

# CSE502: Foundations of Parallel Programming

## Lecture 23: Parallel Programming in Partitioned Global Address Space

Vivek Kumar

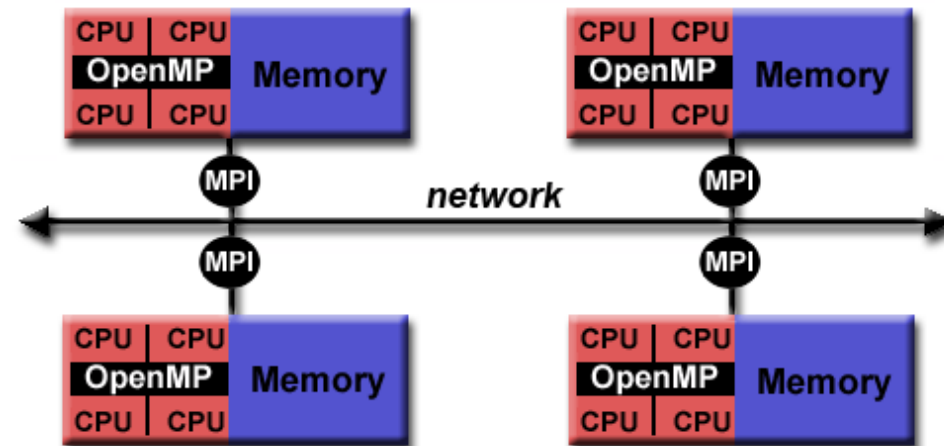
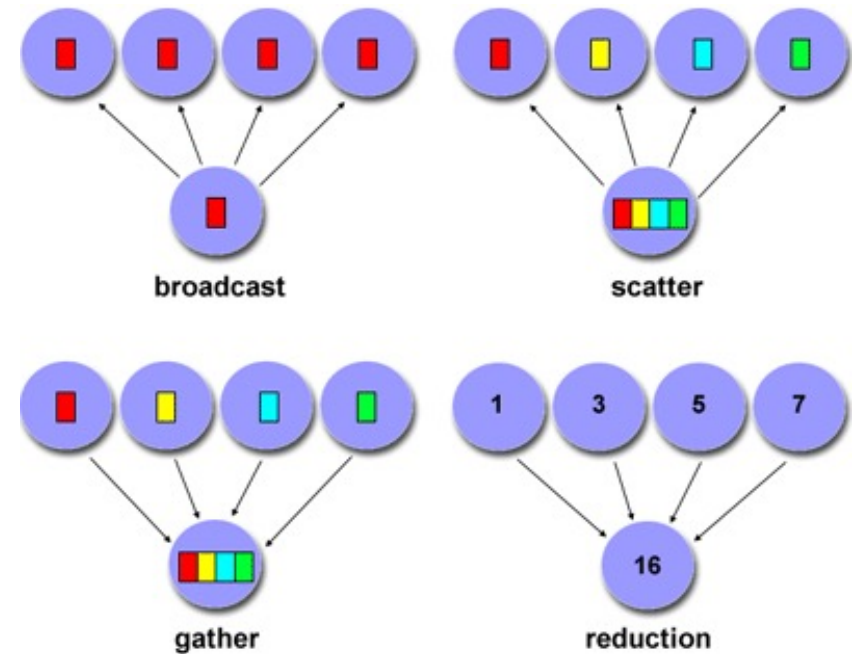
Computer Science and Engineering

IIIT Delhi

[vivekk@iiitd.ac.in](mailto:vivekk@iiitd.ac.in)

# Last Class

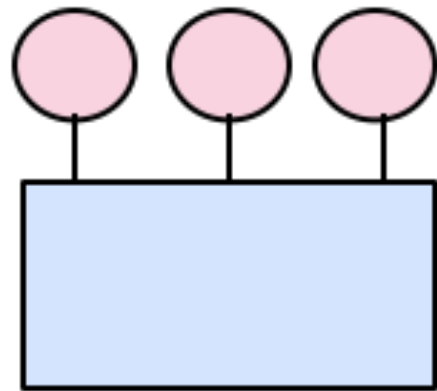
- Collective communications in MPI
  - One to Many (Broadcast, Scatter)
  - Many to One (Reduce, Gather)
  - Many to Many (AllReduce, Allgather)
- Hybrid parallelism by using OpenMP thread-level parallelism in MPI processes



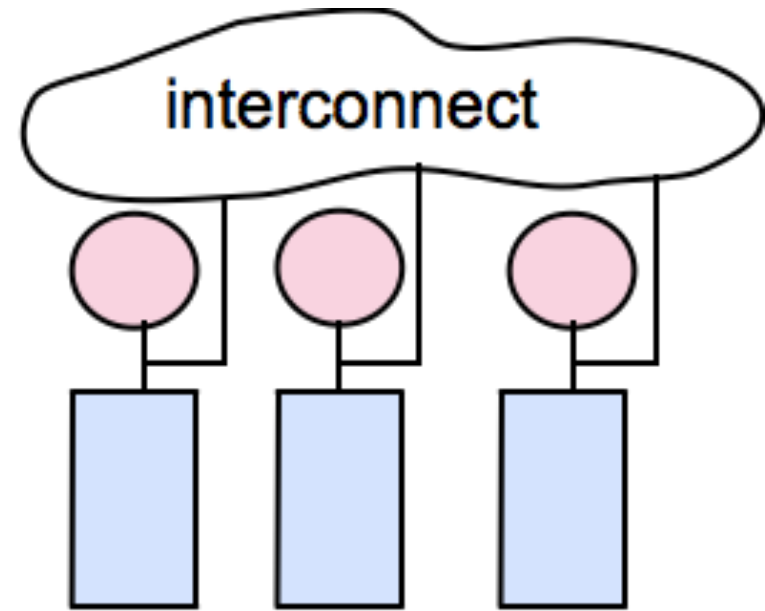
# Today's Class

- PGAS programming model
  - (Slides acknowledgements: Prof. Vivek Sarkar and Prof. John Mellor-Crummey, COMP422 course, Rice University)
- UPC++
- HabaneroUPC++

# Idealized Parallel Architecture



Shared Memory

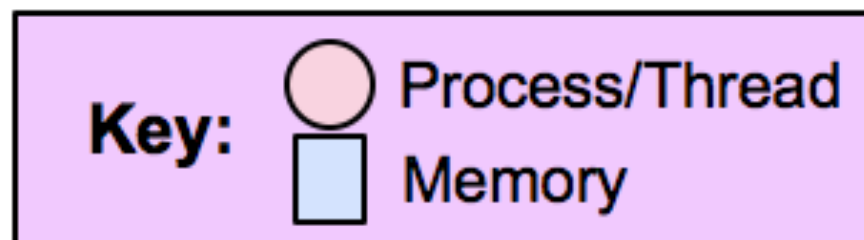


Distributed Memory

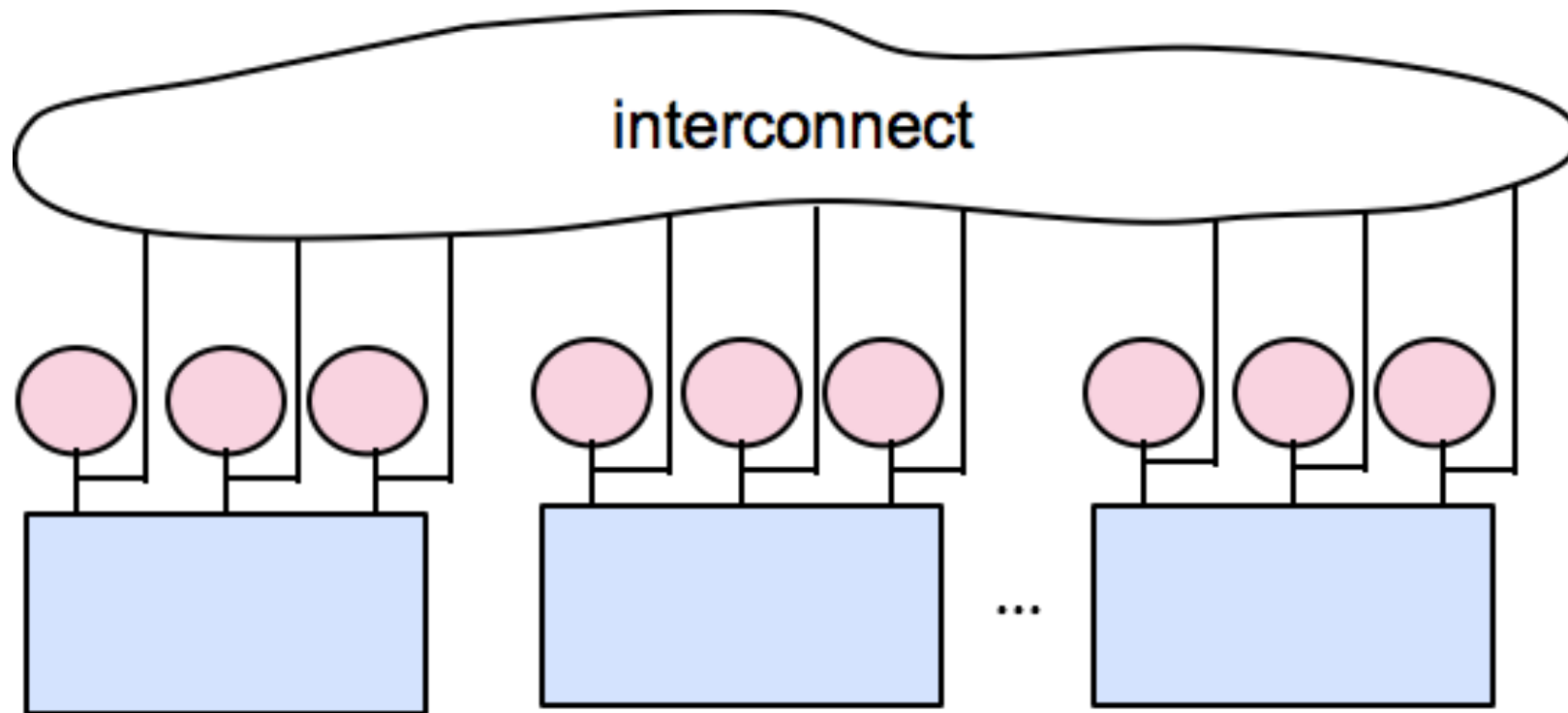
## Programming Models

HClib  
OpenMP  
Pthreads

MPI  
UPC++



# Idealized Parallel Architectures of Today



Hybrid Shared + Distributed Memory

Programming Models

e.g., MPI + OpenMP  
PGAS models

# PGAS Languages

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



Since 1987 - Covering the Fastest Computers in the World and the People Who Run Them

- Home
- Topics
- Sectors
- Exascale
- Specials
- Resource Library
- Podcast**
- Events
- Solution Channels
- Job Bank
- About
- Subscribe

## DARPA Selects Cray and IBM for Final Phase of HPCS

By Michael Feldman  
November 24, 2006

This week, the Defense Advanced Research Projects Agency (DARPA) selected Cray and IBM as the two Phase III developers for the High Productivity Computing Systems (HPCS) program. Initiated in 2002, the program is designed to produce a new generation of cost-effective, highly productive petascale systems for national security, scientific research and industrial users. The first two phases of HPCS were devoted to critical concept studies and assessments, preliminary research and development, and risk reduction engineering. Over the next four years, the third and final phase of the program will encompass development and demonstration of the HPCS technologies, culminating in a prototype system by each of the two vendors in 2010.

"This is a great day for Cray and the worldwide supercomputing community," said Peter Ungaro, Cray's president and CEO. "The DARPA HPCS program is an important force that is shaping the future of HPC and the entire computer industry. With this Phase III award, DARPA has recognized Cray as a leading innovator with the technology, vision and expertise required to deliver world-class, revolutionary supercomputing systems."

"IBM, DARPA and the mission partners will collaborate to develop a powerful and innovative design that will enhance the ability of supercomputers to help government, businesses and individuals," said Bill Zeitler, senior vice

X10 (IBM)  
Chapel (Cray)  
Fortress (Sun)



# General View

A collection of threads (in actual it's a process) operating in a **partitioned global address space** that is logically distributed among threads. Each thread has affinity with a portion of the globally shared address space. Each thread has also a private space.

Elements in partitioned global space belonging to a thread are said to have **affinity** to that thread.

# General Idea (Code from UPC++)

```
operator T*() const
{
    if (this->where() == global_myrank()) {
        // return raw_ptr if the data pointed to is on the same rank
        return this->raw_ptr();
    }

#if GASNET_PSHM
    return (T*)pshm_remote_addr2local(this->where(), this->raw_ptr());
#else
    // return NULL if this global address can't be casted to a valid
    // local address
    return NULL;
#endif
}
```



# Vector Addition in Shared Memory

```
#define N 10000
```

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

```
main(int argc, char** argv) {
```

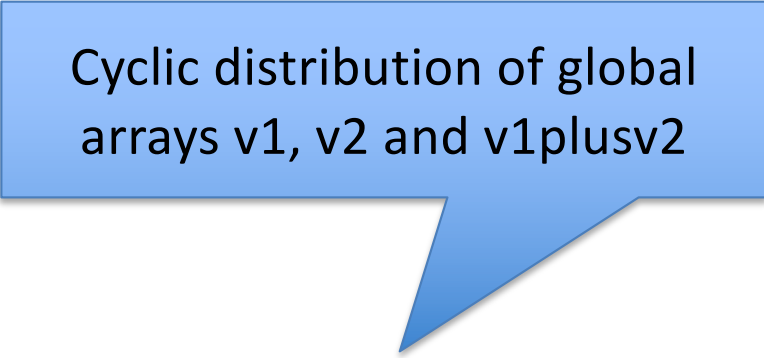
```
    for(int i=0; i<N; i++;)
```

```
        v1plusv2[i]=v1[i]+v2[i];
```

```
}
```

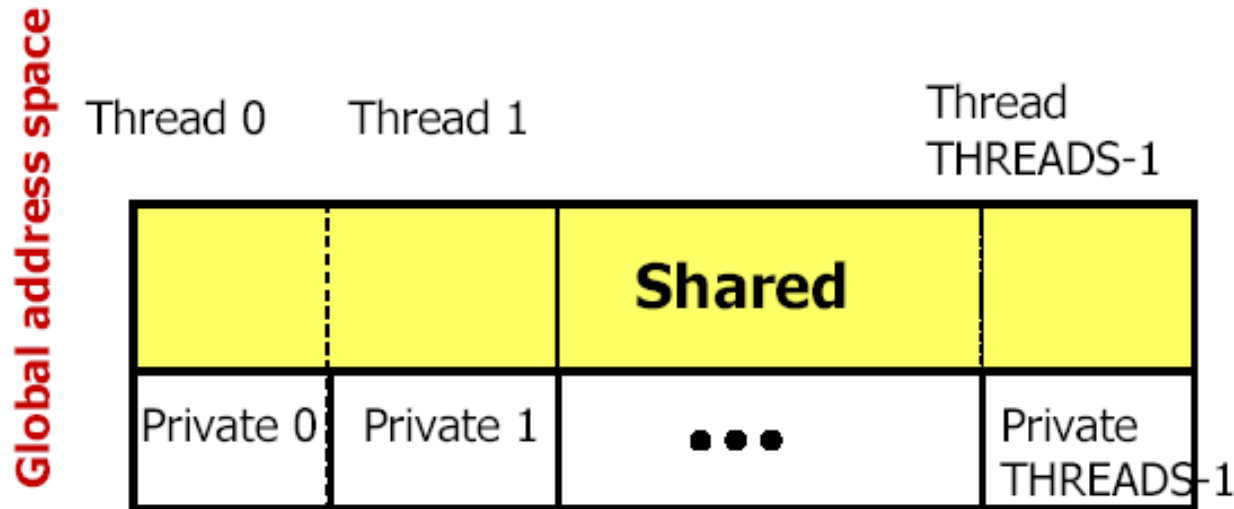
# Vector Addition in UPC++ (PGAS)

```
#include <upcxx.h>
#define N 10000
using namespace upcxx;
shared_array<int> v1(N), v2(N), v1plusv2(N);
main(int argc, char** argv) {
    init(&argc, &argv);
    for(int i=MYTHREAD; i<N; i+=THREADS;)
        v1plusv2[i]=v1[i]+v2[i];
    finalize();
}
```



Cyclic distribution of global arrays v1, v2 and v1plusv2

# UPC++ Memory Model



- 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

# UPC++ Execution Model

- A number of threads working independently in SPMD fashion
  - Similar to MPI
  - MYTHREAD specifies thread index (0..THREADS-1)
  - Number of threads specified at compile-time or run-time
- Synchronization only when needed
  - Barriers
  - Locks

# Shared and Private Data (1/2)

- Static and dynamic memory allocation of each type of data

```
T * local_ptr = (T*) malloc(sizeof(T) * count);
```

```
T* global_ptr = upcxx::allocate(MYTHREAD, count);
```

```
T* local_copy = (T*) global_ptr;
```

```
local_copy [count - 1] = 10;
```

# Shared and Private Data (2/2)

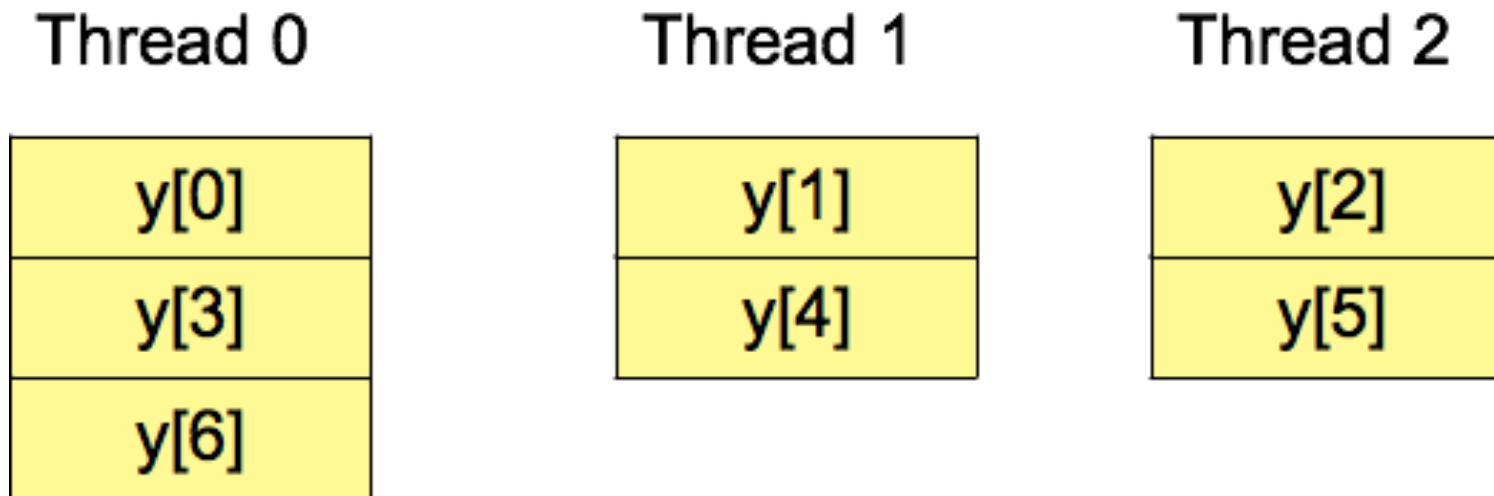
- Static and dynamic memory allocation of each type of data
- Shared objects placed in memory based on affinity
  - shared scalars have affinity to thread 0
  - here, a scalar means a non-array instance of any type (could be a struct, for example)
- by default, elements of shared arrays are allocated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

# A One-Dimensional Array (Cyclic)

- Consider the following data layout

`upcxx::shared_array<int> y(7)`

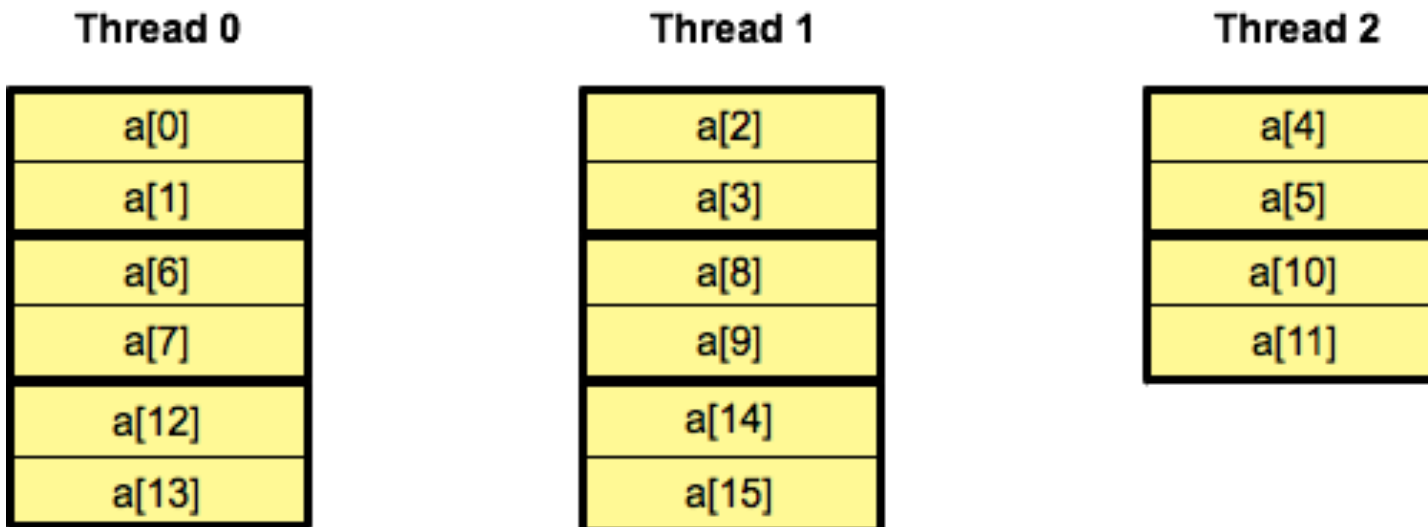
For `THREADS=3`, we get following cyclic layout



# A One-Dimensional Array (Block Cyclic)

- 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

```
upcxx::shared_array<int, 2> a(16); // Block Size = 2
```

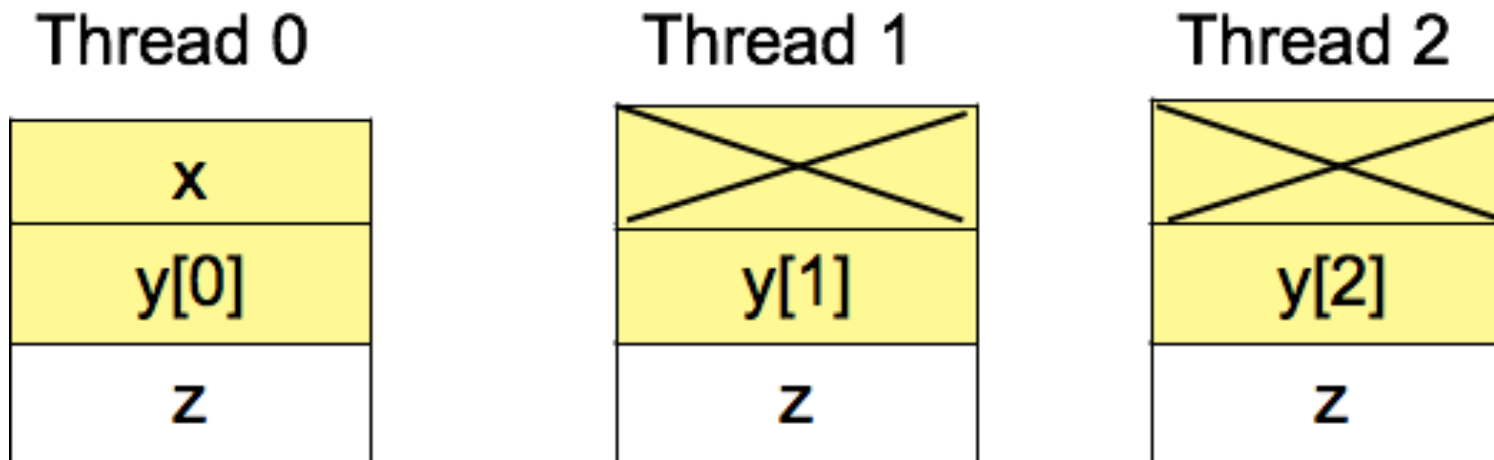




# Shared and Private Data

- Consider the following data layout directives

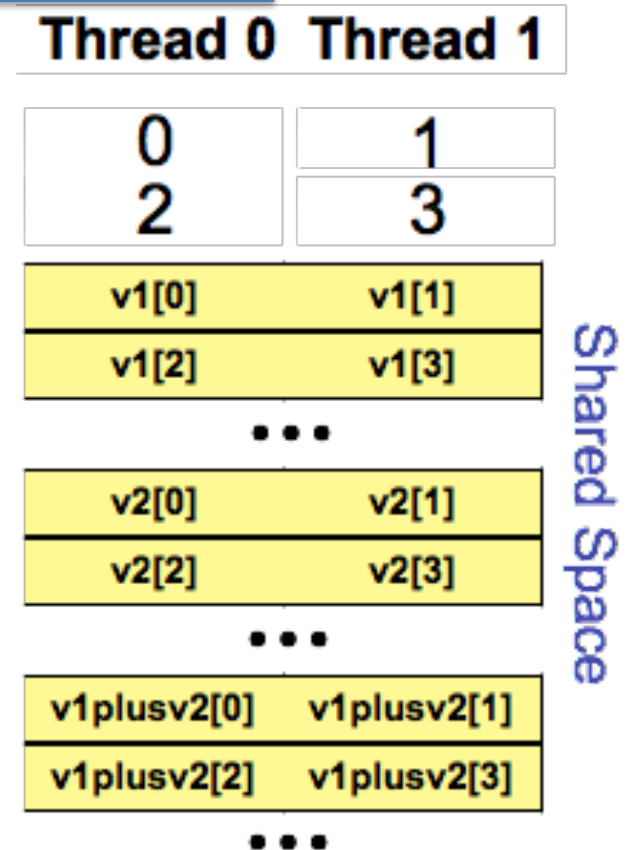
```
upcxx::shared_var<int> x; // x has affinity to thread 0
upcxx::shared_array<int> y(3);
int z; // private
```
- For **THREADS** = 3, we get the following layout



# Vector Addition in UPC++

```
#include <upcxx.h>
#define N 10000
using namespace upcxx;
shared_array<int> v1(N), v2(N), v1plusv2(N);
main(int argc, char** argv) {
    init(&argc, &argv);
    for(int i=MYTHREAD; i<N; i+=THREADS;)
        v1plusv2[i]=v1[i]+v2[i];
    finalize();
}
```

Cyclic distribution of global arrays v1, v2 and v1plusv2



This is an optimized implementation as each thread accesses only its local memory

# Synchronization in UPC++

- Barrier call

```
upcxx::barrier();
```

- Locks

```
shared_lock sv_lock;
```

```
shared_array<shared_lock> sv_lock_array(N);
```

# Collective Communication in UPC++

- `upcxx::reduce<double>(src, dst, count, root, UPCXX_SUM, UPCXX_DOUBLE);`
- Likewise other collective communication APIs as in MPI
  - `upcxx::bcast`
  - `upcxx::gather`
  - `upcxx::allgather`
  - .....

# UPC++: Drawbacks

- No inbuilt support for threading
- Rely on third party libraries (e.g. OpenMP)
  - Loose integration
    - How to overlap computation and communication?
    - Hard to avoid the overheads of enabling thread-safety in UPC++ (e.g., communication calls on OpenMP threads)
  - Not very productive

# HabaneroUPC++: Integrating Asynchronous Task Parallelism in UPC++

- Goals
  - Inter-mixing Habanero-C++ programming model with UPC++ library (SPMD program)
  - Use a modern mainstream programming language (C++)
  - Avoid the need to use a thread-safe implementation of UPC++
    - Use a dedicated communication worker in HClib

# Communication and Computation Workers in HClib

- Option to build HClib with the support for a dedicated communication worker

```
HCLIB_WORKERS = N
```

```
(communication worker_id = [0])
```

```
(computation workers = [1, n-1])
```

- Launching a communication task

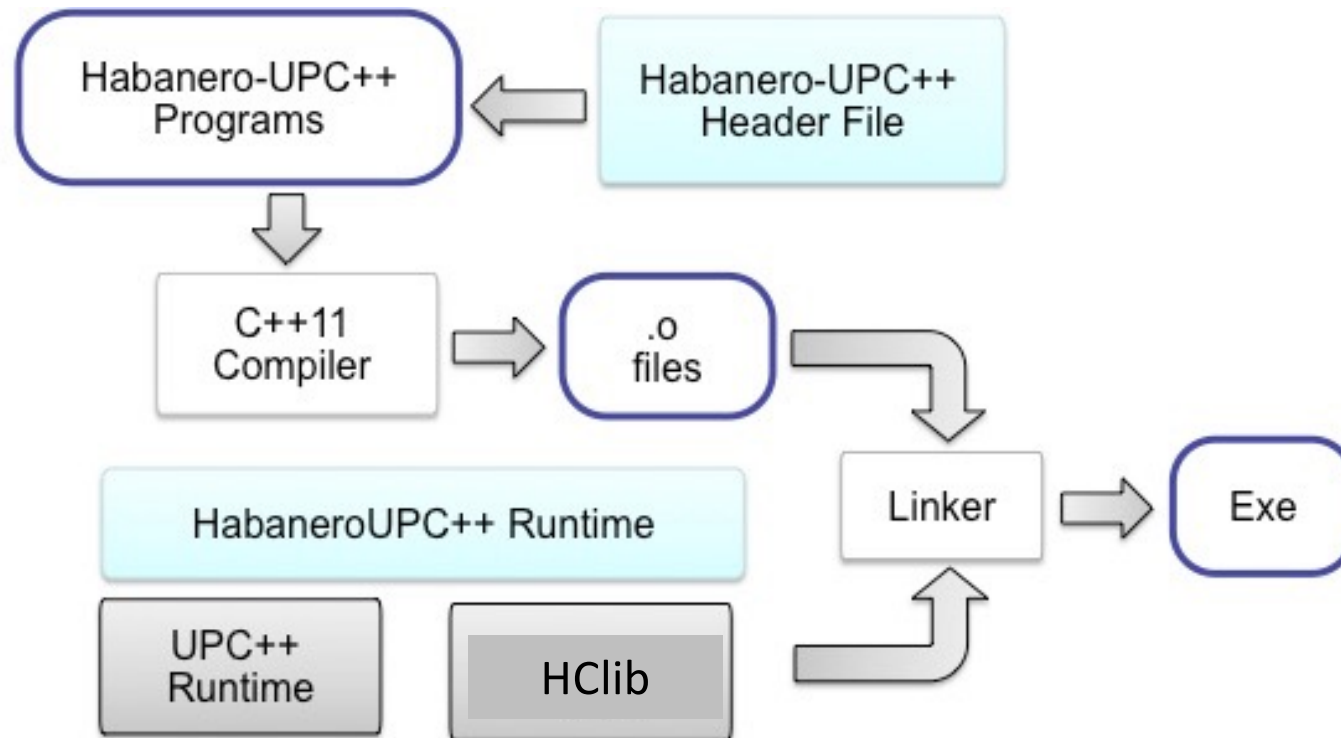
```
hclib::asyncComm ([capture_list] ( ) {  
    <Statements>  
});
```

# HabaneroUPC++: Integrating Asynchronous Task Parallelism in UPC++

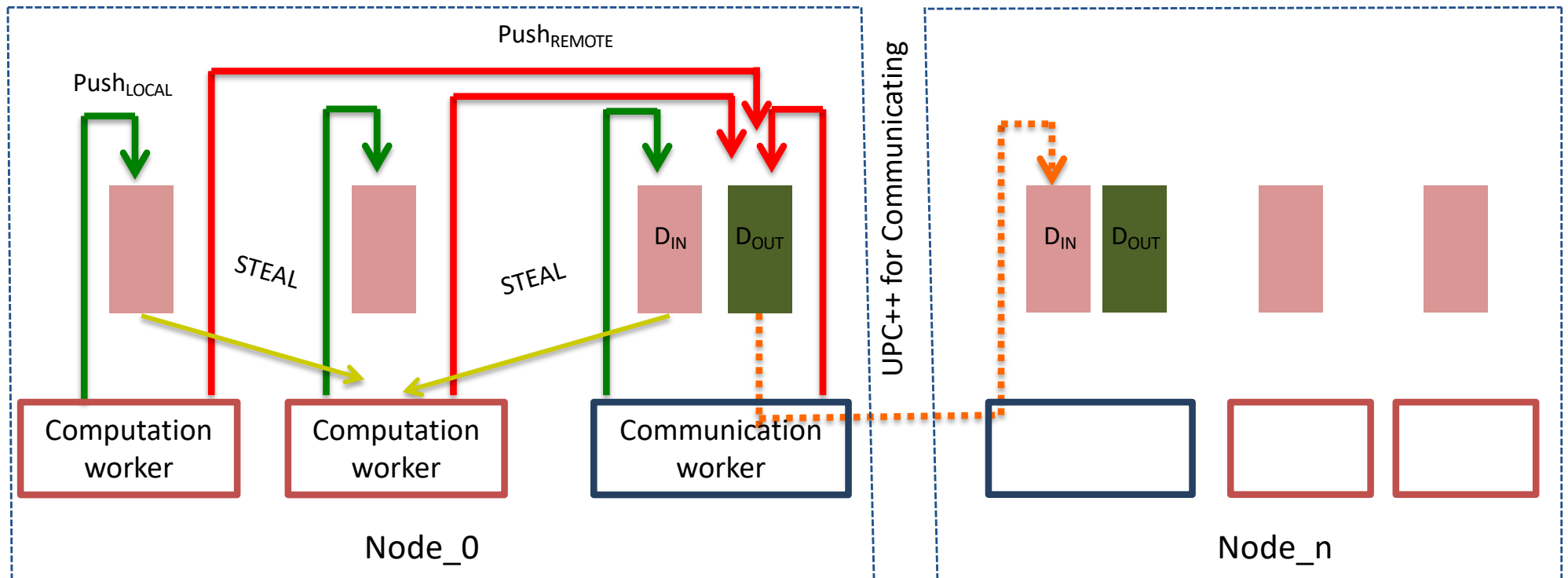
```
finish_spm([=]()) {  
    // “intra-node” asynchronous tasks that is  
    // executed only by computation workers  
    async(..., [=]() {...});  
    forasync(..., [=]() {...});  
    async_await(..., [=]() {...});  
    async_future(..., [=]() {...});  
  
    // “inter-node” asynchronous tasks that is  
    // executed only by communication worker  
    async_copy(src, dst, size, ...);  
    async_at(remote_rank, [=]() { ... });  
  
    // “locality-free” asynchronous tasks  
    // uses distributed work-stealing  
    asyncAny([=]() { ... });  
});
```



# HabaneroUPC++ Software Stack



# Integrating HClib with UPC++



```

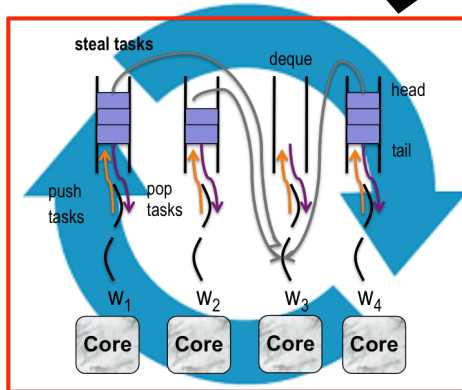
main ( ) {
  finish_spm ( [= ] () {
    local;
    remote;
  });
}

```

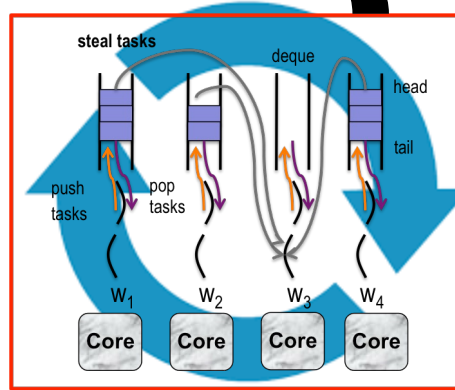
Communication worker (SPMD) ↓

# Distributed Work-Stealing in HabaneroUPC++

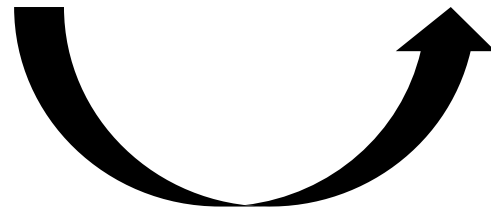
Step-1:  
Failed steal at intra-  
node level



Step-6: Remote communication worker (victim)  
can send tasks, else it's a inter-node failed steal



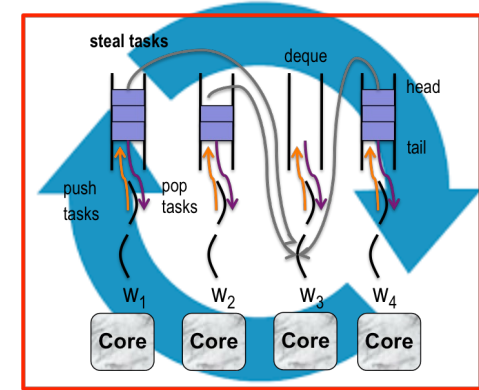
Step-2:  
Local worker  
request the local  
communication  
worker for inter-  
node steal



Step-3: Local communication worker (thief) finds a  
victim that has sufficient number of tasks

Step-4: Lock and wait for tasks from victim

Step-5: Remote communication worker attempts  
to steal tasks from its local thread-pool



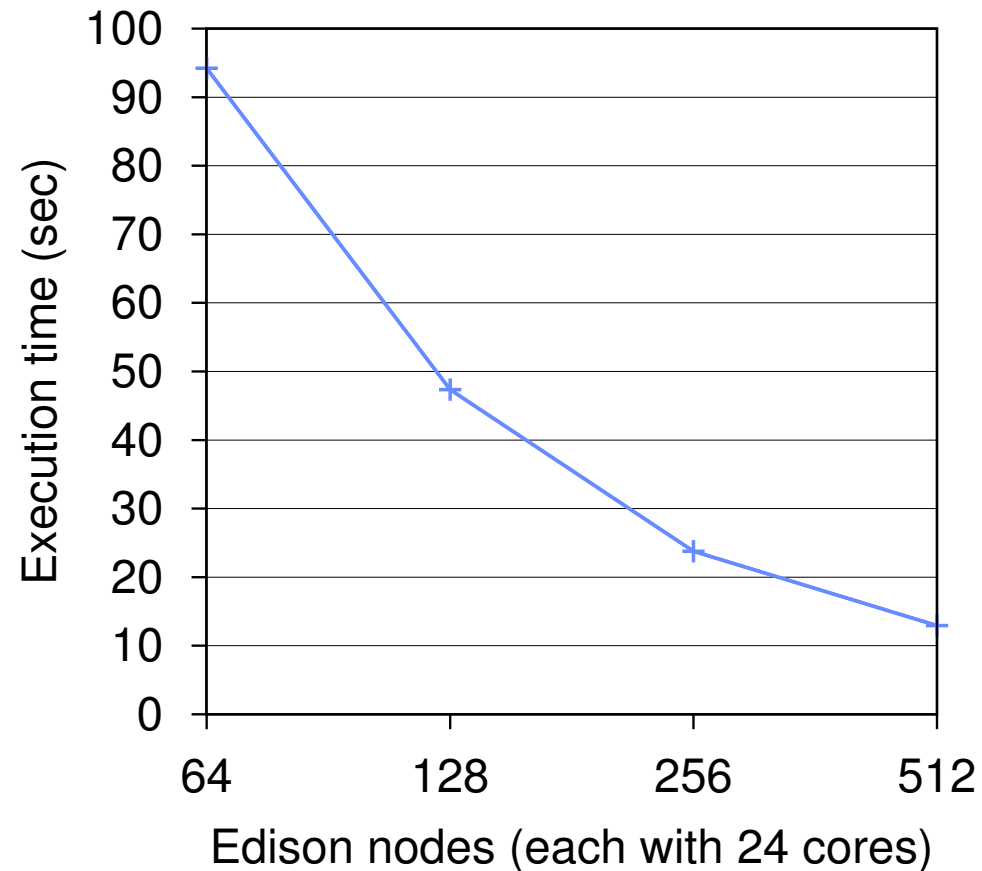
```

void nqueens(data node) {
    if(solution()) return;
    else {
        for(int i<0; i<SIZE; i++) {
            .....
            //child;
            asyncAny([child]() { nqueens(child); });
        }
    }
}
main() {
    .....
    int local_solutions[numWorkers()];
    finish_spm([=]() {
        if(MYTHREAD == 0) {
            nqueens(root);
        }
    });
    int my_sum=0, total;
    for(int i=0; i<numWorkers(); i++) my_sum+= local_solutions[i];
    upcxx::reduce(&my_sum, .....
}

```

# NQueens in HabaneroUPC++

# NQueens using HabaneroUPC++ (Distributed Work-Stealing)



# Next Classes

- End semester review lecture

# Reference Materials

- Programming in the PGAS Model
  - [http://upc.gwu.edu/tutorials/tutorials\\_sc2003.pdf](http://upc.gwu.edu/tutorials/tutorials_sc2003.pdf)
- UPC++
  - <http://ieeexplore.ieee.org/abstract/document/6877339/>
- HabaneroUPC++
  - <http://vivkumar.github.io/papers/pgas14.pdf>