

---

# COMP 322: Fundamentals of Parallel Programming

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

## Lecture 27: Java Threads

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



# Acknowledgments for Today's Lecture

---

- Lecture 27 handout
- OOPSLA 2007 tutorial by Joe Bowbeer and David Holmes
  - <http://www.oopsla.org/oopsla2007/index.php?page=sub/&id=69>



# Homework 6 (due 5pm on Monday, April 4<sup>th</sup>)

---

- **Assignment:** write a parallel Habanero-Java program to compute the *diameter* of an undirected graph
  - The diameter is the maximum *distance* between any two vertices
  - *distance* = length of a shortest path between the two vertices.
  - diameter will provide the maximum degrees of separation between any two people in a social network.
- **Given:**
  - a sequential HJ program to compute the *diameter* of an undirected graph
    - Execute main program using “hj Diameter *graph.txt*” command
  - three test data sets obtained from the Internet Movie DataBase
    - tiny.txt (seq version runs in < 1 second)
    - moviesGPart1.txt (seq version runs in ~ 10 seconds)
    - moviesG.txt (seq version runs in ~ 20 minutes)
  - you can generate additional data sets by using search interfaces from software available at <http://www.imdb.com/interfaces>



# Kernel of Diameter.main()

---

```
// run breadth first search from each vertex
    int best = -1;
    ST verts = (ST)G.vertices();
    for(java.util.Iterator iter = verts.iterator(); iter.hasNext(); ) {
        Object s = iter.next();
        Pathfinder finder = new Pathfinder(G, s.toString());
        ST secondary_verts = (ST)G.vertices();
        for(java.util.Iterator iter2 = secondary_verts.iterator(); iter2.hasNext(); ) {
            Object v = iter2.next();
            if (finder.isReachable(v.toString()) && finder.distanceTo(v.toString()) > best) {
                best = finder.distanceTo(v.toString());
            } // if
        } // for iter2
    } // for iter
```



# Real-World Parallel Programming Models

---

- Library approaches
  - POSIX threads
  - Message-Passing Interface (MPI)
  - MapReduce frameworks
- Pseudocomment “pragma” approaches
  - OpenMP
- Language approaches
  - Unified Parallel C
  - Co-Array Fortran
  - Habanero-Java
  - 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 Asyncns 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

```
1 // Start of Task T1 (main program)
2 sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
3 finish {
4     // Compute sum1 (lower half) and sum2 (upper half) in parallel
5     int len = X.length
6     async for(int i=0 ; i < len/2 ; i++) sum1 += X[i]; // Task T2
7     async for(int i=len/2 ; i < len ; i++) sum2 += X[i]; // Task T3
8 }
9 //Task T1 waits for Tasks T2 and T3 to complete
10 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 by calling r.run()

```
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

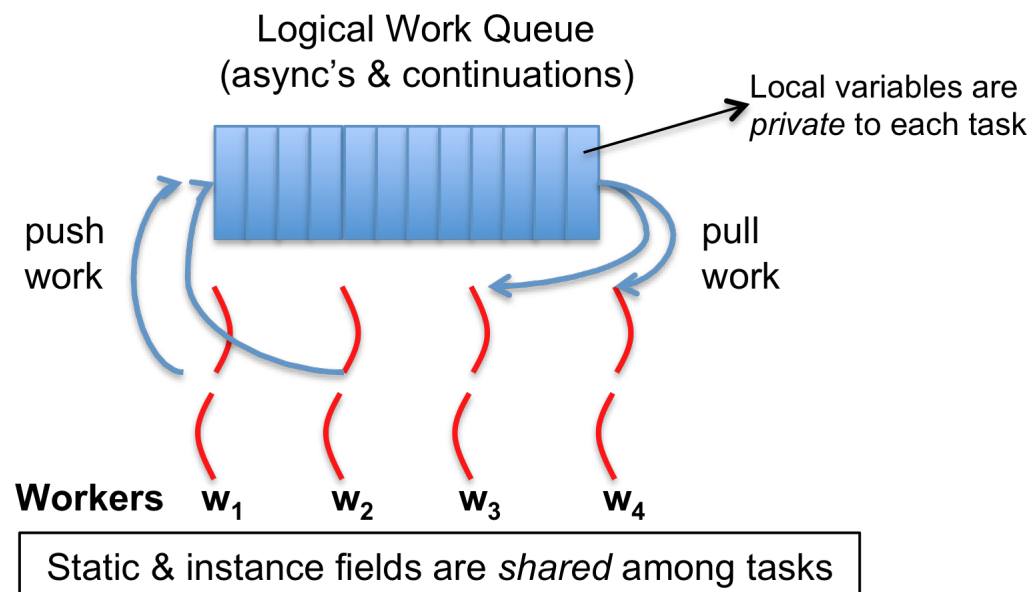
- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.
- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```
1 public class Thread extends Object implements Runnable {
2     Thread() { ... } // Creates a new Thread
3     Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
4     void run() { ... } // Code to be executed by thread
5         // Case 1: If this thread was created using a Runnable object ,
6         //           then that object's run method is called
7         // Case 2: If this class is subclassed, then the run() method
8         //           in the subclass is called
9     void start() { ... } // Causes this thread to start execution
10    void join() { ... } // Wait for this thread to die
11    void join(long m) // Wait at most m milliseconds for thread to die
12    static Thread currentThread() // Returns currently executing thread
13    . . .
14 }
```

Listing 3: java.lang.Thread class



# HJ runtime uses Java threads as workers ...



- 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 11, Table 1: 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.
  - It is common to create an instance of an anonymous inner class that implements Runnable for this purpose, as discussed in Section 2. 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 easy for a programmer to erroneously create a deadlock cycle with join operations
  - No notion of an Immediately Enclosing Finish in Java threads



# Listing 4: Two-way Parallel ArraySum using Java threads

---

```
1 // Start of Task T1 (main program)
2 sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
3 // Compute sum1 (lower half) and sum2 (upper half) in parallel
4 final int len = X.length;
5 Runnable r1 = new Runnable() {
6     public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
7 };
8 Thread t1 = new Thread(r1);
9 t1.start();
10 Runnable r2 = new Runnable() {
11     public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
12 };
13 Thread t2 = new Thread(r2);
14 t2.start();
15 // Wait for threads t1 and t2 to complete
16 t1.join(); t2.join();
17 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

```
1 ImageData image1 = imageInfo.downloadImage(1);
2 ImageData image2 = imageInfo.downloadImage(2);
3 . . .
4 renderImage(image1);
5 renderImage(image2);
```

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

```
1 Callable<ImageData> c1 = new Callable<ImageData>() {
2     public ImageData call() {return imageInfo.downloadImage(1);}};
3 Callable<ImageData> c2 = new Callable<ImageData>() {
4     public ImageData call() {return imageInfo.downloadImage(2);}};
5 . . .
6 renderImage(c1.call());
7 renderImage(c2.call());
```

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



# 4 steps to create future tasks using Callable objects

---

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 get its result by issuing the statement, `Object o = ft.get();`.





# Listing 7: HTML renderer in Java after parallelization of Callable tasks

```
1 Callable<ImageData> c1 = new Callable<ImageData>() {
2     public ImageData call() {return imageInfo.downloadImage(1);}};
3 FutureTask<Object> ft1 = new FutureTask<Object>(c1);
4 new Thread(ft1).start();
5 Callable<ImageData> c2 = new Callable<ImageData>() {
6     public ImageData call() {return imageInfo.downloadImage(2);}};
7 FutureTask<Object> ft2 = new FutureTask<Object>(c2);
8 new Thread(ft2).start();
9 . . .
10 renderImage(ft1.get());
11 renderImage(ft2.get());
```

```
1 future<ImageData> ft1 = async<ImageData>{return imageInfo.downloadImage(1);};
2 future<ImageData> ft2 = async<ImageData>{return imageInfo.downloadImage(2);};
3 . . .
4 renderImage(ft1.get());
5 renderImage(ft2.get());
```

Listing 8: Equivalent HJ code for the parallel Java code in Listing 7

