CSE 332: Parallel Sorting

Richard Anderson
Spring 2016

Announcements

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

Analyzing Parallel Programs

Let $T_P$ be the running time on $P$ processors

Two key measures of run-time:
- **Work**: How long it would take 1 processor = $T_1$
- **Span**: How long it would take infinity processors = $T_\infty$
  - 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”

Divide and Conquer Algorithms

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

In this context, the span ($T_\infty$) 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”

Parallel Speed-up

- **Speed-up** on $P$ processors: $T_1 / T_P$
- 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: $T_1 / T_\infty$
  - If you had infinite processors
### Estimating $T_p$
- How to estimate $T_p$ (e.g., $P = 4$)?
- Lower bounds on $T_p$ (ignoring memory, caching...)
  1. $T^\infty$
  2. $T_1 / P$
    - which one is the tighter (higher) lower bound?
- The ForkJoin Java Framework achieves the following expected time asymptotic bound:
  \[ T_p \in O(T^\infty + T_1 / P) \]
  - this bound is optimal

### Amdahl’s Law
- Most programs have
  1. parts that parallelize well
  2. parts that don’t parallelize at all

### 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?
  - \[ 10 \leq 1 / (S + (1-S)/1000) \]
  - Motivates minimizing sequential portions of your programs

### 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)

### Parallelizable?
Fibonacci (N)
Parallelizable?

Prefix-sum:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 11 10 8 2 7 8</td>
<td></td>
</tr>
</tbody>
</table>

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

First Pass: Sum

First Pass: Sum

<table>
<thead>
<tr>
<th>Sum[0,7]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 11 10 8 2 7 8</td>
</tr>
</tbody>
</table>

2nd Pass: Use Sum for Prefix-Sum

2nd Pass: Use Sum for Prefix-Sum

<table>
<thead>
<tr>
<th>Sum[0,7]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 11 10 8 2 7 8</td>
</tr>
</tbody>
</table>

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 =
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 \( i \) 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

Pack:

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Output array of elements satisfying test, in original order

Parallel Pack?

Pack

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Parallel Pack

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

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>3</th>
<th>11</th>
<th>10</th>
<th>8</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

| pos    | 1  | 2  | 3  | 4  |

3. map input to corresponding positions in output

| output | 6  | 3  | 2  | 7  | 8  |

- if (test\[i\] == 1) output[pos\[i\]] = input\[i\]
Parallel Pack Analysis

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

Sequential Quicksort

Quick sort (review):
1. Pick a pivot $O(1)$
2. Partition into two sub-arrays $O(n)$
   A. values less than pivot $O(n)$
   B. values greater than pivot $O(n)$
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?

Parallel Quicksort

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

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

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:

Parallel Partition

Partition into sub-arrays
A. values less than pivot
B. values greater than pivot

What parallel operation can we use for this?

Parallel Partition

- Pick pivot
  [8 1 4 9 0 3 5 2 7 6]
- Pack (test: <6)
  [1 4 0 3 5 2 0 0 0 0 0]
- Right pack (test: >=6)
  [1 4 0 3 5 2 6 0 0 0 0]
Parallel Quicksort

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

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

Sequential Mergesort

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

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

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

Parallel Merge

How to merge two sorted lists in parallel?

1. Choose median \(M\) of left half \(O(\text{median})\)
2. Split both arrays into \(< M, \geq M\) \(O(\text{split})\)
   - how?
3. Do two submerges in parallel

Parallel Merge

1. Choose median \(M\) of left half \(O(\text{median})\)
2. Split both arrays into \(< M, \geq M\)
   - how?
3. Do two submerges in parallel
Parallel Mergesort Pseudocode

Merge(arr[], left, left, right, right, out[], out[], out[])

int leftSize = left - left;
int rightSize = right - right;
// Assert: out[] = out[] + leftSize + rightSize
// We will assume leftSize < rightSize without loss of generality
if (leftSize + rightSize < CUTOFF)
  sequential merge and copy into out[]
  int mid = (left + left)/2
  binarySearch arr[] to find j such that
  arr[j] ≤ arr[mid] ≤ arr[j+1]
  Merge(arr[], left, mid, right, j, out[], out[])
  Merge(arr[], mid+1, right, j+1, right, out[], out[])

Analysis

Parallel Merge (worst case)
- Height of partition call tree with n elements: O(1)
- Complexity of each thread (ignoring recursive call): O(1)
- Span: O(1)

Parallel Mergesort (worst case)
- Span: O(1)
- Parallelism (work / span): O(1)

Subtlety: uneven splits
- 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