Week 6

Parallel Performance Analysis and Tuning

Parallel Performance and Complexity

- To use a scalable parallel computer well, you must write high-performance parallel programs
- To get high-performance parallel programs, you must understand and optimize performance for the combination of programming model, algorithm, language, platform, …
- Unfortunately, parallel performance measurement, analysis and optimization cannot be an easy process
- Parallel performance is complex

Parallel Performance Evaluation

- Study of performance in parallel systems
  - Models and behaviors
  - Evaluative techniques
- Evaluation methodologies
  - Analytical modeling and statistical modeling
  - Simulation-based modeling
  - Empirical measurement, analysis, and modeling
- Purposes
  - Planning
  - Diagnosis
  - Tuning

Parallel Performance Engineering and Productivity

- Scalable, optimized applications deliver HPC promise
- Optimization through *performance engineering* process
  - Understand performance complexity and inefficiencies
  - Tune application to run optimally on high-end machines
- How to make the process more effective and productive?
- What performance technology should be used?
  - Performance technology part of larger environment
  - Programmability, reusability, portability, robustness
  - Application development and optimization productivity
- Process, performance technology, and its use will change as parallel systems evolve
- Goal is to deliver effective performance with high productivity value now and in the future

Motivation

- Parallel / distributed systems are complex
  - Four layers
    - application
      - algorithm, data structures
    - parallel programming interface / middleware
      - compiler, parallel libraries, communication, synchronization
    - operating system
      - process and memory management, IO
    - hardware
      - CPU, memory, network
- Mapping/interaction between different layers

Scalability of Parallel Architectures

- A parallel architecture is said to be scalable if it can be expanded (reduced) to a larger (smaller) system with a linear increase (decrease) in its performance (cost). This general definition indicates the desirability for providing equal chance for scaling up a system for improved performance and for scaling down a system for greater cost-effectiveness and/or affordability.
- Scalability is used as a measure of the system's ability to provide increased performance, for example, speed as its size is increased. In other words, scalability is a reflection of the system's ability to efficiently utilize the increased processing resources.
- The scalability of a system can be manifested as in the forms; *speed*, *efficiency*, *size*, *applications*, *generation*, and *heterogeneity*.
- **In terms of speed**, a scalable system is capable of increasing its speed in proportion to the increase in the number of processors

Performance Factors

- Factors which determine a program's performance are complex, interrelated, and sometimes hidden
- Application related factors
  - Algorithms dataset sizes, task granularity, memory usage patterns, load balancing. I/O communication patterns
- Hardware related factors
  - Processor architecture, memory hierarchy, I/O network
- Software related factors
  - Operating system, compiler/preprocessor, communication protocols, libraries

Utilization of Computational Resources

- Resources can be under-utilized or used inefficiently
  - Identifying these circumstances can give clues to where performance problems exist
- Resources may be "virtual"
  - Not actually a physical resource (e.g., thread, process)
- Performance analysis tools are essential to optimizing an application's performance
  - Can assist you in understanding what your program is "really doing"
  - May provide suggestions how program performance should be improved

Performance Analysis and Tuning: The Basics

- Most important goal of performance tuning is to reduce a program's wall clock execution time
  - Iterative process to optimize efficiency
  - Efficiency is a relationship of execution time
- So, where does the time go?
- Find your program's hot spots and eliminate the bottlenecks in them
  - ***Hot spot***: an area of code within the program that uses a disproportionately high amount of processor time
  - ***Bottleneck*** : an area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays
- Understand *what*, *where*, and *how* time is being spent

Sequential Performance

- Sequential performance is all about:
  - How time is distributed
  - What resources are used where and when
- "Sequential" factors
  - Computation
    - choosing the right algorithm is important
    - compilers can help
  - Memory systems and cache and memory
    - more difficult to assess and determine effects
    - modeling can help
  - Input / output

Parallel Performance

- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution
  - "Parallel" factors lead to overheads
    - concurrency (threading, processes)
    - interprocess communication (message passing)
    - synchronization (both explicit and implicit)
  - Parallel interactions also lead to parallelism inefficiency
    - load imbalances

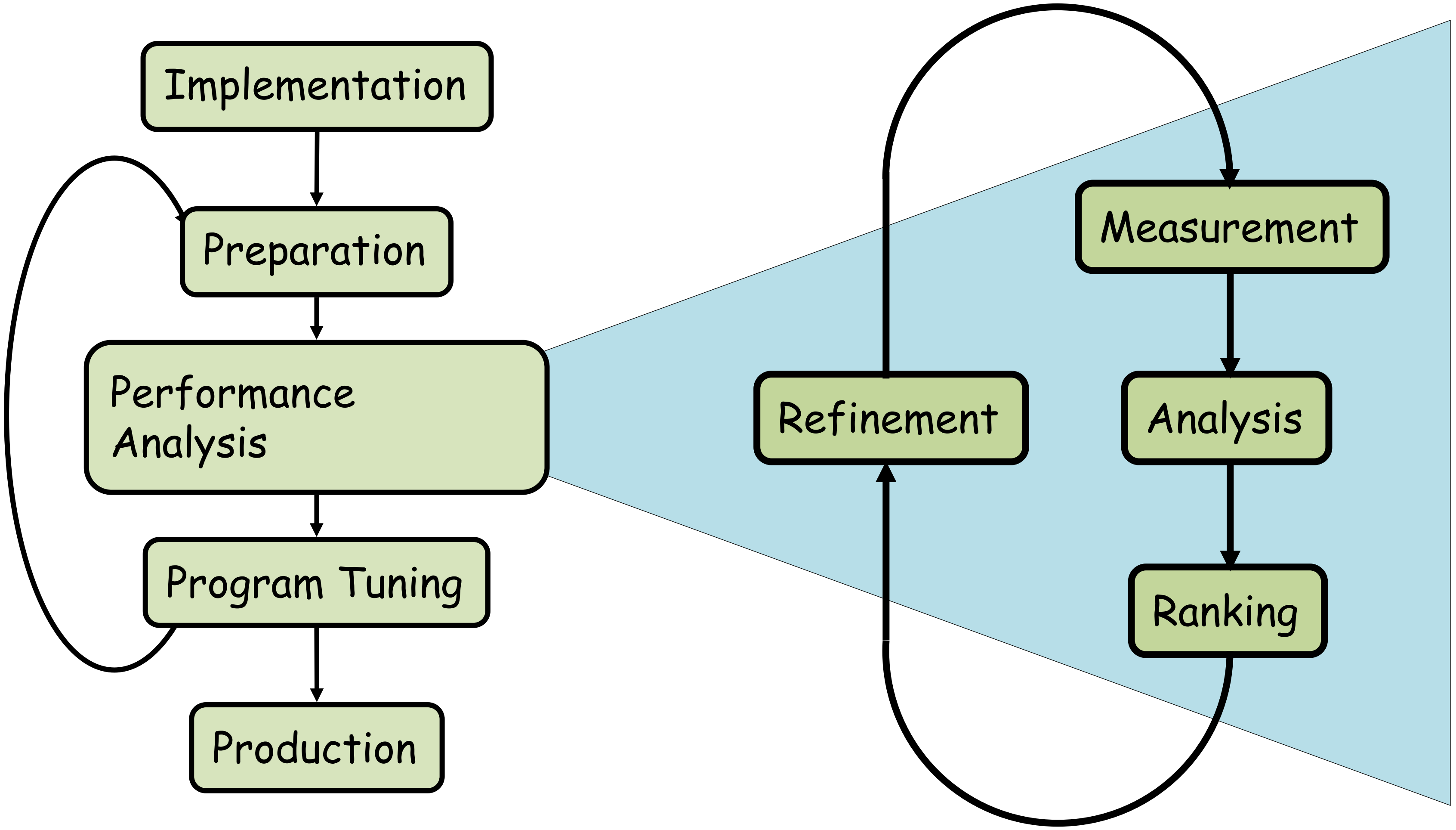
Sequential Performance Tuning

- Sequential performance tuning is a *time-driven* process
- Find the thing that takes the most time and make it take less time (i.e., make it more efficient)
- May lead to program restructuring
  - Changes in data storage and structure
  - Rearrangement of tasks and operations
- May look for opportunities for better resource utilization
  - Cache management is a big one
  - Locality, locality, locality!
  - Virtual memory management may also pay off
- May look for opportunities for better processor usage

Parallel Performance Tuning

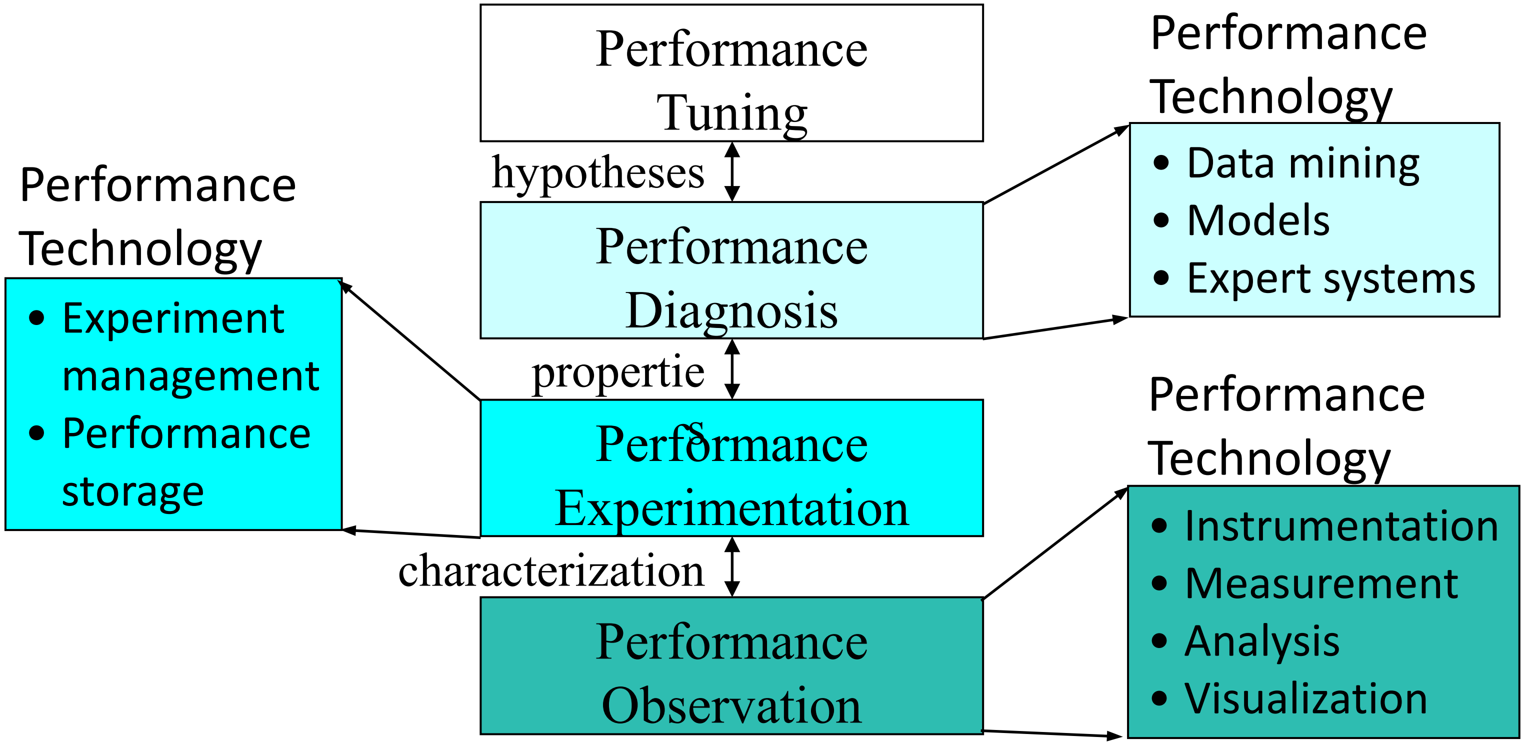
- In contrast to sequential performance tuning, parallel performance tuning might be described as *conflict-driven* or *interaction-driven*
- Find the points of parallel interactions and determine the overheads associated with them
- Overheads can be the cost of performing the interactions
  - Transfer of data
  - Extra operations to implement coordination
- Overheads also include time spent waiting
  - Lack of work
  - Waiting for dependency to be satisfied

Parallel Performance Engineering Process

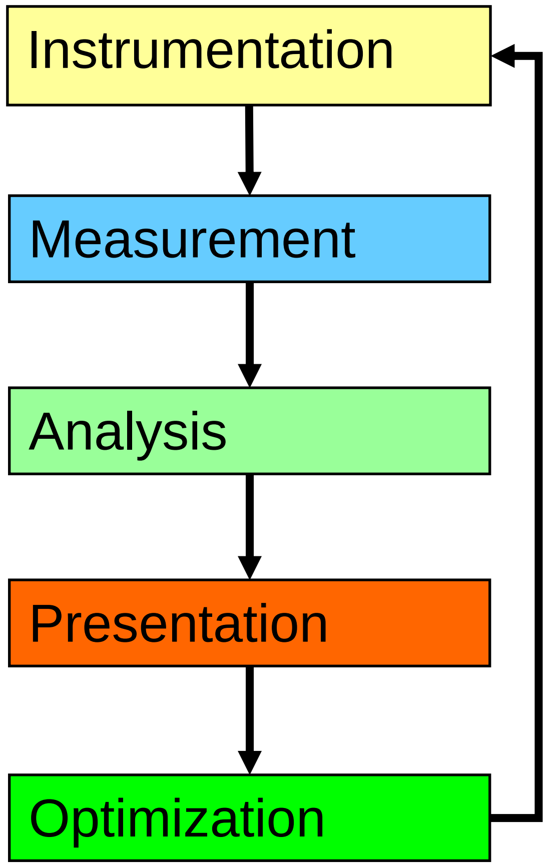


Parallel Performance Engineering Process

- Traditionally an empirically-based approach
  - observation ⇔ experimentation ⇔ diagnosis ⇔ tuning
- Performance technology developed for each level



Performance Analysis and Optimization Cycle



- Insertion of extra code (probes, hooks) into application
- Collection of data relevant to performance analysis
- Calculation of metrics, identification of performance problems
- Transformation of the results into a representation that can be easily understood by a human user
- Elimination of performance problems

Performance Metrics and Measurement

- Observability depends on measurement
- What is able to be observed and measured?
- A metric represents a type of measured data
  - *Count*: how often some thing occurred
    - calls to a routine, cache misses, messages sent, …
  - *Duration*: how long some thing took place
    - execution time of a routine, message communication time, …
  - Size: how big some thing is
    - message size, memory allocated, …
- A measurement records performance data
- Certain quantities can not be measured directly
  - *Derived metric*: calculated from metrics
    - rates of some thing (e.g., flops per second) are one example

Performance Benchmarking

- Benchmarking typically involves the measurement of metrics for a particular type of evaluation
  - Standardize on an experimentation methodology
  - Standardize on a collection of benchmark programs
  - Standardize on set of metrics
- High-Performance Linpack (HPL) for Top 500
- NAS Parallel Benchmarks
- SPEC
- Typically look at MIPS and FLOPS

SPEC: The Standard Performance Evaluation Corporation (SPEC) provides a suite of benchmarking tools and benchmarks for measuring the performance of computer systems in various domains, including CPU, graphics, and more.

Metrics like MIPS (Million Instructions Per Second) and FLOP (Floating-Point Operations Per Second) are often used to measure the computational capabilities of processors and systems.

How Is Time Measured?

- How do we determine where the time goes?

  "A person with one clock knows what time it is, a person with two clocks is never sure."
  Confucious (attributed)

- Clocks are not the same
  - Have different resolutions and overheads for access
- Time is an abstraction based on clock
  - Only as good (accurate) as the clock we use
  - Only as good as what we use it for

Execution Time

- There are different types of time
- ***Wall-clock time***
  - Based on realtime clock (continuously running)
  - Includes time spent in all activities
- ***Virtual process time*** (aka *CPU time*)
  - Time when process is executing (CPU is active)
    - user time and system time (can mean different things)
  - Does not include time when process is inherently waiting
- ***Parallel execution time***
  - Runs whenever *any* parallel part is executing
  - Need to define a global time basis

Observation Types

- There are two types of performance observation that determine different measurement methods
  - Direct performance observation
  - Indirect performance observation
- *Direct performance observation* is based on a scientific theory of measurement that considers the cost (overhead) with respect to accuracy
- *Indirect performance observation* is based on a sampling theory of measurement that assumes some degree of statistical stationarity

Direct Performance Observation

- 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 measurement
  - Event measurement = performance data + context
- Performance experiment
  - Actual events + performance measurements

Indirect Performance Observation

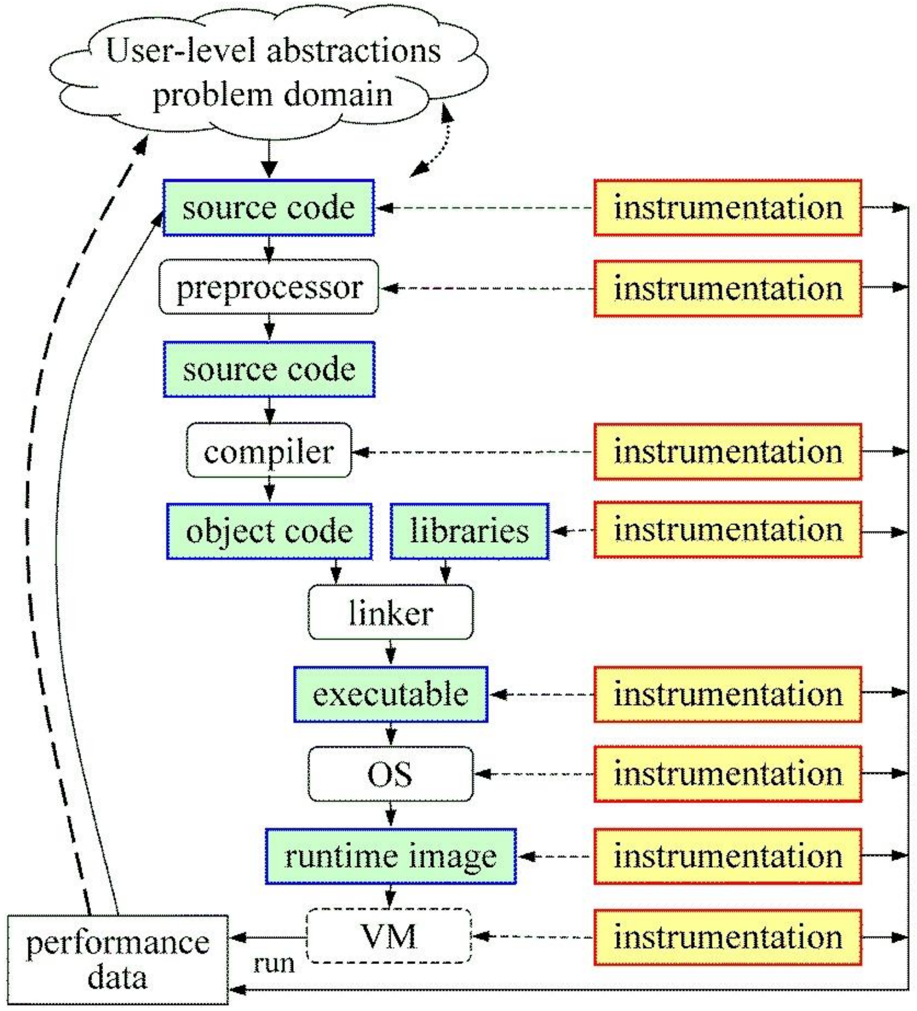
- Program code instrumentation is not used
- Performance is observed indirectly
  - Execution is interrupted
    - can be triggered by different events
  - Execution state is queried (sampled)
    - different performance data measured
  - *Event-based sampling* (EBS)
- Performance attribution is inferred
  - Determined by execution context (state)
  - Observation resolution determined by interrupt period
  - Performance data associated with context for period

Direct Observation: Events

- Event types
  - Interval events (begin/end events)
    - measures performance between begin and end
    - metrics monotonically increase
  - Atomic events
    - used to capture performance data state
- Code events
  - Routines, classes, templates
  - Statement-level blocks, loops
- User-defined events
  - Specified by the user
- Abstract mapping events

Direct Observation: Instrumentation

- Events defined by instrumentation access
- Instrumentation levels
  - Source code
  - Library code
  - Object code
  - Executable code
  - Runtime system
  - Operating system
- Levels provide different information / semantics
- Different tools needed for each level
- Often instrumentation on multiple levels required



Direct Observation: Techniques

- Static instrumentation
  - Program instrumented prior to execution
- Dynamic instrumentation
  - Program instrumented at runtime
- Manual and automatic mechanisms
- Tool required for automatic support
  - Source time: preprocessor, translator, compiler
  - Link time: wrapper library, preload
  - Execution time: binary rewrite, dynamic
- Advantages / disadvantages

Indirect Observation: Events/Triggers

- Events are actions external to program code
  - Timer countdown, HW counter overflow, …
  - Consequence of program execution
  - Event frequency determined by:
    - type, setup, number enabled (exposed)
- Triggers used to invoke measurement tool
  - Traps when events occur (interrupt)
  - Associated with events
  - May add differentiation to events

Indirect Observation: Context

- When events trigger, execution context determined at time of trap (interrupt)
  - Access to PC from interrupt frame
  - Access to information about process/thread
  - Possible access to call stack
    - requires call stack unwinder
- Assumption is that the context was the same during the preceding period
  - Between successive triggers
  - Statistical approximation valid for long running programs assuming repeated behavior

Performance Analysis and Visualization

- Gathering performance data is not enough
- Need to analyze the data to derive performance understanding
- Need to present the performance information in meaningful ways for investigation and insight
- Single-experiment performance analysis
  - Identifies performance behavior within an execution
- Multi-experiment performance analysis
  - Compares and correlates across different runs to expose key factors and relationships