Parallel Algorithms

Yihan Sun

Your feedback is welcome!

- **Today is the last lecture with new knowledge, we are almost done!**
  - Most of you are doing very well so far, congrats!

- **Your feedback is very helpful for improving the course (and hence help future UCR CS students)**
  - Provide your feedback on iEval
  - Finish our survey: https://forms.gle/m6Z698VremJkvCCt7
    - We want to know how you like the course/topics/OH/assignments

- **If you finish both by Dec 2 (the last lecture), you get 2 bonus candies**
- **Both are anonymous, and to do so you attach the screenshots after you submit**

How are candies counted and used?

- **Bonus for programming homework: up to 10**
- **Bonus for written homework: up to 7**
- **Midterm exam & midterm Challenge: up to 6**
- **UCRPC: up to 5**
- **Class participation: up to 9**
  - Class participation (Nov 16 and today): up to 2
  - Discussion participation (week 4): up to 1
  - In-lecture participation (Q&A): up to 3
  - Online (campuswire) participation (Q&A): up to 3
- **Providing feedback: 2 candies**

How are candies counted and used?

- **Total: likely to be up to 20**

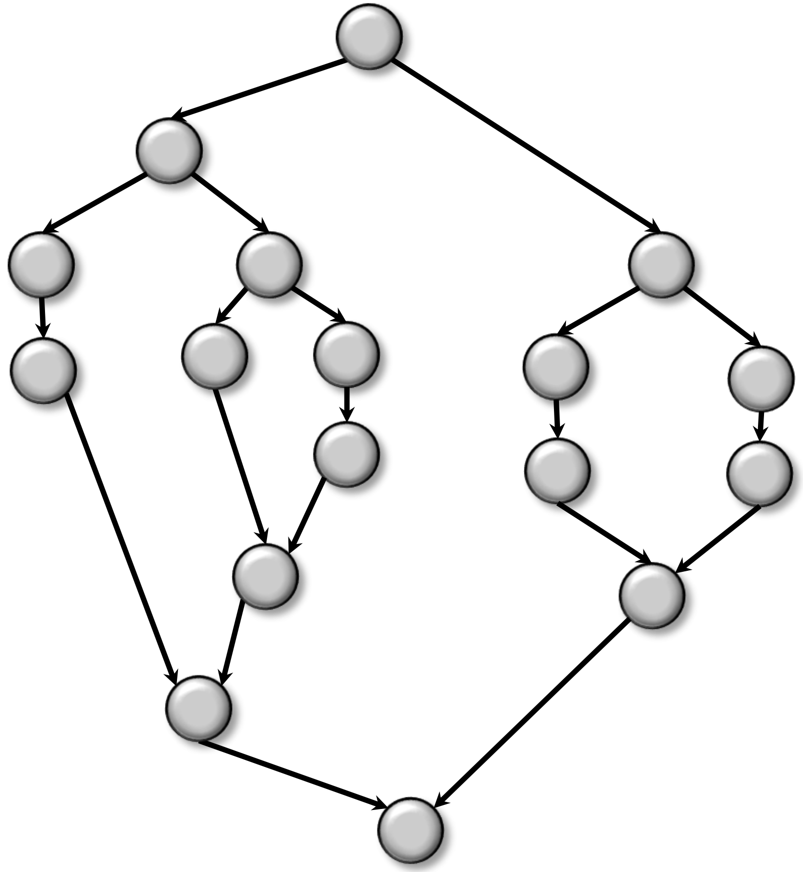
- **Converting to your class participation**
  - If you get 1-3 candies, you are likely to get 1-3 points to your final grade
  - If you get 4-20 candies, we will convert it to 4-10 points, based on some piecewise function

- **Also, those with top 10 most candies and/or top 5 in each session will receive our prizes**
  - Will be given to you on Dec 2's lecture, and you need to come to the classroom and pick up your prizes

- **Lastly: Written HW 5 is due on Nov 30 (Tuesday), one day earlier than usual**
    - We need to finalize your grade by the end of the next week

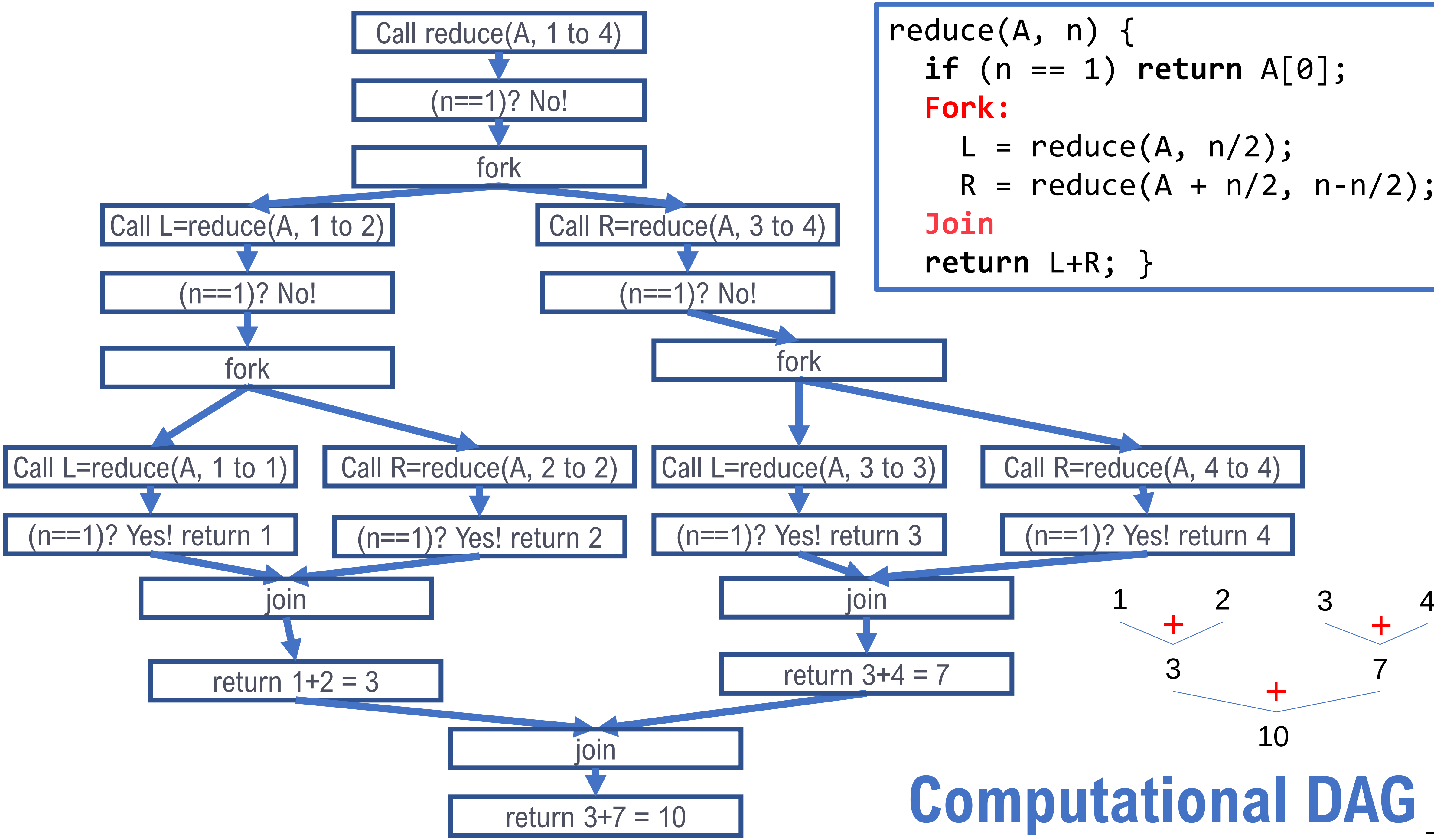
Last Lecture

- **Dynamic multithreading and Scheduler:**
  - Dynamic multithreading: only specify parallelism / dependency for tasks
  - Scheduler: Help you map your parallel tasks to processors
- **Fork-join**
  - Fork: two tasks run in parallel
  - Join: after all forked threads finish, synchronize them
- **Computational DAG**
  - Draw all operations in a DAG
  - A -> B means that operation B must be done after A



```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

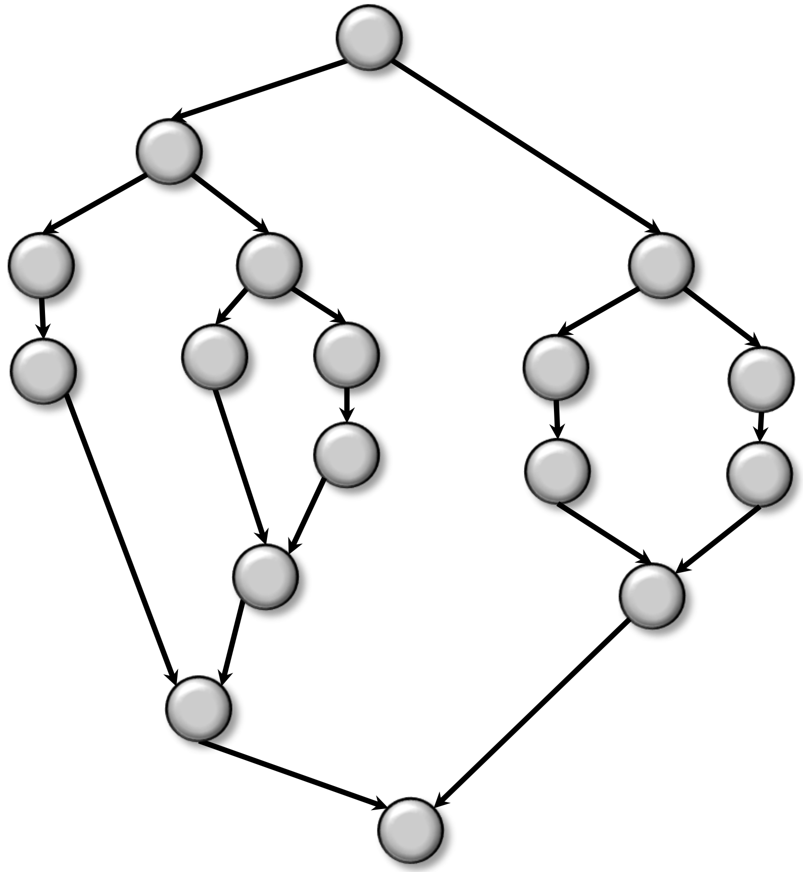
```
reduce(A, n) {
  if (n == 1) return A[0];
  Fork:
    L = reduce(A, n/2);
    R = reduce(A + n/2, n-n/2);
  Join
  return L+R; }
```

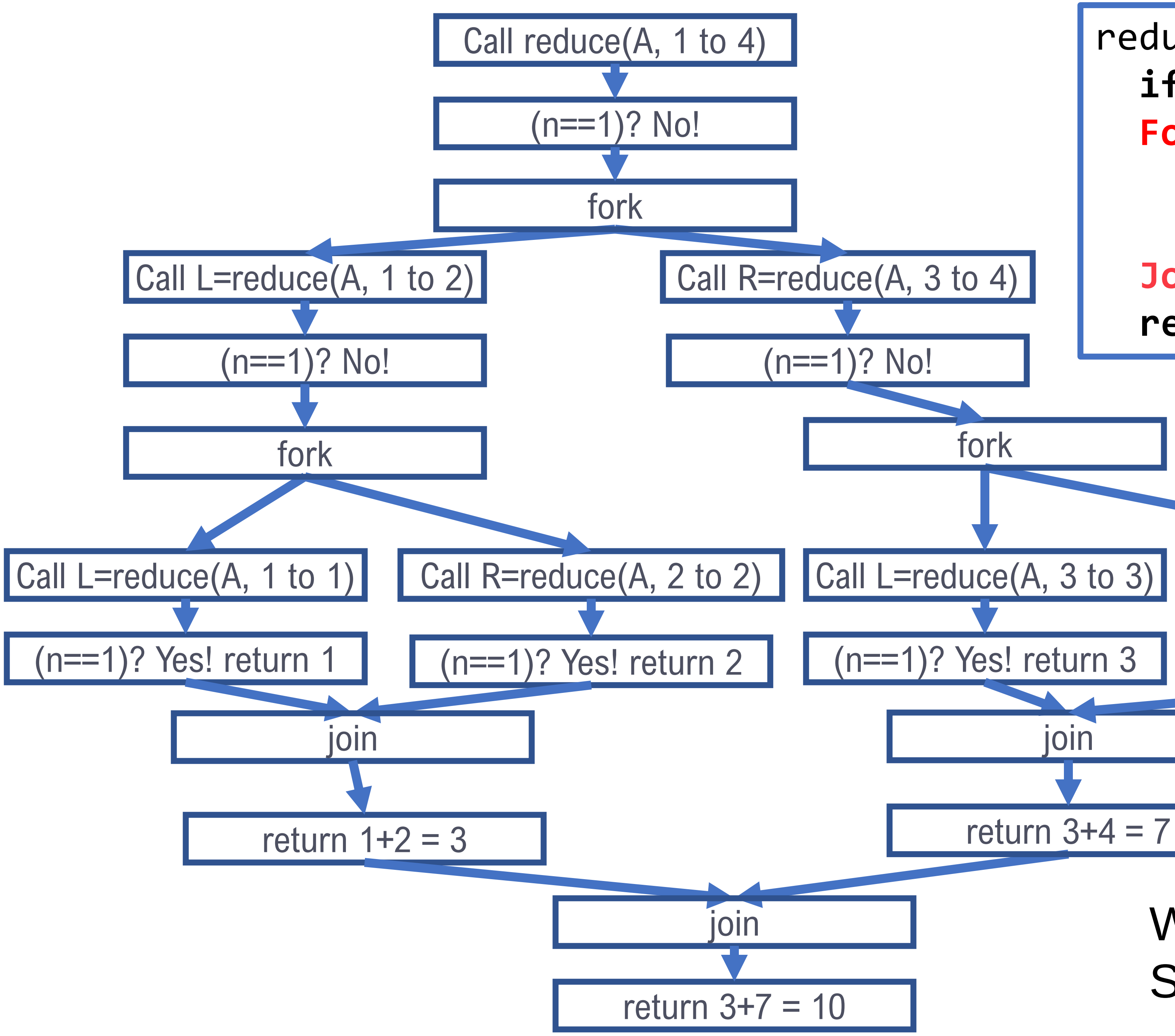


Computational DAG

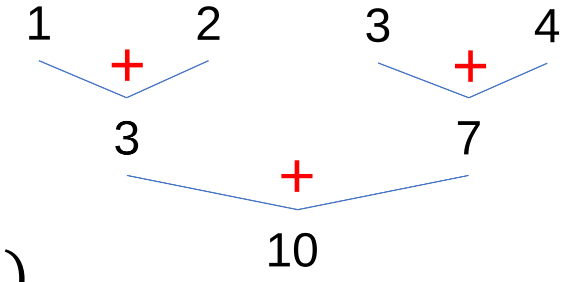
Last Lecture

- **Work-span**
  - Work: total number of operations, sequential complexity
  - Span (depth): the longest chain in the computational DAG






```
reduce(A, n) {
  if (n == 1) return A[0];
  Fork:
    L = reduce(A, n/2);
    R = reduce(A + n/2, n-n/2);
  Join
  return L+R; }
```



Work $O(n)$
Span $O(\log n)$

Computational DAG

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

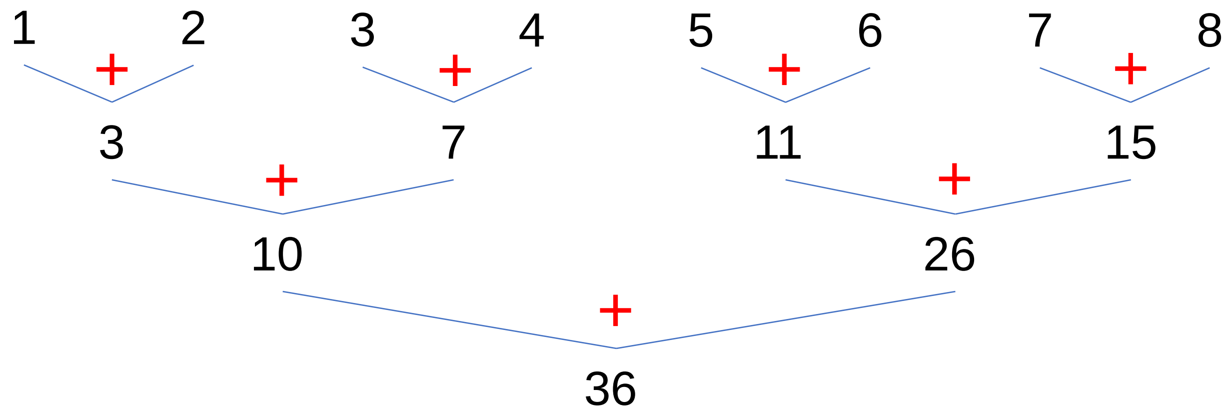
$$\boldsymbol{W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1)}$$

$$\boldsymbol{S(n) = \Theta(1) + S\left(\frac{n}{2}\right)}$$

Work $O(n)$
Span $O(\log n)$

Another way to implement reduce

- $W(n) = \Theta(n) + W\left(\frac{n}{2}\right)$
- $\Rightarrow W(n) = \Theta(n)$



- $S(n) = \Theta(\log n) + S\left(\frac{n}{2}\right)$
- $\Rightarrow S(n) = \Theta(\log^2 n)$

```
reduce(A, n) {
  if (n == 1) return A[0];
  if (n is odd) n=n+1;
  parallel_for i=1 to n/2
    B[i]=A[2i]+A[2i+1];
  return reduce(B, n/2); }
```

Needs $\log n$ rounds to spawn tasks!

How do work and span relate to the real execution and running time?

Scheduling a parallel algorithm

- A DAG with work W and span S can be executed using p processors in time $\boldsymbol{O\left(\frac{W}{p}+S\right)}$

- Both W and S matter!
- For small p, W is more important
- For large p, S is more important

Prefix Sum (Scan)

Prefix sum

In = 1 2 3 4 5 6 7 8

Out = 1 → 3 → 6 → 10 → 15 → 21 → 28 → 36

The most widely-used building block in parallel algorithm design

Two algorithms to implement a reduce

Can we use the same idea in *reduce* to compute prefix sum?

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

Divide-and-conquer:
Dealing with the left and right halves recursively

```
reduce(A, n) {
  if (n == 1) return A[0];
  if (n is odd) n=n+1;
  parallel_for i=1 to n/2
    B[i]=A[2i]+A[2i+1];
  return reduce(B, n/2); }
```

Decrease problem size:
Shrink the original size into a half

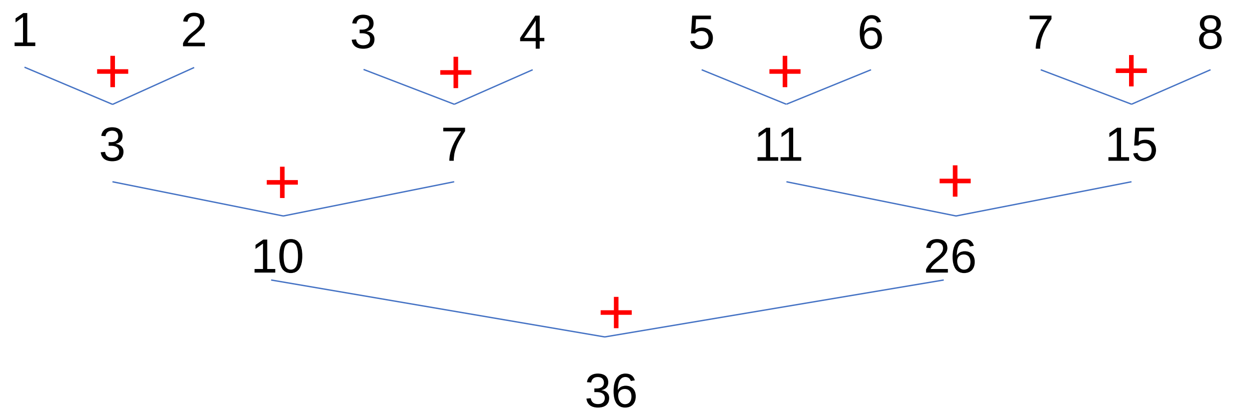
Decrease problem size

Can we use the same idea in *reduce* to compute prefix sum?

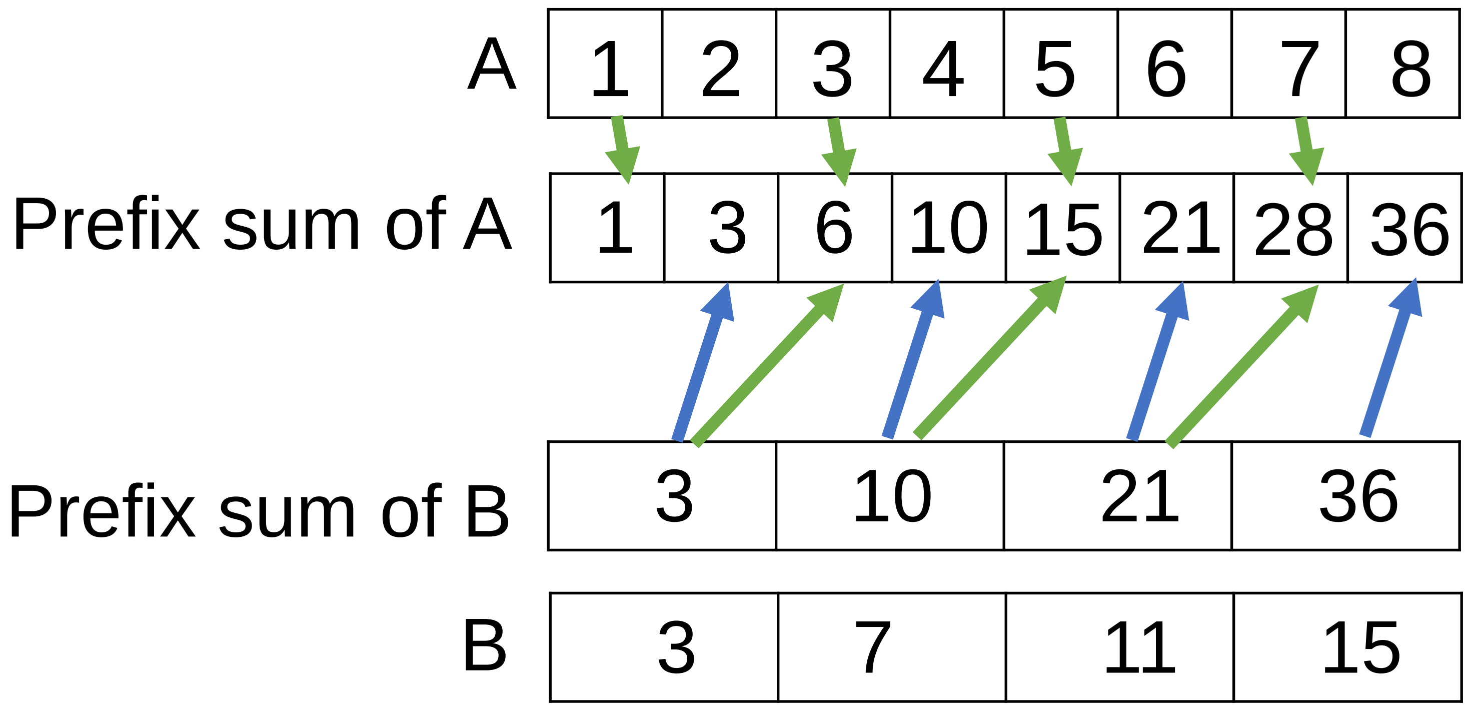
```
reduce(A, n) {
  if (n == 1) return A[0];
  if (n is odd) n=n+1;
  parallel_for i=1 to n/2
    B[i]=A[2i]+A[2i+1];
  return reduce(B, n/2); }
```

Decrease problem size:
Shrink the original size into a half

Prefix Sum

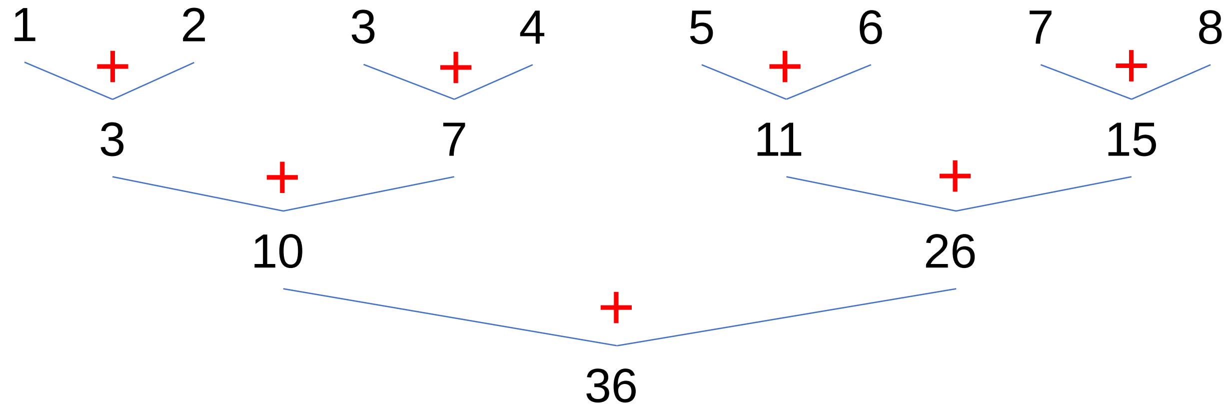



```
reduce(A, n) {
  if (n == 1) return A[0];
  if (n is odd) n=n+1;
  parallel_for i=1 to n/2
    B[i]=A[2i]+A[2i+1];
  return reduce(B, n/2); }
```



- Shrink the problem size into $\frac{n}{2}$, possibly in parallel
- Solve the same problem on $\frac{n}{2}$
- Convert the result of the subproblem to the final answer, possibly in parallel

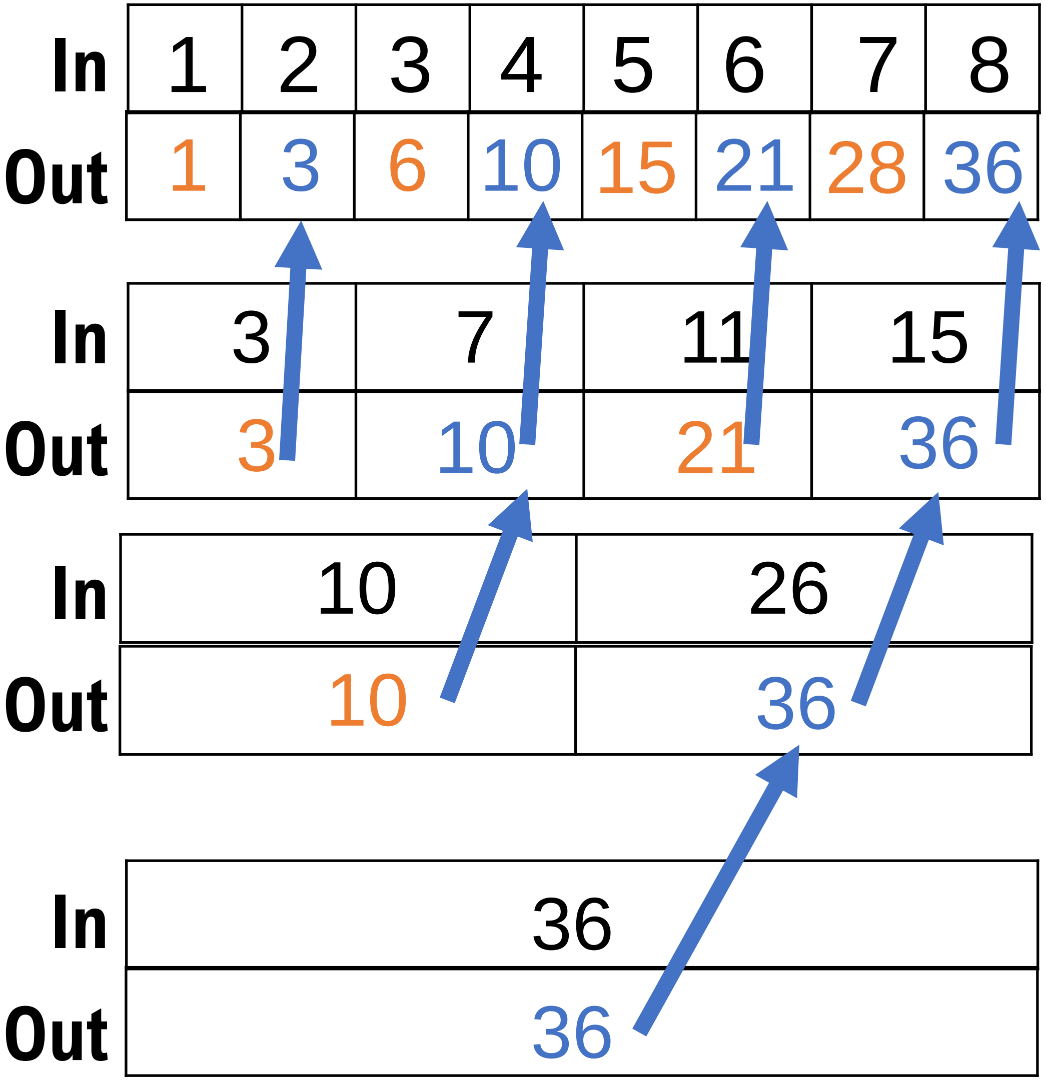
Prefix sum




```
Function Out = PrefixSum(In[1..n]) {
  if (n==1) Out[1] = In[1];
  parallel_for (i=1 to n/2)
    B[i] = In[2i-1]+In[2i]

  C = PrefixSum(B[1..n/2]);

  Out[1] = In[1];
  parallel_for (i=2 to n) {
     if (i%2) Out[i] = C[i/2];
     else Out[i] = C[i/2] + In[i];} }
```



$O(n)$ work

$O(\log^2 n)$ span

Another way to solve prefix sum

- **Based on the divide-and-conquer reduce algorithm, we can also design a prefix sum algorithm**
  - $O(n)$ work, $O(\log n)$ span
  - Slightly more complicated

Filtering / packing

Parallel filtering / packing

- **Given an array A of elements and a predicate function f, output an array B with elements in A that satisfy f**

$$f(x) = \begin{cases} true & if\ x\ is\ odd \\ false & if\ x\ is\ even \end{cases}$$

$A =$

4	2	9	3	6	5	7	11	10	8

$B =$

9	3	5	7	11

Parallel filtering / packing

- **Sequentially, we just read the array from left to right and put those satisfying f into an input array**
- **How can we know the length of B in parallel?**
  - Count the number of red elements – parallel reduce
  - $O(n)$ work and $O(\log n)$ span

$A =$

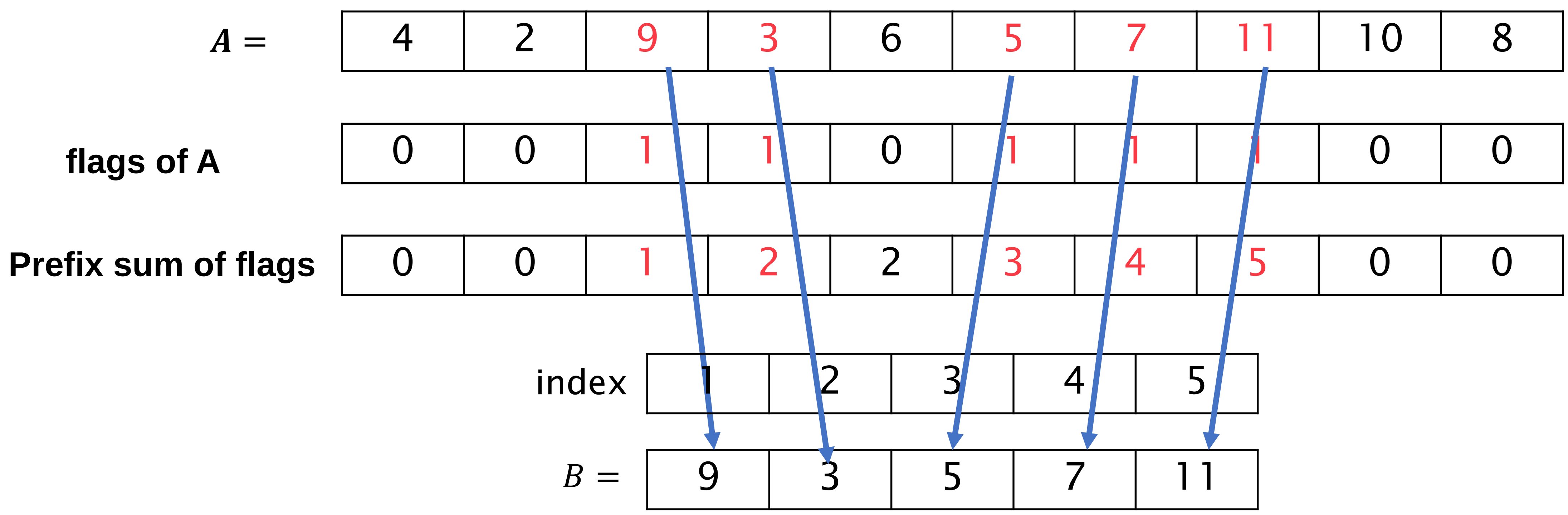
4	2	9	3	6	5	7	11	10	8

0	0	1	1	0	1	1	1	0	0

Parallel filtering / packing

- **How can we know where should 9 go?**
  - 9 is the first red element, 3 is the second, …

```
Filter(A, n, B, f) {
  new array flag[n], ps[n];
  parallel_for (i = 1 to n) {
    flag[i] = f(A[i]); }
  ps = prefix_sum(flag, n);
  parallel_for(i=1 to n) {
    if (f(A[i]))
      B[ps[i]] = A[i];
  } }
```



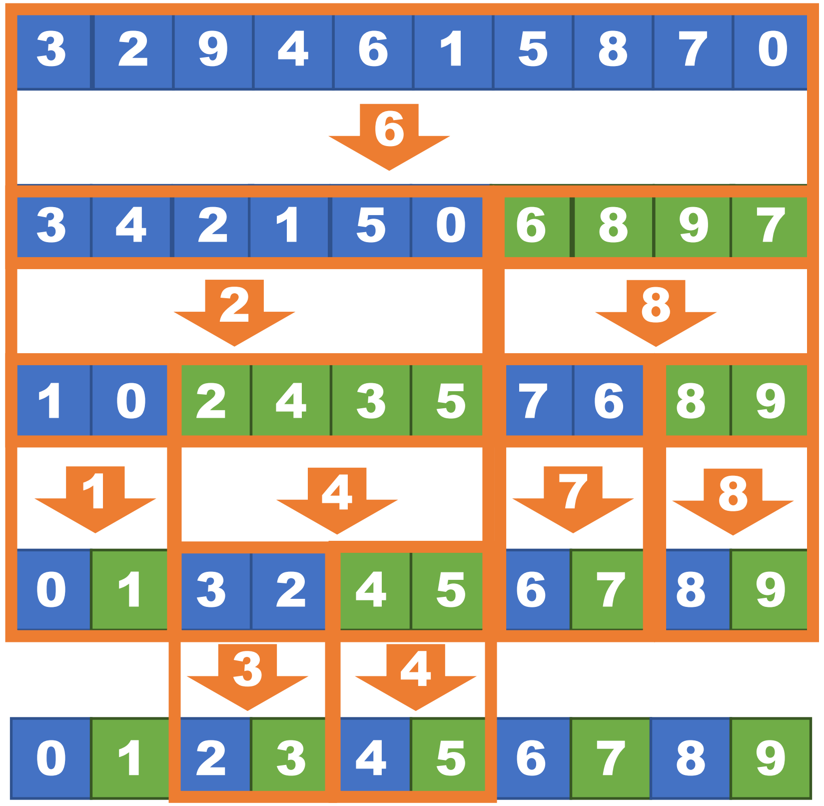
Parallel Filtering/packing

- $O(n)$ **work,**
- $O(\log^2 n)$ **span (if we use the scan algorithm above)**
- **Can be** $O(\log n)$ **span with a better scan algorithm**

Parallel Quicksort

Sequential quicksort algorithm

- Find a random pivot p in the array (e.g., the middle one)
- Put all elements in A that are $< p$ on the left, and all elements in A that are $\geq p$ on the right* ("partition")

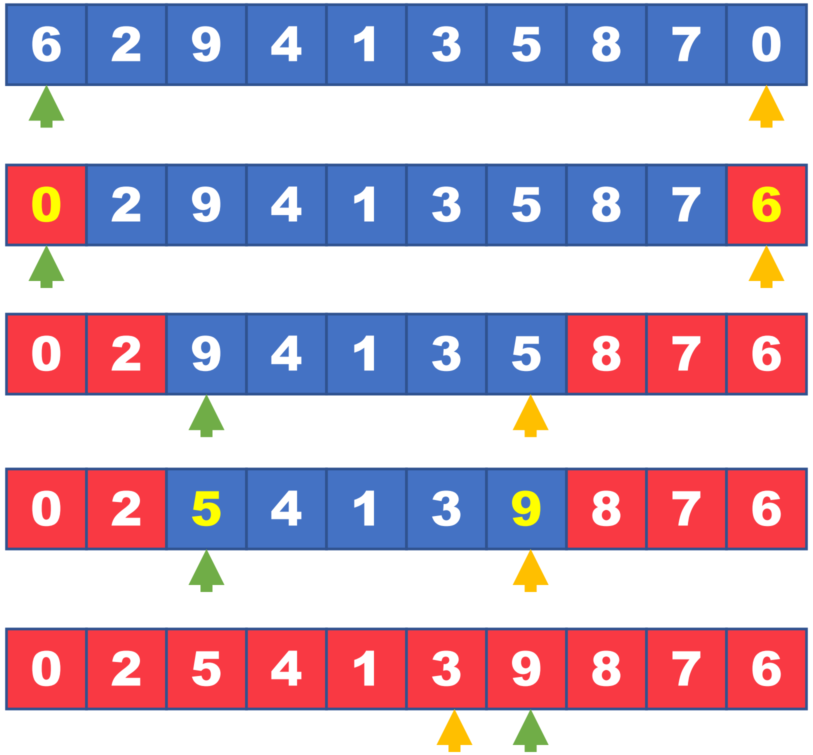


- How to do partition?

*: Usually we need to find >, < and =. Here for simplicity we assume keys are distinct so that doesn't make much difference

Sequential sorting algorithms: quicksort

- How to move elements around? (using 6 as a pivot)




```
Partition(A, n, x) {
  i = 0; j = n-1;
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j++;
    if (i < j) swap A[i] and A[j];
    i++; j--;
  }
}
```

- $O(n)$ time for one round

Sequential quicksort

- Use a pivot and partition the array into two parts
- Sort each of them recursively

```
qsort(A, n) {
  t = partition(A, A[random()]);
  qsort(A, t);
  qsort(A+t, n-t);
}
```

Parallel quicksort

- Use a pivot and partition the array into two parts
- Sort each of them recursively, **in parallel**

```
qsort(A, n) {
  t = partition(A, A[random()]);
  In parallel:
    qsort(A, t);
    qsort(A+t, n-t);
}
```

Parallel quick sort

- **The partitioning algorithm costs $O(n)$ time. So even if the problem is always perfectly partitioned**
  - $W(n) = 2W\left(\frac{n}{2}\right) + O(n)$
  - $S(n) = S\left(\frac{n}{2}\right) + O(n)$
  - $S(n) = O(n)$?

```
qsort(A, n) {
  t = partition(A, A[random()]);
  In parallel:
    qsort(A, t);
    qsort(A+t, n-t);
}
```

- **Have to partition in parallel!**

Application of filter: partition in quicksort

- For an array A, move elements in A smaller than k to the left and those larger than k to the right

A	**6**	**2**	**9**	**4**	**1**	**3**	**5**	**8**	**7**	**0**
Possible output:	**2**	**4**	**1**	**3**	**5**	**0**	**6**	**9**	**8**	**7**

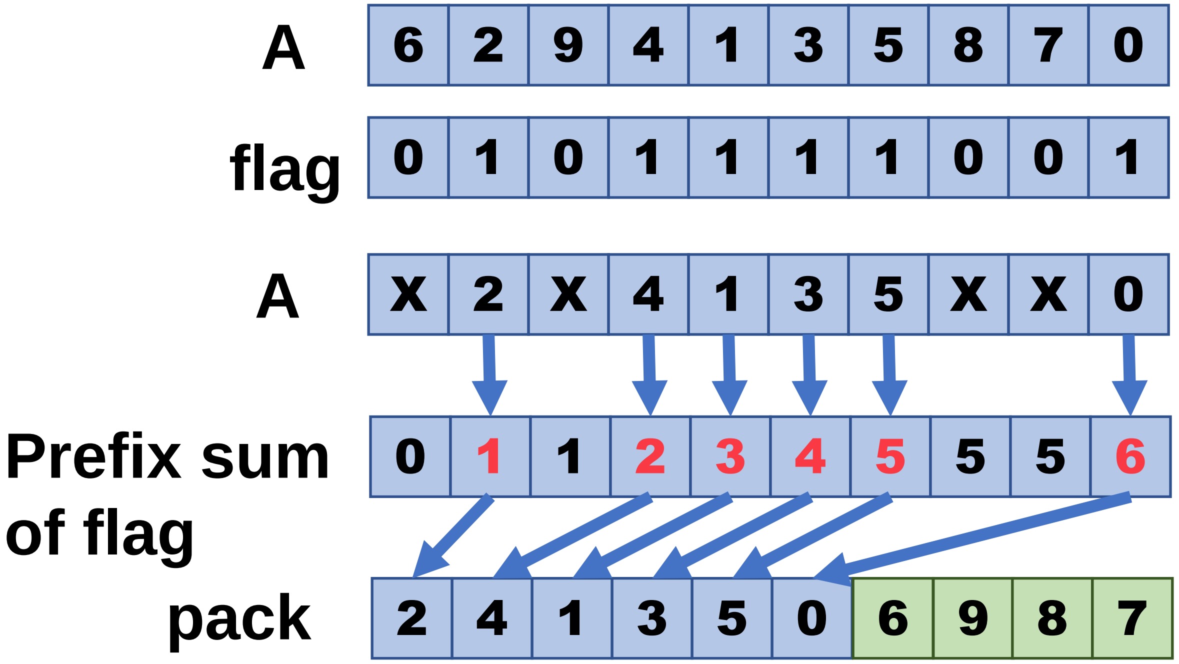
Partition by 6

- Left: all elements < 6
- Right: all elements ≥ 6

Using filter for partition

(Looking at the left part as an example)

using 6 as a pivot




```
Partition(A, n, k, B) {
  new array flag[n], ps[n];
  parallel_for (i = 1 to n) {
    flag[i] = (A[i]<k); }
  ps = scan(flag, n);
  parallel_for(i=1 to n) {
    if (A[i]<k)
      B[ps[i]] = A[i];
  }
  //symmetric for the right half
}
```

Predicator: if A[i]<pivot

Parallel quicksort

```
qsort(A, n) {
  t = parallel_partition(A, A[random()]);
  In parallel:
    qsort(A, t);
    qsort(A+t, n-t);
}
```

- **Work**
  - Exactly the same as sequential version
  - $O(n \log n)$ in expectation
- **Span**
  - $O(\log n) \times$ (#rounds of recursions) $\approx O(\log^2 n)$
    - $O(\log^2 n)$ whp.
- **Actually, quicksort is not the most commonly used sorting algorithm in parallel**
  - It is no longer in-place: lose the advantage

Summary

- **Parallel algorithms: identify / eliminate dependencies between operations**
- **“Time complexity” evaluation**
  - Work and span
  - Work: the total number of operations performed
  - Span: the longest dependency chain
- **Useful primitives:**
  - Reduce: the sum of all elements in an array
  - Scan: prefix sum
  - Filter/pack: extract elements with certain property
  - Can be used to implement parallel partition and quicksort

Summary

Useful ideas:

- **Divide-and-conquer: work on two/multiple subproblems in parallel**
  - Reduce
  - Quicksort
  - (Also parallel merge sort, but we need a parallel merging algorithm)

- **Decrease and conquer**
  - Decrease the size of the problem in parallel (polylog span), usually to $\frac{n}{2}$
  - Solve the smaller problem
  - Finish in $\log n$ rounds. So total span will still be polylog