An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection

Anne Dinning [*]
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York, NY 10012
(dinning@cs.nyu.edu)

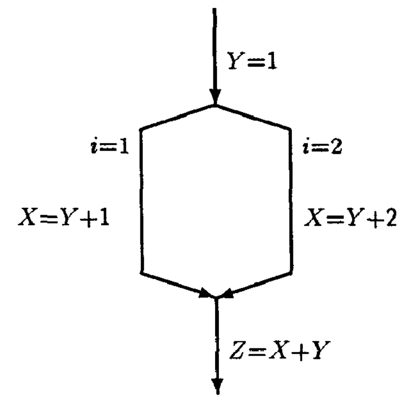
Edith Schonberg [†]
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
(schnbrg@ibm.com)

Abstract

One of the major disadvantages of parallel programming with shared memory is the nondeterministic behavior caused by uncoordinated access to shared variables, known as *access anomalies.* Monitoring program execution to detect access anomalies is a promising and relatively unexplored approach to this problem. We present a new algorithm, referred to as *task recycling*, for detecting anomalies, and compare it to an existing algorithm. Empirical results indicate several significant conclusions: (i) While space requirements are bounded by $O(T \times V)$, where T is the maximum number of threads that may potentially execute in parallel and V is the number of variable monitored, for typical programs space requirements are on average $O(V)$. (ii) Task recycling is more efficient in terms of space requirements and often in performance. (iii) The general approach of monitoring to detect access anomalies is practical.

1 Introduction

Erroneous non-deterministic behavior in shared memory parallel programs is often the result of *access anomalies.* An access anomaly occurs when two concurrent execution threads access the same memory location in an "unsafe" manner: more specifically, when either two concurrent threads both write or one reads and one writes a shared memory location without coordinating these accesses. The program segment in Figure 1 illustrates an access anomaly. The *doall* construct creates two parallel threads that both write the variable X. The



```
Y = 1
doall i = 1 to 2
    X = Y + i
endall
Z = X+ Y
```

Figure 1: Simple Program With an Access Anomaly

value of X subsequently used to calculate Z depends on whether iterate $i = 1$ or $i = 2$ updates X last. Therefore the two writes to X are anomalous. Variable Y is also accessed in both concurrent threads $i = 1$ and $i = 2$; however, this is not an anomaly because both accesses are reads. Similarly, the assignment to Y in the first statement does not cause an anomaly because it is always performed before either read. While certain *internal non-determinism* [5] does not affect the outcome of programs and is safe (e.g. the non-deterministic order in which a lock is granted), access anomalies often introduce *external non-determinism* that modifies execution results. Although some programs are designed to contain access anomalies, this type of external non-determinism is usually unintentional. Traditional debugging techniques are of limited use in finding errors

[*]This research was supported in part by an IBM Graduate Fellowship.

[†]This research was supported in part by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-FG02-88ER25052.

caused by access anomalies because such bugs are sensitive to timing.

Detecting access anomalies by *monitoring program execution* is proposed in [12, 10, 11, 4]. Using this approach, access anomaly are detected much in the same manner that array subscript range checking is performed. When a variable is accessed during execution, an immediate check is make to see whether the access conflicts with a previous access, in which case the error is reported. This approach may be used in conjunction with static analysis [5, 2] and is much more efficient that trace-based post-mortem methods [1, 3, 8].

Thus paper describes a new on-the-fly access anomaly algorithm called *task recycling*[1] and compares it to an existing method, *English-Hebrew labeling* [9, 10]. These methods were implemented in the parallel Fortran compiler on the NYU Ultracomputer [6]. Both algorithms record read and write events in *access histories* that are associated with monitored variables. While in the worst case the length of an access history depends on the amount of parallelism in the program, experimental results show that access histories in fact rarely have more than a few entries[2].

The two algorithms differ in the methods used to determine whether different threads are concurrent. The experiment shows that task recycling is less efficient for maintaining concurrency information at parallel operations than English-Hebrew labeling. However, it is more efficient when checking for anomalies at read and write events and requires considerably less storage. Since read and write events are generally more frequent than parallel operations, we conclude that task recycling is an improvement over English-Hebrew labeling both in performance and space.

Finally, the experiment measures the actual cost of monitoring. For the benchmark programs, monitoring entails a 3-fold to 6-fold slowdown. Although this cost is high, it is not unreasonable during a debugging phase of program development. Moreover, this is the cost of monitoring *every* shared variable reference. In general, static analysis or user directives may be used to limit the number of variables monitored, and thus greatly reduce the cost. Section 2 states the anomaly detection problem in terms of the *partial order execution graph* and describes access histories. The two algorithms for determining concurrency relationships are presented in Sections 3 and 4. The experiment is presented in Section 5, and the results summarized in Section 6.

[1]This algorithm relies of version numbers and parent vectors, which are similarly used in [3] for post-mortem analysis.

[2]Access history based methods are therefore superior to the method described in [11] in which the size of the access data depends on the parallelism

2 Definitions and Framework

The Ultracomputer parallel Fortran compiler provides the nestable *doall-endall* construct as its primary concurrency primitive. Concurrent threads are created by a *doall* operation and terminated by a corresponding *endall* operation. Coordination primitives provided as library routines may be classified as synchronous or asynchronous[3]. If two threads coordinate *synchronously* (e.g. via a barrier operation) neither thread may execute beyond the coordination point until both have reached it. If two threads coordinate *asynchronously* (e.g. via a signal operation) the receiver may not execute beyond the coordination point until the sender has reached it; however, the sender may proceed immediately. A *block* is an instruction sequence, executed by a single thread, that does not include doall, endall or coordination operations. Hence a thread is made up of a sequence of blocks that reflect its interaction with other threads.

The concurrency relationship among blocks is represented by a directed *partial order execution graph, (POEG)*. A POEG captures Lamport's "happens before" relation and imposes a partial order on the set of blocks that make up an execution instance [7]. An edge is either a block or coordination edge; a coordination edge connects vertices of two coordinating blocks. A vertex corresponds to a doall, endall or coordination operation. Concurrency determination is independent of the number and relative execution speeds of processors executing the program. A block b_j is an *ancestor* of a block b_i if there is a path from b_j to b_i in the graph (equivalently, b_i is a *descendant* of b_j). Two blocks are *concurrent* iff neither is an ancestor of the other. We define the *maximum concurrency* of a POEG to be the maximum number of mutually concurrent blocks. To illustrate these definitions, consider the POEG in Figure 2. Block b_3 is concurrent with blocks b_4, b_5 and b_2; it is not concurrent with block b_0 and b_1 (which are ancestors) or with blocks b_6 - b_{13} (which are descendants). The maximum concurrency of the graph is four.

In access history based algorithms, each block has an associated label. The *access history* for a variable X is a set of labels of the blocks which have read and written X. Every time variable X is read or written, the access history is examined to determine whether the current event conflicts with a previous one. When block b reads X, it must be determined whether b is concurrent with the writers in the access history for X. When block b writes X, tests are performed to determine whether b is concurrent with any of the blocks in the access history of X. Therefore, the efficiency of an access anomaly al-

[3]We assume that all interthread coordination is explicit; we do not attempt to solve the problem of automatically detecting coordination

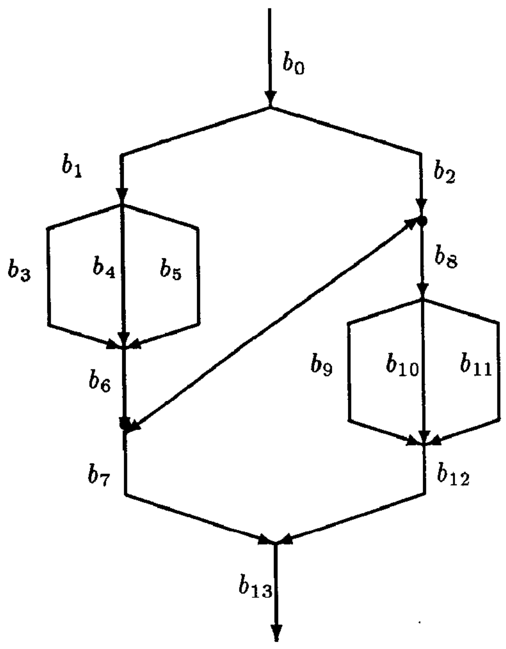


Figure 2: Partial Order Execution Graph

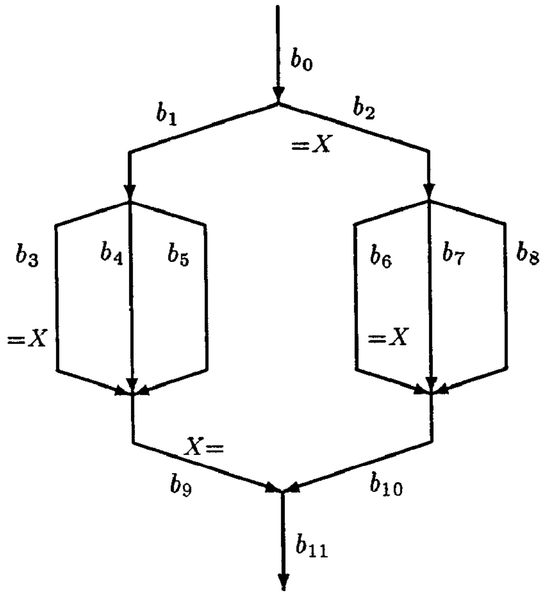


Figure 3: A Partial Order Execution Graph with Access Anomalies

gorithm depends primarily on how quickly the test of concurrency can be made and how many entries are in the access history. We next show how access histories can be compressed.

Consider the accesses to variable X in the POEG in Figure 3. The write of X in b_9 conflicts with the reads of X in b_2 and b_6. Suppose X is read in b_2, b_6, and b_3 in that order, and then written in b_9. After the first read, the access history of X contains b_2. When X is next read in b_6, b_6 is added to the access history and b_2 is deleted. The b_2 read event is no longer needed: any subsequent write event that conflicts with b_6 also conflicts with b_2. On the other hand, when b_3 is added to the access history, b_6 cannot be deleted. Otherwise, no anomaly can be detected when X is written in b_9, since this write does not conflict with the read in b_3.

More generally, a block b in the reader set of a variable X can be deleted when a descendant of b accesses X. Thus, the reader set of an access history contains two blocks only if they are concurrent. (On the other hand, since two concurrent writes always conflict, there is at most one writer in an access history.) By deleting obsolete entries, the size of an access history is therefore bounded by the maximum concurrency of the POEG. We show in Section 5 that this bound is, in fact, much too pessimistic. Although compaction reduces the size of the reader and writer sets, certain information may be lost. In particular, if there are multiple anomalies involving the same variable some of them may not be reported. However, at least one anomaly is guaranteed to be reported for every variable which is accessed in an "unsafe" manner.

3 English-Hebrew Labeling

In the English-Hebrew labeling algorithm, the structure of the POEG - and hence, the concurrency relationship among blocks - is encoded in *tags* [10, 9] associated with blocks. A tag consists of a pair of labels: an ***English label E*** and a ***Hebrew label H.*** Conceptually, the English label is produced by performing a left-to-right preorder numbering of the POEG; each block is assigned a number less than all of the numbers assigned to its children and its siblings to the right. However, since this label must be generated on-line, a complete traversal of the execution graph cannot be performed. Therefore, a label is a string of numbers and labels are ***lexicographically*** ordered. The children blocks $c_0 \ldots c_m$ of a doall vertex with parent block p are assigned English labels:

doall: $E(tag_{c_i}) \leftarrow E(tag_p) \mid i$

where | is the append operation. Similarly, a block c created in a coordination operation which has parent p is assigned an English label:

coordination: $E(tag_c) \leftarrow E(tag_p) \mid 1$

The child block c of an endall vertex with parents $p_0 \ldots p_m$ is assigned English label:

endall: $E(tag_c) \leftarrow max(E(tag_{p_i}))$

The Hebrew labels are created symmetricly for a right-to-left ordering; the only difference is that the i^{th} child of a doall operation has $f+1-i$ as the last entry in its Hebrew tag. As specified above, the length of the labels increase with the number of doall and coordination operations. However, an additional heuristic described in [10] bounds the label length to the level of nesting.

Two tags tag_i and tag_j are *unordered* iff either of the following conditions is met :

$E(tag_i) < E(tag_j)$ and $H(tag_i) > H(tag_j)$ or
$E(tag_i) > E(tag_j)$ and $H(tag_i) < H(tag_j)$

English-Hebrew tags only encode the ordering due to doall and endall operations; if two blocks are ordered in the POEG because of explicit coordination, their tags are unordered. Therefore, an additional mechanism is needed to record execution orderings imposed by coordination. To this end, a *coordination list* is associated with each executing block b and consists of the tags of the ancestors of b such that all tags are unordered. The test for concurrency between two blocks b and c requires determining if tag_b or any of the tags in $list_b$ are ordered with tag_c. Thus, the total amount of work is bounded by the size of the coordination lists times the length of the tags.

The length of a coordination list is bounded by the maximum concurrency of the POEG. When a block terminated, its coordination list is deleted so that the total number of coordination lists is also bounded by the maximum concurrency of the POEG. If tags are stored directly in the access history, storage can potentially be large and there is a larger constant overhead because tags are variable length. On the other hand, if tags are stored indirectly (i.e. if access histories consist of pointers to tags), tags must be saved throughout the execution requiring storage proportional to the number of blocks.

Figure 4 illustrates the use of English-Hebrew labels for a POEG. The coordination list of block 1131, 1231 contains the tag 121, 113 because of the coordination edge in the POEG. Likewise, blocks 1211, 1131 and 123, 113 have coordination lists containing the tag 113, 123. However, block 123, 123 does not have a coordination list because all of the tags in the coordination lists of its parent blocks are ordered with 123,123. We can determine that blocks 111, 123 and 12, 11 are concurrent because their tags are unordered; namely, the first condition is satisfied: 111 < 12 and 123 > 11. However, blocks 1131,1231 and 12,11 are not concurrent because an entry in the concurrency list of 1131,1231 - 121, 113 - is ordered with 12, 11.

4 Task Recycling Algorithm

Task Recycling reduces the cost of testing whether two blocks are concurrent to an array reference while increasing the expense of concurrency information maintenance. Instead of a tag, each block has a unique *task identifier*, which consists of a *task* and a *version number*. Tasks can be recycled; that is, more than one block

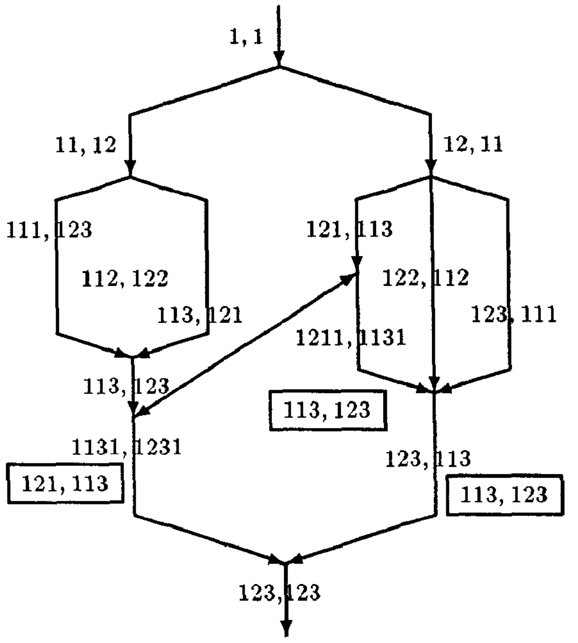


Figure 4: English-Hebrew Labeling

can be assigned to the same task at different times in the program execution. The version number of a task identifier is used to distinguish among different blocks assigned to the same task; every time a block is assigned to a task t, the associated version number v is incremented. The label storage problem is therefore addressed because the number of labels is bounded and they are easy to store directly.

Concurrency information is maintained in a *parent vector* which is associated with each currently executing block[4]. The number of entries in a parent vector is equal to the number of tasks. The t^{th} entry in the parent vector for block b contains the largest version number associated with those ancestors of b which were assigned to task t. New parent vectors are formed from those vectors associated with parent blocks. In particular, when blocks $p_1 \ldots p_m$ with task identifiers $t_{1v_1} \ldots t_{mv_m}$ create a new block c the parent vector of c is initialized as follows:

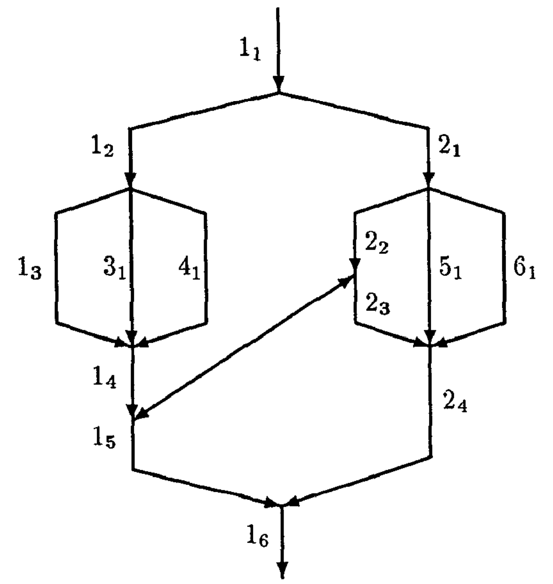
```
for $i = 1$ to $T$ do
    if $\exists_{t_j \in \{t_1 \ldots t_m\}} : i = t_j$ then
        $parent_c[i] \leftarrow v_j$
    else $parent_c[i] \leftarrow \max(parent_{p_1}[i], \ldots, parent_{p_m}[i])$
endfor
```

A block b is concurrent with a block with task identifier t_v iff $parent_b[t] < v$. Thus, there is a constant cost (an array access) for checking whether two blocks are con-

[4]Parent vectors correspond to *before* vectors in [3, 8]. However, because we monitor on-line, we do not need the corresponding *after* vectors used for post-mortem trace analysis.

current. (Coordination lists, on the other hand, must be searched linearly). As with English-Hebrew labels, only currently executing blocks need parent vectors; storing a task identifier in an access history is sufficient to determine whether two blocks are concurrent. Once a block terminates, its parent vector is discarded.

Figure 5 shows a task assignment and parent vectors for a POEG. In this example, block 1_3 is concurrent



Task Ids	*Parent Vector*
1_1	[0,0,0,0,0,0]
$1_2, 2_1$	[1,0,0,0,0,0]
$1_3, 3_1, 4_1$	[2,0,0,0,0,0]
$2_2, 5_1, 6_1$	[1,1,0,0,0,0]
1_4	[3,0,1,1,0,0]
$1_5, 2_3$	[4,2,1,1,0,0]
2_4	[4,3,1,1,1,1]
1_6	[5,4,1,1,1,1]

Figure 5: Task Recycling Assignment

with block 2_1 since the second entry in the parent vector of 1_3 - [2,0,0,0,0,0] - is less than 1. However, block 1_5 is not concurrent with 2_1 because the second entry in its parent vector - [4,2,1,1,0,0] - is greater than 1.

A block b is *validly* assigned to a task t iff it is not concurrent with any other block previously assigned to t. The goal of task assignment is to minimize the number of tasks used while maintaining a valid task assignment since this determines the length of parent vector. Our task assignment algorithm - from which this method derives its name - is based on a "most recently used" (MRU) heuristic and yields a good task assignment; there is no on-line task assignment algorithm which guarantees an optimal assignment [4].

To implement the MRU algorithm, a *free task dag* is maintained. There is a vertex in the dag associated with every doall, endall and coordination operation, and every currently executing block. (These latter vertices are always leaf vertices). The direction of edges in the dag is the reverse of the execution flow. A set of free tasks, i.e. those tasks that are available for recycling, is associated with each vertex. When a block b terminates at a doall or endall operation O, it adds its task to the free task list of the vertex v_O associated with O. When a block terminates in a coordination operation, its task is added to v_O and the children its associated block v_b are added to the child list of v_O. When a block b is created after operation O, an edge is added from the vertex v_b associated with b to vertex v_O. To assign a task to a block b, a topological traversal of the dag is performed starting at v_b until a vertex with a non-empty free task set is found.

Two optimizations minimize the cost of this topological traversal. Whenever a free task set of a vertex v becomes empty, the dag is *collapsed* by deleting v from the dag and adding edges from the parent vertices of v to children vertices of v. Therefore, every vertex in the free task dag has at least one free task associated with it. Whenever a vertex v has a single parent vertex p, v is *subsumed* by p by adding the free task set of v to the free task set of p, deleting v and adjusting the edges as appropriate.

Although the cost of performing task assignment is potentially quite high, for a large class of programs it is very limited. In particular, for programs with no nested parallelism and programs with nested parallelism but no coordination a constant amount of work is performed in assigning a task to a block.

5 Empirical Results

The preceding section presents two algorithms for detecting access anomalies in parallel programs. In general, English-Hebrew labeling will perform better than task recycling for programs with little coordination and frequent fine-grained doall operations because of the relatively high cost of maintaining parent vectors and task assignment. However, task recycling is clearly more efficient than English-Hebrew labeling in one way: the cost per variable access is less. Because this is frequently the most common operation, this is an important attribute of task recycling. Additionally, there is almost always a large storage penalty for saving tags in the English-Hebrew labeling algorithm. Moreover, if there is frequent coordination, the cost of maintaining coordination lists approaches the cost of maintaining parent vectors, so that any advantage of English-Hebrew labeling is diminished.

General discussions of algorithm design, however, give incomplete insight into the costs encountered when detecting anomalies in actual parallel programs. In order to better evaluate the two algorithms - as well as to gain insight into actual concurrency structure and memory access patterns - we monitored several benchmark scientific parallel programs:

Triso	- Solves a sparse triangular linear system of equations using wavefront parallelism
Finite	- Solves a linear system using finite element methods
Simple	- Solves partial differential equations for hydrodynamics and heat conduction
Polymer	- Performs molecular dynamic calculations of polymer systems

Our first goal in performing these benchmarks is to obtain representative values for parameters that measure concurrency structure and shared variable access patterns for parallel program. This gives us a general idea about the applicability of various monitoring techniques. Our second goal is to measure the actual performance impact of monitoring parallel programs. English-Hebrew and task recycling are compared on both their execution time and space requirements. The implementation and benchmark results are discussed in the following sections.

5.1 Implementation

The two monitoring algorithms are implemented for the Ultracomputer parallel Fortran compiler. The access anomaly detection algorithms are implemented by two library packages mostly written in C. The routines for maintaining access histories are written in assembler for reasons of efficiency. A front-end pre-processor allocates the storage for access histories and inserts calls to the run-time libraries in the program being monitored. This implementation is relatively portable to other parallel Fortran system. The pre-processor approach also simplifies the implementation; however, the resulting monitoring efficiency is worse than if the compiler were modified.

Several optimizations were made to task recycling and English-Hebrew algorithms during implementation:

- For every monitored variable there is a "mirror" variable which contains its access history. Thus, when a block reads $A[i]$, the access history in $A_{ah}[i]$ is checked for anomalies and then updated. The access history lengths are fixed because of 16 megabyte memory constraints. While this limitation may result in undetected anomalies, we believe that anomalies will be missed only rarely, since the measurements discussed in Section 5.3 indicate that the size of the reader set is generally very small. Each entry in an access history contains a task identifier or a pointer to the English-Hebrew label and the line number and a pointer to the function name of the last access; this enables us to identify the locations in the program code where access anomalies occur.
- English-Hebrew labels consist of a string of two byte integers and a *length* field. They are stored indirectly in access histories and coordination lists. This decreases the size and complexity of access histories. There is no attempt made to reclaim space for labels no longer referenced. We feel the additional expense of maintaining reference counts outweighed the benefits of possibly decreasing storage requirements.
- Blocks can share parent vectors (in task recycling) or coordination lists (in English-Hebrew labeling) if they have identical sets of ancestors. For example, all blocks created by a doall operation share the same ancestors; one of these blocks must obtain a private parent vector or coordination list only if it terminates before the endall (e.g. by performing a nested doall or coordination operation). For a nested doall operation with outer parallelism of m and inner parallelism of n, this reduces the number of parent vectors and coordination lists by a factor of nm. [5]
- Both algorithms benefit from a "run-until-completed" scheduling paradigm, which is the common scheduling method used in the Ultracomputer and many other parallel Fortran environments. Parallel scientific codes typically exhibit a high degree of *nominal* parallelism. In the run-until-completed model, each block created in a doall will not block until it terminates at its associated endall operation. This limits the number of blocks which can ever run concurrently to n, the underlying parallelism of the machine. This is not a strong optimization for English-Hebrew labeling, since the majority of its space requirement is for block labels; it is more effective for task recycling since parent vector storage is more significant.
- In the Ultracomputer parallel Fortran environment, each of n actual processes perform the work of several parallel blocks. The differences among the parent vectors of these blocks are very small. Therefore, in order to reduce the work performed in maintaining parent vectors, a private "template" parent vector is cached for each level of nesting. When a block b updates its parent vector, the process which is performing the work of b also records the modifications. When block b terminates it must update the parent vectors of its children blocks. Instead of comparing all of the entries in its parent vector, it simply compares the recorded modifications.

[5] If local memory is available and the cost of accessing shared memory is much higher than local memory, sharing of data structures may not be cost-effective.

Likewise, when a new block is created the template parent vector must be reinitialized. The modification records are used to back out the changes made rather than reinitializing all of the entries.

For programs with only doall and endall operation, the per block cost is reduced from the maximum concurrency of the graph to the maximum level of nesting (the number of times that a change must be either added or backed out of a parent vector) and the amount of space and work per process is the product of the maximum concurrency and the level of nesting. Because coordination lists would have to be traversed to perform the updates, there is no corresponding optimizations for the English-Hebrew labeling scheme.

5.2 Concurrency Structure

For all four benchmark parallel programs, the concurrency structure is quite simple: there is very limited nesting of doall constructs and minimal synchronization. However the degree and granularity of parallelism vary considerably.

- Triso has coarse granularity parallelism with limited synchronization. It consists of a single doall operation which creates 8 parallel threads; these subsequently perform two *barrier* synchronization operations.
- Simple has medium granularity parallelism with some locking. It performs 10 doall operations that each create 124 parallel threads, and 130 operations that create from 10 to 30 threads. In addition, during 10 phases of execution approximately half of the 30 concurrent threads obtain a lock (represented by an asynchronous coordination edge in the POEG).
- Finite exhibits a large degree of fine granularity parallelism and does not use coordination operation. It performs 60 doall operations that each create 1000 parallel threads; 50 operations that create 250 threads and 200 operations that create between 2 and 32 threads. Each block performs a very limited amount of computation; in many case a block consists of a single operation on an array element.
- Polymer also exhibits a large degree of fine granularity parallelism and does not use coordination operation; however, it has one level of nested doall operations (the first three benchmark programs have no nested doall operations). It performs 40 nested doall operations; the outer operations create 1000 parallel threads each of which creates 3 parallel subthreads. In addition, it performs 20 doall operations which create 350 parallel threads and 10 doall operations which create 100 parallel threads.

The last two programs have such fine granularity parallelism that monitoring them primarily measures the cost of maintaining concurrency information rather than the cost of checking accesses to shared variables. Table 1 summaries the concurrency parameters for the benchmark programs. The experimental results presented in

Program	*Total #* *Blocks*	*Max* *Concur*	*Average* *Concur*
Triso	24	8	8
Simple	4090	124	28
Finite	74,900	1,000	245
Polymer	128,000	3,000	1,800

Table 1: Concurrency Structure

Sections 5.4 and 5.5 show that the concurrency structure of the program has a significant impact on the cost of maintaining concurrency information. However, the concurrency structure does not significantly impact the cost per variable access, as discussed in the next section.

5.3 Shared Variable Access Patterns

The work and space required for maintaining access histories is proportional to the average size of the reader set of an access history. While theoretically the size of the reader set may grow to the maximum concurrency of the graph, in practice the size is much smaller. Many parallel scientific codes distribute workloads by partitioning data among concurrent threads. Hence a thread often shares data with a neighboring thread, but seldom shares data with all other threads. Therefore, one would expect the number of concurrent readers to be limited.

The shared memory access patterns measured for the benchmark programs support this conclusion and are shown in Table 2. Each column gives the percentage of accessed[6] shared variables with *at most* the specified number of concurrent readers. In the Triso program, for example, 15% of the shared variables have two concurrent readers at some point during execution, but never have more than two.

The last column of Table 2 shows the average maximum reader set size for all variables with less than 10 concurrent readers[7]. For the first three benchmark programs, this includes virtually all shared variables and is an upper bound on the size of the reader set. Since for the Polymer program 10% of the variables have more than 9 concurrent readers, the figure in the last column only reflects statistics for 90% of its shared variables. If we are conservative and assume that all of these variables are accessed by all 3,000 concurrent block, the average reader set size is 279 concurrent readers (which

[6]Because fixed maximum sized arrays are used, many shared variables are never accessed.

[7]It is not possible to measure reader sets larger than 10 due to memory constraints on the Ultracomputer

	Percentage of Variables with Reader Set Sizes										*Ave*
Program	1	2	3	4	5	6	7	8	9	>9	*Size*
Triso	80.2%	15.0%	4.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.25
Simple	49.3%	41.0%	9.6%	†	0.0%	0.0%	0.0%	0.0%	0.0%	†	1.69
Finite	48.4%	32.5%	4.6%	†	†	0.9%	†	†	13.4%	†	2.71
Polymer	72.9%	9.4%	2.6%	1.1%	2.8%	0.8%	2.6%	6.2%	2.0%	9.3%	1.31

† Less than 0.1% of the variables have reader sets of this size

Table 2: Reader Set Sizes

is still much less than 3,000). If we are slightly less conservative and assume that all of these variables are accessed by the average number of concurrent blocks, the average reader set size is 168.

Table 2 reveals two somewhat surprising results:

1. The size of reader sets tends to be very small. In all four programs, more than 80% of the variables are never read by more than two concurrent readers and almost 50% are never read concurrently. Moreover, a variable with maximum measured reader set size of n may actually have a much smaller reader set size throughout most of the execution of the program. Hence our estimate of the average reader set size is pessimistic with respect to the actual average reader set size.
2. There appears to be little correlation between the average reader set size and the degree of parallelism of the program. For example, the Triso and Polymer programs have fairly similar access patterns for the majority of their variables (excepting the 9.3% in the Polymer program with more than 9 concurrent readers). Triso, however, has a very low degree of parallelism while Polymer has nested parallelism of a very high degree.

Since the average reader set size is so much smaller than their worst case bounds, we conclude that the access history based algorithms are preferable to that of [11] in which the storage depends on the maximum concurrency of the POEG.

5.4 Space Requirements

For the results described in the following two sections four versions of each program were executed:

Unmonitor	- unmonitored program
Monitor(1)	- monitors every shared variable using a reader set size of one
Monitor(2)	- monitors every shared variable using a reader set size of two
Concurrency	- maintains concurrency information only; no variables are monitored

Table 3 compares the memory requirements for each of the four versions of the benchmark programs listed above. The static size is the size of the object module which includes additional monitoring code as well as storage needed for access histories. (This is virtually the same for both algorithms). The dynamic size is the amount of storage needed for concurrency information, which is allocated at run-time. As is seen in Table 3, English-Hebrew labeling requires substantially more space than task recycling for maintaining concurrency information. In fact, it is not possible to obtain the actual dynamic memory requirements of Polymer for the English-Hebrew labeling scheme; the amount of storage needed for concurrency information exceeds the capacity of the Ultracomputer and hence an estimate was calculated based on the concurrency structure of the POEG.

	Static Sizes			*Dynamic*	
Program	*Unm*	*Mon(1)*	*Mon(2)*	*TR*	*E-H*
Triso	163	660	1,000	2	8
Simple	314	921	1,270	6	80
Finite	230	745	1,020	29	954
Polymer	639	2,898	4,461	279	7,000 †

†Estimated Value

Table 3: Space Requirements (in Kbytes)

5.5 Execution Times

Table 4 displays user mode execution times for the four benchmark programs using both of the anomaly detection algorithms. *Concurrency* isolates the cost of main-

Task Recycling				
Program	*Unm*	*Conc*	*Mon(1)*	*Mon(2)*
Triso	3.5	4.1	19.6	22.7
Simple	202	263	550	598
Finite	98	734	959	1108
Polymer	789	3330	4367	4607

English-Hebrew †				
Program	*Unm*	*Conc*	*Mon(1)*	*Mon(2)*
Triso	3.5	4.7	20.9	24.4
Simple	202	237	579	676
Finite	98	407	704	1072

†Polymer was not monitored due to memory constraints

Table 4: Total Execution Time (in seconds)

taining the concurrency information from the overall cost of detecting access anomalies. This cost is substantial for the programs with very high degrees of fine-grained parallelism. In the current implementation of the task assignment algorithm, assigning and freeing a task requires locking vertices in the free task dag, which creates a serial bottleneck. If free tasks are stored in parallel access data structures (e.g. parallel access queues), it may be possible to decrease this serialization effect.

For the Triso and Simple programs, the English-Hebrew labeling algorithm requires more time than task recycling for monitoring variables (as is shown in the run times for *Monitor(1)* and *Monitor(2)*), even when the cost of maintaining concurrency information is less (as in shown in the run times for *Concurrency*). The primary cost in the Polymer and Finite programs is the maintenance of concurrency information; this is not surprising given the fine granularity of the parallelism in the two programs. For both algorithms, the overall increase in computation time is too high for transparent monitoring of all shared references. Nevertheless, we not believe the cost it too prohibitive for on-the-fly anomaly detection to be a useful debugging tool. Moreover, static analysis and/or a compiler-based implementation should make it more efficient.

Note that the time measurements presented are the total execution times of all concurrent threads rather than elapsed running time. The maintenance of concurrency information results in a 57% - 160% increase in elapsed running time, while monitoring all accesses to all shared variables with reader sets of size one incurs a 160% to 460% increase in execution times. The difference between the elapsed and execution time increases stems from Amdahl's law: almost all of the additional work is performed in parallel.

From Section 5.3 we know that using two entry reader sets instead of one entry reader sets significantly increases the percentage of variables guaranteed to have at least one anomaly detected (from 50% to 80%). A comparison of the execution times for *Monitor(1)* and *Monitor(2)* in Table 4 shows that we may do so with little additional cost. Table 5 isolates the cost of maintaining access histories and shows the percentage increase when using reader sets of size two instead of one. The isolated times are computed as the total monitored

	Mon(1)		*Mon(2)*		*Increase*	
Program	*TR*	*E-H*	*TR*	*E-H*	*TR*	*E-H*
Triso	16.4	16.3	21.0	22.1	20%	22%
Simple	287	242	335	339	17%	40%
Finite	225	297	354	685	57%	130%

Table 5: Access History Update Time (in seconds)

execution time (*Monitor(1)* and *Monitor(2)*) less the execution time when simply maintaining concurrency information (*Concurrency*). These relative time increases indicate that as reader sets grow in size, the cost of detecting anomalies increases more rapidly for English-Hebrew labeling than for task recycling. Therefore, as larger reader sets are used the cost of the more complex task concurrency verification in the English-Hebrew algorithm becomes increasingly more significant.

6 Concluding Remarks

This paper presents the problem of detecting access anomalies in parallel programs under a models of concurrency which incorporates nested parallel loops and synchronous and asynchronous coordination. An existing on-the-fly algorithm - English-Hebrew labeling - is described and compared with a new algorithm based on task assignment to a partial order execution graph. Experimental data from monitoring four benchmark scientific programs using the two algorithms indicate four important results:

1. The benchmark programs use data partitioning so extensively that over 80% of all variables never have more than two concurrent readers. Therefore, the size of access histories appears to be independent of the degree of parallelism within the program.
2. The task recycling algorithm requires much less storage for concurrency information than English-Hebrew labeling.
3. Because of its efficient concurrency information management, English-Hebrew labeling performs better on programs with frequent doall operations and relatively few accesses to shared variable.
4. However, as reader sets or concurrency lists increase in size the high cost of performing concurrency verification in English-Hebrew labeling outweighs the cost of parent vector management in the task recycling algorithm.

If the benchmark programs are indicative of a wide class of parallel programs, the task recycling algorithm is an important improvement over existing technology.

Acknowledgements

We would like to thank Marc Snir and Larry Rudolph for their introduction to this problem and Ron Cytron, Allan Gottlieb, Bud Mishra and Ed Schonberg for their many helpful suggestions.

References

[1] Todd R. Allen and David A. Padua. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the International Conference on Parallel Processing*, pages 721–717, Aug 1987.

[2] David Callahan and Jaspai Subhlok. Static Analysis of Low Level Synchronization. In *Proceedings on the SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[3] Jong-Deok Choi, Barton P. Miller, and Robert Netzer. *Techniques for Debugging Parallel Programs with Flowback Analysis.* Technical Report, University of Wisconson, Aug 1988.

[4] Anne Dinning and Edith Schonberg. *An Evaluation of Monitoring Algorithms for Access Anomaly Detection.* Technical Report Ultracomputer Note #163, New York University, July 1989.

[5] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminancy in Parallel Programs. In *Proceedings on the SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[6] Allan Gottlieb. An Overview of the NYU Ultracomputer Project. In J.J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25 - 95, Elsevier, 1988.

[7] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), Jul 1978.

[8] Barton P. Miller and Jong-Deok Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings on the SIGPLAN Workshop on Parallel and Distributed Debugging*, May 1988.

[9] Itzhak Nudler and Larry Rudolph. Indeterminancy Considered Harmful. 1988.

[10] Itzhak Nudler and Larry Rudolph. Tools for the Efficient Development of Efficient Parallel Programs. In 1^{st} *Israeli Conference on Computer System Engineering*, 1988.

[11] Edith Schonberg. On-The-Fly Detection of Access Anomalies. In *Proceedings on the SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1989.

[12] Marc Snir. Private correspondence. 1988.