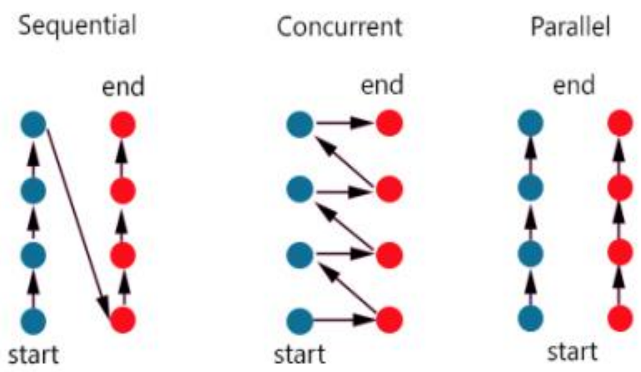
Parallelism, Concurrency and Architecture

CSC2002S

Parallelism and Concurrency



Glossary of Terms

https://vula.uct.ac.za/access/content/group/57872ff9-5c75-4def-b501-80c784752dcf/Parallel_Concurrent%20Programming/Parallel%20Java%20Glossary.pdf

Brief Intro

- Living in multicore era i.e. every computing device generally has more than one core.
- Core: programming element that can operate independently/ run separately from other cores
    - Can run separate algos on separate data from other cores.

Paradigm Shift

- Always assumed one thing happens at a time.
- Previously, made sequential programs: one thing happens at a time
    - Everything part of one sequence
    - Execution starts at the beginning of *"main"* and process one (assignment/arithmetic operations/ call /return opn) at a time.
- Sequential programming – now shift to ||ism
    - Take input, break it up into chunks, send to diff cores so that they can be operated on in ||.
    - Each separate piece still operated on in sequential fashion.
    - Order depends on how and when chunks came to cores.
        - Thus, no longer able to predict order in which operations from diff chunks of work will occur.
- By trying to control order of operations, can end up hindering performance => necessary to give up control.
- Non-determinism: execution of same code on same dataset performs differently.
    - Order of execution can change between runs of program
- Removing one-thing-at-a-time assumption creates complications.

  - Now have *multiple threads of execution* – these need to co-ordinate and work together to complete a task or (at least) not get in each other’s way while doing separate things.
  - Best to avoid multi-threaded programming if you can, since it gets quite complex and difficult.

ANALOGY
Sequential program = having one cook who does each step of a recipe in order. They finish one step before starting the next. **Having more cooks may be better but requires extra co-ordination, and there are limited resources. Thus, multiple cooks present efficiency opportunities but also complicates the process of making a meal.**

- Moore’s Law “died” in 2005 – increasing clock rate became infeasible without generating too much heat and relative cost of memory accesses can become too high for faster processors to help.
  - Solution: have more processors on the same chip.
- For an individual program to run faster with one core, it’ll need to do more than one thing at once – hence, multithreaded programming.

NECESSITY OF NON-SEQUENTIAL PROGRAMMING

Computational demand

- Problems that require so much computational resources, that for one computer to finish them would take years/centuries.
  - Rather use multiple computers working together to solve them.

Timeslicing

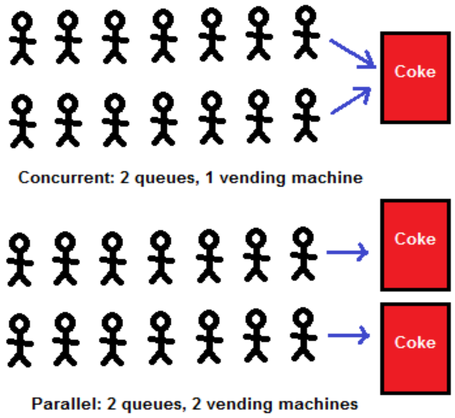
- Systems in which key function is to handle multiple things happening at once.
- More than one prog runs at a time in an OS with timeslicing
  - Have a single processor means only one program actually executes at a time.
  - Threads used to keep track of all running programs and let them take turns.
  - If done fast enough, looks like many programs running at once.

MULTIPLE PROCESSORS

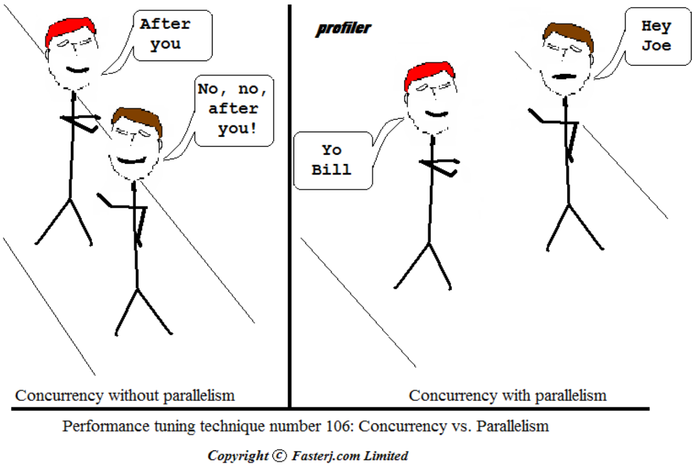
- Have two options:
  - Run multiple totally different programs at the same time

        - Time-slicing allows us to almost achieve this
    - Do multiple things at once in one program.
        - For a program to run faster using more cores, it needs to do more than one thing at once.
- Have x cores – how to take problem and assign it to cores?
    - Task decomposition: assign diff problems to each core; diff programs /apps for each core with its own indep data
        - Difficult when have large # of cores.
    - Data decomposition: all cores solve single problem; same code on all cores; operate on diff pieces of data and combine them at the end (somehow).
    - Can have a combo of these two
- Requires one to rethink everything from asymptotic complexity (difficulty of problems being solved) to how to implement data-structure opns.
- Contention of resources
    - Because of this, just because have y # of cores, doesn't mean you'll work y times as faster.

Parallelism vs concurrency


Coke
Concurrent: 2 queues, 1 vending machine
Coke
Coke
Parallel: 2 queues, 2 vending machines

- ***Distinction*** between parallelism and concurrency.


After you
No, no, after you!
profiler
Yo Bill
Hey Joe
Concurrency without parallelism
Concurrency with parallelism
Performance tuning technique number 106: Concurrency vs. Parallelism
Copyright © Fasterj.com Limited

CONCURRENCY
Two queues
One bowl
PARALLELISM
Two queues
Two bowls

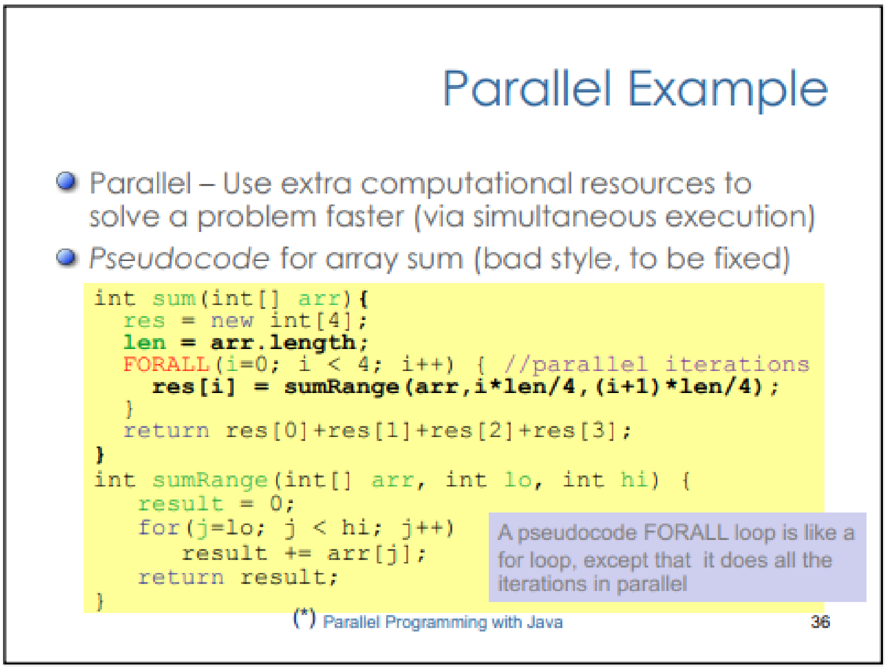
- In CS, use hybrid of data and task decomposition, but data decomp is still more common.

PARALLEL PROGRAMMING

- Using additional resources to produce answers faster
- Use extra *computational resources* to solve a problem faster via *simultaneous execution*.
- When designing and analysing a parallel algo, leave # of processors as variable P.
- Circumstances for use of || program:
  - Reduce time to solve problem
  - Have hard cutoff on performance – want program to run in real time (e.g. for games).

CS Example	Cooking Analogy
Summing up all numbers in an array. **Sequential algo can't do better than O(n) time.** **If have 4 processors, can produce result almost 4x faster – each processor adds ¼ of elements and then add these 4 results together with 3 more additions. O(n/4) is still technically O(n) but sometimes leading constants matter.**	**Use extra cooks/utensils etc to get a large meal finished in less time.** **E.g. if you have a lot of carrots to slice, having more hands and knives will help, but at some point there will be too many people helping, and this leads to inefficiencies (because more people = more communicating and co-ordinating).**

An example:



Parallel Example

- Parallel – Use extra computational resources to solve a problem faster (via simultaneous execution)
- *Pseudocode* for array sum (bad style, to be fixed)

```
int sum(int[] arr){
  res = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; i++) { //parallel iterations
    res[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return res[0]+res[1]+res[2]+res[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

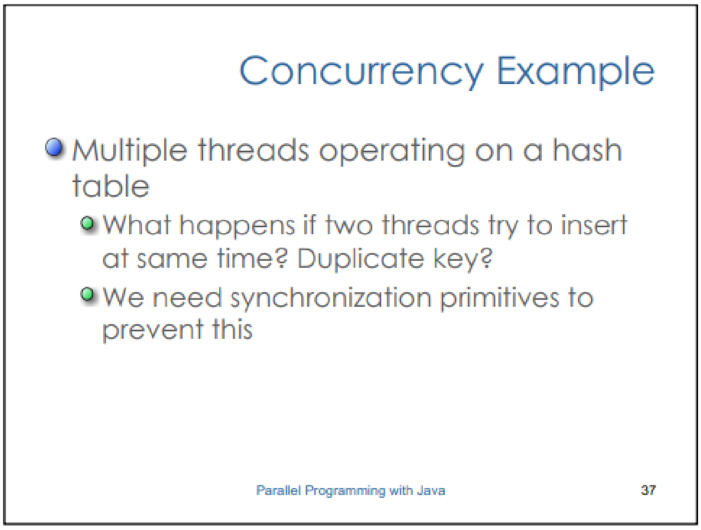
A pseudocode FORALL loop is like a for loop, except that it does all the iterations in parallel

(*) Parallel Programming with Java 36

CONCURRENT PROGRAMMING

- Correctly and efficiently controlling access by multiple threads to shared resources.

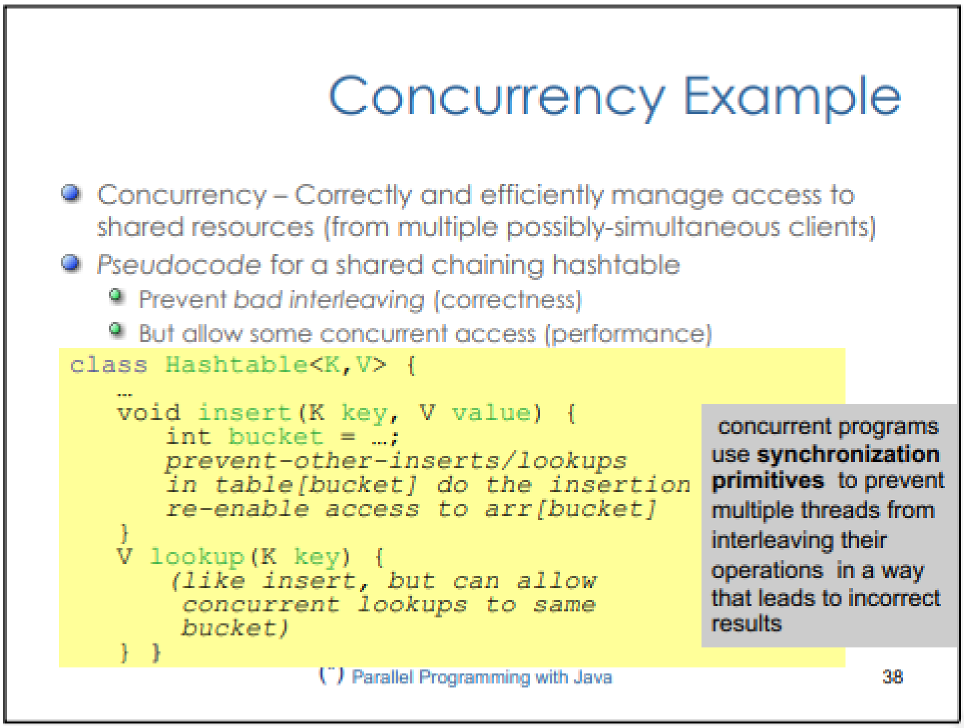
An example:



Concurrency Example

- Multiple threads operating on a hash table
  - What happens if two threads try to insert at same time? Duplicate key?
  - We need synchronization primitives to prevent this

Parallel Programming with Java 37



Concurrency Example

- Concurrency – Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)
- *Pseudocode* for a shared chaining hashtable
  - Prevent *bad interleaving* (correctness)
  - But allow some concurrent access (performance)

```
class Hashtable<K,V> {
   …
   void insert(K key, V value) {
       int bucket = …;
       prevent-other-inserts/lookups
       in table[bucket] do the insertion
       re-enable access to arr[bucket]
   }
   V lookup(K key) {
       (like insert, but can allow
        concurrent lookups to same
        bucket)
   } }
```

concurrent programs use **synchronization primitives** to prevent multiple threads from interleaving their operations in a way that leads to incorrect results

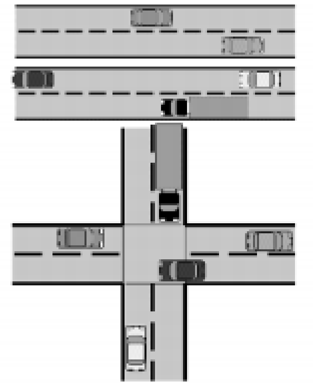
Parallel Programming with Java 38

CS Example	Cooking Analogy

Suppose have a dictionary implemented as a hash table with insert, lookup and delete opns. Suppose inserting an item already in tbl is supposed to update key to map to newly inserted value.
 	
Suppose different threads use *the same* hashtable, possibly at the same time. Suppose two threads try to insert the same key at the same time.
What happens – in sequential code, it's possible that the same key might end up in the tbl twice. Problematic? Yes, because a subsequent delete with that key might only remove one of those keys, leaving the key in the dictionary.
	Have an oven i.e. a shared resource and multiple chefs. Nothing can be put in the oven until the oven is empty. If the oven isn't empty, have to keep checking till it is. Need to learn to check the oven if oven is empty and put a cake in without another chef doing something in the meantime e.g. one chef holds oven door open and wants to put cake in but while oven door is open, other chef puts their lasagne in
	In terms of code, one might be tempted to write this to check for an empty oven:
	
	while(true) {
	
	if(ovenIsEmpty()) {
	
	putCasseroleInOven();
	
	break;}
	
	}
	
	This code (and code like this) breaks if two threads run it at the same time – might both see an empty oven and then both put a cake in. This is the *primary complication* in concurrent programming.

- Concurrent programs prevent problems like those in the CS example above by *interleaving their operations* in a way that leads to correct results.

Concurrency problems



- Car represents program executing on core.
- Top: (desired) cars travelling at same speed, no crashes
- Bottom: depending on speed/order cars enter crossroad, this can cause crashes i.e. problems.
- There is a certain freedom of order of execution when cores execute prog
- Race condition: bug in program where output and/result of process is unexpectedly and critically dependant on the relative sequence/timing of other events.
    - i.e. Lack of order of execution gives problems.
    - Events race each other to influence output first.

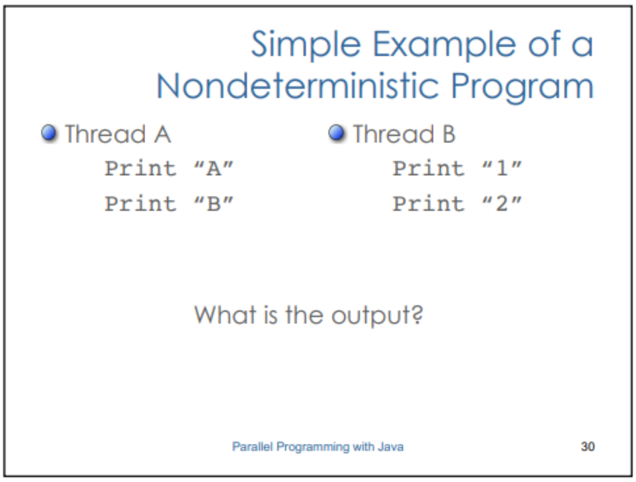
COMPARING PARALLELISM AND CONCURRENCY

- Is the problem one of using extra resources effectively or preventing a bad interleaving of opns from different threads?
- Distinction between parallelism and concurrency is *not* absolute – progs can have aspects of both.
    - E.g. Suppose you had a huge array of values you wanted to insert into a hash table. From the perspective of dividing up the insertions among multiple threads, this is about parallelism. From the perspective of coordinating access to the hash table, this is about concurrency.
- There's some connection:
    - Common to use threads for both
    - Parallel computations sometimes need access to shared resources, this then means concurrency needs to be managed.

Determinism and Non-determinism

- Deterministic:
    - Given certain set of inputs, can predict what outputs will get; always same output on every execution of prog.

    - In sequential progs, instructions executed in fixed order determined by prog and its input.
    - Execution of one procedure doesn't overlap in time with another.
- Non-deterministic:
    - Due to race conds, get diff outputs on each run (for || or concur prgs)
    - In concurrent programs, computational activites may overlap in time and subprogram executions describing these activities proceed concurrently
    - Not possible to tell, by looking at the program, what'll happen when it executes.
    - Example:



Simple Example of a Nondeterministic Program

- Thread A

```
Print "A"
Print "B"
```

- Thread B

```
Print "1"
Print "2"
```

What is the output?

Parallel Programming with Java 30

    - Possible outputs: AB12; A12B; A1B2;12AB;1AB2;1A2B
- With || and concurrent progs, we have a non-deterministic order of exec but *seek deterministic outputs*.
    - **Want** predictability in terms of results of progs.

Summary

Key idea

- Depending on prob – have situation where need to manage contention for resources and need to have certain # of tasks decomposed amongst those resources.
    - There's a sweet spot for # of cores used for problem.

- Change major assumption
- Remove the sequential assumption – creates opportunities and challenges.
    - Programming – divide work among threads of execution and co-ordinate (synchronise) them.
    - Algorithms – more throughput: work done per unit wall-clock time = speedup.

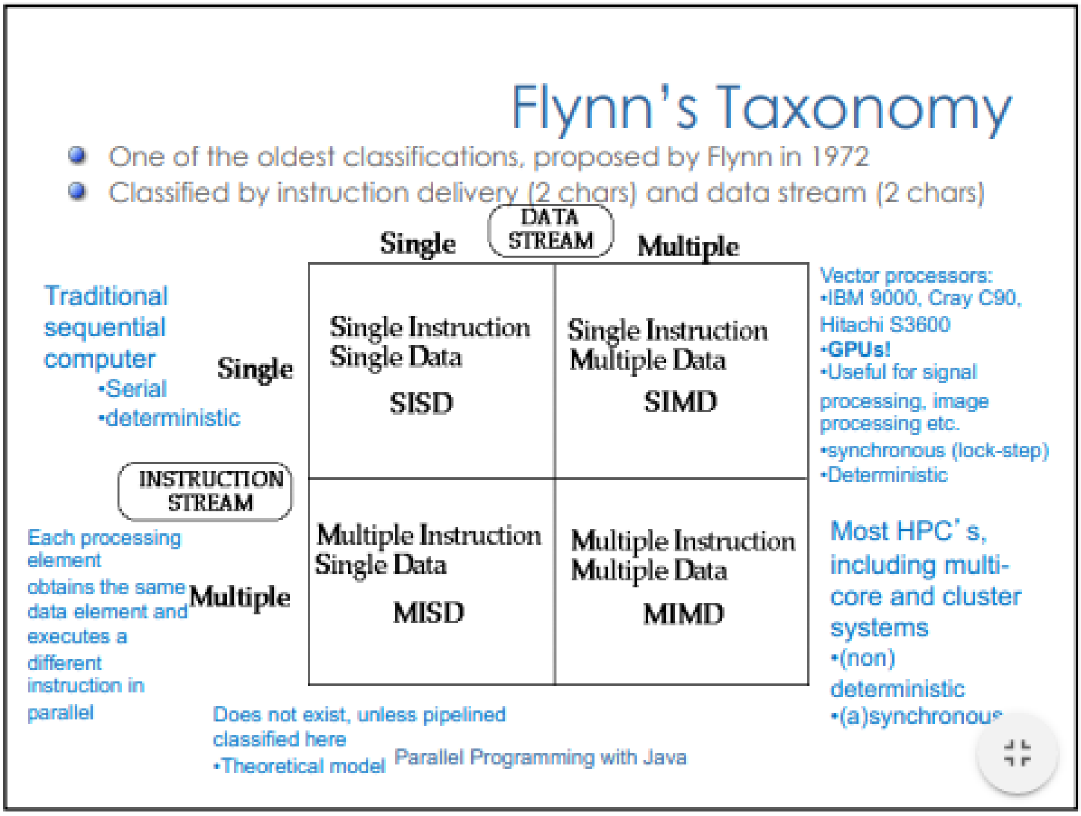
        - More organisation required.
- Data structures – may need to support concurrent access i.e. multiple threads operating on data at the same time.

Hardware

- Hardware impacts || computing, most importantly in terms of memory access.
- Parallel computer can be many things – a cluster (multiple processors on multiple separate comps working together on a prob); a single computer with multiple internal processors; accelerators (GPU's); a combo of these etc.

Flynn's Taxonomy

- Classified by instruction delivery and data stream



Flynn's Taxonomy

- One of the oldest classifications, proposed by Flynn in 1972
- Classified by instruction delivery (2 chars) and data stream (2 chars)

INSTRUCTION STREAM \ DATA STREAM	Single	Multiple
Single	Single Instruction Single Data SISD	Single Instruction Multiple Data SIMD
Multiple	Multiple Instruction Single Data MISD	Multiple Instruction Multiple Data MIMD

Traditional sequential computer
•Serial
•deterministic

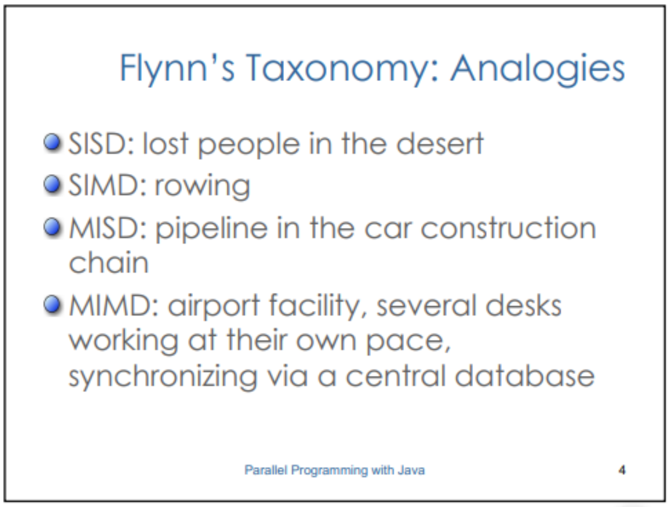
Vector processors:
•IBM 9000, Cray C90, Hitachi S3600
•**GPUs!**
•Useful for signal processing, image processing etc.
•synchronous (lock-step)
•Deterministic

Each processing element obtains the same data element and executes a different instruction in parallel

Does not exist, unless pipelined classified here
•Theoretical model

Most HPC's, including multi-core and cluster systems
•(non) deterministic
•(a)synchronous

Parallel Programming with Java



Flynn's Taxonomy: Analogies

- SISD: lost people in the desert
- SIMD: rowing
- MISD: pipeline in the car construction chain
- MIMD: airport facility, several desks working at their own pace, synchronizing via a central database

Parallel Programming with Java 4

- SIMD – vector machine e.g. GPUs.
    - Operates on time step – on each time step, more data comes in and opns done on data.
- MIMD (*course focus*): separate pieces of data comes in but dealt with in diff ways
    - Could have same algo being run on diff instructions but they're at diff pts in their execution.

Thread confusion

- Software thread:
    - Unit of execution with its own program and state
    - Managed by OS
- Hardware (Hyper)Thread
    - Support at h-ware lvl for threading
        - Have individual core with additional support to run multiple threads on that core.
        - Support swapping on/off threads quickly.
    - Usually two logical threads per physical core
- Usually have much more s-ware threads than h-ware

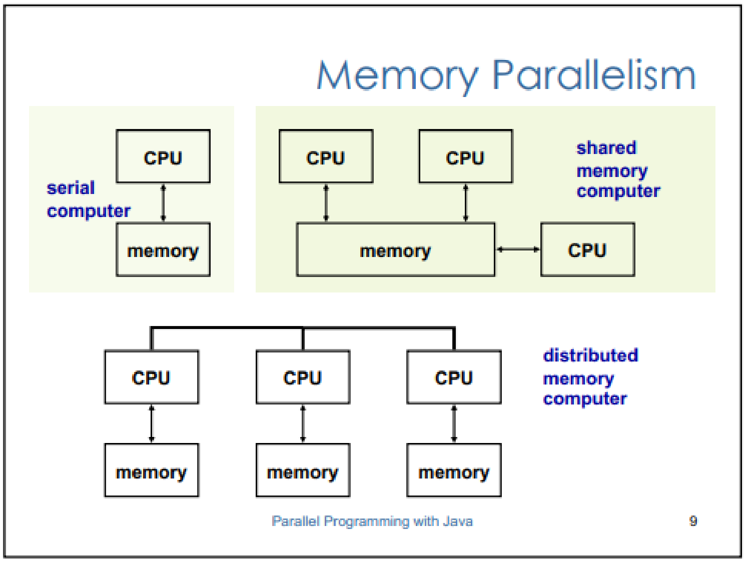
Types of Parallelism

Models allows us to manage complexity of h-ware through lvls of abstraction.

1. Convert parallelism (CPU parallelism)
    a. Multicore + GPU's
    b. Mostly h-ware managed (hidden on microprocessor)
    c. Fine-grained quantities of work/time
2. Overt parallelism (memory parallelism)
    a. 3 kinds:
        i. Shared Memory Multiprocessor Systems (SMMP)
            1. Shared memory = pooled mem that tasks access
            2. Pool can be spit across multiple comps
                a. Memory area located on different machines
                b. Software managed.
            3. Blackboard
        ii. Message Passing Multicomputer
            1. Contrast to SMMP
            2. Individual threads pass private msgs to designated other thread
        iii. Distributed Shared Memory
    b. S-ware managed
    c. Coarse-grained: The time chunks assigned to threads are larger.

MEMORY PARALLELISM

- Memory models (like image below) don't dictate programming mods.
  - But there are cases of better compatibility.
    - Shared Memory Computer is more compatible with SMMP
    - Distributed Memory Computer is more compatible with message passing



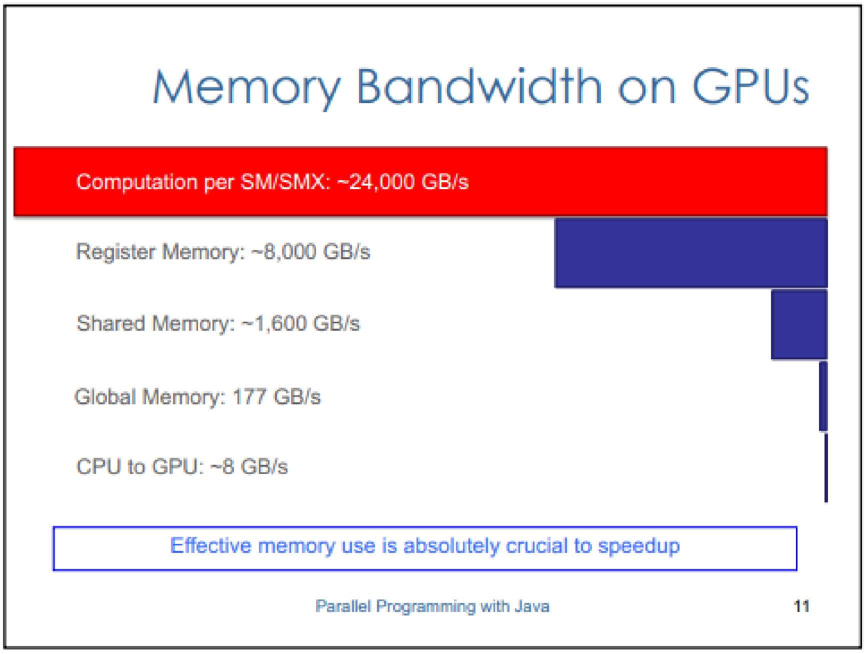
Memory Parallelism

Parallel Programming with Java

9

MEMORY HIERARCHY

- The most important factor that affects || computing performance is how memory is arranged in the memory hierarchy.
- Not all memory is equal
  - Registers < Caches < Main Memory (RAM) < Disk
  - All store diff amounts
- SM/SMX is how much data must be sent to keep processor busy
  - Can get situations where increasing memory requirements by 1KB will lead to flooding/overloading one of the lvls of the hierarchy and thus leading to a steep drop in performance.
- May seem impossible to keep e.g. GPU processor busy BUT – can use algos that reuse the same data items in multiple computations
  - Thus, will occupy/use full computational power of GPU.
- Feeding the processor cores with enough data is crucial
  - i.e. effective memory use is absolutely crucial to speedup



Memory Bandwidth on GPUs

Computation per SM/SMX: ~24,000 GB/s

Register Memory: ~8,000 GB/s

Shared Memory: ~1,600 GB/s

Global Memory: 177 GB/s

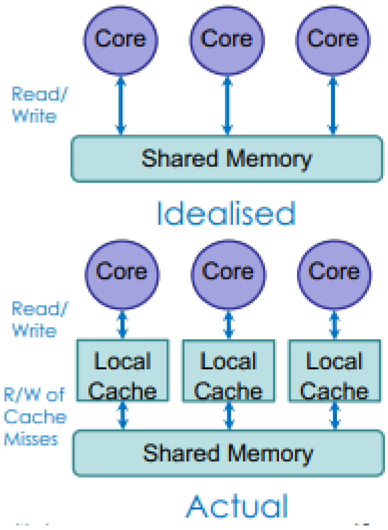
CPU to GPU: ~8 GB/s

Effective memory use is absolutely crucial to speedup

Parallel Programming with Java 11

Shared memory

- NB: talking about physical arrangement of mem in h-ware – this is a base layer on top of which can put a programming model.



ADVANTAGES

- Convenient to share data
- Easier to program

    - Allows parallel apps to be developed incrementally
    - Supports fine-grained communication in a cost-effective manner
        - Fine-grained refers to amount of data here.
        - Fine grained data communication can go down to level of a single byte and put that into/draw from shared mem

DISADVANTAGES

- Overhead through managing memory consistency - – shared memory MUST be left in the correct state before and after mem access
    - Cache is private but main memory is shared
    - Can lead to inconsistencies if data in cache isn't updated to or from shared memory
    - Must be done for performance reasons
- Process of core -> local cache -> shared memory:
    - Core may read and write to local cache – this may/may not percolate down to shared mem.
        - Can lead to stale cache i.e. inconsistent memory – core writes to local cache but changes not distributed to shared memory.
        - When another core tries to access data in shared mem, gets an out of date copy of it.

Distributed memory

- Alt to physical, shared mem
- Have multiple separate comps and data transferred to them over network

ADVANTAGES

- Scalability:
    - Distributed memory multicomp will physically scale easier than a shared memory multicomp
        - Potentially infinite memory and # of prcoessors
- Access to local data fast
    - Data distribution is very important
    - Must minimise communication

DISADVANTAGES

- Big **gap** between programming method and actual hardware primitives
    - Communication is over an interconnection network using OS or library calls.
- Access to remote data slow

Forum Questions

- *In the video for Hardware (ParallelJave2B), it was mentioned that for distributed memory, one if its disadvantages was that there's a "big gap between hardware architecture and the programming model".*
  I'm not quite sure what that means in the context of the programming models that have been taught (i.e shared memory and message passing)?
  - It can be an advantage to have this disconnect between the underlying hardware and the programming model if it ends up hiding unnecessary complexity. In this case abstraction is useful because it makes it easier to program without having to worry about unnecessary detail. It is a disadvantage when the abstraction hides away detail that affects performance. Remembering that performance is the reason we are doing parallel programming in the first place.
  - Let's say we have distributed shared memory at the hardware level shared between separate computers but the programming model makes it look like there is a single unified pool of shared memory. Basically, the programming model is abstracting away the fact that memory is located on different machines. As a programmer you now have no way to predict how long it is going to take to fetch a request from shared memory because it depends if it is on the local machine or one of the remote machines. In this case a message passing programming model may actually be better
- *"The overhead is having to manage memory consistency....For performance reasons" What are these performance reasons? Does inconsistent memory impact performance? And if yes - how? Does it slow down the speed at which things are processed or is it simply a case of producing incorrect outputs?*
  - A computational overhead implies that a computer system will be performing suboptimally. Inconsistent memory, or rather managing memory to ensure consistency introduces a performance hit because CPU computation cycles are wasted whilst the value in cache is compared to the value in local storage to check for staleness, as a simple crude example. Essentially, additional work must be done to ensure that memory is consistent which impacts on the performance of the CPU.

Programming Models

- Sits on top of h-ware
- A || programming mod:
  - Describes a || computing system in terms of the semantics of the programming language.
  - Specifies programmers' view of || computer.
  - Defines how programmer can code algo
- Two of most common models are:
  - Shared memory
  - Message passing.
- These models have diff means of communication bet processes/threads.
- Programming models bridge the gap between the underlying hardware architecture and the supporting layers of software available to applications. Programming models are different

from both programming languages and application programming interfaces (APIs). Specifically, a programming model is an abstraction of the underlying computer system that allows for the expression of both algorithms and data structures. In comparison, languages and APIs provide implementations of these abstractions and allow the algorithms and data structures to be put into practice – a programming model exists independently of the choice of both the programming language and the supporting APIs. (https://asc.llnl.gov/content/assets/docs/exascale-pmWG.pdf)

Parallel Programming Environments

- 3 main environments
- OpenMP:
  - Shared memory (pre-processor directives)
  - Pseudocode that gets compiled down to local code before main compilation cycle
  - Low overhead for ||sm.
  - Language extension to C, C++, and Fortran to write parallel programs for shared memory computers.
- MPI:
  - Msg-passing library used on clusters and other distributed memory computers
- Java:
  - Language features support || programming on shared-memory computers and standard class libraries supporting distributed computing (various models).

Concurrent execution

- With multiple processors, instructions from processes equals to the # of physical processors, which can be executed simultaneously.
  - Referred to as || / real concurrent execution
- Even on a multicore computer, usual to have more active processes than processors
  - In this case, available processes are switched between processors.

Threads and Processes in Java

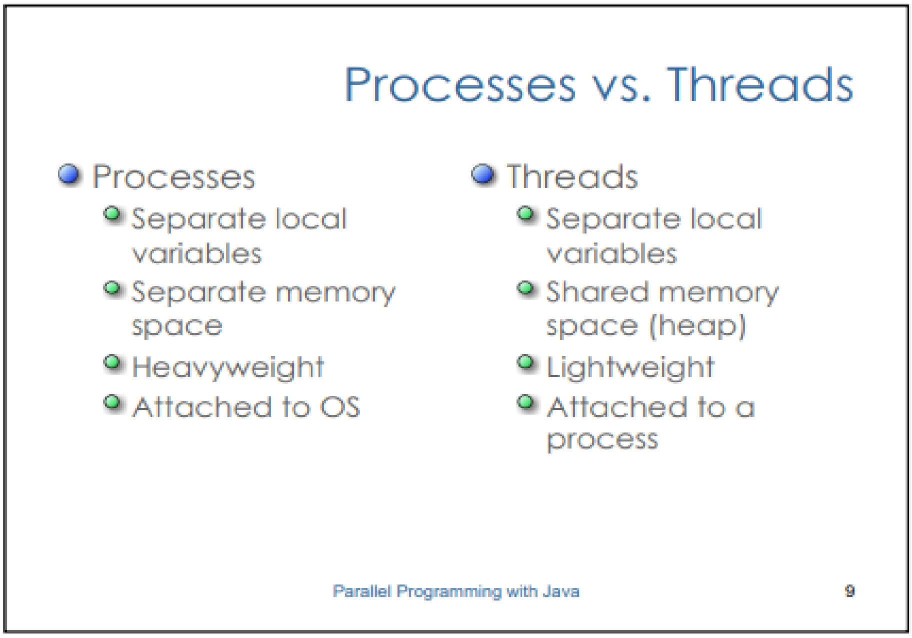
Terms

- Process: A larger unit of program execution with separate local variables, call stack, program counter and heap memory that is not shared with other processes. This is contrasted with threads, which are effectively sub-processes and can share memory.
- Thread: A unit of execution consisting of a sequence of statements. It is attached to a process and has its own separate local variables, call stack, and program counter but shares heap memory with other threads

Introduction

- Process: unit of resource allocation both for CPU time and for memory.
  - Represented by its code, data and state of machine registers.
    - A larger unit of program execution with separate local variables, call stack, program counter and heap memory that is not shared with other processes.
  - Each process has its own address space
    - Separate entities
    - Special action needed for processes to access shared data
- Thread: A unit of execution consisting of a sequence of statements
  - Effectively sub-processes
  - Processes have internal concurrency through lightweight sub-processes/threads.
  - Has its own separate local variables, call stack, and program counter but shares heap memory with other threads
    - Multiple threads of control

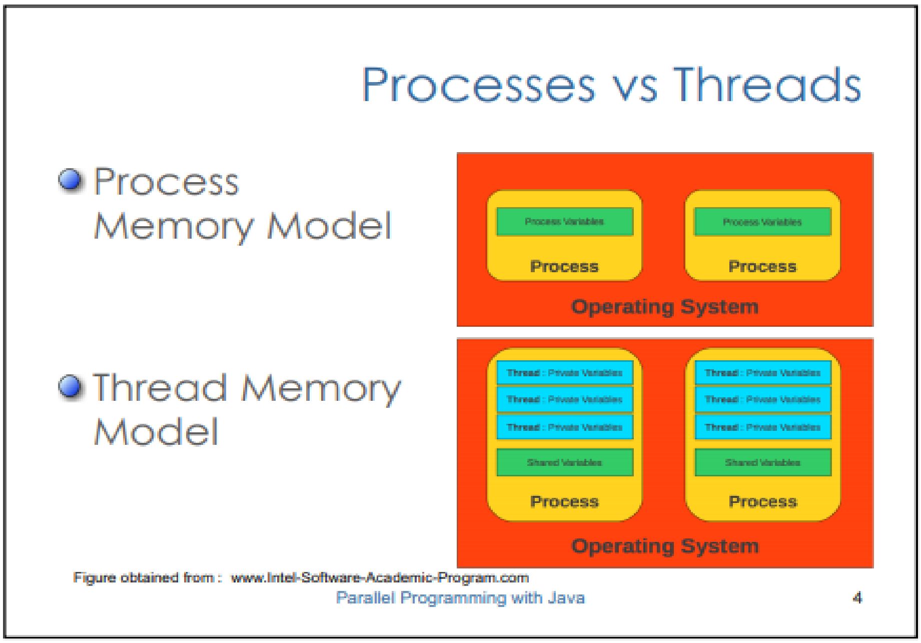
- Processes vs Threads:



Processes vs. Threads

- Processes
  - Separate local variables
  - Separate memory space
  - Heavyweight
  - Attached to OS
- Threads
  - Separate local variables
  - Shared memory space (heap)
  - Lightweight
  - Attached to a process

Parallel Programming with Java 9



Processes vs Threads

- Process Memory Model
- Thread Memory Model

Process Variables
Process
Process Variables
Process
Operating System
Thread : Private Variables
Thread : Private Variables
Thread : Private Variables
Shared Variables
Process
Thread : Private Variables
Thread : Private Variables
Thread : Private Variables
Shared Variables
Process
Operating System

Figure obtained from : www.Intel-Software-Academic-Program.com

Parallel Programming with Java 4

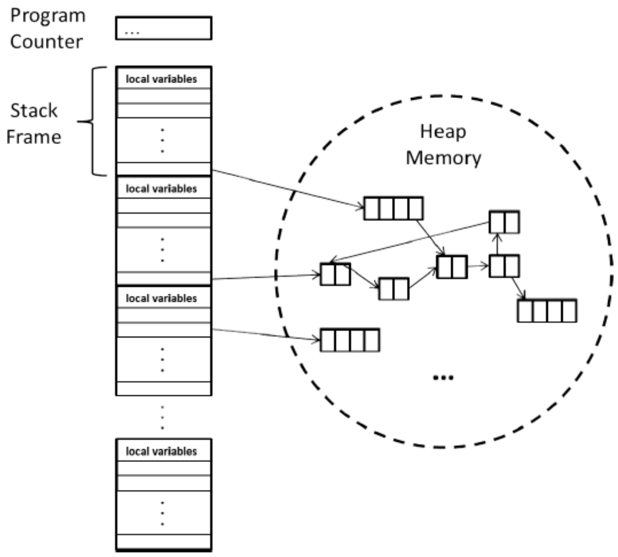
Programming model

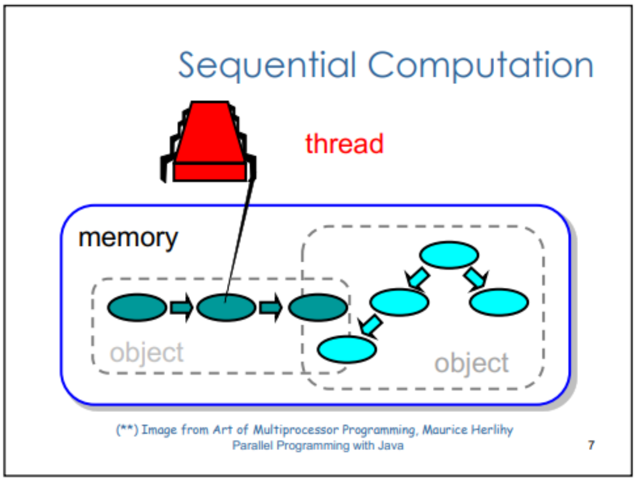
TERMS

- Call Stack : A data structure used implicitly by Java to hold in LIFO (last in, first out) order the stack frames associated with method calls made by a thread. A stack frame contains the local variables and calling parameters of a method among other things.
- Heap Memory: An area of memory used to store runtime data that is visible between threads. Function call data and local variables are stored on the stack instead. This is different from the heap data structure, which is a specific type of tree, and the two should not be confused

SEQUENTIAL PROGRAM STATE

- One program counter – current statement executing
- One call stack:
  - Each stack frame holds local variables for a method until it finishes.
  - Calling a method pushes frame and returning from a method pops it.
- Objects:
  - Objects created by calling **new**.
  - Heap: memory that holds objects (nothing to do with heap data structure).

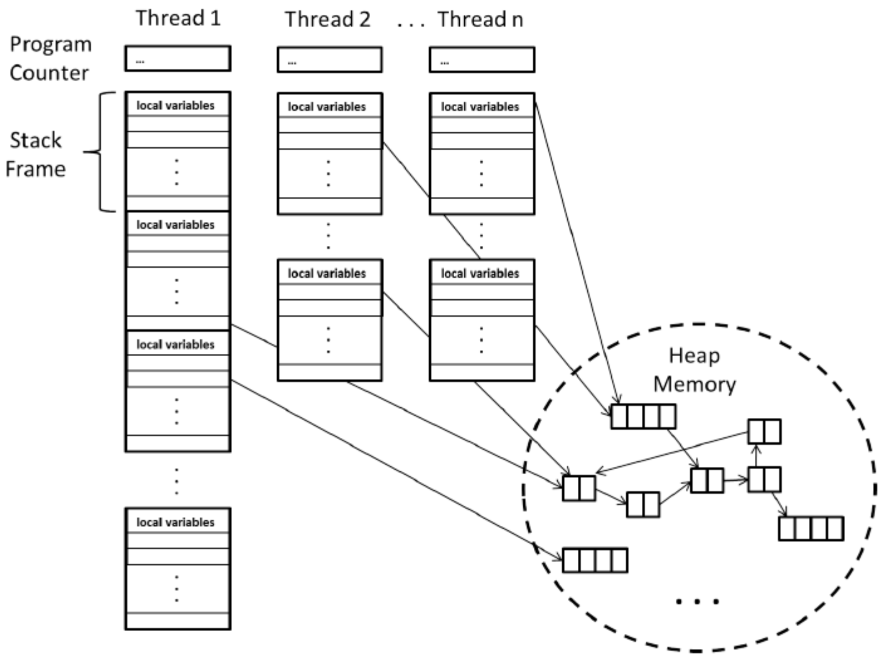


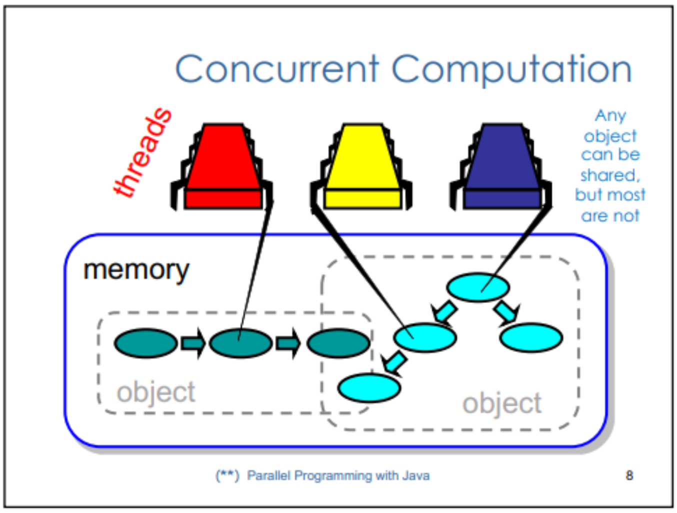

Sequential Computation

(**) Image from Art of Multiprocessor Programming, Maurice Herlihy
Parallel Programming with Java
7

SHARED MEMORY (|| PROGRAMMING MODEL)

- Explicit threads
- Each thread has its own prog counter, call stack and local variables.
- All threads share one collection of objs and static fields.
- Threads communicate through shared objs – implicit communication.
    - To communicate, write somewhere another thread reads.




Concurrent Computation

Any object can be shared, but most are not

(**) Parallel Programming with Java 8

Threads

- How threads run:
  - Programmer creates them
  - OS scheduler determines how those threads are used by different processors.
    - Don't have control over this.
    - Don't know how many processors prog will use.
    - Don't know order in which threads will be executed.
- Asynchrony:
  - Sudden unpredictable delays.
    - Cache misses – moderate
    - Page faults - long
    - Scheduling quantum used up – really long

Java Threads

- JVM
    - Executes as single process under some OS
    - Supports multiple threads
- Each Java thread has its own local vars, organised as a stack.
- Threads can access shared variables.
- Unpredictable asynchronous delays.
    - Arbitrary speed for each thread.
- Interleaving
    - Arbitrary order for threads.
- java.lang.Thread
    - Low level thread constructs – interrupt, join and sleep, notify and wait methods.
    - Some deprecated methods like stop and suspend.
- java.util.concurrent
    - Provides high lvl framework with library of utility classes and interfaces useful in concurrent programming.
        - Executors, queues, timing, synchronisers, concurrent collections.
        - Includes Fork-Join framework for making div-and-conquer algos easier to ||ise.

JAVA THREAD CLASS

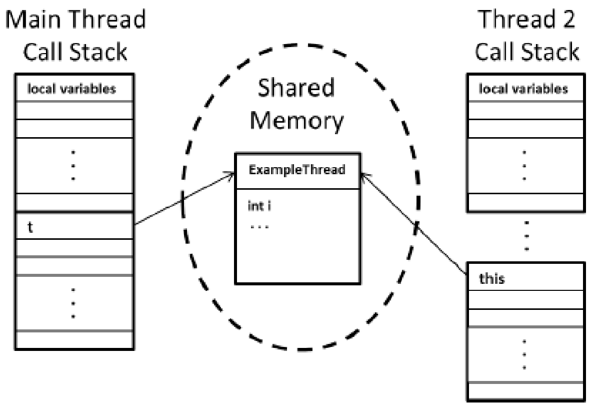
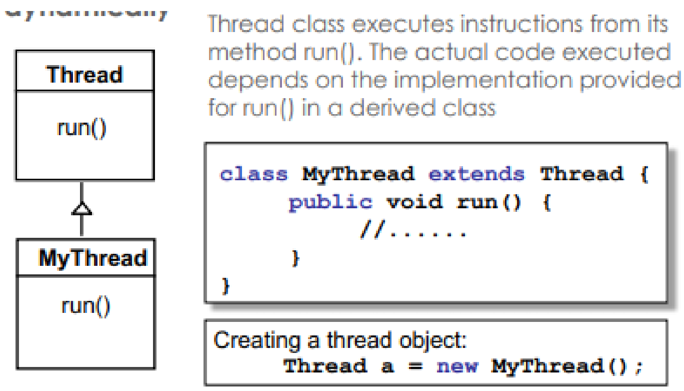


Figure 4: How we can use shared memory, specifically fields in a subclass of Thread, to pass data to a newly created thread.

- Thread class manages single sequential thread of control.
    - Threads may be created and deleted dynamically.
- Thread class executes instructions for its run() method.
    - Actual code executed depends on how run() method is implemented in a derived class.



Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class

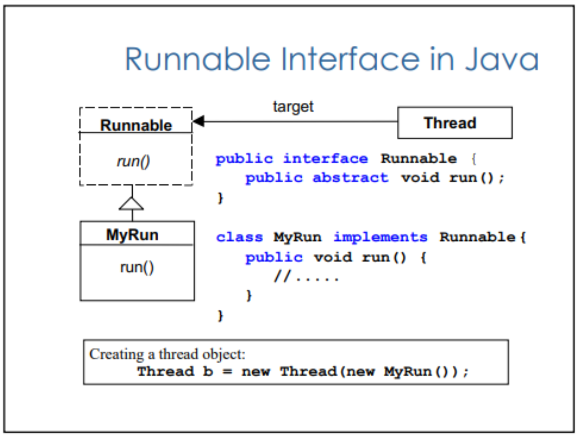
```
class MyThread extends Thread {
      public void run() {
          //......
      }
}
```

Creating a thread object:

```
      Thread a = new MyThread();
```

TYPES OF THREADS IN JAVA

- Java doesn't allow multiple inheritance
    - Sometimes convenient to implement run() for Runnable interface instead of inheriting from thread.



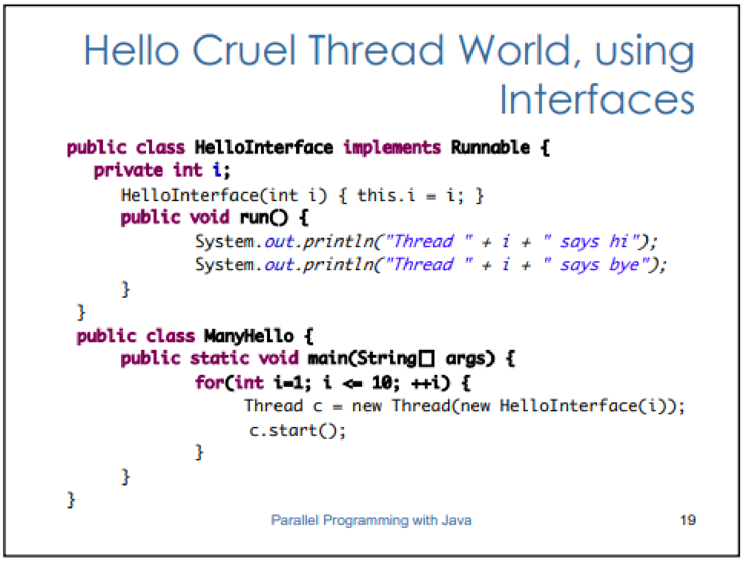
Runnable Interface in Java

```
public interface Runnable {
   public abstract void run();
}

class MyRun implements Runnable{
   public void run() {
     //.....
   }
}
```

Creating a thread object:

```
     Thread b = new Thread(new MyRun());
```

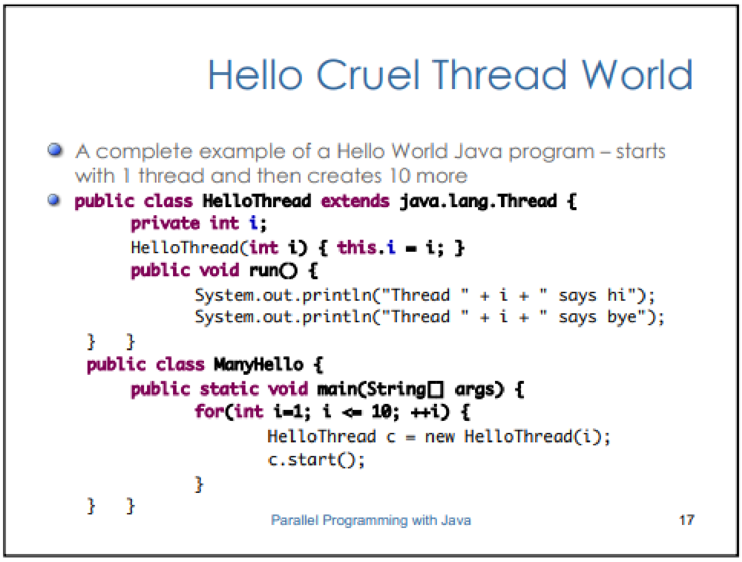
- 2 ways to create basic Java thread:
  1. Implement Runnable interface (java.lang.Runnable)



Hello Cruel Thread World, using Interfaces

```
public class HelloInterface implements Runnable {
   private int i;
      HelloInterface(int i) { this.i = i; }
      public void run() {
            System.out.println("Thread " + i + " says hi");
            System.out.println("Thread " + i + " says bye");
      }
 }
 public class ManyHello {
      public static void main(String[] args) {
            for(int i=1; i <= 10; ++i) {
                  Thread c = new Thread(new HelloInterface(i));
                   c.start();
            }
      }
}
```

2. Extend Thread class (java.lang.Thread)



Hello Cruel Thread World

- A complete example of a Hello World Java program – starts with 1 thread and then creates 10 more
-

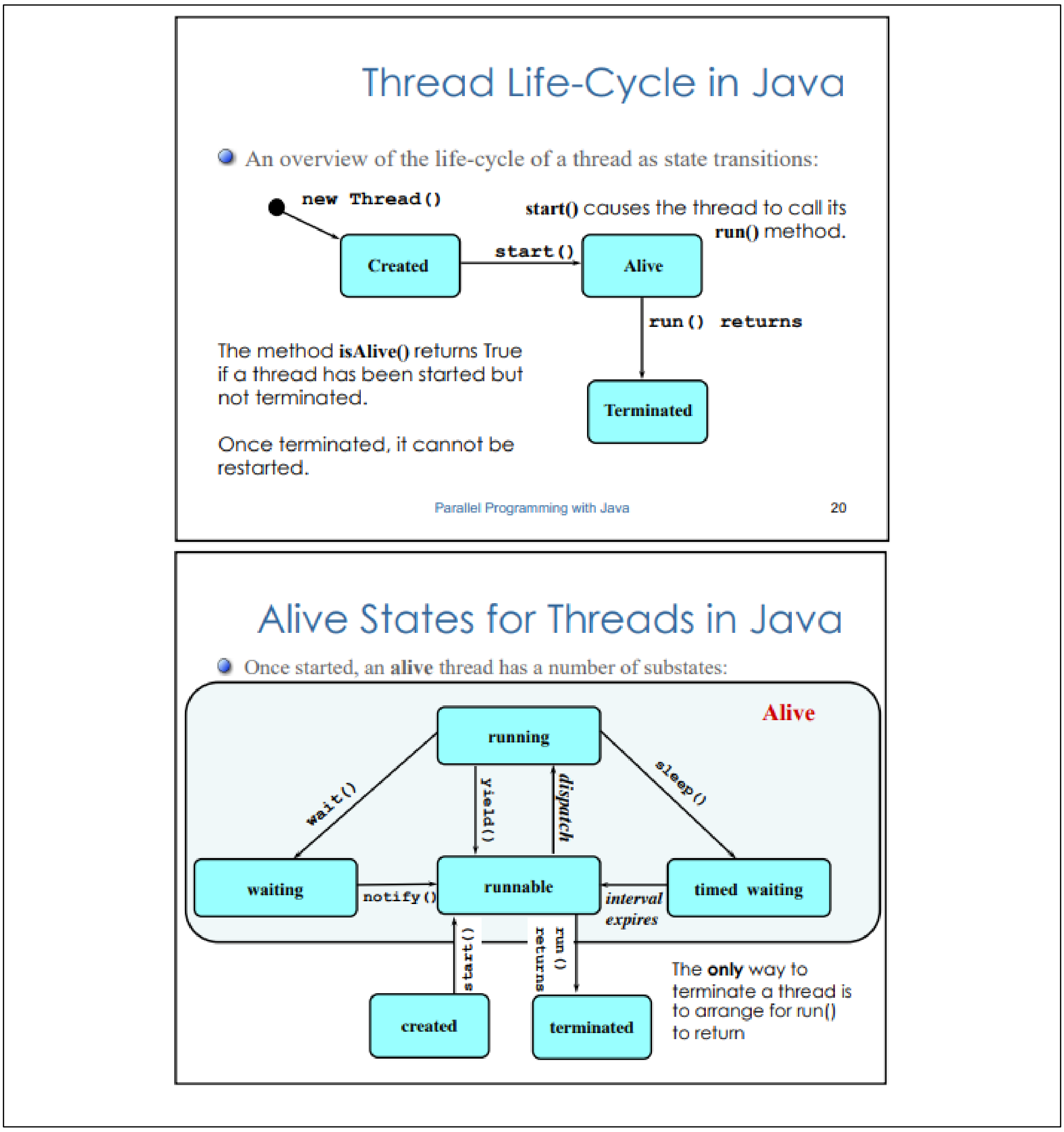
```
public class HelloThread extends java.lang.Thread {
      private int i;
      HelloThread(int i) { this.i = i; }
      public void run() {
            System.out.println("Thread " + i + " says hi");
            System.out.println("Thread " + i + " says bye");
 }  }
 public class ManyHello {
      public static void main(String[] args) {
            for(int i=1; i <= 10; ++i) {
                  HelloThread c = new HelloThread(i);
                  c.start();
            }
 }  }
```

Parallel Programming with Java 17

USING JAVA THREADS

- Construction of a Thread obj doesn’t cause thread to run
- To get new thread running:
  - Define subclass A of java.lang.Thread, overriding run()
  - Create obj of class A
  - Call that obj’s start().
- Don’t call run() – would just be normal method call
- start() sets off new thread, using run() as its “main”.

THREAD LIFE-CYCLE IN JAVA



Thread Life-Cycle in Java

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

start() causes the thread to call its **run()** method.

The method **isAlive()** returns True if a thread has been started but not terminated.

Once terminated, it cannot be restarted.

Alive States for Threads in Java

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

The **only** way to terminate a thread is to arrange for run() to return

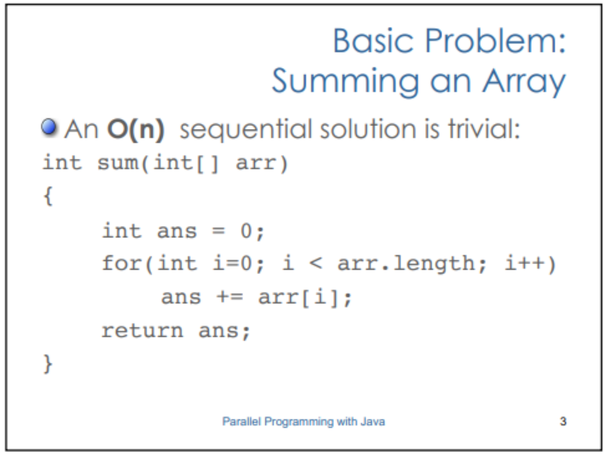
Overview

- To write a shared-memory parallel program, need new programming primitives
- Ways to create and run multiple things at once.
  - Call these things *threads*
- Ways for threads to share memory
  - Usually just threads with references to the same objects
- Ways for threads to coordinate
  - For now, a way for one thread to wait for another to finish.
  - Other primitives used for concurrency

FORALL loop

- FORALL loop does all iterations in ||
    - Code after FORALL loop doesn't execute until all its iterations are done.
    - Iterations must be able to be done at the same time without them interfering with each other.
        - If one loop iteration writes to location, another iteration must NOT read/write to that location.
        - It is fine if two iterations read to the same location – this doesn't cause any interference.
- Java doesn't have FORALL loop.
    - Can use threads to make one:
        - In regular for loop, create one thread to do each iteration of a FORALL loop, passing data needed to constructor
            1. Have threads store their answers in fields of themselves.
            2. Wait for all threads created in Step 1 to terminate.
            3. Combine results by reading answers out of fields of threads created in Step 1

Basic Problem: Summing up n elements in array

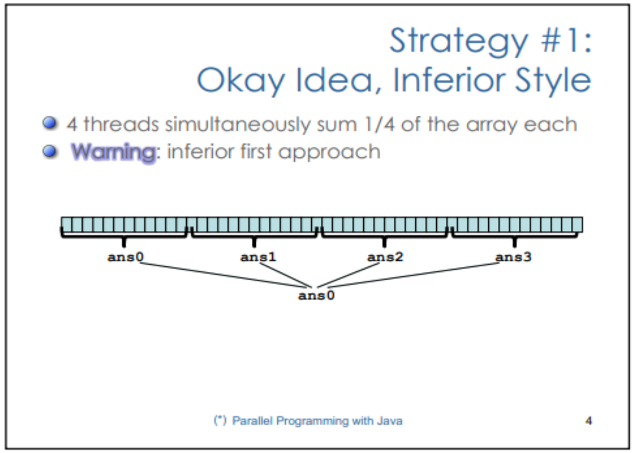


Basic Problem: Summing an Array

- An **O(n)** sequential solution is trivial:

```
int sum(int[] arr)
{
    int ans = 0;
    for(int i=0; i < arr.length; i++)
        ans += arr[i];
    return ans;
}
```

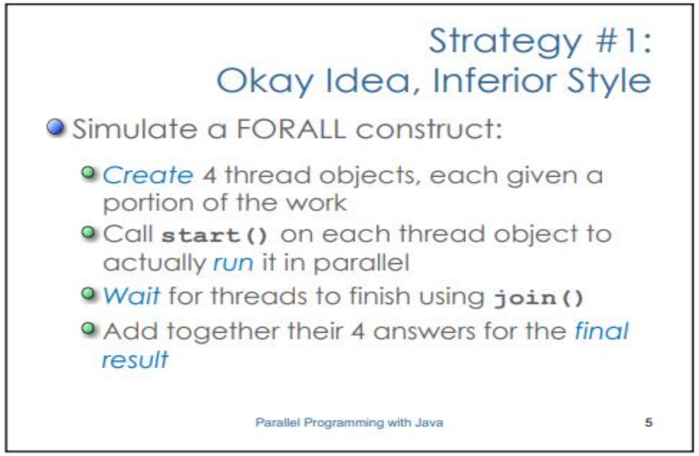
Parallel Programming with Java 3



Strategy #1: Okay Idea, Inferior Style

- 4 threads simultaneously sum 1/4 of the array each
- **Warning**: inferior first approach

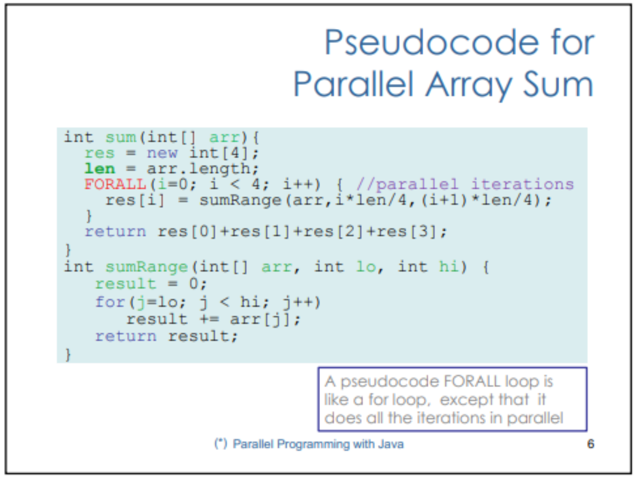
(*) Parallel Programming with Java 4



Strategy #1: Okay Idea, Inferior Style

- Simulate a FORALL construct:
  - *Create* 4 thread objects, each given a portion of the work
  - Call **`start()`** on each thread object to actually *run* it in parallel
  - *Wait* for threads to finish using **`join()`**
  - Add together their 4 answers for the *final result*

Parallel Programming with Java 5

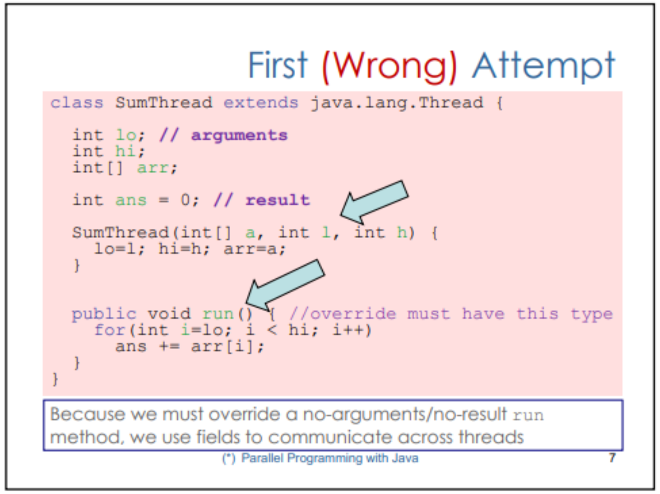


Pseudocode for Parallel Array Sum

```
int sum(int[] arr){
  res = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; i++) { //parallel iterations
    res[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return res[0]+res[1]+res[2]+res[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

A pseudocode FORALL loop is like a for loop, except that it does all the iterations in parallel

(*) Parallel Programming with Java 6

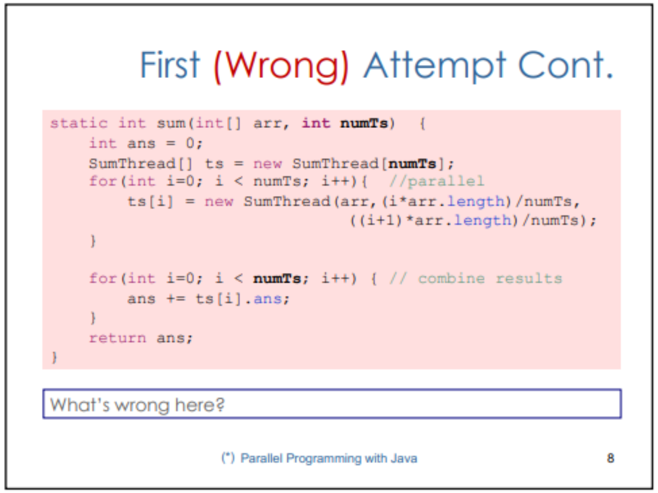


First (Wrong) Attempt

```
class SumThread extends java.lang.Thread {

  int lo; // arguments
  int hi;
  int[] arr;

  int ans = 0; // result

  SumThread(int[] a, int l, int h) {
    lo=l; hi=h; arr=a;
  }


  public void run() { //override must have this type
    for(int i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

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

(*) Parallel Programming with Java 7

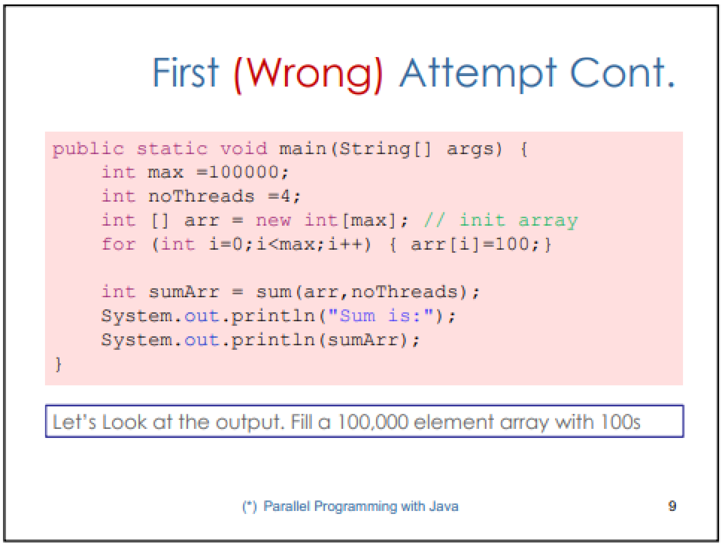


First (Wrong) Attempt Cont.

```
static int sum(int[] arr, int numTs)  {
    int ans = 0;
    SumThread[] ts = new SumThread[numTs];
    for(int i=0; i < numTs; i++){  //parallel
        ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                                  ((i+1)*arr.length)/numTs);
    }

    for(int i=0; i < numTs; i++) { // combine results
        ans += ts[i].ans;
    }
    return ans;
}
```

What's wrong here?

(*) Parallel Programming with Java 8



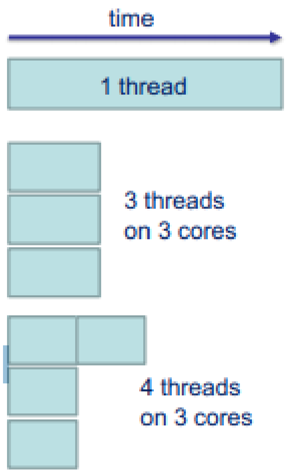
First (Wrong) Attempt Cont.

```
public static void main(String[] args) {
    int max =100000;
    int noThreads =4;
    int [] arr = new int[max]; // init array
    for (int i=0;i<max;i++) { arr[i]=100;}

    int sumArr = sum(arr,noThreads);
    System.out.println("Sum is:");
    System.out.println(sumArr);
}
```

Let's Look at the output. Fill a 100,000 element array with 100s

(*) Parallel Programming with Java 9

Number of Threads



- Want code to be reusable and cross-platform efficient
  - "forward-portable" as core count grows.
- Parameterise by # of threads.
  - Have **P** processors, can divide array into P equal segments
  - Algo runs in O((N/P) + P) time
    - N/P is || part
    - P is for combining stored results.
- Setting num threads = processors is poor practice.
  - Want parallel programs that effectively use the processors available to them.
  - Bad assumption is that every processor is available to the code we are writing. - assumes your program has sole access.
    - Maybe caller is also using parallelism.

        - Available cores can change even while your threads run.
            - Some processors may be needed by other programs or even other parts of the same program. The operating system can reassign processors at any time.

LOAD IMBALANCE

- Cannot always predictably divide the work into approximately equal pieces

- Have assumed that the threads processing equal-size chunks of the array take approximately the same amount of time.

    - They may not, due to memory-hierarchy issues/other architectural effects.

    - More sophisticated algorithms could produce a large load imbalance = different helper threads are given different amounts of work.

- Simple example, suppose we have a large int[] and we want to know how many elements of the array are prime numbers.

    - If one portion of the array has more large prime numbers than another, then one helper thread may take longer.

- In general, subproblems may have very different run-times
    - Example: Apply method f to every array element, but maybe f is much slower for some data items
- Example of a load imbalance
    - If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup
- Giving each helper thread an equal number of data elements is not necessarily the same as giving each helper thread an equal amount of work.
- Any load imbalance hurts efficiency since need to wait until all threads are completed.

MANY THREADS

(Why Not To Use One Thread Per Processor)

- Want || programs that effectively use the processors available to them.

- Some method can take as a parameter the # of threads to use, leaving it to some other part of the program to decide the #.

- The processors available to part of the code can change

    - Bad assumption is that every processor is available to the code we are writing.

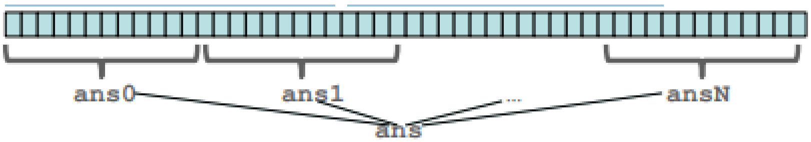
  - But some processors may be needed by other programs or even other parts of the same program. The operating system can reassign processors at any time.
- Num threads >> Num processors is much better.
  - Use substantially more threads than there are processors.
  - Lots of helpers each doing a small piece.
  - Hand out “work chunks” as you go to available processors.
  - If 3 processors available and have 100 threads, then ignoring constant-factor overheads, extra time is < 3% .
  - Load imbalance fixable if slow thread scheduled early.

- For example, suppose to sum the elements of an array we created one thread for each 1000 elements.

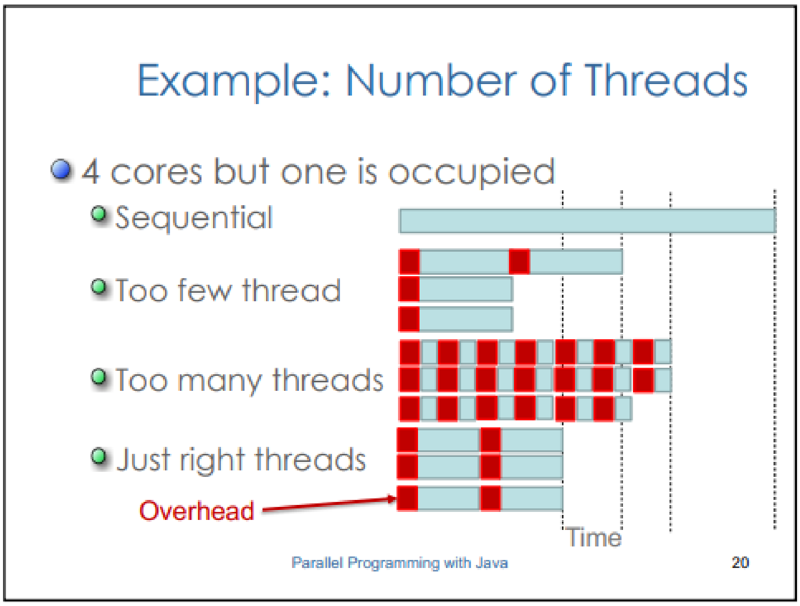
  - Assuming a large enough array (size greater than 1000 times the number of processors), the threads will not all run at once since a processor can run at most one thread at a time.

  - But this is fine: system will keep track of what threads are waiting and keep all the processors busy. Some overhead to creating more threads, use a system where this overhead is small.

- For summing elements example, this will require changing our algorithm and abandoning medium-weight Java threads



Example



Example: Number of Threads

- 4 cores but one is occupied
  - Sequential
  - Too few thread
  - Too many threads
  - Just right threads

Parallel Programming with Java 20

- This approach fixes the first problem: any number of processors will stay busy until the very end when there are fewer 1000-element chunks remaining than there are processors.
- Also fixes the second problem since we just have a “big pile” of threads waiting to run. If the number of processors available changes, that affects only how fast the pile is processed, but we are always doing useful work with the resources available.
- Lastly, this approach helps with the load imbalance problem: Smaller chunks of work make load imbalance far less likely since the threads do not run as long. Also, if one processor has a slow chunk, other processors can continue processing faster chunks.
- Go back to our cutting-potatoes analogy to understand this approach:
  - Rather than give each of 4 cooks (processors), 1/4 of the potatoes, we have them each take a moderate number of potatoes, slice them, and then return to take another moderate number.
  - Since some potatoes may take longer than others (they might be dirtier or have more eyes), this approach is better balanced and is probably worth the cost of the few extra trips to the pile of potatoes —especially if one of the cooks might take a break (processor used for a different program) before finishing his/her pile.

- This approach has two problems
  - 1. We now have more results to combine. Dividing the array into 4 total pieces leaves O(1) results to combine
    - Dividing the array into 1000-element chunks leaves arr.length/1000, which is O(n), results to combine.

    - Combining the results with a sequential for-loop produces an O(n) algorithm, albeit with a smaller constant factor.

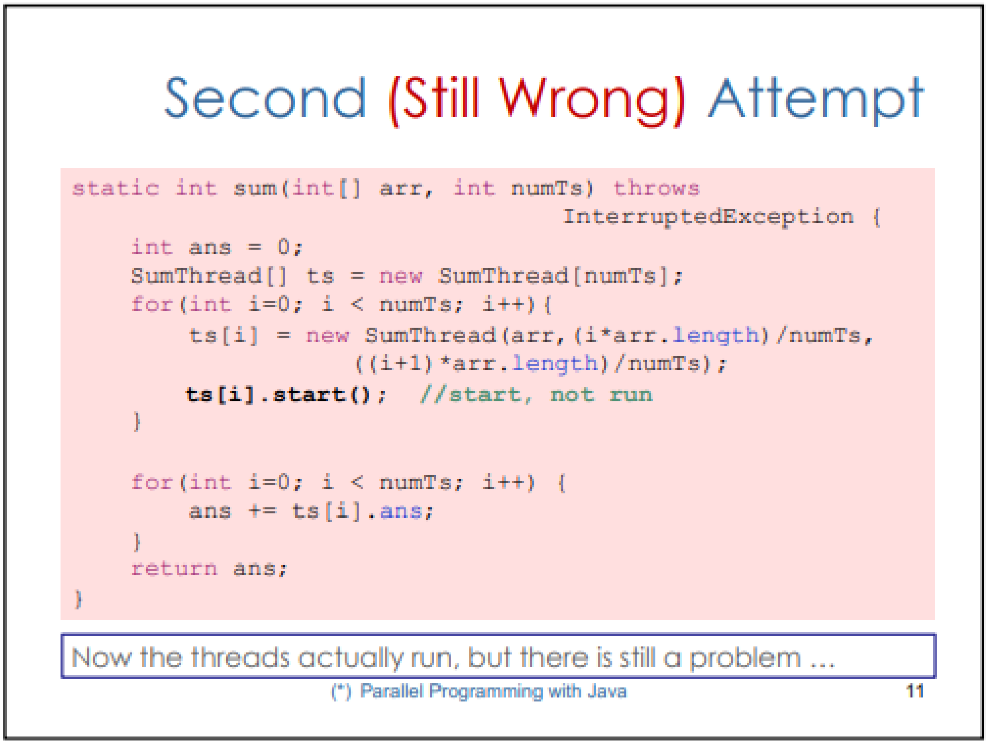
    - To see the problem even more clearly, suppose we go to the extreme and use 1-element chunks — now the results combining reimplements the original sequential algorithm. In short, we need a better way to combine results.

- 2. Java’s threads were not designed for small tasks like adding 1000 numbers.

    - They will work and produce the correct answer, but the constant-factor overheads of creating a Java thread are far too large.

    - A Java program that creates 100,000 threads on a small desktop computer is unlikely to run well at all — each thread just takes too much memory and the scheduler is overburdened and provides no asymptotic run-time guarantee.

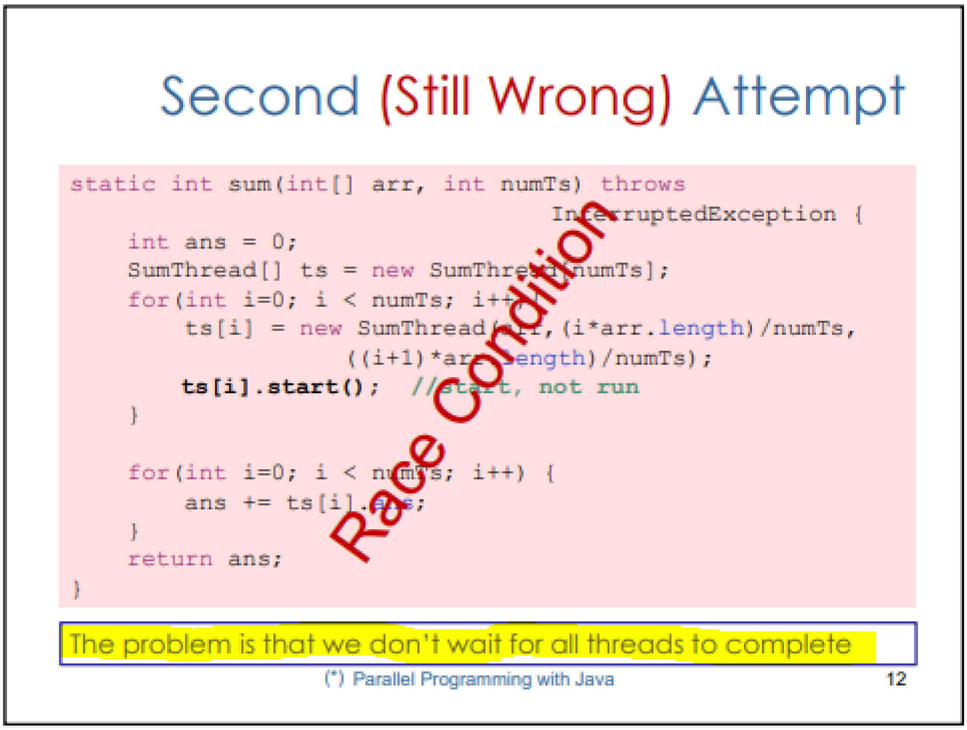
    - In short, we need a different implementation of threads that is designed for this kind of fork/join programming



Second (Still Wrong) Attempt

```
static int sum(int[] arr, int numTs) throws
                                     InterruptedException {
    int ans = 0;
    SumThread[] ts = new SumThread[numTs];
    for(int i=0; i < numTs; i++){
        ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                    ((i+1)*arr.length)/numTs);
        ts[i].start();  //start, not run
    }

    for(int i=0; i < numTs; i++) {
        ans += ts[i].ans;
    }
    return ans;
}
```

Now the threads actually run, but there is still a problem …



Second (Still Wrong) Attempt

```
static int sum(int[] arr, int numTs) throws
                                     InterruptedException {
    int ans = 0;
    SumThread[] ts = new SumThread[numTs];
    for(int i=0; i < numTs; i++){
        ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                    ((i+1)*arr.length)/numTs);
        ts[i].start();  //start, not run
    }

    for(int i=0; i < numTs; i++) {
        ans += ts[i].ans;
    }
    return ans;
}
```

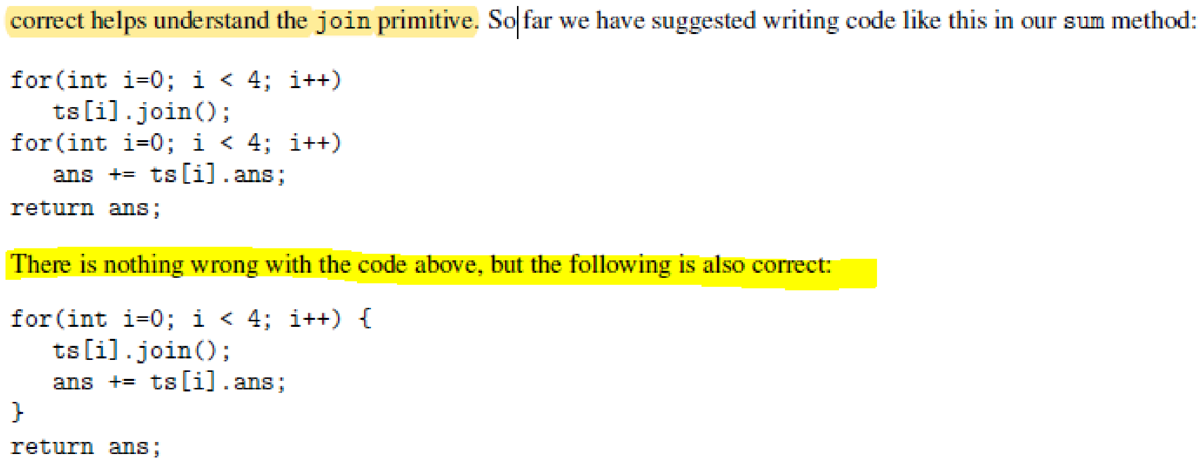
Race Condition

The problem is that we don't wait for all threads to complete

Fork Join

Synchronisation

- join()
  - Synchronisation primitive which causes one thread to wait until another thread has terminated
  - Caller blocks until/unless the receiver is done executing (meaning its run returns)
    - Else we would have a race condition on *ts[i].ans*
- This style of parallel programming is called “fork/join”.
- Synchronisation bug: main thread doesn’t wait for helper threads to finish before sums *ans* field.
  - Start returns immediately – otherwise no ||sm.
  - Having one thread reading to field while another thread is writing to that same field = bug!
  - If a thread calls the join() of another thread obj, then this call blocks (doesn’t return) until/unless the thread corresponding to the obj has terminated.
  - Main thread calls join()
    - Main thread will block until helper thread done – one helper thread e.g. 3 could finish before 2, but this isn’t a problem, since a call to join when helper thread has already terminated just returns right away (no blocking).
  - The idea of a FORALL loop is encoded as thus: using two for loops, where the first one creates the helper threads and the second one waits for them all to terminate.
    - This is Fork-Join ||ism.
  - It’s like we create a “(4-way in this case) fork in the road of execution” and send each helper thread down one path of the fork.
    - Then join all the paths of the fork back together and have the single main thread continue.
  - Fork-join parallelism can also be nested, meaning one of the helper threads forks its own helper threads.
  - It is common to combine the joining for-loop and the result-combining for-loop. Understanding why this is still correct helps understand the join primitive.



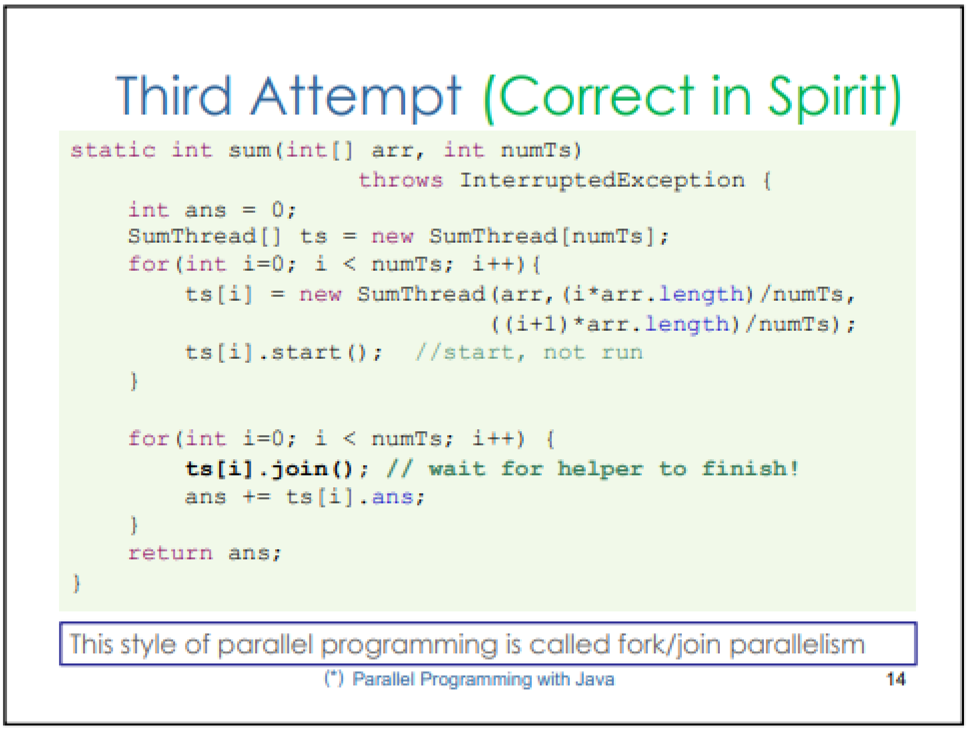
correct helps understand the join primitive. So far we have suggested writing code like this in our sum method:

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

There is nothing wrong with the code above, but the following is also correct:

```
for(int i=0; i < 4; i++) {
   ts[i].join();
   ans += ts[i].ans;
}
return ans;
```

- Here, don't wait for all helper threads to finish before producing final answer.
    - BUT still ensure that main thread doesn't access helper thread's ans field until at least that helper thread has terminated.



Third Attempt (Correct in Spirit)

```
static int sum(int[] arr, int numTs)
                    throws InterruptedException {
    int ans = 0;
    SumThread[] ts = new SumThread[numTs];
    for(int i=0; i < numTs; i++){
        ts[i] = new SumThread(arr,(i*arr.length)/numTs,
                              ((i+1)*arr.length)/numTs);
        ts[i].start();  //start, not run
    }

    for(int i=0; i < numTs; i++) {
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    }
    return ans;
}
```

This style of parallel programming is called fork/join parallelism

(*) Parallel Programming with Java 14

EXCEPTION HANDLING

- join() may throw java.lang.InterruptedException - so a method calling join will not compile unless it catches this exception or declares that it might be thrown
    - Should be fine to catch-and-exit for basic parallel programming.
    - For concurrent programming, it may be bad style to ignore this.

Shared Memory

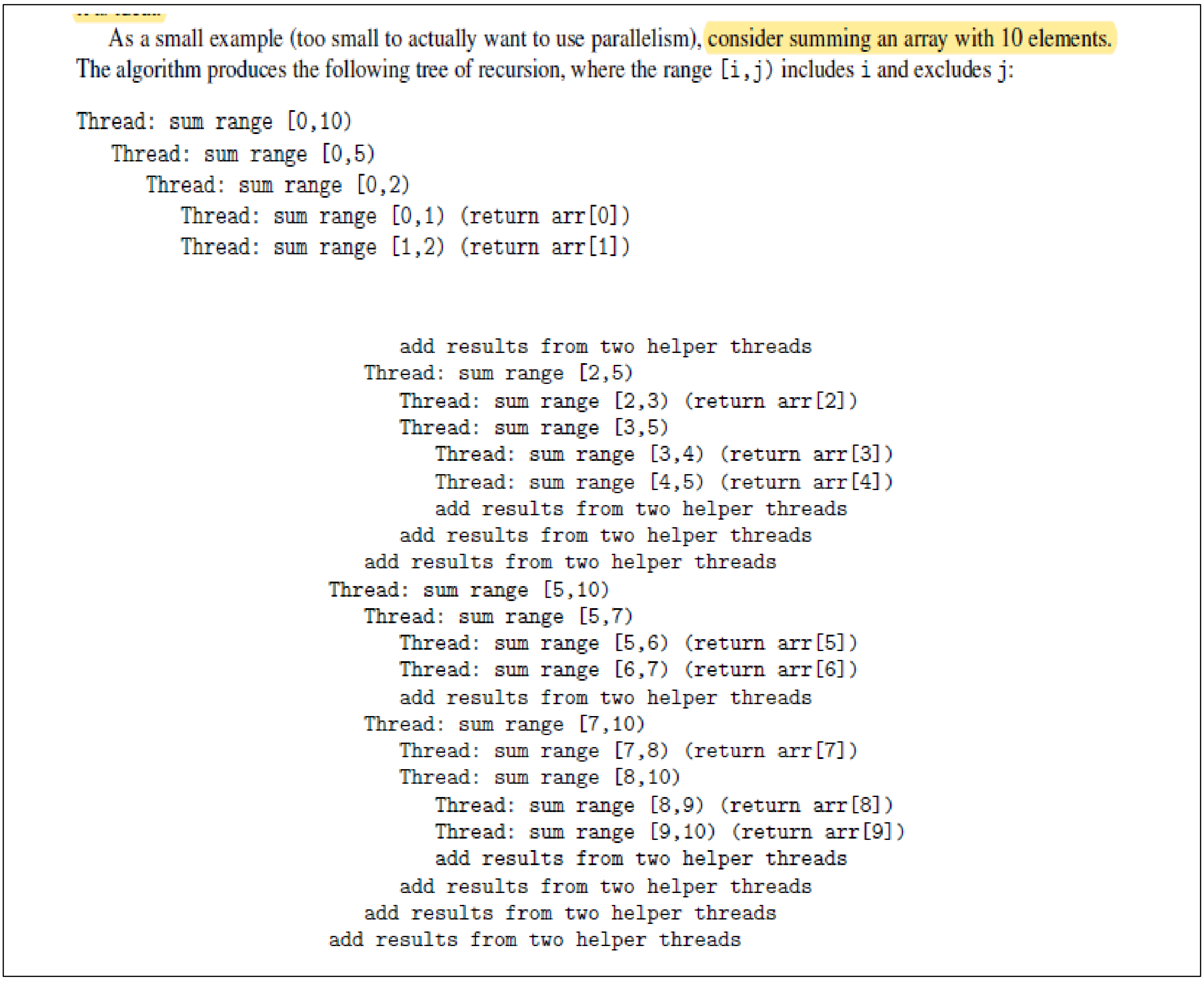
- Fork-Join progs don't require much focus on sharing memory.
- In Java, memory is being shared:
    - In the example:
        - lo, hi, arr fields written by "main" thread, read by helper thread.
        - ans field written by helper thread, read by "main" thread.
- Must avoid race conditions when using shared memory
    - For ||sm, stick with join

Divide and Conquer

- Why is current algo poor?
  - Suppose we create 1 thread to process every 1000 elements
- Then combining results will have (arr.length / 1000) additions to do
  - Still linear array size
- In fact, if we create 1 thread for every 1 element, we recreate a sequential algorithm

DIVIDE-AND-CONQUER PARALLELISM

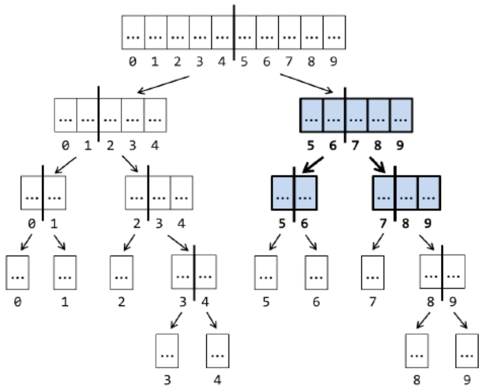
- Idea of divide-and-conquer parallelism using Java's threads.
- The key idea is to change algorithm for summing the elements of an array to use recursive divide-and-conquer.
- To sum all the array elements in some range from lo to hi, do the following:
  1. If the range contains only one element, return that element as the sum. Else in parallel:
  (a) Recursively sum the elements from lo to the middle of the range.
  (b) Recursively sum the elements from the middle of the range to hi.
  2. Add the two results from the previous step.
- The essence of the recursion is that steps 1a and 1b will themselves use parallelism to divide the work of their halves in half again



As a small example (too small to actually want to use parallelism), consider summing an array with 10 elements. The algorithm produces the following tree of recursion, where the range [i,j) includes i and excludes j:

```
Thread: sum range [0,10)
   Thread: sum range [0,5)
      Thread: sum range [0,2)
         Thread: sum range [0,1) (return arr[0])
         Thread: sum range [1,2) (return arr[1])



                            add results from two helper threads
                         Thread: sum range [2,5)
                            Thread: sum range [2,3) (return arr[2])
                            Thread: sum range [3,5)
                               Thread: sum range [3,4) (return arr[3])
                               Thread: sum range [4,5) (return arr[4])
                               add results from two helper threads
                            add results from two helper threads
                         add results from two helper threads
                      Thread: sum range [5,10)
                         Thread: sum range [5,7)
                            Thread: sum range [5,6) (return arr[5])
                            Thread: sum range [6,7) (return arr[6])
                            add results from two helper threads
                         Thread: sum range [7,10)
                            Thread: sum range [7,8) (return arr[7])
                            Thread: sum range [8,10)
                               Thread: sum range [8,9) (return arr[8])
                               Thread: sum range [9,10) (return arr[9])
                               add results from two helper threads
                            add results from two helper threads
                         add results from two helper threads
                      add results from two helper threads
```

- The total amount of work done by this algorithm is O(n) because we create approximately 2n threads and each thread either returns an array element or adds together results from two helper threads it created.
    - If we have O(n) processors, then this algorithm can run in O(log n) time, which is exponentially faster than the sequential algorithm.
    - The key reason for the improvement is that the algorithm is combining results in parallel.
        - The recursion forms a binary tree for summing subranges and the height of this tree is log n for a range of size n.
            - The number of nodes is growing exponentially faster than the tree height. With enough processors, the total running time corresponds to the tree height, not the tree size: this is the fundamental running-time benefit of parallelism.



Algorithm

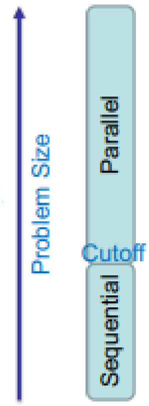
- Spread the accumulation of partial results across threads
- Implement using divide-and-conquer
    - ||ism for the recursive calls

Results

- Really works
- Divide-and-conquer parallelizes the result-combining.
    - If you have enough processors, total time is tree height: O(log n)
        - (exponentially faster than O(n))
- Write all our parallel algorithms in this style

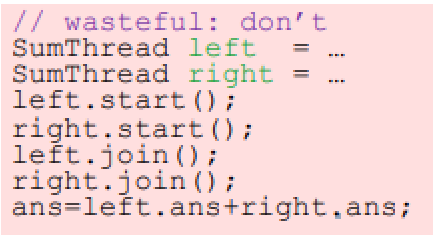
- ○ Using this style often relies on operations being associative (like +) i.e. a+b is same as b+a.

Sequential cutoff

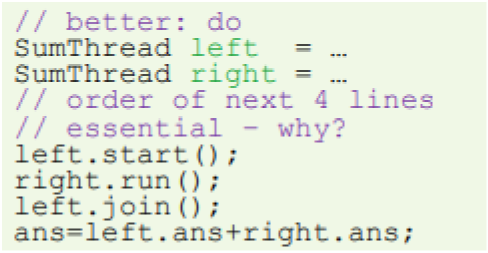


- In theory, dividing down to single elements gives optimal speedup O(n/P + log n) cost.
- In practice, creating all those threads and communicating swamps the savings.
  - Can produce far too many threads to be efficient. To add up four numbers, does it really make sense to create six new threads?
  - Therefore, implementations of fork/join algorithms use a cutoff, below which they switch over to a sequential algorithm.
  - Because this cutoff is a constant, it has no effect on the asymptotic behavior of the algorithm.
    - But it does eliminate the vast majority of the threads created, while still preserving enough parallelism to balance the load among the processors.
- Sequential cutoff typically around 500-1000 elements.
  - Eliminates almost all the recursive thread creation (bottom levels of tree).

Half the Threads



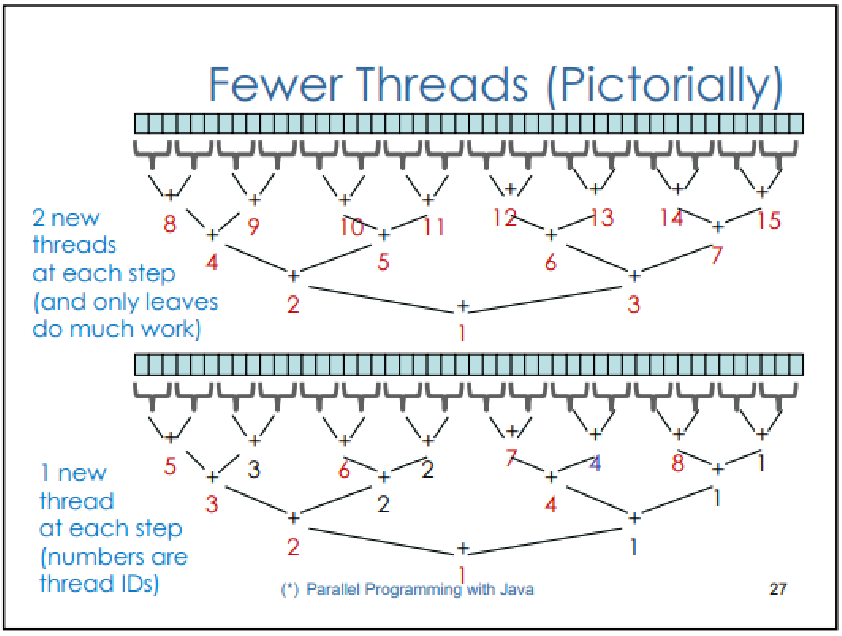
```
// wasteful: don't
SumThread left  = …
SumThread right = …
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```



```
// better: do
SumThread left  = …
SumThread right = …
// order of next 4 lines
// essential – why?
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

- The key is to notice that all threads that create two helper threads are not doing much work themselves: they divide the work in half, give it to two helpers, wait for them to finish, and add the results.

- Better to create one thread and do the other yourself i.e. DON'T create two recursive threads.
- Halves the # of threads.



Fewer Threads (Pictorially)

2 new threads at each step (and only leaves do much work)

1 new thread at each step (numbers are thread IDs)

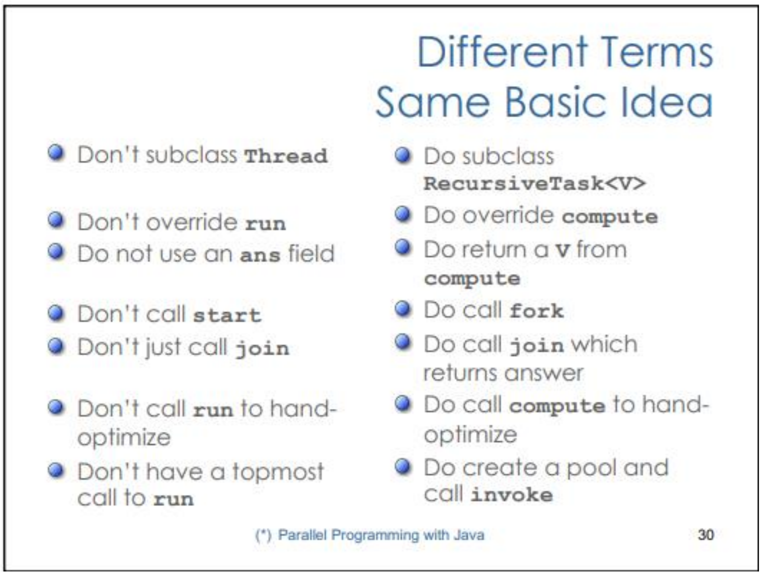
(*) Parallel Programming with Java

27

Fork-Join Framework

https://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html

- Java threads too “heavyweight” – not good for small tasks.
  - Takes much longer to create, start running and dispose of thread.
  - Constant factors and space overhead.
  - The space overhead = prohibitive.
    - Not uncommon for a Java implementation to pre-allocate the maximum amount of space it allows for the call-stack, which might be 2MB or more.
    - So, creating thousands of threads could use gigabytes of space.
- Fork-Join framework meets needs of divide-and conquer fork-join parallelism
  - Supports lightweight threads - small enough that even a million can be created
  - Includes a scheduler and run-time system with provably optimal expected-time guarantees
- Java threads vs Fork-Join framework:



Different Terms
Same Basic Idea

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

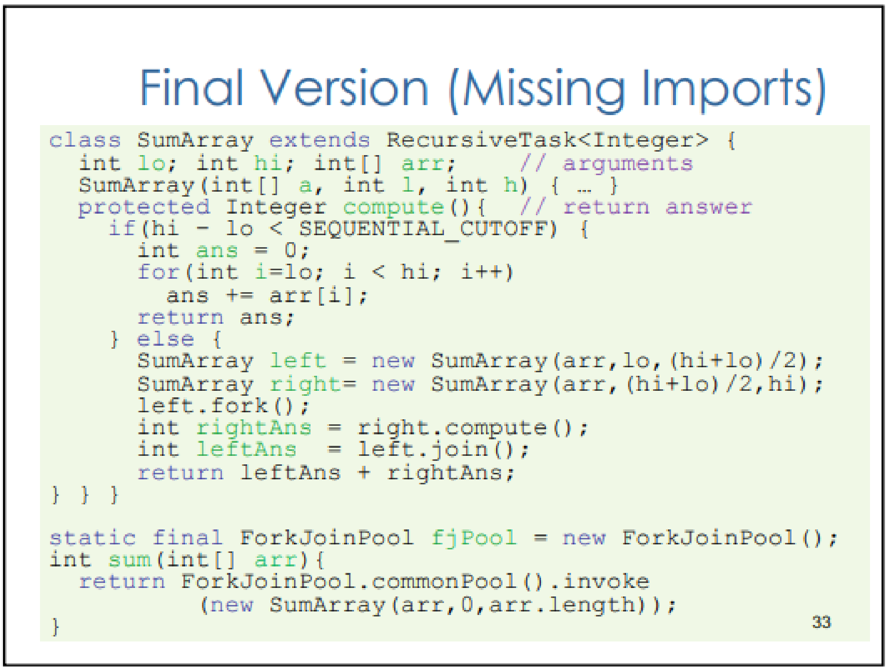
(*) Parallel Programming with Java 30

FORKJOIN FORMS

- Two types of Fork-Join task specialisations:
  - Instances of RecursiveAction do not yield a return value, while instances of RecursiveTask do.
- RecursiveTask generally preferred
  - Because most divide-and-conquer algorithms return a value from a computation over a data set
- Both are actually implemented in terms of the same superclass inside the library.

WORKING WITH THE LIBRARY

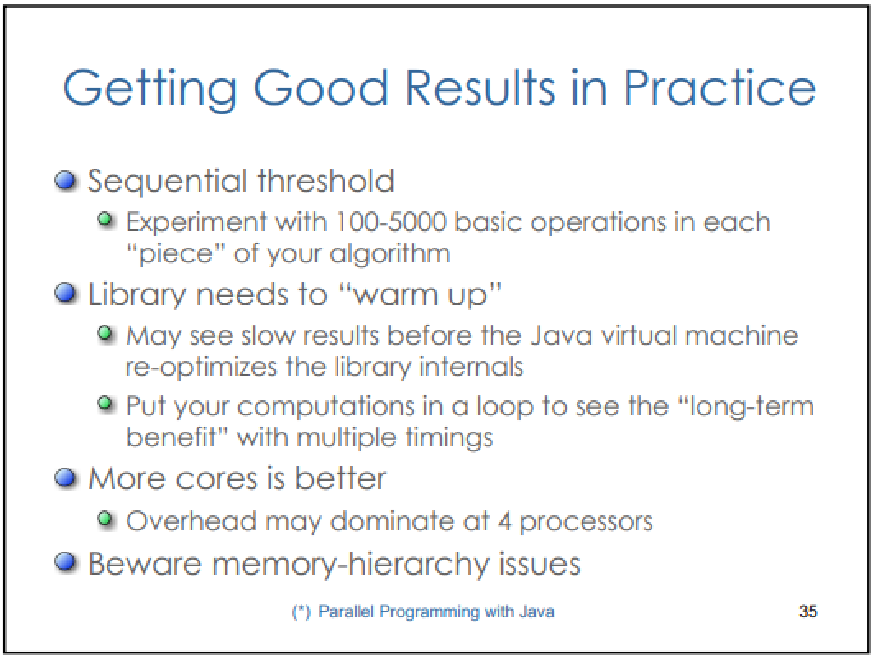
- One ForkJoinPool for the entire program
  - Store in a static field
- Inside a subclass of RecursiveAction, you use fork, compute, and join
  - But can’t use these methods outside the subclass
- To start use the invoke method of ForkJoinPool
  - You pass a subclass of RecursiveAction or RecursiveTask
  - Basically, use invoke() once to start the algorithm and then fork() or compute() for the recursive calls.
- The join method also returns the value returned by compute in the thread that is being joined to.



Final Version (Missing Imports)

```
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 fjPool = new ForkJoinPool();
int sum(int[] arr){
  return ForkJoinPool.commonPool().invoke
          (new SumArray(arr,0,arr.length));
}
```

33



Getting Good Results in Practice

- Sequential threshold
  - Experiment with 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
  - Put your computations in a loop to see the “long-term benefit” with multiple timings
- More cores is better
  - Overhead may dominate at 4 processors
- Beware memory-hierarchy issues

(*) Parallel Programming with Java

35

Maps and Reductions

- The “workhorses” of parallel programming
- Reductions: take collection of data items (in an array) and reduce the info down to a single result.
  - Reductions can work in || to compute answers for two halves recursively and in || and merge these to produce a result.
  - To use, need opns that are associative (a+b = b+a).
  - Note: (Recursive) results don’t have to be single numbers or strings They can be arrays or objects with multiple fields
- Fold: start with an *initial value* and keep updating “answer so far” by applying some binary function *m* to the current answer and the next element of the array.
  - Without any additional information about what m computes, this algorithm cannot be effectively parallelized since we cannot process arr[i] until we know the answer from the first i-1 iterations of the for-loop.
- Map: performs an opn on each input element independently; given an array of inputs, it produces an array of same length outputs.
  - No combining results
  - E.g. vector addition

Other Data Structures

- Div and conq ||ism requires that efficiently (ideally O(1) time) divide problem into smaller pieces.
  - For arrays, dividing problem involves O(1) arithmetic on indices, hence why it’s used.
- Balanced trees (AVL, B for example) also support || algos well.
  - For example, with a binary tree, we can fork to process the left child and right child of each node in ||.
    - For good sequential cut-offs, it helps to have stored at each tree node the number of descendants of the node, something easy to maintain.
    - However, for trees with guaranteed balance properties, other info— like the height of an AVL tree node, suffices.
- Certain tree problems will not run faster with parallelism.
  - For example, searching for an element in a balanced binary search tree takes O(log n) time with or without parallelism.
- Maps and reduces over balanced trees benefit from parallelism.
  - For example, summing the elements of a binary tree takes O(n) time sequentially where n is the number of elements, but with a sufficiently large number of processors, the time is O(h), where h is the height of the tree.
- Hence, tree balance is even more important with parallel programming: for a balanced tree h = O(log n) compared to the worst case h = O(n).
- || algorithms over regular linked lists are typically poor.
  - Any problem that requires reading all n elements of a linked list takes O(n) time regardless of how many processors are available.

- Streams of input data, such as from files, have the same limitation: it takes linear time to read the input and this can be the bottleneck for the algorithm
    - Can still be benefit to parallelism with such “inherently sequential” data structures and input streams.
        - Suppose had a map operation over a list but each operation was itself an expensive computation each map operation took time O(x) and the list had length n, doing each operation in a separate thread (assuming, again, no limit on the number of processors) would produce an O(x + n) algorithm compared to the sequential algorithm.
        - For simple operations like summing or finding a maximum element, O(xn) there would be no benefit

Parallel Algorithms

Analysis of Fork-Join Algorithms

- FJ || algo should be *efficient* and *correct*.
    - For efficiency – focus on asymptotic bounds and an unfixed # of processors.
    - Size of prob, *n*, and *P*, number of processors factor into asymptotic running time
- FJ framework gives optimal expected-time bound for any *P*.

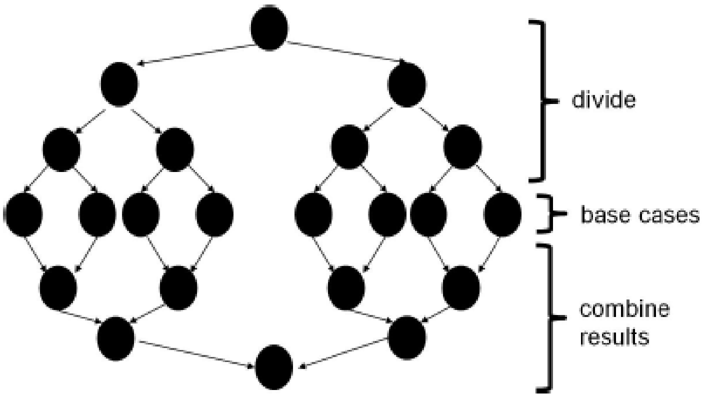
WORK AND SPAN

- T_p: time prog/algo takes to run if there's *P* processors available during its execution.
    - Think of T_p in terms of T_1 and T_∞.
- Work(T_1): how long it takes to run one processor.
    - Total run time of all pieces of algo.
        - Only one processor (thus no ||ism) to do it.
    - In terms of FJ, T_1 does one side of the fork, then the other.
    - Total T_1 does NOT depend on how work is scheduled.
- Span(T_∞): how long it takes to run on unlimited # processors
    - aka critical path length/computational depth.

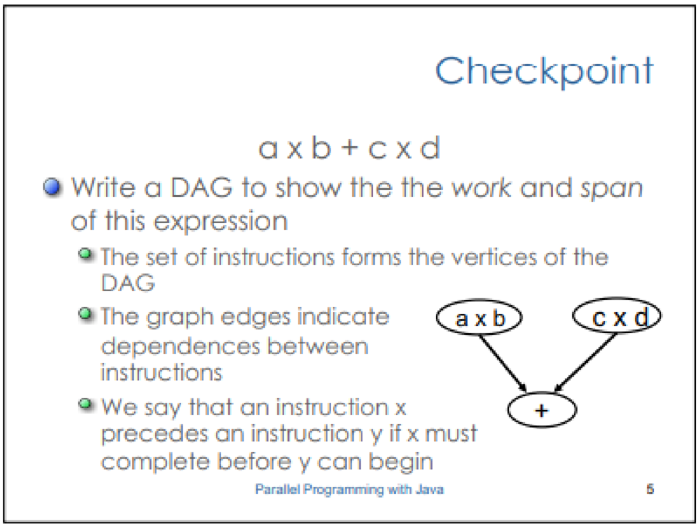
DAG

- Prog execution can be described as Directed Acyclic Graph (DAG):
    - Note that the DAG is NOT a data strc that gets built by the prog.
    - Nodes = pieces of work prog performs.
        - Each node is constant O(1) amount of work performed sequentially.
        - T_1 is # nodes in DAG.
    - Edges = computational depth.
        - Edges represent a source node that must compete before target node begins.
        - Computational dependency along edge.
        - T_∞: with unlimited processors, can start every node as soon as it's ancestors in graph finish.
            - Thus, it's just the length of longest path in DAG.
    - Fork: 'ends a node" and makes two outgoing edges
    - Join: "ends a node" and makes node with two incoming edges.
- With basic fork-join divide-and-conquer parallelism, the execution dags are:
    - The O(1) work to set up two smaller subproblems is one node in the dag.
        - Node has two outgoing edges to two new nodes that start doing the two subproblems.
            - One subproblem might be done by the same thread, but this is not important because ***nodes aren't threads***; they are O(1) pieces of work.)
    - The two subproblems will lead to their own dags.

  - When we join on the results of the subproblems, that creates a node with incoming edges from the last nodes for the subproblems.
    - This same node can do an O(1) amount of work to combine the results.
      - If combining results is more expensive, then it needs to be represented by more nodes.
- DAG for basic || reduction:



  - Root node represents computation that divides array into two equal halves.
  - Bottom node represents computation that adds together the two sums from the halves to produce the final answer.
  - Base cases represent reading from a one-element range assuming no sequential cut-off.
    - Sequential cut-off “just” trims out levels of the dag, which removes most of the nodes but affects the dag’s longest path by “only” a constant amount.
- Parallel reduction thus described by two balanced binary trees - their size proportional to input data size.
  - T_1 is O(n) -there’s approx 2^n nodes.
  - T_∞ is O(log n) – height of each tree is approx. log n.
  - Work in nodes in top half is to create two subproblems, work in nodes in bottom half is to combine two results.



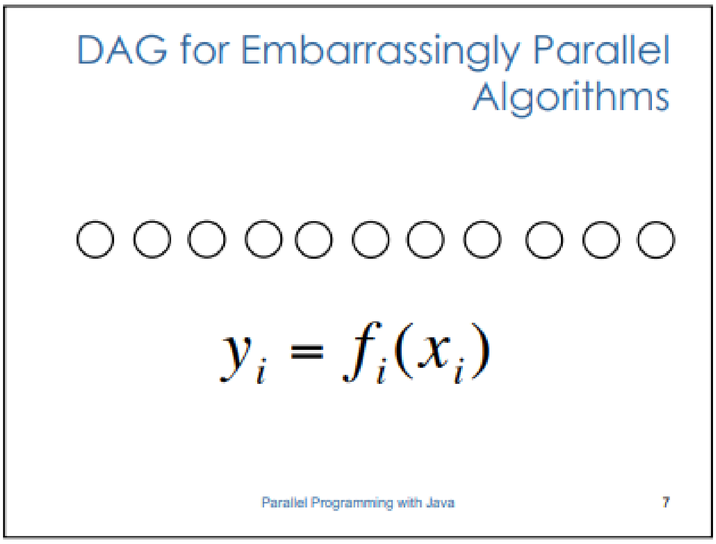
Checkpoint

a x b + c x d

- Write a DAG to show the the *work* and *span* of this expression
  - The set of instructions forms the vertices of the DAG
  - The graph edges indicate dependences between instructions
  - We say that an instruction x precedes an instruction y if x must complete before y can begin

Embarrassingly Parallel Algorithms

- Ideal || case:
  - Computation can be divide into completely separate tasks, each of which can be executed by single processor.
  - No special algorithms or techniques required to get a workable solution
- Examples: Element-wise linear algebra; Image processing; Encryption, compression
- DAG for this "embarrassingly || algos" has no connected nodes – work can be done in any order.



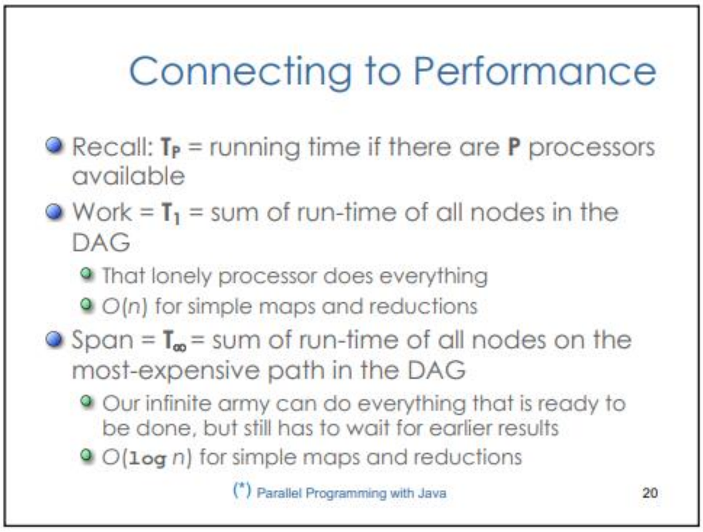
DAG for Embarrassingly Parallel Algorithms

$$y_i = f_i(x_i)$$

Parallel Programming with Java 7

Speedup and Parallelism

- Speedup on *P* processors is T_1/T_P.
  - The ratio of how much faster the prog runs given extra processors.
    - E.g. T1= 20 secs and T4=8, then speedup for P = 4 is 2.5
- A linear speedup of *P* for Tp is rare because of the overhead of creating threads and communicating answers amongst them, memory-hierarchy issues and computational dependencies related to the span.
  - Perfect linear speedup (a unicorn) is when doubling *P* cuts the run time in half – but this is not the limit of speedup.
- T_1/ T_∞ is parallelisation of an algo.
  - Measures how much improvement possible, should be at least as great as speed-up for any *P*.
  - For parallel reductions, parallelism is work/span => O(n/log n) i.e. exponential parallelism.
    - *n* grows faster than *log n*.
    - Means that with enough processors, can hope for exponential speed-up over sequential version.



Connecting to Performance

- Recall: $\mathbf{T_P}$ = running time if there are **P** processors available
- Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG
  - That lonely processor does everything
  - $O(n)$ for simple maps and reductions
- Span = $\mathbf{T_\infty}$ = sum of run-time of all nodes on the most-expensive path in the DAG
  - Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
  - $O(\log n)$ for simple maps and reductions

(*) Parallel Programming with Java

20

FORKJOIN FRAMEWORK BOUND

- Div and conq algos expected time bound:
  - Tp is O(T1/P+T∞)
    - Expected because internally framework uses randomness, so bound can be violated due to “bad luck” (bad luck is very unlikely, so won’t occur in practice).
  - Ignoring constant factors, bound is optimal: given P processors, no framework can do better than T1/P or better than T∞
    - For small P, T1/p term dominates can expect roughly linear speed-up.
      - As P grows, span becomes more important and run-time limit more influenced by T∞.
- Bounds only hold under couple assumptions:
  - All threads created to do subproblems can do approx same amount of work.
  - All threads do a small (not tiny) amount of work.
    - Helps with load balancing.

Overview

- Pick a good || algo
- Implement it in terms of div-and-conq with reasonable sequential cutoff
- Analyse expect run time in terms of the bound.

Divide and Conquer Parallel algorithms

FORK-JOIN ALGORITHMS

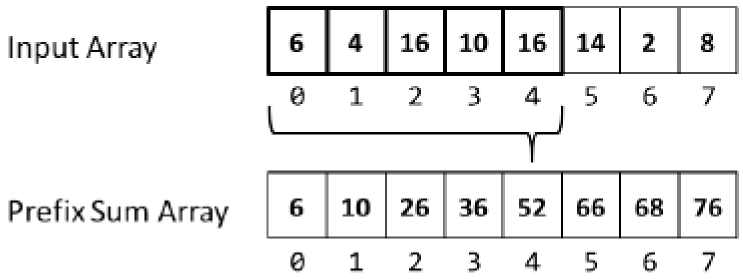
- Inherently sequential probs can have efficient || algos.
- F and J very flexible but div and conq use them basically.
- Can use || algo techniques as building blocks for other larger || algos.
- Can use asymptotic complexity to help decide when one || algo is better than another.

MORE GENERALLY

- Applying the Algorithm Pattern:
    - What should the recursive tasks return?
    - How should we merge the results?
    - Use operations with the same characteristics as summation, and with a different base case

PARALLEL PREFIX SUM

- Problem: Given an array of n integers input, produce an array of n integers output where output[i] is the sum of the first i elements of input.
    - Computing the sum of every prefix of the input array and returning all the results
- This is the prefix-sum problem.



- Efficient || solution for prob.
- Algo overall is O(n) work and O(log n) span.
- Use 2 passes, each with O(n) work and O(log n) span
- Prefix sum pattern helps understand general || prefix pattern.
- Algo works in two passes: up and down

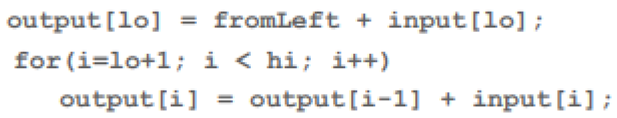
Textbook Explanation	James' Explanation
Up Pass	
• "Up" builds a binary tree from bottom to top – can be produced via FJ computation ▪ Every node holds sum of integers for some range on input array. ▪ Root holds sum for entire range	• Build a binary tree where ○ Root has sum of the range [x,y) ○ If a node has sum of [lo,hi) and hi>lo, ▪ Left child has sum of [lo,middle) ▪ Right child has sum of [middle,hi) ▪ A leaf has sum of [i,i+1), i.e., input[i]

[0,n) (range includes left but excludes right end). ▪ Node's left child holds sum for left half of node's range; node's right child holds sum for right half of node's range. • E.g. root's left child is for range [0,n/2) and root's right child is for range [n/2,n). ▪ Leaves of tree hold sum for 1-element ranges. • There are n leaves • In practice, use a sequential cutoff and have leaves store the sum for a range of x	• This is a fork-join computation: combine results by actually building a binary tree ○ Tree built bottom-up in|| • Could be more clever with an array, as with heaps Analysis: O(n) work, O(log n) span

elements. • To build tree, use FJ computation: ○ Overall goal – produce node for range [0,n). ○ To build node for range [x,y): ▪ If x == y-1, produce node holding input[x]. ▪ Else recursively in || build nodes for [x,(x+y)/2) and [(x+y)/2,y) ; make these left and right children of result node. • Add their answers together for result node's sum. • Basically, result of div-and-conquer is tree node and to "combine results" is to use two recursive results and subtrees.	
Down Pass	
• "Down" pass: ○ Use tree to compute prefix-sum. ○ Process tree from top to bottom, passing "down" as an arg the sum of the	• Pass down a value fromLeft ○ Root given a fromLeft of 0 ○ Node takes its fromLeft value and ▪ Passes its left child the same fromLeft

array indices to left of the node. o Arg passed to root is 0 – no #s to left of range [0,n), so their sum is zero. o Arg passed to node's left child is same as arg passed to node – sum of numbers to left of range [x,(x+y)/2) is sum of numbers to left of range [x,y). o Arg passed to node's right child is arg passed to node plus sum stored at node's left child – sum of numbers to left of range [(x+y)/2, y) is sum to left of x plus some of range [x,(x+y)/2). • When reach leaf, have output[i] is putput[i] plus val passed down to ithleaf. o With a sequential cutoff, then have range of output vals to produce at each leaf. o Val passed down is still just what's needed for sum of all #s to left of range. o Sequential computation can produce all output vals for the range at each leaf proceeding left-to-right through range. • Second pass can also use FJ computation. o Create subprob, just need value being passed down and node it's being passed to. o Just start with value of 0 and tree root.	▪ Passes its right child its fromLeft plus its left child's sum ▪ At the leaf output[i]=fromLeft+input[i] • This is a fork-join computation: traverse the tree from step 1 and produce no result • Leaves assign to output • Invariant: fromLeft is sum of elements left of the node's range Analysis: O(n) work, O(log n) span

Sequential Cutoff	

- Above algo assumes no sequential cut-off – with cut-off, stop recursion when y-x is below cut-off and create one node that holds the sum of the range [x,y) computed sequentially.	- Up:
	  ○ Just a sum, have leaf node hold the sum of a range
	- Down:
	
	`output[lo] = fromLeft + input[lo];`
	`for(i=lo+1; i < hi; i++)`
	`   output[i] = output[i-1] + input[i];`

UP PASS

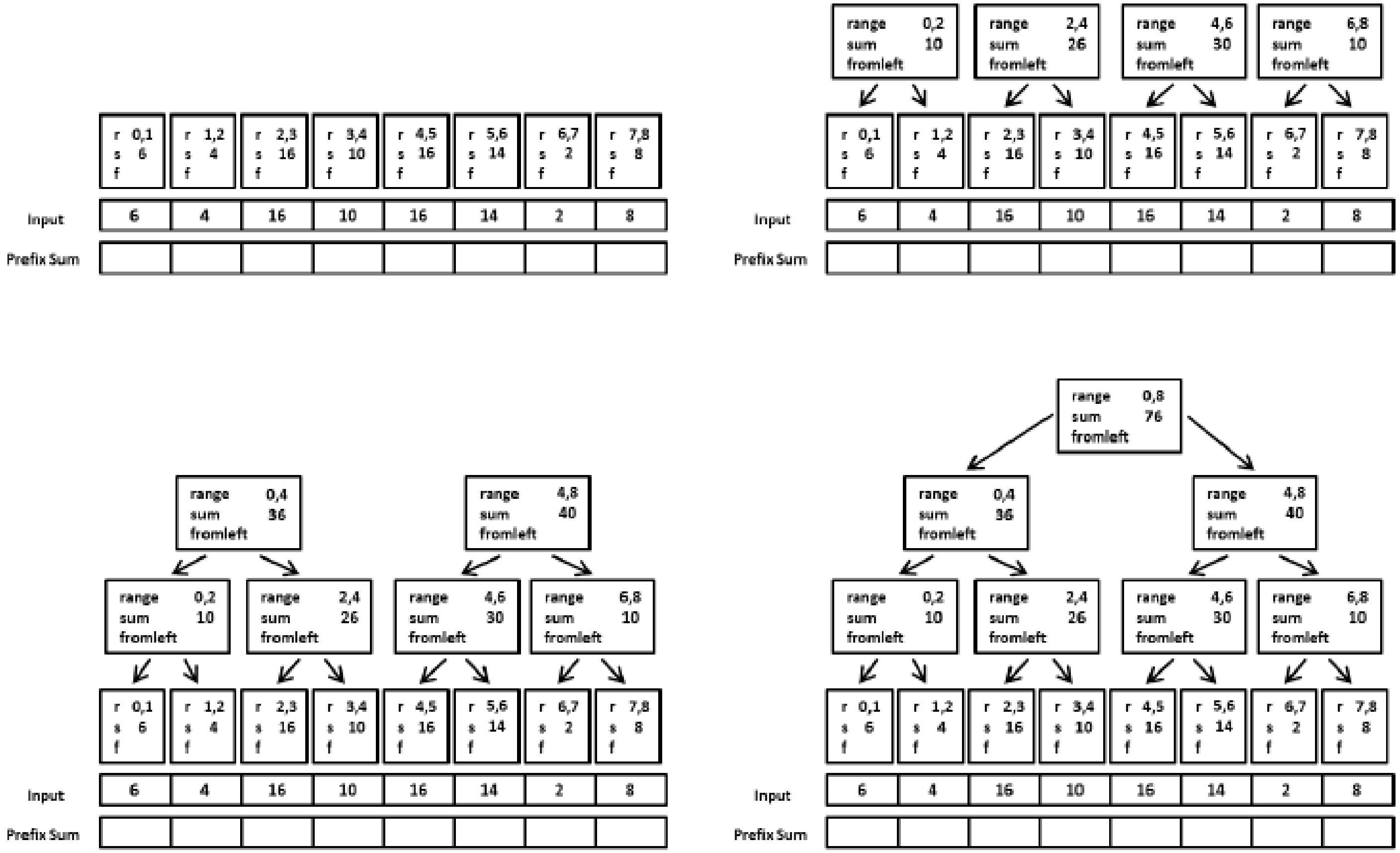
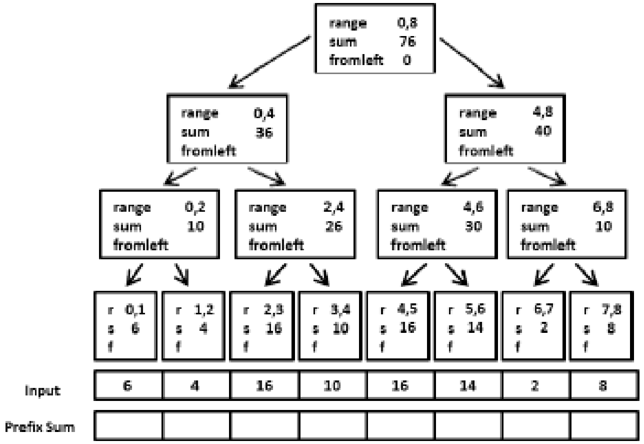
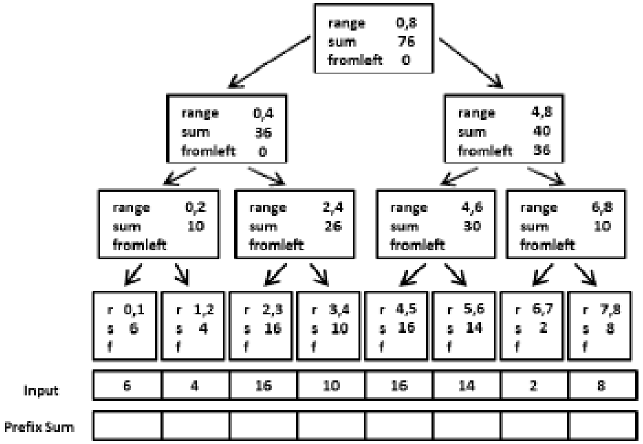
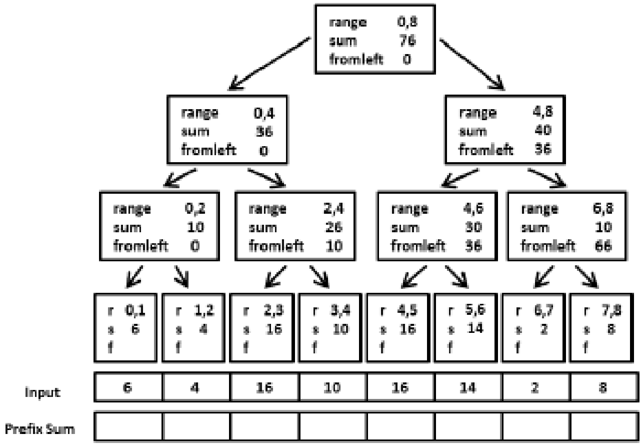


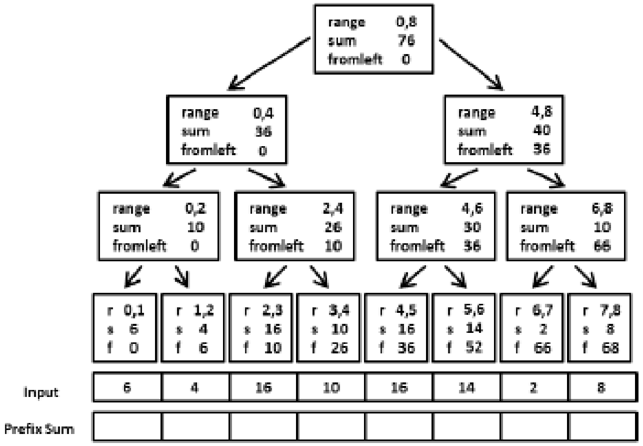
Figure 9: Example of the first pass of the parallel prefix-sum algorithm: the overall result (bottom-right) is a binary tree where each node holds the sum of a range of elements of the input. Each node holds the index range for which it holds the sum (two numbers for the two endpoints) and the sum of that range. At the lowest level, we write r for range and s for sum just for formatting purposes. The fromleft field is used in the second pass. We can build this tree bottom-up with $\Theta(n)$ work and $\Theta(\log n)$ span because a node's sum is just the sum of its children.

DOWN PASS


range 0,8 sum 76 fromleft 0
range 0,4 sum 36 fromleft
range 4,8 sum 40 fromleft
range 0,2 sum 10 fromleft
range 2,4 sum 26 fromleft
range 4,6 sum 30 fromleft
range 6,8 sum 10 fromleft
r 0,1 s 6 f
r 1,2 s 4 f
r 2,3 s 16 f
r 3,4 s 10 f
r 4,5 s 16 f
r 5,6 s 14 f
r 6,7 s 2 f
r 7,8 s 8 f
Input
6
4
16
10
16
14
2
8
Prefix Sum


range 0,8 sum 76 fromleft 0
range 0,4 sum 36 fromleft 0
range 4,8 sum 40 fromleft 36
range 0,2 sum 10 fromleft
range 2,4 sum 26 fromleft
range 4,6 sum 30 fromleft
range 6,8 sum 10 fromleft
r 0,1 s 6 f
r 1,2 s 4 f
r 2,3 s 16 f
r 3,4 s 10 f
r 4,5 s 16 f
r 5,6 s 14 f
r 6,7 s 2 f
r 7,8 s 8 f
Input
6
4
16
10
16
14
2
8
Prefix Sum


range 0,8 sum 76 fromleft 0
range 0,4 sum 36 fromleft 0
range 4,8 sum 40 fromleft 36
range 0,2 sum 10 fromleft 0
range 2,4 sum 26 fromleft 10
range 4,6 sum 30 fromleft 36
range 6,8 sum 10 fromleft 66
r 0,1 s 6 f
r 1,2 s 4 f
r 2,3 s 16 f
r 3,4 s 10 f
r 4,5 s 16 f
r 5,6 s 14 f
r 6,7 s 2 f
r 7,8 s 8 f
Input
6
4
16
10
16
14
2
8
Prefix Sum


range 0,8 sum 76 fromleft 0
range 0,4 sum 36 fromleft 0
range 4,8 sum 40 fromleft 36
range 0,2 sum 10 fromleft 0
range 2,4 sum 26 fromleft 10
range 4,6 sum 30 fromleft 36
range 6,8 sum 10 fromleft 66
r 0,1 s 6 f 0
r 1,2 s 4 f 6
r 2,3 s 16 f 10
r 3,4 s 10 f 26
r 4,5 s 16 f 36
r 5,6 s 14 f 52
r 6,7 s 2 f 66
r 7,8 s 8 f 68
Input
6
4
16
10
16
14
2
8
Prefix Sum

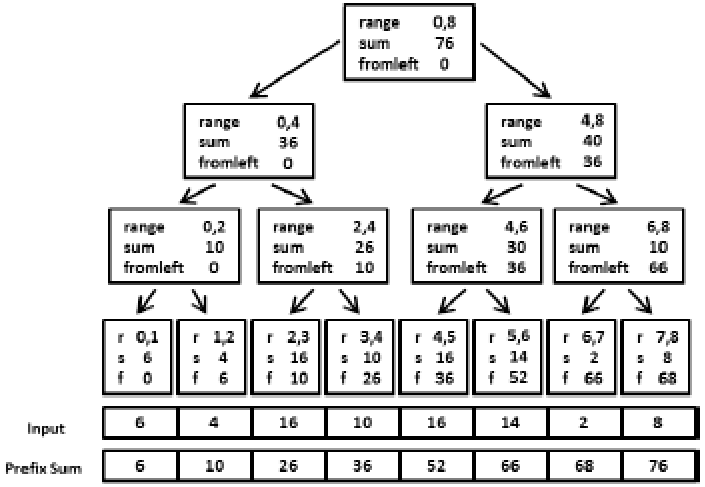
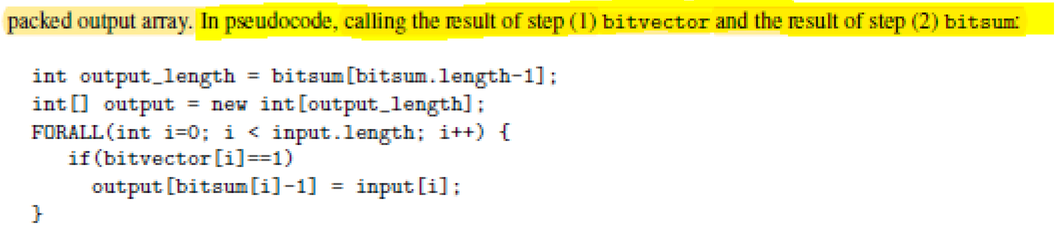


Figure 10: Example of the second pass of the parallel prefix-sum algorithm. Starting with the result of the first pass and a "fromleft" value at the root of 0, we proceed down the tree filling in fromleft fields in parallel, propagating the same fromleft value to the left-child and the fromleft value plus the left-child's sum to the right-value. At the leaves, the fromleft value plus the (1-element) sum is precisely the correct prefix-sum value. This pass is $\Theta(n)$ work and $\Theta(\log n)$ span.

PACK

- || algo for this problem: Given an array input, produce an array output containing only those elements of input that satisfy some property, and in the same order they appear in input.
    - Length of output is unknown in advance but never longer than input.
    - For example if the property is, “greater than 10” and input is {17,4,6,8,11,5,13,19,0,24}, then output is {17,11,13,19,24}.
- This is a pack pattern operation.
    - Useful for || version of quicksort.
- Finding which elements should be part of output is a trivial map operation.
    - But knowing what output index to use for each element requires knowing how many elements to the left also satisfy that property.
        - E.g. knowing how many elements to the left are also greater than 10.
- A prefix computation can do this.
- Pack algo has O(n) work and O(log n) span.
    - 1. Perform || map to produce *bit vector* where 1 indicates the corresponding input element satisfies the property e.g. is greater than 10.
        - For {17,4,6,8,11,5,13,19,0,24} ; this step produces {1,0,0,0,1,0,1,1,0,1}.
    - 2. Perform || prefix sum on the bit vector produced in step 1.
        - For the example, this produces {1,1,1,1,2,2,3,4,4,5}.
    - 3. Array produced in step 2 provides info that a final || map needs to produce the packed output array.



packed output array. In pseudocode, calling the result of step (1) bitvector and the result of step (2) bitsum:

```
int output_length = bitsum[bitsum.length-1];
int[] output = new int[output_length];
FORALL(int i=0; i < input.length; i++) {
   if(bitvector[i]==1)
     output[bitsum[i]-1] = input[i];
}
```

-
    - Note: possible to do step 3 using only bitsum and not bitvector – allows step 2 to do prefix sum in place, updating bitvector array.
    - Either way, each FORALL loop iteration either doesn’t write anything or writes to different element of output than every other iteration.
- Comments:
    - First two steps can be combined into one pass
        - Just using a different base case for the prefix sum
    - Can also combine third step into the down pass of the prefix sum

PARALLEL QUICKSORT

Sequential Quicksort

1. Pick pivot element, O(1).
2. Partition data into: [O(n)]
    a. Elements less than pivot
    b. Pivot
    c. Elements greater than pivot
3. Recursively sort elements less than pivot

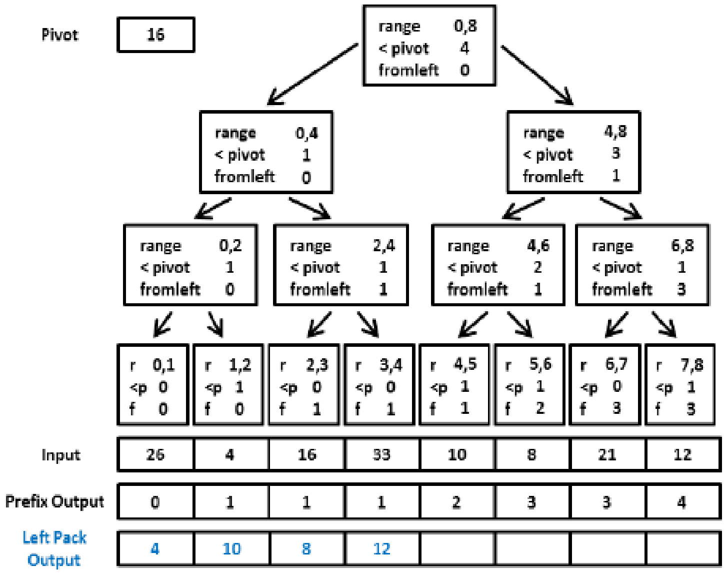
4. Recursively sort elements greater than pivot

- Assume partition is roughly balanced – means that two recursive calls solve problems of approx half the size.
    - R(n) is run time of prob of size n. (Note: R(n) notation used to avoid confusion with work and span T notation; T(n) is actual "correct" notation).
        - Then, except for base case of recursion (finishes in O(1) time), have R(n) = O(n) + 2R(n/2); due to O(n) partition and two probs half the size
        - If run time is O(n) + 2R(n/2), then this is O(n log n).
    - If pivots chosen randomly, expected running time remains O(n log n).
- Quicksort is sequential and in-place.

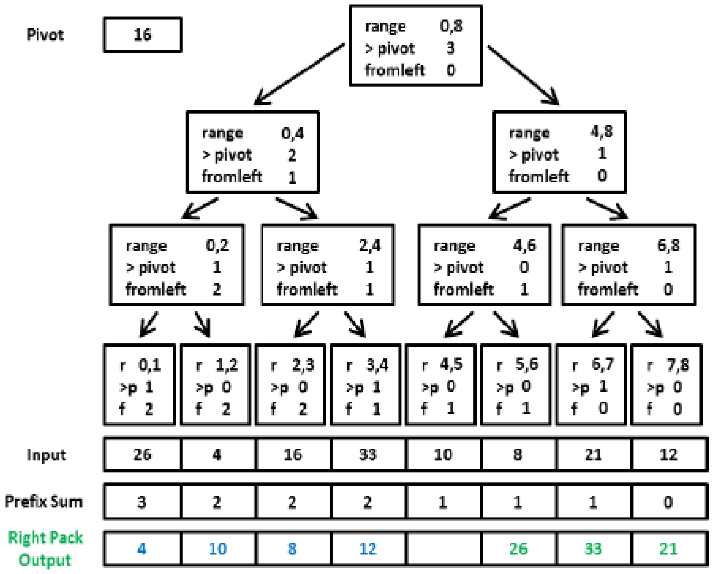
Parallelising Quicksort

- Steps 3 and 4 (recursively sorting elements) can be done in ||.
    - Has no effort on work.
    - Changes span to R(n) = O(n) + 1R(n/2) because can solve two problems of half the size simultaneously (which is O(n)).
- ||ism T1/T∞ is O(n log n)/O(n) i.e O(log n) – not expected exponential speedup.
    - To do better, must ||ise partition creation step.
    - Sequential Quicksort swaps data elements in place – this must be sacrificed for ||ism.
        - Good trade-off (Amdahl's Law): use extra space to achieve extra additional ||ism.
        - Only need one more array of same length as input array.
- Partition into new extra array:
    - Need 2 pack opns for left and right side of array.
        - Pack all elements < pivot into lhs of array – if x elements are less than the pivot, put this data at positions 0 to x -1.
        - Pack all elements > pivot into rhs of array – if x elements are greater than pivot., put this data at positions n -x to n -1.
            - This step works down from end of array instead of from up beginning (step 1 went from the bottom).
- After doing both steps, there's one spot left for pivot between two partitions.

- An example:

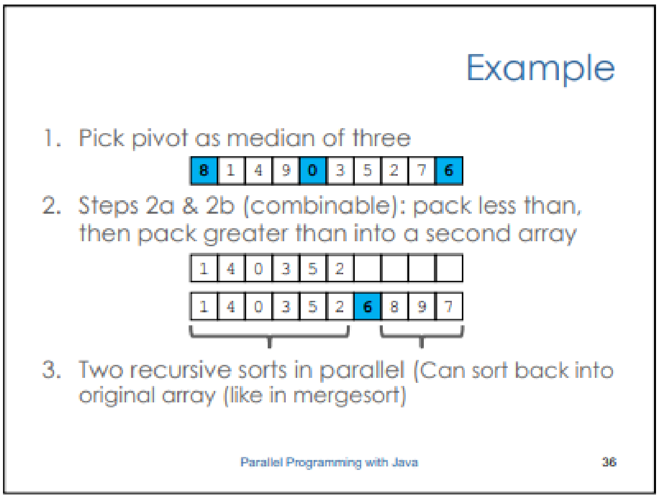


Input	26	4	16	33	10	8	21	12
Prefix Output	0	1	1	1	2	3	3	4
Left Pack Output	4	10	8	12				



Input	26	4	16	33	10	8	21	12
Prefix Sum	3	2	2	2	1	1	1	0
Right Pack Output	4	10	8	12		26	33	21

- Each of the two pack opns is O(n) work and O(log n) span.
  - Look at each element twice – once to decide if it's less than the pivot and once to decide if it's greater than the pivot
    - This is only constant factor more work.
- Two pack opns can be performed in ||.
- After completing partition, continue with recursive sorting (in ||) of the two sides.
  - Next step of recursion can reuse original array.
- Re-analysis of asymptotic complexity of || quicksort using || (and not in place) partition.
  - Assume pivots always divide problems exactly in half.
  - Work is still R(n) = O(n) + 2R(n/2) = O(n log n) – O(n) now includes two pack opns.
  - Span now R(n) = O(log n) + 1R(n/2) – span for pack operations is O(log n)
    - This actually works out to be O($\log^2$ n) – not as good as O(log n), BUT better than O(n).
  - Hence, available ||ism is proportional to n log n/$\log^2$ n = n/log n – exponential speedup (pg 42 textbook).



Example

1. Pick pivot as median of three

8	1	4	9	0	3	5	2	7	6

2. Steps 2a & 2b (combinable): pack less than, then pack greater than into a second array

1	4	0	3	5	2				
1	4	0	3	5	2	6	8	9	7

3. Two recursive sorts in parallel (Can sort back into original array (like in mergesort)

Parallel Programming with Java 36

NB THIS NEXT SECTION WAS NOT MENTIONED IN JAMES' NOTES

PARALLEL MERGESORT

Sequential Mergesort

1. Recursively sort the left half and right half of the input.
2. Merge the sorted results into a new sorted array by repeatedly moving the smallest not-yet-moved element into the new array.

- Run time for this algo is R(n) = 2R(n/2)+ O(n) because there's two subproblems of half the size, and merging is O(n) using a loo g that progresses through the two sorted recursive results.
- To ||ise:
  - Do two recursive sorts in || - this has no effect on work and reduces span to R(n) = 1R(n/2) + O(n) = O(n).
    - ||ism thus O(log n).
- For better ||ism, need to ||ise merge opn.
  - Algo takes two sorted subarrays (length x and y) and merges them.
  - In sequential mergesort, two lengths are equal/almost equal.
  - Recursive || meging creates subproblems that may need to merge arrays of different lengths.
- The algo for this:
  - Determine median element of larger array.
    - This opn cost O(1)
  - Use binary search to find position j in smaller array such that all elements to left of *j* are less than larger array's median.
    - Binary search is O(log m) where m is length of smaller array.
  - Recursively merge half of larger array with positions 0 to j of smaller array.
  - Recursively merge half of larger array with the rest of the smaller array.
- Total # elements this algo merges is x+y, which will be called *n*.
- First two steps are O(log n) since n is greater than the length of the array on which binary search is done.
  - Leaves two subproblems which aren't necessarily size of n/2.

    - Best case is when binary search ends up in middle of smaller array.
    - Worst case is when it ends up at one extreme i.e. all elements of the smaller array are less than the median of the larger array OR all elements of the smaller array are greater than the median of the larger array
        - This scenario isn't too bad - larger array has at least n/2 elements (otherwise it wouldn't be larger).
        - Always split larger array's elements in half for recursive subproblems.
        - So, each subproblem has at least n/4 (half of n/2) elements.
        - Worst case split is n/4 and 3n/4.
        - This is "good enough" for a large amount of ||ism to merge.
- Analysis of algorithm:
    - The worst-case split is n/4 and 3n/4, so the worst-case span is R(n) = R(3n/4) + O(log n).
        - R(n/4) does not appear because it can be done in parallel with the R(3n/4) and is expected to finish first (and the O(log n) is for the binary search).
    - R(n) = R(3n/4) + O(log n) works out to be O(log2 n).
    - Work is R(n) = R(3n/4) + R(n/4) + O(log n), which works out to be O(n).
    - This analysis was just for the merging step.
- Adding O(log2 n) span and O(n) work for merging back into the overall mergesort algorithm, we get a span of R(n) = 1R(n/2) + O(log2 n), which is O(log3 N), and a work of R(n) = 2R(n/2) + O(n), which is O(n log n).
- While the span and resulting parallelism is O(log n) worse than for parallel quicksort, it is a worst-case bound compared to quicksort's expected case.

Parallel Performance

Analyzing Parallel Algorithms

- Like all algorithms, parallel algorithms should be:
  - Correct
  - Efficient

Characteristics of Work and Span

- Work Law: Tp>= T1 / P
  - Each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time.
  - Thus, to do all the work on P processors, it must take at least T1/P time.
  - Ignoring memory hierarchy issues
- Span Law: Tp >= T∞
  - A finite number of processors cannot outperform an infinite number of processors, because the infinite-processor machine could just ignore all but P of its processors and mimic a P-processor machine exactly

SPEEDUP

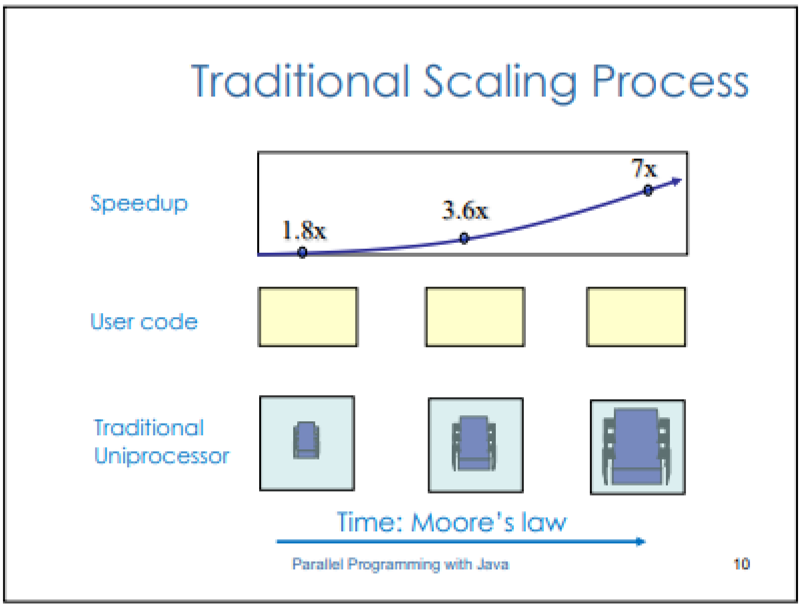
- Speedup for P processors = time for 1 processor/time for P processors = T1/Tp
- In the ideal situation, as P increases, so Tp should decrease by a factor of P
- Often better to report Speedup than Tp.

SCALABILITY

- Scalability: the speed-up of a program as the number of processors increases
- Perfect linear speedup = P (doubling P halves running time)
  - Usually our goal; hard to get in practice
  - Most algorithms deteriorate after a certain point
- An algorithm is termed scalable if the level of parallelism increases at least linearly with the problem size
  - i.e. Does not deteriorate markedly

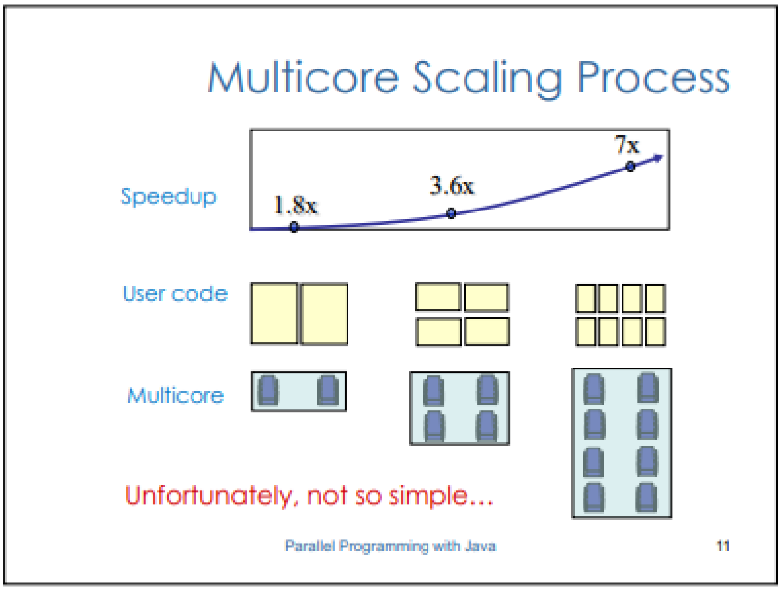
Parallelism

- Parallelism is the maximum possible speed-up: T1/T∞
  - Work/Span
  - At some point, adding processors won't help
  - What that point is depends on the span
- Parallel algorithms are about decreasing span without increasing work too much



Traditional Scaling Process

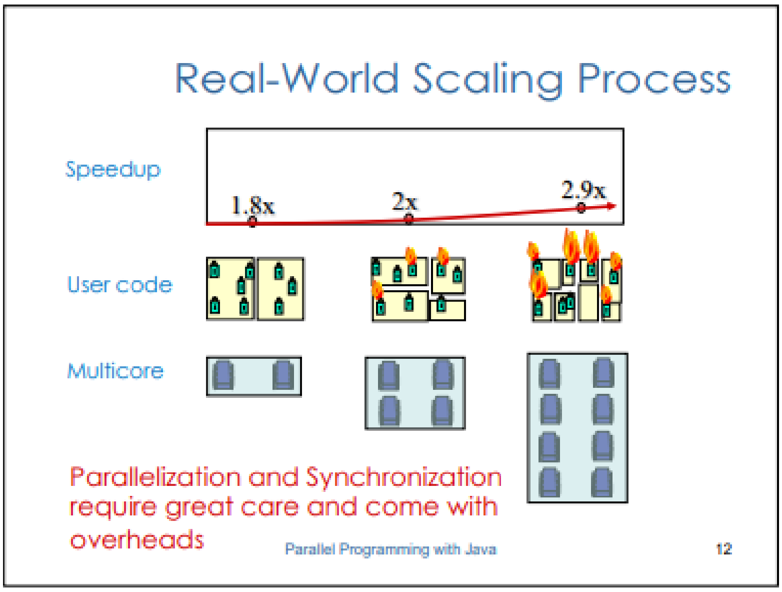
Parallel Programming with Java 10



Multicore Scaling Process

Unfortunately, not so simple…

Parallel Programming with Java 11

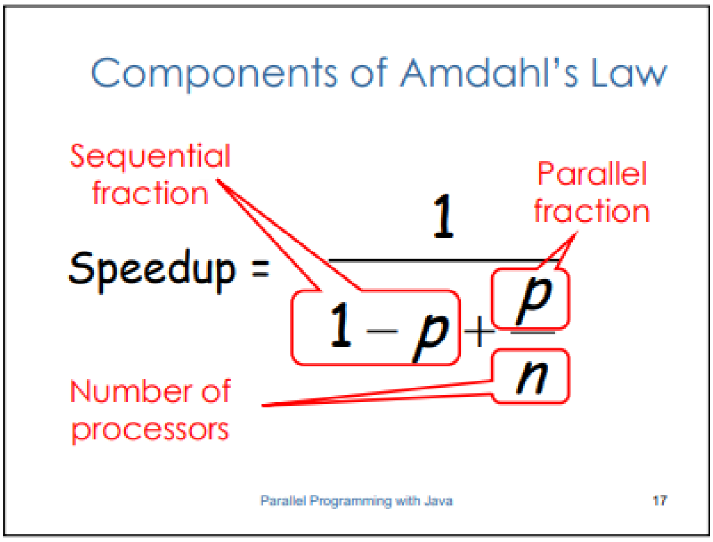


Real-World Scaling Process

Parallelization and Synchronization require great care and come with overheads

Parallel Programming with Java 12

Amdahl's Law



Components of Amdahl's Law

Sequential fraction

Parallel fraction

$$\text{Speedup} = \frac{1}{\boxed{1-p} + \frac{\boxed{p}}{\boxed{n}}}$$

Number of processors

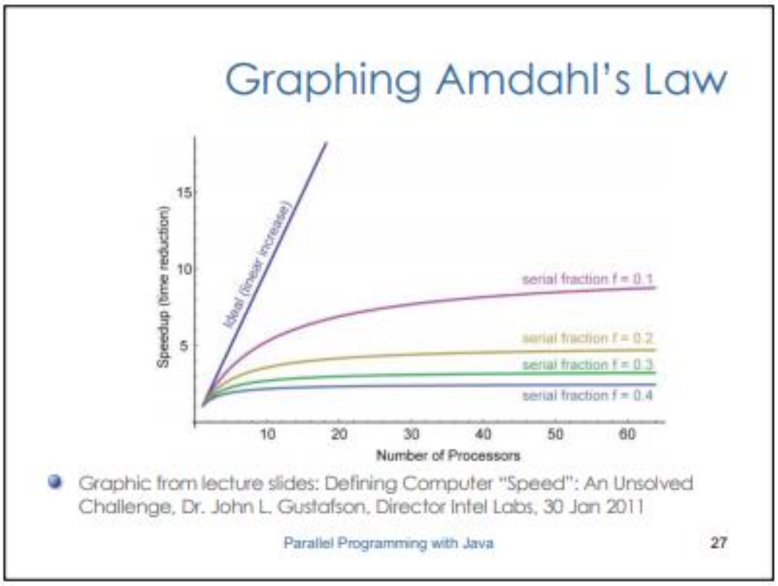
Parallel Programming with Java

17

- || algo can have sequential parts – these parts have either not yet been ||ised or could be inherently sequential.
  - e.g. reading a linked list, getting input, doing computations where each needs the previous step, etc.
- Typically have parts of programs that parallelize well…
  - E.g. maps/reductions over arrays and trees
  - (call this the parallel fraction, p)
- Even a bit of sequential work in prog drastically reduces speed-up once have a significant # of processors.
- Derivation:
  - Suppose T1 (work) is 1 i.e. total prg execution time on 1 processor is 1 "unit time".
  - Let S be portion of execution that is serial (can't be ||ised).
  - Assume rest of execution (1-S) gets perfect linear speedup on P processors for any P.
  - Then:

    T1= S+(1-S) =1

    Tp = S+(1-S)/P
    - Note: have assumed that || portion (1-S) runs in (1-S)/P time.
  - Then speedup, by definition is:
    - Amdahl's Law: T1/Tp = 1/(S+(1-S)/P).
  - Corollary - ||ism simplified as P goes to ∞

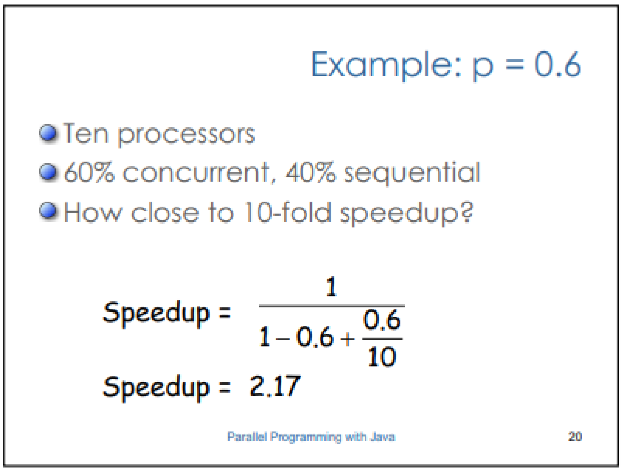


Graphing Amdahl's Law

- Graphic from lecture slides: Defining Computer "Speed": An Unsolved Challenge, Dr. John L. Gustafson, Director Intel Labs, 30 Jan 2011

Parallel Programming with Java 27

- Some examples:



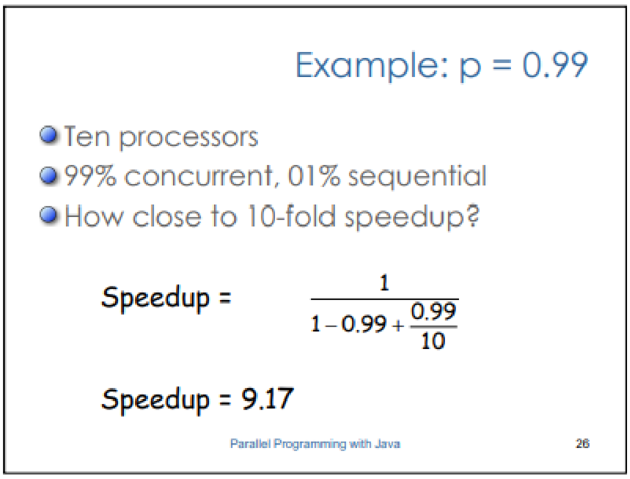
Example: p = 0.6

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = \frac{1}{1-0.6+\frac{0.6}{10}}$$

Speedup = 2.17

Parallel Programming with Java 20



Example: p = 0.99

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = \frac{1}{1-0.99+\frac{0.99}{10}}$$

Speedup = 9.17

Parallel Programming with Java 26

- Problem: expect to get twice the performance from twice the computational resources.
    - If those extra resources are processors, this works only if most of the execution time is still running parallelisable code.
    - Adding 2nd/3rd processors can provide significant speedup – but the benefit diminishes as # of processors grows.
    - Suppose 1/3 of a program is sequential.
        - Then a billion processors won't give a speedup over 3
- ||sim does provide real speed-up for performance-critical parts of progs.

FLAWS IN AMDAHL'S LAW

- Amdahl's law has an assumption that may not hold true:
    - 1. The ratio of TS to TP is not constant for the same program and usually varies with problem size
        - Typically, the TP grows faster that TS
    - 2. The serial algorithm may not be the best solution for a problem
        - Another algorithm may increase the work but reduce the span

WORKAROUNDS TO AMDAHL'S LAW:

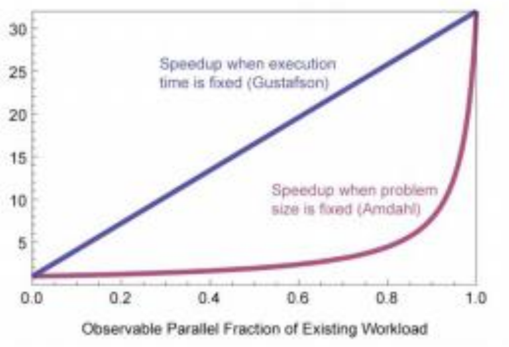
- Can change algorithm – given enough processors, it's worth ||izing something (reducing span) even if it means more total computation (increased work).
- Use ||ism to solve new/bigger probs rather than solving the same prob faster

SCALABILITY REVISITED

- Strong scaling:
    - how the solution time varies with the number of processors for a fixed total problem size.
    - Amdahl's Law applies
- Weak scaling:
    - Defined as how the solution time varies with the number of processors for a fixed problem size per processor
    - i.e., problem size depends on number of processors
    - Gustafson's Law applies

GUSTAFSON'S LAW

- What if time is fixed and work is what changes?



Thread Performance on Multicore Machines

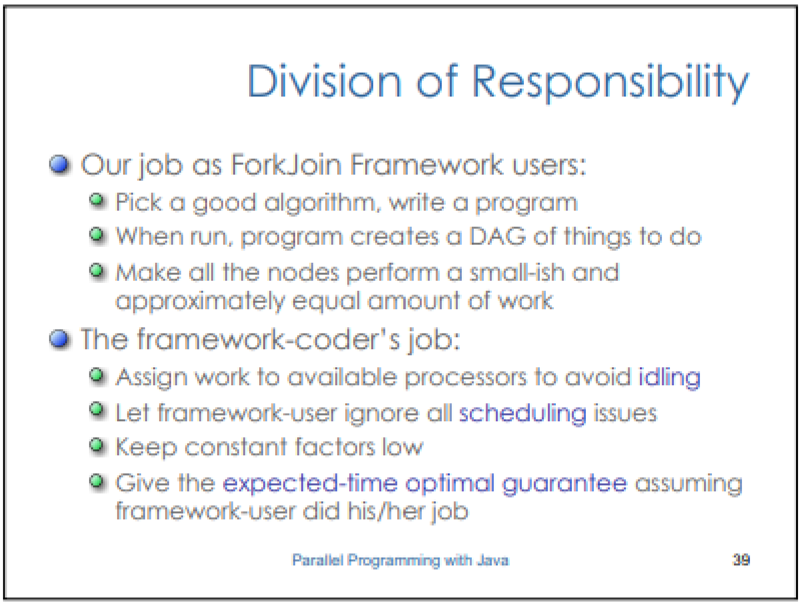
- Optimal scheduling of a multithreaded computation is NP-complete
    - Meaning that it is computationally intractable
- But, practical scheduling algorithms exist that are asymptotically optimal
    - For large inputs, they perform at worst a constant factor (independent of the input size) worse than the best possible algorithm

PERFORMANCE WITH FJ FRAMEWORK

- Have work and span laws.
- The fork-join framework guarantees expected time: Tp <= (T1/P) + O(T∞)
    - First term dominates for small P, second for large P
    - Expected because internally the framework uses randomness, so the bound can be violated from exponentially unlikely "bad luck"
- The FJ Framework gives an expected-time guarantee that is asymptotically optimal!

Implications

- Guaranteed: : Tp <= (T1/P) + O(T∞)
    - Can't beat O(T ∞) by more than a constant factor
    - On P processors can't beat (T1 / P) (ignoring memory-hierarchy issues)
    - Framework on average gets within a constant factor of the best you can do, assuming the user (you) did his/her job
    - So: focus on your algorithm, data structures, and cut-offs, rather than number of processors and scheduling



Division of Responsibility

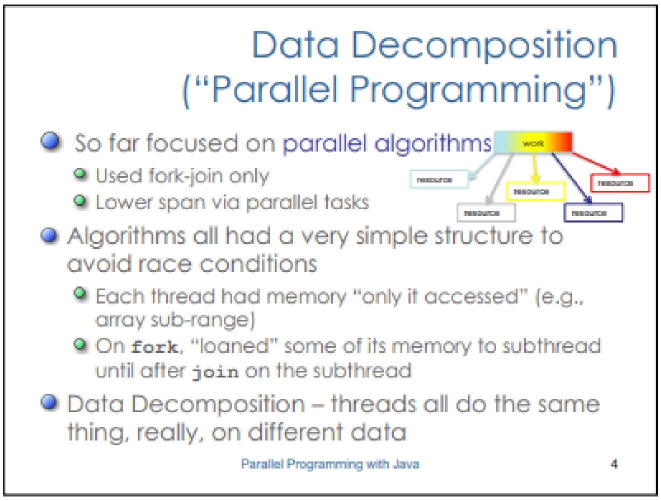
- Our job as ForkJoin Framework users:
  - Pick a good algorithm, write a program
  - When run, program creates a DAG of things to do
  - Make all the nodes perform a small-ish and approximately equal amount of work
- The framework-coder's job:
  - Assign work to available processors to avoid idling
  - Let framework-user ignore all scheduling issues
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did his/her job

Parallel Programming with Java 39

Concurrent Programming

Parallel Programming

- FJ ||ism made it relatively easy to avoid having two threads access the same memory at the same time.
    - Consider a simple parallel reduction like summing an array.
    - Each thread accesses a disjoint portion of the array, so there is no sharing like there potentially is with bank accounts.
    - The sharing is with fields of the thread objects: One thread initializes fields (like the array range) before creating the helper thread.
    - Then the helper thread may set some result fields that the other thread reads after the helper thread terminates.
    - Synchronization here is accomplished entirely via
      (1) thread creation (not calling start or fork until the correct fields are written) and
      (2) join (not reading results until the other thread has terminated).
    - But in our concurrent programming model, this form of synchronization will not work because we will not wait for another thread to finish running before accessing a shared resource like a bank account.



Data Decomposition
("Parallel Programming")

- So far focused on parallel algorithms
    - Used fork-join only
    - Lower span via parallel tasks
- Algorithms all had a very simple structure to avoid race conditions
    - Each thread had memory "only it accessed" (e.g., array sub-range)
    - On **fork**, "loaned" some of its memory to subthread until after **join** on the subthread
- Data Decomposition – threads all do the same thing, really, on different data

Parallel Programming with Java 4

Task Decomposition

- Threads assigned independent tasks
    - Access same resources (rather than implementing the same algorithm)
- Concurrency: Correctly and efficiently managing access to shared resources from multiple (possibly simultaneous) clients
- More complex structure for algorithms
    - Multiple threading doing different things with data
    - e.g. animator threads, updator threads

- Still have threads and shared memory.
    - "Shared resources" will be memory locations (fields of objects) used by more than one thread.
        - Need to write code that provides properly synced access to shared resources even though it may not be known what order the threads access the data.
        - Multiple threads may try to access and/or modify data at same time.
            - This can't be allowed.
- Example of why controlled concurrent access is needed with shared resources:
    - We are writing banking software where we have an object for each bank account. Different threads (e.g., one per bank teller or ATM) deposit or withdraw funds from various accounts.
    - What if two threads try to manipulate the same account at the same time?

Purpose

- Handle issues other than raw speed
- Parallelism: it may well be that a parallel algorithm needs to have different threads accessing some of the same data structures in an unpredictable way.
    - For example, we could have multiple threads search through a graph, only occasionally crossing paths.
- Responsiveness: Many programs, including operating systems and programs with user interfaces, want/need to respond to external events quickly.
    - One way to do this is to have some threads doing the program's expensive computations while other threads are responsible for "listening for" events like buttons being clicked or typing occurring.
    - The listening threads can then (quickly) write to some fields that the computation threads later read.
- Processor utilization: If one thread needs to read data from disk (e.g., a file), this will take a very long time relatively speaking.
    - In a conventional single-threaded program, the program will not do anything for the milliseconds it takes to get the information.
    - But this is enough time for another thread to perform millions of instructions.
    - So, by having other threads, the program can do useful work while waiting for I/O.
    - This use of multithreading is called masking (or hiding) I/O latency.
- Failure/performance isolation: Sometimes having multiple threads is simply a more convenient way to structure a program.
    - In particular, when one thread throws an exception or takes too long to compute a result, it affects only what code is executed by that thread.
    - If we have multiple independent pieces of work, some of which might (due to a bug, a problem with the data, or some other reason) cause an exception or run for too long, the other threads can still continue executing.
    - There are other approaches to these problems, but threads often work well.

Programming Model

- In concurrent programming we have multiple threads that are “largely doing their own thing” but occasionally need to coordinate since they are accessing shared resources.
  - It is like different cooks working in the same kitchen — easy if they are using different utensils and stove burners, but more difficult if they need to share things.
  - The cooks may be working toward a shared goal like producing a meal, but while they are each working on a different recipe, the shared goal is not the focus.
- Basic model comprises of multiple threads that are running in a mostly uncoordinated way.
  - We might create a new thread when we have something new to do.
  - The operations of each thread are interleaved (running alongside, before, after, or at the same time) with operations by other threads.
- For example: we may have 4 threads processing bank account changes as they arrive.
  - While it is unlikely that two threads would access the same account at the same time, it is possible and we must be correct in this case.

Testing/Debugging Concurrent Programs

- Testing and debugging extremely difficult:
  - Memory access highly non-deterministic
  - Exposing race conditions may rely on a particular sequence of events
  - Simultaneous access is rare
  - Number of possible execution sequences can be astronomical
  - Problematic sequences may never occur in the test environment
- Bugs can be very unlikely to occurr.
  - If a program exhibits a bug and you re-run the program another million times with the same inputs, the bug may not appear again.
  - This is because what happens can depend on the order that threads access shared resources, which is not entirely under programmer control.
  - It can depend on how the threads are scheduled onto the processors, i.e., when each thread is chosen to run and for how long, something that is decided automatically

Synchronisation

- In real life we typically use a clock to enforce synchronization constraints
- In computer systems, we often need to synchronize without the benefit of a clock
  - Either because there is no universal clock
  - Or because we don’t know with fine enough resolution when events occur

CONSTRAINTS

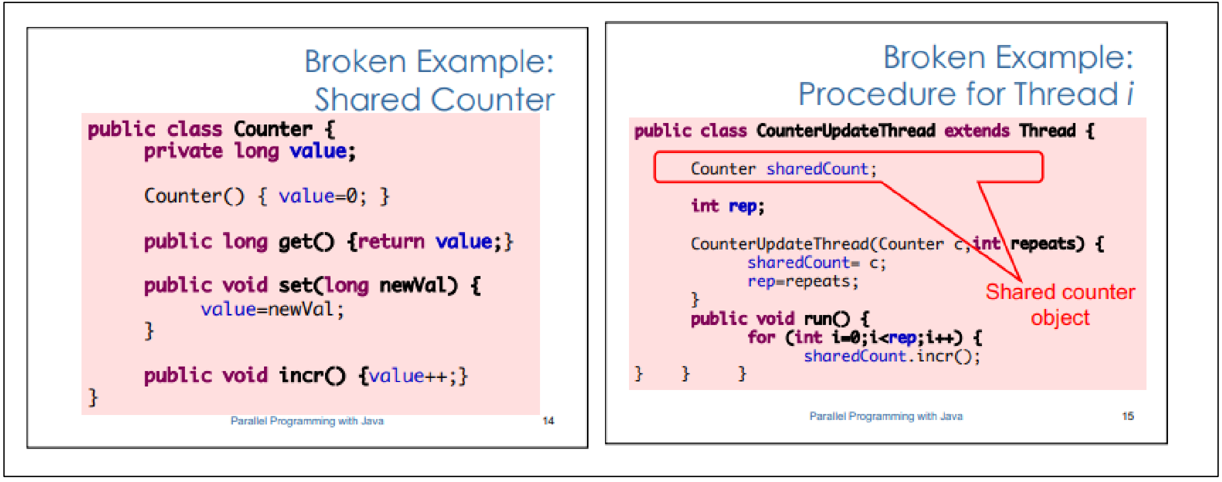
- Synchronization constraints are requirements pertaining to the order of events
- Serialization:
  - Event A must happen before Event B
- Mutual exclusion:

  - Events A and B must not happen at the same time
  - E.g., several processes compete for a resource, but the nature of the resource requires that only one process at a time actually accesses it
    - Like processing different bank-account operations
      - What if 2 threads change the same account at the same time?

BEYOND JOIN

- Events A and B must not happen at the same time
  - Either A or B must block – not proceed until the "winning" process has completed
  - join is overkill – waits until a thread has completely finished executing
- Programmer must implement critical sections
  - "The compiler" has no idea what interleavings should or shouldn't be allowed
  - Need language primitives to do it

Broken Example: Unsafe Counter



Broken Example:
Shared Counter

```
public class Counter {
      private long value;

      Counter() { value=0; }

      public long get() {return value;}

      public void set(long newVal) {
            value=newVal;
      }

      public void incr() {value++;}
}
```

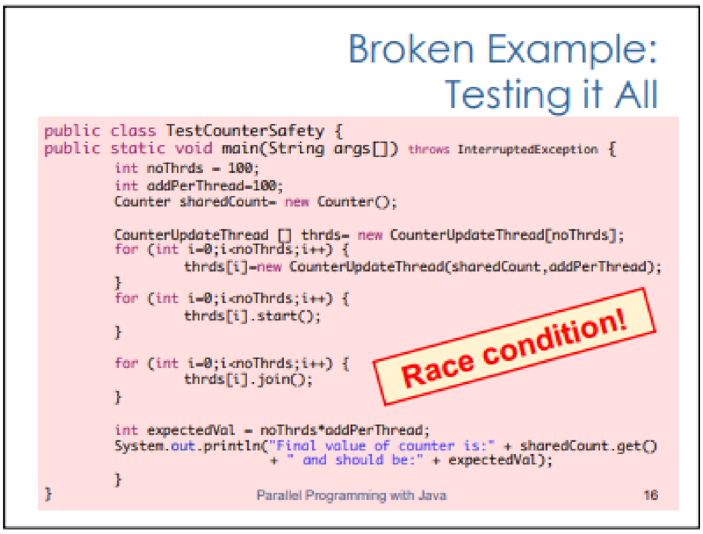
Parallel Programming with Java 14

Broken Example:
Procedure for Thread *i*

```
public class CounterUpdateThread extends Thread {
      Counter sharedCount;

      int rep;

      CounterUpdateThread(Counter c, int repeats) {
            sharedCount= c;
            rep=repeats;
      }
      public void run() {
            for (int i=0;i<rep;i++) {
                  sharedCount.incr();
}     }     }
```

Shared counter object

Parallel Programming with Java 15



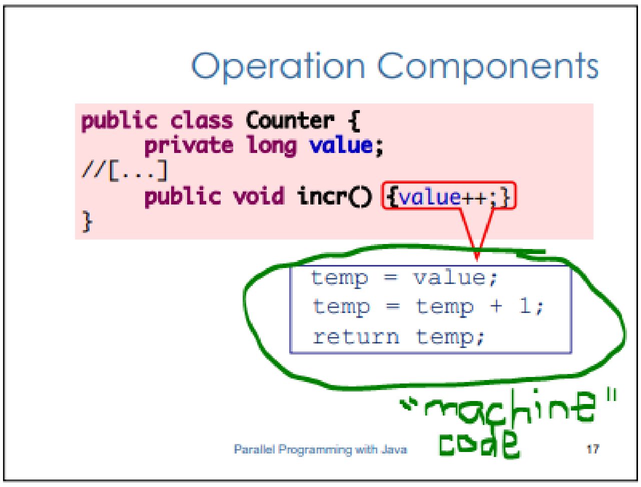
Broken Example: Testing it All

```
public class TestCounterSafety {
public static void main(String args[]) throws InterruptedException {
        int noThrds = 100;
        int addPerThread=100;
        Counter sharedCount= new Counter();

        CounterUpdateThread [] thrds= new CounterUpdateThread[noThrds];
        for (int i=0;i<noThrds;i++) {
                thrds[i]=new CounterUpdateThread(sharedCount,addPerThread);
        }
        for (int i=0;i<noThrds;i++) {
                thrds[i].start();
        }

        for (int i=0;i<noThrds;i++) {
                thrds[i].join();
        }

        int expectedVal = noThrds*addPerThread;
        System.out.println("Final value of counter is:" + sharedCount.get()
                        + " and should be:" + expectedVal);
        }
}
```

Race condition!

Parallel Programming with Java 16

- Have 1000 threads each doing 1000 increments, thus expect result to be 1000000.
    - But can get results like 998997.



Operation Components

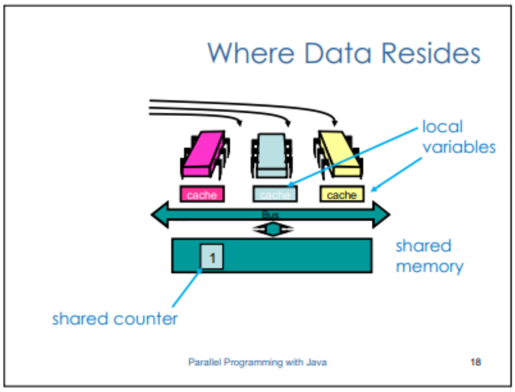
```
public class Counter {
        private long value;
//[...]
        public void incr() {value++;}
}
```

```
temp = value;
temp = temp + 1;
return temp;
```

"machine" code

Parallel Programming with Java 17

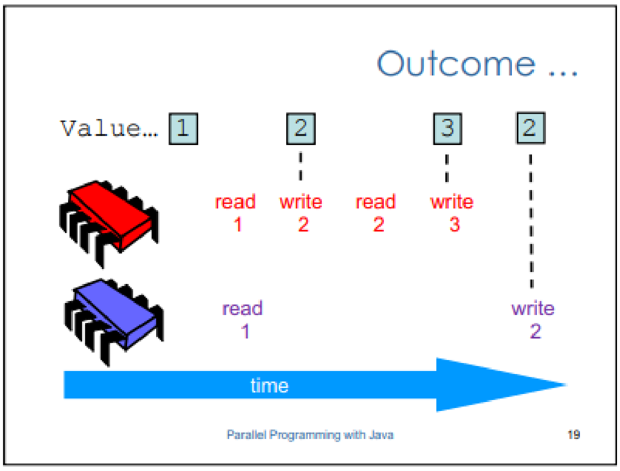
- Why?
    - Issue is with incr() method – but threads can interleave in this line of code.
    - At machine code level, value ++ becomes 3 separate statements at machine code level.
        - temp is in register memory, gets incremented then written back to shared memory.
        - This combines with issue of where data resides.



Where Data Resides

Parallel Programming with Java 18

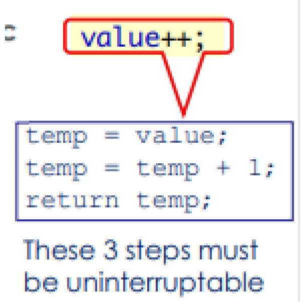
- Data normally in shared memory in shared counter object – but before incrementing, copy into local var that is particular to that thread (and not visible to other threads).
  - Then, locally and internally, increment before writing back.
  - If one of the other threads doesn't update/read during that process, it's going to get incorrect value for shared counter.



Outcome …

Parallel Programming with Java 19

Atomic Operations



```
value++;
```

```
temp = value;
temp = temp + 1;
return temp;
```

These 3 steps must be uninterruptable

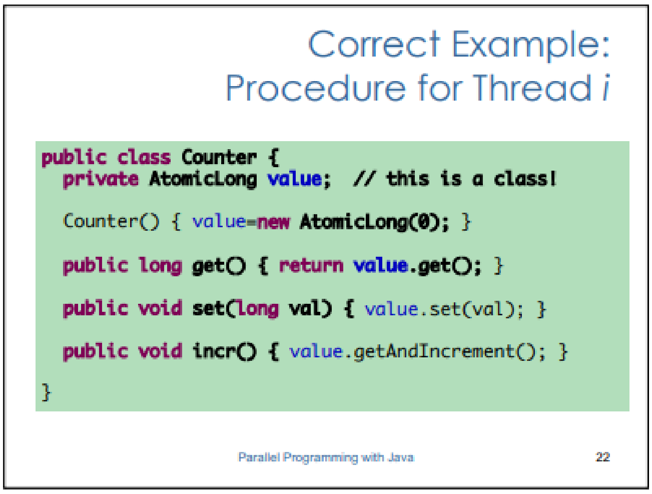
- Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.

- Guarantee of isolation from interrupts by concurrent threads.
- The operation is indivisible

ATOMIC VARIABLES IN JAVA

- One fix is to use an existing thread-safe atomic variable class
    - java.util.concurrent contains atomic variable classes
    - AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference
    - All have getAndSet()methods to indivisibly set the value and return it
    - Integer by default not atomic.
    - No atomic classes for character, double and float (have to use another method of protection for mutual exclusion).

Correct Example



Correct Example:
Procedure for Thread *i*

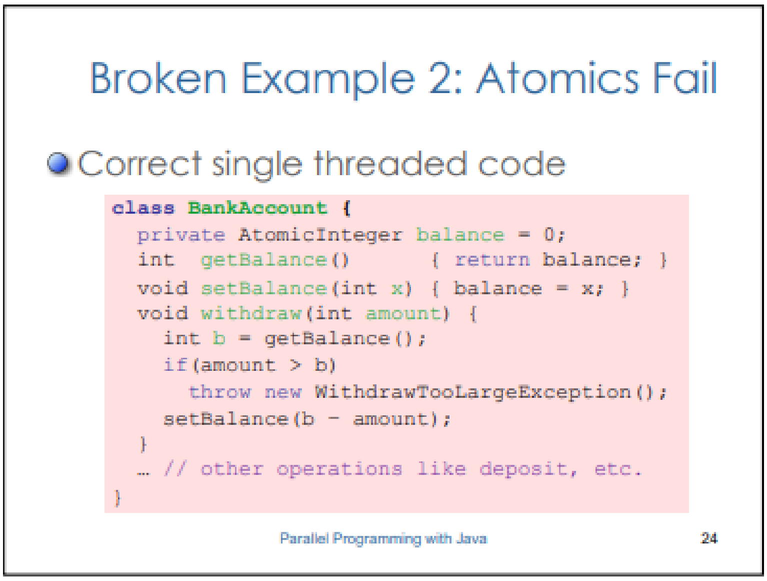
```
public class Counter {
  private AtomicLong value;  // this is a class!

  Counter() { value=new AtomicLong(0); }

  public long get() { return value.get(); }

  public void set(long val) { value.set(val); }

  public void incr() { value.getAndIncrement(); }
}
```

Parallel Programming with Java 22

LIMITATIONS

- Atomic variables only make a class thread-safe if ONE variable defines the class state
    - To preserve state consistency, you should update related state variables in a single atomic operation.
        - i.e. if have class with several vars and they need to be operated on as a collection then atomics not suitable.
- Atomics also unsuitable if you have a variable that needs some complex logic (other than the basic get and set methods provided by atomics).
- Don’t provide wide enough protection – don’t cover a section of code and provide mutual exclusion on that; only cover get, set and increment methods.
- Need another programming construct other than atomics

Broken Example



Broken Example 2: Atomics Fail

- Correct single threaded code

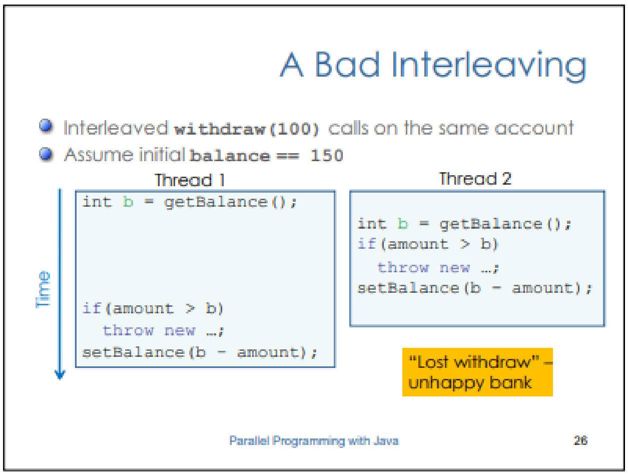
```
class BankAccount {
  private AtomicInteger balance = 0;
  int  getBalance()       { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
  }
  … // other operations like deposit, etc.
}
```

Parallel Programming with Java 24

- There are possible interleaving issues.

Interleaving Issues

- Suppose:
    - Thread T1 calls x.withdraw(100)
    - Thread T2 calls y.withdraw(100)
- The two threads could truly run at the same time on different processors.
- Or they may run one at a time, but the thread scheduler might stop one thread and start the other at any point, switching between them any number of times.
- In many scenarios it is not a problem for these two method calls to execute concurrently:
    - If x and y are not aliases, i.e they refer to distinct bank accounts, then there is no problem because the calls are using different memory.
        - Like two cooks using different pots at the same time
    - If one call happens to finish before the other starts, then the behavior is like in a sequential program.
            - This is like one cook using a pot and then another cook using the same pot.
- But if the calls interleave AND x and y are aliases (point to same place in memory), possible trouble…
        - Note that interleaving can happen even with one processor because a thread can be pre-empted at any point, meaning the thread scheduler stops the thread and runs another one



A Bad Interleaving

- Interleaved `withdraw(100)` calls on the same account
- Assume initial `balance == 150`

Thread 1

```
int b = getBalance();


if(amount > b)
  throw new …;
setBalance(b - amount);
```

Thread 2

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b - amount);
```

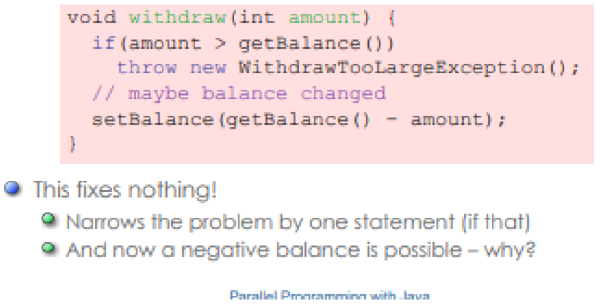
"Lost withdraw" – unhappy bank

Parallel Programming with Java 26

- Can withdraw 100 twice and still have 50 as the balance.
- How?
    - Global balance in 150
    - Two threads execute in || but thread 1 executes first
        - Gets balance and puts into local variable b, then thread 1 **suspends**.
        - Thread 2 executes, gets balance, which is still 150, then continues executing and global balance gets overwritten and becomes 50, then thread 2 **stops**.
        - Thread 1 executes again, but 150 balance stored locally (and no longer a *true reflection* of the balance) and gets used to compare to withdraw amount.
            - Thread 1 continues executing and balance set to 50.

FIXES

- Incorrect fix: Re-arranging or repeating certain opns *won't* fix bad interleavings
- For the example:
    - Balance starts at 150. Let's say Bob tries to withdraw 100 and Alice tries to withdraw 100.
    - Bob withdraws 100, 100 < 150 so the transaction is allowed. But before the new balance can be set, Alice withdraws 100.
        - The current value being stored is 150, so once again 100 < 150 and hence the transaction is allowed.
    - But now both Bob and Alice execute the setBalance() statement and the new balance is updated to be -50.
        - Essentially the problem arises as the balance is checked and updated in different steps, allowing for bad interleaving and outdated balance values.

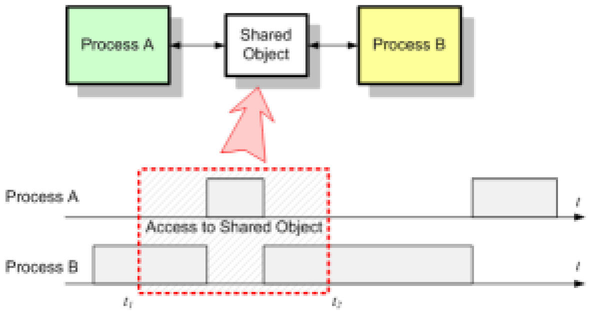


```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new WithdrawTooLargeException();
  // maybe balance changed
  setBalance(getBalance() – amount);
}
```

- This fixes nothing!
  - Narrows the problem by one statement (if that)
  - And now a negative balance is possible – why?

Parallel Programming with Java

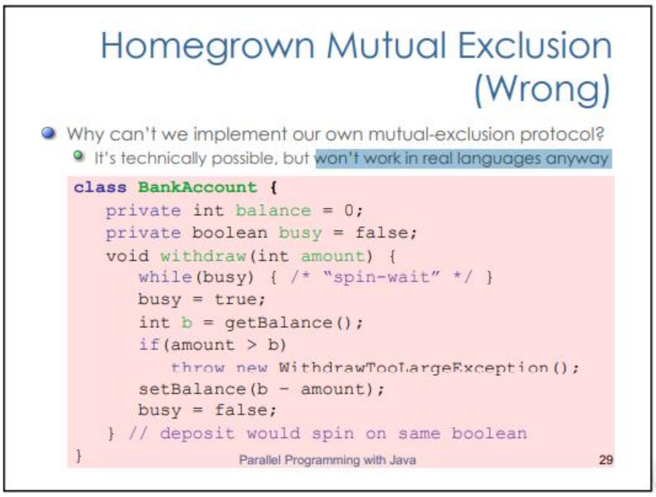
- The “sane” fix: mutual exclusion
  - At most one thread withdraws from account M at a time
    - Exclude other simultaneous operations on M too (e.g., deposit)
  - Critical section: i.e. piece of code with operations that are critical
    - *Critical* that such opns not be interleaved with other conflicting ones.



Parallel Programming with Java

HOMEGROWN MUTUAL EXCLUSION

- WRONG solution
- Can implement own mutual-exclusion protocol (it’s possible) – BUT won’t work in real languages anyway.

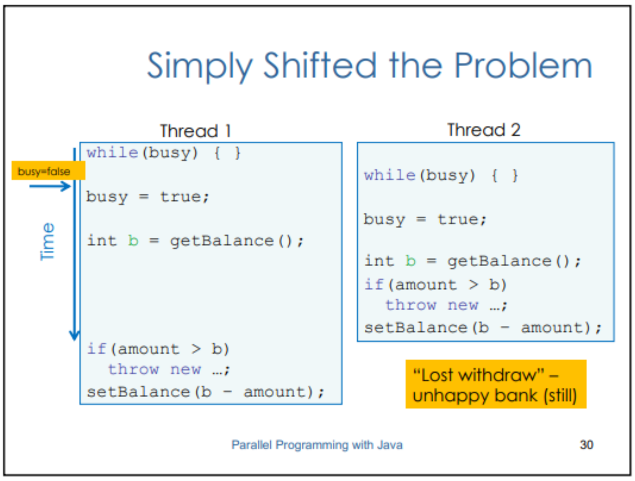


Homegrown Mutual Exclusion (Wrong)

- Why can’t we implement our own mutual-exclusion protocol?
  - It’s technically possible, but won’t work in real languages anyway

```
class BankAccount {
   private int balance = 0;
   private boolean busy = false;
   void withdraw(int amount) {
      while(busy) { /* "spin-wait" */ }
      busy = true;
      int b = getBalance();
      if(amount > b)
         throw new WithdrawTooLargeException();
      setBalance(b – amount);
      busy = false;
   } // deposit would spin on same boolean
}
```

Parallel Programming with Java 29



Simply Shifted the Problem

Thread 1

```
while(busy) { }

busy = true;

int b = getBalance();



if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2

```
while(busy) { }

busy = true;

int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

“Lost withdraw” – unhappy bank (still)

Parallel Programming with Java 30

- Why?

- Think we can use Boolean flag to allow only one person access to shared resources.
    - Then using waiting loop, prevent someone from going past that flag.
- Opportunity for interleaving between while(busy) and busy = true.
    - Can check that busy is false, but then it might get set to true before have a chance to set it to true yourself.
- Say have 3 threads – thread 1 and 2 both want to withdraw from same account.
    - But account busy, thread 3 sets busy flag to true.
    - Thread 1 starts, goes into busy waiting loop, the suspends. Thread 2 starts up and then also suspends.
    - Then thread 3 sets busy to false, finishes executing – frees up thread 1 and 2 to work.
    - Thread 1, then 2 can now go into while loop and set busy to true – end up in same situation as before, allowing multiple withdrawals from same bank balance.
        - Need option to not change Boolean guard on critical section.

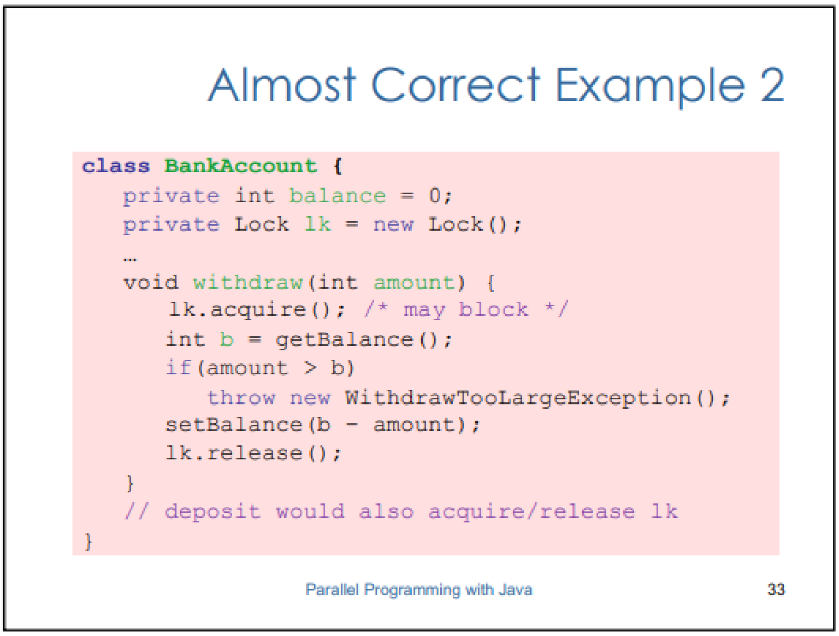
Locks

- Primitive within programming language – can be thought of as abstract datatype.
- Can use this achieve what we need.
- Process to implement mutual exclusion with locks:
    - "new": create lock obj, initially "not held" state.
    - "acquire": used to create mutual exclusion on section of access.
        - Blocks if this lock is already currently "held".
        - Once "not held", makes lock "held" – changes state in **indivisible way**, no-one else can be simultaneously changing it.
            - Thus, the opn is **atomic**.
    - "release":
        - Makes lock "not held"
        - If >1 threads blocked on it (i.e. waiting to acquire it), exactly one will acquire it.

MUTUAL EXCLUSION USING LOCKS

- A lock implementation ensures correct behaviour given simultaneous acquires and/or releases
    - Example: Two acquires – one will "win" and one will block
- How can this be implemented?
    - Uses special hardware and O/S support

Example Continued



Almost Correct Example 2

```
class BankAccount {
   private int balance = 0;
   private Lock lk = new Lock();
   …
   void withdraw(int amount) {
      lk.acquire(); /* may block */
      int b = getBalance();
      if(amount > b)
         throw new WithdrawTooLargeException();
      setBalance(b – amount);
      lk.release();
   }
   // deposit would also acquire/release lk
}
```

Parallel Programming with Java 33

- Flaw here is that, if exception thrown, lock is ***not released*** and then nothing else can acquire it.

USING LOCKS CORRECTLY

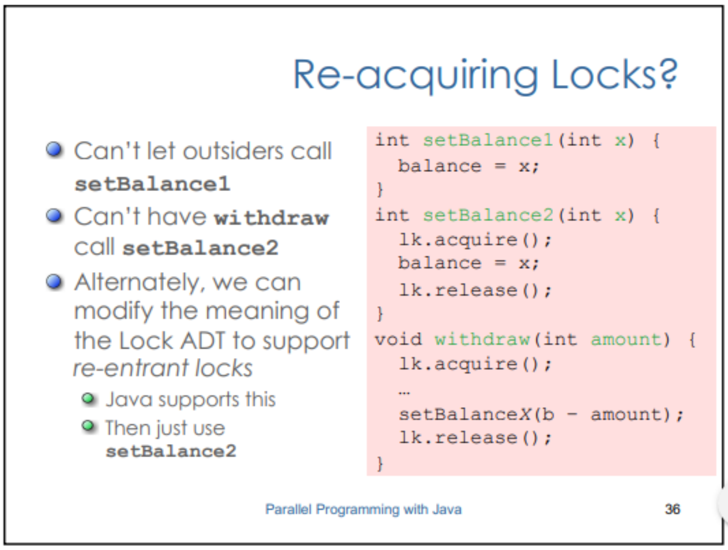
- A lock is a very primitive mechanism
  - Programmer must use correctly to implement critical sections
- For bank account example:
  - Incorrect: Use different locks for withdraw and deposit
    - Mutual exclusion works only when using same lock.
  - Incorrect: Forget to release a lock (blocks other threads forever!)
  - Poor performance: Use same lock for every bank account.
    - Then e.g. only one person at a time (within entire bank system) can withdraw/ deposit
  - Each account can use different lock, so different threads can operate on different accounts *at the same time*.
    - Provides mutual exclusion needed for each account.

OTHER LOCK OPERATIONS

- For Bank Account example:
  - If withdraw and deposit use the same lock, then properly synchronized
  - What about getBalance and setBalance?
    - Assume they're public methods – means they can be called outside bank-account implementation.
      - Thus, should also acquire and release the lock before accessing the account's field's like balance.
    - If they don't acquire the same lock, then a race between setBalance and withdraw could produce a wrong result.

- If they do acquire the same lock, then withdraw would block forever because it tries to acquire a lock it already has.
    - i.e If withdraw calls these helper methods *while holding the lock*, thread blocks forever because it's waiting for "some thread" to release the lock that it, itself is holding.
    - Generally safer to use a catch-statement or finally-statement to ensure lock always gets released.
    - Need re-entrant locks.

RE-ENTRANT LOCKS



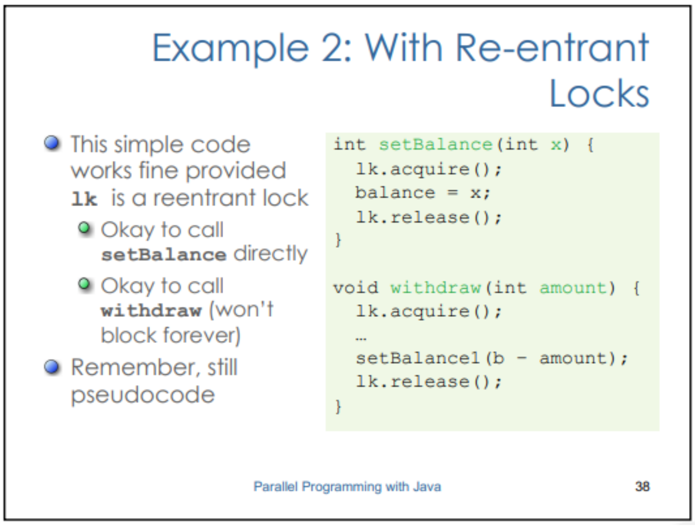
Re-acquiring Locks?

- Can't let outsiders call **setBalance1**
- Can't have **withdraw** call **setBalance2**
- Alternately, we can modify the meaning of the Lock ADT to support *re-entrant locks*
    - Java supports this
    - Then just use **setBalance2**

```
int setBalance1(int x) {
  balance = x;
}
int setBalance2(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}
void withdraw(int amount) {
  lk.acquire();
  …
  setBalanceX(b - amount);
  lk.release();
}
```

Parallel Programming with Java 36

- A re-entrant lock (a.k.a. recursive lock) stores
    - The thread (if any) that currently holds it
    - A count
    - When the lock goes from not-held to held, the count is 0
- If the current holder calls acquire:
    - It does not block
    - It increments the count
- If count on lock > 0 and NOT held by you, then have to block and wait for counter to become zero.
- On release:
    - if the count is > 0, the count is decremented
    - if the count is 0, the lock becomes not-held
    - In other words, a lock is released when the # of release opns by the holding thread equals the # of acquire opns.



Example 2: With Re-entrant Locks

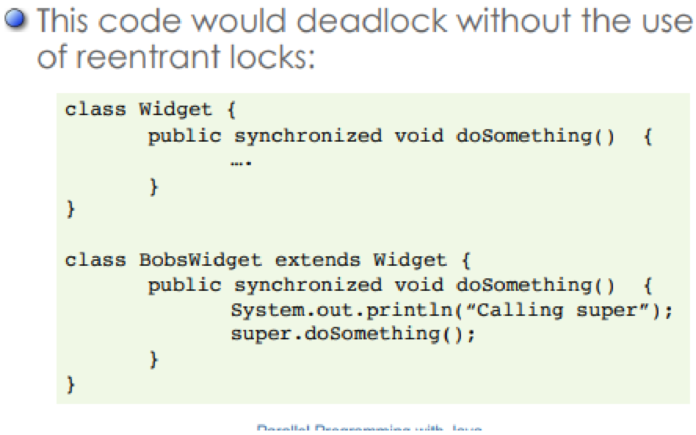
- This simple code works fine provided **lk** is a reentrant lock
  - Okay to call **setBalance** directly
  - Okay to call **withdraw** (won't block forever)
- Remember, still pseudocode

```
int setBalance(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}

void withdraw(int amount) {
  lk.acquire();
  …
  setBalance1(b – amount);
  lk.release();
}
```

Parallel Programming with Java 38

INHERITANCE

- Re-entrant Locks and Inheritance



- This code would deadlock without the use of reentrant locks:

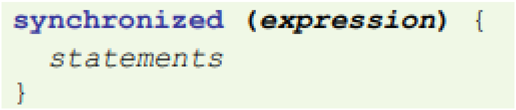
```
class Widget {
       public synchronized void doSomething()  {
              ….
       }
}

class BobsWidget extends Widget {
       public synchronized void doSomething()  {
              System.out.println(“Calling super”);
              super.doSomething();
       }
}
```

Parallel Programming with Java

- Both methods in sub and super class for doSomething() are protected with lock.
  - doSomething() overridden in sub class.
  - Both doSomethings() lock on *same object* i.e. the **this** object.
    - This is a problem – if call super class doSomething from within subclass, then will be trying to lock on same lock.
      - Will only work if have re-entrant locks.
- Hence why re-entrants necessary.

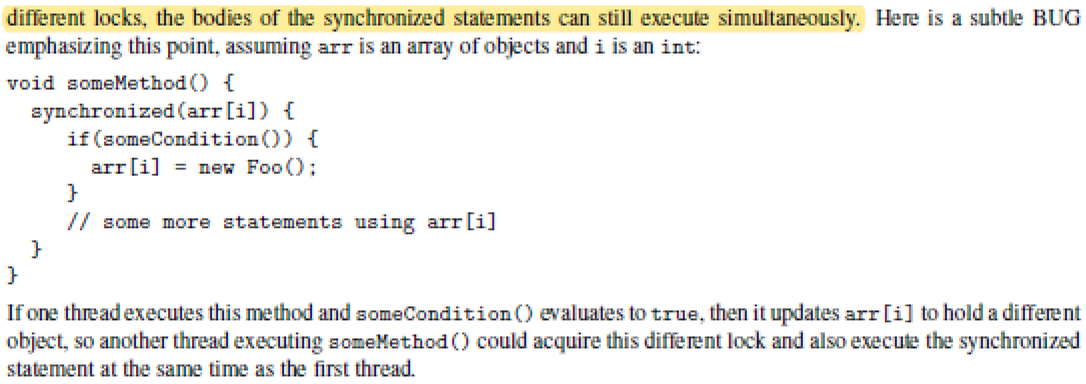
MUTUAL EXCLUSION IN JAVA

- Java has built in support for locks.



```
synchronized (expression) {
   statements
}
```

- Synchronised Block
    - Built-in Java locking mechanism for enforcing atomicity via re-entrant locks
        - 1. Evaluates expression to a reference to an object – NOT null or a number
            - Every object (but not primitive types) “is a lock” that any thread can acquire/release in Java.
        - 2. Acquires the lock, blocking if necessary “If you get past the {, you “have the lock”
            - Statement won’t block if executing thread already holds it.
            - Statements can be any normal code (loops, method calls etc).
        - 3. Releases the lock “at the matching }” or when the program “jumps out of the statement” via an exception, a return, break or continue.
            - “}” is an implicit release statement.
            - So impossible to forget to release the lock
            - There’s no way to release the lock before reaching the ending “}” (without a jump out statement) – not often that you want to do so.
- A synchronized block has two parts:
    - A reference to an object that will serve as the lock
    - A block of code to be guarded by that lock
- Any obj can serve as a lock e.g. can create an instance of object, the superclass of every other class in Java.
    - Best to use instance of the object that has the fields relevant to synchronisation to serve as a lock.
    - Each Java obj represents a **different** lock.
        - If two threads use synced statements to acquire different locks, the bodies of the synchronised statements can execute simultaneously.

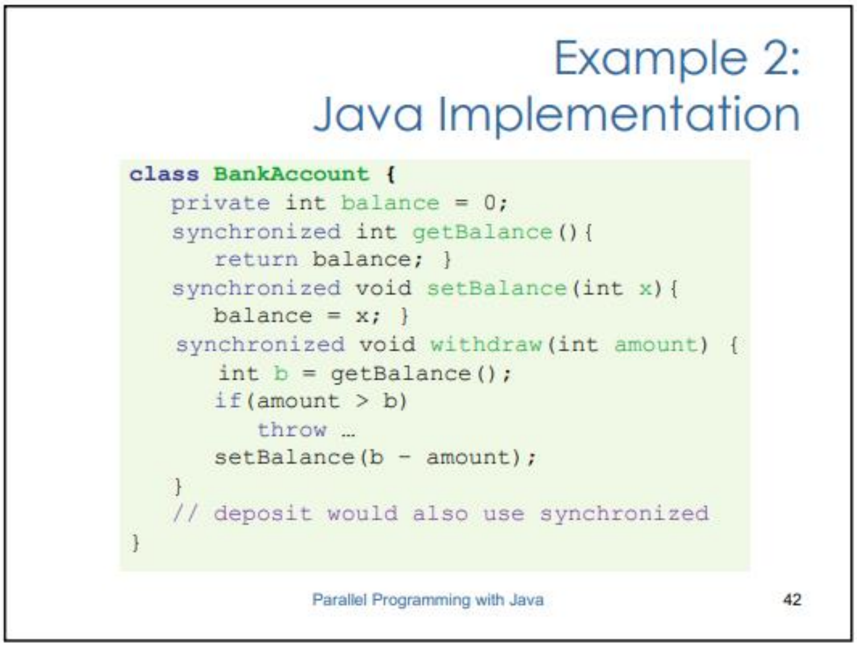


different locks, the bodies of the synchronized statements can still execute simultaneously. Here is a subtle BUG emphasizing this point, assuming `arr` is an array of objects and `i` is an `int`:

```
void someMethod() {
  synchronized(arr[i]) {
     if(someCondition()) {
       arr[i] = new Foo();
     }
     // some more statements using arr[i]
  }
}
```

If one thread executes this method and `someCondition()` evaluates to `true`, then it updates `arr[i]` to hold a different object, so another thread executing `someMethod()` could acquire this different lock and also execute the synchronized statement at the same time as the first thread.

- All methods need to agree on what lock they’re using.
- Synchronized can be either a method or block qualifier:
    - synchronized void f() { body; } is equivalent to:
    - void f(){ synchronized(this) { body; } }
        - Why these two different ways?
            - If you sync a whole method, that gives wider protection.
            - Want min amount of protection that guarantees correctness – min because if expand beyond what’s needed => impact performance.

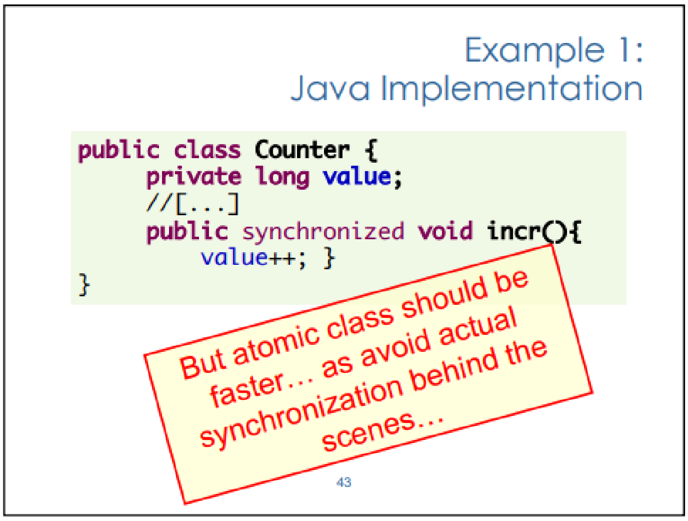


Example 2: Java Implementation

```
class BankAccount {
   private int balance = 0;
   synchronized int getBalance(){
      return balance; }
   synchronized void setBalance(int x){
      balance = x; }
   synchronized void withdraw(int amount) {
      int b = getBalance();
      if(amount > b)
         throw …
      setBalance(b - amount);
   }
   // deposit would also use synchronized
}
```

Parallel Programming with Java 42

- Put synchronized keywords on accessor methods.
- Can do this for a counter:
  - NB that in practice, wouldn't use locks for counter (rather use atomics)



Example 1: Java Implementation

```
public class Counter {
     private long value;
     //[...]
     public synchronized void incr(){
         value++; }
}
```

But atomic class should be faster… as avoid actual synchronization behind the scenes…

43

BLOCK VS METHOD SYNCHRONIZATION

- Synchronisation is NOT a secret sauce/spice that should be sprinkled over entire class.
  - Look at MIN amount within block to sync on.
- Block synchronization is preferred over method synchronization:
  - With block synchronization you only need lock the critical section, instead of whole method
  - Synchronization comes with a performance cost
  - should synchronize only code that absolutely needs to be synchronized

Java Locks Summary

- Every Java object possesses one lock
    - Manipulated only via *synchronized* keyword
    - Scalars like int are not Objects, so can only be locked via their enclosing objects
    - i.e. NB: CAN'T lock on types, only objects!
- Java locks are re-entrant
    - A thread hitting *synchronized* passes if the lock is free or it already possesses the lock, else waits
    - Released after passing as many }'s as {'s for the lock
    - Cannot forget to release lock
- In cases of contention, we cannot control which process gets the lock

Thread Safety

- Writing thread-safe code is about managing an object's state:
  - We need to protect data from concurrent access
- Worried about shared, mutable state
  - shared: accessed by multiple threads
  - mutable: value can change

A Thread Safe Class

- A class can be considered to be thread-safe if it behaves correctly when accessed from multiple threads:
  - Regardless of scheduling or interleaving by the runtime environment
  - And with no additional synchronization on the part of the calling code
  - i.e. A thread-safe class cannot be placed in an invalid state

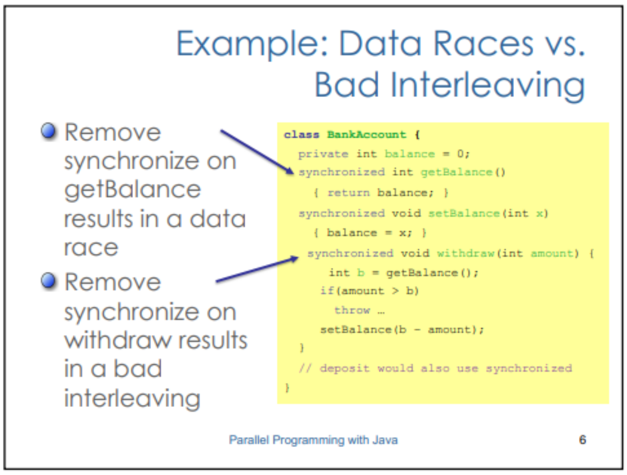
Race Conditions

TYPES

- "Race condition" covers two different outcomes resulting from lack of synchronization
- Data races:
  - Simultaneous read/write or write/write of the same memory location
  - Always an error, due to compiler & hardware
- Bad interleavings:
  - Despite lack of data races, exposing bad intermediate state
  - "Bad" depends on your specification

DATA RACES

- A Data Race occurs when:
  - Two instructions from different threads access the same memory location
  - At least one of these accesses is a write (read/write, or write/write)
  - And there is no synchronization that forces any particular order among these accesses
- Overcome in Java using synchronized keyword, or volatile variables, explicit locks, atomic variables



Example: Data Races vs. Bad Interleaving

- Remove synchronize on getBalance results in a data race
- Remove synchronize on withdraw results in a bad interleaving

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance()
    { return balance; }
  synchronized void setBalance(int x)
    { balance = x; }
   synchronized void withdraw(int amount) {
      int b = getBalance();
     if(amount > b)
       throw …
     setBalance(b - amount);
  }
  // deposit would also use synchronized
}
```

Parallel Programming with Java 6

- For example:
    - Synchronised on get and set methods prevents data race – without sync, could have simultaneous reading and writing (leaves balance in incorrect state).
    - Synchronised on withdraw to prevent bad interleavings – multiple threads could suspend and resume and thus get an incorrect mix of statements.

THREADED JAVA FRAMEWORKS (BEWARE!)

- Even if your code is sequential it may still need to be thread safe
    - Java may be accessing your code in a multithreaded way.
- Several Java frameworks create threads and can call your components from these threads
    - AWT and Swing create threads for managing user interface events
    - Timer creates threads for executing deferred tasks
    - Component frameworks, such as servlets and RMI, create pools of threads and invoke component methods in these threads

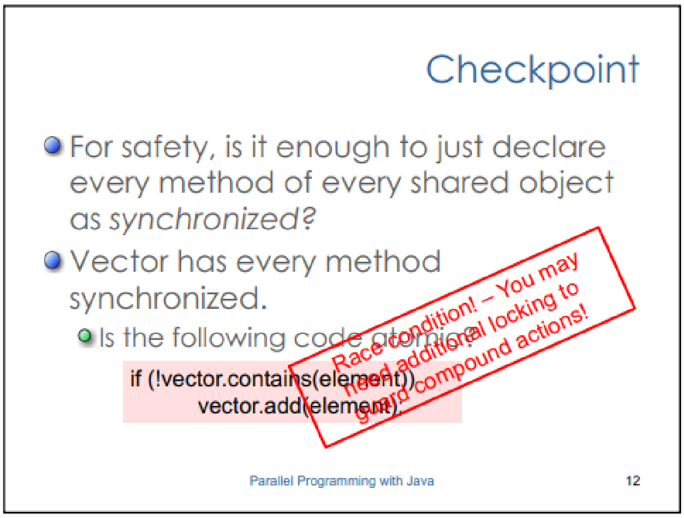
Java Timer Class

- Timer is a convenience mechanism for scheduling tasks to run later
    - Either once or periodically
    - TimerTasks are executed in a Thread managed by the Timer, not the application
    - If TimerTask accesses data that is also accessed by other application threads, then not only must the TimerTask do so in a thread safe manner, but so must any other classes that access that data
- Easiest is to ensure that all objects accessed by TimerTask are themselves thread safe

Java Monitor Pattern

- Use to make class thread safe
- An object following the Java Monitor encapsulates all its mutable state and guards it with the object's **own intrinsic lock**
- Advantage is simplicity

- Disadvantage is poor concurrency
  - Thread safety achieved by serializing all access to collection's state - only process one request at a time



Checkpoint

- For safety, is it enough to just declare every method of every shared object as *synchronized?*
- Vector has every method synchronized.
  - Is the following code atomic?

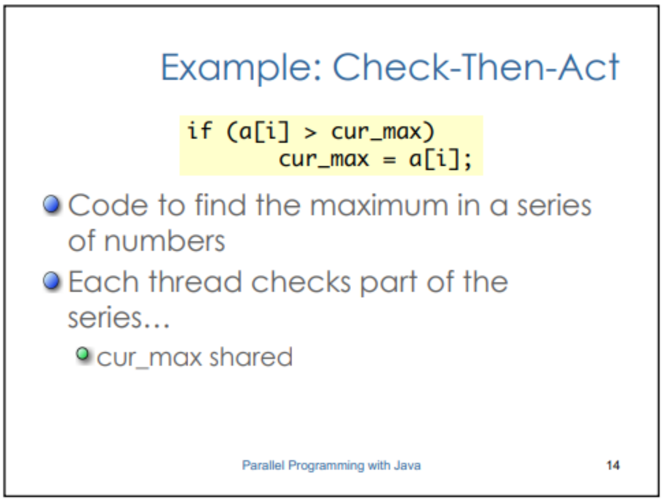
```
if (!vector.contains(element))
    vector.add(element);
```

Race condition! – You may need additional locking to guard compound actions!

Parallel Programming with Java 12

Race Condition: Compound Actions

- Get unsafe read-modify-write compound action
  - Where resulting state is derived from the previous state
- Another example is a check then-act compound action
- Must be executed atomically in order to remain thread-safe



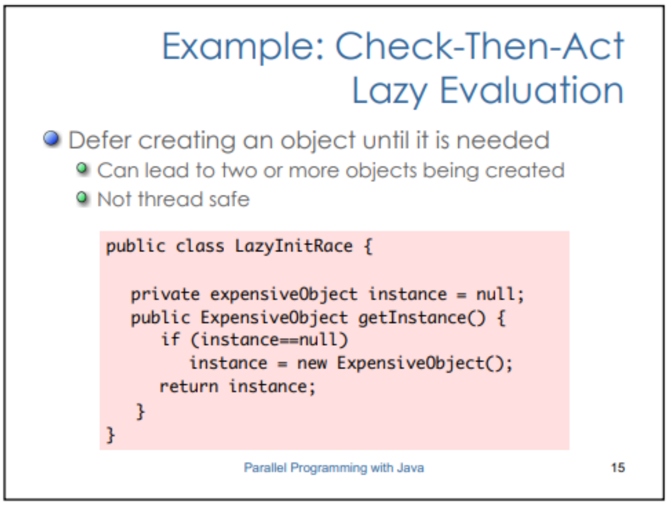
Example: Check-Then-Act

```
if (a[i] > cur_max)
    cur_max = a[i];
```

- Code to find the maximum in a series of numbers
- Each thread checks part of the series…
  - cur_max shared

Parallel Programming with Java 14

- For Example: shared resource is max value
  - If opns are not done atomically, can end up finding local max for parts of the array (instead of an overall max).



Example: Check-Then-Act Lazy Evaluation

- Defer creating an object until it is needed
  - Can lead to two or more objects being created
  - Not thread safe

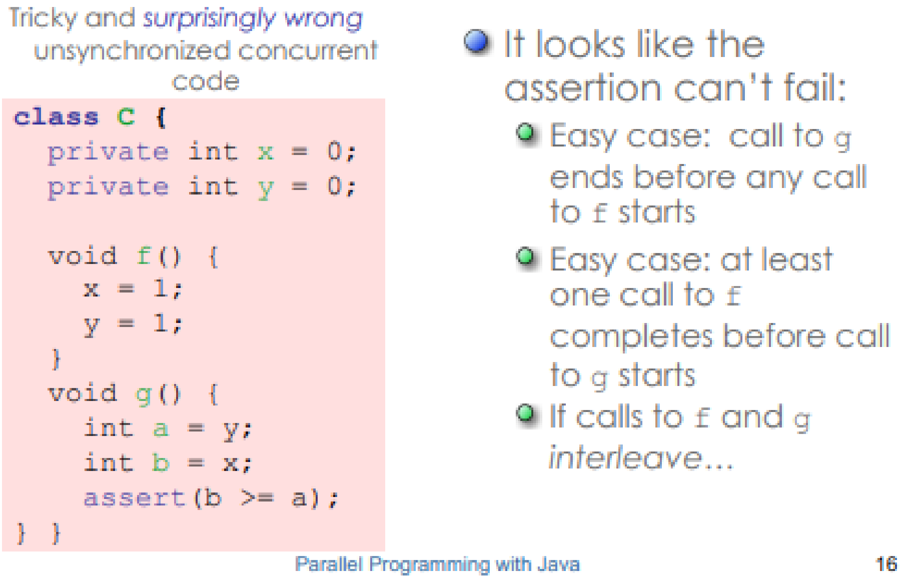
```
public class LazyInitRace {

  private expensiveObject instance = null;
  public ExpensiveObject getInstance() {
     if (instance==null)
        instance = new ExpensiveObject();
     return instance;
   }
}
```

Parallel Programming with Java 15

- Another example: if(instance ==null) thread could suspend here, allowing others to come in.
  - Means many objects can be created
  - Not thread safe

Re-ordering and Interleaving

- Strategy for testing whether need synchronisation or not:
  - Test all possible permutations for possible interleavings.



Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
} }
```

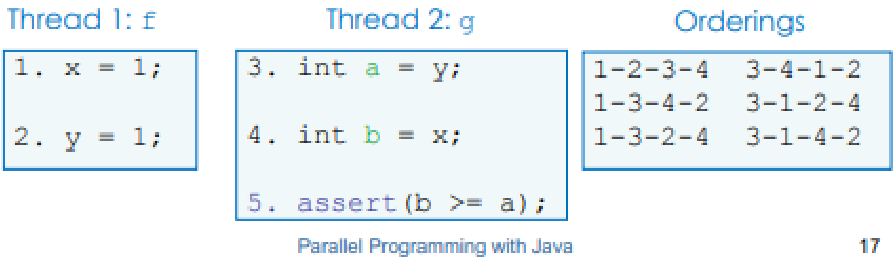
- It looks like the assertion can't fail:
  - Easy case: call to g ends before any call to f starts
  - Easy case: at least one call to f completes before call to g starts
  - If calls to f and g *interleave…*

Parallel Programming with Java 16

- NB: assert is a function
  - If it evals to true, it carries on with code execution.
    - If it's false, prog stops at that point – so good for debugging
    - Failure occurs at point at which bug occurs.

INTERLEAVED CASES

- Can use Proof by Contradiction or Exhaustive Proof
- No interleaving of f and g has the assertion fail
  - Proof #1: Exhaustively consider all possible orderings of access to shared memory



Thread 1: f

```
1. x = 1;

2. y = 1;
```

Thread 2: g

```
3. int a = y;

4. int b = x;

5. assert(b >= a);
```

Orderings

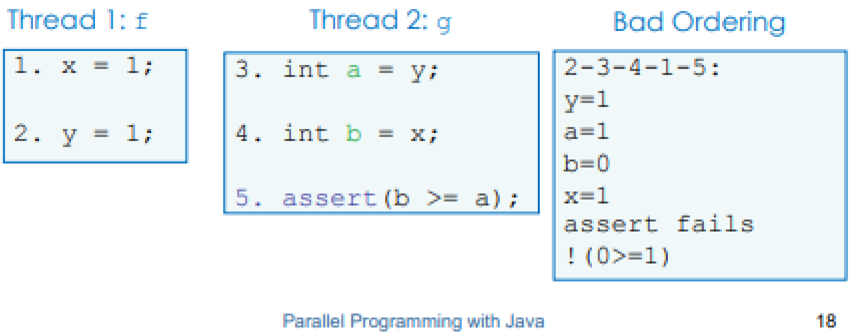
```
1-2-3-4   3-4-1-2
1-3-4-2   3-1-2-4
1-3-2-4   3-1-4-2
```

Parallel Programming with Java 17

- Proof #2: If !(b>=a), then a==1 and b==0. But if a==1, then a=y happened after y=1.
  - And since programs execute in order, b=x happened after a=y and x=1 happened before y=1.
  - So, by transitivity, b==1. Contradiction

FAILURE OF ASSERTION ON RE-ORDERING

- No interleaving of f and g has the assertion fail
- But the compiler is free to re-order non-dependent code
  - Compiler does this in order to get optimal performance
  - Remember, can only re-order statements with no dependency between them.
  - Statements 1 and 2 don't depend on each other (same for 3 and 4) – so compiler can reorder them.



Thread 1: f

```
1. x = 1;

2. y = 1;
```

Thread 2: g

```
3. int a = y;

4. int b = x;

5. assert(b >= a);
```

Bad Ordering

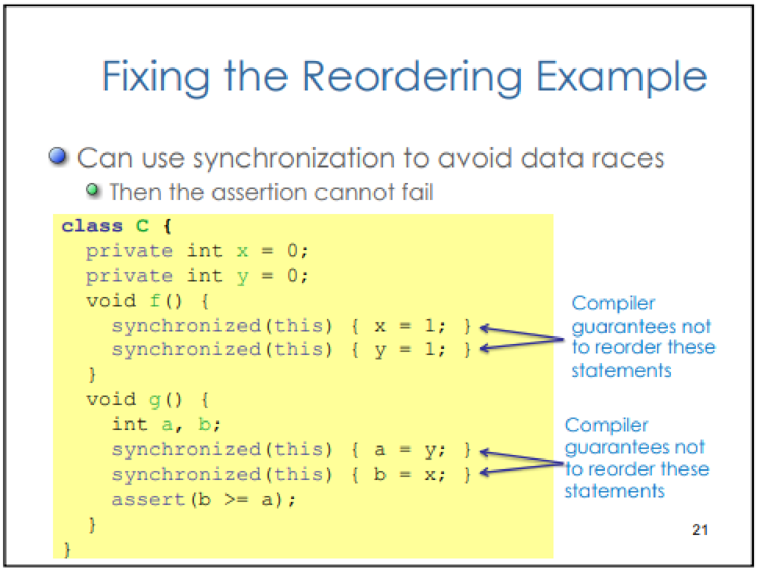
```
2-3-4-1-5:
y=1
a=1
b=0
x=1
assert fails
!(0>=1)
```

Parallel Programming with Java 18

Functionality and Data Races

- Data races are always wrong.
- However, the code has two data races
  - Unsynchronized read/write or write/write of same location
- Rule: If code has data races, you cannot reason about it with interleavings (i.e. can't say "well, no combo of inteleavings will lead to problems" – OS is always out to get you)

    - Assertion can fail because compiler allowed to re-organise internally.
- Best to avoid data races
- How are data races possible?
    - Code Reordering:
        - For performance, the compiler and the hardware are free to reorder operations
        - So long as this is not detectable from within the thread (dependencies still respected)



Fixing the Reordering Example

- Can use synchronization to avoid data races
    - Then the assertion cannot fail

```
class C {
  private int x = 0;
  private int y = 0;
  void f() {
    synchronized(this) { x = 1; }
    synchronized(this) { y = 1; }
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; }
    synchronized(this) { b = x; }
    assert(b >= a);
  }
}
```

Compiler guarantees not to reorder these statements

Compiler guarantees not to reorder these statements

21

- Java has feature to help:
    - If you lock using sync on a lock, then do it again later, re-ordering that compiler does is suspended.
- If lock on the *this* of object C (in example), then put lock acquisition and release around x =1 and then a ***separate*** locking on that *same lock* for y =1, then those 2 lines of code won't be re-ordered (guaranteed)

Visibility (Sharing Objects)

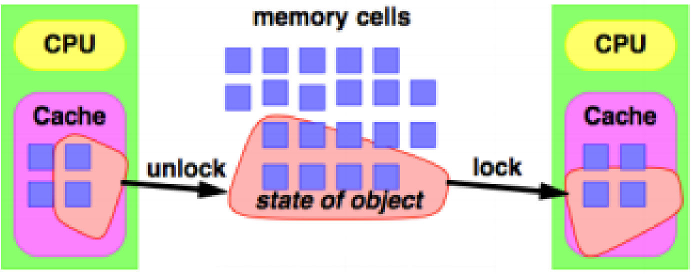
- Remember, have memory hierarchy with different speeds of access
    - When update locally in thread to cache, may not write through to RAM, where other threads can access it.
    - Staleness – not necessarily seeing most up to date version of shared memory if accessing your cache and somebody else has updated it in main memory.
- Synchronization is also about memory visibility:
    - When a thread modifies an object, we need to ensure that other threads can see the changes
    - Unless synchronization is used every time it is possible to see a stale value
    - Worse, staleness is not all-or-nothing
    - Some variable may be up-to-date, while others are stale

STALENESS AND SYNCHRONISATION

- Staleness solved by synchronisation
- synchronized clears locally cached values and forces reloads from main storage
  - So, synchronize all the getters and setters of shared values…on the SAME lock
  - We do take a performance hit by doing this
- Without locking, there are no promises about if and when caches will be flushed or reloaded.

LOCKS AND CACHING

- Locking generates messages between threads and memory
  - Lock acquisition forces reads from memory to thread cache
- Lock release forces writes of cached updates to memory
- E..g have 2 CPUS with each their own cache.
  - They take data (state of object) from the memory cells area.
  - When have lock on something and update it in local cache (on left), and unlock it, that forces data in local cache to be pushed to update obj state in main memory
  - When another thread on CPU tries to get lock and is successful, it'll be forced to invalidate its own cache and get object state from memory.

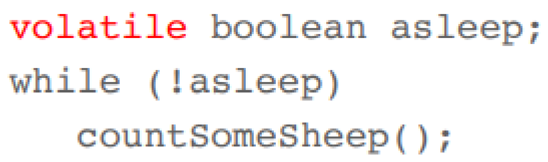


Volatile Variables

- *volatile* keyword controls per-variable flush and reload
  - Variables declared volatile are not cached and a read always returns the most recent write by any thread
- Usage:
  - Slower than regular fields, faster than locks (because no blocking)
  - But limited utility: fragile and code more opaque
  - Really for experts: avoid them; use standard libraries instead

Volatile Flags

- Most common use of volatile is for a flag variable:



```
volatile boolean asleep;
while (!asleep)
   countSomeSheep();
```

- Volatile does **not** mean atomic
  - While locking can guarantee both visibility and atomicity, volatile variables can only guarantee visibility

Atomics and Volatile

- Prefer atomic:
  - Threads don't share primitive types because Java is pass-by-value for primitives
    - So. value loses volatile state when based as a parameter to a method.
  - So, if you want to use a boolean flag shared between threads (to signal something), use AtomicBoolean, not volatile boolean (which won't work)
    - Best to use volatile when access variable as field of a class.

Policies and Guidelines for Thread Safety

CONCEPTUAL SPLIT OF MEMORY USAGE

- Thread-confined (thread-local)
  - Object owned exclusively by and confined to one thread; modified only by owning thread
- Shared read-only (immutable)
  - Can be accessed by multiple threads without synchronization, but no modifications
- Shared thread-safe
  - Performs synchronization internally, so no data races by multiple threads
- Synchronized (Guarded)
  - Accessed only when a lock is held

Thread-Local

- GUIDELINE #1: whenever possible don't share resources
  - Easier to have each thread with own thread-local copy of a resource
  - This is correct only if threads don't need to communicate through the resource
  - Example: Random objects
- Note: Since each call-stack is thread-local, never need to synchronize on local variables
- If data is accessed only from a single thread, no synchronization is needed

Swing

- Swing (Java GUI Widget Toolkit) uses thread confinement extensively:

  - Swing visual components and data model objects are not thread safe
  - Safety achieved by confining them to the Swing Event Dispatch Thread
  - NB code running in other threads should not access these objects

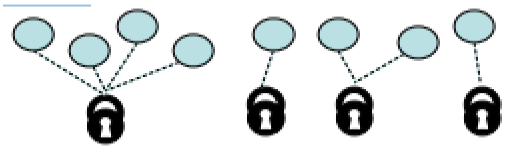
Immutable

- Immutable Objects are always thread safe since they only have one state
  - Use of final guarantees initialization safety
- GUIDELINE #2: Whenever possible, don't update shared objects
  - Make new local objects instead
  - If a location is only read, never written, then no synchronization is necessary!

Shared thread-safe and Synchronised

- After minimizing memory that is (1) thread shared and (2) mutable, we need guidelines for how to use locks
- No data races
  - GUIDELINE #3: Never allow two threads to read/write or write/write the same location at the same time
  - Necessary: In Java or C, a program with a data race is almost always wrong
  - Not sufficient: Our check-then-act example had bad interleavings but no data races

CONSISTENT LOCKING

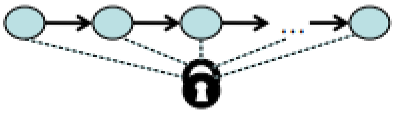


- GUIDELINE #4: For each location needing synchronization, have a lock that is always held when reading or writing the location
  - lock guards the location
  - same lock can (and often should) guard multiple locations
- In Java, often the guard is the object containing the location
- The mapping from locations to guarding locks is conceptual
- Not sufficient: It prevents all data races but still allows bad interleavings
- Not necessary: Can change the locking protocol dynamically…
- But Consistent locking is an excellent guideline
  - A good "default assumption" about program design

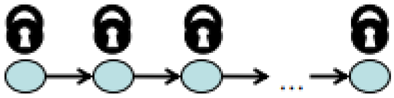
Locking Caveats

- Be aware of what a lock block is doing and how long it will take to execute
  - Holding a lock for a long time introduces the risk of liveness and performance problems
  - Avoid holding locks during lengthy computations or during network or console I/O

Lock Granularity



- Coarse-grained
  - Fewer locks, i.e., more objects per lock
    - Examples:
      - One lock for entire data structure (e.g., array)
      - One lock for all bank accounts



- Fine-grained
  - More locks, i.e., fewer objects per lock
    - Examples:
      - One lock per data element (e.g., array index)
      - One lock per bank account

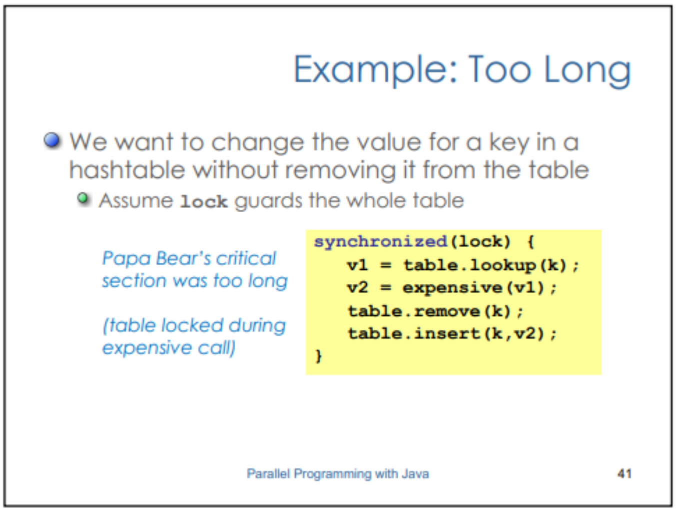
Granularity Trade-offs

- Coarse-grained advantages
  - Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
- Fine-grained
  - Advantages
    - More simultaneous access (coarse-grained can lead to unnecessary blocking)
    - More controlled
    - More ||ism
  - Disadvantages
    - Introduces overheads – memory costs to having multiple locks
- GUIDELINE #5: Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue
  - Locking granularity is a spectrum and we have to iterate along it until find what works for the problem.
  - Alas, often leads to bugs

Critical-section Granularity

- Critical-section size: how much work done while holding lock(s)
  - i.e. the code between lock acquisition and release
  - Don't think in terms of # of lines of code, but rather time it'll take to execute that code.

- If critical sections run for too long (e.g. I/O):
  - Performance loss because other threads are blocked
  - Protecting more than needed
- If critical sections are too short:
  - Bugs because you broke up something where other threads should not be able to see intermediate state
  - Too little protection
- GUIDELINE #6: Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions



Example: Too Long

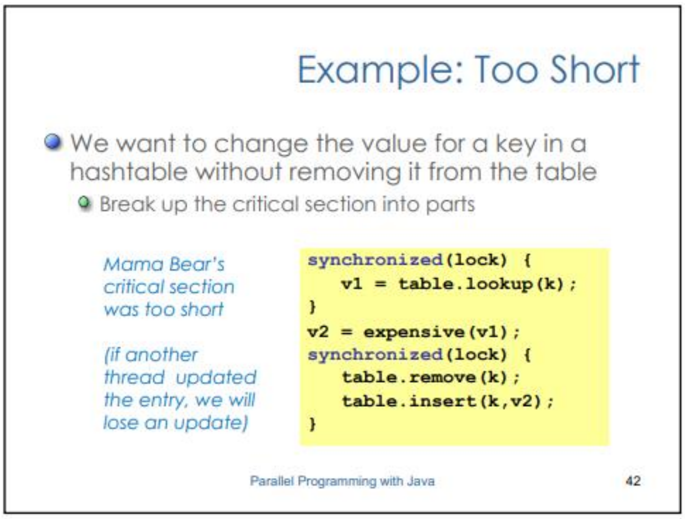
- We want to change the value for a key in a hashtable without removing it from the table
  - Assume **lock** guards the whole table

Papa Bear's critical section was too long

(table locked during expensive call)

```
synchronized(lock) {
   v1 = table.lookup(k);
   v2 = expensive(v1);
   table.remove(k);
   table.insert(k,v2);
}
```

Parallel Programming with Java 41



Example: Too Short

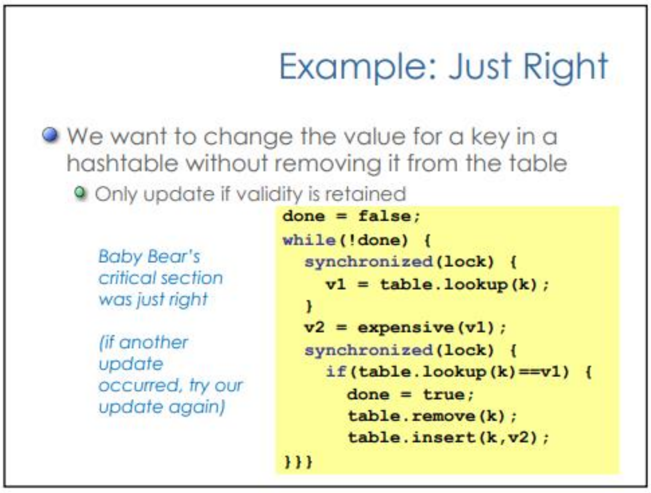
- We want to change the value for a key in a hashtable without removing it from the table
  - Break up the critical section into parts

Mama Bear's critical section was too short

(if another thread updated the entry, we will lose an update)

```
synchronized(lock) {
   v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
   table.remove(k);
   table.insert(k,v2);
}
```

Parallel Programming with Java 42



Example: Just Right

- We want to change the value for a key in a hashtable without removing it from the table
  - Only update if validity is retained

Baby Bear's critical section was just right

(if another update occurred, try our update again)

```
done = false;
while(!done) {
  synchronized(lock) {
    v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  synchronized(lock) {
    if(table.lookup(k)==v1) {
      done = true;
      table.remove(k);
      table.insert(k,v2);
}}}
```

- Too long:
  - No other thread can access data strc while opn being done.
- Too short:
  - No data race here – read and write opns in separate critical sections protected by lock.
  - BUT bad interleavings possible – while running, another thread could update value of k to something else.
    - Then when say table.remove(k), removing the wrong value.
- Just right:
  - Take expensive computation outside of locks
  - Still need to ensure value hasn't change while doing expensive opn i.e. if (table.lookup(k) == v1)
    - If it hasn't changed => safe for us to update

ATOMICITY

- Think about which are the critical sections and which need to be indivisible.
- GUIDELINE #7: Think in terms of what operations need to be atomic
  - Make critical sections just long enough to preserve atomicity
  - Then design the locking protocol to implement the critical sections correctly
- Think about atomicity first and locks second

DON'T ROLL YOUR OWN

- GUIDELINE #8: Use built-in libraries whenever suitable
- Especially true for concurrent data structures
  - Very difficult to provide fine-grained synchronization without race conditions
  - Use standard thread-safe libraries

Concurrent Building Blocks in Java

- Bad – Synchronized collections i.e. “normal” collections with synchronized keyword put in front of all methods.
  - Achieve thread safety by serializing all access to the collection’s state
  - Poor concurrency, because of collection-wide lock
  - Doesn’t allow multiple access from threads in the same way as concurrent collections.
- Better – Concurrent collections
  - Designed for concurrent access from multiple threads
  - ConcurrentHashMap, CopyOnWriteArrayList
  - Can result in dramatic scalability improvement with little risk

Summary

- Must consider both data races and bad interleavings
- Due to complexity of concurrent programming guidelines are helpful
  - Avoid shared mutable objects if possible
  - Apply locking consistently with consideration of both lock and critical section granularity
  - Leverage built-in libraries

Deadlock

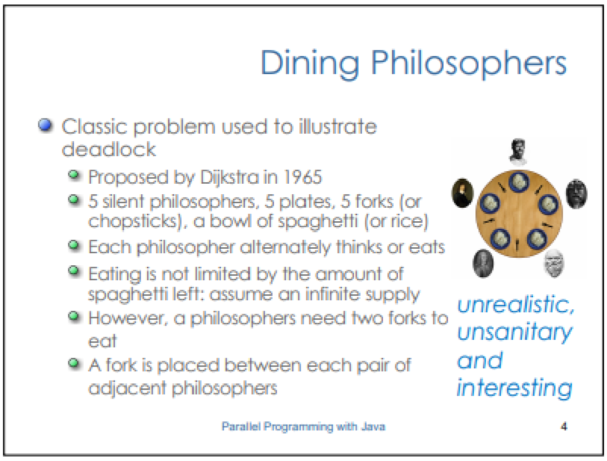
Correctness

- Serial
    - Safety
        - Before state transformed to correct after state
        - Given certain set of inputs, program will run reliably produces correct outputs
        - Property must always be true
- Concurrent
    - Safety
        - Much, much harder to guarantee
        - Complicated by vast number of possible interleavings
        - With re-ordering and interleavings, can be left with state in an incorrect form.
            - Basically, data is corrupted.
    - Liveness:
        - In concurrent situation, always want to be progressing toward solution.
            - aka liveness
        - No serial counterpart
        - A variety of cases
        - Property must eventually become true

Deadlock

- Use locking to ensure safety,
    - But locks are inherently vulnerable to deadlock
    - Where all threads are blocked
    - Indiscriminate locking can cause such lock-ordering deadlocks
- In certain circumstances, can have threads waiting/blocking on lock acquisition such that no thread can progress as all == deadlock!
    - Code never resolves itself; locks never get released and program freezes.
    - Happens under circumstances that you need to have multiple locks *and* lock acquisition in a certain bad order.

Philosophers

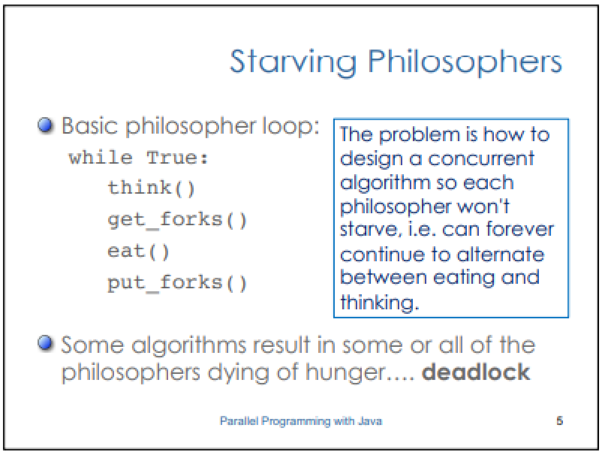


Dining Philosophers

- Classic problem used to illustrate deadlock
  - Proposed by Dijkstra in 1965
  - 5 silent philosophers, 5 plates, 5 forks (or chopsticks), a bowl of spaghetti (or rice)
  - Each philosopher alternately thinks or eats
  - Eating is not limited by the amount of spaghetti left: assume an infinite supply
  - However, a philosophers need two forks to eat
  - A fork is placed between each pair of adjacent philosophers

unrealistic, unsanitary and interesting

Parallel Programming with Java 4



Starving Philosophers

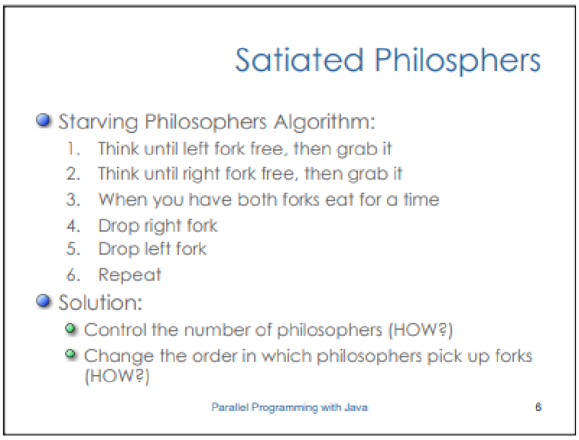
- Basic philosopher loop:

```
while True:
    think()
    get_forks()
    eat()
    put_forks()
```

The problem is how to design a concurrent algorithm so each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking.

- Some algorithms result in some or all of the philosophers dying of hunger.... **deadlock**

Parallel Programming with Java 5



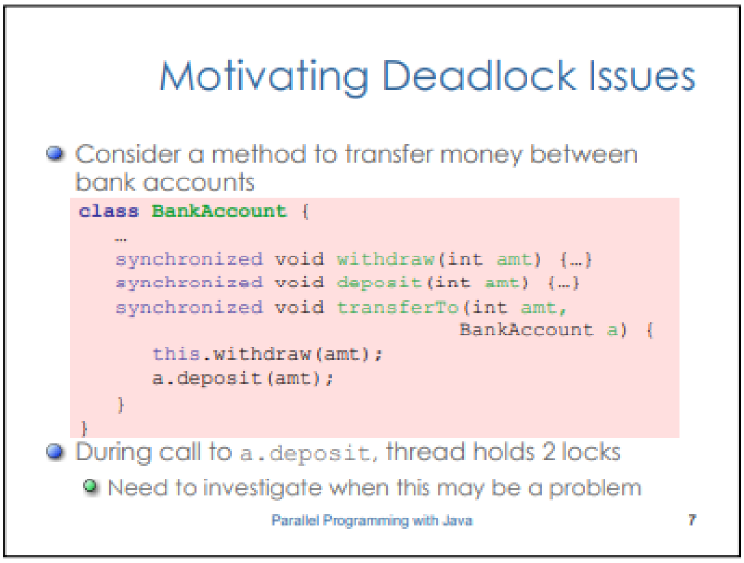
Satiated Philosphers

- Starving Philosophers Algorithm:
  1. Think until left fork free, then grab it
  2. Think until right fork free, then grab it
  3. When you have both forks eat for a time
  4. Drop right fork
  5. Drop left fork
  6. Repeat
- Solution:
  - Control the number of philosophers (HOW?)
  - Change the order in which philosophers pick up forks (HOW?)

Parallel Programming with Java 6

- Need to control # of philosophers and change order in which forks acquired.

Bank Account



Motivating Deadlock Issues

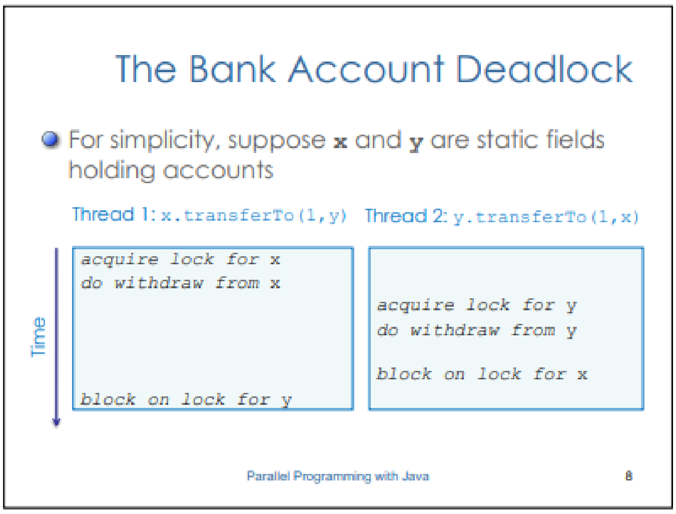
- Consider a method to transfer money between bank accounts

```
class BankAccount {
   …
   synchronized void withdraw(int amt) {…}
   synchronized void deposit(int amt) {…}
   synchronized void transferTo(int amt,
                                BankAccount a) {
      this.withdraw(amt);
      a.deposit(amt);
   }
}
```

- During call to `a.deposit`, thread holds 2 locks
  - Need to investigate when this may be a problem

Parallel Programming with Java 7

- Sync on transferTo to avoid data races
- Have two locks – bank account that you're withdrawing from and bank account you're depositing into
    - Need to acquire each lock before you can do the opn – this is a problem
- E.g. have 2 threads.
    - Thread 1 transfers some amount from bank account x to y.
    - At the same time and interleaved - Thread 2 transfers some amount from bank account y to x.
- Thread 1 starts, withdraws from its account, then suspends.
    - Thread 2 comes in, acquires it's lock on transferTo method, withdraws an amount, then tries to deposit that amount into x.
        - Asks for lock on x, but this **blocks** because lock is already held by thread 1.
    - Thread 1 comes in again and same thing happens.



The Bank Account Deadlock

- For simplicity, suppose **x** and **y** are static fields holding accounts

Thread 1: `x.transferTo(1,y)` Thread 2: `y.transferTo(1,x)`

Thread 1	Thread 2
acquire lock for x do withdraw from x	
	acquire lock for y do withdraw from y
	block on lock for x
block on lock for y	

Parallel Programming with Java 8

TYPES OF DEADLOCK

Deadly Embrace

- Simplest form of deadlock:
    - Two-way deadlock
    - Thread A holds lock L while trying to acquire lock M, while thread B holds lock M while trying to acquire lock L – lock relative to each other

In General

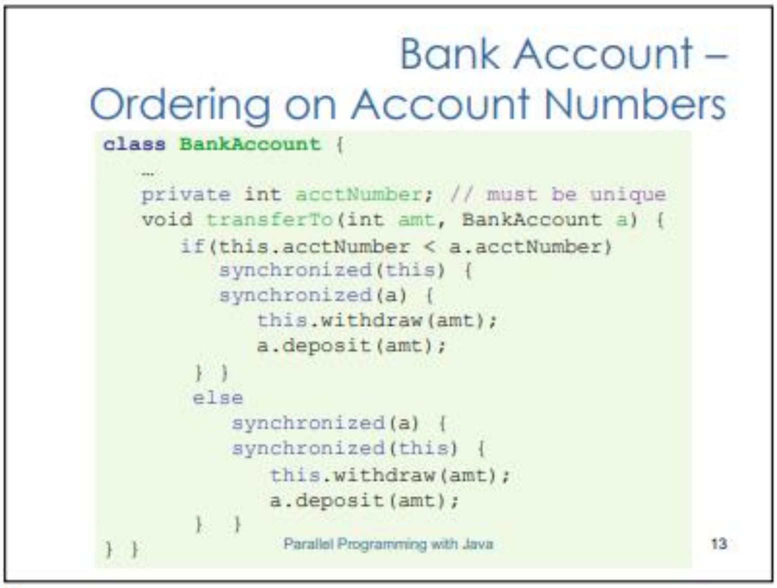
- A deadlock occurs when there are threads T1, …, Tn such that:
    - For i=1,..,n-1, Ti is waiting for a resource held by T(i+1)
    - Tn is waiting for a resource held by T1
- In other words, there is a cycle of waiting
    - Can formalize as a graph of dependencies
    - Deadlock avoidance amounts to ensuring a cycle can never arise in graph dependencies

AVOIDING DEADLOCK

- Java applications do not recover from deadlocks: either part of your prog/all of it is going to freeze up forever
    - When a set of Java threads deadlock, they are permanently out of commission
    - Application may stall completely, a subsystem may stall, performance may suffer
- If there is potential for deadlock it may actually never happen, but usually does under worst possible conditions
    - need to ensure that it can't happen

Example: Bank Account Solutions

- Options for deadlock-proof transfer:
    - 1. Make a smaller critical section: transferTo not synchronized
        - Exposes intermediate state, which may be okay
        - Reducing lock coverage can lead to bad interleavings
    - 2. Coarsen lock granularity: one lock for all accounts allowing transfers between them
        - Works, sacrifices concurrent deposits/withdrawals
        - Damages || performance
    - 3. Give every bank-account a unique number and always acquire locks in the same order (correct solution)
        - Introduce a global ordering
        - Entire program should obey this order to avoid cycles
        - Code acquiring only one lock is fine



Bank Account –
Ordering on Account Numbers

```
class BankAccount {
   …
   private int acctNumber; // must be unique
   void transferTo(int amt, BankAccount a) {
      if(this.acctNumber < a.acctNumber)
         synchronized(this) {
         synchronized(a) {
            this.withdraw(amt);
            a.deposit(amt);
       } }
       else
          synchronized(a) {
          synchronized(this) {
             this.withdraw(amt);
             a.deposit(amt);
       }  }
} }
```

Parallel Programming with Java 13

- Always do lock acquisition from least to highest account number
    - If this account number less than foreign account #, acquire own lock *before* acquire lock of other account
    - If that isn't the case, syn first on that account before acquiring our own.
- Example, say have account 1 and 2, both doing transfers in || - so they're both be trying to acquire each other's locks:
    - Account 1 is first – acquires its own lock, then suspends

- Account 2 goes then – since 2 > 1, executes else part – tries to acquire account 1 lock
    - Can't do that since account 1 holding its own lock – so blocks and suspends.
    - So account 1 goes again.
- Account 1 tries to acquire lock on account 2 – it can do that because account #2 isn't holding that lock.
    - Does the deposit, then releases those locks.
- Locks now available to proceed with account 2.

LOCK-ORDERING DEADLOCKS

- Occur when two threads attempt to acquire the same locks in a different order
    - Threads hold their own locks while trying to acquire locks of the other
    - A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order
    - Requires global analysis of your programs locking behaviour
- A program that never acquires more than one lock at a time will also never deadlock, but often impractical

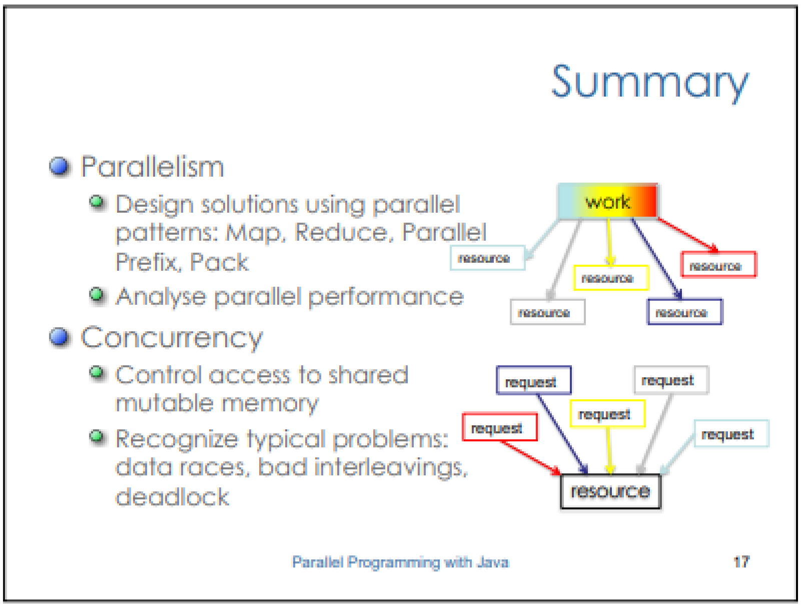
Perspective

- Deadlock in code like account-transfer is difficult to handle
- Don't need to always uses numbers as basis for ordering.
    - Easier to establish type amongst objects and use that as order.
- Easier case: different types of objects
    - Can document a fixed order among types
    - Example: "When moving from hashtable to work queue, never acquire the queue lock while holding the hashtable lock"
- Easier case: objects are in an acyclic structure
    - Can use the data structure to determine a fixed order
    - Binary tree can't have cycles exploit that
    - Example: "If holding a tree node's lock, only acquire other tree nodes' locks if they are children
- Both cases viable alternative to global numbering scheme

Starvation

- Much less common than deadlock
    - A thread is unable to gain regular access to shared resources (e.g., CPU cycles) and is unable to make progress
        - Thread is unable to run for a long time on core
    - Happens when shared resources are made unavailable for long periods by "greedy" threads
        - Some thread wrongly given high priority – dominates computer resources.
        - Or indefinite/semi-definite loop is dominating
- In Java can be caused:

- Inappropriate use of thread priorities
- Indefinite loop



Summary

- Parallelism
  - Design solutions using parallel patterns: Map, Reduce, Parallel Prefix, Pack
  - Analyse parallel performance
- Concurrency
  - Control access to shared mutable memory
  - Recognize typical problems: data races, bad interleavings, deadlock

work

resource resource resource resource resource

request request request request request

resource

Parallel Programming with Java 17