Analyzing Traces of Parallel Programs Containing Semaphore Synchronization

D. P. Helmbold, C. E. McDowell, T. Haining

September 28, 2003

Abstract

One important kind of correctness and performance debugging tool for parallel programs determines and presents temporal ordering relationships between the various synchronization operations in the program. The particular ordering relationship we study is the "always happens before" relationship for arbitrary programs using semaphore synchronization. Our analysis is based on an execution trace of the program rather than the program itself. We have previously published a polynomial time conservative approximation of the "always happens before" relationship. Determining the exact "always happens before" relationship is intractable (co-NP Hard).

We built a random program generator and applied our algorithm to the random programs it generated. Our algorithm's results were compared with the partial orders produced by an exponential time brute force algorithm which appears practical only for the relatively small programs generated. This process quickly identified traces where our algorithm failed to find some of the "always happens before" orderings. The findings from these experiments and the resulting modifications to our algorithm are described in this paper.

1 Introduction

Parallel programs with explicit synchronization can be notoriously difficult to debug. One important kind of correctness and performance debugging tool determines and presents the temporal ordering relationships between the various synchronization operations in the program. The particular ordering relationship we study is the "always happens before" relationship. Informally, event e_1 "always happens before" event e_2 if e_1 happens before e_2 in every execution in which e_2 occurs.

The complexity of determining the ordering relationships varies depending upon the type of synchronization and the branching characteristics of the program. There are polynomial time algorithms for simple models such as message passing with two-way naming or post and wait events with no clear in straight line programs (with no branches except bounded loops) [NG92]. On the other hand, undecidability issues arise when arbitrary programs are considered. We have been studying this problem for arbitrary programs using semaphore synchronization. We avoid the undecidability problems by basing our analysis on a trace

of the program rather than the program itself. Even with this restriction the problem is intractable (co-NP Hard), so we settle for a polynomial time conservative approximation of the "always happens before" relationship.

One possible application of the "always happens before" relationship is the detection of data races. If all accesses to a shared variable are ordered by the "always happens before" relationship then there is no race on the variable. Our conservative approximation of the "always happens before" relationship leads to the detection of more races than can occur, but guarantees the detection of every race that did occur in the execution of the program that was analyzed.

Care must be taken when generalizing from a trace to the entire program. One important situation where our trace results can be generalized is when we detect that there are no data races in the trace. This means that there are also no data races in the program (when executed with the same input).

In [HMWar] we described our basic approach. That algorithm takes an execution trace and produces a partial order representing the temporal orderings that always hold for the given program on the given input with two limitations:

1. It may fail to indicate that two events are ordered when they must always occur in a particular order (i.e. it is a conservative under-approximation of the "always happens before" relationship).

2. It may indicate that event e_1 "always happens before" event e_2 when there is a radically different execution where e_2 happens before e_1. When this happens the radical difference between the executions is caused (directly or indirectly) by some other race in the program that will be detected by our algorithms. (I.e. if we report there are no races then there are none, but we may not report all of the races.)

Other researchers are pursuing this problem from the other end (e.g. [NM91]). Given a set of potential races (as might be reported by our analysis) their techniques identify a subset of the races that that can actually occur. We believe these to be complementary approaches and are continuing to refine our algorithms to reduce the above limitations.

In order to refine our algorithm it was necessary to determine when it failed (i.e. find a program and an execution trace containing two events e_1 and e_2 such that e_1 happens before e_2 in every execution of the program in which they both occur and our algorithm indicates that e_1 and e_2 are unordered). Given the limited set of "real" programs available to us in the programming language we currently support (IBM Parallel Fortran) our algorithm never fails (i.e. it finds exactly those orderings that hold for all executions given a particular input). We therefore built a random program generator and used our algorithm on the random programs it generated. Our algorithm's results were compared with the partial orders produced by an exponential time "brute force" algorithm which appears practical only for the relatively small programs generated. This process quickly identified traces where our algorithm failed to find some of the "always happens before" orderings. These experimental results have lead to considerable modifications to our algorithms.

2 Overview of the Algorithm

Our algorithms use vector timestamps [Fid88] for each event to represent the partial order. We call a partial order *safe* if it contains a proper subset of the edges in the "always happens before" ordering. We first compute a very conservative safe partial order from an execution trace and then attempt to insert new edges into this safe partial order while maintaining the safeness property. Edges are inserted into the partial order by manipulating the timestamps assigned to the events.

We only consider synchronization using counting semaphores with the two semaphore operations *signal* and *wait*[1]. To compute the initial safe partial order we assume that any signal event could have been the signal that releases any wait. Our previous algorithm for inserting additional edges was based upon the following observations:

Observation 1 *If some wait event e_w on semaphore A is known to follow n other waits on semaphore A (given the safe partial order already computed) then we know that e_w must follow $n+1$ signal events on semaphore A. Thus additional edges can be inserted into the partial order by increasing the (vector) timestamp for e_w so that each component is at least as big as the corresponding components in $n+1$ of the timestamps for the signal events on semaphore A.*

Observation 2 *If one of the $n+1$ signals, call it e_s, needed in observation 1 is known to be preceded by an additional wait event on semaphore A that is not one of the n wait events known to precede event e_w, then $n+2$ signals occur before e_w whenever e_s occurs before e_w.*

This second observation implies that $n+1$ signals other than e_s occur before e_w. In the program of Figure 1, the S_1 event in task D corresponds to the e_s event in Observation 2. We call this phenomenon *shadowing*, as the "shadow" cast by the preceding wait prevents e_s from satisfying the signals needed by e_w.

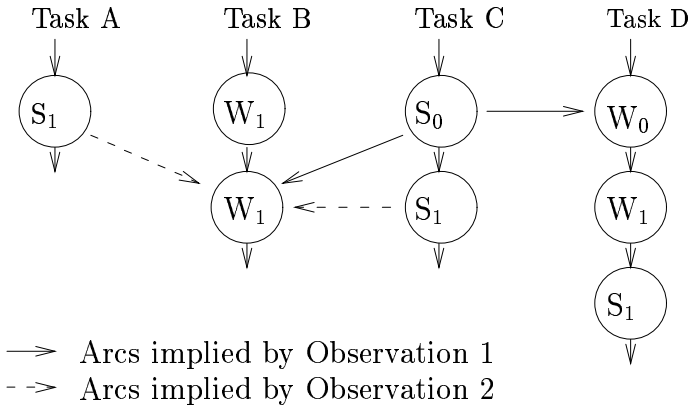

Figure 1: An Example of Observations 1 and 2. Note that the program can deadlock with Task D waiting on semaphore 1.

[1] These are more descriptive for English speakers than the original V and P of Dikjstra.

Our study of random programs has motivated changes in the algorithms so that they exploit the following additional observations:

Observation 3 *If wait event e_w on semaphore A is known to follow n waits on some* other *semaphore B (given the safe partial order already computed) then we know that e_w must follow n signal events on semaphore B.*

This extends observation 1 to apply to semaphores other than the one specified in the event e_w. The example that lead to this observation is shown in Figure 2.

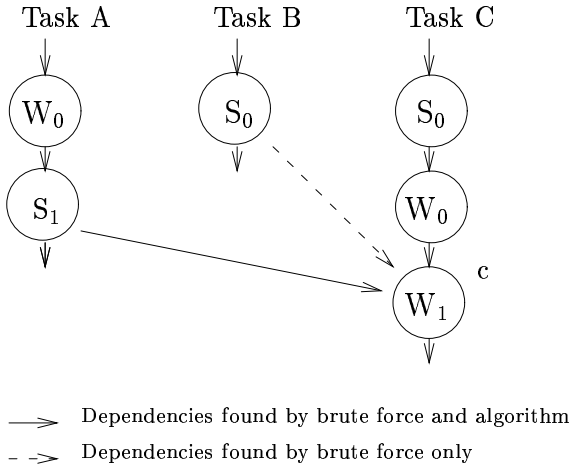

Figure 2: Example motivating observation 3. Event c is preceded by two wait on semaphore 0 events (W_0's) and therefore must be preceded by two S_0's.

Observations 1 and 3 lead to what could be called first order inferences. A particular event must be preceded by a certain number of wait events on various semaphores and therefore the event in question must be preceded by as many signals as there are waits preceding it. Shadowing is a second order inference, i.e. if signal e_s happens before wait e_w then we also need additional signals for the waits that precede e_s (and have not already been accounted for because they also precede e_w). Our next observation is a second-order extension of Observation 3, somewhat like shadowing is a second-order extension of Observation 1.

Observation 4 *If some signal event e_s is going to help satisfy the signals needed for wait event e_w, then e_w will also have to follow any signals needed by wait events that precede e_s.*

This can be seen best by the example shown in Figure 3.

This chaining effect (the execution needs n signals but one of those signals needs m more signals) can be repeated arbitrarily[2] and Figure 4 gives an example that takes this chaining effect one step further. Our goal was to capture this chaining in a recursive algorithm that can be allowed to recurse as deep as time will allow. The algorithm becomes polynomial (and conservative) when the recursion is limited to a fixed bound. The complete algorithm is given in the appendix.

[2]We believe this chaining may occur infrequently if at all in real programs.

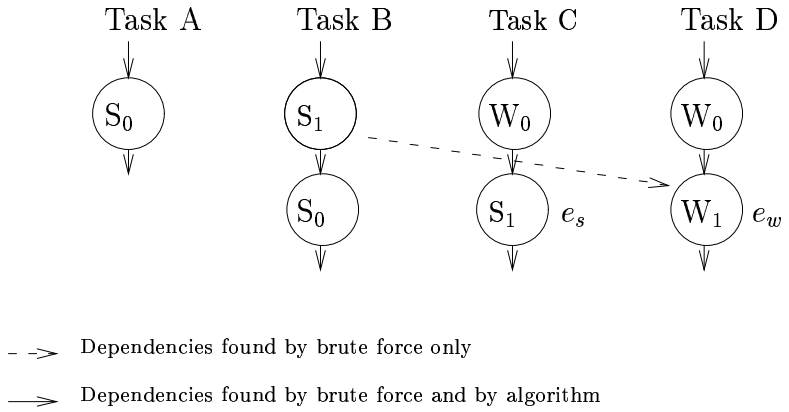

Figure 3: Example motivating observation 4. Events labeled e_s and e_w correspond to those in the observation.

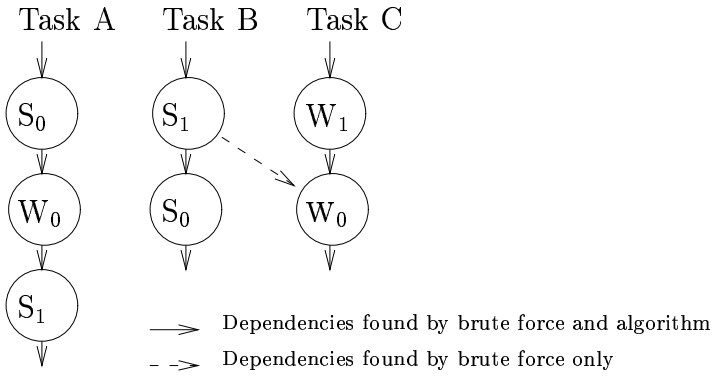

Figure 4: Example requiring two applications of observation 4.

3 Testing Random Traces

As mentioned before, we have a limited number of real programs on which to test our algorithms. Thus we have built a simple random program generator that creates input traces. In order to determine the effectiveness of our algorithms we have also built a simple brute force trace analyzer that exactly determines the "always happens before" relationship.

A trace of an execution is simply a totally ordered list of signal and wait events together with an indication of which task performed each event. As events are entered in the trace only when the associated operation completes, every prefix of the trace contains at least as many signal events on each semaphore as wait events.

The "program" corresponding to the trace can be viewed as in Figures 1, 2, 3, and 4. There the events are grouped in vertical lists by task. Within each vertical list the events appear in the same order as they occurred in the trace. A "state" of the program corresponds to selecting a (possibly empty) prefix of each vertical list. Usually some program states are unreachable by any execution. For example, in Figure 4 the state where Tasks A and B have executed no events but Task C has executed its first event (W_1) is unreachable as the wait cannot complete until after semaphore 1 has been signaled.

The brute force analyzer performs a depth-first search on the program's (exponentially large) state space to discover all of the reachable states. If event e_1 has been executed in every reachable state where event e_2 has been executed then the brute force analyzer indicates that e_1 "must happen before" e_2. Performing this test for each pair of events allows the brute force analyzer to compute the "must happen before" relationship for the "program".

If a real program makes conditional branches that could be affected by races, then the the brute force analyzer may overestimate the "must happen before" relationship. Our algorithms also have this drawback as noted in the introduction. However, even the overestimated "must happen before" relationship allows the race affecting the conditional branch to be flagged. Our experiments concentrate on how often the "must happen before" relationship computed by our other algorithms matches that returned by the brute force analyzer.

The random trace generator uses the parameters MaxTasks, MaxSemaphores, and NumEvents. For each trace pick NumTasks (between 2 and MaxTasks) and NumSemaphores (between 1 and MaxSemaphores) uniformly at random. Once these values have been set, the generator executes the loop in Figure 5 to output the trace of a non-blocking execution.

At the time of this writing we have done only a few tests aimed at determining the accuracy of the algorithm, but the numbers are very encouraging. The results are given in Table 1.

4 Conclusion

Our approach of using randomly generated programs to improve our algorithms has been extremely successful. When applied to small randomly generated straight line parallel programs, our new algorithm failed to find all "must happen before" edges less than 0.25% of the time. We believe that for real programs the failure rate will be significantly lower.

```
while less than NumEvents generated do
      select T randomly between 1 and NumTasks
      select sem randomly between 1 and NumSemaphores
      if more signals than waits have been generated for sem
            then
                  flip a coin
                  if heads then output: W_sem by task T
                  if tails then output: S_sem by task T
            else
                  output: S_sem by task T
end while
```

Figure 5: Random program generator.

# of events	original	depth 1	depth 2	depth 3	total # of traces
30	54	3	0.26	0.05	5726
35	58	3.7	1.4	1.1	840
40	59	3.4	0.75	0.56	1070

Table 1: Percentage of random traces that failed to find at least one edge when compared to the actual "must happen before" partial order. Results are given for the original algorithm and for our revised algorithm with the depth of recursion set as indicated.

References

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988.

[HMWar] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, to appear. Also UCSC Tech. Rep. UCSC-CRL-91-36.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms*, 1988.

[NG92] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with Post/Wait synchronization. In *Proc. International Conf. on Parallel Processing*, 1992.

[NM91] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices (Proc. PPOPP)*, 26(7):133–144, 1991.

A Algorithm Details

Before each trace is analyzed, every event contained in that trace is assigned a vector timestamp. A timestamp contains one entry for each task. For a timestamp τ, $\tau[i]$ is the number of events completed by task T_i at the time the event associated with τ completed. When properly maintained, ordered and unordered event pairs can can easily be distinguished by comparing their time vectors.

An event e with timestamp $\tau(e)$ precedes another event e' with timestamp $\tau(e')$ in the partial order if and only if every component of $\tau(e)$ is less than or equal to the corresponding component of $\tau(e')$. Events e and e' are unrelated in the partial order when both some component of $\tau(e)$ is greater than the corresponding component of $\tau(e')$, and some (other) component of $\tau(e')$ is greater than the corresponding component in $\tau(e)$.

Definition 1 *For any two time vectors τ_1, τ_2 in Z^n*

1. $\tau_1 \leq \tau_2 \iff \forall i(\tau_1[i] \leq \tau_2[i])$
2. $\tau_1 < \tau_2 \iff \tau_1 \leq \tau_2$ *and* $\tau_1 \neq \tau_2$
3. $\tau_1 \not\geq \tau_2 \iff \exists i(\tau_1[i] < \tau_2[i])$

The time vector τ_1 <u>is earlier than</u> *time vector τ_2 (or τ_2* is later than *τ_1) when $\tau_1 < \tau_2$.*

At present, we use three algorithms to build the partial orders which approximate the ordering properties of events in a program trace. The first extracts the corresponding partial order from the event trace. The second modifies this partial order to ensure that it is valid for all possible executions in which the same events occur. The third uses the observations

presented earlier in this paper to add back some ordering arcs which still are present in all executions.

Before defining our algorithms it is necessary to define a few common functions that we employ:

Definition 2 *For any m time vectors $\tau_1, \ldots, \tau_m$ of Z^n*

- $\overline{\min}_k(\tau_1, \ldots, \tau_m), k > 0$ *is the vector of Z^n whose ith component is the k^{th} smallest element in the collection $\tau_1[i], \tau_2[i], \ldots, \tau_m[i]$,*
- $\overline{\max}(\tau_1, \ldots, \tau_m)$ *is the vector in Z^n whose ith component is* $\max(\tau_1[i], \ldots, \tau_m[i])$.

Conventionally, we define $\overline{\min}_0(\tau_1, \ldots, \tau_m)$ to be $\bar{0}$, the all-zero vector.

As an example, $\overline{\min}_3([1,2],[1,3],[2,4],[2,5],[3,2])$ is $[2,3]$.

We also employ two special types of timestamps:

Definition 3 *Given an event e performed by task T_i in a trace, let $\tau^{\#}(e)$ be the time vector containing the local event count for e (one more than the number of events previously performed by T_i in the trace) in the ith component and zeros elsewhere.*

Definition 4 *Given an event e performed by task T_i in a trace, let e^p denote the previous event performed by T_i in that trace (or the all zero vector if no such an event exists).*

To obtain an initial partial order for an execution trace, on which further analysis can be based, we use an algorithm based on the one provided in Fidge [Fid88] and Mattern [Mat88]. This process associates a Wait event with an unused Signal event on the same semaphore, creating a partial order which pairs every Wait event with a Signal event which allowed it to precede. This generates a partial order that contains orderings that are not "must happen before" orderings. Instead, this partial order represents the causal orderings that did occur in a particular execution.

To generalize this partial order information to make it valid for all possible executions containing the same events, we use a process called *rewinding* (Algorithm 1) to decouple Wait events from specific Signal events. After rewinding, every Wait event has a time vector that reflects the assumption that any Signal (on the same semaphore) could have been the Signal that triggered the Wait.

Algorithm 1 (Rewind)

Repeat the following procedure until no further changes are possible.

for each event e in the trace
    if e is a Wait event on semaphore S,
        let $e_1^s \ldots e_k^s$ be all the Signal events on S;
        set $v_s = \overline{\min}(\tau(e_1^s), \ldots, \tau(e_k^s))$;
    else
        set $v_s = \bar{0}$, the all zero vector;
    end if;
    set $\tau(e) = \overline{\max}(\tau(e^p), \tau^{\#}(e), v_s)$;
end for;

Unfortunately, the newly established safe order relation is too conservative because some "must happen before" ordering arcs are deleted as a part of the rewind procedure. At this point we apply an algorithm based on the observations described in Section 2.

The Expand algorithm (Algorithm 2) cycles through the entire timestamp representation of the trace, using a routine called $modify$ to advance the timestamps of Wait events based on Observation 4.

It does this by considering the set of Signal events, R, and the set of Wait events, W, that exist in relation to two timestamps: τ_{in} and $\tau(\hat{e})$. Timestamp τ_{in} is an amalgamated timestamp representing all the events previously considered through recursive calls to $modify$. It will act as the event e_w in the current application of Observation 4. Event $\hat{e}$ functions as the Signal e_s from Observation 4. The quantity $depth$ indicates the number of times that Observation 4 is to be recursively applied.

Function $modify$ builds up a set, T, of virtual timestamps returned by calling $modify$ recursively on the events in R. It then employs Observation 3, taking the k^{th} component-wise minimum of T (where k is the number of Wait events in W) to determine the earliest time that $\hat{e}$ can occur. It then returns $\hat{e}$'s new timestamp in the quantity τ_m.

Finally, the main loop takes the timestamp returned by $modify$ for a specific Wait event, and uses $\overline{\max}$ to combine it with $\tau(e^p)$ and $\tau^{\#}(e)$.

Algorithm 2 (Recursive Expand):

Repeat the following procedure until no more changes are possible.

for each event e in the trace
    if e is a Wait event on semaphore S,
        $\overline{\tau}(e) = modify(\tau(e), \tau(e), e, depth)$;
        set $\tau(e) = \overline{\max}(\tau(e), \tau(e^p), \overline{\tau}(e))$;
    else
        set $\tau(e) = \overline{\max}(\tau(e), \tau(e^p))$;
    end if;
end for;

Function $modify(\tau_{base}, \tau_{in}, \hat{e}, depth)$:
    let $\tau_m = \tau(\hat{e})$
    if $depth = 0$
        return(τ_m);
    end if;
    for each semaphore σ,
        let $T = \emptyset$;
        let $W(\sigma) = \{e_w : e_w$ is a Wait event on σ, and
                either $\tau(e_w) \leq \tau_{in}$ or $\tau(e_w) \leq \tau(\hat{e})\}$;
        let $R(\sigma) = \{e_s : e_s$ is a Signal event on σ,
                $\tau(e_s) \not\geq \tau_{in}$ and $\tau(e_s) < \tau_{base}\}$;
        for each e_s in R,
            $\tau = \overline{\max}(\tau_{in}, \tau(\hat{e}))$;
            $T = \{modify(\tau_{base}, \tau, e_s, (depth - 1))\} \bigcup T$;
        end for;
        let $k = \mid W(\sigma) \mid$;
        let $\tau_s = \overline{\min}_k(T)$;
        let $\tau_m = \overline{\max}(\tau_m, \tau_s)$;
    end for;
    return(τ_m);