Report Number: WUCS-97-27

1997-01-01

Balancing Consistency and Lag in Transaction-Based Computational Steering

Eileen Kraemer, Delbert Hart, and Gruia-Catalin Roman

Computational steering, the interactive adjustment of application parameters and allocation of resources, is a promising technique for higher-productivity simulation, finer-grained optimization of dynamically varying algorithms, and greater understanding of program behavior and the characteristics of data sets and solution spaces. Tools for computational steering must provide monitoring, visualization, and interaction facilities. In addition, these tools must address issues related to the consistency, latency, and scalability at each of these phases, and must consider the perturbation that results. In this paper we describe transaction-based components for a computational steering system and present an approach that guarantees consistent monitoring and displays,... **Read complete abstract on page 2.**

Balancing Consistency and Lag in Transaction-Based Computational Steering

Eileen Kraemer, Delbert Hart, and Gruia-Catalin Roman

Complete Abstract:

Computational steering, the interactive adjustment of application parameters and allocation of resources, is a promising technique for higher-productivity simulation, finer-grained optimization of dynamically varying algorithms, and greater understanding of program behavior and the characteristics of data sets and solution spaces. Tools for computational steering must provide monitoring, visualization, and interaction facilities. In addition, these tools must address issues related to the consistency, latency, and scalability at each of these phases, and must consider the perturbation that results. In this paper we describe transaction-based components for a computational steering system and present an approach that guarantees consistent monitoring and displays, supports scalable monitoring, and provides the user with the ability to adjust the tradeoffs among lag, consistency and perturbation.

Washington

WASHINGTON · UNIVERSITY · IN · ST · LOUIS

School of Engineering & Applied Science

Balancing Consistency and Lag in Transaction-Based Computational Steering

Eileen Kraemer
Delbert Hart
Gruia-Catalin Roman

WUCS-97-27

June 2, 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

Computational steering, the interactive adjustment of application parameters and allocation of resources, is a promising technique for higher-productivity simulations, finer-grained optimization of dynamically varying algorithms, and greater understanding of program behavior and the characteristics of data sets and solution spaces. Tools for computational steering must provide monitoring, visualization, and interaction facilities. In addition, these tools must address issues related to the consistency, latency, and scalability at each of these phases, and must consider the perturbation that results. In this paper we describe transaction-based components for a computational steering system and present an approach that guarantees consistent monitoring and displays, supports scalable monitoring, and provides the user with the ability to adjust the tradeoffs among lag, consistency and perturbation.

Keywords: computational steering, Query-Based Visualization, transaction, visualization

Acknowledgement: This paper is based upon work supported in part by the National Science Foundation under Grant No. CDA-9619831. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Correspondence: All communications regarding this paper should be addressed to

Dr. Eileen Kraemer
Department of Computer Science
Washington University in Saint Louis
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

eileen@cs.wustl.edu
office: (314) 935-6621
secretary (314) 935-6160
fax: (314) 935-7302

http://swarm.cs.wustl.edu/cgi-bin/papers.cgi

1 Introduction

Computational steering is the online, interactive allocation of resources and adjustment of application parameters. This interactivity can be useful for performance optimization in systems where the demands on resources and the availability of those resources may fluctuate over time. Another application of computational steering is the adjustment of parameters in algorithms for which the execution behavior varies randomly, is dependent upon characteristics of the input data, or for which the the execution behavior is not yet well understood. In the case of modeling and simulation codes, the ability to observe and adjust model parameters in an online fashion allows researchers to terminate unproductive executions early, and to develop intuition regarding interactions among model parameters. Finally, computational steering can serve as a tool for knowledge discovery, allowing viewers to more easily detect the cause-and-effect relationships at work, to localize bugs, and to better understand the behavior of algorithms and characteristics of both the target problem and data set.

Tools for computational steering must provide a monitoring function, some type of display and user interface, and a mechanism for propagating steering actions back to the executing program. In addition, computational steering tools must address issues including the *consistency*, *latency*, and *scalability* of these components, and the *perturbation* associated with their presence. In the following paragraphs, we discuss these factors as they relate to computational steering. In subsequent sections, we present transaction-based monitoring and describe how it deals with these issues.

Steering decisions are typically based on visual presentations of some subset of the computation's current state, a historical display of the computation's behavior, or views of metrics based on the program's performance. As in any endeavor, good decisions require accurate, up-to-date information. However, the distributed nature of the collection process may result in inconsistent views that distort the portrayal of the program's execution. These inconsistent views result from merging multiple streams of information originating from distributed collection points into a single stream without enforcing the causal and temporal relationships that held among the program components that produced the information. Visualizations based on such an out-of-order event stream may be misleading (e.g., a receive event portrayed as occurring prior to its corresponding send) or that fail completely (e.g., an attempt is made to update a graphical object that has not yet been created).

As with the monitoring and display components, consistency is important in the steering component. While some steering actions may be applied at any of the participating processes at any point in the computation, others may be correctly applied only at cer-

tain points in the execution of the process, and still others may require some coordination between processes.

For example, in a parallel simulated annealing algorithm, the goal of the computation is to locate a configuration, defined by the values of a set of variables, that maximizes or minimizes some objective function. Other configurations that may be reached from the current configuration through an application-specific operation known as a move are called *neighbors*. Steerable parameters might include the *neighborhood size*, the number of neighbor configurations to consider at each iteration, the *temperature*, a factor used to calculate the probability of accepting a neighbor configuration as the new current configuration, the *cooling rate*, a factor used to reduce the current temperature from one iteration to the next, and the values that define the configuration itself[1]. Although it might be possible to alter the neighborhood size or cooling rate at any process at any point in the computation with no adverse effect on the correctness of the computation (convergence properties aside), an "on-the-fly" change in the values that define the current configuration could produce an erroneous result if the cost function is evaluated as the values are changing. In this case, it is necessary to apply updates only at "safe" points in the execution of the process.

Continuing with the same example, some parallel simulated annealing algorithms actually distribute components of the configuration's state across multiple processors, and parallelism arises from their cooperation in evaluating the objective cost function. Consider a steering action that invokes a redistribution of components across processors to achieve a better load balance. An uncontrolled invocation of this action might result in a transient loss of some components, causing the computation to fail or produce an incorrect value. It is clear that continued correctness will require some coordination of processes.

Overall latency, or lag, is the elapsed time between the occurrence of an event or the existence of a particular state, and the resulting application of some steering action. Lag has several components[2],[3]. For purposes of this paper we consider *presentation lag* as the elapsed time between the occurrence of an event or the existence of a particular state and the presentation of the associated graphical updates to the user. We consider *steering lag* as the elapsed time between a user's interaction to initiate a steering action and the the completion of the application of that steering action at the target process or processes. Substantial presentation lag may lead the user to employ steering actions that are no longer appropriate. In the same vein, even appropriate steering decisions and actions, based on a consistent presentation of accurate data, may go awry if the steering lag is too great. In either instance, such "behind-the-times" steering actions may result

in degradation, rather than optimization, of program performance or solution quality.

Performance optimization is a primary application of computational steering techniques; thus, minimization of perturbation is an important criterion in the design of a computational steering environment. Unfortunately, attempts to ameliorate problems with consistent data often lead to increased perturbation and lag, and attempts to minimize perturbation of the application program's execution often result in increased lag or inconsistency. For example, the addition of distributed time-keeping techniques to resolve consistency problems typically results in additional perturbation of the program. Selective monitoring, collecting data at only a subset of the processes involved in the computation, may lead to inconsistency. Monitoring data may be buffered and forwarded to visualization component in bulk to reduce perturbation; however, an increase in lag results. While all systems for computational steering perturb the monitored system to some degree, it is desirable that this perturbation be both minimal and predictable, so that users may rely upon their experience with typical lag times in evaluating visualizations and performing steering actions.

Finally, because the target applications for computational steering techniques are large, long-running computations, executing on tens, hundreds, or thousands of processors, it is essential that users be able to monitor, view, and steer the execution at a selected (preferably, dynamically selected) subset of the processes, yet still provide consistent, low-latency displays and steering actions. Further, it is desirable that perturbation at processes not selected for monitoring be minimal.

In the following section we present transaction-based monitoring, define the underlying model of computation, and describe an algorithm that permits the collection of consistent global snapshots, and the presentation of consistent views. We then describe extensions of this algorithms to permit selective monitoring. Such selective monitoring promotes scalability, and helps to reduce perturbation. We present several monitoring schemes with varying consistency guarantees and lag characteristics. Finally, we describe the adjustments that users may employ to balance the tradeoffs among lag, consistency, and perturbation to suit their particular application.

2 Transaction-based Monitoring

Transaction-based monitoring, the monitoring technique employed in the Query-Based Visualization model[4](QBV), is a state-based approach to monitoring and visualization, as opposed to the event-based approach used by many computational steering tools. In this approach, the distributed computation is viewed as a database containing the state

of the individual processes executing across a network. The state of each process changes due to local computations, as a result of message passing activities, as a consequence of process creation and termination, and as a result of the application of steering actions.

Queries provide the mechanism by which the state space is examined and explored. In order to avoid consistency problems, all queries are evaluated logically with respect to consistent global states of the executing system. Classes of queries include *one-time queries*, evaluated once, and *persistent queries*, evaluated after each logical change in the system state. Snapshot algorithms examine collections of local snapshot histories and *transaction* information to produce consistent global snapshots[5]. The queries are then evaluated against these consistent global snapshots.

Efficient snapshot algorithms are central to the success of this approach. The ability to minimize the number of processes required to participate in the construction of the snapshots helps to minimize perturbation. Without loss of generality, our initial implementation employs the PVM library and relies on certain assumptions about the way the computation is structured in order to achieve efficient query processing and snapshot collection.

In the following subsections, we describe a computational model for distributed programs, and present algorithms for the construction of consistent snapshots for both comprehensive monitoring and selective monitoring.

2.1 The computational model

Although the ultimate goal is to permit exploration and steering of arbitrary distributed computations, our current work focuses on a restricted class whose communication patterns are sufficiently structured so as to facilitate a reasonably efficient implementation of the Query-Based Visualization model. The key feature we exploit is the fact that the overall computation can be abstracted to an interleaving of atomic state changes involving one or more processes - by analogy with databases, we call such state transitions *transactions*. Transaction processing applications are a natural choice for obtaining global state information, since their structure matches the logical actions performed by the application. Many multi-phased computations also fall in this category of applications whose structure reflects the logical computation. In some cases, the transaction concept can be superimposed on computations that otherwise execute in a highly unstructured manner.

Underlying this view of distributed computing is the reality of message-based communication via reliable FIFO channels with the added complication that processes may be created dynamically and may terminate at any time. By and large, the application

code need not be written in any special way. The only exception is the need to specify the end of each transaction in each of the participating processes. Identifying the end of a transaction is easy in the case of computations where the communication pattern is known before-hand, such as phased computations or server-client message exchanges, and requires minimal application knowledge to add the transaction annotations. To annotate applications without a known communication pattern one must exploit the specifics of the particular application.

In the remainder of this subsection, we provide a characterization of the computational model we employ, and show its relation to global snapshots. A more formal treatment may be found in [6].

2.1.1 Transactions

A distributed computation consists of a set of processes that work together to achieve a common goal. Each process exports a set of attributes that reflect the state of the process. The process's state changes when an event occurs at the process. The event sequence is a history recording the changes undergone by the process. An event e is characterized by:

- the process with which this event is associated.
- the state of the process immediately after e occurs.
- a local sequence number that reflects the event's location in the process's history of events.
- an event type indicating the nature of the event (i.e., *send, receive, mark* or *local*).

A *local* event represents a state transition within a single process. A *mark* event indicates that the process has completed its participation in the current transaction. A matching *send/receive* event pair is called a communication. In addition, the events *init*, *start* and *stop* (special instances of the events *send*, *receive* and *mark*) denote a request to create a process, the start of a process, and the termination of a process, respectively.

We then view the distributed computation as a set of events, with a partial order corresponding to the happened-before relation, and an equivalence relation that captures the notion that two events are part of the same transaction. A distributed computation is *well-formed* (i.e., an appropriate model for transaction-based monitoring) if it satisfies the following properties:

- at each process, a total ordering of events at that process exists
- computation processes interact only via message-passing
- every send event has a corresponding receive at some other process

- transactions are equivalence classes over the events in the computation
- the send and receive events of a communication belong to the same transaction

As described in [4], computations that satisfy the above properties permit the calculation of equivalence classes, reflecting an ordering of the transactions in the computation such that a transaction **a** happened-before a transaction **b** if and only if there is an event in **a** that occurred before some event in **b**. Transactions form the boundaries in the process executions that we will use to construct consistent global snapshots.

2.2 Snapshot collection

In this subsection we present four algorithms for obtaining online global snapshots. We then compare the algorithms in terms of their consistency guarantees, and effects on lag, perturbation and scalability. No algorithm is a clear best choice for all applications. Rather, the choice of algorithm depends on the computation being monitored and the requirements the user has for the sequence of snapshots shown. All of the algorithms construct snapshots of the computation that include the data requested by the queries that the user has issued. Snapshots are taken at the boundaries of the transactions of the application. The sequence of snapshots constructed provides a view of the application's logical progression.

In each case, we assume the computation is well-formed. Processes are augmented with a reporting mechanism that transmits information about transaction completion and the resulting state of a process to a snapshot manager, a process outside of the distributed computation that assembles the monitored data into logically consistent snapshots. Communication with the snapshot manager is assumed to be message based, reliable, and FIFO, and interactions with the snapshot manager are not considered to be events of the computation.

2.2.1 Comprehensive monitoring

Let us assume that we desire to continuously monitor all processes involved in some distributed computation. The global state can be easily constructed from the information available about each of the events. As we are interested only in the global state between transactions, it is reasonable for processes to locally and incrementally store information that is needed by the snapshot manager, and then to send it once, after a *mark* event, the last event in a transaction. The effect of this is to reduce the number of messages that need to be sent to the snapshot manager. This local gathering of information reduces perturbation(fewer messages, less overhead), but may add to lag because the delivery

of state information to the visualization system is delayed until the completion of a transaction.

Message delivery delays may lead to the situation in which some of the *mark* events do not arrive at the snapshot manager in a timely fashion. However, the FIFO nature of the communication ensures that we can extract a well-formed computation from the *mark* events that the snapshot manager receives. Mark events are maintained in a queue of messages from a process until the event can be used to reconstruct a transaction. Reconstruction of a transaction is possible once the *mark* events for all of the processes that participated have arrived. The snapshot manager keeps the unused *mark*s in queues to ensure that it reconstructs the transactions in the order which they occurred in the computation. In this way the snapshot manager constructs a well formed computation that is a prefix of the actual computation.

To compute the global snapshots the snapshot manager needs to have the process's state information after the transaction, and be able to determine transaction ordering and transaction membership. The snapshot manager has the process's state information available because every process sends its monitored attributes to the snapshot manager when the process completes a transaction. The state information is also tagged with the transaction that generated it. A transaction can be uniquely identified by knowing the process id and the sequence number of the *mark* event that ended the transaction. Given the transaction membership of every transaction, transaction ordering can be inferred. If all transaction memberships are known and the communication is FIFO, then the order that *mark* events are received from a process represents the order in which their respective transactions occurred. This provides sufficient information to reconstruct the transaction ordering.

To compute transaction membership, we require each *mark* to include in its report to the snapshot manager the identity of all the processes that messages have been sent to or received from since the previous *mark*. These are the process's *neighbors* during the transaction. The FIFO communication and the *mark* events' reporting of the process's neighbors during the transaction allow the snapshot manager to determine the transaction membership. The snapshot manager does this by looking at the earliest *mark* event from each process not yet associated with a transaction. We know that this *mark* must be part of that process's next transaction in the computation because of the FIFO communication between the process and the snapshot manager. The snapshot manager then takes the transitive closure over the neighbors reported by the *mark* event; that is, it finds the smallest set of *mark* events that have each other's processes as their neighbors, the transaction membership.

Premature decisions regarding transaction membership are not possible. A missing *mark* x must be associated with a process p that either sent a message to or received a message from one of the processes the snapshot manager already knows to have participated in the transaction, call this process q. Let y be the *mark* associated with q in this transaction. The event y will inform the snapshot manager that process p is part of the transaction and the snapshot manager will wait accordingly for the *mark* event x.

2.2.2 Naive selective monitoring

As the number of processes grows, the perturbation induced by comprehensive monitoring increases as well. Further, the lag associated with the construction of global snapshots increases, not only as the result of the additional load on computation and communication resources, but also because the snapshot manager waits for mark events from slow processes before proceeding with the transaction membership calculation, leading to backups in snapshot processing. Thus, a lighter weight method of collecting snapshots is desirable. Of course, selective monitoring precludes visualizations of global state. However, visualizations based on a few representative processes may be preferable to global views in many situations.

The user initiates selective monitoring by issuing a *query* that describes the subset of processes and state variables of interest. A number of queries, the *active query set*, may be active at one time. Only monitored processes, those involved in the evaluation of the active query set, will communicate state information to the snapshot manager. This approach has a number of nice consequences:

- The incoming message load at the snapshot manager now depends on the size, scope, and transaction granularity of the active query set, rather than on the size and transaction granularity of the entire computation.
- Processes that are not monitored are perturbed very little by the monitoring libraries.
- The increased efficiency of the monitoring software decreases the usage of resources shared with the application (i.e. shared communication links) thus reducing the overall perturbation of the system.

Unfortunately, consistency problems arise if only the monitored processes report *mark* events to the snapshot manager. Consider, for example, Figure 2, which depicts 6 processes and 3 transactions. The processes A, C, and F are the only processes being monitored. Transaction *t2* involves only processes B and D and sends no information to the snapshot manager because neither process is currently monitored. Note that in this situation the snapshot manager can no longer reconstruct the ordering relation between

t1 and *t3*. Note also that in *t3* processes C and F have no direct interactions and, since D and E are not required to report to the snapshot manager, it becomes impossible to solve the membership problem for the transaction *t3*. Reconstruction of the transactions requires that the reporting mark events supply additional information if a consistent order and labeling of the transactions is desired.

2.2.3 Selective monitoring

The *naive selective* algorithm has some desirable properties (i.e. scalable, low latency, and small perturbation), but makes inadequate guarantees to create consistent views. Conversely, the comprehensive algorithm guarantees consistent views, but does not scale well to large applications. The *selective* monitoring algorithm is a hybrid between the two previous algorithms. It guarantees consistent views of a subset of the global state while incorporating some of the advantages of the naive selective algorithm.

In selective monitoring we distinguish between the data collection and transaction labeling parts of the algorithm. The responsibility of the data collection portion is to deliver monitored data to the snapshot manager. Transaction labeling, determining the membership and ordering of a transaction, is carried out by a separate protocol. The data collection aspect of selective monitoring is the same as in the previous two algorithms, so we will focus on the transaction labeling protocol.

A transaction is monitored if and only if it contains an event that is monitored. During selective monitoring, the snapshot manager must be able to reconstruct the membership in each transaction as well as the happened-before relation among the monitored transactions. Since some processes may not report in we can not use the same solution that we did for the comprehensive algorithm. To address the membership question, we require *mark* events that are part of the same transaction to perform the "logical equivalent" of a barrier synchronization during which membership information is collected. To find the membership of a transaction each process performs two actions:

1. Assume the existence of a total ordering over processes. Wait to receive information from all neighbors (direct or transitive) with a higher id that the process has knowledge of. Recall that the neighbors of a process are those processes that it communicated with during the transaction. Transitive neighbors are the processes that a process hears of from its higher-id, direct neighbors during this phase.
2. Send all of the known membership information received about other processes who participated in this transaction, including the information received from other processes, to the process with the highest id lower than its own. If there is no process with a lower id, then this process is the transaction leader.

When the transaction leader has completed action (1) it will know the membership of the transaction, and can transmit this information to the snapshot manager. In fact, this operation is a lighter-weight operation than a barrier synchronization. Consider that a barrier synchronization has two steps - a barrier-in, and a barrier-out. In a barrier synchronization, no process may execute the barrier-out operation until *all* processes involved in the barrier have executed the barrier-in, at which point they may all execute a barrier-out, and continue processing. The operation involved in the transaction protocol has more concurrency, and may be viewed as a hierarchical tree, in which each process is a child node of the highest-value lower-id process that the process knows it has communicated with, either directly or transitively. The leaf nodes report to their parents, and may then continue. Parent nodes may continue when they have received information from all of their child nodes (again, either directly or transitively). The only process required to wait for all other processes is the lowest-id process in the transaction. Note however, that the application is not required to wait at any process, as the transaction protocol runs on its own logical thread, independent of the application. A more thorough discussion of this algorithm can be found in [6].

This algorithm guarantees consistent global snapshots. However, an increase in lag and perturbation as compared to the naive selective algorithm is the price of the consistency. Greater lag times result from the time required for the transaction labeling information to reach the snapshot manager. Increase perturbation results from the use of a light weight protocol among the application processes and having unmonitored transactions send a message to the snapshot manager. Compared to the comprehensive monitoring algorithm, this algorithm provides the same level of consistency, but reduces number of messages sent to the snapshot manager per transaction from a factor of n, the total number of processes, to a factor of m, the number of monitored processes.

2.2.4 Scalable monitoring

If an even more scalable consistent monitoring algorithm is needed then the *scalable* monitoring algorithm can be used. The data portion of this algorithm is the same as in selective monitoring (i.e. a monitored process sends its data to the snapshot manager at the end of the transaction). The transaction labeling protocol is modified so that application processes can keep track of which monitored transactions they are causally dependent upon - these transactions are called the *visible predecessors*. By maintaining the transaction ordering information in the application processes, the snapshot manager does not need to receive any information from unmonitored transactions.

Processes maintain knowledge of which monitored transactions they are causally de-

pendent on through use of a vector clock, with an entry for each monitored process that participated in a transaction that directly causally preceded the current transaction. The value of a vector clock entry is the local time at which the monitored process was causally affected by (typically, received a message from) that visible predecessor. Each process of a transaction knows a portion of the transaction's visible predecessors, but none of them necessarily know all of them. So, in addition to determining the transaction membership, the transaction labeling protocol is responsible for determining the transaction's visible predecessors and communicating the visible predecessors to all participants in the transaction. Figure 2 shows an example of why the visible predecessors need to be communicated to all processes in a transaction. If D does not know that t1 occurred before t2, this information is not available when the visible predecessors of t3 are being computed. When A and F report in, they need to know that t1 occurred before t3, otherwise the snapshot manager might present the transactions in an order inconsistent with the partial order over events.

To propagate the ordering information to all processes in a transaction, we make use of the transaction labeling protocol. Each message sent during the protocol will now include the vector clock representing the transactions that directly causally precede the current transaction. Upon receipt of a protocol message, a process now combines the received vector clock with the vector clock for the appropriate transaction. The transaction leader will know the visible predecessors for the transaction.

If there are no monitored processes in the transaction, the transaction leader sends the visible predecessor vector clock to all of the processes of the transaction. If there are any monitored processes, then the visible predecessor vector clock is sent to the snapshot manager along with the monitored transaction membership and a vector clock representing the monitored processes and their local times is passed to the unmonitored processes. The vector clock sent to unmonitored processes represents its new visible predecessor information. (Monitored processes are able to create their own vector clock containing just their id and local time.) Unmonitored processes now have a third action, which is to wait for the transaction leader to send the visible predecessor information before it can perform action (2) for any subsequent transactions. Recall in the example in Figure 4, process D participated in transactions t2 and t3. If D performed action (2) for t3 before receiving the visible predecessors back from the transaction leader, then D could not include the information that t1 was a visible predecessor.

So, by trading some latency and increase message traffic we increased the scalability of the monitoring system while preserving the consistency guarantees. Improving scalability at the cost of other characteristics is one example of the many trade-offs that can be made

Name	Perturbation	Lag	Message	Scalable	Consistency
Comprehensive	2	1	n	4	Yes
Naive Selective	1	1	m	1	No
Selective	3	3	n+m	3	Yes
Scalable	3	4	2n	2	Yes

Table 1: A ranking of global snapshot algorithms (1=Best, 4=Worst, n=number of processes, m=number of monitored processes)

to customize monitoring algorithms to the needs of the user and the application being monitored.

3 Comparisons and Additional Optimizations

In the previous section we presented four algorithms for the collection of global snapshots in a transaction-based monitoring system, with varying effects on consistency, latency, perturbation and scalability. Table 1 shows a ranking of the algorithms (1=Best, 4=Worst) in terms of their ability to reduce perturbation and lag, and promote scalability and consistency. Also included is an indication of the number of messages per transaction for each algorithm (n=number of processes, m=number of monitored processes). It can be seen from the rankings in table 1 that none of these algorithms is the clear best choice. The user must consider the tradeoffs among these factors when selecting the algorithm that best suits the application at hand.

Additional optimization techniques may be applied to each of these algorithms to further customize and balance the factors listed in the table.

- Bounded Lag: If queues become too full, or measured lag exceeds a threshold value, then then old data can be discarded in favor of more recent data. This can be applied at both the snapshot manager and at the application processes. Lag will be reduce. In some snapshot algorithms, consistency guarantees may be affected. In either case, the visualization may then contain some discontinuities.
- Message Buffering: Local snapshots may be buffered and sent to the snapshot manager in bulk. This increases lag, but reduces message traffic and perturbation.
- Message Piggybacking: Where possible, transaction protocol messages may be held back, to be sent along later with application messages. Again, this will increase lag, but reduce message traffic and perturbation.

The issues we have addressed must be considered by every system for computational

steering. In the following section we present an overview of approaches employed in related tools and systems.

4 Related Work

A number of application-specific steering systems have been designed to address the needs of researchers in atmospheric modeling[7], fluid flow[8] and seismic tomography [9], among others. Although these systems must also address the problems of lag, consistency, scalability, and perturbation, they may do so in an application-specific manner.

In our work, we have considered how these problems may be addressed in a more general manner, independent of the particular computation being steered and the particular set of displays presented. Thus, we discuss below several general tools that have been designed to facilitate the steering process, and describe their approaches to solving these problems.

The CUMULVS system[10] for the steering of PVM programs assumes that the application is structured around a main simulation loop. A data transfer routine is placed in this loop. When this routine is executed, the equivalent of a local snapshot is collected, and marked with an iteration number. Local snapshots may then be combined into global snapshots on the basis of the iteration number. For steering, users may specify a range of iteration numbers during which the desired steering operation may be applied.

The Falcon system for interactive program steering[11] relies on the existence of an ordering filter placed at the point at which the streams are merged to ensure a valid ordering of events collected by the monitoring system. This causality filter[12], is based on the causal relationships between the events in the program. The Falcon implementation attempts to minimize program perturbation through the use of per-thread event buffers, emptied by a local monitoring agent. In this system, *perturbation events* are maintained, that permit users to be aware of and evaluate the effects of perturbation on the visualizations of the program's execution[13].

Later work by the same group has produced Progress(PROGram and REsource Steering System), which supports the addition of steering functionality to multithreaded C programs executing on multiprocessors, through the use of a steering toolkit that provides sensors, probes, and actuators. Unconstrained steering updates may be applied through write probes, while actuators ensure that steering operations are applied at "safe points" in a particular process. No facility for coordination of updates across processes has yet been implemented.

The Magellan steering system[14], also from Georgia Tech, applies a language-based

approach to control multithreaded, asynchronous steering servers that cooperatively steer applications, and addresses many of the issues presented in this paper, but in the context of event-based monitoring.

Debuggers may used to provide some of the same functionality as computational steering tools, as in Dynascope[15] However, the level of perturbation associated with debuggers is typically quite high, and consistency of update is left entirely to the user.

Also of interest are shared-memory and dataflow models for computational steering. The VASE system[16] was developed for the steering of SIMD computers. Shared global state simplifies the consistency issue in particular, as consistent global snapshots and steering actions may be achieved by momentarily blocking all processes except the steering or monitoring process. Systems such as SCIRun use dataflow architectures for steering and visualization[17], characterized by large-grain steering control.

5 Summary and Future Work

Tools for computational steering must provide a monitoring function, some type of display and user interface, and a mechanism for propagating steering actions back to the executing program. In addition, computational steering tools must address issues including the *consistency*, *latency*, and *scalability* of these components, and the *perturbation* associated with their presence. In this paper, we have discussed these factors, presented transaction-based monitoring, and described several monitoring algorithms that permit users to prioritize the effects of the above factors.

In our work, we address the issues of consistency, latency and feedback in the visualization component. Due to space considerations, we have not included this component in our discussion. Descriptions of this component may be found in [18, 19, 4, 20].

We are currently developing algorithms for the steering component. Most current steering systems either apply steering updates on a strictly local basis, or force the processes to synchronize. Between these two alternatives are many unexplored options for using steering actions in a distributed environment. We plan to map out this uncharted territory, by characterizing the different approaches available, and determining appropriate approaches for different environments. Ultimately, this will provide the basis for a rich set of tools for designers and maintainers of distributed applications.

Aside from those familiar with distributed computations, steering can benefit end-users of distributed computations. Computations are frequently distributed solely for the purpose of speeding up the computation. In these cases it may not be apparent to the user how the application is logically progressing and how the computation is

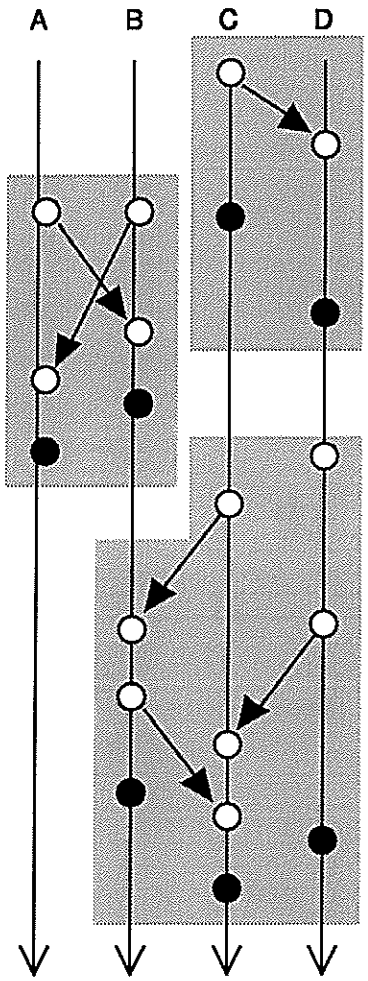

Figure 1: A portion of a well-formed computation. White circles represent application events, including sends and receives. Black circles represent mark events. Shaded areas represent transactions.

distributed across the system.

Tools to support these users and applications must be able to apply steering actions at an arbitrary subset of the computation in a way that updates the application consistently. One approach we are investigating is the use of optimistic steering actions, where steering adjustments will be made and then the consistency is later checked. Violating processes will be stopped and rolled back so that a consistent steering action can be applied. It may also be possible to identify classes of applications that may have efficient consistent steering mechanisms, similar to the way CUMULVS takes advantage of the loop driven nature of the applications it monitors to efficiently create consistent snapshots.

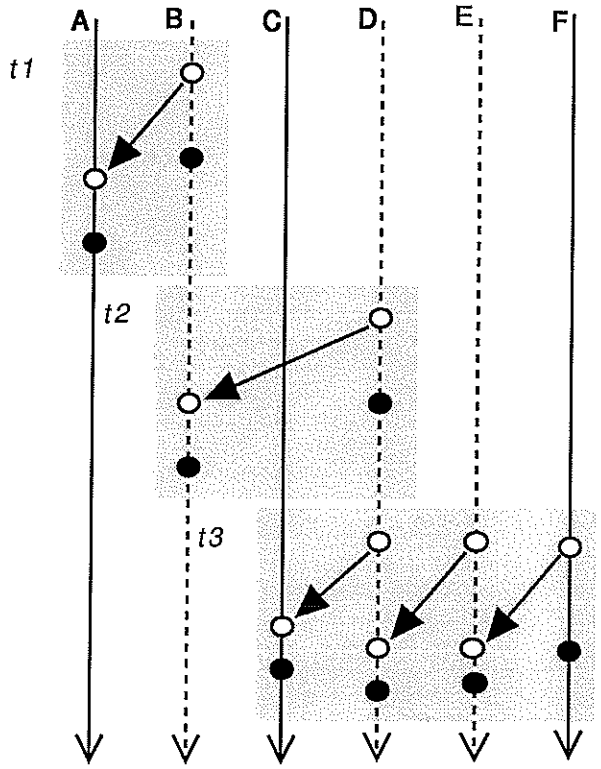

Figure 2: A distributed computation where only processes A, C, and F are being monitored. The local information available at their mark events is inadequate to reconstruct the transaction ordering and membership.

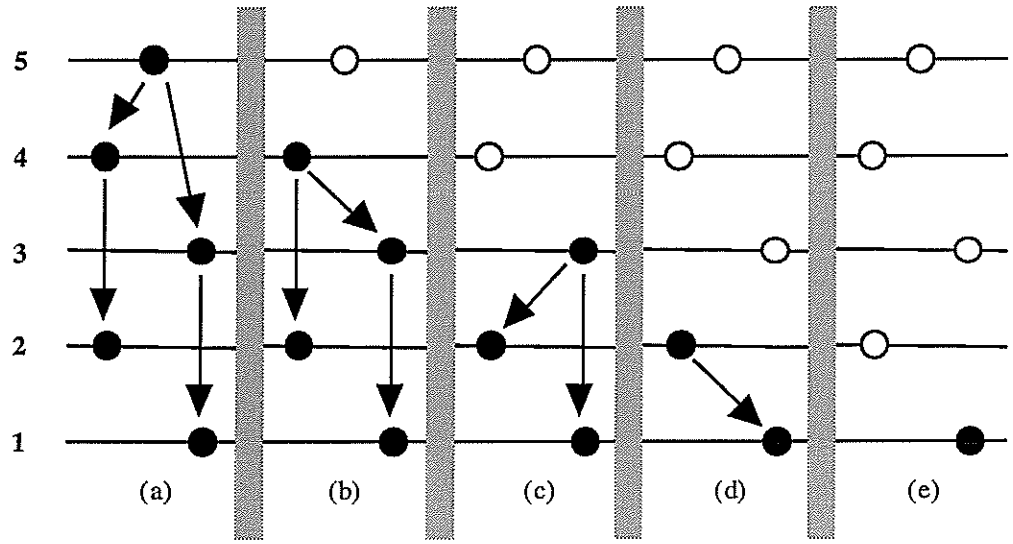

Figure 3: Each process is represented as a circle, their height is based on their ids. A solid circle indicates that the process has not yet completed action (2). a) The transaction has just completed and no actions have been taken yet. b,c,d) The maximal process sends its information to the next lowest process that it knows of. e) Process 1 is the transaction leader since it knows of no process lower than itself and has heard from all of its neighbors.

References

[1] E. Kraemer and J. Wallis, "Interactive steering of simulated annealing," in *Proceedings of the SPDP'96 Workshop on Program Visualization and Instrumention*, (New Orleans), Oct. 1996. To appear.

[2] J. Vetter and K. Schwan, "Models for computational steering," in *Proceedings of the International Conference on Configurable Distributed Systems*, 1996.

[3] V. E. Taylor, J. Chen, T. L. Disz, M. E. Papka, and R. Stevens, "Interactive virtual reality in simulations: Exploring lag time," *Computational Science and Engineering*, vol. 3, no. 4, 1996.

[4] D. Hart, E. Kraemer, and G.-C. Roman, "Query-based visualization of distributed computations," in *Proceedings of the 11th International Parallel Processing Symposium*, (Geneva, Switzerland), Apr. 1997.

[5] K. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[6] D. Hart, E. Kraemer, and G.-C. Roman, "Query-based visualization," Tech. Rep. WUCS-96-23, Washington University, Department of Computer Science, St. Louis, MO, Dec. 1996.

[7] Y. Jean, T. Kindler, and et al., "Case study: An integrated approach for steering, visualization, and analysis of atmospheric simulations," in *Proceedings, Visualization '95*, Oct. 1995.

[8] P. Woodward, "Interactive scientific visualization of fluid flow," *Computer*, vol. 26, no. 10, pp. 13–25, 1993.

[9] J. Cuny, R. Dunn, and et al., "Building domain-specific environments for computational science: a case study in seismic tomography," in *Proceedings, Workshop on Environments and Tools for Parallel Scientific Computing*, 1996.

[10] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "CUMULVS: Providing fault-tolerance, visualization, and steering of parallel applications," *SIAM*, Aug. 1996.

[11] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu, "Falcon: on-line monitoring and steering of large-scale parallel programs," in *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, (McClean, VA), pp. 422–429, Feb. 1995.

[12] E. Kraemer, "Causality filters: A tool for the online visualization and steering of parallel and distributed programs," in *Proceedings of the 11th International Parallel Processing Symposium*, (Geneva, Switzerland), Apr. 1997.

[13] G. Eisenhauer, W. Gu, E. Kraemer, K. Schwan, and J. Stasko, "Online display of parallel programs: Problems and solutions," in *Proceedings, International Conference on Parallel and Distributed Processing Techniques and Applications*, (Las Vegas, NV), July 1997. To appear.

[14] J. Vetter and K. Schwan, "High performance computational steering of physical simulations," in *Proceedings of the 11th International Parallel Processing Symposium*, (Geneva, Switzerland), Apr. 1997.

[15] R. Sosic, "A procedural interface for program directing," *Software: Practice and Experience*, vol. 25, no. 7, pp. 767–787, 1995.

[16] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber, "VASE: The Visualization and Application Steering Environment," in *Proceedings of Sumpercomputing '93*, (Portland, OR), pp. 560–569, Nov. 1993.

[17] S. Parker and C. Johnson, "Scirun: A scientific programming environment for computational steering," in *Supercomputing '95*, 1995.

[18] G.-C. Roman, K. C. Cox, D. Wilcox, and J. Y. Plun, "Pavane: a system for declarative visualization of concurrent computations," *Journal of Visual Languages and Computing*, vol. 3, pp. 161–193, June 1992.

[19] J. T. Stasko and E. Kraemer, "A methodology for building application-specific visualizations of parallel programs," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 258–264, June 1993.

[20] E. T. Kraemer and J. T. Stasko, "Accurate and informative portrayal of concurrent executions," *Concurrency*, 1997. In submission.