© 2024 - Dr. Basit Qureshi



CS435 Distributed systems

PARALLEL PROCESSING (A)

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

TOPICS

- Parallel computing
- Parallel programming
- Java Threads Library
- Parallel program using Shared Memory

© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

PARALLEL COMPUTING

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

So far, most or all of your study of computer science has assumed that **one thing happens at a time**

- This is sequential programming
- Everything is part of one sequence



Parallelism: Divide and Conquer ??

- Opportunities
 - Divide work among threads of execution
 - Faster runtimes
 - Thread programming
 - More throughput = speedup
 - Concurrent access to resources
- Challenges
 - Harder to write parallel code
 - Performance issues
 - Concurrency issues



Supercomputing

- Since 1960s
- First IBM UNIVAC_LARC in Lawrence Livermore National Laboratory
- LARC supported multiprocessing with two CPUs (called Computers) and an input/output (I/O) Processor (called the Processor).
- One addition operation tool about 4 microseconds



https://en.wikipedia.org/wiki/UNIVAC_LARC

Supercomputing

- The ILLIAC IV was the first massively parallel computer.
- Had 256 64-bit floating point units (FPUs).
- 4 central processing units (CPUs) were able to process 1 billion operations per seconds.
- Eventually had 16 processors due to cost escalation.
- 1976: Runs first successful application.
- Cost for one machine > 31 million USD (1972)
- Max 50 Floating Point Operations per Second(FLOPS)





ILLIAC IV Processing Unit and Control Unit https://en.wikipedia.org/wiki/ILLIAC_IV



G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick and R. A. Stokes, "The ILLIAC IV Computer," in *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746-757, Aug. 1968, doi: 10.1109/TC.1968.229158.

1980-2005: Era of Desktop Computers

- Availability of Uni-processor; cheaper computing!
- Uni-(micro) processors became exponentially faster at running sequential programs
- Traditional doubling of clock speeds every 18–24 months
- Need for parallelism declined
- 2005 -> Reached the limits !
 - Power wall: Increasing clock rate generates too much heat. Power consumption increases
 - Memory wall: Increasing clock rate generates larger gaps in memory-CPU speeds
 - Parallelism wall: Increasingly difficult to write parallel programs to keep the processor busy
 - Cooling constraints limit increases in microprocessor clock speeds



Intel 8080 processor: 1975, 4,500 transistors



Intel Pentium Pro, 1995, 5.5 million transistors

2005 onwards: Moores Law

- Make wires exponentially smaller
- System-on-Chip (SoC) Design: Multiple components on one Integrated Circuit
- 2007: First Dual core processors in market (Intel, AMD)
- 2008 onwards: Multiple-cores on one SoC
- 2010+, the level of parallelism on a single microprocessor now rivals the number of nodes in the most massively parallel supercomputers of the 1980s
- 2020+, extreme scale High Performance Cluster (HPC) systems are anticipated to have on the order of 100,000–1,000,000 sockets, with each socket containing between 100 and 1000 heterogeneous cores



Diagram of a generic dual-core processor



Intel Core 2 Duo, 2007



AMD Athlon X2 6400+, 2007

IBM Supercomputing timeline





1954 The Naval Ordnance Research Calculator helped forecast weather and performed

other complex

calculations.

operations per second.

1966 The IBM 360 and its

program.



1997 **Deep Blue** wins its match with chess successors helped power grandmaster NASA's Apollo Garry



2004 Kasparov.



2008

Built for

National

super-

computer

to reach

petaflop

speed.

in the world

Blue Gene ushers in a new era of highperformance computing as it helps biologists explore gene development.



2011 Watson Los Alamos beats human competitors Laboratory, on Jeopardy!, Roadrunner is the first

earning a million-dollar jackpot for charity.



2012 Sequoia, the thirdgeneration **Blue Gene** system, reaches speeds of 16.32

petaflops.



2018 Summit begins work at Oak Ridge National Laboratory; a sister machine, Sierra. launches at Lawrence Livermore National

Laboratory.

super-

energy

resources.



2020 IBM builds IBM helps Pangea III. launch the the world's COVID-19 most powerful High commercial Performance Computing Consortium computer, for Total to to research the accurately COVID-19 locate new virus and its



potential

cures.

1961 The IBM 7030 was capable of 2 million

© 2024 - Dr. Basit Qureshi

10

• Performance

- Supercomputing is measured in floating-point operations per second (FLOPS).
- Petaflops are a measure of a computer's processing speed equal to a thousand trillion flops.
- 1-petaflop computer system can perform one quadrillion (10¹⁵) flops.
- Fastest Computer in the world (2020)
 - Fugaku (Kobe-JAPAN)
 - Cores: 7,630,848
 - Max FLOPS: 442.01 PFLOPS or (537,212,000 Giga FLOPs)
 - Power: 30 MW
 - 24,710 U.S. households annually



https://www.fujitsu.com/global/about/innovation/fugaku/

What to do with multi-processors?

- Run multiple totally different programs at the same time
 - Already do that by time-slicing
- Do multiple things in ONE program
 - More difficult
 - Need to re-think writing parallel programs
 - Using multiple cores
 - Using Memory
 - Using I/O
 - Using appropraite structures to support parallelism
 - Writing parallel programs (Painful!)
 - Manage complexity, follow principles of parallelism
 - Manage concurrency

Parallel: Use extra computational resources to solve a problem faster



Concurrent: Correctly and efficiently manage access to shared resources



A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Dan Grossman, online notes, version Feb 2012

- Understanding Parallelism with Analogy
- A Serial Program:
 - One player dribbles and scores goal
- A parallel Program:
 - Use a team of 11 players. Each player has a role to play (Goal-keeper, Defender, Midfielder, Forward Striker) [Role decomposition]
 - Assign tasks to each set of players (corner, freekick, penalty taking, pressing, defending etc) [Task Decomposition]
 - Have extra help/resources e.g. substitutes [work pool]
 - Problem: Coordination!



- Understanding Concurrency with Analogy
- Concurrency:
 - Too many players on the pitch, but only ONE ball and TWO goal posts.
 - Need to manage the game within gameconstraints (90-minute time, half-time etc)
 - Avoid hand-balls, corners, penalty kicks etc



- In reality: Parallelism and concurrency are mixed
 - Common to use threads for both
 - If parallel computations need access to shared resources, then the concurrency needs to be managed
 - Threads, Processes, Mutual Exclusion, Shared memory, Inter-process-communication, etc
 - In this course, we will first cover parallelism and later Concurrency

- In reality: Parallelism and concurrency is everywhere
 - Operating Systems
 - Real time systems
 - MMPOG games
 - GPU based systems
 - AI/ML based systems
 - ChatGPT
 - Animated Movies: CGI Render Farms



https://sciencebehindpixar.org/pipeline/rendering

https://www.sabrepc.com/blog/Deep-Learning-and-AI/render-farms-for-cgi

• So what is a parallel computer?

- A computer with a CPU with 2-cores?
- or A computer with 2 or more CPU each with 2 or more cores?
- or 2 or more computers with 1 or more multi-core processors?
- or a computer with a GPU (1200+ cores)?
- Or multiple computers with many GPUs?
- Data Center?
- Cloud?

- What about Memory Parallelism?
 - Serial Bus connecting memory and processor
 - A shared memory Computer
 - Distributed Memory Computer



The Shared Memory Multiprocessor (SMP)

- All memory is placed into a single (physical) address space.
- Processors connected by some form of interconnection network.
- Single virtual address space across all of memory.
- Each processor can access all locations in memory.
- Processes must communicate in order to synchronize or exchange data using the memory space.



© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

PARALLEL PROGRAMMING

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

PARALLEL PROGRAMMING

- In recent times, the technology has converged around 3 programming environments:
 - **OpenMP**: simple language extension to C, C++ and Fortran to write parallel programs for shared memory computers (shared memory model)
 - MPI: A Message Passing Library used on clusters and other distributed memory computers (message passing model)
 - Java language features: support parallel programming on shared memory computers and standard class libraries supporting distributed computing (shared memory model AND message passing model)

PARALLEL PROGRAMMING

- To write a shared-memory parallel program, need new primitives from a programming language or library
 - Threads are ways to create and run multiple things at once.
 - Shared memory can be utilized by threads to <u>share/update</u> data.
 - <u>Synchronize</u> the execution of tasks/primitives using threads.

PROCESSES AND THREADS

- Process originates from operating systems.
 - A unit of resource allocation both for CPU time and for memory.
 - A process is represented by its code, data and the state of the machine registers.
 - The data of the process is divided into global variables and local variables, organized as a stack.
 - Generally, each process in an operating system has its <u>own address space</u> and some special action must be taken to allow different processes to access shared data.



PROCESSES AND THREADS

• Thread:

- The traditional OS process has a single thread of control it has **no** internal concurrency.
- Not suitable for shared memory multiprocessors
- Modern operating systems permit an operating system **process** to have multiple threads of control.
- How? Make multiple stacks of local/global variables, one for each thread.



PARALLEL PROGRAM

- A sequential program has a single thread of control
- A parallel program has multiple threads of control
 - Can perform multiple computations in parallel
 - Can control multiple simultaneous external activities
 - Threads from the same process share memory (data and code).
 - They can communicate easily, but it's dangerous if you don't protect your variables correctly.

• Java SE 7 introduced the Fork/Join framework. It was designed to make divide-and-conquer algorithms easy to parallelize

java.lang.Thread

- Operations to create and initialize basic threads and control their execution
- The Java Virtual Machine
 - Executes as a **process** under any operating system
 - Supports **multiple threads**. Each Java thread has its own local variables organized as a stack and threads can access shared variables.



- A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.
- Thread class executes instructions from its method run().
- The actual code executed depends on the implementation provided for **run()** in a derived class.

```
class MyThread extends Thread {
    public void run() {
        //.....
    }
}
```

Creating a thread object: Thread a = new MyThread();

Since Java does not permit multiple inheritance, it is sometimes more convenient to implement the **run()** method in a class not derived from Thread but from the interface **Runnable**



- So, there are two ways to create a basic thread in Java:
 - Implement the Runnable interface (java.lang.Runnable)
 - Extend the Thread class (java.lang.Thread)
 - Allocation and construction of a Thread object do not cause the thread to run.
 - To get a new thread running:
 - 1. Define a subclass C of java.lang.Thread, overriding run
 - 2. Create an object of class C
 - 3. Call that object's start method
 - Not **run**, which would just be a normal method call
 - start sets off a new thread, using run as its "main"

PARALLEL PROGRAM IN JAVA thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:



thread alive states in Java

Once started, an **alive** thread has a number of substates :



© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

SHARED MEMORY IN JAVA: A SIMPLE JAVA PROGRAM

The following set of slides are embedded and are based on "A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency", Dan Grossman, online notes, version Feb 2012

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

The following program starts off with one thread and created 3 threads. Look at the output. What gives??

```
class C extends java.lang.Thread {
     int i;
     C(int i) \{ this.i = i; \}
     public void run() {
           System.out.println("Thread " + i + " says hi");
           System.out.println("Thread " + i + " says bye");
class M {
     public static void main(String[] args) {
           for(int i=1; i <= 5; ++i) {</pre>
                 C c = new C(i);
                 c.start();
                                                   Thread 4 says hi
                                                                       Thread 5 says hi
                                                   Thread 5 says hi
                                                                       Thread 1 says hi
                                                   Thread 2 says hi
                                                                       Thread 3 says hi
                                                   Thread 3 says hi
                                                                       Thread 2 says hi
                                                   Thread 1 says hi
                                                                       Thread 4 says hi
                                                   Thread 4 says bye
                                                                       Thread 3 says bye
                                                   Thread 5 says bye
                                                                       Thread 1 says bye
                                                   Thread 2 says bye
                                                                       Thread 5 says bye
                                                   Thread 3 says bye
                                                                       Thread 4 says bye
                                                   Thread 1 says bye
                                                                       Thread 2 says bye
  © 2024 - Dr. Basit Qureshi
                                                   BUILD SUCCESSFUL (to
                                                                       BUILD SUCCESSFUL (1
```

```
Thread 2 says hi
Thread 5 says hi
Thread 5 says hi
Thread 4 says hi
Thread 5 says bye
Thread 2 says bye
Thread 1 says hi
Thread 4 says bye
Thread 1 says bye
Thread 3 says hi
Thread 3 says bye
BUILD SUCCESSFUL (t
```

JAVA THREAD BASICS

We will first learn some basics built into Java via the provided java.lang.Thread package

• We will learn a better library for parallel programming

To get a new thread running:

- 1. Define a subclass C of java.lang.Thread,
- 2. Override the **run** method
- 3. Create an object of class C
- 4. Call that object's **start** method

start sets off a new thread, using run as its "main"

What if we instead called the **run** method of **C**?

• Just a normal method call in the current thread

PARALLELISM EXAMPLE: SUM AN ARRAY

Have 4 threads simultaneously sum 1/4 of the array

Approach:

- Create 4 thread objects, each given a portion of the work
- Call start() on each thread object to actually run it in parallel
- Somehow 'wait' for threads to finish
- Add together their 4 answers for the final result



Warning: This is the inferior first approach, do not do this

CREATING THE THREAD SUBCLASS

```
class SumThread extends java.lang.Thread {
  int lo; // arguments
  int hi;
  int[] arr;
  int ans = 0; // result
                                              We will ignore handling the case
                                              where:
  SumThread(int[] a, int l, int h) {
    lo=l; hi=h; arr=a;
                                                    arr.length % 4 != 0
  public void run() { //override must have this type
    for(int i=lo; i < hi; i++)</pre>
      ans += arr[i];
```

Because we override a no-arguments/no-result run, we use fields to communicate data across threads

CREATING THE THREADS WRONGLY

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ ... } // override
}
```

```
int sum(int[] arr){ // can be a static method
int len = arr.length;
int ans = 0;
SumThread[] ts = new SumThread[4];
for(int i=0; i < 4; i++) // do parallel computations
ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
for(int i=0; i < 4; i++) // combine results
ans += ts[i].ans;
return ans;
```

CREATING THE THREADS WRONGLY

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ ... } // override
}
```

```
int sum(int[] arr){ // can be a static method
int len = arr.length;
int ans = 0;
SumThread[] ts = new SumThread[4];
for(int i=0; i < 4; i++){ // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start(); // start not run
```

```
for(int i=0; i < 4; i++) /
    ans += ts[i].ans;
return ans;</pre>
```

We start the threads and then assume they finish right away!!! Concurrency??

JOIN: THE 'WAIT FOR THREAD' METHOD

The **Thread** class defines various methods that provide primitive operations you could not implement on your own

• For example: **start**, which calls **run** in a new thread

The join method is another such method, essential for coordination in this kind of computation

- Caller blocks until/unless the receiver is done executing (meaning its **run** method returns after its execution)
- Without join, we would have a 'race condition' on ts[i].ans in which the variable is read/written simultaneously

This style of parallel programming is called **fork/join**"

- If we write in this style, we avoid many concurrency issues
- But certainly not all of them

THIRD ATTEMPT: CORRECT IN SPIRIT

```
int sum(int[] arr) { // can be a static method
   int len = arr.length;
   int ans = 0;
   SumThread[] ts = new SumThread[4];
   for(int i=0; i < 4; i++){// do parallel computations</pre>
     ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
     ts[i].start();
   for(int i=0; i < 4; i++) { // combine results</pre>
ts[i].join(); // wait for helper to finish! 
     ans += ts[i].ans;
   return ans;
```

Note that there is no guarantee that ts[0] finishes before ts[1]

- Completion order is nondeterministic
- Not a concern as our threads do the same amount of work

WHERE IS THE SHARED MEMORY?

Fork-join programs tend not to require [thankfully] a lot of focus on sharing memory among threads

• But in languages like Java, there is memory being shared

In our example:

- lo, hi, arr fields written by "main" thread, read by helper thread
- ans field written by helper thread, read by "main" thread

When using shared memory, the challenge and absolute requirement is to avoid race conditions

- While studying parallelism, we'll stick with join
- With concurrency, we'll learn other ways to synchronize

© 2024 - Dr. Basit Qureshi

© 2024 - Dr. Basit Qureshi

CS435 Distributed systems

BETTER ALGORITHMS: PARALLEL ARRAY SUM

Keep in mind that Java was first released in 1995

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

A POOR APPROACH: REASONS

Our current array sum code is a poor usage of parallelism for several reasons

- 1. Code should be **reusable and efficient** across platforms
 - "Forward-portable" as core count grows
 - At the very least, we should parameterize the number of threads used by the algorithm

```
int sum(int[] arr, int numThreads){
    ... // note: shows idea, but has integer-division bug
    int subLen = arr.length / numThreads;
    SumThread[] ts = new SumThread[numThreads];
    for(int i=0; i < numThreads; i++){
      ts[i] = new SumThread(arr,i*subLen,(i+1)*subLen);
      ts[i].start();
    }
    for(int i=0; i < numThreads; i++) {
        ...
    }
    ...
}</pre>
```

A POOR APPROACH: REASONS

Our current array sum code is a poor usage of parallelism for several reasons

- 2. We want to use only the processors "available now"
 - Not used by other programs or threads in your program
 - Maybe caller is also using parallelism
 - Available cores can change even while your threads run
 - If 3 processors available and 3 threads would take time X, creating 4 threads can have worst-case time of 1.5X

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numThreads){
...
}
```

A POOR APPROACH: REASONS

Our current array sum code is a poor usage of parallelism for several reasons

- 3. Though unlikely for **sum**, subproblems may take significantly different amounts of time
 - Example: Apply method **f** to every array element, but maybe **f** is much slower for some data items
 - Example: Determine if a large integer is prime?
 - If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup
 - Example of a load imbalance

A BETTER APPROACH: COUNTERINTUITIVE

Although counter-intuitive, the better solution is to use a lot more threads beyond the number of processors



- **1.** Forward-Portable: Lots of helpers each doing small work
- 2. Processors Available: Hand out "work chunks" as you go
 - If 3 processors available and have 100 threads, worst-case extra time is < 3% (if we ignore constant factors and load imbalance)
- 3. Load Imbalance: Problem "disappears"
 - Try to ensure that slow threads are scheduled early
 - Variation likely small if pieces of work are also small

BUT DO NOT BE NAÏVE

This approach does not provide a free lunch:

Assume we create 1 thread to process every N elements

```
int sum(int[] arr, int N){
    ...
    // How many pieces of size N do we have?
    int numThreads = arr.length / N;
    SumThread[] ts = new SumThread[numThreads];
    ...
}
```

Combining results will require **arr.length** / **N** additions

- As **N** increases, this becomes linear in size of array
- Previously we only had 4 pieces, $\Theta(1)$ to combine

In the extreme, suppose we create one thread per element

• Using a loop to combine the results requires N iterations

A BETTER IDEA: DIVIDE-AND-CONQUER



Straightforward to implement

Use parallelism for the recursive calls

- Halve and make new thread until size is at some cutoff
- Combine answers in pairs as we return

This starts small but grows threads to fit the problem

DIVIDE-AND-CONQUER

```
public void run() { // override
  if(hi - lo < SEQUENTIAL CUTOFF)</pre>
      for(int i=lo; i < hi; i++)</pre>
       ans += arr[i];
  else {
    SumThread left = new SumThread(arr, lo, (hi+lo)/2);
    SumThread right= new SumThread(arr, (hi+lo)/2,hi);
    left.start();
  right.start();
    left.join(); // don't move this up a line - why?
    right.join();
    ans = left.ans + right.ans;
int sum(int[] arr) {
  SumThread t = new SumThread(arr,0,arr.length);
  t.run();
  return t.ans;
```

DIVIDE-AND-CONQUER REALLY WORKS

The key is to parallelize the result-combining

- With *enough* processors, total time is the tree height: $O(\log n)$
- This is optimal and exponentially faster than sequential O(n)
- But the reality is that we usually have P < O(n) processors



Still, we will write our parallel algorithms in this style

- Relies on operations being associative (as with +)
- But will use a special library engineered for this style
- It takes care of scheduling the computation well © 2024 Dr. Basit Qureshi

BEING REALISTIC

In theory, you can divide down to single elements and then do all your result-combining in parallel and get optimal speedup

In practice, creating all those threads and communicating amongst them swamps the savings,

To gain better efficiency:

- Use a *sequential cutoff*, typically around 500-1000
 - Eliminates *almost all* of the recursive thread creation because it eliminates the bottom levels of the tree
 - This is exactly like quicksort switching to insertion sort for small sub problems, but even more important here
- Be clever and do not create unneeded threads
 - When creating a thread, you are already in another thread
 - Why not use the current thread to do half the work?
 - Cuts the number of threads created by another 2x

HALVING THE NUMBER OF THREADS

```
// wasteful: don't
SumThread left = ...
SumThread right = ...
```

```
// create two threads
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```

```
// better: do
SumThread left = ...
SumThread right = ...
```

```
// order of next 4 lines
// essential - why?
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

If a *language* had built-in support for fork-join parallelism, we would expect this hand-optimization to be unnecessary

But the *library* we are using expects you to do it yourself

- And the difference is surprisingly substantial
- But no difference in theory

ILLUSTRATION OF FEWER THREADS





LIMITS OF THE JAVA THREAD LIBRARY

Even with all this care, Java's threads are too *heavyweight*

- Constant factors, especially space overhead
- Creating 20,000 Java threads just a bad idea

The ForkJoin Framework is designed/engineered to meet the needs of divide-and-conquer forkjoin parallelism

- Included in the Java 7 standard libraries
- Also available as a downloaded .jar file for Java 6
- Section will discuss some pragmatics/logistics
- Similar libraries available for other languages
 - C/C++: Cilk, Intel's Thread Building Blocks
 - C#: Task Parallel Library
- Library implementation is an advanced topic

DIFFERENT TERMS / SAME BASIC IDEAS 57

To use the ForkJoin Framework:

A little standard set-up code (e.g., create a ForkJoinPool)

The Fundamental Differences:

Don't subclass Thread	Do subclass RecursiveTask <v></v>
Don't override run	Do override compute
Do not use an ans field	Do return a v from compute
Do not call start	Do call fork
Do not just call join	Do call join which returns answer
Do not call run to hand-optimize	Do call compute to hand-optimize
Do not have a topmost call to run	Do create a pool and call invoke

See the Dan Grossman's web page for <u>"A Beginner's Introduction to the ForkJoin Framework"</u>

http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/grossmanSP AC_forkJoinFramework.html

FINAL VERSION IN FORKJOIN FRAMEWORK

```
class SumArray extends RecursiveTask<Integer> {
 int lo; int hi; int[] arr; // arguments
 SumArray(int[] a, int l, int h) { ... }
 protected Integer compute() {// return answer
    if(hi - lo < SEQUENTIAL CUTOFF) {</pre>
      int ans = 0;
      for(int i=lo; i < hi; i++)</pre>
        ans += arr[i];
      return ans;
    }
    else {
      SumArray left = new SumArray(arr,lo,(hi+lo)/2);
      SumArray right= new SumArray(arr, (hi+lo)/2,hi);
      left.fork();
      int rightAns = right.compute();
      int leftAns = left.join();
      return leftAns + rightAns;
   }
static final ForkJoinPool pool = new ForkJoinPool();
int sum(int[] arr){
 return pool.invoke(new SumArray(arr,0,arr.length));
```

FORKJOIN VERSION VS THREADS VERSION

```
class SumArray extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // arguments
  SumArray(int[] a, int l, int h) { ... }
```

```
protected Integer compute(){// return answer
if(hi - lo < SEQUENTIAL_CUTOFF) {
    int ans = 0;
    for(int i=lo; i < hi; i++)
        ans += arr[i];
    return ans;
    }
    else {
        SumArray left = new SumArray(arr,lo,(hi+lo)/2);
        SumArray right= new SumArray(arr,(hi+lo)/2,hi);
        left.fork();
        int rightAns = right.compute();
        int leftAns = left.join();
        return leftAns + rightAns;
    }
}
```

```
static final ForkJoinPool pool = new ForkJoinPool();
```

}

}

```
int sum(int[] arr){
    return pool.invoke(new SumArray(arr,0,arr.length));
```

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr;//fields to know what to do
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { ... }
  public void run() {
    if(hi - lo < SEQUENTIAL CUTOFF)</pre>
      for(int i=lo; i < hi; i++)</pre>
        ans += arr[i];
    }
   else { // create 2 threads, each will do \frac{1}{2} the work
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr, (hi+lo)/2, hi);
      left.start();
      right.start();
      left.join(); // don't move this up a line - why?
      right.join();
      ans = left.ans + right.ans;
  }
class C {
 static int sum(int[] arr){
   SumThread t = new SumThread(arr,0,arr.length);
   t.run(); // only creates one thread
   return t.ans;
```

GETTING GOOD RESULTS WITH FORKJOIN

Sequential threshold

• Library documentation recommends doing approximately 100-5000 basic operations in each "piece" of your algorithm

Library needs to "warm up"

- May see slow results before the Java virtual machine re-optimizes the library internals
- When evaluating speed, loop computations to see the "long-term benefit" after these optimizations have occurred

Wait until your computer has more processors

- Seriously, overhead may dominate at 4 processors
- But parallel programming becoming much more important

Beware memory-hierarchy issues

• Will not focus on but can be crucial for parallel performance

SUMMARY

- Writing parallel programs increases the processing speed, but does it??!!
- Threads implement parallelism using shared memory and multi-processors on the same machine.
- Distributed System: Extend the concept to Clusters housing multiple machines
- Java Threads in JDK 6
- ForkJoin Library in JDK 7