Unified Parallel C (UPC)

Vivek Sarkar

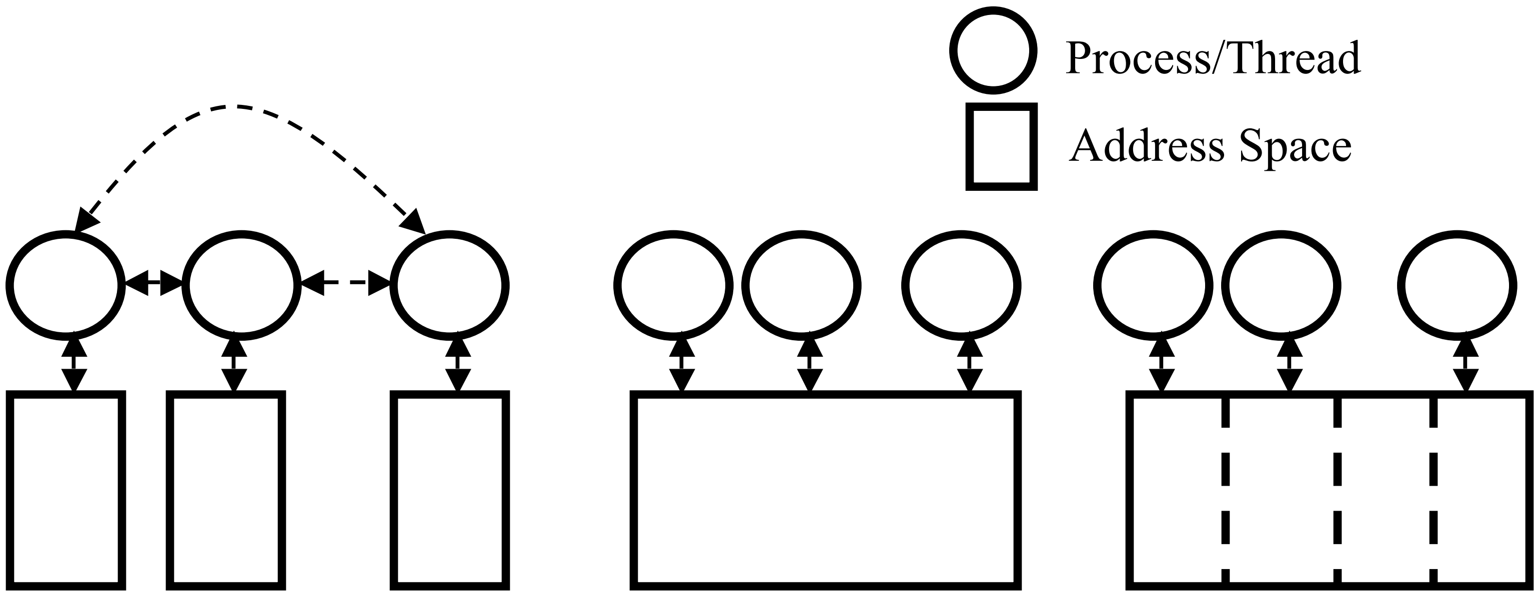
Department of Computer Science
Rice University

vsarkar@cs.rice.edu

Acknowledgments

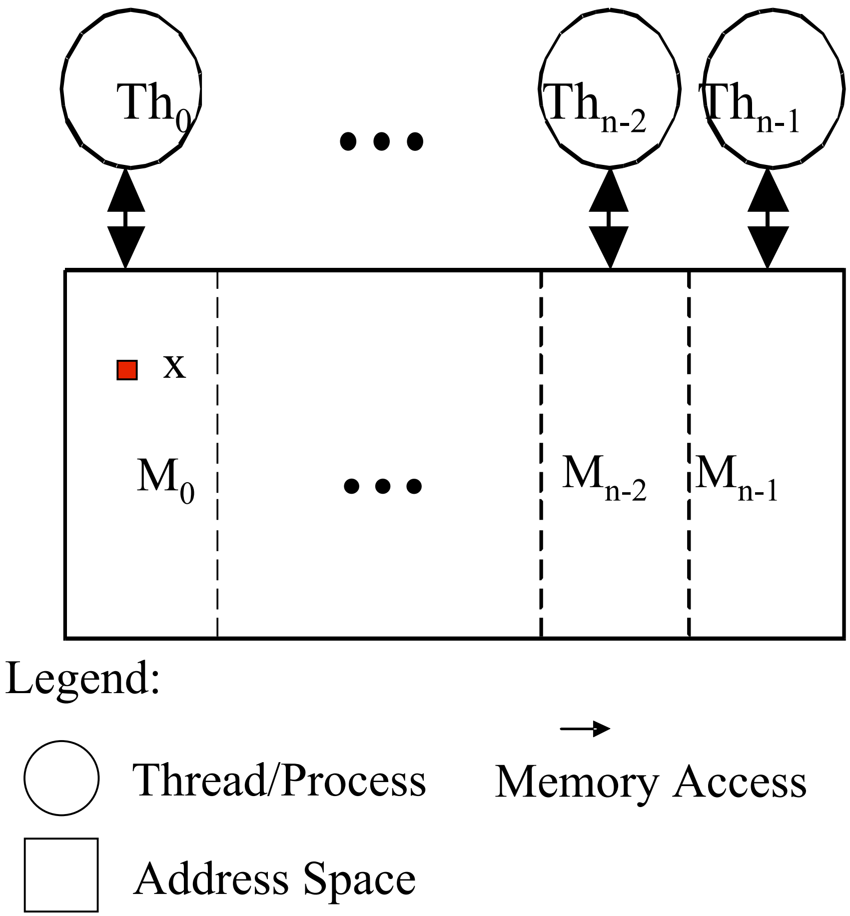
- Supercomputing 2007 tutorial on "Programming using the Partitioned Global Address Space (PGAS) Model" by Tarek El-Ghazawi and Vivek Sarkar
  - —http://sc07.supercomputing.org/schedule/event_detail.php?evid=11029

Programming Models



Message Passing	Shared Memory	**DSM/PGAS**
MPI	**OpenMP**	**UPC**

The Partitioned Global Address Space (PGAS) Model



- Aka the Distributed Shared Memory (DSM) model
- Concurrent threads with a partitioned shared space
  - Memory partition M_i has affinity to thread Th_i
- (+)ive:
  - Data movement is implicit
  - Data distribution simplified with global address space
- (-)ive:
  - Computation distribution and synchronization still remain programmer's responsibility
  - Lack of performance transparency of local vs. remote accesses
- UPC, CAF, Titanium, X10, …

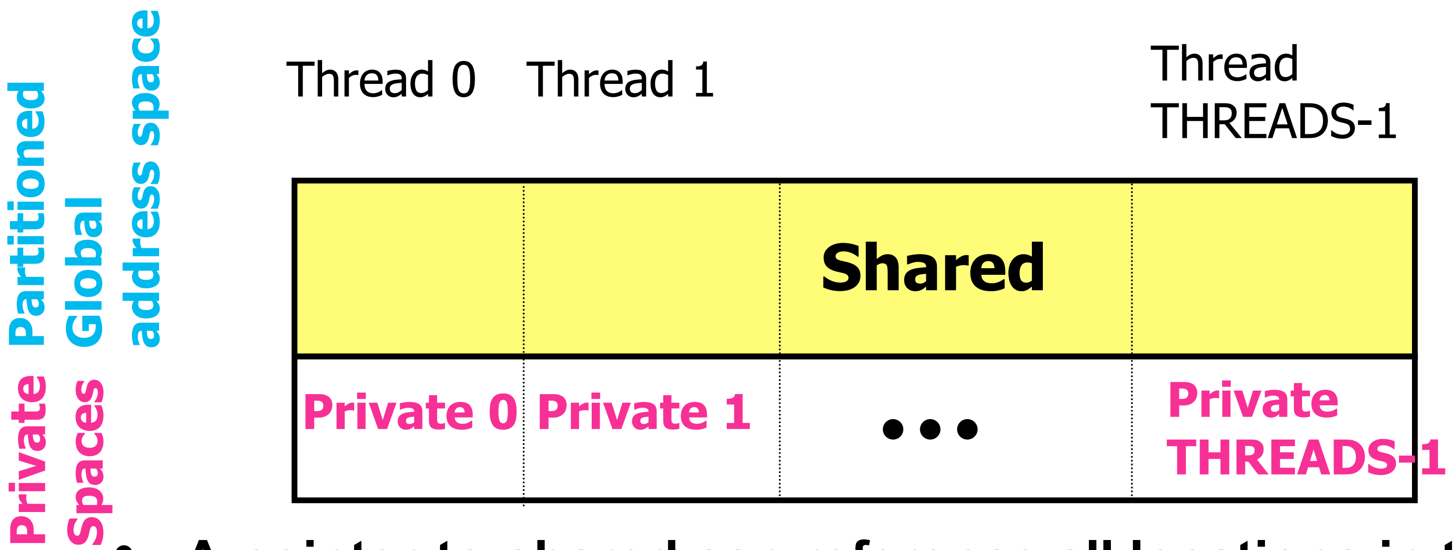
What is Unified Parallel C (UPC)?

- **An explicit parallel extension of ISO C**
- **Single global address space**
- **Collection of threads**
  - **—each thread bound to a processor**
  - **—each thread has some private data**
  - **—part of the shared data is co-located with each thread**
- **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 industry and academia**

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



- **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**

A First Example: Vector addition

```
//vect_add.c

#include <upc_relaxed.h>
#define N 100*THREADS

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

/* cyclic distribution by default */

void main() {
    int i;
    for(i=0; i<N; i++)

            if (MYTHREAD==i%THREADS)
            v1plusv2[i]=v1[i]+v2[i];
}
```

	Thread 0	Thread 1
Iteration #:	0	1
	2	3

Shared Space

v1[0]	**v1[1]**
v1[2]	**v1[3]**

...

v2[0]	**v2[1]**
v2[2]	**v2[3]**

...

v1plusv2[0]	**v1plusv2[1]**
v1plusv2[2]	**v1plusv2[3]**

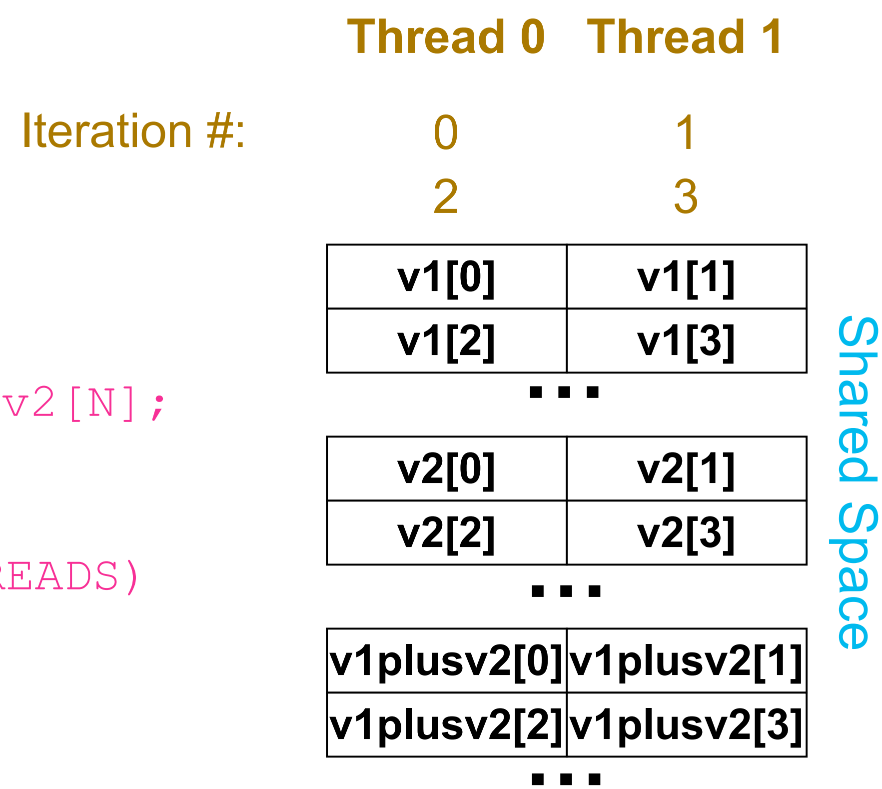
...

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];
}
```



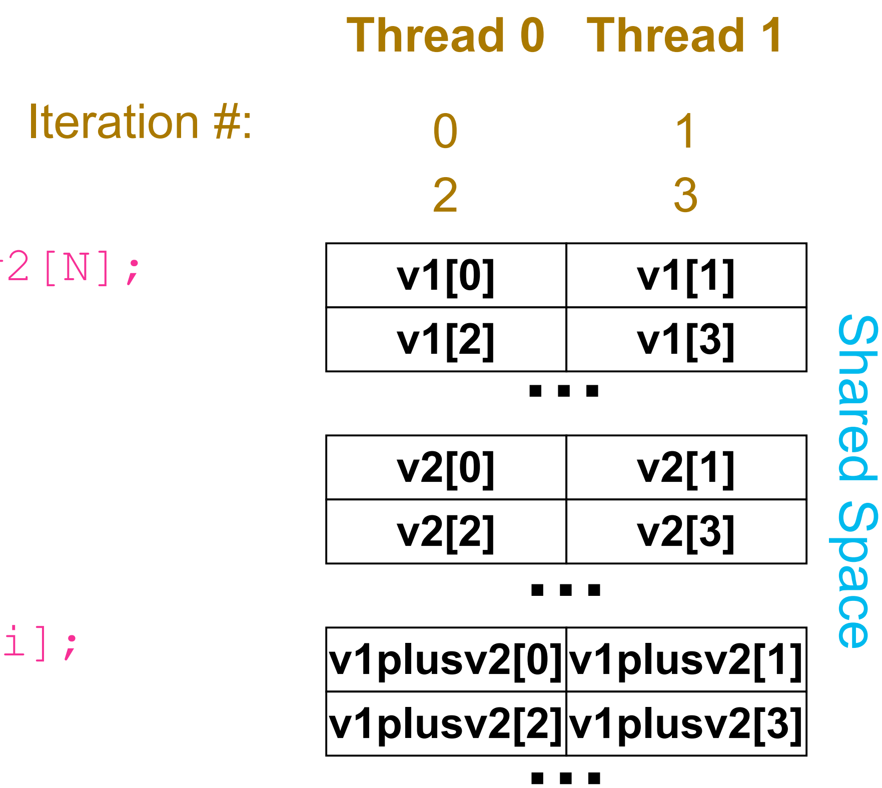
3rd Example: A More Convenient Implementation with upc_forall

```
//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];
}
```



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

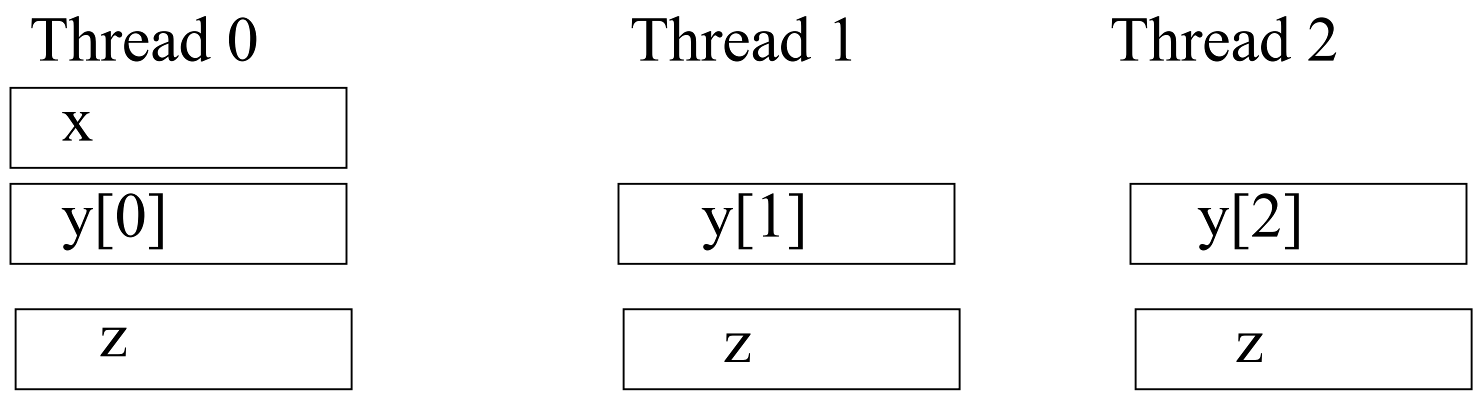
Shared and Private Data

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; /* private by default */
```

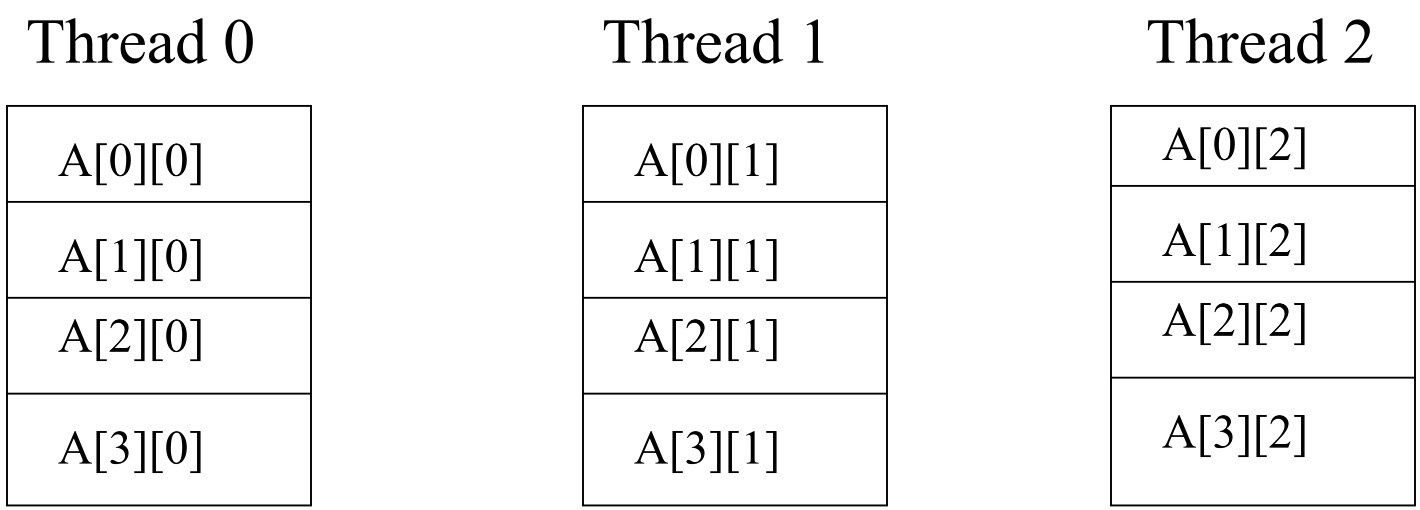
will result in the layout:



Shared and Private Data

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

will result in the following data layout:



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]

Block-Cyclic Distributions for 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];`

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]			

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:**

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

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

- 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
- Shared blocks above must be contiguous with all of its elements having the same affinity
- Private block must also be contiguous

UPC Pointers

Where does it point to?

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

UPC Pointers

Needed for expressive data structures

- **Private pointers pointing locally**
  - **int *p1**
- **Private pointers pointing into the shared space**
  - **shared int *p2**
- **Shared pointers pointing locally** *(not recommended!)*
  - **int *shared p3**
- **Shared pointers pointing to shared space**
  - **shared int *shared p4**

UPC Pointers Challenges

- **Handling shared data**
- **Supporting pointer arithmetic**
- **Supporting pointer casting**

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

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

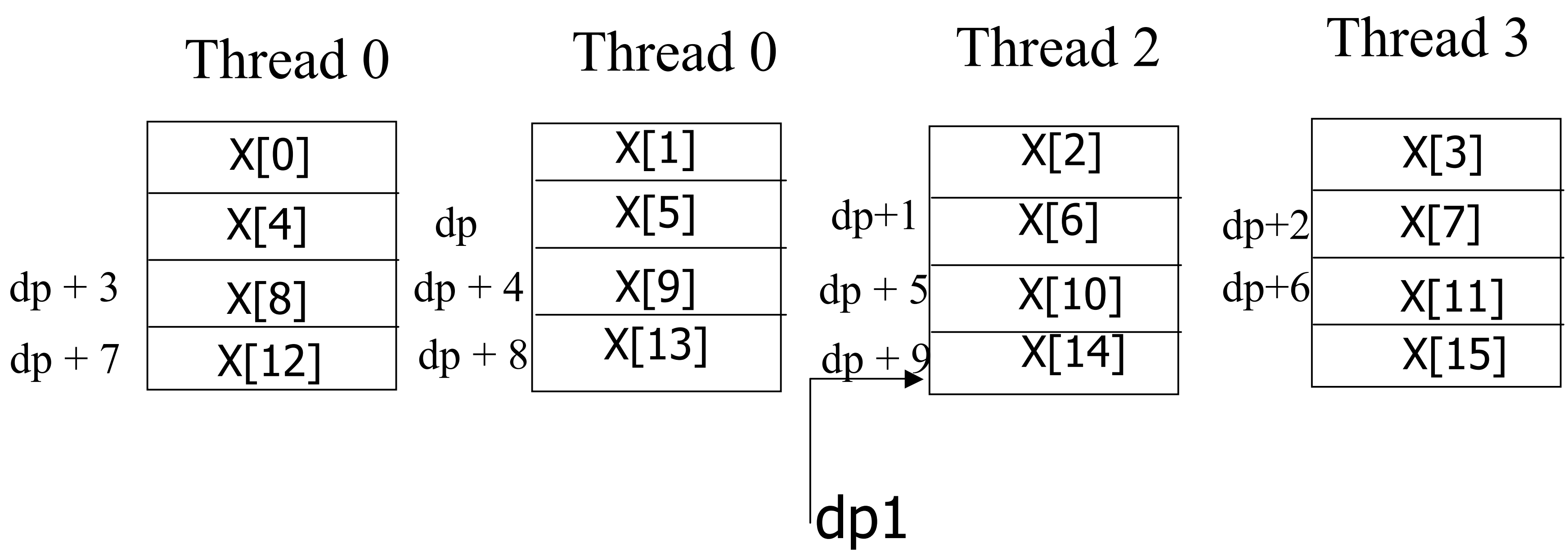
- **size_t upc_threadof(shared void *ptr);**
  returns the thread number that has affinity to the object pointed to by ptr
- **size_t upc_phaseof(shared void *ptr);**
  returns the index (position within the block) of the object which is pointed to by ptr
- **size_t upc_addrfield(shared void *ptr);**
  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

<u>pointer to shared Arithmetic Examples:</u>

Assume THREADS = 4

```
#define N 16
shared int x[N];
shared int *dp=&x[5], *dp1;
dp1 =  dp + 9;
```

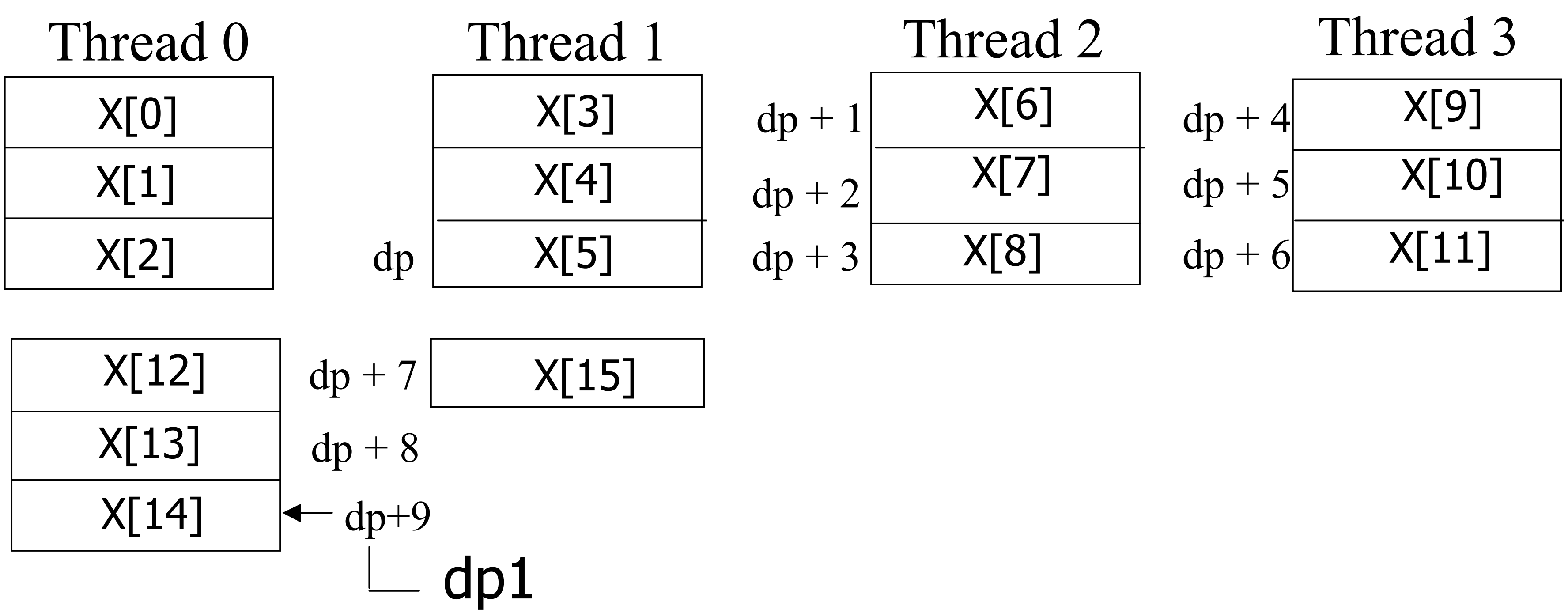


UPC Pointers

Assume THREADS = 4

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

dp1 = dp + 9;



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: Mismatched Assignments

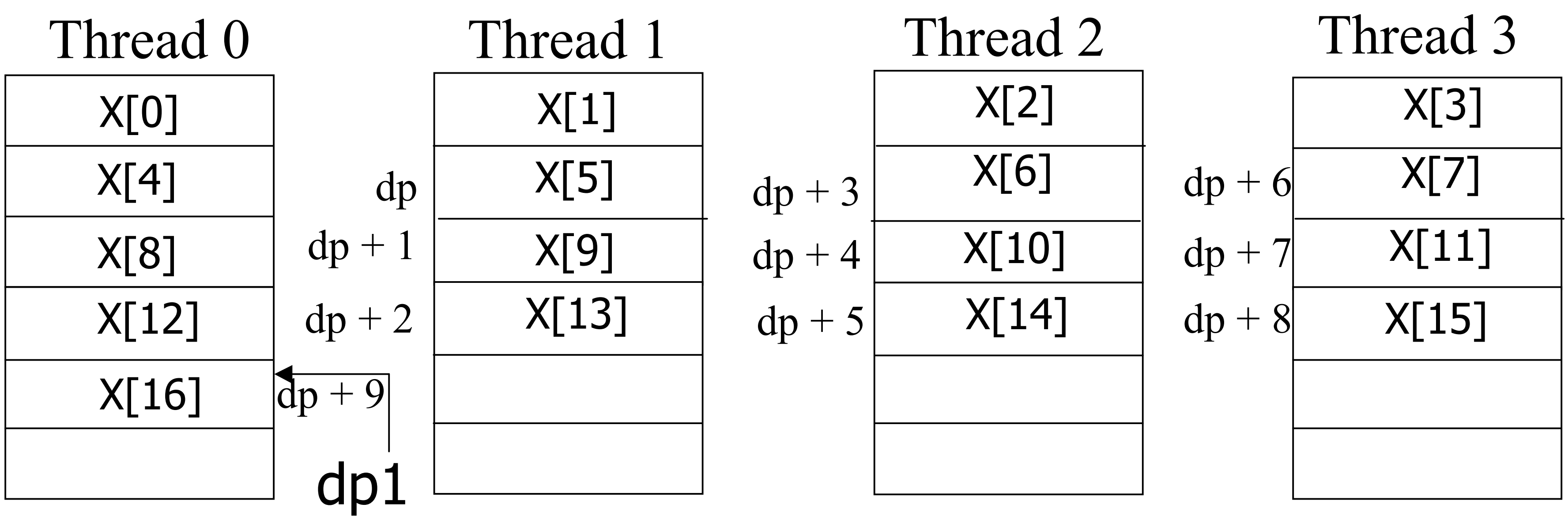
Assume THREADS = 4

```
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

- 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];
```

<u>Note:</u> 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

Synchronization

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

Synchronization - Barriers

- **No implicit synchronization among the threads**
- **UPC provides the following barrier synchronization constructs:**
  - **Barriers (Blocking)**
    - **upc_barrier expr$_{opt}$;**
  - **Split-Phase Barriers (Non-blocking)**
    - **upc_notify expr$_{opt}$;**
    - **upc_wait expr$_{opt}$;**

    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

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- 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.

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*.**

Library Calls for Bulk Data Transfer

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

Library Calls for Non-blocking Data Transfer

upc_handle_t bupc_memcpy_async(shared void *dst, shared const void *src, size_tnbytes);

upc_handle_t bupc_memget_async(void *dst, shared const void *src, size_tnbytes);

upc_handle_t bupc_memput_async(shared void *dst, const void *src, size_tnbytes);

- Same args and semantics as blocking variants
- Returns a upc_handle_t - an opaque handle representing the initiated asynchronous operation
- Complete using one of two functions:
  - —block for completion: void bupc_waitsync(upc_handle_t handle);
  - —Non-blocking test: int bupc_trysync(upc_handle_t handle);