# The Berkeley Container Library

## Applications of Parallel Computing

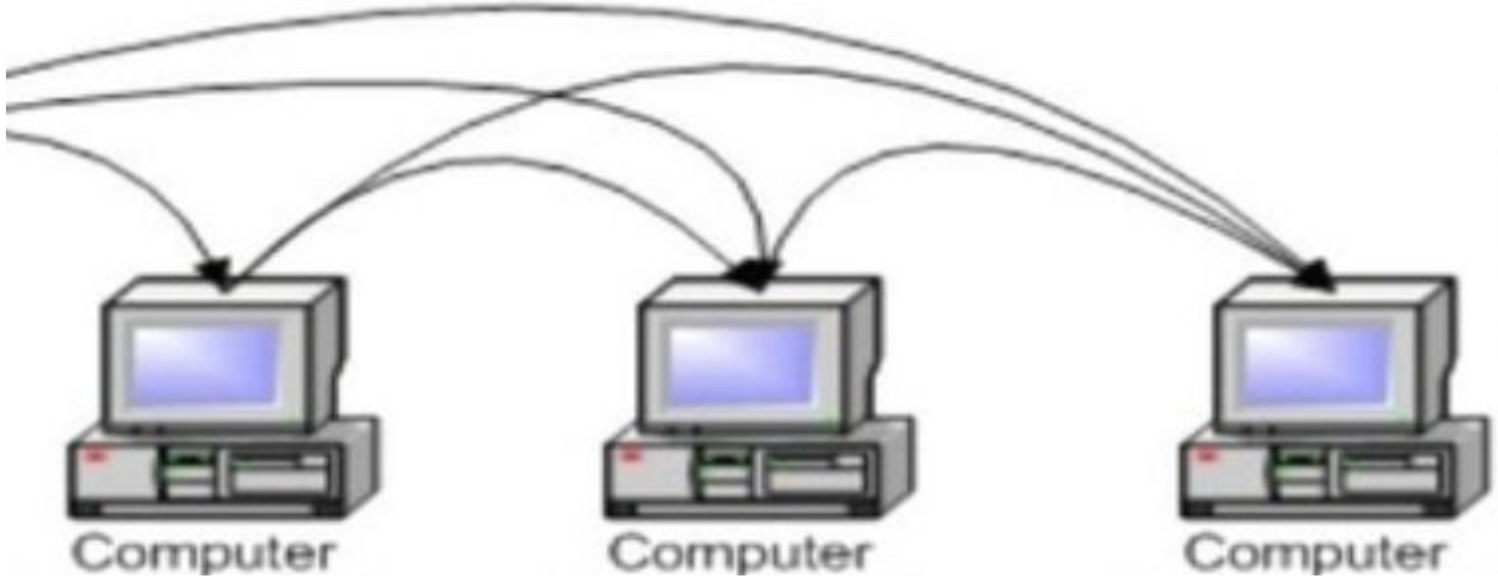
**Benjamin Brock**

February 27, 2020

# Introduction and Background

# What is a Cluster?

- A collection of **nodes**, connected by a **network**.

# How do I program one today?

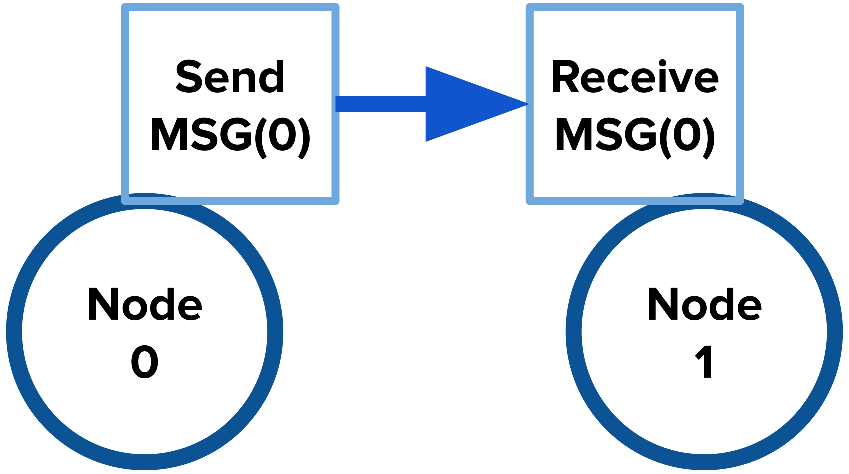

- **Message Passing** - nodes issue matching **send** and **receive** calls

- **RDMA** - directly read/write to **remote memory**

# How do I program one today?


- **Message Passing** - nodes issue matching **send** and **receive** calls

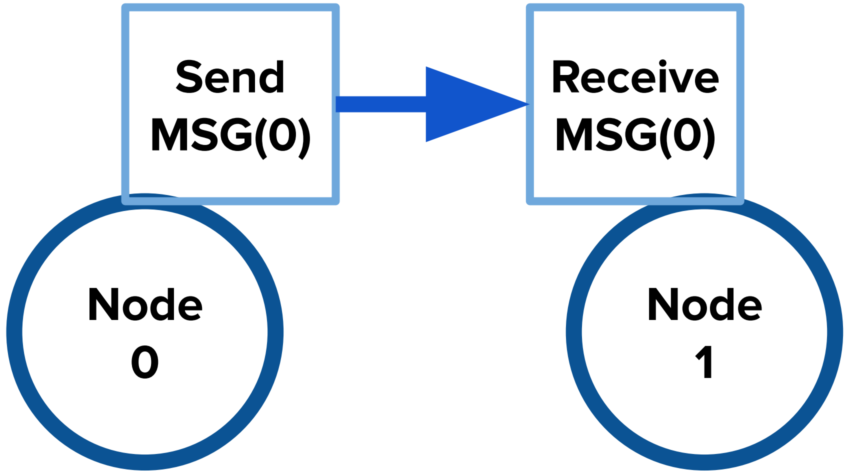
- **RDMA** - directly read/write to **remote memory**

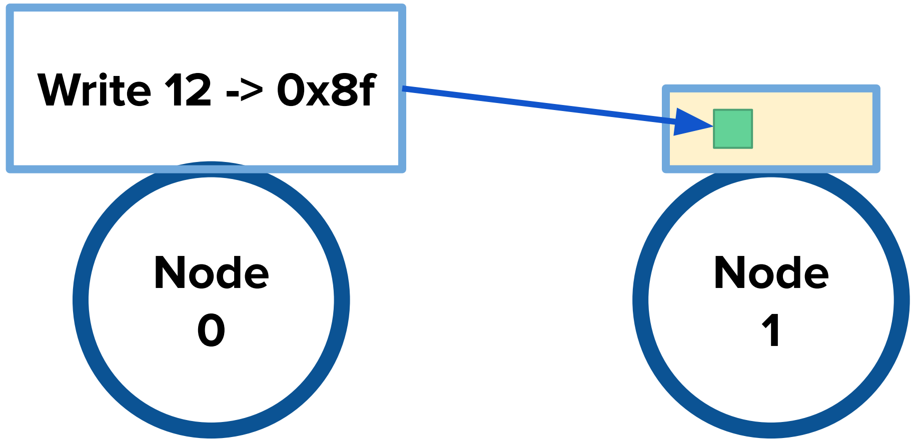


# How do I program one today?


- **Message Passing** - nodes issue matching **send** and **receive** calls

- **RDMA** - directly read/write to **remote memory**



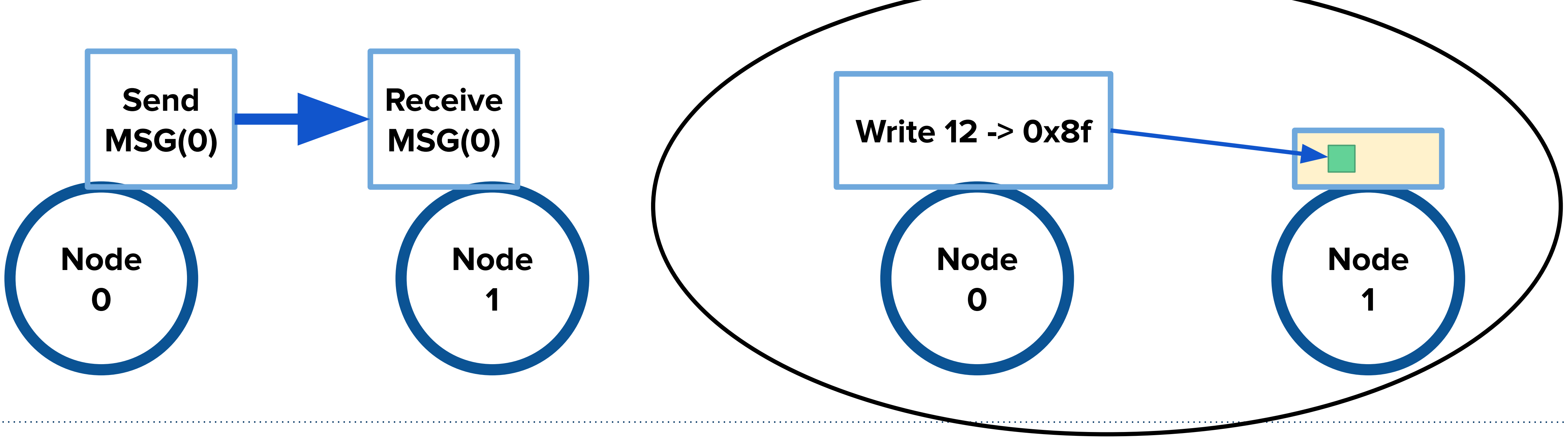




RDMA Availability Backup Slides

# How do I program one today?


- **Message Passing** - nodes issue matching **send** and **receive** calls

- **RDMA** - directly read/write to **remote memory**




RDMA Availability Backup Slides

Berkeley
UNIVERSITY OF CALIFORNIA

# How do I program one today?


- The most common parallel programming frameworks are **low level**

- **Message Passing Interface (MPI)**
- **OpenSHMEM**
- **GASNet-EX**

# Writing to a distributed array

```
global_x = new_array(...);
global_x += local_x;
```

```
dense_vec_t *init_array(int d, int nprocs, int rank) {
  int split = (d + nprocs - 1) / nprocs;
  int my_start = split * rank;
  int my_size = std::min(split, d - my_start);
  dense_vec_t *model = (dense_vec_t *) malloc(sizeof(dense_vec_t));
  model->n = my_size;
  model->win = (MPI_Win *) malloc(sizeof(MPI_Win));
  model->info = (MPI_Info *) malloc(sizeof(MPI_Info));
  MPI_Info_create(model->info);
  MPI_Info_set(*model->info, "accumulate_ordering", "none");
  MPI_Info_set(*model->info, "accumulate_ops", "same_op_no_op");
  MPI_Info_set(*model->info, "same_size", "true");
  MPI_Info_set(*model->info, "same_disp_unit", "true");
  MPI_Win_allocate(split*sizeof(int), sizeof(int), *model->info, MPI_COMM_WORLD,
                   &model->v,model->win);
  for (int i = 0; i < split; i++) {
    model->v[i] = 0.0f;
  }
  return model;
}

float *allocate_local_copy(csr_mat_t *X, int nprocs) {
  int d = X->n;
  int split = (d + nprocs - 1) / nprocs;
  float *lw = (float *) malloc(sizeof(float) * split*nprocs);
  for (int i = 0; i < split*nprocs; i++) {
    lw[i] = 0.0f;
  }
  return lw;
}
```

```
void flush(MPI_Request *requests, int n) {
  for (int i = 0; i < n; i++) {
    MPI_Wait(&requests[i], MPI_STATUS_IGNORE);
  }
}

void retrieve_local_array(dense_vec_t *w, csr_mat_t *X, int nprocs, float *lw) {
  int d = X->n;
  int split = (d + nprocs - 1) / nprocs;
  MPI_Request *requests = (MPI_Request *) malloc(sizeof(MPI_Request) * nprocs);
  for (int i = 0; i < nprocs; i++) {
    MPI_Rget_accumulate(NULL, split, MPI_FLOAT, lw + (i*split), split, MPI_FLOAT, i, 0,
                        split, MPI_FLOAT, MPI_NO_OP, *w->win, &requests[i]);
  }
  flush(requests, nprocs);
  free(requests);
}

void store_global_array(dense_vec_t *w, csr_mat_t *X, int nprocs, float *lw) {
  int d = X->n;
  int split = (d + nprocs - 1) / nprocs;
  MPI_Request *requests = (MPI_Request *) malloc(sizeof(MPI_Request) * nprocs);
  for (int i = 0; i < nprocs; i++) {
    MPI_Raccumulate(lw+(i*split), split, MPI_FLOAT, i, 0, split, MPI_FLOAT, MPI_SUM,
                    *w->win, &requests[i]);
  }
  flush(requests, nprocs);
  free(requests);
}
```

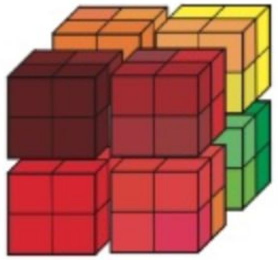
# State of the Practice

- Distributed programs are typically written using **low-level communication libraries** like MPI

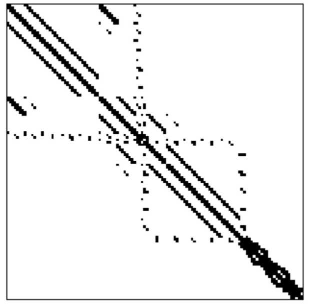
- Lack **types**, other high-level features

- Not uncommon to spend **orders of magnitude** more time **writing a program** than **developing an algorithm.**

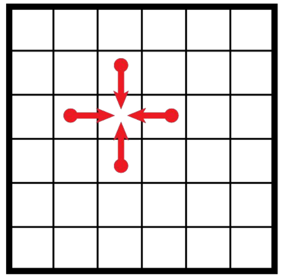
# Spectrum of Data Structures - Regular to Irregular



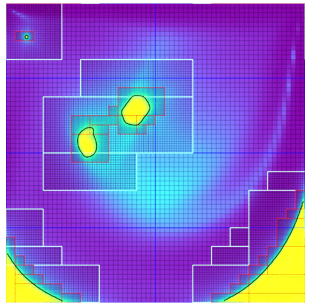
Dense Arrays



Structured Sparse Matrices

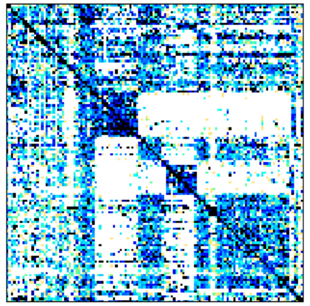


Stencils

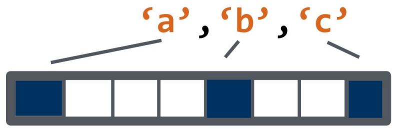


Hierarchical Grids

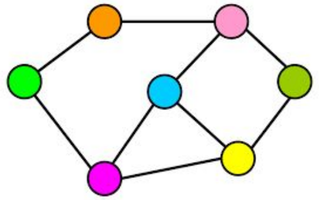
## Regular Data Structures



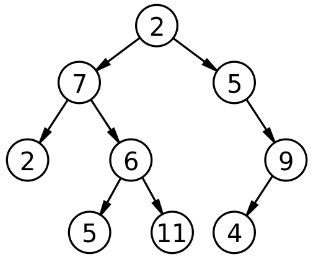
Unstructured Sparse Matrices



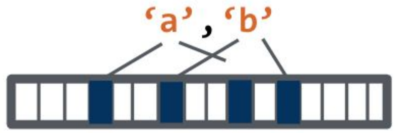

Hash Tables



Graphs




Trees

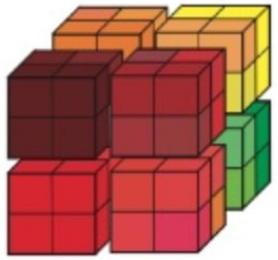



Bloom Filters

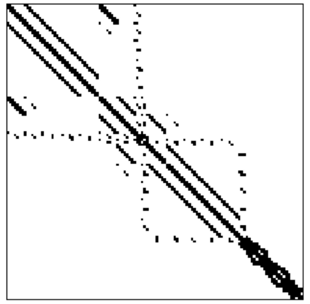
## Irregular Data Structures

Berkeley
UNIVERSITY OF CALIFORNIA

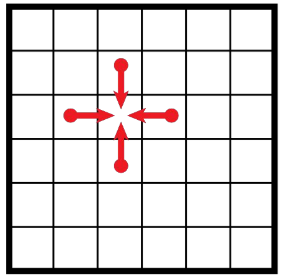
# Spectrum of Data Structures - Regular to Irregular



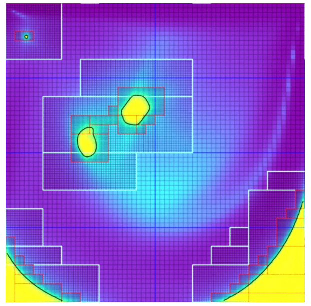
Dense Arrays



Structured Sparse Matrices

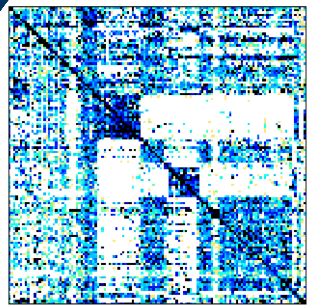


Stencils

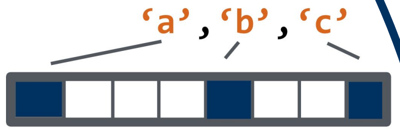


Hierarchical Grids

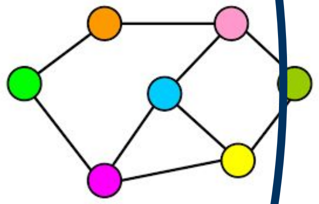
## Regular Data Structures



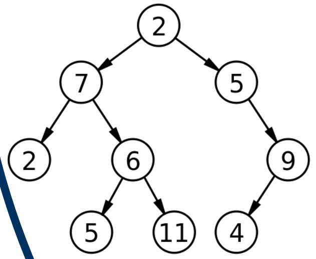
Unstructured Sparse Matrices



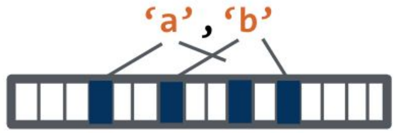

Hash Tables



Graphs




Trees




Bloom Filters

## Irregular Data Structures

# Irregular Applications

- **Dynamic** communication patterns
- **Latency** sensitive
- Coordination can be expensive
- **Asynchronous execution** necessary to prevent load imbalance

# Distributed Data Structures

- Many complex programs use **high-level abstract data structures**
- High-level programming languages and libraries enable developers to write **bespoke** data structures
- However, many programming environments are **missing** data structures libraries, particularly for **irregular problems**

# BCL Goals

- Offer **high-level** data structures for **irregular problems**
- Be **cross-platform**
- Enable **high-performance**

```cpp
BCL::Map<int, std::string> map;

for (auto& d : data) {
  map.insert({d.key, d.val});
}
..
map[key] = val;
...
```

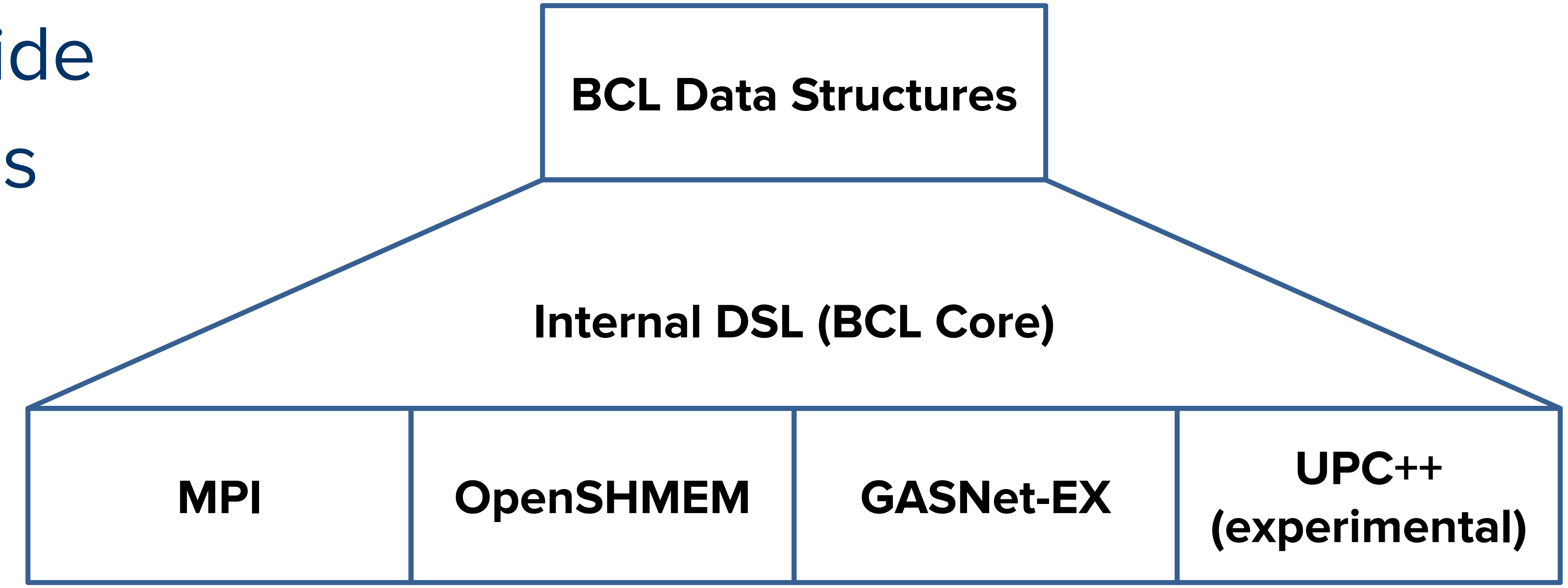
# Berkeley Container Library

# Design

- Build data structures on top of **BCL Core**, small cross-platform DSL

- **Multiple backends** provide communication primitives

- Data structures are **natively cross-platform**

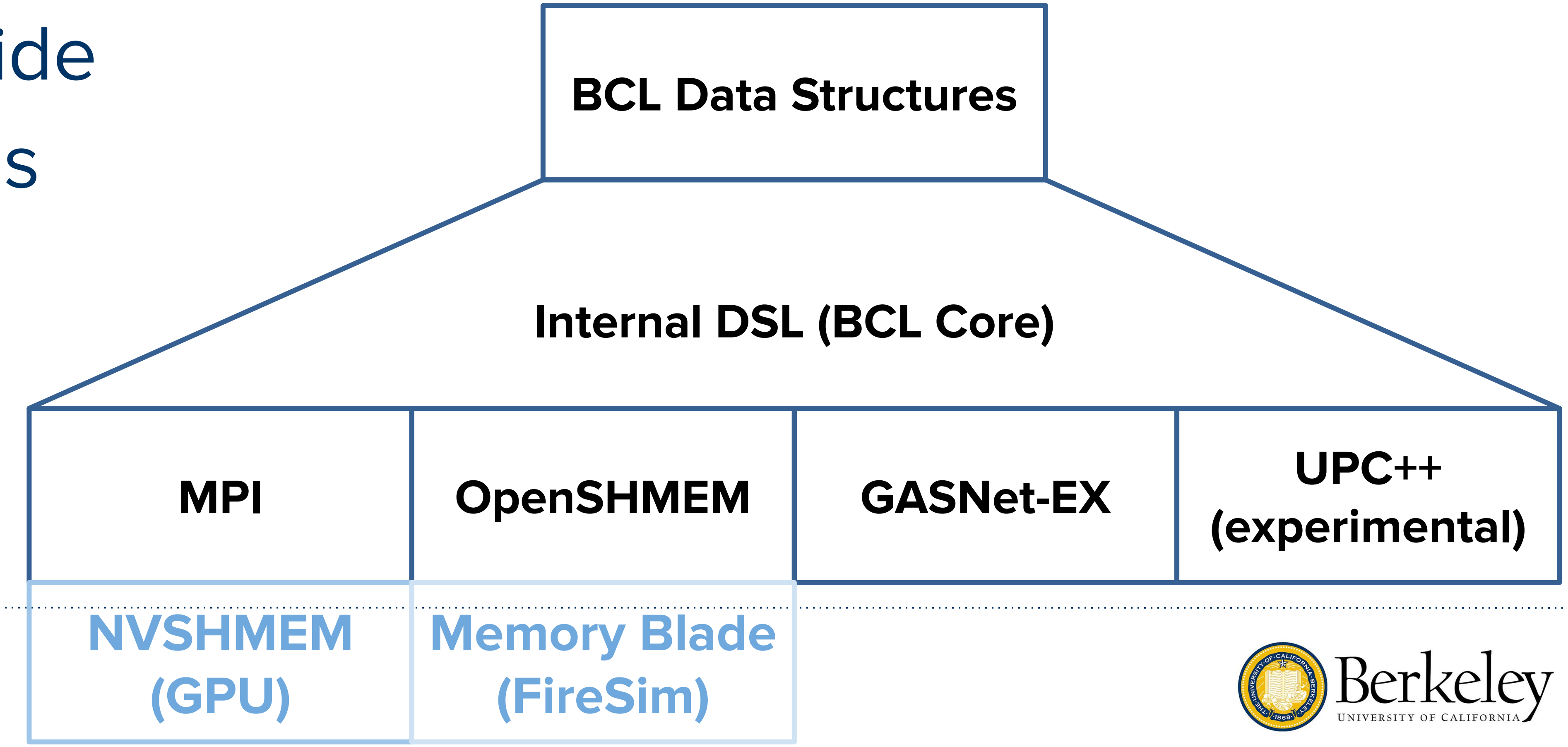


# Design

- Build data structures on top of **BCL Core**, small cross-platform DSL

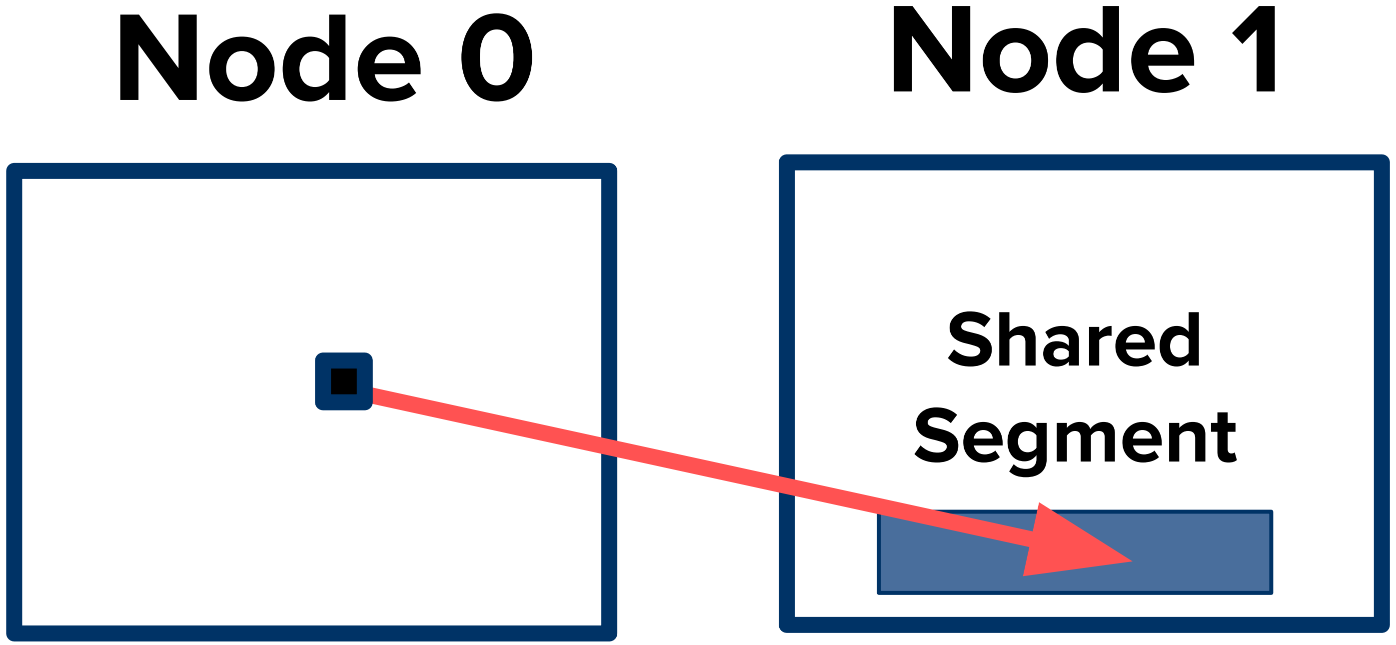
- **Multiple backends** provide communication primitives

- Data structures are **natively cross-platform**



# BCL Core

- Core primitive is **global pointer**
- Points to memory inside another process' **shared segment**
- Can **read**, **write**, perform **atomic** operations



# Data Structure Philosophy

- Use **RDMA** for all principle data structure operations

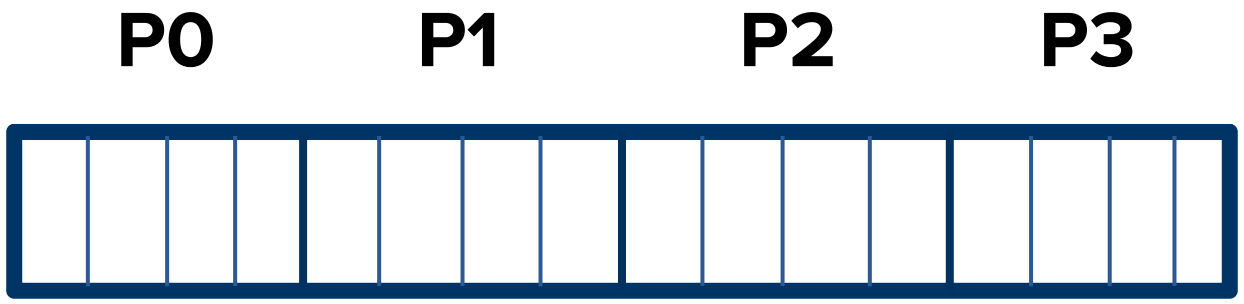
1) Executed efficiently in **hardware**

2) No need to **interrupt** remote CPU

3) **Maps well** to familiar data structure operations

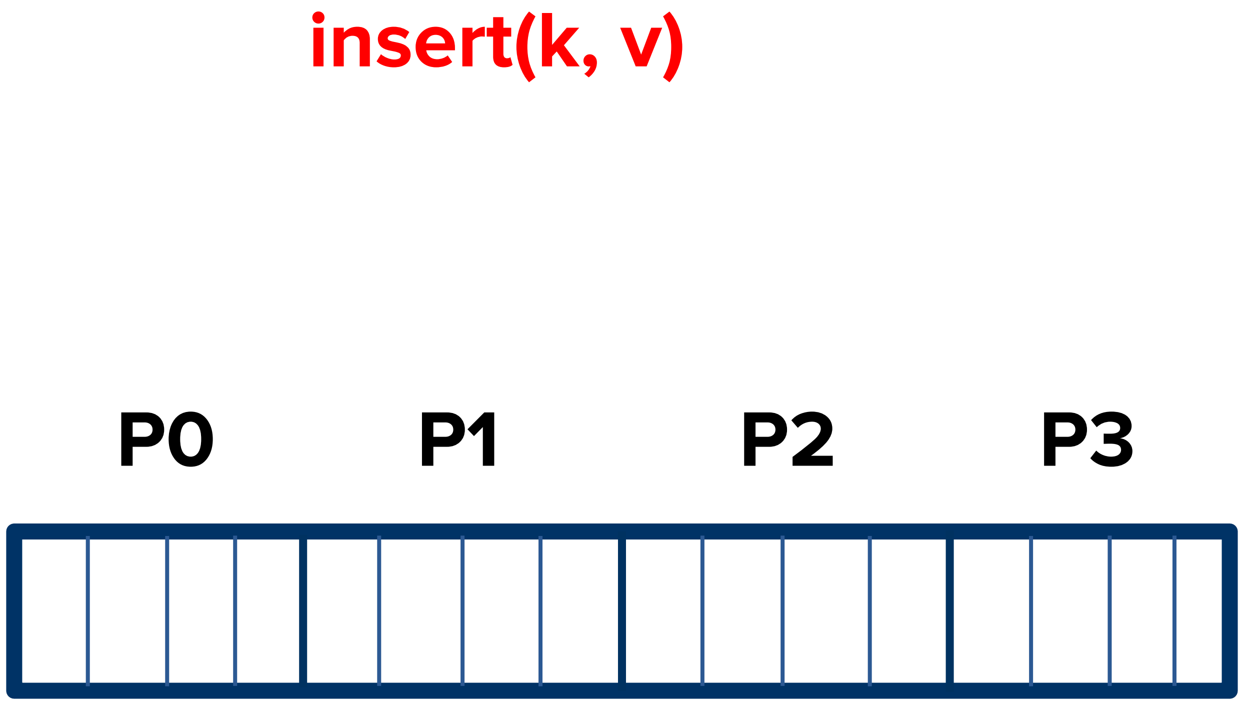
# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively



# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively

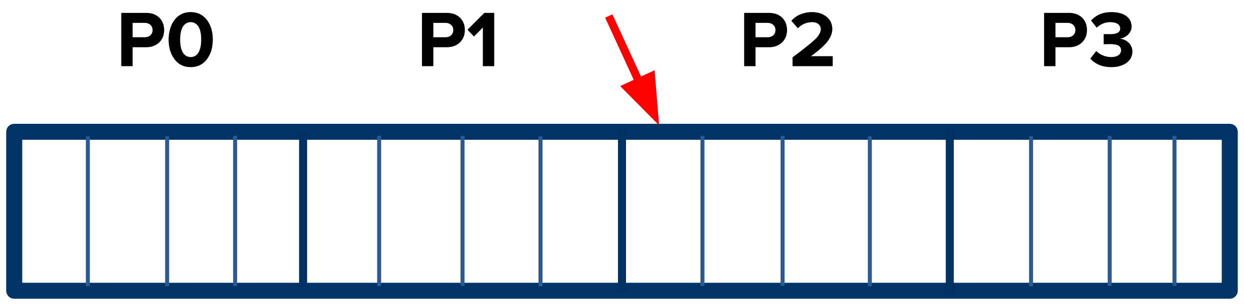


# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively

insert(k, v)

1) Calculate location

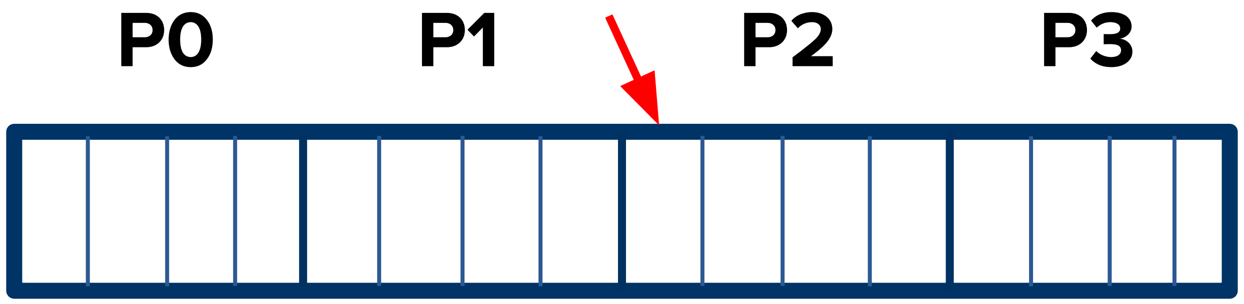


# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively

insert(k, v)

1) Calculate location
2) Request bucket ($A_{FAO}$)



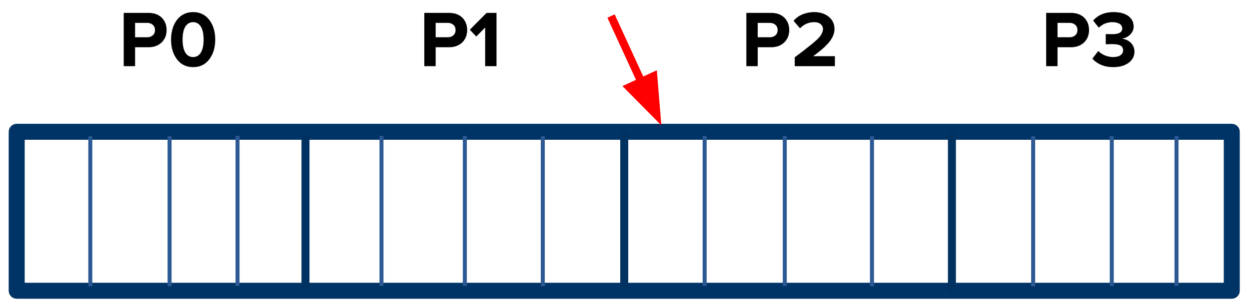

[1] ICPP'19, Brock, et. al

# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively

**insert(k, v)**

1) **Calculate location**
2) **Request bucket ($A_{FAO}$)**
3) **Insert item (W)**

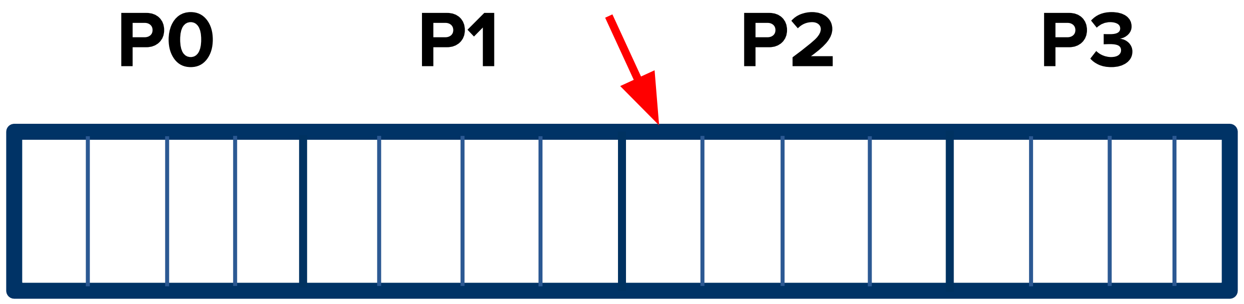


# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively

**insert(k, v)**

1) **Calculate location**
2) **Request bucket ($A_{FAO}$)**
3) **Insert item (W)**
4) **Mark bucket ready ($A_O$)**



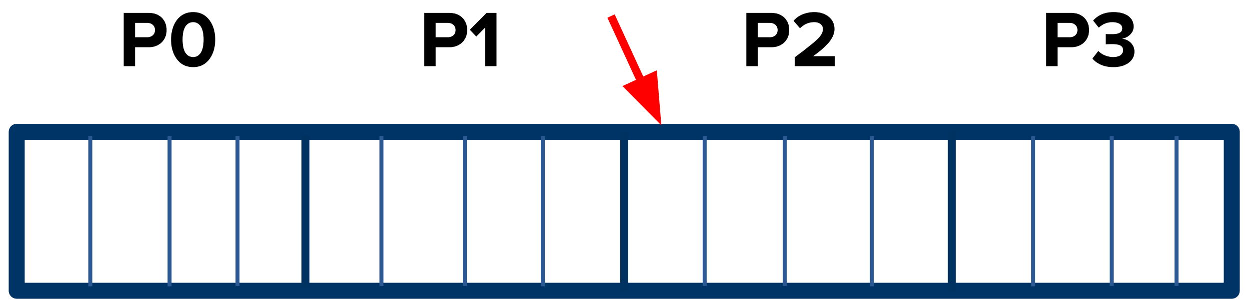
# Distributed Hash Table

- Open addressing -- hash table buckets are split among procs
- To manipulate a bucket, **directly read/write using RDMA**.
- **Resizing** must be done collectively

**Best Case Cost: $A_{FAO} + W\ (+\ A_{O})$**

**insert(k, v)**

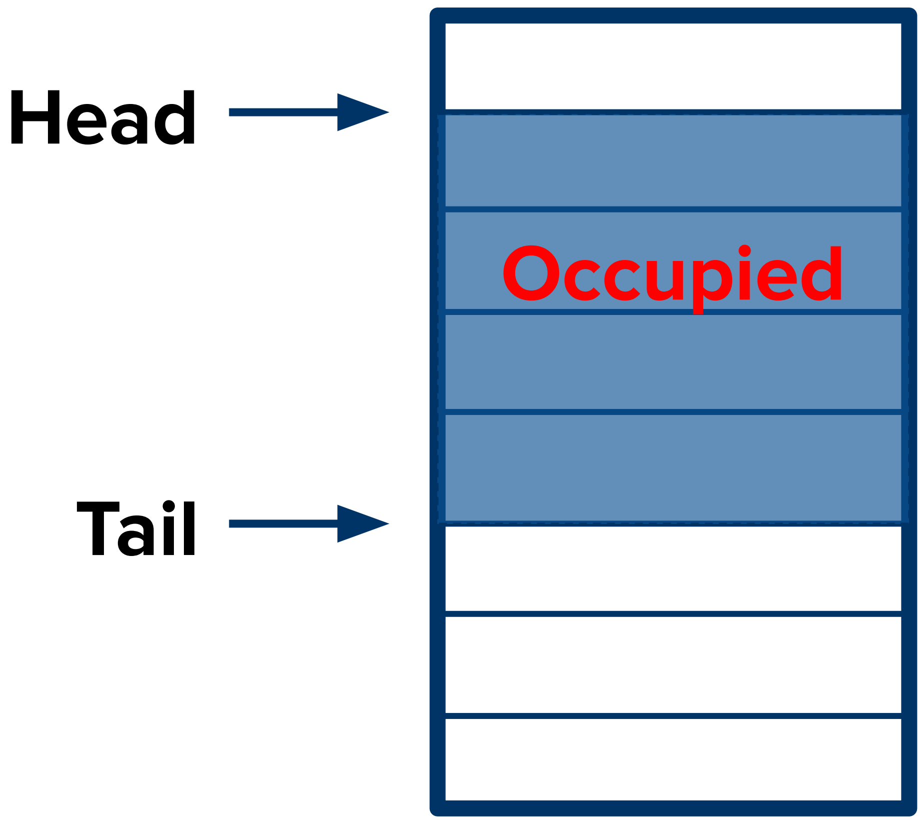
1) Calculate location
2) Request bucket ($A_{FAO}$)
3) Insert item (W)
4) Mark bucket ready ($A_{O}$)




[1] ICPP'19, Brock, et. al

# Remote Queues

- A queue lives on **one process**, but is **globally visible**.
- Atomics control the **head and tail** of the queue.
- **Remote read/write** are used to manipulate queue data

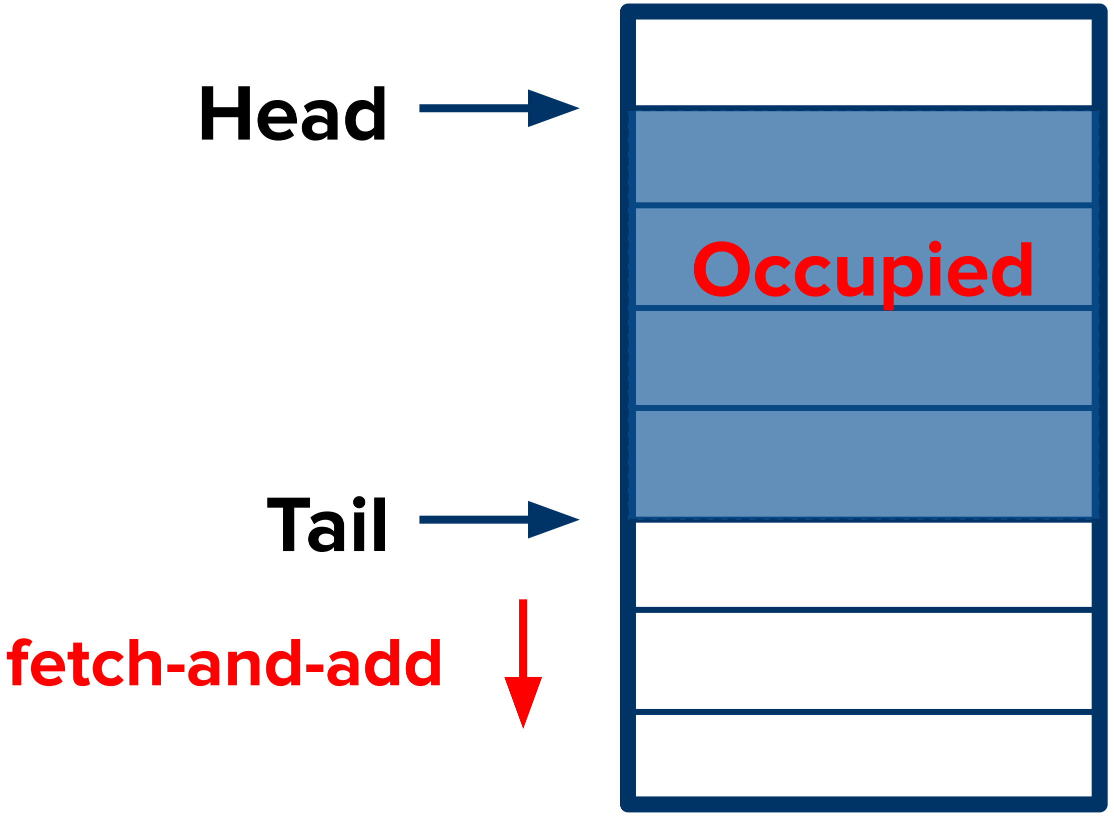



[1] ICPP'19, Brock, et. al

Berkeley
UNIVERSITY OF CALIFORNIA

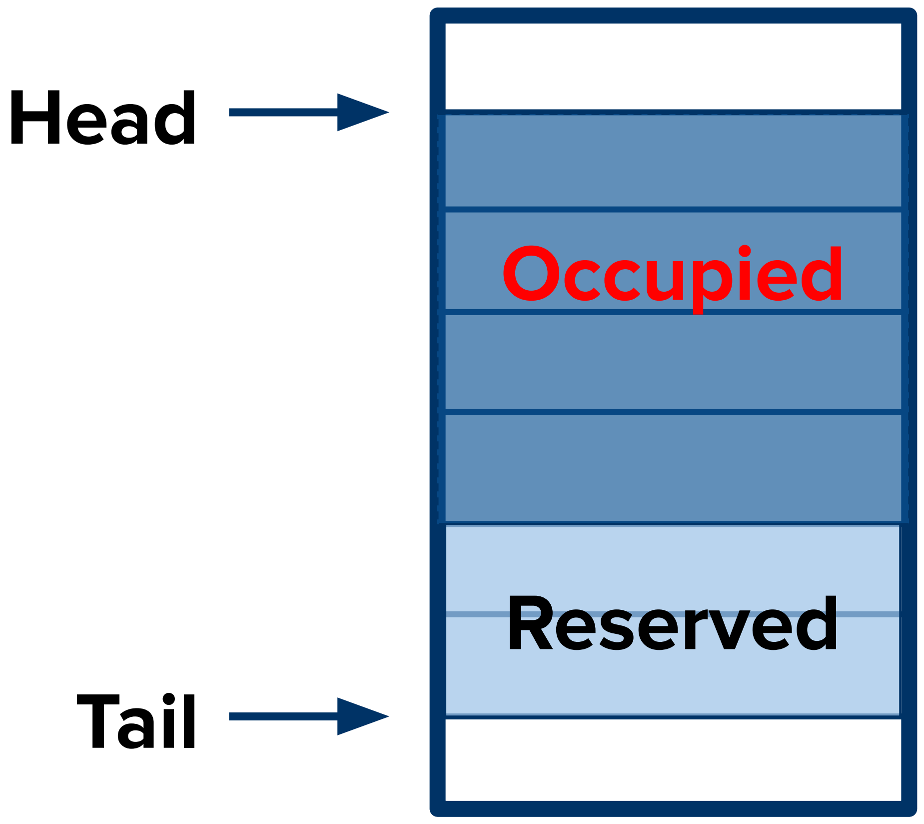
# Remote Queues

- A queue lives on **one process**, but is **globally visible**.
- Atomics control the **head and tail** of the queue.
- **Remote read/write** are used to manipulate queue data



# Remote Queues

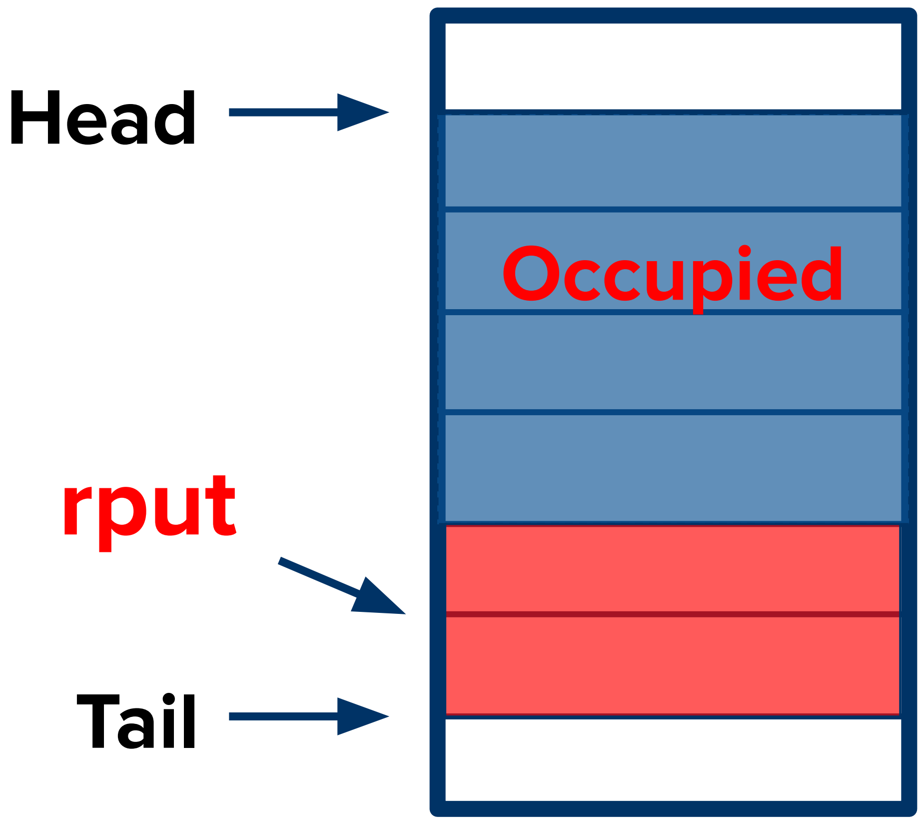
- A queue lives on **one process**, but is **globally visible**.
- Atomics control the **head and tail** of the queue.
- **Remote read/write** are used to manipulate queue data




[1] ICPP'19, Brock, et. al

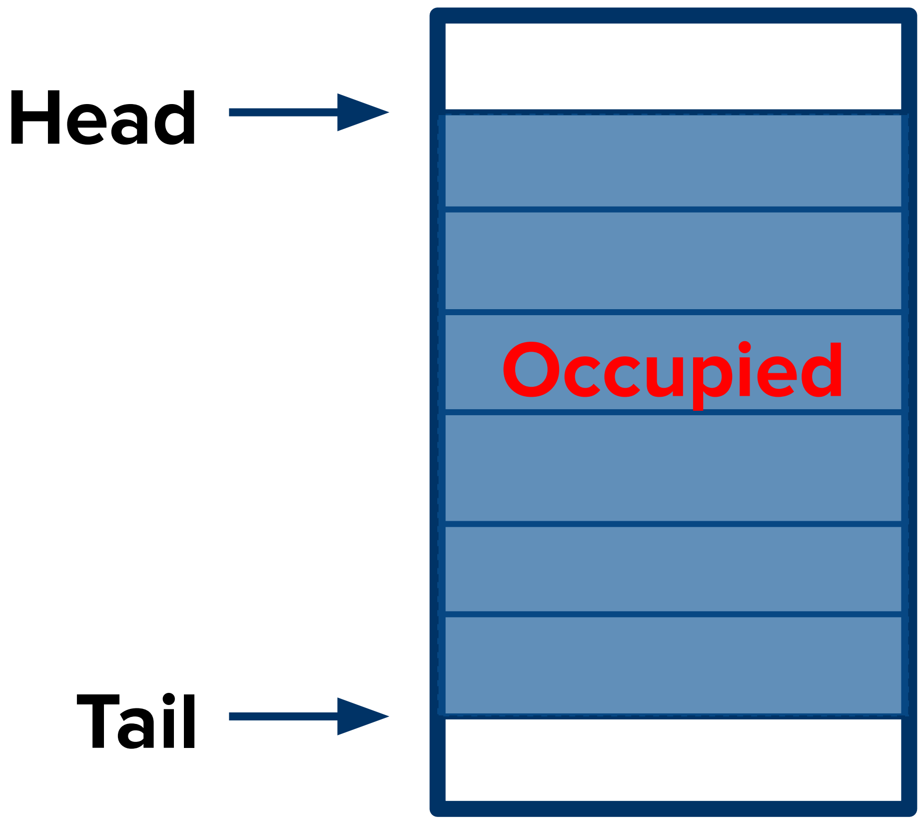
# Remote Queues

- A queue lives on **one process**, but is **globally visible**.
- Atomics control the **head and tail** of the queue.
- **Remote read/write** are used to manipulate queue data



# Remote Queues

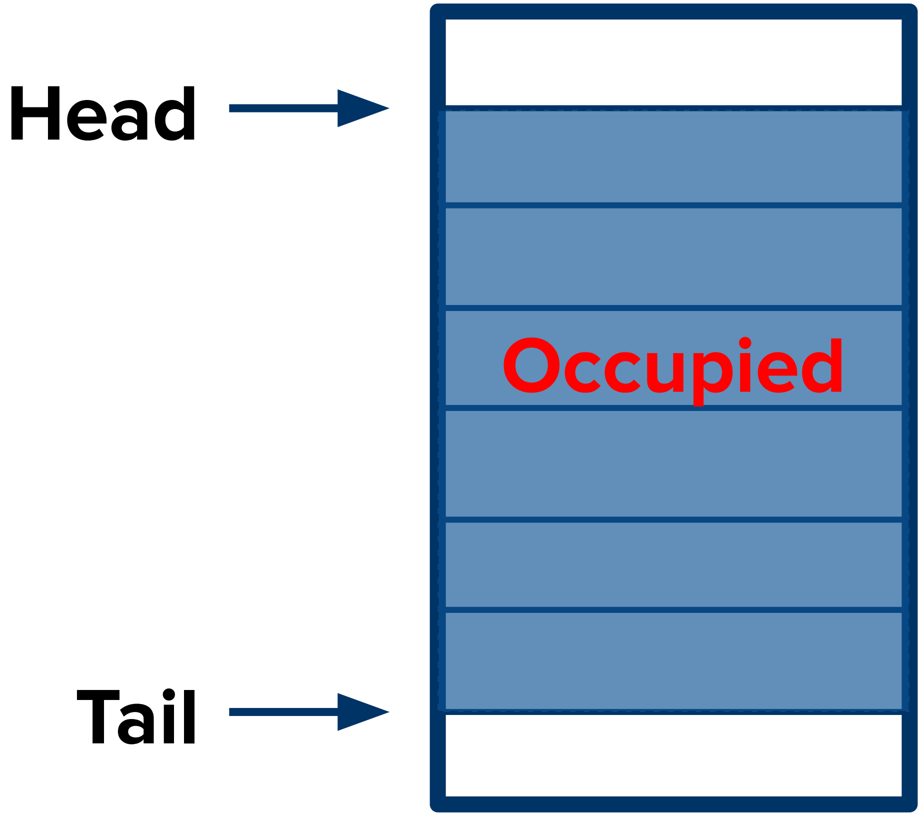
- A queue lives on **one process**, but is **globally visible**.
- Atomics control the **head and tail** of the queue.
- **Remote read/write** are used to manipulate queue data



# Remote Queues

- A queue lives on **one process**, but is **globally visible**.
- Atomics control the **head and tail** of the queue.
- **Remote read/write** are used to manipulate queue data

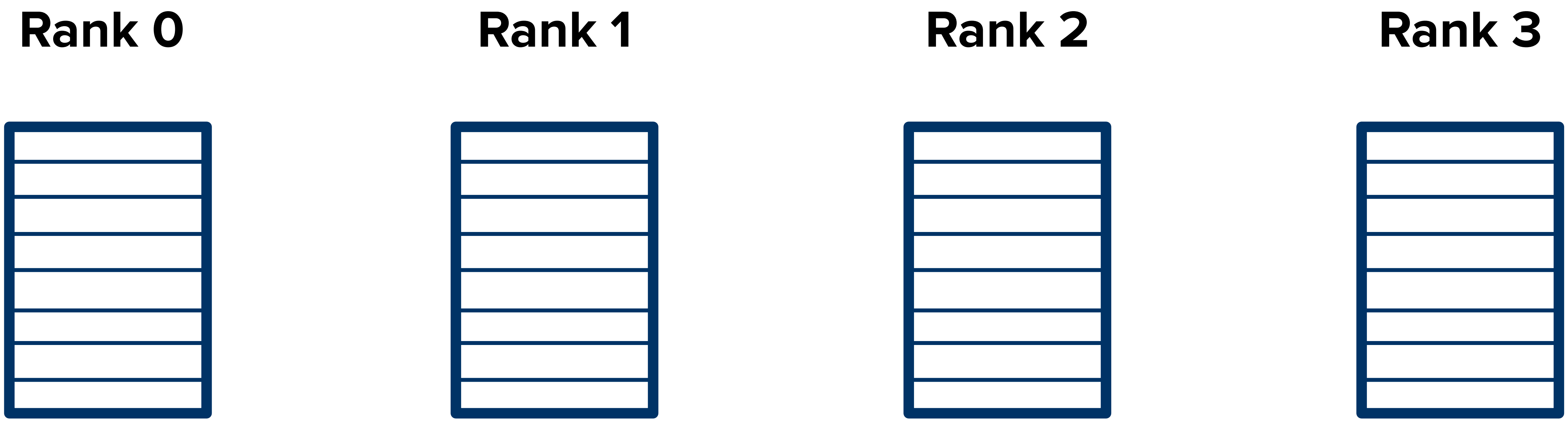
**In fully atomic queue impl., may require additional AMO**



# BCL Circular Queue

Queues allow an easy **asynchronous all-to-all**

**Common pattern**: bucket sort, particle binning, **hash table insert**

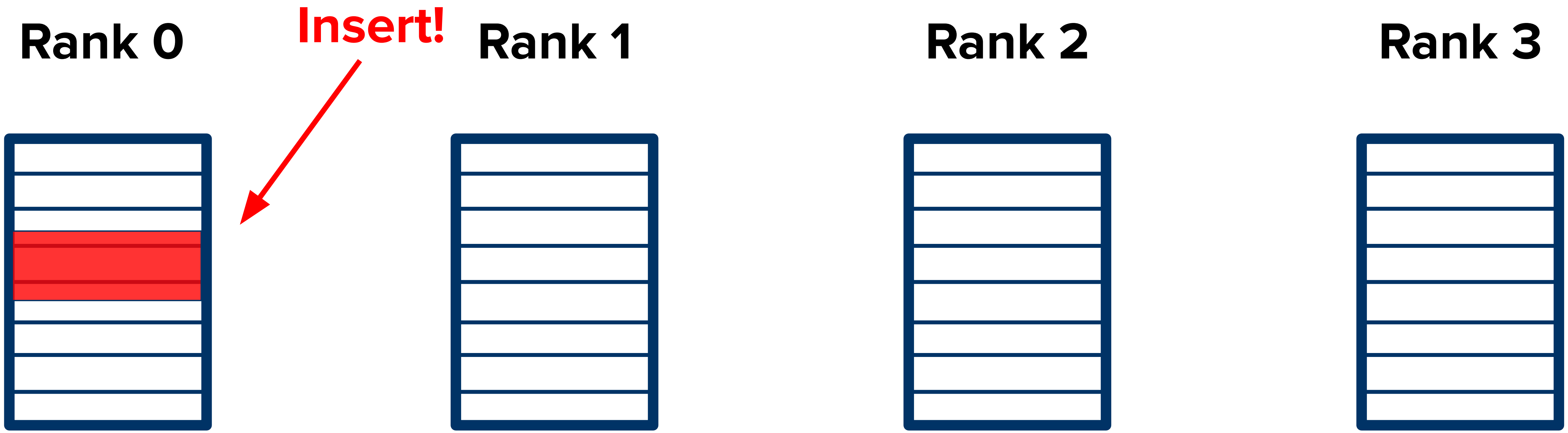



[1] ICPP'19, Brock, et. al

# BCL Circular Queue

Queues allow an easy **asynchronous all-to-all**

**Common pattern**: bucket sort, particle binning, **hash table insert**

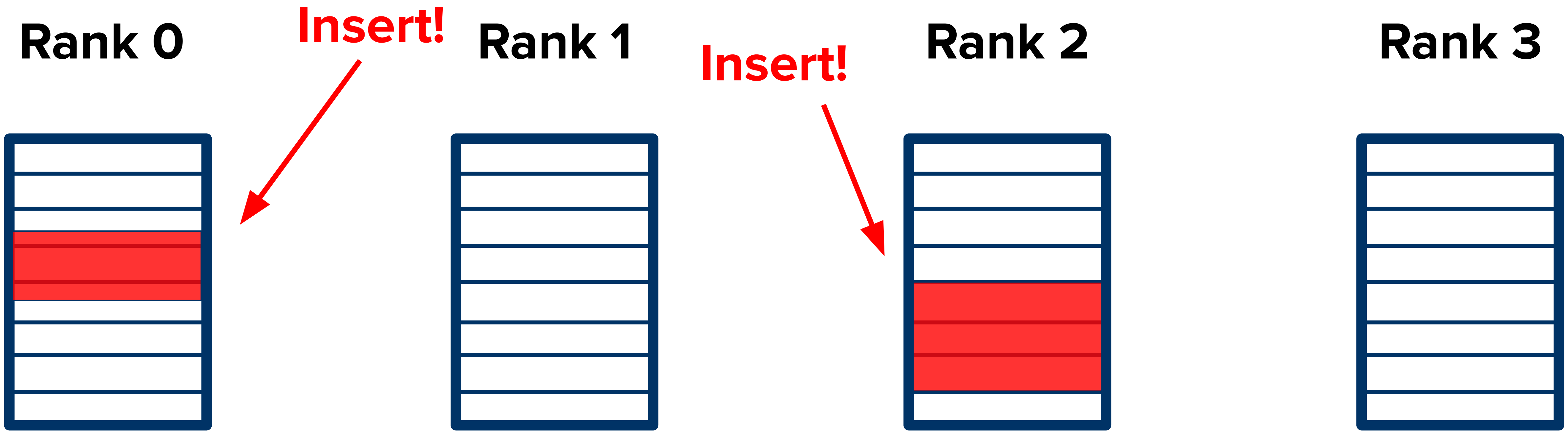



[1] ICPP’19, Brock, et. al

# BCL Circular Queue

Queues allow an easy **asynchronous all-to-all**

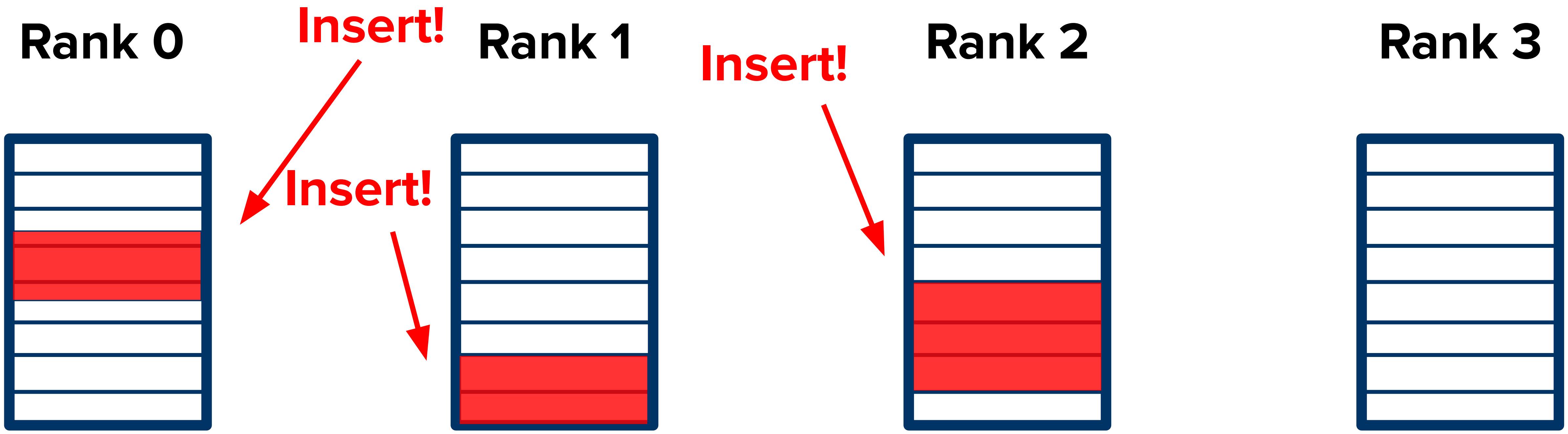
**Common pattern**: bucket sort, particle binning, **hash table insert**



# BCL Circular Queue

Queues allow an easy **asynchronous all-to-all**

**Common pattern**: bucket sort, particle binning, **hash table insert**



# Buffered Hash Table Insertion

Managed transparently in **Hash Map Buffer data structure**

Create a **Hash Map Buffer** on top of a **hash table**

**Same interface** for insert and find operations

**Flush** collectively flushes buffered inserts to hash table

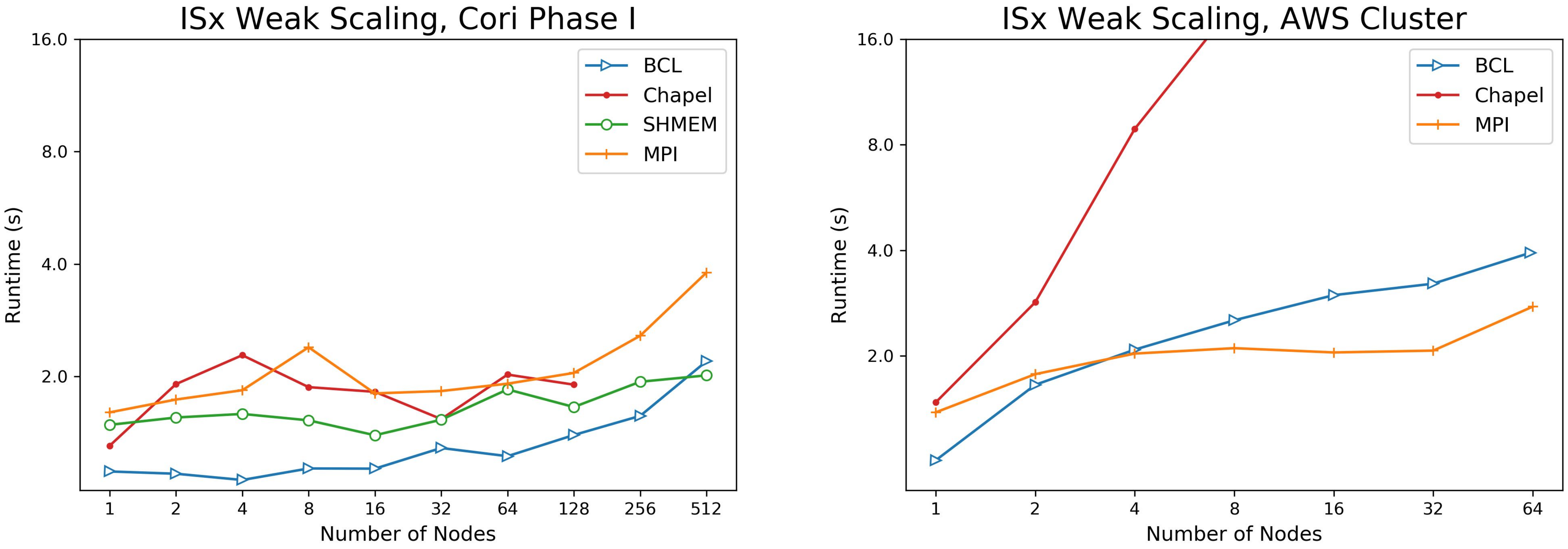
[1] ICPP'19, Brock, et. al

Berkeley
UNIVERSITY OF CALIFORNIA

# Bucket Sort Benchmark

**Insert into queues** to redistribute values

Asynchronous all-to-all allows for **more overlap**.



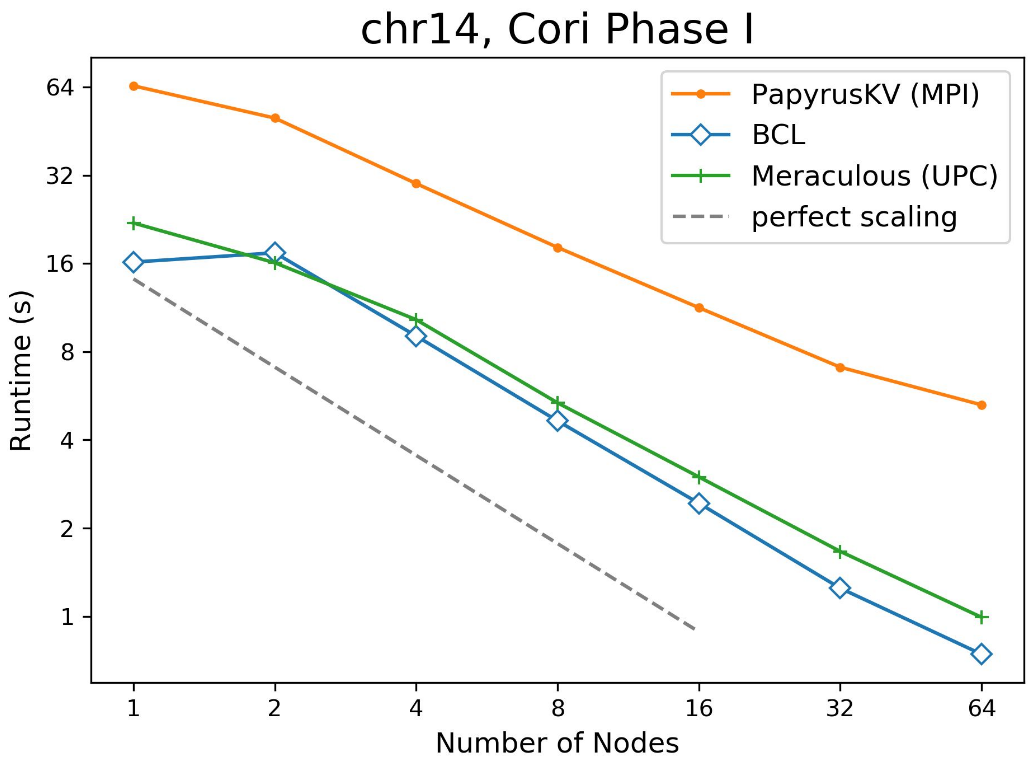

[1] ICPP'19, Brock, et. al

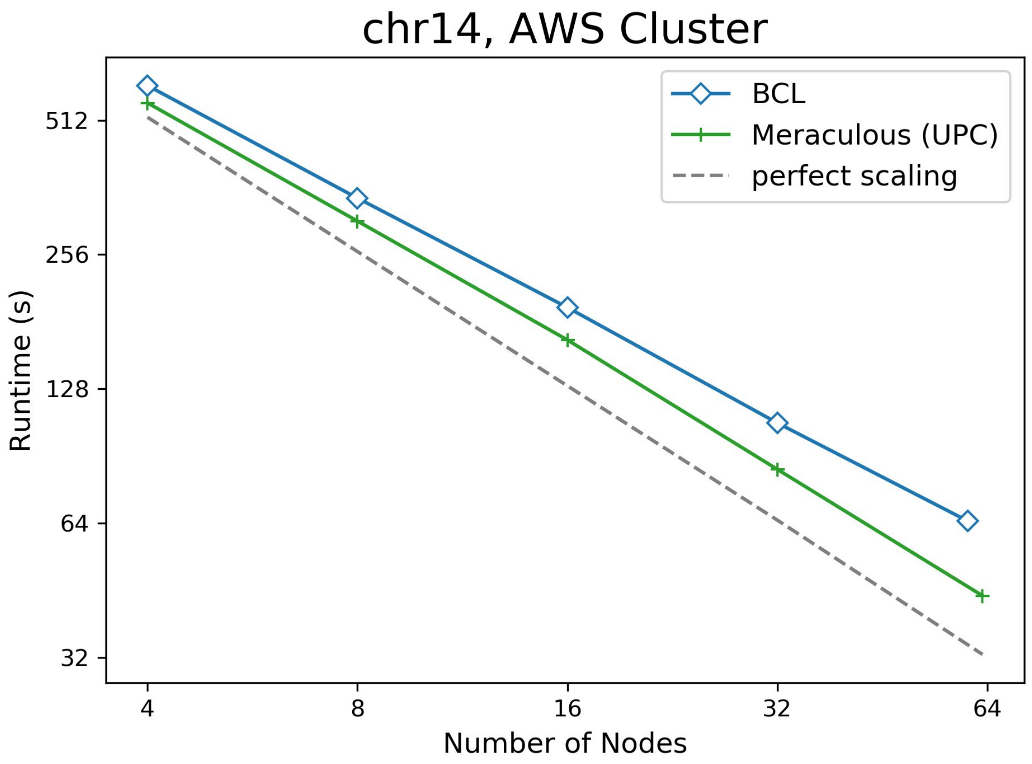
Berkeley
UNIVERSITY OF CALIFORNIA

# Contig Generation Benchmark

A **bulk insertion** phase followed by a **traversal** phase

**Aggregation** and **low-latency find** help performance.

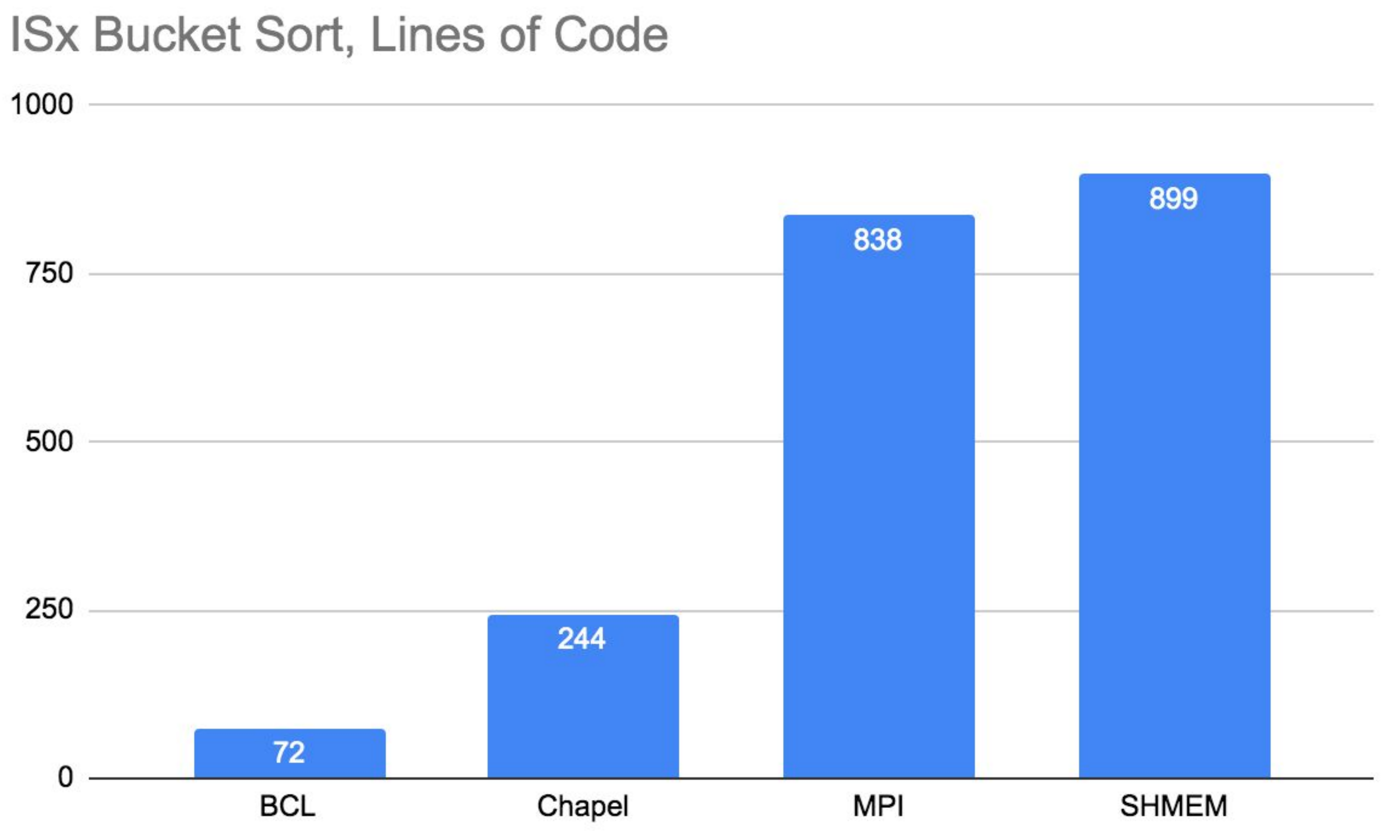


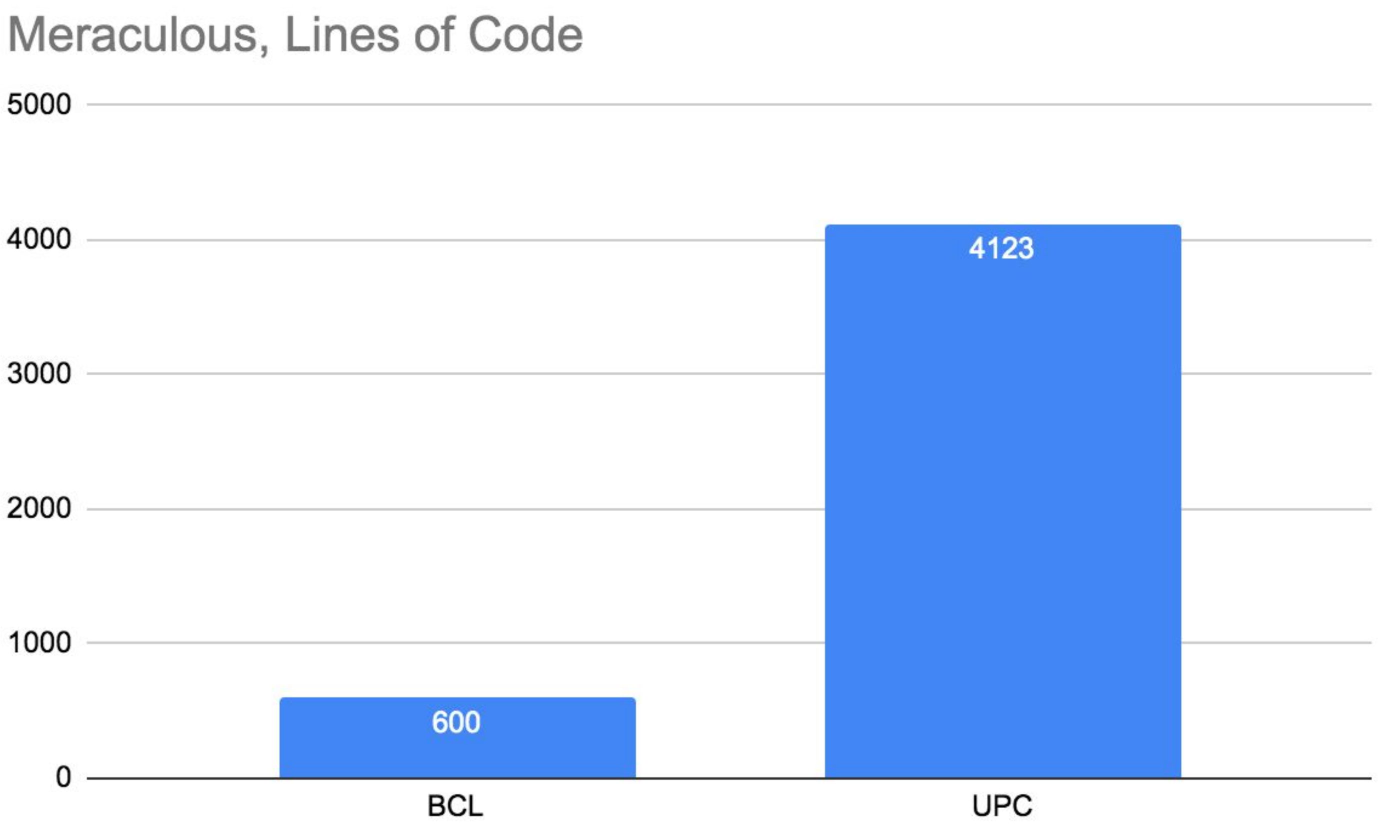





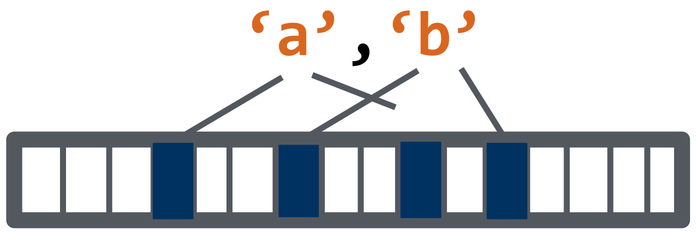

Berkeley
UNIVERSITY OF CALIFORNIA

# Comparison: Lines of Code

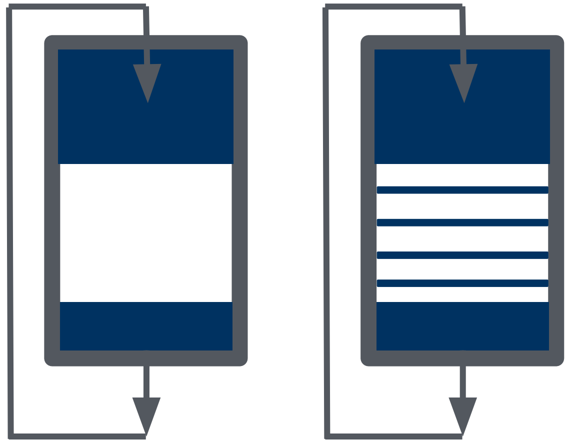


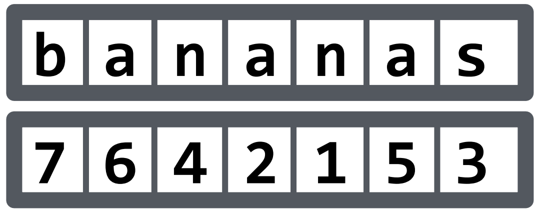
# Additional BCL Data Structures



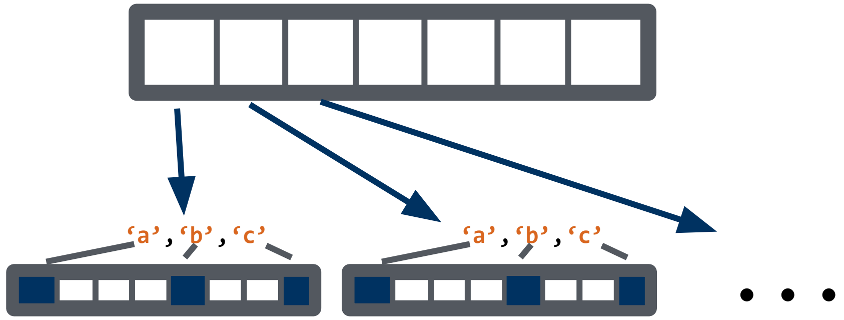

**Bloom filters**



**Queues**




**Suffix arrays**




**Dynamically-sized hash tables**

# Future Areas of Work

- Extending to work across **distributed GPUs**

- Developing **systems for aggregation**

- **Co-designing new RDMA instructions** for data structures

# CS 267 Project Ideas!

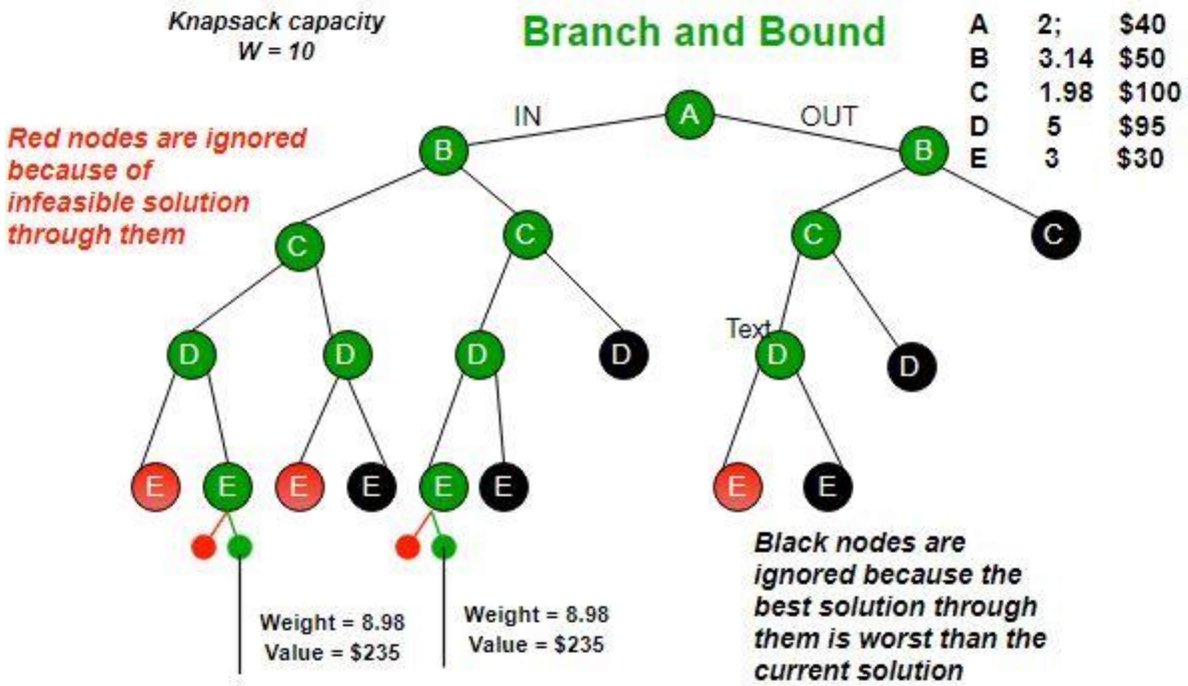
1. **Develop an application** using BCL data structures

2. **Develop a new data structure** in BCL

3. **Expand capabilities** of BCL in some way

# Application: Branch and Bound

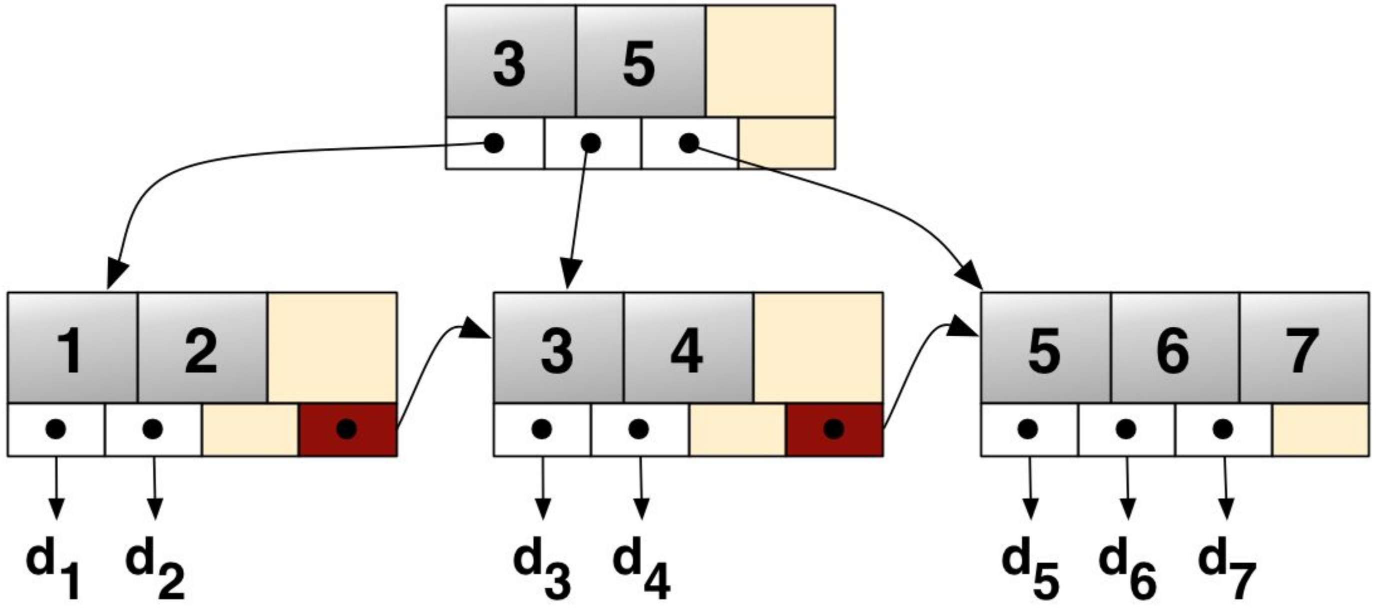
- Used in **combinatorial problems**
- User provides two functions:
  - **Branch**: given a solution, branch off a new solution
  - **Bound**: given a solution, bound how good subtree based off of it might be
- Only explore nodes which might produce a better solution
- Interesting for exploring work stealing / work pushing



# Data Structure: B-Tree

- B-trees are optimized for large block transfers, used in file systems

- Key challenge: efficiently rotate when nodes overfill

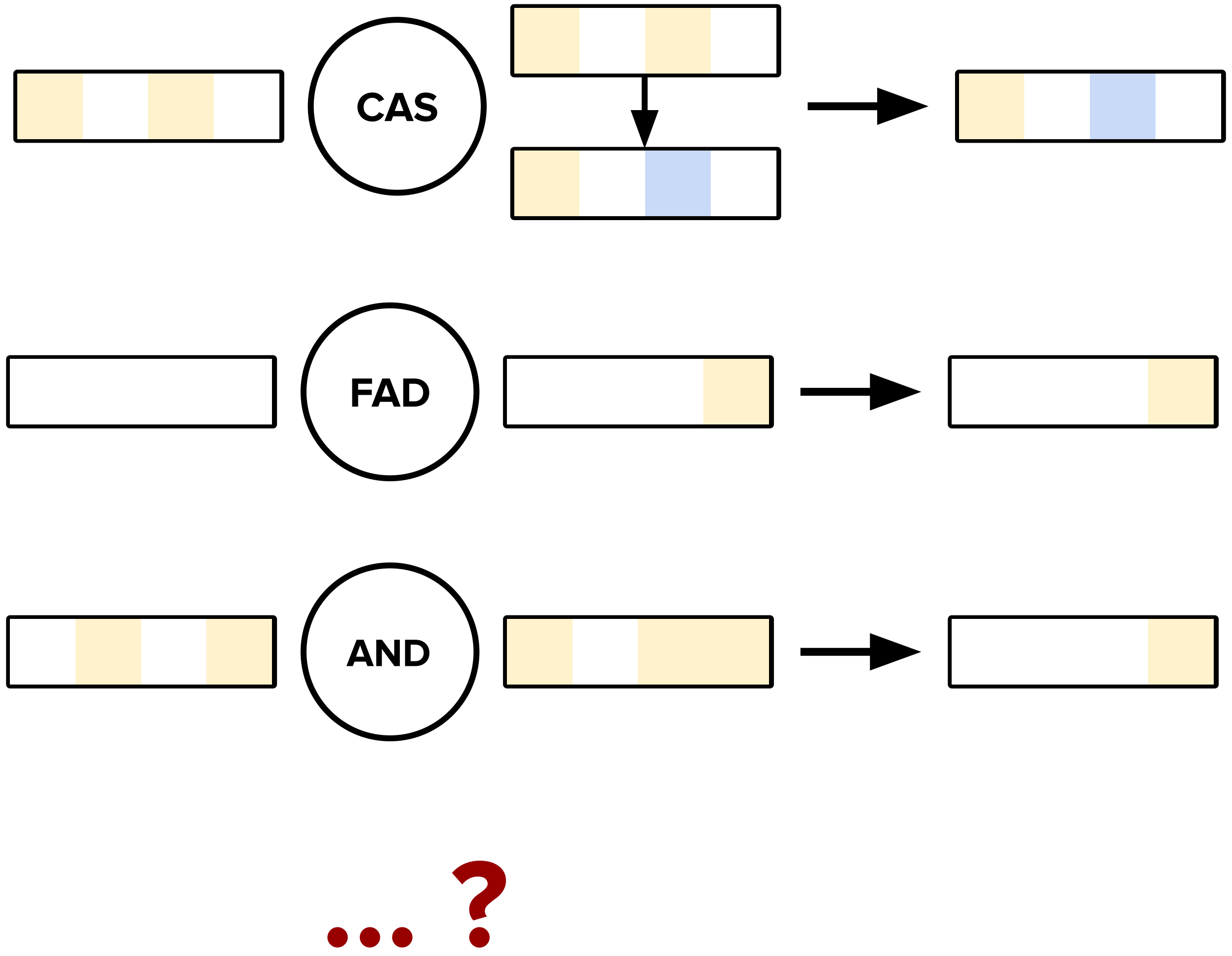


# Beyond: Modeling New RDMA Instructions

RDMA atomics currently limited to **CAS**, **FAD**, etc.

What if we could design **new RDMA operations** specifically for data structures? [1]

Would likely involve functional **simulation** / modifying data structures.




[1] M. Aguilera, K. Keeton, et al. “Designing Far Memory Data Structures: Think Outside the Box,” HotOS 2019.

# Q / A

**Benjamin Brock**

brock@cs.berkeley.edu

**Berkeley Container Library**

https://github.com/berkeley-container-library/bcl