Lecture 23: More Parallel Programming

CS 62
Spring 2011
Kim Bruce & Kevin Coogan

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 ForkJoinFrameworkDivideConquerParallelSum

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 els 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:
  - divide
  - base cases
  - combine results

Often much more complex!

Performance

- Work = $T_1 = \text{sum of run-time of all nodes in DAG}$
  - Any “topological” sort is legal execution
- Span = $T_\infty = \text{sum of run-time of all nodes on most expensive path in DAG}$
  - 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, $T_1/T_\infty$
  - At some point, adding processors won’t help
  - Depends purely on span

Division of Labor

- As user of ForkJoin, we must
  - Pick a good algorithm
  - Write a program, which creates a DAG of tasks
  - Make all nodes small and all take about same amount of work.
- Framework writer’s job:
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give expected-time guarantee $T_P = O(T_1 / P) + T_\infty$
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 averages 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 $T_1$, $T_\infty$, 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 $t$ unit time.
  - Let $S$ be portion of execution that cannot be parallelized
  - $T_1 = S + (1 - S) = t$
  - Suppose get perfect speedup on parallel portion.
    - $T_P = S + (t-S) / P$
  - Then overall speedup with P processors (Amdahl’s law):
    - $T_1/T_P = t / (S + (t-S) / P)$
    - Parallelism (= processors) is: $T_1/T_\infty = t / S$

Bad News!

- $T_1 / T_\infty = t / S$
- 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 $100 \leq 1/(S + (t-S)/P)$
  - 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 $2T(n/2)$
    - 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 $O(\log^3 n)$

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?