

---

# COMP 322: Fundamentals of Parallel Programming

## Lecture 32: Partitioned Global Address Space (PGAS) programming models

Zoran Budimlić and Mack Joyner  
{zoran, mjoyner}@rice.edu

<http://comp322.rice.edu>



# Worksheet #31: PageRank Example

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

Net ID: \_\_\_\_\_

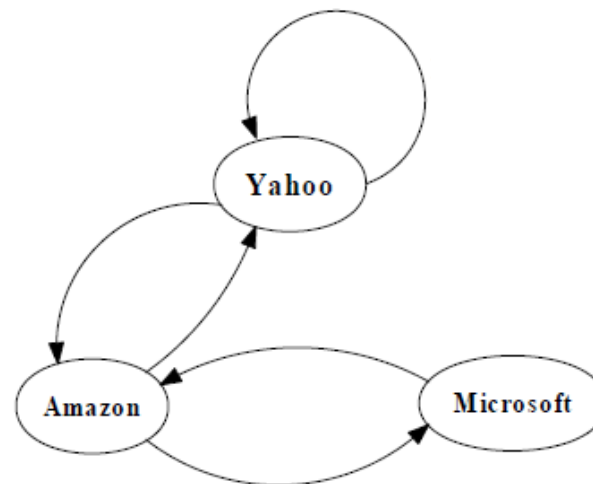
In the space below, indicate what you expect the relative ranking to be for the three pages below (with the given links). Show your computation (approximations are fine).

Final, after 7 iterations:

(1) Amazon = 1.22

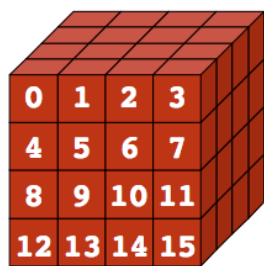
(2) Yahoo = 1.15

(3) Microsoft = 0.65

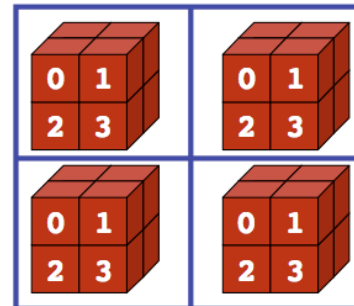


# Partitioned Global Address Space Languages

- Global address space
    - one-sided communication (GET/PUT)
  - Programmer has control over performance-critical factors
    - data distribution and locality control
    - computation partitioning
    - communication placement
  - Data movement and synchronization as language primitives
    - amenable to compiler-based communication optimization
  - “Global view” rather than “local view”
- simpler than two-sided message passing in MPI
  - lacking in thread-based models
  - HJ places (Lecture 34) help with locality control but not with data distribution



Global View



Local View (4 processes)



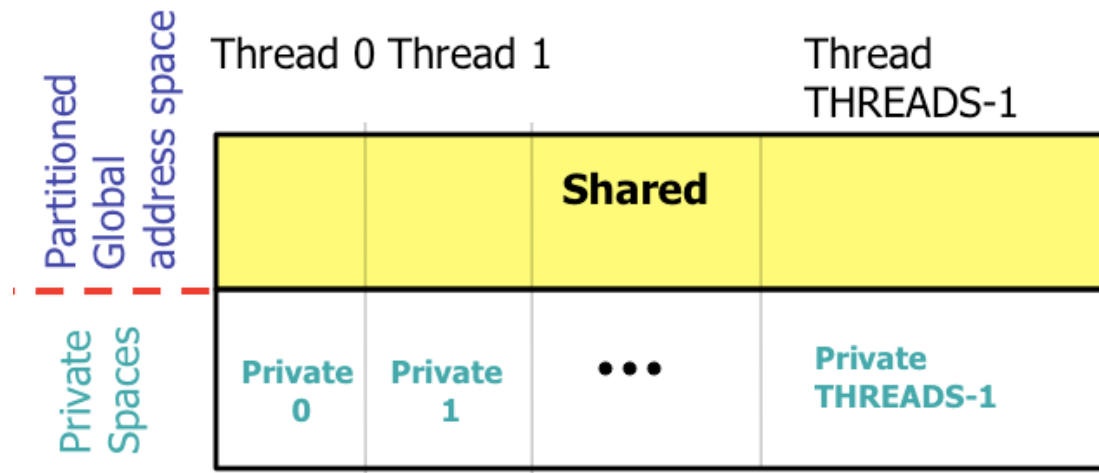
# Partitioned Global Address Space (PGAS) Languages

- Unified Parallel C (C) <http://upc.wikinet.org>
  - Titanium (early Java) <http://titanium.cs.berkeley.edu>
  - Coarray Fortran 2.0 (Fortran) <http://caf.rice.edu>
  - **UPC++** (C++) <https://bitbucket.org/upcxx>
  - **Habanero-UPC++** (C++) <http://habanero-rice.github.io/habanero-upc/>
  - **OpenSHMEM** (C) <http://openshmem.org/site/>
- 
- 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



# PGAS model

- A collection of “threads” (like MPI processes) operating in a partitioned global address space that is logically distributed across threads.
- Each thread has *affinity* with a portion of the *globally shared* address space. Each thread has also a *private* space.
- Elements in the partitioned global space co-located with a thread are said to have *affinity* to that thread.



# Unified Parallel C (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 time
- Partitioned Global Address Space (different from MPI)
  - A pointer-to-shared can reference all locations in the shared space
  - A pointer-to-local (“plain old C pointer”) may only reference addresses in its private space or addresses in its portion of the shared space
  - Static and dynamic memory allocations are supported for both shared and private memory
- 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 non-array instance of any type (could be a struct, for example)
  - by default, elements of shared arrays are allocated “round robin” among memory modules co-located with each thread (cyclic distribution)

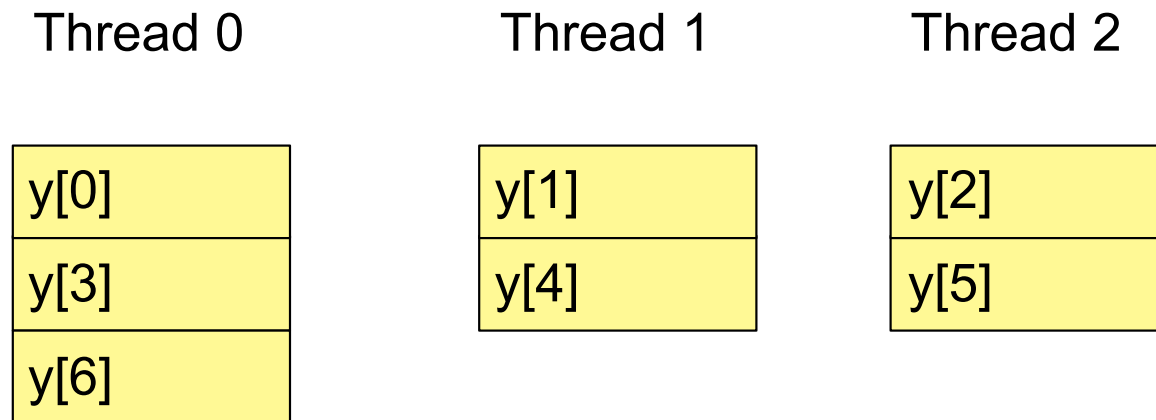


# A One-dimensional Shared Array

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

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]



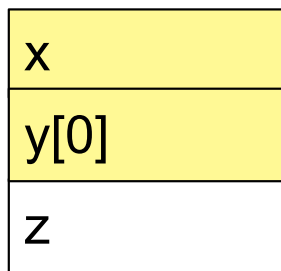
## Shared and Private Data

Consider the following data layout directives

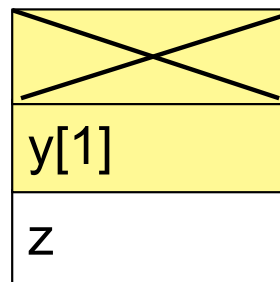
```
shared int x; // x has affinity to thread 0
shared int y[THREADS];
int z;      // private
```

For THREADS = 3, we get the following layout:

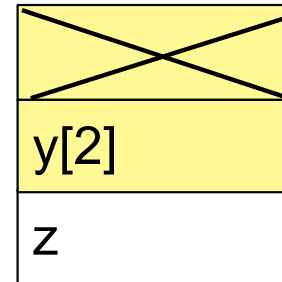
Thread 0



Thread 1

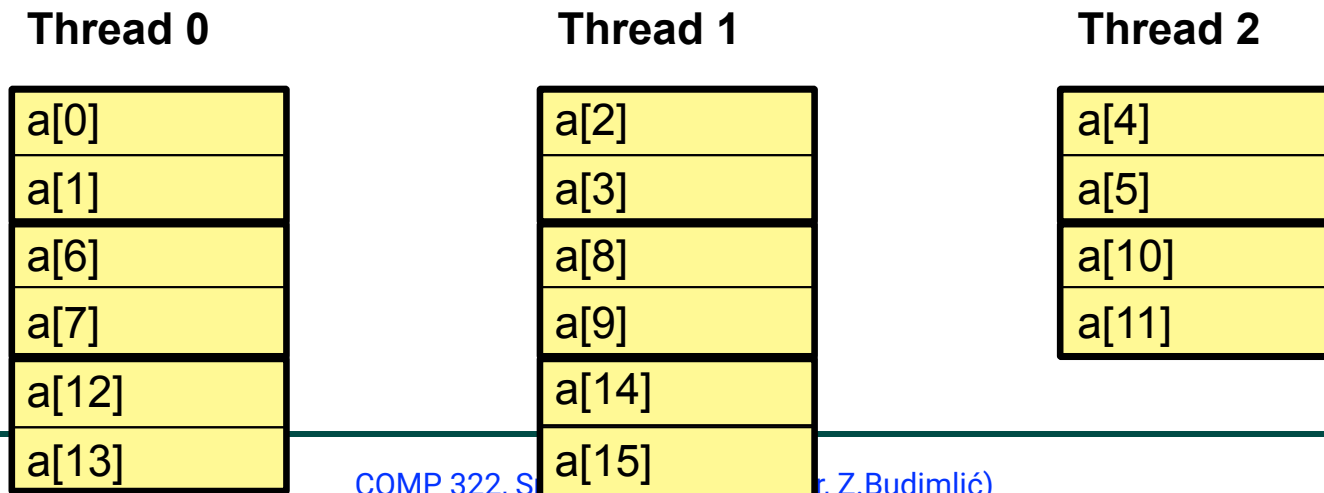


Thread 2



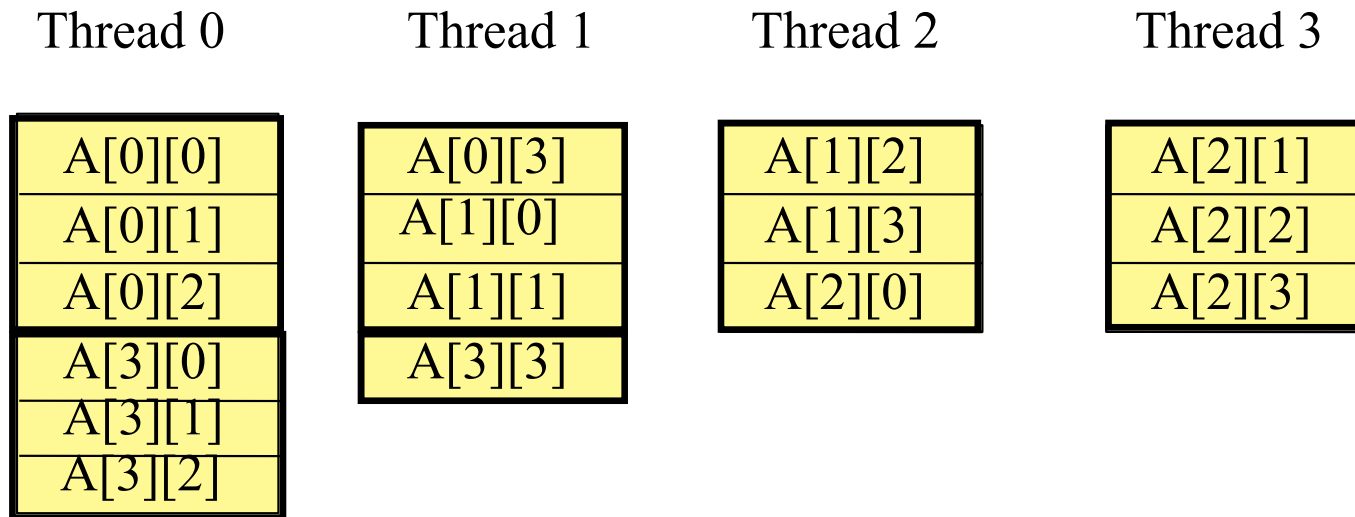
# Controlling the Layout of Shared Arrays

- 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]`



## Blocking Multi-dimensional Data

- Consider the data declaration  
– `shared [3] int A[4][THREADS];`
- When THREADS = 4, this results in the following data layout



# 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 should perform 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, the goal is to map iteration to thread where (all/most of) the iteration's data resides
- Affinity can be
  - an integer expression (with implicit mod on NUMTHREADS), or a
  - reference to (address of) a shared object



## Work Sharing + Affinity with `upc_forall`

- Example 1: explicit data 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
    a[i] = b[i] * c[i];
```

- Example 2: implicit data 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 thread i%THREADS
    a[i] = b[i] * c[i];
```

- Both yield a round-robin distribution of iterations





## 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] * c[i];
```

- Assuming 4 threads, the distribution of `upc_forall` iterations is as follows:

iteration 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

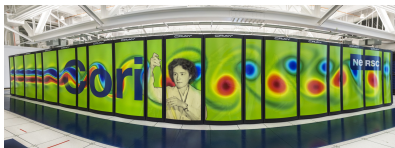
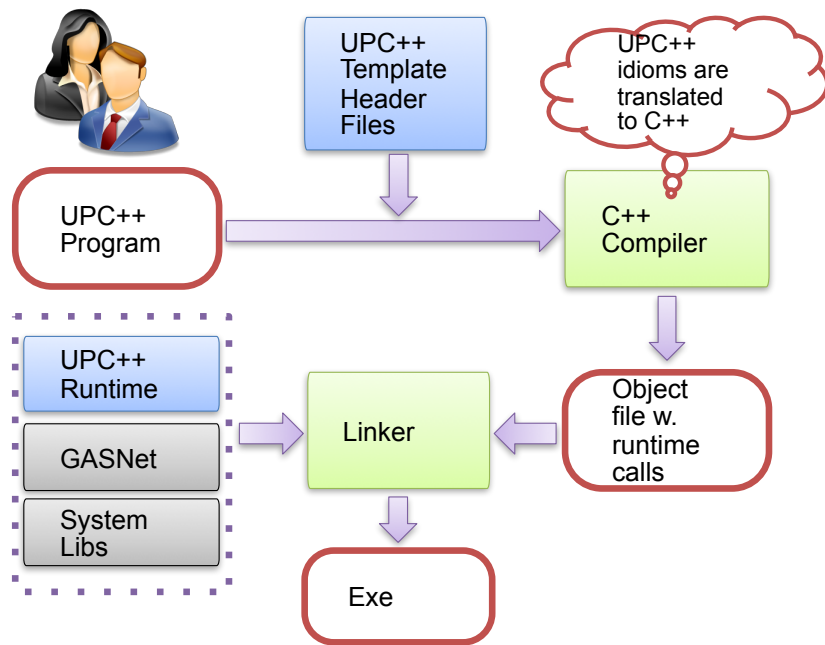


# Synchronization in UPC

- 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)`



# UPC++ library: a “Compiler-Free” Approach for PGAS (source: LBNL)



- Leverage C++ standards and compilers
  - Implement UPC++ as a C++ template library
  - C++ templates can be used as a mini-language to extend C++ syntax
- Many new features in C++11
  - E.g., type inference, variadic templates, lambda functions, r-value references
  - C++ 11 is well-supported by major compilers



# Habanero-UPC++: Extending UPC++ with Task Parallelism (LBNL, Rice)

```
1. finish ( [capture_list1] () {
2.     // Any Habanero dynamic tasking constructs
3.     . . . // finish, async, asyncAwait
4.     . . .
5.     // Remote function invocation
6.     asyncAt ( destPlace, [capture_list2] ( ) {
7.         Statements;
8.     });
9.     . . .
10.    // Remote copy with completion signal in result
11.    asyncCopy ( src, dest, count, ddf=NULL );
12.    . . .
13.    asyncAwait(ddf, ...); // local
14.}); // waits for all local/remote async's to complete
```

"HabaneroUPC++: A Compiler-free PGAS Library." V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, V. Sarkar, PGAS 2015.



# Example code structure from an application run on ORNL supercomputer (LSMS)

## MPI version:

```
// Post MPI_IRecv() calls
...
// Post MPI_Isend() calls
...
// Perform all MPI_Wait()
// calls
...
// Perform tasks
// Each task needs results
// from two MPI_IRecv() calls
... async(...)
```

MPI version waits for all IRecv() calls to complete before executing all tasks (like a barrier)

## Habanero-UPC++ version:

```
// Issue one-sided
// asyncCopy() calls
...
// Issue data-driven tasks
// in any order without any
// wait/barrier operations
hcpp::asyncAwait(
    result1, result2,
    [=]() { task body });
...
```

Habanero-UPC++ version specifies that each asyncAwait() task can complete when its two results become available from asyncCopy() calls

