Exploring the Impact of Overlay Network Topology on Tool and Application Performance

Whit Schonbein
University of New Mexico
2016-03-30
Advisor: Dorian Arnold
HPC monitoring, debugging, and analysis tools often share a common communication structure: Trees.
Introduction

- HPC monitoring, debugging, and analysis tools often share a common communication structure: Trees.

- Tree-based overlay networks provide lightweight communication infrastructure for facilitating this type of communication.

- Tool performance depends on overlay network performance.
Introduction

- While tree structures are common to many software architectures...

- ... there are limited studies of how tools perform under different workloads and tree topologies.
Example: Latency-bound

- Task: Querying nodes for core temperature, filtering only those that exceed a threshold.
  - Simple (low-latency) aggregation filter.
  - Small packet size.
  - Low frequency.
Example: Bandwidth-bound

- Task: Collecting stack traces at milliseconds latency.
  - More computationally substantive filter (merge traces).
  - Potentially larger packet size.
  - High frequency data stream.
Introduction

- There are limited studies of how tools perform under different types of workloads and tree topologies.
  - Arnold et. al. (2006), Goehner et. al. (2013), Groves et. al. (2015)

- Goal: Start to fill in this gap in our knowledge.
Methodology

- Infrastructure: MRNet
  - In use in a variety of tools (TotalView, STAT, AutomaDeD)
  - Data aggregation capabilities via filters
Methodology

- Benchmarking: MRNetBench
  - User-defined topologies
    - Structure
    - Number of nodes
  - Packet size
  - Communication frequency
  - Latency incurred by filters
Modeling Filter Latency

- Original Method:
  - Calibration phase:
    - For each calibration trial, measure duration of matrix multiplication, resting between each trial.
    - Get average duration across trials.
  - Usage phase:
    - Calculate number of iterations of matrix multiplication required given the average duration and target latency.
Modeling Filter Latency

Original Method - Target vs. Actual - 300μs-2 seconds

- target
- 10 trials
- 100 trials
- 1000 trials
Modeling Filter Latency

<table>
<thead>
<tr>
<th></th>
<th>10 trials</th>
<th>100 trials</th>
<th>1000 trials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>$2.076 \times 10^{-1}$</td>
<td>$2.073 \times 10^{-1}$</td>
<td>$2.229 \times 10^{-1}$</td>
</tr>
<tr>
<td>Variance</td>
<td>$6.200 \times 10^{-8}$</td>
<td>$1.168 \times 10^{-7}$</td>
<td>$3.778 \times 10^{-7}$</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>$2.490 \times 10^{-4}$</td>
<td>$3.418 \times 10^{-4}$</td>
<td>$6.146 \times 10^{-4}$</td>
</tr>
</tbody>
</table>
Modeling Filter Latency: (1) New Method

- New Method
  - Calibration phase:
    - For each calibration trial, measure duration of operation performed for n cycles, resting between each trial.
    - Get average duration across trials.
  - Usage phase:
    - Calculate number of cycles required given average duration and target latency.
Modeling Filter Latency: (1) New Method

clock_gettime( CLOCK_MONOTONIC, &start )

uint64_t _sc, _tc, _cc, _sh, _sl, _ch, _cl;
float _bf = 21.1198213341;
float _o;
__asm__ __volatile__ ( "CPUID ;"
"RDTSC ;"
"mov %%rdx, %0 ;"
"mov %%rax, %1 ;"
: "=r" (_sh),
 "=r" (_sl) : :
 "%rax",
 "%rbx",
 "%rcx",
 "%rdx" );
_sc = ( ((uint64_t)_sh << 32) | _sl );
_tc = _sc + loop_num;
do {
__asm__ __volatile__ ( "flds %3;"
"fmulp;"
"CPUID;"
"RDTSC;"
"mov %%rdx, %1;"
"mov %%rax, %2;"
: "=&t" (_o),
 "=r" (_ch),
 "=r" (_cl) :
 "m" (_bf),
 "0" (5.35667) :
 "st(1)",
 "%eax",
 "%ebx",
 "%rcx",
 "%rdx" );
_cc = ( ((uint64_t)_ch << 32) | _cl );
} while (_cc <= _tc);
clock_gettime( CLOCK_MONOTONIC, &end ) == -1 ) {
rest()
Modeling Filter Latency: (2) Avoid Sleeping

- Original method uses sleep(1) to rest
  - Solution: Change to a ‘nosleep’ resting method.
# Modeling Filter Latency: Results

![Graph showing average relative error vs requested µs](image)

<table>
<thead>
<tr>
<th>Requested µs</th>
<th>Mean</th>
<th>Variance</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>original s</td>
<td>$2.258 \times 10^{-1}$</td>
<td>$4.159 \times 10^{-5}$</td>
<td>$6.449 \times 10^{-3}$</td>
</tr>
<tr>
<td>original ns</td>
<td>$3.430 \times 10^{-3}$</td>
<td>$2.375 \times 10^{-5}$</td>
<td>$4.873 \times 10^{-3}$</td>
</tr>
<tr>
<td>new s</td>
<td>$1.564 \times 10^{-5}$</td>
<td>$9.150 \times 10^{-10}$</td>
<td>$3.025 \times 10^{-5}$</td>
</tr>
<tr>
<td>new ns</td>
<td>$9.215 \times 10^{-6}$</td>
<td>$5.800 \times 10^{-10}$</td>
<td>$2.407 \times 10^{-5}$</td>
</tr>
</tbody>
</table>
Modeling Filter Latency: Results

100 calibration trials
1 x 10^8 cycles per trial
Modeling Filter Latency: Summary

- Avoiding sleep rests during calibration can reduce error in original method (why?)
- But with new method:
  - Error can be further reduced
  - Finer temporal grain
  - Can deploy different kinds of operation (logical, integer, floating point)
Goal: Fill in the gap in our knowledge of how tree-based overlay network topologies impact tool and application performance.

I’ve filled a gap so that we can start to fill this gap.

Next: put it to use.

Thanks: Dorian Arnold, Evan Dye, Tyler Groves, Samuel Gutierrez, and everyone else I’ve pestered over this project.