COMP 322: Fundamentals of Parallel Programming

Lecture 27: Introduction to Java Threads

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments for Today’s Lecture

- “Introduction to Concurrent Programming in Java”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
  - http://www.oopsla.org/oopsla2007/index.php?page=sub/&id=69

Classification of Parallel Programming Models

- Library approaches
  - POSIX threads
  - Message-Passing Interface (MPI)
  - MapReduce frameworks
- Pseudocomment "pragma" approaches
  - OpenMP
- Language approaches
  - Habanero-Java
  - Unified Parallel C
  - Co-Array Fortran
  - Chapel
  - X10
  - . . .

==> Java takes a library approach with a little bit of language support (synchronized keyword)

Closures

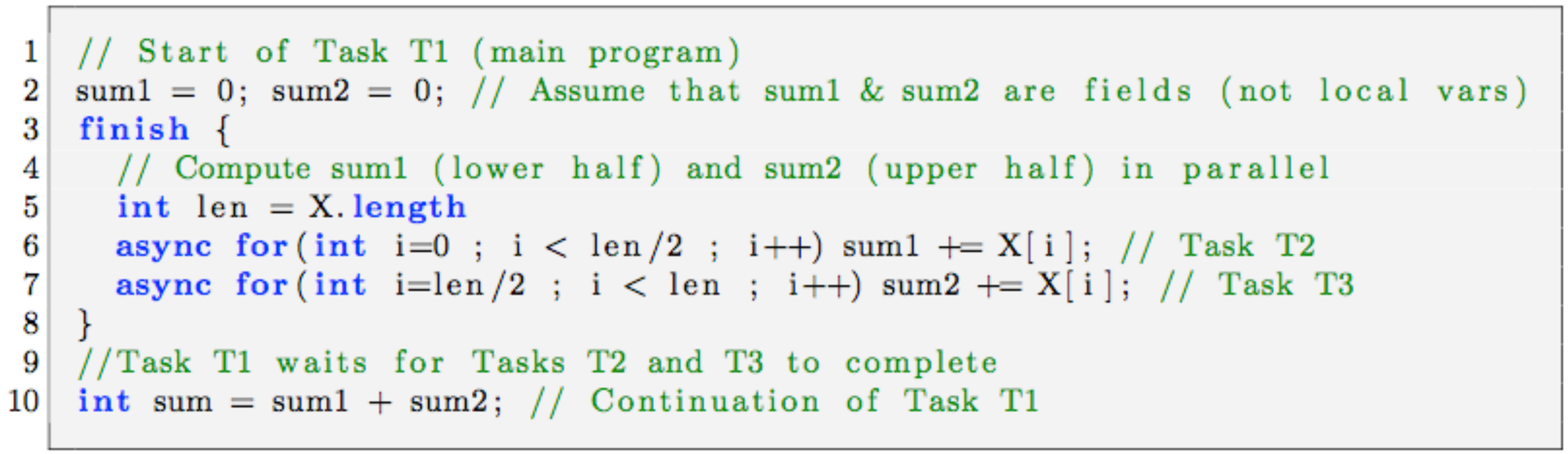
- Library-based approaches to parallel programming require interfaces in which computations can be passed as data
- Recall that a closure is a first-class function with free variables that are bound in function's lexical environment e.g., the anonymous lambda expression in the following Scheme program is a closure

```
; Return a list of all books with at least THRESHOLD copies sold.
(define (best-selling-books threshold)
  (filter
    (lambda (book)
      (>= (book-sales book) threshold))
   book-list))
```

- Note that the value of free variable threshold is captured when the lambda expression is defined

HJ Asyncs and Closures

- The body of an HJ async task is a parameter-less closure that is both created and enabled for execution at the point when the async statement is executed
- An async captures the values of free variables (local variables in outer scopes) when it is created
  - e.g., variable len in Listing 1 below

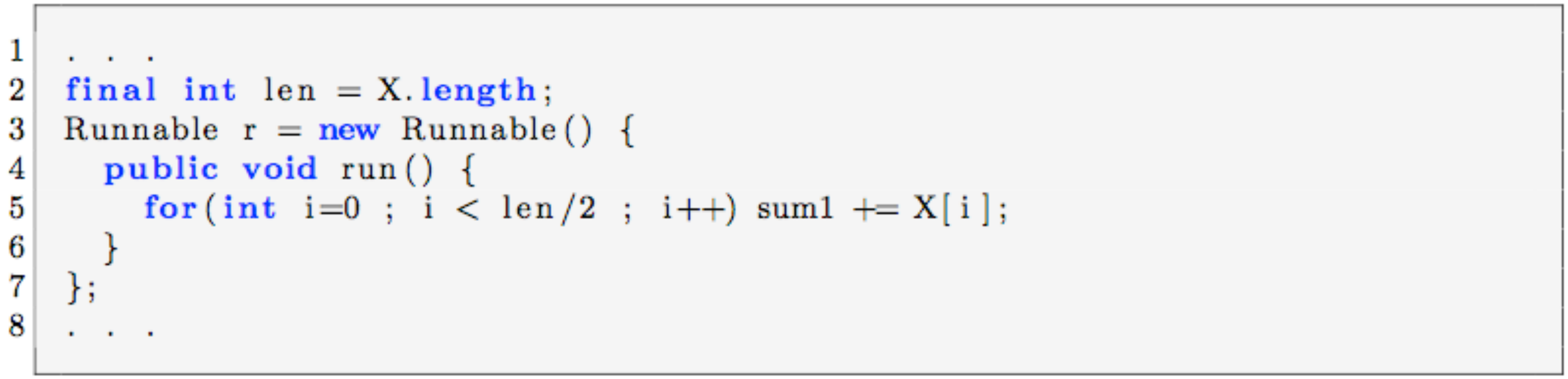


```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
finish {
  // Compute sum1 (lower half) and sum2 (upper half) in parallel
  int len = X.length
  async for(int i=0 ; i < len/2 ; i++) sum1 += X[i]; // Task T2
  async for(int i=len/2 ; i < len ; i++) sum2 += X[i]; // Task T3
}
//Task T1 waits for Tasks T2 and T3 to complete
int sum = sum1 + sum2; // Continuation of Task T1
```

Listing 1: Two-way Parallel ArraySum in HJ

java.lang.Runnable interface

- Any class that implements java.lang.Runnable must provide a parameter-less run() method with void return type
- Lines 3-7 in Listing 2 show the creation of an instance of an anonymous inner class that implements the Runnable interface
- The computation in the run() method can be invoked sequentially by calling r.run()
  - We will see next how it can be invoked in parallel



```
1   . . .
2   final int len = X.length;
3   Runnable r = new Runnable() {
4     public void run() {
5       for(int i=0 ; i < len/2 ; i++) sum1 += X[i];
6     }
7   };
8   . . .
```

Listing 2: Example of creating a Java Runnable instance as a closure

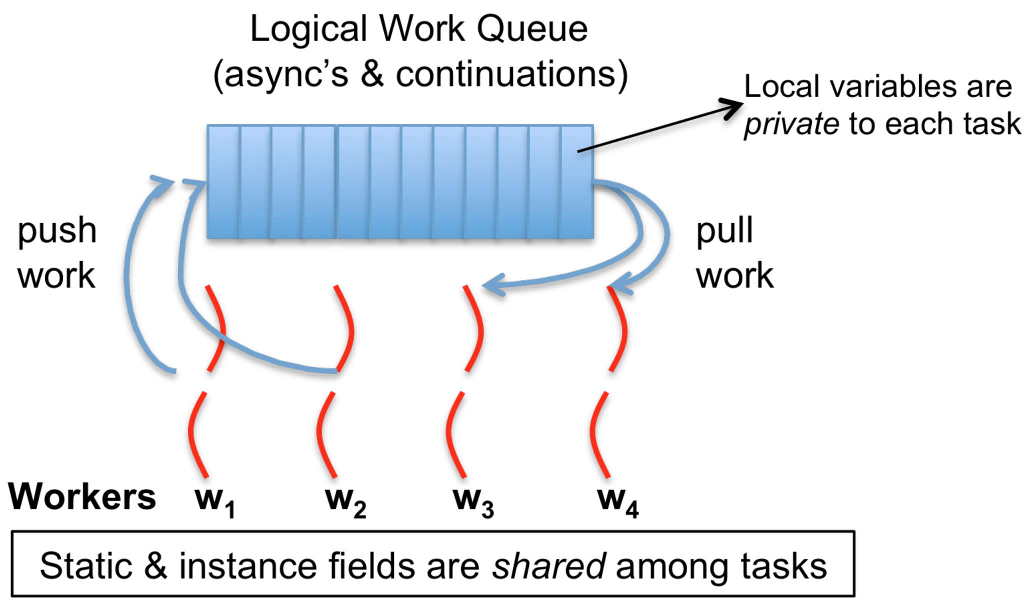
java.lang.Thread class

- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.
- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```java
public class Thread extends Object implements Runnable {
  Thread() { ... } // Creates a new Thread
  Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
  void  run() { ... } // Code to be executed by thread
   // Case 1: If this thread was created using a Runnable object,
   //         then that object's run method is called
   // Case 2: If this class is subclassed, then the run() method
   //         in the subclass is called
  void start() { ... } // Causes this thread to start execution
  void join() { ... } // Wait for this thread to die
  void  join(long m) // Wait at most m milliseconds for thread to die
  static Thread currentThread() // Returns currently executing thread
  . . .
}
```

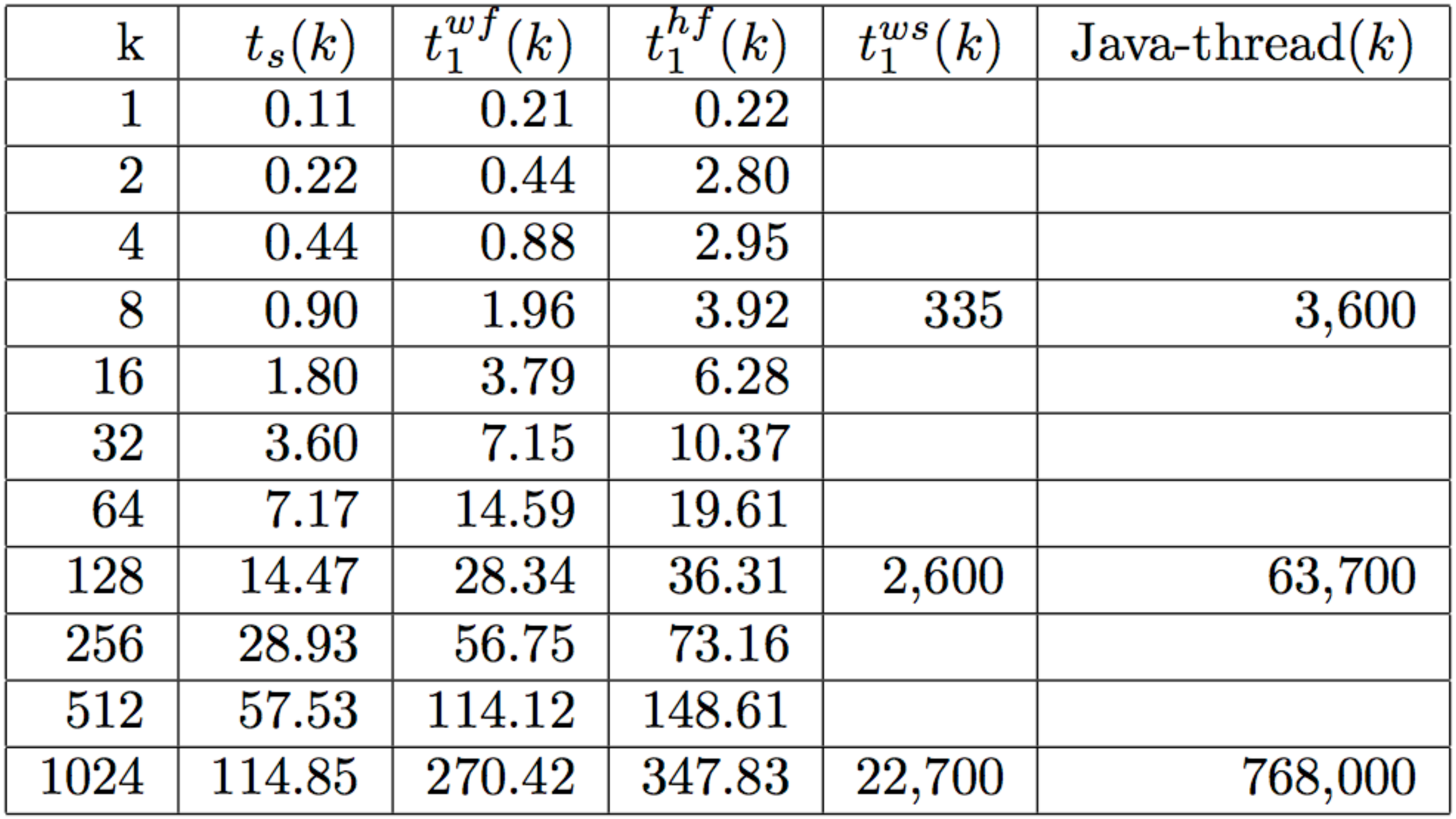
Listing 3: java.lang.Thread class

HJ runtime uses Java threads as workers …


Logical Work Queue
(async's & continuations)
Local variables are *private* to each task
push work
pull work
Workers
w_1
w_2
w_3
w_4
Static & instance fields are *shared* among tasks

- HJ runtime creates a small number of worker threads, typically one per core
- Workers push async's/continuations into a logical work queue
  - when an async operation is performed
  - when an end-finish operation is reached
- Workers pull task/continuation work item when they are idle

… because programming directly with Java threads can be expensive



k	$t_s(k)$	$t_1^{wf}(k)$	$t_1^{hf}(k)$	$t_1^{ws}(k)$	Java-thread(k)
1	0.11	0.21	0.22		
2	0.22	0.44	2.80		
4	0.44	0.88	2.95		
8	0.90	1.96	3.92	335	3,600
16	1.80	3.79	6.28		
32	3.60	7.15	10.37		
64	7.17	14.59	19.61		
128	14.47	28.34	36.31	2,600	63,700
256	28.93	56.75	73.16		
512	57.53	114.12	148.61		
1024	114.85	270.42	347.83	22,700	768,000

Lecture 9: Fork-Join Microbenchmark Measurements (execution time in micro-seconds)

Two ways to specify computation for a Java thread

1. Define a class that implements the Runnable interface and pass an instance of that class to the Thread constructor in line 3 of Listing 3 (slide 7).
   – It is common to create an instance of an anonymous inner class that implements Runnable for this purpose. In this case, the Runnable instance defines the work to be performed, and the Thread instance identifies the worker that will perform the work.
2. Subclass Thread and override the run() method. This is usually inconvenient in practice because of Java's single-inheritance constraint.

start() and join() methods

- A Thread instance starts executing when its start() method is invoked
  - start() can be invoked at most once per Thread instance
  - As with async, the parent thread can immediately move to the next statement after invoking t.start()
- A t.join() call forces the invoking thread to wait till thread t completes.
  - Lower-level primitive than finish since it only waits for a single thread rather than a collection of threads
  - No restriction on which thread performs a join on which thread, so it is possible to create a deadlock cycle using join()
  - No notion of an Immediately Enclosing Finish in Java threads
  - No propagation of exceptions to parent/ancestor threads

Listing 4: Two-way Parallel ArraySum using Java threads

```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Runnable r1 = new Runnable() {
  public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
};
Thread t1 = new Thread(r1);
t1.start();
Runnable r2 = new Runnable() {
  public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
};
Thread t2 = new Thread(r2);
t2.start();
// Wait for threads t1 and t2 to complete
t1.join(); t2.join();
int sum = sum1 + sum2;
```

Callable Objects can be used to create Future Tasks in Java

- Any class that implements java.lang.Callable<V> must provide a call() method with return type V
- Sequential example with Callable interface

```
ImageData image1 = imageInfo.downloadImage(1);
ImageData image2 = imageInfo.downloadImage(2);
. . .
renderImage(image1);
renderImage(image2);
```

Listing 5: HTML renderer in Java before decomposition into Callable tasks

```
Callable<ImageData> c1 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(1);}};
Callable<ImageData> c2 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(2);}};
. . .
renderImage(c1.call());
renderImage(c2.call());
```

Listing 6: HTML renderer in Java after decomposition into Callable tasks

4 steps to create future tasks in Java using Callable objects and Java threads

1. Create a parameter-less callable closure using a statement like

   ```
   Callable<Object> c = new Callable<Object>()
   {public Object call() { return ...; }}; ”
   ```

2. Encapsulate the closure as a task using a statement like

   ```
   FutureTask<Object> ft = new
   FutureTask<Object>(c);
   ```

3. Start executing the task in a new thread by issuing the statement: `new Thread(ft).start();`

4. Wait for the task to complete and obtain its result by issuing the statement: `Object o = ft.get();`

HTML renderer in Java after parallelization of Callable tasks, and corresponding HJ version

Java version

```
Callable<ImageData> c1 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(1);}};
FutureTask<Object> ft1 = new FutureTask<Object>(c1);
new Thread(ft1).start();
Callable<ImageData> c2 = new Callable<ImageData>() {
  public ImageData call() {return imageInfo.downloadImage(2);}};
FutureTask<Object> ft2 = new FutureTask<Object>(c2);
new Thread(ft2).start();
. . .
renderImage(ft1.get());
renderImage(ft2.get());
```

HJ version

```
future<ImageData> ft1 = async<ImageData>{return imageInfo.downloadImage(1);};
future<ImageData> ft2 = async<ImageData>{return imageInfo.downloadImage(2);};
. . .
renderImage(ft1.get());
renderImage(ft2.get());
```

Example of creating Java threads by subclassing Thread (not recommended due to Java's single inheritance rule)

- This program uses two threads: the main thread and a `HelloThread`
  - Each prints a greeting – the order of which is nondeterministic

```
public static void main(String[] args) {
  class HelloThread extends Thread {
     public void run() {
         System.out.println("Hello from thread "
             + Thread.currentThread().getName());}
  }
  Thread t = new HelloThread();    // create HelloThread
  t.start();                       // start  HelloThread
  System.out.println("Hello from main thread");
}
```

- Program execution ends when both user threads have completed

Another Example: Sequential Web Server

```
public class SequentialWebServer {
    public static final int PORT = 8080;
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(PORT);
        while (true) {
            Socket sock = server.accept(); // get next connection
            try {
                processRequest(sock); // do the real work
            } catch (IOException ex) {
                System.err.println("An error occurred ...");
                    ex.printStackTrace();
            }
        }
    }
  // ... rest of class definition
```

Parallelization of Web Server Example using Runnable Tasks

```
public class ThreadPerTaskWebServer { . . .
   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(PORT);
      while (true) {
         final Socket sock = server.accept();
         Runnable r = new Runnable() { // anonymous inner class
            public void run() {
               try {
                  processRequest(sock);
               } catch (IOException ex) {
                  System.err.println("An error occurred ...");
               }
            }
         };
         new Thread(r).start();
      } . . .
```

Possible states for a Java thread (java.lang.Thread.State)

- NEW
  - A thread that has not yet started is in this state.
- RUNNABLE
  - A thread executing in the Java virtual machine is in this state.
- BLOCKED
  - A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  - A thread that is waiting indefinitely for another thread to perform a particular action is in this state e.g., join()
- TIMED_WAITING
  - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state e.g., join() with timeout
- TERMINATED
  - A thread that has exited is in this state.

Summary: Lifecycle of a Java thread

- A thread is created by instantiating a `Thread` object
- A thread is started by calling `Thread.start()` on that object
  - Causes execution of its `run()` method in a new thread of execution
- A thread's state can be inspected by calling Thread.getState()
- A thread terminates by:
  - Returning normally from its `run()` method
  - Throwing an exception that isn't caught by any catch block
  - The VM being shut down
- The JVM shuts down when all user (non-daemon) threads terminate
  - Or when shutdown is requested by `System.exit`, CTRL/C, signal, or other process termination triggers
- Daemon threads are terminated when JVM shuts down
  - Child thread inherits daemon status from parent thread
  - Override by calling `Thread.setDaemon(boolean)` before starting thread
  - Main thread is started as user thread

Announcements

- Homework 4 due today
- Homework 5 (written assignment) due on Friday, April 6th
- Monday's lecture will be a guest lecture by Prof. John Mellor-Crummey on parallel sorting algorithms (Bitonic sort)
- Midterm and HW3 grading is still under way
  - Midterms will be returned by next Wednesday