AN OPTIMISTIC APPROACH TO COMPUTATIONAL STEERING

by

DAVID WADE MILLER

(Under the direction of Eileen Kraemer)

ABSTRACT

Computational Steering is the online, interactive allocation of resources and adjustment of application parameters. Few computational steering systems support the coordinated steering of multiple processes. Of those that do provide such support, our system is unique in its optimistic approach; other systems take a conservative approach. Because this requires global synchronization considerable perturbation can arise. We focus on optimistic steering, which does not require global synchronization before a steering event may take place. To achieve this requires not only the ability to determine the consistency of steering transactions but also the ability to correct any inconsistencies that may occur. To address these issues, we have developed algorithms for consistency detection and a steering system that has the ability to correct inconsistencies through computational rollback and re-execution. Presented in this thesis are both the details of our steering system and a performance analysis of that system.

INDEX WORDS: Checkpoint, Conservative Steering, Consistent Cut, Consistent Steering, Consistent Transaction, Happened Before, Message Logging, Optimistic Steering, Program Event, Program Transaction, Re-execution, Rollback, Steering Event, Steering Transaction

AN OPTIMISTIC APPROACH

TO COMPUTATIONAL STEERING

by

DAVID WADE MILLER

B.S., The University of Georgia, 2000

B.S., The University of Georgia, 2001

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial

Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA
2002

AN OPTMISTIC APPROACH

TO COMPUTATIONAL STEERING

by

DAVID WADE MILLER

ACKNOWLEDGEMENTS

I would like to thank Dr. Eileen Kraemer for all of her support and guidance over the past several years which has helped me to complete this phase of my life and academic career. Also, I would like to thank my parents for their continued support throughout my life, always encouraging me to pursue my goals. Finally, but certainly not least, I want to thank my wife, Mandy, for her undying and ever devoted love and support, without which none of this would have been possible. She has been my stability even at a time when she too has been under enormous stress and pressure as she pursues her dreams at UGA's School of Law. Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND LITERATURE REVIEW

Overview of Computational Steering

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 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 cause-and-effect relationships, 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, a user interface with visualizations capabilities, and a mechanism for propagating steering actions to the executing programs. To this end, we have developed an Exploratory Visualization (EV) and Interactive Steering (IS) system. Through this system, the user may monitor attributes and variables of distributed computations via queries and visualizations and may dynamically manipulate program variables or adjust resource allocation. Such a system must address issues including consistency, latency, and overall computational perturbation. This thesis will focus on the computational perturbation as it relates to the overhead introduced by the Interactive Steering (IS) portion of the system.

Within the IS system, a global state change resulting from each user's steering request is a called a steering transaction. A steering transaction may be decomposed into steering events, modifications of the local state of one process. Criteria for the correctness of steering changes to the executing system can vary considerably depending on the type of change to be made. Some steering changes may be applied at any of the participating processes at any point in the computation. However, if a steering transaction is intended to update critical control parameters or change the global configuration, causal consistency is often required to maintain the correctness of the computation. That is, all local steering changes must be applied concurrently across all participating processes.

Globally consistent steering updates typically require that a computation reach quiescence [2, 11] before a steering change is applied; the computation blocks before a consensus decision is made and steering changes are applied. This is a nontrivial process in a distributed, asynchronous environment in which centralized control of the computation is required, and can result in considerable perturbation of the computation and significant steering lag [7]. In contrast to this conservative or "pessimistic" approach, we present another approach called *optimistic steering*. In optimistic steering, the system invokes each steering event in the steering transaction at the respective process without concern for or knowledge of the state of any other process. Therefore, the consistency of the global state of the computation, and specifically, the consistency of the steering transaction, must be checked. If the consistency is verified, the computation continues. If the steering transaction is found to be inconsistent, then the earliest time at which the steering event could be consistently applied at each process is calculated. The system must then enter a recovery state, rollback to the state before the original steering change(s) occurred, re-execute forward to the consistent time while replaying any messages that may have been logged, apply the steering changes, and then continue re-executing in recovery until all message logs have been emptied, after which normal execution may resume.

The optimistic approach is expected to perform well in the case that steering is relatively rare and that the processes of the computation tend to remain roughly synchronized because of coordination at the application level. This is often the case, as processes typically wait for messages from one another or synchronize at barriers. In this case, the next steerable points will most likely be consistent since processes will receive their steering request in the same global transaction.

Accordingly, while the overhead of state saving and some logging will be incurred for each steering transaction, rollback and re-execution will be incurred only in the case of inconsistent steering, and both logging and re-execution will be of limited scope and duration.

The remainder of this thesis is organized as follows. The proceeding subsection will provide a brief literature review covering both the causal dependencies found in distributed computations and related computational steering research. My research into causal dependencies found in distributed environments is the basis for the consistency detection algorithm that will be presented in next the chapter. Chapter 2 will present a conference paper detailing the points of consistent steering and an algorithm to detect such. Next, Chapter 3 presents a conference paper in submission that provides not only the implementation details for our IS system but also performance testing results and an evaluation of those results. Finally, Chapter 4 provides conclusions to this research.

Literature Review

Causality is fundamental to many problems in distributed computing. For example, determining a consistent global snapshot of a distributed computation [1, 4] requires finding a set of local snapshots such that the causal relation between all events included in the snapshots is respected in the following sense: if e' is contained in the global snapshot formed by the union of local snapshots, and $e \rightarrow e'$ holds, then e must also be included in the global snapshot. Thus, the notion of consistency in distributed systems is basically an issue of correctly reflecting causality. Many important applications of causal consistency are summarized by Schwarz and Mattern in [18].

An important characteristic of distributed systems is that there is no global clock. Consequently, ordering the events in a distributed system can be challenging. Lamport [9] introduced an efficient mechanism called logical clocks for totally ordering the events in a distributed system, but the mechanism is not sufficiently powerful to allow concurrent events to be identified. Mattern [12] and Fidge [3] independently developed vector clocks, which precisely capture the causal ordering between distributed events. The main difference between Mattern and Fidge vector time schemes and ours is the way in which the logical time of a process is measured. Under the Mattern and Fidge schemes, the logical time of a process is measured in "number of past events" at that process. However, under our scheme, the logical time of a process is measured in "number of past local transactions" at that process.

The Z-path and Z-cycle notion, introduced by Netzer and Xu [14], generalizes the notion of a causal path of messages defined by Lamport's happened-before relation [9]. A local checkpoint is useless iff it is involved in a Z-cycle. In the transaction-based computational model, if a checkpoint is taken, all participant processes in a global transaction take the checkpoint at the end of transaction. In this way, no local checkpoint will be involved in a Z-cycle because there is no message in transit at the end of transaction.

The notion of computational steering is not a unique idea specific to this research [13]. Rather, as you will see, our optimistic approach to computational steering is quite unique in that it allows multiple processes to be steered simultaneously and does not require global synchronization before such steering. The later is a result of our continued research into the causality found in distributed computing and the extensions thereof to verify steering consistency, detection steering inconsistencies, and calculate the earliest consistent cuts (times) to rectify inconsistencies.

An early computational steering environment was VASE, the *Visualization and Application Steering Environment,* developed at the University of Illinois [6, 8]. The VASE system requires special annotation to existing Fortran code and permits the user to alter the values of "key" parameters and to add code at "key" points. However, VASE does not support the coordinated steering of multiple processes. SCIRun supports computational steering in a multithreaded application that runs on a single multiprocessor machine [15, 16]. However, it assumes that the underlying program consists of a number of separate modules. The SCIRun system generates a script that controls the invocation of these modules. The steering process involves altering these scripts—thus, changes occur only between modules, not within modules. The script itself is executed sequentially. No support for coordination of distributed changes is required. Progress [19] and its successor Magellan [20] also provide interactive environments for computational steering. Both these systems were designed to run on multiple multiprocessor machines. Progess does not support coordinated steering of multiple processes. Magellan was extended to support such coordinated steering of multiple processes but requires synchronization points be placed in an application. Before a steering change can take place the application must first halt. Thus, Magellan can be said to support the conservative approach to the interactive steering of distributed computations. CUMULVS was developed at Oak Ridge National Laboratory to support the monitoring and steering of distributed computations [5]. To allow steering, the user interface process creates a loosely synchronized connection with the application which guarantees that all tasks apply the steering updates at the same time or point in the application, also falling into a conservative steering model. Yet another computational steering environment is the VIPER project [17]. VIPER is based on a client/server/client architecture. One client is the parallel computation, the other client is the visualization unit, and the server acts as a governing body for both information and data extraction and steering application. Each application has synchronization points at which time the server has the ability to consistently apply the steering changes requested by the user. Finally, the CSE environment provides a computational steering environment similar to those already described [10, 21]. In this system, there exist data manager and satellite worker processes. The data manager is responsible for gathering the monitored data, all communication, and application of steering changes. Like the other environments that support simultaneous steering events, this system also requires its source code be annotated with special synchronization variables. During synchronization, the data manager can consistently apply the steering changes.

CHAPTER 2

ON-THE-FLY CALCULATION AND VERIFICATION OF CONSISTENT STEERING TRANSACTIONS♣

♣ Miller, David W., Jinhua Guo, Eileen Kraemer, and Yin Xiong. 2001. *SuperComputing 2001*. Reprinted here with permission of publisher.

On-the-Fly Calculation and Verification of Consistent Steering Transactions[*]

David W. Miller, Jinhua Guo, Eileen Kraemer, Yin Xiong
{miller, jinhua, eileen, xiong}@cs.uga.edu
Department of Computer Science
The University of Georgia

Abstract

Interactive Steering can be a valuable tool for understanding and controlling a distributed computation in real-time. With Interactive Steering, the user may change the state of a computation by adjusting application parameters on-the-fly. In our system, we model both the program's execution and steering actions in terms of transactions. We define a steering transaction as consistent if its vector time is not concurrent with the vector time of any program transaction. That is, consistent steering transactions occur "between" program transactions, at a point that represents a consistent cut. In this paper, we present an algorithm for verifying the consistency of steering transactions. The algorithm analyzes a record of the program transactions and compares it against the steering transaction; if the time at which the steering transaction was applied is inconsistent, the algorithm generates a vector representing the earliest consistent time at which the steering transaction could have been applied.

[*] This material is based on work supported by the National Science Foundation under Grant No. 9996082.

Keywords: *Program Transaction, Steering Transaction, Consistent Transaction, Consistent Steering, Program Event, Steering Event, Happened Before, Consistent Cut*

1. Introduction

Distributed computing is a powerful tool for performing large, computationally intensive tasks quickly by sharing the work across a network of workstations. However, the complexity of these programs can make it difficult for users to fully understand the runtime behavior of many of these computations [15]. The ability to observe controlled executions, interact with, and "steer" a running computation can aid understanding. We have developed an Exploratory Visualization (EV) system that allows a user to pose queries and visualize program data in a real-time fashion. Through this system, the user may monitor attributes and variables of the distributed computation. In addition, we are developing an Interactive Steering (IS) environment through which users may dynamically manipulate program variables or adjust resource allocation.

Consider a system for performing molecular mechanics calculations that contains modules for functions including distance geometry calculation, energy minimization, and free energy perturbation calculation, with the overall goal of 3D protein structure determination. The user might begin with an input file containing the approximate pairwise distances between atoms, as produced by an NMR study. An initial pass with a distance geometry model would produce a rough 3D structure. This structure would then be refined during a long-running energy minimization phase.

The standard protocols involved in this calculation may produce a structure that is overconstrained in one area and underconstrained in another area, or become trapped in a local energy minimum and fail to produce a reasonable 3D structure. The user might wish to visualize the intermediate results of these calculations and interact to turn constraints off or down in some areas of the molecule, and to apply or increase constraints in another area. It might be desirable to phase in the constraints rather than apply them all at once. (Note: a variety of ad-hoc protocols exist for the phasing-in of forces, but no one protocol is universally accepted.) In addition, the user might wish to load balance or make other performance adjustments during this long-running energy minimization phase.

The IS environment would allow the user to view the 3D structures, the state of the constraints, graphs of the change in total energy as the minimization runs, as well as performance statistics, and permit the user to select from a variety of protocols, to write a "steering agent" to implement a new protocol based on the data gathered from the energy calculations, or to rebalance the computational load by moving the computation associated with some atoms from one processor to another.

Both the visualizations and the steering activities rely on consistency. In the case of visualization, consistency guarantees that the visualization represents a valid state of the computation. In the case of steering, consistency guarantees that the steering operations are applied in a way that maintains the correctness of the computation(i.e., are causally consistent). For example, when moving an atom from one processor to another, consistency would guarantee that the energy calculation included each atom once and only once, that the atom being moved would not be either excluded or double-counted in the energy calculation. More subtly, consistency would guarantee that the new constraints would be phased in at all targeted atoms in the same way, unaffected by differences in speed of communications or processing at the participating processes. In this paper, we address the problems of detecting inconsistency, verifying consistency, and calculating consistent states.

2. Exploratory Visualization (EV) Environment

Given the size and complexity of distributed computations, it is important to have a presentation that provides a simple, accurate, and flexible view of an execution. To simplify the problem, in the EV environment, the overall computation is abstracted to an interleaving of atomic state changes involving one or more processes – by analogy to databases, such state changes are referred to as *transactions*. From a computational standpoint, transaction processing applications are a natural choice for obtaining global state information, since their structure matches the logical actions performed by the application [6]. For example, a money transfer may involve two processes located at different points of the network. It is desirable to treat the debit to one account and credit to another account as an atomic operation on the state of two bank accounts. Many multi-phased computations also fall into this category of applications whose structure reflects the logical computation. For example, the N-body problem employs alternating phases of communication and computation in which information about the state of neighboring regions is exchanged, and the forces on each body and the body's new position in space is then calculated. These phases may be modeled as transactions. Further, the transaction concept can often be superimposed on computations that otherwise execute in an unstructured manner.

The formal definition of transaction can be defined as follows:

Definition A transaction relation is an equivalence relation satisfying the following conditions:

1. A local transaction boundary is specified explicitly by end of transaction (EOT) annotations in an application program. A local transaction is a sequence of events between EOTs (or between the start of the program and the first EOT).
2. Two local transactions of different processes belong to the same global transaction if one local transaction sends at least one message to the other local transaction.
3. A global transaction consists of all local transactions in the same equivalence class.

We then view the local computation of a process as a sequence of local transactions and a distributed computation as a set of partially ordered global transactions.

The EV environment is a monitoring system that allows users to view and control a distributed computation in real-time. At present, the system supports PVM, MPI, and socket communication. Use of the EV system with a distributed computation requires that all communication calls be replaced by their EV counterparts and that the code be annotated with two types of statements: *watch* and *end-of-transaction*. The *watch* statement is used to indicate the variables of possible interest for monitoring or steering, and the *end-of-transaction* statement is used to logically group the code into blocks of computation, transactions, that represent logical actions performed by the application. Currently, this annotation is performed manually; interactive tools that largely automate this process could be implemented in a straightforward manner.

3. The Pathfinder Architecture

The EV environment is viewed as a three part architecture, known as the Pathfinder Architecture [5], composed of Interaction Managers (IM), a Snapshot Manager (SM), and a User Interface (UI), seen in figure 2.1.

The IM is composed of communication layers between the process and its communication environment. The IM implements a transaction labeling protocol. The IM collects information (local snapshots) from the processes that participate in each transaction, as well as the communication relationships between processes (membership), and forwards that information to the SM, which then calculates the dependence relationships between transactions (ordering). Processes that communicate with one another during the execution of a logical block of code that forms a transaction are considered members of the same transaction. However, each process typically knows only of its neighboring processes, those with which it directly communicated. Based on the transitive communication patterns, the complete membership within a transaction can be determined [18]. An example is seen in figure 2.2.

At the SM, globally consistent snapshots are generated based on the local snapshots from the IMs and the transaction labeling information. This labeling information may consist of neighbor lists, message counts, vector clocks, or other means [18]. In this paper, we use the vector clock approach as our basis for discussion. The transaction labeling information relies on a set of vector clocks that encode the inter-process communication that occurred during the represented transactions. This information can be used to group local snapshots into global snapshots that represent the global state of the computation at one instant in time [18]. Note that the existence of these vector clocks for purposes of visualization provides the foundation for the relatively low-cost addition of consistent steering.

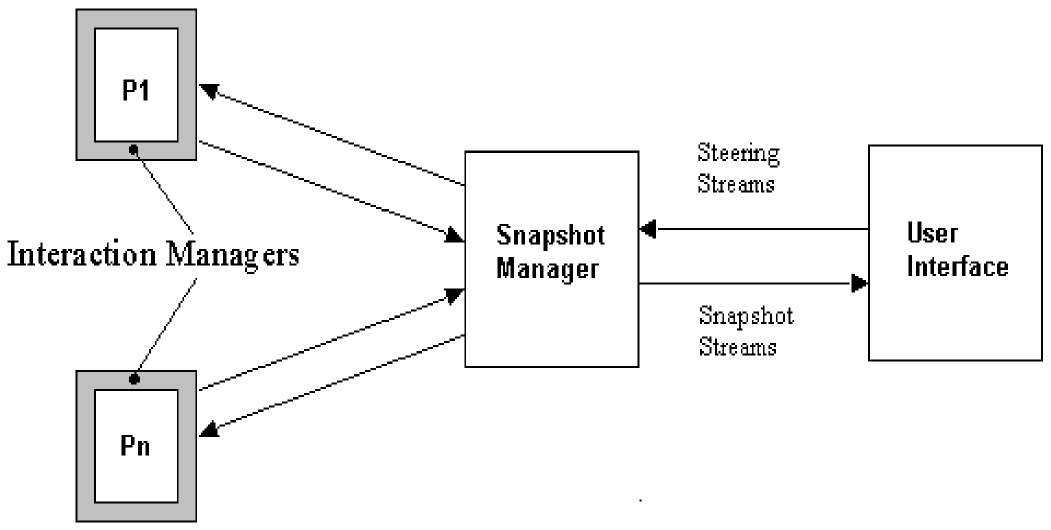


Figure 2.1 – Pathfinder Architecture
Application processes are "wrapped" in Interaction Manager libraries. The Snapshot Manager merges local snapshots into global snapshots. Contents of the global snapshots are visualized at the User Interface

Finally, the global snapshots are sent to the UI to be visualized. Through the UI, the user may pose queries to the system in order to gather application-specific and system-specific values to aid in overall understanding. The UI also provides the interface through which users may directly steer the distributed computation in real-time. The steering request will be issued directly to the SM for processing.

Once the SM receives a steering command, it will issue a steering request to the IM at each process involved in the steering action. Each participating IM will report back to the SM the local process time at which the steering event occurred. The SM will use the event times to form a vector timestamp for the steering transaction. Based on the TLP information already present at the SM, together with steering transaction’s vector time, the SM can determine if a steering command was applied consistently or not. The algorithm for determining consistency of steering transactions is the focus of this paper.

To illustrate the notion of consistent versus inconsistent steering actions, assume the energy minimization calculation described in the Introduction, and imagine that processors 1 and 2 are heavily loaded, while processor 4 is lightly loaded. The user issues a steering command, directing that some atoms be transferred from processors 1 and 2 to processor 4. Figures 2.3 and 2.4 illustrate the times at which the steering events of this steering transaction occurred at each involved process. In figure 2.3, since all the steering events took place either fully before or fully after each represented transaction, the steering command was applied consistently. However, in figure 2.4, the steering event for processes P1 and P2 occurred before transaction Pt1, but the steering event for process P4 occurred after transaction Pt1. Thus, an energy calculation performed during Pt1 might exclude the transferred atoms from consideration in the total energy, resulting in an incorrect energy calculation. In other words, because the steering events did not occur fully before or after each transaction, the steering command was applied inconsistently. Section 5.1 will expand upon the notion of *consistent steering*.

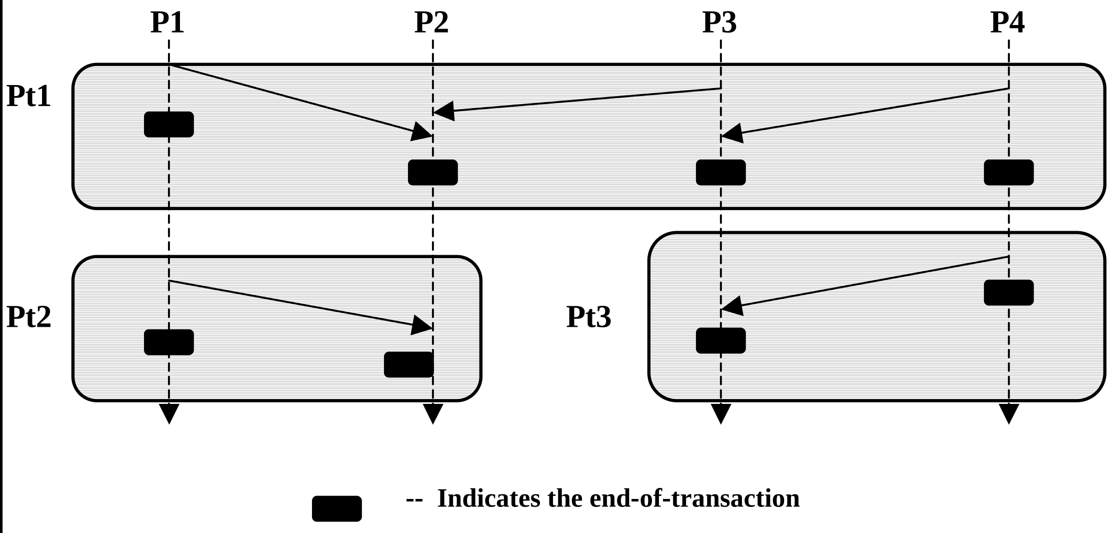


Figure 2.2 – Based on the end-of-transaction events and the communication pattern, three program transactions, Pt1, Pt2, and Pt3, can be formed.

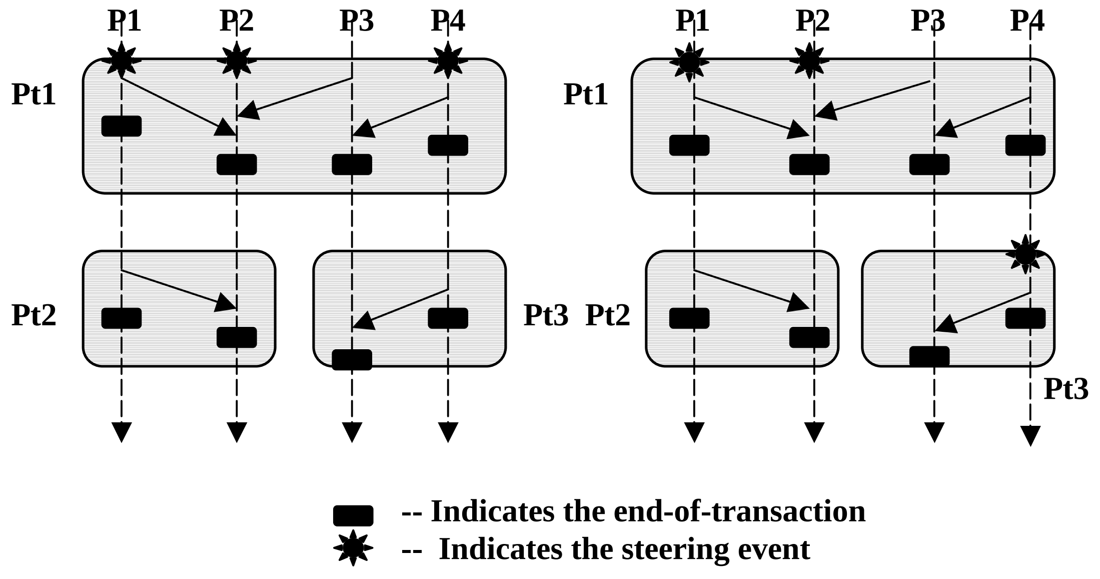

P1
P2
P3
P4
Pt1
Pt2
Pt3
-- Indicates the end-of-transaction
-- Indicates the steering event

Figure 2.3 – Consistent Steering Transaction

Figure 2.4 – Inconsistent SteeringTransaction

4. Model and Definitions

A *distributed computation* consists of a set of dynamic processes that work together to achieve a common goal. In our system, each process exports two sets of attributes: one that reflects the subset of the process state available for monitoring and one that reflects the subset of the process state that is available for steering. The state of the process changes when a program event occurs or when a user-specified steering event occurs. *Program events* of interest are *sends, receives,* and events that mark the end of a process's participation in a *transaction,* known as *end-of-transaction* events. *Steering events* are those actions directly issued by the user to make changes to the state of a computation. The change may be specified to occur at each process in the computation at which a variable resides or may instead affect multiple processes.

Any distributed computation can be decomposed into sets of program events. A *program transaction* is a set of program events collected at one or more processes and representing the same logical point in the program's execution. The events in the set are related through a direct or transitive communication pattern. The time of a program transaction is represented as a vector in which each process involved in a transaction has a timestamp representing its local time when it participated in the transaction. Program transactions are deemed *consistent* if

- the program events within the set belong to one and only one transaction,
- all related send-receive events are in the same transaction,
- for two program transactions Pt1 and Pt2, if the local events at one process in Pt1 occurred before the local events in the same process in Pt2, then, in every other process, all the local events of Pt1 must have occurred before all the local events of Pt2.

Finally, a group of related steering events can be organized into a set, a *steering transaction,* that represents the set of steering events resulting from a single request.

These *transactions* are represented by vectors containing the local timestamps of the processes when they participated in the steering actions.

5. Optimistic versus Pessimistic Steering

Interactive steering may be implemented in two ways: as pessimistic steering or as optimistic steering. Pessimistic steering requires that a computation reach quiescence [2, 10] before a steering change is applied. This is a nontrivial process in a distributed, asynchronous environment in which centralized control of the computation is required. This model can result in considerable perturbation of the computation and significant steering lag [8].

In optimistic steering, the system receives a user-initiated set of steering events, grouped into a steering transaction, to make some application-specific modification. The system invokes each steering event in the steering transaction at the respective process without concern for or knowledge of the state of any other process. Therefore, the consistency of the global state of the computation, and specifically, the consistency of the steering transaction, must be checked. If the consistency is verified, the computation continues. If the steering transaction is found to be inconsistent, the computation must roll back to its state prior to the application of the steering change. In our approach, a new steering transaction is calculated; the process then executes forward to the new steering transaction time, applies the steering changes, and continues under normal execution. In this paper, we address the problem of verifying the consistency of a steering transaction, and, if needed, calculating a new consistent steering transaction.

Note that other systems for interactive steering typically do not address the problem of coordinated, distributed changes to general computations(Falcon[5]), perform steering only between iterations of some main loop(Cumulvs[14]), or leave the problem of consistency to be addressed by the programmer on a per-object basis (Magellan[16], Progress[17]). This third approach, however, provides the opportunity to go beyond causal consistency, and to perform other types of consistency checking as well. Note also that the work described here is part of a larger investigation of optimistic steering**.** Whether optimistic steering is preferable to these other approaches, and if so, under what circumstances, remains to be seen and is an important element of our ongoing work.

5.1 Optimistic Steering

Let m = # processes participating in a distributed computation.

Let u, v be vectors of dimension m, each element a local timestamp of a process.

(1) $u \leq v$ iff $u[k] \leq v[k]$ for $k = 1, \ldots, m$

(2) $u < v$ iff $u \leq v$ and $u \neq v$

(3) $u \parallel v$ iff $\neg (u < v)$ and $\neg (v < u)$ [15].

That is, in cases (1) and (2), when applying the causality relationship defined by Schwarz and Mattern [15], the transaction represented by vector *u happened before* the transaction represented by vector *v* (*u* < *v*) if each process in *u* has a local timestamp less than or equal to that of the corresponding process in *v*, but the vectors are not identical. While, in case (3), the transaction represented by vector *u* does not *happen before* the transaction represented by vector *v*, nor does the transaction represented by vector *v* *happen before* the transaction represented by vector *u*. That is, the transactions correlating with vectors *u* and *v* are concurrent, represented
by $u \parallel v$.

In our system, a steering transaction may involve a subset of processes, with the result that vector elements representing non-participating processes will be undefined. For example, figure 2.5 contains an initial vector representing a steering transaction in which process P2 participated at its local time 2 and process P4 participated at its local time 3. However, processes P1 and P3 did not participate, and, therefore, their entries are undefined. In addition, for participating processes, a steering event at time *t* is assumed to have happened prior to a program event at time *t*. For example, in figure 2.5, the program transaction shows process P2 participated at its local time 2 and process P3 participated at its local time 4. In both the steering transaction and the program transaction, process P2 has a corresponding timestamp of 2. For the purposes of our algorithm, it is assumed that steering transactions are applied immediately prior to program transactions with the same timestamp. Therefore, a steering event is said to *happen before* a program event with the same timestamp. For example, in figures 2.3 and 2.4, the steering event and program event representing the send of a message during Pt1 are both recorded with the same timestamp for process P1.

To correctly represent the transitive dependencies between a steering transaction and a program transaction, we must create an updated vector clock for the steering time. The elements representing the non-participating processes in the steering transaction are updated to represent the time of the earliest program transaction after the steering transaction at which those processes could have been affected. Figure 2.5 shows the update of the steering transaction that indicates process P3 was affected by the steering transaction at local time 4.

We define *consistent steering* such that a steering transaction represented by vector *v* is consistent if and only if it is not *concurrent* with any program transaction represented by vector *u*, denoted $\neg(v \parallel u)$. In section 6, the algorithm presented iterates over each program transaction beginning with the most recent and verifies that it is not concurrent with the steering transaction being checked. If during this process, the steering transaction is found to be concurrent with any program transaction, the algorithm will calculate the earliest, non-concurrent steering transaction that can occur after the present steering transaction

P1	P2	P3	P4
	2		3

Steering Transaction

P1	P2	P3	P4
	2	4	

Program Transaction

P1	P2	P3	P4
	2	4	3

Updated Steering Transaction

Figure 2.5 – Demonstrates the updating of a steering transaction to indicate an indirect affects a steering

6. Algorithm for Consistency Verification

6.1 Idea behind Algorithm

In order to determine the consistency of a steering transaction, the system maintains a list of vectors representing a partial ordering of the program transaction history. In fact, the TLP algorithm used at the SM determines this total ordering. Transaction ordering ensures that the total ordering of snapshots is consistent with the partial order over program transactions. The algorithm described takes as inputs this history and a vector representing the time of the steering transaction. If the steering transaction is consistent, the algorithm returns TRUE. If the steering transaction is inconsistent, the algorithm returns a vector representing the earliest consistent time after the given steering transaction at which a steering transaction could occur.

To accomplish consistency detection, the algorithm creates a consistency vector representing a *consistent cut* [13] at which a steering transaction could be applied. The algorithm works backward, generating vector times for consistent cuts based on comparisons between the program transaction being analyzed and the steering transaction. The algorithm completes once the present steering transaction is reached or an inconsistency has been detected.

To determine membership, the algorithms tag each message sent with the local snapshot id. Receiving processes keep track of this information by a vector time and associate it with the current transaction. At the end of transaction, this vector time information will be exchanged between IMs or sent to the SM for the transaction membership assembly and transaction ordering. Since the vector time information is needed only at the end of a transaction and assembly of the transaction information does not block continued execution of the application processes, this should result in relatively low perturbation [18].

6.2 Algorithm

This algorithm, seen in figure 2.6, requires six data structures and one Boolean variable. First, a *TLP (Transaction Labeling Protocol)* table is used to maintain the chronological history of program transactions. Next, there are four vector times, a Boolean vector, and a Boolean variable*: TV (Transaction Vector), SV (Steering Vector),* *CV (Consistency Vector)*, *CVTemp*, *Verified*, and *consistent*, respectively. Figure 2.7 shows an example of the initialized data structures. The *TV* vector holds the row of the *TLP* table currently being analyzed. The *SV* vector contains the timestamps representing the steering transaction. The *CV* vector represents the time of a consistent steering transaction. The *CVtemp* vector provides a temporary holder of possible new timestamps for the *CV* vector. The values of *CVtemp* should not be committed to the *CV* vector until all elements of the *SV* vector have been compared against corresponding elements in the *TV* vector. Both the *CV* vector and *CVtemp* vector are initially empty. The Boolean *Verified* vector contains flags signifying that the earliest timestamp for a steering event to occur at each respective process has been verified. If a TRUE flag is present, then no new timestamp for that process should be added to the *CV* vector. *Verified* is initialized with all elements set to FALSE. Finally, the Boolean variable *consistent* indicates whether the values stored in *CVtemp* should be committed to the *CV* vector.

At the beginning of each iteration of the WHILE loop beginning on line 16, *consistent* is set to TRUE. This WHILE loop is used to determine the stopping point for the algorithm. The algorithm terminates once all elements of the *Verified* vector that correspond to elements of the *SV* vector have been set to TRUE. The key point of analysis occurs in the FOR loop starting on line 20. Here, each non-empty element of the *TV* vector is compared with the corresponding non-empty element of the *SV* vector. If the element compared in the *TV* vector holds a timestamp equal to or later than that of the element in the *SV* vector, then that timestamp is entered into the corresponding element in *CVtemp*. If *consistent* remains TRUE through all iterations of the FOR loop, then the values of *CVtemp* are committed to the *CV* vector. However, if any element of the *TV* vector occurred earlier than the corresponding element in the *SV* vector, then *consistent* will be changed to FALSE and the loop will terminate, as seen in lines 33 and 34. All entries in *CVtemp* are then purged and all elements of the *Verified* vector corresponding to elements of *TV* are marked as TRUE. This later action indicates that the present *TV* vector was concurrent with the *SV* vector. As explained, any concurrency implies an inconsistent steering transaction.

One other condition will cause the FOR loop to terminate without completing all iterations. If any element of the *TV* vector corresponding to an element of the *Verified* vector has already been set to TRUE, then all elements of the *Verified* vector corresponding to elements of the *TV* vector will be set to TRUE, and the loop terminates. As above, if any element of the *TV* vector has already been verified, then the earliest time at which a steering event could have occurred for that process has happened. Therefore, no other process having direct or transitive communication with that process could consistently apply a steering action during the program transaction represented by that *TV* vector or any earlier *TV* vector.

Once the condition has been satisfied that all elements of the *Verified* vector corresponding to elements of the *SV* vector have been set to TRUE, the WHILE loop will terminate and a comparison between the *CV* vector and *SV* vector occurs. If the *CV* vector and *SV* vector are found to be identical, the algorithm returns TRUE. The IS system can then purge all checkpoints and stop any message logging. If the *CV* vector is not equal to the *SV* vector, then the algorithm returns the *CV* vector. The IS system can then issue a command for each process to rollback to the checkpoint at the time specified

```
Table TLP /*Table containing transaction history*/
Vector TV /*Vector holding information about present transaction in TLP*/
Vector SV /*Steering Vector containing list of processes involved in steering transaction*/
Vector CV /*Consistent Vector representing when the steering transaction should take place*/
Vector CVtemp /*Temporary vector to hold information until it is verified SV has not made TV
    inconsistent*/
Vector Verified /*A Vector of Boolean values set to true when a process listed in SV is at its
    earliest logical time to have invoked the steering command*/
Boolean consistent /*Boolean flag to indicate if SV has made TV inconsistent*/

BEGIN

set all elements of Verified to FALSE
set TV equal to last vector of TLP table

WHILE (all processes in SV have not been set to true in Verified)
        BEGIN
                consistent set to TRUE

                FOR (compare each corresponding, non-empty cell in TV and SV)
                        BEGIN
                                IF(any non-empty cell in TV corresponds to a cell marked TRUE
                                in Verified)
                                        THEN mark all cells in Verified corresponding to non-
                                         empty cells in TV TRUE
                                         set consistent to FALSE
                                         break

                                IF(TV is greater than or equal to SV)
                                        THEN set corresponding cell of CVtemp to TV
                                ELSE
                                        Mark Verified cells corresponding to non-empty TV cells to TRUE
                                        and mark consistent to FALSE
                                        break
                        END

                IF(consistent)
                        THEN
                                FOR(each non-empty element of CVtemp)
                                        set corresponding cells of CV to CVtemp

                IF(all cells of Verified corresponding to all cells SV are marked true)
                        break
                ELSE
                        set TV equal to previous vector in TLP
        END

IF(SV equals CV)
        Return Consistent
ELSE
        Return CV

END
```

Figure 2.6 – Consistency Verification Algorithm

in the *SV* vector, execute forward to the timestamps in the *CV*, then apply the steering command. Again, checkpoints can be purged and message logging may cease.

6.3 Examples of Algorithm in Use

6.3.1 Inconsistency Detection

	P1	P2	P3	P4
T1	0	0	0	0
T2	1	1	1	
T3			2	1
T4			3	2
T5	2	2		

TLP TABLE

P1	P2	P3	P4
1	1	2	

SV VECTOR

P1	P2	P3	P4
2	2		

TV VECTOR

P1	P2	P3	P4
-	-	-	-

CV VECTOR

P1	P2	P3	P4
F	F	F	F

VERIFIED VECTOR

Figure 2.7 – Represents the initialization of the primary data structures at the beginning of the algorithm for a hypothetical execution of a distributed computation

Given the initial input above, we trace the updates to the *CV* and *Verified* vector as the algorithm proceeds. The following figures show the updates to the vectors *CV* and *Verified* after each iteration of the while loop.

Iteration 1

All non-empty cells of T5 are greater than the corresponding cells of the *SV*. The *CV* is updated to contain these values. The *Verified* vector is unchanged.

CV Vector

P1	P2	P3	P4
2	2		

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 2.8 – Represents the updated states of the CV vector and Verified vector

Iteration 2

Again, the non-empty cells of T4 are greater than or equal to the corresponding cells of the *SV*. *CV* is updated accordingly.

CV Vector

P1	P2	P3	P4
2	2	3	

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 2.9 – Represents the updated states of the CV and Verified Vector

Iteration 3

All cells of T3 are greater than or equal to their corresponding cells of the *SV*. Again, the *CV* is updated.

CV Vector

P1	P2	P3	P4
2	2	2	

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 2.10 – Represents the updated states of the CV and Verified Vector

Iteration 4

Unlike the previous iterations, all the cells of T2 are not greater than or equal to the corresponding cells in the *SV*. In fact, the value in the cell for P3 is less than that in the *SV*. Therefore, all cells of processes in the *Verified* vector corresponding to cells of processes in T2 are marked TRUE. The *CV* vector remains unchanged.

CV Vector

P1	P2	P3	P4
2	2	2	

Verified Vector

P1	P2	P3	P4
T	T	T	F

Figure 2.11 – Represents the updated states of the CV and Verified Vector

Iteration 5

Finally, we make a comparison between T1 and *SV* and discover that the cells of T1 contain lesser values than the corresponding cells of the *SV*. We mark the corresponding cells of the *Verified* vector TRUE. Each cell of the *Verified* vector is now marked TRUE.

CV Vector

P1	P2	P3	P4
2	2	2	

Verified Vector

P1	P2	P3	P4
T	T	T	T

Figure 2.12 – Represents the updated states of the CV and Verified Vector

The *SV* is not equal to the *CV*. Thus, the algorithm returns the *CV*. A rollback to the *SV* is issued and the steering action will be applied at the *CV*, both seen in figure 2.13.

SV Vector

P1	P2	P3	P4
1	1	2	

CV Vector

P1	P2	P3	P4
2	2	2	

Figure 2.13 – Represents the original SV vector and the new consistent steering vector time CV

6.3.2 Consistency Verification

Given the initial input belowe, we trace the updates to the *CV* and *Verified* vector as the algorithm proceeds.

	P1	P2	P3	P4
T1	0	0		0
T2	1		0	1
T3			1	2
T4		1	2	
T5	2	2		3

TLP Table

P1	P2	P3	P4
	1	1	2

SV Vector

P1	P2	P3	P4
2	2		3

TV Vector

P1	P2	P3	P4
-	-	-	-

CV Vector

P1	P2	P3	P4
F	F	F	F

Verified Vector

Figure 2.14 – Represents the initialization of the primary data structures at the beginning of the algorithm for a hypothetical execution of a distributed computation.

Iteration 1

Since all the non-empty cells of T5 are greater than corresponding cells of the *SV*, we update the cells of the *CV* vector. The *Verified* vector remains unchanged.

CV Vector

P1	P2	P3	P4
	2		3

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 2.15 – Represents the updated states of the CV and Verified Vector

Iteration 2

Again, the non-empty cells of T4 are greater than or equal to the corresponding cells of the *SV*. The *CV* vector is updated. *Verified* is unchanged.

CV Vector

P1	P2	P3	P4
	1	2	3

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 2.16 – Represents the updated states of the CV and Verified Vector

Iteration 3

All cells of T3 are greater than or equal to the corresponding cells of the *SV*. The *CV* is updated. *Verified* remains unchanged.

CV Vector

P1	P2	P3	P4
	1	1	2

Verified Vector

P1	P2	P3	P4
F	F	F	F

Figure 2.17 – Represents the updated states of the CV and Verified Vector

Iteration 4

Unlike the previous iterations, all the cells of T2 are not greater than or equal to the corresponding cells in the *SV*. In fact, the values in the cell for P3 and P4 are less than the corresponding values in the *SV*. Therefore, all cells of processes in the *Verified* vector corresponding to cells of processes in T2 are marked TRUE. The *CV* vector remains unchanged.

CV Vector

P1	P2	P3	P4
	1	1	2

Verified Vector

P1	P2	P3	P4
T	F	T	T

Figure 2.18 – Represents the updated states of the CV and Verified Vector

Iteration 5

Finally, we make a comparison between T1 and *SV*; the cells of T1 contain lesser values than the corresponding cells of the *SV*. We mark the corresponding cells of the *Verified* vector TRUE. Each cell of the *Verified* vector is now marked TRUE. Since *SV* is equal to *CV*, the algorithm returns TRUE. The IS system is now able to purge all checkpoints and cease all logging.

CV Vector

P1	P2	P3	P4
	1	1	2

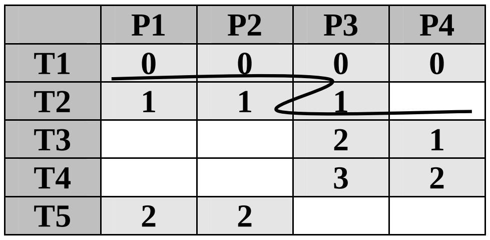
Verified Vector

P1	P2	P3	P4
T	T	T	T

Figure 2.19 – Represents the updated states of the CV and Verified Vector

6.4 Explanation of Algorithm Correctness

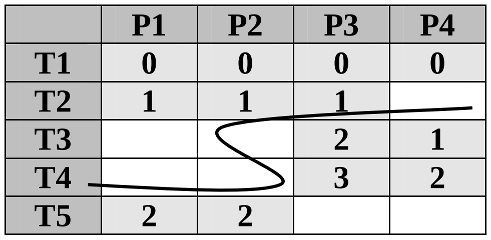
To clarify the point that the original *SV* in section 6.3.1 was not consistent, consider figure 2.20 containing the original *TLP* table. A bold line represents the points at which the steering transaction was applied. Note that program transaction T2 is cut by the original steering transaction; the steering transaction did not happen fully before or after program transaction T2, but rather was *concurrent* with T2.



	P1	P2	P3	P4
T1	0	0	0	0
T2	1	1	1	
T3			2	1
T4			3	2
T5	2	2		

Figure 2.20 – Original SV cutting Program Transaction T2

Figure 2.21 contains the original *TLP* table from section 6.3.1 but illustrates the new steering transaction generated by the *CV* vector. In this figure, it can be seen that no program transaction is cut by the steering transaction, and thus the steering transaction either happened before or after every program transaction but is not concurrent with any of them. Therefore, this steering transaction is consistent. This same reasoning can be applied to the steering transaction in section 6.3.2 to verify that it too is consistent.



	P1	P2	P3	P4
T1	0	0	0	0
T2	1	1	1	
T3			2	1
T4			3	2
T5	2	2		

Figure 2.21 – Consistent SV not cutting any Program Transactions

7. Related Work

Causality is fundamental to many problems in distributed computing. For example, determining a consistent global snapshot of a distributed computation [1, 3] requires finding a set of local snapshots such that the causal relation between all events included in the snapshots is respected in the following sense: if e' is contained in the global snapshot formed by the union of local snapshots, and e→ e' holds, then e must also be included in the global snapshot. Thus, the notion of consistency in distributed systems is basically an issue of correctly reflecting causality. Many important applications of causal consistency are summarized by Schwarz and Mattern in [15].

An important characteristic of distributed systems is that there is no global clock. Consequently, ordering the events in a distributed system can be challenging. Lamport [9] introduced an efficient mechanism called logical clocks for totally ordering the events in a distributed system, but the mechanism is not sufficiently powerful to allow concurrent events to be identified. Mattern [11] and Fidge [4] independently developed vector clocks, which precisely capture the causal ordering between distributed events. The main difference between Mattern and Fidge vector time schemes and ours is the way in which the logical time of a process is measured. Under the Mattern and Fidge schemes, the logical time of a process is measured in “number of past events” at that process. However, under our scheme, the logical time of a process is measured in “number of past local transactions” at that process.

The Z-path and Z-cycle notion, introduced by Netzer and Xu [13], generalizes the notion of a causal path of messages defined by Lamport’s happened-before relation [9]. A local checkpoint is useless iff it is involved in a Z-cycle. In the transaction-based computational model, if a checkpoint is taken, all participant processes in a global transaction take the checkpoint at the end of transaction. In this way, no local checkpoint will be involved in a Z-cycle because there is no message in transit at the end of transaction.

8. Conclusion

The EV and IS systems offers a real-time environment through which users may gain understanding and perform on-the-fly program manipulation. However, with the ability to adjust application parameters on-the-fly, comes the necessity to verify that changes are made in a logically consistent manner, at *consistent cuts* [13]. Through the notions of steering transactions, program transactions, and consistent steering, the algorithm presented here provides a means to verify consistency, detect inconsistency, and calculate the earliest consistent cuts. Consistency guarantees that visualizations presented to the users represent a valid state of the computation and that the steering operations are applied in a way that maintains the correctness of the computation.

The currently developed EV environment and the prototyped IS environment permit users to configure the environments to provide the desired balance among consistency, lag, and perturbation of the underlying program execution. Included in this research is the development of a variety of modular algorithms for the collection of snapshots with varying degrees of consistency, for either selective or comprehensive monitoring, as well as the development of algorithms that permit the consistent application of changes to the program in execution while minimizing the lag and perturbation that result. Also under study are techniques for improving the scalability of these algorithms. As the number of processes participating in steering transactions and the number of steering transactions increases, the message and processing load at the SM will increase significantly and the the SM will become a bottleneck. To address this problem, a hierarchical combination of SMs has been designed, but not yet implemented. Further, as the number of processes participating in steering transactions increases, the probability that a steering transaction will be consistent will likely decrease. Techniques to dynamically cluster related (steered together) processes under the same SM in the SM hierarchy are being considered to address this problem.

References

[1] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems,* February 1985, 3(1):63-75.

[2] Dijkstra and B.P. Scholten, "Termination Detections for Diffusing Computations," *Information Processing Letters*, 1980, 11(1): 1-4.

[3] J. Fowler and W. Zwaenepoel, "Casual Distributed Breakpoints," in Proceedings, *10th International Conference on Distributed Computing Systems*, Paris, France, May 1990, pp. 134-141.

[4] C.J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering", *Australian Computer Science Communications*, February 1988, pp. 56-66.

[5] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko and J. Vetter, "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 Feb 1995, pp. 422-429.

[6] D. Hart, E. Kraemer, and G.C. Roman, "Interactive Visual Exploration of Distributed Computations," in Proceedings, *11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997, pp. 121-127.

[7] D. Hart, E. Kraemer, "An Agent-Based Perspective of Distributed Monitoring and Steering," in Proceedings, *2nd Sigmetrics Symposium on Parallel and Distributed Tools*, Welches, Oregon, August 1998, pp. 151.

[8] E. Kraemer, D. Hart, and G.C. Roman, "Balancing Consistency and Lag in Transaction-Based Computational Steering," in Proceedings, *35th Annual Hawaii International Conference on Software Specification and Design*, January 1998, pp. 137-147.

[9] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM 21*, 1978, 7(558-565).

[10] Lynch, N., *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[11] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, North-Holland, 1989, pp. 215-226.

[12] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." *Journal of Parallel and Distributed Computing*, 18:423-424, 1993.

[13] Robert H. B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Transactions on Parallel and Distributed Systems*, February 1995, 6(2):165-169.

[14] P. M. Papadopoulos, J. A. Kohl, B. D. Semeraro, "CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance," *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*, Kona, Hawaii, January 1998

[15] Reinhard Schwarz and Friedemann Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, 1994, 7(3): 149-174.

[16] J. Vetter and K. Schwan, “High Performance Computational Steering of Physical Simulations,” in *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997, pp127-132.

[17] J. Vetter and K. Schwan. “Progress: A Toolkit for Interactive Program Steering,” *Proceedings of the 1995 International Conference on Parallel Processing*, Urbana, IL, Aug 1995, pp. 139-142.

[18] H. Vuppula, E. Kraemer, and D. Hart, “Algorithms for Collection of Global Snapshots: An Empirical Evaluation,” Proceedings in, *ICSA Conference on Parallel and Distributed Computing Systems*, August 2000, pp. 197-204.

CHAPTER 3

OPTIMISTIC COMPUTATIONAL STEERING[♦]

[♦] Miller, David W., Eileen Kraemer, and Jinhua Guo. Extended Abstract Submitted to *SuperComputing 2002, 4/26/02.*

Optimistic Computational Steering[*]

David W. Miller, Eileen Kraemer[†], Jinhua Guo
{miller, eileen, jinhua} @cs.uga.edu
Department of Computer Science
The University of Georgia

Abstract

Computational steering is the online, interactive allocation of resources and adjustment of application parameters. Although several systems for steering distributed computations have been developed, few support the coordinated steering of multiple processes. Of those that do provide such support, most take a conservative approach. Our system is unique in its optimistic approach. Coordinated steering of either type has the potential to introduce significant perturbation. We focus on optimistic steering, which does not require global synchronization before a steering event may take place. Such an approach requires both the ability to determine the consistency of steering transactions and the ability to correct any inconsistencies that may occur. To address these issues, we have developed algorithms for consistency detection and a steering system that has the ability to correct inconsistencies through computational rollback and re-execution. Presented in this paper are the details of our steering system and a performance analysis of that system.

Keywords: *Checkpoint, Conservative Steering, Message Logging, Optimistic Steering, Program Transaction, Re-execution, Rollback, Steering Transaction*

[*] This work is supported by the National Science Foundation under Grant No. 9996082.
[†] Primary contact author.

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 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 cause-and-effect relationships, 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, a user interface with visualizations capabilities, and a mechanism for propagating steering actions to the executing programs. To this end, we have developed an Exploratory Visualization (EV) and Interactive Steering (IS) system. Through this system, the user may monitor attributes and variables of distributed computations via queries and visualizations and may dynamically manipulate program variables or adjust resource allocation. Such a system must address issues including consistency, latency, and overall computational perturbation. This paper will focus on the computational perturbation as it relates to the overhead introduced by the Interactive Steering (IS) portion of the system.

Within the IS system, a global state change resulting from each user's steering request is a called a steering transaction. A steering transaction may be decomposed into steering events, modifications of the local state of one process. Criteria for the correctness of steering changes to the executing system can vary considerably depending on the type of change to be made. Some steering changes may be applied at any of the participating processes at any point in the computation. However, if a steering transaction is intended to update critical control parameters or change the global configuration, causal consistency is often required to maintain the correctness of the computation. That is, all local steering changes must be applied concurrently across all participating processes.

Towards the goal of computational steering, we introduce an approach called *optimistic steering*. In optimistic steering, the system invokes steering events at the processes, without stopping the computation or checking for initial consistency. Therefore, the consistency of the global state of the computation, and specifically, the consistency of the steering transaction, must be checked. If the steering transaction is consistent, the computation continues under normal execution. However, if the steering transaction is found to be inconsistent, then the earliest time at which the steering event could be consistently applied at each process is calculated. The system must then enter a recovery state, rollback to the state before the original steering change(s) occurred, re-execute forward to the consistent time while replaying any messages that may have been logged, apply the steering changes, and then continue re-executing in recovery until all message logs have been emptied, after which normal execution may resume.

In this paper, we describe our implementation of this approach and an evaluation of its performance. In section 2, we provide our rationale for optimistic steering. Section 3 defines the computational model supported in the EV/IS system and associated terminology. Following, Section 4 provides an overview of our system architecture. In section 5, we discuss the components of the IS system, including consistency detection, checkpointing, message logging, rollback, re-execution, and message replay. Section 6 describes our experiments to evaluate system performance and discusses the results of those experiments. Section 7 provides an overview of related research projects in the area of computational steering. Finally, section 8 provides conclusions and future work.

2. Why Optimistic Steering

Globally consistent steering updates typically require that a computation reach quiescence [1, 9] before a steering change is applied; the computation blocks before a consensus decision is made and steering changes are applied. This is a nontrivial process in a distributed, asynchronous environment in which centralized control of the computation is required, and can result in considerable perturbation of the computation and significant steering lag [6]. In contrast to this conservative or "pessimistic" approach, we present another approach called *optimistic steering*. In optimistic steering, the system invokes each steering event in the steering transaction at the respective process without concern for or knowledge of the state of any other process. Therefore, the consistency of the global state of the computation, and specifically, the consistency of the steering transaction, must be checked. If consistency is verified, the computation continues. If the steering transaction is found to be inconsistent, then the earliest time at which the steering event could be consistently applied at each process is calculated. The computation must then rollback to its state prior to the application of the steering change, execute forward, and then apply the steering change at the consistent time.

The optimistic approach is expected to perform well in the case that steering is relatively rare and that the processes of the computation tend to remain roughly synchronized because of coordination at the application level. This is often the case, as processes typically wait for messages from one another or synchronize at barriers. In this case, the next steerable points will most likely be consistent since processes will receive their steering request in the same global transaction.

Accordingly, while the overhead of state saving and some logging will be incurred for each steering transaction, rollback and re-execution will be incurred only in the case of inconsistent steering, and both logging and re-execution will be of limited scope and duration.

3. Models and Definitions

A *distributed computation* consists of a set of dynamic processes that work together to achieve a common goal. In our system, each process exports a set of attributes that are available for monitoring and steering. The state of the process changes when a program event occurs or when a user-specified steering event occurs. *Program events* of interest are *sends*, *receives*, and events that mark the end of a process's participation in a *transaction*, known as *end-of-transaction* events. A *Steering event* is the result of steering request, which makes changes to the state of a process thus affecting the overall computation.

Any distributed computation can be decomposed into sets of program events. A *program transaction* is a set of program events collected at one or more processes and representing the same logical point in the program's execution. The events in the set are related through a direct or transitive communication pattern. The time of a program transaction is represented as a vector in which each process involved in a transaction has a timestamp representing its local time when it participated in the transaction.

Finally, a group of related steering events can be organized into a set, a *steering transaction that* represents the set of steering events resulting from a single request. These *transactions* are represented by vectors containing the local timestamps of the processes when they participated in the steering actions.

4. EV/IS Architecture

The EV/IS environment is viewed as a three-part architecture, known as the Pathfinder Architecture [3], composed of Interaction Managers (IM), a Snapshot/Steering Manager (SM), and a User Interface (UI), seen in figure 3.1.

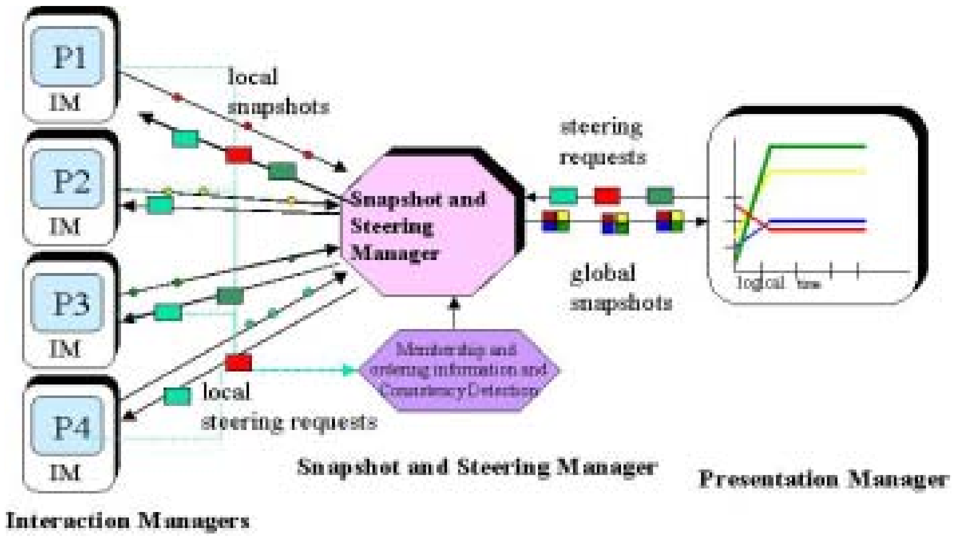


Figure 3.1 – Pathfinder Architecture

The IM is composed of communication layers between the process and its communication environment. The IM implements a transaction labeling protocol. The IM collects information (local snapshots) from the processes that participate in each transaction, as well as the communication relationships between processes (membership), and forwards that information to the SM, which then calculates the dependence relationships between transactions (ordering). Additionally, the IM is responsible for invoking steering requests, reporting process steering event times to the SM, checkpointing, message logging, and process recovery. Processes that communicate with one another during the execution of a logical block of code that forms a transaction are considered members of the same transaction. However, each process typically knows only of its neighboring processes, those with which it directly communicated. Based on the transitive communication patterns, the complete membership within a transaction can be determined [17].

At the SM, globally consistent snapshots are generated based on the local snapshots from the IMs and the transaction labeling information. This labeling information may consist of neighbor lists, message counts, vector clocks, or other means—herein, we focus on vector clocks [17]. The transaction labeling information relies on a set of vector clocks that encode the inter-process communication that occurred during the represented transactions. This information can be used to group local snapshots into global snapshots that represent the global state of the computation at one instant in time [17]. Note that the existence of these vector clocks for purposes of visualization provides the foundation for the relatively low-cost addition of consistent steering.

Finally, the global snapshots are sent to the UI to be visualized. Through the UI, the user may pose queries to the system in order to gather application-specific and system-specific values to aid in overall understanding. The UI also provides the interface through which users may steer the distributed computation in real-time. The steering request will be issued to the SM for processing. Once the SM receives a steering request, it will issue a steering command to the IM at each process involved in the steering action. Each participating IM will report back to the SM the local process time at which the steering event occurred. The SM will use the event times to form a vector timestamp for the steering transaction. Based on the TLP information already present at the SM, and the steering transaction's vector time, the SM can determine if a steering command was applied consistently [10][4]. If not, the SM will issue a rollback and a consistent steering time to each process.

5. Interactive Steering Implementation Aspects

For a system to have the capability to apply steering changes in an optimistic fashion so that each process immediately applies a steering request without concern for or knowledge of any other process, requires not only the ability to confirm consistency and detect inconsistency but also the mechanism to correct inconsistencies. Consistency confirmation requires the capability for the system to determine concurrency relations between steering events. Section 5.1 will briefly discuss two algorithms for consistency confirmation. Correcting inconsistencies is more complicated.

Inconsistency correction requires not only the ability to checkpoint a computation's state before applying a steering change, but also the ability to log messages while steering, recover from checkpoints, re-execute to a consistent point, reapply steering changes, and replay messages that will not be resent, all while maintaining transparency to both the user and the running computation. Each of these aspects will be discussed in the remaining subsections.

5.1 Consistency Detection

A straightforward approach to confirming consistency is to identify the concurrency relation between all steering events. If all steering events are concurrent, then the steering transaction is consistent; otherwise, it is inconsistent. To this end, we have derived two algorithms, history-based and vector time based, to determine the concurrency of steering events and thus the consistency of a steering transaction. Section 5.1.1 discusses the history-based algorithm [10] and section 5.1.2 covers the vector time based algorithm [4]. One key feature of each of these algorithms is their ability to identify all consistent cuts in a program's execution, or subset thereof, for which a steering transaction could take place. If the original steering transaction occurred at one of those cuts, consistency is verified. Otherwise, the system will determine the earliest consistent cut for which a steering transaction can be reapplied, thus alleviating any possibility for cascading rollbacks and numerous consistency re-verifications.

5.1.1 History-Based Algorithm

The history-based algorithm [10] depends on a specific subset of the computation history that contains all program transactions that occurred immediately after the earliest steering event and before the latest steering event. If a steering transaction can be clearly inserted between any two of these program transactions, it is consistent.

Specifically, in order to determine the consistency of a steering transaction, the system maintains a list of vectors representing a partial ordering of the program transaction history. In fact, the algorithm for transaction ordering installed at the SM determines a valid total ordering of these transactions. To accomplish consistency detection, the algorithm creates a consistency vector representing a consistent cut at which a steering transaction could be applied. The algorithm works backwards, generating vector times for consistent cuts based on comparisons between the program transactions being analyzed and the steering transaction. The algorithm terminates once the present steering transaction is reached or an inconsistency has been detected. If an inconsistency is detected, the last consistent cut stored in the consistency vector contains the earliest consistent times for the steering transaction and represents the point at which the steering changes will be reapplied.

5.1.2 Vector Time Based Algorithm

Unlike the history-based algorithm that takes a steering transaction as a whole and relies on the history of the inter-transactional relations between a steering transaction and the program transactions, the vector time based algorithm [4] focuses on the relations among the steering events in a steering transaction. In the vector time based algorithm, the consistency of a steering transaction is detected by directly comparing the vector timestamps of steering events. Further, the earliest consistent transaction can be obtained by checking the vector time of each steering event in a steering transaction.

In the vector time based algorithm, each process is associated with a vector clock V of size N, where N is the number of processes in the system. Each element in the vector corresponds to a process in the system. The value of V[i] denotes the number of past local transactions at that process, as known by this process. The vector time of a steering event in process P_i is the vector time that results from the occurrence of a steering event in process P_i.

In the vector time based algorithm, we decide whether or not two steering events are concurrent by directly comparing the vector timestamps of steering events [4]. If the steering events in a steering transaction are not concurrent, we know that there exists some causal relation between at least two of the steering events. To avoid violating such causal relations, the simplest method is to apply the steering action at the later time of the two. If we can find the latest time for each process involved in the steering transaction and apply the steering at that time, the new steering transaction will be consistent and it will be the earliest consistent transaction, as any steering transaction applied before it is inconsistent.

5.2 Process State

Within the IS system, each process of the distributed computation may be executing in one of three computational states, as seen in the figure 3.2. A *normal* computational state implies that neither the consistency of a steering change is in question nor is the correction of an inconsistency underway. Note that both directly affected and indirectly affected processes can lead to an overall inconsistency in a steering transaction. Thus, a process enters a *steering* state when either a steering request is issued to that process[1] or a message is received from a process already in a steering state; such a message is known as a *tainted* message. [10],[4] provides further insight into the various aspects of steering consistency detection. Finally, a process may only enter a recovering state if it was in a steering state relative to a steering transaction that was determined to be inconsistent.

[1] Presently, a process may only receive a steering request if it is in a normal state.

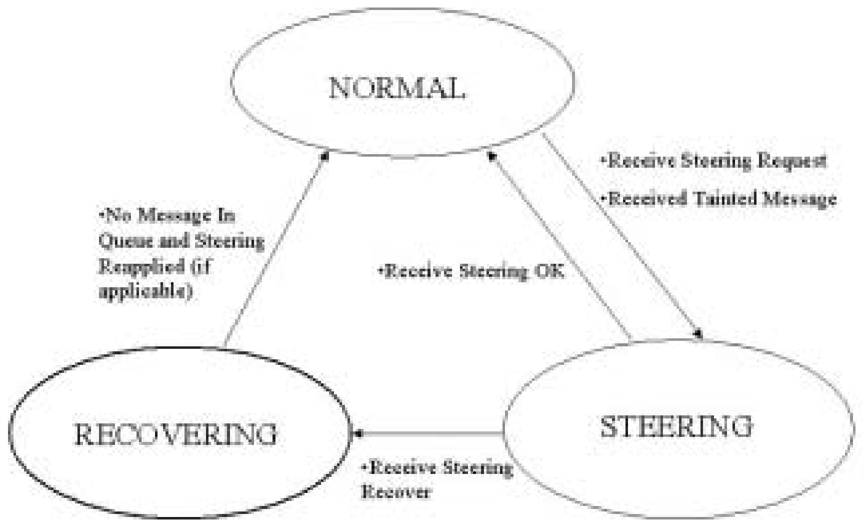

NORMAL
•Receive Steering Request
•Received Tainted Message
•No Message In Queue and Steering Reapplied (if applicable)
•Receive Steering OK
RECOVERING
STEERING
•Receive Steering Recover

Figure 3.2 – Computation State Model

5.3 Checkpointing

In order to allow complete transparency to an application during recovery, the entire computational state of a process must be recorded in the form of a checkpoint before a steering change can be invoked or before a tainted message can be processed. Thus, before a process may enter a steering state, a checkpoint must be taken. Doing so will later allow a process, if necessary, to rollback to the identical state before a steering change affected the process state, in order to correct an inconsistent steering transaction. To accomplish this, the IS system maintains a checkpoint that includes the state of all static variables, all dynamic variables explicitly listed for protection by the programmer, the execution stack, and key registry values, including but not limited to the program counter (PC) and stack pointer (SP). Maintaining the execution stack and PC allows the IS system to seamlessly restart a process's execution at the exact point at which the checkpoint was taken.

As described in Section 4, in the EV/IS system each process in a distributed computation is “wrapped” by an IM, communication layers between the process and its communication environment. Included in these layers is the optimistic steering module, which maintains process state, message logs, and checkpoint file handles. Each layer of the IM is loaded dynamically at runtime and is thus not part of the static set of variables. Rather, each layer resides in the program heap at each process. During a checkpoint, it is neither desirable nor necessary to record the state of the IM, as this would introduce both wasted storage for the checkpoint and excessive checkpoint restoration time during a recovery. Therefore, to allow dynamic memory protection, the IS system provides the programmer with an explicit protocol through which dynamic variables created during a process’s execution can be specified for inclusion in checkpoints. Figure 3.3 provides a code sample illustrating this.

```
QBV *qbv = new QBV(…); //Allocate first IM layer
char *mem = new char[100]; //Dynamically allocate an array of chars
qbv->protectMem(mem, sizeof(char) * 100);//Memory to include in checkpoints
```

Figure 3.3 – Memory Protection Example

At present, all checkpoint data is written to two binary files per process. The static variables and execution stack are written into one file while all protected dynamic memory is written to a second file. Since the static set of variables and execution stack both lie contiguously in the program’s stack, a direct memory dump can be made from the program stack to file during a checkpoint. However, since all data in the program’s heap is not maintained in a checkpoint, a slightly more elaborate scenario exists for dynamic memory checkpointing.

As mentioned, through an IS protocol, a programmer can explicitly protect any dynamic memory that needs to be checkpointed. Through this protocol, the programmer simply specifies the base memory address for a variable and the number of contiguously associated bytes, as seen in figure 3.3. This information is then stored in a data structure. During a checkpoint, the data structure of each process is traversed and all specified memory is contiguously dumped to the second binary file.

Finally, to provide a robust checkpoint, the state of the stack context/environment must be stored out so that on a rollback the process state is just as it was when the checkpoint was taken. Fortunately, the *setjmp.h* library provides an API for both storing out and recovering such environmental states. As such, the IS system makes full use of this standard Unix/Linux library during checkpointing.

5.4 Message Logging

Within the EV/IS system, communication traffic exists between the executing processes and between the various control modules that form the EV/IS system. We refer to these messages, respectively, as application messages and control messages. Every application message piggybacks the sending process's current state, as modeled in figure 3.2, which will include the present steering transaction number that is underway if that process is in a steering state. For the purposes of message logging, the application's state can be grouped into one of two sets, either steering or normal/recovering.

If a process is in a steering state, then it must begin actively logging any message that will not automatically be resent during recovery–each receiving process maintains a separate message queue for each sending process. The system determines which messages to log based on the state information that has been incorporated by the sending process. Any message sent by a process whose state is indicated to be normal/recovering should be logged. The reason for such action is that if a process indicates that it is in a normal/recovering state then it does not have the potential to rollback; thus, any such message will not be resent if the receiving process is forced to rollback. On the other hand, if a process that is presently steering receives a tainted message from another process that is also steering, it can be assumed that on a rollback both the receiving and sending processes will have to recover and thus any such message will be resent and should not be logged. Message replay will be covered in the next section.

5.5 Recovery – Rollback and Re-execution

Because the IS system employs an optimistic steering approach to computational steering, the system must have the ability to not only restore the state to what it was before the inconsistency occurred but to also re-execute a portion of the computation to a consistent state and reapply the steering changes. In order to accomplish this, as previously mentioned, the SM is responsible for verifying the consistency of each steering transaction. If the steering transaction is determined to be consistent, the SM will simply issue a *SteeringOK* message to all processes at which point all checkpoints may be discarded, message queues cleared, and a process will transition back into a normal state. On the other hand, if the steering transaction is determined to be inconsistent, the SM must issue a *SteeringRecovery* message to all processes, at which time any process presently in a steering state will begin the recovery process and transition into a recovery state[2].

Because the SM has no knowledge of processes that have been indirectly affected by a steering change through the receipt of a tainted message, it must issue a SteeringRecovery message to *all processes* within the system. Processes have the ability to receive SteeringRecovery messages at either a transaction boundary or during a *Receive*. Because the scenario exists that a process *A* may receive a SteeringRecovery message, and thus rollback, before another process *B* that is dependent on a message from *A*, the system must never allow a blocking *Receive* to occur—this fact is transparent to the user. Rather, the system first attempts a non-blocking *Receive* of an application message. If that is successful, the system returns control to the application. If there is no message available, the system then checks for a SteeringRecovery control message. If a SteeringRecovery message is available, process recovery will begin; otherwise, the system then performs another non-blocking *Receive* for the application message. This mechanism prevents system deadlock caused by unsynchronized rollbacks.

Once a process that is in a steering state receives a SteeringRecovery message, it then transitions into a recovery state and restores the previously taken checkpoint. At this point, the system will resume execution at the exact point the checkpoint was initially taken, which could either have been at a transaction boundary or during the receipt of a message. Once a process begins re-executing, on each *Receive* it first attempts to replay a logged message from the queue associated with the sender. If no logged message exists, the system then resorts to performing a normal *Receive*.

[2] No process has the ability to transition into a recovery state unless it is presently in a steering state.

An interesting scenario exists in which a process may send a tainted message, which the receiving process does not attempt to process until after the SM has issued either a SteeringOK or SteeringRecovery message. Therefore, each process maintains a mapping indicating which steering transactions it knows of, and whether they have been determined to be consistent or inconsistent. If a process receives a tainted message associated with a steering transaction that was already deemed consistent, it processes the message as normal and will not take a checkpoint. On the other hand, if a process receives a tainted message associated with a steering transaction determined to be inconsistent, it knows that message has or will be resent and thus throws away the present message and performs another *Receive* in order to receive the resent message.

Each recovering process will re-execute forward to the transaction time that was determined to be a consistent cut by the SM. At that time, it will reapply the steering changes that were originally requested by the user. Each process may transition back into a normal state of execution once the steering changes have been reapplied and all of its message queues have been emptied.

6. Performance

To measure the performance of the EV/IS system and particularly the overhead associated with both monitoring and steering, we ran a set of tests that encompass four variables that affect our system overhead: communication pattern, transaction size, number of processes, and monitoring/steering status. Three basic types of distributed computation communication patterns were tested: one in which all processes communicate during each transaction and synchronize at the end of each transaction (Global); one in which processes communicate pair wise during each transaction and every ten transactions all processes communicate and synchronize (Mixed); finally, one in which processes perform only pair wise communication and no global communication or synchronization takes place (Pairwise). For each communication type, we considered short transactions in which each process had roughly a .25 second computation and large transactions in which each process had roughly a 2.5 second computation. Each test was run on 2, 4, and 8 processes, taking the average execution time of 5 runs of 150 transactions for an unmonitored computation, a monitored computation, and a monitored and steered computation. For each of the runs that included steering, 5 steering transactions took place. Except for the tests containing only 2 processes, during each run 2 steering transactions contained all the processes, 2 steering transactions contained 75% of the processes and one steering transaction contained 50% of the processes. Note that processes may be indirectly affected by a steering transaction. For the 2 processes runs, all processes were contained in each steering transaction. This is because steering only 1 process would guarantee consistency, and thus would introduce very little overhead and provide less meaningful performance results. Within the EV/IS, there exist two extra control processes during each run, one for the SM and one for a daemon process responsible for establishing all UI and SM communication channels.

Each test was run on a cluster of 4 Pentium II 450 Mhz workstations with 128 Mg of RAM and running RedHat Linux 7.2. Only the application and control processes were executed on the workstations during the tests, and the GUI was run on a separate server, to avoid further workstation load imbalance.

In figure 3.4, the average execution times for unmonitored, monitored, and steered test sets with small transactions are presented. Also presented in figure 3.4 is the percentage of execution overhead. In the figure, each bar corresponds to the average execution time in seconds of the test set indicated by the legend. Above each bar corresponding to a monitored test set, a percentage is given representing the execution overhead experienced between the monitored and corresponding unmonitored average execution times. Similarly, above each bar corresponding to a steered test set, a percentage is given representing the execution overhead between the monitored and corresponding steered average execution times.

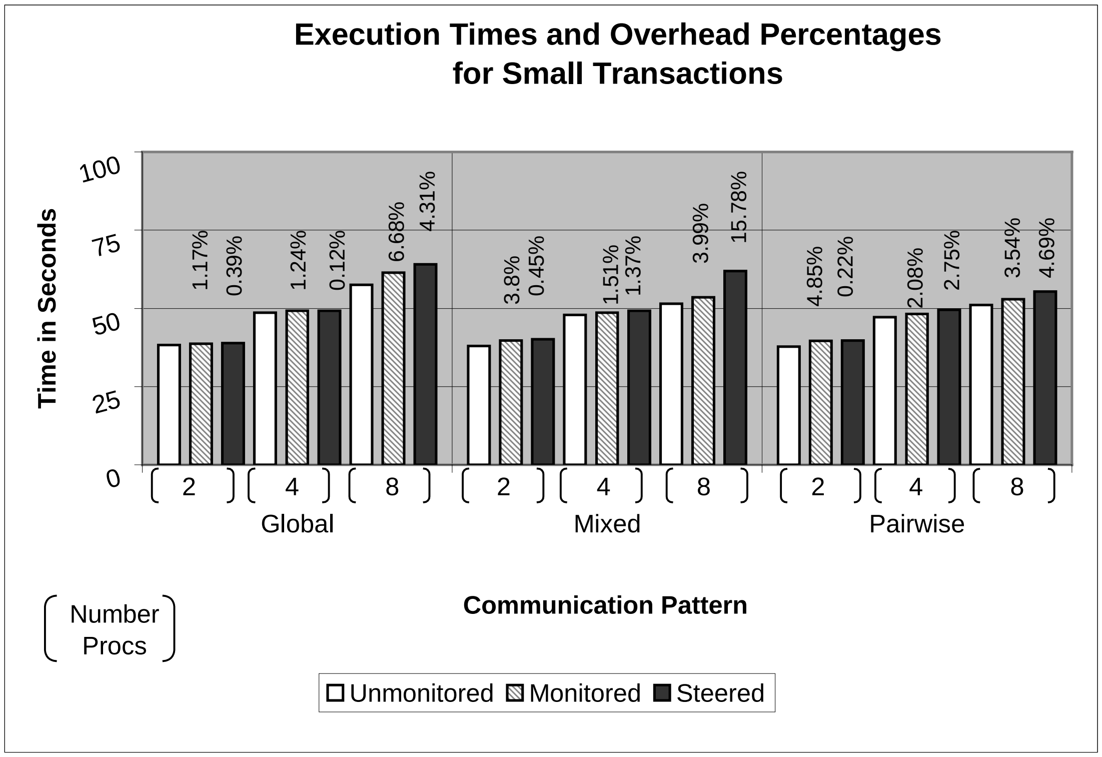


Figure 3.4 – Execution Times and Overhead for Small Transactions

Figure 3.5 presents execution times for unmonitored, monitored, and steered test sets with large transactions. As in figure 3.4, both average executions times and percentage of execution overhead are presented. Finally, figure 3.6 presents the average percentage of steering transactions applied consistently on the first attempt based on the number of application processes, size of transaction, and communication pattern.

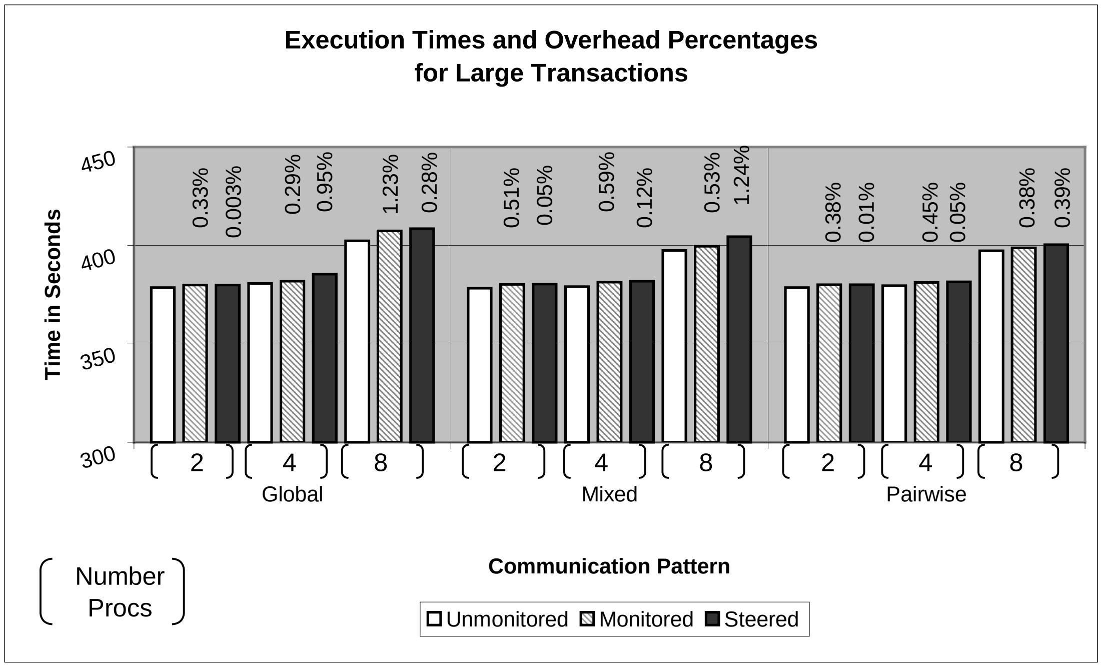


Figure 3.5 – Execution Times and Overhead for Large Transactions

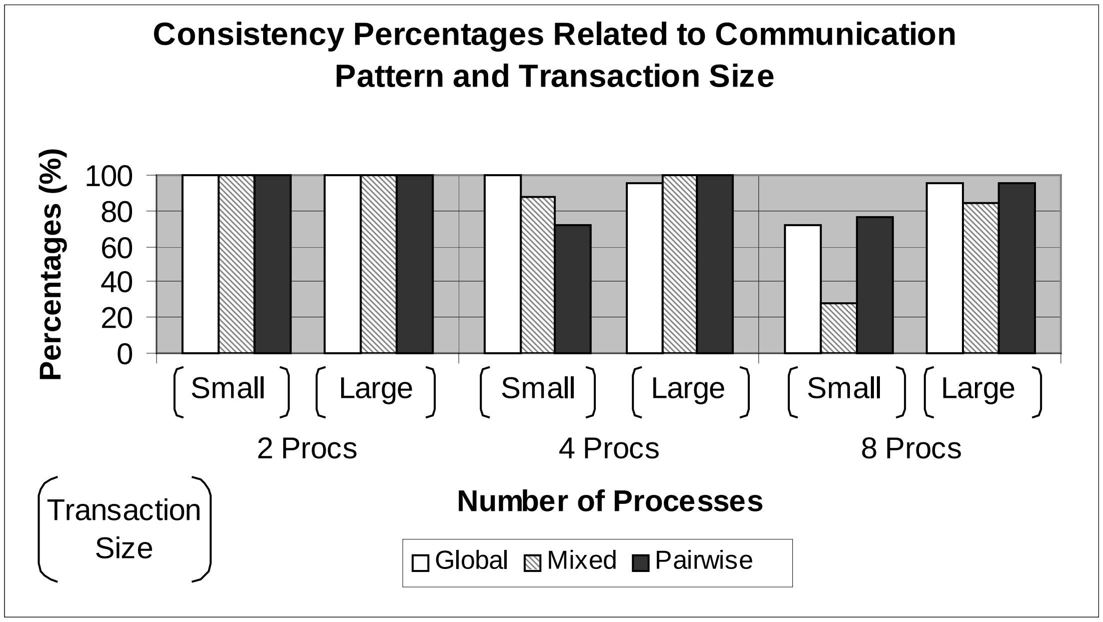


Figure 3.6 – Consistency Percentages

As can be seen from figure 3.6, steering consistency is affected by the average length of time a transaction spends performing computation, the number of possible processes that can be steered, and the communication pattern between the processes. As we expected, the tests with only two processes were highly consistent. But, as the number of steerable processes increases so does the likelihood of inconsistency. Similarly, mixed communication patterns also increase the likelihood of steering inconsistencies, with an average consistency of slightly above 80% while the other two communication patterns were over 90% consistent. [10] points out that steering inconsistencies can be easily visualized as a program transaction boundary being cut by a steering transaction—some steering events occurred before or during the program transaction while others took place after. With mixed communication patterns where faster processes or process sets are allowed to progress uninterrupted and then forced to synchronize with slower processes, this scenario becomes more likely as can be seen in the mixed communication tests for 8 processes.

Transaction computation size also has a large bearing on the frequency of steering consistency. The consistency average for large transactions is over 95% while small transactions barely surpass an 80% average. Large transactions provide a greater window of opportunity for steering requests to be received within the same global transactions. Conversely, small transactions reduce the size of the window and decrease the likelihood for consistency.

Based on these trends, large process sets with small transactions and a mixed communication pattern are expected to have the largest overhead associated with optimistic steering, since this is the case in which the greatest number of inconsistencies can be found. In fact, only 28% of the time did a consistent steering transaction occur in this case leading to approximately a 16% overhead for steering. With large transactions the overhead dramatically drops to just over 1%.

All in all, the system provides reasonable performance for monitoring and steering, both introducing an average overhead of less than 2%. As was hoped, the percentage of consistent steering transactions was quite high, averaging nearly 90% over all cases. While these test cases are not all encompassing, they represent some basic communication patterns. The transaction sizes are meant to serve as benchmarks for performance testing. While the .25 second transactions size is smaller than we expect in practice, the 2.5 second transaction represents a reasonable, though still conservative, value. One the other hand, by performing 5 steering transactions per run, thus steering approximately every minute in the case of the large transactions, is far more frequent than will exist in practical use. The user must have the ability to comprehend and process what is occurring in a computation to make informed and meaningful steering decisions. Steering is more likely on the scale of once every 1000 transactions so as to observe meaningful patterns within the computation. In practice, we expect the overhead to be lower than observed in our study.

One interesting fact should be pointed out; these are actually the second set of tests. Our initial monitoring performance for 8 processes sometimes presented a 95% overhead, prompting us to investigate the cause of such poor performance. As we mentioned in Section 5.5, our system cannot allow blocking receives because of the possibility of deadlock. Therefore, we utilize MPI_Iprobe to search for both application and control messages. Because of the load imbalance in our tests (up to 10 processes over 4 processors), processes, at times, would have to wait for a requested application message, which we search for by polling using MPI_Iprobe. It was discovered that MPI_Iprobe is in fact a very expensive operation causing over 85% of the excess overhead. By inserting a 1-millisecond sleep between each probe, we were able to dramatically reduce the overhead as reflected in figures 3.4 and 3.5.

7. Related Work

Over the years, several computational steering environments have been proposed [11]. The optimistic approach to computational steering we have developed differs from these systems in that it both allows simultaneous steering of multiple processes and does not require global synchronization before steering.

An early computational steering environment was VASE, the *Visualization and Application Steering Environment*, developed at the University of Illinois [5, 7]. The VASE system requires special annotation to existing Fortran code and permits the user to alter the values of "key" parameters and to add code at "key" points. However, VASE does not support the coordinated steering of multiple processes. SCIRun supports computational steering in a multithreaded application that runs on a single multiprocessor machine [12, 13]. However, it assumes that the underlying program consists of a number of separate modules. The SCIRun system generates a script that controls the invocation of these modules. The steering process involves altering these scripts—thus, changes occur only between modules, not within modules. The script itself is executed sequentially. No support for coordination of distributed changes is required. Progress [15] and its successor Magellan [16] also provide interactive environments for computational steering. Both these systems were designed to run on multiple multiprocessor computers. Progess does not support coordinated steering of multiple processes. Magellan was extended to support such coordinated steering of multiple processes but requires synchronization points be placed in an application. Before a steering change can take place the application must first halt. Thus, Magellan can be said to support the conservative approach to the interactive steering of distributed computations. CUMULVS was developed at Oak Ridge National Laboratory to support the monitoring and steering of distributed computations [2]. To allow steering, the user interface process creates a loosely synchronized connection with the application, which guarantees that all tasks apply the steering updates at the same time or point in the application, also falling into a conservative steering model. Yet another computational steering environment is the VIPER project [14]. VIPER is based on a client/server/client architecture. One client is the parallel computation, the other client is the visualization unit, and the server acts as a governing body for both information and data extraction and steering application. Each application has synchronization points at which time the server has the ability to consistently apply the steering changes requested by the user. Finally, the CSE environment provides a computational steering environment similar to those already described [8, 18]. In this system, there exist data manager and satellite worker processes. The data manager is responsible for gathering the monitored data, all communication, and application of steering changes. Like the other environments that support simultaneous steering events, this system also requires its source code be annotated with special synchronization variables. During synchronization, the data manager can consistently apply the steering changes.

8. Conclusions and Future Work

The EV/IS system offers a real-time environment through which users may gain understanding and perform on-the-fly program manipulation. Although numerous systems exists that provide these functionalities, as discussed in section 7, few support coordinated steering of multiple processes; of those, a conservative steering model is applied. We believe that a more optimistic approach to steering can provide reduced perturbation, as global synchronization is not required before steering events.

While optimistic steering is more complex than conservative steering, the performance shown in section 6 demonstrates that an average overhead of less than 2% is introduced into the monitoring system. Further, the monitoring system itself has an average overhead of less than 2%.   While cases exists in which optimistic steering can produce a large amount of overhead due to rollback, the average case shows that nearly 90% of all steering transactions can occur consistently on the first attempt, thus introducing minimal overhead.

In some situations, conservative steering may be a more ideal model for steering. To explore this possibility, we are presently implementing a conservative steering module into the EV/IS system. Based on the results we obtain from the testing of that system, we will be able to provide a more definitive discussion of the relative costs and benefits of the two approaches.

References

1. Dijkstra and B.P. Scholten, "Termination Detections for Diffusing Computations," *Information Processing Letters,* 1980, 11(1): 1-4.

2. G.A. Geist II, J.A. Kohl, and P.M. Papadopoulos, "CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications." *The International Journal of Supercomputer Applications and High Performance Computing,* 11(2):224-235, 1997.

3. W. Gu, G. Eisenhauser, E. Kraemer, K. Schwan, J. Stasko and J. Vetter, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," *in Proceedings of the Fifth Symposium of the Frontiers of Massively Parallel Computation,* Mclean, VA Feb. 1995, pp. 422-429.

4. Jinhua Guo, Eileen Kraemer, and David W. Miller, "Consistency Detection in a Transaction Based Model." *In submission PODC 2002*.

5. R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "A Distributed Environment for Running Time Visualization and Application Steering in Computational Mechanics." *Computing Systems in Engineering,* 3(1-4):501-515, 1992.

6. E. Kraemer, D. Hart, and G.-C. Roman, "Balancing Consistency and Lag in a Transaction-Based Computational Steering Environment," *in Proceedings, Thirty-First Annual Hawaii International Conference on System Sciences*, 137-147, Jan. 1998.

7. D.J. Jablonowski, J.D. Bruner, B. Bliss, and R.B. Haber, "VASE: The Visualization and Application Steering Environment." *In Proceedings of SuperComputing '93*, pages 560-569, 1993.

8. R. van Liere, J.D. Mulder, and J.J. van Wijk, "Computational Steering." *Future Generation Computer Systems,* 12(5):441-450, April 1997.

9. N. Lynch, "Distributed Algorithms," Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

10. David W. Miller, Jinhua Guo, Eileen Kraemer, and Yin Xiong, "On-the-Fly Calculation and Verification of Consistent Steering Transactions." *In Proceedings SuperComputing 2001*, Denver, CO. November 2001.

11. Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere, "A Survey of Computational Steering Environments." Technical Report SEN-R9816, Stichting Mathematisch Centrum. September 1998.

12. S.G. Parker and C.R. Johnson, "SCIRun: A Scientific Programming Environment for Computational Steering." *In Proceedings of SuperComputing '95*, 1995.

13. S.G. Parker, D.M. Weinstein, and C.R. Johnson, "The SCIRun Computational Steering Software System." In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 1-40. Birkauser Verlag AG, Switzerland, 1997.

14. S. Rathmayer and M. Lenke, "A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications." *In Proceedings of the 11th International Parallel Processing Symposium, IPPS 97*, pages 181-186, 1997.

15. J. Vetter. Computational Steering annotated Bibliogragy. *SIGPLAN Notices*, 32(6):40-44, June 1997.

16. J. Vetter and K. Schwan, "High Performance Computational Steering of Physical Simulations." *In Proceedings of the 11th International Parallel Processing Symposium, IPPS 97*, pages 128-132, 1997.

17. H. Vuppula, E. Kramer, and D. Hart, "Algorithms for Collection of Global Snapshots: An Empirical Evaluation," *in Proceedings, ICSA Conference on Parallel and Distributed Computing Systems*, August 2000, pp. 197-204.

18. J.J. van Wijk and R. van Liere, "An Environment for Computational Steering." In G.M. Nielson, H. Muller, and H. Hagen, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques,* pages 89-110. Computer Society Press, 1997.

CHAPTER 4

CONCLUSIONS

The EV/IS system offers a real-time environment through which users may gain understanding and perform on-the-fly program manipulation. However, with the ability to adjust application parameters on-the-fly in an optimistic manner, comes the necessity to verify the changes are made in a logically consistent manner, at *consistent cuts* [14]. Through the notions of steering transactions, program transactions, and consistent steering, the history-based algorithm presented in Chapter 2 and the vector time based algorithm discussed in Chapter 3 provide a means to verify consistency, detect inconsistency, and calculate the earliest consistent cuts. However, simply finding these inconsistencies is not enough in a robust computational steering system.

As the IS system applies an *optimistic* approach to computational steering, it must possess the ability to correct any inconsistencies that may occur as a result of a steering transactions. To accomplish this, the IS system has the means to checkpoint the state of a computation, log messages, rollback, re-execute, replay messages, and reapply steering changes at consistent cuts. While the complexities of this approach are large and the perturbation can be significant when encountering numerous inconsistencies, in many cases, the benefits of not requiring global synchronization before steering outweigh this fact. To always require that a computation reach quiescence [2, 11] before a steering change takes place guarantees significant computational perturbation as it relates to runtime overhead. As was shown in Chapter 3, based on the tests performed, this system performs relatively well introducing an average overhead of less than 2%. Coupled together with the fact that we found an average steering consistency of nearly 90%, the IS system's optimistic approach can be ideal.

Future work in the area of computational steering will focus the on development of a conservative steering module within the IS system. The results of comparison testing between these approaches will allow this research to provide more definitive discussion as to when one steering approach is more ideal than another based on the number of processes, communication patterns, and transaction size, to name just a few variables of a distributed computation.

REFERENCES

1. K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems,* February 1985, 3(1):63-75.

2. Dijkstra and B.P. Scholten, "Termination Detections for Diffusing Computations," *Information Processing Letters,* 1980, 11(1): 1-4.

3. C.J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering", *Australian Computer Science Communications,* February 1988, pp. 56-66.

4. J. Fowler and W. Zwaenepoel, "Casual Distributed Breakpoints," in Proceedings, *10th International Conference on Distributed Computing Systems,* Paris, France, May 1990, pp. 134-141.

5. G.A. Geist II, J.A. Kohl, and P.M. Papadopoulos, "CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications." *The International Journal of Supercomputer Applications and High Performance Computing,* 11(2):224-235, 1997.

6. R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "A Distributed Environment for Running Time Visualization and Application Steering in Computational Mechanics." *Computing Systems in Engineering*, 3(1-4):501-515, 1992.

7. E. Kraemer, D. Hart, and G.-C. Roman, "Balancing Consistency and Lag in a Transaction-Based Computational Steering Environment," *in Proceedings, Thirty-First Annual Hawaii International Conference on System Sciences,* 137-147, Jan. 1998.

8. D.J. Jablonowski, J.D. Bruner, B. Bliss, and R.B. Haber, "VASE: The Visualization and Application Steering Environment." *In Proceedings of SuperComputing '93*, pages 560-569, 1993.

9. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM 21*, 1978, 7(558-565).

10. R. van Liere, J.D. Mulder, and J.J. van Wijk, "Computational Steering." *Future Generation Computer Systems,* 12(5):441-450, April 1997.

11. N. Lynch, "Distributed Algorithms," Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

12. F. Mattern, “Virtual Time and Global States of Distributed Systems,” *Parallel and Distributed Algorithms*, North-Holland, 1989, pp. 215-226.

13. Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere, “A Survey of Computational Steering Environments.” Technical Report SEN-R9816, Stichting Mathematisch Centrum. September 1998.

14. Robert H. B. Netzer and J. Xu, “Necessary and Sufficient Conditions for Consistent Global Snapshots,” *IEEE Transactions on Parallel and Distributed Systems*, February 1995, 6(2):165-169.

15. S.G. Parker and C.R. Johnson, “SCIRun: A Scientific Programming Environment for Computational Steering.” *In Proceedings of SuperComputing ’95*, 1995.

16. S.G. Parker, D.M. Weinstein, and C.R. Johnson, “The SCIRun Computational Steering Software System.” In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 1-40. Birkauser Verlag AG, Switzerland, 1997.

17. S. Rathmayer and M. Lenke, “A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications.” *In Proceedings of the 11th International Parallel Processing Symposium, IPPS 97*, pages 181-186, 1997.

18. Reinhard Schwarz and Friedemann Mattern, “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail,” *Distributed Computing*, 1994, 7(3): 149-174.

19. J. Vetter. Computational Steering annotated Bibliogragy. *SIGPLAN Notices*, 32(6):40-44, June 1997.

20. J. Vetter and K. Schwan, “High Performance Computational Steering of Physical Simulations.” *In Proceedings of the 11th International Parallel Processing Symposium, IPPS 97*, pages 128-132, 1997.

21. J.J. van Wijk and R. van Liere, “An Environment for Computational Steering.” In G.M. Nielson, H. Muller, and H. Hagen, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques,* pages 89-110. Computer Society Press, 1997.