Lecture 13: Correctness of Nondeterminator

In this lecture we shall prove the correctness of the nondeterminator. We shall prove that the nondeterminator signals a race if and only if a race exists. We start by proving some structural lemmas about series-parallel dags.

Lemma 1 *If three threads e_1, e_2 and e_3 execute serially in order, then $e_1 \prec e_2$ and $e_1 \parallel e_3$ implies $e_2 \parallel e_3$.*

Proof. By contradiction: suppose $e_2 \prec e_3$, then $e_1 \prec e_3$, because $\prec$ is transitive. The contradiction proves the lemma. $\square$

In the above proof we used the transitivity of $\prec$. Note that $\parallel$ is not transitive, $e_1 \parallel e_2$ and $e_2 \parallel e_3$ does not necessarily mean $e_1 \parallel e_3$. However, a weaker form of transitivity holds for $\parallel$:

Lemma 2 (Pseudotransitivity) *If three threads e_1, e_2 and e_3 execute serially in order, then $e_1 \parallel e_2$ and $e_2 \parallel e_3$ implies $e_1 \parallel e_3$.*

Proof. The condition that e_1, e_2 and e_3 execute serially in order implies that there are only 2 possible forms of the parse tree:

Let $a_1 = \text{LCA}(e_1, e_2)$ and $a_2 = \text{LCA}(e_2, e_3)$. By hypothesis, we know that both a_1 and a_2 are P nodes. In both cases, therefore, $\text{LCA}(e_1, e_3)$ is a P node. Thus e_1 and e_3 execute in parallel order. $\square$

1 Analysis of the SP-bags Algorithm

For a description of the SP-bags algorithm, please refer to the previous lecture notes.

1.1 Set Manipulation

The sets in the SP-bags algorithm are updated as follows:

Spawn F : $S_F \leftarrow \text{Make\_Set}(F)$, and $P_F \leftarrow \emptyset$.

Sync in procedure F : $S_F \leftarrow \text{Union}(S_F, P_F)$, and $P_F \leftarrow \emptyset$.

Return from F' **to** F : $P_F \leftarrow \text{Union}(P_F, S_{F'})$.

We postulate a "*Procedurification*" function h that maps parse tree nodes to procedures. The function h is needed for making a correspondence between the algorithm (expressed in terms of procedures) and the lemmas proven in the last section (expressed in terms of nodes in the tree).

Figure 1: Case 1

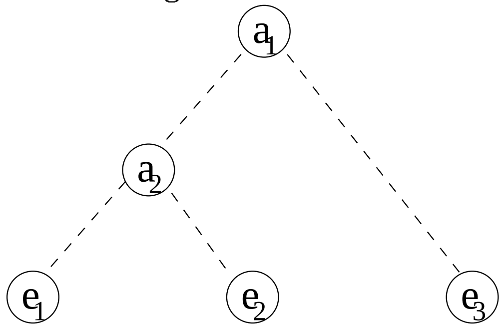
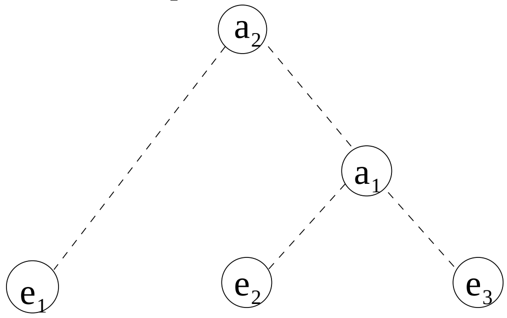


Figure 2: Case 2



Lemma 3 *If e_1 executes before e_2 serially, and $a = LCA(e_1, e_2)$ in the parse tree, then*

$$e_1 \prec e_2 \Rightarrow h(e_1) \in S_{h(a)} \quad \textit{when } e_2 \textit{ executes },$$

and

$$e_1 \parallel e_2 \Rightarrow h(e_1) \in P_{h(a)} \quad \textit{when } e_2 \textit{ executes }.$$

Proof. We prove the two statements separately.

Statement 1. If $e_1 \prec e_2$, then $a = \mathrm{LCA}(e_1, e_2)$ is an S-node. The parse tree for a Cilk computation has the general form shown in Figure 3: there is a **spine**, which is a sequence of **sync blocks**. Sync blocks can spawn procedures but are always terminated by a sync.

The LCA a may be in the spine or in a sync block. If it is in the spine, then e_1 belongs to a's left subtree, and e_2 belongs to a's right subtree. When e_2 executes, which set does $h(e_1)$ belong to? In order to answer the previous question, we argue as follows. When e_1 returns, $h(e_1)$ is moved into the P-bag of the parent. When the sync block containing e_1 syncs, $h(e_1)$ is placed in the S-bag and never moved thereafter. So we can see that at this case, when e_2 executes, e_1 will be in P-bag of a.

If the LCA a is in a sync block, then e_1 belongs to some spawned procedure F. We have that $h(e_1) = h(a) = F$. F is $\in S_F$ when F spawned, and in never moved before the completion of e_2. So in this case when e_2 executes, $h(e_1)$ will also be in $S_{h(a)}$.

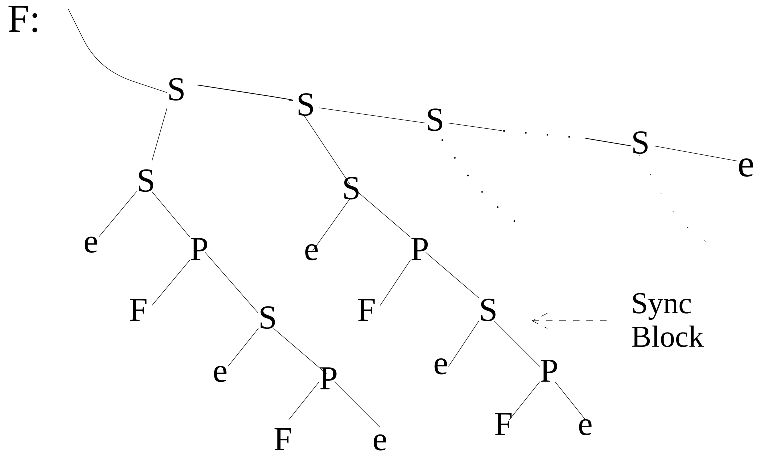


Figure 3: Form of the parse tree for a Cilk program

Statement 2 If $e_1 \parallel e_2$, then $a = \text{LCA}(e_1, e_2)$ is a P-node. Since all nodes in spine are S-nodes, so a must be in a sync block. e_1 will be in the left subtree of a and e_2 in the right subtree of a. Note that the left subtree of a P-node is just a spawned procedure (In this case, it is $h(e_1)$), so when $h(e_1)$ returns to $h(a)$, it will be put in the P-bag of a, and the execution continues on the right subtree of a. Thus, when $h(e_2)$ executes, $h(e_1)$ will be in $P_{h(a)}$. (Since $h(a)$ will not pass the sync until all its spawned procedures have completed.)
$\square$

1.2 Correctness of the Nondeterminator

The following is the actual race-detecting pseudocode used by the nondeterminator:

```
Write(l) by procedure F:
        If Find-set(reader(l)) is a P-bag or
           Find-set(writer(l)) is a P-bag
                Then race
        writer(l) ← F

Read(l)  by procedure F:
        If Find-set(writer(l)) is a P-bag
                Then race
        If Find-set(reader(l)) is a S-bag
                Then reader(l) ← F
```

We claim that the above race-detecting code is actually doing what it is designed to do, i.e., we have the following theorem:

Theorem 4 (Correctness of the Nondeterminator) *A race will be detected if and only if a race exists.*

Proof. The "$\Rightarrow$" direction is rather straightforward, since if a race is detected, it basically says that at some point, two parallel procedures are trying to *Concurrent write* or *Concurrent write/read* some memory location, thus it will produce a determinancy race.

For the "$\Leftarrow$" direction: Assume a buggy parallel program has determinancy between 2 procedures $h(e_1)$ and $h(e_2)$ at some memory location l.(e_1 and e_2 are the two procedures' corresponding parse tree nodes.)

W.l.o.g, we can assume that when run by nondeterminator, $h(e_1)$ runs before $h(e_2)$.

If several races exist at the same memory location l, choose e_2 to be the parse tree node that corresponds to the first procedure after $h(e_1)$ which has a race with $h(e_1)$ at l.

There are several possible race types:

e_1 **writes** l **and** e_2 **reads** l . When e_2 is executed, suppose that $writer(l) = h(e)$ for some thread e. If $e = e_1$, then since $e_1 \parallel e_2$, $writer(l)$ belongs to a P-bag and the determinacy race is reported. If $e \neq e_1$, however, then e must be executed after e_1 but before e_2, because otherwise e's write to l would be overwritten by e_1's write, and $writer(l)$ would likewise be overwritten. We have two possibilities: either $e_1 \prec e$ or $e_1 \parallel e$. If $e_1 \prec e$, then we must have $e \parallel e_2$. Consequently, $h(e) = writer(l)$ belongs to a P-bag, and the determinacy race between e and e_2 is detected. If $e_1 \parallel e$, however, then since both e_1 and e write l, a write/write determinacy race exists between e_1 and e, contradicting the assumption that e_2 executes earliest in the depth-first execution order of the program, over all determinacy races on location l.

$h(e_1)$ **writes** l**,** $h(e_2)$ **writes to** l Note that in the proof of the previous case we never used the fact that $h(e_2)$ is writing l, so the above proof can be used here with no change.

$h(e_1)$ **reads** l **and** $h(e_2)$ **writes** l In this case, a little more analysis is needed, please refer to the paper handed out at the detail of the proof. *Idea of the proof*: Let $h(e) = reader(l)$. If $e_1 \parallel e$, then $reader(l)$ will be $h(e)$ when $h(e_2)$ executes, but that's okay since we can catch the race between $h(e)$ and $h(e_2)$.(by pseudotransitivity, we know that there is a race between them.) If $e_1 \prec e$, then $h(e_2)$ and $h(e)$ has a race by lemma 1, so race will also be detected.

$\square$

2 Disjoint-Set Data Structure

SP-bags algorithm is built on disjoint-set data structures. In this section we discuss a clever implementation of such a data structure.

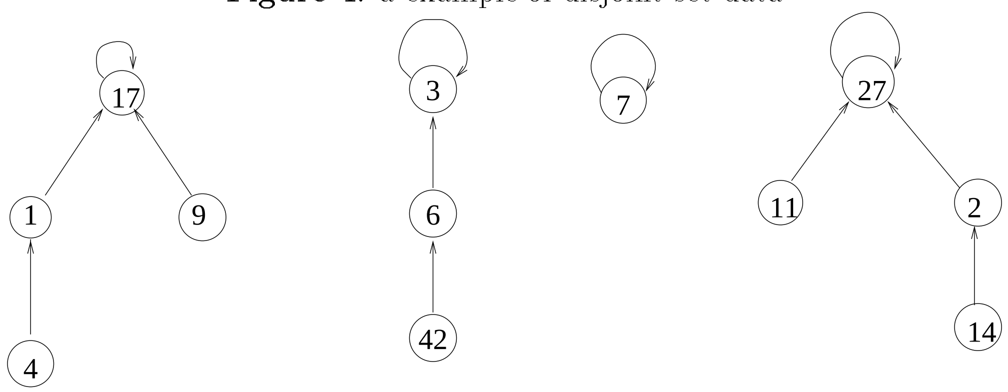
A **disjoint-set data structure** maintains a collection $S = \{S_1, S_2, ..., S_k\}$ of disjoint dynamic sets. Each set is identified by a **representative**, which is some member of the set. The disjoint-set data structure supports the following operations.

Make-Set(x) creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x be not already in a set.

Union(x,y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of this two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$. Since we require that the sets in the collection to be disjoint, we "destroy" sets S_x and S_y, removing them from collection S.

Find-Set(x) returns a pointer to the representative of the (unique) set containing x.

Figure 4: a example of disjoint-set data



Example 1 Let

$$\mathcal{A} = \{\{1, 4, 9, 17\}, \{3, 6, 42\}, \{7\}, \{11, 27, 2, 14\}\}$$

This is very simple set of disjoint sets of data.

Now we study some implementation details of the operations surported by disjoin-set data structure.

Naive implementation of Find-set and Union:

Find-Set(X) $\Rightarrow$ follow pointers to the root (The set representative)

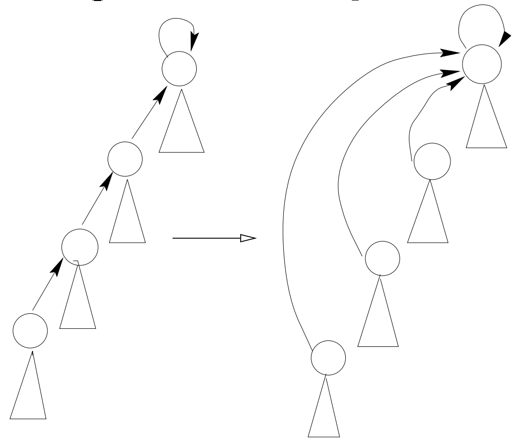
Union(X,Y) $\Rightarrow$link(Find-Set(X),Find-Set(Y)) (redirect Find-Set(x)'s parent pointer to Find-Set(Y))

In the worst case, n operations take $\Theta(n^2)$ time. (Obviously this is a bad approach)

We can improve the algorithm by 2 *heuristics*,

heuristic 1 Always make smaller tree points to larger one. "Small" generally means height or size of the tree, but either one is hard to implement, so we will use an approximation.

Figure 5: Path Compression



heuristic 2 *Path Compression* : After tracing pointers up tree, make all pointers along the way point to the root of the tree.

Either heuristic alone will make m operation operate in $O(m \log n)$ time, but when we use them together, will use only $O(m\alpha(m,n))$ time.

We imagine that every tree has a **rank**, which is an approximation of its size. Things work properly if we "union by rank". Here is the union by rank procedure:

```
Link(X,Y):
          If (rank[X]>rank[Y])
                then p[Y]←X
                else p[X]←Y
                        if (rank[X]=rank[Y])
                             then rank[Y]←rank[Y]+1
Initialization in Make-Set: rank[X]=0 for all X
```

We state the following lemma without proof:

Lemma 5 *If X is the root of a set, then $height[X] \leq rank[X] \leq \log(size(X))$*

Corollary 6 *m operations operate in $O(m \log n)$ time.*

(Timeout here—read CLR Chapter 22 for more details)