Unified Parallel C (contd)

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@cs.rice.edu

Schedule for rest of semester

- 4/3/08 - No class (midterm recess)
- 4/8/08 - Guest lecture by John Mellor-Crummey on Co-Array Fortran
- 4/11/08 - Deadline for finalizing your choice for Assignment #4
- 4/22/08 - In-class final exam
- 4/30/08 - Deadline for Assignment #4

Acknowledgments

- Supercomputing 2007 tutorial on "Programming using the Partitioned Global Address Space (PGAS) Model" by Tarek El-Ghazawi and Vivek Sarkar
  - —http://sc07.supercomputing.org/schedule/event_detail.php?evid=11029
- UPC Language Specification (V 1.2)
  - —http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf
- UPC Manual
  - —http://upc.gwu.edu/downloads/Manual-1.2.pdf
- UC Berkeley CS 267 Lecture 14, Kathy Yelick, Spring 2007
  - —http://www.cs.berkeley.edu/%7Eyelick/cs267_sp07/lectures/lecture14/lecture14_upc_ky07.ppt

Questions from Previous Lecture

Q1: Do the UPC non-collective "string" library calls support third-party transfers?

A: Yes for upc_memcpy, but not for upc_memget and upc_memput.

upc_memcpy(dst, src, n) copies shared to shared memory.
upc_memput(dst, src, n) copies from private to shared memory.
upc_memget(dst, src, n) copies from shared to private memory.

"The function **upc_memcpy(dst, src, n)** copies **n** bytes from a shared object (**src**) with affinity to one thread, to a shared object (**dst**) with affinity to the same or another thread. (Neither thread needs to be the calling thread.)"

Questions from Previous Lecture

Q2: What memory model is assumed for UPC non-collective library functions?

A: Relaxed. "For non-collective functions in the UPC standard library (e.g. upc mem {put, get, cpy}), any implied data accesses to shared objects behave as a set of relaxed shared reads and relaxed shared writes of unspecified size and ordering, issued by the calling thread."

We will discuss strict vs. relaxed UPC memory models in today's lecture

Questions from Previous Lecture

Q3: Can the block size be non-constant in a block-cyclic distribution?

A: The two choices for the block size are constant-expr or *.

"If the layout qualifier is of the form [*], the shared object is distributed as if it had a block size of

(sizeof(a) / upc_elemsizeof(a) + THREADS - 1) / THREADS,

where a is the array being distributed."

Questions from Previous Lecture

Q4: Does upc_localsizeof depend on the calling thread?

A: No.

> **"The upc_localsizeof operator returns the size, in bytes, of the local portion of its operand, which may be a shared ob ject or a shared-qualified type. It returns the same value on all threads; the value is an upper bound of the size 15allocated with affinity to any single thread and may include an unspecified amount of padding. The result of upc localsizeof is an integer constant."**

However, upc_affinitysize can depend on threadid:

> **size_t upc_affinitysize(size_t totalsize, size_t nbytes, size_t threadid);**
>
> **"upc affinitysize is a convenience function which calculates the exact size of the local portion of the data in a shared ob ject with affinity to threadid."**

Dynamic Memory Allocation

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
- As a convention, the name of a collective function typically includes “all”

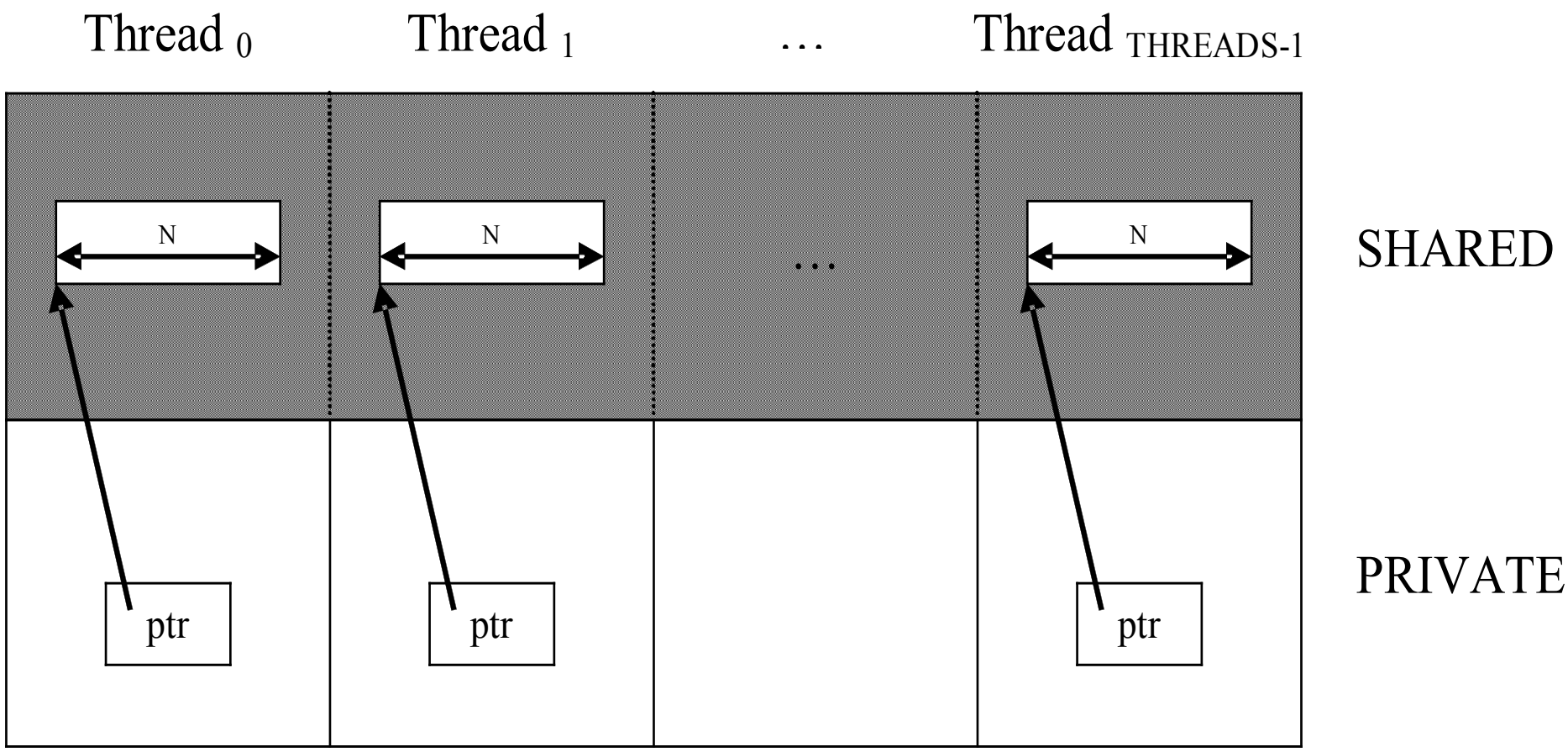
Local-Shared Memory Allocation

shared void *upc_alloc (size_t nbytes);

nbytes: block size

- **Non collective, expected to be called by one thread**
- **The calling thread allocates a contiguous memory region in the local-shared space of the calling thread**
- **Space allocated per calling thread is equivalent to :**
  shared [] char[nbytes]
- **If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer**

Local-Shared Memory Allocation




```
shared [] int *ptr;
ptr = (shared [] int *)upc_alloc(N*sizeof( int ));
```

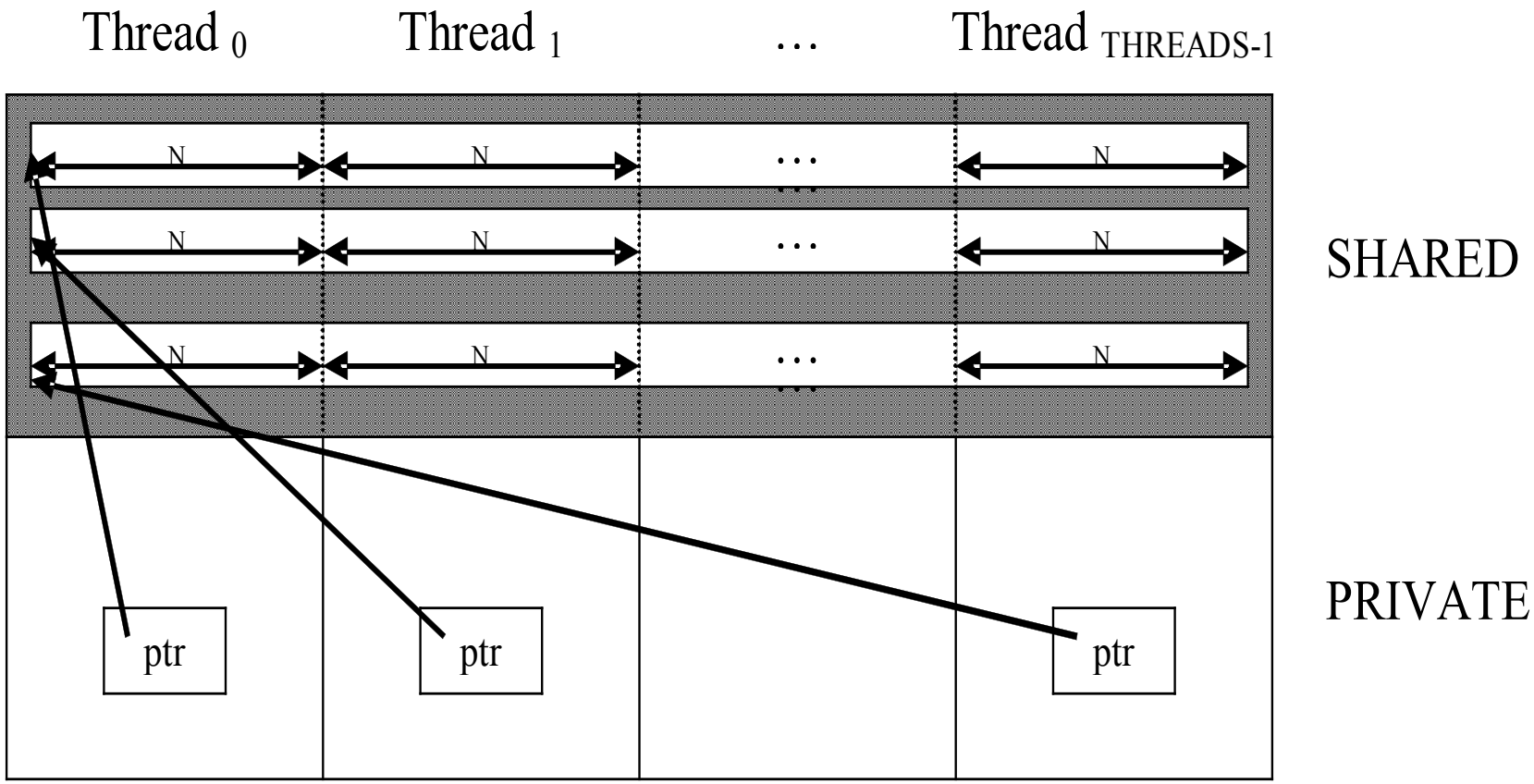
Global Memory Allocation

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

nblocks : number of blocks
nbytes : block size

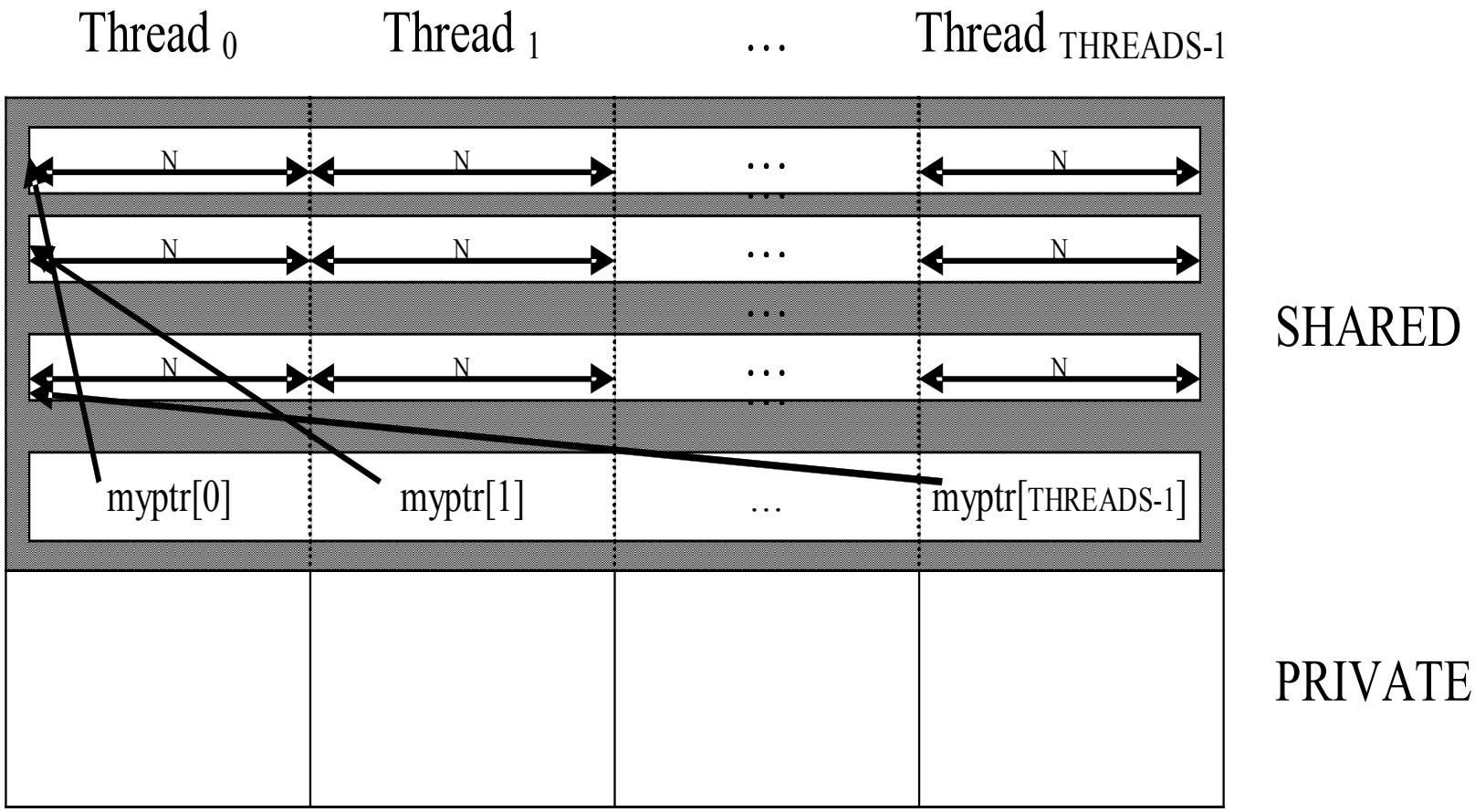
- **Non collective, expected to be called by one thread**
- **The calling thread allocates a contiguous memory region in the shared space**
- **Space allocated per calling thread is equivalent to :**
  shared [nbytes] char[nblocks * nbytes]
- **If called by more than one thread, multiple regions are allocated and each calling thread gets a different pointer**

Global Memory Allocation: Example 1




```
shared [N] int *ptr;

ptr = (shared [N] int *)
      upc_global_alloc( THREADS, N*sizeof( int ));
```

Global Memory Allocation: Example 2




```
shared [N] int *shared myptr[THREADS];

myptr[MYTHREAD] = (shared [N] int *)
  upc_global_alloc( THREADS, N*sizeof( int ));
```

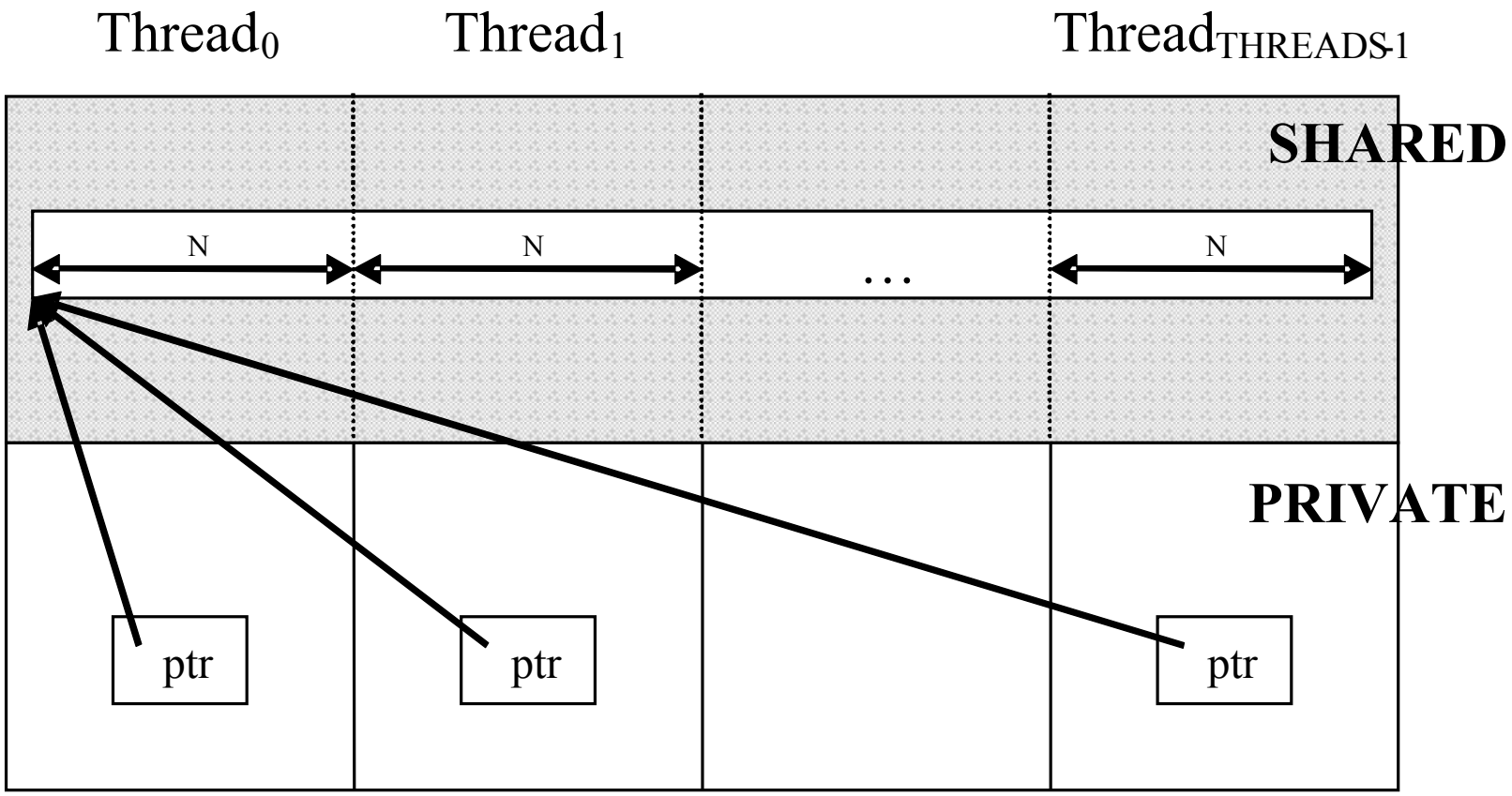
Collective Global Memory Allocation

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

nblocks: number of blocks
nbytes: block size

- **This function has the same result as** upc_global_alloc**. But this is a collective function, which is expected to be called by all threads**
- **nblocks and nbytes must be *single-valued* i.e., must have the same value on every thread. (The behavior of the operation is otherwise undefined.)**
- **All the threads will get the same pointer**
- **Equivalent to :**
  shared [nbytes] char[nblocks * nbytes]

Collective Global Memory Allocation




```
shared [N] int *ptr;
ptr = (shared [N] int *)
        upc_all_alloc( THREADS, N*sizeof( int ) );
```

Memory Space Clean-up

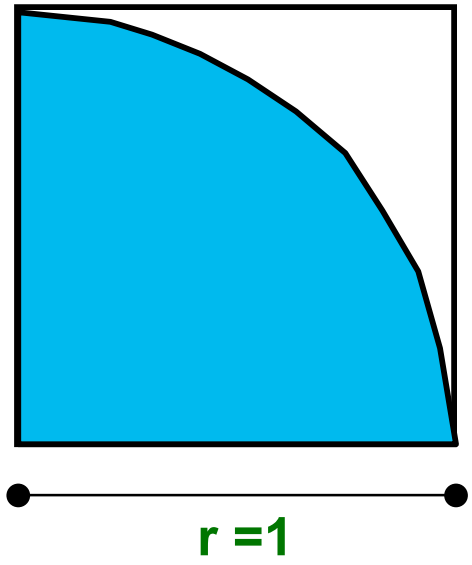
```
void upc_free(shared void *ptr);
```

- **The upc_free function frees the dynamically allocated shared memory pointed to by ptr**
- **upc_free is not collective**

MonteCarlo Example

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 = $¼ * \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - —# points inside / # points total
  - — π = 4*ratio



Pi in UPC

- **Independent estimates of pi:**

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

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

  srand(MYTHREAD*17);

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

Each thread gets its own copy of these variables

Each thread can use input arguments

Initialize random in math library

Each thread calls “hit” separately

Helper Code for Pi in UPC

- **Required includes:**

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

- **Function to throw dart and calculate where it hits:**

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

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?

Pi in UPC: Lock Version

- 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

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

all_hits is shared by all processors, just as hits was

```
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);
  }
}
```

update element with local affinity

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 two versions of Pi work, but they are not scalable:
  - —On a large # of threads, the locked region will be a bottleneck or Thread 0 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.

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.

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

Matrix Multiply Example

Example: Matrix Multiplication in UPC

- Given two integer matrices A(NxP) and B(PxM), we want to compute C =A x B.
- Entries c_{ij} in C are computed by the formula:

$$c_{ij} = \sum_{l=1}^{p} a_{il} \times b_{lj}$$

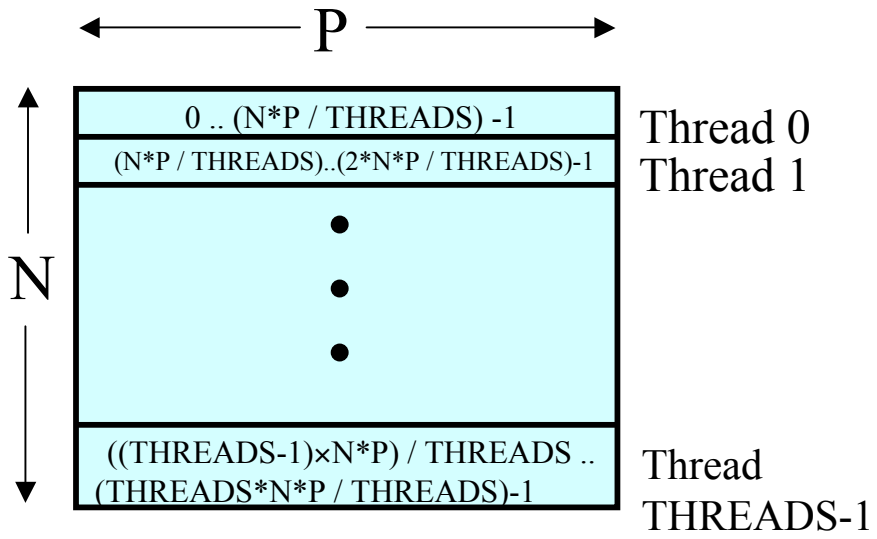
Sequential C version

```
#include <stdlib.h>

#define N  4
#define P  4
#define M 4
int a[N][P] =  {1,2,3,4,5,6,7,8,9,10,11,12,14,14,15,16}, c[N][M];
int b[P][M] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

void main (void) {
      int i, j , l;
      for (i = 0 ; i<N ; i++) {
          for (j=0 ; j<M ;j++) {
                    c[i][j] = 0;
                    for (l = 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
          }
      }
}
```

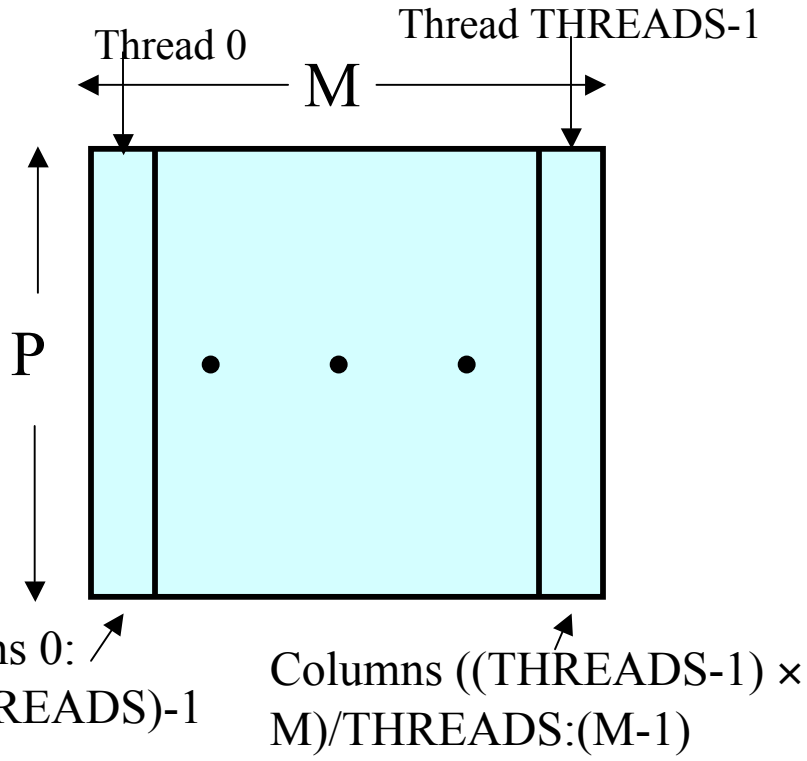
Domain Decomposition for UPC

Exploiting locality in matrix multiplication

- **A (N × P) is decomposed row-wise into blocks of size (N × P) / THREADS as shown below:**



- **B(P × M) is decomposed column-wise into M/ THREADS blocks as shown below:**



- **Note:** N and M are assumed to be multiples of THREADS

UPC Matrix Multiplication Code

```
#include <upc_relaxed.h>
#define N  4
#define P  4
#define M 4

shared [N*P /THREADS] int a[N][P];
shared [N*M /THREADS] int c[N][M];
// a and c are blocked shared matrices, initialization is not currently implemented
shared[M/THREADS] int b[P][M];
void main (void) {
      int i, j , l; // private variables

      upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
                for (j=0 ; j<M ;j++) {
                                  c[i][j] = 0;
                                  for (l= 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
}     }         }
```

UPC Matrix Multiplication Code with Privatization

```
#include <upc_relaxed.h>
#define N  4
#define P  4
#define M 4

shared [N*P /THREADS] int a[N][P]; // N, P and M divisible by THREADS
shared [N*M /THREADS] int c[N][M];
shared[M/THREADS] int b[P][M];
int *a_priv, *c_priv;
void main (void) {
      int i, j , l; // private variables
      upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
            a_priv = (int *)a[i]; c_priv = (int *)c[i];
            for (j=0 ; j<M ;j++) {
                  c_priv[j] = 0;
                  for (l= 0 ; l<P ; l++)
                        c_priv[j] += a_priv[l]*b[l][j];
}     }        }
```

UPC Matrix Multiplication Code with Block Transfer

```
#include <upc_relaxed.h>
shared [N*P /THREADS] int a[N][P];
shared [N*M /THREADS] int c[N][M];
// a and c are blocked shared matrices, initialization is not currently implemented
shared[M/THREADS] int b[P][M];
int b_local[P][M];

void main (void) {
        int i, j , l; // private variables

        for( i=0; i<P; i++ )
                for( j=0; j<THREADS; j++ )
                                upc_memget(&b_local[i][j*(M/THREADS)], &b[i][j*(M/THREADS)], (M/THREADS)*sizeof(int));
        upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
                for (j=0 ; j<M ;j++) {
                                c[i][j] = 0;
                                for (l= 0 ; l<P ; l++) c[i][j] += a[i][l]*b_local[l][j];
}       }       }
```

UPC Matrix Multiplication Code with Privatization and Block Transfer

```
#include <upc_relaxed.h>
shared [N*P /THREADS] int a[N][P]; // N, P and M divisible by THREADS
shared [N*M /THREADS] int c[N][M];
shared[M/THREADS] int b[P][M];
int *a_priv, *c_priv, b_local[P][M];
void main (void) {
      int i, priv_i, j , l; // private variables

      for( i=0; i<P; i++ )
            for( j=0; j<THREADS; j++ )
            upc_memget(&b_local[i][j*(M/THREADS)], &b[i][j*(M/THREADS)], (M/THREADS)*sizeof(int));

      upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
            a_priv = (int *)a[i]; c_priv = (int *)c[i];
            for (j=0 ; j<M ;j++) {
                        c_priv[j] = 0;
                        for (l= 0 ; l<P ; l++)
                                    c_priv[j] += a_priv[l]*b_local[l][j];
}     }        }
```

Matrix Multiplication with dynamic memory

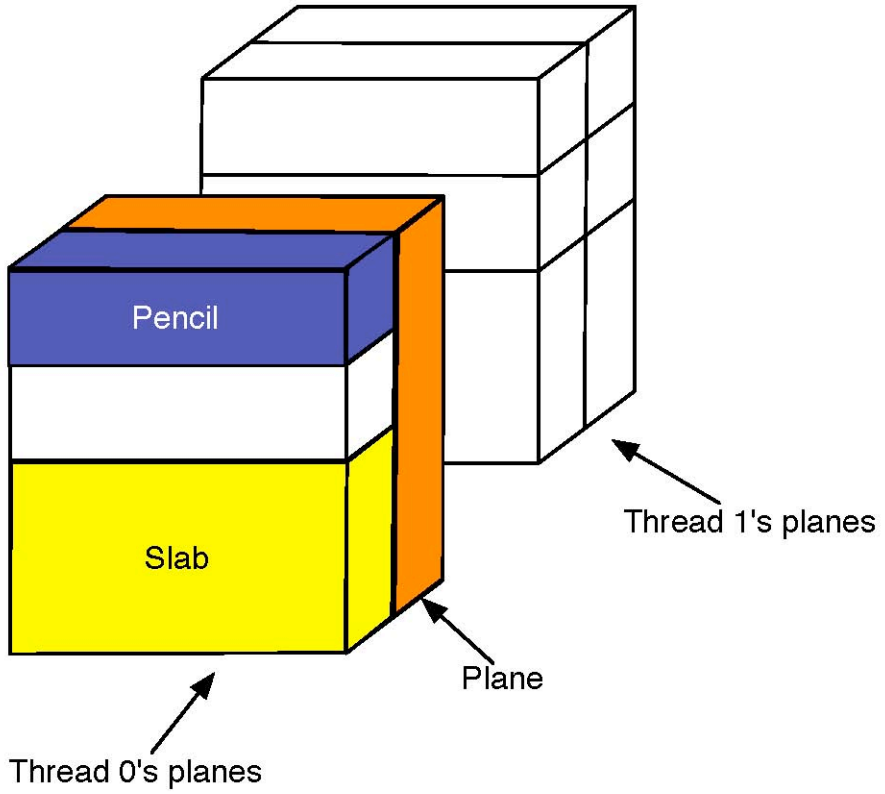
```
#include <upc_relaxed.h>
shared [N*P /THREADS] int *a;
shared [N*M /THREADS] int *c;
shared [M/THREADS] int *b;

void main (void) {
        int i, j , l; // private variables

        a=upc_all_alloc(THREADS,(N*P/THREADS)*upc_elemsizeof(*a));
        c=upc_all_alloc(THREADS,(N*M/THREADS)* upc_elemsizeof(*c));
        b=upc_all_alloc(P*THREADS, (M/THREADS)*upc_elemsizeof(*b));

        upc_forall(i = 0 ; i<N ; i++; &c[i*M]) {
                for (j=0 ; j<M ;j++) {
                        c[i*M+j] = 0;
                        for (l= 0 ; l<P ; l++) c[i*M+j] += a[i*P+l]*b[l*M+j];
}       }       }
```

Performance of UPC

Case Study: NAS FT

- **Performance of Exchange (Alltoall) is critical**
  - —1D FFTs in each dimension, 3 phases
  - —Transpose after first 2 for locality
  - —Bisection bandwidth-limited
    - – Problem as #procs grows

- **Three approaches:**
  - —Exchange:
    - – wait for 2nd dim FFTs to finish, send 1 message per processor pair
  - —Slab:
    - – wait for chunk of rows destined for 1 proc, send when ready
  - —Pencil:
    - – send each row as it completes



Source: Kathy Yelick, Chris Bell, Rajesh Nishtala, Dan Bonachea

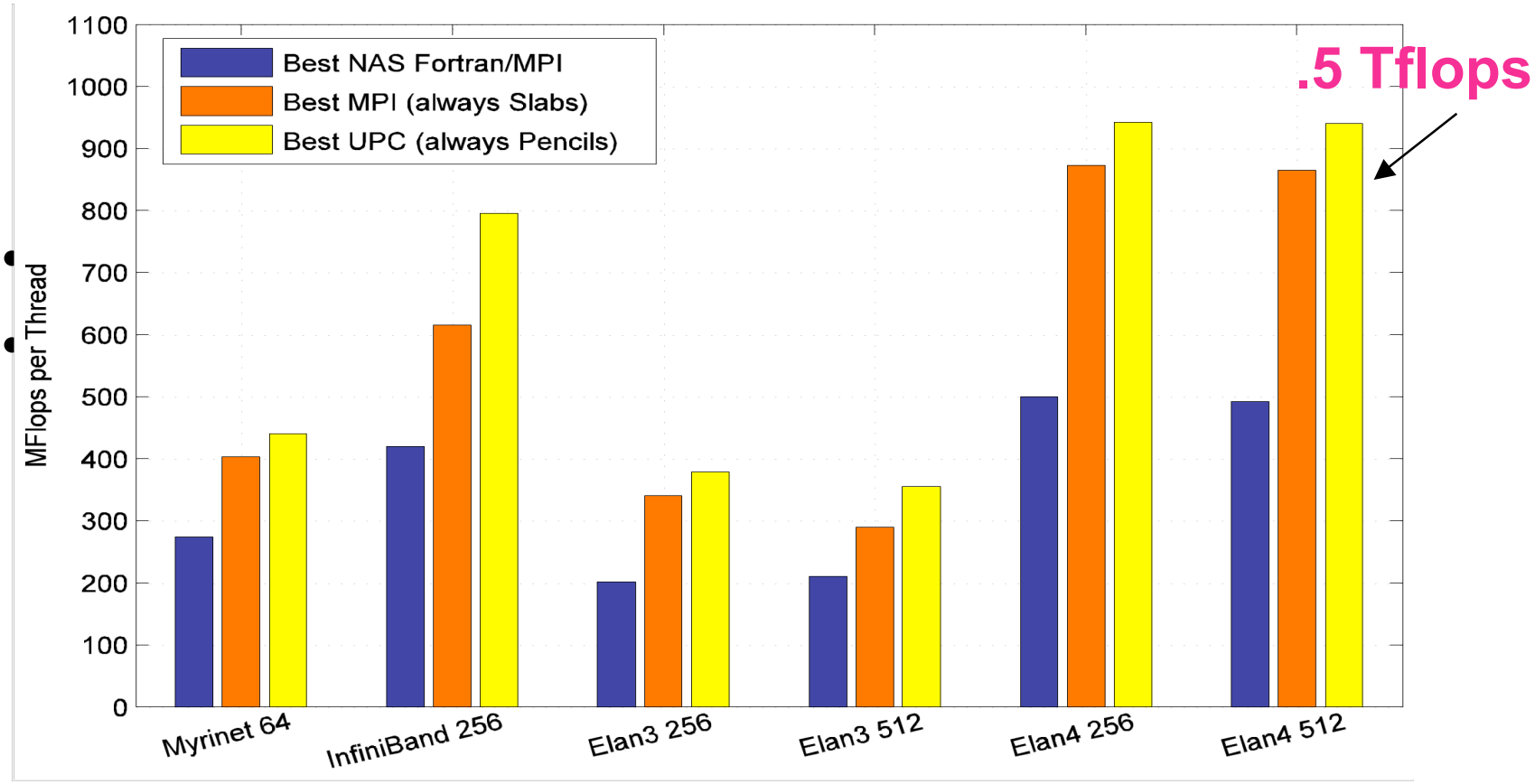
Overlapping Communication

- Goal: make use of "all the wires all the time"
  - —Schedule communication to avoid network backup
- Trade-off: overhead vs. overlap
  - —Exchange has fewest messages, less message overhead
  - —Slabs and pencils have more overlap; pencils the most
- Example: Class D problem on 256 Processors

Exchange (all data at once)	512 Kbytes
Slabs (contiguous rows that go to 1 processor)	64 Kbytes
Pencils (single row)	16 Kbytes

Source: Kathy Yelick, Chris Bell, Rajesh Nishtala, Dan Bonachea

NAS FT Variants Performance Summary



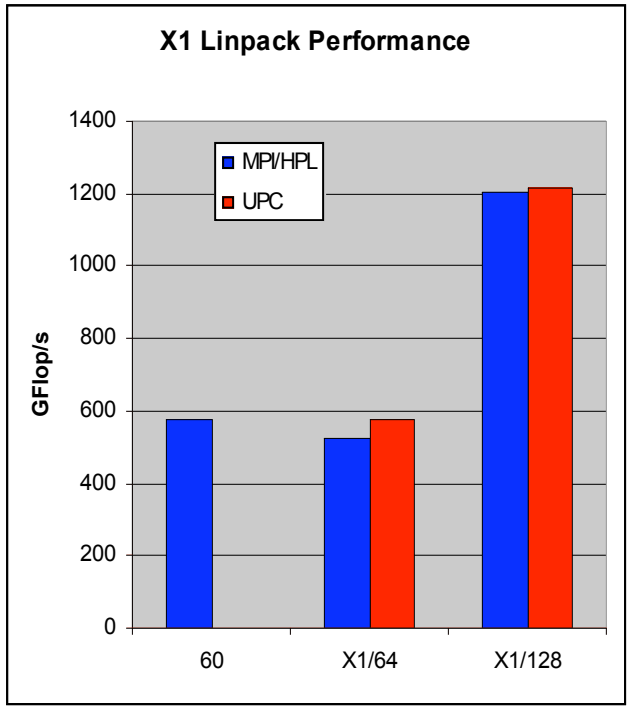
Source: Kathy Yelick, Chris Bell, Rajesh Nishtala, Dan Bonachea

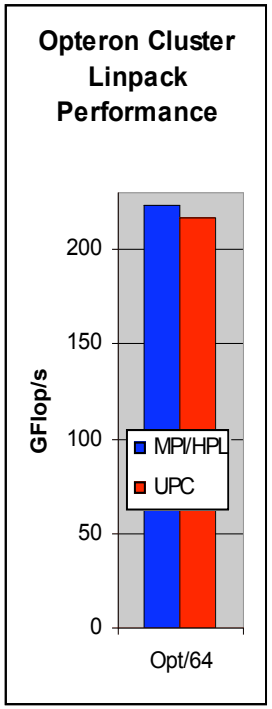
Case Study: LU Factorization

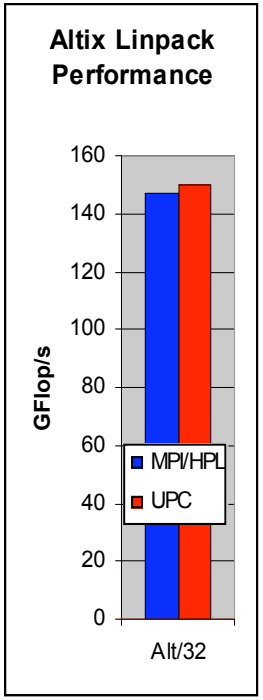
- Direct methods have complicated dependencies
  - —Especially with pivoting (unpredictable communication)
  - —Especially for sparse matrices (dependence graph with holes)
- LU Factorization in UPC
  - —Use overlap ideas and multithreading to mask latency
  - —Multithreaded: UPC threads + user threads + threaded BLAS
    - Panel factorization: Including pivoting
    - Update to a block of U
    - Trailing submatrix updates
- Status:
  - —Dense LU done: HPL-compliant
  - —Sparse version underway

Source: Kathy Yelick, Parry Husbands

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)**

Source: Kathy Yelick, Parry Husbands

Concluding Remarks

- UPC is relatively comfortable for people proficient in C
- UPC and PGAS programming model combines
  - —ease of shared memory programming
  - —flexibility of message passing
- Hand tuning may sometimes make parts of the code slightly looking like message passing
  - —when necessary to make local copies of data
  - —unlikely to be necessary for the majority of the data

UPC Resources

- **UPC WWW site http://upc.gwu.edu**
  - —**Documentation, tutorials, compilers, test suites**
- **Documentation: http://upc.gwu.edu/documentation.html**
  - —Language specification v1.2 completed June 2005
  - —UPC Manual, May 2005
  - —Getting Started with UPC, June 2001
  - —UPC Quick Reference Card
  - —UPC Book: Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick, UPC: Distributed Shared Memory Programming, John Wiley & Sons, ISBN: 978-0-471-22048-0, June 2005.