In Search of Sweet-Spots in Parallel Performance Monitoring

Aroon Nataraj, Allen D. Malony, Alan Morris
University of Oregon
{anataraj,malony,amorris}@cs.uoregon.edu

Dorian C. Arnold, Barton P. Miller
University of Wisconsin, Madison
dorian.arnold@gmail.com
bart@cs.wisc.edu
Motivation

- Performance problem analysis increasingly complex
  - Multi-core, heterogeneous, and extreme scale computing

- Shift of performance measurement and analysis perspective
  - Static, offline analysis $\Rightarrow$ dynamic, online analysis
  - Support for performance monitoring (measurement + query)
  - Enabling of adaptive applications with performance feedback

- Prerequisites for performance measurement
  - Low overhead and low intrusion
  - Runtime analysis antithetical to performance tool orthodoxy

- Neo-performance perspective
  - Co-allocation of additional (tool specific) system resources
  - Make dynamic, performance-driven optimization viable
Monitoring for Performance Dynamics

- Runtime access to parallel performance data
  - Scalable and lightweight
  - Support for performance-adaptive, dynamic applications
- Raises vital concerns of overhead and intrusion
  - Bigger issue in online systems due to global effects
- **Alternative 1**: Extend existing performance measurement
  - Create own integrated monitoring infrastructure
  - Disadvantage: maintain own monitoring framework
- **Alternative 2**: Couple with other monitoring infrastructure
  - Leverage scalable middleware from other supported projects
  - Challenge: measurement system / monitor integration
  - TAU over Supemon (ToS) (UO, LANL)
  - TAU over MRNet (ToM) (UO, University of Wisconsin)
Performance Monitoring: Contradictory Goals

- Application semantics dictate monitoring scheme
  - Performance across iterations / phases

- Requirements determining a performance monitoring scheme
  - Performance events to monitor
  - Access frequency
  - Data analysis operation (reduction type)

- Performance monitoring costs
  - Intrusion to application
  - Extra monitoring resource allocation (# processors)

- Opposing goals (leads to trade-offs)
  - Acceptable performance data granularity (temporal / spatial)
  - Acceptable level of application intrusion from offloads
  - Acceptable monitoring resource requirements
Scalable Low-Overhead Performance Monitoring

- Key is to match ...
  - Application monitoring demands with ...
  - Effective operating range of monitoring infrastructure
- Over-provisioning (more monitor resources assigned)
  - Leads to wasted resources and lost performance potential
- Under-provisioning (less monitor resources assigned)
  - Poor scalability, high overhead, low performance data quality
- Not simply a question of # processors - but configuration
  - Transport topology
  - Transport-level reduction operations
- Try to find the monitoring *Sweet-Spot*
  - "Area on a bat or racket where it makes most effective contact with the ball" (New Oxford English Dictionary)
Talk Outline

- Motivation
- Monitoring for performance dynamics
- Contradictory goals of performance monitoring
- Key to scalable low-overhead performance monitoring
- TAUoverMRNet (ToM)
- Naive monitoring choices and consequences
- Estimating the bottleneck
- Characterization and finding sweet-spots
- Future plans and conclusion
What is MRNet?

- Multicast Reduction Network (University of Wisconsin)
  - Software infrastructure, API, utilities (written in C++)
  - Create and manage network overlay trees (TBON model)
  - Efficient control through root-to-leaf multicast path
  - Reductions (transformations) on leaf-to-root data path
  - Packed binary data representation

- Uses thread-per-connection model
  - Supports multiple concurrent “streams”

- Filters on intermediate nodes (upstream, downstream)
  - Default filters (e.g., sum, average)
  - Loads custom filters through shared-object interface

- MRNet-base tools (Paradyn, STAT debugger, ToM)
TAUoverMRNet (ToM)

- Back-End (BE) TAU adapter offloads performance data
- Filters
  - reduction
  - distributed analysis
  - upstream / downstream
- Front-End (FE) unpacks, interprets, stores
- Paths
  - reverse data reduction path
  - multicast control path
- Push-Pull model
  - source pushes, sink pulls
ToM Filters

- Ideally there would be no need for filtering
  - Retrieve and store *all* performance data provided
  - Acceptability depends on performance monitor use

- High application intrusion, transport and storage costs
  - Need to trade-off queried performance data granularity
  - Which events, time intervals, application ranks?

- Reduce performance data as it flows through transport
  - Distribute FE analysis out to intermediate filters

- Three filtering schemes
  - 1-phase: summary
  - 3-phase: histogram, classification histogram
  - Progressive temporal/spatial detail with added complexity
**Histogram Filter (FLASH)**

**FLASH Sod 2D | N=1024 | Allreduce**

- 1024 MPI Ranks
- FLASH 2D Sod
- ToM Fanout=8
- Offload performance every iteration

N profiles of 300 events
- 300K events total

FLASH: astrophysical thermonuclear flashes
**Histogram Filter (FLASH)**

FLASH Sod 2D | N=1024 | Allreduce

- 1024 MPI Ranks
- FLASH 2D Sod
- ToM Fanout=8
- Offload performance every iteration

Temporal information
Spike at Iteration 100

N profiles of 300 events
- 300K events total

FLASH: astrophysical thermonuclear flashes

---

**CLUSTER 2008, Tsukuba, Japan**

*Sweet-Spots in Parallel Performance Monitoring*
**Histogram Filter (FLASH)**

**FLASH Sod 2D | N=1024 | Allreduce**

- **Temporal information**
  - Spike at Iteration 100

- **Spatial information**
  - Unevenness of across ranks
  - Evolution of unevenness over iterations

- **Hist Filter**
  - 1024 MPI Ranks
  - FLASH 2D Sod
  - ToM Fanout=8
  - Offload performance every iteration

- **N profiles of 300 events**
  - 300K events total

**Application Iteration #**

**FLASH: astrophysical thermonuclear flashes**
Evaluating Monitoring Choices

- Simple SPMD workload outputs profile to ToM
  ```c
  for (i=0; i<iterations; i++) {
    work (usecs);   TAU_DB_DUMP();   MPI_Barrier();
  }
  ```
  - Number of profile events fixed (64)
  - Monitoring offload interval (usecs) = 100ms and 6ms
  - # of application ranks = 64, # iterations = 1000

- Simple (1-phase) statistics filter
  - ToM Fanout (FO) = 8 (two-level tree)

- Offload Cost (OC) metric
  - Maximum time within offload operation across ranks

- One Way Delay (OWD) metric
  - Time difference between sink receive and earliest BE send
Naive Monitoring : Consequence

---

**Figure 1:**

Varying Offload Interval: Offload Cost (ms)

- Offload Interval: 6ms
- Offload Interval: 100ms

The impact to the application follows directly from the blocked send operations. The actual offload can be performed in a separate consumer thread with the main application thread placing the profiles into an unbounded buffer. This decouples the application from the actual offload and provides latency hiding.

---

**Figure 2:**

Plots the results from such a non-blocking scheme. The OC plot of both configurations is relatively small and stable since the actual offload is occurring in a worker thread. But the OWD of the 6ms case shows an early large growth followed by a continuous steady growth unlike the plateau in A4. The final OWD is in the range of 6,000 ms, an order of magnitude larger than the blocking.
Naive Monitoring: Consequence

100 ms interval
OC stable (2 to 4 ms)
OWD stable (60 ms)
Naive Monitoring: Consequence

Offload Iteration #

Offload Interval: 6ms
Offload Interval: 100ms

Offload Cost (ms)

OWD (ms)

Offload Iteration #

One Way Delay (ms)

Offload Interval: 6ms
Offload Interval: 100ms

6 ms interval: OWD
Quickly grows and plateaus
Periodic, growth / sudden drops

Figure 3B4 plots the results from such a non-blocking scheme. The OC of both configurations is relatively small and stable since the actual offload is occurring in a worker thread. But the OWD of the 6ms case shows an early large growth followed by a continuous steady growth. Unlike the plateau in 3A44. The final OWD is in the range of 6,,,,ms an order of magnitude larger than the blocking.
Naive Monitoring: Consequence

6 ms interval: OC
Sudden spikes corresponding to sudden drops in OWD

6 ms interval: OWD
Quickly grows and plateaus
Periodic, growth / sudden drops

Offload Iteration #
Naive Monitoring : Consequence

Figure (Varying Offload Interval: Offload Cost)

The impact to the application follows directly from the blocked send operations. Figure 3 plots the mean OC on the left side for the three offload intervals as the curve labeled OC Blocking. These OC values represent overheads ranging from 3.1 ms to 779 ms at 6 ms of application runtime. While consistent global snapshots are being delivered, the choice of 6 ms intervals leads to an excessive overhead level. Given that the overheads are attributable to the large blocking times, would a non-blocking offload solution help? The actual offload can be performed in a separate consumer thread with the main application thread placing the profiles into an unbounded buffer. This decouples the application from the actual offload and provides latency hiding. Figure (3B4) plots the results from such a non-blocking scheme. The OC (top plot) of both configurations is relatively small and stable since the actual offload is occurring in a worker thread. But the OWD of the 6 ms case shows an early large growth followed by a continuous steady growth (unlike the plateau in 3A44). The final OWD is in the range of 6 ms, an order of magnitude larger than the blocking.
Naive Monitoring: Consequence

Behavior of 6 ms offload-interval
- Transport: Tree of TCP connections
- 6 ms offload interval > Service capacity
- Queuing $\Rightarrow$ Large OWD values
- Buffer overflows $\Rightarrow$ TCP flow-control
- Back-pressure propagates to Back-End
- send() call blocks $\Rightarrow$ Large OC spike
- Temporary rate reduction $\Rightarrow$ OWD drop
- Pattern repeats
Non-Blocking Monitoring Scheme

- Impact to application directly from blocking `send()`
  - Large OC spikes
  - 779.4% overhead due to large blocking time
- Will simple non-blocking approach help?
- Decouple application from offloading
  - Separate consumer thread performs actual offload
  - Application places profile into shared, unbounded buffer
  - Latency hiding
- Repeat same experiments, with non-blocking scheme
Non-Blocking Monitoring: Consequence

Offload Cost (ms)

Offload Iteration #

Offload Interval: 6ms

Offload Interval: 100ms

One Way Delay (ms)

Offload Iteration #

Offload Interval: 6ms

Offload Interval: 100ms

The impact to the application follows directly from the blocked send operations. Figure plots the mean OC on the left axis for the three offload intervals as the curve labeled OC Blocking. These OC values represent overheads ranging from ~3 at 6ms to ~79 at 6ms of application runtime. While consistent global snapshots are being delivered, the choice of 6ms intervals leads to an excessive overhead level. Given that the overheads are attributable to the large blocking times, would a non-blocking offload solution help? The actual offload can be performed in a separate consumer thread with the main application thread placing the profiles into an unbounded buffer. This decouples the application from the actual offload and provides latency hiding. Figure (3B4) plots the results from such a non-blocking scheme. The OC plots of both configurations is relatively small and stable since the actual offload is occurring in a worker thread. But the OWD of the 6ms case shows an early large growth followed by a continuous steady growth unlike the plateau in A4. The final OWD is in the range of 6ms, an order of magnitude larger than the blocking...
Non-Blocking Monitoring: Consequence

Separate thread offloads OC small for both
100 ms case: OWD stable
Non-Blocking Monitoring: Consequence

Offload Cost (ms)

One Way Delay (ms)

Offload Iteration #

Offload Interval: 6ms
Offload Interval: 100ms

6 ms case: $OWD$
Continuous steady growth
$OWD=60,000$ ms $>>$ blocking-$OWD$
Non-Blocking Monitoring: Consequence

Figure (A) plots the mean One Way Delay (OWD) for the three offload intervals as the curve labeled Blocking. These OWD values represent overheads ranging from 2.5 ms to 779.0 ms at 6 ms of application runtime. While consistent global snapshots are being delivered, the choice of 6 ms intervals leads to an excessively high overhead. Even with that the overheads are attributable to the large blocking times. Given that the overheads are due to the large blocking times, would a non-blocking offload solution help?

The actual offload can be performed in a separate consumer thread with the main application thread placing the profiles into an unbounded buffer. This decouples the application from the actual offload and provides latency hiding. Figure (B) plots the results from such a non-blocking scheme. The OWD plot of both configurations is relatively small and stable since the actual offload is occurring in a worker thread. But the OWD of the 6 ms case shows an early large growth followed by a continuous steady growth with no plateau in (A). The final OWD is in the range of 6,000 ms, an order of magnitude larger than the blocking.
Non-Blocking Monitoring: Consequence

Behavior of 6 ms offload-interval
- No blocking in producer
- No temporary rate reductions
- OWD increases without bound

Application overhead reduced (small OC)

Problem
- Performance data remains queued
- At end of run, only few profiles available
- Wait for remaining profiles increases overhead again (607%)
Non-Blocking and Lossy Monitoring Scheme

- Previous schemes do not reduce number of offloads
  - Non-blocking simply delays the problem

- Application can detect problems
  - *Blocking case*: Locally detect spike in *OC*
  - *Non-blocking case*: Locally detect full buffer

- Application can do something
  - *Locally back-off*
  - Drop the current profile instead of offloading

- Lossy, non-blocking scheme
  - With bounded buffers
  - With local back-off
Lossy, Non-Blocking : Loss Map

Offload Interval 6ms : Loss

Offload Interval 24ms : Loss

Offload Iteration #

CLUSTER 2008, Tsukuba, Japan  Sweet-Spots in Parallel Performance Monitoring
Lossy, Non-Blocking : Loss Map

- Offload Interval 6ms : Loss
- Offload Interval 24ms : Loss

*24ms Offload Interval*
- Every 8th rank is lossy
- ToM Fan-out = 8
Lossy, Non-Blocking : Loss Map

Offload Interval 6ms : Loss

Offload Interval 24ms : Loss

6ms Offload Interval
- Loss structure different

24ms Offload Interval
- Every 8th rank is lossy
- ToM Fan-out = 8

CLUSTER 2008, Tsukuba, Japan
Sweet-Spots in Parallel Performance Monitoring

Loss Occurred | Interval: 6ms
Loss Occurred | Interval: 24ms

Offload Iteration #
Lossy, Non-Blocking : Loss Map

Complete global offloads:
- 15% in 24ms
- 11.4% in 6ms
Complete profiles from only initial 200 iterations

Offload Interval 6ms : Loss

Offload Interval 24ms : Loss

Offload Iteration #

CLUSTER 2008, Tsukuba, Japan  Sweet-Spots in Parallel Performance Monitoring
Inconsistent, Incomplete Performance Data

- OC spike / full-buffer signal is inconsistent
  - Some ranks repeatedly penalized
  - Some ranks are never penalized
  - Bursts of loss - large intervals unmonitored
    - Local backoff reaction inconsistent
  - Late reaction - OC spike / full-buffer implies damage done

- Spatially and temporally inconsistent performance views
  - Which ranks are monitored when? No control.
  - Which intervals / iterations are monitored? No control.

- Need globally consistent performance views
- Need globally consistent method to determine ToM capacity
Queueuing Theory 101

BOI: bottleneck offload interval

A. Offload Interval $\geq$ BOI

B. Offload Interval $<$ BOI

Offload Interval: 6ms
Estimation of Bottleneck Interval

- Need an estimator of operating capacity
  - Minimum offload interval without queueing
  - Per profile size and filter type
  - Metric: *Bottleneck Offload Interval (BOI)*

**Estimation Method 1**
- If offload interval < \(BOI\), then departure interval \(\approx BOI\)
- Set offload interval = 0
- Measure departure interval

**Estimation Method 2**
- If offload interval < \(BOI\), then queue builds, \(OWD\) increases
- If offload interval \(\geq BOI\), then \(OWD\) stable or decreases
- Use increasing \(OWD\) as heuristic to *binary-search* for \(BOI\)
**BOI Binary Search: Example**

![Diagram showing the BOI Binary Search process](Diagram.png)

**OWD metric**

A new offload interval is chosen based on reaction of the *OWD* metric (above)

**Search Progress Step**

**One Way Delay (ms)**

**Offload Interval (ms)**

**Curr. OWD**

**Rest. OWD + threshold**

**Growth**

---

*CLUSTER 2008, Tsukuba, Japan*  
*Sweet-Spots in Parallel Performance Monitoring*
Bottleneck Offload Interval depends on many factors

- Underlying network latencies and bandwidth
- TCP/IP stack processing
- MRNet (de)packetization
- Intermediate custom filter and sink operations
- TAU wrapper processing

Cannot use standard capacity estimation tools (e.g., Nettimer)

- Probes will not encounter all costs involved in ToM
- Meant for one-to-one paths, not many-to-one ToM trees

Instead corroborate BOI estimates from Methods 1 and 2

- Test under various configurations
- Result: Estimates agree to within 10%
Once BOI is determined, how should it be used?

- ToM provides estimation APIs to query
  - BOI, OWD, OC and several other metrics
- Application can use metrics to decide profile granularity
- Example: Iterative application - 75 ms per iteration
  - Estimated BOI provided by ToM = 100 ms
  - All application ranks can decide to drop every 4th profile
  - Average offload interval (over 4 rounds) will match BOI
- How to bridge monitoring requirements and costs?
  - Given application size and type of reductions ...
  - How to choose: ToM fanout, offload intervals, # profile events
- Answer: Characterization
  - Need to characterize various ToM configurations using BOI
Characterizations using BOI

- Configurations
  - Simple statistics filter:
    - mean, min, max, standard deviation
  - Profile sizes
    - 16 to 1024 events with power of two increments
  - Application size $N = 8$
    - ToM fanouts $FO = 2, 8$
  - Application size $N = 16$
    - ToM fanouts $FO = 2, 4, 16$
  - Application size $N = 64$
    - ToM fanouts $FO = 2, 4, 8, 64$
- Estimate $BOI$ for each configuration
  - Pick median of 3 trials for each point
Application Size $N = 8$; Fanouts $FO = 2, 8$
**Application Size N = 8; Fanouts FO = 2, 8**

- FO = 8 outperforms FO = 2!
  - Initial large difference
  - Difference shrinks at large #events
  - Not related to queueing costs (BOI)
  - Not related to networking costs

MRNet tree fits in single 2x4 core node
Reduction and (De)Packetization Costs

- Reduction Cost \( (TR) \) for single binary reduction operation
  - For \( N = 8 \), \( 7 \times TR \) cycles for reduction
  - Reduction performed on arrival of last profile
  - \( FO = 8 \): single thread performs all \( 7 \times TR \) cycles serially
  - \( FO = 2 \): \( 7 \times TR \) cycles split across 7 threads in tree

- (De)Packetization Cost \( (TP) \) to (un)pack intermediate profile
  - \( FO = 8 \): No intermediate (de)packetizations
  - \( FO = 2 \): Every-level in tree adds (de)packetizations

- At small #events, \( TP \) dominates costs in \( FO = 2 \)
- \( FO = 8 \) \( TR \) costs quickly rise as single processor saturated
- Intuitively: Too many resources allocated in \( FO = 2 \)
  - Serial costs and parallelization overheads dominate
Application Size $N = 16$; Fanouts $FO = 2, 4, 16$

FO = 16 may cross-over FO = 4 eventually

Similar Trends as N = 8 case
FO = 4 crosses-over FO = 2

FO = 16 crosses-over FO = 4 at 256 events

CLUSTER 2008, Tsukuba, Japan

Sweet-Spots in Parallel Performance Monitoring
Application Size $N = 64$; Fanouts $FO = 2, 4, 8, 64$

$FO = 2, 4, 8$ in “correct” order

Already crossed-over

$FO = 64$ makes the crossover
Monitoring Requirements and Infrastructure Costs

- BOI characterization shows importance of careful match
  - Under-provisioning may be bad for performance
  - But so can over-provisioning!
- Other metrics (direct costs, limiting overhead) also in paper
- Sweet-spot configurations for specific requirements exist
- ToM helps find these sweet-spots
  - APIs
  - Framework
  - Metrics and evaluation methodology
Conclusion and Future Work

- **Sweet-spot**
  - “Spot on a bat that produces the least shock when a ball is hit” (New Oxford American Dictionary)
- Parallel performance monitoring must meet overhead, latency, responsiveness, data consistency requirements
- **Sweet-spots** are those configuration choices that meet the requirements or allow acceptable trade-offs
- Methodology and framework help find *sweet-spots* and make informed monitoring decisions
- Would like to extend the characterizations to other filters
- Analysis of irregular, less periodic or non-uniform behaviors
- Dynamic estimation and feedback to application to stay within sweet-spot during execution
Credits

University of Oregon
- Aroon Nataraj
- Alan Morris
- Allen D. Malony
- TAU group members
- “Extreme Performance Scalable Operating Systems,” DOE FastOS-2 grant, with Argonne National Lab

University of Wisconsin
- Dorian C. Arnold (soon to be at University of New Mexico)
- Michael Brim
- Barton P. Miller