Parallel Design Patterns

Implementation Strategies – Distributed Array, Shared Data/Queue

Reusing this material

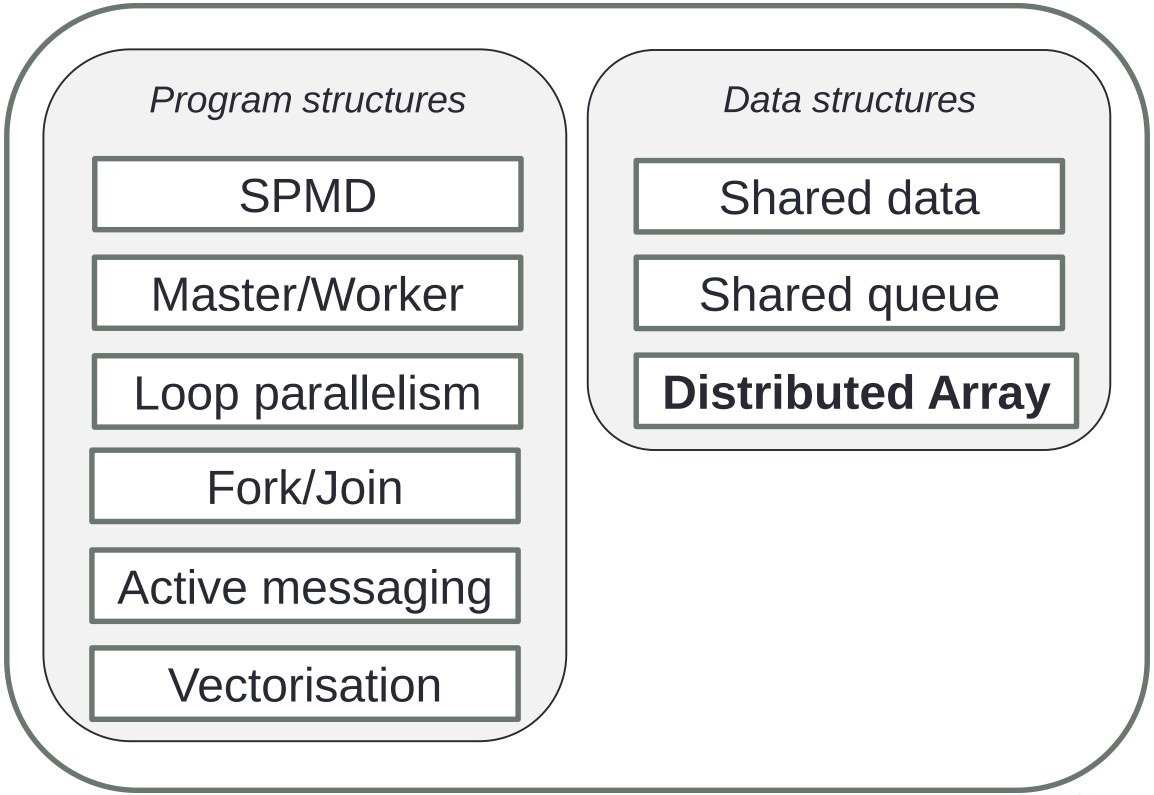
Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

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?



Program structures

- SPMD
- Master/Worker
- Loop parallelism
- Fork/Join
- Active messaging
- Vectorisation

Data structures

- Shared data
- Shared queue
- **Distributed Array**

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

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

An 8 × 8 Array

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

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

- Mapping an $M \times N$ matrix to P UEs...
  - 1D block: element $a_{i,j}$ is assigned to p_k where
  - 1D block-cyclic
- Mapping an $M \times N$ matrix to $P \times Q$ UEs...
  - 2D block: element $a_{i,j}$ is assigned to $p_{k,l}$ *where*
  - 2D block-cyclic

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

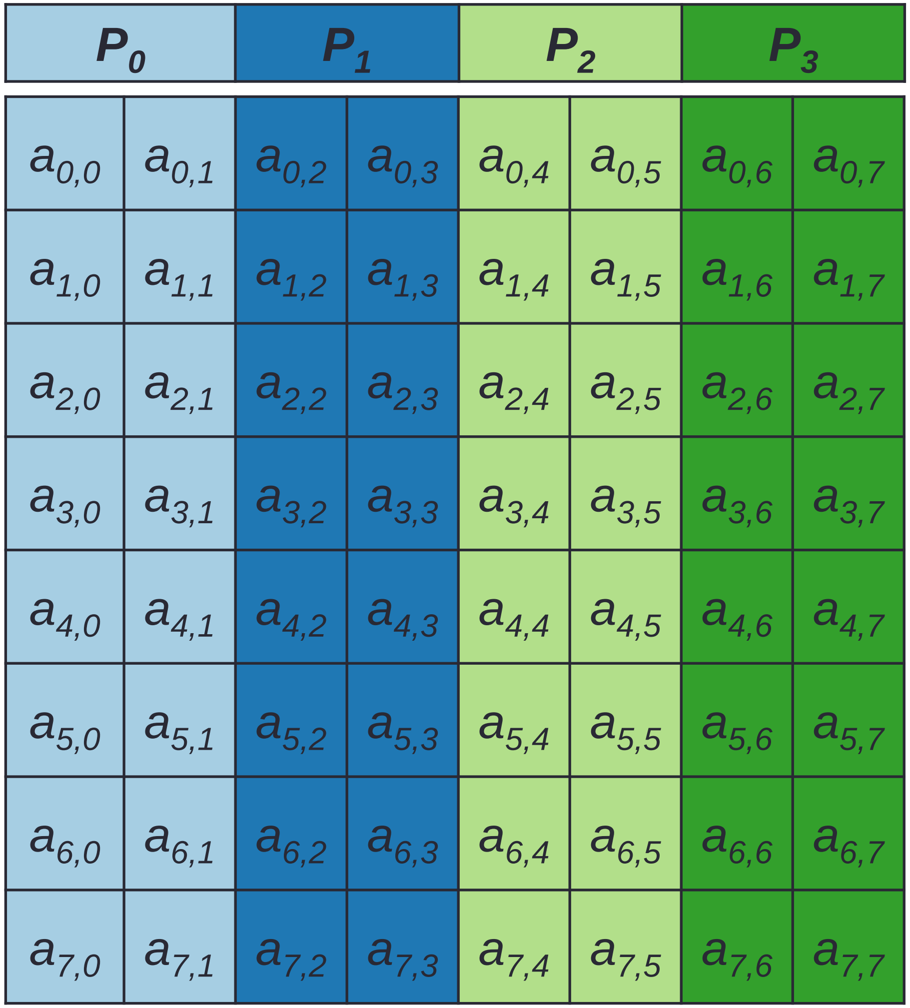
1D Block with $P = 4$

- Mapping an $M \times N$ matrix to P UEs...

$a_{i,j}$ assigned to p_k

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

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



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

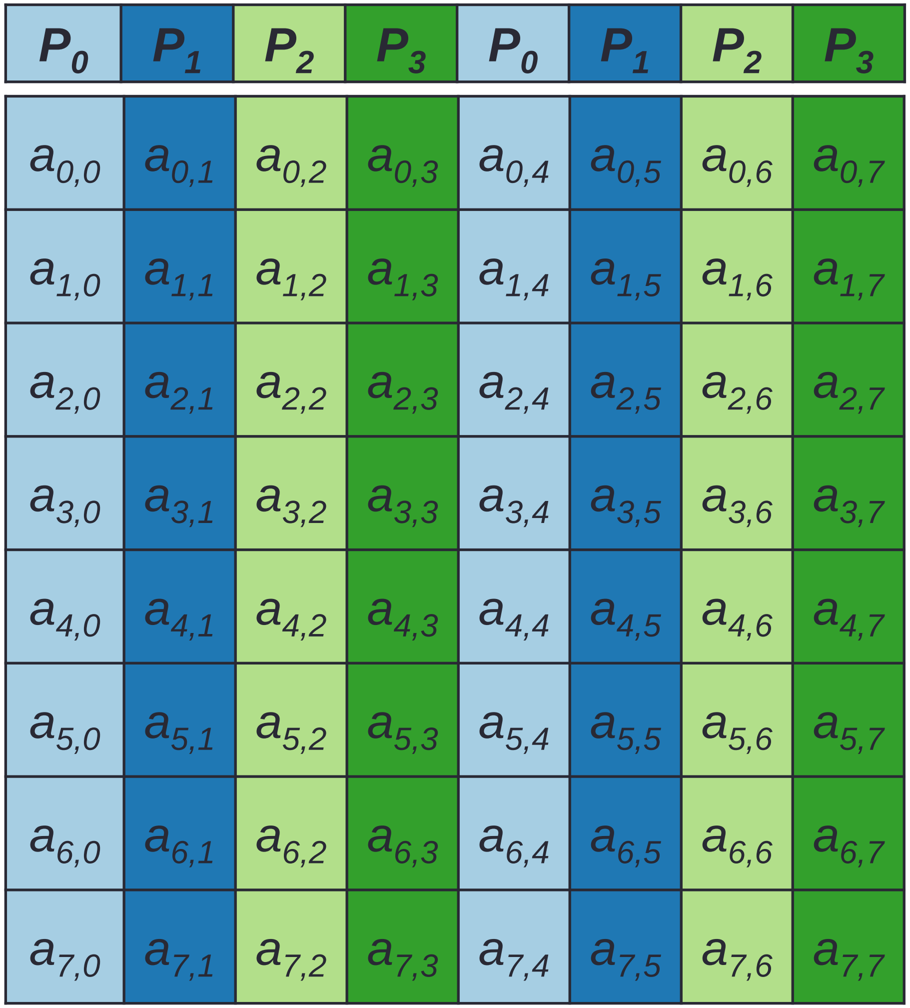
1D Block-cyclic with $P = 4$

- Mapping an $M \times N$ matrix to P UEs...

$a_{i,j}$ 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 $P \times Q = 2 \times 2$

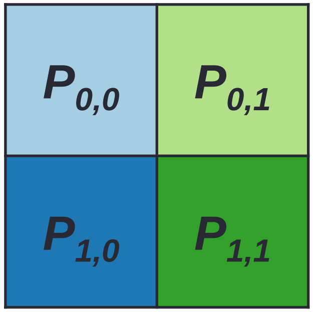
- Mapping an $M \times N$ matrix to $P \times Q$ UEs...

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

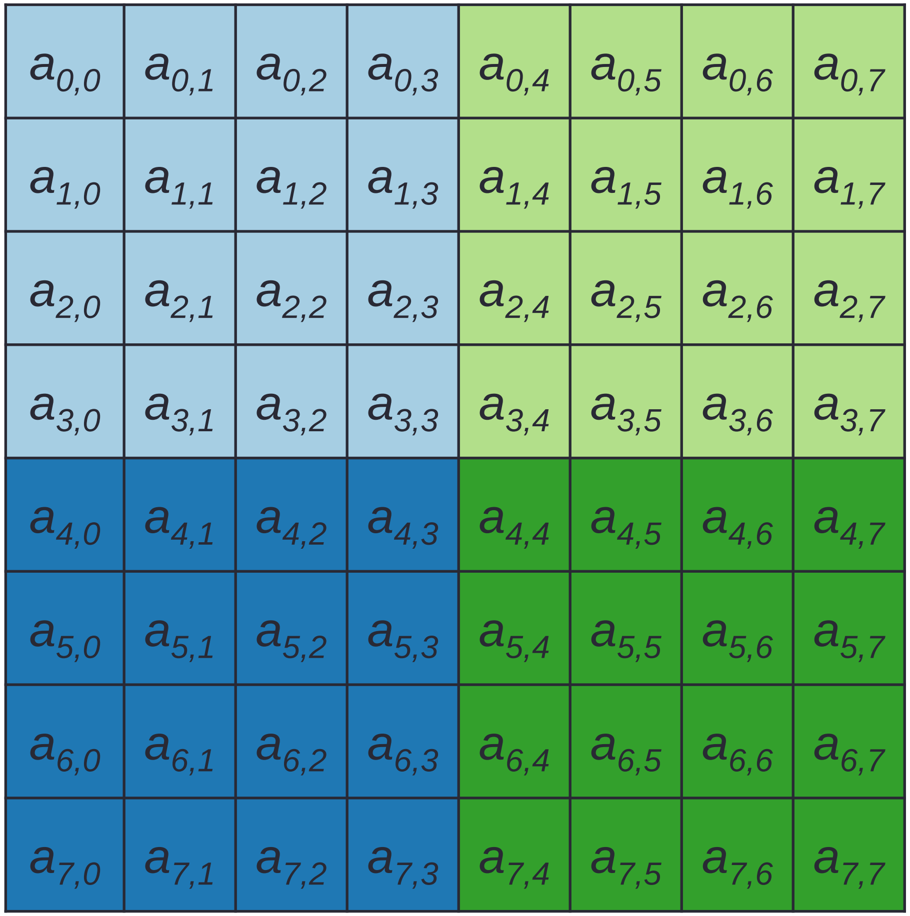
  $$k = \lfloor(i/\lceil(N/P))$$

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

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



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



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

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

- Mapping an $M \times N$ matrix to $P \times Q$ UEs...

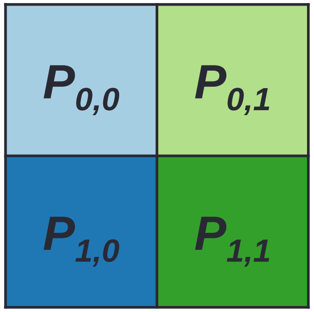
$a_{i,j}$ assigned to $p_{k,l}$

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

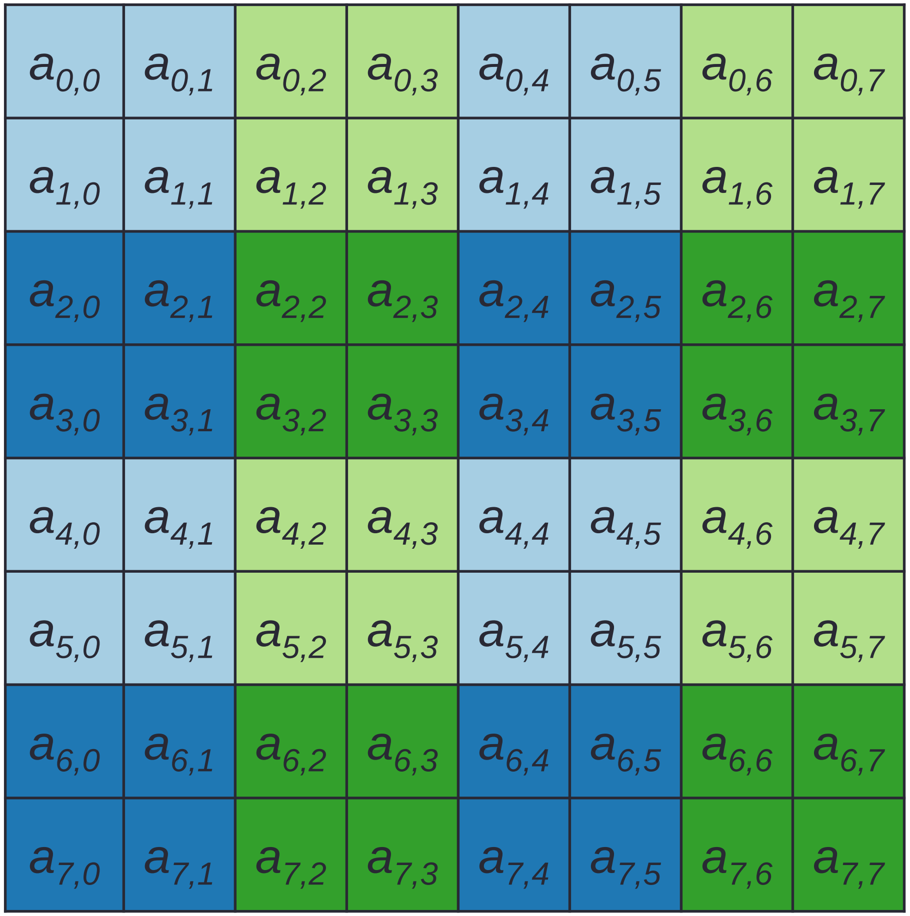
$$l = \lfloor(j/\lceil(M/PQ)) \% Q$$

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

$$M = N = 8$$



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



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

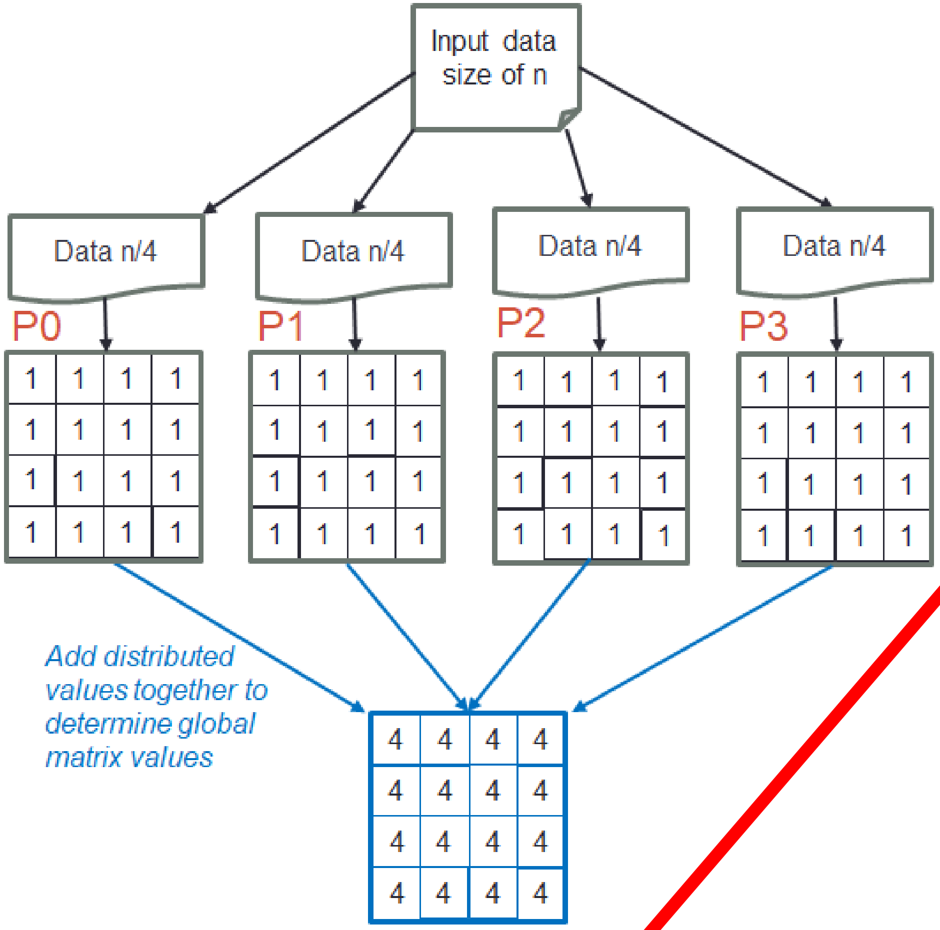
Uneven distribution

- For simplicity sake some codes don’t support an uneven distribution (e.g. 8x8 matrix over 3 UEs)

- Those that do often calculate an extra step for the number of rows held locally
  - ***if (myrank < size - local_size * P) local_size++;***

- To find my starting location determine how many of the chunks before me had an extra one and add this extra increment
- Can be a source of bugs!

Distributed Array: Example



Input data size of n

Data n/4

Data n/4

Data n/4

Data n/4

P0

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

P1

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

P2

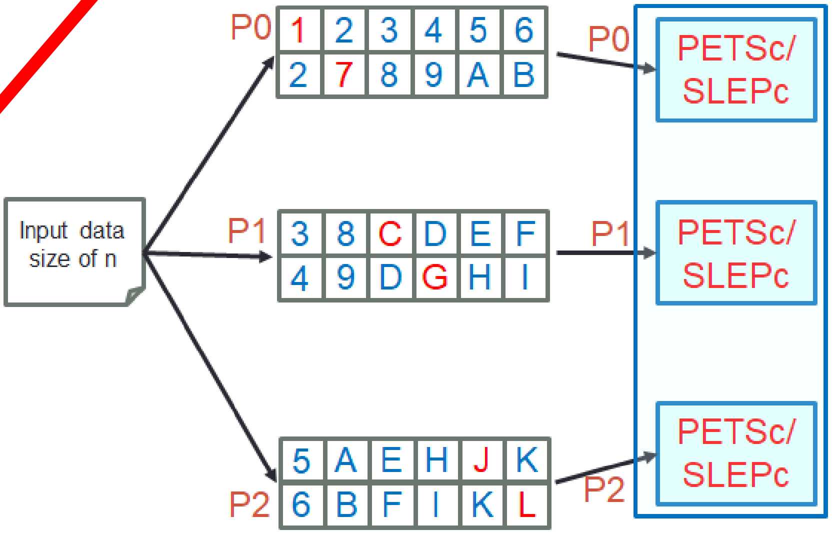
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

P3

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

Add distributed values together to determine global matrix values

4	4	4	4
4	4	4	4
4	4	4	4
4	4	4	4



Input data size of n

P0

1	2	3	4	5	6
2	7	8	9	A	B

P1

3	8	C	D	E	F
4	9	D	G	H	I

P2

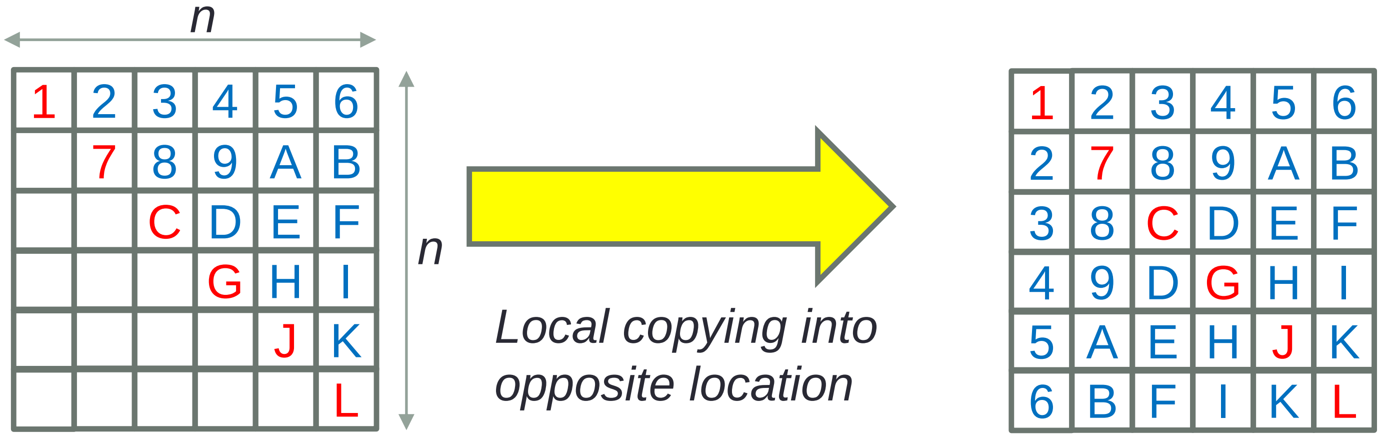
5	A	E	H	J	K
6	B	F	I	K	L

P0 PETSc/ SLEPc

P1 PETSc/ SLEPc

P2 PETSc/ SLEPc

Distributed Array: Example

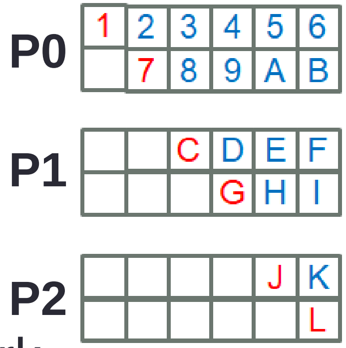


1	2	3	4	5	6
	7	8	9	A	B
		C	D	E	F
			G	H	I
				J	K
					L

Local copying into opposite location

1	2	3	4	5	6
2	7	8	9	A	B
3	8	C	D	E	F
4	9	D	G	H	I
5	A	E	H	J	K
6	B	F	I	K	L

- With entirety of matrix on each process, the symmetry is simple to deal with as only compute the diagonal and upper part, and copy upper elements into lower locations
  - When we split the matrix up could just calculate upper elements and communicate to the lower elements
    - But significant load imbalance!
  - Or could compute all elements, but duplication of work



P0

1	2	3	4	5	6
	7	8	9	A	B

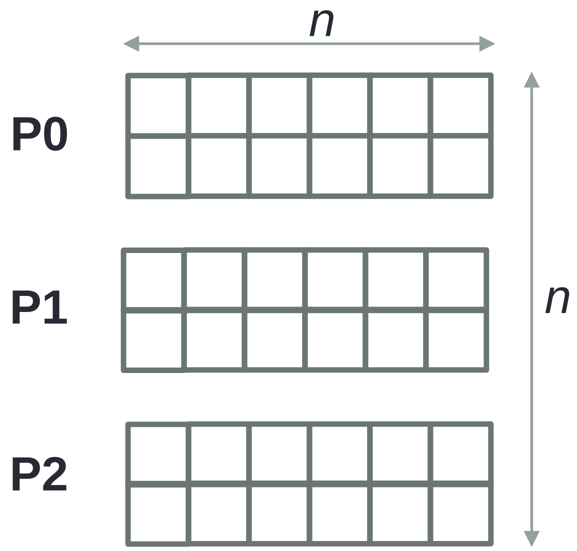
P1

		C	D	E	F
			G	H	I

P2

				J	K
					L

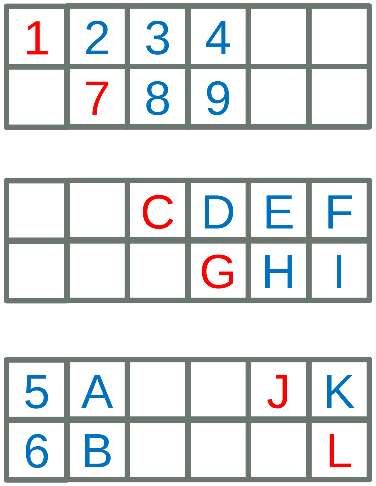
Distributed Array: Example



- Total number of points to be explicitly calculated $f = \frac{n^2 - n}{2} + n$
- Base points per row to be explicitly calculated $r = \frac{f}{n}$

$f=21$

$r=3.5$



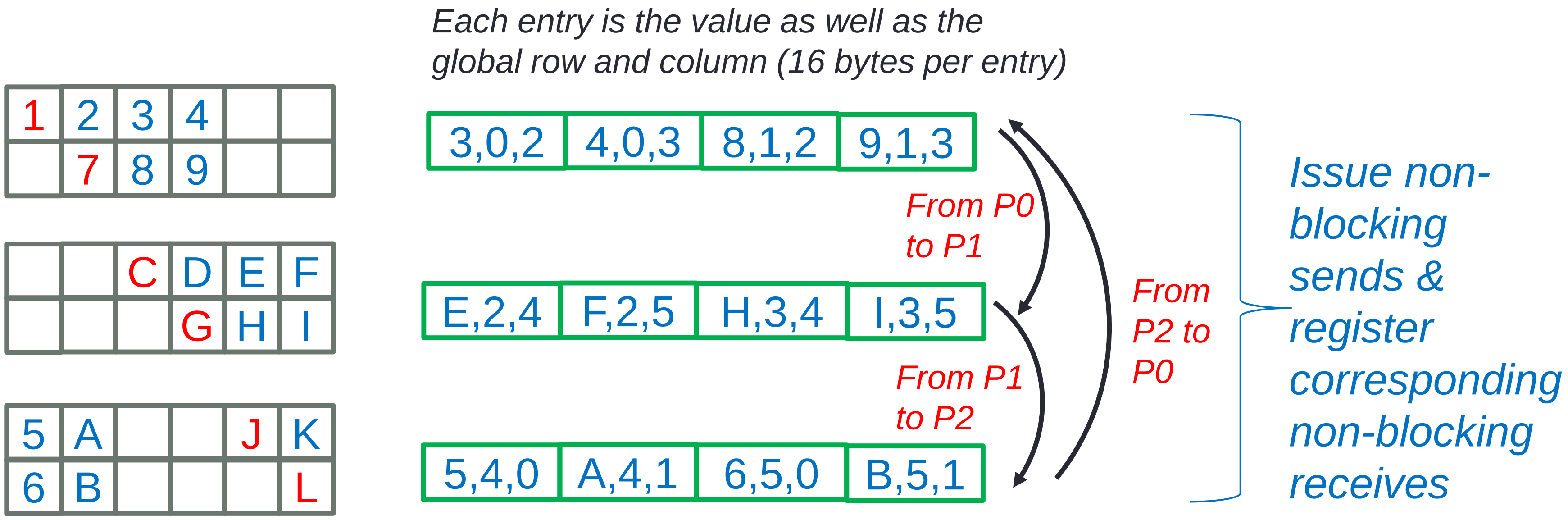
1	2	3	4		
	7	8	9		

		C	D	E	F
			G	H	I

5	A			J	K
6	B				L

- Starting at the diagonal, start calculating r local points.
  - If r is fractional (n is even), alternate between ceil(r) and floor(r) points for each row
  - If the number of rows/2 is even, then in the second half of the matrix swap over ceil/floor

Distributed Array: Example



Each entry is the value as well as the global row and column (16 bytes per entry)

1	2	3	4		
	7	8	9		

		C	D	E	F
			G	H	I

5	A			J	K
6	B				L

3,0,2	4,0,3	8,1,2	9,1,3

From P0 to P1

E,2,4	F,2,5	H,3,4	I,3,5

From P1 to P2

5,4,0	A,4,1	6,5,0	B,5,1

From P2 to P0

Issue non-blocking sends & register corresponding non-blocking receives

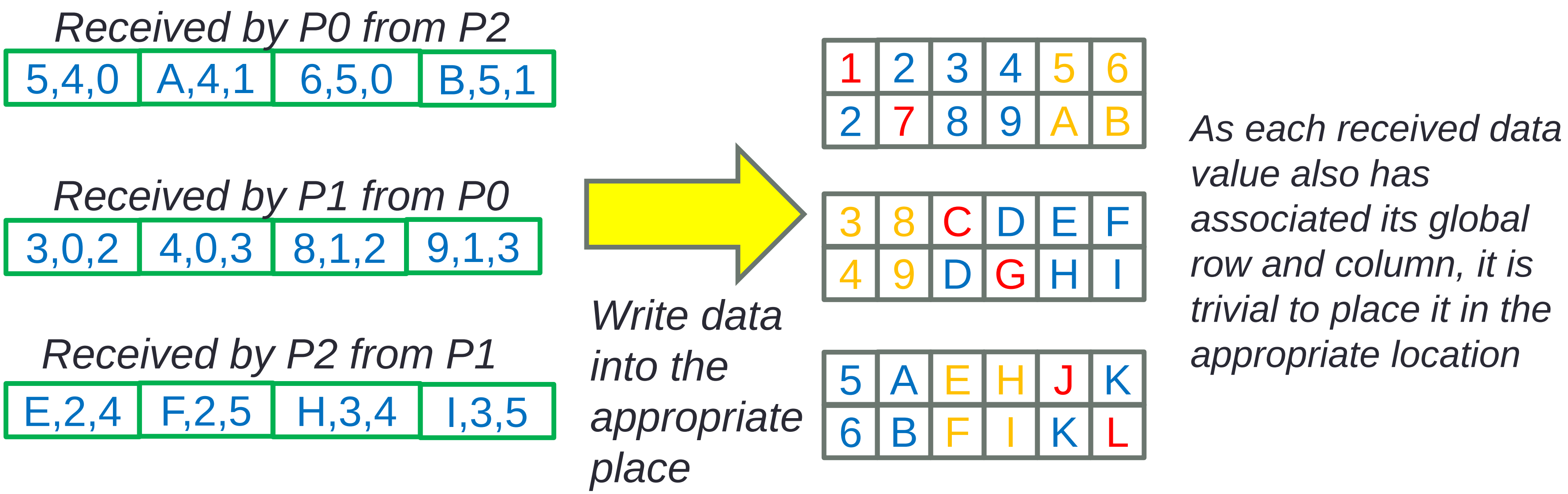
- Next we copy all local values (between locally held rows)
- Once we have done this wait for all communications to complete
  - Overlapping the local data copy with the communications

1	2	3	4		
2	7	8	9		

		C	D	E	F
		D	G	H	I

5	A			J	K
6	B			K	L

Distributed Array: Example



Received by P0 from P2

5,4,0	A,4,1	6,5,0	B,5,1

Received by P1 from P0

3,0,2	4,0,3	8,1,2	9,1,3

Received by P2 from P1

E,2,4	F,2,5	H,3,4	I,3,5

Write data into the appropriate place

1	2	3	4	5	6
2	7	8	9	A	B

3	8	C	D	E	F
4	9	D	G	H	I

5	A	E	H	J	K
6	B	F	I	K	L

As each received data value also has associated its global row and column, it is trivial to place it in the appropriate location

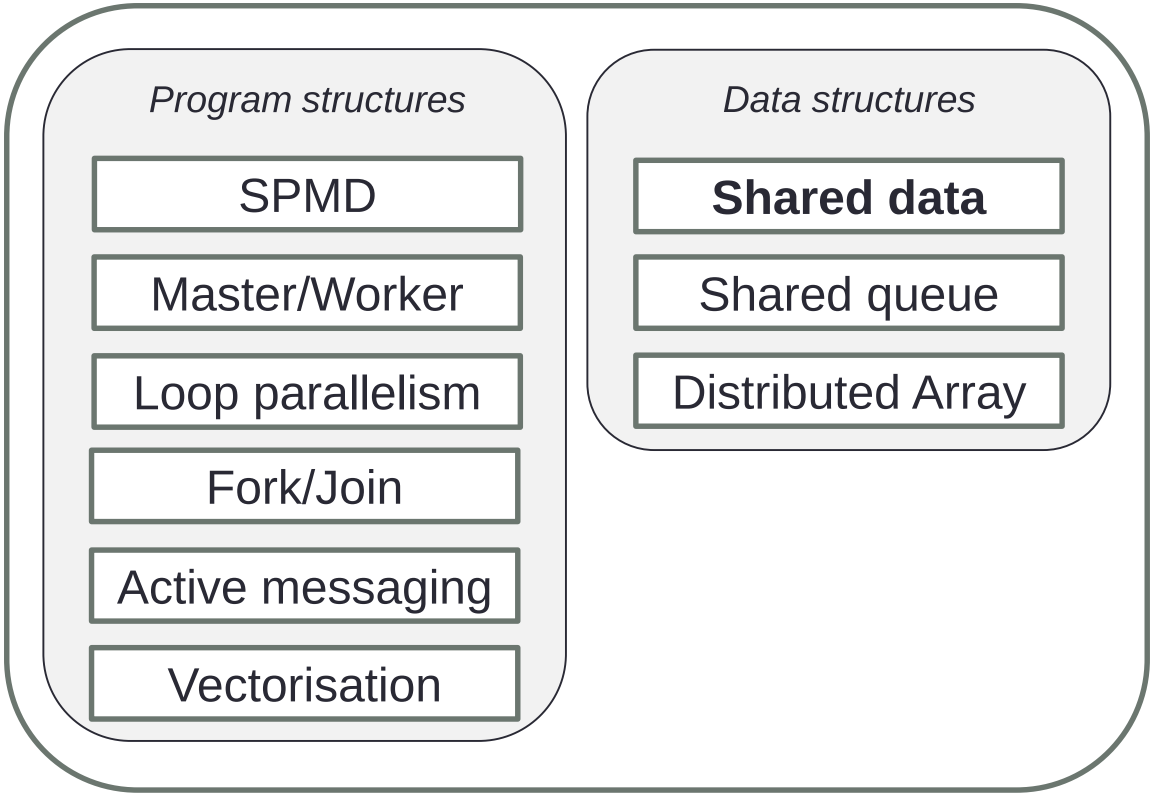
- Whilst we still need communication of the values, we don’t need communication to coordinate which process calculates what
  - At worst each process needs to communicate with every other process, but this is 1 single large message

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).



Program structures

SPMD

Master/Worker

Loop parallelism

Fork/Join

Active messaging

Vectorisation

Data structures

Shared data

Shared queue

Distributed Array

Shared Data: Context

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

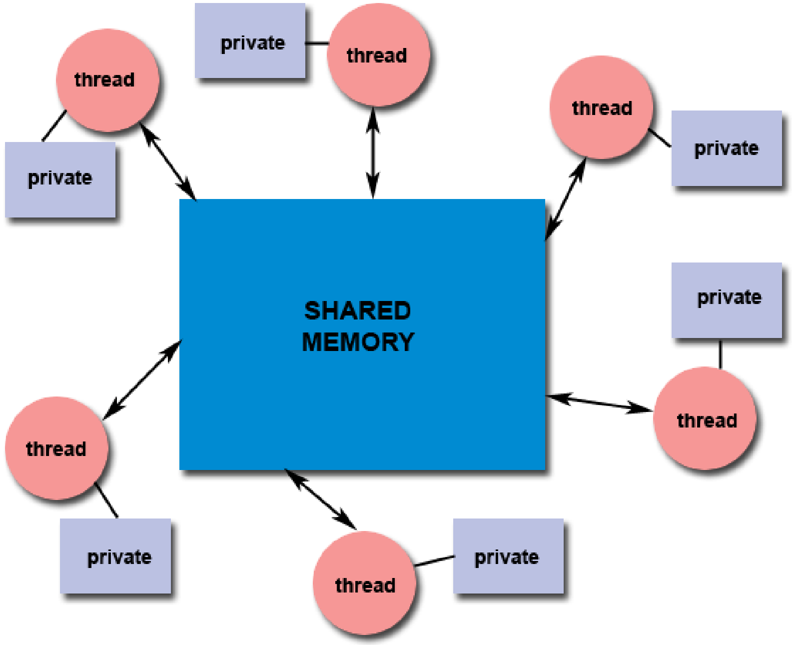
- Some parallel algorithm patterns handle shared data by extracting it from the task.
  - Replication & Reduction with Task Parallelism
  - Halo-swapping with Geometric Decomposition

- 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: Context (2)

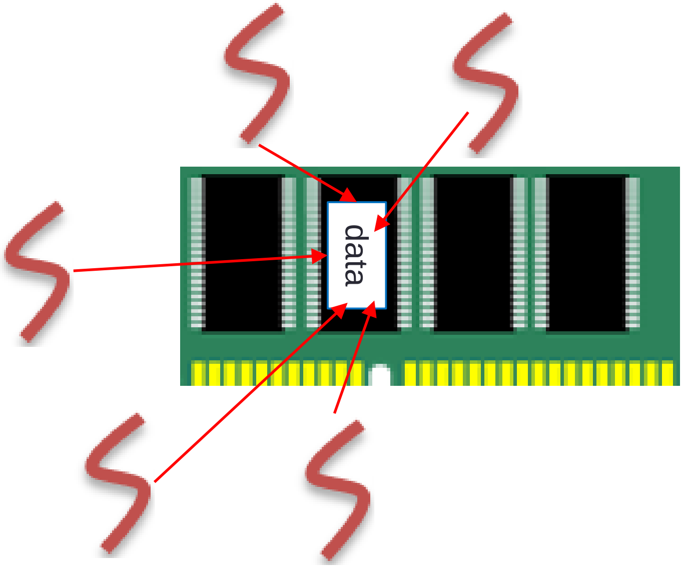
- Common attributes for problems that need the *Shared Data* pattern:
  - At least one data structure is accessed by multiple tasks in the course of the program's execution
  - At least one task modifies the shared data structure, and
  - The tasks potentially need to use the modified value during the concurrent computation

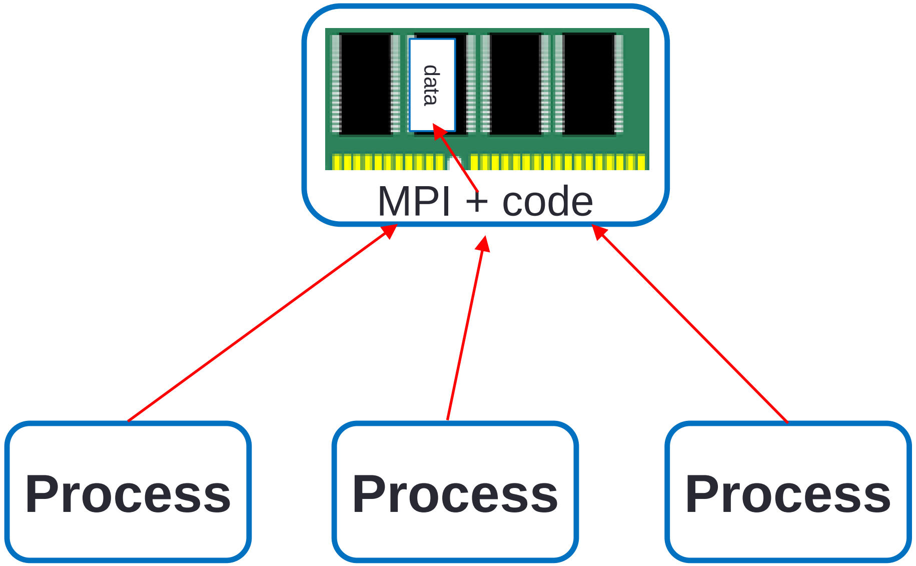
- Most commonly assume this is with shared memory (threaded programming) but can be required with distributed memory too



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





MPI + code

Process

Process

Process

- Techniques for managing shared data can limit the number of tasks that can run concurrently, thereby potentially reducing scalability
- If the constructs used to manage shared data are not easy to understand, the program will be harder to maintain

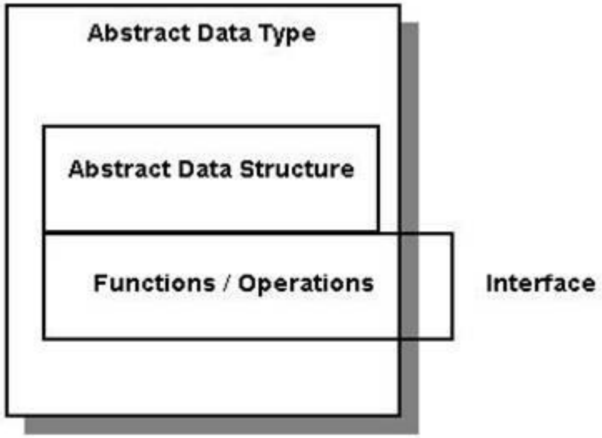
Solution

- Ensure this pattern is needed
  - By revisiting earlier decisions can we find an approach matching one of the algorithm strategy patterns without the need for shared data?

1. Make use of abstract data types (ADTs)
2. 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
3. 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
  - e.g. for a shared queue, you might have *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 the property that they each leave the data in a consistent, meaningful state
- Implementation of individual operations should be such that lower-level actions should not be visible to other tasks/UEs

Concurrency Control Protocols

- Once you have defined an ADT and its operations, 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
  - Uses a Critical Section
    - Provided directly by language, or indirectly through mutex locks, synchronised blocks, OpenMP critical
  - Usually straightforward to implement, but often overly conservative resulting in bottlenecks.

```
function operation1 {
  synchronised {
    ……
  }
}
function operation2 {
  synchronised {
    ……
  }
}
function operation3 {
  synchronised {
    ……
  }
}
```

Concurrency Control Protocols

- Noninterfering 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

```
function operation1 {
   synchronised A {
      ……
   }
}
function operation2 {
   synchronised A {
      ……
   }
}
function operation3 {
   synchronised B {
      ……
   }
}
```

Concurrency Control Protocols

- Readers/Writers
  - If operations cannot be separated out but if some operations modify the data and others only read it then we can go from here.
  - 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 Bs can execute concurrently. *This is the basis for RW locks in pthreads*
  - Introduces some overhead, some thought needed by lock writers

```
function get {
  synchronise read {
    ......
  }
}
function put {
  synchronise write {
    ......
  }
}
function getSize {
  synchronise read {
    ......
  }
}
```

Concurrency Protocols

- Reducing the size of the critical section
  - Don’t put the whole operation in a critical section
  - Analyze the operations in more detail, does only one aspect cause interference?
  - Very easy to get wrong, so be careful!
  - Repeated locking and unlocking can be expensive

```
function operation1 {
  synchronised {
    ……
  }
}
function operation2 {
  ……
  synchronised {
    ……
  }
  ……
}
function operation3 {
  ……
  synchronised {
    ……
  }
  ……
  synchronised {
    ……
  }
}
```

Concurrency Protocols

- 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 x and y, and B reads and writes to y then strictly speaking these interfere. However, can place a lock around A's y access to allow for additional concurrency
  - Increased potential for deadlock

```
function operation1 {
  synchronised A {
    ......
    synchronised B {
      ......
    }
  }
}
function operation2 {
  synchronised B {
    ......
  }
}
function operation3 {
  synchronised B {
    ......
    synchronised A {
      ......
    }
  }
}
```

Concurrency Protocols

- Application specific semantic relaxation
  - e.g. partially replicate shared data, and don’t keep all of the copies completely in sync
  - In some cases may involve a duplication of work (i.e. a number of tasks searching for an answer based upon the same starting conditions) but this can be more efficient than managing shared data to avoid this.
  - Application logic means that conflict can never happen in reality

```
function operation1 {
   ……
}
function operation2 {
   ……
}
function operation3 {
   ……
}
```

Other considerations

- Memory synchronisation
  - Caching and compiler optimisation can result in unexpected behaviour.
  - I.e. a stale value might be read from a cache or a new value 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 all variables marked with *volatile*.
  - In C or FORTRAN have the *volatile* keyword too, often needed!

- Task scheduling
  - Will a task be idle, waiting for access to some shared data?
  - If so can we assign tasks to UEs in such a way that minimises this?
  - Or can we assign multiple tasks to UEs such that there is always one that is not waiting and doing some work?

Shared data – Summary

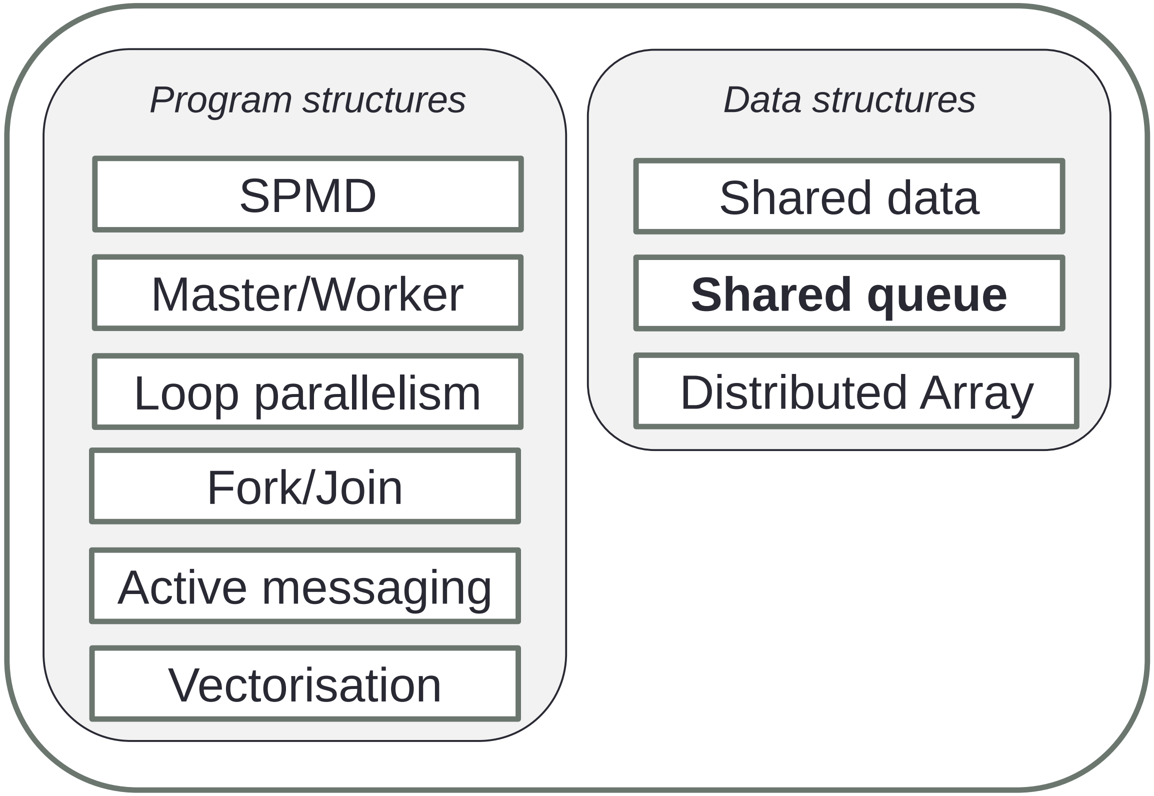
- First consider if you really have to use this pattern.

- 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?
- 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.



Program structures

SPMD

Master/Worker

Loop parallelism

Fork/Join

Active messaging

Vectorisation

Data structures

Shared data

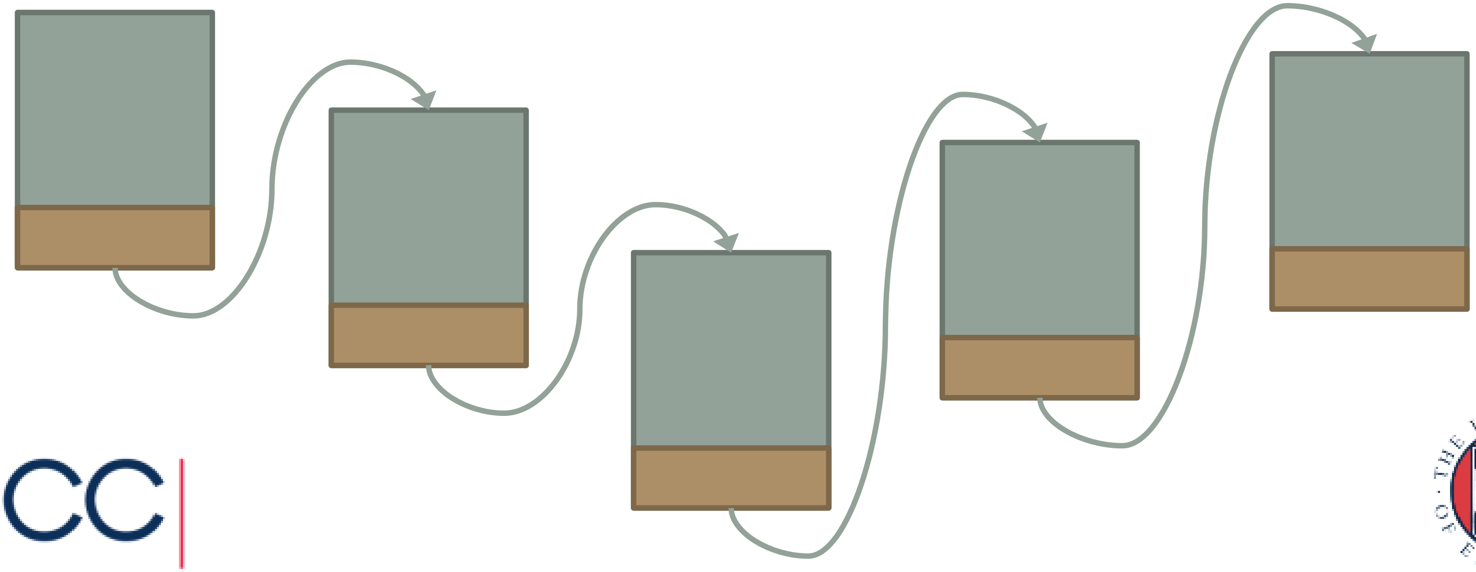
Shared queue

Distributed Array

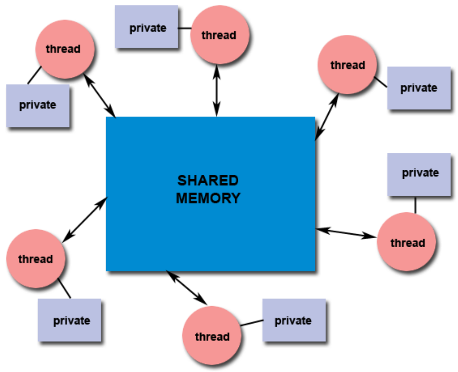
Shared Queue – Solution

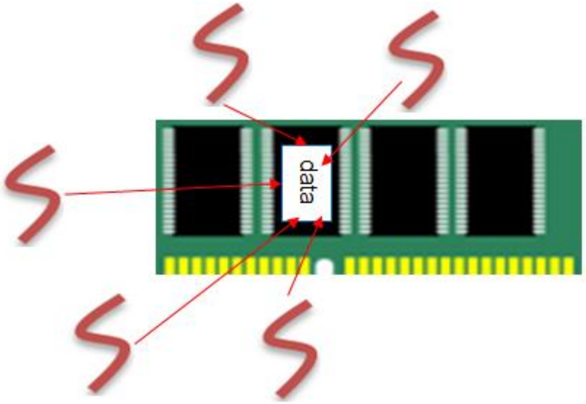
- The queue is a FIFO data type.

- Often implemented as a linked list.



Effect of Concurrency-Control Protocol





- Most 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

Simple but slow

↓

Complex but fast

Shared Queue: Forces

- Simple concurrency-control protocols provide greater clarity of abstraction and make it easier for the programmer to verify that the shared queue has been correctly implemented
  - *Aim for clarity first, then optimise*

- Concurrency-control protocols that encompass too much of the shared queue in a single synchronisation construct increase the chances UEs will remain blocked waiting to access the queue and will limit concurrency

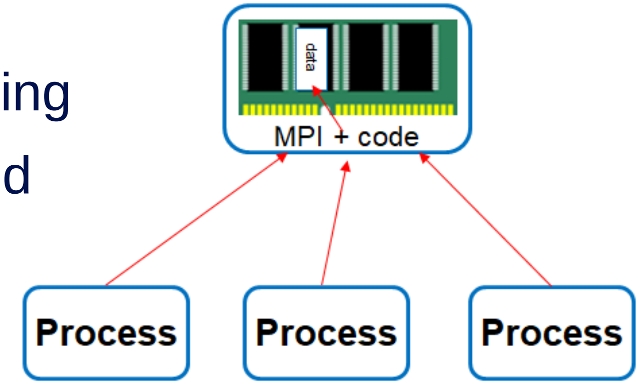
- A concurrency-control protocol finely tuned to the queue and how it will be used increases the available concurrency, at the cost of more complicated, more error-prone synchronisation constructs

Solution

- Ideally the shared queue would be implemented as part of the target programming language
  - e.g. Java has an implementation available in java.util.concurrent

- No provided mechanism in common HPC languages (MPI, OpenMP)
- Most common use of shared queue is with *shared memory*

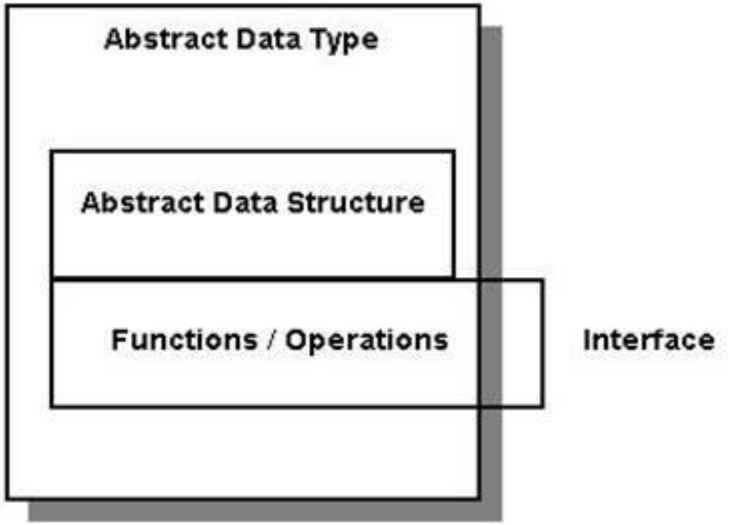
- Can be implemented in *message passing* by having the queue owned by one process, and putting and taking from the queue implemented by sending messages to and from the owner process



Solution

Apply the shared data pattern

- Define the ADT
- Choose the concurrency protocol



Defining the ADT

- The operations:
  - Put *(enqueue)*
  - Take *(dequeue)*
  - Other operations are possible, e.g. peek, takeall, clear, isEmpty

- Details:
  - What do you 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

Concurrency control protocol

- Implementing a shared queue can be tricky
  - but well-written, it 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 SharedQueue1 {
  class Node { //inner class defines list 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) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
  }

  public synchronized Object take() {
    //returns first task in queue or null if queue is empty
    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;} }
```

OpenMP version

- A simple queue of ints, for illustration purposes:

```
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 queue empty

```
public class SharedQueue2 {
  class Node {
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }
  private Node head = new Node(null);
  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;
    notifyAll();
  }

  public synchronized Object take() {
    //returns first task in queue, waits if queue is empty
    Object task = null;
    while (isEmpty()) {
      try{wait();}catch(InterruptedException ignore){}
    }
    Node first = head.next;
    task = first.task;
    first.task = null;
    head = first;
    return task; } }
```

- Wait will release lock
  - Waits until notified
- notifyAll wakes all threads
  - In tern as lock on take method
- Pthreads has condition variables
  - Wait and signal

```
public class SharedQueue1 {
  class Node { //inner class defines list 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) {
    assert task != null: "Cannot insert null task";
    Node p = new Node(task);
    last.next = p;
    last = p;
  }

  public synchronized Object take() {
    //returns first task in queue or null if queue is empty
    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;} }
```

Non-interfering operations

```
public class SharedQueue3 {
  class Node {
    Object task;
    Node next;
    Node(Object task) {this.task = task; next = null;}
  }

  private Node head = new Node(null);
  private Node last = head;

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

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

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

- Put and take are independent as do not access the same variables

- Therefore use different locks

- Only works for non blocking
- Could be two different mutexes in pthreads

OpenMP version

- A simple queue of ints, for illustration purposes:

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

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

Nested locks

```
pubic class SharedQueue4 {
  class Node {
    Object task; Node next;
    Node(Object task) {
      this.task = task; next = null;}
  }
  private Node head = new Node(null);
  private Node last = head;
  private int w;
  private Object putLock = new Object();
  private Object takeLock = new Object();

  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 Object take() {
    Object task = null;
    synchronized(takeLock) {
      //returns first task in queue, waits if queue is empty
      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; } }
```

- Blocking on empty
- Waits on the putLock lock
- Need to be very careful to avoid deadlock

Readers and writers

```
private Node last = head;

  Rwlock rw_lock=new Rwlock();

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

  public Object viewlast() {
    Object task = null;
    rw_lock.readLock();
    if (!isEmpty()) {
      task=last.task;
    }
    rw_lock.release();
    return task; } }
```

- Here *last* is used in both the functions
  - But one writes whilst the other reads
  - The reader can operate concurrently
  - Only one writer exclusively
- An example of this is rwlocks in pthreads

Shrinking the critical section

```
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();
  }

  public Object viewlast() {
    Object task = null;
    rw_lock.readLock();
    if (!isEmpty()) {
      task=last.task;
    }
    rw_lock.release();
    return task; } }
```

Distributed shared queues

- One central queue can be a bottleneck
  - Can we split this up so there is a queue per UE and distribute the contents?
- 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

- E.g. Allocating tasks to each UE to execute, queue these up and then allow for work stealing once completed.

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.