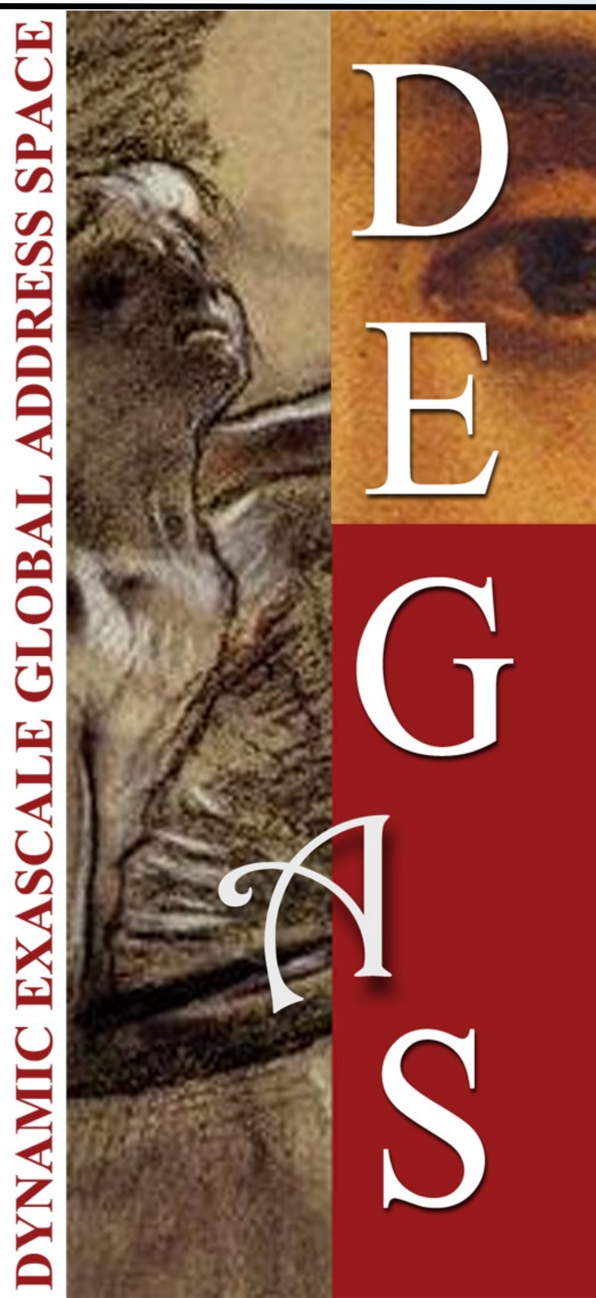


Evaluation of PGAS Communication Paradigms With Geometric Multigrid

Hongzhang Shan, **Amir Kamil**, Samuel Williams, Yili Zheng, and Katherine Yelick
Lawrence Berkeley Lab
Berkeley, CA, USA

October 10, 2014

Overview

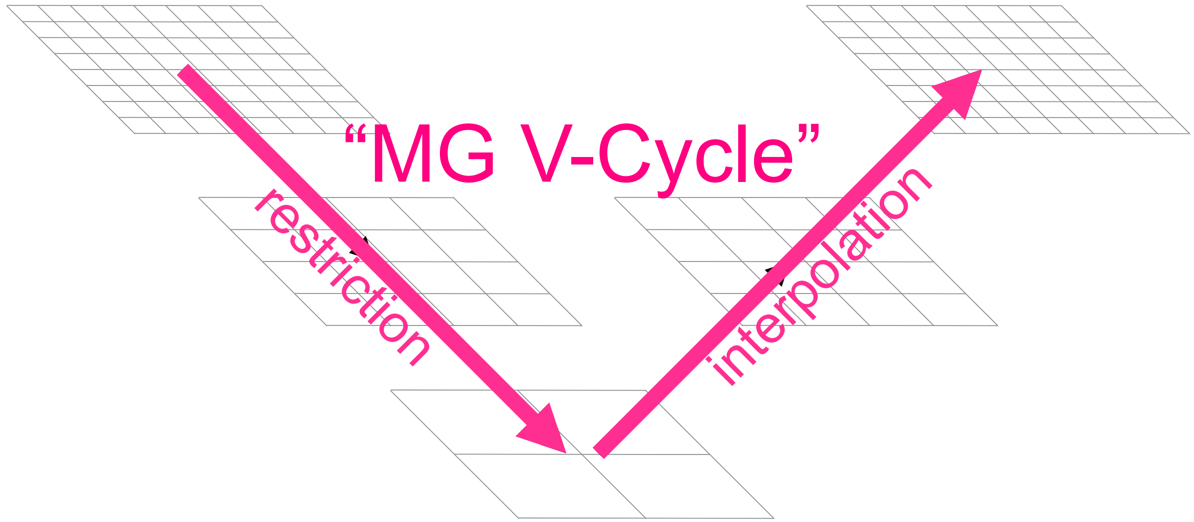
- We evaluate the productivity and performance of three implementations of **miniGMG**, a multigrid benchmark
- The three implementations use different communication strategies enabled by the PGAS model
  1. Fine-grained communication, at the natural granularity of the algorithm
  2. Bulk communication, with manual packing and unpacking by the user
     - One-sided analogue of message passing
  3. Higher-level array-based communication that offloads the work to an array library
     - Still semantically one-sided
- We evaluate performance on two current platforms

Implementation Strategy

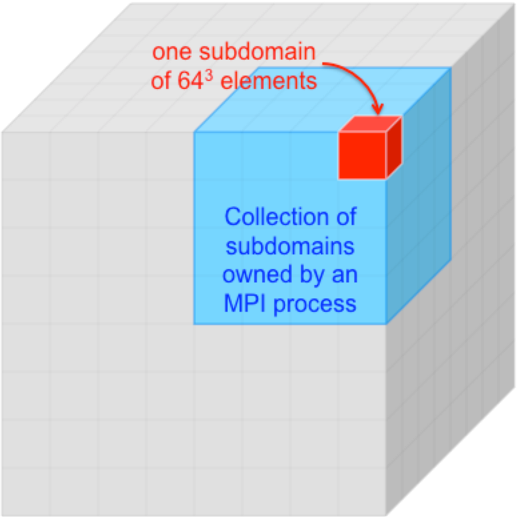
- We use UPC++ to implement the three algorithms
  - C++ library that implements the PGAS model
  - Provides UPC-like shared arrays, which simplify coordination between ranks but can still scale to hundreds of thousands of ranks
  - Includes a multidimensional array library that supports fine-grained and bulk remote access
  - Seamlessly interoperates with OpenMP, MPI, and other parallel libraries
- We do not claim in this work that UPC++ is superior to MPI or any other system
  - Main focus is to evaluate alternative communication algorithms
  - Results applicable to other PGAS implementations

Multigrid Overview

- Linear Solvers (**`Ax=b`**) are ubiquitous in scientific computing
  - Combustion, Climate, Astrophysics, Cosmology, etc.
- Multigrid exploits the nature of elliptic PDEs to provide a hierarchical approach with **O(N) computational complexity**
- **Geometric Multigrid** is specialization in which the linear operator (A) is simply a stencil on a structured grid (i.e. *matrix-free*)



miniGMG Overview



- 3D Geometric Multigrid benchmark designed to proxy MG solves in BoxLib and CHOMBO-based AMR applications
- Defines a cubical problem domain
  - Decomposed into cubical subdomains (boxes)
  - Rectahedral collections of subdomains are assigned to processes
  - Decomposition preserved across all levels of V-Cycle
- MPI+OpenMP parallelization
- Configured to use…
  - Fixed 10 U-Cycles (V-Cycle truncated when boxes are coarsened to 4^3)
  - 7-pt stencil with Gauss Seidel Red-Black (GSRB) smoother that requires nearest-neighbor communication for each smooth or residual calculation.
  - BiCGStab coarse-grid (bottom) solver
- Communication pattern is thus…
  - Fixed 6 nearest-neighbor communication
  - Message sizes vary greatly as one descends through the V-Cycle (128KB -> 128 bytes -> 128KB)
  - Requires neighbor synchronization on each step (e.g. two-sided MPI)

UPC++ Overview

- A C++ PGAS extension that combines features from:
  - UPC: dynamic global memory management and one-sided communication (put/get)
  - Titanium/Chapel/ZPL: multidimensional 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++ rank
  - OpenMP and CUDA can be easily mixed with UPC++ in the same way as MPI+X

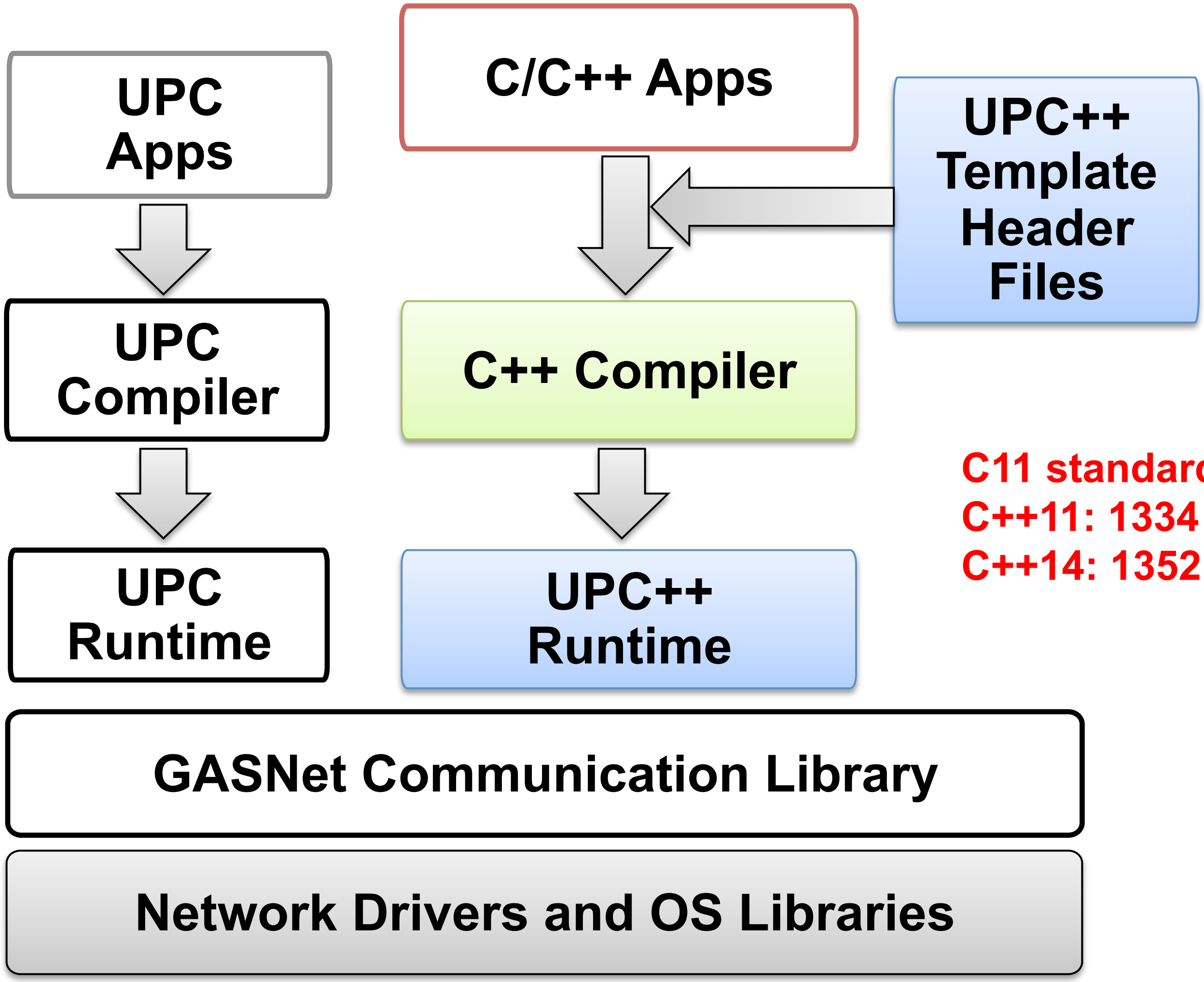
Related Work

- PGAS variants and extensions
  - AGAS, APGAS, APGNS, HPGAS…
- PGAS languages
  - CAF, Chapel, Habanero, X10, XscaleMP, UPC
- PGAS libraries
  - ARMCI, GASNet, Global Arrays, GASPI/GPI, MPI-3 RMA, OpenSHMEM, XPI
- Parallel C++ libraries (distributed-memory)
  - Charm++, Co-Array C++, DASH, HPX, HTA, Phalanx, STAPL…
- Parallel C++ libraries (shared-memory)
  - TBB, Thrust and many more

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
  - C++11 is well-supported by major compilers

UPC++ Software Stack



C11 standard: 701 pages
C++11: 1334 pages
C++14: 1352 pages

C++ Generic Programming for PGAS

- C++ templates enable generic programming
  - Parametric template definition

```
template<class T> struct array {
  T *elements;
  size_t sz;
};
```

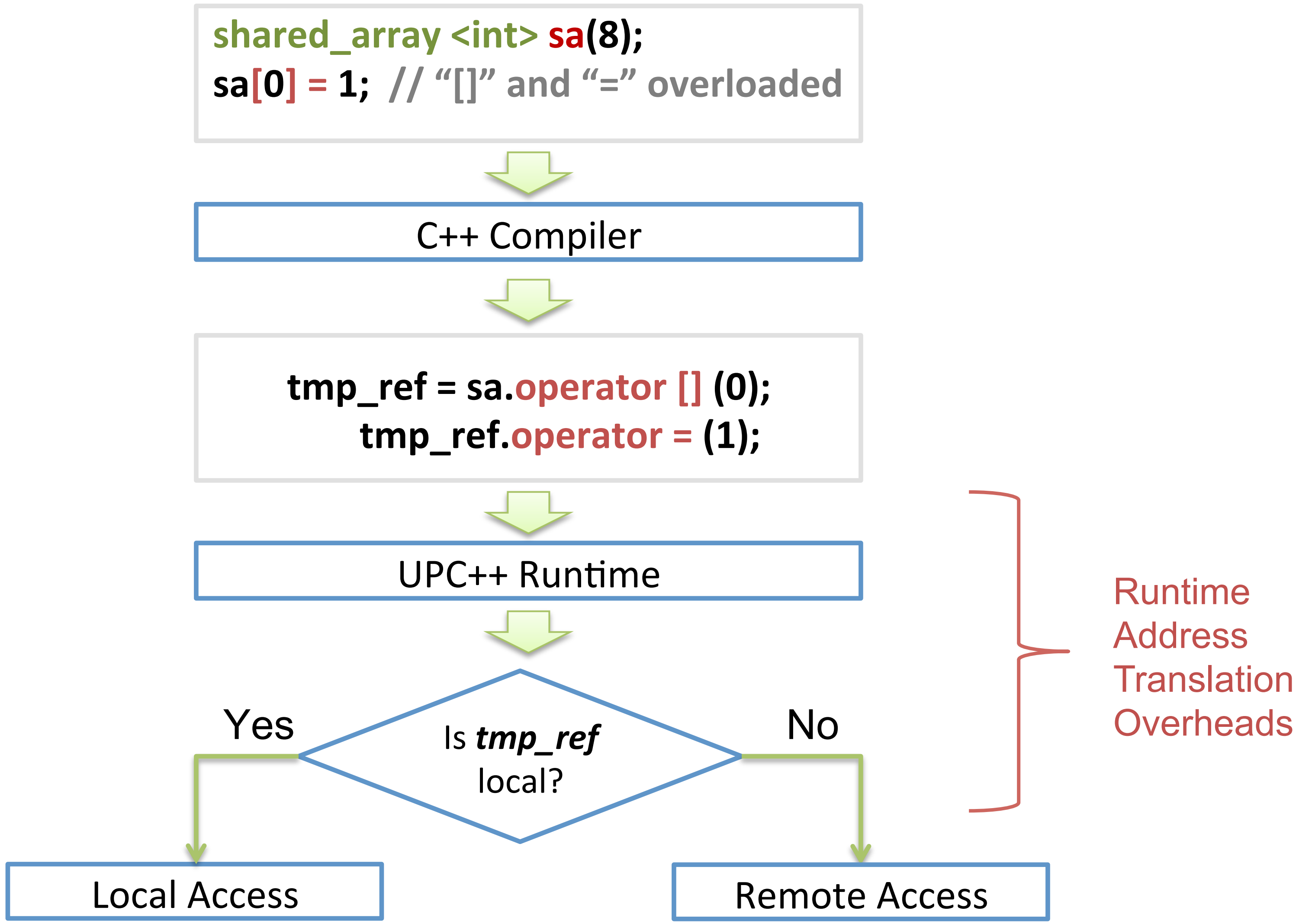
  - Template instantiation

```
array<double> A;
array<complex> B;
```

- UPC++ uses templates to express shared data

```
shared_var<int> s;  // shared int s in UPC
shared_array<int> sa(8); // shared int sa[8]
                         // in UPC
```

UPC++ Translation Example



One-Sided Data Transfer Functions

```
// Copy count elements of T from src to dst
upcxx::copy<T>(global_ptr<T> src,
               global_ptr<T> dst,
               size_t count);

// Non-blocking version of copy
upcxx::async_copy<T>(global_ptr<T> src,
                     global_ptr<T> dst,
                     size_t count);

// Synchronize all previous asyncs
upcxx::async_wait();
```

Similar to *upc_memcpy_nb* extension in UPC 1.3

UPC++ Equivalents for UPC Users

	UPC	UPC++
Num. of ranks	THREADS	THREADS or ranks()
My ID	MYTHREAD	MYTHREAD or myrank()
Shared variable	shared Type s	shared_var<Type> s
Shared array	shared [bf] Type A[sz]	shared_array<Type> A A.init(sz, bf)
Pointer-to-shared	shared Type *ptr	global_ptr<Type> ptr
Dynamic memory allocation	shared void * upc_alloc(nbytes)	global_ptr<Type> allocate<Type>(place, count)
Bulk data transfer	upc_memcpy(dst, src, sz)	copy<Type>(src, dst, count)
Affinity query	upc_threadof(ptr)	ptr.where()
Synchronization	upc_lock_t	shared_lock
	upc_barrier	barrier()

Multidimensional Arrays

- Multidimensional arrays are a common data structure in HPC applications
- However, they are poorly supported by the C family of languages, including UPC
  - Layout, indexing must be done manually by the user
  - No built-in support for subviews
- Remote copies of array subsets pose an even greater problem
  - Require manual packing at source, unpacking at destination
  - In PGAS setting, remote copies that are logically one-sided require two-sided coordination by the user

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
- *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

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

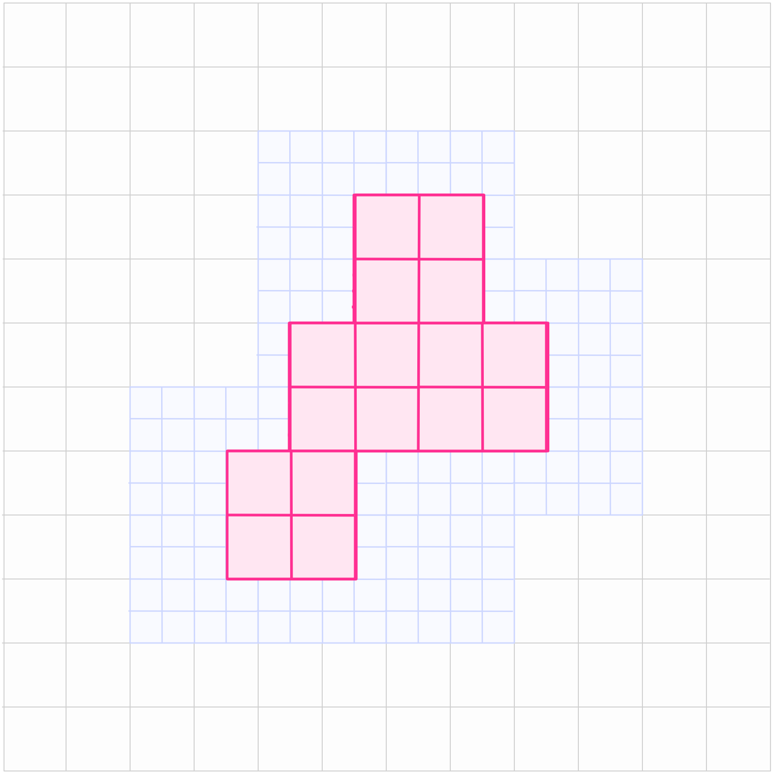
```
ndarray<double, 2> A(r); A[lb] = 3.14;
```

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

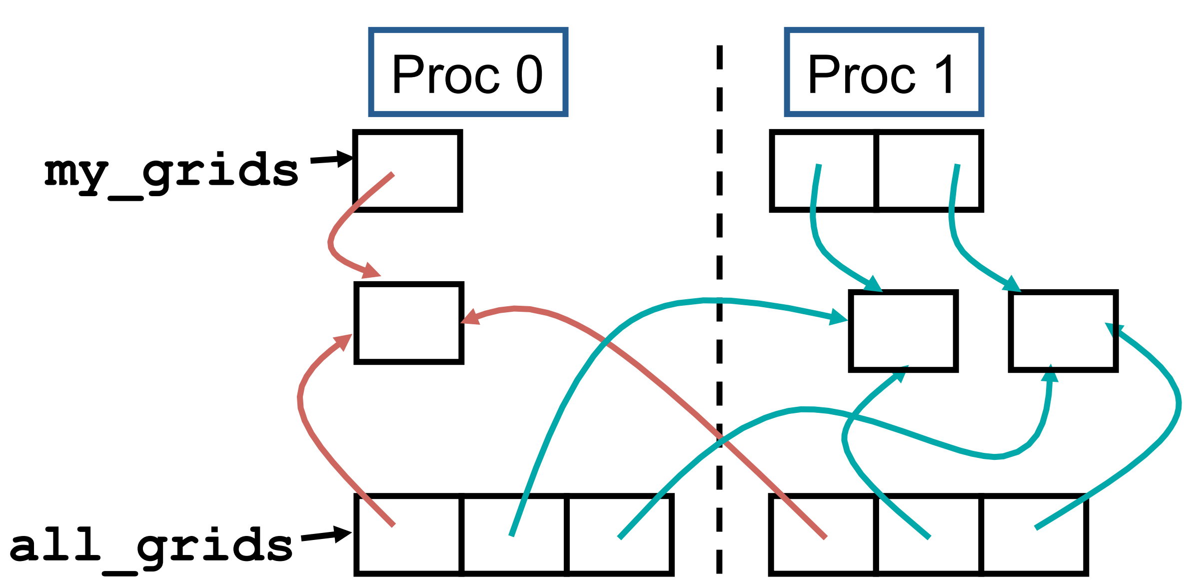
```
ndarray<double, 2, global> B = ...;
B.async_copy(A); // copy from A to B
async_wait(); // wait for copy completion
```

Arrays in Adaptive Mesh Refinement

- AMR starts with a coarse grid over the entire domain
- Progressively finer AMR levels added as needed over subsets of the domain
- Finer level composed of union of regular subgrids, but union itself may not be regular
- Individual subgrids can be represented with UPC++ arrays
- Directory structure can be used to represent union of all subgrids



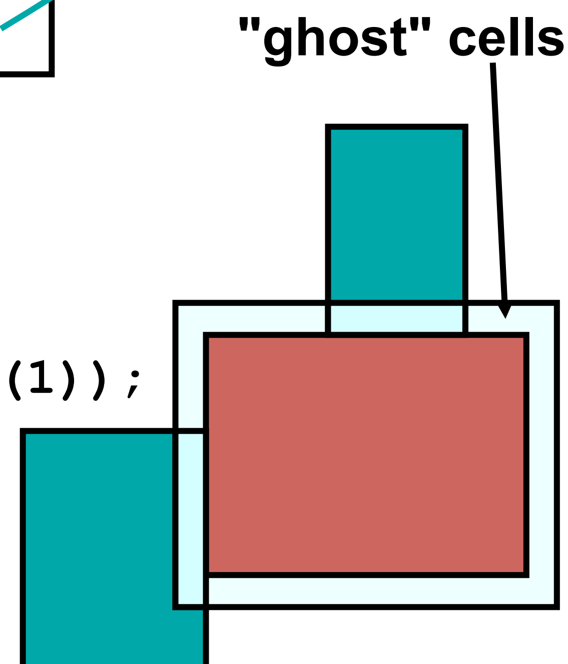
Example: Ghost Exchange in AMR




```
foreach (l, my_grids.domain())
 foreach (a, all_grids.domain())
  if (l != a)
   my_grids[l].copy(all_grids[a].shrink(1));
```

Avoid null copies (refers to `if (l != a)`)

Copy from interior of other grid (refers to `.shrink(1)`)



- Can allocate arrays in a global index space
- Let library compute intersections

Array Creation in miniGMG

```
void create_grid(..., int li, int lk, int lk, int szi,
                 int szj, int szk, int ghosts) {
  ...
  double *grid = upcxx::allocate<double>(...);

  rectdomain<3> rd(PT(li-ghosts, lj-ghosts, lk-ghosts),
                   PT(li+szi+ghosts, lj+szj+ghosts,
                      lk+szk+ghosts));
  point<3> padding = ...;
  ndarray<double, 3> garray(grid, rd, true, padding);
  ...
}
```

Existing Grid Creation Code

Logical Domain of Grid

Padding of Grid Dimensions

Create Array Descriptor over Existing Grid Memory

Grid Domain

Column-Major Layout

Padding

Communication Setup for miniGMG Arrays

```
point<3> dirs = {{ di, dj, dk }}, p0 = {{ 0, 0, 0 }};
```

Circular Domain Shift at Boundaries

```
for (int d = 1; d <= 3; d++) {
  if (dirs[d] != 0)
    dst = dst.border(ghosts, -d * dirs[d], 0);
  if (dirs[d] == -1 && src.domain().lwb()[d] < 0)
    src = src.translate(p0.replace(d, dst.domain().upb()[d] -
                                            ghosts));
  else if (dirs[d] == 1 && dst.domain().lwb()[d] < 0)
    src = src.translate(p0.replace(d, -src.domain().upb()[d] +
                                            ghosts));
}
```

Compute Intersection

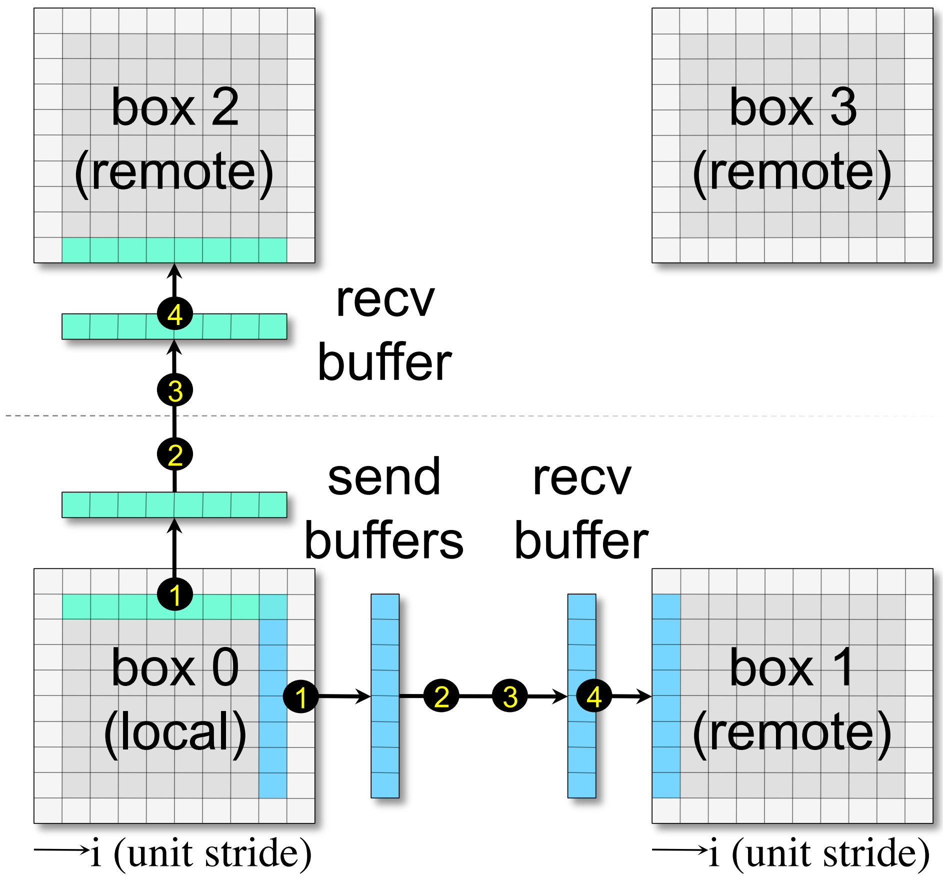
```
rectdomain<3> isct = dst.domain()*src.domain().shrink(ghosts);
```

Save Views of Source and Destination Restricted to Intersection

```
send_arrays[PT(level, g, nn, i, j, k)] = src.constrict(isct);
recv_arrays[PT(level, g, nn, i, j, k)] = dst.constrict(isct);
```

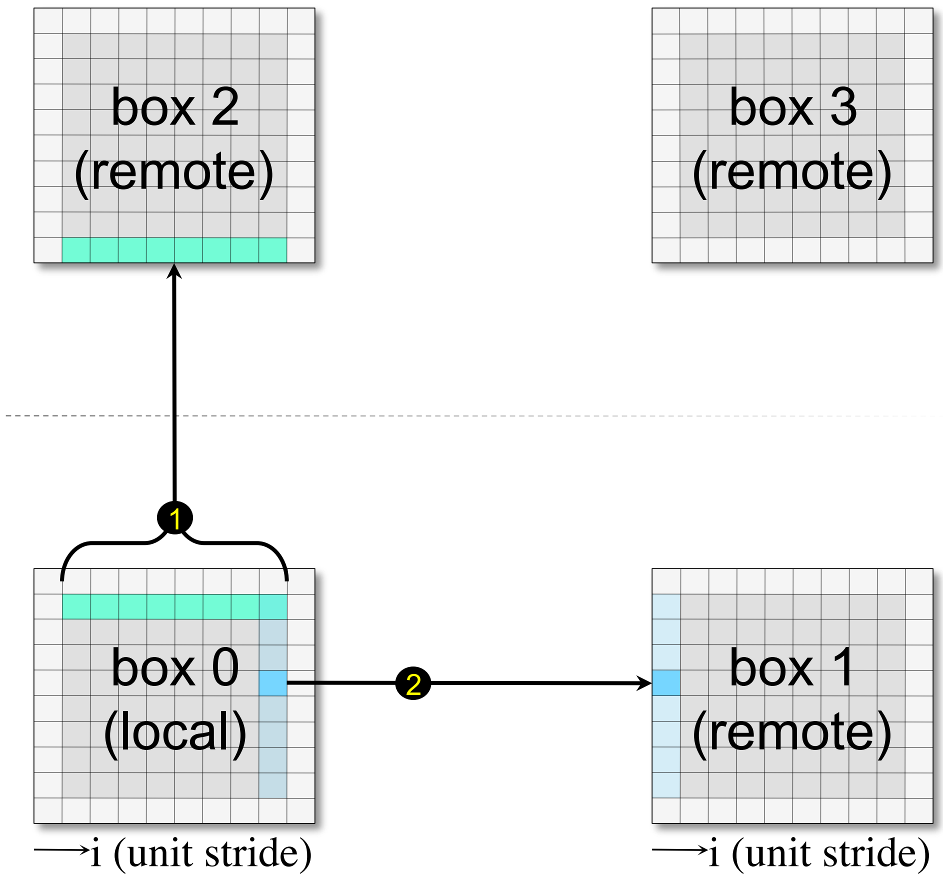
Bulk Communication Strategy

- ***Bulk*** version uses manual packing/unpacking
  - Similar to MPI code, but with one-sided puts instead of two-sided messaging



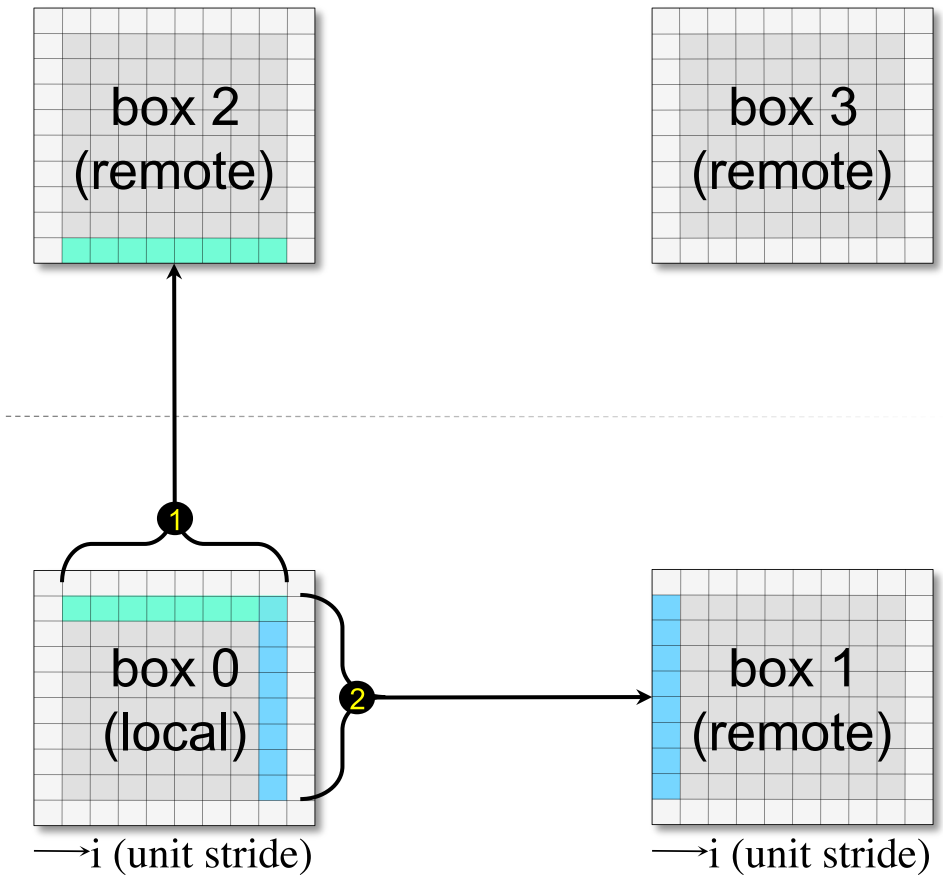
Fine-Grained Communication Strategy

- ***Fine-Grained*** version does multiple one-sided puts of contiguous data
  - Puts are at natural granularity of the algorithm



Array Communication Strategy

- ***Array*** version logically copies entire ghost zones, delegating actual procedure to array library
  - Copies have one-sided semantics



Communication Coordination

- Shared array used to coordinate communication

```
shared_array<global_ptr<subdomain_type>, 1>
global_boxes;
```

- Bulk version must carefully coordinate send and receive buffers between ranks
  - Must ensure right buffers are used, same ordering for packing and unpacking elements
  - Special cases for ghost zones at faces, edges, and corners
  - Most difficult part of code
- Minimal coordination required for fine-grained and array
  - Only need to obtain location of target grid from shared array

Ghost-Zone Exchange Algorithms

	Bulk	Fine-Grained	Array
Barrier	Yes	Yes	Yes
Pack	Yes	No	No
Async Puts/ Copies	1 per neighboring rank	1 for each contiguous segment	1 per neighboring grid
Async Wait	Yes	Yes	Yes
Barrier	Yes	Yes	Yes
Unpack	Yes	No	No
~ Line Count of Setup + Exchange	884	537	399

- Pack/unpack parallelized using OpenMP in bulk version
  - Effectively serialized in fine-grained and array

Bulk Copy Code

- Packing/unpacking code in bulk version:

```
...
for (int k = 0; k < dim_k; k++) {
  for (int j = 0; j < dim_j; j++) {
    for (int i = 0; i < dim_i; i++) {
      int read_ijk  = (i+ read_i) + (j+ read_j)*
           read_pencil + (k+ read_k)* read_plane;
      int write_ijk = (i+write_i) + (j+write_j)*
          write_pencil + (k+write_k)*write_plane;
      write[write_ijk] = read[read_ijk];
    }
  }
}
```

- Code must be run on both sender and receiver

Fine-Grained Copy Code

- Fine-grained code matches shared-memory code, but with **`async_copy`** instead of **`memcpy`**:

```
...
for (int k = 0; k < dim_k; k++)
  for (int j = 0; j < dim_j; j++) {
    int roff = recv_i + (j+recv_j)*rpencil +
      (k+recv_k)*rplane;
    int soff = send_i + (j+send_j)*spencil +
      (k+send_k)*splane;
    async_copy(sbuf+soff, rbuf+roff, dim_i);
  }
}
```

- Takes advantage of fact that source and destination layouts match

Array Copy Code

- Array version delegates actual copies to array library:

```
rcv = recv_arrays[PT(level, g, nn, i, j, k)];
rcv.async_copy(send_arrays[PT(level, g, nn, i, j, k)]);
```

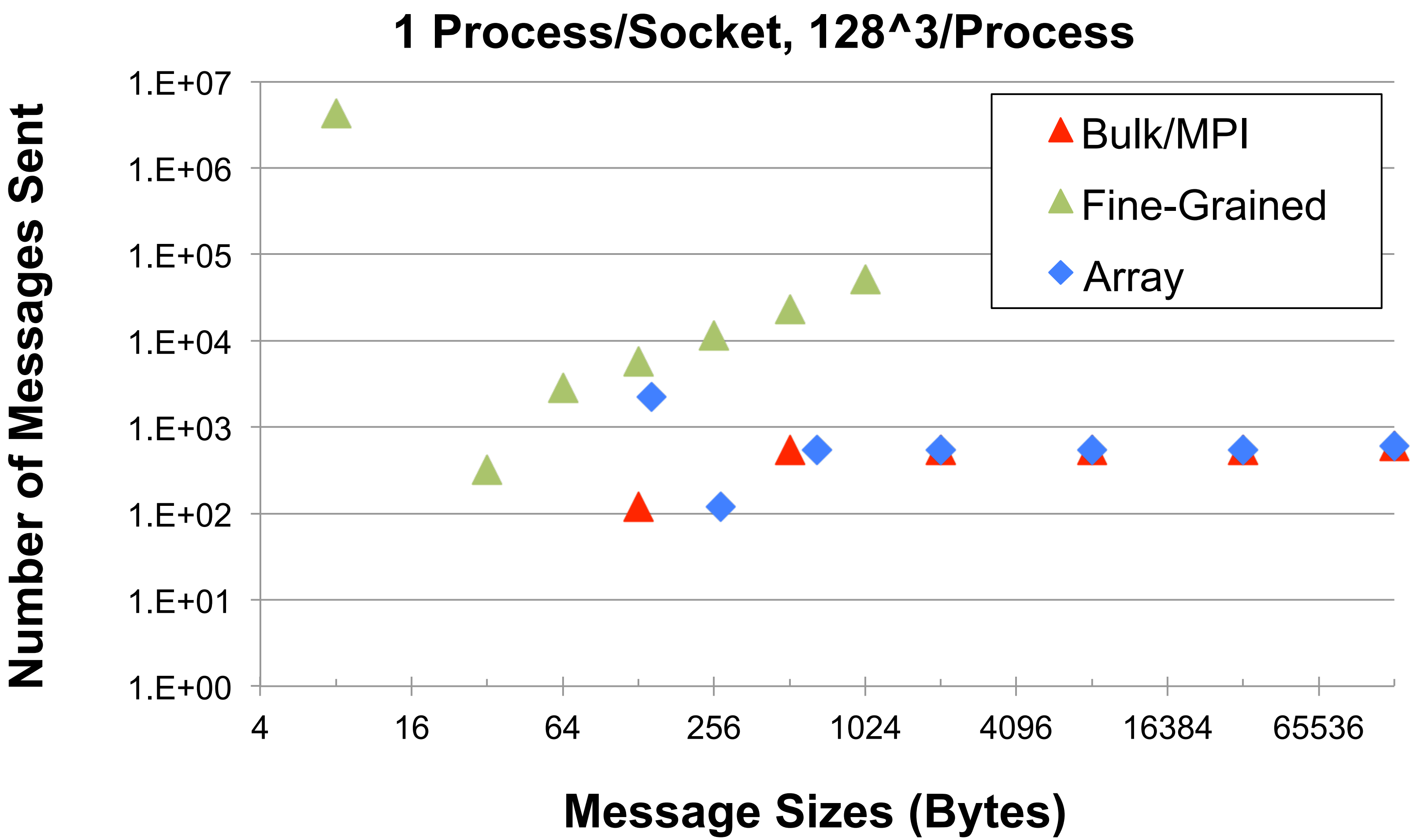
- Array library behavior for cases that occur in miniGMG:
  1. If the source and destination are contiguous, then one-sided put directly transfers data
  2. Otherwise, elements packed into contiguous buffer on source
     a) If the elements and array metadata fit into a medium active message (AM), a medium AM is initiated
        – AM handler on remote side unpacks into destination
     b) Otherwise, a short AM is used to allocate a remote buffer
        – Blocking put transfers elements to remote buffer
        – Medium AM transfers array metadata
        – AM handler on remote side unpacks and deallocates buffer

Platforms and Experimental Setup

- Cray XC30 (Edison), located at NERSC
  - Cray Aries Dragonfly network
  - Each node has two 12-core sockets
  - We use 8 threads/socket
- IBM Blue Gene/Q (Mira), located at Argonne
  - 5D torus network
  - Each node has 16 user cores, with 4 threads/core
  - We use 64 threads/socket
- Fixed (weak-scaling) problem size of 128^3 grid/socket
- Two experiments on each platform
  - 1 MPI process, 8 or 64 OpenMP threads per socket
  - 8 MPI processes, 1 or 8 OpenMP threads per socket

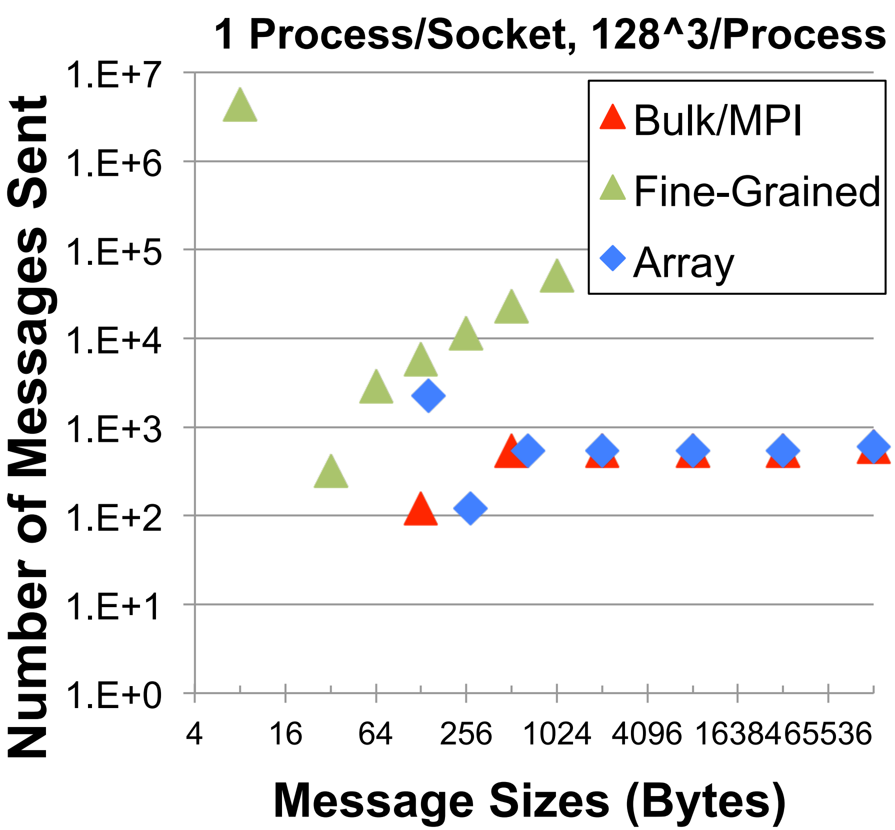
Communication Histogram

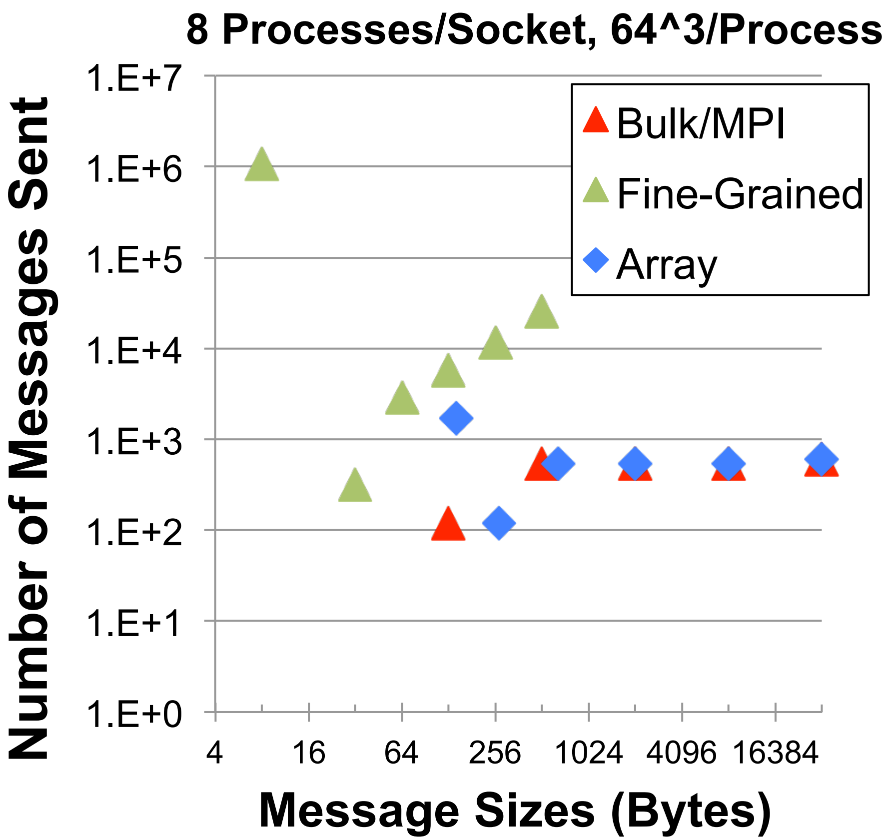
- Histogram of message sizes per process, when using 1 process/socket, for all three versions on Cray XC30



Histogram of 1 MPI Process vs. 8/Socket

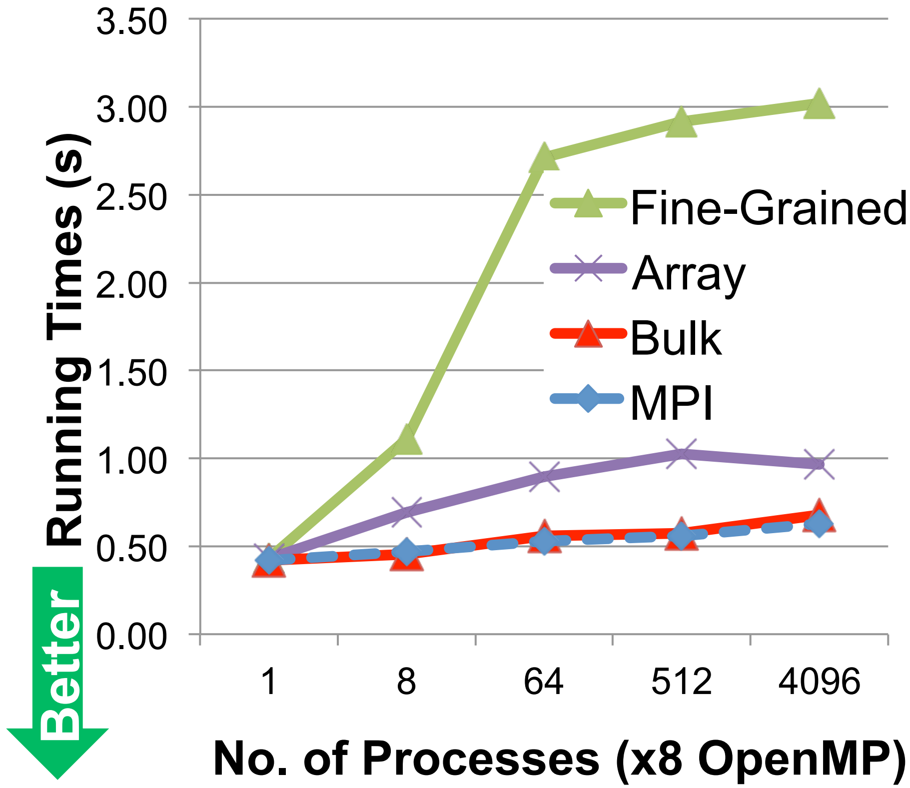
- Same overall problem size per socket
- Fewer small messages per process when using 8 processes, but more small messages per socket

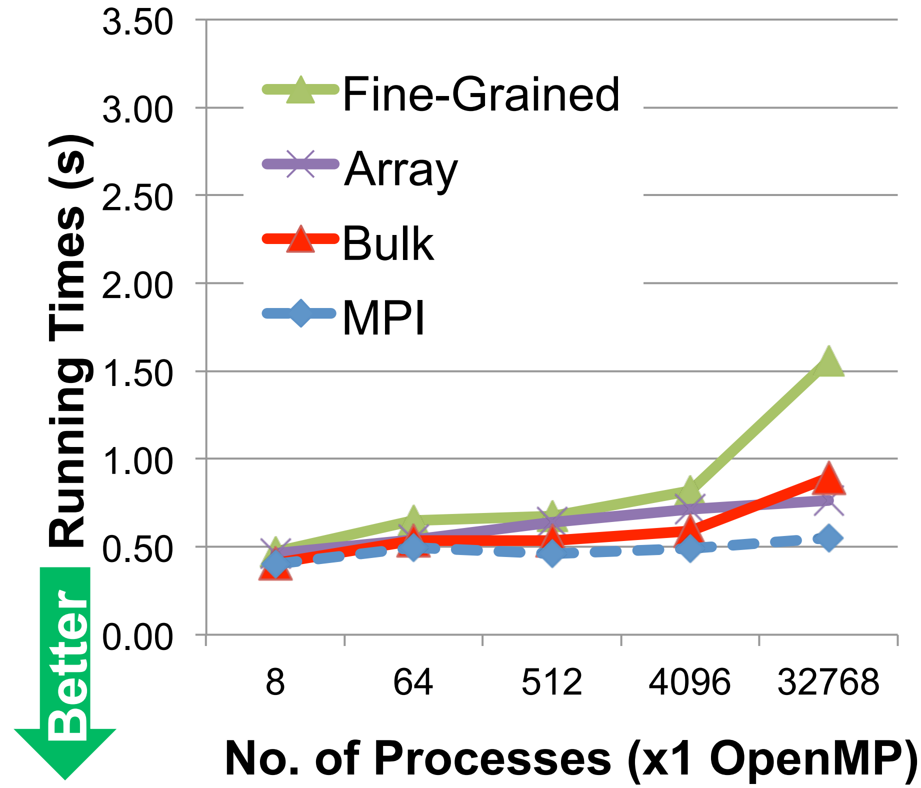




Performance Results on Cray XC30

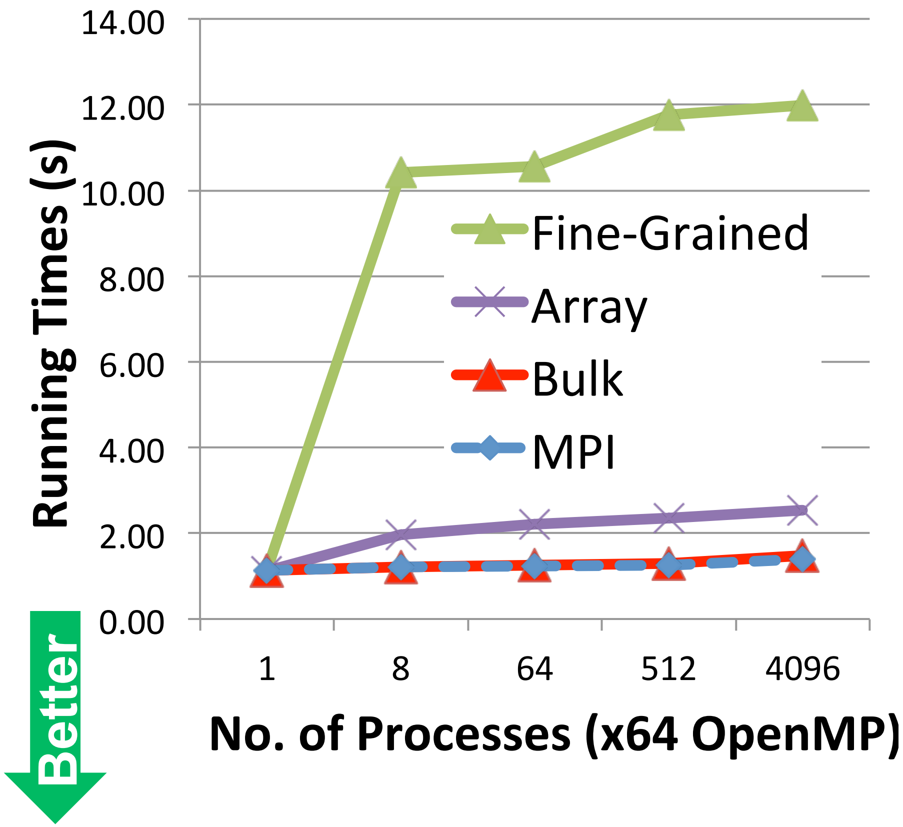
- Fine-grained and array versions do much better with higher injection concurrency
  - Array version does not currently parallelize packing/ unpacking, unlike bulk/MPI

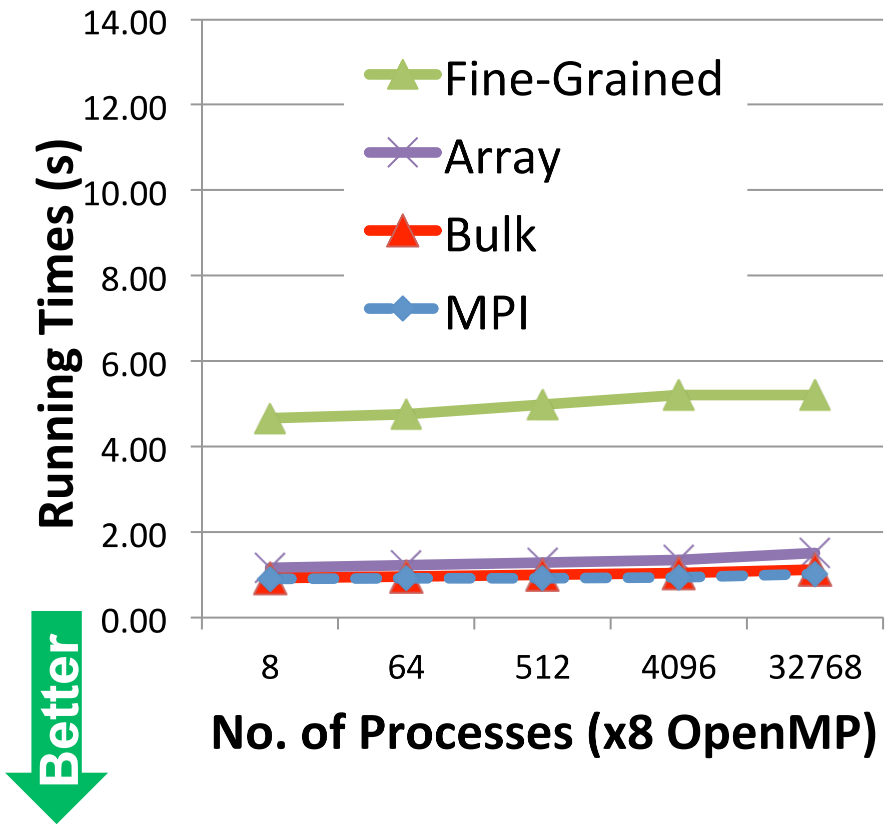




Performance Results on IBM Blue Gene/Q

- Fine-grained does worse, array better on IBM than Cray
- Using more processes improves fine-grained and array performance, but fine-grained still significantly slower





Summary of Results

- Array abstraction can provide better productivity than even fine-grained, shared-memory-style code, while getting close to bulk performance
  - Unlike bulk, array code doesn't require two-sided coordination
  - Further optimization (e.g. parallelize packing/unpacking) can reduce the performance gap between array and bulk
  - Existing code can be easily rewritten to take advantage of array copy facility, since changes localized to communication part of code
- Fine-grained code not as bad as expected
  - 3x slowdown over bulk at scale on Cray XC30, 5x on IBM BG/Q, when using multiple processes/socket
  - On manycore machines, fine-grained performance will be crucial, since there will be significantly less memory/core