Introduction to Performance Analysis

Yury Oleynik
oleynik@in.tum.de
Outline

- Motivation for performance tuning
- Performance optimization cycle
- Basics of performance analysis
- Performance analysis techniques
Motivation

• **Goal of performance optimization**
  • Reduce resource consumption to the acceptable limit or
  • Produce more results under a given consumption limit

• **Resources:**
  • Execution time
  • Memory consumption
  • Power consumption
  • ...

• **Derive non-functional requirements**
Motivation – Parallelism

Need for Parallelism

Parallelism is crucial for optimal performance

- Platform Potential With Parallelism
- Growing gap!
- Without parallelism
- GHz Era ↔ Multi-core
- Time
Performance optimization challenges

• High complexity in parallel and distributed systems
  • Application
    • Algorithm, data structures
  • Parallel programming interface
    • Compiler, parallel libraries, communication, synchronization
  • Operating system
    • Process and memory management, IO
  • Hardware
    • CPU, memory, network
• Motivation for performance tuning
• **Performance optimization cycle**
• Basics of performance analysis
• Performance analysis techniques
Performance optimization cycle

1. Define requirements
2. Prepare experiment
3. Measure performance
4. Analyze bottlenecks
5. Tune application
6. Validate results

The cycle then repeats.
Defining requirements

- **Identify performance optimization objective**
  - Execution time
  - Memory footprint
  - ...

- **Identify the target execution context**
  - Data set
  - Environment configuration
  - Algorithms

- **Define the acceptance criteria**
  - Execution time below 4 hours

- **Refine requirements if needed**
  - Consider abortion of the tuning to avoid wasting resources
Preparing application

• Prepare the test configuration reflecting the target execution context

• Design performance experiment
  • Execution aspect to be analyzed
  • Performance analysis tool to be used
  • ...

• Instrument application
  • Insertion of the probe functions
  • Consider granularity vs overhead
  • Manual or automatic
Measuring performance

• Prepare execution environment
  • Batch scripts
  • Environment settings

• Start instrumented application using the selected performance analysis tool

• Raw performance data is produced
Analyzing performance

• Process raw performance data
  • Visualize
  • Compute derived metrics

• Identify application hotspots
  • Against the target optimization criteria

• Relate hotspots back to the source code

• Compute severity of the hotspots

• Rank the hotspots, identify bottleneck

• Select hotspots for optimization
Tuning performance

- Identify possible bottleneck optimization recipes
  - Compiler optimization
  - Loop transformation
  - Hardware specific pitfalls
  - Choose better algorithms
  - …

- Is done manually by application developer…
Validating achieved performance

- Check the acceptance criteria to be satisfied
- Check the correctness of the produced results
- Record the achieved performance and the applied tuning actions
Outline

• Motivation for performance tuning
• Performance optimization cycle
• **Basics of performance analysis**
• Performance analysis techniques
Performance analysis, prediction and benchmarking

- **Performance analysis** determines the performance on a given machine.

- **Performance prediction** allows to evaluate programs for a hypothetical machine. It is based on:
  - runtime data of an actual execution
  - machine model of the target machine
  - analytical techniques
  - simulation techniques

- **Benchmarking** determines the performance of a computer system on the basis of a set of typical applications.
Sequential vs parallel performance

- Factors which influence performance of parallel programs
  - “Sequential” factors
    - Computation
    - Cache and memory
    - Input / output
  - “Parallel” factors
    - Communication (Message passing)
    - Threading
    - Synchronization
How to decide whether a code performs well:

- Comparison of measured MFLOPS with peak performance
- Comparison with a sequential version

\[ \text{speedup}(p) = \frac{t_s}{t_p} \]

- Estimate distance to ideal time via overhead classes
  - \( t_{\text{mem}} \)
  - \( t_{\text{comm}} \)
  - \( t_{\text{sync}} \)
  - \( t_{\text{red}} \)
  - ...

![Graph showing speedup vs. number of processors]
The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

Other Performance Metrics

• **Scaled speedup**
  - Problem size grows with machine size

\[
scaled\_speedup(p) = \frac{t_s(n_p)}{t_p(n_p)}
\]

• **Parallel efficiency:** Percentage of ideal speedup

\[
\text{efficiency (p)} = \frac{\text{speedup(p)}}{p} = \frac{t_s}{t_p * p}
\]
• Motivation for performance tuning
• Performance optimization cycle
• Basics of performance analysis
• Performance analysis techniques
What the tools can offer?

- **Performance Tuning**
- **Performance Diagnosis**
- **Performance Experimentation**
- **Performance Observation**

Performance Technology
- Experiment management
- Performance data storage

Performance Technology
- Data mining
- Models
- Expert systems

Performance Technology
- Instrumentation
- Measurement
- Analysis
- Visualization
• Understanding performance requires observation of performance properties.
• Performance tools and methodologies are primarily distinguished by what observations are made and how.
  – How application program is instrumented
  – What performance data are obtained
• Tools and methods cover broad range.
• Execution *actions* exposed as *events*
  – In general, actions reflect some execution state
    • presence at a code location or change in data
    • occurrence in parallelism context (thread of execution)
  – Events encode actions for observation

• Observation is *direct*
  – Direct instrumentation of program code (probes)
  – Instrumentation invokes performance monitoring
  – Measurement = event + performance data + context
Direct Observation: Instrumentation

- **Static instrumentation**
  - Program instrumented prior to execution

- **Dynamic instrumentation**
  - Program instrumented at runtime

- **Manual and automatic mechanisms**

- **Tools required for automatic support**
  - Source time: preprocessor, translator, compiler
  - Link time: wrapper library
  - Execution time: binary rewrite, dynamic

- **Advantages / disadvantages**
Indirect Performance Observation

• Events are actions external to program code
• Program code instrumentation is not used
• Performance is observed indirectly
  – Execution is interrupted
    • can be triggered by different events
  – Execution state is queried (sampled)
    • different performance data measured
• Performance attribution is inferred
  – Determined by execution context (state)
  – Observation resolution determined by interrupt period
  – Performance data associated with context for period
Direct / Indirect Comparison

• Direct performance observation
  😊 Measures performance data exactly
  😊 Links performance data with application events
  😞 Requires instrumentation of code
  😞 Measurement overhead can cause execution intrusion and possibly performance perturbation

• Indirect performance observation
  😊 Argued to have less overhead and intrusion
  😊 Can observe finer granularity
  😊 No code modification required (may need symbols)
  😞 Inexact measurement and attribution without hardware support
Critical issues

• Accuracy
  – Timing and counting accuracy depends on resolution
  – Any performance measurement generates overhead
    • Execution on performance measurement code
  – Measurement overhead can lead to intrusion
  – Intrusion can cause perturbation
    • alters program behavior

• Granularity
  – How many measurements are made
  – How much overhead per measurement

• Tradeoff (general wisdom)
  – Accuracy is inversely correlated with granularity
• How are measurements made?
  – Profiling
    • summarizes performance data during execution
    • per process / thread and organized with respect to context
  – Tracing
    • trace record with performance data and timestamp
    • per process / thread
Profiling

• Recording of aggregated information
  – Counts, time, …

• … about program and system entities
  – Functions, loops, basic blocks, …
  – Processes, threads

• Methods
  – Event-based sampling (indirect, statistical)
  – Direct measurement (deterministic)
Flat and Callpath Profiles

• Flat profile
  – Performance metrics for when event is active
  – Exclusive and inclusive

• Callpath profile
  – Performance metrics for calling path (event chain)
  – Differentiate performance with respect to program execution state
  – Exclusive and inclusive
Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}

void worker {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}

void master {
    ...
    send(B, tag, buf);
    ...
}

void worker {
    ...
    recv(A, tag, buf);
    ...
}

Process A:

Process B:

<table>
<thead>
<tr>
<th>Event</th>
<th>Process</th>
<th>Type</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTER</td>
<td>A</td>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>ENTER</td>
<td>B</td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>SEND</td>
<td>A</td>
<td>B</td>
<td>62</td>
</tr>
<tr>
<td>EXIT</td>
<td>A</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>RECV</td>
<td>B</td>
<td>A</td>
<td>68</td>
</tr>
<tr>
<td>EXIT</td>
<td>B</td>
<td>2</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>master</td>
<td>worker</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>master</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>worker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER</td>
<td>2</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND</td>
<td>B</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV</td>
<td>A</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**
- **A** (master) and **B** (worker) are represented with different colors.
- **A** and **B** interact with each other through **Send** and **Receive** actions.
- The timeline indicates the sequence of events:
  - 58 (A enters)
  - 60 (B enters)
  - 62 (A sends)
  - 64 (A exits)
  - 68 (B receives)
  - 69 (B exits)

**Legend:**
- **yellow** represents the main thread.
- **red** represents the master thread.
- **blue** represents the worker thread.
Profiling / Tracing Comparison

- **Profiling**
  - 😊 Finite, bounded performance data size
  - 😊 Applicable to both direct and indirect methods
  - 😐 Loses time dimension (not entirely)
  - 😞 Lacks ability to fully describe process interaction

- **Tracing**
  - 😊 Temporal and spatial dimension to performance data
  - 😊 Capture parallel dynamics and process interaction
  - 😐 Some inconsistencies with indirect methods
  - 😞 Unbounded performance data size (large)
  - 😞 Complex event buffering and clock synchronization
Questions?