

SC|05 Tutorial

High Performance Parallel Programming with Unified Parallel C (UPC)

Tarek El-Ghazawi
tarek@gwu.edu



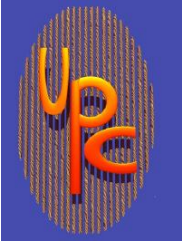
THE GEORGE
WASHINGTON
UNIVERSITY
WASHINGTON DC

The George Washington U.

Phillip Merkey
Steve Seidel
{merk,steve}@mtu.edu

Michigan Technological U.
MichiganTech



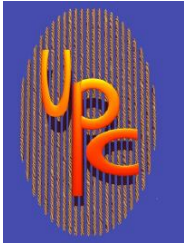


UPC Tutorial Web Site

This site contains the UPC code segments discussed in this tutorial.

<http://www.upc.mtu.edu/SC05-tutorial>





UPC Home Page

<http://www.upc.gwu.edu>



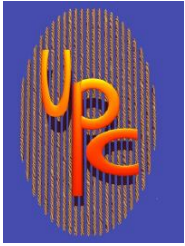
The High Performance Computing Laboratory

Department of Electrical and Computer Engineering
School of Engineering and Applied Science
The George Washington University

UNIFIED PARALLEL C

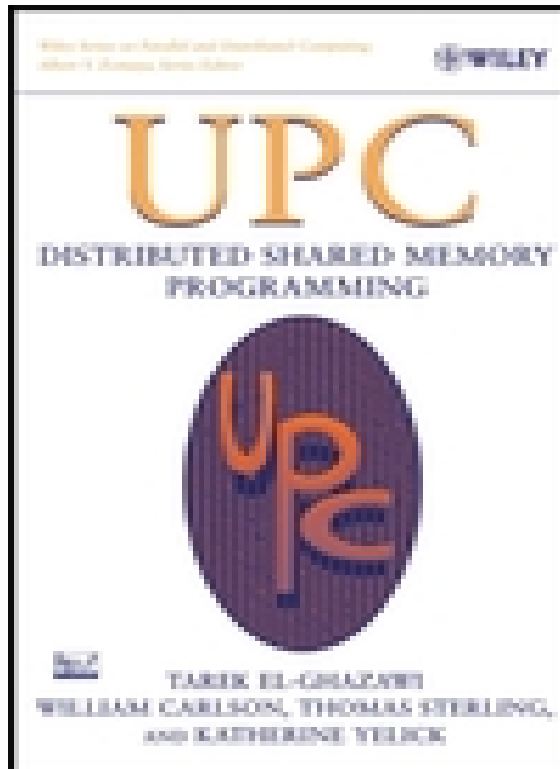
Projects	The logo for Unified Parallel Computing (UPC) features the letters 'UPC' in a stylized, orange, 3D font. The letters are set against a blue circular background with a grid of small, vertical, golden-brown lines.	News
Tutorials		Forum
Publications		Events
Documentation		Working Groups
Downloads		FAQ





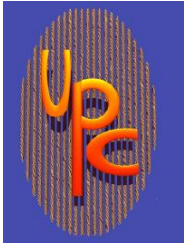
UPC textbook now available

<http://www.upcworld.org>



- *UPC: Distributed Shared Memory Programming*
Tarek El-Ghazawi
William Carlson
Thomas Sterling
Katherine Yelick
- Wiley, May, 2005
- ISBN: 0-471-22048-5

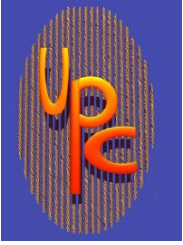




Section 1: The UPC Language

- Introduction *El-Ghazawi*
- UPC and the PGAS Model
- Data Distribution
- Pointers
- Worksharing and Exploiting Locality
- Dynamic Memory Management
- (10:15am - 10:30am break)
- Synchronization
- Memory Consistency



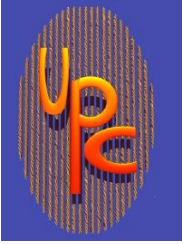


Section 2: UPC Systems

Merkey & Seidel

- Summary of current UPC systems
 - Cray X-1
 - Hewlett-Packard
 - Berkeley
 - Intrepid
 - MTU
- UPC application development tools
 - totalview
 - upc_trace
 - performance toolkit interface
 - performance model





Section 3: UPC Libraries

Seidel

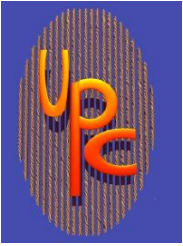
- Collective Functions
 - Bucket sort example
 - UPC collectives
 - Synchronization modes
 - Collectives performance
 - Extensions

Noon – 1:00pm lunch

- UPC-IO
 - Concepts
 - Main Library Calls
 - Library Overview

El-Ghazawi





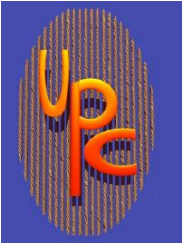
Sec. 4: UPC Applications Development

- Two case studies of application design *Merkey*
 - histogramming
 - locks revisited
 - generalizing the histogram problem
 - programming the sparse case
 - implications of the memory model

(2:30pm – 2:45pm break)

- generic science code (advection):
 - shared multi-dimensional arrays
 - implications of the memory model
- UPC tips, tricks, and traps *Seidel*

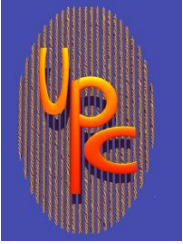




Introduction

- UPC – Unified Parallel C
- Set of specs for a parallel C
 - v1.0 completed February of 2001
 - v1.1.1 in October of 2003
 - v1.2 in May of 2005
- Compiler implementations by vendors and others
- Consortium of government, academia, and HPC vendors including IDA CCS, GWU, UCB, MTU, UMN, ARSC, UMCP, U of Florida, ANL, LBNL, LLNL, DoD, DoE, HP, Cray, IBM, Sun, Intrepid, Etnus, ...

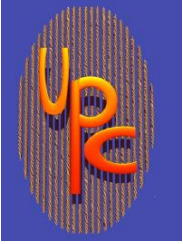




Introductions cont.

- UPC compilers are now available for most HPC platforms and clusters
 - Some are open source
- A debugger is available and a performance analysis tool is in the works
- Benchmarks, programming examples, and compiler testing suite(s) are available
- Visit www.upcworld.org or upc.gwu.edu for more information

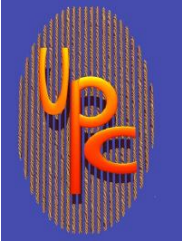




Parallel Programming Models

- What is a programming model?
 - An abstract virtual machine
 - A view of data and execution
 - The logical interface between architecture and applications
- Why Programming Models?
 - Decouple applications and architectures
 - Write applications that run effectively across architectures
 - Design new architectures that can effectively support legacy applications
- Programming Model Design Considerations
 - Expose modern architectural features to exploit machine power and improve performance
 - Maintain Ease of Use

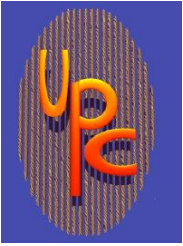




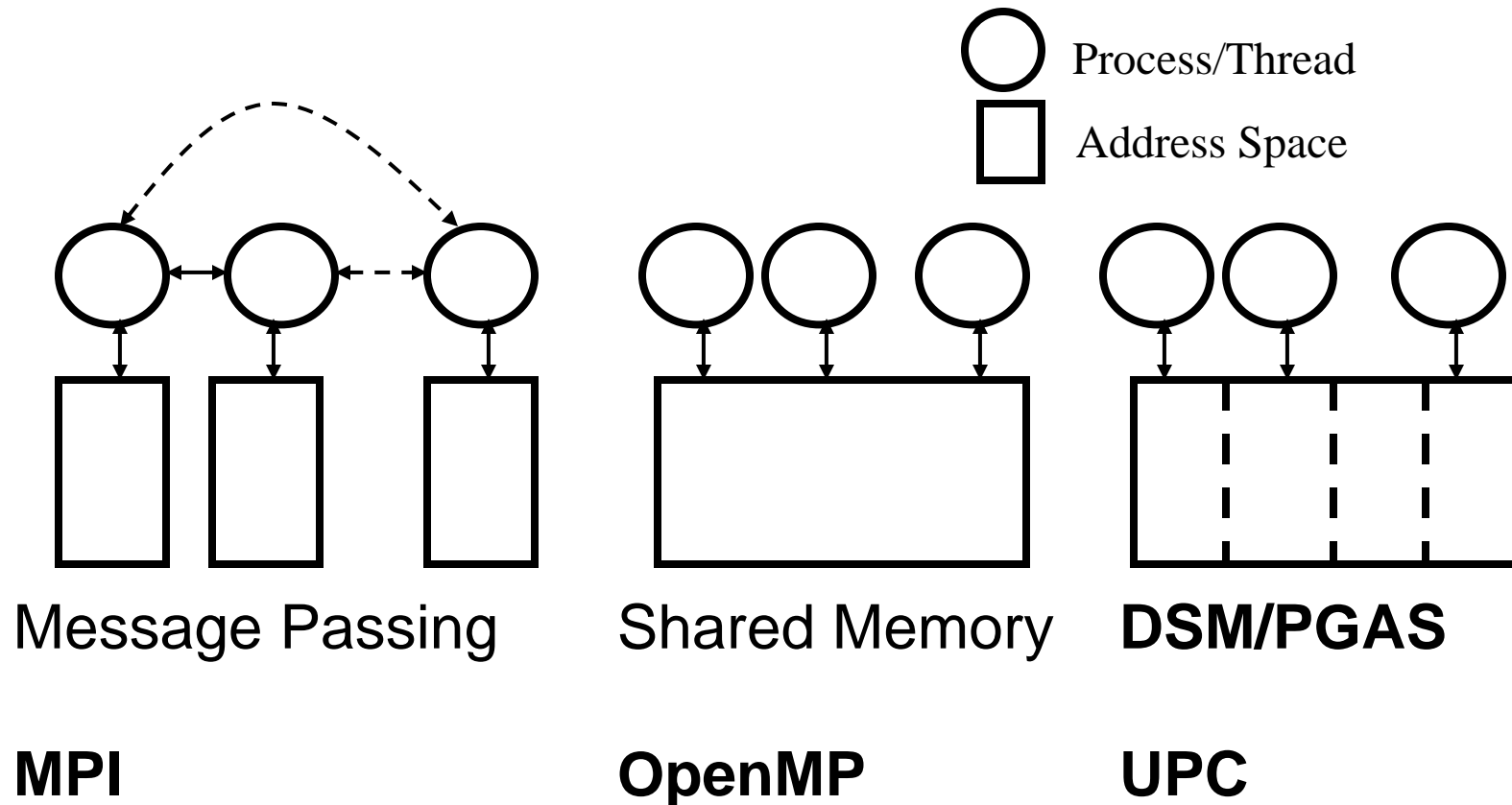
Programming Models

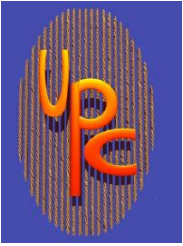
- Common Parallel Programming models
 - Data Parallel
 - Message Passing
 - Shared Memory
 - Distributed Shared Memory
 - ...
- Hybrid models
 - Shared Memory under Message Passing
 - ...



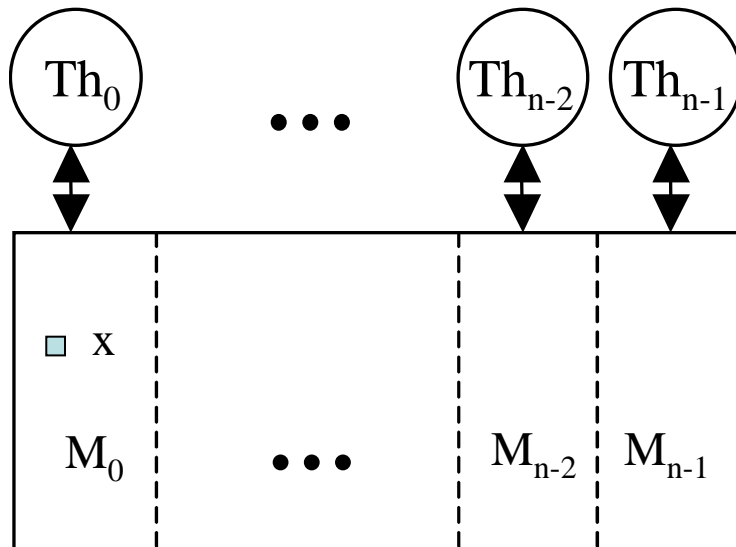


Programming Models





The Partitioned Global Address Space (PGAS) Model



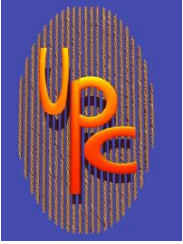
Legend:

○ Thread/Process → Memory Access

□ Address Space



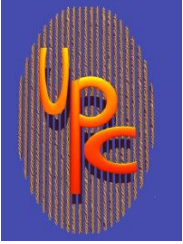
- Aka the DSM model
- Concurrent threads with a partitioned shared space
 - Similar to the shared memory
 - Memory partition M_i has affinity to thread Th_i
- (+)ive:
 - Helps exploiting locality
 - Simple statements as SM
- (-)ive:
 - Synchronization
- UPC, also CAF and Titanium



What is UPC?

- Unified Parallel C
- An explicit parallel extension of ISO C
- A partitioned shared memory parallel programming language

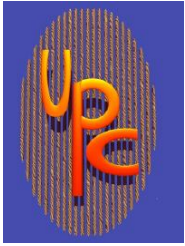




UPC Execution Model

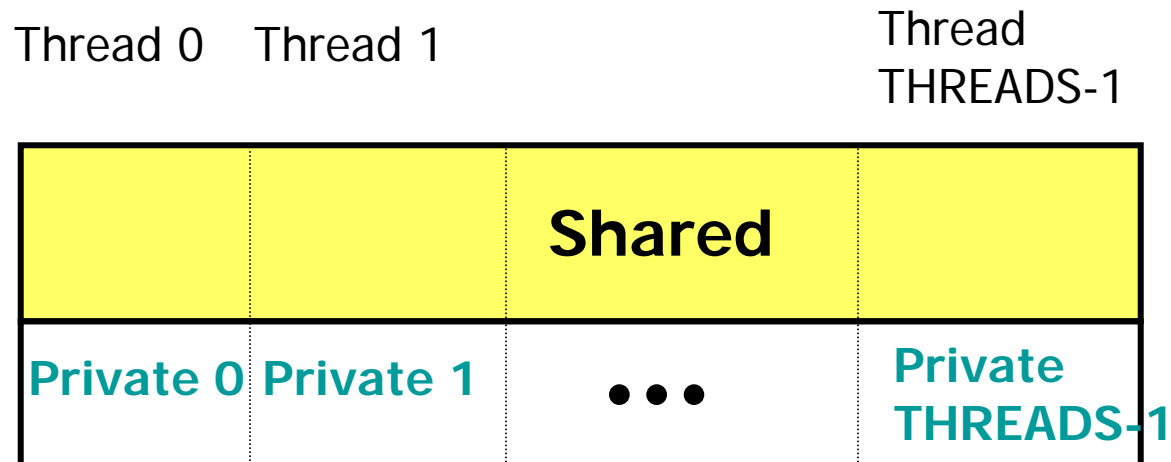
- A number of threads working independently in a SPMD fashion
 - MYTHREAD specifies thread index (0..THREADS-1)
 - Number of threads specified at compile-time or run-time
- Synchronization when needed
 - Barriers
 - Locks
 - Memory consistency control





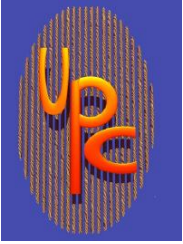
UPC Memory Model

Private Spaces
Partitioned Global address space



- A pointer-to-shared can reference all locations in the shared space, but there is data-thread **affinity**
- A private pointer may reference addresses in its private space or its local portion of the shared space
- Static and dynamic memory allocations are supported for both shared and private memory

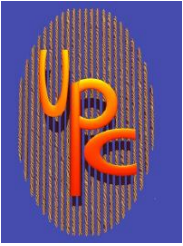




User's General View

A collection of threads operating in a single global address space, which is logically partitioned among threads. Each thread has affinity with a portion of the globally shared address space. Each thread has also a private space.





A First Example: Vector addition

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

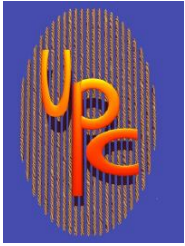
shared int v1[N], v2[N], v1plusv2[N];
void main() {
    int i;
    for(i=0; i<N; i++)
        if (MYTHREAD==i%THREADS)
            v1plusv2[i]=v1[i]+v2[i];
}
```

Iteration #:

Thread 0	Thread 1
0	1
2	3
v1[0]	v1[1]
v1[2]	v1[3]
...	
v2[0]	v2[1]
v2[2]	v2[3]
...	
v1plusv2[0]	v1plusv2[1]
v1plusv2[2]	v1plusv2[3]
...	

Shared Space





2nd Example: A More Efficient Implementation

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

shared int v1[N], v2[N], v1plusv2[N];
void main() {
    int i;
    for(i=MYTHREAD; i<N; i+=THREADS)

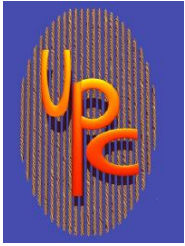
        v1plusv2[i]=v1[i]+v2[i];
}
```

Iteration #:

Thread 0	Thread 1
0	1
2	3
v1[0]	v1[1]
v1[2]	v1[3]
...	
v2[0]	v2[1]
v2[2]	v2[3]
...	
v1plusv2[0]	v1plusv2[1]
v1plusv2[2]	v1plusv2[3]
...	

Shared Space





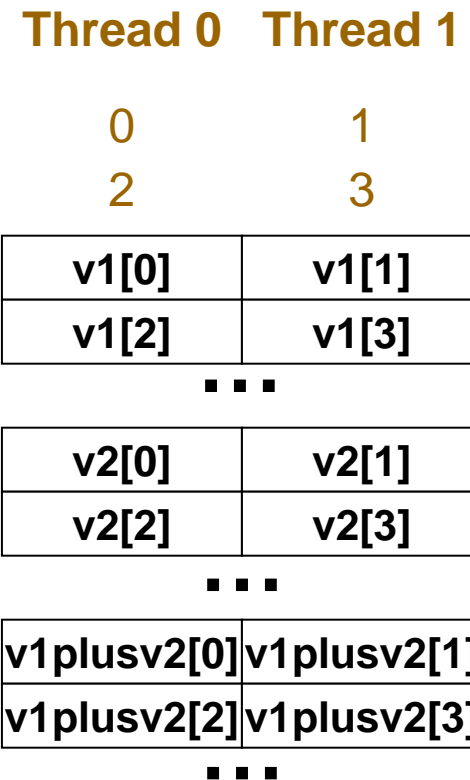
3rd Example: A More Convenient Implementation with upc_forall

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

shared int v1[N], v2[N], v1plusv2[N];

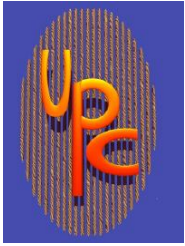
void main()
{
    int i;
    upc_forall(i=0; i<N; i++; i)
        v1plusv2[i]=v1[i]+v2[i];
}
```

Iteration #:



Shared Space



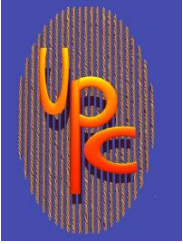


Example: UPC Matrix-Vector Multiplication- Default Distribution

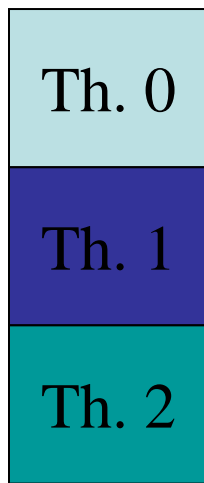
```
// vect_mat_mult.c
#include <upc_relaxed.h>

shared int a[THREADS][THREADS] ;
shared int b[THREADS], c[THREADS] ;
void main (void) {
    int i, j;
    upc_forall( i = 0 ; i < THREADS ; i++)
    {
        c[i] = 0;
        for ( j= 0 ; j < THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```



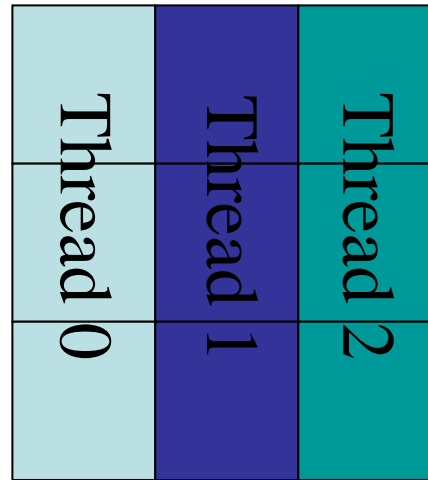


Data Distribution



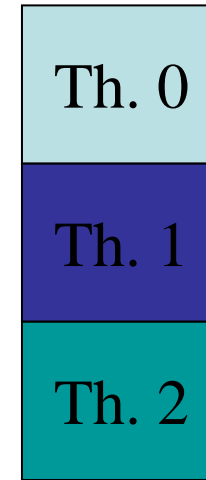
C

=

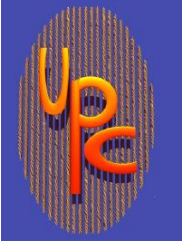


A

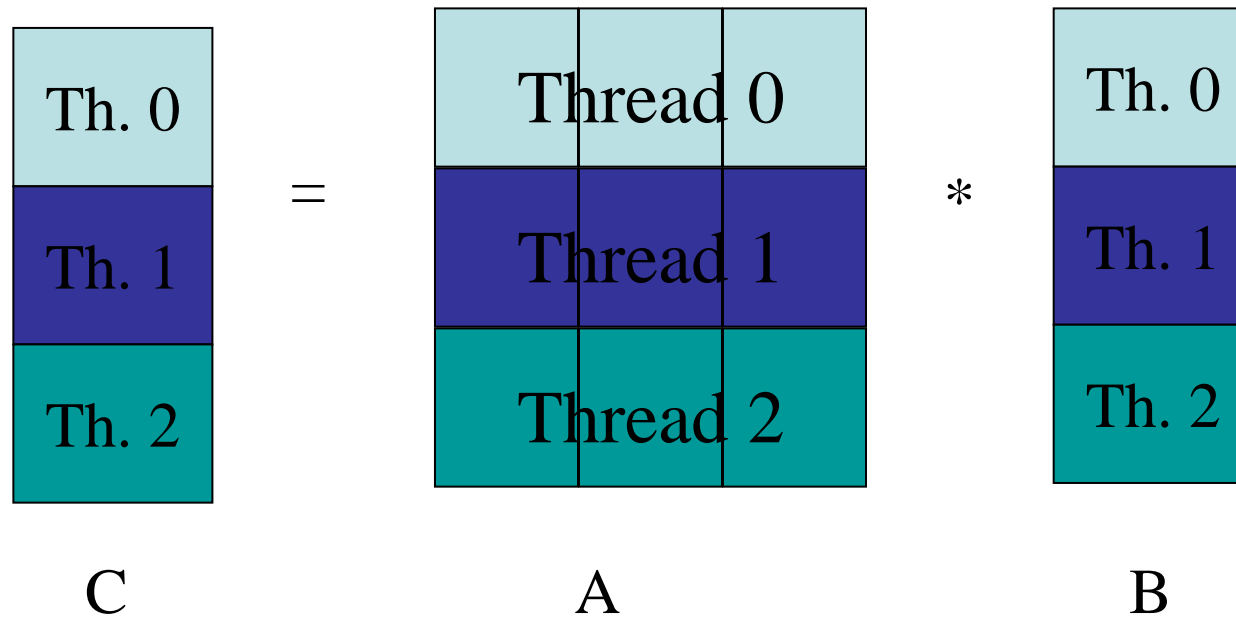
*

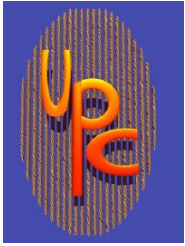


B



A Better Data Distribution





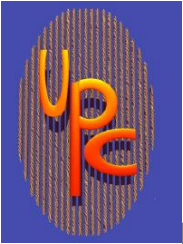
Example: UPC Matrix-Vector Multiplication- The Better Distribution

```
// vect_mat_mult.c
#include <upc_relaxed.h>

shared [THREADS] int a[THREADS][THREADS];
shared int b[THREADS], c[THREADS];

void main (void) {
    int i, j;
    upc_forall( i = 0 ; i < THREADS ; i++; i)
    {
        c[i] = 0;
        for ( j= 0 ; j< THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```





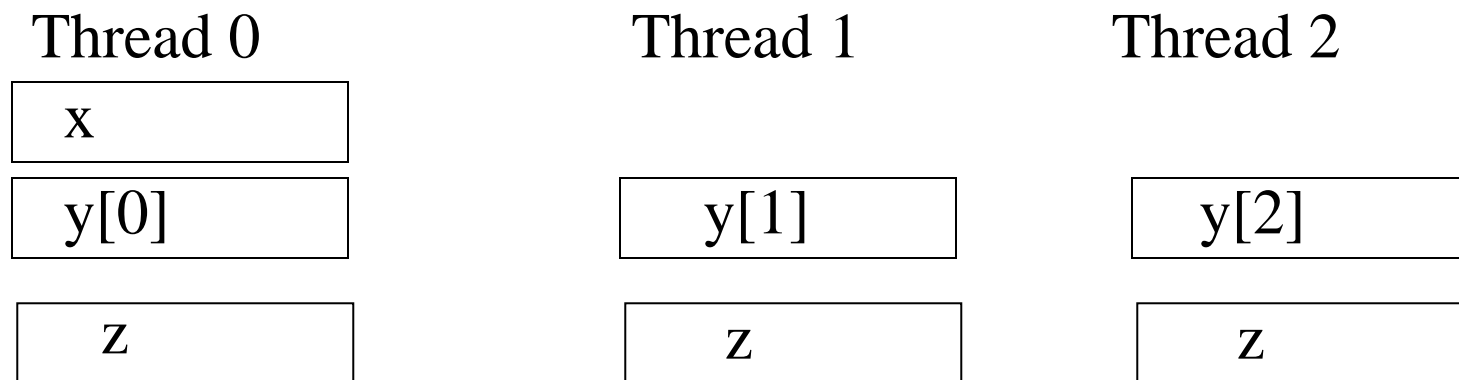
Shared and Private Data

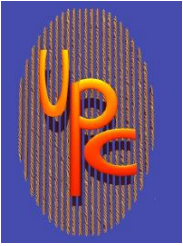
Examples of Shared and Private Data Layout:

Assume THREADS = 3

```
shared int x; /*x will have affinity to thread 0 */  
shared int y[THREADS];  
int z;
```

will result in the layout:





Shared and Private Data

```
shared int A[4][THREADS];
```

will result in the following data layout:

Thread 0

A[0][0]
A[1][0]
A[2][0]
A[3][0]

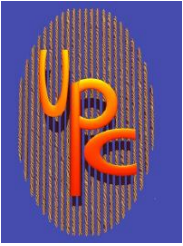
Thread 1

A[0][1]
A[1][1]
A[2][1]
A[3][1]

Thread 2

A[0][2]
A[1][2]
A[2][2]
A[3][2]





Shared and Private Data

```
shared int A[2][2*THREADS];
```

will result in the following data layout:

Thread 0

A[0][0]
A[0][THREADS]
A[1][0]
A[1][THREADS]

Thread 1

A[0][1]
A[0][THREADS+1]
A[1][1]
A[1][THREADS+1]

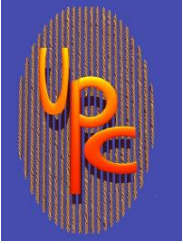
...

Thread (THREADS-1)

A[0][THREADS-1]
A[0][2*THREADS-1]
A[1][THREADS-1]
A[1][2*THREADS-1]

...

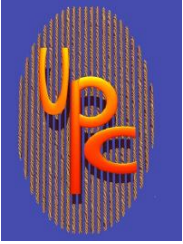




Blocking of Shared Arrays

- Default block size is 1
- Shared arrays can be distributed on a block per thread basis, round robin with arbitrary block sizes.
- A block size is specified in the declaration as follows:
 - `shared [block-size] type array[N];`
 - e.g.: `shared [4] int a[16];`

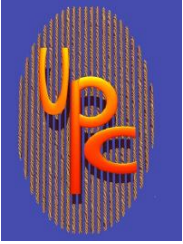




Blocking of Shared Arrays

- Block size and THREADS determine affinity
- The term affinity means in which thread's local shared-memory space, a shared data item will reside
- Element i of a blocked array has affinity to thread:

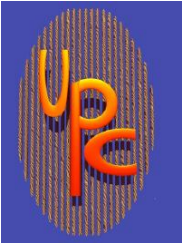
$$\left\lfloor \frac{i}{\text{blocksize}} \right\rfloor \bmod \text{THREADS}$$



Shared and Private Data

- Shared objects placed in memory based on affinity
- Affinity can be also defined based on the ability of a thread to refer to an object by a private pointer
- All non-array shared qualified objects, i.e. shared scalars, have affinity to thread 0
- Threads access shared and private data





Shared and Private Data

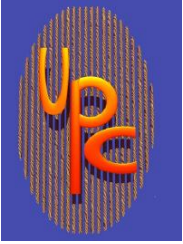
Assume THREADS = 4

```
shared [3] int A[4][THREADS];
```

will result in the following data layout:

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

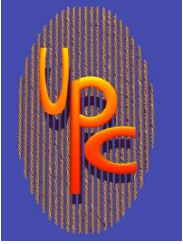




Special Operators

- `upc_localsizeof(type-name or expression);`
returns the size of the local portion of a shared object
- `upc_blocksizeof(type-name or expression);`
returns the blocking factor associated with the argument
- `upc_elemsizeof(type-name or expression);`
returns the size (in bytes) of the left-most type that is not an array



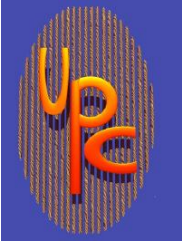


Usage Example of Special Operators

```
typedef shared int sharray[10*THREADS];  
sharray a;  
char i;
```

- `upc_localsizeof(sharray) → 10*sizeof(int)`
- `upc_localsizeof(a) → 10 *sizeof(int)`
- `upc_localsizeof(i) → 1`
- `upc_blocksizeof(a) → 1`
- `upc_elementsizeof(a) → sizeof(int)`

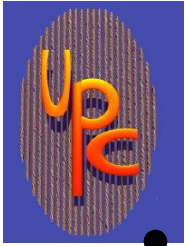




String functions in UPC

- UPC provides standard library functions to move data to/from shared memory
- Can be used to move chunks in the shared space or between shared and private spaces

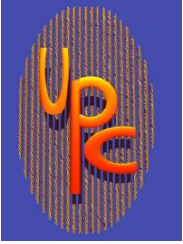




String functions in UPC

- Equivalent of memcpy :
 - `upc_memcpy(dst, src, size)`
 - copy from shared to shared
 - `upc_memput(dst, src, size)`
 - copy from private to shared
 - `upc_memget(dst, src, size)`
 - copy from shared to private
 - Equivalent of memset:
 - `upc_memset(dst, char, size)`
 - initializes shared memory with a character
 - The shared block must be a contiguous with all of its elements having the same affinity
-



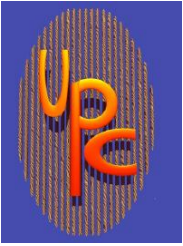


UPC Pointers

Where does it point to?

		Private	Shared
Where does it reside?	Private	PP	PS
Shared		SP	SS

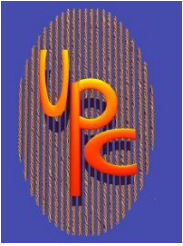




UPC Pointers

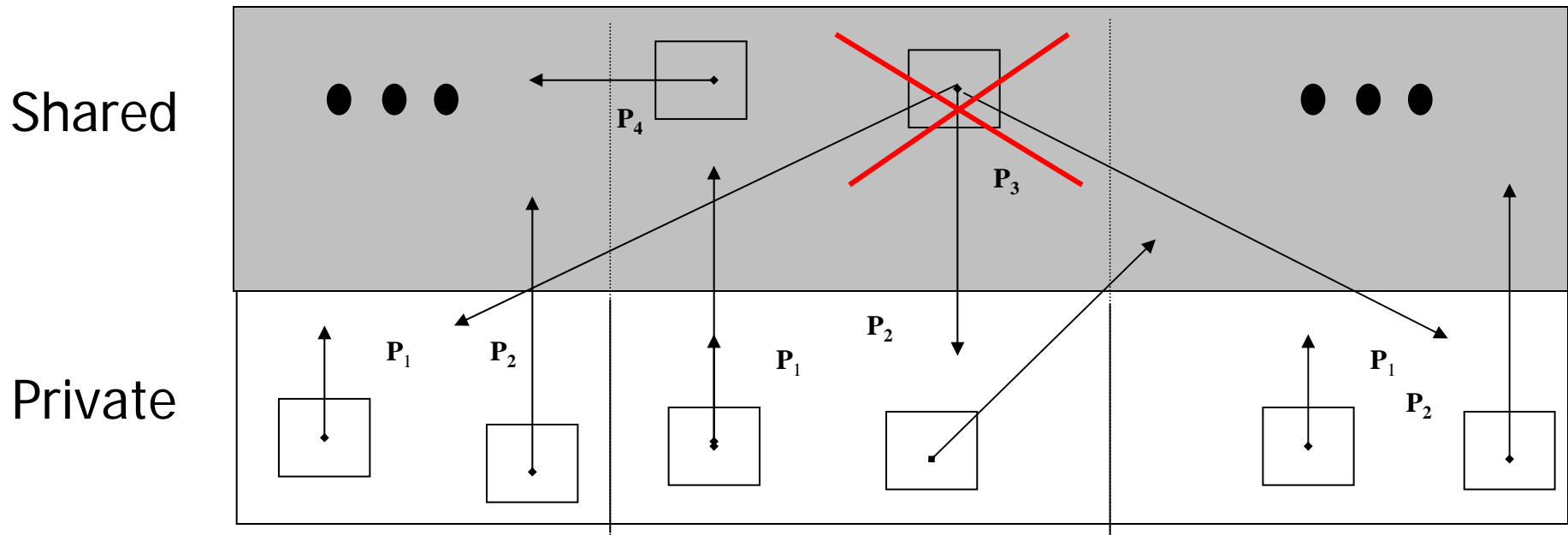
- How to declare them?
 - `int *p1; /* private pointer pointing locally */`
 - `shared int *p2; /* private pointer pointing into the shared space */`
 - ~~▪ `int *shared p3; /* shared pointer pointing locally */`~~
 - `shared int *shared p4; /* shared pointer pointing into the shared space */`
- You may find many using “shared pointer” to mean a pointer pointing to a shared object, e.g. equivalent to p2 but could be p4 as well.

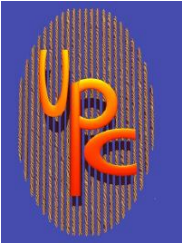




UPC Pointers

Thread 0

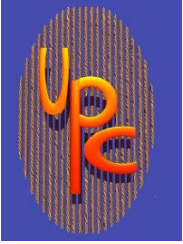




UPC Pointers

- What are the common usages?
 - `int *p1; /* access to private data or to local shared data */`
 - `shared int *p2; /* independent access of threads to data in shared space */`
 - `int *shared p3; /* not recommended*/`
 - `shared int *shared p4; /* common access of all threads to data in the shared space*/`



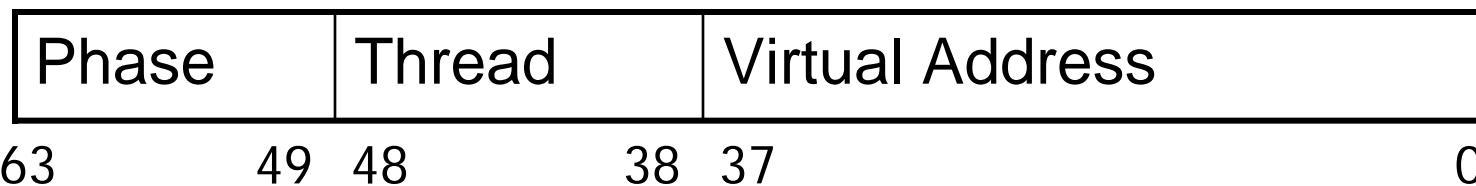


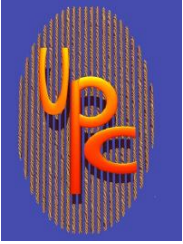
UPC Pointers

- 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

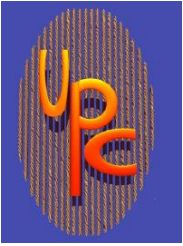




UPC Pointers

- 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 private pointer, the thread number of the pointer-to-shared may be lost
- Casting of a pointer-to-shared to a private pointer is well defined only if the pointed to object has affinity with the local thread

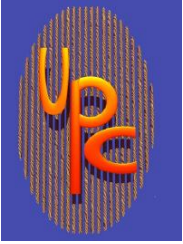




Special Functions

- `size_t upc_threadof(shared void *ptr);`
returns the thread number that has affinity to the object pointed to by `ptr`
- `size_t upc_phaseof(shared void *ptr);`
returns the index (position within the block) of the object which is pointed to by `ptr`
- `size_t upc_addrfield(shared void *ptr);`
returns the address of the block which is pointed at by the pointer to `shared`
- `shared void *upc_resetphase(shared void *ptr);`
resets the phase to zero
- `size_t upc_affinitysize(size_t ntotal, size_t nbytes, size_t thr);`
returns the exact size of the local portion of the data in a shared object with affinity to a given thread





UPC Pointers

pointer to shared Arithmetic Examples:

Assume THREADS = 4

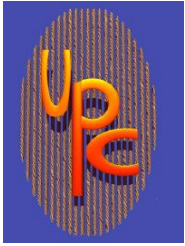
```
#define N 16
```

```
shared int x[N];
```

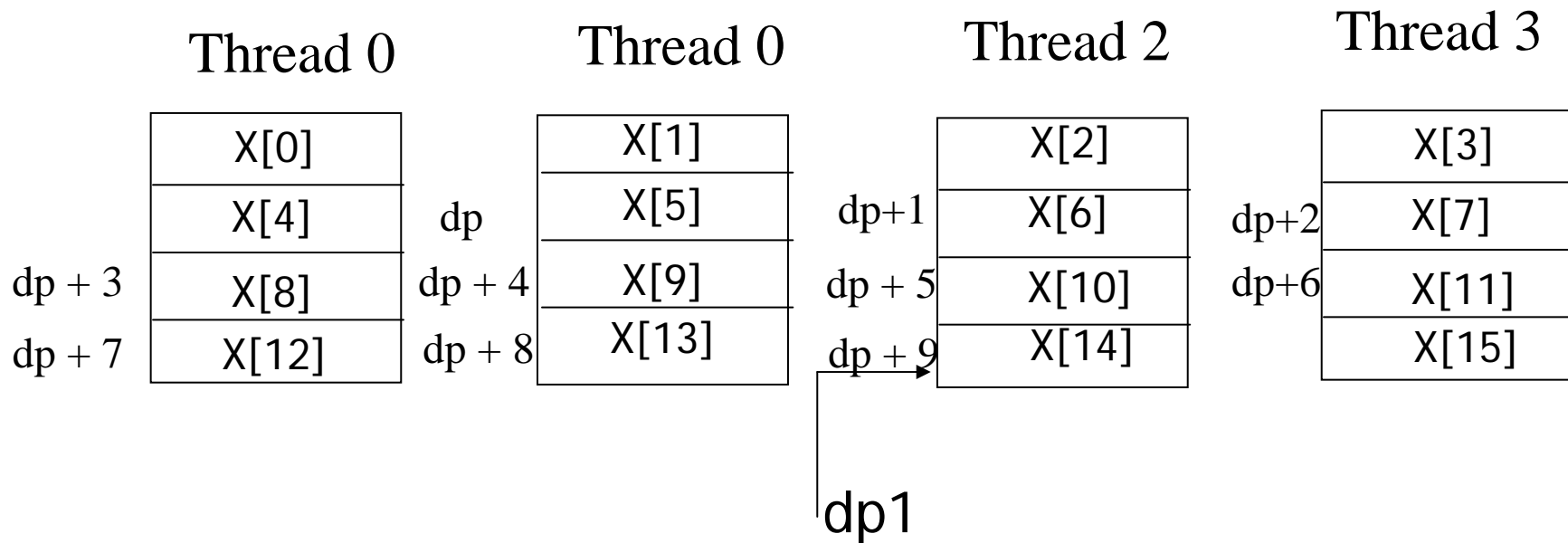
```
shared int *dp=&x[5], *dp1;
```

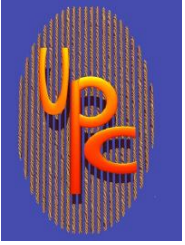
```
dp1 = dp + 9;
```





UPC Pointers



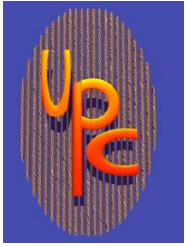


UPC Pointers

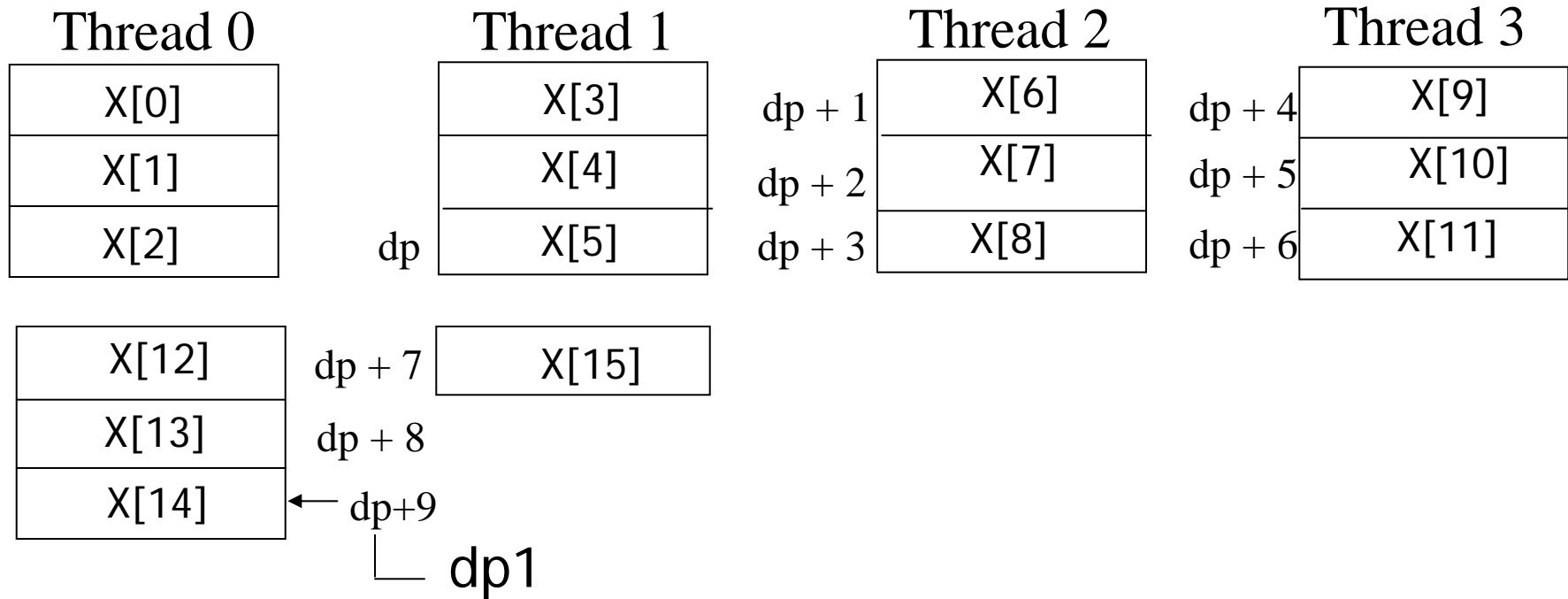
Assume `THREADS = 4`

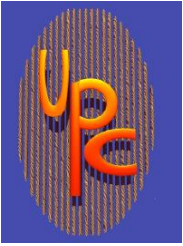
```
shared[3] int x[N], *dp=&x[5], *dp1;  
dp1 = dp + 9;
```





UPC Pointers





UPC Pointers

Example Pointer Castings and Mismatched Assignments:

- **Pointer Casting**

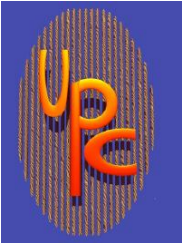
```
shared int x[THREADS];
```

```
int *p;
```

```
p = (int *) &x[MYTHREAD]; /* p points to  
x[MYTHREAD] */
```

- Each of the private pointers will point at the x element which has affinity with its thread, i.e. MYTHREAD





UPC Pointers

- Mismatched Assignments

Assume THREADS = 4

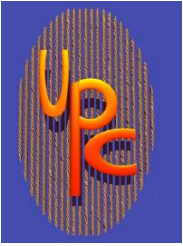
```
shared int x[N];
```

```
shared[3] int *dp=&x[5], *dp1;
```

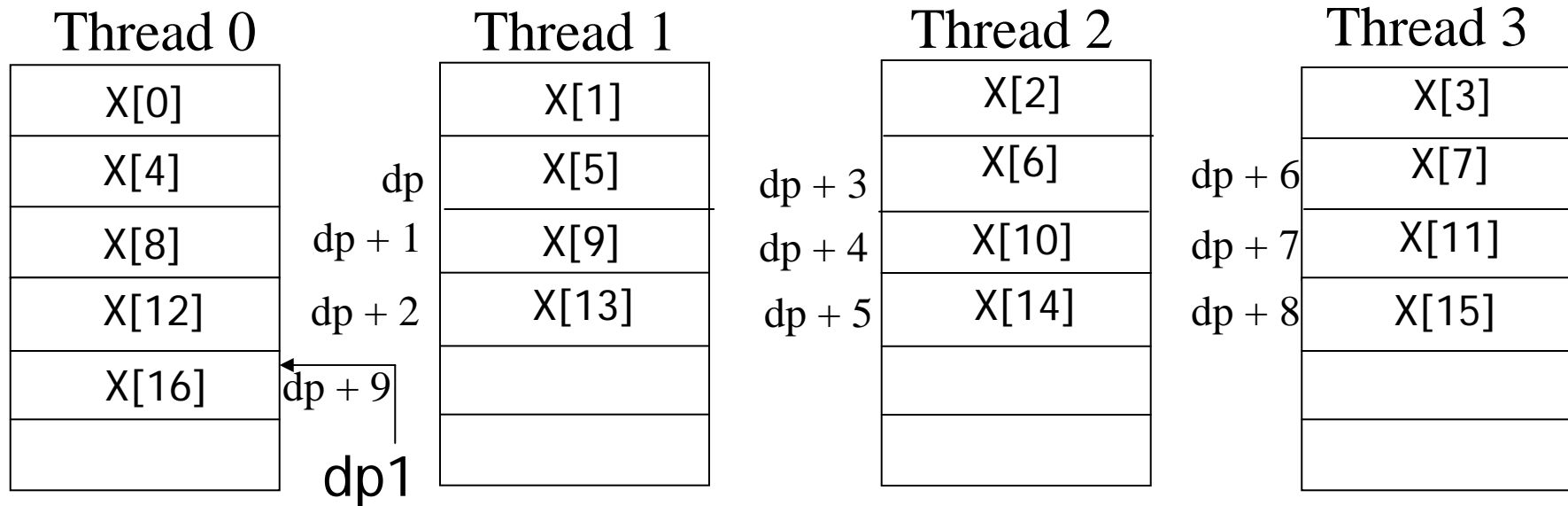
```
dp1 = dp + 9;
```

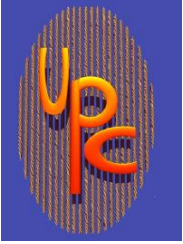
- The last statement assigns to dp1 a value that is 9 positions beyond dp
- The pointer will follow its own blocking and not that of the array





UPC Pointers





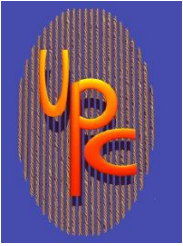
UPC Pointers

- Given the declarations

```
shared[3] int *p;  
shared[5] int *q;
```
- Then

```
p=q; /* is acceptable (an implementation may  
require an explicit cast, e.g. p=(*shared [3])q;) */
```
- Pointer p, however, will follow pointer arithmetic for blocks of 3, not 5 !!
- A pointer cast sets the phase to 0





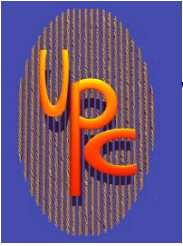
Worksharing with upc_forall

- Distributes independent iteration across threads in the way you wish— typically used to boost locality exploitation in a convenient way
- Simple C-like syntax and semantics

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

- Affinity could be an integer expression, or a
- Reference to (address of) a shared object





Work Sharing and Exploiting Locality via `upc_forall()`

- Example 1: explicit affinity using shared references

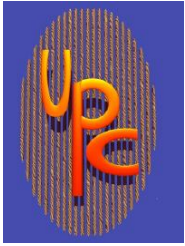
```
shared int a[100],b[100], c[100];  
int i;  
upc_forall (i=0; i<100; i++; &a[i])  
    a[i] = b[i] * c[i];
```

- Example 2: implicit affinity with integer expressions and distribution in a round-robin fashion

```
shared int a[100],b[100], c[100];  
int i;  
upc_forall (i=0; i<100; i++; i)  
    a[i] = b[i] * c[i];
```

Note: Examples 1 and 2 result in the same distribution



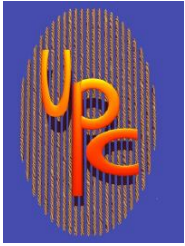


Work Sharing: upc_forall()

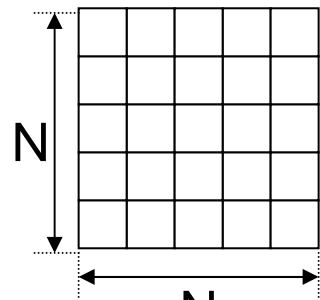
- Example 3: Implicitly with distribution by chunks
shared int a[100], b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
a[i] = b[i] * c[i];
- **Assuming 4 threads, the following results**

i	i*THREADS	i*THREADS/100
0..24	0..96	0
25..49	100..196	1
50..74	200..296	2
75..99	300..396	3



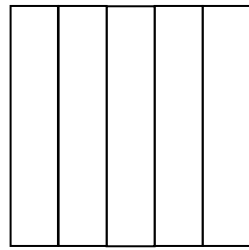


Distributing Multidimensional Data

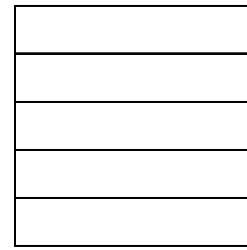


Default

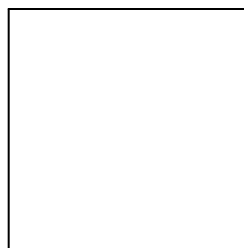
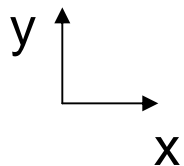
BLOCKSIZE=1



Column Blocks
BLOCKSIZE=
N/THREADS

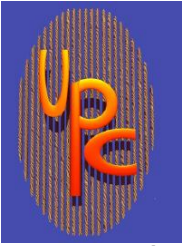


Distribution by
Row Block
BLOCKSIZE=N



BLOCKSIZE=N*N
or
BLOCKSIZE =
infinite

- Uses the inherent contiguous memory layout of C multidimensional arrays
- shared
[BLOCKSIZE]
double
grids[N][N];
 - Distribution depends on the value of BLOCKSIZE,

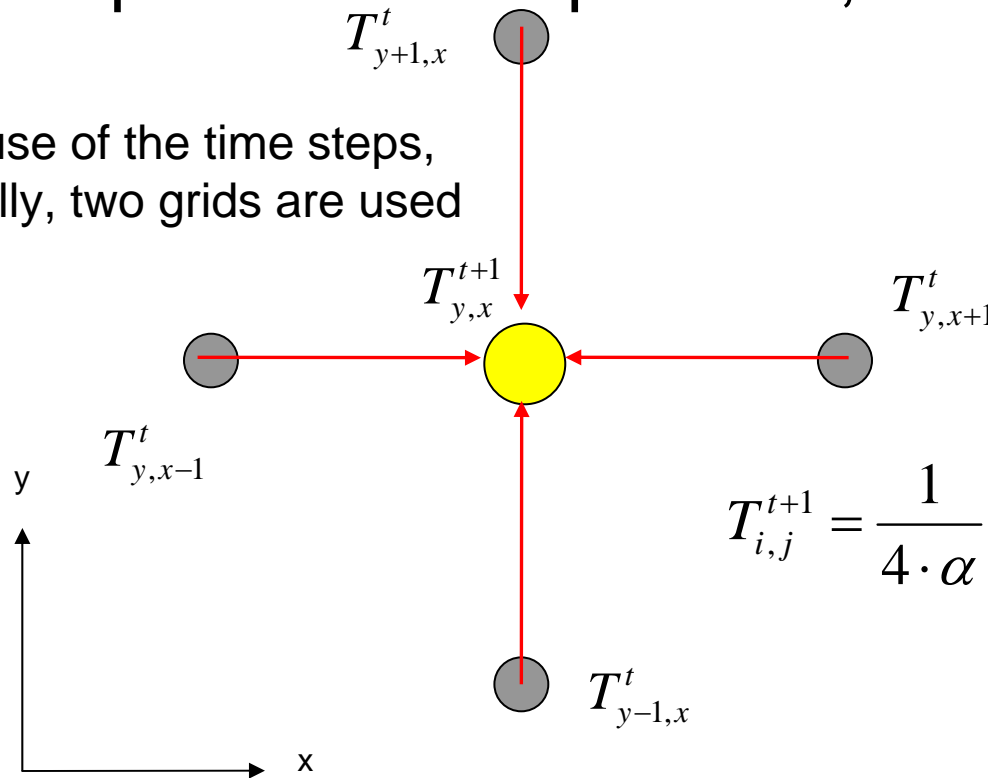


2D Heat Conduction Problem

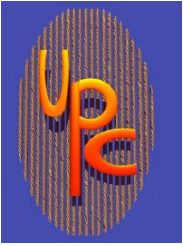
- Based on the 2D Partial Differential Equation (1), 2D Heat Conduction problem is similar to a 4-point stencil operation, as seen in (2):

Because of the time steps,
Typically, two grids are used

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{\alpha} \frac{\partial T}{\partial t} \quad (1)$$



$$T_{i,j}^{t+1} = \frac{1}{4 \cdot \alpha} (T_{i-1,j}^t + T_{i+1,j}^t + T_{i,j-1}^t + T_{i,j+1}^t) \quad (2)$$



2D Heat Conduction Problem

```
shared [BLOCKSIZE] double grids[2][N][N];
shared double dTmax_local[THREADS];
int nr_iter,i,x,y,z,dg,sg,finished;
double dTmax, dT, T;
do {
    dTmax = 0.0;

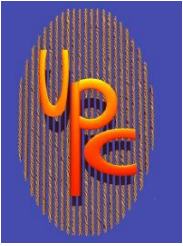
    for( y=1; y<N-1; y++ )
    {
        upc_forall( x=1; x<N-1; x++; &grids[sg][y][x] )
        {
            T = (grids[sg][y-1][x] + grids[sg][y+1][x] +
                grids[sg][z][y][x-1] + grids[sg][z][y][x+1])
                / 4.0;
            dT = T - grids[sg][y][x];
            grids[dg][y][x] = T;

            if( dTmax < fabs(dT) )
                dTmax = fabs(dT);
        }
    }
}
```

Work distribution, according to the defined BLOCKSIZE of grids[][][]
HERE, generic expression, working for any BLOCKSIZE

4-point
Stencil



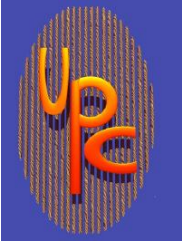


```
dTmax_local[MYTHREAD] =  
dTmax;  
upc_barrier;  
  
dTmax = dTmax_local[0];  
for( i=1; i<THREADS; i++ )  
    if( dTmax < dTmax_local[i])  
        dTmax = dTmax_local[i];  
upc_barrier;
```

Reduction
operation

```
if( dTmax < epsilon )  
    finished = 1;  
else  
{  
    // swapping the source  
    & destination  
    "pointers"  
    dg = sg;  
    sg = !sg;  
}  
nr_iter++;  
} while( !finished );  
upc_barrier;
```

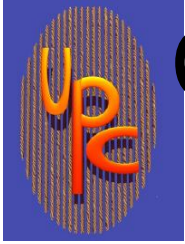




Dynamic Memory Allocation in UPC

- 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”





Collective Global Memory Allocation

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

nblocks: number of blocks

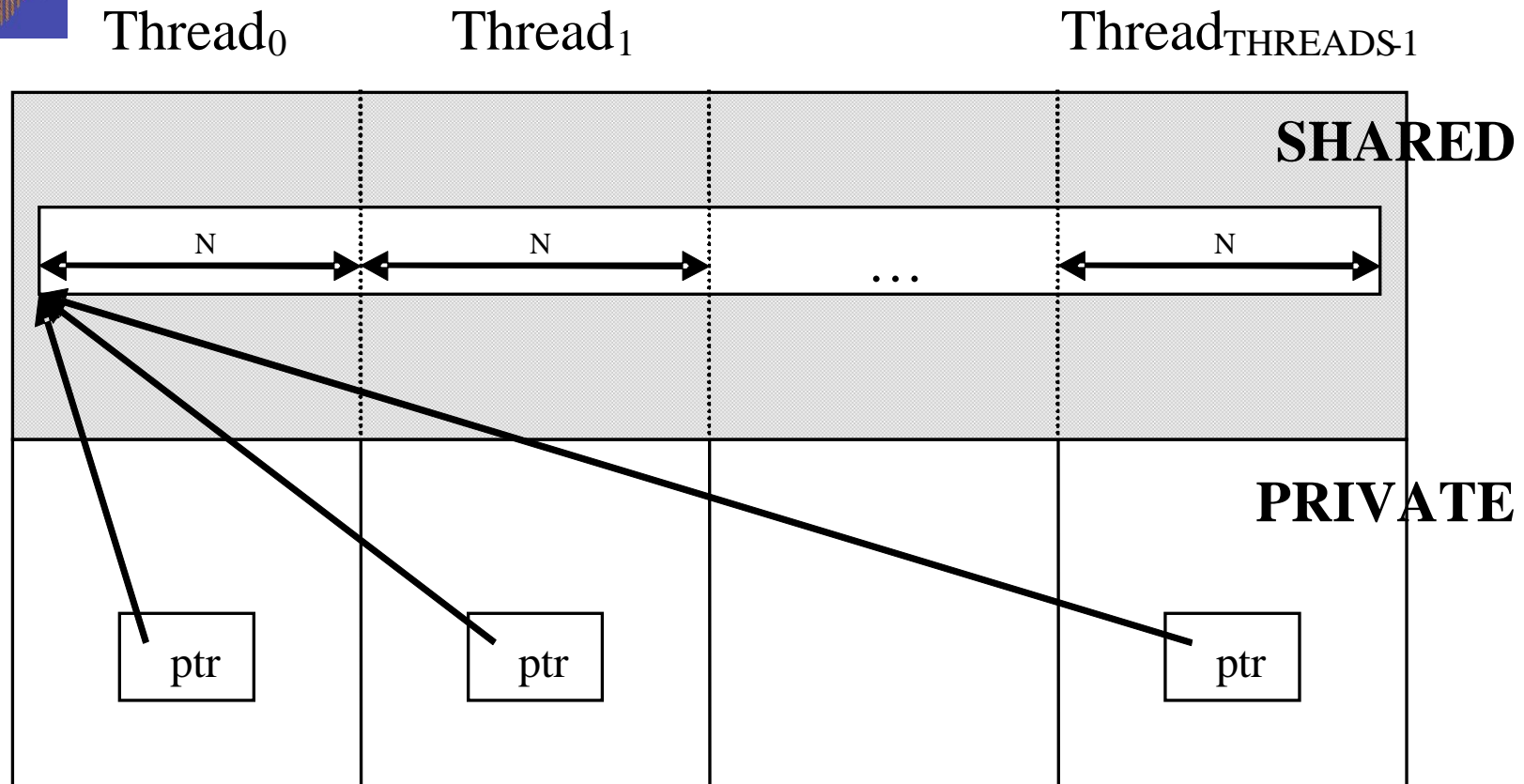
nbytes: block size

- 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
- All the threads will get the same pointer
- Equivalent to :
shared [nbytes] char[nblocks * nbytes]



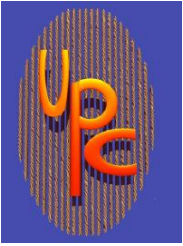


Collective Global Memory Allocation



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



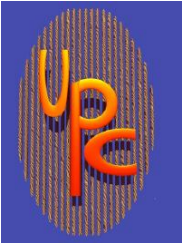


2D Heat Conduction Example

```
#define CHECK_MEM(var){\  
    if( var == NULL )\  
    {\  
        printf("TH%02d: ERROR: %s ==\  
        NULL\n",\  
            MYTHREAD, #var ); \  
        upc_global_exit(1); \  
    } }  
  
shared [BLOCKSIZE] double  
    *sh_grids;  
  
void heat_conduction(shared  
    [BLOCKSIZE] double  
    (*grids)[N][N])  
{
```

```
    for( y=1; y<N-1; y++ )  
    {  
        upc_forall( x=1; x<N-1;  
            x++; &grids[sg][y][x] )  
        {  
            T = (grids[sg][y+1][x] + ...  
                ...  
            } while( finished == 0 );  
        return nr_iter;  
    }  
}
```



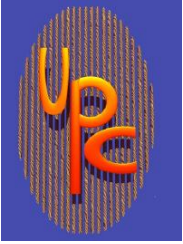


```
int main(void)
{
    int nr_iter;

    /* allocate the memory required for grids[2][N][N] */
    sh_grids = (shared [BLOCKSIZE] double *)
        upc_all_alloc( 2*N*N/BLOCKSIZE,
            BLOCKSIZE*sizeof(double));
    CHECK_MEM(sh_grids);
    ...
    /* performs the heat conduction computations */
    nr_iter = heat_conduction(
        (shared [BLOCKSIZE] double (*)[N][N])
            sh_grids);
    ... }
}
```

**Casting here to a 2-D
shared pointer!**





Global Memory Allocation

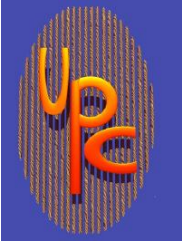
```
shared void *upc_global_alloc  
    (size_t nblocks, size_t nbytes);
```

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



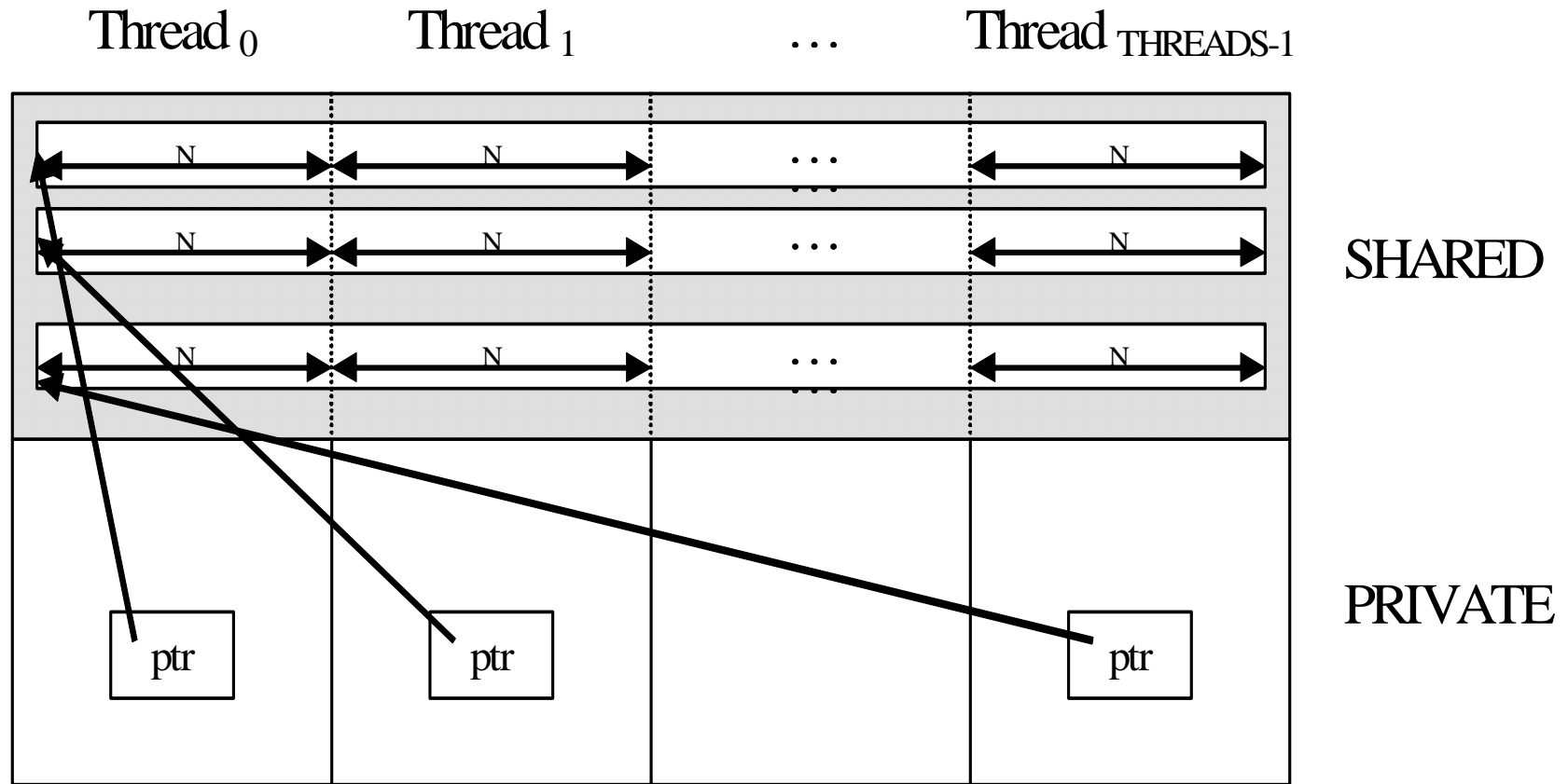
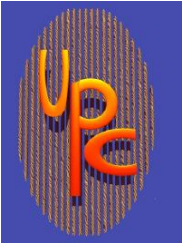


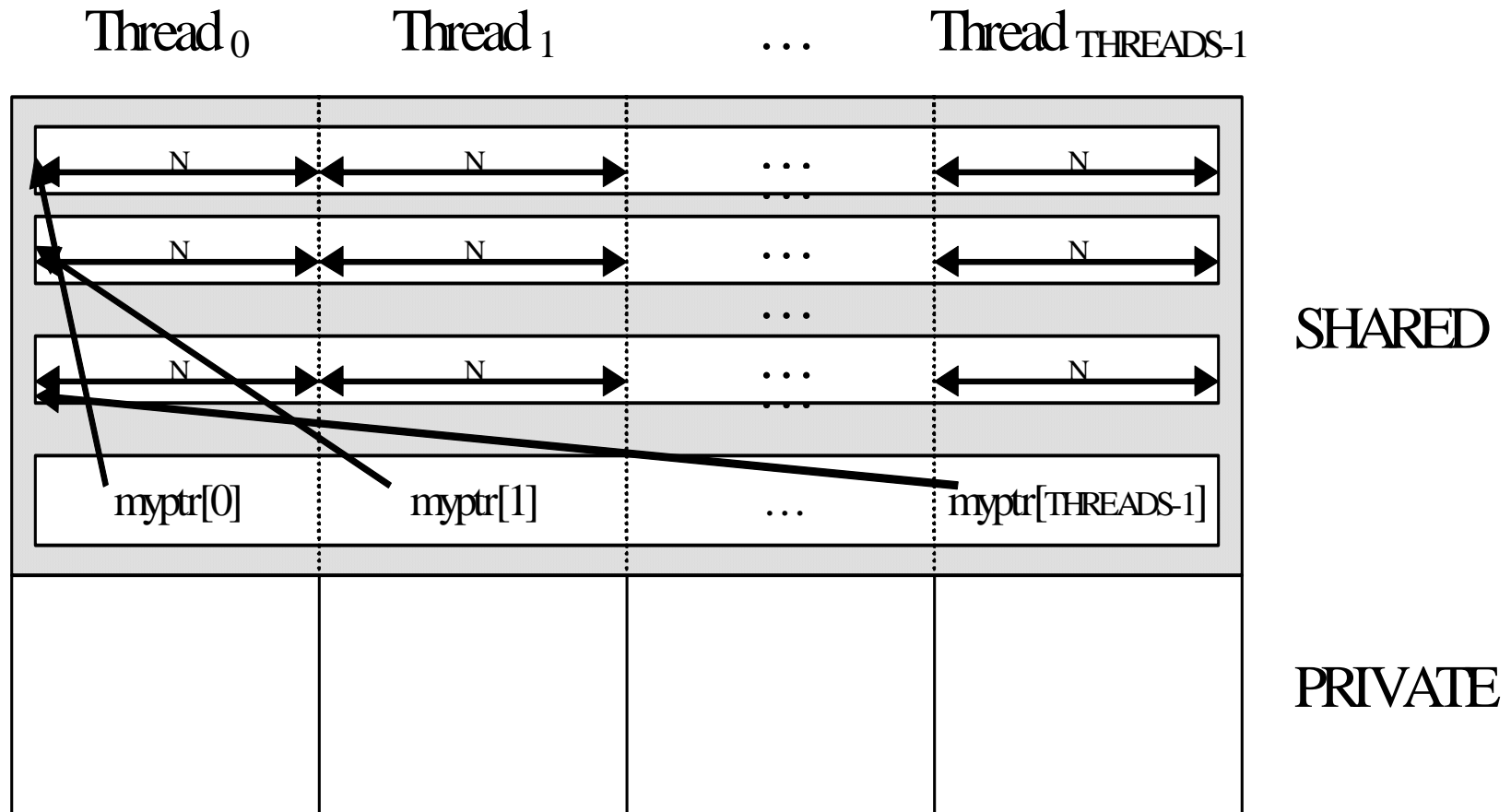
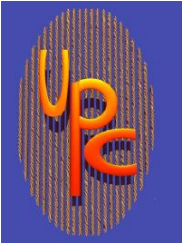
Global Memory Allocation

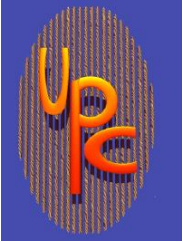
```
shared [N] int *ptr;  
  
ptr =  
    (shared [N] int *)  
    upc_global_alloc( THREADS, N*sizeof( int ));
```

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









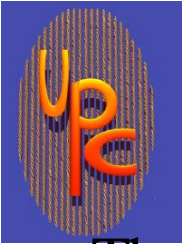
Local-Shared Memory Allocation

shared void *upc_alloc (size_t nbytes);

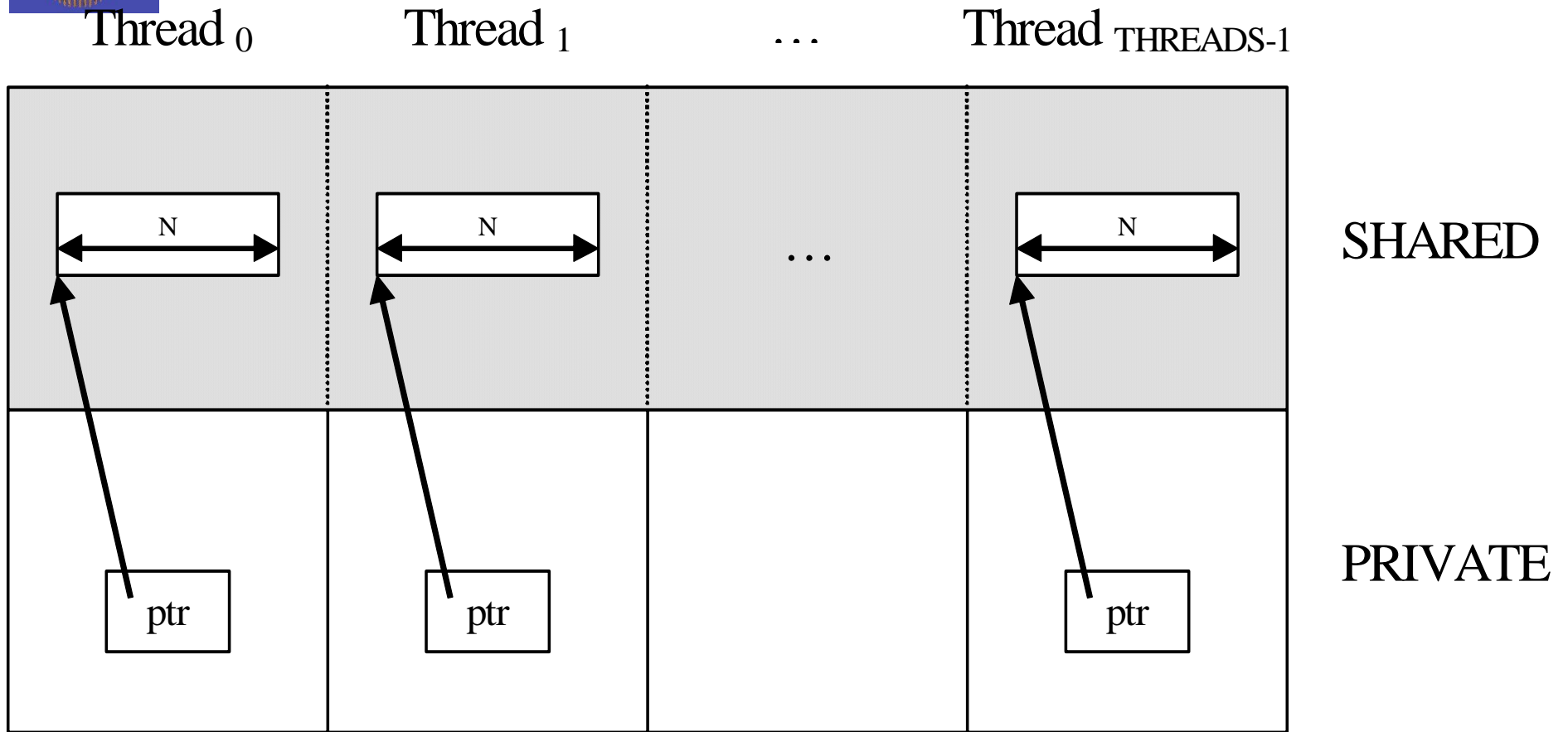
nbytes: block size

- 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



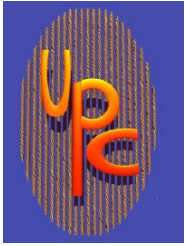


Local-Shared Memory Allocation



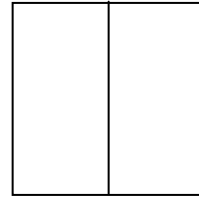
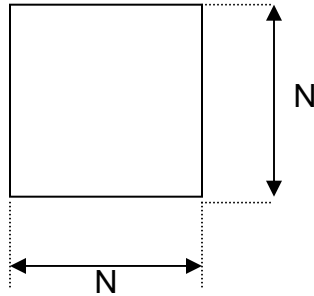
```
shared [] int *ptr;  
ptr = (shared [] int *)upc_alloc(N*sizeof( int ));
```





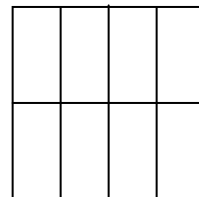
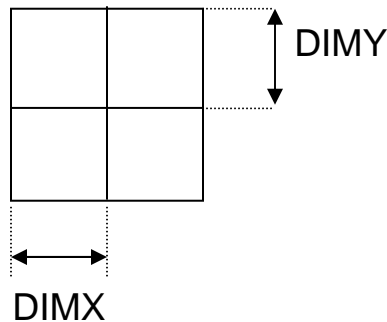
Blocking Multidimensional Data by Cells

THREADS=1
NO_COLS=1
NO_ROWS=1

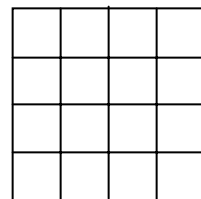


THREADS=2
NO_COLS=2
NO_ROWS=1

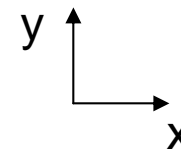
THREADS=4
NO_COLS=2
NO_ROWS=2



THREADS=8
NO_COLS=4
NO_ROWS=2

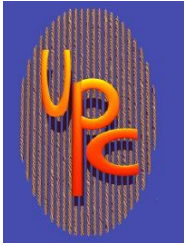


THREADS=16
NO_COLS=4
NO_ROWS=4



- Blocking can also be done by 2D “cells”, of equal size across **THREADS**
- Works best with N being a power of 2





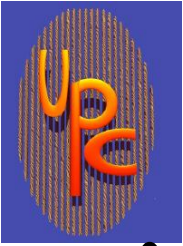
Blocking Multidimensional Data by Cells

- Determining DIMX and DIMY:

```
NO_COLS = NO_ROWS = 1;
for( i=2, j=0; i<=THREADS;
    i<<=1, j++ ) {
    if( (j%3)==0 )
        NO_COLS <<= 1;
    else if((j%3)==1)
        NO_ROWS <<= 1;
}
DIMX = N / NO_COLS;
DIMY = N / NO_ROWS;
```

i	j	NO_COLS	NO_ROWS
1	0	1	1
2	0	2	1
4	1	2	2
8	2	4	2
16	3	4	4





- Accessing one element of those 3D shared cells (by a macro):

```
#define CELL_SIZE DIMY*DIMX
struct gridcell_s {
    double cell[CELL_SIZE];
};
```

Definition of a cell

```
typedef struct gridcell_s gridcell_t;
```

2* One cell per thread

```
shared gridcell_t cell_grids[2][THREADS];
```

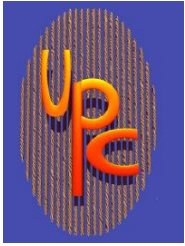
```
#define grids(gridno, y, x) \
cell_grids[gridno][((y)/DIMY)*NO_COLS + ((x)/DIMX)].cell[ \
((y)%DIMY)*DIMX + ((x)%DIMX)]
```

Linearization – 2D into a 1+1D
(using a C Macro)

Which THREAD?

Which Offset in the cell?





2D Heat Conduction Example w 2D-Cells

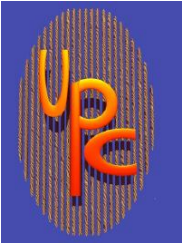
```
typedef struct chunk_s chunk_t;
struct chunk_s {
    shared [] double *chunk;
};

int N;
shared chunk_t sh_grids[2][THREADS];
shared double dTmax_local[THREADS];

#define grids(no,y,x) sh_grids[no]
    [ ((y)*N+(x))/(N*N*N/THREADS) ].chunk
    [ ((y)*N+(x))%(N*N*N/THREADS) ]

int heat_conduction(shared chunk_t (*sh_grids)[THREADS])
{
    // grids[][][] has to be changed to grids(,,,)
}
```

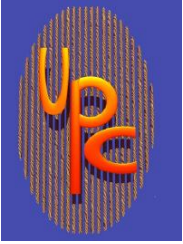




```
int main(int argc, char **argv)
{
    int nr_iter, no;
    // get N as parameter
    ...

    for( no=0; no<2; no++ ) /* allocate */
    {
        sh_grids[no][MYTHREAD].chunk =
            (shared [] double *) upc_alloc
            (N*N/THREADS*sizeof( double ));
        CHECK_MEM( sh_grids[no][MYTHREAD].chunk );
    }
    ...
    /* performs the heat conduction computation */
    nr_iter = heat_conduction(sh_grids);
    ...
}
```



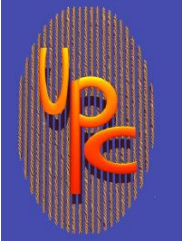


Memory Space Clean-up

```
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



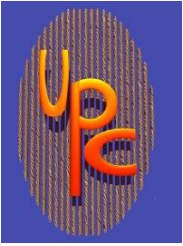


Example: Matrix Multiplication in UPC

- Given two integer matrices $A(N \times P)$ and $B(P \times M)$, we want to compute $C = A \times B$.
- Entries c_{ij} in C are computed by the formula:

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





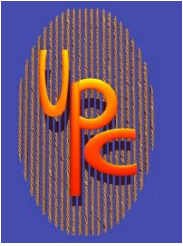
Doing it in C

```
#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};

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];
        }
    }
}
```

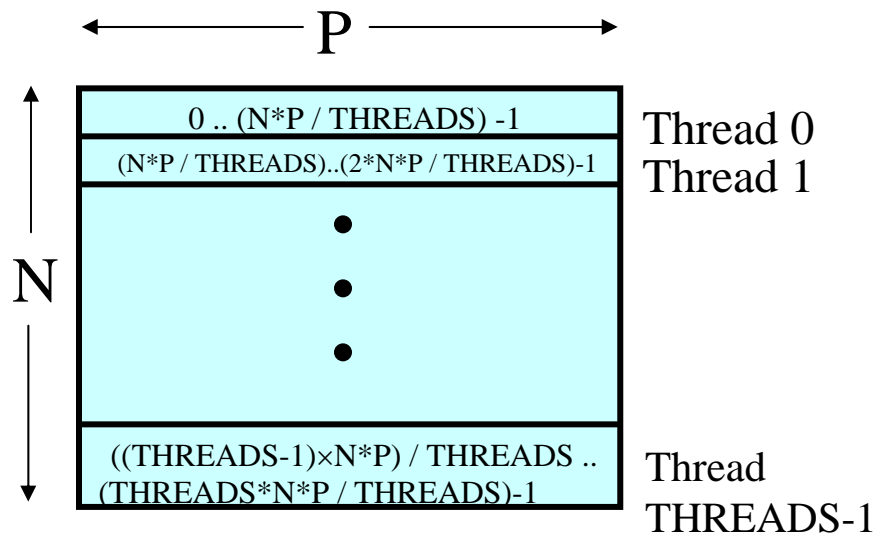




Domain Decomposition for UPC

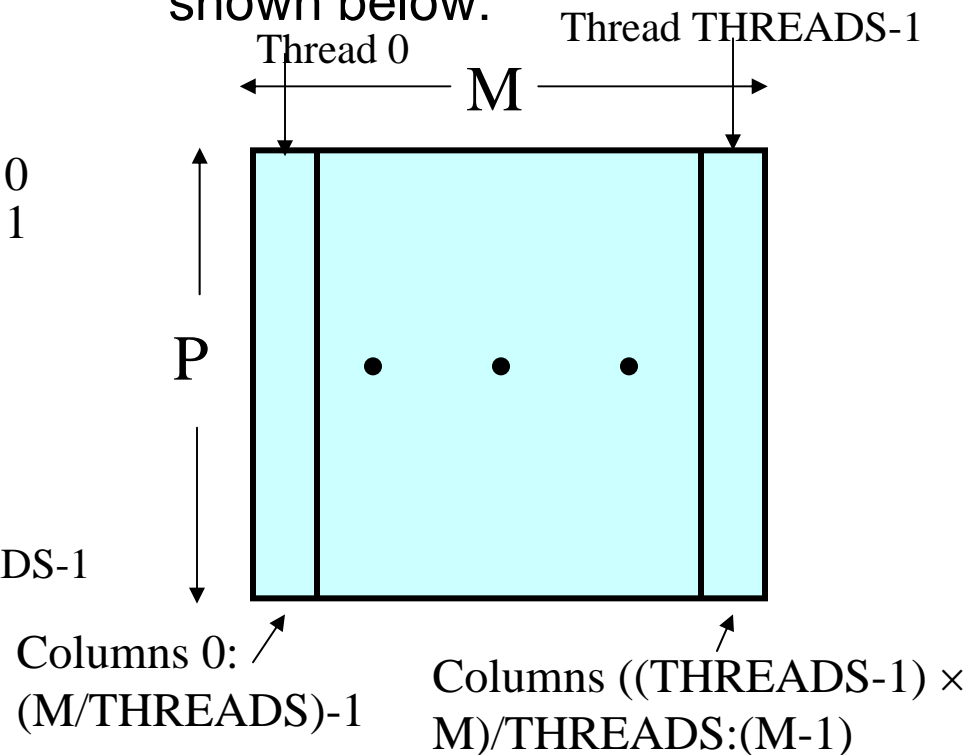
Exploiting locality in matrix multiplication

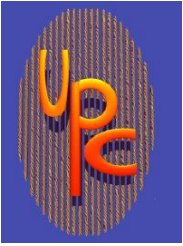
- A ($N \times P$) is decomposed row-wise into blocks of size $(N \times P) / \text{THREADS}$ as shown below:



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

- B ($P \times M$) is decomposed column-wise into $M / \text{THREADS}$ blocks as shown below:





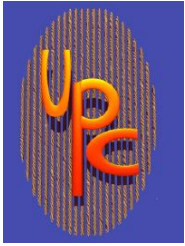
UPC Matrix Multiplication Code

```
#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];
        }
    }
}
```



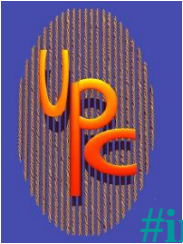


UPC Matrix Multiplication Code with Privatization

```
#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];
        }
    }
}
```



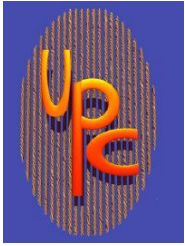


UPC Matrix Multiplication Code with block copy

```
#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];
        }
    }
}
```

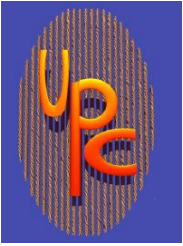




UPC Matrix Multiplication Code with Privatization and Block Copy

```
#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

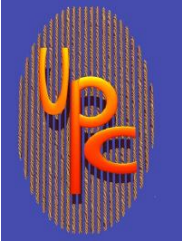
```
#include <upc_relaxed.h>
shared [N*P /THREADS] int *a;
shared [N*M /THREADS] int *c;
shared [M/THREADS] int *b;

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];
        }
    }
}
```

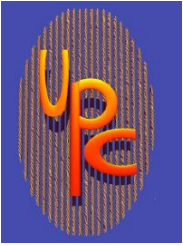




Synchronization

- No implicit synchronization among the threads
- UPC provides the following synchronization mechanisms:
 - Barriers
 - Locks

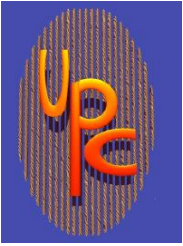




Synchronization - Barriers

- No implicit synchronization among the threads
 - UPC provides the following barrier synchronization constructs:
 - Barriers (Blocking)
 - `upc_barrier expropt;`
 - Split-Phase Barriers (Non-blocking)
 - `upc_notify expropt;`
 - `upc_wait expropt;`
- Note: `upc_notify` is not blocking `upc_wait` is

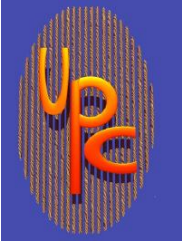




Synchronization - Locks

- In UPC, shared data can be protected against multiple writers :
 - `void upc_lock(upc_lock_t *l)`
 - `int upc_lock_attempt(upc_lock_t *l) //returns 1 on success and 0 on failure`
 - `void upc_unlock(upc_lock_t *l)`
- Locks are allocated dynamically, and can be freed
- Locks are properly initialized after they are allocated

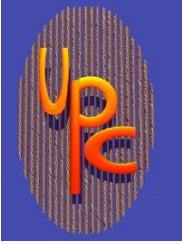




Dynamic lock allocation

- The locks can be managed using the following functions:
- **collective lock allocation (à la `upc_all_alloc`)**
`upc_lock_t * upc_all_lock_alloc(void);`
- **global lock allocation (à la `upc_global_alloc`)**
`upc_lock_t * upc_global_lock_alloc(void)`
- **lock freeing**
`void upc_lock_free(upc_lock_t *ptr);`





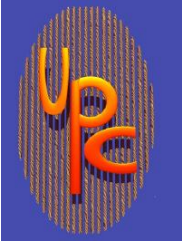
Collective lock allocation

- **collective lock allocation**

```
upc_lock_t * upc_all_lock_alloc(void);
```

- Needs to be called by all the threads
- Returns a single lock to all calling threads





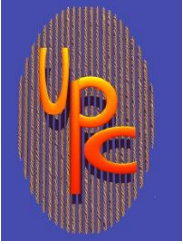
Global lock allocation

- **global lock allocation**

```
upc_lock_t * upc_global_lock_alloc(void)
```

- Returns one lock pointer to the calling thread
- This is not a collective function





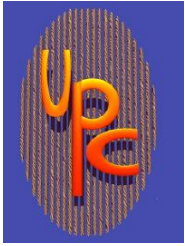
Lock freeing

- **Lock freeing**

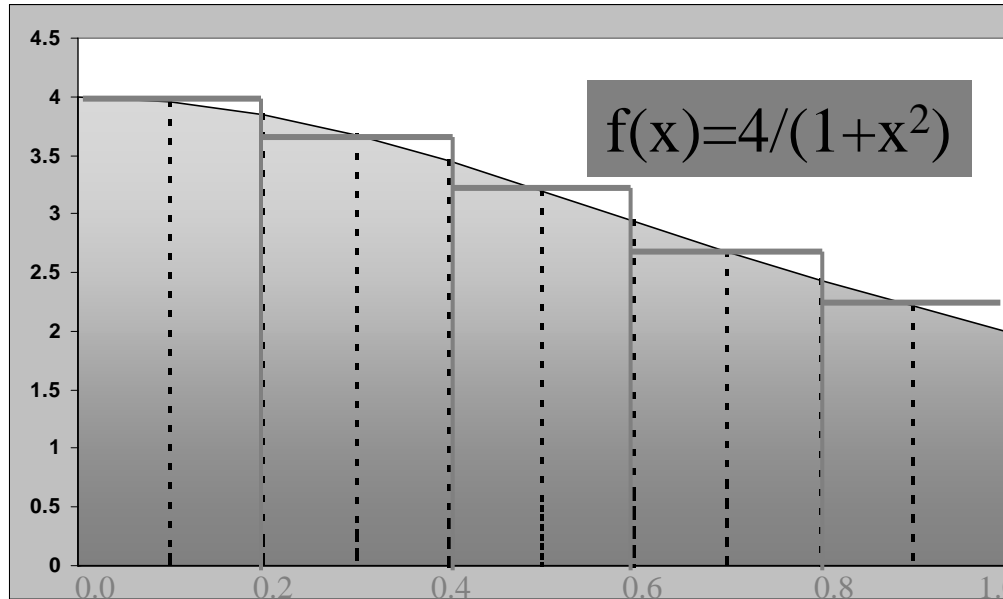
```
void upc_lock_free(upc_lock_t *l);
```

- This is not a collective function



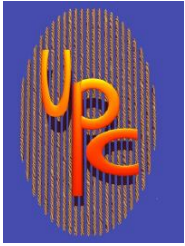


Numerical Integration (computation of π)



$$4 \int_0^1 \frac{1}{1+x^2}$$

- Integrate the function f (which equals π)



Example: Using Locks in Numerical Integration

```
// Example – The Famous PI  
- Numerical Integration
```

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

```
#define N 1000000
```

```
#define f(x) 1/(1+x*x)
```

```
upc_lock_t *l;
```

```
shared float pi;
```

```
void main(void)
```

```
{
```

```
    float local_pi=0.0;
```

```
    int i;
```

```
    l = upc_all_lock_alloc();
```

```
    upc_barrier;
```

```
    upc_forall(i=0;i<N;i++; i)  
        local_pi +=(float) f((.5+i)/(N));
```

```
    local_pi *= (float) (4.0 / N);
```

```
    upc_lock(l);    /*better with collectives*/
```

```
    pi += local_pi;
```

```
    upc_unlock(l);
```

```
    upc_barrier(); // Ensure all is done
```

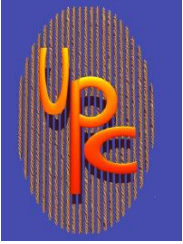
```
    upc_lock_free( l );
```

```
    if(MYTHREAD==0)
```

```
        printf("PI=%f\n",pi);
```

```
}
```

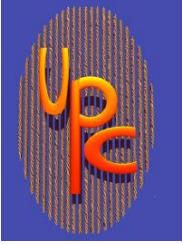




Memory Consistency Models

- Has to do with ordering of shared operations, and when a change of a shared object by a thread becomes visible to others
- Consistency can be *strict* or *relaxed*
- Under the relaxed consistency model, the shared operations can be reordered by the compiler / runtime system
- The strict consistency model enforces sequential ordering of shared operations. (No operation on shared can begin before the previous ones are done, and changes become visible immediately)

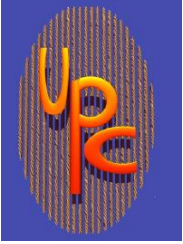




Memory Consistency

- Default behavior can be controlled by the programmer and set at the program level:
 - To have strict memory consistency
`#include <upc_strict.h>`
 - To have relaxed memory consistency
`#include <upc_relaxed.h>`

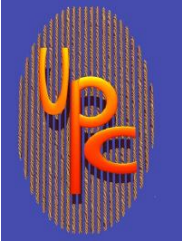




Memory Consistency

- Default behavior can be altered for a variable definition in the declaration using:
 - Type qualifiers: *strict* & *relaxed*
- Default behavior can be altered for a statement or a block of statements using
 - `#pragma upc strict`
 - `#pragma upc relaxed`
- Highest precedence is at declarations, then pragmas, then program level

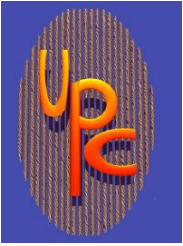




Memory Consistency- Fence

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





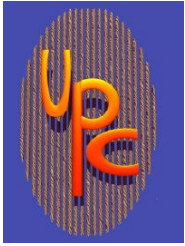
Memory Consistency Example

```
strict shared int flag_ready = 0;  
shared int result0, result1;
```

```
if (MYTHREAD==0)  
    { results0 = expression1;  
      flag_ready=1; //if not strict, it could be  
      // switched with the above statement    }  
else if (MYTHREAD==1)  
    { while(!flag_ready); //Same note  
      result1=expression2+results0;    }
```

- We could have used a barrier between the first and second statement in the if and the else code blocks. Expensive!! Affects all operations at all threads.
- We could have used a fence in the same places. Affects shared references at all threads!
- The above works as an example of point to point synchronization.



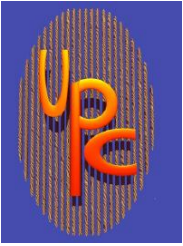


Section 2: UPC Systems

Merkey & Seidel

- Summary of current UPC systems
 - Cray X-1
 - Hewlett-Packard
 - Berkeley
 - Intrepid
 - MTU
- UPC application development tools
 - totalview
 - upc_trace
 - work in progress
 - performance toolkit interface
 - performance model

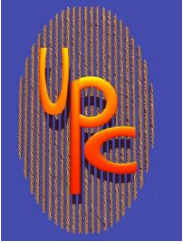




Cray UPC

- Platform: Cray X1 supporting UPC v1.1.1
- Features: shared memory architecture
 - UPC is compiler option => all of the ILP optimization is available in UPC.
 - The processors are designed with 4 SSP's per MSP.
 - A UPC thread can run on a SSP or a MSP, a SSP-mode vs. MSP-mode performance analysis is required before making a choice.
 - There are no virtual processors.
 - This is a high-bandwidth, low latency system.
 - The SSP's are vector processors, the key to performance is exploiting ILP through vectorization.
 - The MSP's run at a higher clock speed, the key to performance is having enough independent work to be multi-streamed.

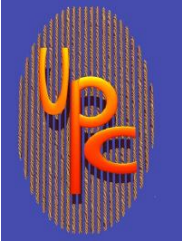




Cray UPC

- Usage
 - Compiling for arbitrary numbers of threads:
`cc -hupc filename.c` (MSP mode, one thread per MSP)
`cc -hupc,ssp filename.c` (SSP mode, one thread per SSP)
 - Running
`aprun -n THREADS ./a.out`
 - Compiling for fixed number of threads
`cc -hssp,upc -X THREADS filename.c -o a.out`
 - Running
`./a.out`
- URL:
 - <http://docs.cray.com>
 - Search for “UPC” under Cray X1

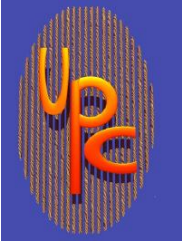




Hewlett-Packard UPC

- Platforms: Alphaserver SC, HP-UX IPF, PA-RISC, HP XC ProLiant DL360 or 380.
- Features:
 - UPC version 1.2 compliant
 - UPC-specific performance optimization
 - Write-through software cache for remote accesses
 - Cache configurable at run time
 - Takes advantage of same-node shared memory when running on SMP clusters
 - Rich diagnostic and error-checking facilities

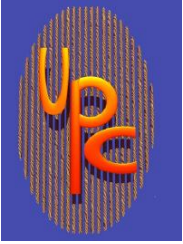




Hewlett-Packard UPC

- Usage
 - Compiling for arbitrary number of threads:
`upc filename.c`
 - Compiling for fixed number of threads:
`upc -fthreads THREADS filename.c`
 - Running:
`prun -n THREADS ./a.out`
- URL: <http://h30097.www3.hp.com/upc>

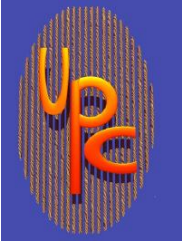




Berkeley UPC (BUPC)

- Platforms: Supports a wide range of architectures, interconnects and operating systems
- Features
 - Open64 open source compiler as front end
 - Lightweight runtime and networking layers built on GASNet
 - Full UPC version 1.2 compliant, including UPC collectives and a reference implementation of UPC parallel I/O
 - Can be debugged by Totalview
 - Trace analysis: `upc_trace`

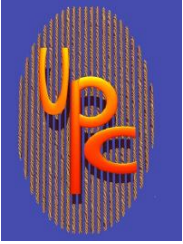




Berkeley UPC (BUPC)

- Usage
 - Compiling for arbitrary number of threads:
`upcc filename.c`
 - Compiling for fixed number of threads:
`upcc -T=threads THREADS filename.c`
 - Compiling with optimization enabled (experimental)
`upcc -opt filename.c`
 - Running:
`upcrun -n THREADS ./a.out`
- URL: <http://upc.nersc.gov>

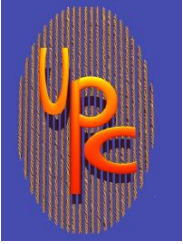




Intrepid GCC/UPC

- Platforms: shared memory platforms only
 - Itanium, AMD64, Intel x86 uniprocessor and SMP's
 - SGI IRIX
 - Cray T3E
- Features
 - Based on GNU GCC compiler
 - UPC version 1.1 compliant
 - Can be a front-end of the Berkeley UPC runtime

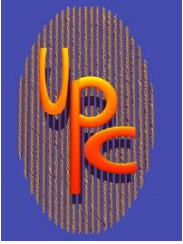




Intrepid GCC/UPC

- Usage
 - Compiling for arbitrary number of threads:
`upc -x upc filename.c`
 - Running
`mpprun ./a.out`
 - Compiling for fixed number of threads:
`upc -x upc -fupc-threads-THREADS filename.c`
 - Running
`./a.out`
- URL: <http://www.intrepid.com/upc>

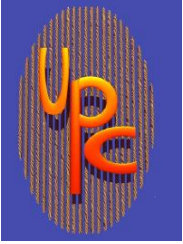




MTU UPC (MuPC)

- Platforms: Intel x86 Linux clusters and AlphaServer SC clusters with MPI-1.1 and Pthreads
- Features:
 - EDG front end source-to-source translator
 - UPC version 1.1 compliant
 - Generates 2 Pthreads for each UPC thread
 - user code
 - MPI-1 Pthread handles remote accesses
 - Write-back software cache for remote accesses
 - Cache configurable at run time
 - Reference implementation of UPC collectives

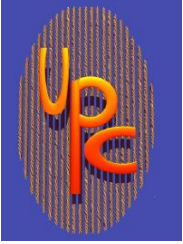




MTU UPC (MuPC)

- Usage:
 - Compiling for arbitrary number of threads:
`mupcc filename.c`
 - Compiling for fixed number of threads:
`mupcc -f THREADS filename.c`
 - Running:
`mupcrun -n THREADS ./a.out`
- URL: <http://www.upc.mtu.edu>

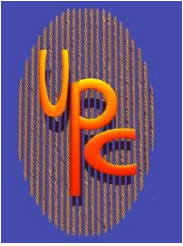




UPC Tools

- Etnus Totalview
- Berkeley UPC trace tool
- U. of Florida performance tool interface
- MTU performance modeling project

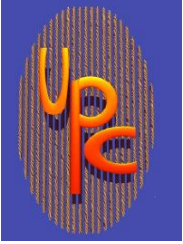




Totalview

- Platforms:
 - HP UPC on Alphaservers
 - Berkeley UPC on x86 architectures with MPICH or Quadrics' elan as network.
 - Must be Totalview version 7.0.1 or above
 - BUPC runtime must be configured with `--enable-trace`
 - BUPC back end must be GNU GCC
- Features
 - UPC-level source examination, steps through UPC code
 - Examines shared variable values at run time





Totalview

- Usage

- Compiling for totalview debugging:

```
upcc -tv filename.c
```

- Running when MPICH is used:

```
mpirun -tv -np THREADS ./a.out
```

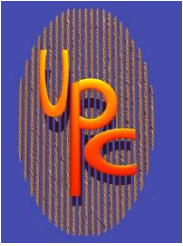
- Running when Quadrics' elan is used:

```
totalview prun -a -n THREADS ./a.out
```

- URL

- <http://upc.lbl.gov/docs/user/totalview.html>
- <http://www.etnus.com/TotalView/>

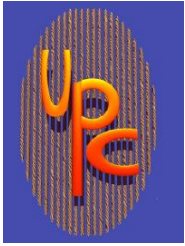




UPC trace

- `upc_trace` analyzes the communication behavior of UPC programs.
- A tool available for Berkeley UPC
- Usage
 - `upcc` must be configured with `--enable-trace`.
 - Run your application with
`upcrun -trace ...` or
`upcrun -tracefile TRACE_FILE_NAME ...`
 - Run `upc_trace` on trace files to retrieve statistics of runtime communication events.
 - Finer tracing control by manually instrumenting programs:
`bupc_trace_setmask()`, `bupc_trace_getmask()`,
`bupc_trace_gettracelocal()`,
`bupc_trace_settracelocal()`, *etc.*

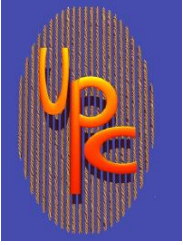




UPC trace

- **upc_trace** provides information on
 - Which lines of code generated network traffic
 - How many messages each line caused
 - The type (local and/or remote gets/puts) of messages
 - The maximum/minimum/average/combined sizes of the messages
 - Local shared memory accesses
 - Lock-related events, memory allocation events, and strict operations
- URL: http://upc.nersc.gov/docs/user/upc_trace.html

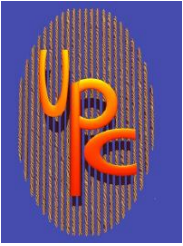




Performance tool interface

- A platform independent interface for toolkit developers
 - A callback mechanism notifies performance tool when certain events, such as remote accesses, occur at runtime
 - Relates runtime events to source code
 - Events: Initialization/completion, shared memory accesses, synchronization, work-sharing, library function calls, user-defined events
- Interface proposal is under development
- URL: <http://www.hcs.ufl.edu/~leko/upctoolint/>

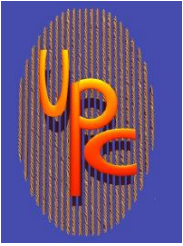




Performance model

- Application-level analytical performance model
- Models the performance of UPC fine-grain accesses through platform benchmarking and code analysis
- Platform abstraction:
 - Identify a common set of optimizations performed by a high performance UPC platform: aggregation, vectorization, pipelining, local shared access optimization, communication/computation overlapping
 - Design microbenchmarks to determine the platform's optimization potentials

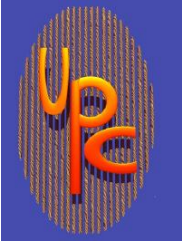




Performance model

- Code analysis
 - High performance achievable by exploiting concurrency in shared references
 - Reference partitioning:
 - A dependence-based analysis to determine concurrency in shared access scheduling
 - References are partitioned into groups, accesses of references in a group are subject to one type of envisioned optimization

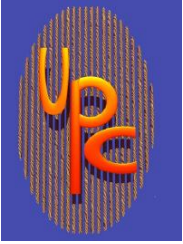
- Run time prediction:
$$T_{comm} = \sum_i^{RefGroups} \left\{ \frac{N_i}{r(N_i, type)} \right\}$$



Section 3: UPC Libraries

- Collective Functions
 - Bucket sort example
 - UPC collectives
 - Synchronization modes
 - Collectives performance
 - Extensions
- UPC-IO
 - Concepts
 - Main Library Calls
 - Library Overview

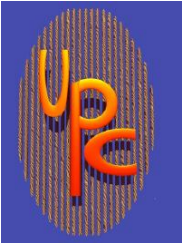




Collective functions

- A collective function performs an operation in which *all* threads participate.
- Recall that UPC includes the collectives:
 - `upc_barrier`, `upc_notify`, `upc_wait`,
`upc_all_alloc`, `upc_all_lock_alloc`
- Collectives covered here are for bulk data movement and computation.
 - `upc_all_broadcast`, `upc_all_exchange`,
`upc_all_prefix_reduce`, *etc.*





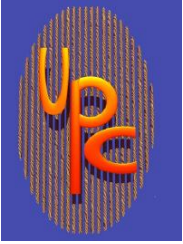
A quick example: Parallel bucketsort

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

Assume the keys in **A** are uniformly distributed.

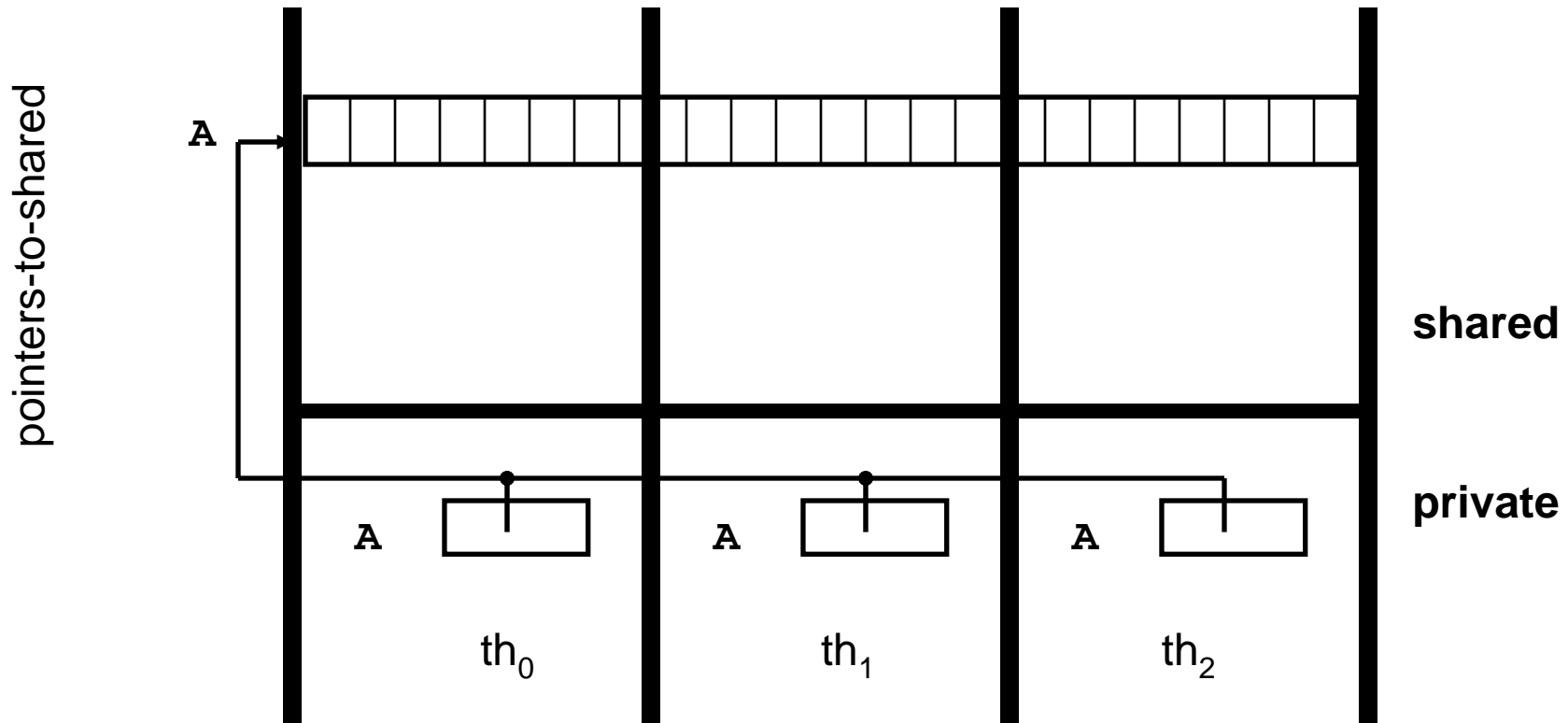
1. Find global min and max values in **A**.
2. Determine max bucket size.
3. Allocate bucket array and exchange array.
4. Bucketize **A** into local shared buckets.
5. Exchange buckets and merge.
6. Rebalance and return data to **A** if desired.

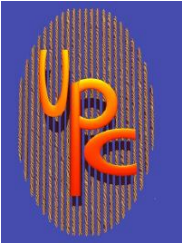




Sort shared array **A**

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





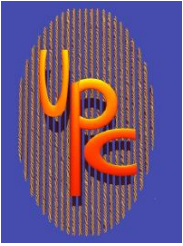
1. Find global min and max values

```
shared [] int minmax0[2];    // only on Thr 0
shared [2] int MinMax[2*THREADS];

// Thread 0 receives min and max values
upc_all_reduce(&minmax0[0],A,...,UPC_MIN,...);
upc_all_reduce(&minmax0[1],A,...,UPC_MAX,...);

// Thread 0 broadcasts min and max
upc_all_broadcast(MinMax,minmax0,
                 2*sizeof(int),NULL);
```

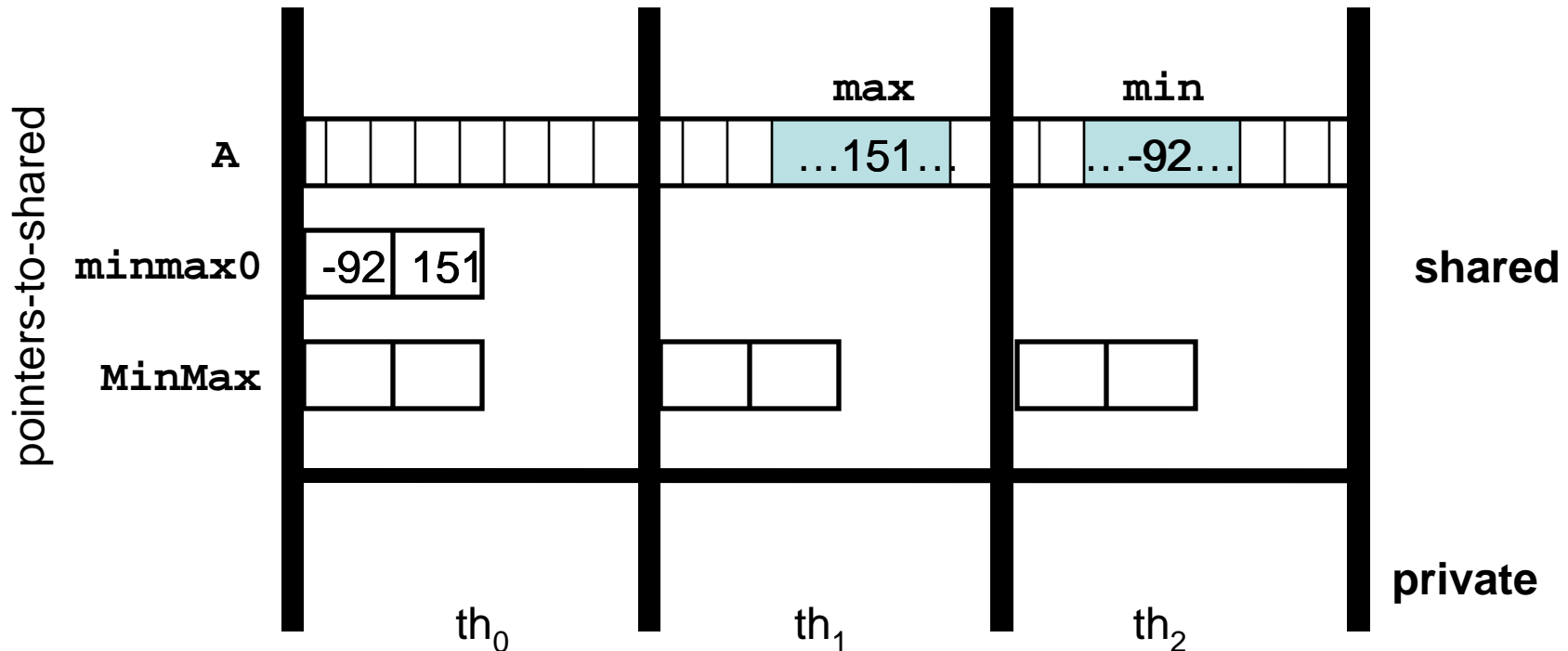


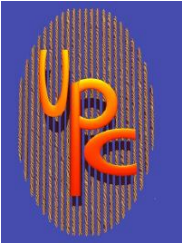


1. Find global min and max values

(animation)

```
shared [] int minmax0[2]; // only on Thread 0
shared [2] int MinMax[2*THREADS];
upc_all_reduce(&minmax[0],A,...,UPC_MIN,...);
upc_all_reduce(&minmax[1],A,...,UPC_MAX,...);
upc_all_broadcast(MinMax,minmax,2*sizeof(int),NULL);
```





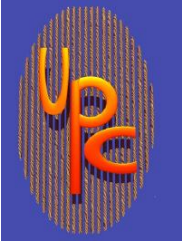
2. Determine max bucket size

```
shared [THREADS] int BSizes[THREADS][THREADS];
shared int bmax0;    // only on Thread 0
shared int Bmax[THREADS];

// determine splitting keys (not shown)
// initialize Bsize to 0, then
upc_forall(i=0; i<N*THREADS; i++; &A[i])
    if (A[i] will go in bucket j)
        ++BSizes[MYTHREAD][j];

upc_all_reduceI(&bmax0, BSizes, ..., UPC_MAX, ...);
upc_all_broadcast(Bmax, &bmax0, sizeof(int), ...);
```



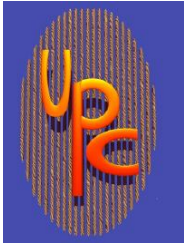


3. Allocate bucket and exchange arrays

```
shared int *BuckAry;  
shared int *BuckDst;  
int Blen;
```

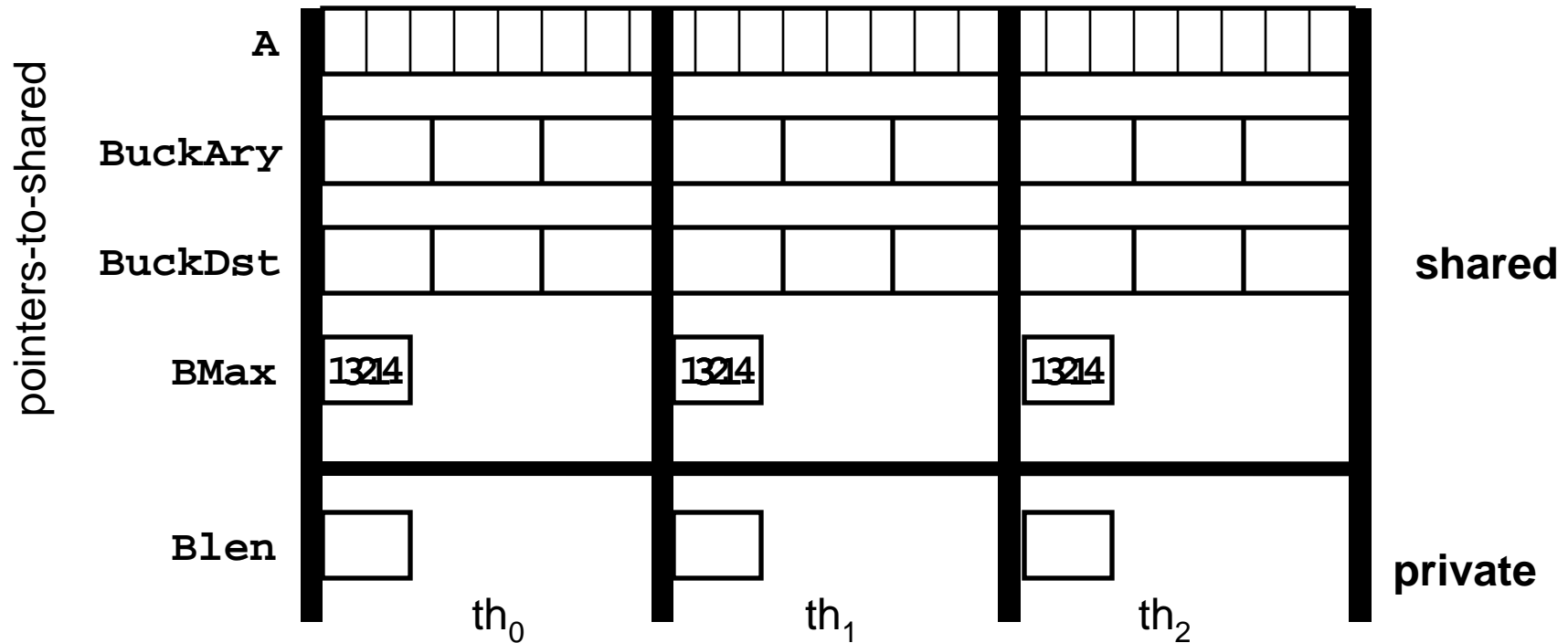
```
Blen = (int)Bmax[MYTHREAD]*sizeof(int);  
BuckAry = upc_all_alloc(Blen*THREADS,  
                        Blen*THREADS*THREADS);  
BuckDst = upc_all_alloc(Blen*THREADS,  
                        Blen*THREADS*THREADS);  
Blen = Blen/sizeof(int);
```

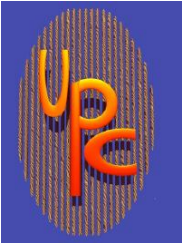




3. Allocate bucket and exchange arrays (animation)

```
int Blen;  
Blen = (int)Bmax[MYTHREAD]*sizeof(int);  
shared int *BuckAry, *BuckDst;  
BuckAry = upc_all_alloc(...);  
BuckDst = upc_all_alloc(...);
```





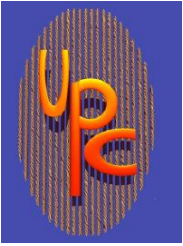
4. Bucketize A

```
int *Bptr;    // local ptr to BuckAry
shared [THREADS] int Bcnt[THREADS][THREADS];

// cast to local pointer
Bptr = (int *)&BuckAry[MYTHREAD];

// init bucket counters Bcnt[MYTHREAD][i]=0
upc_forall (i=0; i<N*THREADS; ++i; &A[i])
    if (A[i] belongs in bucket j)
    { Bptr[Blen*j+Bcnt[MYTHREAD][j]] = A[i];
      ++Bcnt[MYTHREAD][j];  }
```

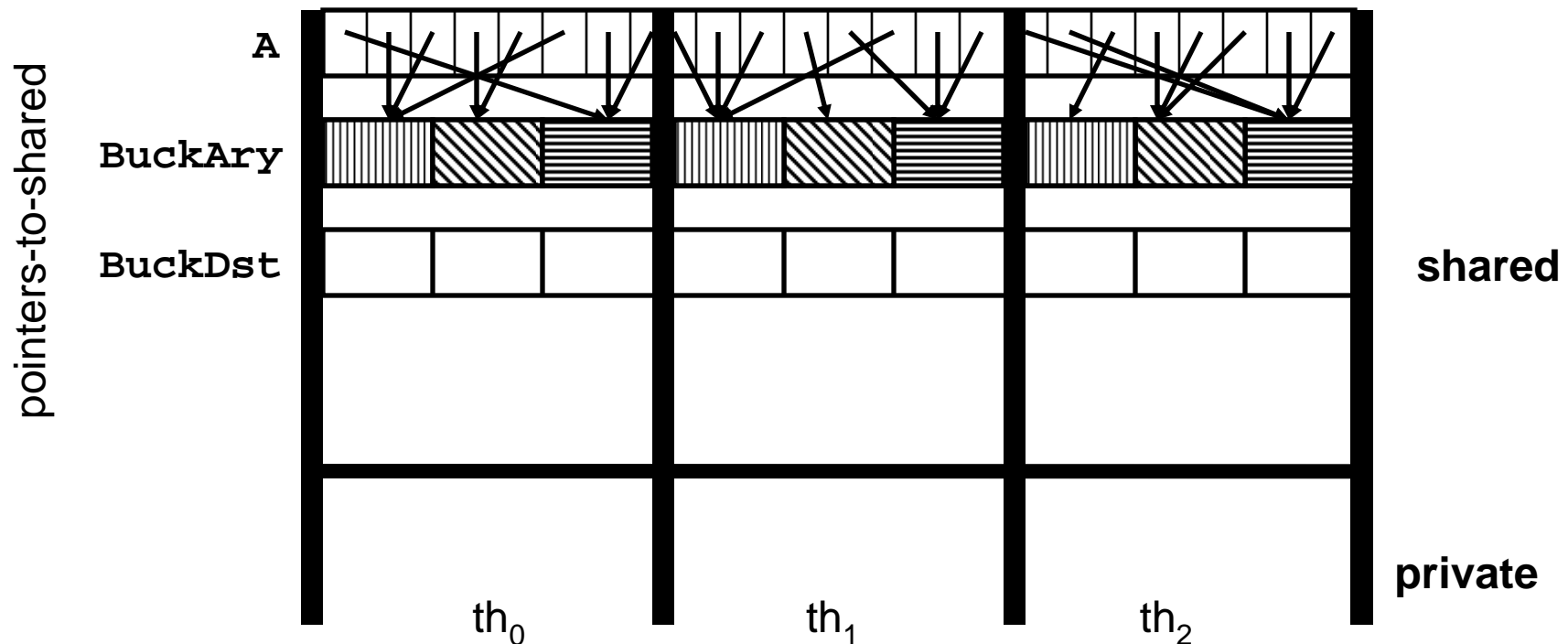


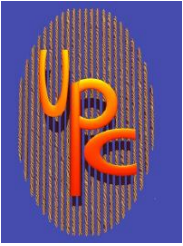


4. Bucketize A

(animation)

```
Bptr = (int *)&BuckAry[MYTHREAD];  
if (A[i] belongs in bucket j)  
{ Bptr[Blen*j+Bcnt[MYTHREAD][j]] = A[i];  
  ++Bcnt[MYTHREAD][j]; }
```

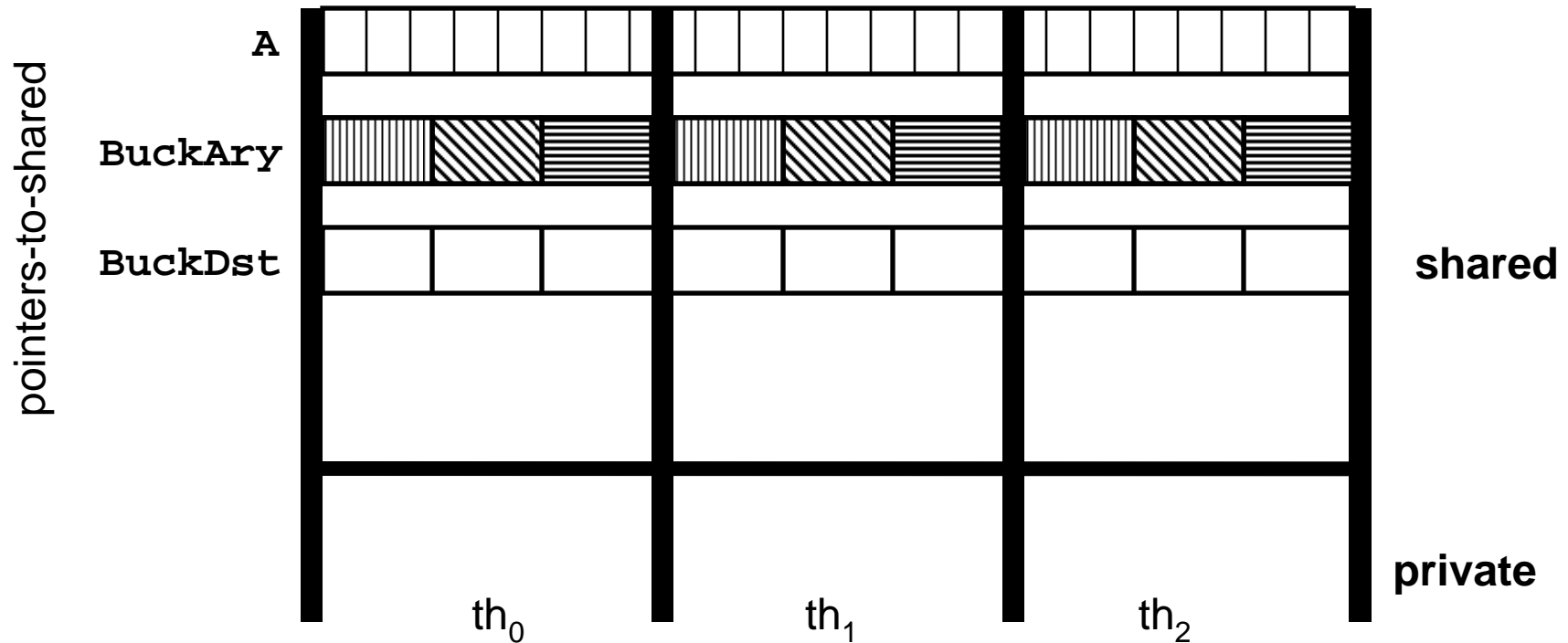


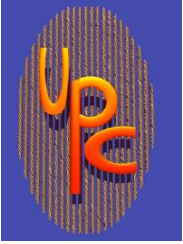


5. Exchange buckets

(animation)

```
upc_all_exchange(BuckDst, BuckAry, Blen*sizeof(int), NULL);
```

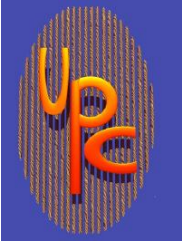




6. Merge and rebalance

```
Bptr = (int *)&BuckDst[MYTHREAD];  
  
// Each thread locally merges its part of  
// BuckDst.  Rebalance and return to A  
// if desired.  
  
if (MYTHREAD==0)  
{  upc_free(BuckAry);  
   upc_free(BuckDst); }
```

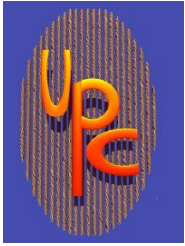




Collectives in UPC 1.2

- “Relocalization” collectives change data affinity.
 - `upc_all_broadcast`
 - `upc_all_scatter`
 - `upc_all_gather`
 - `upc_all_gather_all`
 - `upc_all_exchange`
 - `upc_all_permute`
- “Computational” collectives for data reduction
 - `upc_all_reduce`
 - `upc_all_prefix_reduce`

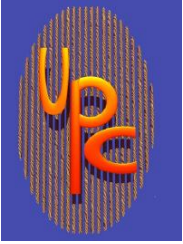




Why have collectives in UPC?

- Sometimes bulk data movement is the right thing to do.
- Built-in collectives offer better performance.
- Caution: UPC programs can come out looking like MPI code.

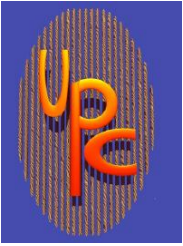




An animated tour of UPC collectives

- The following illustrations serve to define the UPC collective functions.
- High performance implementations of the collectives use more sophisticated algorithms.



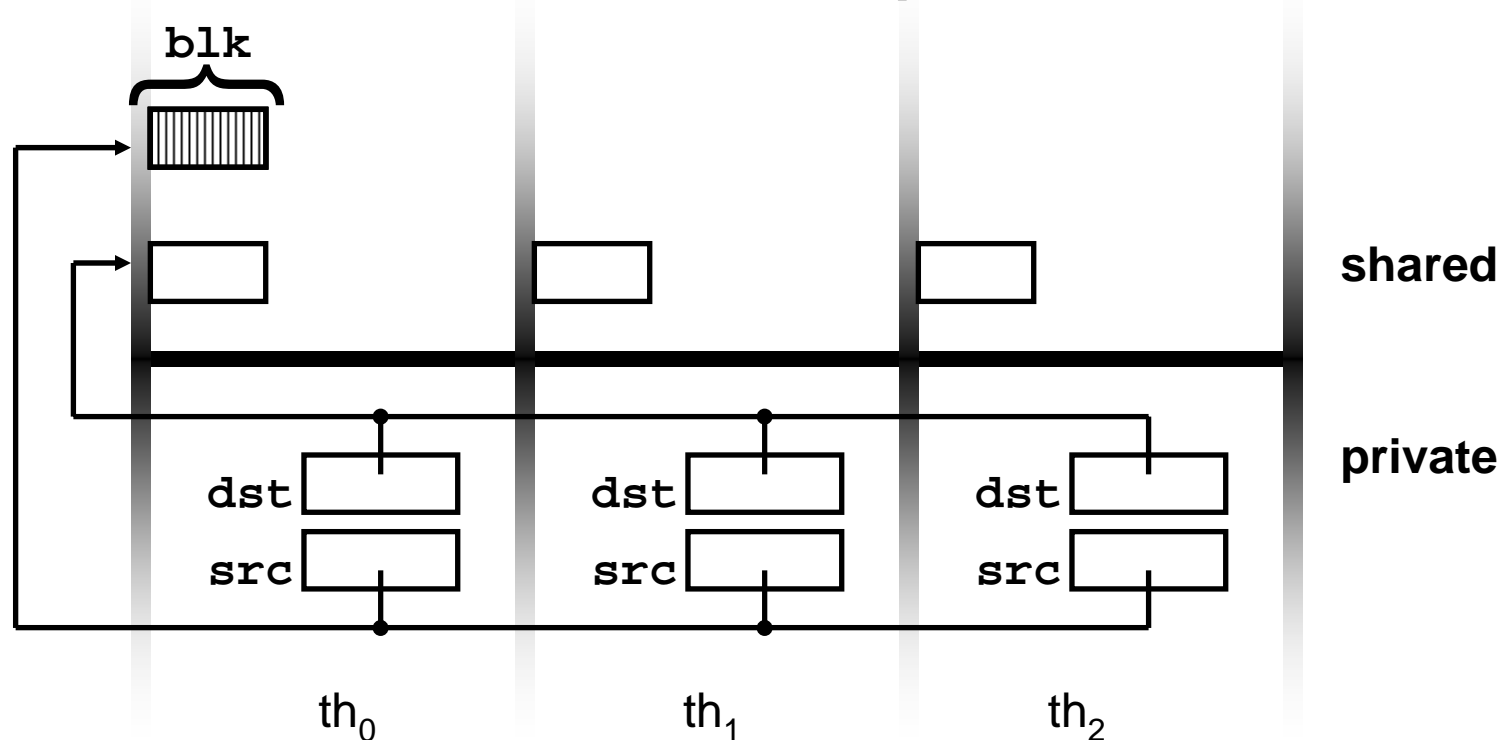


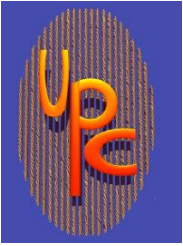
upc_all_broadcast

(animation)

Thread 0 sends the same block of data to each thread.

```
shared [blk] char dst[blk*THREADS];  
shared [] char src[blk];
```



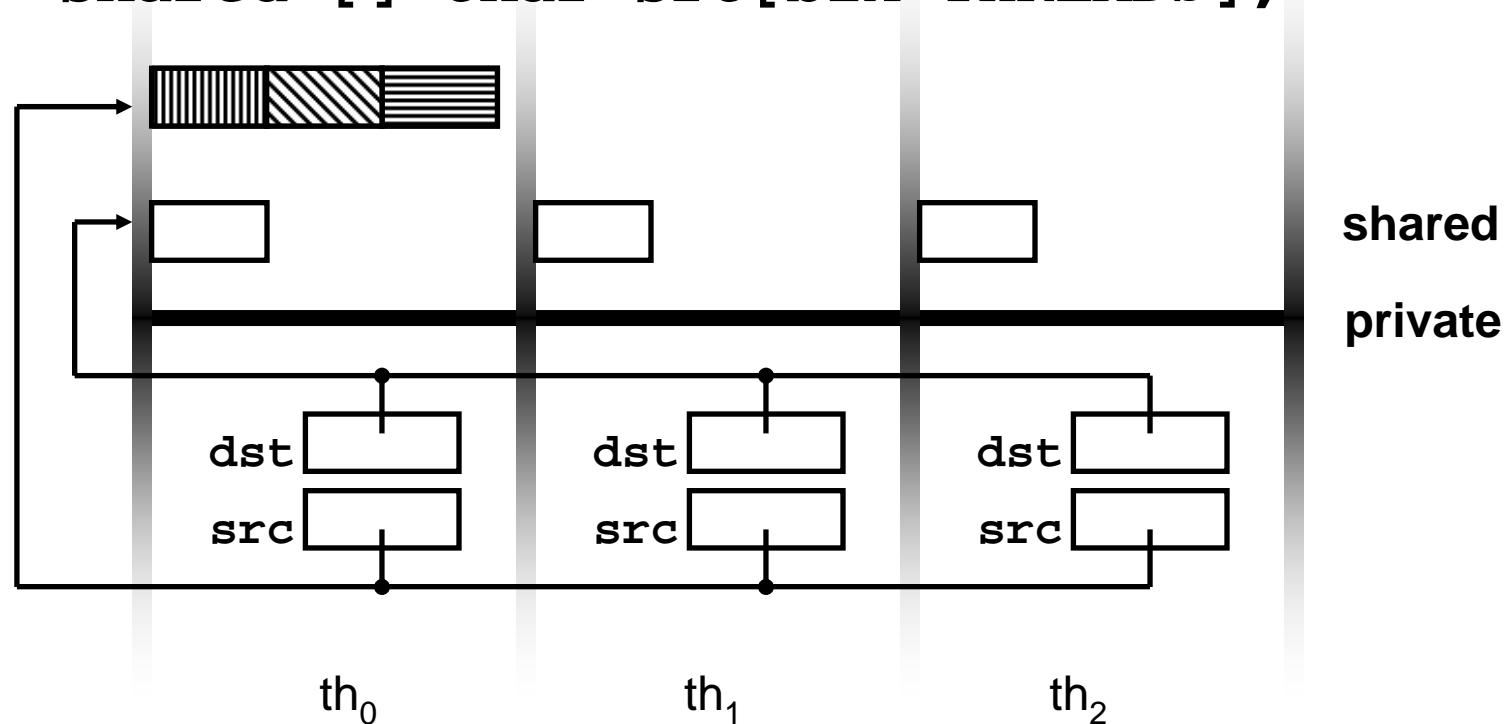


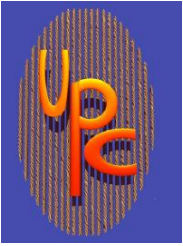
upc_all_scatter

(animation)

Thread 0 sends a unique block of data to each thread.

```
shared [blk] char dst[blk*THREADS];  
shared [] char src[blk*THREADS];
```



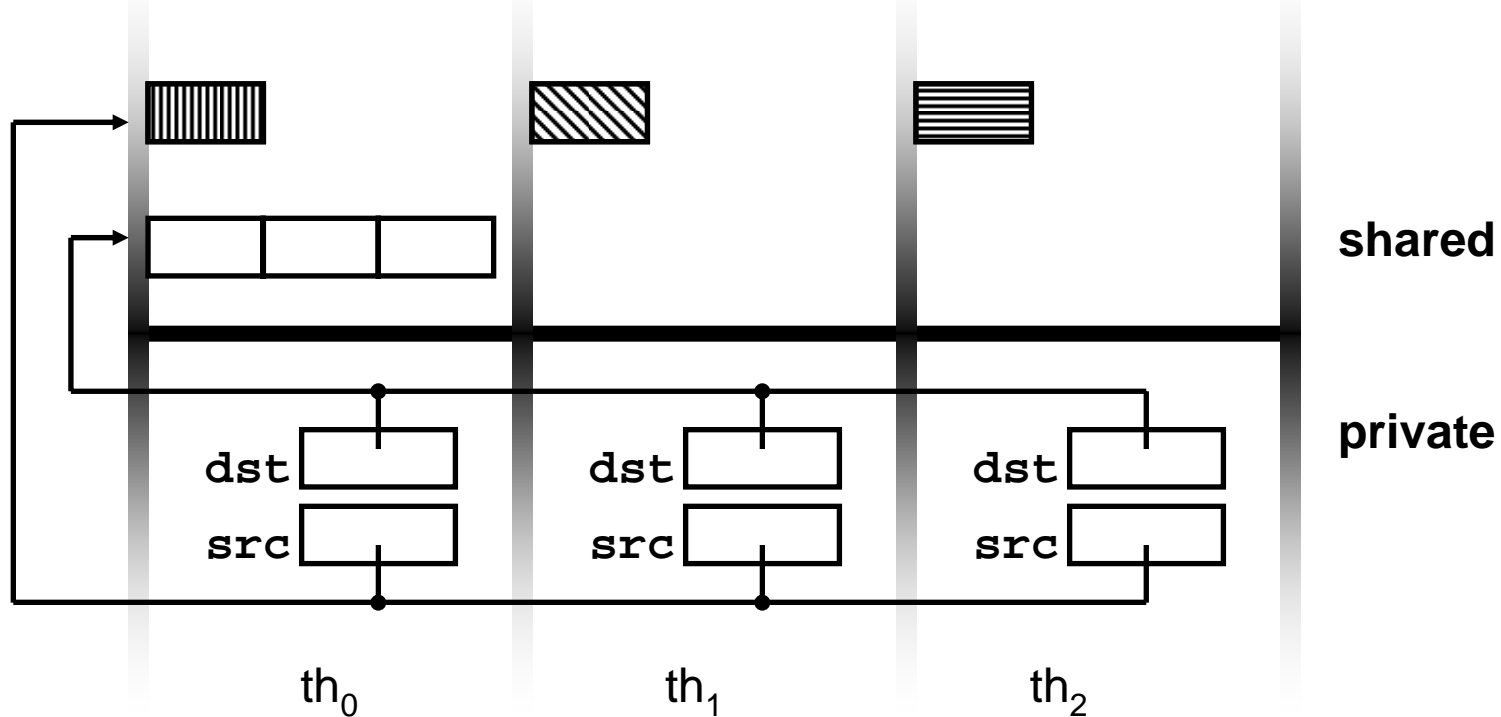


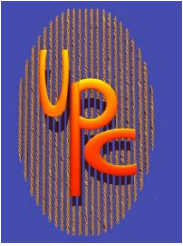
upc_all_gather

(animation)

Each thread sends a block of data to thread 0.

```
shared [] char dst[blk*THREADS];  
shared [blk] char src[blk*THREADS];
```

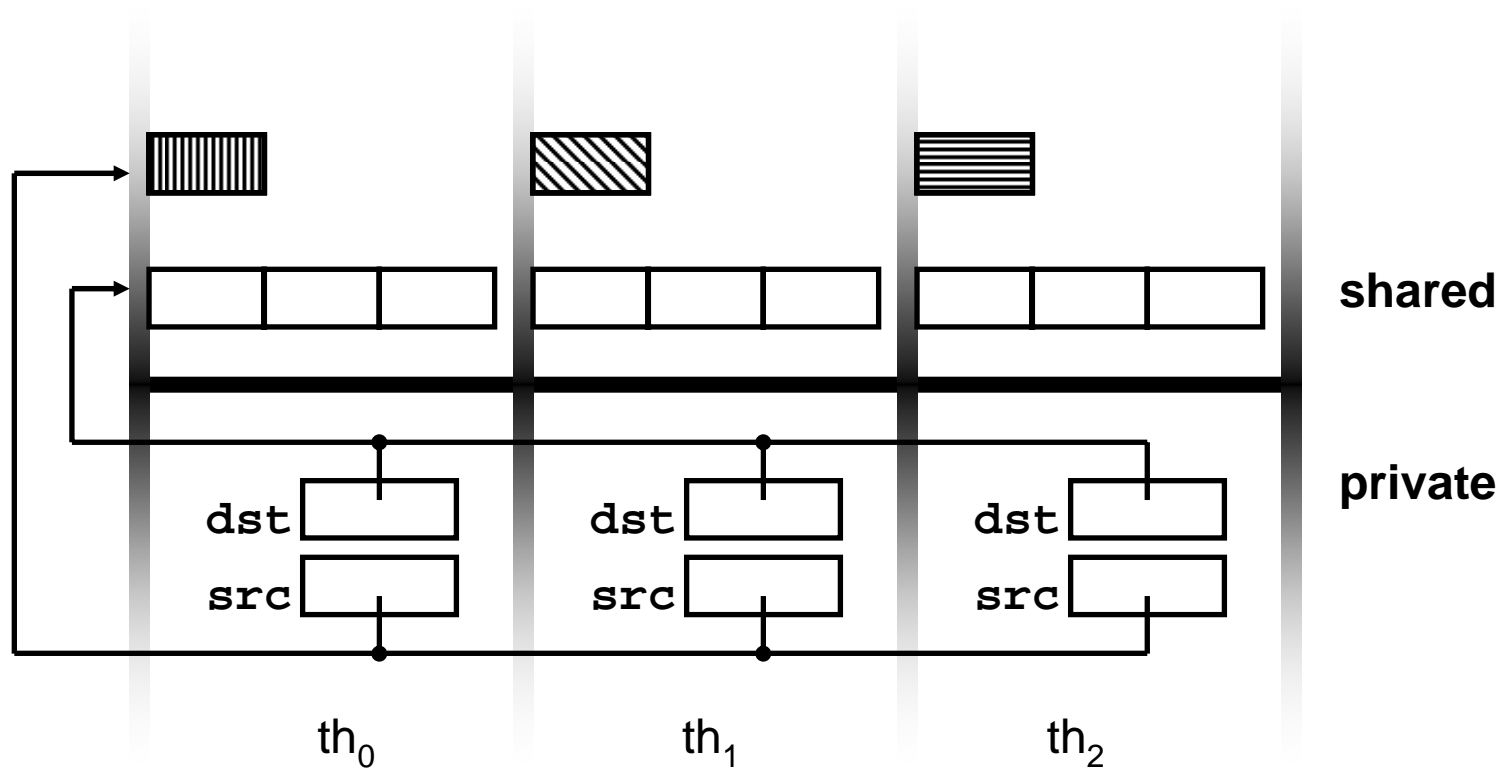


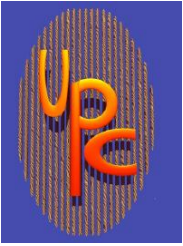


upc_all_gather_all

(animation)

Each thread sends one block of data to all threads.

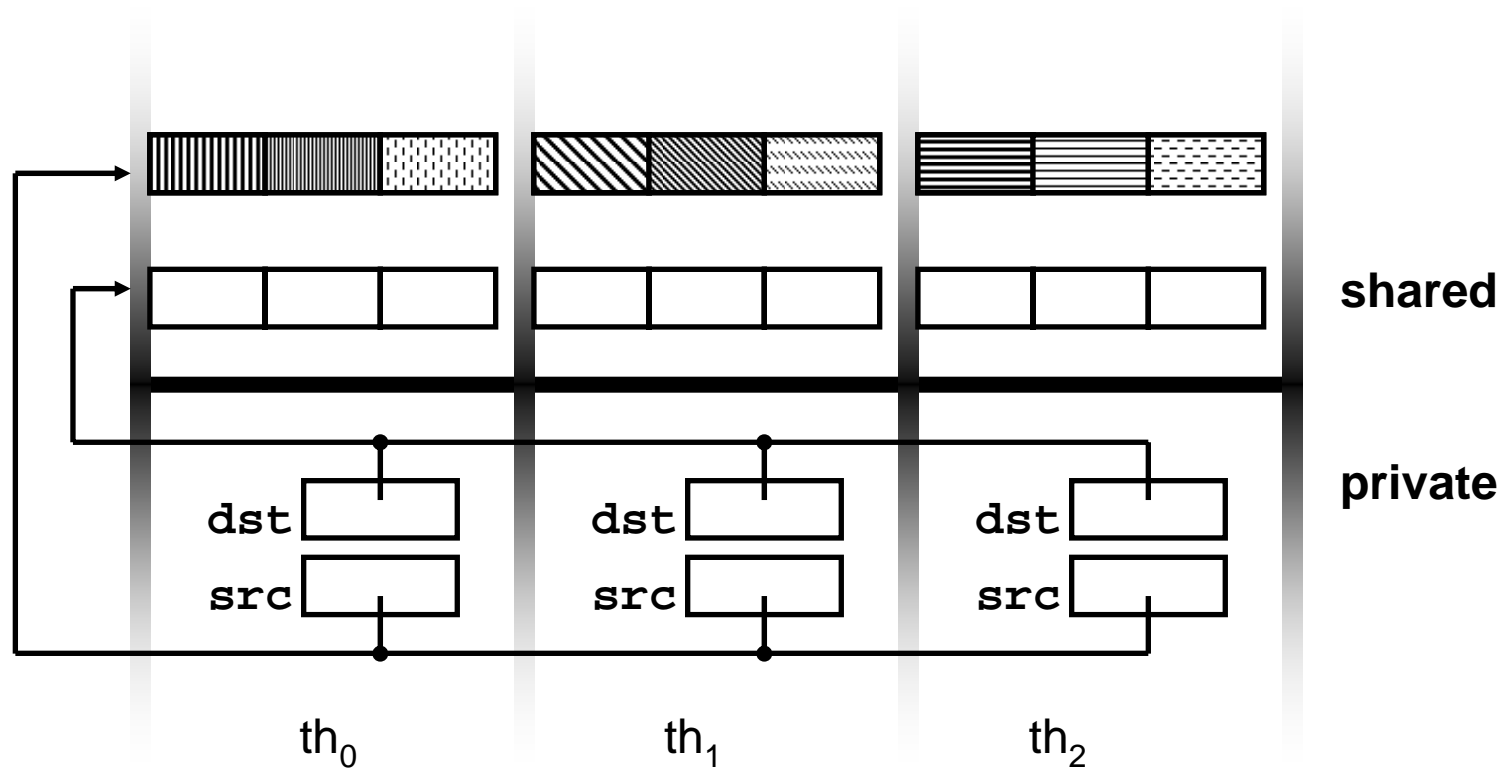


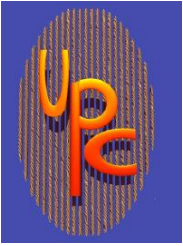


upc_all_exchange

(animation)

Each thread sends a unique block of data to each thread.

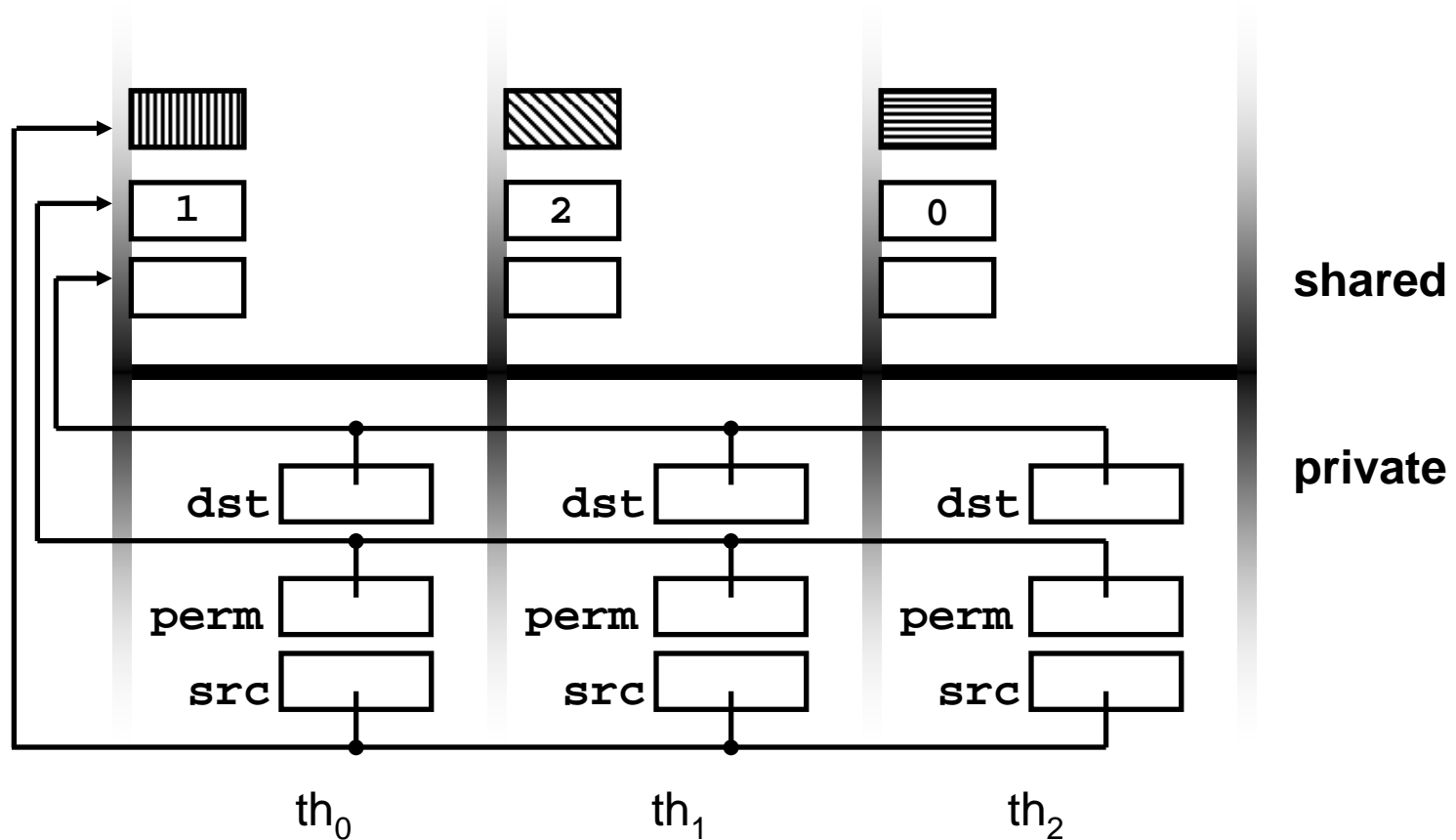


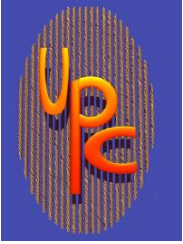


upc_all_permute

(animation)

Thread i sends a block of data to thread $perm(i)$.

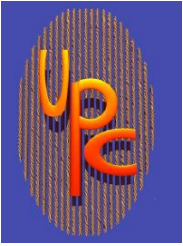




Reduce and prefix_reduce

- One function for each C scalar type, *e.g.*,
`upc_all_reduceI(...)` returns an Integer
- Operations
 - +, *, &, |, xor, &&, ||, min, max
 - user-defined binary function
 - non-commutative function option

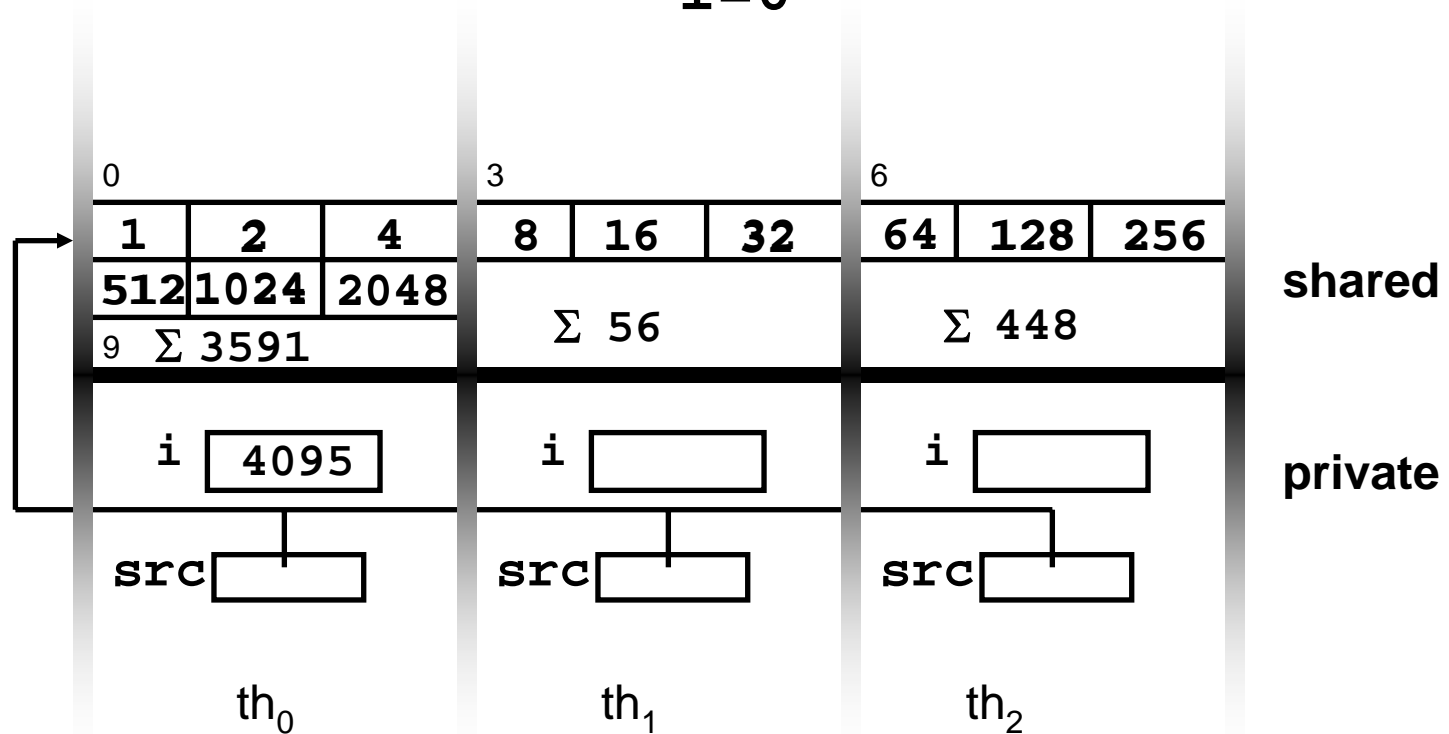


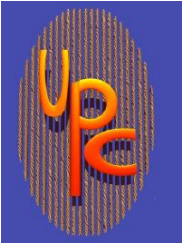


upc_all_reduce*TYPE*

(animation)

Thread 0 receives $\text{UPC_OP } \sum_{i=0}^n \text{src}[i]$.

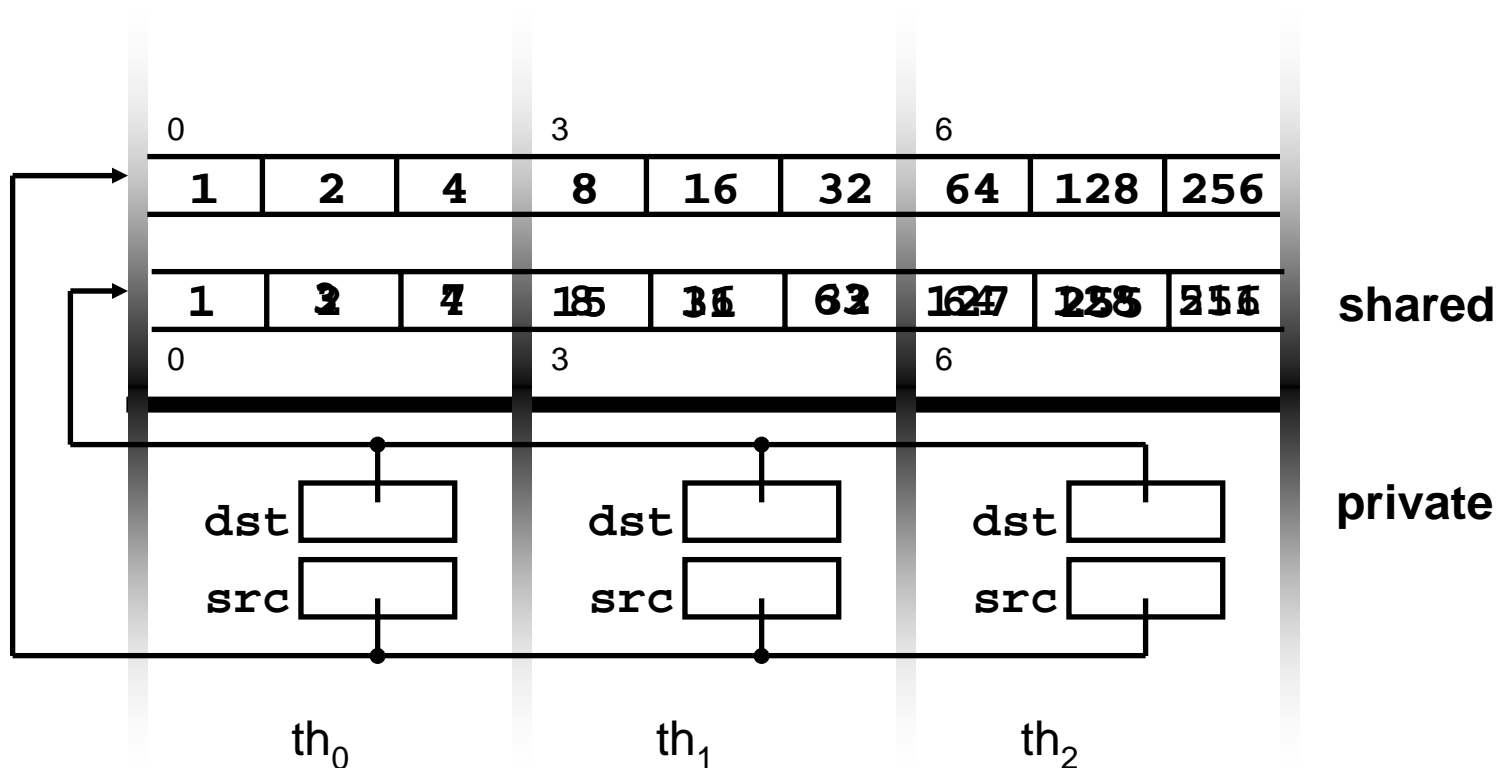


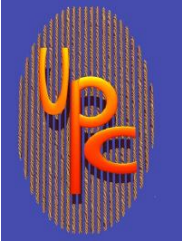


upc_all_prefix_reduce*TYPE*

(animation)

Thread k receives $UPC_OP\ src[i]$.
 $i=0$

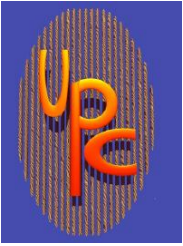




Common collectives properties

- Collectives function arguments are *single-valued*: corresponding arguments must match across all threads.
- Data blocks must have identical sizes.
- Source data blocks must be in the same array and at the same relative location in each thread.
- The same is true for destination data blocks.
- Various *synchronization modes* are provided.

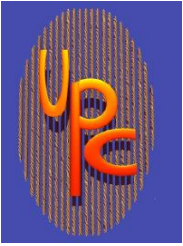




Synchronization modes

- Sync modes determine the “strength” of synchronization between threads executing a collective function.
- Sync modes are specified by flags for function entry and for function exit:
 - `UPC_IN_...`
 - `UPC_OUT_...`
- Sync modes have three “strengths”:
 - `...ALLSYNC`
 - `...MYSYNC`
 - `...NOSYNC`





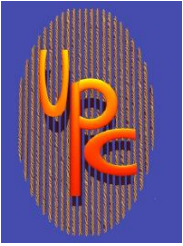
...ALLSYNC

- ...ALLSYNC provides barrier-like synchronization. It is the strongest and most convenient mode.

```
upc_all_broadcast(dst, src, nbytes,  
                 UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

- No thread will access collective data until all threads have reached the collective call.
- No thread will exit the collective call until all threads have completed accesses to collective data.





...NOSYNC

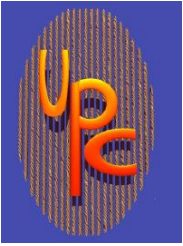
- ...NOSYNC provides weak synchronization. The programmer is responsible for synchronization.

Assume there are no data dependencies between the arguments in the following two calls:

```
upc_all_broadcast(dst0, src0, nbytes,  
                 UPC_IN_ALLSYNC | UPC_OUT_NOSYNC);  
upc_all_broadcast(dst1, src1, mbytes,  
                 UPC_IN_NOSYNC | UPC_OUT_ALLSYNC);
```

Chaining independent calls by using ...NOSYNC eliminates the need for synchronization between calls.





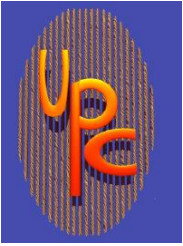
...MYSYNC

- Synchronization is provided with respect to data read (`UPC_IN...`) and written (`UPC_OUT...`) by each thread.
- ...**MYSYNC** provides an intermediate level of synchronization.

Assume thread 0 is the source thread. Each thread needs to synchronize only with thread 0.

```
upc_all_broadcast(dst, src, nbytes,  
                 UPC_IN_MYSYNC | UPC_OUT_MYSYNC);
```

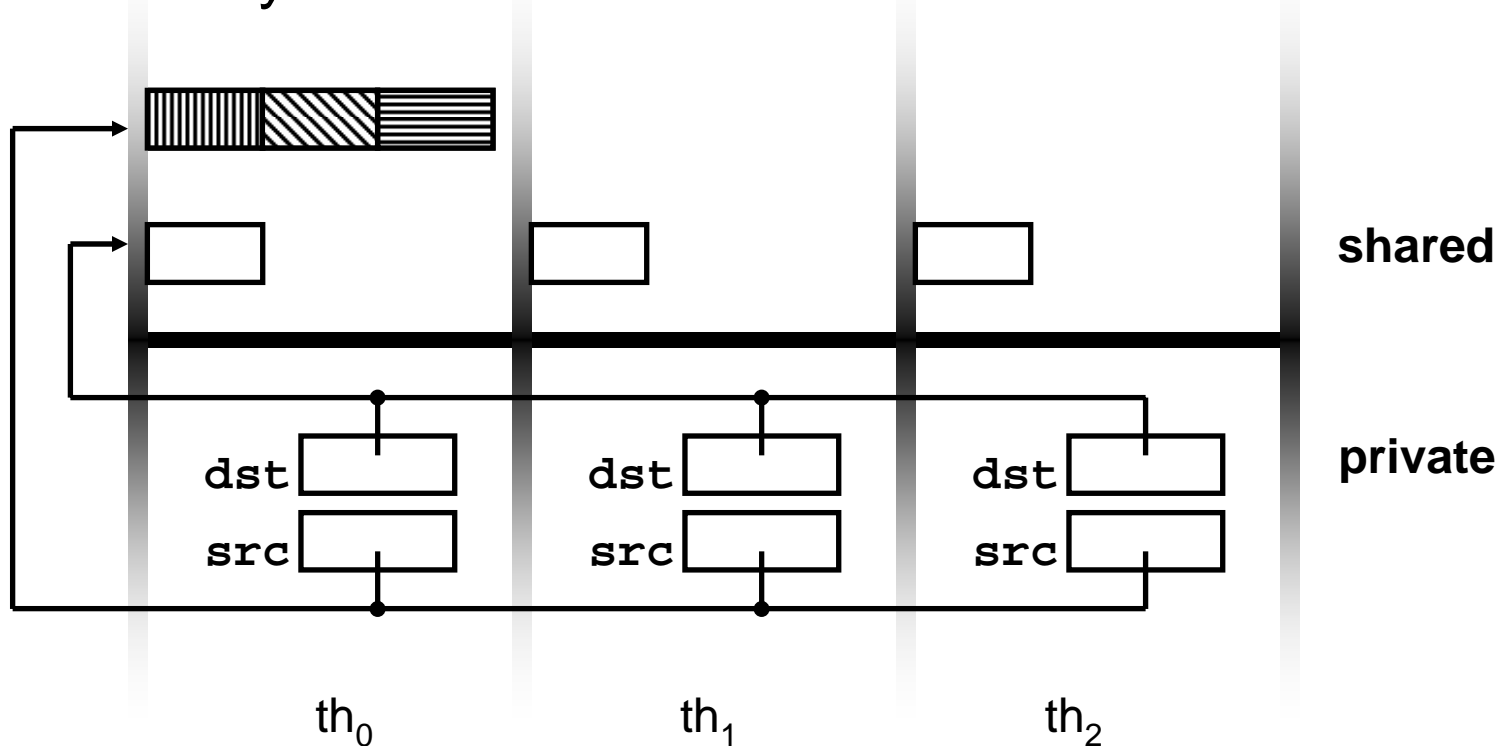


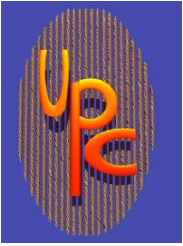


...MYSYNC example

(animation)

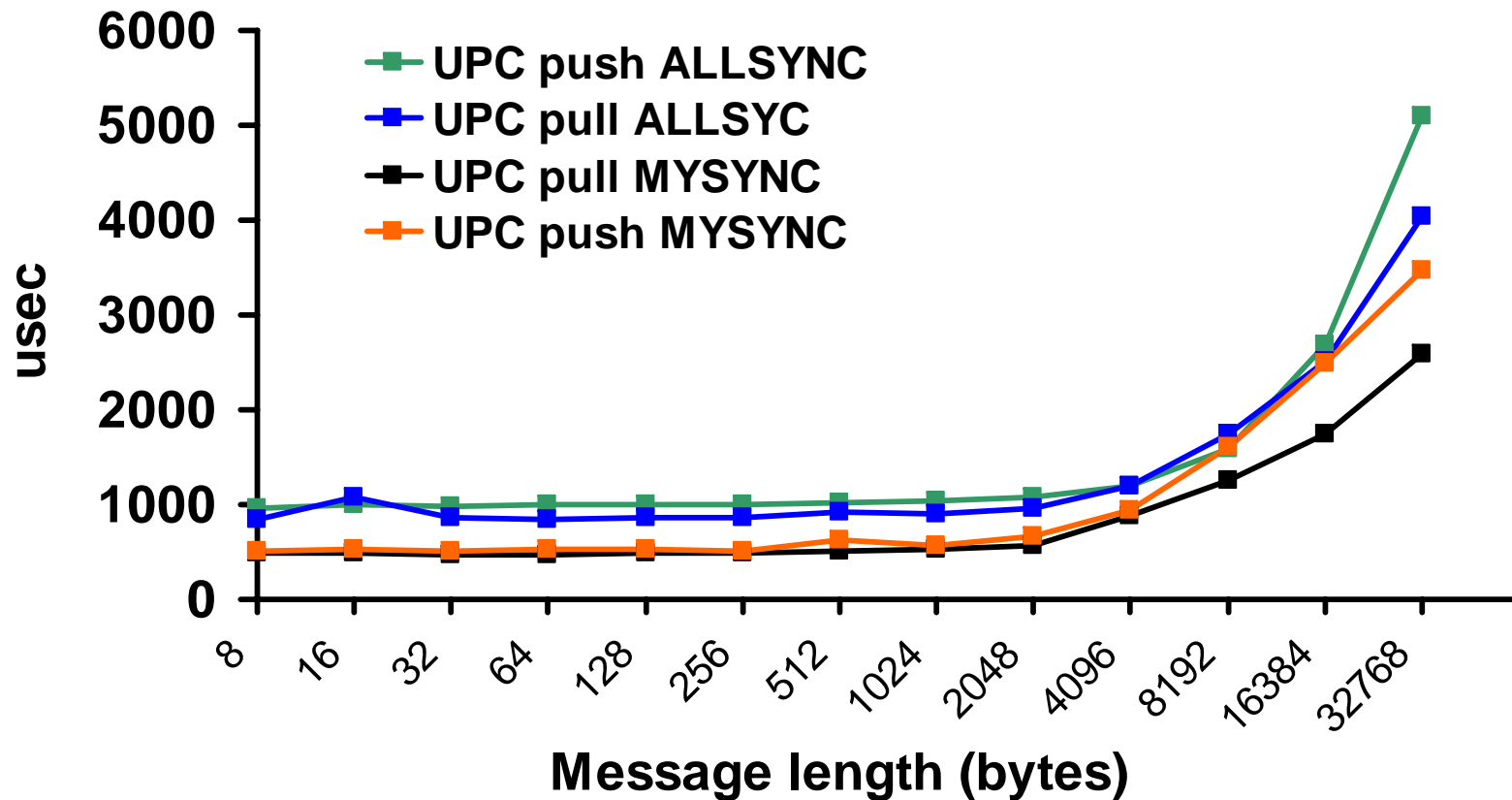
- Each thread synchronizes with thread 0.
- Threads 1 and 2 exit as soon as they receive the data.
- It is not likely that thread 2 needs to read thread 1's data.

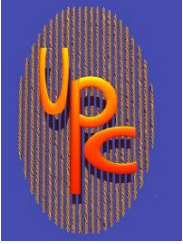




ALLSYNC vs. MYSYNC performance

`upc_all_broadcast()` on a Linux/Myrinet cluster, 8 nodes

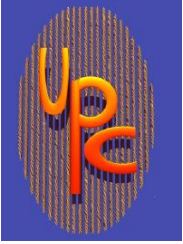




Sync mode summary

- ...**ALLSYNC** is the most “expensive” because it provides barrier-like synchronization.
- ...**NOSYNC** is the most “dangerous” but it is almost free.
- ...**MYSYNC** provides synchronization only between threads which need it. It is likely to be strong enough for most programmers’ needs, and it is more efficient.

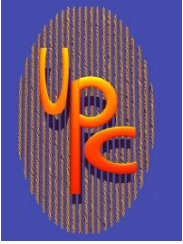




Collectives performance

- UPC-level implementations can be improved.
- Algorithmic approaches
 - tree-based algorithms
 - message combining (*cf.* chained broadcasts)
- Platform-specific approaches
 - RDMA “put” and “get” (e.g., Myrinet and Quadrics)
 - broadcast and barrier primitives may be a benefit
 - buffer management
 - static: permanent but of fixed size
 - dynamic: expensive if allocated for each use
 - pinned: defined RMDA memory area, best solution

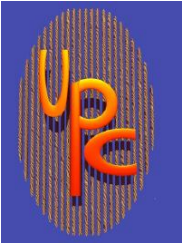




“Push” and “pull” animations

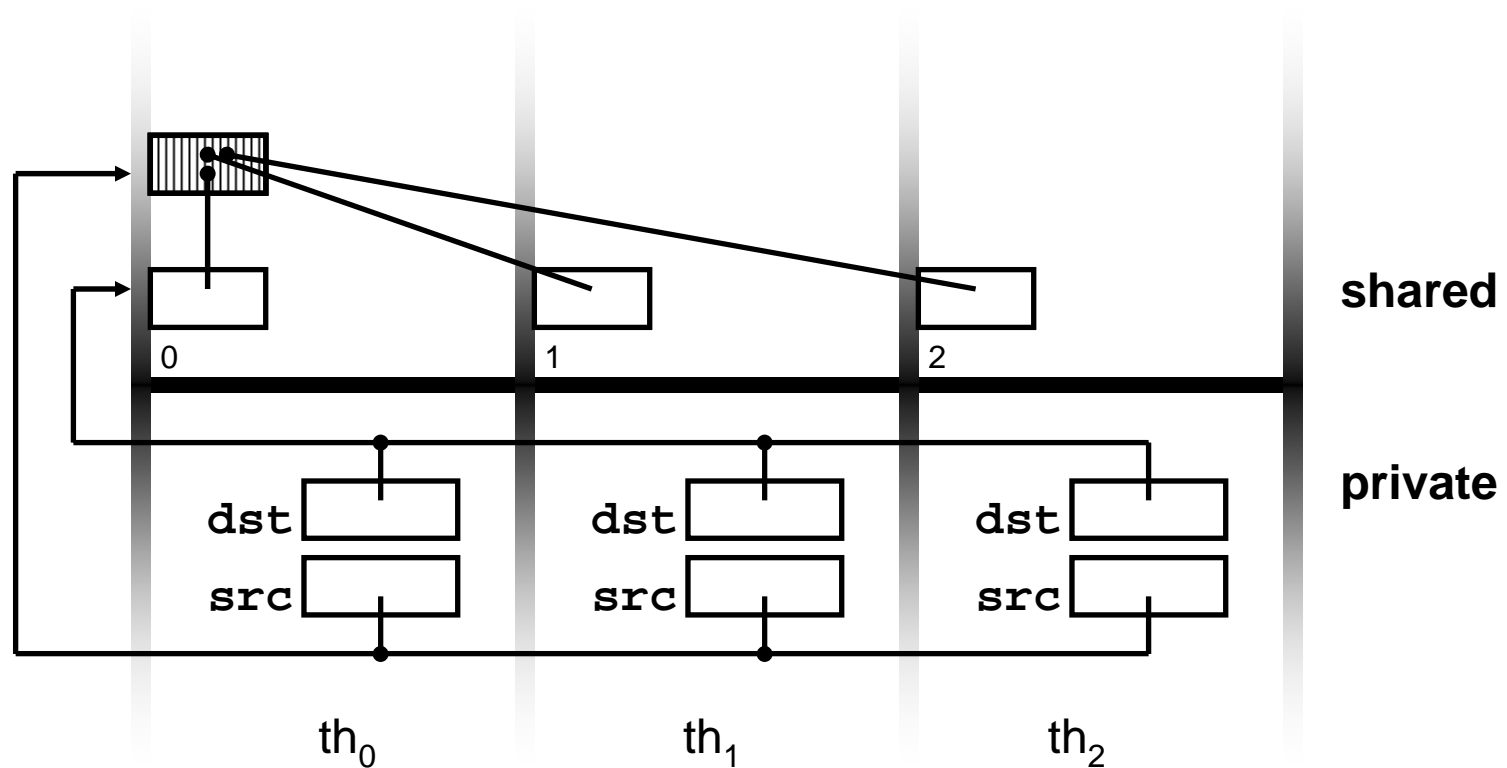
- The next two slides illustrate the concept of “push” and “pull” collectives implementations.
- UPC-level code is given, but these can be imagined to take place at any lower level in the transport network.

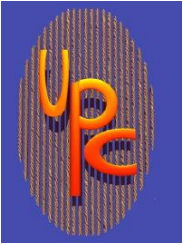




A “pull” implementation of `upc_all_broadcast` (animation)

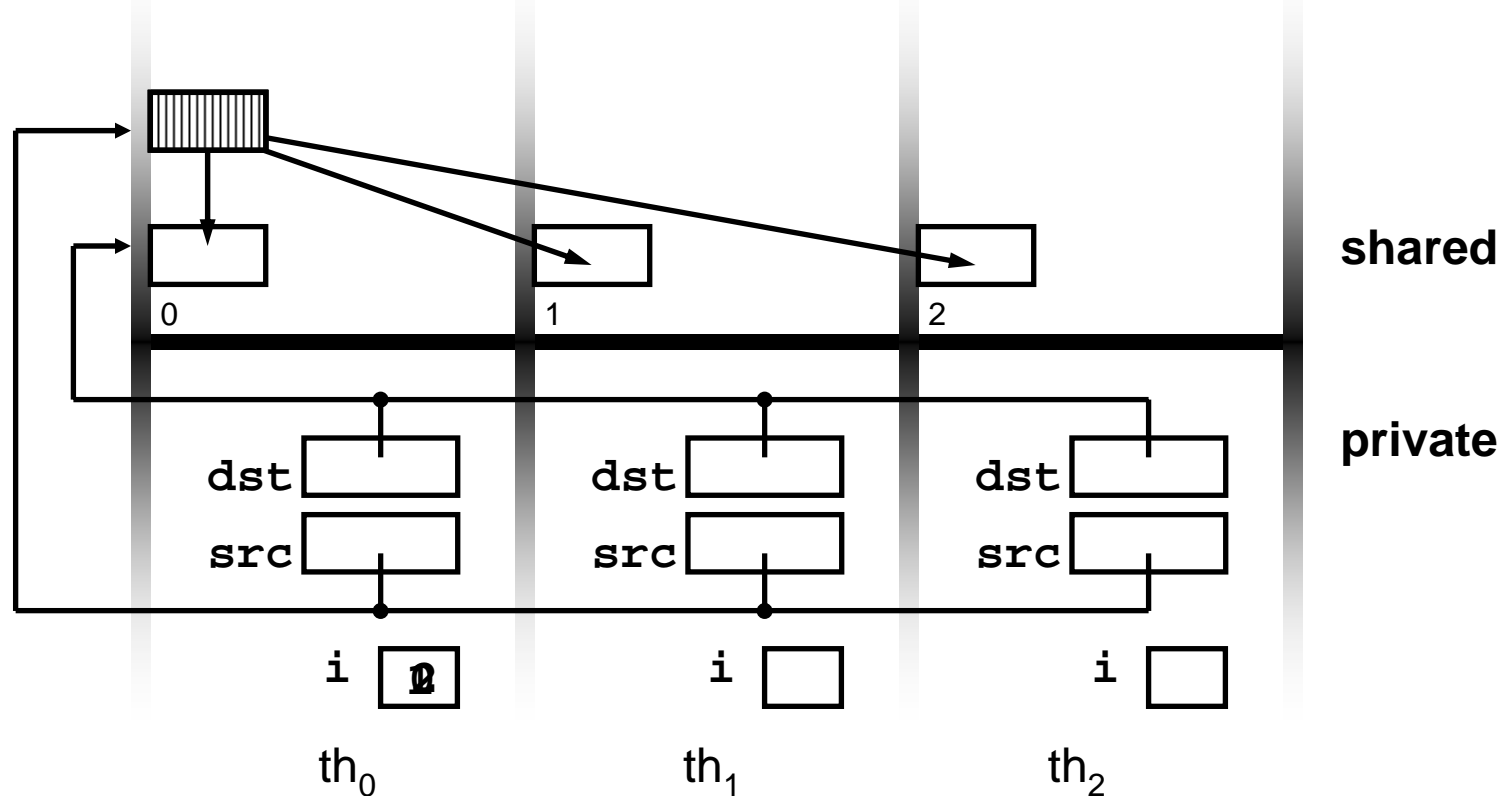
```
void upc_all_broadcast( shared void *dst, shared const void *src, size_t blk )  
{  
    upc_memcpy( (shared char *)dst + MYTHREAD, (shared char *)src, blk );  
}
```

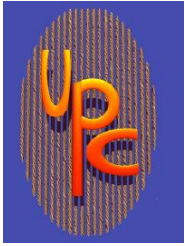




A “push” implementation of `upc_all_broadcast` (animation)

```
void upc_all_broadcast( shared void *dst, shared const void *src, size_t blk )  
{ int i;  
  upc_forall( i=0; i<THREADS; ++i; 0)    // Thread 0 only  
    upc_memcpy( (shared char *)dst + i, (shared char *)src, blk );  
}
```

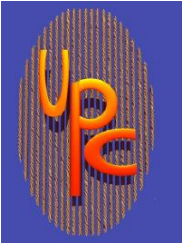




Sample performance improvements

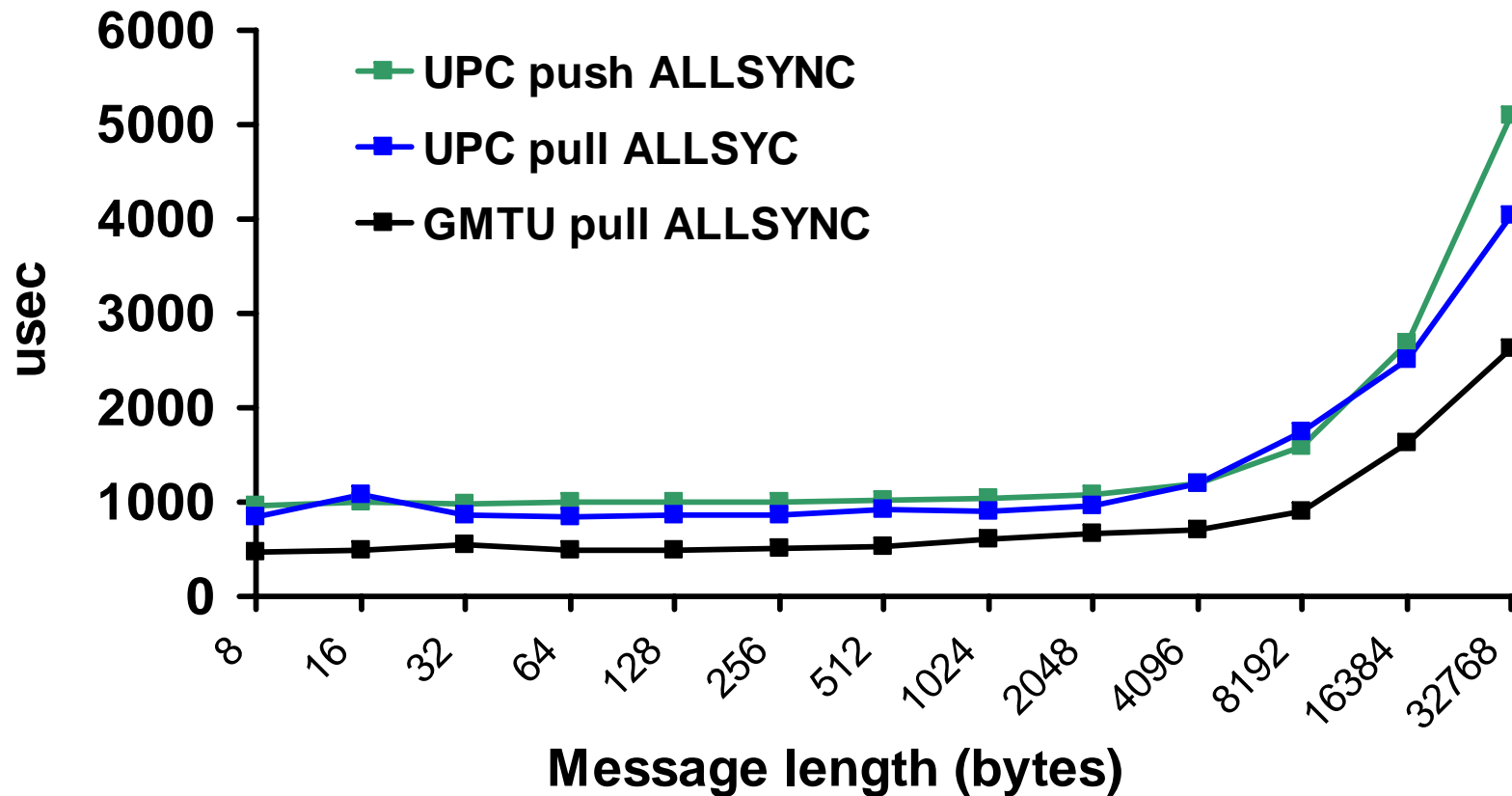
- Baseline: UPC collectives “reference implementation”. (*cf.* previous two slides)
- Algorithms
 - direct
 - tree-based
 - hand-coded tree-based barrier
- Platform
 - 16 dual-processor nodes with 2GHz Pentiums
 - Myrinet/GM-1 (allows only remote writes)

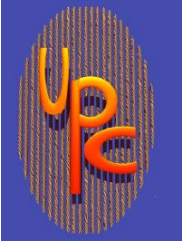




UPC-level vs. low-level implementation

`upc_all_broadcast()` on a Linux/Myrinet cluster, 8 nodes

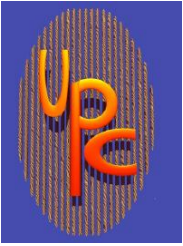




Performance issue: block size

- The performance of `upc_all_prefix_reduce` is affected by block size.
- Block size [*] is best. (*cf.* earlier animation)
- Penalties of small block size:
 - many more remote memory access
 - much more synchronization
- The following animation illustrates the penalties.

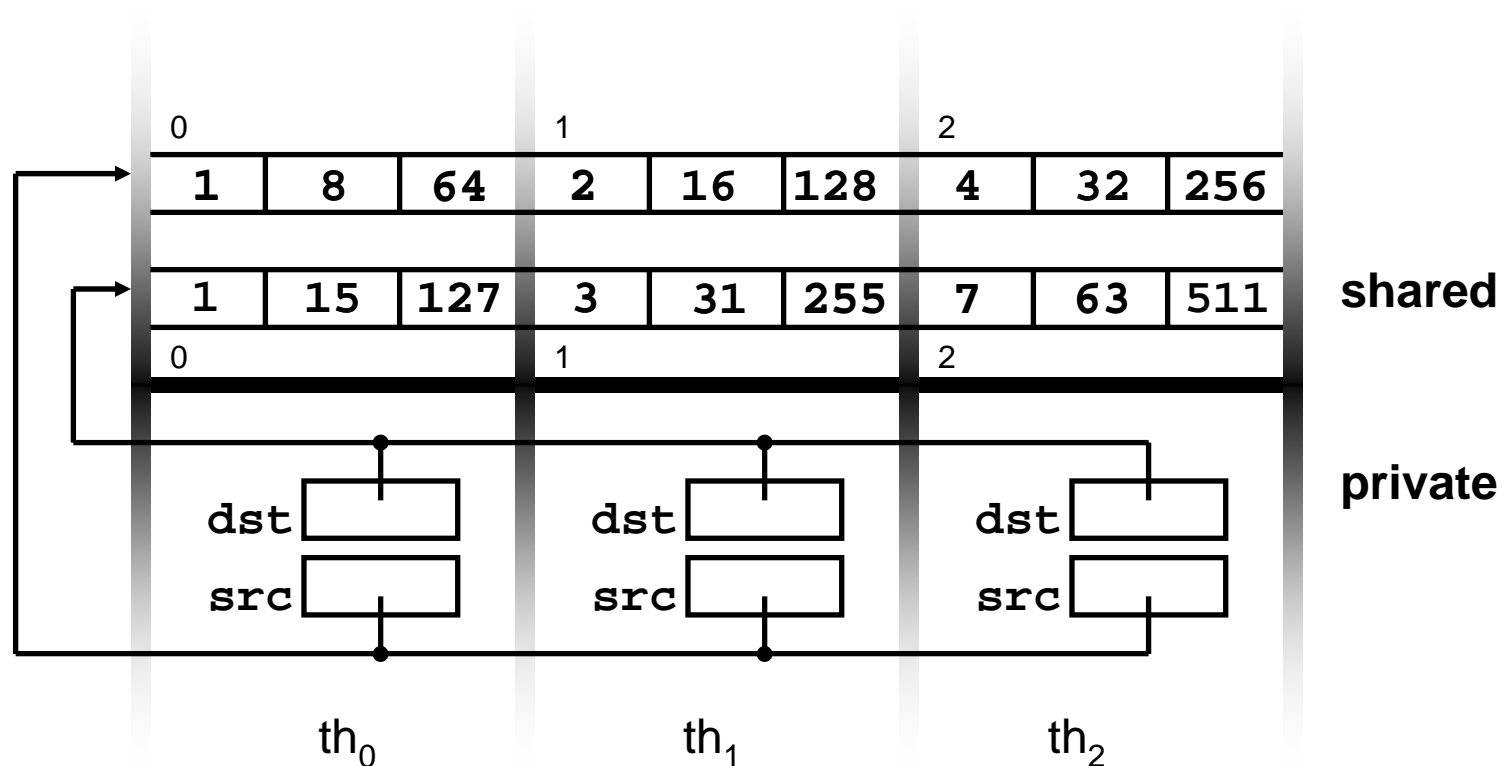


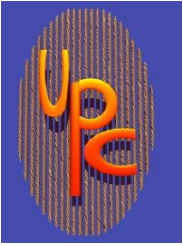


Penalties for small block size

(animation)

```
shared [1] int src[3*THREADS], dst[3*THREADS];  
upc_all_prefix_reduceI(dst, src, UPC_ADD, n, blk, NULL);
```

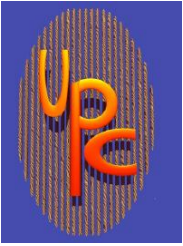




Performance improvements

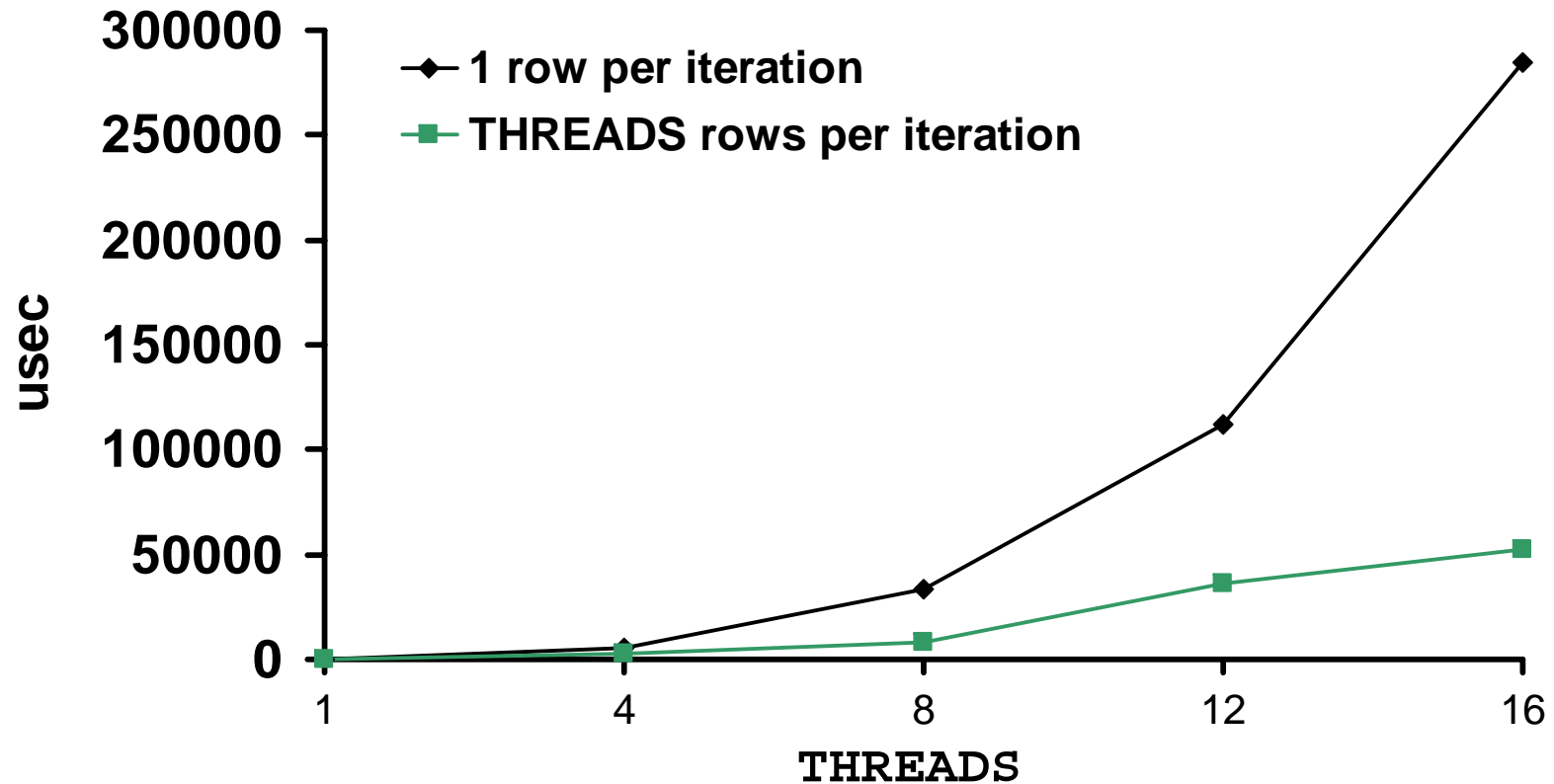
- Performance of `upc_all_prefix_reduce` cannot be immediately improved with commonly used algorithms because of additional synchronization costs.
 - odd-even
 - Fisher-Ladner
- Computing over *groups of blocks* during each synchronization phase reduces the number of remote references and synchronization points in proportion to group size at the cost of more memory.

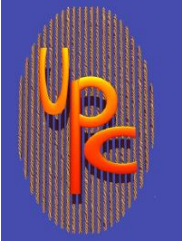




upc_all_prefix_reduce() performance

Problem size: $1024 * \text{THREADS}^3$
Compaq Alphaserver SC-40

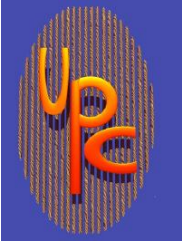




Collectives extensions

- The UPC consortium is working on a draft extension of the set of collectives, including:
 - asynchronous variants
 - non-single-valued arguments
 - arbitrary locations for sources and destinations
 - variable-sized data blocks
 - an in-place option
 - private-to-private variants
 - thread subsetting

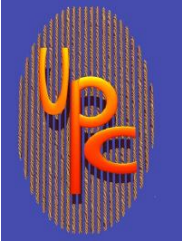




Overview of UPC-IO

- All UPC-IO functions are collective
 - Most arguments are single-valued
- Library based API
 - Enables ease of plugging into existing parallel IO facilities (e.g. MPI-IO)
- UPC-IO data operations support:
 - shared and private buffers
 - Two kinds of file pointers:
 - individual (i.e. per-thread)
 - common (i.e. shared by all threads)
 - Blocking and non-blocking (asynchronous) operations
 - Currently limited to one asynchronous operation in-flight per file handle

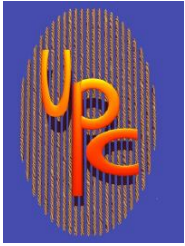




Overview of UPC-IO (cont.)

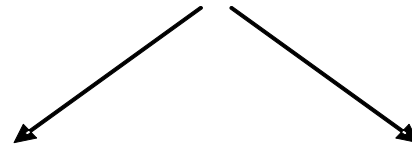
- Supports List-IO Access
 - Each thread provides a list of:
 - (address, length) tuples referencing shared or private memory
 - (file offset, length) tuples referencing the file
 - Supports arbitrary size combinations, with very few semantic restrictions (e.g. no overlap within a list)
 - Does not impact file pointers
 - Fully general I/O operation
 - Generic enough to implement a wide range of data movement functions, although less convenient for common usage





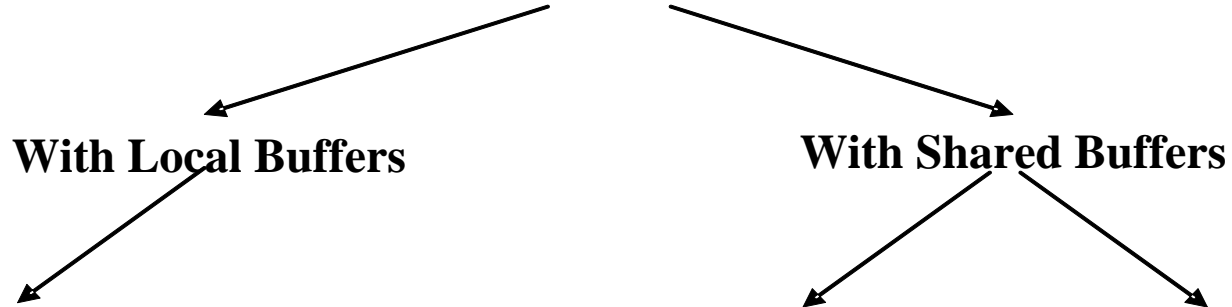
File Accessing and File Pointers

List I/O Access using Explicit Offsets



With Local Buffers With Shared Buffers

File I/O with File Pointers



With Individual FP

With Individual FP

With Common FP

All Read/Write operations have blocking and asynchronous (non-blocking) variants

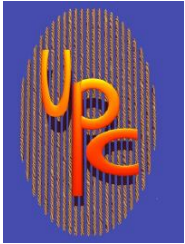




Consistency & Atomicity Semantics

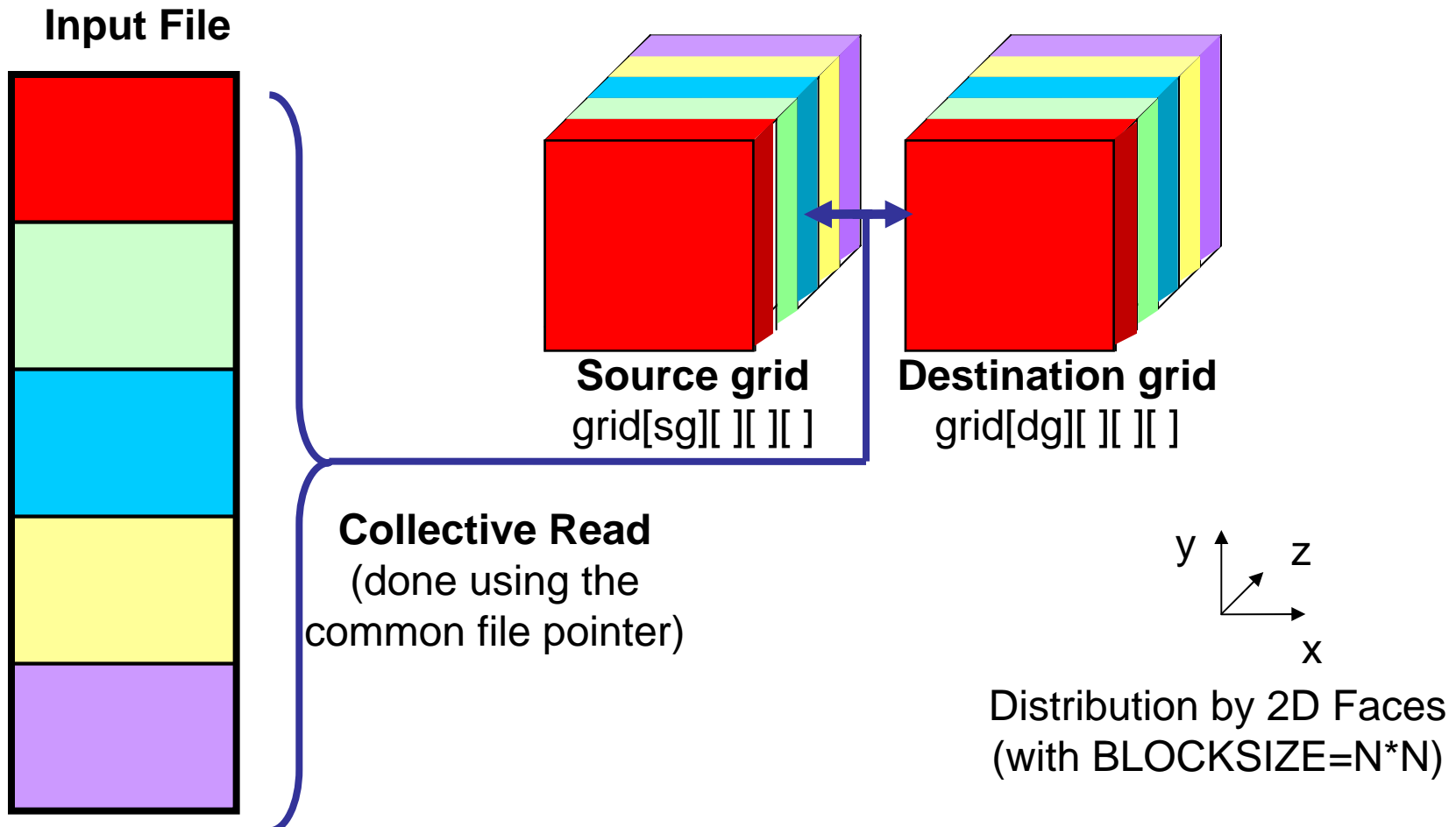
- UPC-IO atomicity semantics define the outcome of overlapping concurrent write operations to the file from separate threads
- Default mode: weak semantics
 - overlapping or conflicting accesses have undefined results, unless they are separated by a `upc_all_fsync` (or a file close)
- The semantic mode is chosen at file open time, and can be changed or queried for an open file handle using `upc_all_fcntl`
- Strong consistency and atomicity enforces the semantics for the case of writes from multiple threads to overlapping regions for which the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order

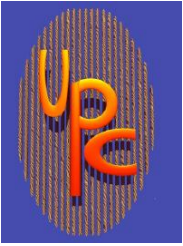




3D Heat Conduction - Initialization from file

- Read the initial conditions from an input file





3D Heat Conduction - Initialization from file

```
shared [BLOCKSIZE] double grids[2][N][N][N];

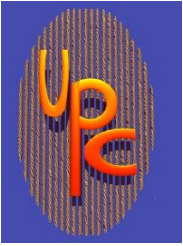
void initialize(void)
{
    upc_file_t *fd;

    fd = upc_all_fopen( "input.file", UPC_RDONLY|UPC_COMMON_FP,
        0, NULL );
    /* because the BLOCKSIZE is equal to N*N*N
       each thread reads blocksize for 2 times */
    upc_all_fread_shared( fd, (shared void *)grids[0][0][0][0],
        BLOCKSIZE, sizeof(double), N*N*N, UPC_IN_NOSYNC |
        UPC_OUT_NOSYNC);
    upc_all_fseek( fd, 0, UPC_SEEK_SET );
    upc_all_fread_shared( fd, (shared void *)grids[1][0][0][0],
        BLOCKSIZE, sizeof(double), N*N*N, UPC_IN_NOSYNC |
        UPC_OUT_NOSYNC);

    upc_all_fclose( fd ); }

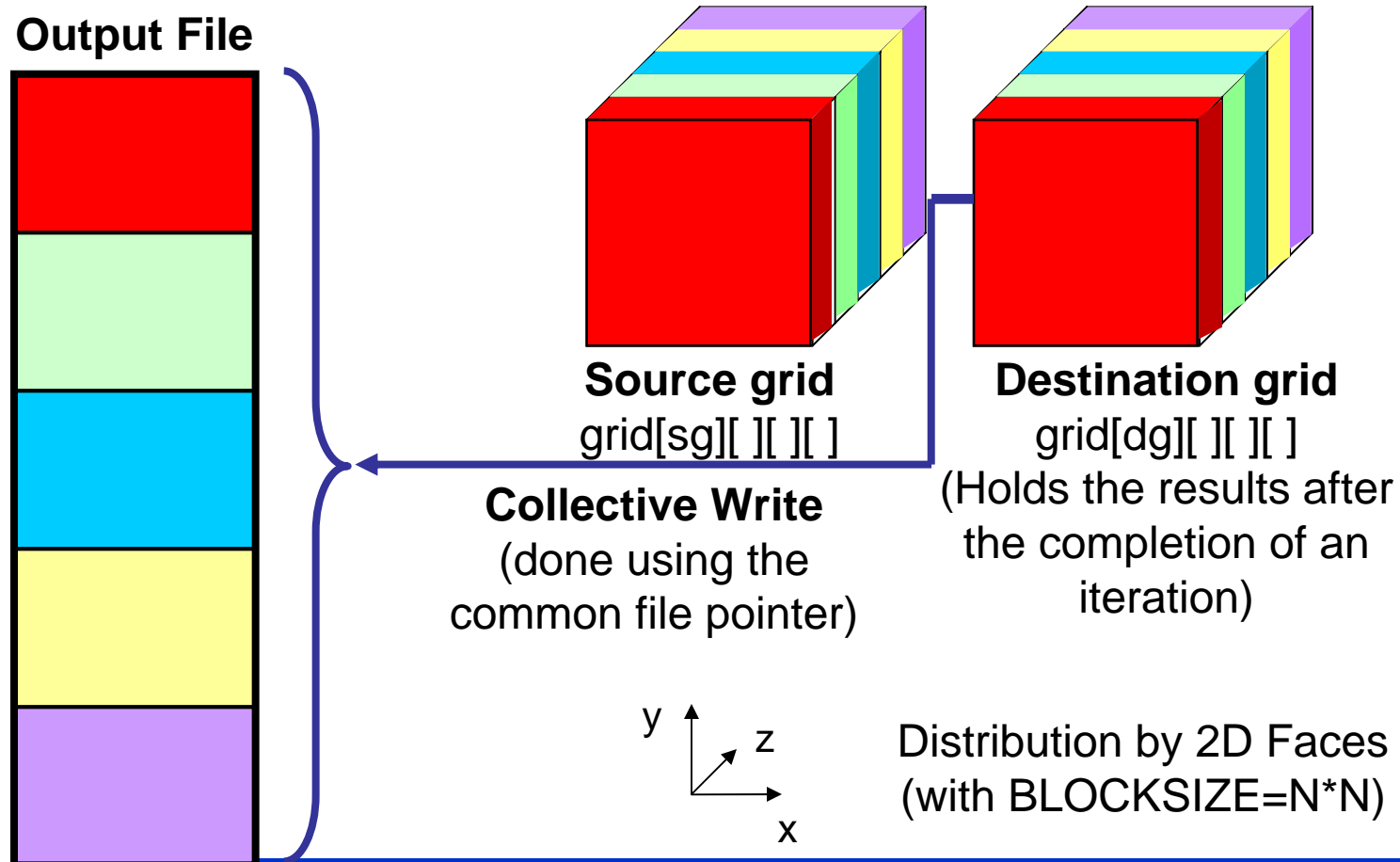
```

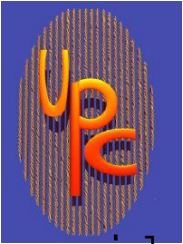




3D Heat Conduction - Output to file

- Output collectively an iteration in a file



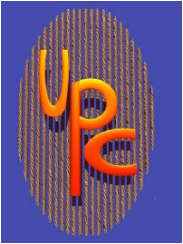


3D Heat Conduction – Outputting each iteration

```
void file_output(int dg, int
  iteration)
{
  upc_file_t *fd;  char
  filename[100];
  sprintf( filename,
    "output.file.iter%d",
    iteration );
  fd = upc_all_fopen( filename,
    UPC_WRONLY|UPC_COMMON_FP|UPC
    _CREATE, 0, NULL );
  upc_all_fwrite_shared( fd,
    (shared void *)
    &grids[dg][0][0][0],
    BLOCKSIZE, sizeof(double),
    N*N*N, UPC_IN_ALLSYNC |
    UPC_OUT_NOSYNC );
  upc_all_fclose( fd ); }
```

```
int heat_conduction() {
  ...
  if( dTmax < epsilon ) { ... }
  else {
    if( (nr_iter%100) == 0 ) //
    each 100 iterations
      file_output(dg, nr_iter);
    ... // swapping the source and
    destination "pointers"
  }
```

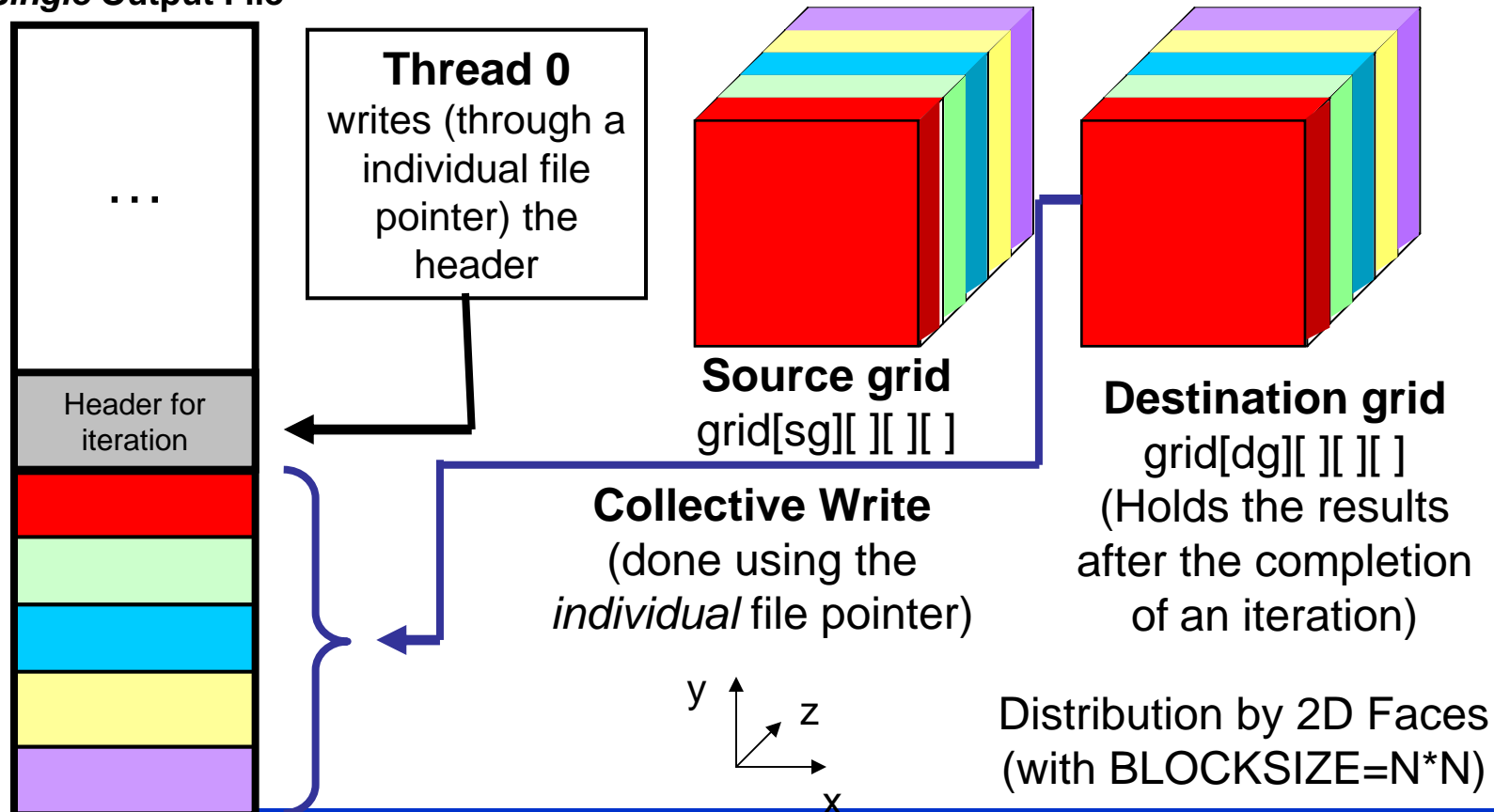




3D Heat Conduction – Output to file

- Appends iteration results to a ‘huge’ result file (as well as a header, prior the data of each iteration)

Single Output File



3D Heat Conduction – Outputting each iteration as part of a same file

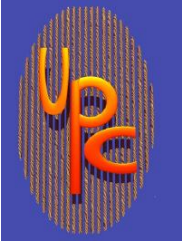
```
void file_output(int dg, int
    iteration)
{
    upc_file_t *fd; char
    buf[100];

    if( MYTHREAD == 0 ) {
        sprintf( buf,
            "\n\nIteration #%d\n\n",
            iteration );
        length = strlen( buf );
    } else length = 0;

    fd = upc_all_fopen(
        "output.file",
        UPC_WRONLY|UPC_INDIVIDUAL_
        FP|UPC_APPEND, 0, NULL );
```

```
    upc_all_fwrite_local( fd,
        (void *)buf, sizeof(char),
        length, UPC_IN_NOSYNC |
        UPC_OUT_ALLSYNC );
        upc_all_fseek( fd,
        BLOCKSIZE*MYTHREAD*sizeof(doub
        le), UPC_SEEK_END );
        upc_all_fwrite_shared( fd,
        (shared void *)
        (grids+dg*N*N*N+BLOCKSIZE*MYTH
        READ), BLOCKSIZE,
        sizeof(double), BLOCKSIZE,
        UPC_IN_ALLSYNC |
        UPC_OUT_NOSYNC );
        upc_all_fclose( fd );
    }
```

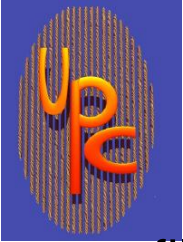




List of Functions defined by UPC-IO

- upc_all_fopen
- upc_all_fclose
- upc_all_fseek
- upc_all_fsync
- upc_all_fset_size
- upc_all_fget_size
- upc_all_fpreallocate
- upc_all_fcntl
- upc_all_fread_local [*_async*]
- upc_all_fread_shared [*_async*]
- upc_all_fwrite_local [*_async*]
- upc_all_fwrite_shared [*_async*]
- upc_all_fread_list_local [*_async*]
- upc_all_fread_list_shared [*_async*]
- upc_all_fwrite_list_local [*_async*]
- upc_all_fwrite_list_shared [*_async*]
- upc_all_fwait_async
- upc_all_ftest_async





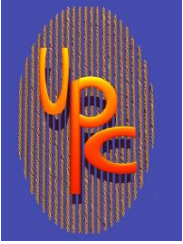
Basic UPC-IO functions

```
upc_file_t *upc_all_fopen( const char *filename, int flags,  
                           size_t numhints, upc_hints_t const *hints)
```

- Opens file, returns a pointer to a file handle, or NULL w/ errno
- File open flags:
 - **Must explicitly choose one read/write mode:**
 - UPC_RDONLY or UPC_WRONLY or UPC_RDWR
 - **and one file pointer mode:**
 - UPC_INDIVIDUAL_FP or UPC_COMMON_FP
 - **other optional flags (analogous to POSIX):**
 - UPC_APPEND, UPC_CREATE, UPC_EXCL, UPC_TRUNC
 - UPC_DELETE_ON_CLOSE
 - **UPC-specific optional flags:**
 - UPC_STRONG_CA - select strong consistency/atomicity mode

• Optional list of hints: for providing parallel I/O access hints





Basic UPC-IO functions (cont.)

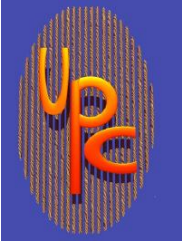
int **upc_all_fclose**(upc_file_t *fd);

- Close file, clean up file handler related metadata
- Implicit upc_all_fsync operation inferred

int **upc_all_fsync**(upc_file_t *fd);

- Ensures that any data that has been written to the file associated with *fd* but not yet transferred to the storage device is transferred to the storage device – a flush!



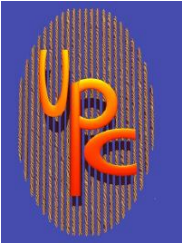


Basic UPC-IO functions (cont.)

`upc_off_t upc_all_fseek(upc_file_t *fd, upc_off_t offset, int origin)`

- sets and queries the current position of the file pointer for `fd`
- Threads pass an origin analogous to C99:
 - **UPC_SEEK_SET or UPC_SEEK_CUR or UPC_SEEK_END**
 - **and an offset from that origin**
- All arguments must be single-valued for common file pointer
 - **may be thread-specific for individual file pointer**
- Returns the new file pointer position to each thread
 - **Can also be used to simply query file pointer position using:
`upc_all_fseek(fd, 0, UPC_SEEK_CUR)`**
- Seeking past EOF is permitted
 - **writes past EOF will grow the file (filling with undefined data)**



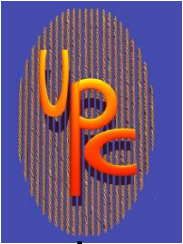


Basic UPC-IO functions (cont.)

```
ssize_t upc_all_fread/fwrite_local( upc_file_t *fd,  
                                     void *buffer,  
                                     size_t size, size_t nmemb,  
                                     upc_flag_t sync_mode)
```

- Reads/Writes data from a file to/from a private buffer on each thread
 - **File must be open for read/write with individual file pointers**
 - **Each thread passes independent buffers and sizes**
 - **Number of bytes requested is size * nmemb (*may be zero for no-op*)**
- Returns the number of bytes successfully read/written for each thread
 - **Each individual file pointer is incremented by corresponding amount**
 - **May return less than requested to indicate EOF**
 - **On error, it returns -1 and sets errno appropriately**



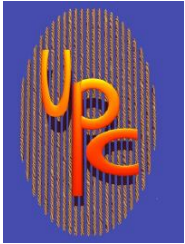


Basic UPC-IO functions (cont.)

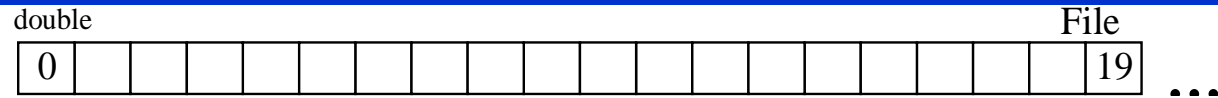
```
ssize_t upc_all_fread/fwrite_shared( upc_file_t *fd,  
                                     shared void *buffer, size_t blocksize,  
                                     size_t size, size_t nmemb,  
                                     upc_flag_t sync_mode)
```

- Reads/Writes data from a file to/from a shared buffer in memory:
 - **Number of bytes requested is $size * nmemb$ (*may be zero for no-op*)**
 - **Buffer may be an arbitrarily blocked array, but input phase ignored (assumed 0)**
- When file is open for read/write with an *individual* file pointer:
 - **Each thread passes independent buffers and sizes**
 - **Returns the number of bytes successfully read/written for each thread and increments individual file pointers by corresponding amount**
- When file is open for read/write with a *common* file pointer:
 - **All threads pass same buffer and sizes (all arguments single-valued)**
 - **Returns the total number of bytes successfully read to all threads and increments the common file pointer by a corresponding amount**





Basic UPC-IO Examples



Thread 0
File offset = 0

Thread 1
File offset = 5*sizeof(double)

Thread 2
File offset = 10*sizeof(double)

Thread 3
File offset = 15*sizeof(double)

```
double buffer[10]; // and assuming a
... // total of 4 THREADS
upc_file_t *fd =
    upc_all_fopen( "file", UPC_RDONLY |
        UPC_INDIVIDUAL_FP, 0, NULL ); /* no hints */

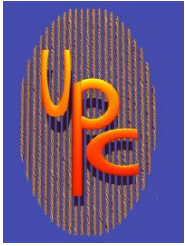
upc_all_fseek( fd, 5*MYTHREAD*sizeof(double), UPC_SEEK_SET );

upc_all_fread_local( fd, buffer, sizeof( double ), 10, 0 );

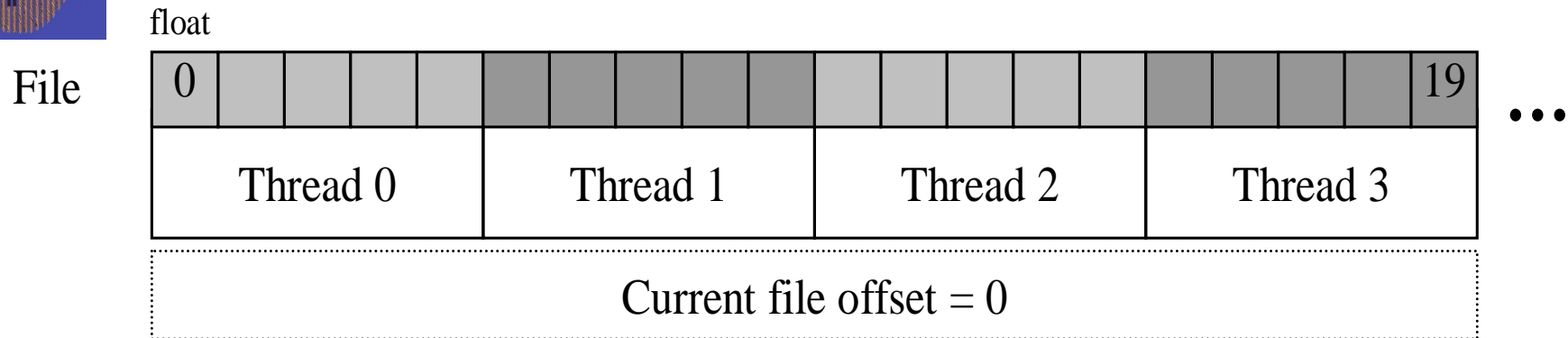
upc_all_fclose(fd);
```

Example 1: Collective read into private
can provide canonical file-view





Basic UPC-IO Examples (cont.)

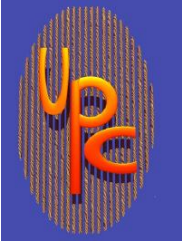


```
shared [5] float buffer[20];
// and assuming a total of 4 static THREADS
...
upc_file_t *fd =
    upc_all_fopen( "file", UPC_RDONLY | UPC_COMMON_FP, 0, NULL );

upc_all_fread_shared( fd, buffer, 5, sizeof(float), 20, 0);
```

Example 2: Read into a blocked shared buffer
can provide a partitioned file-view

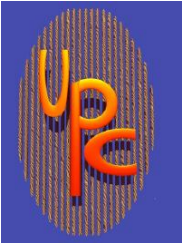




Advanced UPC-IO functions (List-IO)

- List-IO functions are provided for high performance non-contiguous data read/write operations.
- List-IO functions are analogous to MPI-IO file view together with collective operations
- List-IO functions are easy to use
 - Use ***upc_filevec_t*** to specify file chunks with explicit offsets
 - Use ***upc_local_memvec_t*** or ***upc_shared_memvec_t*** to specify private/shared memory chunks



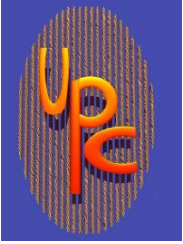


List UPC-IO functions

```
ssize_t upc_all_fread/fwrite_list_local( upc_file_t *fd, size_t memvec_entries,  
                                           upc_local_memvec_t const *memvec,  
                                           size_t filevec_entries,  
                                           upc_filevec_t const *filevec,  
                                           upc_flag_t sync_mode)
```

- Reads/Writes data from/to non-contiguous file positions to/from non-contiguous chunks of private buffer on each thread
 - **File must be open for read/write**
 - **Both *Individual* and *Common* File pointer is ok**
 - **Each thread passes independent list of memory chunks and sizes**
 - **Each thread passes independent list of file chunks and sizes**
- Returns the number of bytes successfully read/written for each thread
 - **File pointers are not updated as a result of list-IO functions**
 - **On error, it returns -1 and sets errno appropriately**



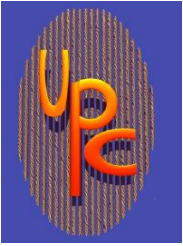


List UPC-IO functions (cont.)

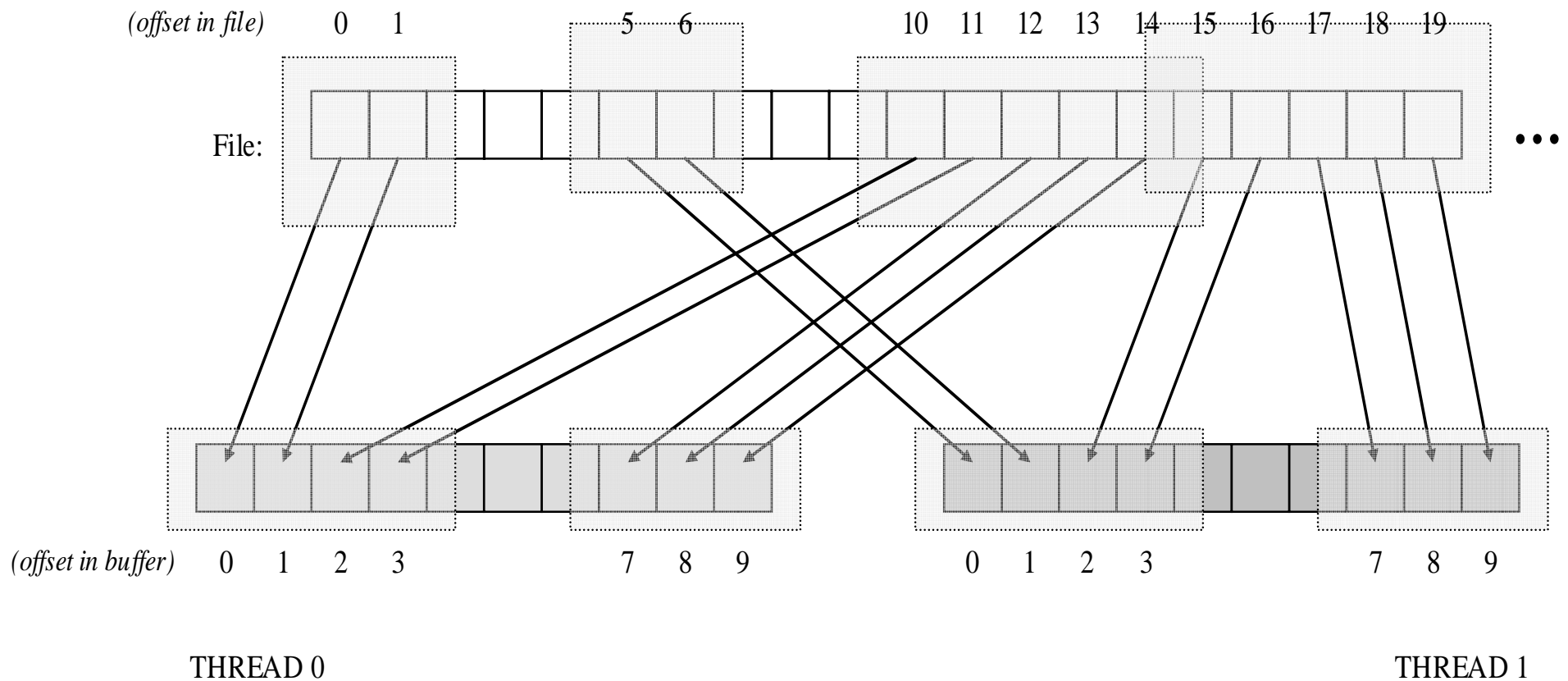
```
ssize_t upc_all_fread/fwrite_list_shared( upc_file_t *fd, size_t memvec_entries,  
                                           upc_shared_memvec_t const *memvec,  
                                           size_t filevec_entries,  
                                           upc_filevec_t const *filevec,  
                                           upc_flag_t sync_mode)
```

- Reads/Writes data from/to non-contiguous file positions to/from non-contiguous chunks of shared buffer on each thread
 - **File must be open for read/write**
 - **Both *Individual* and *Common* File pointer is ok**
 - **Each thread passes independent list of memory chunks and sizes**
 - **The *baseaddr* of each *upc_shared_memvec_t* element must have phase 0**
 - **Each thread passes independent list of file chunks and sizes**
- Returns the number of bytes successfully read/written for each thread
 - **File pointers are not updated as a result of list-IO functions**
 - **On error, it returns -1 and sets errno appropriately**



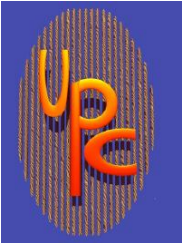


List UPC-IO Examples



Example 3: I/O read of noncontiguous parts of a file to private noncontiguous buffers





```
char buffer[12];
upc_local_memvec_t memvec[2] = { { &buffer[0], 4 }, {
    &buffer[7], 3 } };

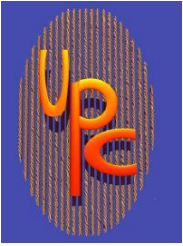
upc_filevec_t filevec[2] = { { MYTHREAD*5, 2 }, {
    10+MYTHREAD*5, 5 } };

upc_file_t *fd =
    upc_all_open( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0,
        NULL);

upc_all_fread_list_local( fd, 2, &memvec, 2, &filevec,
    UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
```

Example 3: I/O read of noncontiguous parts of a file to private noncontiguous buffers



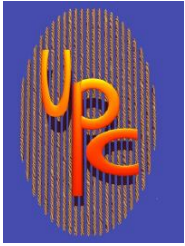


Non-Blocking UPC-IO Functions

```
void upc_all_fread/fwrite_local_async(  
    upc_file_t *fd,  
    void *buffer,  
    size_t size, size_t nmemb,  
    upc_flag_t sync_mode)
```

- Reads/Writes data from a file to/from a private buffer on each thread
 - File must be open for read/write with individual file pointers
 - Each thread passes independent buffers and sizes
 - Number of bytes requested is $\text{size} * \text{nmemb}$ (*may be zero for no-op*)
- Async operations shall be finished by *upc_all_fwait_async* or *upc_all_ftest_async*



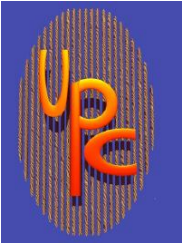


Non-Blocking UPC-IO Functions (cont.)

```
void upc_all_fread/fwrite_shared_async( upc_file_t *fd,  
                                         shared void *buffer,  
                                         size_t blocksize,  
                                         size_t size, size_t nmemb,  
                                         upc_flag_t sync_mode)
```

- Reads/Writes data from a file to/from a shared buffer in memory:
 - **Number of bytes requested is size * nmemb (*may be zero for no-op*)**
 - **Buffer may be an arbitrarily blocked array, but input phase must be 0**
- When file is open for read/write with an *individual* file pointer:
 - **Each thread passes independent buffers and sizes**
- When file is open for read/write with a *common* file pointer:
 - **All threads pass same buffer and sizes (all arguments single-valued)**
- Async operations shall be finished by *upc_all_fwait_async* or *upc_all_ftest_async*



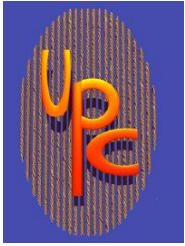


Non-Blocking UPC-IO Functions (cont.)

```
void upc_all_fread/fwrite_list_local_async( upc_file_t *fd, size_t memvec_entries,  
                                             upc_local_memvec_t const *memvec,  
                                             size_t filevec_entries,  
                                             upc_filevec_t const *filevec,  
                                             upc_flag_t sync_mode)
```

- Reads/Writes data from/to non-contiguous file positions to/from non-contiguous chunks of private buffer on each thread
 - **File must be open for read/write**
 - **Both *Individual* and *Common* File pointer is ok**
 - **Each thread passes independent list of memory chunks and sizes**
 - **Each thread passes independent list of file chunks and sizes**
- Async operations shall be finished by `upc_all_fwait_async` or `upc_all_ftest_async`



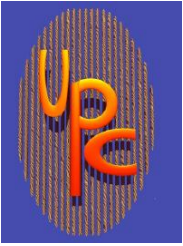


Non-Blocking UPC-IO Functions (cont.)

```
void upc_all_fread/fwrite_list_shared_async(  
    upc_file_t *fd, size_t memvec_entries,  
    upc_shared_memvec_t const *memvec,  
    size_t filevec_entries,  
    upc_filevec_t const *filevec,  
    upc_flag_t sync_mode)
```

- Reads/Writes data from/to non-contiguous file positions to/from non-contiguous chunks of shared buffer on each thread
 - **File must be open for read/write**
 - **Both *Individual* and *Common* File pointer is ok**
 - **Each thread passes independent list of memory chunks and sizes**
 - **The *baseaddr* of each *upc_shared_memvec_t* element must have phase 0**
 - **Each thread passes independent list of file chunks and sizes**
- Async operations shall be finished by *upc_all_fwait_async* or *upc_all_ftest_async*



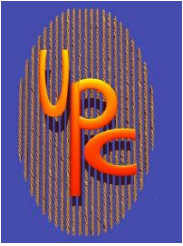


Non-Blocking UPC-IO Functions (cont.)

`upc_off_t upc_all_ftest_async(upc_file_t *fh, int *flag);`

- Non-blocking tests whether the outstanding asynchronous I/O operation associated with *fd* has completed.
- If the operation has completed, the function sets `flag=1` and returns the number of bytes that were read or written. The asynchronous operation becomes no longer outstanding; otherwise it sets `flag=0`.
- On error, it returns `-1` and sets *errno* appropriately, and sets the `flag=1`, and the outstanding asynchronous operation (if any) becomes no longer outstanding.
- It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with *fd*.



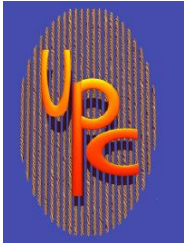


Non-Blocking UPC-IO Functions (cont.)

`upc_off_t upc_all_fwait_async(upc_file_t *fh);`

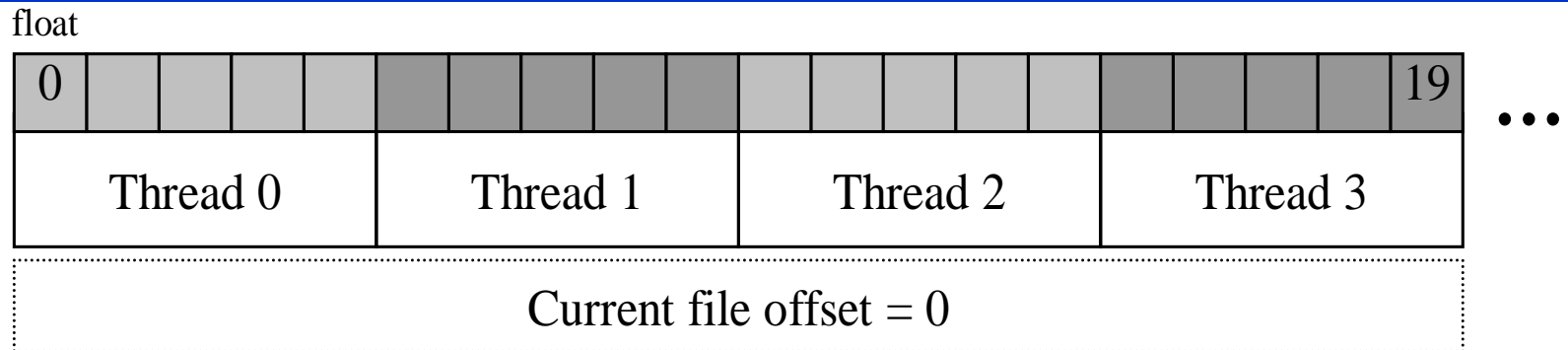
- This function completes the previously issued asynchronous I/O operation on the file handle *fd*, blocking if necessary.
- It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with *fd*.
- On success, the function returns the number of bytes read or written
- On error, it returns `-1` and sets *errno* appropriately, and the outstanding asynchronous operation (if any) becomes no longer outstanding.





Non-Block UPC-IO Examples (cont.)

File

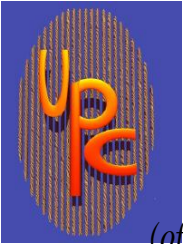


```
shared [5] float buffer[20]; // and assuming a total of 4 static THREADS
int flag; // flag passed to upc_all_ftest_async
...
upc_file_t *fd =
    upc_all_fopen( "file", UPC_RDONLY | UPC_COMMON_FP, 0, NULL );

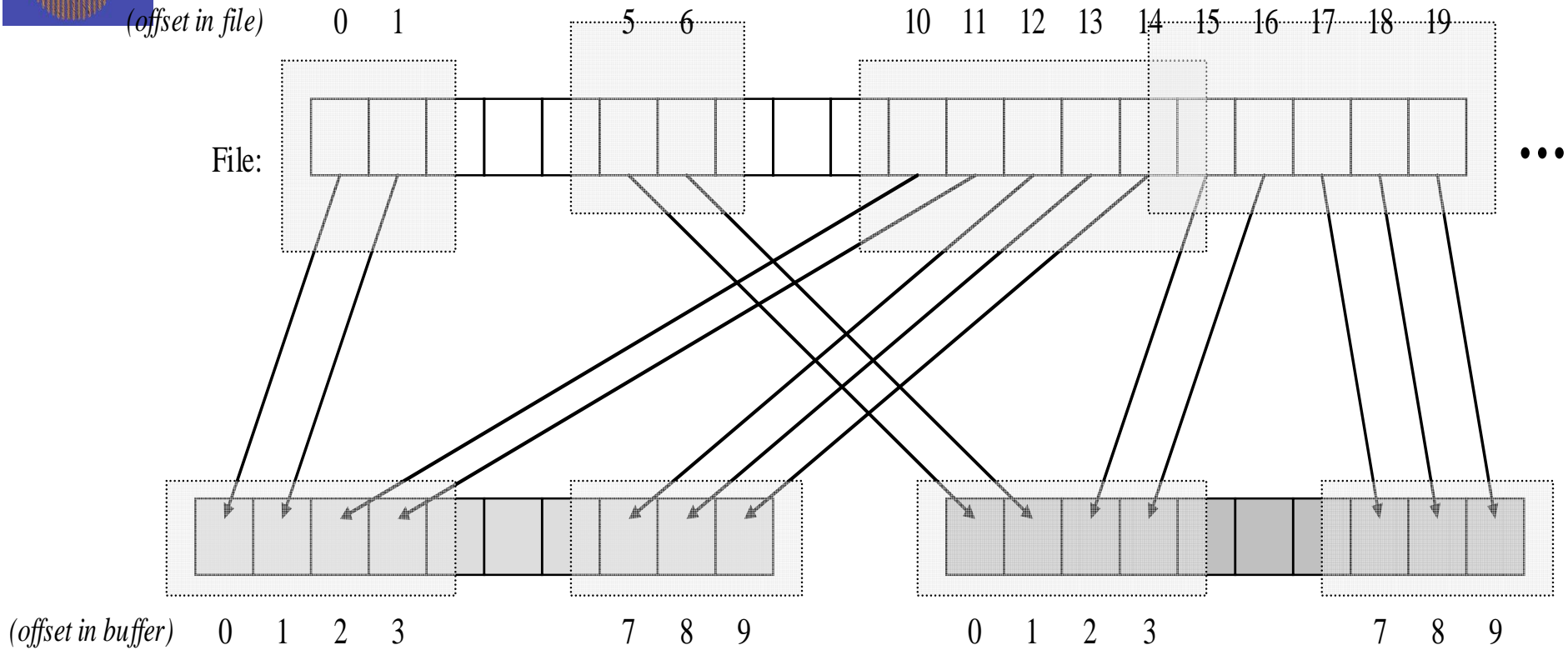
upc_all_fread_shared_async( fd, buffer, 5, sizeof(float), 20, 0);
...
upc_all_ftest_async(fd, &flag);
if( !flag )
    upc_all_fwait_async(fd);
```

Example 5: Non-blocking read into a blocked shared buffer with non-blocking test



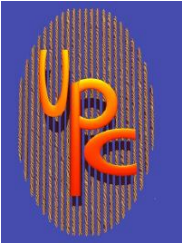


Non-Block UPC-IO Examples (cont.)



THREAD 0
Example 6: Non-blocking I/O read of noncontiguous parts of a file
to private noncontiguous buffers
THREAD 1





```
char buffer[12];
upc_local_memvec_t memvec[2] = { { &buffer[0], 4 }, { &buffer[7], 3 } };

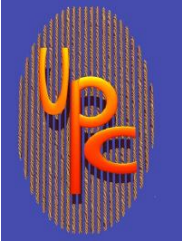
upc_filevec_t filevec[2] = { { MYTHREAD*5, 2 }, { 10+MYTHREAD*5, 5 } };

upc_file_t *fd =
    upc_all_open( "file", UPC_RDONLY | UPC_INDIVIDUAL_FP, 0, NULL);

upc_all_fread_list_local_async( fd, 2, &memvec, 2, &filevec,
                                UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC);
...
upc_all_fwait_async(fd);
```

Example 6: Non-blocking I/O read of noncontiguous parts of a file to private noncontiguous buffers

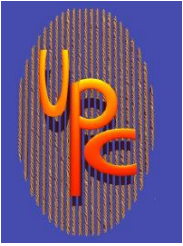




Sec. 4: UPC Applications Development

- Two case studies of application design
 - histogramming
 - locks revisited
 - generalizing the histogram problem
 - programming issues
 - implications of the memory model
 - generic science code (advection):
 - shared multi-dimensional arrays
 - implications of the memory model
- UPC tips, tricks, and traps

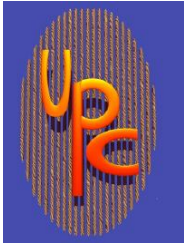




Critical Section without locks

```
#include<stdio.h>
#include<upc.h>
main()
{
    char cmd[80];
    sprintf(cmd,"sleep %d", (THREADS-MYTHREAD)%3);
    system(sleepcmd); // random amount of work
    printf("Threads %2d: ", MYTHREAD);
    system("date");
}
```

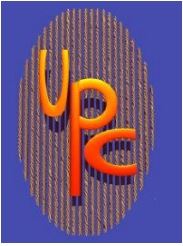




Critical Section without locks

```
Threads 2: Threads 5: Fri Sep 2 04:55:54 EDT 2005
Fri Sep 2 04:52:39 EDT 2005
Threads 1: Threads 4: Fri Sep 2 04:54:38 EDT 2005
Threads 7: Fri Sep 2 04:51:07 EDT 2005
Fri Sep 2 04:53:43 EDT 2005
Threads 0: Fri Sep 2 04:26:35 EDT 2005
Threads 3: Threads 6: Fri Sep 2 04:52:32 EDT 2005
Fri Sep 2 04:56:12 EDT 2005
```



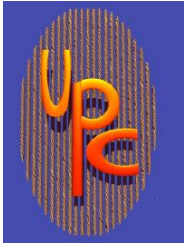


Serializing the Critical Section

```
#include<stdio.h>
#include<upc.h>
main()
{
    int t;
    char cmd[80];
    sprintf(cmd, "sleep %d", (THREADS-MYTHREAD) % 3 );
    system(sleepcmd);

    for(t=0; t<THREADS; t++) {
        upc_barrier;
        if( t != MYTHREAD ) continue;
        printf("Threads %2d: ", MYTHREAD);
        system("date");
    }
}
```

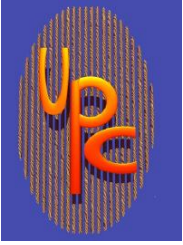




Serializing the Critical Section

```
Threads 0: Fri Sep 2 04:29:13 EDT 2005
Threads 1: Fri Sep 2 05:02:18 EDT 2005
Threads 2: Fri Sep 2 04:58:35 EDT 2005
Threads 3: Fri Sep 2 04:55:10 EDT 2005
Threads 4: Fri Sep 2 04:58:51 EDT 2005
Threads 5: Fri Sep 2 05:11:59 EDT 2005
Threads 6: Fri Sep 2 04:59:05 EDT 2005
Threads 7: Fri Sep 2 05:08:36 EDT 2005
```





Locks Revisited

Syntax and Semantics of locks and lock related functions were covered above

Locks are used to protect critical sections of code

Locks can be used to protect memory references by creating atomic memory operations

Locks need not require global cooperation, so the use of locks can scale with the number of threads. This depends on the precise lock semantics and on the hardware support.





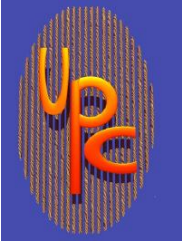
Memory Model Revisited

Syntax and Semantics of 'strict' and 'relaxed' are covered above

A "working" definition is:

- strict references must appear to all threads as if they occur in program order
- relaxed references only have to obey C programming order within a thread





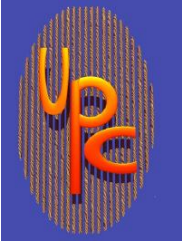
Locks and the Memory Model

If we protect a critical section of the code with a lock, then only one thread can get into the “Critical Section” at a time.

```
upc_lock( lockflag );  
    <<Critical Section of Code>>  
upc_unlock(lockflag );
```

Note, this works because there is an implied *null strict* reference before a `upc_lock` and after a `upc_unlock`. All threads agree on who has the lock.





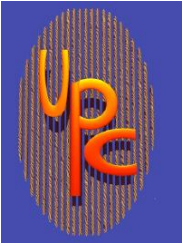
Locking Critical Section

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

upc_lock_t *instr_lock;

main()
{
char sleepcmd[80];
// allocate and initialize lock
instr_lock = upc_all_lock_alloc();
upc_lock_init( instr_lock );
sprintf(sleepcmd, "sleep %d", (THREADS-MYTHREAD) % 3 );
system(sleepcmd); // random amount of work
upc_lock (instr_lock );
    printf("Threads %2d: ", MYTHREAD);
    system("date");
upc_unlock (instr_lock );
}
```

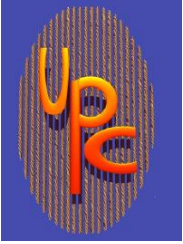




Locking Critical Section

```
Threads 2: Fri Sep 2 05:01:29 EDT 2005
Threads 5: Fri Sep 2 04:59:22 EDT 2005
Threads 7: Fri Sep 2 04:59:32 EDT 2005
Threads 1: Fri Sep 2 05:06:31 EDT 2005
Threads 4: Fri Sep 2 10:31:18 EDT 2005
Threads 0: Fri Sep 2 04:33:27 EDT 2005
Threads 6: Fri Sep 2 04:58:00 EDT 2005
Threads 3: Fri Sep 2 05:02:48 EDT 2005
```

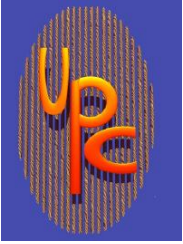




Using Locks to protect memory

- Can't really lock memory
 - No error is reported if you touch “locked memory”
 - You can't stall waiting for memory to be “unlocked”
- Must use the convention that certain shared variables are only referenced inside a locked part of the instruction stream during a certain synchronization phase.





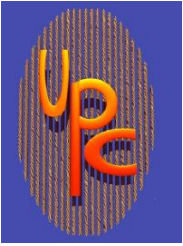
Using Locks to protect memory

For a shared variable `global_sum` and every thread has a private variable `l_sum`, then

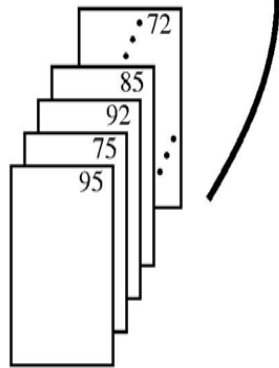
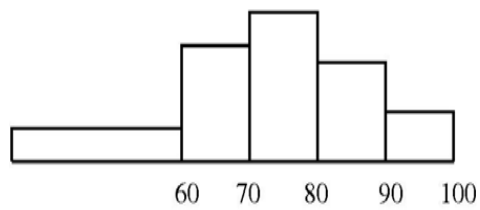
```
upc_lock( reduce_lock );  
    global_sum += l_sum;  
upc_unlock( reduce_lock );
```

computes the reduction of the `l_sum` variables by making access to `global_sum` "atomic".





Histogramming



Collapsing

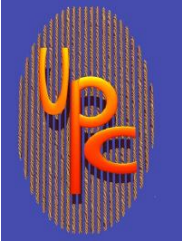
CTAGCTTCG....

k



Expanding

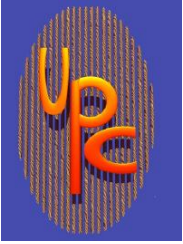




Histogramming

- *Histogramming* is just making a bar chart
- That is, we want count the number of objects in a sample that have certain characteristics
- Might allow a simple operation, for example:
 - counting the number of A, B, C, ... in ones class
 - computing average intensity in a picture
 - HPCS *random access* benchmark
- The computational challenge depend on the parameters:
 - Size of sample space
 - Number of characteristic bins
 - The operator
 - Number of threads



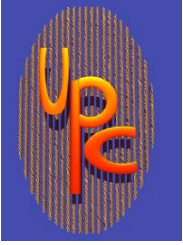


Histogramming

High level pseudo code would look like:

```
foreach sample
  c = characteristic of the sample
  H[c] = H[c] + f(sample);
```



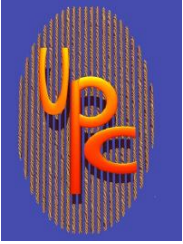


Histogramming

At one extreme histogramming is a reduction

- There is one bin and $f(\text{sample}) = \text{sample}$
- You are trying to collect lot of stuff
- The problem is everybody wants to write to one spot
- This should be done with a reduction technique so that one gets **log** behavior.
- The locking trick works fine and for a small number of threads the preformance is reasonable.



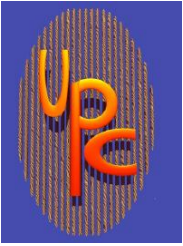


Histogramming

On the other end you have the 'random access' benchmark

- Number of bin is half of core
- samples are random numbers
- c = part of the random number
- $f(\text{sample})$ the identity
- The problem is to spray stuff out across the machine.
- In some versions you are allowed to miss some of the updates.





Programming the Sparse Case

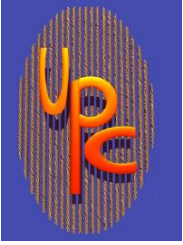
Plan 0: Collect updates locally and then update.

- This is essentially bucket sort. Note this is your only option in a message passing model.
 - You lose the advantage of shared memory,
 - Can't take advantage of the sparseness in the problem.

Plan 1: Use locks to protect each update

Plan 2: Ignore the race condition and see what happens





Using locks to protect each update

```
foreach sample
```

```
    c = characteristic of the sample
```

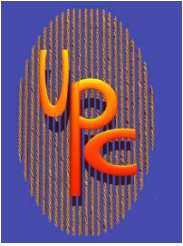
```
    upc_lock( lock4h[c] );
```

```
        H[c] += f(sample);
```

```
    upc_unlock( lock4h[c] );
```

- You don't lose any updates
- With one lock per $H[i]$, you use lots of locks





Using locks to protect each update

To save space try the "hashing the locks trick":

foreach sample

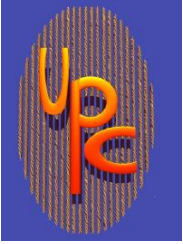
 c = characteristic of the sample

```
    upc_lock( lckprthrd[upc_threadof(H[c])] );
```

```
        H[c] += f(sample);
```

```
    upc_unlock(lckprthrd[upc_threadof(H[c])]);
```





Memory Model Consequences

Plan 0 ignores the memory model

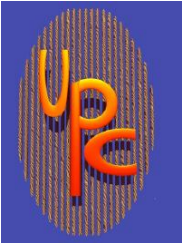
Plan 1 makes everything strict (inherit the lock semantics)

Plan 2 is interesting and machine dependent

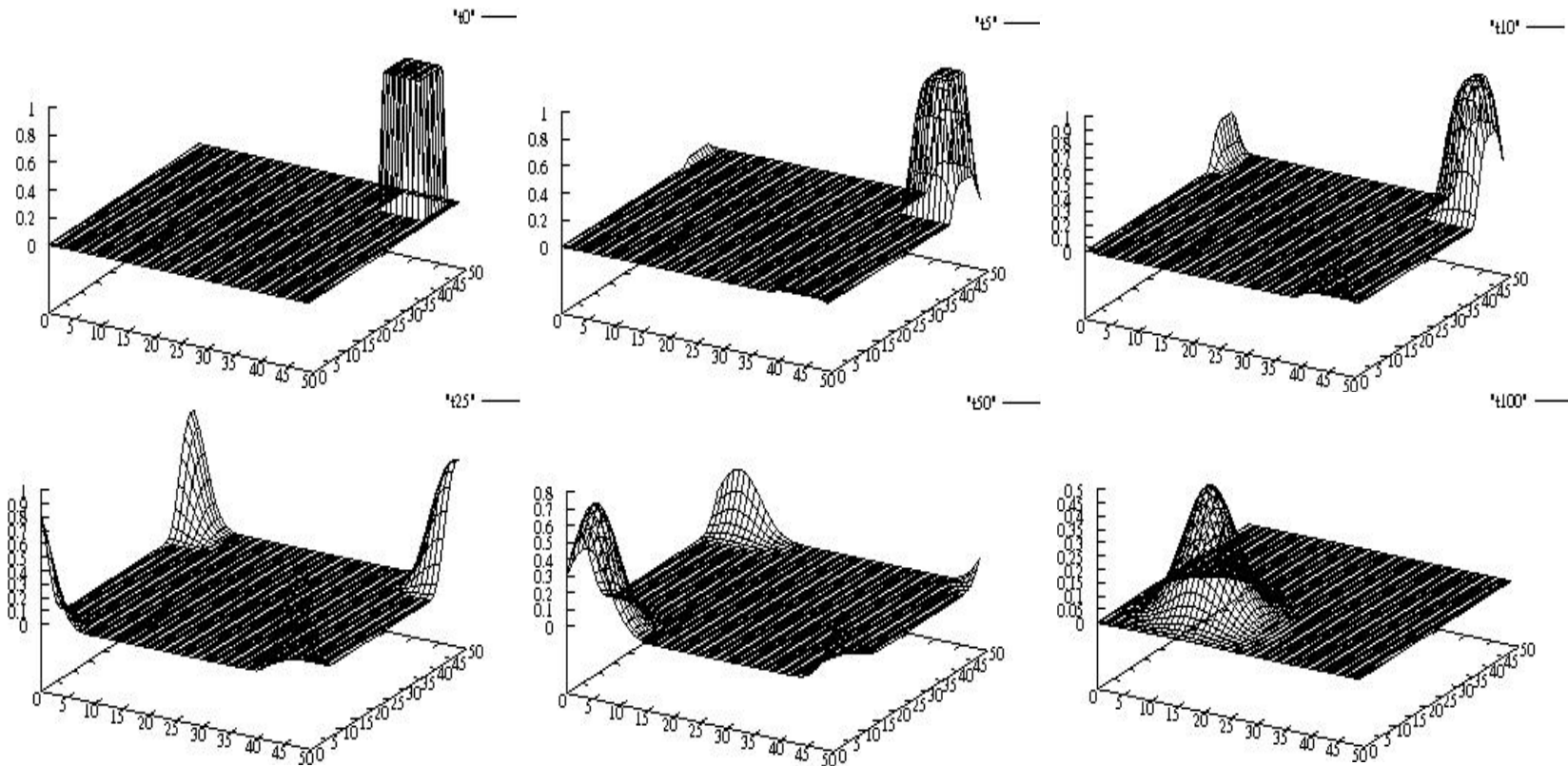
- If the references are strict, you get most of the updates and lose performance enhancing constructs.
- If the references are relaxed, then caching collects the updates.
 - This naturally collects the off affinity writes, but
 - the collected writes have higher contention rate and you miss too many updates.

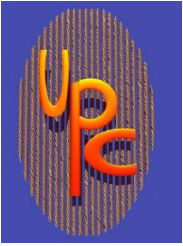
Plan 3 an active area of research





2D Advection



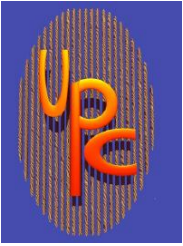


2D Advection

Solve

$$\frac{d\rho}{dt} = \frac{\partial \rho v_x}{\partial x} + \frac{\partial \rho v_y}{\partial y} = 0$$

where $v_x = \frac{\partial \phi}{\partial y}$ and $v_y = -\frac{\partial \phi}{\partial x}$ for a stream function ϕ , on a 2-D uniform mesh with periodic boundary conditions.



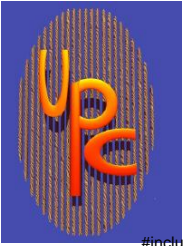
2D Advection

Trick to handle periodic boundary conditions

```
#include<stdio.h>
#include<math.h>
...
double vx[Nx+2][Ny+2], vy[Nx+2][Ny+2];
double rho[Nx+2][Ny+2], flux_x[Nx+2][Ny+2], flux_y[Nx+2][Ny+2];
...
main()
{
  <<initialize_gridpoints and compute dt>>

  for(k=0; k<nsteps; k++) {
    for(i=0; i<=Nx; i++){ for(j=0; j<=Ny; j++){ //flux in x-direction
      vxh = 0.5*(vx[i][j] + vx[i+1][j]);
      flux_x[i][j] = vxh*0.5*(rho[i][j] + rho[i+1][j]) - 0.5*fabs(vxh)*(rho[i+1][j] - rho[i][j]);
    }}
    for(i=0; i<=Nx; i++){ for(j=0; j<=Ny; j++){ //flux in y-direction
      vyh = 0.5*(vy[i][j] + vy[i][j+1]);
      flux_y[i][j] = vyh*0.5*(rho[i][j] + rho[i][j+1]) - 0.5*fabs(vyh)*(rho[i][j+1] - rho[i][j]);
    }}
    for(i=1; i<=Nx;i++){ for(j=1; j<=Ny;j++){ // update pressures
      rho[i][j] = rho[i][j] - fact * ( flux_x[i][j] - flux_x[i-1][j] + flux_y[i][j] - flux_y[i][j-1] );
    }}
    <<fix_boundary_conditions>>
  }
  <<print output>>
}
```



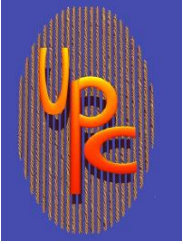


2D Advection

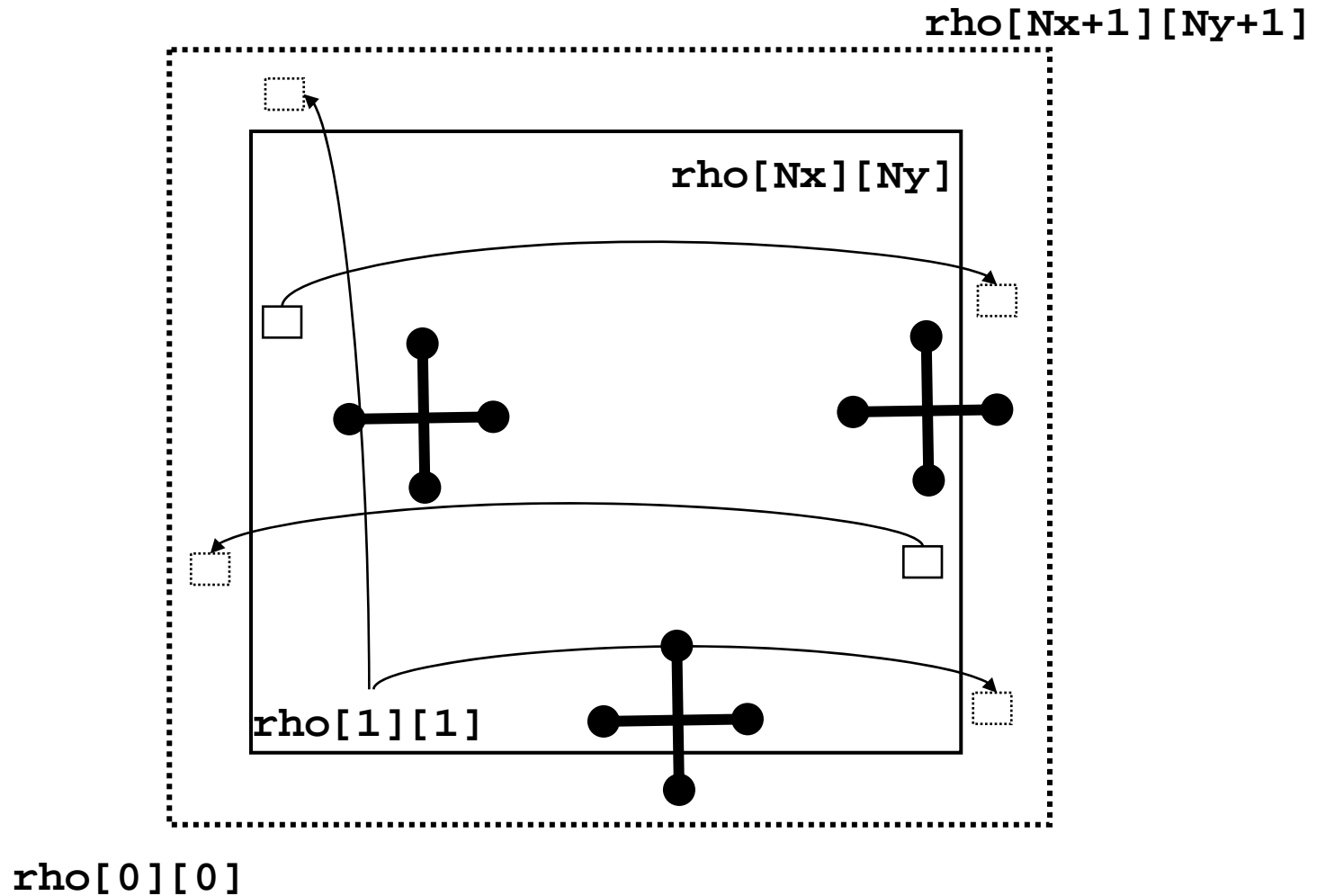
```
#include<stdio.h>
#include<math.h>
...
double vx[Nx+2][Ny+2], vy[Nx+2][Ny+2];
double rho[Nx+2][Ny+2], flux_x[Nx+2][Ny+2], flux_y[Nx+2][Ny+2];
...
main()
{
  <<initialize_gridpoints and compute dt>>

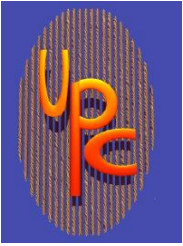
  for(k=0; k<nsteps; k++) {
    for(i=0; i<=Nx; i++){ for(j=0; j<=Ny; j++){ //flux in x-direction
      vxh = 0.5*(vx[i][j] + vx[i+1][j]);
      flux_x[i][j] = vxh*0.5*(rho[i][j] + rho[i+1][j]) -0.5*fabs(vxh)*(rho[i+1][j] - rho[i][j]);
    }
    for(i=0; i<=Nx; i++){ for(j=0; j<=Ny; j++){ //flux in y-direction
      vyh = 0.5*(vy[i][j] + vy[i][j+1]);
      flux_y[i][j] = vyh*0.5*(rho[i][j] + rho[i][j+1]) -0.5*fabs(vyh)*(rho[i][j+1] - rho[i][j]);
    }
    for(i=1; i<=Nx;i++){ for(j=1; j<=Ny;j++){ // update pressures
      rho[i][j] = rho[i][j] - fact * ( flux_x[i][j] - flux_x[i-1][j] + flux_y[i][j] - flux_y[i][j-1] );
    }
    <<fix_boundary_conditions>>
  }
  <<print output>>
}
```



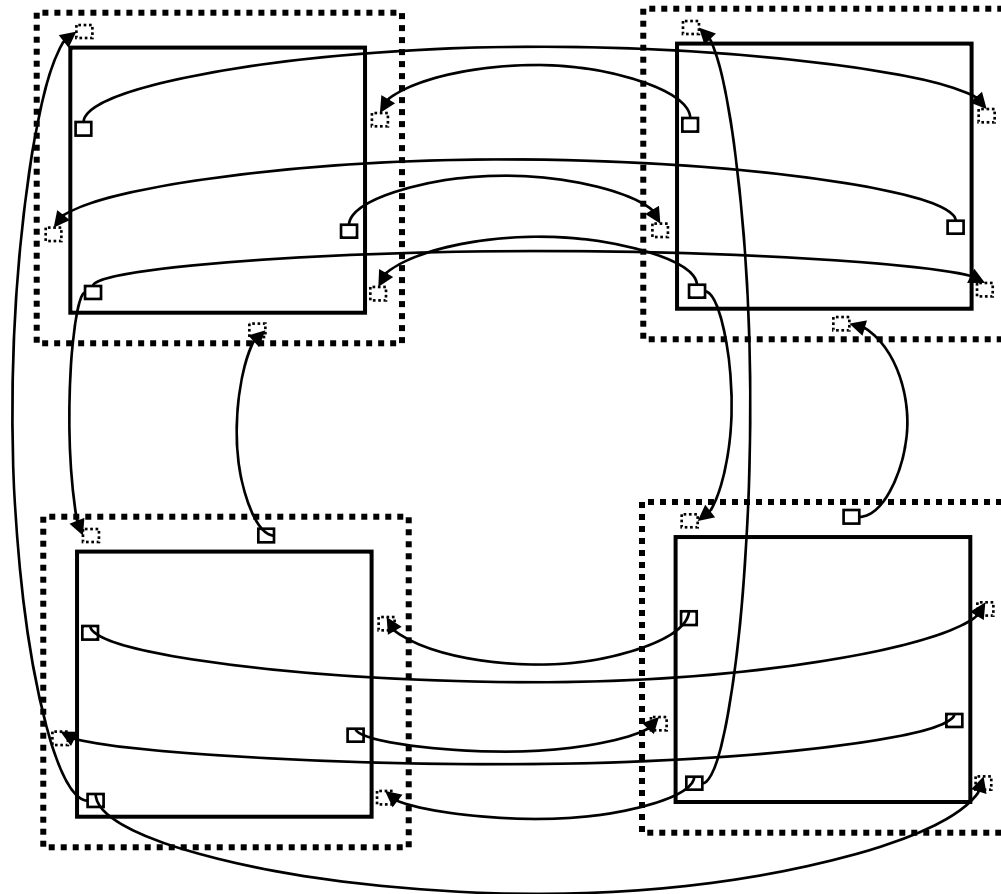


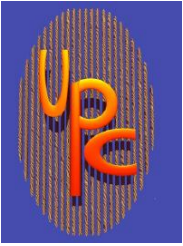
2D Advection (Periodic Boundaries)





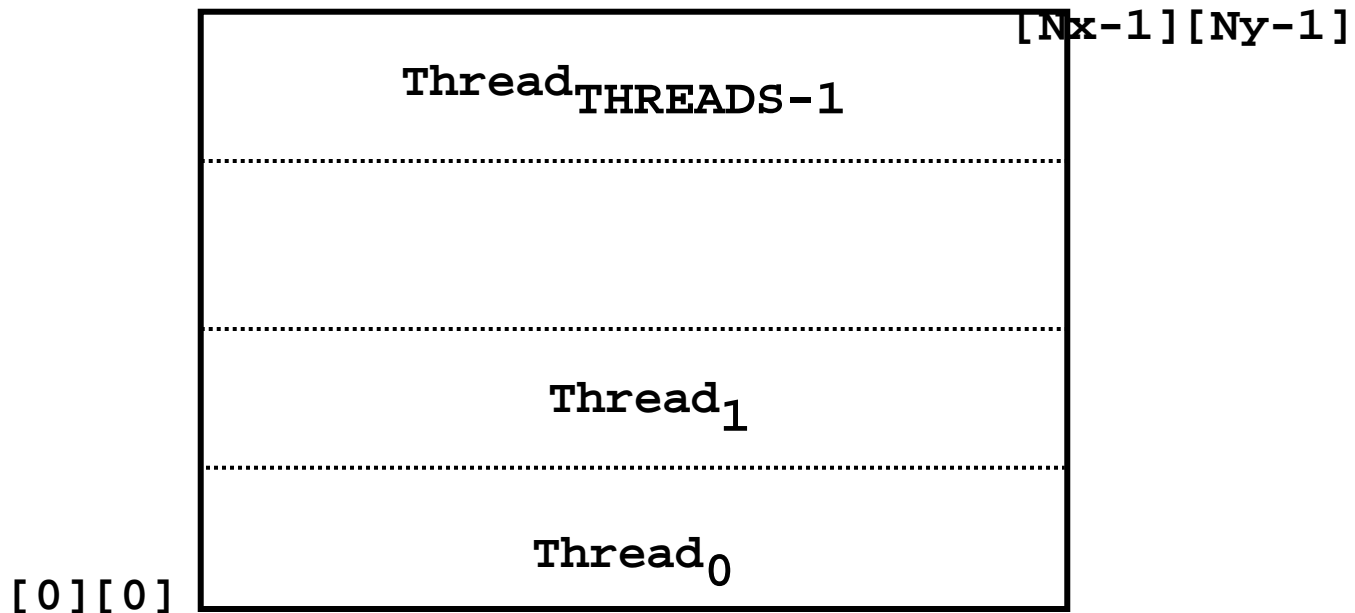
2D Advection MPI style

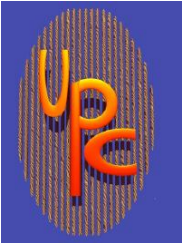




2D Advection UPC version 1

- “Row Blocked” data distribution
- No ghostcells, so we have to handle the periodic boundary conditions with conditionals.



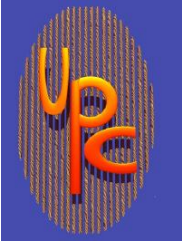


2D Advection UPC version

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

#define Kx 20 // number of rows per block
#define Nx (Kx*THREADS)
shared [Kx*Ny] double vx[Nx][Ny], vy[Nx][Ny];
shared [Kx*Ny] double rho[Nx][Ny];
shared [Kx*Ny] double flux_x[Nx][Ny], flux_y[Nx][Ny];
<<subroutines>>
main()
{
    <<initialize the grid and compute dt>>
    <<MAIN TIMESTEP LOOP>>
    <<print output>>
}
```

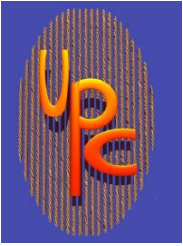




2D Advection UPC version

```
<<MAIN TIMESTEP LOOP>>=  
for(k=0; k<nsteps; k++){  
    <<compute the flux_x>>  
    <<compute the flux_y>>  
    <<update rho>>  
}
```

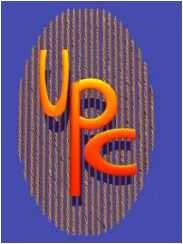




2D Advection UPC version

```
for(k=0; k<nsteps; k++){
  for(li=0; li<Kx; li++){ // compute the flux_x
    i = MYTHREAD*Kx + li;
    ip1 = ((i+1) > Nx -1 ) ? 0 : (i+1);
    for(j=0; j<Ny; j++){
      vxh = 0.5*(vx[i][j] + vx[ip1][j]);
      flux_x[i][j] = vxh*0.5*(rho[i][j] + rho[ip1][j]) -
        0.5*fabs(vxh)*(rho[ip1][j] - rho[i][j]);
    }
  }
  << compute flux_y>>
  upc_barrier;
  << update rho>>
  upc_barrier;
}
```

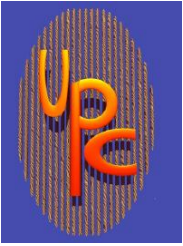




2D Advection UPC version

```
for(k=0; k<nsteps; k++){
  << compute flux_x>>
  for(li=0; li<Kx; li++){ // compute flux_y
    i = MYTHREAD*Kx + li;
    for(j=0; j<Ny-1; j++){
      vyh = 0.5*(vy[i][j]+vy[i][j+1]);
      flux_y[i][j] = vxh*0.5*(rho[i][j] + rho[i][j+1])
                    -0.5*fabs(vyh)*(rho[i][j+1] - rho[i][j]);
    }
    vyh = 0.5*(vy[i][Ny-1]+vy[i][0]) * dy;
    flux_y[i][Ny-1] = vxh*0.5*(rho[i][Ny-1] + rho[i][0])
                    -0.5*fabs(vyh)*(rho[i][0] - rho[i][Ny-1]);
  }
  upc_barrier;
  <<update rho>>
  upc_barrier;
}
```

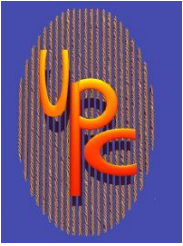




2D Advection UPC version

```
for(k=0; k<nsteps; k++){
  << compute flux_x>>
  << compute flux_y>>
  upc_barrier;
  for(li=0; li<Kx; li++){// update rho
    i = MYTHREAD*Kx + li;
    im1 = ((i-1)<0) ? Nx-1 : (i-1);
    rho[i][0] = rho[i][0] -
      fact*(flux_x[i][0] - flux_x[im1][0] +
            flux_y[i][0] - flux_y[i][Ny-1]);
    for(j=1; j<Ny; j++ ) {
      rho[i][j] = rho[i][j] -
        fact*(flux_x[i][j] - flux_x[im1][j] +
              flux_y[i][j] - flux_y[i][j-1]);
    }
    upc_barrier;
  }
}
```

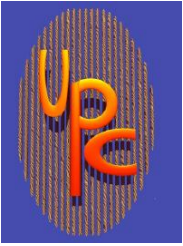




Multi-dimensional Array (Checkerboard distribution)

Consider the matrix:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} \end{bmatrix}$$

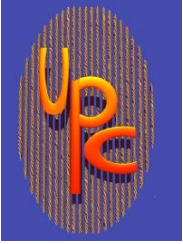


Multi-dimensional Array (Checkerboard distribution)

Consider the matrix:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & | & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & | & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & | & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & | & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & | & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & | & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\ \hline a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & | & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & | & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & | & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} \end{bmatrix}$$

distribute it “onto” 6 threads



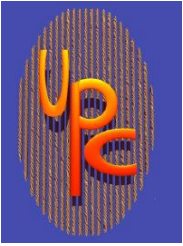
Multi-dimensional Array (Checkerboard distribution)

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

#define N 3
#define M 4
shared int A[N][M][THREADS];
main()
{
    int k,i,j,t;

    if( MYTHREAD == 0 ){
        for(k=0; k<N*M*THREADS; k++) {
            *(&A[0][0][0]+k) = 10000*(k/(THREADS*M))
                + 100*((k/THREADS)%M) + (k%THREADS);
        }
        for(i=0; i<N; i++) { for(j=0; j<M; j++) {
            for(t=0; t<THREADS; t++)
                printf(" %06d", A[i][j][t]);
        } } printf("\n");
    }
    upc_barrier;
    <rest of program>
}
```

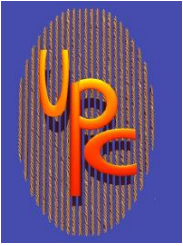




Multi-dimensional Array (Checkerboard distribution)

```
000000 000001 000002 000003 000004 000005
000100 000101 000102 000103 000104 000105
000200 000201 000202 000203 000204 000205
000300 000301 000302 000303 000304 000305
010000 010001 010002 010003 010004 010005
010100 010101 010102 010103 010104 010105
010200 010201 010202 010203 010204 010205
010300 010301 010302 010303 010304 010305
020000 020001 020002 020003 020004 020005
020100 020101 020102 020103 020104 020105
020200 020201 020202 020203 020204 020205
020300 020301 020302 020303 020304 020305
```





Multi-dimensional Array (Checkerboard distribution)

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

#define N 3
#define M 4
shared int A[N][M][THREADS];
main()
{
    int k,i,j,t;
    int *B[N];

    if( MYTHREAD == 0 ){
        for(k=0; k<N*M*THREADS; k++) {
            *(&A[0][0][0]+k) = 10000*(k/(THREADS*M)) + 100*((k/THREADS)%M) + (k%THREADS);
        }
        for(i=0; i<N; i++) { for(j=0; j<M; j++) {
            for(t=0; t<THREADS; t++)
                printf(" %06d", A[i][j][t]);
            } } printf("\n");
        }
        upc_barrier;

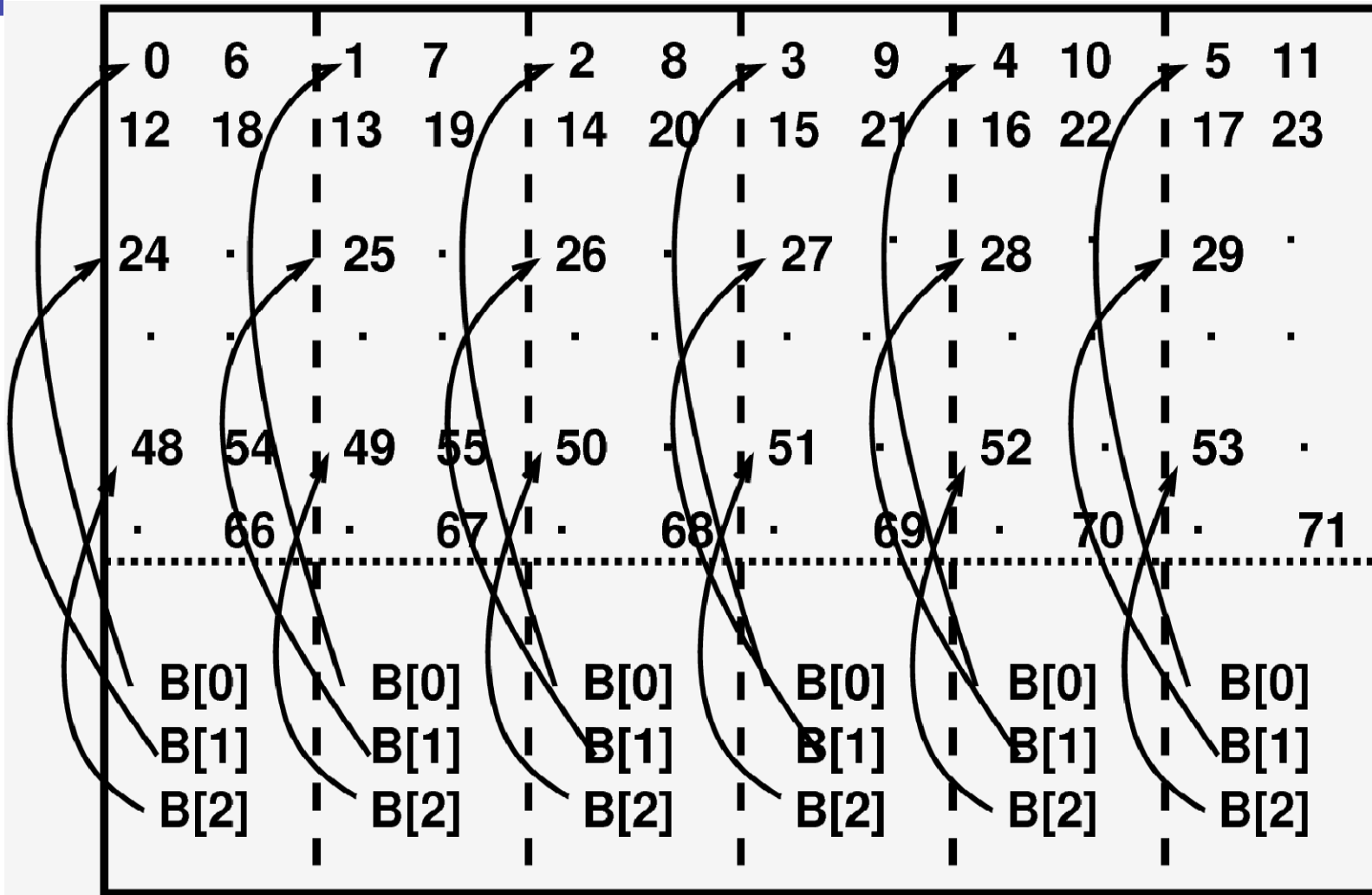
        for(i=0; i< N; i++)
            B[i] = (int *)&A[i][0][MYTHREAD];

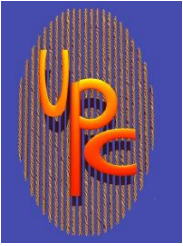
        <rest of program>
    }
}
```





Multi-dimensional Array (Checkerboard distribution)



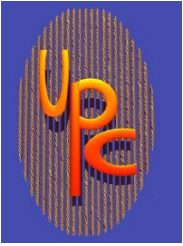


Multi-dimensional Array (Checkerboard distribution)

```
for(i=0; i< N; i++)
    B[i] = (int *)&A[i][0][MYTHREAD];

// Can work with B[][] locally
for( t=0; t< THREADS; t++ ){
    upc_barrier;
    if( t != MYTHREAD) continue;
    printf("In THREAD %d, B[][] is\n", MYTHREAD);
    for(i=0; i<N; i++) {
        for(j=0; j<M; j++)
            printf("  %06d", B[i][j]);
        printf("\n");
    }
}
```





Multi-dimensional Array (Checkerboard distribution)

In THREAD 0, B[][] is

```
000000 000100 000200 000300
010000 010100 010200 010300
020000 020100 020200 020300
```

In THREAD 1, B[][] is

```
000001 000101 000201 000301
010001 010101 010201 010301
020001 020101 020201 020301
```

In THREAD 2, B[][] is

```
000002 000102 000202 000302
010002 010102 010202 010302
020002 020102 020202 020302
```

In THREAD 3, B[][] is

```
000003 000103 000203 000303
010003 010103 010203 010303
020003 020103 020203 020303
```

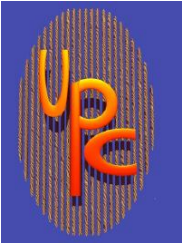
In THREAD 4, B[][] is

```
000004 000104 000204 000304
010004 010104 010204 010304
020004 020104 020204 020304
```

In THREAD 5, B[][] is

```
000005 000105 000205 000305
010005 010105 010205 010305
020005 020105 020205 020305
```





Multi-dimensional Array (Checkerboard distribution)

Now we can match the 'local' arrays $B[][]$ with any subblocks we want.

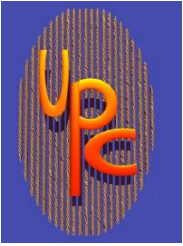
In thread 0:

$$B[][] = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

In thread 1:

$$B[][] = \begin{bmatrix} a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \end{bmatrix}$$





Multi-dimensional Array (Checkerboard distribution)

$$T_0 : B] [] = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

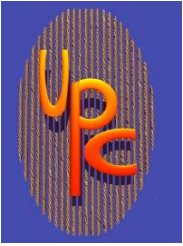
$$T_1 : B [] [] = \begin{bmatrix} a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \end{bmatrix}$$

$$T_2 : B] [] = \begin{bmatrix} a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} \end{bmatrix}$$

$$T_3 : B [] [] = \begin{bmatrix} a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \end{bmatrix}$$

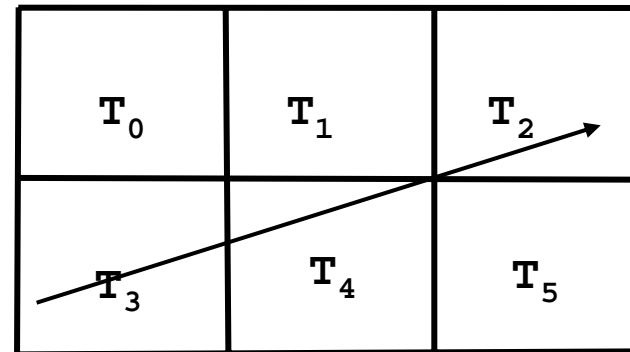
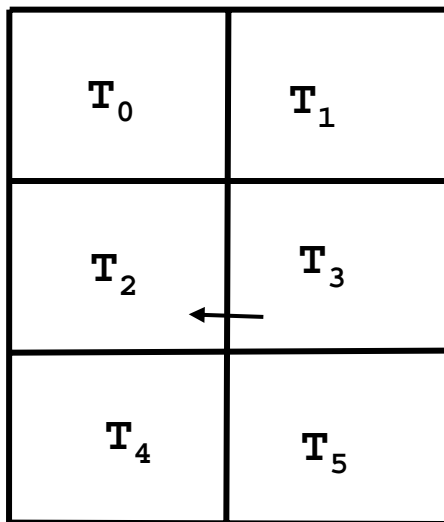
$$T_4 : B] [] = \begin{bmatrix} a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} \end{bmatrix}$$

$$T_5 : B [] [] = \begin{bmatrix} a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \\ a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} \end{bmatrix}$$

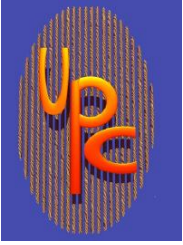


Multi-dimensional Array (Checkerboard distribution)

You can arrange the $B[][]$ blocks any way you want.



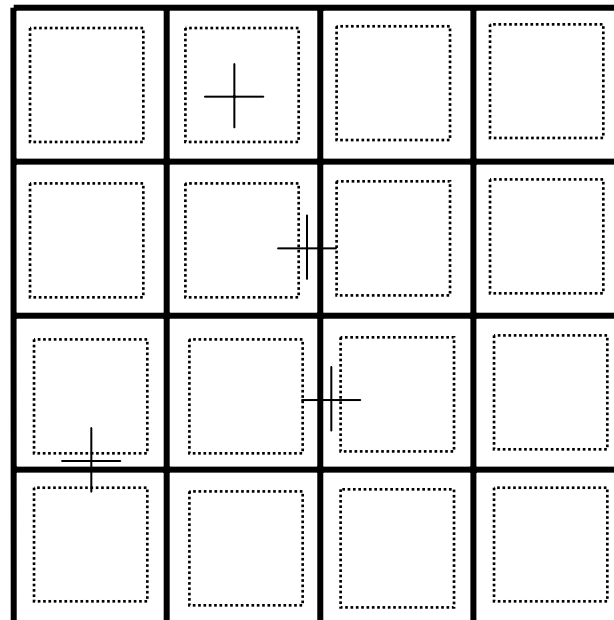
$$A[3][4][2] = A[3][0][3];$$

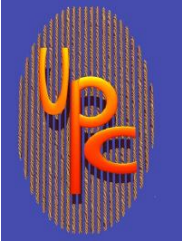


Memory Model Consequences

The relaxed memory model enables ILP optimizations (and this is what compilers are good at).

Consider a typical grid based code on shared data:





Memory Model Consequences

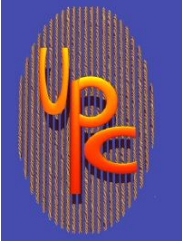
Say the thread code looks like:

```
for (i=0; . . . . .)
  for (j=0; . . . . .)
    gridpt[i][j] = F(neighborhood)
```

If the references are strict:

- You can't unroll loops
- You can't move loop invariants
- You can't remove dead code



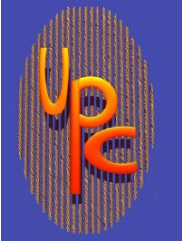


Memory Model Consequences

Say the thread code looks like:

```
for (i=0; . . . . .)
  for (j=0; . . . . .)
    a[i][j] = F(neighborhood)
for (j=0; . . . . .)
  b[i][j] = F(neighborhood)
```

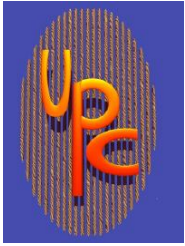




Memory Model Consequences

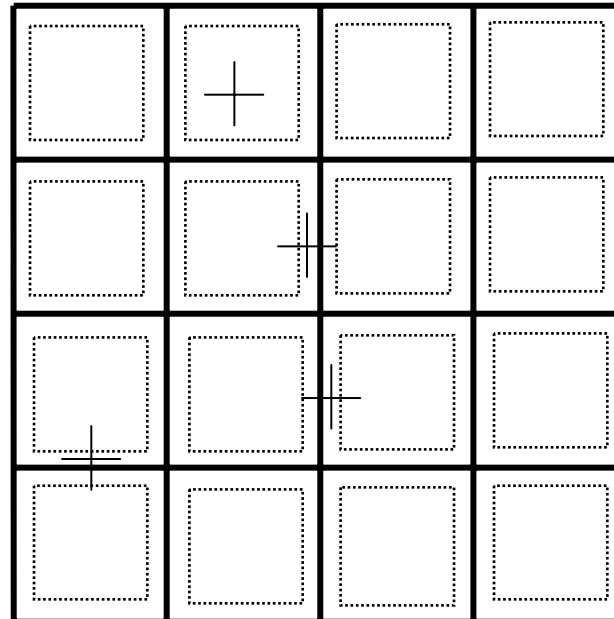
And for performance reasons you want
(the compiler) to transform the code to:

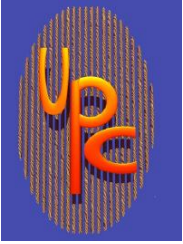
```
for(i=0;.....)
  for(j=0;.....){
    a[i][j] = F(neighborhood)
    b[i][j] = F(neighborhood)
  }
```



Memory Model Consequences

The relaxed memory enables natural parallelization constructs.
Consider a typical grid based code on shared data:





Memory Model Consequences

Now say the thread code looks like:

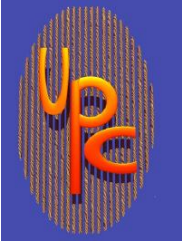
```
for(i=0;.....)
  for(j=0;.....){
    a[i][j] = F(neighborhood)
    b[i][j] = F(neighborhood)
  }
```

The strict model say that the write have to occur:

a[0][0], b[0][0], a[0][1], b[0][1], a[0][2], b[0][2],

So you can't delay the writes so that you can send a buffer's worth of **a**'s and buffer's worth of **b**'s.



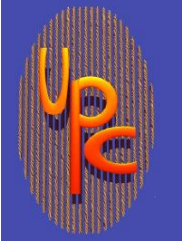


Memory Model Consequences

The relaxed model enables all standard ILP optimizations on a per thread basis.

The relaxed model enables bulk shared memory references. This can occur either by caching strategies or smart runtime systems.





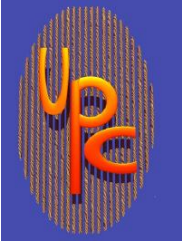
UPC tips, traps, and tricks

This section covers some detailed issues in UPC programming. Many of these arise from UPC's definition of array blocking and its representation of pointers to shared objects.

These observations may be particularly useful to library designers and implementers since they often write “generic” code.

High performance and high productivity (don't repeat the same mistakes) are covered here.





Ex. 1: Typedefs in shared objects

- Consider:

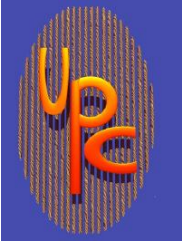
```
typedef double BlockType[10][10];  
shared BlockType A[THREADS];
```
- What is the size of `A[0]`?

```
sizeof(A[0]) == 100*sizeof(double)?
```
- If `THREADS > 1`, then `A[0][0][1]` is on thread 1.
- The correct answer is:

```
sizeof(A[0]) == sizeof(double)
```

(cf: 6.5.2.1.9)

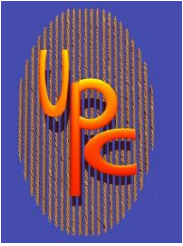




Ex. 1: Solution

```
typedef struct  
    t_{double data[10][10]} BlockType;  
shared BlockType A[THREADS];
```





Ex. 2: Does [*] guarantee uniform data distribution?

(a) Let `THREADS==4`. Consider:

```
shared [ * ] double A[9];
```

- How many elements does thread 3 contain?

```
(sizeof(A)/upc_elemsizeof(A) + THREADS-1) / THREADS
```

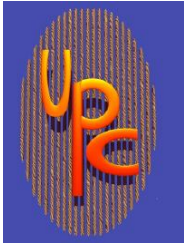
- Thread 3 contains 0 elements.

(b) Consider:

```
shared [ * ] double A[5];
```

- The elements are distributed: 2 2 1 0
- You might prefer the distribution: 2 1 1 1





Ex. 2: Does [*] guarantee good data distribution?

(a) Suppose `THREADS==4`. Consider:

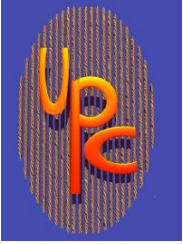
```
shared [ * ] double A[9];
```

- How many elements does thread 3 contain?

Answer: 0. The elements are distributed: 3 3 3 0.

```
(sizeof(A)/upc_elemsizeof(A) + THREADS-1) / THREADS
```





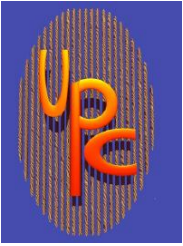
A larger example of [*]

(c) Consider:

```
shared [ * ] double B[135];
```

- On 8 threads each one has 16 or 17 elements.
- On 128 threads the first 67 threads contain 2 elements, thread 67 contains 1 element, and the remaining 67 threads contain 0 elements.
- The underlying addressing scheme does not permit the intuitively appealing data distribution.





Ex. 3: A generic array copy function

- We want to write a library function `UPC_AryCpy` to copy shared array `src` to shared array `Dst`:

```
shared [SrcBlkSize] TYPE Src[N];
```

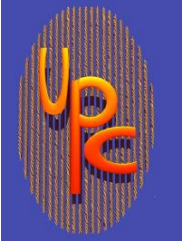
```
shared [DstBlkSize] TYPE Dst[N];
```

- If the block size and type are known at compile time, this is as simple as:

```
upc_forall (i=0; i<N; i++; &Dst[i])
```

```
    Dst[i] = Src[i];
```





Arguments needed for a generic fn

- A generic version of this function requires the following arguments:

```
UPC_AryCpy(  
  shared void * Dst, Src,  
  size_t DstBlkSize, SrcBlkSize,  
  size_t DstElemSize, SrcElemSize,  
  size_t N);
```

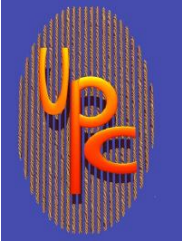




Motivation

- This example was motivated by experience implementing generic reduction and sorting functions.
- The central problem is to compute the address of an arbitrary array element at run time.
- A key aspect of this problem is that the block size of the array is not known until run time.

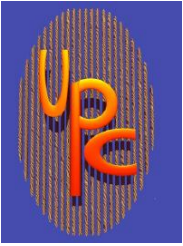




UPC review

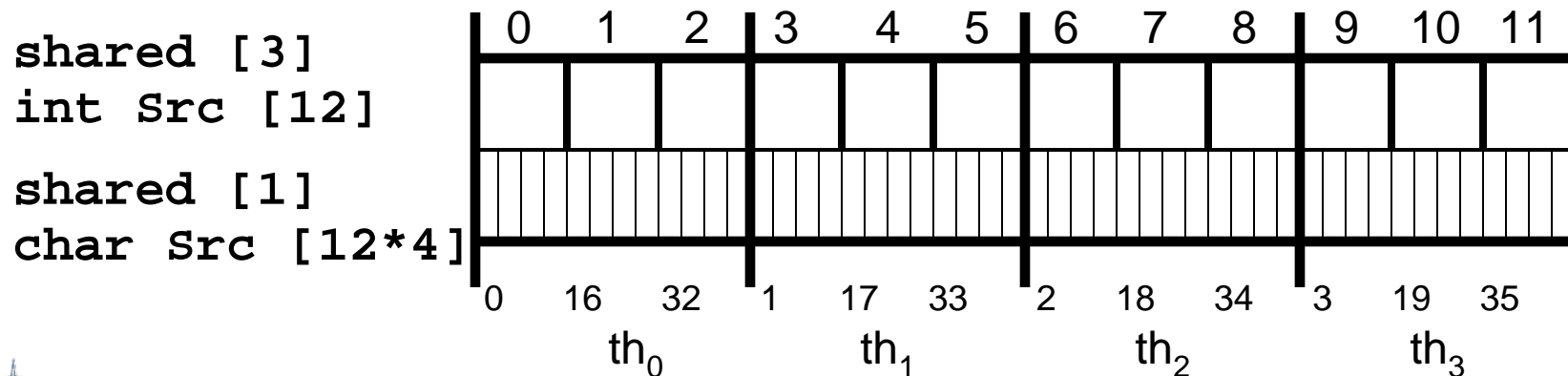
- UPC features relevant to this problem are:
 - Block size is part of the type of pointer-to-shared.
 - The compiler uses the block size to generate address arithmetic at compile time, but the programmer must generate the address arithmetic if the block size is not known until run time.
- The generic pointer-to-shared is
shared void *
 - The block size of a generic pointer-to-shared is 1.
 - The element size of such a pointer is also 1 (byte).

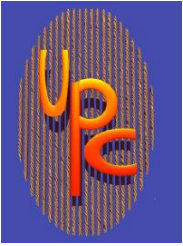




Mapping to a generic shared array

- The copy operation requires a mapping between
`shared [SrcBlkSize] TYPE A[N] ← →`
`shared [1] char Src[N*sizeof(TYPE)]`
- Assume that `src` (*i.e.*, `&Src[0]`) has phase 0 and affinity to thread 0.





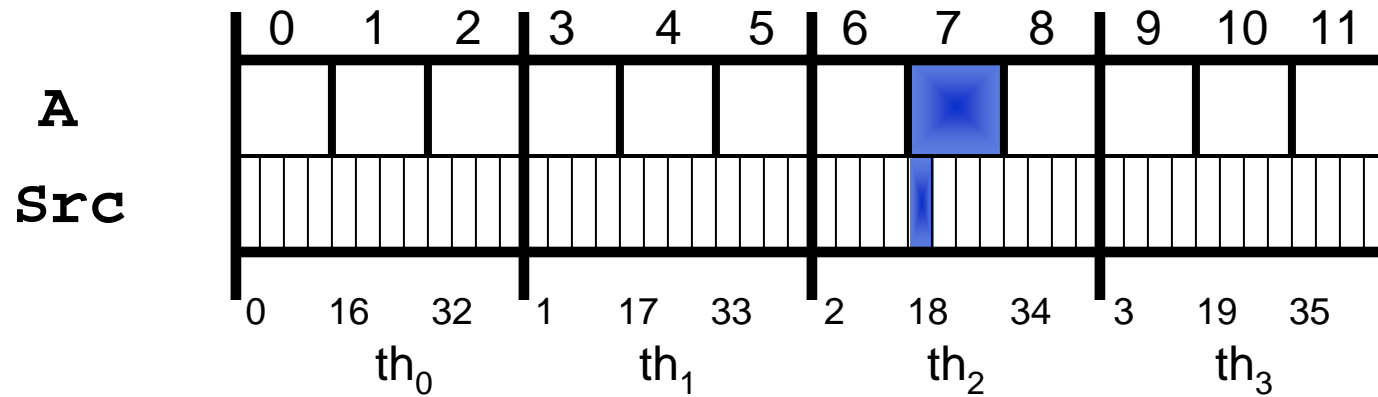
A simple case of mapping to generic (animation)

- For example, in a simple case we might have

```
shared [3] int A[12]
```

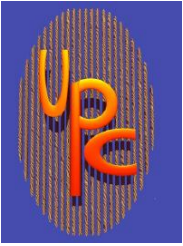
which is mapped to the function argument

```
shared void *Src
```



- So, `A[7]` corresponds to `Src+18`.



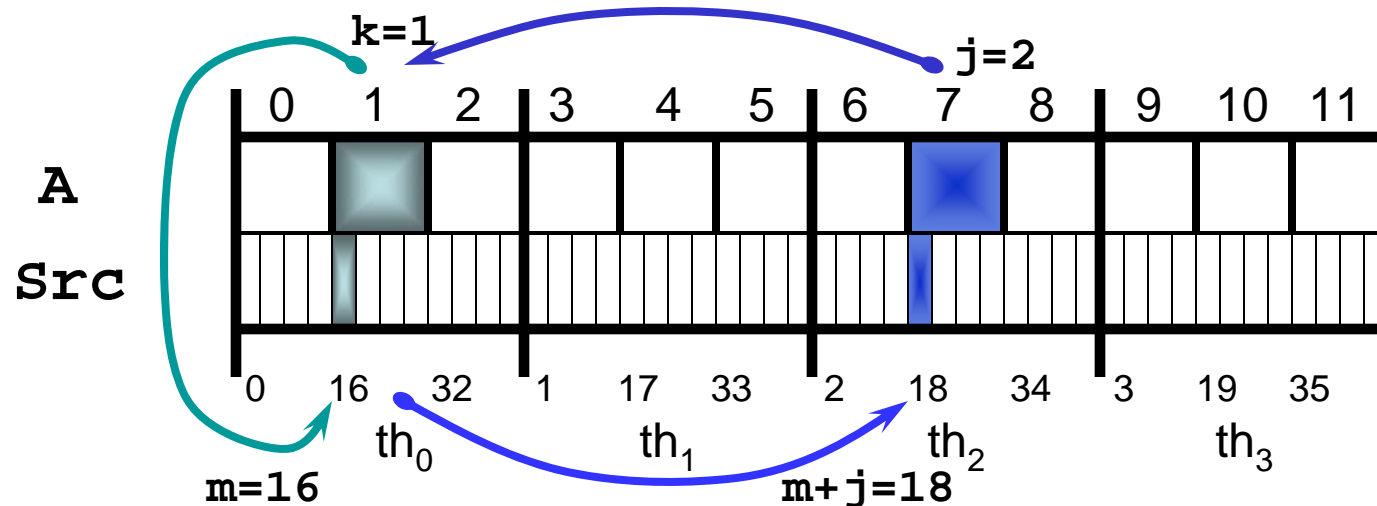


How to compute the mapping

(animation)

- Also assume $N \leq \text{srcBlkSize} * \text{THREADS}$.

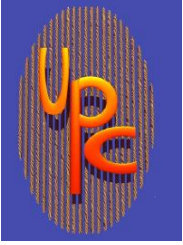
Determine the element of `src` that maps to `A[i]`, $i=7$.



The corresponding element of the source is $th_2 = 18$.

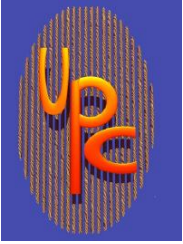
$$m = (k \% \text{srcBlkSize}) * \text{blkElemSize} * \text{THREADS} = 16.$$





(For hardcopy readers)

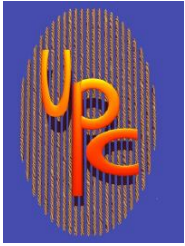
- The animated text at the bottom of the previous slide is: Let $i=7$.
 - (a) $A[i]$ is on thread $j = i / \text{SrcBlkSize} = 2$.
 - (b) The corresponding element of A on thread 0 is
 $k = i - j * \text{SrcBlkSize} = 1$.
 - (c) The corresponding element of src is
 $m = (k \% \text{SrcBlkSize}) * \text{SrcElemSize} * \text{THREADS} = 16$.
 - (d) The desired element of src is at offset $m+j = 18$.



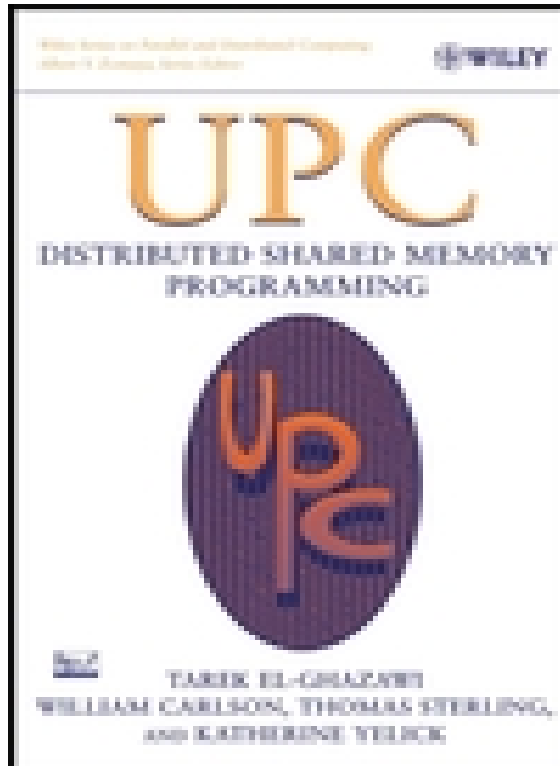
Thank you

- Enjoy UPC!

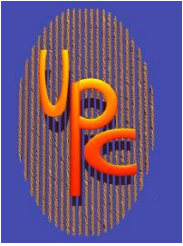




UPC textbook now available



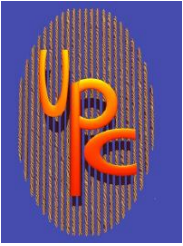
- *UPC: Distributed Shared Memory Programming*
Tarek El-Ghazawi
William Carlson
Thomas Sterling
Katherine Yelick
- Wiley, May, 2005
- ISBN: 0-471-22048-5



UPC v1.2

- Specification document
 - http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf
 - http://upc.nersc.gov/docs/user/upc_spec_1.2.pdf
- UPC Manual
 - http://upc.gwu.edu/downloads/Manual-v1_0.pdf
- HP UPC Programmers Guide
 - <http://h30097.www3.hp.com/upc/upcus.htm>

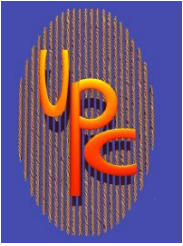




Collected URLs

- **UPC** <http://www.upcworld.org>
- **GWU** <http://www.upc.gwu.edu>
- **Cray** <http://docs.cray.com> (search for UPC)
- **HP** <http://h30097.www3.hp.com/upc>
- **Berkeley** <http://upc.nersc.gov>
- **Intrepid** <http://www.intrepid.com/upc>
- **U. Florida** <http://www.hcs.ufl.edu/~leko/upctoolint/>
- **MTU** <http://www.upc.mtu.edu>
- **Totalview** <http://www.etnus.com/TotalView/>
<http://upc.lbl.gov/docs/user/totalview.html>
- **Trace** http://upc.nersc.gov/docs/user/upc_trace.html

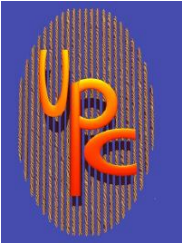




References

- El-Ghazawi, T., K. Yelick, W. Carlson, T. Sterling, *UPC: Distributed Shared-Memory Programming*, Wiley, 2005.
- Coarfa, C., Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanty, Y. Yao, An evaluation of global address space languages: Co-array Fortran and Unified Parallel C, PPOPP2005.
- El-Ghazawi, T., F. Cantonnet, Y. Yao, J. Vetter, Evaluation of UPC on the Cray X1, Cray Users Group, 2005.
- Chen, W., C. Iancu, K. Yelick, Communication optimizations for fine-grained UPC applications, PACT 2005.
- Zhang, Z., S. Seidel, Performance benchmarks of current UPC systems, IPDPS 2005 PMEOWorkshop.

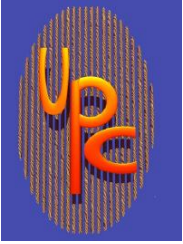




References

- Bell, C., W. Chen, D. Bonachea, K. Yelick, Evaluating support for global address space languages on the Cray X1, ICS 2004.
- Cantonnet, F., Y. Yao, M. Zahran, T. El-Ghazawi, Productivity analysis of the UPC language, IPDPS 2004 PMEOW Workshop.
- Kuchera, W., C. Wallace, The UPC memory model: Problems and prospects, IPDPS 2004.
- Su, H., B. Gordon, S. Oral, A. George, SCI networking for shared-memory computing in UPC: Blueprints of the GASNet SCI conduit, LCN 2004.





Bibliography

- Cantonnet, F., Y. Yao, S. Annareddy, AS. Mohamed, T. El-Ghazawi, Performance monitoring and evaluation of a UPC implementation on a NUMA architecture, IPDPS 2003.
- Chen, W., D. Bonachea, J. Duell, P. Husbands, C. Iancu, K. Yelick, A Performance analysis of the Berkeley UPC compiler, ICS 2003.
- Cantonnet, F., T. El-Ghazawi, UPC performance and potential: A NPB experimental study, SC 2002.
- El-Ghazawi, T., S. Chauvin, UPC benchmarking issues, ICPP 2001.

