Parallelism & Concurrency

- Single-processor computers going gone away.
  - Hit a wall in terms of speed!
- Want to use separate processors to speed up computing by using them in parallel.
- Also have programs on single processor running in multiple threads. Want to control them so that program is responsive to user: Concurrency
- Often need concurrent access to data structures (e.g., event queue). Need to ensure don't interfere w/each other.

What can you do with multiple cores?

- Run multiple totally different programs at the same time
  - Already do that? Yes, but with time-slicing
- Do multiple things at once in one program
  - Our focus – more difficult
  - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

Models Change

- Model: Shared memory w/explicit threads
- Program on single processor:
  - One call stack:
    - each stack frame holds local variables and references to parameters
  - One program counter (current statement executing)
  - Static fields
  - Objects (created by new) in the heap (nothing to do with heap data structure)
Multiple Threads/Processors

- New story:
  - A set of threads, each with its own call stack & program counter
  - No access to another thread's local variables
  - Threads can (implicitly) share static fields / objects
  - To communicate, write somewhere another thread reads

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

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
     - Same kind of issue as paint-repaint!
     - Alternatively, define class implementing Runnable, create thread w/it as parameter, and send start message
       - Allows class to extend a different one.
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

First Attempt

class SumThread extends Thread {
  int lo, int hi, int[] arr; // fields to know what to do
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }
  }

int sum(int[] arr){
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++)// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
  ts[i].start(); // use start not run
  }
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}

Correct Version

class SumThread extends Thread {
  int lo, int hi, int[] arr; // fields to know what to do
  int ans = 0; // for communicating result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }
  }

int sum(int[] arr){
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++)// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
  ts[i].start(); // start not run
  }
  for(int i=0; i < 4; i++) // combine results
    ts[i].join(); // wait for helpers to finish!
    ans += ts[i].ans;
  return ans;
  }

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: array is shared
- 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 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 1 thread very slow, many others will get scheduled on other processors while that one runs.
- Will work well if slow thread scheduled relatively early.

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 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.
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(t).
- 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)