Lecture 19
Parallelization and Dependent Tasks
Recall Parallelization of Sequential Algorithms

• Goal of parallelization: make algorithms run faster by performing computations in parallel
• Tasks give a framework for studying parallelism
  – Tasks are (often) independent
  – They can therefore be done in parallel
  – When tasks are independent, performance tuning can be done by:
    • Restricting number of threads in thread pools
    • Relaxing task boundaries so that overhead associated with task management is kept reasonable
  – Example: Quicksort
What About Dependent Tasks?

• Traditional issue: *thread-starvation deadlock*
  – Tasks being executed in a thread pool can block waiting for their dependents
  – If more tasks block than there are threads: deadlock

• But: if you make number of threads unbounded, this may have a negative effect on performance
  – Thread creation imposes overhead itself
  – Benefit of thread limits: limit this overhead
Example: Mergesort

- Another commonly used sorting algorithm
- Works on lists (linked) rather than arrays
- Basic strategy
  - Split list into two sublists
  - Recursively sort each sublist
  - Merge sorted sublists into one sorted list
- Often used for secondary-storage sorting
  - Do not need to store entire lists in main memory
  - Do not need “random access” to elements being sorted
How To Parallelize Mergesort?

• Basic strategy for recursive algorithms
  – Make recursive calls tasks
  – When algorithm is tail-recursive, this works very well
    • No need to wait for tasks to complete after spawning them
    • Thus, tasks that creates subtasks can be allowed to terminated

• Problem with Mergesort: it is not tail-recursive
  – After recursive calls (sorting of sublists) terminate, there is more work to be done (merging)
  – This means tasks that creates subtasks must wait for results
Mergesort and Thread-Starvation

Deadlock

• Because of task dependencies in Mergesort, deadlock can happen!
  – Suppose the thread-pool contains a fixed number of worker threads
  – If number of pending sorting tasks is greater than number of workers, then workers may all be occupied by tasks awaiting completion of subtasks

• For algorithms like Mergesort, it is unsafe to use fixed-size thread pools
  – Use unbounded ones, like cached thread pool returned by `Executors.newCachedThreadPool()`
  – Then deadlock problem cannot arise
  – But what about performance? Isn’t thread overhead a problem?
Performance Tuning for Tasks with Dependencies

• Can still use same strategy for performance tuning of tail-recursive algorithms!
  – Identify threshold for problem size below which sequential algorithm is used
  – For problems larger than threshold size:
    • Break problem into sub-problems
    • Create tasks for sub-problems
    • Give tasks to executor
    • Take results of subtasks, assemble final solution (this step is not needed when algorithm is tail-recursive)

• Limits on number of worker threads in this case is enforced indirectly by problem-size threshold, rather than directly via thread-pool size limit
public class ParallelMergeSortTunable implements IntSort {
    final int NUMCPUS = Runtime.getRuntime().availableProcessors();
    private int SEQUENTIALSIZETARGET; // Integer used to determine when to switch to sequential
    private class PMSTask implements Callable<Node> {
        private Node list; // List to sort
        private final int size; // Size of list
        public Node call () {
            if (size <= SEQUENTIALSIZETARGET) return IntNodeSortUtils.mergeSortList(list); // Small list
            else { // Large list
                Node secondHalf = IntNodeSortUtils.splitList (list);
                int size1 = (size / 2) + (size % 2); // First list is half, rounded up
                int size2 = size - size1;
                Future<Node> result1 = threadPool.submit(new PMSTask(list, size1));
                Future<Node> result2 = threadPool.submit(new PMSTask(secondHalf, size2));
                try {
                    Node list1 = result1.get();
                    Node list2 = result2.get();
                    return (IntNodeSortUtils.mergeLists(list1, list2));
                } ...
            }
        }
    }
    public void sort (int[] elts) {
        int size = elts.length;
        SEQUENTIALSIZETARGET = size / (NUMCPUS+1); // Size for switching to sequential computation
        Node list = IntNodeSortUtils.copyIntoList (elts);
        Future<Node> result = threadPool.submit(new PMSTask (list, size)); ...
    }
}