows the performance of these variants on a 7-point 3D stencil code, along with layout and loop specializations, on five different compilers. All experiments were run with bounds checking disabled. The baseline is directly indexing the array memory within a `foreach3` loop, computing the offsets manually, using the same compiler. The **standard** variant uses a standard `foreach` loop on `strided` arrays, while **simple** uses a standard `foreach` with `simple` arrays. The **foreach3**, **chained**, **function**, **macro**, and **spec-macro** variants use `foreach3` loops over `simple` arrays, but with point indexing, chained indexing, function-call syntax, macro indexing, and specialized macro indexing, respectively. In the case of the GNU compiler, the Titanium implementation, using the GNU compiler as its backend, is provided as a reference.

The results show that for all five compilers, using both the `simple` layout specialization and the specialized `foreach3` loop improve performance. For three of the compilers, at least one indexing variant meets or exceeds the performance of manually computing indices; for the other two compilers, the best variant achieves 75% and 89% of manual performance. In cases where manual indexing is necessary, UPC++ arrays provide a `base_ptr` method that returns a pointer to the underlying array memory, allowing an array to be indexed manually. Directly accessing the array memory precludes any bounds checking, but it does allow external libraries to be used with UPC++ arrays.

The GNU results also show that despite being implemented as a library, UPC++ can perform better than Titanium with a minimal effort by using the appropriate specializations.

5. Evaluation

Since the main goal of the UPC++ array library is to provide the productivity and performance of Titanium's array library, we evaluated UPC++ arrays by porting four benchmarks from Titanium to UPC++: three of the NAS Parallel Benchmarks [2], conjugate gradient (CG), Fourier transform (FT), and multigrid (MG), and a 7-point stencil over a 3D grid that uses a Jacobi (out-of-place) iteration strategy. Aside from syntax, Titanium and UPC++ arrays are nearly interchangeable. As a result, each benchmark required only a few hours to port, with the bulk of the time spent translating Titanium/Java features to C++ (e.g. determining when to use C++ references or pointers, converting Java arrays to C++ arrays, etc.). The only significant change we made was to use the specialized layouts and loops described in §4.2, with point-indexing as in Titanium. Excluding declarations, the UPC++ and Titanium code are nearly identical in size, providing some indication that UPC++ arrays are as productive as Titanium[1].

In measuring performance, we focused on single-node runs, since any difference in local computation time or memory usage would be apparent there. UPC++ and Titanium both use the GASNet communication layer for remote accesses, so communication performance should be the same for both UPC++ and Titanium code. Our experiments were run on the Edison system at NERSC, which is a Cray XC30 with two 12-core 2.4 GHz Ivy Bridge processors per node and the Cray Aries interconnect. We used a separate process on each core, and when using more than one core on a node, we divided processes evenly among the two processors in order to maximize available memory bandwidth. Both UPC++ and Titanium code were compiled using version 4.8.2 of the GNU compiler as the backend.

Figure 3 shows the performance of the three NAS benchmarks on up to 16 cores of a single node. The class B problem size was

[1] C++ requires many declarations to be placed in headers in order to be accessible to other compilation units, which causes declarations to require more code in C++ than in Titanium. However, we do not believe that this has a significant impact on productivity.

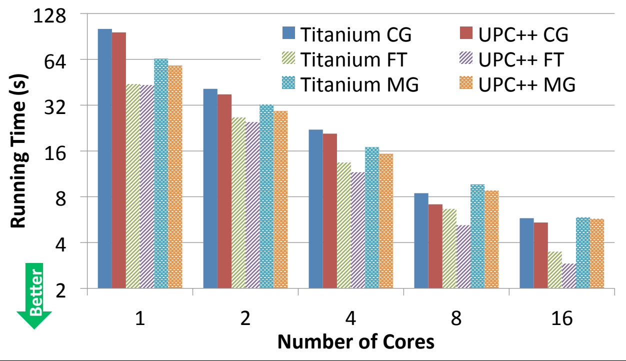

Figure 3. Performance of NAS CG, FT, and MG on one node.

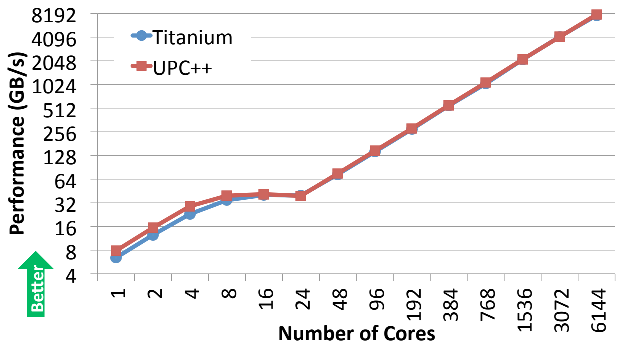

Figure 4. Weak scaling results for stencil, 256^3 grid per core.

used for CG and FT, while class C was used for MG. In all cases, UPC++ performs better than Titanium, up to a maximum of about 20% in FT on 8 and 16 cores.

For the stencil code, we ran a weak-scaling experiment up to 6144 cores (256 nodes), with a grid size of 256^3 per core. The 7-point stencil computation is memory bound, with no memory-hierarchy optimizations (e.g. cache blocking), and Figure 4 shows the effective memory bandwidth achieved in each configuration. At small numbers of cores on a single node, the UPC++ code is around 15-25% faster than Titanium until the memory bandwidth limit is reached at 16 cores, at which point both UPC++ and Titanium provide the same performance. On multiple nodes, UPC++ and Titanium again match, achieving nearly linear speedup.

The results show that UPC++ arrays match or exceed the performance of Titanium, despite being implemented as a library rather than in a compiler. And since the UPC++ and Titanium benchmark code are nearly identical modulo differences in the C++ and Java base languages, we believe that UPC++ arrays also provide the same level of productivity as Titanium arrays.

6. Conclusion

In this paper, we presented a multidimensional array library for C++ programs, particularly partitioned global address space (PGAS) programs written using the UPC++ library. Since the arrays in the Titanium language have proven to provide good productivity and performance, we chose to base our array library on Titanium's, with the goal of matching it in productivity and performance. By providing specializations to optimize array accesses, we have achieved this goal despite the lack of array-specific static analysis and optimization. Our experience with porting Titanium benchmarks to UPC++ demonstrates that our array library provides similar productivity as Titanium, while our experimental results show that the library's performance matches or exceeds Titanium's by up to 25%.

Our current library provides a local-view abstraction of data, where each rank builds its own piece of the global data structure. This provides maximum flexibility in the varieties of global structures that can be represented. However, we do recognize the productivity benefits of global-view data structures. In the future, we plan to use the current local-view library as a building block for creating global-view array libraries in UPC++.

Acknowledgment

This work was supported in part by the Department of Energy's Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory (LBNL) Contract DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the same LBNL contract.

References

[1] Global Arrays webpage. http://www.emsl.pnl.gov/docs/global/.

[2] D. Bailey et al. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] G. Bikshandi et al. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, 2006.

[4] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, 2002.

[5] Z. Budimlic et al. Parallel object-oriented scientific computing with Habanero-Java. In *9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'10)*, October 2010.

[6] W. Carlson et al. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[7] B. L. Chamberlain et al. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.

[8] Cray Inc. *Chapel Specification 0.4*, Feb. 2005.

[9] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.

[10] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Supercomputing 2012*, November 2012.

[11] A. Kamil and K. Yelick. Hierarchical computation in the SPMD programming model. In *The 26th International Workshop on Languages and Compilers for Parallel Computing*, September 2013.

[12] G. R. Pike. *Reordering and Storage Optimizations for Scientific Programs*. PhD thesis, University of California, Berkeley, 2002.

[13] J. V. W. Reynders et al. POOMA: A framework for scientic simulations of paralllel architectures. In *Parallel Programming in C++*. MIT Press, 1996.

[14] V. Saraswat. *Report on the Experimental Language X10, Version 0.41*. IBM Research, Feb. 2006.

[15] T. Wen and P. Colella. Adaptive mesh refinement in Titanium. In *The 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05)*, April 2005.

[16] K. Yelick et al. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.

[17] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *The 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS14)*, May 2014.