Efficient Detection of Determinacy Races in Cilk Programs

Mingdong Feng
Department of ISCS
National University of Singapore
10 Lower Kent Ridge Road
Republic of Singapore 119260
`fengmd@iscs.nus.sg`

Charles E. Leiserson
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
USA
`cel@mit.edu`

Abstract

A parallel multithreaded program that is ostensibly deterministic may nevertheless behave nondeterministically due to bugs in the code. These bugs are called ***determinacy races***, and they result when one thread updates a location in shared memory while another thread is concurrently accessing the location. We have implemented a provably efficient determinacy-race detector for Cilk, an algorithmic multithreaded programming language. If a Cilk program run on a given input data set has a determinacy race, our debugging tool, which we call the "Nondeterminator," guarantees to detect and localize the race.

The core of the Nondeterminator is an asymptotically efficient serial algorithm (inspired by Tarjan's nearly linear-time least-common-ancestors algorithm) for detecting determinacy races in series-parallel directed acyclic graphs. For a Cilk program that runs in T time on one processor and uses v shared-memory locations, the Nondeterminator runs in $O(T\,\alpha(v,v))$ time, where α is Tarjan's functional inverse of Ackermann's function, a very slowly growing function which, for all practical purposes, is bounded above by 4. The Nondeterminator uses at most a constant factor more space than does the original program. On a variety of Cilk program benchmarks, the Nondeterminator exhibits a slowdown of less than 12 compared with the serial execution time of the original optimized code, which we contend is an acceptable slowdown for debugging purposes.

1 Introduction

Cilk [5, 20] is an algorithmic multithreaded programming language whose threads can concurrently access (read or write) shared memory without blocking. Many Cilk programs are intended to be deterministic, in that a given program produces the same behavior no matter how its threads are scheduled. If a thread updates a location while another thread is concurrently accessing the location, however, a ***determinacy race*** occurs, which may cause the program to behave nondeterministically. That is, different runs of the same program may produce different behaviors. Determinacy-race bugs are notoriously hard to detect by normal debugging techniques, such as breakpointing, because they are not easily repeatable. This paper describes a system we call the "Nondeterminator" to detect determinacy races in Cilk programs.

This research was supported in part by the Defense Advanced Research Projects Agency under Grant N00014-94-1-0985. Mingdong Feng did this work as a Postdoctoral Fellow in the MIT Laboratory for Computer Science. Parallel computing facilities were provided by the MIT Xolas Project through a generous donation by Sun Microsystems, Inc.

Determinacy races have been given many different names in the literature. For example, they are sometimes called ***access anomalies*** [7], ***data races*** [12], ***race conditions*** [19], or ***harmful shared-memory accesses*** [16]. Netzer and Miller [15] clarify different types of races and define a ***general race*** or ***determinacy race*** to be a race that causes a supposedly deterministic program to behave nondeterministically. (They also define a ***data race*** or ***atomicity race*** to be a race in a nondeterministic program involving nonatomic accesses to critical regions.) We prefer the more descriptive term "determinacy race." Emrath and Padua [9] call a deterministic program ***internally deterministic*** if the program execution on the given input exhibits no determinacy race and ***externally deterministic*** if the program has determinacy races but its output is deterministic because of the commutative and associative operations performed on the shared locations. Our Nondeterminator program checks whether a Cilk program is internally deterministic, and we have also extended the Nondeterminator to check whether a Cilk program is externally deterministic when the program contains "atomic accumulations."

To illustrate how a determinacy race can occur, consider the simple Cilk program shown in Figure 1. The Cilk procedure `main()` forks a subprocedure `foo()` using the `spawn` keyword, thereby allowing `main()` to continue executing concurrently with the spawned subprocedure `foo()`. Then, the `main()` procedure spawns another instance of `foo()`. The `sync` statement causes `main()` to join with its two subprocedures by suspending until both of these instances of `foo()` complete. (In general, the `sync` statement causes a procedure to suspend until all subprocedures that it has spawned have completed.) A ***spawn tree*** for this program, which illustrates the relationship between Cilk procedures, is shown in Figure 2.

The parallel control flow of the Cilk program from Figure 1 can be viewed as a directed acyclic graph, or ***dag***, as illustrated in Figure 3. The vertices of the dag represent parallel control constructs, and the edges represent Cilk ***threads***, which are maximal sequences of instructions not containing any parallel control constructs. In Figure 3, the threads of the program are labeled to correspond to code fragments from Figure 1, and the subdags representing the two instances of `foo()` are shaded. (Threads e_1 and e_2 contain no instructions.) In this program, both of the parallel instantiations of procedure `foo()` update the shared variable `x` in the `x = x + 1` statement. This statement actually causes the processor executing the thread to perform a read on `x`, increment the value, and then write the value back into `x`. Since these operations are not atomic, both might update `x` at the same time. Figure 4 shows how this determinacy race can cause `x` to take on different values if the threads comprising the two instantiations of `foo()` are scheduled simultaneously.

```
int x;

cilk void foo()
{
    x = x + 1;
}

cilk int main()                 /* F */
{
    x = 0;                      /* e0 */
    spawn foo();                /* F1 */
                                /* e1 */
    spawn foo();                /* F2 */
                                /* e2 */
    sync;
    printf("x is %d\n", x);     /* e3 */
    return 0;
}
```

Figure 1: A simple Cilk program that contains determinacy races. In the comments at the right, the Cilk threads that make up the procedure `main()` are labeled.

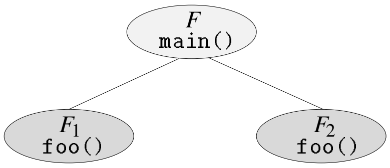

Figure 2: A spawn tree for the Cilk program in Figure 1. Each node corresponds to a procedure, and its children in the tree are the procedures it spawns.

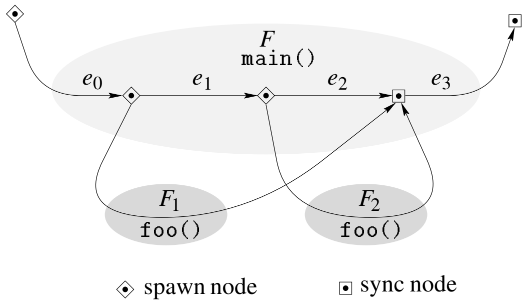

Figure 3: The parallel control-flow dag of the program in Figure 1. A spawn node of the dag represents a `spawn` construct, and a sync node represents a `sync` construct. The edges of the dag are labeled to correspond with code fragments from Figure 1. We assume that the start of the program is "spawned" by the operating system, and the end of the program is "synced" by the operating system.

Case 1			**Case 2**		
F_1	F_2	e_3	F_1	F_2	e_3
`read x`			`read x`		
`write x`				`read x`	
	`read x`		`write x`		
	`write x`			`write x`	
		"`x is 2`"			"`x is 1`"

Figure 4: An illustration of a determinacy race in the code from Figure 1. The value of the shared variable `x` read and printed by thread e_3 can differ depending on how the instructions in the two instances F_1 and F_2 of the `foo()` procedure are scheduled.

The Nondeterminator determinacy-race detector takes as input a Cilk program and an input data set and either determines which locations in the program are subject to determinacy races when the program is run on the data set, or else certifies that the program is race free when run on the data set. In general, a determinacy race occurs whenever two threads access the same location and one of the accesses is a write. If a determinacy race exists, the Nondeterminator localizes the bug, providing variable name, file name, line number, and dynamic context (state of runtime stack, heap, etc.). The Nondeterminator is not a program verifier, because the Nondeterminator cannot certify that the program is race free for all input data sets. Rather, it is a debugging tool. The Nondeterminator only checks a program on a particular input data set. What it verifies is that every possible scheduling of the program execution produces the same behavior. Moreover, even though the Nondeterminator detects determinacy races in parallel programs, it is itself a serial program.

The Nondeterminator was implemented by modifying the ordinary 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 then executes the user's program in a serial, depth-first fashion (like a C execution), but it performs race-checking actions when reads, writes, and parallel control statements occur. Figure 5 shows the performance of the Nondeterminator on several Cilk application benchmarks running on a 167-megahertz SUN Ultrasparc with the Solaris 2.5.1 operating system. Since the Nondeterminator is a serial program, our comparisons are with one-processor executions of the benchmarks.

The heart of the Nondeterminator's runtime system is an algorithm for determinacy-race detection that we call the ***SP-bags algorithm***, which was inspired by Tarjan's nearly linear-time least-common-ancestors algorithm [22]. Like Tarjan's algorithm, the SP-bags algorithm uses an efficient data structure [6, Chapter 22] to manage disjoint sets of elements. Figure 6 compares the asymptotic time and space for the SP-bags algorithm with other race-detection algorithms in the literature.

The remainder of this paper is organized as follows. Section 2 presents the SP-bags algorithm that underlies the Nondeterminator's runtime system. Section 3 reviews the basic properties of series-parallel dags and relates them to the parallel control flow of Cilk programs, and then Section 4 proves that the SP-bags algorithm correctly detects determinacy races in Cilk programs. The SP-bags algorithm only considers "pure" Cilk programs that contain `spawn` and `sync` statements, but none of Cilk's more advanced constructs that allow nondeterministic programming. (For a complete specification of the Cilk language, see [20].) Section 5 shows how to extend the SP-bags algorithm to detect determinacy races in more general Cilk programs containing atomic "accumulations," where a variable can be updated when a spawned procedure returns. Section 6 describes how we implemented the SP-bags algorithm in the Nondeterminator and provides empirical data on its performance. Section 7 discusses related work, and we offer some concluding remarks in Section 8.

2 The SP-bags algorithm

This section describes the SP-bags algorithm for determinacy-race detection. We first review the disjoint-set data structure used in the algorithm, and then we present the algorithm itself, which is inspired by Tarjan's least-common-ancestors algorithm [22]. Finally,

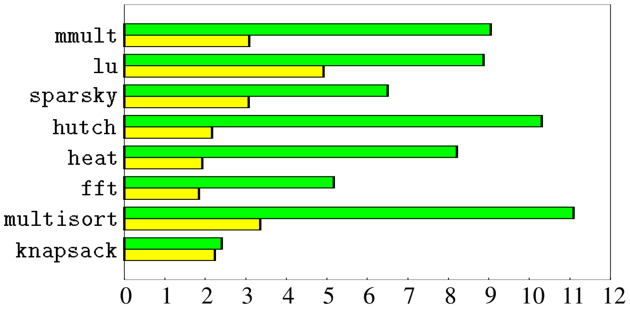

Figure 5: The slowdown of eight benchmark Cilk programs checked with the Nondeterminator. The slowdown, shown as a dark bar, is the ratio of the Nondeterminator runtime to the original optimized runtime (`gcc -O3`) of the benchmark. For comparison, the slowdown of an ordinary debugging version (`gcc -g`) of each benchmark is shown as a light bar.

Algorithm	Time: Thread creation & termination	Time: Per access	Space
English-Hebrew labeling [16]	$O(p)$	$O(pt)$	$O(vt+\min(np,vtp))$
Task recycling [7]	$O(t)$	$O(t)$	$O(vt+t^2)$
Offset-span labeling [12]	$O(p)$	$O(p)$	$O(v+\min(np,vp))$
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 6: Comparison of determinacy-race detection algorithms. The function α 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.

we prove that the running time of the algorithm is $O(T\alpha(v,v))$ when run on a Cilk program that takes time T on one processor and uses v shared-memory locations, where α is Tarjan's functional inverse of Ackermann's function [21].

The SP-bags algorithm is a serial algorithm. It uses the fact that any Cilk program can be executed on one processor in a depth-first (C-like) fashion and conforms to the semantics of the C program that results when all `spawn` and `sync` keywords are removed. As the SP-bags algorithm executes, it employs several data structures to determine which procedure instances have the potential to execute "in parallel" with each other, and is thereby able to check for determinacy races.

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[1] 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.

[1]Technically, by "procedure" we mean "procedure *instance*," that is, the runtime state of the procedure.

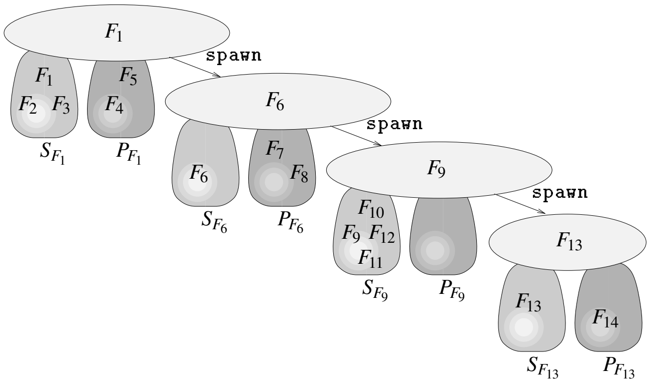

Figure 7: A snapshot of the SP-bags data structures during the execution of a Cilk program. The ovals in the figure represent procedures that are currently on the runtime stack: F_1 spawns F_6, which spawns F_9, which spawns F_{13}. Each procedure contains an S-bag and a P-bag. Each descendant of a completed 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.

The SP-bags algorithm uses the fast disjoint-set data structure [6, Chapter 22] analyzed by Tarjan [21]. The data structure maintains a dynamic collection Σ of disjoint sets and provides three elementary operations:

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 $x \in X$.

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

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 7. 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.

The S-bags and P-bags are represented as sets using a disjoint-set data structure.

The SP-bags algorithm itself is given in Figure 8. As the Cilk program executes in a serial, depth-first fashion, the SP-bags algorithm performs additional operations whenever one of the five following actions occurs: `spawn`, `sync`, `return`, `write`, and `read`. The correctness of the SP-bags algorithm is presented in Section 4, but we give an informal explanation of its operation here.

As the SP-bags algorithm executes, it updates the contents of the S-bags and P-bags whenever one of the actions `spawn`, `sync`, `return` occurs. Whenever a procedure F is spawned, S_F is initially made to contain F, because F's subsequent instructions are in series with its earlier instructions. Whenever a subprocedure F' returns to its parent F, the contents of $S_{F'}$ are emptied into P_F, since the procedures in $S_{F'}$ can execute in parallel with any subprocedures that F might spawn in the future before performing a `sync`. When a `sync` occurs, P_F is emptied into its S_F, since all of F's previously

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

`sync` in a procedure F:
  $S_F \leftarrow$ UNION(S_F, P_F)
  $P_F \leftarrow \emptyset$

`return` from procedure F' to F:
  $P_F \leftarrow$ UNION$(P_F, S_{F'})$

`write` a shared location l by procedure F:
  if FIND-SET$(\textit{reader}(l))$ is a P-bag
    or FIND-SET$(\textit{writer}(l))$ is a P-bag
   then a determinacy race exists
  $\textit{writer}(l) \leftarrow F$

`read` a shared location l by procedure F:
  if FIND-SET$(\textit{writer}(l))$ is a P-bag
   then a determinacy race exists
  if FIND-SET$(\textit{reader}(l))$ is an S-bag
   then $\textit{reader}(l) \leftarrow F$

Figure 8: 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.

spawned subprocedures and their descendants logically precede any future subprocedures spawned by F.

Determinacy races are detected by the code for `write` and `read`. A race occurs if a procedure F writes 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 accessor of l operate logically in parallel. Similarly, a race occurs whenever F reads a location l and discovers that the previous writer is in a P-bag. In the normal case, whenever a location l is written, location l in the *writer* shadow space is updated to be F. The reader of l is updated to be F when a read occurs, but only if the previous reader operates logically in series with F. The logic behind this subtle piece of code is explained in Section 4, where the SP-bags algorithm is proved correct.

To conclude this section, we analyze the asymptotic performance of the SP-bags algorithm.

Theorem 1 Consider a Cilk program that executes in time T on one processor and references v shared memory locations. The SP-bags algorithm can be implemented to check this program for determinacy races in $O(T\alpha(v,v))$ time using $O(v)$ space.

Proof: Let n be the number of spawned procedures during the execution of the SP-bags algorithm, which is also the the total number of procedure ID's used by the algorithm. The total number of all MAKE-SET, UNION, and FIND-SET operations is at most the serial running time T. Consequently, by using the fast disjoint-set data structure analyzed by Tarjan, we obtain a running time of $O(T\alpha(T,n))$. Since the two shadow spaces take $O(v)$ space and the disjoint-set data structure takes $O(n)$ space, the total space used by the algorithm is $O(v+n)$.

By using garbage collection, the time and space can be reduced to $O(T\alpha(v,v))$ and $O(v)$, respectively. The idea is to run the basic SP-bags algorithm for v steps, and then scan through the shadow spaces marking which procedure ID's are in use. Then, we remove the unused ID's from the disjoint-set data structure, which can be done in

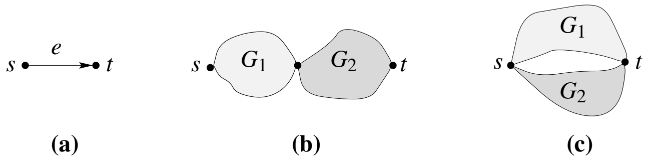

Figure 9: The three ways that a series-parallel dag can be constructed. **(a)** A base graph. **(b)** Series composition of two series-parallel dags. **(c)** Parallel composition of two series-parallel dags.

$O(v\alpha(v,v))$ time. We repeat the garbage collection every v steps. Thus, in T time, we perform $\lceil T/v \rceil$ garbage collections, resulting in a running time of $O(T\alpha(v,v))$. Because the amount of space in use after each garbage collection is $O(v)$ and at most $O(v)$ additional space can accumulate during the v steps between garbage collections, the algorithm uses a total of $O(v)$ space. ■

In practice, it is probably not worthwhile to implement the garbage collection, and the Nondeterminator does not implement it. Also, the worst-case bounds can easily be improved if they are expressed using more detailed parameters than T and v.

3 Series-parallel dags

Series-parallel dags [23] are a straightforward extension of the notion of series-parallel graphs [8, 11, 17]. In this section, we review basic properties of series-parallel dags and show how a Cilk program execution corresponds to a series-parallel dag. These properties will be used in Section 4 to prove the correctness of the SP-bags algorithm.

We first define various relationships among Cilk threads. A thread e_1 ***precedes*** a thread e_2, denoted $e_1 \prec e_2$, if there is a path in the Cilk dag that includes both e_1 and e_2 in that order. Two distinct threads e_1 and e_2 operate logically ***in parallel***, denoted $e_1 \parallel e_2$, if $e_1 \not\prec e_2$ and $e_2 \not\prec e_1$. Informally, $e_1 \prec e_2$ means that e_1 must execute before e_2 in any legal scheduling of a Cilk program, while $e_1 \parallel e_2$ means that e_1 and e_2 can execute at the same time. The precedence relation $\prec$ is transitive.

A ***series-parallel dag*** $G = (V, E)$ is a directed acyclic graph with two distinguished vertices, a ***source*** $s \in V$ and a ***sink*** $t \in V$, which is constructed recursively in one of the following ways, as illustrated in Figure 9:

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 G_1 and G_2 with disjoint edge sets in which the source of G_1 is s, the sink of G_2 is t, and the sink of G_1 is the source of G_2.

Parallel composition: The graph consists recursively of two series-parallel dags G_1 and G_2 with disjoint edge sets in which the sources of G_1 and G_2 are both s and the sinks of G_1 and G_2 are both t.

Note that for series composition, it makes a difference which subgraph precedes the other, but the order of parallel composition does not matter. Moreover, one can prove by induction that any series-parallel dag is indeed a dag.

The following properties of series-parallel dags are presented without proof.

Lemma 2 Let G' be a series-parallel dag, let G be a series-parallel subdag of G', and let s and t be the source and sink of G, respectively. Then, the following properties hold:

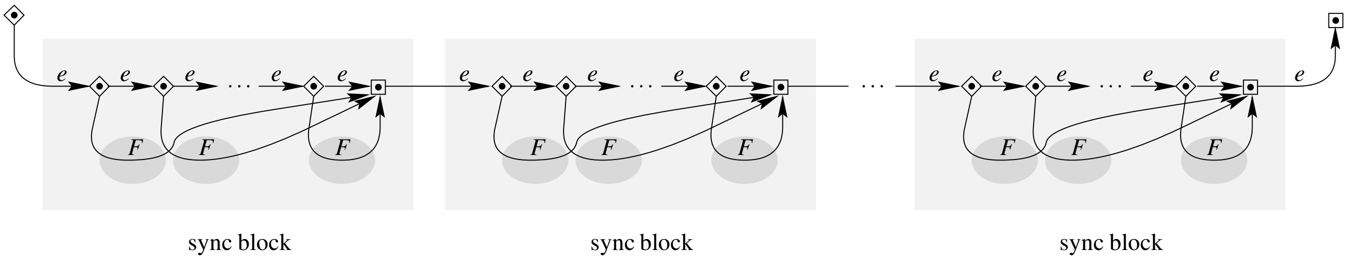

Figure 10: The dag of a spawned Cilk procedure that contains `spawn` and `sync` statements. It consists of a linear sequence of sync blocks (rectangles in the figure) terminated by a `return` statement. Each e corresponds to a thread of the Cilk procedure, and each F corresponds to a spawned subprocedure.

1. There exists a path in G from s to any edge in G.
2. There exists a path in G from any edge in G to t.
3. Every path in G' that begins outside of G and enters G passes through s.
4. Every path in G' that begins within G and leaves G passes through t. ■

The following theorem shows that any Cilk parallel control-flow dag, such as that in Figure 3, is series-parallel.

Theorem 3 A Cilk parallel control-flow dag is a series-parallel dag.

Proof: We use induction on the depth of the Cilk spawn tree. A Cilk procedure that contains no `spawn` or `sync` statements is a leaf of the spawn tree and is trivially a base series-parallel dag.

Consider a Cilk procedure that contains `spawn` and `sync` statements. The parallel control flow of the procedure at runtime can be viewed as a linear sequence of ***sync blocks*** terminated by a `return` statement, where each sync block consists of a sequence of `spawn` statements interleaved with C code and terminated by a `sync`. In other words, the execution of a procedure has the form

e; `spawn` F; e; `spawn` F; e; …; `spawn` F; e; `sync`;
e; `spawn` F; e; `spawn` F; e; …; `spawn` F; e; `sync`;
…
e; `spawn` F; e; `spawn` F; e; …; `spawn` F; e; `sync`;
e; `return`;

where each e is a thread of the Cilk procedure, each F is a spawned subprocedure, and each line except the last is a sync block. The dag corresponding to the parallel control flow is shown in Figure 10. By induction, the computation arising from each spawned subprocedure is a series-parallel dag. Each sync block is a series-parallel dag created by alternating series and parallel compositions of threads, the spawned procedures, and the spawn and sync nodes. The Cilk dag representing the procedure and all its descendants can now be assembled by serially composing all the sync blocks. ■

A series-parallel dag can be represented by a binary ***parse tree***, as illustrated in Figure 11 for the Cilk procedure from Figure 10. The leaf nodes of the parse tree correspond to edges of the dag (Cilk threads), and each internal node is either an S-node **S**, which corresponds to a series composition of its two children, or a P-node **P**, which corresponds to a parallel composition of its children.

A canonical parse tree for a Cilk dag can be constructed as follows. We first build a parse tree recursively for each child of the root procedure. For each sync block of the root procedure, we apply alternating parallel and series composition on the child parse tree to create a parse tree for the sync block. Finally, we string the parse trees for the sync blocks together into a ***spine*** for the procedure by applying a sequence of series compositions to the sync blocks. Sync blocks are composed serially, because a `sync` statement is never passed until *all* previously spawned subprocedures have completed. The only ambiguities that might arise in the parse tree occur because of the associativity of series composition and the commutativity of parallel composition. If, as shown in Figure 11, the alternating S-nodes and P-nodes in a sync block always place threads and subprocedures on the left, and the series compositions of the sync blocks are applied in order from last to first, then the parse tree is unique. Such a ***canonical*** parse tree is shown in Figure 12 for the Cilk dag in Figure 3.

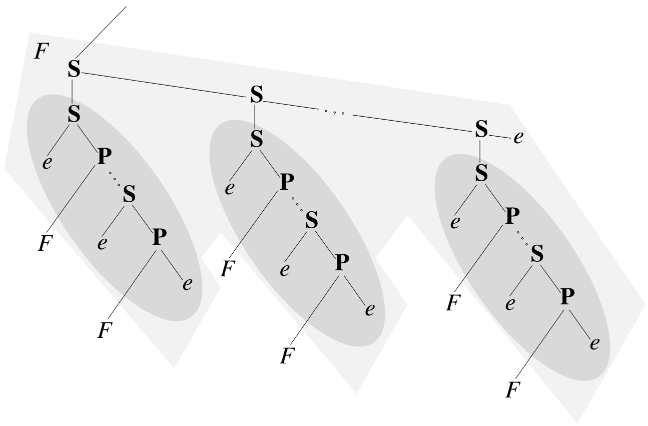

Figure 11: The canonical parse tree for a generic Cilk procedure. The notation F represents the parse tree of any subprocedure spawned by this procedure, and e represents any thread of the procedure. All nodes in the shaded areas belong to the procedure, and the nodes in each oval belong to the same sync block. A sequence of S-nodes forms the spine of the parse tree, composing all sync blocks in series. Each sync block contains an alternating sequence of S-nodes and P-nodes. Observe that the left child of an S-node in a sync block is always a thread, and that the left child of a P-node is always a subprocedure.

The canonical parse tree satisfies an interesting property with respect to a serial, depth-first execution of the Cilk program. Specifically, an ordinary depth-first tree walk (see [6, p. 245]) of the parse tree visits the threads of the computation in the same order as the threads are encountered when the Cilk program is executed in a depth-first (C-like) fashion on a single processor.

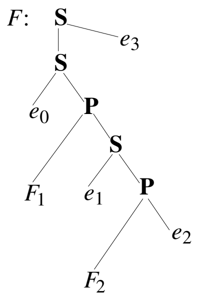

Figure 12: The canonical parse tree for the Cilk dag in Figure 3.

4 Correctness of the SP-bags algorithm

In this section, we prove the correctness of the SP-bags algorithm. We begin by showing how either a precedence relation $\prec$ or a parallel relation $\parallel$ between two threads in a Cilk dag can be inferred from the threads' least common ancestor in the parse tree of the dag. We then prove a lemma that characterizes the contents of S-bags and P-bags during the execution of the SP-bags algorithm. We conclude by showing that the SP-bags algorithm correctly detects determinacy races.

The SP-bags algorithm hinges on the notion of the least common ancestor of two nodes in a tree. Given two nodes x and y in a rooted tree, their ***least common ancestor***, denoted $\mathrm{LCA}(x,y)$, is the deepest node in the tree that is a common ancestor of both x and y. Alternatively, if one traces the unique simple path from x to y, their least common ancestor is the node on the path that is closest to the root. The next lemma and its corollary show how the least common ancestor of two threads in the parse tree can be used to determine whether the threads operate logically in parallel or whether one precedes the other.

Lemma 4 Let e_1 and e_2 be distinct threads in a Cilk dag, and let $\mathrm{LCA}(e_1,e_2)$ be their least common ancestor in a parse tree for the dag. Then, $e_1 \parallel e_2$ if and only if $\mathrm{LCA}(e_1,e_2)$ is a P-node.

Proof: ($\Rightarrow$) Assume for the purpose of contradiction that $e_1 \parallel e_2$ and $\mathrm{LCA}(e_1,e_2)$ is an S-node. Let G_1 be the graph corresponding to the left subtree of $\mathrm{LCA}(e_1,e_2)$, and let G_2 be the graph corresponding to the right subtree. By Lemma 2, there exists a path from e_1 to the sink of G_1 and a path from the source of G_2 to e_2. Since G_1 and G_2 are composed in series, the sink of G_1 and the source of G_2 are the same node, and hence $e_1 \prec e_2$, contradicting the assumption that $e_1 \parallel e_2$.

($\Leftarrow$) Assume for the purpose of contradiction that $e_1 \prec e_2$ and $\mathrm{LCA}(e_1,e_2)$ is a P-node. Let G_1 be the graph corresponding to the left subtree of $\mathrm{LCA}(e_1,e_2)$, and let G_2 be the graph corresponding to the right subtree. By Lemma 2, the path from e_1 to e_2 must go through the sink of G_1 and the source of G_2. Since G_1 and G_2 are composed in parallel, the sink of G_1 is the sink of G_2 and the source of G_1 is the source of G_2. Thus, we have a path from the sink of G_1 to the source of G_1, contradicting the fact that G_1 is a dag. ■

Corollary 5 Let e_1 and e_2 be distinct threads in a Cilk dag, and let $\mathrm{LCA}(e_1,e_2)$ be their least common ancestor in a parse tree for the dag. Then, $e_1 \prec e_2$ if and only if $\mathrm{LCA}(e_1,e_2)$ is an S-node, e_1 is in the left subtree of $\mathrm{LCA}(e_1,e_2)$, and e_2 is in the right subtree of $\mathrm{LCA}(e_1,e_2)$. ■

As an example of the use of Lemma 4, in Figure 12 we have $F_1 \parallel F_2$, because $\mathrm{LCA}(F_1,F_2)$ is a P-node. In contrast, since e_1 occurs to the left of F_2 in the parse tree and $\mathrm{LCA}(e_1,F_2)$ is an S-node, by Corollary 5 we can conclude that $e_1 \prec F_2$.

The SP-bags algorithm takes advantage of relationships among threads that can be derived from the serial, depth-first execution order of the dag. The following two lemmas exploit the depth-first execution order of the algorithm to determine when threads operate logically in parallel.

Lemma 6 Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk dag, and suppose that $e_1 \prec e_2$ and $e_1 \parallel e_3$. Then, we have $e_2 \parallel e_3$.

Proof: Assume for the purpose of contradiction that $e_2 \prec e_3$. Then, since $e_1 \prec e_2$, we have $e_1 \prec e_3$ by transitivity, contradicting the assumption that $e_1 \parallel e_3$. ■

Lemma 7 (Pseudotransitivity of $\parallel$) Suppose that three threads e_1, e_2, and e_3 execute in order in a serial, depth-first execution of a Cilk dag, and suppose that $e_1 \parallel e_2$ and $e_2 \parallel e_3$. Then, we have $e_1 \parallel e_3$.

Proof: Consider the parse tree of the Cilk dag with e_1, e_2, and e_3. Let $a_1 = \mathrm{LCA}(e_1,e_2)$ and $a_2 = \mathrm{LCA}(e_2,e_3)$. Lemma 4 implies that both a_1 and a_2 are P-nodes. Because e_1, e_2, and e_3 execute in order, one can show that either a_1 or a_2 is the least common ancestor of e_1 and e_3, and since both a_1 and a_2 are P-nodes, it follows from Lemma 4 that $e_1 \parallel e_3$. ■

From the construction of the canonical parse tree for the Cilk dag, it is apparent that each procedure in the spawn tree is represented by an assembly of threads and internal nodes in the parse tree. We define the mapping h of threads or nodes in the canonical parse tree to procedures in the spawn tree to be the ***procedurification*** function for the parse tree. This procedurification function is used in the next lemma to relate the S-nodes and P-nodes in the parse tree to procedure ID's in the S-bags and P-bags during the execution of the SP-bags algorithm.

Lemma 8 Consider an execution of the SP-bags algorithm on a given Cilk dag. Let h be the procedurification function mapping the canonical parse tree for the dag to procedures in the spawn tree. Suppose thread e_1 is executed before thread e_2, and let $a = \mathrm{LCA}(e_1,e_2)$ be their least common ancestor in the parse tree. If a is an S-node, then the procedure ID for $h(e_1)$ belongs to the S-bag of $h(a)$ when e_2 is executed. Similarly, if a is a P-node, then the procedure ID for $h(e_1)$ belongs to the P-bag of $h(a)$ when e_2 is executed.

Proof: We shall first show that if a is an S-node of the parse tree, then the procedure ID for $h(e_1)$ belongs to the S-bag of the procedure $h(a)$. There are two possibilities (as can be seen from Figure 11) depending on whether a belongs to the spine or a sync block of the parse tree.

If a belongs to the spine, then e_1 belongs to a's left subtree, which is rooted in a sync block of the parse tree. At the time thread e_2 is executed, in what bag does the procedure ID for $h(e_1)$ reside? From the code for the SP-bags algorithm in Figure 8, we can see that when e_1 is executed, the ID for $h(e_1)$ is placed in $h(e_1)$'s own S-bag by the `spawn` action. From the construction of the canonical parse tree (see Figure 11), we observe that either $h(e_1) = h(a)$ or $h(e_1)$ is a descendant of $h(a)$. From the time that e_1 is executed to the time e_2 is executed, the only operations that move the ID for $h(e_1)$ are `sync` and `return`, which never move the ID down the spawn tree, and indeed, $h(e_1)$'s ID moves up exactly when one of its ancestors returns. Consequently, when the `sync` corresponding to a is executed, $h(e_1)$'s ID is placed into the S-bag of $h(a)$, if it is not already there. From that point until e_2 is executed, no operations remove $h(e_1)$'s ID from $h(a)$'s S-bag.

If a belongs to one of $h(a)$'s sync blocks, then the construction of the canonical parse tree implies that e_1 is the left child of a, as can be seen in Figure 11. Consequently, we have $h(e_1) = h(a)$, and the SP-bags algorithm places the procedure ID for $h(e_1)$ into $h(a)$'s S-bag at the moment that $h(a)$ is spawned. From that moment until the time e_2 is executed, the S-bag of $h(a)$ is never emptied, since $h(a)$ does not return until after executing e_2. Thus, $h(e_1)$'s ID belongs to $h(a)$'s S-bag when e_2 is executed.

We now show that if a is a P-node of the parse tree, then the procedure ID for $h(e_1)$ belongs to the P-bag of the procedure $h(a)$. If a is a P-node, then the thread e_1 belongs to the left subtree of a and the thread e_2 belongs to a's right subtree. As in the argument for when a is an S-node in $h(a)$'s spine, when e_2 is executed, the procedure $h(e_1)$ must belong to a bag in the procedure $h(a)$ of their least common ancestor a. In this case, however, the procedure ID for $h(e_1)$ belongs to $h(a)$'s P-bag, since $h(e_1)$ is a proper descendant of $h(a)$, the ID for $h(e_1)$ is placed in $h(a)$'s P-bag when $h(a)$'s left child returns, and the P-bag of $h(a)$ is not emptied until a's entire sync block is executed. ∎

Corollary 9 Consider an execution of the SP-bags algorithm on a given Cilk dag, and let h be the procedurification function mapping the canonical parse tree for the dag to procedures in the spawn tree. Suppose thread e_1 is executed before thread e_2. Then, $e_1 \prec e_2$ if and only if the procedure ID for $h(e_1)$ belongs to an S-bag when e_2 is executed. Similarly, $e_1 \parallel e_2$ if and only if the procedure ID for $h(e_1)$ belongs to a P-bag when e_2 is executed.

Proof: Combine Lemma 4, Corollary 5, and Lemma 8. ∎

We now prove that the SP-bags algorithm is correct.

Theorem 10 The SP-bags algorithm detects a determinacy race in a Cilk program if and only if a determinacy race exists.

Proof: (⇒) Suppose that the SP-bags algorithm detects a determinacy race when executing a thread e_2. According to the SP-bags algorithm (see Figure 8), one of three cases occurs:

1. e_2 performs a `write` and *reader*(l) belongs to a P-bag;
2. e_2 performs a `write` and *writer*(l) belongs to a P-bag;
3. e_2 performs a `read` and *writer*(l) belongs to a P-bag.

In the first case, the procedure ID stored in *reader*(l) is set by a thread e_1 which executes before e_2 and reads l, and hence by Corollary 9, we have $e_1 \parallel e_2$. Since e_1 reads l, e_2 writes l, and the two threads operate logically in parallel, a determinacy race exists. The other two cases are similar.

(⇐) We now show that if a program contains a determinacy race on a location l, then the SP-bags algorithm reports a determinacy race on location l. Let e_1 and e_2 be two threads involved in a determinacy race on location l, where if there are several determinacy races on l, we choose the determinacy race whose second thread executes earliest in the depth-first execution order of the program. By definition of a determinacy race, we have $e_1 \parallel e_2$, and without loss of generality, e_1 executes before e_2.

There are three possible ways the determinacy race could occur:

1. e_1 writes l and e_2 reads l;
2. e_1 writes l and e_2 writes l;
3. e_1 reads l and e_2 writes l.

In each case, let h be the procedurification function mapping threads or nodes of the canonical parse tree to procedures in the spawn tree.

Case 1. Suppose that 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 $e_1 \parallel e_2$, Corollary 9 implies that *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 $e \parallel e_2$ by Lemma 6. Consequently, Corollary 9 implies that $h(e) =$ *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.

Case 2. This case is similar to Case 1.

Case 3. In this case, when e_2 is executed, suppose that *reader*$(l) = h(e)$ for some thread e. If $e = e_1$, then since $e_1 \parallel e_2$, Corollary 9 implies that *reader*(l) belongs to a P-bag and the determinacy race is reported. Consequently, we may assume that $e \neq e_1$. We consider two situations depending on whether e_1 updates *reader*(l) when it executes.

If e_1 updates *reader*(l), then consider the sequence of updates to *reader*(l) from the time e_1 executes up to and including the time e executes. Let the threads performing the updates be $e'_1, e'_2, \ldots, e'_k$, where $e'_1 = e_1$ and $e'_k = e$. From the code for `read` in Figure 8, we must have for $i = 1, 2, \ldots, k-1$ that $e'_i \prec e'_{i+1}$, since by Corollary 9, the ID of $h(e'_i)$ belongs to an S-bag when e'_{i+1} executes. By transitivity, therefore, we have $e_1 \prec e$. Since $e_1 \parallel e_2$, by Lemma 6, it follows that $e \parallel e_2$. Consequently, by Corollary 9, the determinacy race between e and e_2 is detected.

If e_1 does not update *reader*(l), then when e_1 executes, we must have $h(e') =$ *reader*(l) for some thread $e' \parallel e_1$ that executes before e_1. Since $e_1 \parallel e_2$, by pseudotransitivity (Lemma 7) it follows that $e' \parallel e_2$. Looking at the sequence of updates of *reader*(l) from the execution of e' up to and including the execution of e, we can conclude that $e' \prec e$. Since $e' \parallel e_2$, Lemma 6 implies that $e \parallel e_2$, and hence the determinacy race between e and e_2 is reported. ∎

5 Support for atomic accumulation

Cilk supports the atomic *accumulation* of results returned by spawned procedures. If the operators used to augment the accumulated variable are commutative—they are all `+=` or `-=`, for example—we would like the concurrent accessing of the updates not to be viewed as races, because the order of accumulation does not affect the "external determinacy" [9] of the computation. That is, the behavior of the program is deterministic, even though different executions may cause some variables to pass through different intermediate states. In this section, we show how to extend the SP-bags algorithm to detect determinacy races in Cilk code where races between accumulations are considered to be "legal."

Consider the Cilk procedure `foo()` from Figure 13. Cilk guarantees that this code produces the same result for the integer variable `x` no matter how threads are scheduled. The basic idea is that accumulations of this kind are performed atomically with respect to one another, and the updates to `x` are ***commutative***: no matter what order they are executed, `x` has the same value after the `sync`. Thus, even though different executions may cause `x` to pass through different intermediate states, the final result is the same. A determinacy race in an externally deterministic program is called a ***legal*** determinacy race, and it is ***illegal*** otherwise.

Cilk guarantees the atomicity of accumulations only for accumulations within the same procedure instance. Accumulations by other procedure instances that operate logically in parallel are not guaranteed to be atomic by Cilk's runtime system, and they can cause nondeterminism. Atomicity alone is not sufficient for a race to be legal, however. It must also involve commutative updates. For example, if the accumulation operator "`-=`" in `foo()` is replaced by the operator "`*=`", the race is illegal, because the order of execution can

```
cilk int foo()
{   ...
    x += spawn bar();
    x -= spawn baz();
    x += 1;
    sync;
    ...
}
```

Figure 13: An illustration of the use of accumulation in a Cilk program. The integer variable `x` may or may not be local to the procedure `foo()`. Although determinacy races occur between updates to `x`, the races are legal, since the updates occur atomically.

affect the final value of `x`, even though the updates are performed atomically.

The SP-bags algorithm can be modified to accommodate legal races. There are two key changes to the data structures. First, we create a shadow space to record the operator whenever an accumulation or assignment occurs. (The assignment operator `=` is considered to be a degenerate accumulation operator which does not commute with any other operators, including itself.) Second, in addition to procedure ID's, the SP-bags algorithm assigns each sync block a distinct ID. The sync-block ID and operator are stored in a shadow space whenever an accumulation occurs.

Figure 14 extends the SP-bags algorithm of Figure 8 to detect the determinacy races in Cilk code containing accumulations. In addition to the introduction of a new action `accumulate` that deals with the case when the returned result of a spawned procedure is accumulated, only the `write` action needs to be extended. A new shadow space called *operator* stores the operator for each location in the shared memory. When a procedure F in a sync block B writes a location l with accumulation operator *op*, and it discovers that the previous writer of l belongs to a P-bag, a determinacy race occurs only if the previous writer is not B or if the previous writer's operator does not commutate with *op*. If no determinacy race occurs, the operator of l is updated to *op*. When a spawned procedure returns its result to procedure F and accumulates the result into a location l, the operations are almost the same as the `write` action except that it is necessary to check whether the current sync block B belongs to any bag. If not, the unique ID for B is placed into the P-bag of F, if it is not there already.

Theorem 11 The extended SP-bags algorithm detects a determinacy race in a Cilk program containing accumulations if and only if an illegal determinacy race exists.

Proof sketch: The proof is similar to that of Theorem 10. Once again, the "only if" direction is straightforward, and the hard part is the "if" direction. The extended SP-bags algorithm contains an additional check when a thread performs a `write` and $writer(l)$ belongs to a P-bag. If $writer(l)$ is the ID of the current sync block, then l has been accumulated by the returned result of a previously spawned procedure in the same sync block. If the operator is also commutative with $operator(l)$, then the determinacy race is legal, because the accumulations are performed atomically. Otherwise, the determinacy race is illegal and is reported. Determinacy races caused by the `accumulate` action are checked similarly to the ones by the `write` action. ■

`write` a shared location l with operator *op* by procedure F in sync block B:

- **if** $\text{FIND-SET}(reader(l))$ is a P-bag
  - **then** a determinacy race exists
- **if** $\text{FIND-SET}(writer(l))$ is a P-bag
  - **and** $(writer(l) \neq B$
    - **or** *op* does not commutate with $operator(l))$
  - **then** a determinacy race exists
- $writer(l) \leftarrow F$
- $operator(l) \leftarrow op$

`accumulate` returned result of spawned procedure into a shared location l with operator *op* by procedure F in sync block B:

- **if** $\text{FIND-SET}(reader(l))$ is a P-bag
  - **then** a determinacy race exists
- **if** $\text{FIND-SET}(writer(l))$ is a P-bag
  - **and** $(writer(l) \neq B$
    - **or** *op* does not commutate with $operator(l))$
  - **then** a determinacy race exists
- **if** $\text{FIND-SET}(B) = \emptyset$
  - **then** $P_F \leftarrow \text{UNION}(P_F, \text{MAKE-SET}(B))$
- $writer(l) \leftarrow B$
- $operator(l) \leftarrow op$

Figure 14: The extended SP-bags algorithm of Figure 8 for Cilk code containing accumulations. Operations for the `write` action are extended. Operations for the `accumulate` action are performed when atomic accumulation occurs.

6 The Nondeterminator

This section presents the implementation of the Nondeterminator, our determinacy-race detector for Cilk programs. We discuss how the Nondeterminator implements the SP-bags algorithm by modifying the Cilk compiler and runtime system. We describe some modifications to the SP-bags algorithm that enhance the Nondeterminator's performance. Empirical data from a variety of benchmark Cilk programs shows that the Nondeterminator typically runs in less than 12 times the execution time of the original optimized program.

The first phase of checking a user's Cilk program is to run the code through the Cilk compiler with an option that turns on determinacy-race detection. This compiler option produces object code with calls to the Nondeterminator's runtime system for every read and write of shared memory. In addition, the compiler inserts hooks that allow the Nondeterminator's runtime system to perform actions for every `spawn`, `sync`, and `return`.

At runtime, before it starts executing the user code, the Nondeterminator sets up the *reader* and *writer* shadow spaces. We use the Unix memory-mapping primitive `mmap()` to fix the starting address of each shadow space so that the shadow-space address can be obtained quickly from the corresponding shared-memory address. It also initializes the disjoint-set data structure.

During execution of the user program, the Nondeterminator performs the SP-bags algorithm (without garbage collection), modified slightly to improve performance. First, if the compiler can determine that a memory reference is to a nonshared memory region, such as a local variable whose address is never computed, no determinacy-race check is necessary, because no determinacy race is possible. Second, we modify the SP-bags algorithm to update $reader(l)$, as well as $writer(l)$, whenever a write or accumulate to a location l occurs. This change allows us to check only $reader(l)$ in the code for `write` and `accumulate` (see Figure 8 and Figure 14); and in the code for `read`, we need only check $writer(l)$ when $reader(l)$ belongs to a P-bag. Third, during the execution of a thread, we save addresses that have previously been checked in a software cache to avoid checking them again within the same thread.

We have measured the performance of the Nondeterminator on

Program	Original (seconds)	Nondeterminator (seconds)	Slowdown	Number of actions	Average overhead (nanoseconds)	Cache-hit ratio
`mmult`	3.54	32.06	9.05	317,947,466	89.69	77.26%
`lu`	2.36	20.93	8.87	184,738,250	100.52	89.14%
`sparsky`	14.91	97.02	6.51	289,381,593	283.74	44.17%
`hutch`	4.71	48.58	10.31	200,693,060	218.57	77.29%
`heat`	2.60	21.37	8.21	125,143,001	149.94	78.23%
`fft`	4.17	23.03	5.18	39,411,729	471.43	7.39%
`multisort`	5.22	57.94	11.09	179,988,858	292.86	42.15%
`knapsack`	7.39	17.73	2.41	34,752,741	298.30	33.34%

Figure 15: Eight benchmark Cilk programs that were checked with the Nondeterminator. The slowdown is the ratio of the Nondeterminator runtime and the original optimized runtime of the benchmark. The total number of actions (spawns, syncs, returns, shared reads, and shared writes) is given, along with the average overhead of the Nondeterminator for each action and the fraction of accesses that hit the Nondeterminator's software cache.

eight benchmark Cilk programs:

- `mmult` — Block multiplication of two dense 512×512 matrices, written by Keith Randall.
- `lu` — LU-decomposition of a dense 512×512 matrix, written by Robert D. Blumofe.
- `sparsky` — Cholesky factorization of a sparse 3600×3600 matrix with 15,100 nonzeros, written by Aske Plaat and Keith Randall.
- `hutch` — Barnes-Hut n-body calculation with 4096 cells, written by Keith Randall.
- `heat` — Heat diffusion on a 4096×16 mesh, written by Volker Strumpen.
- `fft` — Fast Fourier transformation of a vector of length 2^{20}, written by Matteo Frigo.
- `multisort` — Sort a random permutation of 4 million 32-bit integers, written by Matteo Frigo and Andrew Stark.
- `knapsack` — Solve the 0-1 knapsack problem on 30 items using branch and bound, written by Matteo Frigo.

The results of our tests, which were run on a 167-megahertz SUN Ultrasparc with the Solaris 2.5.1 operating system, are shown in Figure 15. As we can see from Figure 8 and Figure 14, the SP-bags algorithm is invoked when a `spawn`, `sync`, `return`, shared read, or shared write occurs. Each of these invocations, which we call an ***action***, contributes to the overhead incurred by the Nondeterminator. The number of actions in each benchmark program is given in Figure 15.

We observe that the average overhead per action varies among these benchmark programs, ranging from 90 nanoseconds to 472 nanoseconds. The variation is due to the Nondeterminator's software cache. Whenever the cache-hit ratio is large (*i.e.*, a thread exhibits substantial temporal locality in its shared-memory access patterns), relatively few shared read or write accesses need to incur the full overhead of the SP-bags algorithm. Thus, the average overhead per action is small. For example, the `fft` and `knapsack` programs exhibit small cache-hit ratios, and thus the overhead per action is comparatively high. For other benchmarks, the software cache is reasonably effective, and the overhead per action is within 300 nanoseconds.

The Nondeterminator has caught determinacy races in several Cilk programs. For example, it caught a subtle bug in a program to solve the N-queens puzzle which was included as a programming example in the Cilk software distribution. The goal of the N-queens puzzle is to find a configuration of n queens on an $n \times n$ chessboard such that no queen attacks another. The standard backtrack algorithm to solve this puzzle is to place queens row by row, and backtrack whenever a developed configuration contains two queens that attack each other.

```
cilk char *nqueens(char *board, int n, int row)
{ char *new_board;
  ...
  new_board = malloc(row+1);
  memcpy(new_board, board, row);/* read *board */
  for (j = 0; j < n; j++) {
    ...
    new_board[row] = j;    /* write *new_board */
    spawn nqueens(new_board, n, row+1);
    ...
  }
  sync;
  ...
}
```

Figure 16: A fragment of a Cilk program solving the N-queens puzzle. A determinacy race exists involving the commented lines in the code.

The recursive Cilk procedure `nqueens` in Figure 16 illustrates the bug in the original implementation of this backtrack algorithm. It is called with three arguments: `board`, which is the current configuration of queens on the chessboard; `n`, which is the size of the chessboard; and `row`, which is the row number where a queen will be placed. Before a queen is placed, space for a new configuration `new_board` is allocated using `malloc` so that the child that will be recursively spawned to solve the new configuration does not overwrite the storage in the parent. The current configuration `board` is copied into `new_board` using `memcpy`. The `spawn` in the `for` loop causes the searches to be spawned in parallel to solve configurations in which the just-placed queen is in different columns of the current row.

When the `nqueens` code ws run through the Nondeterminator, it reported that `board` and `new_board` are involved in races. Specifically, a race exists between the read of `board` in a spawned subprocedure and the write of `new_board` in its parent procedure. Since the passed `board` argument of the subprocedure points to the same storage as the `new_board` of its parent procedure, when the subprocedure is reading the `board` in `memcpy`, the parent procedure may be updating the `new_board` at the same time, resulting in a determinacy race.

Besides the N-queens puzzle, several Cilk users have used the Nondeterminator to discover determinacy-race bugs in their programs, which have included a radiosity calculation for graphics rendering, enumeration of magic squares, and an old version of our heat-diffusion benchmark. Some Cilk users have not taken advantage of this tool, however, much to their detriment. In a student assignment at MIT to implement Strassen's matrix multiplication algorithm in Cilk, half of the submitted codes turned out to have determinacy races that were not detected during the students' repeated test runs. These bugs were instantly caught when the instructors ran the programs through the Nondeterminator. The students could have easily run the Nondeterminator themselves (the theory of which was taught in their class), but their overconfidence was natural, since their code worked on every test run. Indeed, determinacy races are latent bugs that can escape extensive testing, rearing their ugly heads only intermittently and confounding naive debugging attempts. With the release of the Nondeterminator as part of the overall Cilk system, we hope more Cilk programmers will routinely use the Nondeterminator as a debugging tool to produce more reliable parallel code.

7 Related work

This section briefly reviews related work on the problem of detecting determinacy races in parallel programs. A comparison of the asymptotic time and space requirements of the Nondeterminator with work in the literature was presented in Figure 6.

Bernstein [3] identifies determinacy races as a cause of nondeterministic behavior. Netzer and Miller [15] present a formal model for understanding race conditions in parallel programs, distinguishing determinacy races from atomicity races. They reference several algorithms for atomicity-race detection, but we do not discuss this type of race detection here. Static analysis of parallel programs to uncover nondeterminacy has been studied extensively, for example, in [9, 13]. Various systems have been developed for determinacy-race detection that do not allow nested parallelism, as for example [2].

We now review related work on determinacy-race detection for programs with nested parallelism.

Nudler and Rudolph [16] give an "English-Hebrew labeling" algorithm that detects determinacy races in programs with series-parallel dependences, but their model also allows messages between threads, which produces a richer and more difficult class of programs to check. Their algorithm assigns to each thread a pair of labels: an "English" label, which is produced by performing a left-to-right preorder numbering on the task tree, and a "Hebrew" label, which is produced symmetrically for a right-to-left ordering. To determine whether two threads operate logically in parallel, a comparison of the labels of two threads suffices.

Dinning and Schonberg [7] improve the performance of the English-Hebrew labeling algorithm by "task recycling," but at the cost of failing to detect some determinacy races. Each thread (task) has a unique task identifier, and a version number. In order to save space, a task identifier can be reassigned to another thread during the program execution. Each thread also maintains a parent vector containing the largest version number of its ancestor threads. With the parent vector, checking whether two blocks are logically parallel is reduced to one access of the parent vector and one comparison, which are constant-cost operations. Dinning and Schonberg give performance data indicating a slowdown of between 3 and 11 to check between 50 and 80 percent of potential determinacy races.

Mellor-Crummey [12] proposes a scheme called "offset-span labeling" in programs with nested fork-join parallelism, a model that exhibits only series-parallel dependences. The idea of his scheme is to store a list of labels for each executing thread. Whenever a thread spawns, the length of the list grows by one, and whenever a thread syncs, the length is reduced by one. This strategy avoids a problem in the English-Hebrew labeling algorithm whereby the length of a label might grow in proportion to the number of spawn operations encountered in the execution path.

Min and Choi [14] propose a determinacy-race detection algorithm that piggybacks on a protocol for distributed shared-memory. The idea is that a determinacy race occurs when a processor accesses memory that was previously accessed by another processor. Consequently, determinacy-race detection can be performed at the same time as the distributed shared-memory protocol, thereby avoiding individual access checks. This reduced overhead is achieved at the cost of additionally storing the history of accesses of each shared location, however. Moreover, the length of the history is proportional to the depth of nested parallelism.

Steele [19] proposes a scheme to detect determinacy races in a programming model with asynchronous threads of control. His scheme requires each location to maintain state information recording the sequence of threads that have accessed the location as well as the type of access performed. In addition, each thread maintains a responsibility set of which locations it has accessed. The Nondeterminator's *reader* and *writer* shadow spaces are similar to Steele's location state, but rather than keeping lists, in our scheme only a single reader and writer need be stored per location. Although he does not mention it, the programs that he is capable of checking exhibit series-parallel dependences. Steele provides an implementation of his algorithm in the Scheme programming language.

The space and time requirements of all these determinacy-race detection algorithms are larger than those for the SP-bags algorithm. Our algorithm spends almost constant time checking each read and write access, and it uses only a constant factor more memory than does the program itself.

8 Conclusion

To conclude this paper, we shall discuss some of the open problems arising out of our work. These problems include how to parallelize our algorithm, whether a faster algorithm for determinacy-race detection might exist, and how to tolerate intended nondeterminism while still catching other determinacy races.

The SP-bags algorithm seems inherently serial, because it heavily relies on the serial execution order of the parallel program. Nevertheless, we feel that it may be possible to develop a parallel version of the SP-bags algorithm. We have started investigating a parallel scheme in which each of several processors executing the program uses the SP-bags algorithm locally, but when a remote child procedure returns, it reconciles its shadow spaces in a manner similar to the BACKER algorithm [4] for maintaining dag consistency. Such a result may be mostly of theoretical interest, however, since debugging is usually done in the development phase of a program using small data sets, and thus typically, the performance of the debugger is not a crucial concern.

Linear-time algorithms for the least-common-ancestors algorithm exist in the literature [10, 18], and it is natural to wonder whether a determinacy-race detector exists that operates in linear time, instead of the almost-linear-time performance of the SP-bags algorithm. The attraction of Tarjan's algorithm, as opposed to existing linear-time algorithms and the seminal algorithm given by Aho, Hopcroft, and Ullman [1], is that it operates, in Mellor-Crummey's words [12], "on the fly." That is, the least common ancestors can be queried during a simple tree walk without ever requiring the entire tree to be expanded at any time. We expect that the discovery of a linear-time on-the-fly least-common-ancestors algorithm would have direct application to determinacy-race detection.

Some programs may intentionally contain nondeterminism. How can a debugging program, such as the Nondeterminator, tolerate intended nondeterminism while still catching unintentional determinacy races?

One strategy that Cilk users have used successfully in debugging nondeterministic codes is for the user to "turn off" the intentional nondeterminism in his code so that he can debug a deterministic version of his program. Our experience is that intentional nondeterminism does not occur in many places in user programs, and the user usually has the ability to disable it. For example, in our ⋆Socrates chess-playing program, a switch was included that could turn off the aspects of the program that produced nondeterministic behavior. Of course, if the user's bug is in the nondeterministic part of his code, this strategy will not work, but knowing that the deterministic part contains no determinacy races is nevertheless extremely helpful during debugging.

Another strategy that the Nondeterminator supports is to allow the user to turn off monitoring of certain variables. For example, our benchmark `knapsack` has an intentional determinacy race when independent threads atomically update the variable containing the bound in its branch-and-bound search. To check this code, we simply disabled the monitoring of the location containing the bound. A disadvantage of this strategy, however, is that turning off the monitoring of one location may hide inadvertent nondeterminism in other locations. Thus, it is not clear what is guaranteed when such a program passes the Nondeterminator test. Nevertheless, turning off the monitoring of certain locations seems to be a useful strategy.

The Nondeterminator has been included in the latest Cilk release [20]. The Nondeterminator in the release runs about 25 percent slower than the one in this paper. We traded off some performance for usability and simplicity. The released version provides more user options in the runtime system, such as a switch for deciding whether floating-point operations should be deemed commutative. (They are not, due to round-off error, but sometimes users wish to ignore the minor nondeterminacies that result.) To simplify the maintenance of the code, the released version also lacks some aggressive compiler optimizations that reduce the amount of instrumentation. Software, the user's manual, and other related information about Cilk and its Nondeterminator are available via the World Wide Web at `http://theory.lcs.mit.edu/~cilk`.

Acknowledgments

We would like to thank Arvind and his dataflow group at MIT for their insightful discussions about the determinacy-race problem. Larry Rudolph of MIT acquainted us with much related work, including his own. Bradley Kuszmaul of Yale University was helpful in providing pointers on related work. Long ago, Guy Blelloch of Carnegie Mellon University pointed out the series-parallel structure of Cilk dags. David Karger and Matt Levine of MIT serendipitously renewed our interest in Tarjan's least-common-ancestors algorithm just before we discovered it was relevant to the determinacy-race problem. The members of our Cilk development group—Robert Blumofe of University of Texas at Austin and Matteo Frigo, Ching Law, Phil Lisiecki, Aske Plaat, Keith Randall, Bin Song, Andrew Stark, and Volker Strumpen of MIT—provided many helpful suggestions and donated their Cilk application programs for testing. Also, many thanks to Keith (Schwartzenegger) Randall for suggesting the name "Nondeterminator."

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestor in trees. *SIAM Journal on Computing*, 5(1):115–132, March 1976.

[2] T. R. Allen and D. A. Padua. Debugging Fortran on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, August 1987.

[3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.

[4] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1990.

[7] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM Press, 1990.

[8] R. J. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10:303–318, 1965.

[9] Perry A. Emrath and Davis A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, May 1988.

[10] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[11] P. A. MacMahon. The combination of resistances. *The Electrician*, April 1892.

[12] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[13] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, San Diego, California, May 1993. ACM Press.

[14] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–244, Palo Alto, California, April 1991.

[15] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[16] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[17] John Riordan and C. E. Shannon. The number of two-terminal series-parallel networks. *Journal of Mathematics and Physics*, 21:83–93, 1942.

[18] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, December 1988.

[19] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 218–231. ACM Press, 1990.

[20] Supercomputing Technology Group, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139. *Cilk-5.0 (Beta 1) Reference Manual*, March 1997. Available on the World Wide Web at URL "`http://theory.lcs.mit.edu/~cilk`".

[21] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, April 1975.

[22] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, October 1979.

[23] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978. STAN-CS-78-682.