Analysis of Event Synchronization in Parallel Programs

J. Ramanujam and **A. Mathew**
Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803
(jxr@gate.ee.lsu.edu)

Abstract. The increase in the number and complexity of parallel programs has led to a need for better approaches for synchronization error detection and debugging of parallel programs. This paper presents an efficient and precise algorithm for the detection of nondeterminacy (race conditions) in parallel programs. Nondeterminacy exists in a program when the program yields different outputs for different runs with the same input. We limit our attention to nondeterminacy due to errors in synchronization and to race conditions due to these unsynchronized accesses to shared variables. A directed acyclic graph called a task graph is used to represent the accesses to shared variables in a parallel program with edges representing guaranteed ordering. The algorithm proposed here constructs an augmented task graph, and then uses a modification of depth-first search to classify the edges in the augmented task graph. The edges are analyzed and the nodes that are guaranteed to execute before an event are linked to these events by edges representing guaranteed ordering among events. This ordering is used to detect any race conditions in parallel programs.

1 Introduction

The difficulty of writing and debugging parallel programs is becoming apparent with the increase in the number and the complexity of parallel programs. Debugging parallel programs is considerably more difficult than debugging sequential programs, because of potential nondeterminacy. Nondeterminacy is the possibility that a parallel program yields different outputs for different runs with the same input data. Such nondeterminacy may be intentional; in this paper, we limit our attention to nondeterminacy caused by synchronization errors. Race conditions arise when the synchronization operations do not guarantee ordering among two or more accesses to the same memory location, when at least one of these accesses is a write.

Debugging of parallel programs requires specialized tools and algorithms to detect race conditions. In recent years, work in this area has led to better definition of various forms of race conditions and improved algorithms, but much remains to be done. The objective of this paper is to present an algorithm for race detection in parallel programs. The algorithm uses the task graph of the parallel program as input; it constructs an augmented task graph by adding a directed edge from every `post` on an event to every `wait` on the same event. It should be noted that while the task graph is acyclic, the augmented task graph may contain cycles. The algorithm uses a modification of depth-first search to classify the edges of the augmented task graph, using a heuristic to decide the order in which nodes are visited. Based on the classification of edges and further analysis, the guaranteed ordering among the nodes is obtained. This information is used to identify possible race conditions and nondeterminacy in the program.

This paper is organized as follows. In Section 2, we present the necessary background for this paper, including the program model and constructs for which the techniques presented in this paper apply; in addition, we review related work. Section 3 describes the improved algorithm that is used in trace analysis of parallel programs using event

[1] Supported in part by an NSF Young Investigator Award CCR–9457768, an NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

synchronization without `clears`. Section 4 illustrates the use of the algorithm on sample program segments. Section 5 concludes with a summary and a discussion of approaches to extending the work to parallel programs that use `clears`.

2 Background and Related Work

We consider parallel programs consisting of tasks running on the processors of shared address space machines. Sequential programs due to their sequential semantics, guarantee ordering between accesses to shared memory. In parallel programs, this ordering can be enforced by either the sequential semantics of events within a task, or by explicit synchronization between events in different tasks that may execute concurrently. Incorrect synchronization leads to incorrect ordering between events and accesses to shared memory.

A *determinate* program produces the same output for different runs with the same input. Sequential programs are determinate. A *nondeterminate* program is one that produces different output for different runs with the same input and code. Parallel programs usually exhibit nondeterminacy due to errors in synchronization, but the nondeterminacy may be intentional as in the case of asynchronous parallel algorithms.

Race conditions are time-dependent errors that occur due to lack of ordering of accesses (at least one of which is a write access) to shared data by different processes. Bernstein [3] presents the requirements that must be true for a program to be determinate. A *conclusive* race is a race that is not dependent on the input data, whereas an *input-dependent* race is one that is dependent on the input data, and which may manifest itself for certain input data but not for others. Inaccurate race detection algorithms may report races that are not genuine and these are called *spurious* races.

2.1 Language Model

We assume a language with parallel constructs and synchronization primitives similar to those in CEDAR Fortran [1] and other dialects of parallel Fortran that support programming on a shared address space. The following parallel constructs and synchronization primitives are considered.

An *event* is a synchronization variable that can have only two states, posted or cleared. A `post(x)` on an event `x` sets the state of the event `x` to posted. A `post` does not change the state of an event that has already been posted; the event continues to remain posted. A `wait(x)` on an event `x` whose state is not posted, suspends the task that called it till the event is posted. A `clear(x)` on an event `x` resets the state of the variable `x` to cleared. The event may be reused after a clear on the event; the next `post(x)` sets the state of `x` to posted.

2.2 Race Detection

Most approaches to detection of races in parallel programs are a combination of static analysis and dynamic methods, since neither of the two provide sufficient information when used alone. Static analysis is faster, but may report nondeterminacy where one may not exist. Trace analysis is usually more accurate than static analysis. Static analysis is valid for a wider range of data as compared to trace analysis that is just valid for the input data used in a particular execution. Practical approaches to race detection therefore usually use static analysis to determine ranges of input data which might exhibit races (and therefore identify potential races) and use trace analysis to examine these cases one by one and pinpoint the sources of nondeterminacy.

2.3 Related Work

Model for Reasoning About and Characterizing Race Conditions: A formal model that characterizes the actual, observed and potential behaviors of programs was first proposed by Netzer [17, 20]. This model provides a conceptual framework on which the theoretical and practical aspects of dynamic race detection and debugging for shared-memory parallel programs can be based.

Netzer defines actual behavior of parallel programs as that which precisely represents program execution, observed behavior as the partial information that can be recorded and potential behavior as the alternate executions possibly allowed by the nondeterminism in the program. These behaviors of a parallel program are used to characterize the races into general and data races. General races generally apply to programs intended to be deterministic. Data races apply to nondeterministic parallel programs that contain critical sections.

Static and Dynamic Analysis: Static analysis methods use the program listing to detect races and assume that all the execution paths through the program are possible. Explicit synchronization in the program is used to determine the potential ordering in the program. Two basic approaches to static analysis are generally used. In the first approach, all possible states that the program can exist in are traversed and a graph of the state space is constructed either explicitly [15] or implicitly. In the second approach, dataflow analysis is used to detect potential event orderings [2, 5]. In general, dataflow methods are less accurate, but have better time and space complexity than state-space methods.

Static analysis is used to augment dynamic analysis [1, 6, 7, 8, 9, 14, 22]. Dynamic analysis detects races for a particular execution of the program; these methods generally differ in the way the information that is used in the analysis is collected and analyzed. The two approaches used are: post-mortem [1, 6, 7, 8, 9, 12, 13, 18] and on-the-fly [14, 18, 22] analysis. Allen and Padua [1] describe a method for the post-mortem analysis of Fortran programs in a shared-memory parallel machine. They use static analysis to reduce the trace information collected. Choi, Miller and Netzer [6] describe the techniques used in the debugging parallel programs using flowback analysis. For C programs using counting semaphores and the fork/join construct, Miller and Choi [16] present a parallel debugger using post-mortem data race analysis. These methods typically detect data races and not general races. Helmbold, McDowell and Wang [12, 13] present models and algorithms to find possible event orderings in traces of programs that use anonymous synchronization.

On-the-fly analysis techniques for detecting data races in programs using fork/join and arbitrary synchronization have been developed by Schoenberg [22]. A technique for detecting data races in Fortran programs using parallel do loops and synchronization using `send` and `wait` primitives, is described by Hood, Kennedy and Mellor-Crummey [14].

Algorithms That Compute Guaranteed Ordering: Emrath, Ghosh and Padua [7, 8, 9] present methods for detecting general races in programs using event synchronization. They present two algorithms for post-mortem analysis, namely, the Exhaustive Pairing Algorithm (EPA) and the Common Ancestor Algorithm (CAA) to compute guaranteed orderings, which are described next.

Common Ancestor Algorithm (CAA): Emrath, Ghosh and Padua [7, 8] have proposed an algorithm that computes the guaranteed ordering for programs containing the synchronization operations `post`, `wait` and `clear`. The algorithm is based on the property that a node that precedes all the posts that might trigger a wait necessarily precedes the wait node. Arcs are added not from all the ancestors of the posts that may trigger the wait, but from the closest common ancestor of those nodes. The CAA works in polynomial time but has the disadvantage that it may miss out on some of the guaranteed ordering even for program that do not use `clears`.

Exhaustive Pairing Algorithm (EPA): Emrath, Ghosh and Padua [8] propose an algorithm that computes every possible execution ordering due to the synchronization operations `post`, `wait` and `clear`. The algorithm produces an execution graph for each of the possible pairings, and detects the feasible execution graphs; it computes the transitive closure of the execution graphs to determine the feasibility of the execution graph. The intersection of the transitive closures of all feasible execution graphs is computed to yield the result. The exhaustive-pairing algorithm works in exponential time and always computes ordering accurately.

Netzer and Ghosh Algorithm: Netzer and Ghosh [18] present a precise and efficient algorithm for trace analysis of programs with post-wait synchronization, without `clears`; the complexity of their algorithm is $O(np)$ where n is the number of synchronization events and p is the number of processes.

3 Depth-First Search Algorithm for Event Ordering

The algorithm presented in this paper computes the ordering for a trace of a parallel program using synchronization with events. The algorithm is based on depth-first search (DFS) [10, 23] and traverses the augmented task graph containing synchronization and task edges. The order in which the edges are traversed is determined by a heuristic. The edges are then classified based on the DFS algorithm. Some edges are discarded and a final graph is constructed from the remaining edges. This graph is used to determine ordering between events.

3.1 Necessary Conditions for an Ordering to be Feasible

Ghosh [8, 11] notes that with regard to the Exhaustive Pairing Algorithm (EPA), that the following conditions are true for every feasible execution ordering of a parallel program using explicit event synchronization (`post`/`wait` synchronization) without `clears`.

- Operations within the same task are sequential and are hence ordered.
- Every `wait` that is triggered is preceded by at least one `post` on the event.
- These two requirements must be satisfied by the execution order without leading to any cycles in the order.

3.2 Task Graphs

A task graph is a directed acyclic graph that represents the ordering among the nodes that represent coordination operations; coordination operations are those operations that determine ordering (which comprise of task spawning, task merging and synchronization operations) and those that represent accesses to shared memory. The arcs represent the guaranteed ordering between the nodes in the same process. A unique root node represents the start of the parallel program; there is a directed edge from the root node to the first nodes in each process. Task graphs are acyclic.

3.3 Depth-First Search Algorithm for Event Ordering

The algorithm computes the guaranteed ordering and gives all possible ordering for a program that uses event synchronization using `post`, `wait` and no `clears`. The graph showing guaranteed ordering for a program is built based on the three conditions given above that are guaranteed to hold in every feasible ordering. The algorithm uses an augmented task graph. The augmented task graph is constructed from a task graph by adding an edge from every post on an event to every wait on the same event. All operations within a task are connected by directed arcs, the direction of the arcs representing the sequential execution within a task. The edges of the task graph can be classified into two types:

Task edges: These edges join two operations within the same task and represent the sequential execution ordering of the operations within a task.

Synchronization edges: These edges represent the synchronization between two tasks. For synchronization using `post`/`wait` without `clears`, the only valid synchronization edges are those from a `post` on an event to a `wait` on the event.

Even though task graphs are acyclic, the augmented task graph may contain cycles. The algorithm presented next performs a depth-first search of the augmented task graph, classifies edges, and then computes guaranteed ordering. Fig. 1 defines the notation used in the algorithm. Note that we do not present pseudocode for the procedures, *discard_edges()*, *compute_ordering()*, *delete_synchronization_edges()*, and *new_edge_discovered()*.

n ∈ N	≡ Set of nodes in the task graph.	
e ∈ E	≡ Set of edges in the task graph.	
	$n_1^{t_i} \xrightarrow{e} n_2^{t_k}$	
$n_1^{t_i}$	≡ Source node of the edge (Belongs to task t_i).	
$n_2^{t_k}$	≡ Destination node of the edge (Belongs to task t_k).	
R	≡ Root node for the task graph.	
seq_num(n)	≡ Sequence number assigned to the node; it is the order in which the nodes are visited by the algorithm.	
seq_num	≡ Global sequence number.	
f(n)	≡ Father of node n.	
last(t_i)	≡ Last node of task t_i that has already been processed	
type(e)	≡ Type of Edge	
	Unused	: Edge has not been visited.
	Tree	: Edge e was used to discover destination node n_2.
	Forward	: Edge e where *seq_num*(n_1) $\leq$*seq_num*(n_2).
	Back	: Edge e where *seq_num*(n_1) $>$*seq_num*(n_2) AND n_2 is an ancestor of n_1.
	Cross	: Edge e where *seq_num*(n_1) $>$*seq_num*(n_2) AND n_2 is not an ancestor of n_1.

Fig. 1. Notation used in the algorithm

The procedure $init()$ (see Fig. 2) initializes the sequence number of all nodes to zero and the father of all the nodes to UNDEFINED. These parameters are assigned their proper values by the algorithm. All edges including synchronization edges are initialized to edge type UNUSED. The search is started with the root node of the augmented task graph as the source node and with the global sequence number of zero, as no nodes have

procedure $init()$ {
∀ e ∈ E   *type*(e) ⟵ UNUSED
∀ n ∈ N   *seq_num*(n) ⟵ 0; $f(n)$ ⟵ UNDEFINED ;
t_i ⟵ 0 ; $n_1^{t_i}$ ⟵ R ; /* root node */
df_order$(n_1^{t_i}, 0)$;
discard_edges$()$;
compute_ordering$()$;
delete_synchronization_edges$()$; }

Fig. 2. The init procedure

procedure *df_order*$(n_1^{t_i}$,*seq_num*$)$ {
  seq_num++; *seq_num*$(n_1^{t_i})$ ⟵*seq_num*;
  process_edge$(n_1^{t_i}$,*seq_num*$)$; }

Fig. 3. The df_order procedure

procedure *process_edge*$(n_1^{t_i}$,*seq_num*$)$ {
  while $(((n_1^{t_i} \longrightarrow n_2^{t_k}) \longleftarrow$ *unused_incident_edge*$(n_1^{t_i})) \neq$ NULL) {
    type(e) ⟵ USED
    if (*seq_num*$(n_2^{t_k}) \neq 0$) {
      new_edge_discovered(e)
      continue }
    else {
      if ((*perform_synchronization_operation*$(n_2^{t_k}) \neq$ WAIT)
        &&(*prev_nodes_of_task_already_processed*(t_k))) {
        new_edge_discovered(e)
        $f(n_2^{t_k}) \longleftarrow n_1^{t_i}$;   *last*$(t_k) \longleftarrow n_2$; $n_1^{t_i} \longleftarrow n_2^{t_k}$
        df_order$(n_1^{t_i}$,*seq_num*$)$ }
      else {
        new_edge_discovered(e)
        continue } } }
if ($f(n_1^{t_i}) \neq$UNDEFINED) {
  $n_1^{t_i} \longleftarrow f(n_1^{t_i})$
  process_edge$(n_1^{t_i}$,*seq_num*$)$ }
else if (there exists an $n_2^{t_k}$ such that *seq_num*$(n_2^{t_k}) > 0$) {
  $n_1^{t_i} \longleftarrow n_2^{t_k}$
  df_order$(n_1^{t_i}$,*seq_num*$)$ } }

Fig. 4. The process_edge procedure

been visited by the algorithm. The root node belongs to task 0 and the rest of the nodes belong to tasks 1 to total number of tasks in the augmented task graph.

The procedure *df_order*($n_1^{t_i}$,*seq_num*) (Fig. 3) is a recursive procedure that is invoked each time a new node is visited. This procedure calls the procedure *process_edge* to differentiate between discovery of a node and visit of a node; a node could be discovered but not visited. When a node is discovered (or rediscovered), but not visited, the node is not assigned (or reassigned) a sequence number, but the edge used to discover the node is marked USED or assigned a edge type. The sequence number of a node is the the global sequence number when the node was visited for the first time. After the node is assigned a sequence number the node is further processed by the procedure *process_edge*(n_1,*seq_num*($n_1^{t_1}$)).

The procedure *process_edge*($n_1^{t_i}$,*seq_num*) (Fig. 4) processes all incident edges from the source node, $n_1^{t_i}$. The procedure *unused_incident_edge*($n_1^{t_i}$) returns unused incident edges based on a heuristic. The heuristic assigns weights to the edges depending on the type of source node and the type of the destination node. The edge with the greatest weight is returned by the procedure.

An edge that is returned by the *unused_incident_edge*($n_1^{t_i}$) procedure is marked USED. If the sequence number of the destination node $n_2^{t_k}$ is not zero *i.e.,* the node has already been visited and hence assigned a sequence number, the procedure continues to the next unused edge if any. If the sequence number of the destination node $n_2^{t_k}$ is zero, *i.e.,* the node has not been visited, then the node is examined to determine if the node is a synchronization node or an ordinary node. If the node is a synchronization node, the synchronization operation is performed on the event. If the operation is a `post` on the event, the event is posted. The algorithm keeps track of the order in which events become posted during depth-first search in the array *sigorder;* the event variable that becomes posted at the ith position is stored in *sigorder*[i]. If the operation is a `wait` then the node can be visited only if the event is posted. This ensures the second necessary condition for a feasible ordering is satisfied. The algorithm also checks if all previous nodes of the task t_k have been processed which ensures that the first condition for a feasible ordering is satisfied. If the first two conditions have been satisfied, the destination node $n_2^{t_k}$ is visited.

If the algorithm visits the destination node $n_2^{t_k}$, the father of the destination node is the source node $n_1^{t_i}$. The destination node $n_2^{t_k}$ is then assigned to the last node processed for the task t_k. The destination node $n_2^{t_k}$ becomes the source node $n_1^{t_i}$ and the procedure *df_order*($n_1^{t_i}$,*seq_num*) is invoked. If the destination node has not yet been visited but cannot be visited in this iteration of the algorithm because one one of the two conditions is not satisfied, the algorithm processes any other unused incident edge for the source node $n_1^{t_i}$.

If all the incident edges from the source node have been processed, the procedure *process_edge* ($n_1^{t_i}$,*seq_num*) checks if the father of the source node $n_1^{t_i}$ is defined. If so, the procedure backtracks by assigning the father of the source node $f(n_1^{t_i})$ as the source node $n_1^{t_i}$ and recursively calling itself. If all the incident edges from the source node have been processed and the father of the source node is not defined, the algorithm checks if there are any unvisited nodes in the augmented task graph *i.e.,* any node which has its sequence number undefined. If the augmented task graph is weakly connected, *i.e.,* the underlying undirected graph is connected, the presence of nodes whose sequence numbers are undefined indicate potential deadlocks in the program. If the augmented task graph is not weakly connected, then procedure *df_order*($n_1^{t_i}$,*seq_num*) is invoked with $n_1^{t_i}$ equal to a node whose sequence number is undefined, and which has no incoming edges.

It should be noted that while the task graph is a directed acyclic graph, the augmented task graph may contain cycles. depth-first search of directed graph that results in *back edges* indicates the presence of cycles. Since cycles are impossible in event orderings, back edges discovered by DFS can be discarded. After termination of the algorithm the modified DFS tree obtained is further processed to determine guaranteed ordering between the accesses to shared memory. The following rules are used to process the modified DFS tree obtained by the routine *discard_edges()*.

- All back edges are discarded. (Lemma 1 in Section 3.5)
- All task edges that are forward edges are discarded. (Lemma 2 in Section 3.5)
- Forward edges from the root node can be discarded.
- All other edges *i.e.,* tree, cross and forward that have not been discarded are considered as they represent ordering or potential ordering.

After the termination of the modified DFS, the edges that can be discarded are discarded. The remaining edges are analyzed to determine potential races and nondeterminism by the routine *compute_ordering()*. The procedure *compute_ordering()* adds the Guaranteed to Execute Before (GEB) edges to the task graph based on the classification of the edges remaining in the augmented task graph after the termination of the modified DFS and *discard_edges()*. The procedure *delete_synchronization_edges()* deletes all synchronization edges that are not GEB edges from the task graph.

If more than one edge terminates on a `wait` of an event, these edges represent possibly *different* execution paths that may be taken by the program; such multiple paths represent possible nondeterministic behavior due to shared memory accesses along these paths. If the set of edges ending at a `wait` node, say wait(X), contains a tree edge, and either a cross-edge or an undiscarded forward edge, then there is no guaranteed ordering between any `post(X)` and that `wait(X)`. Otherwise, there is a guaranteed ordering from `post(X)` to `wait(X)`. The procedure *compute_ordering()* first adds all such guaranteed edges. For all events Y for which no guaranteed ordering exists from `post(Y)` to `wait(Y)`, the procedure uses the array *sigorder* to find the guaranteed ordering edges as follows. Let k be an integer such that *sigorder*[k] = Y. Find the largest index $j(j < k)$ such that *sigorder*[j] = Z is an event for which there is a guaranteed ordering from `post(Z)` to `wait(Z)`. Add an edge from the node `post(Z)` to `wait(Y)`. If no such Z exists, there are no events in the program that are guaranteed to execute before Y (except the root node and all nodes preceding it in within the same process).

3.4 Heuristic Used in the Algorithm

Table 1. Assignment of weights to task edges by heuristic

	WAIT(A)	POST(A)	X	WAIT(B)	POST(B)
POST(A)	$\gg \mathcal{N}$	$> \mathcal{N}$	$\mathcal{N}$	$< \mathcal{N}$	$> \mathcal{N}$
WAIT(A)	$< \mathcal{N}$	$> \mathcal{N}$	$\mathcal{N}$	$< \mathcal{N}$	$> \mathcal{N}$
X	$< \mathcal{N}$	$> \mathcal{N}$	$\mathcal{N}$	$< \mathcal{N}$	$> \mathcal{N}$
R	$< \mathcal{N}$	$> \mathcal{N}$	$\mathcal{N}$	$< \mathcal{N}$	$> \mathcal{N}$

$\mathcal{N}$ denotes a Normal weight. R denotes the root node. X denotes a node which corresponds to a local (non-synchronization) operation in a task.

The heuristic assigns the weights to the edges based on the tables (See Table 1 and Table 2). The rows denote the type of source nodes and the columns indicate the types of destination nodes for each edge. The entry in the table denotes the weight assigned to the edge; if an edge can not exist, it is denoted by an asterisk. The heuristic is used by the routine *unused_incident_edge* to determine the next edge to be returned to the modified DFS of the algorithm. The heuristic is biased to ensure that coordination edges are processed before task edges.

3.5 Correctness of DFS Algorithm for Event Synchronization without `Clears`

In order to prove that correctness of the algorithm, we make the following observations regarding the DFS.

Observation 1: A DFS always traverses an edge of a given digraph in any direction only once.

Observation 2: A DFS traverses all edges of a digraph that is at least minimally connected.

Observation 3: A forward and cross edge with respect to one DFS tree of a digraph can be a tree edge with respect to another DFS tree of the digraph.

Observation 4: If $i \xrightarrow{f} j$ is a forward edge, then there exists a path composed of tree edges of length $n > 1$, $i \longrightarrow k_1 \longrightarrow k_2 \longrightarrow \cdots \longrightarrow k_{n-1} \longrightarrow j$.

The algorithm presented here is a modification of DFS and preserves the properties of the DFS. The four observations regarding DFS hold true here as well. Since, there is an edge from the root node to the first node in each task and since the search is always started at the root, there is a single DFS tree in the case of weakly connected task graphs. The algorithm only modifies the order in which the edges of the task graph are traversed. The routine *unused_incident_edge* returns an unused edge according to a heuristic. This routine also ensures that the two factors that determine ordering are always met while the task graph is being traversed by the algorithms. The two conditions that determine ordering that are enforced by the routine *unused_incident_edge* are:

1. The ordering of operations within a task is sequential. The i^{th} operation of a task t always executes before operation $(i + 1)$ of task t;
2. A `wait` on an event cannot continue until there has been at least one `post` on that event.

Lemma 1. All edges that are designated as back edges by the modified DFS of the algorithm can be ignored.

Proof: If *b* is a back edge of a DFS tree such that $i \xrightarrow{b} j$ then,

– The DFS sequence number of *i* is greater the that of *j*. The DFS sequence number is the DFS sequence number that represents the order in which the nodes were discovered by the DFS. Back edges are edges in a DFS tree, from node that were discovered later on in the search to those discovered earlier.
– Node *i* is an ancestor of node *j*.

Back edges can not be task edges. A back edge *b* can only be a synchronization edge, specifically a synchronization edge from a `post` in a task to a `wait` in another task. From the definition of a back edge this indicates that the `wait` node *j* executed before the `post` node *i*. Considering the semantics of `post`/`wait` synchronization this implies that the `wait` was triggered by another `post` that executed before the `post` *i*. Since a `post` on an event has no effect on an event that has already been posted, the `post` *i* may be ignored and the back edge *b* may be ignored. Also a back edge in a DFS tree can never a tree edge in a different DFS tree of the same digraph. □

Lemma 2. Task edges are either tree or forward edges. Forward task edges may be ignored in the analysis of the task graph.

Proof: If *T* is a task edge of a DFS tree such that $i \xrightarrow{T} j$ then,

- The DFS sequence number of node *i* is lower than that of node *j*, since *unused_incident_edge* ensured that a node earlier in a task is discovered earlier by the modified DFS than one later on.
- The node *j* is always an descendant of node *i*.

It has been already proved in Lemma 1 that tree edges cannot be back edges.

The destination node of a task edge is a descendant of the source node and therefore there exists an ancestor/descendant relationship between the two node, since no such relationships can exist between the nodes of a cross edge, task edges cannot be cross edges.

If a task graph is a forward edge then it may be ignored in the analysis of the task graph. This follows from observation 4 on DFS trees. Any ordering information that may be obtained from a forward task edge $i \xrightarrow{T_f} j$ about potential ordering between the nodes *i* and *j* and other nodes can be obtained by considering the path(s) $i \longrightarrow k_1 \longrightarrow \cdots \longrightarrow k_{n-1} \longrightarrow j$ composed of either tree or tree and cross edges. The ordering information that can be obtained from the forward task graph is a subset of the information that can be obtained from the alternate path. The forward tree edge contains information regarding the sequential execution within the task only, whereas the alternate path contains information regarding ordering involving other tasks as well. □

Lemma 3. After the termination of the modified DFS and after deletion of unwanted edges by *discard_edges()* , there exists a synchronization edge $n_1^{t_i} \xrightarrow{S} n_2^{t_j}$ if the edge is a triggering synchronization edge TSE, $n_1^{t_i} \xrightarrow{\text{TSE}} n_2^{t_j}$ i.e., the synchronization edge is traversed and the `post` corresponding to the edge is the first `post` to trigger the `wait`, in any one of the possible execution orderings.

Proof: To aid in the proof, the concept of a Triggering Synchronization Edge (TSE) is introduced. A TSE is a synchronization edge from a `post` on an event to a corresponding `wait`, which first sets the state of the event to posted. All other other `posts` and therefore their corresponding edges have no effect on the event since the event is already posted (and remains posted since `clears` on the event are not allowed). An edge $n_1^{t_i} \xrightarrow{\text{TSE}} n_2^{t_j}$ is a TSE iff the edge is executed at least in one of the possible executions and if the edge triggers the event *i.e.,* the first to set the event to posted. To prove that if an edge is a TSE it is present as a synchronization edge in the modified DFS tree after the algorithm terminates and after unnecessary edges are deleted by *discard_edges()*.

Observation 2 on the DFS states that all edges of a digraph are traversed if the digraph is minimally connected. Since the the original task graph has an edge from each `post` on an event to all `waits` on the same event, all these edges are traversed by the modified DFS. These edges are classified by the search into tree, cross, forward and back edges. The set of all TSE is a subset of the set of edges that the search traverses, therefore all TSE are processed by the algorithm. Since the only synchronization edges that are discarded are back edges, proof that no TSE are back edges will complete the proof that if an edge is a TSE, the synchronization edge will be present in the modified DFS tree. Assume that the edge TSE is a TSE such that $n_1^{t_i} \xrightarrow{\text{TSE}} n_2^{t_j}$ then $n_1^{t_i}$ is a `post` on an event and $n_2^{t_j}$ is the `wait` that is triggered for the first time, by the `post`. If TSE were to be a back edge this would imply that the DFS sequence number of $n_2^{t_i}$

would be greater than DFS sequence number of $n_2^{t_j}$, and therefore imply that $n_2^{t_j}$ has been already visited by the modified DFS. Since $n_2^{t_j}$ is a `wait` this would imply that the `wait` has already been posted, therefore the edge would not be a TSE. This violates our original assumption, and therefore a TSE cannot be a back edge.

Assume that a synchronization edge is not a TSE. Then for every possible execution, the node $n_2^{t_j}$ or the `wait` node is always triggered before the synchronization edge. $\square$

Theorem 4. After the termination of the algorithm, for all nodes in the task graph there exists a path from n_1 to n_2, only if, $n_1^{t_i} \xrightarrow{\text{GEB}} n_2^{t_j}$.

Proof: After the termination of the algorithm the only paths that exist are the paths within a task and the GEB arcs. Since task paths are sequential, portions of the path between n_1 and n_2 that are along tasks obey the GEB property. We have to prove that the begin and end nodes of every GEB arc added by the algorithm obey the GEB property, since these are the only arcs that remain between tasks after the algorithm terminates.

For every event in the task graph, after the algorithm ensures that all TSE remain (Lemma 3). The procedure *compute_ordering()* using the array *sigorder* adds GEB arcs from the closest common node that executes before all the TSE; therefore, the source node of the GEB must execute before the destination node of the GEB, which is the event. This is true for all events in the graph.

Since both the sequential path within tasks and the GEB arcs both obey the GEB property and these are the only two type of paths that remain in the task graph, any combination of the two would obey the GEB property. $\square$

Table 2. Assignment of weights by heuristic to synchronization edges

	WAIT(A)	POST(A)	X	WAIT(B)	POST(B)
POST(A)	$\gg \mathcal{N}$	*	*	*	*
WAIT(A)	*	*	*	*	*
X	*	*	*	*	*
R	$< \mathcal{N}$	$> \mathcal{N}$	$\mathcal{N}$	$< \mathcal{N}$	$> \mathcal{N}$

An asterisk denotes that the edge cannot exist. $\mathcal{N}$ denotes a Normal weight. R denotes the root node. X denotes a node which corresponds to a local (non-synchronization) operation in a task.

Within each task, we only need to consider the first operation (either a `post` or a `wait`) on any event variable, because of the GEB edges within each task. The complexity of the algorithm presented here is $O(ep)$ where e is the number of event variables and p is the number of processes [21]. From this algorithm, one can derive an $O(e)$ algorithm for detecting deadlocks and infinite waits [21]. The reader is referred to [21] for details.

4 Application of the Algorithm on Sample Program Segments

This section shows the application of the DFS-based heuristic to sample program segments that have been used in the literature. Consider program segment 1 in Fig. 5 from [18]. On a cursory examination, this segment appears to be race-free. The program when analyzed by the algorithm, shows the existence of a race between the accesses to S in task 2 and task 3 *i.e., I = S* and *S = 1* respectively. The access task 1 is ordered with respect to the other two accesses, and does not constitute a race. The augmented task

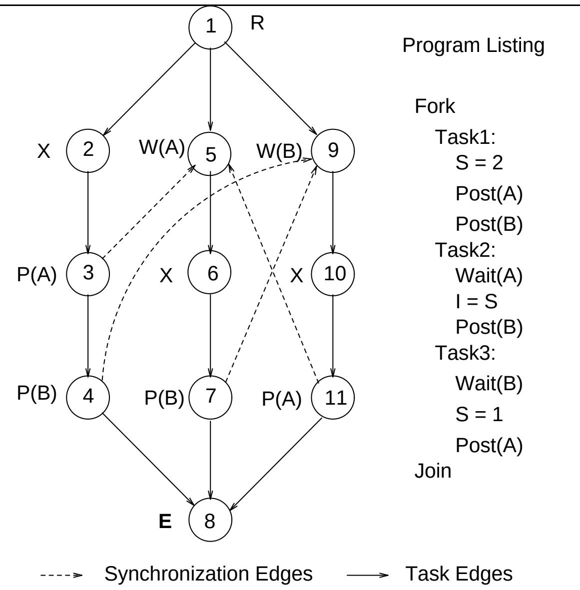

Fig. 5. Augmented task graph for program segment 1

graph representing the program segment is illustrated in Fig. 5. The synchronization edges are added from all `posts` on an event to all `waits` on the same event. The modified DFS traverses the task graph and terminates. The tree generated by the search is illustrated in Fig. 6. The tree edges are then examined and all forward edges within a task and all back edges are discarded, by the routine *discard_edges()*. These edges are marked with an asterisk in Fig. 6. The procedure *compute_ordering()* then examines the remaining edges and computes the closest common edge to each event in the graph.

In the task graph the only edge that could potentially trigger the event A is that from node 3 (edge [3,T] in Fig. 6), since the only other edge that could trigger event A is the back edge [9,B] that is discarded. Therefore, node 3 is guaranteed to execute before node 5. When we consider the wait on event B (node 9), there are two `posts` (nodes 4 and 7) that could trigger it; there are two edges that terminate at the node in the DFS tree, namely [6,T] and [13,C]. Thus, there is no guaranteed from either node 4 or node 7, to node 9. Since *sigorder*[1] is A and *sigorder*[2] is B and we have a GEB edge from node 3 to node 5, we add an edge from node 3 to node 9.

There is no guaranteed order between the accesses that lie along any of the paths with any access on the other path, or with an access along the path that follows the event. Therefore there is no ordering between the access at node 6 and that node 10. These accesses represent a race condition and is reported. The access at node 2 is ordered with respect to the other two orders. The graph showing the guaranteed ordering, and the GEB edges is illustrated in Fig. 7.

Program Segment for which the Common Ancestor Algorithm Generates Spurious Races
The second program segment (shown in Fig. 8) is one for which the *common ancestor algorithm* generates spurious races, as is shown by Emrath, Ghosh and Padua [8]. The algorithm proposed gives the correct ordering and does not report a spurious race for the program segment. The algorithm uses an augmented task graph with edges added as in Fig. 8. The DFS tree produced by the algorithm is shown in Fig. 9; the discarded edges are marked with asterisks.

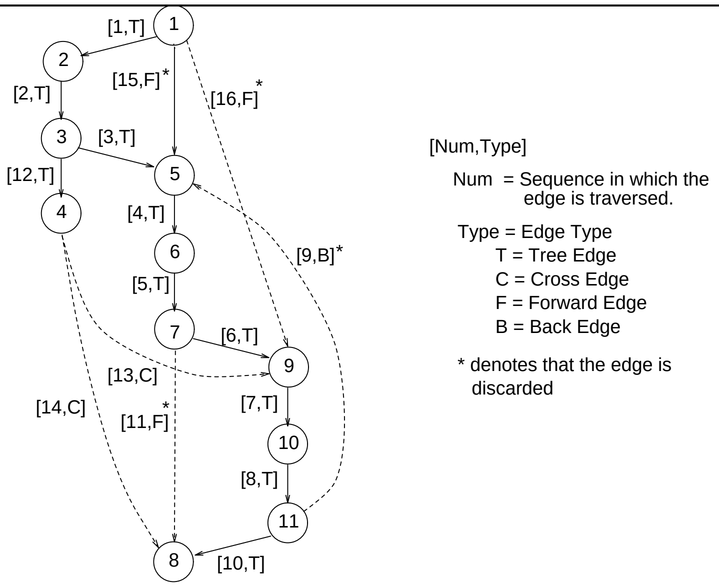

Fig. 6. DFS tree for program segment 1

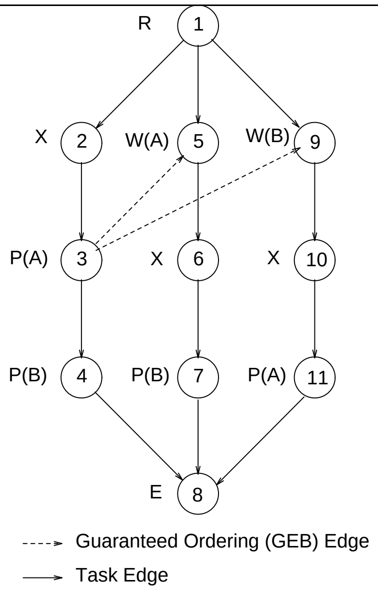

Fig. 7. Guaranteed ordering graph for program segment 1

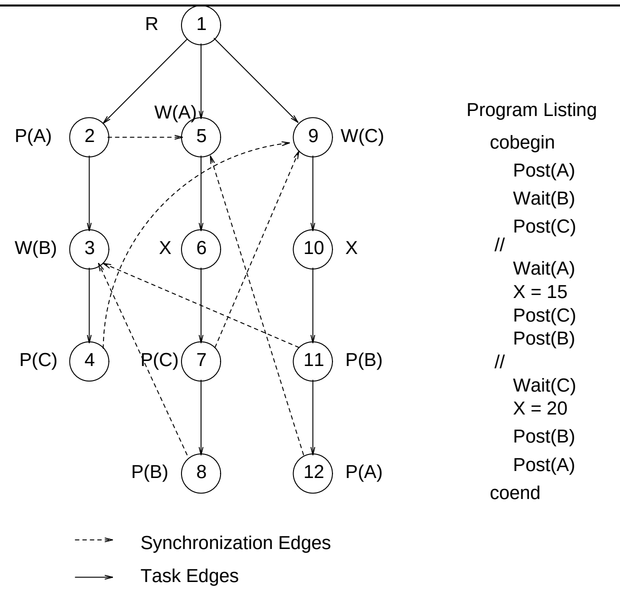

Fig. 8. Augmented task graph for program segment 2

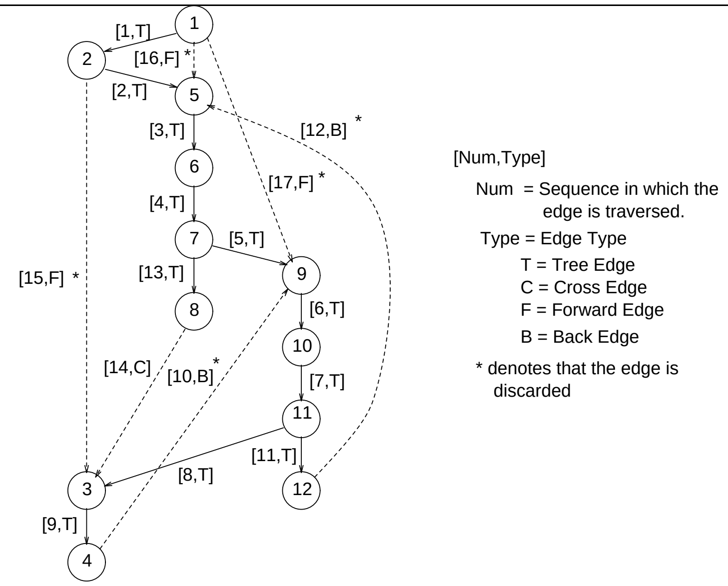

Fig. 9. DFS tree for program segment 2

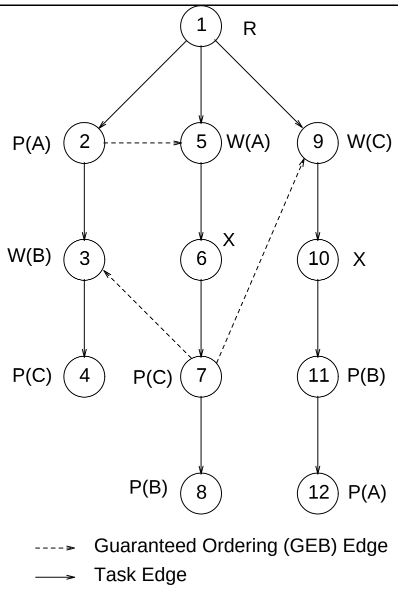

Fig. 10. Guaranteed ordering graph for program segment 2

The procedure *compute_ordering()* then examines the remaining tree, cross and forward edges and computes the closest common node for each event. In this program segment, the closest common node for event *A* is node 2, since the back edge from node 12 is discarded and the only TSE is [2,T]. For event *C* the only TSE is [5,T] since the other edge [17,F] is a forward edge in a task. For the event *B* there are two possible execution paths [14,C] and [8,T], since the forward edge [15,F] is discarded, the closest common node is the node where the paths containing the two possible TSE originate, node 7. This is because the events are triggered in the order A, C and then B, and both events A and C have GEB edges from posts to waits. Since a path exists between the two shared memory accesses *i.e.,* nodes 6 and 10, the accesses are ordered and no race is detected by the algorithm.

5 Summary

This paper presents an $O(ep)$ (e is the number of event variables and p is the number of processes) algorithm that computes guaranteed ordering for parallel programs using `post`/`wait` synchronization without `clears`. The algorithm first constructs an augmented task graph by adding a directed edge from every `post` on an event to every `wait` on the same event. The algorithm then uses a modification of depth-first search to classify the edges of the augmented task graph. While the task graph is acyclic, the augmented task graph may contain cycles. The order in which the edges are traversed is determined by a heuristic. Depending on the classification, some of the edges are discarded. The remaining edges are used to construct a graph representing the guaranteed ordering between events. This graph is then used to detect nondeterminacy in the program.

For programs that use `post`/`wait` synchronization with `clears`, the problem of detecting guaranteed ordering is NP-complete [8, 11]. In such cases, we use a heuristic [21] to recognize certain structures in the graph such as a `clear(X)` followed by a `wait(X)` within the same process (where X is an event), or a `wait(X)` followed by a `clear(X)` within the same process. Work is also in progress in deriving a DFS-based algorithm for semaphore synchronization.

References

1. Allen, T. R. and D. A. Padua, "Debugging Fortran on a Shared Memory Machine," *Proc. Intl. Conf. Parallel Processing*, pp. 721-727, St. Charles, IL, Aug. 1987.
2. Balasundaram, V. and K. Kennedy, "Compile-time Detection of Race Conditions in a Parallel Program," *Proc. 3rd Intl. Conf. Supercomputing*, pp. 175-185, Jun. 1989.
3. Bernstein, A. J., "Analysis of Programs for Parallel Processing," *Proc. IEEE Trans. on Electronic Computers*, EC-15(5), pp. 757-763, Oct. 1966.
4. Callahan, D., K. Kennedy and J. Subhlok, "Analysis of Event Synchronization in a Parallel Programming Tool," *Proc. 2nd Symposium on the Principle and Practice of Parallel Programming*, pp. 21-30, Mar. 1990.
5. Callahan D. and J. Subhlok, "Static Analysis of Low-Level Synchronization," *Proc. SIGPLAN Workshop on Parallel and Distributed Debugging*, pp. 100-111, May. 1988.
6. Choi, J., B. P. Miller and R. H. B. Netzer "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Trans. Programming Languages and Systems*, pp. 491-530,Vol. 13, No. 4, Oct. 1991.
7. Emrath, P. A., S. Ghosh and D. A. Padua, "Event Synchronization Analysis for Debugging Parallel Programs," *Proc. Supercomputing 89*, ACM Press, New York pp. 580-588, 1989.
8. Emrath, P. A., S. Ghosh and D. A. Padua, "Detecting Nondeterminacy in Parallel Programs," *IEEE Software*, pp. 69-77, Jan. 1992.
9. Emrath, P. A. and D. A. Padua, "Automatic Detection of Nondeterminacy in Parallel Programs," *Proc. SIGPLAN Workshop on Parallel and Distributed Debugging*, pp. 89-99, May. 1988.
10. Even, Shimon, *Graph Algorithms*, Computer Science Press, 1976.
11. Ghosh, S., "Automatic Detection of Nondeterminacy and Scalar Optimizations in Parallel Programs," *Ph.D. Thesis,*, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1992.
12. Helmbold, D. P., C. E. McDowell and J. Wang, "Analyzing Traces for Anonymous Synchronization," *Proc. Intl. Conf. Parallel Processing*, pp. 70-77, St. Charles, IL, Aug. 1990.
13. Helmbold, D. P., C. E. McDowell and J. Wang, "Determining Possible Event Orders by Analyzing Sequential Traces," *IEEE Trans. Parallel and Distributed Systems*, pp. 827-839, Vol. 4, No. 7, Jul. 1993.
14. Hood, R., K. Kennedy and J. Mellor-Crummey, "Parallel Program Debugging with On-the-fly Anomaly Detection," *Proc. Supercomputing 90*, New York pp. 74-81, Nov. 1990.
15. McDowell, C. E. and D. P. Helmbold, "Computing Reachable States of Parallel Programs," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 89-99, May. 1991.
16. Miller, B. P. and J. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proc. ACM SIGPLAN Conf. on Programming Lang. Design and Implementation*, pp. 135-144, Jun. 1988.
17. Netzer, R. H. B., "Race Condition Detection for Debugging Shared-Memory Parallel Programs," *Ph.D. Thesis*, Dept. of Computer Science, University of Wisconsin-Madison, 1991.
18. Netzer, R. H. B. and S. Ghosh, "Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization," *Proc. Intl. Conf. Parallel Processing*, Aug. 1992.
19. Netzer, R. H. B. and B. P. Miller, "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions," *Proc. Intl. Conf. Parallel Processing*, II-93-II-97, Aug. 1990.
20. Netzer, R. H. B. and B. P. Miller, "What are Race Conditions? Some Issues and Formalizations," *ACM Letters on Programming Languages and Systems*, Vol. 1 No. 1, Mar. 1992.
21. Ramanujam, J. and Ashvin Mathew, "Detection of Nondeterminacy in Parallel Programs," *Technical Report* 94-01-03, Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, Jan. 1994.
22. Schonberg, E., "On-the-fly Detection of Access Anomalies," *Proc. ACM SIGPLAN Conf. on Programming Lang. Design and Implementation*, pp. 285-297, Jul. 1989.
23. Tarjan, R., "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, Vol. 1, pp. 146-160, 1972.

This article was processed using the LaTeX macro package with LLNCS style