



**PGAS09**

# **Unified Parallel C – UPC Tutorial**

**Tarek El-Ghazawi**  
[tarek@gwu.edu](mailto:tarek@gwu.edu)

**The George Washington University**



1

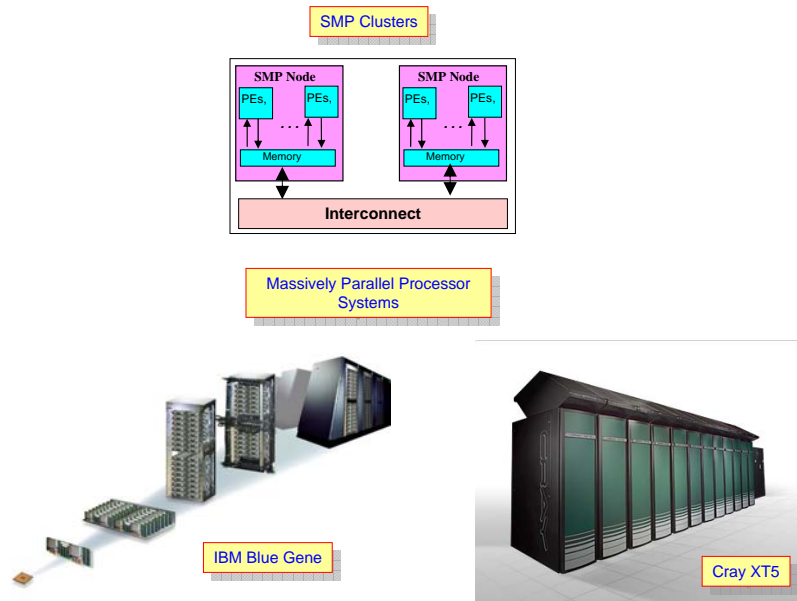
## **Overview**

- A. Introduction to PGAS**
- B. Unified Parallel C - UPC**
- C. Discussion**

2

# A. Introduction to PGAS

## The common architectural landscape



## Programming Models

- ◆ **What is a programming model?**
  - 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

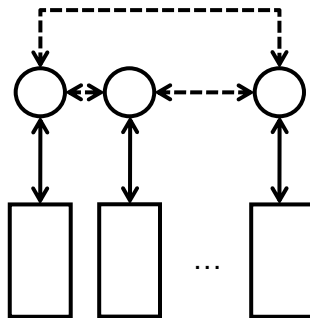
5

## Examples of Parallel Programming Models

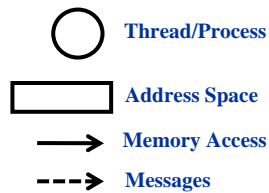
- ◆ Message Passing
- ◆ Shared Memory (Global Address Space)
- ◆ Partitioned Global Address Space (PGAS)

6

## The Message Passing Model



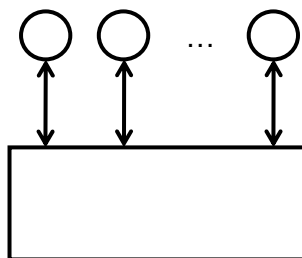
### Legend



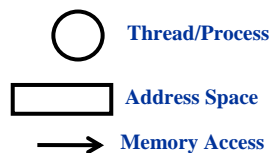
- ◆ Concurrent sequential processes
- ◆ Explicit communication, two-sided
- ◆ Library-based
- ◆ Positive:
  - Programmers control data and work distribution.
- ◆ Negative:
  - Significant communication overhead for small transactions
  - Excessive buffering
  - Hard to program in
- ◆ Example: MPI

7

## The Shared Memory Model



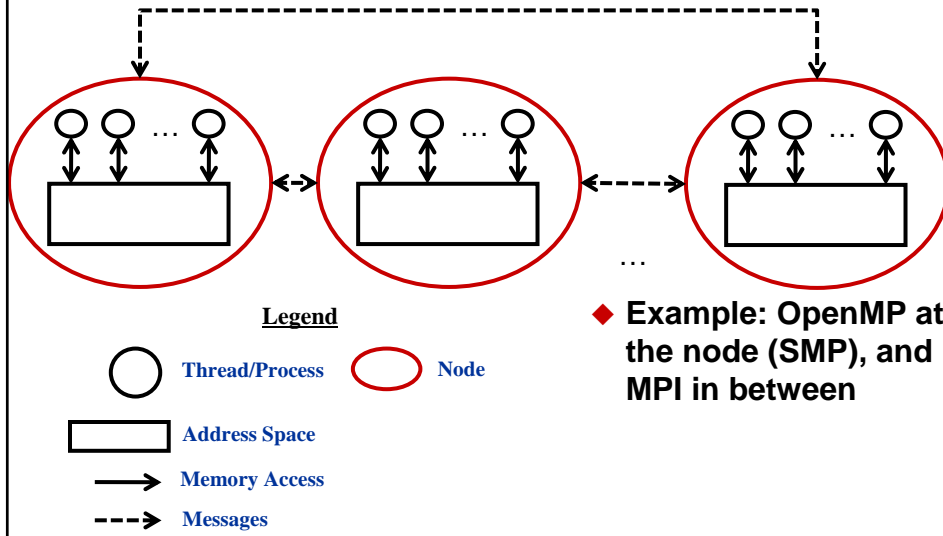
### Legend



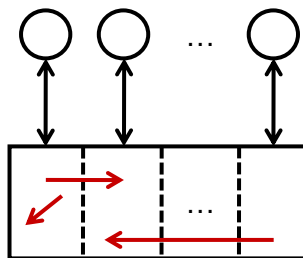
- ◆ Concurrent threads with shared space
- ◆ Positive:
  - Simple statements
  - Read remote memory via an expression
  - Write remote memory through assignment
- ◆ Negative:
  - Manipulating shared data leads to synchronization requirements
  - Does not allow locality exploitation
- ◆ Example: OpenMP, Java

8

## Hybrid Model(s) Example: Shared + Message Passing



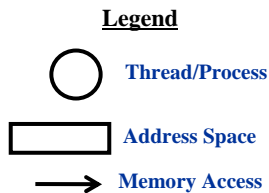
## The PGAS Model



- ◆ Concurrent threads with a partitioned shared space
  - A datum may reference data in other partitions
  - Global arrays have fragments in multiple partitions

- ◆ Positive:
  - Helps in exploiting locality
  - Simple statements as shared memory

- ◆ Negative:
  - sharing all memory can result in subtle bugs and race conditions
  - Examples: UPC, X10, Chapel, CAF, Titanium



## PGAS vs. other programming models/languages

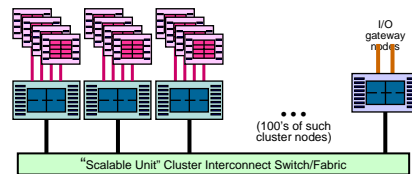
	UPC, X10, Chapel, CAF, Titanium	MPI	OpenMP
Memory model	PGAS (Partitioned Global Address Space)	Distributed Memory	Shared Memory
Notation	Language	Library	Annotations
Global arrays?	Yes	No	No
Global pointers/references?	Yes	No	No
Locality Exploitation	Yes	Yes, necessarily	No

11

## The heterogeneous/accelerated architectural landscape



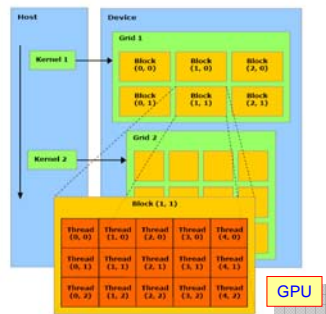
Cray XT5H: FPGA/Vector-accelerated Opteron



Road Runner: Cell-accelerated Opteron

12

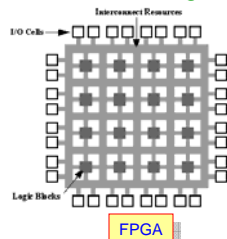
## Example accelerator technologies



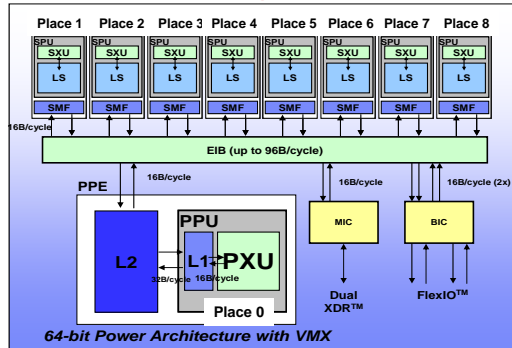
The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks.

Figure 2-1. Thread Batching

From NVIDIA CUDA Programming Guide

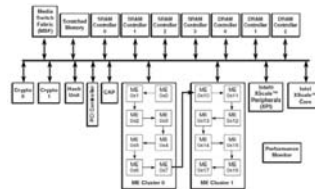


FPGA



64-bit Power Architecture with VMX

Cell Processor



Multi-core w/ accelerators (Intel IXP 2850)

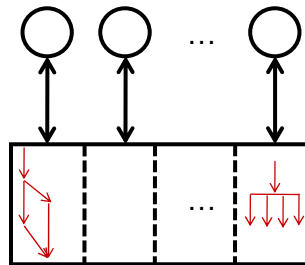
1.3

## The current architectural landscape

- ◆ Substantial architectural innovation is anticipated over the next ten years.
  - Hardware situation remains murky, but programmers need stable interfaces to develop applications
- ◆ Heterogenous accelerator-based systems will exist, raising serious programmability challenges.
  - Programmers must choreograph interactions between heterogenous processors, memory subsystems.
- ◆ Multicore systems will dramatically raise the number of cores available to applications.
  - Programmers must understand concurrent structure of their applications.
- ◆ Applications seeking to leverage these architectures will need to go beyond data-parallel, globally synchronizing MPI model.
- ◆ These changes, while most profound for HPC now, will change the face of commercial computing over time.

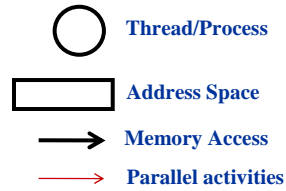
1.4

## PGAS with dynamic parallelism



X10, Chapel

### Legend



- ◆ Explicit concurrency
- ◆ SPMD is a special case
- ◆ Asynchronous activities can be started and stopped in a given space partition
- ◆ Asynchronous activities can be used for active messaging
  - DMAs,
  - fork/join concurrency, do-all/do-across parallelism

**Concurrency is made explicit and programmable.**

15

## How do we realize dynamic parallelism in PGAS?

- ◆ Through a dynamic PGAS library in C, Fortran, Java (co-habiting with MPI) which implements
  - remote references
  - global data-structures
  - inter-place messaging
  - global and/or collective operations
  - intra-place concurrency
  - atomic operations
- ◆ Through languages
  - Asynchronous CAF
    - ◆ extension of CAF with asyncs
  - Asynchronous UPC
    - ◆ extension of UPC with asyncs
  - X10 (already asynchronous)
    - ◆ extension of sequential Java
  - Chapel (already asynchronous)
- ◆ Leveraged runtimes such as XL UPC runtime, GASNet, ARMCI, LAPI, DCMF, DaCS, Cilk runtime, Chapel runtime
- ◆ Libraries reduce cost of adoption, languages offer enhanced productivity benefits

16

## Overview

- A. Introduction to PGAS
- B. Unified Parallel C - UPC**
- C. Discussion

17

## **B. Unified Parallel C UPC**

18

## UPC Overview

### 1) UPC in a nutshell

- Memory model
- Execution model
- UPC Systems

### 4) Advanced topics in UPC

- Dynamic Memory Allocation
- Synchronization in UPC
- UPC Libraries

### 2) Data Distribution and Pointers

- Shared vs Private Data
- Examples of data distribution
- UPC pointers

### 5) UPC Productivity

- Code efficiency

### 3) Workload Sharing

- `upc_forall`

19

## 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, ...**

20

## Introduction cont.

- ◆ UPC compilers are now available for most HPC platforms and clusters
  - Some are open source
- ◆ A debugger (Totalview from Etnus), a performance analysis tool (PPW from Univ. of Florida), Eclipse development environment (IBM)
- ◆ Benchmarks, programming examples, and compiler testing suite(s) are available
- ◆ Visit [www.upcworld.org](http://www.upcworld.org) or [upc.gwu.edu](http://upc.gwu.edu) for more information

21

## UPC Systems

- ◆ Current UPC Compilers
  - Hewlett-Packard
  - Cray
  - IBM
  - Berkeley
  - Intrepid (GCC UPC)
  - MTU
- ◆ UPC application development tools
  - Totalview
  - PPW from UF
  - Eclipse tools from IBM

22

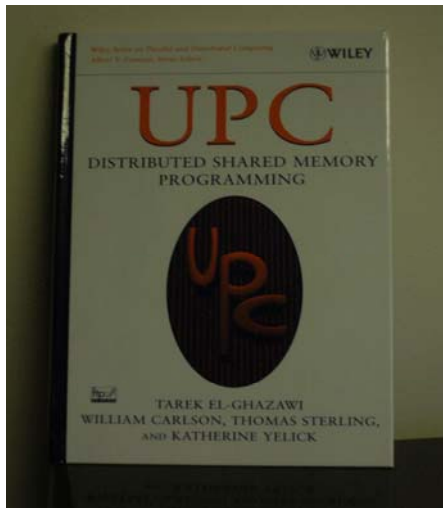
## UPC Home Page

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



23

## 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

24

## What is UPC?

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

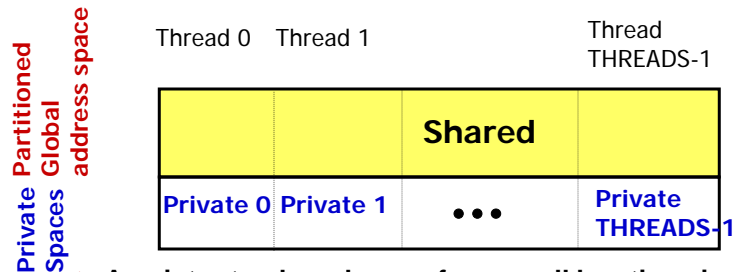
25

## 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**

26

## UPC Memory Model



- ◆ 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

27

## UPC Overview

- 1) UPC in a nutshell
  - Memory model
  - Execution model
  - UPC Systems
- 2) Data Distribution and Pointers
  - Shared vs. Private Data
  - Examples of data distribution
  - UPC pointers
- 3) Workload Sharing
  - `upc_forall`
- 4) Advanced topics in UPC
  - Dynamic Memory Allocation
  - Synchronization in UPC
  - UPC Libraries
- 5) UPC Productivity
  - Code efficiency

28

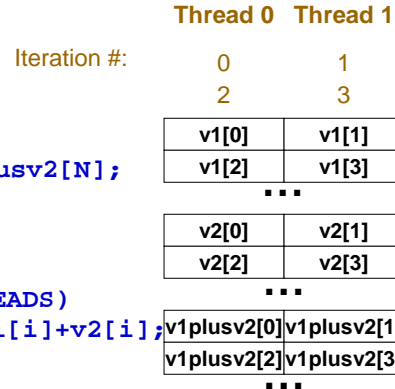
## A First Example: Vector addition

```

//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];
}

```



Shared Space

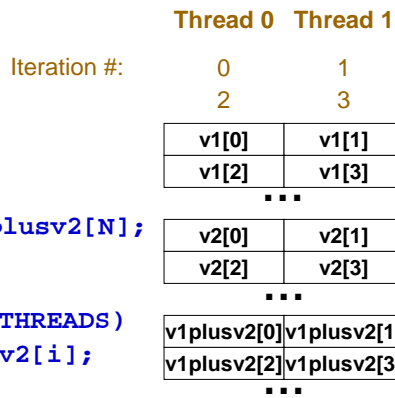
## 2<sup>nd</sup> Example: A More Efficient Implementation

```

//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];
}

```



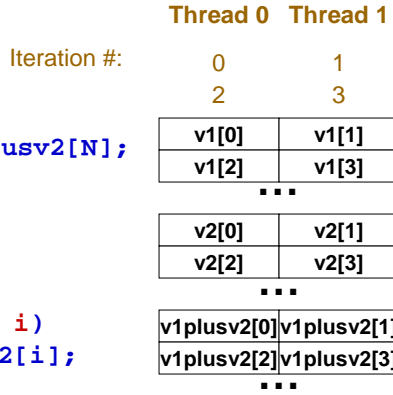
Shared Space

### 3<sup>rd</sup> 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];
}
```



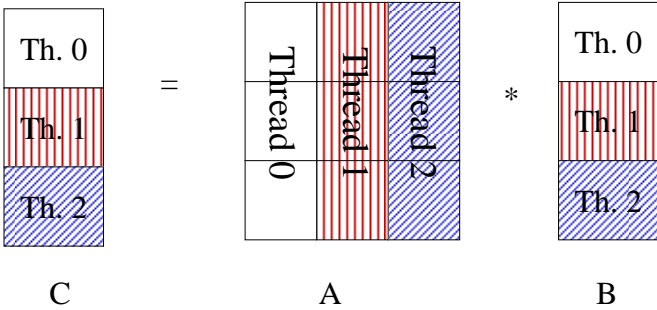
Shared Space

### Example: UPC Matrix-Vector Multiplication- Default Distribution

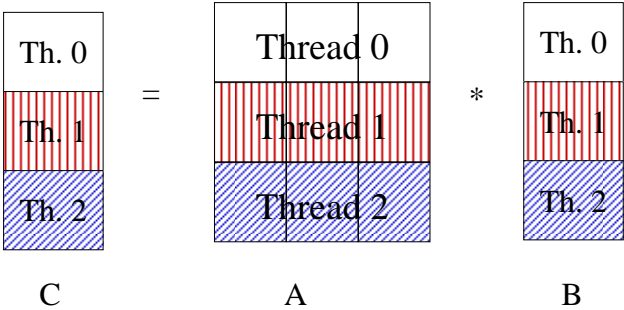
```
// vect_mat_mult.c
#include <upc_relaxed.h>

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



# 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++){
        c[i] = 0;
        for ( j= 0 ; j< THREADS ; j++)
            c[i] += a[i][j]*b[j];
    }
}
```

35

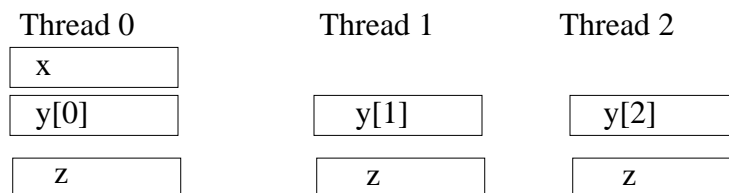
## Shared and Private Data

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:**



36

## Shared and Private Data

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

will result in the following data layout:

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

37

## Shared and Private Data

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

will result in the following data layout:

Thread 0	Thread 1	...	Thread (THREADS-1)
A[0][0]	A[0][1]	...	A[0][THREADS-1]
A[0][THREADS]	A[0][THREADS+1]	...	A[0][2*THREADS-1]
A[1][0]	A[1][1]	...	A[1][THREADS-1]
A[1][THREADS]	A[1][THREADS+1]	...	A[1][2*THREADS-1]

38

## 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];
```

39

## 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:

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

40

## 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

41

## 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]			

42

## 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**

43

## 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_elementsizedof(a) → sizeof(int)`

44

## 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

45

## 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

46

## UPC Pointers

Where does it point to?

		Private	Shared
Private		<b>PP</b>	<b>PS</b>
Shared		<b>SP</b>	<b>SS</b>

Where does it reside?

47

## UPC Pointers

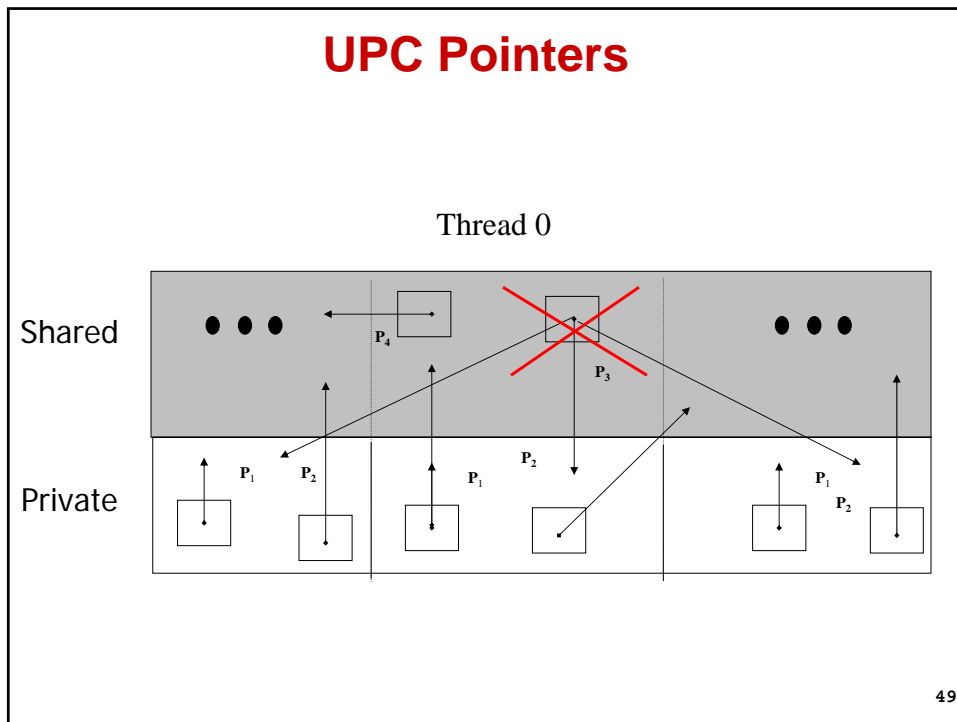
### ◆ How to declare them?

```
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.

48

## UPC Pointers



## 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)

Thread #	Block Address	Phase
----------	---------------	-------

- ◆ Example: Cray T3E implementation

Phase	Thread	Virtual Address
63	49 48	38 37
		0

51

## 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

52

## 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

53

## UPC Pointers

pointer to shared Arithmetic Examples:

Assume `THREADS = 4`

```
#define N 16
```

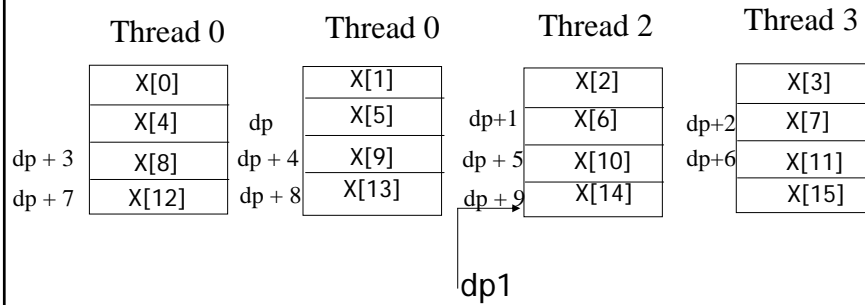
```
shared int x[N];
```

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

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

54

## UPC Pointers



55

## UPC Pointers

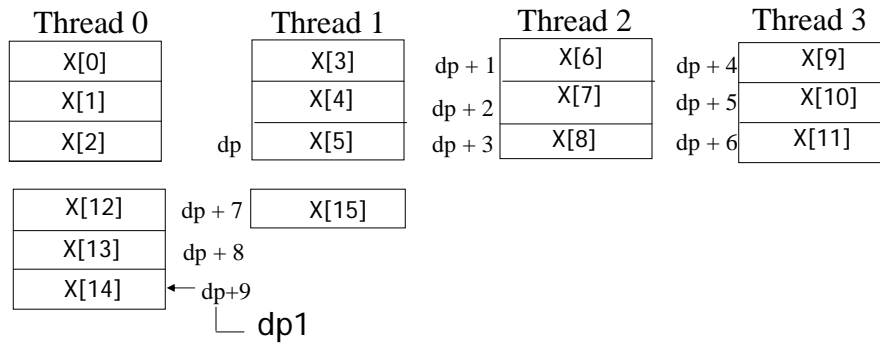
Assume THREADS = 4

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

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

56

## UPC Pointers



57

## 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

58

## UPC Pointers

### ◆ Mismatched Assignments

Assume THREADS = 4

**shared int x[N];**

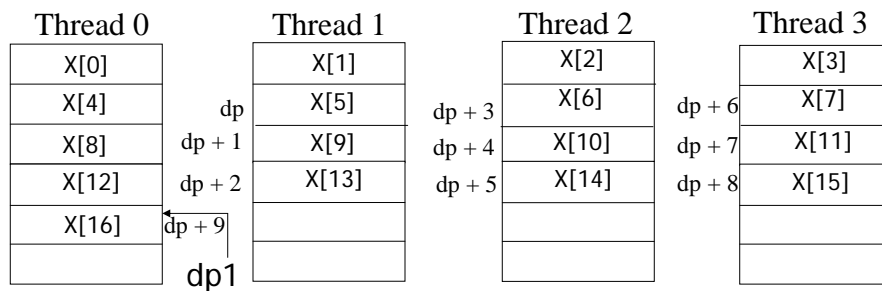
**shared[3] int \*dp=&x[5], \*dp1;**

**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

59

## UPC Pointers



60

## 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

61

## UPC Overview

- 1) UPC in a nutshell
  - Memory model
  - Execution model
  - UPC Systems
- 2) Data Distribution and Pointers
  - Shared vs Private Data
  - Examples of data distribution
  - UPC pointers
- 3) Workload Sharing
  - upc\_forall
- 4) Advanced topics in UPC
  - Dynamic Memory Allocation
  - Synchronization in UPC
  - UPC Libraries
- 5) UPC Productivity
  - Code efficiency

62

## 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

63

## 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

64

## 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

65

## UPC Overview

1) UPC in a nutshell

- Memory model
- Execution model
- UPC Systems

2) Data Distribution and Pointers

- Shared vs Private Data
- Examples of data distribution
- UPC pointers

3) Workload Sharing

- upc\_forall

4) Advanced topics in UPC

- Dynamic Memory Allocation
- Synchronization in UPC
- UPC Libraries

5) UPC Productivity

- Code efficiency

66

## 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”

67

## Collective Global Memory Allocation

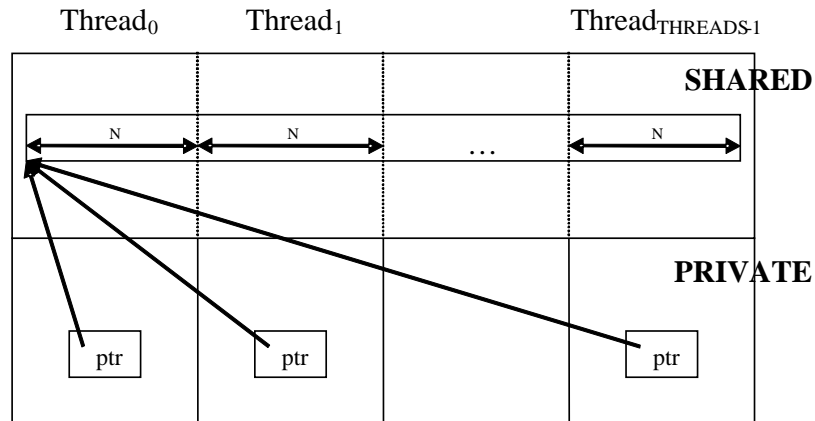
```
shared void *upc_all_alloc  
    (size_t nblocks, size_t nbytes);
```

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]`

68

## Collective Global Memory Allocation



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

69

## Global Memory Allocation

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

nblocks : number of blocks  
nbytes : block size

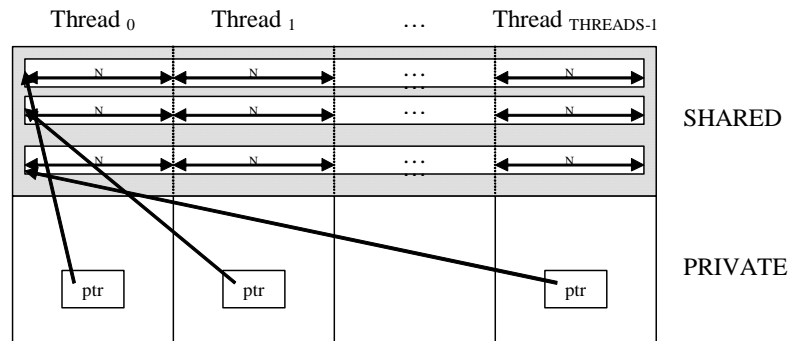
- ◆ Non collective, expected to be called by one thread
- ◆ The calling thread allocates a contiguous memory region in the shared space
- ◆ Space allocated per calling thread is equivalent to :  
`shared [nbytes] char[nblocks * nbytes]`
- ◆ If called by more than one thread, multiple regions are allocated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

70

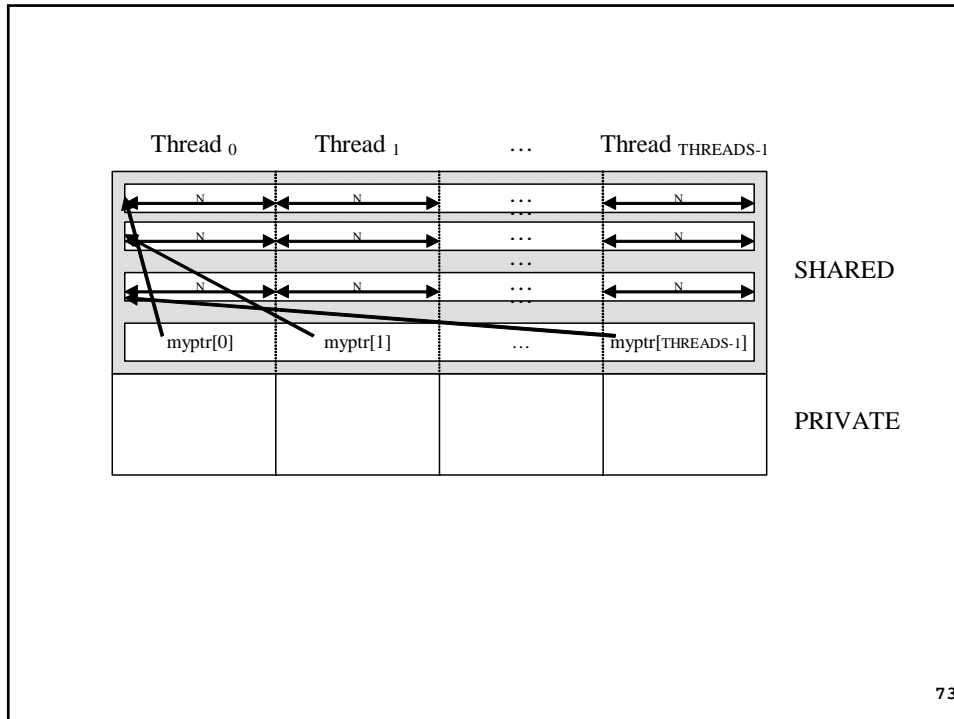
# Global Memory Allocation

```
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 ) );
```

71



72



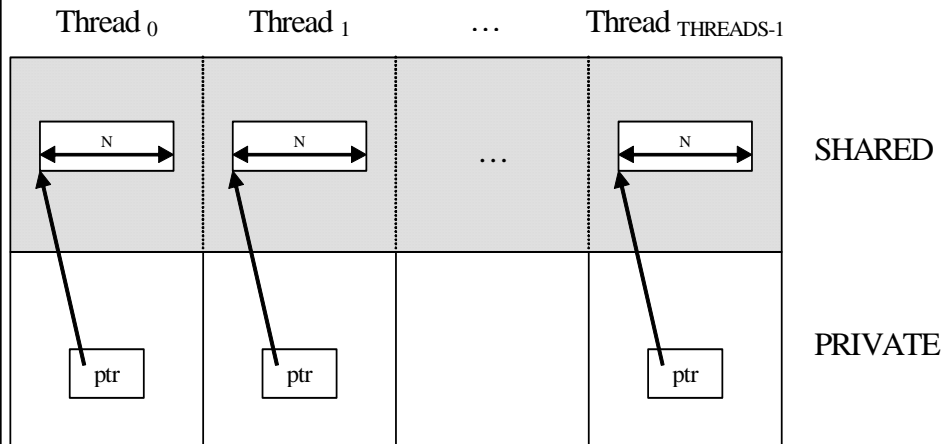
## Local-Shared Memory Allocation

```
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 ));
```

75

## 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

76

## Example: Matrix Multiplication in UPC

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

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

77

## 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] = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};

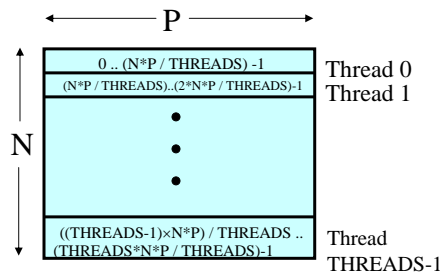
void main (void) {
    int i, j , l;
    for (i = 0 ; i<N ; i++) {
        for (j=0 ; j<M ;j++) {
            c[i][j] = 0;
            for (l = 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```

78

## Domain Decomposition for UPC

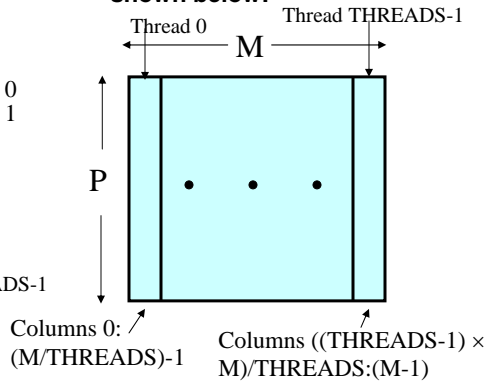
Exploiting locality in matrix multiplication

- ◆ **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  $\text{THREADS}$

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



79

## 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];
        }
    }
}
```

80

## 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];
        }
    }
}
```

81

## 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];
        }
    }
}
```

82

## 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];
        }
    }
}
```

83

## 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];
        }
    }
}
```

84

## 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

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

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

85

## Heat Transfer in Pictures

repeat until max change  $< \epsilon$

$$\sum \left( \begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{array} \right) \div 4 \implies \begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{array}$$

86

## Example: 2D Heat Conduction Problem

```
shared [BLOCKSIZE] double grids[2][N][N];
shared double dTmax_local[THREADS], dTmax_shared;
int x, y, nr_iter = 0, finished = 0;
int dg = 1, sg = 0;
double dTmax, dT, T, epsilon = 0.0001;
do {
    dTmax = 0.0;
    for( x=1; x<N-1; x++){
        upc_forall( y=1; y<N-1; y++; &grids[sg][x][y] ){
            T = (grids[sg][x-1][y] + grids[sg][x+1][y] +
                grids[sg][x][y-1] + grids[sg][x][y+1])
                / 4.0;
            dT = T - grids[sg][x][y];
            grids[dg][x][y] = T;
            if( dTmax < fabs(dT) )
                dTmax = fabs(dT);
        }
    }
}
```

Affinity field, used for work distribution

4-pt stencil

87

## Example: 2D Heat Conduction Problem

```
dTmax_local[MYTHREAD]=dTmax; if( dTmax_shared < epsilon )
upc_all_reduceD(                finished = 1;
    &dTmax_shared,                else{
    dTmax_local, UPC_MAX,         /*swapping the source &
    THREADS, 1, NULL,            destination "pointers"*/
    UPC_IN_ALLSYNC |
    UPC_OUT_ALLSYNC );          dg = sg;
                                sg = !sg;
                                }
                                nr_iter++;
                                } while( !finished );
upc_barrier;
```

reduction operation using UPC  
collectives library

88

## Example: RandomAccess

**Description of the problem:**

Let T be a table of size  $2^n$  and let S be a table of size  $2^m$  filled with random 64-bit integers.

Let  $\{A_i\}$  be a stream of 64-bit integers of length  $2^{n+2}$  generated by the primitive polynomial over GF(2),  $X_{63}+1$ .

For each  $a_i$  :

$$T[\text{LSB}_{n-1\dots 0}(a_i)] = T[\text{LSB}_{n-1\dots 0}(a_i)] \text{ XOR } S[\text{MSB}_{m-1\dots 0}(a_i)]$$

**2 Sets of typical problem sizes:**

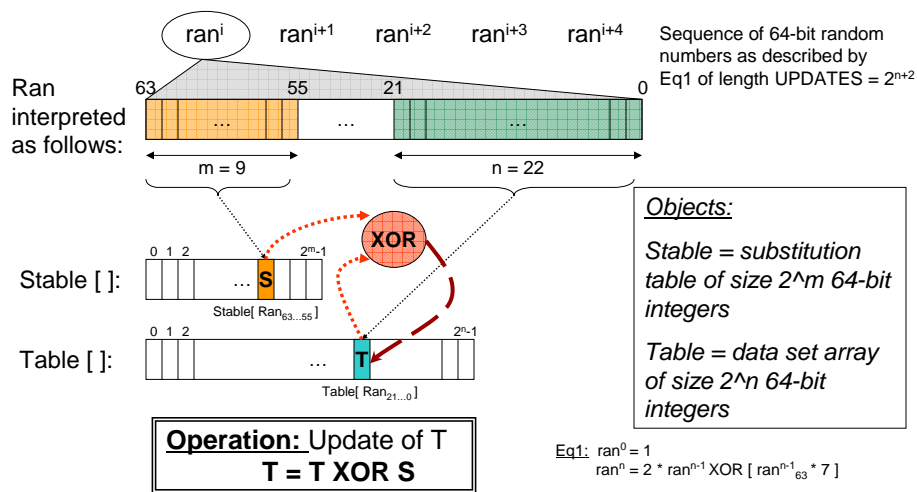
(a)  $m=9, n=8, 9$ , max integer size possible

(b)  $m$  such as  $2^m$  is half of the size of the cache

$n$  such as  $2^n$  is equal to

half of the total memory

## Example: RandomAccess

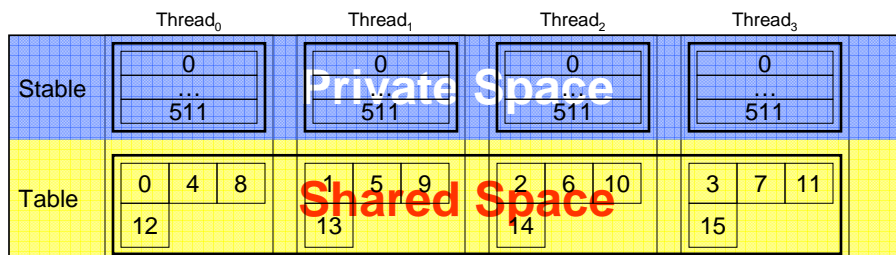


## RandomAccess – UPC

```

u64Int Stable[STSIZE]; // private
shared u64Int *Table; // shared
// Table[] allocated dynamically at run-time by:
Table = (shared u64Int *)
    upc_all_alloc(TableSize, sizeof(u64Int));
// STSIZE=2m      where m=9,
// TableSize=2n  where n=4 in this example
...
Table[ran&(TableSize-1)] ^= Stable[ran>>B_SHR];

```



91

## RandomAccess: Computational Core

```

void RandomAccessUpdate(u64Int TableSize) {
    s64Int i;
    u64Int ran[128], ind;
    int j;
    /* Initialize main table */
    upc_forall( i=0; i<TableSize; i++; i )
        Table[i] = i;
    upc_barrier;
    for (j=0; j<128; j++)
        ran[j] = starts ((NUPDATE/128) * j);
    for (i=0; i<NUPDATE/128; i++) {
        upc_forall( j=0; j<128; j++; j ){
            ran[j] = (ran[j] << 1) ^ ((s64Int) ran[j] < 0 ? POLY : 0);
            Table[ran[j] & (TableSize-1)] ^= Stable[ran[j] >>(64-LSTSIZE)];
        }
    }
}

```

Initialize the **local** part of Table[]

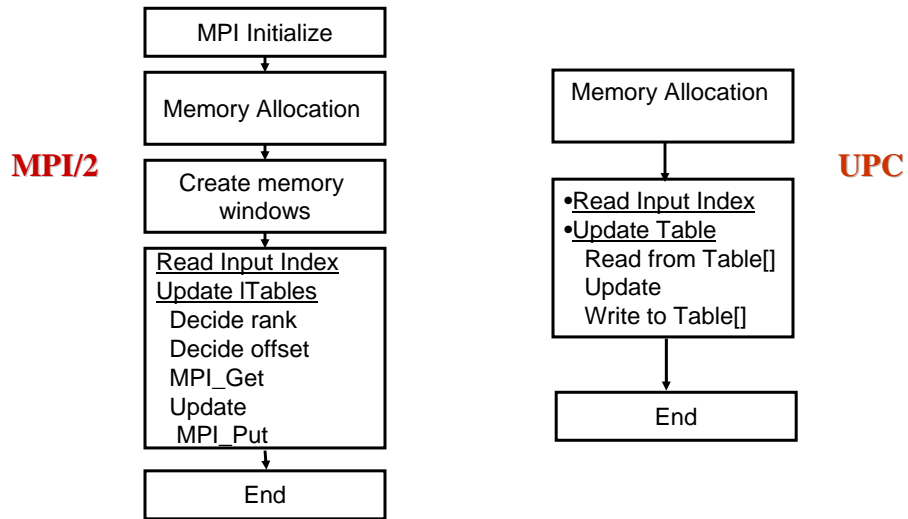
**Synchronization**

**Workload distribution**

As-it-is in SEQ code

92

## Expressing Memory Operations Efficiently: Large Histogram (HPCC RA)



93

## Compact Code

	Random Access (#lines)	%Increase
<b>C</b>	<b>144</b>	-
<b>UPC</b>	<b>158</b>	<b>9.72%</b>
<b>MPI/2</b>	<b>210</b>	<b>45.83%</b>
<b>MPI/1</b>	<b>344</b>	<b>138.89%</b>

\*Study done with HPCC 0.5alpha compliant code

94

## Less Conceptual Complexity

		Work Distr.	Data Distr.	Comm.	Synch. & Consist.	Misc. Ops	Sum	Overall Score
RandomAccess MPI/2	# Parameters	26	9	35	5	6	81	151
	# Function Calls	0	2	8	4	4	18	
	# Keywords with rank and np	15	6	8	0	2	31	
	# MPI Types	0	5	10	4	2	21	
	Notes	11 If 5 For	1 memalloc 1 window create	4 for collective operation 4 one-sided	1 fence 3 barriers (1 implicit)	mpi_init mpi_finalize mpi_comm_rank mpi_comm_size		
RandomAccess UPC	# Parameters	19	2	0	0	2	23	42
	# Function Calls	0	1	0	5	2	8	
	# Keywords	5	1	0	0	0	6	
	# UPC Constructs & UPC Types	3	2	0	0	0	5	
	Notes	3 forall 4 if 1 for	2 shared 1 upc_all_alloc		5 barriers	2 global_exit		

95

## Synchronization

- ◆ **Explicit synchronization with the following mechanisms:**
  - **Barriers**
  - **Locks**
  - **Memory Consistency Control**
  - **Fence**

96

## 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

97

## 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

98

## 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)

99

## 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

100

## 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.

101

## UPC Libraries

- ◆ UPC Collectives
- ◆ UPC-IO

102

## Overview UPC Collectives

- ◆ 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 library include functions for bulk data movement and computation.
  - `upc_all_broadcast`, `upc_all_exchange`,  
`upc_all_prefix_reduce`, **etc.**

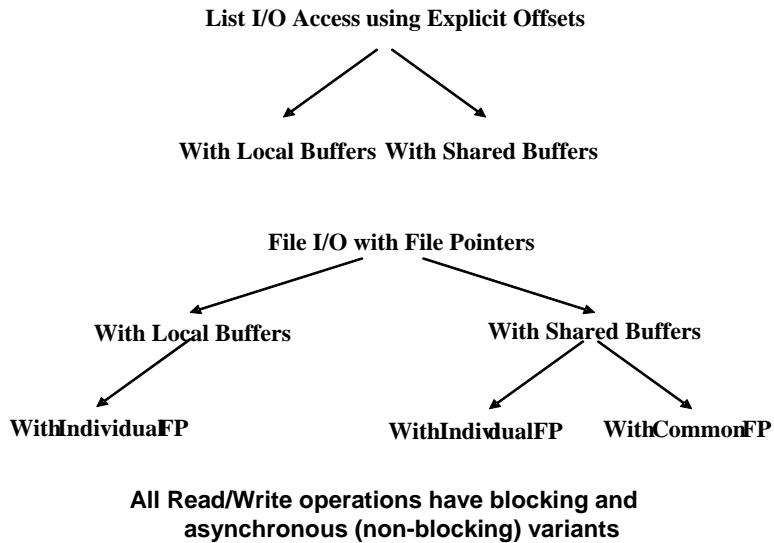
103

## Overview of UPC-IO

- ◆ Most UPC-IO functions are collective
  - Function entry/exit includes implicit synchronization
  - Single return values for specific functions
- ◆ API provided through extension libraries
- ◆ UPC-IO data operations support:
  - shared or private buffers
  - Blocking (`upc_all_fread_shared()`, ...)
  - Non-blocking (async) operations (`upc_all_fread_shared_async()`, ...)
- ◆ Supports List-IO Access
- ◆ Several reference implementations by GWU

104

## File Accessing and File Pointers



105

## UPC Overview

- 1) UPC in a nutshell
  - Memory model
  - Execution model
  - UPC Systems
- 2) Data Distribution and Pointers
  - Shared vs Private Data
  - Examples of data distribution
  - UPC pointers
- 3) Workload Sharing
  - `upc_forall`
- 4) Advanced topics in UPC
  - Dynamic Memory Allocation
  - Synchronization in UPC
  - UPC Libraries
- 5) UPC Productivity
  - Code efficiency

106

## Reduced Coding Effort is Not Limited to Random Access– NPB Examples

		SEQ1	UPC	SEQ2	MPI	UPC Effort (%)	MPI Effort (%)
NPB-CG	#lines	665	710	506	1046	6.77	106.72
	#chars	16145	17200	16485	37501	6.53	127.49
NPB-EP	#lines	127	183	130	181	44.09	36.23
	#chars	2868	4117	4741	6567	43.55	38.52
NPB-FT	#lines	575	1018	665	1278	77.04	92.18
	#chars	13090	21672	22188	44348	65.56	99.87
NPB-IS	#lines	353	528	353	627	49.58	77.62
	#chars	7273	13114	7273	13324	80.31	83.20
NPB-MG	#lines	610	866	885	1613	41.97	82.26
	#chars	14830	21990	27129	50497	48.28	86.14

$$UPC_{effort} = \frac{\#UPC - \#SEQ1}{\#SEQ1}$$

$$MPI_{effort} = \frac{\#MPI - \#SEQ2}{\#SEQ2}$$

SEQ1 is C

SEQ2 is from NAS, all FORTRAN except for IS

107

## C. Discussion

108