Lecture 28: More Parallelism

CS 62
Fall 2017
Kim Bruce & Alexandra Papoutsaki

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

New CS Curriculum

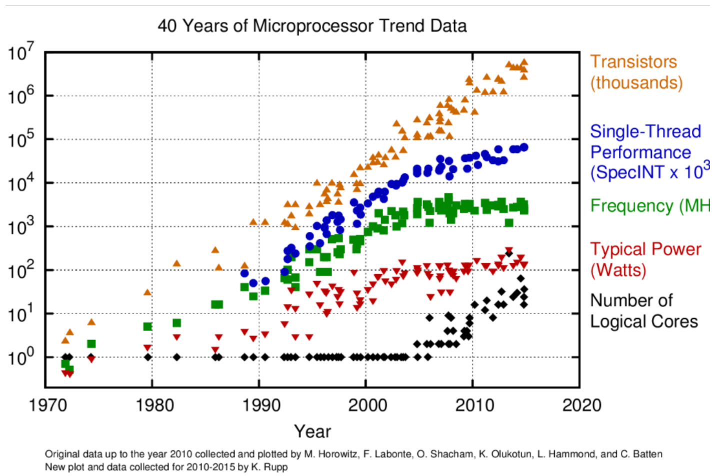
- Being phased in
  - Multiple 51s in different languages
  - CS 52 and 55 replaced by 54: Discrete Math & Functional Programming
  - CS 62 not assume Java (S'19), not teach C (now)
  - CS 105 will teach C
  - Changes later to other advanced courses
    - Won't affect you

History

- Writing correct and efficient multithread code is more difficult than for single-threaded (sequential).
- From roughly 1980-2005, desktop computers got exponentially faster at running sequential programs
  - About twice as fast every 18 months to 2 years

More History

- Nobody knows how to continue this
- Increasing clock rate generates too much heat
- Relative cost of memory access is too high
- Can keep making "wires exponentially smaller" (Moore's "Law"), so put multiple processors on the same chip ("multicore")
- Now double number of cores every 2 years!



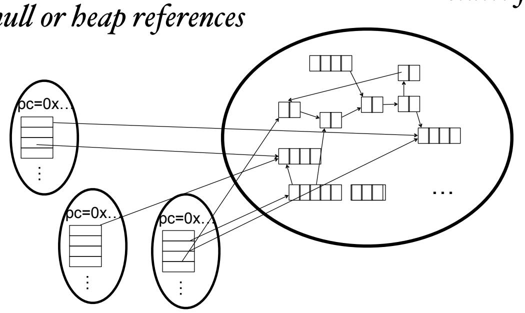
Analogy

- Typical CS1 idea:
  - Writing a program is like writing a recipe for one cook who does one thing at a time!
- Parallelism:
  - Hire helpers, hand out potatoes and knives
  - But not too many chefs or you spend all your time coordinating (*or you'll get hurt!*)

Shared Memory

Threads, each with own unshared call stack and current statement (pc for "program counter") local variables are primitives/null or heap references

Heap for all objects and static fields



Other Models

- Message-passing:
  - Each thread has its own collection of objects. Communication is via explicit messages; language has primitives for sending and receiving them.
  - Cooks working in separate kitchens, with telephones
- Dataflow:
  - Programmers write programs in terms of a DAG and a node executes after all of its predecessors in the graph
  - Cooks wait to be handed results of previous steps
- Data parallelism:
  - Have primitives for things like "apply function to every element of an array in parallel"

CPU vs GPU

From Mythbusters:

https://www.youtube.com/watch?v=-P28LKWTzrI&feature=youtu.be

In a bit more detail:

https://www.youtube.com/watch?v=1kypaBjJ-pg

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

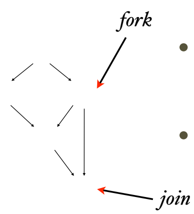
- Documentation recommends 100-50000 basic ops in each piece of program
- Library needs to warm up, like rest of java, to see good results
- Works best with more processors (> 4)

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 $100 \le 1/(S + (1-S)/P)$
  - Solve to get solution $S \le .0061$, 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.