Continued from last time:
Race Detection in Cilk Computations
Canonical Series-Parallel (SP) Parse Tree

F:
e_1; spawn F_1;
e_2; spawn F_2;
e_3;
sync;
e_4; spawn F_4;
e_5; spawn F_5;
e_6;
sync;
e_7; spawn F_4;
e_8; spawn F_8;
e_9;
sync;
e_10;
**Lemma 1 [LCA in SP tree]:** The least common ancestor (LCA) of two strands determines whether the strands are logically in series or in parallel:

- if $e < e'$ if LCA($e$, $e'$) is an S node and $e$ is to the left of $e'$ (e precedes e')
- if $e \parallel e'$ if LCA($e$, $e'$) is an P node.
**Lemma 1 [LCA in SP tree]:** The LCA of two strands determines whether the strands are logically in series or in parallel:

- if \( e \parallel e' \) if and only if \( \text{LCA}(e, e') \) is an P node.

(The other case, where \( e < e' \) is just a Corollary of this Lemma.)

**Case 1 (\( \Rightarrow \))**: Assume for the purpose of contradiction, \( e \parallel e' \), but their LCA is an S node.

Since the LCA is an S node, the dag \( G_1 \) containing \( e_1 \) must be connected in series with the dag \( G_2 \) containing \( e_2 \):

Then there must be a path from \( e_1 \) to \( e_2 \). Contradiction!
**Lemma 1 [LCA in SP tree]:** The LCA of two strands determines whether the strands are logically in series or in parallel:

- if $e \parallel e'$ if and only if $\text{LCA}(e, e')$ is an P node.
  
  (The other case, where $e < e'$ is just a Corollary of this Lemma.)

**Case 2 ($\Leftarrow$):** Assume for the purpose of contradiction $e < e'$, but their LCA is a P node. That means, $G_1$ and $G_2$ are connected w/ parallel composition.

Since there is a path from $e_1$ to $e_2$, there is a path from the sink of $G_1$ to source of $G_2$, making the graph cyclic. Contradiction!
Lemma 1 [LCA in SP tree]: The least common ancestor (LCA) of two strands determines whether the strands are logically in series or in parallel:

- if $e < e'$ if LCA($e$, $e'$) is an S node and $e$ is to the left of $e'$ (e precedes $e'$)
- if $e \parallel e'$ if LCA($e$, $e'$) is an P node.
Overview of Nondeterminator

• A serial tool --- it executes a Cilk computation serially, but analyzes the parallel constructs for a given input.
• The program is compiled so that every load and store in the user program is instrumented.
• As the program executes, the Nondeterminator maintains:
  • a *shadow space* that keeps track of the memory accesses seen by the execution thus far;
  • an *SP-bag data structure* that keeps track of the series-parallel relationship among strands (so implicitly it’s keep track of the shape of the SP parse tree).
• Race is reported when two logically parallel strands access the same memory location in a conflicting way.
• **Guarantee:** reports a race if and only if the computation (program + input) contains a race.
The SP-Bags Data Structure

For each active procedure on the call stack, the Nondeterminator maintains an S bag and a P bag:

- **S-Bag $S_F$**: Contains IDs of F’s completed descendants (including F itself) that logically precede the currently executing strand.
- **P-Bag $P_F$**: Contains IDs of F’s completed descendants that operate logically in parallel with the currently executing strand.
The SP-Bags Data Structure

For each active procedure on the call stack, the Nondeterminator maintains an S bag and a P bag:

- **S-Bag $S_F$**: Contains IDs of F’s completed descendants (including F itself) that logically precede the currently executing strand.
- **P-Bag $P_F$**: Contains IDs of F’s completed descendants that operate logically in parallel with the currently executing strand.

When $e_1$ executes, $S_F = \{F_1, F_2, F_3\}$.
When $e_2$ executes, $S_F = \{F_1, F_2, F_3\}$ and $P_F = \{F_4, F_5\}$.
The SP-Bags Data Structure

The Nondeterminator uses a disjoint-set data structure to maintain the S and P bags of procedures on the call stack.

Definition [Disjoint-Set Data Structure (Union-Find)]: Union-Find maintains a collection \( \Sigma \) of disjoint sets. For two sets \( X \) and \( Y \), \( X \& Y \in \Sigma \implies X \cap Y = \emptyset \). For each set \( X \in \Sigma \) typically has a designated "leader" element \( x \in X \) which is used to "name" the set. The data structure maintains the collection \( \Sigma \) and answers the following queries:

- **Make-set(e):** \( \Sigma \leftarrow \Sigma \cup \{ \{e\} \} \)
  Adds a new set \( \{e\} \) into the collection \( \Sigma \).

- **Union(X, Y):** \( \Sigma \leftarrow \Sigma - \{X,Y\} \cup \{X \cup Y\} \)
  Removes individual sets \( X \) and \( Y \) and replaces them with the union of \( X \) and \( Y \).

- **Find-set(e):** Returns \( X \in \Sigma \) such that \( e \in X \). Note that sets in \( \Sigma \) is named by their leaders, so this returns the leader representing the set.
The SP-Bags Data Structure

The Nondeterminator uses a disjoint-set data structure to maintain the S and P bags of procedures on the call stack.

**Definition [Disjoint-Set Data Structure (Union-Find)]:** Union-Find maintains a collection $\Sigma$ of disjoint sets. For two sets $X$ and $Y$, $X \& Y \in \Sigma \implies X \cap Y = \emptyset$. For each set $X \in \Sigma$ typically has a designated "leader" element $x \in X$ which is used to "name" the set. The data structure maintains the collection $\Sigma$ and answers the following queries:

- Make-set(e)
- Union(X, Y)
- Find-set(e)

**Theorem [Operations on Disjoint-set Data structure] (Tarjan 1975):** Any sequence of $m$ operations on $n$ sets can be performed in $O(m \ A(m, n))$, where $A$ is the inverse Ackermann's function (a really really slow growing function).
The SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator and it performs two types of operations.

• **spawn procedure F:**
  \[ S_F \leftarrow \text{Make-set}(F); \quad \text{(F is the leader)} \]
  \[ P_F \leftarrow \emptyset \]

• **sync in procedure F:**
  \[ S_F \leftarrow \text{Union}(S_F, P_F); \]
  \[ P_F \leftarrow \emptyset \]

• **return from F' to F (F' is spawned):**
  \[ P_F \leftarrow \text{Union}(P_F, S_{F'}); \]
  \( \text{(Note that } P_{F'} \text{ must be empty at this point.)} \)

The first type updates the S and P bags for all procedures on the call stack, which is triggered during the DFS traversal of the SP parse tree.
The SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator and it performs two types of operations.

**Shadow memory:**
- **writer[v]:** the ID of the last procedure that wrote to v.
- **reader[v]:** the ID of the a procedure that read v (not necessarily the last one).

**write location v by procedure F:**
- if (Find-set(reader[v]) is a P-bag
- Or Find-set(writer[v]) is a P-bag)
  then report race;
- writer[v] ← F;

**read location v by procedure F:**
- if (Find-set(writer[v]) is a P-bag)
  then report race;
- if (Find-set(reader[v]) is an S-bag)
  then reader[v] ← F;
  (Replace reader only when it's in an S-bag)

The second type uses the SP-bags data structure to detect determinacy races when the user program accesses a memory location.
Justification of the SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator and it performs two types of operations.

• **spawn procedure \( F \):**
  
  \[ S_F \leftarrow \text{Make-set}(F); \ (F \text{ is the leader}) \]
  
  \[ P_F \leftarrow \emptyset \]

Recall:

• **S-Bag \( S_F \):** Contains IDs of \( F \)'s completed descendants (including \( F \) itself) that logically precede the currently executing strand.

• **P-Bag \( P_F \):** Contains IDs of \( F \)'s completed descendants that operate logically in parallel with the currently executing strand.

This operation is valid since the S-bag of \( F \) by definition contains itself, and \( F \) has no valid child yet.
Justification of the SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator and it performs two types of operations.

- **sync in procedure F:**
  
  \[ S_F \leftarrow \text{Union}(S_F, P_F); \]
  
  \[ P_F \leftarrow \emptyset \]

Recall:

- **S-Bag** \( S_F \): Contains IDs of F’s completed descendants (including F itself) that logically precede the currently executing strand.

- **P-Bag** \( P_F \): Contains IDs of F’s completed descendants that operate logically in parallel with the currently executing strand.

After a sync, we switch to a strand e right after sync from some strand e' right before sync. Originally \( P_F \) contains IDs of F’s completed descendants that operate logically in parallel with e'. These procedures now must operate in series with e (and anything else that F will spawn). Thus, it's valid to move the IDs in \( P_F \) into \( S_F \).
Justification of the SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator and it performs two types of operations.

- **return from F' to F (F' is spawned):**
  \[ P_F \leftarrow \text{Union}(P_F, S_{F'}); \]
  (Note that \( P_{F'} \) must be empty at this point.)

Recall:

- **S-Bag \( S_F \):** Contains IDs of F’s completed descendants (including F itself) that logically precede the currently executing strand.
- **P-Bag \( P_F \):** Contains IDs of F’s completed descendants that operate logically in parallel with the currently executing strand.

Before a function F' returns, \( P_{F'} \) is empty, since there is always an implicit sync. Also, \( S_{F'} \) contains all the logical descendants of F', which are also logical descendants of F, and can now execute in parallel with any procedures that F might spawn in the future (before the next sync).
Justification of the SP-Bags Algorithm

To understand the second type of operations, we need some lemmas first.

**Recall Lemma 1 [LCA in SP tree]:**

- if $e \parallel e'$ if and only if LCA($e, e'$) is an P node.

**Lemma 2:** Let strands $e_1$, $e_2$, and $e_3$ execute serially in order. If $e_1 \prec e_2$ and $e_1 \parallel e_3$, then $e_2 \parallel e_3$.

**Proof:** Suppose for the sake of contradiction that $e_2 \prec e_3$. Then, by transitivity, we'd have $e_1 \prec e_3$. Contradiction.

Note that the parallel relation $\parallel$, unlike precedes $\prec$, is not transitive.

In this tree, $e_1 \parallel e_3$ and $e_1 \parallel e_3$ but $e_1 \not\parallel e_3$. 
Justification of the SP-Bags Algorithm

To understand the second type of operations, we need some lemmas first.

**Recall Lemma 1 [LCA in SP tree]:**
- if $e \parallel e'$ if and only if $\text{LCA}(e, e')$ is an P node.

**Lemma 3 [Pseudotransitivity of $\parallel$]:**
Let strands $e_1$, $e_2$, and $e_3$ execute serially in order.
If $e_1 \parallel e_2$ and $e_2 \parallel e_3$, then $e_1 \parallel e_3$.

**Proof:** Since we do a depth-first traversal of the tree, the only possible options for the tree that have them in the right serial order are:

In both cases, we know that both $\text{LCA}(e_1, e_2)$ and $\text{LCA}(e_2, e_3)$ are P nodes. So the $\text{LCA}(e_1, e_3)$, which is $a_1$, must also be a P node.
Justification of the SP-Bags Algorithm

Define $h(a)$ to be the procedure that immediately enclose strand $a$.

**Lemma 4 [SP-Bags maintenance]:** Let $e_1$ be executed before $e_2$, and let $a = \text{LCA}(e_1, e_2)$ in the SP parse tree.

- if $e_1 \prec e_2 \implies h(e_1)$ is in an $S$-bag($h(a)$) when $e_2$ executes.
- if $e_1 \parallel e_2 \implies h(e_1)$ is in a $P$-bag($h(a)$) when $e_2$ executes.

**Proof sketch:**
Case 1: Since $a$ is an $S$-node, $a$ must belongs to either the spine or within a sync block.

If $a$ belongs to the spine, then $e_1$ belongs to $a$'s left subtree and $e_2$ to $a$'s right subtree. Either $h(e_1) = h(a)$ or $h(e_1)$ is a descendant of $h(a)$.

If $h(e_1)$ is $h(a)$, $h(e_1)$ is already in $h(a)$'s $S$ bag.
Justification of the SP-Bags Algorithm

Define $h(a)$ to be the procedure that immediately enclose strand $a$.

**Lemma 4 [SP-Bags maintenance]**: Let $e_1$ be executed before $e_2$, and let $a = \text{LCA}(e_1, e_2)$ in the SP parse tree.

- if $e_1 \prec e_2 \Rightarrow h(e_1)$ is in an $S$-bag($h(a)$) when $e_2$ executes.
- if $e_1 \parallel e_2 \Rightarrow h(e_1)$ is in a $P$-bag($h(a)$) when $e_2$ executes.

**Proof sketch:**
Case 1: Since $a$ is an $S$-node, $a$ must belongs to either the spine or within a sync block.

If $h(e_1)$ is not $h(a)$, $h(e_1)$ moves up into some bags in the call stack when its ancestor returns.
Once the sync corresponds to $a$'s left subtree executes, $h(e_1)$ moves into $h(a)$'s $S$ bag (and stays there).
Justification of the SP-Bags Algorithm

Define \( h(a) \) to be the procedure that immediately encloses strand \( a \).

**Lemma 4 [SP-Bags maintenance]**: Let \( e_1 \) be executed before \( e_2 \), and let \( a = \text{LCA}(e_1, e_2) \) in the SP parse tree.
- if \( e_1 \prec e_2 \Rightarrow h(e_1) \) is in an \( S\text{-bag}(h(a)) \) when \( e_2 \) executes.
- if \( e_1 \parallel e_2 \Rightarrow h(e_1) \) is in a \( P\text{-bag}(h(a)) \) when \( e_2 \) executes.

**Proof sketch:**
Case 2: Since \( a \) is an \( P\)-node.

In this case, \( a \) must be within a sync block and \( e_1 \) belongs to the left subtree and \( e_2 \) to the right. At this point, the left child of a \( P \) node is always a spawned procedure \( F' \) that gets placed into \( h(a) \)'s \( P \) bag when the \( F' \) returns. Since no sync has occurred yet, \( F' \) must still be in a \( P \) bag when \( e_2 \) executes.
Proof of the SP-Bags Race Detection

**Theorem [SP-Bags correctness]**: The SP-Bags algorithm reports a race in a Cilk computation if and only if a determinacy race exists.

**Proof sketch**: The \((\Rightarrow)\) case is straight-forward. If the SP-Bags reports a race, that means it detected two strands logically in parallel that accesses the same memory location in a conflicting way. Thus, if it reports a race, a determinacy race exists.
Proof of the SP-Bags Race Detection

**Theorem [SP-Bags correctness]:** The SP-Bags algorithm reports a race in a Cilk computation if and only if a determinacy race exists.

**Proof sketch:** The (⟸) case is trickier. We want to show that if a det. race exists, the SP-Bags algorithm reports it. Let \( e_1 \parallel e_2 \) and have a race on \( v \). Assume \( e_1 \) executes before \( e_2 \). If there are several races, choose \( e_2 \) to be the race whose strand executes earliest in the serial order.

Case 1: Say \( e_1 \) writes to \( v \) and \( e_2 \) reads it.
Suppose when \( e_2 \) executes, \( \text{writer}[v] = h(e) \) for some \( e \).
If \( e = e_1 \) then we are done, since we know that \( h(e_1) \) is in a P bag of \( \text{LCA}(e_1, e_2) \) (by Lemma SP-bags maintenance).
If \( e \) is not \( e_1 \), then \( e \) must have been executed after \( e_1 \) and before \( e_2 \). Then either \( e_1 < e \), then \( e \parallel e_2 \) by Lemma 2* shown earlier.
Or \( e_1 \parallel e \), then there is already a race between \( e_1 \) and \( e \), which contradicts our assumption about \( e_2 \) being the earliest race.

* **Lemma 2:** Let strands \( e_1, e_2, \) and \( e_3 \) execute serially in order. If \( e_1 < e_2 \) and \( e_1 \parallel e_3 \), then \( e_2 \parallel e_3 \).
Proof of the SP-Bags Race Detection

**Theorem [SP-Bags correctness]:** The SP-Bags algorithm reports a race in a Cilk computation if and only if a determinacy race exists.

**Proof sketch:** The (\(\iff\)) case is trickier. We want to show that if a det. race exists, the SP-Bags algorithm reports it. Let \(e_1 \parallel e_2\) and have a race on \(v\). Assume \(e_1\) executes before \(e_2\). If there are several races, choose \(e_2\) to be the race whose strand executes earliest in the serial order.

Case 2: Say \(e_1\) writes to \(v\) and \(e_2\) writes it. This is similar to case 1.
**Proof of the SP-Bags Race Detection**

**Theorem [SP-Bags correctness]**: The SP-Bags algorithm reports a race in a Cilk computation iff a determinacy race exists.

**Proof sketch**: The \((\Leftarrow)\) case is trickier. We want to show that if a det. race exists, the SP-Bags algorithm reports it. Let \(e_1 \parallel e_2\) and have a race on \(v\). Assume \(e_1\) executes before \(e_2\). If there are several races, choose \(e_2\) to be the race whose strand executes earliest in the serial order.

Case 3: Say \(e_1\) reads to \(v\) and \(e_2\) writes it.

Again, suppose \(\text{reader}[v] = e\). If \(e = e_1\), then then we are done, since we know that \(k(e_1)\) is in a P bag of \(\text{LCA}(e_1, e_2)\) (by Lemma SP-bags maintenance). So, we can assume \(e \neq e_1\). There are two possibilities.

Case 3.1: \(\text{reader}[v]\) was \(e_1\) at some point, but eventually got overwritten by \(e\) (there can be some \(e'\) in \(\text{reader}[v]\) in between \(e_1\) and \(e\)). This can occur only if \(e_1 \prec e\). Since \(e_1 \prec e\) and \(e_1 \parallel e_2, e \parallel e_2\), (again by Lemma 2*) so a race is reported.

* **Lemma 2**: Let strands \(e_1, e_2, \) and \(e_3\) execute serially in order. If \(e_1 \prec e_2\) and \(e_1 \parallel e_3\), then \(e_2 \parallel e_3\).
Proof of the SP-Bags Race Detection

Theorem [SP-Bags correctness] : The SP-Bags algorithm reports a race in a Cilk computation iff a determinacy race exists.

Proof sketch: The (\(\iff\)) case is trickier. We want to show that if a det. race exists, the SP-Bags algorithm reports it. Let \(e_1 \| e_2\) and have a race on \(v\). Assume \(e_1\) executes before \(e_2\). If there are several races, choose \(e_2\) to be the race whose strand executes earliest in the serial order.

Case 3: Say \(e_1\) reads to \(v\) and \(e_2\) writes it.

Again, suppose \(\text{reader}[v] = e\). If \(e = e_1\), then then we are done, since we know that \(h(e_1)\) is in a P bag of \(\text{LCA}(e_1, e_2)\) (by Lemma SP-bags maintenance). So, we can assume \(e \neq e_1\). There are two possibilities.

Case 3.2: \(\text{reader}[v]\) was never updated to be \(e_1\). Let's assume when \(e_1\) executes, \(\text{reader}[v] = e'\). Then it must be that \(e \| e_1\) or we'd have updated \(\text{reader}[v]\). Then by Lemma Pseudotransivity of \(\|\), \(e' \| e_2\), and a race is reported.

*Pseudotransitivity of \(\|\) : \(e_1, e_2,\) and \(e_3\) execute serially in order. If \(e_1 \| e_2\) and \(e_2 \| e_3\), then \(e_1 \| e_3\).
Extensions for Parallel Race Detection
What We Need in a Det. Race Detector

• SP-Bags data structure: maintaining the series-parallel ordering of strands.

• Shadow space that contains:
  – The last writer to a location $v$; and
  – The last serial reader to a location $v$.
    (But we are totally dropping the parallel readers.)

**Question:** Can we extend the SP-Bags algorithm to race detect a Cilk computation executing in parallel?
Where Things Break

• The SP-Bags data structure maintenance is inherently serial: it keeps track of procedure IDs that are in series / parallel with respect to the "currently executing strand."

• The shadow memory only keeps track of the last serial reader (that the execution encounters), which is insufficient.
On-the-Fly Maintenance of Series-Parallel Relationships

The *English-Hebrew* orderings:

The nodes in the left subtree of an S-node always precede those in the right subtree.
On-the-Fly Maintenance of Series-Parallel Relationships

The *English-Hebrew* orderings:

The nodes in the left subtree of an S-node always precede those in the right subtree.

*English order:* the nodes in the left subtree of a P-node precede those in the right subtree.
On-the-Fly Maintenance of Series-Parallel Relationships

The *English-Hebrew* orderings:

The nodes in the left subtree of an S-node always precede those in the right subtree.

*English order*: the nodes in the left subtree of a P-node precede those in the right subtree.

*Hebrew order*: the nodes in the right subtree of a P-node precede those in the left.

Key observation:

Under a S-node:

- $E[u_{left}] < E[u_{right}]$
- $H[u_{left}] < H[u_{right}]$

Under a P-node:

- $E[u_{left}] < E[u_{right}]$
- $H[u_{left}] > H[u_{right}]$
On-the-Fly Maintenance of Series-Parallel Relationships

The *English-Hebrew* orderings:

![Diagram](image)

**Observation #1:**
Under a S-node:
\[ E[u_{\text{left}}] < E[u_{\text{right}}] \]
\[ H[u_{\text{left}}] < H[u_{\text{right}}] \]

Under a P-node:
\[ E[u_{\text{left}}] < E[u_{\text{right}}] \]
\[ H[u_{\text{left}}] > H[u_{\text{right}}] \]

**Question:** *Can we maintain the two labeling on the fly as the computation executes?*

**Observation #2:** One doesn't need to assign specific labels for each strand; a relative-ordering suffices.
The SP-Order Algorithm

SP-Order(X): // X is a node is the SP tree
if IsLeaf(X)
    execute strand X
return
// otherwise X is an internal node
OM-Insert(Eng, X, left[X], right[X])
if IsSNode(X)
    OM-Insert(Heb, X, left[X], right[X])
else
    OM-Insert(Heb, X, right[X], left[X])
SP-Order(left[X])
SP-Order(right[X])

To detect race between two strands, check if they are in the same relative order to each other in both Eng and Heb.

OM-Insert(L, X, Y₁, Y₂):
In the ordering L, insert new element Y₁ and Y₂ immediately after X.
The SP-Order Algorithm

SP-Order(X): // X is a node is the SP tree
if IsLeaf(X)
   execute strand X
return
// otherwise X is an internal node
OM-Insert(Eng, X, left[X], right[X])
if IsSNode(X)
   OM-Insert(Heb, X, left[X], right[X])
else
   OM-Insert(Heb, X, right[X], left[X])
SP-Order(left[X])
SP-Order(right[X])

To detect race between two strands, check if they are in the same relative order to each other in both Eng and Heb.

Naïve parallelization: the Order Maintenance data structure becomes a scalability bottleneck.
SP-Hybrid

- Recall: between successful steals, each worker's behavior mirrors the serial execution.
  - trace: the execution done by a worker between steals.

- A two-tier scheme:
  - global tier: use a global Order Maintenance data structure to maintain the ordering between traces.
    - a clever design of a concurrent data structure allows one to query the data structure without locking.
  - local tier: within a trace, query SP relationships using the SP-bags data structure.

**Challenge:** traces are defined dynamically as steals occur, so how do we keep track of that?
SP-Hybrid

• Recall: between successful steals, each worker's behavior mirrors the serial execution.

  trace: the execution done by a worker between steals.

• A two-tier scheme:

  global tier: use a global Order Maintenance data structure to maintain the ordering between traces.

    - a clever design of a concurrent data structure allows one to query the data structure without locking.

  local tier: within a trace, query SP relationships using the SP-bags data structure.

Challenge: traces are defined dynamically as steals occur, so how do we keep track of that?
Splitting Traces On-the-Fly

Each $U^{(i)}$ is a trace containing a set of strands.

- $U^{(1)}$: the strands that precedes $X$
- $U^{(2)}$: the strands that is in parallel w/ $X$
- $U^{(3)}$: the strands in $X$'s left subtree (that is currently being executed by the victim)
- $U^{(4)}$: the strands in $X$'s right subtree (initially empty and will be populated by the thief)
- $U^{(5)}$: the strands that follows $X$ (initially empty)

Upon a steal, insert into the global tier:

**English:**

```
U^{(1)} -> U^{(2)} -> U^{(3)} -> U^{(4)} -> U^{(5)}
```

**Hebrew:**

```
U^{(1)} -> U^{(4)} -> U^{(3)} -> U^{(2)} -> U^{(5)}
```
What We Need in a Det. Race Detector

• SP-Hybrid:
  global tier: an Order-Maintenance data structure maintains the series-parallel ordering of traces.
  local tier: within a single trace, query the SP-Bags data structure.

• Shadow space that contains:
  – The last writer to a location \( v \); and
  – The last serial reader to a location \( v \).
    (But we are totally dropping the parallel readers.)
Where Things Break

• The shadow memory only keeps track of the last serial reader (that the execution encounters), which is insufficient.

Recall the lemmas we need to show that SP-Bags algorithm works correctly:

**Lemma 2:** Let strands $e_1$, $e_2$, and $e_3$ execute serially in order.
If $e_1 \prec e_2$ and $e_1 \parallel e_3$, then $e_2 \parallel e_3$.

**Lemma 3 [Pseudotransitivity of $\parallel$]:**
Let strands $e_1$, $e_2$, and $e_3$ execute serially in order.
If $e_1 \parallel e_2$ and $e_2 \parallel e_3$, then $e_1 \parallel e_3$.

**Question:** When executing in parallel, what do we need to maintain in the shadow space?
Ex: Keeping One Reader Is Not Enough

Recall how to update shadow memory:
• write location v by procedure F:
  \[ \text{writer}[v] \leftarrow F; \]
  (Always update writer)

• read location v by procedure F:
  if \((\text{Find-set}(\text{reader}[v])\) is an S-bag
    then \[ \text{reader}[v] \leftarrow F; \]
    (Replace only serial reader)

Say e₁, e₂, and e₃ executed in that order in parallel execution.
Say e₁ and e₂ read v and e₃ wrote to v.

When e₃ executes, reader[v] contains e₁, since e₁ \parallel e₂. We miss a race!

What if we always update the reader[v] with the last reader?
Ex: Keeping One Reader Is Not Enough

Recall how to update shadow memory:

- **write location v by procedure F:**
  
  \[
  \text{writer}[v] \leftarrow F;
  \]

  (Always update writer)

- **read location v by procedure F:**
  
  \[
  \text{if (Find-set(reader[v]) is an S-bag)}
  \]
  
  \[
  \text{then reader[v] \leftarrow F;}
  \]

  (Replace only serial reader)

Say \(e_1, e_2,\) and \(e_3\) executed in that order in parallel execution.
Say \(e_1\) and \(e_2\) read \(v\) and \(e_3\) wrote to \(v\).

When \(e_3\) executes, \(\text{reader}[v]\) contains \(e_1\), since \(e_1 \parallel e_2\). We miss a race!

What if we always update the \(\text{reader}[v]\) with the last reader?

Then when \(e_3\) executes, \(\text{reader}[v]\) contains \(e_2\) and we can still miss a race!
Keeping two readers

It turns out that, it's sufficient to keep two readers --- we just need to keep track of the "left-most" \( R_l \) and "right-most" reader \( R_r \) for each memory location \( v \).

When \( e \) reads \( v \):
if \( e \) comes before \( R_l[v] \) in serial order, or \( e \prec R_l[v] \)
   \[ R_l[v] = e \]
if \( e \) comes after \( R_r[v] \) in serial order, or \( R_r[v] \prec e \)
   \[ R_r[v] = e \]
References

[1] Nondeterminator and SP-Bags algorithms: 

[2] On parallel race detection in Cilk computations:


[4] Keeping two readers for parallel race detection in fork-join multithreaded programs: