CIS 631
Parallel Processing

Lecture 4: Parallel Performance Analysis and Engineering

Allen D. Malony

malony@cs.uoregon.edu

Department of Computer and Information Science

University of Oregon

Acknowledgements

- Portions of the lectures slides were adopted from:
  - Bernd Mohr, “Parallel Programming Models, Tools and Performance Analysis,” Tutorial, Research Centre Juelich, 2002.
  - Allen D. Malony, “Computational Science, Scalable Parallel Computing, and Performance Tools,” Alexander von Humboldt lecture, Research Centre Juelich, 2002.
  - Lawrence Livermore National Laboratory, “Parallel Performance Tools Tutorial,” 2002.
  - Various TAU presentations and tutorials.

Outline

- Review
- Parallel performance analysis problem
- Parallel performance analysis methodology
- Measurement and analysis techniques
- Performance engineering approach

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?

Source: Bernd Mohr

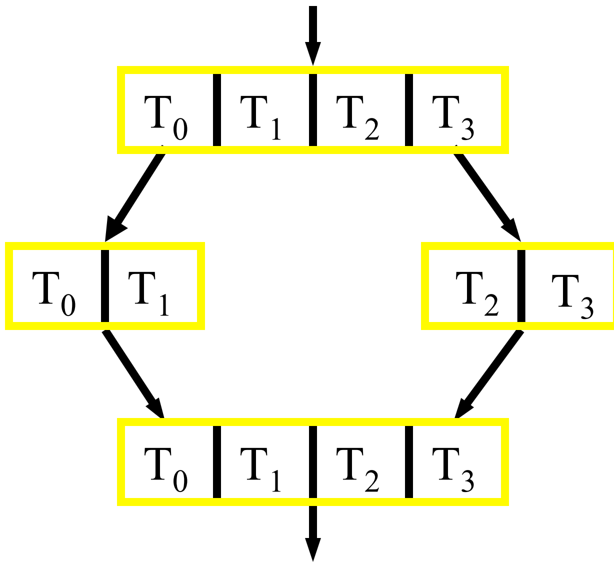
Parallel Programming Models

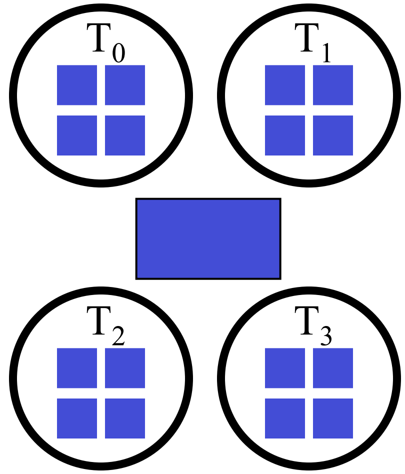
- Two general models of parallel program
  - Task parallel
    - Problem is broken down into tasks to be performed
    - Individual tasks are created and communicate to coordinate operations
  - Data parallel
    - Problem is viewed as operations of parallel data
    - Data distributed across processes 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)

Parallel Programming Model: Threads



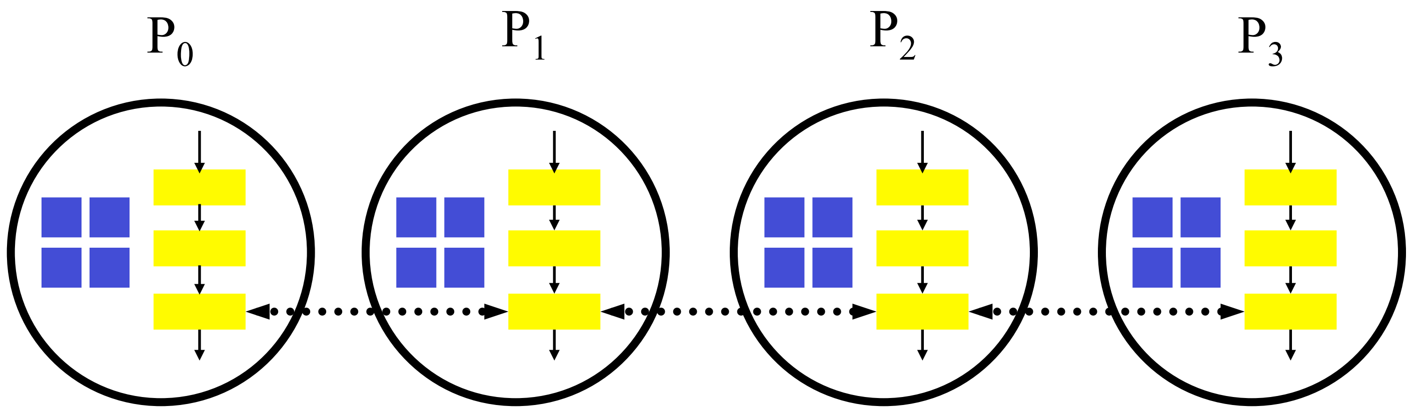


- Global style
- Shared and private data
- Work distribution onto threads for global operations
- Domain decomposition determines work distribution

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
  - Shared virtual memory

Paralle Programming Model: Message Passing



- Local style
- Domain decomposition leads to data distribution
- Explicit communication and synchronization
- Higher programming overhead
- Message passing libraries

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 inside the program

```
if (processor == 42) then
  call do_something()
else
  call do_something_else()
endif
```

- Parallelism is “programmed in”
- Easier to manage program for scalability

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

- To use a scalable parallel computer well, you must be able to 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 analysis and optimization is not an easy process

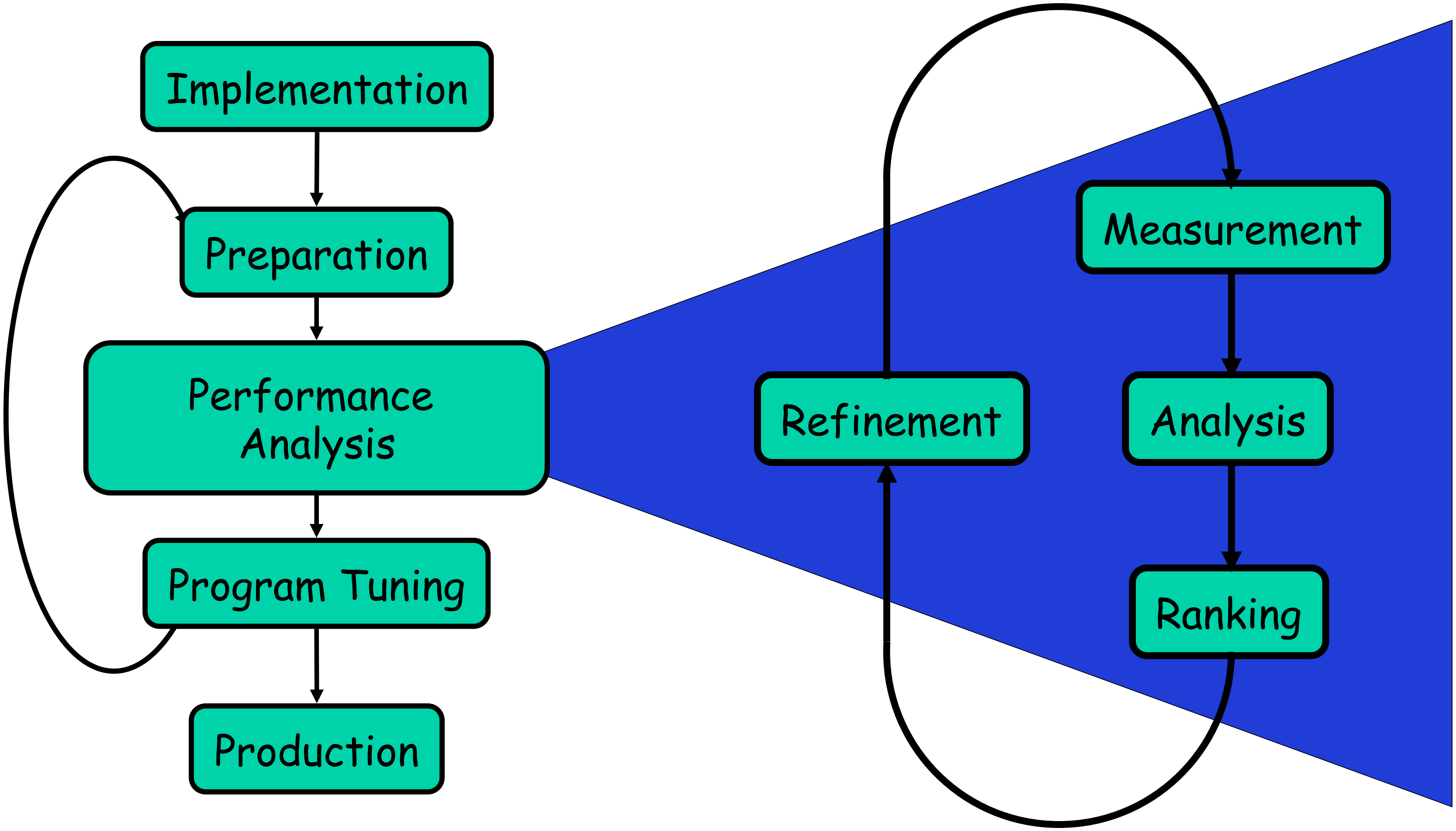
Parallel Performance Evaluation

- 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

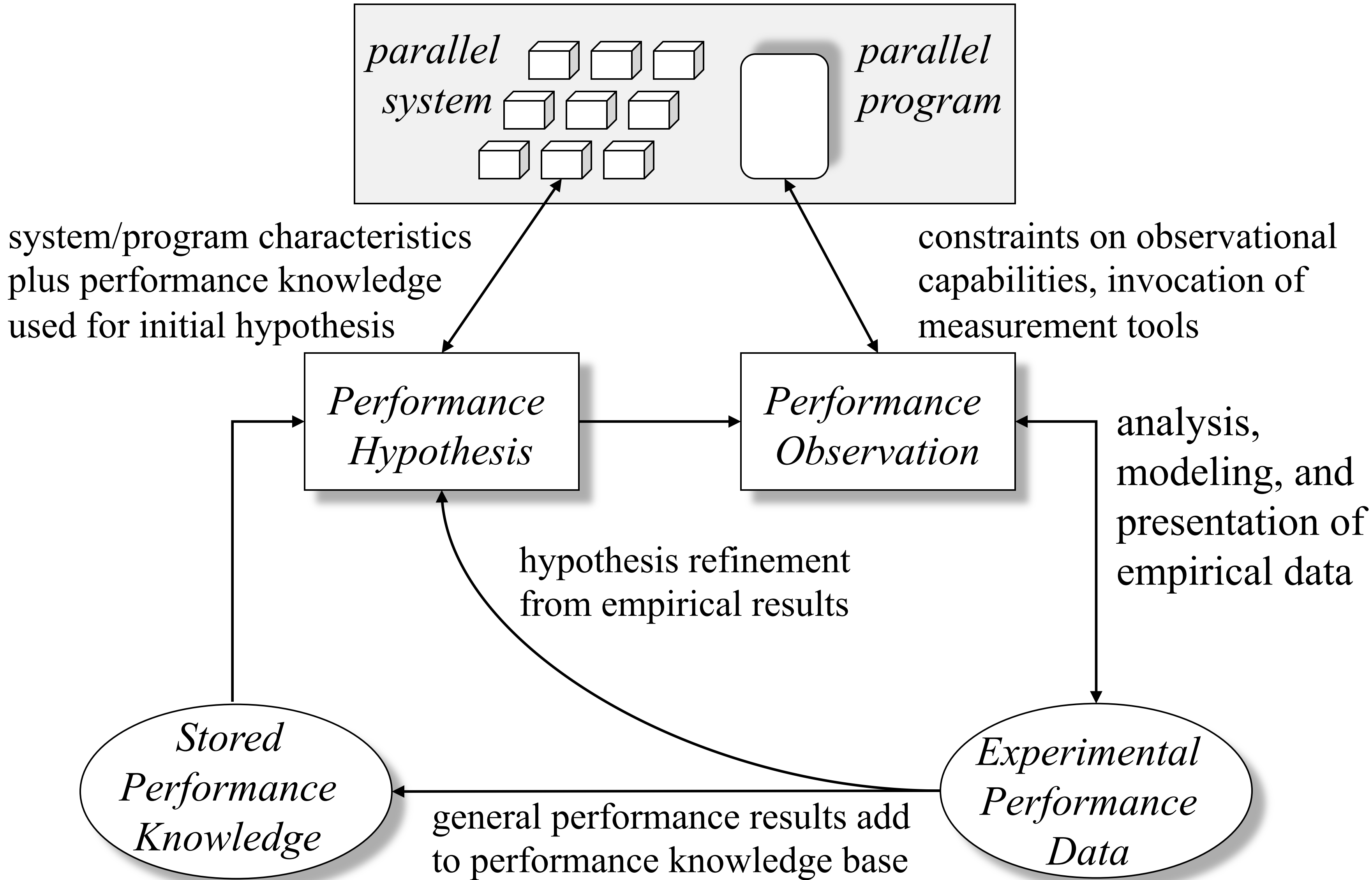
Performance Observability (My Guiding Thesis)

- 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 Analysis Process



Parallel Performance Analysis Environment



Performance Analysis and Tuning

- Successful parallel performance tuning process
  - Characterization: finding critical performance problems
  - Diagnosis: determining performance problem causes
  - Hypothesis testing: selection of performance optimization
  - Hypothesis validation: analyzing tuning results
- Reasoning and intuition only take you so far
- Need to make empirical observations
  - Performance instrumentation tools
  - Performance measurement tools
  - Performance analysis tools

Performance Factors

- Factors which determine a program's performance are complex, interrelated, and sometimes hidden
- Application related factors
  - Algorithms dataset sizes
  - Memory usage patterns
  - I/O communication patterns
  - Task Granularity
  - Load Balancing
  - Amdahl's Law
- Hardware related factors
  - Processor architecture
  - Memory hierarchy
  - I/O network
- Software related factors
  - Operating system
  - Compiler preprocessor
  - Communication protocols
  - Libraries

Utilization of Computational Resources

- Often resources are under-utilized or used inefficiently
- Identifying these circumstances can give clues to where performance problems exist
- Resources may be "virtual" (i.e., not a physical resource)
  - Thread or 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 versus Parallel Performance

- Sequential performance is all about how time is distributed and what resources are used where and when
- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution (i.e., its sequential performance)
  - Parallel interactions lead to overheads
    - synchronization
    - communication
  - 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 versus Sequential

- 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

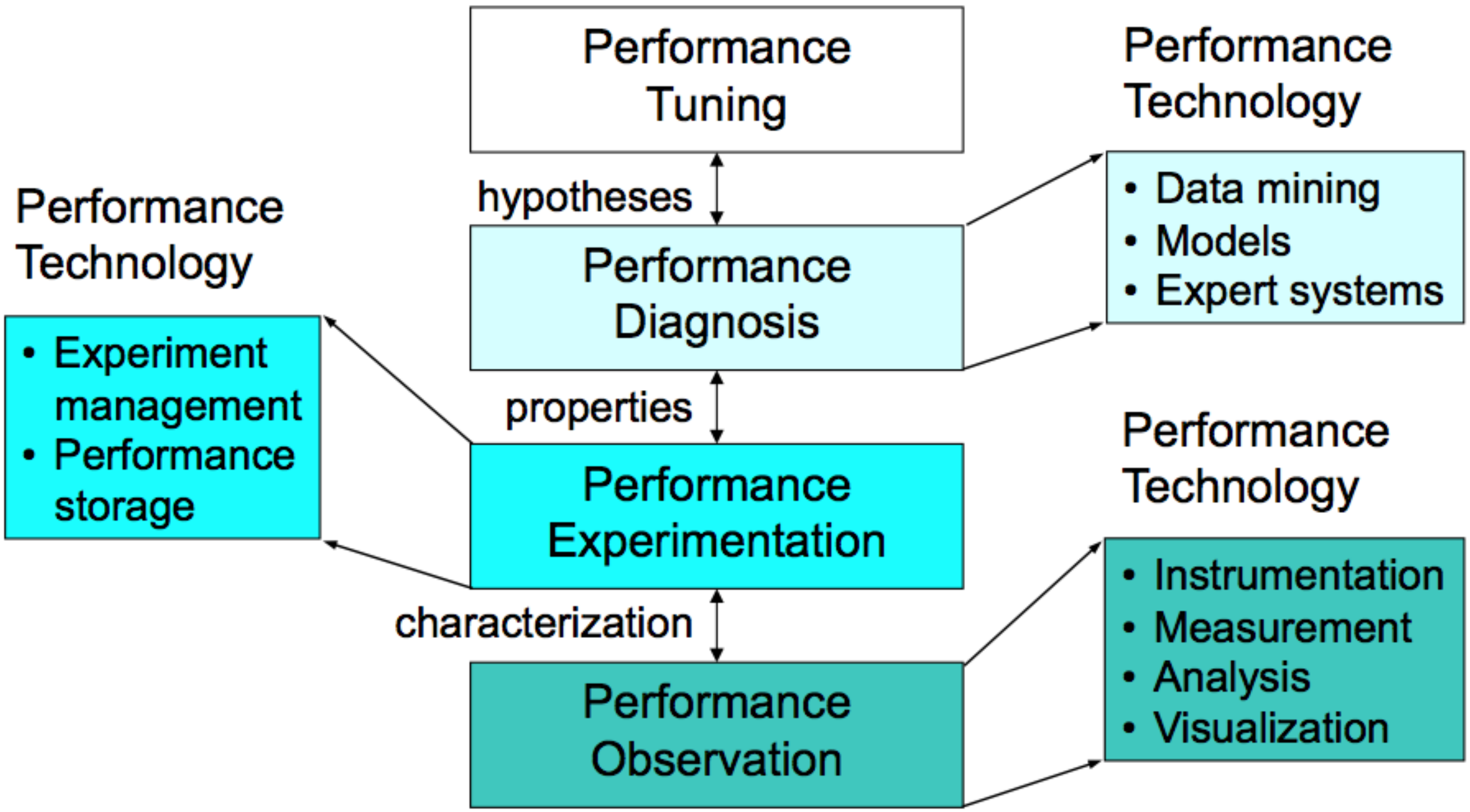
- Superlinear speedup
  - Speedup in parallel execution is greater than linear
  - $S_p > p$
  - How can this happen?
- Need to keep in mind the relationship of performance and resource usage
- Computation time (i.e., real work) is not simply a linear distribution to parallel threads of execution
- Resource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

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
- *"Define time."* Bill Clinton (attributed)
- Time is only as good (accurate) as the clock we use
- Clocks are not the same and, thus, time is not the same
  - *Wallclock time* – measured against "real" time
  - *CPU (virtual) time* – time accumulates (i.e., "ticks") only when process is executing
  - Clocks have different resolutions and overheads for access
    - affects accuracy

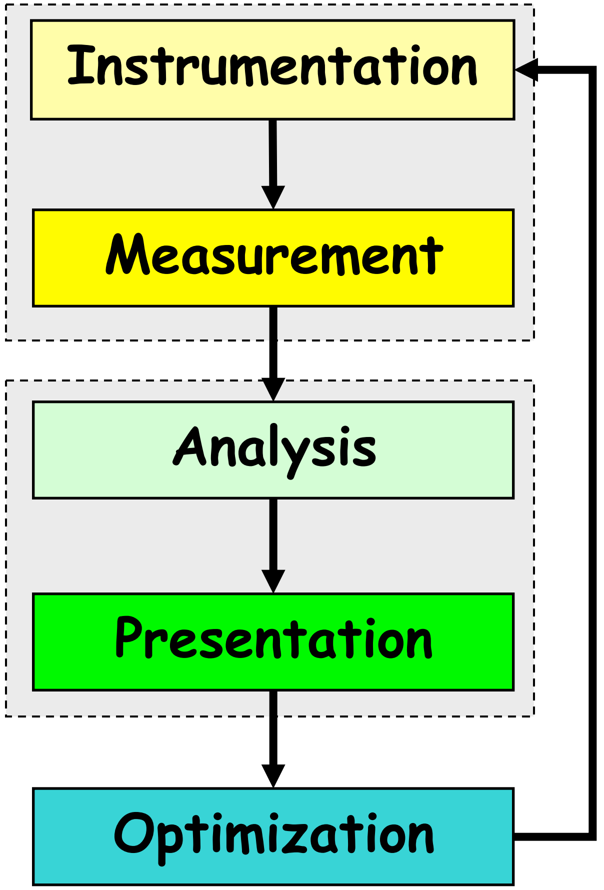
Performance Engineering

- Optimization process
- Effective use of performance technology



Performance Optimization Cycle

- Expose factors
- Collect performance data
- Calculate metrics
- Analyze results
- Visualize results
- Identify problems
- Tune performance



Parallel Performance Properties

- Parallel code performance is influenced by both sequential and parallel factors?
- Sequential factors
  - Computation and memory use
  - Input / output
- Parallel factors
  - Thread / process interactions
  - Communication and synchronization

Performance Observation

- Understanding performance requires observation of performance properties
- Performance tools and methodologies are primarily distinguished by what observations are made and how
  - What aspects of performance factors are seen
  - What performance data is obtained
- Tools and methods cover broad range

Metrics and Measurement

- Observability depends on measurement
- A metric represents a type of measured data
  - Count, time, hardware counters
- A measurement records performance data
  - Associates with program execution aspects
- Derived metrics are computed
  - Rates (e.g., flops)
- Metrics / measurements decided by need

Execution Time

- Wall-clock time
  - Based on realtime clock
- Virtual process time
  - Time when process is executing
    - user time and system time
  - Does not include time when process is stalled
- Parallel execution time
  - Runs whenever any parallel part is executing
  - Global time basis

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

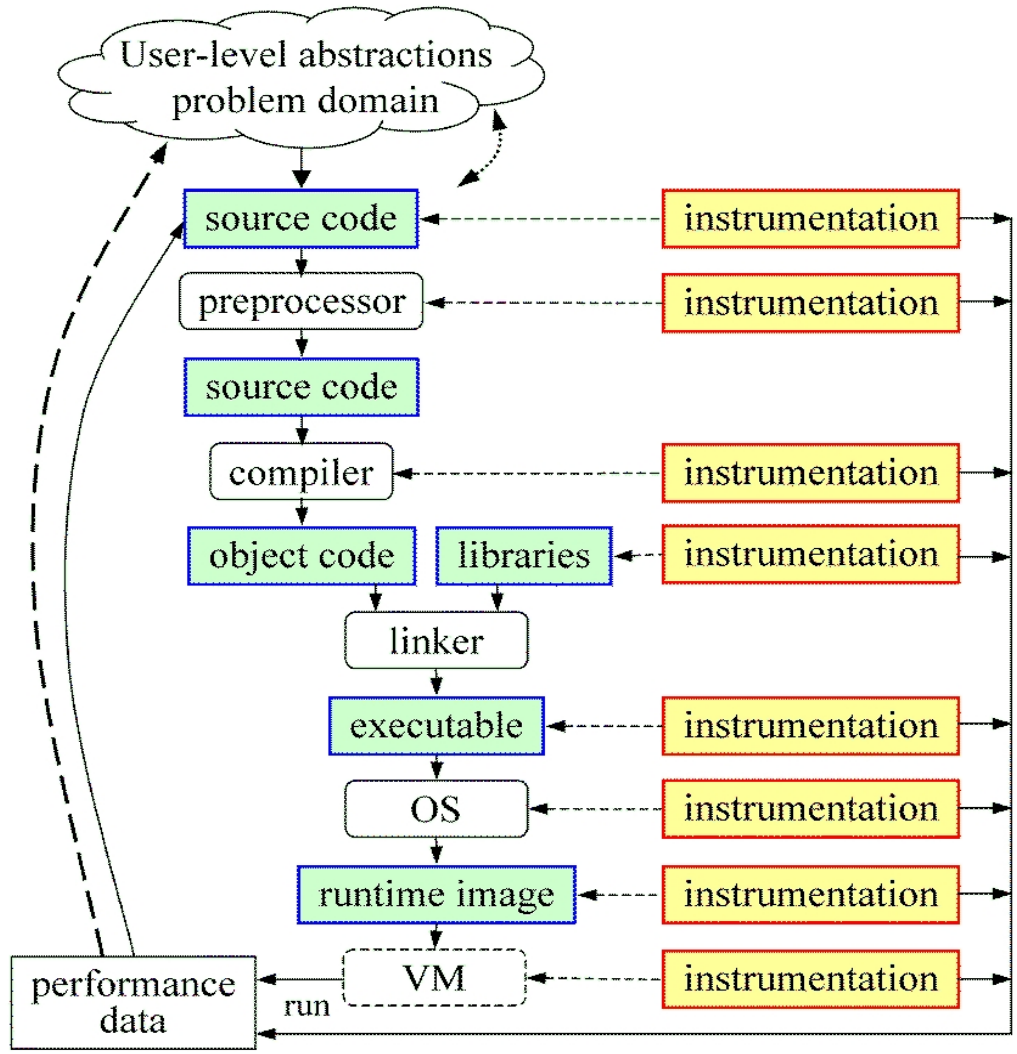
- Events defined by instrumentation access
- Instrumentation levels
  - Source code
  - Library code
  - Object code
  - Executable code
  - Runtime system
  - Operating system
- Different levels provide different information
- Different tools needed for each level
- Levels can have different granularity

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

Direct Observation: Mapping

- Associate performance data with high-level semantic abstractions
- Abstract events at user-level provide semantic context



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

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)
    - interrupts, hardware counter overflow, …
  - Internal agent (direct, synchronous)
    - through code modification
- 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

Measured Performance

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

Critical issues

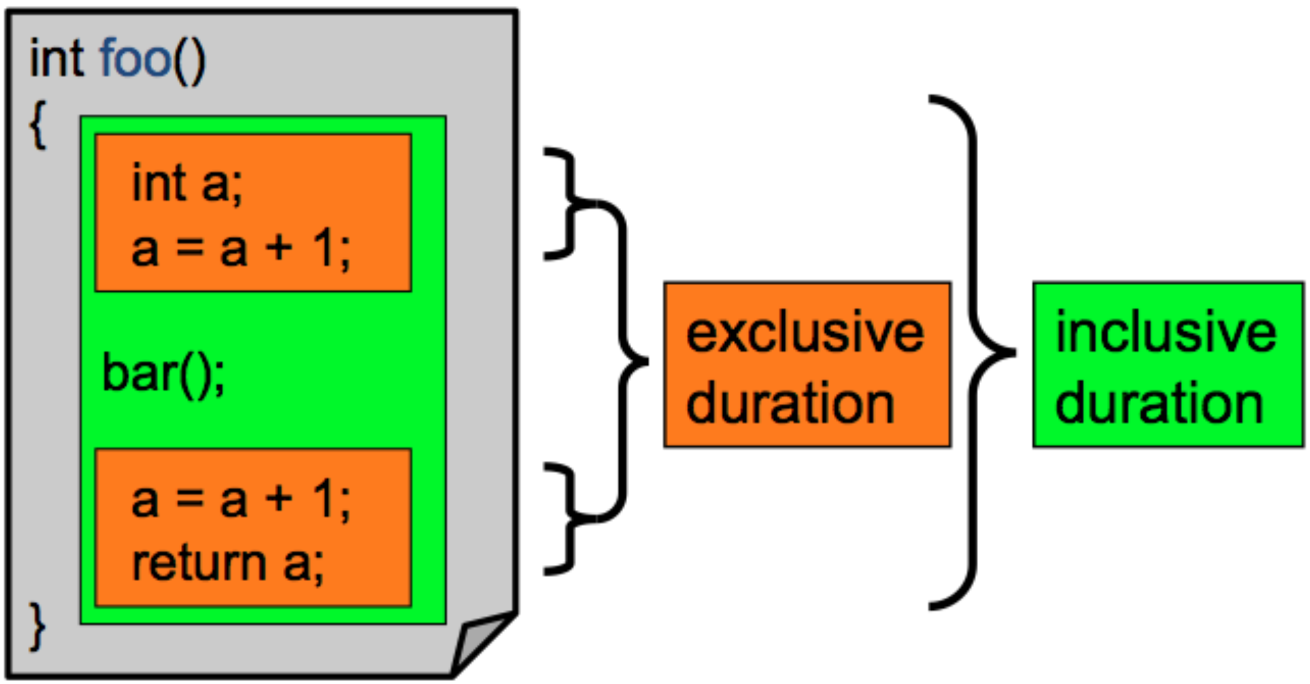
- Accuracy
  - Timing and counting accuracy depends on resolution
  - Any performance measurement generates overhead
    - Execution on performance measurement code
  - Measurement overhead can lead to intrusion
  - Intrusion can cause perturbation
    - alters program behavior
- Granularity
  - How many measurements are made
  - How much overhead per measurement
- Tradeoff (general wisdom)
  - Accuracy is inversely correlated with granularity

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

Tracing Measurement

Process A:

```
void master {
  trace(ENTER, 1);
  ...
  trace(SEND, B);
  send(B, tag, buf);
  ...
  trace(EXIT, 1);
}
```

Process B:

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

MONITOR

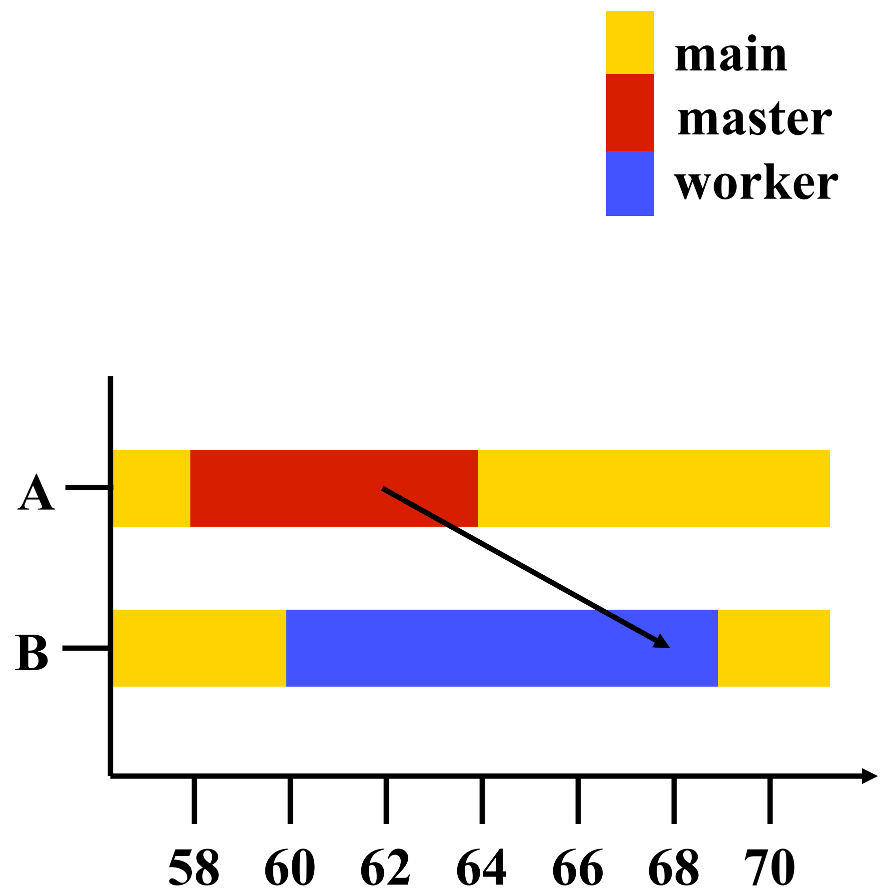
1	master
2	worker
3	...

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

Tracing Analysis and Visualization

1	master
2	worker
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 Formats

- Different tools produce different formats
  - Differ by event types supported
  - Differ by ASCII and binary representations
    - Vampir Trace Format (VTF)
    - KOJAK (EPILOG)
    - Jumpshot (SLOG-2)
    - Paraver
- Open Trace Format (OTF)
  - Supports interoperation between tracing tools

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

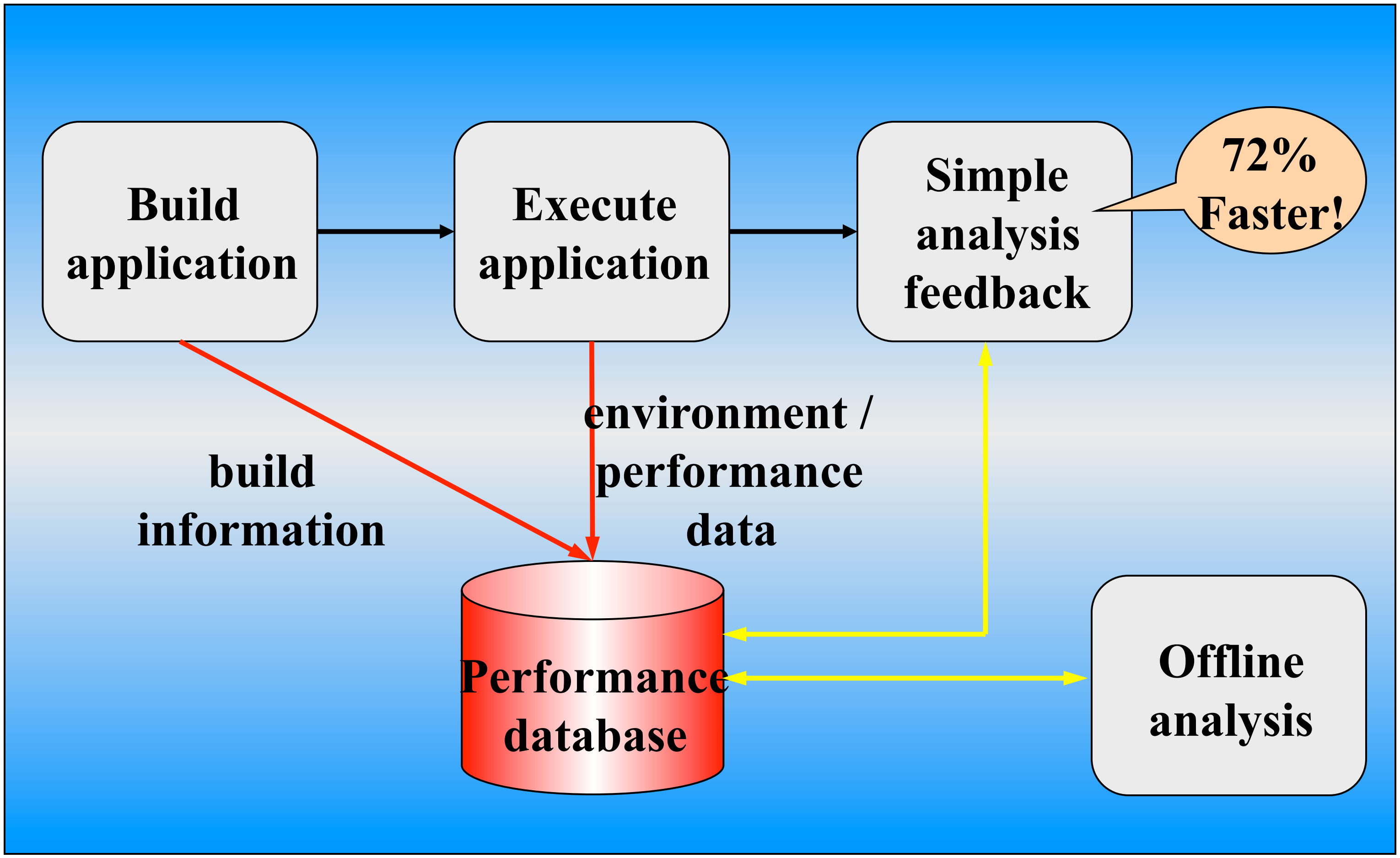
Performance Problem Solving Goals

- Answer questions at multiple levels of interest
  - High-level performance data spanning dimensions
    - machine, applications, code revisions, data sets
    - examine broad performance trends
  - Data from low-level measurements
    - use to predict application performance
- Discover general correlations
  - performance and features of external environment
  - Identify primary performance factors
- Benchmarking analysis for application prediction
- Workload analysis for machine assessment

Performance Analysis Questions

- How does performance vary with different compilers?
- Is poor performance correlated with certain OS features?
- Has a recent change caused unanticipated performance?
- How does performance vary with MPI variants?
- Why is one application version faster than another?
- What is the reason for the observed scaling behavior?
- Did two runs exhibit similar performance?
- How are performance data related to application events?
- Which machines will run my code the fastest and why?
- Which benchmarks predict my code performance best?

Automatic Performance Analysis



Performance Data Management

- Performance diagnosis and optimization involves multiple performance experiments
- Support for common performance data management tasks augments tool use
  - Performance experiment data and metadata storage
  - Performance database and query
- What type of performance data should be stored?
  - Parallel profiles or parallel traces
  - Storage size will dictate
  - Experiment metadata helps in meta analysis tasks
- Serves tool integration objectives

Metadata Collection

- Integration of metadata with each parallel profile
  - Separate information from performance data
- Three ways to incorporate metadata
  - Measured hardware/system information
    - CPU speed, memory in GB, MPI node IDs, …
  - Application instrumentation (application-specific)
    - Application parameters, input data, domain decomposition
    - Capture arbitrary name/value pair and save with experiment
  - Data management tools can read additional metadata
    - Compiler flags, submission scripts, input files, …
    - Before or after execution
- Enhances analysis capabilities

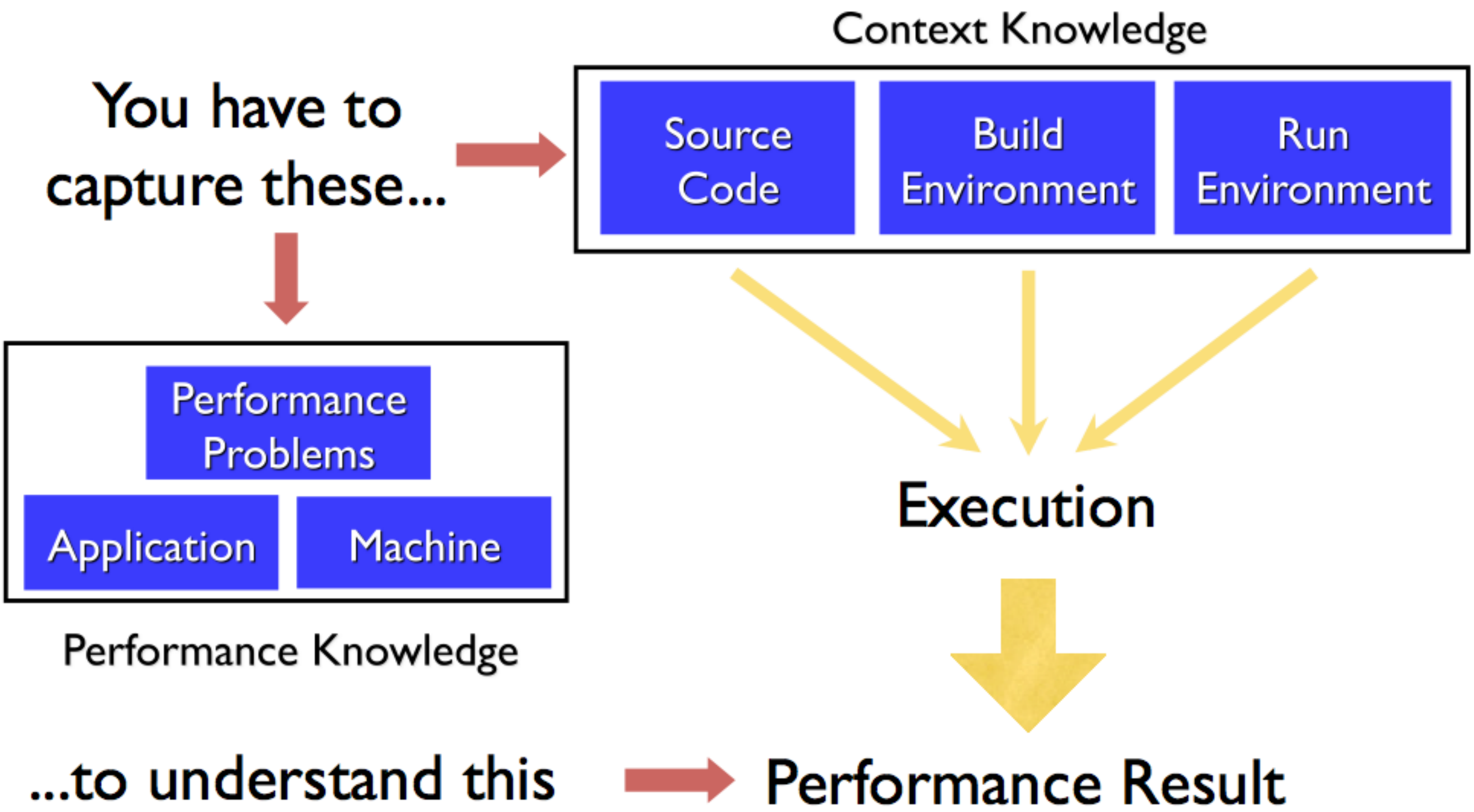
Performance Data Mining

- Conduct parallel performance analysis in a systematic, collaborative and reusable manner
  - Manage performance complexity and automate process
  - Discover performance relationship and properties
  - Multi-experiment performance analysis
- Data mining applied to parallel performance data
  - Comparative, clustering, correlation, characterization, …
  - Large-scale performance data reduction
- Implement extensible analysis framework
  - Abstraction / automation of data mining operations
  - Interface to existing analysis and data mining tools

How to explain performance?

- Should not just redescribe performance results
- Should explain performance phenomena
  - What are the causes for performance observed?
  - What are the factors and how do they interrelate?
  - Performance analytics, forensics, and decision support
- Add knowledge to do more intelligent things
  - Automated analysis needs good informed feedback
  - Performance model generation requires interpretation
- Performance knowledge discovery framework
  - Integrating meta-information
  - Knowledge-based performance problem solving

Metadata and Knowledge Role



Performance Optimization Process

- Performance characterization
  - Identify major performance contributors
  - Identify sources of performance inefficiency
  - Utilize timing and hardware measures
- Performance diagnosis (Performance Debugging)
  - Look for conditions of performance problems
  - Determine if conditions are met and their severity
  - What and where are the performance bottlenecks
- Performance tuning
  - Focus on dominant performance contributors
  - Eliminate main performance bottlenecks

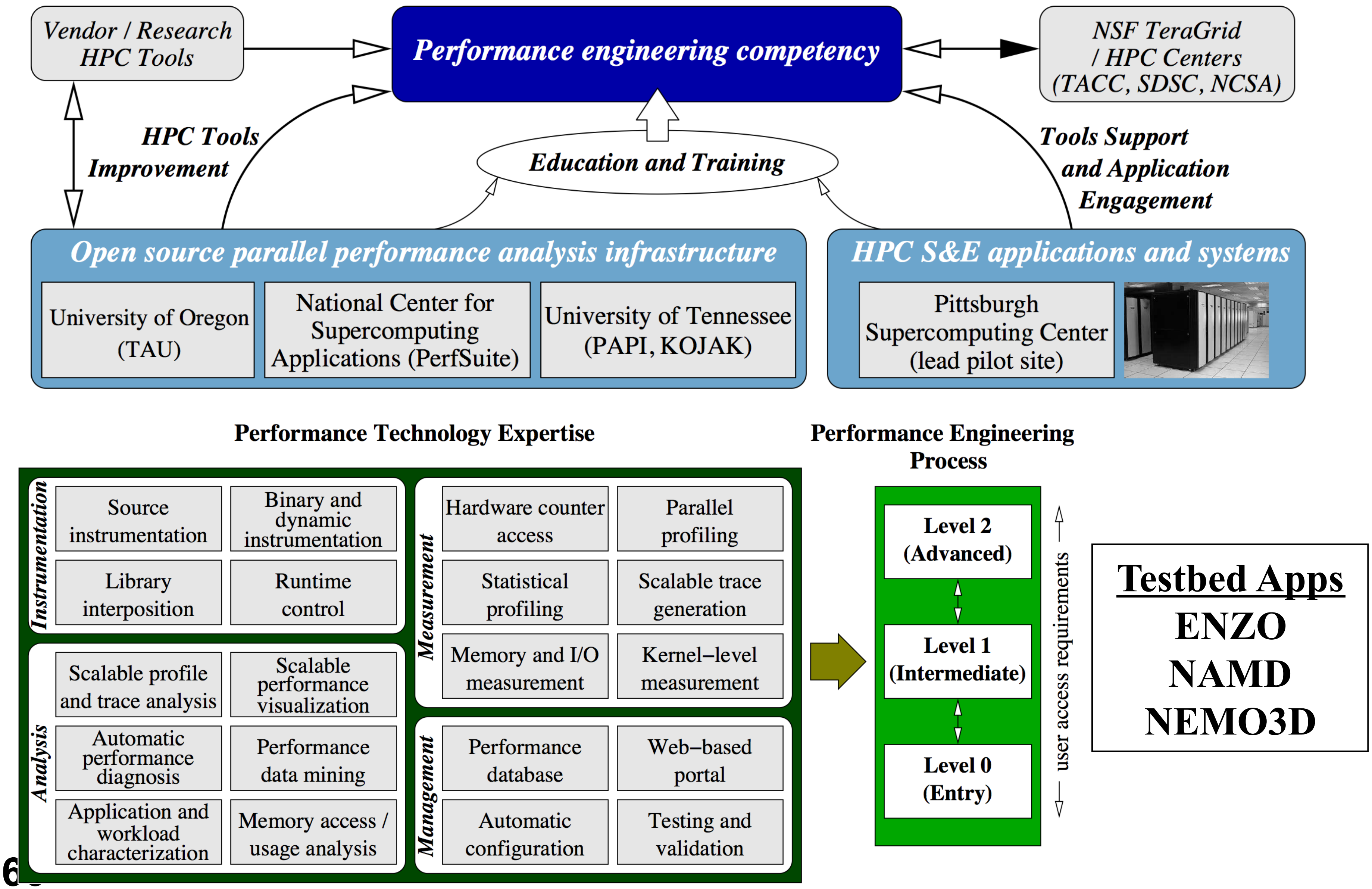
POINT Project

- "High-Productivity Performance Engineering (Tools, Methods, Training) for NSF HPC Applications"
  - NSF SDCI, Software Improvement and Support
  - University of Oregon, University of Tennessee, National Center for Supercomputing Applications, Pittsburgh Supercomputing Center
- POINT project
  - Petascale Productivity from Open, Integrated Tools
  - http://www.nic.uoregon.edu/point

Motivation

- Promise of HPC through scalable scientific and engineering applications
- Performance optimization through effective performance engineering methods
  - Performance analysis / tuning “best practices”
- Productive petascale HPC will require
  - Robust parallel performance tools
  - Training good performance problem solvers

POINT Project Organization



Parallel Performance Technology

- PAPI
  - University of Tennessee, Knoxville
- PerfSuite
  - National Center for Supercomputing Applications
- TAU Performance System
  - University of Oregon
- Kojak / Scalasca
  - Research Centre Juelich
- Vampir and VampirTrace
  - T.U. Dresden

Next Class

- Parallel performance tools
- TAU Performance System