6.895 Theory of Parallel Systems Lecture 4

Debugging Multithreaded Programs

Lecturer: Michael Bender *Scribe: Sid Sen and Jim Sukha*

Lecture Summary

1. *Determinacy Races*
   This section introduces the concept of a determinacy race in parallel programs, and provides two examples of Cilk programs where such races occur.
2. *Introduction to the Nondeterminator*
   This section introduces the Nondeterminator, discusses its performance, and describes how Cilk programs can be represented using series-parallel directed acyclic graphs and series-parallel parse trees.
3. *The SP-Bags Algorithm*
   This section presents the SP-Bags algorithm, the algorithm used by the Nondeterminator to detect determinacy races in Cilk programs.
4. *Nondeterminator Correctness*
   This section presents a proof of correctness for the SP-Bags algorithm for the Nondeterminator.

1 Determinacy Races

In a parallel program, two threads running concurrently may want to access the same memory location. If the order of the accesses is not well defined, this conflict is a determinacy race, which can lead to incorrect program behavior. This section defines a determinacy race and analyzes two Cilk programs that have determinacy races.

1.1 A Cilk Program with a Determinacy Race

In this section, we provide an example of a simple Cilk program with a determinacy race. We consider the following Cilk program:

```
int x;

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

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

a)

Thread 1	Thread 2
read x	
write x	
	read x
	write x

b)

Thread 1	Thread 2
read x	
	read x
write x	
	write x

Figure 1: Two possible execution interleavings of the spawned threads in our simple Cilk program: (a) At the end of this sequence, the value of x is 2. (b) At the end of this sequence, the value of x is 1.

If a thread updates a location while another thread is concurrently accessing the location, then we say that a ***determinacy race*** occurs. In the example program, the two threads that are spawned are both trying to modify the value of x.

Each increment of x involves both a read and a write to x's memory location. Figure 1 shows two possible ways the executions of the two threads can overlap. Depending on how these read and write operations are interleaved, at the end of the program, x may have a value of 1 or 2.

1.2 Determinacy Races in Cilk

In this section, we formally define a determinacy race in the context of Cilk programs.

Definition 1 (Determinacy Race) *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.*

We can have read/write races or write/write races, as illustrated in Figure 2.

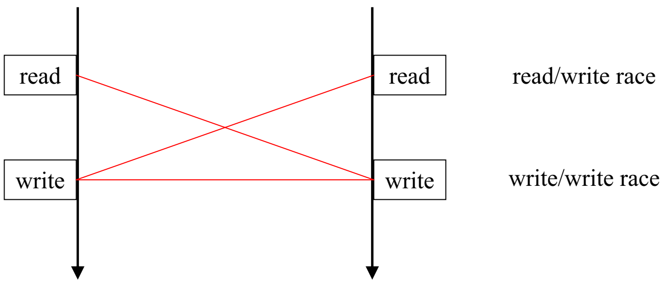


Figure 2: Two types of determinacy races.

In general, a determinacy race can cause a multithreaded parallel program to act nondeterministically. In most cases, programmers intend to write deterministic code, so a determinacy race is usually a bug.

However, it is still possible for a correct program to be nondeterministic. Suppose that the code in Section 1.1 has another procedure `bar` that decrements x, and assume that we are somehow guaranteed that both `foo` and `bar` execute atomically, so that operations from the two procedures are never interleaved. Then, if we spawn one thread executing `foo` and another executing `bar`, there is a determinacy race because we do not know which thread executes first. The resulting program is nondeterministic, but in this case we really do not care because the order of execution does not matter—that is, the resulting value in x is the same either way, making the output of the program deterministic.

The following definition summarizes the difference between these two situations:

Definition 2 (Internal and External Determinism) *A program is* ***internally deterministic*** *if the program has no determinacy races. A program is* ***externally deterministic*** *if the program has determinacy races but the output is deterministic.*

1.3 Example: the N-Queens Puzzle

The example in Section 1.1 showed a determinacy race that was fairly easy to spot. In more complicated programs, however, finding determinacy races can be much more difficult. In this section, we consider a subtle determinacy race in a program that solves the N-Queens puzzle.

Definition 3 (N-Queens Puzzle) *Find a configuration of N queens on an N by N chessboard such that no two queens attack each other.*

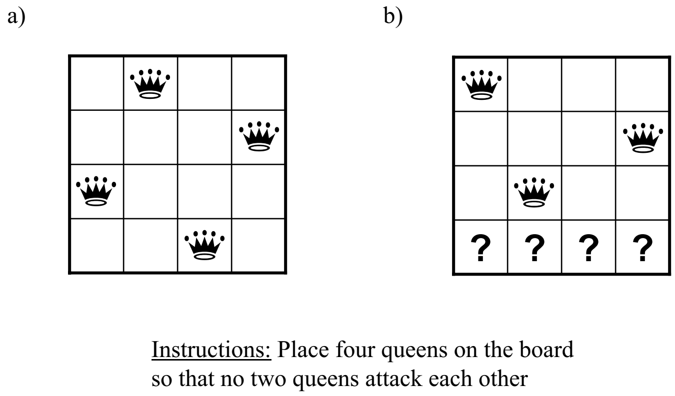


Figure 3: N-Queens problem for a 4 by 4 chessboard ($N = 4$): (a) A valid configuration. (b) An invalid configuration.

Figure 3 illustrates a valid and invalid solution to the N-Queens puzzle for the specific case when $N = 4$.

One common approach to solving the N-Queens puzzle makes use of a recursive backtracking algorithm, where queens are placed row by row (starting with the first row) and we backtrack to the previous row whenever a valid configuration of queens cannot be found for the current row.

An abbreviated version of this program is shown below (see Figure 4 for an explanation of the `board` and `newboard` parameters).

```
1:  cilk char *nqueens(char *board,         // current configuration of queens
2:                     int n,               // board size
3:                     int row) {           // row number where queen is
                                            // placed next

4:    char *new_board;
5:    ...
6:    new_board = malloc(row+1);            // malloc new space for child's board
7:    memcpy(new_board, board, row);        // copy existing board into new board
8:    for (j = 0; j < n; j++) {             // consider all places for new queen
9:      ...
10:     new_board[row] = j;                 // assign row queen to jth column
11:     spawn nqueens(new_board, n, row+1); // proceed to (row+1)th row
12:     ...
13:   }
14:   sync;
15:   ...
16: }
```

To highlight the data race, we include only the relevant sections of code (steps such as checking for a valid board configuration are omitted). Figure 4 illustrates the representation of the chessboard used in the program for the specific case where $N = 4$.

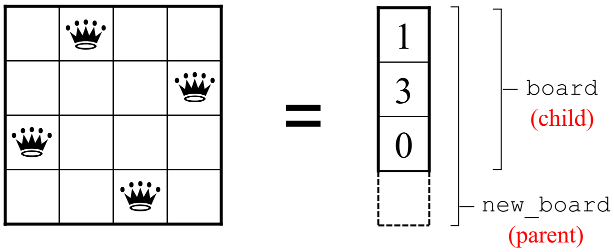


Figure 4: Representation of a 4 by 4 chessboard in the `nqueens` procedure.

When a parent procedure spawns a call to `nqueens` in line 11, it passes `new_board` to the child. However, a conflict may arise in the use of this board data when the child reads the `board` variable in line 7, since the parent writes to the board in the next iteration of the for loop (line 10). Thus, the old board may be updated by the parent before the child has a chance to copy it over into the new board, resulting in a determinacy race.

2 Nondeterminator Background/Theory

As the previous examples show, determinacy races in programs are usually bugs that a programmer should eliminate. Cilk has a tool called the Nondeterminator that helps programmers detect determinacy races. In this section, we introduce the Nondeterminator, discuss its performance, and describe how to represent Cilk programs using series-parallel directed acyclic graphs (or ***dag***s) and series-parallel parse trees. These abstractions will help us when we explain the execution of the Nondeterminator and prove its correctness in Sections 3 and 4.

2.1 Definition and Performance

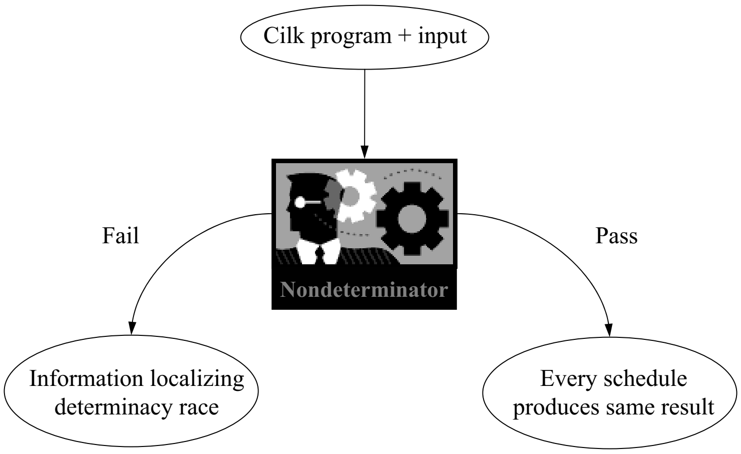


Figure 5: High-level behavior of the Nondeterminator.

Figure 5 illustrates the input-output behavior of the Nondeterminator. The Nondeterminator takes as input a Cilk program and a particular input data set to the program. The Nondeterminator passes the program if every scheduling of the computation produces the same result; otherwise, it fails the program and returns information for locating the determinacy race. It is important to note that the Nondeterminator is a debugging tool, not a program verifier. Although a Cilk program may pass on a particular data set, the same program may fail on other data sets.

The Nondeterminator has provable performance. The following theorem bounds its running time:

Theorem 4 (Nondeterminator Performance) *For a Cilk program that runs in T time serially and uses v shared-memory locations, the Nondeterminator runs in $O(T\alpha(v,v))$ time and uses $O(v)$ space, where α is Tarjan's functional inverse of Ackermann's function.*

Note that $\alpha(v,v)$ is much, much smaller than $\log^*(v)$, which is the number of times you need to take the log of v before you get a value below 1. For all practical purposes, $\alpha(v,v) \leq 4$.

We achieve this good performance for the Nondeterminator by exploiting the special structure of Cilk dags, as shown in the sections below.

2.2 Series-Parallel (SP) Dags

In this section, we show that Cilk programs can be represented as dags that are series-parallel. We say that a graph is ***series-parallel*** if it can be derived from a base graph using series and parallel compositions. The base graph and composition rules are summarized in the list below and illustrated in Figure 6.

- *Base Graph*: This graph is a single directed edge from a source node s to a sink node t.
- *Series Composition*: Given two series-parallel graphs, G_1 and G_2, with source and sink nodes (s_1, t_1) and (s_2, t_2), respectively, we compose the two graphs in series to form a new graph G with source node $s = s_1$ and sink node $t = t_2$. The series composition merges the sink node of G_1 with the source node of G_2, so that $t_1 = s_2$ in the resulting graph G.
- *Parallel Composition*: Given two series-parallel graphs, G_1 and G_2, with source and sink nodes (s_1, t_1) and (s_2, t_2), respectively, we compose the two graphs in parallel to form a new graph G with source node $s = s_1 = s_2$ and sink node $t = t_1 = t_2$.

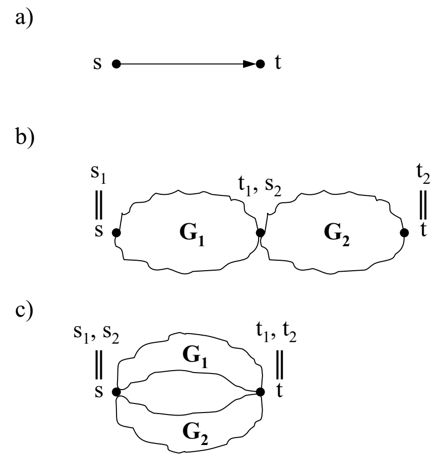


Figure 6: Series-Parallel Dags: (a) The base graph. (b) A series composition of graphs G_1 and G_2. (c) A parallel composition of graphs G_1 and G_2.

The execution of a Cilk program can be represented as an SP-dag, with threads corresponding to edges and spawn/sync steps corresponding to nodes. For example, consider a slightly modified version of the program in Section 1.1:

```
F:       cilk int main(void)
         {
    e0:     x = 0;
    F1:     spawn foo();
    e1:     // some additional serial code
    F2:     spawn foo();
    e2:     // some more serial code
            sync;
    e3:     printf("%d", x);
            return 1;
         }
```

Figure 7 illustrates the corresponding SP-dag for this program. The sections marked with the e_i identify the threads, while the F_i labels are function ID's for the Cilk procedures.

From our construction of the SP-dag above, it is not difficult to show the following theorem:

Theorem 5 (Cilk Dags) *A Cilk dag is a series-parallel dag with out-degree of at most 2 at each node.*

Proof Our proof is a simple extension of the proof for Theorem 3 in [1], which verifies the weaker statement that a Cilk dag is in fact a series-parallel dag. To show that each node in the resulting SP-dag has out-degree of at most two, we consider what happens when a `spawn` statement interrupts the control-flow of a given thread.

Every time a `spawn` statement is reached, we create a new node in our graph with two edges directed out. One edge corresponds to the spawned procedure, and the other corresponds to the continuation of the parent procedure (i.e. everything following the `spawn` statement). Both of these paths eventually converge at the node corresponding to the `sync` at the end of the parent procedure. Recall that the Cilk compiler inserts an implicit `sync` at the end of the parent procedure if none is included by the programmer.

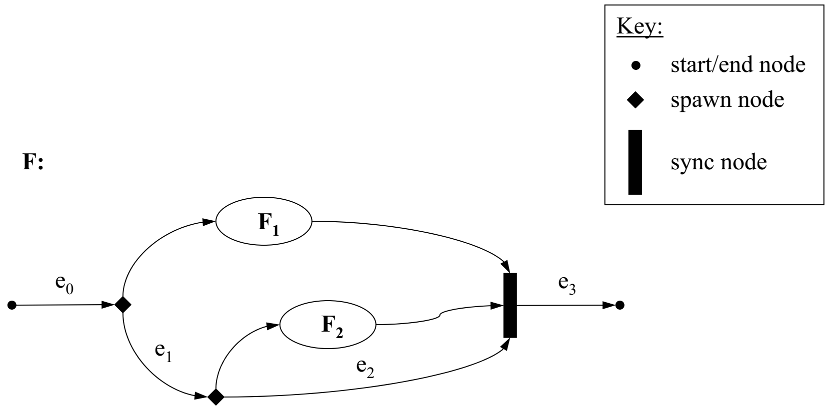


Figure 7: The SP-dag for procedure F.

Figure 7 illustrates the `spawn` and `sync` nodes clearly for procedure F above. A more complicated example (with multiple `sync` nodes) is shown in Figure 10. □

2.3 Series-Parallel (SP) Parse Trees

Another way of looking at the structure of Cilk procedures is in terms of series-parallel parse trees. In this section, we describe how to construct an SP parse tree for a given Cilk procedure.

In an SP parse tree, the leaves of the tree correspond to threads or (spawned) procedures, and the non-leaf nodes are either S-nodes or P-nodes. For S-nodes, everything in the left subtree is executed before everything in the right subtree. For P-nodes, everything in the left the left subtree can run logically in parallel with everything in the right subtree. Figure 8 shows an SP parse tree for procedure F in Section 2.2.

From [1], we recall that a sync block has the following form:

e_0; **spawn** F_0; e_1; **spawn** F_1; ... e_j; **sync** ;

To create an SP parse tree for a single sync block, first create a tree with root S and left child e_0. Set the right child of S to be a tree with root P, left child F_0, and right child equal to the subtree for the rest of the code up until the sync. Observe that the left child of an S-node in a sync block is always a thread, while the left child of a P-node is always a (spawned) procedure [1].

To create an SP parse tree for an entire Cilk procedure, string the parse trees for the sync blocks together using a ***spine*** of S-nodes (one S-node for each sync block). The left child of each S-node in the spine is the parse tree for the corresponding sync block, and the right child of the last S-node in the spine is the thread that immediately precedes the `return` statement of the procedure.

Figure 9 shows the SP parse tree for a generic Cilk procedure.

Following this construction of an SP parse tree, we see that a standard depth-first traversal of the parse tree visits the threads in the same execution order as if the program were run on a single processor (i.e., it follows a sequential execution of the procedure).

Note that since the S relationship is associative and the P relationship is commutative, it is possible to have multiple SP parse trees that are logically equivalent.

From the tree shown in Figure 9, we notice the following:

Observation 6 *The least common ancestor (LCA) of two threads determines whether the threads are logically in series or in parallel. We use the notation:*

F:

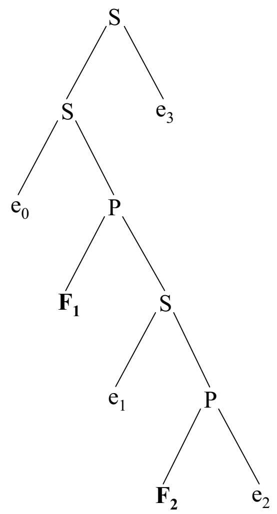


Figure 8: An SP parse tree for procedure F.

- $e \prec e'$ *if* $LCA(e, e')$ *is* S *and* e *is to the left of* e'*.*
- $e \parallel e'$ *if* $LCA(e, e')$ *is* P*.*

Every SP-dag has a corresponding SP parse tree. These representations of Cilk procedures help us explain the execution of the Nondeterminator and prove its correctness later on.

3 Execution of the Nondeterminator (SP-Bags Algorithm)

In this section, we describe the algorithm and data structures used by the Nondeterminator to detect determinacy races in Cilk programs.

3.1 Overview

The following is a high-level overview of the Nondeterminator's execution. It also introduces some of the concepts we discuss in this section.

- The `cilk2c` compiler instruments every load and store operation in the user program.
- While the user program executes, reader and writer ***shadow spaces*** are used to keep track of accesses to memory. We discuss this in more detail in Section 3.4.
- An SP-bags data structure based on Tarjan's least-common-ancestor algorithm maintains the series-parallel relationship between the threads. This data structure is based on the disjoint-set data structure discussed in Section 3.3.
- When a determinacy race is discovered, file names, line numbers, and the conflicting variables are reported.

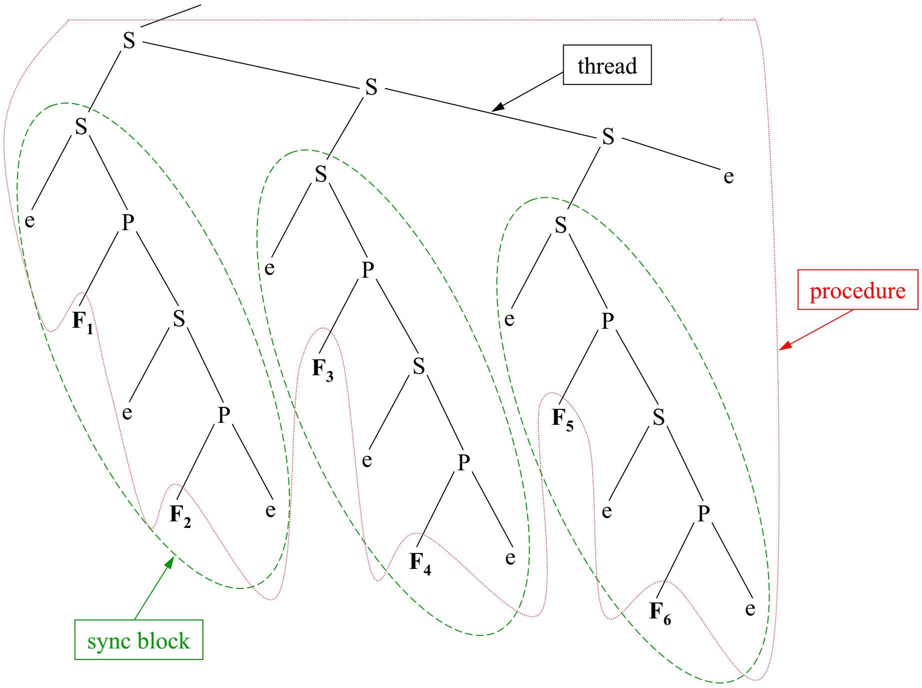


Figure 9: A detailed SP parse tree for a cilk procedure; threads are represented as edges, synchronization blocks are circled in green (dashed line), and the top-level procedure is circled in red (dotted line). Each F_i leaf represents an SP parse tree for another cilk procedure.

3.2 Properties of the Nondeterminator

Although the Nondeterminator is used to analyze parallel programs, the Nondeterminator itself is a sequential program: it executes tasks in the DFS order of the program's SP parse tree. At any given time in the Nondeterminator's execution, we can define the ***current thread*** to be the thread that is currently executing in the sequential DFS order.

The execution of the Nondeterminator maintains the following invariant: as the nondeterminator executes, it maintains two "bags" for each procedure on the call stack:

- S-Bag S_F – Contains the IDs of F's completed descendants (including F) that logically precede the current thread.
- P-Bag P_F – Contains the IDs of F's completed descendants that operate logically in parallel with the current thread.

Since the current thread changes over time, the contents of S_F and P_F also change.

3.2.1 Illustration of S-Bags and P-Bags

In this section, we analyze a sample Cilk procedure to better understand the notions of S-bags and P-bags.

Figure 10 shows the SP-dag for a Cilk procedure, which we call F_0 for ease of discussion. When the current thread being executed is e_1, the completed descendants of F_0 that logically precede e_1 are the procedures that were spawned prior to the most recent sync. Thus, $S_F = \{F_1, F_2, F_3\}$. Since there are no descendants of F_0 that have been spawned after the last sync (assuming the current thread is e_1), $P_F = \emptyset$.

When the current thread is e_2, the completed descendants of F_0 that logically precede e_2 are still the procedures that were spawned prior to the last sync. So $S_F = \{F_1, F_2, F_3\}$. However, the procedures that have been spawned after the last sync now operate logically in parallel with e_2 (since we have not reached the sync node), and thus $P_F = \{F_4, F_5, F_6\}$.

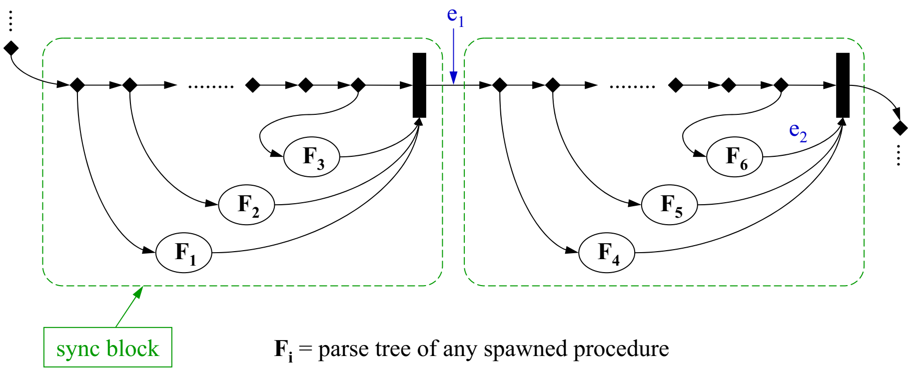


Figure 10: An SP-dag for a Cilk procedure.

3.3 Disjoint-Set Data Structures

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

Definition 7 (Disjoint-Set Data Structure (UNION-FIND)) Union-Find *maintains a collection* Σ *of disjoint sets. So for two sets* X *and* Y*,* $X, Y \in \Sigma \Rightarrow X \cap Y = \emptyset$*. 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* Σ *subject to the following operations:*

- Make-Set(e)*:* $\Sigma \leftarrow \Sigma \cup \{\{e\}\}$*. This operation adds a new set* $\{e\}$ *to* Σ*.*
- Union(X, Y)*:* $\Sigma \leftarrow \Sigma - \{X, Y\} \cup \{X \cup Y\}$ *This removes the individual sets* X *and* Y *from* Σ *and replaces them with the union of* X *and* Y*.*
- Find-Set(e)*: Returns* $X \in \Sigma$ *such that* $e \in X$*. Since the sets in* Σ *are typically "named" by a distinguishing element,* Find-Set(e) *means "take me to your leader".*

The following theorem describes the performance of the above operations on the disjoint-set data structure:

Theorem 8 (Operations on Disjoint-Set Data Structure (Tarjan 1975)) *Any sequence of* m *operations on* n *sets can be performed in* $O(m\alpha(m, n))$ *time.*

3.4 SP-Bags Algorithm

The SP-Bags algorithm is the algorithm used by the Nondeterminator. It uses the disjoint-set data structure from the Section 3.3 to maintain the S and P-bags for all procedures in the call stack.

The SP-Bags algorithm has two types of operations. The first type updates the S and P-bags of the procedures to maintain the data structure invariant from Section 3.2. These operations are triggered during the DFS traversal of the Cilk procedure's SP parse tree; there is a separate operation for each type of node that is encountered in the corresponding SP-dag:

- `spawn` procedure F: $S_F \leftarrow \text{MAKE-SET}(F)$; $P_F \leftarrow \emptyset$
- `sync` in a procedure F: $S_F \leftarrow \text{UNION}(S_F, P_F)$; $P_F \leftarrow \emptyset$
- `return` from F' to F: $P_F \leftarrow \text{UNION}(P_F, S_{F'})$

The second type of operation uses the data structure to find determinacy races when the user program is accessing a memory location. As a Cilk program is executed, each shared memory location ℓ has two corresponding ***shadow locations*** that are updated by the S-P bags algorithm:

- WRITER[ℓ]: the ID of the last procedure that wrote ℓ
- READ[ℓ]: the ID of a procedure that read ℓ (not necessarily the most recent reader)

The operations of the second type make use of these definitions:

- write location ℓ by procedure F:

  if (FIND-SET(READER[ℓ]) is a P-bag
          or FIND-SET(WRITER[ℓ]) is a P-bag)
      then determinacy race exists
  WRITER[ℓ] $\leftarrow F$

- read location ℓ by procedure F:

  if FIND-SET(WRITER[ℓ] is a P-bag)
      then determinacy race exists
  if FIND-SET(READER[ℓ] is an S-bag)
      then READER[ℓ] $\leftarrow F$

4 Nondeterminator Correctness

In this section, we prove the correctness of the Nondeterminator and the S-P bags algorithm. We begin by justifying the set manipulation operations described in Section 3.4 and proving some useful lemmas.

4.1 Justification of Set Manipulation in SP-Bags Algorithm

We show that the set manipulation operations described in Section 3.4 (the first type of operation in the SP-bags algorithm) maintain the data structure invariant from Section 3.2.

- `spawn` procedure F: $S_F \leftarrow \text{MAKE-SET}(F)$; $P_F \leftarrow \emptyset$

  Proof This operation is valid since the S-bag of F by definition contains itself, and F has no valid children yet. □

- `sync` in a procedure F: $S_F \leftarrow \text{UNION}(S_F, P_F)$; $P_F \leftarrow \emptyset$

  Proof After a `sync` operation, we switch current threads, from some thread e before the `sync` to the single thread e' after the `sync`. Originally, P_F contains the procedures that were logically parallel to e, but when we switch to e', these procedures now logically precede the current thread (and any future procedures spawned by F). Therefore, we move the elements from P_F into S_F. □

- `return` from F' to F: $P_F \leftarrow \textsc{Union}(P_F, S_{F'})$

  Proof Before a function returns, there is an implicit sync, so we know that $P_{F'}$ is empty, and $S_{F'}$ contains all the logical descendants of F'. But all logical descendants of F' are also descendants of F, and can now execute in parallel with any procedures that F might spawn in the future (before performing a sync). □

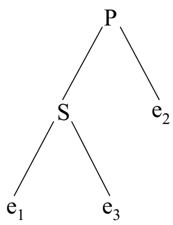
4.2 Some Lemmas

To help us prove the correctness of the second type of operation in the SP-bags algorithm (Section 3.4), we first prove some useful lemmas. We use the symbol $\prec$ to indicate an S relationship between two threads; that is, $e_1 \prec e_2 \Rightarrow LCA(e_1, e_2)$ is an S-node. Similarly, we use the $\|$ symbol to indicate a P relationship between two threads; that is, $e_1 \| e_2 \Rightarrow LCA(e_1, e_2)$ is a P-node.

Lemma 9 *Let threads e_1, e_2, and e_3 execute serially in order. If $e_1 \prec e_2$ and $e_1 \| e_3$, then $e_2 \| e_3$.*

Proof Suppose for sake of contradiction that $e_2 \prec e_3$. Then, by transitivity, $e_1 \prec e_3$, which is a contradiction. □

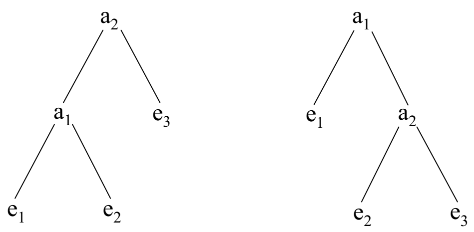
Is the $\|$ relation transitive? By considering a simple parse tree, we see that the answer is no:



In the tree above, $e_1 \| e_2$ and $e_2 \| e_3$, but $e_1 \nparallel e_3$. However, the following lemma shows that the $\|$ relationship has a weaker property called ***pseudotransitivity***.

Lemma 10 (Pseudotransitivity of $\|$) *Let threads e_1, e_2, and e_3 execute serially in order. If $e_1 \| e_2$ and $e_2 \| e_3$, then $e_1 \| e_3$.*

Proof Since we do a depth-first traversal of the parse tree, the only possible options for the tree that enable us to execute the three threads in order are:



In both of the cases above, we see that $a_1 = LCA(e_1, e_2)$ and $a_2 = LCA(e_2, e_3)$. From these observations, we conclude that a_1 and a_2 must both be P-nodes. Since $LCA(e_1, e_3)$ is either a_1 or a_2, $e_1 \parallel e_3$. □

Before stating the next lemma, we define the "procedurification" function h, taken from [1]. From our construction of an SP parse tree for a given Cilk dag (described in Section 2.3), we see that each procedure in the spawn tree is represented by an assembly of threads and internal nodes in the parse tree [1]. The procedurification function h maps threads or nodes in the parse tree to procedures in the spawn tree.

The next lemma makes use of the procedurification function to relate the S and P-nodes of an SP parse tree to the contents of the S and P-bags during the execution of the SP-bags algorithm.

Lemma 11 *Let e_1 be executed before e_2 serially, and let node $a = LCA(e_1, e_2)$ in the parse tree.*

- $e_1 \prec e_2 \Rightarrow h(e_1) \in$ S-bag$(h(a))$ when e_2 executes (i.e. is the current thread).
- $e_1 \parallel e_2 \Rightarrow h(e_1) \in$ P-bag$(h(a))$ when e_2 executes.

Sketch of Proof We present a sketch the proof here. For more details, please refer to Lemma 8 in [1].

Case 1: $e_1 \prec e_2 \Rightarrow a$ is an S-node.

1. If a belongs to the spine, then e_1 belongs to a's left subtree and e_2 belongs to a's right subtree. When e_2 is executed, which bag is $h(e_1)$ in? From our construction of the tree, either $h(e_1) = h(a)$ or $h(e_1)$ is a descendant of $h(a)$. Between the execution of e_1 and the execution of e_2, operations that move $h(e_1)$ are `sync` and `return`. This implies that we never move the procedure ID down the spawn tree; instead, $h(e_1)$'s ID moves up whenever any of its ancestors returns.
2. If a belongs to a sync block, then e_1 is in procedure F, and $h(e_1) = h(a) = F$. F is automatically placed in S_F when F is spawned.

Case 2: $e_1 \parallel e_2 \Rightarrow a$ is a P-node.

In this case, a must be in a sync block, since only S-nodes are on the spine. e_1 belongs to the left subtree of a. Note that the left child of a P-node is always a spawned procedure that is placed in P_F when F returns. At this point, a sync has not yet occurred, so $h(e_1)$ is in P_F when e_2 is executed.

□

4.3 Proof of SP-Bags Race Detection

In this section, we use the lemmas presented in Section 4.2 to prove the correctness of the SP-bags algorithm used by the Nondeterminator.

Theorem 12 *The SP-bags algorithm detects a determinacy race in a Cilk program if and only if a determinacy race exists.*

Sketch of Proof We provide a sketch of the proof here. For more details, please refer to Theorem 10 in [1].

The $\Rightarrow$ case is straightforward.

To prove the other direction, let $e_1 \parallel e_2$ have a race on ℓ, and assume e_1 executes before e_2. If there are several races on ℓ, choose e_2 to be determinacy race whose second thread executes earliest in the serial execution order. There are three possible types of determinacy races:

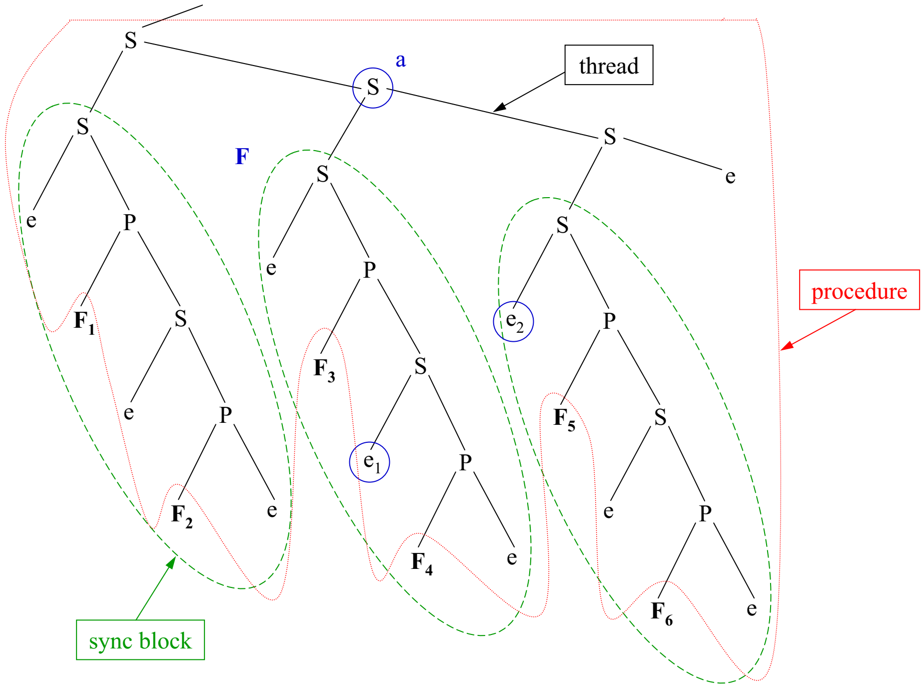


Figure 11: Case 1 in the proof of Lemma 11.

- Case 1: e_1 writes ℓ, e_2 reads ℓ

  Suppose when e_2 executes, $\textsc{writer}[\ell] = h(e)$ for some thread e. If $e = e_1$, we are done, since $h(e_1) \in$ P-bag$((LCA(e_1, e_2)))$. If $e \neq e_1$, then e is executed after e_1 and before e_2.

  1. $e_1 \prec e$: Then $e \parallel e_2$ by Lemma 9.
  2. $e_1 \parallel e$: Then a write/write race exists between e_1 and e, which contradicts the minimality of e.

- Case 2: e_1 writes ℓ, e_2 writes ℓ

  This case is similar to Case 1.

- Case 3: e_1 reads ℓ and e_2 writes ℓ

  This case is slightly more involved than the other two. Suppose when e_2 is executed, $h(e) = \textsc{reader}[\ell]$. The main idea is that if $e \parallel e_1$, then e_1 did not set $\textsc{reader}[\ell]$. However, this is okay since we catch $e \parallel e_2$ by pseudotransitivity (Lemma 10). If $e_1 \prec e$, then $e_1 \parallel e_2 \Rightarrow e \parallel e_2$.

$\square$

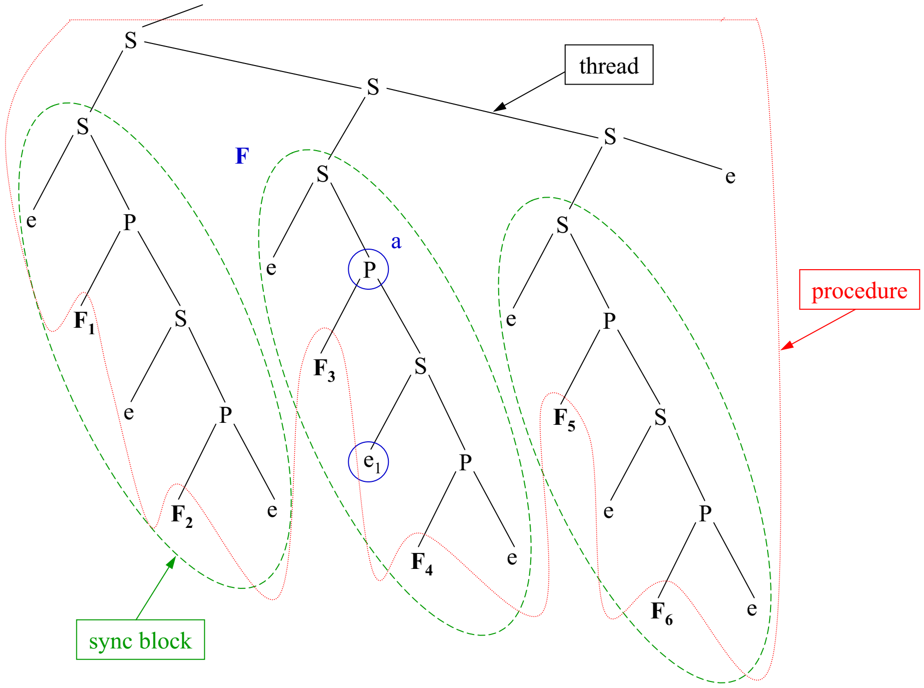


Figure 12: Case 2 in the proof of Lemma 11.

References

[1] C. E. Leiserson and M. Feng. Efficient detection of determinacy races in cilk programs. In *Proceedinges of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.