UPC and UPC++:
Partitioned Global Address Space Languages

Kathy Yelick
Associate Laboratory Director for Computing Sciences
Lawrence Berkeley National Laboratory
Professor of EEECS, UC Berkeley

BERKELEY LAB

U.S. DEPARTMENT OF ENERGY | Office of Science

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

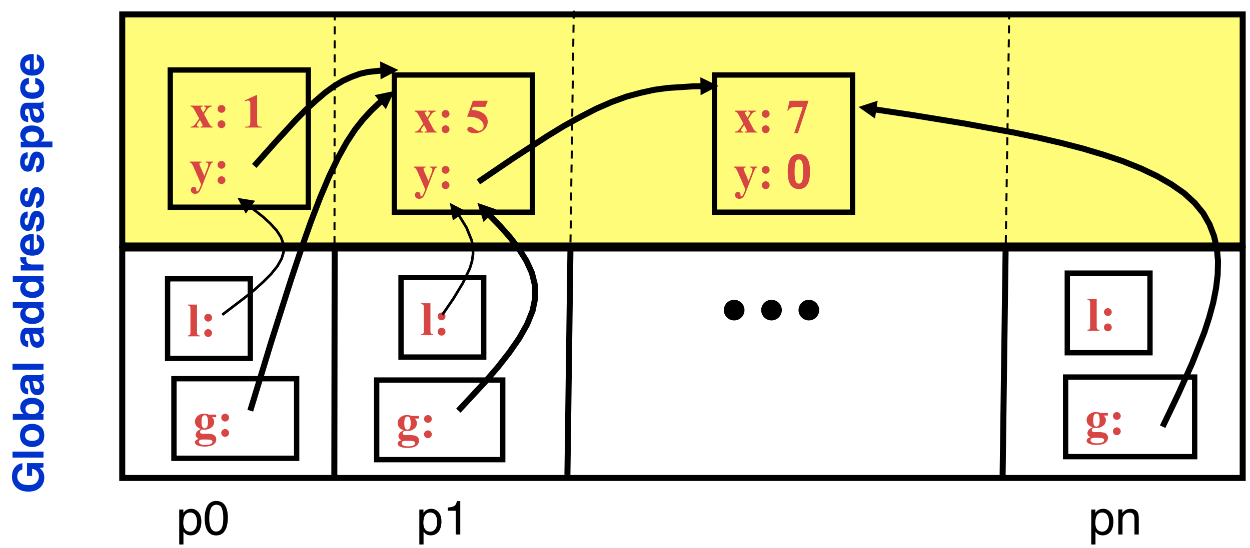
A's	B's	C's	…	Y's	Z's

- Distributed memory: the array is huge and spread out
  - Each processor has a substream and sends +1 to the appropriate processor… and that processor "receives"

A's	B's	C's	D's	…	Y's	Z's	

PGAS = Partitioned Global Address Space

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



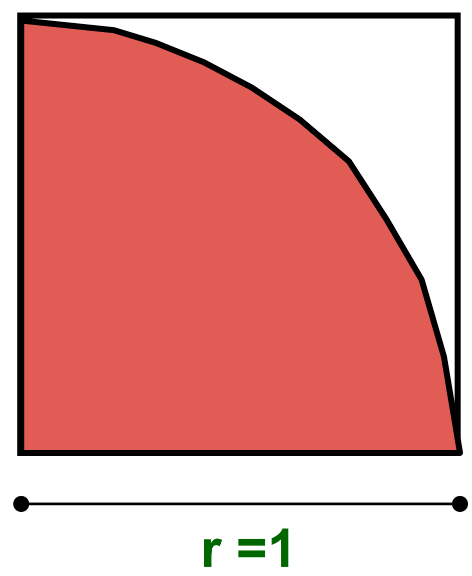
Hello World in UPC

- 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
  - $\pi = 4$*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);
  }
}
```

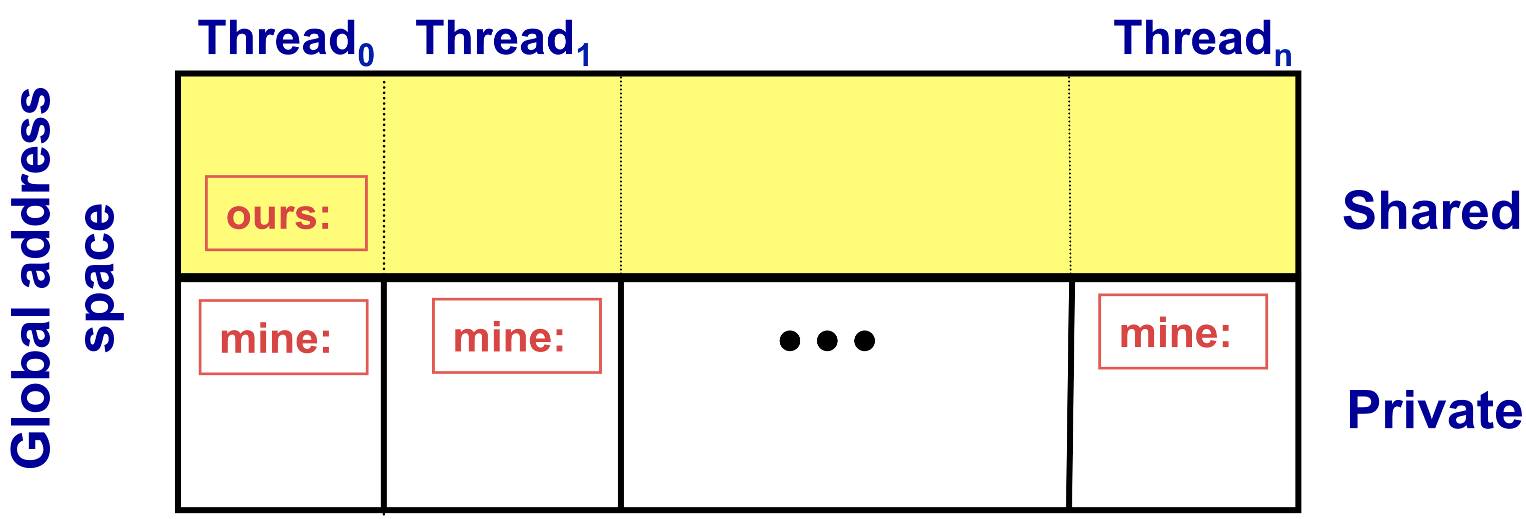
Shared vs. Private Variables

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, 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;
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 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

```
#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

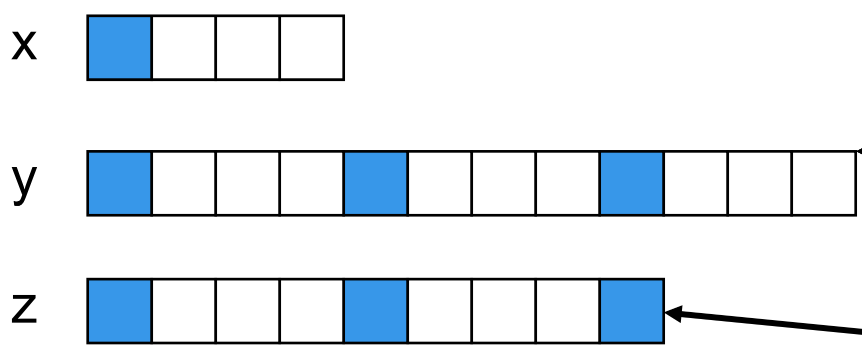
Shared Arrays Are Cyclic By Default

- 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

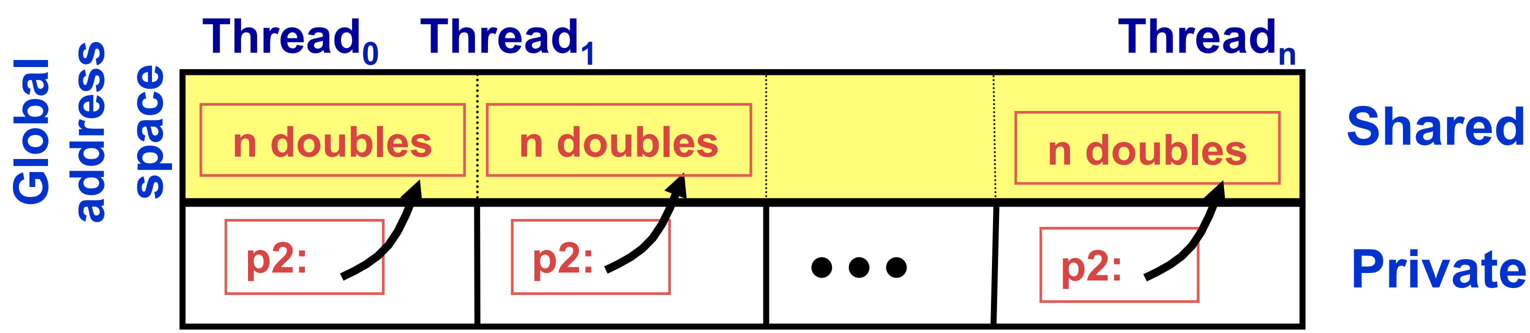
Global Memory Allocation

```
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);
```



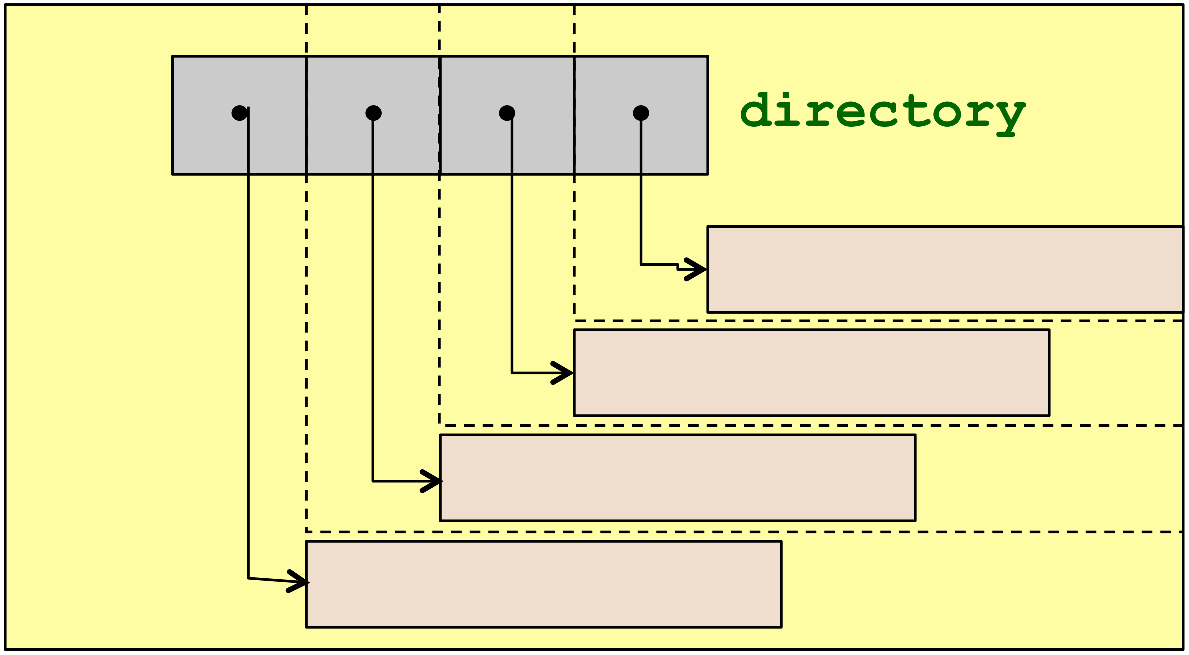

```
void upc_free(shared void *ptr);
```

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

Distributed Arrays Directory Style

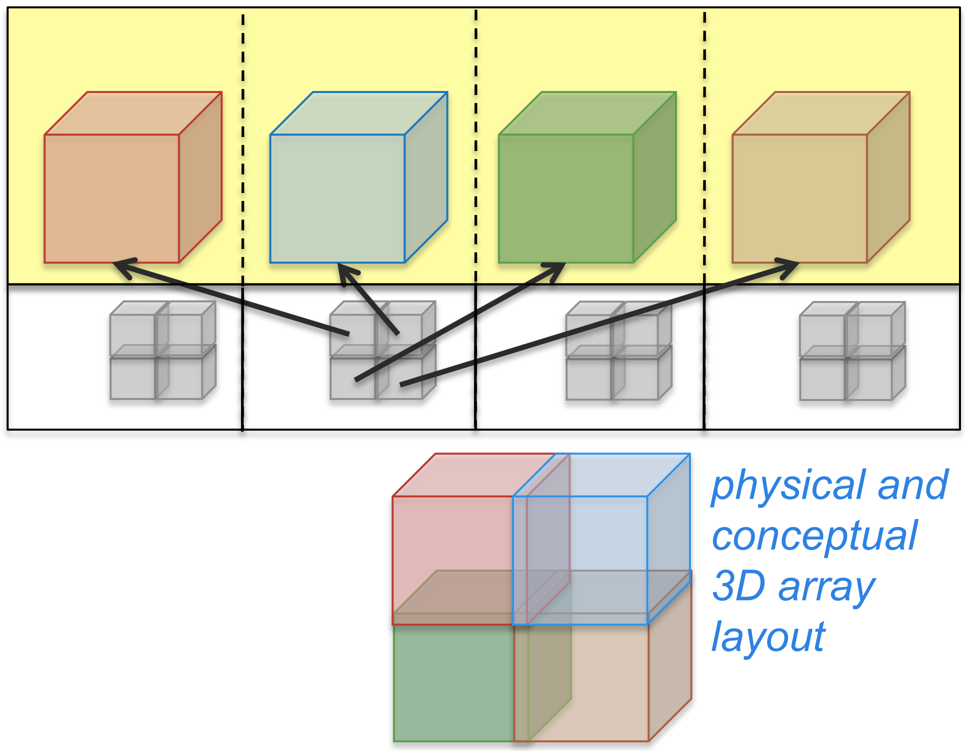
- Many UPC programs avoid the UPC style arrays in factor of directories of objects

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



Distributed Arrays Directory Style

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



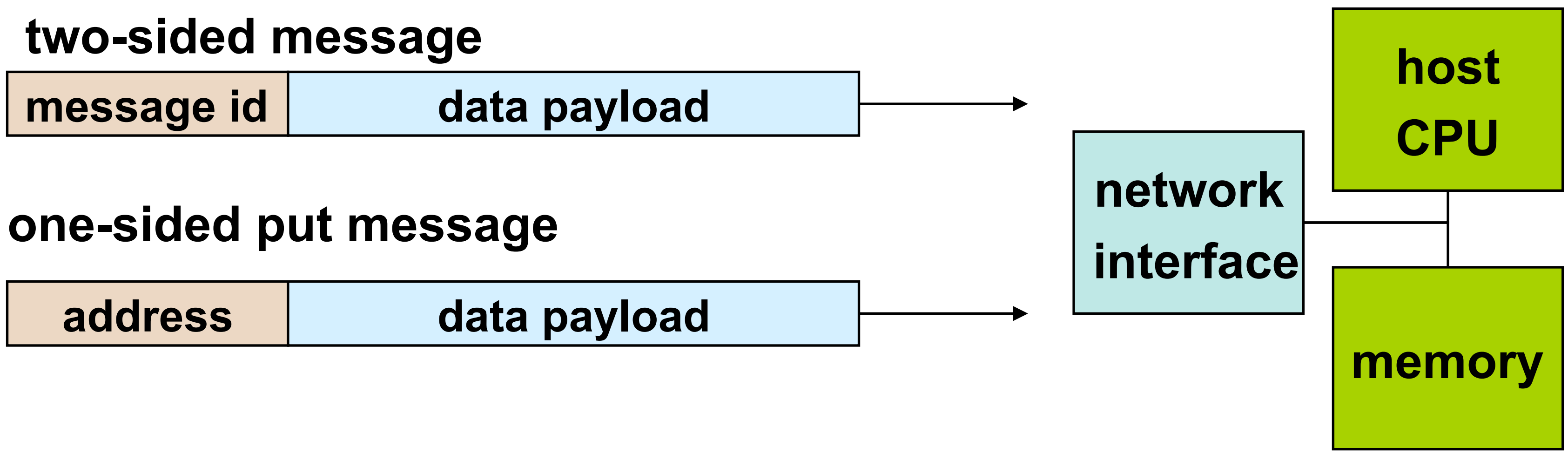
UPC Non-blocking Bulk Operations

Important for performance:

- **Communication overlap with computation**
- **Communication overlap with communication (pipelining)**
- **Low overhead communication**

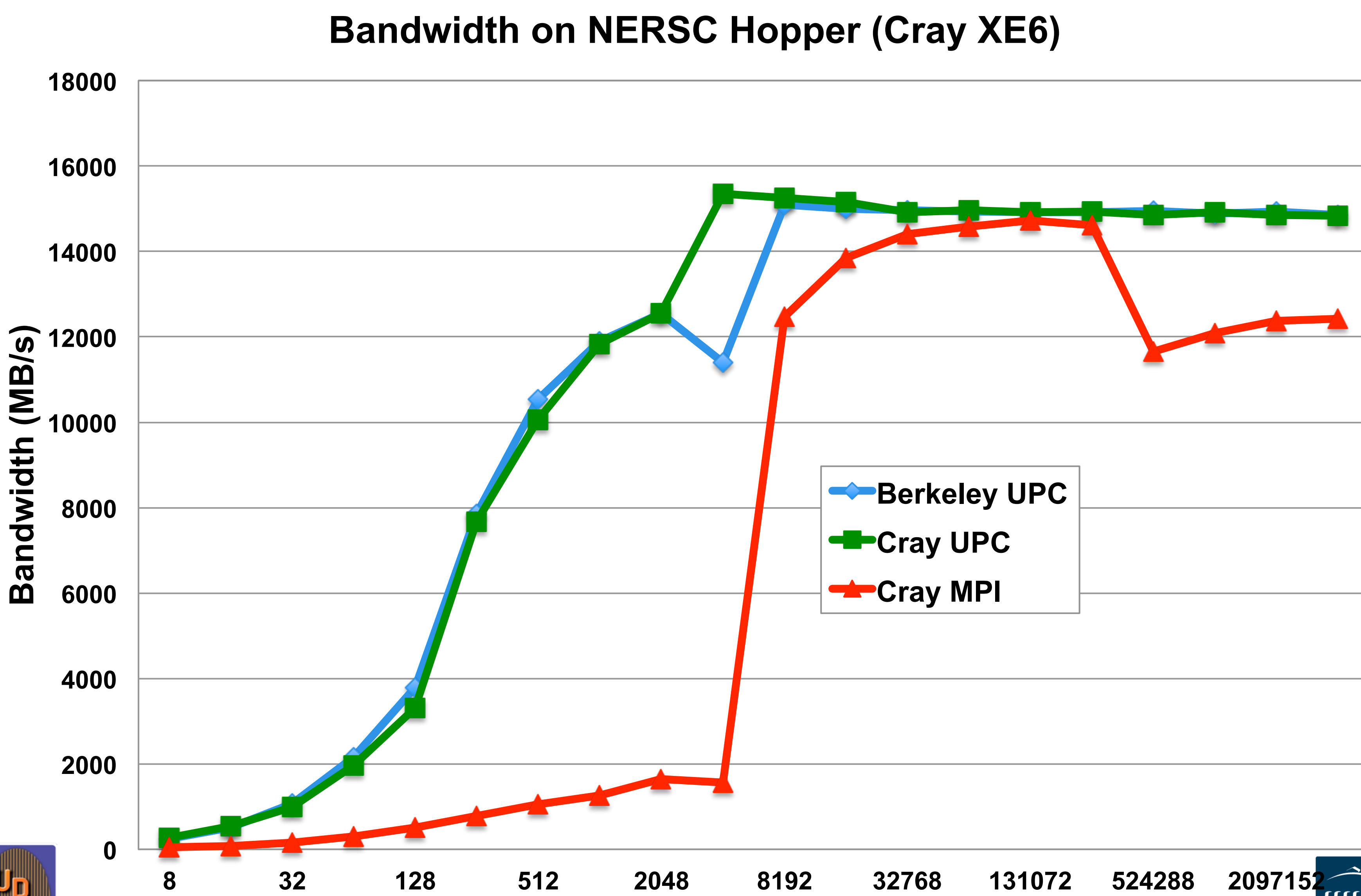
```
#include<upc_nb.h>

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

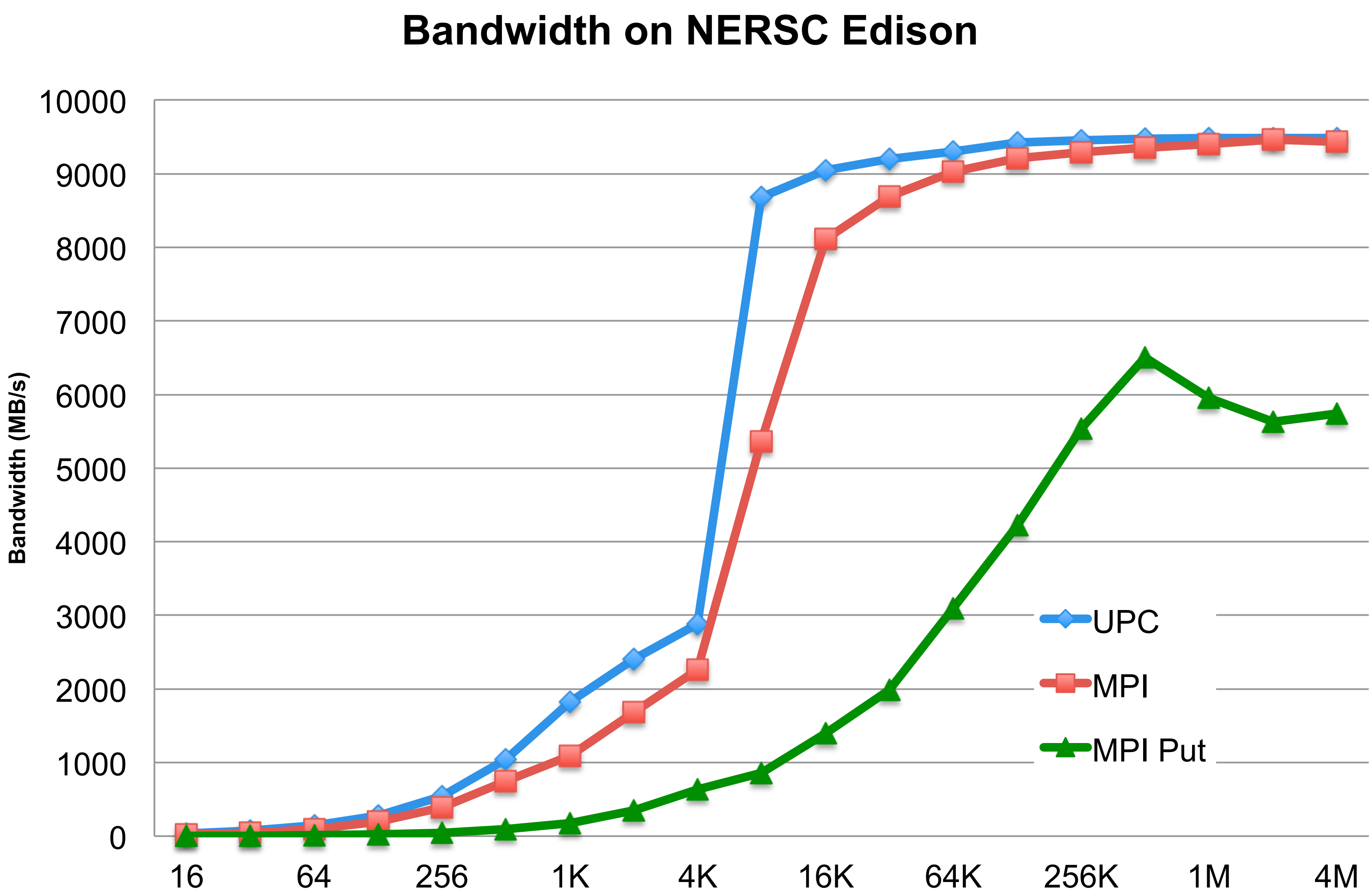


- A two-sided messages needs to be matched with a receive
  - Ordering requirements on messages can also hinder bandwidth
- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Decouples transfer from synchronization
  - Avoids interrupting the CPU or storing data from CPU (preposts)

Bandwidths on Cray XE6 Gemini Network (as on Titan)

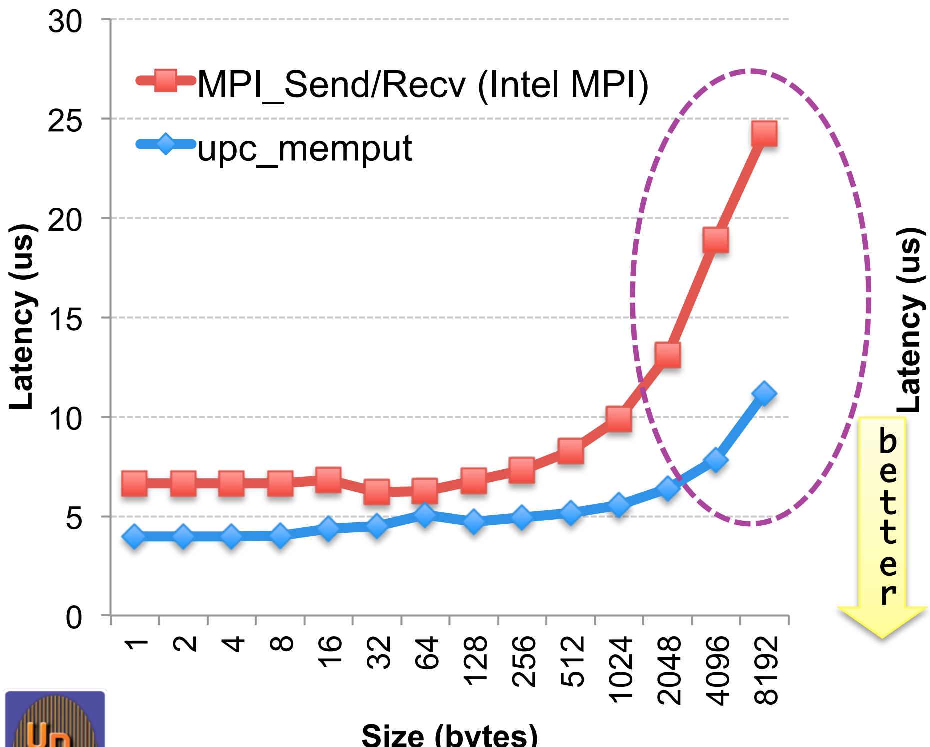


Bandwidths on Cray XC30 Aries Network (Edison)

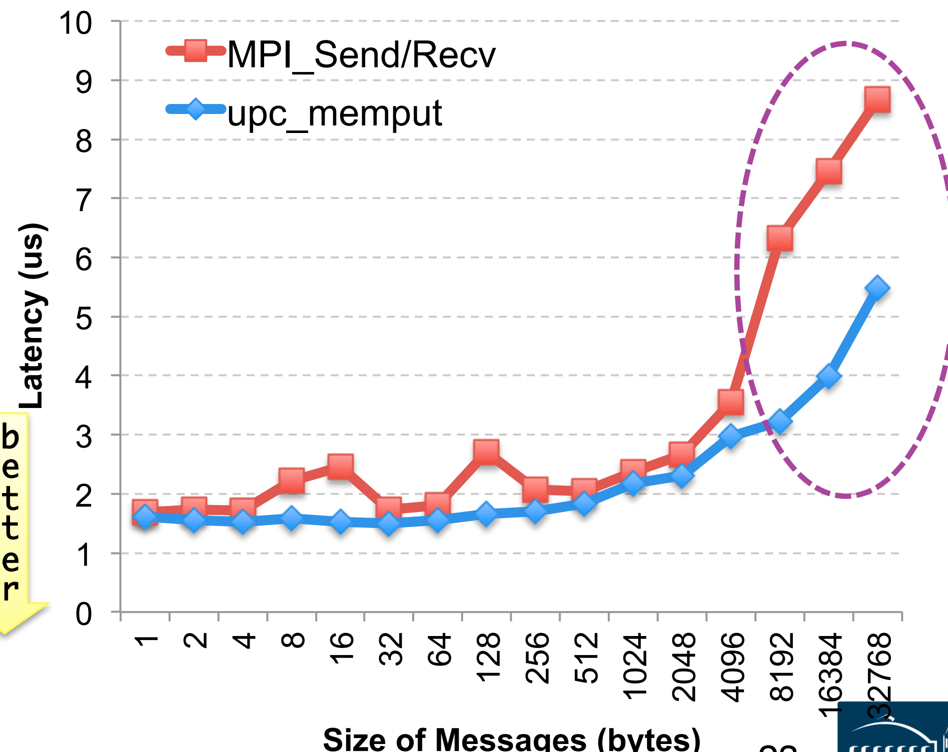


Why Should You Care about PGAS?

Latency between 2 Xeon Phi's via Infiniband



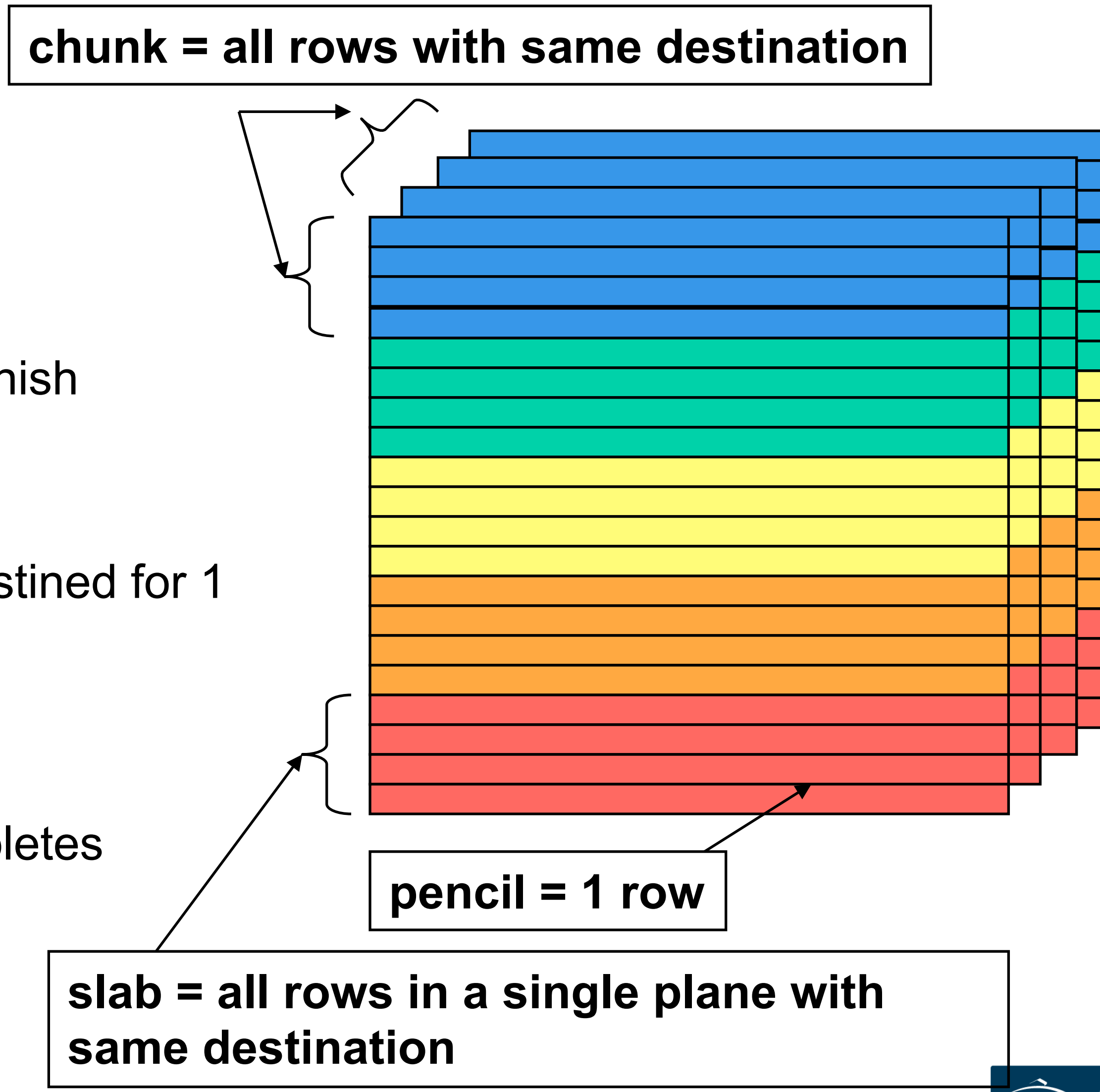
Latency between 2 Intel IvyBridge nodes on NERSC Edison (Cray XC30)



Application Challenge: Fast All-to-All

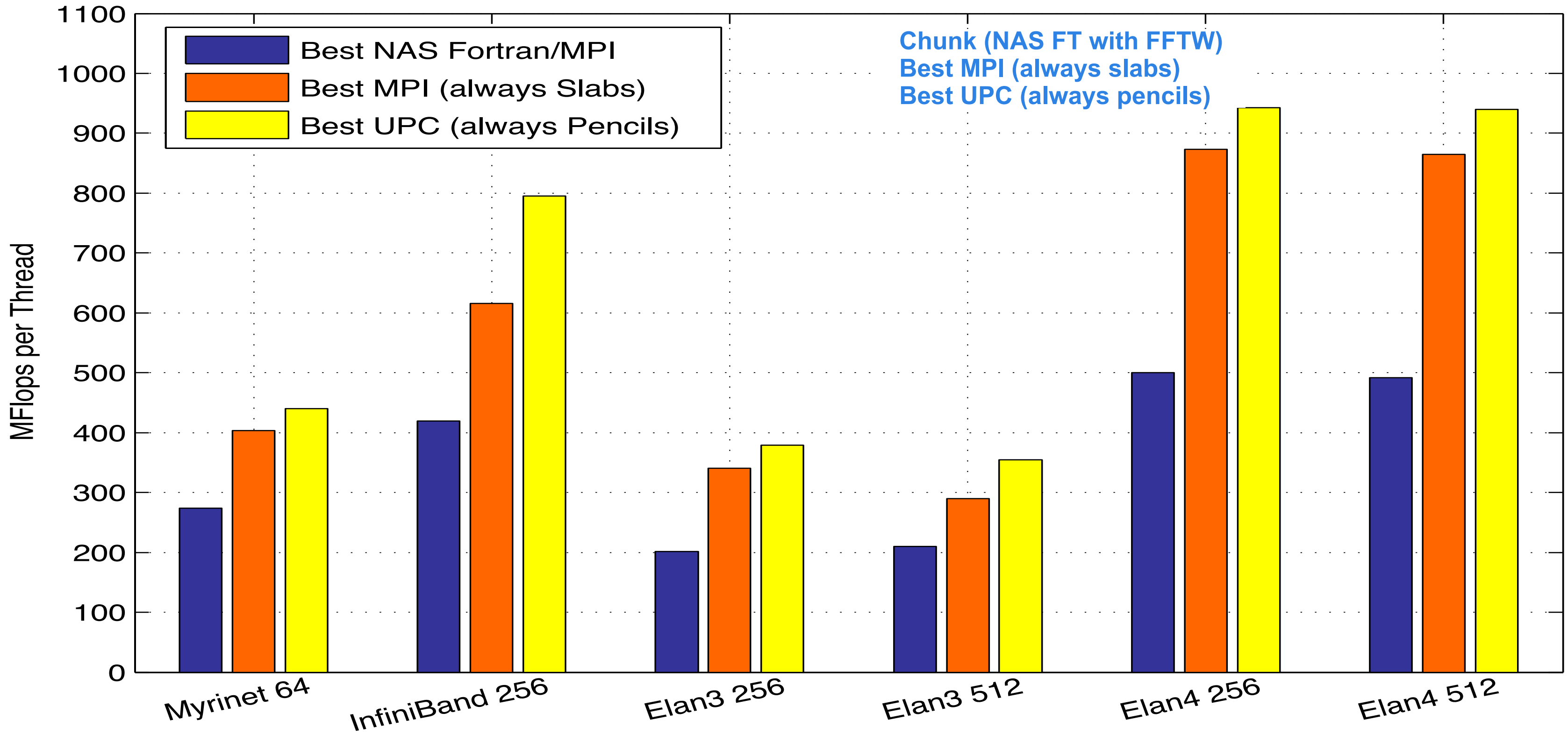
Transpose in 3D FFT

- 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

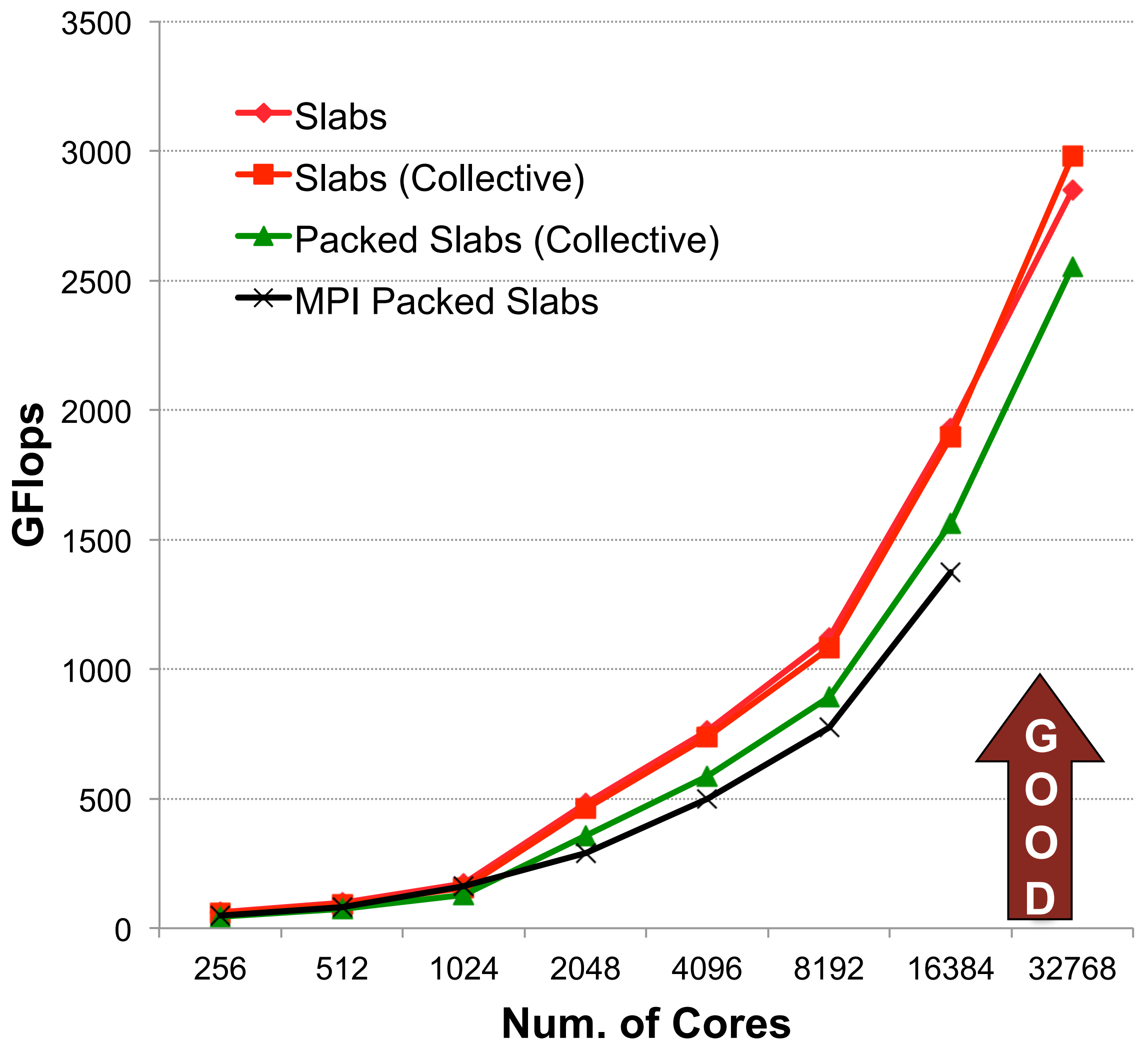
Bisection Bandwidth



- Avoid congestion at node interface: allow all cores to communicate
- Avoid congestion inside global network: spread communication over longer time period (send early and often)

FFT Performance on BlueGene/P (Mira)

- **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



UPC Atomic Operations

- More efficient than using locks when applicable

```
upc_lock();
update();                    vs     atomic_update();
upc_unlock();
```

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

- Example: atomic fetch-and-add

  int bupc_atomici_fetchadd_relaxed (shared void *ptr, int op);

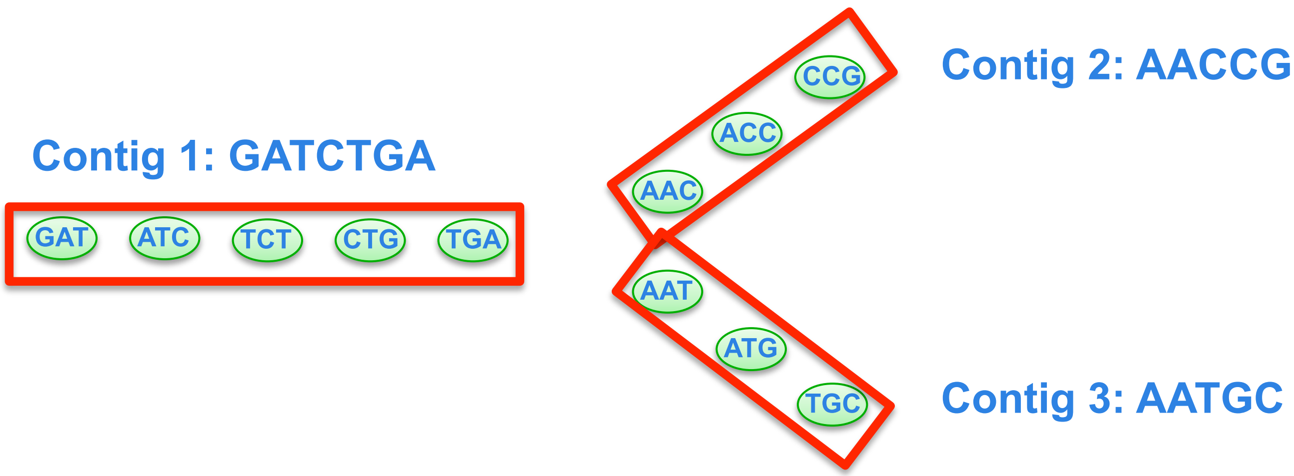
- See more examples on the web page:
  http://upc.lbl.gov/docs/user/#atomics

De novo Genome Assembly

- **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

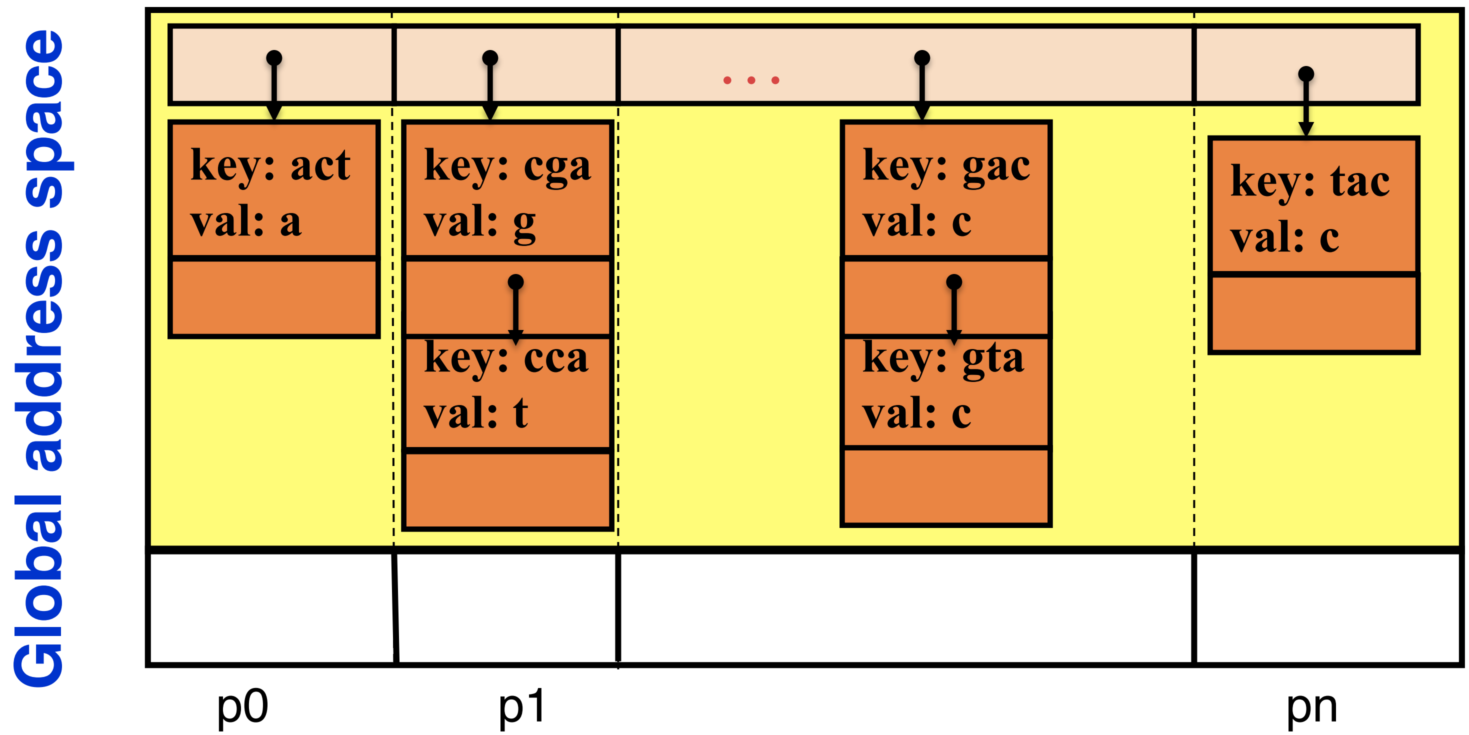
PGAS in Genome Assembly

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



- Hash tables used here (and in other assembly phases)
    - Different use cases, different implementations
- Some tricky synchronization to deal with conflicts

Partitioned Global Address Space Programming

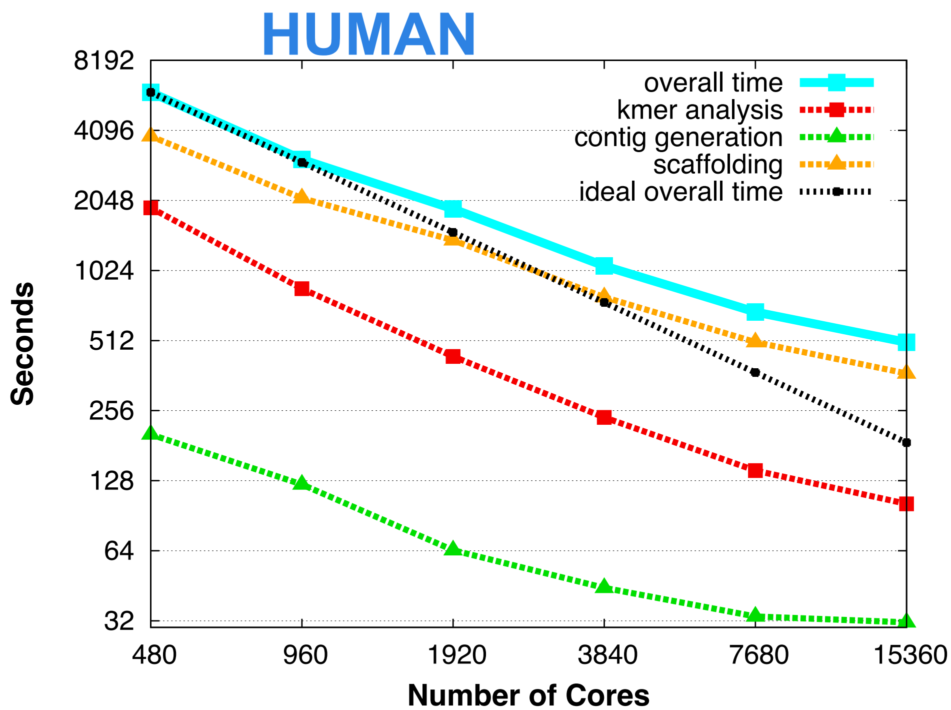


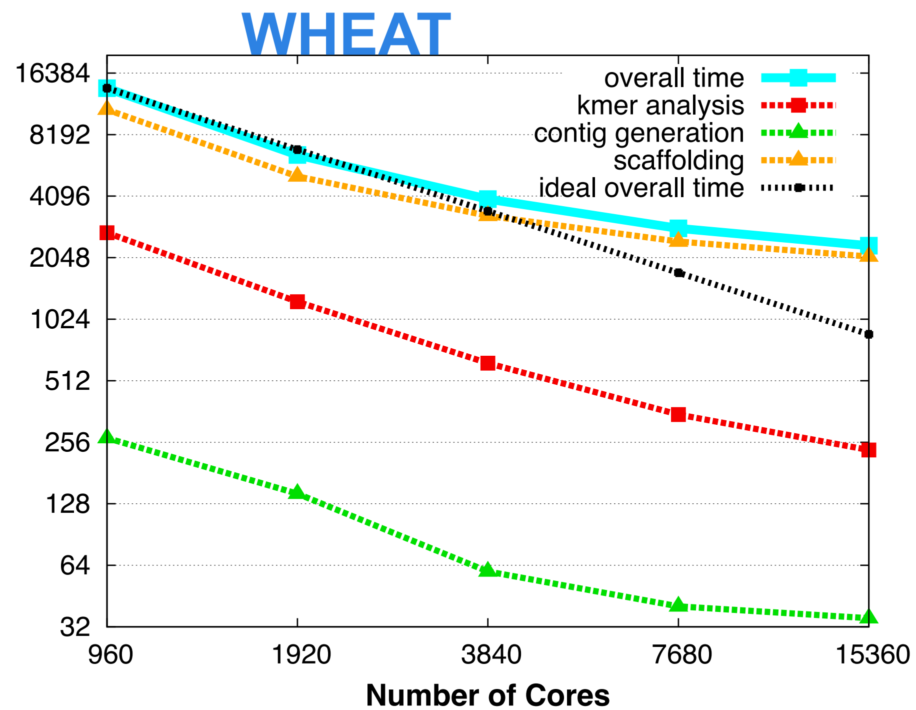
- Store the connections between read fragments (K-mers) in a hash table
- Allows for TB-PB size data sets

HipMer (High Performance Meraculous) Assembly Pipeline

Distributed Hash Tables in PGAS

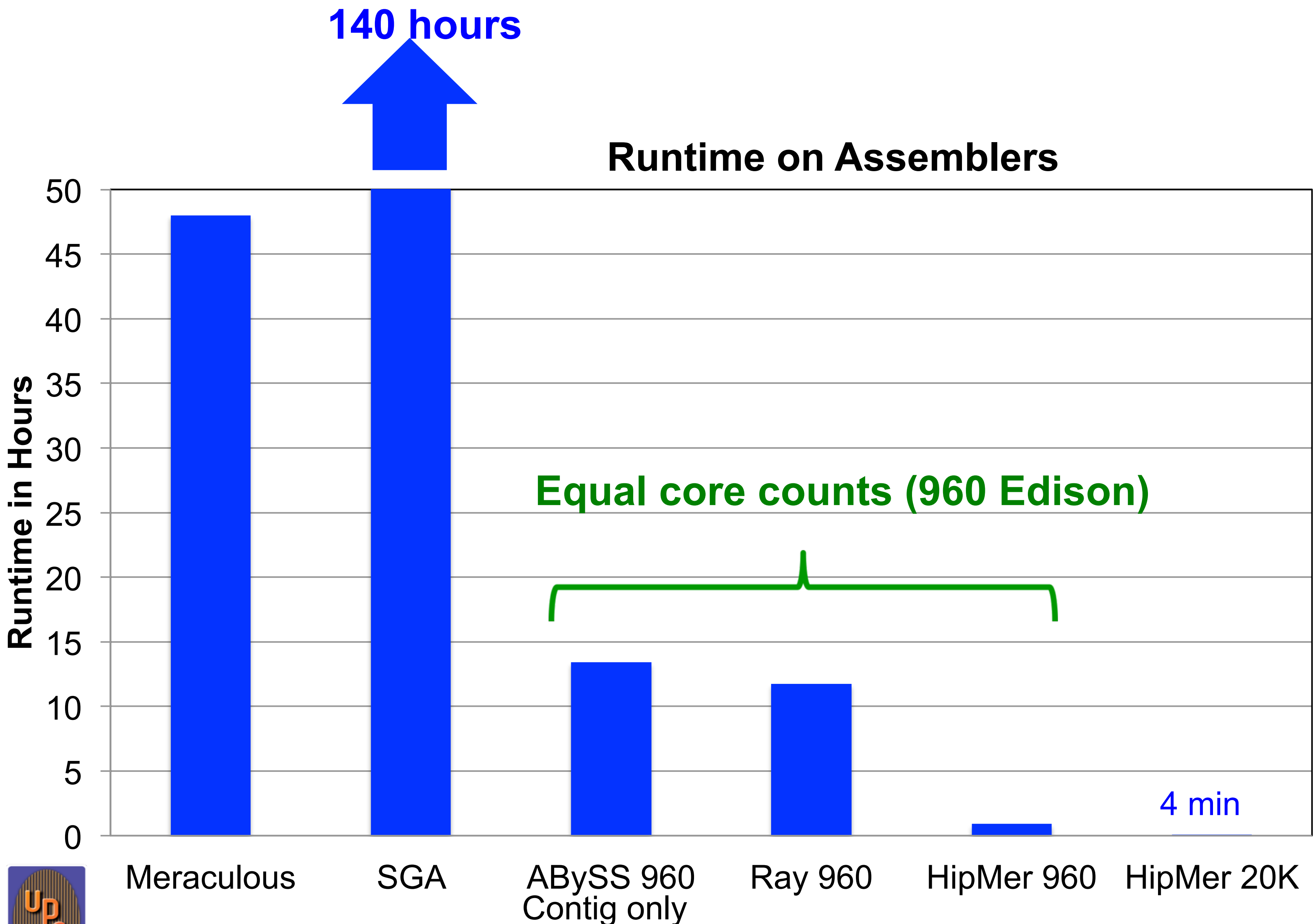
- Remote Atomics, Dynamic Aggregation, Software Caching
- 13x Faster than MPI code (Ray) on 960 cores





Evangelos Georganas, Aydın Buluç, 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

Comparison to other Assemblers



Science Impact: HipMer is transformative

- Human genome (3Gbp) “de novo” assembled :
  - Meraculous: 48 hours
  - HipMer: 4 minutes (720x speedup relative to Meraculous)

Makes unsolvable problems solvable!

- Wheat genome (17 Gbp) “de novo” assembled (2014):
  - Meraculous (did not run):
  - HipMer: 39 minutes; 15K cores (first all-in-one assembly)

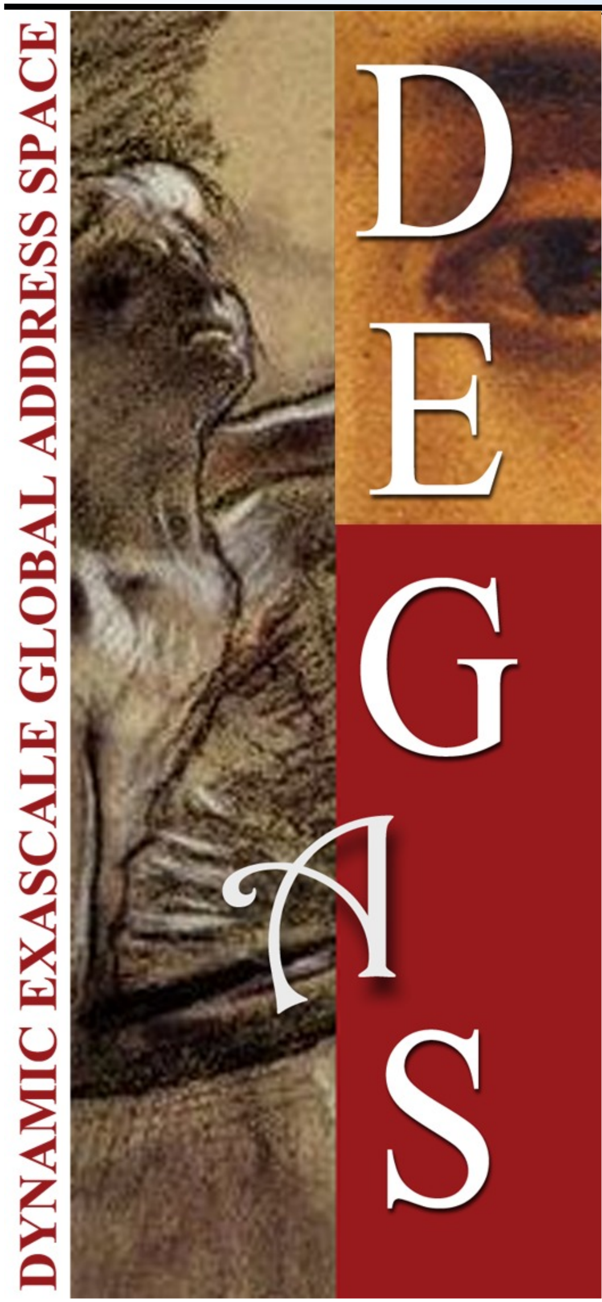
- Pine genome (20 Gbp) “de novo” assembled (2014) :
  - Masurca : 3 months; 1 TB RAM

- Wetland metagenome (1.25 Tbp) analysis (2015):
  - Meraculous (projected): 15 TB of memory
  - HipMER: Strong scaling to over 100K cores (contig gen only)

DEGAS

Georganas, Buluc, Chapman, Oliker, Rokhsar, Yelick, [Aluru,Egan,Hofmeyr] in SC14, IPDPS15, SC15

BERKELEY LAB



UPC++

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

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

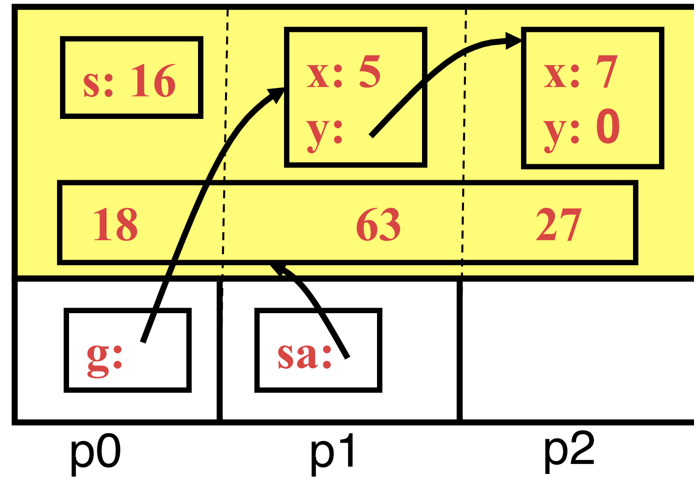
DEGAS is a DOE-funded X-Stack with Lawrence Berkeley National Lab, Rice Univ., UC Berkeley, and UT Austin.

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



- Remote methods, async

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

- Teams for hierarchical algorithms and machines

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

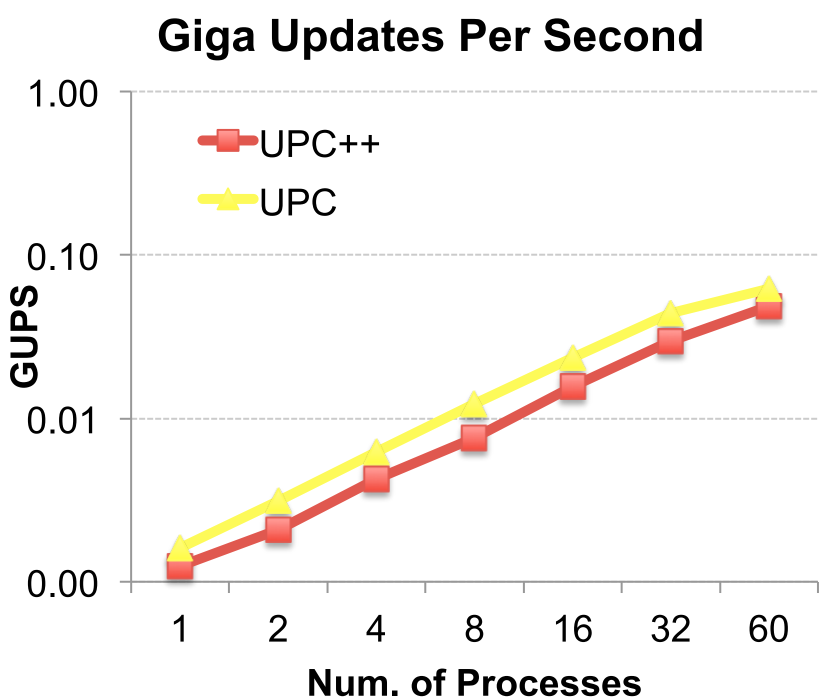
- Interoperability is key; UPC++ can be use with OpenMP or MPI

UPC++ Performance Close to UPC

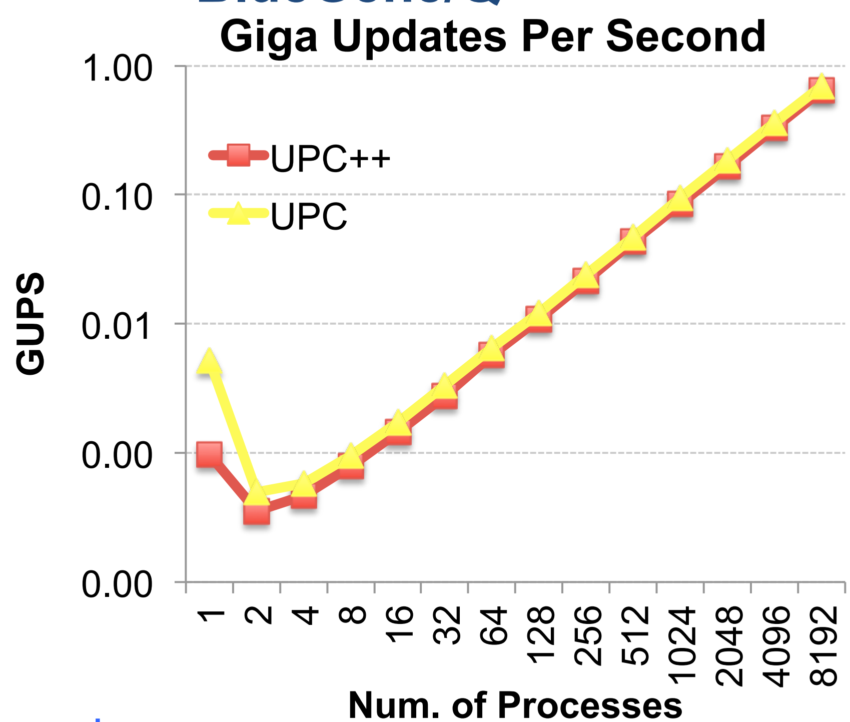
UPC++ is a library, not a compiled language, yet performance is comparable

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

MIC



BlueGene/Q



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

DEGAS

BERKELEY LAB
Lawrence Berkeley National Laboratory

Bulk Communication with One-Sided Data Transfers

```
// 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)

```
global_ptr<data_type> ptr;
```

- Dynamic shared memory allocation

```
global_ptr<T> allocate<T>(uint32_t where,
                          size_t count);
void deallocate(global_ptr<T> ptr);
```

Example: allocate space for 512 integers on rank 2

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

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

Async Task Example

```
#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 remote tasks to complete
  return 0;
}
```

DEGAS

BERKELEY LAB

Async with Anonymous Functions (C++ Lambda)

```
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
```

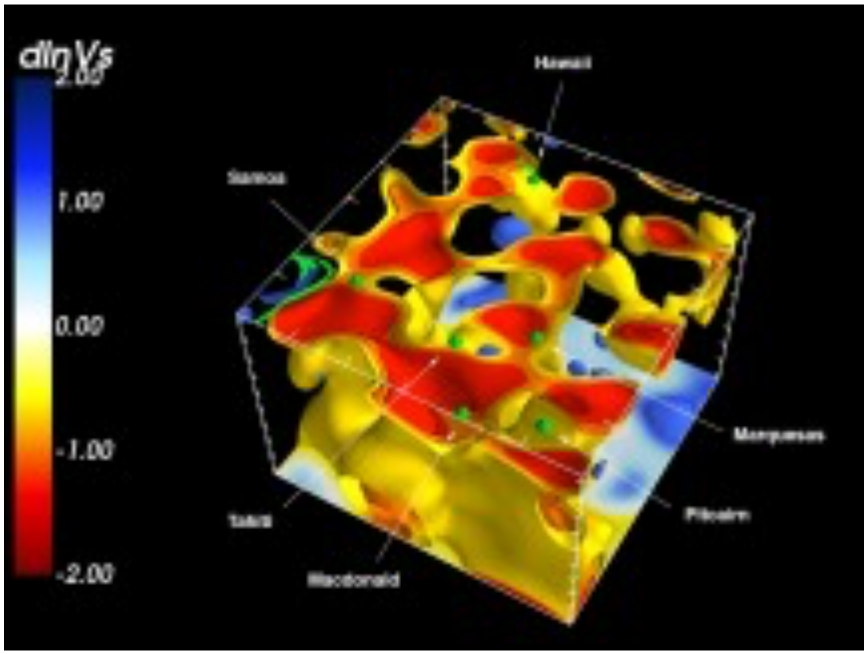
Function arguments and lambda-captured values must be std::is_trivially_copyable.

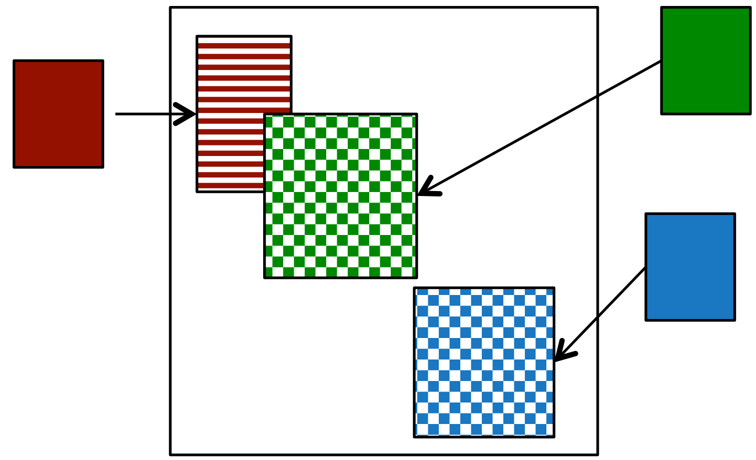
DEGAS

BERKELEY LAB
Lawrence Berkeley National Laboratory

Application Challenge: Data Fusion in UPC++

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

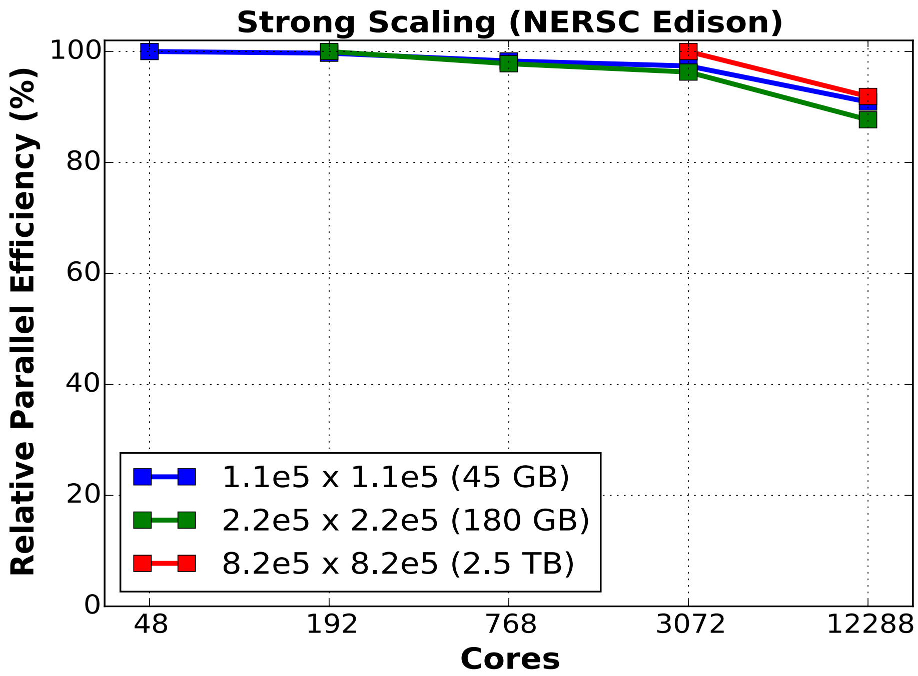




First ever sharp, three-dimensional scan of Earth’s interior that conclusively connects plumes of hot rock rising through the mantle with surface hotspots that generate volcanic island chains like Hawaii, Samoa and Iceland.

French and Romanowicz use code with UPC++ phase to compute *first ever* whole-mantle global tomographic model using numerical seismic wavefield computations (F & R, 2014, GJI, extending F et al., 2013, Science).

Application Challenge: Data Fusion in UPC++



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

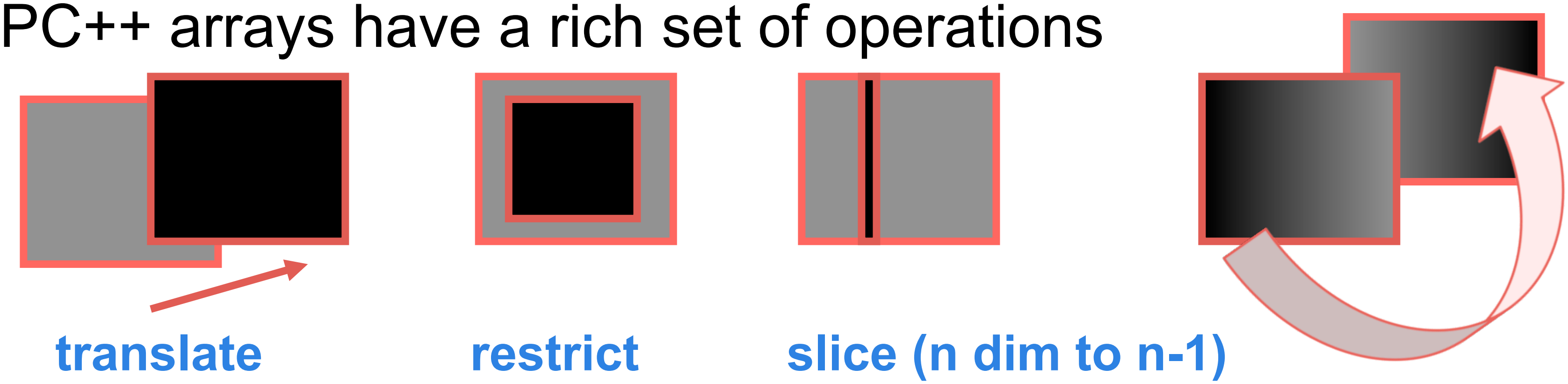
See French et al, IPDPS 2015 for parallelization overview.

DEGAS

BERKELEY LAB
Lawrence Berkeley National Laboratory

Multidimensional Arrays in UPC++ (and Titanium)

- UPC++ arrays have a rich set of operations

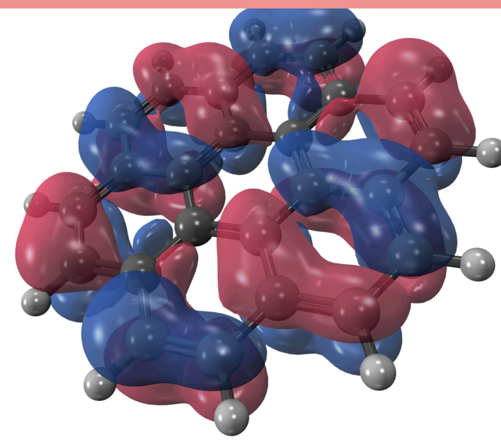


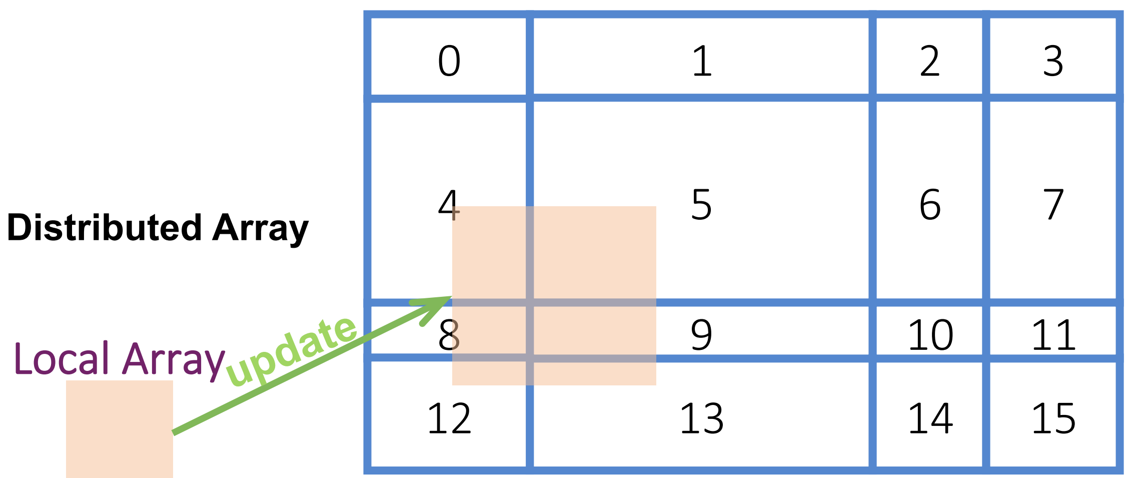
- None of these modify the original array, they just create another view of the data in that array
- You create arrays with a RectDomain and get it back later using A.domain() for array A
  - A Domain is a set of points in space
  - A RectDomain is a rectangular one
- Operations on Domains include +,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

Load Balancing and Irregular Matrix Transpose

- **Hartree Fock example (e.g., in NWChem)**
  - **Inherent load imbalance**
- **UPC++**
  - Work stealing and fast atomics
  - Distributed array: easy and fast transpose
- **Impact**
  - *20% faster than the best existing solution (GTFock with Global Arrays)*

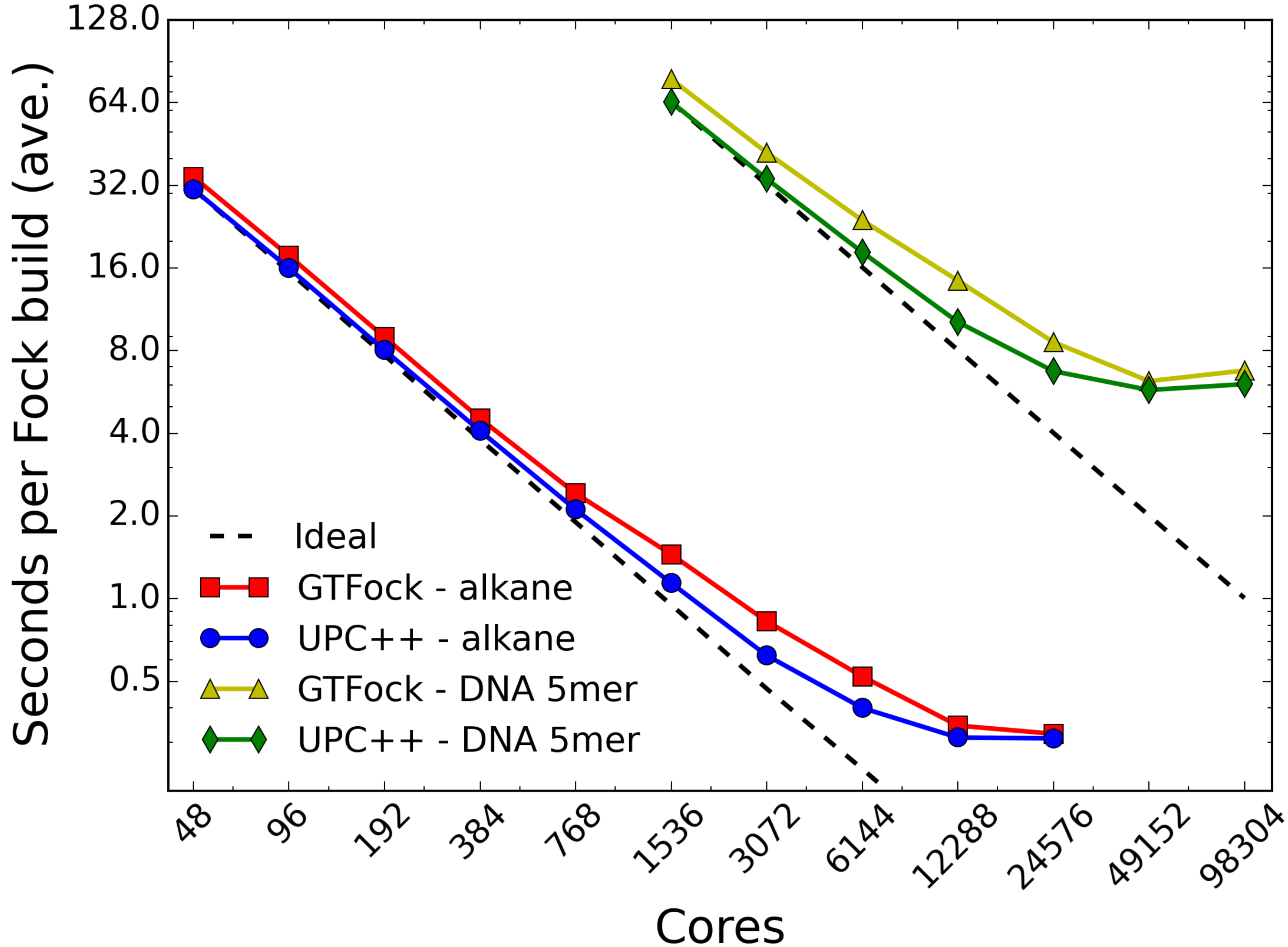
Increase scalability!





David Ozog , Amir Kamil , Yili Zheng, Paul Hargrove , Jeff R. Hammond, Allen Malony, Wibe de Jong, Katherine Yelick

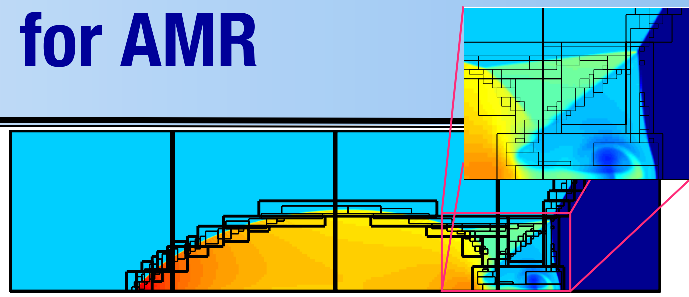
Hartree Fock Code in UPC++



Strong Scaling of UPC++ HF Compared to GTFock with Global Arrays on NERSC Edison (Cray XC30)

David Ozog , Amir Kamil , Yili Zheng, Paul Hargrove , Jeff R. Hammond, Allen Malony, Wibe de Jong, Katherine Yelick

Arrays in a Global Address Space for AMR

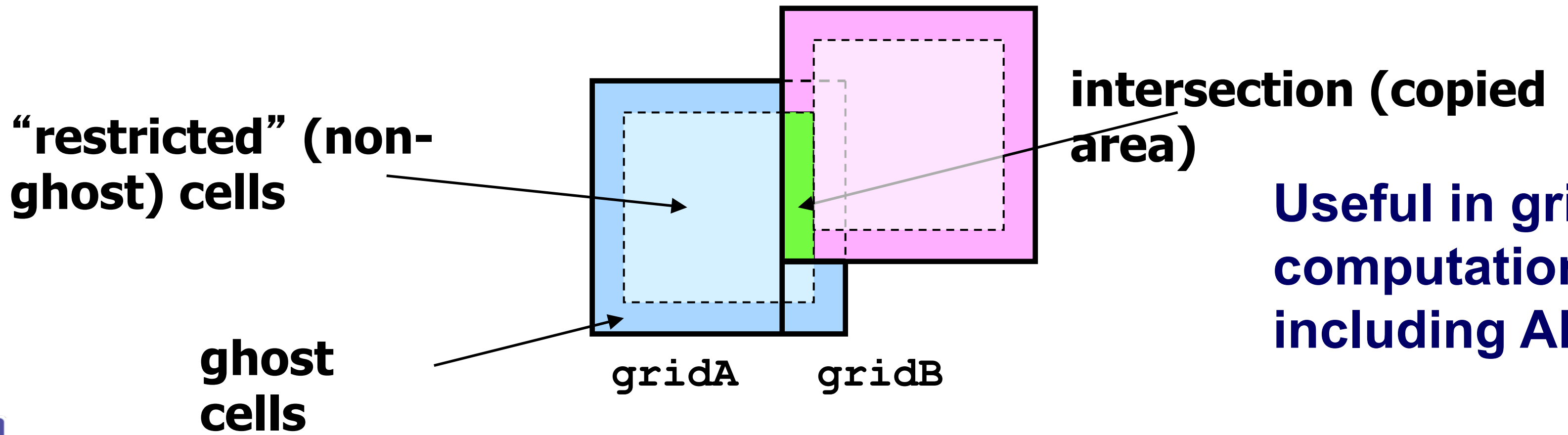


- Key features of UPC++ arrays
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to iterate over interior:

```
foreach (idx, gridB.shrink(1).domain())
```

- Array copies automatically work on intersection

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

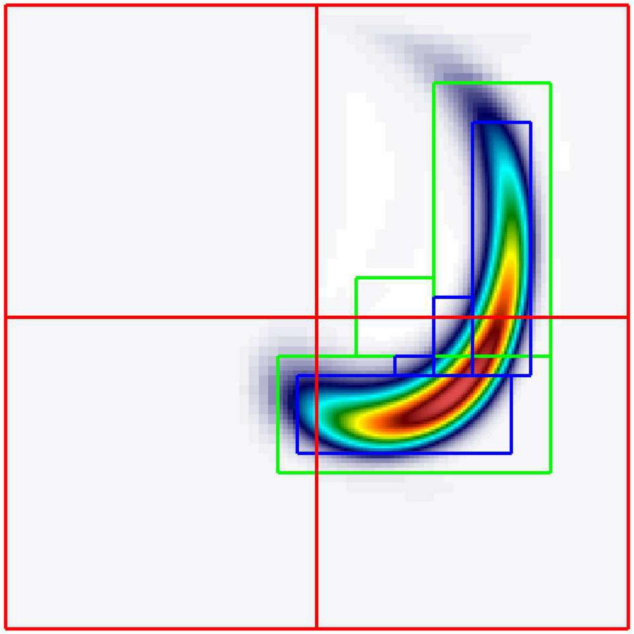


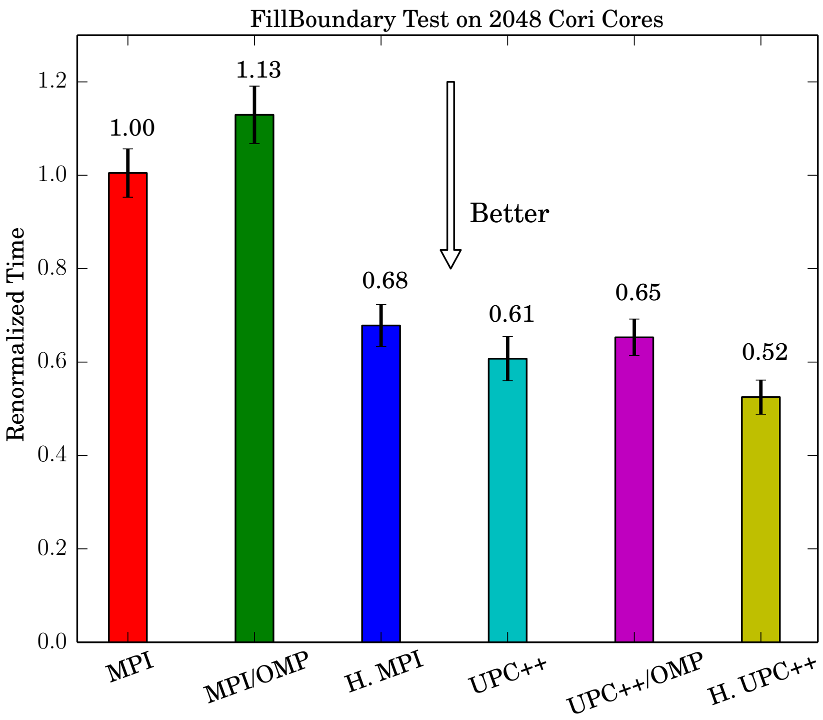
Useful in grid computations including AMR

UPC++ arrays based on Titanium Arrays

UPC++ Communication Speeds up AMR

- Adaptive Mesh Refinement on Block-Structured Meshes
  - Used in ice sheet modeling, climate, subsurface (fracking),





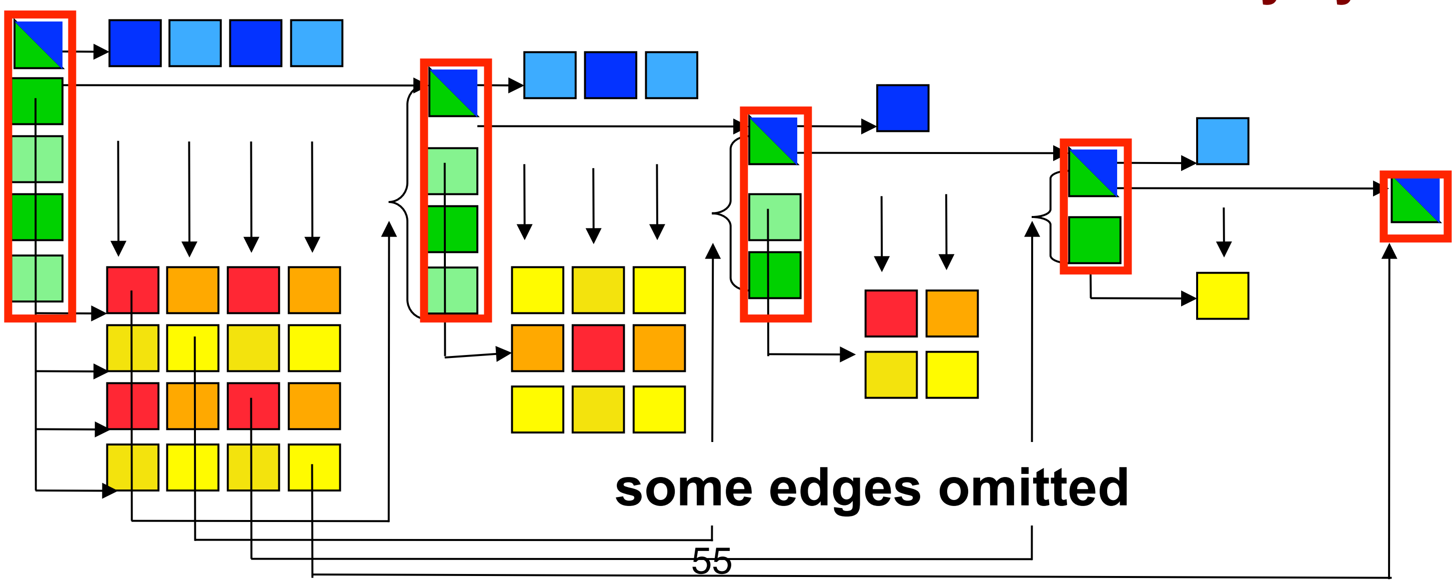
Hierarchical UPC++ (distributed / shared style)

- UPC++ plus UPC++ is 2x faster than MPI plus OpenMP
- MPI + MPI also does well

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

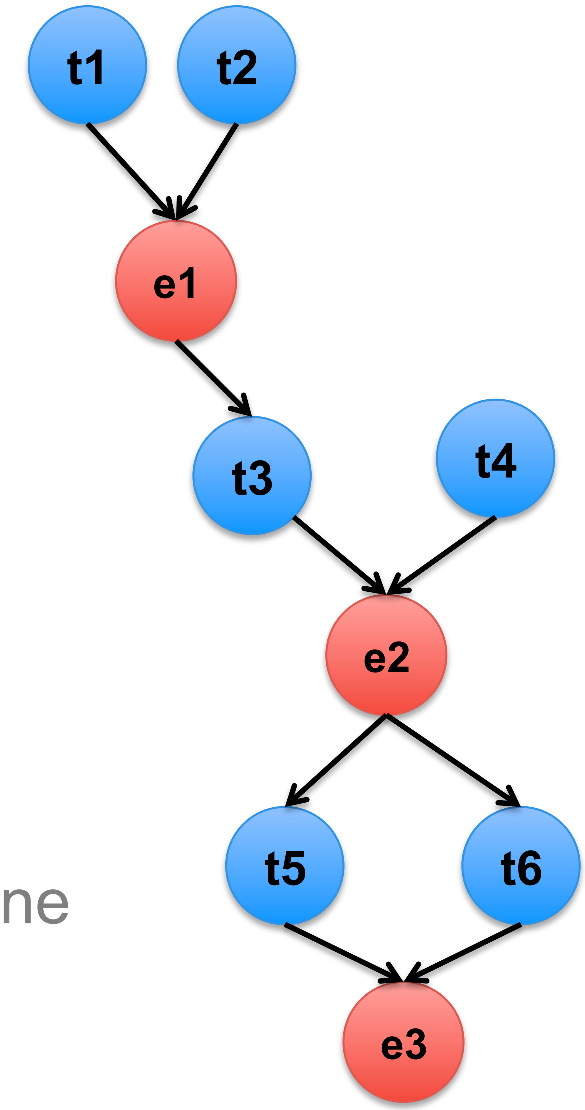
Uses a Berkeley extension to UPC to remotely synchronize



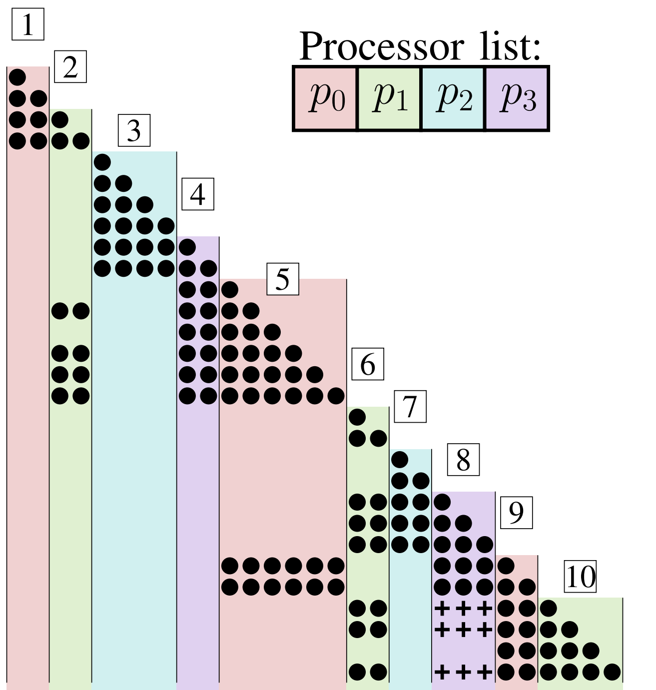
some edges omitted

Example: Building A Task Graph

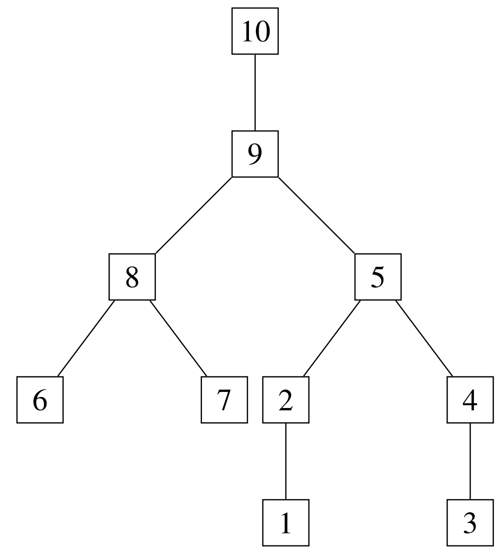
```
using namespace upcxx;
event e1, e2, e3;

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
```



symPACK: Sparse Cholesky



(a) Structure of Cholesky factor **L** (



) Supernodal elimination tree of atrix **A**

- Sparse Cholesky using fan-both algorithm in UPC++
  - Uses asyncs with dependencies

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

symPACK: Sparse Cholesky

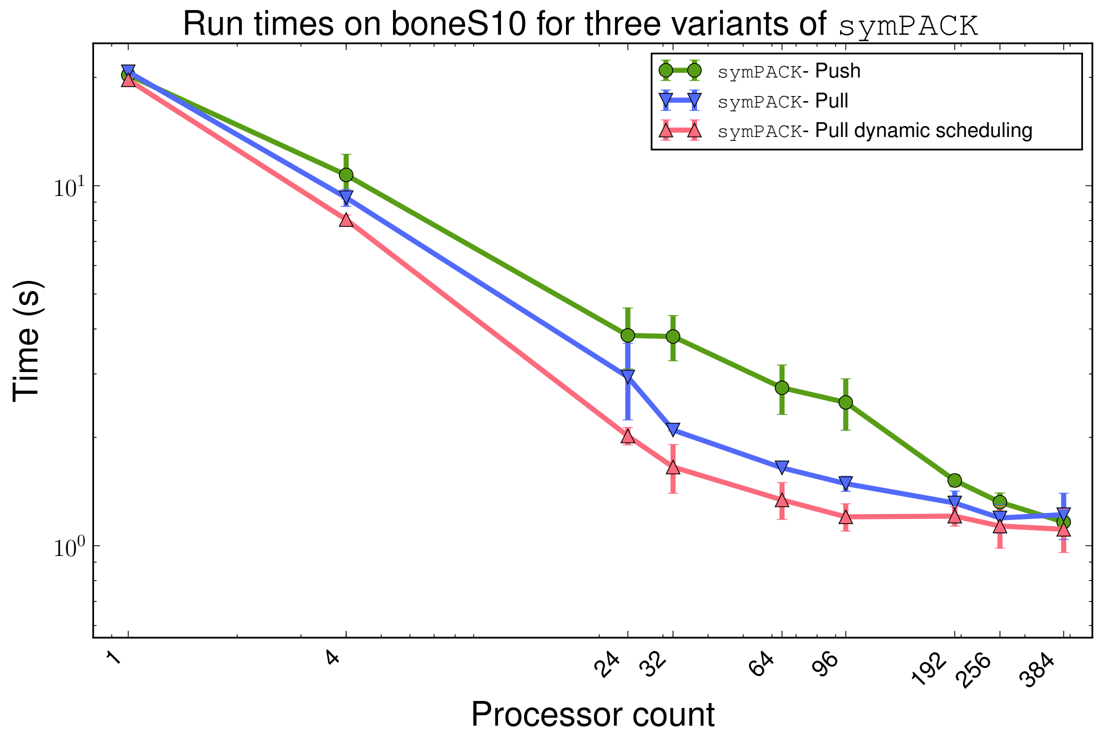


Figure 7: Impact of communication strategy and scheduling on symPACK performance

- Scalability of symPACK on Cray XC30 (Edison)
  - Comparable or better than best solvers (evaluation in progress)
  - Notoriously hard parallelism problem

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

Summary: PGAS for Irregular Applications

- Lower overhead of communication makes PGAS useful for latency-sensitive problems or bisection bandwidth problems
- Specific application characteristics that benefit:
  - Fine-grained updates (Genomics HashTable construction)
  - Latency-sensitive algorithms (Genomics DFS)
  - Distributed task graph (Cholesky)
  - Work stealing (Hartree Fock)
  - Irregular matrix assembly / transpose (Seismic, HF)
  - Medium-grained messages (AMR)
  - All-to-all communication (FFT)
- There are also benefits of thinking algorithmically in this model: parallelize things that are otherwise hard to imagine

Summary: PGAS for Modern HPC Systems

- The lower overhead of communication is also important given current machine trends
  - **Many lightweight cores** per node (do not want a hefty serial communication software stack to run on them)
  - **RDMA mechanisms** between nodes (decouple synchronization from data transfer)
  - **GAS on chip:** direct load/store on chip without full cache coherence across chip
  - **Hierarchical machines:** fits both shared and distributed memory, but supports hierarchical algorithms
  - **New models of memory:** High Bandwidth Memory on chip or NVRAM above disk

Installing Berkeley UPC++, UPC, and GASNet

Available on Mac OSX, Linux, Infiniband clusters, Ethernet clusters, and most HPC systems

- UPC++ Open source with BSD license

https://bitbucket.org/upcxx

- UPC++ installation

https://bitbucket.org/upcxx/upcxx/wiki/Installing%20UPC++

- GASNet communication

https://gasnet.lbl.gov

- Examples
  - DAXPY, Conjugate Gradient, FFT, GUPS, MatrixMultiply, Mutigrid, Minimum Degree Ordering, Sample Sort, Sparse Matrix-Vector mutliply

Using Berkeley UPC at NERSC or ALCF

Load the bupc module via

```
module load bupc
```

Compile code with the upcc

```
upcc -V  // shows version
```

Add the following line to your ~/.soft file:

```
PATH += /home/projects/pgas/berkeley_upc-2.22.3/V1R2M2/gcc-narrow/bin/
```

OR, if using the xl compilers, add:

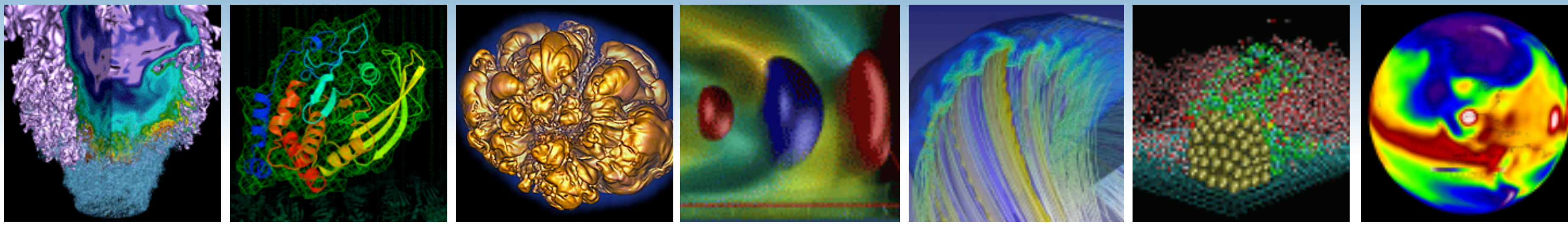
```
PATH +=  /home/projects/pgas/berkeley_upc-2.22.3/V1R2M2/xlc-narrow/bin/
```

Run

```
resoft
```

Compile with upcc. To see the version and configuration, run

```
upcc -V
```



LBNL / UCB Collaborators

- Yili Zheng*
- Amir Kamil*
- Paul Hargrove
- Eric Roman
- Dan Bonachea
- Marquita Ellis
- Khaled Ibrahim
- Costin Iancu
- Michael Driscoll
- Evangelos Georganas
- Penporn Koanantakool
- Steven Hofmeyr
- Leonid Oliker

** Former LBNL/UCB*

- John Shalf
- Erich Strohmaier
- Samuel Williams
- Cy Chan
- Didem Unat*
- James Demmel
- Scott French
- Edgar Solomonik*
- Eric Hoffman*
- Wibe de Jong

Thanks!

External collaborators (& their teams!)

- Vivek Sarkar, Rice
- John Mellor-Crummey, Rice
- Mattan Erez, UT Austin