CSE 332

AUGUST 2ND – SCAN, PACK AND SYNCHRONIZATION

ADMINISTRIVIA

- **Para repo**

ADMINISTRIVIA

- **Para repo**
  - Everyone should have their own para repo for the four parallelism assignments

ADMINISTRIVIA

- **Para repo**
  - Everyone should have their own para repo for the four parallelism assignments

ADMINISTRIVIA

- **Para repo**
  - Everyone should have their own para repo for the four parallelism assignments
- **P3**

ADMINISTRIVIA

- **Para repo**
  - Everyone should have their own para repo for the four parallelism assignments
- **P3**
  - Checkpoint on Friday

PARALLEL PRIMATIVES

- So far we've seen two parallel primatives

PARALLEL PRIMATIVES

- **So far we've seen two parallel primatives**
  - Reduce
    - Return some constant value from the whole array

PARALLEL PRIMATIVES

- **So far we've seen two parallel primatives**
  - Reduce
    - Return some constant value from the whole array
  - Map

PARALLEL PRIMATIVES

- **So far we've seen two parallel primatives**
  - Reduce
    - Return some constant value from the whole array
  - Map
    - Apply some function to each element in the array

PARALLEL PRIMATIVES

- **So far we've seen two parallel primatives**
  - Reduce
    - Return some constant value from the whole array
  - Map
    - Apply some function to each element in the array
- **Together, these are powerful tools of parallelism, but they may not be sufficient**

PARALLEL PRIMATIVES

- We’re going to introduce two new types of problems

PARALLEL PRIMATIVES

- **We're going to introduce two new types of problems**
  - Scan
    - Returns a modified array where each answer depends on the answer before it

PARALLEL PRIMATIVES

- **We're going to introduce two new types of problems**
  - Scan
    - Returns a modified array where each answer depends on the answer before it
  - Pack
    - Filter the array subject to some conditions

PARALLEL PRIMATIVES

- **Scan**

PARALLEL PRIMATIVES

- **Scan**
  - Given an array of integers, mutate the array so that each element contains the sum of the numbers up to that point

PARALLEL PRIMATIVES

- **Scan**
  - Given an array of integers, mutate the array so that each element contains the sum of the numbers up to that point
  - (i.e.) [1,2,3,4] becomes [1,3,6,10]

PARALLEL PRIMATIVES

- **Scan**
  - Given an array of integers, mutate the array so that each element contains the sum of the numbers up to that point
  - (i.e.) [1,2,3,4] becomes [1,3,6,10]
  - This is more complicated than a simple map, the function requires input from all the data before it.

PARALLEL PRIMATIVES

- **Scan**
  - Given an array of integers, mutate the array so that each element contains the sum of the numbers up to that point
  - (i.e.) [1,2,3,4] becomes [1,3,6,10]
  - This is more complicated than a simple map, the function requires input from all the data before it.
  - What are some ways we can parallelize this process?

PARALLEL PRIMATIVES

- **Scan**
  - Given an array of integers, mutate the array so that each element contains the sum of the numbers up to that point
  - (i.e.) [1,2,3,4] becomes [1,3,6,10]
  - This is more complicated than a simple map, the function requires input from all the data before it.
  - What are some ways we can parallelize this process?
    - How do you find the value of a particular node?

PARALLEL PRIMATIVES

- **Partial sum problem**

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?
    - What are some ideas?

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?
    - What are some ideas?
  - What is the actual function?

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?
    - What are some ideas?
  - What is the actual function?
    - Value is going to be the presum + the current value

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?
    - What are some ideas?
  - What is the actual function?
    - Value is going to be the presum + the current value
    - These presum values are going to be reused!

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?
    - What are some ideas?
  - What is the actual function?
    - Value is going to be the presum + the current value
    - These presum values are going to be reused!
    - Think about applying a sum reduce!

PARALLEL PRIMATIVES

- **Partial sum problem**
  - Each node needs information from all the numbers before it
  - How to parallelize?
    - What are some ideas?
  - What is the actual function?
    - Value is going to be the presum + the current value
    - These presum values are going to be reused!
    - How would you apply a sum reduce!
  - Scan trees!

PARALLEL PRIMATIVES

- **While we're waiting for the previous values, we can also be performing our own partial sums, only now are we dependent on a certain number of threads to give us our prescan**
  - Consider it a bunch of sum reduce processes that can communicate with each other

PARALLEL PRIMATIVES

- **While we're waiting for the previous values, we can also be performing our own partial sums, only now are we dependent on a certain number of threads to give us our prescan**
  - Consider it a bunch of sum reduce processes that can communicate with each other
  - Still gives us log n span!

PARALLEL PRIMATIVES

- **Final parallel primative**

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.
  - How do we solve this?

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.
  - How do we solve this?
  - What are the parts of the problem?

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.
  - How do we solve this?
  - What are the parts of the problem?
    - Which numbers are greater than 7

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.
  - How do we solve this?
  - What are the parts of the problem?
    - Which numbers are greater than 7
    - How many of them are there?

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.
  - How do we solve this?
  - What are the parts of the problem?
    - Which numbers are greater than 7
    - How many of them are there? (size of final array)
    - Get the elements in their correct location

PARALLEL PRIMATIVES

- **Final parallel primative**
  - Pack (also known as filter)
  - Return an array of all the numbers greater than 7.
  - How do we solve this?
  - What are the parts of the problem?
    - Which numbers are greater than 7
    - How many of them are there? (size of final array)
    - Get the elements in their correct location

PARALLEL PRIMATIVES

- **Packing**

PARALLEL PRIMATIVES

- **Packing (naïve approach)**
  - First we need to find which elements are greater than 7

PARALLEL PRIMATIVES

- **Packing (naïve approach)**
  - First we need to find which elements are greater than 7
    - This is just a map
  - Second, we need to know how many elements are going to be in our final array

PARALLEL PRIMATIVES

- **Packing (naïve approach)**
  - First we need to find which elements are greater than 7
    - This is just a map
  - Second, we need to know how many elements are going to be in our final array
    - This is just a sum-reduction of the >7 map

PARALLEL PRIMATIVES

- **Packing (naïve approach)**
  - First we need to find which elements are greater than 7
    - This is just a map
  - Second, we need to know how many elements are going to be in our final array
    - This is just a sum-reduction of the >7 map
  - Third, we need to put the correct elements from the original array into their correct place

PARALLEL PRIMATIVES

- **Packing (naïve approach)**
  - First we need to find which elements are greater than 7
    - This is just a map
  - Second, we need to know how many elements are going to be in our final array
    - This is just a sum-reduction of the >7 map
  - Third, we need to put the correct elements from the original array into their correct place
    - How do we parallelize this?

PARALLEL PRIMATIVES

- **Packing**
  - Need to know the final size of the final array

PARALLEL PRIMATIVES

- **Packing**
  - Need to know the final size of the final array
  - Also need to know *where* the elements are going to end up

PARALLEL PRIMATIVES

- **Packing**
  - Need to know the final size of the final array
  - Also need to know *where* the elements are going to end up
  - Our sum-reduce isn't helping us there

PARALLEL PRIMATIVES

- **Packing**
  - Need to know the final size of the final array
  - Also need to know *where* the elements are going to end up
  - Our sum-reduce isn't helping us there
  - Use a sum-scan instead

PARALLEL PRIMATIVES

- **Packing**
  - Need to know the final size of the final array
  - Also need to know *where* the elements are going to end up
  - Our sum-reduce isn't helping us there
  - Use a sum-scan on the map instead
  - The final value will still be the size

PARALLEL PRIMATIVES

- **Packing**
  - Need to know the final size of the final array
  - Also need to know *where* the elements are going to end up
  - Our sum-reduce isn't helping us there
  - Use a sum-scan on the map instead
  - The final value will still be the size
  - Intermediate values will be where the objects are supposed to be stored

PARALLEL PRIMATIVES

- **Packing**
  - How is this useful?

PARALLEL PRIMATIVES

- **Packing**
  - How is this useful?
  - What algorithm do we know that uses a recursive filter?

PARALLEL PRIMATIVES

- **Packing**
  - How is this useful?
  - What algorithm do we know that uses a recursive filter?
  - Quicksort, but there are many others

PARALLEL PRIMATIVES

- **Packing**
  - How is this useful?
  - What algorithm do we know that uses a recursive filter?
  - Quicksort, but there are many others
- **Four primitives**
  - Map – applies a function to an array
  - Reduce – gets a single result from an array
  - Scan – produces an array where results are dependent
  - Pack – filters the array

PARALLELISM

- That covers the basics of parallelism

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover
  - Synchronization and Concurrency

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover
  - Synchronization and Concurrency
- **Right now, we only have the ForkJoin infrastructure**

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover
  - Synchronization and Concurrency
- **Right now, we only have the ForkJoin infrastructure**
  - Two assumptions

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover
  - Synchronization and Concurrency
- **Right now, we only have the ForkJoin infrastructure**
  - Two assumptions
    - Threads are the same code

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover
  - Synchronization and Concurrency
- **Right now, we only have the ForkJoin infrastructure**
  - Two assumptions
    - Threads are the same code
    - Threads only interact at creation and death

PARALLELISM

- **That covers the basics of parallelism**
  - Still two more concepts to cover
  - Synchronization and Concurrency
- **Right now, we only have the ForkJoin infrastructure**
  - Two assumptions
    - Threads are the same code
    - Threads only interact at creation and death
- **If we lift these assumptions, there are other new constraints that we have to consider**

CONCURRENCY

- **The computer has finite resources and it needs to use them as well as it can**

CONCURRENCY

- **The computer has finite resources and it needs to use them as well as it can**
  - Multiple threads allow speed up (work v. span) but they can also allow multiple things to happen at once

CONCURRENCY

- **The computer has finite resources and it needs to use them as well as it can**
  - Multiple threads allow speed up (work v. span) but they can also allow multiple things to happen at once
  - If a process has to go to the disk, the processor has other things it can be doing

CONCURRENCY

- **The computer has finite resources and it needs to use them as well as it can**
  - Multiple threads allow speed up (work v. span) but they can also allow multiple things to happen at once
  - If a process has to go to the disk, the processor has other things it can be doing
  - We can have multiple tasks accessing the same resources I/O, the monitor, the CPU, disk, the terminal, etc…

CONCURRENCY

- **The computer has finite resources and it needs to use them as well as it can**
  - Multiple threads allow speed up (work v. span) but they can also allow multiple things to happen at once
  - If a process has to go to the disk, the processor has other things it can be doing
  - We can have multiple tasks accessing the same resources I/O, the monitor, the CPU, disk, the terminal, etc…
  - Competition needs a moderator, and much of this work is done by the OS

CONCURRENCY

- **The computer has finite resources and it needs to use them as well as it can**
  - Multiple threads allow speed up (work v. span) but they can also allow multiple things to happen at once
  - If a process has to go to the disk, the processor has other things it can be doing
  - We can have multiple tasks accessing the same resources I/O, the monitor, the CPU, disk, the terminal, etc…
  - Competition needs a moderator, and much of this work is done by the OS
  - But as we saw before, this constraint may not be enough

CONCURRENCY

- **Who was born on the 14th?**

CONCURRENCY

- **Who was born on the 14th?**
  - We discussed a few options before, and have analyzed a couple of them

CONCURRENCY

- **Who was born on the 14th?**
  - We discussed a few options before, and have analyzed a couple of them
  - If we don't control write access, we may get the incorrect answer
  - In fact, the answer becomes non-deterministic – we cannot tell what the answer is going to be in advance

CONCURRENCY

- **Start 26 threads that each try to print one of the English characters to the terminal**

CONCURRENCY

- **Start 26 threads that each try to print one of the English characters to the terminal**
  - We can determine that 26 characters will be printed (unless there's a kernel panic)

CONCURRENCY

- **Start 26 threads that each try to print one of the English characters to the terminal**
  - We can determine that 26 characters will be printed (unless there's a kernel panic, or something)

CONCURRENCY

- **Start 26 threads that each try to print one of the English characters to the terminal**
  - We can determine that 26 characters will be printed (unless there's a kernel panic, or something)
  - But, we can't say what order they'll be printed in

CONCURRENCY

- **Start 26 threads that each try to print one of the English characters to the terminal**
  - We can determine that 26 characters will be printed (unless there's a kernel panic, or something)
  - But, we can't say what order they'll be printed in
  - This is called a *race condition* – the output is determined by which processes complete first

CONCURRENCY

- **Start 26 threads that each try to print one of the English characters to the terminal**
  - We can determine that 26 characters will be printed (unless there's a kernel panic, or something)
  - But, we can't say what order they'll be printed in
  - This is called a *race condition* – the output is determined by which processes complete first
  - If this can affect the correctness of our solution, we have a big problem

CONCURRENCY

- **Atomicity**

CONCURRENCY

- **Atomicity**
  - Some sections of code must be performed all at once without any interleaving (when two threads are running at the same time)
  - We call these bits of code a *critical section*

CONCURRENCY

- **Atomicity**
  - Some sections of code must be performed all at once without any interleaving (when two threads are running at the same time)
  - We call these bits of code a *critical section*
  - What is the critical section of our bornOnThe14th problem?

CONCURRENCY

- **Atomicity**
  - Some sections of code must be performed all at once without any interleaving (when two threads are running at the same time)
  - We call these bits of code a *critical section*
  - What is the critical section of our bornOnThe14th problem?
    - Read
    - Sum
    - Write

CONCURRENCY

- **Atomicity**
  - Some sections of code must be performed all at once without any interleaving (when two threads are running at the same time)
  - We call these bits of code a *critical section*
  - What is the critical section of our bornOnThe14th problem?
    - Read
    - Sum
    - Write
  - Only one thread at a time can be doing this. We need *mutual exclusion*

CONCURRENCY

- **Locking**

CONCURRENCY

- **Locking**
  - To preserve this, we need to lock pieces of memory and sections of code
  - To do this, we use a Mutex

CONCURRENCY

- **Locking**
  - To preserve this, we need to lock pieces of memory and sections of code
  - To do this, we use a Mutex
    - Has two fundamental functions
      - Lock() – the thread attempts to monopolize the resource and stalls if the resources is being used

CONCURRENCY

- **Locking**
  - To preserve this, we need to lock pieces of memory and sections of code
  - To do this, we use a Mutex
    - Has two fundamental functions
      - Lock() – the thread attempts to monopolize the resource and stalls if the resources is being used
      - Unlock() – the thread releases the resource for other threads to use
  - The mutex needs to be unique for each resource, NOT for each thread. If the mutex is unique for each thread, then no stalling actually occurs.

CONCURRENCY

- **Locking**
  - What happens if we fail to *unlock* the mutex when we've finished?

CONCURRENCY

- **Locking**
  - What happens if we fail to *unlock* the mutex when we've finished?
    - Other threads will stall forever and never complete

CONCURRENCY

- **Locking**
  - What happens if we fail to *unlock* the mutex when we've finished?
    - Other threads will stall forever and never complete
    - This is *deadlock* – but this isn't hard to prevent, unlock your resources!

CONCURRENCY

- **Locking**
  - What happens if we fail to *unlock* the mutex when we've finished?
    - Other threads will stall forever and never complete
    - This is *deadlock* – but this isn't hard to prevent, unlock your resources!
  - When can deadlock occur, realistically?

CONCURRENCY

- **Locking**
  - What happens if we fail to *unlock* the mutex when we've finished?
    - Other threads will stall forever and never complete
    - This is *deadlock* – but this isn't hard to prevent, unlock your resources!
  - When can deadlock occur, realistically?
    - What if there are multiple resources, and a process needs exclusive access to more than one in order to complete the critical section

CONCURRENCY

- **Locking**
  - Anytime multiple threads have access to the same data structure (no longer constrained to just arrays), access to that data structure has to be constrained by the mutexes

CONCURRENCY

- **Locking**
  - Anytime multiple threads have access to the same data structure (no longer constrained to just arrays), access to that data structure has to be constrained by the mutexes
  - If two threads need the same two resources and each has one lock, they will never complete

CONCURRENCY

- **Locking**
  - Anytime multiple threads have access to the same data structure (no longer constrained to just arrays), access to that data structure has to be constrained by the mutexes
  - If two threads need the same two resources and each has one lock, they will never complete

CONCURRENCY

- **Solutions?**

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again
    - This can introduce a lot of randomization in runtimes

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again
    - This can introduce a lot of randomization in runtimes
  - Recognize when locks come together, and create a lock around getting the lock!

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again
    - This can introduce a lot of randomization in runtimes
  - Recognize when locks come together, and create a lock around getting the lock!
    - Dining philosophers

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again
    - This can introduce a lot of randomization in runtimes
  - Recognize when locks come together, and create a lock around getting the lock!
    - Dining philosophers
  - This could end up with a lot of locks, how do we resolve who gets what?

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again
    - This can introduce a lot of randomization in runtimes
  - Recognize when locks come together, and create a lock around getting the lock!
    - Dining philosophers
  - This could end up with a lot of locks, how do we resolve who gets what?
    - We usually enforce some sort of ordering, where higher priority threads get access first

CONCURRENCY

- **Solutions?**
  - Let go of all of your locks, wait a bit, and try again
    - This can introduce a lot of randomization in runtimes
  - Recognize when locks come together, and create a lock around getting the lock!
    - Dining philosophers
  - This could end up with a lot of locks, how do we resolve who gets what?
    - We usually enforce some sort of ordering, where higher priority threads get access first
    - Guarantees that computation will finish

CONCURRENCY

- **Not everything has this problem**

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific -

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific – isn't a problem because no other thread can access it

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific – isn't a problem because no other thread can access it
    - Immutable memory – isn't a problem because no other thread can change it

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific – isn't a problem because no other thread can access it
    - Immutable memory – isn't a problem because no other thread can change it
    - Shared memory – these are our problem resources

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific – isn't a problem because no other thread can access it
    - Immutable memory – isn't a problem because no other thread can change it
    - Shared memory – these are our problem resources
  - We can resolve some concurrency problems by copying shared memory into thread specific memory

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific – isn't a problem because no other thread can access it
    - Immutable memory – isn't a problem because no other thread can change it
    - Shared memory – these are our problem resources
  - We can resolve some concurrency problems by copying shared memory into thread specific memory
    - Provided we don't write to shared memory based on that access – this needs to be an atomic, critical section

CONCURRENCY

- **Not everything has this problem**
  - Break resources into three types
    - Thread specific – isn't a problem because no other thread can access it
    - Immutable memory – isn't a problem because no other thread can change it
    - Shared memory – these are our problem resources
  - We can resolve some concurrency problems by copying shared memory into thread specific memory
    - Provided we don't write to shared memory based on that access – this needs to be an atomic, critical section
  - If we force input data to be immutable by design, we also don't have to worry about this—this is why in-place sorting isn't always good

FRIDAY

- Concurrency and locking
- Concurrent design
- Granularity
- P3 checkpoint