Today:

– Multithreaded Algs.

COSC 581, Algorithms

March 25, 2014

Many of these slides are adapted from several online sources

Reading Assignments

- Today’s class:
  - Chapter 27.3

- Reading assignment for next class:
  - Chapter 29.1

- Announcement: Exam #2 on Tuesday, April 1
  - Will cover greedy algorithms, amortized analysis
  - HW 6-9

Remember Example from last time?

- Consider a program prototyped on 32-processor computer, but aimed to run on supercomputer with 512 processors
- Designers incorporated an optimization to reduce run time of benchmark on 32-processor machine, from $T_{32} = 65$ to $T'_{32} = 40$
- But, can show that this optimization made overall runtime on 512 processors slower than the original! Thus, optimization didn't help.
- Analysis for 32 processors:

Original:

$T_1 = 2048$

$T_\infty = 1$

$T_P = {T_1}/{P} + T_\infty$

$\Rightarrow T_{32} = 2048/32 + 1 = 65$

Optimized:

$T'_1 = 1024$

$T'_\infty = 8$

$T'_P = {T'_1}/{P} + T'_\infty$

$\Rightarrow T'_{32} = 1024/32 + 8 = 40$

- Analysis for 512 processors:

Original:

$T_1 = 2048$

$T_\infty = 1$

$T_P = {T_1}/{P} + T_\infty$

$\Rightarrow T_{512} = 2048/512 + 1 = 5$

Optimized:

$T'_1 = 1024$

$T'_\infty = 8$

$T'_P = {T'_1}/{P} + T'_\infty$

$\Rightarrow T'_{512} = 1024/512 + 8 = 10$

Remember Example from last time?

- Consider a program prototyped on 32-processor computer, but aimed to run on supercomputer with 512 processors
- Designers incorporated an optimization to reduce run time of benchmark on 32-processor machine, from $T_{32} = 65$ to $T'_{32} = 40$
- But, can show that this optimization made overall runtime on 512 processors slower than the original! Thus, optimization didn't help.
- Analysis for 32 processors:

Original:

$T_1 = 2048$

$T_\infty = 1$

$T_P = {T_1}/{P} + T_\infty$

$\Rightarrow T_{32} = 2048/32 + 1 = 65$

Optimized:

$T'_1 = 1024$

$T'_\infty = 8$

$T'_P = {T'_1}/{P} + T'_\infty$

$\Rightarrow T'_{32} = 1024/32 + 8 = 40$

- Analysis for 512 processors:

Original:

$T_1 = 2048$

$T_\infty = 1$

$T_P = {T_1}/{P} + T_\infty$

$\Rightarrow T_{512} = 2048/512 + 1 = 5$

Optimized:

$T'_1 = 1024$

$T'_\infty = 8$

$T'_P = {T'_1}/{P} + T'_\infty$

$\Rightarrow T'_{512} = 1024/512 + 8 = 10$

Question: For how many processors do the 2 versions of the program run equally fast?

In-Class Exercise #1

Consider the following procedures:

```
PROC-1(x)
    n = x.length
    let y[1..n] be a new array
    PROC-SUB(x, y, 1, n)
    return y

PROC-SUB(x, y, i, j)
    if i == j
        y[i] = x[i]
    else k = ⌊(i + j)/2⌋
        spawn PROC-SUB(x, y, i, k)
        PROC-SUB(x, y, k + 1, j)
        sync
        parallel for l = k + 1 to j
            y[l] = y[k] + y[l]
```

What is the work of PROC-1?

What is the span of PROC-1?

What is the parallelism of PROC-1?

In-Class Exercise #2a

Consider the following multithreaded algorithm:

P-FUNC(X, Y, N)
if N = 1
then Y[1,1] ← X[1,1]
Else Partition X into 4 (N/2) × (N/2) submatrices X_{11}, X_{12}, X_{21}, X_{22}
Partition Y into four (N/2) × (N/2) submatrices Y_{11}, Y_{12}, Y_{21}, Y_{22}
spawn P-FUNC (X_{11}, Y_{11}, N/2)
spawn P-FUNC (X_{12}, Y_{21}, N/2)
spawn P-FUNC (X_{21}, Y_{12}, N/2)
spawn P-FUNC (X_{22}, Y_{22}, N/2)
sync

What is the work of P-FUNC?

What is the span of P-FUNC?

What is the parallelism of P-FUNC?

In-Class Exercise #2b

Consider the following revised version of the multithreaded algorithm:

P-FUNC-REV(X, Y, N)
    if N = 1
    then Y[1,1] ← X[1,1]
    else
        Partition X into four (N/2) × (N/2)
            submatrices X_{11}, X_{12}, X_{21}, and X_{22}
        Partition Y into four (N/2) × (N/2)
            submatrices Y_{11}, Y_{12}, Y_{21}, and Y_{22}
        spawn P-FUNC-REV (X_{11}, Y_{11}, N/2)
        sync
        spawn P-FUNC-REV (X_{12}, Y_{21}, N/2)
        sync
        spawn P-FUNC-REV (X_{21}, Y_{12}, N/2)
        P-FUNC-REV (X_{22}, Y_{22}, N/2)
        sync

What is the work of P-FUNC-REV?

What is the span of P-FUNC-REV?

What is the parallelism of P-FUNC-REV?

Multithreaded Merge Sort

```
MERGE-SORT'(A, p, r)
    if p < r
        q = ⌊(p + r)/2⌋
    spawn MERGE-SORT'(A, p, q)
    MERGE-SORT'(A, q +1, r)
    sync
    MERGE(A, p, q, r)
```

Same as original merge-sort, except we execute the 2 recursive calls in parallel

Multithreaded Merge Sort

MERGE-SORT′(A, p, r)
    if p < r
        $q = \lfloor (p + r)/2 \rfloor$
    spawn MERGE-SORT′(A, p, q)
    MERGE-SORT′(A, q +1, r)
    sync
    MERGE(A, p, q, r)

Same as original merge-sort, except we execute the 2 recursive calls in parallel

Analysis

Work:

Span:

Parallelization:

Multithreaded Merge Sort

MERGE-SORT′(*A*, *p*, r)
    if *p* < r
        $q = \lfloor (p + r)/2 \rfloor$
    spawn MERGE-SORT′(*A*, *p*, *q*)
    MERGE-SORT′(*A*, *q* +1, *r*)
    sync
    MERGE(*A*, *p*, *q*, *r*)

Analysis

Work:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Span:

Parallelization:

Same as original merge-sort, except we execute the 2 recursive calls in parallel

Multithreaded Merge Sort

MERGE-SORT′(*A*, *p*, r)

if p < r

$q = \lfloor (p + r)/2 \rfloor$

spawn MERGE-SORT′(*A*, *p*, *q*)

MERGE-SORT′(*A*, *q* +1, *r*)

sync

MERGE(*A*, *p*, *q*, *r*)

Same as original merge-sort, except we execute the 2 recursive calls in parallel

Analysis

Work:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Span:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n)$$

Parallelization:

Multithreaded Merge Sort

MERGE-SORT′(A, p, r)
  if p < r
    $q = \lfloor (p + r)/2 \rfloor$
  spawn MERGE-SORT′(A, p, q)
  MERGE-SORT′(A, q +1, r)
  sync
  MERGE(A, p, q, r)

Same as original merge-sort, except we execute the 2 recursive calls in parallel

Analysis

Work:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Span:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n)$$

Parallelization:

$$= \frac{\Theta(n \lg n)}{\Theta(n)} = \Theta(\lg n)$$

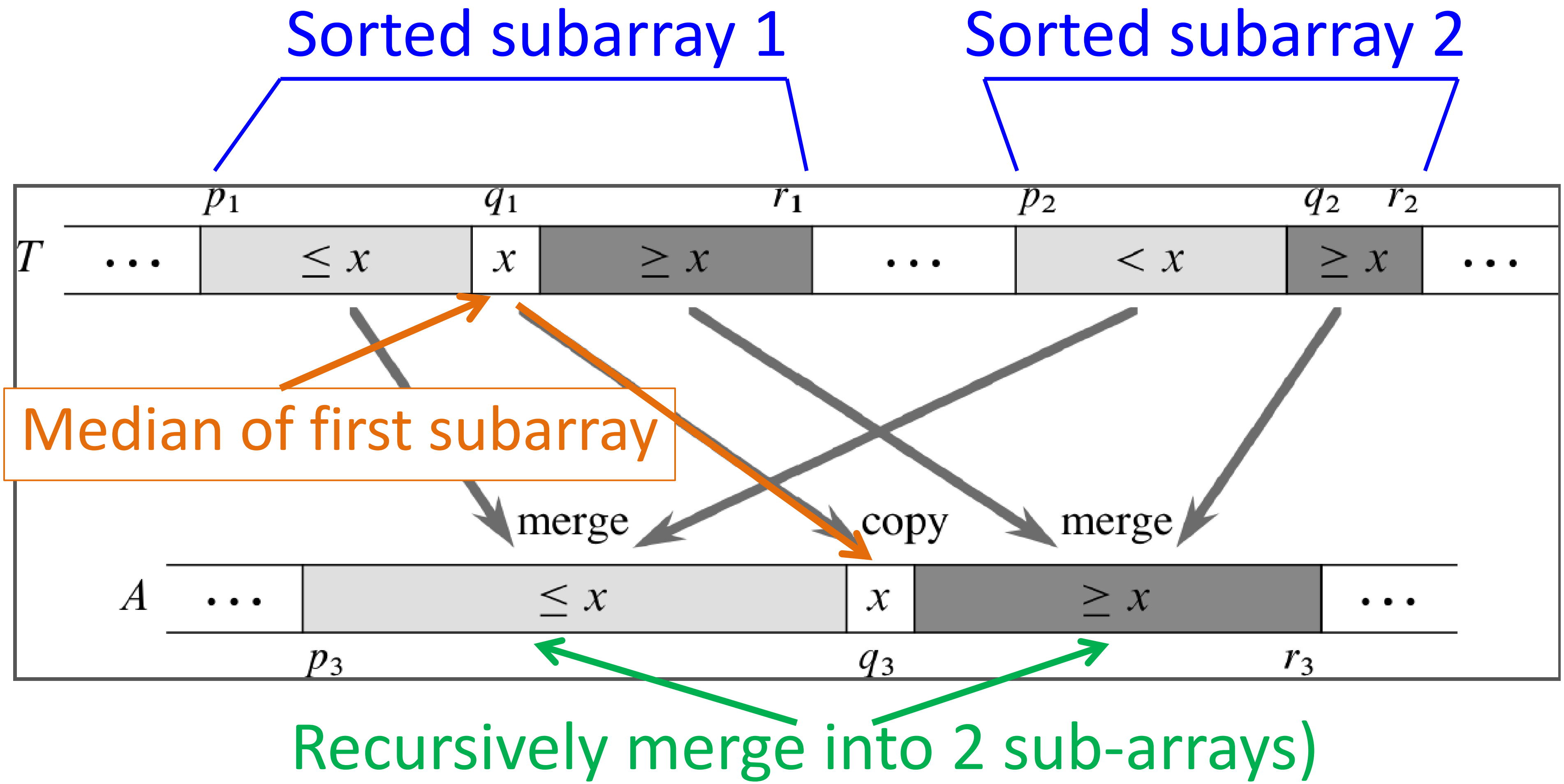
Problem with Merge

- Serial MERGE is dominating the performance
- How can we parallelize MERGE?

Problem with Merge

- Serial MERGE is dominating the performance
- How can we parallelize MERGE?
- Divide-and-conquer:
  - Put the middle element, *z*, of the larger of the two lists in the correct position
  - Merge the subarrays containing elements smaller than *z*
  - Merge the subarrays containing elements greater than *z*

Parallel Merge Idea



Parallel Merge

P-MERGE(T, p_1, r_1, p_2, r_2, A, p_3)
$n_1 = r_1 - p_1 + 1$
$n_2 = r_2 - p_2 + 1$
if $n_1 > n_2$
    swap P's, r's and n's
if $n_1 == 0$
    return
else
    $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
    $q_2 = \text{BINARY-SEARCH}(T[q_1], T, p_2, r_2)$
    $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$ // Where to put $T[q_1]$
    $A[q_3] = T[q_1]$
    spawn P-MERGE(T, p_1, $q_1 - 1$, p_2, $q_2 - 1$, A, p_3)
    P-MERGE(T, q_1 + 1, r_1, q2, r_2, A, q_3 + 1)
    sync

Parallel Merge Analysis

- Span:
  - Identify the maximum number of elements in the largest call to P-MERGE
  - The worst case merges ${}^{n_1}/_2$ elements (from the larger subarray) with all n_2 elements (from the smaller subarray):

$$\begin{aligned}\lfloor n_1/2 \rfloor + n_2 &\leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} \\ &= \frac{n_1 + n_2}{2} + \frac{n_2}{2} \\ &\leq \frac{n}{2} + \frac{n}{4} \\ &= 3n/4\end{aligned}$$

$$\begin{aligned}T_\infty(n) &= T_\infty\left(\frac{3n}{4}\right) + \Theta(\lg n) \\ &= \Theta(\lg^2 n)\end{aligned}$$

Parallel Merge Analysis

- Work:

$$T_1(n) = T_1(\alpha\, n) + T_1\big((1-\alpha)n\big) + O(\lg n)$$

where ${}^1/_4 \leq \alpha \leq {}^3/_4$

Can show that $T_1(n) \leq c_1 n - c_2 \lg n$ for constants c_1, c_2, and thus prove that $T_1(n) = \Theta(n)$

Parallel Merge Sort

```
P-MERGESORT(A, p, r, B, s)
n = r – p + 1
if n == 1
      B[s] = A[p]
else
    let T[n] be a new array
    q = ⌊(p + r)/2⌋
    q′ = q – p + 1
    spawn P-MERGE-SORT(A, p, q, T, 1)
    P-MERGE-SORT(A, q + 1, r, T, q′ +1)
    sync
    P-MERGE(T, 1, q′,q′ +1, n, B, s)
```

Analysis

Work:

Span:

Parallelization:

Parallel Merge Sort

```
P-MERGESORT(A, p, r, B, s)
n = r – p + 1
if n == 1
      B[s] = A[p]
else
    let T[n] be a new array
    q = ⌊(p + r)/2⌋
    q′ = q − p + 1
    spawn P-MERGE-SORT(A, p, q, T, 1)
    P-MERGE-SORT(A, q + 1, r, T, q′ +1)
    sync
    P-MERGE(T, 1, q′,q′ +1, n, B, s)
```

Analysis

Work:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Span:

Parallelization:

Parallel Merge Sort

P-MergeSort(A, p, r, B, s)
n = r – p + 1
if n == 1
    B[s] = A[p]
else
    let T[n] be a new array
    $q = \lfloor (p+r)/2 \rfloor$
    $q' = \mathrm{q} - \mathrm{p} + 1$
    spawn P-Merge-Sort(A, p, q, T, 1)
    P-Merge-Sort(A, q + 1, r, T, q′ +1)
    sync
    P-Merge(T, 1, q′,q′ +1, n, B, s)

<u>Analysis</u>

Work:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Span:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(\lg^2 n)$$
$$= \Theta(\lg^3 n)$$

Parallelization:

Parallel Merge Sort

```
P-MERGESORT(A, p, r, B, s)
n = r – p + 1
if n == 1
      B[s] = A[p]
else
    let T[n] be a new array
    q = ⌊(p + r)/2⌋
    q′ = q − p + 1
    spawn P-MERGE-SORT(A, p, q, T, 1)
    P-MERGE-SORT(A, q + 1, r, T, q′ +1)
    sync
    P-MERGE(T, 1, q′,q′ +1, n, B, s)
```

<u>Analysis</u>

Work:

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Span:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + \Theta(\lg^2 n)$$
$$= \Theta(\lg^3 n)$$

Parallelization:

$$= \frac{\Theta(n \lg n)}{\Theta(\lg^3 n)} = \Theta\left({}^{n}/_{\lg^2 n}\right)$$

Summary of Multithreading

We've looked at the following:

- How to create a computation dag, and analyze it in terms of work and span.
- How to write parallel code using parallel, spawn, and sync.
- How to analyze parallel code in terms of work, span, and parallelism.
- How to determine whether code has a race condition.
- Parallel algorithms for:
  - multithreaded matrix multiplication
  - multithreaded merge sort

Reading Assignments

- Reading assignment for next class:
  - Chapter 29.1

- Announcement: Exam #2 on Tuesday, April 1
  - Will cover greedy algorithms, amortized analysis
  - HW 6-9