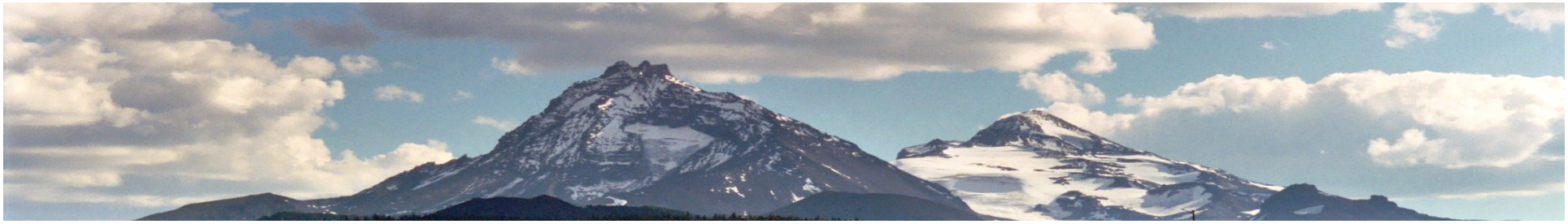


Lecture 2:
Principles and Methods

Allen D. Malony

Department of Computer and Information Science

UNIVERSITY OF OREGON

Parallel Programming

- To use a scalable parallel computer, you must be able to write parallel programs
- You must understand the programming model and the programming languages, libraries, and systems software used to implement it
- Unfortunately, parallel programming is not easy

Parallel Programming: Are we having fun yet?

Parallel Programming Models

- Two general models of parallel program
  - Task parallel
    - problem is broken down into tasks to be performed
    - individual tasks are created and communicate to coordinate operations
  - Data parallel
    - problem is viewed as operations of parallel data
    - data distributed across processes and computed locally
- Characteristics of scalable parallel programs
  - Data domain decomposition to improve data locality
  - Communication and latency do not grow significantly

Shared Memory Parallel Programming

- Shared memory address space
- (Typically) easier to program
  - Implicit communication via (shared) data
  - Explicit synchronization to access data
- Programming methodology
  - Manual
    - multi-threading using standard thread libraries
  - Automatic
    - parallelizing compilers
    - OpenMP parallelism directives
  - Explicit threading (e.g. POSIX threads)

Distributed Memory Parallel Programming

- Distributed memory address space
- (Relatively) harder to program
  - Explicit data distribution
  - Explicit communication via messages
  - Explicit synchronization via messages
- Programming methodology
  - Message passing
    - plenty of libraries to chose from (MPI dominates)
    - send-receive, one-sided, active messages
  - Data parallelism

Basic Parallel Programming Paradigm: SPMD

- SPMD: Single Program Multiple Data
- One program executes on all processors
  - Basic paradigm for implementing parallel programs
  - Process-dependent cases are handled in the program
  - Parallelism is "programmed in"
- Easier to manage program for scalability
- MPMD: Multiple Program Multiple Data
  - Support compositing of multiple program together

Parallel Programming: Still a Problem?

Parallel Computing and Scalability

- Scalability in parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network
  - Avoid critical architecture bottlenecks
- Scalability in computational problem
  - Problem size
  - Computational algorithms
    - computation to memory access ratio
    - computation to communication ratio
- Parallel programming models and tools
- Performance scalability

Amdahl's Law

- T_{seq} : sequential execution time that cannot be parallelized
- T_{par} : sequential execution time that can be parallelized
- $T_1 = T_{seq} + T_{par} \Rightarrow T_{par} = T_1 - T_{seq}$
- $T_p = T_{seq} + T_{par} / p$ (assume fully parallelized)
- As $p \rightarrow \infty$, $T_p \rightarrow T_{seq}$
- Let f_{seq} be the fraction T_{seq} / T_1 and $S_p = T_1 / T_p$
- $Speedup = S_p = T_1 / T_p = T_1 / (T_{seq} + T_{par} / p)$
  $= 1 / (f_{seq} + T_{par} / pT_1) = 1 / (f_{seq} + (1 - f_{seq}) / p)$
  - As $p \rightarrow \infty$, $S_p = S_\infty \rightarrow 1 / f_{seq}$
- Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!

Amdahl's Law and Scaled Speedup

- Amdahl's Law makes it hard to obtain good speedup

f_{seq} * 100%	10%	5%	2%	1%	.1%
S_∞	10	20	50	100	1000

- Change perspective on the problem
- Consider scaling of problem size as # processors scale
- T_{seq} : sequential execution time (1 and p processors)
- T_{par} : execution time in parallel mode on p processors
- $T_p = T_{seq} + T_{par}$, $T_1 = T_{seq} + pT_{par}$
- Let f_{par} be the fraction T_{par} / T_p
- *Scaled speedup* $= S_p = 1 + (p\text{-}1)T_{par} / T_p = 1 + (p\text{-}1)f_{par}$

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 can 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

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

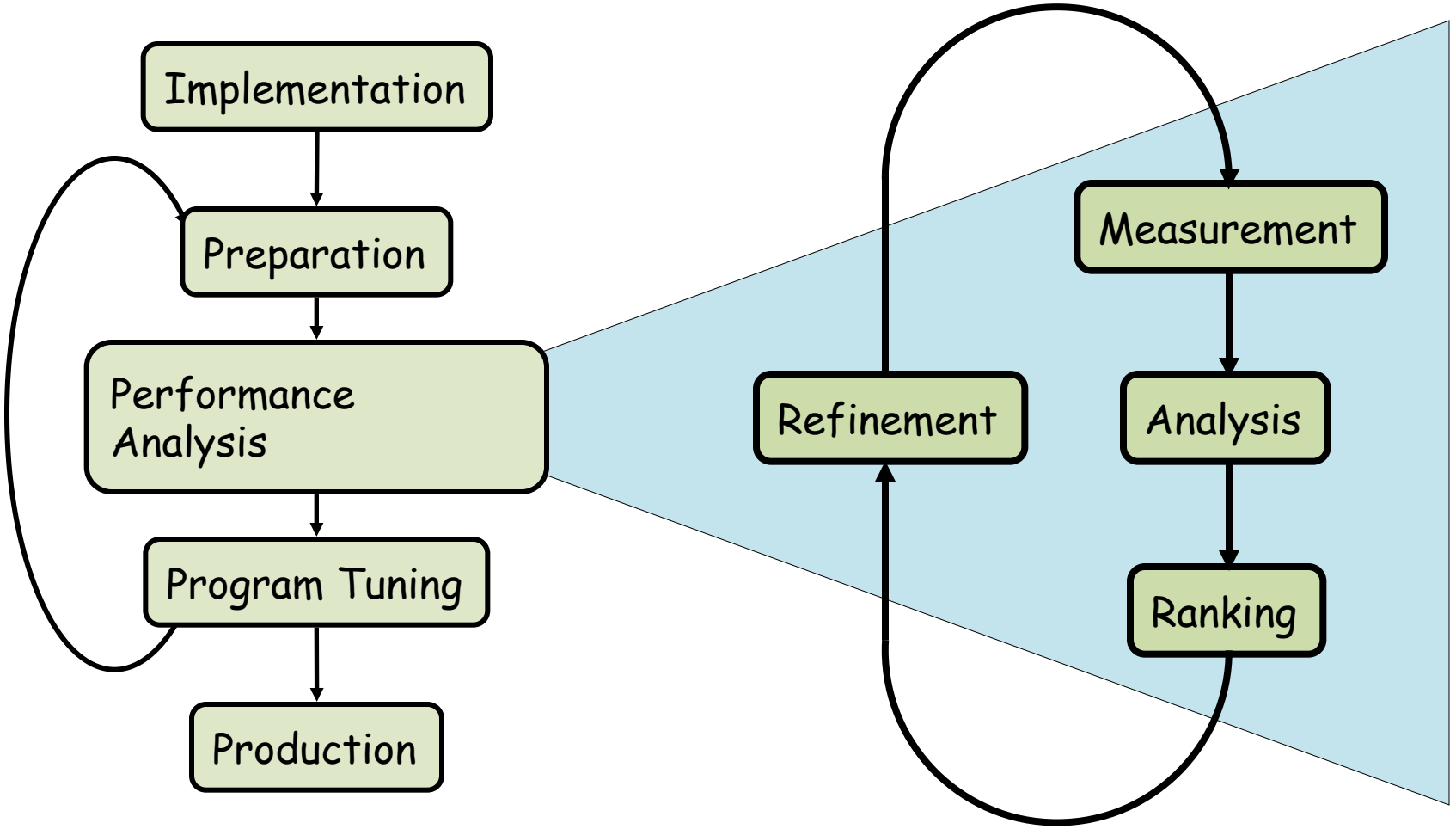
Parallel Performance Tuning

- 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

Interesting Performance Phenomena

- Superlinear speedup
  - Speedup in parallel execution is greater than linear
  - $S_p > p$
  - How can this happen?
- Need to keep in mind the relationship of performance and resource usage
- Computation time (i.e., real work) is not simply a linear distribution to parallel threads of execution
- Resource utilization thresholds can lead to performance inflections

Parallel Performance Engineering Process

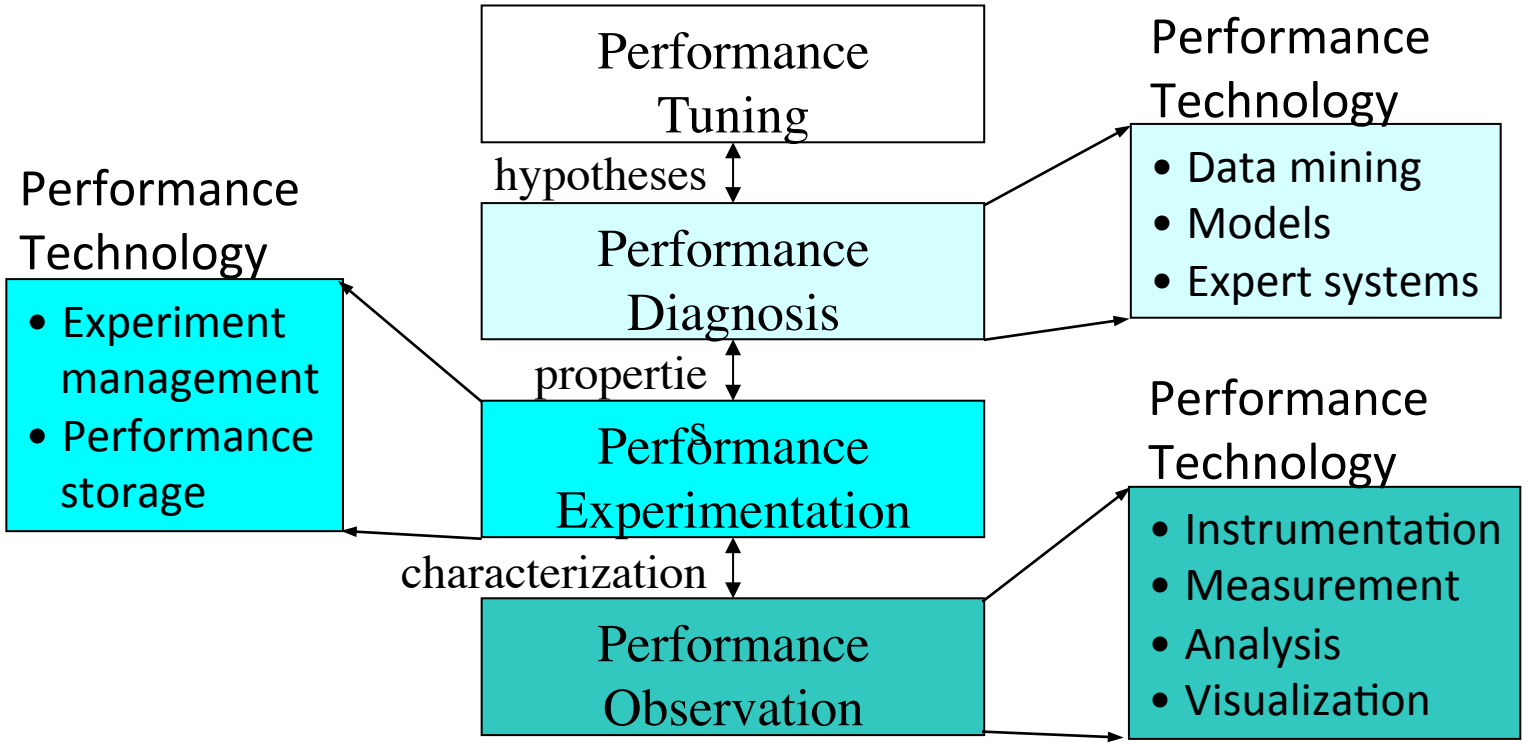


Performance Observability (Guiding Thesis)

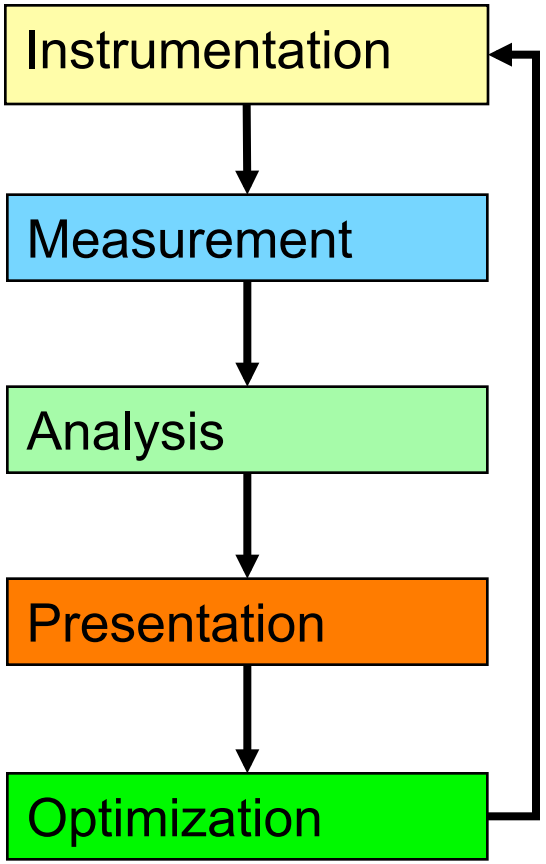
- Performance evaluation problems define the requirements for performance analysis methods
- Performance observability is the ability to "accurately" capture, analyze, and present (collectively observe) information about computer system/software performance
- Tools for performance observability must balance the need for performance data against the cost of obtaining it (environment complexity, performance intrusion)
  - Too little performance data makes analysis difficult
  - Too much data perturbs the measured system.
- Important to understand performance observability complexity and develop technology to address it

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

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

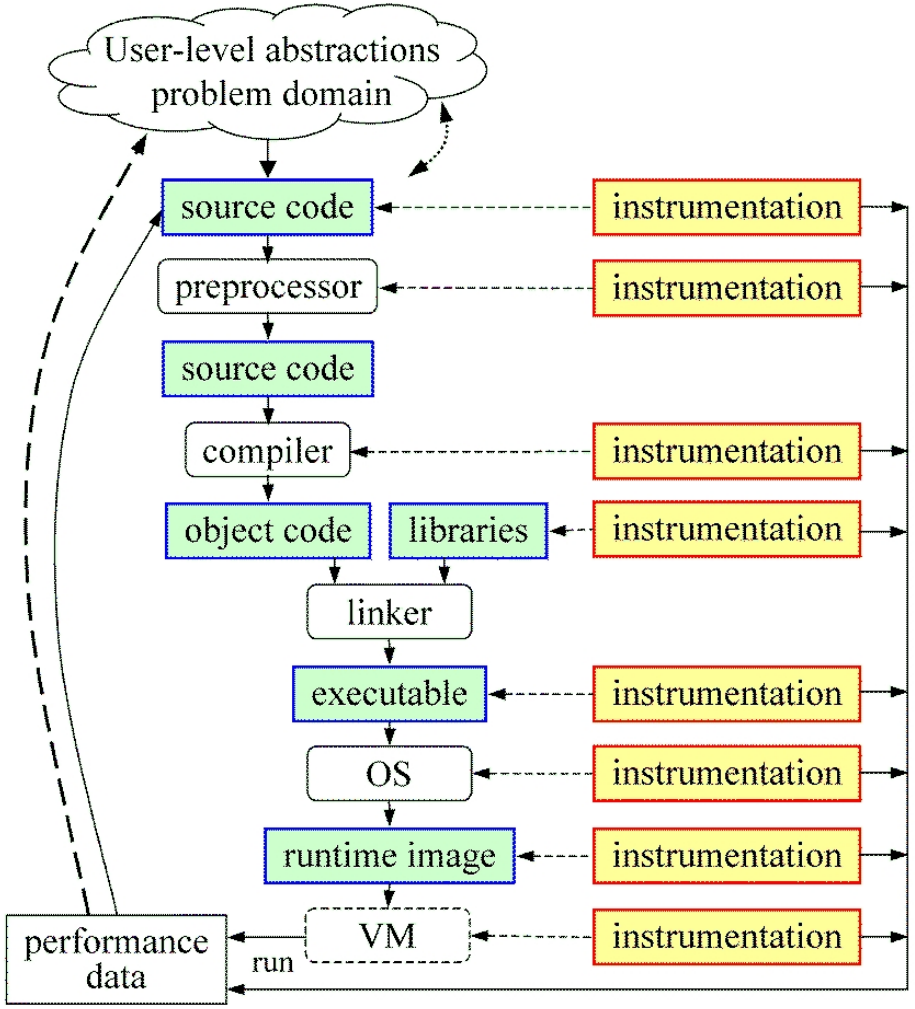
- Program code instrumentation is not used
- Performance is observed indirectly
  - Execution is interrupted
    - can be triggered by different events
  - Execution state is queried (sampled)
    - different performance data measured
  - *Event-based sampling* (EBS)
- Performance attribution is inferred
  - 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

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

Measurement Techniques

- When is measurement triggered?
  - External agent (indirect, asynchronous)
    - sampling via interrupts, hardware counter overflow, …
  - Internal agent (direct, synchronous)
    - through code modification (instrumentation)
- How are measurements made (data recorded)?
  - 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

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

Measured Performance

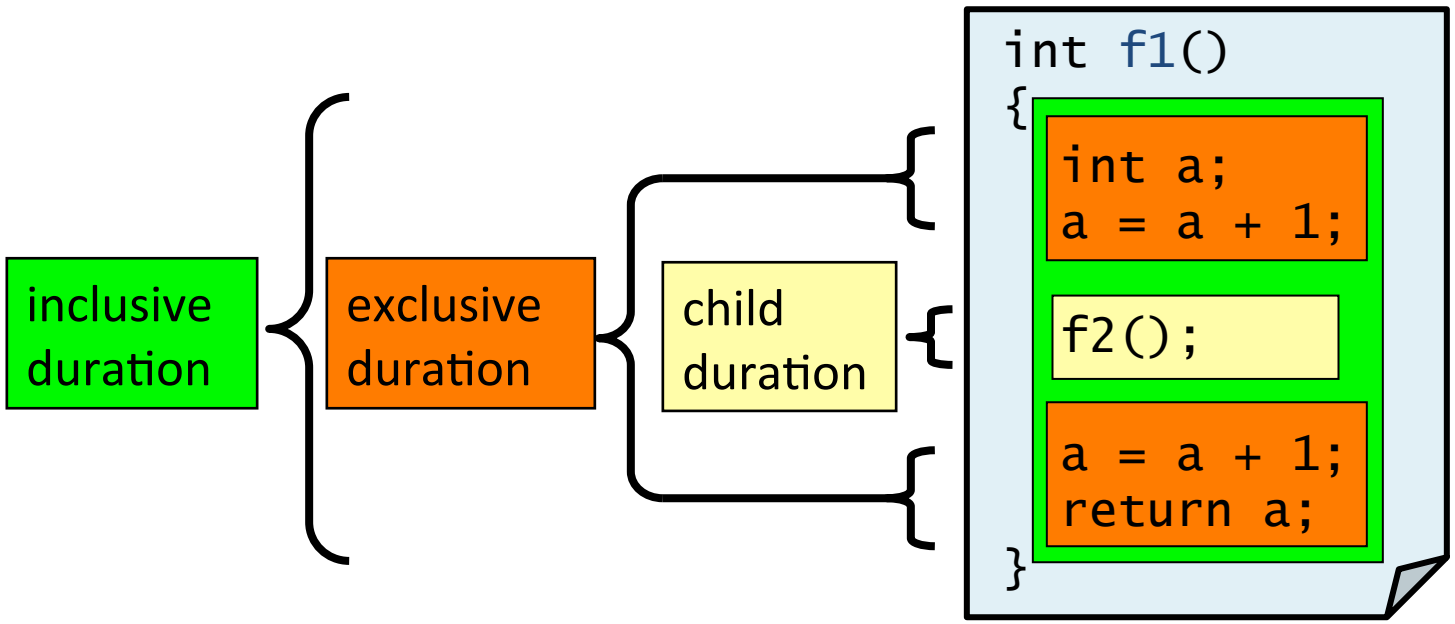
- Counts
- Durations
- Communication costs
- Synchronization costs
- Memory use
- Hardware counts
- System calls

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)

Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions



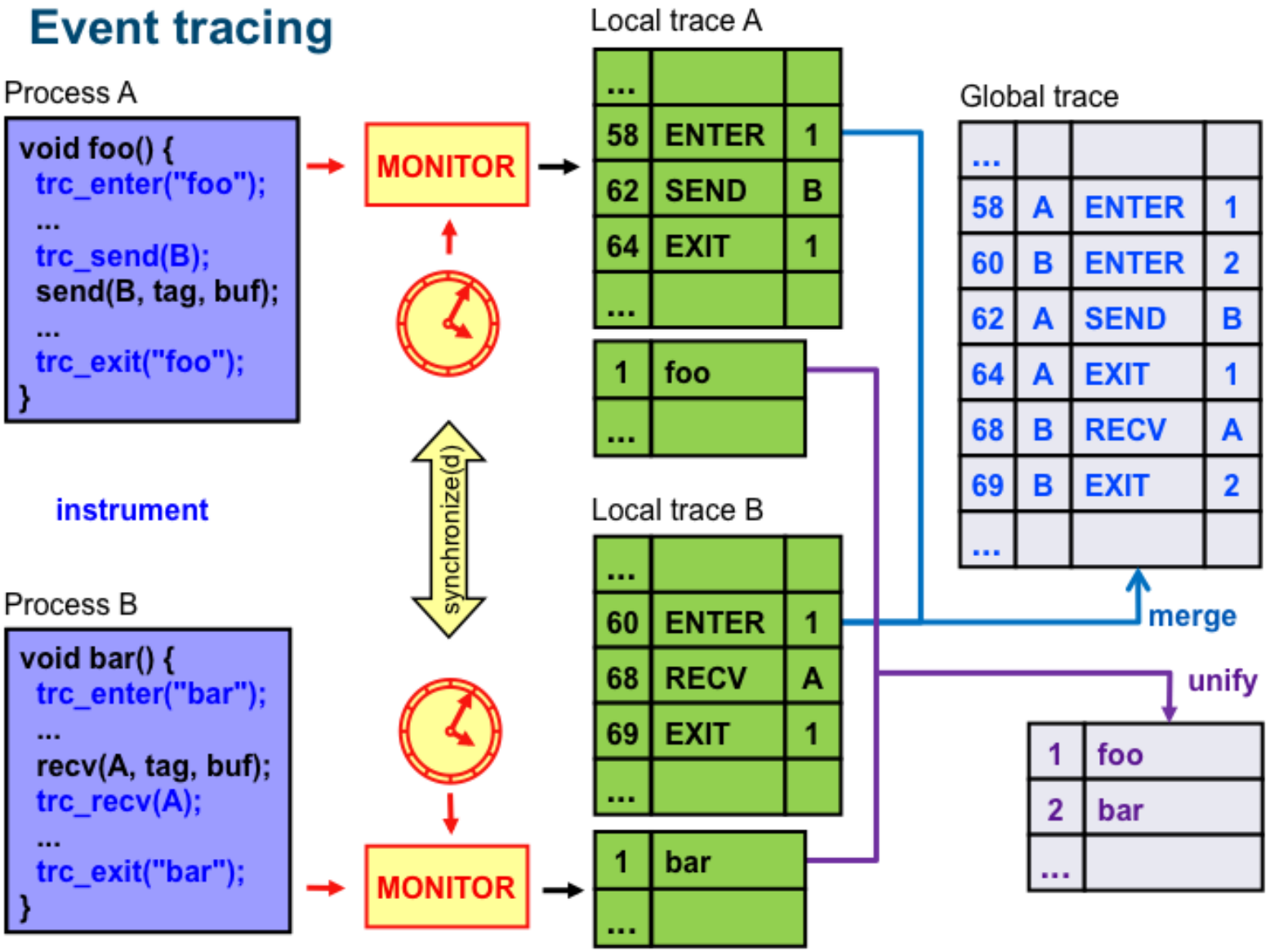
Flat and Callpath Profiles

- Static call graph
  - Shows all parent-child calling relationships in a program
- Dynamic call graph
  - Reflects actual execution time calling relationships
- 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

Measurement Methods: Tracing

- Recording information about significant points (events) during execution of the program
  - Enter/leave a code region (function, loop, …)
  - Send/receive a message ...
- Save information in *event record*
  - Timestamp, location ID, event type
  - Any event specific information
- An *event trace*
  - Stream of event records sorted by time
- Main advantage is that it can be used to reconstruct the dynamic behavior of the parallel execution
  - Abstract execution model on level of defined events

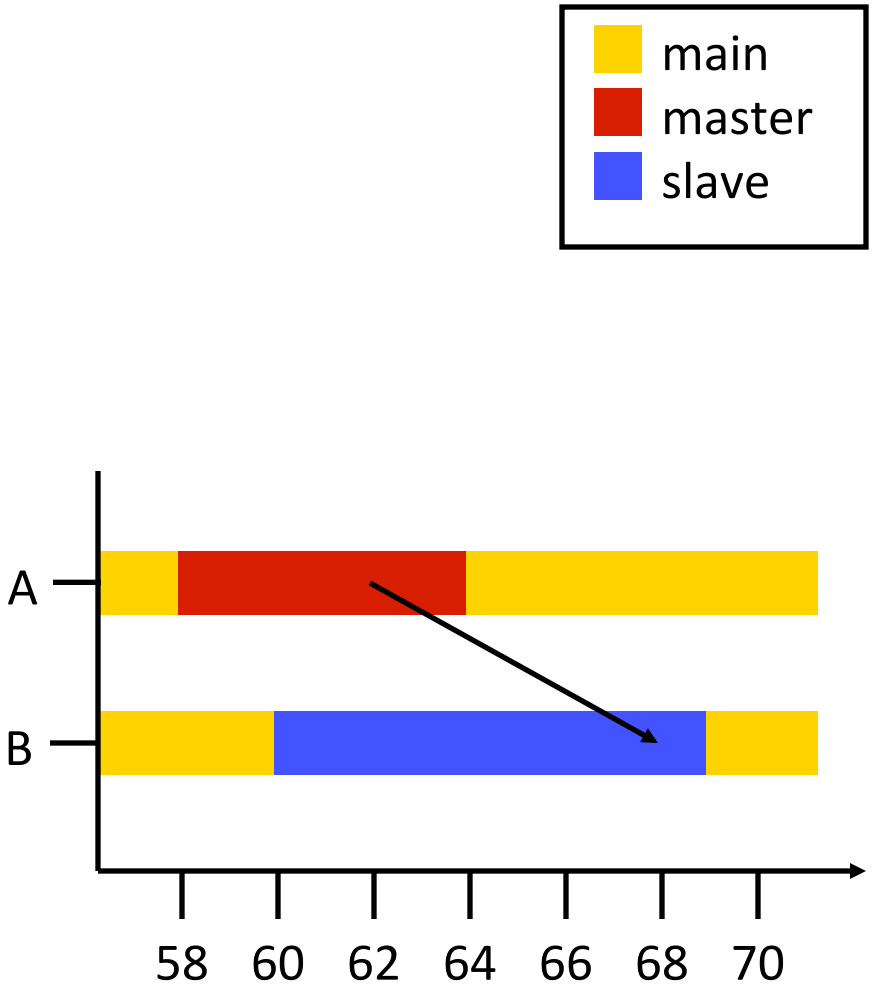
Event Tracing



Tracing: Time-line Visualization

1	master
2	slave
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



Trace File Formats

- There have been a variety of tracing formats developed over the years and supported in different tools
- Vampir
  - *VTF*: family of historical ASCII and binary formats
- MPICH / JumpShot
  - *ALOG, CLOG, SLOG, SLOG-2*
- Scalasca
  - *EPILOG* (Jülich open-source trace format)
- Paraver (BSC, CEPBA)
- TAU Performance System
- Convergence on *Open Trace Format (OTF)*

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
  - ☺ Can derive parallel profiles for any time region
  - 😐 Some inconsistencies with indirect methods
  - ☹ Unbounded performance data size (large)
  - ☹ Complex event buffering and clock synchronization

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