On-The-Fly Detection Of Access Anomalies

by

Edith Schonberg

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #149
October, 1988

This work was supported under the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-FG02-88ER25052.

ABSTRACT

Access anomalies are a common class of bugs in shared-memory parallel programs. An access anomaly occurs when two concurrent execution threads both write (or one thread reads and the other writes) the same shared memory location. Approaches to the detection of access anomalies include static analysis, post-mortem trace analysis, and on-the-fly monitoring.

A general on-the-fly algorithm for access anomaly detection is presented, which can be applied to programs with both nested fork-join and synchronization operations. The advantage of on-the-fly detection over post-mortem analysis is that the amount of storage used can be greatly reduced by data compression techniques and by discarding information as soon as it becomes obsolete. In the algorithm presented, the amount of storage required at any time depends only on the number V of shared variables being monitored and the number N of threads, not on the number of synchronizations. Data compression is achieved by the use of two techniques called *merging* and *subtraction*. Upper bounds on storage are shown to be $V \times N^2$ for merging and $V \times N$ for subtraction.

1. Introduction

In shared-memory parallel programs, a common and vexing class of parallel bugs arise from *access anomalies*. By shared-memory parallel programs, we mean programs in which concurrent execution streams directly read and write shared-memory. An *access anomaly* occurs when two concurrent execution streams both write (or one reads and the other writes) the same memory location. Access anomalies are almost always bugs.[1] They result either from a violation of data dependences or from miscellaneous program logic errors, such as in subscripting or addressing. In Ada access anomalies render a program erroneous, that is, they have undefined semantics.

To illustrate an access anomaly, consider the small code sequence below, written in Fortran extended by a parallel **doall**:

[1] An exception to this is chaotic relaxation [Bau].

```
doall i = 1,2
        A[i] = B[i] + 1
        ....
        doall j = 1,2
                C[i, j] = A[i+1] + B[j]
        endall
        ....
endall
```

The dynamic parallel flow of control for this program is shown in Figure 1, where each separate execution stream is labeled by an identifier T_i. Since location A[2] is written by execution stream T_2 and read by execution streams T_3 and T_4, there is an access anomaly. (In this case, the **doall** does not respect some data dependence in the original sequential program.)

Access anomalies are often difficult to locate using conventional debugging techniques because they are sources of non-determinism. Different executions of the same program may lead to different orderings in the code sections containing the anomaly. For example, in Figure 1, location A[2] in T_3 may or may not be accessed before the assignment in T_2, depending on relative execution speeds; thus the value of C[1,1] may vary from run to run.

In this paper we present an algorithm for detecting access anomalies for a general model of parallel programs. Detection is performed *on-the-fly*, and the algorithm has the property that the space needed to perform the analysis depends only on the number of shared variables and the number of parallel execution threads, not on the number of synchronizations or communications among the parallel threads. The algorithm therefore offers a practical solution to the anomaly detection problem for large and long-running programs.

In the rest of this section, we discuss other work that has been done in this area. Section 2 presents a formal framework for our algorithm. A general high-level algorithm is presented in Section 3 for programs with pairwise synchronization. The storage properties of the algorithm are derived from two data compression techniques— the first, called *merging*, is described in Section 4, and the second, called *subtraction*, is described in Section 5. We prove that the amount of storage used is bounded by $V \times N^2/2$ for merging and $V \times N$ for subtraction, where V is the number of shared variables monitored and N is the number of execution threads. Section 6 extends these results to programs with nested fork-join operations; and Section 7 shows how to apply the algorithm to a variety of programming language constructs, including the Ada rendezvous, barrier synchronization, lock/unlock operations, and message passing primitives. Conclusions are drawn in Section 8.

1.1. Related Work

A variety of approaches have been explored for detecting access anomalies. One approach is static analysis [App, Car, Tay], the goal of which is to detect all *potential* access anomalies prior to execution.

In [App] and [Tay], sections of code which are potentially concurrent are identified; shared variables read and written in these sections are potential anomalies. To reduce the set of potential anomalies found, further static analysis is required, such as subscript range and alias analysis, as well as data dependence analysis [Bur, Car]. At best, static analysis can only indicate a conservative superset of potential anomalies. Faced with too many false reports, it is easy for a programmer to disregard the advice of an overly conservative static tool.

Dynamic access anomaly detection, on the other hand, is a complementary approach that can guarantee that there are no anomalies in any particular *execution instance* of a program. Dynamic detection tests whether a potential anomaly revealed by static methods is a real anomaly, while static analysis is useful to reduce the number of variables that have to be monitored at run-time. In [Mil], the dynamic access anomaly detection problem is defined for a variety of parallel programming constructs, but no efficient algorithm is presented.

The approach of [All] combines static analysis with dynamic detection. A history trace is generated during program execution, and the trace is analysed in a port-mortem phase to detect anomalies. Static analysis is used to reduce the number of variables that have to be checked at run-time, in order to compress the dynamic trace, and to locate potential access anomalies that may be *hidden* by earlier access anomalies. The drawback of methods based on history traces and post-mortem analysis is that traces readily grow too large, even when compression techniques are used. In [All], a record must be stored for each basic block executed and additionally for each array operation.

On-the-fly detection is a more promising dynamic approach, in which anomalies are found while the program is executing, rather than in a post-mortem phase. Additional storage is used to save shared variable access information necessary to perform the analysis, but the size of this storage is much less than is required by trace-based methods.

Variants of on-the-fly algorithms are given in [Nud, Sni]. In both of these algorithms, a solution is presented only for a limited shared-memory programming model in which parallelism is expressed using a nestable **doall-endall** construct, but the model does not include primitives for communication and synchronization among parallel threads.

The on-the-fly anomaly detection algorithm presented here is more general than both those of [Nud] and [Sni]. The algorithm of [Sni] is in fact a special case of this more general algorithm, and so we first summarize [Sni] below.

1.2. On-The-Fly Anomaly Detection For Doall-Endall Programs

In the anomaly detection algorithm of [Sni], each execution stream created by a **doall** has its own *display*. The display holds a read flag and a write flag for each location that is being monitored. If a monitored location is read (written) in an execution stream T, the read (write) flag is set in the display of T. If T is a child of Q, *and* if this is the first read (write) of the monitored location within T, the corresponding bit must also be set in the display of Q, (and the parent of Q, etc.). On subsequent reads (writes) to the same location in T, no further updating is required. For example, in Figure 1, T_3 is a child of T_1. If location A[2] is accessed for the first time in T_3, the displays of both T_3 and T_1 must be updated.

The set of concurrent streams that start at a given **doall** terminate together at a corresponding **endall**; at termination time, displays are compared and any access anomalies are revealed. In our example, the anomaly involving A[2] is revealed when streams T_1 and T_2 terminate (see Figure 2). The only additional storage needed for this algorithm is a bitstring display for each current execution stream.

A shortcoming of this method is that when an anomaly is detected, the exact source location of the statements that caused the anomaly is not known. However, since the address of the variable involved and the sections of code where the anomaly occurs are known, it is possible to perform more detailed subsequent monitoring to precisely locate the anomaly.

The more general problem that we solve is formalized in the sections that follow.

2. Programming Model and Problem Definition

Given that parallel programs are often inherently non-deterministic, the access anomaly detection algorithm presented below applies only to a single *execution instance* of a program. An *execution instance* of a parallel program **P** is written as P.

We refer to the multiple execution threads of control in a parallel program execution P as *tasks*. Tasks are created and destroyed via a closed, nestable *fork-join* construct. One or more tasks are created at a *fork* statement. Each fork statement has a corresponding *join* statement, and all tasks created by a specific fork statement terminate together at the corresponding join. The parent task that executes the fork is blocked until all its children terminate.

Additionally, there is a primitive for pairwise synchronous task coordination. Synchronous coordination is a symmetric operation. If two tasks coordinate synchronously, then neither task can execute past the coordination point until the other task has reached the coordination point.

Below we refer to the fork, join, and coordination primitive as *parallel operations*, and the sequence of program statements executed in a single task between two parallel operations is referred to as a *sequential block*. A task is a series of sequential blocks.

Let *P* be an execution instance of a parallel program **P**, and let B_x and B_y be sequential blocks within tasks T_i and T_j of *P* respectively. Block B_y is said to be *dependent* on B_x if the sequences of operations among the tasks in *P* forces the execution of B_y to occur after B_x, regardless of the relative execution speeds of the tasks T_i and T_j. Dependences are determined by the particular parallel operations used, and are more formally specified as follows.

B_y executing in T_j is dependent on B_x executing in T_i iff any of the following hold.

(i) T_i is the same task as T_j, and B_x executes before B_y;

(ii) Block B_y begins with

- a *fork* operation, and T_i is the parent of T_j,
- a *join* operation, and T_i is one of the tasks executing the join, or
- a *synchronous coordination* operation between T_i and T_j, and B_x ends before the coordination point; or

(iii) There is another sequential block instance B_z in task T_k such that B_y is dependent on B_z and B_z is dependent on B_x.

If block B_y is not dependent on B_x and B_x is not dependent on B_y, blocks B_x and B_y are *concurrent*.

As examples, in Figure 3, B_2 is dependent on B_1, and concurrent with B_4, B_5, B_6, B_7, and B_8. The block B_3 is dependent on B_1, B_2, and B_4, and concurrent with all of the other sequential blocks.

Figure 4(a) illustrates synchronous coordination, in which vertical lines represent tasks, and horizontal lines represent coordination points. In this figure, sequential block B_4 is dependent on B_1 and B_3. Block B_5 is dependent on B_1, B_3, B_4, B_6, B_7, and B_9. Figure 4(b) indicates for each sequential block the set of other sequential blocks concurrent with it.

An access anomaly occurs when at least two sequential blocks are concurrent, and either each block writes to the same shared memory location, or one block writes and the other reads the same location. An *on-the-fly* access anomaly detection algorithm finds anomalies in a program while the program executes. To accomplish this, the algorithm must be able to determine the following information for each sequential block *B*:

(1) the shared variables read and written in *B*, and

(2) the set of sequential blocks that are concurrent with *B*.

While any given sequential block is executing, each block concurrent with it may have already finished executing, be currently executing, or not have commenced execution; it is necessary to save the information (1) and (2) until it is known that all concurrent blocks have completed.

Our goal in constructing an efficient algorithm is to minimize the amount of additional storage needed at any time. In particular, if the amount of storage required is not significantly less than the size of the trace data needed for post-mortem analysis, there is no advantage to on-the-fly analysis.

For clarity of exposition, we restrict attention in Sections 3-5 to programs with synchronous coordination operations only. In Section 3, a general high-level algorithm is specified; compression techniques are presented in Sections 4 and 5. Section 6 shows how to extend the algorithm for programs with fork-join operations also.

3. General Algorithm

For an execution instance P with *m* synchronous coordination operations, we choose a total ordering of the operations that is consistent with the partial ordering in P, and present a general anomaly detection algorithm by specifying the actions performed at each timestep $t = 1, 2, \ldots, m$.

At time t, we consider every sequential block B_x that is currently executing, starting to execute, or has finished executing, and associate with each such block a *shared variable set* $S_x(t)$**.** $S_x(t)$ **consists of two subsets: the shared variables read in** B_x **and the shared variables written in** B_x **by time** t**. Two shared variable sets are said to be** *concurrent* **if the sequential blocks associated with them are concurrent.**

If block B_x has finished executing before or at time t, $S_x(t)$ is said to be *complete*. If a block B_x is currently executing at time t, or just beginning to execute, $S_x(t)$ is said to be *incomplete*. Assuming **N** tasks at time t, there are **N** blocks currently executing, and **N** associated incomplete shared variable sets. The **N** incomplete shared variable sets are all concurrent with each other.

To illustrate these definitions, in Figure 5, each coordination point is labeled with a timestep. At time (2), the shared variable sets for blocks B_1, B_4, B_5, and B_8 are complete; the shared variable sets for blocks B_2, B_6, B_9, and B_{11} are incomplete; and no other shared variable sets exist.

The algorithm, presented below, detects anomalies by comparing complete concurrent shared variable sets at each timestep. At time t, each newly completed shared variable set $S_x(t)$ is compared with other completed shared variable sets that are concurrent with it. However, not all blocks concurrent with $S_x(t)$ are complete at time t.

To keep track of concurrent shared variable sets that will complete at a later time, we associate a *concurrency list* with each shared variable set. The concurrency list for a shared variable set $S_x(t)$ is the set of all tasks T such that the block B currently executing (or about to begin executing) in T at time t is concurrent with B_x. The concurrency list associated with $S_x(t)$ is written as $CL_x(t)$.

A concurrency list decreases in size over time as concurrent blocks finish executing. If a concurrency list $CL_x(t)$ becomes empty, the shared variable set S_x has been compared with all shared variable sets concurrent with it; thus, S_x can be deleted.

We trace the above steps for blocks B_1 and B_2 in Figure 5. Initially, there are four incomplete shared variable sets, with associated concurrency lists $CL_1(0) = \{T_2, T_3, T_4\}$, $CL_4(0) = \{T_1, T_3, T_4\}$, $CL_8(0) = \{T_1, T_2, T_4\}$, and $CL_{11}(0) = \{T_1, T_2, T_3\}$. At time (1), B_1 and B_4 are finished, and so $S_1(1)$ is compared with $S_4(1)$. However, there are other blocks executing, or not yet started, whose shared variable sets must be compared with S_1, namely B_8 and B_{11}, but the comparison can only be made at a later time. The concurrency list $CL_1(1)$ for shared variable set $S_1(1)$ is $\{T_3, T_4\}$, since after (1), the executing blocks B_8 and B_{11} are concurrent with B_1, but the executing blocks B_2 and B_5 are not concurrent with B_1. The concurrency list for the new incomplete shared variable set $S_2(1)$ is $\{T_2, T_3, T_4\}$.

At time (2), B_8 finishes, so $S_1(2)$ can be compared with $S_8(2)$. Since B_1 is not concurrent with B_9, $CL_1(2)$ becomes $\{T_4\}$. On the other hand, $CL_2(2)$ is the same as $CL_2(1)$.

At time (3), B_2 finishes, and is compared with B_5, B_6, and B_8. The concurrency list $CL_2(3)$ is $\{T_3, T_4\}$. The concurrency list $CL_1(3)$ equals $CL_1(2)$.

Finally at time (4), $S_1(4)$ is compared with $S_{11}(4)$. All blocks concurrent with B_1 have finished executing, the concurrency list $CL_1(4)$ is empty, and so $S_1(4)$ is deleted. The concurrency list $CL_2(4)$ has not changed.

Note that the size of a concurrency list can be at most **N**-1, and so there are at most $2^{\mathbf{N}}$ distinct concurrency lists. A concurrency list of size **N**-1 is always associated with an incomplete shared variable set $S_x(t)$; $CL_x(t)$ is $\{T_j : j \neq i\}$, where B_x is the *current* block in task T_i. Concurrency lists associated with complete shared variable sets at time t are always of size less than **N**-1.

This algorithm is described more precisely below.

Algorithm. At each timestep t, suppose a synchronous coordination operation is executed by block B_x in task T_i and block B_y in task T_j. The following four steps are performed:

(1) **Compare**. To check for anomalies, compare $S_x(t)$ with each shared variable set $S_z(t)$, provided that $S_z(t)$ is complete and T_i is in the concurrency list $CL_z(t-1)$; compare $S_y(t)$ with each complete shared variable set $S_z(t)$, provided that T_j is in the concurrency list $CL_z(t-1)$.

(2) **Initialize**. For each *new* sequential block B_z that begins executing at t, a new incomplete shared variable set $S_z(t)$ and concurrency list $CL_z(t)$ are generated.

(3) **Update**. Update all concurrency lists associated with shared variable sets that are complete at time t, to reflect the transition from time $t-1$ to t. This step is specified more precisely below.

(4) **Delete**. Delete any shared variable sets whose concurrency lists have become empty as a result of step (3).

The concurrency list update rule (step (3)) is derived from the following lemma.

Lemma 1. Suppose block B_x in task T_i and B_y in task T_j coordinate synchronously at time t. Let B_x' be the new block executing in T_i after the coordination point. Let B_z be another block that is complete at time t. If B_x is concurrent with B_z and B_y is not concurrent with B_z, then the new block B_x' is also not concurrent with B_z. Conversely, if B_x and B_y are both concurrent with B_z, and T_i and T_j coordinate, then the new incomplete blocks B_x' and B_y' executing after the coordination are also concurrent with B_z.

Proof. Since B_y is not concurrent with B_z, B_y depends on the block B_z. (B_z cannot be dependent on B_y, since B_z has finished executing first.) B_x' is dependent on B_y, and so it is also dependent on B_z. The converse also follows immediately from the definition of dependency and concurrency.□

The update rule for synchronous coordination operations is therefore specified as follows:

(3a) **Update**. For each complete shared variable set $S_z(t)$:

(i) if T_j is in $CL_z(t-1)$ and T_i is not, $CL_z(t) = CL_z(t-1) - \{T_i\}$; similarly,

(ii) if T_i is in $CL_z(t-1)$ and T_j is not, $CL_z(t) = CL_z(t-1) - \{T_j\}$;

(iii) otherwise, $CL_z(t) = CL_z(t-1)$.

Note that updating concurrency lists involves at most a linear scan over all lists. It is not necessary to know the history of synchronous coordination operations to perform the update step.

If shared variable sets are never deleted, this algorithm is not significantly better than post-mortem analysis. Figure 5 illustrates when the deletion step can be applied, namely, after time (4) when all tasks have synchronized with each other. Overall, however, the delete step can only be applied under limited circumstances. If there is no **N**-way task synchronization pattern such as that shown in Figure 5, no shared variable sets are ever deleted. Improvements derive from *merging*, which reduces the total number of shared variable sets, and *subtraction*, which reduces the size of each shared variable set.

4. Refinement 1: Shared Variable Set Merging

In this version, shared variable sets are represented as bit strings, so that all shared variable sets are the same size and proportional to the number of monitored variables. A new **merge** step is introduced to reduce the *number* of shared variable sets, as follows:

> *Shared variable sets are combined by a set union operation at time* t *iff their associated concurrency lists are equal.*

We note that a bitstring representation is convenient both for performing the comparison step described in Section 3 and for the set union operation used in this merge step.

With merging, all concurrency lists are unique, and so the number of shared variable sets is clearly bounded by the number of possible concurrency lists. While the space of possible concurrency lists is exponential, there is in fact a much better $O(\mathbf{N}^2)$ bound on the number of distinct concurrency lists, where **N** is the number of tasks. This result follows from Lemma 2:

Lemma 2. Consider a task T at time t and the sequential blocks $B_1, \ldots, B_n$ in T, such that B_1 is the first block executed, B_n is the current block executing, and B_{j+1} is executed after B_j. For $1 \leq k < n$, the concurrency list for block B_k is either equal to or a proper subset of the concurrency list for block B_{k+1}.

Proof Suppose not. There is an executing block B_j in another task that is not concurrent with B_{k+1}, but is concurrent with B_k. It must be the case that B_j is dependent on B_{k+1}, since B_j is executing and B_{k+1} is finished executing. (B_{k+1} cannot be B_n since B_n is concurrent with all other executing blocks.) If B_j is dependent on B_{k+1}, it must also be dependent on B_k.□

Since the size of the concurrency list associated with the currently executing block for each task T is **N**-1, the total number of distinct concurrency lists associated with blocks in T is **N**-1. It follows that the total number of concurrency lists at any time is bounded by $\mathbf{N}^2$. Therefore, for any program, regardless of the pattern of synchronous coordination operations, the amount of storage required for shared variable sets is bounded and depends only on the number of tasks and the number of monitored variables.

We make the following changes to the formulation in Section 3:

(a) A *shared variable set* is associated with one or more sequential blocks, and consists of two subsets: the set of all shared variables read in any of the associated blocks, and the set of all shared variables written in any of the associated blocks. A shared variable set associated with blocks $B_{x_1}, \ldots, B_{x_k}$ at time t is written as $S_{x_1,\ldots,x_k}(t)$.

(b) We add the following additional algorithm step:

(5) **Merge.** For any two complete shared variable sets $S_{x_1,\ldots,x_n}(t)$ and $S_{y_1,\ldots,y_m}(t)$ such that their associated concurrency lists $CL_{x_1,\ldots,x_n}(t)$ and $CL_{y_1,\ldots,y_m}(t)$ are equal, replace $S_{x_1,\ldots,x_n}(t)$ and $S_{y_1,\ldots,y_m}(t)$ by a new merged shared variable set $S_{x_1,\ldots,x_n,y_1,\ldots,y_m}(t)$. The merged set is formed by taking the union of the subsets of $S_{x_1,\ldots,x_n}(t)$ and $S_{y_1,\ldots,y_m}(t)$; that is, $S_{x_1,\ldots,x_n,y_1,\ldots,y_m}(t)$ consists of the set of variables read in any of $B_{x_1}, \ldots, B_{x_n}$ or $B_{y_1}, \ldots, B_{y_m}$ and the set of variables written in any of $B_{x_1}, \ldots, B_{x_n}$ or $B_{y_1}, \ldots, B_{y_m}$.

Merging is justified for the following reasons. Suppose two shared variable sets $S_x(t)$ and $S_y(t)$ have equal concurrency lists. If there is no merging, the concurrency list for these shared variable sets will remain equal through any subsequent operation executed in the program. This result follows from the update rule given in Section 3. Any shared variable set $S_z(t')$, $t' > t$, subsequently compared with $S_x(t')$ will also be compared with $S_y(t')$, and vice-versa. Therefore, any error detected without merging, will be detected with merging.

4.1. Applications

We return to Figure 5 which shows four tasks $T_1, \ldots T_4$ executing concurrently. The shared variable sets and their concurrency lists after each timestep are shown in Figure 6. Shared variable sets that are labeled by a ‘‘*’’ are incomplete. The concurrency list

$$\{T_{i_1}, T_{i_2}, \ldots, T_{i_r}\}$$

is abbreviated as:

$$\{i_1, i_2, , \ldots, i_r\}.$$

In this example, there are 4 incomplete shared variable sets before any coordinations. At time (1), two shared variable sets complete and are merged to form shared variable set $S_{1,4}$. At time (2), task T_3 is removed from concurrency list $CL_{1,4}$, because block B_9 is not concurrent with B_1 and B_4. Merged shared variable set $S_{5,8}$ is formed. At time (3), $S_{2,6}$ is a new merged shared variable set, and task T_1 is removed from concurrency list $CL_{5,8}$. The concurrency lists $CL_{1,4}$ and $CL_{5,8}$ are now equal, so there is an additional merging of shared variable sets $S_{1,4}$ and $S_{5,8}$ at this point. At time (4), the concurrency list of this merged shared variable set $S_{1,4,5,8}$ becomes empty, and so the shared variable set is deleted.

In general, assuming **N** tasks, every time a coordination operation occurs at time t between two tasks T_i and T_j, two shared variable sets $S_x(t)$ and $S_y(t)$ are completed. (Two new shared variable sets

$S_{x_{new}}(t)$ and $S_{y_{new}}(t)$ are also created, since tasks T_i and T_j continue to execute.) The concurrency lists $CL_x(t)$ and $CL_y(t)$ are of size **N**-2 and are equal:

$$CL_x(t) = CL_y(t) = \{T_k : k \neq i \text{ and } k \neq j\}.$$

Therefore, $S_x(t)$ and $S_y(t)$ are merged into $S_{x,y}(t)$. At least one merge operation is always performed at each timestep, so that the number of shared variable sets increases by at most one per coordination operation. Moreover, since each complete shared variable set is a merger of at least two shared variable sets, a more precise bound on the number of shared variable sets is $\mathbf{N}^2/2$.

In fact, the worst case coordination pattern that has been constructed has $\mathbf{N}^2/4+\mathbf{N}-1$ shared variable sets[2]; the construction for an execution with eight tasks is shown in Figure 7. In this figure, there are 15 coordinations. The values of concurrency lists at time $t=14$ appear over the horizontal lines representing synchronous coordination points. Each concurrency list belongs to the two blocks that executed the operation. Any subsequent operation at $t=16$ causes a shared variable set to be deleted or merged.

4.2. General Comments

While the worst case storage is quadradic, simulations with randomly generated coordination patterns show that this worst case is not easy to obtain. A simulation with 100 tasks synchronizing randomly 100,000 times produces a maximum number of 502 shared variable sets, which is achieved after 51,828 coordination operations. Testing the algorithm on real programs will give a better understanding of the storage requirements for typical programs.

Merging loses diagnostic precision. If an access anomaly is detected in a merged shared variable set, it is not necessarily obvious in which block one (or more) or the conflicting reference(s) occurred. As mentioned in Section 1.2, since one of the erroneous sequential blocks and the exact variable involved in the anomaly is revealed by the algorithm, subsequent monitoring can be performed to precisely pinpoint the anomaly.

The algorithm of Snir described in Section 1.2 is a special case of this refined version for programs with only a fork-join construct. (See Section 6 for a discurssion of fork-join update rules.) At each join operation, the child shared variable sets are merged with the parent shared variable set; the delete step is never applicable. Concurrency lists need not be kept explicitly in [Sni] because of the semantics of fork-join.

5. Refinement II: Shared Variable Set Subtraction

As an alternative compression technique, the size of individual shared variable sets can be reduced. Variables are removed from shared variable sets by a new **subtraction** step, as follows:

---

[2]This construction is due to Peter Frankl [Fr].

A monitored variable v is removed from the read (resp. write) subset of $S_y(t)$ *iff v is in the read (resp. write) subset of* $S_x(t)$ *and* $CL_y(t)$ *is a subset of* $CL_x(t)$.

We justify subtraction as follows. Suppose $CL_y(t)$ is a subset of $CL_x(t)$. Subsequently, any shared variable set $S_z(t')$, $t' > t$ compared with $S_y(t')$ must also be compared with $S_x(t')$. From the update rule in Section 3, CL_y remains a subset of CL_x after any subsequent coordination operation. Therefore, if v is a variable in the read (resp. write) subset of both $S_x(t')$ and $S_y(t')$, and if there is an access anomaly conflict involving v between $S_x(t')$ and $S_z(t')$, there is also an access anomaly between $S_y(t')$ and $S_z(t')$. After subtraction, the latter anomaly is detected.

Consider the shared variable sets $S_{x_1}, \ldots, S_{x_n}$ that are associated with a task T at time t. If the subtraction step is performed, it follows from Lemma 2 that v can belong to the read subset of only one shared variable set $S_{x_i}(t)$, $1 \le i \le n$ Therefore, the total space needed to store read information is bounded by $V \times \mathbf{N}$, where V is the number of variables being monitored.

Even less space is needed for write subsets— the storage for write information is bounded by V. More specifically, for each variable v, v belongs to the write subset of at most one shared variable set. This follows from the argument below.

Suppose z is written by block B_x in task T_i at time t, and suppose v is in the write subset of $S_y(t)$. If B_y is concurrent with B_x there is an access anomaly. If B_y is not concurrent with B_x, then $CL_y(t)$ is a subset of $CL_x(t)$. Therefore, v is removed from $S_y(t)$ by the subtraction step.

With the subtraction technique, a different implementation strategy is possible: shared variable set data are not kept per se, but rather information is distributed among the monitored variables. For each variable v, we associated a list of references to concurrency lists CL_x such that v is read in B_x. Because of subtraction, each list is bounded by **N**. We also associate with v a reference to concurrency list CL_x such that the most recent write operation was performed in B_x. Every time a monitored variable v is *read or written*, a test is made for a possible anomaly, and the information associated with v is updated. The **compare** step (1) in the algorithm in Section 3 is thereby eliminated. (Operations on Concurrency list described in steps (2)-(4) remain unchanged.)

In Figure 8, variable v is read in blocks B_1, B_2, and B_5 and written in block B_{11}. The second read of v in block B_2 supercedes the read of v in block B_1, so by subtraction, the concurrency list CL_1 is replaced by CL_2 in the read list of v. After the second synchronization between tasks T_1 and T_2, CL_5 is removed from the read list of v. Finally, at the write operation in B_{11}, T_4 is in the concurrency list $CL_2(t)$, so an anomaly will be detected.

While version II is $O(\mathbf{N})$ and version I is $O(\mathbf{N}^2)$ in space, version II may in fact be slower because it may be more costly to implement subtraction. Moreover, space needed for concurrency lists is not bounded. It is possible to combine version I and version II to obtain a hybrid algorithm in several ways:

(1) A **merge** step can be added to version II, whereby concurrency lists are combined according to the rule described in Section 4 (but no actual shared variable sets are maintained.) Instead of shared

variable sets, references to concurrency lists are associated with each monitored variable. A union-find approach [AHU] can then be applied to manage these reference lists for all monitored variables v.

(2) A **subtraction** step can be added to version I, whereby a shared variable set difference operation is performed after each merge operation. In this case, the shared variable set representation must be adapted to exploit sparse sets.

Any combination of these approaches result in more storage compression than either approach alone.

6. Adding Fork-Join Operations

We next consider programs with both synchronous coordination and nested fork-join operations. To extend the algorithm presented in Section 3, we specify concurrency list update rules for these operations:

Fork. Suppose task T_i executes a fork, generating new tasks $T_{i_1}, \ldots, T_{i_n}$ at time t.

(3b) **Update**. For each complete shared variable set $S_z(t)$:

(i) if $CL_z(t-1)$ includes T_i, $CL_z(t) = CL_z(t-1) - \{T_i\} \cup \{T_{i_1}, \ldots, T_{i_n}\}$;

(ii) otherwise, $CL_z(t) = CL_z(t-1)$.

Since concurrency lists contain only tasks that are currently running, and since task T_i is blocked until the subsequent join operation, this rule replaces references to the task T_i by references to the forked children tasks. The parent T_i may possibly be added again at the corresponding join operation.

Join. The join operation is similar to a multiway synchronous coordination point. Therefore, if the tasks $T_{i_1}, \ldots, T_{i_n}$ execute a join together, the update operation is similar to the synchronous coordination rule:

(3c) **Update**. For each complete shared variable set $S_z(t)$:

(i) If *all* of $T_{i_1}, \ldots, T_{i_n}$ are in the concurrency list $CL_z(t-1)$,

$$CL_z(t) = CL_z(t-1) - \{T_{i_1}, \ldots, T_{i_n}\} \cup \{T_i\};$$

(ii) otherwise, $CL_z(t) = CL_z(t-1) - \{T_{i_1}, \ldots, T_{i_n}\}$.

These rules are illustrated in the next Section.

6.1. Fork-Join Example

Consider the program graph in Figure 9(a). The concurrency lists at each parallel operation step are shown in Figure 9(b). Initially, there are four incomplete shared variable sets. At time (1), three new tasks are created, and shared variable set S_1 completes. At time (2) after the first coordination between tasks T_6 and T_3, concurrency lists for S_4 and S_{10} are equal and so these states are merged. Task T_3 is removed from the concurrency list of shared variable set S_1, as well as from the concurrency list $CL_{4,10}$.

After the second coordination between tasks T_6 and T_2, T_2 is removed from the concurrency lists for shared variable sets S_1 and $S_{4,10}$. Finally, at the first join point, since tasks T_5, T_6, and T_7 terminate, they are removed from all concurrency lists. Shared variable sets S_1 and $S_{4,10}$ have equal concurrency lists and are merged. The final result is shown at time (4).

6.2. Bounds on the Number of Distinct Concurrency Lists with Nested Fork-Joins

Given a program execution instance P with both synchronous coordination operations and nested fork-join operations, the number of tasks executing varies over time. We will say that two tasks T_i and T_j are *concurrently executable* if for any sequential block B_x in T_i and B_y in T_j, B_x and B_y are concurrent. A *covering* of P is an assignment AS, the domain of which are the tasks in P, such that if T_i and T_j are concurrently executable, then $AS(T_i)$ and $AS(T_j)$ are not equal. In Figure 1, there is a covering of cardinality 4, and in Figure 8(a), there is a covering of cardinality 6. Let **M** be the cardinality of a minimum covering for P. Then the upper bound on the number of shared variable sets that can be obtained at any time is $\mathbf{M}^2/2$. The proof of this bound is given in the Appendix, and it is obtained using Lemma 2. Below, we remark on how big **M** can be.

Let N be the maximum concurrency of P, where the maximum concurrency is defined as the size of the largest set of tasks that are concurrently executable. It is clear that $\mathbf{M} \geq \mathrm{N}$.

Let W be the maximum width of P, where the maximum width of an execution P is defined as follows:

(i) The maximum width of a task T that does not executed any fork operations is 1.

(ii) The maximum width of a fork operation F is the sum of the maximum widths of all tasks created by the fork.

(iii) For a task T that is a parent task, let F be the fork operation executed by T that has the largest maximum width. The maximum width of T is the maximum width of F.

Finally, the maximum width W of P is the sum of the maximum widths of all tasks initially executing in P. It is clear that W is greater than or equal to the maximum concurrency M. (In Figure 8(a), the maximum concurrency equals the maximum width.) Furthermore, it is not hard to generate a covering for P of size W. Therefore, $\mathrm{N} \leq \mathbf{M} \leq \mathrm{W}$. We conjecture, although no proof is currently provided, that in fact $\mathbf{M} = \mathrm{N}$.

7. Applying The Algorithm To Real Language Constructs

While the programming model that we have described is somewhat simplified compared with real parallel programming languages, the algorithm presented can easily be adapted to handle many common parallel programming language synchronization and coordination primitives. We consider several language primitives below.

7.1. Ada Rendezvous

The Ada rendezvous [DOD] is a pairwise synchronous coordination primitive which is asymmetric. When a rendezvous occurs, the caller waits while the called task executes the body of the accept statement. When the called task finishes the accept statement, the caller continues. The called task can engage in other rendezvous before returning.

To use our algorithm, a rendezvous is modeled as two coordinations between the caller task T_i and the called task T_j. The beginning of the rendezvous occurs at time t, and the end of the rendezvous occurs at later time t'. There is no block B_x executed in the caller T_i between t and t', so the associated shared variable set is empty. Figure 9 illustrates a program execution instance with five tasks and three rendezvous. First, tasks T_1 and T_2 rendezvous, where T_2 is the caller. Then task T_3 and T_4 rendezvous. Before T_4 returns, T_4 and T_5 rendezvous, where T_4 is the caller. This execution is treated as a program with six synchronous coordination operations.

7.2. Barrier Synchronization

A barrier synchronization is a multiway synchronous coordination point. More specifically, if a task reaches an **N**-way barrier synchronization point during execution, it waits until **N**-1 other tasks also reach the barrier. For the anomaly detection algorithm, an **N**-way barrier synchronization is easily handled as $2\times(\mathbf{N}-1)$ pairwise synchronous coordination operations. More specifically, tasks $T_1, \ldots, T_{\mathbf{N}}$ executing a barrier is treated as the following sequence of synchronous coordinations among the tasks:

T_1 and T_2
T_2 and T_3
.
.
$T_{\mathbf{N}} - 1$ and $T_{\mathbf{N}}$
$T_{\mathbf{N}} - 1$ and $T_{\mathbf{N}} - 2$
.
.
T_3 and T_2
T_2 and T_1.

The barrier synchronization update rule is similar to the join operation rule.

7.3. Asynchronous Coordination

Many coordination primitives, such as **doacross** coordination, message send/receive, and locking operations, cannot be modeled by synchronous coordination operations, because they are inherently asynchronous. An *asynchronous pairwise coordination* is an asymmetric coordination between two tasks: a sender and a receiver. If the receiver reaches the coordination point first, the receiver waits until the sender reaches the coordination point. However, if the sender arrives first, the sender does not wait for the receiver. A message is posted to notify the receiver to proceed executing.

Asynchronous coordination is illustrated in Figure 10(a). Arrows indicate asynchronous coordination operations, where the arrow originates at the sender. Unlike the synchronous case, the block executing in the sender after a coordination occurs is concurrent with the block executing in the receiver before the coordination occurs. Figure 10(b) indicates which blocks in Figure 10(a) are concurrent with each other.

It should be possible to extend the algorithm in Section 3 to handle asynchronous coordination, although this is beyond the scope of this paper. With an extended algorithm, a variety of asynchronous programming language primitives can be handled. For example, for locking operations, the task executing the **unlock** operation is considered to be the sender, and the task executing the **lock** operation is the receiver. Other primitives are treated in a similar fashion.

8. Conclusion

The anomaly detection algorithm presented provides the basis for a monitoring tool that automates an important part of the iterative debugging process. When integrated with other parallel debugging tools, such as a trace-and-replay system (e.g. [Mil] [LeB]) which provides reproducibility, and static analysis which reduces the number of variables that need to be monitored, the effectiveness of the method will be further enhanced. Alternatively, this algorithm can be used to analyize shared variable trace data in a post-mortem phase. For certain real-time programs, it is not be possible to monitor on-the-fly, and the techniques described are also beneficial for processing large traces subsequent to execution.

Experience with these methods is required to determine appropriate data structures, additional algorithm refinement, and other profitable compression techniques. The design and engineering of practical monitoring tools for finding parallel bugs in shared memory programs is our eventual goal.

9. References

[AHU] Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., 1974.

[All] T.R. Allen, D.A. Padua, ‘‘Debugging Fortran on a Shared Memory Machine’’, *Proc. International Conf. on Parallel Processing*, August 1987, pp. 721-727.

[App] D. Applebe, C. McDowell, ‘‘Developing Multitasking Applications Programs’’, *Proc. Hawaii International Conference on System Sciences*, Jan. 1988, pp. 94-102.

[Bau] G.M. Baudet, ‘‘Asynchronous Iterative Methods for Multiprocessors’’, *Journal of the ACM*, Vol. 25, No. 2, April 1978, pp. 226-244.

[Bur] M. Burke, R. Cytron, ‘‘Interprocedural Dependence Analysis and Parallelization’’, *Proc. SIGPLAN Symposium on Compiler Construction*, Vol. 21, No. 7, July 1986, pp. 162-175.

[Car] A. Carle, K.D. Cooper, R.T. Hood, K. Kennedy, L. Torczon, S.K. Warren, ‘‘A Practical Environment for Scientific Programming’’, *Computer*, Vol 20, No. 11, Nov. 1087, pp. 75-89.

[DOD] Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815 A, 1983.

[Fr] P. Frankl, Private Communication, Sept, 1988.

[LeB] T.J. LeBlanc, J.M. Mellor-Crummey, ‘‘Debugging Parallel Programs with Instant Replay’’, *IEEE Trans. on Computers*, Vol. C-36, No. 4, April 1987, pp. 471-482.

[Mil] B.P. Miller, J.D. Choi, ‘‘A Mechanism for Efficient Debugging of Parallel Programs’’, *SIGPLAN Conference on Compiler Construction*, Atlanta, June 1988.

[Nud] I. Nudler, L. Rudolph, ‘‘Tools for the Efficient Development of Efficient Parallel Programs’’, *First Israeli Conference on Computer Systems Engineering*, May 1986.

[Sni] M. Snir, Private Communications, March 1988.

[Tay] R.N. Taylor, ‘‘A General-purpose Algorithm for Analyzing Concurrent Programs’’, *CACM*, Vol. 26, No. 5, May 1983, pp. 362-376.

APPENDIX

We prove an upper bound on the number of shared variable sets obtainable for the general case of programs with both fork-join and synchronous coordination operations.

Lemma 3. Let P be an execution instance, and let $\mathbf{M}$ be the cardinality of a minimum covering for P. The number of shared variable sets that can be obtained at any time is bounded by $\mathbf{M}^2/2$.

Proof. Consider two program execution instances P and Q. We will say that Q *simulates* P iff

(i) for every operation O in P, there are one (or more) operations in Q corresponding to O; and

(ii) if $t_1, ..., t_m$ is a total ordering of the operations in P consistent with the partial ordering of operations in P and $CL^P(t)$ is a concurrency list in P at time t, then for the corresponding total ordering of operations in Q, there is a unique concurrency list $CL^Q(t)$ in Q corresponding to $CL^P(t)$.

To prove Lemma 3, we show that for an execution instance P that includes fork, join, and synchronous coordination operations, there is an execution instance Q that simulates P, such that Q consists of $\mathbf{M}$ tasks executing only synchronous coordination operations (and no nested fork-join operations). It then follows from Lemma 2 that the number of shared variable sets in P at any time is bounded by $\mathbf{M}^2/2$.

Let $R_1, \ldots, R_{\mathbf{M}}$ be a set of tasks not in P, and let AS be an assignment of tasks in P to $R_1, \ldots, R_{\mathbf{M}}$, such that AS is a covering for P. The simulation, which contains only synchronous coordinations, is performed as follows:

(i) For each synchronous coordination operation between T_{i_1} and T_{i_2} in P, the corresponding operation in Q is a synchronous coordination operation between R_{j_1} and R_{j_2}, where $R_{j_1} = AS(T_{i_1})$ and $R_{j_2} = AS(T_{i_2})$.

(ii) For each fork operation, such that T_i creates tasks $T_{i_1}, \ldots, T_{i_k}$, the corresponding operations in Q is the set of synchronous coordinations among the assigned tasks $AS(T_i)$, $AS(T_{i_1})$, ..., $AS(T_{i_k})$ that implements a barrier synchronization among these tasks (see Section 6.2).

(iii) For each join operation executed by tasks $T_{i_1}, \ldots, T_{i_k}$, where T_i is the task executing after the join, the corresponding operations in Q are again the set of synchronous coordination operations among the assigned tasks $AS(T_i)$, $AS(T_{i_1})$, ..., $AS(T_{i_k})$ implementing a barrier synchronization.

Let $t_1, \ldots, t_m$ be a total ordering of operations in P consistent with the partial ordering of operations in P. (For simplicity of exposition, if an operation performed in P at time t corresponds to a set of operations in Q, the set of operations in Q are considered all to be performed at time t.)

Claim 1. For each concurrency list $CL^P(t)$, there is a corresponding concurrency list $CL^Q(t)$ such that for each T in $CL^P(t)$, $AS(T)$ is in $CL^Q(t)$.

At time $t=0$, the claim clearly is true.

Assuming the hypothesis is true at time t, we consider concurrency lists at time $t+1$. For each new concurrency list $CL^P(t+1)$, associated with a new block executing in task T, we let the corresponding concurrency list $CL^Q(t+1)$ be the concurrency list associated the new block in task $AS(T)$.

Suppose $CL^P(t)$ is associated with a complete shared variable set $S_x(t)$, and $CL^Q(t)$ is associated with $S_y(t)$ in Q. Concurrency lists $CL^P(t)$ and $CL^Q(t)$ are updated according to the operation performed at $t+1$:

Case 1. The operation at time $t+1$ is a synchronous coordination. Applying the update rule to both concurrency lists $CL^P(t)$ and $CL^Q(t)$, it is clear that T is removed from $CL^P(t)$ iff $AS(T)$ is removed from $CL^Q(t)$. Therefore, the claim holds at $t+1$.

Case 2. The operation at time $t+1$ is a fork executed by task T_i, creating k tasks. Let T_j be a task created by the fork. If T_i is in $CL^P(t)$, then by the update rule for fork, T_j is in $CL^P(t+1)$. To prove the claim, we must show that $AS(T_j)$ is in $CL^Q(t)$. Because T_i is in $CL^P(t)$, the block B_z executing initially in T_j, is concurrent with block B_x associated with CL^P. For any task T that exists between the time that B_x finishes and time $t+1$, $AS(T) \neq AS(T_j)$. Otherwise, the assignment AS is not a covering for P. Therefore, $AS(T_j)$ cannot engage in any operations in Q between the time that B_y associated with $CL^Q(t)$ finishes and $t+1$, so that $AS(T_j)$ must be in the concurrency list $CL^Q(t+1)$.

Case 3. The operation at time $t+1$ is a join. If T_i is added to $CL_x^P(t+1)$ by the update rule, then the blocks executing the join operation are all concurrent with B_x associated with $CL^P(t+1)$. Therefore, $AS(T_i)$ cannot be assigned to any task T existing between the time that B_x finishes and $t+1$. It follows that $AS(T_i)$ must be in the concurrency list of $CL^Q(t+1)$.□

To complete the proof, we must show that the correspondence of concurrency lists in P to concurrency lists in Q is 1-1. More precisely, if two concurrency lists $CL_{y_1}^Q(t)$ and $CL_{y_2}^Q(t)$ are equal, then the corresponding concurrency lists $CL_{x_1}^P(t)$ and $CL_{x_2}^P(t)$ are also equal, so that no two states in Q are merged unless there is a corresponding merger in P. This result follows from Claim 2 below.

Claim 2. Let $CL^Q(t)$ be the concurrency list associated with $CL^P(t)$. If R in $CL^Q(t)$ is not assigned to any task T in $CL^P(t)$, then R is not assigned to any task that is currently executing at time t.

At time $t=0$, the claim clearly is true.

Assuming the hypothesis is true at time t, we consider concurrency lists at time $t+1$. The claim clearly holds for new concurrency lists associated with incomplete shared variable sets. Suppose $CL^P(t)$ and $CL^Q(t)$ are complete. By inductive hypothesis, any task R in $CL^Q(t)$ that is not assigned to a task T in $CL^P(t)$ is unassigned at time t.

Case 1. The operation at time $t+1$ is a synchronous coordination between T_i and T_j in P. In the simulation, there is a synchronous coordination between $AS(T_i)$ and $AS(T_j)$. Applying the update rule to both executions, $AS(T_i)$ is in $CL^Q(t+1)$ iff T_i is in $CL^P(t+1)$. Similarly for $AS(T_j)$ and T_j. Therefore, the claim holds.

Case 2. The operation at time $t+1$ is a fork in which T_i creates tasks k in P, and in Q there is a barrier synchronization sequence among $k+1$ tasks to the T_i and the k forked tasks. Consider one of the forked tasks T_j. If T_i is not in $CL^P(t)$, then after the barrier synchronization operations, $AS(T_j)$ is not in $CL^Q(t+1)$. If T_i is in $CL^P(t)$, then T_j is in $CL^P(t+1)$. Therefore, if $CL^Q(t+1)$ contains a task R that is assigned to a task T, T is also in $CL^P(t+1)$.

Case 3. The operation at time $t+1$ is a join. If a task T_j executes the join operation and is not in $CL^P(t+1)$, T_j is not a currently executing at time $t+1$. If T_i begins executing after the join, and is not in $CL^P(t+1)$, then for some T_j that executes the join, T_j is not in $CL^P(t)$, and $AS(T_j)$ is not in $CL^Q(t)$. After the barrier synchronization operations corresponding to the join, $AS(T_i)$ is not in $CL^Q(t+1)$. Therefore, $AS(T_i)$ is not in $CL^Q(t+1)$ unless T_i is in $CL^P(t+1)$.□

Now suppose that concurrency lists $CL^Q_{y_1}(t)$ and $CL^Q_{y_2}(t)$ are equal, but $CL^P_{x_1}(t)$ and $CL^P_{x_2}(t)$ are not. Then there must be a task R that is in $CL^Q_{y_1}(t)$, and assigned to a task in $CL^P_{x_1}(t)$, (or vice versa), but is not assigned to any task currently executing at time t. But this cannot happen, so that the correspondence must be 1-1.□