Parallelism Idea

- Example: Sum elements of an array
  - Use 4 threads, which each sum 1/4 of the array
- Steps:
  - Create 4 thread objects, assigning each their portion of the work
  - Call start() on each thread object to actually run it
  - Wait for threads to finish
  - Add together their 4 answers for the final result

Parallel Programming in Java

- Creating a thread:
  1. Define a class C extending Thread
     - Override public void run() method
  2. Create object of class C
  3. Call that thread's start method
     - Creates new thread and starts executing run method.
     - Direct call of run won't work, as just be a normal method call
     - Alternatively, define class implementing Runnable, create thread with it as parameter, and send start message
       - Allows class to extend a different one.

Thread Class Methods

- void start(), which calls void run()
- void join() -- blocks until receiver thread done
- Style called fork/join parallelism
  - Need try-catch around join as it can throw exception InterruptedException
- Some memory sharing: lo, hi, arr, ans fields
- Later learn how to protect using synchronized.
Actually not so great.

- If do timing, it’s slower than sequential!!
- Want code to be reusable and efficient as core count grows.
  - At minimum, make #threads a parameter.
- Want to effectively use processors available now
  - Not being used by other programs
  - Can change while your threads running

Problem

- Suppose 4 processors on computer
- Suppose have problem of size n
  - can solve w/3 processors each taking time t on n/3 elts.
- Suppose linear in size of problem.
  - Try to use 4 threads, but one processor busy playing music.
  - First 3 threads run, but 4th waits.
    - First 3 threads scheduled & take time $((n/4)/(n/3)) \cdot t = 3/4 \cdot t$
    - After 1st 3 finish, run 4th & takes another $3/4 \cdot t$
    - Total time $1.5 \cdot t$, runs 50% slower than with 3 threads!!

Other Possible Problems

- On some problems, different threads may take significantly different times to complete
- Imagine applying f to all members of an array, where f applied to some elts takes a long time
- If unlucky, all the slow elts may get assigned to same thread.
  - Certainly won't see n time speedup w/ n threads.
  - May be much worse! Load imbalance problem!

Other Possible Problems

- May not have as many processors available as threads
- On some problems, different threads may take significantly different times to complete
**Toward a Solution**

- To avoid having to wait too long for any one thread, instead create lots of threads.
- Schedule threads as processors become available.
- If one thread very slow, many others will get scheduled on other processors while that one runs.
- Will work well if slow thread scheduled relatively early.

**Naive Algorithm Not Work**

- Suppose divide up work into threads which each handle 100 elts.
- Then will be n/100 threads.
  - Adding them up linear in size of array
  - If each thread handles only 1 sum then back to sequential algorithm.

**Divide & Conquer**

- Divide in half, w/ one thread per half.
  - Each half further subdivided w/ new threads, etc.
  - Depth is O(log n), which is optimal
  - If have numProc processors then total time \( O(n/\text{numProc} + \log n) \)

**In practice**

- Creating all threads and communication swamps savings so
  - use sequential cutoff of about 500
  - Don't create two recursive threads
    - one new and reuse old.
    - Cuts number of threads in half.
Even Better

- Java threads too heavyweight — space and time overhead.
- ForkJoin Framework solves problems
- Standard as of Java 7.
  - We'll use additions as of Java 8

To Use Library

- Create a ForkJoinPool via
  - `fjPool = ForkJoinPool.commonPool()`
- 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
- Start by writing `fjPool.invoke(t)` where `t` is initial thread
- To optimize, call compute instead of fork (rather than run)
- See `ForkJoinFrameworkDivideConquerPSum`

Considerations

- Entire program should have one ForkJoinPool.
  - Might as well make it static — use commonPool() method
- Start up everything with `fjPool.invoke(new ...)`
  - Once you are inside, use fork or compute.
- Use:
  - `RecursiveTask<T>` when return a value of type `T`
  - `RecursiveAction` when there is nothing to return

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 els 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 = $T_\infty$ = 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 $T_1$, $T_\infty$, and $P$ to analyze running time