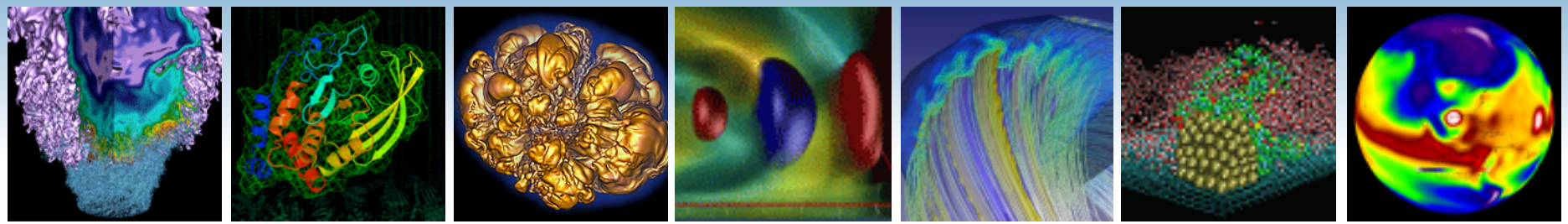


Supporting Irregular Applications with Partitioned Global Address Space Languages: UPC and UPC++

Kathy Yelick
Lawrence Berkeley National Laboratory

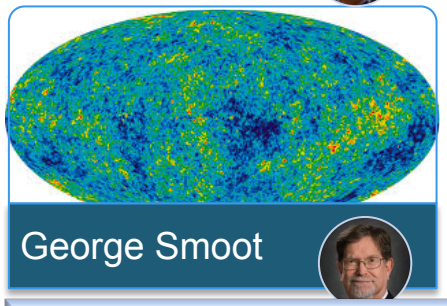
With results from the DEGAS and UPC groups

NERSC Celebrates 40th Birthday

5000 users, 1900+ publications per year

John Kuriyan for Martin Karplus

Saul Perlmutter



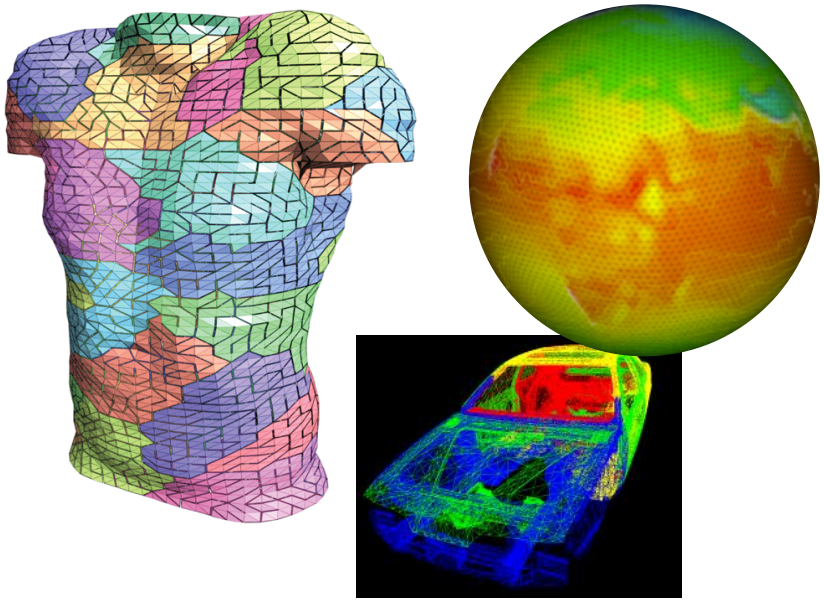
George Smoot

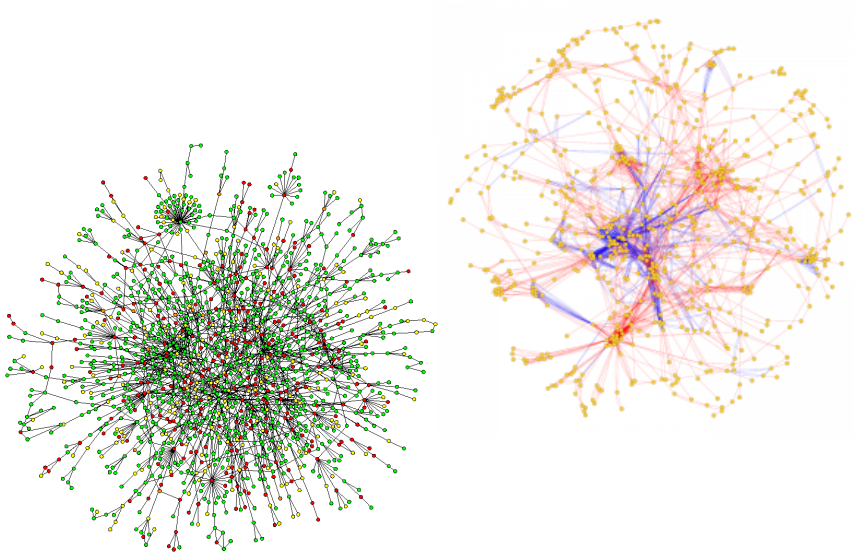
Warren Washington

Lectures available at www.nersc.gov

Petaflop and Petabyte systems for science

Programming Challenges and Solutions





Message Passing Programming

Divide up domain in pieces

Each compute one piece

Exchange (send/receive) data

PVM, MPI, and many libraries

Global Address Space Programming

Each start computing

Grab whatever you need whenever

Global Address Space Languages and Libraries

~10% of NERSC apps use some kind of PGAS-like model

Shared Memory vs. Message Passing

Shared Memory

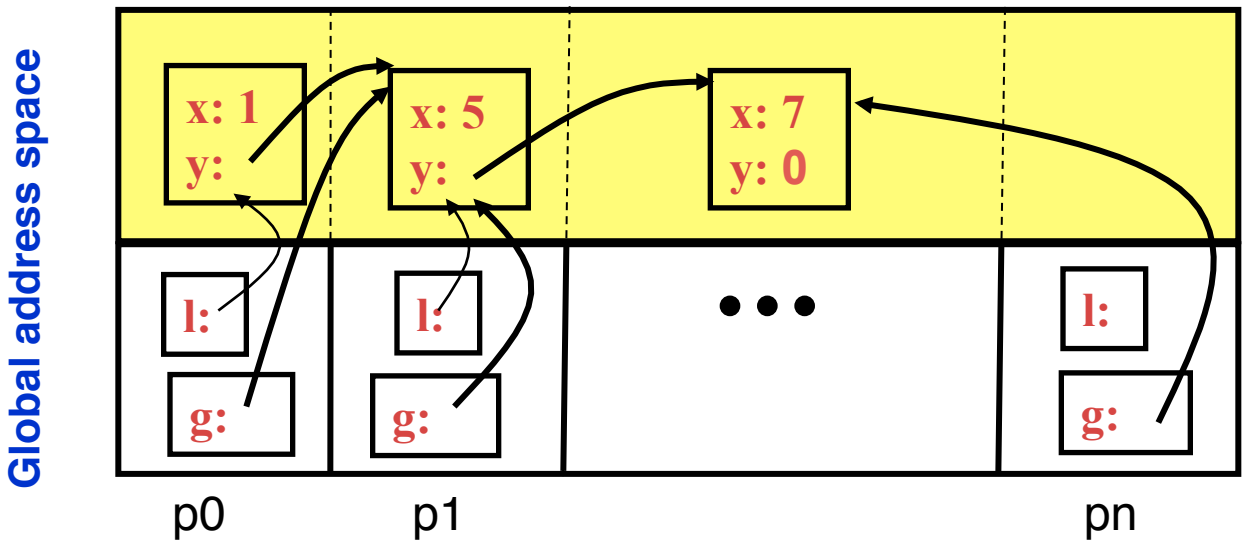
- Advantage: Convenience
  - Can share data structures
  - Just annotate loops
  - Closer to serial code
- Disadvantages
  - No locality control
  - Does not scale
  - Race conditions

Message Passing

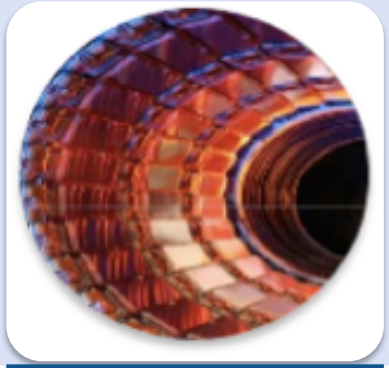
- Advantage: Scalability
  - Locality control
  - Communication is all explicit in code (cost transparency)
- Disadvantage
  - Need to rethink data structures
  - Tedious pack/unpack code
  - When to say “receive”

PGAS Languages

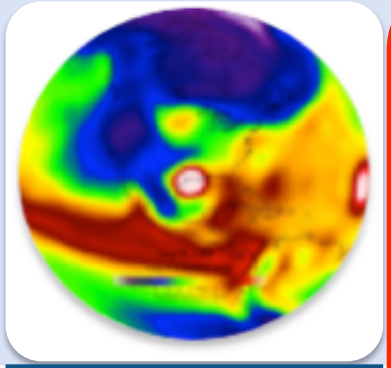
- ***Global address space:*** thread may directly read/write remote data
  - Hides the distinction between shared/distributed memory
- ***Partitioned:*** data is designated as local or global
  - Does not hide this: critical for locality and scaling



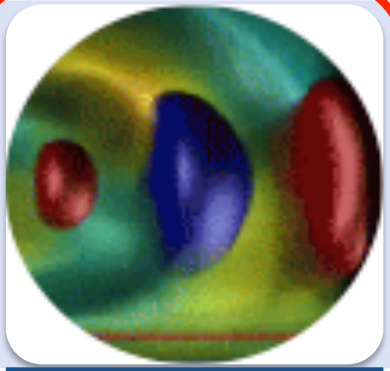
Science Across the "Irregularity" Spectrum



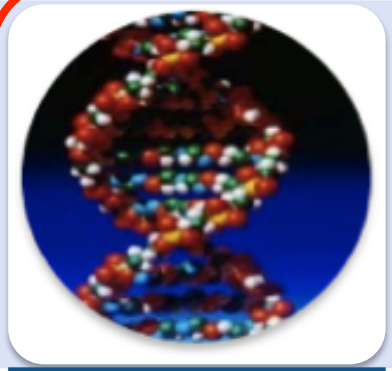
Massive Independent Jobs for Analysis and Simulations



Nearest Neighbor Simulations



All-to-All Simulations



Random access, large data Analysis

Data analysis and simulation

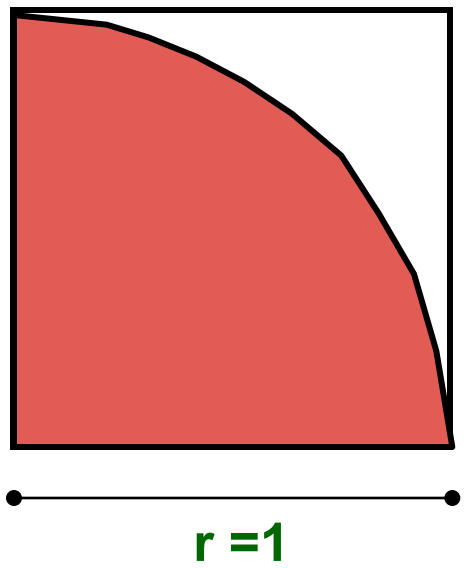
Hello World in UPC

- 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:

```
#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
  - π = 4*ratio



Pi in UPC

- Independent estimates of pi:

```
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);
}
```

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls "hit" separately

Helper Code for Pi in UPC

- Required includes:

```
#include <stdio.h>
#include <math.h>
#include <upc.h>
```

- Function to throw dart and calculate where it hits:

```
int hit(){
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) / RAND_MAX;
  double y = ((double) rand()) / RAND_MAX;
  if ((x*x + y*y) <= 1.0) {
      return(1);
  } else {
      return(0);
  }
}
```

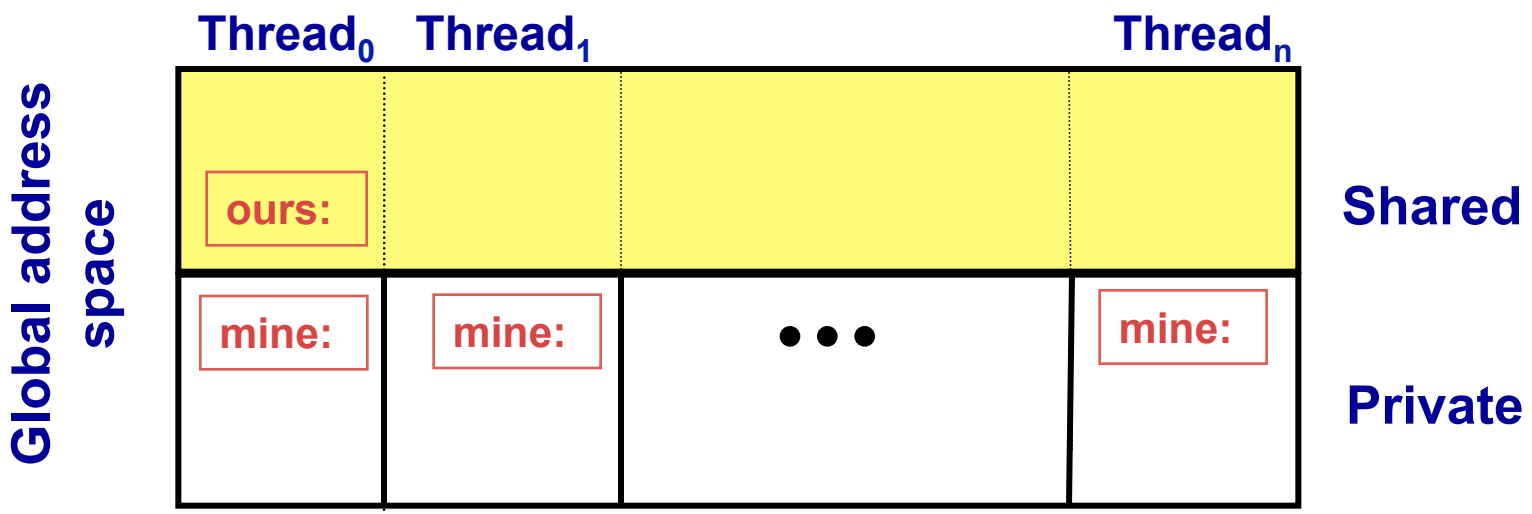
Shared vs. Private Variables

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each 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: may not occur in a function definition, except as static. Why?



Pi in UPC: Shared Memory Style

- **Parallel computing of pi, but with a bug**

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

shared variable to record hits

divide work up evenly

accumulate hits

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

    do computation unrelated to 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;
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    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++)
        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);
}
```

one shared scalar variable

other private variables

create a lock

accumulate hits locally

accumulate across threads

Pi in UPC: Data Parallel Style with 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

```
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
    ...
    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);
}
```

Berkeley collectives

no shared variables

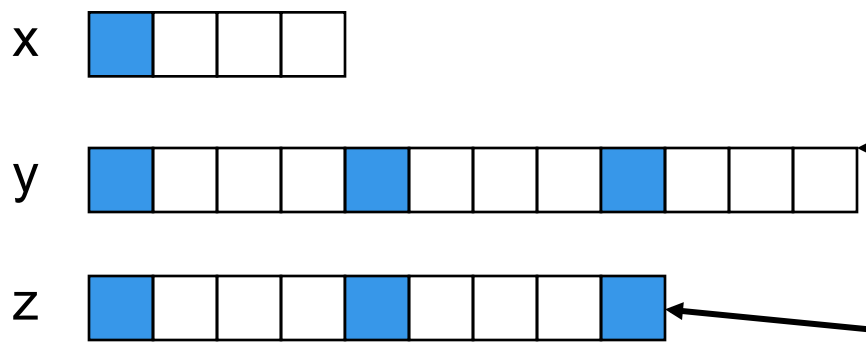
barrier implied by collective

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



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

As a 2D array, y is logically blocked by columns

z is not

Pi in UPC: Shared Array Version

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

```
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);
  }
}
```

all_hits is shared by all processors, just as hits was

update element with local affinity

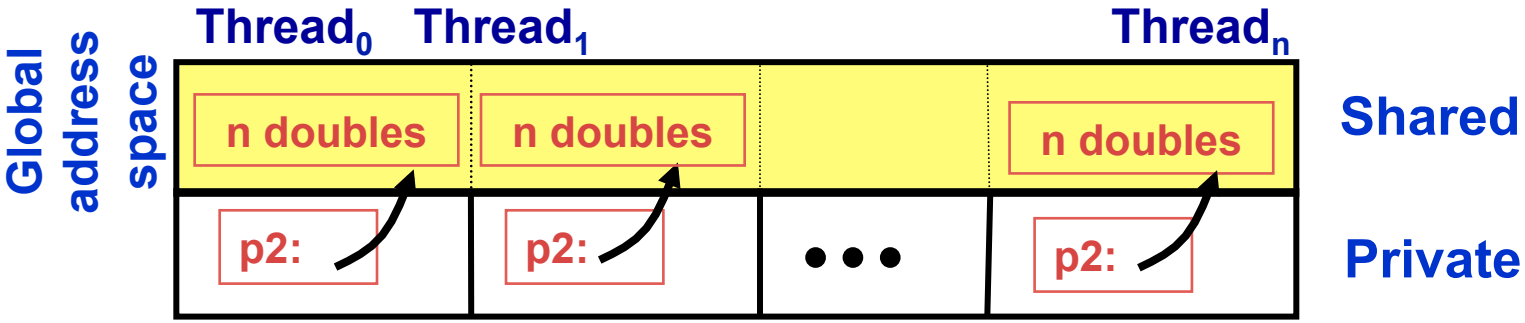
Global Memory Allocation

```
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);
```



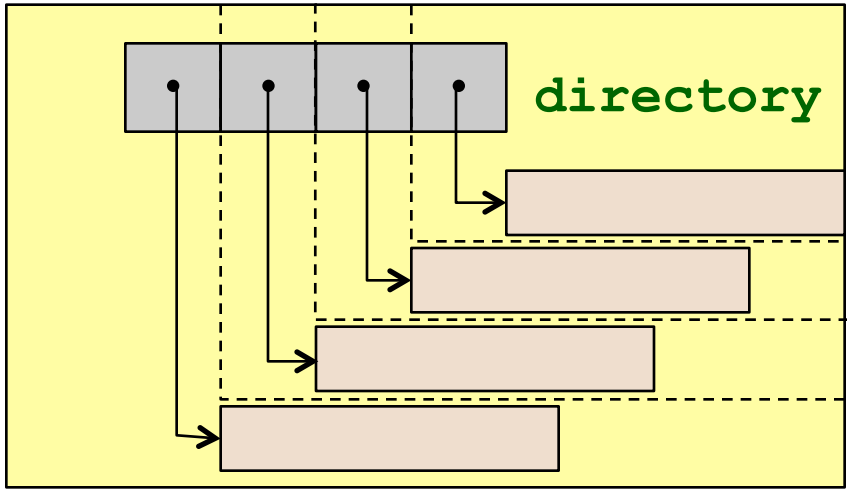

```
void upc_free(shared void *ptr);
```

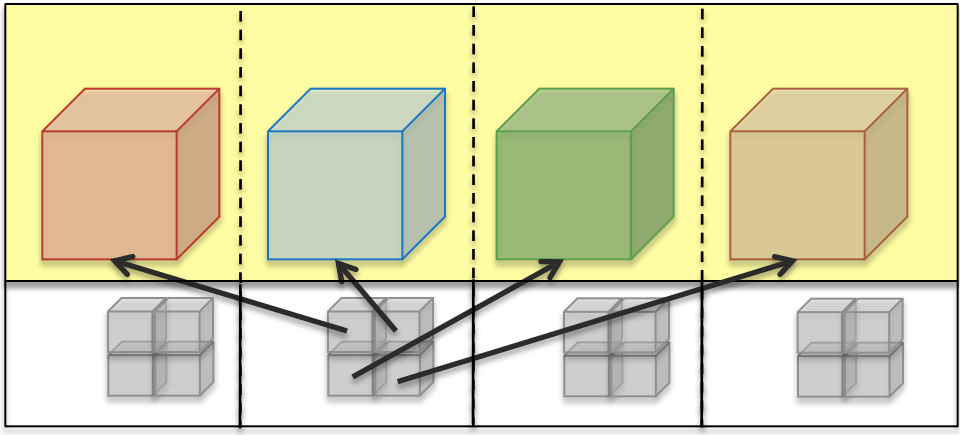
- Non-collective function; frees the dynamically allocated shared memory pointed to by ptr

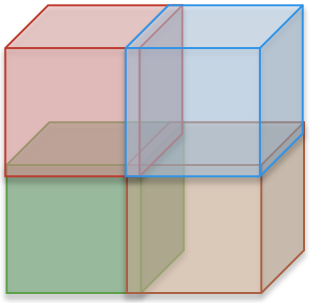
Distributed Arrays Directory Style

- 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));
```



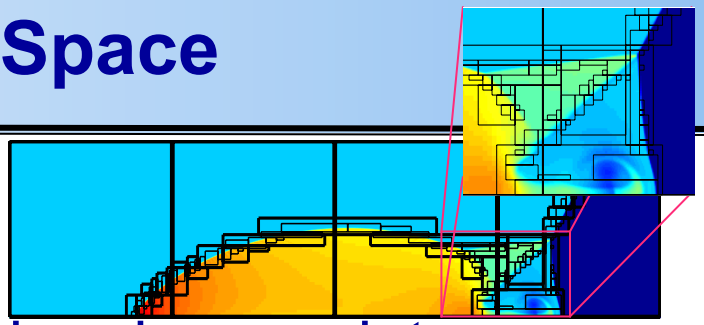




physical and conceptual 3D array layout

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

Arrays in a Global Address Space

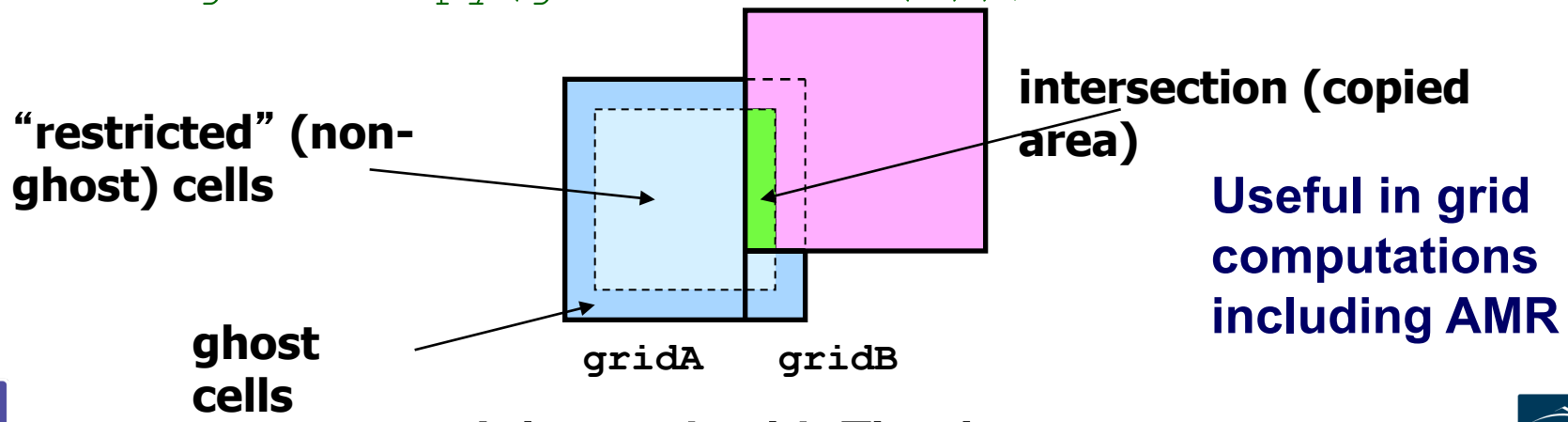


- Key features of Titanium arrays
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to identify ghosts and iterate:

```
foreach (p in gridA.shrink(1).domain()) ...
```

- Array copies automatically work on intersection

```
gridB.copy(gridA.shrink(1));
```

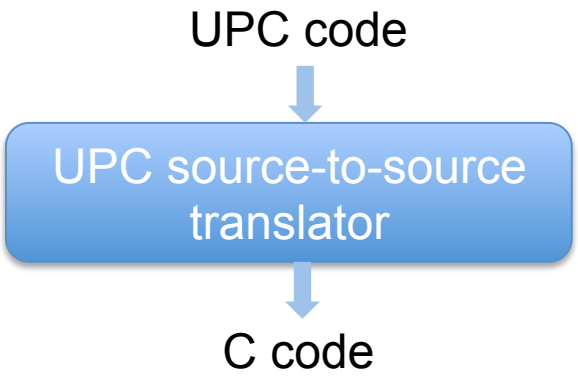


Useful in grid computations including AMR

Joint work with Titanium group

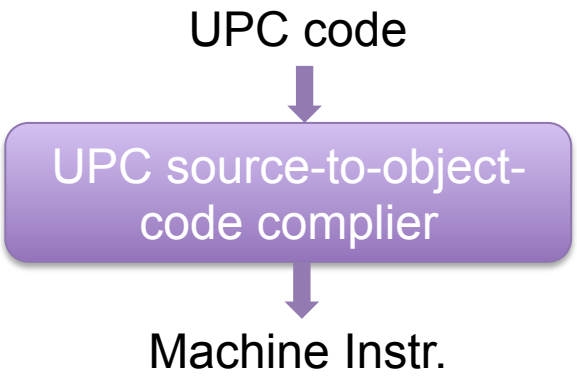
UPC Compiler Implementation

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

New in UPC 1.3 Non-blocking Bulk Operations

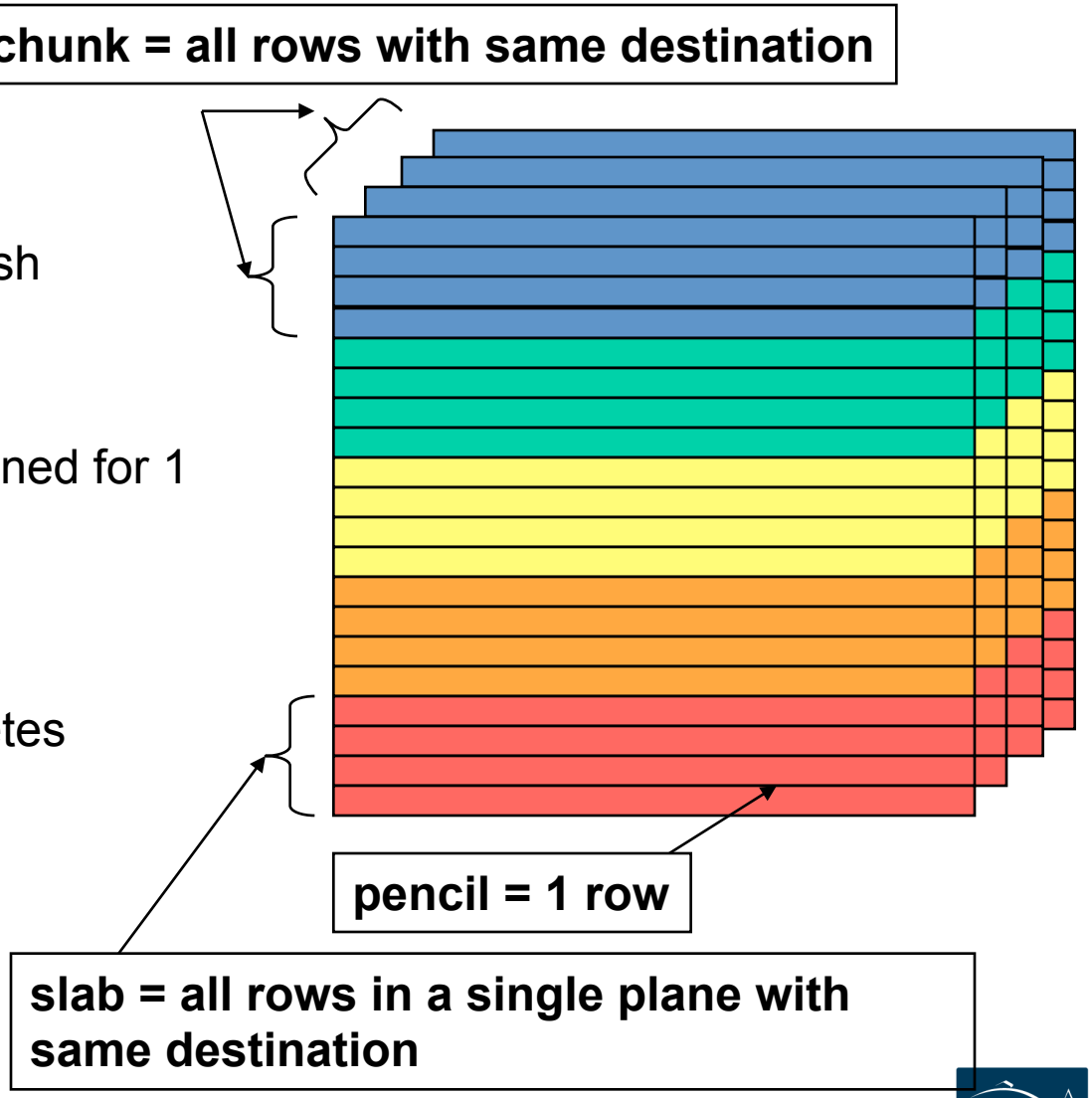
Important for performance:

- **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
```

Communication Strategies for 3D FFT

- 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

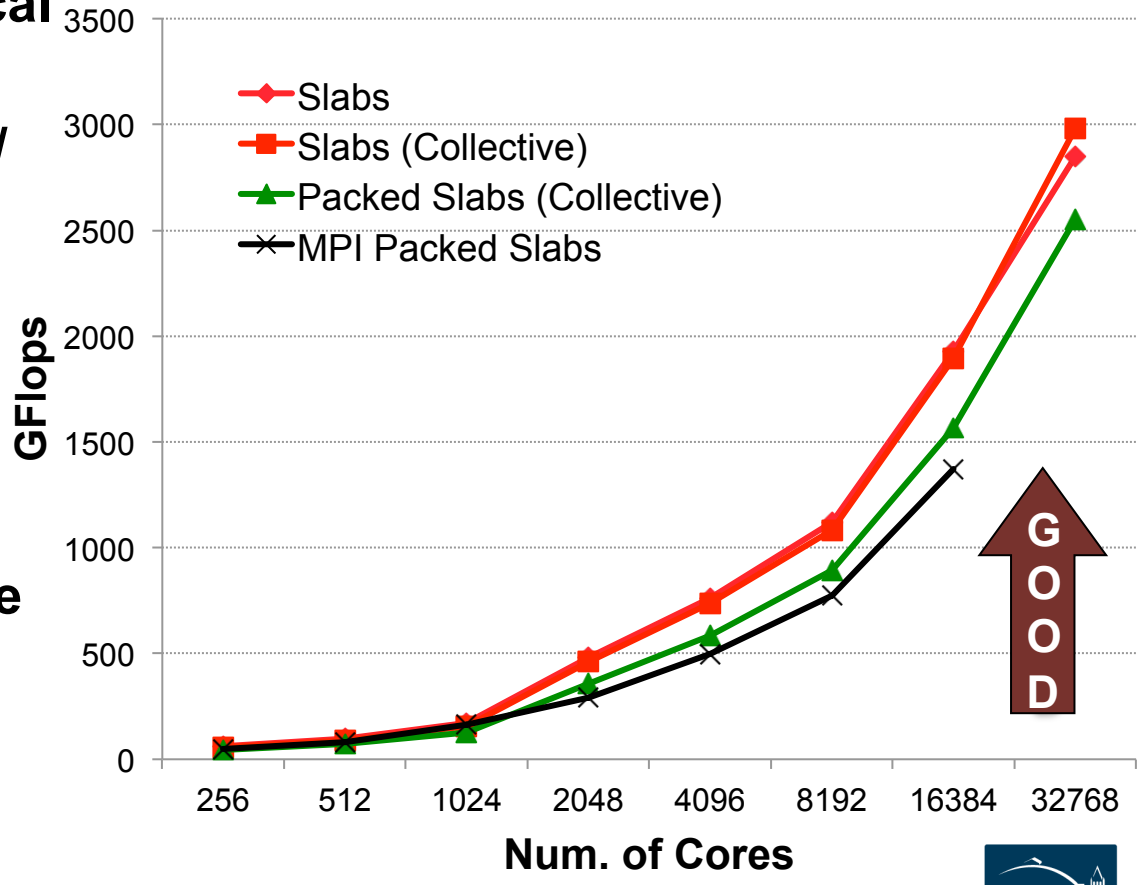


Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea

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**

HPC Challenge Peak as of July 09 is ~4.5 Tflops on 128k Cores



UPC 1.3 Atomic Operations

- More efficient than using locks when applicable

```
upc_lock();
update();
upc_unlock();
```

vs

```
atomic_update();
```

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

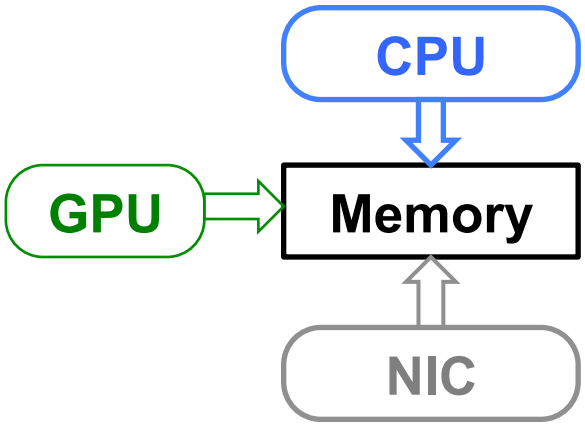
Only support limited operations on a subset of data types. e.g.,

Atomic_CAS on uint64_t

Memory

Atomic_Add on double

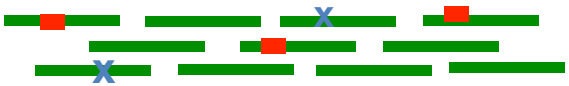
Atomic ops from different processors *may not* be atomic to each other



UPC + Remote Invocation for Scalable Meraculous Application used in Genomics Grand Challenge

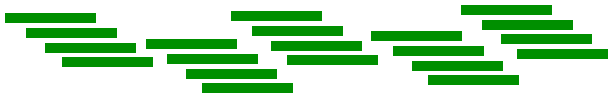
Meraculous Assembly Pipeline

reads



New fast I/O using SeqDB over HDF5

k-mers



New analysis filters errors using probabilistic "Bloom Filter"

contigs

Graph algorithm scales to 15K cores on NERSC's Edison using DEGAS language rather than shared memory hardware

Human: 44 hours to 20 secs
Wheat: "doesn't run" to 32 secs

Future work: Scaffolds using Scalable Alignment

Meraculous assembler is use in production at the Joint Genome Institute

- **Wheat assembly is a "grand challenge"**
- **Hardest part is contig generation (large in-memory hash table)**

DEGAS X-Stack project

- **Gives tera- to petabtye "shared" memory**
- **Combines with new math/data algorithm for mapping to anchor 92% of wheat chromosome**

DEGAS

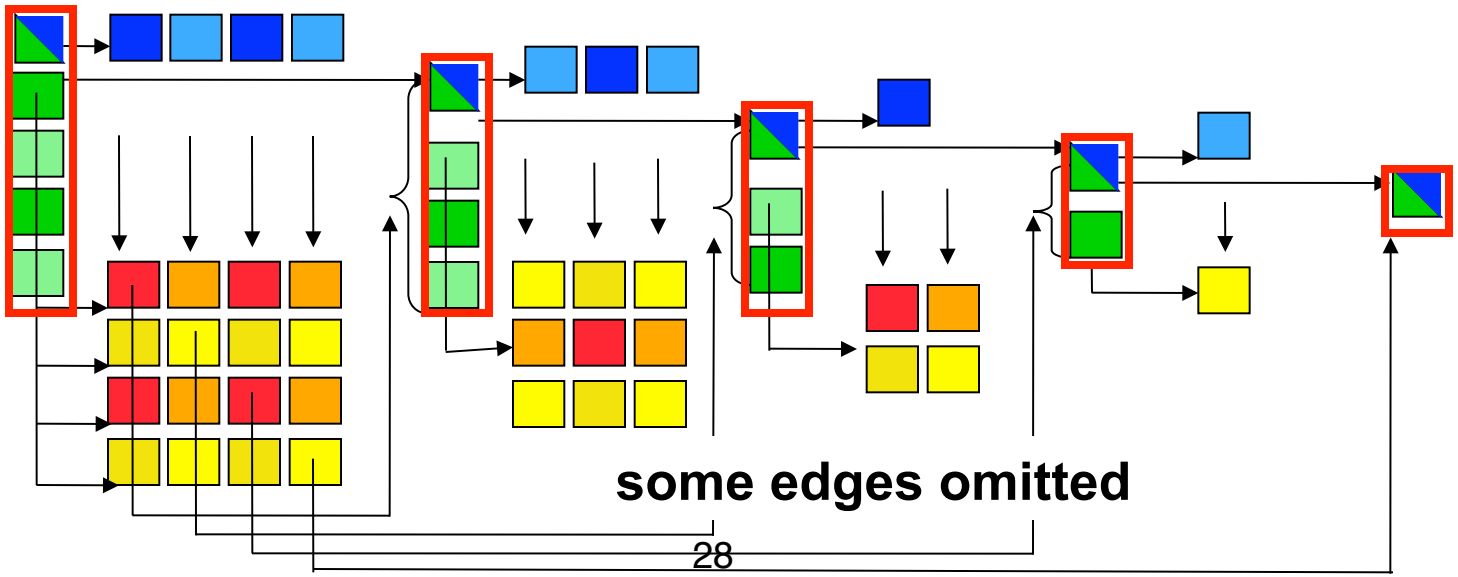
Dynamic Exascale Global Address Space project, joint work JGI, Early Career and Mantissa

BERKELEY LAB

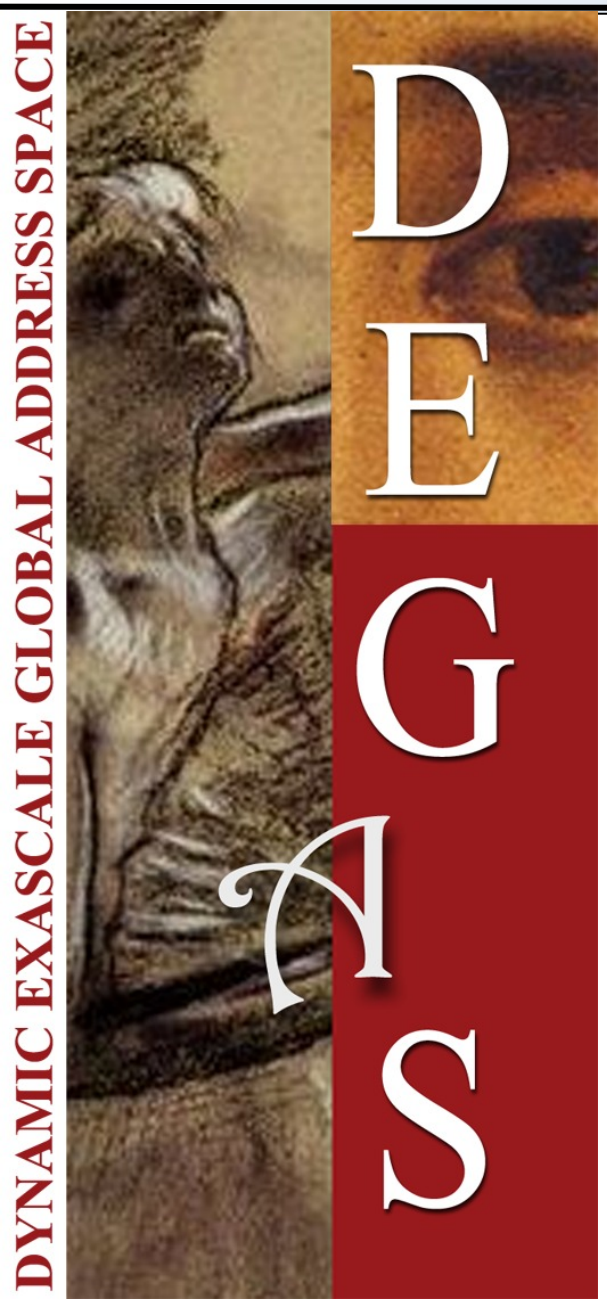
Beyond Put/Get: Event-Driven Execution

- DAG Scheduling in a distributed (partitioned) memory context
- 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

Uses a Berkeley extension to UPC to remotely synchronize



DEGAS Programming System: UPC++



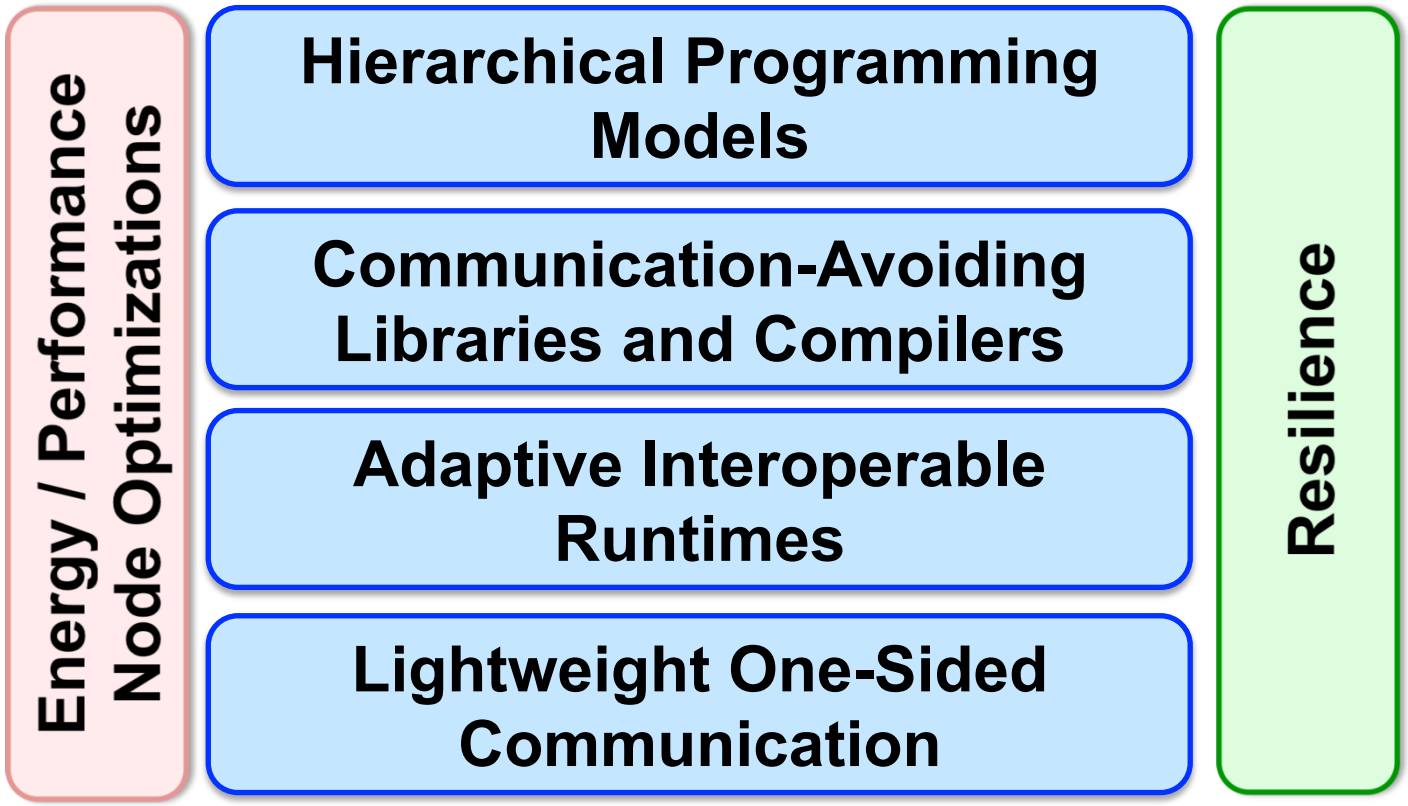
UPC++

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.

BERKELEY LAB
Lawrence Berkeley National Laboratory

DEGAS: Dynamic Exascale Global Address Space



Communication-avoiding algorithms generalized to compilers, and communication optimizations in PGAS

Making PGAS more Dynamic; DAG Programming more Locality-Aware

PGAS

- Asynchronous remote put/get for random access
- Good locality control and scaling

E.g. *p = … or … = a[i];

DAGs

- Asynchronous invocation
- Good for dynamic load balancing and event-driven execution

finish { … async f (x)…}

DEGAS

Hierarchical locality control

(1) Remote put/get and atomics

(2) Remote invocation

(3) Distributed load balance

UPC++ Generic Programming for PGAS

- Enable "modern" language features with PGAS
  - Interoperable with MPI, OpenMP, CUDA,…

- UPC++ uses templates to express shared data

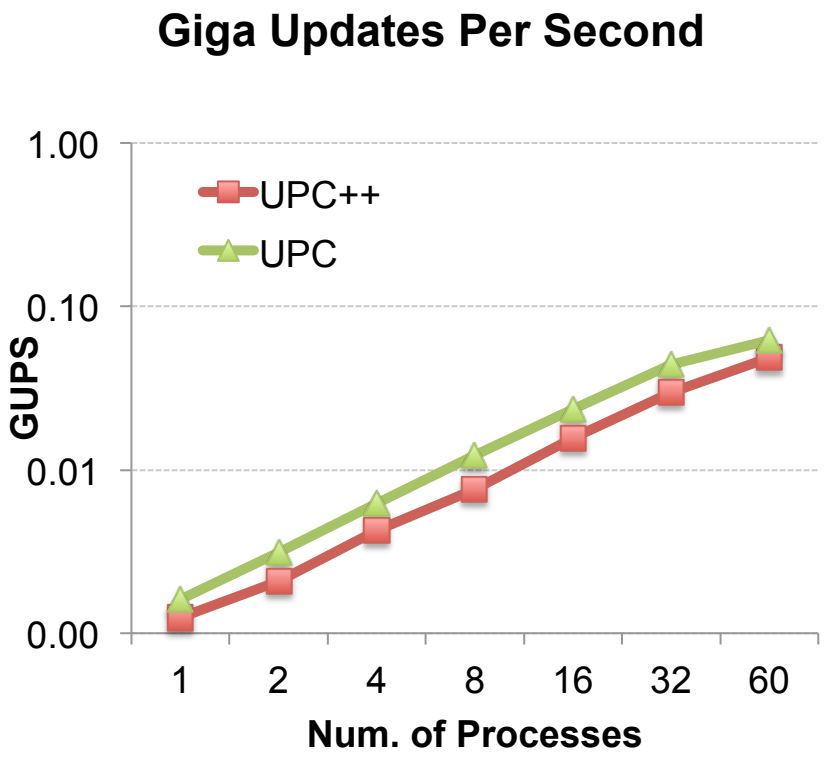
```
shared_var<int> s;  // shared int s in UPC
shared_array<int> sa(8); // shared int sa[8]
                         // in UPC
```

- UPC++ provides remote invocation

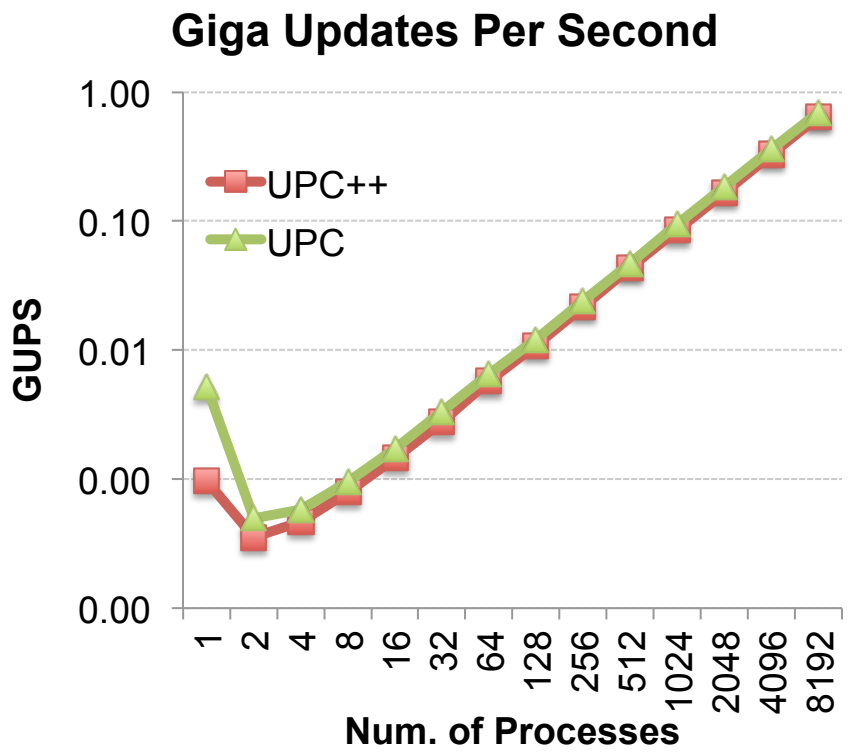
```
// Remote Procedure Call
upcxx::async(place)(Function f, T1 arg1, T2 arg2,…);
upcxx::wait();

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

GUPS Performance on MIC and BlueGene/Q

MIC

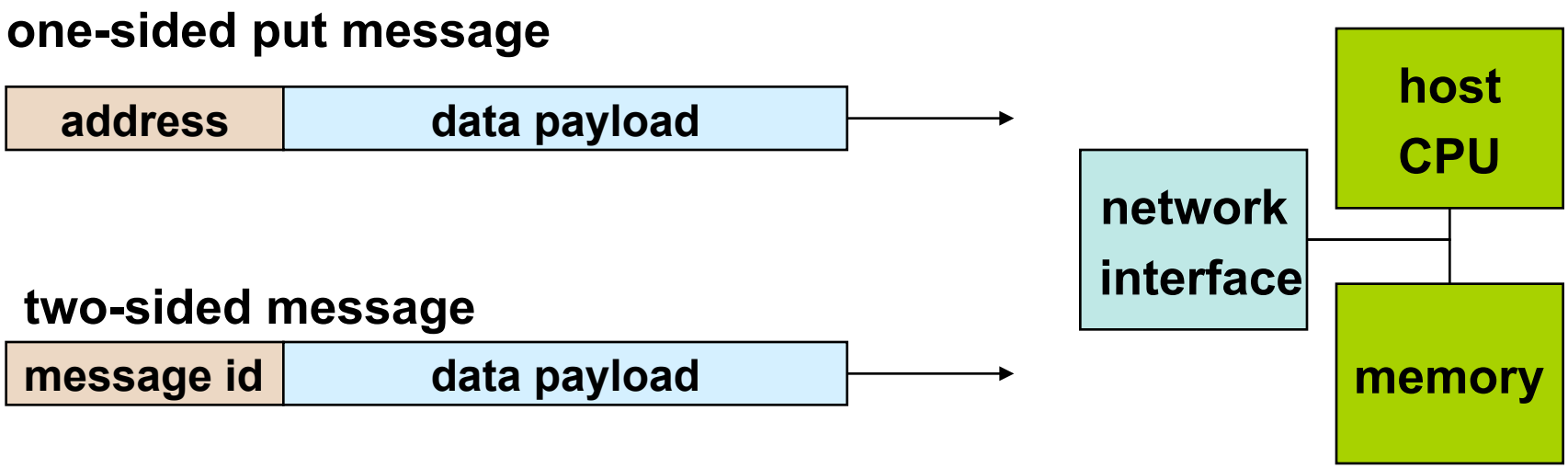


BlueGene/Q



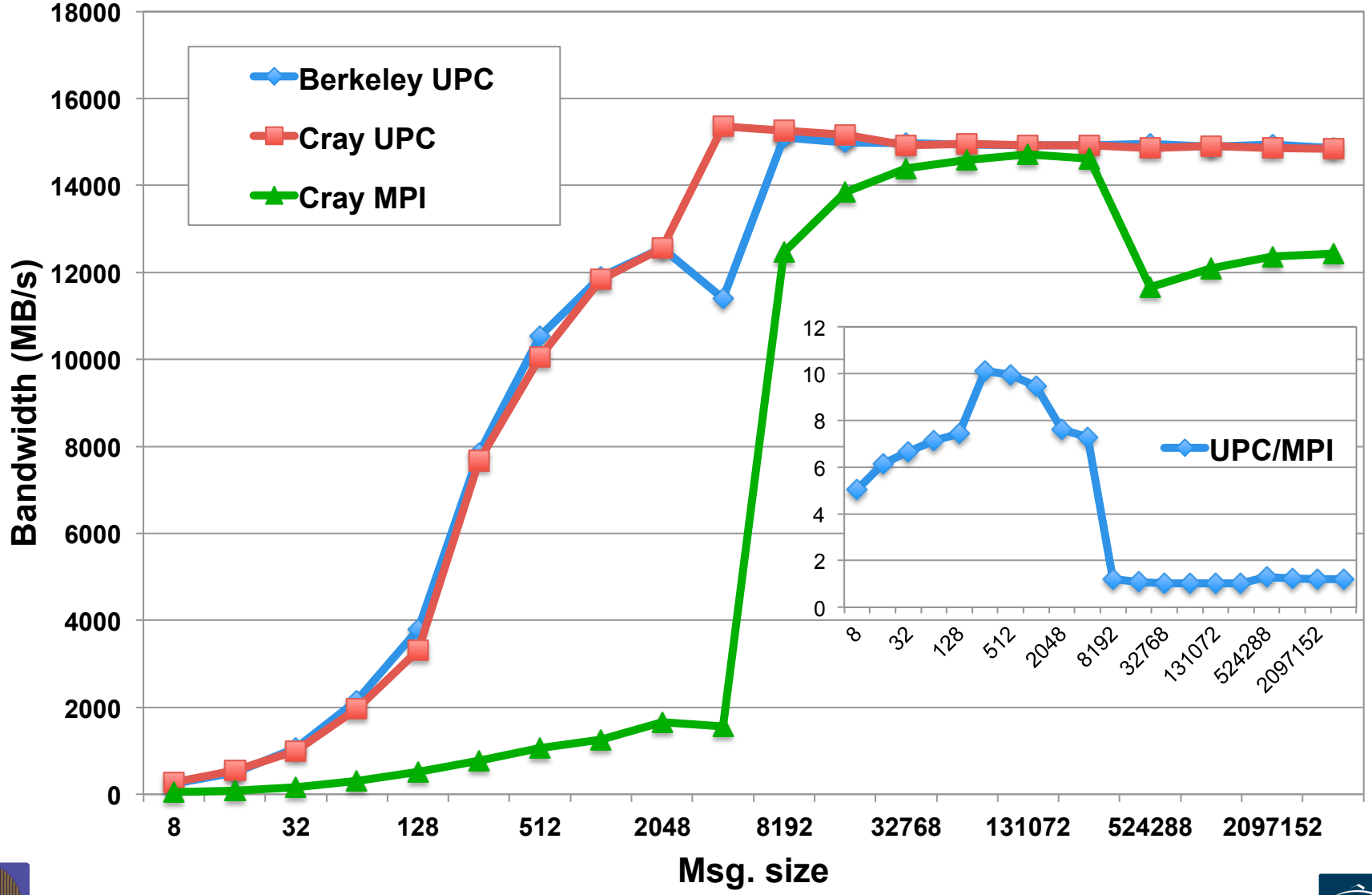
Difference between UPC++ and UPC is about 0.2 μs (~220 cycles)

One-Sided vs Two-Sided

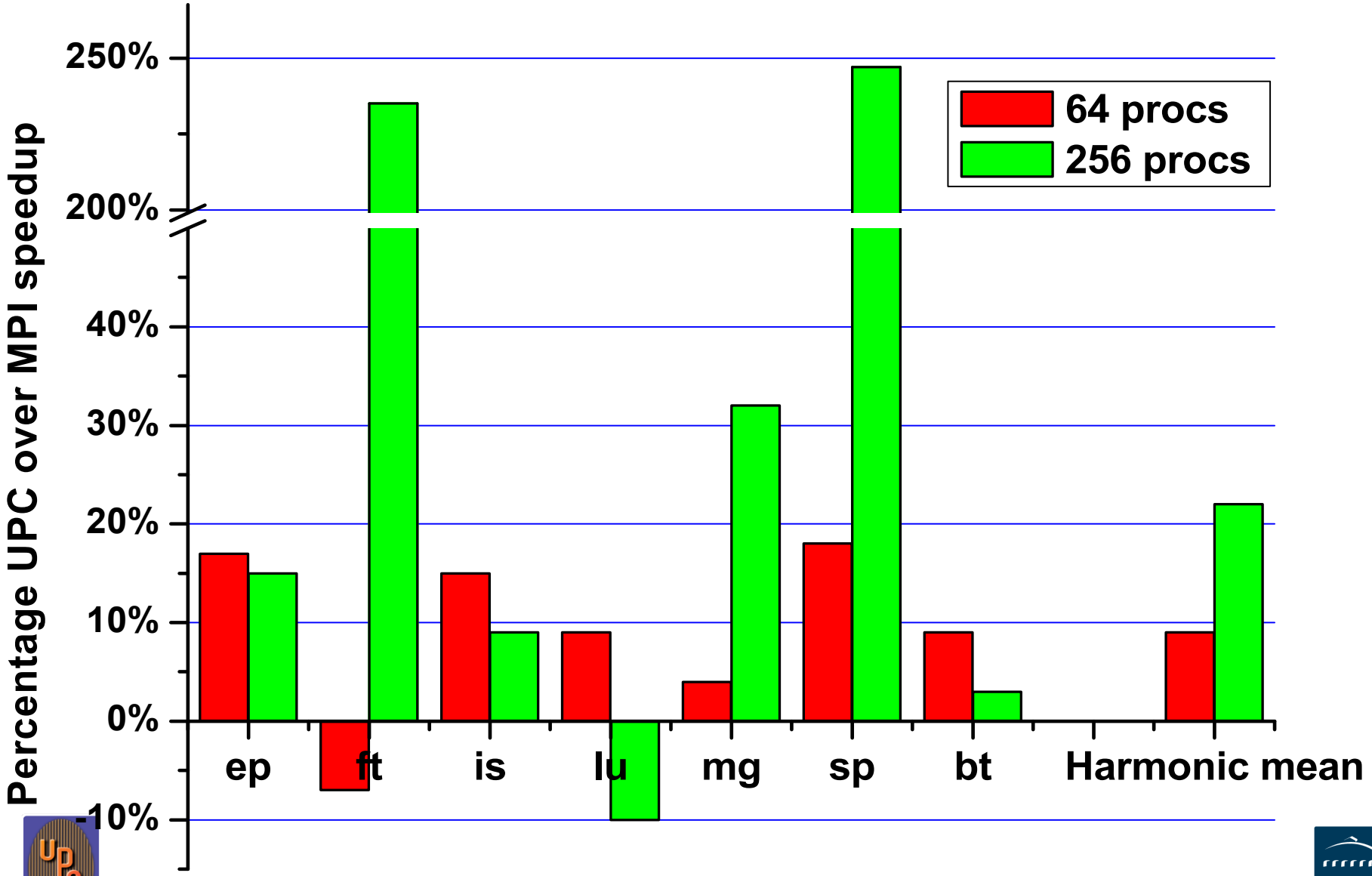


- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks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

Bandwidths on Cray XE6 (Hopper)



Cray XE6 Application Performance



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:

  http://www.gccupc.org/
- Berkeley compiler

  http://upc.lbl.gov
- Language specification and other documents

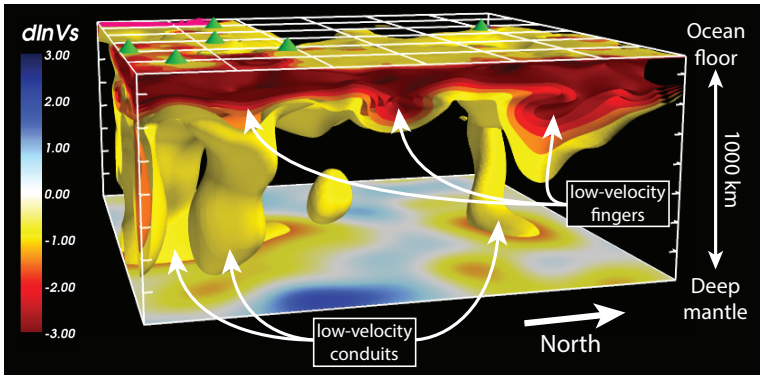
  https://code.google.com/p/upc-specification

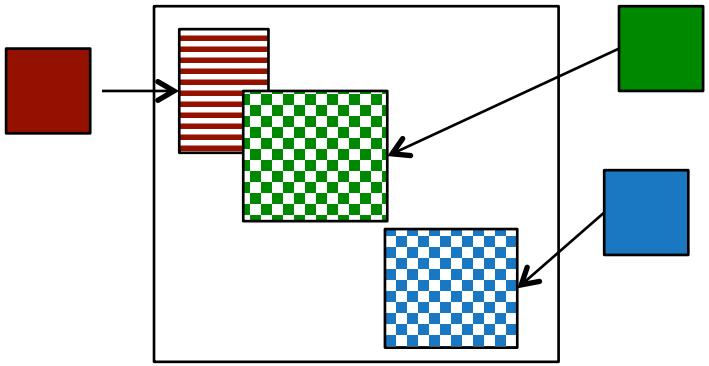
  https://upc-lang.org
- Vendor compilers: Cray, IBM, HP, SGI,…

UPC++ Asynchronous Remote Execution Enables Scalable Data Fusion

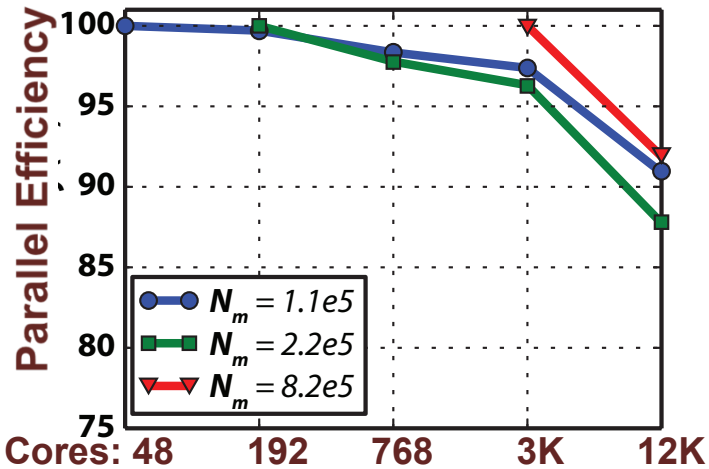
PGAS before X-Stack
- Asynchronous remote put/get
- Good locality control and scaling
  E.g. *p = … or … = a[i];ç

New: Asynchronous invocation
- Event-driven execution & load balancing
- Hierarchical synchronization and places
  finish { … async f (x)…}





- Seismic modeling for energy applications "fuses" observational data into simulation.
- PGAS illusion of scalable shared memory to construct matrix and measure data "fit"
- New UPC++ dialect supports PGAS libraries; future distributed data structure library

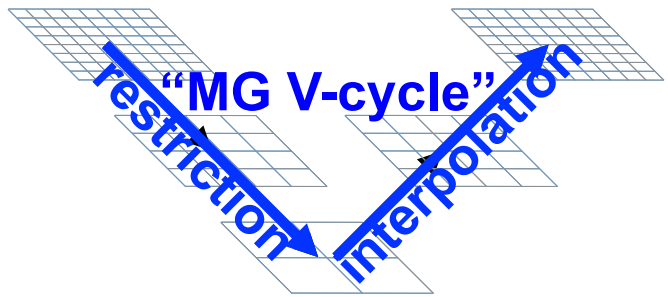


Dynamic Exascale Global Address Space from LBNL, Rice, UTAustin, UCB, LLNL

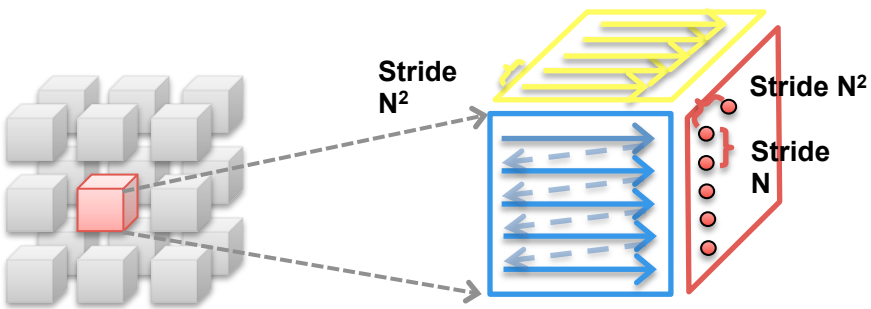
Mini-GMG in UPC++ uses high level array library for Productivity and Performance

Before X-Stack

- MPI's explicit communication inhibits productivity and performance portability



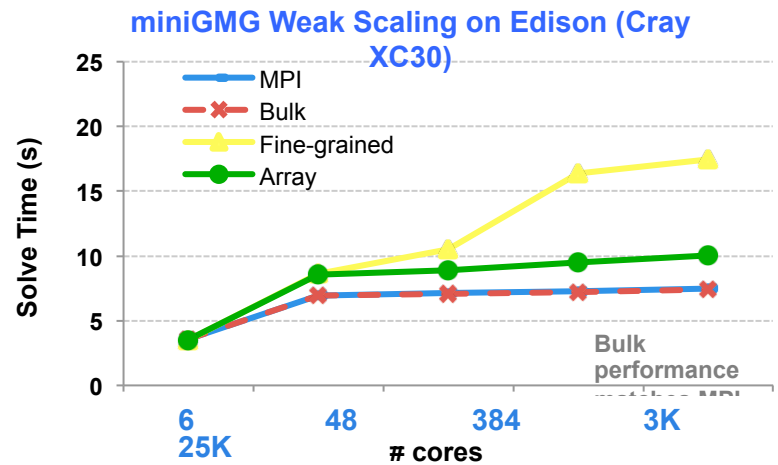
Used miniGMG benchmark which proxies MG solver in combustion codesign center



Each process exchanges data with 26 neighbors. UPC++ multidimensional arrays give an easy interface to users and optimize strided data accesses automatically.

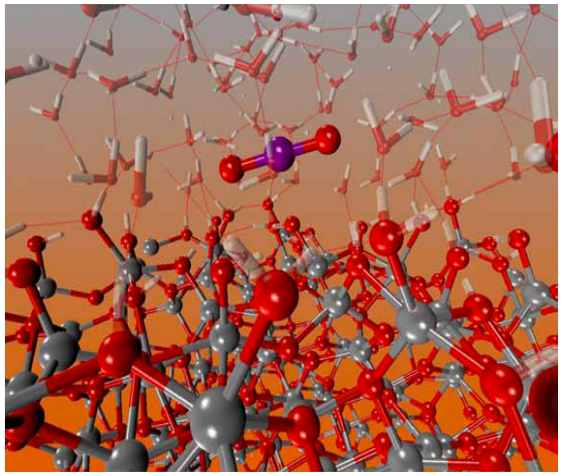
New: UPC++ Multidimensional Arrays

- Provides productivity via high-level array abstraction
- Encapsulates performance critical



- "Fine-grained" like OpenMP -- insufficient locality control
- "Bulk" like MPI with 1-sided communication; perfect match to scalability but no productivity advantage
- "Array" version uses multi-dimensional array constructs for productivity and ~MPI performance
- Future runtime optimizations should close Array/ Bulk gap

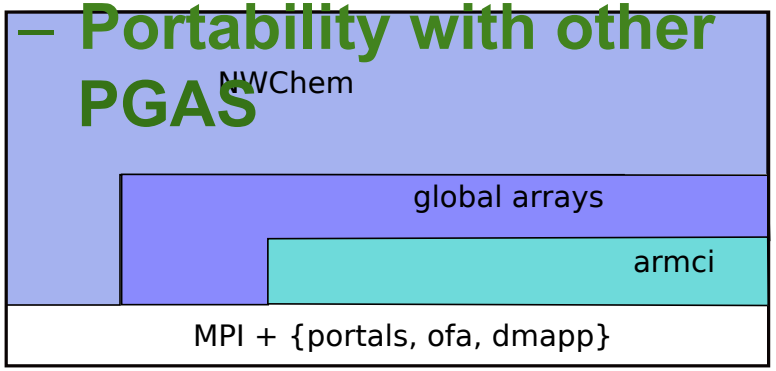
GASNet Asynchronous One-Sided Communication Aids in Performance Portability and Scaling for NWChem

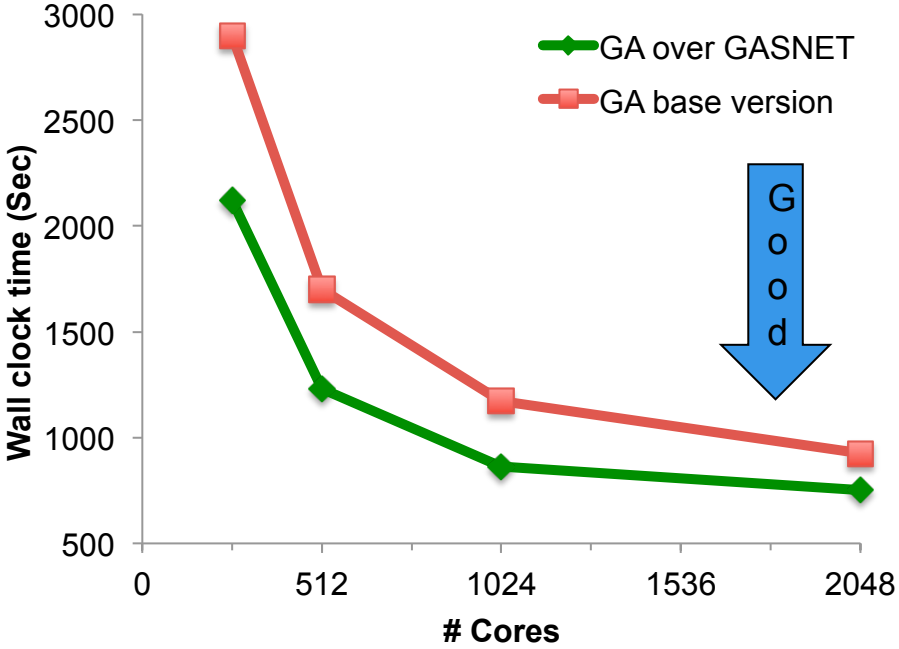


credit:nwchem-sw.org

- **Production chemistry code**
  - 60K downloads world wide
  - 200-250 scientific application publications per year
  - Over 6M LoC, 25K files

- **New version on GASNet for**
  - Improved performance
  - Portability with other PGAS



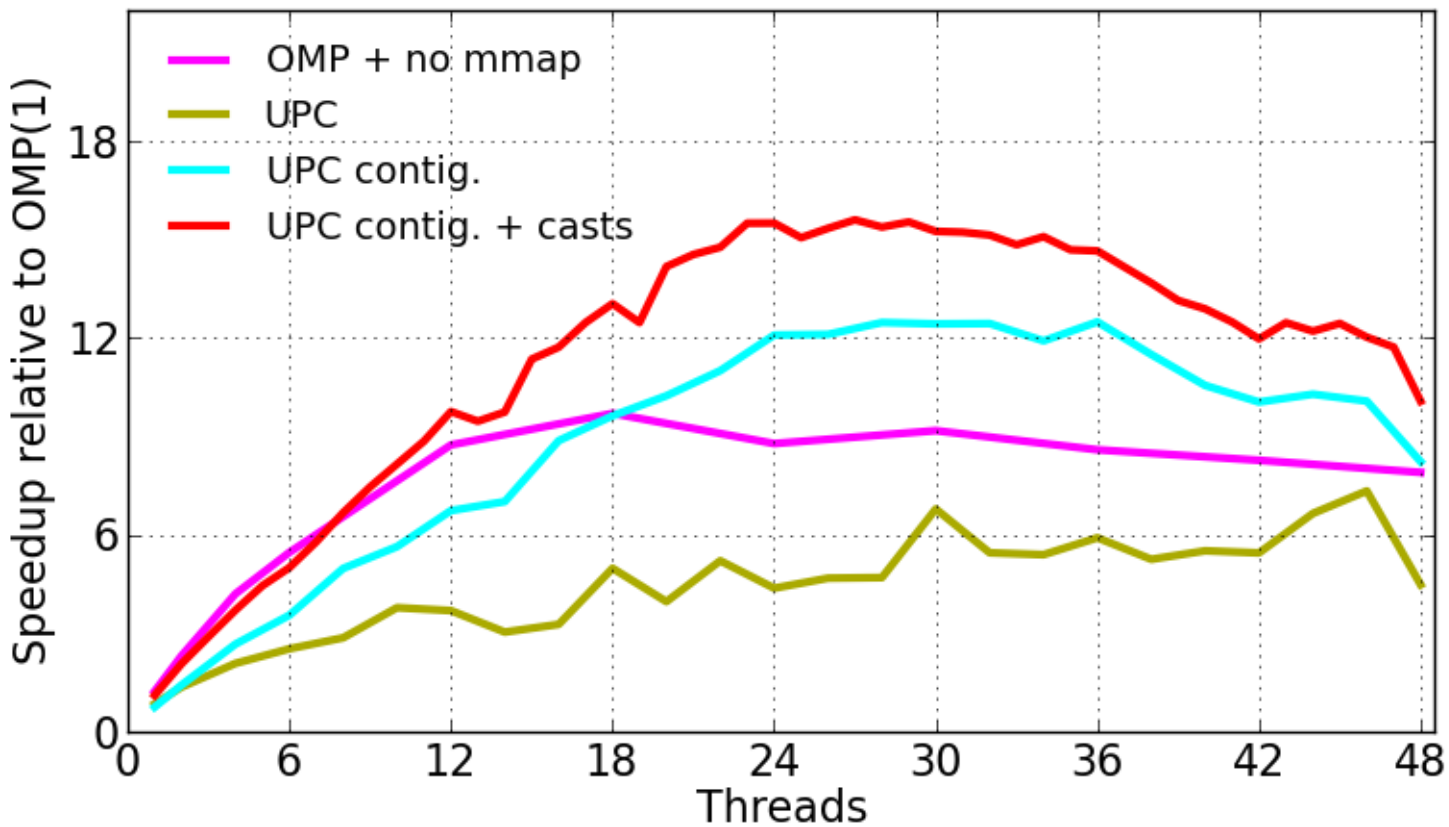


LULESH - https://codesign.llnl.gov/lulesh.php

Optimizations:

- Blocked vs. cyclic (default) array layout
- Use private pointer to the thread block in shared array

ZE)



Task Library API (under development)

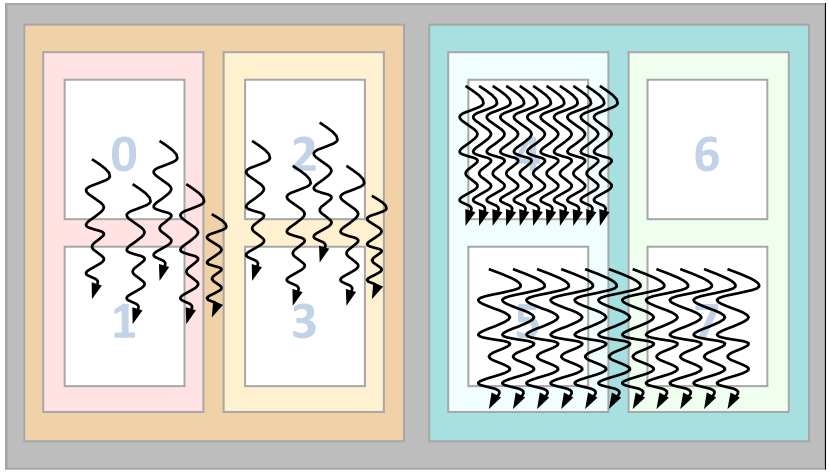
Dynamic load balancing in UPC (and UPC++) is an option

```
taskq_put(…);
taskq_execute(…);
int taskq_all_isEmpty(taskq_t *taskq);
Etc.
```

Can be used optionally within a node, across nodes, on a certain subproblem, etc.

Hierarchical Work Stealing on Manycore Clusters
Min, Iancu,Yelick. PGAS 2011

Hierarchical machines and Applictions



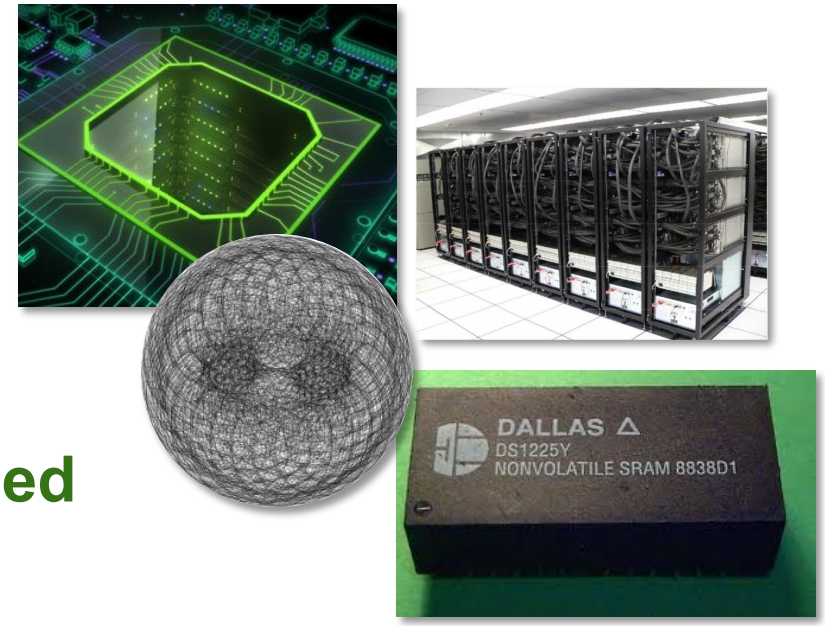
- **Hierarchical memory model may be necessary (what to expose vs hide)**
- **Two approaches to supporting the hierarchical control**

- Option 1: Dynamic parallelism creation (e.g., Chapel)
  - 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);
```



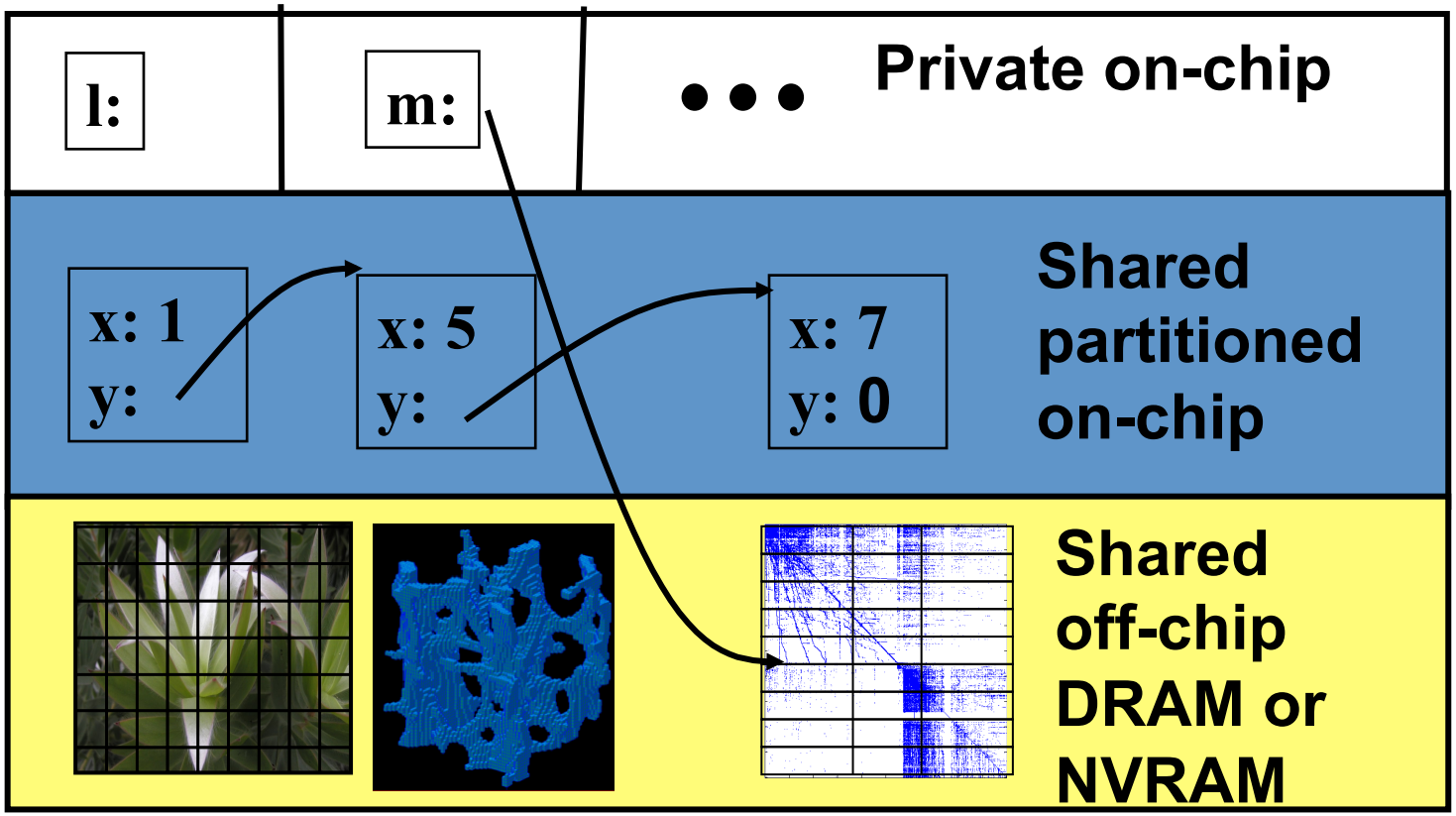
It is implemented using one-sided communication: put/get

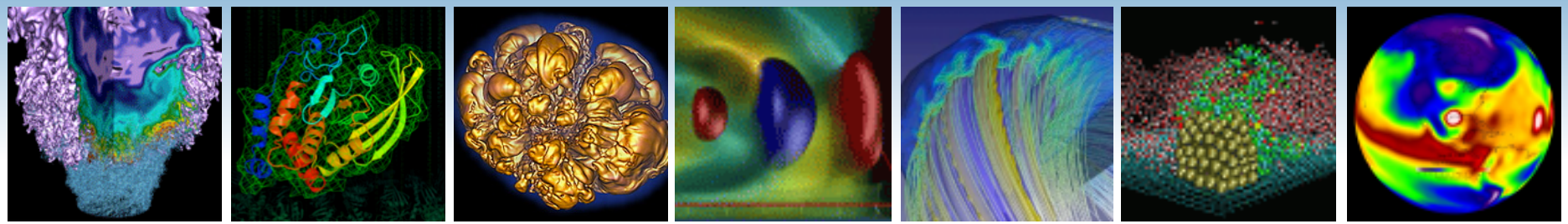
Support for one-sided communication (DMA) appears in:

- 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

Vertical PGAS

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





LBNL / UCB Collaborators

- Dan Bonachea
- Paul Hargrove
- Amir Kamil
- Khaled Ibrahim
- Costin Iancu
- Yili Zheng
- Michael Driscoll
- Evangelos Georganas
- Penporn Koanantakool
- Steven Hofmeyr
- Leonid Oliker
- Eric Roman
- John Shalf
- Erich Strohmaier
- Samuel Williams
- Cy Chan
- Didem Unat
- James Demmel, UCB
- Scott French
- Edgar Solomonik
- Eric Hoffman
- Wibe de Jong

External collaborators (& their teams!)

- Vivek Sarkar, Rice
- John Mellor-Crummey, Rice
- Krste Asanoviç UCB
- Mattan Erez, UT Austin
- Dan Quinlan, LLNL

Thanks!