COMP 322: Fundamentals of Parallel Programming

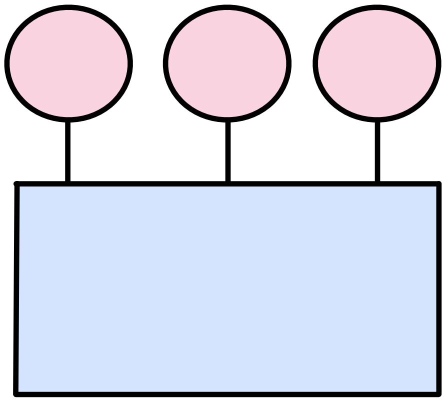
Lecture 35: Partitioned Global Address Space (PGAS) languages

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University
(Ack: many slides are courtesy of John Mellor-Crummey)

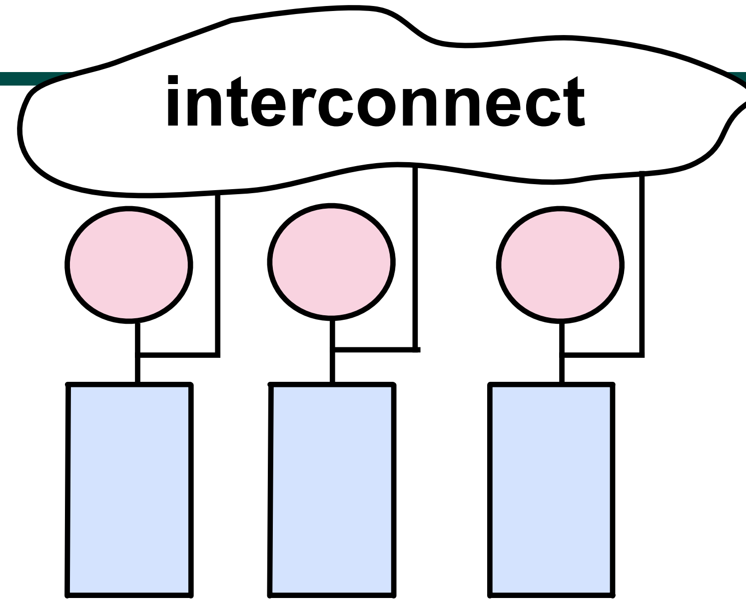
Contact email: vsarkar@rice.edu

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

Parallel Architectures



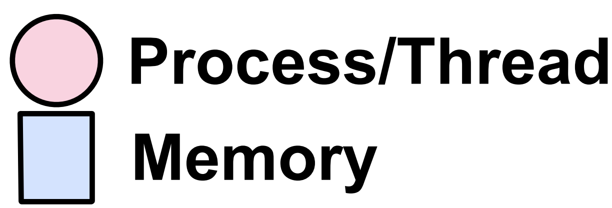
Shared Memory



Distributed Memory

Programming Models

Habanero-Java
Java Threads
Cilk
OpenMP
Pthreads



MPI
Map-Reduce
UPC
CAF

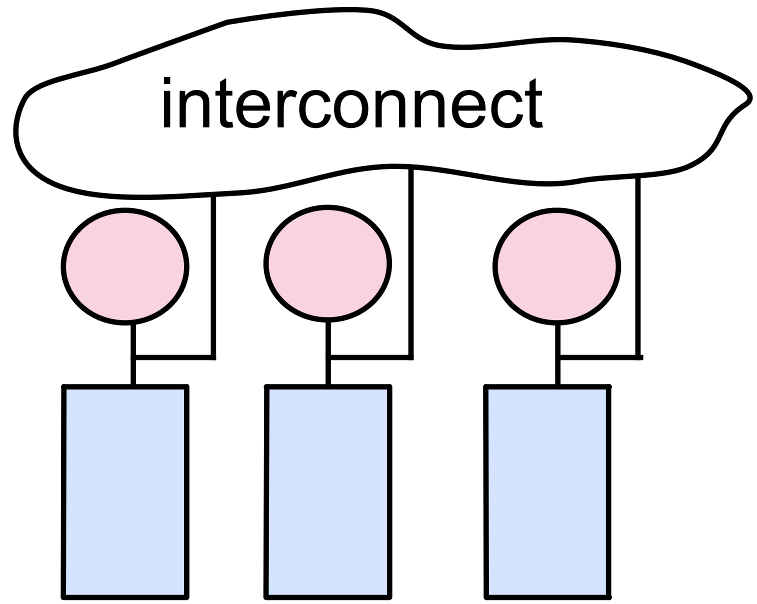
Performance and Programmability Concerns

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

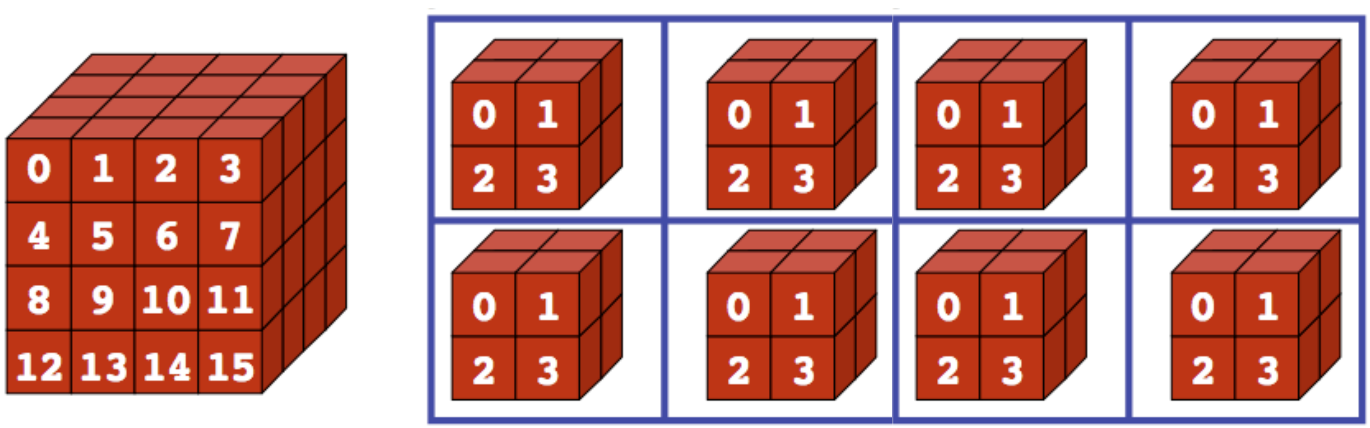


Distributed Memory

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)**

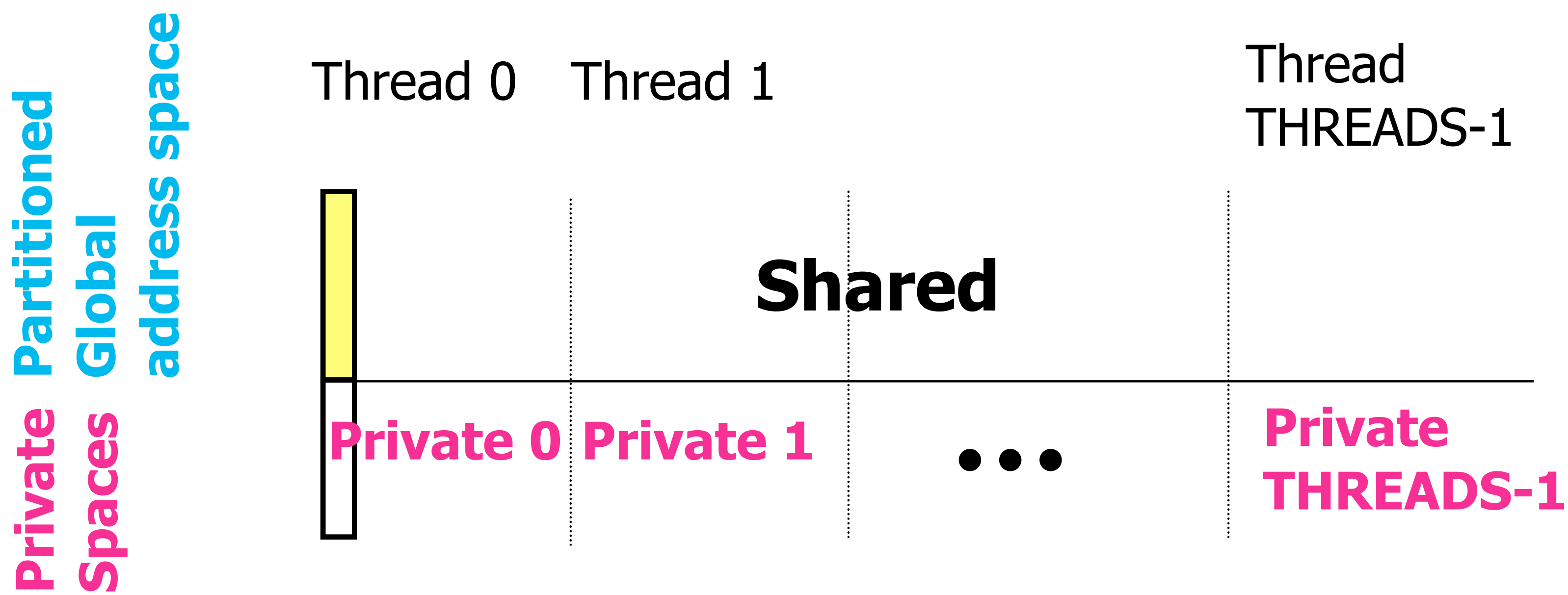
Partitioned Global Address Space (PGAS) Languages

- 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 (starting point for Habanero-Java)
  - Cray: Chapel
  - Oracle/Sun: Fortress

Data Distributions

- Motivation for distributions: partitioning and mapping arrays elements to processors
- In HJlib, distributions are used to map computations to places for affinity
- For Unified Parallel C (UPC), distributions map data onto distributed-memory parallel machines (Thread = Place)



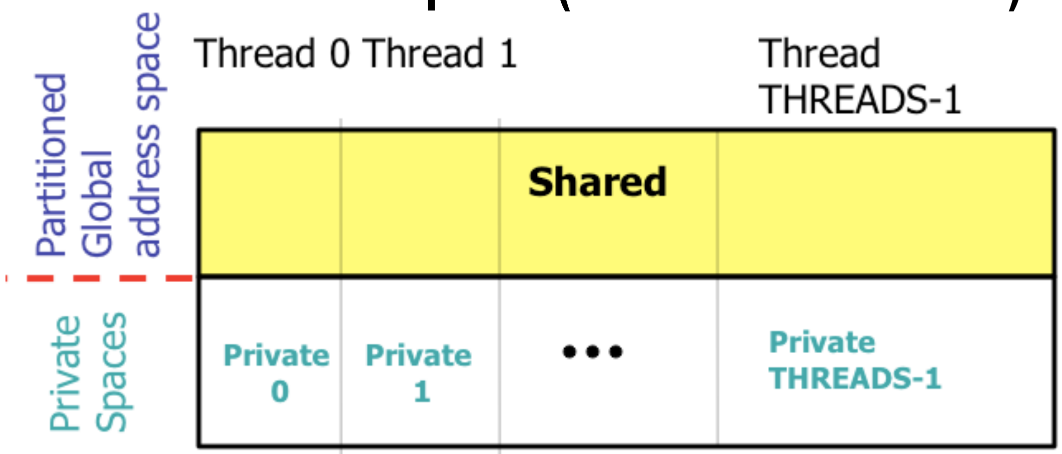
Like shared vs. private/local data in HJ, except now each datum also has an "affinity" with a specific thread/place

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
  - —synchronization primitives
  - —shared variables

Shared and Private Data

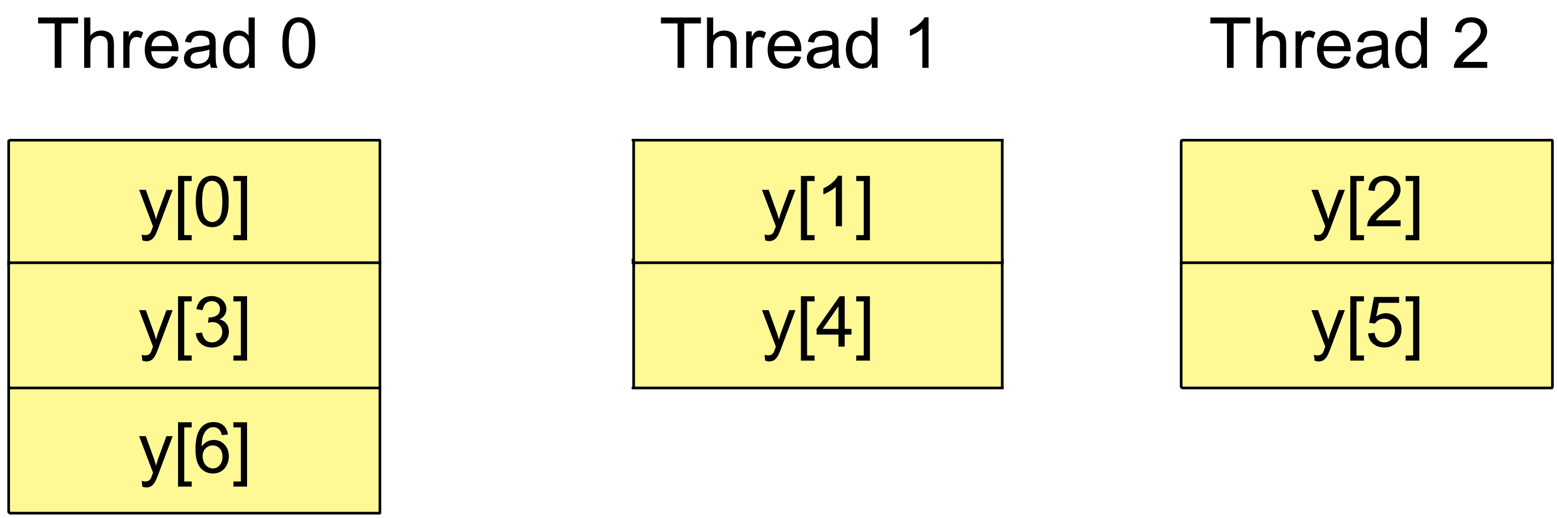
- Static and dynamic memory allocation of each type of data
- Shared objects placed in memory based on affinity
  - shared scalars have affinity to thread 0
    - here, a scalar means a singleton instance of any type
  - by default, elements of shared arrays are allocated “round robin” among memory modules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

A One-dimensional Shared Array

Consider the following data layout directive

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

For THREADS = 3, we get the following cyclic layout



A Multi-dimensional Shared Array

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

For THREADS = 3, we get the following cyclic layout

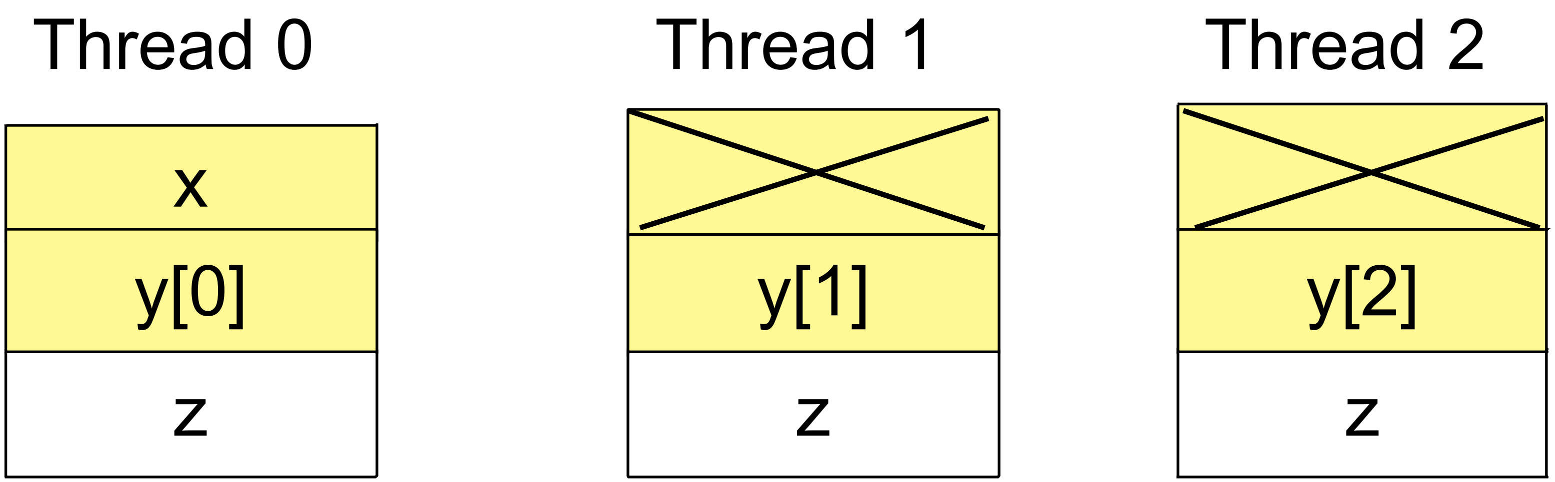
Thread 0	Thread 1	Thread 2
A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

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



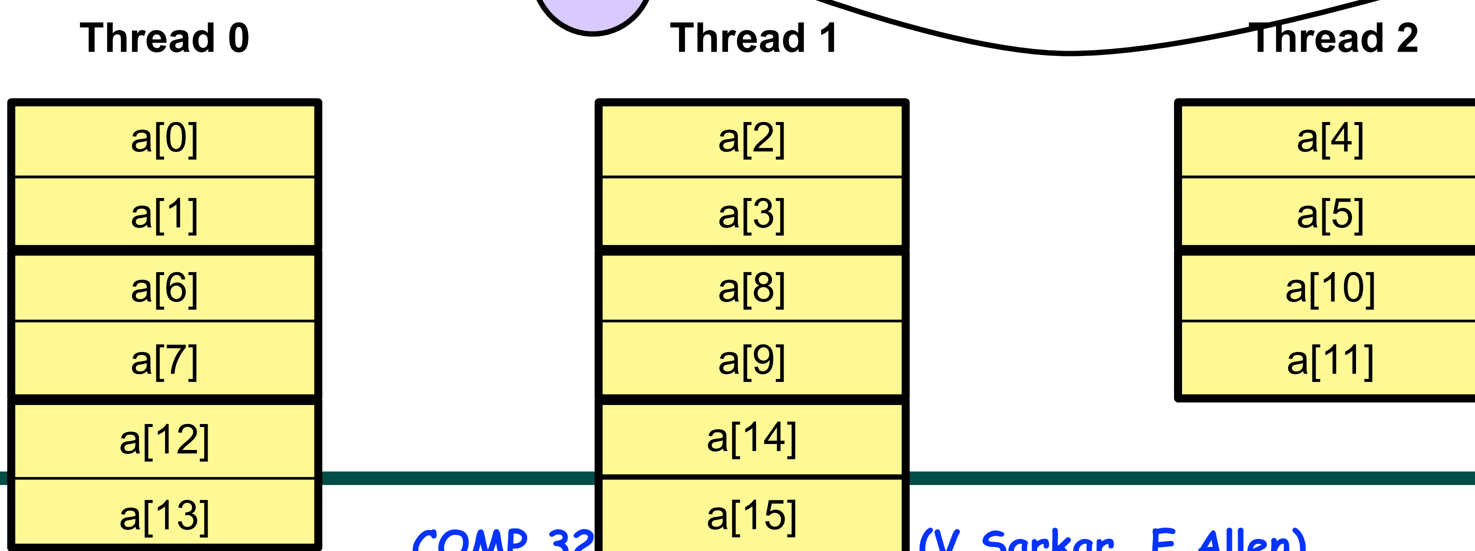
Controlling the Layout of Shared Arrays

- **Can specify a blocking factor for shared arrays to obtain block-cyclic distributions**
  - **default block size is 1 element ⇒ cyclic distribution**

- **Shared arrays are distributed on a block per thread basis, round robin allocation of block size chunks**

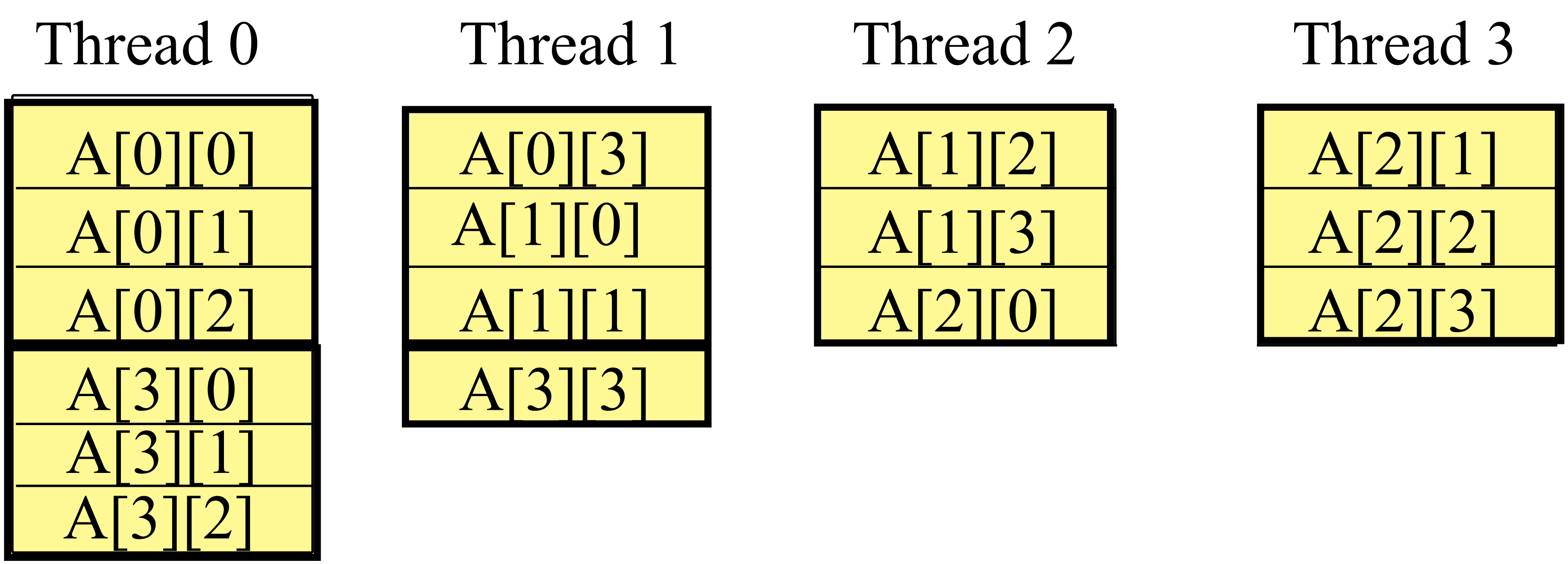
- **Example layout using block size specifications**
  - **e.g., `shared [2] int a[16]`**

block size



Blocking Multi-dimensional Data

- 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

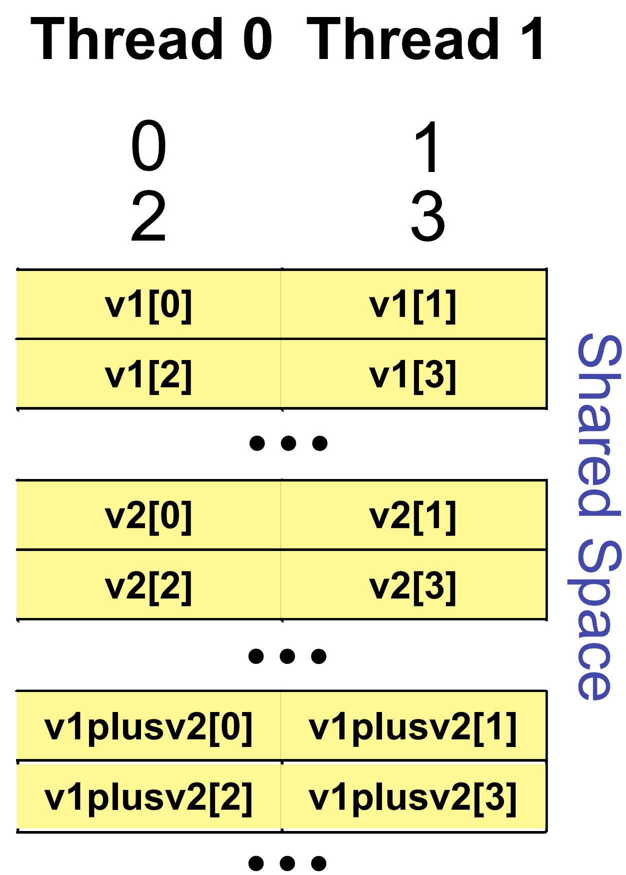
```
//vect_add.c
#include <upc_relaxed.h>
#define N 100*THREADS

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];
}
```

	Thread 0	Thread 1
Iteration #:	0 2	1 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]
	…	…

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];
}
```



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

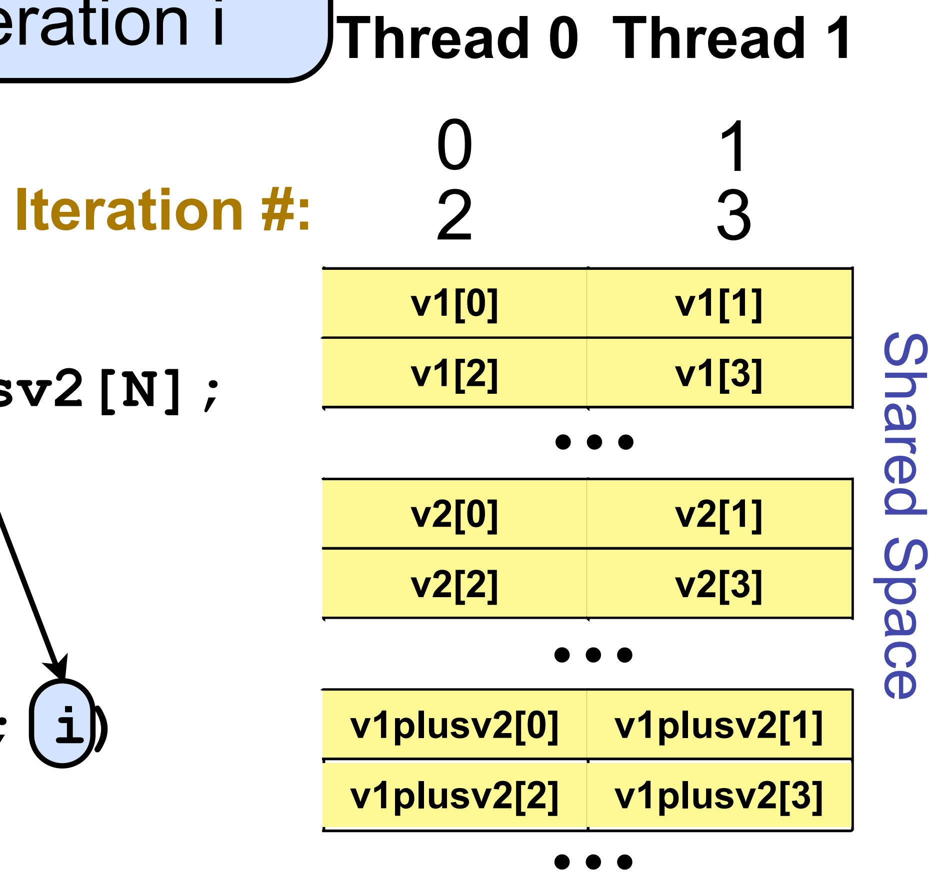
```
shared int a[100],b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; i)
  // Execute iteration i at place i%THREADS
  a[i] = b[i] * c[i];
```

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



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

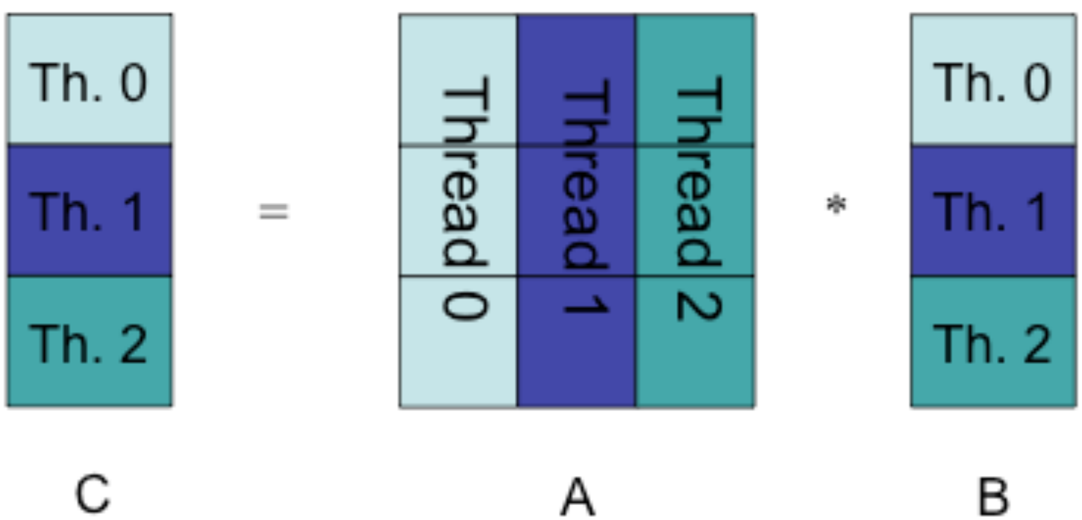
```
shared [25] int a[100],b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
    a[i] = b[i-1] * c[i+1];
```

- 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

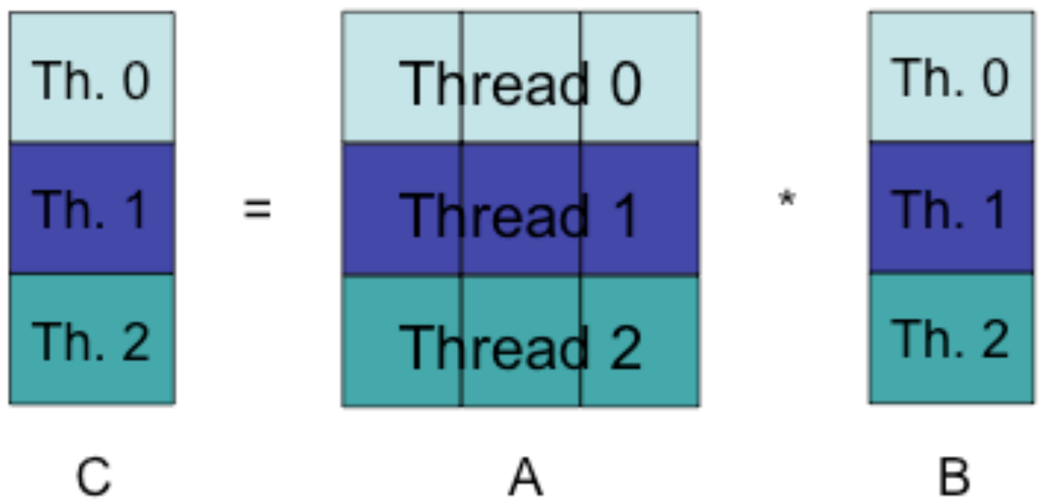
Matrix-Vector Multiply (Default Distribution)

```
// 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];
      }
}
```



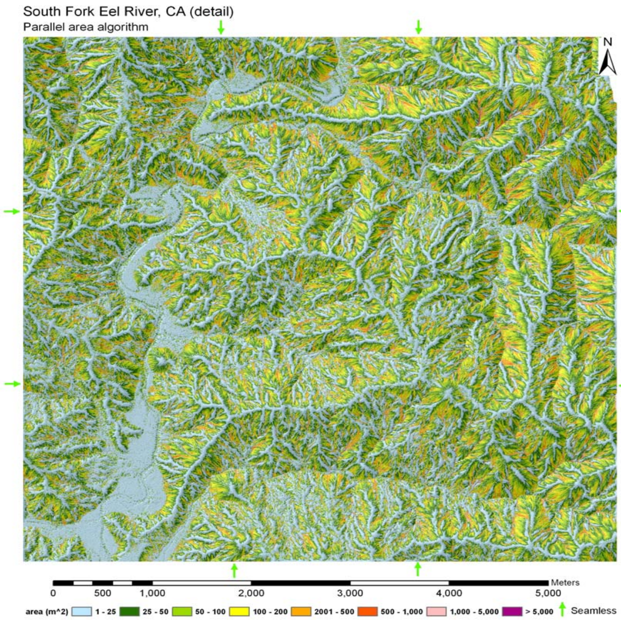
Synchronization

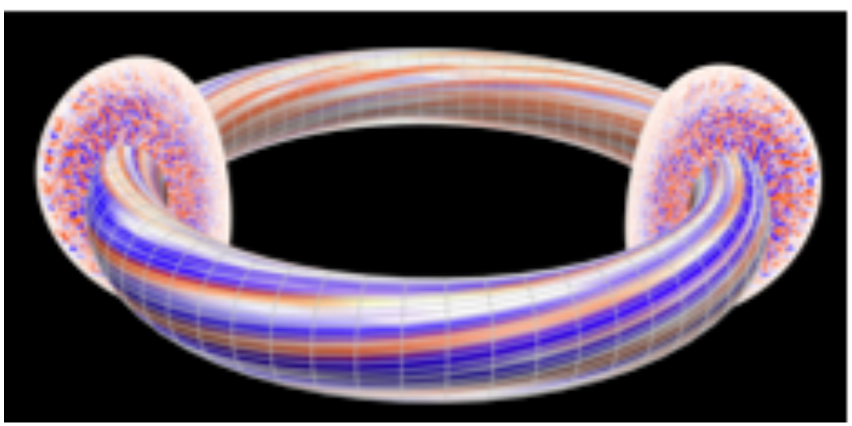
- Barriers (blocking)
  - upc_barrier
    - like "next" operation in HJ
- Split-phase barriers (non-blocking)
  - upc_notify
    - like explicit (non-blocking) signal on an HJ phaser
  - upc_wait
    - upc_wait is like explicit wait on an HJ phaser

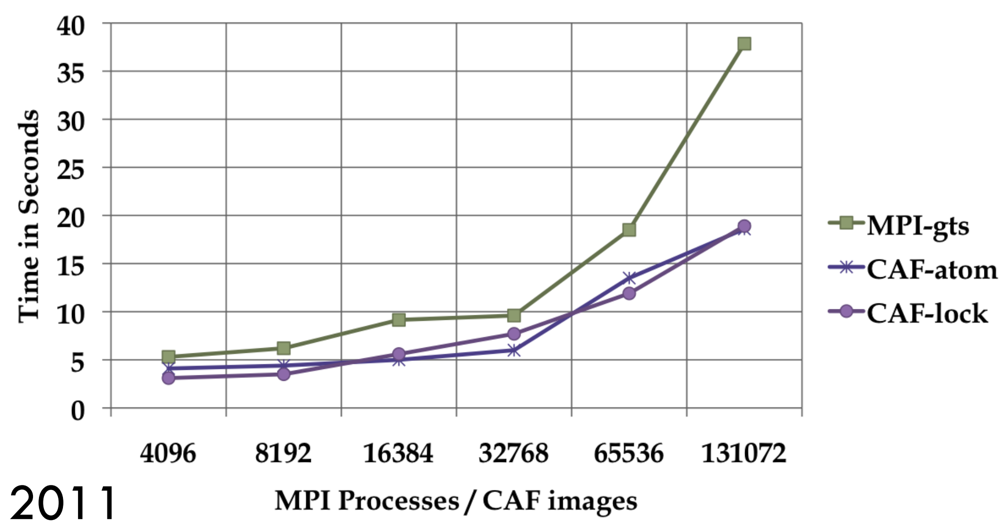
- Lock primitives
  - void upc_lock(upc_lock_t *l)
  - int upc_lock_attempt(upc_lock_t *l) // like trylock()
  - void upc_unlock(upc_lock_t *l)

Application Work in PGAS

- **Network simulator in UPC (Steve Hofmeyr)**
- **Barnes-Hut in UPC (Marc Snir et al)**
- **Landscape analysis**
  - **"Contributing Area Estimation" in UPC (Brian Kazian, UCB)**
- **Gyrokinetic Tokamak Simulation Shifter code in CoArray Fortran (CAF)**
  - **Preissl, Wichmann, Long, Shalf, Ethier, Koniges (LBNL, Cray, PPPL)**







Slide credit: Kathy Yelick, January 2011