Lecture 24: More Parallel Programming

Today

- Reading
  - P&C Sections 4 and 5
- Objectives
  - Finish Divide and Conquer Parallelism
  - Work and span
  - Amdahl's Law

Announcements

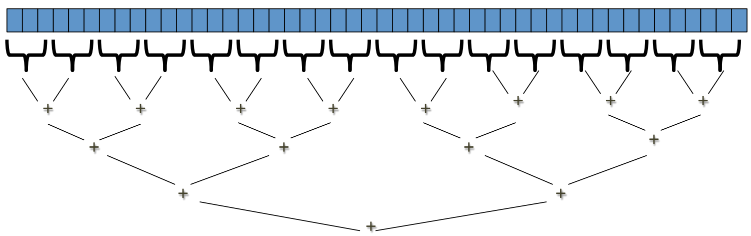
- Start HW assignment and come on Wednesday ready to discuss `GameTree` class
- No quiz this Friday (Cesar Chavez Day)
- One-on-one tutoring through the QSC
  - The tutor is Sarah!
  - Make an appointment
- Review midterms in lab on Wednesday

Recap: Divide and Conquer Parallelism

- Running example: summing an array of integers
- Problems encountered
  - Don't want to hard code the number of threads
  - Use all/only the processors available to us now
  - Load imbalance
- Solution
  - Use lots of threads! Much more than the number of processors
  - Each thread does a little bit of work

Final Attempt

- Divide-and-conquer Parallelism!
  - Change our algorithm
  - Use parallelism for the recursive calls



Divide and Conquer Parallelism

```
class SumThread extends java.lang.Thread {
  int lo; int hi; int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ // override
    if(hi – lo < SEQUENTIAL_CUTOFF)
      for(int i=lo; i < hi; i++)
        ans += arr[i];
    else {
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.start();
      left.join(); // don't move this up a line – why?
      right.join();
      ans = left.ans + right.ans;
    }
  }
}
```

Divide and Conquer Parallelism

- Divide array in half with one thread per half
  - There is a distinction between **work** and **time** when we work in parallel
  - ~2N threads each doing O(1) work results in O(N) work
- How much time does it take P processors to do O(N) work?

Divide and Conquer Parallelism

- How much time does it take P processors to do O(N) work?
  - If we have O(N) processors, run time is O(logN) because each level is done in parallel
- If we have P processors, takes O(N/P + logN) time

Divide and Conquer Parallelism

- Final improvements
  - Choose cutoff value: below cutoff switch to sequential programming
  - Don't create two threads: create one thread and have the calling thread do the other half of the work

Divide and Conquer Parallelism

```
class SumThread extends java.lang.Thread {
  static int SEQUENTIAL_CUTOFF = 1000;
  int lo; int hi; int[] arr;  // arguments
  int ans = 0;                // result

  SumThread(int[] a, int l, int h) { … }

  public void run(){ // override
    if(hi – lo < SEQUENTIAL_CUTOFF)
      for(int i=lo; i < hi; i++)
        ans += arr[i];
    else {
      SumThread left = new SumThread(arr,lo,(hi+lo)/2);
      SumThread right= new SumThread(arr,(hi+lo)/2,hi);
      left.start();
      right.run();  // call run instead of start!
      left.join();  // don't move this up a line – why?
      ans = left.ans + right.ans;
    }
  }
}
```

Java ForkJoin Framework

- In the end, Java threads are still to heavyweight!
- Use `java.util.concurrent` package available in Java 7 standard libraries
- To use create a `ForkJoinPool`
- ForkJoin documentation recommends 500-50000 basic operations per thread for optimal performance guarantees

Java ForkJoin Framework

```
class SumArray extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // arguments
  SumArray(int[] a, int l, int h) { … }
  protected Integer compute(){// return answer
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      int ans = 0;
      for(int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;
    } else {
      SumArray left = new SumArray(arr,lo,(hi+lo)/2);
      SumArray right= new SumArray(arr,(hi+lo)/2,hi);
      left.fork();
      int rightAns = right.compute();
      int leftAns  = left.join();
      return leftAns + rightAns;
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr){
  return fjPool.invoke(new SumArray(arr,0,arr.length));
}
```

Different terms, same basic idea

Don't subclass `Thread`	Do subclass `RecursiveTask<V>`
Don't override `run`	Do override `compute`
Do not use an `ans` field	Do return a `V` from `compute`
Don't call `start`	Do call `fork`
Don't just call `join`	Do call `join` which returns answer
Don't call `run` to hand-optimize	Do call `compute` to hand-optimize
Don't have a topmost call to `run`	Do create a pool and call `invoke`

See the handouts page for a link to:

"A Beginner's Introduction to the ForkJoin Framework"

13

Reductions

- Computations of this form are called reductions
- Reduce collection to a single answer via an associative operator
  - Examples: max, count, leftmost, rightmost, sum, product, …
  - Non-examples: median, subtraction, exponentiation

Maps (Data Parallelism)

- A map operates on each element of a collection independently to create a new collection of the same size
  - No combining results
- Exercise: how could you code up vector addition using ForkJoin Framework?

```
int[] vector_add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  result = new int[arr1.length];
  FORALL(i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

Maps and Reductions

- Maps and reductions are the "workhorses" of parallel programming
  - Learn to recognize when an algorithm can be written in terms of maps and reductions
  - Programming them becomes "trivial" with a little practice
    - Exactly like sequential for-loops seem second-nature
  - Google's MapReduce framweork

Analyzing ForkJoin Algorithms

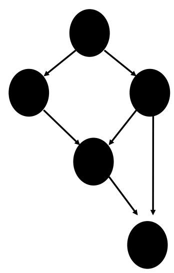
- Focus on efficiency (instead of correctness)
  - Want asymptotic bounds
  - Analyze the algorithm for any number of processors
  - ForkJoin Framework guarantees expected run-time performance is asymptotically optimal for given number of processors
    - So we can analyze algorithms assuming this guarantee

Work and Span

- Let T_P be the running time if there are P processors available
- Two key measures of run-time for fork-join parallelism:
- Work (T_1): How long it takes to run on 1 processor
  - "Sequentialize" the recursive forking algorithm
- Span (T_∞) : How long it would take with infinite processors
  - The longest dependence-chain
  - Example: $O(\texttt{log}\ n)$ for summing an array
    - Notice having > $n/2$ processors is no additional help

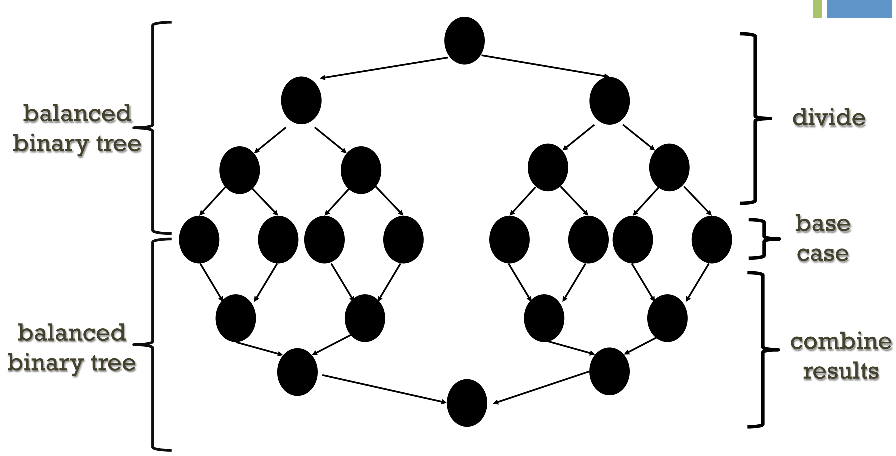
Program Execution as a DAG

- A program execution using `fork` and `join` can be viewed as a DAG
  - Nodes: Pieces of O(1) work
  - Edges: Source must finish before destination starts



- A fork "ends a node" and makes two outgoing edges
- A join "ends a node" and makes a node with two incoming edges

Summing an array of integers



Work and Span in terms of DAG

- Recall: T_P = running time if there are P processors available
- Work (T_1): How long it takes to run on 1 processor
  - Corresponds to the number of nodes in the DAG
  - O(N) for simple maps and reductions
- Span (T_∞) : How long it would take with infinite processors
  - Length of the longest path in the DAG
  - Infinite processors must still wait for earlier results to be done
  - $O(\log N)$ for simple maps and reductions

Speed-Up

- Speed-up on P processors is defined as T_1 / T_P
- If speed-up is P as we vary P, we call it perfect linear speed-up
  - Perfect linear speed-up means doubling P halves running time
  - Usually our goal – hard to get in practice
- Parallelism is the maximum possible speed-up: T_1 / T_∞
  - At some point, adding processors won't help
  - Where that point is depends on the span

ForkJoin provides optimal T_P

- ForkJoin guarantees $T_P = O((T_1 / P) + T_\infty)$
  - No implementation can be better than $O(T_\infty)$
  - No implementation with P processors can be better than $O(T_1/P)$
  - First term dominates for small P, second for large P
- The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!
  - Guarantee requires a few assumptions about your code…

Division of responsibility

- Our job as ForkJoin Framework users:
  - Pick a good algorithm, write a program
  - *All threads do approximately equal amount of work*
  - *All threads do small but not tiny amount of work*
- The framework-writer's job:
  - Assign work to available processors to avoid idling
    - Let framework-user ignore all scheduling issues
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did his/her job

$$T_P = O((T_1 / P) + T_\infty)$$

Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect (ignoring overheads): $T_P = O(n/P + \log n)$
- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads): $T_P = O(n^2/P + n)$

Amdahl's Law (derive on board)

- Provides upper-bound on speedup given that only part of algorithm can be parallelized
- Amdahl's Law states: $$\frac{T_1}{T_P} = \frac{1}{S + \frac{1-S}{P}}$$
- where S is the proportion of the algorithm that cannot be parallelized
- Corollary of Amdahl's Law: $$\frac{T_1}{T_\infty} = \frac{1}{S}$$

Amdahl's Law is Bad News!

- Suppose 33% of a program's execution is sequential
  - Then a billion processors won't give a speedup over 3x
- From 1980-2005, every 12 years gave 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - To get 100x speedup, we need

$$100 \leq 1 / (S + (1\text{-}S)/256)$$

Which means $S \leq .0061$ (i.e., 99.4% perfectly parallelizable)

Take home message

Amdahl's Law is a bummer!
  - Unparallelized parts become a bottleneck very quickly
  - But it doesn't mean additional processors are worthless

- We can find new parallel algorithms
  - Some things that seem sequential are actually parallelizable
- We can change the problem or do new things
  - Example: Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering more beautiful(?) monsters