CSE 332

JULY 26TH – PARALLELISM
FUNDAMENTALS

- Concurrency
FUNDAMENTALS

• Concurrency
  • Even a single core can “do” multiple things at once
FUNDAMENTALS

• Concurrency
  • Even a single core can “do” multiple things at once
  • Processor swapping / Time-slicing
FUNDAMENTALS

• Concurrency
  • Even a single core can “do” multiple things at once
  • Processor swapping / Time-slicing

• Parallelism
  • Breaking the problem into smaller pieces that can be done at once
FUNDAMENTALS

• Concurrency
  • Even a single core can “do” multiple things at once
  • Processor swapping / Time-slicing

• Parallelism
  • Breaking the problem into smaller pieces that can be done at once
  • Born-on-the-14\textsuperscript{th} problem
FUNDAMENTALS

• Synchronization
FUNDAMENTALS

• Synchronization
  • Dealing with shared resources between threads
FUNDAMENTALS

• Synchronization
  • Dealing with shared resources between threads
  • Mutating a single piece of memory
FUNDAMENTALS

• Synchronization
  • Dealing with shared resources between threads
  • Mutating a single piece of memory
  • Write-locking: remember `sum++` is actually a three operation call and how it’s ordered with other operations makes a difference
FUNDAMENTALS

• Threads v. Processes
FUNDAMENTALS

• Threads v. Processes
  • In a standard OS course, you might consider forking a new process when you try and start a new program
FUNDAMENTALS

• Threads v. Processes
  • In a standard OS course, you might consider forking a new process when you try and start a new program
  • Threads work over the same shared memory
FUNDAMENTALS

• Threads v. Processes
  • In a standard OS course, you might consider forking a new process when you try and start a new program
  • Threads work over the same shared memory
    • Each thread has its own calls stack and program counter
FUNDAMENTALS

• Threads v. Processes
  • In a standard OS course, you might consider forking a new process when you try and start a new program
  • Threads work over the same shared memory
    • Each thread has its own calls stack and program counter
    • Can modify freely information in the heap (memory allocated when you call new)
FUNDAMENTALS

• Forking and Joining
FUNDAMENTALS

• Forking and Joining
  • Fork(): Creates a new thread and begins work
• Forking and Joining
  • Fork(): Creates a new thread and begins work
  • Join(): Tells the current thread to wait for the result of a thread it has created
FUNDAMENTALS

• Forking and Joining
  • Fork(): Creates a new thread and begins work
  • Join(): Tells the current thread to wait for the result of a thread it has created

• Example
FUNDAMENTALS

• Forking and Joining
  • Fork(): Creates a new thread and begins work
  • Join(): Tells the current thread to wait for the result of a thread it has created

• Example

RecursiveTask left = new RecursiveTask(/*leffthal*/)
RecursiveTask right = new RecursiveTast(/*Righthalf*/)
left.fork()
result = right.compute()
return combine(left.join,result)
• Forking and Joining
  • Fork(): Creates a new thread and begins work
  • Join(): Tells the current thread to wait for the result of a thread it has created

• Example (Why do it this way?)

```java
RecursiveTask left = new RecursiveTask</*lefthalf*/>()
RecursiveTask right = new RecursiveTask</*Righthalf*/>()
left.fork()
result = right.compute()
return combine(left.join, result)
```
RECURSIVE TASKS

• Basic ideas for good parallel compute functions
  • When given a job, a RecursiveTask is also required to start other recursive tasks.
  • So, the compute function needs to divide the work and create new RecursiveTask objects to do smaller portions of the work.
  • Eventually, once we reach a cutoff point, we want to do the work sequentially (not in parallel)
  • Creating a new thread takes time!
  • Then, we just need to join together all of their tasks
  • The master thread should also do some work
PARALLEL ANALYSIS

• How do we analyze parallel programs?
PARALLEL ANALYSIS

• How do we analyze parallel programs?
  • They do the same amount of work overall, but since they can do multiple things at once, the overall time to completion may be different.
  • We can recognize this difference using work and span
PARALLEL ANALYSIS

How do we analyze parallel programs?

- They do the same amount of work overall, but since they can do multiple things at once, the overall time to completion may be different.
- We can recognize this difference using work and span.
- Work is the total amount of work that needs to be done in the problem (standard sequential computations).
PARALLEL ANALYSIS

• How do we analyze parallel programs?
  • They do the same amount of work overall, but since they can do multiple things at once, the overall time to completion may be different.
  • We can recognize this difference using work and span
  • Work is the total amount of work that needs to be done in the problem (standard sequential computations)
  • Span is the largest amount of work some processor must complete
PARALLEL ANALYSIS

• How do we analyze parallel programs?
  • They do the same amount of work overall, but since they can do multiple things at once, the overall time to completion may be different.
  • We can recognize this difference using work and span
  • Work is the total amount of work that needs to be done in the problem (standard sequential computations)
  • Span is the largest amount of work some processor must complete (this assumes we have infinite processors)
PARALLEL ANALYSIS

• Consider work and span for MergeSort
PARALLEL ANALYSIS

• Consider work and span for MergeSort
  • Work is $O(n \log n)$
PARALLEL ANALYSIS

• Consider work and span for MergeSort
  • Work is $O(n \log n)$
  • Span is only $O(n)$ --
PARALLEL ANALYSIS

• Consider work and span for MergeSort
  • Work is $O(n \log n)$
  • Span is only $O(n)$ – This is the final merge
• We can conduct these as a recurrence
PARALLEL ANALYSIS

• Consider work and span for MergeSort
  • Work is $O(n \log n)$
  • Span is only $O(n)$ – This is the final merge
• We can conduct these as a recurrence
  • Work: $T(N) = O(n) + 2T(n/2)$
PARALLEL ANALYSIS

- Consider work and span for MergeSort
  - Work is $O(n \log n)$
  - Span is only $O(n)$ – This is the final merge
- We can conduct these as a recurrence
  - Work: $T(N) = O(n) + 2T(n/2)$
  - Span: $T(N) = O(n) + \max(T_L(n/2), T_R(n/2))$
PARALLEL ANALYSIS

• Consider work and span for MergeSort
  • Work is $O(n \log n)$
  • Span is only $O(n)$ – This is the final merge
• We can conduct these as a recurrence
  • Work: $T(N) = O(n) + 2T(n/2)$
  • Span: $T(N) = O(n) + \max(T_L(n/2), T_R(n/2))$
• This is the amount of time it will take to complete given infinite processors
PARALLEL ANALYSIS

• Consider work and span for MergeSort
  • Work is $O(n \log n)$
  • Span is only $O(n)$ – This is the final merge
• We can conduct these as a recurrence
  • Work: $T(N) = O(n) + 2T(n/2)$
  • Span: $T(N) = O(n) + \max(T_L(n/2), T_R(n/2))$
• This is the amount of time it will take to complete given infinite processors
  • What we care about emperically is speed up
PARALLEL ANALYSIS

• Speed up
  • Speed up is the time it takes to complete the work sequentially divided by the time it takes to do the work on \( P \) processors
  • If we have four processors and the work takes 1/4 as long, then our speed up is 4.
PARALLEL ANALYSIS

• **Speed up**
  - Speed up is the time it takes to complete the work sequentially divided by the time it takes to do the work on $P$ processors
  - If we have four processors and the work takes 1/4 as long, then our speed up is 4.
  - If the speed up is equal to $P$, then we have perfect linear speed up
PARALLEL ANALYSIS

• **Speed up**
  
  • Speed up is the time it takes to complete the work sequentially divided by the time it takes to do the work on P processors
  
  • If we have four processors and the work takes 1/4 as long, then our speed up is 4.
  
  • If the speed up is equal to P, then we have perfect linear speed up
  
  • However, there is overhead in allocating new threads, and this makes speed up difficult
PARALLEL ANALYSIS

• Speed up
  • Speed up is the time it takes to complete the work sequentially divided by the time it takes to do the work on P processors
  • If we have four processors and the work takes 1/4 as long, then our speed up is 4.
  • If the speed up is equal to P, then we have perfect linear speed up
  • However, there is overhead in allocating new threads, and this makes speed up difficult
  • The theoretical parallelism level is how long the computation would take given infinite processors
PARALLEL ANALYSIS

• Infinite processors
  • Let $T_n$ be the computation time for the problem with $n$ processors and let $T_{\text{inf}}$ be the span
  • Since this is an unrealistic assumption, we can find the lower bound for our operations given $p$ processors
  • $T_p$ is lower bounded by $T_1/P + T_{\text{inf}}$
  • This is where each processor does $1/P$th of the work, but we must also take into account the maximum dependency path
  • Consider finding an element in a BST in parallel
PARALLEL ANALYSIS

- Parallel BST find
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when $T_1$ and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when $T_1$ and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both $O(\text{height})$, creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when \( T_1 \) and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both \( O(\text{height}) \), creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
  • What if the problem was changed to finding an object in an arbitrary non-search tree?
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when \( T_1 \) and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both \( O(\text{height}) \), creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
  • What if the problem was changed to finding an object in an arbitrary non-search tree?
    • Work?
    • Span?
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when $T_1$ and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both $O(\text{height})$, creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
  • What if the problem was changed to finding an object in an arbitrary non-search tree?
    • Work? $O(n)$ need to check all nodes
    • Span?
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when $T_1$ and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both $O(\text{height})$, creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
  • What if the problem was changed to finding an object in an arbitrary non-search tree?
    • Work? $O(n)$ need to check all nodes
    • Span? What is the longest dependency chain?
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when \( T_1 \) and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both \( O(\text{height}) \), creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
  • What if the problem was changed to finding an object in an arbitrary non-search tree?
    • Work? \( O(n) \) need to check all nodes
    • Span? \( O(\text{height}) \)
PARALLEL ANALYSIS

• Parallel BST find
  • What is the work()? What is the analysis of this problem when $T_1$ and we only have one processor
  • What is the span()? What is the analysis of this work considering we have an infinite number of processors?
  • They are both $O(\text{height})$, creating a new thread for both subtrees is pointless, we can eliminate a subtree at each level with a single comparison
  • What if the problem was changed to finding an object in an arbitrary non-search tree?
    • Work? $O(n)$ need to check all nodes
    • Span? $O(\text{height})$ – again this illustrates why linked lists may be poor data structures for parallelization
RECURSIVE TASKS

• Merge sort
RECURSIVE TASKS

• Merge sort
  • What is the work?
RECURSIVE TASKS

• Merge sort
  • What is the work? (Standard sequential $O(n \lg n)$)
RECURSIVE TASKS

• Merge sort
  • What is the work? (Standard sequential $O(n \lg n)$)
  • What is the span?
RECURSIVE TASKS

• **Merge sort**
  • What is the work? (Standard sequential $O(n \lg n)$)
  • What is the span?
    • Break it down into a recurrence!
RECURSIVE TASKS

• Merge sort
  • What is the work? (Standard sequential $O(n \lg n)$)
  • What is the span?
    • Break it down into a recurrence!
  • $T(n) = O(n) + 2T(N/2)$
RECURSIVE TASKS

• **Merge sort**
  - What is the work? (Standard sequential $O(n \log n)$)
  - What is the span?
    • Break it down into a recurrence!
  - $T(n) = O(n) + 2T(N/2)$
  - $T(n) = O(n) + \max(T(\text{left}), T(\text{right}))$
RECURSIVE TASKS

• **Merge sort**
  • What is the work? (Standard sequential $O(n \lg n)$)
  • What is the span?
    • Break it down into a recurrence!
  • $T(n) = O(n) + 2T(N/2)$
  • $T(n) = O(n) + \max(T(\text{left}), T(\text{right}))$
    • ← important distinction for merge sort
RECURSIVE TASKS

• **Merge sort**
  - What is the work? (Standard sequential $O(n \lg n)$)
  - What is the span?
    - Break it down into a recurrence!
  - $T(n) = O(n) + 2T(N/2)$
  - $T(n) = O(n) + \max(T(\text{left}), T(\text{right}))$
    - ← important distinction for merge sort
  - Span then is $O(n)$
PARALLEL ANALYSIS

- Data storage
PARALLEL ANALYSIS

• Data storage
  • Parallel operations don’t have to work only on arrays, they can work on other data structures as well, but they don’t always give a benefit
PARALLEL ANALYSIS

• Data storage
  • Parallel operations don’t have to work only on arrays, they can work on other data structures as well, but they don’t always always give a benefit
  • Why might we not want to parallelize a process over a linked list?
PARALLEL ANALYSIS

• Data storage
  • Parallel operations don’t have to work only on arrays, they can work on other data structures as well, but they don’t always give a benefit
  • Why might we not want to parallelize a process over a linked list?
  • Difficult to break the problem into parts of equal size
PARALLEL ANALYSIS

• Data storage
  • Parallel operations don’t have to work only on arrays, they can work on other data structures as well, but they don’t always always give a benefit
  • Why might we not want to parallelize a process over a linked list?
  • Difficult to break the problem into parts of equal size
  • Exception, if creating a new thread is has lower overhead than the function being performed, i.e. if we are “mapping” a difficult problem onto the result.
COMMON PARALLEL PROBLEMS

- There are two simple “parallelizable” problems that we want to introduce
COMMON PARALLEL PROBLEMS

• There are two simple “parallelizable” problems that we want to introduce
  • Reduction:
    • The input is an array of data
    • The output is some single characteristic of the whole data
    • Examples: Max, sum, contains, count, is-sorted
COMMON PARALLEL PROBLEMS

- There are two simple “parallelizable” problems that we want to introduce
  - Reduction:
    - The input is an array of data
    - The output is some single characteristic of the whole data
    - Examples: Max, sum, contains, count, is-sorted
  - Map
    - The input is an array of data
    - The output is an array of the same length where each element has had the same function applied to it
COMMON PARALLEL PROBLEMS

• These are two of what are called parallel primitives
  • They are operations that can be applied to solve parallel problems
COMMON PARALLEL PROBLEMS

• These are two of what are called parallel primitives
  • They are operations that can be applied to solve parallel problems
  • Many of the things that we’ll look at will simply be combinations of these two primitives
COMMON PARALLEL PROBLEMS

• These are two of what are called parallel primitives
  • They are operations that can be applied to solve parallel problems
  • Many of the things that we’ll look at will simply be combinations of these two primitives
  • How would we solve a problem to count primes between two values?
COMMON PARALLEL PROBLEMS

- These are two of what are called parallel primitives
  - They are operations that can be applied to solve parallel problems
  - Many of the things that we’ll look at will simply be combinations of these two primitives
  - How would we solve a problem to count primes between two values?
    - Fundamentally, it is a reduction, summing the primes, but it is also a mapping of a function which returns 1 if the number is a prime and 0 otherwise
EXAMPLE PROBLEMS

- Coding can be difficult
EXAMPLE PROBLEMS

• Coding can be difficult
  • Let’s look through some code which implements the java interfaces and get you some practice
• Find the second smallest element in an array?
EXAMPLE PROBLEMS

• Coding can be difficult
  • Let’s look through some code which implements the java interfaces and get you some practice
• Find the second smallest element in an array?
  • What are the immediate challenges?
EXAMPLE PROBLEMS

• Coding can be difficult
  • Let’s look through some code which implements the java interfaces and get you some practice

• Find the second smallest element in an array?
  • What are the immediate challenges?
  • What does the recursive task need to return?
EXAMPLE PROBLEMS

• Coding can be difficult
  • Let’s look through some code which implements the java interfaces and get you some practice
• Find the second smallest element in an array?
  • What are the immediate challenges?
  • What does the recursive task need to return?
  • How do we break up the data?