Lecture 22: More Parallel Programming

CS 62
Spring 2011
Kim Bruce & Kevin Coogan

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

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 start0 on each thread object to actually run it
  - Wait for threads to finish
  - Add together their 4 answers for the final result

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 helper to finish!
    ans += ts[i].ans;
    return ans;
}

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

ParallelSum project.
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.

Naive Algorithm Not Work

- Suppose divide up work into threads which each handle 100 elts.
- Then will be \( \frac{n}{100} \) 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 \( O(n/\text{numProc} + \log n) \)
- Each layer is \( O(1) \) in parallel
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
- In Java 7, but can also use early release in Java 6 with jsr166.jar.

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
- 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)