Parallel Algorithms

Fork and Join

The next general structuring method to look at is *fork and join*

Parallel Algorithms

Fork and Join

The next general structuring method to look at is *fork and join*

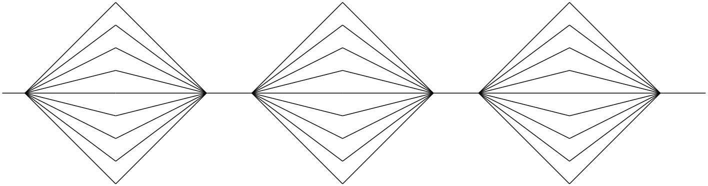
We have seen this before, as it is just the superstep

Parallel Algorithms

Fork and Join

The next general structuring method to look at is *fork and join*

We have seen this before, as it is just the superstep



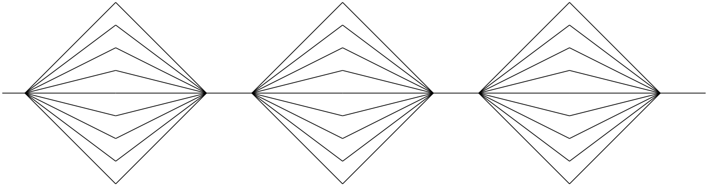
Superstep

Parallel Algorithms

Fork and Join

The next general structuring method to look at is *fork and join*

We have seen this before, as it is just the superstep



Superstep

Of course, we would like to make the sequential parts between the forks as small as possible

This is quite popular, as many problems decompose this way

Parallel Algorithms

Fork and Join

This is quite popular, as many problems decompose this way

For example, multiply two matrices together *then* add in a third matrix

Parallel Algorithms

Fork and Join

This is quite popular, as many problems decompose this way

For example, multiply two matrices together *then* add in a third matrix

The processing forks to multiply the matrices using parallel sub-tasks, then joins after that

Parallel Algorithms

Fork and Join

This is quite popular, as many problems decompose this way

For example, multiply two matrices together *then* add in a third matrix

The processing forks to multiply the matrices using parallel sub-tasks, then joins after that

We could use barriers between the two phases

Parallel Algorithms

Fork and Join

Take care not to confuse the structure of fork and join with the creation and joining of threads

Parallel Algorithms

Fork and Join

Take care not to confuse the structure of fork and join with the creation and joining of threads

“Fork and join” describes the concurrency in the execution, not the mechanism for execution

Parallel Algorithms

Fork and Join

Take care not to confuse the structure of fork and join with the creation and joining of threads

“Fork and join” describes the concurrency in the execution, not the mechanism for execution

We might want to do the sub-tasks provider/consumer, or manager/worker or thread pool or whatever

Parallel Algorithms

Fork and Join

Take care not to confuse the structure of fork and join with the creation and joining of threads

"Fork and join" describes the concurrency in the execution, not the mechanism for execution

We might want to do the sub-tasks provider/consumer, or manager/worker or thread pool or whatever

It is very unlikely we would want to use `pthread_create` and `pthread_join` every time

Parallel Algorithms

Pipelines/Systolic

Another structuring method we have seen before is the *pipeline*, also called *systolic array*

Pipeline

Parallel Algorithms

Pipelines/Systolic

Another structuring method we have seen before is the *pipeline*, also called *systolic array*

Pipeline

Input data is transformed by several separate stages by several separate processors

Parallel Algorithms

Pipelines/Systolic

Another structuring method we have seen before is the *pipeline*, also called *systolic array*

Pipeline

Input data is transformed by several separate stages by several separate processors

A well-balanced pipeline (eventually) gives perfect speedup and efficiency

Parallel Algorithms

MapReduce

Finally, for now, we look at another concept imported from the functional style: *MapReduce*

Finally, for now, we look at another concept imported from the functional style: *MapReduce*

This is a combination of a *map* and a *reduce*, and is a kind of divide and conquer

Parallel Algorithms

MapReduce

Finally, for now, we look at another concept imported from the functional style: *MapReduce*

This is a combination of a *map* and a *reduce*, and is a kind of divide and conquer

A map takes a function and a structure (a list or vector or tree or whatever) of data, and applies that function to each element in the structure

Parallel Algorithms

MapReduce

Finally, for now, we look at another concept imported from the functional style: *MapReduce*

This is a combination of a *map* and a *reduce*, and is a kind of divide and conquer

A map takes a function and a structure (a list or vector or tree or whatever) of data, and applies that function to each element in the structure

As long as there is no interference between the items of data, this is trivially parallelisable: stick different items of data on different processors and execute the function on each

Parallel Algorithms

MapReduce

The reduce step then gathers together all the sub-results and merges them together to produce the required answer

Parallel Algorithms

MapReduce

The reduce step then gathers together all the sub-results and merges them together to produce the required answer

Depending on what kind of reduction we require, this can be extensively parallelised, too

Parallel Algorithms

MapReduce

The reduce step then gathers together all the sub-results and merges them together to produce the required answer

Depending on what kind of reduction we require, this can be extensively parallelised, too

E.g., the merge in a parallel sum being done in a tree-like way

Parallel Algorithms

MapReduce

The reduce step then gathers together all the sub-results and merges them together to produce the required answer

Depending on what kind of reduction we require, this can be extensively parallelised, too

E.g., the merge in a parallel sum being done in a tree-like way

E.g., the merge of URLs that result from a Web search can be done similarly, perhaps a sort in order of relevance

Parallel Algorithms

MapReduce

The reduce step then gathers together all the sub-results and merges them together to produce the required answer

Depending on what kind of reduction we require, this can be extensively parallelised, too

E.g., the merge in a parallel sum being done in a tree-like way

E.g., the merge of URLs that result from a Web search can be done similarly, perhaps a sort in order of relevance

Other reductions might be less or more parallelisable

Parallel Algorithms

MapReduce

For example, given a vector of numbers compute the sum of the squares of the values

Parallel Algorithms

MapReduce

For example, given a vector of numbers compute the sum of the squares of the values

Map: do the squares in parallel

Parallel Algorithms

MapReduce

For example, given a vector of numbers compute the sum of the squares of the values

Map: do the squares in parallel

Reduce: add them together in parallel

Parallel Algorithms

MapReduce

Another example: Web search. The data is distributed in chunks across many machines

Parallel Algorithms

MapReduce

Another example: Web search. The data is distributed in chunks across many machines

Map: a machine searches its own chunk

Parallel Algorithms

MapReduce

Another example: Web search. The data is distributed in chunks across many machines

Map: a machine searches its own chunk

Reduce: merging and sorting the partial results

Parallel Algorithms

MapReduce

Another example: Web search. The data is distributed in chunks across many machines

Map: a machine searches its own chunk

Reduce: merging and sorting the partial results

MapReduce is much used by Google for their various services, not just searching

Parallel Algorithms

MapReduce

This clearly scales well to huge systems!

This clearly scales well to huge systems!

This is helped a lot helped by the source data being stationary and sending the map function to the machine that hosts the data: a reversal of the way we normally think about things

Parallel Algorithms

MapReduce

This clearly scales well to huge systems!

This is helped a lot helped by the source data being stationary and sending the map function to the machine that hosts the data: a reversal of the way we normally think about things

MapReduce also copes well with less than 100% reliability of the hardware

Parallel Algorithms

Aside: Reliability

A quick word on reliability: modern machines are pretty reliable and we are not used to them breaking down too often

Parallel Algorithms

Aside: Reliability

A quick word on reliability: modern machines are pretty reliable and we are not used to them breaking down too often

Huge clusters are a different proposition entirely

Parallel Algorithms

Aside: Reliability

A quick word on reliability: modern machines are pretty reliable and we are not used to them breaking down too often

Huge clusters are a different proposition entirely

When you have 100s of thousands of machines in your system, you must plan for one to break down in the middle of your computation!

Parallel Algorithms

Aside: Reliability

A quick word on reliability: modern machines are pretty reliable and we are not used to them breaking down too often

Huge clusters are a different proposition entirely

When you have 100s of thousands of machines in your system, you must plan for one to break down in the middle of your computation!

So another issue large systems and the algorithms that run on them have to contend with is machines failing

Parallel Algorithms

Aside: Reliability

For example, you might want to run the same sub-task on more than one processor for reliability: if one breaks you'll still get the result

Parallel Algorithms

Aside: Reliability

For example, you might want to run the same sub-task on more than one processor for reliability: if one breaks you'll still get the result

At one point Hector, a UK academic cluster, was having a failure rate of one node per day

Parallel Algorithms

Classical Problems

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

Parallel Algorithms

Classical Problems

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

The first is *readers/writers*, which looks at synchronisation in the shared use of data, in, for example, a database

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

The first is *readers/writers*, which looks at synchronisation in the shared use of data, in, for example, a database

Some processes may want to simply read data, a *reader*

Parallel Algorithms

Classical Problems

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

The first is *readers/writers*, which looks at synchronisation in the shared use of data, in, for example, a database

Some processes may want to simply read data, a *reader*

Others might want to read and then update data, a *writer*

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

The first is *readers/writers*, which looks at synchronisation in the shared use of data, in, for example, a database

Some processes may want to simply read data, a *reader*

Others might want to read and then update data, a *writer*

To ensure consistency in the data, a writer must have exclusive access to the database

Parallel Algorithms

Classical Problems

We now turn to look at a few classical problems that are used to illustrate the issues that arise in designing parallel programs

The first is *readers/writers*, which looks at synchronisation in the shared use of data, in, for example, a database

Some processes may want to simply read data, a *reader*

Others might want to read and then update data, a *writer*

To ensure consistency in the data, a writer must have exclusive access to the database

(A simplification of reality, if you know anything about databases)

Parallel Algorithms

Readers/Writers

When there is no writer using the database, any number of readers can access it simultaneously

When there is no writer using the database, any number of readers can access it simultaneously

Note, as a consequence of exclusive access, a writer cannot access the database while there is any reader using it

Parallel Algorithms

Readers/Writers

When there is no writer using the database, any number of readers can access it simultaneously

Note, as a consequence of exclusive access, a writer cannot access the database while there is any reader using it

One solution is to use simple primitives

Parallel Algorithms

Readers/Writers

```
int readers = 0;
rlock = make_lock();      // protect readers
wsem = make_semaphore(1);// sync writers
```

```
void reader()
{
  lock(rlock);
  readers++;
  if (readers == 1) wait(wsem);
  unlock(rlock);
  ... read ...
  lock(rlock);
  readers--;
  if (readers == 0) signal(wsem);
  unlock(rlock);
}
```

```
void writer()
{
  wait(wsem);
  ... write ...
  signal(wsem);
}
```

Parallel Algorithms

Readers/Writers

The `rlock` is to protect the count of the number of readers

Parallel Algorithms

Readers/Writers

The `rlock` is to protect the count of the number of readers

The `wsem` synchronises the readers and writers: a writer must wait until all readers have left, and a reader must wait until a writer has left

The `rlock` is to protect the count of the number of readers

The `wsem` synchronises the readers and writers: a writer must wait until all readers have left, and a reader must wait until a writer has left

`if (readers == 1) wait(wsem);` the first reader in sets the write semaphore

The `rlock` is to protect the count of the number of readers

The `wsem` synchronises the readers and writers: a writer must wait until all readers have left, and a reader must wait until a writer has left

`if (readers == 1) wait(wsem);` the first reader in sets the write semaphore

`if (readers == 0) signal(wsem);` the last reader out releases the semaphore

The `rlock` is to protect the count of the number of readers

The `wsem` synchronises the readers and writers: a writer must wait until all readers have left, and a reader must wait until a writer has left

`if (readers == 1) wait(wsem);` the first reader in sets the write semaphore

`if (readers == 0) signal(wsem);` the last reader out releases the semaphore

This works, but has a problem

Parallel Algorithms

Readers/Writers

The problem is that this code is unfair in the way it treats readers and writers

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`

Parallel Algorithms

Readers/Writers

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`

Parallel Algorithms

Readers/Writers

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`
- reader 2 arrives; it can continue

Parallel Algorithms

Readers/Writers

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`
- reader 2 arrives; it can continue
- reader 1 leaves

Parallel Algorithms

Readers/Writers

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`
- reader 2 arrives; it can continue
- reader 1 leaves
- reader 3 arrives; it can continue

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`
- reader 2 arrives; it can continue
- reader 1 leaves
- reader 3 arrives; it can continue
- reader 2 leaves

The problem is that this code is unfair in the way it treats readers and writers

A writer can be excluded for an arbitrarily long time while readers come and go

- reader 1 arrives and sets the `wsem`
- a writer arrives; it waits on `wsem`
- reader 2 arrives; it can continue
- reader 1 leaves
- reader 3 arrives; it can continue
- reader 2 leaves
- and so on

This is called *readers' preference*

This is called *readers' preference*

The continuing stream of readers conspire to keep out the writer: the readers never signal the `wsem`

This is called *readers' preference*

The continuing stream of readers conspire to keep out the writer: the readers never signal the `wsem`

With low probability, but it happens

This is called *readers' preference*

The continuing stream of readers conspire to keep out the writer: the readers never signal the `wsem`

With low probability, but it happens

This is *starvation* of the writer

We might try to fix the writer starvation by having a writer pending count, and have readers wait if there is a writer (or some suitable number of writers) waiting

We might try to fix the writer starvation by having a writer pending count, and have readers wait if there is a writer (or some suitable number of writers) waiting

Exercise Do this

We might try to fix the writer starvation by having a writer pending count, and have readers wait if there is a writer (or some suitable number of writers) waiting

Exercise Do this

But now we have a writers' preference and readers can be starved

Making this fair for both readers and writers is harder than you think

Parallel Algorithms

Readers/Writers

Making this fair for both readers and writers is harder than you think

Though having a readers' preference is not as bad as you might think, as typical code has more reads than writes

Making this fair for both readers and writers is harder than you think

Though having a readers' preference is not as bad as you might think, as typical code has more reads than writes

Exercise Go and read up on the many suggested solutions to readers/writers

Exercise Read about the POSIX `pthread_rwlock`

Exercise Read about *read-copy-update* (RCU) and its choice of compromises

Exercise Think about how you might use GCD queues

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

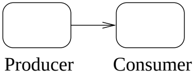
For example, how a manager might feed data to a worker

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

For example, how a manager might feed data to a worker



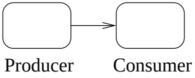
Producer/Consumer

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

For example, how a manager might feed data to a worker



Producer/Consumer

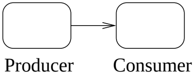
If the producer sends directly to the consumer, this would require a synchronisation between them for every data item

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

For example, how a manager might feed data to a worker



Producer/Consumer

If the producer sends directly to the consumer, this would require a synchronisation between them for every data item

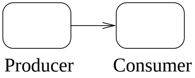
And it would require the consumer to process data at the same rate as the producer produces it (as in a pipeline)

Parallel Algorithms

Producers/Consumers

The next classical problem looks at how two or more processes can communicate: passing data between processes

For example, how a manager might feed data to a worker



Producer/Consumer

If the producer sends directly to the consumer, this would require a synchronisation between them for every data item

And it would require the consumer to process data at the same rate as the producer produces it (as in a pipeline)

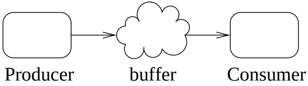
Exercise Compare with MPI

So, typically, there is a *buffer* between them

Parallel Algorithms

Producers/Consumers

So, typically, there is a *buffer* between them

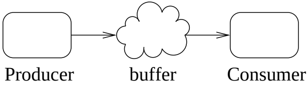


Buffered Producer/Consumer

Parallel Algorithms

Producers/Consumers

So, typically, there is a *buffer* between them



Buffered Producer/Consumer

This is just some area of memory in a shared memory system; or a message queue for a distributed memory system

The advantage is that we can *decouple* the producer and consumer

The advantage is that we can *decouple* the producer and consumer

- each can work at their own rate, until the buffer fills or empties

Parallel Algorithms

Producers/Consumers

The advantage is that we can *decouple* the producer and consumer

- each can work at their own rate, until the buffer fills or empties
- there is less synchronisation, thus less waiting around

Parallel Algorithms

Producers/Consumers

The advantage is that we can *decouple* the producer and consumer

- each can work at their own rate, until the buffer fills or empties
- there is less synchronisation, thus less waiting around
- the producer and consumer are now working *asynchronously*: not synchronising on every message

Parallel Algorithms

Producers/Consumers

When the producer produces data, it writes it into the next free place in the buffer

Parallel Algorithms

Producers/Consumers

When the producer produces data, it writes it into the next free place in the buffer

Unless the buffer is full, when the producer must wait until a place becomes free by the consumer reading some data

Parallel Algorithms

Producers/Consumers

When the producer produces data, it writes it into the next free place in the buffer

Unless the buffer is full, when the producer must wait until a place becomes free by the consumer reading some data

Symmetrically, when the consumer want to consume data, it reads it from the next position in the buffer

Parallel Algorithms

Producers/Consumers

When the producer produces data, it writes it into the next free place in the buffer

Unless the buffer is full, when the producer must wait until a place becomes free by the consumer reading some data

Symmetrically, when the consumer want to consume data, it reads it from the next position in the buffer

Unless the buffer is empty, when the consumer must wait until some data arrives by the producer writing it

When the producer produces data, it writes it into the next free place in the buffer

Unless the buffer is full, when the producer must wait until a place becomes free by the consumer reading some data

Symmetrically, when the consumer want to consume data, it reads it from the next position in the buffer

Unless the buffer is empty, when the consumer must wait until some data arrives by the producer writing it

So there *is* synchronisation, but only when necessary, dictated by the size of the buffer

Parallel Algorithms

Producers/Consumers

When the producer produces data, it writes it into the next free place in the buffer

Unless the buffer is full, when the producer must wait until a place becomes free by the consumer reading some data

Symmetrically, when the consumer want to consume data, it reads it from the next position in the buffer

Unless the buffer is empty, when the consumer must wait until some data arrives by the producer writing it

So there *is* synchronisation, but only when necessary, dictated by the size of the buffer

We need to see how to manage this synchronisation

For example, a buffer of size 1, using two semaphores, called `empty` and `full`

```
              empty = make_semaphore(1);
               full = make_semaphore(0);
producer() {                       consumer() {
  produce data                       wait(full);
  wait(empty);                       take from buffer
  insert in buffer                   signal(empty);
  signal(full);                      consume data
}                                  }
```

Parallel Algorithms

Producers/Consumers

A simple extension to a buffer of size *n* is to use counting semaphores `data` and `free` with `free` initialised to *n*

```
    free = make_counting_semaphore(n);
    data = make_counting_semaphore(0);
```

```
producer() {
  produce data
  wait(free);
  append to buffer
  signal(data);
}
```

```
consumer() {
  wait(data);
  remove from buffer
  signal(free);
  consume data
}
```

Parallel Algorithms

Producers/Consumers

But this works only if appending to and reading from the buffer are independent operations

But this works only if appending to and reading from the buffer are independent operations

In this code as written, the producer and consumer might be acting simultaneously on the buffer: we need to make sure the update does not have a data race

Parallel Algorithms

Producers/Consumers

But this works only if appending to and reading from the buffer are independent operations

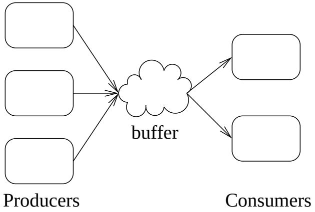
In this code as written, the producer and consumer might be acting simultaneously on the buffer: we need to make sure the update does not have a data race

So, for example, might want a lock on the buffer, or make sure the buffer can otherwise safely support a simultaneous read and write (e.g., for a hash table this might be difficult)

Parallel Algorithms

Producers/Consumers

And things get more interesting when there is more than more producer, or more than one consumer



Multiple Produces/Consumers

Now concurrent access to the buffer is really a problem

Now concurrent access to the buffer is really a problem

We might use a lock to do this

Parallel Algorithms

Producers/Consumers

Now concurrent access to the buffer is really a problem

We might use a lock to do this

```
               free = make_semaphore(1);
               data = make_semaphore(0);
              buffy =  make_lock();
producer() {                  consumer() {
  produce data                  wait(data);
  wait(free);                   get_lock(buffy);
  get_lock(buffy);              take from buffer
  insert in buffer              free_lock(buffy)
  free_lock(buffy);             signal(free);
  signal(data);                 consume data
}                             }
```

Exercise Prove that this cannot deadlock

Exercise Prove that this cannot deadlock

Using one lock means that we cannot insert into the buffer at the same time as reading from it

Exercise Prove that this cannot deadlock

Using one lock means that we cannot insert into the buffer at the same time as reading from it

This is often an unnecessary restriction, e.g., the buffer is an area of memory where we can read one element at the same time as writing a different one

Exercise Prove that this cannot deadlock

Using one lock means that we cannot insert into the buffer at the same time as reading from it

This is often an unnecessary restriction, e.g., the buffer is an area of memory where we can read one element at the same time as writing a different one

Again, this might not be possible if the buffer was some more sophisticated kind of datastructure

So, often we have two locks, one for the insert position and one for the remove position

Parallel Algorithms

Producers/Consumers

So, often we have two locks, one for the insert position and one for the remove position

And we have to be careful when they coincide, e.g., when the buffer is full or empty

Implementations of buffers tend to be either

Implementations of buffers tend to be either

- linked lists (unbounded size)

Implementations of buffers tend to be either

- linked lists (unbounded size)
- fixed arrays, used circularly

Implementations of buffers tend to be either

- linked lists (unbounded size)
- fixed arrays, used circularly

In any case, the buffers are usually actually *queues*, namely first in first out

Parallel Algorithms

Producers/Consumers

More advanced use of queues is possible

More advanced use of queues is possible

If you have just **one** producer, you can implement a *lockless* insert into the queue: namely the insert end does not need a lock (or other synchronisation mechanism)

Parallel Algorithms

Producers/Consumers

More advanced use of queues is possible

If you have just **one** producer, you can implement a *lockless* insert into the queue: namely the insert end does not need a lock (or other synchronisation mechanism)

The “gap” between testing for a space in the buffer and inserting is not a problem as no-one else is inserting data

More advanced use of queues is possible

If you have just **one** producer, you can implement a *lockless* insert into the queue: namely the insert end does not need a lock (or other synchronisation mechanism)

The “gap” between testing for a space in the buffer and inserting is not a problem as no-one else is inserting data

You still have to think carefully about the interaction of this with the removal of data

Symmetrically, if there is just **one** consumer, it is possible to have a lockless read

Parallel Algorithms

Producers/Consumers

Symmetrically, if there is just **one** consumer, it is possible to have a lockless read

These require *extremely* careful programming, but can be useful in reducing overheads

Parallel Algorithms

Producers/Consumers

Symmetrically, if there is just **one** consumer, it is possible to have a lockless read

These require *extremely* careful programming, but can be useful in reducing overheads

Consequently, it is possible to implement a single producer/single consumer entirely lock-free

Parallel Algorithms

Producers/Consumers

Symmetrically, if there is just **one** consumer, it is possible to have a lockless read

These require *extremely* careful programming, but can be useful in reducing overheads

Consequently, it is possible to implement a single producer/single consumer entirely lock-free

Exercise Find out how to do this (it involves memory barriers!)