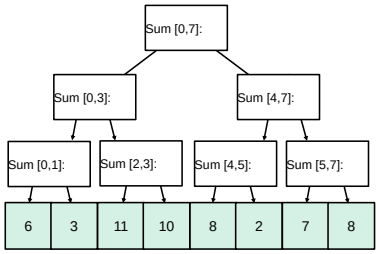
CSE 332: Parallel Sorting II

Richard Anderson, Steve Seitz
Winter 2014

Announcements

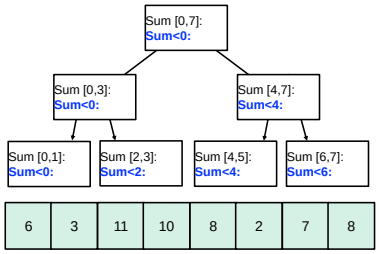
- Project 3 Part A due Thursday night

Review: Parallel prefix



2nd Pass:

Passing partial sums back down



Parallel Prefix

Work: O(n)
Span: O(log n)
Applies to any associative operation
A + (B + C) = (A + B) + C

Example: Pack
1. Map to 0-1 indicator vector for property
2. Compute prefix sum of indicator vector
3. Move elements to locations in result vector

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?

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( )

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

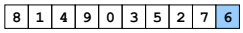
Partition into sub-arrays

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

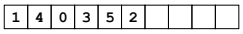
What parallel operation can we use for this?

Parallel Partition

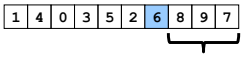
- Pick pivot



- Pack (test: **<6**)



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



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( )

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)

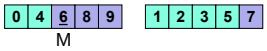
How to parallelize?

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

Parallel Merge

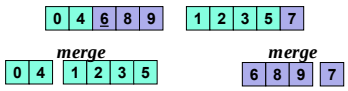
How to merge two sorted lists in parallel?

Parallel Merge

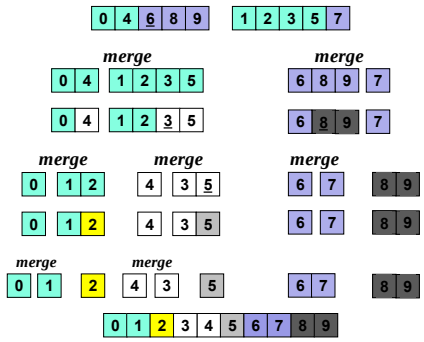


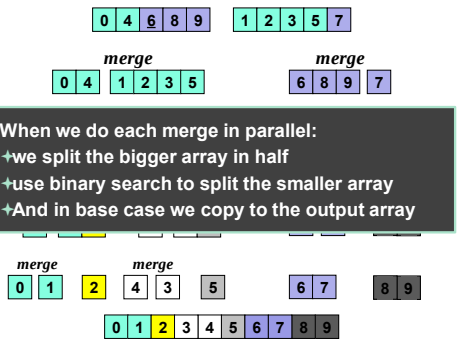
1. Choose median M of left half O( )
2. Split both arrays into < M, >=M O( )
   - how?

Parallel Merge



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





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)
```

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

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

19

Parallel Quicksort vs. Mergesort

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

20

CSE 332: Concurrency

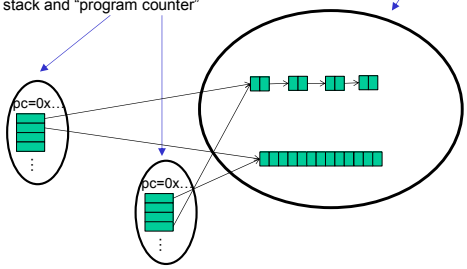
Richard Anderson, Steve Seitz
Winter 2014

21

Really sharing memory between Threads

Heap for all objects and static fields, *shared* by all threads

2 Threads, each with own *unshared* call stack and "program counter"



11/18/2013

22

Banking

Two threads both trying to **`withdraw(100)`** from the same account:
- Assume initial **`balance`** 150

```
class BankAccount {
  private int balance = 0;
  int  getBalance()       { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
  }
  … // other operations like deposit, etc.
}
```

Thread 1
```
x.withdraw(100);
```

Thread 2
```
x.withdraw(100);
```

23

A bad interleaving

Interleaved **`withdraw(100)`** calls on the same account
- Assume initial **`balance == 150`**

Time

Thread 1
```
int b = getBalance();

if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2
```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

- **How to fix?**

24

Concurrent Programming

Concurrency:

Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires *coordination*, particularly

- synchronization to avoid incorrect simultaneous access:
- make others *block* (wait) until the resource is free

Concurrent applications are often non-deterministic

- how threads are scheduled affects what operations happen first
- non-repeatability complicates testing and debugging

25

Concurrency Examples

What if we have 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 (e.g., hashtable) of recent files
   - What if 2 threads insert the same file at the same time?
3. Creating a pipeline (think assembly line) with a queue for handing work from one thread to next thread in sequence?
   - What if enqueuer and dequeuer adjust a circular array queue at the same time?

26

Why threads?

Unlike parallelism, not about implementing algorithms faster

But threads still useful for:

- *Code structure for responsiveness*
  - Example: 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

11/18/2013 27

Sharing, again

It is common in concurrent programs that:

- Different threads might access the same resources in an unpredictable order or even at about the same time
- Program correctness requires that simultaneous access be prevented using synchronization
- Simultaneous access is rare
  - Makes testing difficult
  - Must be much more disciplined when designing / implementing a concurrent program
  - Will discuss common idioms known to work

11/18/2013 28