AMS 250: An Introduction to High Performance Computing

Parallel Performance Tools

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

University of California, Santa Cruz

Outline

- Introduction to Parallel Performance Analysis and Tuning
- Performance Observation
- Performance Metrics and Measurement
  - Profiling
  - Tracing
- Performance Technologies
  - Timers
  - Counters
  - Instrumentation
- Performance Tools

Parallel Performance and Complexity

- To use a scalable parallel computer well, you must write high-performance parallel programs
- To write 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 a difficult process
- Parallel performance is complex!

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

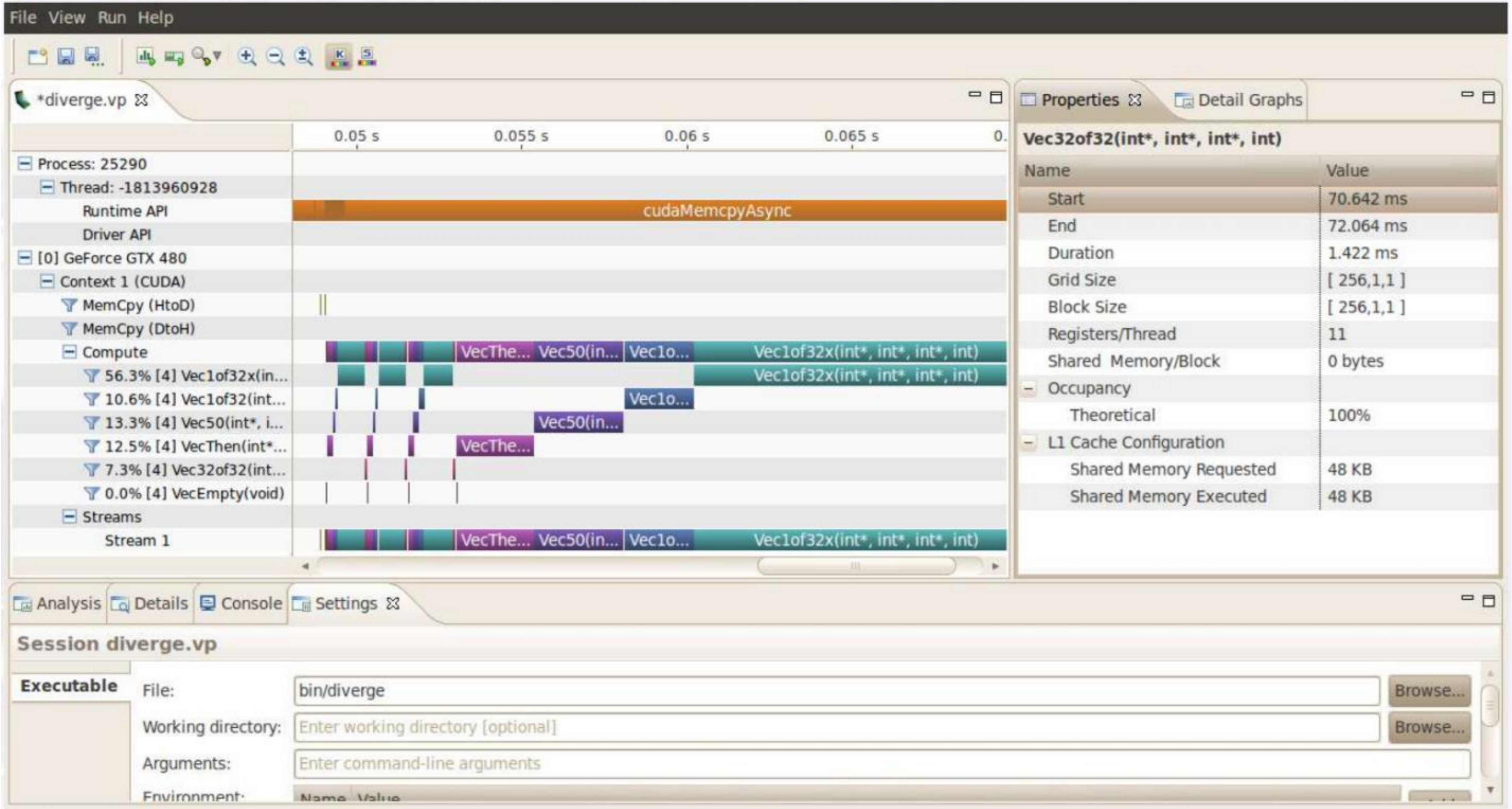
Utilization of Computational Resources

- 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 on 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
  - Efficiency is a relationship of execution time
  - To optimize efficiency is an iterative process
- 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

Example of Performance Analysis



Nvidia Visual Profiler

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: 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 can also lead to parallelism inefficiency
    - load imbalances

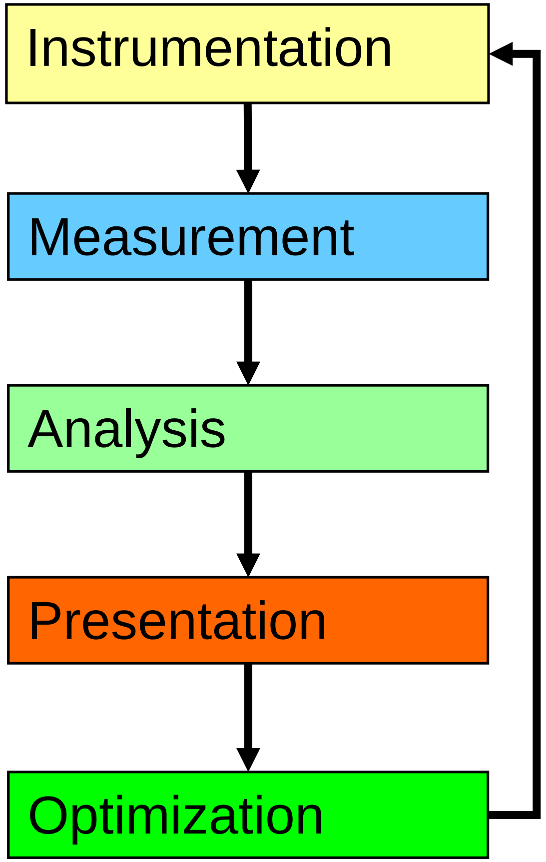
Sequential Performance Tuning

- 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

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 Observability

- 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

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

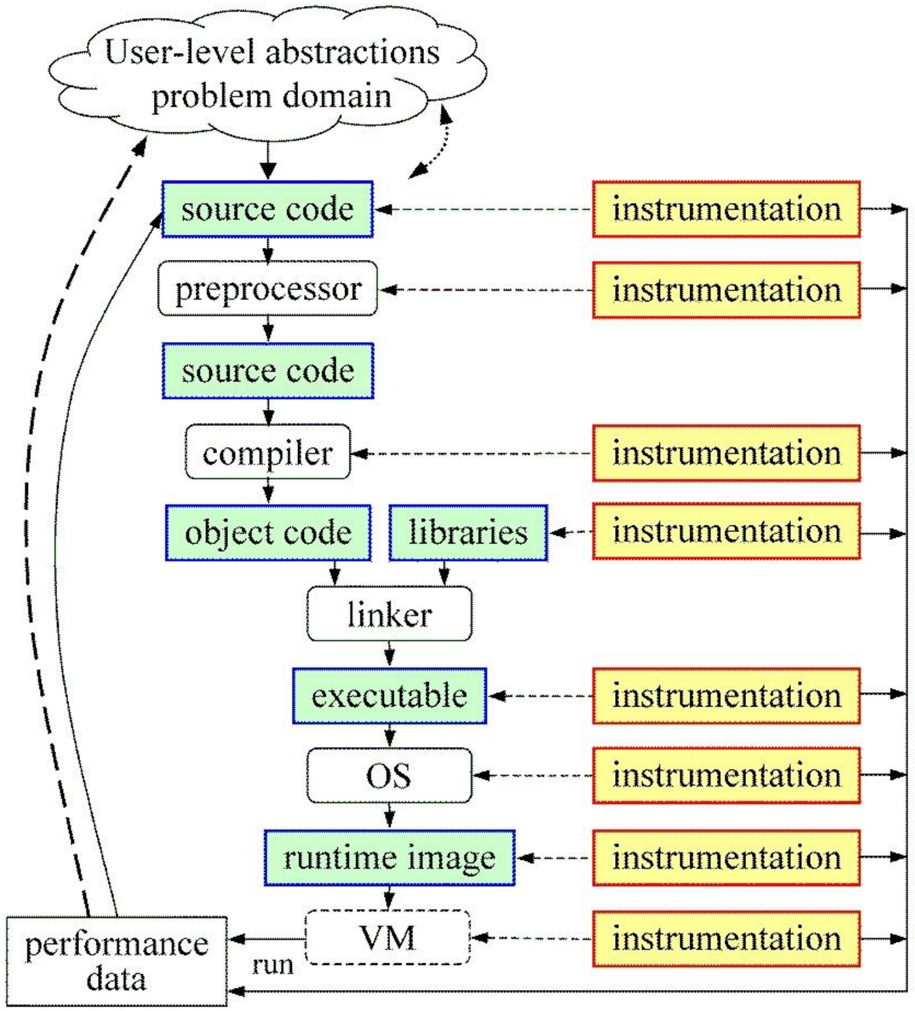
- Program code instrumentation is *not* used
- Performance is observed indirectly
  - Execution is interrupted
    - can be triggered by different events
  - Execution state is queried (sampled)
    - different performance data measured
  - *Event-based sampling* (EBS)
- Performance attribution is inferred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

Indirect Observation: Events/Triggers

- Events are actions external to program code
  - Timer countdown, hardware counter overflow, …
  - Consequence of program execution
  - Event frequency determined by:
    - type, setup, number enabled (exposed)
- Triggers are 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 is determined at time of trap (interrupt)
  - Access to processor 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 Observation 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

Performance Metrics and Measurement

- Observability depends on measurement
- A metric represents a type of measured data
  - *Count*: how often something occurred
    - calls to a routine, cache misses, messages sent, …
  - *Duration*: how long something took place
    - execution time of a routine, message communication time, …
  - Size: how big something is
    - message size, memory allocated, …
- A measurement records performance data
- Certain quantities can not be measured directly
  - *Derived metric*: calculated from metrics
    - flops per second, ...

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

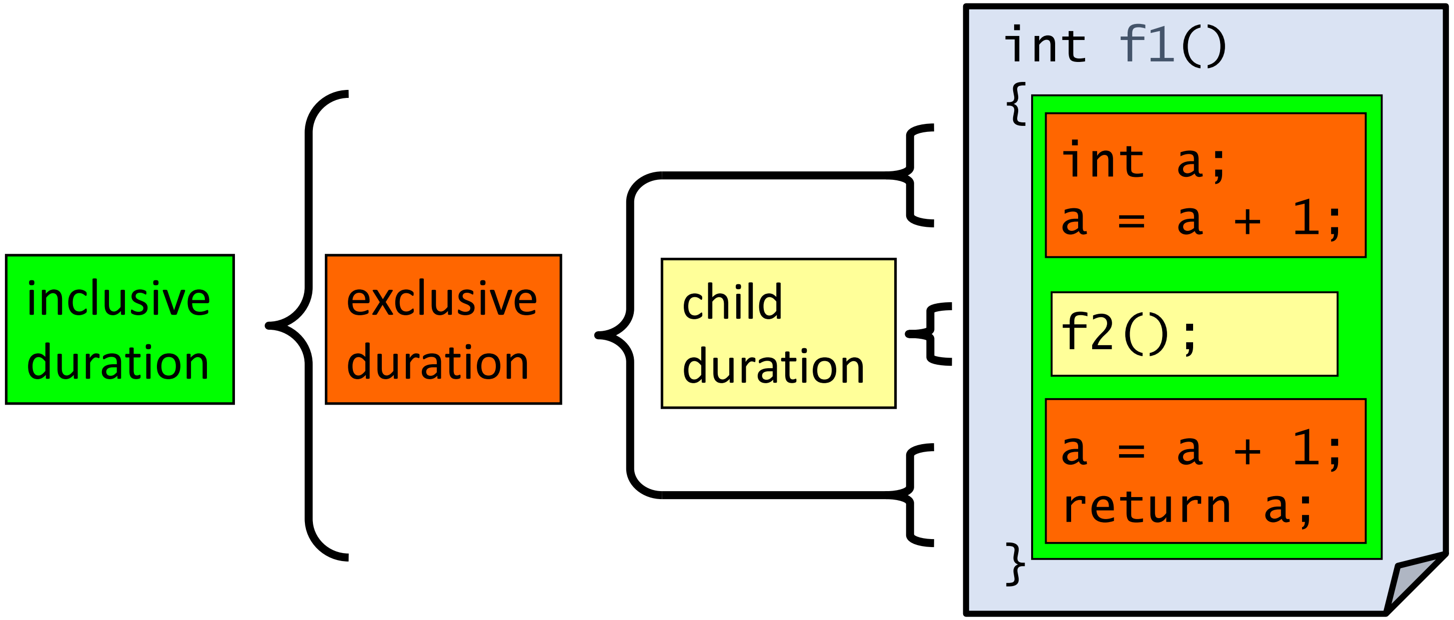
- Accuracy
  - Timing and counting accuracy depends on resolution
  - Any performance measurement generates *overhead*
    - execution on performance measurement code
  - Measurement overhead can lead to *intrusion*
  - Intrusion can cause *perturbation*
    - alters program behavior
- Granularity
 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

Profiling

- Recording of aggregated information
  - Counts, time, …
- Aggregated statistics about program and system entities
  - Functions, loops, basic blocks, …
  - Processes, threads
- Methods
  - Event-based sampling (indirect, statistical)
  - Direct measurement (deterministic)
- Serial example: GNU gprof
  - https://sourceware.org/binutils/docs/gprof/
  - Tutorial: http://www.thegeekstuff.com/2012/08/gprof-tutorial/

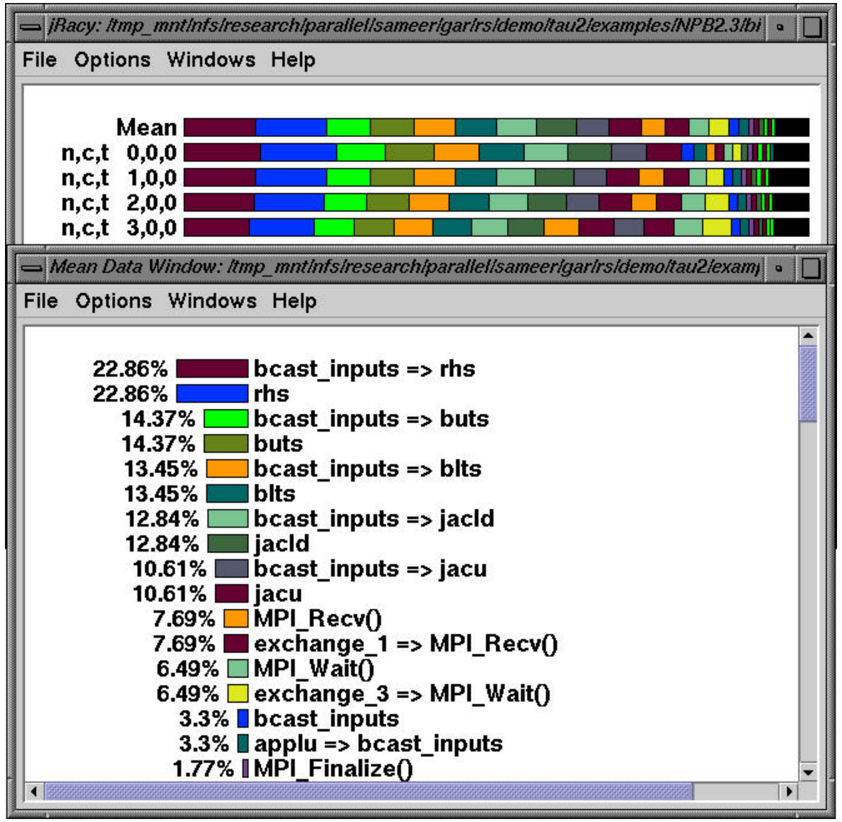
Inclusive and Exclusive Profiles

- 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

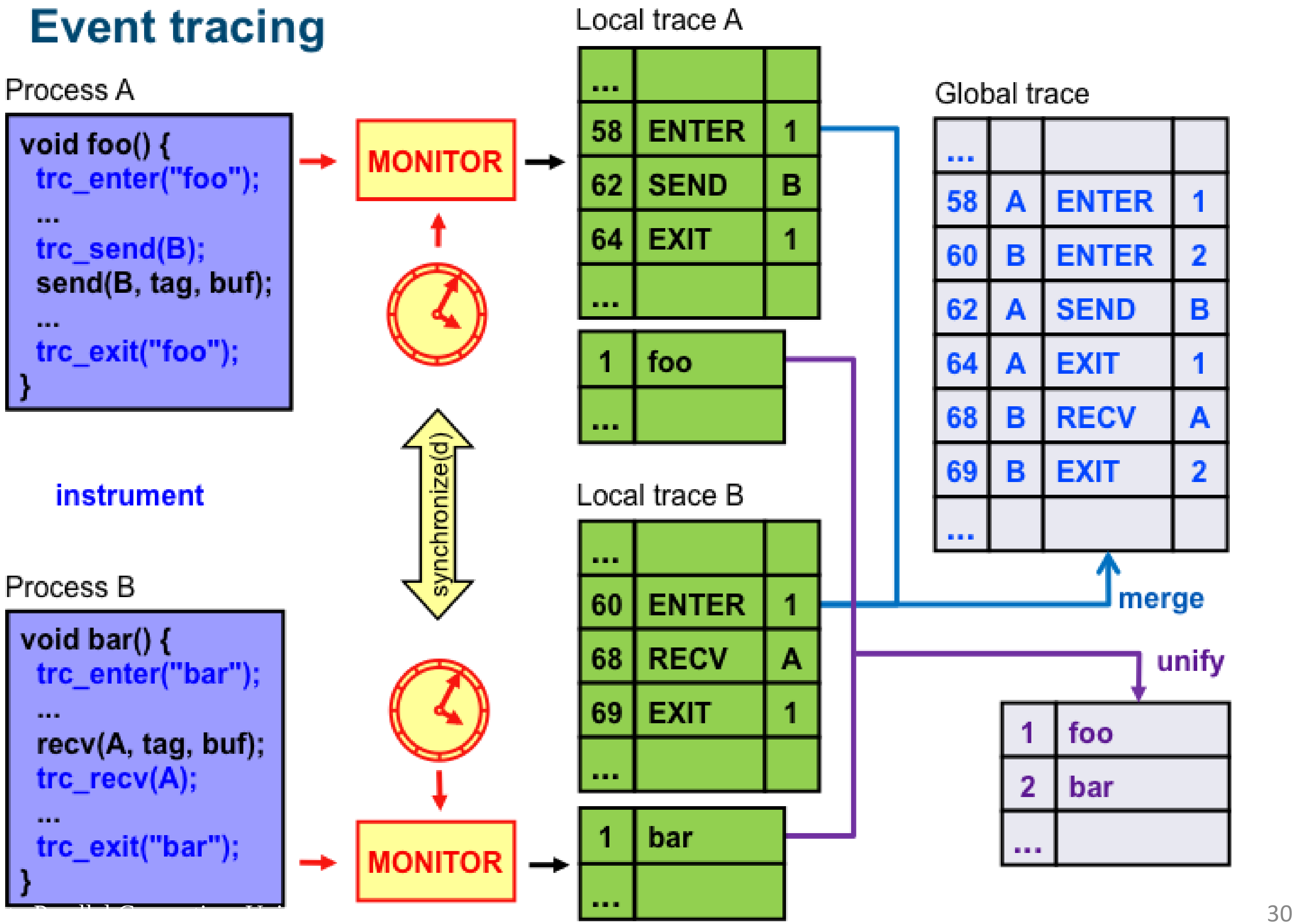


Tracing

- 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* is a stream of event records sorted by time
- Main advantage is that it can be used to reconstruct the dynamic behavior of the parallel execution
- Serial examples: Linux tracers

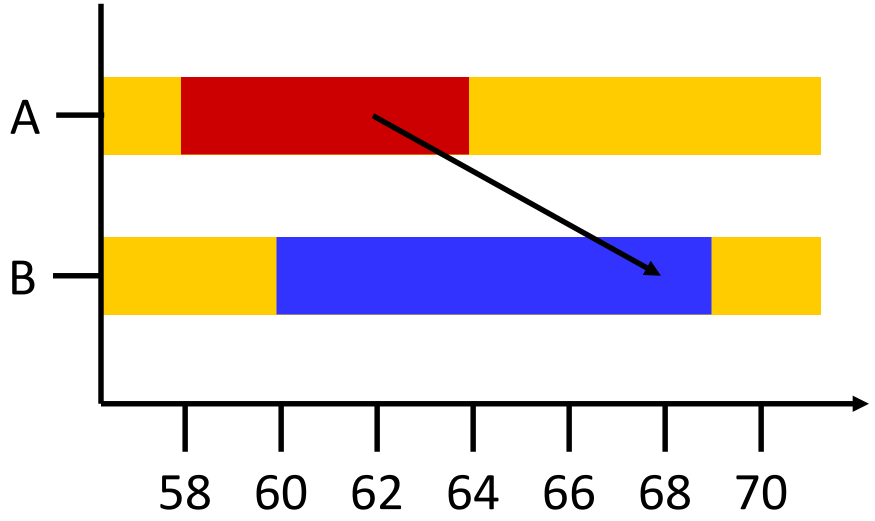
  http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html

Event Tracing



Tracing: Timeline Visualization

...			
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
  - https://www.vampir.eu/
  - *VTF*: family of historical ASCII and binary formats
- MPICH / JumpShot
  - http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/
  - *ALOG, CLOG, SLOG, SLOG-2*
- Scalasca
  - http://www.scalasca.org/
  - *EPILOG* (Jülich open-source trace format)
- Paraver
  - http://www.bsc.es/computer-sciences/performance-tools/paraver
- TAU Performance System
  - http://www.cs.uoregon.edu/research/tau/home.php
- Convergence on *Open Trace Format (OTF)*
  - http://www.paratools.com/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

Performance Technologies

- Timers
- Counters
- Instrumentation
  - Source level
  - Library wrapping (PMPI)
  - Compiler instrumentation
  - Binary (Dyninst, PEBIL, MAQAO)
  - Runtime Interfaces
- Program address resolution
- Stack Walking
- Heterogeneous (accelerator) timers and counters

Time

- How is time measured in a computer system?
- How do we derive time from a clock?
- What clock/time technologies are available to a measurement system?
- How are clocks synchronized in a parallel computer in order to provide a “global time” common between nodes?
- Different technologies are available
  - Issues of resolution and accuracy

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
  - 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

Timer: gettimeofday()

- UNIX function
- Returns wall-clock time in seconds and microseconds
- Actual resolution is hardware-dependent
- Base value is 00:00 UTC, January 1, 1970
- Some implementations also return the timezone

```
#include <sys/time.h>

struct timeval tv;
double walltime;  /* seconds */

gettimeofday(&tv, NULL);
walltime = tv.tv_sec + tv.tv_usec * 1.0e-6;
```

Timer: clock_gettime()

- POSIX function
- For *clock_id* **CLOCK_REALTIME** it returns wall-clock time in seconds and nanoseconds
- More clocks may be implemented but are not standardized
- Actual resolution is hardware-dependent

```
#include <time.h>

struct timespec tv;
double walltime;  /* seconds */

clock_gettime(CLOCK_REALTIME, &tv);
walltime = tv.tv_sec + tv.tv_nsec * 1.0e-9;
```

Timer: getrusage()

- UNIX function
- Provides a variety of different information
  - Including user time, system time, memory usage, page faults, and other *resource use* information
  - Information provided system-dependent!

```
#include <sys/resource.h>

struct rusage ru;
double usrtime;  /* seconds */
int memused;

getrusage(RUSAGE_SELF, &ru);
usrtime = ru.ru_utime.tv_sec +
          ru.ru_utime.tv_usec * 1.0e-6;
memused = ru.ru_maxrss;
```

Timer: MPI & OpenMP

- MPI provides portable MPI wall-clock timer

```
#include <mpi.h>
double walltime;   /* seconds */

walltime = MPI_Wtime();
```

  - Not required to be consistent/synchronized across ranks!
- OpenMP 2.0 also provides a library function

```
#include <omp.h>
double walltime;   /* seconds */

walltime = omp_get_wtime();
```

- Hybrid MPI/OpenMP programming?
  - Interactions between both standards (yet) undefined

Timer: Others

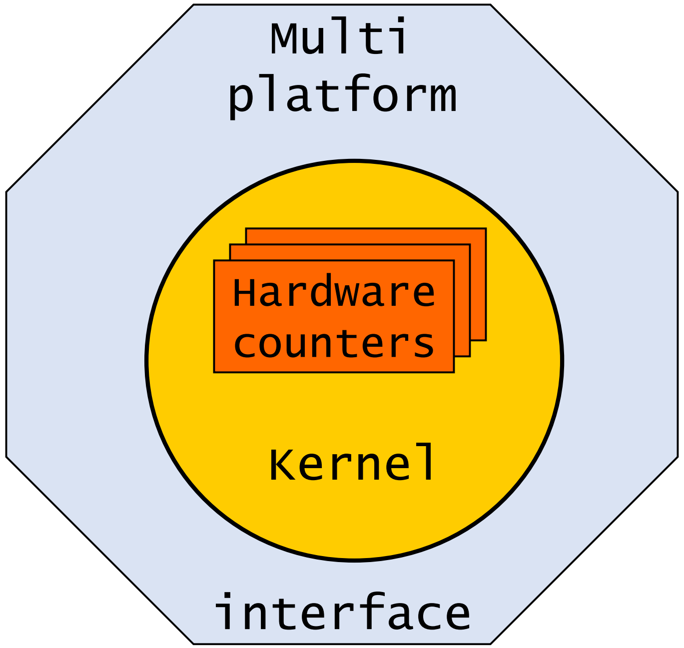
- Fortran 90 intrinsic subroutines
  - cpu_time()
  - system_clock()
- Hardware counter libraries typically provide “timers” because underlying them are cycle counters
  - Vendor APIs
    - PMAPI, HWPC, libhpm, libpfm, libperf, …
  - PAPI (Performance API)
    - http://icl.cs.utk.edu/papi/

Performance Counters

- Extra **processor logic** inserted to count specific events
- Updated at every cycle (or when some event occurs)
- Strengths
  - Non-intrusive
  - Very accurate
  - Low overhead
- Weaknesses
  - Provides only hard counts
  - Specific for each processor
  - Access is not appropriate for the end user
    - nor is it well documented
  - Lack of standard on what is counted

Hardware Counters Interfaces

- Kernel level
  - Handling of overflows
  - Thread accumulation
  - Thread migration
  - State inheritance
  - Multiplexing
  - Overhead
  - Atomicity
- Multi-platform interfaces
  - Performance API (*PAPI*)
    - University of Tennessee, USA
    - http://icl.cs.utk.edu/papi/
  - Lightweight Performance Tools (*LIKWID*)
    - University of Erlangen, Germany
    - https://github.com/RRZE-HPC/likwid



Hardware Measurement

Typical measured events account for:

- Functional units status
  - float point operations
  - fixed point operations
  - load/stores
- Access to memory hierarchy
- Cache coherence protocol events
- Cycles and instructions counts
- Speculative execution information
  - instructions dispatched
  - branches mispredicted

Hardware Metrics

Typical hardware counter	**Useful derived metrics**
Cycles / Instructions	IPC
Floating point instructions	FLOPS
Integer instructions	computation intensity
Load/stores	instructions per load/store
Cache misses	load/stores per cache miss
Cache misses	cache hit rate
Cache misses	loads per load miss
TLB misses	loads per TLB miss

- Derived metrics allow users to correlate the behavior of the application to hardware components
- Define threshold values acceptable for metrics and take actions regarding optimization when below/above thresholds

Hardware Counters Access on Linux

- perf
  - Linux profiling with performance counters: https://perf.wiki.kernel.org/index.php/Main_Page
  - Available from Linux kernel 2.6.31 and newer
  - Comprised of performance counters subsystem in kernel and userspace utility
  - Can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing)
  - Capable of statistical profiling of the entire system (both kernel and userland code)
- Intel Performance Counter Monitor
  - https://software.intel.com/en-us/articles/intel-performance-counter-monitor

PAPI – Performance API

- Middleware to provide a consistent and portable API for the performance counter hardware in microprocessors
- Countable events are defined in two ways:
  - Platform-neutral *preset* events
  - Platform-dependent *native* events
- Presets can be derived from multiple native events
- Two interfaces to the underlying counter hardware:
  - *High-level* interface simply provides the ability to start, stop and read the counters for a specified list of events
  - Low-level interface manages hardware events in user defined groups called *EventSets*
- Events can be multiplexed if counters are limited

http://icl.cs.utk.edu/papi/

PAPI High Level API

- Meant for application programmers wanting simple but accurate measurements
- Calls the lower level API
- Allows only PAPI preset events
- 10 functions:
  - *PAPI_accum_counters*
  - *PAPI_num_counters*
  - *PAPI_num_components*
  - *PAPI_start_counters, PAPI_stop_counters*
  - *PAPI_read_counters*
  - *PAPI_flips, PAPI_flops*
  - *PAPI_ipc, PAPI_epc*

http://icl.cs.utk.edu/papi/docs/db/d93/group__high__api.html

PAPI Low Level API

- Increased efficiency and functionality over the high level PAPI interface
- Access to native events
- Obtain information about the executable, the hardware, and memory
- Set options for multiplexing and overflow handling
- System V style sampling (profil())
- Thread safe

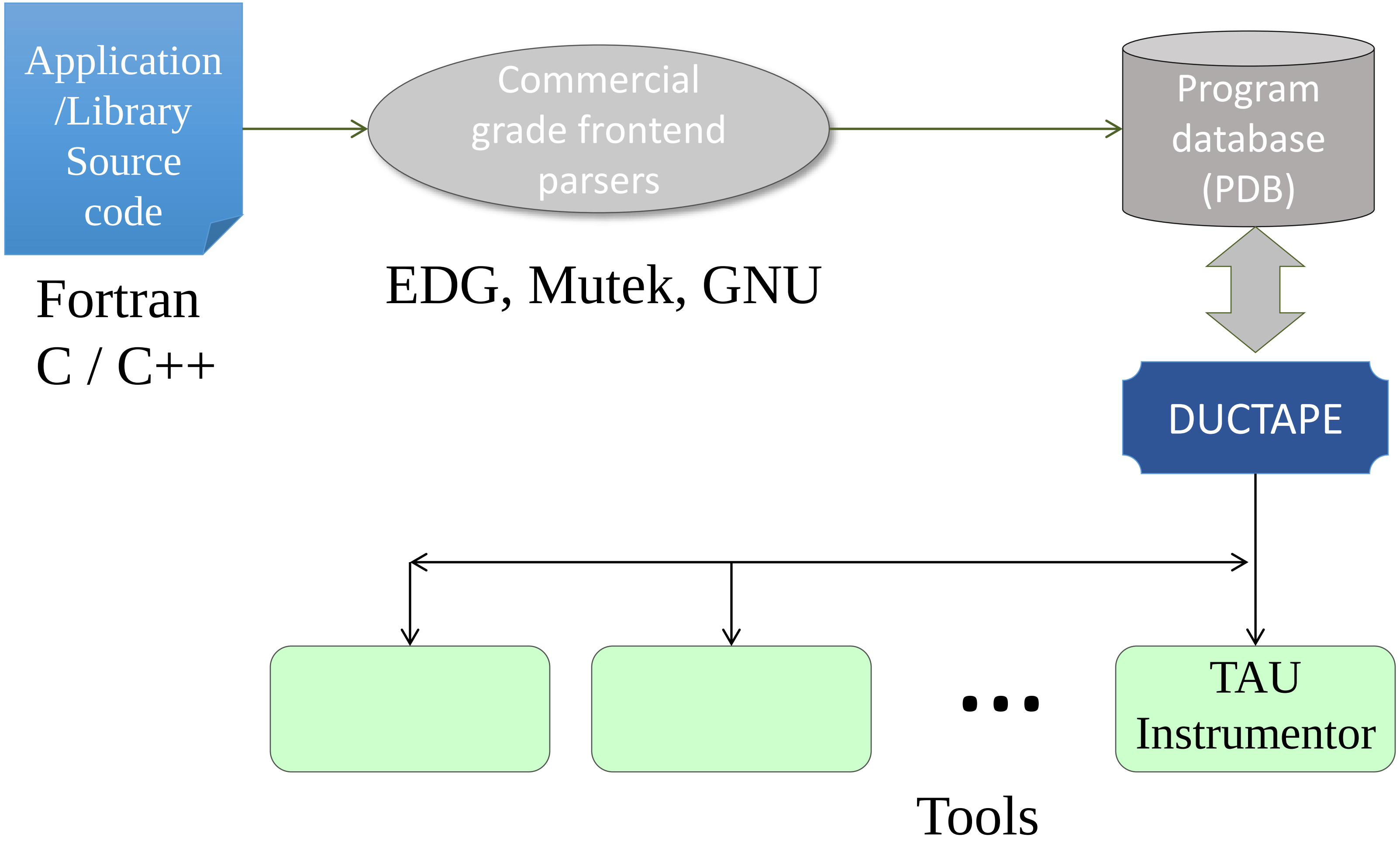
Source Instrumentation with Timers

- Measuring performance using *timers* requires instrumentation
  - Have to uniquely identify code region (name)
  - Have to add code for timer start and stop
  - Have to compute delta and accumulate statistics
- Hand-instrumentation becomes tedious very quickly, even for small software projects
- Also a requirement for enabling instrumentation only when wanted
  - Avoids unnecessary overheads when not needed

Program Database Toolkit (PDT)

- https://www.cs.uoregon.edu/research/pdt/home.php
- Used to automate instrumentation of C/C++, Fortran source code
- Source code parser(s) identify blocks such as function boundaries, loop boundaries, …
- Instrumentor uses parse results to insert API calls into source code files at block enter/exit, outputs an instrumented code file
- Instrumented source passed to compiler
- Linker links application with measurement library

PDT Architecture



PMPI – MPI Standard Profiling Interface

- The **MPI** (Message Passing Interface) standard defines a mechanism for instrumenting all API calls in an MPI implementation
- Each **MPI_*** function call is actually a *weakly defined* interface that can be re-defined by performance tools
- Each **MPI_*** function call eventually calls a corresponding **PMPI_*** function call which provides the expected MPI functionality
- Performance tools can redefine **MPI_*** calls

PMPI Example

- Original MPI_Send() definition:

```
int __attribute__((weak))
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
         int tag, MPI_Comm comm) {
    PMPI_Send(buf, count, datatype, dest, tag, comm);
}
```

- *Possible* Performance tool definition:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm) {
    MYTOOL_Timer_Start("MPI_Send");
    PMPI_Send(buf, count, datatype, dest, tag, comm);
    MYTOOL_Timer_Stop("MPI_Send");
    MYTOOL_Message_Size("MPI_Send", count * sizeof(datatype));
}
```

Compiler Instrumentation

- Modern compilers provide the ability to instrument functions at compile time
- Can exclude files and/or functions
- GCC example:
  - Use the compiler option `-finstrument-functions`
  - Instrument function entry and exit

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

Trace and profile function calls with GCC:
https://balau82.wordpress.com/2010/10/06/trace-and-profile-function-calls-with-gcc/

Compiler Instrumentation – Tool Interface

- Measurement libraries have to implement those two functions:

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

- The function and call site pointers are instruction addresses
- How to resolve those addresses to source code locations?
  - Binutils: libbfd, libiberty

Binary Instrumentation

- Source Instrumentation not possible in all cases
  - Exotic / Domain Specific Languages (no parser support)
  - Pre-compiled system libraries
  - Utility libraries without source available
- Binary instrumentation modifies the existing executable and all libraries, adding user-specified function entry/exit API calls
- Can be done once, or as first step of execution

Binary Instrumentation Tools

- **Dyninst API**
  - http://www.dyninst.org/dyninst
  - Dynamic binary instrumentation for runtime code patching
- **PEBIL**
  - http://www.sdsc.edu/pmac/tools/pebil.html
  - Static binary instrumentation for x86/Linux
  - Lightweight binary instrumentation tool that can be used to capture information about the behavior of a running executable
- **MAQAO**
  - http://www.maqao.org/
  - Tool for analyzing and optimizing binary codes
  - Provides an API to insert user code at any point of the binary

Performance Tools

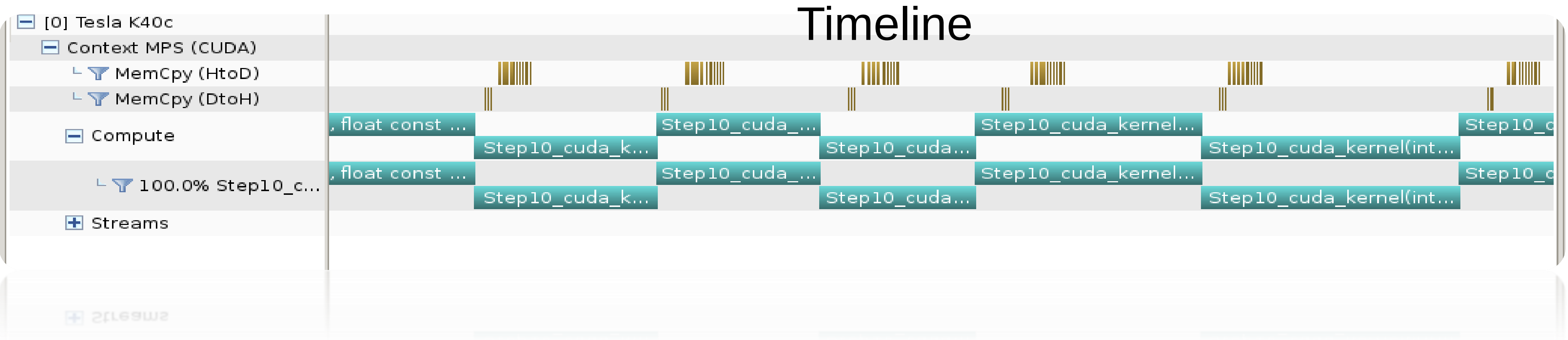
- **nvprof**: http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
- **Nvidia Visual Profiler**: http://docs.nvidia.com/cuda/profiler-users-guide/#visual
- **Intel Vtune Amplifier**
- **Intel Trace Analyzer and Collector**
- **Open|SpeedShop**: https://openspeedshop.org/
- **HPCToolkit**: http://hpctoolkit.org/
- **Vampir**: https://www.vampir.eu/
- **Scalasca**: http://www.scalasca.org/
- **TAU**: https://www.cs.uoregon.edu/research/tau/home.php
- **Periscope Tuning Framework**: http://periscope.in.tum.de/
- **mpiP**: http://mpip.sourceforge.net/
- **Paraver**: http://www.bsc.es/computer-sciences/performance-tools/paraver/
- **PerfExpert**: https://www.tacc.utexas.edu/research-development/tacc-projects/perfexpert

nvprof

- **nvprof** is the command line profiling tool provided with the CUDA Toolkit
  http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
- **nvprof** enables the collection of a timeline of CUDA-related activities on both CPU and GPU, including:
  - kernel execution
  - memory transfers
  - memory set
  - CUDA API calls and events or metrics for CUDA kernels
- No need to recompile!
- Profiling results are displayed in the console after the profiling data is collected, and may also be saved for later viewing by either nvprof or the Visual Profiler.

Nvidia Visual Profiler

- Nvidia Visual Profiler (nvvp) displays a timeline of the application's activity on both CPU and GPU
  http://docs.nvidia.com/cuda/profiler-users-guide/#visual



- Nvidia Visual Profiler does not require any application changes; however, by making some simple modifications and additions, you can greatly increase its usability and effectiveness.

Intel Vtune Amplifier

- A commercial **Performance Profiler** for serial and multithreaded applications
  - GUI: **amplxe-gui**
  - CLI: **amplxe-cl**
- Use Vtune Amplifier to locate or determine the following:
  - The most time-consuming (*hot*) functions in your application and/or on the whole system
  - Sections of code that do not effectively utilize available processor time
  - The best sections of code to optimize for sequential performance and for threaded performance
  - Synchronization objects that affect the application performance
  - Whether, where, and why your application spends time on input/output operations
  - The performance impact of different synchronization methods, different numbers of threads, or different algorithms
  - Thread activity and transitions
  - Hardware-related issues in your code such as data sharing, cache misses, branch misprediction, and others
- Tutorials:
  - https://software.intel.com/en-us/articles/intel-vtune-amplifier-tutorials
  - Using VTune on Edison and Cori

Intel Trace Analyzer and Collector

- Intel **Trace Analyzer and Collector** is a graphical tool for understanding MPI application behavior, quickly finding bottlenecks, improving correctness, and achieving high performance for parallel cluster applications.
- Trace Collector
  - intercepts all MPI calls and generates tracefiles (*.stf*)
  - can also trace non-MPI applications, like socket communication in distributed applications or serial programs
  - formerly known as Vampirtrace (VT)
- Trace Analyzer
  - GUI tool that analyzes the tracefiles (*.stf*): **traceanalyzer**
- Documentation:
  - https://software.intel.com/en-us/articles/intel-trace-analyzer-and-collector-documentation
  - Using Intel Trace Analyzer and Collector on Edison and Cori

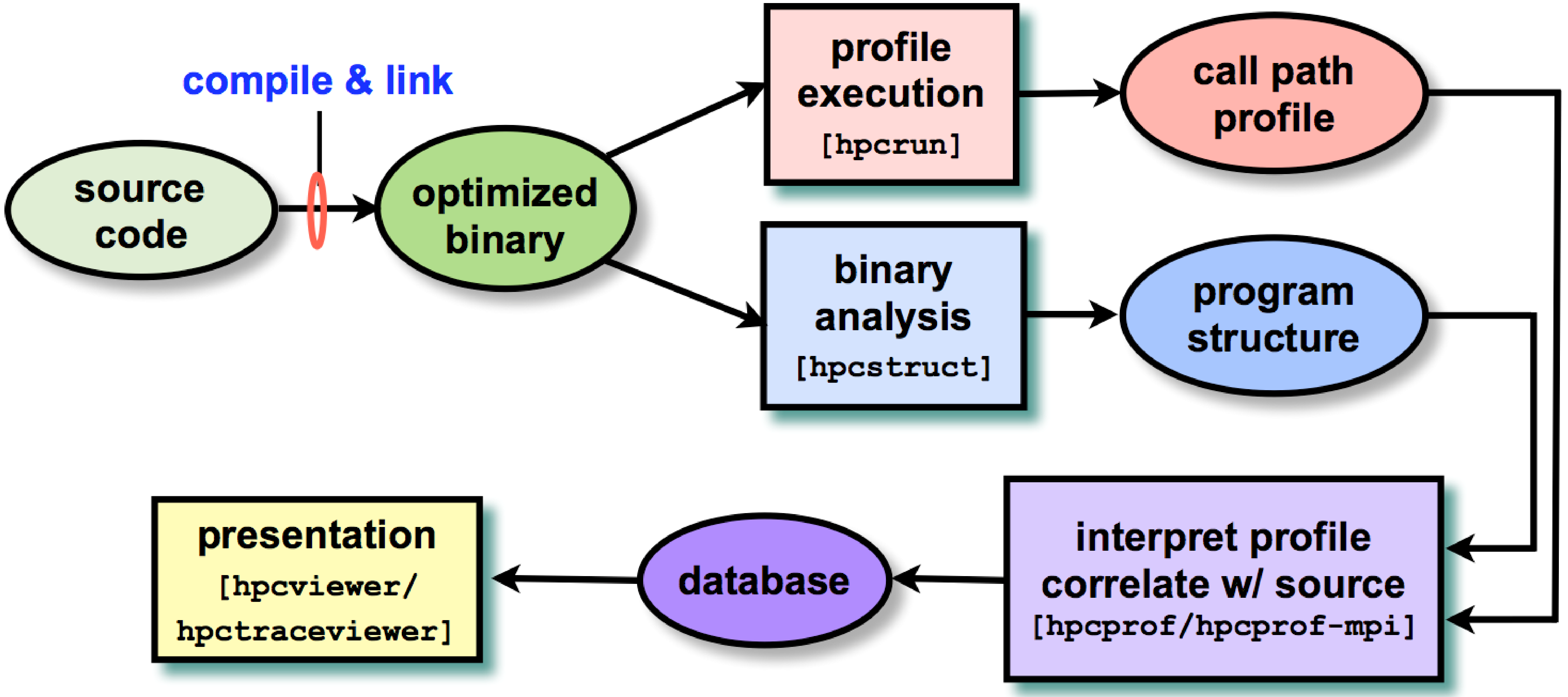
Open|SpeedShop

- https://openspeedshop.org/
- Base functionality include:
  - Program Counter Sampling
  - Support for Callstack Analysis
  - Hardware Performance Counter Sampling and Threshold based
  - MPI Lightweight Profiling and Tracing
  - I/O Lightweight Profiling and Tracing
  - Floating Point Exception Analysis
  - Memory Trace Analysis
  - POSIX Thread Trace Analysis
- Tutorials: https://openspeedshop.org/category/tutorials/

HPCToolkit

- http://hpctoolkit.org/
- Integrated suite of tools for measurement and analysis of program performance
- Uses statistical sampling of timers and hardware performance counters
- Works with multilingual, fully optimized applications that are statically or dynamically linked
- Supports measurement and analysis of serial codes, threaded codes (e.g., pthreads, OpenMP), MPI, and hybrid (MPI+threads) parallel codes
- Documentation: http://hpctoolkit.org/documentation.html

HPCToolkit Workflow



Vampir

- https://www.vampir.eu/
- Mission: Visualization of dynamics of complex parallel processes
- Requires two components
  - Monitor/Collector (**Score-P**)
  - Charts/Browser (**Vampir**)
- Typical questions that Vampir helps to answer:
  - What happens in my application execution during a given time in a given process or thread?
  - How do the communication patterns of my application execute on a real system?
  - Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

https://www.alcf.anl.gov/files/Vampir.pdf

Scalasca

- http://www.scalasca.org/
- Scalable parallel performance analysis toolset
  - Focus on communication and synchronization
- Integrated performance analysis process
  - Callpath profiling
  - Event tracing
- Supported programming models
  - MPI-1, MPI-2 one-sided communication
  - OpenMP
  - Hybrid (MPI + OpenMP)
- Available for all major HPC platforms
- Documentation:
  http://www.scalasca.org/software/scalasca-2.x/documentation.html

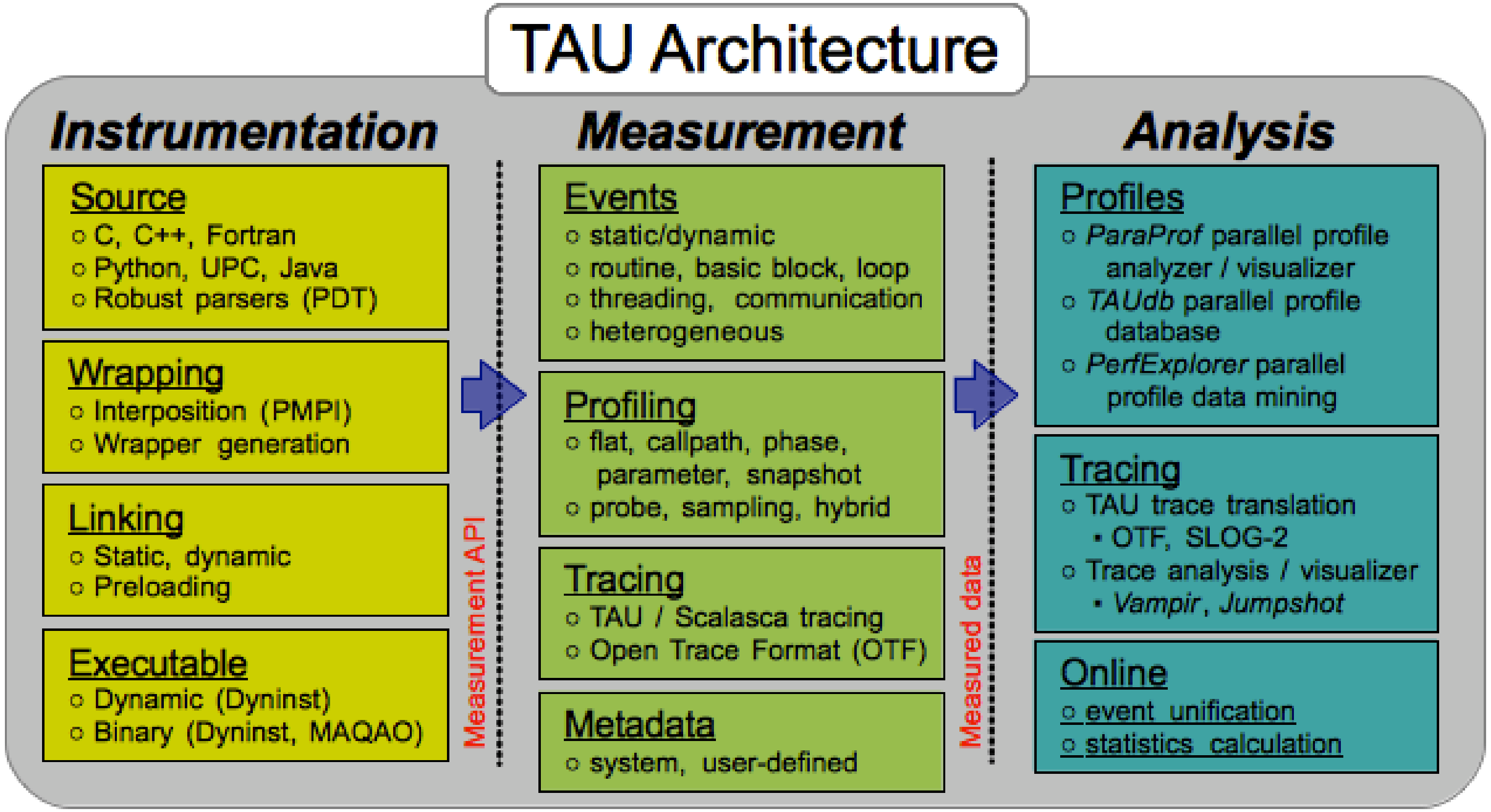
TAU Performance System

- https://www.cs.uoregon.edu/research/tau/home.php
- A portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python
- Capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling
- Tutorial: http://tau.uoregon.edu/tau.ppt
- Documentation: https://www.cs.uoregon.edu/research/tau/docs.php

TAU Architecture

- TAU is a parallel performance framework and toolkit
- Software architecture provides separation of concerns

  Instrumentation | Measurement | Analysis



TAU Components

- Instrumentation
  - Fortran, C, C++, OpenMP, Python, Java, UPC, Chapel
  - Source, compiler, library wrapping, binary rewriting
  - Automatic instrumentation
- Measurement
  - Internode: MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  - Intranode: Pthreads, OpenMP, hybrid, …
  - Heterogeneous: GPU, MIC, CUDA, OpenCL, OpenACC, …
  - Performance data (timing, counters) and metadata
  - Parallel profiling and tracing (with **Score-P** integration)
- Analysis
  - Parallel profile analysis and visualization (**ParaProf**)
  - Performance data mining / machine learning (**PerfExplorer**)
  - Performance database technology (**TAUdb**)
  - Empirical autotuning

Further Readings

- NERSC Performance and Debugging Tools:
  http://www.nersc.gov/users/software/performance-and-debugging-tools/
- More Profiling Tools at NERSC:
  http://www.nersc.gov/assets/Uploads/NUG-hackathon-Profiling-Tools.pdf