

---

# COMP 322: Fundamentals of Parallel Programming

## Lecture 36: Partitioned Global Address Space (PGAS) languages

**John Mellor-Crummey**  
Department of Computer Science, Rice University  
[johnmc@rice.edu](mailto:johnmc@rice.edu)

<https://wiki.rice.edu/confluence/display/PARPROG/COMP322>



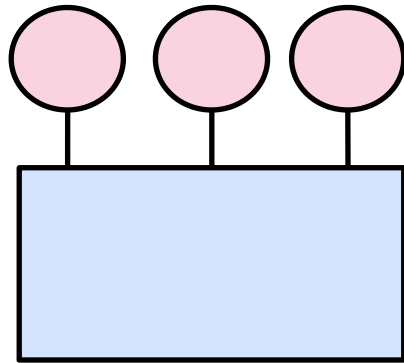
# Acknowledgments

---

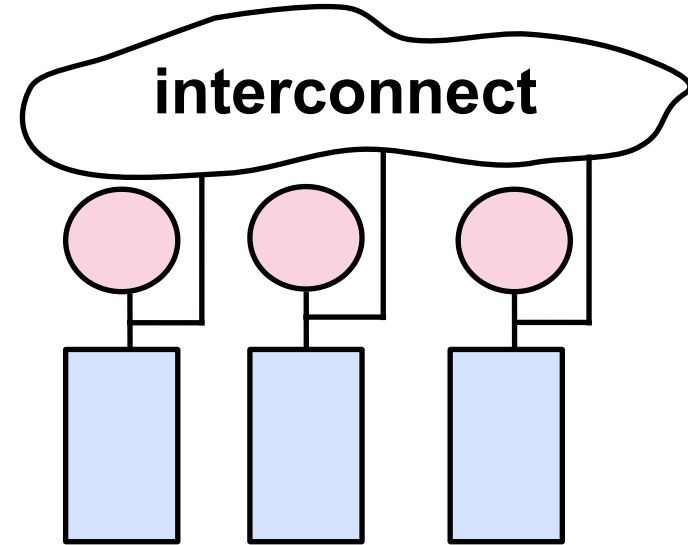
- **“Principles of Parallel Programming,” Calvin Lin & Lawrence Snyder**
  - **Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>**
- **Tarek El-Ghazawi, Steven Seidel. High Performance Parallel Programming with Unified Parallel C. SC05 Tutorial. <http://upc.gwu.edu/tutorials/UPC-SC05.pdf>**

# Parallel Architectures

---



**Shared Memory**



**Distributed Memory**

## Programming Models

**Habanero-Java**  
**Java Threads**  
**Cilk**  
**OpenMP**  
**Pthreads**



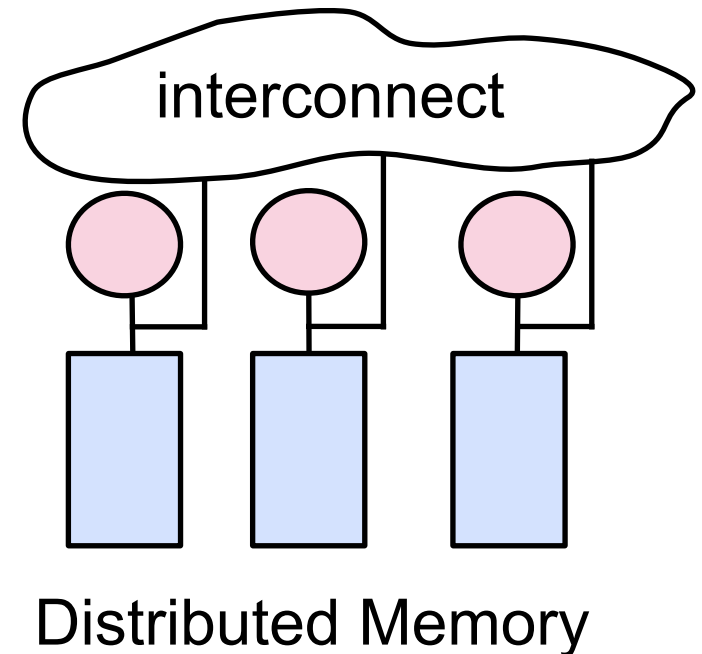
**MPI**  
**Map-Reduce**  
**UPC**  
**CAF**

# Performance Concerns for Distributed Memory

**Data movement and synchronization are expensive**

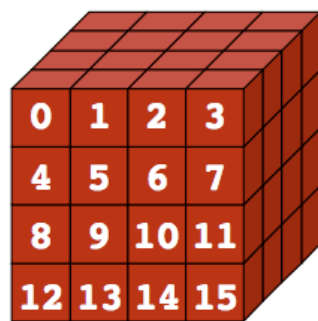
**To minimize overheads**

- **Co-locate data with processes**
  - **Aggregate multiple accesses to remote data**
  - **Overlap communication with computation**
- ⇒ **Significant programmability challenges with addressing these overheads in a shared-nothing programming model like MPI**

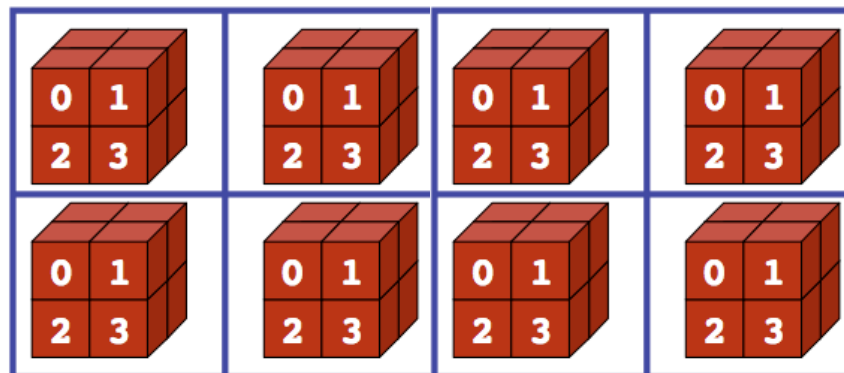


# Partitioned Global Address Space Languages

- **Global address space**
  - one-sided communication (GET/PUT) simpler than msg passing
- **Programmer has control over performance-critical factors**
  - data distribution and locality control lacking in thread-based models
  - computation partitioning
  - communication placement HJ places help with locality control but not data distribution
- **Data movement and synchronization as language primitives**
  - amenable to compiler-based communication optimization
- **Global view rather than local view**



Global View



Local View (8 processes)

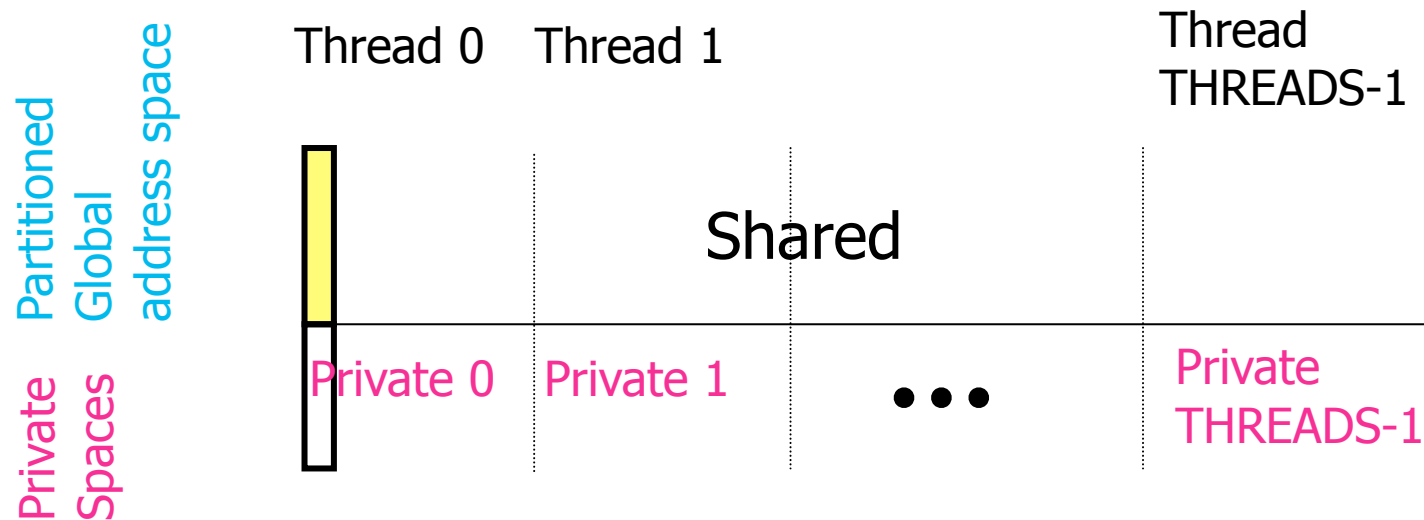
# Partitioned Global Address Space Languages

---

- **Unified Parallel C** (extension of C)
- **Coarray Fortran** (extension of Fortran)
- **Titanium** (extension of early version of Java)
  
- **Related efforts: newer languages developed since 2003 as part of the DARPA High Productivity Computing Systems (HPCS) program**
  - IBM: X10 (foundation for Habanero-Java)**
  - Cray: Chapel**
  - Oracle/Sun: Fortress**

# Data Distributions

- In HJ, distributions are used to guide computation mappings for affinity
- The idea of distributions was originally motivated by mapping data (array elements) to processors
- e.g., Unified Parallel C (UPC) language for distributed-memory parallel machines (Thread = Place)



Like shared vs. private/local data in HJ, except now each datum also has an “affinity” with a specific thread/place

# Unified Parallel C (UPC)

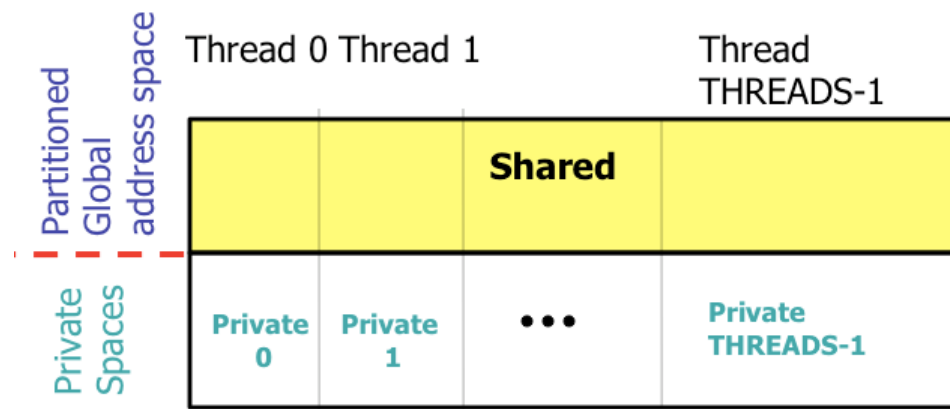
---

- **An explicit parallel extension of ISO C**
  - a few extra keywords
    - **shared, MYTHREAD, THREADS, upc\_forall**
- **Language features**
  - partitioned global address space for shared data
    - part of shared data co-located with each thread
  - threads created at application launch
    - each bound to a CPU
    - each has some private data
  - a memory model
    - defines semantics of interleaved accesses to shared data
  - synchronization primitives
    - barriers
    - locks
    - load/store



# UPC Execution Model

- Multiple threads working independently in a SPMD fashion
  - MYTHREAD specifies thread index (0..THREADS-1)
    - Like MPI processes and ranks
  - # threads specified at compile-time or program launch
- Partitioned Global Address Space (different from MPI)



- Threads synchronize as necessary using using
  - synchronization primitives
  - shared variables

# Shared and Private Data

---

- **Static and dynamic memory allocation of each type of data**
- **Shared objects placed in memory based on affinity**
  - shared scalars have affinity to thread 0**
    - **here, a scalar means a singleton instance of any type**
  - elements of shared arrays are allocated round robin among memory modules co-located with each thread**

# A One-dimensional Shared Array

---

Consider the following data layout directive

```
shared int y[2 * THREADS + 1];
```

For  $\text{THREADS} = 3$ , we get the following cyclic layout

Thread 0

y[0]
y[3]
y[6]

Thread 1

y[1]
y[4]

Thread 2

y[2]
y[5]

# A Multi-dimensional Shared Array

---

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

For **THREADS = 3**, we get the following cyclic layout

Thread 0

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

Thread 1

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

Thread 2

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

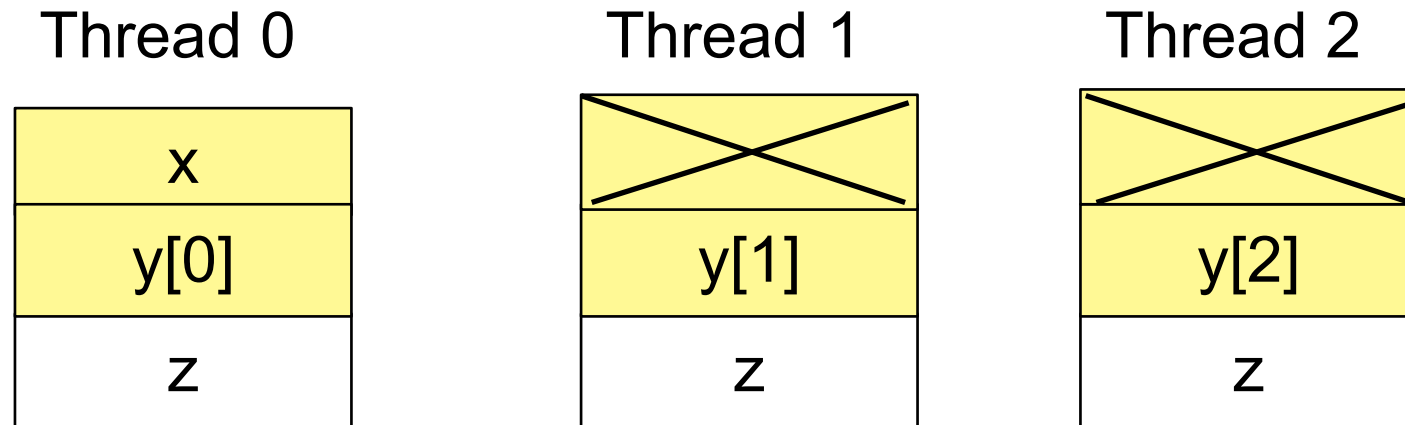
# Shared and Private Data

---

Consider the following data layout directives

```
shared int x; // x has affinity to thread 0
shared int y[THREADS];
int z; // private
```

For THREADS = 3, we get the following layout

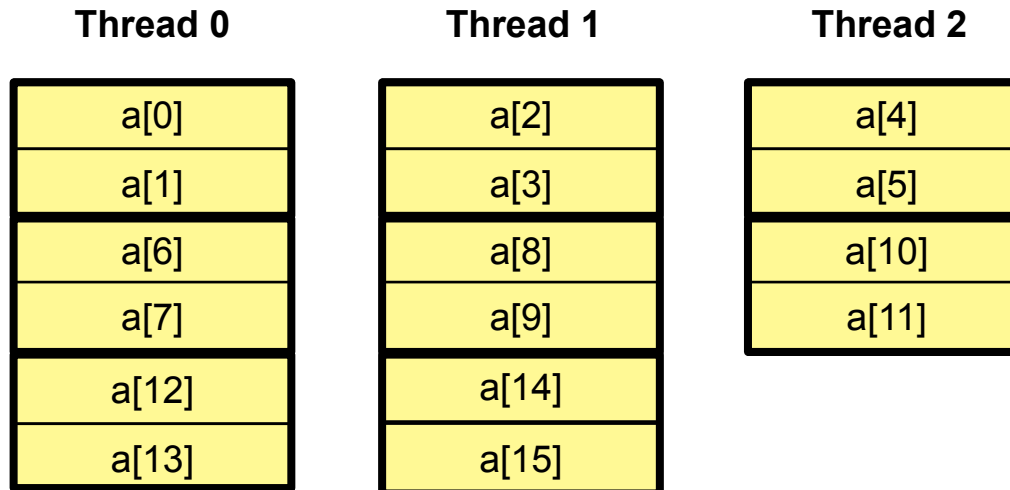


# Controlling the Layout of Shared Arrays

- Can specify a blocking factor for shared arrays to obtain block-cyclic distributions
  - default block size is 1 element  $\Rightarrow$  cyclic distribution
- Shared arrays are distributed on a block per thread basis, round robin allocation of block size chunks
- Example layout using block size specifications

—e.g., **shared** [2] **int** a[16]

block size



# Blocking of Shared Arrays

---

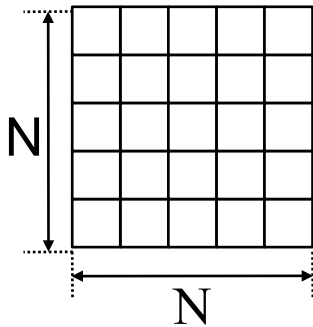
- Block size and THREADS determine *affinity*  
—with which thread will a datum be co-located
- Element  $i$  of a blocked array has affinity to thread:

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

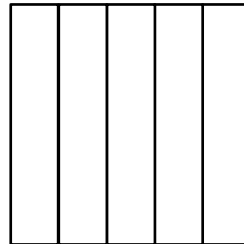
# Blocking Multi-dimensional Data I

---

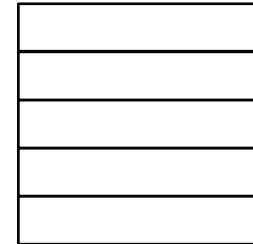
- Manage the interaction between
  - contiguous memory layout of C multi-dimensional arrays
  - blocking factor for shared layout
- Consider layouts for different block sizes for
  - shared **[BLOCKSIZE]** double grids[N][N];



Default  
BLOCKSIZE=1



Column Blocks  
BLOCKSIZE=N/THREADS



Distribution by Row  
BLOCKSIZE=N

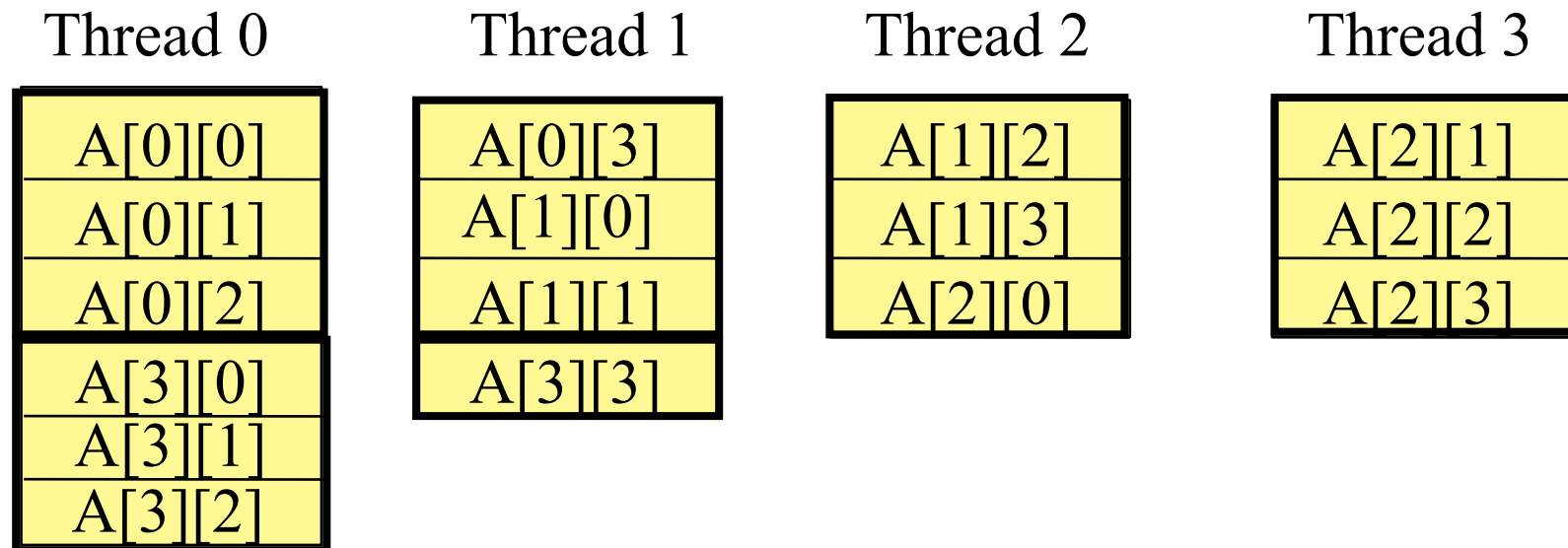


# Blocking Multi-dimensional Data II

- Consider the data declaration

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

- When THREADS = 4, this results in the following data layout



The mapping is not pretty for most blocking factors

# A Simple UPC Program: Vector Addition

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

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

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

Iteration #:

Thread 0 Thread 1

0 1  
2 3

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

...

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

...

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

...

Shared Space

Each thread executes each iteration to check if it has work

# A More Efficient 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 = MYTHREAD; i < N; i += THREADS)
        v1plusv2[i]=v1[i]+v2[i];
}
```

Iteration #:

Thread 0 Thread 1

0 1  
2 3

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

...

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

...

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

...

Shared Space

Each thread executes only its own iterations

# Worksharing with `upc_forall`

---

- Distributes independent iterations across threads
- Simple C-like syntax and semantics
  - `upc_forall`(`init`; `test`; `loop`; `affinity`)
- Affinity is used to enable locality control
  - usually, map iteration to thread where the iteration's data resides
- Affinity can be
  - an integer expression, or a
  - reference to (address of) a shared object

# Work Sharing + Affinity with `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])
    // Execute iteration i at a[i]'s thread/place
    a[i] = b[i] * c[i];
```

- **Example 2: implicit affinity with integer expressions**

```
shared int a[100], b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; i)
    // Execute iteration i at place i%THREADS
    a[i] = b[i] * c[i];
```

- **Both yield a round-robin distribution of iterations**

# Vector Addition Using `upc_forall`

thread affinity for work: have thread  $i$  execute iteration  $i$

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

Iteration #:

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

Shared Space

Each thread executes subset of global iteration space as directed by the affinity clause

# Work Sharing + Affinity with `upc_forall`

- **Example 3: implicit affinity 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**

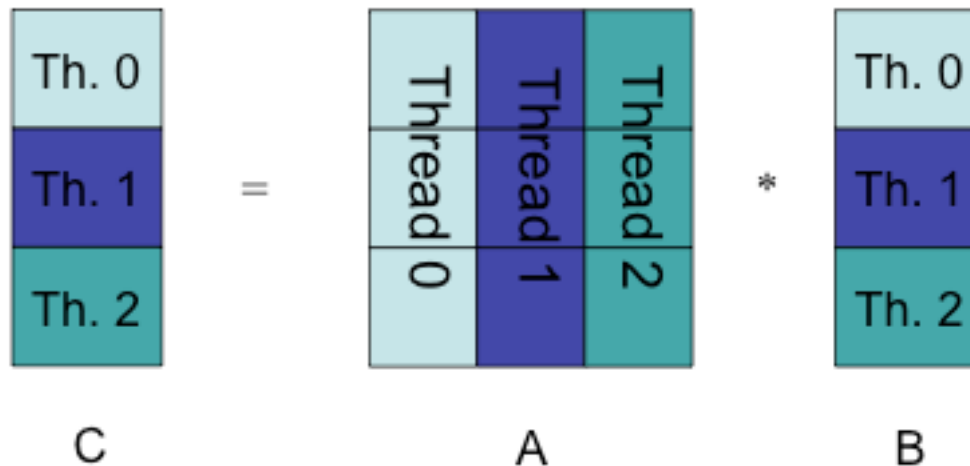
i	i*THREADS	i*THREADS/100
0..24	0..96	0
25..49	100..196	1
50..74	200..296	2
75..99	300..396	3

*Let's explore this further in worksheet 36!*

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

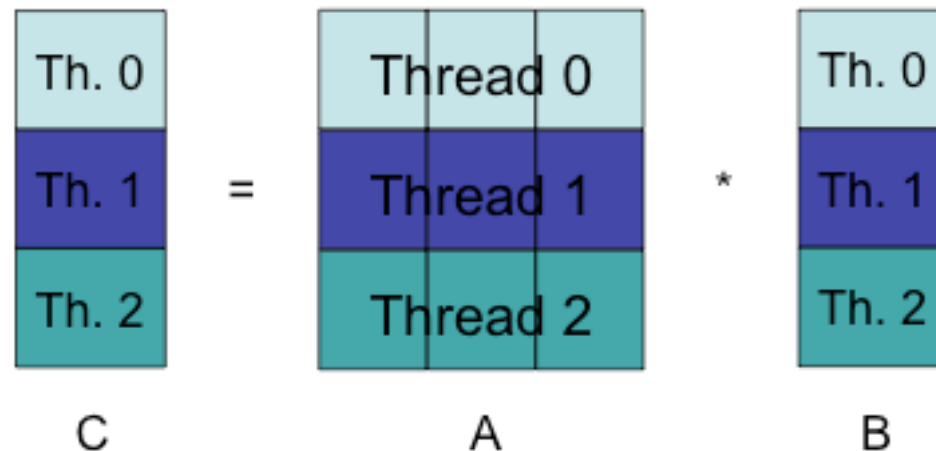




# Matrix-Vector Multiply (Better Distribution)

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

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



# Synchronization - Barriers

---

- **Barriers (blocking)**
  - `upc_barrier expr_opt;`
    - like “next” operation in HJ
- **Split-phase barriers (non-blocking)**
  - `upc_notify expr_opt;`
    - like explicit signal on an HJ phaser
  - `upc_wait expr_opt;`
    - note: `upc_notify` is not blocking `upc_wait` is
    - like explicit wait on an HJ phaser

# Synchronization - Locks

---

- **Lock primitives**

- void upc\_lock(upc\_lock\_t \*l)

- int upc\_lock\_attempt(upc\_lock\_t \*l) // success returns 1

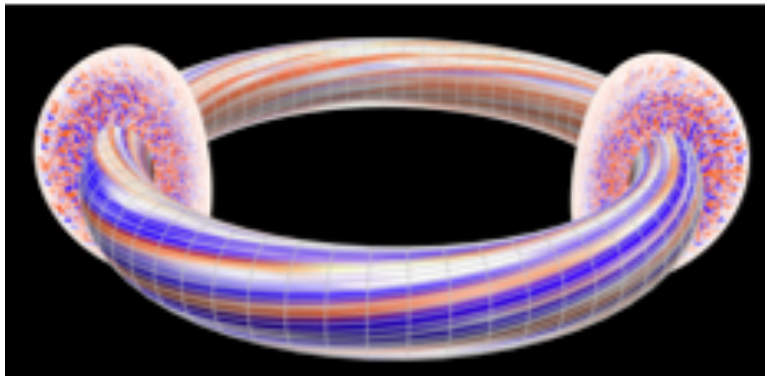
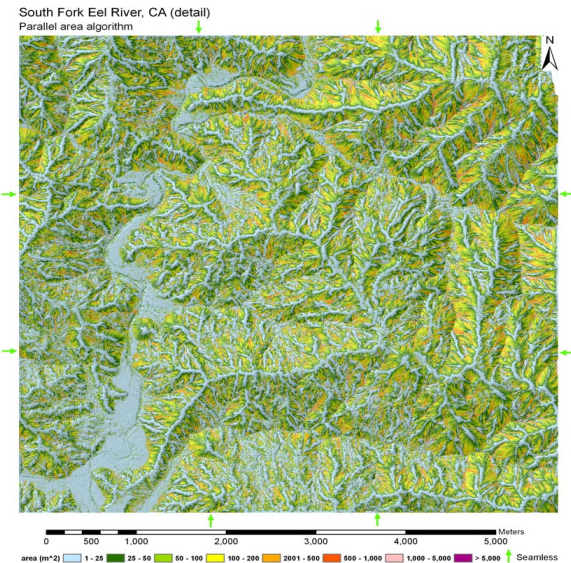
- void upc\_unlock(upc\_lock\_t \*l)

- **Locks are allocated dynamically, and can be freed**

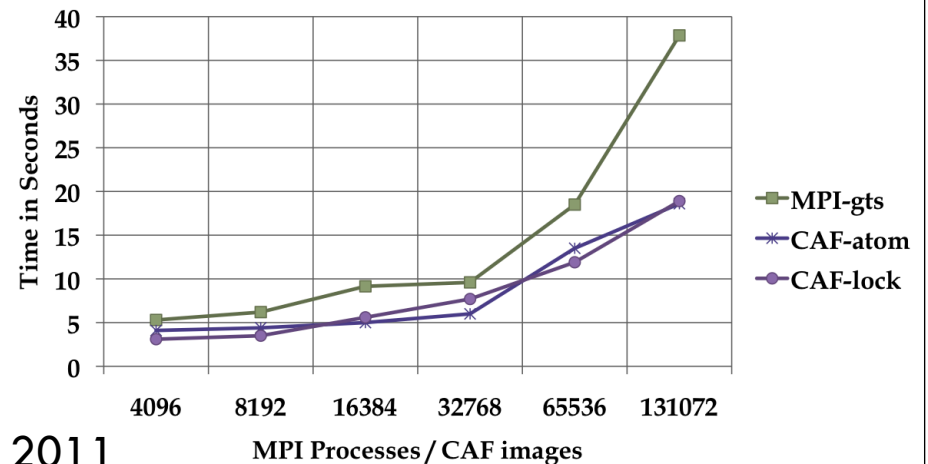
- **Locks are properly initialized after they are allocated**

# Application Work in PGAS

- Network simulator in UPC (Steve Hofmeyr, LBNL)
- Barnes-Hut in UPC (Marc Snir et al)
- Landscape analysis
  - “Contributing Area Estimation” in UPC (Brian Kazian, UCB)
- GTS Shifter in CAF
  - Preissl, Wichmann, Long, Shalf, Ethier, Koniges (LBNL, Cray, PPPL)



Slide credit: Kathy Yelick, January 2011



# Worksheet #36: UPC data distributions

---

Name 1: \_\_\_\_\_

Name 2: \_\_\_\_\_

In the following example from slide 23, assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, identify an iteration of the `upc_forall` construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote). Explain your answer in each case.

```
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];
```