Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs

Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Jeffrey Vetter

Presented by Amy Sliva

What is interactive steering?

- On-line configuration of a program by algorithms or by human users with the purpose of affecting the program's performance or execution behavior
- Targets parallel code itself
  - Rapid changes made by on-line algorithms
  - Implementation of single program abstractions
  - User-directed improvement of high-level attributes
- Useful for understanding and improving program behavior

So, what is Falcon?

- System for on-line monitoring and steering of threads-based parallel programs
- Contributions:
  - Application-specific monitoring
  - Scalable, dynamically controlled monitoring performance
  - On-line analysis, steering, and graphical display
  - Extension to multiple heterogeneous computing platforms

Requirements of Steering

- Application builders must write their code so steering is possible
- Users must provide the program and performance information necessary for making steering decisions
- This information must be obtainable with the latency required by the desired rate of steering

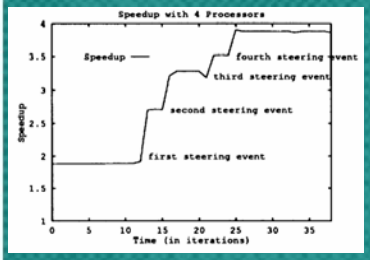
MD Application

- Interactive molecular dynamics simulation
- Simulation Process:
  - Obtain location information from neighbors
  - Calculate intra-molecular forces
  - Compute inter-molecular forces
  - Apply forces to yield new particle position
- Parallelism by domain decomposition

Steering MD-Experimentation

- Opportunities for performance improvement through on-line interactions:
  - Decomposition geometries changed to respond to changes in physical systems
  - Modification of cutoff radius to improve solution speed
  - Dynamic load balancing
  - Local rather than global temperature calculations
- Falcon used to monitor process loads on-line
  - Analyze and display workload information

Steering MD-Results



- Steering improves performance by successive adjustment of domain boundaries

Importance of Results

- Performance improves due to steering, rather than degrading because of steering costs
- User interactions can be replaced by on-line steering algorithms
- Indicate the potential of on-line steering for helping users experiment with and understand the behavior of complex scientific codes

Design Goals

- Reduce or control monitoring latency while maintaining acceptable monitoring workload
- Supports application-specific monitoring/steering, analysis, and display of program information
  - Users only capture, process, understand, and steer exactly the relevant program attributes
- Support for scalable monitoring
  - Vary resources consumed by runtime system in accordance with machine size and program needs

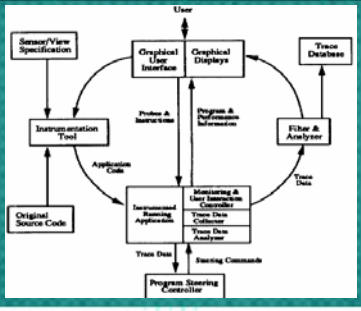
System Design

- Four conceptual components:
  - Monitoring specificaion and instrumentation
    - Low-level sensor specification language
    - High-level view specification constructs
    - Instrumentation tool
  - Runtime libraries for information capture, collection, filtering, and analysis
  - Mechanisms for program steering
  - Graphical user interface and displays

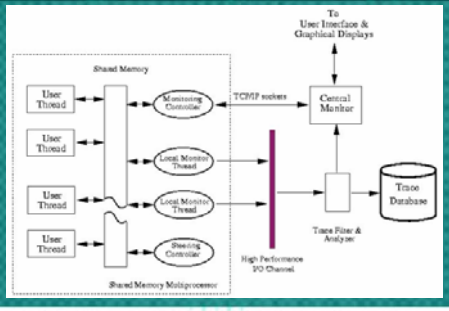
Steps to use Falcon

- Instrument application code with sensors and probes
- Inserted sensors capture information during runtime
- Runtime monitoring facilities collect and analyze data
- Partially processed monitoring data sent to steering mechanisms or central monitor and graphical displays

Falcon Architecture



Falcon Implementation



System Implementation

- Implemented with Cthreads library
- Hardware platforms
  - Kendall Square Research KSR-1 and KSR-2 supercomputer
  - GP1000 BBN Butterfly multiprocessor
  - Sequent multiprocessor
  - SGI workstations
  - SUN SPARC stations

System Implementation (cont.)

- User-defined application-specific sensors capture:
  - Program and performance behaviors to be monitored
  - Program attributes based on which steering may be performed
- Sensor code can be inserted into many different places in a program
- Local monitoring threads

Sensor Types

- Sampling sensors
  - Write to shared memory
  - Less overhead
  - Less detailed and accurate information
- Tracing sensors
  - Generate timestamped event records
- Extended sensors
  - Performs simple analyses and filtering before output

Sensor Control

- Combine sensor types to balance low monitoring latency against accuracy
- Dynamically turn sensors' instrumentation points on and off
- Sensors dynamically adjust their own behavior
- Employ different monitoring mechanisms at different points in time

Concurrent Monitoring

- Local monitoring and steering threads act concurrently and asynchronously with the target application
- Can change the number of local monitoring threads and communication buffers
  - Adapt configuration to dynamic workload changes

On-line Steering Mechanisms

- Steering server
  - Separate execution thread
  - Receives monitoring events relevant to steering
  - 'Decide' what actions to take using steering event database
- Steering client
  - User interface and control
  - Enable/Disable steering actions
  - Display/Update database

On-line Steering Mechanisms

- Probes
  - Update program attributes asynchronously to the program execution
- Actuators
  - Used to enact steering actions
  - Actions operate on program attributes
  - Execute functions to ensure that modifications do not violate correctness criteria

Experimental Setup

- Kendall Square Research KSR-2
- 64 processors interconnected by two rings
- Non-uniform shared memory cache-only architecture
  - Hierarchically connected rings
  - Support 32 nodes each
    - 64-bit processor
    - 32 Mbytes main memory used as local cache
    - 0.5 Mbyte sub-cache
    - Ring interface

Sensor Performance

- Cost of executing each sensor
  - Accessing a sensor switch flag
  - Computing the value of sensor attributes
  - Writing the record to a monitoring buffer
- Perturbation, latency, and throughput factors:
  - Size of the event data structure
  - Cost of event transmission and buffering from sensors to local monitors
  - Sensor type

Sensor Performance (cont.)

Event size	32 bytes	64 bytes	128 bytes
Cost	9.4	10.6	12.2

- Direct program perturbation from sensors acceptable for many applications for moderate rates of monitoring
- Dominant factor is cost of accessing the buffer shared between the application and monitoring threads
  - Falcon uses multiple monitoring buffers
- Measured costs do not include bottlenecks in event processing and transmission

Monitoring Latency

- Monitoring latency
  - Elapsed time between the time of sensor record generation and the time of sensor record receipt and (minimal) processing by a monitoring thread
- Low latency means steering algorithms can rapidly react to changes in program state

Minimum Monitoring Latency

Buffer size (bytes)	Record length		
	32 bytes	64 bytes	128 bytes
256	69	73	87
1,024	68	71	84
4,096	68	70	83
16,384	69	73	85

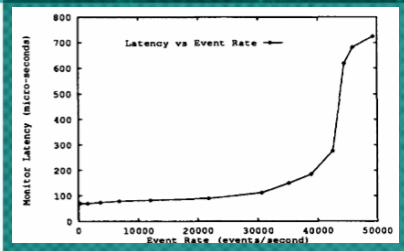
- Monitoring latency evaluated under low loads
- Approximate lower bound on latency
- Monitoring latency independent from the size of the monitoring buffer

Latency at Moderate Rates

Buffer size (bytes)	Record length		
	32 bytes	64 bytes	128 bytes
256 bytes	164	181	242
1,024	201	264	294
4,096	211	277	498
16,384	256	347	556

- Larger monitoring buffers reduce perturbation, but increase latency
- Perturbation can be larger with small buffers because of wait time for buffer space

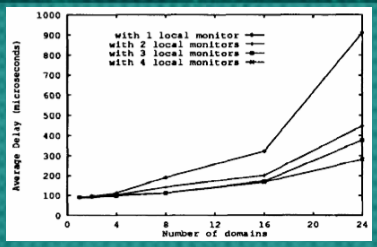
Monitoring Latency (cont.)



- Saturation: 40,000-45,000 events/s

- Latency under control when rate below saturation point

Monitoring Latency (cont.)



- Bottlenecks can be remedied by use of parallelism (multiple local monitors)

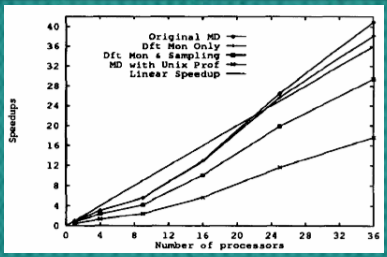
Monitoring Latency-Conclusions

- No general means of attaining low latency and perturbation at arbitrary monitoring rates
- Falcon permits configuration of the monitoring system itself
- Configuration can be dynamic

Monitoring MD

- Four sets of MD runs
- Performance and perturbation from Falcon compared in five cases:
  - No monitoring is performed
  - Only trace Cthreads calls
  - Trace Cthreads events and sample 10 most frequent MD procedure calls
  - Use Unix Prof profiler on KSR-2
  - Trace Cthreads and 10 most frequent MD procedure calls

MD Monitoring Performance



- MD speedup degradation with different amounts of monitoring

MD Monitoring Results

- Dft Mon Only
  - Default monitoring does not perturb execution
  - Monitoring overheads increase with number of processors
- Dft Mon and Sampling & MD with Unix Prof
  - Procedures monitored by sampling sensors
  - Compare Falcon overheads and Unix Prof
  - Falcon overhead much lower (35% v. 130%)
- Tracing all Mon Events
  - Tracing sensors used in place of sampling sensors
  - Tracing sensors too expensive for procedure profiling

Execution time and Monitoring Overheads

Number of Processors	Execution Time of Each Iteration (seconds) & Monitoring Overhead				
	Original MD	Dft Mon Only	Dft Mon & Sampling	Tracing All Mon Events	MD with Unix Prof
1	3.92	3.94(< 1%)	4.95(26%)	76.4(1850%)	9.07(132%)
4	1.28	1.28(< 1%)	1.68(30%)	47.5(3610%)	2.95(131%)
9	0.704	0.707(< 1%)	0.936(33%)	34.9(4860%)	1.61(129%)
16	0.302	0.306(1%)	0.388(29%)	18.6(6080%)	0.69(130%)
25	0.148	0.152(3%)	0.197(34%)	13.8(9270%)	0.337(128%)
36	0.096	0.103(7%)	0.133(39%)	8.96(9280%)	0.222(132%)

- Introduction
- Monitoring and Steering a Parallel Code
- Design and Implementation of Falcon
- System Evaluation
- Conclusions

Conclusions

- Falcon's selective monitoring with multiple monitoring mechanisms allow scalability and overhead control
- High performance of Falcon's run-time monitoring enable efficient on-line, interactive steering
- Future work
  - Improvements in performance
  - Further study of scalability
  - Integration of Falcon and LOOM