CSE 332

JULY 24TH –INTRO TO PARALLELISM

PARALLELISM

- **Just like B-trees and memory, processors do not work exactly like we assume they do**

PARALLELISM

- **Just like B-trees and memory, processors do not work exactly like we assume they do**
  - New processors “speed up” not by increasing the clock speed, but by increasing the number of cores available for processing

PARALLELISM

- **Just like B-trees and memory, processors do not work exactly like we assume they do**
  - New processors “speed up” not by increasing the clock speed, but by increasing the number of cores available for processing
  - Multiple things can be calculated at once

PARALLELISM

- **Just like B-trees and memory, processors do not work exactly like we assume they do**
  - New processors “speed up” not by increasing the clock speed, but by increasing the number of cores available for processing
  - Multiple things can be calculated at once
  - There are limitations to this, but we can formalize and understand them

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**
  - Synchronization:

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**
  - Synchronization: How can we time our multiple operations so that they are actually running in unison

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**
  - Synchronization: How can we time our multiple operations so that they are actually running in unison
  - Algorithm design:

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**
  - Synchronization: How can we time our multiple operations so that they are actually running in unison
  - Algorithm design: Do we need to change our algorithmic approach so that it can be parallelized

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**
  - Synchronization: How can we time our multiple operations so that they are actually running in unison
  - Algorithm design: Do we need to change our algorithmic approach so that it can be parallelized
  - Concurrent access:

PARALLELISM

- **Three main factors are going to impact our ability to parallelize:**
  - Synchronization: How can we time our multiple operations so that they are actually running in unison
  - Algorithm design: Do we need to change our algorithmic approach so that it can be parallelized
  - Concurrent access: Do we need to modify data structures so they can be safely accessed

PARALLELISM

- Concurrency

PARALLELISM

- **Concurrency**
  - Your computer does multiple things at once

PARALLELISM

- **Concurrency**
  - Your computer does multiple things at once
    - Even when just on one core

PARALLELISM

- **Concurrency**
  - Your computer does multiple things at once
    - Even when just on one core
    - Running one program does not prohibit you from starting another

PARALLELISM

- **Concurrency**
  - Your computer does multiple things at once
    - Even when just on one core
    - Running one program does not prohibit you from starting another
  - Key here becomes how these programs share hardware resources

PARALLELISM

- **Concurrency**
  - Your computer does multiple things at once
    - Even when just on one core
    - Running one program does not prohibit you from starting another
  - Key here becomes how these programs share hardware resources
    - Need to moderate access to memory and CPU process time

PARALLELISM

- **Parallelism**

PARALLELISM

- **Parallelism**
  - Break the work of a single problem down so that it can be completed by many smaller agents

PARALLELISM

- **Parallelism**
  - Break the work of a single problem down so that it can be completed by many smaller agents
  - How many students in this class were born on the 14th of their month?

PARALLELISM

- **Parallelism**
  - Break the work of a single problem down so that it can be completed by many smaller agents
  - How many students in this class were born on the 14th of their month?
    - Two approaches:

PARALLELISM

- **Parallelism**
  - Break the work of a single problem down so that it can be completed by many smaller agents
  - How many students in this class were born on the 14th of their month?
    - Two approaches:
      - Poll each student and keep a counter

PARALLELISM

- **Parallelism**
  - Break the work of a single problem down so that it can be completed by many smaller agents
  - How many students in this class were born on the 14th of their month?
    - Two approaches:
      - Poll each student and keep a counter
      - Find a way for students to talk to each other and communicate back to the main thread

PARALLELISM

- **Synchronization**
  - What if each student tries to modify the “master record”?

PARALLELISM

- **Synchronization**
  - What if each student tries to modify the “master record”?
  - What is the process involved in a computer?

PARALLELISM

- **Synchronization**
  - What if each student tries to modify the “master record”?
  - What is the process involved in a computer?
  - If you increase by `count++`, what does this entail?

PARALLELISM

- **Synchronization**
  - What if each student tries to modify the “master record”?
  - What is the process involved in a computer?
  - If you increase by `count++`, what does this entail?
  - A read, a calculation and then a write

PARALLELISM

- **Synchronization**
  - What if each student tries to modify the “master record”?
  - What is the process involved in a computer?
  - If you increase by `count++`, what does this entail?
  - A read, a calculation and then a write
  - If someone writes between your read and write, the data will be incorrect!

PARALLELISM

- **Synchronization**
  - What if each student tries to modify the “master record”?
  - What is the process involved in a computer?
  - If you increase by `count++`, what does this entail?
  - A read, a calculation and then a write
  - If someone writes between your read and write, the data will be incorrect!
  - Need to “lock” that resource so that only you can modify it for that timeframe

PARALLELISM

- These are both concurrent approaches

PARALLELISM

- **These are both concurrent approaches**
  - Parallelism: Have each thread perform a portion of the task (with exclusive access to their piece of memory)

PARALLELISM

- **These are both concurrent approaches**
  - Parallelism: Have each thread perform a portion of the task (with exclusive access to their piece of memory)
  - Synchronization: Have multiple threads work over the same piece of data

PARALLELISM

- **These are both concurrent approaches**
  - Parallelism: Have each thread perform a portion of the task (with exclusive access to their piece of memory)
  - Synchronization: Have multiple threads work over the same piece of data
  - Can be both! Consider matrix multiplication

PARALLELISM

- Great to consider, but how do we actually go about producing parallelism

PARALLELISM

- **Great to consider, but how do we actually go about producing parallelism**
  - Start a new process through the OS

PARALLELISM

- **Great to consider, but how do we actually go about producing parallelism**
  - Start a new process through the OS
    - This allocates new memory and a new program stack, so data can't easily be shared

PARALLELISM

- **Great to consider, but how do we actually go about producing parallelism**
  - Start a new process through the OS
    - This allocates new memory and a new program stack, so data can't easily be shared
  - Create a new "thread" on the current process

PARALLELISM

- **Great to consider, but how do we actually go about producing parallelism**
  - Start a new process through the OS
    - This allocates new memory and a new program stack, so data can't easily be shared
  - Create a new "thread" on the current process
    - Runs over the same memory

PARALLELISM

- **Threads are usually more efficient because of the ease with which they can communicate information with each other**

PARALLELISM

- **Threads are usually more efficient because of the ease with which they can communicate information with each other**
  - How do we make and moderate threads?

PARALLELISM

- **Threads are usually more efficient because of the ease with which they can communicate information with each other**
  - How do we make and moderate threads?
  - ForkJoin infrastructure:

PARALLELISM

- **Threads are usually more efficient because of the ease with which they can communicate information with each other**
  - How do we make and moderate threads?
  - ForkJoin infrastructure:
    - Fork(): creates a new thread and returns which of the two threads the current execution runs on

PARALLELISM

- **Threads are usually more efficient because of the ease with which they can communicate information with each other**
  - How do we make and moderate threads?
  - ForkJoin infrastructure:
    - Fork(): creates a new thread and returns which of the two threads the current execution runs on
    - Join(): waits for the other thread to finish execution and return data (when the thread has finished its task)

PARALLELISM

- **ForkJoin threads implement the `RecursiveTask` object and come from the `ForkJoinPool` infrastructure.**

PARALLELISM

- **ForkJoin threads implement the `RecursiveTask` object and come from the `ForkJoinPool` infrastructure.**
  - On Monday, we will begin looking at how this actually looks in code

PARALLELISM

- Creating new threads takes time and overhead

PARALLELISM

- **Creating new threads takes time and overhead**
  - Two important solutions:

PARALLELISM

- **Creating new threads takes time and overhead**
  - Two important solutions:
    - Parallelize the parallelization process

PARALLELISM

- **Creating new threads takes time and overhead**
  - Two important solutions:
    - Parallelize the parallelization process
    - I'm So Meta Even This Acronym

PARALLELISM

- **Creating new threads takes time and overhead**
  - Two important solutions:
    - Parallelize the parallelization process
    - **I**'m **S**o **M**eta **E**ven **T**his **A**cronym

PARALLELISM

- **Creating new threads takes time and overhead**
  - Two important solutions:
    - Parallelize the parallelization process
    - **I**'m **S**o **M**eta **E**ven **T**his **A**cronym
    - Once the problem is small enough, we want to use sequential approach. Cutoffs are very important

PARALLELISM

- **Parallelize the parallelization process**

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!
  - In Java, these parallel tasks must use the same code

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!
  - In Java, these parallel tasks must use the same code
    - If a thread is created,

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!
  - In Java, these parallel tasks must use the same code
    - If a thread is created,
      - Check to see if my work is below the cutoff, if so, perform the work

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!
  - In Java, these parallel tasks must use the same code
    - If a thread is created,
      - Check to see if my work is below the cutoff, if so, perform the work
      - If not, divide the work in half and issue a new thread for each

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!
  - In Java, these parallel tasks must use the same code
    - If a thread is created,
      - Check to see if my work is below the cutoff, if so, perform the work
      - ~~If not, divide the work in half and issue a new thread for each~~
      - Issue a new thread to do one half, and then do the other half yourself

PARALLELISM

- **Parallelize the parallelization process**
  - The “master” thread could allocate all of the other threads on its own and at once
    - This takes time and isn’t very parallel!
  - In Java, these parallel tasks must use the same code
    - If a thread is created,
      - Check to see if my work is below the cutoff, if so, perform the work
      - ~~If not, divide the work in half and issue a new thread for each~~
      - Issue a new thread to do one half, and then do the other half yourself
  - This will parallelize the thread creation process and limit the number of threads that are waiting to be joined.

RECURSIVE TASKS

- **Parallelism in Java works around RecursiveTask objects**

RECURSIVE TASKS

- **Parallelism in Java works around RecursiveTask objects**
  - They have a constructor, which passes in all of the relevant information needed to perform the work
  - Also, they have a function compute() which is where the work is conducted

RECURSIVE TASKS

- **Parallelism in Java works around RecursiveTask objects**
  - They have a constructor, which passes in all of the relevant information needed to perform the work
  - Also, they have a function compute() which is where the work is conducted
  - Recursive Tasks have a generic type that is the type their final compute should return

RECURSIVE TASKS

- **Parallelism in Java works around RecursiveTask objects**
  - They have a constructor, which passes in all of the relevant information needed to perform the work
  - Also, they have a function compute() which is where the work is conducted
  - Recursive Tasks have a generic type that is the type their final compute should return
  - In order to start a parallel procedure, we create a RecursiveTask that we've defined and then give it to the ForkJoinPool using invoke(RecursiveTask)

RECURSIVE TASKS

```
public static final ForkJoinPool POOL = new ForkJoinPool();
static class SumTask extends RecursiveTask<Long>{
        long[] arr; int lo; int hi;
        public SumTask(long [] arr, int lo, int hi){
                this. arr = arr; this.lo = lo; this.hi = hi;
        }
        protected long compute() {
                long result = 0;
                for(int i = lo; i<hi; i++){
                        result += arr[i];
                }
                return result;
        }
public static long sum(long[] arr){
        SumTask task = new SumTask(arr,0,arr.length;
        POOL.invoke(task);
}
```

RECURSIVE TASKS

- **This implementation isn't in parallel**
  - We've created the pool and the task, but we're only calling it once, there aren't any other threads being made.
  - We need to utilize fork() and join() to create parallel threads.

RECURSIVE TASKS

- **This implementation isn't in parallel**
  - We've created the pool and the task, but we're only calling it once, there aren't any other threads being made.
  - We need to utilize fork() and join() to create parallel threads.
  - While we call POOL.invoke() to start the process, we use task.fork() to recursively start a new thread which runs compute() for a task.
  - When we call task.join(), we wait for that parallel process to finish with its own work

RECURSIVE TASKS

- **This implementation isn't in parallel**
  - We've created the pool and the task, but we're only calling it once, there aren't any other threads being made.
  - We need to utilize fork() and join() to create parallel threads.
  - While we call POOL.invoke() to start the process, we use task.fork() to recursively start a new thread which runs compute() for a task.
  - When we call task.join(), we wait for that parallel process to finish with its own work

RECURSIVE TASKS

- **Basic ideas for good parallel compute functions**

RECURSIVE TASKS

- **Basic ideas for good parallel compute functions**
  - When given a job, a RecursiveTask is also required to start other recursive tasks.

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.

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)

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!

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

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

NEXT CLASS

- How do we implement this concept explicitly in Java?

NEXT CLASS

- How do we implement this concept explicitly in Java?
- What types of problems can be parallelized easily?

NEXT CLASS

- How do we implement this concept explicitly in Java?
- What types of problems can be parallelized easily?
- Are there common “parallel operations” that can make solving this problem easier?

NEXT CLASS

- How do we implement this concept explicitly in Java?
- What types of problems can be parallelized easily?
- Are there common “parallel operations” that can make solving this problem easier?
- How much faster can we actually get with parallelism?