Parallel Design Patterns

Implementation Strategies – Distributed Array, Shared Data/Queue

Reusing this material

Distributed Array – Introduction

- Distributed Array is an Implementation Strategy that comes under the Data Structures sub-group.

- Arrays often need to be partitioned between multiple UEs.
- How can this be done so that the program is both readable and efficient?

Distributed Array – Introduction

- Large arrays are fundamental data structures in scientific computing problems.
- Most systems have memory access times that vary substantially depending on which UE is accessing a particular array element.
  - even if that system supports a global address space
  - the challenge is to ensure that data elements are "*nearby*" at the right times during the computation
- For distributed systems, must explicitly distribute data.
- For NUMA systems, no need to split the data, but it's still desirable to have the right memory "*nearby*".

Distributed Array – Forces

- Load Balance
- Effective Memory Management
  - make good use of the cache
- Clarity of Solution
  - aim to have a clear mapping between local and global arrays

Distributed Array – Solution

- The "*solution*" is the mapping between local and global arrays.

$$\lfloor(\cdots) \equiv floor(\cdots)$$
$$\lceil(\cdots) \equiv ceiling(\cdots)$$

- Mapping an $M \times N$ matrix to P UEs...
  - 1D block: element $a_{i,j}$ is assigned to p_k where $k = \lfloor(j/\lceil(M/P))$
  - 1D block-cyclic $k = j \,\%\, P$

- Mapping an $M \times N$ matrix to $P \times Q$ UEs...
  - 2D block: element $a_{i,j}$ is assigned to $p_{k,l}$ *where* $k = \lfloor(i/\lceil(N/P))$
    $l = \lfloor(j/\lceil(M/Q))$
  - 2D block-cyclic

$$k = \lfloor(i/\lceil(N/P)) \,\%\, P$$
$$l = \lfloor(j/\lceil(M/Q)) \,\%\, Q$$

An 8 × 8 Array

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

1D Block with P = 4

$a_{i,j}$ assigned to p_k

$$k = \lfloor (j / \lceil (M/P))$$

$$j = [0..7]$$
$$M = 8$$

P_0		P_1		P_2		P_3	
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

1D Block-cyclic with P = 4

$a_{i,j}$ assigned to p_k

$$k = j \% P$$

$$j = [0..7]$$

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

2D Block with $P \times Q = 2 \times 2$

$a_{i,j}$ assigned to $p_{k,l}$

$$k = \lfloor (i / \lceil (N/P))$$

$$l = \lfloor (j / \lceil (M/Q))$$

$$i, j = [0..7]$$
$$M = N = 8$$

$P_{0,0}$	$P_{0,1}$
$P_{1,0}$	$P_{1,1}$

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

2D Block-cyclic with $P \times Q = 2 \times 2$

$a_{i,j}$ assigned to $p_{k,l}$

$$k = \lfloor (i / \lceil (N/PQ)) \,\%\, P$$

$$l = \lfloor (j / \lceil (M/PQ)) \,\%\, Q$$

$$i, j = [0..7]$$

$$M = N = 8$$

$P_{0,0}$	$P_{0,1}$
$P_{1,0}$	$P_{1,1}$

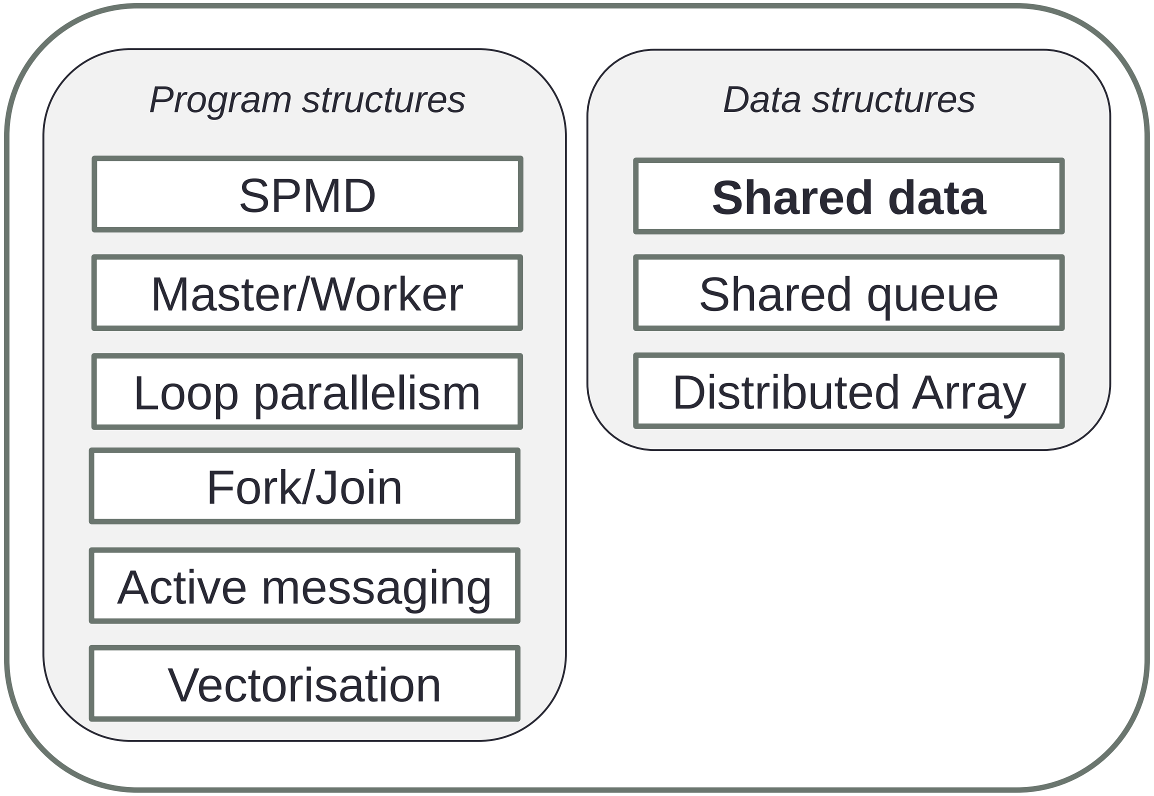
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

Distributed Array – Comments

- Complex mappings between co-ordinate systems are often best-expressed by use of macros.
  - aids readability and harder to make mistakes when writing
  - no performance hit
- ScaLAPACK is an example of a library that is based around the 2D block-cyclic array distribution
  - good for load balance and memory locality
    - http://netlib.org/scalapack/slug/node75.html
- Distributed Array is often used with the Geometric Decomposition and SPMD patterns.

Shared Data – Introduction

- Shared Data is an Implementation Strategy (or Supporting Structure).



Shared Data – Introduction

- How does one explicitly manage shared data for a set of parallel tasks?

- Some parallel algorithm patterns handle shared data by extracting it from the task.
  - Replication & Reduction with Task Parallelism
  - Halo-swapping with Geometric Decomposition

- The Shared Data pattern is required when data cannot be extracted from the tasks.
  - such as when dependencies are neither removable or separable

Shared Data – Introduction

- Some common attributes for problems that need the Shared Data pattern are...
  - at least one data structure is accessed by multiple tasks in the course of the program’s execution
  - at least one task modifies the shared data structure, and
  - the tasks potentially need to use the modified value during the concurrent computation

Shared Data – Forces

- The results of the computation must be correct for any ordering of the tasks that could occur during the computation.
- Explicitly managing shared data can incur parallel overhead, which must be kept small if the program is to run efficiently.
- Techniques for managing shared data can limit the number of tasks that can run concurrently, impacting scalability.
- If the constructs used to manage shared data are not easy to understand, the program will be harder to maintain.

Shared Data – Solution

- Ensure this pattern is needed.
  - is there an approach that matches one of the other algorithm strategy patterns without the need for shared data?
- Make use of Abstract Data Types (ADTs).
- Implement appropriate concurrency-control protocol.
  - One-at-a-time execution
  - Noninterfering sets of operations
  - Readers/Writers
  - Reducing the size of the critical section
  - Nested locks
  - Application-specific semantic relaxation

Shared Data – Solution continued

- Review other considerations.
  - Memory synchronisation
  - Task scheduling

Using an Abstract Data Type

- Consider the shared data type as an ADT with a fixed set of (possibly complex) operations on the data.
  - `put`, `get`, `remove`, `isEmpty`, `getSize`
- Each task will typically perform a sequence of these operations, along with operations on other (non-shared) data.
- Operations should have always leave the data in a consistent and meaningful state.
- Implementation of individual operations should be such that results of lower-level actions should not be visible to other tasks/Ues.

Concurrency Control Protocols

- We need to ensure that the operations provide the same results as if they were executed in serial.
- One-at-a-time execution...
  - the simplest approach, ensure operations indeed do execute in serial
  - use a Critical Section
    - provided directly by language, or indirectly through mutex locks, synchronised blocks, or semaphores
  - in a message-passing environment, assign the data structure to one UE and ensure all access to the data is through this UE
    - usually straightforward to implement, but often overly conservative resulting in bottlenecks

Concurrency Control Protocols

- Create non-interfering sets of operations.
  - analyze the interference between operations
    - operation **A** interferes with operation **B** if **A** writes a variable that **B** reads or writes.
  - maintain disjoint sets of interfering operations, where operations in different sets do not interfere
  - within each disjoint set operations execute one at a time, but operations in different sets can proceed concurrently

Concurrency Control Protocols

- Readers/Writers
  - separate operations into those that modify the data and those that are read only.
  - if **A** is a writer (both modify and read) but **B** is reader (only read) then **A** interferes with itself and **B**, but **B** interferes with nothing.
    - therefore if one task is performing **A** then no other task should be able to execute **A** or **B**; but, any number of **B**s can execute concurrently.
    - this is the basis for RW locks in `pthreads`
  - introduces some overhead, so some thought needed when implementing lock writers

Concurrency Protocols

- Reduce the size of the critical section.
  - don’t put the whole operation in a critical section
  - determine precisely the feature that causes interference
  - be careful, critical sections are easy to get wrong!
- Nested locks...
  - a hybrid of noninterfering operations and reducing the CS size
  - if you have *almost* non-interfering operations, an extra lock can be placed around just the interfering part of the operation
    - if **A** reads and writes to *y*, and **B** reads and writes to *y* then these operations interfere, so placing a lock around **A**’s *y* access should enable additional concurrency
  - increased potential for deadlock

Concurrency Protocols

- Application specific semantic relaxation
  - partially replicate shared data and don't keep all of the copies completely in sync
  - this may involve duplication of work
    - a number of tasks searching for an answer based upon the same starting conditions
  - this duplication however can be more efficient than a shared data scheme

Shared Data – Other considerations

- Memory synchronisation
  - caching and compiler optimisation can result in unexpected behaviour
    - a stale value is read from a cache or a new value is not flushed to memory
  - in OpenMP, there is a flush directive which is invoked by several other directives (such as after a `for`, `critical`, `single`, `barrier`.)
  - in Java, memory is explicitly synchronised when entering and leaving synchronised blocks, when locking and unlocking locks and for all variables marked with volatile
  - in C or FORTRAN, we have the volatile keyword too, often needed!

- Task scheduling
  - will a task be idle, waiting for access to some shared data?
  - can we assign tasks to UEs in such a way that minimises idle time?

Shared data – Summary

- First consider if you really have to use this pattern.
- Make use of Abstract Datatypes.
- Carefully consider the appropriate concurrency protocol.
  - usually a trade off between simplicity and performance
  - can I do other things (such as clever task scheduling) to minimise the impact this will have?

Shared Queue – Introduction

- How can concurrently-executing UEs safely share a queue data structure?

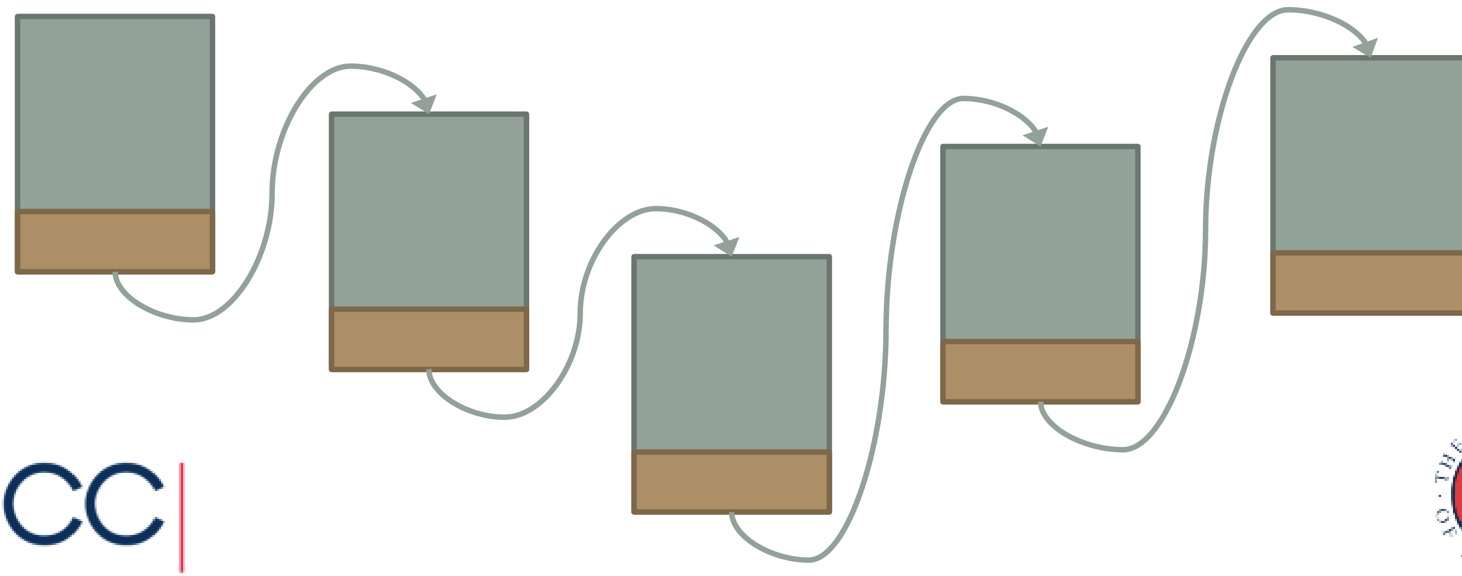
- Effective implementation of many parallel algorithms requires a queue that is to be shared among UEs.

- An example we've already talked about is the "task pool" in the Master/Worker pattern.

Shared Queue – Solution

- The queue is a FIFO data type.

- Often implemented as a linked list.



Effect of Concurrency Control Protocol

- The majority of the important forces relate to the choice of concurrency-control protocol.
  - One-at-a-time execution
  - Non-interfering sets of operations
  - Readers/Writers
  - Splitting or Shrinking the Critical Section
  - Nested Locks
  - Application specific semantic relaxation

Shared Queue – Forces

- Simple concurrency-control protocols provide greater clarity of abstraction making it easier to check correctness.
  - optimise only when clarity has been achieved

- Bloated synchronisation constructs increase the chance that UEs will remain blocked waiting to access the queue, limiting concurrency.
- A concurrency-control protocol finely tuned to the queue and how it will be used maximises the available concurrency, at the cost of more complicated and more error-prone synchronisation constructs.

Shared Queue – Solution

- Ideally the shared queue would be implemented as part of the target programming language
  - Java has an implementation available in `java.util.concurrent`
- Unfortunately, no mechanism available in common HPC languages such as MPI and OpenMP.
- Possible to implement shared Queue within message-passing paradigm.
  - queue owned by one process
  - queue access (`put` and `take`) done by messaging queue-owning process

Shared Queue – Solution

- Apply the shared data pattern.
- Define the ADT.
- Choose the concurrency protocol.

Shared Queue – Defining the ADT

- `put` (enqueue message)
- `take` (dequeue message)
- Other operations could be supported.
  - peek, takeall, clear, isEmpty

- What to do when a queue is empty?
  - block and wait for something to arrive
    - could be used in Master-Worker with poison pill approach
  - non-blocking queue
    - return null or special value

Shared Queue – Concurrency control protocol

- Implementing a shared queue can be tricky.
  - but if done well the implementation can be re-used widely

- Choice of protocols...
  - One-at-a-time execution
  - Non-interfering sets of operations
  - Readers/Writers
  - Splitting or Shrinking the Critical Section
  - Nested Locks
  - Application specific semantic relaxation

One-at-a-time (non-blocking)

```
public class SharedQueue {
  class Node { //inner class defines list of nodes
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }
  private Node head = new Node(null); //dummy node
  private Node last = head;

  public synchronized void put(Object task) {...}
  public synchronized Object take() {...}
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time (non-blocking) – put

```java
public class SharedQueue {
  class Node {...}
  private Node head = new Node(null); //dummy node
  private Node last = head;

  public synchronized void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
  }
  public synchronized Object take() {...}
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time (non-blocking) – take

```java
public class SharedQueue {
  class Node {...}
  private Node head = new Node(null); //dummy node
  private Node last = head;

  public synchronized void put(Object task) {...}
  public synchronized Object take() {
    Object task = null;
    if (!isEmpty()) {
      Node first = head.next;
      task = first.task;
      first.task = null;
      head = first;
    }
    return task;
  }
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time – OpenMP

- A simple queue of integers...

```
void put (int i) {
#pragma omp critical
...
#pragma omp end critical
}

int take() {
#pragma omp critical
...
#pragma omp end critical
}
```

One-at-a-time (block on empty) – put

```java
public class SharedQueue {
  class Node {...}
  ...

  public synchronized void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
    notifyAll();
  }
  public synchronized Object take() {...}
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time (block on empty) – take

```
public class SharedQueue {
  class Node {...}
  ...

  public synchronized void put(Object task) {...}
  public synchronized Object take() {
    Object task = null;
    while (isEmpty()) {
      try { wait(); }
      catch (InterruptedException ignore) {}
    }
    Node first = head.next;
    task = first.task;
    first.task = null;
    head = first;
    return task;
  }
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time (non-interfering ops)

```java
public class SharedQueue {
  class Node {...}
  ...

  private Object putLock = new Object();
  private Object takeLock = new Object();

  public void put(Object task) {
      synchronized(putLock) {...}
  }
  public Object take() {
    Object task = null;
    synchronized(takeLock) {...}
    return task;
  }
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time – OpenMP

- A simple queue of integers...

```
void put (int i) {
#pragma omp critical(put)
...
#pragma omp end critical(put)
}

int take() {
#pragma omp critical(take)
...
#pragma omp end critical(take)
}
```

One-at-a-time (nested locks)

```
public class SharedQueue {
  class Node {...}
  ...

  private int w;
  private Object putLock = new Object();
  private Object takeLock = new Object();

  public void put(Object task) {
      synchronized(putLock) {...}
  }
  public Object take() {
    Object task = null;
    synchronized(takeLock) {...}
    return task;
  }
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time (nested locks) – put

```java
public class SharedQueue {
  class Node {...}
  ...

  public void put(Object task) {
    synchronized(putLock) {
      assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p; last = p;
      if(w>0) putLock.notify();
    }
  }
  public synchronized Object take() {...}
  private boolean isEmpty() { return head.next == null; }
}
```

One-at-a-time (nested locks) – take

```
public Object take() {
  Object task = null;
  synchronized(takeLock) {
    while (isEmpty()) {
      try {
        synchronized(putLock) { w++; putLock.wait(); w--; }
      }
      catch (InterruptedException error) { assert false; }
    }
    Node first = head.next;
    task = first.task;
    first.task = null; head = first;
  }
  return task;
}
```

One-at-a-time (readers and writers) – put

```
public class SharedQueue {
  ...
  private Node last = head;

  Rwlock rw_lock = new Rwlock();

  public void put(Object task) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    rw_lock.writeLock();
    last.next = p; last = p;
    rw_lock.release();
  }
  ...
}
```

One-at-a-time (readers and writers) – viewLast

```
public class SharedQueue {
  ...
  private Node last = head;

  Rwlock rw_lock = new Rwlock();

  public void put(Object task) {...}
  public Object viewLast() {
    Object task = null;
    rw_lock.readLock();
    if (!isEmpty()) {
      task = last.task;
    }
    rw_lock.release();
    return task;
  }
  private boolean isEmpty() { return head.next == null; }
}
```

Distributed shared queues

- One central queue can be a bottleneck, so let's have one queue per UE and distribute the tasks across *P* queues.
  - if my local queue becomes empty then a `take` might "*steal*" an element from a neighbour's queue
  - if my local queue becomes full then a `put` might add the element to a neighbour's queue

- In other words...
  - each UE queues the tasks it receives
  - the tasks are then executed in turn
  - work stealing is permitted once a UE has completed its tasks

Shared Queue – Related Patterns

- Shared Data
  - Shared Queue pattern is an instance of Shared Data pattern
- Master/Worker
  - Shared Queue pattern is often used to represent the task queues in algorithms that use the Master/Worker pattern
- Fork/Join pattern:
  - thread-pool-based implementation of Fork/Join pattern is supported by this pattern

Shared Queue – Summary

- A shared queue encapsulates the synchronisation required inside an abstract data type.
- Examples follow an object-orientated paradigm, but you can “encapsulate” internal `put` and `take` routines.
- Different implementations can vary in performance and complexity.
- Shared queue is a key component of various other parallel patterns.