

Lecture 7

Announcements

- Section

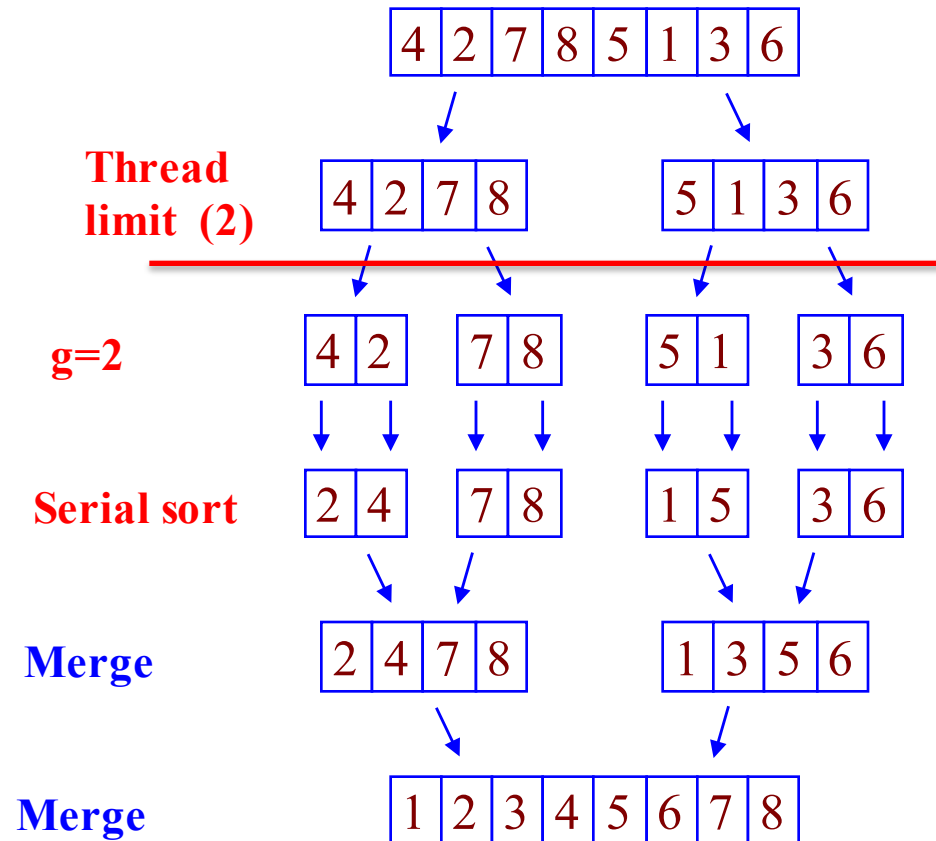
Have you been to section; why or why not?

- A. I have class and cannot make either time
- B. I have work and cannot make either time
- C. I went and found section helpful
- D. I went and did **not** find section helpful

What else can you say about section?

- A. It's not clear what the purpose of section is
- B. There other things I'd like to see covered in section
- C. I didn't go
- D. Both A and B
- E. Both A and C

Recapping from last time: Merge Sort



In general, $N/g \ll N/\#$ threads and you'll reach the 'g' limit before the thread limit

What should be done if the maximum limit on the number of threads is reached and the block size is still greater than g ?

- A. We should continue to split the work further until we reach a block size of g , but without spawning any more threads
- B. We should switch to the serial MergeSort algorithm
- C. We should stop splitting the work and use some other sorting algorithm
- D. A & B
- E. A & C

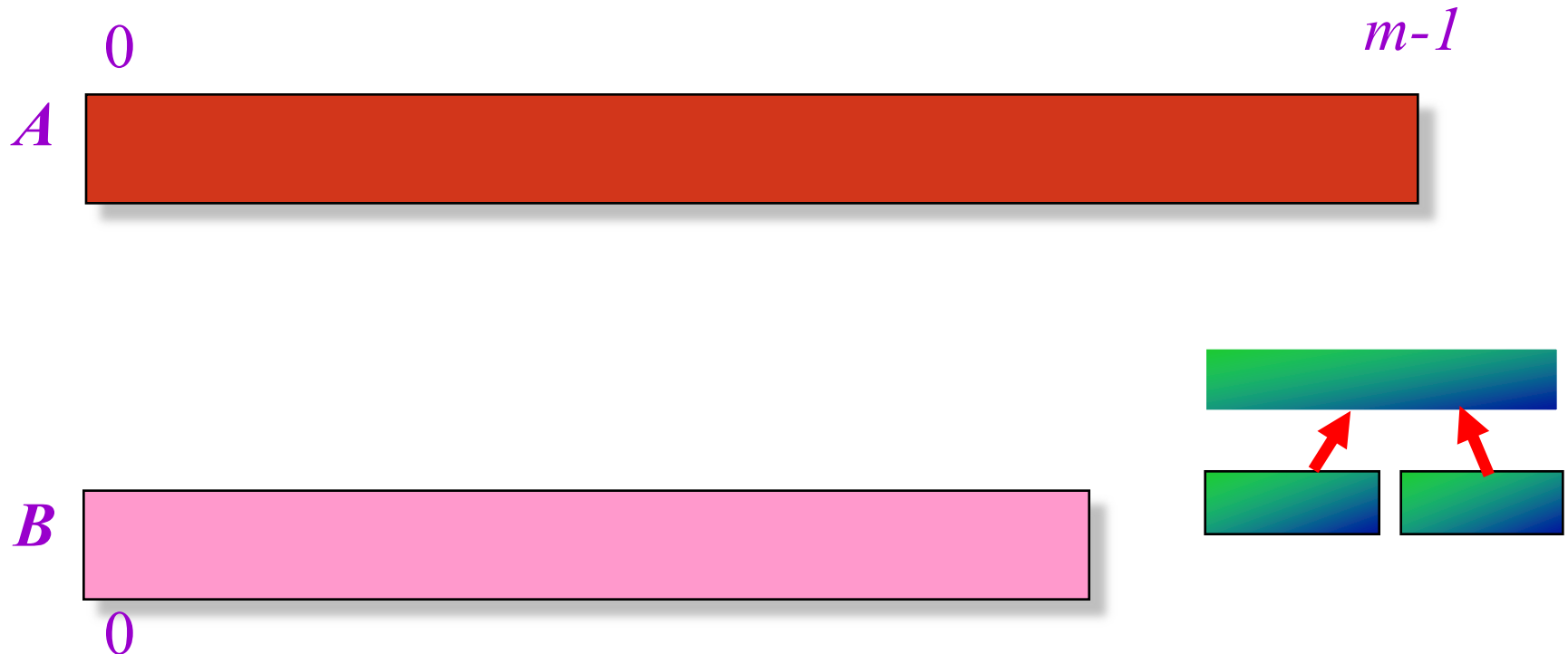
Merge

- Recall that we handled merge with just 1 thread
- But as we return from the recursion we use fewer and fewer threads: at the top level, we are merging the entire list on just 1 thread
- As a result, there is $\Theta(\lg N)$ parallelism
- There is a parallel merge algorithm that can do better



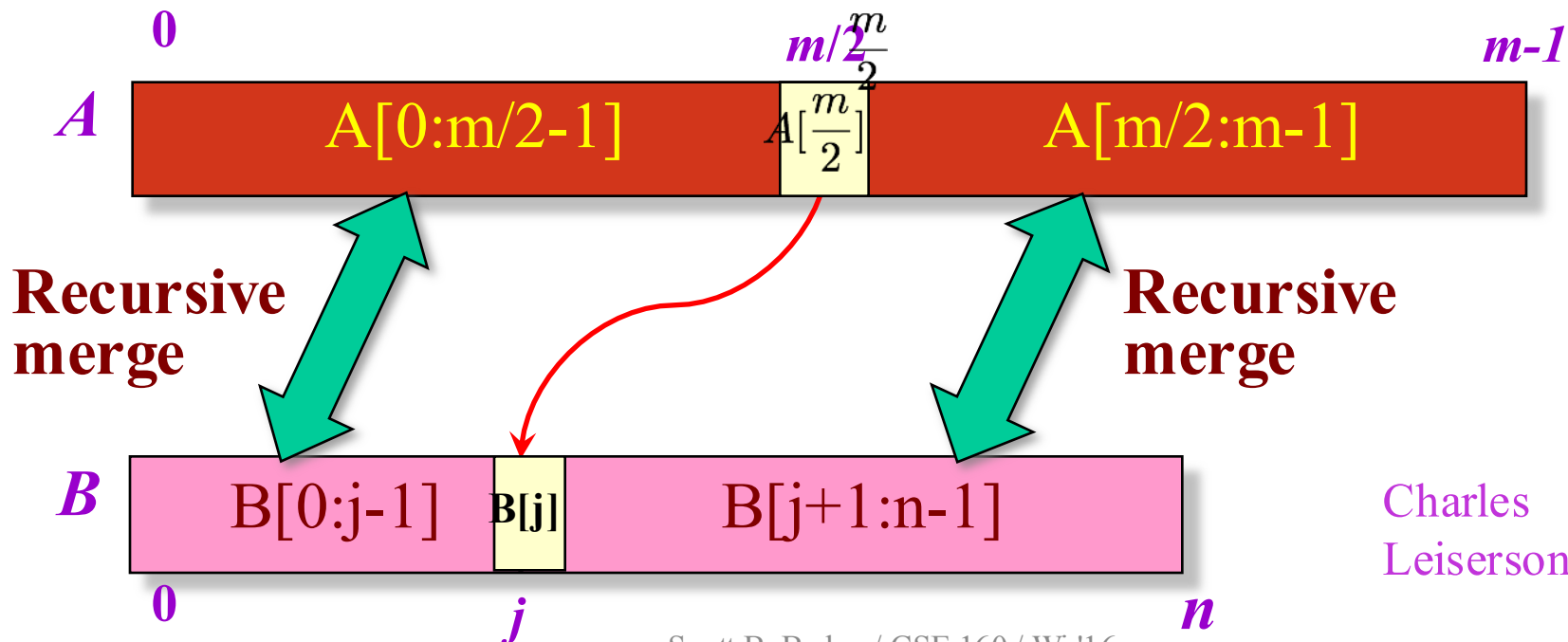
Parallel Merge - Preliminaries

- Assume we are merging $N=m+n$ elements stored in two arrays A and B of length m and n , respectively
- Assume $m \geq n$ (switch A and B if necessary)
- Locate the median of A ($@m/2$)



Parallel Merge Strategy

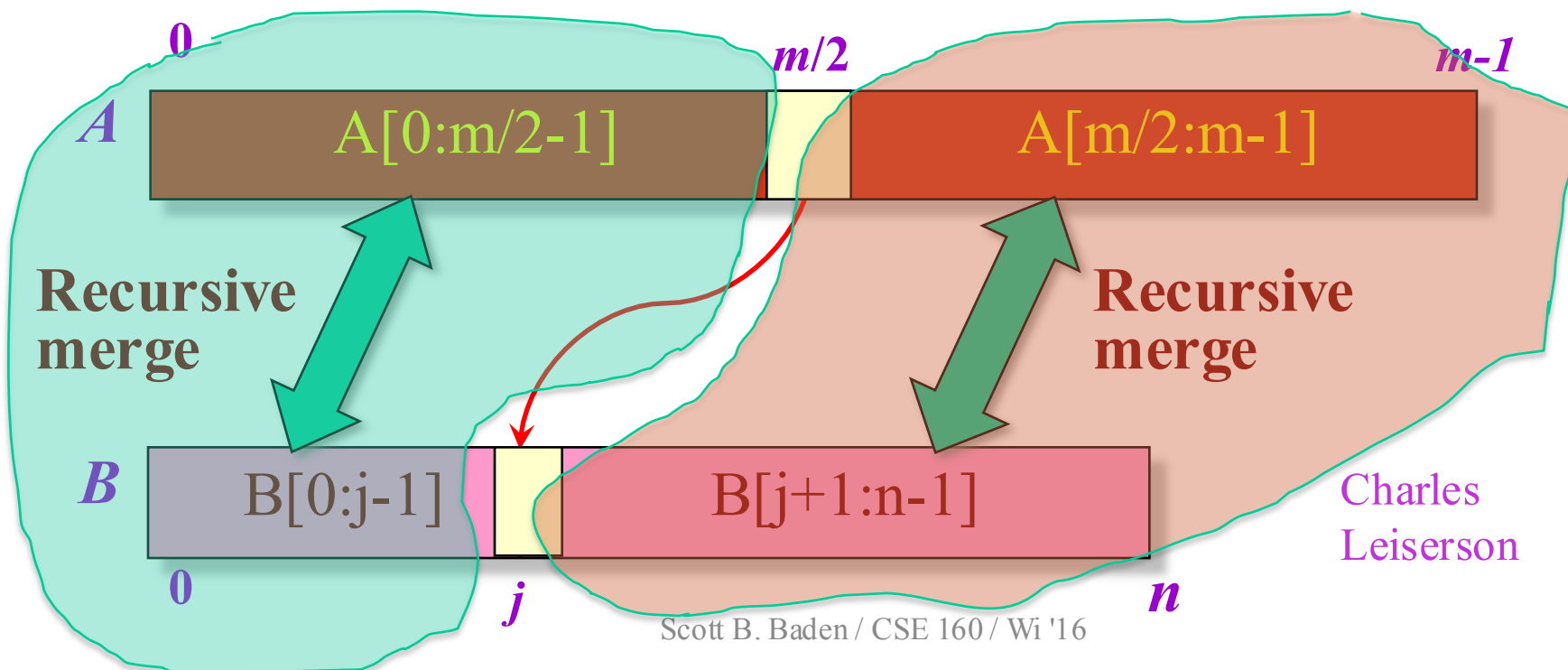
- Search for the $B[j]$ closest to, but not larger than, the median @ $A[m/2]$ (assumes no duplicates)
- Thus, when we insert $A[m/2]$ between $B[0:j-1]$ & $B[j:n-1]$, the list remains sorted
- Recursively merge into a new array $C[]$
 - $C[0:j+m/2-2] \leftarrow (A[0:m/2-1], B[0:j-1])$
 - $C[j+m/2:N] \leftarrow (A[m/2:m-1], B[j:n-1])$
 - $C[j+m/2-1] \leftarrow A[m/2]$



Charles
Leiserson

Parallel Merge - II

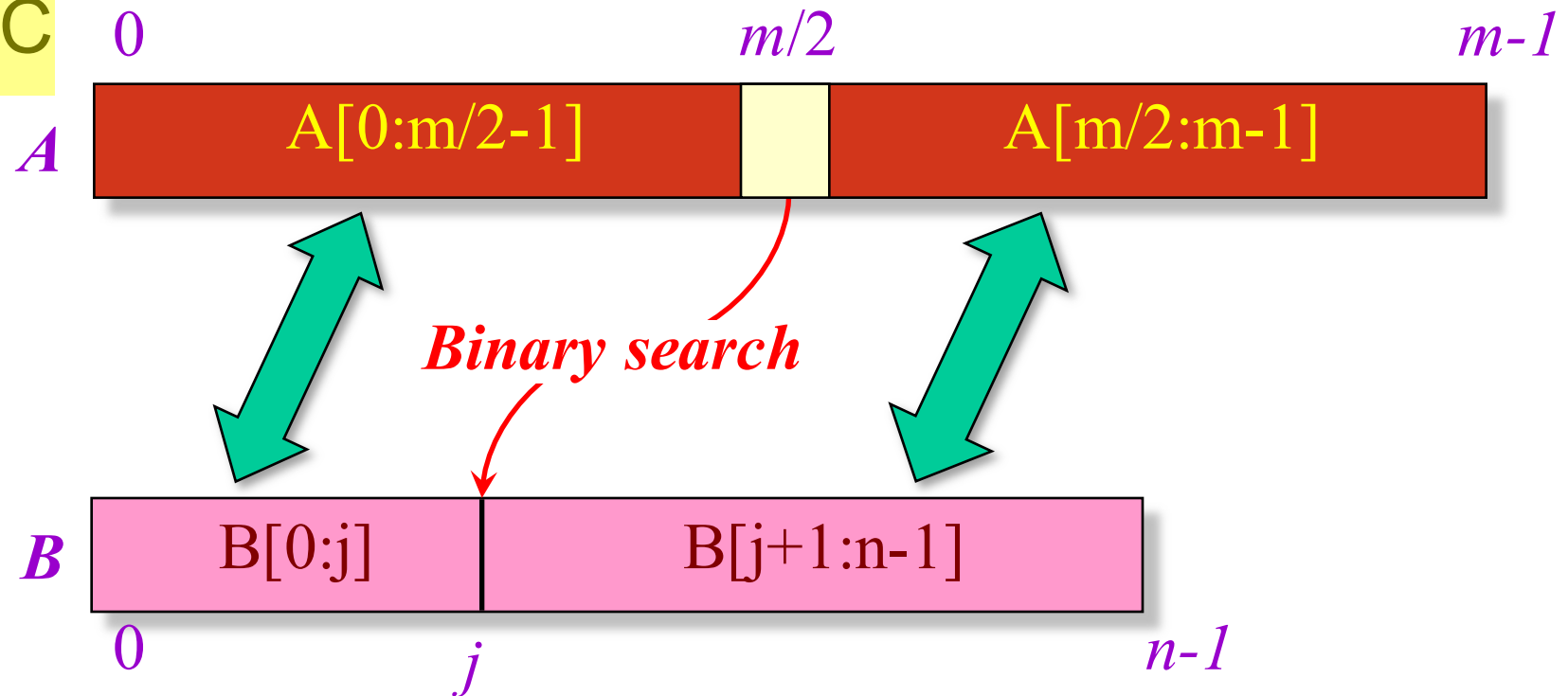
- Search for the $B[j]$ closest to, but not larger than, the median (assumes no duplicates)
- Thus, when we insert $A[m/2]$ between $B[0:j-1]$ & $B[j:n-1]$, the list remains sorted
- Recursively merge into a new array $C[]$
 - $C[0:j+m/2-2] \leftarrow \{A[0:m/2-1], B[0:j-1]\}$
 - $C[j+m/2:N] \leftarrow \{A[m/2:m-1], B[j:n-1]\}$
 - $C[j+m/2-1] \leftarrow A[m/2]$



Assuming that $B[j]$ holds the value that is closest to the median of A ($m/2$), which are true?

- A. All of $A[0:m/2-1]$ are smaller than all of $B[0:j]$
- B. All of $A[0:m/2-1]$ are smaller than all of $B[j+1:n-1]$
- C. All of $B[0:j-1]$ are smaller than all of $A[m/2:m-1]$
- D. A & B

E. B & C



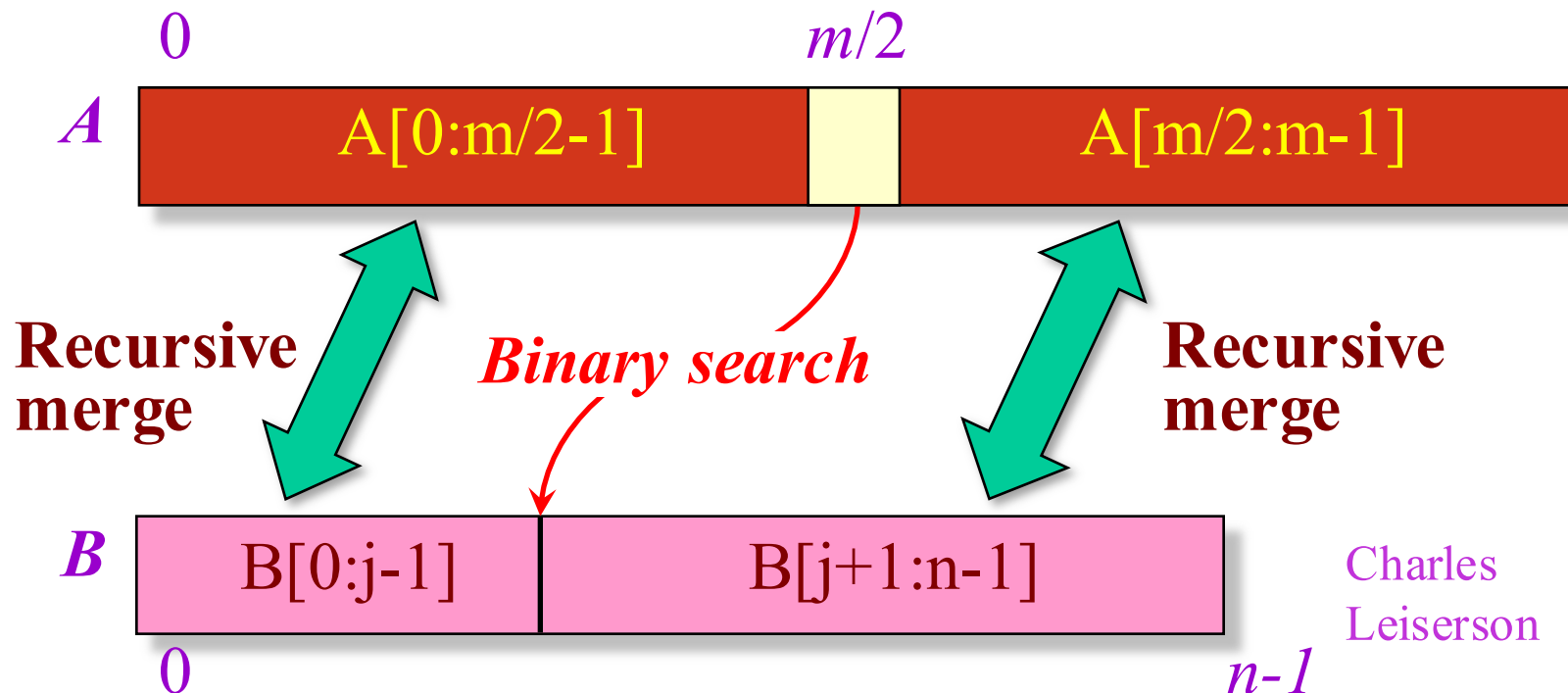
Charles Leiserson

Recursive Parallel Merge Performance

- If there are $N = m+n$ elements ($m \geq n$), then the larger of the merges can merge as many as $k \cdot N$ elements, $0 \leq k \leq 1$
- What is k and what is the worst case that establishes this bound?



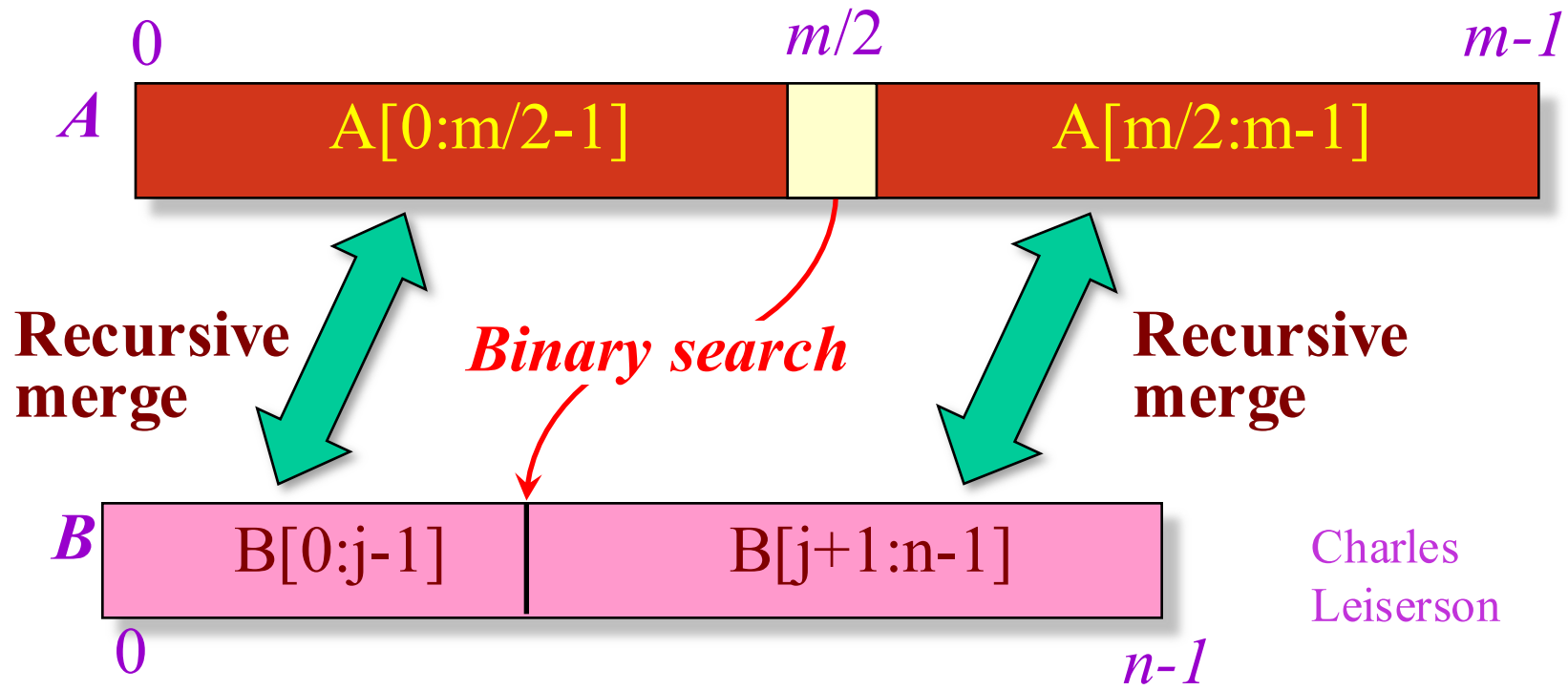
$m-1$



Charles
Leiserson

Recursive Parallel Merge Performance - II

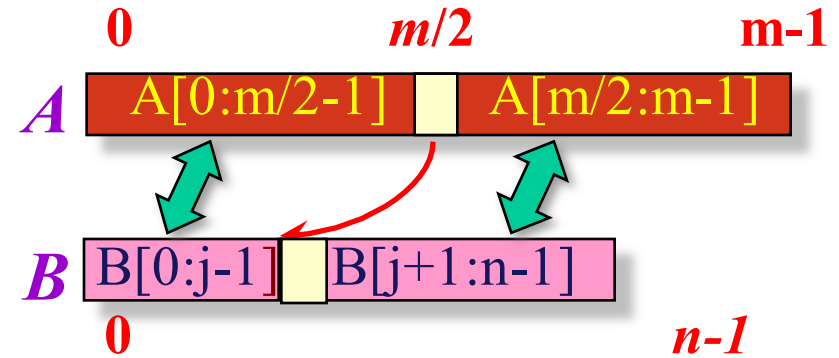
- If there are $N = m+n$ elements ($m \geq n$), then the larger of the recursive merges processes $\frac{3}{4}N$ elements
- What is the worst case that establishes this bound?
- Since $m \geq n$, $n = 2n/2 \leq (m+n)/2 = N/2$
- In the worst case, we merge $m/2$ elements of A with all of B



Recursive Parallel Merge Algorithm



```
void P_Merge(int *C, int *A, int *B, int m, int n) {  
    if (m < n) {  
        ... thread(P_Merge, C, B, A, n, m);  
    } else if (m + n is small enough) {  
        SerialMerge(C, A, B, m, n);  
    } else {  
        int m2 = m/2;  
        int j = BinarySearch(A[m2], B, n);  
        ... thread(P_Merge, C, A, B, m2, j);  
        ... thread(P_Merge, C+m2+j, A+m2, B+j, m-m2, nb-j);  
    }  
}
```



Charles Leiserson

Assignment #1

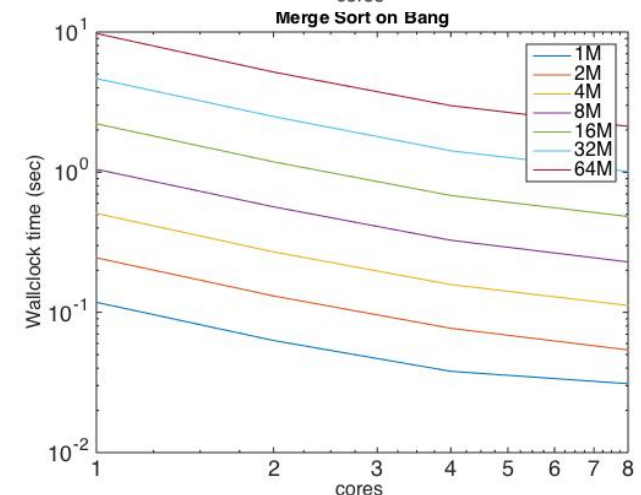
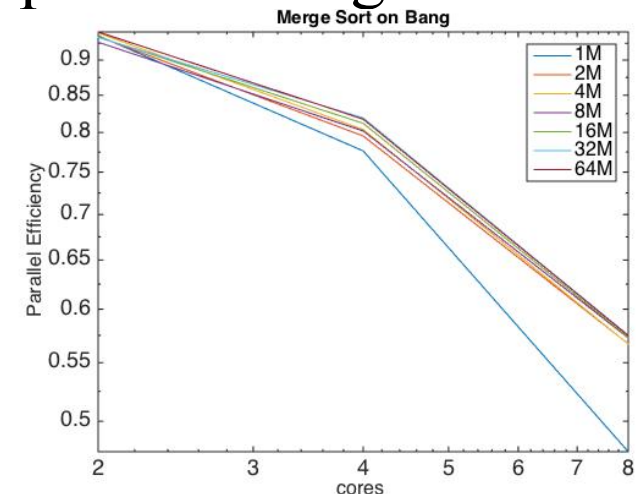
- Parallelize the provided serial merge sort code
- Once running correctly, and you have conducted a strong scaling study...
- Implement parallel merge and determine how much it helps
- Do the merges without recursion, just parallelize by a factor of 2. If time, do the merge recursively

Performance Programming tips

- Parallelism diminishes as we move up the recursion tree, so parallel merge will likely help much more at the higher levels (at the leaves, it's not possible to merge in parallel)
- Payoff from parallelizing the divide and conquer will likely exceed that of replacing serial merge by parallel merge



- Performance programming tips
 - ▶ Stop the recursion at a threshold value g
 - ▶ There is an optimal g , depends on P
 - $P = 1$: N
 - $P > 1$: $< N$
 - The parallel part of the divide and conquer will usually stop before we reach the g limit



Why are factors limiting the benefit of parallel merge, assuming the non-recursive merge?

- A. We get at most a factor of 2 speedup
- B. We move a lot of data relative to the work we do when merging
- C. Both

Today's lecture

- Merge Sort
- Barrier synchronization

Other kinds of data races

```
int64_t  global_sum = 0;
```

```
void sumIt(int TID) {  
    mtx.lock();  
    sum += (TID+1);  
    mtx.unlock();  
    if (TID == 0)  
        cout << "Sum of 1 : " << NT << " = " << sum <<  
        endl;  
}
```



```
% ./sumIt 5  
# threads: 5  
The sum of 1 to 5 is 1  
After join returns, the sum of 1 to 5 is: 15
```

Why do we have a race condition?

```
int64_t  global_sum = 0;
void sumIt(int TID) {
    mtx.lock();
    sum += (TID+1);
    mtx.unlock();
    if (TID == 0)
        cout << "Sum... ";
}
```

- A. Threads are able to print out the sum before all have contributed to it
- B. The critical section cannot fix this problem
- C. The critical section should be removed
- D. A & B
- E. A&C

Fixing the race - barrier synchronization

- The sum was reported incorrectly because it was possible for thread 0 to read the value before other threads got a chance to add their contribution (*true dependence*)
- The barrier repairs this defect: no thread can move past the barrier until all have arrived, and hence have contributed to the sum

```
int64_t  global_sum = 0;
void sumIt(int TID) {
    mtx.lock();
    sum += (TID+1);
    mtx.unlock();
    barrier();
    if (TID == 0)
        cout << "Sum . . . ";
}
```

```
% ./sumIt 5
# threads: 5
The sum of 1 to 5 is 15
```



Barrier synchronization



[wikipedia](#)



[www.shutterstock.com - 37859410](#)



[theknightsky racing.wordpress.com](#)



[www.galleryofchampions.com](#)

Today's lecture

- Merge Sort
- Barrier synchronization
- **An application of barrier synchronization**

Compare and exchange sorts

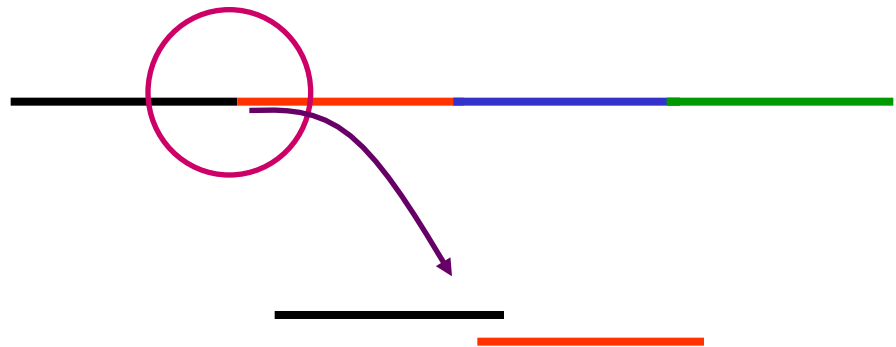
- Simplest sort, AKA bubble sort
- The fundamental operation is compare-exchange
- **Compare-exchange(a[j] , a[j+1])**
 - Swaps arguments if they are in decreasing order: (7,4) → (4, 7)
 - Satisfies the post-condition that $a[j] \leq a[j+1]$
 - Returns **false** if a swap was made

```
for i = 1 to N-2 do
  done = true;
  for j = 0 to N-i-1 do // Compare-exchange(a[j] , a[j+1])
    if (a[j] > a[j+1]) { a[j] ↔ a[j+1];
                        done=false; }
  end do
  if (done) break;
end do
```

Loop carried dependencies

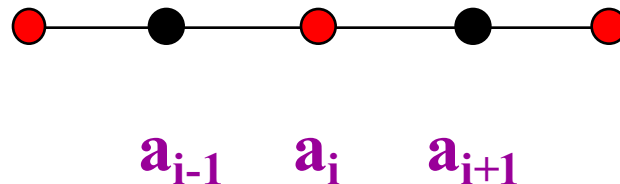
- We cannot parallelize bubble sort owing to the *loop carried dependence* in the inner loop
- The value of $a[j]$ computed in iteration j *depends* on the $a[i]$ computed in iterations $0, 1, \dots, j-1$

```
for i = 1 to N-2 do
  done = true;
  for j = 0 to N-i-1 do
    done = Compare-exchange( $a[j]$  ,  $a[j+1]$ )
  end do
  if (done) break;
end do
```

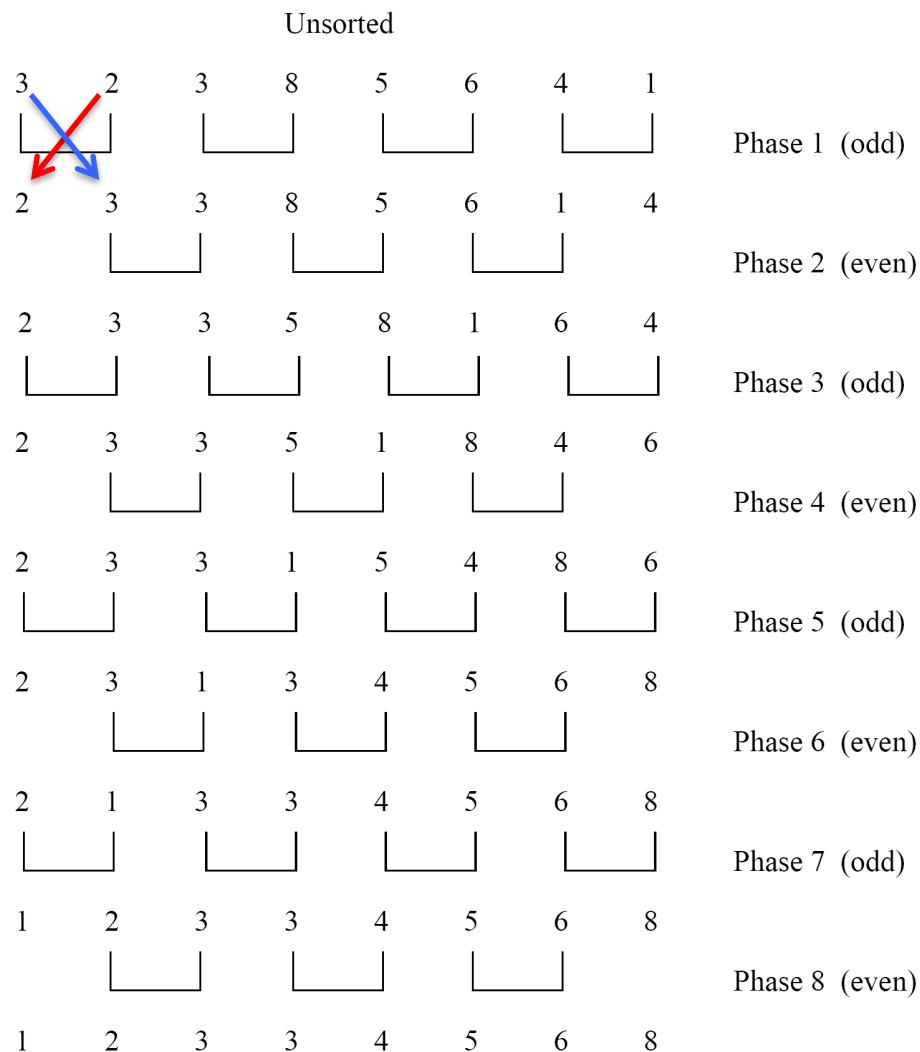


Odd/Even sort

- If we re-order the comparisons we can parallelize the algorithm
 - number the points as even and odd
 - alternate between sorting the odd and even points
- This algorithm parallelizes since there are no loop carried dependences
- All the odd (even) points are decoupled



Odd/Even sort in action

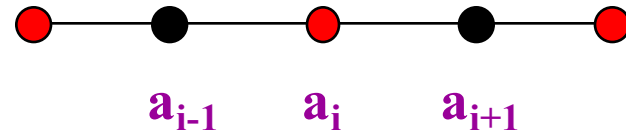


Introduction to Parallel Computing, Grama et al, 2nd Ed.

The algorithm

```
for i = 0 to N-2 do
  done = true;
  for j = 0 to N-1 by 2 do          // Even
    done &= Compare-exchange(a[j] , a[j+1]);
  end do
```

```
  for j = 1 to N-1 by 2 do          // Odd
    done &= Compare-exchange(a[j] , a[j+1]);
  end do
  if (done) break;
```



```
end do
```

```
// Bubble sort
for i = 1 to N-1 do
  done = true;
  for j = 0 to N-i-1 do
    done = Compare-Exchange(a[j] , a[j+1])
  end do
  if (done) break;
end do
```

What costs does odd/even sort add to the serial code?

A. More memory accesses

B. More comparisons

C. Both A& B

Odd/Even Sort Code

- Where do we need synchronization?



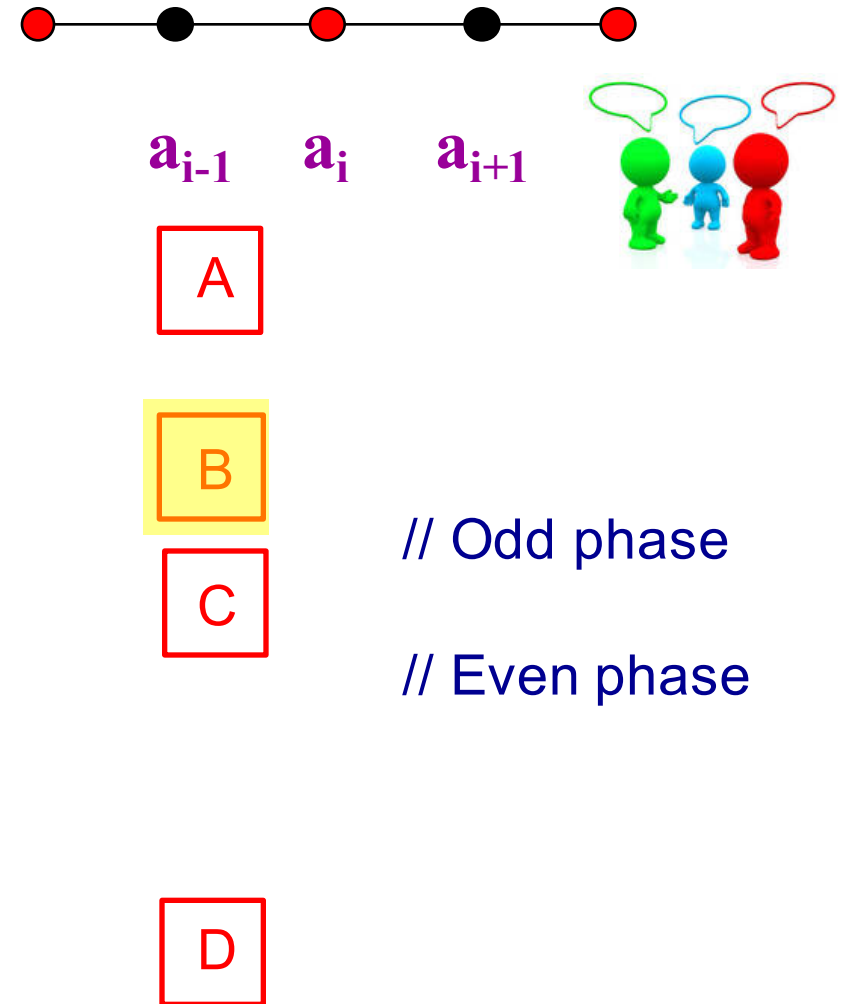
```
(1) Global bool AllDone;  
(2) int OE = lo % 2;  
(3) for (s = 0; s < MaxIter; s++) {  
(4)     int done = Sweep(Keys, OE, lo, hi); /* Odd phase */  
(5)     done &= Sweep(Keys, 1-OE, lo, hi); /* Even phase */  
  
(6)     AllDone &= done;  
  
(7)     if (AllDone)  
(8)         break;  
  
(9) } // End For
```

```
bool Sweep(int *Keys, int OE, int lo, int hi){  
    int Hi = hi;  
    if (TID == (NT-1))  
        Hi --;  
    bool myDone = true;  
    for (int i = OE+lo; i <= Hi; i+=2) {  
        if (Keys[i] > Keys[i+1]){  
            Keys[i] ↔ Keys[i+1];  
            myDone = false;  
        }  
    }  
    return myDone ;  
}
```

Which barrier synchronization points can we remove?

```
Global bool AllDone;
int OE = lo % 2;
for (s = 0; s < MaxIter; s++) {
    barr.sync();
    if (!TID)
        AllDone = true;
    barr.sync();
    int done = Sweep(Keys, OE, lo, hi);
    barr.sync();
    done &= Sweep(Keys, 1-OE, lo, hi);
    mtx.lock();
    AllDone &= done;
    mtx.unlock();
    barr.sync();

    if (allDone)
        break;
}
```



Building a linear time barrier with locks

```
class barrier {  
    int count, _NT;  
    mutex arrival, mutex departure;  
  
public:  
    Barrier(int NT=2): arrival(UNLOCKED),  
        departure(LOCKED), count(0), _NT(NT)  
    {};  
  
    void bsync( ){  
        arrival.lock( );  
        if (++count < NT)  
            arrival.unlock( );  
        else  
            departure.unlock( );  
        departure.lock( );  
        if (--count > 0)  
            departure.unlock( );  
        else  
            arrival.unlock( );  
    }  
};
```