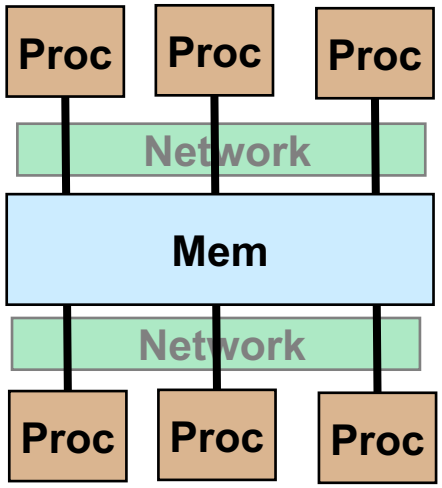
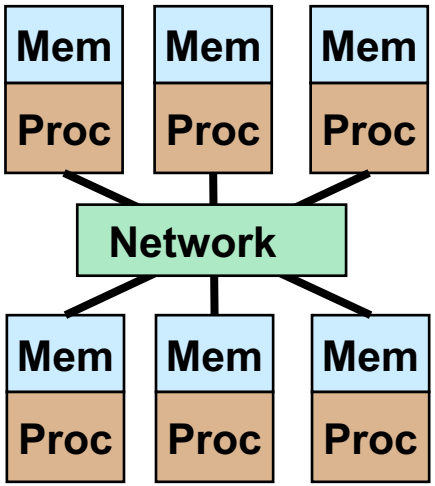
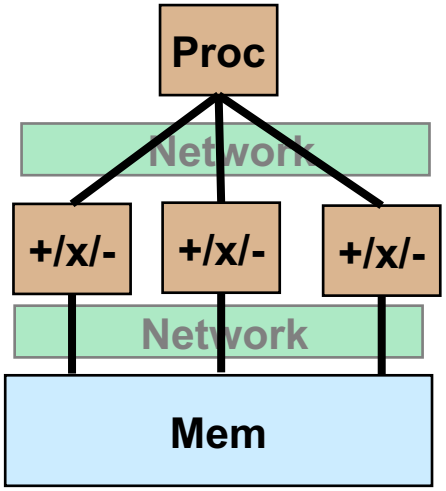
Partitioned Global Address Space Languages
CS267 Lecture 11

Kathy Yelick

Parallel Machines and Programming







Shared Memory	Distributed Memory	Single Instruction Multiple Data (SIMD)
OpenMP, Threads	**MPI (send/receive)**	**Data parallel (collectives)**
Processors execute own instruction stream	Processors execute own instruction stream	One instruction stream (all run same instruction)
Communicate by reading/writing memory	Communicate by sending messages	Communicate through memory
Ideal cost: Cost of a read/write is constant	Memory access time depends on size and whether local vs. remote	Assume unbounded # of arithmetic units

Advantages and disadvantages of each

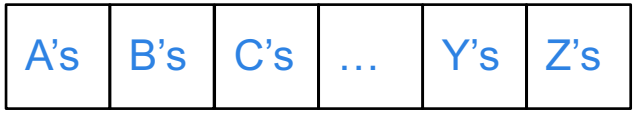
- Shared memory / OpenMP
  - + **Ease:** Easier to parallelize existing serial code
  - – **Correctness:** Race conditions
  - – **Scalability:** No locality control for multi-node scaling
  - – **Performance:** False sharing, lack of parallelism, etc.

- Message Passing / two-sided MPI
  - – **Ease:** More work up front to partition data
  - + **Correctness:** Harder to create races (deadlocks are more of a problem)
  - + **Scalability**: Effectively unlimited
  - + **Performance:** More transparent, but messages are expensive (need to pack/unpack)

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

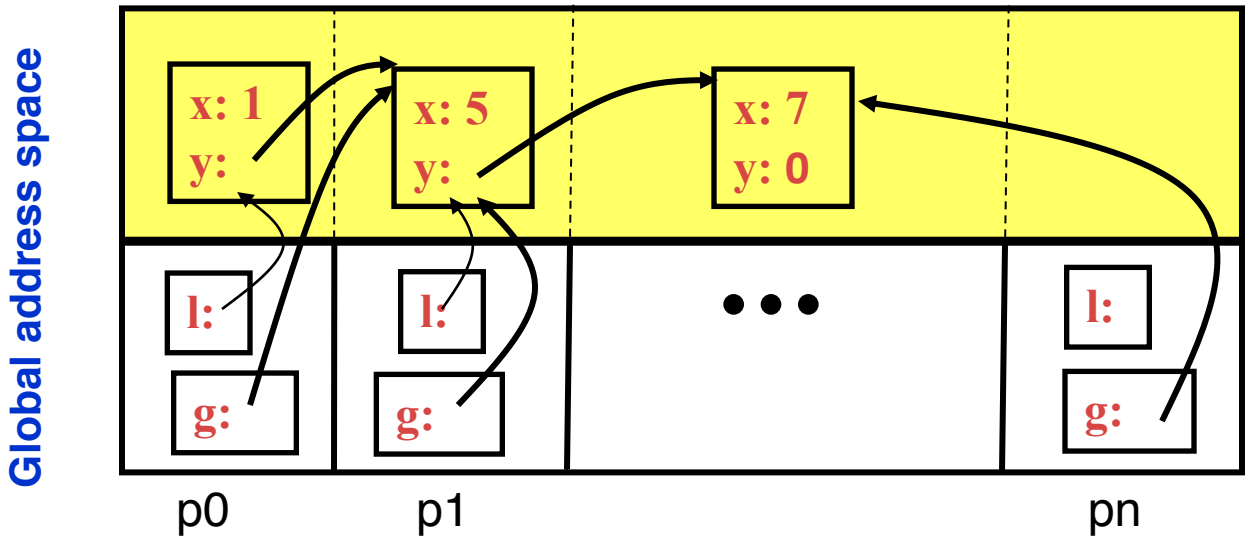
- Shared memory
  - Lock each bucket



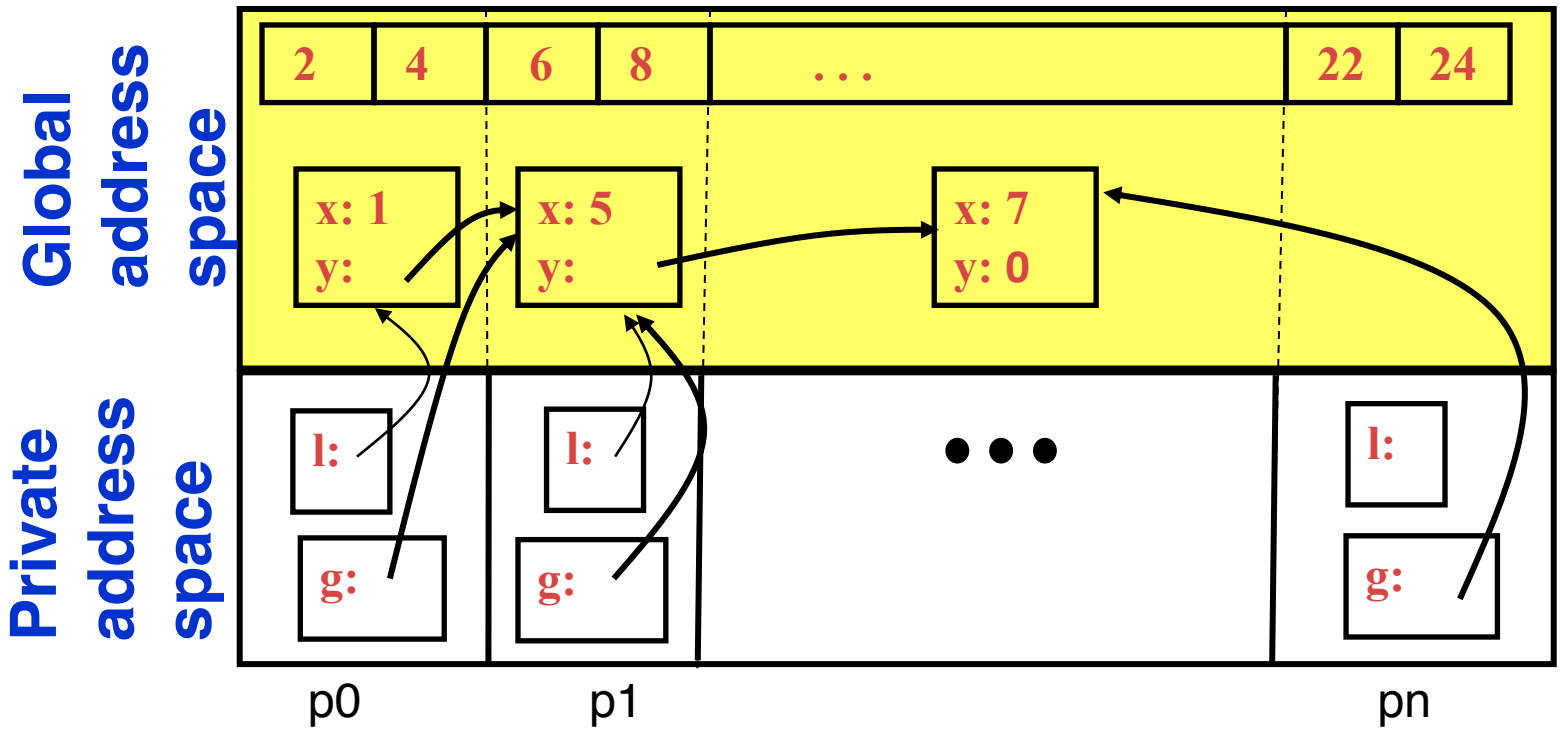
- Message passing: the array is huge and spread out
  - Each processor has a substream and sends +1 to the appropriate processor…
  - When does that processor "receive"?

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

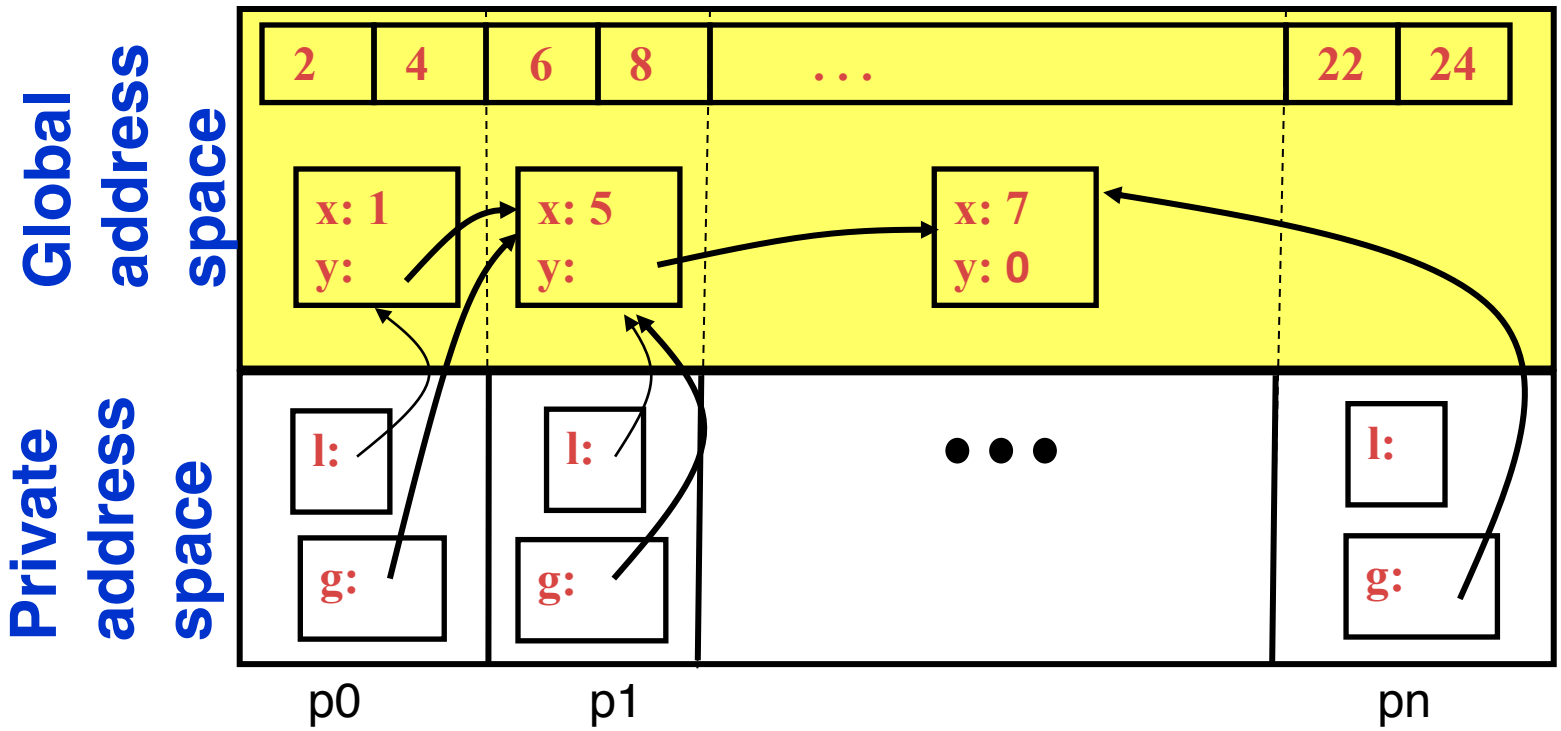


Partitioned Global Address Space (PGAS)



- Need a way to name remote memory (UPC syntax)
  - Global pointers: shared int * p = upc_malloc(4);
  - Distributed arrays: shared int a [12];

One-sided communication in PGAS



- Directly read/write remote memory; partitioned for locality
- One-sided communication underneath (UPC syntax):

  Put: a[i] = … ; *p = ...; upc_mem_put(..)

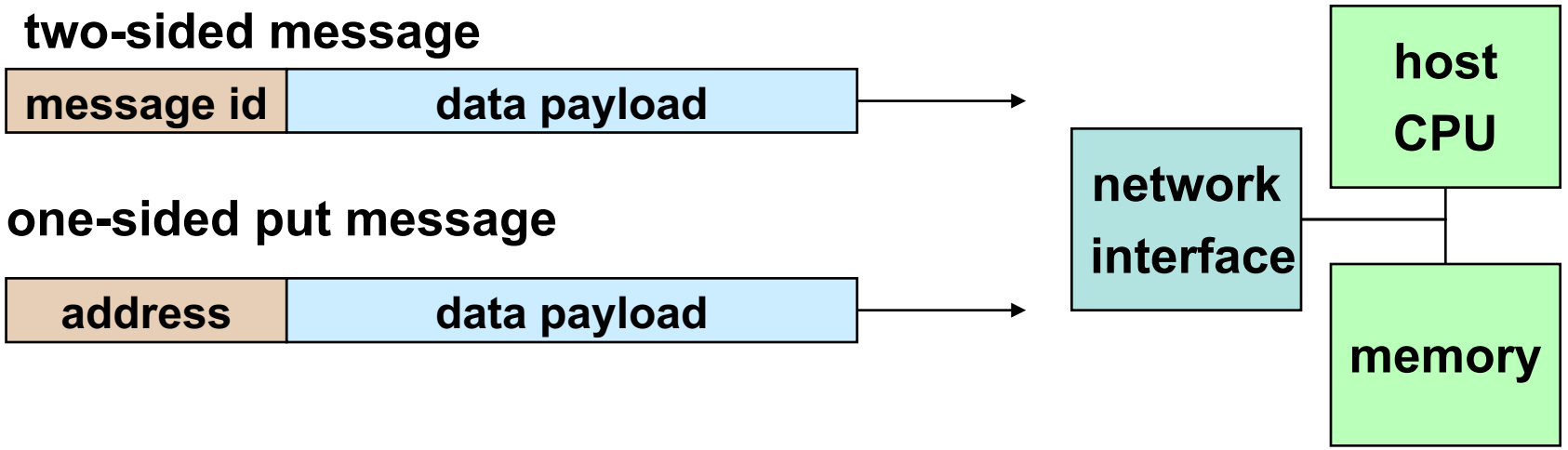
  Get: ... = a[i]...; ... = *p; upc_mem_get(...)

Goals of PGAS Programming

- Applications: convenient programming of irregular codes
  - Graphs
  - Hash tables
  - Sparse matrices
  - Adaptive (hierarchical) meshes

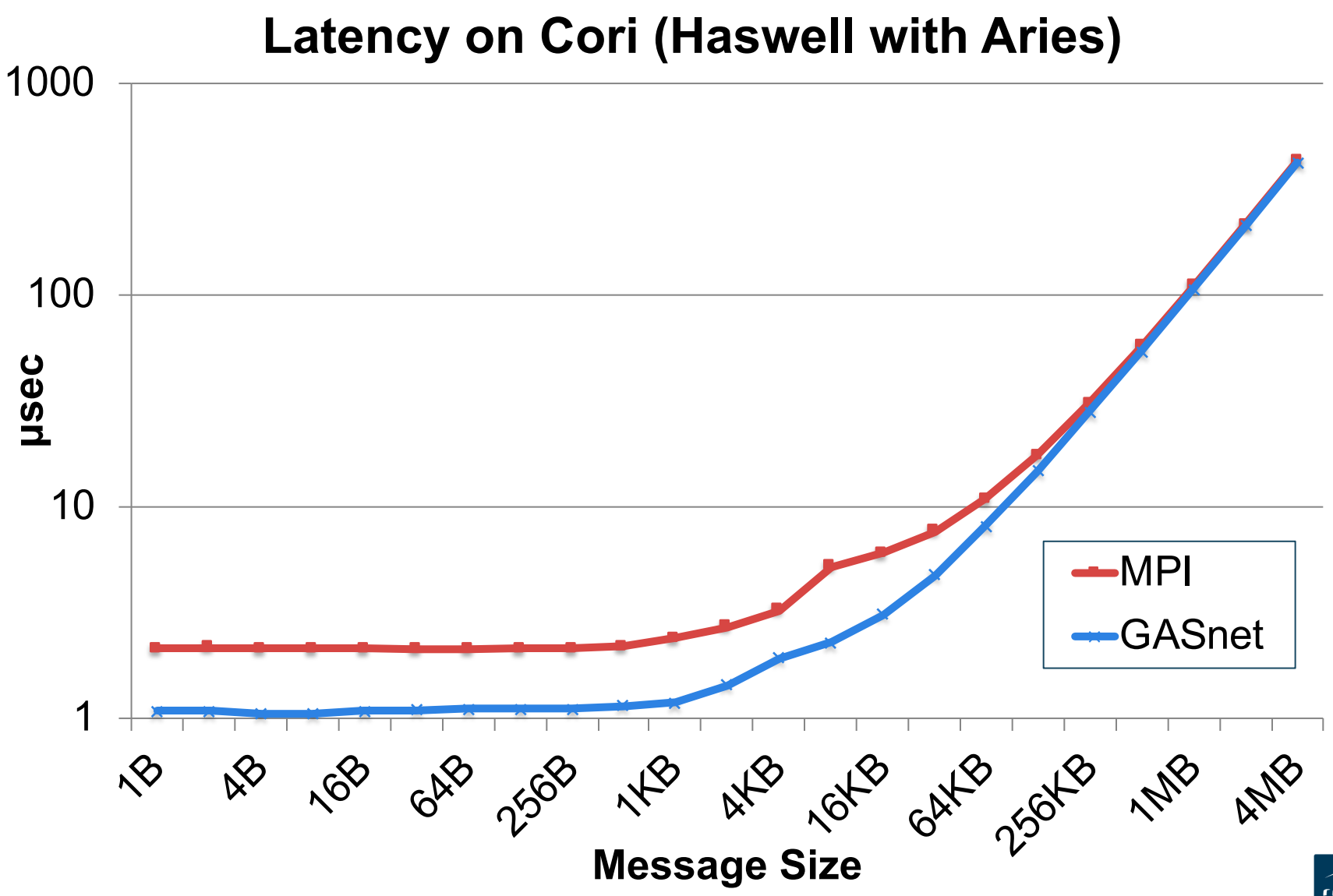
- Machines: expose best available performance on a given machine
  - Low latency for small messages
  - High bandwidth even for medium sized messages
  - High injection rate (number of messages / second)
  - Minimize software overhead and match hardware

One-Sided Communication in PGAS (e.g., GASnet inside)

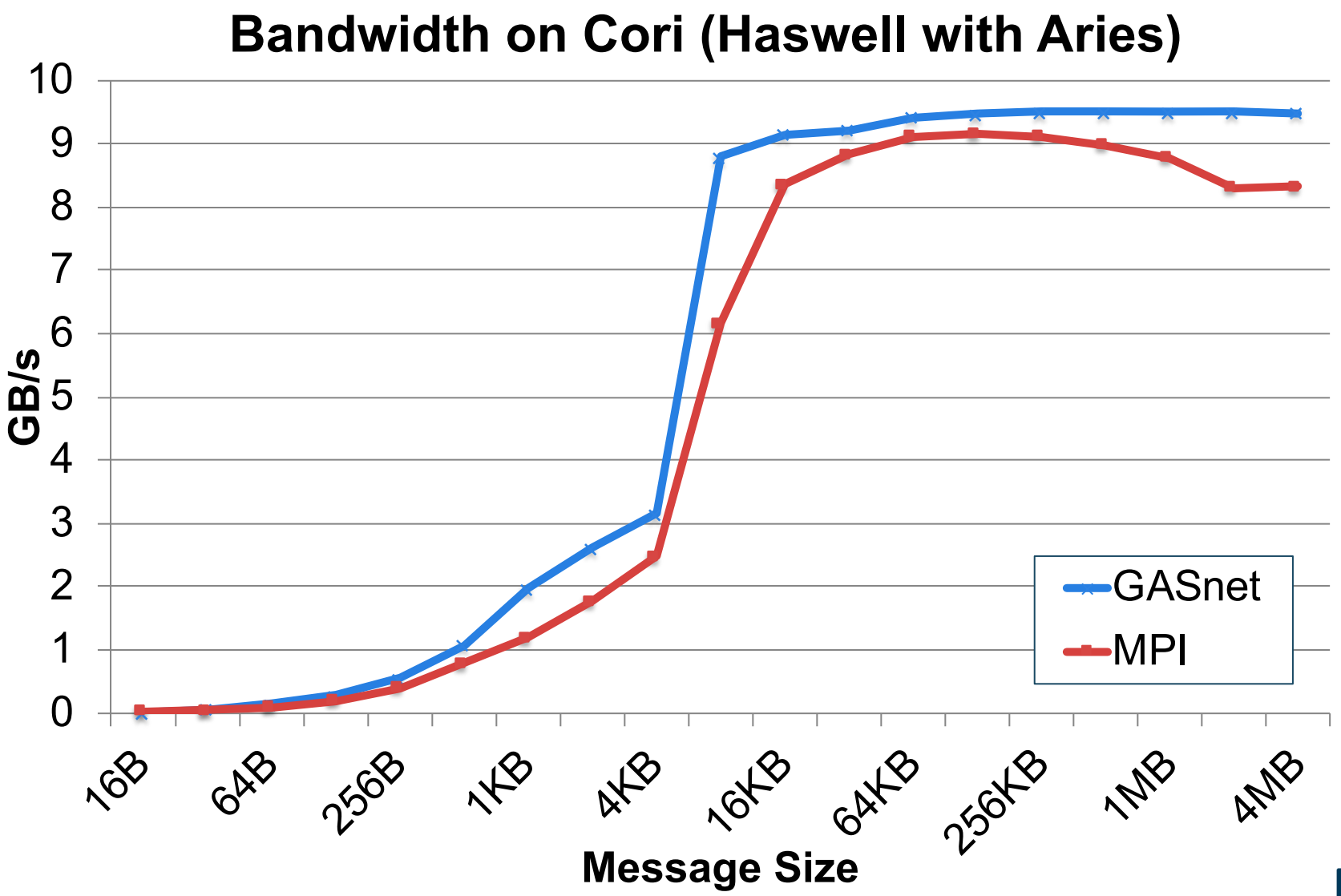


- A two-sided message needs to be matched with a receive
  - Ordering requirements can also hinder bandwidth
- A put/get can be handled by a network interface with RDMA
  - Decouples transfer from synchronization
  - Avoids interrupting the CPU or getting address from CPU
- Will still want to pack data and overlap
  - Pipeline messages, overlap with computation

Latency on a Cray Aries (NERSC Cori-P1)

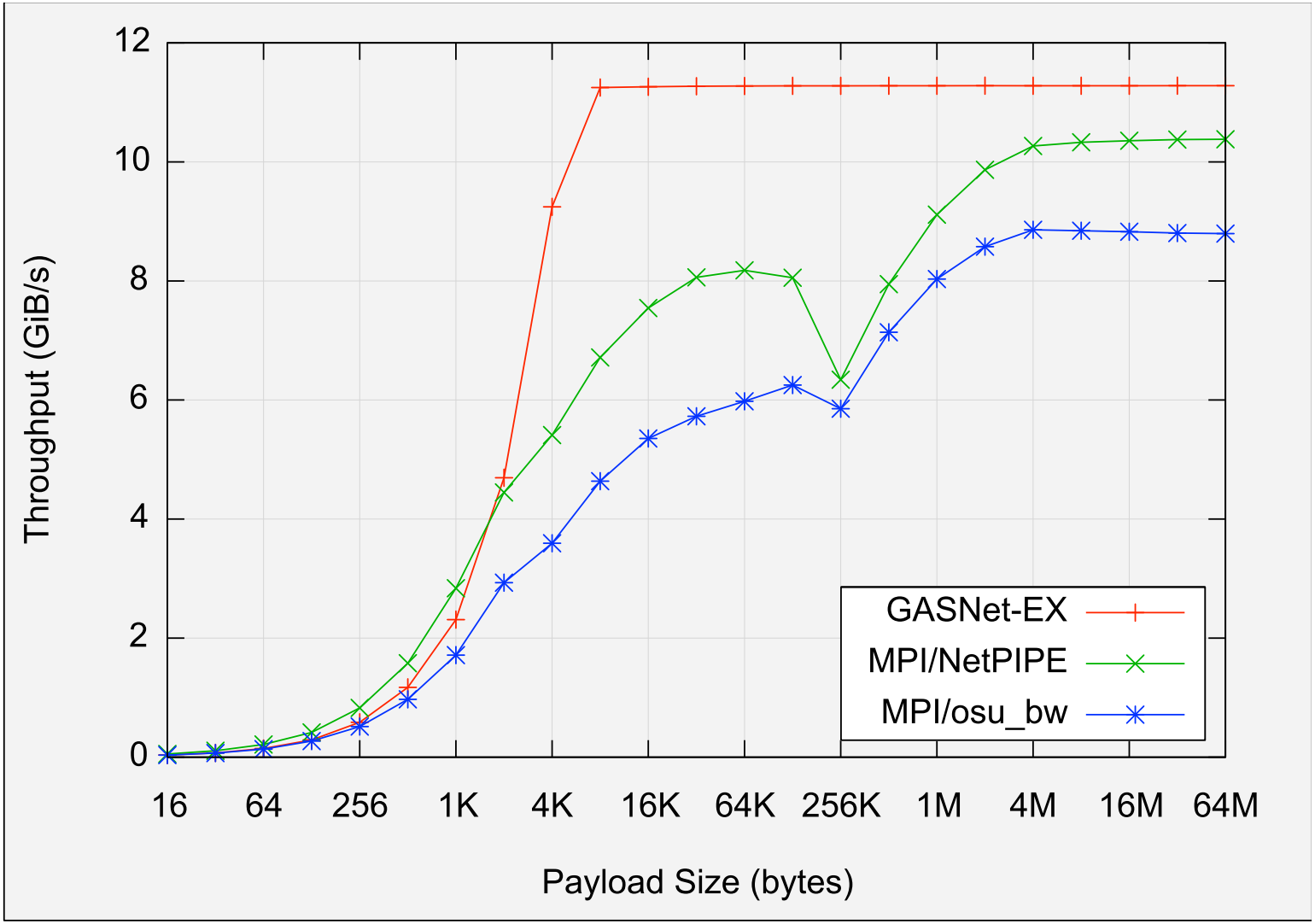


Bandwidth on a Cray Aries (NERSC Cori-P1)



BERKELEY LAB
Lawrence Berkeley National Laboratory

Bandwidth on Infiniband (Summit-Dev at ORNL)



Flood Bandwidth Graphs report tests of achievable one-way bandwidth for point-to-point data transfer between two nodes. All bandwidths have been converted for uniform reporting in units of Gibibytes/sec (GiB/sec), where GiB = 2^{30} bytes.

BERKELEY LAB
Lawrence Berkeley National Laboratory

Introduction to UPC++

Led by Scott B. Baden and Paul Hargrove (LBNL)
Slides and examples from Amir Kamil

See also the Programmer's Guide at:
https://bitbucket.org/upcxx/upcxx/downloads/upcxx-guide-2018.1.0.pdf

Hello World in UPC++

- UPC++ programs are just C++ programs
- They run as SPMD programs (similar to MPI)
- They use the GASNet library for communication

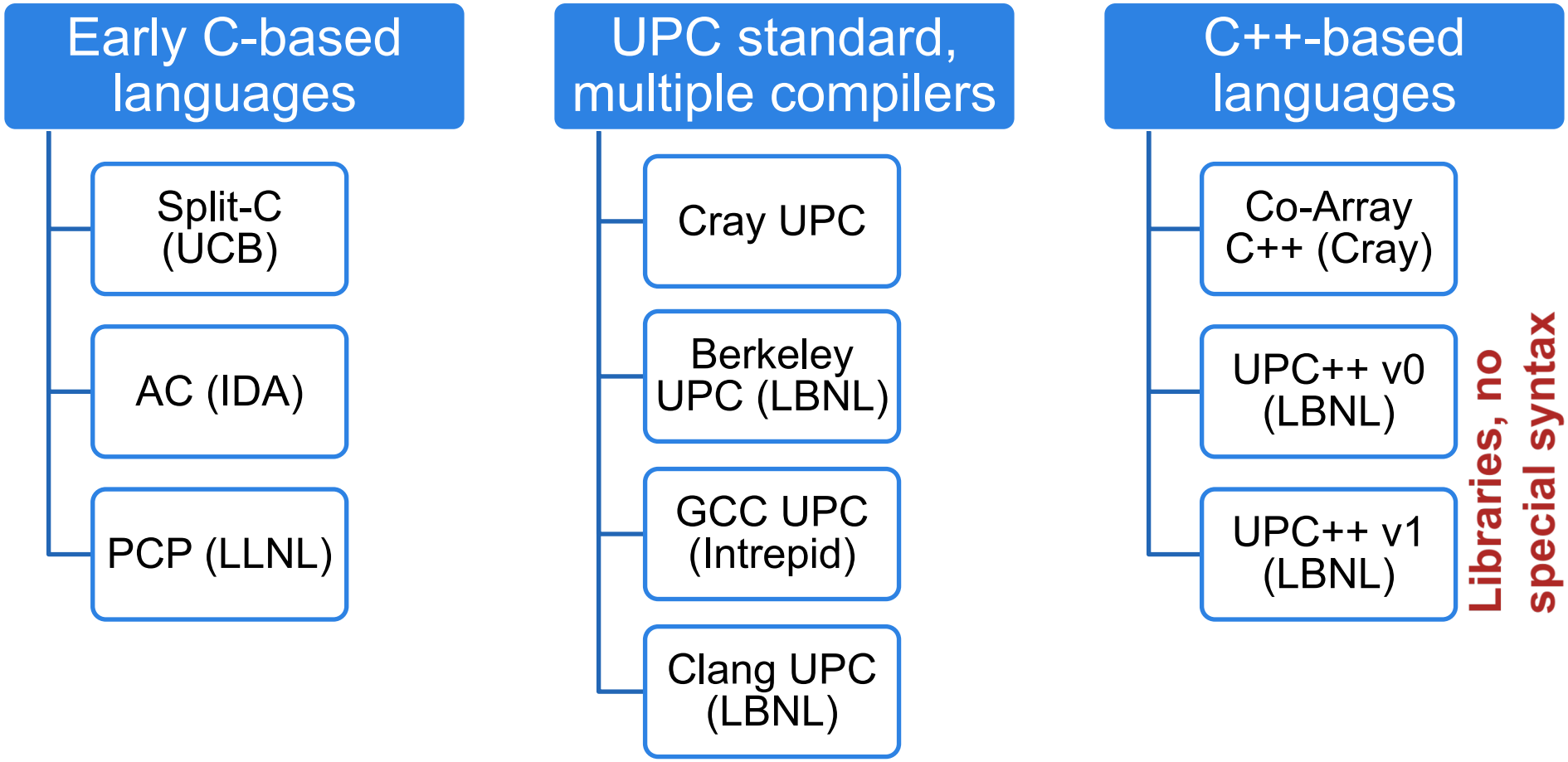
```
#include <upcxx/upcxx.hpp>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    upcxx::init(); // setup UPC++ runtime
    cout << "Hello from " << upcxx::rank_me() << endl;
    upcxx::finalize(); // close down UPC++ runtime
    return 0;
}
```

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

Roadmap of the C/C++ PGAS languages



There are also languages Co-Array Fortran (now in standard Fortran), Chapel, Titanium (Java), Legion and more

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

UPC++ V1.0 Overview

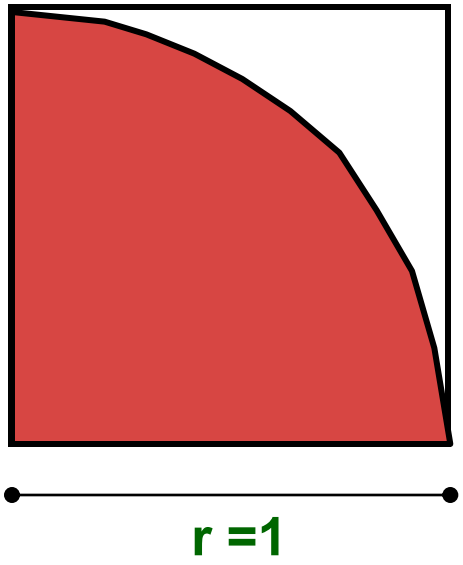
- A complete redesign of UPC++ that leverages GASNet-EX to deliver better performance and scalability
- A “compiler-free” approach for PGAS
  - Leverage C++ standards and compilers
  - Influence future directions of the C++ standard
- Interoperates with existing programming systems
  - 1-to-1 mapping between MPI rank and UPC++ rank
  - OpenMP and CUDA can be easily mixed with UPC++ in the same way as MPI+X
- Design philosophy:
  - All communication is explicit
  - Most operations are non-blocking
  - No non-scalable data structures

The API

- Foundational types
  - Global Pointers
  - Futures (and Promises)
  - Distributed Objects
- Communication
  - 1-sided Communication
    - rput/rget (bulk and single element), non-contiguous transfers, memory kinds
  - RPC (remote procedure call)
- Callbacks
- Remote Atomics
- Teams (mechanism for grouping ranks together)
- Progress and the Memory Model

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:

```
int main(int argc, char **argv) {
  upcxx::init();
  int hits, trials = 0;
  double pi;

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

  generator.seed(upcxx::rank_me()*17);

  for (int i=0; i < trials; i++) hits += hit();
  pi = 4.0*hits/trials;
  cout << "PI estimated to " << pi << endl;
  upcxx::finalize();
}
```

Each rank gets its own copy of these variables

Each rank can use input arguments

Initialize random in math library

Each rank calls “hit” separately

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

C++ Helper Code for Pi (in C++11)

- Required includes and variables:

```
#include <iostream>
#include <random>
#include <upcxx/upcxx.hpp>
default_random_engine generator;
uniform_real_distribution<> dist(0.0, 1.0);
```

- Function to throw dart and calculate where it hits:

```
int hit() {
  double x = dist(generator);
  double y = dist(generator);
  if (x*x + y*y <= 1.0) {
      return 1;
  } else {
      return 0;
  }
}
```

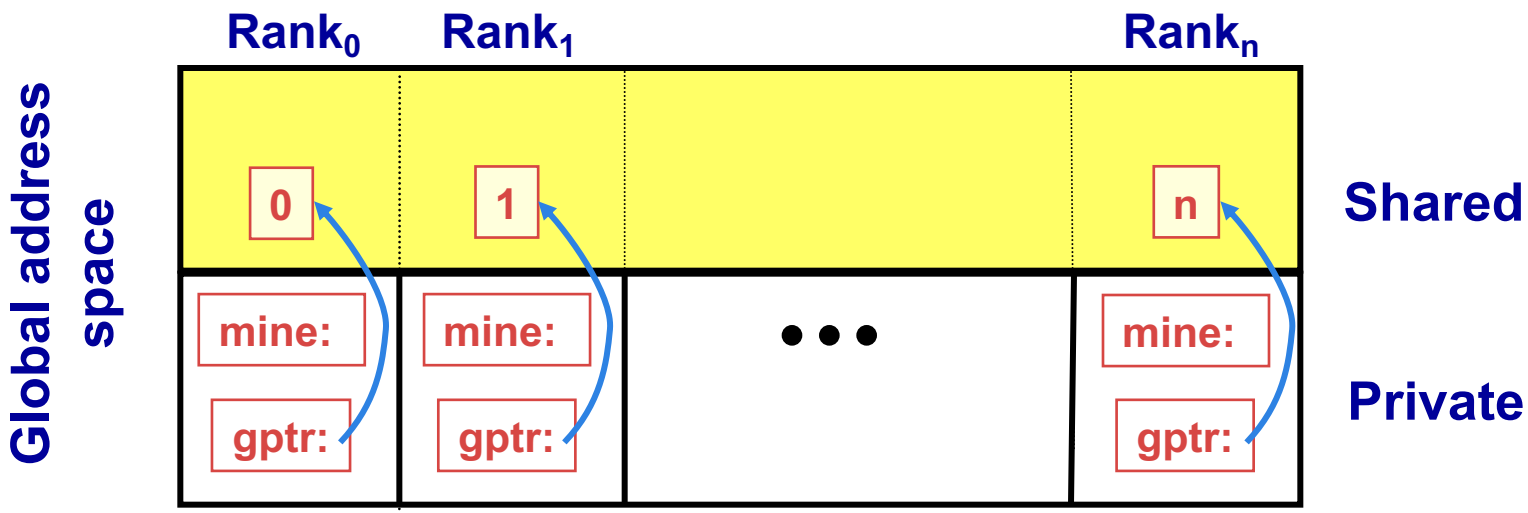
UPC++ allows full use of the C++ Standard Template Library

Private vs. Shared Memory in UPC++

- Normal C++ variables and objects are allocated in the private memory space for each thread
- Allocate in shared space with **new_**
- Free with **delete_**
- There are also array versions

upcxx:: qualifier elided from here on out
UPC++ names in green

```
int mine;
global_ptr<int> gptr = new_<int>(rank_me());
```



Broadcast in UPC++

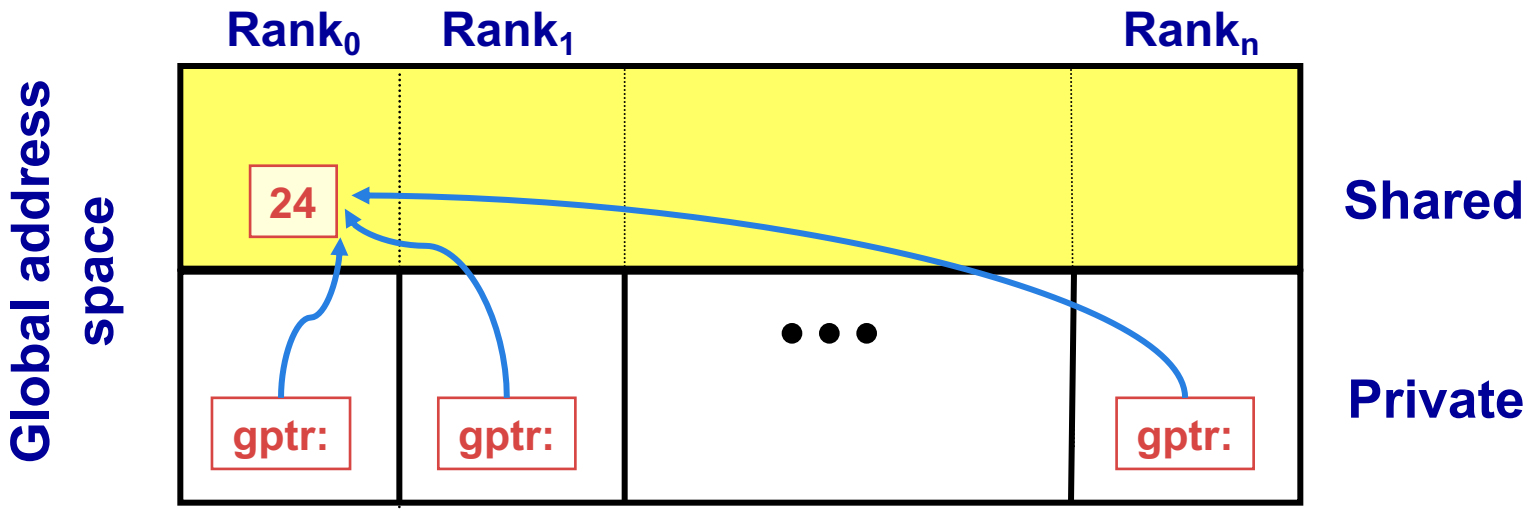
- To write an interesting program, we need to have global pointers refer to remote data
- One approach is to broadcast the pointer

```
global_ptr<int> gptr =
     broadcast(new_<int>(24),0).wait();
```

New int variable in shared space, set to 24

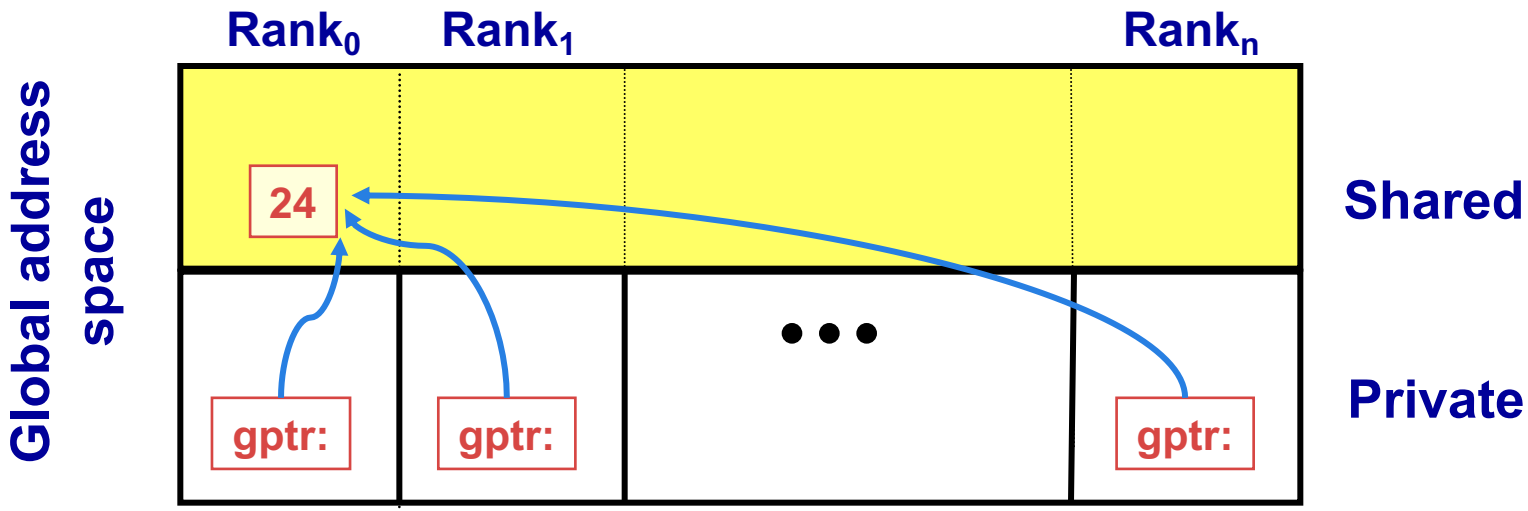
broadcast from rank 0

Will explain wait…



Remote access: Put / Get

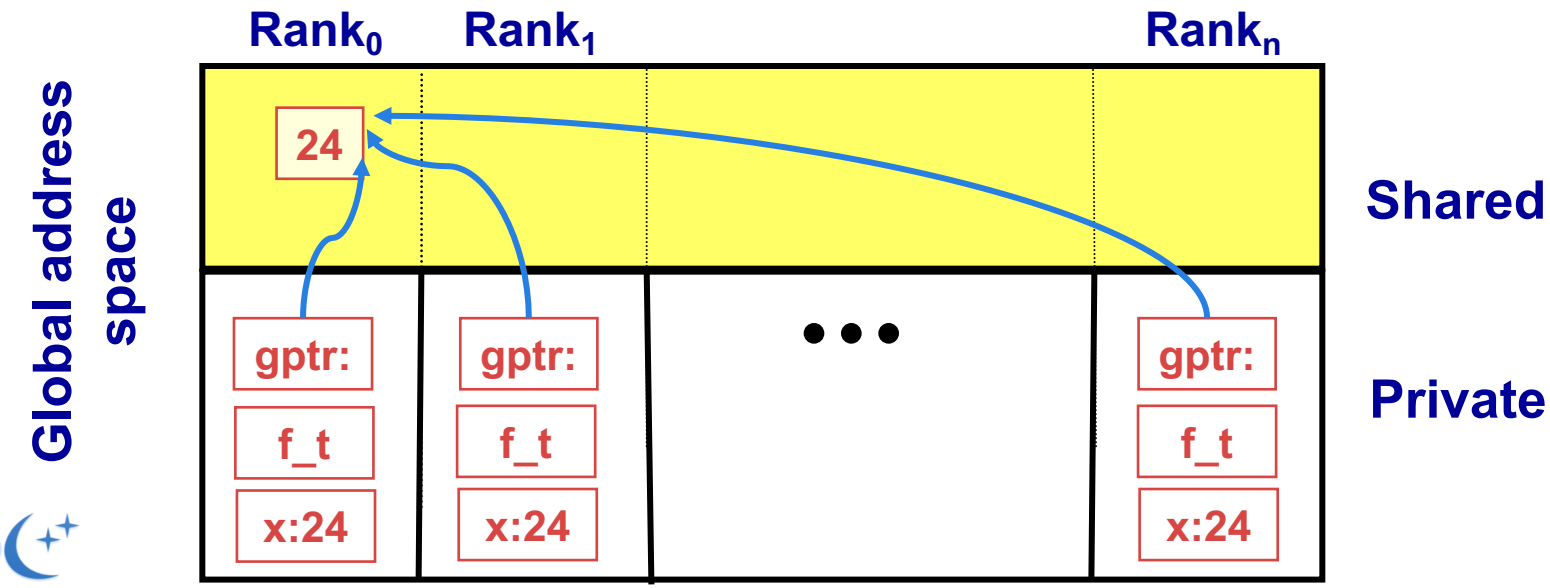
- We access shared memory from a remote rank
  - Remote get – read a variable on a remote rank
  - Remote put – write a variable on a remote rank
- Both done via a global pointer to the remote variable
- But will take microseconds at best!
  - And for most of that time, the processor is just waiting...



Asynchronous remote operations

- Asynchronous execution used to hide remote latency
  - Asynchronous get: start reading, but how to tell if you're done?
  - Put the results into a special "box" called a future

```
future<int> fut_temp = rget(gptr);
// ... Do something expensive
int x = fut_temp.wait();
// all ranks have x = 24
```



Futures in general

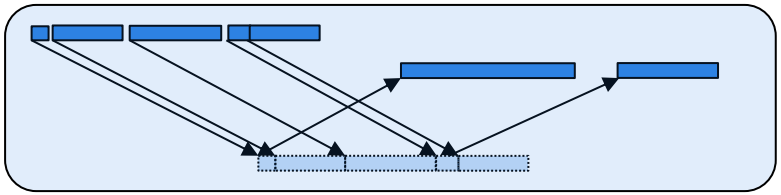
- A *future* holds a sequence of values and a state (ready / not ready)
- Waiting on the returned future lets user tailor degree of asynchrony they desire

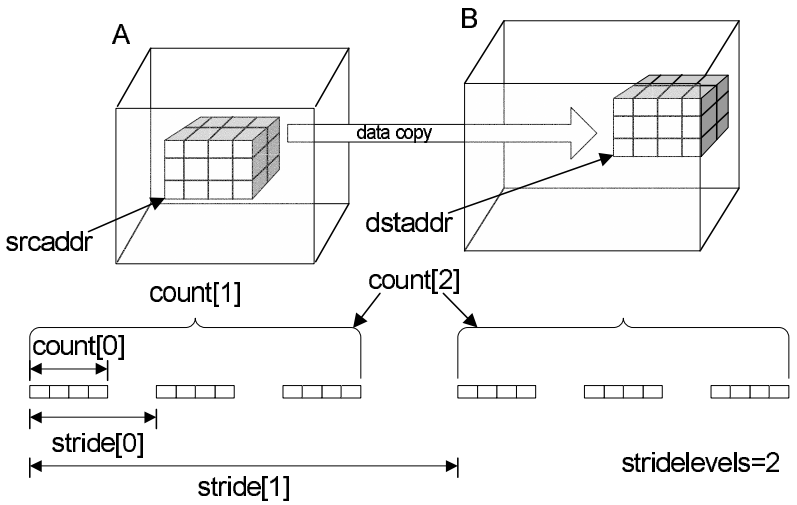
```
future<T> f1 = rget(gptr1);     // asynchronous op
future<T> f2 = rget(gptr2);
// unrelated work...
bool ready = f1.ready();       // non-blocking poll
T t = f1.wait();              // waits if not ready
```

- UPC++ has no *implicit* blocking
  - Except for synchronizing operations like "wait" others are implicitly nonblocking (asynchronous)

One-Sided Communication in General

- In addition to these basic remote put/get operations
  - There is also a version that runs a handler at the destination *after* rput operation is visible at the target
  - And support for non-contiguous transfers





UPC++ Synchronization

- UPC++ has two basic forms of barriers:
  - Barrier: block until all other threads arrive

```
barrier();
```

  - Asynchronous barriers

```
future<> f =
   barrier_async(); // this thread is ready for barrier
// do computation unrelated to barrier
wait(f);      // wait for others to be ready
```

- We will underline all blocking operations in these examples

Pi in UPC++: Shared Memory Style

- Parallel computing of pi, but with a bug

```
int main(int argc, char **argv) {
  init();
  int trials = atoi(argv[1]);
  int my_trials = (trials+rank_me())/rank_n();
  global_ptr<int> hits =
    broadcast(new_<int>(0), 0)).wait();
  generator.seed(rank_me()*17);
  for (int i=0; i < my_trials; i++) {
    int old_hits = rget(hits).wait();
    rput(old_hits+hit(), hits).wait();
  }
  barrier();
  if (rank_me() == 0)
    cout << "PI estimated to "
         << 4.0*(*hits.local())/trials;
  finalize();
}
```

divide work up evenly

broadcast pointer to shared memory from rank 0

accumulate hits

block on communication

Local (C++) pointer to global pointer that points locally

What is the problem with this program?

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

Atomics in UPC++

- Atomics are indivisible read-modify-write operations
- As if you put a lock around each operation, but may have hardware support (e.g., within the network interface)

Create "atomic domain" for shared ints that works with fetch-and-add and load operations.

```
atomic_domain<int> ad_int({atomic_op::load,
                           atomic_op::fetch_add});
```

Locally "throw darts"

```
int my_hits = 0;
for (int i=0; i < my_trials; i++) my_hits += hit();
```

Atomically update shared hits variable

```
ad_int.fetch_add(hits, my_hits,
                 memory_order_relaxed).wait();
```

C++ memory order: relaxed says other memory accessed not constrained

```
barrier();
if (rank_me() == 0)
  cout << "PI estimated to " << 4.0*ad_int.load(
                 hits, memory_order_relaxed).wait()
                 /trials;
```

Once a variable is used with atomics, always use atomics

BERKELEY LAB
Lawrence Berkeley National Laboratory

Remote Procedure Call

```
future<R> rpc(intrank_t r,
              F func, Args&&... args);
```

- Executes `func(args...)` on rank `r` and returns the result
- R is the return type of `func`
  - Empty future if func returns void
- There is also a 'fire and forget' version that returns no result
- Some restrictions apply to what UPC++ operations can be issued in an RPC: the *restricted context*
  - Limits on blocking operations from within an RPC

Pi in UPC++: RPC

- RPC used to synchronize updates

```
int hits = 0;
int main(int argc, char **argv) {
  init();
  int trials = atoi(argv[1]);
  int my_trials = (trials+rank_me())/rank_n();
  generator.seed(rank_me()*17);
  for (int i=0; i < my_trials; i++) {
    rpc(0, [](int hit) { hits += hit; },
            hit()).wait();
  }
  barrier();
  if (rank_me() == 0)
    cout << "PI estimated to " << 4.0*hits/trials;
  finalize();
}
```

RPC can refer to global variable

send update to rank 0
block on the update

Similar to atomics, but always runs on the remote processor (not in network) and can be arbitrary function (not just simple operation)

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

UPC++ Collectives

- UPC++ has a small set of collectives (so far)
- They are all asynchronous
  - Broadcast (value, sender): get a value from one rank

```
template <typename T> future <T>
  broadcast (T && value , intrank_t root);
```

  - Broadcast (buffer, count, sender): get a buffer of values from one rank

```
template <typename T> future <T>
  broadcast (T * buffer, std::size_t count,
             intrank_t sender);
```

  - Reduce (value, op): combine values across ranks

```
template <typename T, typename BinaryOp>
future <T> reduce_all (T && value, BinaryOp &&op);
```

- All of these take an optional “team” of ranks and have more general completion semantics (details in UPC++ guide)

Pi in UPC++: Data Parallel Style w/ Collectives

- The previous version of Pi works, but is not scalable:
  - Updates are serialized on rank 0, ranks block on updates
- Use a reduction for better scalability:

```
// int hits;
int main(int argc, char **argv) {
  ...
  for (int i=0; i < my_trials; i++)
     my_hits += hit();
  int hits = reduce_all(my_hits,
                        op_fast_add).wait();
  // barrier();
  if (rank_me() == 0)
    cout << "PI: " << 4.0*hits/trials;
  finalize();
}
```

no global variables or shared memory

barrier implied by reduce +wait

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

Distributed Objects

- Any C++ type can be made into a *distributed object*
- One instance on every rank of a team

Mesh is a 3D array

```
dist_object<int> counter(0); // over world team
```

  - Collectively allocate a set of shared counters
- Can access remote instances

```
future<int> f = fetch(counter, someRank);
int x = f.wait();
```

Pi in UPC++: Distributed Object Version

- Alternative fix to the race condition
- Have each rank update a separate counter:
  - Do it in a distributed object, have one rank compute sum

```
int main(int argc, char **argv) {
  … declarations and initialization code omitted
  dist_object<int> all_hits(0);
  for (int i=0; i < my_trials; i++)
    *all_hits += hit();
  barrier();
  if (rank_me() == 0) {
    for (int i=0; i < rank_n(); i++)
      hits += fetch(all_hits, i).wait();
    cout << "PI estimated to " << 4.0*hits/trials;
  }
  finalize();
}
```

all_hits distributed across all ranks

update element with local affinity

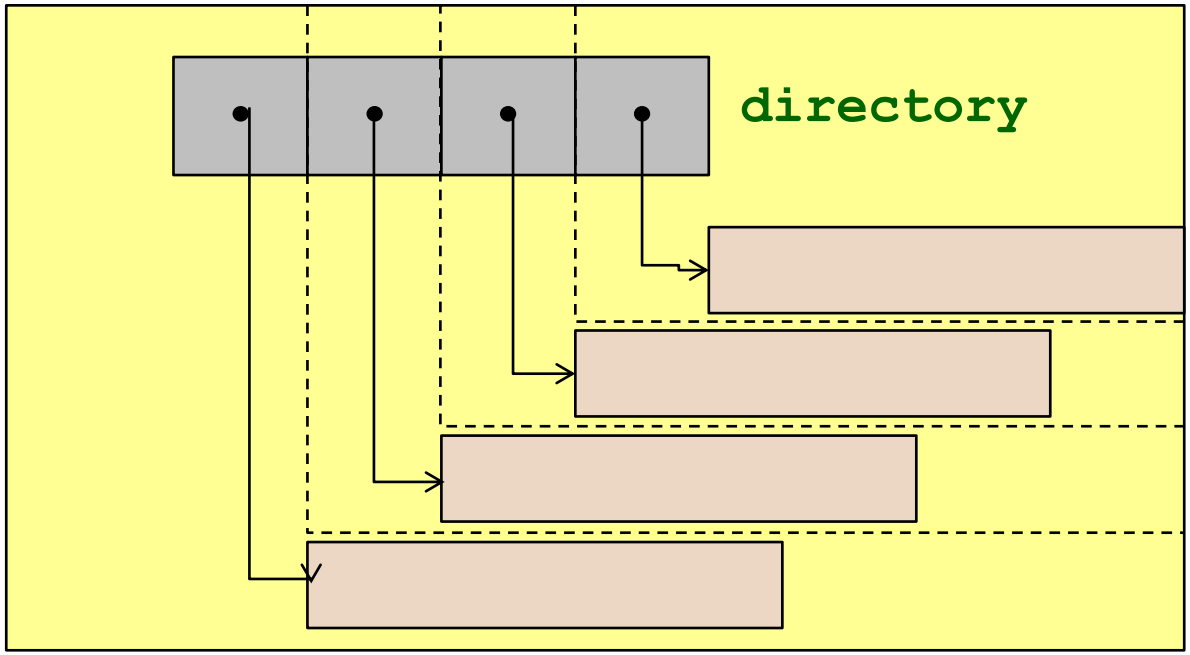
collect each rank's contribution

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

Distributed Arrays Directory Style

- A distributed object can be much more complicated than a set of ints
- Many UPC++ programs use these kinds of “directories” for distributed data structures



Distributed Objects in Stencil Code

- Communication in 1D stencil (nearest-neighbor computation):

```
int main(int argc, char **argv) {
  … declarations and initialization code omitted
  global_ptr<double> my_grid =
    new_array<int>(interior+2);
  dist_object<global_ptr<double>> grids(my_grid);
  global_ptr<double> left =
    fetch(grids, (rank_me()+rank_n()-1)%rank_n()).wait();
  global_ptr<double> right =
    fetch(grids, (rank_me()+1)%rank_n()).wait();
  for (int i=0; i < timesteps; i++) {
    future<double> f1 = rget(left+interior);
    future<double> f2 = rget(right+1);
    … wait on futures and do computation
  }
  …
}
```

construct local grids and distributed object

get pointers to neighbors' grids

get ghost cells

Summary

- UPC++ is a PGAS library that supports lightweight communication over GASNet-EX
- Close to the metal performance, lean interface
  - Trade offs to reduce overheads and increase flexibility
    - Asynchronous and explicit communication
    - Reduced consistency guarantees
- Advanced features not covered in talk:
  - Promises, callbacks, remote atomics, progress, memory model, teams
- V1.0 release targeted for September 30, 2017
  - Will include programmer's guide

upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

Acknowledgements

- Early work with UPC++ involved Yili Zheng, Amir Kamil, Kathy Yelick, and others [IPDPS ‘14]
- Pagoda Project (GASNet-EX and UPC++): Scott B. Baden (PI), Paul Hargrove (co-PI), John Bachan, Dan Bonachea, Steven Hofmeyer, Khaled Ibrahim, Mathias Jacquelin, Amir Kamil, Brian van Straalen
- This research was supported by the Exascale Computing Project (17-SC-20-SC), funded by the U.S. Department of Energy

- UPC++ V1.0 draft specification available at
  https://bitbucket.org/upcxx/upcxx

PGAS Applications

UPC and old UPC++

Others (not shown) in Co-Array Fortran (Ocean model),Titanium (heart model) and Chapel

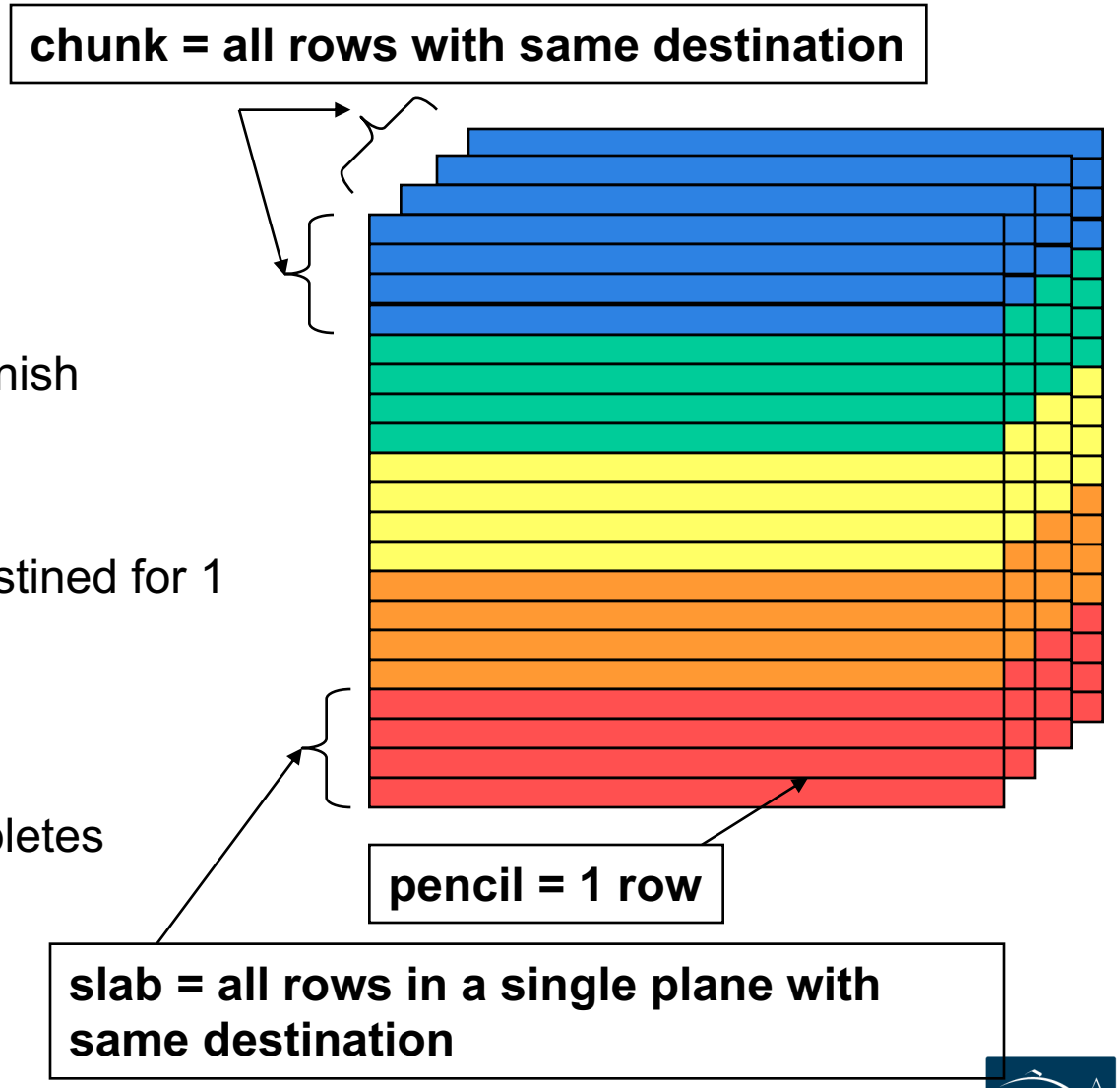
Some of these started as CS267 projects

- Project ideas for this year:
- Contribute to the Berkeley Container Library
  - Data structure library (hash tables, bloom filters, matrices,… -- talk to Ben Brock)
- Compare UPC++ to something else
  - MPI, Ray, …
- Write a more substantial algorithm in UPC++

Application Challenge: Fast All-to-All

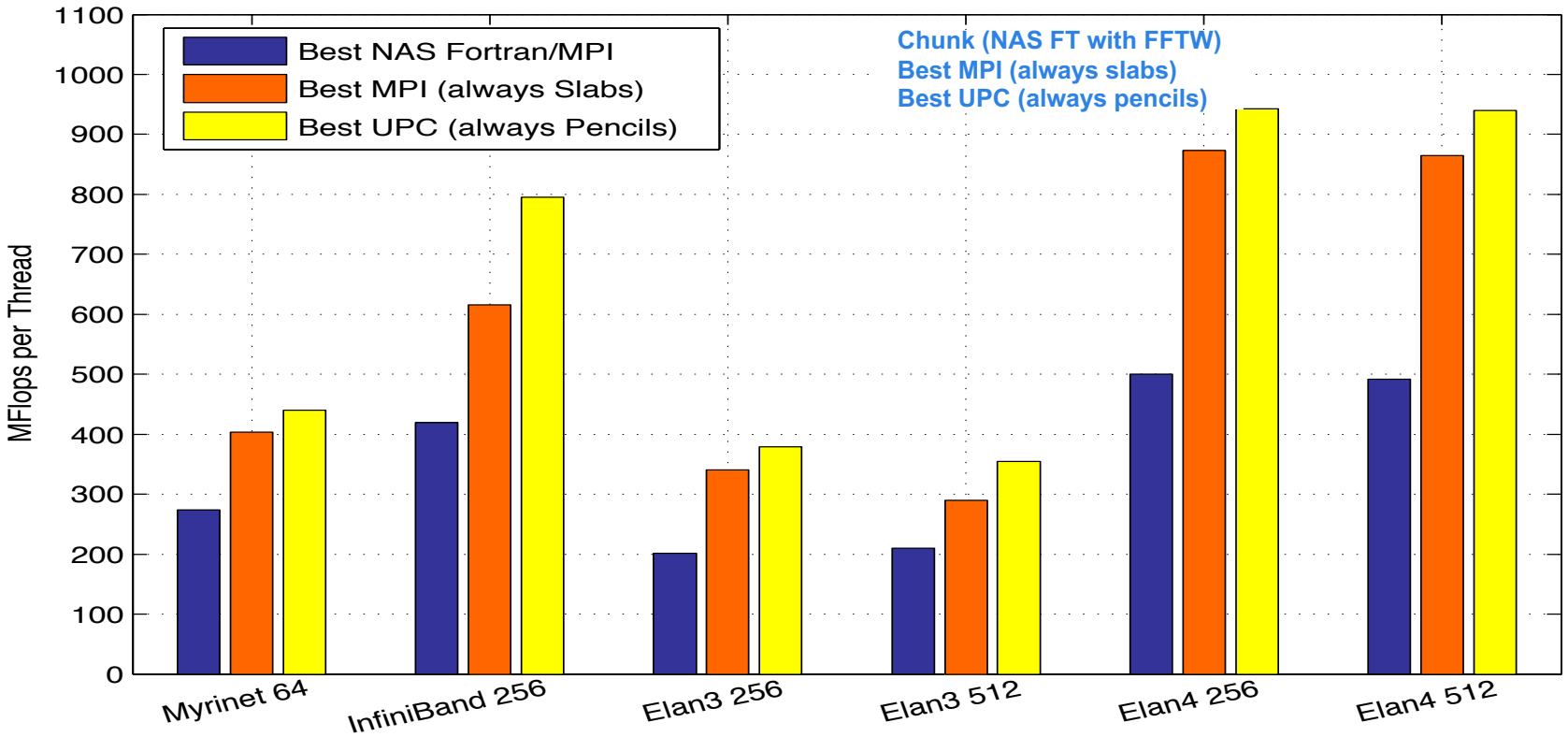
Transpose in 3D FFT

- 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

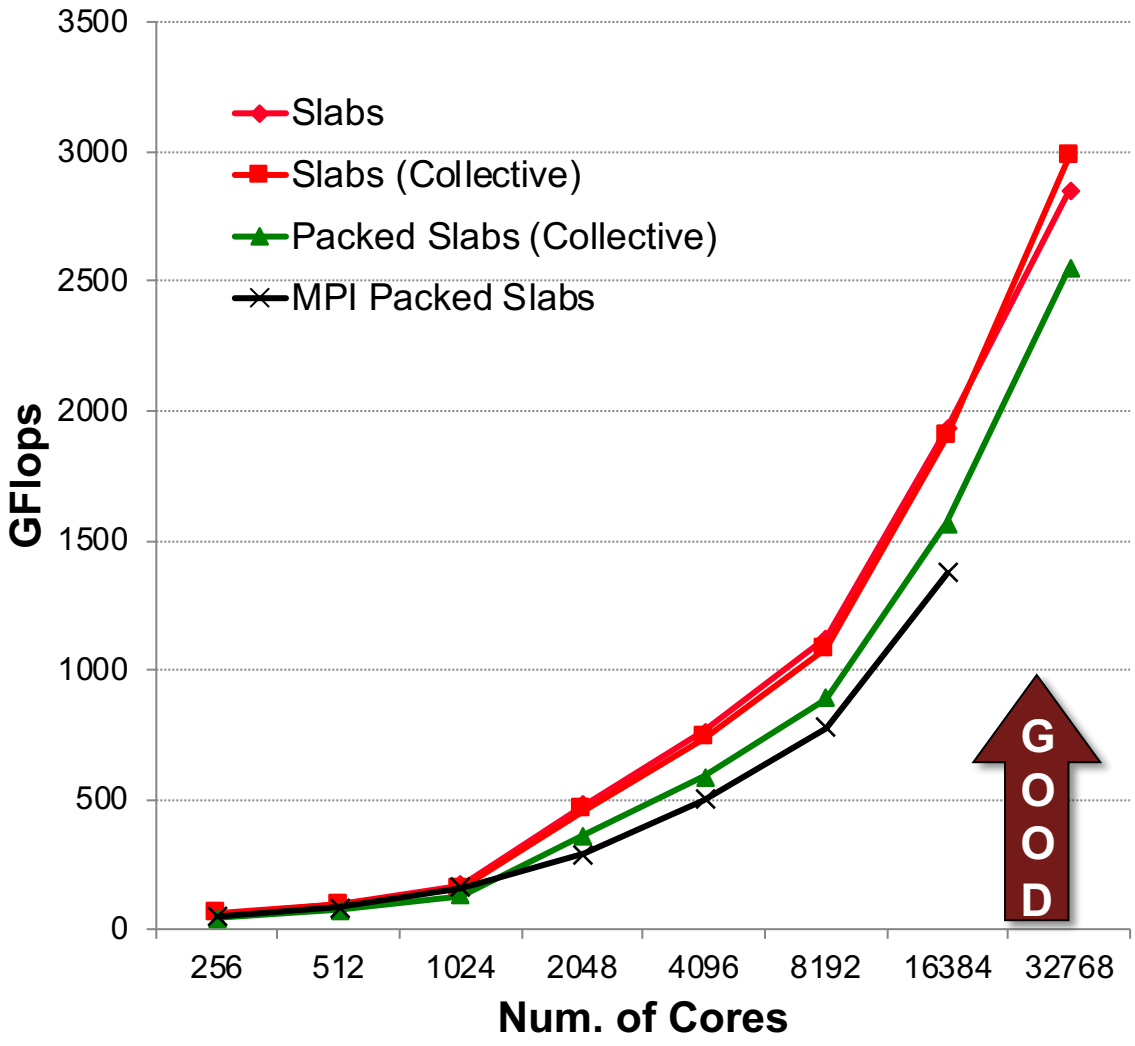
Bisection Bandwidth



- 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



BERKELEY LAB
Lawrence Berkeley National Laboratory

De novo Genome Assembly

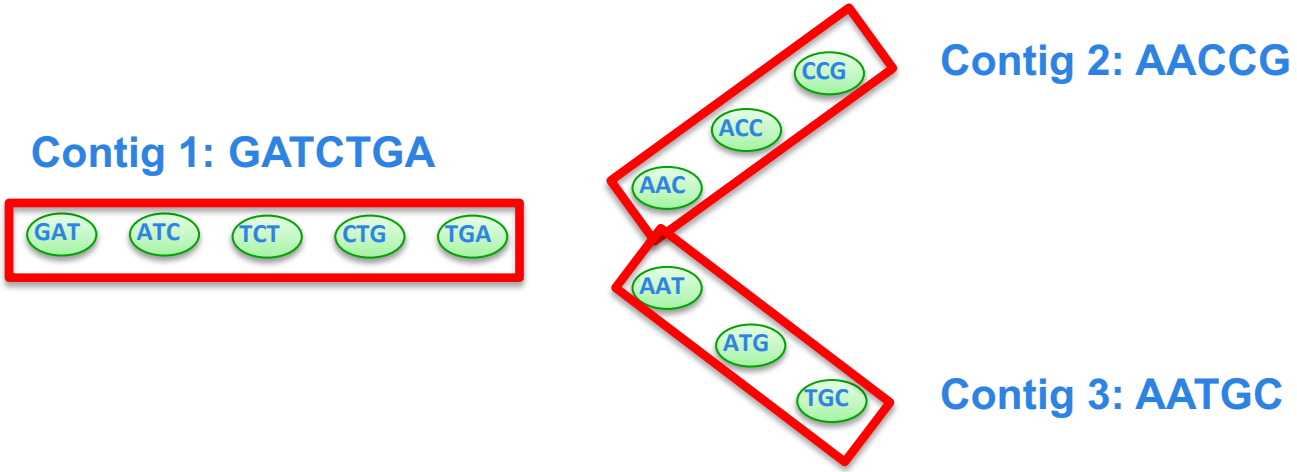
- **DNA sequence consists of 4 bases**: A/C/G/T
- **Read**: short fragment of DNA
- **De novo assembly: Construct a** genome (chromosomes) from a collection of reads

DEGAS

BERKELEY LAB
Lawrence Berkeley National Laboratory

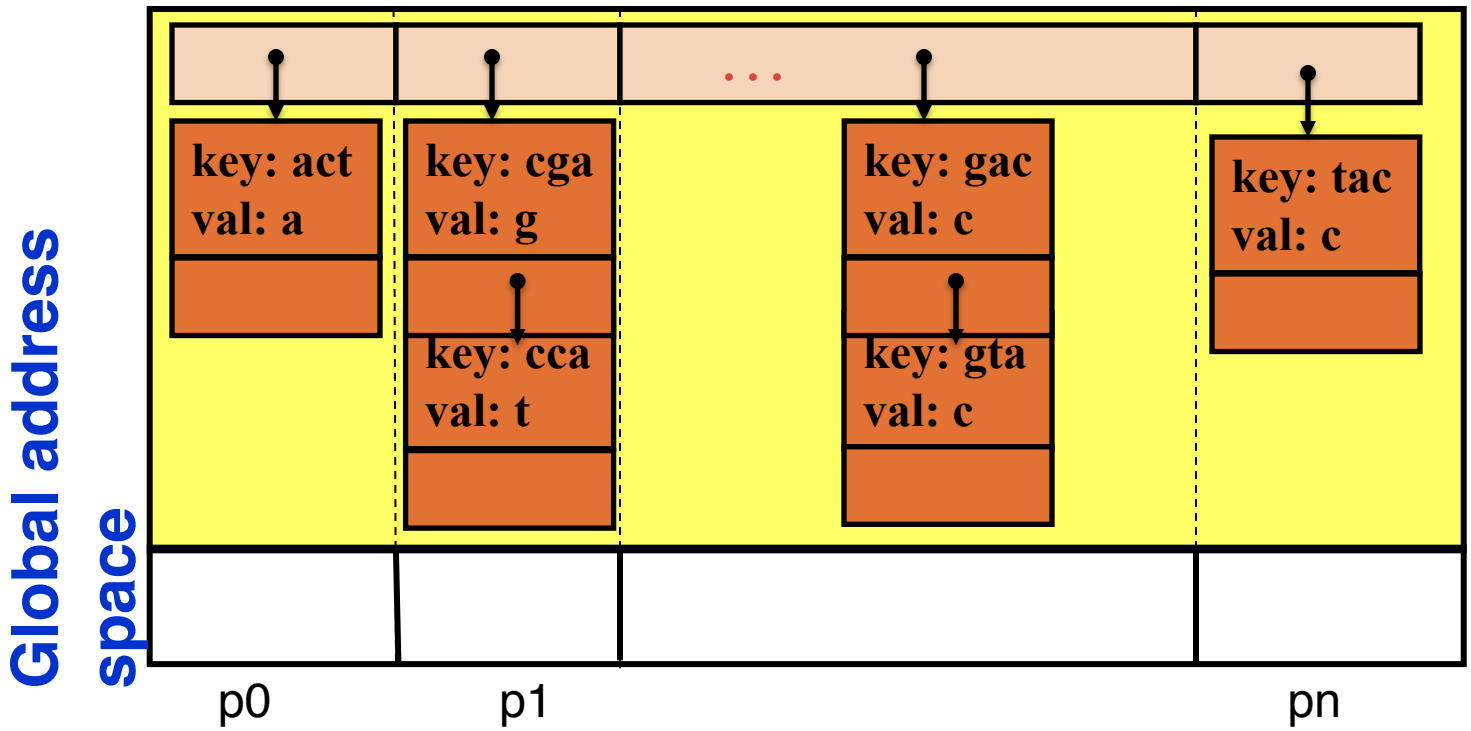
PGAS in Genome Assembly

- 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

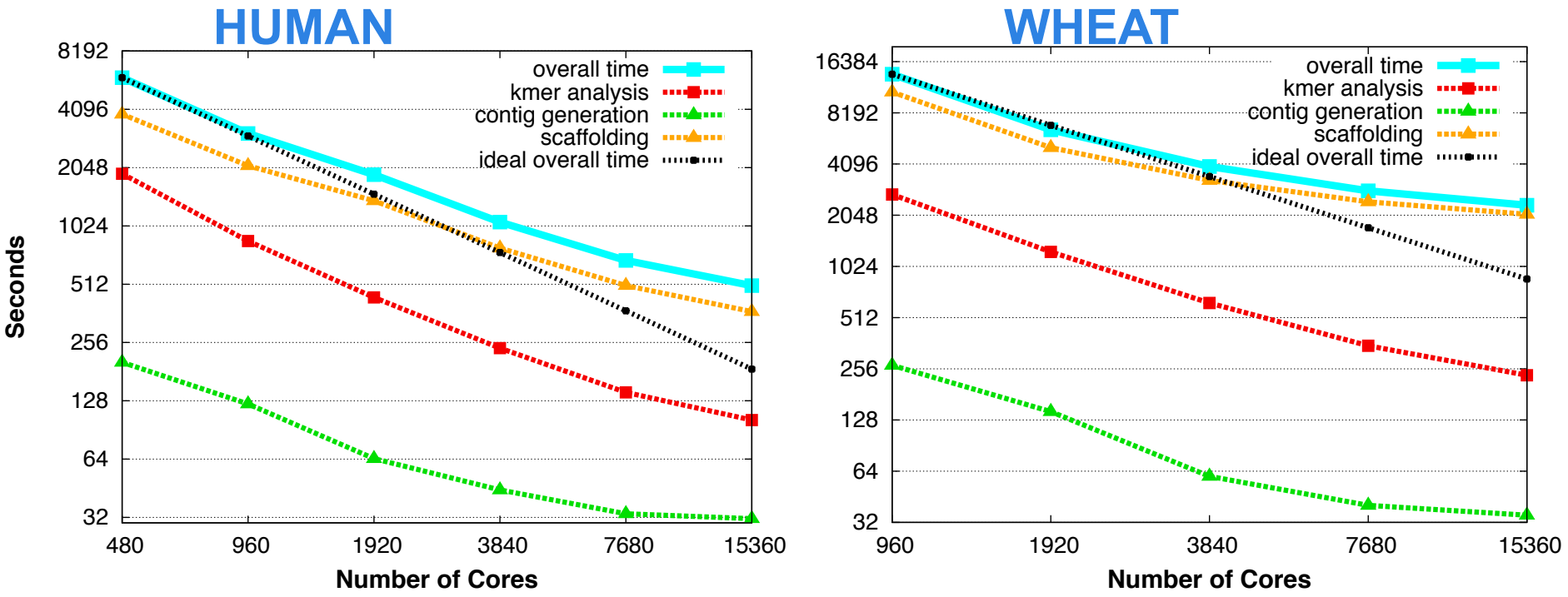
DEGAS

BERKELEY LAB
Lawrence Berkeley National Laboratory

HipMer (High Performance Meraculous) Assembly Pipeline

Distributed Hash Tables in PGAS

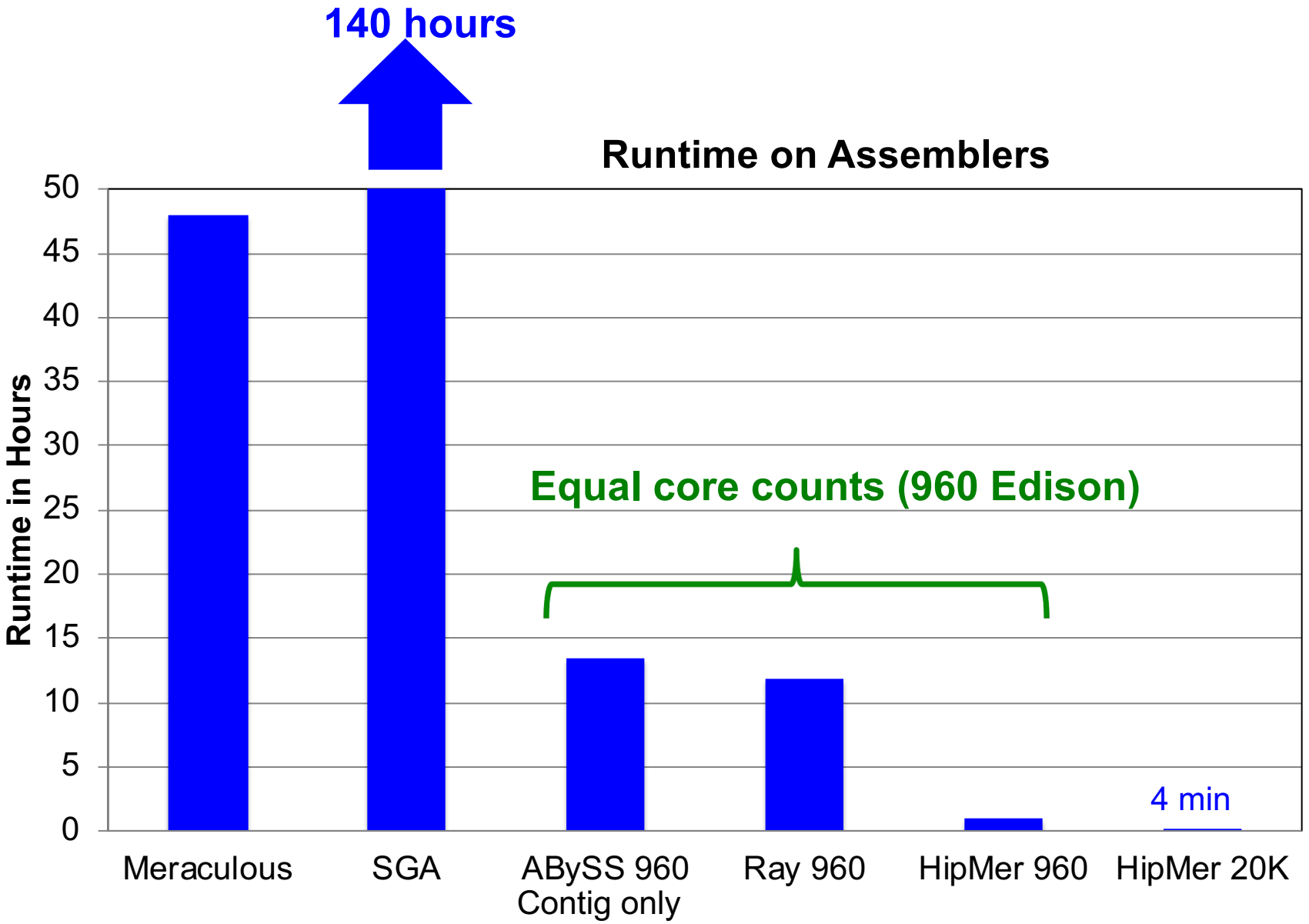
- Remote Atomics, Dynamic Aggregation, Software Caching
- 13x Faster than MPI code (Ray) on 960 cores



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

BERKELEY LAB
Lawrence Berkeley National Laboratory

Comparison to other Assemblers



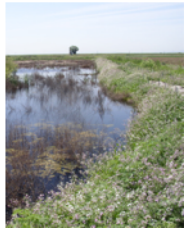
Science Impact: HipMer is transformative

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

Makes unsolvable problems solvable!

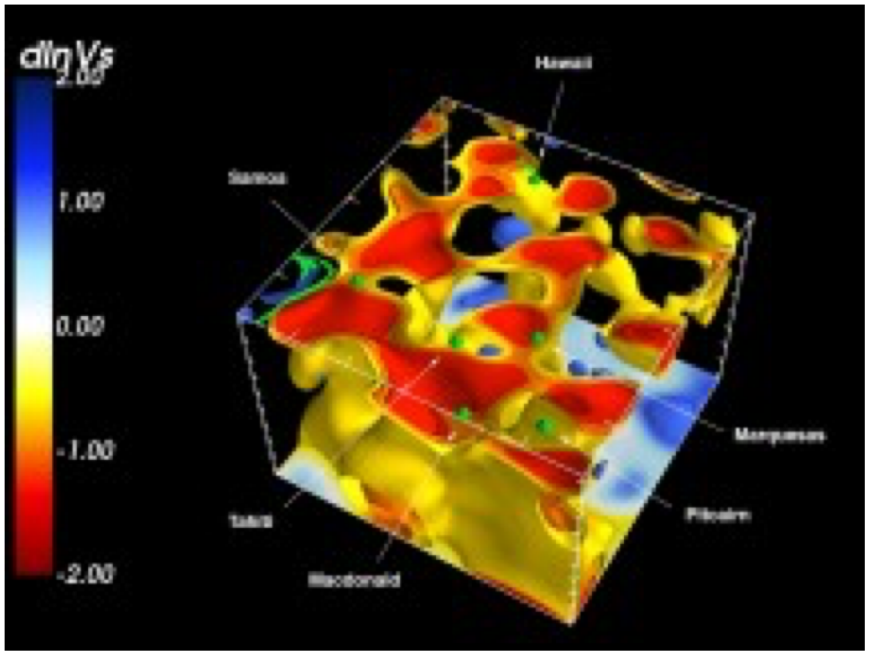
- Wheat genome (17 Gbp) “de novo” assembled (2014):
  - Meraculous (did not run):
  - HipMer: 39 minutes; 15K cores (first all-in-one assembly)

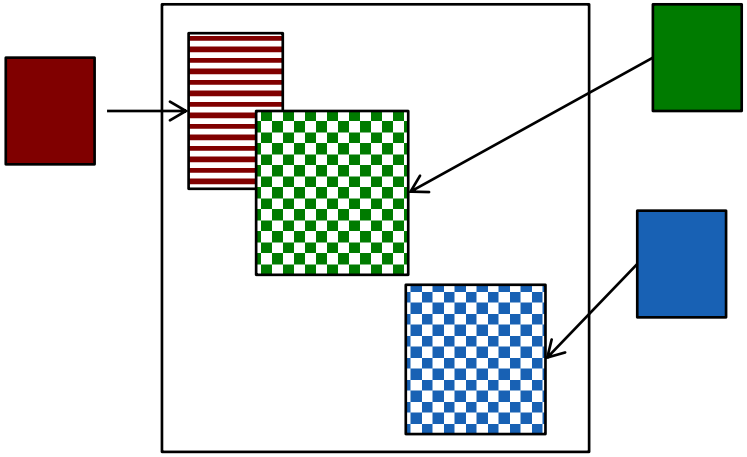
- Pine genome (20 Gbp) “de novo” assembled (2014) :
  - Masurca : 3 months; 1 TB RAM
- Wetland metagenome (1.25 Tbp) analysis (2015):
  - Meraculous (projected): 15 TB of memory
  - HipMER: Strong scaling to over 100K cores (contig gen only)



Application Challenge: Data Fusion in UPC++

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





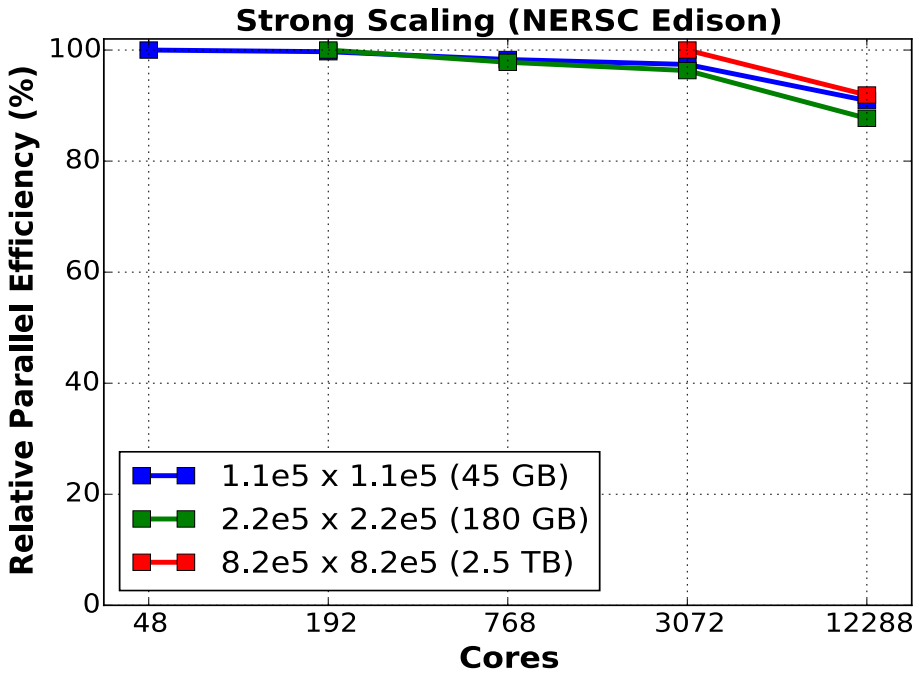
First ever sharp, three-dimensional scan of Earth's interior that conclusively connects plumes of hot rock rising through the mantle with surface hotspots that generate volcanic island chains like Hawaii, Samoa and Iceland.

DEGAS

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).

BERKELEY LAB
Lawrence Berkeley National Laboratory

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.

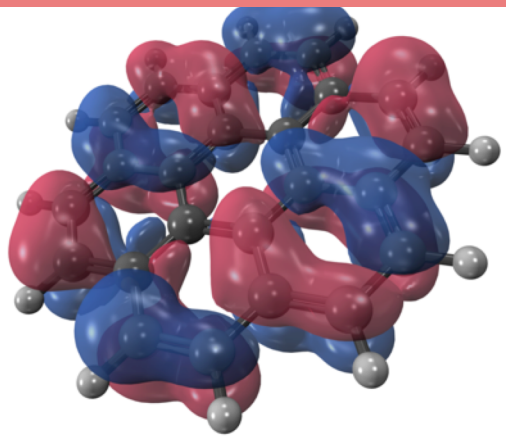
DEGAS

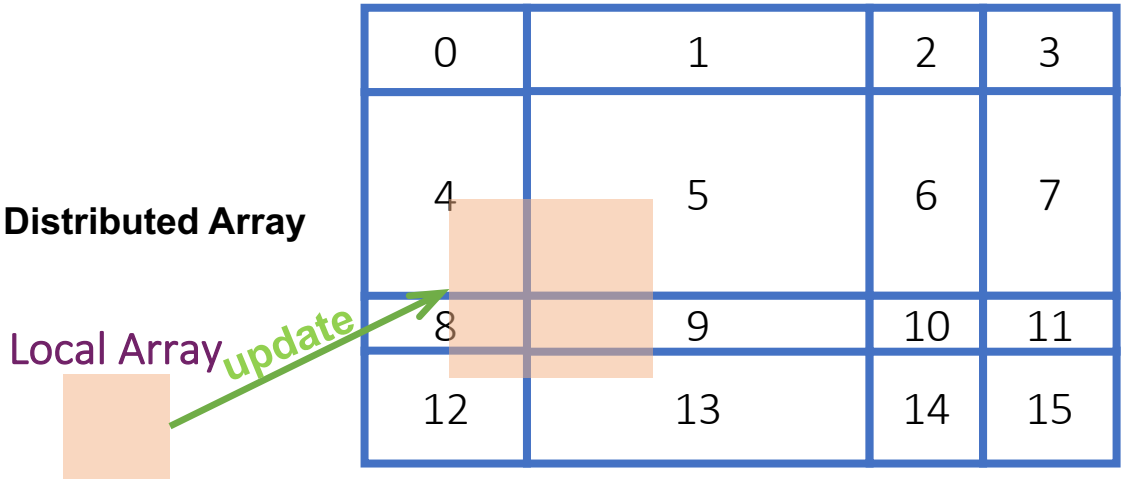
BERKELEY LAB

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)*

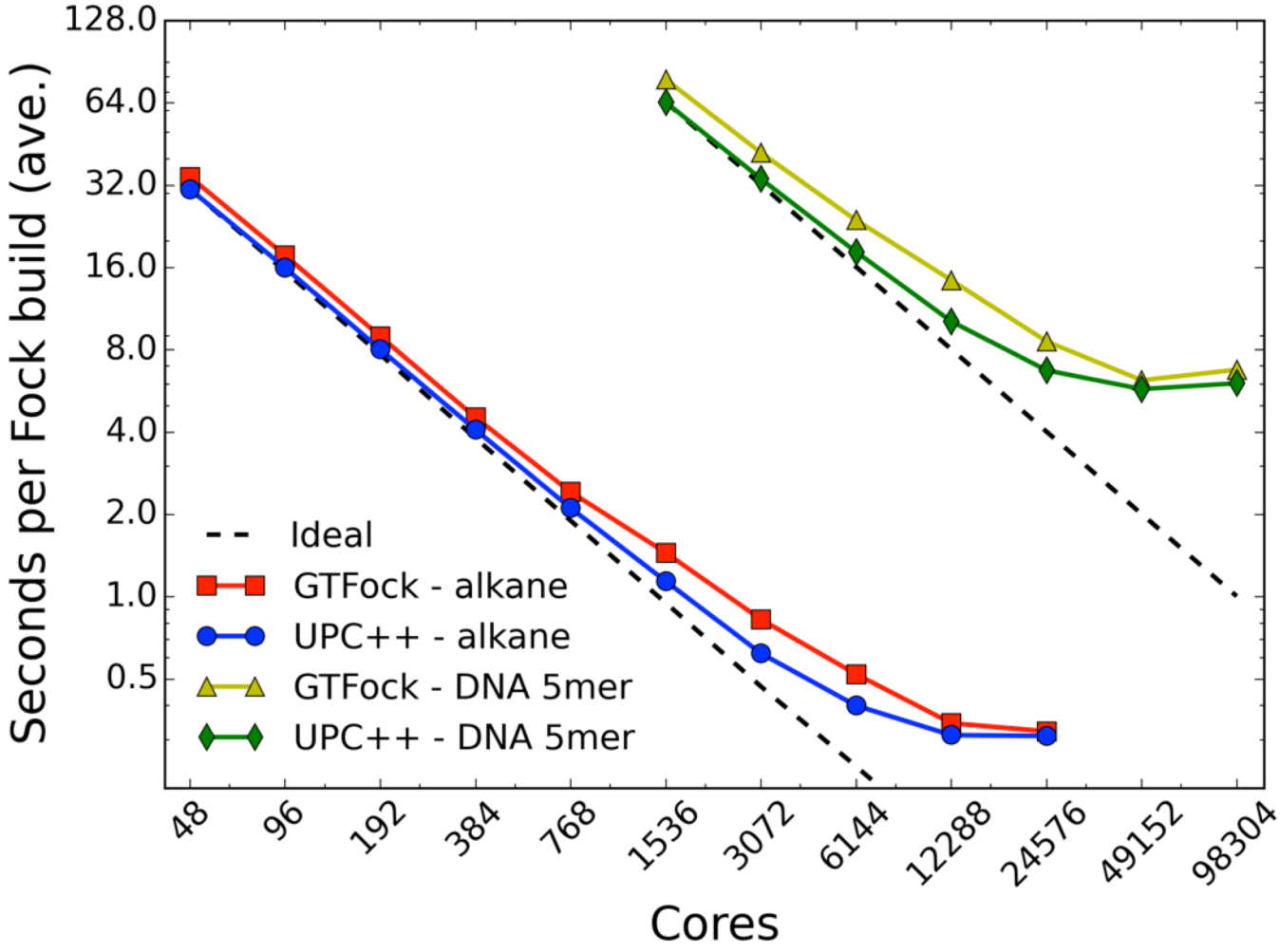
Increase scalability!





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

Hartree Fock Code in UPC++

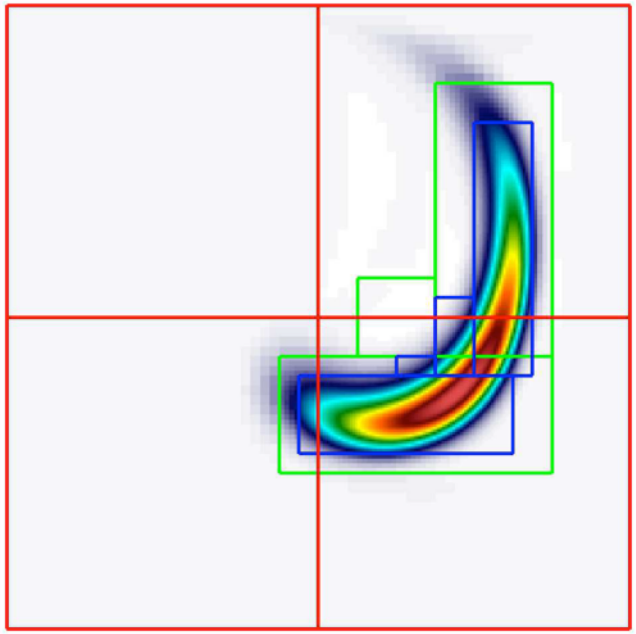


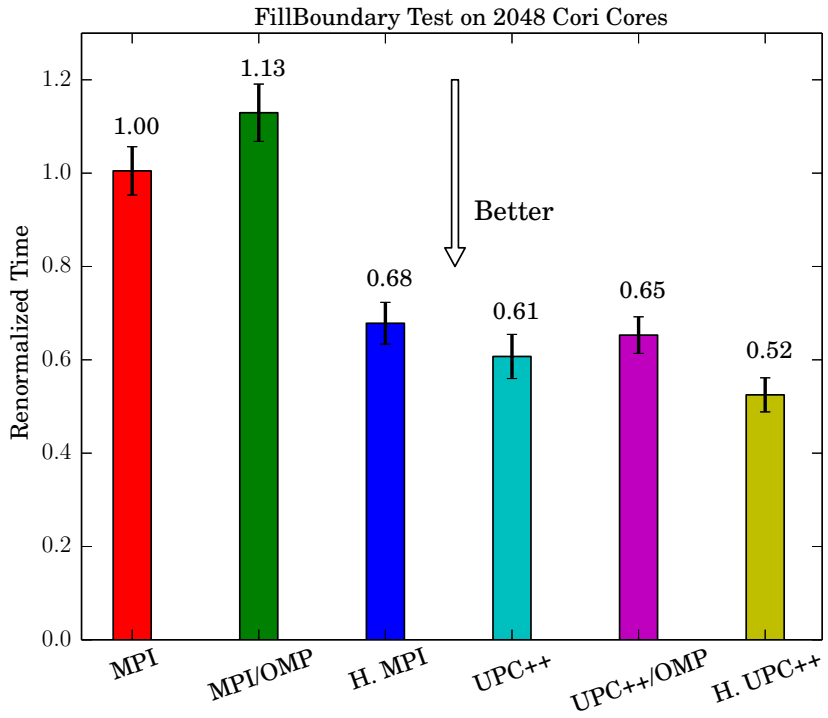
Strong Scaling of UPC++ HF Compared to GTFock with Global Arrays on NERSC Edison (Cray XC30)

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

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

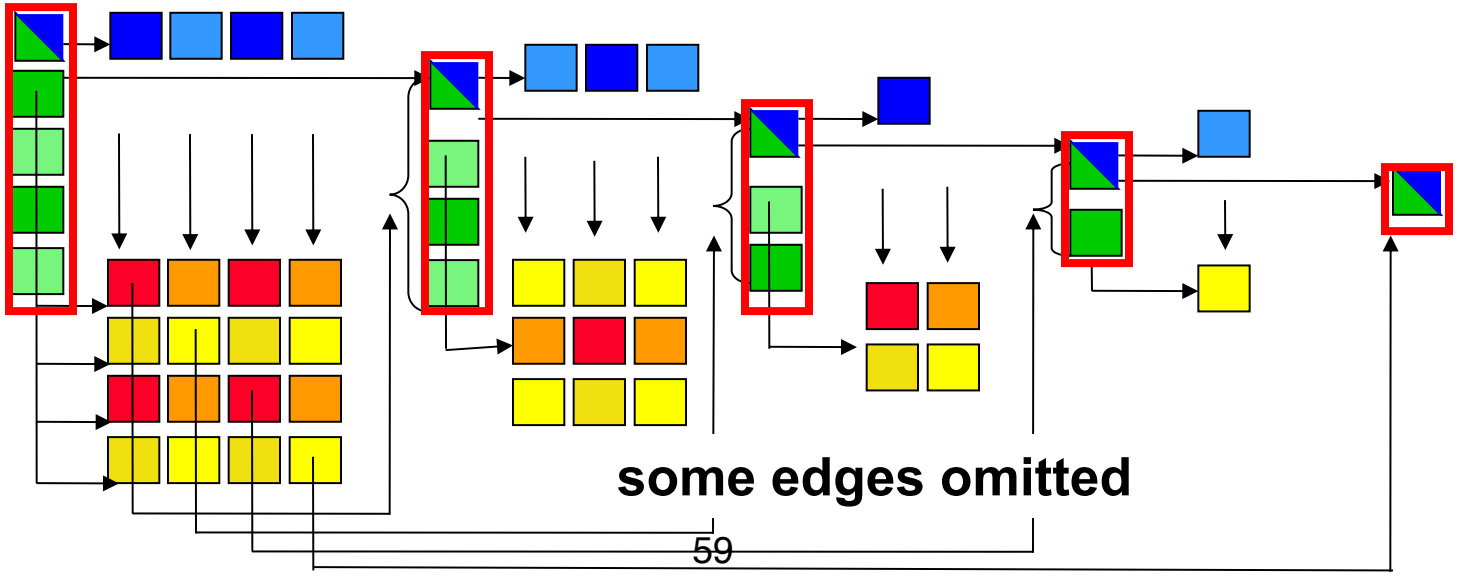
upc++

BERKELEY LAB
Lawrence Berkeley National Laboratory

Beyond Put/Get: Event-Driven Execution

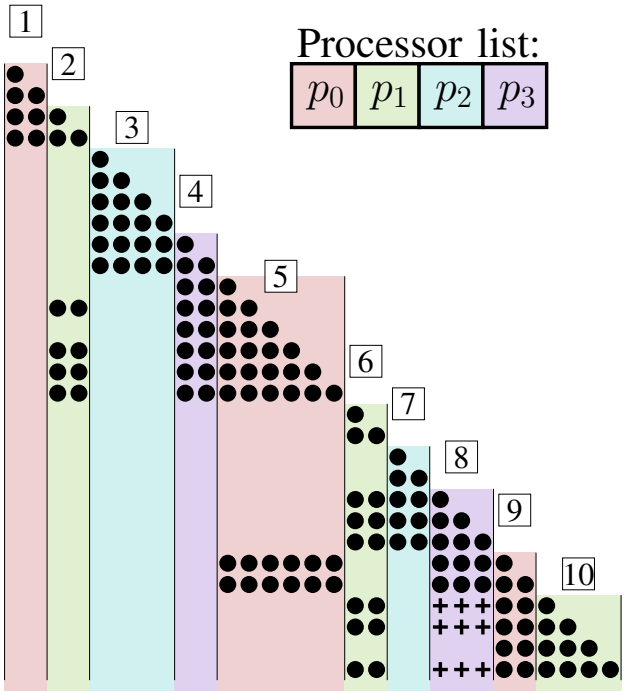
- DAG Scheduling in a distributed (partitioned) memory context
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - "memory constrained" lookahead

Uses a Berkeley extension to UPC to remotely synchronize

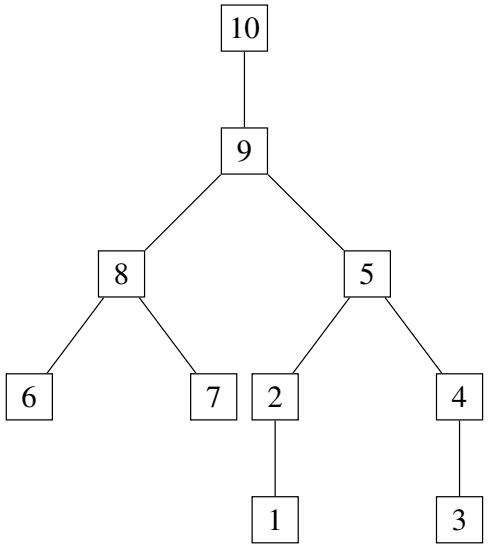


Parry Husbands, Katherine Yelick

symPACK: Sparse Cholesky



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



ɔ) Supernodal elimination tree of
ıatrix **A**

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

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

symPACK: Sparse Cholesky

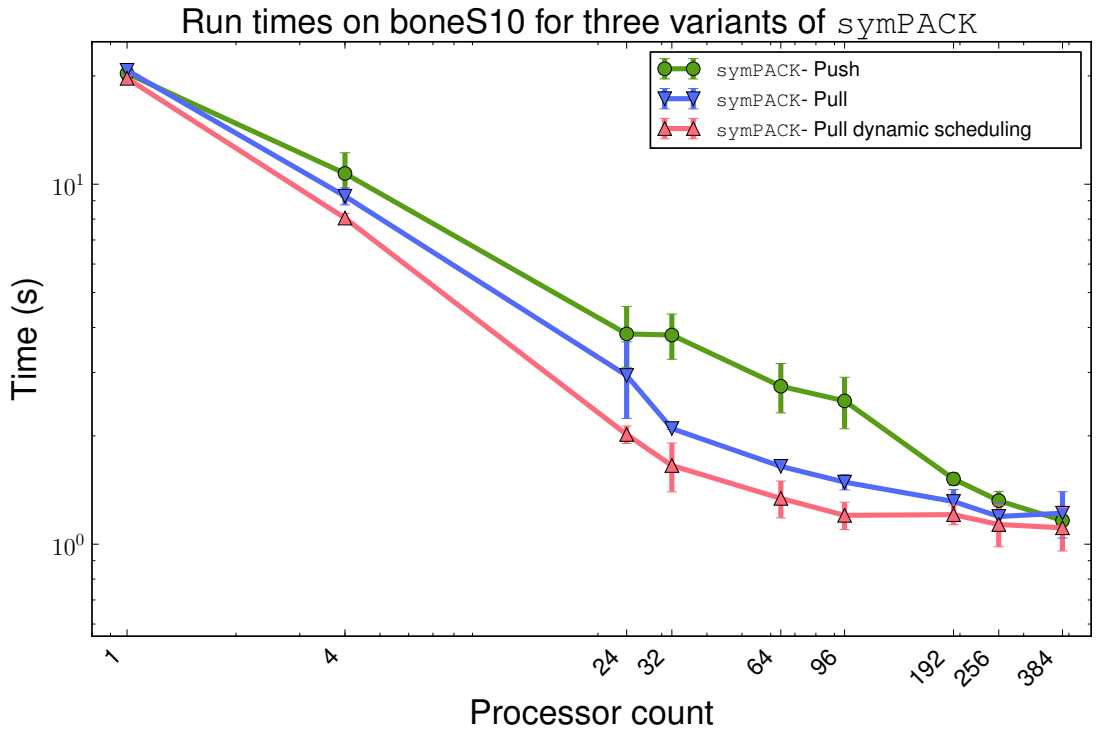


Figure 7: Impact of communication strategy and scheduling on symPACK performance

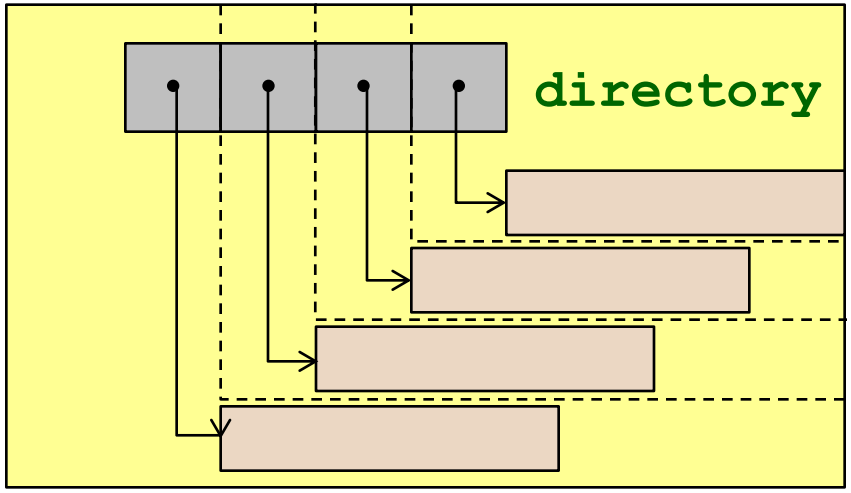
- Scalability of symPACK on Cray XC30 (Edison)
  - Comparable or better than best solvers (evaluation in progress)
  - Notoriously hard parallelism problem

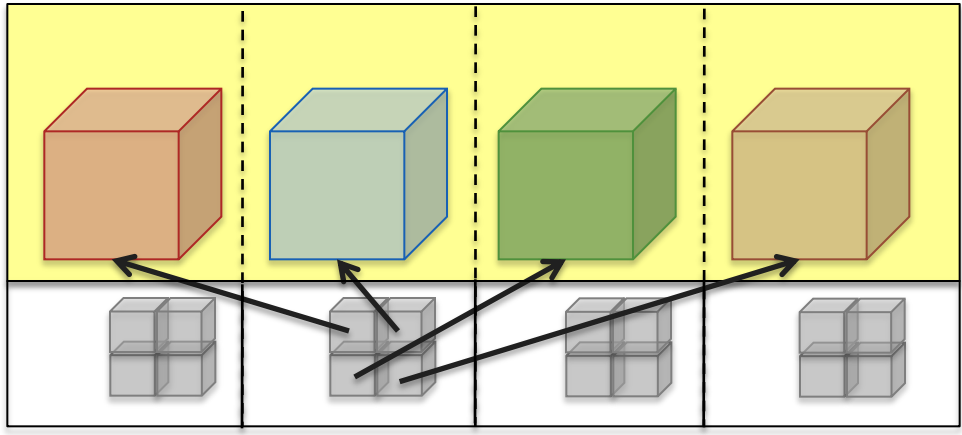
Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

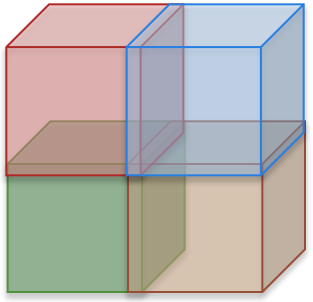
Common Pattern for Distributed Data Structures

- 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

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

Using UPC++ at NERSC

Switch to the gnu C++ compiler

```
module switch PrgEnv-intel PrgEnv-gnu
```

Load the UPC++ module via

```
module load upcxx
```

Compile code with the upcc

```
upcxx="<upcxx-install-path>/bin/upcxx-meta"
g++ --std=c++11 hello-world.cpp $($upcxx PPFLAGS) \
            $($upcxx  LDFLAGS) $($upcxx  LIBFLAGS)
```

Or see the Makefile example…

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/berkeleylab/upcxx/

  Has specification, user guide, codes, and instructions
- GASNet communication
  https://gasnet.lbl.gov

Not used in this course (this year)

- Cray UPC (installed at NERSC)
  http://docs.cray.com/books/S-2179-50/
- Berkeley UPC
  http://upc.lbl.gov