Massachusetts Institute of Technology | Lecture 12
6.892: Theory of Parallel Systems | March 18, 1997
Professor Charles E. Leiserson | Scribe: Francine Fang

# Lecture 12: Efficient Detection of Determinacy Races in Cilk Programs

## 1 Determinacy Races

A Cilk program contains a determinacy race if two logically parallel threads access the same shared location, and one of the accesses is a write. To illustrate how a determinacy race can occur, consider the following simple Cilk program:

```
int x;
cilk void foo()
{
    x ++;
    return;
}

cilk int main()
{
    x= 0;
    spawn foo();
    spawn foo();
    sync;
    printf ("x is %d", x);
    return 1;
}
```

In this program, both of the parallel instantiations of procedure `foo()` update the shared variable $x$ and $x = x + 1$ statement. This statement actually causes the processor executing the thread to perform a read on $x$, increment the value by 1, and then write the value back into $x$. Since these operations are not atomic, both might update $x$ at the same time. Figure 1 show how this determinacy race can cause x to take on different values. A spawn tree for this program, which illustrates the relationship between Cilk procedures, is shown in figure 2.

The parallel control flow of the above Cilk program can be viewed as a directed acyclic graph (**dag**), as shown in figure 3 The vertices of the dag represent parallel control constructs, and the edges represent Cilk threads.

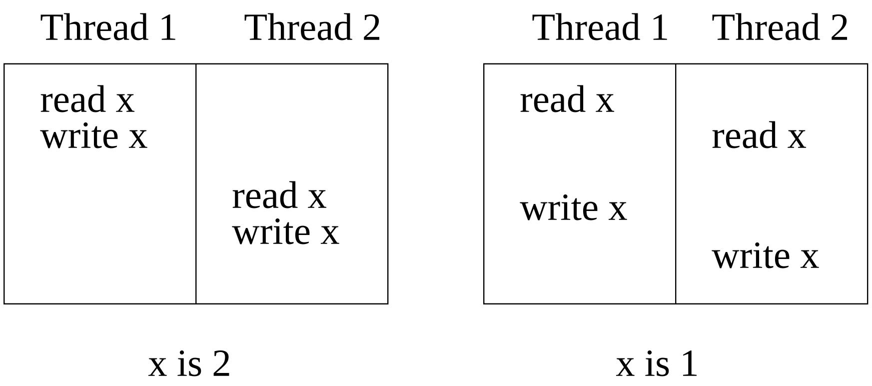


**Figure 1**: Possible results for 2 parallel executions of foo.

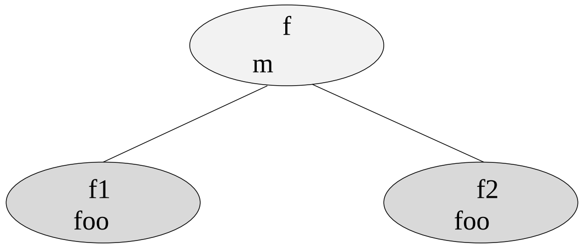


**Figure 2**: A spawn tree for the Cilk program described above. Each node corresponds to a procedure, and its children in the tree are the procedures it spawns.

## 2 The Nondeterminator

The Nondeterminator is a Cilk determinacy-race detector that takes as input a Cilk program and an input data set. It either determines which locations in the program are subject to potential determinacy races when the program is run on the set, or else certifies that the program is race free when run on the particular data set. If a determinacy race exists, the Nondeterminator will provide a file name, a line number, and a state runtime stack or heap. The Nondeterminator is not a program verifier, it is a debugging tool since it can not certify that the program is race-free for all input data sets, just the one specified.

The Nondeterminator was implemented by modifying the Cilk compiler and runtime system. Each read and write in the user's program is instrumented by the Nondeterminator's compiler to perform determinacy-race checking at runtime. The Nondeterminator executes the user's program in a serial depth-first fashion like regular C-execution, and performs race-checking actions when reads, writes, and parallel control statements occur. Since the Nondeterminator instruments every memory operation, it is somewhat slower than normal execution, but this is fine if you are debugging.

The Nondeterminator is based on an algorithm for determinacy race detection called SP-bags algorithm, which is inspired by Tarjan's nearly liner-time least-common-ancestors

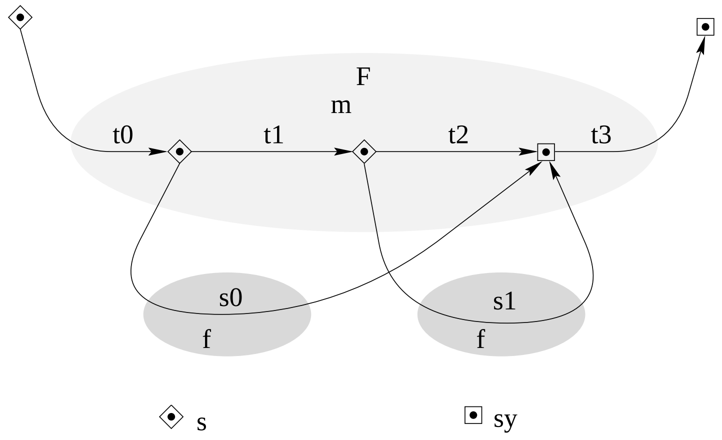


**Figure 3**: The parallel control-flow dag of the Cilk program. A spawn node of the dag represents a `spawn` construct, and a sync node represents a `sync` construct. The edges of the dag are labeled to correspond with code fragments from above. We assume that the start of the program is "spawned" by the operating system, and the end of the program is "synced" by the operating system.

algorithm. The Nondeterminator has a provable performance.

> **Theorem:** For a Cilk program that runs in $T$ time serially and uses $v$ shared-memory locations, the Nondeterminator runs in $\mathcal{O}(T, \alpha(v, v))$ time, where $\alpha$ is Tarjan's functional inverse of Ackermann's function.

Figure 4 compares the asymptotic time and space for the SP-bags algorithm with other race-detection algorithms in the literature. In the following sections, brief descriptions of Serial-Parallel Dags, Tarjan's Least Common Ancestor Algorithm, and the SP-Bags Algorithm will be presented to give a little insights into the correctness and the $\mathcal{O}(T, \alpha(v, v))$ performance of the Nondeterminator.

# 3 Series-Parallel DAGS

Series-parallel dags are a straightforward extension of the notion of series- parallel graphs. A Cilk program execution corresponds to a series-parallel dag. A series-parallel dag $G = (V, E)$ is an acyclic directed graph with two distinguished vertices, a source $s$ belongs in $V$ and a sink $t$ belongs to $V$, which is constructed recursively in one of the three following ways, as illustrated in Figure 5.

> **Base**: the graph consists of a single edge e connecting the source $s$ to the sink $t$.
>
> **Series composition**: The graph consists recursively of two-series-parallel dags G1 and G2 with disjoint edge sets in which the source of G1 is s, the sink of G2 is t, and the sink of G1 is the source of G2.

| Algorithm | Time | | Space |
|---|---|---|---|
| | Thread creation & termination | Per access | |
| English-Hebrew labeling | $O(p)$ | $O(pt)$ | $O(vt + \min(np, vtp))$ |
| Task recycling | $O(t)$ | $O(t)$ | $O(vt + t^2)$ |
| Offset-span labeling | $O(p)$ | $O(p)$ | $O(v + \min(np, vp))$ |
| SP-bags algorithm | $O(\alpha(v, v))$ | $O(\alpha(v, v))$ | $O(v)$ |

$p =$ maximum depth of nested parallelism
$t =$ maximum number of logical concurrent threads
$v =$ number of shared locations being monitored
$n =$ number of threads in an execution

**Figure 4**: Comparison of determinacy-race detection algorithms. The function $\alpha$ is the very slowly growing inverse of Ackermann's function introduced by Tarjan in his analysis of an efficient disjoint-set data structure. For all conceivably practical inputs, the value of this function is at most 4. The time for the SP-bags algorithm is an amortized bound.

**Parallel composition**: The graph consists recursively of two series-parallel dags G1 and G2 with disjoint edge sets in which the source of G1 and G2 are both s and the sinks of G1 and G2 are both t.

# 4 Tarjan's LCA Algorithm

The SP-bag algorithm was inspired by Tarjan's near linear-time least-common-ancestors algorithm. This LCA algorithm uses disjoint-set data structure.

## 4.1 Disjoint-Set Data Structure

The disjoint-set data structure maintains a dynamic collection $\Sigma$ of disjoint sets and provides three elementary operations. ($\Sigma$ is a collection of disjoint sets. Therefore, $X, Y$ in $\Sigma$ implies $X \cap Y = \emptyset$).

*Make-Set*$(x)$: $\Sigma \leftarrow \Sigma \cup \{\{x\}\}$.

*Union*$(X, Y)$: $\Sigma \leftarrow \Sigma - \{X, Y\} \cup \{X \cup Y\}$. The sets $X$ and $Y$ are destroyed.

*Find-Set*$(x)$: Returns the set $X \in \Sigma$ such that $x \in X$.

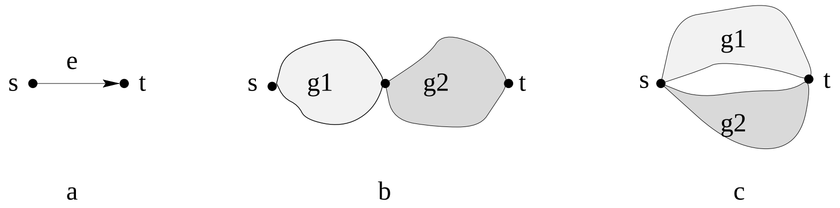


**Figure 5**: The three ways that a series-parallel dag can be constructed. **(a)** A base graph. **(b)** Series composition of two series-parallel dags. **(c)** Parallel composition of two series-parallel dags.

Tarjan shows that any $m$ of these operations on $n$ sets take a total of $O(m\,\alpha(m,n))$ time.

## 4.2 Tarjan's LCA Algorithm

The least common ancestor (**LCA**) of two nodes $u$ and $v$ in a rooted tree $T$ is the node $w$ that is ancestor of both $u$ and $v$ and that has the greatest depth in $T$. The Tarjan's LCA algorithm is like a depth-first tree walk.

**Depth-first tree walk**:
1. Visit node $v$: $S[v] \leftarrow Make - Set(v)$
2. Return from $v'$ to $v$: $S[v] \leftarrow Union(S[v].S'[v])$
3. Encounter $(u, v)$ at $v$ and $u \prec v$: $LCA(u, v) = Find - Set(u)$

Basically every time this algorithm encounters a new node in its depth-first tree walk, it makes a new set that contains only the new node. When it returns from a leaf and encounters a node for the second time, it joins the node into the set that contains all nodes that it has encountered twice. The least common ancestor of two nodes ($u$, $v$) and $u$ precedes $v$ will be the node returned by $Find\text{-}Set(u)$ when the set $v$ belongs to encounters the set $u$ is in.

# 5 SP-Bags Algorithm

During the execution of the SP-bags algorithm, two "bags" of procedure ID's are maintained for every Cilk procedure on the call stack, as illustrated in figure 6. These bags have the following contents:

- The **S-bag**, $S_F$, of a procedure $F$ contains the ID's of those descendants of $F$'s completed children that logically "precede" the currently executing thread, as well as the ID for $F$ itself.
- The **P-bag**, $P_F$, of a procedure $F$ contains the ID's of those descendants of $F$'s completed children that operate logically "in parallel" with the currently executing thread.

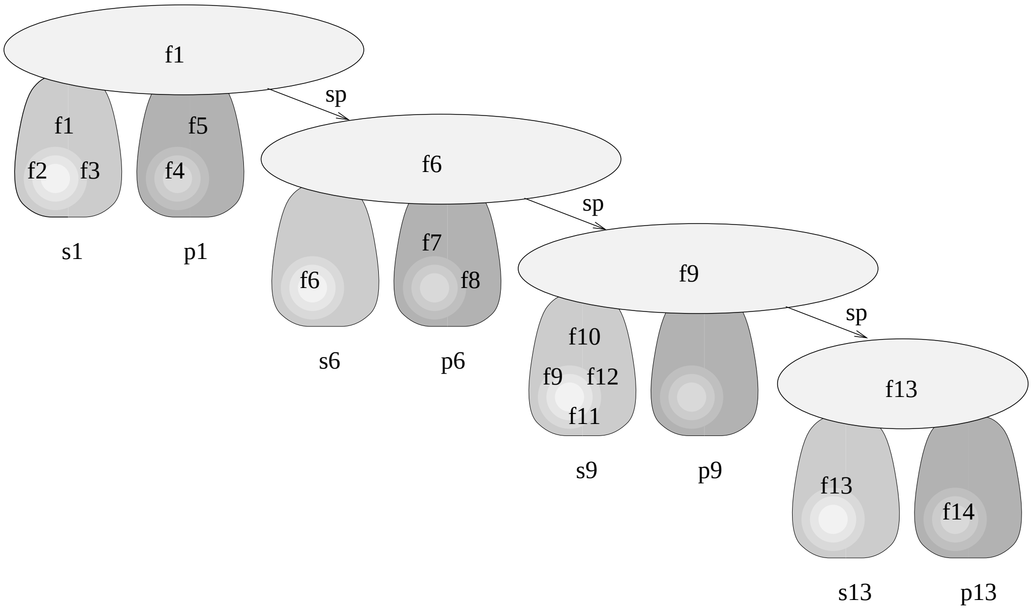


**Figure 6**: A snapshot of the SP-bags data structures during the execution of a Cilk program. The ovals in the figure represent procedures that are currently on the runtime stack: $F_1$ spawns $F_6$, which spawns $F_9$, which spawns $F_{13}$. Each procedure contains an S-bag and a P-bag. Each descendant of a completed child of a procedure $F$ belongs either to $F$'s S-bag or to $F$'s P-bag. For example, $F_2$, $F_3$, $F_4$, and $F_5$ are descendants of $F_1$ that complete before $F_1$ spawns $F_6$, and so these procedures belong to either $F_1$'s S-bag or its P-bag. In addition, every procedure $F$ belongs to its own S-bag.

All S-bags and P-bags are represented as sets using a disjoint-set data structure. The SP-bags algorithm maintains two shadow spaces of shared memory called writer and reader. For each location of shared memory, each shadow space has a corresponding location. Every spawned procedure is given a unique ID at runtime. For each location $l$ in shared memory, the ID of the procedure that wrote the location is stored in location $l$ of the writer shadow space. Similarly, location $l$ of the reader shadow space stores the ID of a procedure which previously read location $l$, although in this case, the ID is not necessarily that of the most recent reader. The SP-bags algorithm updates the shadow spaces as it executes.

The SP-bags algorithm itself is given at figure 7. The key idea of the SP-bags algorithm is that a procedure $F$'s completed descendants are in $F$'s S-bag if they precede the current;y executing thread, and they are in $F$'s P-bag if they could execute in parallel. Whenever a procedure $F$ is spawned, the S-bag of $F$ is made to contain just $F$, because $F$'s subsequent instructions are in series with its earlier instructions. Whenever a procedure $F'$ returns to its parent $F$, the contents of $S_{F'}$ are emptied into $P_F$, $F$'s P-bag, since the procedure in $S_{F'}$ can execute in parallel with any sub-procedures that $F$ might spawn in the future before it performs a sync. When a sync occurs, $F$'s P-bag is emptied into its S-bag, since all the

```
spawn procedure F:
    S_F ← Make-Set(F)
    P_F ← ∅

sync in a procedure F:
    S_F ← Union(S_F, P_F)
    P_F ← ∅

return from procedure F' to F:
    P_F ← Union(P_F, S_F')
-------------------------------------------
write a shared location l by procedure F:
    if Find-Set(reader(l)) is a P-bag
          or Find-Set(writer(l)) is a P-bag
      then a determinacy race exists
    writer(l) ← F

read a shared location l by procedure F:
    if Find-Set(writer(l)) is a P-bag
      then a determinacy race exists
    if Find-Set(reader(l)) is an S-bag
      then reader(l) ← F
```

**Figure 7**: The SP-bags algorithm. Whenever one of the five actions occurs during the serial, depth-first execution of a Cilk program, the operations in the figure are performed. Operations for `spawn`, `sync` and `return` actions manipulate the S-bags and P-bags of the disjoint-set data structure. Operations for `write` and `read` actions affect the shadow spaces and detect determinacy races.

procedures $F$ previously spawned logically preceded any future procedures spawned by $F$.

Determinancy races are detected by the code for write and read. A race occurs whenever a procedure $F$ write a location $l$ and discovers that either the previous reader or the previous writer of $l$ belongs to a P-bag, which means that $F$ and the past ancestor operate logically in parallel, If no determinacy race occurs whenever $F$ reads a location $l$ and discovers that the previous writer is in a P-bag. If no determinacy race occurs, the reader of $l$ is updated to be $F$, but only if the previous reader operateds logically in series.

# 6 Acknowledgement

Most of the figures for this lecture note are extracted from Mingdong Feng's paper on the Efficient Detection of Determinacy Races in Cilk Programs.