Slide Presentation on Multi-Core Programming Using Java

to accompany

Parallel Programming

CS 471

by Bruce P. Lester

Computer Science Department
Maharishi University of Management
Fairfield, Iowa 52556

Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester

Lesson 6

Java Threads

Creating a Thread

The Runnable interface defines a single method, *run*( ), meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {

  public void run() {
    System.out.println("Hello from a thread!");
  }

  public static void main(String args[]) {
    Runnable x = new HelloRunnable(); //create object
    Thread tx = new Thread(x);    //create Thread object
    tx.start;                     //start the thread
  }
}
```

Join

- The *join*( ) method allows one thread to wait for the completion of another.
- If *t* is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until *t*'s thread terminates.
- Overloads of join allow the programmer to specify a waiting period. t.join(35) waits until t’s thread terminates, or 35 milliseconds, whichever comes first.

Simple Thread Example

```
public class Message implements Runnable {
    int myid;
    public Message(int myid) { this.myid = myid; }
    public void run() {
      System.out.print("Hello from thread ");
      System.out.println(myid);
    }
  }

public static void main(String args[]) {
    Thread t1 = new Thread(new Message(1));
    Thread t2 = new Thread(new Message(2));
    t1.start(); t2.start();
    try  { t1.join(); t2.join() }
    catch (InterruptedException e) {}
}
```

Locks

- Create a new lock object:
  ```
  Lock L = new ReentrantLock( );
  ```
- *Lock* interface and *ReentrantLock* object are found in java.util.concurrent.locks.
- Use the Lock L:
  ```
  L.lock( );
  L.unlock( );
  ```
- Operation of Java locks same as locks in OpenMP.

```
#include <omp.h>
#define n 20    /*dimension of the image*/
#define max 10  /*maximum pixel intensity*/
int i,j,intensity, image[n][n], hist[max];
omp_lock_t L[max];

main( )  {
  ...  /*Initialize the image array*/
  for (i = 0; i < max; i++) hist[i] = 0;
  #pragma omp parallel for private(j,intensity)
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)  {
        intensity = image[i][j];
        omp_set_lock(&L[intensity]);
          hist[intensity] = hist[intensity] + 1;
        omp_unset_lock(&L[intensity]);
      }
    }
}
```

Parallel program for histogram (OpenMP Version)

```
public class Scan_Array implements Runnable {

 public void run(){
  int i;
  for (i = start; i < end; i++) {
    int j, intensity;
    for (j = 0; j < n; j++) {
      intensity = image[i][j];
      L[intensity].lock();
        hist[intensity] = hist[intensity]+1;
      L[intensity].unlock();
    }
  }
 }
```

Java Thread to scan rows “start” to “end”

Create Java Lock Array

```
// create array of lock objects
private static Lock[] L =
     new ReentrantLock[max+1];

// initialize the lock array
public static void initlocks(){
  int i;
  // one lock for each element of hist[]
  for (i=0; i <= max; i++)
    L[i] = new ReentrantLock();
}
```

```
public class Application {
  static int n = 2000; // dimension of the image
  static int max = 10; // maximum pixel intensity
  static int image[][] = new int[n][n]; // image array
  static int hist[] = new int[max+1];   // histogram

  public static void main(String[] args) {
    // create one thread for top half of image
    Runnable top = new Scan_Array(image,hist,0,n/2);
    Thread ttop = new Thread(top);

    // create one thread for bottom half of image
    Runnable bot = new Scan_Array(image,hist,n/2,n);
    Thread tbot = new Thread(bot);

    Scan_Array.initlocks();  //initialize lock array
    ttop.start();  tbot.start();
    try {ttop.join(); tbot.join();}
    catch (InterruptedException e) {};
  }
```

```
public class Scan_Array implements Runnable {
    int image[][];
    int hist[];
    int start;   // start row in image array
    int end;     // ending row in image array

  public Scan_Array(int image[][],int hist[],
                   int start,int end){
    this.image = image;
    this.hist = hist;
    this.start = start;
    this.end = end;
  }

  public void run(){
    // shown in earlier slide
  }
  public static void initlocks(){
    // shown in earlier slide
  }
```

Matrix Multiplication

- Serial Matrix Multiply
- Parallel Matrix Multiply: one thread per element
- Parallel Matrix Multiply: one thread per row
- Parallel Matrix Multiply: one thread per core
- See *Mastering Concurrency Programming with Java 9*, Ch 2, for code listings.

Matrix Multiplication Exercise

- Execute the four versions of the Matrix Multiplication program.
- Compare the execution times for various sizes of the matrix.
- Which versions have the best execution time? Explain the reasons for this.

Programming Project: Merging Sorted Lists

Programming Project: Merging Sorted Lists

- Write a (sequential) Java program to merge two sorted lists, using the algorithm from the OpenMP slides. Record the execution time for list length 2,000,000.
- Write a multi-threaded version of this program and record the execution time.
- Use two threads:
  ThreadX – for each element X[i], binary search Y to find its location j, then write X[i] to Z[i+j].
  ThreadY – for each element Y[i], binary search X to find its location j, then write Y[i] to Z[i+j].

Merging Sorted Lists (con't)

- main( ) method in Application class initializes lists X and Y, and creates ThreadX and ThreadY.
- Use the shell shown on the following slide as a starting point for your program.
- Determine the speedup achieved by your program: sequential execution time divided by parallel execution time.
- Speedup should be close to two using a dual-core processor.

```
import java.lang.Thread;
import java.util.*;
public class Application {
      public static final int n = 2000000;
      public static void main(String[] args) {
            int i;
            int[] X = new int[n];
            int[] Y = new int[n];
            int[] Z = new int[2*n];
            /* input sorted lists X and Y */
              for (i = 0; i < n; i++)
                X[i] = i * 2;
              for (i = 0; i < n; i++)
                Y[i] = 1 + i*2;
         Date t = new Date();
            // insert your code to create
            //    and start the threads here
         Date s = new Date();
         System.out.print("Elapsed Time: ");
         System.out.println(s.getTime()-t.getTime());
```

Lesson 7

Locks in Java

Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester

Synchronized Keyword

- Every Java object created, including every Class loaded, has an associated lock.
- Putting code inside a synchronized block makes the compiler append instructions to acquire the lock on the specified object before executing the code, and release it afterwards (either because the code finishes normally or abnormally).
- Between acquiring the lock and releasing it, a thread is said to "own" the lock. At the point of Thread A wanting to acquire the lock, if Thread B already owns the it, then Thread A must wait for Thread B to release it.

Synchronized Keyword Example

- Parking Cash program
- ParkingCash class
- ParkingStats class
- Sensor class
- Data race causes incorrect results
- Synchronized methods produce correct results.
- See *Java 9 Concurrency Cookbook*, Ch 2, for code listings (Recipe 1).

Read-Write Locks

- A read-write lock allows for a greater level of concurrency in accessing shared data than a mutual exclusion lock.
- It exploits the fact that while only a single thread at a time (a *writer* thread) can modify the shared data, any number of threads can concurrently read the data (*reader* threads).

Peformance Improvement

- Whether or not a read-write lock will improve performance over the use of a mutual exclusion lock depends on the frequency that the data is read compared to being modified, the duration of the read and write operations, and the contention for the data.
- For example, a collection that is initially populated with data and thereafter infrequently modified, while being frequently searched (such as a directory of some kind) is an ideal candidate for the use of a read-write lock.

Choice of Locks

- If updates become frequent, then the data spends most of its time being exclusively locked and there is little, if any increase in parallelism from read-write locks.
- If the read operations are too short, the overhead of the read-write lock implementation (which is inherently more complex than a mutual exclusion lock) can dominate the execution cost.
- Ultimately, only profiling and measurement will establish whether the use of a read-write lock is suitable for your application.

Locks and Collections

- ReentrantReadWriteLocks can be used to improve concurrency in some uses of some kinds of Collections.
- This is typically worthwhile only when the collections are expected to be large, accessed by more reader threads than writer threads, and entail operations with overhead that outweighs synchronization overhead.
- For example, the next slide shows a class using a TreeMap that is expected to be large and accessed in parallel.

Example: Tree Map

```
class RWDictionary {
  private final Map<String, Data> m =
      new TreeMap<String, Data>();
  private final ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
  private final Lock r = rwl.readLock();
  private final Lock w = rwl.writeLock();

public Data get(String key) {  // Read Operation
        r.lock();
        try { return m.get(key); }
        finally { r.unlock(); }
    }
 public Data put(String key, Data value) {  // Write
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
```

Lock Fairness

- The constructor for the *ReentrantLock* class accepts an optional *fairness* parameter.
- When set true, under contention, locks favor granting access to the longest-waiting thread.
- Otherwise the lock does not guarantee any particular access order.
- Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Fairness of Read-Write Locks

- When constructed as non-fair (the default), the order of entry to the read and write lock is unspecified.
- A nonfair lock that is continously contended may indefinitely postpone one or more reader or writer threads, but will normally have higher throughput than a fair lock.

Fair Mode Read-Write Locks

- In fair mode, threads contend for entry using an approximately arrival-order policy.
- When the currently held lock is released, either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock.
- A thread that tries to acquire a fair read lock will block if either the write lock is held, or there is a waiting writer thread. The thread will not acquire the read lock until after the oldest currently waiting writer thread has acquired and released the write lock.
- A thread that tries to acquire a fair write lock will block unless both the read lock and write lock are free (which implies there are no waiting threads).

Lock Timeouts

- Create a new lock object:
```
Lock L = new ReentrantLock( );
```
- Wait indefinitely for Lock L:
```
L.lock( );
```
- Try the lock with a timeout parameter:
```
L.trylock(100,
     Timeout.MILLISECONDS);
```
- Returns true if the lock is acquired within the specified time; otherwise returns false.

Potential Deadlock

Linked List
(protected by Lock L)

Stack
(protected by Lock M)

<u>Thread A</u>

```
L.lock( );
M.lock( );
Remove item from
Linked list and push
onto stack
L.unlock( );
M.unlock( );
```

<u>Thread B</u>

```
M.lock( );
L.lock( );
Pop item from stack
and insert in linked
list
M.unlock( );
L.unlock( );
```

Timeouts to Prevent Deadlock

Linked List (protected by Lock L)

Stack (protected by Lock M)

Thread A

```
L.trylock(100,units);
M.trylock(100,units);
if both locks acquired {
   Remove item from
   Linked list and push
   onto stack
  L.unlock( );
  M.unlock( );  }
else release locks,
wait random time interval,
and start acquiring locks
again
```

Thread B

```
M.trylock(100,units);
L.trylock(100,units);
if both locks acquired {
   Remove item from
   Linked list and push
   onto stack
  M.unlock( );
  L.unlock( );  }
else release locks,
wait random time interval,
and start acquiring locks
again
```

Improving Performance of Locks

- Modify the algorithm to reduce the amount of shared data. This will reduce the need for locking.
- Reduce duration of the locked region by moving some instructions outside of the locked region.
- Use fine-grain locking.

Exercises

J1 The slide “Timeouts to Prevent Deadlock” provides a description of code for acquiring locks. Write this code in Java. Each Thread should continue attempting to acquire the locks until it finally succeeds. Your code should never allow the possibility of deadlock.

J2 How does a *fair* Read-Write lock prevent many active readers from starving a writer? Can more than one writer acquire the lock at the same time?

Exercises

J3 In the slide “Potential Deadlock,” describe a series of event that could lead to a Deadlock.

J4 In the slide “Potential Deadlock,” why do we need two locks? Would the code still work if we only used one lock?

Exercises

J5 A *Parallel Stack* can be implemented using an array to store the values, and a *Top* pointer to indicate the index of the current top element. The following three methods are needed:

Push(x): put item x on top of stack

Pop(y): remove top of stack and return in y

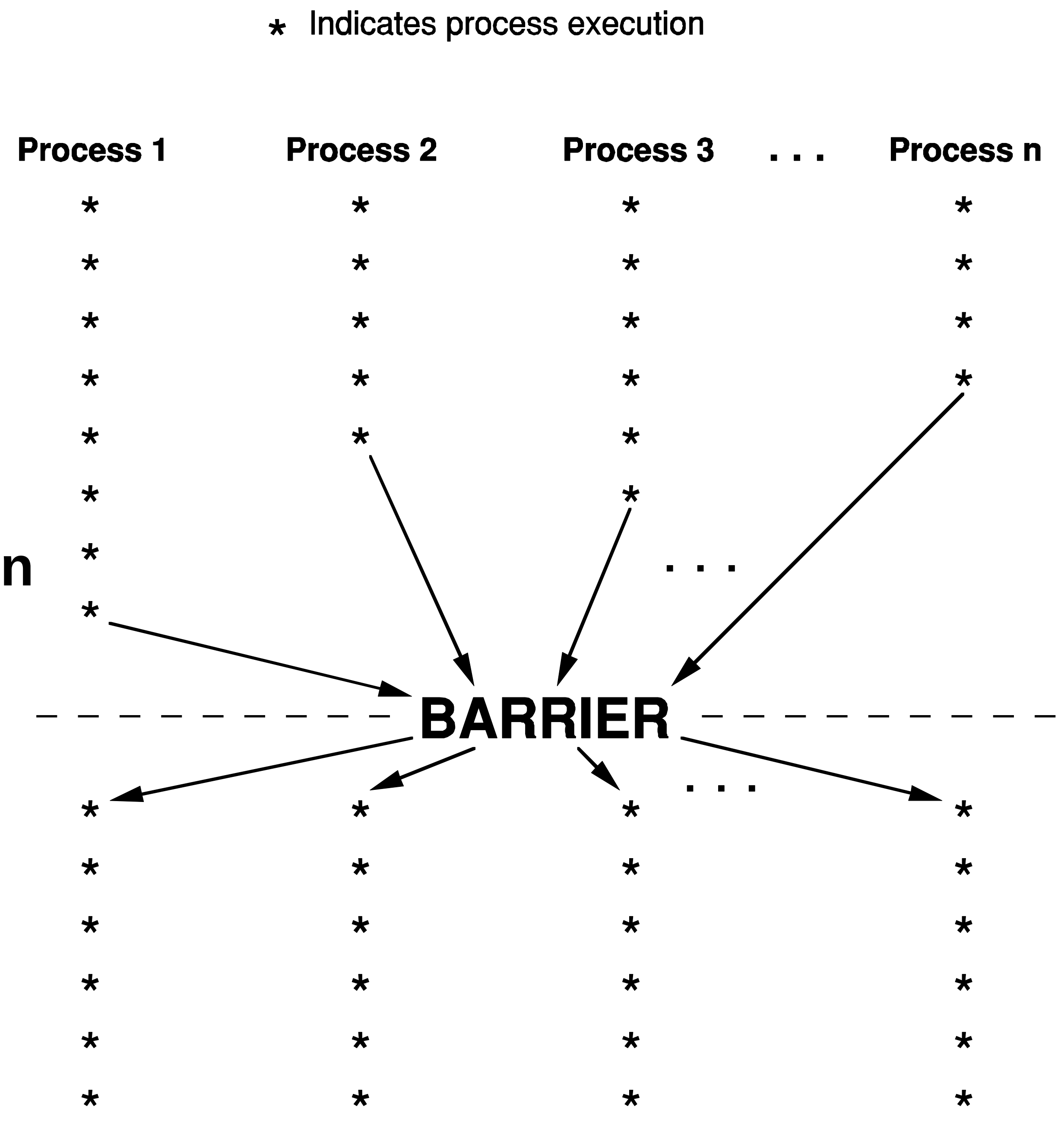
Clear: initialize stack to empty

Implement a Parallel Stack class in Java and test it using many parallel threads.

Lesson 8

Thread Synchronization in Java

Barrier synchronization of processes



Java Barriers

java.util.concurrent has the class CyclicBarrier.

```
// create a barrier object
CyclicBarrier barrier =
     new CyclicBarrier(nthreads);
// use the barrier
     barrier.await( );
```

```
public class Scan_Array implements Runnable{
  int start, end;
  CyclicBarrier barrier;
  public void run() {
    try {
    for (k = 1; k <= numiter; k++) {
      for (i=start; i <= end; i++)
        for (j = 0; j < Application.n; j++)
          /*Compute average of four neighbors*/
          B[i][j] = (A[i-1][j] + A[i+1][j] +
                A[i][j-1] + A[i][j+1]) / 4;
     barrier.await();
     for (i=start; i <= end; i++)
       for (j = 0; j < Application.n; j++)
         A[i][j] = B[i][j];
     barrier.await();}
    }
    catch (BrokenBarrierException b) {}
    catch (InterruptedException e) {}
  }
}
```

```
public class Scan_Array implements Runnable{
  private float A[][] = new float[n][n];
  private float B[][] = new float[n][n];
  int start, end;
  CyclicBarrier barrier;

  public Scan_Array(int start, int end, float A[][],
               float B[][], CyclicBarrier barrier){
    this.start = start;
    this.end = end;
    this.A = A;
    this.B = B;
    this.barrier = barrier;
  }

  public void run() {
  // shown on previous slide
  }
}
```

Jacobi Relaxation using Java Threads

```
public class Application {
  public static int n = 32;
  public static int numiter = 100;

  public static void main(String[] args) {
    float A[][] = new float[n][n];
    float B[][] = new float[n][n];
    CyclicBarrier barrier = new CyclicBarrier(2);
    // each thread gets half the array
    Runnable x = new Scan_Array(0,n/2,A,B,barrier);
    Runnable y = new Scan_Array(n/2+1,n,A,B,barrier);
    Thread tx = new Thread(x);
    Thread ty = new Thread(y);
    tx.start();
    ty.start();
    try {tx.join();
         ty.join();}
    catch (InterruptedException e) { };
  }
```

Jacobi Relaxation using Java Threads

Sequential Jacobi Relaxation using C

```
float  A[n][n], B[n][n], change;

main( ) {
   ...   /* array initialization */
   do {
   for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
        B[i][j] = (A[i-1][j] + A[i+1][j] +
                   A[i][j-1] + A[i][j+1]) / 4.0;
    done = 1;
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        change = fabs(B[i][j] - A[i][j]);
        if (change > tolerance) done = 0;
        A[i][j] = B[i][j];
      }
    }
   } while (!done); /* convergence test */
}
```

Java Barrier Action

CyclicBarrier has optional parameter that specifies a barrier action to be executed after all threads have reached the barrier:

```
// create a barrier action
Runnable barrierAction = ...  ;
// create barrier object
CyclicBarrier barrier =
   new CyclicBarrier(nthreads, barrierAction);
// each thread calls the barrier as before
     barrier.await( );
```

Barrier with Convergence Test

Barrier action can perform aggregation required for convergence test in Jacobi Relaxation:

```
Runnable aggregate = new Runnable() {
    public void run() {
    ...
    // compute global done boolean as AND of
       local done booleans from all threads
    }
};
```

Aggregation Barrier

```
// create barrier action object
Runnable aggregate = new Runnable() {
    public void run() {
    ...
    // compute global done boolean as AND of
       local done booleans from all threads
    }
};

// create barrier object
CyclicBarrier aggregationBarrier =
   new CyclicBarrier(nthreads, aggregate);

// each thread calls the barrier as before
      aggregationBarrier.await( );
```

Matrix Search Example

- See *Java 9 Concurrency Cookbook*, Ch 3, for code listings (Recipe 3).

Programming Project: Jacobi Relaxation using Cyclic Barrier

- Jacobi Relaxation for solving a differential equation (as described in Java slides Part Three) uses a fixed number of iterations.
- The slides outline a parallel version with number of iterations determined by a convergence test.
- Your job in this project is to write a parallel version using Cyclic Barrier action to perform the converence test.
- Also, write a sequential version and compare the performance to the parallel version for n = 10,000 and tolerance = .1

Lesson 9

Java Executors
and Lambda Expressions

References:
http://tutorials.jenkov.com/java/lambda-expressions.html

Thread Pools

- Thread pools manage a pool of worker threads.
- The thread pools contains a work queue which holds tasks waiting to get executed (*Runnable* objects).
- These threads are constantly running and are checking the work query for new work. If there is new work to be done, they execute this *Runnable*.
- The Executor framework contains interfaces and classes to support thread pools.
- Executors.newFixedThreadPool(int n) will create n worker threads.

Example Runnable Class

```
public class Task implements Runnable {
  private final int countUntil;

  Task(int countUntil)
    { this.countUntil = countUntil; }

  public void run() {
    long sum = 0;
    for (int i = 1; i < countUntil; i++) {
      sum += i;
    }
    System.out.println(sum);
  }
}
```

Run with Executor Framework

```
public static void main(String[] args) {
  ExecutorService executor =
    Executors.newFixedThreadPool(nThreads);
  for (int i = 0; i < 500; i++) {
    Runnable worker = new Task(10000 + i);
    executor.execute(worker);
  }
  // Accept no new runnables
  //    and finish all in the queue
  executor.shutdown();
  // Wait until all threads finished
  executor.awaitTermination();
  System.out.println("Finished threads");
}
```

Lambda Expressions

- A *lambda expression* is a block of code you can pass around to be executed later, once or multiple times.
- A *lambda expression* is the Java counterpart of a free-standing function definition, for example:

```
(String first, String second) ->
     first.length() – second.length();
```

- If the body of the lambda carries out a computation, you can write it like a method – enclosed in { } and with *return* statements:

Lambda Expressions

- A Java lambda expression is a function which can be created without belonging to any class.
- A Java lambda expression can be passed around as if it was an object and executed on demand.

Lambda Expressions

```
(String first, String second) -> {
  int difference = first.length() < second.length();
  if (difference < 0) return -1;
  else if (difference > 0) return 1;
  else return 0;
}
```

- If lambda has no parameters, supply empty parentheses:

  ```
  Runnable task = () -> {
      for (int i = 0; i < 1000; i++) doWork();  }
  ```

- If the parameter types can be inferred, they can be omitted:

  ```
   (first, second) ->
       first.length() - second.length();
  ```

Functional Interfaces

- An interface with a single abstract method is called a *functional interface* (example: *Runnable* interface).
- Whenever an object of a functional interface is expected, a lambda expression can be supplied instead:

```
Runnable task = ( ) -> {
   for (int i = 1; i < 100; i++)
    System.out.println(“Hello” + i);
};
Thread t = new Thread(task);
```

Functional Interfaces

Matching a Java lambda expression against a functional interface is divided into these steps:

- Does the interface have only one abstract (unimplemented) method?
- Does the parameters of the lambda expression match the parameters of the single method?
- Does the return type of the lambda expression match the return type of the single method?
- If the answer is yes to these three questions, then the given lambda expression is matched successfully against the interface.

Zero Parameters

- If the lambda expression takes no parameters, then you can write your lambda expression like this:

```
( ) -> System.out.println("Zero
parameter lambda");
```

- Notice how the parentheses have no content in between. That is to signal that the lambda takes no parameters.

One Parameter

- If the lambda expression against takes one parameter, you can write like this:

```
(param) -> System.out.println("One
parameter: " + param);
```

- Or you can also omit the parentheses:

```
param -> System.out.println("One
parameter: " + param);
```

Multiple Parameters

- If the lambda expression has multiple parameters, the parameters need to be listed inside parentheses:

```
(p1, p2) ->
    System.out.println("Multiple
    parameters: " + p1 + ", " + p2);
```

- Only when the method takes a single parameter can the parentheses be omitted.

Parameter Types

- Specifying parameter types for a lambda expression is necessary if the compiler cannot infer the parameter types from the functional interface method the lambda is matching.

```
(Car car) -> System.out.println("The
car is: " + car.getName());
```

- The type (*Car*) of the *car* parameter is written in front of the parameter name itself, just like declaring a parameter in a method.

Returning a Value From a Lambda Expression

- You can return values from Java lambda expressions, just like you can from a method.
- You just add a return statement to the lambda function body:

```
(param) -> {
    System.out.println("param: " + param);
    return "return value";
 }
```

Lambdas as Objects

- A Java lambda expression is essentially an object.
- You can assign a lambda expression to a variable and pass it around, like you do with any other object.

```
public interface MyComparator {
   public boolean compare(int a1, int a2); }
MyComparator myComparator =
     (a1, a2) -> return a1 > a2;
boolean result = myComparator.compare(2, 5);
```

Local Variable Capture

A Java lambda can capture the value of a local variable declared outside the lambda body.

```
public interface MyFactory {
    public String create(char[] chars);
}
String myString = "Test";
MyFactory myFactory = (chars) -> {
    return myString + ":" + new
String(chars);
};
```

Effectively Final

- The lambda body now references the local variable *myString* which is declared outside the lambda body.
- This is possible if, and only if, the variable being references is "effectively final", meaning it does not change its value after being assigned.
- If the *myString* variable had its value changed later, the compiler would complain about the reference to it from inside the lambda body.

Instance Variable Capture

A lambda expression can also capture an instance variable in the object that creates the lambda.

```
public class EventConsumerImpl {
    private String name = "MyConsumer";
    public void attach
      (MyEventProducer eventProducer){
        eventProducer.listen(e -> {
            System.out.println(this.name);
        });
    }
}
```

Static Variable Capture

```
public class EventConsumerImpl {
    private static String
      someStaticVar = "Some text";
    public void attach
      (MyEventProducer eventProducer){
        eventProducer.listen(e -> {
          System.out.println(someStaticVar);
        });
      }
}
```

- The value of a static variable is allowed to change after the lambda has captured it.

Method References as Lambdas

If all your lambda expression does is to call another method with the parameters passed to the lambda, the Java lambda implementation provides a shorter way to express the method call.

```
MyPrinter myPrinter = s ->
    System.out.println(s);
MyPrinter myPrinter =
    System.out::println;
```

Method References as Lambdas

You can reference the following types of methods:

- Static method
- Instance method
- Constructor

Constructor References

- Lambda Expression

```
Factory factory =
    chars -> new String(chars);
```

- The constructor can be used as a Lambda (equivalent to above):

```
Factory factory = String::new;
```

Variable Scope in Lambdas

- The body of a lambda expression has the same scope as a nested block.
- The lamdba can access local variables from the enclosing scope, provided their value does not change:

```
public static void out(String text, int count) {
    Runnable r = ( ) -> {
            for (int i = 0; i < count; i++)
                    System.out.println(text);
            }
```

- Lambda can only access variables from an enclosing scope that are *effectively final*.

Free Variables in Lambdas

- The following is a compile-time error:

```
for (int i = 0; i < n; i++) {
  new Thread(() -> System.out.println(i)).start();
}
```

- The lambda tries to capture *i*, but this is not legal because *i* changes.
- This error can be corrected by assigning *i* to a local *final* variable:

```
for (int i = 0; i < n; i++) {
  final locali = i;  // outside the lambda
  new Thread(() ->
    System.out.println(locali)).start();
}
```

Lambdas and Executors

```
public static void main(String[] args) {
  ExecutorService executor =
    Executors.newFixedThreadPool(nThreads);
  for (int i = 0; i < 500; i++) {
    final countUntil = 10000 + i;
    Runnable worker = () -> {
     long sum = 0;
     for (int i = 1; i < countUntil; i++)
          sum += i;
     System.out.println(sum);
    }
    executor.execute(worker);
  }
     executor.shutdown();
     executor.awaitTermination();
}
```

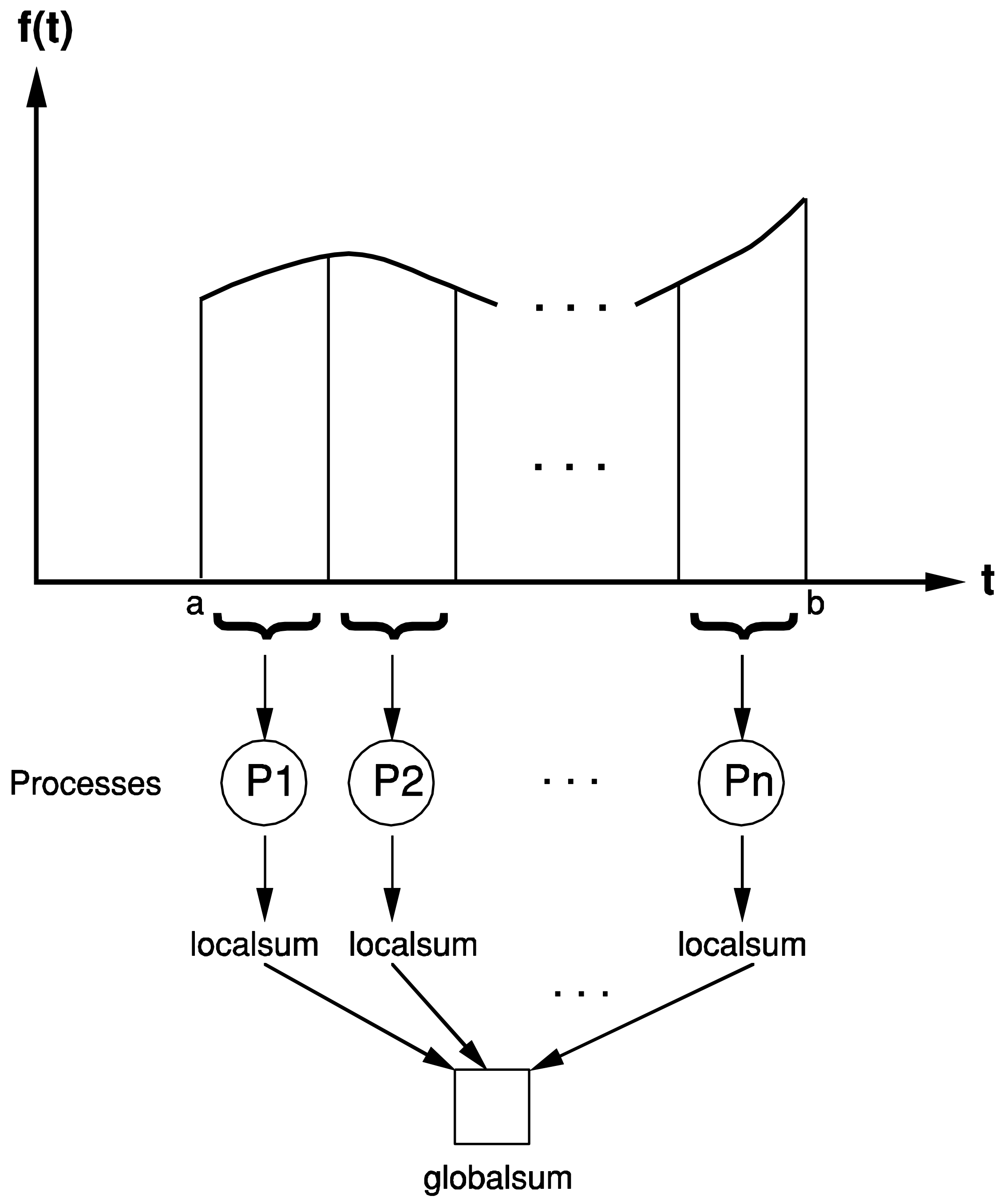
$$\int_a^b f(t)\,dt$$

$w * [\ f(a)/2 + f(a+w) + f(a+2w) + ... + f(a+nw) + f(b)/2\ ]$

Trapezoid Rule for Numerical Integration

Sequential Numerical Integration

```
sum = 0.0;
t = a;
for (i = 1;, i <= n; i++) {
   t = t + w;  /*Move to next point*/
   sum = sum + f(t); /*Add point to sum*/
}
sum = sum + (f(a) + f(b)) / 2;
answer = w * sum;
```



Parallel Numerical Integration

Numerical Integration using Executors and Lambdas

```
for (i = 0; i < n/size; i++) {
     final int me = i;
     Runnable task = () -> {
       for (int j = 0; j < size; j++)
          result[me] =+ (f(a+w*(me*size+j)));
     };
     exec.execute(task);
}
exec.shutdown();
try {
  exec.awaitTermination(10,TimeUnit.SECONDS);}
catch (InterruptedException e) { };
```

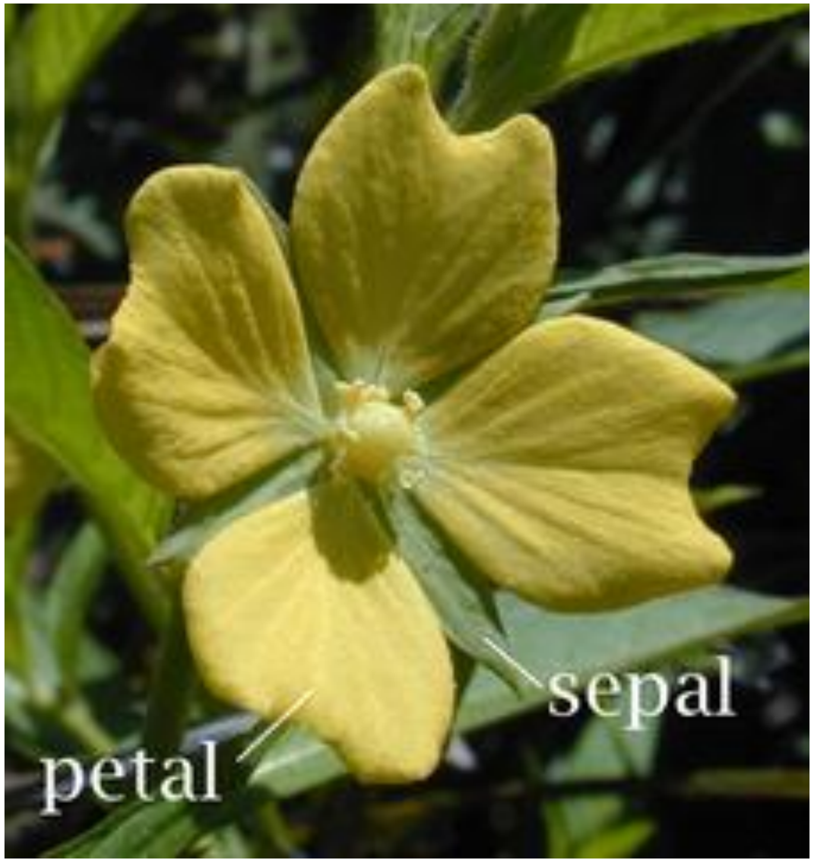
Numerical Integration using Executors and Lambdas

- Using:

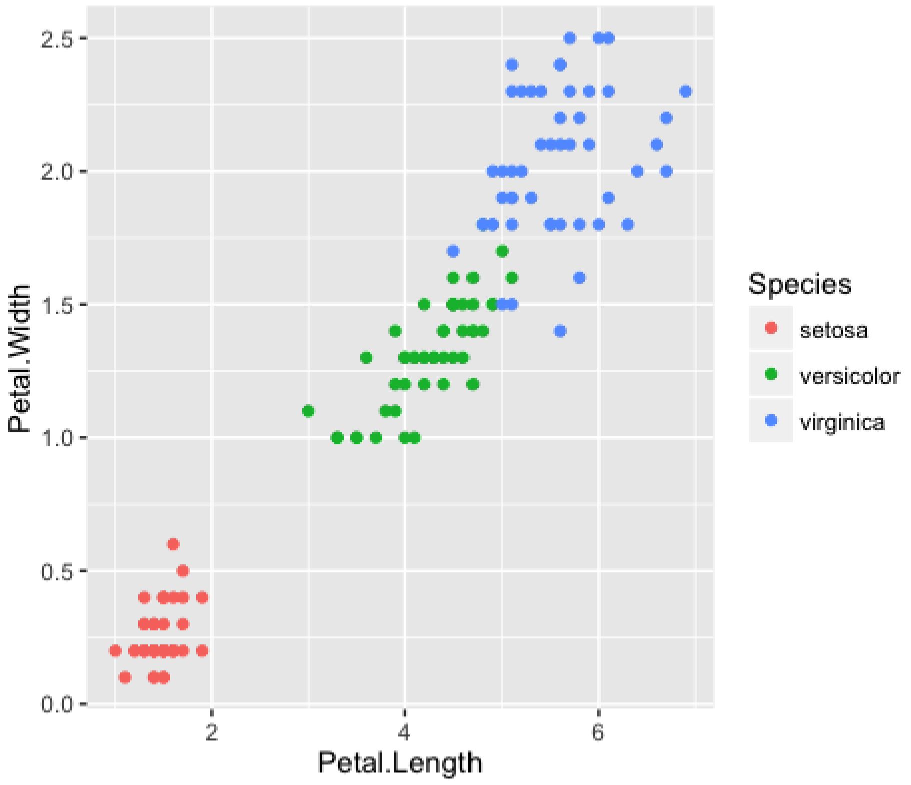
  function f(x) = Sqrt(4 - x*x)
  n = 2,000,000,000 total sample points
  size = 10,000 points per parallel task

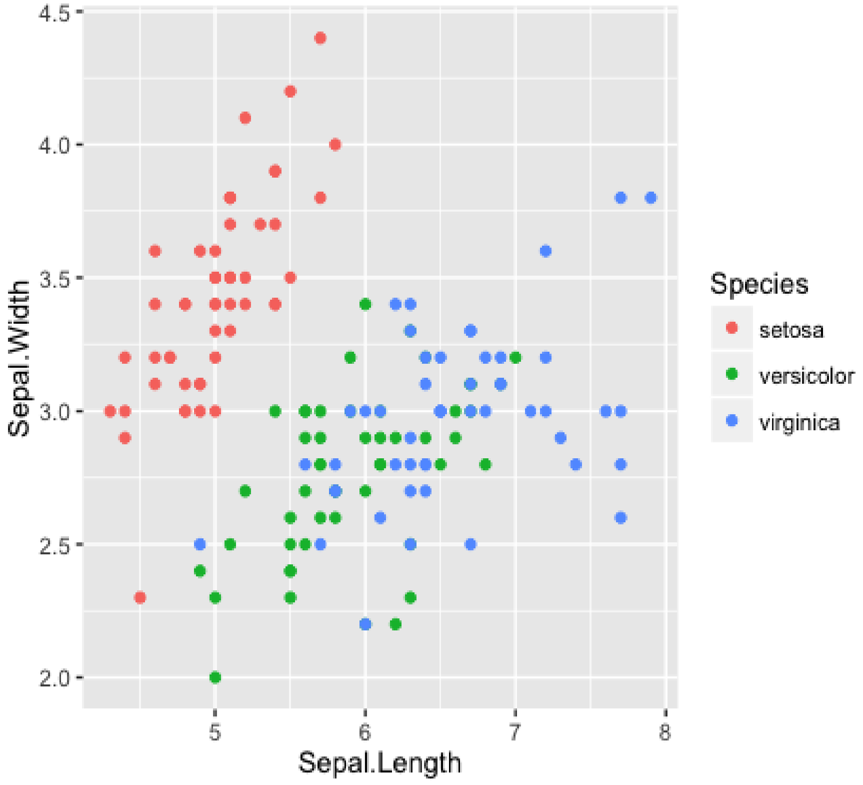
- 6-core processor
- Sequential Execution Time: 16558 milliseconds
- Parallel Execution Time: 5061 milliseconds
- Speedup: 3.3

Iris Flower Species Classification



Species Characteristics





k-Nearest Neighbor Method

- To find species of a flower F, determine Petal length and width, and Sepal length and width.
- Put a point on the scatter graph for F, and find the K nearest neighbor points.
- Determine the species with the highest count among these K neighbors.
- This is the species of the flower F.

k-Nearest Neighbor Method

- There is a *train* dataset and a *test* dataset.
- Each dataset is a collection of Vectors.
- Each Vector contains *n* attributes (each of which is a number), and a *tag*.
- A *Distance* function is needed to determine how far apart two Vectors are.
- For each element in the test dataset, find its distance to every element of test dataset.
- Then select K elements of test dataset with the shortest distance.
- Classify the test Vector using the majority tag among these K test elements.

k-Nearest Neighbor Method

- See *Mastering Concurrency Programming with Java 9*, Ch 3, for code listings.

Programming Project: k-Nearest Neighbor Method

- Write a Sequential and Parallel version of the k-Nearest Neighbor Method.
- For the Parallel version, use the fine-grain concurrent version in the *Mastering* text (Ch 3) as a model.
- Use Executors and Thread Pool to create the Parallel version.
- Whenever a Runnable task is needed, use a Lambda expression (rather than a Runnable class).

Programming Project: k-Nearest Neighbor Method

- Use *train* dataset and *test* dataset as described on p. 66 of Mastering text.
- *Train* dataset has 39,129 instances.
- *Test* dataset has 2,059 instances.
- Compute the speedup for number of threads from 1 up to number of cores on the computer.

Lesson 10

Callable and Future Interfaces

References:
https://www.journaldev.com/1090/java-callable-future-example
https://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.htm

Callable Interface

- In addition to *Runnable*, executors support another kind of task named *Callable*.
- *Callables* are functional interfaces just like runnables, but instead of being *void*, they return a value.
- It is a generic interface.
- Single type parameter corresponding to the return type of the *call*( ) method.

Futures

- *call*( ) method is executed by the Executor and must return an object of the required type.
- When a *Callable* task is sent to an Executor, it will return a *Future* object.
- The *get*( ) method of the Future object gets the value returned by the Callable task.
- If the Callable task has not finished, the execution of get( ) is suspended.

Callable Example

```
Callable<Integer> task = () -> { return 123; };
ExecutorService executor =
      Executors.newFixedThreadPool(4);
Future<Integer> future = executor.submit(task);
System.out.println("future done?" + future.isDone());
Integer result = future.get();
System.out.println("future done?" + future.isDone());
System.out.print("result: " + result);
```

Output:

```
future done? false
future done? true
result: 123
```

```
public class MyCallable implements Callable<String> {
  public String call() throws Exception {
    Thread.sleep(1000);
    return Thread.currentThread().getName();
  }
 public static void main(String args[]){
  ExecutorService executor = Executors.newFixedThreadPool(10);
  List<Future<String>> list = new ArrayList<Future<String>>();
  Callable<String> callable = new MyCallable();
  for(int i=0; i<100; i++){
     Future<String> future = executor.submit(callable);
     list.add(future);
  }
  for(Future<String> fut : list)
     System.out.println(new Date()+ "::"+fut.get());
  executor.shutdown();
 }
}
```

Output

```
Mon Dec 31 20:40:15 PST 2012::pool-1-thread-1
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-2
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-3
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-4
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-5
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-6
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-7
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-8
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-9
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-10
Mon Dec 31 20:40:16 PST 2012::pool-1-thread-2
```

Callable Interface

- *Callable* interface was added in Java 5 to complement existing Runnable interface, which is used to wrap a task and pass it to a Thread or thread pool for asynchronous execution.
- *Callable* actually represent an asynchronous computation, whose value is available via *Future* object.
- All the code which needs to be executed asynchronously goes into *call*() method.

Callable

- *Callable* is also a single abstract method type, so it can be used along with lambda expression on Java 8.
- Both *Callable* and *Future* are parametric type and can be used to wrap classes like Integer, String or anything else.
- When you pass a *Callable* to thread pool, it chooses one thread and executes the *Callable*.
- It immediately return a *Future* object which promises to hold result of computation once done.

Future

- You can then call *get*() method of *Future*, which will return result of computation or block if Computation is not complete.
- If you don't like indefinite blocking then you can also use overloaded *get*() method with timeout.
- *Future* also allows you to cancel the task if its not started or interrupt if its started.

Factorial Example

```
public class HelloWorldApp {
  public static void main(String... args) {
ExecutorService es =
    Executors.newSingleThreadExecutor();
Future result10 = es.submit(new Factorial(10));
Future result15 = es.submit(new Factorial(15));
Future result20 = es.submit(new Factorial(20));
long factorial10 = result10.get();
System.out.println("factorial of 10: " + factorial10);
long factorial15 = result15.get();
System.out.println("factorial of 15: " + factorial15);
long factorial20 = result20.get();
System.out.println("factorial of 20: " + factorial20);
  }
```

```
class Factorial implements Callable<Long> {
    private int number;
    public Factorial(int number){
        this.number = number;
    }
    public Long call() throws Exception {
        return factorial(number);
    }
    private long factorial(int n) {
        long result = 1;
        while (n != 0) {
            result = n * result;
            n = n - 1;
            Thread.sleep(100);
        }
        return result;
    }
}
```

invokeAll( ) method

- *invokeAll*( ) method receives list of Callable objects and returns list of Future objects.
- *submit*( ) method returns immediately.
- *invokeAll*( ) method returns when all the Callable tasks have ended.
- This means all returned Future objects will return true if *isDone*( ) method is called.

invokeAll( ) example

```
ExecutorService executor =
      Executors.newFixedThreadPool(4);
List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");
List<String> results = executor.invokeAll(callables);
executor.shutdown( );
for (Future<String> future: results) {
  String data = future.get( );
  System.out.println(data);
}
```

Best Matching Algorithm for Words

- Use *Levenshtein* distance to measure similarity between two strings: minimum number of insertions, deletions, or substitutions to transform one string into the other.
- For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other:

  kitten → sitten (substitution of "s" for "k")

  sitten → sittin (substitution of "i" for "e")

  sittin → sitting (insertion of "g" at the end).

Programming Project: Best Matching Algorithm for Words

- See *Mastering Concurrency Programming with Java 9*, Ch 5, for code listings of Sequential and Parallel versions.
- Your job is to use a lambda expression for callable task definition in the Parallel version.
- Dictionary of all words download:

  www.crosswordman.com/wordlist.html

Lesson 11

Streams

Streams

- Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections. For example:

```
  int sum = widgets.stream()
                   .filter(b -> b.getColor() == RED)
                   .mapToInt(b -> b.getWeight())
                   .sum();
```

- Here we use *widgets*, a *Collection*<*Widget*>, as a source for a stream, and then perform a filter-map-reduce on the stream to obtain the sum of the weights of the red widgets. (Summation is an example of a reduction operation.)

Streams differ from Collections

- **No storage:** A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
- **Functional in nature:** An operation on a stream produces a result, but does not modify its source.
- For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

Streams differ from Collections

- **Laziness-seeking:** Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization.
- For example, "find the first String with three consecutive vowels" need not examine all the input strings.
- Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations.
- Intermediate operations are always lazy.

Lazy Evaluation of Streams

Produces ten line of output:

```
int result = IntStream.range(1,10)
                .map(i -> {
                   System.out.println(i);
                   return i;
                   })
                .sum();
-----------------------------------------
```

Produces no output:

```
IntStream nresult = IntStream.range(1,10)
                .map(i -> {
                   System.out.println(i);
                   return i;
                   })
```

Streams differ from Collections

- **Possibly unbounded:** While collections have a finite size, streams need not.
- Short-circuiting operations such as *limit(n*) or *findFirst*() can allow computations on infinite streams to complete in finite time.
- **Consumable:** The elements of a stream are only visited once during the life of a stream.
- Like an *Iterator*, a new stream must be generated to revisit the same elements of the source.

Generating a Stream

- From a Collection via the *stream*() and *parallelStream*() methods:

```
ArrayList<Integer> x = new ArrayList<>();
Stream<Integer> sx = x.stream();
```

- From an array via *Arrays.stream(Object*[]*);*
- From static factory methods on the stream classes, such as *IntStream.range(int, int)*;
- The lines of a file can be obtained from *BufferedReader.lines*();
- Streams of random numbers can be obtained from *Random.ints*();

Stream Pipelines

- Stream operations are divided into *intermediate* and *terminal* operations, and are combined to form stream *pipelines*.
- A stream *pipeline* consists of a source (such as a *Collection*, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as *Stream.filter* or *Stream.map*; and a terminal operation such as *Stream.sum* or *Stream.reduce*.
- Intermediate operations return a new stream. They are always *lazy*.
- Traversal of the pipeline source does not begin until the *terminal* operation of the pipeline is executed.

Intermediate Stream Operations

- distinct( ): returns stream with repeated elements eliminated.
- filter( ): returns stream with elements that satisfy a certain condition.
- flatMap( ): converts a stream of streams into a single stream.
- limit( ): returns stream with specified number of elements from original stream.

Intermediate Stream Operations

- map( ): transform elements of steam from one type to another.
- skip(n): ignores first n elements of stream.
- sorted( ): sorts elements of stream.

Terminal Stream Operations

- collect( ): reduce number of elements in stream (see later slide for details).
- count( ): returns number of elements in stream.
- max( ): maximum element of stream.
- min( ): minimum element of stream.
- reduce( ): transforms element of steam into unique object (details shown later).

Terminal Stream Operations

- forEach( ): applies an action to every element of stream.
- findFirst( ): returns first element of stream.
- anyMatch( ): returns true if any element of stream satisfies the specified predicate.
- allMatch( ): true if all elements satisfy predicate.
- toArray( ): returns array with elements of the stream.

Terminal Stream Operations

- *Terminal* operations, such as *Stream.count* or *IntStream.sum*, may traverse the stream to produce a result or a side-effect.
- After the *terminal* operation is performed, the stream *pipeline* is considered consumed, and can no longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream.
- In almost all cases, *terminal* operations are *eager*, completing their traversal of the data source and processing of the pipeline before returning.

Parallel Streams

- All streams operations can execute either in serial or in parallel.
- The stream implementations in the JDK create serial streams unless parallelism is explicitly requested.
- For example, Collection has methods *Collection.stream*() and *Collection.parallelStream*(), which produce sequential and parallel streams respectively:

```
int sum = widgets.parallelstream()
              .filter(b -> b.getColor() == RED)
              .mapToInt(b -> b.getWeight())
              .sum();
```

Sequential Numerical Integration

```
sum = 0.0;
t = a;
for (i = 1;, i <= n; i++) {
  t = t + w;  /*Move to next point*/
  sum = sum + f(t); /*Add point to sum*/
}
sum = sum + (f(a) + f(b)) / 2;
answer = w * sum;
```

Numerical Integration using Parallel Streams

```
w = (b-a)/n;
result = IntStream.range(1,n)
                  .asDoubleStream()
                  .parallel()
                  .map(i -> f(a+i*w))
                  .sum();
answer = w * (result + (f(a)+f(b))/2.0);
```

IntStream, *LongStream*, and *DoubleStream* store primitive values directly without an “object” wrapper.

Numerical Integration using Parallel Streams

- Using:

  function f(x) = Sqrt(4 - x*x)
  n = 2,000,000,000 total sample points

- 6-core processor
- Sequential Execution Time: 16575 milliseconds
- Parallel Execution Time: 4500 milliseconds
- Speedup: 3.7

Sources of a Stream

- Collection interface in Java 8 has *stream*( ) method and *parallelstream*( ) method.
- Any collection (e.g. List, ArrayList, Set) can generate a stream.
- *Array* has methods to generate corresponding stream.

Parallel Streams Example

- See Numerical Summarization Application in *Mastering* text, Ch 8.
- Uses the Map-Reduce pattern.

Reference

Following *collect*( ) method examples taken from:

http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

collect( ) method

- *collect*( ) is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map.
- *collect*( ) accepts a Collector which consists of four different operations: a supplier, an accumulator, a combiner and a finisher.
- Java 8 supports various built-in collectors via the Collectors class. So for the most common operations you don't have to implement a collector yourself.

collect( ) example

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());
```

collect( ) example

```
Map<Integer, List<Person>> personsByAge = persons
     .stream()
     .collect(Collectors.groupingBy(p -> p.age));

personsByAge
     .forEach((age, p) ->
          System.out.format("age %s: %s\n", age, p));
```

collect( ) example

```
Double averageAge = persons
  .stream()
  .collect(Collectors.averagingInt(p->p.age));

System.out.println(averageAge);
```

collect( ) example

```
IntSummaryStatistics ageSummary = persons
    .stream()
    .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12,
            average=19.000000, max=23}
```

Reference

Following twelve slides taken from:

Introduction to Java 8 Streams

https://www.baeldung.com/java-8-streams-introduction

Stream Creation

Streams can be created from different element sources e.g. collection or array with the help of stream() and of() methods:

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> stream = Arrays.stream(arr);
stream = Stream.of("a", "b", "c");
```

Stream from Collection

A stream() default method is added to the Collection interface and allows creating a Stream<T> using any collection as an element source:

```
Stream<String> stream = list.stream();
```

Stream Operations

- Stream Operations are divided into intermediate operations (return Stream<T>) and terminal operations (return a result of definite type).
- Intermediate operations allow chaining.
- Operations on streams do not change the source.

```
long count =
     list.stream().distinct().count();
```

Iterating on a Stream

- Stream API helps to substitute for, for-each and while loops.
- It allows concentrating on operation’s logic, but not on the iteration over the sequence of elements.

```
boolean isExist =
     list.stream().anyMatch(element ->
                              element.contains("a"));
```

Filtering

The *filter()* method allows us to pick stream of elements which satisfy a predicate.

```
Stream<String> stream = list.stream()
          .filter(element ->
               element.contains("d"));
```

Mapping

- To convert elements of a Stream by applying a special function to them and to collect these new elements into a Stream, use the *map*() method:

```
List<String> uris = new ArrayList<>();
uris.add("C:\\My.txt");
Stream<Path> stream = uris.stream()
                    .map(uri -> Paths.get(uri));
```

Flat Map

If you have a stream where every element contains its own sequence of elements, create a stream of these inner elements using the *flatMap*() method:

```
List<Detail> details = new ArrayList<>();
details.add(new Detail());
Stream<String> stream
  = details.stream().flatMap(detail ->
               detail.getParts().stream());
```

Matching

```
boolean isValid =
     list.stream().anyMatch(element ->
          element.contains("h")); // true

boolean isValidOne =
     list.stream().allMatch(element ->
          element.contains("h")); // false

boolean isValidTwo =
     list.stream().noneMatch(element ->
          element.contains("h")); // false
```

Reduction

Stream API allows reducing a sequence of elements to some value according to a specified function with the help of the reduce() method of the type Stream. This method takes two parameters: first – start value, second – an accumulator function.

Reduction Example

Have List<Integer> and want to sum all these elements and some initial Integer (in this example 23). The following code will result in 26 (23 + 1 + 1 + 1):

```
List<Integer> integers =
          Arrays.asList(1,1,1);
Integer reduced = integers.stream()
          .reduce(23, (a, b) -> a + b);
```

Collecting

- The reduction can also be provided by the *collect*( ) method of type Stream.
- This operation will convert a stream to a Collection or a Map.
- There is a utility class *Collectors* which provide a solution for almost all typical collecting operations.
- For some, not trivial tasks, a custom *Collector* can be created.

Collecting Example

This code uses the terminal *collect()* operation to reduce a *Stream*<*String*> to the *List*<*String*>*.*

```
List<String> resultList =
    list.stream()
    .map(element ->
        element.toUpperCase())
    .collect(Collectors.toList());
```

Stream Creation

- There are many ways to create a stream instance of different sources.
- Once created, the instance **will not modify its source,** therefore allowing the creation of multiple instances from a single source.

Reference

Following 24 slides taken from:

The Java 8 Stream API Tutorial

https://www.baeldung.com/java-8-streams

Stream Creation

- Once created, the instance will not modify its source.
- The empty() method should be used in case of a creation of an empty stream:

```
Stream<String> streamEmpty = Stream.empty();
```

- Streams can also be created of any type of Collection (Collection, List, Set):

```
Collection<String> collection =
                  Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection =
                  collection.stream();
```

Stream Creation

- Array can also be a source of a Stream:

```
Stream<String> streamOfArray =
             Stream.of("a", "b", "c");
```

- They can also be created out of an existing array or of a part of an array:

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull =
             Arrays.stream(arr);
Stream<String> streamOfArrayPart =
             Arrays.stream(arr, 1, 3);
```

Stream Builder

- When *builder* is used the desired type should be additionally specified, otherwise it will create an instance of Stream<Object>:

```
Stream<String> streamBuilder =
     Stream.<String>builder().add("a")
     .add("b").add("c").build();
```

Stream.generate( )

- The *generate*() method accepts a Supplier<T> for element generation.
- As the resulting stream is infinite, must specify the desired size.

```
Stream<String> streamGenerated =
Stream.generate(() -> "element").limit(10);
```

- The above creates a sequence of ten strings with the value – “element”.

Stream.iterate( )

- Another way of creating an infinite stream is by using the iterate() method:

```
Stream<Integer> streamIterated =
Stream.iterate(40, n -> n + 2).limit(20);
```

- The first element of the resulting stream is a first parameter of the *iterate*() method.
- For creating every following element the specified function is applied to the previous element.

Stream of Primitives

- Java 8 offers a possibility to create streams out of three primitive types: *int*, *long* and *double*.
- Using the new interfaces alleviates unnecessary auto-boxing allows increased productivity:

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream =
          LongStream.rangeClosed(1, 3);
```

Random Streams

- *Random* class provides a wide range of methods for generation streams of primitives.
- For example, the following code creates a *DoubleStream*, which has three elements:

```
Random random = new Random();
DoubleStream doubleStream =
random.doubles(3);
```

Stream of File

- Generate a *Stream*<*String*> of a text file through the *lines*() method.
- Every line of the text becomes an element of the stream:

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings =
               Files.lines(path);
Stream<String> streamWithCharset =
Files.lines(path, Charset.forName("UTF-8"));
```

Referencing a Stream

- It is possible to have an accessible reference to a Stream it as long as only intermediate operations were called.
- Executing a terminal operation makes a stream inaccessible.

```
Stream<String> stream =
  Stream.of("a", "b", "c")
  .filter(element -> element.contains("b"));
Optional<String> anyElement =
     stream.findAny();
```

- To make previous code work properly some changes should be made:

```
List<String> elements =
Stream.of("a", "b", "c")
  .filter(element -> element.contains("b"))
  .collect(Collectors.toList());
Optional<String> anyElement =
     elements.stream().findAny();
Optional<String> firstElement =
     elements.stream().findFirst();
```

Stream Pipelines

- To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed – the source, intermediate operation(s) and a terminal operation.
- Intermediate operations return a new modified stream.

```
Stream<String> onceModifiedStream =
  Stream.of("abcd", "bbcd", "cbcd").skip(1);
```

Stream Pipelines

- If more than one modification is needed, intermediate operations can be chained.
- If we need to substitute every element of current *Stream*<*String*> with a sub-string of the first few chars:

```
Stream<String> twiceModifiedStream =
   stream.skip(1).map(element ->
            element.substring(0, 3));
```

Stream Pipelines

- A stream by itself is worthless.
- The real thing a user is interested in is a result of the terminal operation, which can be a value of some type or an action applied to every element of the stream.
- Only one terminal operation can be used per stream.

Stream Pipeline Example

```
List<String> list =
 Arrays.asList("abc1","abc2","abc3");

long size = list.stream()
     .skip(1)
     .map(element ->
               element.substring(0, 3))
     .sorted().count();
```

Lazy Invocation

- Intermediate operations are lazy.
- They will be invoked only if it is necessary for the terminal operation execution.

```
private long counter;
private void wasCalled() {
    counter++;
}
// Call method wasCalled() from operation filter():
List<String> list =
     Arrays.asList(“abc1”, “abc2”, “abc3”);
counter = 0;
Stream<String> stream =
     list.stream().filter(element -> {
          wasCalled();
          return element.contains("2");
     });
// Running this code does not change counter.
```

Order of Execution

From the performance point of view, the right order is one of the most important aspects of chaining operations in the stream pipeline:

```
long size = list.stream()
          .map(element -> {
             wasCalled();
             return element.substring(0, 3);
          })
          .skip(2).count();
```

Order of Execution

- Execution of this code will increase the value of the counter by three.
- This means that the *map*() method of the stream was called three times.
- If we change the order of the *skip*() and the *map*() methods, the counter will increase only by one. So, the method *map*() will be called just once.

Order of Execution

```
long size =
     list.stream()
     .skip(2).map(element -> {
          wasCalled();
          return element.substring(0, 3);
          })
     .count();
```

Intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.

collect( ) method

- Reduction of a stream can also be executed by another terminal operation – the collect() method.
- It accepts an argument of the type *Collector*, which specifies the mechanism of reduction.
- There are already created predefined collectors for most common operations.

Collectors

Use the following List as a source for streams:

```
List<Product> productList =
    Arrays.asList(
    new Product(23, "potatoes"),
    new Product(14, "orange"),
    new Product(13, "lemon"),
    new Product(23, "bread"),
    new Product(13, "sugar"));
```

Collectors

Converting a stream to Collection, List or Set:

```
List<String> collectorCollection =
productList.stream().map(Product::get
Name).collect(Collectors.toList());
```

Reducing to String

The joining() method can have from one to three parameters (delimiter, prefix, suffix).

```
String listToString =
    productList.stream()
     .map(Product::getName)
     .collect(
  Collectors.joining(", ", "[", "]")
      );
```

Average or Sum

```
double averagePrice =
      productList.stream()
      .collect(
Collectors.averagingInt(Product::getPrice)
      );

int summingPrice =
      productList.stream()
      .collect(
Collectors.summingInt(Product::getPrice));
```

Parallel Streams Example

- See Searching Data without an Index in *Mastering* text, Ch 9.
- Uses the Map-Collect pattern.

Student Grades Project

Input Data: list of student grade records

Format of input:

Student ID

Department ID

Course Number

Course Date (month, year)

Credits

Student Grade

Student Grades Project

- Your program should use streams to compute the following statistics:
  GPA of every student
  Students with the highest and lowest GPA
  Average GPA of all students in each Department
  Average GPA of each course
- Use random number generator to create input list of 100,000 records for testing.

Programming Project: Histogram of an Image

Programming Project: Histogram of an Image

- The Histogram of a Visual Image program for Java threads is found in Part One of the Java slides.
- The important parameters in determining program performance are:

n	dimension of the pixel array
max	maximum pixel intensity
p	number of cores (threads)

- Also, the method used for storing and accessing the *hist* array will have a profound effect on the overall performance.

Histogram of an Image

- Your job is to write a sequential version and four parallel versions of the Histogram program, and compare the relative performance. (Use n = 20,000 for all versions.)
- Version 1: A central hist array (max = 10) implemented with Atomic Variables.
- Version 2: A central hist array (max = 100) implemented with Atomic Variables.
- Version 3: A central hist array (max = 100) implemented with ordinary variables (no locking).
- Version 4: A local hist array (max = 100) for each thread implemented with ordinary variables, plus a global hist array for the final result.

Histogram of an Image

- All parallel version are implemented *Threads*.
- For each version, record the execution time. Then give a complete explanation of why the parallel execution time gets lower with each successive version.
- Also, give an explanation of why the answers are incorrect for parallel Version 3.
- Have a cover page on your final copy showing the execution time and your explanation for each Version.

Histogram of an Image

- Use the following to initialize the *Image* array:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        image[i][j] = (i+j) % max;
}
```

- The correct final result for the *hist* array should show all elements equal to *n/max*.

Lesson 12

Java
Concurrent Data Structures

Blocking Queue

- **java.util.concurrent contains interface BlockingQueue**
- **Class LinkedBlockingQueue implements the interface:**

```
BlockingQueue<Float> aStream =
           new LinkedBlockingQueue<Float>( );

aStream.put(x);
aStream.take(y);
```

put( ) will block calling thread if queue is full.

take( ) will block calling thread if queue is empty.

Float values stored in a linked list.

Producer-Consumer in Java

```
class Setup {
  void main() {
    BlockingQueue q =
          new LinkedBlockingQueue();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

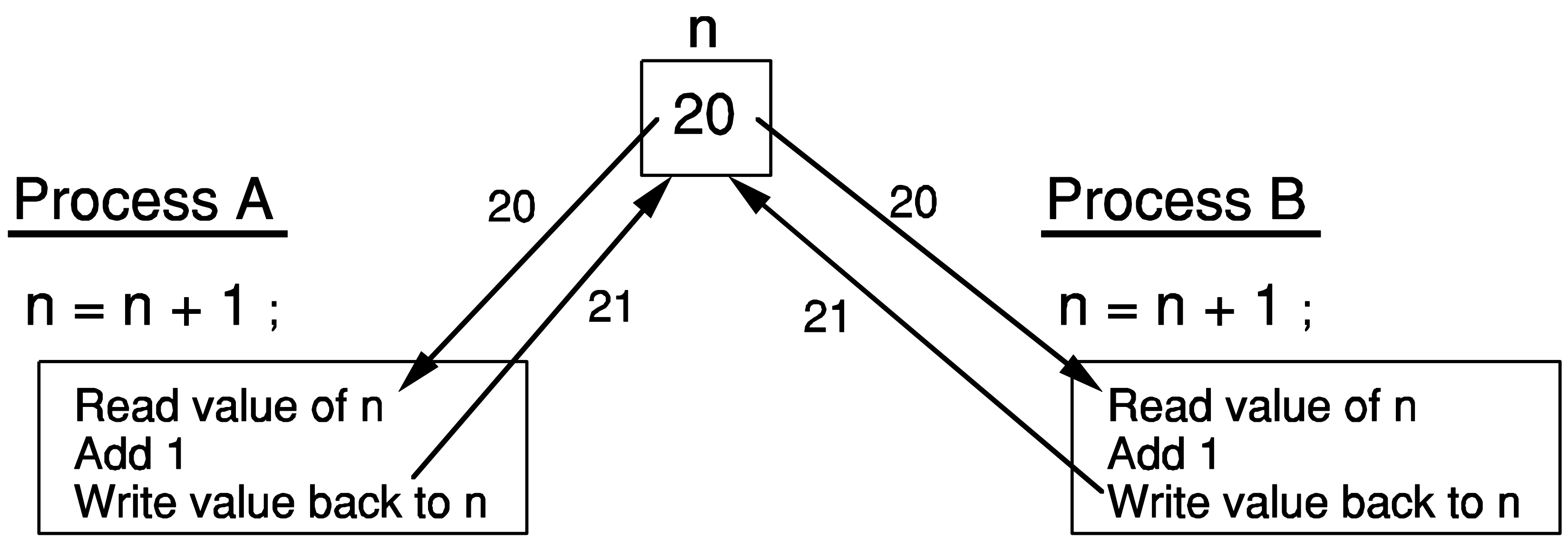
```
class Producer implements Runnable {
   private final BlockingQueue queue;
   Producer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while (true) { queue.put(produce()); }
     } catch (InterruptedException ex) { ... }
   }
   Object produce() { ... }
 }

 class Consumer implements Runnable {
   private final BlockingQueue queue;
   Consumer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while (true) { consume(queue.take()); }
     } catch (InterruptedException ex) { ... }
   }
   void consume(Object x) { ... }
 }
```

Blocking Queue Methods

- **`put(e)`**
  Inserts e into this queue, waiting if necessary for space to become available.
- **`take( )`**
  Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.
- **`offer(e)`**
  Inserts e into this queue if space is available and returns true; returns false if no space is currently available.
- **`peek( )`**
  Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- **`poll(timeout, timeunit)`**
  Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.

Blocking Queue Implementations

- LinkedBlockingQueue: an unbounded queue implemented as a linked list.
- ArrayBlockingQueue(int capacity): a queue with the given capacity, implemented as a circular array.
- PriorityBlockingQueue( ): elements removed in order of their priority, implemented as a heap.
- SynchronousQueue( ): each insert must wait for the corresponding remove; there is no internal storage (like rendezvous in ADA).



Error with shared data

Parallel Stream with Data Race

- Following gives compiler error because free variable *n* is modified in the lambda expression:

```
int n = 0;
result = IntStream.range(1,100)
.map(i -> { n++; return i; } )
.sum();
```

- Compiler error can be avoided by making free variable *n* a *static* variable.

Parallel Stream with Data Race

- Avoid Lamdba expression compiler error:

```
static int n = 0;
  ...
result = IntStream.range(1,100)
           .parallel( )
           .map(i -> {n++; return i;})
           .sum();

System.out.println(n);
```

Results of Data Race

- Parallel Stream with Data Race produces following output on successive executions using a 6-core processor:

```
95
94
93
94
92
91
96
```

- Correct answer produced by sequential form of the Stream is 99.

Atomic Variables

- Data Race can be corrected using Lock (as seen in previous slides).
- Data Race can also be prevented using Atomic Variables: *AtomicInteger*, *AtomicLong*, *AtomicBoolean*.

```
static AtomicInteger n = new AtomicInteger(0);

result = IntStream.range(1,100)
          .parallel( )
          .map(i -> n.incrementAndGet();)
          .sum();
```

Contention for Atomic Variables

- If many parallel threads are frequently accessing a shared variable with a lock or an Atomic Variable, the resulting contention causes threads to wait for each, thus slowing down the program.
- The performance degradation can be reduced by *decentralizing* the shared variable into many shared variables, and assigning groups of threads to separate elements of the group.
- This decentralization can also be accomplished by using *DoubleAdder* or *LongAdder*.

Atomic Adders

- *LongAdder* (or *DoubleAdder*) consists of a group of shared variables with an initially zero sum.
- Method *add(n*) will update the sum by increasing one of the shared variables.
- If there is contention among the threads for access to the *LongAdder* (or *DoubleAdder*), the system will automatically increase the number of shared variables in the group to reduce the contention.
- Method *sum*( ) returns the current total combined across the shared variables maintaining the sum.

Atomic Adders

- If contention for an *Atomic Variable* is high, performance can be improved using *Atomic Adder*:

```
static LongAdder n = new LongAdder();

result = IntStream.range(1,100)
           .parallel( )
           .map(i -> n.increment())
           .sum();

System.out.println(n.sum());
```

Thread-Safe Collections

- Be careful when calling library functions that modify shared data structures, such as collections.
- If parallel threads call library functions to modify a shared collection, a data race may occur.
- If the library function is not “thread-safe,” you can use a lock in your code to provide atomic access to the library function.
- java.util.concurrent library supplies some thread-safe, efficient implementations for collections, including linked lists and hash tables.
- Thread-safe collections use sophisticated algorithms that minimize contention by allowing parallel access to different parts of the data structure.

Example Data Race

- Hash Table Collection
- Thread A begins to insert new element.
- During insertion process, the Hashtable library function must reroute links between hashtable buckets.
- Another parallel Thread B starts traversing the same list, and may follow an invalid link because insertion is not yet complete.

Concurrent HashMap

- Can efficiently support a large number of readers and a fixed number of writers (up to 16) in parallel.
- Obeys same functional specifications as ordinary Hashtable class.
- Even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access.
- Retrieval operations (including *get*) generally do not block, so may overlap with update operations (including *put* and *remove*).

Concurrent LinkedQueue

- Constructs an unbounded, nonblocking queue based on linked nodes that can be safely accessed by multiple threads.
- New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
- This implementation employs an efficient wait-free algorithm based on one described in “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” by M. Michael and M. Scott,1996.

Copy on Write Arrays

- CopyOnWriteArrayList: A thread-safe variant of *ArrayList* in which all modification operations (*add*, *set*, and so on) are implemented by making a fresh copy of the underlying array.
- This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber modifications.
- When an iterator is constructed, it contains a reference to the current array.
- If the array is lated modified, the iterator still has the old array, but the collection's array is replaced.
- As a result, the older iterator has a consistent (but potentially outdated) view that it can access without any synchronization expense.

Exercises

J6 When is it necessary to use a Thread-Safe collection in a Java program?

J7 If a Thread-Safe collection is needed, but not available, what are your other options?

Programming Project: Jacobi Relaxation Using Steams

Programming Project: Jacobi Relaxation

- The multi-core slides for OpenMP show a parallel version of Jacobi Relaxation with a convergence test.
- A parallel Java version implemented with Threads is shown in Part Two of the Java Parallel Programming slides.
- Your job in this project is to write a parallel version using Java 8 parallel streams (with convergence test).
- Also, write a sequential version and compare the performance to the parallel version for n = 10,000 and tolerance = .1

Jacobi Relaxation using Streams

Use the following to initialize the A array:

```
for (i = 1; i <= n; i++)
     for (j = 1; j <= n; j++)
          A[i][j] = 0;
for (i = 0; i < n+2; i++) {
     A[i][0] = 10; A[i][n+1] = 10;
     A[0][i] = 10; A[n+1][i] = 10;
}
```

Jacobi Relaxation using Streams

- For correctness testing, use *n* = 32 and print out the entire *A* array. You should see the 10-values from the borders gradually moving in towards the center of the array with each successive iteration.
- Hint: If done correctly, the Java code for using parallel streams is very compact and simple. Try using an IntStream as the starting point for your code.