3rd natESM Technical Training: Parallel Performance Analysis

Markus Geimer
Jülich Supercomputing Centre

Virtual Institute – High Productivity Supercomputing

- **Goal**: Improve the quality and accelerate the development process of complex simulation codes running on highly-parallel computer systems
- Start-up funding (2006–2011) by Helmholtz Association of German Research Centres
  - Initially 4 partner institutions, meanwhile 15 partners

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

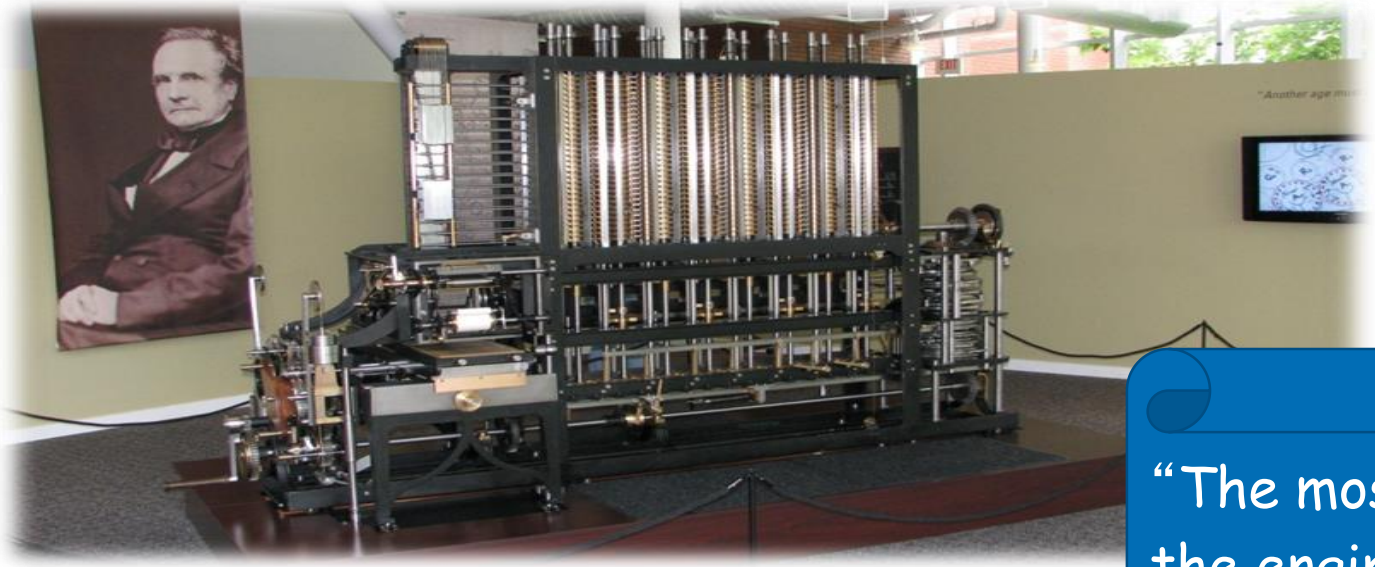
- Activities
  - Development and integration of HPC application tools
    - Primarily correctness checking & performance analysis
  - Academic workshops: e.g. ProTools@SC24 (Monday, November 19, 2024)
  - Tools training via conference tutorials and multi-day "bring-your-own-code" Tuning Workshops
    - Face-to-face & side-by-side hands-on coaching now successfully migrated to virtual/on-line events

https://www.vi-hps.org

Introduction to Parallel Performance Engineering

(with content used with permission from tutorials by Bernd Mohr/JSC and Luiz DeRose/Oracle)

Performance: An old problem



Difference Engine

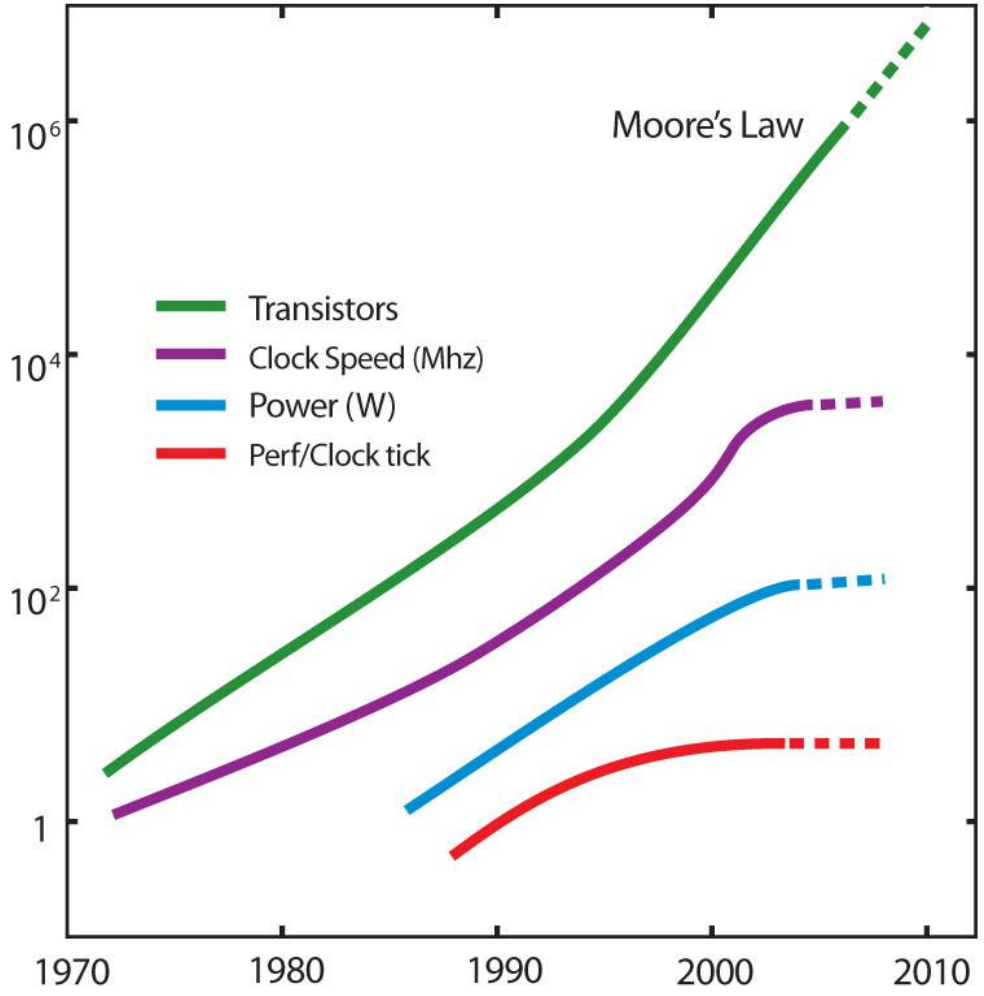
"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

Charles Babbage
1791 – 1871

Today: The "free lunch" is over

- Moore's law is still in charge, but
  - Clock rates no longer increase
  - Performance gains only through increased parallelism
- Optimizations of applications more difficult
  - Increasing application complexity
    - Multi-physics
    - Multi-scale
  - Increasing machine complexity
    - Hierarchical networks / memory
    - More CPUs / multi-core / accelerators

☞ Every doubling of scale reveals a new bottleneck!



Performance factors of parallel applications

- "Sequential" performance factors
  - Computation
  - Cache and memory
  - Input / output
- "Parallel" performance factors
  - Partitioning / decomposition
  - Communication (i.e., message passing)
  - Multithreading
  - Synchronization / locking

Parallel performance engineering in practice

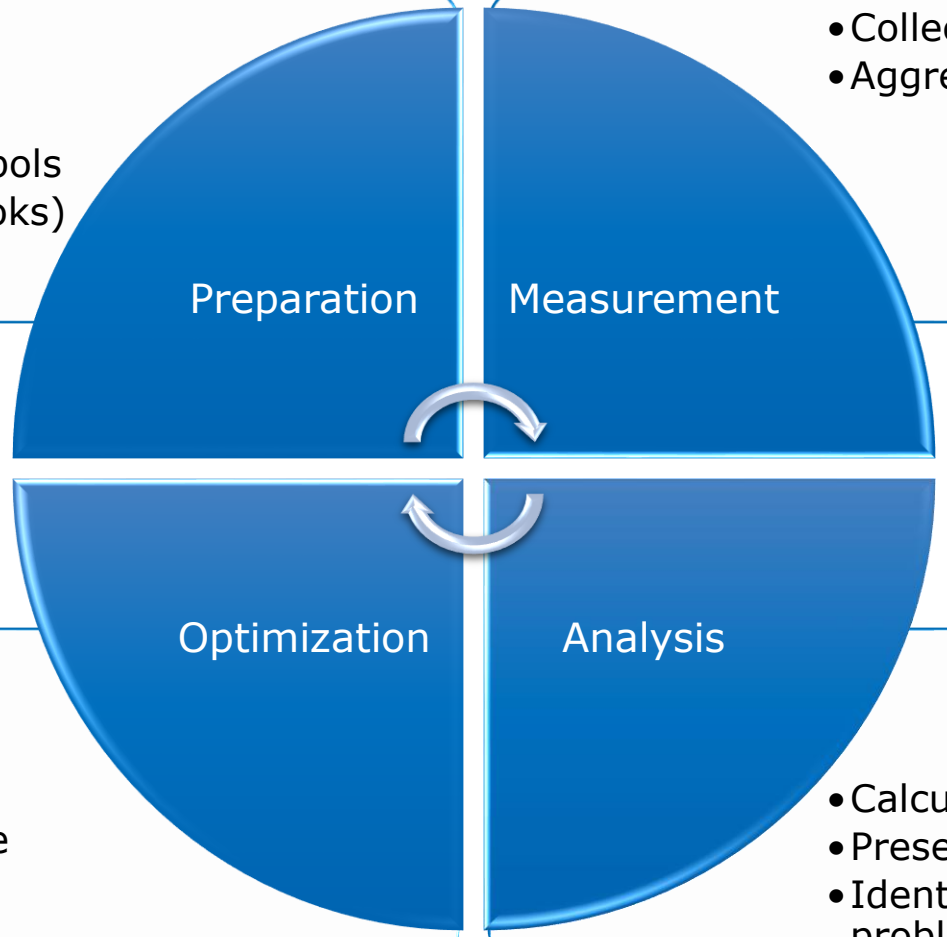
- Successful engineering is a combination of
  - Careful setting of various tuning parameters
  - The right algorithms and libraries
  - Compiler flags and directives
  - …
  - Thinking !!!
- Measurement is better than guessing…
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
  - ☞ *After each step!*
- … but avoid excessive "do-it-yourself" solutions!
  - Simple time measurements for phases OK to get a coarse overview, but specialized tools can provide many more insights

Parallel performance engineering in practice (cont.)

- Starting point: Well-understood application running at scale N
  - Make sure it produces correct results
  - It is advantageous to have (automatic) verification tests in place
- Define your goal
  - Reduce runtime/resource consumption at scale N
  - Retain good scalability going to M >> N
- Predict behavior
  - What is the current bottleneck?
  - What performance/scalability should we see?
- Measure possible bottlenecks
  - Idle resources
  - Changes in profile
- Compare observed behavior with expectation and draw conclusions

Performance engineering workflow

- Build model of predicted performance
- Select data to measure
- Prepare application with symbols
- Insert extra code (probes/hooks)



- Collection of performance data
- Aggregation of performance data

- Modifications intended to eliminate/reduce performance problem

- Calculation of metrics
- Presentation of results
- Identification of performance problems

The 80/20 rule

- Programs typically spend 80% of their time in 20% of the code
- Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application
  - ☞ *Know when to stop!*
- Don't optimize what does not matter
  - ☞ *Make the common case fast!*

"If you optimize everything, you will always be unhappy."

Donald E. Knuth

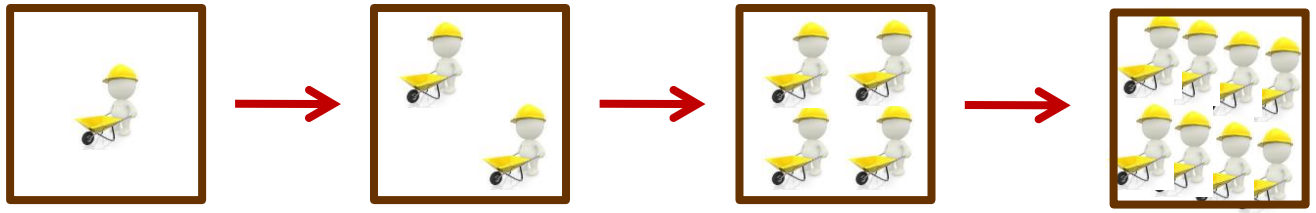
Performance modeling: Predicting behavior

- Simplest models: Scaling properties
  - Which parts of the code are serial and parallel?
  - How much time is spent in each?
  - How efficient are they currently?

- More complex concepts
  - Roofline model (comparing throughput to theoretical maxima)
  - Load balancing: What code is responsible for idle resources?
  - Critical path analysis (e.g., Scalasca)

Strong scaling

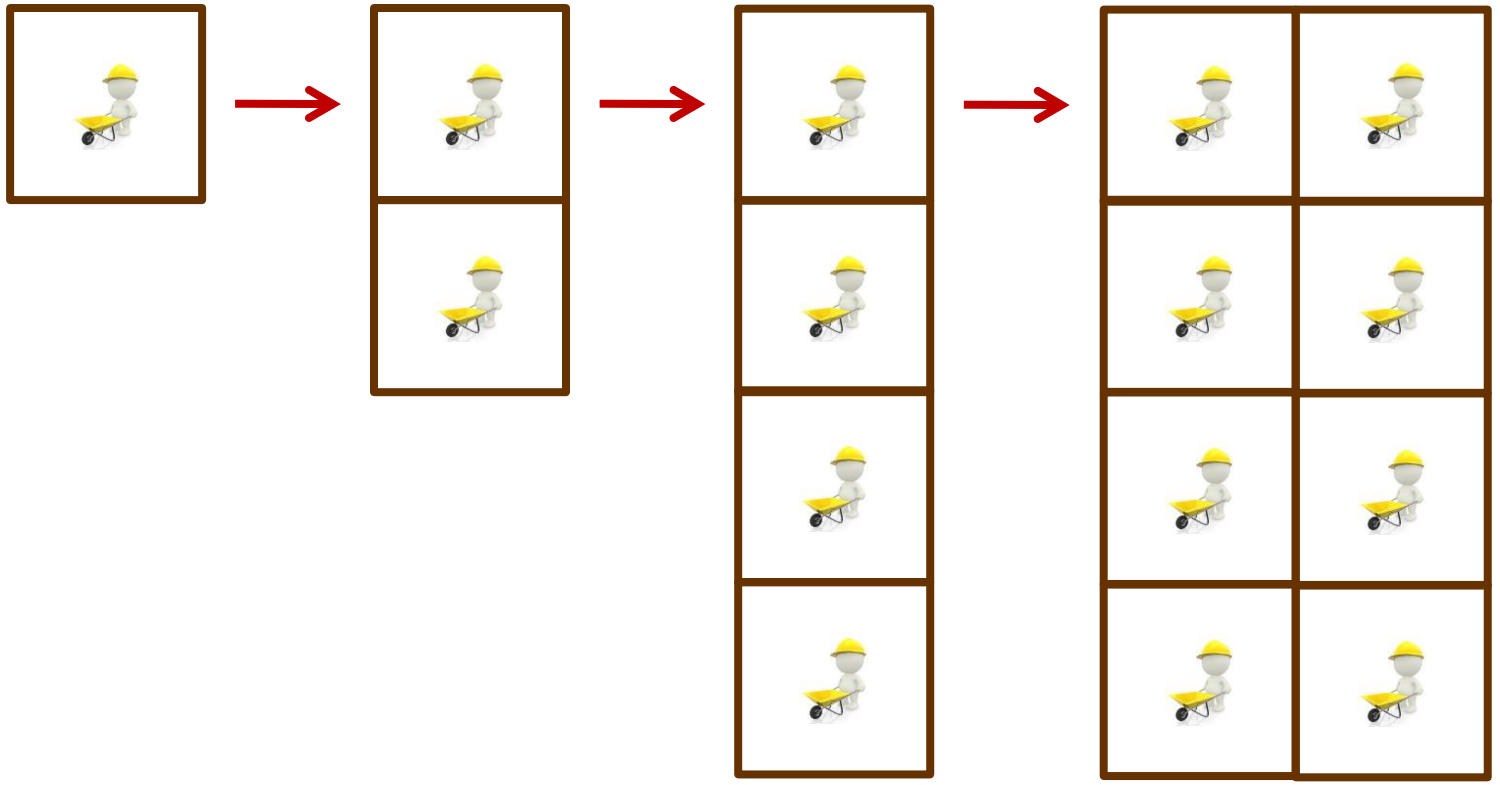
- Increasing compute power yields faster solution on the same problem



- Limited by Amdahl's law
  - Speedup ≤ (serial + parallel) / (serial + parallel / N) = 1 / (serial + parallel / N)

Weak scaling

- Increasing compute power yields larger problems solved in the same time



What to measure

- You have some hypothesis about how your code will behave
- This requires certain data
  - Simple scaling models: execution time, possibly subdivided between serial and parallel parts
  - Roofline model: operations/second and bytes/second corresponding to one or more rooflines
  - Load balancing: distribution of time spent in computation and communication
  - Critical path: detailed measurement of execution time across all nodes and threads
- Allows you to ignore certain other data
  - Example: Load balancing
  - Detection typically based on communication wait states
  - Don’t need to analyze computation details for that
- When possible, measure only what you need to test your hypothesis
  - All-in-one-run only when it’s unavoidable

Metrics of performance

- What can be measured?
  - A **count** of how often an event occurs
    - E.g., the number of MPI point-to-point messages sent
  - The **duration** of some interval
    - E.g., the time spent these send calls
  - The **size** of some parameter
    - E.g., the number of bytes transmitted by these calls
- Derived metrics
  - E.g., rates / throughput
  - Needed for normalization

Measurement practices

- Measurements on HPC systems are noisy
  - Shared resources: network, file systems
  - Nondeterminism: cache effects, which nodes were allocated, small race conditions
- Particularly relevant to wall time, but can affect other metrics
- As with all scientific measurements, repeat the experiment
  - Especially if the initial results look weird!

Measurement issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behaviour
    - E.g., memory access pattern
  - Accuracy of timers & counters
- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

☞ *Tradeoff: Accuracy vs. Expressiveness of data*

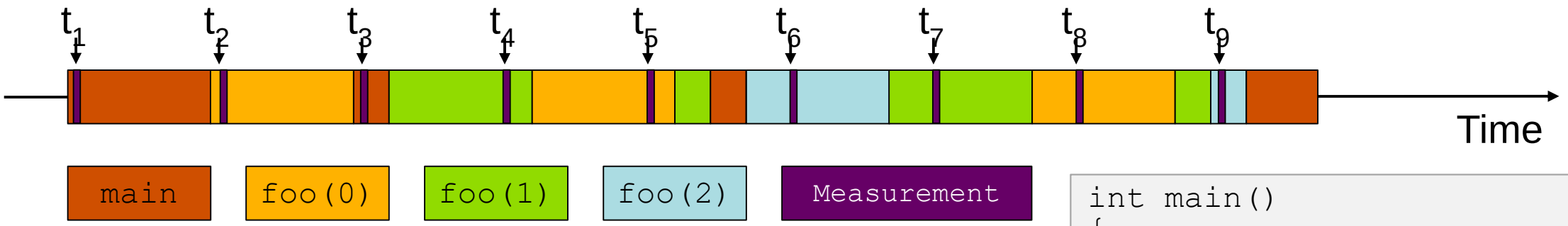
Classification of measurement techniques

- **How are performance measurements triggered?**
  - **Sampling**
  - **Code instrumentation**

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem

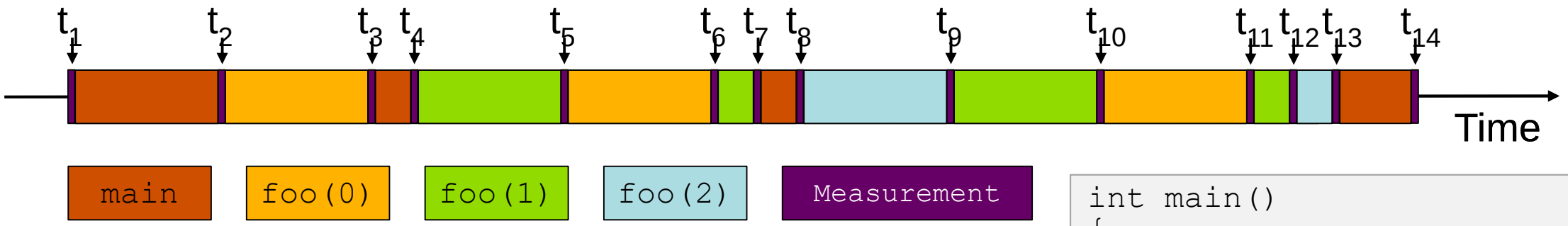
Sampling



- Running program is periodically interrupted to take measurement
  - Timer interrupt, OS signal, or HWC overflow
  - Service routine examines return-address stack
  - Addresses are mapped to routines using symbol table information
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i - 1);

}
```

Instrumentation



- Measurement code is inserted such that every event of interest is captured directly
  - Can be done in various ways
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions

```
int main()
{
  int i;
  Enter("main");
  for (i=0; i < 3; i++)
    foo(i);
  Leave("main");
  return 0;
}

void foo(int i)
{
  Enter("foo");
  if (i > 0)
    foo(i - 1);
  Leave("foo");
}
```

Instrumentation techniques

- **Static** instrumentation
  - Program is instrumented prior to execution
- **Dynamic** instrumentation
  - Program is instrumented at runtime

- Code is inserted
  - Manually
  - Automatically
    - By a preprocessor / source-to-source translation tool
    - By a compiler
    - By linking against a pre-instrumented library / runtime system
    - By binary-rewrite / dynamic instrumentation tool

Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation
- **How is performance data recorded?**
  - **Profiling / Runtime summarization**
  - **Tracing**
- How is performance data analyzed?
  - Online
  - Post mortem

Profiling / Runtime summarization

- Recording of aggregated information
  - Total, maximum, minimum, …
- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters
- Over program and system entities
  - Functions, call sites, basic blocks, loops, …
  - Processes, threads

☞ *Profile = summarization of events over execution interval*

Tracing

- Recording detailed information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events

☞ *Event trace = Chronologically ordered sequence of event records*

Event tracing

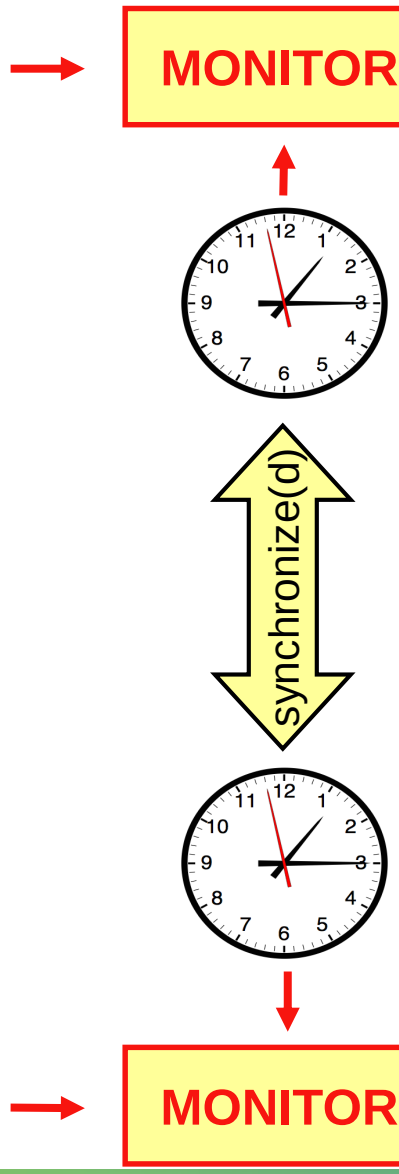
Process A

```
void foo() {
  trc_enter("foo");
  ...
  trc_send(B);
  send(B, tag, buf);
  ...
  trc_exit("foo");
}
```

instrument

Process B

```
void bar() {
  trc_enter("bar");
  ...
  recv(A, tag, buf);
  trc_recv(A);
  ...
  trc_exit("bar");
}
```



Local trace A

...	
58	ENTER foo
62	SEND to B
64	EXIT foo
...	

Local trace B

...	
60	ENTER bar
68	RECV from A
69	EXIT bar
...	

(Virtual merge)

Global trace view

...		
58	A	ENTER foo
60	B	ENTER bar
62	A	SEND to B
64	A	EXIT foo
68	B	RECV from A
69	B	EXIT bar
...		

Tracing Pros & Cons

- Tracing advantages
  - Event traces preserve the temporal and spatial relationships among individual events (☞ context)
  - Allows reconstruction of dynamic application behavior on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Writing events to file at runtime may causes perturbation

Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation
- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing
- **How is performance data analyzed?**
  - **Online**
  - **Post mortem**

Online analysis

- Performance data is processed during measurement run
  - Process-local profile aggregation
  - Requires formalized knowledge about performance bottlenecks
  - More sophisticated inter-process analysis using
    - “Piggyback” messages
    - Hierarchical network of analysis agents
- Online analysis often involves application steering to interrupt and re-configure the measurement

Post-mortem analysis

- Performance data is stored at end of measurement run
- Data analysis is performed afterwards
  - Automatic search for bottlenecks
  - Visual trace analysis
  - Calculation of statistics

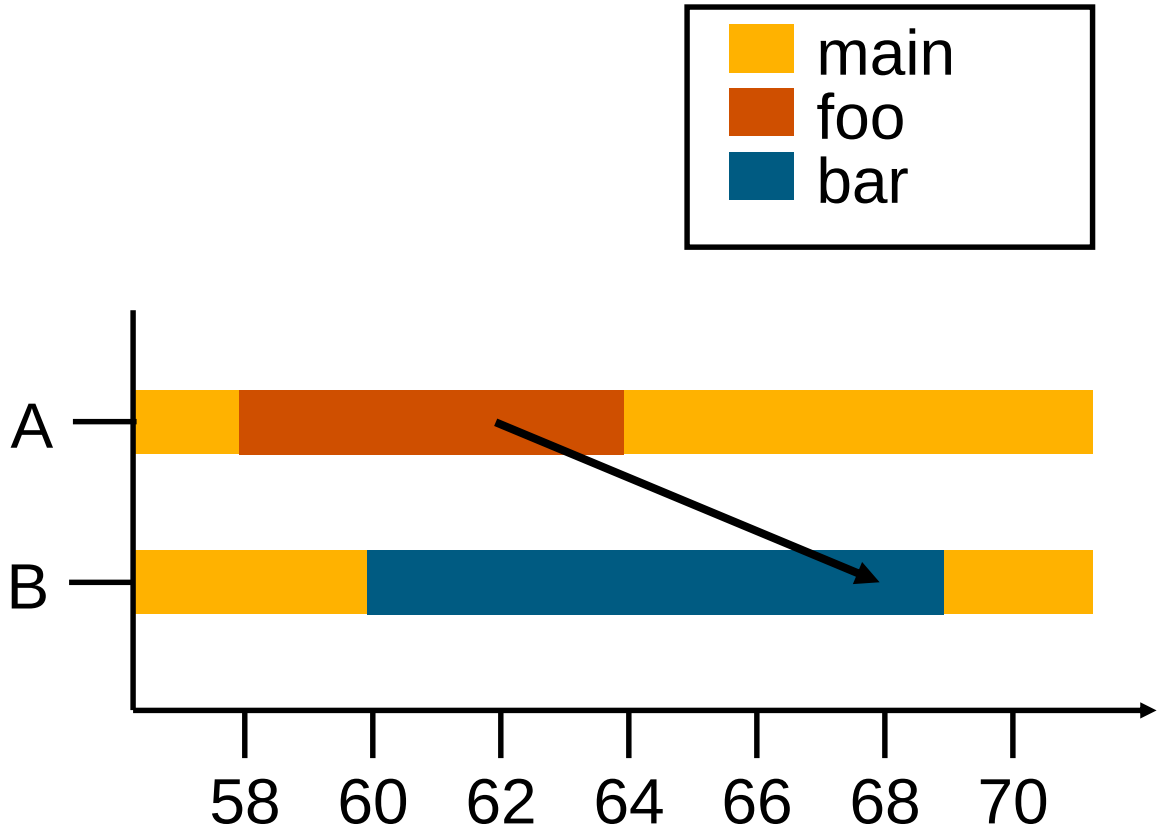
Example: Time-line visualization

Global trace view

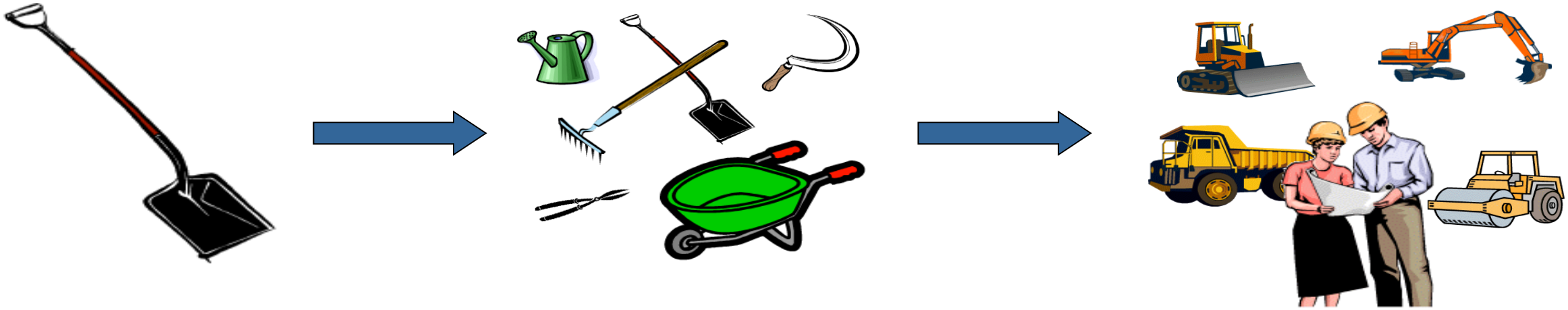
...		
58	A	ENTER foo
60	B	ENTER bar
62	A	SEND to B
64	A	EXIT foo
68	B	RECV from A
69	B	EXIT bar
...		

Post-Mortem

Analysis



No single solution is sufficient!



A combination of different methods, tools and techniques is typically needed!

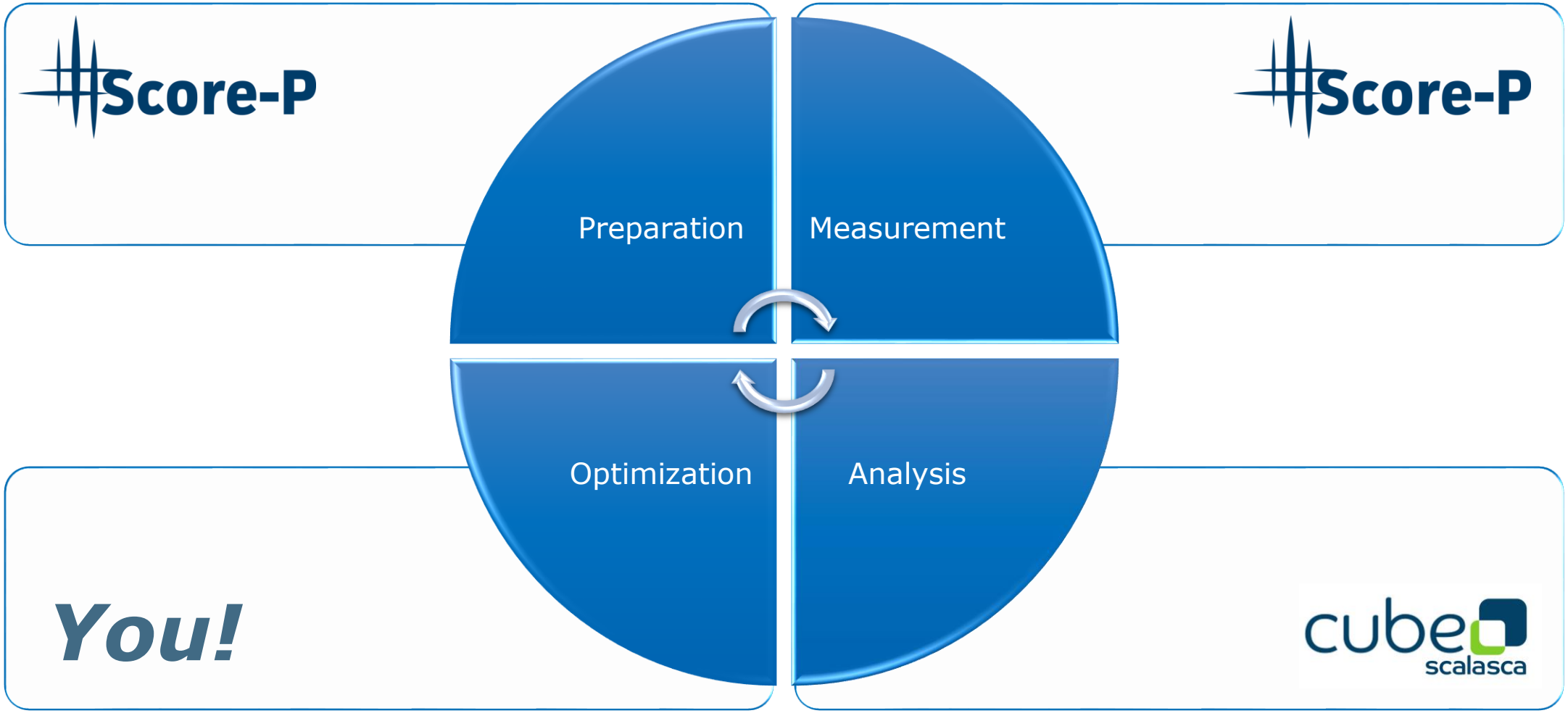
- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...

Typical performance analysis procedure

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements
- **What** is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling
- **Where** is the key bottleneck?
  - Call-path profiling, detailed basic block profiling
- **Why** is it there?
  - Hardware counter analysis, trace selected parts to keep trace size manageable
- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function

Tools Overview

Performance engineering workflow



Score-P

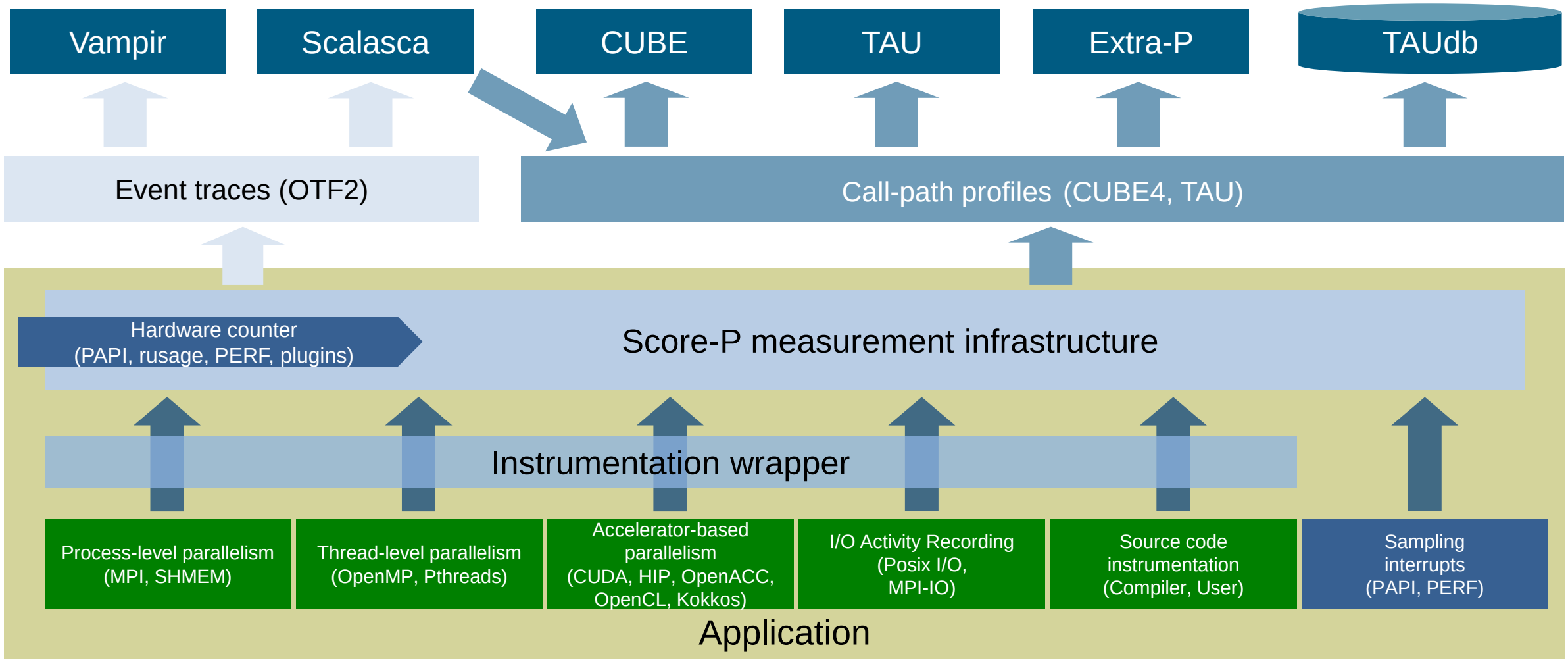
DOI 10.5281/zenodo.1240731

- Infrastructure for instrumentation and performance measurements
- Instrumented application can be used to produce several results:
  - Call-path profiling: CUBE4 data format used for data exchange
  - Event-based tracing: OTF2 data format used for data exchange
- Supported parallel paradigms:
  - Multi-process: MPI, SHMEM
  - Thread-parallel: OpenMP, POSIX threads
  - Accelerator-based: CUDA, HIP, OpenCL, OpenACC, Kokkos

- Open Source; portable and scalable to all major HPC systems
- Initial project funded by BMBF
- Further developed in multiple 3rd-party funded projects

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Score-P ecosystem



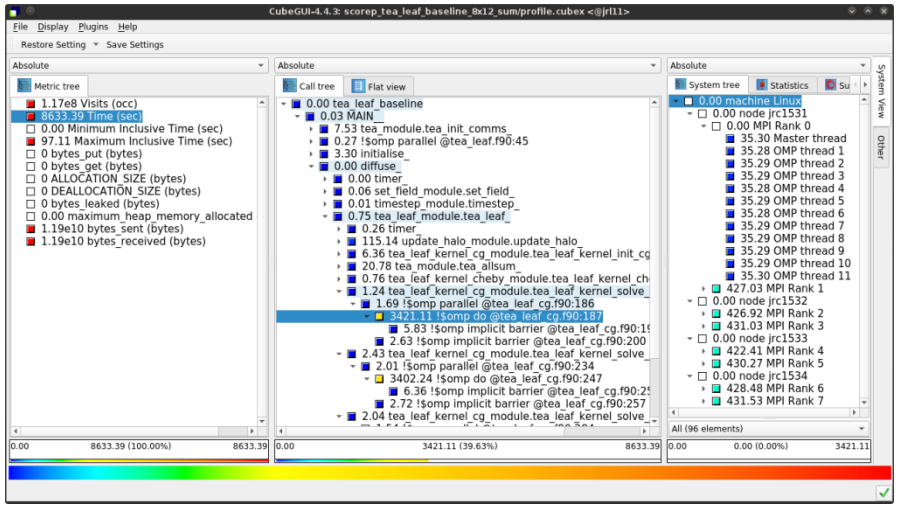
Score-P features

- Open source: 3-clause BSD license
  - Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners
- Portability: supports all major HPC platforms
- Scalability: successful measurements with >1M threads
- Functionality:
  - Generation of call-path profiles and event traces (supporting highly scalable I/O)
  - Using direct instrumentation and sampling
  - Flexible measurement configuration without re-compilation
  - Recording of time, visits, communication data, hardware counters
  - Support for MPI, SHMEM, OpenMP, Pthreads, CUDA, HIP, OpenCL, OpenACC, Kokkos (and many combinations)
- Latest release: Score-P 8.4 (March 2024)

Cube

CubeLib DOI 10.5281/zenodo.1248078
CubeGUI DOI 10.5281/zenodo.1248087

- Parallel program analysis report exploration tools
  - Libraries for XML+binary report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
    - Requires Qt ≥ 5



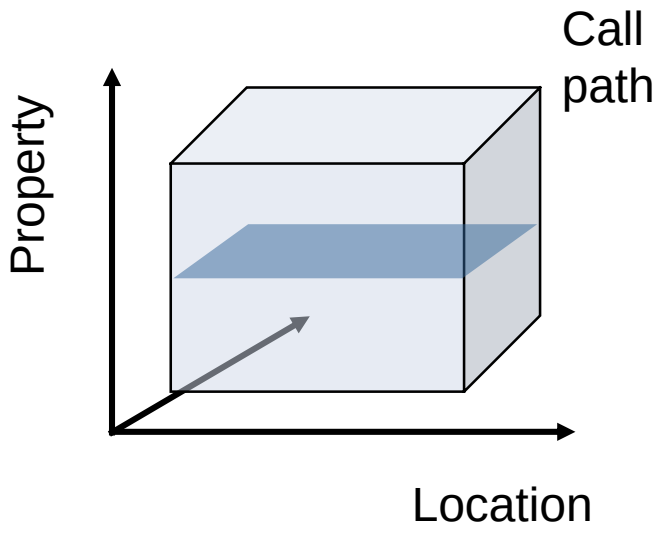
- Originally developed as part of the Scalasca toolset

- Now available as a separate component
  - Can be installed independently of Score-P and Scalasca, e.g., on laptop or desktop
  - Latest release: Cube v4.8.2 (September 2023)

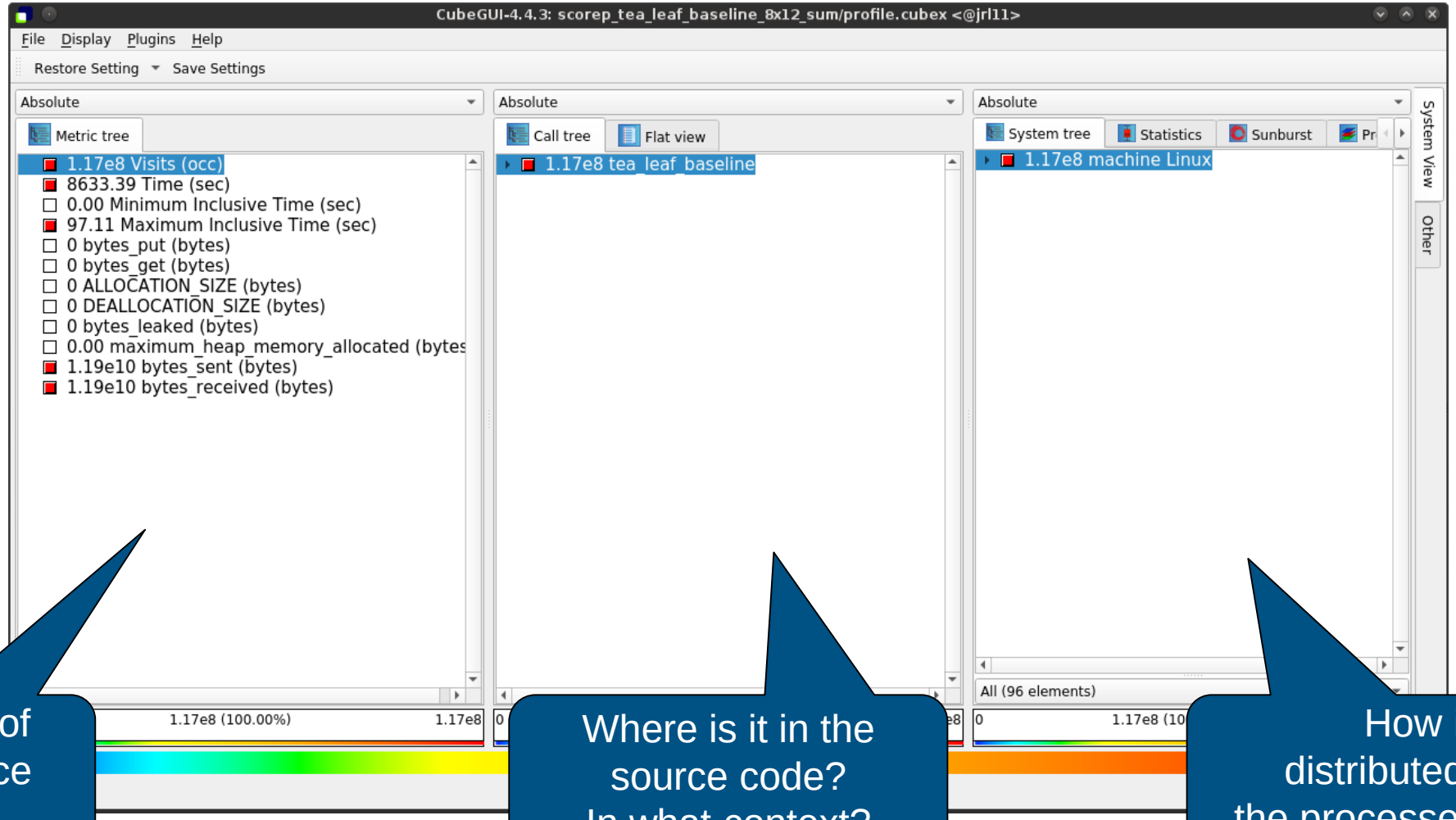
Note: source distribution tarballs for Linux, as well as binary packages provided for Windows & MacOS, from **www.scalasca.org** website in software/Cube-4x

Analysis presentation and exploration

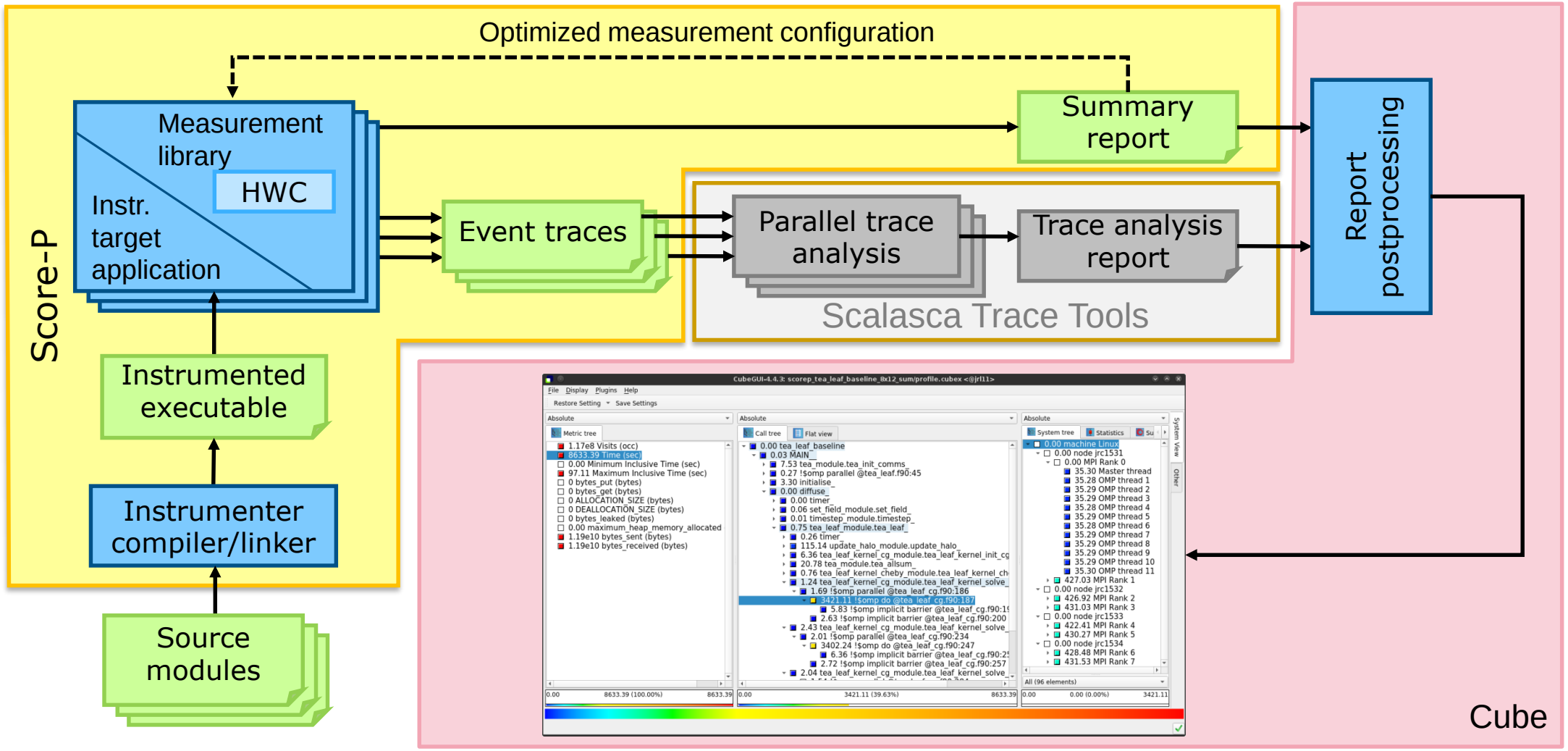
- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call path (program location)
  - System location (process/thread)
- Three coupled tree browsers
- Cube displays severities
  - As value: for precise comparison
  - As color: for easy identification of hotspots
  - Inclusive value when closed & exclusive value when expanded
  - Customizable via display modes



Analysis presentation



Putting it all together



Sponsors

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

SEVENTH FRAMEWORK
PROGRAMME

European
Commission

Horizon 2020
European Union funding
for Research & Innovation

EuroHPC
Joint Undertaking

Funded by
DFG Deutsche
Forschungsgemeinschaft
German Research Foundation

HELMHOLTZ
ASSOCIATION

ITEA 2

OAK
RIDGE
National Laboratory

intel

AMD

SIEMENS

NVIDIA