Pipelining with Futures

Guy E. Blelloch Margaret Reid-Miller

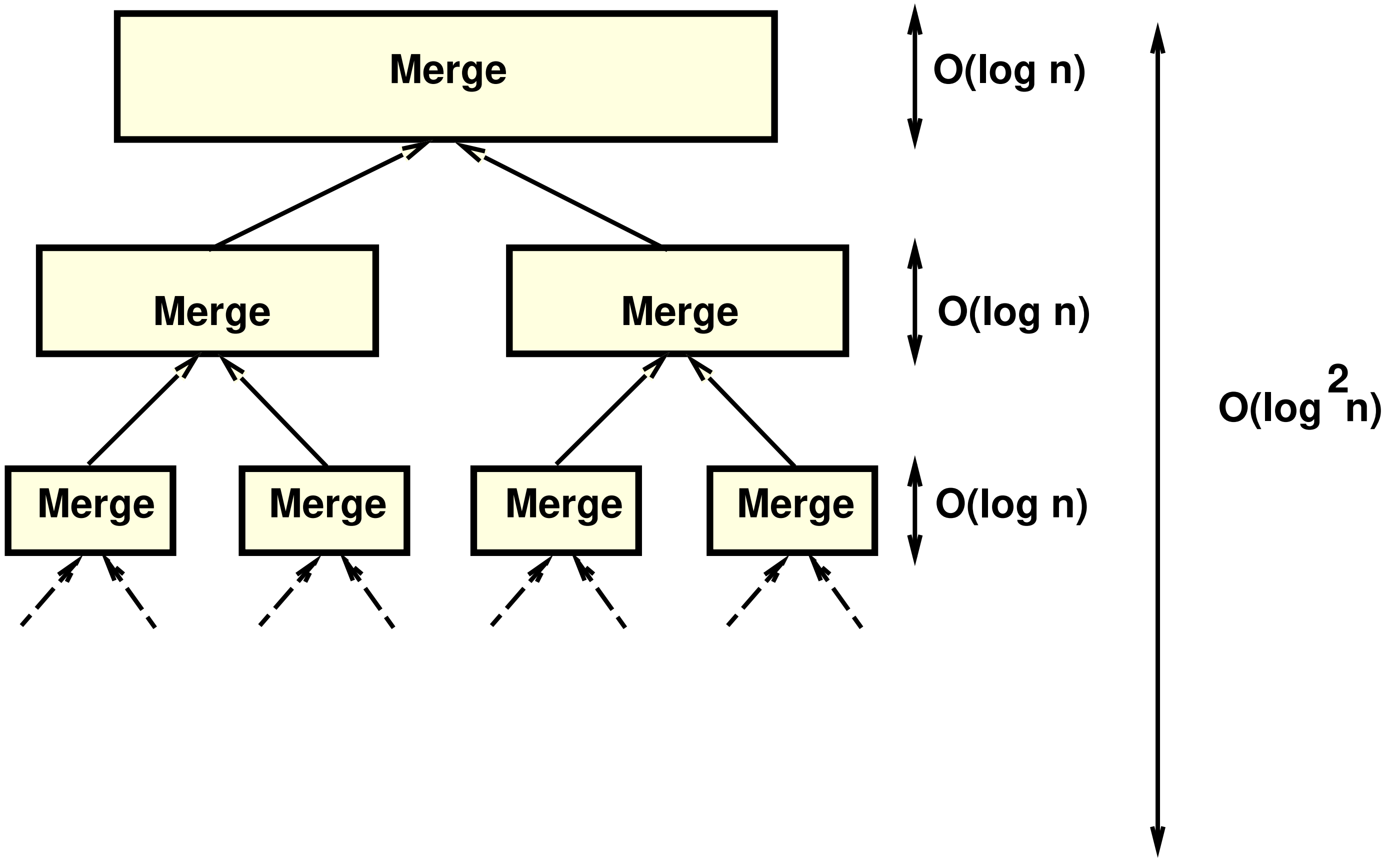
School of Computer Science
Carnegie Mellon University

SPAA'97
Newport, Rhode Island
22-25 June 1997

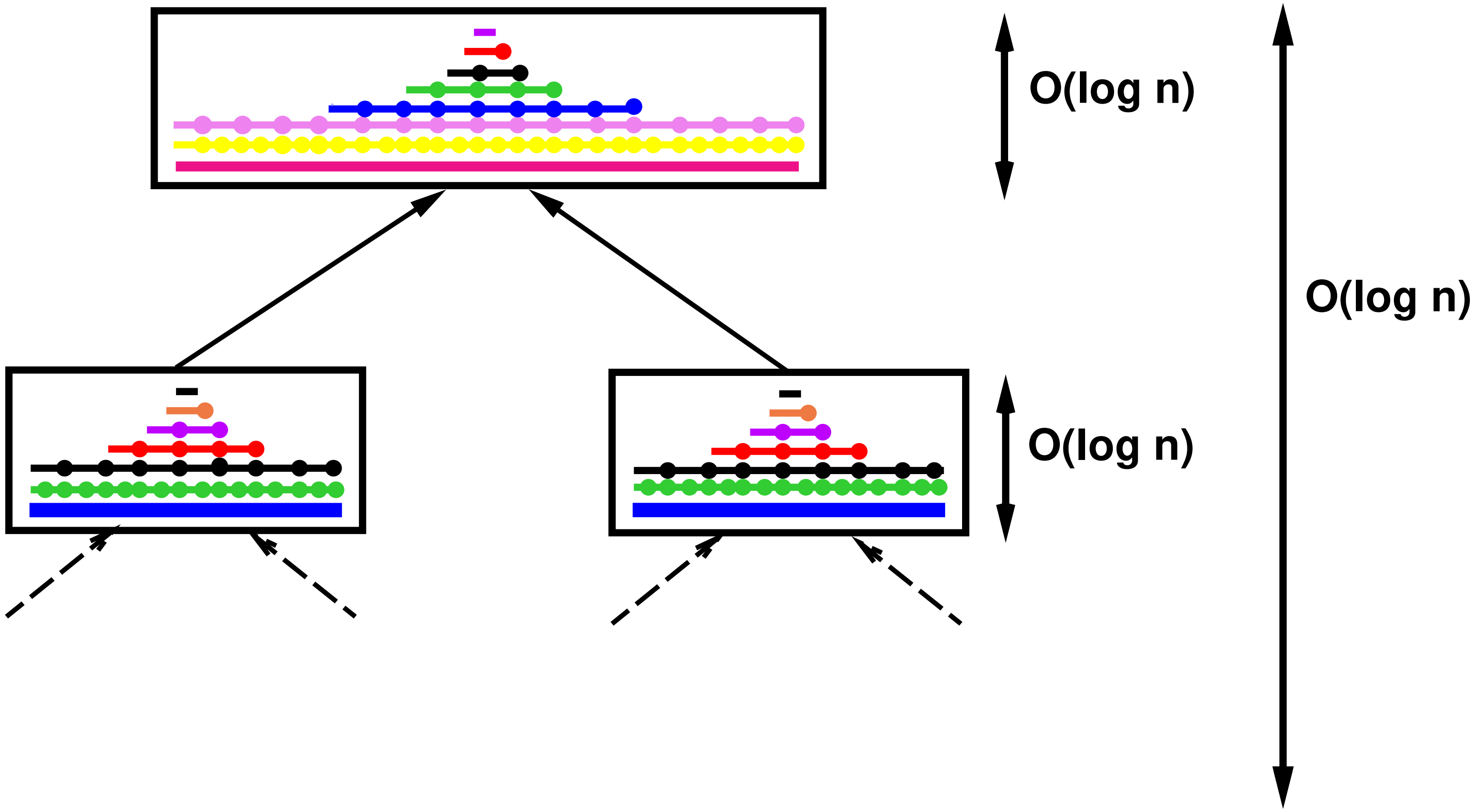
Outline

- What is pipelining?
- What are futures?
- Our model
- Applications
- Implementation
- Conclusions

Parallel Merge Sort



Cole's Pipelined Merge Sort



Why pipelining?

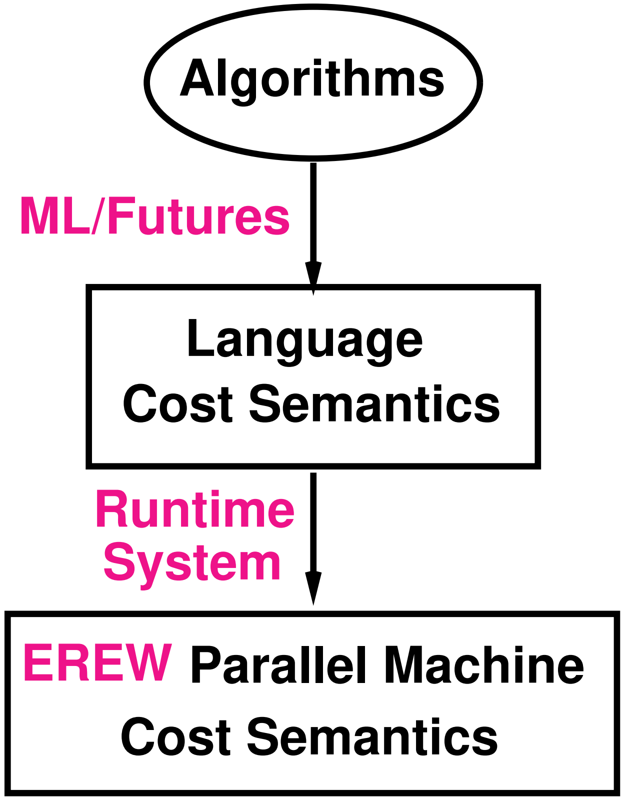
Reduces the depth of the computation:

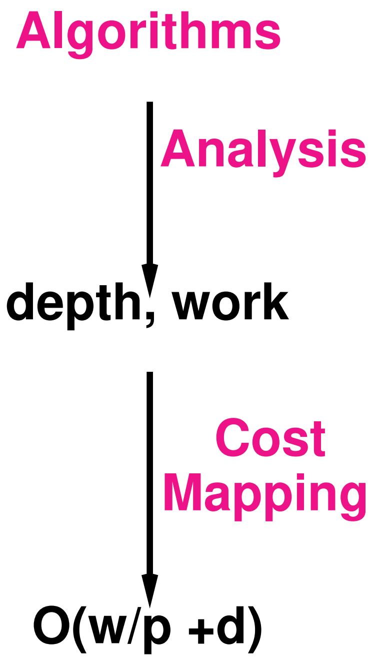
- 2-3 trees $O(\log n)$ (Paul et *al.* '83),
- merge sort $O(\log n)$ (Cole '88)
- cascading divide-and-conquer (Atallah et *al.* '89)
  e.g., 3-D maxima, visibility from a point

However,

- hard enough to program synchronous algorithm
- even harder to program asynchronous algorithms

Automatic Pipelining





Futures Language Construct

Conceptually a future

- forks a new thread to evaluate an expression and
- *immediately* returns a pointer to where result will be written.

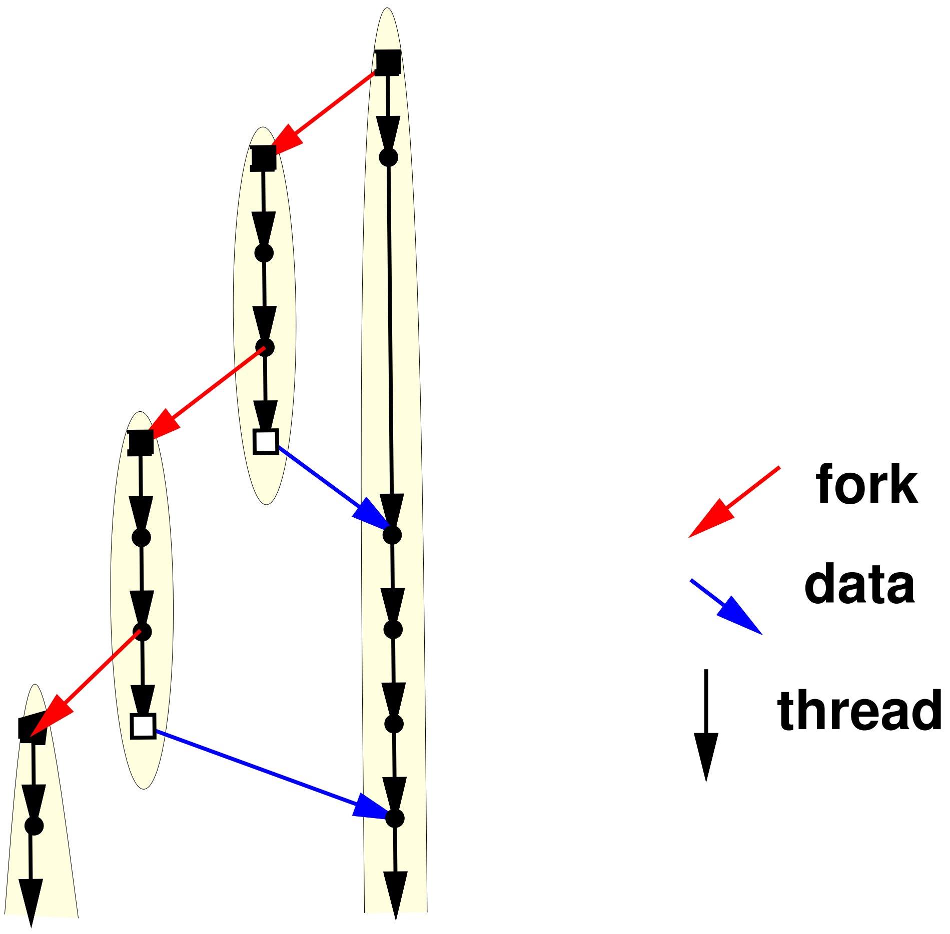
Used in functional languages to express

parallelism

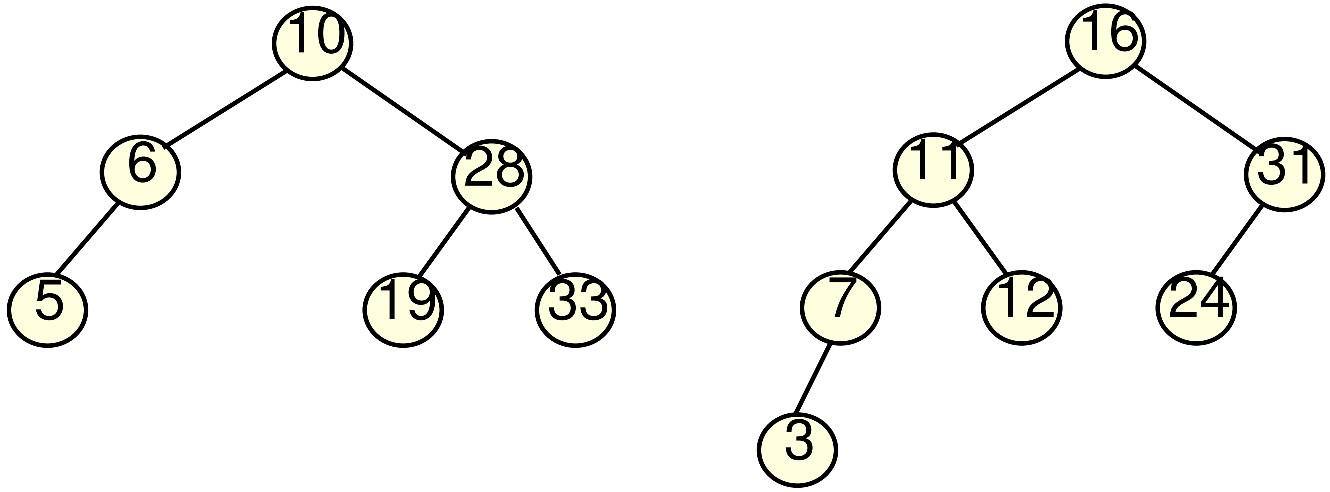
but also expresses

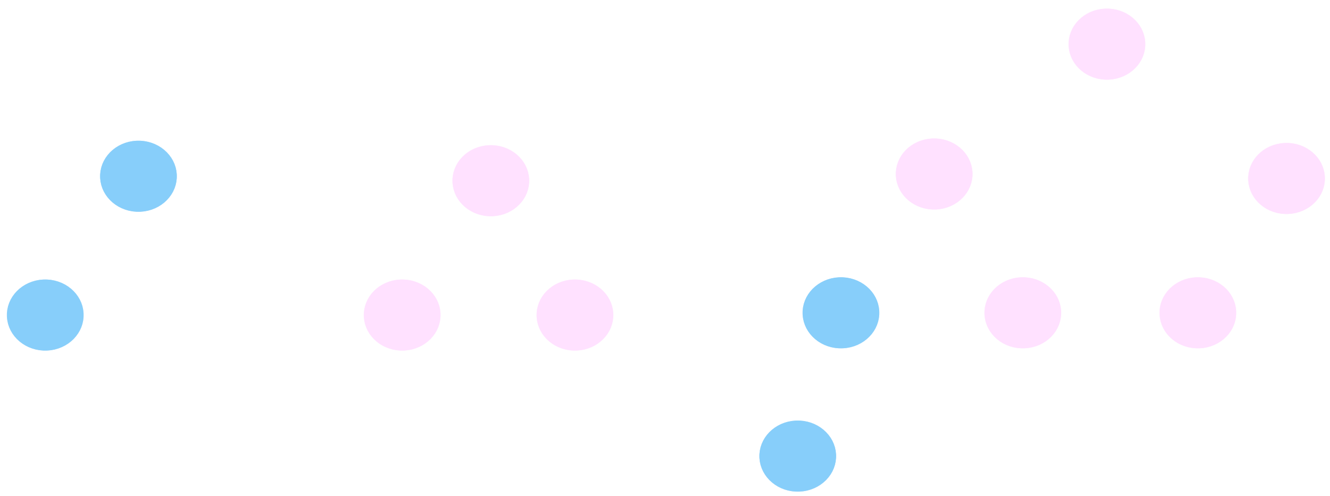
pipelining

DAG of Computations

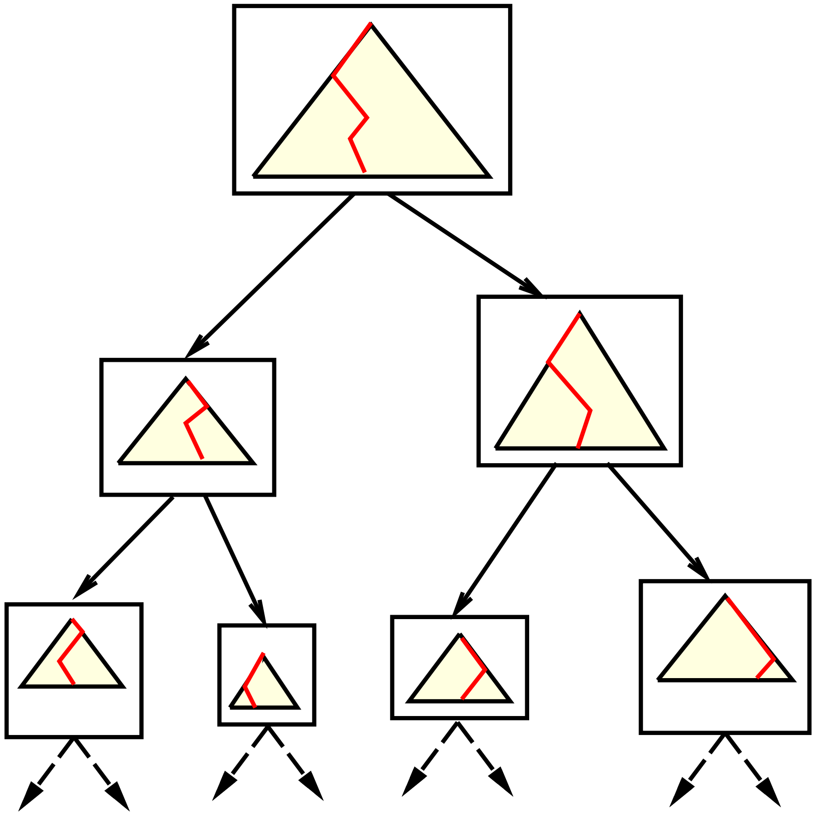


Merging Two Trees

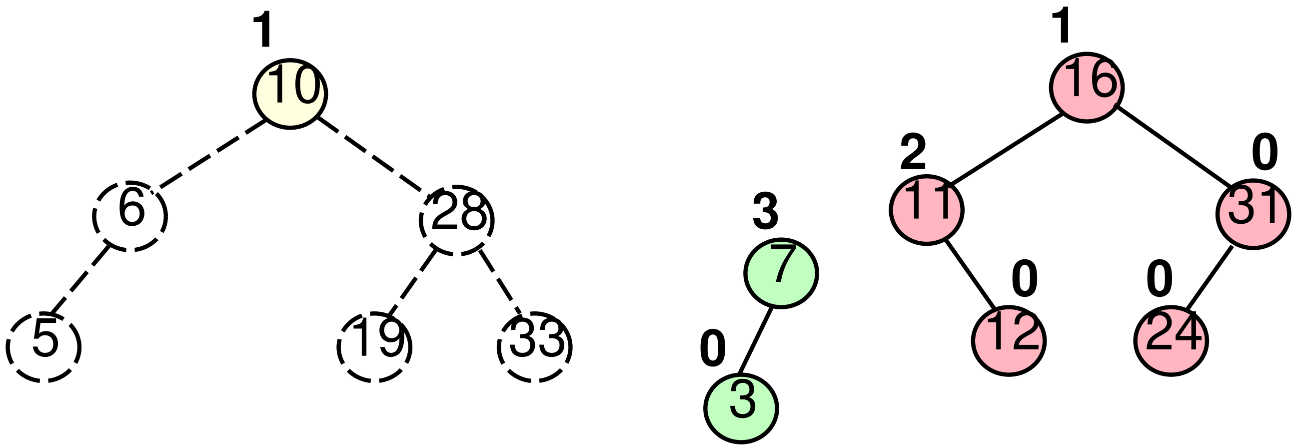


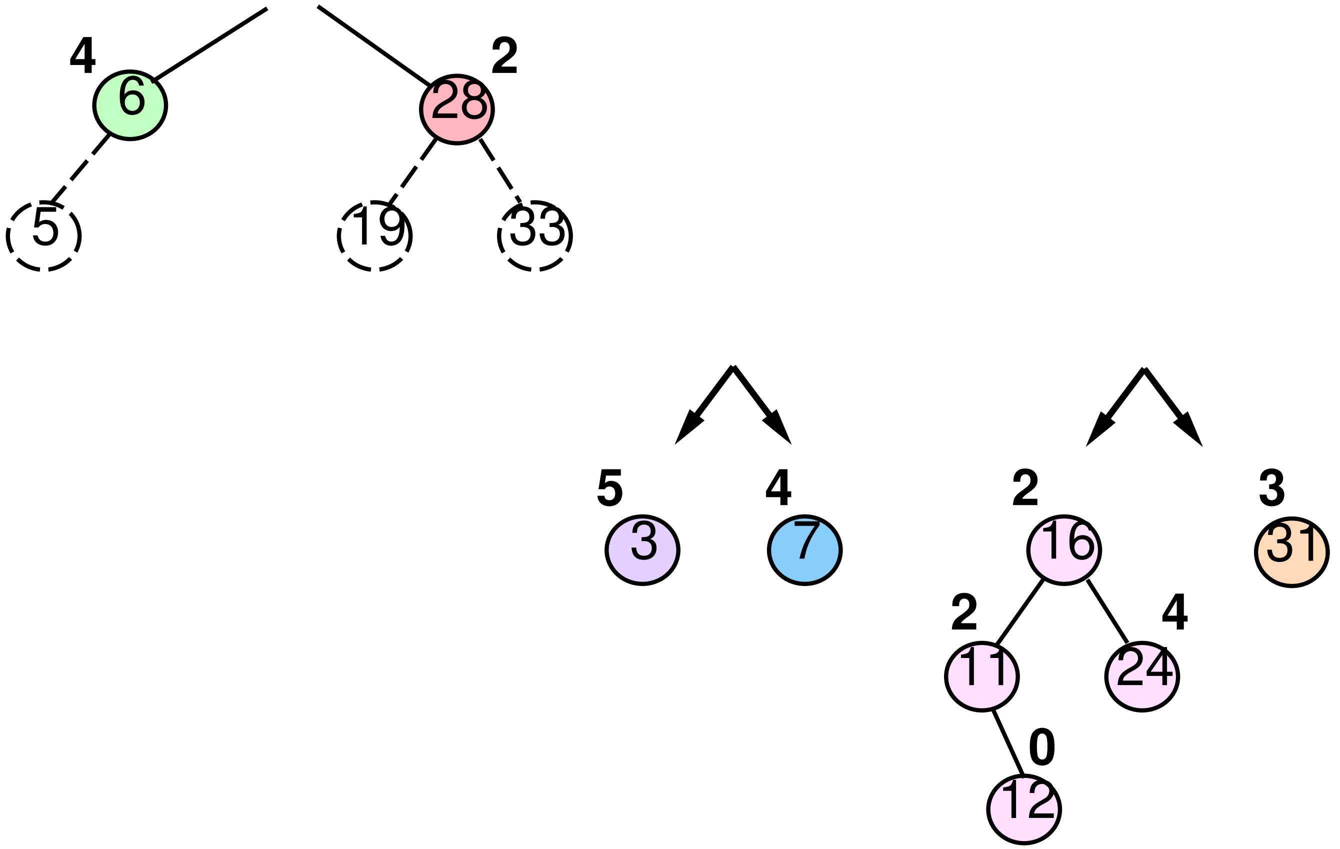


Parallel Merge



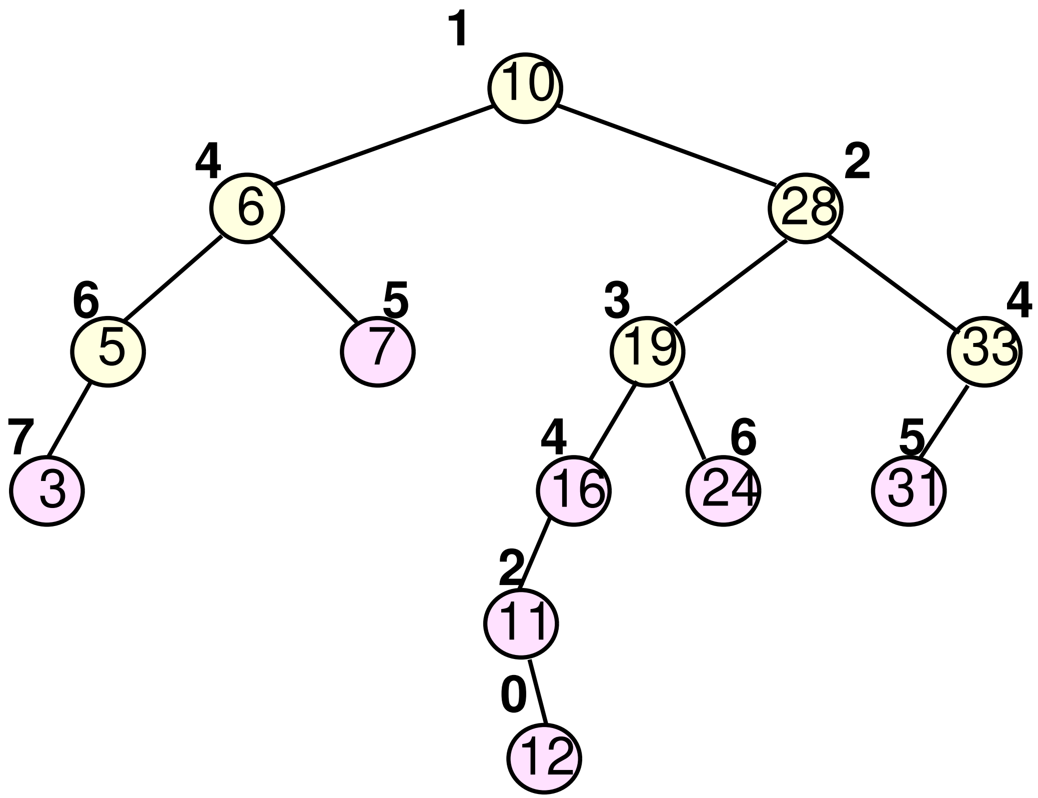
Time Stamps for Merge





4
6
2
28
5
19
33
5
3
4
7
2
16
3
31
2
11
4
24
0
12

Merged Trees



Code for Merge

```
datatype tree = node of int*tree*tree | empty;

fun split (s,empty) = (empty,empty)
  | split (s,node(v,L,R)) =
       if s < v then
          let val (L1,R1) = ?split(s,L)
          in (L1,node(v,R1,R)) end
       else
          let val (L1,R1) = ?split(s,R)
          in (node(v,L,L1),R1) end;

fun merge (empty,T2) = T2
  | merge (T1,empty) = T1
  | merge (node(v,L,R),T2) =
       let val (L2,R2) = ?split(v,T2)
       in node(v,?merge(L,L1), ?merge(R,R1)) end;
```

Analysis of Algorithms

Based on finding τ-values, each of which bounds the time stamps of all nodes in a tree.

Problem	**Depth** w/o pipelining	**Depth** w/ pipelining	**Work**
Merge	$O(d_1 d_2)$	$O(d_1 + d_2)$	$O(n d_2)$
Rebalance	$O(d_1 d_2)$	$O(d_1 + d_2)$	$O(n + m)$
Treap Merge	$O(\log n \log m)$	$O(\log n + \log m)$	$O(m \log(n/m))$
Treap Difference	$O(\log n \log m)$	$O(\log n + \log m)$	$O(m \log(n/m))$
2-6 Tree Insertion	$O(\log n \log m)$	$O(\log n + \log m)$	$O(m \log n)$

Runtime System

Implementation:

- maintains parallel queue of active threads
  (threads not waiting on a future)
- maintains (set of) threads waiting on each future
- activates threads waiting on a future when value is written
- schedules threads greedily

Guarantees the implementation is efficient.

Guaranteeing Exclusive Access: Linearity

Linear code: Only one reader of a variable

Drawback:

- must (deep) copy variables for each additional use
  $\longrightarrow$ can increase work or depth
  (but not for algorithms we present)

Gain:

- at most one thread waits on future
- all reads are exclusive

Mapping to a PRAM

In general

CRCW PRAM in $O(w/p + d \cdot T_f(p))$ and large constants,
where T_f time to fetch-and-add (Greiner and Blelloch '96)

Linearalized

EREW PRAM in $O(w/p + d \cdot T_s(p))$ and smaller constants,
where T_s time to scan

Previous Work

Algorithms: Paul et *al.* '83, Cole '88

Language-based cost models: Greiner and Blelloch '96, Blelloch and Greiner '96, Greiner '97

Futures: Friedman and Wise '78, Baker and Hewitt '77

Scheduling: Blumofe and Leiserson'93, '94, Blelloch et *al.* '95, '97

Linearity: Girard '87, Lafont '88

Conclusions

Implementation of automatic pipelining:

Futures – simple to code, asynchronous
Runtime system – load balances threads
– maintains the pipeline

Two phase analysis:

1. language-based cost model
2. cost to implement model
   on various machines—efficient

Four pipelined algorithms:

- Reduces depth by $O(\log n)$