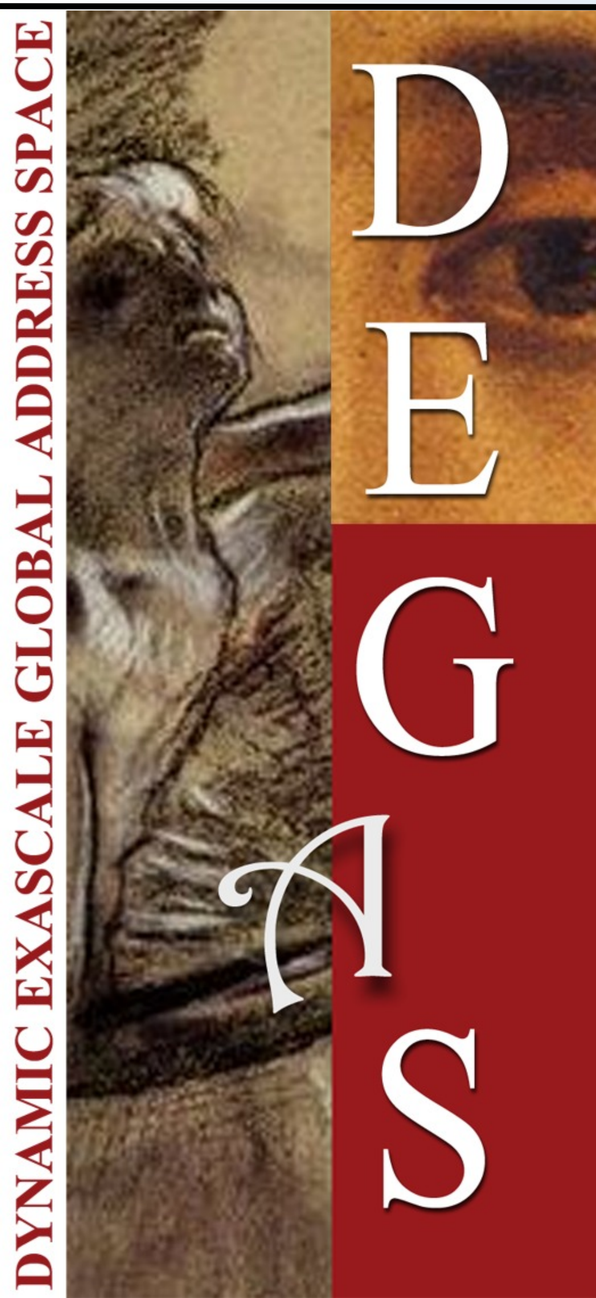
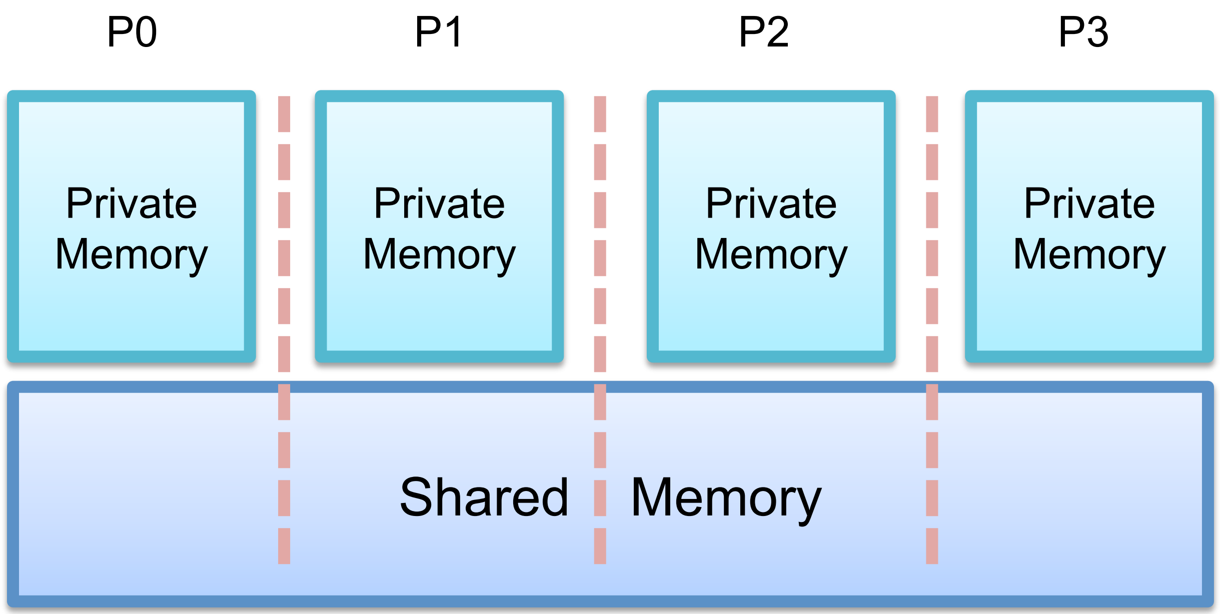


A Local-View Array Library for Partitioned Global Address Space C++ Programs

Amir Kamil, Yili Zheng, and Katherine Yelick
Lawrence Berkeley Lab
Berkeley, CA, USA

June 13, 2014

Partitioned Global Address Space Memory Model



PGAS Abstraction

Variants and Extensions: AGAS, APGAS, APGNS, HPGAS…

UPC++ Overview

- A C++ PGAS extension that combines features from:
  - UPC: dynamic global memory management and one-sided communication (put/get)
  - Titanium/Chapel/ZPL: multi-dimensional arrays
  - Phalanx/X10/Habanero: async task execution
- Execution model: ***SPMD + Aysnc***
- Good interoperability with existing programming systems
  - 1-to-1 mapping between MPI rank and UPC++ thread
  - OpenMP and CUDA can be easily mixed with UPC++ in the same way as MPI+X

A “Compiler-Free” Approach for PGAS

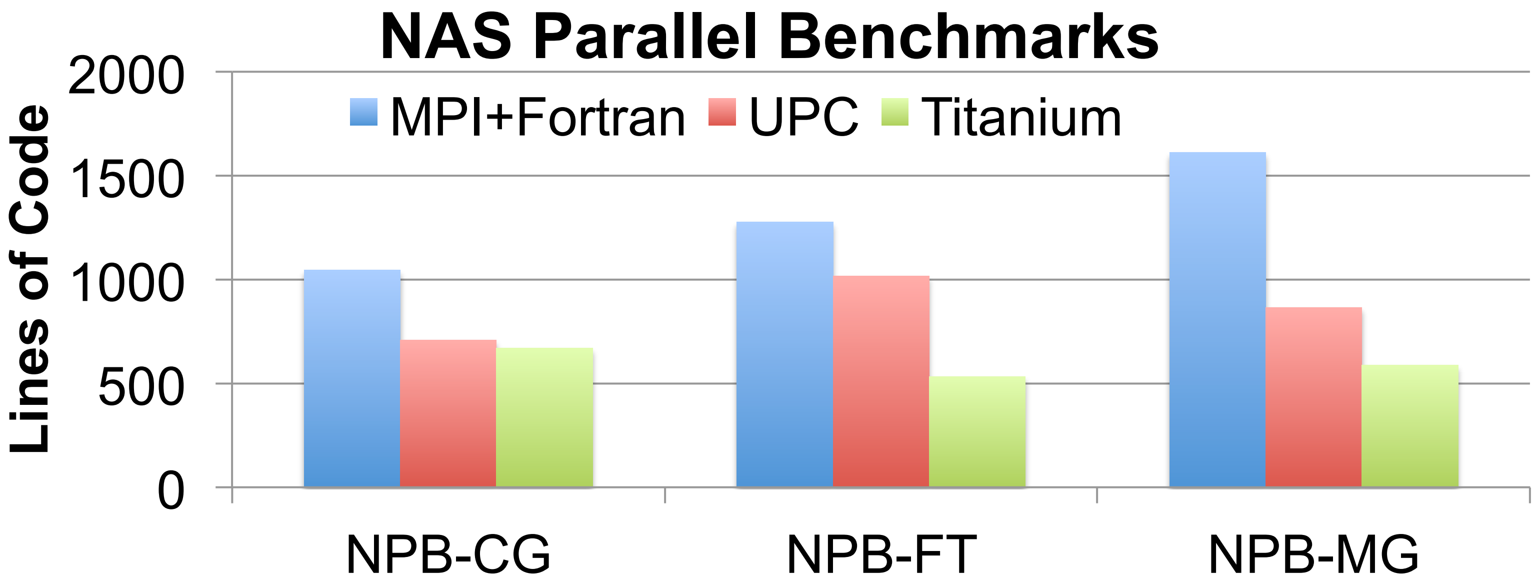
- Leverage C++ standards and compilers
  - Implement UPC++ as a C++ template library
  - C++ templates can be used as a mini-language to extend C++ syntax
- New features in C++11 are very useful
  - E.g., type inference, variadic templates, lambda functions, rvalue references
  - However, C++11 is not required by UPC++

UPC++ Multidimensional Arrays

- True multidimensional arrays with sizes specified at runtime
- Support subviews without copying (e.g. view of interior)
- Can be created over any rectangular index space, with support for strides
  - Striding important for AMR and multigrid applications
- *Local-view* representation makes locality explicit and allows arbitrarily complex distributions
  - Each rank creates its own piece of the global data structure
- Allow fine-grained remote access as well as one-sided bulk copies

UPC++ Arrays Based on Titanium

- Titanium is a PGAS language based on Java
- Line count comparison of Titanium and other languages:



AMR Chombo	C++/Fortran/MPI	Titanium
AMR data structures	35000	2000
AMR operations	6500	1200
Elliptic PDE Solver	4200*	1500

* Somewhat more functionality in PDE part of C++/Fortran code

BERKELEY LAB
Lawrence Berkeley National Laboratory

Titanium vs. UPC++

- Main goal: provide similar productivity and performance as Titanium in UPC++
- Titanium is a language with its own compiler
  - Provides special syntax for indices, arrays
  - PhD theses have been written on compiler optimizations for multidimensional arrays (e.g. Geoff Pike specifically for Titanium)
- Primary challenge for UPC++ is to provide Titanium-like productivity and performance in a library
  - Use macros, templates, and operator/function overloading for syntax
  - Provide specializations for performance

Overview of UPC++ Array Library

- A *point* is an index, consisting of a tuple of integers

```
point<2> lb = {{1, 1}}, ub = {{10, 20}};
```

- A *rectangular domain* is an index space, specified with a lower bound, upper bound, and optional stride

```
rectdomain<2> r(lb, ub);
```

- An array is defined over a rectangular domain and indexed with a point

```
ndarray<double, 2> A(r); A[lb] = 3.14;
```

- One-sided copy operation copies all elements in the intersection of source and destination domains

```
ndarray<double, 2, global> B = ...;
B.async_copy(A); // copy from A to B
async_wait(); // wait for copy completion
```

Example: 3D 7-Point Stencil

- Code for each timestep:

```
// Copy ghost zones from previous timestep.
for (int j = 0; j < NEIGHBORS; j++)
  allA[neighbors[j]].async_copy(A.shrink(1));
async_wait(); // sync async copies
barrier(); // wait for puts from all nodes
// Local computation.
foreach (p, interior_domain)
  B[p] = WEIGHT * A[p] +
    A[p + PT(0, 0, 1)] + A[p + PT(0, 0, -1)] +
    A[p + PT(0, 1, 0)] + A[p + PT(0, -1, 0)] +
    A[p + PT(1, 0, 0)] + A[p + PT(-1, 0, 0)];
// Swap grids.
SWAP(A, B); SWAP(allA, allB);
```

View of interior of A

One-line copy

Special *foreach* loop iterates over arbitrary domain

Point constructor

Syntax of Points

- A **`point<N>`** consists of N coordinates
- The **`point`** class template is declared as plain-old data (POD), with an N-element array as its only member

```
template<int N> struct point {
  cint_t x[N];
  ...
};
```

  - Can be constructed using initializer list

```
point<2> lb = {{1, 1}};
```

- The **`PT`** function creates a point in non-initializer contexts

```
point<2> lb = PT(1, 1);
```

  - Implemented using variadic templates in C++11, explicit overloads otherwise

Array Template

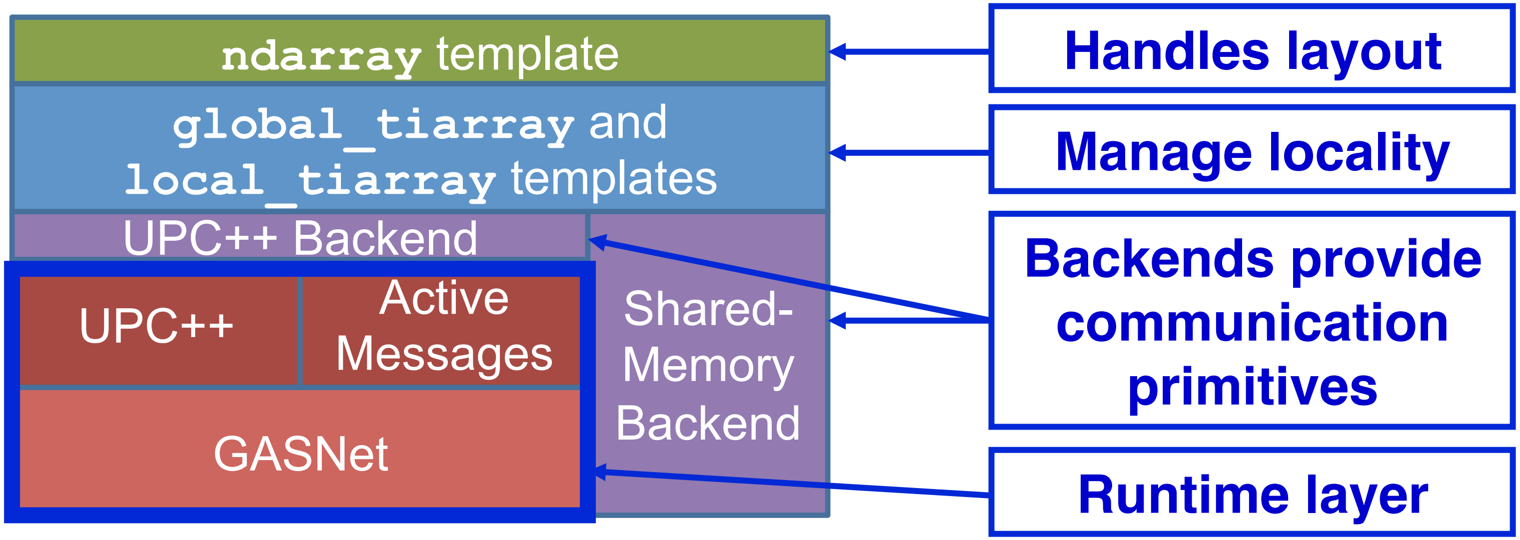
- Arrays represented using a class template, with element type and dimensionality arguments

```
template<class T, int N,
              class F1, class F2>
class ndarray;
```

- Last two (optional) arguments specify locality and layout
  - Locality can be **`local`** (i.e. elements are located in the local memory space) or **`global`** (elements may be located elsewhere)
  - Layout can be **`strided`**, **`unstrided`**, **`simple`**, **`simple_column`**; more details later
- Template metaprogramming used to encode type lattices for implicit conversions

Array Implementation

- Local and global arrays have significant differences in their implementation
  - Global arrays may require communication
- Layout only affects indexing
- Implementation strategy:



- Macros and template metaprogramming used to interface between layers

Foreach Implementation

- Macros allow definition of **`foreach`** loops
- C++11 implementation using type inference:

```
#define foreach(p, dom)                               \
  foreach_(p, dom, UNIQUIFYN(foreach_ptr_, p))

#define foreach_(p, dom, ptr_)                        \
  for (auto ptr_ = (dom).iter(); !ptr_.done;          \
       ptr_.done = true)                              \
    for (auto p = ptr_.start(); ptr_.next(p);)
```

- Pre-C++11 implementation also possible using **`sizeof`** operator

Layout Specializations

- Arrays can be created over any logical domain, but are laid out contiguously
  - Physical domain may not match logical domain
  - Non-matching stride requires division to get from logical to physical

```
(px[0] - base[0])*side_factors[0]/stride[0] +
(px[1] - base[1])*side_factors[1]/stride[1] +
(px[2] - base[2])*side_factors[2]/stride[2]
```

- Introduce template specializations to restrict layout
  - **`strided`**: any logical or physical stride
  - **`unstrided`**: logical and physical strides match
  - **`simple`**: matching strides + row-major format
  - **`simple_column`**: matching strides + column-major

Loop Specializations

- A **`foreach`** loop is implemented as an iterator over the points in a domain
- Loop over multidimensional array requires full index computation in each iteration

```
(px[0] - base[0])*side_factors[0]/stride[0] +
(px[1] - base[1])*side_factors[1]/stride[1] +
(px[2] - base[2])*side_factors[2]/stride[2]
```

- Solution: implement specialized *N*-D **`foreachN`** loops that translate into *N* nested **`for`** loops
  - Declare *N* integer indices rather than a point
  - Allow compiler to lift parts of index expression

Example: CG SPMV

- Unspecialized local SPMV in conjugate gradient kernel

```
void multiply(ndarray<double, 1> output,
              ndarray<double, 1> input) {
  double sum = 0;
  foreach (i, lrowRectDomains.domain()) {
    sum = 0;
    foreach (j, lrowRectDomains[i]) {
      sum += la[j] * input[lcolidx[j]];
    }
    output[i] = sum;
  }
}
```

- 3x slower than hand-tuned code (sequential PGCC on Cray XE6)

BERKELEY LAB

Example: CG SPMV

- Specialized local SPMV

```
void multiply(ndarray<double, 1, simple> output,
              ndarray<double, 1, simple> input) {
  double sum = 0;
  foreach1 (i, lrowRectDomains.domain()) {
    sum = 0;
    foreach1 (j, lrowRectDomains[i]) {
      sum += la[j] * input[lcolidx[j]];
    }
    output[i] = sum;
  }
}
```

- Comparable to hand-tuned code (sequential PGCC on Cray XE6)

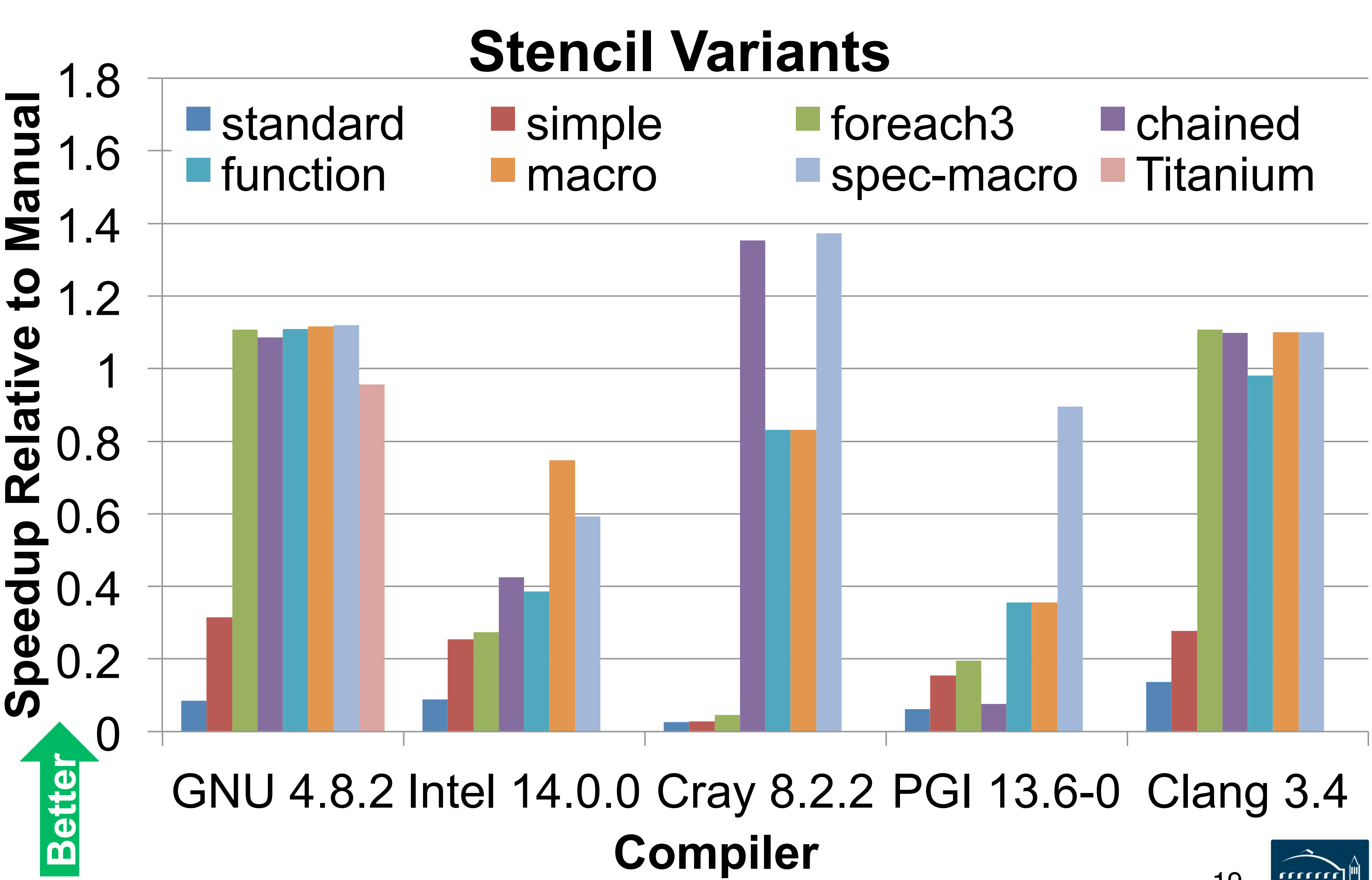
Indexing Options

- Must rely on C++ compiler to optimize indexing
- Some compilers have trouble with point indexing, so we provide many alternatives
  - Point indexing: `A[PT(i, j, k)]`
  - Chained indexing: `A[i][j][k]`
  - Function-call syntax: `A(i, j, k)`
  - Macros: `AINDEX3(A, i, j, k)`
  - Specialized macros: `AINDEX3_simple(A, i, j, k)`
- Latter two alternatives require preamble before loop:

```
AINDEX3_SETUP(A);
```

- Arrays can also be manually indexed using data pointer

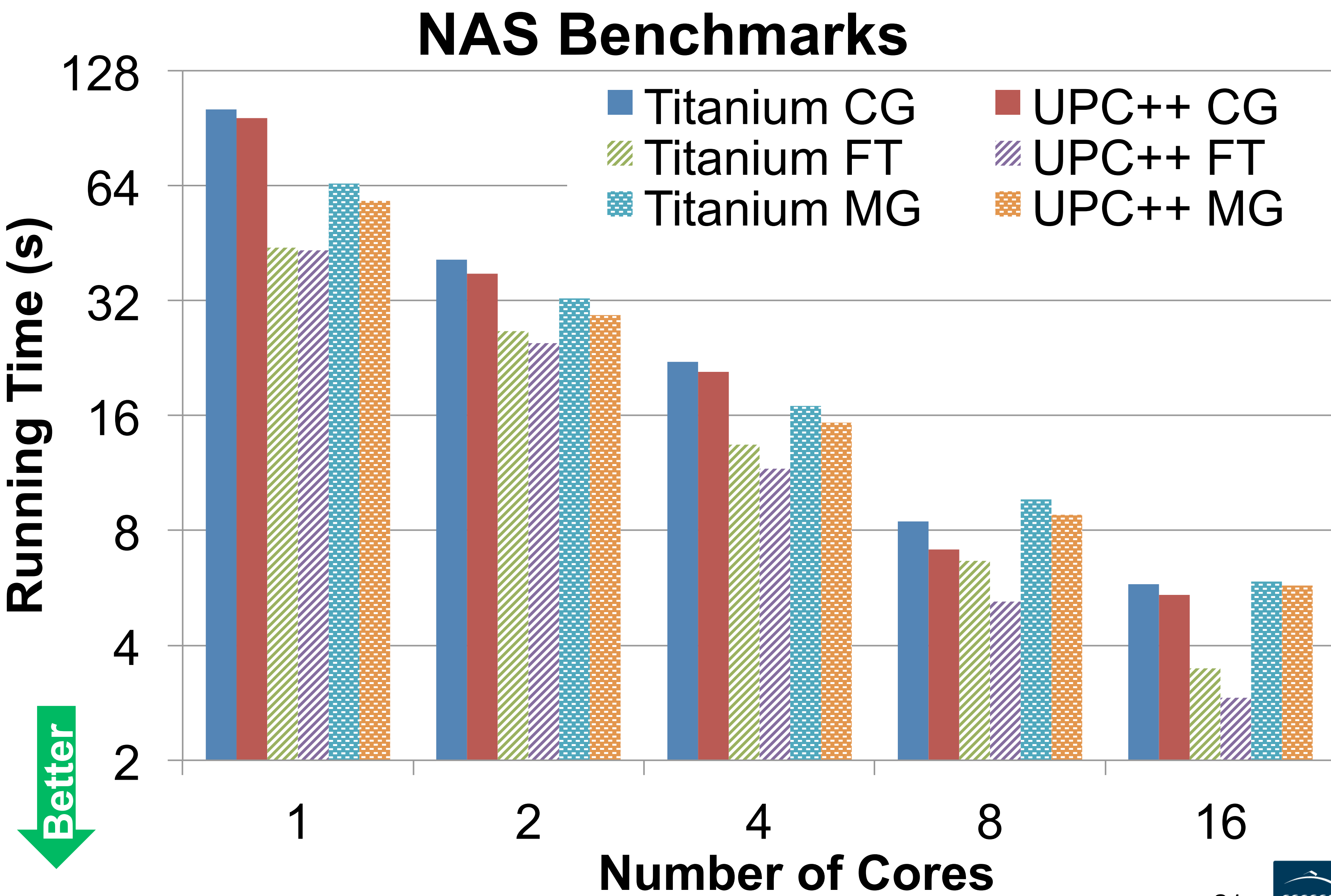
Specializations and Indexing in Stencil



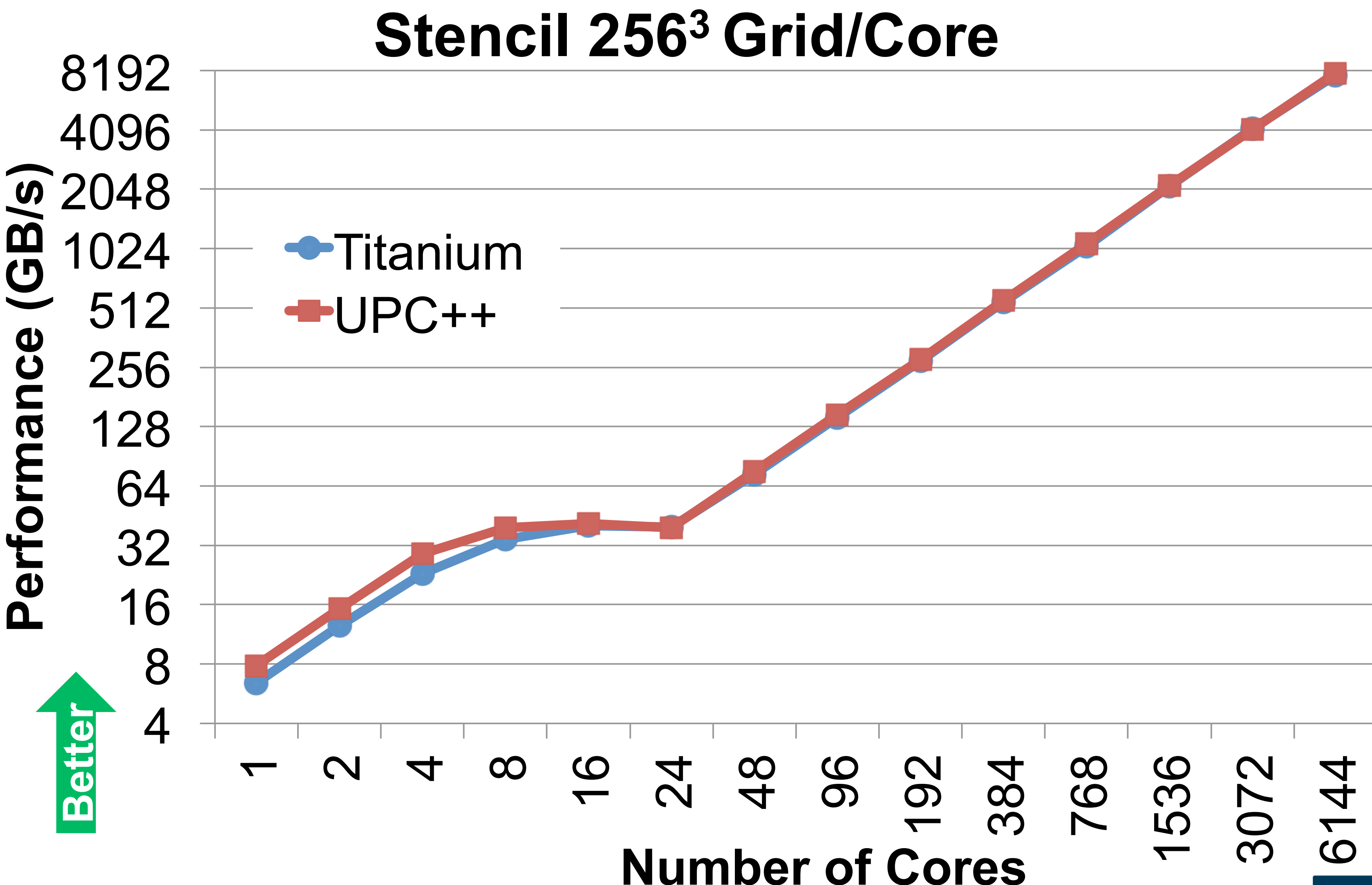
Evaluation

- Evaluation of array library done by porting benchmarks from Titanium to UPC++
  - Again, goal is to match Titanium's productivity and performance without access to a compiler
- Benchmarks: 3D 7-point stencil, NAS CG, FT, and MG
- Minimal porting effort for these examples, providing some evidence that productivity is similar to Titanium
  - Less than a day for each kernel
  - Array code only requires change in syntax
  - Most time spent porting Java features to C++

NAS Benchmarks on One Node



Stencil Weak Scaling



Conclusion

- We have built a multidimensional array library for UPC++
  - Macros and template metaprogramming provide a lot of power for extending the core language
  - UPC++ arrays can provide the same productivity gains as Titanium
  - Specializations allow UPC++ to match Titanium's performance

- Future work
  - Improve performance of one-sided array copies
    - Stencil code is about 10% slower than MPI
  - Build global-view distributed array library on top of current local-view library