Java 8 Parallel Stream Internals

(Part 3)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

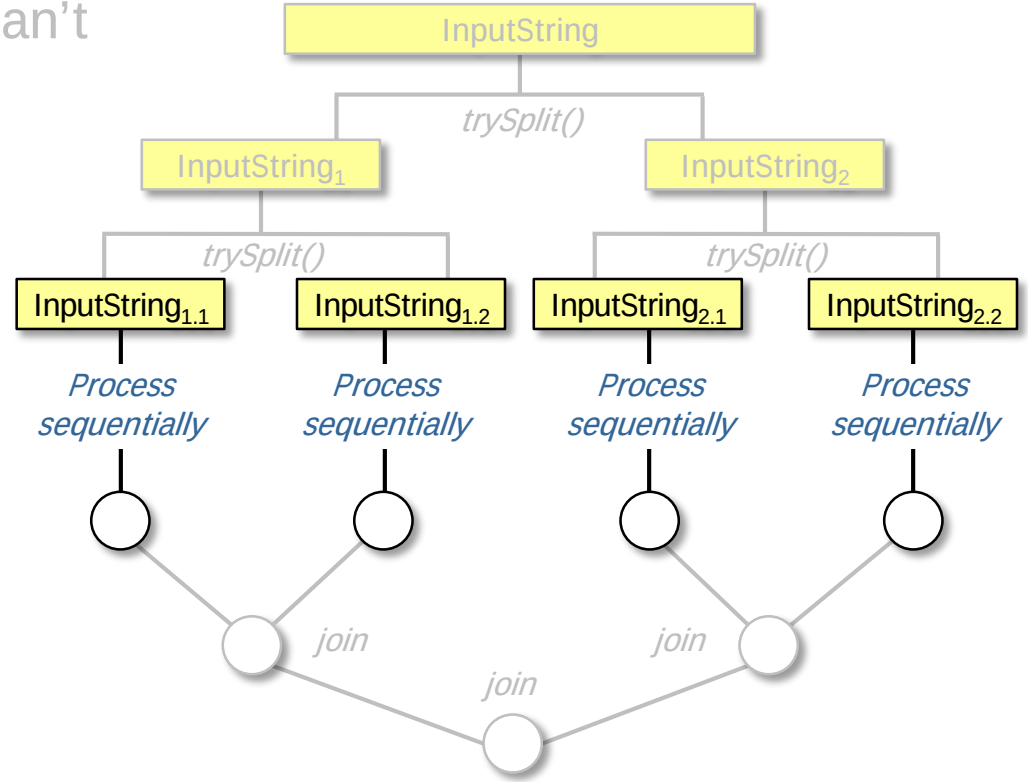
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change & what can't
  - Partition a parallel stream data source into "chunks"
  - Process chunks in parallel

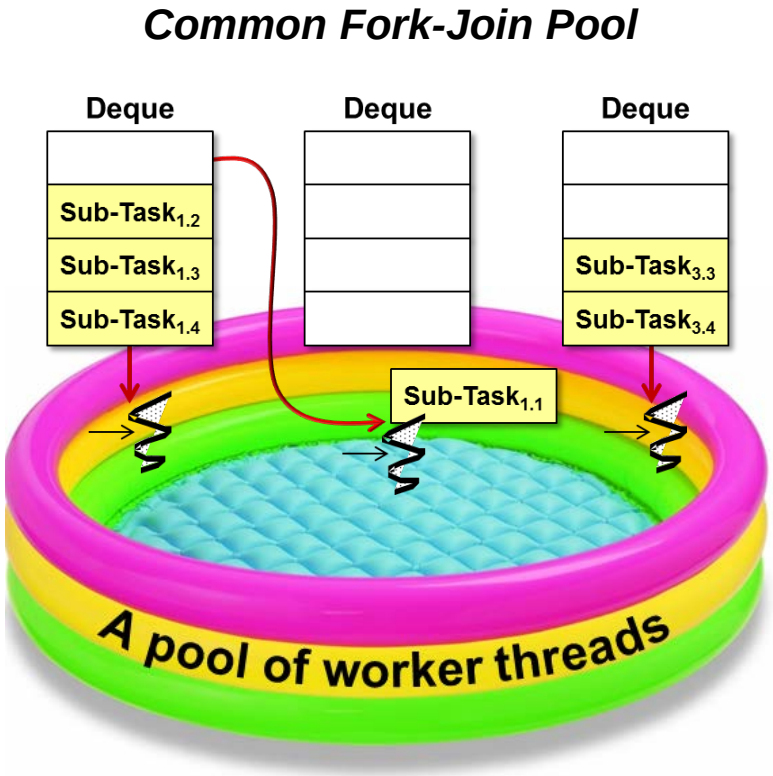


See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz

Processing Chunks in a Parallel Stream

Processing Chunks in a Parallel Stream

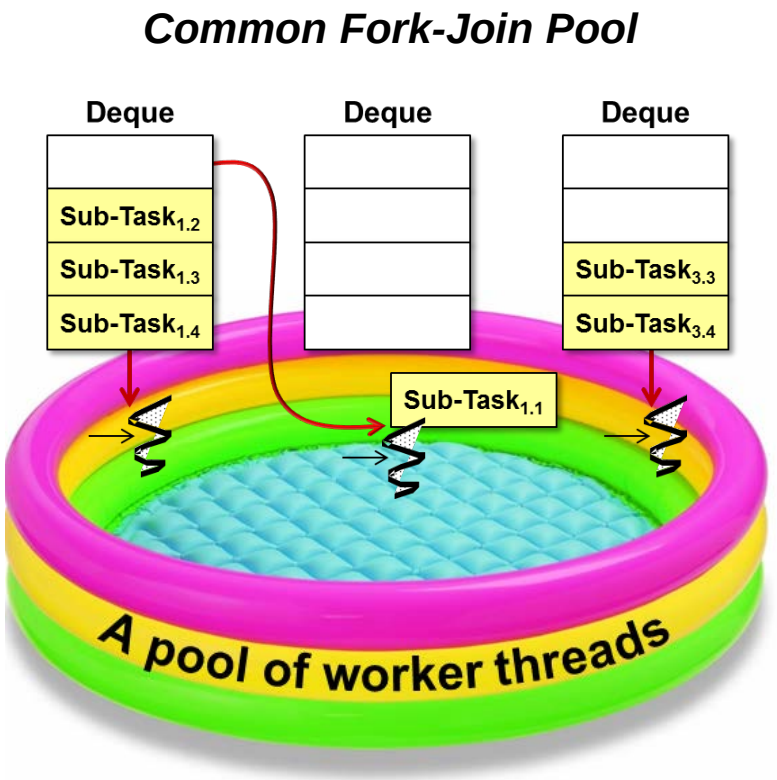
- The chunks created by a spliterator are processed in a common fork-join pool



See gee.cs.oswego.edu/dl/papers/fj.pdf

Processing Chunks in a Parallel Stream

- The chunks created by a spliterator are processed in a common fork-join pool
  - All parallel streams in a process share this common fork-join pool by default



See gee.cs.oswego.edu/dl/papers/fj.pdf

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks

Class ForkJoinPool

java.lang.Object
    java.util.concurrent.AbstractExecutorService
        java.util.concurrent.ForkJoinPool

All Implemented Interfaces:

`Executor, ExecutorService`

```
public class ForkJoinPool
extends AbstractExecutorService
```

An `ExecutorService` for running `ForkJoinTasks`. A `ForkJoinPool` provides the entry point for submissions from non-`ForkJoinTask` clients, as well as management and monitoring operations.

A `ForkJoinPool` differs from other kinds of `ExecutorService` mainly by virtue of employing *work-stealing*: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most `ForkJoinTasks`), as well as when many small tasks are submitted to the pool from external clients. Especially when setting *asyncMode* to true in constructors, `ForkJoinPools` may also be appropriate for use with event-style tasks that are never joined.

A static `commonPool()` is available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use).

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

Processing Chunks in a Parallel Stream

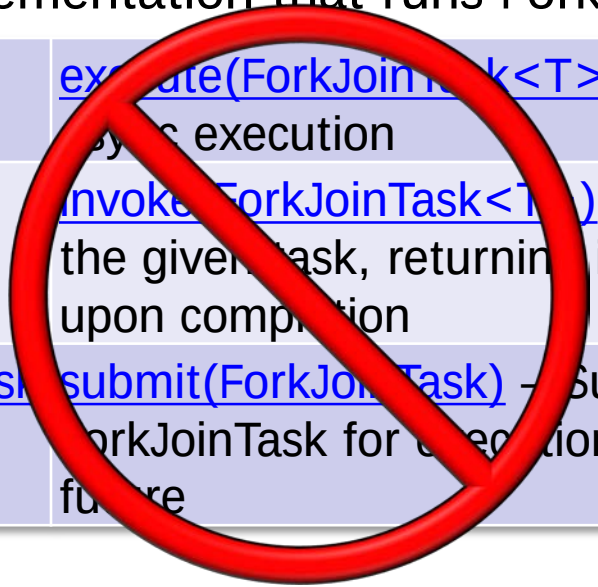
- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks
  - It provides the entry point for submissions from non-ForkJoinTask clients

void	execute(ForkJoinTask<T>) – Arrange async execution
T	invoke(ForkJoinTask<T>) – Performs the given task, returning its result upon completion
ForkJoinTask<T>	submit(ForkJoinTask) – Submits a ForkJoinTask for execution, returns a future

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks
  - It provides the entry point for submissions from non-ForkJoinTask clients

void	execute(ForkJoinTask<T>) – Arrange async execution
T	invoke(ForkJoinTask<T>) – Performs the given task, returning its result upon completion
ForkJoinTask<T>	submit(ForkJoinTask) – Submits a ForkJoinTask for execution, returns a future



These methods are not used by the Java 8 parallel streams framework

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks
  - It provides the entry point for submissions from non-ForkJoinTask clients
  - It also provides management & monitoring operations

int	getParallelism() – Returns the targeted parallelism level of this pool
int	getPoolSize() – Returns the number of worker threads that have started but not yet terminated
int	getQueuedSubmissionCount() – Returns an estimate of the number of tasks submitted to this pool that have not yet begun executing
long	getStealCount() – Returns an estimate of the total number of tasks stolen from one thread's work queue by another

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks
  - It provides the entry point for submissions from non-ForkJoinTask clients
  - It also provides management & monitoring operations

int	getParallelism() – Returns the targeted parallelism level of this pool
int	getPoolSize() – Returns the number of worker threads that have started but not yet terminated
int	getQueuedSubmissionCount() – Returns an estimate of the number of tasks submitted to this pool that have not yet begun executing
long	getStealCount() – Returns an estimate of the total number of tasks stolen from one thread's work queue by another

The Java 8 parallel streams framework uses getParallelism() indirectly..

Processing Chunks in a Parallel Stream

- A ForkJoinTask is a chunk of data along with functionality on that data

Class ForkJoinTask<V>

java.lang.Object
    java.util.concurrent.ForkJoinTask<V>

All Implemented Interfaces:

`Serializable, Future<V>`

Direct Known Subclasses:

`CountedCompleter, RecursiveAction, RecursiveTask`

```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable
```

Abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations.

A "main" `ForkJoinTask` begins execution when it is explicitly submitted to a `ForkJoinPool`, or, if not already engaged in a ForkJoin computation, commenced in the `ForkJoinPool.commonPool()` via `fork()`, `invoke()`, or related methods. Once started, it will usually in turn start other subtasks. As indicated by the name of this class, many programs using `ForkJoinTask` employ only methods `fork()` and `join()`, or derivatives such as `invokeAll`. However, this class also provides a number of other methods that can come into play in advanced usages, as well as extension mechanics that allow support of new forms of fork/join processing.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

Processing Chunks in a Parallel Stream

- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods

ForkJoinTask <T>	fork() – Arranges to asynchronously execute this task in the appropriate pool
V	join() – Returns the result of the computation when it is done
V	invoke() – Commences performing this task, awaits its completion if necessary, and returns its result

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

Processing Chunks in a Parallel Stream

- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods
    - The Java 8 parallel streams framework calls fork()/join() via CountedCompleter

Class CountedCompleter<T>

java.lang.Object
  java.util.concurrent.ForkJoinTask<T>
    java.util.concurrent.CountedCompleter<T>

All Implemented Interfaces:

`Serializable, Future<T>`

```
public abstract class CountedCompleter<T>
extends ForkJoinTask<T>
```

A `ForkJoinTask` with a completion action performed when triggered and there are no remaining pending actions. CountedCompleters are in general more robust in the presence of subtask stalls and blockage than are other forms of ForkJoinTasks, but are less intuitive to program. Uses of CountedCompleter are similar to those of other completion based components (such as `CompletionHandler`) except that multiple *pending* completions may be necessary to trigger the completion action `onCompletion(CountedCompleter)`, not just one. Unless initialized otherwise, the `pending count` starts at zero, but may be (atomically) changed using methods `setPendingCount(int)`, `addToPendingCount(int)`, and `compareAndSetPendingCount(int, int)`. Upon invocation of `tryComplete()`, if the pending action count is nonzero, it is decremented; otherwise, the completion action is performed, and if this completer itself has a completer, the process is continued with its completer. As is the case with related synchronization components such as `Phaser` and `Semaphore`, these methods affect only internal counts; they do not establish any further internal bookkeeping. In particular, the identities of pending tasks are not maintained. As illustrated below, you can create subclasses that do record some or all pending tasks or their results when needed. As illustrated below, utility methods supporting customization of completion traversals are also provided. However, because CountedCompleters provide only basic synchronization mechanisms, it may be useful to create further abstract subclasses that maintain linkages, fields, and additional support methods appropriate for a set of related usages.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountedCompleter.html

Processing Chunks in a Parallel Stream

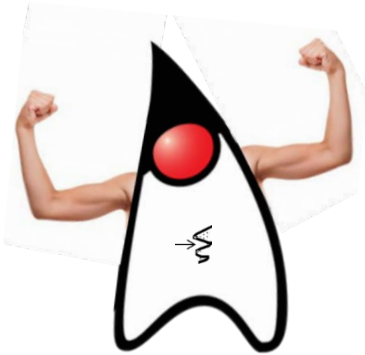
- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods
  - invoke() is essentially fork(); join();

ForkJoinTask <T>	fork() – Arranges to asynchronously execute this task in the appropriate pool
V	join() – Returns the result of the computation when it is done
V	invoke() – Commences performing this task, awaits its completion if necessary, and returns its result

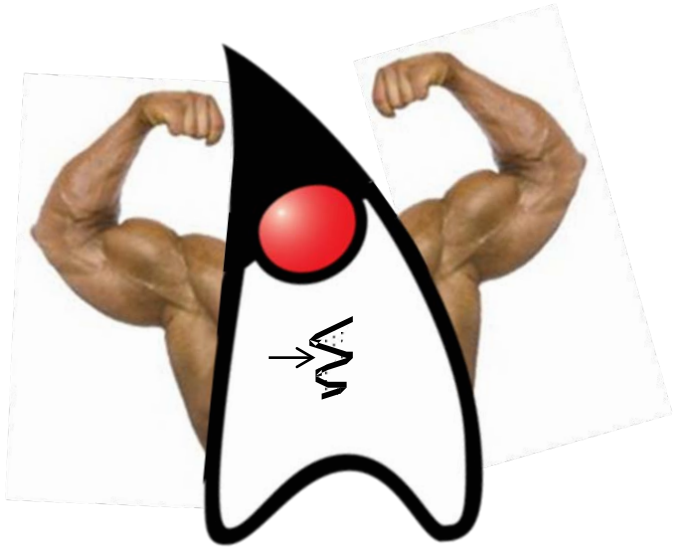
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

Processing Chunks in a Parallel Stream

- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods
  - invoke() is essentially fork(); join();
  - A ForkJoinTask is lighter weight than a Java thread



ForkJoinTask

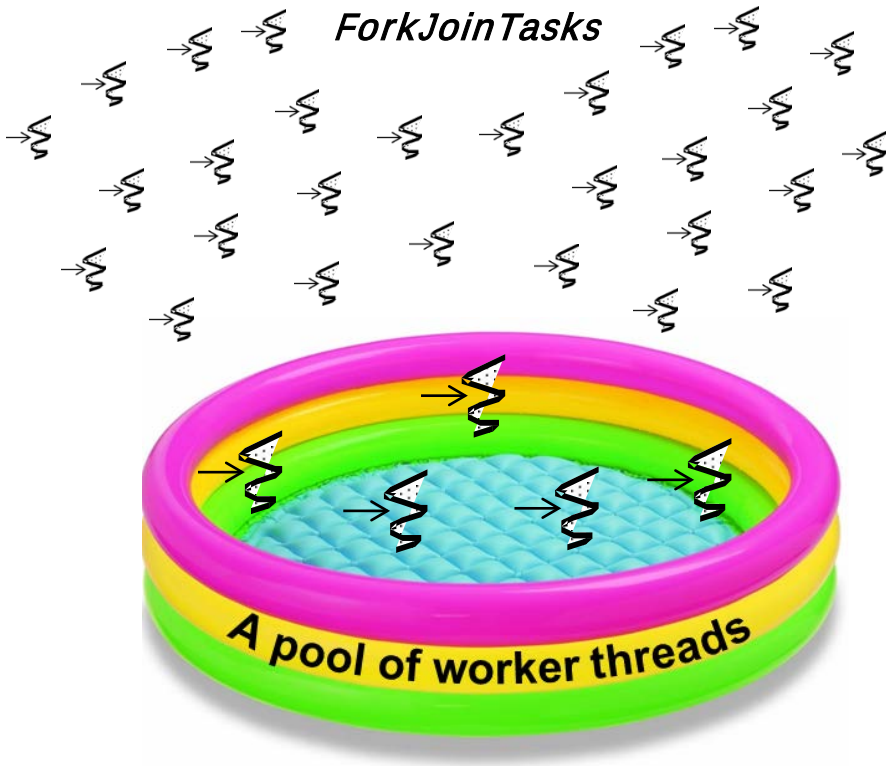


Thread

e.g., it doesn't maintain its own run-time stack

Processing Chunks in a Parallel Stream

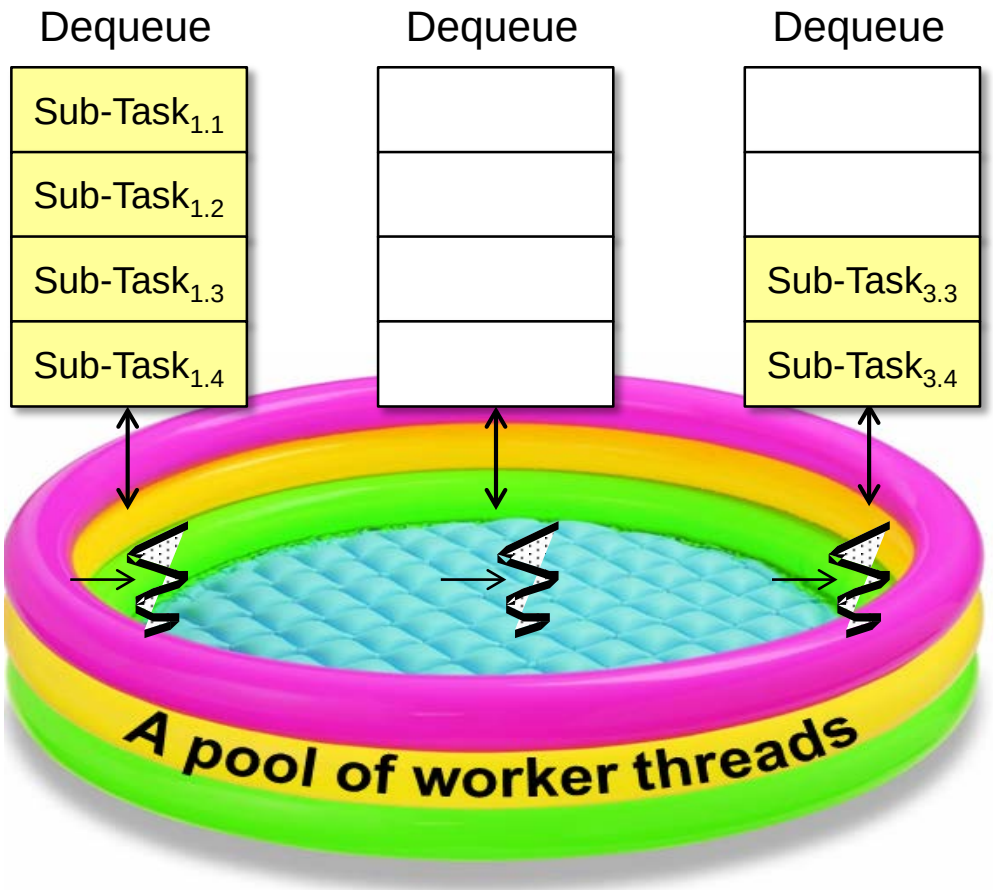
- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods
  - invoke() is essentially fork(); join();
  - A ForkJoinTask is lighter weight than a Java thread
  - A large # of ForkJoinTasks thus run in a small # of Java threads in a ForkJoinPool



See www.infoq.com/interviews/doug-lea-fork-join

Processing Chunks in a Parallel Stream

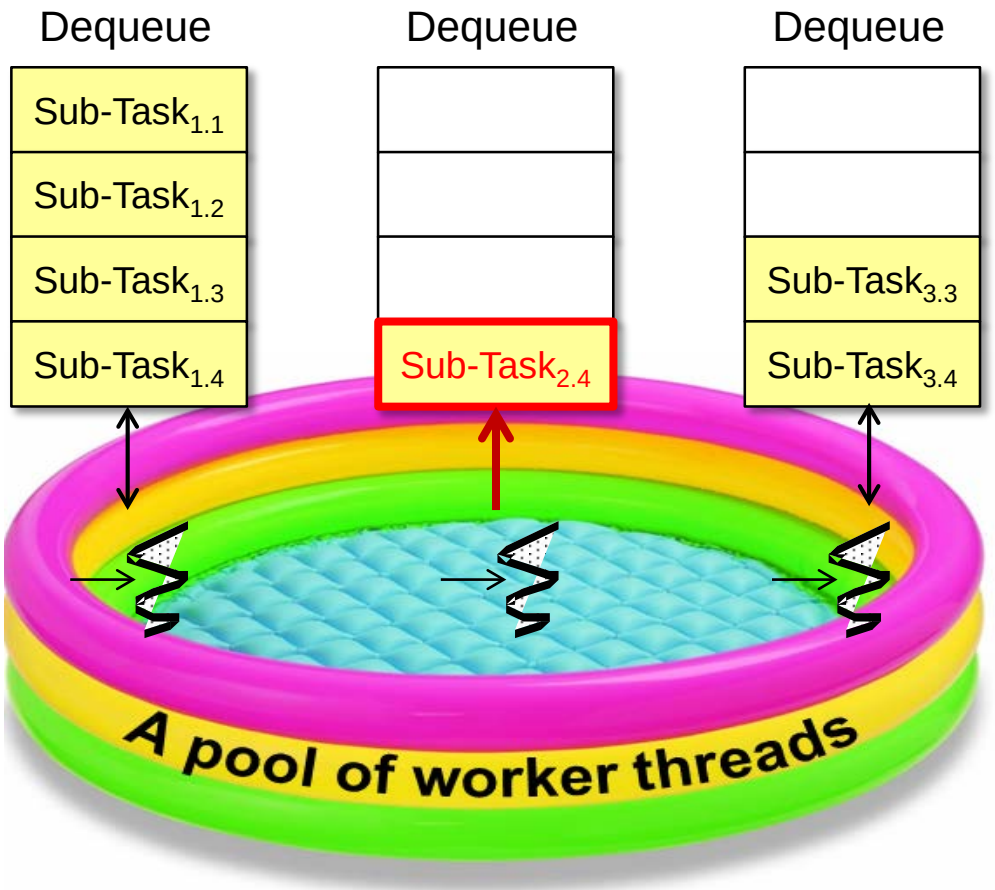
- A circular dequeue is associated with each ForkJoinPool thread



See en.wikipedia.org/wiki/Double-ended_queue

Processing Chunks in a Parallel Stream

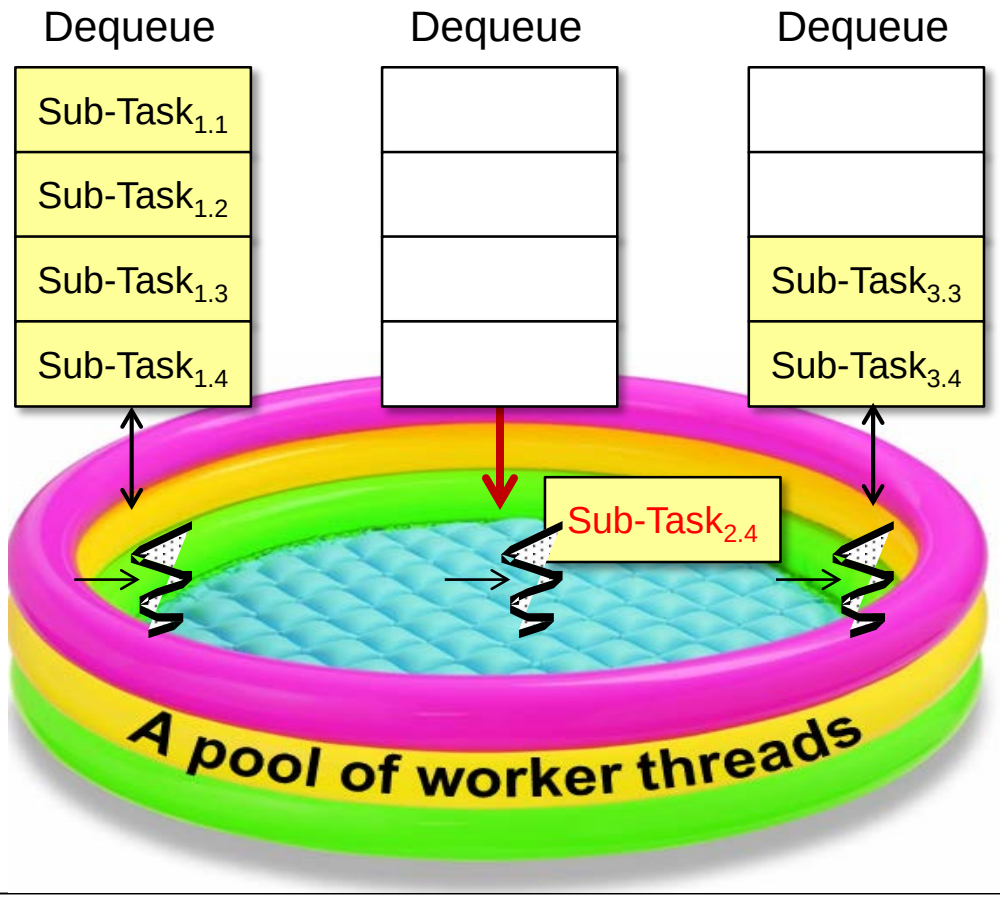
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue



See gee.cs.oswego.edu/dl/papers/fj.pdf

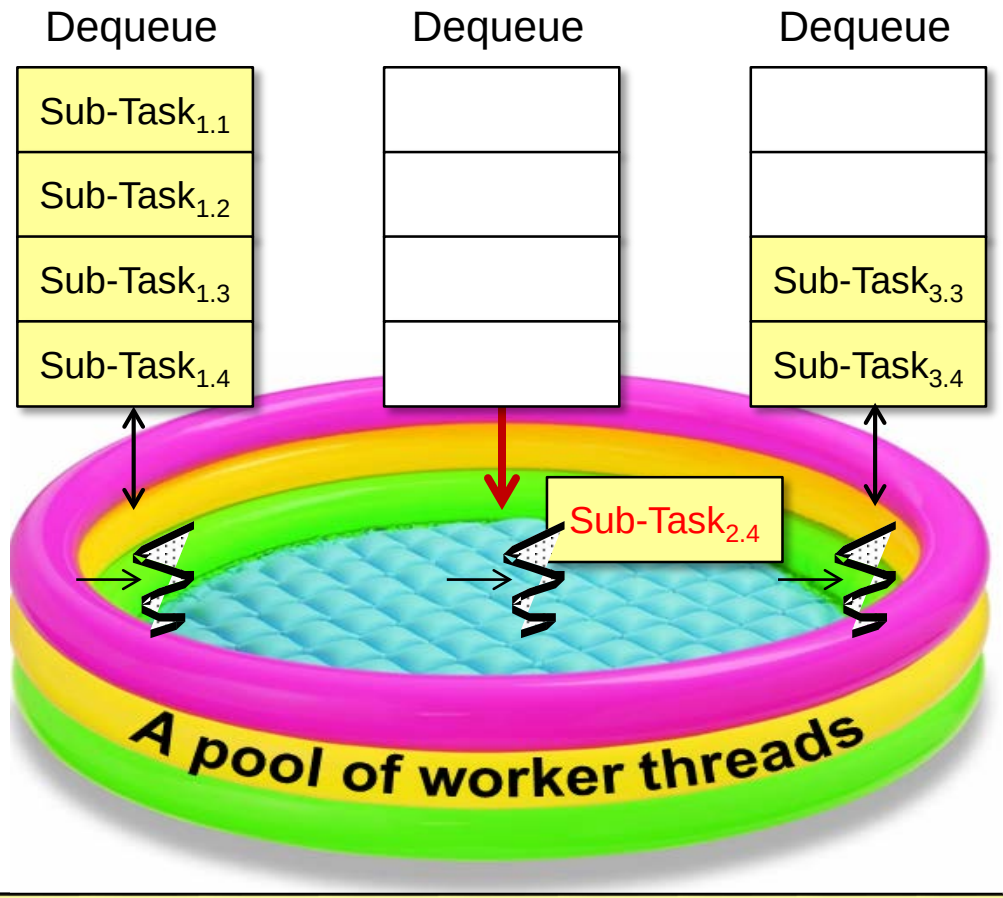
Processing Chunks in a Parallel Stream

- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue



Processing Chunks in a Parallel Stream

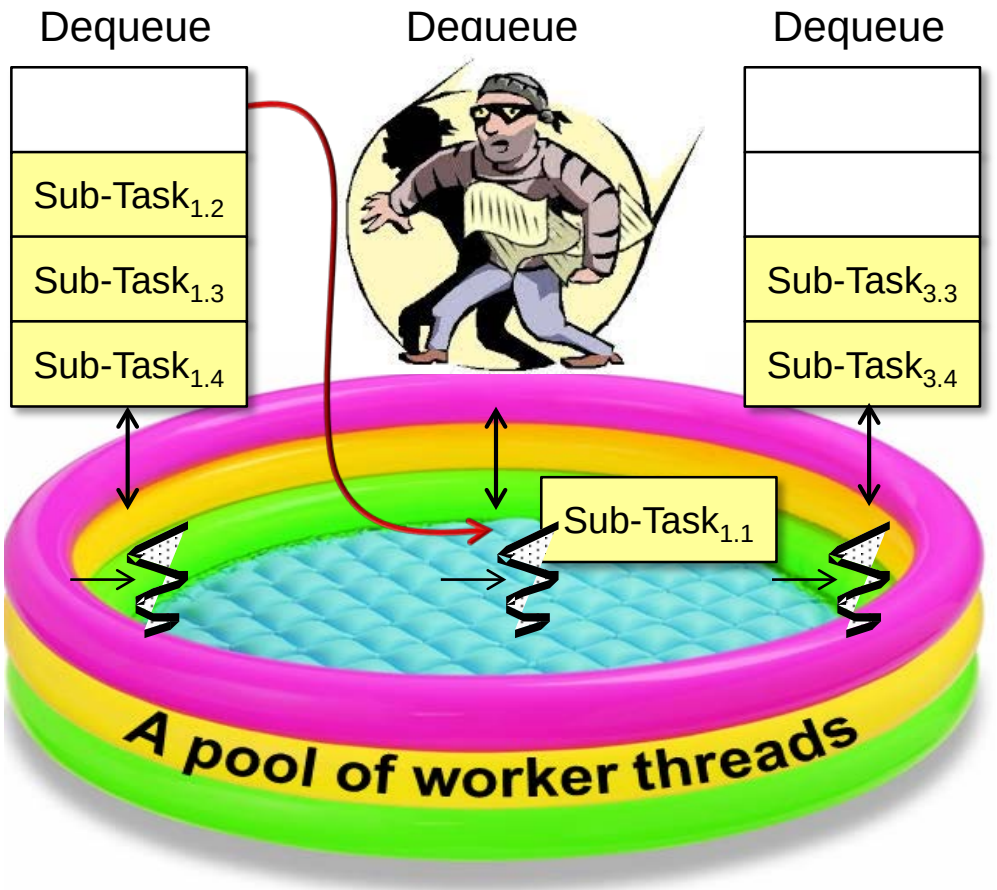
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue



"FIFO" pop/push enhances locality of reference & improves cache performance

Processing Chunks in a Parallel Stream

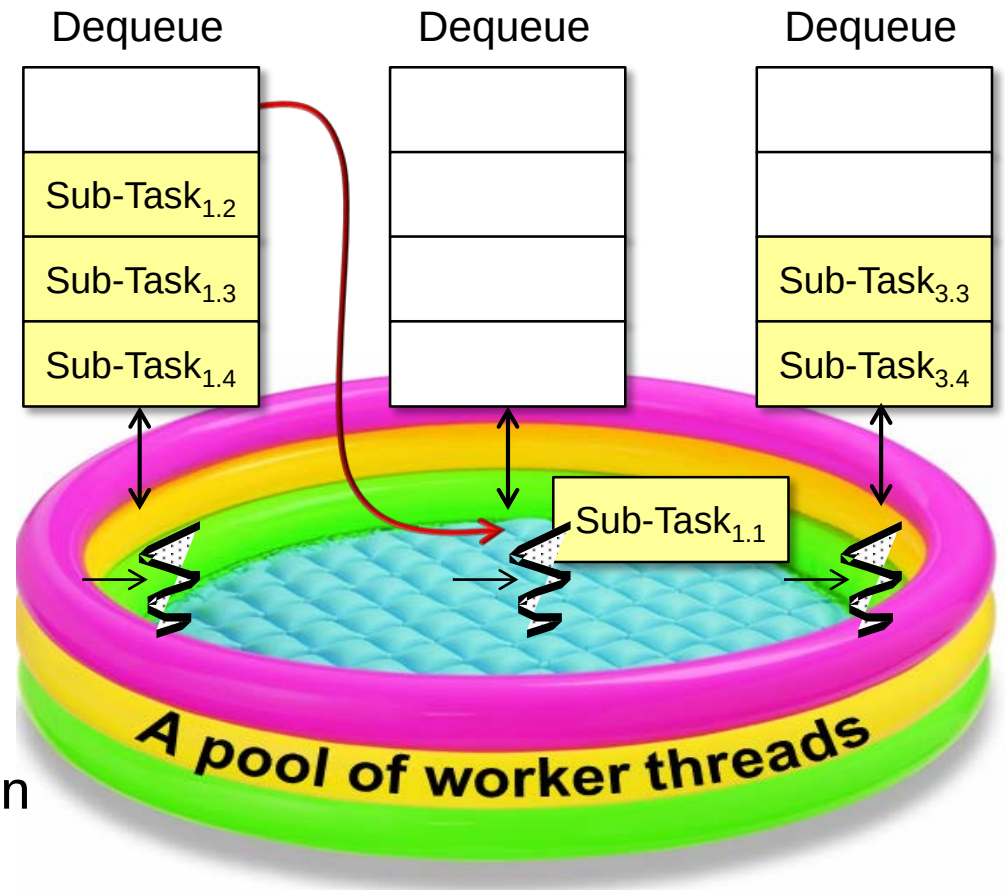
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue
  - An idle thread “steals” work from the tail of a busy thread to maximize core utilitization



See docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Processing Chunks in a Parallel Stream

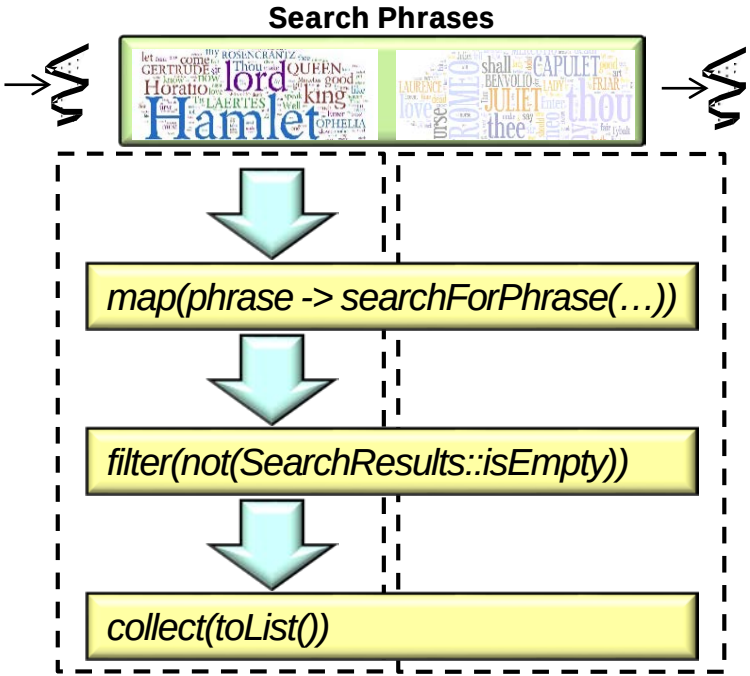
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue
- An idle thread "steals" work from the tail of a busy thread to maximize core utilitization
  - The circular dequeue used for work-stealing lowers contention

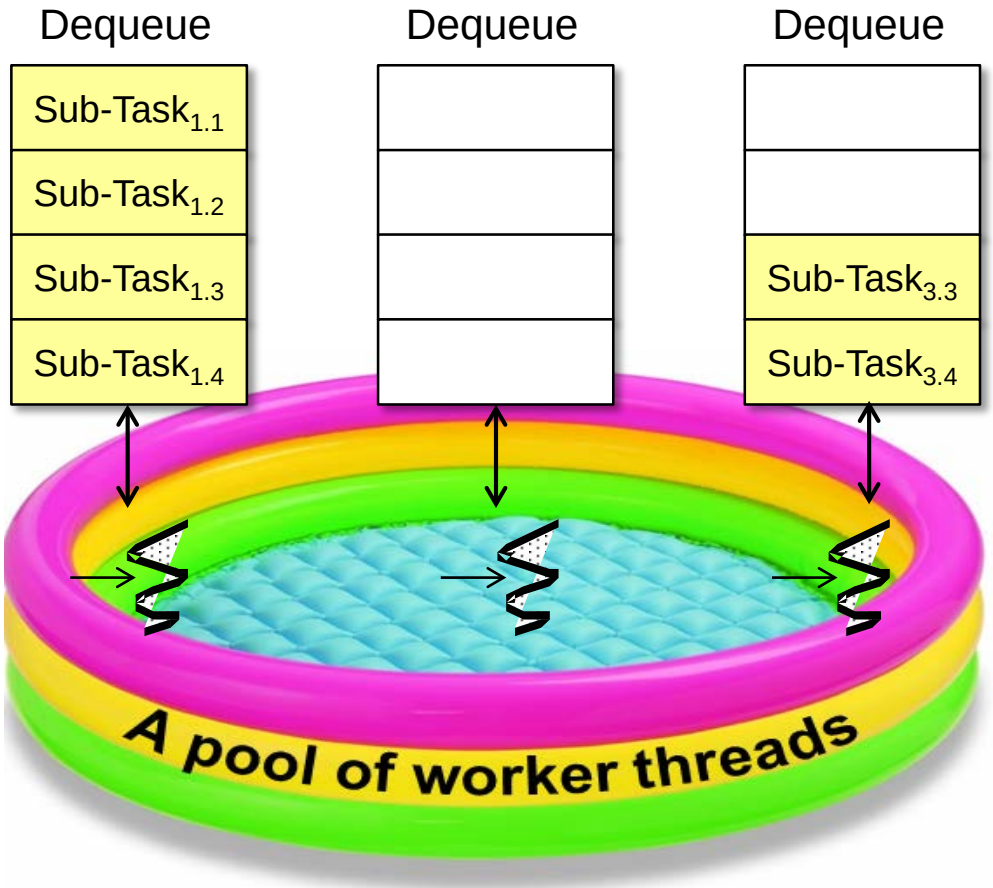


See www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade





See espressoprogrammer.com/fork-join-vs-parallel-stream-java-8

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade
  - You can program directly to the ForkJoinPool API, though it can be somewhat painful!

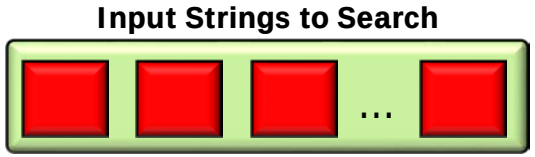
```
List<List<SearchResults>>
  listOfListOfSearchResults =
    ForkJoinPool.commonPool()
      .invoke(new
        SearchWithForkJoinTask
          (inputList,
           mPhrasesToFind, ...));
```

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade
  - You can program directly to the ForkJoinPool API, though it can be somewhat painful!

```
List<List<SearchResults>>
  listOfListOfSearchResults =
    ForkJoinPool.commonPool()
      .invoke(new
        SearchWithForkJoinTask
          (inputList,
           mPhrasesToFind, ...));
```

Use the common fork-join pool to search input strings for phrases that match



See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamForkJoin

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade
  - You can program directly to the ForkJoinPool API, though it can be somewhat painful!
    - Used for algorithms that don’t match Java 8’s parallel streams programming model

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count = count + task.join();
  return count;
}
```

See www.oracle.com/technetwork/articles/java/fork-join-422606.html

End of Java 8 Parallel Stream Internals (Part 3)