Lecture 19
Fork/Join Parallelism
Divide and Conquer

• Quicksort, Mergesort are examples of *divide-and-conquer* algorithms
  – Basic structure of divide-and-conquer algorithms:
    1. If problem is small enough, solve it directly
    2. Otherwise
      a. Break problem into subproblems
      b. Solve subproblems recursively
      c. Assemble solutions of subproblems into over-all solution
  – If algorithm is tail-recursive, step 2.c. is not necessary

• Other examples
  – Depth-first search
  – Binary search
  – Euclid’s algorithm (200 B.C.!)
Parallelizing Divide-and-Conquer Algorithms

• The basic strategy: turn recursive calls into tasks
  – Solve the small instances directly
  – For larger instances requiring recursive calls, create tasks for each recursive call

• Performance tuning
  – Use a larger threshold than that specified in the original algorithm for switch to sequential solving
  – Threshold should take account of original problem size, number of CPUs
Fork/Join Parallelism

- Parallelizing divide-and-conquer algorithms is frequent enough that Java has specialized support: **Fork/Join parallelism**
- Components of Fork/Join framework
  - **Specialized executor class**: ForkJoinPool
    - Implements `ExecutorService` interface
    - Uses specialized thread-pool management, work distribution strategies tuned for divide and conquer
  - **Specialized task class**: ForkJoinTask<V>
    - Implements `Future<V>` interface
    - Has numerous specialized operations
    - Two important subclasses
      - RecursiveTask: like Callable in that value is returned
      - RecursiveAction: like Runnable in that no value is returned
- Basic idea: exploit specialized structure of divide-and-conquer dependencies to improve **parallelism** (i.e. execution time)
ForkJoinPool

• The executor for fork-join tasks
  – Maintains thread pool
  – Allocates work among worker threads

• Key attributes
  – Limits number of workers to number of CPUs (default) or user-specified number
  – Workers that are waiting for subtasks to complete are put to work on other subtasks
  – *Work-stealing* used to keep workers busy
    • Each worker has its own work queue (actually, a work *deque*)
    • When a workers deque is empty, it takes work from another workers deque
Key **ForkJoinPool** Methods

- **Constructors**
  - **ForkJoinPool()**
    Creates a ForkJoinPool with parallelism equal to `Runtime.availableProcessors()`, using the default thread factory, no `UncaughtExceptionHandler`, and non-async LIFO processing mode.
  - **ForkJoinPool(int parallelism)**
    Creates a ForkJoinPool with the indicated parallelism level, the default thread factory, no `UncaughtExceptionHandler`, and non-async LIFO processing mode.
  - **ForkJoinPool(int parallelism, ForkJoinPool.ForkJoinWorkerThreadFactory factory, Thread.UncaughtExceptionHandler handler, boolean asyncMode)**
    Creates a ForkJoinPool with the given parameters

- **<V> V invoke(ForkJoinTask<V> task)**
  Performs the given task, returning its result upon completion.
Deque? Work Stealing?

• Short for “double-ended queue” (pronounced “deck”); allows adding / removing elements from both ends
• Each worker thread has a deque containing tasks to work on (analogous to “stack” used for recursion)
  – When new tasks are created, they are “pushed” onto the front of the deque (i.e. opposite of what you do with a queue)
  – When a worker finishes a task, or blocks on the current one, it “pops” the next task from the front of its deque
• When a worker’s deque is empty it tries to steal a task from the back of one of the other workers’ deques
  – If it is successful it works on this task, using its own deque to push / pop subtasks
  – Future-like features ensure results of “stolen” tasks are available to original task owner
ForkJoinTask<V>

- Tasks that are managed by ForkJoinPool
- Besides usual Future methods (e.g. get()), other key methods are:
  - ForkJoinTask<V> fork()
    Arranges to asynchronously execute this task
  - 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, or throws an (unchecked) RuntimeException or Error if the underlying computation did so.
  - static ForkJoinPool getPool()
    Returns the pool hosting the current task execution, or null if this task is executing outside of any ForkJoinPool

- getPool()?
  - ForkJoinTasks contain internal reference to the ForkJoinPool they belong to
  - When a ForkJoinTask forks another task, the new task inherits the ForkJoinPool from the caller
More on `fork()`, `join()`

- **`fork()`** has effect of submitting task to ForkJoinPool
  - Task is placed in deque of “parent task” (i.e. one that performed `fork()`)
  - Task performing `fork()` keeps executing
- **`join()`** has effect like `get()` in `Future<V>`
  - Task performing `join()` waits until result of subtask is available
  - While it is waiting it may start work on other tasks in its deque or engage in work-stealing
  - **Note:** unlike `get()`, `join()` is not a “blocking operation” in the standard sense: no `InterruptedException` can be thrown!
  - Using `join()` also forestalls thread-starvation deadlock
    - Although number of worker threads is fixed ...
    - ... `join()` doesn’t block
Structure of a Fork/Join Application

• Define class of ForkJoinTasks
  ForkJoinTasks create subtasks, call fork, join, etc.

• Client application (i.e. one calling Fork/Join application) does this:
  – Create ForkJoinPool
  – Create task for entire problem to be solved
  – Call invoke method of ForkJoinPool with this task

• Note that ForkJoinTasks do not usually call invoke method of ForkJoinPool!
Performance Tuning of Fork/Join Applications

• ForkJoinPools automatically manage number of threads to try to maximize parallelism

• Application must manage task-creation overhead
  – Use thresholds, just like with other executor-based parallelizing approaches
  – Thresholds determine when to create new tasks vs. using sequential solutions
When To Use Fork/Join?

- Fork/Join in Java tuned for maximizing parallelism
  - Idea is to give solutions to big problems fast
  - Algorithms should be in a divide-and-conquer style
- Other performance considerations are not paramount
  - Throughput
  - Responsiveness
- So, when to use Fork/Join?
  - When problem has natural divide-and-conquer formulation
  - Parallelism is primary performance criterion
Examples

• ForkJoinSum.java
• ForkJoinSumTunable.java
• ForkJoinMergeSort.java