COMP 322: Fundamentals of Parallel Programming

Lecture 13: Java's ForkJoin Library

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

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

Worksheet #12 Solution: Iterative Averaging Revisited

Answer the questions in the table below for the versions of the Iterative Averaging code shown in slides 5, 7, 8, 10. Write in your answers as functions of m, n, and nc.

	Slide 5	Slide 7	Slide 8	Slide 10
How many tasks are created (excluding the main program task)?	m*n	m*nc	n	nc
How many barrier operations (calls to next per task) are performed?	0	0	m	m

The SPMD version in slide 10 is the most efficient because it only creates nc tasks. (Task creation is more expensive than a barrier operation.)

Updating all Elements in an Array

- Suppose we have a large array *a* of integers
- We wish to update each element of this array:
  - *a[i] = a[i] / (i + 1)*
- How would we write this as a parallel program using asyncs and finish?

Update Using Asyncs and Finish

```
sequentialUpdate(a, lo, hi)
    for (i = lo; i < hi; i++)
        a[i] = a[i] / (i + 1)


parallelUpdate(a, lo, hi)
    if ((hi - lo) < THRESHOLD
        sequentialUpdate(a, lo, hi)
    else
        mid = lo + hi >>> 1
        finish
            async parallelUpdate(a, lo, mid)
            async parallelUpdate(a, mid, hi)
```


Recursive Decomposition

```
solve(problem)
    if problem smaller than threshold
        solveDirectly(problem)
    else
    in parallel:
        l = solve(left-half)
        r = solve(right-half)
    combine(l, r)
```


Using Java's Fork/Join Library

- Thus far, your introduction to parallel programming has been through HJlib's API with lambda parameters
- Today, we will look at popular library for task parallelism available since Java 7 (pre-dates Java 8 lambdas)
- We can perform recursive subdivision using the Fork/Join libraries provided in Java 7 as follows:

```
public abstract class RecursiveAction extends ForkJoinTask<Void>
{
    protected abstract void compute();
    ...
}
```


Implementing Compute

```
class DivideTask extends RecursiveAction {
  static int THRESHOLD = 5; final long[] array; final int lo, hi;

  DivideTask(long[] array, int lo, int hi) {
    this.array = array; this.lo = lo; this.hi = hi;
  }
  protected void compute() {…}
}
```


Implementing Compute

```
protected void compute() {
  if (hi - lo < THRESHOLD) {
    for (int i = lo; i < hi; ++i)
      array[i] = array[i] / (i + 1);
  }
  else {
    int mid = (lo + hi) >>> 1;
    invokeAll(new DivideTask(array, lo, mid),
              new DivideTask(array, mid, hi));
  }
}
```


invokeAll

- Defined in parent class ForkJoinTask

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


static void invokeAll(ForkJoinTask<?>… tasks)

static void invokeAll(Collection<T> tasks)

…

}
```

- There are many helper methods in ForkJoinTasks; we highlight just a few
- See the Java API for more (Google is your friend)

ForkJoinTask

- Similar to a finish block enclosing a collection of asyncs
- Other Fork/Join methods in superclass ForkJoinTask<V>

```
class ForkJoinTask<V> extends Object
   implements Serializable, Future<V>
{
   ForkJoinTask<V> fork()   // asynchronously executes
   V join()                 // returns result when execution
                            // completes
   V invoke()               // forks, joins, returns result
   …
}
```


ForkJoinTasks and Futures

- ForkJoinTasks implement the Future interface
- Acts very much like HJLib futures

```
interface Future<V> {
    V get()
    V get(long timeout, TimeUnit unit)
    boolean isCancelled()
    boolean isDone()
}
```


ForkJoinTasks and Futures

- Because ForkJoinTasks are Futures, they are the values returned from fork()
- We can obtain the result of a ForkJoinTask using join() or get()
- When calling invoke or invokeAll, we never get a handle on the future explicitly
  - —Similar to finish/async blocks in HJLib

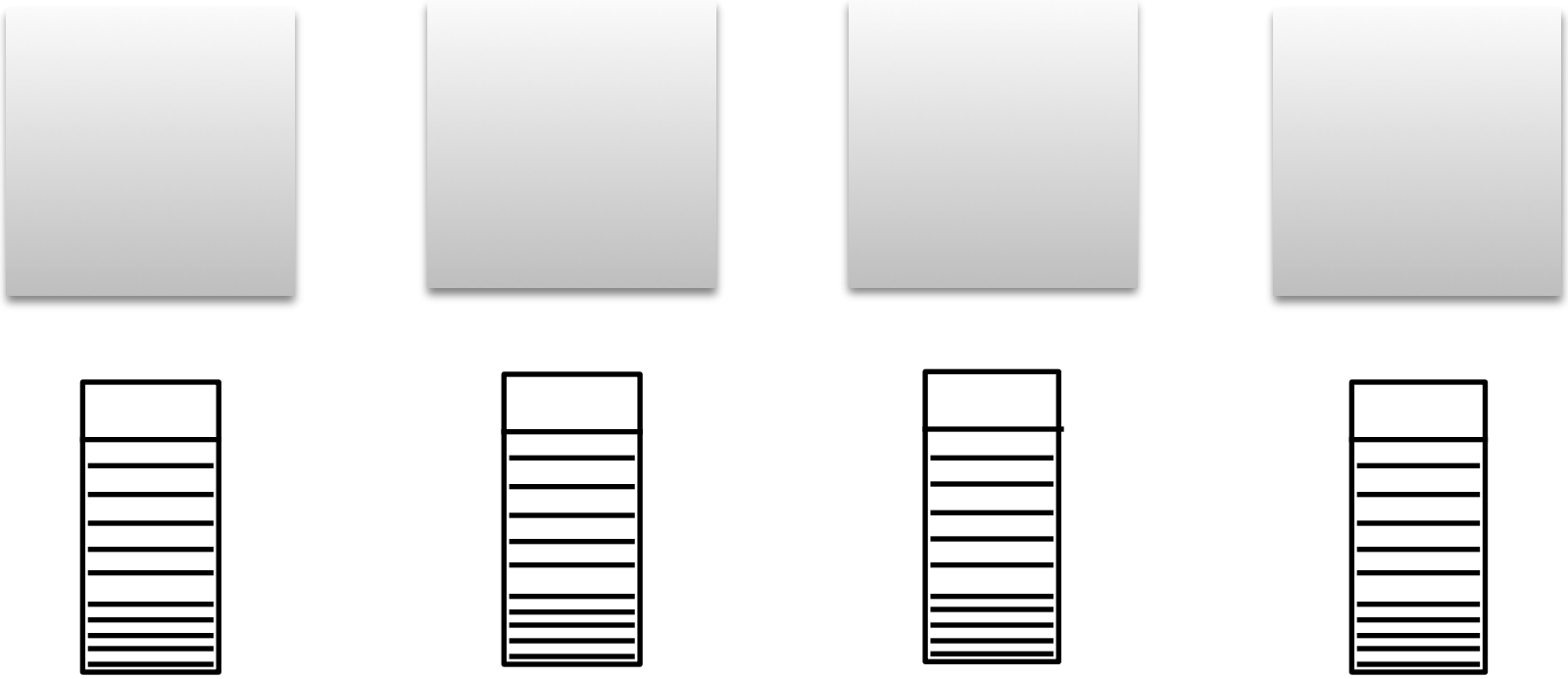
ForkJoinPools

- ForkJoinTasks are executed by the threads in a *ForkJoinPool*
- *By default, contains a number of threads equal to the number of available processors*
- You can create your own ForkJoinPools
  - —But you hardly ever need to

- class ForkJoinPool {
- static ForkJoinPool commonPool()
- …
- }
- The commonPool is used by any ForkJoinTask not explicitly submitted to a specific pool

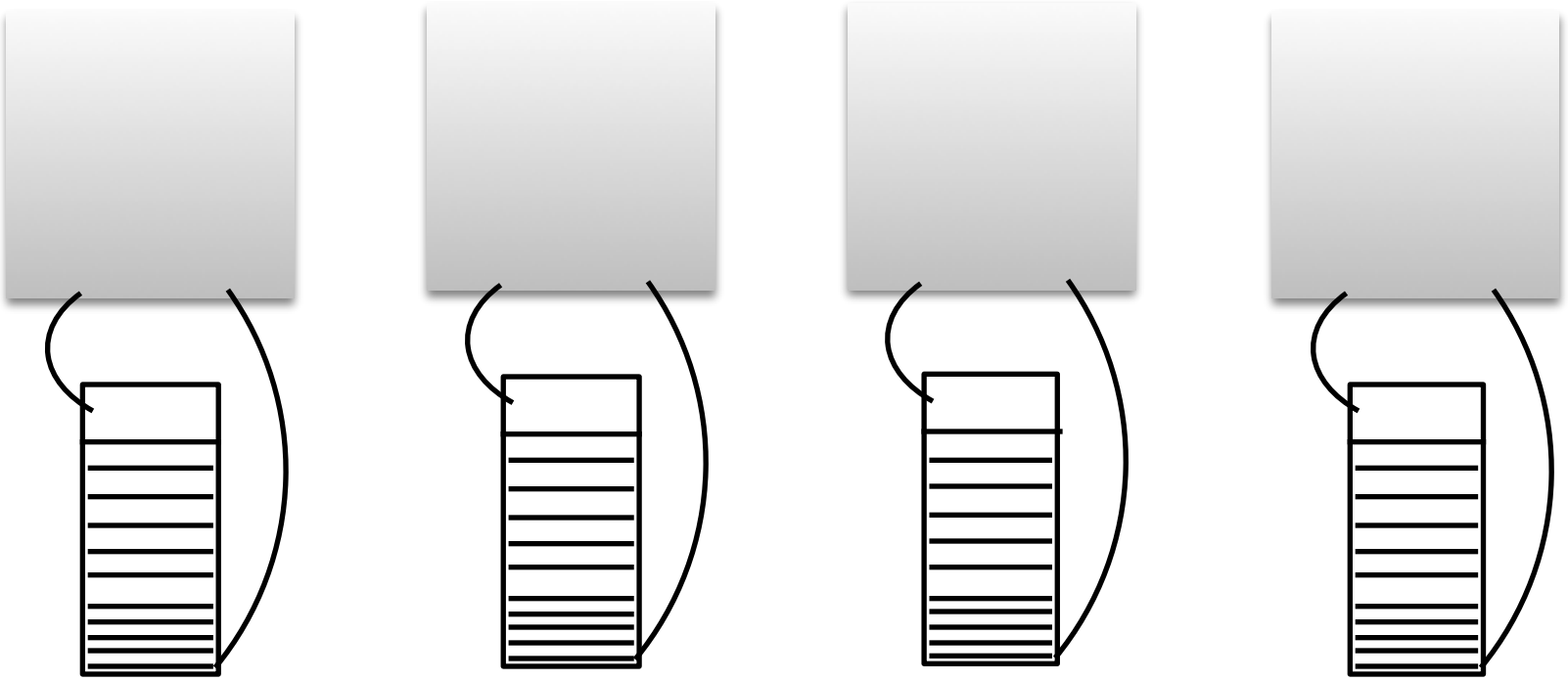
Work Stealing

- The threads in a ForkJoinPool attempt to dynamically balance the load of work among them via *"work stealing"*
- *Each worker thread keeps a double-ended queue (dequeue)*



A Worker Performs a Step of Work By:

- *Taking a ForkJoinTask t off the end of its dequeue*
- *Calling the compute method of t*
- *Inserting any recursive subtasks of t on the front of its dequeue*



If a Worker Thread's Dequeue is Empty

- *It removes a ForkJoinTask off the end of another worker thread's dequeue*
- *Requires the dequeues to be thread-safe (why?)*
- *Removing from the end assures the largest tasks are taken, minimizing interaction among threads*