

Partitioned Global Address Space (PGAS) Model

Bin Bao

Contents

- PGAS model introduction
- Unified Parallel C (UPC) introduction
 - Data Distribution, Worksharing and Exploiting Locality
 - Synchronization and Memory Consistency
- Implementation and performance issue
 - GASNet
 - A case study: FFT on BlueGene/P
- Summary

Programming Models

Shared Memory \dashrightarrow Shared Memory \dashrightarrow ~~Shared Memory~~
Message Passing



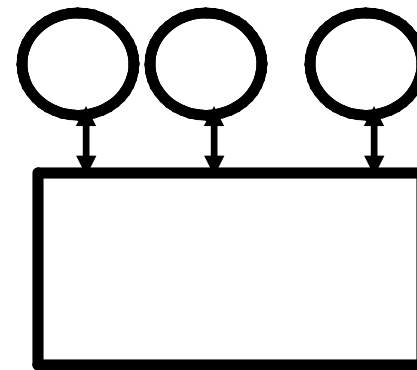
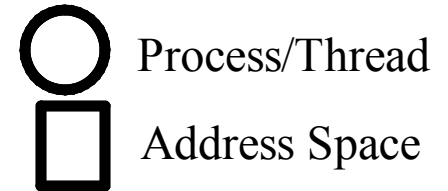
CPU: 10+
SMP

100+
Hardware: Sun Fireplane
Software: TreadMarks

1000+
Blue Gene/L

Shared Memory

- Hard to scale

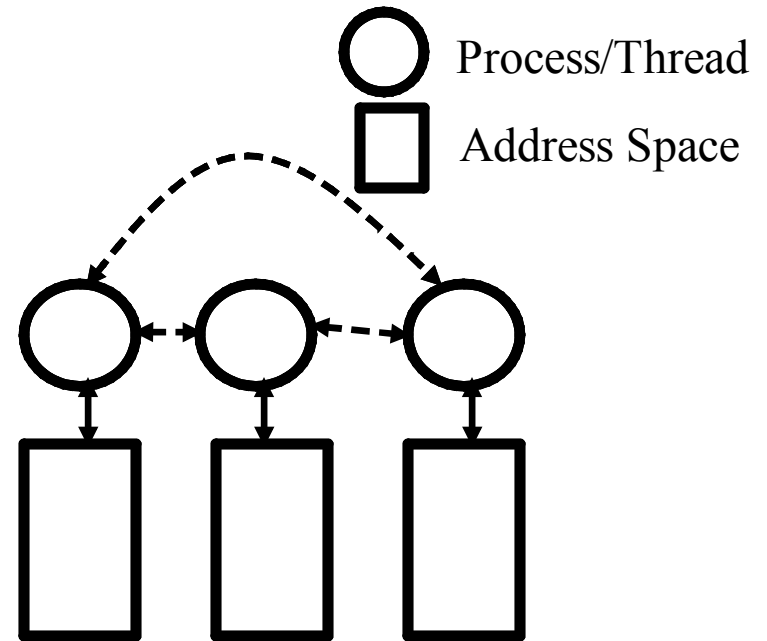


Shared Memory

OpenMP, Pthread

Message Passing

- Hard to program
- Example: look at our course discussion board for Programming Assignment 3



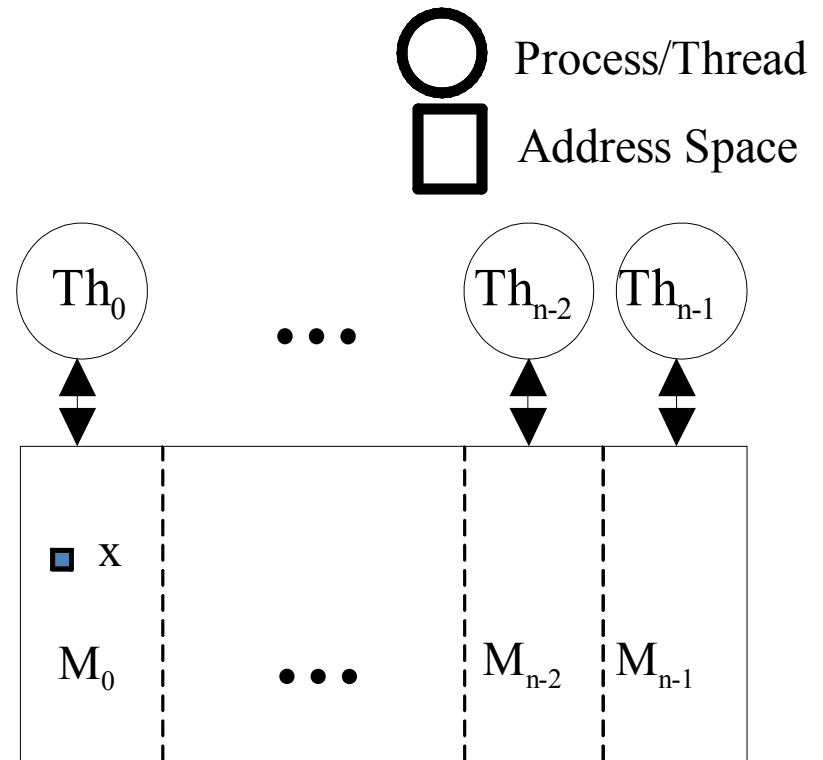
MPI

Some Questions

- Is shared memory a good programming model for distributed memory system?
- Do we really want to hide the fact that the memory access is non-uniform?
- Is there any model between shared memory and message passing?

Partitioned Global Address Space Model

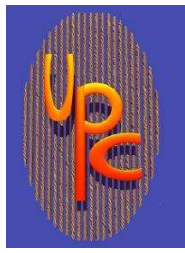
- Aka the DSM model
- Concurrent threads with a partitioned shared space
 - Similar to the shared memory
 - Memory partition M_i has affinity to thread Th_i
- Pros:
 - Helps exploiting locality
 - Simple statements as SM
- Cons:
 - Synchronization



UPC, CAF, Titanium

PGAS Languages

- Unified Parallel C (UPC)
 - Extension to ISO C
- Co-Array Fortran (CAF)
 - Extension to Fortran 95
- Titanium
 - Extension to Java

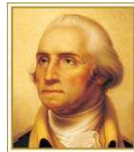


SC|05 Tutorial

High Performance Parallel Programming with Unified Parallel C (UPC)

Tarek El-Ghazawi

tarek@gwu.edu



The George Washington U.



Technological U.

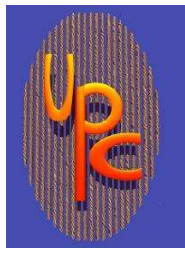
Phillip Merkey

Steve Seidel

{merk,steve}@mtu.edu

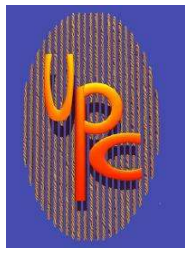
MichiganTech





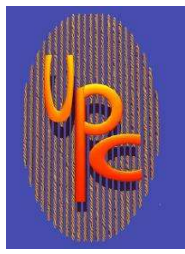
Introduction

- UPC – Unified Parallel C
- Set of specs for a parallel C
 - C v1.0 completed February of 2001
 - C v1.1.1 in October of 2003
 - C 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, ...



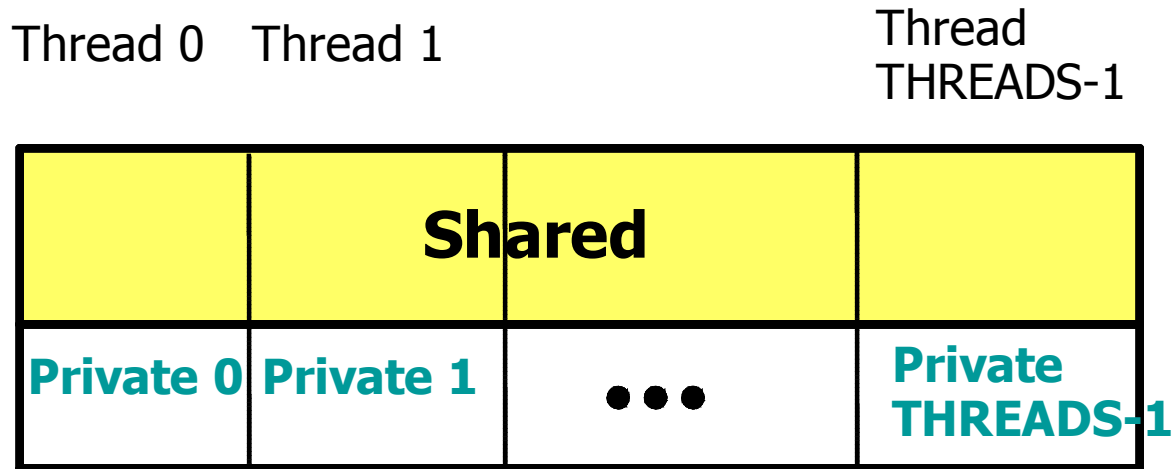
UPC Execution Model

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



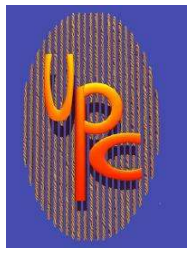
UPC Memory Model

Partitioned
Global
address space
Private
Spaces



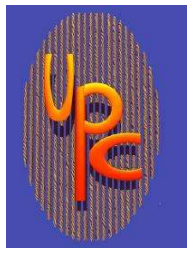
- A pointer-to-shared can reference all locations in the shared space, but there is data-thread **affinity**
- A private pointer may reference addresses in its private space or its local portion of the shared space
- Static and dynamic memory allocations are supported for both shared and private memory





Shared and Private Data

- Shared objects placed in memory based on affinity
- Shared scalars always have affinity to thread 0
- Shared arrays are spread over the threads in cyclic way by default



Shared and Private Data

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

will result in the following data layout:

Thread 0

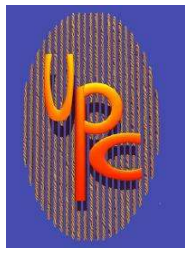
A[0][0]
A[1][0]
A[2][0]
A[3][0]

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]

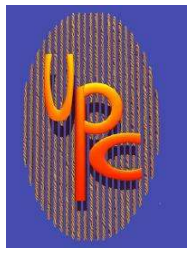


Blocking of Shared Arrays

- 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:

```
C shared [block-size] type array[N];
```

```
R 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

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

Thread 1

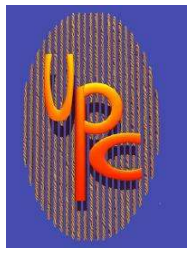
A[0][3]
A[1][0]
A[1][1]
A[3][3]

Thread 2

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

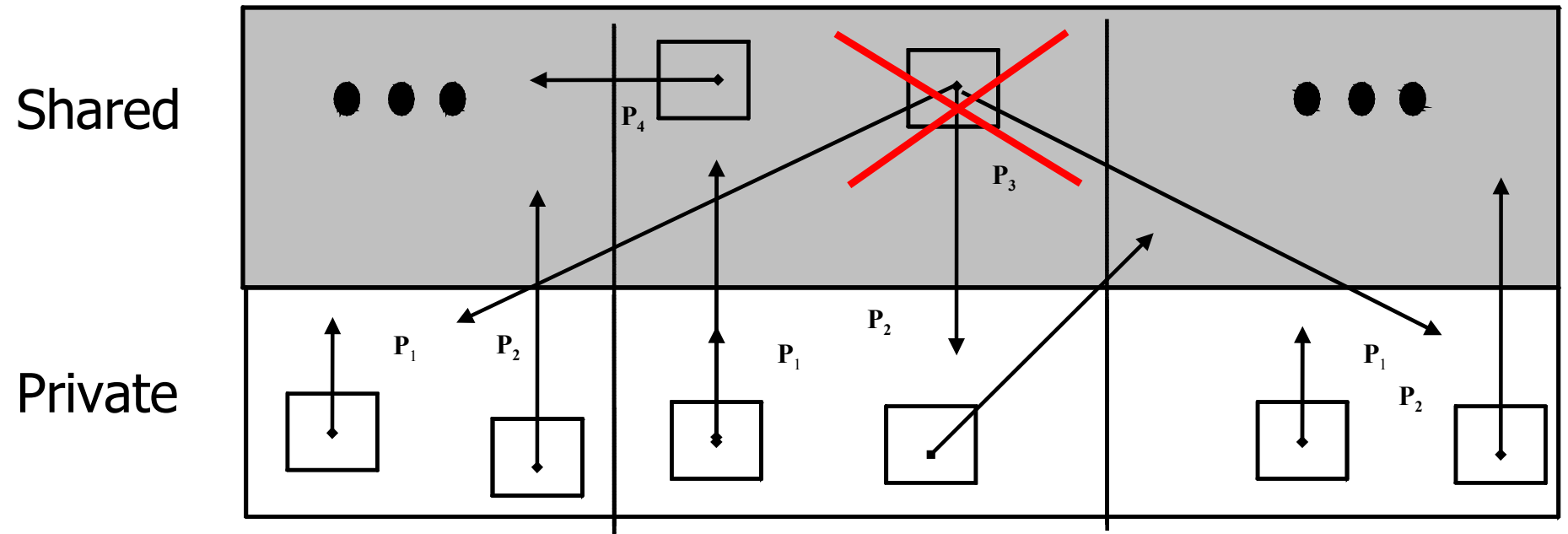
Thread 3

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



UPC Pointers

Thread 0





UPC Pointers

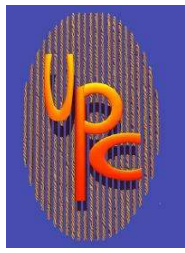
- What are the common usages?

`C int *p1; /* access to private data or to local shared data, just like C pointer */`

`C shared int *p2; /* independent access of threads to data in shared space */`

`C int *shared p3; /* not recommended*/`

`C shared int *shared p4; /* build shared linked structure */`



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
```

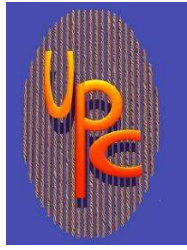
C Affinity could be an integer expression, or a

C Reference to (address of) a shared object

- Programmer indicates the iterations are independent

C Undefined if there are dependencies across threads





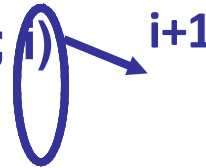
Work Sharing and Exploiting Locality via `upc_forall()`

- 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];
```

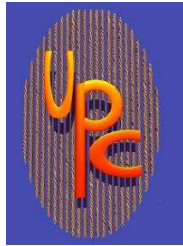
- Example 2: implicit affinity with integer expressions and distribution in a round-robin fashion

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



Correct but slow

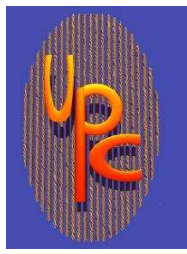
Note: Examples 1 and 2 result in the same distribution



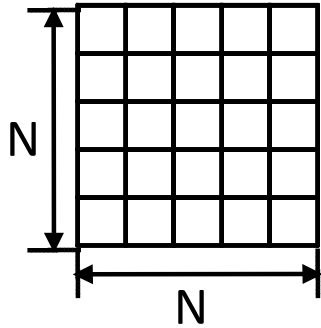
Work Sharing: upc_forall()

- 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**

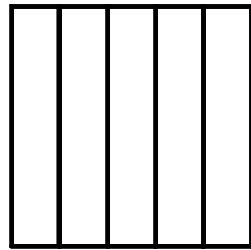
i	i*THREADS	i*THREADS/100
0..24	0..96	0
25..49	100..196	1
50..74	200..296	2
75..99	300..396	3



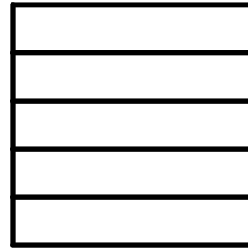
Distributing Multidimensional Data



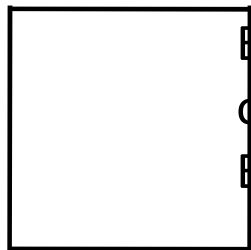
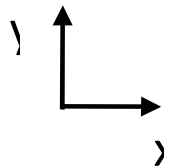
default
BLOCKSIZE=1



Column Blocks
BLOCKSIZE=
N/THREADS



Distribution by
Row Block
BLOCKSIZE=N



BLOCKSIZE=N*N
or
BLOCKSIZE = infinite

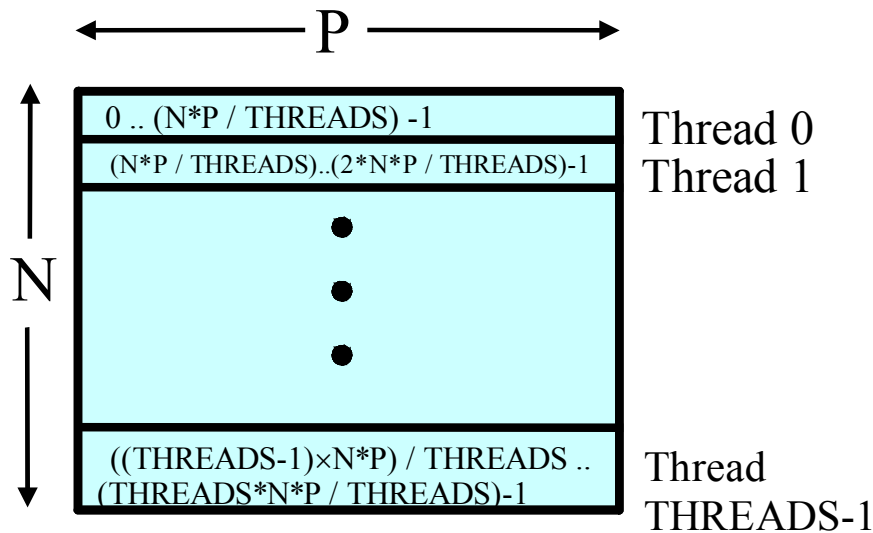
- Uses the inherent contiguous memory layout of C multidimensional arrays
- `shared [BLOCKSIZE] double grids[N][N];`
 - Distribution depends on the value of BLOCKSIZE,



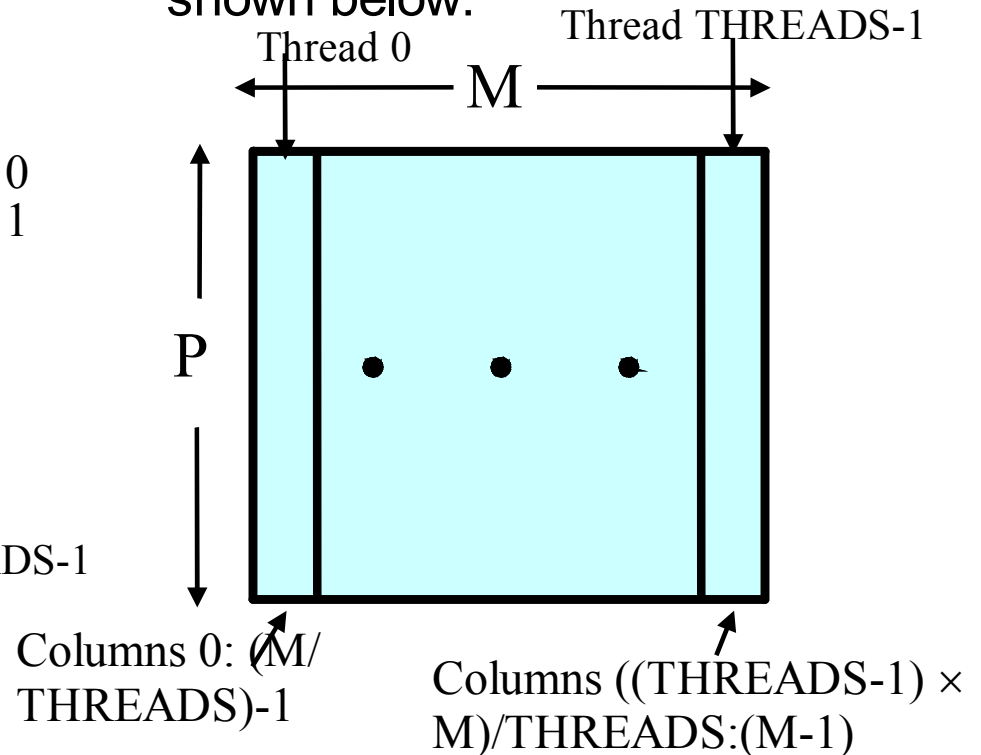
Matrix Multiplication in UPC

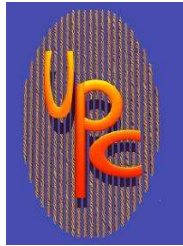
Exploiting locality in matrix multiplication

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



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



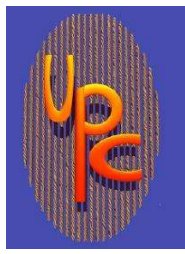


UPC Matrix Multiplication Code

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

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

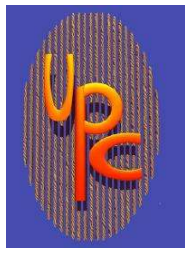
    upc_forall(i = 0 ; i<N ; i++; &c[i][0]) {
        for (j=0 ; j<M ;j++) {
            c[i][j] = 0;
            for (l= 0 ; l<P ; l++) c[i][j] += a[i][l]*b[l][j];
        }
    }
}
```



Another Example: Gaussian Elimination in UPC (cyclic distribution)

```
//static array
shared [nsize] double matrix[nsize][nsize];
shared double B[nsize];
int i, j; //private

for(i = 0; i < nsize; i++){
    if(MYTHREAD == i% THREADS) getPivot(nsize, i);
    upc_barrier;
    upc_forall (j=i+1; j< nsize; j++; &matrix[j][0]) {
        pivotVal = matrix[j][i];
        matrix[j][i] = 0.0;
        for (k = i + 1 ; k < nsize; k++){
            matrix[j][k] -= pivotVal * matrix[i][k];
        }
        B[j] -= pivotVal * B[i];
    }
    upc_barrier;
}
```



Synchronization - Barriers

- No implicit synchronization among the threads
- UPC provides the following barrier synchronization constructs:

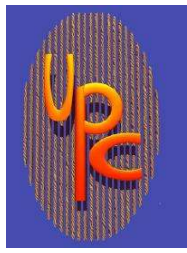
C Barriers (Blocking)

- `upc_barrier expropt;`

C Split-Phase Barriers (Non-blocking)

- `upc_notify expropt;`
- `upc_wait expropt;`

Note: `upc_notify` is not blocking `upc_wait` is



Synchronization - Locks

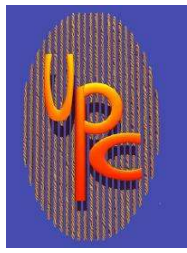
- In UPC, shared data can be protected against multiple writers :

```
C void upc_lock(upc_lock_t *l)
```

```
C int upc_lock_attempt(upc_lock_t *l) //returns 1 on  
    success and 0 on failure
```

```
C void upc_unlock(upc_lock_t *l)
```

- Locks are allocated dynamically, and can be freed
- Locks are properly initialized after they are allocated



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**

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

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



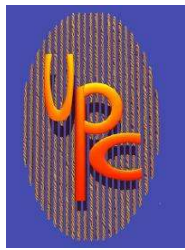


Lock freeing

- **Lock freeing**

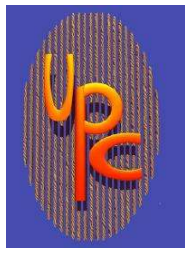
```
void upc_lock_free(upc_lock_t *l);
```

- This is not a collective function



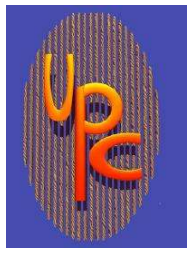
Memory Consistency Models

- 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 Can Be Controlled

- Variable declaration: Type qualifiers, *strict* or *relaxed*
- Statement level: *#pragma upc strict* or *#pragma upc relaxed*
- Program level: *#include <upc_strict.h>* or *#include <upc_relaxed.h>*
- Declarations have the highest precedence, then pragmas, then program level.



Memory Consistency- Fence

- UPC provides a fence construct
 - C Equivalent to a null strict reference, and has the syntax
 - `upc_fence;`
 - C 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  
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.

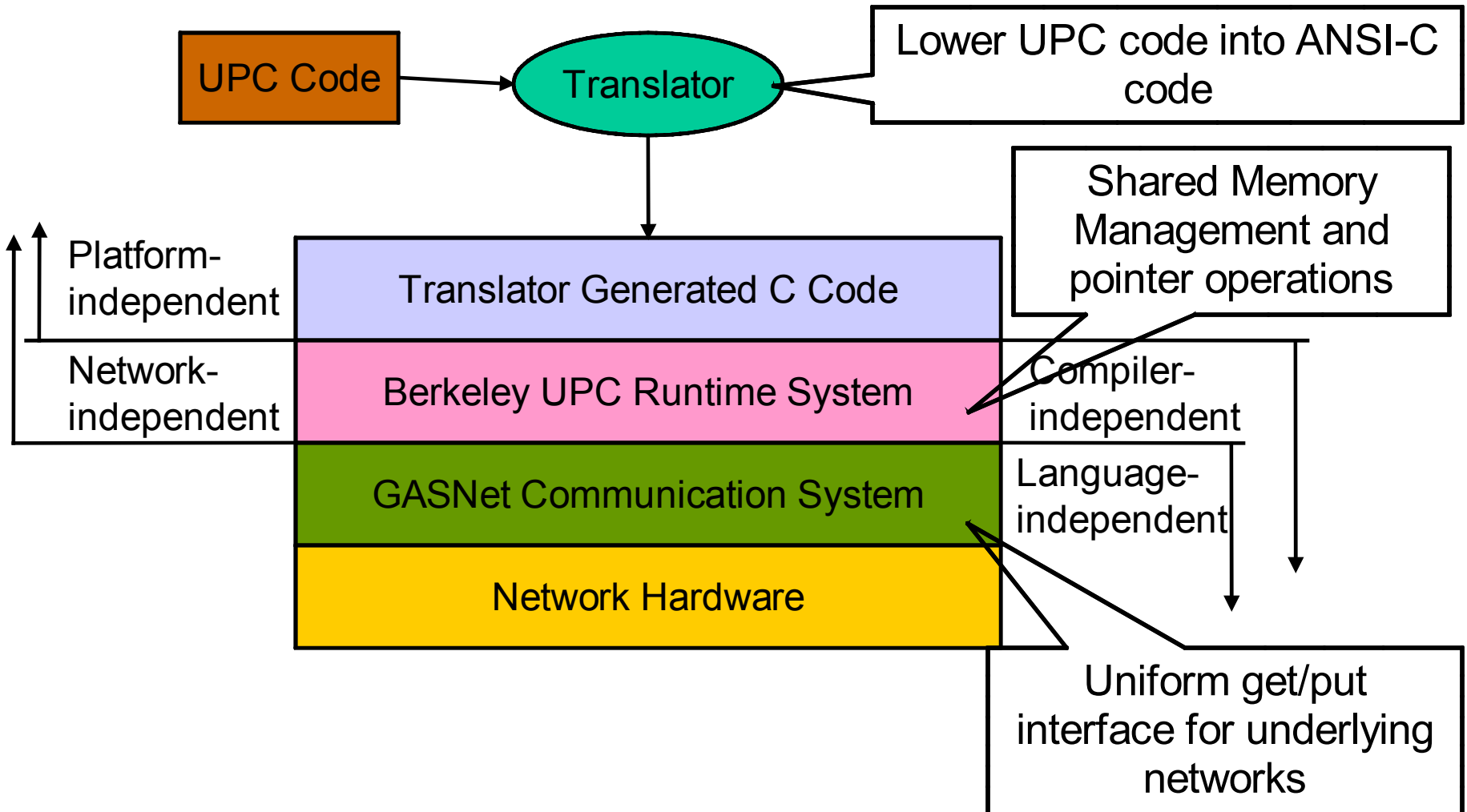
We could have used a fence in the same places. Affects shared references at all threads!

The above works as an example of point to point synchronization.

Implementation and Performance Issue

Scaling Communication-Intensive
Applications on BlueGene/P Using One-
Sided Communication and Overlap

Overview of the Berkeley UPC Compiler



Communication Matters

- Accessing global data on a remote memory requires communication
- Network speed is much slower than the CPU speed

Problem of MPI Two-Sided Communication

- Message sending and receiving must be matched to complete a transfer.
- Involving both send and receive sides increases the communication latency, and reduces the communication/computation overlapping ability.
- Point-to-point message ordering need to be guaranteed, probably by the software.

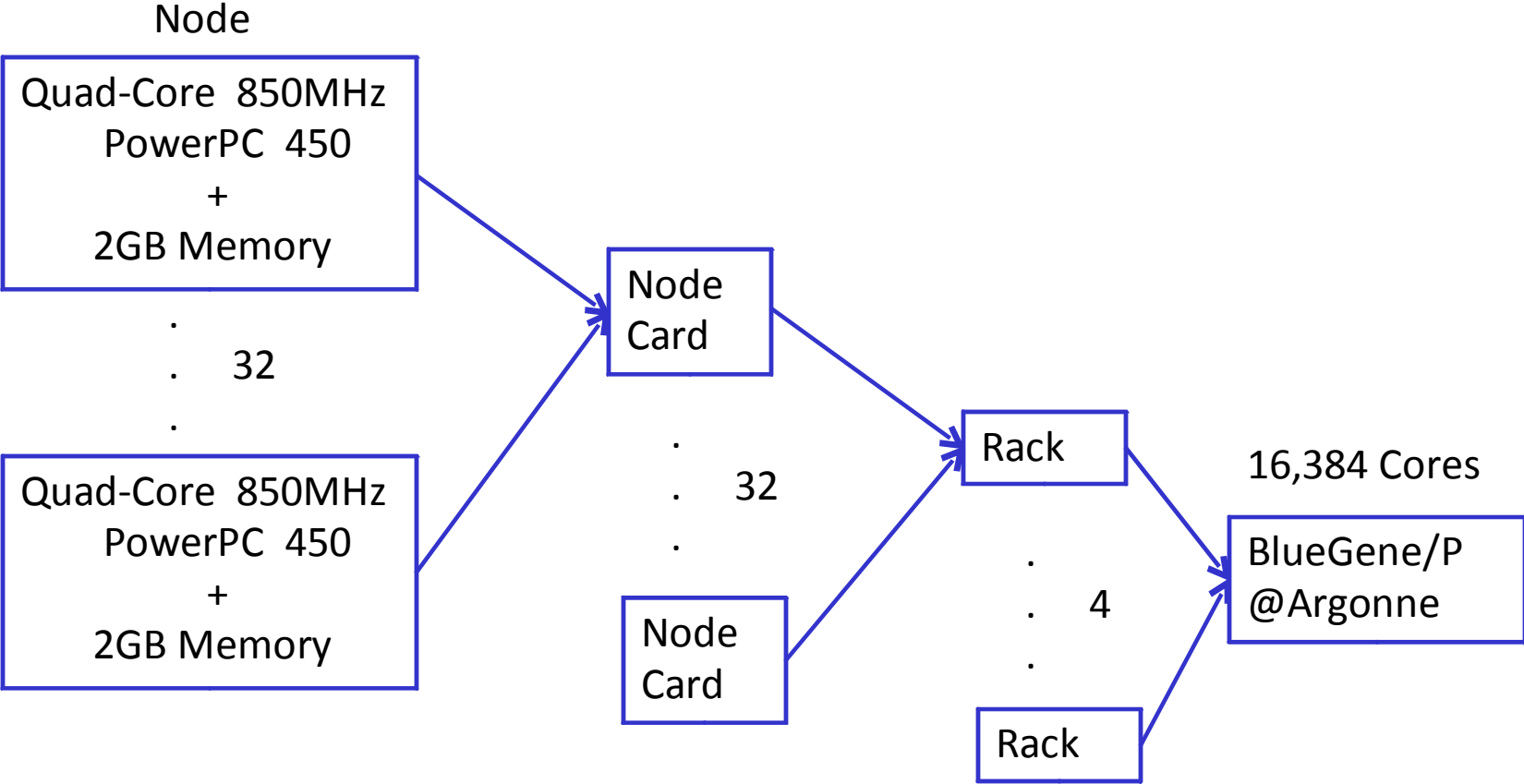
One-Sided Communication

- Naturally fit to modern cluster networks
 - Remote Direct Memory Access (RDMA)
 - Save CPU resources
 - Relax point-to-point ordering constraint
 - Better communication/computation overlap

GASNet

- A portable, language-independent communication interface
- Some primitives
 - One-sided communication: Get, Put
 - Active messages: lightweight, event-based, handler in the message

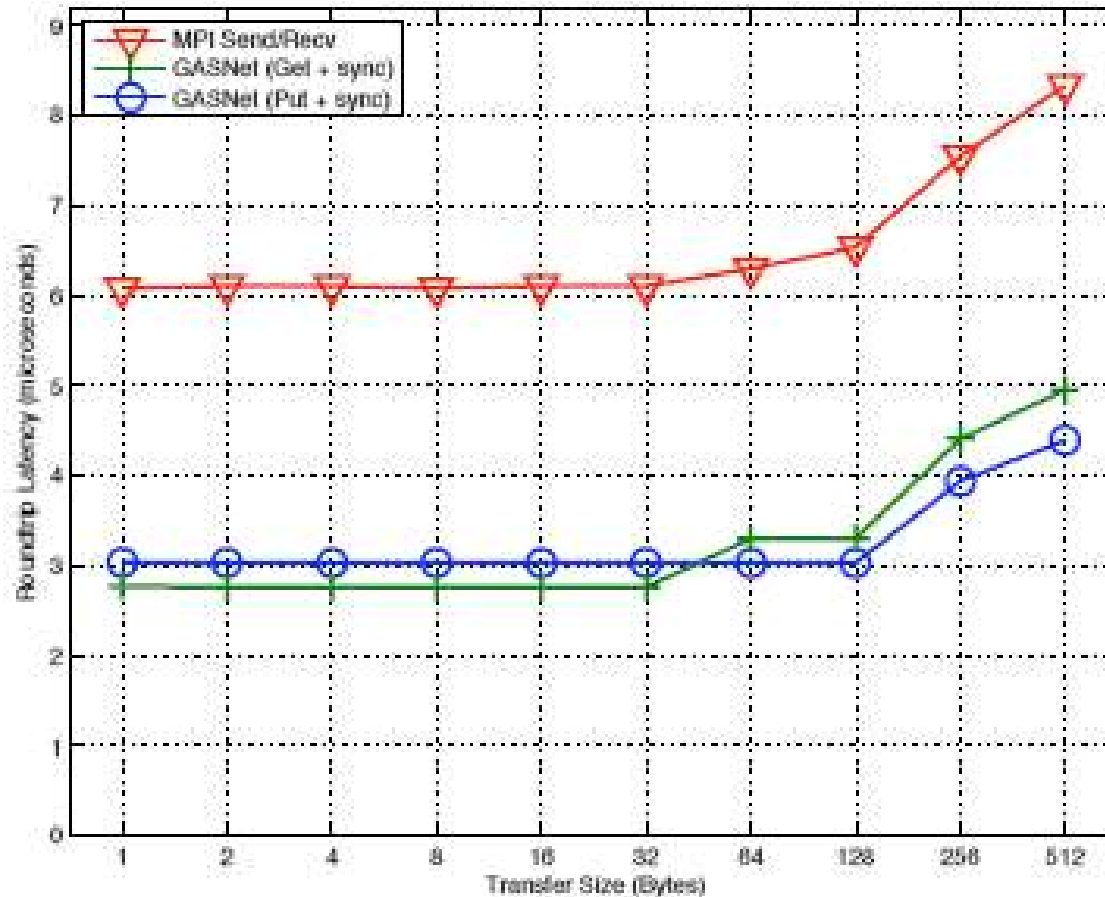
BlueGene/P



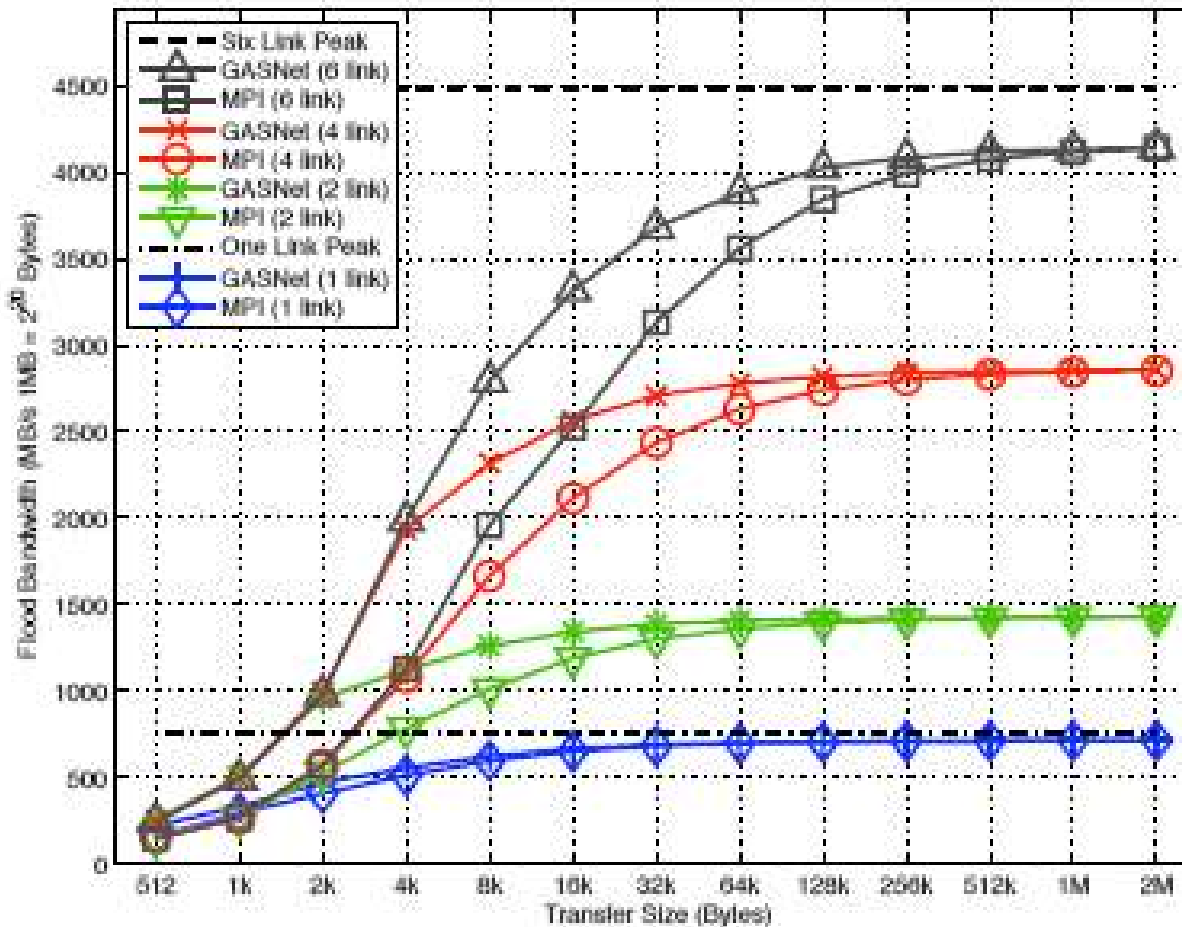
Communication in BlueGene/P

- 3D torus, 425MB/s each link direction
- Deep Computing Messaging Framework (DCMF)
 - Put and Get in GASNet can be implemented directly over DCMF_Put() and DCMF_Get()
 - Active Message: DCFM_Send()
- Lacking hardware assistance, MPI message matching brings overhead to the relatively weak core

Roundtrip Latency Comparison



Flood Bandwidth Comparison



Decompositions for NAS FT Algorithm

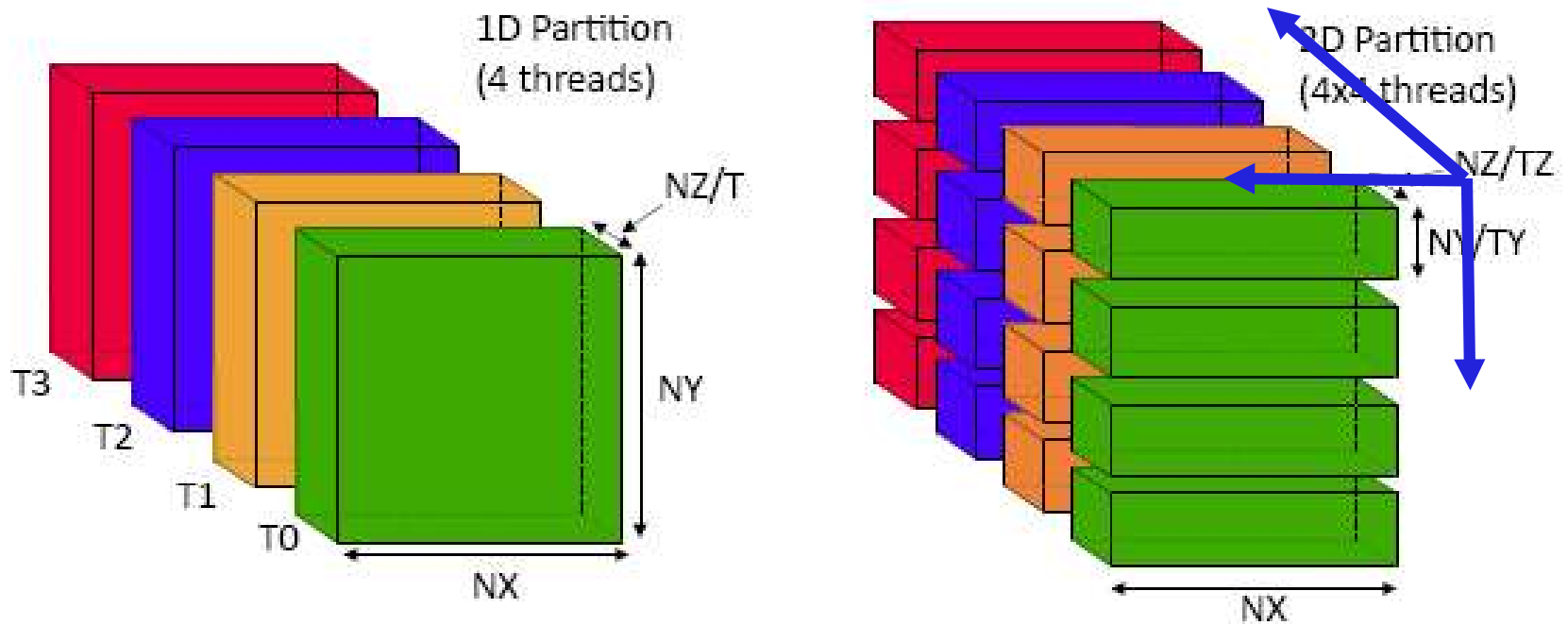
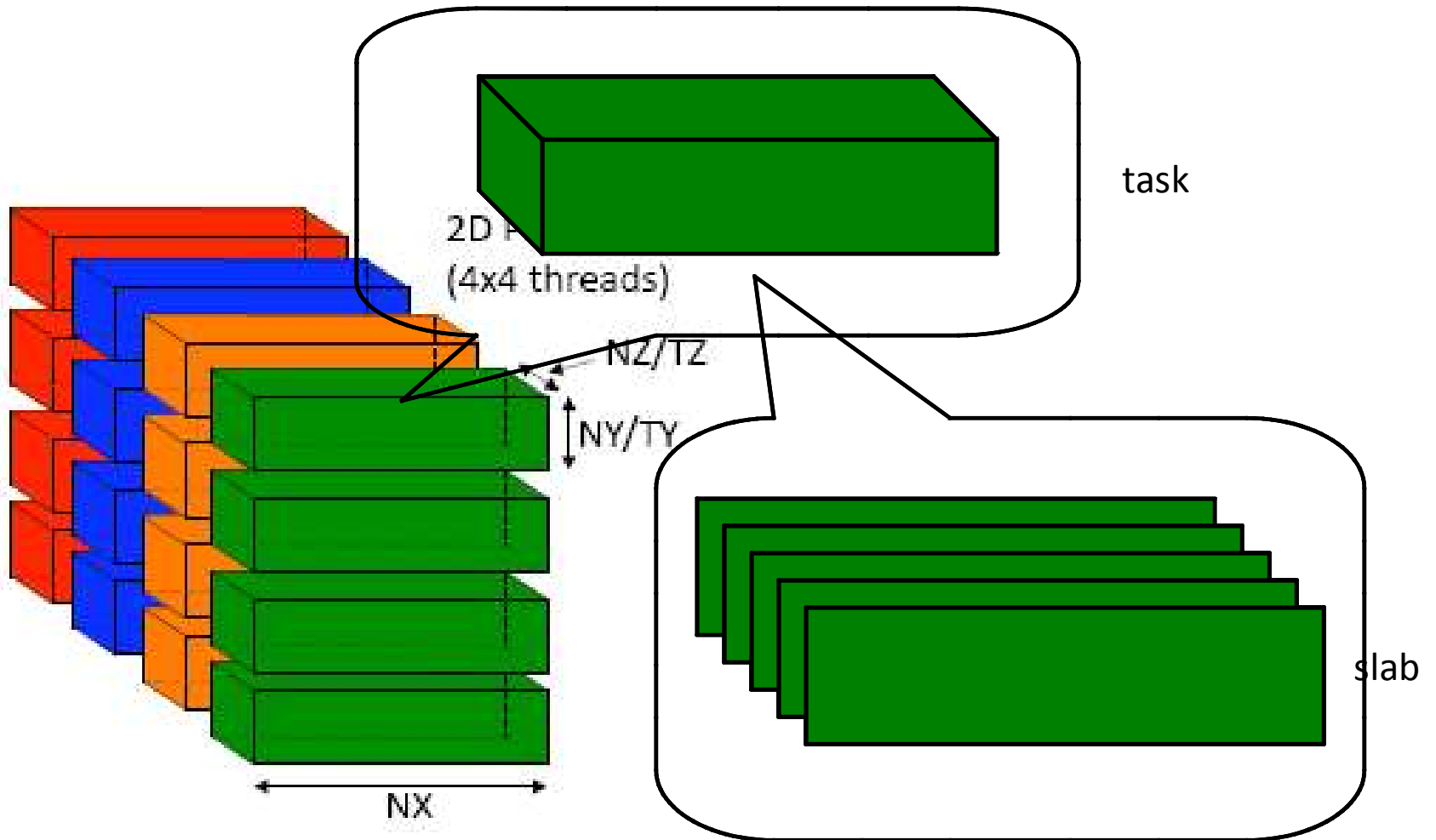


Figure 3: Comparison of 1D and 2D decompositions

Finer view

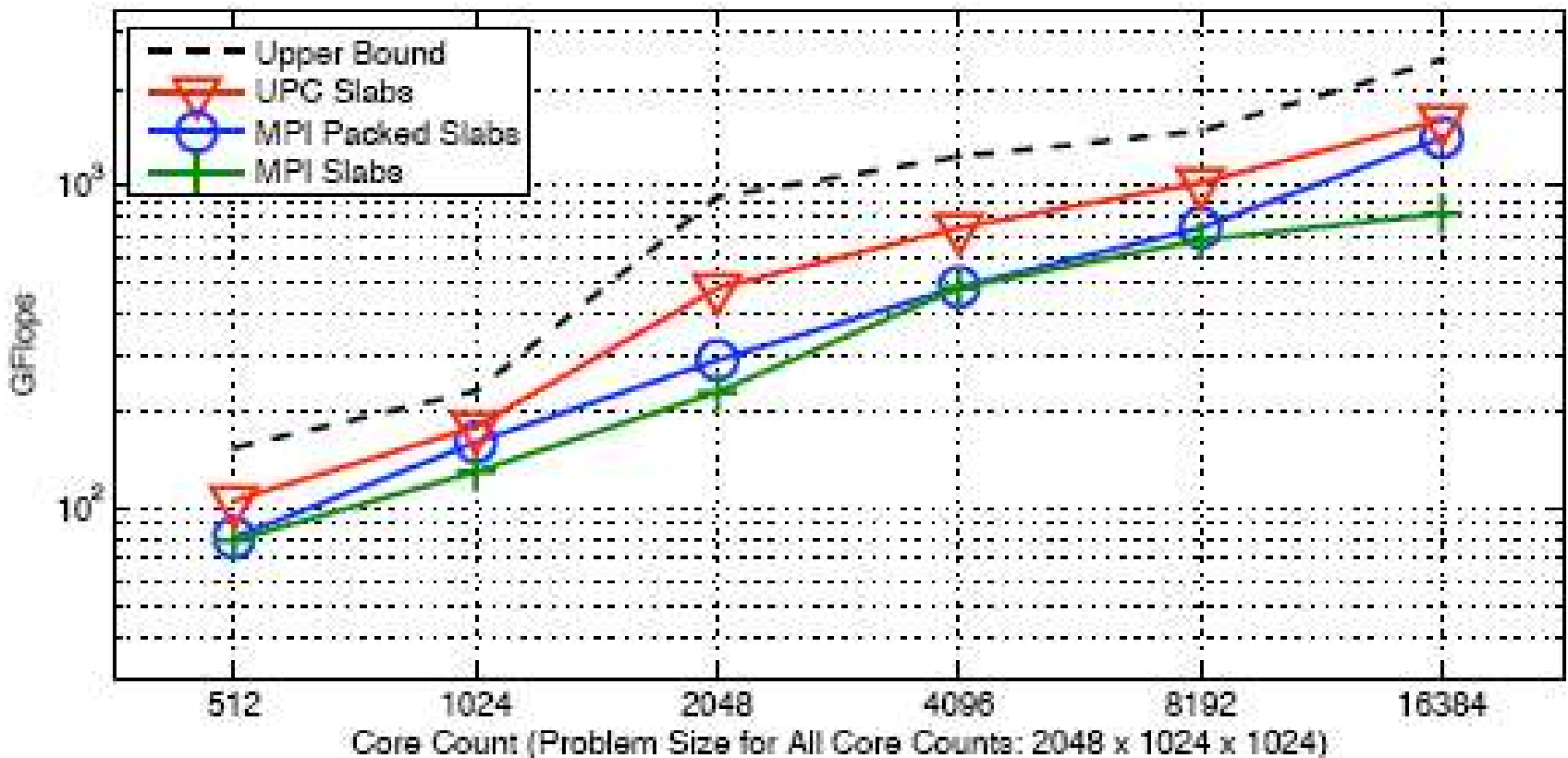


Slab and Packed Slab

- Current MPI implementation of FT in NAS benchmark used *packed slab*
 - Communication for small or median size message is expensive
 - Communicate a chunk of data
 - Poor communication/computation overlap
- The precondition does not hold for UPC program
 - Communicate at fine grain, *slab by slab*

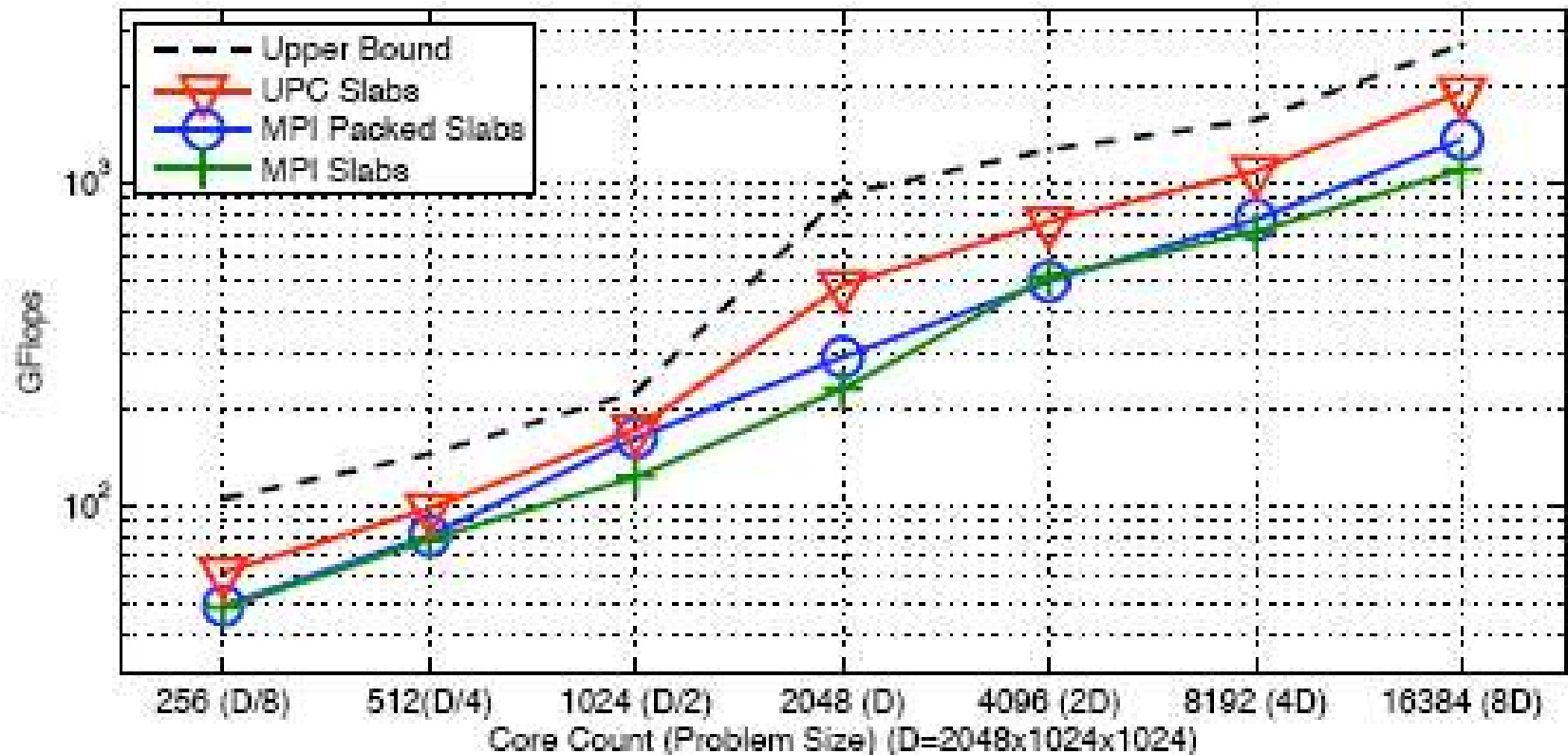
NAS FT Performance: Strong Scaling

- Fixed total problem size



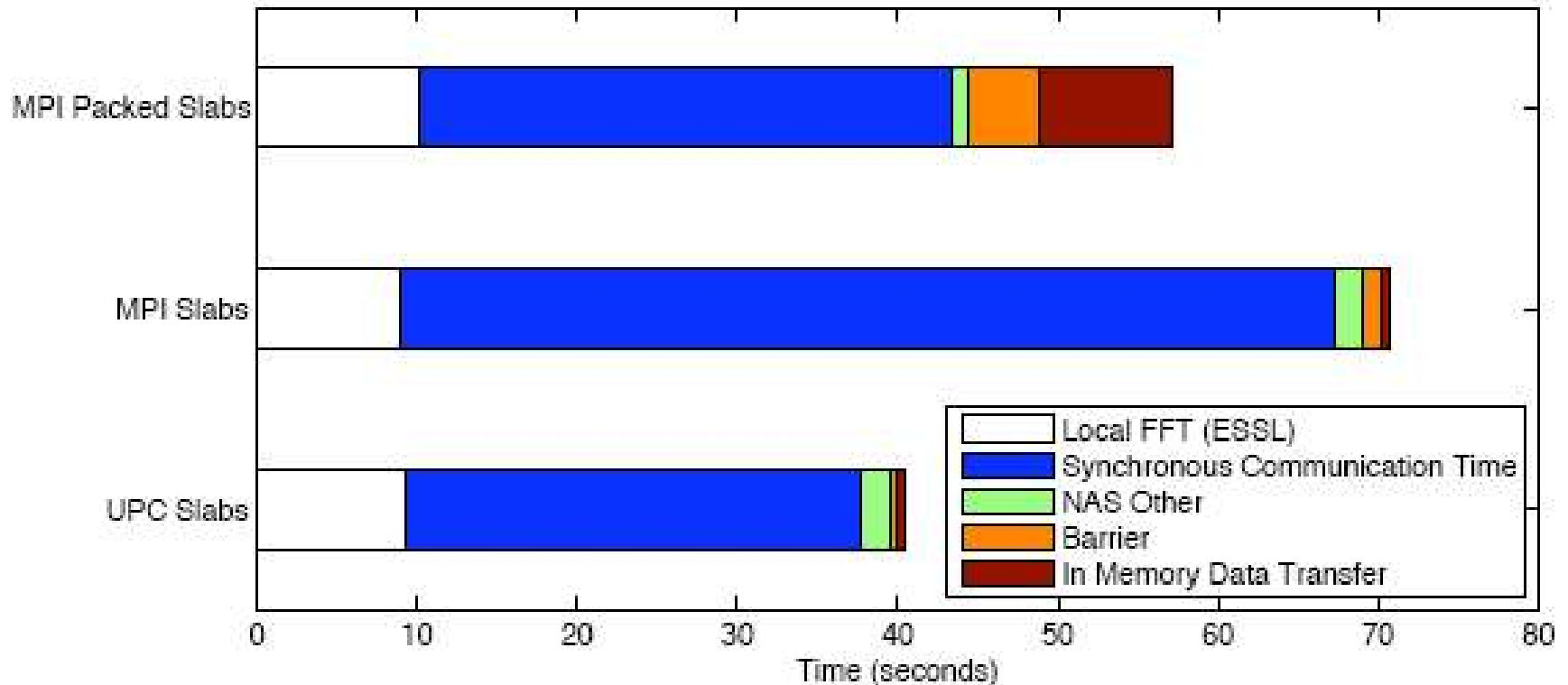
NAS FT Performance: Weak Scaling

- Fixed problem size per core



NAS FT Performance Breakdown

- Communication matters



Summary

- PGAS model
- UPC
 - Cyclic data distribution and `upc_forall`
 - Synchronizations: Barrier, Lock
 - Strict and Relaxed Memory Consistency
- Performance: FT example on BlueGene/P
 - RDMA, active message paradigm, one-sided put and get
 - New way to exploit overlapping

Comparison

	OpenMP	UPC	MPI
Execution Model	Thread	Thread	Process
Memory Model	Shared	PGAS	Distributed
Notation	Annotation	Language	Library
Locality Exploitation	No	Yes	Yes
Programmability	Good	Good	Poor (arguable)
Scalability	Poor	Good	Good
Communication	N/A	Active message, One-sided	Two-sided (limited one-sided)

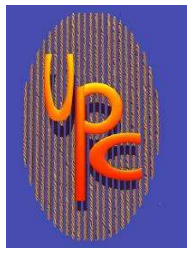
Compiler's View

- HPF (High Performance Fortran) requires the compiler to parallelize the code.
- MPI lets the programmer to write parallel code, e.g. partition data, and communicate data.
- UPC is somewhere between HPF and MPI.

References

- *High Performance Parallel Programming with Unified Parallel C*, SC 2005 tutorial.
- *Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap*, IPDPS 2009.
- *Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap*, IDPDS 2006.
- *A Performance Analysis of the Berkeley UPC Compiler*, ICS 2003.

Auxiliary Slides

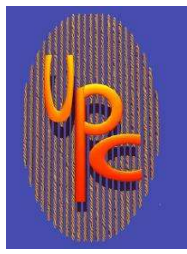


UPC Pointers

Where does it point to?

		Private	Shared
Private		PP	PS
Shared		SP	SS

Where does it reside?



UPC Pointers

- How to declare them?

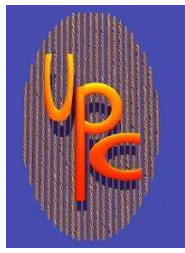
```
C int *p1;          /* private pointer pointing locally */
```

```
C shared int *p2; /* private pointer pointing into  
the shared space */
```

```
C int *shared p3; /* shared pointer pointing locally */
```

```
C 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

- In UPC pointers to shared objects have three fields:

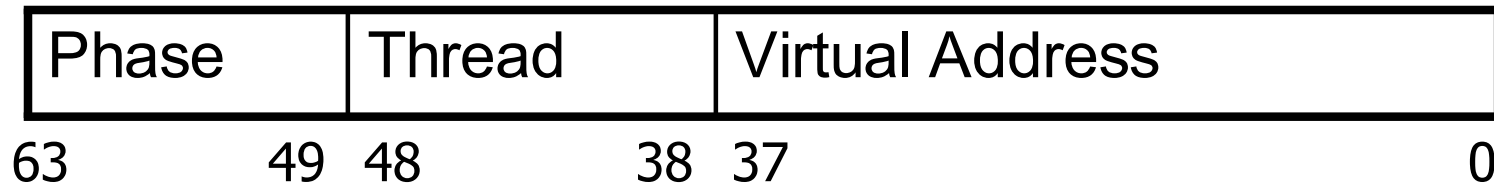
C thread number

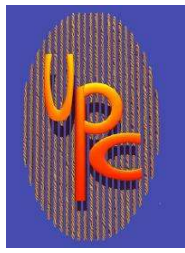
C local address of block

C phase (specifies position in the block)



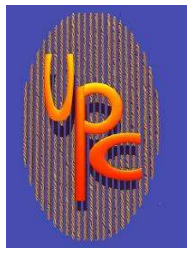
- Example: Cray T3E implementation





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



UPC Pointers

pointer to shared Arithmetic Examples:

Assume THREADS = 4

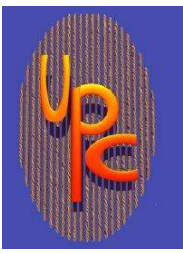
```
#define N 16
```

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

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

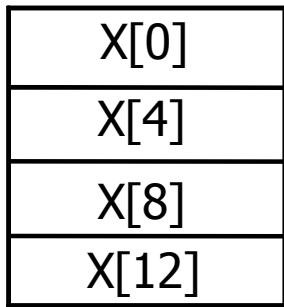
```
dp1 = dp + 9;
```



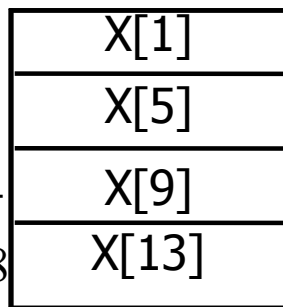


UPC Pointers

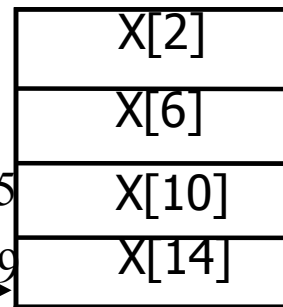
Thread 0



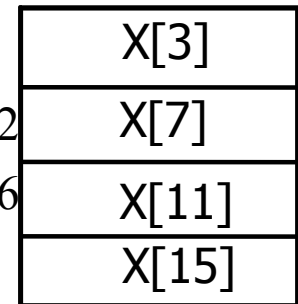
Thread 0



Thread 2



Thread 3



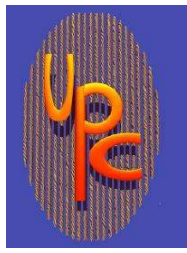
dp + 3
dp + 7

dp
dp + 4
dp + 8

dp+1
dp + 5
dp + 9

dp+2
dp+6

dp1

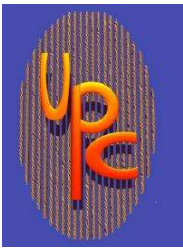


UPC Pointers

Assume THREADS = 4

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





UPC Pointers

Thread 0

X[0]
X[1]
X[2]

Thread 1

X[3]
X[4]
X[5]

dp

Thread 2

dp + 1	X[6]
dp + 2	X[7]
dp + 3	X[8]

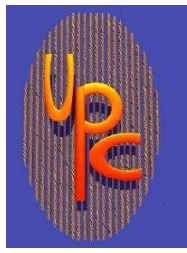
Thread 3

dp + 4	X[9]
dp + 5	X[10]
dp + 6	X[11]

X[12]
X[13]
X[14]

dp + 7	X[15]
--------	-------

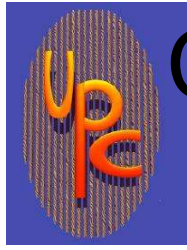
dp + 8
← dp + 9
└ dp1



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

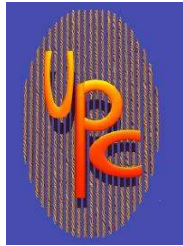
```
shared void *upc_all_alloc  
    (size_t nblocks, size_t nbytes);
```

nblocks: number of blocks

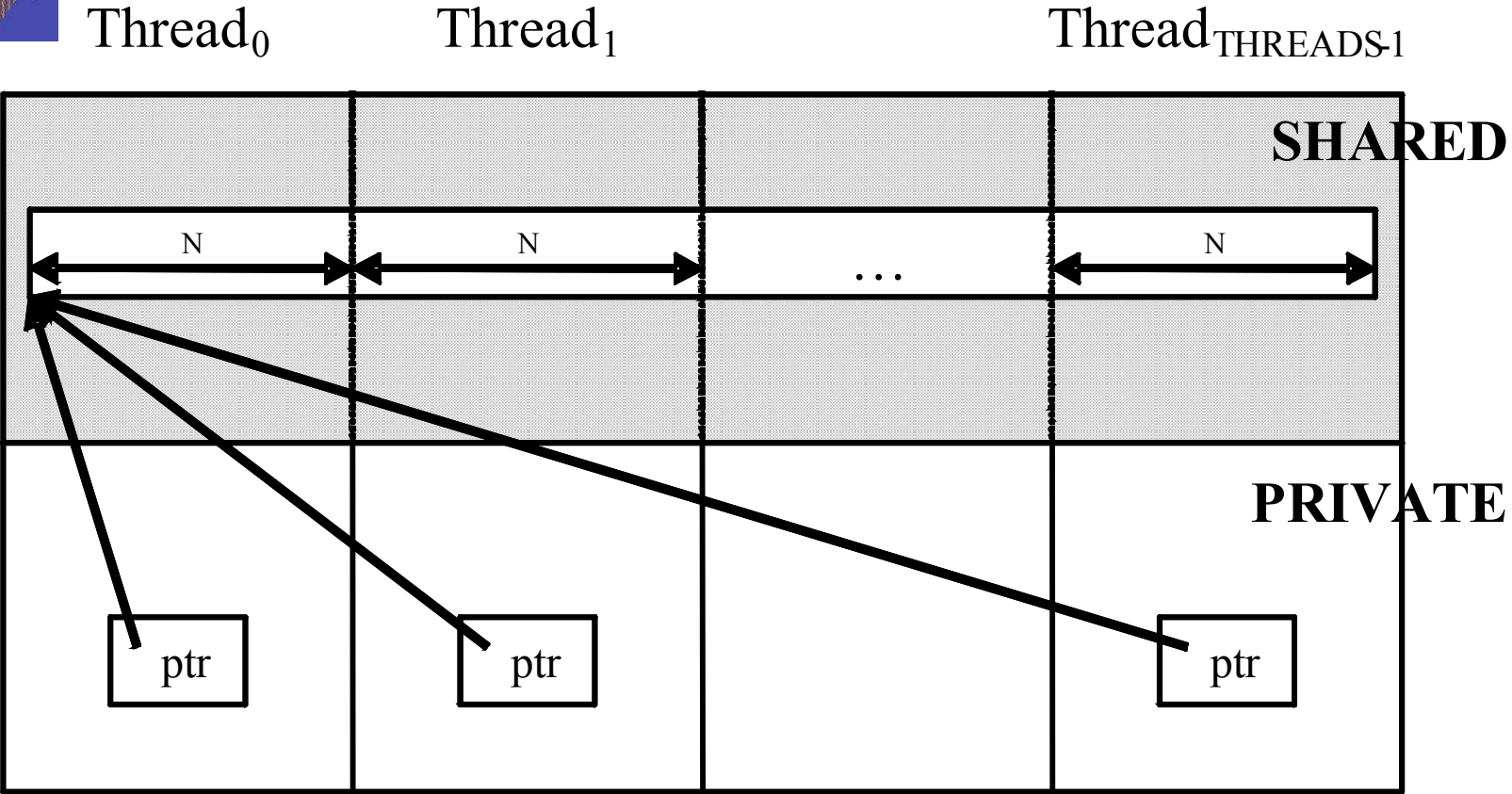
nbytes: block size

- 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]



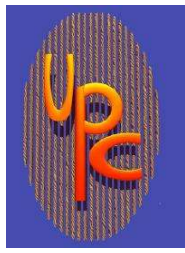


Collective Global Memory Allocation



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





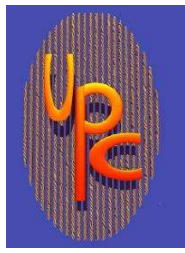
Global Memory Allocation

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

nblocks : number of blocks

nbytes : block size

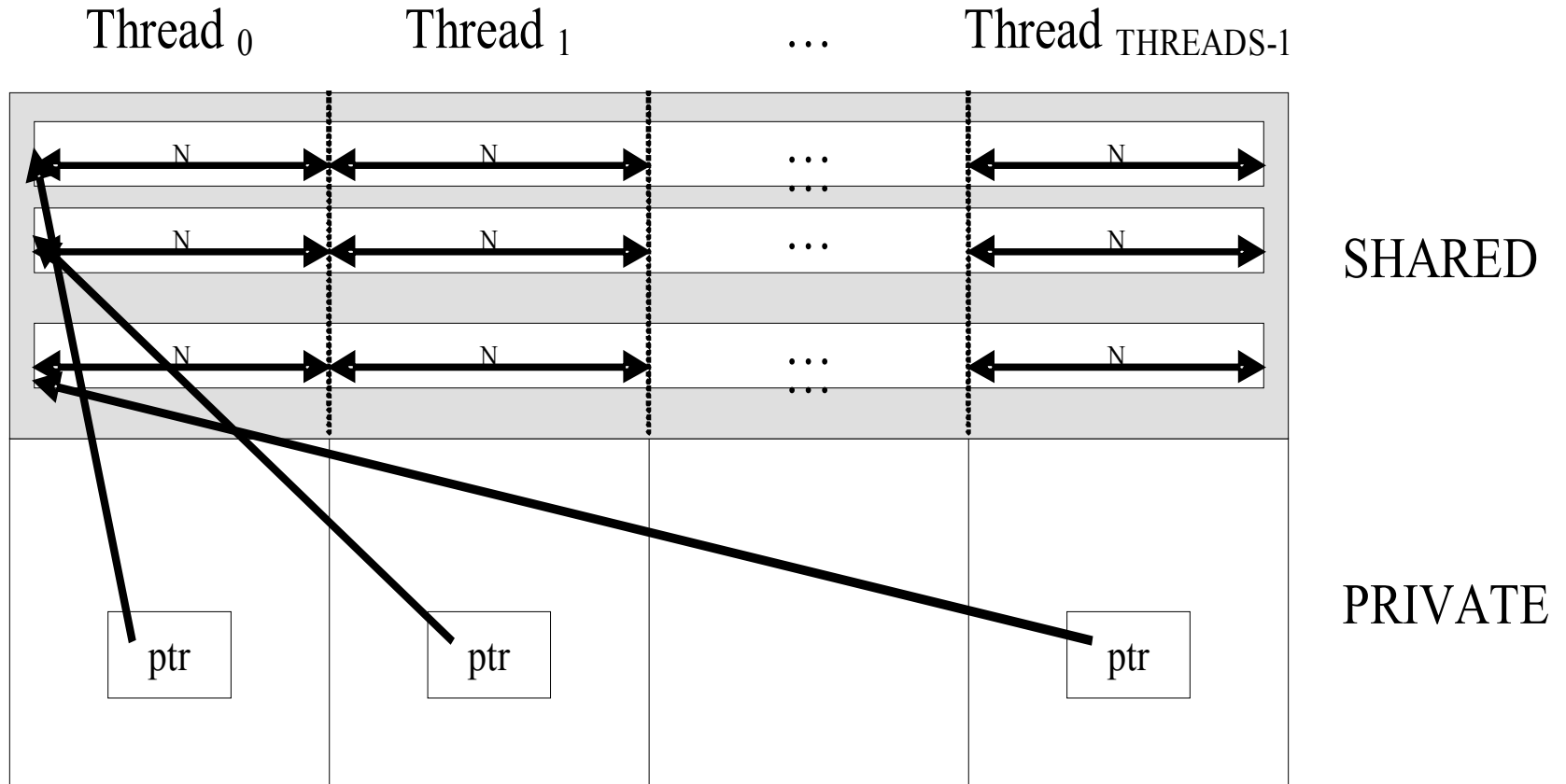
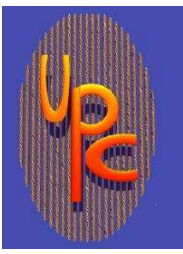
- Non collective, expected to be called by one thread
- The calling thread allocates a contiguous memory region in the shared space
- Space allocated per calling thread is equivalent to :
shared [nbytes] char[nblocks * nbytes]
- If called by more than one thread, multiple regions are allocated and each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

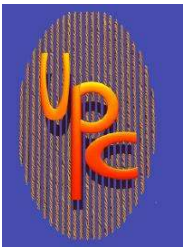


Global Memory Allocation

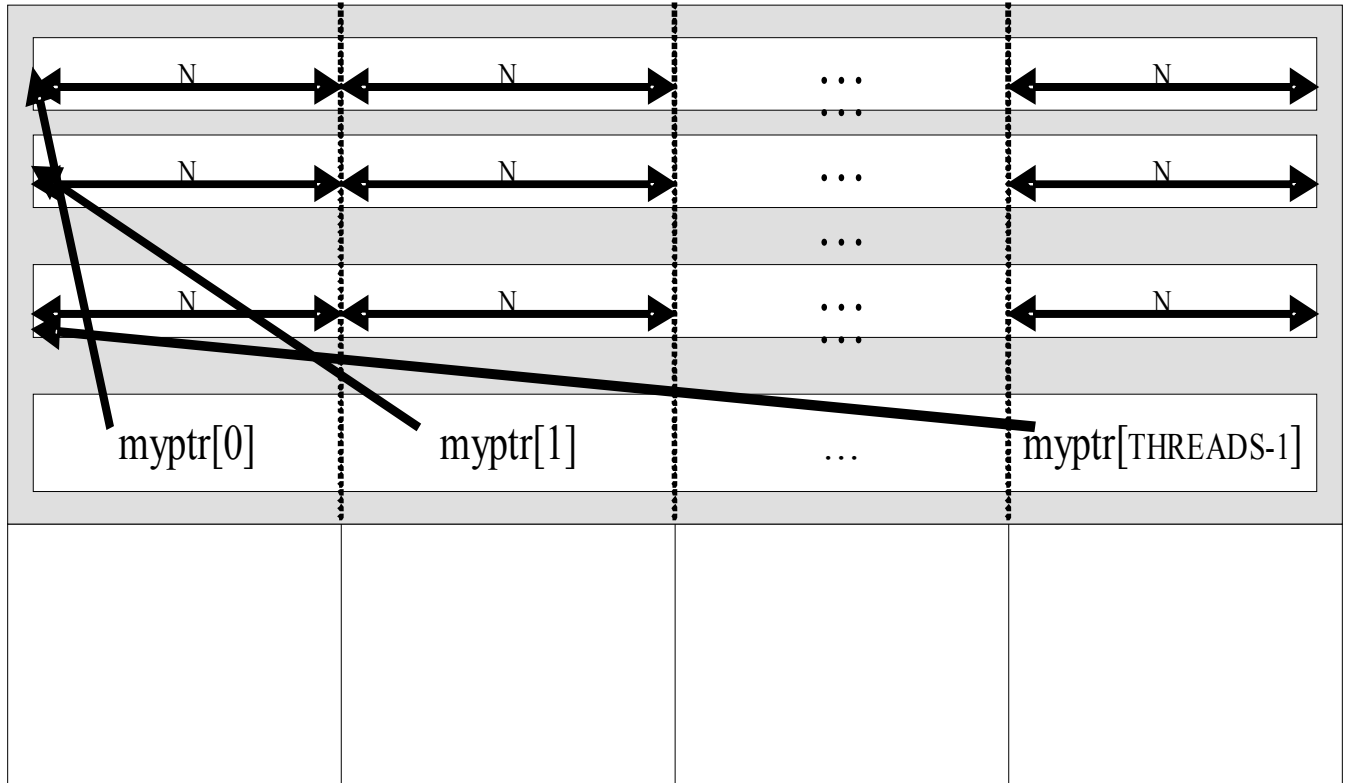
```
shared [N] int *ptr;  
  
ptr =  
    (shared [N] int *)  
    upc_global_alloc( THREADS, N*sizeof( int ) );
```

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



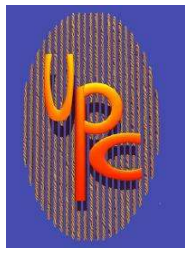


Thread₀ Thread₁ ... Thread_{THREADS-1}



SHARED

PRIVATE



Local-Shared Memory Allocation

shared void *upc_alloc (size_t nbytes);

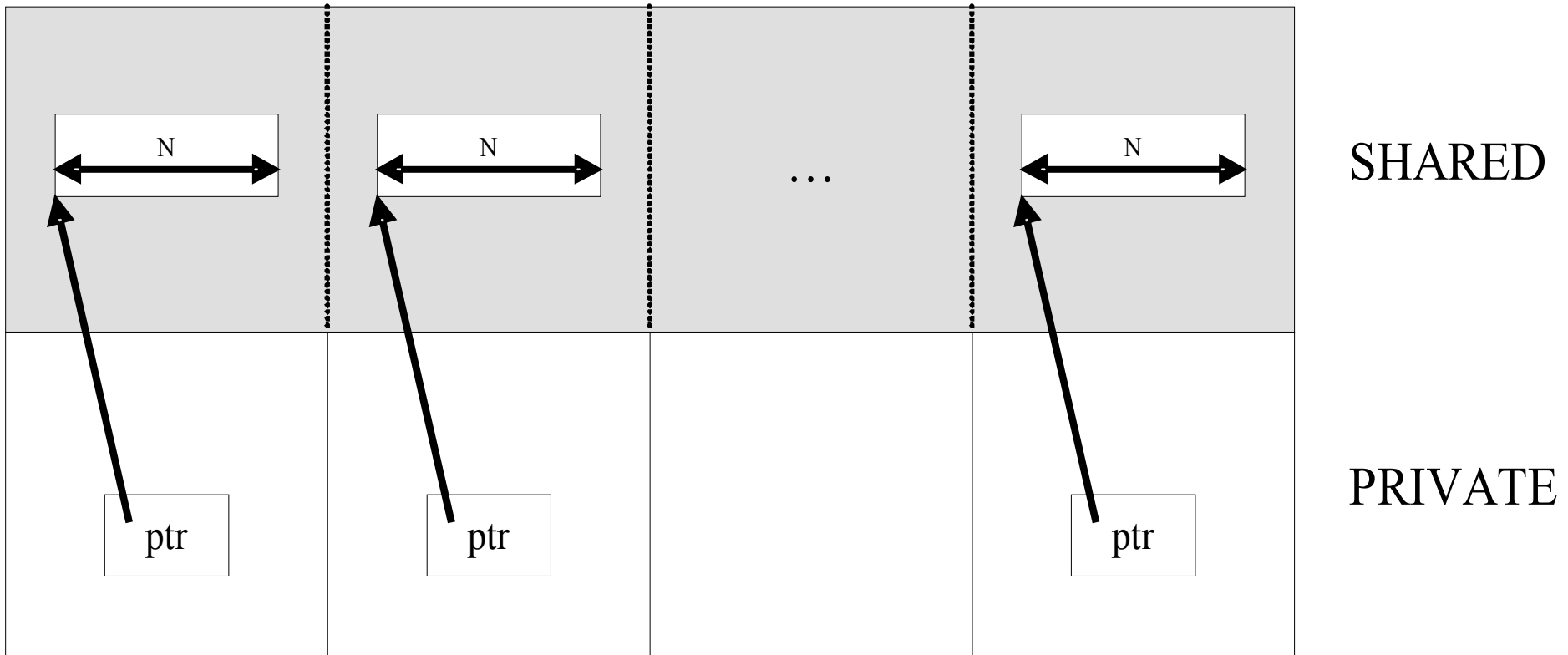
nbytes: block size

- 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

Thread ₀ Thread ₁ ... Thread _{THREADS-1}



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



Memory Space Clean-up

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

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