UPC (Unified Parallel C) Tutorial June 15, 2011

Based on Materials from Tutorial #S10 at SuperComputing 2010

by

Alice E. Koniges – NERSC, Lawrence Berkeley National Laboratory (LBNL)
Katherine Yelick – University of California, Berkeley and LBNL
Rolf Rabenseifner – High Performance Computing Center Stuttgart (HLRS), Germany
Reinhold Bader – Leibniz Supercomputing Centre (LRZ), Munich/Garching, Germany
David Eder – Lawrence Livermore National Laboratory
Filip Blagojevic, Robert Preissl and Paul H. Hargrove – LBNL

Outline

- **Basic PGAS concepts**
  - Execution model, memory model, resource mapping, …
  - Comparison with other paradigms
- **UPC basic syntax**
  - Declaration of shared data, synchronization
  - Dynamic objects, pointers, allocation
- **Advanced topics**
  - Locks and split-phase barriers, atomic procedures
  - Collective operations
  - Parallel patterns
- **Applications and Hybrid Programming**
  - As much as time permits…

- **BREAK near 3:00pm**

Basic PGAS Concepts

- Trends in hardware
- Execution model
- Memory model
- Comparison with other paradigms
- Run time environments
- PGAS application styles
- Resources

Summary of Hardware Trends

- **OBERVATIONS**
  - Moore’s Law maintained via Core count rather than Clock Speed
  - Concurrency has long been part of the HPC performance growth
  - Relative to Logic, Memory is getting slower and more costly
  - Movement of data between processors dominates energy budget
- **CONCLUSIONS:**
  - Nearly all future performance increases will be from concurrency
  - Energy is the key challenge in improving performance
  - Memory per floating point unit is shrinking

Programming model requirements

- **Control over layout and locality to minimize data movement**
- **Ability to share memory to minimize footprint**
- **Massive fine and coarse-grained parallelism**

Partitioned Global Address Space (PGAS) Languages

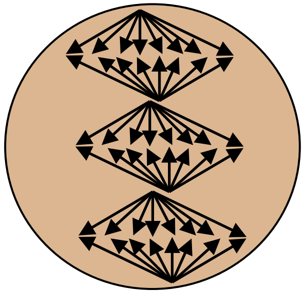
- **Coarray Fortran (CAF)**
  - Compilers from Cray, Rice and PGI (more soon)
- **Unified Parallel C (UPC)**
  - Compilers from Cray, HP, Berkeley/LBNL, Intrepid (gcc), IBM, SGI, MTU, and others
- **Titanium (Java based)**
  - Compiler from Berkeley

DARPA High Productivity Computer Systems (HPCS) language project:

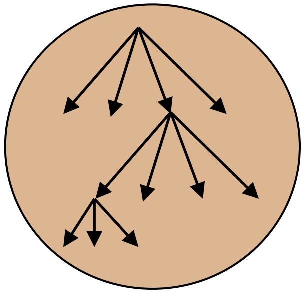
- **X10 (based on Java, IBM)**
- **Chapel (Cray)**
- **Fortress (SUN)**

Two Parallel Language Questions

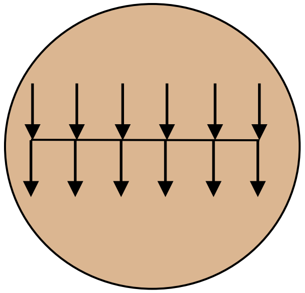
- What is the parallel control model?



data parallel
(singe thread of control)

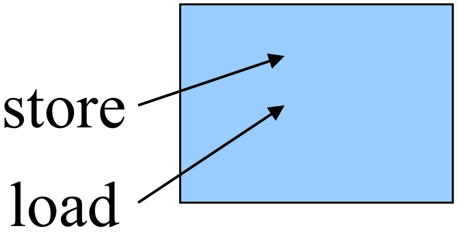


dynamic
threads

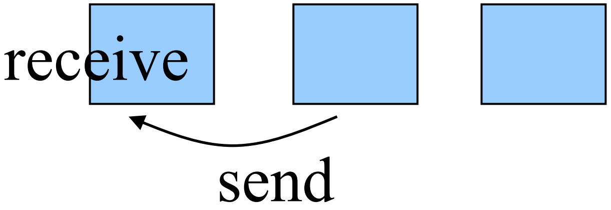


single program
multiple data (SPMD)

- What is the model for sharing/communication?



shared memory



message passing

implied synchronization for message passing, not shared memory

SPMD Execution Model

- **Single Program Multiple Data (SPMD) execution model**
  - Matches hardware resources: static number of threads for static number of cores ➔ no mapping problem for compiler/runtime
  - Intuitively, a copy of the main function on each processor
  - Similar to most MPI applications
- **A number of threads working independently in a SPMD fashion**
  - Number of threads given as program variable, e.g., `THREADS`
  - Another variable, e.g., `MYTHREAD` specifies thread index
  - There is some form of global synchronization, e.g., `upc_barrier`
  - Control flow (branches) are independent – not lock-step

- **UPC, CAF and Titanium all use a SPMD model**
- **HPCS languages, X10, Chapel, and Fortress do not**
  - They support dynamic threading and data parallel constructs

Data Parallelism – HPF

```
Real :: A(n,m), B(n,m)
!HPF$ DISTRIBUTE A(block,block), B(...)

do j = 2, m-1
  do i = 2, n-1
    B(i,j) =  ... A(i,j)
              ... A(i-1,j) ... A(i+1,j)
              ... A(i,j-1) ... A(i,j+1)
  end do
end do
```

→ Data definition

→ Loop over y-dimension

→ Vectorizable loop over x-dimension

→ Calculate B,
→ using upper and lower,
→ left and right value of A

- Data parallel languages use array operations (A = B, etc.) and loops
- Compiler and runtime map n-way parallelism to p cores
- Data layouts as in HPF can help with assignment using "owner computes"

- This mapping problem is one of the challenges in implementing HPF that does not occur with UPC or CAF

Dynamic Tasking - Cilk

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

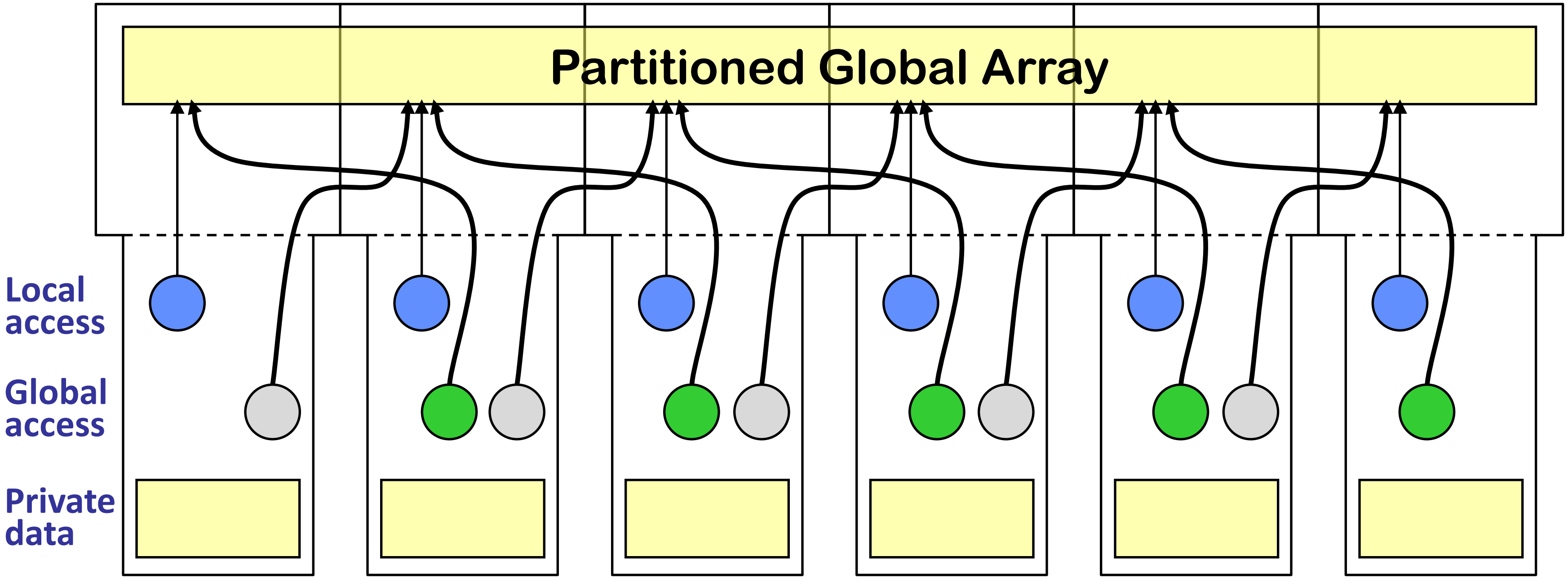
The computation dag and parallelism unfold dynamically.

processors are virtualized; no explicit processor number

- **Task parallel languages are typically implemented with shared memory**
- **No explicit control over locality; runtime system will schedule related tasks nearby or on the same core**
- **The HPCS languages support these in a PGAS memory model which yields an interesting and challenging runtime problem**

Partitioned Global Address Space (PGAS) Languages

- **Defining PGAS principles:**
  1) The *Global Address Space* memory model allows any thread to read or write memory anywhere in the system
  2) It is *Partitioned* to indicate that some data is local, whereas other date is further away (slower to access)

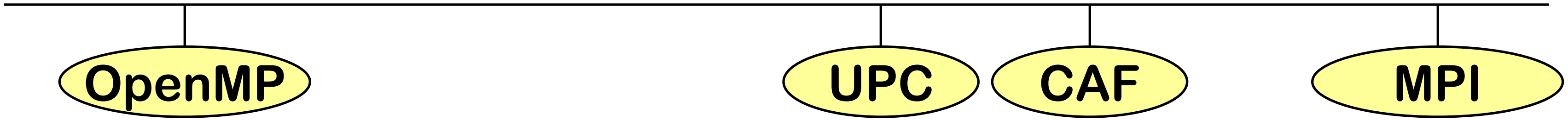


Two Concepts in the Memory Space

- **Private data: accessible only from a single thread**
  - Variable declared inside functions that live on the program stack are normally private to prevent them from disappearing unexpectedly
- **Shared data: data that is accessible from multiple threads**
  - Variables allocated dynamically in the program heap or statically at global scope may have this property
  - Some languages have both private and shared heaps or static variables (UPC is one of these)
- **Local pointer or reference: refers to local data**
  - Local may be associated with a single thread or a shared memory node
- **Global pointer or reference: may refer to “remote” data**
  - Remote may mean the data is off-thread or off-node
  - Global references are potentially remote; they *may* refer to local data

Other Programming Models

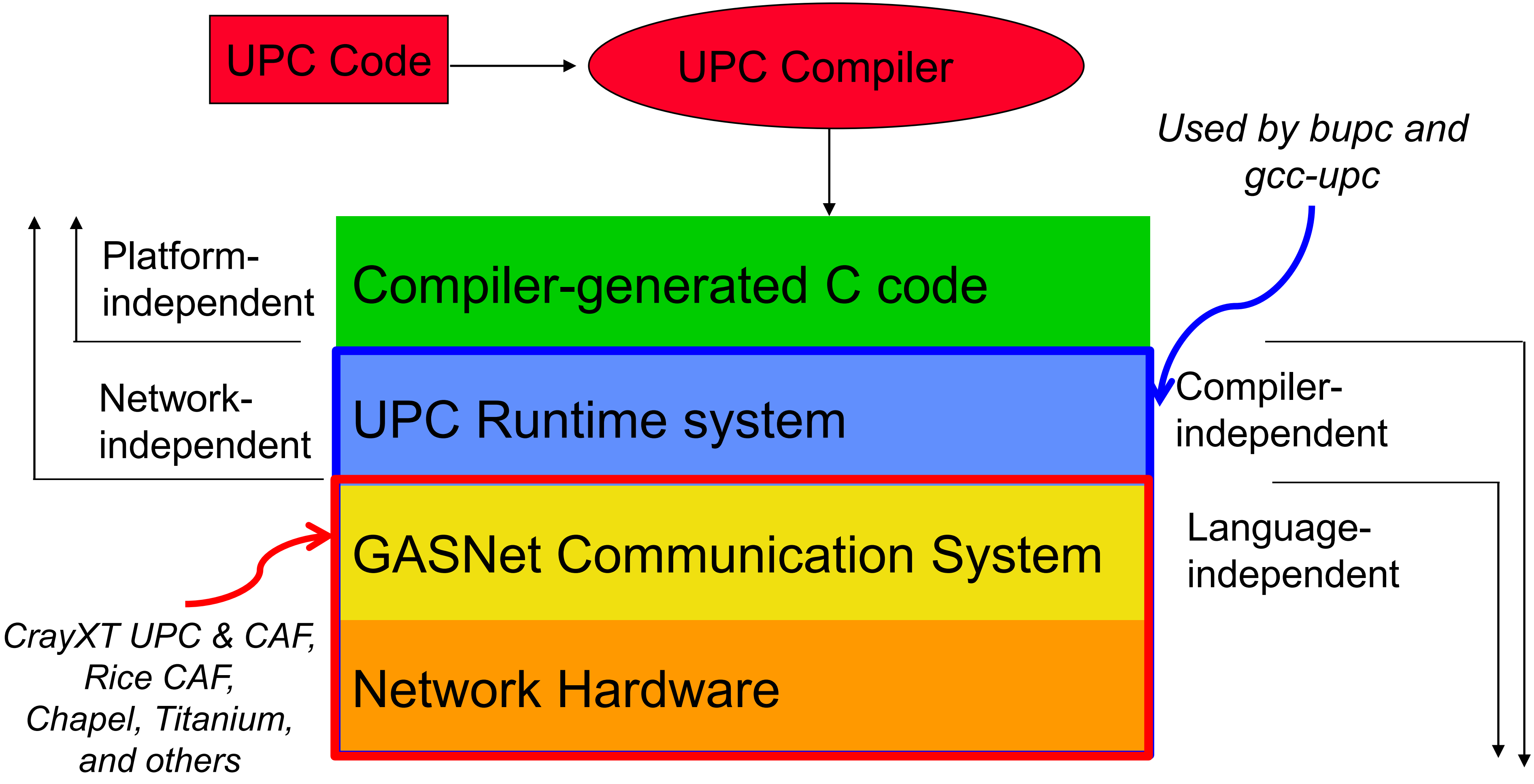
- **Message Passing Interface (MPI)**
  - Library with message passing routines
  - Unforced locality control through separate address spaces
- **OpenMP**
  - Language extensions with shared memory worksharing directives
  - Allows shared data structures without locality control



OpenMP UPC CAF MPI

- **UPC / CAF data accesses:**
  - Similar to OpenMP but with locality control
- **UPC / CAF worksharing:**
  - Similar to MPI

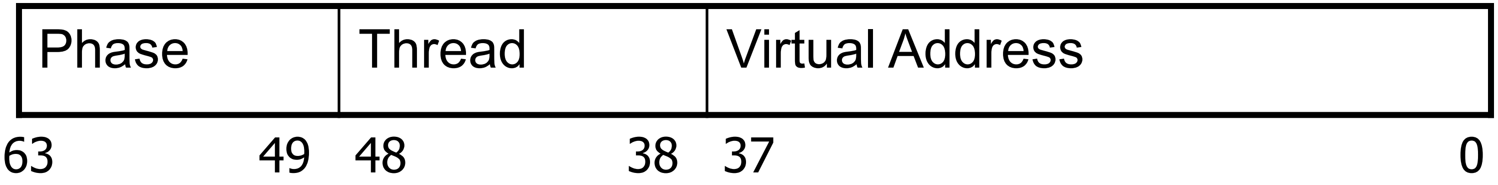
Understanding Runtime Behavior - Berkeley UPC Compiler



UPC Pointers

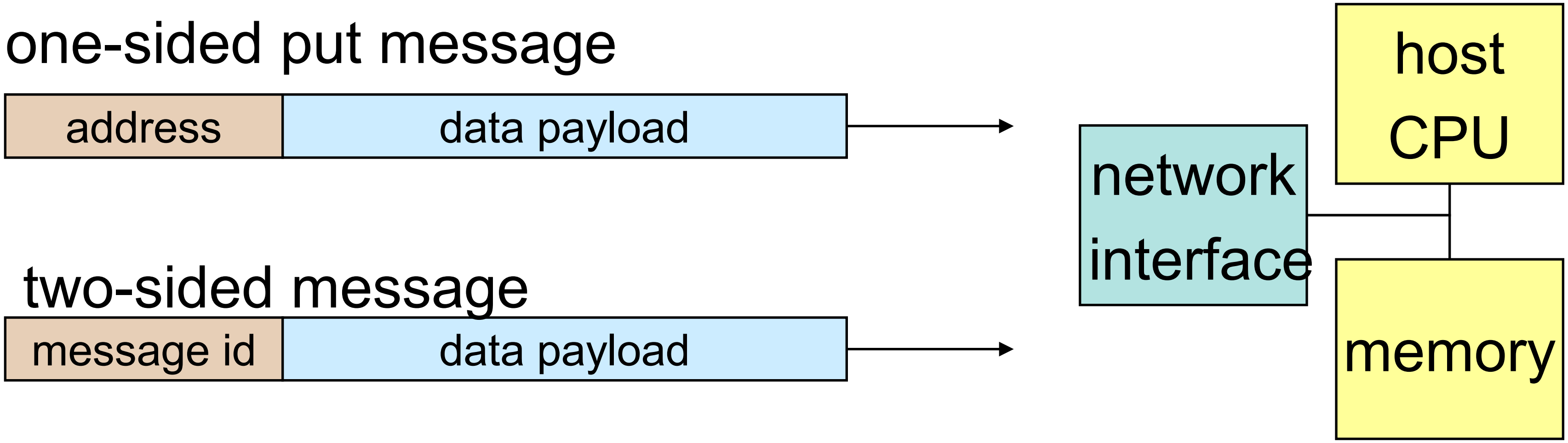
- **UPC pointers to shared objects have (conceptually) three fields:**
  - thread number
  - local address of block
  - phase (specifies position in the block) so that pointer arithmetic operations (like ++) move through the array correctly

- **Example implementation**



Phase	Thread	Virtual Address
63 – 49	48 – 38	37 – 0

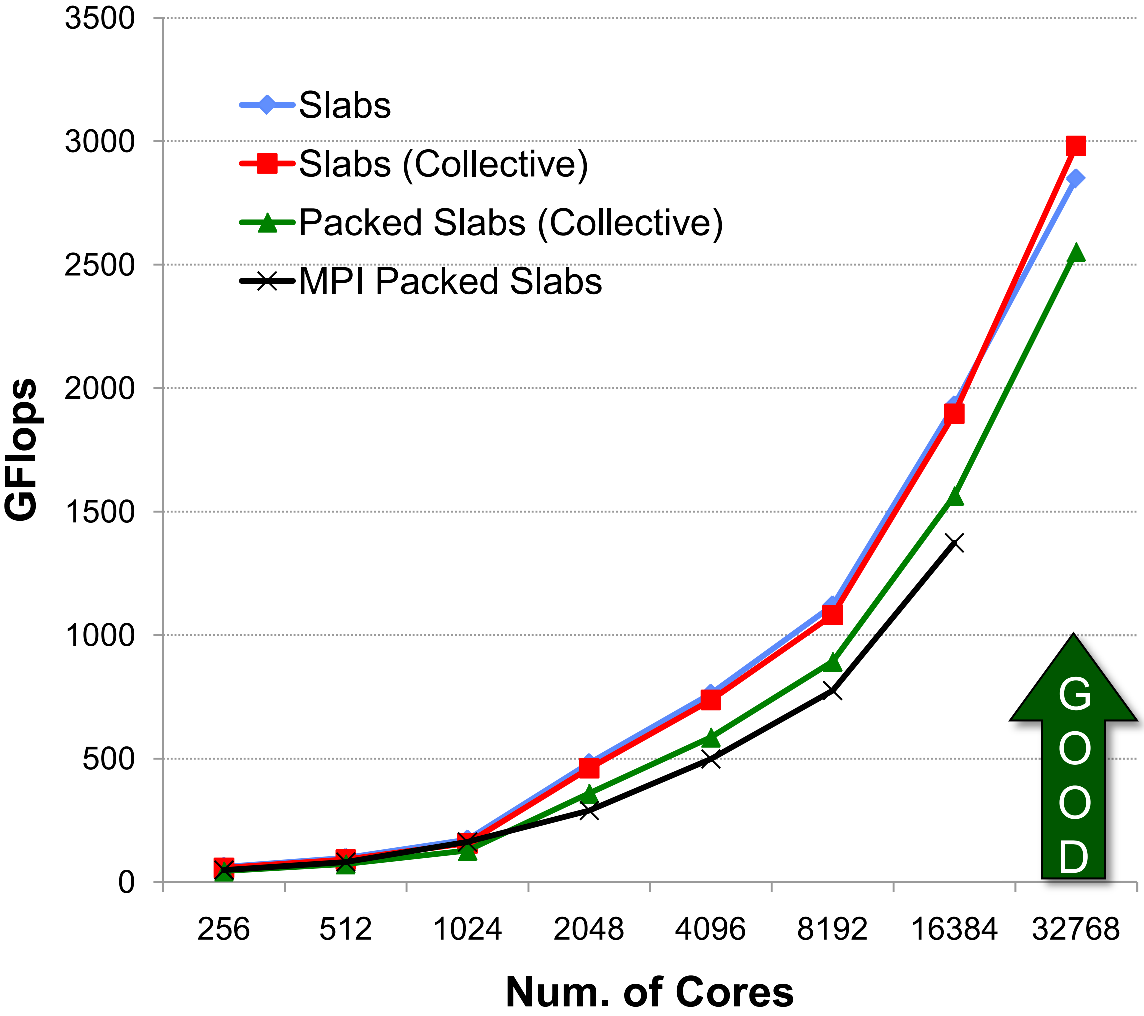
One-Sided vs Two-Sided Communication



- **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

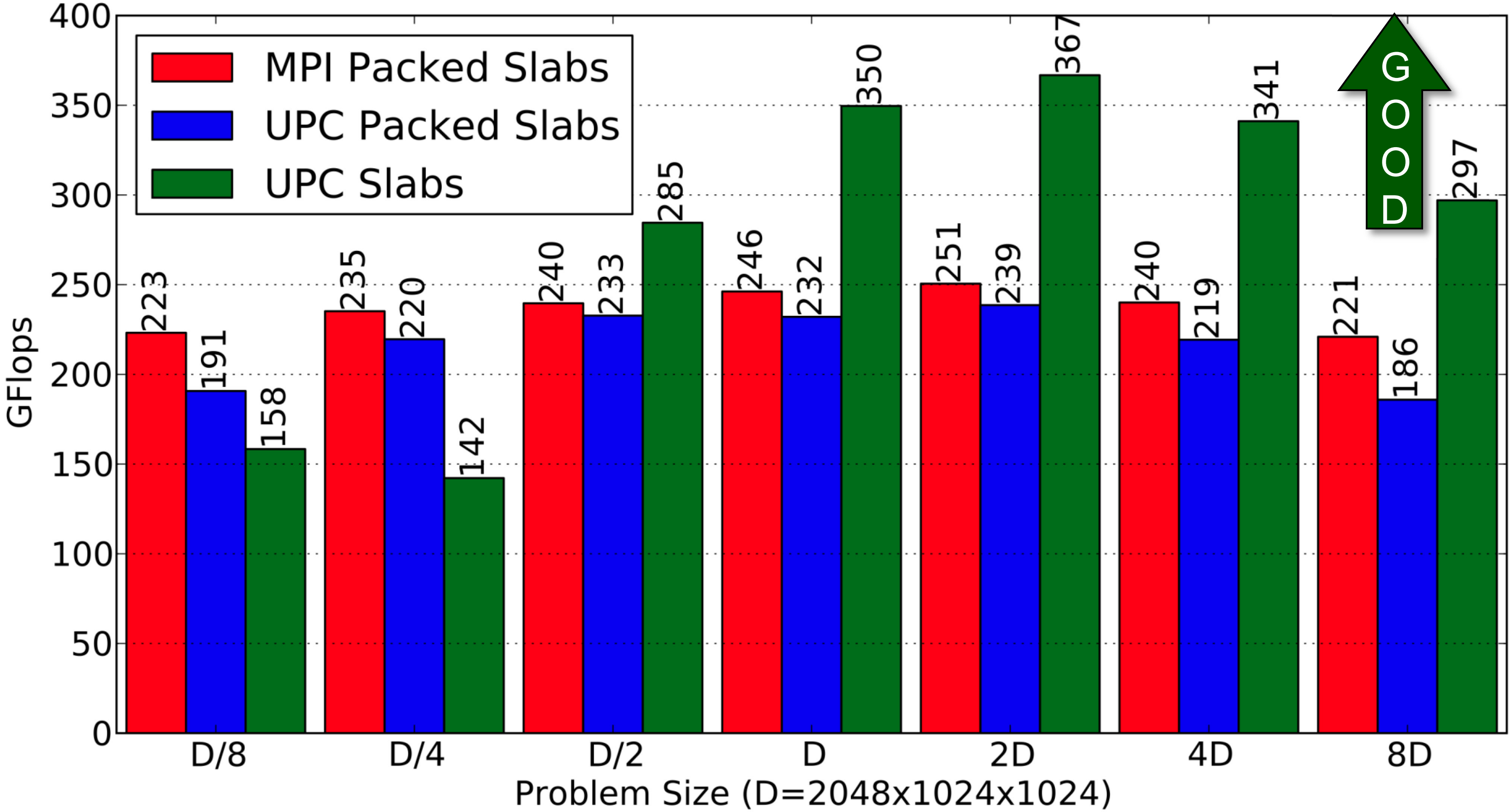
FFT Performance on BlueGene/P

- Three UPC implementations consistently outperform MPI
- Leveraging communication/computation overlap yields best performance
  - More collectives in flight and more communication leads to better performance
  - At 32k cores, overlap algorithms yield 17% improvement in overall application time



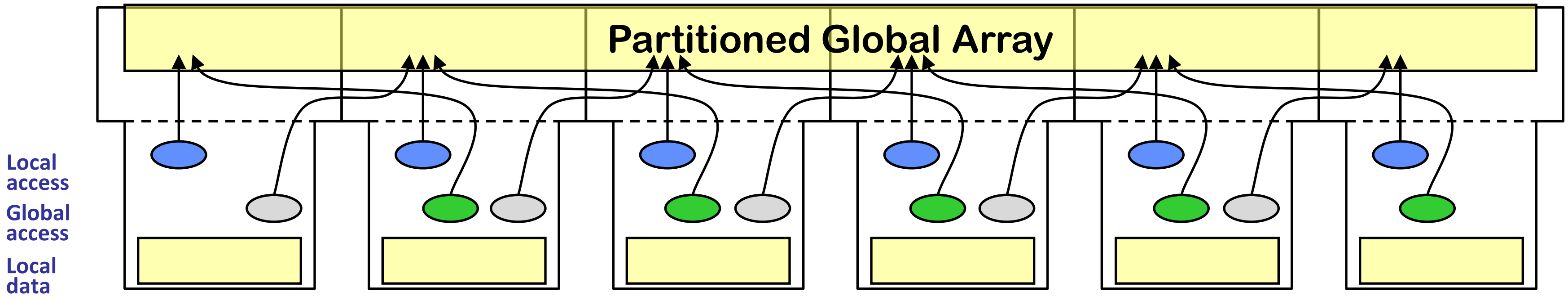
FFT Performance on Cray XT4

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



Programming styles with PGAS

- **Data is partitioned among the processes, i.e.,** without halos
  - Fine-grained access to the neighbor elements when needed
  - ➢ Compiler expected to implement automatically (and together)
    - ▪ pre-fetches
    - ▪ bulk data transfer (instead of single-word remote accesses)
  - ➢ May be very slow if compiler's optimization fails
- **Application implements** halo **storage**
  - Application organizes halo updates with bulk data transfer
  - ➢ Advantage: High speed remote accesses
  - ➢ Drawbacks: Additional memory accesses and storage needs



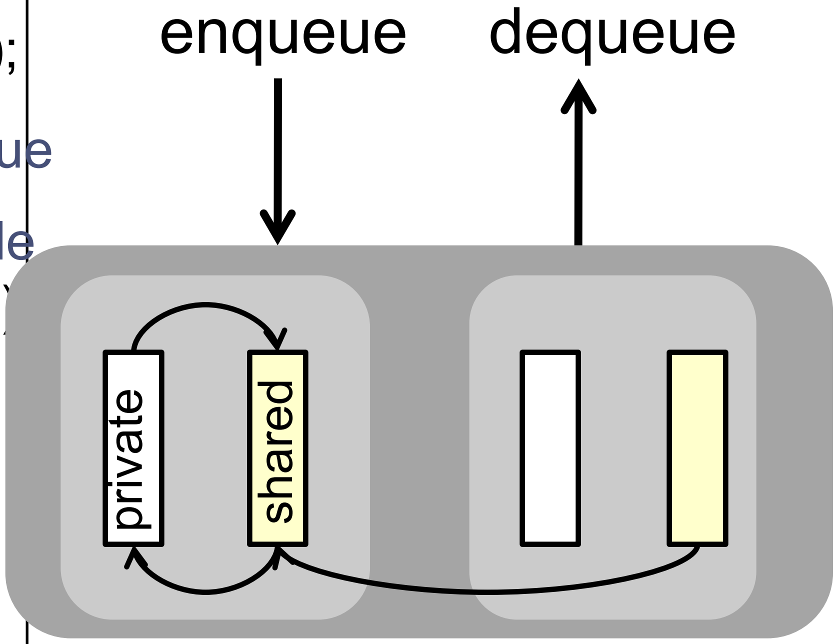
Irregular Applications

- **The SPMD model is too restrictive for some “irregular” applications**
  - The global address space handles irregular *data* accesses:
    - Irregular in space (graphs, sparse matrices, AMR, etc.)
    - Irregular in time (hash table lookup, etc.): for reads, UPC handles this well; for writes you need atomic operations
  - Irregular *computational* patterns are more difficult:
    - Not statically load balanced (even with graph partitioning, etc.)
    - Some kind of dynamic load balancing needed (e.g. a task queue)
- **Design considerations for dynamic scheduling UPC**
  - For locality reasons, SPMD still appears to be best for regular applications; aligns threads with memory hierarchy
  - UPC serves as “abstract machine model” so dynamic load balancing as an add-on may be written in portable UPC

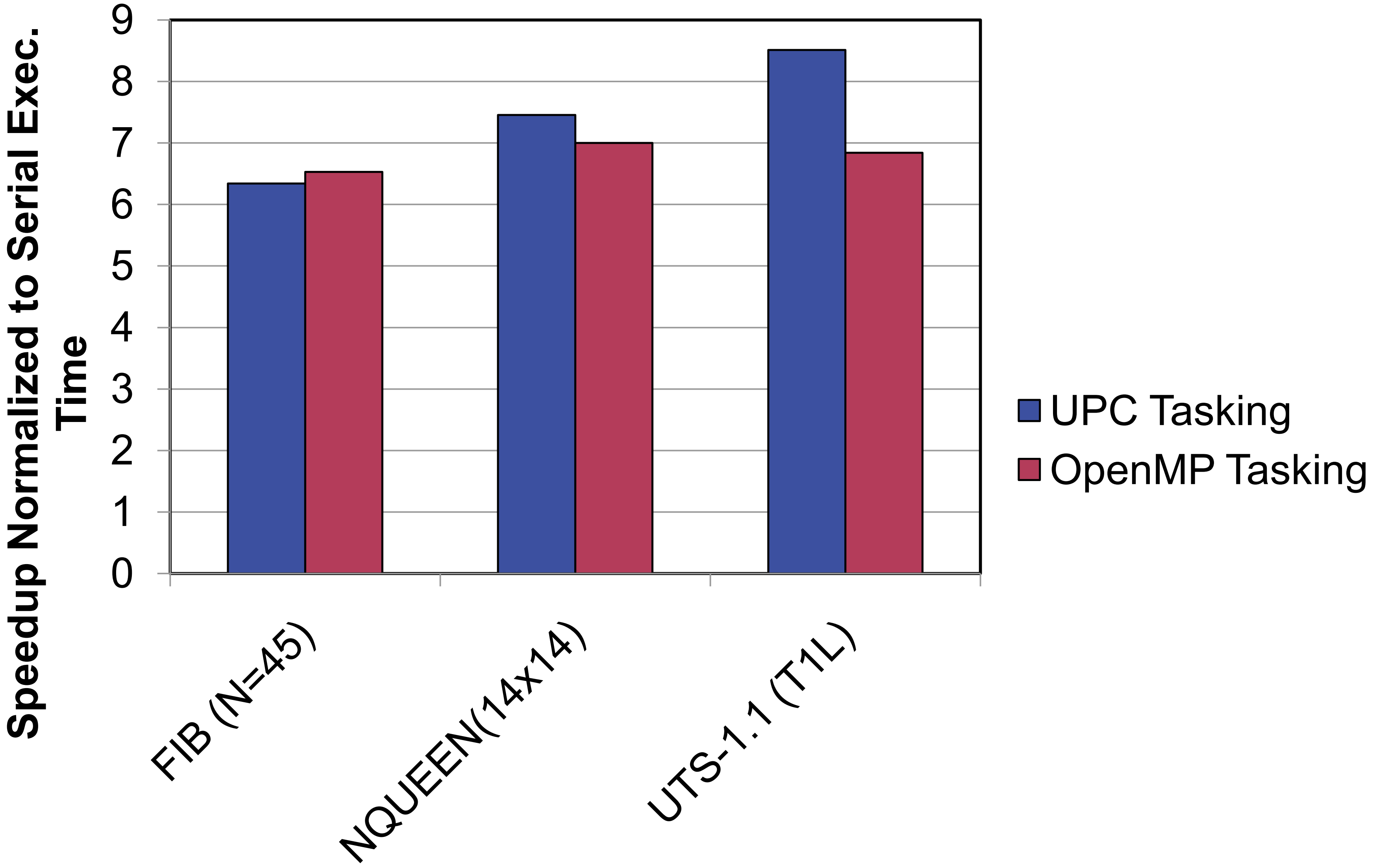
Distributed Tasking API for UPC
(a work in progress)

```
// allocate a distributed task queue
taskq_t * all_taskq_alloc();

// enqueue a task into the distributed queue
int taskq_put(upc_taskq_t *, upc_task_t*);

// dequeue a task from the local task queue

// returns null if task is not readily available
int taskq_get(upc_taskq_t *, upc_task_t *

// test whether queue is globally empty
int  taskq_isEmpty(bupc_taskq_t *);

// free distributed task queue memory
int  taskq_free(shared bupc_taskq_t *);
```

internals are hidden from user, except that dequeue operations may fail and provide hint to steal

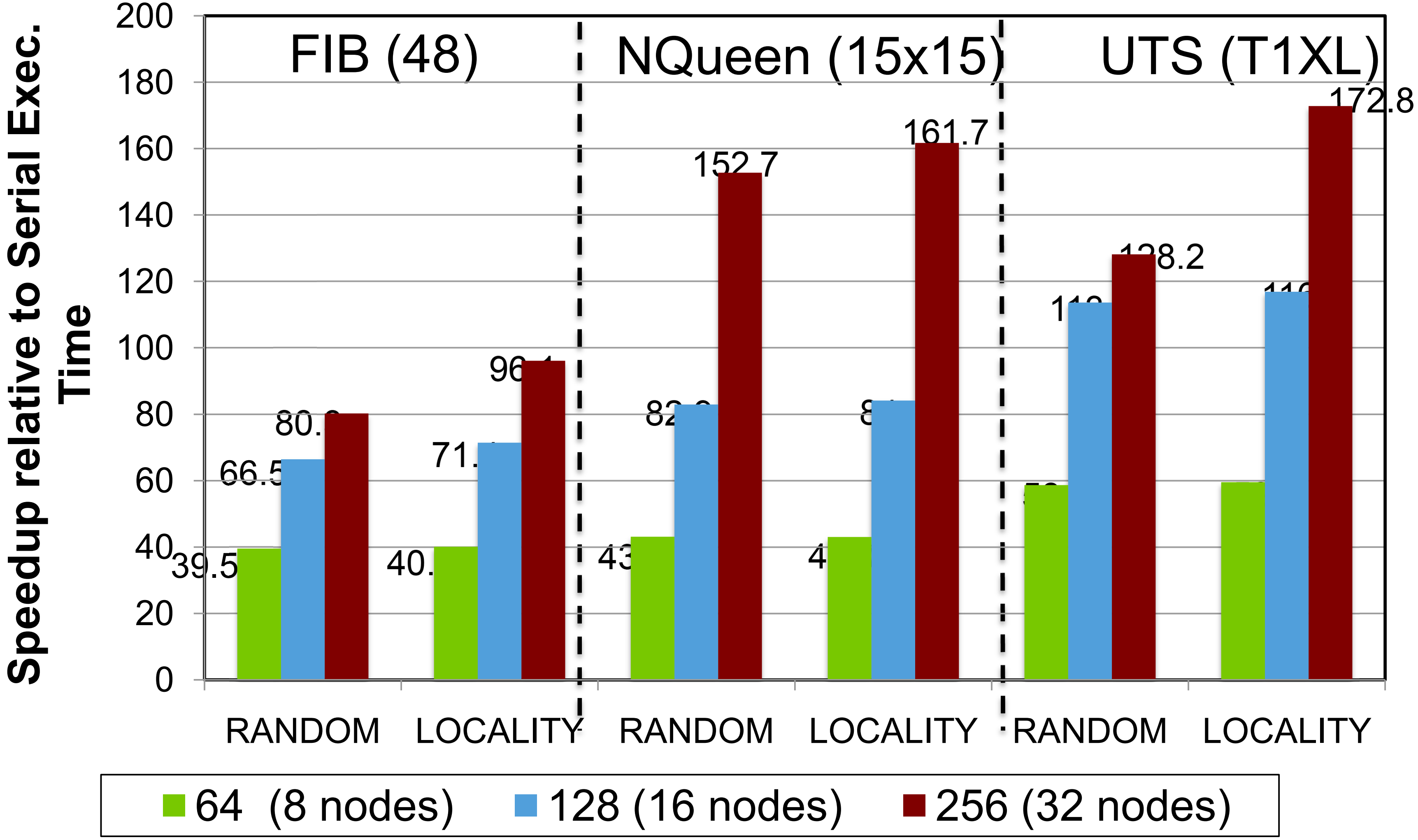


UPC Tasking on 8-core Nehalem node



Multi-Core Cluster Performance
Random vs. Locality-aware work-stealing

Speedup 16.5 % 5.6% 25.9%



Support

- **PGAS in general**
  - http://en.wikipedia.org/wiki/PGAS
  - http://www.pgas-forum.org/ → PGAS conferences
- **UPC**
  - http://en.wikipedia.org/wiki/Unified_Parallel_C
  - http://upc.gwu.edu/ → Main UPC homepage
  - http://upc.gwu.edu/documentation.html → Language specs
  - http://upc.gwu.edu/download.html → UPC compilers

UPC Basic Syntax

- Declaration of shared data
- Handling shared data and work sharing
- Synchronization:
  - motivation – race conditions;
  - rules for access to shared data by different threads
- Dynamic data and their management:
  - UPC pointers and allocation calls

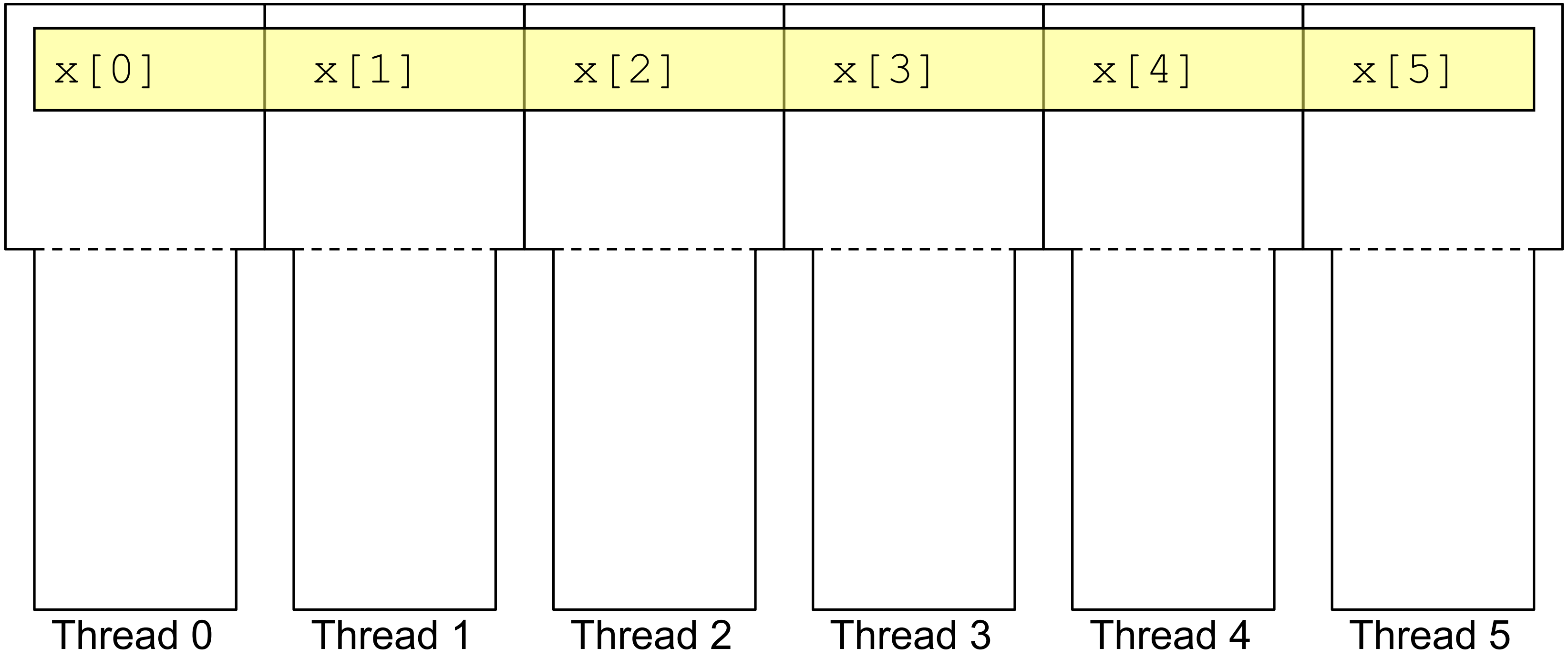
Example 0

```
#include <upc.h>
#include <stdio.h>
shared int x[THREADS];
int main(int argc, char** argv)
{
  x[MYTHREAD] = MYTHREAD;
  if (MYTHREAD == 0)
     printf("hello world\n");
  printf("I am thread number %d of %d threads\n",
                       MYTHREAD, THREADS);
  if (MYTHREAD > 0)
     printf("I see x[0] = %d\n", x[0]);
  return 0;
}
```

- **Shows a generalization of a classic first C program**
- **Contains a bug – can you spot it?**

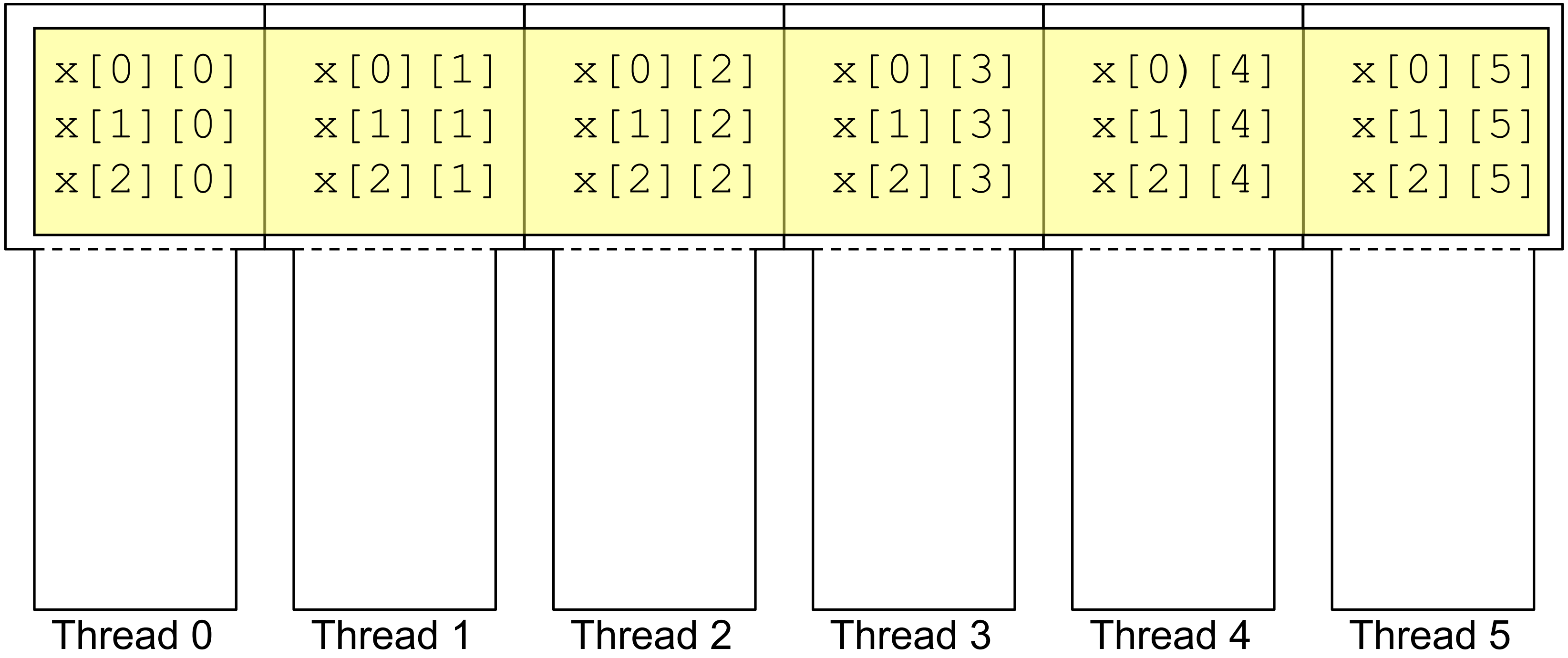
Distributed 1D Array

- **Declaration:**
  - `shared float x[THREADS];` // **statically allocated outside of functions**
- **Data distribution:**



Distributed 2D Array

- **Declaration:**
  - `shared float x[3][THREADS];` // **statically allocated outside of functions**
- **Data distribution:**



x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0) [4]	x[0][5]
x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]	x[1][5]
x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]	x[2][5]
Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5

Distributed arrays with UPC

- UPC shared objects may be statically allocated
- Definition of shared data:
  - **shared** [**blocksize**] type variable_name;
  - **shared** [**blocksize**] type array_name[dim1];
  - **shared** [**blocksize**] type array_name[dim1][dim2];
  - …
- Default: blocksize=1 if no “[…]” given (different from “[]” which we see later)
- The distribution is round robin with chunks of **blocksize** elements
- Blocked distribution is achieved if last dimension==THREADS and blocksize==1

the dimensions define which elements exist

See next slides

UPC shared data – examples

```
shared [1] float a[20];  // or
shared     float a[20];
```

a[0] a[4] a[8] a[12] a[16]	a[1] a[5] a[9] a[13] a[17]	a[2] a[6] a[10] a[14] a[18]	a[3] a[7] a[11] a[15] a[19]
Thread 0	Thread 1	Thread 2	Thread 3

```
shared [1] float a[5][THREADS];
// or
shared     float a[5][THREADS];
```

a[0][0] a[1][0] a[2][0] a[3][0] a[4][0]	a[0][1] a[1][1] a[2][1] a[3][1] a[4][1]	a[0][2] a[1][2] a[2][2] a[3][2] a[4][2]	a[0][3] a[1][3] a[2][3] a[3][3] a[4][3]
Thread 0	Thread 1	Thread 2	Thread 3

```
shared [5] float a[20];  // or
define N 20
shared [N/THREADS] float a[N];
```

a[0] a[1] a[2] a[3] a[4]	a[5] a[6] a[7] a[8] a[9]	a[10] a[11] a[12] a[13] a[14]	a[15] a[16] a[17] a[18] a[19]
Thread 0	Thread 1	Thread 2	Thread 3

THREADS=1st dim!

identical at compile time

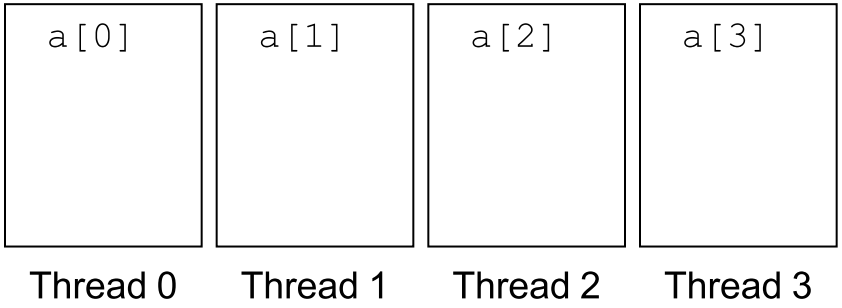
```
shared [5] float a[THREADS][5];
```

a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]	a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]	a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]	a[3][0] a[3][1] a[3][2] a[3][3] a[3][4]
Thread 0	Thread 1	Thread 2	Thread 3

Courtesy of Andrew Johnson

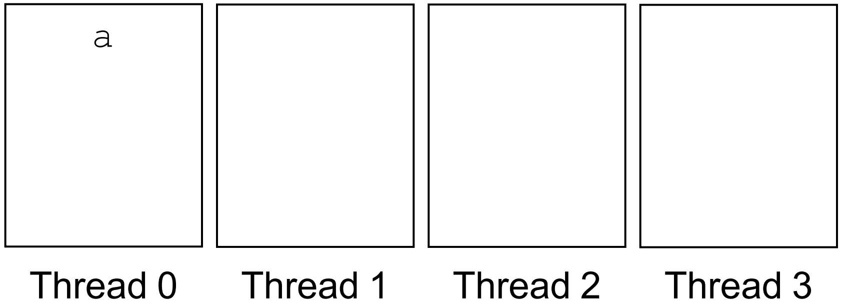
UPC shared data – examples (continued)

```
shared float a[THREADS]; // or
shared [1] float a[THREADS];
```



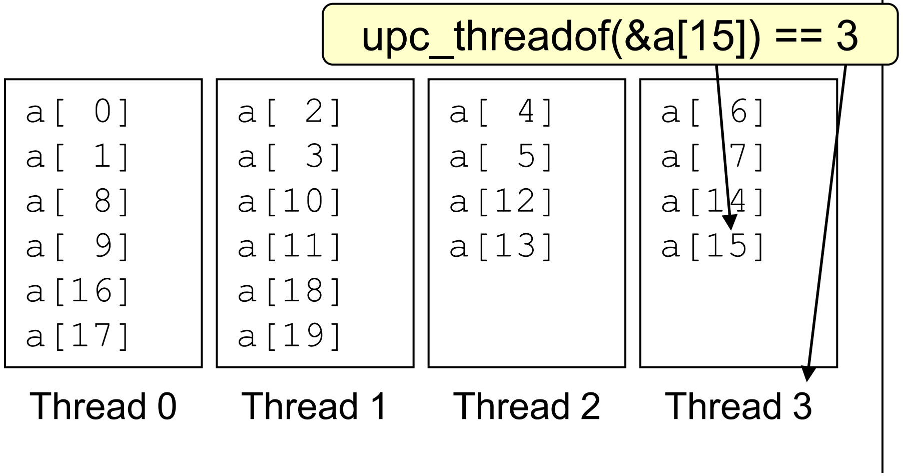
Thread 0	Thread 1	Thread 2	Thread 3
a[0]	a[1]	a[2]	a[3]

```
shared float a;
// located on thread 0
```



Thread 0	Thread 1	Thread 2	Thread 3
a			

```
shared [2] float a[20];
```

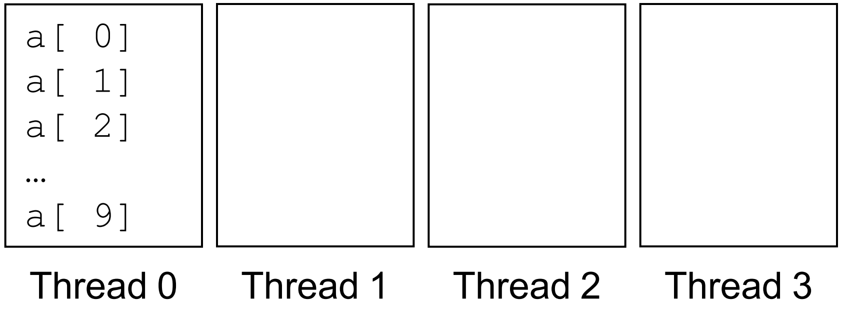


upc_threadof(&a[15]) == 3

Thread 0	Thread 1	Thread 2	Thread 3
a[0] a[1] a[8] a[9] a[16] a[17]	a[2] a[3] a[10] a[11] a[18] a[19]	a[4] a[5] a[12] a[13]	a[6] a[7] a[14] a[15]

Blank blocksize → located entirely on thread 0

```
shared [ ] float a[10]; //or [0]
```



Thread 0	Thread 1	Thread 2	Thread 3
a[0] a[1] a[2] … a[9]			

Courtesy of Andrew Johnson

Integration with the type system
(static type components)

```
typedef struct {
  float mass;
  float coor[3];
  float velocity[3];
} Body;
```

declare and use entities of this type:

```
shared [100] Body asteroids[THREADS][100];
Body s;
⋮
if (MYTHREAD == 1) {
  s = asteroids[0][4];
}
```

– compare this with effort needed implement the same with MPI (dispense with **all** of `MPI_TYPE_*` API)

– what about dynamic type components? → later in this talk

Local access to local part of distributed variables

```
shared float x[THREADS];
float *x_local;
⋮
x_local = (float *) &x[MYTHREAD];
```

- **`*x_local` now equals `x[MYTHREAD]`**
- **Can be used in its place for**
  - Clearer code
  - More efficient code
  - Passing to C libraries (e.g. standard numerical libraries)

Querying affinity

`upc_threadof(address)` yields thread containing that location

```
// Examples from an earlier slide
shared [1] float a[5][THREADS];
shared [5] float b[THREADS][5];

for (int i=0; i<5; ++i)
  for (int j=0; j<THREADS; ++j)
    assert(upc_threadof(&a[i][j]) == j);

for (int i=0; i<THREADS; ++i)
  for (int j=0; j<5; ++j)
    assert(upc_threadof(&b[i][j]) == i);
```

Work sharing:
naïve versions of „owner computes“

```
// generic for any array layout
for (int i=0; i<N; ++i)
  if (upc_threadof(&a[i]) == MYTHREAD)
    { /* do work on a[i] */ }
```

```
shared [1] int a[N]; // cyclic array layout
for (int i=MYTHREAD; i<N; i+=THREADS)
  { /* do work on a[i] */ }
```

```
shared [*] int a[N]; // blocked array layout – see below
int bs = (N+THREADS-1)/THREADS;
for (int i=bs*MYTHREAD; i<bs*(MYTHREAD+1); ++i)
  { /* do work on a[i] */ }
```

- All three loops iterate over the elements that are local to each thread
- New syntax introduced in blocked example:

  ```
  shared [*] int a[N];
  ```

  is equivalent to

  ```
  shared [(N+THREADS-1)/THREADS] int a[N];
  ```

  and is valid for arrays but not for pointers (which have no N).

Work sharing with `upc_forall`

```
// generic for any array layout – naive version again
for (int i=0; i<N; ++i)
  if (upc_threadof(&a[i]) == MYTHREAD)
    { /* do work on a[i] */ }
```

```
// generic for any array layout using upc_forall
upc_forall (int i=0; i<N; ++i; &a[i])
  { /* do work on a[i] */ }
```

```
// valid/equivalent for cyclic array layout only:
upc_forall (int i=0; i<N; ++i; i)
  { /* do work on a[i] */ }
```

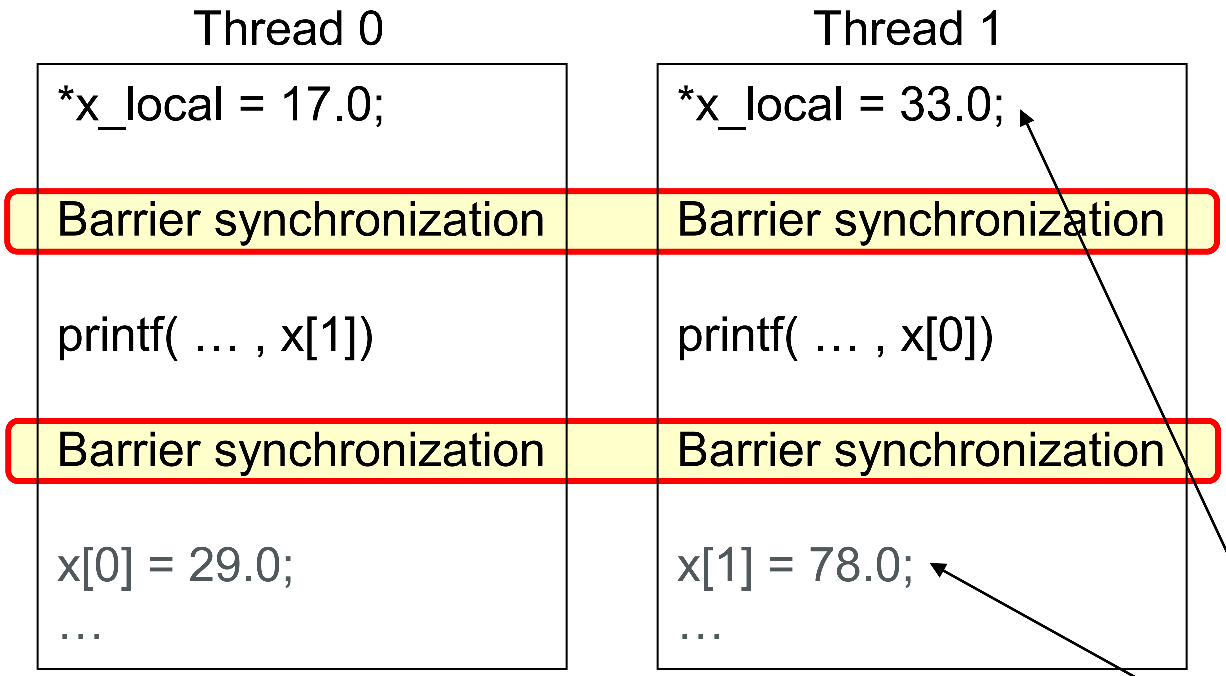
Notice that all 3 versions iterate `i` the over *full* range 0…(N-1).

Fourth expression in `upc_forall` is the "affinity expression"

- **Type is shared address → execute a given interation**
  `if (upc_threadof(expr) == MYTHREAD)`
- **Type is integer → execute a given iteration**
  `if ((expr % THREADS) == MYTHREAD)`

Special rules for nesting – you should avoid nesting `upc_forall`

Parallel execution with access epochs a.k.a synchronization phases



Thread 0

```
*x_local = 17.0;
```

Barrier synchronization

```
printf( … , x[1])
```

Barrier synchronization

```
x[0] = 29.0;
…
```

Thread 1

```
*x_local = 33.0;
```

Barrier synchronization

```
printf( … , x[0])
```

Barrier synchronization

```
x[1] = 78.0;
…
```

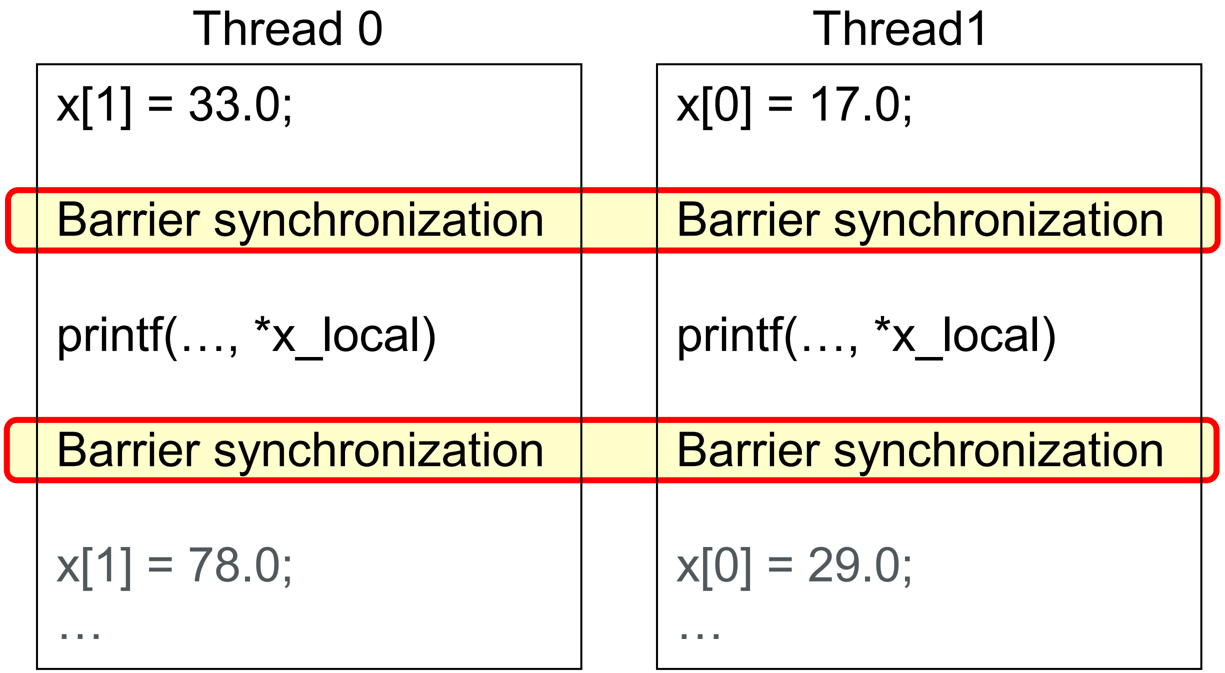
1. Local accesses to shared data
2. Barrier until all processes have finished their local accesses
3. Remote accesses
4. Barrier until all processes have finished their remote accesses
5. Local accesses to shared data

Both notations are equivalent

Barrier synchronization is required to ensure

- Local writes in step 1 precede remote reads in step 3
- Remote reads in step 3 precede local writes in step 5

Parallel execution – same with remote write / local read



Thread 0	Thread1
x[1] = 33.0;	x[0] = 17.0;
Barrier synchronization	Barrier synchronization
printf(…, *x_local)	printf(…, *x_local)
Barrier synchronization	Barrier synchronization
x[1] = 78.0; …	x[0] = 29.0; …

Previous example with local/remote exchanged:
Barrier synchronization is required to ensure

- Remote writes in step 1 precede local reads in step 3
- Local reads in step 3 precede remote writes in step 5

Synchronization

- Between a **write access** and a (subsequent or preceding) **read or write access** of the **same data** from **different processes**, a synchronization of the processes must be done!

Otherwise race condition!

- Most simple synchronization:
  → **barrier between all processes**
- Simple:

```
Accesses to distributed data by some/all processes
upc_barrier;
Accesses to distributed data by some/all processes
```

- Split-phase:

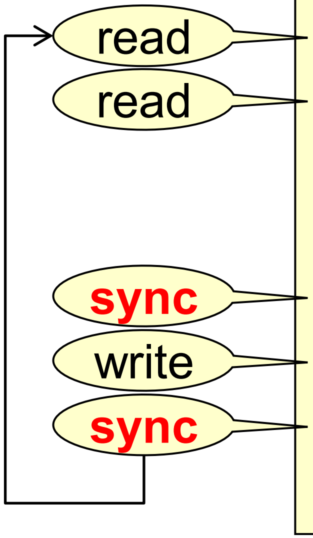
```
Accesses to distributed data by some/all processes
upc_notify;
// do work that does not require synchronization
upc_wait;
Accesses to distributed data by some/all processes
```

Example 0 revisited

```
#include <upc.h>
#include <stdio.h>
shared int x[THREADS];
int main(int argc, char** argv)
{
  x[MYTHREAD] = MYTHREAD;
  upc_notify;
  if (MYTHREAD == 0)
     printf("hello world\n");
  printf("I am thread number %d of %d threads\n",
                       MYTHREAD, THREADS);
  upc_wait;
  if (MYTHREAD > 0)
     printf("I see x[0] = %d\n", x[0]);
  return 0;
}
```

- **Repairs the bug (data race) present in the original example**

Another Example:



```
shared float x[THREADS];
while (!converged(x)) {
  float tmp = 0.5*x[MYTHREAD]
              - 0.25*x[(MYTHREAD-1)%THREADS]
              - 0.25*x[(MYTHREAD+1)%THREADS];
  upc_barrier;
  x[MYTHREAD] = tmp;
  upc_barrier;
}
```

Note that real applications must do more work between synchronizations or performance would be horrible.

Dynamic entities: Pointers

- **Remember C pointer semantics**

```
<type> *ptr;

ptr = &var;       // ptr holds address of var
```

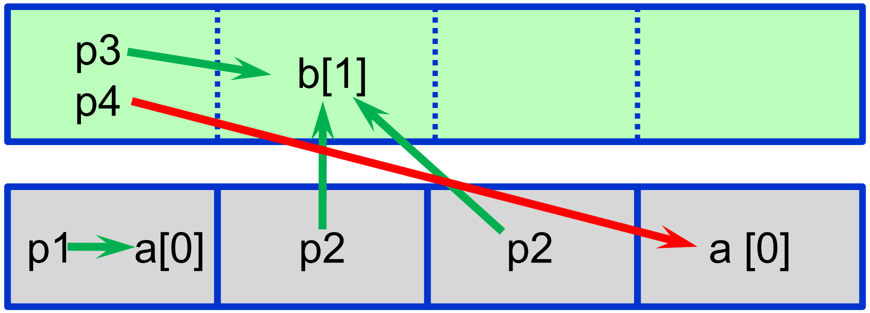
Topics:
pointer arithmetic
pointer-to-pointer
pointer-to-void / recast

- **Pointers and PGAS memory categorization**
  - Both pointer (ptr) and pointee (var) may be either private or shared
    → **4 combinations** theoretically possible
  - In UPC **three** of these combinations are useful in practice

Pointers continued …

```
// variables in private space: p1, p2 and a
int *p1;
shared int *p2;
int a[N];

// variables in shared space: p3, p4 and b
shared int *shared p3;
int *shared p4;
shared int b[N];
```

issue:
Where does p4 point?
Other threads must not rereference p4.



Pointer-to-shared: local views of shared data (review)

- **Cast from a pointer-to-shared to a pointer-to-private:**

```
shared float a[5][THREADS];
float *a_local;

a_local = (float *) &a[0][MYTHREAD];

a_local[0] is identical with a[0][MYTHREAD]
a_local[1] is identical with a[1][MYTHREAD]
…
a_local[4] is identical with a[4][MYTHREAD]
```

address must have affinity to **local** thread

pointer arithmetic selects local part

- **May have performance advantages**
- **May improve code readability**
- **Required when passing to non-UPC numerical libraries**

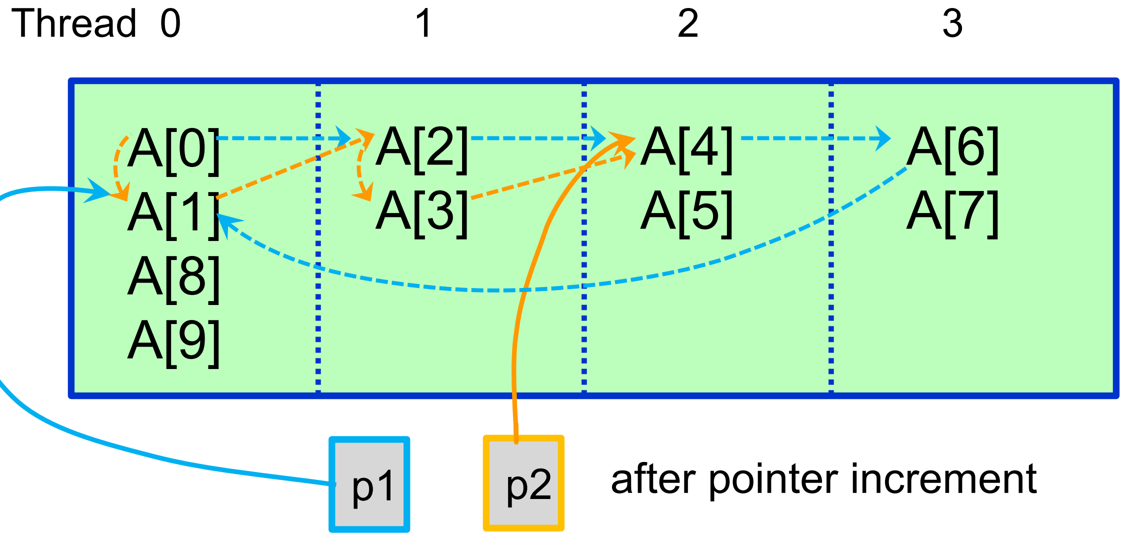
- **Breaking the local-affinity rule results in undefined behavior**
  – Cannot count on the compiler to inform you of your mistakes

Pointer-to-shared: blocking and casting

- **Assume 4 threads:**

```
shared [2] int A[10];
shared int *p1;
shared [2] int *p2;
```

```
if (MYTHREAD == 1) {
 p1 = (shared int *)&A[0];
 p1 += 4;
 p2 = &A[0];
 p2 += 4;
}
```



- **Block size is a part of the variable‘s type**
- **One may cast between pointer w/ different block sizes**
  - Pointer arithmetic follows blocking („phase“) of pointer
  - Cast changes the view but does not move any data

UPC dynamic Memory Allocation

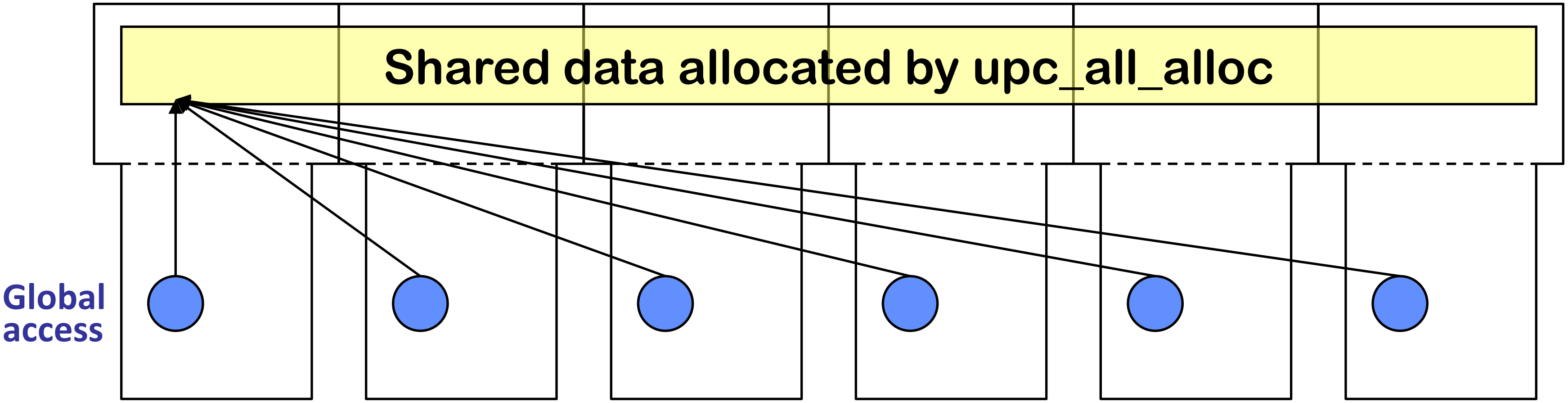
- **upc_all_alloc**
  - Collective over all threads (i.e., all threads must call)
  - All threads get a copy of the same pointer to shared memory

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

Run time arguments

  - Similar result as with static allocation at compile time:

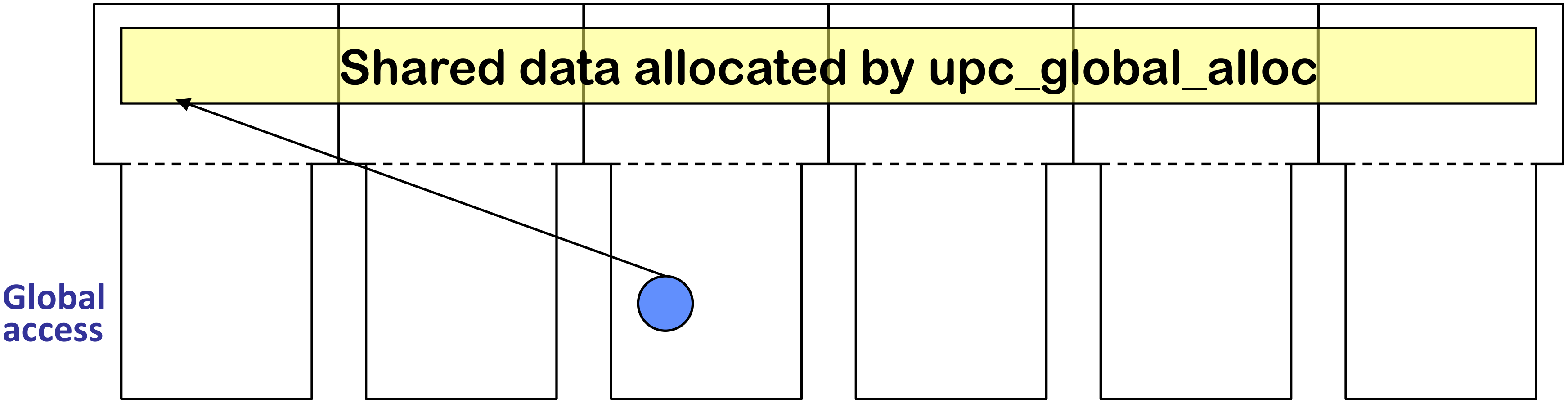
```
shared [nbytes] char[nblocks*nbytes];
```



UPC dynamic Memory Allocation (2)

- **upc_global_alloc**
  - Only the calling thread gets a pointer to shared memory

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



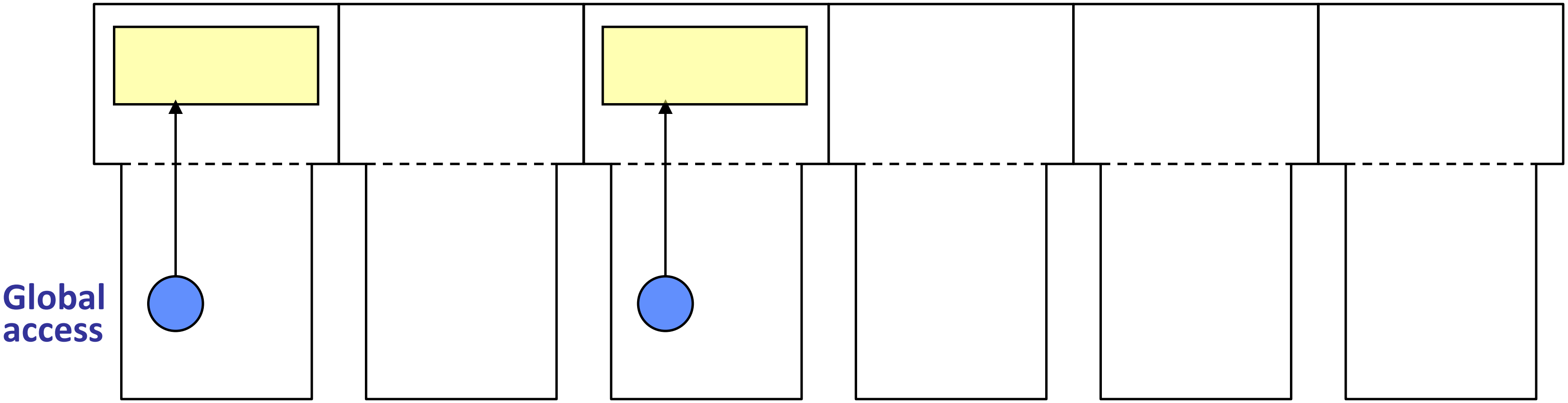
UPC dynamic Memory Allocation (3)

- **upc_alloc**
  - Allocates memory in the local thread that is accessible by all threads
  - Only on calling processes

```
shared void *upc_alloc( size_t nbytes )
```

  - Similar result as with static allocation at compile time:

```
shared [] char[nbytes]; // but with affinity to the calling thread
```



Common mistakes with dynamic allocation

- `shared int *p1 = upc_alloc(…);`
  - `p1` is cyclic, but the allocation is indefinite (all on calling thread)
  - Use of `p1[1]` might crash or might silently access wrong datum
  - Probably meant either of the following:

    ```
    shared int *p1 = upc_global_alloc(…); //cyclic
    shared [] int *p1 = upc_all_alloc(…); //indefinite
    ```
- `shared [2] int *p2 = upc_all_alloc(2, N*sizeof(int))`
  - Not *always* an error, but pretty often:
    first 2 is the ***size*** of a block, second 2 is the ***number*** of blocks
  - Probably meant either of the following:

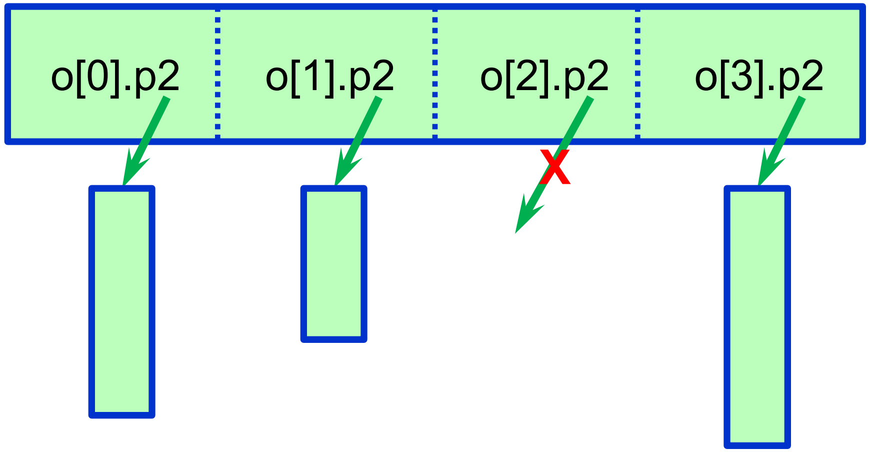
    ```
    upc_all_alloc(N, 2*sizeof(int));   // 2*N elements
    upc_all_alloc(N/2, 2*sizeof(int)); // N elements
    ```
- Multiple calls to `upc_free()` for memory allocated by `upc_all_alloc()`
  - Even though all threads call `upc_all_alloc()`, only ***one*** object is allocated and it must be freed (at most) ***once.***

Integration of the type system
UPC pointer components

- **Type definition**

```
typedef struct {
  shared int *p2;
} Ctr;
```

dynamically allocated entity **should** be in shared memory area



  – Must avoid undefined results when transferring data between threads

- **Example step-by-step:**

```
shared [1] Ctr o[THREADS];
#define SZ …

int main() {
  if (MYTHREAD == p) {
   o[MYTHREAD].p2 = (shared int *)
           upc_alloc(SZ*sizeof(int));
  }
  upc_barrier;
  if (MYTHREAD == q) {
    for (i=0; i<SZ; i++) {
      o[p].p2[i] = … ;
    }
  }
}
```

1. **Local (on thread p) memory allocation initializes pointer p2**
2. **Synchronization**
3. **Remote (on thread q) initialization of memory located on thread p.**

Major Features of UPC

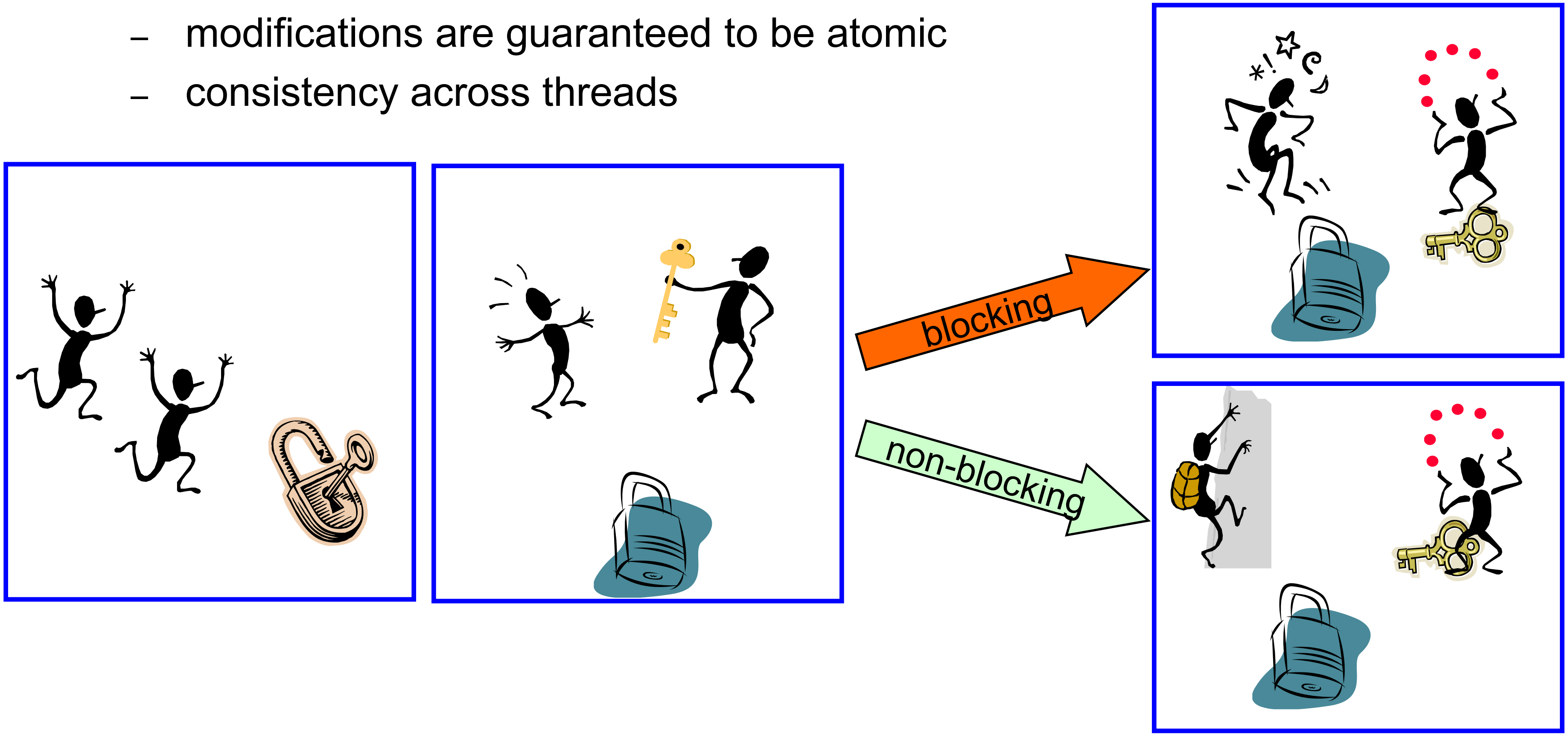
- Global view of data
- Shared arrays are distributed and array indices mapped to threads.
- Index mapping is part of the type system
- Block-wise distributions can be hard to handle
  - Last index x[……][THREADS] implies round robin distribution
  - Possibility of asymmetric distribution ([*] can be helpful)
- Allocation is part of the runtime library
- Multiple variants of dynamic allocation

Advanced Topics

- Partial synchronization
  - mutual exclusion
  - memory fences and atomic subroutines
  - split-phase barrier
- Collective operations
- Some parallel patterns and hints on library design:
  - parallelization concepts with and without halo cells
  - work sharing; distributed structures
  - procedure interfaces

Locks – a mechanism for mutual exclusion

- **Coordinate access to shared (= sensitive) data**
  - sensitive data represented as “red balls”
- **Use a shared lock variable**
  - modifications are guaranteed to be atomic
  - consistency across threads



Simplest examples for usage of locks

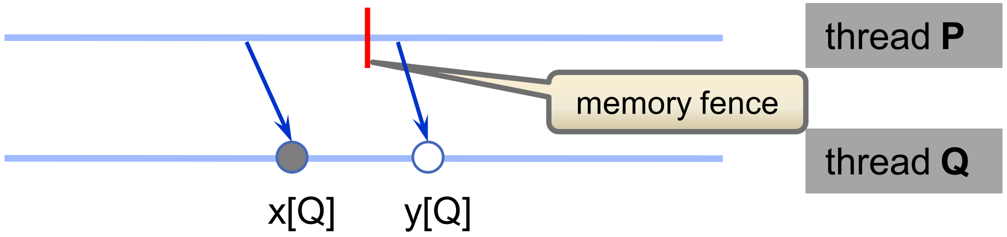
- single pointer lock variable
- thread-individual lock generation is also possible (non-collective)
- lock/unlock imply memory fence (next slide)

```
#include <upc.h>

upc_lock_t *lock;   // private pointer
                    // to a shared entity

lock = upc_all_lock_alloc();

// Blocking example
upc_lock(lock);
 // play with red balls
upc_unlock(lock);
upc_barrier; // separates examples

// Non-blocking example
for (;;) {
  if (upc_lock_attempt(lock)) break;
  // go climb that mountain
}
// play with red balls
upc_unlock(lock);
upc_barrier; // separates examples

// Single free call from arbitrary thread
if (MYTHREAD == THREADS-1)
  upc_lock_free(lock);
```

collective call
same result on
each thread

Memory fence

- **Goal: allow implementation of user-defined synchronization**
- **Requirement: ensure memory operations are observed in-order**
- **UPC solution: `upc_fence` provides a „null strict access“**



Note:
A memory fence is implied by **most other** synchronization statements

- **Assurance given by `upc_fence` :**
  - operations on x[Q] and y[Q] via statements on P
  - action on x[Q] precedes action on y[Q]
    → reordering by compiler and runtime are prohibited
  - P is subdivided into two segments / access epochs
  - **but:** segment on Q is unordered with respect to both segments on P

Atomic operations

Remember synchronization rule for relaxed memory model:
A shared entity may not be modified and read from two different threads
in unordered access epochs
Atomic subroutines allow a **limited exception** to this rule

- **Berkeley UPC extensions:**

```
shared int64_t *ptr;
int64_t value;
bupc_atomicI64_set_relaxed(ptr, value);
value = bupc_atomicI64_read_relaxed(ptr);
value = bupc_atomicI64_fetchadd_relaxed(ptr, op);
...and more...
```

  - (signed/unsigned) **`int`**, **`long`**, plus 64- and 32-bit integer types available
  - „**`_relaxed`**“ indicates relaxed (default) memory model
  - „**`_strict`**“ model also available (more info on later slides)

Semantics:

- location always has a well-defined value if **only** the above subroutines are used
- for multiple updates on the same location, **no assurance** is given about the order in which updates are observed → programmers‘ responsibility

Example: Producer/Consumer using BUPC Extensions

```
shared int ready = 0;
int val;

if (MYTHREAD == p) {
  // produce data
  upc_fence;
  bupc_atomicI_fetchadd_relaxed(&ready, 1);
} else if (MYTHREAD == q) {
  do {
    val = bupc_atomic2_read_relaxed(&ready);
  } while (!val);
  bupc_atomicI_fetchadd_relaxed(&ready, -1);
  upc_fence;
  // consume data
}
```

- **Fences**
  - in producer ensures data written before increment of **`ready`**
  - in consmer ensures data reads are not reordered before read of **`ready`**

- **Additional atomic functions:**
  - swap, compare-and-swap, fetch-and-<*logical-operation*>

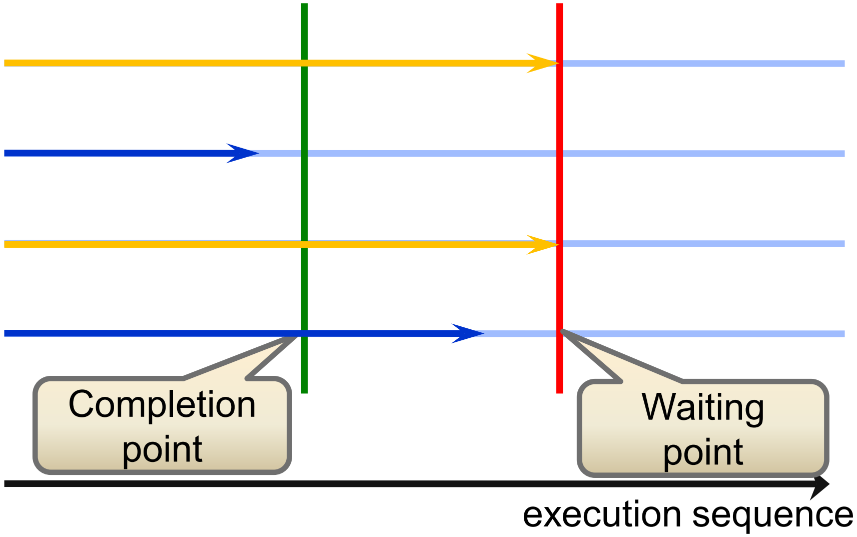
Recommendation

- **Functionality from the last three slides (fences and atomics)**
  - should be used only in exceptional situations
  - can easily be used in an unportable way
    (works on one system, fails on another) → beware

UPC split-phase barrier

- **Already seen in „Example 0 revisited“**
- **Separate barrier completion point from waiting point**
  - this allows threads to continue computations once all others have reached the completion point → reduce impacts of load imbalance

```
for (…) a[n][i]= …;
upc_notify;
// do work (on b?) not
// involving a
upc_wait;
for (…) b[i]= b[i]+a[q][i];
```



  - completion of `upc_wait` once all threads reach `upc_notify`
  - collective – **all** threads must execute both calls in same order

UPC memory consistency modes

- **How are shared entities accessed?**
  - relaxed mode → program **assumes** no concurrent accesses from different threads
  - strict mode → program **ensures** that accesses from different threads are separated, and **prevents** code movement across these synchronization points
  - relaxed is default; strict may have **large** performance **penalty**
- **Options for synchronization mode selection**
  - variable level:

    (at declaration
    or in a cast)

```
strict shared int flag = 0;
relaxed shared [*] int c[THREADS][3];
```

example for a spin lock

Thread q

```
c[q][i] = …;
flag = 1;
```

Thread p

```
while (!flag) {…};
… = c[q][j];
```

q has same value on thread p as on thread q

  - code block level:

```
{ // start of block
  #pragma upc strict
  … // block statements
}
// return to default mode
```

  - file level

```
#include <upc_strict.h>
// or upc_relaxed.h
```

consistency mode on variable declaration **overrides** code block or file level specification

What strict memory consistency does and doesn‘t do for you

- **„strict“ cannot prevent all race conditions**
  - example: „ABA“ race

```
strict shared int flag;
int val, val1, val2;
```

thread 0

```
flag = 0;
upc_barrier;
flag = 1;
flag = 0;
```

thread 1

```

upc_barrier;
val = flag;

```

may end up with 0 or 1

- **„strict“ does not make `a[i]+=j` atomic (read/modify/write)**
- **„strict“ does assure that changes on (complex) objects appear in the same order on other threads**

```
flag = 0;
upc_barrier;
flag = 1;
flag = 2;
```

```

upc_barrier;
val1 = flag;
val2 = flag;
```

may obtain (val1 <= val2), but **not** (val1 > val2). e.g. (2,1), (2,0) and (1,0) are not possible.

Collective functions (1)

- **Two types:**
  - data redistribution (scatter, gather, ...)
  - computation operations (reduce, prefix)
- **Separate include file:**

```
#include <upc_collective.h>
```

- **Synchronization mode:**
  - constants of type `upc_flag_t`

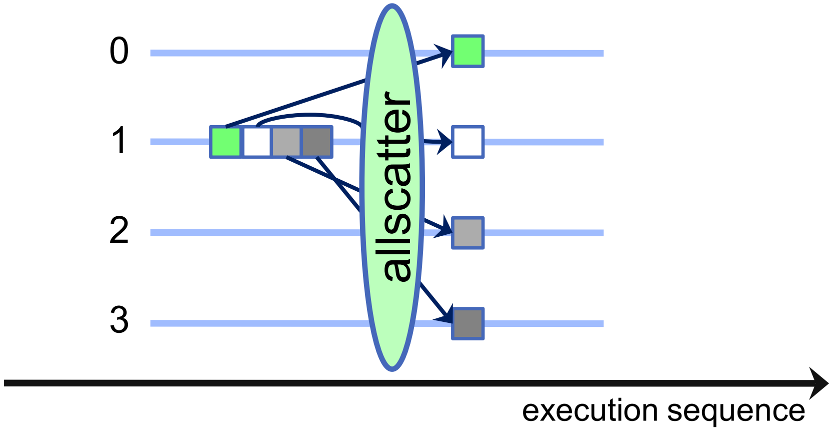
$$\texttt{UPC\_}\left\{\begin{array}{l}\texttt{IN}\\\texttt{OUT}\end{array}\right\}\texttt{\_}\left\{\begin{array}{l}\texttt{NOSYNC}\\\texttt{MYSYNC}\\\texttt{ALLSYNC}\end{array}\right.$$

- **IN/OUT:**
  - refers to whether the specified synchronization applies at the entry or exit to the call
- **Synchronization:**
  - NOSYNC – threads do not synchronize at entry or exit
  - MYSYNC – start processing of data only if owning threads have entered the call / exit function call only if all local read/writes complete
  - ALLSYNC – synchronize all threads at entry / exit to function
- **Combining modes:**
  - `UPC_IN_NOSYNC | UPC_OUT_MYSYNC`
  - `UPC_IN_NOSYNC` same as `UPC_IN_NOSYNC | UPC_OUT_ALLSYNC`
  - `0` same as `UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC`

Collectives (2): Example of redistribution

- **Scatter**

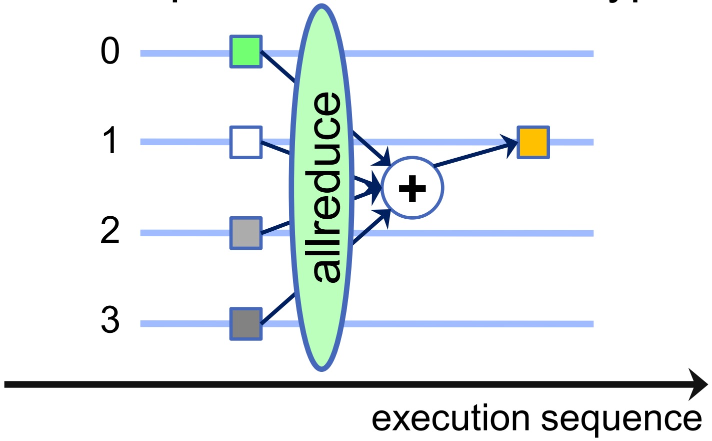
```
void upc_all_scatter (
     shared void *dst,
     shared const void *src,
     size_t nbytes,
     upc_flag_t sync_mode);
```



  - **src** has affinity to a single thread
  - i-th block of size **nbytes** is copied to portion of **dst** with affinity to thread i

Collectives (3): Reductions

- **Reduction concept:**
  - distributed set of objects
  - operation defined on type



  - destination resides in shared space

- **Reduction type codes**

C/UC – signed/unsigned char	**L/UL** – signed/unsigned long
S/US – signed/unsigned short	**F/D/LD** – float/double/long double
I/UI – signed/unsigned int	

- **Operations:**

Numeric	Logical	User-defined function
UPC_ADD	UPC_AND	UPC_FUNC
UPC_MULT	UPC_OR	UPC_NONCOMM_FUNC
UPC_MAX	UPC_XOR	
UPC_MIN	UPC_LOGAND	
	UPC_LOGOR	

  - are constants of type `upc_op_t`

Collectives (4): Reduction prototype

```
void upc_all_reduceT(
  shared void *restrict dst,
  shared const void *restrict src,
  upc_op_t op,
  size_t nelems,
  size_t blk_size,
  T(*func)(T, T),
  upc_flag_t flags);
```

destination and source, respectively

number of elements of type T

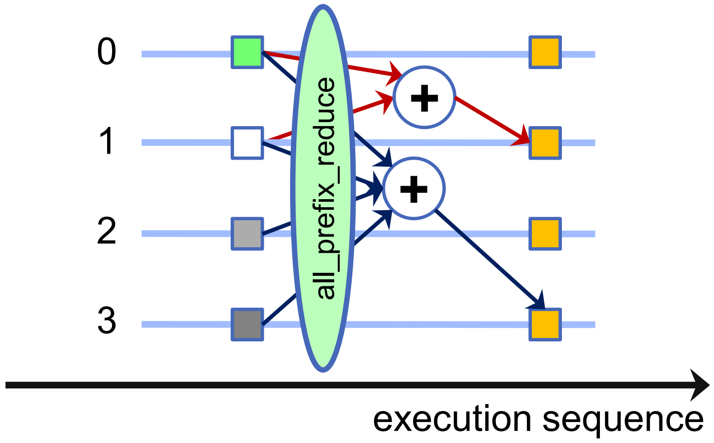
source pointer block size
or 0 for indefinate

- `src` and `dst` may not be aliased
- replace `T` by type code (C, UC, etc.)
- function argument will be `NULL` unless `op` specifices a user-defined function

Collectives (5): further functions

- **Redistribution functions**
  - upc_all_broadcast()
  - upc_all_gather_all()
  - upc_all_gather()
  - upc_all_exchange()
  - upc_all_permute()

- **Prefix reductions**
  - upc_all_prefix_reduceT()
  - semantics:

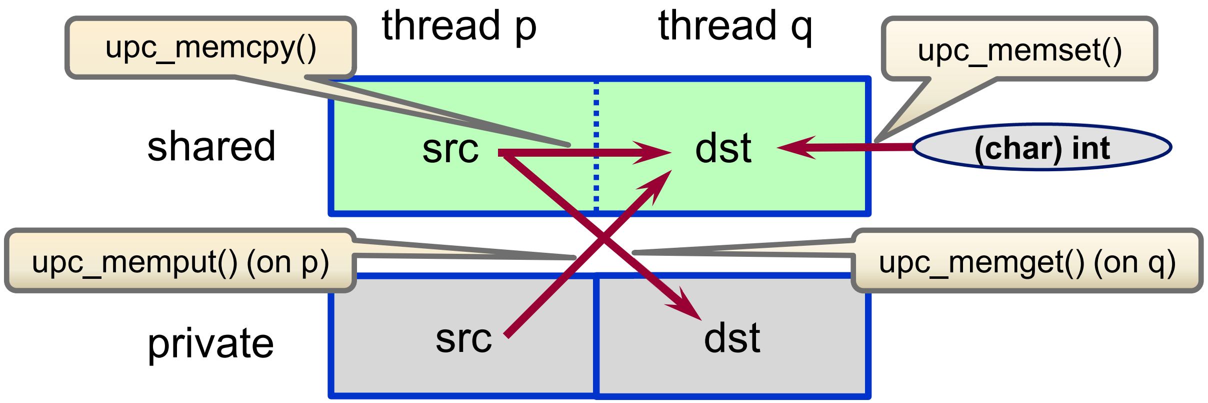


for UPC_ADD, thread i gets $\sum_{k=0}^{i} src[k]$

(thread-dependent result)

→ consult the UPC language specification for details

One-sided bulk memory transfers



- **Available for efficiency**
  - operate in units of bytes
  - use restricted pointer arguments

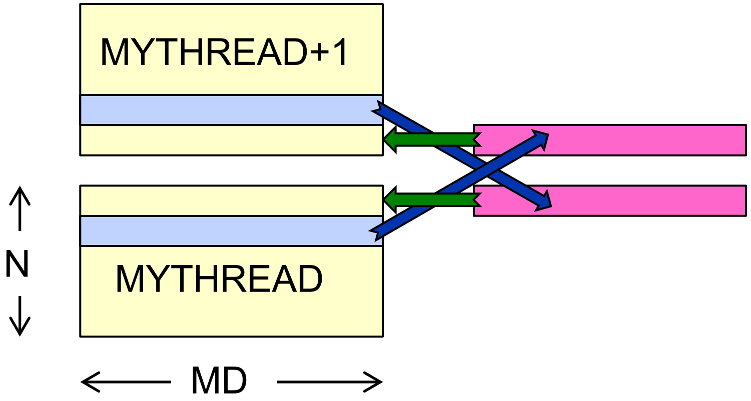
prototypes from `upc.h`

```
void upc_memcpy(shared void *dst,
    shared const void *src, size_t n);
void upc_memget(void *dst,
    shared const void *src, size_t n);
void upc_memput(shared void *dst,
    void *src, size_t n);
void upc_memset(shared void *dst,
    int c, size_t n);
```

Work sharing:
data exchange avoiding haloed shared data

- **Use following data layout**

```
double a[N][MD]; // private
shared [MD] double
       a_top[THREADS][MD],
       a_bot[THREADS][MD];
size_t sz = MD*sizeof(double);
```



- **does not require entire data field to be shared**

Note:
for 2-D blocking this is not fun. A strided block transfer facility is a BUPC extension.

- **Communication code**

```
if (MYTHREAD+1 < THREADS) {
  upc_memput(&a_bot[MYTHREAD+1][0],
             &a[N-2][0], sz);
}
if (MYTHREAD-1 > 0) {
  upc_memput(&a_top[MYTHREAD-1][0],
             &a[1][0], sz);
}
upc_barrier;
if (MYTHREAD > 0) {
  upc_memget(&a[0][0],
             &a_bot[MYTHREAD][0], sz);
}
if (MYTHREAD < THREADS) {
  upc_memget(&a[N-1][0],
             &a_top[MYTHREAD][0], sz);
}
```

- **maintains one-sided semantics, but one more copy needed**
- `memcpy()` **could replace** `upc_memget()` **in this example with the addition of casts**

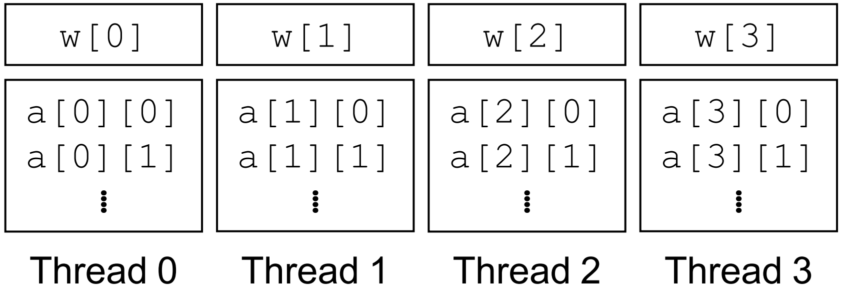
Subroutine/function interface design

```
void subr(int n, shared float *w)
{
  int i;
  float *wloc;
  wloc = (float *) &w[MYTHREAD];
  for (i=0; i<n; i++) {
     … = w[i] + …
  }
  // exchange data
  upc_barrier;
  // etc.
}
```

- **subr** assumes local size is n
- cast to local pointer for safety of use **and performance** if only local accesses are required
- declarations with *fixed* block size > 1 also possible (default is 1, as usual)

```
shared [*] float x[THREADS][NDIM]

int main(void) {
  // initialize x[][]
  upc_barrier;
  subr(NDIM, (shared float *) x);
}
```

- cast to cyclic to match the prototype
- This approach of passing cyclic pointer and blocksize as arguments is a common solution to UPC library design.
- cyclic is “good enough” in most cases because function can recover actual layout via pointer arithmetic
- in this example w[i] aliases a[i][0]



w[0]	w[1]	w[2]	w[3]
a[0][0] a[0][1] ⋮	a[1][0] a[1][1] ⋮	a[2][0] a[2][1] ⋮	a[3][0] a[3][1] ⋮
Thread 0	Thread 1	Thread 2	Thread 3

Factory procedures

- **Function returning pointer-to-shared**

```
shared *float factory(…) {
  // determine size, n, to allocate
  wk = (shared float *)upc_all_alloc(THREADS, sizeof(float)*n);
  // fill wk with data
  return wk;
}
```

  - use of upc_all_alloc() means this must be called collectively
  - remember: allocation functions do not synchronize

Distributed Structures (1)

- **Irregular data distribution**
  - use a data structure
  - recursive processing
- **Binary tree example:**

```
typedef struct tree {
  upc_lock_t *lk;
  shared struct tree *left;
  shared struct tree *right;
  shared Content *data;
};
typedef struct tree Tree;
```

regular „serial“ type definition

  - prerequisite: ordering relation

```
int lessthan(shared Content *a,
             Content *b);
```

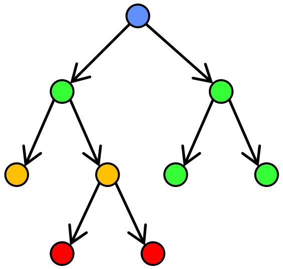
- **Constructor for Tree object**
  - To be called by **one** thread

```
shared Tree *Tree_init() {
  shared Tree *this;
  this = (shared Tree *)
          upc_alloc(sizeof(Tree));
  this->lk = upc_global_lock_alloc();
  this->data = (shared Content *)
          upc_alloc(sizeof(Content));
  this->left = this->right = NULL;
  return this;
}
```

  - initializes storage for lock and data components, NULL for children

Distributed Structures (2)

- **Concurrent population**
  - locking ensures race-free processing



color ↔ thread number

```
void insert(shared Tree *this, Content *stuff) {
  upc_lock(this->lk);
  if (this->left) { // Interior node (contains data)
    upc_unlock(this->lk);
    if (lessthan(this->data, stuff)) {
      insert(this->left, stuff);
    } else {
      insert(this->right, stuff);
    }
  } else { // Leaf node (no data value yet)
    this->left = Tree_init();
    this->right = Tree_init();
    upc_memput(this->data, stuff, sizeof(Content));
    upc_unlock(this->lk);
  }
}
```

invoke constructor

copy object to (**remote**) shared entity

Distributed Structures (3)

- **Assumptions**
  - structure is written once or rarely
  - many operations performed on entries, in access epochs separated from insertions

```
void traverse(shared Tree *this, Params *op) {
  if (this->left) { // Non-empty node
    if (upc_threadof(this->data) == MYTHREAD) {
      process((Content *)this->data, op);
    }
    traverse(this->left, op);
    traverse(this->right, op);
  }
}
```

guarantees locality

  - To be complete, **`traverse()`** must be executed by all threads which called **`insert()`**, but not necessarily collectively.

Real Applications and Hybrid Programming

- NAS parallel benchmarks
  - Optimization strategies
  - Hybrid concepts for optimization
- Hybrid programming
  - MPI allowances for hybrid models
  - Hybrid PGAS examples

The eight NAS parallel benchmarks (NPBs) have been written in various languages including hybrid for three

MG	Multigrid	Approximate the solution to a three-dimensional discrete Poisson equation using the V-cycle multigrid method	
CG	Conjugate Gradient	Estimate smallest eigenvalue of sparse SPD matrix using the inverse iteration with the conjugate gradient method	
FT	Fast Fourier Transform	Solve a three-dimensional PDE using the fast Fourier transform (FFT)	
IS	Integer Sort	Sort small integers using the bucket sort algorithm	
EP	Embarrassingly Parallel	Generate independent Gaussian random variates using the Marsaglia polar method	
BT SP LU	Block Tridiagonal Scalar Pentadiag Lower/Upper	Solve a system of PDEs using 3 different algorithms	MZ

The NPBs in UPC are useful for studying various PGAS issues

- **Using customized communication to avoid hot-spots**
  - UPC Collectives do not support certain useful communication patterns
- **Blocking vs. Non-Blocking (Asynchronous) communication**
  - In FT and IS using non-blocking gave significantly worse performance
  - In MG using non-blocking gave small improvement
- **Benefits of message aggregation depends on the arch./interconnect**
  - In MG message aggregation is significantly better on Cray XT 5 w/ SeaStar2 interconnect, but almost no difference is observable on Sun Constellation Cluster w/ InfiniBand
- **UPC – Shared Memory Programming studied in FT and IS**
  - Less communication but reduced memory utilization
- **Mapping BUPC language-level threads to Pthreads and/or Processes**
  - Mix of processes and pthreads often gives the best performance

Using customized communication to avoid hot-spots

- UPC Collectives might not support certain type of communication patterns (for example, vector reduction)
- Customized global communication is sometimes necessary!
- Collective communication naïve approach (FT example):

```
for (i=0; i<THREADS; i++)
    upc_memget( … thread i … );
```

- Same communication ordered to avoid hot-spots:

```
for (i=0; i<THREADS; i++){
     peer = (MYTHREAD + i) % THREADS;
     upc_memget( … thread peer … );
}
```

- Communication performance difference can exceed 50% (observed on Carver/NERSC – 2 quad-core Intel Nehalem cluster with InfiniBand Interconnect)

Blocking vs. Non-Blocking (Asynchronous) communication

- **Berkeley UPC includes extensions for non-blocking bulk transfers (for efficient computation/communication overlap):**
  - `upc_handle_t bupc_memget_async(void *dst, shared const void *src, size_t nbytes);`
    – starts communication
  - `void bupc_waitsync(upc_handle_t handle);`
    – waits for completion
  - Asynchronous versions of memcpy and memput also exist
- **Not always beneficial:**
  - Non-blocking communication can inject large number of messages into the network causing congestion
  - Lower levels of the network stack (firmware, switches) may then employ internal flow-control which reduces the effective bandwidth

Blocking vs. Non-Blocking (Asynchronous) communication (cont.)

- FT – no communication/computation overlap possible, but non-blocking communication can be used:

```
bupc_handle_t handles[THREADS];
for(i = 0; i < THREADS; i++) {
    peer = (MYTHREAD+i) % THREADS; // avoids hot-spots
    handles[i] = bupc_memget_async( … thread peer … );
}
for(i=0; i < THREADS; i++)
    bupc_waitsync(handles[i]);
```

- Using non-blocking communication, FT (also IS) experiences up to 60% communication performance degradation. For MG we observed ~2% performance improvement.
- Slowdown is caused by a large number of messages injected into the network (there is no computation that could overlap communication and thus reduce the injection rate)

In addition to asynchronous, one can study strided communication and message aggregation

- **Using strided communication is generally an improvement**
  - Again BUPC has extensions for this purpose
- **Message aggregation reduces the number of messages, but introduces some packing/unpacking overhead**
- **Message aggregation increases programming effort.**
- **Example:**

Fine-grained Communication

Thread A → Thread B

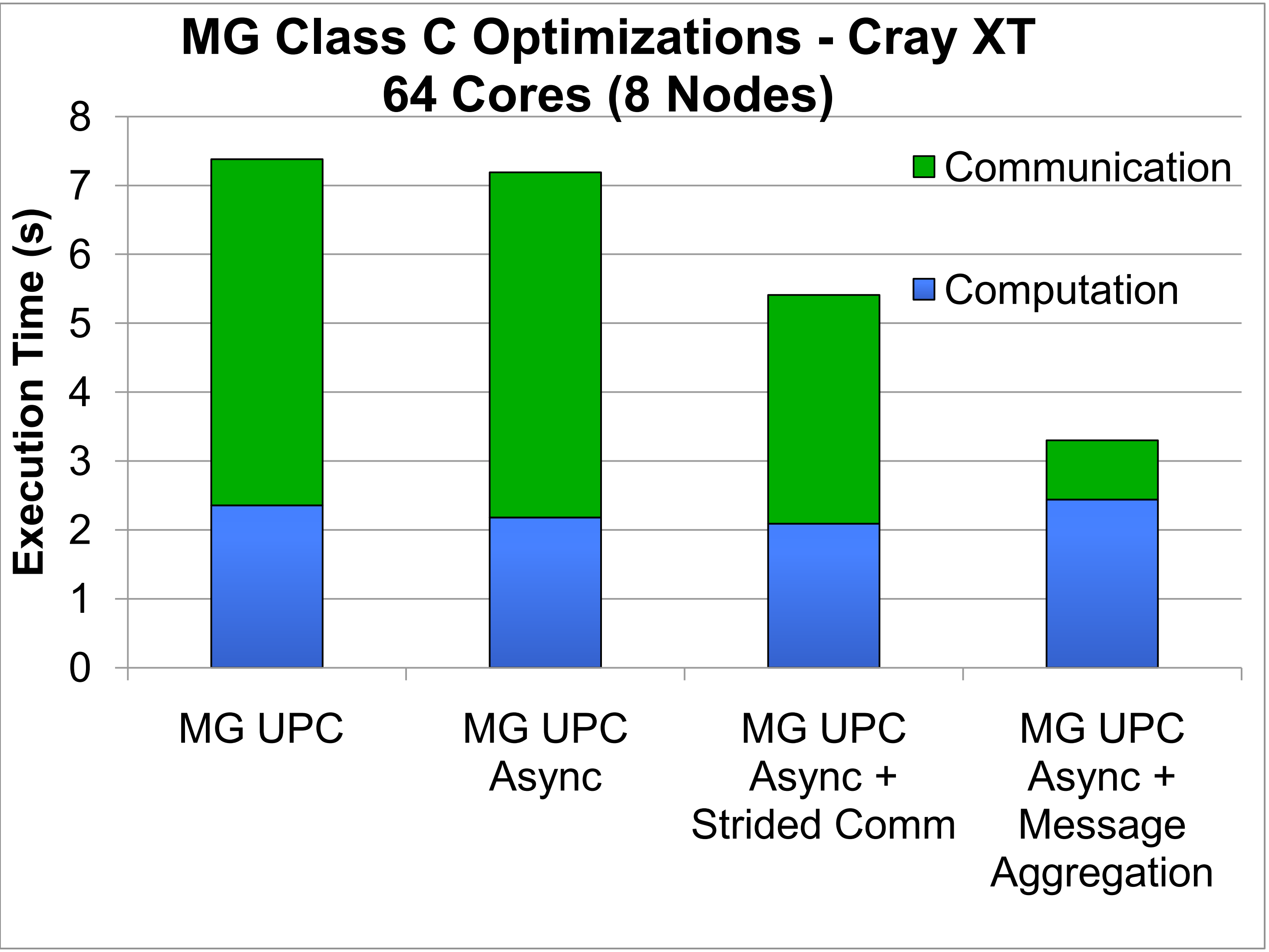
```
for(i=0; i<n1; i++)
  upc_memput(
    &k[i], &u[i],
    n2 * sizeof( double ));
```

Message Aggregation

Thread A: **Thread B:**

```
buff = pack(u);
upc_memput(
  &k[0], &buff,
  n1*n2*sizeof(double));
upc_barrier;                upc_barrier;
                            unpack(k);
```

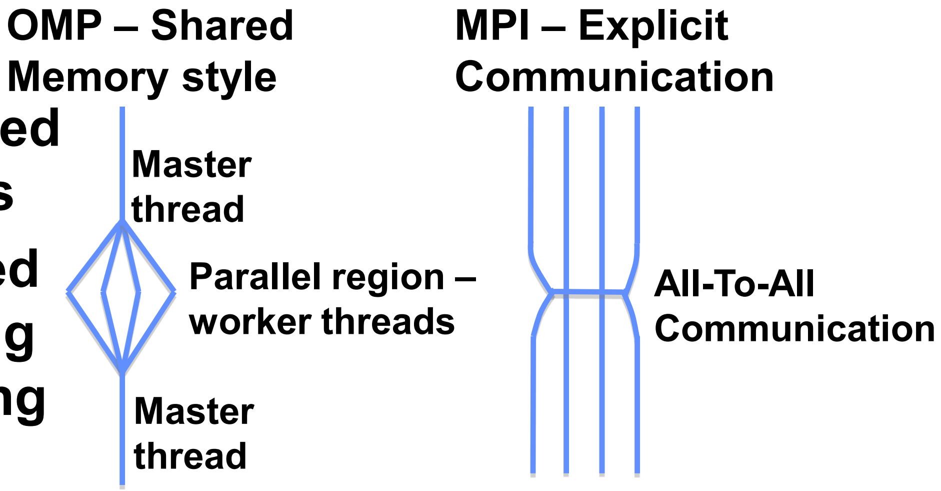
MG message aggregation is significantly better on Cray SeaStar2 interconnect



- MG message aggregation had almost no difference on Ranger InfiniBand interconnect

UPC – Hierarchical Shared Memory Programming reduces communication time

- **UPC designed for pure distributed or pure shared memory systems**
- **UPC capable of exploiting shared memory (OMP-like) programming style within a node (thus avoiding some explicit communication)**



OMP – Shared Memory style

Master thread

Parallel region – worker threads

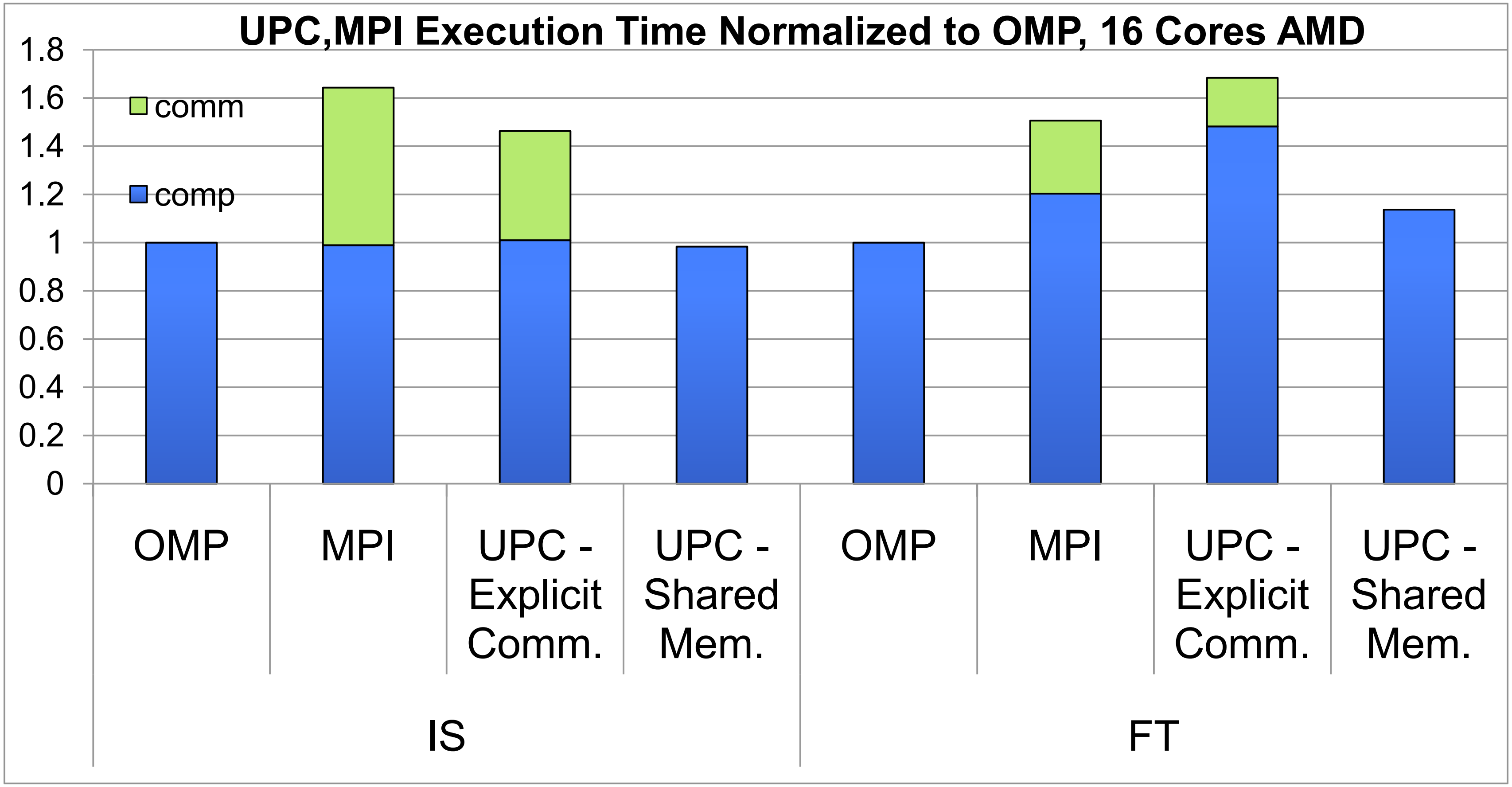
Master thread

MPI – Explicit Communication

All-To-All Communication

- **Drawback: reduced memory utilization (large fraction unusable)**
  - In the UPC hierarchical model, only the shared heap allocated by the master thread is used
  - In BUPC all threads have equally sized shared-heaps
  - In *any* UPC `upc_{all,global}_alloc()` allocate across all threads
  - Can result in large fraction of node memory potentially unusable
  - Careful data placement capable of increasing memory utilization
  - Berkeley is working on enabling uneven heap distribution in BUPC

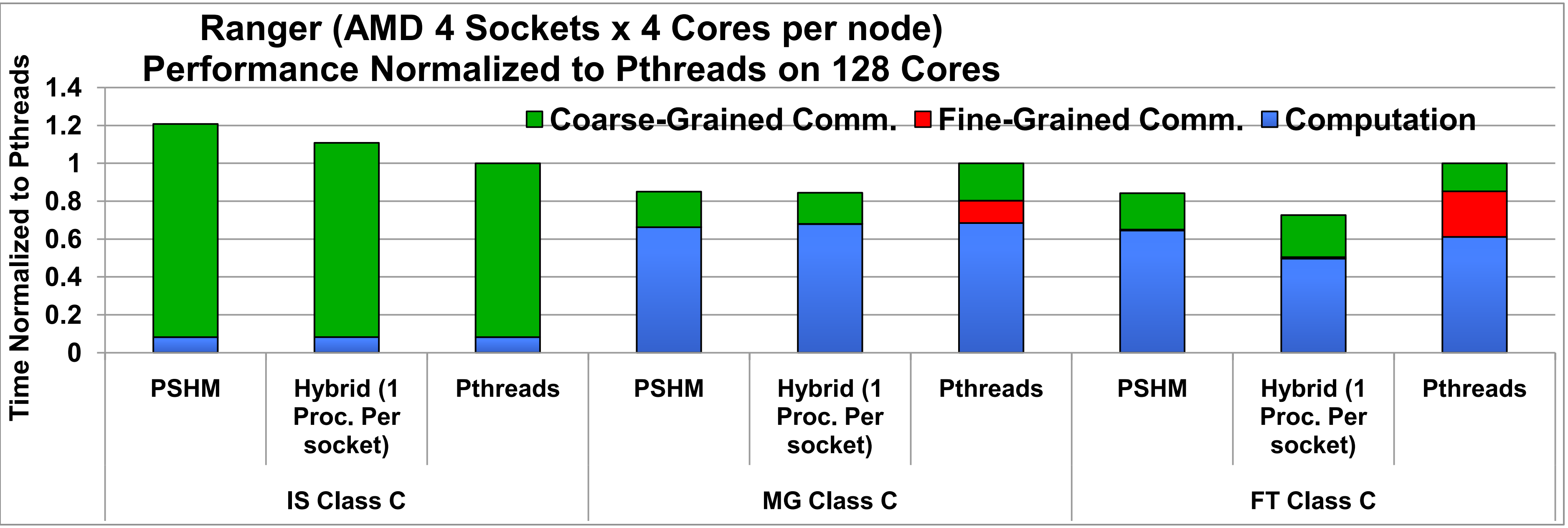
Use of UPC shared memory to remove a transpose operation in FT



BUPC language-level threads can be mapped to Pthreads and/or Processes

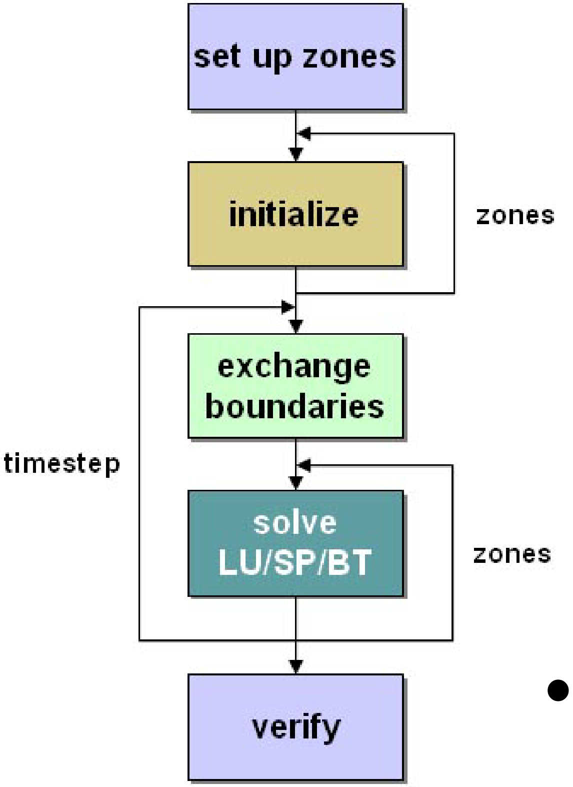
- **Pthreads – shared memory communication through shared address space**
- **Processes – shared memory communication through shared memory segments (POSIX, SysV or mmap(file)) called PSHM**
- **NPBs performance depends on Pthreads/Processes**
  - Pthreads share one network endpoint; PSHM has network endpoint per process
  - Due to sharing of one network endpoint, pthreads experience messaging contention, resulting in throttled injection rate
  - Processes (PSHM) can inject messages into the network faster (but large messages count may decrease effective bandwidth)
  - PSHM avoids contention overhead when interacting with external libraries/drivers
  - Contention and injection rate compete for dominance

Mix of both processes and pthreads often achieves the best performance



For FT the hybrid approach (1 process per socket and pthreads within a socket) is best and is a “reasonable” approach for the other NPBs

Some NAS Parallel Benchmarks have been written in multi-zone hybrid versions (currently with OpenMP)



	MPI/OpenMP Version
Time step	Sequential
Inter-zones	MPI Processes
Exchange boundaries	Call MPI
Intra zones	OpenMP

- Multi-zone versions of the NPSs LU,SP, and BT are available from:

www.nas.nasa.gov/Resources/Software/software.html

Figure adapted from Gabriele Jost, et al., ParCFD2009 Tutorial

Hybrid coding can yield improved performance for some benchmarks

- **BT-MZ: (Block-tridiagonal Solver)**
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance

Pure MPI: Load-balancing problems!

Good candidate for MPI+OpenMP

- **LU-MZ: (Lower-Upper Symmetric Gauss Seidel Solver)**
  - Size of the zones identical:
    - no load-balancing required
    - limited parallelism on outer level

Limited MPI Parallelism:
→ MPI+OpenMP increases Parallelism

- **SP-MZ: (Scalar-Pentadiagonal Solver)**
  - Size of zones identical
    - no load-balancing required

Load-balanced on MPI level: Pure MPI should perform best

Adapted from Gabriele Jost, et al., ParCFD2009 Tutorial

PGAS languages can also be combined with MPI for hybrid

- **MPI is designed to allow *coexistence* with other parallel programming paradigms and uses the same *SPMD* model:**

  ➔ MPI and UPC (or CAF) can exist together in a program

- **When mixing communications models, each will have its own progress mechanism and associated rules/assumptions**

- **Deadlocks can happen if some processes are executing blocking MPI operations while others are in “PGAS communication mode” and waiting for images (e.g. sync all)**

  ➔ *"MPI phase" should end with MPI barrier, and a "PGAS phase" should end with a PGAS barrier to avoid communication deadlocks*

We give one example of hybrid MPI and Cray CAF interoperability

main.F90

```
program MPI_and_CAF

   integer ::  ntasks,ierr,rank,size
   integer,pointer,dimension(:) :: array

   call MPI_Init(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD,ntasks,ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)

   size = 1000
   allocate(array(1:size))
   array = 1

   call mpi_routine1(array)

   call MPI_BARRIER(MPI_COMM_WORLD,ierr)

   call caf_routine(rank,size,array)

   call MPI_BARRIER(MPI_COMM_WORLD,ierr)

   call mpi_routine2(array)

   deallocate(array)
   call MPI_FINALIZE(ierr)

end program MPI_and_CAF
```

mpi.F90

```
subroutine mpi_routine1…
subroutine mpi_routine2 …
```

caf.F90

```
subroutine caf_routine(mpi_rank,size,mpi_array)

   integer :: mpi_rank,size,world_rank,world_size
   integer,dimension(size ) :: mpi_array
   integer,allocatable :: co_array(:)[:]

   SYNC ALL ! Full barrier; wait for all images

   world_rank = THIS_IMAGE() ! equal to mpi_rank
   world_size = NUM_IMAGES()


   … ! some computation on mpi_array and co_array

   SYNC ALL

end subroutine caf_routine
```

```
# building for Hopper/Franklin @ NERSC:
module swap PrgEnv-pgi PrgEnv-cray
ftn –static –O3 –h caf caf.F90
ftn –static –O3 mpi.F90
ftn –static –O3 main.F90
ftn –static –o exec caf.o mpi.o main.o
```

Hybrid MPI and UPC

- **Hybrid MPI and UPC still an area of active research/development**
  - Works on many clusters but not yet on Cray or IBM BG/P
  - Most significant hurdle is those systems' custom job launchers
- **There are three hybrid models† in the literature that vary the level of nesting and number of instances of each models**
  - Flat model: provides a non-nested common MPI and UPC execution where each process is a part of both the MPI and the UPC execution
  - Nested-funneled model: provides an operational mode where only the master thread per group (of one of more compute nodes) gets an MPI rank and can make MPI calls
  - Nested-multiple model: provides a mode where every UPC thread in each group gets its own MPI rank and can make MPI calls independently

† from "Hybrid Parallel Programming with MPI and Unified Parallel C"
by James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur

Thank you.