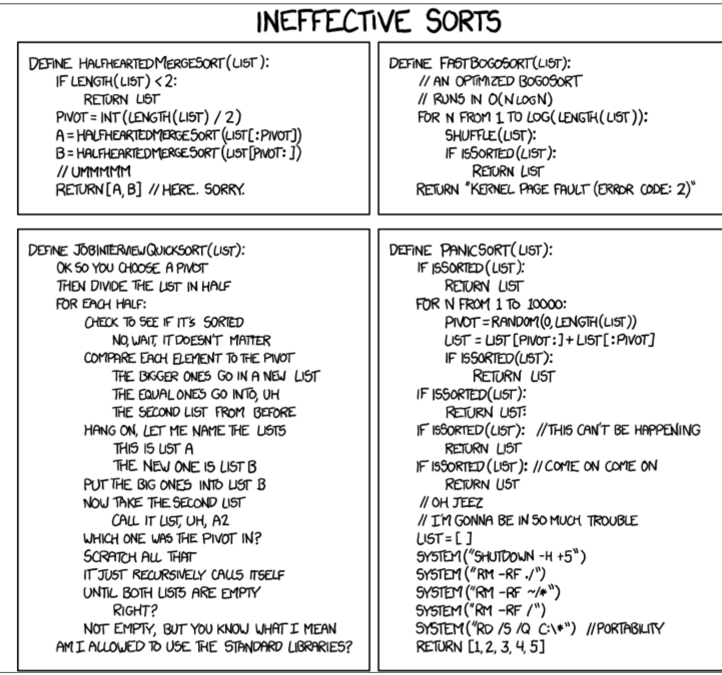
Lecture 23: More Parallel Programming

CS 62
Spring 2015
Kim Bruce & America Chambers

Some slides based on those from Dan Grossman, U. of Washington



To Use Library

- Create a ForkJoinPool
- Instead of subclass Thread, subclass RecursiveTask<V>
- Override compute, rather than run
- Return answer from compute rather than instance vble
- Call fork instead of start
- Call join that returns answer
- To optimize, call compute instead of fork (*rather than run*)

Getting Good Results

- Documentation recommends 100-50000 basic ops in each piece of program
- Library needs to warm up, like rest of java, to see good results

Data Parallel Operations

- Maps
  - apply function to all elements of data structure, producing new structure (no reductions)
  - Example:
    - ParallelVectorAdd

Maps & Reduce

- Google MapReduce is key framework in search.
  - Hadoop is open source version
- Idea: Perform maps and reduces using many computers
  - System distributes data and manages fault-tolerance
  - Programmer writes code to map one element and reduce elts for combined result.
  - Separates how to do recursive divide and conquer from actual computation to be performed.
    - Lifted from functional programming!

Analyzing Parallel Algos

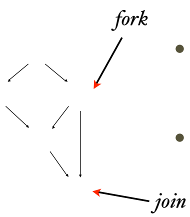
- Must be correct & efficient
  - Correctness obvious so far
- Efficiency
  - Want asymptotic bounds (big-O)
  - Analyze with any number of processors
  - ForkJoin framework guarantees get expected run-time performance asymptotically optimal for given # of processors
  - We'll assume that!

Work & Span

- Let T_P be running time if there are P processors
- Two key measures of run-time for fork-join
  - Work: How long would it take 1 processor? T_1
    - Just sequentialize all the recursive forking
  - Span: How long would it take an infinite # of processors?
    - Look for longest dependence chain
    - O(log n) for summing as no advantage with > n/2 processors
    - Called "critical path length"

Program Graph

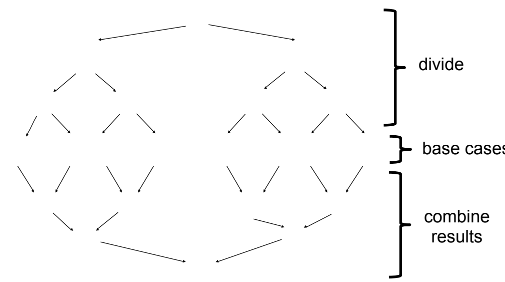
- Program using fork and join can be seen as directed acyclic graph (DAG).
  - Nodes: pieces of work
  - Edges: dependencies - source must finish before start destination



- Fork command finishes node and makes two edges out:
  - New thread & continuation of old
- Join ends node & makes new node w/ 2 edges coming in

Fork/Join: Divide & Conquer

- Basic pattern of our divide & conquer:



Often much more complex!

Performance

- Work = T_1 = sum of run-time of all nodes in DAG
  - Any “topological” sort is legal execution
- Span = T_∞ = sum of run-time of all nodes on most expensive path in DAG
  - Costs are all on nodes, not edges.
  - With unlimited processors can do everything that is ready, but still have to wait for earlier results.

Measuring Speed-Up

- Speed-up on P processors = T_1/T_P
- If speed-up on P processors is P for all P, say have *perfect* speed-up
  - Goal -- but rarely achieve except in simplest cases.
- Parallelism is max possible speed-up, T_1/T_∞
  - At some point, adding processors won’t help
  - Depends purely on span