Kathy Yelick\textsuperscript{1,2}, Yili Zheng\textsuperscript{1}, Amir Kamil\textsuperscript{1,3}

\textsuperscript{1}Lawrence Berkeley National Laboratory
\textsuperscript{2}Department of EECS, UC Berkeley
\textsuperscript{3}Department of EECS, University of Michigan

PGAS 2015 Tutorial
September 16, 2015
Programming Challenges and Solutions

Message Passing Programming
- Divide up domain in pieces
- Each compute one piece
- Exchange (send/receive) data

PVM, MPI, and many libraries

Global Address Space Programming
- Each start computing
- Grab whatever you need whenever

Global Address Space Languages and Libraries

~10% of NERSC apps use some kind of PGAS-like model
Parallel Programming Problem: Histogram

• Consider the problem of computing a histogram:
  – Large number of “words” streaming in from somewhere
  – You want to count the # of words with a given property

• In shared memory
  – Lock each bucket

<table>
<thead>
<tr>
<th>A’s</th>
<th>B’s</th>
<th>C’s</th>
<th>…</th>
<th>Y’s</th>
<th>Z’s</th>
</tr>
</thead>
</table>

• Distributed memory: the array is huge and spread out
  – Each processor has a substream and sends +1 to the appropriate processor… and that processor “receives”

| A’s | B’s | C’s | D’s | … | Y’s | Z’s |
PGAS = Partitioned Global Address Space

- **Global address space**: thread may directly read/write remote data
  - Convenience of shared memory
- **Partitioned**: data is designated as local or global
  - Locality and scalability of message passing
UPC: production (multiple compilers)

CAF: Features in FORTRAN standard

PGAS

UPC++: DEGAS research (HPGAS)

Others: Chapel, Titanium, X10, …
UPC++ Features

- Function shipping across nodes
- Multidimensional arrays
- Local task queue
- Multi-threading
- Private address space
- Global address space
UPC++: PGAS with “Mixins”

- UPC++ uses templates (no compiler needed)
  ```cpp
  shared_var<int> s;
  global_ptr<LLNode> g;
  shared_array<int> sa(8);
  ```
- Default execution model is SPMD, but

- Remote methods, async
  ```cpp
  async(place) (Function f, T1 arg1,…);
  async_wait(); // other side does poll();
  ```

- Research in teams for hierarchical algorithms and machines
  ```cpp
  teamsplit (team) { ... }
  ```

- Interoperability is key; UPC++ can be use with OpenMP or MPI
Why Should You Care about PGAS?

Latency between Two MICs via Infiniband

Point-to-Point Latency Comparison on Edison (Cray XC30)

Why Should You Care about PGAS?
Random Access to Large Memory

Meraculous Genome Assembly Pipeline

- Remote Atomics
- Dynamic Aggregation
- Software Caching
- Fast I/O (HDF5)
- Bloom filters, locality-aware hashing,…

Contig generation step:
- Human: 44 hours to 20 secs
- Wheat: “doesn’t run” to 32 secs

Grand Challenge: Metagenomes

~20% of Edison @ NERSC can assemble all human genomes produced worldwide in 2015
UPC++ Execution Model
UPC++ Basics

• UPC++ reserves all names that start with UPCXX or upcxx, or that are in the upcxx namespace

• Include “upcxx.h” for using UPC++

• Init and finalize the runtime

  int upcxx::init(&argc, &argv);
  int upcxx::finalize();

• Number of processes in the parallel job and my ID

  uint32_t upcxx::ranks(); // THREADS in UPC
  uint32_t upcxx::myrank(); // MYTHREAD in UPC

Tip: Add “using namespace upcxx;” to save typing “upcxx::”
Hello World in UPC++

- Any legal C/C++ program is also a legal UPC++ program
- If you compile and run it with P processes, it will run P copies of the program, also known as SPMD

```cpp
#include <upcxx.h>
#include <iostream>

using namespace upcxx; // save typing “upcxx::”

int main (int argc, char **argv)
{
    init(&argc, &argv); // initialize UPC++ runtime
    std::cout << "Hello, I'm rank " << myrank() << " of " << ranks() << ".\n";
    finalize(); // shut down UPC++ runtime
    return 0;
}
```
Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $\frac{1}{4} \times \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - $\pi = 4 \times $ratio
Pi in UPC++ (ported from the UPC version)

- Independent estimates of pi:

```c
main(int argc, char **argv) {
    int i, hits, trials = 0;
    double pi;

    if (argc != 2) trials = 1000000;
    else trials = atoi(argv[1]);

    srand(myrank() * 17);

    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;
    printf("PI estimated to \%f.", pi);
}
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls “hit” separately
Helper Code for Pi in UPC++ (same as UPC)

• Required includes:

```c
#include <stdio.h>
#include <math.h>
#include <upcxx.h>  // #include <upc.h> for UPC
```

• Function to throw dart and calculate where it hits:

```c
int hit() {
    int const rand_max = 0xFFFFFFFF;
    double x = ((double) rand()) / RAND_MAX;
    double y = ((double) rand()) / RAND_MAX;
    if ((x*x + y*y) <= 1.0) {
        return(1);
    } else {
        return(0);
    }
}
```
Shared vs. Private Variables
Private vs. Shared Variables in UPC++

- Normal C++ variables and objects are allocated in the private memory space for each rank.

- Shared variables are allocated only once, with thread 0
  
  ```
  shared_var<int> ours; // use sparingly: performance
  int mine;
  ```

- Shared variables may not have dynamic lifetime: may not occur in a function definition, except as static. Why?
Shared Variables

Declaration:

\[
\text{shared\_var}<T> \text{ ours};
\]

Explicit read and write with member functions get and put

\[
T \text{ ours.get();}
\]
\[
\text{ours.put(const } T& \text{ val);}
\]

Implicit read and write a shared variable in an expression

- Type conversion operator “\(T()\)” is overloaded to call get

\[
\text{int mine = ours; } // \text{ C++ compiler generates an implicit type conversion from shared\_var}<T> \text{ to } T
\]

- Assignment operator “\(=\)” is overloaded to call put

\[
\text{ours = 5;}
\]

- Compound operators such as “\(+=\)” and “\(-=\)” involve both a read and a write. Note that these are not atomic operations.
Pi in UPC++: Shared Memory Style

- Parallel computing of pi, but with a bug

```c
shared_var<int> hits;  // shared variable to record hits

main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + ranks() - 1)/ranks();
    srand(myrank()*17);
    for (i=0; i < my_trials; i++)
        hits += hit();  // accumulate hits
    barrier();
    if (myrank() == 0) {
        printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

What is the problem with this program?
Pi in UPC: Shared Memory Style

- Like pthreads, but use shared accesses judiciously

```c
shared_var<int> hits;
shared_lock hit_lock;

main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
        my_hits += hit();
    hit_lock.lock();
    hits += my_hits;
    hit_lock.unlock();
    barrier;
    if (myrank == 0)
        printf("PI: %f", 4.0*hits/trials);
}
```
Shared Arrays

Declaration:

    shared_array<Type> sa;

Initialization (should be called collectively):

    sa.init(size_t array_size, sizt_t blk_size);

Finalization (should be called collectively)

    sa.finalize();

Accessing Arrays elements:

    sa[index] = ...;
    ... = sa[index];
    cout << sa[index];
Shared Arrays Are Cyclic By Default

- Shared scalars always live in thread 0
- Shared arrays are spread over the ranks
- Shared array elements are spread across the processes

```cpp
shared_array<int> x, y, z;
x.init(ranks()); /* 1 element per process */
y.init(3*ranks()); /* 3 elements per process */
z.init(3*3); /* 2 or 3 elements per process */
```

- In the pictures below, assume ranks() = 4
  - Blue elts have affinity to rank 0

Think of linearized C array, then map in round-robin
As a 2D array, y is logically blocked by columns
z is not
Pi in UPC: Shared Array Version

• Alternative fix to the race condition
• Have each thread update a separate counter:
  – But do it in a shared array
  – Have one thread compute sum

```cpp
shared_array<int> all_hits;

main(int argc, char **argv) {
    all_hits.init(ranks());
    for (i=0; i < my_trials; i++)
        all_hits[myrank()] += hit();
    barrier();
    if (myrank() == 0) {
        for (i=0; i < ranks(); i++)
            hits += all_hits[i];
        printf("PI estimated to \%f\", 4.0*hits/trials);
    }
}
```

all_hits is shared by all processors, just as hits was

update element with local affinity
Asynchronous Task Execution
**UPC++ Async**

- C++ 11 async function
  ```
  std::future<T> handle = std::async(Function&& f, Args&&... args);
  handle.wait();
  ```

- UPC++ async function
  ```
  // Remote Procedure Call
  upcxx::async(rank)(Function f, T1 arg1, T2 arg2,...);
  upcxx::async_wait();
  ```

  ```
  // Explicit task synchronization
  upcxx::event e;
  upcxx::async(place, &e)(Function f, T1 arg1, ...);
  e.wait();
  ```
Async Task Example

```c
#include <upcxx.h>

void print_num(int num)
{
    printf("myid %u, arg: %d\n", MYTHREAD, num);
}

int main(int argc, char **argv)
{
    for (int i = 0; i < upcxx::ranks(); i++) {
        upcxx::async(i)(print_num, 123);
    }
    upcxx::async_wait(); // wait for all remote tasks to complete
    return 0;
}
```
using namespace upcxx;

// Rank 0 spawns async tasks
for (int i = 0; i < ranks(); i++) {
    // spawn a task expressed by a lambda function
    async(i)([] (int num)
    {
        printf("num: %d\n", num);
    },
    1000+i); // argument to the λ function
}
async_wait(); // wait for all tasks to finish

mpirun -n 4 ./test_async

Output:
num:  1000
num:  1001
num:  1002
num:  1003
using namespace upcxx;

// Thread 0 spawns async tasks
finish {
    for (int i = 0; i < ranks(); i++) {
        async(i)([] (int num)
        {
            printf("num: %d
", num);
        }, 1000+i);
    }
} // All async tasks are completed
Example: Building A Task Graph

using namespace upcxx;
event e1, e2, e3;

async(P1, &e1)(task1);
async(P2, &e1)(task2);
async_after(P3, &e1, &e2)(task3);
async(P4, &e2)(task4);
async_after(P5, &e2, &e3)(task5);
async_after(P6, &e2, &e3)(task6);
async_wait(); // all tasks will be done
Progress Function for Async Tasks

• Each UPC++ rank decides when to execute incoming tasks and send outgoing tasks by polling the task queue:

\[
\text{int } \text{advance(int } \text{max}_\text{in}, \text{ int } \text{max}_\text{out})
\]
- \text{max}_\text{in} maximum number of incoming tasks to be processed before returning
- \text{max}_\text{out} maximum number of outgoing tasks to be processed before returning

• Support different progress models, for example:
  - Call advance() from the default thread
  - Create a dedicated progress thread for polling

• Important Progress Properties
  - Blocking functions in UPC++ call advance() internally to guarantee progress. These only include: async_wait(), barrier(), event.wait(), finish and finalize().
  - Other UPC++ functions are non-blocking
Dynamic Memory Management and Bulk Data Transfer
Dynamic Global Memory Management

- Global address space pointers (pointer-to-shared)
  
  ```
  global_ptr<Type> ptr;
  ```

- Dynamic shared memory allocation
  
  ```
  global_ptr<T> allocate<T>(uint32_t where, size_t count);
  ```

  ```
  void deallocate(global_ptr<T> ptr);
  ```

Example: allocate space for 512 integers on rank 2

```
  global_ptr<int> p = allocate<int>(2, 512);
  ```

Remote memory allocation is not available in MPI-3, UPC or SHMEM.
More on Global Pointers

- Global pointers examples:
  ```
  global_ptr<int> p1;
  global_ptr<void> p2;
  global_ptr<void *> p3;
  ```
- Query the location (owner) of the data
  ```
  Uint32_t where()
  ```
- Get the local pointer (virtual memory address)
  ```
  T* raw_ptr()
  ```
- Pointer arithmetic is the same as that for local pointers

  – There is no phase field in the global pointer
- Can dereference a pointer to read from or write to the global location
  ```
  *ptr or ptr[i]
  ```
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);

// Implicit non-blocking 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++ Translation Example

```cpp
shared_array <int> sa;
sa.init(100, 1);
sa[0] = 1; // "[]" and "=" overloaded

C++ Compiler

tmp_ref = sa.operator [](0);
tmp_ref.operator = (1);

UPC++ Runtime

Yes

Is tmp_ref local?

Yes: Local Access
No: Remote Access

Runtime Address Translation Overheads
```
When Address Translation Overheads Matter?

Case 1: access local data
1. Get the partition id of the global address (1 cycle)
2. Check if the partition is local (1 cycle)
3. Get the local address of the partition (1 cycle)
4. Access data through the local address (1 cycle)

3 CPU cycles for address translation vs. 1 cycle for real work
(Bad: 3X overhead)

Case 2: access remote data
1. Get the partition id of the global address (1 cycle)
2. Check if the partition is local (1 cycle)
3. Get the local address of the partition (1 cycle)
4. Access data through the network (~10^4 cycles)

3 CPU cycles for address translation vs. ~10^4 cycles for real work
(Good: 0.3% overhead)
How to Amortize Address Translation Overheads

• Move data in chunks

```c
void copy(src, dst, count);

non-blocking async_copy is even better
```

• Cast pointer-to-shared to pointer-to-local

```c
int *p1 = (int *)sp1;
int *p2 = (int *)sp2;
```

![Diagram showing process 1's perspective and two processes P1 and P2]
Completion Events for Non-blocking Put

User owns buffers

System owns buffers

Source buffer

Dest. buffer

NB op starts

Local completion

Remote completion
async_copy_and_signal(global_ptr<T> src,
global_ptr<T> dst,
size_t count,
event *signal_event,
event *local_completion,
event *remote_completion);

- Three key events for a non-blocking put
  - Initiator side events:
    - local completion: the src buffer is reusable
    - remote completion: the data has arrived in the dst buffer
  - Target side event:
    - signal event: the data has arrived in the dst buffer
How-to
# UPC++ Cheat Sheet for UPC Programmers

<table>
<thead>
<tr>
<th></th>
<th>UPC</th>
<th>UPC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of threads</td>
<td>THREADS</td>
<td>ranks()</td>
</tr>
<tr>
<td>My ID</td>
<td>MYTHREAD</td>
<td>myrank()</td>
</tr>
<tr>
<td>Shared variable</td>
<td>shared Type s</td>
<td>shared_var&lt;Type&gt; s</td>
</tr>
<tr>
<td>Shared array</td>
<td>shared [bf] Type A[sz]</td>
<td>shared_array&lt;Type&gt; A A.init(sz, bf)</td>
</tr>
<tr>
<td>Pointer-to-shared</td>
<td>shared Type *ptr</td>
<td>global_ptr&lt;Type&gt; ptr</td>
</tr>
<tr>
<td>Dynamic memory allocation</td>
<td>shared void *</td>
<td>global_ptr&lt;Type&gt; allocate&lt;Type&gt;(place, count)</td>
</tr>
<tr>
<td>Bulk data transfer</td>
<td>upc_memcpy(dst, src, sz)</td>
<td>copy&lt;Type&gt;(src, dst, count)</td>
</tr>
<tr>
<td>Affinity query</td>
<td>upc_threadof(ptr)</td>
<td>ptr.where()</td>
</tr>
<tr>
<td>Synchronization</td>
<td>upc_lock_t</td>
<td>shared_lock</td>
</tr>
<tr>
<td></td>
<td>upc_barrier</td>
<td>barrier()</td>
</tr>
</tbody>
</table>

Homework: how to translate *upc forall*?
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
- Many new features in C++11
  - E.g., type inference, variadic templates, lambda functions, r-value references
  - C++ 11 is well-supported by major compilers
Installing UPC++

- Get source from Bitbucket
  
git clone https://bitbucket.org/upcxx/upcxx.git

- Get the optional multidimensional arrays package
  
cd upcxx/include
  
git clone https://bitbucket.org/upcxx/upcxx-upcxx-arrays.git

- Standard autotools build process
  
  ./Bootstrap
  
  ## Create a separate build directory and cd to it
  
  configure --with-gasnet=/path/to/${conduit}--{seq|par}.mak
  
  --prefix=/path/to/install CXX=upc++_backend_compiler
  
  make; make install

- UPC++ is preinstalled on NERSC Edison (Cray XC30)
  
  export MODULEPATH=$MODULEPATH:/usr/common/usg/degas/modulefiles
  
  module load upc++
  
  Or
  
  . /usr/common/usg/degas/upcxx/default-intel/bin/upcxx_vars.sh

For details about installation instructions, please see https://bitbucket.org/upcxx/upcxx/wiki/Installing%20UPC++
The `upc++` compiler wrapper works like the MPI equivalent `mpic++`. For example,

```
## compile hello.cpp to hello.o
upc++ -c hello.cpp

## compile hello.cpp and link it to a.out
upc++ hello.cpp

## print the command line that upc++ would execute
upc++ -show

## print the help message
upc++ -h
```

You can also get UPC++ makefile definitions and shell environment variables to customize for your app.

[https://bitbucket.org/upcxx/upcxx/wiki/Compiling%20UPC++%20Applications](https://bitbucket.org/upcxx/upcxx/wiki/Compiling%20UPC++%20Applications)
Running UPC++ Programs

• Run it like a MPI (multi-process) program, for example,
  – On systems with MPI installed, `mpirun`
  – On a Cray, `aprun`

• Use the conduit-specific gasnet spawner

• Commonly used GASNet env variables
  ```
  ## Increase the size of the global partition per rank
  export GASNET_MAX_SEGSIZE=256MB

  ## Disable process-shared memory nodes
  export GASNET_MAX_SUPERNODE=1
  ```
Application Examples
// shared uint64_t Table[TableSize]; in UPC
shared_array<
uint64_t
>
Table(TableSize);

void RandomAccessUpdate()
{
    uint64_t ran, i;
    ran = starts(NUPDATE / ranks() * myrank());
    for(i = myrank(); i < NUPDATE; i += ranks()) {
        ran = (ran << 1) ^ ((int64_t)ran < 0 ? POLY : 0);
        Table[ran & (TableSize-1)] ^= ran;
    }
}
Performance difference is negligible at large scale
A Software Framework for Block-Structured AMR Applications

Used in many active research projects:
- MAESTRO – low Mach number astrophysics
- CASTRO – compressible radiation/hydrodynamics
- Nyx – cosmology (baryon plus dark matter evolution)
- LMC – low Mach number combustion
- CNSReact – compressible reacting flow
- ACTuARy – atmospheric chemical transport
- PMAMR – subsurface modeling (AMANZI-S)

Source: “BoxLib: A Software Framework for Block-Structured AMR Applications” by Ann Almgren
http://www.speedup.ch/workshops/w42_2013/ann.pdf
Comm. Patterns in BoxLib

Each process does the following:

// Pack data and figure out communication neighbors

MPI_Irecv(...);
MPI_Irecv(...);

...

MPI_Isend(...);
MPI_Isend(...);

...

// Local computation for overlap

MPI_Waitall(...);

// Unpack data and continue

Cells in each box are stored in column-major order. Boxes are laid out in Z-order in 3D space. Each processor gets a contiguous chunk of boxes of equal size.
Message passing protocols

**Eager protocol for short msgs.**

- Send header and payload together
- Message buffered
- Message matched by Recv

**Rendezvous protocol for long msgs.**

- Send message header
- Tell sender the recv addr
- Send message payload
- Message matched by Recv

Message matching is done at the receiver. The sender needs to know the receive buffer address to do RDMA and avoid buffering.

Inattentive CPUs may cause extra lags.
Active Receive (Sender Side Message Matching)

- Message matching is done at the sender
- The sender uses signaling put to transfer the message payload and notify the receiver for completion
- The completion event is like a semaphore and can be used to count multiple operations
## BoxLib Communication Performance

SMC benchmark on Edison, 128 processes, 1 process per numa node, 12 openmp threads per process

<table>
<thead>
<tr>
<th></th>
<th>MPI w. OpenMP</th>
<th>UPC++ w. OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Time : 8.2</td>
<td>Total Time : 7.9</td>
</tr>
<tr>
<td>No overlap</td>
<td>Communication time: 1.8</td>
<td>Communication time: 1.6</td>
</tr>
<tr>
<td></td>
<td>Chemistry time: 2.6</td>
<td>Chemistry time: 2.6</td>
</tr>
<tr>
<td></td>
<td>Hyp-Diff time: 3.8</td>
<td>Hyp-Diff time: 3.8</td>
</tr>
<tr>
<td>Overlap</td>
<td>Total Time : 8.4</td>
<td>Total Time : 7.8</td>
</tr>
<tr>
<td></td>
<td>Communication time: 1.8</td>
<td>Communication time: 1.3</td>
</tr>
<tr>
<td></td>
<td>Chemistry time: 2.9</td>
<td>Chemistry time: 2.8</td>
</tr>
<tr>
<td></td>
<td>Hyp-Diff time: 3.8</td>
<td>Hyp-Diff time: 3.8</td>
</tr>
</tbody>
</table>
Progress thread in UPC++

- Mitigate CPU inattentiveness for better communication and computation overlaps
  
  ```
  progress_thread_start()
  progress_thread_stop()
  ```

- Three threading modes
  
  - Non thread-safe: main thread explicitly transfers progress control to the progress thread and stop it before making UPC++ calls
  
  - Thread-safe with GASNet PAR mode: will need non-thread-specific handle support from GASNet-EX to match the UPC++ usage model
  
  - Thread-safe with pthread mutex and GASNet SEQ mode: use a coarse-grained lock for gasnet calls
Application: Full-Waveform Seismic Imaging

- Method for developing models of earth structure, applicable to ...
  - basic science: study of interior structure and composition
  - petroleum exploration and environmental monitoring
  - nuclear test-ban treaty verification
- Model is trained to predict (via numerical simulation) seismograms recorded from real earthquakes or controlled sources
- Training defines a non-linear regression problem, solved iteratively

Minimize:

\[ \chi(m) = \frac{1}{2} \| d - g(m) \|^2 \]

Collaboration with Scott French et al, Berkeley Seismological Lab

Above: global full-waveform seismic model SEMum2 (French et al., 2013, Science)
Application: Full-Waveform Seismic Imaging

**Convergent Matrix Library**

```
ConvergentMatrix<float,...> GtG( M, M );
...
// for each locally computed update
GtG.update( GtG_i, slice_idx_i );
```

**Internal binning, upcxx::copy and upcxx::async invocation**

```
GtG.commit(); // barrier
// fetch local pointer
float *mat = GtG.get_local_data();
// ScaLAPACK
// MPI-IO collective write
```
Alternative implementation: MPI-3 RMA

- Have to “design for” the MPI implementation
  - NERSC Edison (XC30), so using Cray MPICH 7.0.3 (MPICH 3.0.x)
  - Per-accumulate lock / unlock with exclusive locks
    - Faster than shared (with or without single epoch)
- Would another implementation be faster? (possibly, but hard to say ...)
- In any case, similar code complexity to UPC++

Weak scaling vs. UPC++

- Distributed matrix size fixed (180 GB)
- Dataset size scaled w/ concurrency
  - 64 updates per MPI or UPC++ task + threads in NUMA domain

Setup

- NERSC Edison (Cray XC30)
- GNU Compilers 4.8.2 (-O3)
- Cray MPICH 7.0.3
- Up to 12,288 cores
- Matrix size: 180GB

https://github.com/swfrench/convergent-matrix-mpi
Scientific results: A whole-mantle model

Whole-mantle radially anisotropic shear velocity structure from spectral-element waveform tomography

- First-ever whole-mantle seismic model from numerical waveform tomography
- Reveals new details of deep structure not seen before
- Made feasible by Gauss-Newton scheme, enabled by UPC++

Right: Broad plumes in the earth's lower mantle, including those beneath Pitcairn, Samoa, and other hotspots.

Left: 3D rendering of low-velocity structure beneath the Hawaii hotspot.

(French and Romanowicz, 2015, in revision)
## Alternative implementation: MPI-3 RMA

### Why the performance disparity?
- Very different approaches to achieving “generality”
- Determines what optimizations are available to programmer

<table>
<thead>
<tr>
<th>upcxx::async (general functions)</th>
<th>vs.</th>
<th>MPI_Accumulate (general data types)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Explicit buffer management</td>
<td></td>
<td>• Opaque internal MPI buffers</td>
</tr>
<tr>
<td>• Customized update function</td>
<td></td>
<td>• Generalized MPI data types + pre-defined merge ops</td>
</tr>
<tr>
<td>with domain knowledge</td>
<td></td>
<td>• Progress is impl.-specific and not controllable at target</td>
</tr>
<tr>
<td>• Progress at both source and target is controllable</td>
<td></td>
<td>• Data may take more than one trip to ensure passive target (ex: bulk accumulate in foMPI)</td>
</tr>
<tr>
<td>• One way bulk data movement can be guaranteed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More opportunities to exploit problem / domain specific knowledge
Multidimensional Arrays in UPC++
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
UPC++ Arrays Based on Titanium

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

<table>
<thead>
<tr>
<th>AMR Chombo</th>
<th>C++/Fortran/MPI</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR data structures</td>
<td>35000</td>
<td>2000</td>
</tr>
<tr>
<td>AMR operations</td>
<td>6500</td>
<td>1200</td>
</tr>
<tr>
<td>Elliptic PDE Solver</td>
<td>4200*</td>
<td>1500</td>
</tr>
</tbody>
</table>

* Somewhat more functionality in PDE part of C++/Fortran code
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
  ```cpp
  point<2> lb = {{1, 1}}, ub = {{20, 10}};
  ```

- A rectangular domain is an index space, specified with a lower bound, upper bound, and optional stride
  ```cpp
  rectdomain<2> r(lb, ub);
  ```

- An array is defined over a rectangular domain and indexed with a point
  ```cpp
  ndarray<double, 2> A(r); A[lb] = 3.14;
  ```

- One-sided copy operation copies all elements in the intersection of source and destination domains
  ```cpp
  ndarray<double, 2, global> B = ...;
  B.async_copy(A); // copy from A to B
  async_wait(); // wait for copy completion
  ```
Multidimensional Arrays in UPC++ (and Titanium)

- Titanium arrays have a rich set of operations
- None of these modify the original array, they just create another view of the data in that array
- You create arrays with a RectDomain and get it back later using A.domain() for array A
  - A Domain is a set of points in space
  - A RectDomain is a rectangular one
- Operations on Domains include +, -, * (union, different intersection)
Example: 3D 7-Point Stencil

- Code for each timestep:

```c
// 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) {
        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
- Implemented using lambda, so ";" needed
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

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

- Can allocate arrays in a global index space
- Let library compute intersections
- Avoid null copies
- Copy from interior of other grid

"ghost" cells
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

\[
\text{template<int }N\text{> struct point } \{
    \text{cint_t } x[N];
    \ldots
\};
\]

  – Can be constructed using initializer list

  \[
  \text{point<2> lb = } \{\{1, 1\}\};
  \]

• The \(\text{PT}\) function creates a point in non-initializer contexts

  \[
  \text{point<2> lb = PT(1, 1);}\]

  – Implemented using variadic templates in C++11, explicit overloads otherwise
Array Template

- Arrays represented using a class template, with element type and dimensionality arguments
  
  \[
  \text{template} < \text{class } T, \text{ int } N, \\
  \quad \text{class } F1, \text{ class } F2 > \\
  \text{class ndarray};
  \]

- Last two (optional) arguments specify locality and layout
  - Locality can be \textit{local} (i.e. elements are located in the local memory space) or \textit{global} (elements may be located elsewhere)
  - Layout can be \textit{strided}, \textit{unstrided}, \textit{row}, \textit{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 and templates allow definition of `foreach` loops

- C++11 implementation using type inference and lambda:

  ```cpp
  #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) 
  ptr_ = [&](const decltype(ptr_.dummy_pt()) &p)
  ```

- Pre-C++11 implementation also possible using `sizeof` operator
  - However, loop is flattened, so performance is much slower
C++11 Foreach Translation

• Lambda encapsulates body, passed to loop template

```cpp
template<int N> struct rditer {
    template<class F> rditer &operator=(const F &func) {
        rdloop<N>::loop(func, p0.x, p1.x, loop_stride.x);
        return *this;
    }
};
```

• Loop template implemented recursively, with base case a
  template specialization that calls body (not shown)
  – Result is N-d loop; GCC and Clang optimize it well

```cpp
template<int N> struct rdloop {
    template<class F, class... Is>
    static void loop(const F &func, const int *lwb, const int *upb,
                     const int* stride, Is... is) {
        for (int x = *lwb, u = *upb, s = *stride; x < u; x += s)
            rdloop<N-1>::loop(func, lwb+1, upb+1, stride+1, is..., x);
    }
};
```
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

\[
\begin{align*}
(p_x[0] - base[0]) \times side_factors[0]/stride[0] + \\
(p_x[1] - base[1]) \times side_factors[1]/stride[1] + \\
\end{align*}
\]

• Introduce template specializations to restrict layout
  – \texttt{strided}: any logical or physical stride
  – \texttt{unstrided}: logical and physical strides match
  – \texttt{row}: matching strides + row-major format
  – \texttt{column}: matching strides + column-major format

\texttt{upc++}

• Default in UPC++ to provide best performance
Array Library 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 (GCC)

### NAS Benchmarks

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Titanium CG</th>
<th>Titanium FT</th>
<th>Titanium MG</th>
<th>UPC++ CG</th>
<th>UPC++ FT</th>
<th>UPC++ MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>128</td>
<td>64</td>
<td>16</td>
<td>128</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>32</td>
<td>8</td>
<td>64</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>16</td>
<td>4</td>
<td>32</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>8</td>
<td>2</td>
<td>16</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Better

upC++
Stencil Weak Scaling (GCC)

Stencil 256³ Grid/Core

Performance (GB/s)

Number of Cores

Titanium
UPC++

Better
Array Library Summary

• 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

• Some issues remain
  – Improve performance of one-sided array copies
    • Performance somewhat slower than manual packing/unpacking, as will be shown in miniGMG results
  – GCC and Clang optimize complex template code well, but other compilers do not
    • We are not the only ones to run into this (e.g. Raja, HPX)
      Need to lean on compiler implementers to do a better job
Case Study: miniGMG

- 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.
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)
Array Creation in miniGMG

```c
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
Padding
Column-Major Layout
**Communication Setup for miniGMG Arrays**

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

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));
}

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

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

- **Circular Domain Shift at Boundaries**
- **Compute Intersection**
- **Save Views of Source and Destination Restricted to Intersection**
Bulk Communication Strategy

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

![Diagram of communication strategy](image)
**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
  ```cpp
  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

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

- 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 \texttt{async\_copy} instead of \texttt{memcpy}:

\begin{verbatim}
... 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);
    }
}
\end{verbatim}

• Takes advantage of fact that source and destination layouts match
Array Copy Code

• Array version delegates actual copies to array library:

```python
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

1 Process/Socket, $128^3$/Process

Number of Messages Sent

Message Sizes (Bytes)
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

![Graphs showing running times for different process counts and OpenMP variants.](image-url)

<table>
<thead>
<tr>
<th>No. of Processes</th>
<th>Fine-Grained</th>
<th>Array</th>
<th>Bulk</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td>Fine-Grained</td>
<td>Array</td>
<td>Bulk</td>
<td>MPI</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Better: Increased performance with higher injection concurrency.
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
miniGMG Summary

- 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
Lessons Learned

- Many productive language features can be implemented in C++ without modifying the compiler
  - Macros and template metaprogramming provide a lot of power for extending the core language
- Many Titanium applications can be ported to UPC++ with little effort
  - UPC++ can provide the same productivity gains as Titanium
- However, analysis and optimization still an open question
  - Can we build a lightweight standalone analyzer/optimizer for UPC++?
  - Can we provide automatic specialization at runtime in C++?
Takeaways

• Communicate more often
  – use non-blocking one-sided operations
• Move computation instead of data
  – use async and event-driven execution
• Express algorithms with high-level data structures
  – use Titanium-style multidimensional arrays
• Easy on-ramp
  – interoperate w. existing MPI+OpenMP codes
• We look forward to collaboration!
  – share knowledge and experience beyond tools

UPC++: https://bitbucket.org/upcxx