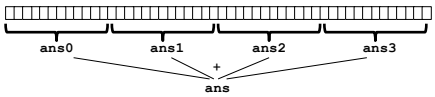
# Lecture 37: Java Parallelism & Concurrency

CSC 131
Fall, 2011

Kim Bruce

# Parallelism

# 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

# 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(){ … // add from a[l] to a[h]}
}

static 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;
}
```

*See program ParallelSum*

# Thread Class Methods

- void start( ), which calls void run( )
- void join( ) -- blocks until receiver thread done
- Style called fork/join parallelism
  - Code on previous slide generates error message as join can throw exception InterruptedException
- Some memory sharing: arr field
- Later learn how to protect using synchronized.

# Actually not so great.

- If do timing, it's slower w/ small arrays 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)*t = 3/4 t
    - After 1st 3 finish, run 4th & takes another 3/4 t
    - Total time 1.5 * t , runs 50% slower than with 3 threads!!!

# Other Possible Problems

- On some problems, different threads may take significantly different times to complete
- Apply f to all members of an array, where f applied to some elts takes a long time
- All 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!

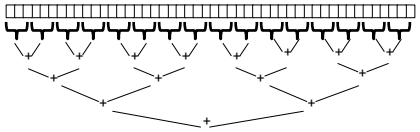
# Toward a Solution

- 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 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. until down to single elements
  - Depth is O(log n), which is optimal
  - Then total time w/numProc processors

    O(n/numProc + log n)

*straight-line code cost in step 1*

*each layer is O(1) in parallel*

# In practice

- Creating all threads and communication swamps savings so
  - use sequential cutoff about 1000
  - Don't create two recursive threads
    - one new and reuse old.
    - Cuts number of threads in half.

*EfficentDivideConquerParallelSum*

# Even Better

- Java threads too heavyweight -- space and time overhead.
- ForkJoin Framework solves problems
- Will be in Java 7, but early release in 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*

# Handling Concurrency in Java

*See ATM example*