CSE 332: Parallel Sorting

Richard Anderson
Spring 2016

1

Announcements

2

Recap

Last lectures
- simple parallel programs
- common patterns: map, reduce
- analysis tools (work, span, parallelism)

Now
- Amdahl's Law
- Parallel quicksort, merge sort
- useful building blocks: prefix, pack

3

Analyzing Parallel Programs

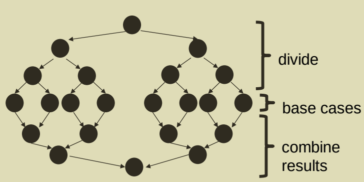
Let $\mathbf{T_P}$ be the running time on **P** processors

Two key measures of run-time:
- Work: How long it would take 1 processor = $\mathbf{T_1}$
- 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
  - Example: *O*(`log` *n*) for summing an array
  - Also called "critical path length" or "computational depth"

4

Divide and Conquer Algorithms

Our **`fork`** and **`join`** frequently look like this:



In this context, the span (T_∞) is:
- The longest dependence-chain; longest 'branch' in parallel 'tree'
- Example: *O*(log *n*) for summing an array; we halve the data down to our cut-off, then add back together; *O*(log *n*) steps, O(1) time for each
- Also called "critical path length" or "computational depth"

5

Parallel Speed-up

- Speed-up on **P** processors: $\mathbf{T_1 / T_P}$
- If speed-up is **P**, we call it perfect linear speed-up
  - e.g., doubling **P** halves running time
  - hard to achieve in practice
- Parallelism is the maximum possible speed-up: $\mathbf{T_1 / T_\infty}$
  - if you had infinite processors

6

Estimating T_p

- How to estimate $\mathbf{T_P}$ (e.g., P = 4)?
- Lower bounds on $\mathbf{T_P}$ (ignoring memory, caching...)
  1. $\mathbf{T_\infty}$
  2. $\mathbf{T_1 / P}$
  - which one is the tighter (higher) lower bound?
- The ForkJoin Java Framework achieves the following expected time asymptotic bound:

  $$\mathbf{T_P} \in \mathbf{O}(\mathbf{T_\infty} + \mathbf{T_1 / P})$$

  - this bound is optimal

7

Amdahl's Law

- Most programs have
  1. parts that parallelize well
  2. parts that don't parallelize at all
- The latter become bottlenecks

8

Amdahl's Law

- Let T_1 = 1 unit of time
- Let S = proportion that can't be parallelized

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

- Suppose we get perfect linear speedup on the parallel portion:

  $$\mathbf{T_P} =$$

- So the overall speed-up on P processors is (Amdahl's Law):

  $$\mathbf{T_1 / T_P} =$$

  $$\mathbf{T_1 / T_\infty} =$$

- If 1/3 of your program is parallelizable, max speedup is:

9

Pretty Bad News

- Suppose 25% of your program is sequential.
  - Then a billion processors won't give you more than a 4x speedup!
- What portion of your program must be parallelizable to get 10x speedup on a 1000 core GPU?
  - 10 <= 1 / (S + (1-S)/1000)
- Motivates minimizing sequential portions of your programs

10

Take Aways

- Parallel algorithms can be a big win
- Many fit standard patterns that are easy to implement
- Can't just rely on more processors to make things faster (Amdahl's Law)

11

Parallelizable?

Fibonacci (N)

12

Parallelizable?

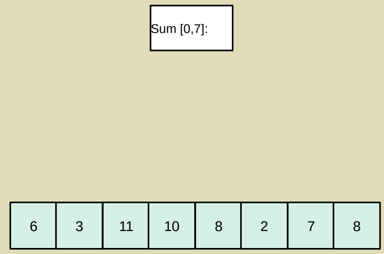
Prefix-sum:

`input`	6	3	11	10	8	2	7	8
`output`								

$output[i] = \sum_{0}^{i-1} input[i]$

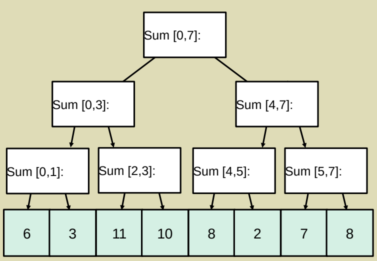
13

First Pass: Sum



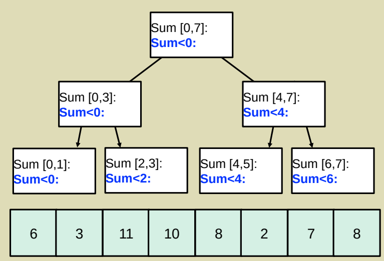
14

First Pass: Sum



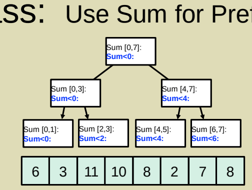
15

2nd Pass: Use Sum for Prefix-Sum



16

2nd Pass: Use Sum for Prefix-Sum



Go from root down to leaves

Root

- sum<0 =

Left-child

- sum<K =

Right-child

- sum<K =

17

Prefix-Sum Analysis

- First Pass (Sum):
  - span =
- Second Pass:
  - single pass from root down to leaves
    - update children's sum<K value based on parent and sibling
  - span =
- Total
  - span =

18

Parallel Prefix, Generalized

Prefix-sum is another common pattern (prefix problems)

- maximum element to the left of i
- is there an element to the left of i i satisfying some property?
- count of elements to the left of i satisfying some property
- …

We can solve all of these problems in the same way

19

Pack

Pack:

input	6	3	11	10	8	2	7	8	**test: X < 8?**
output									

Output array of elements satisfying `test`, in original order

20

Parallel Pack?

Pack

input	6	3	11	10	8	2	7	8	**test: X < 8?**
output	6	3	2	7					

- Determining **which** elements to include is **easy**
- Determining **where** each element goes in output is **hard**
  - seems to depend on previous results

21

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8	**test: X < 8?**
test	1	1	0	0	0	1	1	0	

22

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8	**test: X < 8?**
test	1	1	0	0	0	1	1	0	

2. transform bit vector into array of indices into result array

pos	1	2				3	4	

23

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8	**test: X < 8?**
test	1	1	0	0	0	1	1	0	

2. prefix-sum on bit vector

pos	**1**	**2**	2	2	2	**3**	**4**	4

3. map input to corresponding positions in output

output	6	3	2	7				

- `if (test[i] == 1) output[pos[i]] = input[i]`

24

Parallel Pack Analysis

- Parallel Pack
  1. map: O( ) span
  2. sum-prefix: O( ) span
  3. map: O( ) span
- Total: O( ) span

25

Sequential Quicksort

Quicksort (review):

1. Pick a pivot — O(1)
2. Partition into two sub-arrays — O(n)
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B — 2T(n/2), avg

Complexity (avg case)

- $T(n) = n + 2T(n/2)$ $T(0) = T(1) = 1$
- $O(n \log n)$

How to parallelize?

26

Parallel Quicksort

Quicksort

1. Pick a pivot — O(1)
2. Partition into two sub-arrays — O(n)
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B in parallel — T(n/2), avg

Complexity (avg case)

- $T(n) = n + T(n/2)$ $T(0) = T(1) = 1$
- Span: O( )
- Parallelism (work/span) = O( )

27

Taking it to the next level…

- O(`log n`) speed-up with infinite processors is okay, but a bit underwhelming
  - Sort 10^9 elements 30x faster
- Bottleneck:

28

Parallel Partition

Partition into sub-arrays

A. values less than pivot
B. values greater than pivot

What parallel operation can we use for this?

29

Parallel Partition

- Pick pivot

8	1	4	9	0	3	5	2	7	6

- Pack (test: `<6`)

1	4	0	3	5	2				

- Right pack (test: `>=6`)

1	4	0	3	5	2	6	8	9	7

30

Parallel Quicksort

Quicksort
1. Pick a pivot — O(1)
2. Partition into two sub-arrays — O( ) span
   A. values less than pivot
   B. values greater than pivot
3. Recursively sort A and B in parallel — T(n/2), avg

Complexity (avg case)
- T(n) = O(   ) + T(n/2)   T(0) = T(1) = 1
- Span: O(   )
- Parallelism (work/span) = O(   )

31

Sequential Mergesort

Mergesort (review):
1. Sort left and right halves — 2T(n/2)
2. Merge results — O(n)

Complexity (worst case)
- T(n) = n + 2T(n/2)   T(0) = T(1) = 1
- O(n logn)

How to parallelize?
- Do left + right in parallel, improves to O(n)
- To do better, we need to…

32

Parallel Merge

0	4	6	8	9

1	2	3	5	7

How to merge two sorted lists in parallel?

33

Parallel Merge

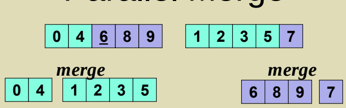
0	4	6	8	9
		M		

1	2	3	5	7

1. Choose median M of left half — O(   )
2. Split both arrays into < M, >=M — O(   )
   - how?

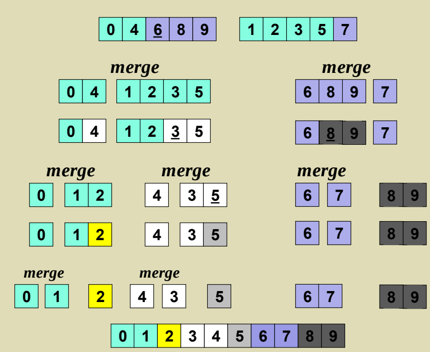
34

Parallel Merge

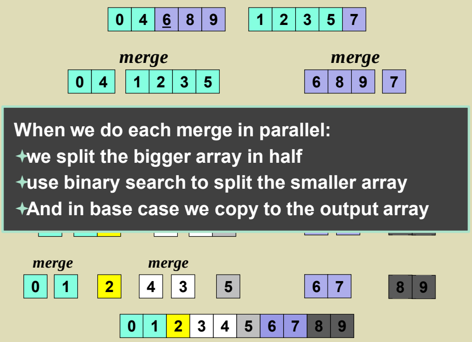


1. Choose median M of left half
2. Split both arrays into < M, >=M
   - how?
3. Do two submerges in parallel

35



36



37

Parallel Mergesort Pseudocode

```
Merge(arr[], left1, left2, right1, right2, out[], out1, out2 )
    int leftSize = left2 – left1
    int rightSize = right2 – right1
    // Assert: out2 – out1 = leftSize + rightSize
    // We will assume leftSize > rightSize without loss of generality

    if (leftSize + rightSize < CUTOFF)
        sequential merge and copy into out[out1..out2]

    int mid = (left2 – left1)/2
    binarySearch arr[right1..right2] to find j such that
        arr[j] ≤ arr[mid] ≤ arr[j+1]

    Merge(arr[], left1, mid, right1, j, out[], out1, out1+mid+j)
    Merge(arr[], mid+1, left2, j+1, right2, out[], out1+mid+j+1, out2)
```

38

Analysis

Parallel Merge (worst case)
- Height of partition call tree with n elements: O(   )
- Complexity of each thread (ignoring recursive call): O(   )
- Span: O(   )

Parallel Mergesort (worst case)
- Span: O(   )
- Parallelism (work / span): O(   )

Subtlety: uneven splits

0	4	6	8		1	2	3	5

- but even in worst case, get a 3/4 to 1/4 split
  - still gives O(log n) height

39

Parallel Quicksort vs. Mergesort

Parallelism (work / span)
- quicksort: O(n / log n) avg case
- mergesort: O(n / $\log^2$ n) worst case

40