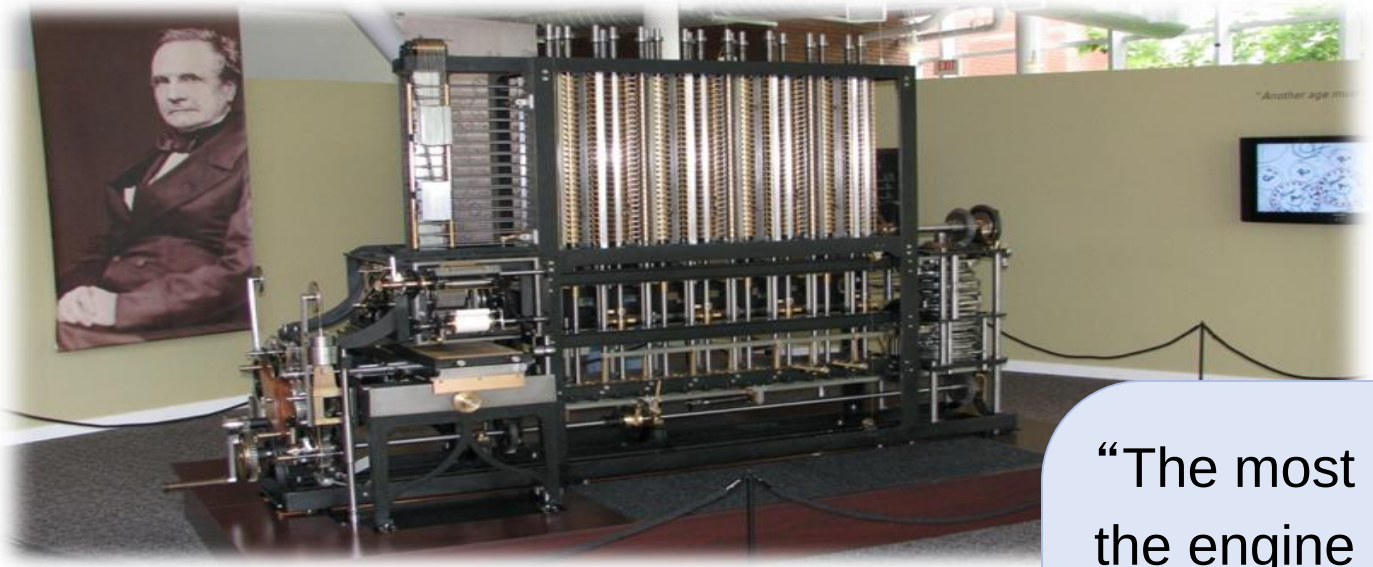
INTRODUCTION TO PARALLEL PERFORMANCE ENGINEERING

JUNE 2, 2026 I MICHAEL KNOBLOCH I ILYA ZHUKOV | {M.KNOBLOCH,I.ZHUKOV}@FZ-JUELICH.DE

PERFORMANCE: AN OLD PROBLEM



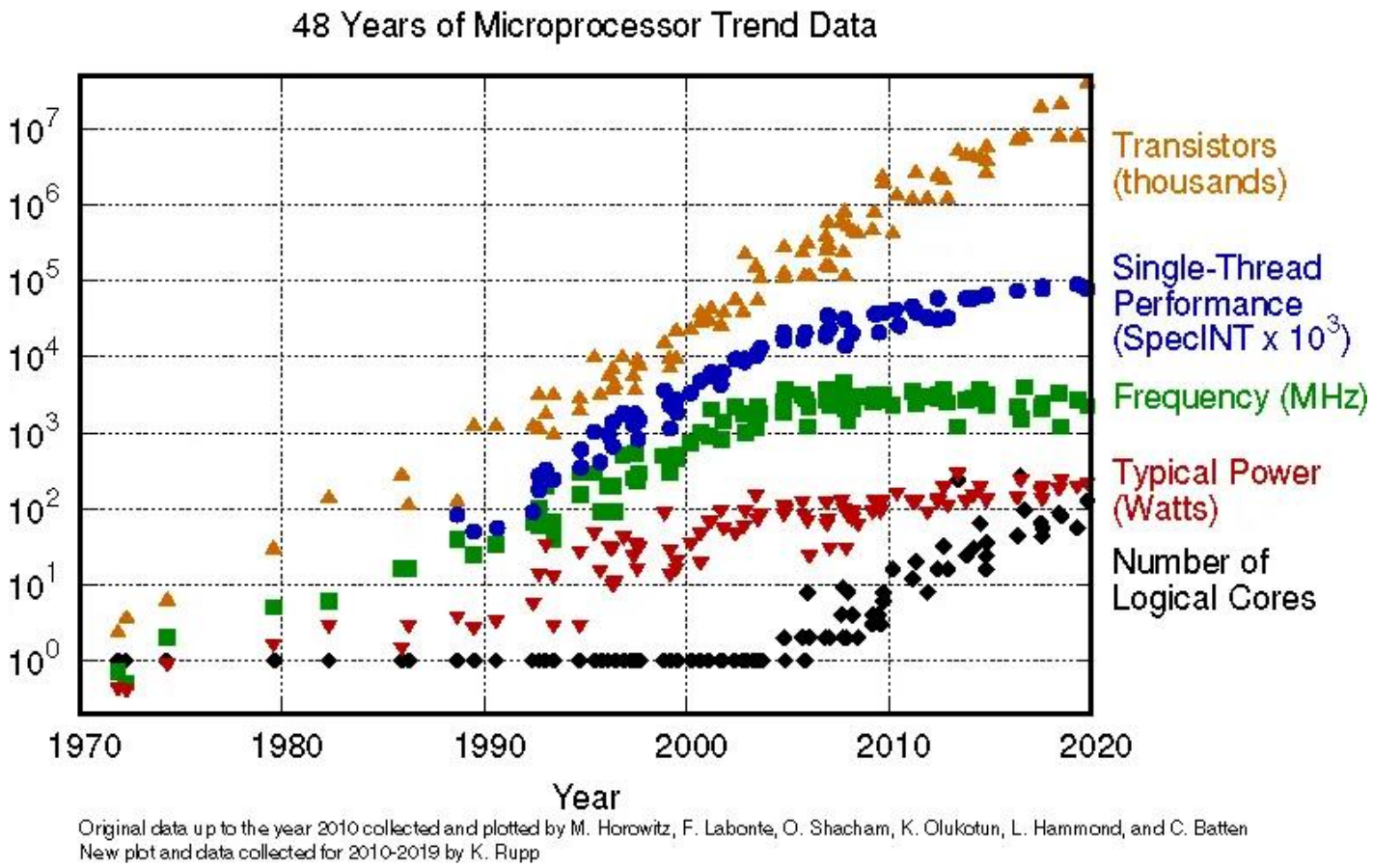
Difference Engine (1820s)

> “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
- Optimization of applications more difficult
  - Increasing application complexity
    - Multi-physics
    - Multi-scale
  - Increasing machine complexity
    - Hierarchical networks / memory
    - Many-core CPUs and Accelerators
    - Modular Supercomputing Architecture

☞ Every doubling of scale reveals a new bottleneck!



PERFORMANCE FACTORS

- “Sequential” (single core) factors
  - Computation (Pipelining, Out-of-order execution, …)
    - ☞ Choose right algorithm, use optimizing compiler
  - Vectorization
    - ☞ Choose right algorithm, use optimizing compiler
  - Cache and memory
    - ☞ Choose the right data structures and data layout

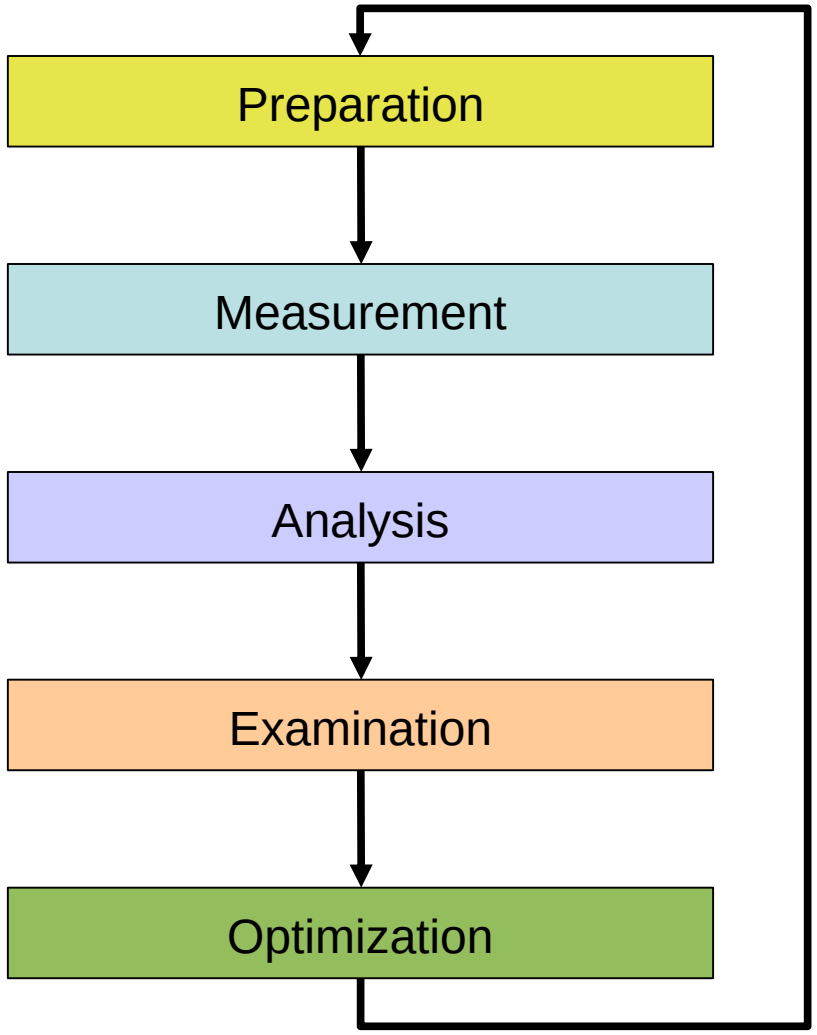
PERFORMANCE FACTORS

- “Parallel” (multi core/node) factors
  - Partitioning / decomposition
    - ☞ Load balancing
  - Communication (i.e., message passing)
  - Multithreading
  - Core binding / NUMA
  - Synchronization / locking
  - I/O
    - ☞ Often not given enough attention
    - ☞ Parallel I/O matters

TUNING BASICS

- Carefully set various tuning parameters
  - The right (parallel) algorithms and libraries
  - Compiler flags and directives
  - Correct machine usage (mapping and bindings)
    - ☞Get the most performance before analysis!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - ☞After each step!

PERFORMANCE ENGINEERING WORKFLOW



- Prepare application (with symbols), insert extra code (probes/hooks)
- Collection of data relevant to execution performance analysis
- Calculation of metrics, identification of performance metrics
- Presentation of results in an intuitive/understandable form
- Modifications intended to eliminate/reduce performance problems

THE 80/20 RULE

- Programs typically spend 80% of their time in 20% of the code
  - ☞ *Know what matters!*
- Developers 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!*

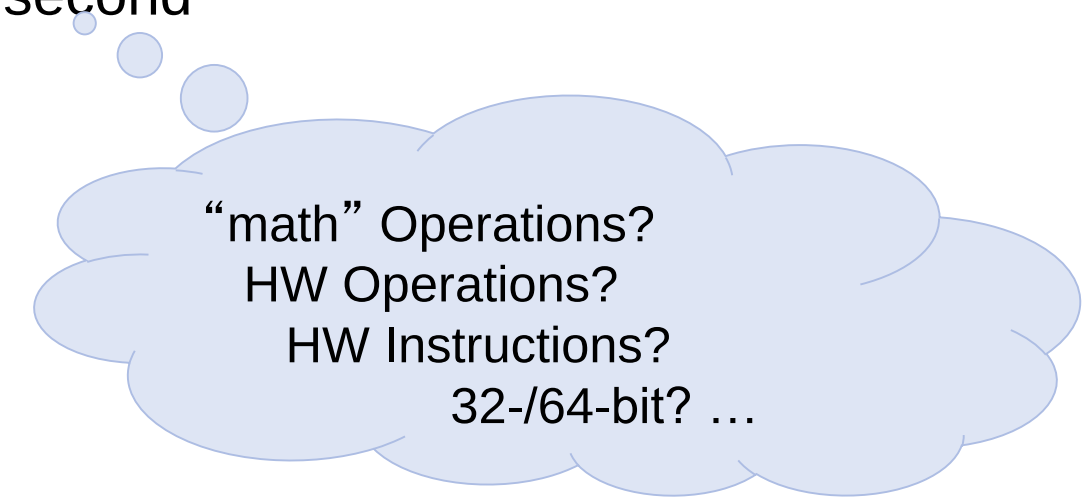
METRICS OF PERFORMANCE

- 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
  - Rates (e.g. FLOPS), Throughput (e.g. network, memory), …
  - Needed for normalization (qualitative vs. quantitative analysis)

EXAMPLE METRICS

- Execution time
- Number of function calls
- CPI
  - CPU cycles per instruction
- FLOPS
  - Floating-point operations executed per second



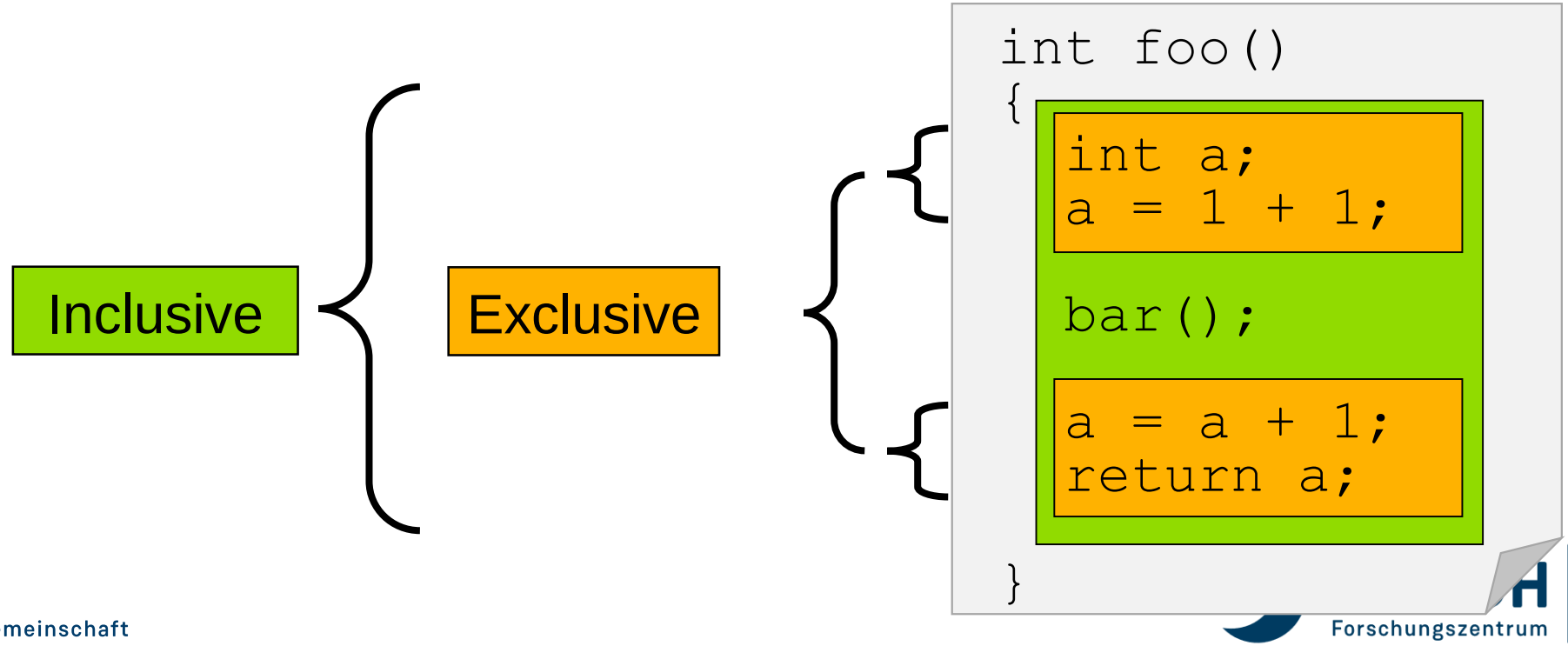
EXECUTION TIME

- Wall-clock time
  - Includes waiting time: I/O, memory, other system activities
  - In time-sharing environments also the time consumed by other applications
- CPU time
  - Time spent by the CPU to execute the application
  - Does not include time the program was context-switched out
    - Problem: Does not include inherent waiting time (e.g., I/O)
    - Problem: Portability? What is user, what is system time?

- Problem: Execution time is non-deterministic
  - Use mean or minimum of several runs

INCLUSIVE VS. EXCLUSIVE VALUES

- Inclusive
  - Information of all sub-elements aggregated into single value
- Exclusive
  - Information cannot be subdivided further



MEASUREMENT TECHNIQUE CLASSIFICATION

Two dimensions

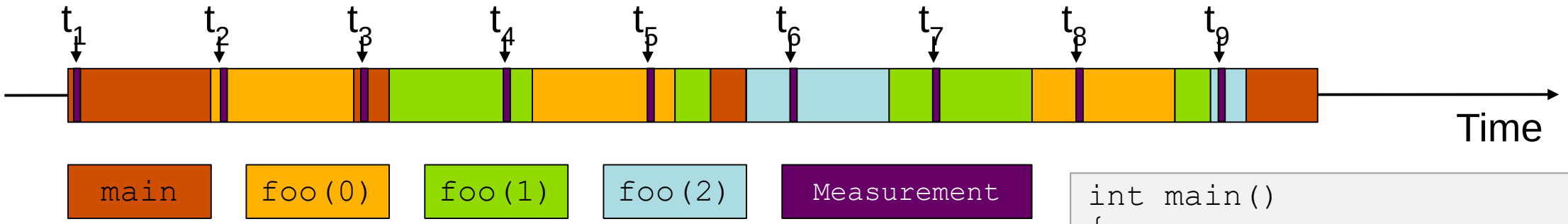
When performance measurement is triggered

- **External trigger** (asynchronous)
  - **Sampling**
    - Trigger: Timer interrupt OR Hardware counters overflow
- **Internal trigger** (synchronous)
  - Code **instrumentation** (automatic or manual)

How performance data is recorded

- **Profile**
  - Summation of events over time
- **Trace**
  - Sequence of events over time

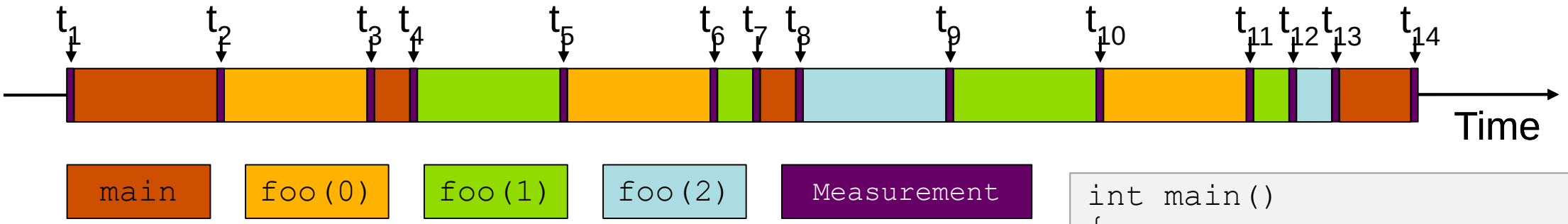
SAMPLING



- 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

CRITICAL ISSUES

- Accuracy
  - Intrusion overhead
    - Measurement takes 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*

MEASUREMENT METHODS: PROFILING

- Recording of **aggregated information**
  - Time
  - Counts
    - Calls
    - Hardware counters
- **Across program and system entities**
  - Functions, call sites, loops, basic blocks, …
  - Processes, threads
- **Statistical information**
  - Min, max, mean and total number of values

Advantages
+ **Works also for long-running programs**

Disadvantages
– **Variations over time get lost**

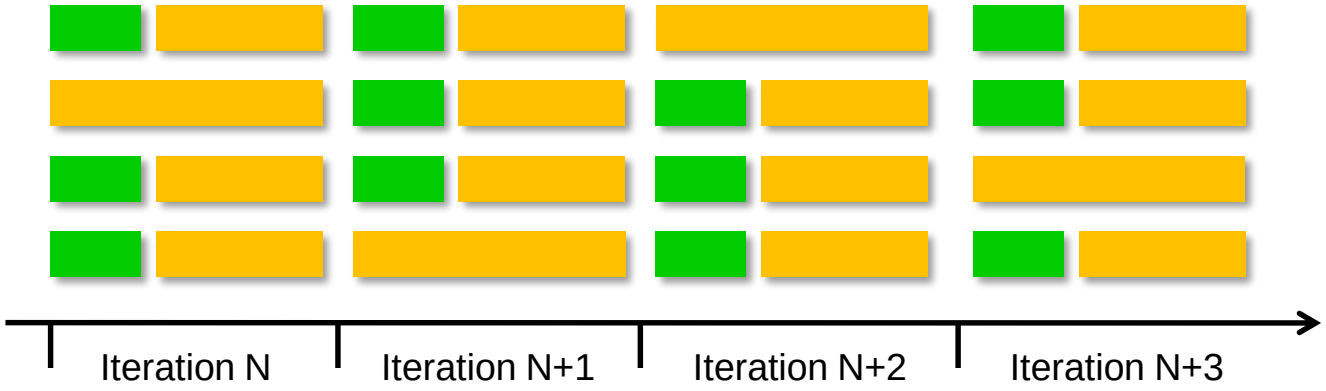
⇨ *Profile = summarization of events over execution interval*

TYPES OF PROFILES

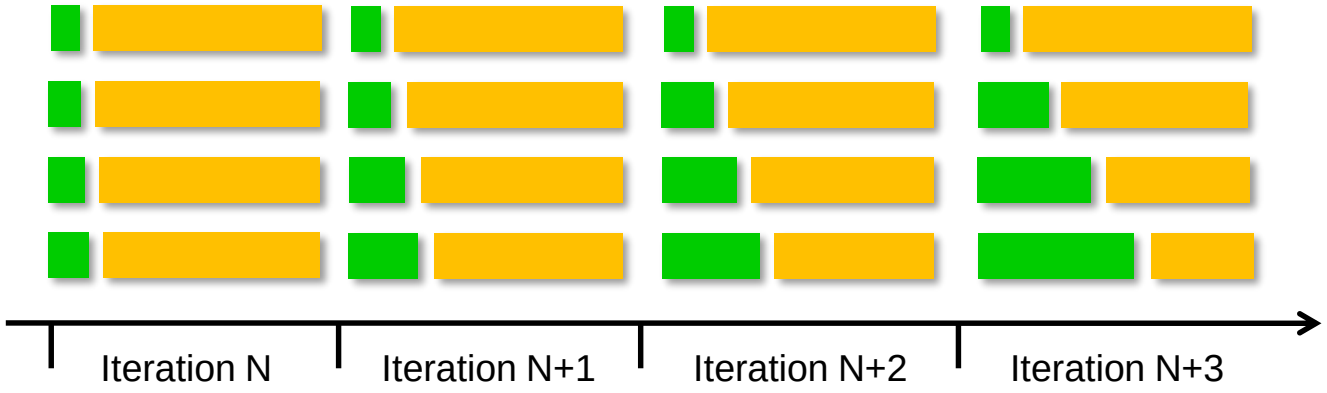
- Flat profile
  - Shows distribution of metrics per routine / instrumented region
  - Calling context is not taken into account
- Call-path profile
  - Shows distribution of metrics per executed call path
  - Sometimes only distinguished by partial calling context (e.g., two levels)
- Special-purpose profiles
  - Focus on specific aspects, e.g., MPI calls or OpenMP constructs
  - Comparing processes/threads

PROFILING: ISSUES RELATED TO "AVERAGING"

- Moving bottleneck across processors can "average out" imbalances



- Imbalance changes over time ⇨ problem might not appear in short runs!



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
  - plus event specific information
- **Event trace** := stream of event records sorted by time

Advantages

+ **Can be used to reconstruct the dynamic behavior**
+ **Profiles can be calculated out of trace data**

Disadvantages

– **HUGE trace files**
– **Can only be used for short durations or small configurations**

⇨ *Trace = Abstract execution model on level of defined events*

EVENT TRACING

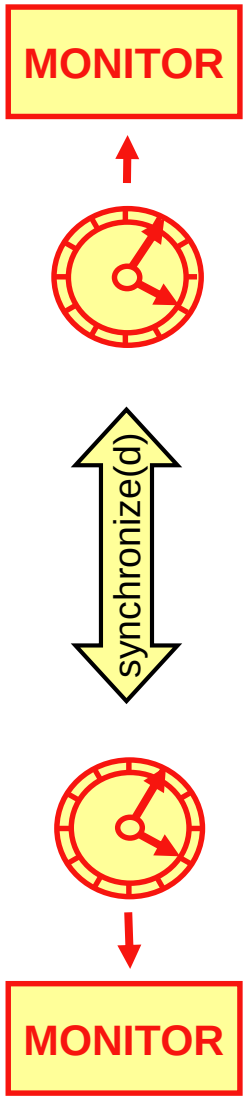
Process A

```
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	1
62	SEND	B
64	EXIT	1
...		

1	foo
...	

Local trace B

...		
60	ENTER	1
68	RECV	A
69	EXIT	1
...		

1	bar
...	

Global trace

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

merge

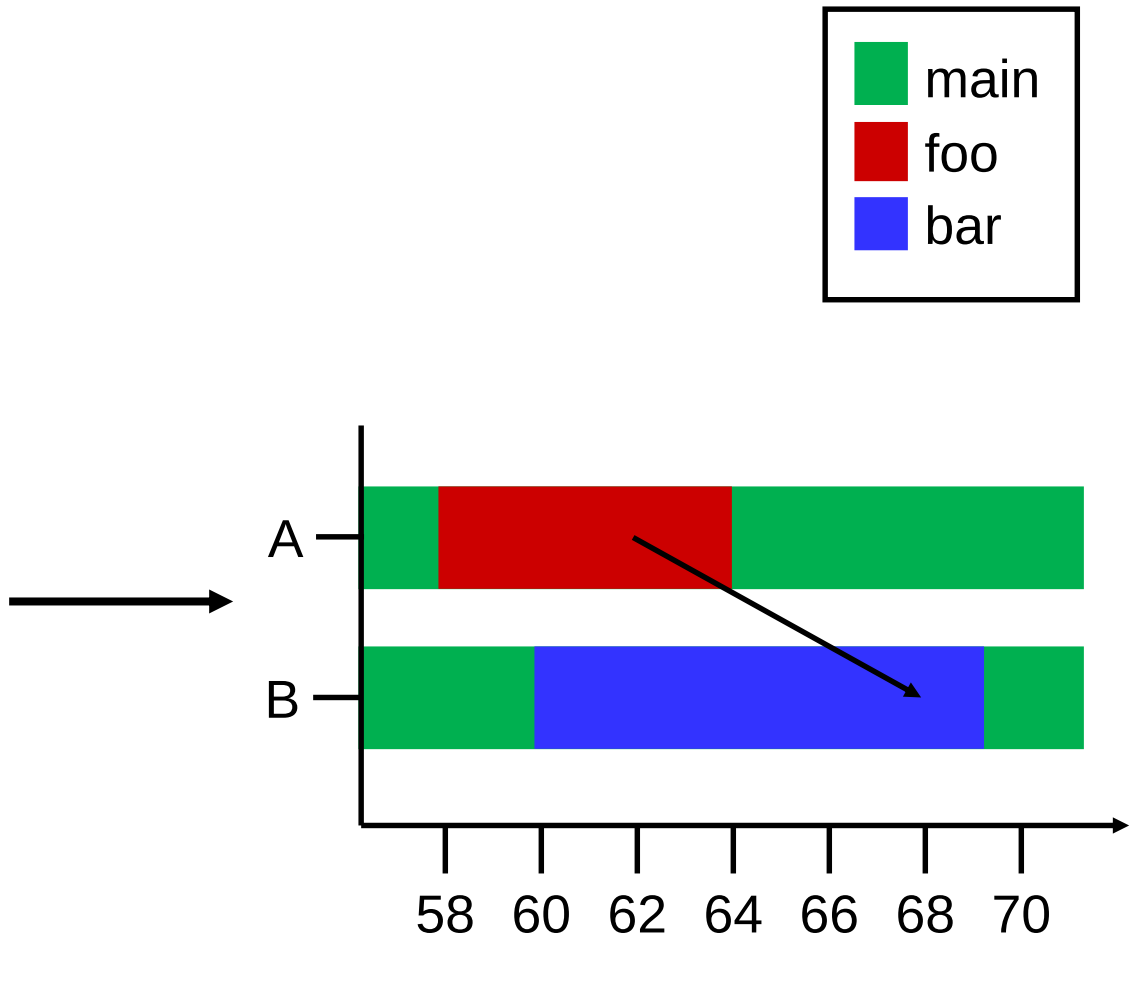
unify

1	foo
2	bar
...	

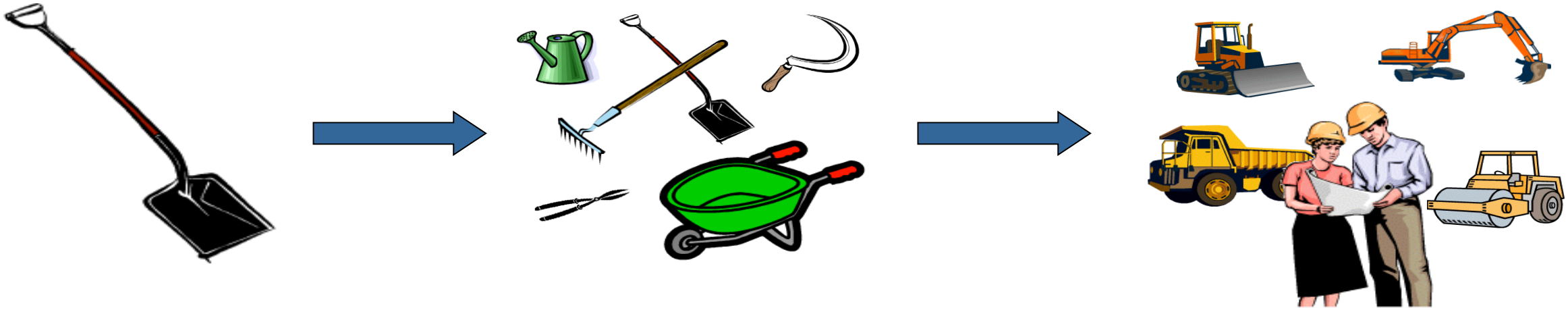
EVENT TRACING: “TIMELINE” VISUALIZATION

1	foo
2	bar
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
...			



REMARK: 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, ...

PERFORMANCE ANALYSIS RECOMMENDATIONS

- Measure and analyze at the desired scale (once you have a reasonable measurement setup)
- Get performance overview (e.g. Linaro Performance Reports, Intel APS)
  - CPU Issues:
    - Use low-level profiler (e.g. Linaro MAP, Vtune (on Intel nodes), or uProf (on AMD nodes))
    - Analyze hardware counters (e.g. perf, LIKWID, or PAPI)
  - MPI Issues: Use Scalasca/Vampir
  - GPU Issues: Use vendor tools (e.g. NVIDIA Nsight Compute, AMD ROCm Compute Profiler)
  - I/O Issues: Use DARSHAN
- OR: Do it all with Score-P/Scalasca/Vampir (cf. next webinar in series)

NEED HELP?

- Talk to the experts
  - Use local 1st-level support
  - Use tools mailing lists
  - VI-HPS Tuning Workshops
    - Bring-your-own-code workshop series covering multiple performance analysis tools
  - POP CoE
    - Performance assessment as a service
  - EPICURE
    - Support for performance analysis and debugging

☞Successful performance engineering often is a collaborative effort

QUESTIONS