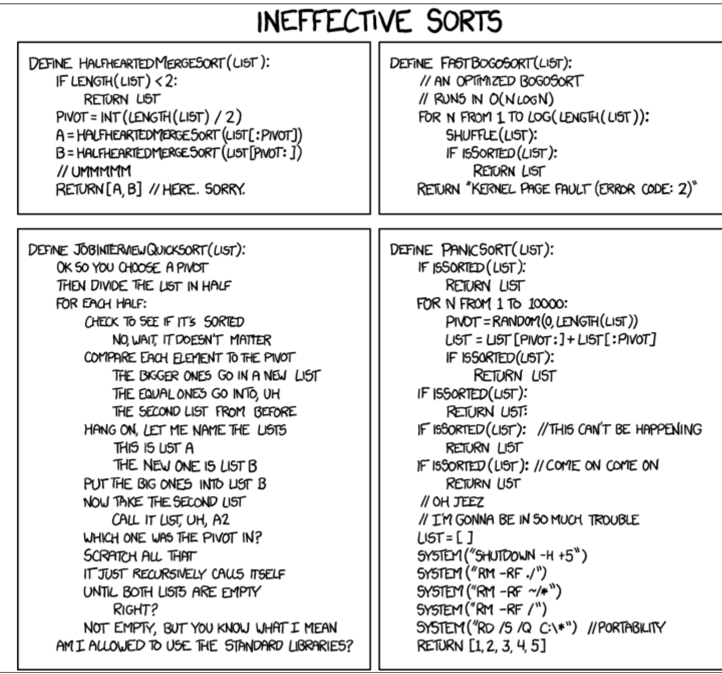
Lecture 23: Parallelism

CS 62
Fall 2016
Kim Bruce & Peter Mawhorter

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*)
- See *ForkJoinFrameworkDivideConquerPSum*

Getting Good Results

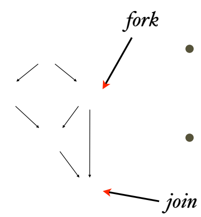
- Documentation recommends 100-50000 basic ops in each piece of program
- Library needs to warm up, like rest of java, to see good results
- Works best with more processors (> 4)

Similar Problems

- Speed up to $O(\log n)$ if divide and conquer and merge results in time $O(1)$.
- Other examples:
  - Find max, min
  - Find (leftmost) elt satisfying some property
  - Count elts satisfying some property
  - Histogram of test results
  - Called *reductions*
- Won't work if answer to 1 subproblem depends on another (e.g. one to left)

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

Performance

- Let T_P be running time if there are P processors
- Work = T_1 = sum of run-time of all nodes in DAG
- Span = T_∞ = sum of run-time of all nodes on most expensive path in DAG
- Speed-up on P processors = T_1/T_P

What does it mean?

- Guarantee: $T_P = O((T_1 / P) + T_\infty)$
  - No implementation can beat $O(T_\infty)$ by more than constant factor.
  - No implementation on P processors can beat $O((T_1 / P)$
  - So framework on average gives best can do, assuming user did best possible.
- Bottom line:
  - Focus on your algos, data structures, & cut-offs rather than # processors and scheduling.
  - Just need T_1, T_∞, and P to analyze running time

Examples

- Recall: $T_P = O((T_1 / P) + T_\infty)$
- For summing:
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect $T_p = O(n/P + \log n)$
- If instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - Then expect $T_p = O(n^2/P + n)$

Amdahl's Law

- Upper bound on speed-up!
  - Suppose the work (time to run w/one processor) is 1 unit time.
  - Let S be portion of execution that cannot be parallelized
  - $T_1 = S + (1 - S) = 1$
  - Suppose get perfect speedup on parallel portion.
    - $T_P = S + (1-S) / P$
  - Then overall speedup with P processors (Amdahl's law):
    - $T_1 / T_P = 1 / (S + (1-S) / P)$
    - Parallelism (∞ processors) is: $T_1 / T_\infty = 1 / S$

Bad News!

- $T_1 / T_\infty = 1 / S$
- If 33% of program is sequential, then millions of processors won't give speedup over 3.
- From 1980 - 2005, every 12 years gave 100x speedup
  - Now suppose clock speed is same but 256 processors instead of 1.
  - To get 100x speedup, need $100 \leq 1/(S + (1-S)/P)$
  - Solve to get solution $S \leq .0061$, so need 99.4% perfectly parallel.

Moral

- May not be able to speed up existing algos much, but might find new parallel algos.
- Can change what we compute
  - Computer graphics now much better in video games with GPU's -- not much faster, but much more detail.

A Last Example: Sorting

- Quicksort, sequential, in-place, expected time $O(n \log n)$
  - Pick pivot elt $O(1)$
  - Partition data into $O(n)$
    - A: less than pivot
    - B: pivot
    - C: greater than pivot
  - Recursively sort A, C $2*T(n/2)$
    - Now do in parallel, so $T(n/2)$
    - $n + n/2 + n/4 \ldots = 2n$, which is $O(n)$
  - With work, can improve more and get $O(\log^2 n)$

OO-Design

Because we're a bit ahead of schedule!

What are objects?

- Objects have
  - State/Properties — represented by instance variables
  - Behavior — represented by methods
    - accessor and mutator methods

Calculator

- Calculator class: User interface
  - including buttons and display
  - No real methods — construct & associate listeners
- State class: Current state of computation
  - Methods invoked by listeners
  - Communicate results to user interface
- Listener classes: Communicate from interface to state

Model-View-Controller

State

- Instance variables:
  - partialNumber, numberInProgress?, numStack, calcDisplay
- Methods:
  - addDigit(int Value)
  - doOp(char op)
  - enter, clear, pop

Model-View-Controller

- Dissociate user interface with the "model"
  - "model" represents actual computation
  - May have multiple alternate user interfaces
    - Mobile vs laptop versions of UI
- Model should be unaffected by change in UI.
- In Java UI generally served by "event thread"
  - If tie up event-thread with computation then user-interface stops being responsive.

Designing Programs

- Identify the objects to be modeled
  - E.g., Frogger game, Shell game
- List properties and behaviors of each object
  - Model properties with instance variables
  - Model behavior with methods (*write spec*)
- Refine by filling in the details
  - Hold off committing to details of representation as long as possible.

Implementation

- Write in small pieces. Test thoroughly before moving on.
- Solve simpler problem first — use "stubs" if necessary.
- Refactor as code becomes more complex.

Reading on Object-Oriented Design

- **Practical Object-Oriented Design in Ruby: An Agile Primer** by Sandi Metz, 2013
- **Design Patterns: Elements of Reusable Object-Oriented Software** by "Gang of Four", 1994

Shared Memory Concurrency

Sharing Resources

- Have been studying parallel algorithms using fork-join
  - Reduce span via parallel tasks
- Algorithms all had a very simple structure to avoid race conditions
  - Each thread had memory "only it accessed"
    - Example: array sub-range
  - On fork, "loaned" some of its memory to "forkee" and did not access that memory again until after join on the "forkee"

But ...

- Strategy won't work well when:
  - Memory accessed by threads is overlapping or unpredictable
  - Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)
- How do we control access?