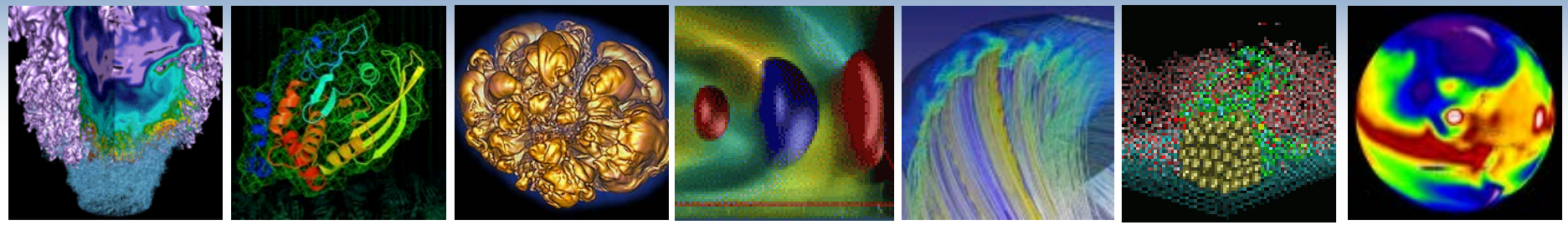


Partitioned Global Address Space Programming

with

Unified Parallel C (UPC)

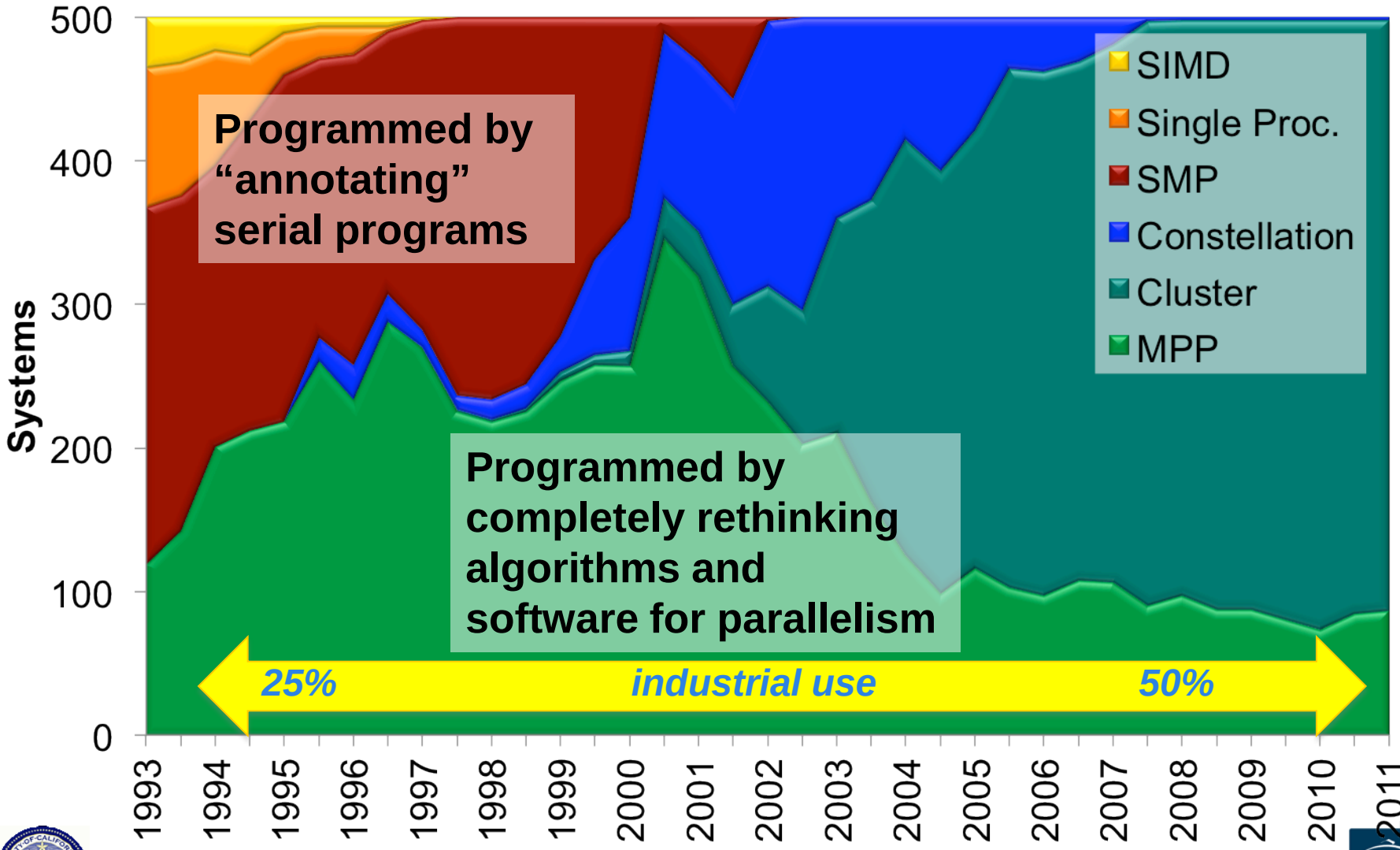
Kathy Yelick

Associate Laboratory Director for Computing Sciences and Acting NERSC Director

Lawrence Berkeley National Laboratory

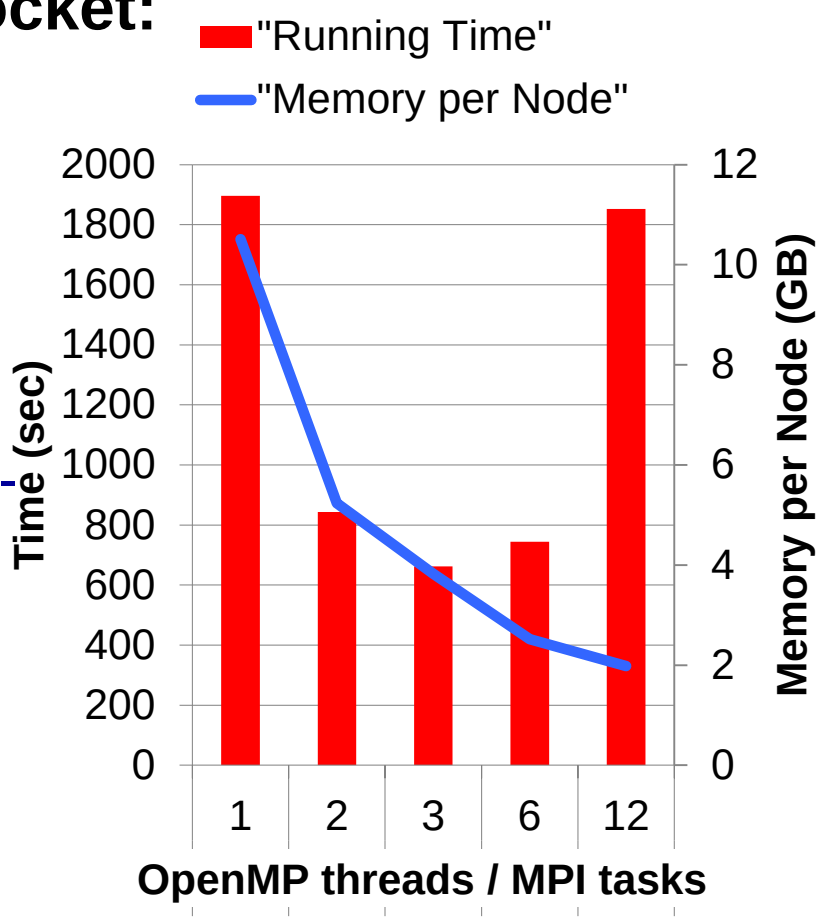
EECS Professor, UC Berkeley

HPC: From Vector Supercomputers to Massively Parallel Systems



Limitations of Existing Programming Models

- **We can run 1 MPI process per core, but there are problems with 6-12+ cores/socket:**
  - Insufficient memory: user level data and internal buffers
  - Runtime overheads: copying and synchronization
- **OpenMP, Pthreads, or other shared memory models**
  - No control over locality, e.g., Non-Uniform Memory Access
  - No explicit memory movement, e.g., accelerators or NVRAM
- **Tuning is non-obvious**
  - Tradeoff between speed and memory footprint



Shared Memory vs. Message Passing

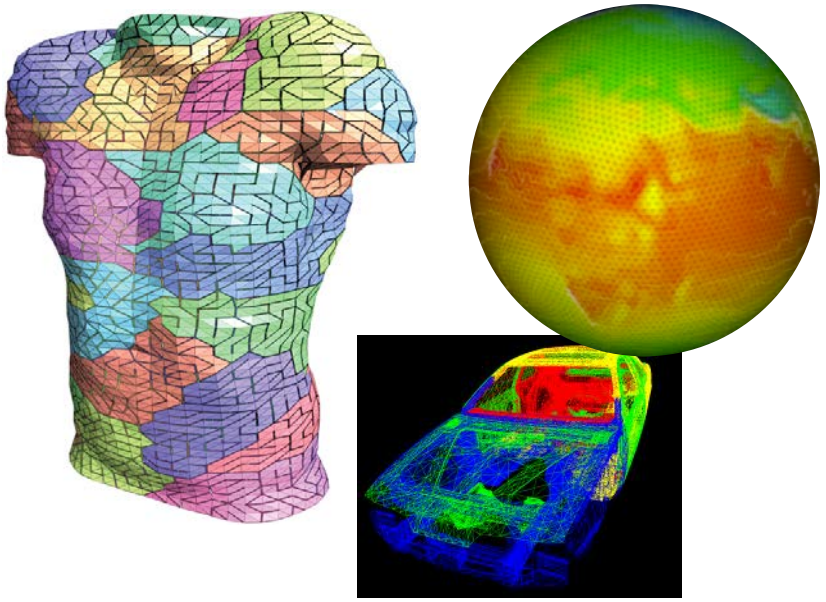
Shared Memory

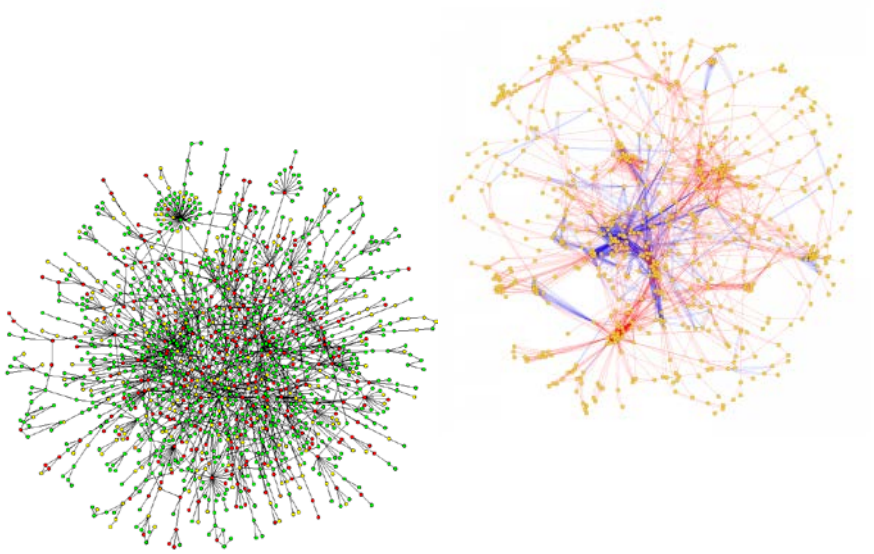
- Advantage: Convenience
  - Can share data structures
  - Just annotate loops
  - Closer to serial code
- Disadvantages
  - No locality control
  - Does not scale
  - Race conditions

Message Passing

- Advantage: Scalability
  - Locality control
  - Communication is all explicit in code (cost transparency)
- Disadvantage
  - Need to rethink entire application / data structures
  - Lots of tedious pack/unpack code
  - Don’t know when to say “receive” for some problems

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

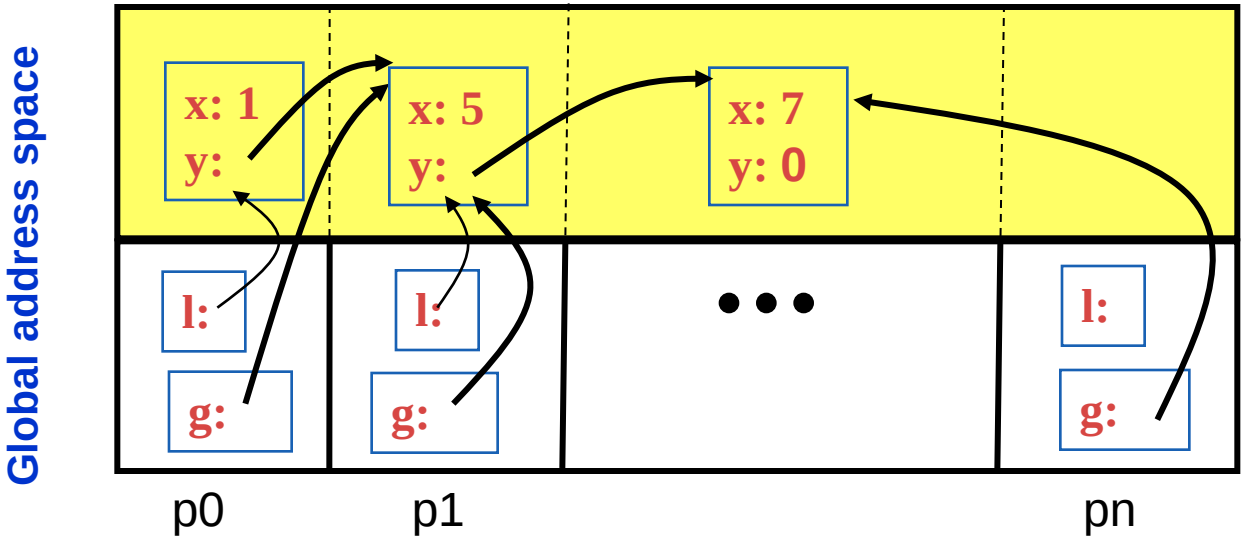
Each start computing

Grab whatever you need whenever

Global Address Space Languages and Libraries

PGAS Languages

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



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

UPC Outline

1. Background
2. UPC Execution Model
3. Basic Memory Model: Shared vs. Private Scalars
4. Synchronization
5. Collectives
6. Data and Pointers
7. Dynamic Memory Management
8. Performance
9. Beyond UPC

History of UPC

- Initial Tech. Report from IDA in collaboration with LLNL and UCB in May 1999 (led by IDA).
  - Based on Split-C (UCB), AC (IDA) and PCP (LLNL)
- UPC consortium participants (past and present) are:
  - ARSC, Compaq, CSC, Cray Inc., Etnus, GMU, HP, IDA CCS, Intrepid Technologies, LBNL, LLNL, MTU, NSA, SGI, Sun Microsystems, UCB, U. Florida, US DOD
  - *UPC is a community effort, well beyond UCB/LBNL*
- Design goals: high performance, expressive, consistent with C goals, …, portable
- UPC Today
  - Multiple vendor and open compilers (Cray, HP, IBM, SGI, gcc-upc from Intrepid, Berkeley UPC)
  - "Pseudo standard" by moving into gcc trunk
  - Most widely used on irregular / graph problems today

UPC Execution Model

UPC Execution Model

- A number of threads working independently in a SPMD fashion
  - Number of threads specified at compile-time or run-time; available as program variable **`THREADS`**
  - **`MYTHREAD`** specifies thread index (**`0..THREADS-1`**)
  - **`upc_barrier`** is a global synchronization: all wait
  - There is a form of parallel loop that we will see later
- There are two compilation modes
  - Static Threads mode:
    - THREADS is specified at compile time by the user
    - The program may use THREADS as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads

Hello World in UPC

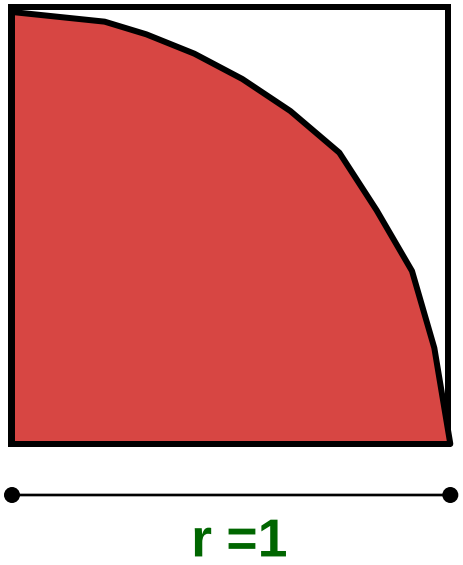
- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Using this fact, plus the identifiers from the previous slides, we can parallel hello world:

```
#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;
```

Each thread gets its own copy of these variables

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

Each thread can use input arguments

```
  srand(MYTHREAD*17);
```

Initialize random in math library

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

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);
  }
}
```

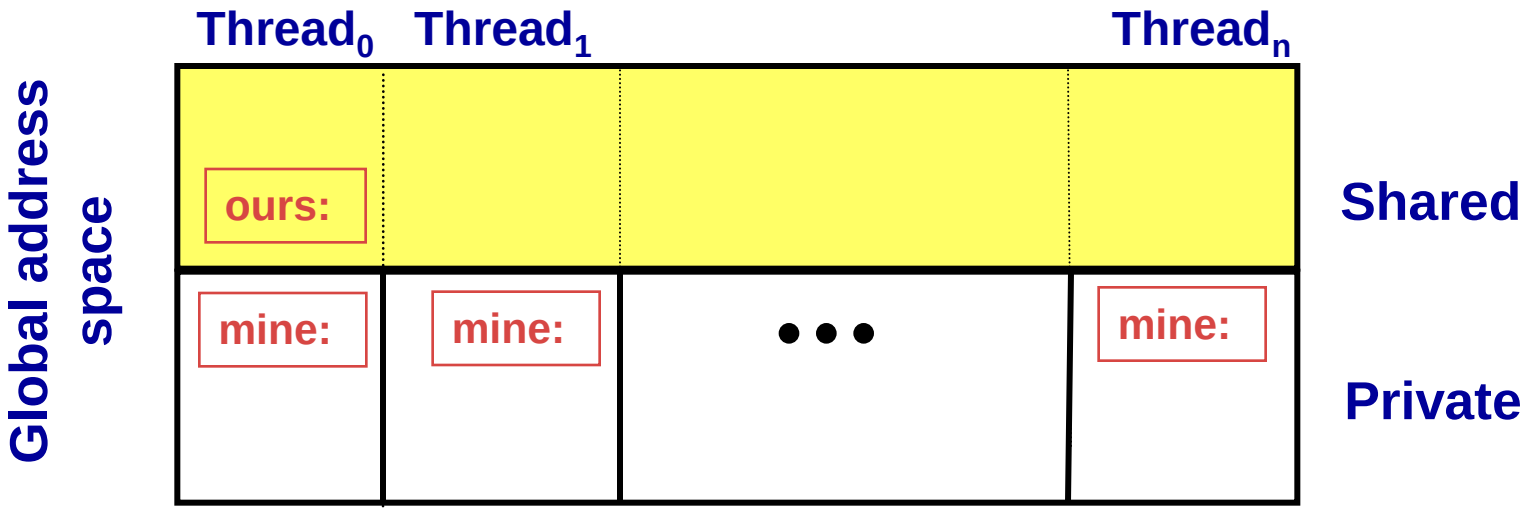
Shared vs. Private Variables

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

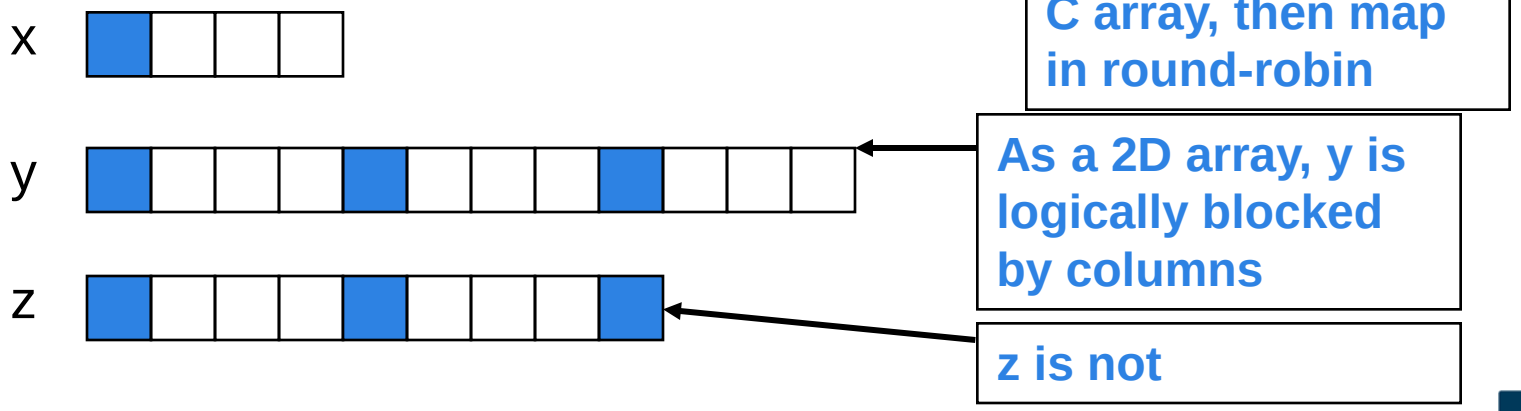
What is the problem with this program?

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
  - Red elts have affinity to thread 0



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

UPC Synchronization

UPC Global 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
- Optional labels allow for debugging

```
#define MERGE_BARRIER 12
if (MYTHREAD%2 == 0) {
    ...
    upc_barrier MERGE_BARRIER;
} else {
    ...
    upc_barrier MERGE_BARRIER;
}
```

Synchronization - Locks

- Locks in UPC are represented by an opaque type:
  `upc_lock_t`
- Locks must be allocated before use:
  `upc_lock_t *upc_all_lock_alloc(void);`

  allocates 1 lock, pointer to all threads

  `upc_lock_t *upc_global_lock_alloc(void);`

  allocates 1 lock, pointer to one thread
- To use a lock:
  `void upc_lock(upc_lock_t *l)`
  `void upc_unlock(upc_lock_t *l)`

  use at start and end of critical region
- Locks can be freed when not in use
  `void upc_lock_free(upc_lock_t *ptr);`

Pi in UPC: Shared Memory Style

- Parallel computing of pi, without the bug

```
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);
}
```

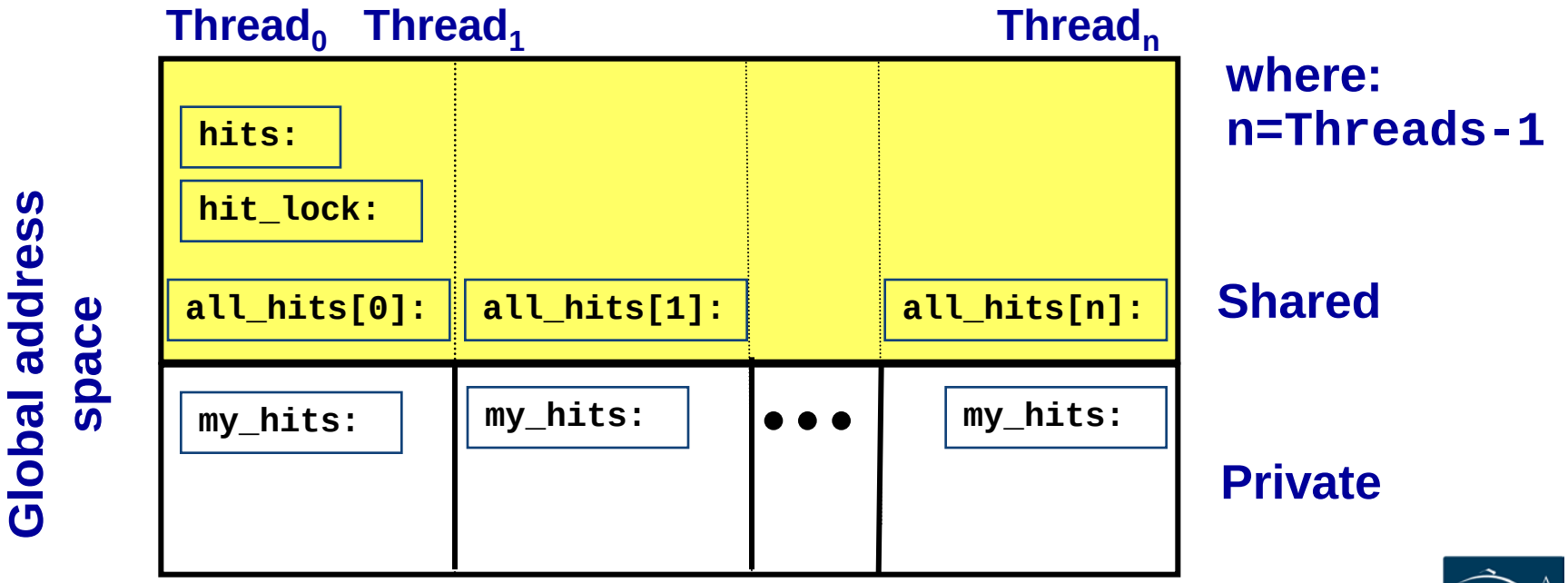
create a lock

accumulate hits locally

accumulate across threads

Recap: Private vs. Shared Variables in UPC

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



UPC Collectives

UPC Collectives in General

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

Pi in UPC: Data Parallel Style

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

```
#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

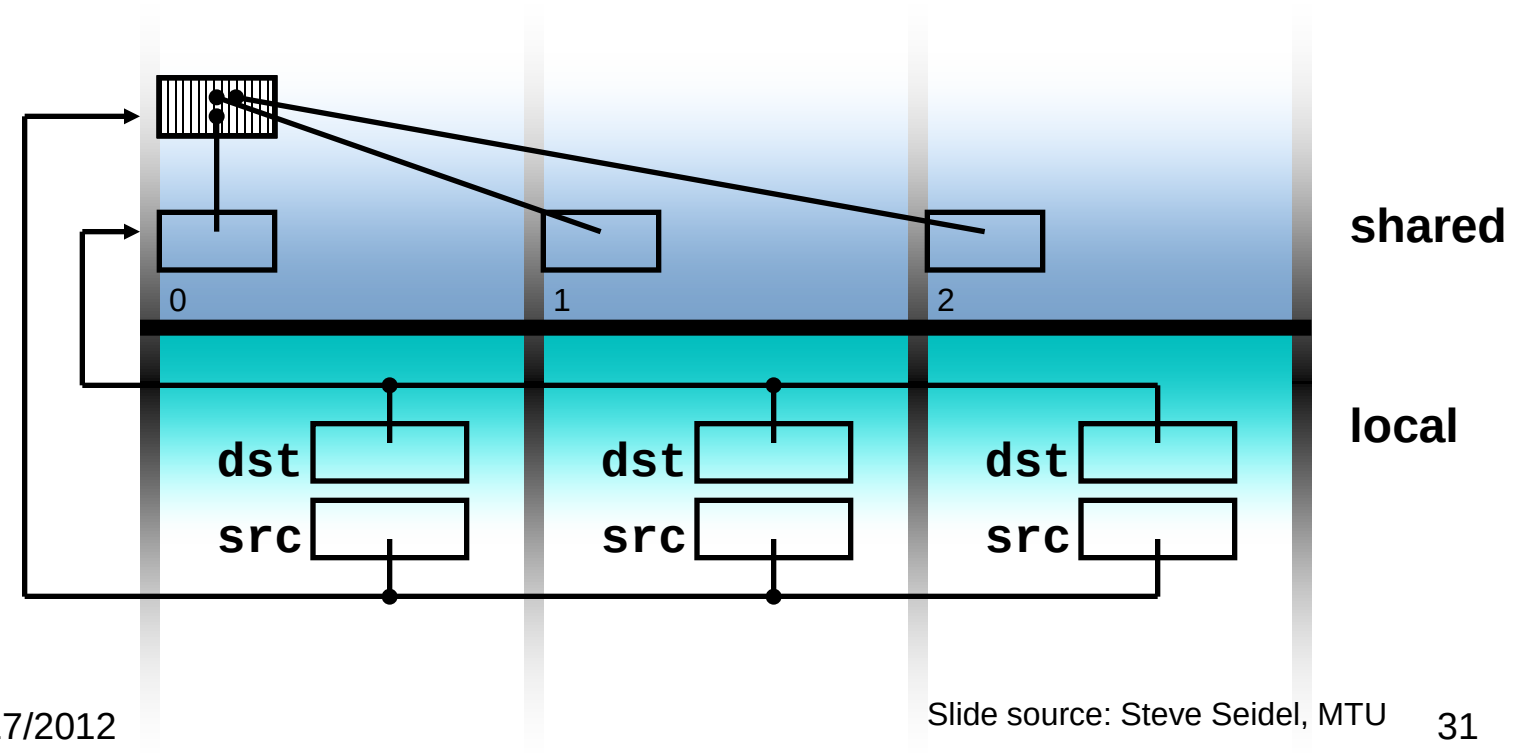
barrier implied by collective

UPC (Value-Based) Collectives in General

- General arguments:
  - rootthread is the thread ID for the root (e.g., the source of a broadcast)
  - All 'value' arguments indicate an l-value (i.e., a variable or array element, not a literal or an arbitrary expression)
  - All 'TYPE' arguments should the scalar type of collective operation
  - upc_op_t is one of: UPC_ADD, UPC_MULT, UPC_AND, UPC_OR, UPC_XOR, UPC_LOGAND, UPC_LOGOR, UPC_MIN, UPC_MAX
- Computational Collectives
  - TYPE bupc_allv_reduce(TYPE, TYPE value, int rootthread, upc_op_t reductionop)
  - TYPE bupc_allv_reduce_all(TYPE, TYPE value, upc_op_t reductionop)
  - TYPE bupc_allv_prefix_reduce(TYPE, TYPE value, upc_op_t reductionop)
- Data movement collectives
  - TYPE bupc_allv_broadcast(TYPE, TYPE value, int rootthread)
  - TYPE bupc_allv_scatter(TYPE, int rootthread, TYPE *rootsrcarray)
  - TYPE *bupc_allv_gather(TYPE, TYPE value, int rootthread, TYPE *rootdestarray)
    - Gather a 'value' (which has type TYPE) from each thread to 'rootthread', and place them (in order by source thread) into the local array 'rootdestarray' on 'rootthread'.
  - TYPE *bupc_allv_gather_all(TYPE, TYPE value, TYPE *destarray)
  - TYPE bupc_allv_permute(TYPE, TYPE value, int tothreadid)
    - Perform a permutation of 'value's across all threads. Each thread passes a value and a unique thread identifier to receive it - each thread returns the value it receives.

Full UPC Collectives

- Value-based collectives pass in and return scalar values
- But sometimes you want to collect over arrays
- When can a collective argument begin executing?
  - Arguments with affinity to thread *i* are ready when thread *i* calls the function; results with affinity to thread *i* are ready when thread *i* returns.
  - This is appealing but it is incorrect: In a broadcast, thread 1 does not know when thread 0 is ready.



Slide source: Steve Seidel, MTU

UPC Collective: Sync Flags

- In full UPC Collectives, blocks of data may be collected
- A extra argument of each collective function is the sync mode of type upc_flag_t.
- Values of sync mode are formed by or-ing together a constant of the form UPC_IN_*X*SYNC and a constant of the form UPC_OUT_*Y*SYNC, where *X* and *Y* may be NO, MY, or ALL.
- If sync_mode is (UPC IN_*X*SYNC | UPC OUT *Y*SYNC), then if *X* is:
  - NO the collective function may begin to read or write data when the first thread has entered the collective function call,
  - MY the collective function may begin to read or write only data which has affinity to threads that have entered the collective function call, and
  - ALL the collective function may begin to read or write data only after all threads have entered the collective function call
- and if *Y* is
  - NO the collective function may read and write data until the last thread has returned from the collective function call,
  - MY the collective function call may return in a thread only after all reads and writes of data with affinity to the thread are complete3, and
  - ALL the collective function call may return only after all reads and writes of data are complete.

Work Distribution Using `upc_forall`

Example: Vector Addition

- Questions about parallel vector additions:
  - How to layout data (here it is cyclic)
  - Which processor does what (here it is "owner computes")

```
/* vadd.c */
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD == i%THREADS)
            sum[i]=v1[i]+v2[i];
}
```

cyclic layout

owner computes

Work Sharing with upc_forall()

- The idiom in the previous slide is very common
  - Loop over all; work on those owned by this proc
- UPC adds a special type of loop

```
upc_forall(init; test; loop; affinity)
    statement;
```

- Programmer indicates the iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression indicates which iterations to run on each thread. It may have one of two types:
  - Integer: **`affinity%THREADS`** is **`MYTHREAD`**
  - Pointer: **`upc_threadof(affinity)`** is **`MYTHREAD`**
- **Syntactic sugar for loop on previous slide**
  - *Some* compilers *may* do better than this, e.g.,

```
for(i=MYTHREAD; i<N; i+=THREADS)
```

  - Rather than having all threads iterate N times:

```
for(i=0; i<N; i++) if (MYTHREAD == i%THREADS)
```

Vector Addition with upc_forall

- The **vadd** example can be rewritten as follows
  - Equivalent code could use "**&sum[i]**" for affinity
  - The code would be correct but slow if the affinity expression were **i+1** rather than **i**.

```
#define N 100*THREADS

  shared int v1[N], v2[N], sum[N];

  void main() {
      int i;
      upc_forall(i=0; i<N; i++; i)
                  sum[i]=v1[i]+v2[i];
  }
```

The cyclic data distribution may perform poorly on some machines

Distributed Arrays in UPC

Blocked Layouts in UPC

- If this code were doing nearest neighbor averaging (3pt stencil) the cyclic layout would be the worst possible layout.
- Instead, want a blocked layout
- Vector addition example can be rewritten as follows using a blocked layout

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

   void main() {
        int i;
        upc_forall(i=0; i<N; i++; &sum[i])

                    sum[i]=v1[i]+v2[i];
   }
```

blocked layout

Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:
  - Empty (cyclic layout)
  - [*] (blocked layout)
  - [0] or [] (indefinite layout, all on 1 thread)
  - [b] or [b1][b2]…[bn] = [b1*b2*…bn] (fixed block size)
- The affinity of an array element is defined in terms of:
  - block size, a compile-time constant
  - and THREADS.
- Element i has affinity with thread

  `(i / block_size) % THREADS`
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping

2D Array Layouts in UPC

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

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

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

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

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

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

- or equivalently

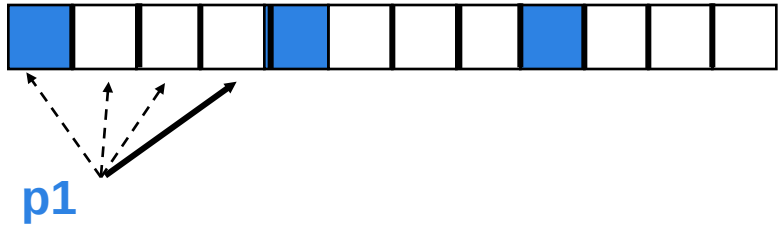
  shared [b1*b2] int a5 [m][n][r][c][b1][b2];

Pointers to Shared vs. Arrays

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

```
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
  int i;
  shared int *p1, *p2;

  p1=v1; p2=v2;
  for (i=0; i<N; i++, p1++, p2++ )
     if (i %THREADS= = MYTHREAD)
           sum[i]= *p1 + *p2;
  }
```



UPC Pointers

Where does the pointer point?

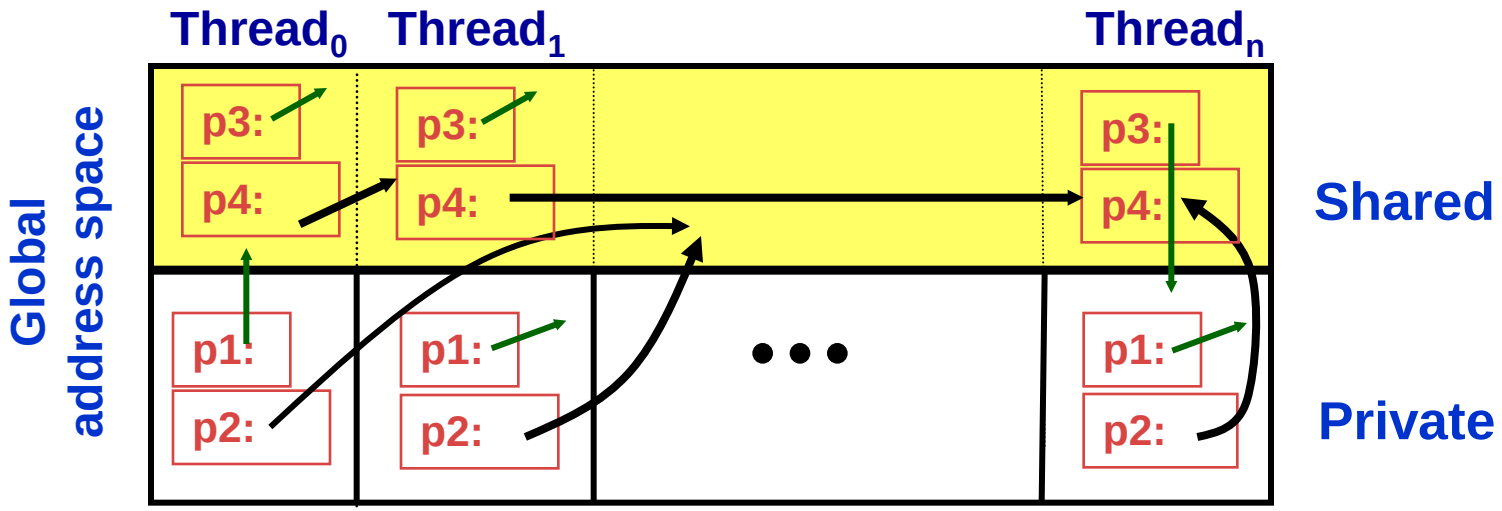
Where does the pointer reside?

	Local	Shared
Private	**p1**	**p2**
Shared	**p3**	**p4**

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

Shared to local memory (p3) is not recommended.

UPC Pointers




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

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.

Common Uses for UPC Pointer Types

```
int *p1;
```

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

```
shared int *p2;
```

- Use to refer to remote data
- Larger and slower due to test-for-local + possible communication

```
int *shared p3;
```

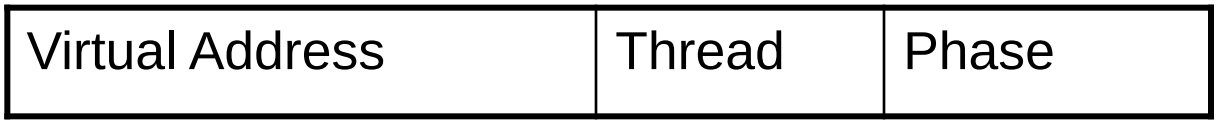
- Not recommended

```
shared int  *shared p4;
```

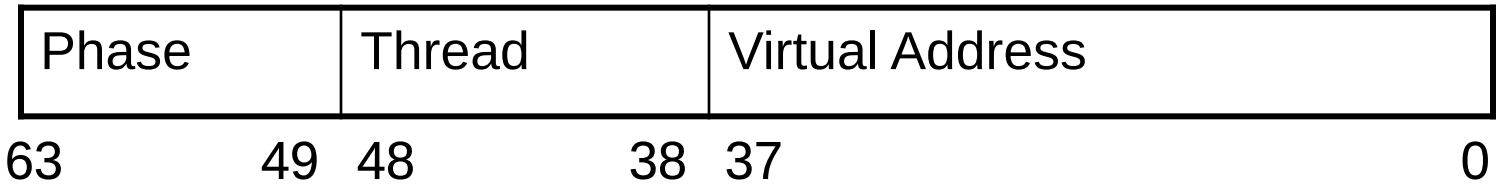
- Use to build shared linked structures, e.g., a linked list

UPC Pointers

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



- Example: Cray T3E implementation



UPC Pointers

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

Special Functions

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

Dynamic Memory Allocation in UPC

- Dynamic memory allocation of shared memory is available in UPC
- Functions can be collective or not
  - A collective function has to be called by every thread and will return the same value to all of them

Global Memory Allocation

```
shared void *upc_global_alloc(size_t nblocks,
   size_t nbytes);
```

nblocks : number of blocks
nbytes : block size

- Non-collective: called by one thread
- The calling thread allocates a contiguous memory space in the shared space with the shape:

```
shared [nbytes] char[nblocks * nbytes]
```

```
shared void *upc_all_alloc(size_t nblocks,
   size_t nbytes);
```

- The same result, but must be called by all threads together
- All the threads will get the same pointer

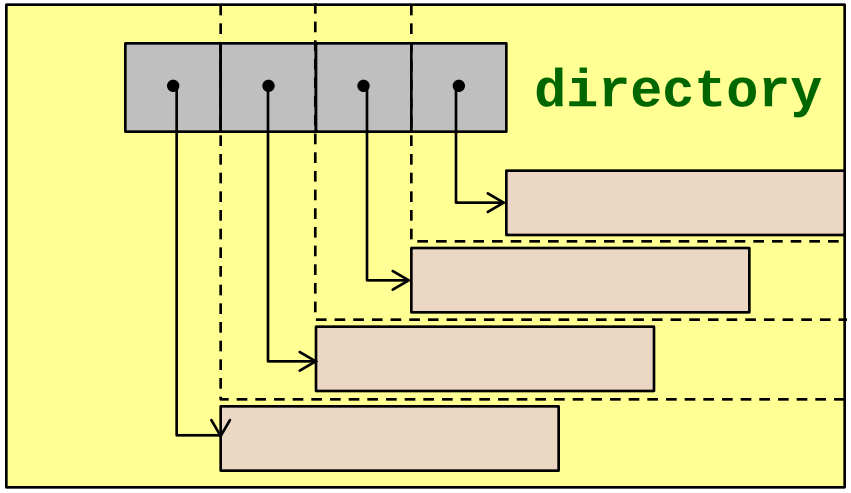
```
void upc_free(shared void *ptr);
```

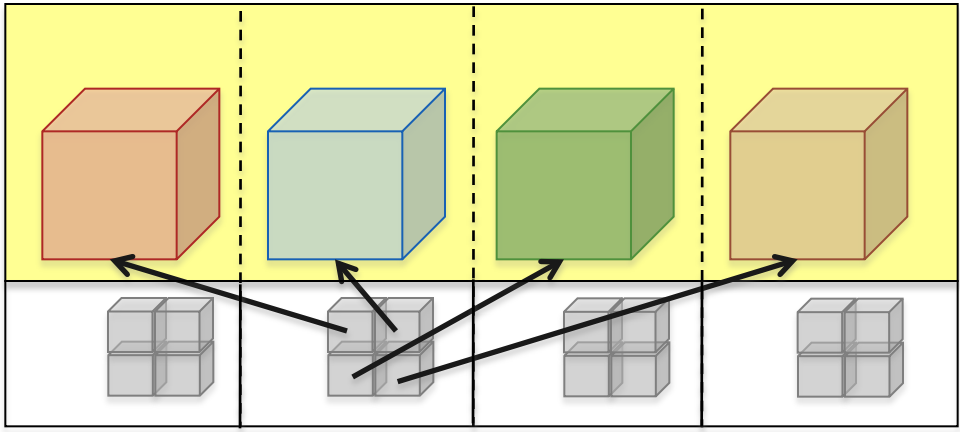
•Non-collective function; frees the dynamically allocated shared memory pointed to by ptr

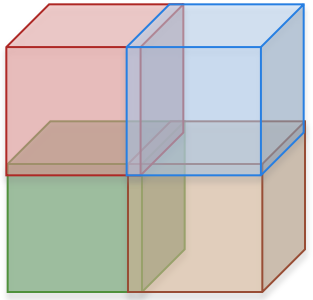
Distributed Arrays Directory Style

- 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

Memory Consistency in UPC

- The consistency model defines the order in which one thread may see another threads accesses to memory
  - If you write a program with unsychronized accesses, what happens?
  - Does this work?

```
data = …                    while (!flag) { };
flag = 1;                   … = data;   // use the data
```

- UPC has two types of accesses:
  - Strict: will always appear in order
  - Relaxed: May appear out of order to other threads
- There are several ways of designating the type, commonly:
  - Use the include file:

```
#include <upc_relaxed.h>
```

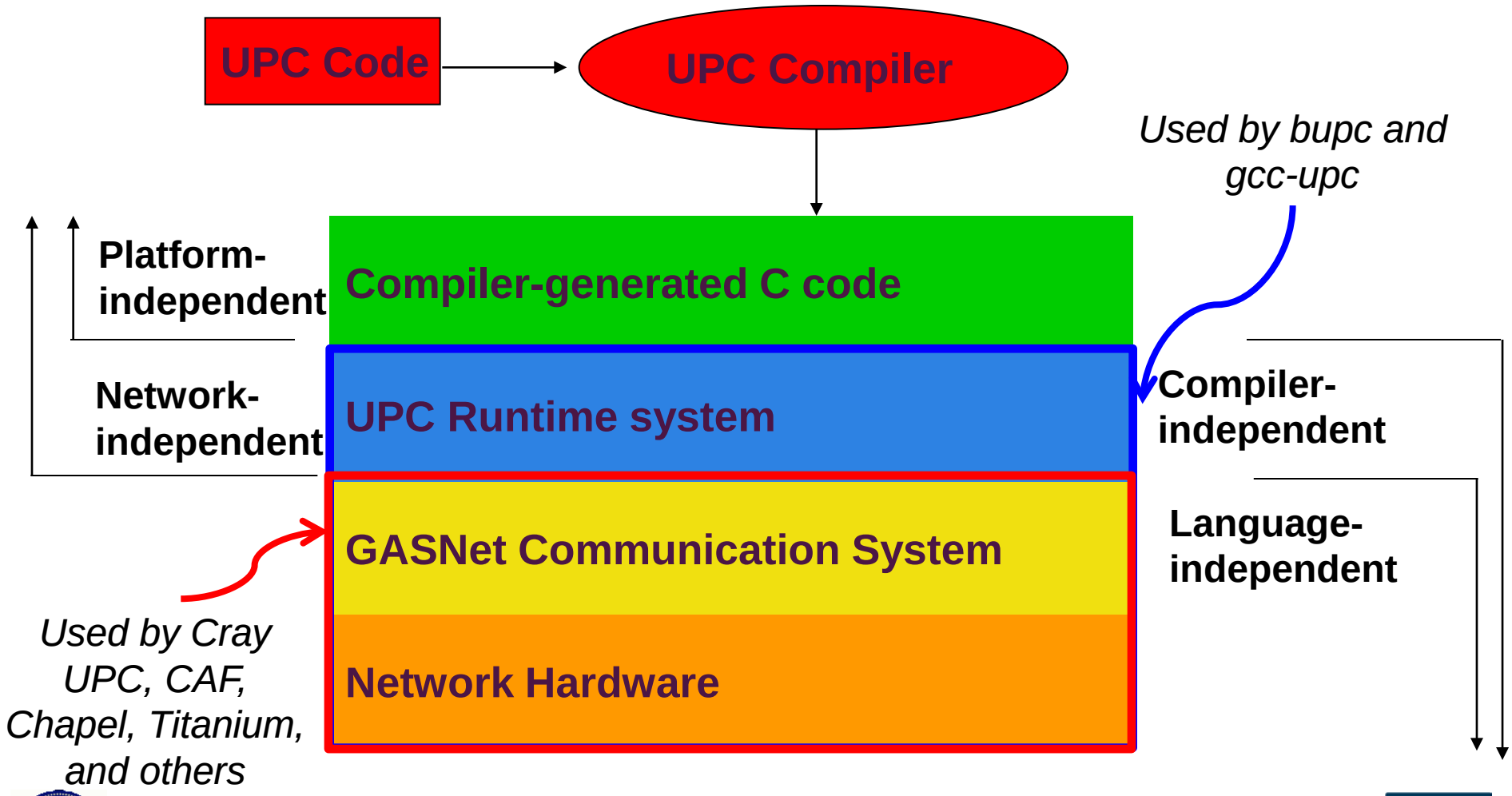
  - Which makes all accesses in the file relaxed by default
  - Use strict on variables that are used as synchronization (**`flag`**)

Synchronization- Fence

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

Performance of UPC

Berkeley UPC Compiler



PGAS Languages have Performance Advantages

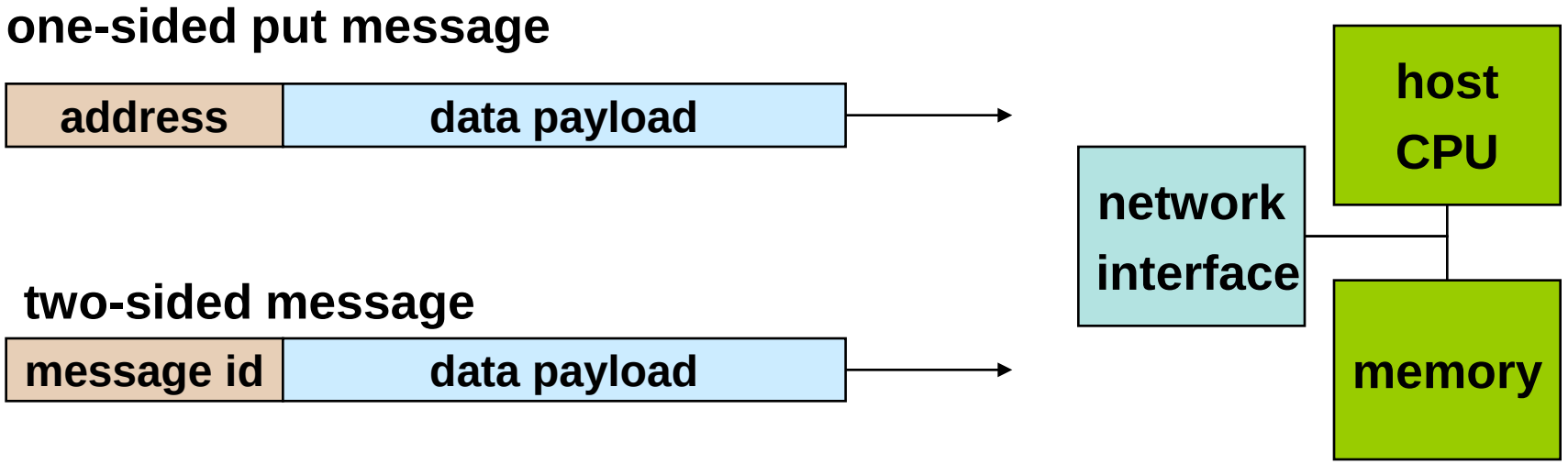
Strategy for acceptance of a new language

- Make it run faster than anything else

Keys to high performance

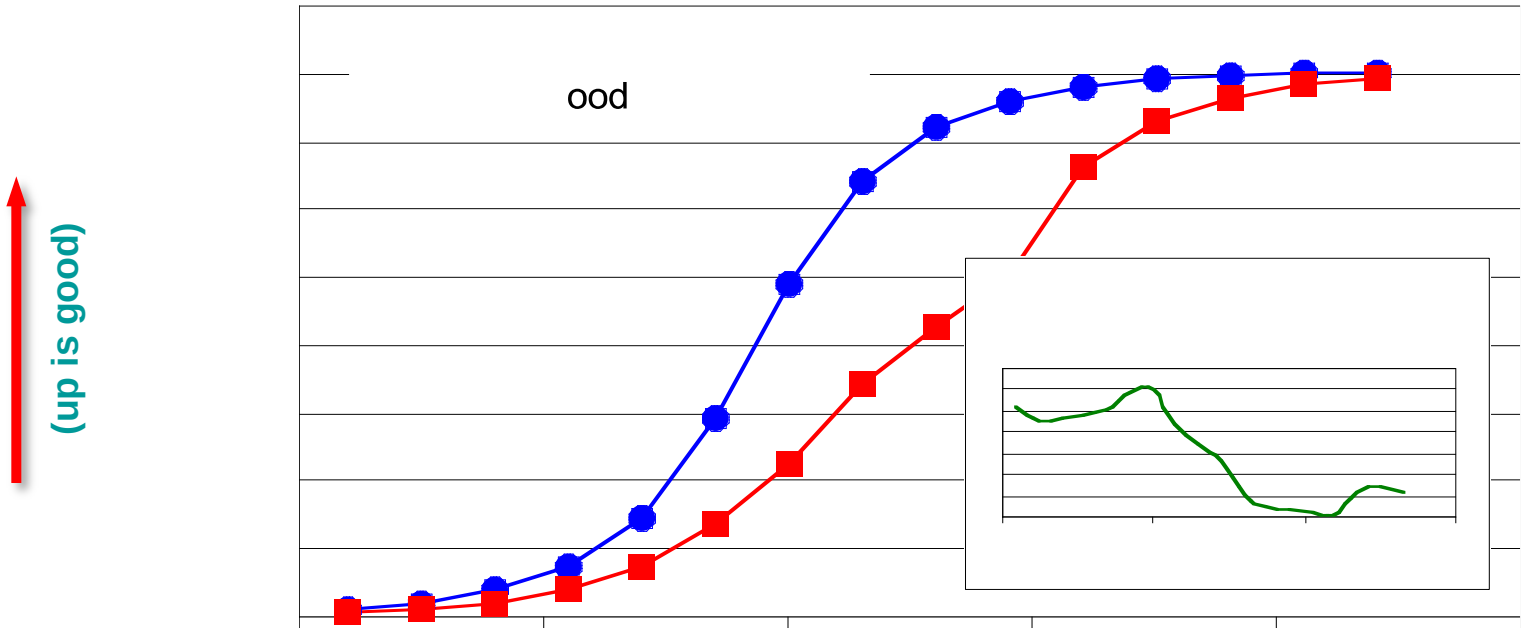
- Parallelism:
  - Scaling the number of processors
- Maximize single node performance
  - Generate friendly code or use tuned libraries (BLAS, FFTW, etc.)
- Avoid (unnecessary) communication cost
  - Latency, bandwidth, overhead
  - Berkeley UPC and Titanium use GASNet communication layer
- Avoid unnecessary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

One-Sided vs Two-Sided



- 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

One-Sided vs. Two-Sided: Practice

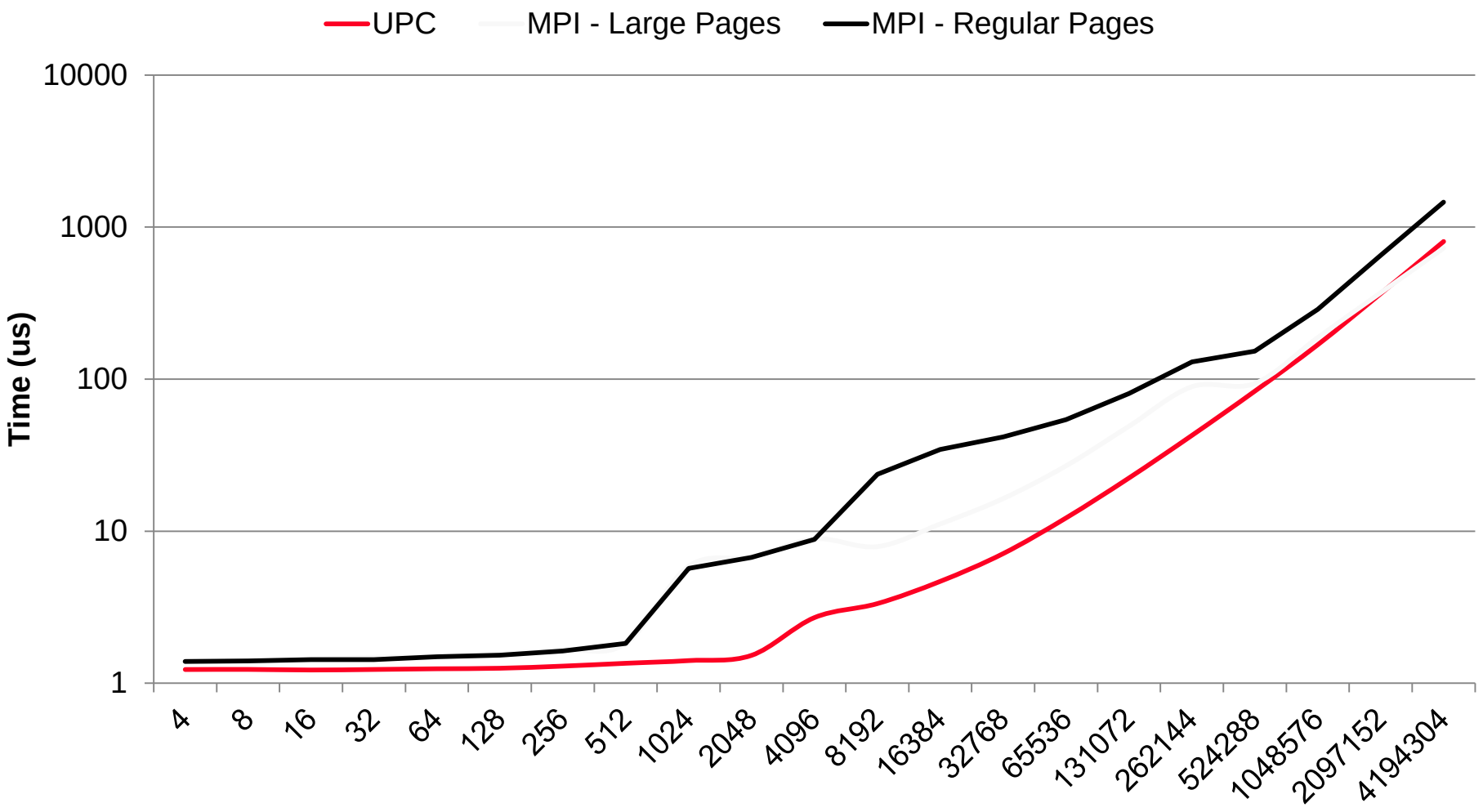


NERSC Jacquard machine with Opteron processors

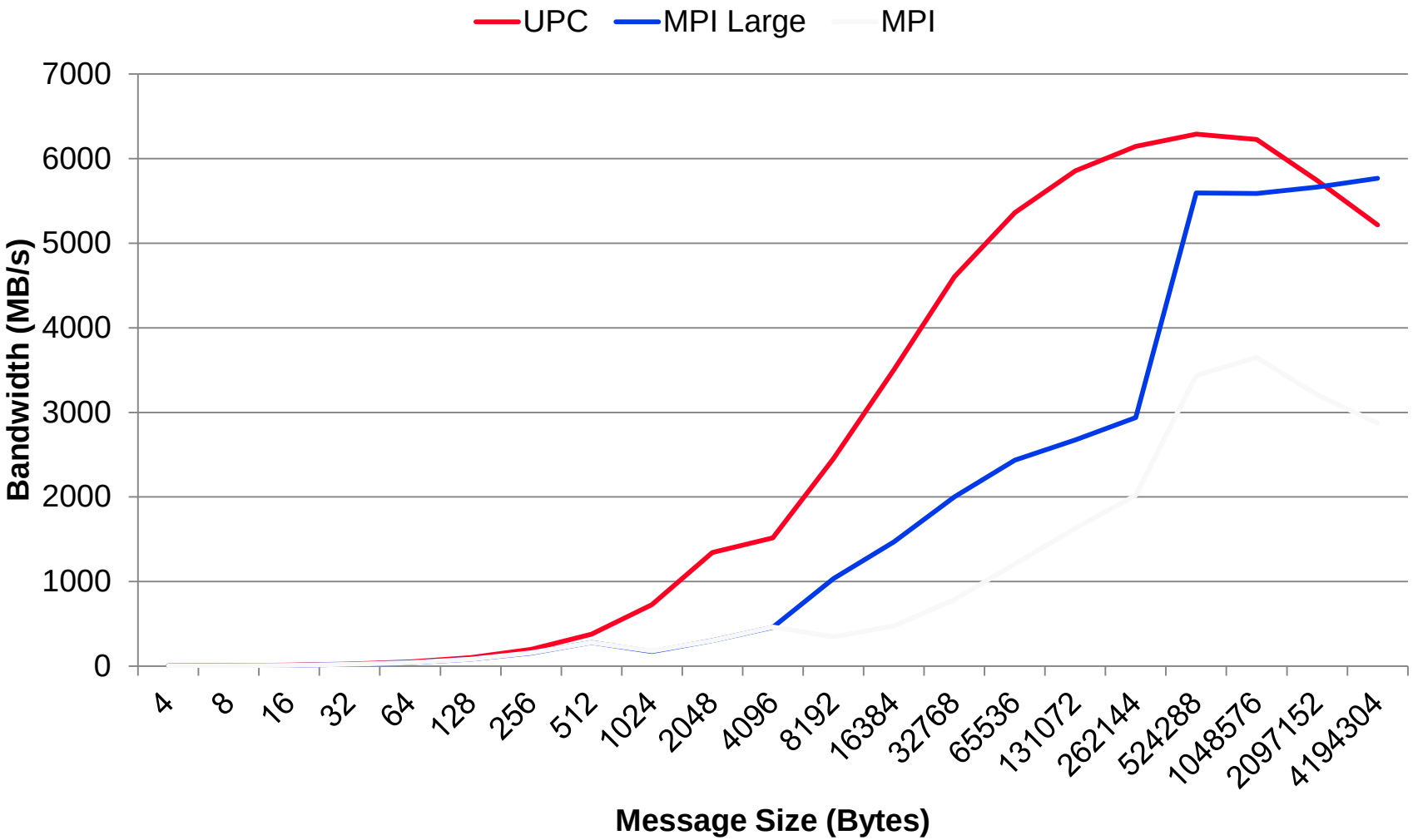
- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point (N ½) differs by *one order of magnitude*
- This is not a criticism of the implementation!

Joint work with Paul Hargrove and Dan Bonachea

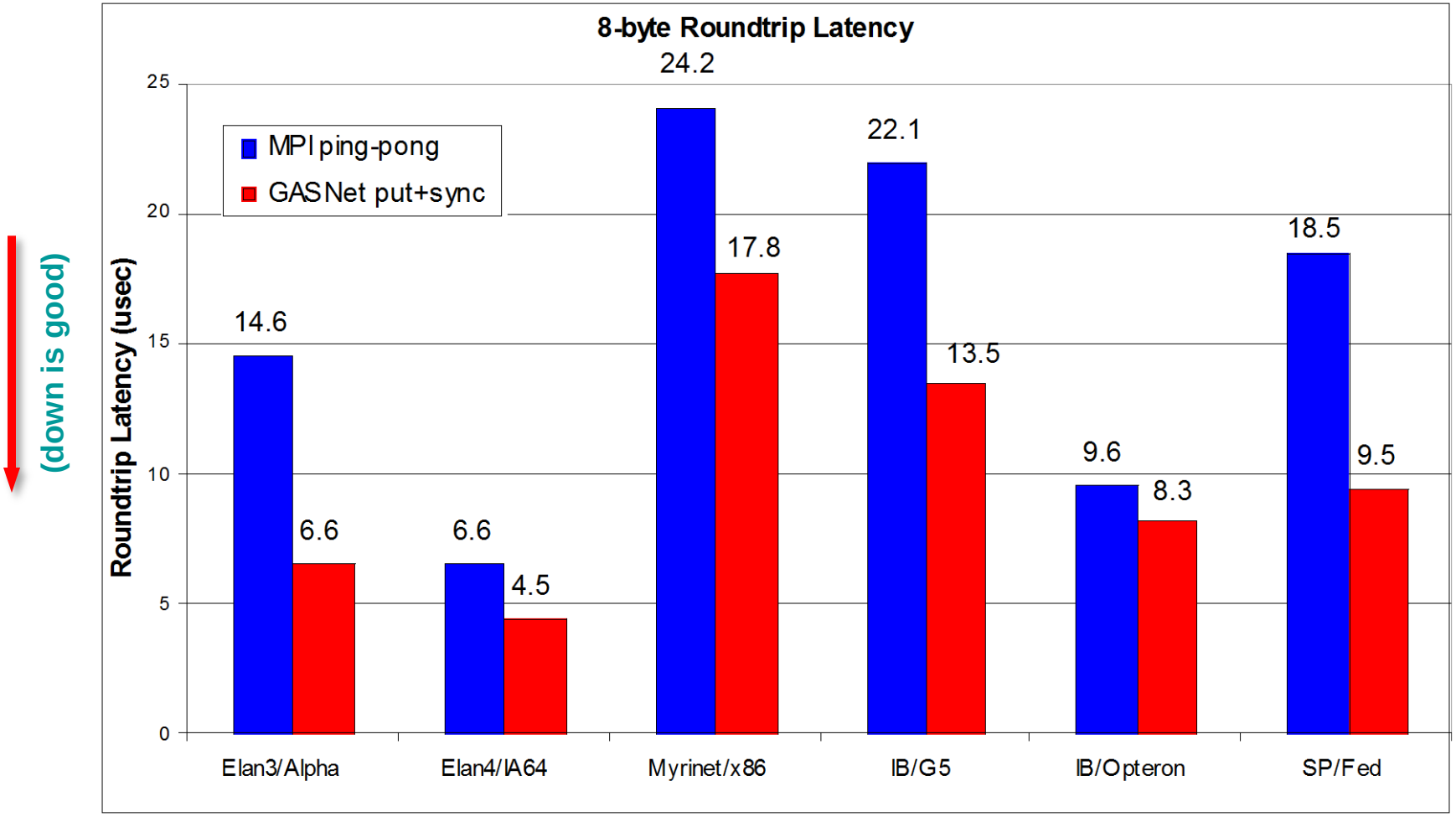
Ping Pong Latency on a Cray XE6 (Hopper)



Bandwidths on Cray XE6 (Hopper)



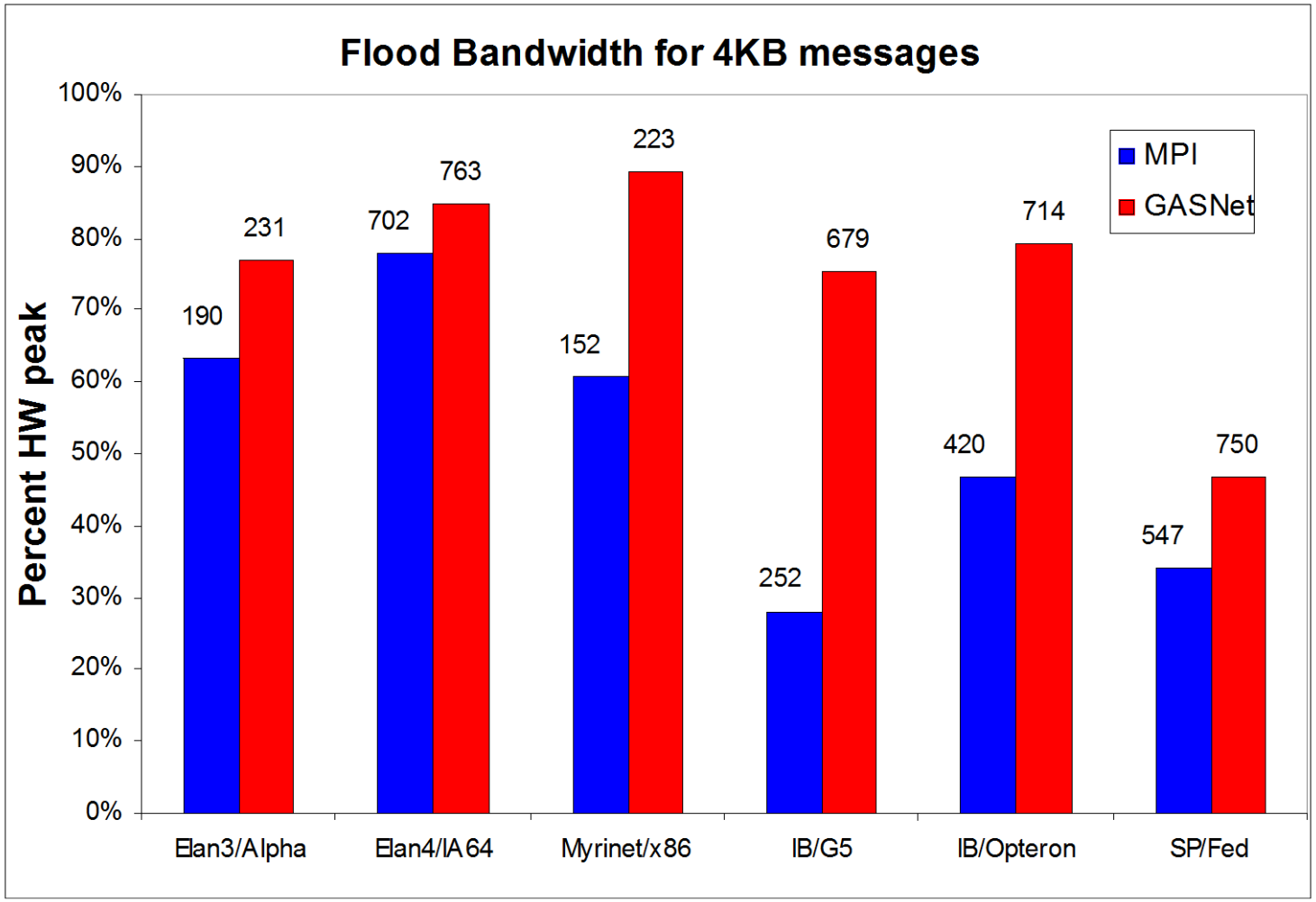
GASNet: Portability *and* High-Performance



GASNet better for latency across machines

Joint work with UPC Group; GASNet design by Dan Bonachea

GASNet: Portability *and* High-Performance



(up is good)

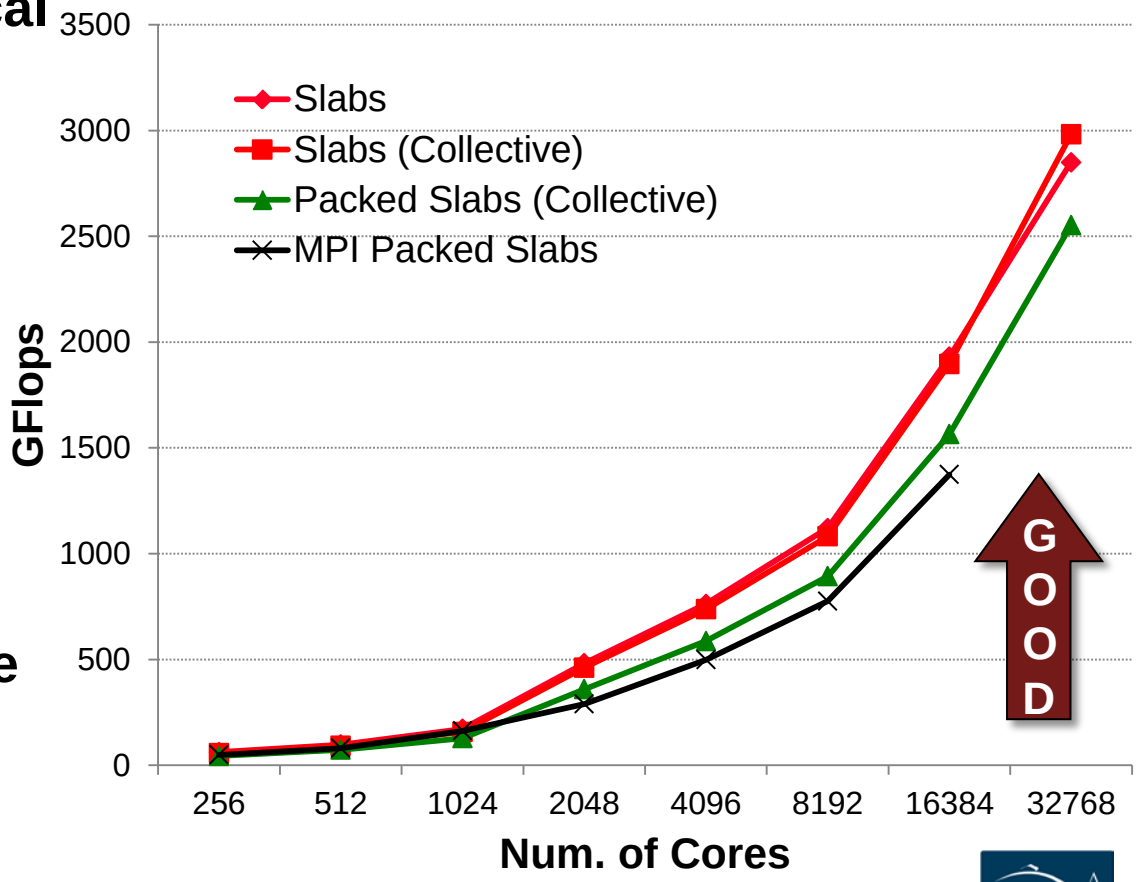
GASNet excels at mid-range sizes: important for overlap

Joint work with UPC Group; GASNet design by Dan Bonachea

FFT Performance on BlueGene/P

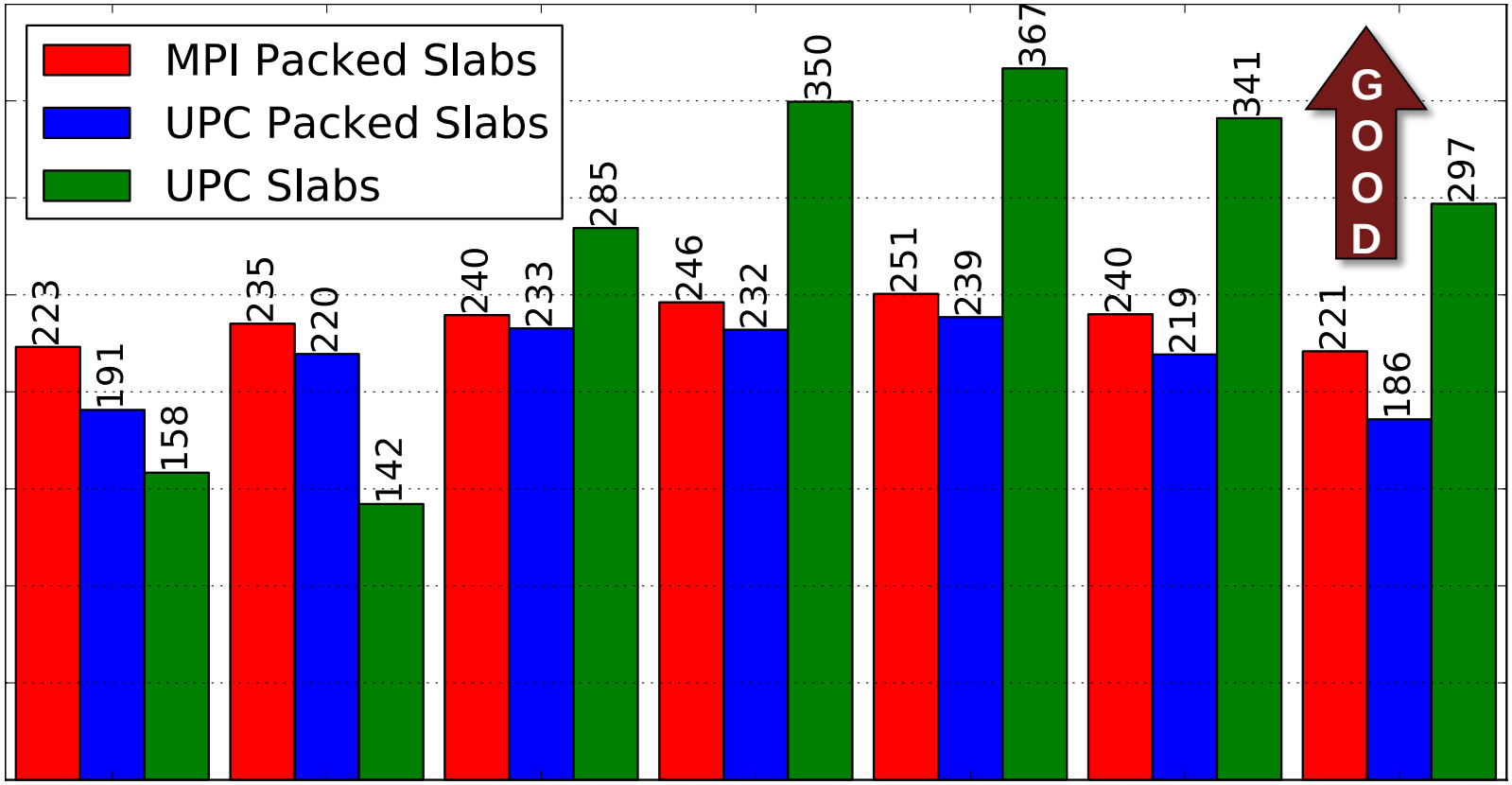
- **UPC implementation consistently outperform MPI**
- **Uses highly optimized local FFT library on each node**
- **UPC version avoids send/receive synchronization**
  - Lower overhead
  - Better overlap
  - Better bisection bandwidth
- **Numbers are getting close to HPC record on BG/P**

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



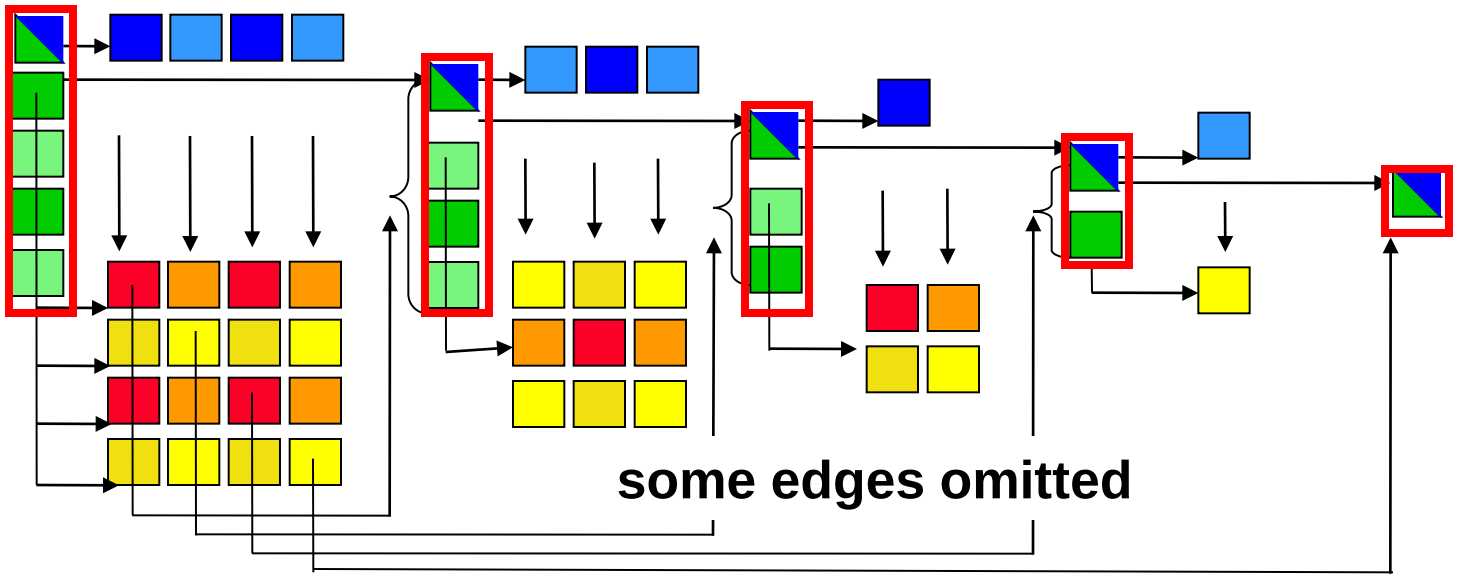
FFT Performance on Cray XT4

- 1024 Cores of the Cray XT4
  - Uses FFTW for local FFTs
  - Larger the problem size the more effective the overlap

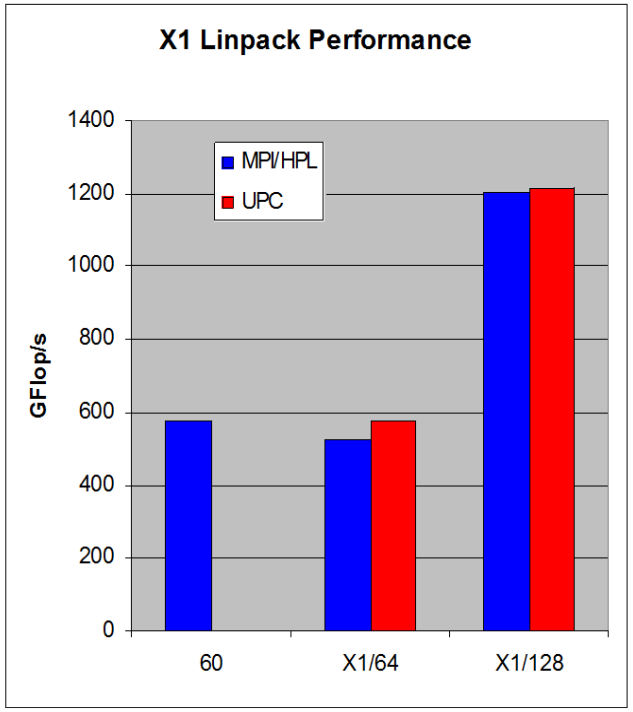


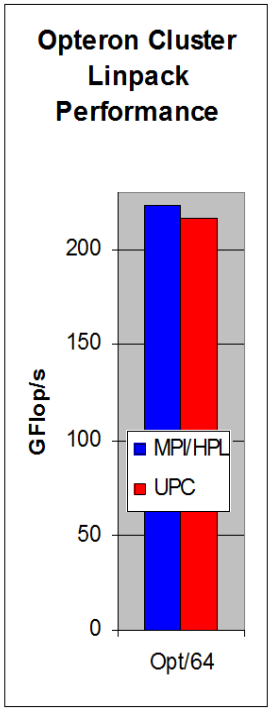
Event Driven LU in UPC

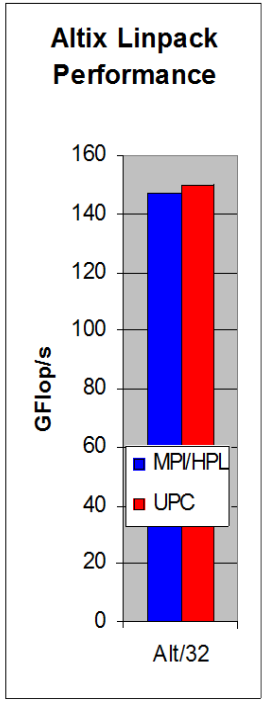
- DAG Scheduling before it's time
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - "memory constrained"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



UPC HPL Performance





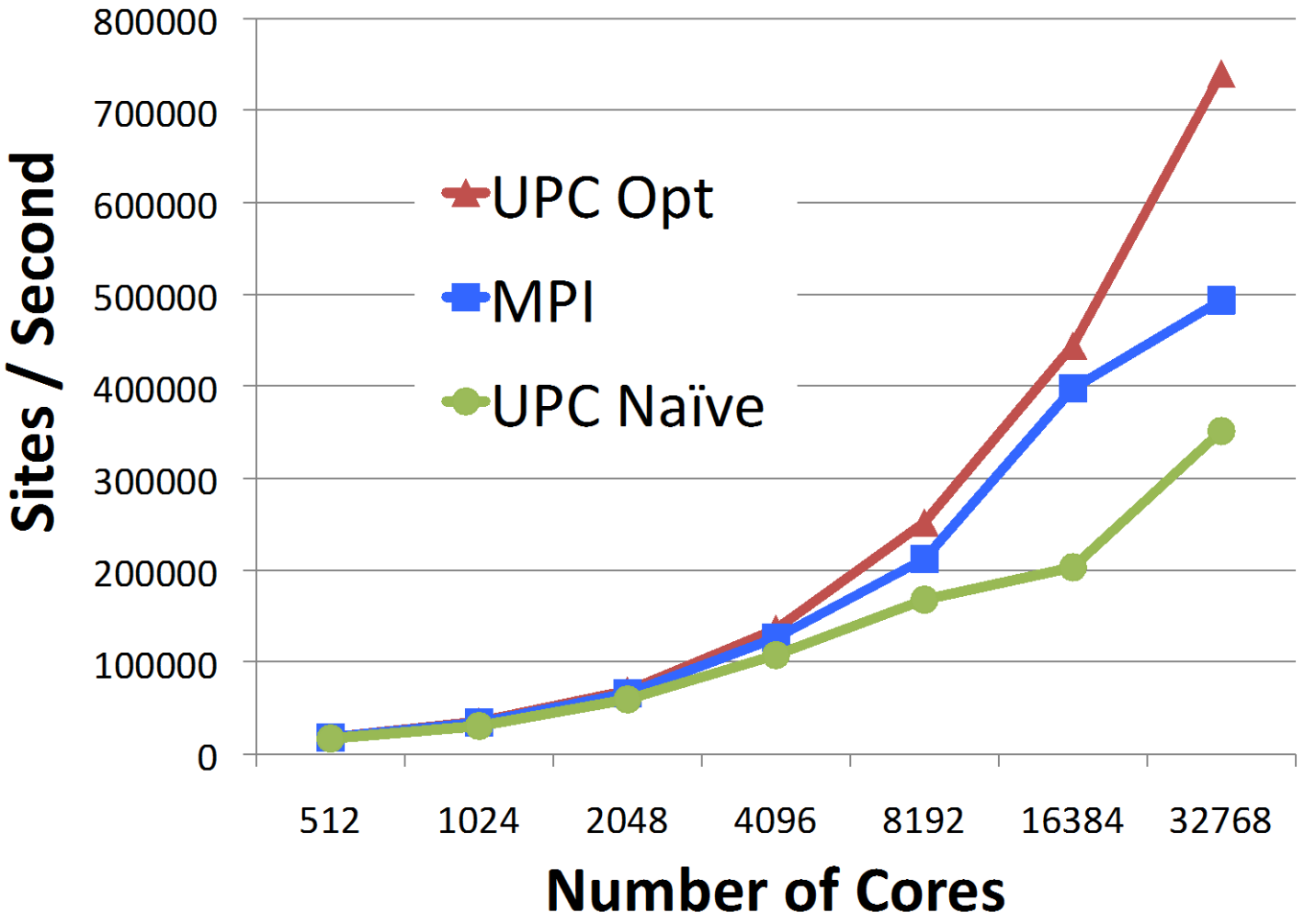


- **MPI HPL numbers from HPCC database**
- **Large scaling:**
  - 2.2 TFlops on 512p,
  - 4.4 TFlops on 1024p (Thunder)

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

Joint work with Parry Husbands

MILC (QCD) Performance in UPC



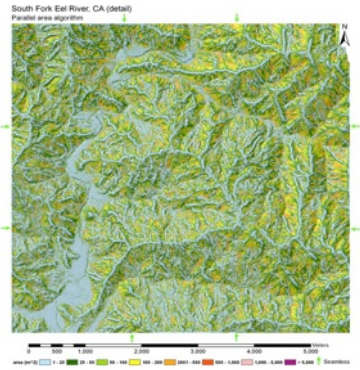
- MILC is Lattice Quantum Chromo-Dynamics application
- UPC scales better than MPI when carefully optimized

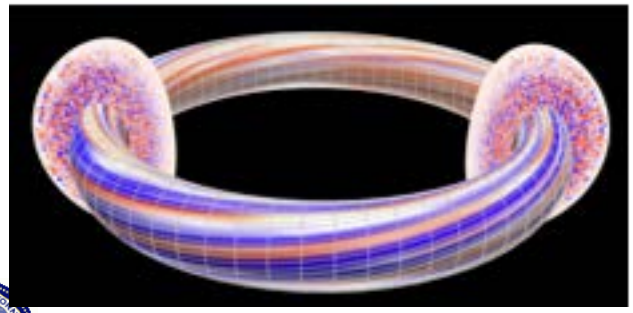
A Family of PGAS Languages

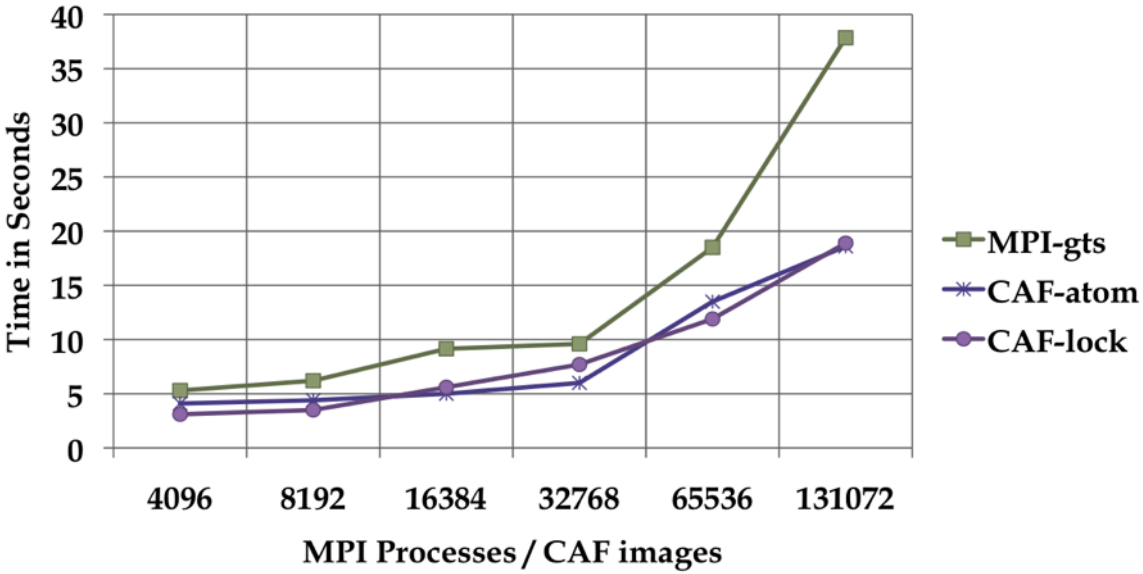
- UPC based on C philosophy / history
  - http://upc-lang.org
  - Free open source compiler: http://upc.lbl.gov
  - Also a gcc variant: http://www.gccupc.org
- Java dialect: Titanium
  - http://titanium.cs.berkeley.edu
- Co-Array Fortran
  - Part of Stanford Fortran (subset of features)
  - CAF 2.0 from Rice: http://caf.rice.edu
- Chapel from Cray (own base language better than Java)
  - http://chapel.cray.com (open source)
- X10 from IBM also at Rice (Java, Scala,…)
  - http://www.research.ibm.com/x10/
- Phalanx from Echelon projects at NVIDIA, LBNL,…
  - C++ PGAS languages with CUDA-like features for GPU clusters
- Coming soon…. PGAS for Python, aka PyGAS

Application Work in PGAS

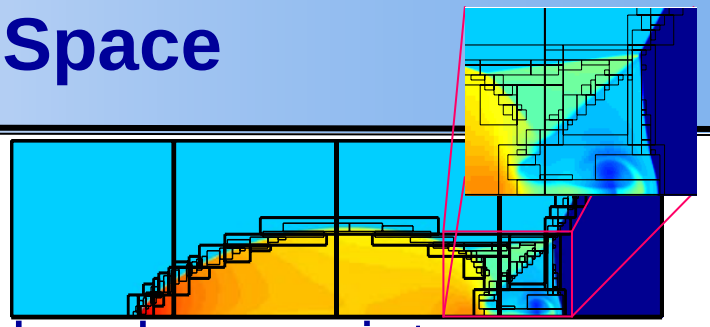
- Network simulator in UPC (Steve Hofmeyr, LBNL)
- Real-space multigrid (RMG) quantum mechanics (Shirley Moore, UTK)
- Landscape analysis, i.e., "Contributing Area Estimation" in UPC (Brian Kazian, UCB)
- GTS Shifter in CAF (Preissl, Wichmann, Long, Shalf, Ethier, Koniges, LBNL, Cray, PPPL)







Arrays in a Global Address Space

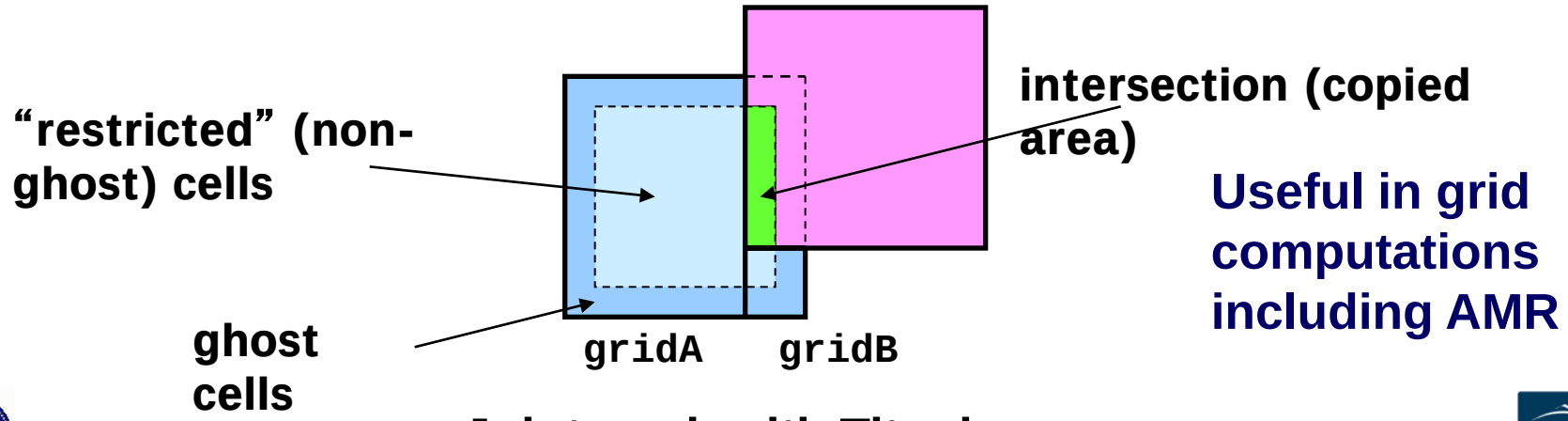


- Key features of Titanium arrays
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- Use domain calculus to identify ghosts and iterate:

```
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

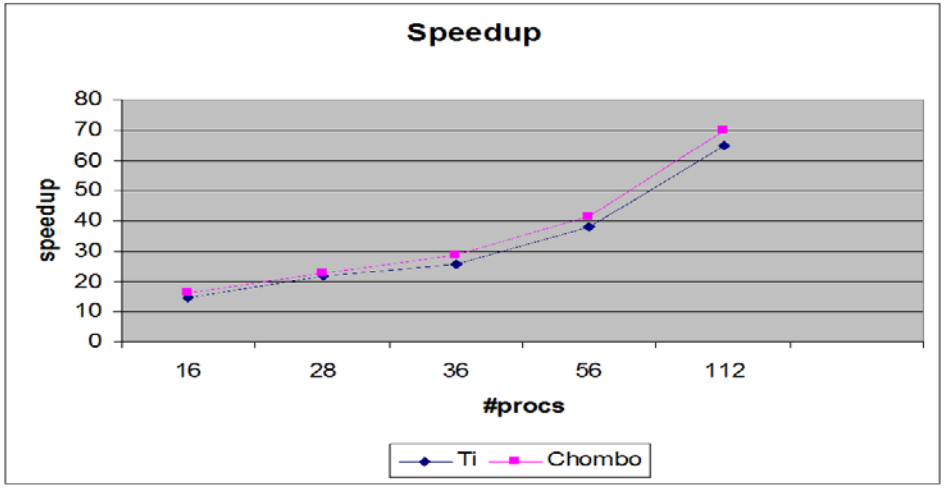
Languages Support Helps Productivity

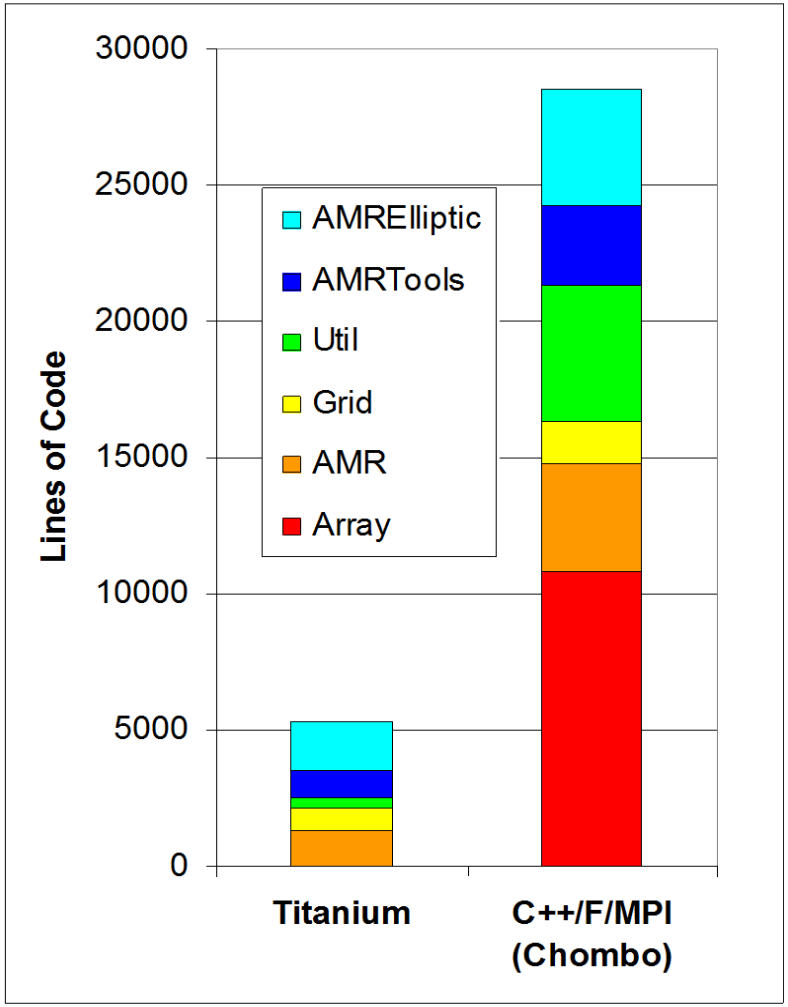
C++/Fortran/MPI AMR

- Chombo package from LBNL
- Bulk-synchronous comm:
  - Pack boundary data between procs
  - All optimizations done by programmer

Titanium AMR

- Entirely in Titanium
- Finer-grained communication
  - No explicit pack/unpack code
  - Automated in runtime system
- General approach
  - Language allow programmer optimizations
  - Compiler/runtime does some automatically



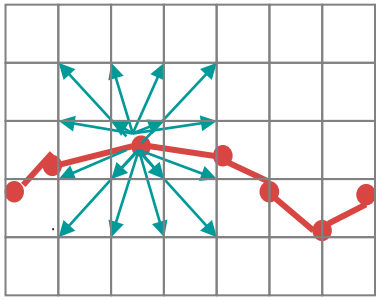


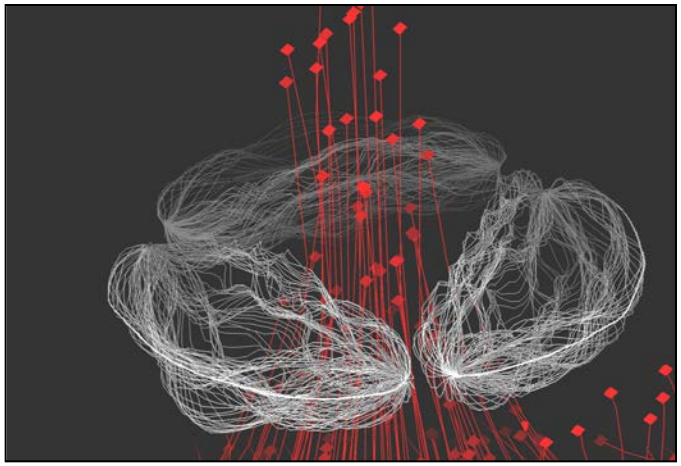
Work by Tong Wen and Philip Colella; Communication optimization

Particle/Mesh Method: Heart Simulation

- Elastic structures in an incompressible fluid.
  - Blood flow, clotting, inner ear, embryo growth, …
- Complicated parallelization
  - Particle/Mesh method, but "Particles" connected into materials (1D or 2D structures)
  - Communication patterns irregular between particles (structures) and mesh (fluid)

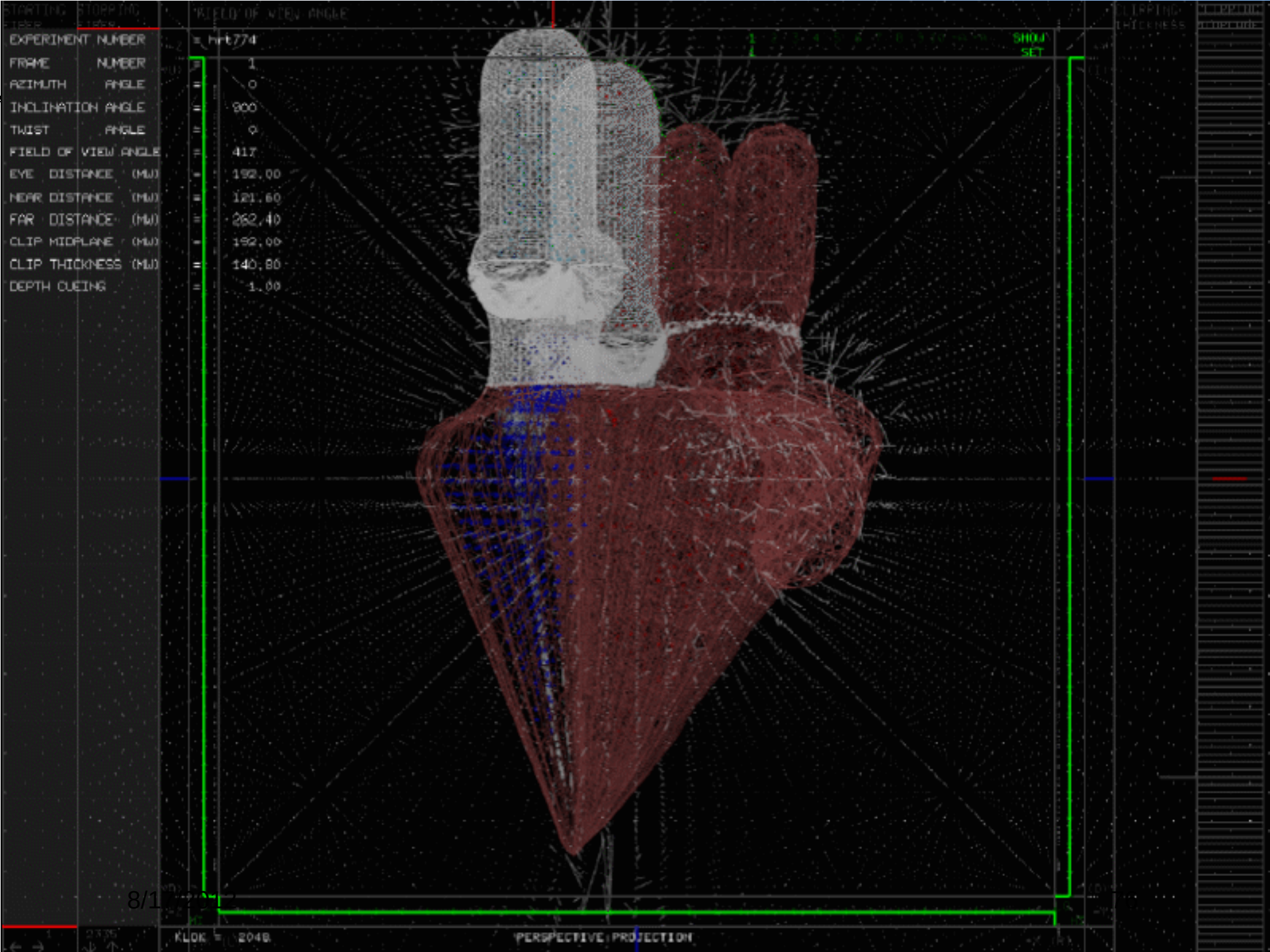
2D Dirac Delta Function



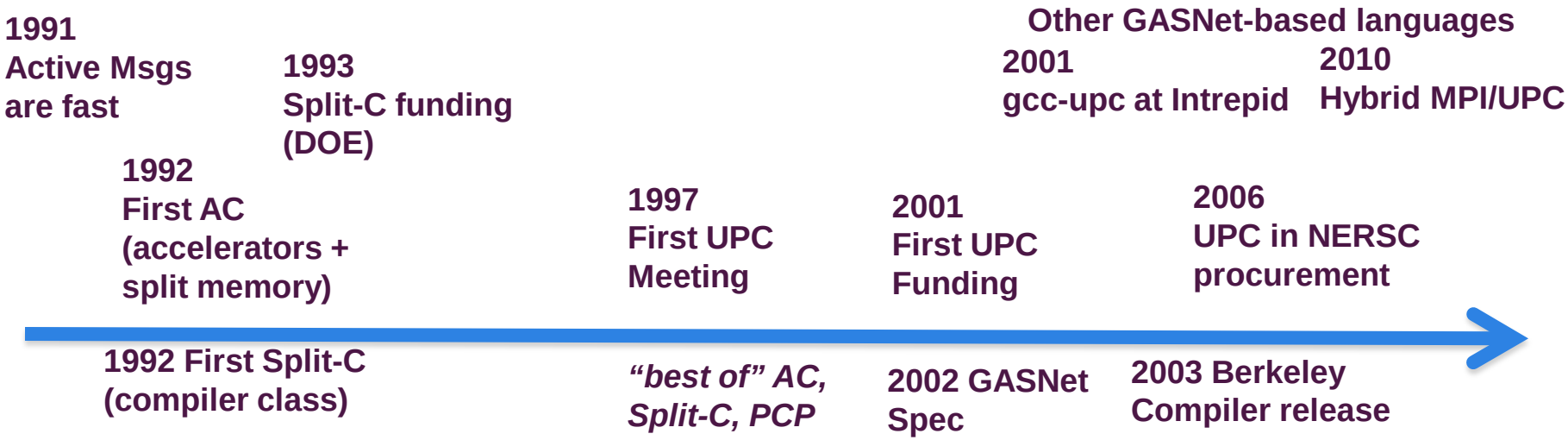


Code Size in Lines	
Fortran	Titanium
8000	4000

Note: Fortran code is not parallel


EXPERIMENT NUMBER = hrt774
FRAME NUMBER = 1
AZIMUTH ANGLE = 0
INCLINATION ANGLE = 900
TWIST ANGLE = 0
FIELD OF VIEW ANGLE = 417
EYE DISTANCE (MW) = 192.00
NEAR DISTANCE (MW) = 121.60
FAR DISTANCE (MW) = 262.40
CLIP MIDPLANE (MW) = 192.00
CLIP THICKNESS (MW) = 140.80
DEPTH CUEING = 1.00
SHOW SET
8/1
KLOK = 2048
PERSPECTIVE PROJECTION

Bringing Users Along: UPC Experience



- Ecosystem:
  - Users with a need (fine-grained random access)
  - Machines with RDMA (not full hardware GAS)
  - Common runtime; Commercial and free software
  - Sustained funding and Center procurements
- Success models:
  - Adoption by users: vectors → MPI, Python and Perl, UPC/CAF
  - Influence traditional models: MPI 1-sided; OpenMP locality control
  - Enable future models: Chapel, X10,…

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
  http://upc.gwu.edu
- Vendor compilers: Cray, IBM, HP, SGI,…