CSE332: Data Abstractions
Lecture 18: Parallel Sort

James Fogarty
Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman
Reductions

- Computations of this form are called reductions
- Produce single answer from collection via an associative operator
  - Examples: max, count, leftmost, rightmost, sum, ...
  - Non-example: median
- Recursive results don’t have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: Histogram of test results is a variant of sum
- But some things are inherently sequential
  - How we process arr[i] may depend entirely on the result of processing arr[i-1]
Maps and Data Parallelism

- A **map** operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support

- Canonical example: Vector addition

```java
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;
}
```
Parallel Prefix

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

[Non-standard terminology, filter does not emphasize stability]

Given an array input, produce an array output containing only elements such that \( f(\text{elt}) \) is true

Example: input \([17, 4, 6, 8, 11, 5, 13, 19, 0, 24]\)
\( f: \) is elt > 10
output \([17, 11, 13, 19, 24]\)

Parallelizable
  – Finding elements for the output is easy
  – But getting them in the right place seems hard
Pack as Map, Parallel Prefix, Map

1. Parallel map to compute a bit-vector for true elements
   
   | input   | [17, 4, 6, 8, 11, 5, 13, 19, 0, 24] |
   | bits    | [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]   |

2. Parallel-prefix sum on the bit-vector
   
   | bitsum  | [1, 1, 1, 1, 2, 2, 3, 4, 4, 5] |

3. Parallel map to produce the output
   
   | output  | [17, 11, 13, 19, 24] |

output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
**Quicksort Review**

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

**Best / expected case work**

1. Pick a pivot element  \hspace{1cm} O(1)
2. Partition all the data into:
   A. The elements less than the pivot  \hspace{1cm} O(n)
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C  \hspace{1cm} 2T(n/2)

How should we parallelize this?
Quicksort

Best / expected case work

1. Pick a pivot element \( O(1) \)
2. Partition all the data into:
   a. The elements less than the pivot \( O(n) \)
   b. The pivot
   c. The elements greater than the pivot
3. Recursively sort A and C \( 2T(n/2) \)

Easy: Do the two recursive calls in parallel

- Work: unchanged \( O(n \log n) \)
- Span: \( T(n) = O(n) + T(n/2) = O(n) + O(n/2) + T(n/4) = O(n) \)
- So parallelism is \( O(\log n) \) (i.e., work / span)
Doing Better

- $O(\log n)$ speed-up with infinite number of processors is okay, but a bit underwhelming
  - Sort $10^9$ elements 30 times faster

- Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized
  - The Internet has been known to be wrong
  - But we need auxiliary storage (will no longer in place)
  - In practice, constant factors may make it not worth it, but remember Amdahl’s Law and the long-term situation

- Already have everything we need to parallelize the partition
**Parallel Partition with Auxiliary Storage**

Partition all the data into:

A. The elements less than the pivot  
B. The pivot  
C. The elements greater than the pivot  

- This is just two packs  
  - We know a pack is $O(n)$ work, $O(\log n)$ span  
  - Pack elements less than pivot into left side of `aux` array  
  - Pack elements greater than pivot into right side of `aux` array  
  - Put pivot between them and recursively sort  
  - With a little more cleverness, can do both packs at once  
    - But no effect on asymptotic complexity
Analysis

With $O(\log n)$ span for partition, the total span for quicksort is

$$T(n) = O(\log n) + T(n/2)$$

$$= O(\log n) + O(\log n/2) + T(n/4)$$

$$= O(\log n) + O(\log n/2) + O(\log n/4) + T(n/8)$$

$$\ldots$$

$$= O(\log^2 n)$$

So parallelism (work / span) is $O(n / \log n)$
Example

• Step 1: pick pivot as median of three

8 1 4 9 0 3 5 2 7 6

• Steps 2a and 2c (combinable):
  pack less than and pack greater than into a second array
  – Fancy parallel prefix to pull this off not shown

  1 4 0 3 5 2
  1 4 0 3 5 2 6 8 9 7

• Step 3: Two recursive sorts in parallel
  – Can sort back into original array (swapping like in mergesort)
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 $O(n)$

Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the span to $T(n) = O(n) + 1T(n/2) = O(n)$

- Again, parallelism is $O(\log n)$
- To do better, need to parallelize the merge
  - The trick this time will not use parallel prefix
Parallelizing the Merge

Need to merge two sorted subarrays (may not have the same size)

| 0 | 1 | 4 | 8 | 9 | 2 | 3 | 5 | 6 | 7 |

Idea: Suppose the larger subarray has \( n \) elements. In parallel:

- merge the first \( n/2 \) elements of the larger half with the “appropriate” elements of the smaller half
- merge the second \( n/2 \) elements of the larger half with the remainder of the smaller half
Parallelizing the Merge

\[
\begin{array}{cccccc}
0 & 4 & 6 & 8 & 9 \\
1 & 2 & 3 & 5 & 7
\end{array}
\]
Parallelizing the Merge

1. Get median of bigger half:
Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value:
Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value: $O(\log n)$ to do binary search on the sorted small half
Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value: $O(\log n)$ to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: $O(1)$
Parallelizing the Merge

1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value: $O(\log n)$ to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: $O(1)$
4. Do two submerges in parallel
When we do each merge in parallel, we split the bigger array in half, and use binary search to split the smaller array, in base case we do the copy.
Analysis

• Sequential recurrence for mergesort:
  \[ T(n) = 2T(n/2) + O(n) \] which is \( O(n \log n) \)

• Doing the two recursive calls in parallel but a sequential merge:
  work: same as sequential  \[ \text{span}: T(n) = 1T(n/2) + O(n) \] which is \( O(n) \)

• Parallel merge makes work and span harder to compute
  – Each merge step does an extra \( O(\log n) \)
    binary search to find how to split the smaller subarray
  – To merge \( n \) elements total,
    do two smaller merges of possibly different sizes
  – But worst-case split is \((1/4)n\) and \((3/4)n\)
    • When subarrays same size and “smaller” splits “all” / “none”
**Analysis**

For just a parallel merge of $n$ elements:
- Span is $T(n) = T(3n/4) + O(\log n)$, which is $O(\log^2 n)$
- Work is $T(n) = T(3n/4) + T(n/4) + O(\log n)$ which is $O(n)$
- Neither bound is immediately obvious, but “trust us”

So for mergesort with parallel merge overall:
- Span is $T(n) = 1T(n/2) + O(\log^2 n)$, which is $O(\log^3 n)$
- Work is $T(n) = 2T(n/2) + O(n)$, which is $O(n \log n)$

So parallelism (work / span) is $O(n / \log^2 n)$
- Not quite as good as quicksort’s $O(n / \log n)$
  - But worst-case guarantee
  - And as always this is just the asymptotic result
Toward Sharing Resources

Have been studying parallel algorithms using fork-join
  – Lower span via parallel tasks

Algorithms all had a very simple structure to avoid race conditions
  – Each thread had memory “only it accessed”
    • Example: array sub-range
  – Or used fork and join as contract for who “had” memory
    • On fork, “loan” some memory to “forkee” and do not access that memory again until after join on the “forkee”

Strategy will not work well when:
  – Memory accessed by threads is overlapping or unpredictable
  – Threads are doing independent tasks needing access to same resources (as opposed to implementing the same algorithm)
Concurrency:
Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires *coordination*, particularly *synchronization* to avoid incorrect simultaneous access: make somebody *block*
  - *join* is not what we want
  - Want to block until another thread is “done with what we need”, not the more extreme “until completely done executing”

Even correct concurrent applications are usually highly *non-deterministic*: how threads are scheduled affects what each thread sees in its different operations
  - non-repeatability complicates testing and debugging
Examples

Multiple threads:

1. Processing different bank-account operations
   – What if 2 threads change the same account at the same time?

2. Using a shared cache of recent files (e.g., hashtable)
   – What if 2 threads insert the same file at the same time?

3. Creating a pipeline with a queue for handing work to next thread in sequence (i.e., a virtual assembly line)?
   – What if enqueuer and dequeuer adjust a circular array queue at the same time?
Why Threads?

Unlike parallelism, not about implementing algorithms faster

But threads still useful for:

- **Code structure for responsiveness**
  - Respond to GUI events in one thread while another thread is performing an expensive computation

- **Processor utilization (mask I/O latency)**
  - If 1 thread “goes to disk,” have something else to do

- **Failure isolation**
  - Convenient structure if want to *interleave* multiple tasks and do not want an exception in one to stop the other