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

- 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”

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
Hello World in UPC

- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Using this fact, plus the a few UPC keywords:

```c
#include <upc.h> /* needed for UPC extensions */
#include <stdio.h>

main() {
    printf("Thread %d of %d: hello UPC world\n", MYTHREAD, THREADS);
}
```

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 = $\frac{1}{4} \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - $\#$ points inside / $\#$ points total
  - $\pi = 4 \times \text{ratio}$

Pi in UPC

- 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(MYTHREAD*17);

    for (i=0; i < trials; i++) hits += hit();
    pi = 4.0*hits/trials;

    printf("PI estimated to %f.", pi);
}
```

Helper Code for Pi in UPC

- Required includes:
  ```c
  #include <stdio.h>
  #include <math.h>
  #include <upc.h>
  ```
- 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

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0
  ```
  shared int ours; // use sparingly: performance
  int mine;
  ```
- Shared variables may not have dynamic lifetime, i.e., may not occur in a function definition, except as static.

### Pi in UPC: Shared Memory Style

- Parallel computing of pi, but with a bug
  ```
  shared int hits; // shared variable to record hits
  main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]); // divide work up evenly
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
      hits += hit(); // accumulate hits
    upc_barrier;
    if (MYTHREAD == 0) {
      printf("PI estimated to \%f\.", 4.0*hits/trials);
    }
  }
  ```
- What is the problem with this program?

### UPC Synchronization

- UPC has two basic forms of barriers:
  - Barrier: block until all other threads arrive
    ```
    upc_barrier
    ```
  - Split-phase barriers
    ```
    upc_notify; // this thread is ready for barrier
    upc_wait; // wait for others to be ready
    ```
- UPC also has locks for protecting shared data:
  - Locks are an opaque type (details hidden):
    ```
    upc_lock_t *upc_global_lock_alloc(void);
    ```
  - Critical region protected by lock/unlock:
    ```
    void upc_lock(upc_lock_t *l);
    void upc_unlock(upc_lock_t *l);
    ```
    use at start and end of critical region
Pi in UPC: Shared Memory Style

- Like pthreads, but use shared accesses judiciously
- `shared int hits;` (one shared scalar variable)
- `int i, my_hits, my_trials = 0;` (other private variables)
- `upc_lock_t *hit_lock = upc_all_lock_alloc();` (create a lock)
- `my_trials = (trials + THREADS - 1)/THREADS;` (accumulate hits locally)
- `srand(MYTHREAD*17);` (accumulate across threads)
- `for (i=0; i < my_trials; i++)` (compute sum)
- `my_hits += hit();` (update element with local affinity)
- `upc_lock(hit_lock);` (barrier implied by collective)
- `upc_unlock(hit_lock);` (no shared variables)
- `upc_barrier;` (Berkeley collectives)
- `if (MYTHREAD == 0)` (barrier implied by collective)
- `printf("PI: %f", 4.0*my_hits/trials);` (all hits is shared by all processors, just as hits was)

Pi in UPC: Data Parallel Style w/ Collectives

- The previous version of Pi works, but is not scalable:
  - On a large # of threads, the locked region will be a bottleneck
- Use a reduction for better scalability

```c
#include <bupc_collectivev.h>
main(int argc, char **argv) {
    ... declarations an initialization code omitted...
    for (i=0; i < my_trials; i++)
        my_hits += hit();
    my_hits =  // type, input, thread, op
        bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc_barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*my_hits/trials);
}
```

Shared Arrays Are Cyclic By Default

- Shared scalars always live in thread 0
- Shared arrays are spread over the threads
- Shared array elements are spread across the threads
- `shared int x[THREADS] /* 1 element per thread */`
- `shared int y[3][THREADS] /* 3 elements per thread */`
- `shared int z[3][3] /* 2 or 3 elements per thread */`
- In the pictures below, assume THREADS = 4
- Blue elts have affinity to thread 0
- As a 2D array, y is logically blocked by columns
- z is not

Think of linearized C array, then map in round-robin

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

```c
shared int all_hits [THREADS];
main(int argc, char **argv) {
    ... declarations an initialization code omitted...
    for (i=0; i < my_trials; i++)
        all_hits[MYTHREAD] += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
        for (i=0; i < THREADS; i++)
            hits += all_hits[i];
        printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```
Global Memory Allocation

shared void *upc_alloc(size_t nbytes);

- nbytes: size of memory in bytes
- Non-collective: called by one thread
- The calling thread allocates a contiguous memory space in the shared space with affinity to itself.

shared [] double [n] p2 = upc_alloc(n*sizeof(double));

void upc_free(shared void *ptr);

- Non-collective function; frees the dynamically allocated shared memory pointed to by ptr

Distributed Arrays Directory Style

- Many UPC programs avoid the UPC style arrays in factor of directories of objects

typedef shared [] double *sdblptr;

shared sdblptr directory[THREADS];
directory[i]=upc_alloc(local_size*sizeof(double));

Distributed Arrays Directory Style

- These are also more general:
  - Multidimensional, unevenly distributed
  - Ghost regions around blocks

UPC Non-blocking Bulk Operations

Important for performance:
- Communication overlap with computation
- Communication overlap with communication (pipelining)
- Low overhead communication

#include<upc_nb.h>

upc_handle_t h = upc_memcpy_nb(shared void * restrict dst,
                                 shared const void * restrict src,
                                 size_t n);

void upc_sync(upc_handle_t h); // blocking wait
int upc_sync_attempt(upc_handle_t h); // non-blocking
One-Sided Communication in GASNet

- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided message needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
  - Ordering requirements on messages can also hinder bandwidth

Why Should You Care about PGAS?

- Latency between 2 Xeon Phi's via Infiniband
- Latency between 2 Intel IvyBridge nodes on NERSC Edison (Cray XC30)

Bandwidths on Cray XE6 (Hopper)

Cray XE6 Application Performance
Application Challenge: Fast All-to-All

Transpose in 3D FFT

- **chunk = all rows with same destination**
- **pencil = 1 row**
- **slab = all rows in a single plane with same destination**

**Three approaches:**

- **Chunk:**
  - Wait for 2nd dim FFTs to finish
  - Minimize # messages
- **Slab:**
  - Wait for chunk of rows destined for 1 proc to finish
  - Overlap with computation
- **Pencil:**
  - Send each row as it completes
  - Maximize overlap and
  - Match natural layout

FFT Performance on BlueGene/P

- **UPC implementation outperforms MPI**
- **Both use highly optimized FFT library on each node**
- **UPC version avoids send/receive synchronization**
  - Lower overhead
  - Better overlap
  - Better bisection bandwidth

Bisection Bandwidth

- **Avoid congestion at node interface:** allow all cores to communicate
- **Avoid congestion inside global network:** spread communication over longer time period (start early, send often)
- **Synchronize only when needed:** sometimes fine-grained, sometimes one global barrier (after all incoming counts are reached) is best

UPC Atomic Operations

- More efficient than using locks when applicable
  
  ```c
  upc_lock();
  update();
  upc_unlock();
  ```
  
  vs
  
  ```c
  atomic_update();
  ```

- Hardware support for atomic operations are available, but need to be careful about atomicity w.r.t nonatomics

- Example: atomic fetch-and-add
  
  ```c
  int upc_atomic_i_fetchadd_relaxed (shared void *ptr, int op);
  ```

- See more examples on the web page:
  
  [http://upc.lbl.gov/docs/user/#atomics](http://upc.lbl.gov/docs/user/#atomics)
De novo Genome Assembly (Next Homework!)

- DNA sequence consists of 4 bases: A/C/G/T
- Read: short fragment of DNA
- De novo assembly: Construct a genome (chromosomes) from a collection of reads

PGAS in Genome Assembly

- Sequencers produce fragments called “reads”
- Chop them into overlap fixed-length fragments, “K-mers”
- Parallel DFS (from randomly selected K-mers) → “contigs”

Science Impact: HipMer is transformative

- Human genome (3Gbp) “de novo” assembled:
  - SGA assembler: 140 hours
  - Meraculous: 48 hours
  - HipMer: 8 minutes (360x speedup)
- Wheat genome (17 Gbp) “de novo” assembled (2014):
  - Meraculous (did not run):
  - HipMer: 39 minutes; 15K cores (first all-in-one assembly)
- Pine genome (20 Gbp) “de novo” assembled (2014):
  - Masurca: 3 months; 1 TB RAM
- Wetland metagenome (1.25 Tbp) analysis (2015):
  - Meraculous (projected): 15 TB of memory
  - HipMER: 11 minutes; 20K cores (assembly TBD)
HipMer (High Performance Meraculous) Assembly Pipeline

Distributed Hash Tables in PGAS
- Remote Atomics, Dynamic Aggregation
- Software Caching (sometimes)
- Clever algorithms (bloom filters, locality-aware hashing)

De novo assembly is important

All metagenomes

De novo assembly is important

HipMer: An Extreme-Scale De Novo Genome Assembler, SC'15

Elmer Georganas, Aydog Buluc, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Lenny Oliker, Dan Rokhsar, and Kathy Yelick. HipMer: An Extreme-Scale De Novo Genome Assembler, SC’15

DEGAS is a DOE-funded X-Stack with Lawrence Berkeley National Lab, Rice Univ., UC Berkeley, and UT Austin.

DEGAS

UPC++

Led by Yili Zheng (LBNL)
with Amir Kamil (U Mich)

And host of others: Paul Hargrove, Dan Bonachea, John Bachan,

UPC++: PGAS with “Mixins”

- UPC++ uses templates (no compiler needed)

  ```
  shared_var<int> s;
  global_ptr<LLNode> g;
  shared_array<int> sa(8);
  ```

- Default execution model is SPMD, but

  ```
  async(place) (Function f, T1 arg1,..);
  wait(); // other side does poll();
  ```

- Remote methods, async

- Research in teams for hierarchical algorithms and machines

  ```
  teamsplit (team) { ... }
  ```

- Interoperability is key; UPC++ can be use with OpenMP or MPI

JGI
UPC++ Performance Close to UPC

UPC++ is a library, not a compiled language, yet performance is comparable

GUPS (fine-grained) Performance on MIC and BlueGene/Q

<table>
<thead>
<tr>
<th>Num. of Processes</th>
<th>MIC Giga Updates Per Second</th>
<th>BlueGene/Q Giga Updates Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>8</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>16</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>32</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>64</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td>128</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>256</td>
<td>1.28</td>
<td>1.28</td>
</tr>
<tr>
<td>512</td>
<td>2.56</td>
<td>2.56</td>
</tr>
<tr>
<td>1024</td>
<td>5.12</td>
<td>5.12</td>
</tr>
<tr>
<td>2048</td>
<td>10.24</td>
<td>10.24</td>
</tr>
<tr>
<td>4096</td>
<td>20.48</td>
<td>20.48</td>
</tr>
<tr>
<td>8192</td>
<td>40.96</td>
<td>40.96</td>
</tr>
</tbody>
</table>

Difference between UPC++ and UPC is about 0.2 µs (~220 cycles)

Locality Control On-Node is Important

Optimizations:
- Blocked vs. cyclic (default) array layout
- Use private pointer to the thread block in shared array
  
  ```cpp
double* my_x = (double*)(x + MYTHREAD * BSIZE)
  ```

Bulk Communication with One-Sided Data Transfers

```cpp
// 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

Dynamic Global Memory Management

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

- Dynamic shared memory allocation
  
  ```cpp
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
  
  ```cpp
global_ptr<int> p = allocate<int>(2, 512);
```

Remote memory allocation is not available in MPI-3, UPC or SHMEM.
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;
}
```

Async with C++11 Lambda Function

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

mpirun -n 4 ./test_async

Output:
num: 1000
num: 1001
num: 1002
num: 1003

Function arguments and lambda-captured values must be `std::is_trivially_copyable`.

Application Challenge: Data Fusion in UPC++

- Seismic modeling for energy applications “fuses” observational data into simulation
- With UPC++ “matrix assembly” can solve larger problems

First ever sharp, three-dimensional scan of Earth’s interior that conclusively connects plumes of hot rock rising through the mantle with surface hotspots that generate volcanic island chains like Hawaii, Samoa and Iceland.

Distributed Matrix Assembly
- Remote asyncs with user-controlled resource management
- Remote memory allocation
- Team idea to divide threads into injectors / updaters
- 6x faster than MPI 3.0 on 1K nodes
- Improving UPC++ team support

See French et al, IPDPS 2015 for parallelization overview.
Multidimensional Arrays in UPC++ (and Titanium)

- Titanium arrays have a rich set of operations: translate, restrict, slice (n dim to n-1)
- 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, intersection)

Arrays in a Global Address Space for AMR

- Key features of UPC++ arrays:
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to iterate over interior:
  ```
  foreach (idx, gridB.shrink(1).domain())
  ```
- Array copies automatically work on intersection
  ```
  gridB.copy(gridA.shrink(1));
  ```

Mini-GMG in UPC++ uses high level array library for Productivity and Performance

- “Fine-grained” like OpenMP
- “Bulk” like MPI with 1-sided communication;
- “Array” version uses multi-dimensional array constructs for productivity and ~MPI performance
- Future runtime optimizations should close Array/Bulk gap

NWChem on GASNet

- Production chemistry code
  - 60K downloads worldwide
  - 200-250 scientific application publications per year
  - Over 6M LoC, 25K files
- New version on GASNet for
  - Improved performance
  - Portability with other PGAS
**Application Challenge: Dynamic Load Balancing**

- **Static:** Equal size tasks
- **Semi-Static:** Tasks have different but estimable times
- **Dynamic:** Times are not known until mid-execution

Dynamic (on-the-fly) useful when:

- Load imbalance penalty > communication to balance
- Load balancing can’t solve lack of parallelism

- **Regular meshes, dense matrices, direct n-body**
- **Adaptive and unstructured meshes, sparse matrices, tree-based n-body, particle-mesh methods**
- **Search (UTS), irregular boundaries, subgrid physics, unpredictable machines**

**Impact**

- Hartree Fock example (e.g., in NWChem)
  - Inherent load imbalance
- UPC++
  - Dynamic work stealing and fast atomic operations enhanced load balance
  - New distributed array abstraction enabled productivity and performance
- **Strong Scaling of UPC++ HF Compared to GTFock with Global Arrays**

**Beyond Put/Get: Event-Driven Execution**

- DAG Scheduling in a distributed (partitioned) memory context
- Assignment of work is static; schedule is dynamic
- **Strong Scaling of UPC++ HF Compared to GTFock with Global Arrays**
"memory constrained" lookahead

Some edges omitted
**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
```

**Hierarchical machines and Applications**

- Hierarchical memory model may be necessary (what to expose vs hide)
- Two approaches to supporting the hierarchical control
  - **Option 1: Dynamic parallelism creation**
    - Recursively divide until you run out of work (or hardware)
    - Runtime needs to match parallelism to hardware hierarchy
  - **Option 2: Hierarchical SPMD with “Mix-ins” (e.g., UPC++)**
    - Hardware threads can be grouped into units hierarchically
    - Add dynamic parallelism with voluntary tasking on a group
    - Add data parallelism with collectives on a group

**One-sided communication works everywhere**

**PGAS programming model**

```
*p1 = *p2 + 1;
A[i] = B[i];
upc_memput(A, B, 64);
```

It is implemented using one-sided communication: put/get

**Support for one-sided communication (DMA) appears in:**

- Fast one-sided network communication (RDMA, Remote DMA)
- Move data to/from accelerators
- Move data to/from I/O system (Flash, disks,..)
- Movement of data in/out of local-store (scratchpad) memory

**CS267 Project Ideas**

- You will experiment with UPC in a future homework
- UPC++ or UPC could be basis of final projects:
  - UPC++ building blocks for genome analysis + higher level language (Perl/Julia/Python/Jupyter?)
  - Other genome analysis problems
  - Protein assembly
  - Machine learning algorithms (store "model parameters" in a shared vector a la Hogwild!)
  - Data fusion problems in simulation
  - Other NWChem component in UPC++ (dynamic load balancing + tensor ops)
  - Hierarchical PGAS: team implementations
  - PGAS-ify Julia, Python (done once)...
  - UPC++ on EC2 (and/or comparison)
Summary

- UPC is a mature language with multiple implementations
  - Cray compiler
- Language specification and other documents
  [https://code.google.com/p/upc-specification](https://code.google.com/p/upc-specification)
  [https://upc-lang.org](https://upc-lang.org)
- UPC++
  - Newer “language” under development
  - Adds dynamic parallelism on top of SPMD default
  - Powerful Multi-D arrays
  - Hierarchical parallelism mapped to machine

Science Across the “Irregularity” Spectrum

- Massive Independent Jobs for Analysis and Simulations
- Nearest Neighbor Simulations
- All-to-All Simulations
- Random access, large data Analysis

Data analysis and simulation

Thanks!

External collaborators (& their teams!)
- Vivek Sarkar, Rice
- John Mellor-Crummey, Rice
- Mattan Erez, UT Austin

**UPC Synchronization**
UPC Global Synchronization

- UPC has two basic forms of barriers:
  - Barrier: block until all other threads arrive
    ```c
    upc_barrier
    ```
  - Split-phase barriers
    ```c
    upc_notify: this thread is ready for barrier
    do computation unrelated to barrier
    upc_wait: wait for others to be ready
    ```
- Optional labels allow for debugging
  ```c
  #define MERGE_BARRIER 12
  if (MYTHREAD%2 == 0) {
      ...
      upc_barrier MERGE_BARRIER;
  } else {
      ...
      upc_barrier MERGE_BARRIER;
  }
  ```

Synchronization - Locks

- Locks in UPC are represented by an opaque type:
  ```c
  upc_lock_t
  ```
- Locks must be allocated before use:
  ```c
  upc_lock_t *upc_all_lock_alloc(void);
  ```
  allocates 1 lock, pointer to all threads
  ```c
  upc_lock_t *upc_global_lock_alloc(void);
  ```
  allocates 1 lock, pointer to one thread
- To use a lock:
  ```c
  void upc_lock(upc_lock_t *l)
  void upc_unlock(upc_lock_t *l)
  ```
  use at start and end of critical region
- Locks can be freed when not in use
  ```c
  void upc_lock_free(upc_lock_t *ptr);
  ```

Pi in UPC: Shared Memory Style

- Like pthreads, but use shared accesses judiciously
  ```c
  shared int hits;  // one shared scalar variable
  main(int argc, char **argv) {
      int i, my_hits, my_trials = 0;  // other private variables
      upc_lock_t *hit_lock = upc_all_lock_alloc();
      int trials = atoi(argv[1]);
      my_trials = (trials + THREADS - 1)/THREADS;
      srand(MYTHREAD*17);
      for (i=0; i < my_trials; i++) {  // accumulate hits locally
          my_hits += hit();
          upc_lock(hit_lock);
          hits += my_hits;
      }
      upc_unlock(hit_lock);
      upc_barrier;
      if (MYTHREAD == 0)
          printf("PI: %f", 4.0*hits/trials);
  }
  ```

Recap: Private vs. Shared Variables in UPC

- We saw several kinds of variables in the pi example
  - Private scalars (`my_hits`)
  - Shared scalars (`hits`)
  - Shared arrays (`all_hits`)
  - Shared locks (`hit_lock`)

Global address space

<table>
<thead>
<tr>
<th>Thread_0</th>
<th>Thread_1</th>
<th>...</th>
<th>Thread_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>hits:</td>
<td>hit_lock:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all_hits[0]:</td>
<td>all_hits[1]:</td>
<td>all_hits[n]:</td>
<td></td>
</tr>
<tr>
<td>my_hits:</td>
<td>my_hits:</td>
<td>...</td>
<td>my_hits:</td>
</tr>
</tbody>
</table>

where:

n=Threads-1

Shared

Private

CS267 Lecture 2
UPC Collectives

UPC Collectives in General

• The UPC collectives interface is in the language spec:
  - http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf
• It contains typical functions:
  - Data movement: broadcast, scatter, gather, ...
  - Computational: reduce, prefix, ...
• Interface has synchronization modes:
  - Avoid over-synchronizing (barrier before/after is simplest semantics, but may be unnecessary)
  - Data being collected may be read/written by any thread simultaneously
• Simple interface for collecting scalar values (int, double, ...)
  - Berkeley UPC value-based collectives
  - Works with any compiler
  - http://upc.lbl.gov/docs/user/README-collectivev.txt

Pi in UPC: Data Parallel Style

• The previous version of Pi works, but is not scalable:
  - On a large # of threads, the locked region will be a bottleneck
• Use a reduction for better scalability

```c
#include <bupc_collectivev.h>

// shared int hits;

main(int argc, char **argv) {
    ...
    for (i=0; i < my_trials; i++)
        my_hits += hit();

    my_hits = bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc barrier;
    if (MYTHREAD == 0)
        printf("PI: %f", 4.0*my_hits/trials);
}
```

UPC (Value-Based) Collectives

• A portable library of collectives on scalar values (not arrays)

Example: `x = bupc_allv_reduce(double, x, 0, UPC_ADD)`

```c
TYPE bupc_allv_reduce(TYPE, TYPE value, int root, upc_op_t op)
```

- `TYPE` is the type of value being collected
- `root` is the thread ID for the root (e.g., the source of a broadcast)
- `value` is both the input and output (must be a “variable” or l-value)
- `op` is the operation: UPC_ADD, UPC_MULT, UPC_MIN, ...

• Computational Collectives: reductions and scan (parallel prefix)
• Data movement collectives: broadcast, scatter, gather

• Portable implementation available from:
  - http://upc.lbl.gov/download/dist/upcr_preinclude/bupc_collectivev.h
• UPC also has more general collectives over arrays
  - http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf

Berkeley collectives
no shared variables
barrier implied by collective
Full UPC Collectives
- Value-based collectives pass in and return scalar values
- But sometimes you want to collect over arrays
- When can a collective argument begin executing?
  - Arguments with affinity to thread $i$ are ready when thread $i$ calls the function; results with affinity to thread $i$ are ready when thread $i$ returns.
  - This is appealing but it is incorrect: In a broadcast, thread 1 does not know when thread 0 is ready.

UPC Collective: Sync Flags
- In full UPC Collectives, blocks of data may be collected
- A extra argument of each collective function is the sync mode of type upc_flag_t.
- Values of sync mode are formed by or-ing together a constant of the form UPC_IN_X_SYNC and a constant of the form UPC_OUT_YSYNC, where $X$ and $Y$ may be NO, MY, or ALL.
- If sync_mode is (UPC_IN_XSYNC | UPC_OUT_YSYNC), then if $X$ is:
  - NO the collective function may begin to read or write data when the first thread has entered the collective function call,
  - MY the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call, and
  - ALL the collective function may begin to read or write data only after all threads have entered the collective function call
- and if $Y$ is
  - NO the collective function may read and write data until the last thread has returned from the collective function call,
  - MY the collective function may return in a thread only after all reads and writes of data with affinity to the thread are complete, and
  - ALL the collective function call may return only after all reads and writes of data are complete.

Example: Vector Addition
- Questions about parallel vector additions:
  - How to layout data (here it is cyclic)
  - Which processor does what (here it is “owner computes”)

```c
/* vadd.c */
#include <upc_relaxed.h>
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
  int i;
  for(i=0; i<N; i++)
    if (MYTHREAD == i%THREADS)
      sum[i] = v1[i] + v2[i];
}
```
Work Sharing with upc forall()

- A common idiom:
  - Loop over all elements; work on those owned by this thread
- UPC adds a special type of loop
  \[
  \text{upc forall}(\text{init}; \text{test}; \text{loop}; \text{affinity})
  \text{statement};
  \]
- Programmer indicates the iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression indicates which iterations to run on each thread. It may have one of two types:
  - Integer: \(\text{affinity} \% \text{THREADS} \text{ is MYTHREAD}\)
  - Pointer: \(\text{upc_threadof(affinity)} \text{ is MYTHREAD}\)
- Syntactic sugar for:
  \[
  \text{for}(i=0; i<N; i++) \text{ if (MYTHREAD == i} \% \text{THREADS)}
  \]
- Compilers will sometimes do better than this, e.g.,
  \[
  \text{for}(i=\text{MYTHREAD}; i<N; i+=\text{THREADS})
  \]

Vector Addition with upc forall

- Vector addition can be written as follows

```c
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];

void main() {
    int i;
    upc forall(i=0; i<N; i++; i) Cyclic data distribution default
        sum[i]=v1[i]+v2[i];
}
```

- The code would be correct but slow if the affinity expression were \(i+1\) rather than \(i\).
- Equivalent code could use "&sum[i]" for affinity and would still work if you change the layout of \(sum\)

Distributed Arrays in UPC

- Array layouts are controlled by blocking factors:
  - Empty (cyclic layout)
  - [*] (blocked layout)
  - [b] (fixed block size)
  - [0] or [] (indefinite layout, all on 1 thread)
- Vector addition example can be rewritten as follows using a cyclic or (maximally) blocked layout

```c
#define N 100*THREADS
shared int [*] v1[N], v2[N], sum[N];

void main() {
    int i;
    upc forall(i=0; i<N; i++; sum[i]) blocked layout
        sum[i]=v1[i]+v2[i];
}
```
### Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:
  - Empty (cyclic layout)
  - [*] (blocked layout)
  - [0] or [] (indefinite layout, all on 1 thread)
  - [b] or [b1][b2]...[bn] = [b1*b2*...bn] (fixed block size)
- The affinity of an array element is defined in terms of:
  - block size, a compile-time constant
  - and THREADS.
- Element \(i\) has affinity with thread \((i / \text{block\_size}) \% \text{THREADS}\)
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping

### 2D Array Layouts in UPC

- Array \(a1\) has a row layout and array \(a2\) has a block row layout.

```c
shared [m] int a1 [n][m];
shared [k*m] int a2 [n][m];
```

- If \((k + m) \% \text{THREADS} = = 0\) them \(a3\) has a row layout

```c
shared int a3 [n][m+k];
```

- To get more general HPF and ScaLAPACK style 2D blocked layouts, one needs to add dimensions.
- Assume \(r*c = \text{THREADS}\);

```c
shared [b1][b2] int a5 [m][n][r][c][b1][b2];
```

- or equivalently

```c
shared [b1*b2] int a5 [m][n][r][c][b1][b2];
```

### Pointers to Shared vs. Arrays

- In the C tradition, array can be access through pointers
- Here is the vector addition example using pointers

```c
#define N 100*\text{THREADS}
shared int v1[N], v2[N], sum[N];
void main() {
  int i;
  shared int *p1, *p2; // \text{p1} points to \text{v1}, \text{p2} points to \text{v2}
  p1=v1; p2=v2;
  for (i=0; i<N; i++, p1++, p2++)
    if (i \% \text{THREADS}== \text{MYTHREAD})
      sum[i]= *p1 + *p2;
}
```

### UPC Pointers

#### Where does the pointer point?

<table>
<thead>
<tr>
<th>Where does the pointer reside?</th>
<th>Local</th>
<th>Global (to shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>(p1)</td>
<td>(p2)</td>
</tr>
<tr>
<td>Shared</td>
<td>(p3)</td>
<td>(p4)</td>
</tr>
</tbody>
</table>

```c
int *p1; /* private pointer to local memory */
shared int *p2; /* private pointer to shared space */
int *shared p3; /* shared pointer to local memory */
shared int *shared p4; /* shared pointer to shared space */
```

Shared to local memory (p3) is not recommended.
**UPC Pointers**

![Diagram of UPC Pointers]

**Common Uses for UPC Pointer Types**

- `int *p1;`  
  - These pointers are fast (just like C pointers)
  - Use to access local data in part of code performing local work
  - Often cast a pointer-to-shared to one of these to get faster access to shared data that is local

- `shared int *p2;`  
  - Use to refer to remote data
  - Larger and slower due to test-for-local + possible communication

- `int *shared p3;`  
  - Not recommended

- `shared int *shared p4;`  
  - Use to build shared linked structures, e.g., a linked list

**UPC Pointers**

- In UPC pointers to shared objects have three fields:
  - thread number
  - local address of block
  - phase (specifies position in the block)

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Thread</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase</td>
<td>Thread</td>
<td>Virtual Address</td>
</tr>
<tr>
<td>63</td>
<td>49</td>
<td>38</td>
</tr>
<tr>
<td>48</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>0</td>
</tr>
</tbody>
</table>

- Example implementation

**UPC Pointers**

- Pointer arithmetic supports blocked and non-blocked array distributions
- Casting of shared to private pointers is allowed but not vice versa!
- When casting a pointer-to-shared to a pointer-to-local, the thread number of the pointer to shared may be lost
- Casting of shared to local is well defined only if the object pointed to by the pointer to shared has affinity with the thread performing the cast
Special Functions

- `size_t upc_threadof(shared void *ptr);` returns the thread number that has affinity to the pointer to shared.
- `size_t upc_phaseof(shared void *ptr);` returns the index (position within the block) field of the pointer to shared.
- `shared void *upc_resetphase(shared void *ptr);` resets the phase to zero.

Global Memory Allocation

```c
shared void *upc_alloc(size_t nbytes);
```
- `nbytes : size of memory in bytes`
- Non-collective: called by one thread
- The calling thread allocates a contiguous memory space in the shared space with affinity to itself.

```c
shared [] double [n] p2 = upc_alloc(n*sizeof(double));
```

```c
void upc_free(shared void *ptr);
```
- Non-collective function; frees the dynamically allocated shared memory pointed to by `ptr`.

Distributed Arrays Directory Style

- Many UPC programs avoid the UPC style arrays in factor of directories of objects

```c
typedef shared [] double *sdblptr;
shared sdblptr directory[THREADS];
directory[i]=upc_alloc(local_size*sizeof(double));
```

- These are also more general:
  - Multidimensional, unevenly distributed
  - Ghost regions around blocks
Memory Consistency in UPC

• The consistency model defines the order in which one thread may see another thread's accesses to memory.
  - If you write a program with unsynchronized accesses, what happens?
  - Does this work?
    ```
    data = ...   while (!flag) { }; 
    flag = 1;  ...
    ```
    // use the data
  - Use strict on variables that are used as synchronization (`flag`)

• UPC has two types of accesses:
  - **Strict**: will always appear in order
  - **Relaxed**: May appear out of order to other threads
  - Use the include file:
    ```
    #include <upc_relaxed.h>
    ```

• There are several ways of designating the type, commonly:
  - `strict` on variables that are used as synchronization

Properties of UPC memory model

• Definitions:
  - **A data race is**:
    • Two concurrent memory operations from two different threads to the same memory location in which at least one is a write.
  - **A race-free program is one in which**:
    • All executions of the program are free of data races (would be nice if the user could only worry about naive implementations)

• And states that programs will be sequentially consistent (behave as if all operations from each thread execute in order) if either of the following holds:
  - The program is race-free
  - The program contains no relaxed operations

Intuition on Strict Orderings

• Each thread may “build” its own total order to explain behavior
• They all agree on the strict ordering shown above in black, but
  - Different threads may see relaxed writes in different orders
    • Allows non-blocking writes to be used in implementations
  - Each thread sees own dependencies, but not those of other threads
    • Weak, but otherwise there would place consistency requirements on some relaxed operations (e.g., local cache control insufficient)
    • Preserving dependencies requires usual compiler/hw analysis

Synchronization- Fence

• Upc provides a fence construct
  - Equivalent to a null strict reference, and has the syntax
    ```
    upc_fence;
    ```
  - UPC ensures that all shared references issued before the upc_fence are complete
### UPC Performance Features

#### UPC Compiler Implementation

- **UPC-to-C translator**
  - **Pros:** portable, can use any backend C compiler
  - **Cons:** may lose program information between the two compilation phases
  - **Example:** Berkeley UPC

- **UPC-to-object-code compiler**
  - **Pros:** better for implementing UPC specific optimizations
  - **Cons:** less portable
  - **Example:** GCC UPC and most vendor UPC compilers

#### Exemplar Programming System Stack on Cray

- Chapel Apps
  - Chapel Compiler
  - Chapel Runtime
  - GASNet GNI/Conduit
  - Cray Communication Libraries (DMAPP/GNI)
  - Cray Networks (Gemini/Aries)

- UPC Apps
  - BUPC Compiler
  - BUPC Runtime
  - GASNet GNI/Conduit
  - Cray Communication Libraries (DMAPP/GNI)
  - Cray Networks (Gemini/Aries)

- CAF Apps
  - CAF Compiler
  - Cray CAF Compiler
  - Cray PGAS Runtime
  - Cray MPI / MPICH2

- MPI Apps

#### Berkeley UPC Software Stack

- **UPC Applications**
  - **UPC-to-C Translator**
  - **C code with Runtime Calls**
  - **UPC Runtime**
  - **GASNet Communication Library**
  - **Network and OS API**

Tip: you can choose your favorite C compiler (e.g., clang, icc, gcc, nvcc, xlc) as the backend compiler with BUPC.
GASNet Software Stack

- PGAS Programming Systems (e.g., BUPC, CAF 2.0, Chapel, OpenSHMEM, Titanium, and DEGAS)
- GASNet
  - Active Messages
  - One-sided Communication
  - Collective Communication
- Low-level communication APIs (e.g., Cray GNI, IBM PAMI, IB Verbs, Portals 4, UDP, shared-memory)
- Interconnect

Implementing UPC Shared Data Access

```
shared int s;
s = 5;
UPC->C Translator
UPCR.PUT.PSHARED.VAL(s, 0, 5, 4);
```

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)

Performance: Pointer-to-local vs. Pointer-to-shared

```
Local pointer
Local pointer
Pointer to shared Berkeley UPC
Pointer to shared GCC/UPC
```

Tip: Cast a pointer-to-shared to a regular C pointer for accessing the local portion of a shared object. E.g., `int *p = (int *)pts; p[0] = 1;`
**How to Amortize Address Translation Overheads**

- Move data in chunks
  
  ```cpp
  upc_mem(cpy|put|get)(...)
  ```
  
  Non-blocking `upc_mem(cpy|put|get)` are even better

- Cast pointer-to-shared to pointer-to-local
  
  ```cpp
  #include<upc_castable.h> // in UPC 1.3
  void *upc_cast(const shared void *ptr);
  ```

---

**Non-blocking Memcpy is crucial to performance**

Hardware can reorder operations to improve performance (e.g., network adaptive routing), but possible data dependencies may prohibit it.

By using non-blocking memcpy, the user gives the permission to complete memory operations in arbitrary order.

---

**UPC 1.3 Non-blocking Memcpy**

```cpp
#include<upc_nb.h>

upc_handle_t h =
upc_memcpy_nb(shared void * restrict dst, 
shared const void * restrict src, 
size_t n);
void upc_sync(upc_handle_t h); // blocking wait
int upc_sync_attempt(upc_handle_t h); // non-blocking

// Implicit handle version, no handle management by user
void upc_memcpy_nbi(...); // parameters the same as upc_memcpy
void upc_synci(); // sync all issued implicit operations
int upc_sync_attempti(); // test the completion status of // implicit operations
```

---

**UPC 1.3 Atomic Operations**

- More efficient than using locks when applicable
  
  ```cpp
  upc_lock();
  update();
  upc_unlock();
  ```

  vs

  ```cpp
  atomic_update();
  ```

- Hardware support for atomic operations are available, **but**

  Only support limited operations on a subset of data types. e.g.,

  Atomic ops from different processors **may not** be atomic to each other
UPC 1.3 Atomic Operations (cont.)

• Key new idea: atomicity domain
  Users specify the operand data type and the set of
  operations over which atomicity is needed

  // atomicity domain for incrementing 64-bit integers
  upc_atomicdomain_t *domain =
  upc_all_atomicdomain_alloc(UPC_INT64, UPC_INC, 0);

  upc_atomic_strict(upc_atomicdomain_t *domain,
                  void * restrict fetch_ptr,
                  upc_op_t op,
                  shared void * restrict target,
                  const void * restrict operand1,
                  const void * restrict operand2);

  upc_atomic_relaxed(...); // relaxed consistency version

Performance of UPC

PGAS Languages have Performance Advantages

Strategy for acceptance of a new language
  • Make it run faster than anything else

  Keys to high performance
  • Parallelism:
    – Scaling the number of processors
  • Maximize single node performance
    – Generate friendly code or use tuned libraries
      (BLAS, FFTW, etc.)
  • Avoid (unnecessary) communication cost
    – Latency, bandwidth, overhead
    – Berkeley UPC and Titanium use GASNet
      communication layer
  • Avoid unnecessary delays due to dependencies
    – Load balance; Pipeline algorithmic dependencies
### One-Sided vs Two-Sided

**one-sided put message**
- address
- data payload

**two-sided message**
- message id
- data payload

- A one-sided put/get message can be handled directly by a network interface with RDMA support
- Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
- Ordering requirements on messages can also hinder bandwidth

---

### Why Should You Care about PGAS?

**Latency between Two MICs via Infiniband**
- MPI_Send/Recv
- upc_memput

**Latency between Two Nodes on Edison**
- (Cray XC30)

---

### Bandwidths on Cray XE6 (Hopper)

![Graph showing bandwidths on Cray XE6](image)

- Berkeley UPC
- Cray UPC
- Cray MPI

---

### One-Sided vs. Two-Sided: Practice

- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point (N½) differs by *one order of magnitude*
- This is not a criticism of the implementation!

Joint work with Paul Hargrove and Dan Bonachea
GASNet: Portability and High-Performance

GASNet better for latency across machines

Joint work with UPC Group; GASNet design by Dan Bonachea

GASNet excels at mid-range sizes: important for overlap

Joint work with UPC Group; GASNet design by Dan Bonachea

Communication Strategies for 3D FFT

• Three approaches:
  • **Chunk:**
    • Wait for 2ⁿ⁻¹ FFTs to finish
    • Minimize # messages
  • **Slab:**
    • Wait for chunk of rows destined for 1 proc to finish
    • Overlap with computation
  • **Pencil:**
    • Send each row as it completes
    • Maximize overlap and
    • Match natural layout

chunk = all rows with same destination

pencil = 1 row

slab = all rows in a single plane with same destination

Joint work with Chris Bell, Rajesh Nathala, Dan Bonachea
**Overlapping Communication**

- Goal: make use of “all the wires all the time”
  - Schedule communication to avoid network backup
- Trade-off: overhead vs. overlap
  - Exchange has fewest messages, less message overhead
  - Slabs and pencils have more overlap; pencils the most
- Example: Class D problem on 256 Processors

<table>
<thead>
<tr>
<th>Method</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exchange</td>
<td>512 Kbytes</td>
</tr>
<tr>
<td>Slabs (contiguous rows that go to 1 processor)</td>
<td>64 Kbytes</td>
</tr>
<tr>
<td>Pencils (single row)</td>
<td>16 Kbytes</td>
</tr>
</tbody>
</table>

**NAS FT Variants Performance Summary**

- Slab is always best for MPI; small message cost too high
- Pencil is always best for UPC; more overlap

**FFT Performance on BlueGene/P**

- UPC implementation consistently outperform MPI
- Uses highly optimized local FFT library on each node
- UPC version avoids send/receive synchronization
  - Lower overhead
  - Better overlap
  - Better bisection bandwidth
- Numbers are getting close to HPC record on BG/P

**FFT Performance on Cray XT4**

- 1024 Cores of the Cray XT4
  - Uses FFTW for local FFTs
  - Larger the problem size the more effective the overlap
Event Driven LU in UPC

- DAG Scheduling before it’s time
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - "memory constrained" lookahead

UPC HPL Performance

- Comparison to ScaLAPACK on an Altix, a 2 x 4 process grid
  - ScaLAPACK (block size 64) 25.25 GFlop/s (tried several block sizes)
  - UPC LU (block size 256) - 33.60 GFlop/s, (block size 64) - 26.47 GFlop/s
- n = 32000 on a 4x4 process grid
  - ScaLAPACK - 43.34 GFlop/s (block size = 64)
  - UPC - 70.26 Gflop/s (block size = 200)

MILC (QCD) Performance in UPC

- MILC is Lattice Quantum Chromo-Dynamics application
- UPC scales better than MPI when carefully optimized

Communication Overlap Complements Avoidance

- Even with communication-optimal algorithms (minimized bandwidth) there are still benefits to overlap and other things that speed up networks
- Communication Avoiding and Overlapping for Numerical Linear Algebra, Georganas et al. 2012
Summary

- UPC designed to be consistent with C
  - Ability to use pointers and arrays interchangeably
- Designed for high performance
  - Memory consistency explicit; Small implementation
  - Transparent runtime
- gcc version of UPC:
- Berkeley compiler
  [http://upc.lbl.gov](http://upc.lbl.gov)
- Language specification and other documents
  [https://code.google.com/p/upc-specification](https://code.google.com/p/upc-specification)
  [https://upc-lang.org](https://upc-lang.org)
- Vendor compilers: Cray, IBM, HP, SGI,…

Application Development in UPC

Topics

- Starting a project
  - Choosing the right SDK
  - Interoperability with other programming models
    - OpenMP, MPI, CUDA,…
- Shared memory programming
  - Data layout and allocation
  - Computational efficiency ("serial" performance
  - Synchronization
  - Managing parallelism – data parallel & dynamic tasking
  - UPC and OpenMP

Topics (2)

- Distributed memory programming
  - UPC and MPI
- Tuning communication performance
- Hybrid parallelism
UPC SDKs

- Multiple SDKs are available
  - Portable
    - BUPC provided by LBL is portable – available at [http://upc.lbl.gov](http://upc.lbl.gov)
    - GUPC provided by Intrepid, gcc based, portable, uses BUPC runtime
  - Vendor SDKs – Cray UPC XT/XE

- UPC has been shown to interoperate with
  - MPI, OpenMP, CUDA, Intel TBB, Habanero-C
  - Any pthreads based library e.g. MKL

- Some interoperability aspects are implementation specific, e.g. who owns main()
  - E.g. [http://upc.lbl.gov/docs/user/interoperability.shtml](http://upc.lbl.gov/docs/user/interoperability.shtml)

Shared Memory Programming

- Performance determined by
  - Locality – placement, data initialization
  - Computational efficiency
  - Synchronization performance
  - Management of parallelism

  When should memory be shared (shared) ?
  When should memory be blocked (shared []) ?

Pointer Arithmetic and Data Placement

- Memory is allocated with upc_alloc, upc_all_alloc with affinity to a certain thread
- The pointer type determines the address arithmetic rules and the “locality” of access

```c
shared double *pl;
shared [*] double *ps;
shared [] double *pi;
for(i=0; i < N; i++) {
    pl[i] = i; 1 3 2 4
    ps[i] = i; 1 2 3 4
    pi[i] = i; 1 2 3 4
}
```
2-D Stencil – Laplace Filter – block cyclic

shared double matrix[ROWS][COLS];

main() {
for(i=0; i < ROWS; i++)
    for(j = 0; j < COLS; j++) {
        up = (i == 0) ? 0 : matrix[i-1][j];
        down = (i == ROWS-1) ? 0 : matrix[i+1][j];
        left = (j == 0) ? 0 : matrix[i][j-1];
        right = (j == COLS - 1) ? 0 : matrix[i][j+1];
        tmp[i][j] = 4 * matrix[i][j] - up - down - left - right;
    }
}

Block cyclic layout easy to choose when porting codes, bad for locality

2-D Stencil – Laplace Filter – directory

typedef shared [] double * SDPT;
shared SDPT matrix[ROWS];
SDPT local_dir[ROWS];

main() {
    .matrix[my_row] = upc_alloc(.);
    upc_barrier;
    .local_dir[i] = matrix[i];

for(i=0; i < ROWS; i++)
    for(j = 0; j < COLS; j++) {
        up = (i == 0) ? 0 : local_dir[i-1][j];
        down = (i == ROWS-1) ? 0 : local_dir[i+1][j];
        left = (j == 0) ? 0 : local_dir[i][j-1];
        right = (j == COLS - 1) ? 0 : local_dir[i][j+1];
        tmp[i][j] = 4 * local_dir[i][j] - up - down - left - right;
    }

Directory based approach provides locality and portability

2-D Stencil – Laplace Filter – block layout

shared [*] double matrix[ROWS][COLS];

main() {
for(i=0; i < ROWS; i++)
    for(j = 0; j < COLS; j++) {
        up = (i == 0) ? 0 : matrix[i-1][j];
        down = (i == ROWS-1) ? 0 : matrix[i+1][j];
        left = (j == 0) ? 0 : matrix[i][j-1];
        right = (j == COLS - 1) ? 0 : matrix[i][j+1];
        tmp[i][j] = 4 * matrix[i][j] - up - down - left - right;
    }

Blocked layout easy to choose when porting codes, good for locality, code not portable

Computational Efficiency
(ALWAYS Cast to C)
**Computational Intensity – ALWAYS cast to C**

Cast a pointer-to-shared to a regular C pointer for accessing the local portion of a shared object.

E.g., `int *p = (int *)pts; p[0] = 1;`

---

**Application Examples**

---

**LULESH - https://codesign.llnl.gov/lulesh.php**

- Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
- Models explicit hydrodynamics portion of ALE3D
- Particular application is a Sedov blast wave problem
- Used to explore various programming models, e.g. Charm++, Chapel, Loci, Liszt
- Solves equations on a staggered 3D spatial mesh
- Most communication is nearest neighbor on a hexahedral 3D grid

---

**LULESH OMP**

- Doesn’t scale beyond 12 cores (2 NUMA nodes)
LULESHPARALLELINITIALIZATION

- Parallel initialization helps only slightly
- Still doesn't scale beyond 18 cores
- Uses temporary arrays with malloc and free in many calls

LULESHERTSCOMMENTS

- LULESH authors advise: "Do not make simplifications"
- None-the-less, I made some simplifications:
  - Primarily for readability and clarity
  - Why follow certain impl. choices? (e.g. temp arrays)
- Performance improvements in UPC at scale
  - Primarily due to locality management, not simplifications
- UPC with one thread is slower than C++ serial
  - Best UPC 298s, best C++ serial 283s

LULESHELAYOUT

- Cyclic layout poor fit for communication pattern
- Contiguous layout (blocked) reduces communication

```c
shared [*] double x[N * THREADS];
```
**XSBench - Embarrassingly parallel**

- Monte Carlo simulation of paths of neutrons traveling across a reactor core
  - 85% of runtime in calculation of macroscopic neutron cross sections

```plaintext
random_sample
binary_search
for each nuclide
  lookup_bounding_micro_xs
  interpolate
  accumulate_macro_xs
```

- Uses a lot of memory

**XSBench OMP Doesn’t Scale**

- Option to add flops; according to README:
  “Adding flops has so far shown to increase scaling, indicating that there is in fact a bottleneck being caused by the memory loads”
XSBench OMP Initialization

- But memory locality is the problem (on NUMA)
- Adding parallel initialization makes it scale

XSBench UPC

- Private replication of data
- Except: make largest memory array shared

XSBench UPC No Shared Memory

- Improves if all memory is private
- Can’t do for large problems, e.g. 355 isotopes requires 60GB for full replication on 48 cores

Synchronization Performance
Barriers, locks, atomics, collectives….

- **OpenMP** provides an implicit model of synchronization
- The **UPC** language provides rich synchronization primitives
  - e.g. UPC 1.3 atomics
- **Some are well optimized for multicore performance**
  "Optimizing Collective Communication on Multicore", Nishtala&Yelick, HotPar'09
- In general, UPC synchronization performs much better than OpenMP synchronization or other pthreads based libraries (implementation does matter)

  ```
  #pragma omp critical
  -> bupc_allv_reduce_all()
  ```

LULESH UPC Procs vs Pthreads

- At 48 cores, pthreads takes 33s, processes only 22s
- Top non-app code functions with pthreads:
  - `upcr_wait_internal` 15% (barrier)
  - `gasnete_coll_broadcast` 2%
  - `gasnete_coll_gather` 2%
- Top non-app code functions with pinned procs:
  - `gasnete_pshmbARRIER_WAIT` 5%
- For comparison, collectives with pinned procs:
  - `gasnete_coll_broadcast` 0.2% (15x)
  - `gasnete_coll_gather` 0.04% (75x)

Lessons Learned

- On a large NUMA system, managing remote memory access is key
  - Good preparation for distributed memory?
- **UPC**
  - Contiguous blocking is effective at reducing communication
  - Explicitly cast to private whenever possible
  - Procs can be significantly faster than pthreads

  *Hybrid PGAS Runtime Support for Multicore Nodes*
  Blagojevic, Hargrove, Iancu, Yelick, PGAS 2010

  - Replication to private can help, but limited by available memory -> replicate fixed amount?
Managing Parallelism

- **Data parallel** constructs in UPC – **upc_forall**
  - SPMD, shorthand for filtering the computation performed by a task
  - Not real equivalent of #pragma omp for..

- **Task parallelism** in OMP: #pragma omp task

- **UPC tasking library** – available at [http://upc.lbl.gov](http://upc.lbl.gov)
  - Written in stock UPC, works on
    - shared memory - comparable to OpenMP tasking
    - distributed memory – akin to Charm++
  - Provides:
    - Init, termination
    - Locality aware distributed work-stealing
    - Synchronization for dependent task graphs

Task Library API

taskq_t *taskq_all_alloc(int nFunc, void *func1, int input_size1, int output_size1, ...);

int taskq_put(taskq_t *taskq, void *func, void *in, void *out);

int taskq_execute(taskq_t *taskq);

void taskq_steal(taskq_t *taskq);

void taskq_fence(taskq_t *taskq);

int taskq_all_isEmpty(taskq_t *taskq);

UPC Task Library – Shared Memory

**Performance of Victim Selection Policies on 8 Core Nehalem SMP**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Exec. Time Normalized to gcc-OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc-OpenMP</td>
<td>1.0</td>
</tr>
<tr>
<td>icc-OpenMP</td>
<td>1.2</td>
</tr>
<tr>
<td>UPC (Intra-Socket)</td>
<td>1.5</td>
</tr>
<tr>
<td>UPC (HVS)</td>
<td>2.0</td>
</tr>
<tr>
<td>UPC (RAND)</td>
<td>2.5</td>
</tr>
<tr>
<td>UPC (RAND+BestChunk)</td>
<td>3.0</td>
</tr>
</tbody>
</table>

UPC Task Library – Distributed Memory

**Performance of Victim Selection Policies on 256 cores on Carver Cluster**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Speedup Normalized to the Random Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRA-NODE</td>
<td>0.8</td>
</tr>
<tr>
<td>HVS</td>
<td>1.0</td>
</tr>
<tr>
<td>RANDOM</td>
<td>1.2</td>
</tr>
</tbody>
</table>
Distributed Memory Programming

UPC and MPI

- Send/Recv carry both data and synchronization
- One-sided carries only data
- When porting codes from MPI two-sided to one sided, a Send/Recv pair needs to be replaced with Put/Get and producer-consumer semantics

- There are also performance differences
  - UPC can saturate the network with fewer cores active per node
  - It alleviates the need for packing messages

Cray XE6 BW Saturation (hopper @ NERSC)

Cray XE6 Application Performance
Tuning Communication Performance

**UPC Trends**

- In MPI, large messages or large message concurrency (messages per core, ranks per node) is required for performance.
- In UPC, communication overlap is beneficial:
  - with other communication
  - with other computation
- In UPC:
  - Pays to think about increasing the message concurrency
  - Sometimes need to take care to avoid congestion

*Congestion Avoidance on Manycore HPC Systems*

Luo, Panda, Ibrahim, Iancu. ICS’12

Again, avoiding pthreads improves performance

**Saturation IB**

*IB Put Saturation - 2 Nodes*

- Messages < 1024 benefit from concurrent injection
- Messages > 8K benefit from throttling

**Throughput and Message Concurrency**

*Throughput Variation with Msg/Core - Gemini*

- Limiting the number of outstanding messages provides 5X speedup (expected 32X slower)
When To Use It?

- With irregular parallelism with "natural small messages"
- When hybrid parallelism makes packing complex

- Need to mix with pthreads based libraries and want to perform communication from within pthreads
  - Implementation specific, but available

- Do not want to worry about matching communication concurrency to intra-node concurrency…

Challenges:
- Exporting data, do not want to modify data structures
- One-sided is different, need to understand it…

Beyond UPC

DEGAS Programming System: UPC++

A template-based programming system enabling PGAS features for C++ applications

DEGAS is a DOE-funded X-Stack project led by Lawrence Berkeley National Lab (PI: Kathy Yelick), in collaboration with LLNL, Rice Univ., UC Berkeley, and UT Austin.

C++ is Important in Scientific Computing

Languages use at NERSC: 75% Fortran, 45% C/C++, 10% Python with C++ at least as important as C

- DOE’s Exascale Co-Design Centers
  - ExaCT: Combustion simulation (uniform and adaptive mesh)
  - ExMatEx: Materials (multiple codes)
  - CESAR: Nuclear engineering (structures, fluids, transport)
  - NNSA Center: umbrella for 3 labs

- “Proxy apps” to represent them
  - 10 codes
  - 4 in C++
UPC++: PGAS with Enhancements

- Function shipping across nodes
- Multi-threading option
- Local task queue
- Global address space
- Multidimensional arrays

A “Compiler-Free” Approach for PGAS

- Leverage the C++ standard and compilers
  - Implement UPC++ as a C++ template library
  - C++ templates can be used as a mini-language to extend the C++ grammar
- New features in C++ 11 makes UPC++ more powerful
  - E.g., async, auto type inference, lambda functions
  - C++ 11 is well-supported by major compilers

UPC++ Software Stack

- UPC Apps
- C/C++ Apps
- C/C++ Compiler
- UPC Compiler
- UPCC++ Template Header Files
- C++ Runtime
- UPC++ Runtime
- GASNet Communication Library
- Network Drivers and OS Libraries

C++11 standard: 701 pages
C++11 standard: 1334 pages

UPC++ Introduction

UPC++ “Language” (no compiler involved)

- Shared variable
  
  shared_var<int> s; // int in the shared space

- Global pointers (to remote data)
  
  global_ptr<LLNode> g; // pointer to shared space

- Shared arrays
  
  shared_array<int> sa(8); // array in shared space

- Locks
  
  shared_lock l; // lock in shared space

- Default execution model is SPMD, but with optional async

  async(place)(Function f, T1 arg1, ...);

  wait(); // other side does poll()
UPC++ Translation Example

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

C++ Compiler

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

UPC++ Runtime

```cpp
if (tmp_ref.local?)
    Local Access
else
    Remote Access
```

Dynamic Global Memory Management

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

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

Example: allocate space for 512 integers on rank 2
```cpp
global_ptr<int> p = allocate<int>(2, 512);
```

Remote memory allocation is not available in MPI-3, UPC or SHMEM.

Optimization Opportunities for Async_copy

```cpp
upcxx::async_copy<T>(global_ptr<T> src,
                      global_ptr<T> dst,
                      size_t count);
```

Template specialization plus runtime compilation may translate this into a few load and store instructions!

This would be very difficult to do with a heavy-weight MPI API

```cpp
MPI_Put(origin_addr, origin_count, origin_datatype,
         target_rank, target_disp, target_count,
         target_datatype, win)
```

One-Sided Data Transfer Functions

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

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

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

Similar to upc_memcpy_nb extension in UPC 1.3
### UPC++ Equivalents for UPC Users

<table>
<thead>
<tr>
<th>UPC</th>
<th>UPC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. of threads</td>
<td>THREADS</td>
</tr>
<tr>
<td>My ID</td>
<td>MYTHREAD</td>
</tr>
<tr>
<td>Shared variable</td>
<td>shared _Type s</td>
</tr>
<tr>
<td>Shared array</td>
<td>shared _[BS] Type A(sz)</td>
</tr>
<tr>
<td>Pointer-to-shared</td>
<td>shared _Type *pts</td>
</tr>
<tr>
<td>Dynamic memory</td>
<td>shared void *</td>
</tr>
<tr>
<td>allocation</td>
<td>upc_allo(type nbytes)</td>
</tr>
<tr>
<td>Bulk data transfer</td>
<td>upc_memcpy(dst, src, nbytes);</td>
</tr>
<tr>
<td>Affinity query</td>
<td>upc_threadof(ptr)</td>
</tr>
<tr>
<td>Synchronization</td>
<td>upc_lock_t</td>
</tr>
<tr>
<td></td>
<td>upc_barrier</td>
</tr>
</tbody>
</table>

*Homework: how to translate `upc forall`?*

### Asynchronous Task Execution

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

- **UPC++ async function**
  ```cpp
  // Remote Procedure Call
  upcxx::async(place)(Function f, T1 arg1, T2 arg2,...);
  upcxx::wait();
  ```

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

### Async Task Example

```cpp
#include <upcxx.h>
#include <forkjoin.h> // using the fork-join execution model

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

int main(int argc, char **argv)
{
    int tgs[1, THREADS, 2]; // threads 1,3,5,...
    // call a function on a group of remote processes
    upcxx::async(tgs)(print_num, 123);
    upcxx::wait(); // wait for the remote tasks to complete
    return 0;
}
```

### Async with Lambda Function

```cpp
#include <upcxx.h>
#include <forkjoin.h> // using the fork-join execution model

int main(int argc, char **argv)
{
    for (int i = 0; i < THREADS; i++) {
        // spawn a task at place "i"
        // the task is expressed by a lambda (anonymous) function
        upcxx::async(i)[] (int num) { printf("num: %d\n", num); },
        1000+i); // argument to the lambda function
    }
    upcxx::wait(); // wait for all tasks to finish
    return 0;
}
```

```cpp
mpirun -n 4 ./test_async
```

**Output:**
```
num: 1000
num: 1001
num: 1002
num: 1003
```
X10-style Finish-Async Programming Idiom

using namespace upcxx;

// Thread 0 spawns async tasks
finish {
    for (int i = 0; i < THREADS; i++) {
        async(i)[] (int num) {
            printf("num: %d\n", num);
            1000+i);
        }
    } // All async tasks are completed
}

How We Did It?

// finish { => macro expansion =>
for (f_scope _fs; _fs.done == 0; _fs.done = 1) {
    // f_scope constructor call generated by compiler
    // push the current scope in a stack
    f_scope() { push_event(&_fs.e); }
    for (int i = 0; i < THREADS; i++) {
        // register the async with the current scope
        async(i, e = peek_event()(...));
    }
    // f_scope destructor call generated by compiler
    ~f_scope() { pop_event(); _fs.e.wait(); }
    // All registered tasks are waited for completion
}

Leverage C++ Programming Idiom Resource
Acquisition Is Initialization (RAII)

Random Access Benchmark (GUPS)

// shared uint64_t Table[TableSize]; in UPC
shared_array<uint64_t> Table(TableSize);

void RandomAccessUpdate()
{
    uint64_t ran, i;
    ran = starts(NUPDATE / THREADS * MYTHREAD);
    for(i = MYTHREAD; i < NUPDATE; i += THREADS) {
        ran = (ran << 1) ^ ((int64_t)ran < 0 ? POLY : 0);
        Table[ran & (TableSize-1)] ^= ran;
    }
}

Manycore - A Good Fit for PGAS

Main update loop

Logical data layout

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Physical data layout

Thread 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

PGAS Abstraction

Logic Block Diagram of the Intel Knights Corner (MIC) micro-architecture
GUPS Performance on MIC

<table>
<thead>
<tr>
<th>Num. of Processes</th>
<th>Random Access Latency</th>
<th>Giga Updates Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>0.96</td>
</tr>
<tr>
<td>8</td>
<td>0.04</td>
<td>0.94</td>
</tr>
<tr>
<td>16</td>
<td>0.08</td>
<td>0.92</td>
</tr>
<tr>
<td>32</td>
<td>0.16</td>
<td>0.90</td>
</tr>
<tr>
<td>60</td>
<td>0.32</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Difference between UPC++ and UPC is only about 0.2 µs (~220 cycles)

GUPS Performance on BlueGene/Q

<table>
<thead>
<tr>
<th>Num. of Processes</th>
<th>Random Access Latency</th>
<th>Giga Updates Per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>0.96</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.94</td>
</tr>
<tr>
<td>16</td>
<td>0.00</td>
<td>0.92</td>
</tr>
<tr>
<td>32</td>
<td>0.00</td>
<td>0.90</td>
</tr>
<tr>
<td>60</td>
<td>0.00</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Difference is negligible at large scale

UPC++ Application: Embree

- Intel open-source ray tracing toolkit written in C++
- Ported to UPC++ by Michael Driscoll
- Performance scaled on Edison (Cray XC30)

Embree Performance on Edison

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Performance Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>384</td>
<td>384</td>
</tr>
<tr>
<td>768</td>
<td>768</td>
</tr>
<tr>
<td>1536</td>
<td>1536</td>
</tr>
<tr>
<td>3072</td>
<td>3072</td>
</tr>
<tr>
<td>6144</td>
<td>6144</td>
</tr>
</tbody>
</table>

Hybrid UPC++ for internode communication and OpenMP within a NUMA node
LULESH Proxy Application

• Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
• Proxy App for UHPC, ExMatEx, and LLNL ASC
• Written in C++ with MPI, OpenMP, and CUDA versions

LULESH 3-D Data Partitioning

LULESH Communication Pattern

Cross-section view of the 3-D processor grid

26 neighbors
• 6 faces
• 12 edges
• 8 corners

Data Layout of Each Partition

• 3-D array A[x][y][z]
• row-major storage
• z index goes the fastest

• Blue planes are contiguous
• Green planes are stride-N² chunks
• Red planes are stride-N elements
Convert MPI to UPC++

Pseudo code

```c
// Post Non-blocking Recv
MPI_Irecv(RecvBuf1);
...
MPI_Irecv(RecvBufN);

Pack_Data_to_Buf();
// Get neighbors' RecvBuf addresses
// Post Non-blocking Copy
upcxx::async_copy(SendBuf1, RecvBuf1);
...
upcxx::async_copy(SendBufN, RecvBufN);

MPI_Wait();
...
Unpack_Data();
```

Example: Building A Task Graph

```c
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
```

LULESH Performance on Cray XC30 (Edison)

Take advantage of PGAS without the pain of adopting a new language

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:

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

Observed Data
Predicted waveform
Seismic model
Minimize:

Collaboration with Scott French et al, Berkeley Seismological Lab
Problem 2: Combining Data Sets

• Merge measurement data into simulation and evaluate fit
• Matrix is too large for single shared memory
• Assembly: Strided writes into a global array
• Goal is scalability in context of full code

Application: Full-Waveform Seismic Imaging

New implementation
• Scales to larger dataset size and matrix dimension (currently ~2x in both)
• Earlier runs that required 4+ phases now achieved in a single phase on the same aggregate number of cores and with ~40% wallclock time reduction

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>NAS Parallel Benchmarks</th>
<th>MPI+Fortran</th>
<th>UPC</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB-CG</td>
<td>500</td>
<td>500</td>
<td>2000</td>
</tr>
<tr>
<td>NPB-FT</td>
<td>1000</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>NPB-MG</td>
<td>1500</td>
<td>1500</td>
<td>2000</td>
</tr>
</tbody>
</table>

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
Overview of UPC++ Array Library

- A point is an index, consisting of a tuple of integers
  
  ```cpp
  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
  
  ```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
  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 is not 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

```cpp
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

NASDAQ Benchmarks on One Node of Cray XC30

![Graph showing NAS Benchmarks comparison between Titanium and UPC++ for CG, FT, and MG algorithms.](image)
**Case Study: miniGMG**

- Compact 3D geometric multigrid code
  - Can be used to evaluate performance bottlenecks in MG+Krylov methods and prototype new algorithms.
  - Highly instrumented for detailed timing analysis.

- Can be configured to proxy BoxLib AMR applications
  - Finite-volume (cell-centered) multigrid
  - 7pt variable-coefficient Helmholtz operator (stencil)
  - Cubical domain decomposed into one $128^3$ subdomain per socket
  - Restriction terminated when subdomains are coarsened to $2^3$ (U-Cycle)
  - Gauss Seidel, Red-Black ("GSRB") smoother
  - BiCGStab bottom solver (matrix is never explicitly formed)

**miniGMG Results**

- Savings of ~200 lines of communication and setup code over **Bulk** and **Fine-Grained** versions
- Performance results on IBM Blue Gene/Q

**miniGMG Communication Paradigms**

- One programming system w. three communication paradigms
  - **Bulk** version that uses manual packing/unpacking with one-sided puts
  - **Fine-Grained** version that does multiple one-sided puts of contiguous data
  - **Array** version that logically copies entire ghost zones, delegating actual procedure to array library

**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](image-url)
UPC++ HPGMG (work in progress)

- Ghost Exchange
  - 380 lines for comm. setup
  - same level
- Restriction
  - 330 lines for comm. setup
  - between two levels
  - finer level to coarser level, 1-1 or many-1, different owner
- Interpolation
  - 330 lines for comm. setup
  - between two levels
  - coarser level to finer level, 1-1, 1-many, different owner

PGAS Summary

- Productivity through shared memory convenience
  - Especially for irregular communication
- Ensure scalability through locality control
- Expose lightweight RDMA communication
  - Possibly for “PGAS on a chip” systems
- Minimally invasive, interoperable features
- Open source and vendor (e.g., Cray) compilers
  - [http://upc.lbl.gov](http://upc.lbl.gov)
  - [http://www.gccupc.org](http://www.gccupc.org)
  - [https://bitbucket.org/upcxx](https://bitbucket.org/upcxx)

HPGMG Performance (Box size = 2^7)

Though the naturally-grained version is about 3X slower but it saves over 1000 lines of very difficult code (testified by the original HPGMG developer) and saves auxiliary data structures for packing and unpacking.

Can Interconnect innovations bridge the performance gap between large and small messages?

A Family of PGAS Languages

- UPC based on C philosophy / history
  - [http://upc-lang.org](http://upc-lang.org)
  - Free open source compiler: [http://upc.lbl.gov](http://upc.lbl.gov)
  - Also a gcc variant: [http://www.gccupc.org](http://www.gccupc.org)
- Java dialect: Titanium
  - [http://titanium.cs.berkeley.edu](http://titanium.cs.berkeley.edu)
- Co-Array Fortran
  - Part of Stanford Fortran (subset of features)
  - CAF 2.0 from Rice: [http://caf.rice.edu](http://caf.rice.edu)
- Chapel from Cray (own base language better than Java)
  - [http://chapel.cray.com](http://chapel.cray.com) (open source)
- X10 from IBM also at Rice (Java, Scala,…)
- Phalanx from Echelon projects at NVIDIA, LBNL,…
  - C++ PGAS languages with CUDA-like features for GPU clusters
- Coming soon…. PGAS for Python, aka PyGAS
Productivity of the Titanium Language

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

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

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

Productive Features in Titanium

- UPC++ already provides many of Titanium’s productivity features
  - Basic high-level language features (e.g. object orientation, memory management)
  - Templates and operator overloading
  - SPMD execution model and PGAS memory model
- Titanium features we want to implement in UPC++
  - True multidimensional rectangular arrays
    - Not distributed, but may be located on a remote thread
  - Hierarchical teams
  - Global object model (future work)

C and UPC Arrays

- C/C++ arrays are limited in many ways
  - Multidimensional arrays must specify sizes of all but first dimension as compile-time constants
    - These sizes are part of the type, which makes it hard to write generic code
  - Easy to get view of contiguous subset of an array, but non-contiguous view must be handled manually
- UPC shared arrays have their own limitations
  - Can only be distributed in one dimension
    - User must manually linearize a multidimensional array, use a directory structure, or both
  - Blocking factor must be a compile-time constant
  - `upc_memcpy` only supports contiguous source and destination

Example: Ghost Zones

- Copying ghost zones requires manually packing/unpacking elements at source/destination
  - In effect, turns one-sided operation into two-sided
- Strided copy is not enough for ghost cell thickness > 1
  - Need “side factors” to specify how many elements to skip at end of each dimension
Multidimensional Arrays in Titanium

- True multidimensional arrays
  - Supports subarrays without copies
    - Can refer to rows, columns, slabs, interior, boundary, etc.
  - Indexed by Points (tuples of ints)
  - Built on a rectangular set of Points, RectDomain
  - Points and RectDomains are built-in immutable classes, with useful literal syntax
- Support for AMR and other grid computations
  - Domain operations: intersection, shrink, border
- Arrays are located on a single thread, but can be a remote thread

Points, RectDomains, Arrays in General

- Points specified by a tuple of ints
  \[
  \text{Point<2> lb = [1, 1];}
  \text{Point<2> ub = [10, 20];}
  \]
- RectDomains given by 3 points:
  - lower bound, upper bound (and optional stride)
  \[
  \text{RectDomain<2> r = [lb : ub];}
  \]
- Array declared by number of dimensions and type
  \[
  \text{double [2d] a;}
  \]
- Array created by passing RectDomain
  \[
  a = \text{new double [r];}
  \]

Unordered Iteration

- Motivation:
  - Memory hierarchy optimizations are essential
  - Compilers sometimes do these, but hard in general
- Titanium has explicitly unordered iteration
  - Helps the compiler with analysis
  - Helps programmer avoid indexing details
  \[
  \text{foreach (p in r) { ... A[p] ... }}
  \]
  - \( p \) is a Point (tuple of ints), can be used as array index
  - \( r \) is a RectDomain
- Note: \textit{foreach} is not a parallelism construct

Simple Array Example

- Matrix sum in Titanium
  \[
  \text{Point<2> lb = [1,1];}
  \text{Point<2> ub = [10,20];}
  \text{RectDomain<2> r = [lb:ub];}
  \]
  \[
  \text{double [2d] a = new double [r];}
  \text{double [2d] b = new double [1:10,1:20];}
  \text{double [2d] c = new double [lb:ub:[1,1]];}
  \]
  \[
  \text{for (int i = 1; i <= 10; i++)}
  \text{for (int j = 1; j <= 20; j++)}
  \text{c[i,j] = a[i,j] + b[i,j];}
  \]
  \[
  \text{foreach (p in c.domain()) { c[p] = a[p] + b[p]; }}
  \]

Syntactic sugar

Optional stride

Equivalent loops
More Array Operations

- 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
- Most important array operation: one line copy between any two arrays with same element type and arity
  \[ \text{dst.copy(src)} \]
- Copies all elements in intersection of source and destination domains
- Both source and destination can be located on any thread

Example: Setting Boundary Conditions

```
foreach (l in local_grids.domain()) {
    foreach (a in all_grids.domain()) {
        local_grids[l].copy(all_grids[a]);
    }
}
```

Implementation of Titanium Arrays in UPC++

- UPC++ implementation built using C++ templates and operator overloading
  - Template parameters specify arity and element type
  - Overload element access operator []
- Macros provide simple syntax for domain/array literals
  - Titanium
    \[[1, 3] \]
    RectDomain<3> rd = {{1, 1, 1}, {3, 3, 3}};
    int[3d] local_arr = new int[{{1, 1, 1}, {3, 3, 3}}];
- UPC++
  \[[1, 3] \]
  POINT(1, 3)
  ndarray<int, 3> arr =
  ARRAY(int, {{1, 1, 1}, {3, 3, 3}});

Foreach Implementation

- Macros also allow definition of foreach loops
  \[ \text{#define foreach(p, dom)} \]
  \[ \text{foreach(p, dom, UNIQUIFYNS(foreach_ptr_, p))} \]
  \[ \text{#define foreach_(p, dom, ptr_)} \]
  \[ \text{for (auto ptr_ = (dom).iter(); !ptr_.done; \}
     \text{ptr_.done = true)} \]
  \[ \text{for (auto p = ptr_.start(); ptr_.next(p);)} \]
Preliminary Results

- Currently have full implementation of Titanium-style domains and arrays in UPC++
- Additionally have ported useful pieces of the Titanium library to UPC++
  - e.g. timers, higher-level collective operations
- Four kernels ported from Titanium to UPC++
  - 3D 7-point stencil, NAS conjugate gradient, Fourier transform, and multigrid
- Minimal porting effort for these examples
  - Less than a day for each kernel
  - Array code only requires change in syntax
  - Most time spent porting Java features to C++
- Larger applications will require global object model to be defined and implemented in UPC++

Performance Tuning

- Since UPC++ is a library, cannot rely on compiler to optimize array accesses
  - Array library is very general, but generality results in overhead in simple cases
- Preliminary approach is to provide template specializations that allow users to bypass inefficient, general code
- In the future, we plan to explore automatic dynamic specialization
  - Potentially leverage SEJITS work at UCB

Example: CG SPMV

- Unspecialized local SPMV in conjugate gradient kernel
  ```cpp
  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)

- Specialized local SPMV
  ```cpp
  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)
Hierarchical Programming

• Applications can reduce communication costs by adapting to machine hierarchy
  - Slow, Fast, allow
  - Applications may also have inherent, algorithmic hierarchy
    - Recursive algorithms
    - Composition of multiple algorithms
    - Hierarchical division of data

Algorithm Example: Merge Sort

• Task parallel
  ```c
  int[] mergeSort(int[] data) {
    int len = data.length;
    if (len < threshold)
      return sequentialSort(data);
    d1 = fork mergeSort(data[0:len/2-1]);
    d2 = mergeSort(data[len/2:len-1]);
    join d1;
    return merge(d1, d2);
  }
  ```

• Cannot fork threads in SPMD
  - Must rewrite to execute over fixed set of threads

Algorithm Example: Merge Sort

• SPMD
  ```c
  int[] mergeSort(int[] data, int[] ids) {
    int len = data.length;
    int threads = ids.length;
    if (threads == 1) return sequentialSort(data);
    if (myId in ids[0:threads/2-1])
      d1 = mergeSort(data[0:len/2-1],
                     ids[0:threads/2-1]);
    else
      d2 = mergeSort(data[len/2:len-1],
                     ids[threads/2:threads-1]);
    barrier(ids);
    if (myId == ids[0]) return merge(d1, d2);
  }
  ```

Hierarchical Teams

• Thread teams are basic units of cooperation
  - Groups of threads that cooperatively execute code
  - Collective operations over teams
  - Structured, hierarchical teams provide many benefits over flat teams
    - Expressive: match structure of algorithms, machines
    - Safe: eliminate many sources of deadlock
    - Composable: enable existing code to be composed without being rewritten to explicitly use teams
    - Efficient: allow users to take advantage of machine structure, resulting in performance gains
Team Data Structure

- Threads comprise teams in tree-like structure
- First-class object to allow easy creation and manipulation

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

- Work in progress: add ability to automatically construct team hierarchy from machine structure

Team Usage Construct

- Syntactic construct specifies that all enclosed operations are with respect to the given team
  - Collectives and constants such as MYTHREAD are with respect to currently scoped team

```
teamsplit(row_team) {
  Reduce::add(mtmp, myresults, rpivot);
}
```

Team Construct Implementation

- `teamsplit` implemented exactly like `finish`

```
// teamsplit(row_team) { => macro expansion =>
for (ts_scope _ts(row_team); !_ts.done == 0;
    _ts.done = 1) {
  // ts_scope constructor call generated by compiler
  // descend one level in team hierarchy
  ts_scope(team &t) { descend_team(t->mychild()); }

  // collective operation on current team
  Reduce::add(mtmp, myresults, rpivot);
  // ts_scope destructor call generated by compiler
  ~ts_scope() { ascend_team(); }
}
```

Merge Sort Team Hierarchy

- Team hierarchy is binary tree
- Trivial construction

```
void divide_team(team &t) {
  if (THREADS > 1) {
    t.split(MYTHREAD % 2, MYTHREAD / 2);
    teamsplit(t) {
      divide_team(t.mychild());
    }
  }
}
```

- Threads walk down to bottom of hierarchy, sort, then walk back up, merging along the way
Merge Sort Implementation

• Control logic for sorting and merging
  ```cpp
def sort_and_merge(team &t) {
    if (THREADS == 1) {
        allres[myidx] = sequential_sort(mydata);
    } else {
        teamsplit(t) {
            sort_and_merge(t.mychild());
        }
        barrier();
        if (MYTHREAD == 0) {
            int other = myidx + t.mychild().size();
            narray<int, 1> myres = allres[myidx];
            narray<int, 1> otherres = allres[other];
            narray<int, 1> newres = target(depth(t), myres, otherres);
            allres[myidx] = merge(myres, otherres, newres);
        }
    }
  }
  ```

Hierarchical Teams Results (Titanium)

• Titanium has full hierarchical team implementation, including machine model
• Hierarchical sort algorithm has both algorithmic hierarchy (merge sort) and machine-level hierarchy (mixed sample sort and merge sort)

Distributed Sort (Cray XE6)

Future Work

• Arrays
  – Investigate dynamic optimization using just-in-time specialization
  – Design and build distributed array library on top of current library
• Hierarchical teams
  – Design hierarchical machine model for UPC++
  – Add ability to query machine structure at runtime
• Global object model
  – Explore template metaprogramming techniques for implementing a global object interface
  – Build a tool for generating global analogs from local class definitions

Summary

• 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++?
Application Work in PGAS

- Network simulator in UPC (Steve Hofmeyr, LBNL)
- Real-space multigrid (RMG) quantum mechanics (Shirley Moore, UTK)
- Landscape analysis, i.e., “Contributing Area Estimation” in UPC (Brian Kazian, UCB)
- GTS Shifter in CAF (Preissl, Wichmann, Long, Shalf, Ethier, Konigis, LBNL, Cray, PPPL)

Two Distinct Parallel Programming Questions

- What is the parallel control model?
  - SPMD “default” plus data parallelism through collectives and dynamic tasking within nodes or between nodes through libraries
    - data parallel
    - dynamic
    - single program (single thread of control)
    - multiple data (SPMD)

- What is the model for sharing/communication?
  - PGAS load/store with partitioning for locality, but need a “signaling store” for producer consumer parallelism

PyGAS: Combine two popular ideas

- Python
  - No. 6 Popular on http://langpop.com and extensive libraries, e.g., Numpy, Scipy, Matplotlib, NetworkX
  - 10% of NERSC projects use Python
- PGAS
  - Convenient data and object sharing
- PyGAS: Objects can be shared via Proxies with operations intercepted and dispatched over the network:
  ```python
  num = 1+2*j
  print pxy.real # shared read
  pxy.imag = 3   # shared write
  print pxy.conjugate() # invoke
  ```

  - Leveraging duck typing:
    - Proxies behave like original objects.
    - Many libraries will automatically work.

Arrays in a Global Address Space

- Key features of Titanium arrays
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to identify ghosts and iterate:
  ```python
  foreach (p in gridA.shrink(1).domain()) ...
  ```
- Array copies automatically work on intersection
  ```python
  gridB.copy(gridA.shrink(1));
  ```

- Useful in grid computations including AMR

Joint work with Titanium group
Languages Support Helps Productivity

C++/Fortran/MPI AMR
- Chombo package from LLNL
- Bulk-synchronous comms
  - Pack boundary data between procs
  - All optimizations done by programmer

Titanium AMR
- Entirely in Titanium
- Finer-grained communication
  - No explicit pack/unpack code
  - Automated in runtime system

• General approach
  - Language allow programmer optimizations
  - Compiler/runtime does some automatically

Work by Tong Wen and Philip Colella; Communication optimizations joint with Jimmy Su

Particle/Mesh Method: Heart Simulation

• Elastic structures in an incompressible fluid.
  - Blood flow, clotting, inner ear, embryo growth, ...

• Complicated parallelization
  - Particle/Mesh method, but “Particles” connected into materials (1D or 2D structures)
  - Communication patterns irregular between particles (structures) and mesh (fluid)

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen

Compiler-free “UPC++” eases interoperability

global_array_t<int, 1> A(10); // shared [1] int A[10];
L-value reference (write/put)
R-value reference (read/get)
int n = A[1] + 1; // A[1] -> global_ref_t ref(A, 1); n = (int)ref + 1;

Hierarchical SPMD (demonstrated in Titanium)

• Thread teams may execute distinct tasks
  partition(T) {
    { model_fluid(); }
    { model_muscles(); }
    { model_electrical(); }
  }

• Hierarchy for machine / tasks
  - Nearby: access shared data
  - Far away: copy data

• Advantages:
  - Provable pointer types
  - Mixed data / task style
  - Lexical scope prevents some deadlocks
Hierarchical machines → Hierarchical programs

- Hierarchical memory model may be necessary (what to expose vs hide)
- Two approaches to supporting the hierarchical control
  - Option 1: Dynamic parallelism creation
    - Recursively divide until you run out of work (or hardware)
    - Runtime needs to match parallelism to hardware hierarchy
  - Option 2: Hierarchical SPMD with "Mix-ins"
    - Hardware threads can be grouped into units hierarchically
    - Add dynamic parallelism with voluntary tasking on a group
    - Add data parallelism with collectives on a group

Option 1 spreads threads, option 2 collects them together

One-sided communication works everywhere

**PGAS programming model**

```c
*p1 = *p2 + 1;
A[i] = B[i];
upc_memput(A, B, 64);
```

It is implemented using one-sided communication: put/get

Support for one-sided communication (DMA) appears in:
- Fast one-sided network communication (RDMA, Remote DMA)
- Move data to/from accelerators
- Move data to/from I/O system (Flash, disks,..)
- Movement of data in/out of local-store (scratchpad) memory

Vertical PGAS

- New type of wide pointer?
  - Points to slow (offchip memory)
  - The type system could get unwieldy quickly

<table>
<thead>
<tr>
<th>l:</th>
<th>m:</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 1</td>
<td>x: 5</td>
<td>x: 7</td>
</tr>
<tr>
<td>y: 0</td>
<td>y:</td>
<td></td>
</tr>
</tbody>
</table>

Private on-chip

- Shared partitioned on-chip
- Shared off-chip DRAM or NVRAM

PGAS Languages

- **Global address space**: thread may directly read/write remote data
- Hides the distinction between shared/distributed memory
- **Partitioned**: data is designated as local or global
- Does not hide this: critical for locality and scaling

**UPC, CAF, Titanium**: Static parallelism (1 thread per proc)
- Does not virtualize processors
- **X10, Chapel and Fortress**: PGAS, but not static (dynamic threads)
A Brief History of Languages

- When vector machines were king
  - Parallel "languages" were loop annotations (IVDEP)
  - Performance was fragile, but there was good user support

- When SIMD machines were king
  - Data parallel languages popular and successful (CMF, "Lisp, C*, …)
  - Quite powerful: can handle irregular data (sparse mat-vec multiply)
  - Irregular computation is less clear (multi-physics, adaptive meshes, backtracking search, sparse matrix factorization)

- When shared memory multiprocessors (SMPs) were king
  - Shared memory models, e.g., OpenMP, POSIX Threads, were popular

- When clusters took over
  - Message Passing (MPI) became dominant

- With multicore building blocks for clusters
  - Mixed MPI + OpenMP is the preferred choice