More Parallelism

CSE 373
Data Structures & Algorithms
Ruth Anderson

Today's Outline

- **Admin**:
  - HW #6 – Sorting! due Thurs Dec 8 at 11pm
- **More Parallelism**
  - Reduces and Maps
  - Parallel runtimes
  - Limits of parallelism: Ahmdal's Law
  - Parallel Sorting

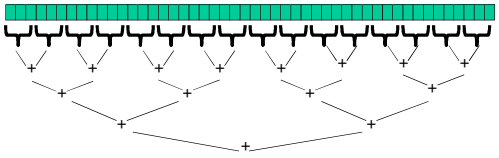
Parallelism Outline

Done:
- Intro to Parallelism vs. Concurrency
- Java threads
- How to use **`fork`** and **`join`** to write a parallel algorithm in Java
- Why using divide-and-conquer with lots of small tasks is best
  - Combines results in parallel

Now:
- More examples of simple parallel programs (map & reduce)
- Asymptotic analysis for fork-join parallelism
- Amdahl's Law
- Parallel Sorting

We looked at summing an array

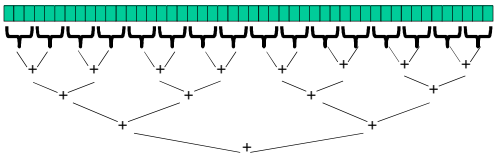
- Summing an array went from $O(n)$ sequential to $O(\log n)$ parallel (*assuming **a lot** of processors and very large n*)
  - An exponential speed-up in theory



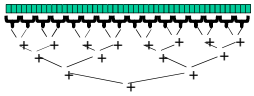
- Anything that can use results from two halves and merge them in $O(1)$ time has the same property…

Extending Parallel Sum

- We can tweak the 'parallel sum' algorithm to do all kinds of things; just specify 2 parts (usually)
  - Describe how to compute the result for base problem size (Sum: Iterate through values sequentially and add them up)
  - Describe how to merge results (Sum: Just add 'left' and 'right' results)



Examples



Parallelization (for some algorithms)
- Describe how to compute result at the 'cut-off'
- Describe how to merge results

How would we do the following (assuming data is given as an array)?
1. Maximum or minimum element
2. Is there an element satisfying some property (e.g., is there a 17)?
3. Left-most element satisfying some property (e.g., first 17)
4. Counts; for example, number of strings that start with a vowel
5. Are these elements in sorted order?

Reductions

- This class of computations are called reductions
  - We 'reduce' a large array of data to a single item
- Note: Recursive results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: create a Histogram of test results from a much larger array of actual test results
- While many can be parallelized due to nice properties like associativity of addition, some things are inherently sequential
  - How we process **arr[i]** may depend entirely on the result of processing **arr[i-1]**

7

Even easier: Data Parallel (Maps)

- While reductions are a simple pattern of parallel programming, maps are even simpler
  - Operate on set of elements to produce a new set of elements (no combining results); generally input and output are of the same length
  - All operations can be done in parallel
  - E.g. Multiply each element of an array by 2.
- Example:

```
int[] mult_by_two(int[] arr){
  result = new int[arr.length];
  FORALL(i=0; i < arr.length; i++) {
    result[i] = arr[i] * 2;
  }
  return result;
}
```

8

Another Map Example:

Maps operate on a set of elements to produce a new set of elements (no combining results); generally input and output are of the same length

– All operations can be done in parallel

Another Example: Vector addition

```
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;
}
```

9

Vector Add Map in ForkJoin Framework

```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
      for(int i=lo; i < hi; i++)
        res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

10

(*Java Details*) Map vs reduce in ForkJoin framework

- In our examples:
- Reduce:
  - Parallel-sum extended RecursiveTask
  - Result was returned from compute()
- Map:
  - Class extended was RecursiveAction
  - Nothing returned from compute()
  - In the above code, the 'answer' array was passed in as a parameter
- Doesn't *have* to be this way
  - Map can use RecursiveTask to, say, return an array
  - Reduce could use RecursiveAction; depending on what you're passing back via RecursiveTask, could store it as a class variable and access it via 'left' or 'right' when done

11

Digression on maps and reduces

- You may have heard of Google's "map/reduce"
  - Or the open-source version Hadoop
- Idea: Want to run algorithm on enormous amount of data; say, sort a petabyte (10^6 gigabytes) of data
  - Perform maps and reduces on data using many machines
    - The system takes care of distributing the data and managing fault tolerance
    - You just write code to map one element and reduce elements to a combined result
  - Separates "how to do recursive divide-and-conquer" from *what* computation to perform

12

Analyzing algorithms

- Like all algorithms, parallel algorithms should be:
  - Correct
  - Efficient
- For our algorithms so far, correctness is "obvious" so we'll focus on efficiency:
  - We still want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors

13

Work and Span

Let $\mathbf{T_P}$ be the running time if there are **P** processors available
Type/power of processors doesn't matter; $\mathbf{T_P}$ used asymptotically, and to compare improvement by adding a few processors

Two key measures of run-time for a fork-join computation:

- Work: How long it would take 1 processor = $\mathbf{T_1}$
  - Just "sequentialize" all the recursive forking
- Span: How long it would take infinity processors = $\mathbf{T_\infty}$
  - The hypothetical ideal for parallelization
  - This is the longest "dependence chain" in the computation
  - e.g. depth of the recursive calls in parallel sum

14

Speed-up

- **Speed-up** on **P** processors: $\mathbf{T_1 / T_P}$
- 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
- E.g. If $T_1 = 100$, and we have P = 4 processors, then perfect speed-up would be: $T_4 = 25$

Speed-up = T_1 / T_4 = 100 / 25 = 4x speedup

15

Amdahl's Law (mostly bad news)

- So far: talked about a parallel program in terms of work and span
- In practice, it's common that your program has:

a) parts that **parallelize well:**
  - Such as maps/reduces over arrays

b) …and parts that **don't parallelize at all:**
  - Such as reading a linked list, getting input, or just doing computations where each step needs the results of previous step

- These **unparallelized** parts can turn out to be a big bottleneck

16

Amdahl's Law (mostly bad news)

Let the ***work*** (time to run on 1 processor) be 1 unit time.

Let **S** be the portion of the execution that can't be parallelized (i.e. must be run sequentially)

Then: $\mathbf{T_1 = S + (1-S) = 1}$

Suppose we get perfect linear speedup *on the parallel portion*

Then: $\mathbf{T_P = S + (1-S)/P}$

So the overall speedup with **P** processors is (Amdahl's Law):

$\mathbf{T_1 / T_P = 1 / (S + (1-S)/P)}$

17

Amdahl's Law Example

Suppose: $\mathbf{T_1 = S + (1-S) = 1}$ (aka total program execution time)
$\mathbf{T_1 = 1/3 + 2/3 = 1}$
$\mathbf{T_1 = 33\ sec + 67\ sec = 100\ sec}$

Time on P processors: $\mathbf{T_P = S + (1-S)/P}$

So: $\mathbf{T_P = 33\ sec + (67\ sec)/P}$
$\mathbf{T_3 = 33\ sec + (67\ sec)/3 =}$

18

Why such bad news?

$T_1 / T_P = 1 / (S + (1-S)/P)$ $T_1 / T_\infty = 1 / S$

- Suppose 33% of a program is sequential
  - Then a billion processors won't give a speedup over 3!!!
- No matter how many processors you use, your speedup is bounded by the sequential portion of the program.

19

The future and Amdahl's Law

Speedup: $T_1 / T_P = 1 / (S + (1-S)/P)$
Max Parallelism: $T_1 / T_\infty = 1 / S$

- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - What portion of the program must be parallelizable to get 100x speedup?

20

The future and Amdahl's Law

Speedup: $T_1 / T_P = 1 / (S + (1-S)/P)$
Max Parallelism: $T_1 / T_\infty = 1 / S$

- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - What portion of the program must be parallelizable to get 100x speedup?

For 256 processors to get at least 100x speedup, we need
$100 \leq 1 / (S + (1-S)/256)$
Which means $S \leq .0061$ (i.e., 99.4% must be parallelizable)

21

Plots you have to see

1. Assume 256 processors
   - x-axis: sequential portion **S**, ranging from .01 to .25
   - y-axis: speedup T_1 / T_P (will go down as **S** increases)
2. Assume **S** = .01 or .1 or .25 (three separate lines)
   - x-axis: number of processors **P**, ranging from 2 to 32
   - y-axis: speedup T_1 / T_P (will go up as **P** increases)

I encourage you to try this out!
- Chance to use a spreadsheet or other graphing program
- Compare against your intuition
- A picture is worth 1000 words, especially if you made it

22

All is not lost

Amdahl's Law is a bummer!
- But it doesn't mean additional processors are worthless

- We can find new parallel algorithms
  - Some things that seem entirely sequential turn out to be parallelizable
  - Eg. How can we parallelize the following?
    - Take an array of numbers, return the 'running sum' array:

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

  - At a glance, not sure; we can!
- We can also change the problem we're solving or do new things
  - Example: Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering richer environments and more beautiful (terrible?) monsters

23

Moore and Amdahl


- Moore's "Law" is an observation about the progress of the semiconductor industry
  - Transistor density doubles roughly every 18 months
- Amdahl's Law is a mathematical theorem
  - Implies diminishing returns of adding more processors
- Both are incredibly important in designing computer systems

24

Sequential Quicksort review

Recall quicksort was sequential, in-place, expected time $O(n \log n)$

	Best / expected case *work*
1. Pick a pivot element	**O(1)**
2. Partition all the data into:	**O(n)**
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	**2T(n/2)**

Recurrence (assuming a good pivot):
T(0)=T(1)=1
T(n)=n + 2T(n/2) = O(nlogn)
Run-time: O(nlogn)

How should we parallelize this?

25

Review: Common recurrences

$T(n) = O(1) + T(n-1)$	linear (e.g. recursive linear search)
$T(n) = O(1) + T(n/2)$	logarithmic (e.g. binary search)
$T(n) = O(n) + T(n-1)$	quadratic (e.g. Quicksort worst case)
$T(n) = O(n) + T(n/2)$	linear (see next few slides)
$T(n) = O(n) + 2T(n/2)$	$O(n \log n)$ (e.g. Mergesort)

26

Parallel Quicksort

	Best / expected case *work*
1. Pick a pivot element	**O(1)**
2. Partition all the data into:	**O(n)**
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	**2T(n/2)**

Do the two recursive calls in parallel!

- **Work**: unchanged of course, O(n log n)
- **Span :** Now recurrence takes the form:
  T(n) = O(n) + 1T(n/2) = **O(n)**
- (In fact it is possible to do even better is we parallelize the partition – we won't do that here.)

27

Now Mergesort!

Recall mergesort: sequential, not-in-place, worst-case $O(n \log n)$

1. Sort left half and right half	**2T(n/2)**
2. Merge results	**O(n)**

Similar to quicksort: do the two recursive sorts in parallel!

- **Work**: unchanged of course, O(n log n)
- **Span** : Now recurrence takes the form:
  T(n) = O(n) + 1T(n/2) = **O(n)**
- (In fact it is possible to do even better is we parallelize the merge – we won't do that here.)

28