# 1 Introduction

Debugging parallel programs is inherently more difficult than their sequential counterparts. Large amount of trace data, confusing effects of asynchrony, and the evasiveness of consistent global states are some of the hurdles that a debugger must handle in order to provide the user with useful information regarding the execution of the program that can help locate an error. Event-based behavioral abstraction provides a mechanism to handle the volume of data by allowing users to specify the program behavior in a hierarchical fashion, which is then compared with the actual execution [2, 3, 14, 16]. Transformations of logical time helps to filter out the perturbations of physical asynchrony, and allow the user to examine the program behavior from different temporal perspectives [14, 19, 6]. Many systems try to attack the problem of comprehending large amount of data, by providing the user with an animation system [14, **?**, **?**]. However all the approaches suffer from two major drawbacks: firstly they do not provide sufficient feedback, nor do they scale up for massive parallelism. Modeling languages using complex recognition strategy are good for expressing very complex behavior succinctly, but are terribly inept at giving informative feedback on abstractions that do not correspond to the observed behavior. Processor-oriented animation schemes, although informative when the observed behavior matches the expected pattern, do not scale well to even moderate number of processors (such as 64), and depends on the user for the specification of the processor placement [14, **?**]. Animation schemes that do not need any user assistance in the placement of processors [**?**] only provides animation of primitive events that are susceptible to the effects of physical asynchrony: even with logical time manipulation such animations are hardly useful, due to the lack of high level abstraction mechanisms. We address both these problems in our new debugger Ave/Ariadne.

Ave/Ariadne is an integrated debugging environment. Ariadne is the modeling component that provides a simple language to specify behavioral abstractions as patterns of events in logical time. The patterns are detected by a collection of finite-state recognizers. Ave is the scalable visual feedback engine, which provides feedback on complete and partial matches, highlighting the error path if the match fails. There are four salient features of Ave/Ariadne environment: The first is the ability to provide feedback on failures, the second is the scalability of the patterns, the third is the conciseness of its modeling language, and the fourth is the provision for incremental exploration of the program execution from different perspectives.

Debugging is an exercise in exploration. Programmers often do not anticipate bugs. So the initial description of the behavior is bound to be coarse. Ave/Ariadne provides the user with an environment where a coarse behavioral description can be refined at successive iterations guided by the feedback from the debugger. This exploration of different facets of program behavior will either lead the user to the error, or convince him/her that the behavior is indeed the correct one. Ariadne provides the user with the functional queries to explore the characteristics of the match. As an example, the user can match a number of multicasts, and then use a functional query to find out the which processes actually sent the multicasts. The user can also calculate new attributes for the matched abstractions, and thus explore different dimensions of the behavior. Ave's strategy of projecting the new attributes in a color domain makes it easier for users to group similar abstract events together. As an example, the user may match a number of data exchanges between processors. He or she can then ask Ariadne to compute a new attribute for each exchange which gives the distance between the processors that exchanged data values. Different distances get projected to different colors. The display in Ave highlights events with the same distance attribute with the same color. The user can easily identify events that are similar from this color scheme.

Section 2 gives a brief overview of Ave/Ariadne environment. Section 3 provides several sample debugging sessions illustrating its capabilities.

# 2 Our Approach

Ave/Ariadne is a *post mortem* debugger for massively parallel systems. Although originally designed for MIMD message-passing system, the current incarnation can also handle shared memory program traces. It is designed for correctness debugging, and it supports the user in investigating global interprocess communication patterns.

Ariadne operates on traces produced by parallel programs.[1] Within these traces, processes, identified by integer process ids (PIDs), execute sequences of primitive events. The debugger supports four primitive event types: Read, Write, Multicast, and Phase Marker. This repertoire can easily be extended by the user at the run-time. Reads, Writes, and Multicasts denote interprocess communication, and Phase Markers denote the ends of logical phases of computation. The traces are stored internally in an *execution history graph* where the nodes represent events and the edges represent communication events; from these traces we can derive Lamport's *happened before* relation [17]. A number of debuggers provide visualizations of execution history graphs [9, 11, 12, 21] but such visualizations do not scale well for massively parallel systems. Ave/Ariadne supports interactive, textual and graphical explorations of the graph.

Here we briefly describe three aspects of this support: the modeling language, the facilities for manipulating logical temporal orderings, and the visual interface that is available for investigating the characteristics of pattern matches and mismatches.

## 2.1 The Ariadne Modeling Language

As mentioned above, previous attempts at using event-based abstraction in debugging have been limited by the complexity of the modeling language. Ariadne's language is quite simple. It employs a three level description of communication patterns in terms of *chains*, *p-chains*, and *pt-chains*.

- *Chains* are patterns representing "local views" of communication. They are described by extensions of regular expressions. When they are matched against an execution history graph, all events in the chain must occur exactly in the order specified with the exception of communication events that are not physically realized because of "edge effects" on process array boundaries [1].

- *p-Chains* are patterns representing the concurrent execution of a chain by a set of processes. They are described by binding a chain to a process set. When a p-chain

[1] Currently some of our traces are taken from a simulator.

is matched against a behavior, a copy of its chain is matched on each element of its process set (events can be shared across, but not within, chains).

- *pt-Chains* are patterns representing the logical, temporal composition of a set of p-chains. They are matched in a two step process: first events matching the p-chains are located in the graph and then the specified logical relations between those events are verified. When a pt-chain has been successfully matched, it returns an *abstract event* which is then assigned to a *match variable* thereby producing a tree structure containing the matched instances of events with the match variable as the root. We will call this tree of matched events the *match tree.*

  Each hierarchical level translates to a level in the tree. So all the internal nodes of the tree correspond to some abstract event as defined by the user. The leaf nodes of the *match tree* are the primitive events that are consumed by the match. The level containing the children of the *root* contains the set of events that constitute the pt-chain that was matched. The primitive events are removed from the trace and are unavailable for further matching unless explicitly restored, although the user retains the capability to superimpose other patterns on the same set of primitive events.

These three definitional levels appear to form a natural mechanism for describing parallel systems, as evidenced by their use in other contexts such as the XYZ levels of Phase Abstractions [24] and the LaRCS specification language [22]. The matching algorithm for our language is straightforward: a pt-chain is recognized by a finite state machine that invokes copies of other finite state machines to recognize chains on specific processes. This matching can be done efficiently, avoiding the costliness of pattern matching approaches [3, 13] and the expensive implementations of previous languages [16]. At the same time, our matching algorithm can provide precise information on the reasons for a match failure.

## 2.2 Logical Time Manipulation in Ariadne

Programmers are often confused by the results of asynchronous executions because they can not foresee all possible interleavings of events. In fact, most of these interleavings are irrelevant: the programmer does not care about the arbitrary orderings of events by physical time. Instead, the programmer is concerned with the *logical ordering of events.* At the primitive event level, this ordering is captured by Lamport's *happened before* relation. Other debuggers have used temporal logic to express assertions about *happened before* [5, 10] but they do not use behavioral abstraction. We extend the relation to abstract events, defining three relations: **precedes**, **parallels**, and **overlaps** [15]. Informally, if $A$ and $B$ are abstract events, then

$A$ **precedes** $B$ (denoted $A \rightarrow B$) iff there is some dependency from an event in $A$ to an event in $B$ but no dependency from an event in $B$ to an event in $A$,

$A$ **parallels** $B$ (denoted $A||B$) iff there is no dependency from an event in $A$ to an event in $B$ and no dependency from an event in $B$ to an event in $A$, and

$A$ **overlaps** $B$ (denoted $A \leftrightarrow B$) iff there is both a dependency from an event in $A$ to an event in $B$ and a dependency from an event in $B$ to an event in $A$.

Similar extensions have been proposed [8, 16, 18] but we have found our definitions to be more appropriate for debugging. In particular, our **precedes** relation captures the notion that just some part of a complex event *happens before* some part of another event; other definitions require a total ordering which is rarely found in the programs we have examined.

## 2.3 Ave: The Visual Feedback Scheme for Ariadne

In our earlier paper we presented Ariadne's [7] support for interactive, textual exploration of the program execution graph. The textual feedback, although complete, can overwhelm the user when the pattern to be matched is data dependent. As an example, consider the case when the user specifies the following behavior:
There can be a number of global communication phases. At each phase atleast one of the processors in the set {0,1,2,3} will send atleast one message followed by a synchronization operation performed by all processors. Each synchronization operation produces a *barrier* event, which is user-defined. The user can add this event to Ariadne's repertoire at run-time. The user may write a description like:

**? ch msg = W @ R**
**? pch Comm = msg+ onsome {0,1,2,3};**
**? pch Synch = barrier onall {0,1,2,3};**
**? match Scatter = (Comm Synch)*;**

The chain *msg* describes a single message by matching the send (W) and the receive (R) of the same message (@ implies following the edge that connects the send and the receive of the message). At the *pchain* level *onsome* suggests that the chain should match on atleast one processor.

Say there were 2 communication phases. On the first communication phase processors 0 and 2 sent 2 and 3 messages respectively. In the second phase only processor 3 sent 2 messages.

With data dependency permeating different hierarchical abstraction levels, we must allow the user to control the feedback. In the above example, the user is interested in feedback at different levels:

PtChain level: How many phases were there?
Pchain level: Which processors took part in different phases?
Chain level: How many messages did a particular processor send in a given phase?

Also depending on the abstraction, a pt-chain can have more than one hierarchical level, and the user would want to know about aspects of different hierarchical abstractions separately. So we need to give users the control to meaningfully explore the *match tree* and provide the feedback in a hierarchical manner.

Ave produces the display shown in Figure 1 at the beginning. Note that Ave has collapsed the behavior at the p-chain level. The node in green rectangle represents behavior collapsed over a processor set. Although each processor in the set executes a pattern msg+, the number of messages sent by individual processors may be different. Ave shows it by printing the total number of processors in the set, followed by number of partitions. (2,2)

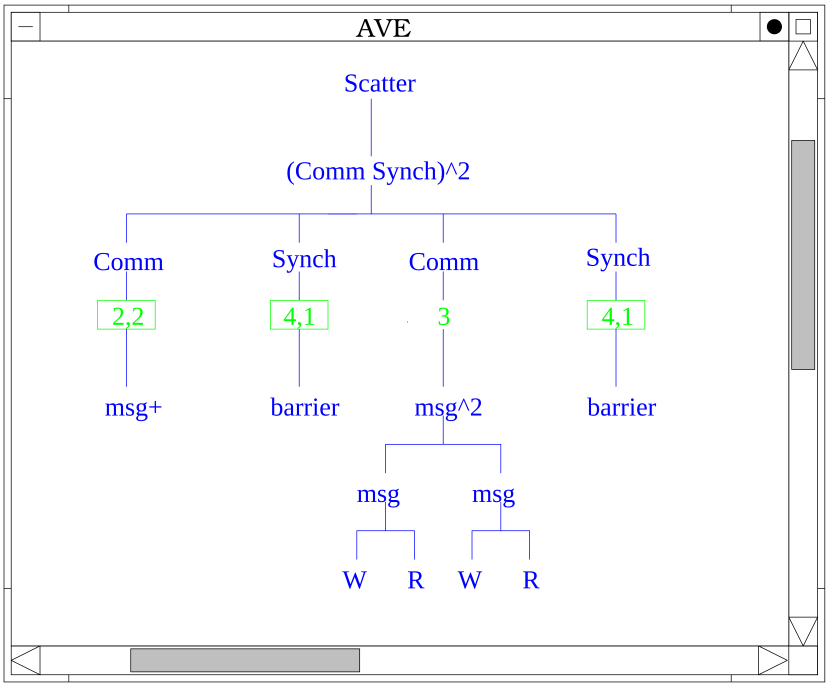


Figure 1: Initial feedback of Ave.

says that there are 2 processors in the set, and there are 2 partitions: that is the processors behaved differently. In the second phase since the number of processors is 1, the processor number is shown in green, followed by a node *msg^2* showing how many messages were sent by the processor. The user can click on the collpased node, and ask Ave to expand it fully, and get the display shown in Figure 2. Ave uses the following color scheme to show the events: All events are colored blue, the processor sets over which behavior is collapsed is colored green. If an event did not match, then it is colored red. The red events, if they are present, must form the right boundary of the display: since Ariadne stops as soon as it fails to match an event. The only exception is when the match fails due to the failure of logical assertions: in this case the events between which the assertion fails for the first time is surrounded by a red rectangle (see the Gaussian elimination example).

Ave's *match tree* oriented feedback can present information in a hierarchical fashion, and can make it easy for the user to comprehend. Later in Section 3 we give two examples where the animating the *match tree* itself leads the user to the bug without any further exploration.

Ariadne's scalability is translated to Ave's animation scheme, since the pchain level pattern description collapses similar behavior on the part of different processes onto a single abstract event. Once the user specified pattern has been matched, the user is free to compute different attributes of the abstract events defined in the match. Ave partitions the attribute space into equivalent classes — each class being assigned a distinct color. This color scheme makes it possible for the user to mentally group similar abstract events.

Although the pchain level description collapses similar behavior across sets of processes,

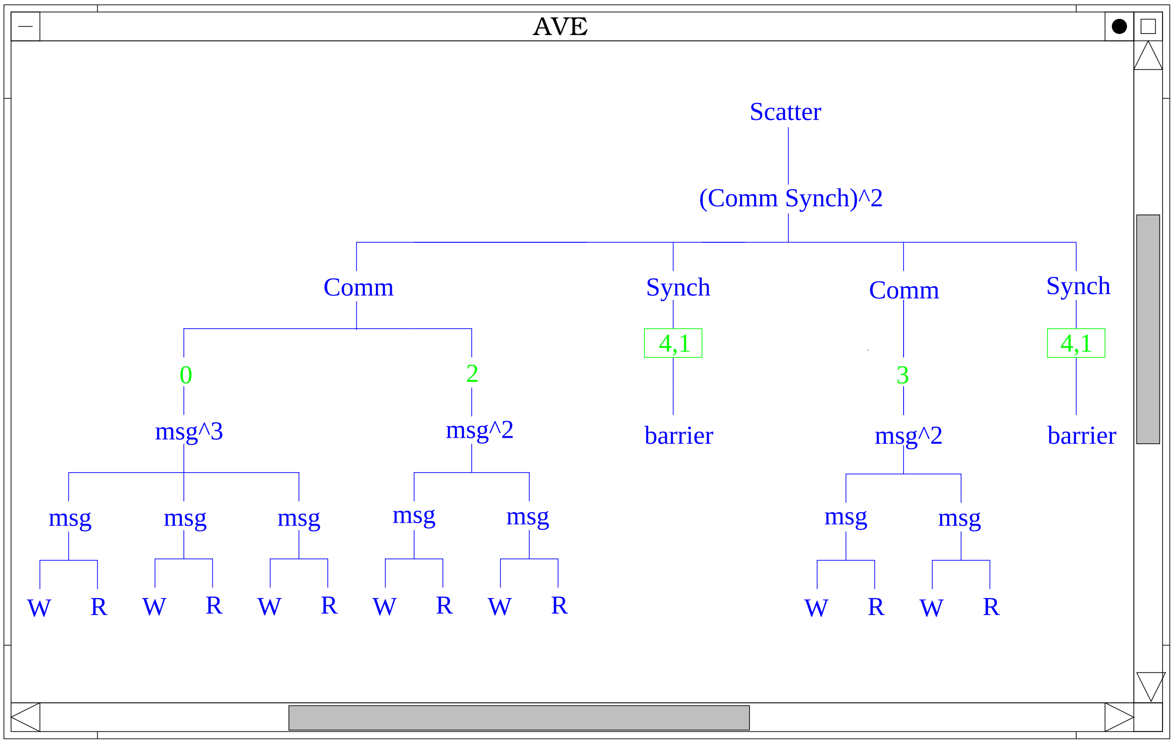


Figure 2: Ave feedback with behaviors uncollapsed.

Ave can show the behavior of individual processes if so asked (as is needed in the data dependent pattern described earlier).

If the match fails, Ave displays the partial match tree (the portion that matched), with the failure point highlighted in red.

# 3 Sample Debugging Sessions

In this section we illustrate the usefulness of our visual feedback strategy and the power of the modeling language. The examples will include the use of logical time, and integrated query facilites and the incremental dynamic attribute calculation. We describe four sample debugging session with Ave/Ariadne prototype.

## 3.1 Dictionary Search on a Hypercube

In our previous paper, we use this example to show that textual feedback alone does not suffice in some cases. The use of the simple visual feedback mechanism as provided by Ave however reveals and locate the bug [2]. The program implements a dictionary search in which queries are pipelined from a host to a database of key-ordered records stored in a hypercube.

[2]This should be compared with the performance of Belvedere, where the complex animation scheme, with user having to specify the layout of the processors in the screen, only reveals the bug, but does not locate it

Queries are routed within the cube to the proper node using binary search. More than one query is active at a time. The program as written contained a routing error.

We consider an 8 processor hypercube with processors having PIDs 0 to 7, and a host process with PID 8. We model the behavior of the program as a series of queries, each query starting at the host, traversing the cube, and eventually returning to the host. Each query has two parts to it: the first is the routing of the query from the host to the appropriate node (*Find* phase), and the second is the routing of the reply to the host from the answering node (*Reply* phase). The messages that are sent as a part of the *Find* phase are tagged as "ask" messages, and the messages that are part of the *Reply* phase are tagged as "ans" messages. The routing of the query from the host to the appropriate node should be according to the binary search pattern, the routing of the response should follow the shortest path between the node and the host. The behavioral model is described using Ariadne as follows:

**? set cube = {0 .. 7};**
**? ch ask = (W_ask @ R_ask) ;**
**? ch ans = (W_reply @ R_reply) ;**
**? pch Find = ask+ onall {8};**
**? pch Reply = ans+ onsome cube;**
**? match Query = (Find Reply)+;**

The behavioral description defines a set *cube* of integers 0 to 7. These are the PIDs of the processors that make the hypercube. The chain *ask* defines a single message of the *Find* phase, and *ans* defines a single message of the *Reply* phase. A query is composed of a *Find* followed by a *Reply*. The pchain *Find* starts with matching a message tagged as "ask" on processor 8, and follows the subsequent messages tagged "ask" until it reaches the processor, that answers the query with a message tagged as "ans". Since the identity answering processor is not known beforehand, *onsome* specifies that the *Reply* phase should start on one of the processor in the hypercube. The Ave feedback is shown below:

The feedback shows that the match is composed of 2 queries. The first question took 5 hops to reach the appropriate node as is shown by the feedback *ask^5* and took 5 hops to reach the host processors from the answering node as evidenced by the feedback *ans^5*. Note that the distance of the host processor from any answering node can atmost be one greater than the degree of the hypercube, in our case of a 8 node hypercube, the distance should not be greater than 4. Since the answer of a query is supposed to be routed to the host through the shortest path, clearly there is an error in the routing code. The first part of the query (from the host to the appropriate node) follows the binary search pattern which may involve backtracking: so 5 hops in search of the appropriate node is not an error. The combined power of the visual feedback scheme (Ave) and the hierarchical modeling approach (Ariadne) not only reveals the bug but makes it possible to hint at its location (routing code specific to the answer).

## 3.2 FFT on a Mesh

This debugging session with the MIMD message-passing version of the FFT on a mesh illustrates the use of dynamic attribute calculation and the difference that Ave's visual feedback makes. The program was written for 16 processors, with static data partitioning. At suc-

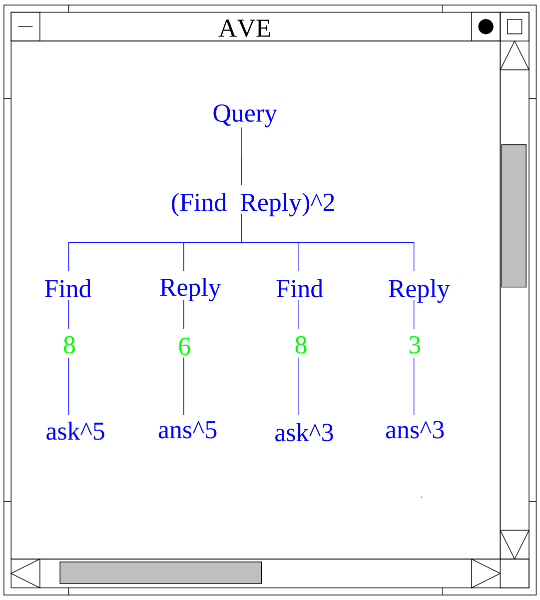


Figure 3: Ave feedback in the dictionary search example.

cessive iterations, processors 8 units apart, 4 units apart, 2 units apart and 1 unit apart exchange their data values to do the transformation. The program produced incorrect results. Initially, the user modelled the behavior with Ariadne as follows:

**? set proc = {0..15};**
**? ch exch = W @ R ;**
**? pch comm = exch onall proc;**
**? match fft = < comm^4 >;**

*proc* is a set that contains the PIDs of the processors. *exch* defines a single message, and *comm* is a collection of 16 messages each being sent by one of the processors. The whole behavior consists of 4 phases, one causally following the other, where each phase consists of 16 messages being sent. The angular brackets in the *match* statement indicate that **precedes** should hold between successive phases. The initial behavioral description produced the following Ave feedback without the colored bars appearing at the bottom. Note that the display collapses the behavior at the p-chain level. But since the chain level description is not data-dependent, the number of partitions is 1 in all cases, and is not shown.

Notice that in Ariadne, we do not have a way to specify the distances over which the communication should take place, or the receiver's PID at the chain level. This is not a limitation however, since once the match is done with a loose pattern definition, we can then compute the attributes of the matched events very easily.

**? attach to exch {int distance = pid(W) - pid(R)};**

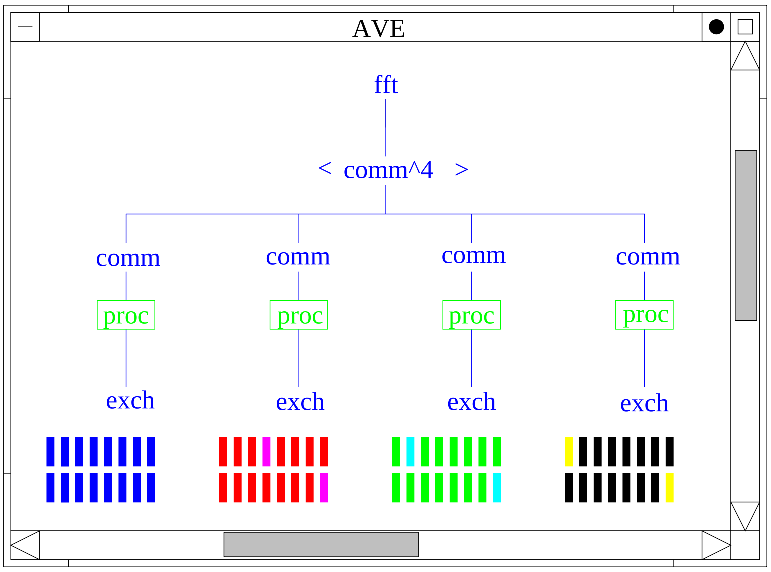


Figure 4: Ave feedbak in the FFT example.

In Ave, the attachment of attributes are handled in two ways: Ave can show the attributes in numerical form along the nodes, or it can do an attribute collapsing using colors. For instance, after the attributes are calculated, the attribute set is partitioned, with each partition getting a distinct color. Each uncollapsed node then gets colored by the color of the attribute. In case of collapsed nodes, the colored bar appears at the bottom of the node. The color scheme makes the grouping of related events readily obvious.

Note in this case, each phase of the pattern **fft** should have 16 bars all with the same color under them (all nodes who communicate in the *i-th* phase should be at a distance $2^{3-i}$ apart, where $i = 0..3$). However Ave produced the same color only for the phase that communicates with processors at a distance 8 from each other. For all other phases there were 2 nodes with a different color. This led us to the bug in the code segment which identifies the partner with whom to exchange the values. The user made an off-by-one error in the partner calculation, which resulted in erroneous data exchanges between some processors.

## 3.3 Gaussian Elimination

This is a shared memory version of the Gaussian elimination program featuring static data partitioning. The program consists of two phases: an elimination phase followed by a back substitution phase. The parent process spawns a number of child processes (as requested by the user). The elimination phase consists of $n$ iterations for a $n$x$n$ system of equations. At each iteration, the parent process does *pivoting* if necessary, which is followed by all the processes taking part in the *elimination* phase.

The code segment is produced below (with the location of the bug pointed out in comments).

The program produced correct results most of the time. With a certain 10x10 matrix,

```
pid = m_get_myid();
current_row = 0;
do {
    /* Pivoting phase: Parent-only code */
    if (pid == 0) {
     if (matrix[current_row][current_row] == 0.0)
       get_pivot(current_row, pid)
     }

    m_sync();  /* Synchronization */

    /* Elimination phase: every process executes it */
    row = current_row + pid + 1;

    while (row < num_rows) {
      do_elimination(current_row, row, pid);
      row += processors;
    }
    current_row++;                      /* Move on to the next row. */

    /* Missed Synchronization */

 } while (current_row != num_rows);
```

Figure 5: Code segment showing the elimination phase of Gaussian elimination.

it produced incorrect results only when the number of processes exceeded 8.

The back substitution phase was traced, and checked, and was found to be correct. We then proceeded to test the row elimination phase by modeling the behavior with Ariadne. Next we explain the instrumentation procedure for this shared memory program.

Data dependency in a shared memory program manifests itself through the access of the shared variables. As a result we need to trace the shared variable accesses in order to capture the causal data dependency prevalent in the program execution. In the gaussian elimination program, each process accesses a row as an aggregate (the whole row is read or written), thus we need not distinguish between accesses to different elements of the matrix as long as they belong to the same row. We instrumented our program in a way, that attaches a logical clock to each row[3]. The effect of accessing a row thus manifests itself by producing events which are timestamped by the logical clock maintained for that particular row. Since a row access in our example should be atomic, we need to produce events so as to demarcate the interval over which the access takes place: violation of atomicity can then be found by searching for overlapping access intervals. The time interval can be captured by producing

[3]Each row of the matrix thus behaves as a "virtual" process.

an event at the beginning of the access, and another at the end of the access. So every time a process performs a shared access, it produces in its trace an event timestamped by the logical clock maintained for that row (this produces the correct timestamps since reads are blocking); the event also contains the identifier of the row, so that accesses to same shared location can be compared for checking atomicity.

The accesses that are part of the *pivoting* phase are tagged as "pivot", and the accesses that are part of the *elimination* phase are tagged as "elim".

The *get_pivot* function is called only by the parent process. It requires swapping the row with a zero pivot element with one that has a non-zero element in the pivot position. So this requires *read* accesses to two rows, followed by *write* accesses to the same two rows. Thus a *pivoting* phase can be described simply as:

**R_pivot R_pivot W_pivot W_pivot**

Similarly each process in the elimination of a row coefficient reads two rows and writes to a single row. This can be described in the chain level as

**R_elim R_elim W_elim ;**

The complete behavioral description for a 9 process system can then be described as:

**? set proc = { 0..8 };**
**? pch pivot = R_pivot R_pivot W_pivot W_pivot onall {0};**
**? pch elim = (R_elim R_elim W_elim)* onall proc ;**
**? match Gauss = < (pivot* elim)* >;**

The angular brackets in the match statement specifies that **precedes** should hold between *pivoting* and *elimination* of a single iteration, and also between two successive iterations themselves. The Kleene star in *pivot** is to model the fact, that pivoting may or may not occur in an iteration. The Ave feedback is shown below.

Note that in the first iteration, there was no pivoting. The *elimination* phase of iteration 1 did not *precede* the *pivoting* phase of iteration 2. There was a race between process 0 (the parent) doing *pivoting* on row 2 (where it swaps rows 2 and 3) on iteration *2* with process 1 still eliminating coefficient of row 3 in iteration *1*. The programmer needed to insert a synchronization call at the end of each iteration (shown in the code segment with in comments at the end of the while loop).

The program worked fine when number of processes were less than 9, because in this case with the given input the parent process had to take care of more than rows in the first iteration. This gave process 1 enough time to finish working on row 3.

## 3.4 Sample Example from Sequent's Guide To Parallel Programming

This example comes from the Sequent's Guide to Parallel Programming (page 5-25). The sample (supposedly "correct") shared memory program uses dynamic job allocation and given a list of cities with cartesian coordinates finds the city closest to the origin. Each process calls *m_next* a system call in Sequent, which increments a global counter atomically, and

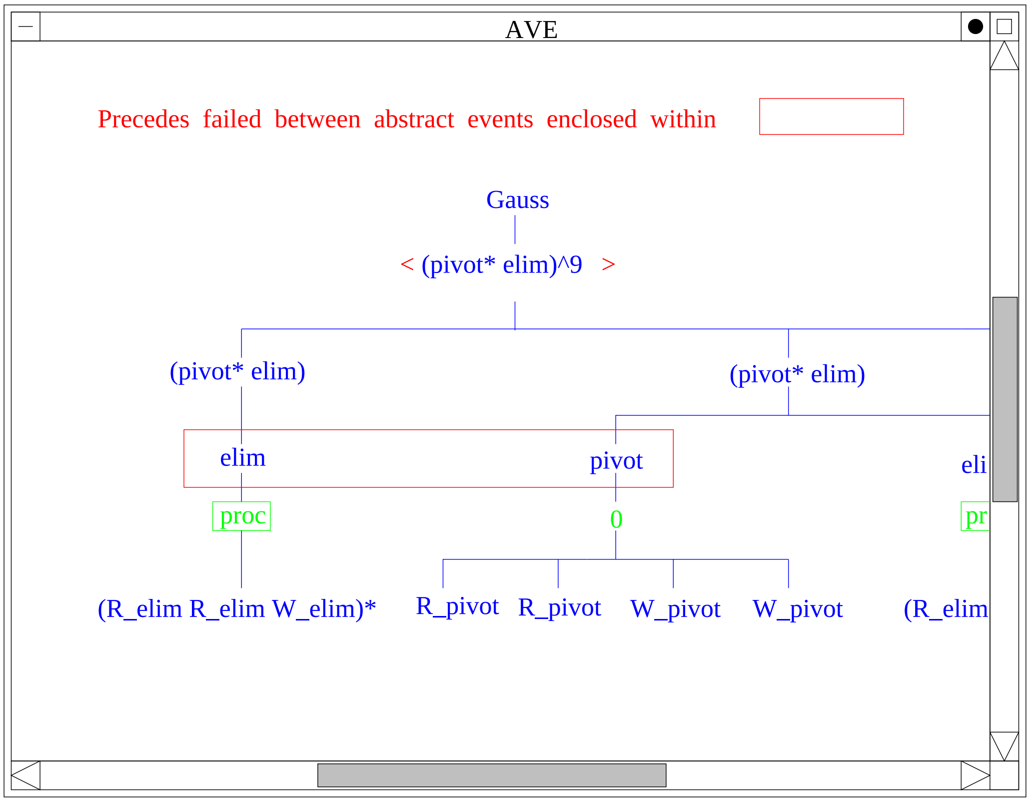


Figure 6: Ave feedback showing *precedes* failure in Gaussian Elimination.

returns a queue location. The process computes the queue locations that it should examine according to the counter value, and the value of a variable called *BITE* that determines the chunk of the queue a process should have. It then reads the data, computes the distance between the cities and the origin, and if the computed distance is less than the shared variable *closest*, overwrites *closest*. Access to the shared variable is protected with a semaphore. We were trying out our prototype on a shared memory example, and we wrote the following process-centric model when the system was run on a 4 process system with 10 cities in the list. BITE was set to 1, forcing every process to work on a single queue location at a given time. We instrumented the program such that every time a process reads an element from the list, an R_list event is generated and an R_var (W_var) event is generated everytime a process reads(writes) the variable *closest*.

**? set proc = {0..3};**
**? ch axs = R_list R_var W_var*;**
**? pch comp = axs* onsome proc;**
**? match find = comp;**

*proc* is a set containing the identifiers of each process. At each iteration, a process reads the list of cities, computes the distance, reads the shared variable *closest*, and if the distance is less than *closest*, updates the variable, which is described by the chain *axs*. The total behavior of the system is zero or more *axs* chains matching on each process. The Ave feedback with the user asking for uncollapsing of the process set is shown below.

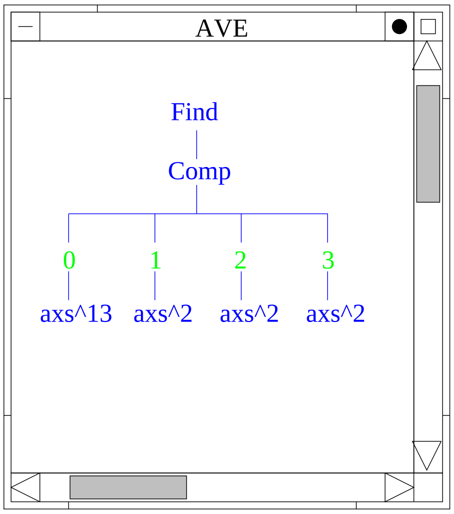


Figure 7: Ave feedback for the example program from Sequent manual.

Since we had only 10 cities in our list, the queue should not be accessed more than 10 times. However a quick look at the feedback shows that process 0 alone had accessed the queue 13 times.

Each process accesses the queue within a *for* loop, where the index element of the loop, gives the location within the queue to be examined. The lower limit of the loop corresponds to the first element ot be accessed in the queue in any given iteration, and was set to *base = (global counter value - 1)*BITE*. The lower limit is ofcourse unchanged in the loop. We instrumented the program in such a way, that every event also has an attribute that contains the current value of the local variable *base*. Since *BITE* is set to 1, the *for* loop should be executed only once for a given value of *base*. The new trace was then processed using the same behavioral description. We then uncollapsed the behavior of process 1, and found that both the iterations of *axs^2* was done with the same value of *base*. Exploring the other processes revealed the same pattern with each process accessing two consecutive queue locations with the same value of *base*. So each process was looking at two queue locations in each iteration instead of one. A look at the setting of upper limit revealed that the upper limit was incorrectly set to *upper = base+BITE* instead *upper = base + BITE -1*. This resulted in each queue element except 9 to be examined twice, when *BITE = 1*.

# 4 Conclusion

The combination of the modeling approach of Ariadne [7] with the visual feedback scheme of Ave is a powerful tool to tackle the debugging issues in masssively parallel computing. Ave's approach of giving the visual feedback in terms of the *match tree* frees the user from the layout considerations that needed to be tackled with other visualization systems [14, **?**]. At the same time, Ave imposes the user's view of abstraction over the primitive events which makes it more useful than the tools that only provide process time graph displays of the computation [?]. Ave's technique of graph collapsing (which it gets practically for free from its association with Ariadne's modeling hierarchy) alleviates the scalability problem that is often associated with visualizing massively parallel systems. Its capability to generate additional attributes from the *match tree*, and its display scheme of partitioning the generated attribute domain in terms of color give the user an easier way of aggregating behavior, and hence locating errors serendipitiously.

Parallel programs are best understood from different perspectives. Ave gives the user the ability to transform the *match tree* by a mechanism similar to *set comprehension* that is available in functional languages like *Haskell* [?]. This allows the user to check whether the same set of events satisfy a number of constraints, and gain more confidence about their modeling. All functions in Ariadne walks the same *match tree*, and so it is quite easy for the user to build new functional queries. All that is needed is the definition of the function that Ariadne should perform while it walks the tree. The user can also extend the event vocabulary of Ariadne at run time, and expresss behavior composed of the newly defined events.

Ave/Ariadne is an event-based abstraction based approach for program debugging. However as revealed in the last example, sometimes tracking an error may involve accumulating local state information as well. Currently this requires re-instrumentation of the program to collect the local state information as attributes to our events which is clearly unacceptable. We are currently investigating an integrated approach of debugging, that collects a minimal state information during execution, which will help the users to refine their event-based behavioral description in an incremental fashion by adding state attributes to track down bugs.

# References


[1] G. Alverson, W. Griswold, D. Notkin and L. Snyder. A flexible communication abstraction for nonshared memory parallel computing. *Proceedings of Supercomputing '90*, 1990.

[2] F. Baiardi, N. De Fransesco and G. Vaglini. Development of a debugger for a concurrent language. In *IEEE Transactions on Software Engineering*, SE-12(4):547–553, Apr. 1986.

[3] P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts, Amherst, MA 01003, 1986. Also COINS Technical Report 86–05.

[4] B. Bruegge and P. Hibbard. Generalized path expressions: A high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium in High-Level Debugging*, pages 34-44, 1983.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, 1991.

[6] J. E. Cuny, A. Hough, and J. Kundu. Logical time in visualizations produced by parallel programs. *Proceedings of Visualization '92*, pages 186–193 (1992).

[7] J. E. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The Ariadne Debugger: Scalable Application of Event-Based Abstraction. *SIGPLAN Notices*, 28(12), pages 85–95, 1994.

[8] C. J. Fidge. Partial orders for parallel debugging. *SIGPLAN Notices*, 24(1), pages 183–194, 1989.

[9] R. J. Fowler, T. J. Leblanc, and J. M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *SIGPLAN Notices*, 24(1), pages 163–173, 1989.

[10] G. S. Goldszmidt, S. Katz, and S. Yemini. High level language for debugging concurrent programs. *ACM Transactions on Computer Systems*, 8(4), pages 311–336, Nov. 1990.

[11] P. K. Harter, D. M. Heimbigner and R. King. IDD: an interactive distributed debugger. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 498–506, 1985.

[12] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[13] D. Hembold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2), pages 47-57, Mar. 1985.

[14] A. A. Hough. *Debugging Parallel Programs Using Abstract Visualizations*. PhD thesis, University of Massachusetts, Amherst, MA 01003, 1991. Also COINS Technical Report 91–53.

[15] A. A. Hough and J. E. Cuny. Perspective views: A technique for enchancing visualizations of parallel programs. In *1990 International Conference on Parallel Processing*, pages II 124–132, Aug. 1990.

[16] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–20, March 1990.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[18] L. Lamport. The mutual exclusion problem: Part I-A theory of interprocess communication. *Journal of the Association for Computing Machinery*, 33(2):313-326, April 1986.

[19] R. J. LeBlanc and A. D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 515–522, 1985.

[20] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[21] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[22] V. M. Lo, S. Rajopadhye, M. A. Mohamed, S. Gupta, B. Nitzberg, J. A. Telle, X. X. Zhong. LaRCS: A language for describing parallel computations for the purpose of mapping. Technical Report CIS-TR-90-16, University of Oregon Dept. of Computer Science, 1990.

[23] B. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Notices*, 24(1), pages 141–150, 1989.

[24] L. Snyder. The XYZ abstraction levels of Poker-like languages. *Languages and Compilers for Parallel Computing*, David Gelernter and Alexandru Nicolau and David Padua(eds.), MIT Press, pages 470–489, 1990.