From the Proceedings of the Workshop on Debugging and Tuning for Parallel Computer Systems
Chatham, MA, October 1994.
Also available as Tech Report GIT-CC-94-58

Opportunities and Tools for Highly Interactive Distributed and Parallel Computing

Greg Eisenhauer
Weiming Gu
Thomas Kindler
Karsten Schwan
Dilma Silva
Jeffrey Vetter

Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Advances in networking, visualization and parallel computing signal the end of the days of batch-mode processing for computationally intensive applications. The ability to control and interact with these applications in real-time offers both opportunities and challenges. This paper examines two computationally intensive scientific applications and discusses the ways in which more interactivity in their computations presents opportunities for gain. It briefly examines the requirements for systems trying to exploit these opportunities and discusses Falcon, a system that attempts to fulfill these requirements.

1 Introduction

The world of computationally intensive computing is moving away from the batch-oriented style of processing. Users accustomed to spreadsheets and WYSIWYG word processing are not satisfied with the traditional hands-off, you'll-get-your-data-when-the-batch-queue-empties mode of running parallel programs. At the same time, high-speed network interfaces and the proliferation of high-end graphics workstations offer an opportunity to open new windows into application behavior. Falcon, a system developed at Georgia Tech, provides tools and techniques for exploiting these developments, and it uses them to create new opportunities for application understanding, debugging and tuning.

Traditional debuggers rely on halting the system in order to examine and modify the program state. While such debuggers are useful, they are often inadequate to detect the race conditions, synchronization errors or other problems endemic to parallel and distributed programs. Similarly, traditional uniprocessor code profilers and analyzers have a role in tuning parallel programs, but they are ineffectual for analyzing synchronization overheads, bursty computational demands, or other problems more unique to non-sequential applications. Neither type of tool provides the insight into dynamic program behavior that is often necessary to debug and tune parallel and distributed programs. Perhaps more importantly in the long term, neither type of tool encompasses mechanisms for dynamically manipulating running programs.

To address these deficiencies one needs mechanisms for "observing" a running application and "adjusting" its state or behavior. Collectively, these mechanisms are a *monitoring and steering* system[GVS94]. The on-line manipulation or steering of parallel and distributed programs has been shown to result in performance improvement in many domains. Examples of such improvement include the automatic configuration of small program fragments for maintaining real-time response in uniprocessor systems[MP89], the on-line adaptation of functional program components for realizing reliability versus performance tradeoffs in parallel and real-time applications [BS91, GS89, GS93], and the load balancing or program configuration for enhanced reliability in distributed systems[SGB87, MW91, Bec94].

Further benefits are gained if monitoring and steering mechanisms are not limited to system-level constructs but are instead made available in a reasonable way at the application level. In addition to supporting standard program tuning practices, application-level monitoring and steering have the potential to produce real gains in application productivity by allowing users to accomplish more useful work with the same number of compute cycles. How is this possible? Truly interactive parallel programs, created with application-level monitoring and steering, will give users significant insight into the *progress* of the computation. If users have access to the computation and the ability to guide or direct the computations at runtime, they have an unparalleled power to evaluate and experiment with the program.

Consider the case of scientific computing, where many applications are trying to model or simulate the real world. A truly interactive program would let users interact with that world as it evolves. Rather than planning a dozen batch-style simulation runs with a dozen different parameter values, the user can adjust the application dynamically and examine the response. Rather than discovering at the end of a twenty hour simulation run that the system wandered into an unreasonable state early on, it can be monitored for reasonableness as it progresses. These *process tuning* situations are often neglected because they do not fall into the traditional realm of program tuning or debugging. However, the advantages of additional high-level insight into the application in these cases are an important benefit of interactive parallel computing.

This paper first examines two computationally intensive scientific applications and discusses the ways in which more interactivity in their computations presents opportunities for gain. It then briefly discusses the requirements and techniques for exploiting these opportunities and examines the aspects of Falcon which fulfill these requirements. The paper also presents our conclusions and plans for future work.

2 Interaction Opportunities in Selected Applications

Parallel and distributed programming models and applications vary widely, as do the situations in which one might wish to employ a monitoring and steering system. The utility of a general, low-perturbation monitoring system in both debugging and performance tuning is widely accepted. Unfortunately, most such systems do not make their facilities easily accessible at the application level. If we are to realize the application-level process gains discussed in the introduction we must consider the demands of application-level monitoring and determine how steering might be used. This section examines two large parallel applications and discusses ways in which a monitoring and steering system might benefit each.

2.1 MD

MD is an interactive molecular dynamics simulation developed at Georgia Tech in cooperation with a group of physicists exploring the statistical mechanics of complex liquids [XORL92, EGSM94]. The specific molecular dynamics systems being simulated are n-hexadecane (C_{16}-H_{34}) films on a crystalline substrate $Au(001)$. In the simulation, the alkane system is described via intramolecular and intermolecular interactions between pseudoatoms (CH_2 and terminal CH_3 segments) and the substrate atoms. The calculational cell is a square cylinder which is periodically repeated in the x-y directions. Temperature is controlled via infrequent scaling of the particles' velocities. The alkanes remain associated in a chain with very predictable bond lengths throughout the simulation. A typical small simulation contains 4800 particles in the alkane film and 2700 particles in the crystalline base. A visual representation of this physical system appears as Figure 1.

For each particle in the MD system, the basic simulation process takes the following steps: (1) obtain location information from its neighboring particles, (2) calculate forces asserted by particles in the same molecule (*intra-molecular forces*), (3) compute forces due to particles in other molecules (*inter-molecular forces*), (4) apply the calculated forces to yield new particle position, and (5) publish the particle's new position. The dominant computational requirement is calculating the long-range forces between particles, but other required computations with different characteristics also affect the application's structure and behavior. These computations include finding the bond forces within the hydrocarbon chains, determining system-wide characteristics such as atomic temperature, and performing analysis and on-line visualization.

The implementation of the MD application attains parallelism by domain decomposition. The simulation system is divided into regions, and the responsibility for computing forces on the particles in each region is assigned to a specific processor. In the case of MD, we can assume that the decomposition changes only slowly over time and that computations in different sub-domains are independent outside some cutoff radius. Inside

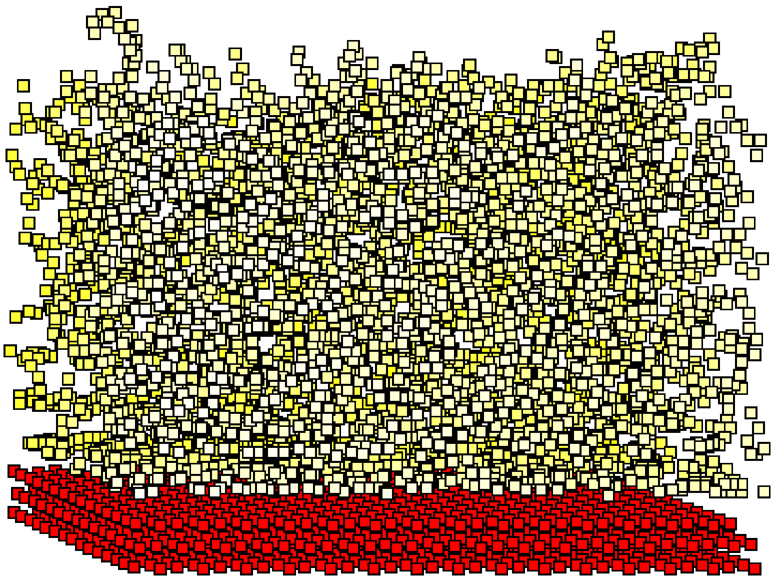

Figure 1: A visual representation of a sample system for the molecular dynamics simulation. The white-yellow particles are the pseudoatoms of the alkane chains. The red particles represent the gold substrate.

this radius information must be exchanged between neighboring particles, so that different processors must communicate and synchronize between simulation steps. The resulting overheads are moderate for fairly coarse decompositions (e.g., 100-1000 particles per process) but unacceptable for finer grain decompositions (e.g., 10 particles per process).

The MD simulation offers many opportunities to improve the performance through both on-line interactions with the end user and program steering by algorithms, including:

- Decomposition geometries could be changed to respond to changes in physical systems. For example, a slab-based decomposition may be useful for an initial system, but a pyramidal decomposition might be a better choice if a probe is lowered into the simulated physical system.
- The interactive modification of the cutoff radius could improve solution speed by computing uninteresting time steps with some loss of fidelity, if this is desired by the end user.
- The boundaries of spatial decompositions could be shifted for dynamic load balancing among multiple processes operating on different sub-domains. This can be performed by an algorithm or by end users.
- Global temperature calculations, which are expensive operations requiring a globally consistent state, could be replaced by less accurate local temperature control. On-line analysis could determine how often global computations must be performed based on the temperature stability of the system.

From our experience with MD, we believe that these are important opportunities to exploit in order to increase the usability and efficiency of the application. For example, we have seen that the performance of the application is extremely sensitive to load balance shifts which can dramatically limit efficiency with even moderate numbers of processors. The ability to dynamically rebalance and perhaps even reconfigure the decomposition to match the evolving physical system is essential to performance for a long-running system.

2.2 Atmospheric Modeling

The simulation of complex global natural phenomena is one of the biggest challenges facing computational science because of its extreme computational and data handling requirements. The ultimate goal in climate modeling, the simultaneous simulation on a global scale of physical and chemical interactions in ocean and atmosphere, is still far from reach. It is difficult to run and test a model with typical runtimes of hours for each simulation day. One simple reason for this is that changes to a model often do not have the desired effects upon the model results. This occurrence is particularly common when parameters must be chosen to simulate processes that are not well understood or whose influence can only be approximated at the scale of the current model. The result in these cases is a set of sometimes arbitrarily chosen parameters that must be adjusted individually. On-line visualization, interaction and program steering have potential to simplify and significantly shorten model development time and improve model results as well as to help to improve traditional measures of simulation performance.

Earth and atmospheric scientists at Georgia Tech have developed a global chemical transport model (GCTM)[KSS+94] which uses assimilated windfields [SO93] for the transport calculations. These types of models are important tools to answer scientific questions concerning the stratospheric-tropospheric exchange mechanism or the distribution of species such as chlorofluorocarbons (CFC's), hydrochlorofluorocarbon (HCFC's) and ozone. This model uses a spectral approach to solve the transport equation for each species. In a spectral model, all variables are expanded into a set of orthogonal spherical basis functions, called spherical harmonics. Derivatives with respect to the latitude or the longitude are more easily and accurately calculated in this spectral domain, though the variables must be transformed back into a grid domain for the chemistry calculations. Details of this solution approach, which is quite common in global models, can be found in [Hau40], [Sil54], [KHYK61], [WP86] or [FW94]. Our model contains 37 layers, which represent segments of the earth's atmosphere from the surface to approximately 50 km, with a horizontal resolution of 42 waves or 946 spectral values. In a grid system, this corresponds to a resolution of about 2.8 degrees by 2.8 degrees. Thus in each layer 8192 gridpoints have to be updated every time step. A typical time step increment is 15 simulated minutes. Figure 2 represents a visual sample from this application.

There are many ways in which more interactivity in this parallel application could significantly benefit end users. For example, a typical problem in model development is that there are dramatic differences in scale between some global phenomenon and the many physical processes that comprise it. Gross measures such as vertical windfields have small values on a global scale, yet on a smaller scale phenomenon such as thunderstorms cause large vertical air displacements and play important roles in vertical mixing in the atmosphere. Computing the entire globe on a scale where all such phenomena could be accurately represented is far too computationally expensive to consider. One way of approaching this problem is to use parameterizations inside models which bear an indirect relationship to the small-scale phenomenon and that attempt to match observed phenomenon on the global scale. Unfortunately, the construction of these parameterized models is an exploratory and error-prone process. Section 3.1.1 describes ways in which basic interactive monitoring and steering can aid in this model construction process.

Other more ambitious approaches to the same problem might involve allowing the user to interactively identify interesting subareas for simulation at a higher resolution in time and/or space. This differential focus approach would allow those regions to be modeled with better fidelity without invoking the huge computational cost of using a higher resolution uniformly over the entire model. In some cases these areas might be selected algorithmically, but in other cases what constitutes an interesting situation or area could depend upon a subjective judgment by a human observer.

One can certainly imagine writing a program with a user interface that allows this level of interaction, but our goal is to achieve this without turning scientists into graphical user interface (GUI) programmers. The next section presents some tools and techniques that we have developed to further this goal.

3 Requirements and Tools for Interactive Computing

This section examines the general techniques and tools required to support program tuning in general and specifically to support the user/program interactions presented above. We first examine examples of some displays that support the interaction goals discussed previously. Then we discuss the monitoring and steering systems required to create and support these displays.

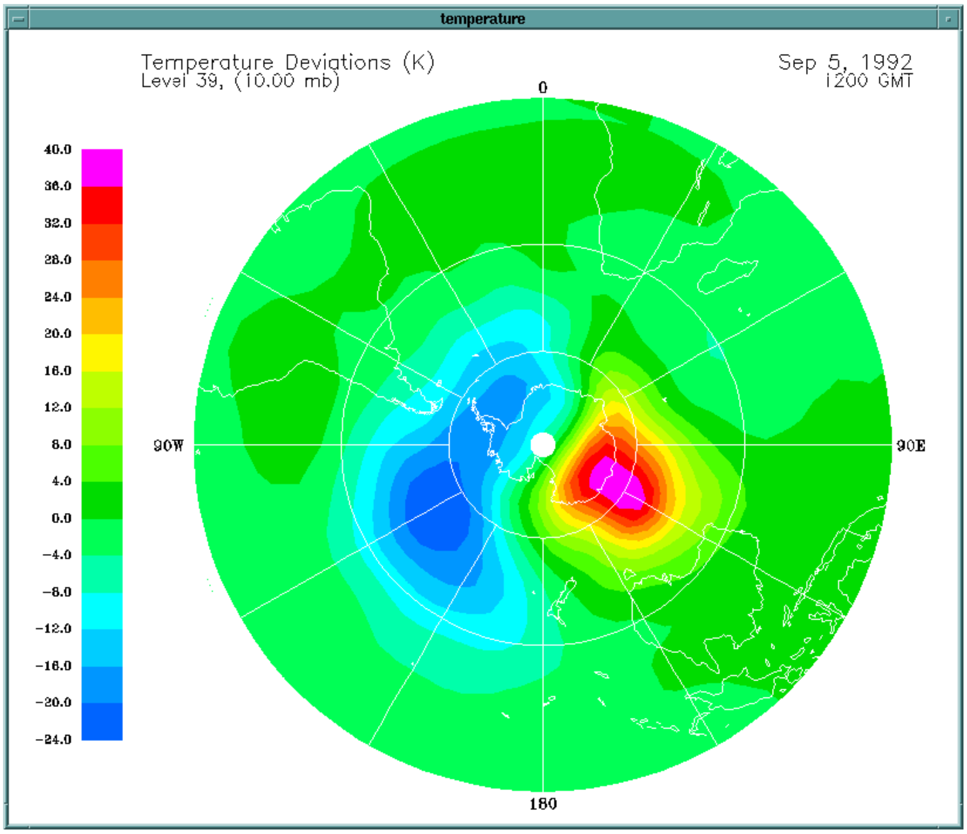

Figure 2: A sample plot of southern hemisphere temperature distribution as used by the global climate transport model.

3.1 Displays

Displays which are useful for understanding application behavior vary as widely as do applications and programming models. It is not possible, within the bounds of this paper, to survey all possible displays or even all useful approaches to display construction. Instead we present sample displays and interactions so that we can explain the monitoring and steering infrastructure required to support them. This section presents two types of displays targeted to different levels of abstraction in a arallel program. The first example is an application-specific display of the type required to achieve some of the process-oriented gains in application development discussed above. The second example is a programming-model specific display useful for program debugging and tuning.

3.1.1 Application Specific Displays

The previous section has indicated that user interactive steering has the potential to improve an application's performance and functionality. However, many uses of steering are application specific and so are graphical displays that are used to present the run-time program and performance information to the end user and to accept the user's steering commands. By examining a sample display used for steering the atmospheric modeling code, we can explore how these displays are used to understand and control the application and how they are interfaced with other parts of the monitoring and steering system.

The graphical display discussed in this section is specifically built for interactive steering of an atmospheric modeling code that simulates the distribution of atmospheric species such as Carbon 14 (C^{14}) and CFC. Figure 3 shows a screen display of the distribution of C^{14} at a latitude of 70° N. The display has two logical

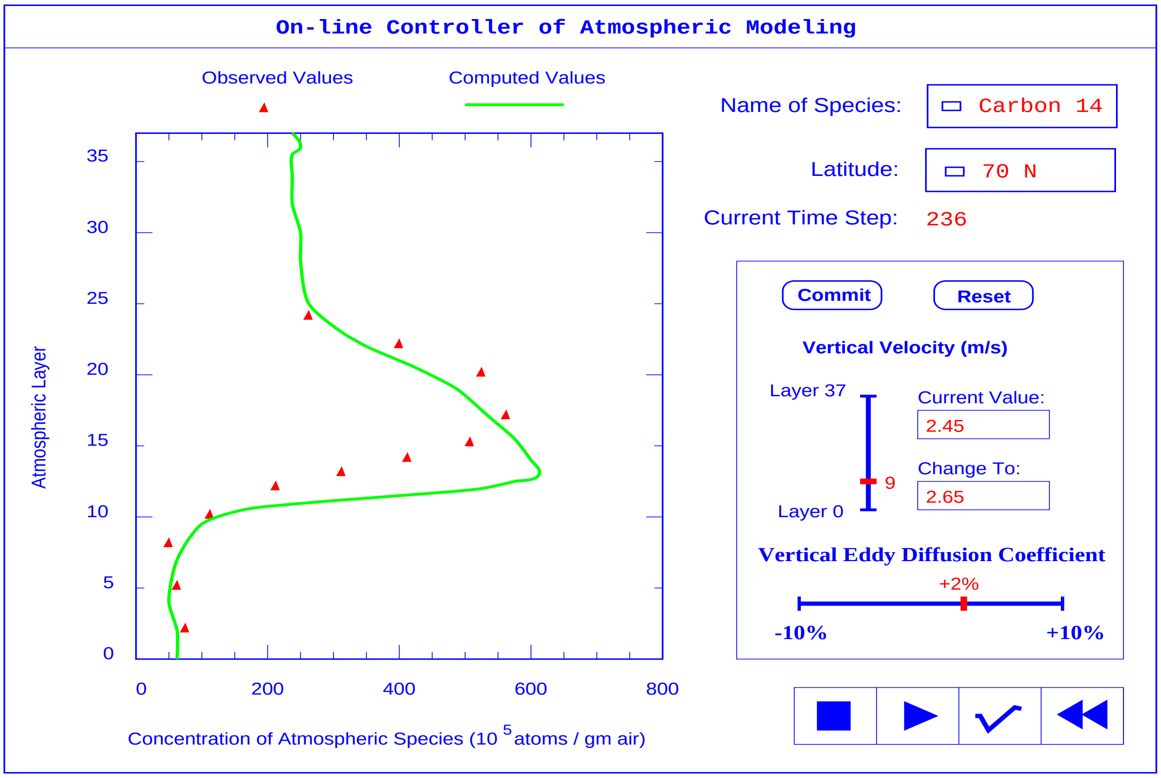

Figure 3: An application specific display for on-line control of the atmospheric modeling code.

parts: one for showing both the computed and the observed concentration values of C^{14} atoms in air to the end user, and the other for accepting steering requests from the user. The computed results of the C^{14} distribution is represented by a plotted curve from atmospheric layer 0 to 37, and it is updated for every model time step. The observed C^{14} concentration at a number of atmospheric layers is represented by discrete triangular points, and is used to judge whether the current computation is "correct" or "wrong." When noticeable discrepancies between the calculated values and the observed values are detected, the user can dynamically modify the application execution to "correct" the computations. For example, the curve shown in Figure 3 demonstrates that the computed concentration of C^{14} is consistently higher than the observed values from layer 10 to 15, but it is lower from layer 16 to about 23. The simulation may be adjusted to remove this discrepancy; the end user can alter the vertical wind velocity at these atmospheric layers. After typing in new vertical wind velocity values, the user needs to click the commit button on the display to send the steering command to the application. The program will use these new parameters for computations from the next model time step. The user can also stop the application's execution (by clicking the □ or stop button), change parameters, and restart the execution (by clicking the ▷ or play button). Before restart, the user can rollback the computation to a previously checkpointed time step (by clicking the ◁◁ or rewind button). At any point the user can checkpoint the application execution (by pressing the checkpoint button). The user can also use the application's default checkpointing policy which automatically saves execution history after a predefined number of time steps.

The above application specific display has a two-way communication link with the application code. In one direction, it receives computed and observed concentration values of atmospheric species from the application, and displays these values to the user. In the other direction, the display accepts steering commands and sends them to the application. A clean interface between the display and the application code is needed. Our Falcon system provides a flexible mechanism of dynamically connecting and disconnecting displays to the application. This mechanism will be addressed in Sections 3.2 and 3.3.

3.1.2 Programming Model Specific Displays

Both of the application programs described in Section 2 have been implemented on shared-memory multiprocessors using a threads-based programming model. In this model, independent threads of computation are created on the various processors, and they control access to shared data by using mutex locks and conditions[CD88]. The amount of time a thread spends waiting to be granted a lock or for a particular condition to occur directly impacts the amount of useful work it can perform in a given time. Therefore, understanding the interactions of threads over time is one of the most important aspects in understanding the behavior of these programs.

This section examines one display that we have found useful in diagnosing performance problems in threads-based programs. The threads lifetime view depicted in Figure 4 shows thread behavior over time. In particular, it represents each thread as a horizontal bar which assumes different colors and patterns when the thread is in different states, such as running, waiting for a mutex, or waiting for a condition. A vertical line is drawn from the parent thread to the child thread at the time of thread fork event. When a thread joins another thread after it exits or when a detached thread calls thread_exit, the narrow bar representing the thread terminates. In the case of thread_join, another vertical line is drawn from the caller thread to the thread to which it joins. The space after a joined thread can be reused by threads forked later. The display contains buttons with which one can move around and zoom in on different threads and regions of time. In addition to this lifetime display, there is another simultaneous display that relates thread colors to their names. We have excluded this name mapping view for space considerations.

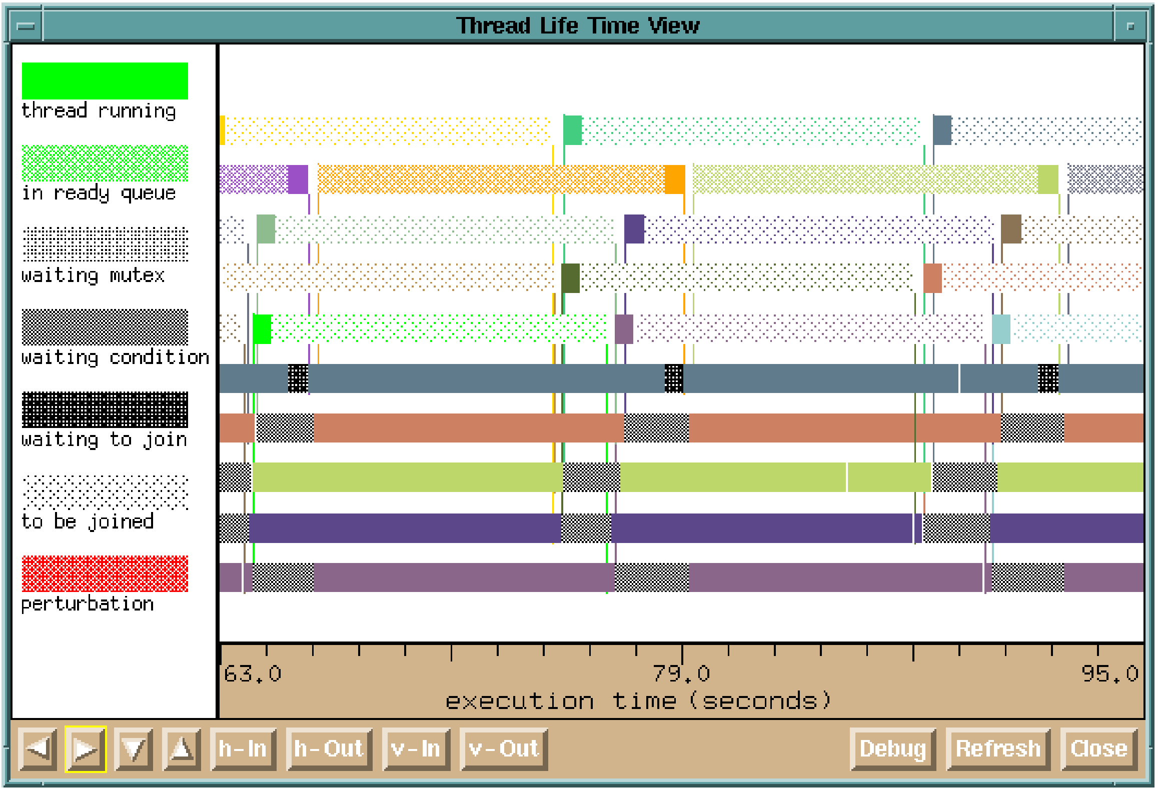

Figure 4: A thread life-time display derived from traces of MD program behavior.

The particular set of threads shown in Figure 4 represents a snapshot of the MD application's execution with the molecules partitioned into five domains. The bottom five threads are threads responsible for calculating the inter-molecular forces for each domain. These threads live for the duration of the calculation. The top five positions are held by threads that calculate intra-molecular forces. These threads are forked by the inter-molecular thread only for a single timestep in the simulation. At the end of their calculations for that timestep, they perform a join operation and exit when both they and the thread that forked them complete the calculations for that timestep. When the next iteration begins, a different thread (with a different color) is forked and the process repeats.

Figure 4 is interesting for MD because it makes clear some aspects of thread synchronization that are difficult to determine without such a display. Each domain must acquire updated particle location information from its neighboring domains before it can proceed with the next iteration. This waiting time appears as the grey "waiting for condition" state in the display. Rather than requiring *all* threads to finish before *any* thread begins the next iteration, MD domains only synchronize with their immediate neighboring domains. This allows individual domains to begin the next iteration even before other threads have completed the current iteration. This flexibility helps MD compensate for the effects of minor variances in load balance between domains. In the figure one can see that blocks of solid compute time, which occur when a domain starts a new iteration, occur first in the second thread from the bottom and later in threads of more distant domains.

To produce displays of this type, the important constructs in the programming model must be instrumented. In this case, we have instrumented the Cthreads parallel programming library[Muk91] so that every operation that can affect the state of a thread produces a record in an event stream. In order to produce a reasonable display, these events must contain accurate timestamps and they should not excessively disturb normal execution of the program. The next section will discuss the system requirements and tools in Falcon that support the sample displays presented above.

3.2 On-line Monitoring

The first step to interactivity is gaining easy access to the applications' run-time information. This information ranges from records of the utilization of processors to detailed execution and waiting times spent by each processor and from values of certain variables (e.g., "temperature" of a simulated molecular system, "concentration" of an atmospheric species) to complete current program states of the application. Therefore, the capture, collection, and analysis of on-line program and performance information should be an integral component of any system which supports interactivity. Instead of focusing on supporting on-line interactivity, however, past work in program monitoring has focused on helping programmers understand the performance of their parallel codes, minimizing or correcting program perturbation due to monitoring, reducing the amounts of monitoring or trace information captured for parallel or distributed program debugging [OSS93, HMC94], and the effective replay [LMC87] or long-term storage of monitoring information. In comparison, interactivity, in the form of on-line program steering, specifically requires its *on-line* monitoring system to be able to: (1) capture *application-specific* information, (2) impose *controlled overheads* on the execution of monitored applications, (3) deliver monitoring information with *low latency*, and (4) provide incremental analysis of monitoring information vital for on-line steering.

The monitoring system is required to handle application-specific data because much of program steering is inherently application-specific. With MD, for example, steering can be used to improve load balance based on the molecule partitions and boundaries of these partitions. The boundaries can be adjusted by the user during program execution to obtain a better load balance. In steering the atmospheric modeling code, parameters concerning certain atmospheric species can be dynamically changed to effect different results on these atmospheric species. In addition, application-specific monitoring permits non-computer science end users to view, analyze, and steer their programs in terms of their specific attributes (e.g., "time step size" or "current energy").

Controlled monitoring overheads are useful for several reasons. First, since one purpose of application steering is to improve program performance, excessive monitoring overheads can easily offset the performance gains obtained by steering. Second, steering decisions based on inaccurate information may produce unexpected results. In the case of MD steering based on the work load information of each processor, perturbed information can cause inaccurate, sometimes unnecessary, adjustments of partition boundaries. In the worst case, thrashing of boundaries can occur and application execution will actually be slowed.

Steering latency is the period of time between the occurrence of a program activity or state and the time when it is acted upon by a steering agent; monitoring latency is the period of time between the capture of an activity by the on-line monitor and the passage of that activity to the steering mechanism. Excessive monitoring and steering latencies can cause steering decisions to be made based on obsolete program and performance information, which can result in unpredictable and often negative effects on an application's execution. In the atmospheric modeling code, if the visualized windfield and values of atmospheric species are presented to end users several time steps behind the actual application execution, users may adjust parameters based on "old" information.

Falcon – an integrated system for on-line monitoring and steering of large-scale parallel and distributed applications – is designed to incorporate the attributes necessary for effective on-line monitoring and steering. An overview of the Falcon monitoring system is presented next, followed by discussions of its mechanisms for code instrumentation, event collection, and on-line trace data analysis[1]. Falcons program steering system will be described in Section 3.3.

3.2.1 System Overview of Falcon

Falcon is a set of tools that collectively support on-line program monitoring and steering of parallel and distributed applications. There are three major conceptual components of the on-line monitoring component of Falcon: (1) a monitoring specification and instrumentation mechanism, which consists of a low-level *sensor specification language*, a high-level *view specification language*, and an instrumentation tool, (2) mechanisms for on-line information capture, collection, filtering, and analysis, and (3) a graphical user interface and some graphical displays for interfacing with the end user. These components are shown in Figure 5.

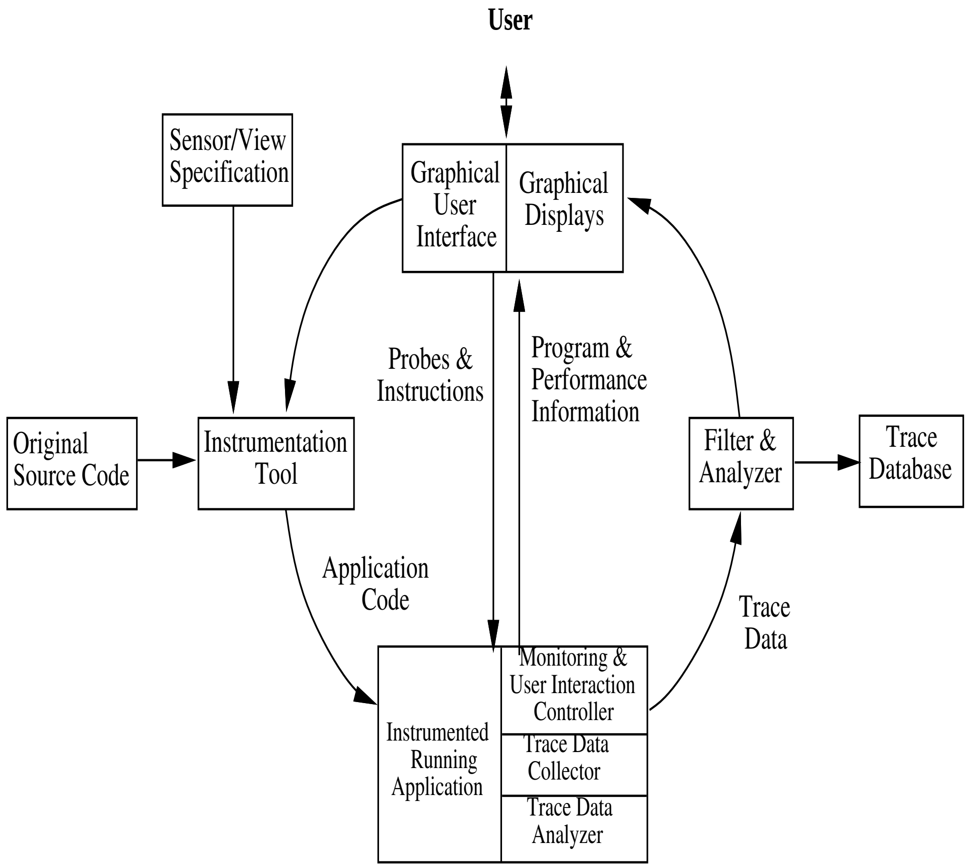

Figure 5: Conceptual components of Falcon.

The following steps are taken when using Falcon. First, application code is instrumented with sensors

[1] A more detailed description of the complete Falcon system and its performance can be found in [GEK+94]

and probes generated from sensor and view specifications. Such monitoring specifications allow users to express specific program attributes to be monitored and based on which steering may be performed. During program execution, program and performance information of interest to the user and to steering algorithms is captured by the inserted sensors and probes, and the information collected is partially analyzed. Falcon's runtime facilities consist of monitoring data output queues attaching the monitored user program to a variable number of additional components performing low-level processing of monitoring output. Partially processed monitoring information is then fed to the central monitor and graphical displays for further analysis and for display to end users. Trace information can also be stored in a trace data base for postmortem analysis.

The monitoring and user interaction 'controllers' in the Falcon runtime system activate and deactivate sensors, execute probes or collect information generated by sampling sensors, and also react to commands received from the monitor's user interface. For performance, these controllers are divided into several *local monitors* residing on the monitored program's machine so that they are able to rapidly interact with the running program. In contrast, the *central monitoring controller* is typically located on a front end workstation or on a processor providing user interface functionality.

3.2.2 Instrumentation and Monitoring Specification

Instrumentation of a target application and its run-time system is the first step toward application steering. Hardware monitoring and data collection require instrumentation of the hardware platform on which the target application is running. Software monitoring and data collection require instrumentation of the program's source code, the system libraries, the compiler, or any combination of the above. We do not rely on hardware monitoring due to its cost, inherent inflexibility and inability to provide high-level application-specific monitoring information.

Software instrumentation points are called *sensors* in Falcon. Falcon offers three types of sensors: sampling sensors, tracing sensors, and extended sensors. A *sampling sensor* is associated with a counter or an accumulator. When a sampling sensor is activated, the associated counter value is updated. A *tracing sensor* generates timestamped event records that may be used immediately for program steering or stored for postmortem analysis. In either case, trace records are stored in *trace queues* from which they are removed by local monitors. An *extended sensor* is similar to a tracing sensor except that it also performs simple data filtering or processing required for steering before producing output data. Sampling sensors inflict less overhead on the target application's execution than tracing and extended sensors. However, the more detailed information collected by tracing sensors may be required for diagnosis of certain performance problems in parallel codes. Furthermore, the combined use of all three sensor types enables users to balance low monitoring latency against accuracy requirements concerning the program information required for program steering.

In order to control monitoring overheads, sensors can be controlled dynamically *selectively* to monitor only the information currently being used by the end user or the steering algorithms. First, sensors may be turned off if events captured by those sensors are not currently used by the end user or the steering algorithm.[2] Second, sampling and tracing rates can be dynamically reduced or increased depending on monitoring load and tolerance of inaccuracies in monitored information. For example, a tracing sensor that monitors a frequently accessed mutex lock can reduce its tracing rate to every five mutex lock accesses, thereby improving monitoring perturbation at the cost of reducing trace accuracy. A selective monitoring example can be found in the MD code, where a large amount of execution time is spent in a three-level nested loop computing forces between particles. At each loop level, distances between closest points of particles and bounding boxes of molecules are calculated and compared with the cutoff radius to eliminate unnecessary computations at the next loop level where specific particles are considered. To evaluate the efficiency of this scheme, at each loop level we use a "cheap" sampling sensor to monitor the hit ratio of distance checks and a more "expensive" tracing sensor to monitor the correlations between the calculated distance and hit ratio at the next loop level. To reduce the perturbation, the "expensive" tracing sensor is not turned on until ineffective distance checks are detected.

Using Falcon's monitoring specification language [Sno87], programmers may define application-specific sensors for capturing both the program and performance behavior to be monitored and the program attributes

[2] Related work by Hollingsworth and Miller [HMC94] removes instrumentation points completely to reduce the overheads of these turned-off instrumentation points to zero.

based on which steering may be performed. The specification of a sample tracing sensor is shown below:

```
sensor work_load {
        attributes {
                int     domain_num;
                double  work_load;
        }
};
```

The sensor work_load is used to monitor the work load of each molecule domain partition in MD. It simply describes the structure of the application data to be contained in the trace record generated by this sensor. This declaration generates the following sensor subroutine.

```
int
user_sensor_work_load(int process_num, double work_load)
{
    if (sensor_switch_flag(SENSOR_NUMBER_WORK_LOAD) == ON) {
        sensor_type_work_load  data;
        data.type = SENSOR_NUMBER_WORK_LOAD;
        data.perturbation = 0;
        data.timestamp = cthread_timestamp();
        data.thread = cthread_self();
        data.process_num = process_num;
        data.work_load = work_load;

        while (write_buffer(get_buffer(cthread_self()), &data,
                     sizeof(sensor_type_work_load)) == FAILED) {
            data.perturbation = cthread_timestamp() - data.timestamp;
        }
    }
}
```

Note that there are four *implicit fields* for any event record that describe the event's sensor type, timestamp, thread id, and perturbation. The body of this subroutine generates entries for an event data structure, then writes that structure into a trace buffer. A local monitor later retrieves this structure from the buffer. Each sensor's code body is also surrounded by an `if` statement, so that it can be turned on or off during program execution.

3.2.3 Event Collection and On-line Trace Analysis

In many monitoring systems, all monitoring activities, including trace data capture, collection, and analysis, are performed by code inline with the thread of user computation. One problem with this approach is that the target application's execution is interrupted whenever a monitoring event is generated and processed. The lengths of such interruptions are arbitrary and unpredictable if complicated on-line trace analysis is used. This may be acceptable with off-line monitoring mechanisms in which monitoring events are written into files for postmortem consumption. For on-line monitors, however, this approach can produce unacceptable perturbation. Instead of performing monitoring activities in the user's code, Falcon uses concurrent monitoring, where most monitoring activities are on processors not running application code.

As depicted in Figure 5, local monitors perform trace data collection and processing, concurrently and asynchronously with the target application's execution. Local monitors and steering controllers typically execute on the target program's machine, but they may run concurrently on different processors, using a buffer-based mechanism for communication between the application and the monitoring mechanism. Therefore, the only direct program perturbation caused by Falcon is the execution of embedded sensors and the insertion of trace records into monitoring buffers. Such perturbation is generally predictable, and its effects on the correctness of timing information can be eliminated using straightforward techniques for perturbation analysis [MRW92].

In order to control monitoring overheads and latency, Falcon's runtime system may itself be configured or steered in several ways, including changing the number of local monitors and communication buffers to configure the system for parallel programs and machines of different sizes. Such changes permit the selection of suitable monitoring performance for specific monitoring and steering tasks, and they may be used to adapt the monitoring system to dynamic changes in workload imposed by the target application. For example, when heavy monitoring is detected by a simple monitor-monitor mechanism, new local monitoring threads may be forked. Similarly, when bursty monitoring traffic is expected with moderate requirements on monitoring latency, then buffer sizes may be increased to accommodate the expected heavy monitoring load. Such parallelization and configuration of monitoring activities is achieved by partitioning user threads into groups, each of which is assigned to a specific local monitor. When a new application thread is forked, it can be added to the local monitor with the least amount of work.

The amount of trace data generated by inserted sensors and collected by the run-time monitoring mechanism is usually too large and the information too low-level to be directly useful to any human user. Trace data filtering and analysis must be performed to generate information that is interesting to end users. Related research concerning on-line trace analysis includes Snodgrass' work on *update networks* [Sno82] and our own past work on real-time monitoring [OSS93]. In [Sno87], information to be monitored is modeled by temporal relations in a hierarchical structure with primitive relations at the bottom of the structure and composed relations at the top. The resulting hierarchy of relations is transformed into an update network – a directed acyclic graph, in which the tuples of the primitive relations enter the nodes at the bottom and the tuples of the composed relations flow out of the nodes at the top.

Falcon offers a flexible on-line trace analysis mechanism similar to update networks. However in Falcon's approach, trace data is processed in different physical components of the monitoring system. At the lowest level, simple trace data filtering and analysis can be performed by the extended sensors. For example, in the atmospheric modeling application, values of windfields may be filtered or eliminated since their complete visualization is expensive. At the local monitor level, trace data is further analyzed to produce higher level information. As in the steering of the atmospheric modeling code in Section 3.1.1, discrepancies between the computed values of an atmospheric species and the observed values can be detected by simple algorithms. Finally, trace data analysis can be performed by separate processes linked with the central monitor. An example of such an analysis process is presented next, and problems to be dealt with when performing on-line trace analysis will be discussed in more detail.

3.2.4 On-line Event Ordering

Displays like the thread life-time view of Figure 4 can provide users with insights into program progress and correctness. However, such displays generally have strict requirements in terms of the accuracy of the timestamps that they expect and the order in which events are presented to them. Misorderings can both confuse users and cause failures of the animation itself. For example, natural causal ordering would require that a thread_fork event precede any event executed by the newly created thread. A display that shows a child running before it has been forked by its parent does not make any sense. Furthermore, suppose that the first event for this child thread is a condition_wait event. In the thread life-time view of Figure 4, this event is represented by a change in the color and fill pattern of that thread's horizontal bar. However, if the thread_fork event has not been received by the display system, the horizontal bar does not yet exist. When the display system attempts to perform a color-change action on this non-existent object, it may crash.

The out-of-order events that cause problems for the display system cannot have occurred in the program's execution. Instead, misorderings existing in the event stream are due to the buffering and processing methods employed in the monitoring system. The diagnosis and correction of out-of-order events is a common problem in parallel and distributed monitoring systems. Existing systems (e.g., ParaGraph[HE91] and SIEVE[SG92]) rely on a sort by timestamp value to impose a total order on all events stored in event files. The on-line nature of the Falcon monitoring system precludes any use of such a solution, and sorting by timestamp order does not entirely eliminate the problem of out-of-order events[BS93]. In addition, coarse clock granularities and poor clock synchronization among different processors may lead to event timestamps that do not accurately reflect the actual order of program execution.

Falcon offers a general mechanism for approaching this problem. In particular, all events are processed by an *ordering filter* before they are sent to the display system. This filtering algorithm follows a "minimumintervention policy." Specifically, it examines each event in the stream arriving from the monitoring system, checks the applicable ordering rules for this event type, and if no rules are violated, forwards the event to the display system. If a rule violation is indicated, the event is held back until the rules are satisfied. As an example, consider the ordering rule that the lifetime view of Figure 4 uses to enforce orderings for a mutex lock event. Actually, a mutex lock is recorded as two separate events: a mutex_begin_lock event indicating that a thread has attempted to obtain the lock and a mutex_end_lock event indicating that a thread has succeeded in obtaining the lock. The following ordering rule is observed by the filter for a mutex_end_lock:

```
mutex_end_lock t m n <- ((thread_init t || thread_fork pt t) &&
                          (mutex_init m || mutex_alloc m) &&
                          (mutex_unlock m n-1) )
```

The parameters associated with the event mutex_end_lock are t, the id of the thread attempting to obtain the lock, m, the id of the mutex variable, and n, the sequence number indicating the number of successful lock attempts on this particular mutex variable. This rule may then be translated as: "a mutex_end_lock event with parameters t, m, and n, may be passed on to the display system if thread t has been initialized or forked by a parent thread, mutex variable m has been initialized or allocated, and the mutex_unlock event for variable m, sequence number n - 1 has already been passed on to the display system." Armed with similar rules for other events, the ordering filter can enforce sufficient ordering to ensure proper functioning of the thread lifetime display. Note that at present, this system only addresses the issue of event ordering. This is adequate to compensate for minor clock variations, but perhaps insufficient when the clocks on different processors vary widely. However this system may provide the basis for more general approach to the timestamp problem as we extend Falcon to more distributed systems.

This section has examined the basic components of the on-line monitoring system of Falcon. The next section presents our approach to the other component of interactivity, on-line steering.

3.3 Interactive Steering

As high performance computing applications move away from the batch-oriented style of processing, making these applications interactive is a daunting task. The challenge exists not only in building new applications with interactivity, but also in reengineering existing applications to become interactive ones. A few programmers turn directly to integrated graphical user interfaces to build interactivity into their applications, but this approach is fraught with difficulties. First, most developers of the high performance computing applications are non-computer scientists, who may not have the background or the inclination to become GUI programmers. Second, most high performance computing systems are not known for high performance graphics support. Increasingly high performance front-end workstations tend to offer better graphics and visualization support, both in hardware and software, and are therefore a better place for running graphics-intensive code. However, the construction of such distributed computation and visualization systems is far from easy.

The interactive steering discussed in this paper offers an alternative way of providing interactivity to the high-performance applications. This approach separates the interactive activities from the computation-intensive part of the application and provides a dynamic link between these components. The responsibilities of such a steering component are to receive the application's run-time information from its coupled on-line monitoring system, display the information to the end user or submit it to a steering agent, accept steering commands, and enact changes that affect the application's execution. The application code is not directly exposed to the interaction with the user or other steering agents, but it needs to be instrumented with sensors which capture run-time information and provide entries for steering commands which may change the program's execution behavior. The basic requirement for steering is that the application code should behave correctly under any valid steering command. Other requirements can be derived by examining its use.

3.3.1 What is interactive steering?

Interactive steering can be defined as the interactive control and tuning of an application and its resources to improve application functionality and performance. This control and tuning is interactive in that an external entity interacts with the application to accomplish it. That outside entity may be a user sitting at a workstation, or it may be another program responding to application events and driven by a previously-encoded steering algorithm.

We call steering *human-interactive* if a human watching a display is the primary initiator of a steering action. If instead the initiator is an outside program we call the steering *algorithmic*. Algorithmic steering may not be commonly associated with interactive programs, but it is a natural extension of the facilities and requirements presented earlier in this paper. For example, to expect a human watch an application and adjust it to compensate for load imbalances may be reasonable on an occasional basis, but no one is likely to babysit a 36-hour simulation that requires adjustments every five minutes. In this situation the solution is to feed the load information to an algorithm which is capable of balancing the load without human involvement. Using steering for this instead of embedding the load balancing algorithm into the application is still beneficial because it allows the algorithm to be expressed separately, where it is more easily understood, replaced and reused in other applications.

The different goals and types of steering exert different requirements on the steering system. For steering for performance, low overhead costs in monitoring and steering support are critical, simply because excessive overheads can easily offset performance gain obtained by on-line steering. Low steering latency may also be a critical requirement, particularly for algorithmic steering. Program events related to steering must be captured and processed, and the corresponding steering decision must be made while the decision is still relevant to the situation. Consider the on-line configuration of mutex locks presented in [MS93], where on-line algorithms change lock behavior from spin to blocking locks. Lock type is determined at runtime based on the time a lock call must wait before it obtains the lock. When the waiting time is above a certain threshold, the lock is a blocking lock. When the waiting time is relatively short, the lock is a spin lock. Since the reaction times required are on the order of a few tens of processor cycles, this application presents a formidable challenge for a steering system.

In the case of human-interactive steering, the demands on the steering system are not so extreme, as human response times will typically dwarf the latency times imposed by the system. However, if human interaction is to develop basic insight or to experiment with alternative solution methods and experimental parameters, more cooperation from the application may be necessarily. To accomplish the parameter tuning described in Section 3.1.1 for example, it is necessary to synchronize the parameter modifications with the phases of the application to ensure that steering does not invalidate the computations. In some cases the design of the application makes this easy. The load balancing of the MD application described in [EGSM94] was facilitated because mechanisms were in place to handle molecules moving from domain to domain. These worked without modification when the domain boundaries themselves moved. In other cases it is clear that desired manipulations cannot be carried out without the direct cooperation of the application. A good example of this is the checkpointing and rollback facility discussed in Section 3.1.1. It is unlikely that such functionality could be provided without the knowledge of the application. A continuing challenge in steering is to define the application interface to the steering system.

3.3.2 Falcon's Steering System

Falcon's on-line steering component is a natural extension of its monitoring facilities. Figure 6 depicts some internal features of steering as well as its relationship with other components of Falcon. Similar to local and central monitors, a *steering server* on the target machine performs steering, and a *steering client* provides the user interface and control facilities remotely. The steering server is typically created as a separate execution thread to which local monitors forward only those monitoring events that are of interest to steering activities. Such events tend to represent a small proportion of the total number of monitoring events, in part because simple event analysis and filtering is done by local monitors rather than by the steering server. Steering decisions are then made based on specific attributes of those events by human users or steering algorithms. Therefore, the primary task of each steering server is to read incoming monitoring events and to take the appropriate action in response. These responses are based on previously encoded decision routines and actions, which are encoded in an steering event/action repository in the server. This repository contains entries for each type of steering event, specifying the appropriate action to take in response. The responses represented here may perform some actual steering action on the application, note the occurrence of some monitoring event for future reference, or simply forward the event to the client for

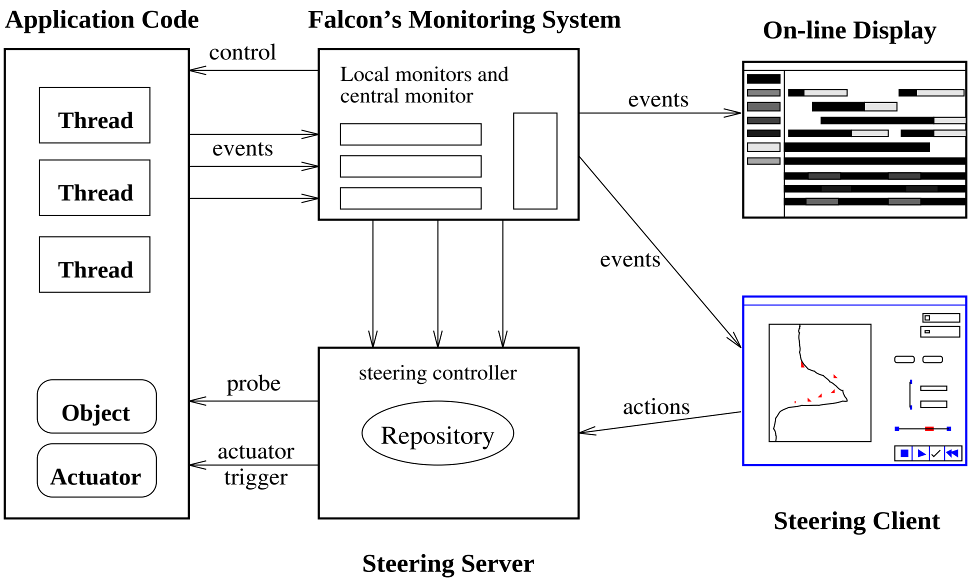

Figure 6: Overall structure of the steering system.

display or further processing. The secondary task of each steering server is to interact with the remote steering client. The steering client is used to enable/disable particular steering actions, display and update the contents of the steering event repository, and input steering commands directly from end users to the server.

Falcon's steering library introduces several abstractions. The first of these is *program attributes*. Program attributes are defined by application developers and they represent values or characteristics in the application that can be modified by the steering system. They are defined in an object-based fashion, where developers may associate with each specific program abstraction one or multiple attributes and then export methods for operating on these attributes. This type of association is called a *steerable object* and it must be "registered" with the steering system. The steering repository in the steering server maintains a list of all registered steerable objects and their associated program attributes. In our initial implementation, we assume that all attributes correspond to specific variables in the application program.

Steering actions are composite operations to be performed by the steering system in response to monitoring events in the program. Steering actions may examine and modify program attributes, perform computation, and even initiate other actions. Falcon defines two mechanisms for modifying program attributes, *steering probes* and *actuators*. A steering probe is the simplest form of steering action. It is used in actions to query or update a specific program attribute asynchronously to the program's execution. However, if a program attribute can only be updated synchronously, it must be associated with an actuator.

An actuator is a portion of code that the developer inserts into his code at locations in which it is "safe" for the steering system to take some action concerning the program attribute. Most of the time when the application executes the actuator code there are no pending actions and the actuator immediately returns control to the application. However, if the program attribute is to be synchronously modified by the steering system, the actuator becomes the instrument of that action. In particular, to update the program attribute synchronously, the steering system *asynchronously* sets the actuator so that the next time it is executed by the application code, it invokes a particular action *in the context of the application's thread of execution*. In this way, the responsibility for managing the synchronization of the steering system with the application rests with the application programmer and depends solely upon the placement of the actuators in the code. In simple situations, the action programmed into the actuator may just write a new value into the program attribute. For example, in the implementation of the steering of the atmospheric model as described in Section 3.1.1, the program variables corresponding to "Vertical Velocity" and "Vertical Eddy

Diffusion Coefficient" would be identified by the application programmer as program attributes and be registered with the steering system. The programmer would place actuators at points in his code, perhaps between iterations, where those values could be changed without invalidating the calculations. When a human triggered a change at the user interface, an actuator action would be "programmed" or "armed" with an action which would write the changed value into the target program variable at the next opportunity. However, actuator actions are capable of encoding much more complex operations than this. For example, they should be capable of the operations necessary to ensure that modifications of program state do not violate program correctness criteria as in [BS91].

The discussion above presents a brief overview of the abstractions in the Falcon steering library and the manner in which they interact with the remainder of the Falcon system. The implementation and integration of the steering library and other steering facilities is not yet complete, though proof-of-concept demonstrations as in [EGSM94] have been quite successful. However, we believe that the steering system, together with Falcon's monitoring facilities represent a powerful and flexible basis upon which to build interactive computing and through which users can exploit the opportunities presented in this paper.

4 Conclusions and Future Research

We have discussed the utility and potential of interactive parallel programming in the context of two large-scale parallel application programs. We have also explained how an on-line program steering and monitoring system can assist in realizing this potential. At present, ambitious and determined applications programmers can create their own interactivity by building user interfaces for their applications. These are valid interactive programs, but they are point solutions. Scientists are interested in computing to the extent that it helps them do science. Accordingly, the goal of our work on Falcon and monitoring and steering in general is to make this functionality more easily available to non-expert users.

The MD and atmospheric modeling codes as well as the Falcon system are implemented on a 64-node KSR shared memory supercomputer. Falcon is also available on several other shared memory platforms, including SGI and SUN Sparc parallel workstations. A version of Falcon currently being completed also works with PVM across networked execution platforms. Similar portability is attained for the graphical displays used with Falcon. Notably, the Polka animation library can be executed on any Unix platform on which Motif is available [SK93]. Falcon's low-level monitoring mechanisms have been available via the Internet since early Summer 1994. A version of Falcon offering on-line user interfaces for monitoring and monitor control will be released in 1995.

Current extensions of Falcon not only address additional platforms (e.g., an IBM SP machine now available at Georgia Tech and the monitoring of PVM programs running Cthreads, C, or Fortran programs), but also address several essential additions to its functionality. Currently users can insert into their code simple tracing or sampling sensors, where sensor outputs are forwarded to and then analyzed by the local and central monitors. We are now generalizing the notion of sensors to permit programmers to specify higher level 'views' of monitoring data like those described in [KS91, OSS93, Sno88]. Such views will be implemented with library support resident in both local and central monitors. We are also developing notions of composite and extended sensors that can perform moderate amounts of data filtering and combining before tracing or sampling information is actually forwarded to local and central monitors. Such filtering is particularly important in networked environments, where strong constraints exist on the available bandwidths and latencies connecting application programs to local and central monitors.

Our future work will address how such customized mechanisms may be used in conjunction with the remainder of the Falcon system. In addition, work in progress is addressing the monitoring of object-oriented, parallel programs, including the provision of default monitoring views and performance displays[MSSG95].

An important component of our future research is the use of Falcon with very large-scale parallel programs, either using thousands of execution threads or exhibiting high rates of monitoring traffic. For these applications it will be imperative that monitoring mechanisms are dynamically controllable and configurable. It must be possible for users to focus their monitoring on specific program components, to alter such monitoring dynamically, and to process monitoring data with dynamically enabled filtering or analysis algorithms. Moreover, such changes must be performed so that monitoring overheads are experienced primarily by the program components being inspected. Dynamic control of monitoring is also important for the efficient on-line steering of parallel programs of even moderate size. Specifically, program steering requires that monitoring overheads are controlled continuously so that end users or algorithms can perform steering actions in a timely fashion.

Longer term research with Falcon will address the integration of higher level support for program steering, including graphical steering interfaces, and the embedding of Falcon's functionality into a programming environment supporting the process of developing, tuning, and steering threads-based parallel programs, called LOOM. In addition, Falcon will be a basis for the development of distributed laboratories in which scientists can inspect, control, and interact on-line with virtual or physical instruments (typically represented by programs) spread across physically distributed machines. The specific example being constructed by our group is a laboratory for atmospheric modeling research, where multiple models use input data received from satellites, share and correlate their outputs, and generate inputs to on-line visualizations. Moreover, model outputs (e.g., data visualizations), on-line performance information, and model execution control may be performed by multiple scientists collaborating across physically distributed machines.

References

[Bec94] Thomas Becker. Application-transparent fault tolerance in distributed systems. In *Proc. of the Second International Workshop in Configurable Distributed Systems*. IEEE Computer Society Press, May 1994.

[BS91] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[BS93] Adam Beguelin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, Pittsburgh, PA, June 1993.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical report, Computer Science, Carnegie-Mellon University, CMU-CS-88-154, June 1988.

[EGSM94] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3)*, pages 26–34, San Francisco, CA, August 1994.

[FW94] I.T. Foster and P.H. Worley. Parallel algorithms for the spectral transform method. Technical Report ORNL/TM-12507, Oak Ridge National Laboratory, April 1994.

[GEK+94] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94-21, Georgia Institute of Technology, Atlanta, GA 30332-0280, April 1994.

[GS89] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106–125, July 1989.

[GS93] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[GVS94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29(9):140–148, September 1994.

[Hau40] B. Haurwitz. The motion of atmospheric disturbances on the spherical earth. *Journal of Mar. Res.*, 3:254–267, 1940.

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, pages 841–850, Knoxville, TN, May 1994.

[KHYK61] S. Kubota, M. Hirose, Y.Kichuchi, and Y. Kurihara. Barotropic forecasting with the use of surface spherical harmonic representation. *Pap. Meteorol. Geophys.*, 12:199–215, 1961.

[KS91] Carol E. Kilpatrick and Karsten Schwan. ChaosMON – application-specific monitoring and display of performance information for parallel and distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 57–67, Santa Cruz, California, May 20-21 1991.

[KSS+94] T. Kindler, K. Schwan, D. Silva, M. Trauner, and F. Alyea. A parallel spectral model for atmospheric transport processes. Technical report, Georgia Institute of Technology, Atlanta, 30332 GA, 1994.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–201. SIGOPS, Assoc. Comput. Mach., December 1989.

[MRW92] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[MS93] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable microkernel. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 45–60, September 1993.

[MSSG95] Bodhisattwa Mukherjee, Dilma Silva, Karsten Schwan, and Ahmed Gheith. Ktk: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*, 1995. To Appear.

[Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[MW91] Keith Marzullo and Mark Wood. Making real-time reactive systems reliable. *ACM Operating Systems Review*, 25(1):45–48, January 1991.

[OSS93] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[SG92] Sekhar R. Sarukkai and Dennis Gannon. Parallel program visualization using SIEVE.1. In *International Conference on Supercomputing*. ACM, July 1992.

[SGB87] Karsten Schwan, Prabha Gopinath, and Win Bo. CHAOS – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[Sil54] I.S. Silberman. Planetary waves in the atmosphere. *J. Meteorol.*, 11:27–34, 1954.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[Sno82] Richard Snodgrass. *Monitoring Distributed Systems: A Relational Approach*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, December 1982.

[Sno87] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[SO93] R. Swinbank and A. O'Neill. A stratosphere - troposphere data assimilation system. Climate Research Technical Note CRTN 35, Hadley Centre Meteorological Office, London Road Bracknell Berkshire RG12 2SY, March 1993.

[WP86] W.M. Washington and C.L. Parkinson. *An introduction to three-dimensional climate modeling*. Oxford University Press, 1986.

[XORL92] T. K. Xia, Jian Ouyang, M. W. Ribarsky, and Uzi Landman. Interfacial alkane films. *Physical Review Letters*, 69(13):1967–1970, 28 September 1992.