On-the-Fly Detection of Determinacy Races in Fork-Join Programs

Jeremy T. Fineman

December 15, 2003

Abstract

Quick determination of series-parallel thread relationships is at the core of some determinacy-race-detections schemes. In this paper, we present two provably good algorithms for maintaining the series-parallel relationships between threads on shared memory systems, on-the-fly. The first algorithm, called the ***SP-order*** algorithm, is sequential, executing a fork-join program asymptotically optimally on one processor. For any two threads that have already executed, we can determine the relationship between them in constant time. For any fork-join program that runs in T_1 time on a single processor, an execution of the program with the SP-order algorithm run asymptotically optimally in $O(T_1)$ time. The SP-order algorithm depends on an order-maintenance structure similar to that proposed by Dietz and Sleator.

The second algorithm, called the ***parallel-SP-order*** algorithm, runs on any number P of processors. As with the serial algorithm, we can determine the relationship between two threads that have already executed in constant time. However, we rely on the work-first principle of the Cilk scheduler to get a good bound. For a Cilk program that runs in $T_P = \Theta(T_1/P + T_\infty)$ time on P processors, where T_1 is the work, and T_∞ is the critical-path, the parallel-SP-order algorithm runs in $O(T_1/P + PT_\infty)$.

1 Introduction

A multithreaded program in the fork-join model contains ***threads***, which are maximal sequences of instructions not containing parallel control statements. Multiple threads concurrently access shared memory. Generally, fork-join programs are meant to be deterministic. That is to say, the program should generate the same output no matter how the threads are scheduled. A ***determinacy race*** can occur, however, when a thread writes to a memory location while another thread is accessing the same location.

Determinacy races are difficult to detect through normal debugging techniques, as the races depend on timing and scheduling of the threads. Inserting breakpoints in a debugger or diagnostic statements into the code alters the timings.

Figure 1 shows an example of a program containing a determinacy race in the Cilk [2] multithreaded language.[1] The `main()` procedure uses the **spawn** keyword to fork a procedure `foo()` and continue executing concurrently with the spawned procedure. Then, the `main()` procedure spawns another instance of the `foo()` procedure. The **sync** keyword suspends `main()` until both instances of `foo()` return. A determinacy race occurs in this program because both instances of `foo()` can execute and write to a location x concurrently.

On-the-fly techniques for detecting determinacy races involve augmenting the original program to report races as they occur. At the core of these techniques is an efficient algorithm for determining the series-parallel relationship between threads efficiently. Mellor-Crummey [5] reviews different techniques for reporting determinacy-races, but we only consider on-the-fly techniques in this paper.

The ***SP-order*** algorithm is an on-the-fly technique for maintaining the series-parallel relationships between threads. This algorithm relies on efficient order-maintenance structures inspired by Dietz and Sleator [3].

The remainder of this paper is organized as follows. In Section 2, we review the series-parallel DAG and parse tree representations of fork-join programs. We present a serial version of the SP-order algorithm in

[1]This example is taken from [4].

```
int x;

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

cilk int main()
{
    x=0;                          /* $e_0$ */
    spawn foo();                  /* $F_1$ */
    . . .                         /* $e_1$ */
    spawn foo();                  /* $F_2$ */
    . . .                         /* $e_2$ */
    sync;
    printf("x is %d\n", x);       /* $e_3$ */
    return
}
```

Figure 1: A simple Cilk program that contains a determinacy race. The thread labels are given in comments on the right. The "..." represent arbitrary sequential code segments.

Section 3. The serial version works efficiently for any serial execution of a fork-join program. In Section 4, we prove the correctness of the SP-order algorithm. In Section 5, we present a naive version of the parallel-SP-order algorithm, which has poor performance due to concurrent accesses in the order-maintenance structure. Section 6 contains a quick review of the Cilk scheduling properties necessary for a good bound on the improved order-maintenance structure and algorithm presented in Section 7. Finally, in Section 8, we prove that the Cilk parallel-SP-order algorithm executes in $O(T_1/P + PT_\infty)$ time on P processors.

2 A model for fork-join program executions

In this section, we review the representations of fork-join programs as series-parallel DAGs and parse trees.

We can represent the parallel control flow of fork-join program as a series-parallel directed acyclic graph, or ***series-parallel DAG***, $G = (V, E)$. Each vertex represents a unique thread in the program. Each edge in the DAG represents a dependency between threads introduced by a parallel control construct. Before formalizing the construction of a series-parallel DAG, we review some notation. On a **spawn** statement, we add a ***continuation edge*** to the thread representing the next sequence of instruction in the current procedure, and we add a ***spawn edge*** to the first thread in the spawned procedure. On a **sync**, we add edges from the last thread of each spawned procedure to the thread following the **sync**. We say that a thread $e_1 \in V$ ***precedes*** a thread $e_2 \in V$, denoted $e_1 \prec e_2$, if there is a path from e_1 to e_2 in the DAG. Conversely, we say that threads e_1 and e_2 operate ***logically in parallel***, denoted $e_1 \parallel e_2$, if $e_1 \not\prec e_2$ and $e_2 \not\prec e_1$.

The DAG for Figure 1 is shown in Figure 2. We see that there is a path from e_1 to e_3, for example, so $e_1 \prec e_3$. There is no directed path between F_1 and F_2, so $F_1 \parallel F_2$.

As a more formal definition, series-parallel DAGs can be constructed recursively as follows:

Base The graph has a single edge $e \in E$ connecting s to t.

Series Composition The graph consists of two series-parallel DAGs, G_1 and G_2, with disjoint edges such that s is the source of G_1, t is the sink of G_2, and the sink of G_1 is the source of G_2.

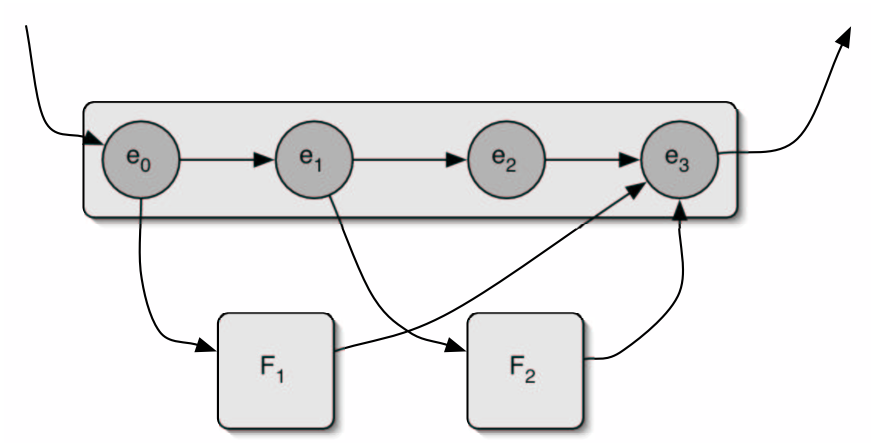

Figure 2: A DAG representing the parallel control flow of the program shown in Figure 1. Threads and procedures are represented by circles and rectangles respectively. Downward edges represent **spawn** dependencies, upward edges represent **return** dependencies, and horizontal edges represent the continuation of a procedure.

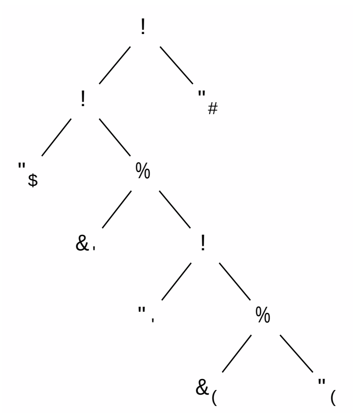

Figure 3: A parse tree representing the parallel control flow of the program shown in Figure 1. The S-nodes indicate a series relationship, while the P-nodes represent a parallel relationship. Each of the leaves is a thread in the program.

Parallel Composition The graph consists of two series-parallel DAGs, G_1 and G_2, with disjoint edges such that s is the source of G_1 and G_2, and t is the sink of G_1 and G_2.

For most of this paper, we use the ***parse tree*** representation of the series-parallel DAG, as described by Feng and Leiserson [4]. Figure 3 shows the parse tree for Figure 1. Each of the leaves in the tree represent threads in the computation. The S-nodes represent series composition, while the P-nodes indicate parallel composition.

A nice property about the parse tree representation is that the least common ancestor of two threads determines their relationship. We revisit this concept in Section 4

3 The SP-order algorithm

In this section, we present a serial version of the SP-order algorithm. First, we discuss what it means to take a total order of a DAG. Then, we review a data structure for maintaining a total order on the fly. Next, we present a serial implementation of the SP-order algorithm. Finally, we prove that the serial SP-order algorithm executes in $O(T_1)$ time, where T_1 is the execution time of the original program.

A total order of a control flow DAG is essentially a topological sort of the DAG, which represents a valid execution of the program. That is to say, for any two threads e_1 and e_2, if $e_1 \prec e_2$ in the DAG, then e_1 precedes e_2 in the total order.

The SP-order algorithm maintains total orders on the fly using a data structure. At the very least, we need the following two operations:

1. INSERT(X, Y): Insert a new element Y immediately after the element X in the order.

2. PRECEDES(X, Y): Return true if X precedes Y in the order.

A naive implementation of the order maintenance structure is a linked list. The INSERT operation a normal insert into the linked list. The PRECEDES query can be implemented simply by starting at X and walking forward in the list until hitting the end (return false) or Y (return true). While this implementation is correct, it will result in poor performance for our algorithm. An insert takes $O(1)$ time, but a query takes $O(n)$ time, where the size of the list is n.

Instead of a simple linked list, the SP-order algorithm uses the Dietz and Sleator [3] order maintenance structure with. This structure is essentially a linked list with labels. A query takes $O(1)$ time by simply comparing the labels of two nodes. Inserts, however, are no longer trivial—we may need to relabel elements in the list to make room for the new element. Even so, the amortized cost of an insert is still $O(1)$.

The SP-order algorithm uses two complementary total-orders to determine whether threads are logically parallel. Recall that for any fork-join program, we have a corresponding parse tree. In both orders, the nodes in the left subtree of an S-node precede those in the right subtree of the S-node. In the ***left-to-right order***, the nodes in the left subtree of a P-node precede those in the right subtree of the P-node. In a ***right-to-left order***, the nodes in the right subtree of a P-node precede those in the left. For example, the left-to-right order for the threads in the parse tree shown in Figure 3 would be $e_0, F_1, e_1, F_2, e_2, e_3$, while the right-to-left order is $e_0, e_1, e_2, F_2, F_2, e_3$.

For pedagogical purposes, suppose that the SP-order algorithm takes as input a parse tree. In practice, one might implement the algorithm by augmenting the fork and join (or, in Cilk, spawn and sync) constructs. We want to execute the program while maintaining relationships between threads. Thus, the algorithm performs a valid walk of the tree, executing the threads of the original program. While walking the tree, the SP-order algorithm code performs INSERT operations into the complementary total-orders. The queries would be found in the leaves which represent the computation of the input program. A determinacy-race detector like the Nondeterminator would perform a query on each memory access of the original program.

The SP-order algorithm, shown in Figure 4, is a serial algorithm, taking advantage that a fork-join program can be executed on a single processor without changing the semantics of the program by executing the leaves of the parse tree in a left-to-right order. This algorithm essentially performs a left-to-right preorder

```
walkTree($X$)
{
    if $X$ is a leaf
       then execute($X$)
            return

    insert left($X$), right($X$) after $X$ in Order1
    if $X$ is an S-node
       then insert left($X$), right($X$) after $X$ in Order2
    if $X$ is a P-node
       then insert right($X$), left($X$) after $X$ in Order2

    walkTree(left($X$))
    walkTree(right($X$))
}
```

```
boolean inSeries($X$, $Y$)
{
    if $X$ precedes $Y$ in both orders
       then return true
       else  return false
{

boolean inParallel($X$, $Y$)
{
    if inSeries($X$, $Y$) or
       inSeries($Y$, $X$)
       then return false
       else  return true
}
```

Figure 4: The SP-order algorithm. The `walkTree` procedure maintains the relationships between threads nodes in a parse tree. A non-leaf node X has a left child, left(X), and a right child, right(X). If X is an S-node, we insert its children in the same order in both order structures. If X is a P-node, we insert its children in different orders in each order structure. The inSeries and inParallel procedures return the relationship between two nodes by querying the order structures.

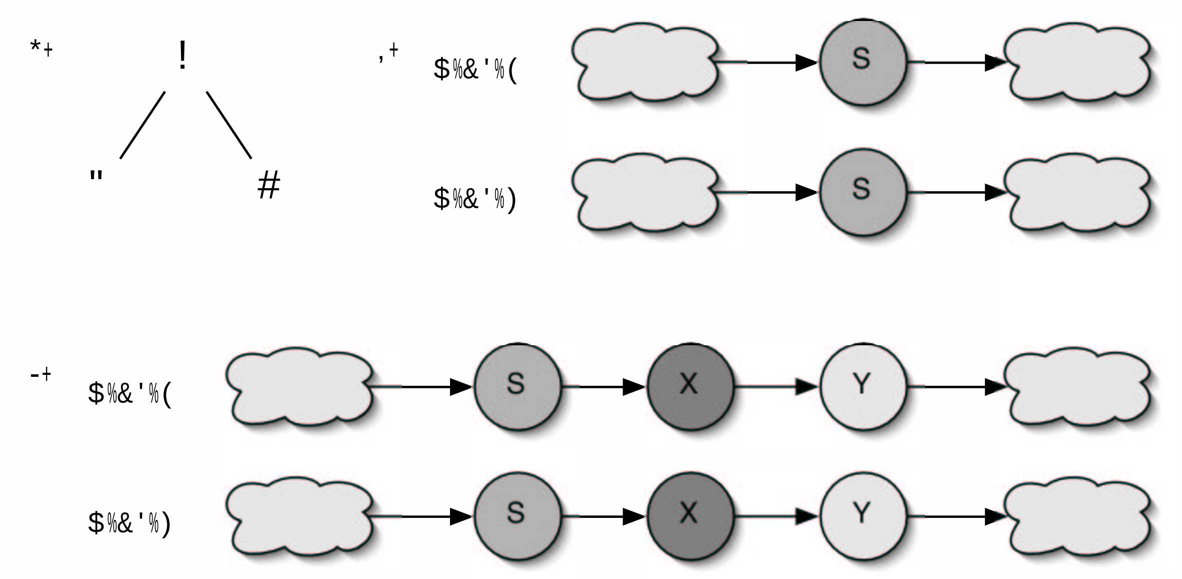

Figure 5: An illustration of the SP-order algorithm at an S-node. (a) shows a simple parse tree with an S-node S and two children L and R. (b) shows the order structures before traversing to S. The clouds are the rest of the order structure, which does not change when traversing to S. (c) shows the result of the inserts after traversing to S. The left child L then the right child R are inserted after S in both lists.

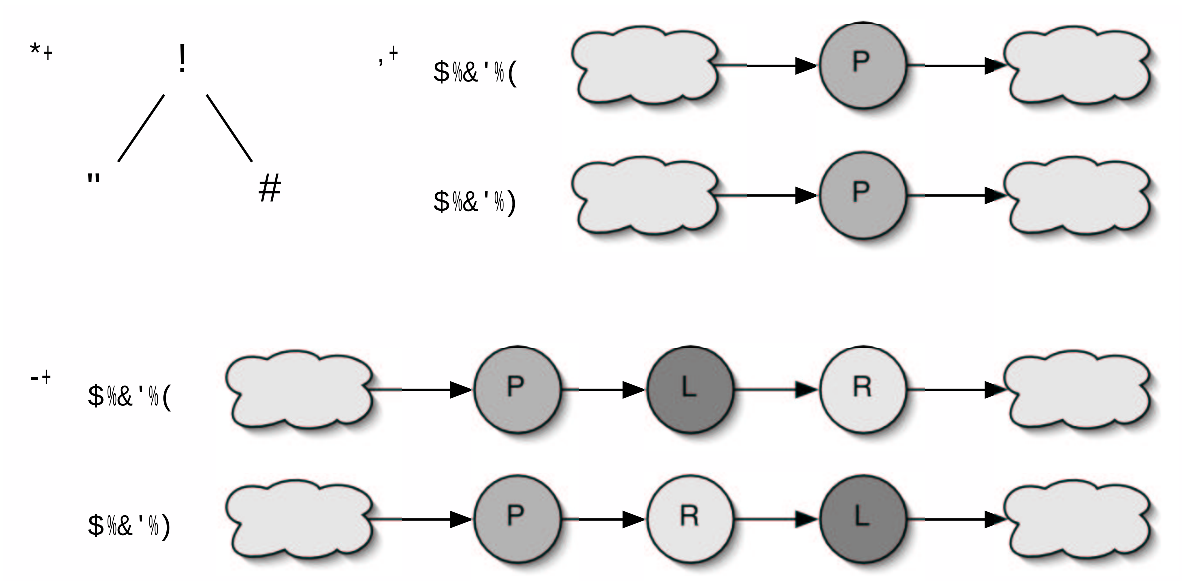

Figure 6: An illustration of the SP-order algorithm at a P-node. (a) shows a simple parse tree with an P-node P and two children L and R. (b) shows the order structures before traversing to P. The clouds are the rest of the order structure, which does not change when traversing to P. (c) shows the result of the inserts after traversing to P. The left child L then the right child R are inserted after P in Order1, and R then L are inserted after P in Order2.

traversal of the parse tree while maintaining two order structures, Order1 and Order2. We maintain the following invariant about these order structures.

- Order1 represents a left-to-right order of the parse tree.
- Order2 represents a right-to-left order of the parse tree.

When traversing to a node X, we perform one of three different actions depending on what type of node X is:

X **is a leaf** X just represents a thread in the original program. Just execute it.

X **is an S-node** Insert the left then right child of X after X in both order structures.

X **is a P-node** Insert the left then right child of X after X in the Order1 (left-to-right) order structure. Insert the right then left child of X after X in the Order2 (right-to-left) order structure.

Figures 5 and 6 illustrate the inserts for an S-node and P-node respectively.

To determine the relationship of two threads e_1 and e_2, we simply compare the relationship of both threads in both orders. If PRECEDES(e_1, e_2) in both orders, then $e_1 \prec e_2$. We prove the correctness of this algorithm in Section 4.

Finally, we analyze the asymptotic running time of the serial SP-order algorithm.

Theorem 1 *Consider a fork-join program that executes in T_1 time on a single processor. Then the SP-order algorithm executes the same computation while maintaining thread relationships in $O(T_1)$ time.*

Proof If there are t threads in the original computation, the parse tree contains $t - 1$ non-leaf nodes. Thus, a walk of the tree takes $O(t)$. Obviously, there can not be more than $O(T_1)$ threads, so the tree walk takes $O(T_1)$ time.

When traversing to each non-leaf node, we insert each child into two order maintenance structures. In other words, we insert each node, exactly once, into two structures. Thus, in total, we perform $2(2t-1) = O(T_1)$ inserts. Using a Dietz and Sleator structure, the amortized cost of all the inserts is $O(T_1)$.

A leaf node is just a thread in the original program. At a leaf node, we simply execute the thread. Since we walk to each leaf exactly once, the computation time is $O(T_1)$. In general, one would augment the original program to include some number of queries to determine thread relationships. As long as the augmentation inserts no more than one query for each line of code, we execute $O(T_1)$ queries, each taking a constant time.

Combining all three of these asymptotic bounds, we get a total of $O(T_1)$ for the SP-order algorithm. $\square$

4 Correctness of the SP-order algorithm

In this section, we prove the correctness of the SP-order algorithm. First, we show that the series-parallel relationship between threads can be inferred from the parse tree. Then we show that our queries into both orders do in fact determine the correct relationship between threads. Finally, we show that the SP-order algorithm maintains the left-to-right and right-to-left orders correctly.

We have the following properties of series-parallel parse trees from Feng and Leiserson:

Lemma 2 *Let e_1 and e_2 be distinct threads in a series-parallel DAG, and let 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 LCA(e_1, e_2) is a P-node.* $\square$

Corollary 3 *Let e_1 and e_2 be distinct threads in a series-parallel DAG, and let 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 LCA(e_1, e_2) is an S-node, e_1 is in the left subtree of LCA(e_1, e_2), and e_2 is in the right subtree of LCA(e_1, e_2).* $\square$

Next, we show that given a left-to-right and right-to-left total order of the DAG, we can determine the relationship between two threads.

Lemma 4 *Consider a series-parallel DAG G and the corresponding parse tree. Consider also the left-to-right and right-to-left total orders as defined in Section 3. Then for any two threads e_1 and e_2 in the DAG, $e_1 \prec e_2$ if and only if e_1 precedes e_2 in the left-to-right and right-to-left orders.*

Proof ($\Longrightarrow$) Suppose $e_1 \prec e_2$ in the DAG G. Then from Corollary 3, we know that LCA(e_1, e_2) is an S-node S in the parse tree, e_1 is in the left subtree of S, and e_2 is in the right subtree of S. By definition of both orders, the nodes in the left subtree of an S-node precede those in the right subtree, so e_1 precedes e_2 in both the left-to-right and right-to-left orders.

($\Longleftarrow$) Suppose e_1 precedes e_2 in both the left-to-right and right-to-left total orders of the DAG. Let X be the LCA(e_1, e_2).

Since e_1 appears before e_2 in the left-to-right order, e_1 must appear in the left subtree of X, and e_2 must appear in the right subtree of X.

Assume for the sake of contradiction that X is not an S-node. Then X is a P-node. Since e_1 precedes e_2 in the right-to-left order, e_1 must appear in the right subtree of X and e_2 must appear in the left subtree of X, which generates a contradiction. Thus, we can apply to Corollary 3 to conclude that $e_1 \prec e_2$. $\square$

Corollary 5 *Consider a series-parallel DAG G and the corresponding parse tree. Consider also the left-to-right and right-to-left orders. then for any two threads e_1 and e_2 in the DAG, $e_1 \parallel e_2$ if and only if e_1 precedes e_2 in one order, but e_2 precedes e_1 in the other.*

Proof This corollary follows from Lemma 4. □

Finally, we show that the SP-order algorithm shown in Figure 4 correctly maintains the series-parallel relationships between threads. First, we have a lemma proving this algorithm maintains the left-to-right and right-to-left orders. We finish this section with a theorem stating that the SP-order-algorithm queries return the correct results.

Lemma 6 *Consider an execution of the SP-order algorithm shown in Figure 4. Suppose x and y are two nodes in the parse tree that have already been inserted into the order-maintenance structures. Then x precedes y in Order1 if and only if x precedes y in the left-to-right order of the parse tree. Similarly, x precedes y in Order2 if and only if x precedes y in the right-to-left order of the parse tree.*

Proof We use induction on the depth of the parse tree.

For a base case, consider the start of the computation (depth of 0 in the parse tree). Both structures are initialized to contain only the root of the tree.

For the inductive step, we assume that the lemma holds for all nodes inserted before executing `walkTree(`z`)`. We need to consider the case of Order1 and Order2 separately. The Order2 case is more complicated, so we consider that first. The proof for Order1 is similar.

Consider Order2 at `walkTree(`z`)`. Suppose that x is any node other than z that is already in Order2. Then x precedes z in Order2 if and only if x precedes z in the right-to-left order. Let a be the least common ancestor of x and z in the parse tree.

Case 1: Suppose that a is an S-node and that x precedes z in Order2. Then by definition of the right-to-left order, x is in the left subtree of a, and z is in the right subtree of a. Thus, left(z) and right(z) are also in the right subtree of a, so left(z) and right(z) follow x in the right-to-left order. Inserting left(z) and right(z) after z is consistent with the right-to-left order.

Case 2: Suppose that a is an S-node and that z precedes x in Order2. Then z is in the left subtree of a and x is in the right subtree of a. Thus, left(z) and right(z) precede x in the right-to-left order. Inserting left(z) and right(z) immediately after z in Order2 means that they are inserted somewhere before x, which is consistent with the right-to-left order.

Case 3: Suppose that a is a P-node and that x precedes z in Order2. Then x is in the right subtree and a and z is in the left subtree of a. The proof for this case is similar to Case 2.

Case 4: Suppose that a is a P-node and that z precedes x in Order2. Then z is in the right subtree of a and x is in the left subtree of a. The proof for this case is similar to Case 1.

Since we do not move any nodes in the order structures, relationships between any two nodes x and y do not change on `walkTree(`z`)`.

So far we have shown that for any node x already in the order structures and either child of z, the relationships between x and the child of z are the same in the left-to-right order and Order1 and for the right-to-left order and Order2. For completeness, we also need to show that the relationships between the two children of z also match. This fact is obvious given the algorithm. For Order1, left(z) is always inserted before right(z). For Order2, left(z) is inserted before right(z) if z is an S-node, but right(z) is inserted before left(z) if z is a P-node. □

Theorem 7 *Consider an execution of the SP-order algorithm. Consider any two threads e_1 and e_2 that have already executed or are currently being executed. Then $e_1 \prec e_2$ if and only if e_1 precedes e_2 in Order1 and Order2. Similarly, $e_1 \parallel e_2$ if and only if e_1 precedes e_2 in exactly one of the orders.*

Proof The SP-order algorithm always inserts a node x into the order structures before executing `walkTree(`x`)`. Thus, any thread is in the order structures before it is executed. Therefore, we can apply Lemma 4, Corollary 5 and Lemma 6 to prove this theorem. □

```
cilk void walkTree($X$)
{
    **if** $X$ is a leaf
        **then** execute($X$)
            **return**

    insert left($X$), right($X$) after $X$ in Order1
    **if** $X$ is an S-node
        **then** insert left($X$), right($X$) after $X$ in Order2
        **spawn** walkTree(left($X$))
        **sync**
        **spawn** walkTree(right($X$))
        **sync**
    **if** $X$ is a P-node
        **then** insert right($X$), left($X$) after $X$ in Order2
        **spawn** walkTree(left($X$))
        **spawn** walkTree(right($X$))
        **sync**
}
```

Figure 7: A naive implementation of the parallel-SP-order algorithm, based on the SP-order algorithm shown in Figure 4. At an S-node, the left child must execute before the right child. At a P-node, both children may execute in parallel.

5 The parallel-SP-order algorithm

In this section, we look at parallelizing the SP-order algorithm. First, we notice that the correctness of the algorithm is not a difficulty to overcome. We propose a slight variation of the SP-order algorithm to create the parallel-SP-order algorithm. For efficient asymptotic running time, however, we need an efficient concurrent-order-maintenance structure, which we will discuss in Section 7.

If we look at the proofs for Theorem 7 and Lemma 6, we see that they do not rely on the serial execution of the SP-order algorithm. Thus, we can perform any *valid* serial or parallel walk of the parse tree while maintaining the same correctness properties. A ***valid walk*** of the parse tree is one that reflects a valid execution of the program. That is to say, a thread is not executed before any of the threads it depends on. Thus, at a P-node, both children can execute in parallel, but at an S-node, we need to execute the left child before the right child. Figure 7 shows the parallel-SP-order algorithm, with Cilk-like syntax, performing a valid walk of the parse tree.

The parallel-SP-order algorithm depends on a concurrent-order-maintenance structure. A naive implementation of this structure would be simply to lock the entire structure on every insert. While this approach does yield a correct implementation, it can perform quite poorly. If there are $\Theta(T_1)$ threads in the program, then an insert into one of the order structures may need to relabel $\Theta(T_1)$ nodes. Thus, the critical path of the computation can increase to T_1, yielding a very poor performance of $\Omega(T_1/P + T_1) = \Omega(T_1)$ in the worst case.

The biggest hit to performance with the naive implementation of the concurrent-order-maintenance structure is the size of the list—if we bound the size of the order structure to less than T_1, we can significantly improve the performance.

6 Cilk scheduler

In this section, we review the Cilk scheduler [1], which uses a provably good work-stealing algorithm. Then we present a bound of $O(PT_\infty)$ steal attempts with high probability.

The Cilk scheduler uses a ready deque for each processor to queue up threads that can be stolen. Threads are always inserted into the deque on the bottom of the deque, but they can be deleted from either end. A processor p works on a procedure α until α

- **spawns procedure β:** p working pushes the next thread in α on the ready deque and starts working on the procedure β.
- **returns:**
  - if the deque is nonempty, p pops a thread from the bottom of its deque and begins working on it.
  - if the deque is empty and no processor is working on α's parent—the procedure that spawned α—then p begins resumes working on α's parent.
  - if the deque is empty and α's parent is busy, then work steal.
- **syncs:** If α has outstanding children, then work-steal. Note that the deque is empty in this case.

When a processor p tries to steal, it operates as follows:

- p chooses a victim uniformly at random.
- if the victim's deque is empty, p tries to steal again.
- if the victim's deque is non-empty, p steals the top thread of the victim's deque and begins working on it.

One important property of the Cilk scheduler is that we can bound the number of steals attempts to $O(PT_\infty)$. We state the following lemma from Blumofe /citecilkscheduler without proof:

Lemma 8 *Consider an execution of any fully strict multithreaded computation with critical path T_∞ by the Work-Stealing Algorithm on a parallel computer with P processors. For any $\epsilon > 0$, with probability at least $1 - \epsilon$, at most $O(P(T_\infty + \lg(1/\epsilon)))$ work-steal attempts occur. The expected number of steal attempts is $O(PT_\infty)$.* □

7 Concurrent order-maintenance of Cilk threads

In this section, we consider how to apply the Cilk work-stealing algorithm with a bounded number of steals to the parallel-SP-order algorithm. First, we look at what a steal means in terms of the parse tree. Then we propose a modified order-maintenance structure to use for the parallel-SP-order algorithm. Finally, we rewrite the algorithm to more explicitly manage the data structures.

First, let us consider a steal attempt in the parallel-SP-order algorithm shown in Figure 7. The procedure `walkTree(`X`)` essentially only has one point at which a steal can occur: `walkTree(right(`X`))`.

We define a ***subcomputation*** to be a largest subtree of the original parse tree such that:

- Every node in the subtree that has been traversed has been executed by the same processor.
- Suppose X is a node in the subcomputation. Then the left child of X is in the subcomputation if and only if the right child of X is in the subcomputation.

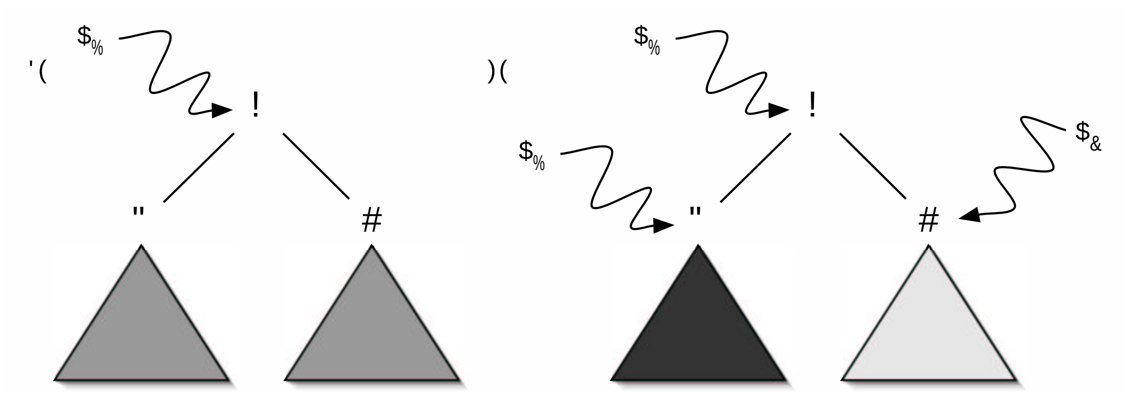

Figure 8: An illustration of the subcomputations created through a steal in terms of the parse tree. (a) shows a single subcomputation begin working on by the processor p_1. In (b), the processor p_2 steals `walkTree(right(`X`))` from the processor p_1. There are now three subcomputations.

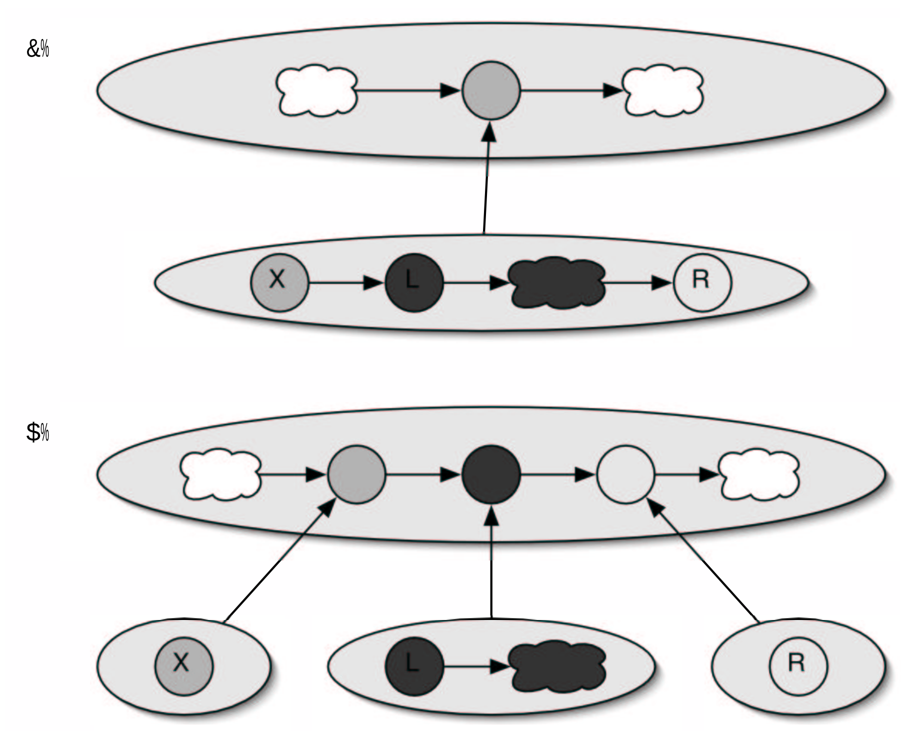

Figure 9: An illustration of the two-level order-maintenance structure representing the left-to-right order of the parse tree generated from the subcomputations shown in Figure 8. The ordering of subcomputations is shown in the top level of each list, while the lower level shows the ordering of nodes within a subcomputation. The clouds indicate other nodes in the list we are not focusing on. a) shows the single subcomputation before the steal. In (b), we see that the subcomputation splits into three subcomputations, and new nodes are inserted into the top level to represent the new subcomputations.

For example, at the start of the computation, only the root node has been traversed, so the entire tree is a subcomputation. Once a steal occurs, however, we create more subcomputations. Figure 8 illustrates the subcomputations created in a steal. At first, p_1 owns the entire subcomputation rooted at the node X. Then, p_2 steals from p_1. The procedure `walkTree(right(`X`))` must be at the top of p_1's deque at this point, so p_2 steals the right subtree of X. Thus, we now have three subcomputations.

Now, let us revisit the concurrency problem of the order-maintenance structure. By definition, only one processor handles all the inserts related to a single subcomputation. Thus, we propose a two level oder-maintenance structure. The top level is the concurrent order-maintenance structure that locks on every insert, while the bottom level is the regular, serial, Dietz and Sleator order-maintenance structure ordering the nodes within a subcomputation. On a steal, we split a subcomputation and perform an insert into the top level structure. Figure 9 shows the left-to-right order structure generated by subcomputations shown in Figure 8.

In this new scheme, all the nodes in the parse tree occur within a subcomputation. That is to say, the nodes of the parse tree are on the bottom level in the order structure. To query the relationship between two nodes, we just need to query the relationship between the subcomputations in the top level. If both nodes belong to the same subcomputation, we query the relationship between the nodes in the lower level order structure.

Next, we rewrite the parallel-SP-order algorithm in Figure 10 to deal with the new data structure. We now allow inserts before or after an element in the order structures. Since we have a lot of different order structures around, data types can get confusing. Thus, we are more explicit about typing the variables.

8 Analysis of the parallel-SP-order algorithm

In this section, we show that the parallel-SP-order algorithm executes in $O(T_1/P + PT_\infty)$ time on P processors, where T_1 is the work and T_∞ is the critical-path.

Theorem 9 *The parallel-SP-order algorithm shown in Figure 10 executes in $O(T_1/P + PT_\infty)$ time with high probability, where P is the number of processors, T_1 is the work, and T_∞ is the critical path.*

Proof Applying Lemma 8, there are $2 \cdot O(PT_\infty) = O(PT_\infty)$ subcomputations. Each subcomputation is inserted into the top-level, concurrent, order-maintenance structure. The entire top-level structure is locked on each insert. Let us suppose that all other processors stop working entirely on an insert into the top-level structure. Then we have $O(PT_\infty)$ inserts occurring serially, taking a total of $O(PT_\infty)$ time.

As for the inserts into the lower level, there can be $O(T_1)$ threads, so there can be $O(T_1)$ inserts. However, these inserts always occur within a subcomputation, so there is no locking involved. Thus, we increase the work by $O(T_1)$.

Adding up these two asymptotic bounds, we get a total running time of $O(T_1/P + PT_\infty)$. $\square$

9 Conclusions

We have presented two efficient algorithms for determining series-parallel thread relationships. The serial version of the SP-order algorithm is asymptotically optimal.

We do not prove the Cilk, parallel-SP-order algorithm to be asymptotically optimal, but it still quite efficient. It has several advantages which are not reflected in the upper bound shown in Section 8. First, the upper bound is a worst case scenario. For normal programs, we would expect steals to be more evenly distributed. If the steals occur more evenly across the parse tree, then we do not need to relabel the top-level list as many times. Furthermore, given that we expect there to be quite a bit of computation in the average thread, we would not expect there to be P processors trying to perform concurrent inserts very frequently.

```
OrderNode* newOrder(void *data);
OrderNode* insertBefore(OrderNode* X, void *data);
OrderNode* insertAfter(OrderNode* X, void *data);

cilk void walkTree(TreeNode *X)
{
    if (leaf(X)) {
        execute(X);
        return;
    }

    TreeNode *L = left(X);
    TreeNode *R = right(X);
    L->Order1 = insertAfter(X->Order1, L);
    R->Order2 = insertAfter(L->Order1, R);

    if (SNode(X)) {
        L->Order2 = insertAfter(X->Order2, L);
        R->Order2 = insertAfter(L->Order2, R);
        spawn walkTree(left(X));
        sync;
        spawn walkTree(right(X));
        sync;
    }
    if (PNode(X)) {
        R->Order2 = insertAfter(X->Order2, R);
        L->Order2 = insertAfter(R->Order2, L);
        spawn walkTree(left(X));
        if (!SYNCHED) {
            /* a steal happened */
            /* split the subcomputation for Order1
            OrderNode *subc1 = newOrder(X);
            OrderNode *subc2 = newOrder(R);
            X->Order1 = insertBefore(parent(L->Order1), subc1);
            R->Order1 = insertAfter(parent(L->Order1), subc2);
            /* split the subcomputation for Order2
            subc1 = newOrder(X);
            subc2 = newOrder(R);
            X->Order2 = insertBefore(parent(L->Order1), subc1);
            R->Order2 = insertBefore(parent(L->Order1), subc2);
        }
        spawn walkTree(right(X));
        sync;
}
```

Figure 10: The efficient parallel-SP-order algorithm. This implementation is similar to that shown in Figure 7 except that we add the "if (!SYNCHED)" block to execute on a steal. On a steal, we create two new subcomputations representing the node X and the node R. In both orders, we insert the subcomputation for X before that for L. In Order1, R comes after L. In Order2, it comes before.

Finally, our the label length is constant. Thus, queries always take constant time. This fact is important since a determinacy race detector may perform $O(T_1)$ queries.

10 Related Work

Nudler and Rudolph [6] present an English-Hebrew labeling for fork-join programs. Each thread is assigned two labels, similar to our left-to-right and right-to-left labeling schemes. They do not, however, use a centralized data structure. Instead, label size can grow proportionally to the maximum concurrency of the program.

Mellor-Crummey [5] proposes an "offset-span labeling" for fork-join programs that has shorter label lengths than the English-Hebrew scheme. However, the offset-span solution still has label lengths proportional to the maximum fork nesting.

References

[1] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

[2] 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.

[3] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *STOC*, pages 365–372, 1987.

[4] Charles. E. Leiserson and Mingdong 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.

[5] 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, 1991.

[6] 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.