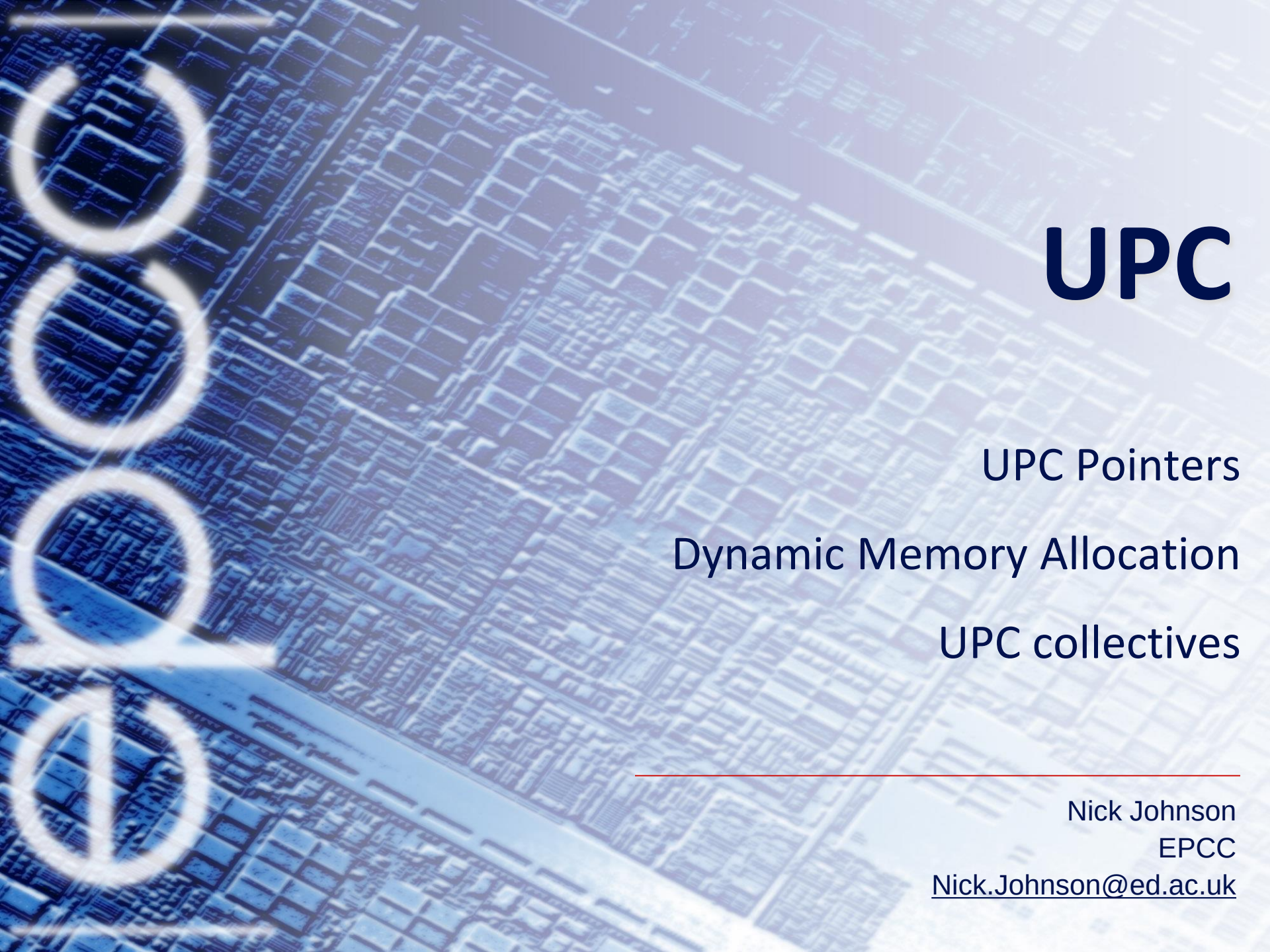


UPC

UPC Pointers

Dynamic Memory Allocation

UPC collectives

Nick Johnson
EPCC
Nick.Johnson@ed.ac.uk

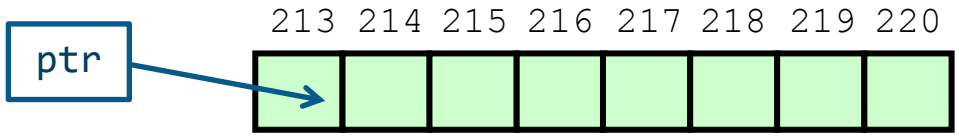
Advanced use of UPC

- C and UPC pointers
  - dynamic memory allocation
  - locks
- UPC collectives

C pointers

A *pointer* in C is a data type whose value points to another variable's memory address

```
float[2] array;

float* ptr = &array[0];  // Value of ptr is 213
```

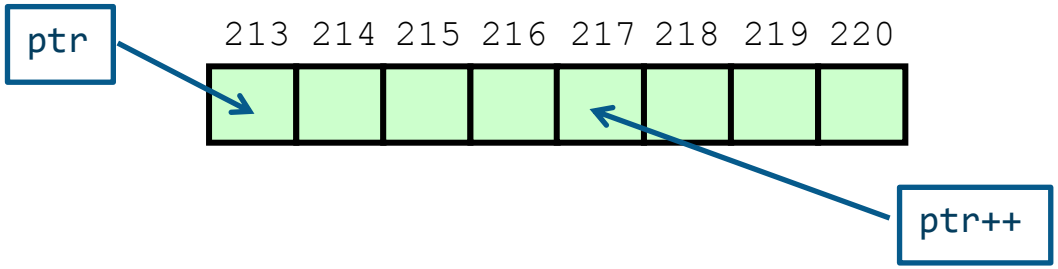


Pointer arithmetic

change what object a pointer is referring to through pointer arithmetic:

- is type dependent
- incrementing a *float* pointer will move by **`sizeof(float)`**

```
float[2] array;
float* ptr = &array[0];  // Value of ptr is 213
ptr = ptr++;  // Value of ptr is now 217
```



UPC pointers

similar concept as in C

pointers are variables that contain addresses of other variables

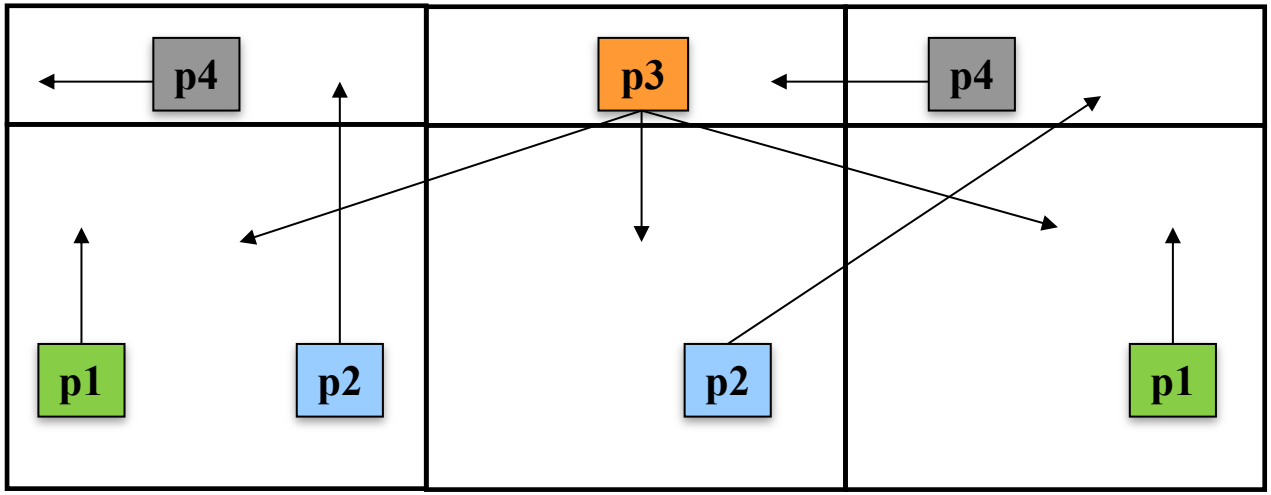
UPC pointers can

reside in private or shared memory space

reference private or shared memory space

Types of pointers

```
private to private   → int *p1;
private to shared    → shared int *p2;
shared to private    → int *shared p3; (not recommended)
shared to shared     → shared int *shared p4;
```



Shared Memory Space

Private Memory Space

UPC pointers

UPC pointers have three fields

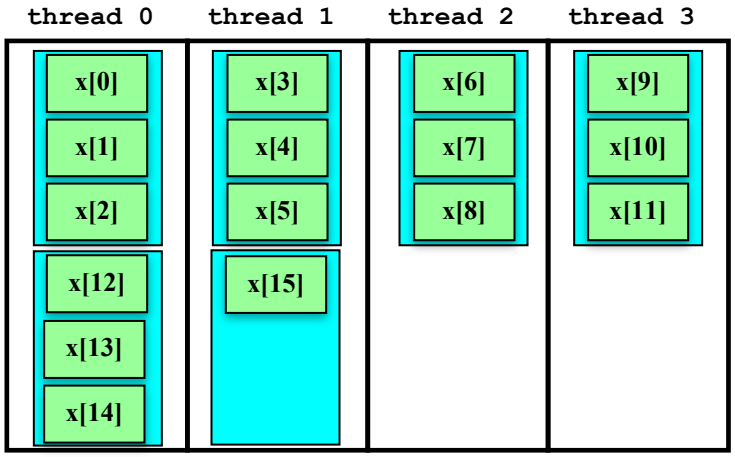
- `thread` : the thread affinity of the pointer
- `address` : the virtual address of the block
- `phase` : indicates the element location within that block

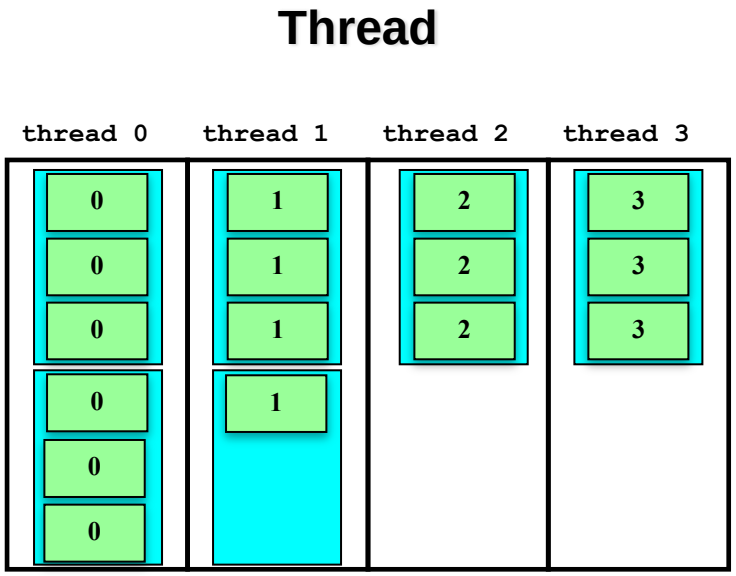
the values of these fields are obtained from the functions

```
size_t upc_threadof (shared void *ptr)
   size_t upc_phaseof (shared void *ptr)
      size_t upc_addrfield (shared void *ptr)
```

Pointer fields: thread

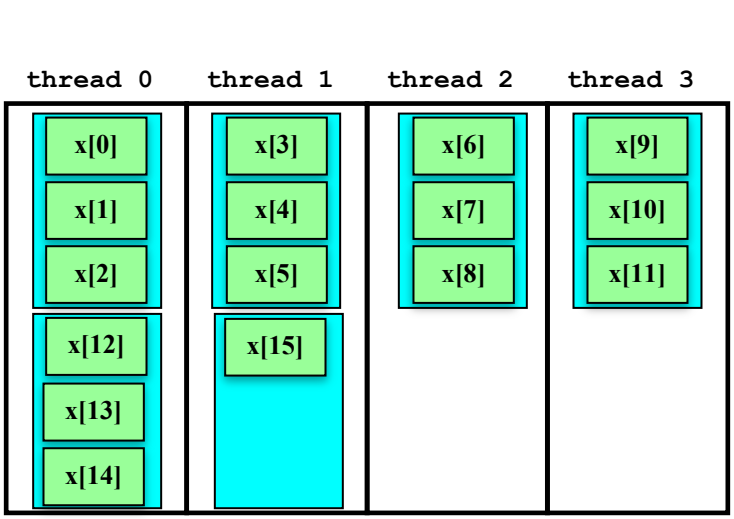
```
shared [3] float x[16];
```

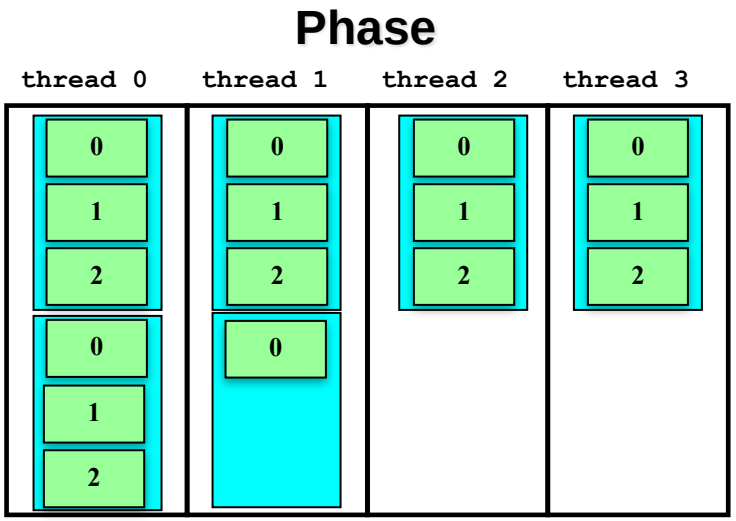




Pointer fields: phase

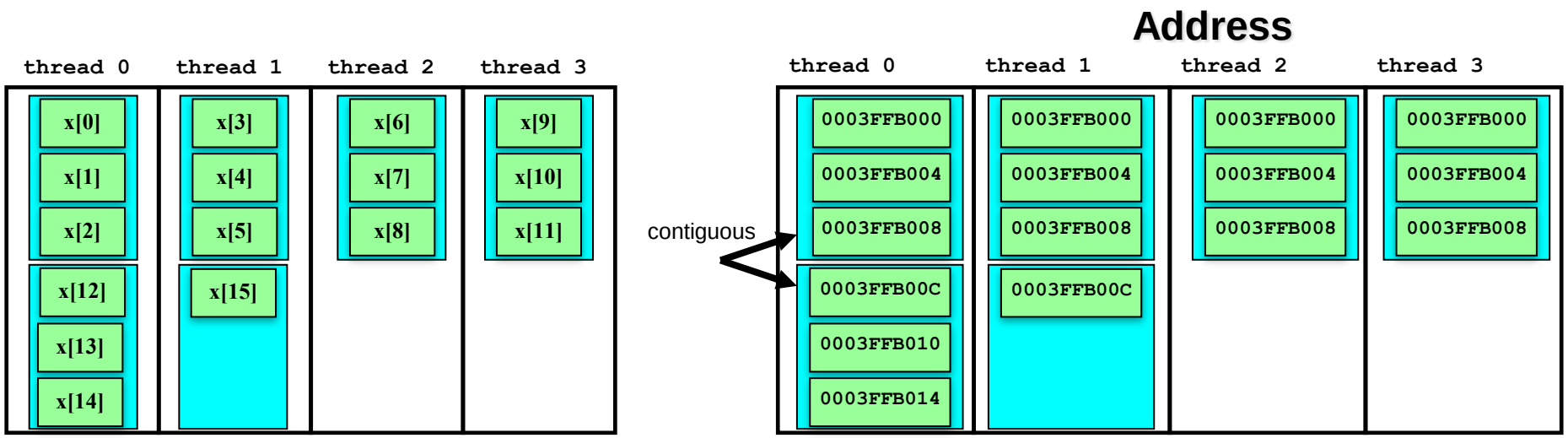
```
shared [3] float x[16];
```





Pointer fields: address

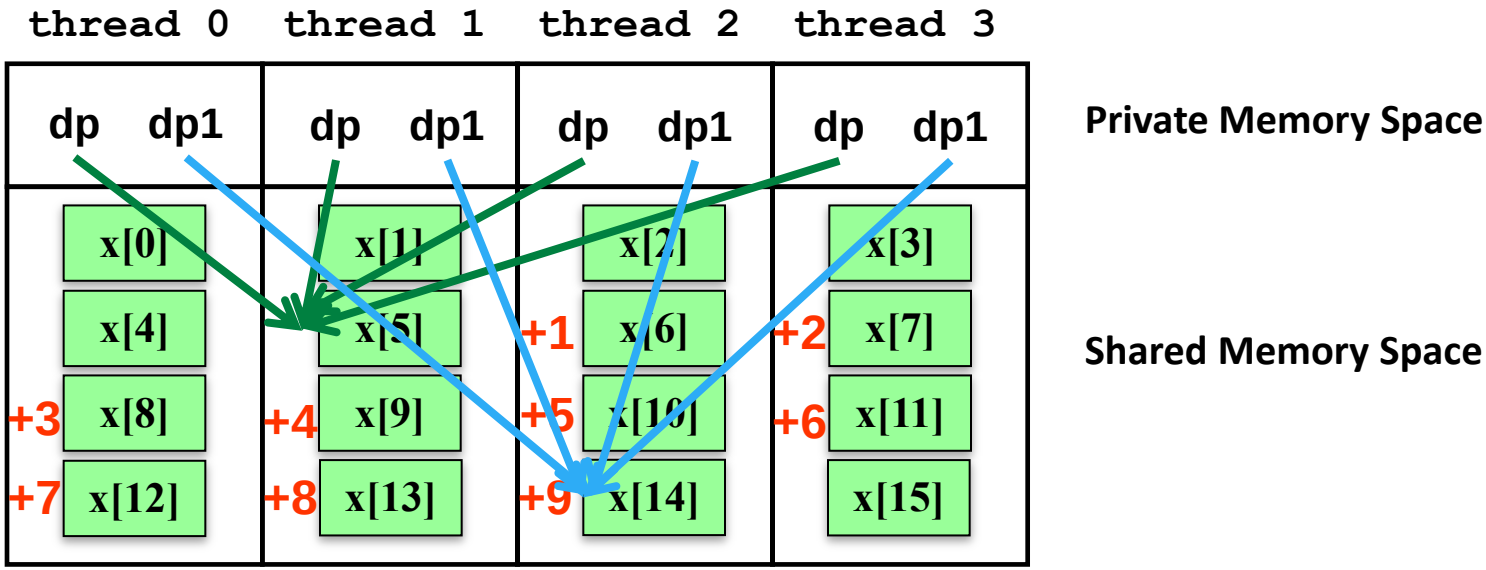
```
shared [3] float x[16];
```



Pointer properties (1/3)

Pointer arithmetic takes into account the blocking factor

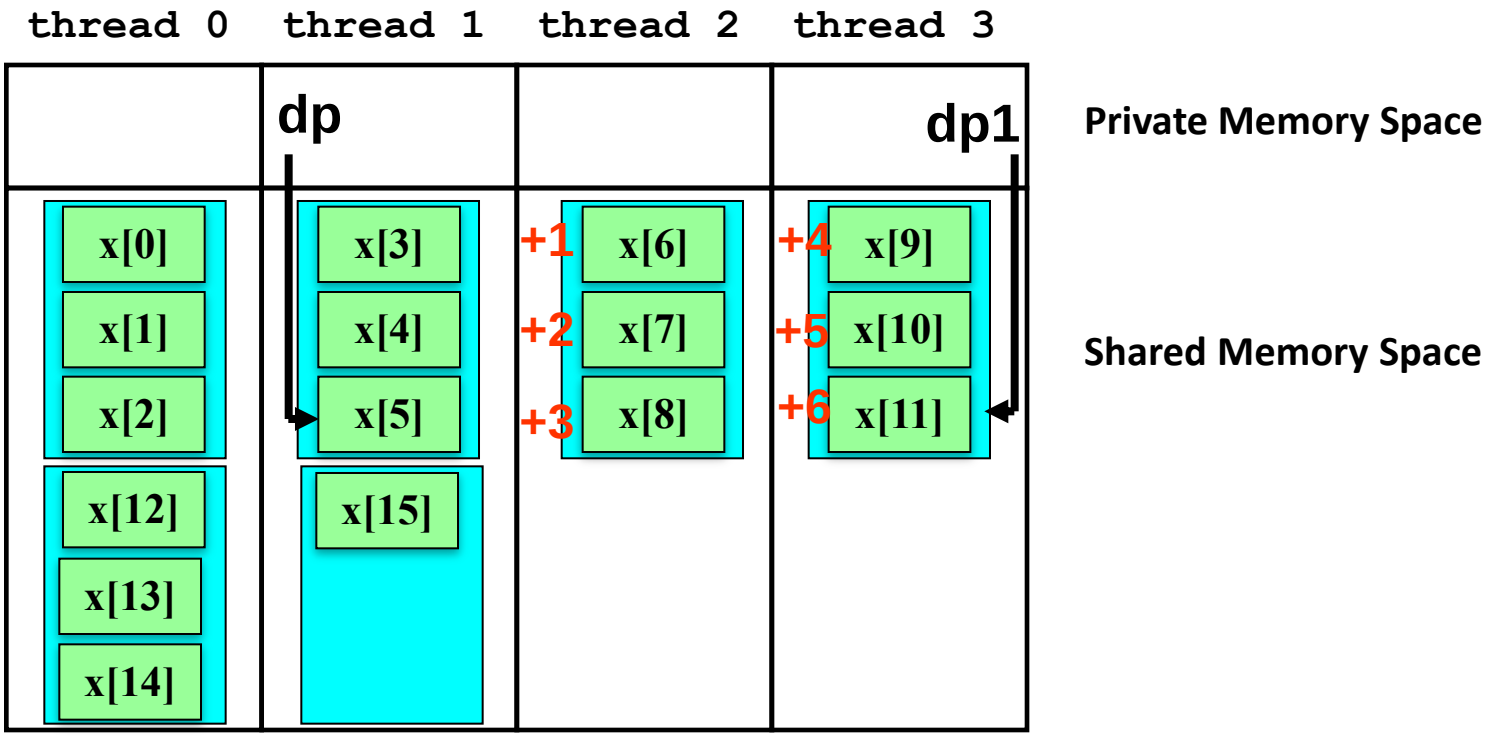
```
shared int x[16]; //shared int array
shared int *dp = &x[5], *dp1; // (private) pointers to shared int
dp1 = dp + 9; // default blocking factor 1
```



Pointer properties (2/3)

The pointer will follow its own blocking factor

```
shared [3] int x[16];
shared [3] int *dp = &x[5], *dp1;
dp1 = dp + 6; // blocking factor 3
```



Pointer properties (3/3)

- casting a shared pointer to a private pointer is allowed but not the other way around

- casting a shared pointer to private will result in loss of information
  - thread & phase

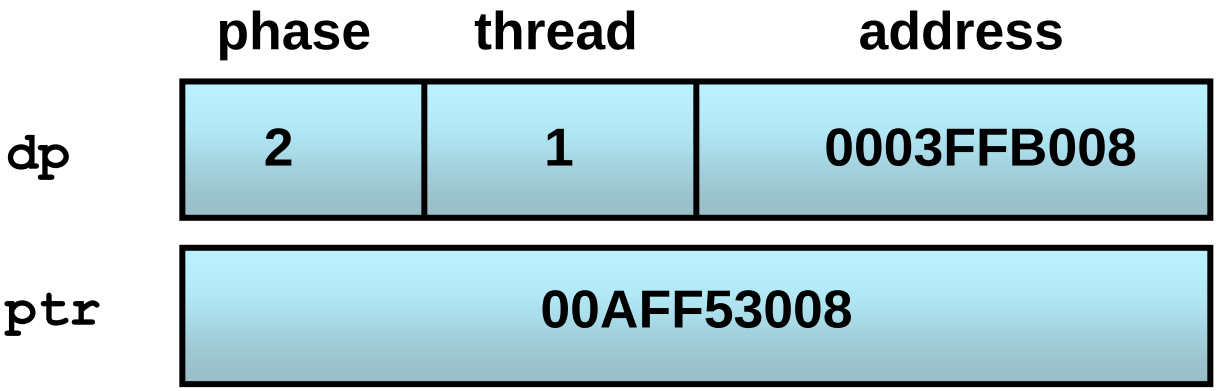
- casting is only *well defined* if the object pointed to by the shared pointer has local affinity

Pointer casting

Casting a shared pointer to a private pointer results in information loss

```
shared [3] int x[16];
shared int *dp = &x[5];
int *ptr;
ptr = (int *) dp;
```

➔ `ptr != upc_addrfield(dp)`



Dynamic memory allocation

so far only seen how to allocate memory statically

dynamic memory allocation in UPC is of course possible

- provides flexibility
- allows object sizes to changes during runtime

→ this is one of the cases where pointers are required

dynamic memory allocation in private space is done using standard C functions

dynamic memory allocation in shared space is achieved using special UPC functions

Dynamic memory allocation (2)

two types of memory allocation functions

- non-collective: **`upc_global_alloc`**, **`upc_alloc`**
- collective: **`upc_all_alloc`**

collective calls are called by ***all*** threads and return the same address value (pointer) to all of them

non-collective calls can be executed by multiple threads. Each call will allocate a different shared block.

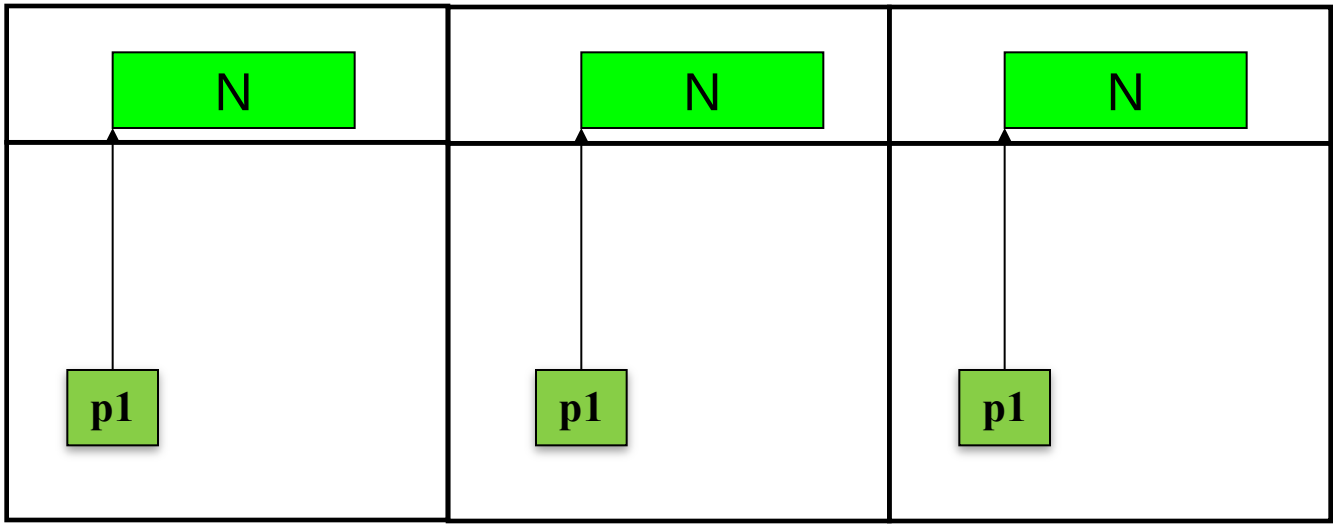
free the allocated memory using **`upc_free`**

- not a collective call

Non-collective dynamic allocation

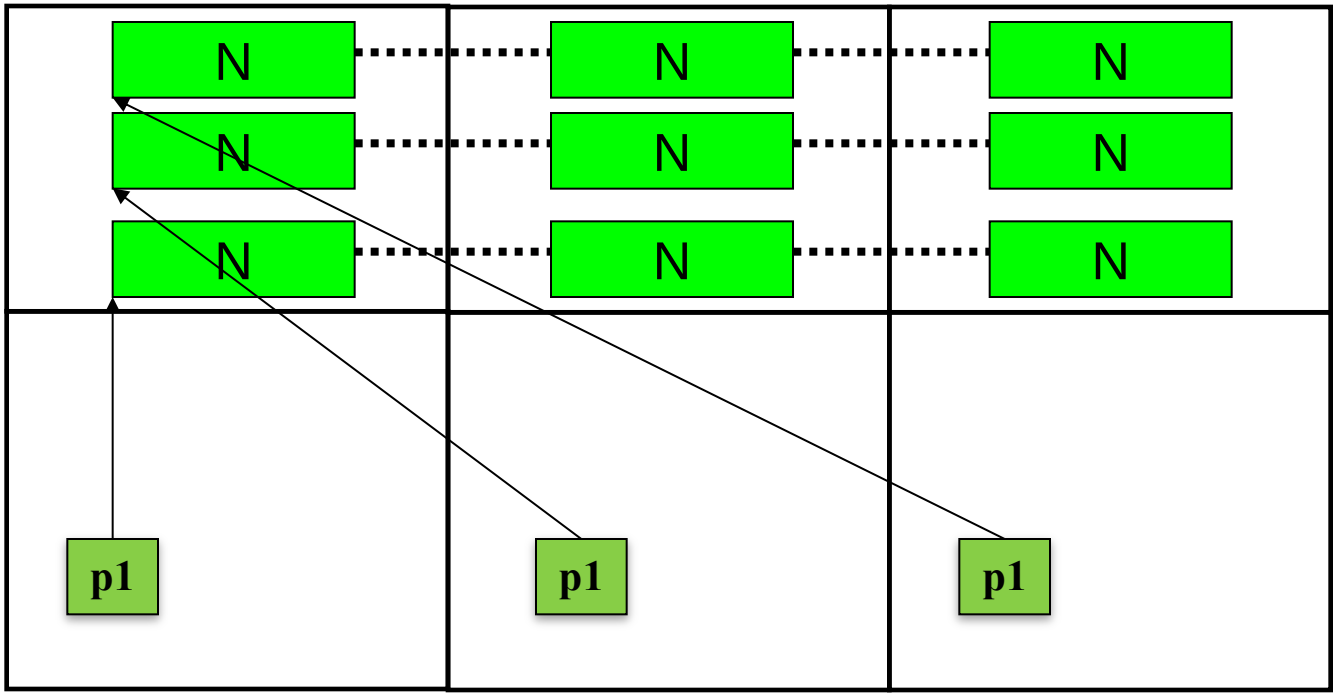
each thread allocates a memory block in its own shared memory space

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



Non-collective dynamic allocation (2)

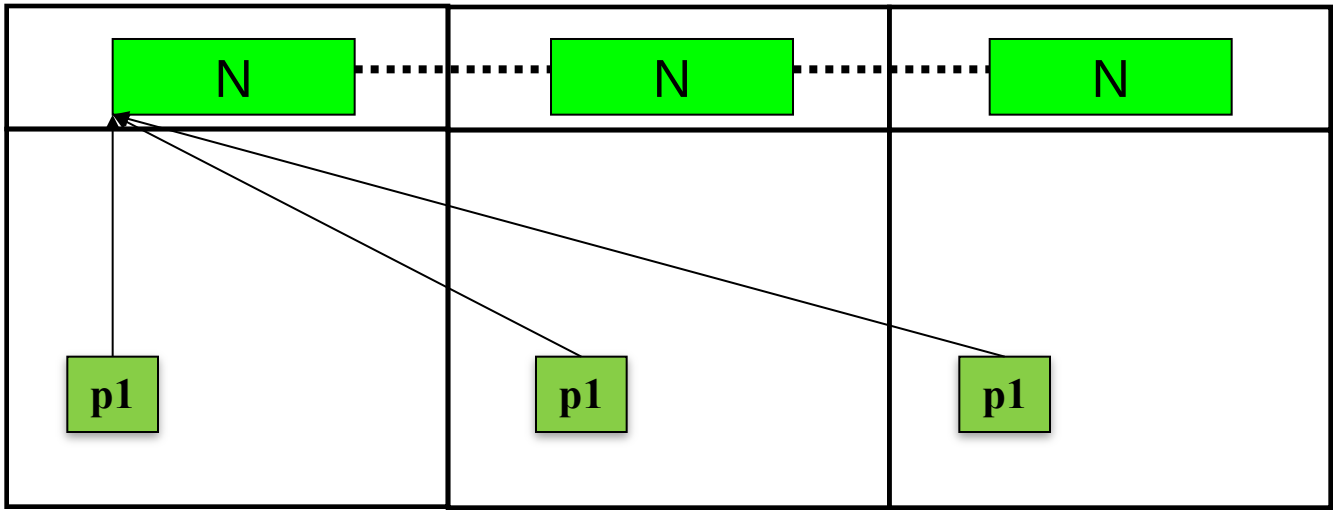
```
shared [N] int *ptr;
ptr = (shared [N] int *) upc_global_alloc(THREADS, N*sizeof(int));
```



Collective dynamic allocation

allocate contiguous segments of shared memory

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



Locks

Access control mechanism for critical sections

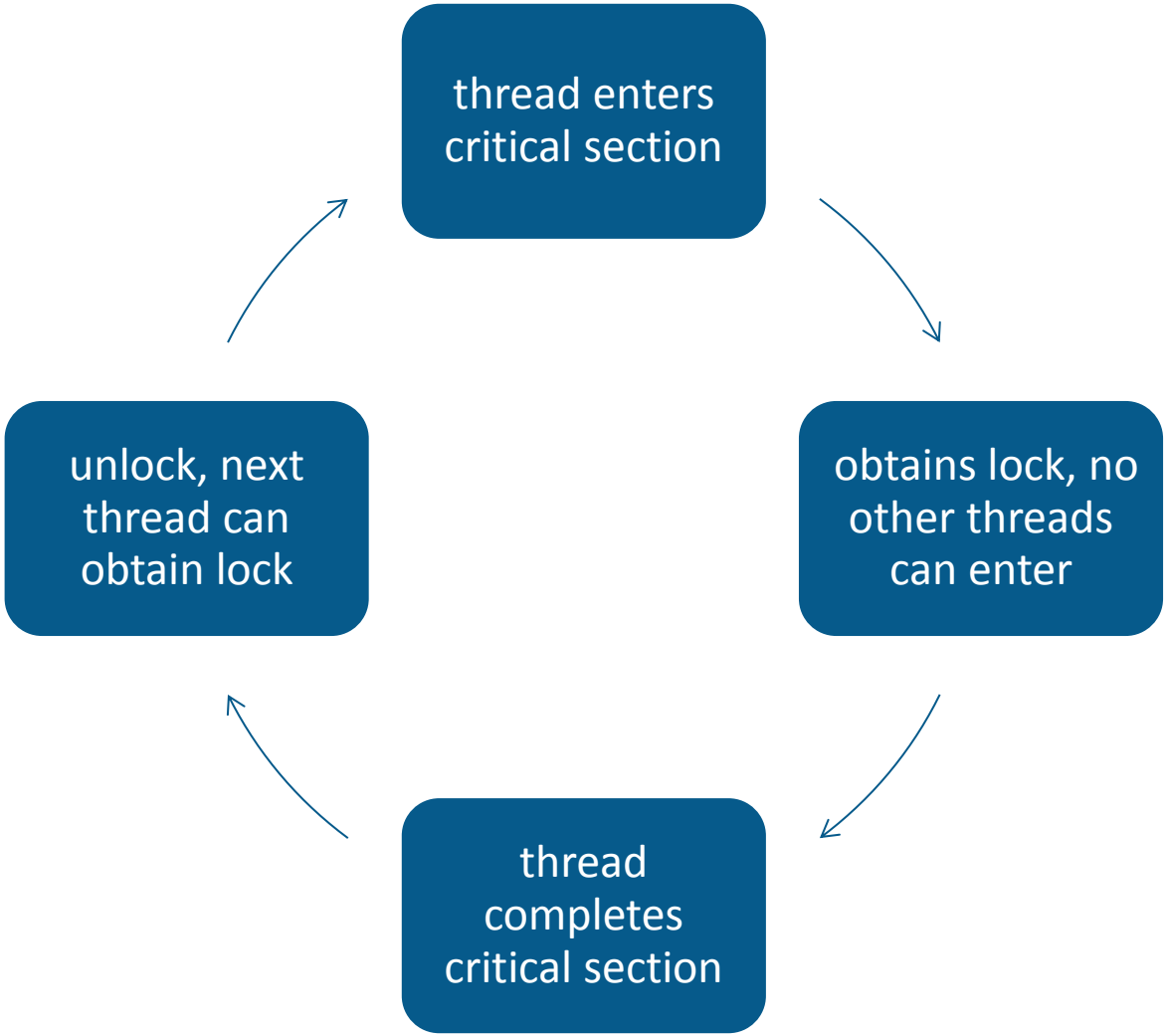
- Sections which should be executed by one thread at a time
- Serialised execution

UPC data type `upc_lock_t`

- Can have one of two states: locked or unlocked
- Can be seen by all threads → I mentally add **`shared`** in front of declarations
- ***`shared`*** **`upc_lock_t`**
- Technically an opaque type in C

Locks need to be manipulated through pointers

Program flow with locks



Creating locks

initial state of a new lock object is unlocked

locks can be created collectively

→ return value on every thread points to the same object

→ `upc_lock_t *upc_all_lock_alloc(void);`

or non-collectively

→ all threads that call the function obtain different locks

→ `upc_lock_t *upc_global_lock_alloc(void);`

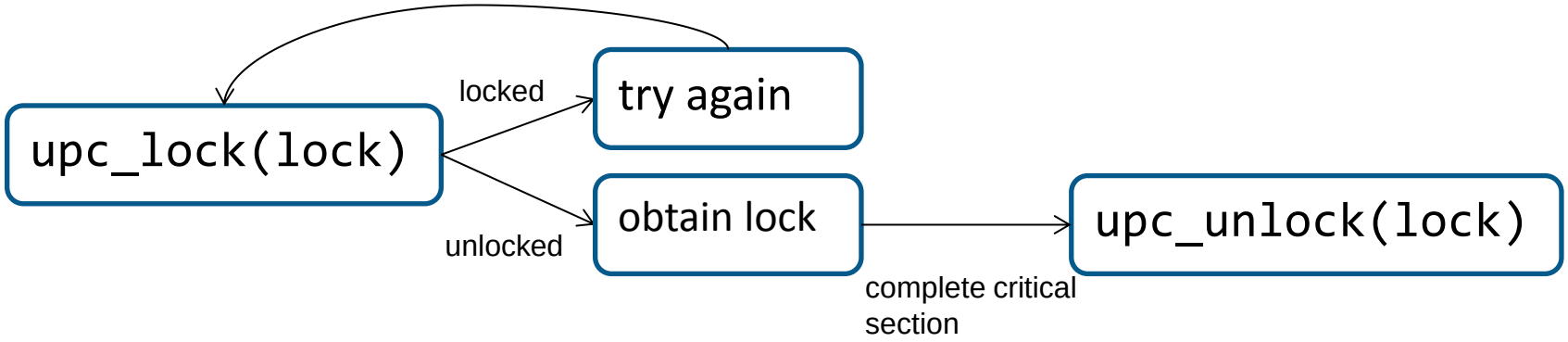
resources allocated by locks need to be freed

→ `upc_lock_free (upc_lock_t *ptr);`

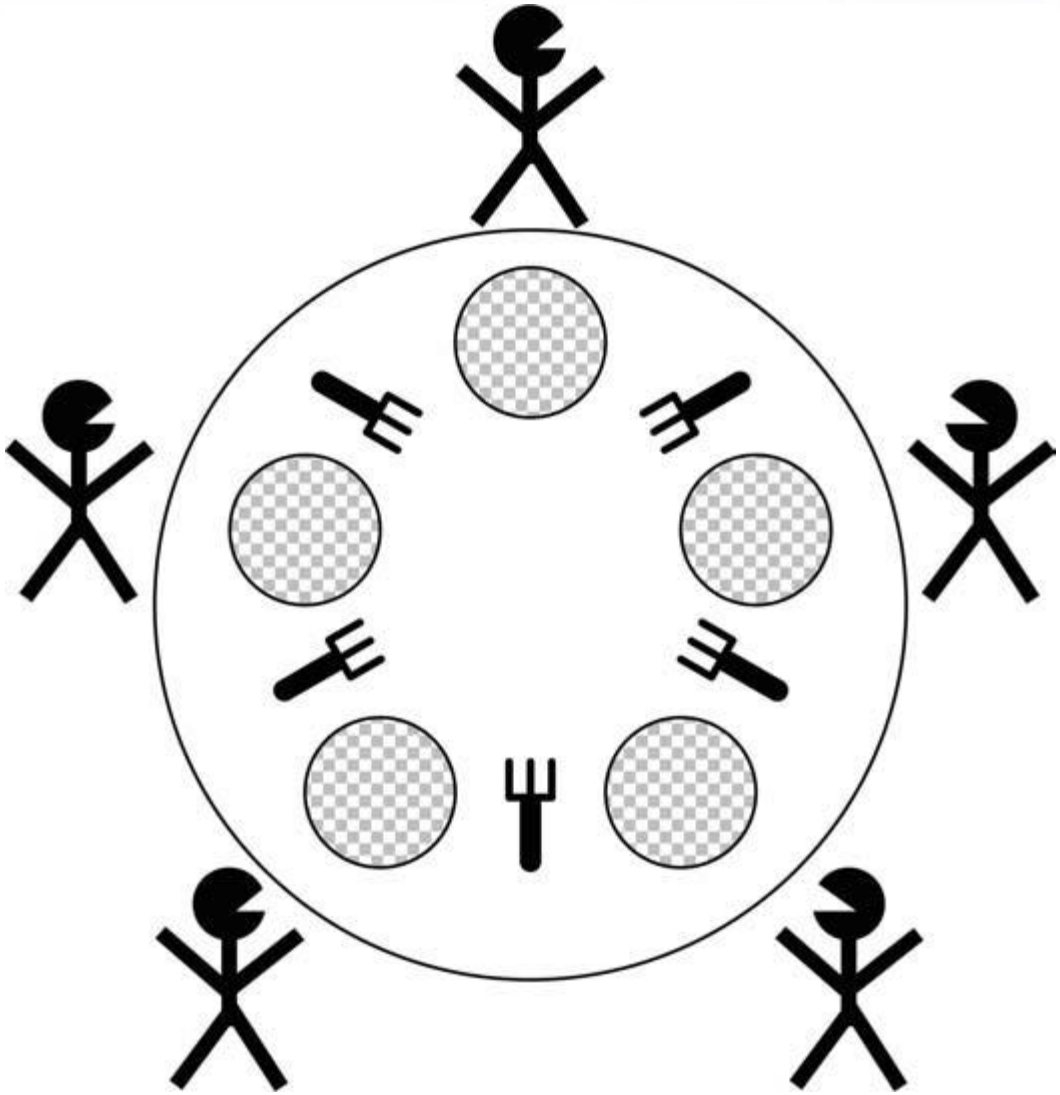
Using locks

threads need to lock and unlock

- → `upc_lock(upc_lock_t *ptr); // blocking`
- → `upc_lock_attempt(upc_lock_t *ptr); // non-blocking`
- → `upc_unlock(upc_lock_t *ptr);`



Example: Dining Philosophers (1/3)



Example: Dining Philosohpers (2/3)

Model the forks as a shared array of locks, then allocate a lock (fork) dynamically and return a pointer to it:

```
upc_lock_t *shared fork[THREADS]
fork[MYTHREAD]=upc_global_lock_alloc();
```

Now attempt to get the locks on either side:

```
left_fork=upc_lock_attempt(fork[MYHTREAD]);
right_fork=upc_lock_attempt(fork[(MYHTREAD+1)%THREADS]);
```

Example: Dining Philosophers (3/3)

If both forks are unlocked, lock them, eat, and release when finished.

```
upc_unlock(fork[MYTHREAD]);
upc_unlock(fork[(MYTHREAD+1)%THREADS]);
```

If only one fork is available, unlock (release) it and try again until two are available.

UPC collectives

supported by most compilers

- readable code
- but not necessarily optimised for performance

requires separate header file

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

Collectives

Two types of collective operations defined as part of the UPC standard specification:

① relocalisation collectives

`upc_all_broadcast`, `upc_all_scatter`, `upc_all_gather`, `upc_all_gather_all`, `upc_all_exchange`, `upc_all_permute`

② computational collectives

`upc_all_reduceT`, `upc_all_prefix_reduceT`, `upc_all_sort`

Supported operations: `UPC_SUM`, `UPC_MULT`, `UPC_AND`, `UPC_OR`, `UPC_XOR`, `UPC_LOGAND`, `UPC_LOGOR`, `UPC_MIN`, `UPC_MAX`

user specified functions supported via: `UPC_FUNC`, `UPC_NONCOMM_FUNC`

→ Calls to these functions must be performed by all threads

Computational collectives

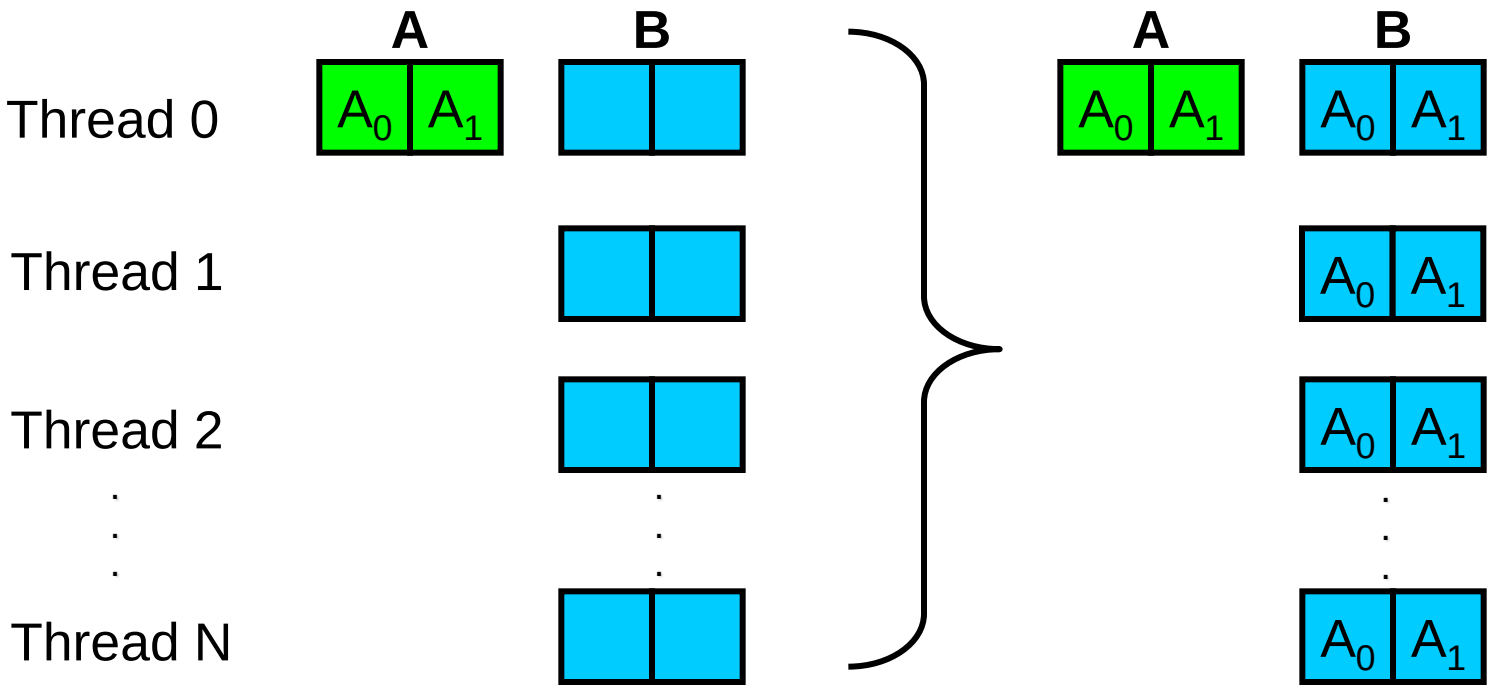
11 variations of the **upc_all_reduce*T*** and **upc_all_prefix_reduce*T***

→ ***T*** needs to be replaced with the type used in the reduction operation

T	Type	*T*	Type
C	signed char	**L**	signed long
UC	unsigned char	**UL**	unsigned long
S	signed short	**F**	Float
US	unsigned short	**D**	double
I	signed int	**LD**	long double
UI	unsigned int		

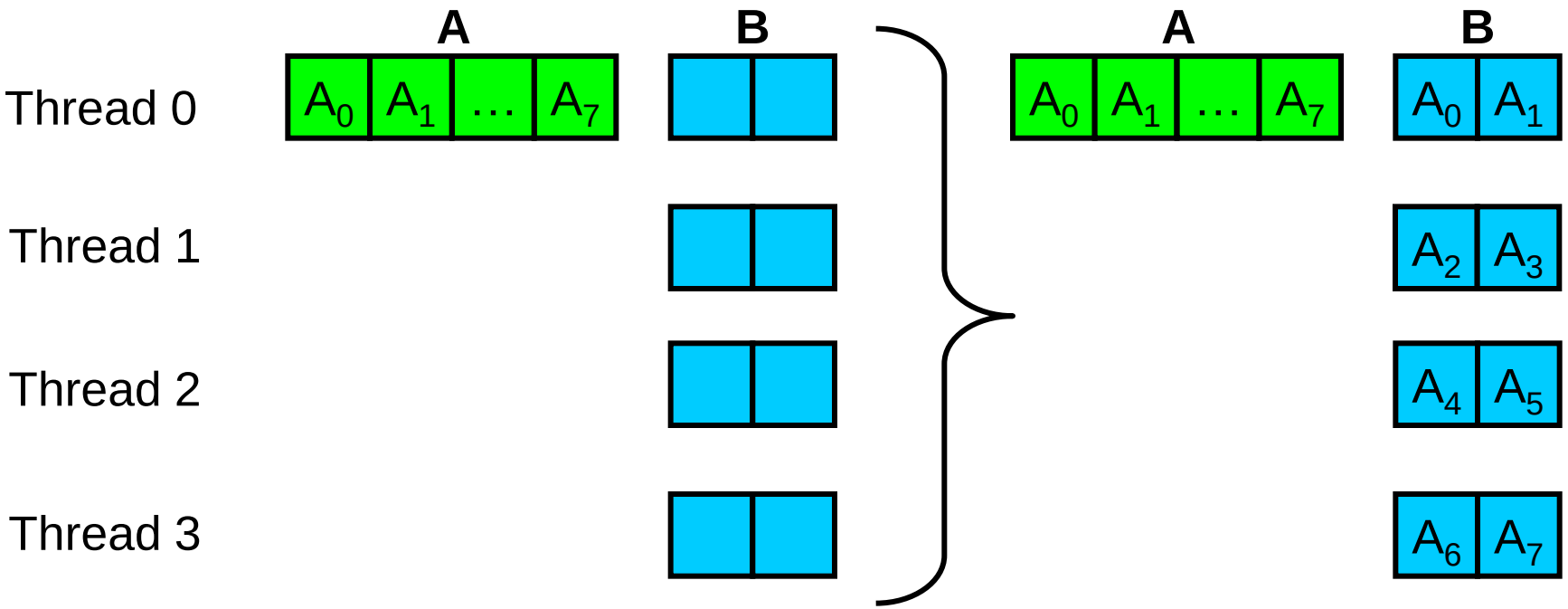
Broadcast

```
shared[]  int A[2];
shared[2] int B[8];

upc_all_broadcast(B, A, 2*sizeof(int), UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```



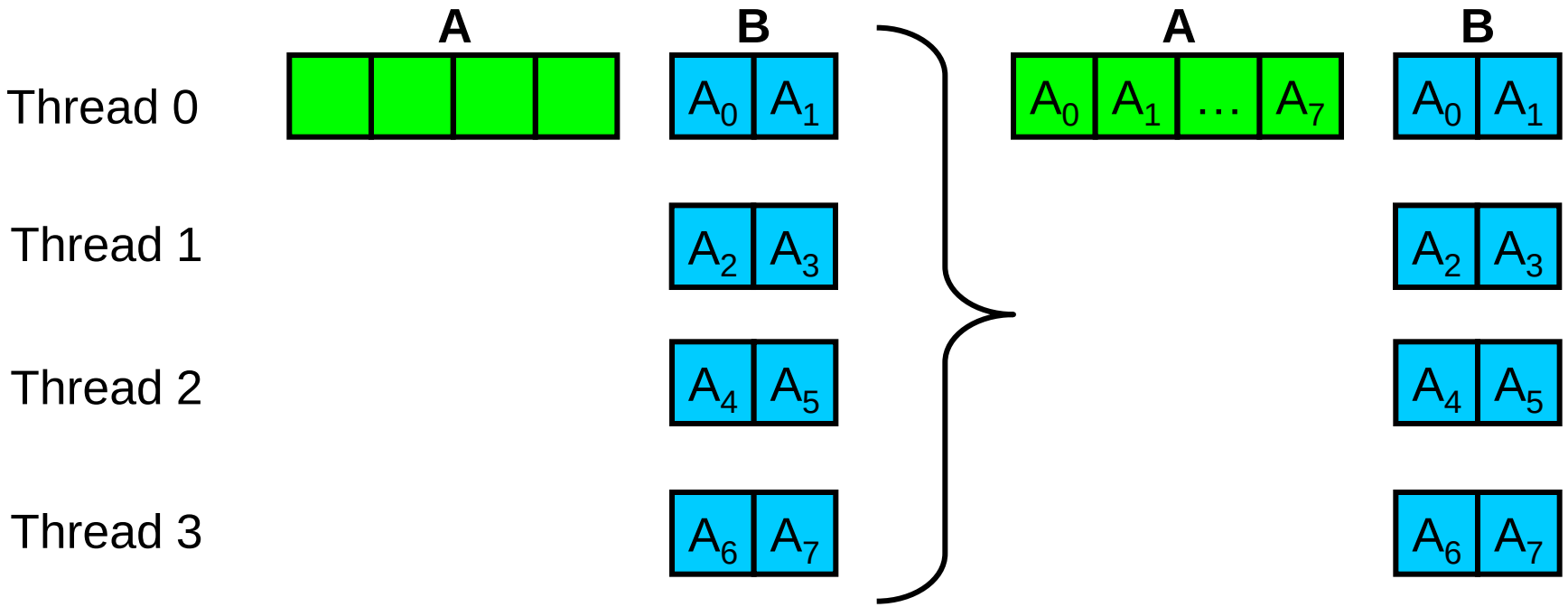
Scatter

```
shared[]  int A[8];
shared[2] int B[8];

upc_all_scatter(B, A, 2*sizeof(int), UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```



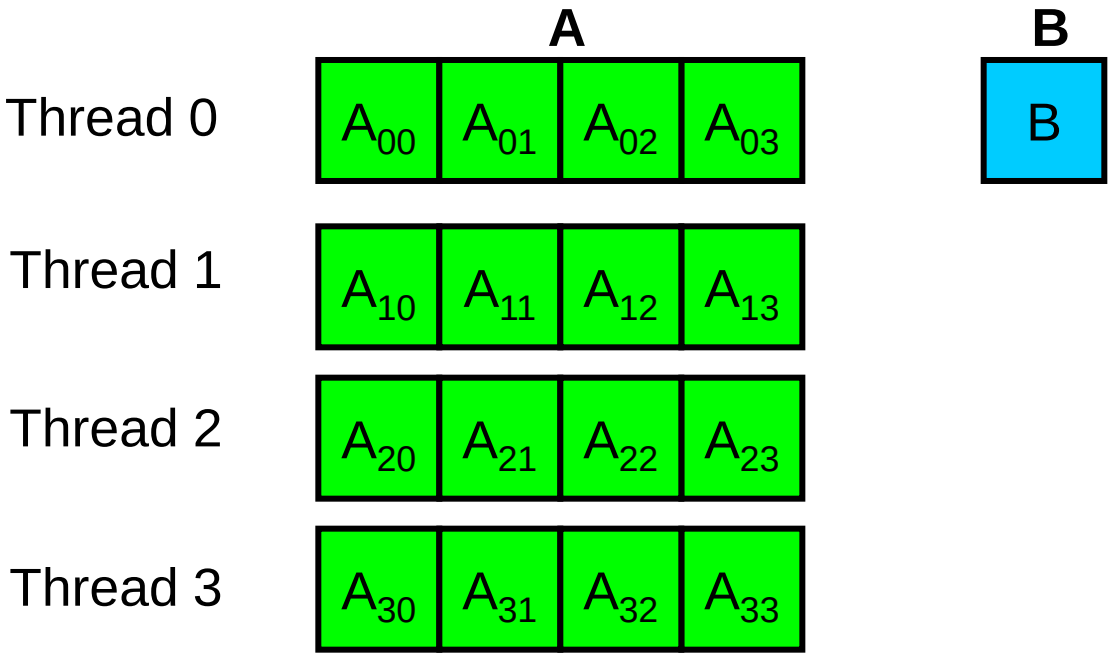
Gather

```
shared[]  int A[8];
shared[2] int B[8];

upc_all_gather(A, B, 2*sizeof(int), UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```

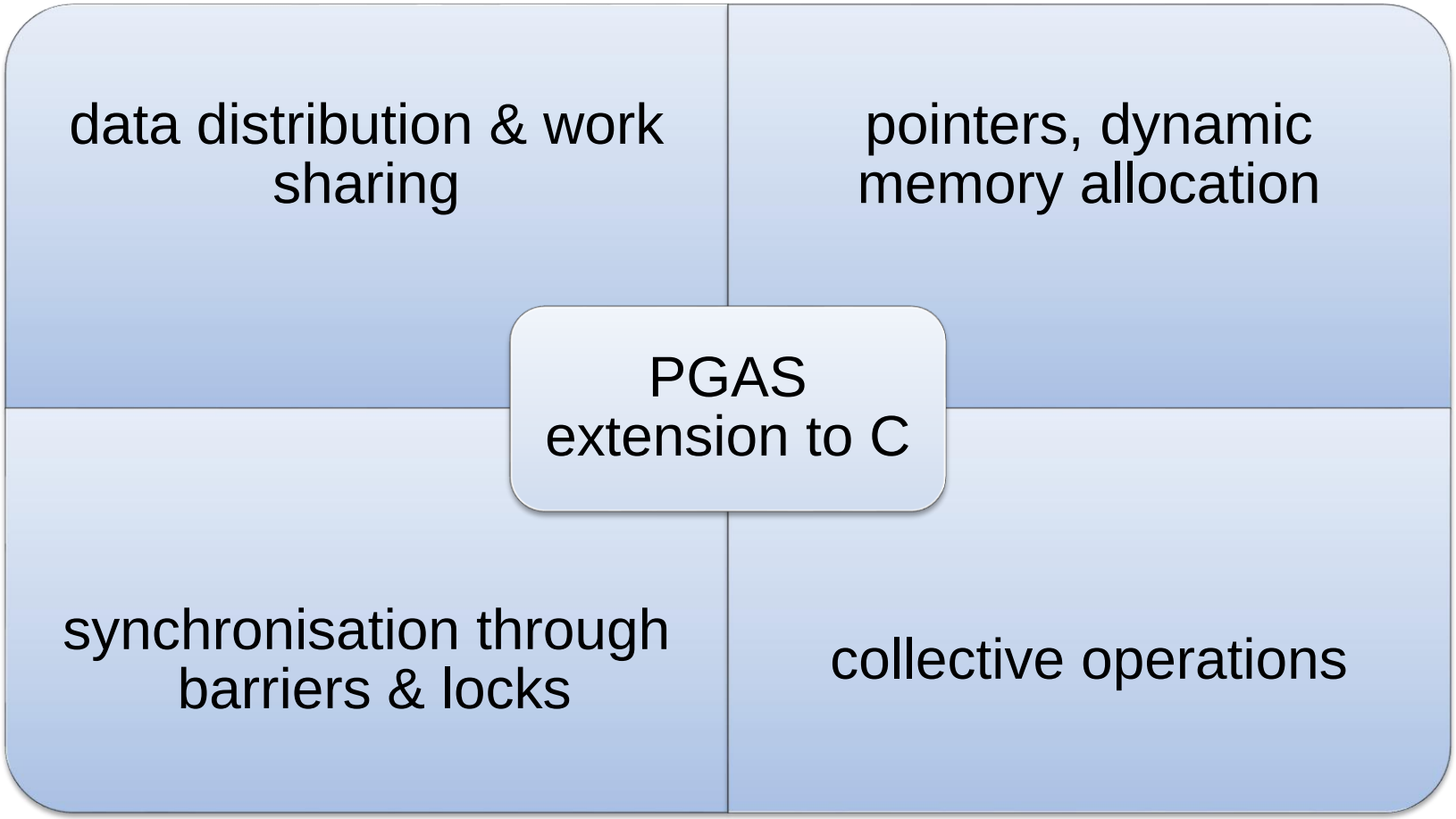


Reduction

```
shared[4] double A[16];
shared    double B;

upc_all_reduceD(&B, A, UPC_SUM, 4, 16, NULL, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
```



Final summary



References

http://upc.gwu.edu/documentation.html

- Language Specification (version 1.2)
- UPC Manual
- UPC Collective Operations Specification (version 1.0)