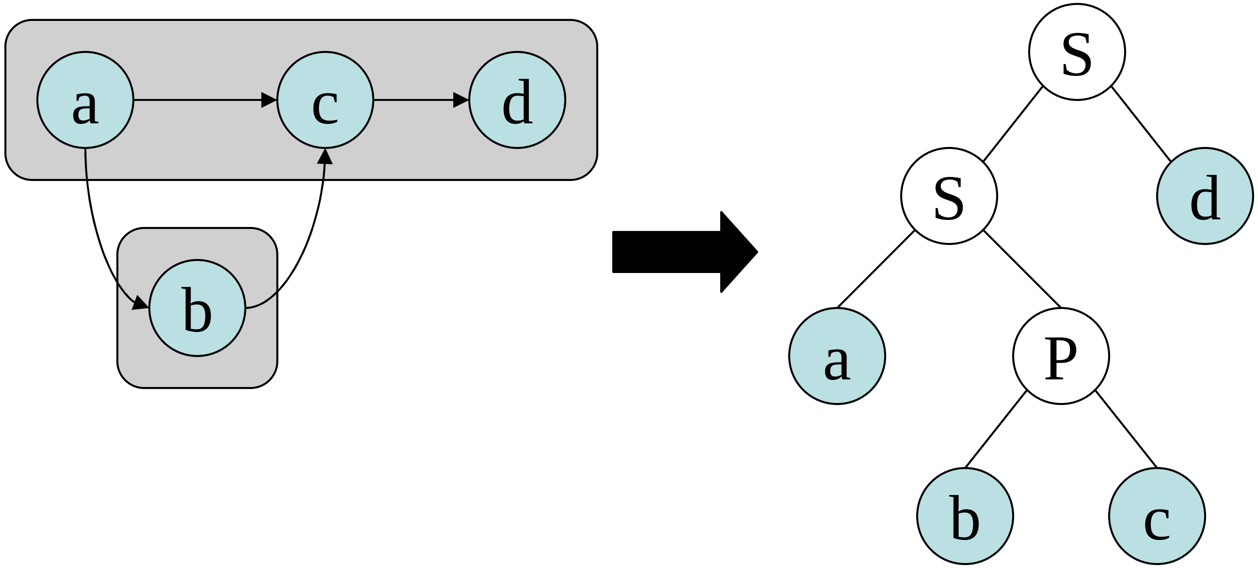
Maintaining SP Relationships Efficiently, on-the-fly

Jeremy Fineman

The Problem

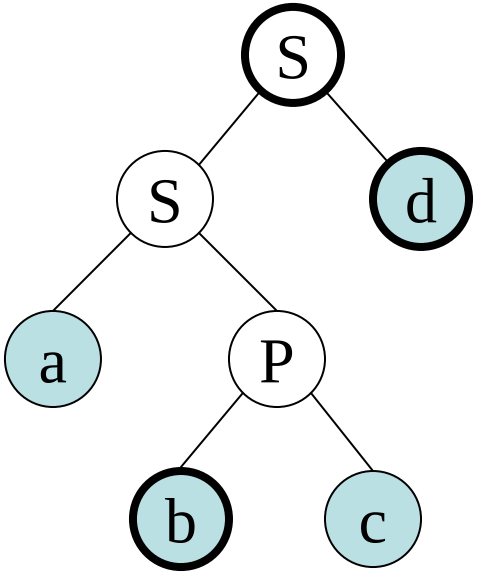
- Fork-Join (e.g. Cilk) programs have threads that operate either in series or logically parallel.
- Want to query relationship between two threads as the program runs.
- For example, Nondeterminator uses relationship between two threads as basis for determinacy race.

Parse Tree



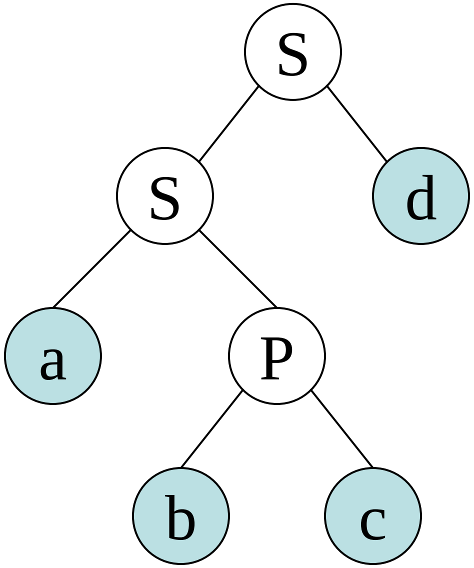
- Represent SP-DAG as a parse tree
- S nodes show series relationships
- P nodes are parallel relationships

Least Common Ancestor



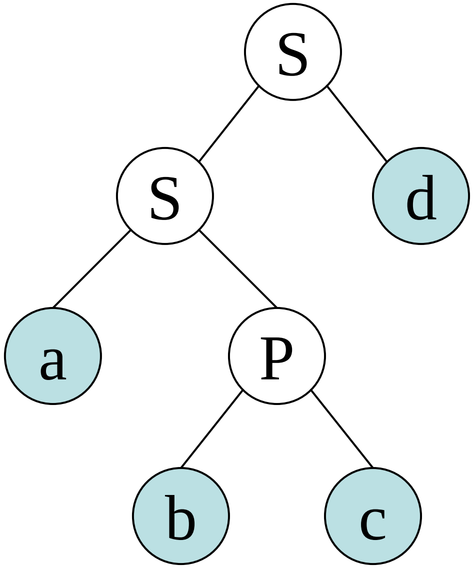
- SP-Bags uses LCA lookup.
- LCA of *b* and *d* is an S-node
  - So *b* and *d* are in series
- Cost is $\alpha(v,v)$ per query (in Nondeterminator)

Two Complementary Walks



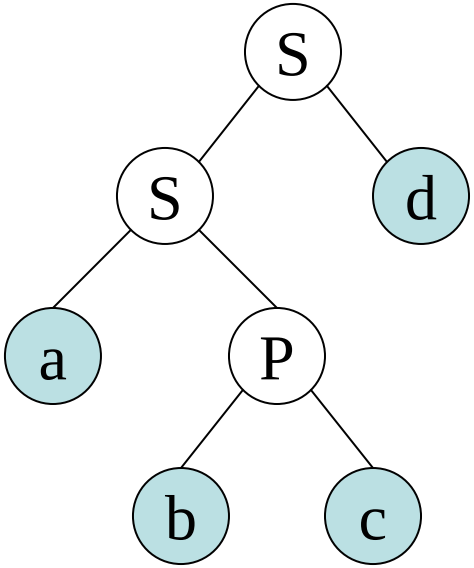
- At S-node, always walk left then right
- At P-node, can go left then right, or right then left

Two Complementary Walks



- Produce two orders of threads:
  - *a b c d*
  - *a c b d*
- Notice *b* || *c*, and orders differ between *b* and *c*.

Two Complementary Walks



- Claim: If e_1 precedes e_2 in one walk, and e_2 precedes e_1 in the other, then $e_1 \parallel e_2$.

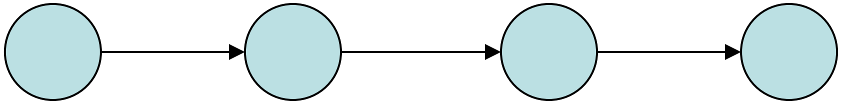
Maintaining both orders in a single tree walk

- Walk of tree represents execution of program.
  - Can execute program twice, but execution could be nondeterministic.
  - Instead, maintain both thread orderings on-the-fly, in a single pass.

Order Maintenance Problem

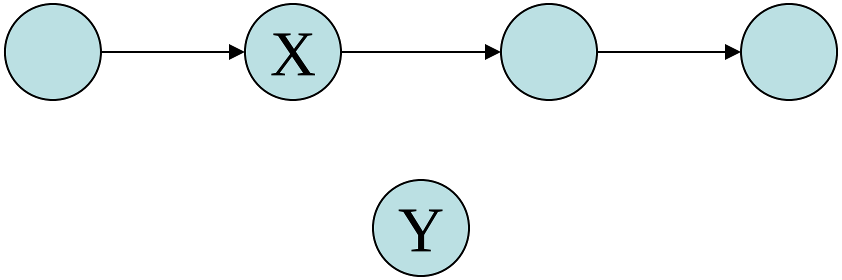
- We need a data structure which supports the following operations:
  - Insert(X,Y): Place Y after X in the list.
  - Precedes(X,Y): Does X come before Y?

Naïve Order Maintenance Structure



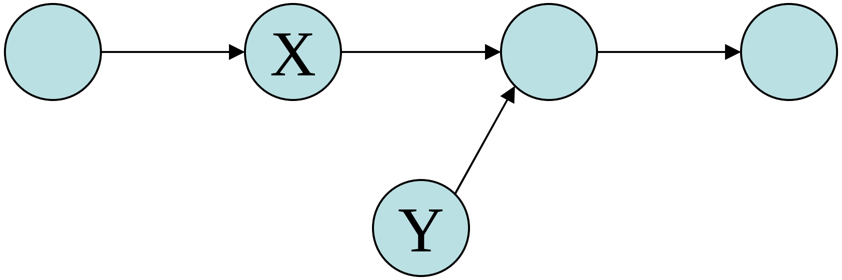
- Naïve Implementation is just a linked list

Naïve Order Maintenance Insert



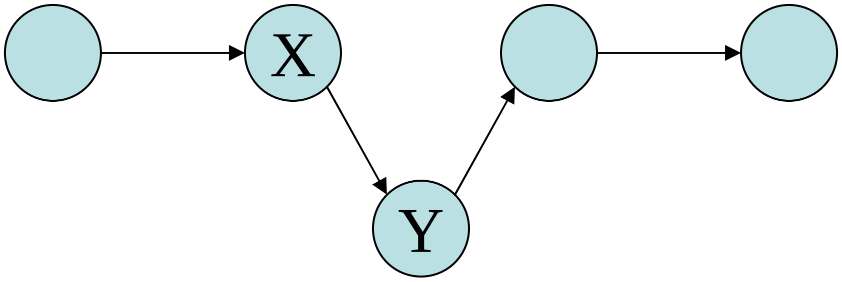
- Insert(X,Y) does standard linked list insert

Naïve Order Maintenance Insert



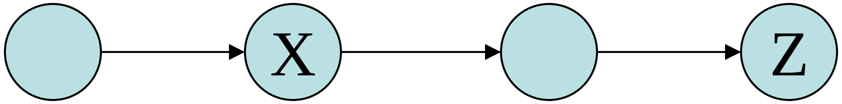
- Insert(X,Y) does standard linked list insert

Naïve Order Maintenance Insert



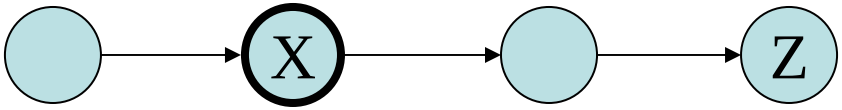
- Insert(X,Y) does standard linked list insert

Naïve Order Maintenance Query



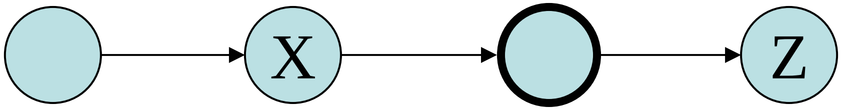
- Precedes(X,Z) looks forward in list.

Naïve Order Maintenance Query



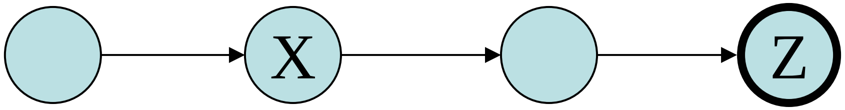
- Precedes(X,Z) looks forward in list.

Naïve Order Maintenance Query



- Precedes(X,Z) looks forward in list.

Naïve Order Maintenance Query

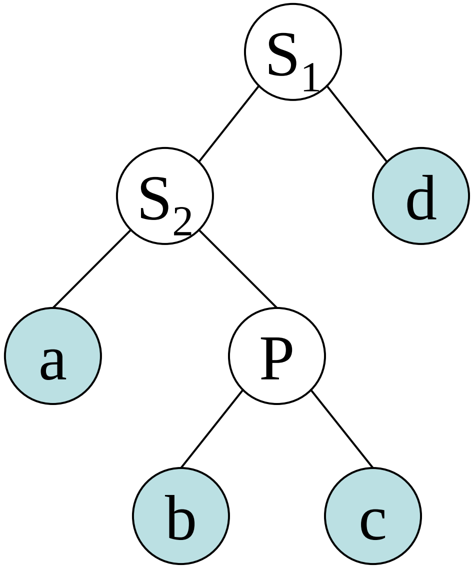


- Precedes(X,Z) looks forward in list.

The algorithm

- Recall, we are thinking in terms of parse tree.
- Maintain two order structures.
- When executing node *x*:
  - Insert children of *x* after *x* in the lists.
  - Ordering of children depends on whether *x* is an S or P node.

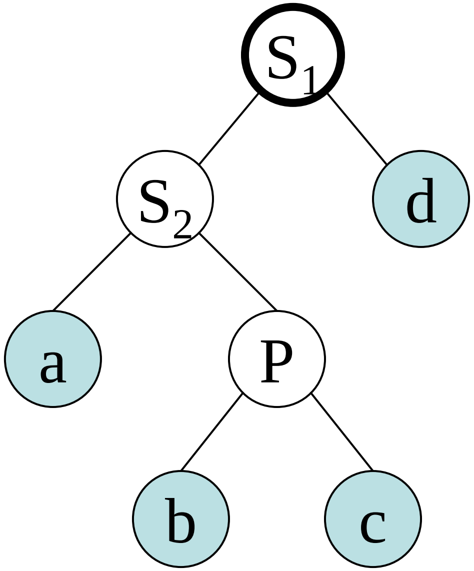
Example

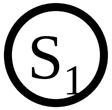


Order 1:

Order 2:

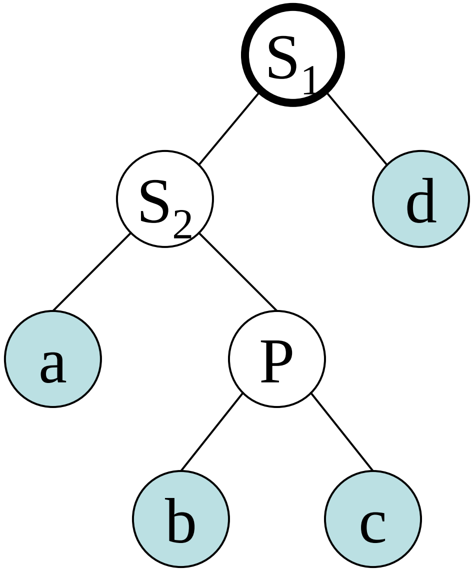
Example



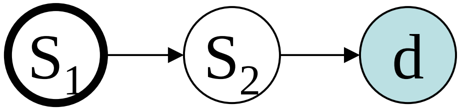
Order 1: 

Order 2: 

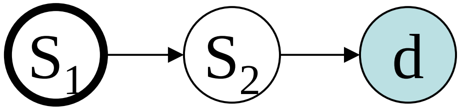
Example



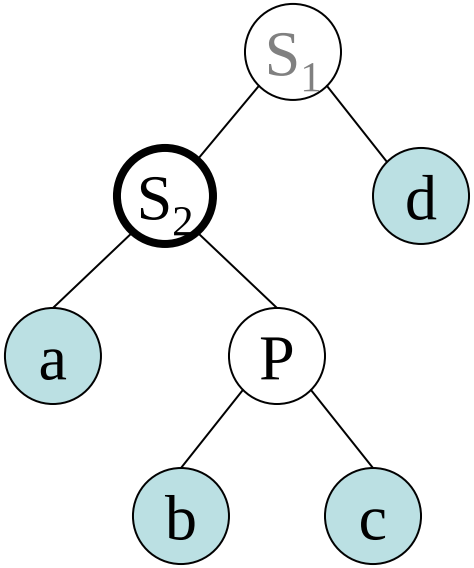
Order 1:



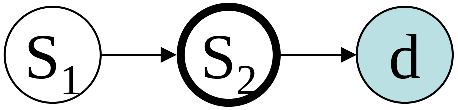
Order 2:



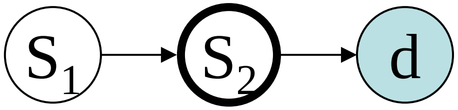
Example



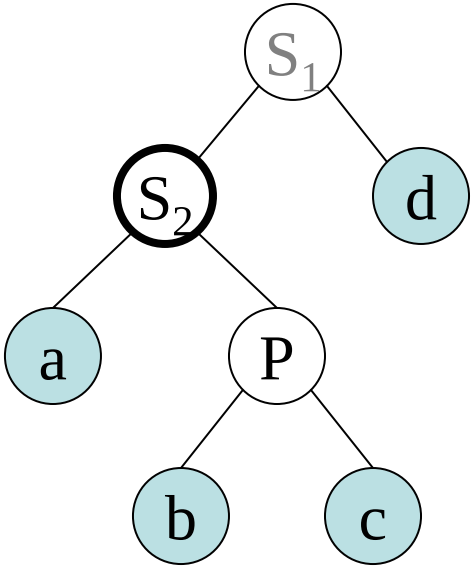
Order 1:



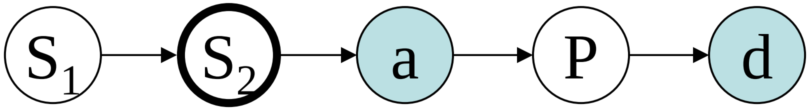
Order 2:



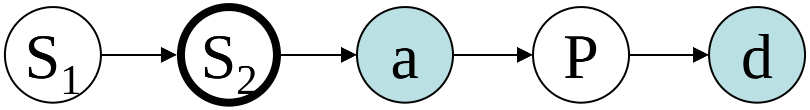
Example



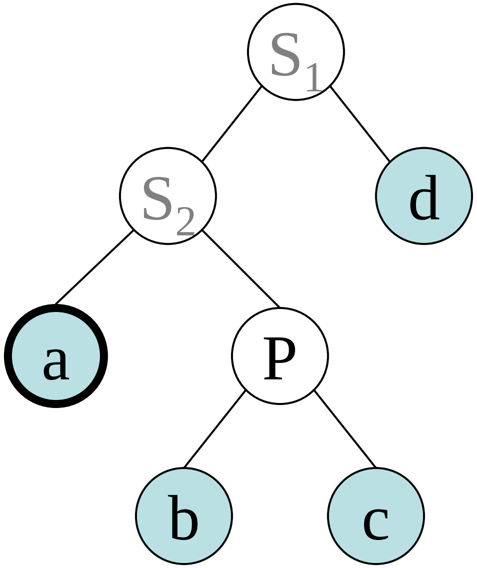
Order 1:



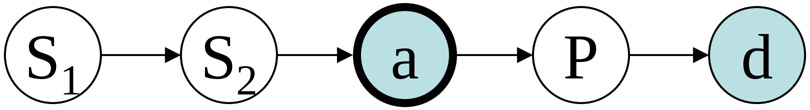
Order 2:



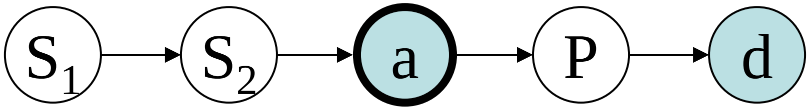
Example



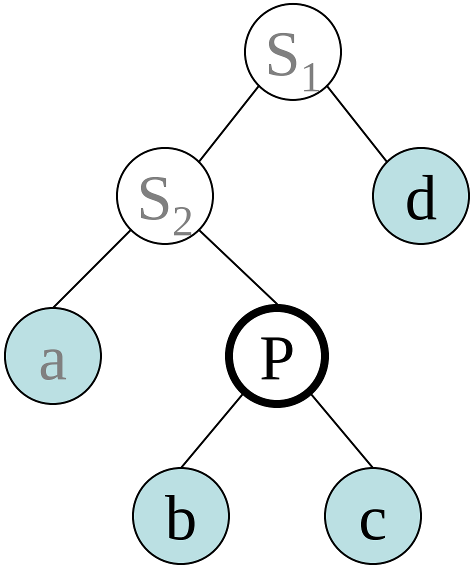
Order 1:



Order 2:

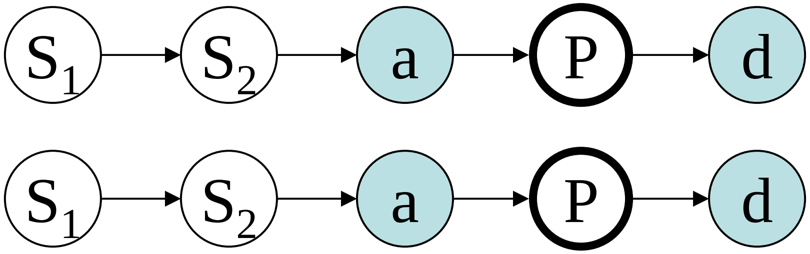


Example

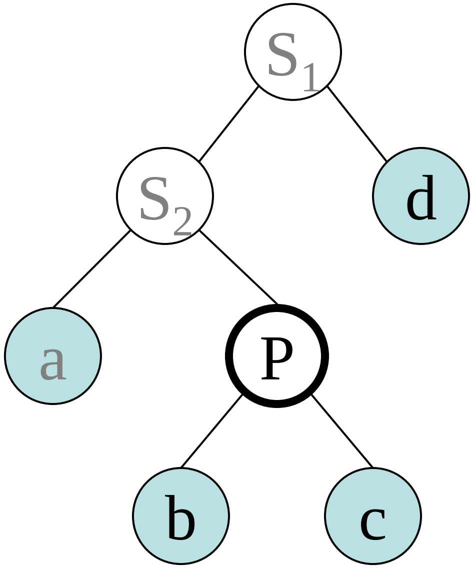


Order 1:

Order 2:



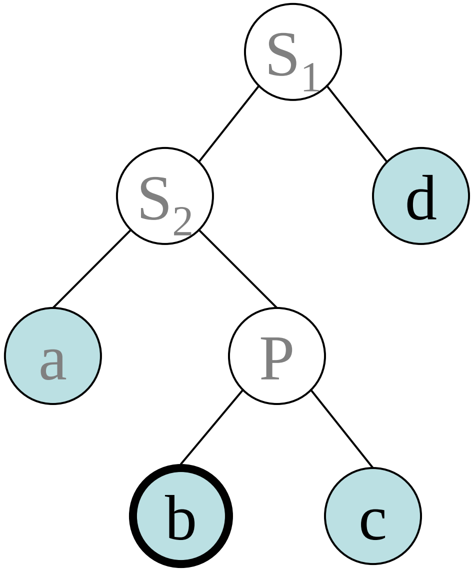
Example

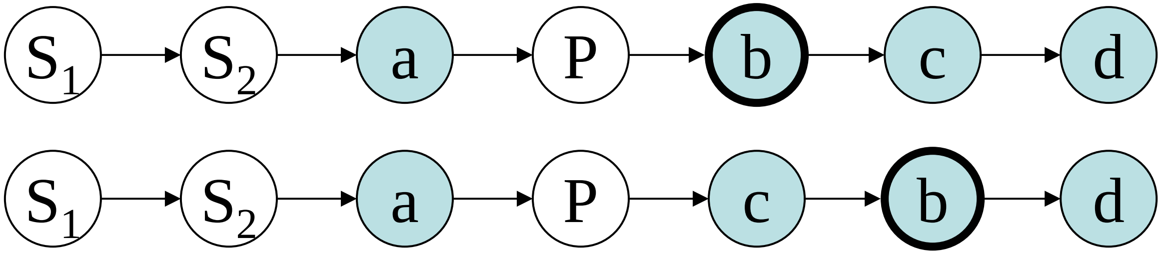


Order 1: S_1 → S_2 → a → P → b → c → d

Order 2: S_1 → S_2 → a → P → c → b → d

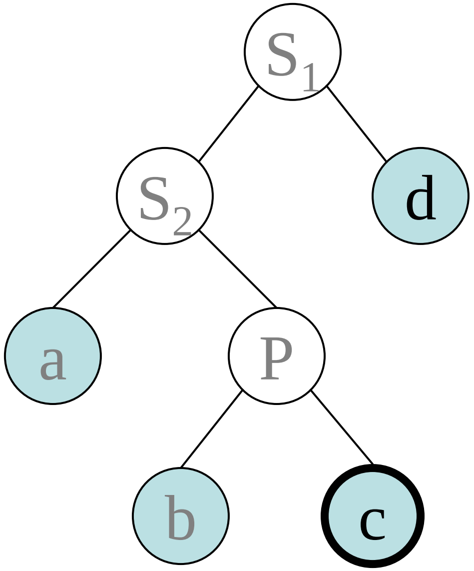
Example

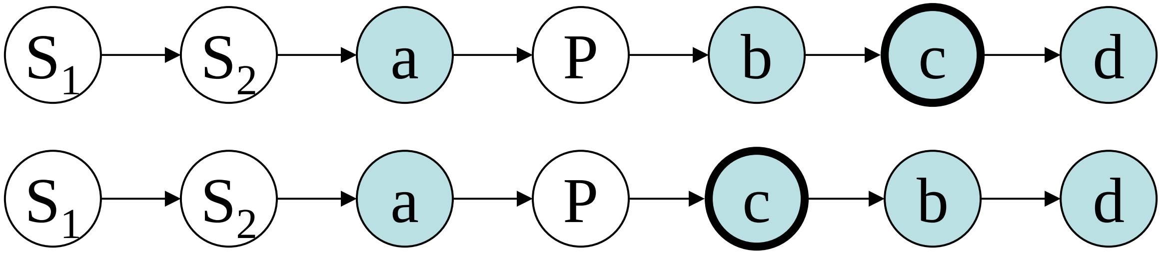



Order 1:

Order 2:

Example

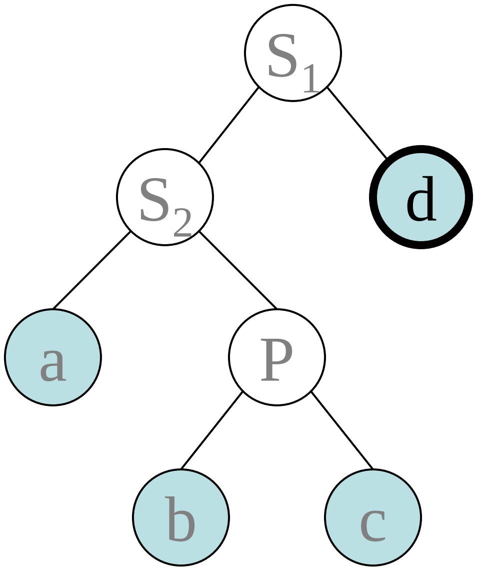




Order 1: $S_1 \rightarrow S_2 \rightarrow a \rightarrow P \rightarrow b \rightarrow c \rightarrow d$

Order 2: $S_1 \rightarrow S_2 \rightarrow a \rightarrow P \rightarrow c \rightarrow b \rightarrow d$

Example



Order 1: S_1 → S_2 → a → P → b → c → d

Order 2: S_1 → S_2 → a → P → c → b → d

Analysis

- Correctness does not depend on execution
  - Any valid serial or parallel execution produces correct results.
    - Inserts after x in orders only happen when x executes.
    - Only one processor will ever insert after x.
- Running time depends on implementation of order maintenance data structure.

Serial Running Time

- Current Nondeterminator does serial execution.
- Can have $O(T_1)$ queries and inserts.
- Naïve implementation is
  - $O(n)$ time for query of n-element list.
  - $O(1)$ time for insert.
  - Total time is very poor: $O(T_1^2)$

Use Dietz and Sleator Order Maintenance Structure

- Essentially a linked list with labels.
- Queries are $O(1)$: just compare the labels.
- Inserts are $O(1)$ amortized cost.
  - On some inserts, need to perform relabel.
- $O(T_1)$ operations only takes $O(T_1)$ time.
  - Gives us linear time Nondeterminator. Better than SP-bags algorithm.

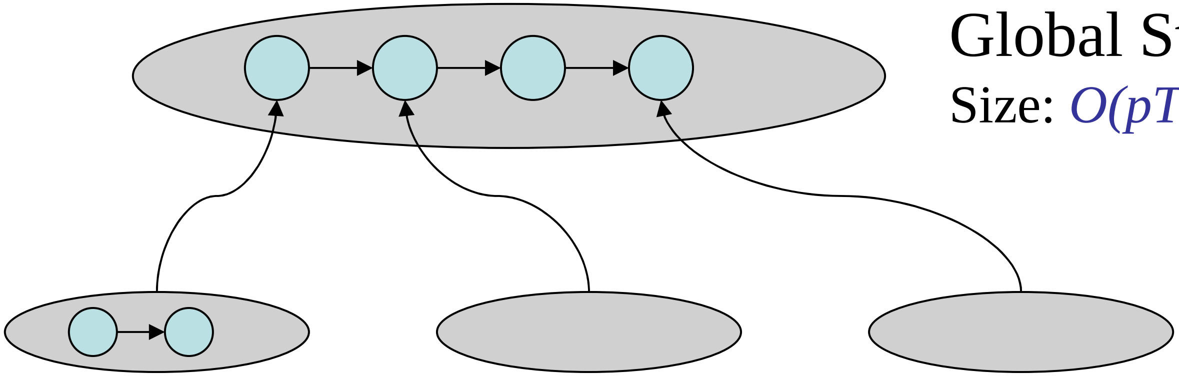
Parallel Problem

- Dietz and Sleator relabels on inserts
  - Does not work concurrently.
- Lock entire structure on insert?
  - Query is still $O(1)$.
  - Single relabel can cost $O(T_1)$ operations.
    - Critical path increases to $O(T_1)$
    - Running time is $O(T_1/p + T_1)$.

Parallel Problem Solution

- Leverage the Cilk scheduler:
  - There are only $O(pT_\infty)$ steals
- There is no contention on subcomputations done by single processor between steals.
  - We do not need to lock every insert.

Parallel Problem Solution



Global Structure

Size: $O(pT_\infty)$

- Top level is global ordering of subcomputations.
- Bottom level is local ordering of subcomputation performed on single processor.
- On a steal, insert $O(1)$ nodes into global structure.

Parallel Running Time

- An insert into a local order structure is still $O(1)$.
- An insert into the global structure involves locking the entire global structure.
  - May need to wait for p processors to insert serially.
  - Amortized cost is $O(p)$ per insert.
  - Only $O(pT_\infty)$ inserts into global structure.
- Total work and waiting time is $O(T_1 + p^2T_\infty)$
  - Running time is $O(T_1/p + pT_\infty)$

Questions?