Parallel Algorithms for the Circuit Value Update Problem

Charles E. Leiserson
cel@mit.edu

Keith H. Randall
randall@theory.lcs.mit.edu

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

The **circuit value update problem** is the problem of updating values in a representation of a combinational circuit when some of the inputs are changed. We assume for simplicity that each combinational element has bounded fan-in and fan-out and can be evaluated in constant time. This problem is easily solved on an ordinary serial computer in $O(W + D)$ time, where W is the number of elements in the altered subcircuit and D is the subcircuit's embedded depth (its depth measured in the original circuit).

In this paper, we show how to solve the circuit value update problem efficiently on a P-processor parallel computer. We give a straightforward synchronous, parallel algorithm that runs in $O(W/P + D \lg P)$ expected time. Our main contribution, however, is an optimistic, asynchronous, parallel algorithm that runs in $O(W/P + D + \lg W + \lg P)$ expected time, where W and D are the size and embedded depth, respectively, of the "volatile" subcircuit, the subcircuit of elements that have inputs which either change or glitch as a result of the update. To our knowledge, our analysis provides the first analytical bounds on the running time of an optimistic algorithm.

1 Introduction

Many applications—including event-driven logic simulation [2, 3, 11], spreadsheet calculation, and constraint propagation in artificial intelligence programs [13]—maintain a directed acyclic graph (dag) in which each vertex contains a value and each edge represents a data dependency between two values. The value of a vertex is determined by a vertex-specific function of the values of the vertex's immediate predecessors. During the execution of the application, it is common for either the value or the function at a node to be changed, in which case the effects of that change must be propagated throughout the dag. Since propagating changes is an expensive operation, it is important that as few elements be reevaluated as possible.

This research was supported in part by the Advanced Research Projects Agency under Grant N00014-94-1-0985. Keith H. Randall is supported in part by a Department of Defense NDSEG Fellowship.

We abstract this class of problems as the **circuit value update problem**. We are given a combinational circuit as a dag $G = (V, E)$, where the vertices are either combinational elements or circuit inputs and the edges represent interconnections. Figure 1 illustrates the circuit value update problem using a combinational logic circuit. Each combinational element $v \in V$ computes a function f_v of its inputs to produce a single value $value(v)$ which is fanned out to v's immediate successors in the circuit. In the figure, the functions are the boolean connectives AND, OR, and NOT, represented by traditional logic gates. We presume that the circuit has already been evaluated, as is shown in part (a) of the figure. The circuit value update problem is to update the values as efficiently as possible when one or more of the circuit inputs change. For example, part (b) of the figure shows the values that must be updated when Input C is changed from a 0 to a 1.

The amount of work required to solve an instance of the circuit value update problem depends on which part of the circuit is affected by the input change. We define the **altered subcircuit**, shown with darkened gates in Figure 1(b), as the subcircuit of elements whose inputs settle to new values after a change to the circuit inputs. A race condition may cause some values outside the altered subcircuit to change temporarily, or **glitch**, however, thereby affecting other elements. In Figure 1(c), Gate H computes the OR of two inputs, one of which changes from 0 to 1 and the other of which changes from 1 to 0. If the second signal arrives before the first, Gate H's value goes to 0 temporarily before returning to 1, thereby creating a glitch. A glitch can propagate through the circuit and affect combinational elements outside the altered subcircuit. In our example, the glitch in the output of Gate H can cause the output of Gate K to glitch. We define the **volatile subcircuit**, shown with darkened gates in Figure 1(c), as the altered subcircuit together with any elements whose inputs might glitch if the circuit has indeterminate delays.

The depth of a circuit is the number of edges in a longest path in the circuit, but we shall also be interested in the "embedded depth" of a subcircuit $G' = (V', E')$ within a circuit $G = (V, E)$, which we define as follows. The **depth** $d(v)$ of a vertex $v \in V$ is the number of edges in a longest path in G that ends at v. The **embedded depth** of G' within G is $\max_{u,v \in V'} \{d(u) - d(v)\}$. Although the embedded depth of a subgraph can be larger than its true depth, this situation can only occur when the change to an input value of a vertex does not cause a change in the vertex's output. The volatile subcircuit shown in Figure 1(c) has depth 4 and embedded depth 6.

In this paper, we present three algorithms for the circuit value update problem. Each algorithm is preceded by a prepro-

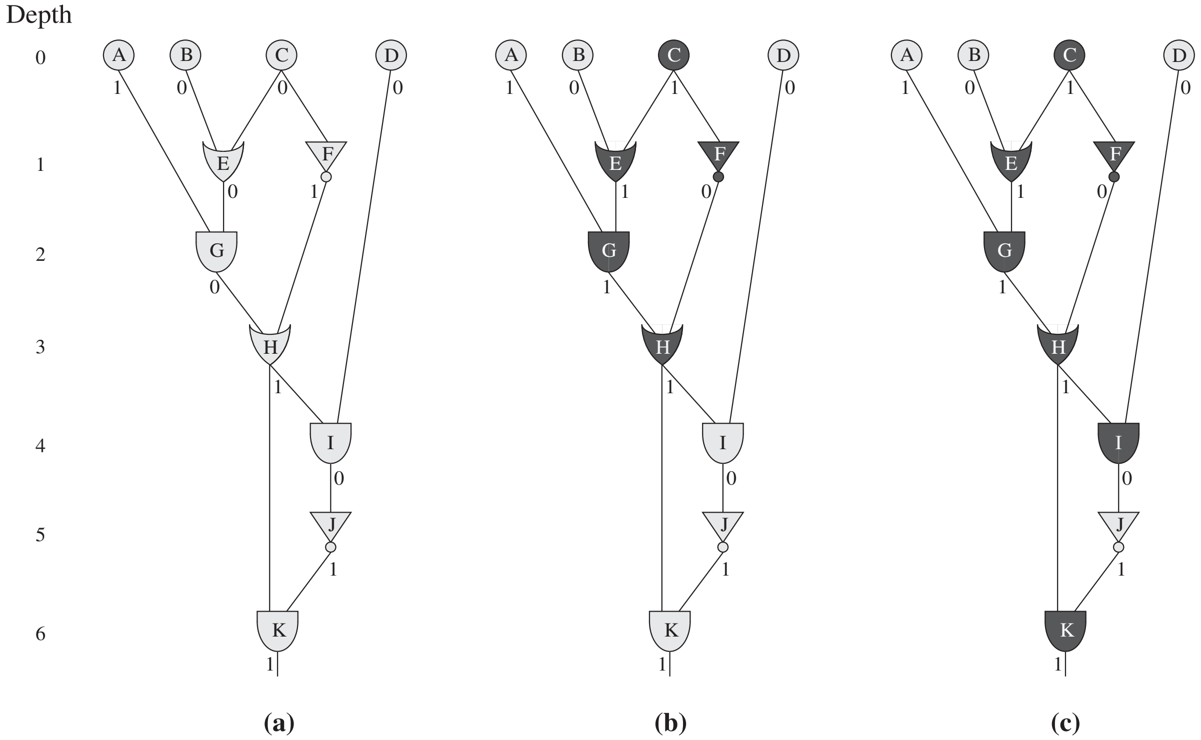

Figure 1: A combinational logic circuit illustrating the circuit value update problem. **(a)** The original circuit whose elements are labeled with their values. The depths of elements are shown on the left side. **(b)** The altered subcircuit when input C is changed from a 0 to a 1. **(c)** The volatile subcircuit. The outputs of Gates H and K may glitch.

cessing phase in which the entire circuit is evaluated and static properties of the circuit are calculated. The running times of the preprocessing phases are not included in the running times of the update algorithms. In general, the preprocessing phases run in time proportional to a standard algorithm for computing all values in the circuit. Our goal, however, is to make an incremental update much faster than a complete reevaluation, since we expect many updates for a single circuit.

For simplicity, we shall assume that the input circuit has bounded degree, although analagous results can be obtained for the unbounded-degree case. We shall also assume that every combinational element takes constant time to evaluate.

The first algorithm we present is a straightforward serial algorithm which runs in $O(W + D)$ time, where W and D are the size and embedded depth, respectively, of the altered subcircuit. This simple algorithm is nearly optimal (for a serial computer), since $\Omega(W)$ is a lower bound on the running time of any serial algorithm. Moreover, the $O(D)$ term, which comes from the need to maintain a priority queue, is dominated by the $O(W)$ term except possibly when the altered subcircuit does not populate consecutive depths in the input circuit.

The second algorithm we present is a synchronous, distributed-memory, parallel algorithm that assumes that circuit elements have been allocated at random to processors. The algorithm runs in $O(W/P + D \lg P)$ expected time on P processors, where W and D are the size and embedded depth of the altered subcircuit. The W/P term in this running time is tight, because at least W work must be done and there are at most P processors to do it. The $D \lg P$ term reflects a blow-up of a factor of $\lg P$ compared to the serial algorithm, due to barrier synchronization.

Our main result is an asynchronous, distributed-memory, parallel algorithm for the circuit update problem which is "optimistic" in the sense that it does not wait for all inputs to a circuit element to settle before it enqueues the element for evaluation. Consequently, some elements may be evaluated repeatedly before the algorithm completes. Nevertheless, we prove that the algorithm runs in $O(W/P + D + \lg W + \lg P)$ expected time on P processors, where W and D are the size and embedded depth of the volatile subcircuit. Since the volatile subcircuit is at least as large as the altered subcircuit, this algorithm may be no better than the synchronous algorithm, but it is simpler in that it requires less global control. Moreover, for glitch-free updates—as necessarily occur for monotone circuits, for example—this optimistic algorithm outperforms the synchronous one.

Other researchers have investigated similar concerns. Chandy and Misra [2] present an asynchronous, parallel algorithm for performing event-driven simulation of a combinational or sequential circuit. Although they attempt to minimize the amount of recalculation, they do not perform sparse updating as do our algorithms. Jefferson [7] proposes an optimistic strategy called "time-warping" for parallel event-driven simulation. Time-warping evaluates elements optimistically and uses "antimessages" to rollback computations that evaluate prematurely. Matsumoto and Taki [11] present a concrete implementation of Jefferson's ideas, and Choi, Hwang, and Blank [3] present a time-warping strategy for performing incremental updates on a circuit. Time-warping is apparently a practical strategy for performing sparse updating, but to date, its effectiveness has not been demonstrated theoretically. Although our algorithms apply in a more restricted setting, we have been able to provide provable bounds on their performance, and they require no complex mechanisms such as antimessages.

The remainder of this paper is organized as follows. Section 2 presents the simple serial algorithm for the circuit value update problem. Section 3 describes the synchronous, parallel algorithm, and Section 4 describes the optimistic, asynchronous, parallel algorithm. The running time of the optimistic, asynchronous algorithm is analyzed in Section 5 using a delay-sequence argument. Finally, Section 6 offers directions for future research.

2 A Serial Algorithm

In this section, we give a naive serial algorithm for the circuit value update problem on a circuit $G = (V, E)$. This algorithm serves as a good "strawman" for the parallel algorithms presented in Sections 3 and 4. On a bounded-degree circuit, the serial algorithm can solve the circuit value update problem in $O(W + D)$ time, where W and D are the size and embedded depth, respectively, of the altered subcircuit.

Before the update algorithm is executed, the circuit is preprocessed to compute the depth of each combinational element. During the update algorithm, element depths are used as keys in a priority queue to determine the evaluation order.

Pseudocode for the serial update algorithm is given in Figure 2. To start a circuit update, line 1 inserts the base set of updated elements into a priority queue Q. Lines 2–8 are executed until Q is empty, in which case all updates to the altered subcircuit have been performed. Lines 3–4 remove an element u from Q and recompute u's value by applying u's combinational function f_u to its inputs. Line 5 determines whether u's value has changed, and if it has then lines 6–8 update u's value and insert all of u's immediate successors into Q, since their values may have changed. (In the pseudocode, the set of immediate successors of u is denoted $Adj[u]$.)

```
SERIAL
1  Q ← set of updated elements
2  while Q ≠ ∅
3     do u ← EXTRACTMIN(Q)
4        new ← f_u(u's inputs)
5        if new ≠ value[u]
6         then value[u] ← new
7              for v ∈ Adj[u]
8                 do INSERT(Q, v)
```

Figure 2: Pseudocode for the serial algorithm.

This algorithm evaluates the elements in order of depth so as to maintain the invariant that no element is evaluated until all of its inputs attain their correct values. Consequently, the one and only time that each element is evaluated, it attains its final value. Thus, the only elements evaluated by this algorithm are those in the altered subcircuit, because these elements are the only elements whose inputs change. The total work in evaluating these elements is $O(W)$.

The priority queue Q can be implemented using an array of linked lists indexed by depth. This implementation yields $O(1)$ time for each insertion, and since there are $O(W)$ insertions, the total time for insertions is $O(W)$. To extract the minimum element from Q requires scanning from the location in the array of the previous minimum to the first nonempty slot and then removing an element from the linked list. Since there are $O(W)$ EXTRACTMIN operations performed, the total time is $O(W)$ plus the $O(D)$ time to scan the array from the initial index to the final index, where D is the embedded depth of the altered subcircuit. Thus, the total time is $O(W + D)$. We can also use other priority queue implementations to obtain different tradeoffs. For instance, using a binary heap [4, Chapter 7] yields a running time of $O(W \lg W)$, and using Johnson's $O(\lg\lg D)$ queue [8] produces a running time of $O(W \lg\lg D)$.

3 A Synchronous, Parallel Algorithm

This section describes a synchronous, parallel algorithm for the circuit value update problem on a circuit $G = (V, E)$. This algorithm extends the serial algorithm from Section 2 to run on multiple processors while maintaining the invariant that no element is evaluated until all of its inputs attain their correct values. This algorithm can solve the circuit value update problem on P processors in $O(W/P + D \lg P)$ expected time, where W is the number of elements in the altered subcircuit and D is the embedded depth of the altered subcircuit. This synchronous algorithm provides us with a baseline against which we can compare the optimistic, asynchronous, parallel algorithm in Section 4.

As a preprocessing phase, we compute the depths of each circuit element and distribute the circuit across the P processors by assigning each element to a random processor. We use a random mapping, because a deterministic mapping that can handle all incremental updates in a load-balanced fashion does not exist: for any mapping one can choose an incremental update that lies mostly on one processor. Randomization, on the other hand, gives us a good mapping for "most" incremental updates without knowing in advance what updates will be performed.

Pseudocode for the synchronous algorithm is given in Figure 3. At each step of the algorithm, all processors reevaluate elements with the same depth d. Line 1 initializes d to the minimum depth of any updated element. Lines 2 and 3 initialize a priority queue $Q[p]$ on each processor p to hold the updated elements assigned to that processor. While there is still work left, lines 5–12 evaluate every updated element at depth d on every processor. Line 13 performs a barrier to synchronize all of the processors and make sure that they have all finished depth d before moving on to depth $d + 1$, and then line 14 increments the current depth when all processors are ready to proceed. Line 4 determines when there is no work left on the entire machine. (In the pseudocode, the processor to which element v is assigned is denoted $proc[v]$.)

The barrier in line 13 maintains the same invariant as in the serial algorithm: no element is evaluated until its inputs attain their correct values. Without the barrier, some processors might start working on depth $d + 1$ before depth d is finished. If this situation were allowed to occur, a depth-$(d + 1)$ element might be executed before receiving some input from a depth-d element that has not yet executed, thereby breaking the invariant.

The running time of the synchronous algorithm depends on the speed of the global barrier in line 13 and of the global control implicit in line 4. Both of these can be implemented to run in $O(\lg P)$ time by forming a tree of processors.

We now analyze the running time of the synchronous algorithm. We can use the same priority queue implementation for each $Q[p]$ as we did for Q in the serial algorithm. Priority queue operations cost $O(1)$ for each insertion and deletion, as well as $O(D)$ overall for scanning the array. The time spent at each depth d depends on the number W_d of elements of

SYNCHRONOUS

1 $d \leftarrow$ minimum depth of updated elements
2 **forall** processors $p \in [1 \,..\, P]$
3   **do** $Q[p] \leftarrow$ set of updated elements assigned to p
4 **while** $\exists i$ such that $Q[i] \neq \emptyset$
5   **do forall** processors $p \in [1 \,..\, P]$
6   **do while** $depth[\text{MIN}(Q[p])] = d$
7   **do** $u \leftarrow \text{EXTRACTMIN}(Q[p])$
8                 $new \leftarrow f_u(u\text{'s inputs})$
9   **if** $new \neq value[u]$
10   **then** $value[u] \leftarrow new$
11   **for** $v \in Adj[u]$
12   **do** $\text{INSERT}(Q[proc[v]], v)$
13   **barrier**
14         $d \leftarrow d + 1$

Figure 3: Pseudocode for the synchronous algorithm.

the altered subcircuit at that depth. Because of the barrier, every processor must wait for the slowest processor to finish. If $W_d = \Omega(P \lg P)$, then with high probability, no processor has more than $O(W_d/P)$ work, which follows from basic properties of the binomial distribution. If $W_d = O(P \lg P)$, however, then with high probability, no processor gets more than $O(\lg P)$[1] work. Thus, the expected time to complete depth d is $O(W_d/P + \lg P)$. Summing over the D levels, we obtain an expected total running time of $O(W/P + D \lg P)$.

4 An Optimistic, Asynchronous, Parallel Algorithm

In this section we describe an optimistic, asynchronous parallel algorithm for solving the circuit value update problem on a circuit $G = (V, E)$. This algorithm improves on the synchronous algorithm from Section 3 because it has no barrier synchronization, making its scheduling decisions simple and local. This algorithm can solve the circuit value update problem on P processors in $O(W/P + D + \lg W + \lg P)$ expected time, where W is the number of elements in the volatile subcircuit and D is the embedded depth of the volatile subcircuit. In Section 5, we actually prove the stronger result that for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the optimistic algorithm runs in time $O(W/P + D + \lg W + \lg P + \lg(1/\epsilon))$.

The bound on the running time of this algorithm depends on the size of the volatile subcircuit, as opposed to the size of altered subcircuit. This fact is an inevitable consequence of using an asynchronous, event-driven algorithm [7], because if a glitch can occur in a circuit, it can occur in the simulation as well. If the altered subcircuit is glitch-free, however, the optimistic algorithm is asymptotically faster than the synchronous one.

Some aspects of the optimistic algorithm are similar to the synchronous algorithm. The preprocessing phase for the optimistic algorithm computes the depth of each element and distributes the elements randomly across the processors exactly as in the preprocessing phase for the synchronous algorithm. The optimistic algorithm also maintains local priority queues $Q[p]$ on each processor p, as in the synchronous algorithm. These priority queues are once again implemented as arrays of linked lists indexed by depth and hold elements that need to be reevaluated.

Unlike the synchronous algorithm, however, the optimistic algorithm does not use a barrier, and thus, at a given time, different processors may be evaluating elements from different depths. The algorithm is "optimistic" in that it schedules an element to be evaluated as soon as one of its inputs has changed without worrying about whether all of its inputs have attained their final values. Consequently, elements may need to be evaluated more than once.

Pseudocode for the optimistic algorithm is given in Figure 4. Lines 1 and 2 initialize the priority queue $Q[p]$ on each processor p to hold the updated elements assigned to that processor. While there is still work left, lines 4–11 execute the shallowest element on each processor in the same manner as in the serial and synchronous parallel algorithms. These lines are repeated until line 3 determines that there is no work left.

OPTIMISTIC

1 **forall** processors $p \in [1 \,..\, P]$
2   **do** $Q[p] \leftarrow$ set of updated elements assigned to p
3 **while** $\exists i$ such that $Q[i] \neq \emptyset$
4   **do forall** processors $p \in [1 \,..\, P]$
5   **do if** $Q[p] \neq \emptyset$
6   **then** $u \leftarrow \text{EXTRACTMIN}(Q[p])$
7                 $new \leftarrow f_u(u\text{'s inputs})$
8   **if** $new \neq value[u]$
9   **then** $value[u] \leftarrow new$
10   **for** $v \in Adj[u]$
11   **do** $\text{INSERT}(Q[proc[v]], v)$

Figure 4: Pseudocode for the optimistic algorithm.

It is worthwhile to compare the pseudocode for the optimistic algorithm with that for the serial and synchronous algorithms. The optimistic algorithm is essentially the same as the serial algorithm, except that the optimistic algorithm uses distributed queues and requires distributed termination detection. The distributed termination detection in line 3 of the optimistic algorithm is triggered when all queues are empty. The optimistic algorithm also resembles the synchronous algorithm, except that it lacks the global barrier and the test for performing an EXTRACTMIN that the synchronous algorithm uses in line 6.

Since the optimistic algorithm does not have a barrier, it does not maintain the invariant that no element is evaluated until its inputs attain their correct values, as the serial and synchronous algorithms do. The algorithm is nevertheless correct. Some element v may be evaluated optimistically before its predecessor u is evaluated. When u is later evaluated, v might need to be reevaluated to obtain its correct value. Although the optimistic execution of v causes extra work, all elements eventually attain their final values.

The global synchronization for termination detection in line 3 of the optimistic algorithm should be implemented using a split-phase or fuzzy barrier [5, 6, 10] so that work can continue while the barrier is being evaluated. This implementation

[1] A tighter analysis gives $O(\lg P / \lg(P \lg P / W_d))$ for $W_d \leq P \lg P/2$ [9, Theorem 3.24].

imposes an $O(1)$-time cost per barrier per processor, and a barrier latency of $O(\lg P)$. We incur this latency penalty only when the algorithm terminates, which means that the total running time is only $O(\lg P)$ more than the running time without termination detection.

5 Analysis of the Optimistic Algorithm

In this section, we prove that for any $\epsilon > 0$, with probability $1-\epsilon$ the optimistic algorithm runs in $O(W/P + D + \lg W + \lg P + \lg(1/\epsilon))$ time. We prove this bound by using a delay-sequence argument [12]. We show that when the running time of an execution is large, there must be a long delay sequence in that execution. We then show that with high probability there are no long delay sequences. These facts together imply that with high probability the running time of an execution is small.

First, we define a delay sequence. In a circuit $G = (V, E)$, a **delay sequence** is a sequence of k pairs $\langle (v_1, p_1), (v_2, p_2), \ldots, (v_k, p_k)\rangle$ where for $i \in [1\,..\,k]$ we have $v_i \in V$ and $p_i \in [1\,..\,P]$. Furthermore, the following conditions must hold:

A. $depth(v_1) \le depth(v_2) \le \cdots \le depth(v_k)$.

B. For $i \in [2\,..\,k]$, if $p_{i-1} \ne p_i$, then there exists an edge $(v_{i-1}, v_j) \in E$ with $j \ge i$ such that $p_j = p_i$ and $depth(v_j) = depth(v_i)$.

C. The combinational elements $v_1, v_2, \ldots, v_k$ are distinct.

We now show that if the running time of the optimistic algorithm is large, then there exists a long delay sequence.

Lemma 1 *Suppose that an update is performed by the optimistic algorithm on a circuit $G = (V, E)$, that D is the embedded depth of the volatile subcircuit, and that the running time ignoring termination detection is at least T. Then, there exists a delay sequence $\langle (v_1, p_1), (v_2, p_2), \ldots, (v_k, p_k)\rangle$ of length $k = T - D$ such that for all $i \in [1\,..\,k]$, the preprocessing phase assigns element v_i to processor p_i.*

Proof. We shall construct the delay sequence by starting with v_k and working backwards, where $k = T - D$. For each element v_i in the delay sequence, we shall determine a time t_i, where $1 \le t_i \le T$, such that element v_i is executed at time t_i on processor p_i. We shall maintain the invariant that for $i \in [2\,..\,k]$, we have

$$t_i - t_{i-1} \le depth(v_i) - depth(v_{i-1}) + 1\,. \qquad (1)$$

Since the running time is at least T, let v_k be an element that is executed at time step $t_k = T$, and let p_k be the processor to which the preprocessing phase assigns v_k. For $i \in [2\,..\,k]$, consider the task of finding v_{i-1}, p_{i-1}, and t_{i-1} given that we have already chosen v_j, p_j, and t_j for all $j \in [i\,..\,k]$. There are two cases that may apply.

The first case is the situation when at time $t_i - 1$, an element v_j with $j \ge i$ satisfying $depth(v_j) = depth(v_i)$ is enqueued on p_i. In this case, let v_{i-1} be the enqueuer of v_j, and let p_{i-1} be the processor to which v_{i-1} is assigned. The enqueuer v_{i-1} exists, unless v_j was enqueued as one of the base set of updated elements at time $t_i - 1 = 0$. For $i > 1$, we can show that the enqueuer v_{i-1} does exist, because the invariant (1) holds for all $\ell \in [i+1\,..\,k]$, which implies that

$$\begin{aligned} t_i &= t_k - \sum_{\ell=i+1}^{k} (t_\ell - t_{\ell-1}) \\ &\ge t_k - \sum_{\ell=i+1}^{k} (depth(v_\ell) - depth(v_{\ell-1}) + 1) \\ &= t_k - (depth(v_k) - depth(v_i)) - (k-i) \\ &\ge T - D - k + i \\ &= i \\ &> 1\,, \end{aligned}$$

which implies that $t_i - 1 \ne 0$. Thus, the enqueuer v_{i-1} does exist. Condition A of the delay sequence definition holds, since the enqueuer of v_j is strictly shallower than v_j, and v_j is at the same depth as v_i. The enqueuer v_{i-1} executed at time $t_i - 1$, so we let $t_{i-1} = t_i - 1$. The invariant (1) holds, because $t_i - t_{i-1} = 1$, and $depth(v_i) > depth(v_{i-1})$. The edge (v_{i-1}, v_j) satisfies the requirement in Condition B. Finally, because v_{i-1} is strictly shallower than v_i, it must be distinct from all v_ℓ with $\ell \in [i\,..\,k]$, which shows that Condition C holds.

The second case is the situation when at time $t_i - 1$, no element v_j with $j \ge i$ satifying $depth(v_j) = depth(v_i)$ is enqueued on p_i. Let v_{i-1} be the previous element executed on processor p_i, and let $p_{i-1} = p_i$. Since v_i is not enqueued at time $t_i - 1$, it must be in the queue at time $t_i - 1$. Therefore, we must have $depth(v_{i-1}) \le depth(v_i)$ or else v_i would have been executed before v_{i-1}, which establishes Condition A. If we define $x = depth(v_i) - depth(v_{i-1})$, then x steps of moving a pointer through the priority queue intervened between the execution of v_{i-1} and v_i. Consequently, we have $t_{i-1} = t_i - (depth(v_i) - depth(v_{i-1})) - 1$, which implies that the invariant (1) holds. Condition B holds trivially, because $p_{i-1} = p_i$. We show that Condition C holds by contradiction. Suppose that $v_{i-1} = v_\ell$ for some $\ell \in [i\,..\,k]$. From Conditions A and B, it follows that all the elements $\langle v_{i-1}, v_i, \ldots, v_\ell\rangle$ have the same depth d and all are assigned to the same processor. Consquently, the second case must have applied when we chose $v_{\ell-1}$ through v_{i-1}, which means that no element of depth d was enqueued on the processor throughout this time. But after v_{i-1} is executed, it must be reenqueued to be executed later as v_ℓ, which is a contradiction. □

The next lemma counts the number of possible delay sequences.

Lemma 2 *Suppose that the optimistic algorithm run on a computer with P processors performs an update on a circuit $G = (V, E)$ with maximum degree Δ, and suppose that D and W are the embedded depth and size of the volatile subcircuit, respectively. Then, there are at most*

$$DW\,(6\Delta)^D \cdot \max_{X \in [1..D]} \left\{ P^X \left(\frac{3eW}{k-X}\right)^{k-X} \right\}$$

delay sequences of length k.

Proof. We can construct every possible delay sequence of length k satisfying Conditions A, B, and C by choosing the following items:

- The number X of **segments** through which the delay sequence passes. A segment is a maximal piece of the delay sequence local to a processor. (Alternatively, $X-1$ is the number of processor changes in the delay sequence). Because a delay sequence can only change processors between distinct depths, we have $X \le D$. Therefore, there are D choices for X.

- A processor for each segment. Since there are X segments, there are at most P^X choices.
- An interval of depths for each segment. Call the lengths of these intervals $s_1, s_2, \ldots, s_X$. There are $\binom{D-1}{X-1}$ possible choices for the s_i.
- The number of elements that belong to each segment. Call these counts $k_1, k_2, \ldots, k_X$, where $\sum_{r=1}^{X} k_r = k$. There are $\binom{k-1}{X-1}$ possible choices for the k_r.
- The particular elements in the delay sequence. For each segment we must first choose an immediate predecessor in the circuit of one of the elements of the previous segment, of which there are at most $k_{r-1}\Delta$, followed by $k_r - 1$ other elements. The only exception is that the first segment has no preceding segment. Hence, there are at most
$$\binom{W_1}{k_1} \prod_{r=2}^{X} k_{r-1}\Delta \binom{W_r - 1}{k_r - 1}$$
choices for the elements, where W_r is the number of elements in the volatile subcircuit that have depths from segment r. Note that the W_r depend on the s_r which were chosen previously. Only elements from the volatile circuit can be in the delay sequence because all other elements will never be enqueued.

An upper bound on the number DS_k of delay sequences of length k can be obtained by multiplying these values together:

$$\begin{aligned}
DS_k \;&\le\; D \max_X \left\{ P^X \binom{D-1}{X-1} \binom{k-1}{X-1} \right. \\
&\qquad \left. \max_{k_r, s_r} \left\{ \binom{W_1}{k_1} \prod_{r=2}^{X} k_{r-1}\Delta \binom{W_r - 1}{k_r - 1} \right\} \right\} && (2) \\
&\le\; D 2^D 2^k \Delta^D \max_X \left\{ P^X \right. \\
&\qquad \left. \max_{k_r, s_r} \left\{ \frac{W_1}{k_1} \left(\prod_{r=1}^{X} k_r \right) \prod_{r=1}^{X} \binom{W_r - 1}{k_r - 1} \right\} \right\} && (3) \\
&\le\; D (2\Delta)^D 2^k \max_X \left\{ P^X \max_{k_r, s_r} \left\{ W \left(\frac{k}{X} \right)^X \right. \right. \\
&\qquad \left. \left. \prod_{r=1, k_r > 1}^{X} \left(\frac{(W_r - 1)e}{k_r - 1} \right)^{k_r - 1} \right\} \right\} && (4) \\
&\le\; D (2\Delta)^D 2^k W e^{k/e} \\
&\qquad \max_X \left\{ P^X \left(\frac{(W - X)e}{k - X} \right)^{k - X} \right\} && (5) \\
&\le\; DW (2\Delta)^D 3^k \max_X \left\{ P^X \left(\frac{We}{k - X} \right)^{k - X} \right\} && (6) \\
&\le\; DW (6\Delta)^D \max_X \left\{ P^X \left(\frac{3eW}{k - X} \right)^{k - X} \right\} && (7)
\end{aligned}$$

In this reasoning, Inequality (3) follows from Inequality (2), because $\binom{n}{k} \le 2^n$ and $\binom{n}{k} = \left(\frac{n}{k}\right)\binom{n-1}{k-1}$. To obtain Inequality (4), we use the inequalities $\prod_{r=1}^{n} a_r \le \left(\frac{1}{n}\sum_{r=1}^{n} a_r\right)^n$ and $\binom{n}{k} \le \left(\frac{ne}{k}\right)^k$. To derive Inequality (5) from Inequality (4), we used the inequality $\left(\frac{a}{x}\right)^x \le e^{a/e}$ and the inequality

$$\prod_{r=1}^{n} \left(\frac{b_r}{a_r} \right)^{a_r} \le \left(\frac{\sum_{r=1}^{n} b_r}{\sum_{r=1}^{n} a_r} \right)^{\sum_{r=1}^{n} a_r} , \qquad (8)$$

which we prove in Appendix A. Finally, to obtain Inequality (6), we use $e^{1/e} \le 3/2$. $\square$

We now compute an upper bound on the probability distribution of the optimistic algorithm's running time.

Theorem 3 *Suppose that the optimistic algorithm run on a computer with P processors performs an update on a bounded-degree circuit $G = (V, E)$, and suppose that W and D are the size and embedded depth, respectively, of the volatile subcircuit. Then, for all $\epsilon > 0$, the probability is at most ϵ that the optimistic algorithm runs in more than $O(W/P + D + \lg W + \lg P + \lg(1/\epsilon))$ time.*

Proof. If the running time of the optimistic algorithm, ignoring termination detection, is at least T, then there must be a delay sequence of length $k = T - D$ from Lemma 1. The probability that a particular delay sequence of length k occurs in an execution is at most $(1/P)^k$, because for a delay sequence to occur in an execution, it is necessary that the processor numbers be correct for all the elements, and the probability of any one processor number being correct is $1/P$. Therefore, the probability of any delay sequence of length k occuring is at most the number of delay sequences of length k (calculated in Lemma 2) times the probability of a particular delay sequence occurring. Letting Δ be the maximum degree of any element in V, we can upper bound this probability as

$$\begin{aligned}
&\Pr\{\text{any delay sequence of length } k \text{ occurs}\} \\
&\quad\le\; DW (6\Delta)^D \max_X \left\{ P^X \left(\frac{3eW}{k - X} \right)^{k - X} \right\} \left(\frac{1}{P} \right)^k \\
&\quad\le\; DW (6\Delta)^D \max_X \left\{ \left(\frac{3eW}{P(k - X)} \right)^{k - X} \right\} .
\end{aligned}$$

Using the fact that $X \le D$, the probability that any delay sequence of length $k \ge 6eW/P + D$ occurs is at most

$$DW (6\Delta)^D \left(\frac{1}{2} \right)^{k - D} ,$$

which for $k = 6eW/P + D + \lg D + \lg W + D \lg 6\Delta + \lg(1/\epsilon)$ is at most ϵ. Consequently, with probability at least $1 - \epsilon$, the optimistic algorithm without termination detection runs in $k + D = O(W/P + D + \lg W + \lg(1/\epsilon))$ time. Adding $O(\lg P)$ as the additional latency for termination detection completes the proof. $\square$

Corollary 4 *The optimistic algorithm takes $O(W/P + D + \lg W + \lg P)$ expected time on a bounded-degree circuit, where P is the number of processors, and W and D are the size and depth, respectively, of the volatile subcircuit.*

Proof. Setting $t = \lg(1/\epsilon)$, the probability of a running time of $O(W/P + D + \lg W + \lg P + t)$ is at least $1 - 2^{-t}$. Consequently, the expected running time is $O(W/P + D + \lg W + \lg P + \sum_{t=0}^{\infty} t2^{-t}) = O(W/P + D + \lg W + \lg P + 2) = O(W/P + D + \lg W + \lg P)$. $\square$

6 Conclusions

Our asynchronous, optimistic, parallel algorithm is by far the most intriguing of the three algorithms for the circuit value update problem that we have presented. It is a simple and natural generalization of the serial algorithm, and we find it remarkable that it works as well as it does. To our knowledge, our analysis provides the first analytical bounds on the running time of an optimistic, asynchronous algorithm. We now briefly discuss some directions for extending our work.

The optimistic algorithm may actually be better than our current analysis shows. In our analysis, we have assumed that worst-case glitching causes all of the volatile subcircuit to be reevaluated. An element is most prone to glitch if its predecessors have widely varying depths. In this case, however, when the shallower predecessor enqueues the element, it is placed deep in its priority queue. Consequently, there is increased opportunity for the element's deeper predecessor to provide a value before the element is evaluated. It may be that this phenomenon can be exploited to provide a tighter analysis, especially when the work per processor of the altered subcircuit is large compared with its embedded depth.

The time bounds we have proved have the form $O(W/P + D)$ with some additional logarithmic terms, where W and D are the size and embedded depth, respectively, of either the altered or volatile subcircuit. These bounds are reminscent of Brent's provably good bounds [1] for off-line scheduling of dags, except that in Brent's work, D is the true depth of the subcircuit. In our analysis, we rely heavily on the embedded depths of circuit elements as priority queue keys, however. Thus, it seems unlikely that we can improve our results to use true depth without a major restructuring of the algorithms.

As we have defined it, the circuit value update problem addresses circuits whose elements can be evaluated in constant time. It is natural to try to generalize the problem to arbitrary propagation times. It is unlikely that our optimistic algorithm will generalize if we are not allowed to preempt the evaluation of long-running circuit elements, however, because they can block the execution of higher-priority elements that are enqueued later. We are currently studying the problem when preemption is allowed.

We are optimistic that our proof techniques can be applied to obtain bounds on other optimistic algorithms, such as simulation by time-warping [7, 11, 3]. For example, we are studying the problem of extending our results, which currently are restricted to combinational circuits, to optimistic simulation of sequential circuits.

7 Acknowledgments

Thanks to Chris Joerg of MIT for acquainting us with related work and pointing out a bug in our time bounds. Thanks to Bobby Blumofe and Matteo Frigo of MIT for their helpful comments.

References

[1] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[2] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[3] K. Choi, S. Y. Hwang, and T. Blank. Incremental-in-time algorithm for digital simulation. In *Proc. 25th Design Automation Conference*, pages 501–505, June 1988.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, MIT Press, 1990.

[5] Rajiv Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, April 1989.

[6] Rajiv Gupta and Charles R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, June 89.

[7] David R. Jefferson. Virtual time. *ACM Transactions of Programming Languages and Systems*, 7(3):404–425, July 1985.

[8] Donald B. Johnson. A priority queue in which initialization and queue operations take O($\log\log D$) time. *Mathematical Systems Theory*, 15(4):295–309, December 1982.

[9] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*. Morgan Kaufman, 1992.

[10] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, San Diego, California, June 1992.

[11] Yukinori Matsumoto and Kazuo Taki. Parallel logic simulation on a distributed memory machine. In *Proceedings of the European Conference on Design Automation*, pages 76–80, 1992.

[12] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, June 1991.

[13] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 1992.

A Proof of Inequality (8)

Lemma 5 *Let a_i and b_i, $i = 1, \ldots, k$, be partitions of A and B, i.e. $\sum_{i=1}^{k} a_i = A$ and $\sum_{i=1}^{k} b_i = B$. Furthermore, let $a_i, b_i > 0$. Then*

$$\prod_{i=1}^{k} \left(\frac{b_i}{a_i}\right)^{a_i} \quad \leq \quad \left(\frac{B}{A}\right)^{A}$$

Proof. Let $N = \prod_{i=1}^{k} (b_i/a_i)^{a_i}$. Consider N as a function of b_1 through b_{k-1} (i.e. substitute $b_k = B - \sum_{i=1}^{k-1} b_i$ into N, and keep the a_i fixed). We shall show that there is a unique extremum of $N(b_1, \ldots, b_{k-1})$, that this extremum is a maximum, and that $N = (B/A)^A$ at this maximum. These facts together imply that N is at most $(B/A)^A$ everywhere.

Any extremum of N must satisfy the conditions $\partial N/\partial b_j = 0$ for $j = 1, \ldots, k-1$. These conditions can be computed as follows:

$$
\begin{aligned}
0 &= \frac{\partial N}{\partial b_j} \\
&= \frac{\partial}{\partial b_j}\left[\prod_{i=1}^{k-1}\left(\frac{b_i}{a_i}\right)^{a_i}\left(\frac{B-\sum_{i=1}^{k-1} b_i}{a_k}\right)^{a_k}\right] \\
&= \prod_{\substack{i=1\\ i\neq j}}^{k-1}\left(\frac{b_i}{a_i}\right)^{a_i}\frac{\partial}{\partial b_j}\left[\left(\frac{b_j}{a_j}\right)^{a_j}\left(\frac{B-\sum_{i=1}^{k-1} b_i}{a_k}\right)^{a_k}\right] \\
&= \prod_{\substack{i=1\\ i\neq j}}^{k-1}\left(\frac{b_i}{a_i}\right)^{a_i}\left[a_j\left(\frac{b_j}{a_j}\right)^{a_j-1}\frac{1}{a_j}\left(\frac{B-\sum_{i=1}^{k-1} b_i}{a_k}\right)^{a_k}\right. \\
&\qquad \left. + \left(\frac{b_j}{a_j}\right)^{a_j} a_k\left(\frac{B-\sum_{i=1}^{k-1} b_i}{a_k}\right)^{a_k-1}\frac{-1}{a_k}\right] \\
&= \prod_{\substack{i=1\\ i\neq j}}^{k-1}\left(\frac{b_i}{a_i}\right)^{a_i}\left(\frac{b_j}{a_j}\right)^{a_j}\left(\frac{B-\sum_{i=1}^{k-1} b_i}{a_k}\right)^{a_k} \\
&\qquad \left[\frac{a_j}{b_j}-\frac{a_k}{B-\sum_{i=1}^{k-1} b_i}\right] \\
&= N\left[\frac{a_j}{b_j}-\frac{a_k}{B-\sum_{i=1}^{k-1} b_i}\right] .
\end{aligned}
$$

These conditions imply that at any extremum, we have

$$
b_j = a_j\left(\frac{B-\sum_{i=1}^{k-1} b_i}{a_k}\right) \tag{9}
$$

for $j = 1, \ldots, k-1$. Solving for the b_j in terms of b_1 yields

$$
b_j = a_j\left(\frac{b_1}{a_1}\right)
$$

for $j = 1, \ldots, k-1$. Summing these equations over j, we obtain

$$
\begin{aligned}
\sum_{j=1}^{k-1} b_j &= \sum_{j=1}^{k-1} a_j\left(\frac{b_1}{a_1}\right) \\
&= (A-a_k)\left(\frac{b_1}{a_1}\right) .
\end{aligned}
$$

Substituting this equation back into Equation (9) with $j = 1$ yields

$$
\begin{aligned}
b_1 &= a_1\left(\frac{B-\frac{b_1}{a_1}(A-a_k)}{a_k}\right) , \\
b_1 a_k &= a_1 B - b_1 A + b_1 a_k , \\
\frac{b_1}{a_1} &= \frac{B}{A} ,
\end{aligned}
$$

and for $j = 1, \ldots, k-1$,

$$
\frac{b_j}{a_j} = \frac{B}{A} .
$$

These equations show that the extremum is unique and give the values of the b_j at that extremum. To show the extremum is a maximum, we perform the second derivitive test:

$$
\begin{aligned}
\frac{\partial^2 N}{\partial {b_j}^2} &= \frac{\partial}{\partial b_j}\left[N\left(\frac{a_j}{b_j}-\frac{a_k}{B-\sum_{i=1}^{k-1} b_i}\right)\right] \\
&= \frac{\partial N}{\partial b_j}\left(\frac{a_j}{b_j}-\frac{a_k}{B-\sum_{i=1}^{k-1} b_i}\right) \\
&\quad + N\left(-\frac{a_j}{{b_j}^2}+\frac{a_k}{\left(B-\sum_{i=1}^{k-1} b_i\right)^2}(-1)\right) .
\end{aligned}
$$

Substituting in Equation (9) we obtain

$$
\begin{aligned}
\frac{\partial^2 N}{\partial {b_j}^2} &= \frac{\partial N}{\partial b_j}0 + N\left(-a_j\left(\frac{a_k}{a_j\left(B-\sum_{i=1}^{k-1} b_i\right)}\right)^2\right. \\
&\qquad \left. - \frac{a_k}{\left(B-\sum_{i=1}^{k-1} b_i\right)^2}\right) \\
&= \frac{-N}{\left(B-\sum_{i=1}^{k-1} b_i\right)^2}\left(\frac{{a_k}^2}{a_j}+a_k\right) \\
&< 0 .
\end{aligned}
$$

Since the second derivative is negative for every b_j, the extremum is a maximum. Substituting $b_j/a_j = B/A$ into the definition of N yields

$$
\begin{aligned}
N &\leq \prod_{i=1}^{k-1}\left(\frac{B}{A}\right)^{a_i}\left(\frac{B-\sum_{i=1}^{k-1} a_i\left(\frac{B}{A}\right)}{a_k}\right)^{a_k} \\
&= \left(\frac{B}{A}\right)^{A-a_k}\left(\frac{B}{A}\right)^{a_k}\left(\frac{A-(A-a_k)}{a_k}\right)^{a_k} \\
&= \left(\frac{B}{A}\right)^{A} .
\end{aligned}
$$

□