CSE 332: Parallel Sorting

Richard Anderson, Steve Seitz
Winter 2014

1

Announcements

- Project 3 PartA due Thursday night

2

Recap

Last week
- simple parallel programs
- common patterns: map, reduce
- analysis tools (work, span, parallelism)
- Amdahl's Law

Now
- parallel quicksort, merge sort
- useful building blocks: prefix, pack

3

Parallelizable?

Fibonacci (N)

4

Parallelizable?

Prefix-sum:

input	6	3	11	10	8	2	7	8
output								

$output[i] = \sum_0^{i-1} input[i]$

5

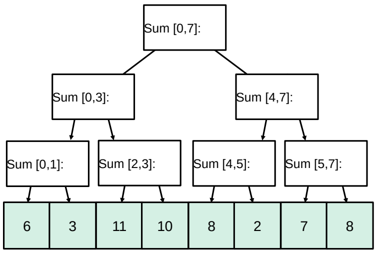
First Pass: Sum

Sum [0,7]:

6	3	11	10	8	2	7	8

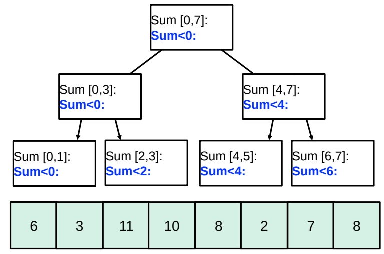
6

First Pass: Sum



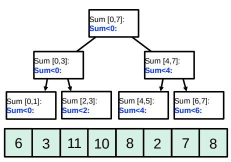
7

2nd Pass: Use Sum for Prefix-Sum



8

2nd Pass: Use Sum for Prefix-Sum



Go from root down to leaves

Root
- sum<0 =

Left-child
- sum<K =

Right-child
- sum<K =

9

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 =

10

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

11

Pack

Pack:

`input`	6	3	11	10	8	2	7	8	`test: x < 8?`
`output`									

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

12

Parallel Pack?

Pack

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

`test: x < 8?`

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

13

Parallel Pack

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

`input`	6	3	11	10	8	2	7	8
`test`	1	1	0	0	0	1	1	0

`test: x < 8?`

14

Parallel Pack

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

`input`	6	3	11	10	8	2	7	8
`test`	1	1	0	0	0	1	1	0

`test: x < 8?`

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

`pos`	1	2				3	4	

15

Parallel Pack

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

`input`	6	3	11	10	8	2	7	8
`test`	1	1	0	0	0	1	1	0

`test: x < 8?`

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]`

16

Parallel Pack Analysis

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

17

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 logn)

How to parallelize?

18

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(   )

19

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:

20

Parallel Partition

Partition into sub-arrays

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

What parallel operation can we use for this?

21

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

22

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(   )

23

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…

24

Parallel Merge

0 4 6 8 9   1 2 3 5 7

How to merge two sorted lists in parallel?

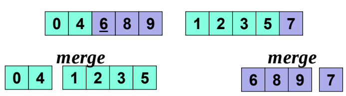
Parallel Merge

0 4 6 8 9   1 2 3 5 7

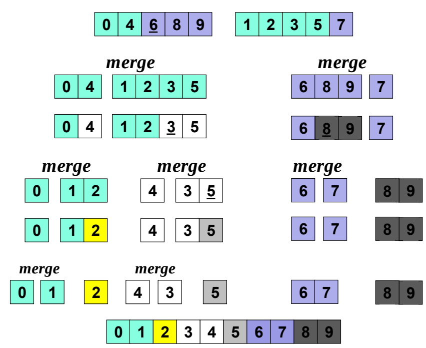
M

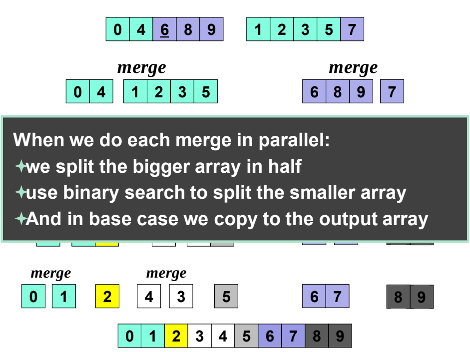
1. Choose median M of left half   O(   )
2. Split both arrays into < M, >=M   O(   )
   - how?

Parallel Merge



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





When we do each merge in parallel:
- we split the bigger array in half
- use binary search to split the smaller array
- And in base case we copy to the output array

Parallel Mergesort Pseudocode

Merge(arr[], $left_1$, $left_2$, $right_1$, $right_2$, out[], out_1, out_2)

    int leftSize = $left_2$ – $left_1$

    int rightSize = $right_2$ – $right_1$

    // Assert: out_2 – out_1 = 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 = ($left_2$ – $left_1$)/2

    binarySearch arr[right1..right2] to find j such that

        arr[j] ≤ arr[mid] ≤ arr[j+1]

    Merge(arr[], $left_1$, mid, $right_1$, j, out[], out_1, out_1+mid+j)

    Merge(arr[], mid+1, $left_2$, j+1, $right_2$, out[], out_1+mid+j+1, out_2)

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

Parallel Quicksort vs. Mergesort

Parallelism (work / span)

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