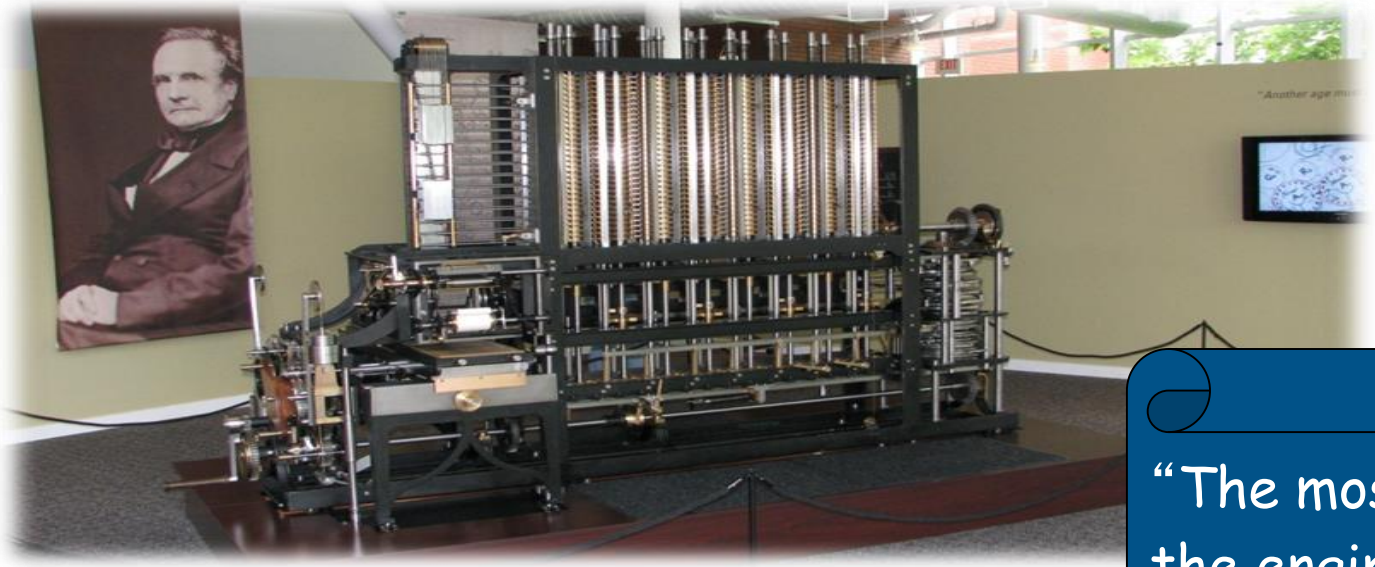
# Introduction to Parallel Performance Engineering

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

# Performance: an old problem



Difference Engine

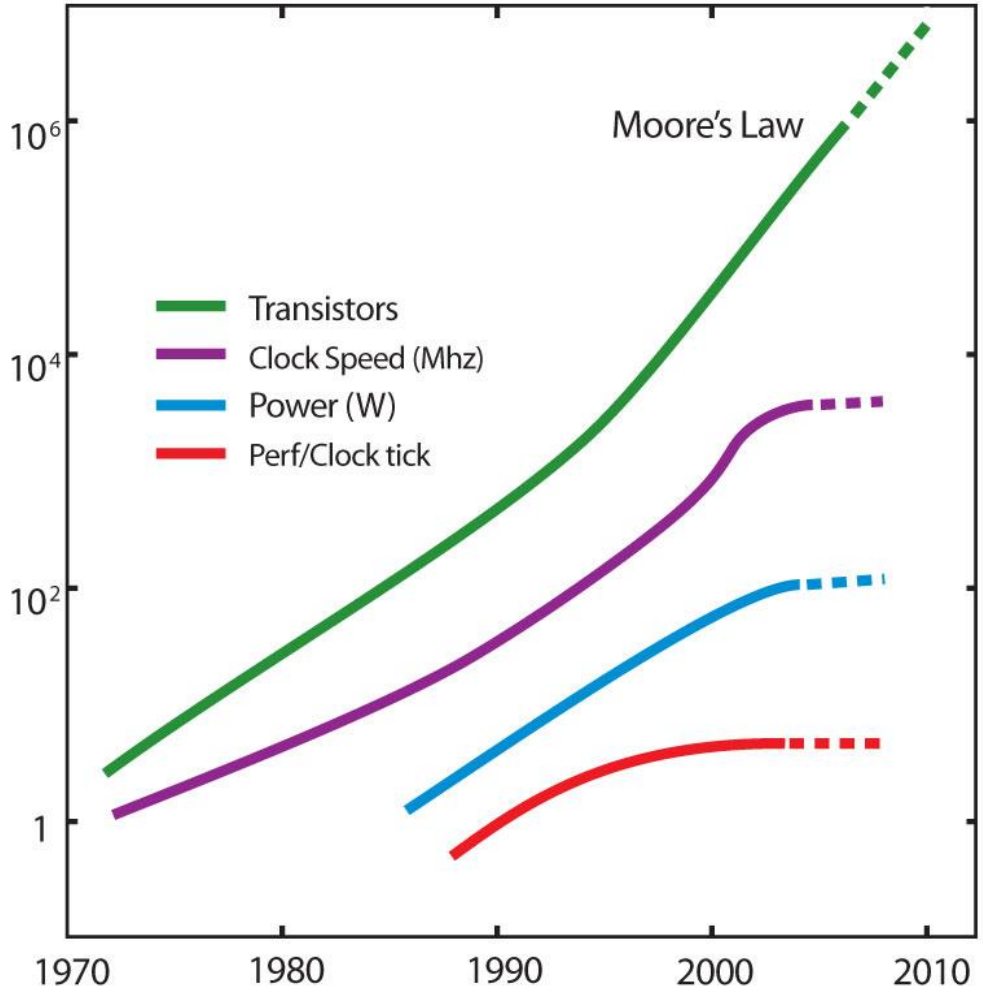
"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

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

☐Every doubling of scale reveals a new bottleneck!



# Consequences

- Machine complexity
  - Increasing number of different architectures
  - Additional optimisation challenges related to parallelism
  - CPU core performance issues tied to increased vector length and memory hierarchy

☐ The optimisation process remains key to maintain a reasonable performance level

- Code complexity
  - Codes are harder to optimise and maintain manually
  - Optimisation is time consuming and error-prone

☐ Understanding application execution behaviour is critical

# Performance factors of parallel applications

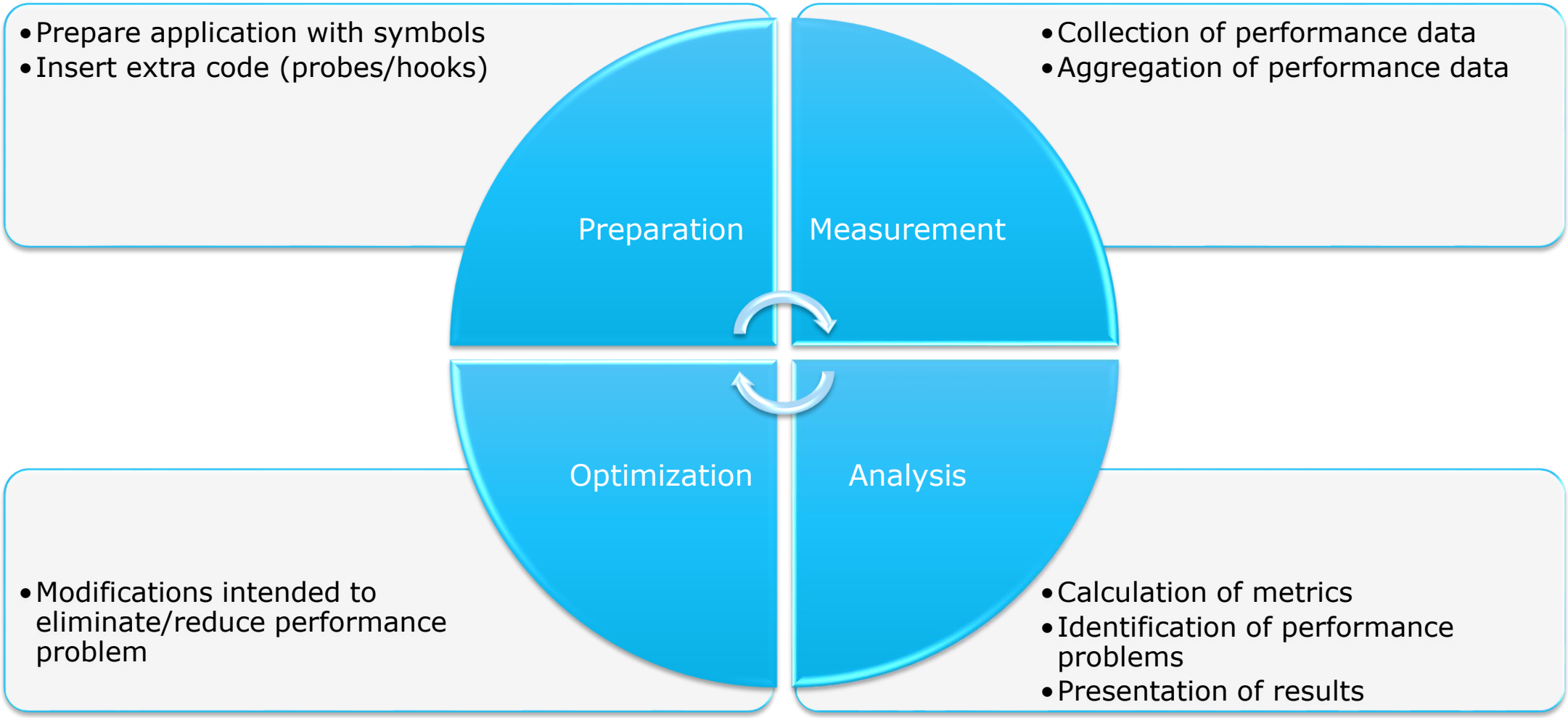
- "Sequential" performance factors
  - Computation
    - Choose right algorithm, use optimizing compiler
  - Cache and memory
    - Tough! Only limited tool support, hope compiler gets it right
  - Input / output
    - Often not given enough attention

- "Parallel" performance factors
  - Partitioning / decomposition
  - Communication (i.e., message passing)
  - Multithreading
  - Synchronization / locking
    - More or less understood, good tool support

# Tuning basics

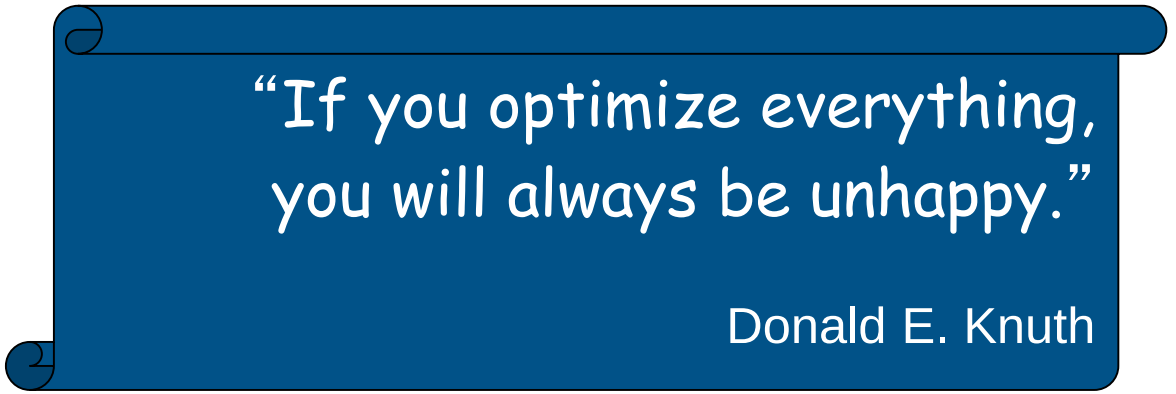
- Successful engineering is a combination of
  - Careful setting of various tuning parameters
  - The right algorithms and libraries
  - Compiler flags and directives
  - …
  - Thinking !!!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - After each step!
- Modeling is extremely useful but very difficult and rarely available
  - Allows to evaluate performance impact of optimization without implementing it
  - Simplifies search in large parameter space

# Performance engineering workflow



# 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!

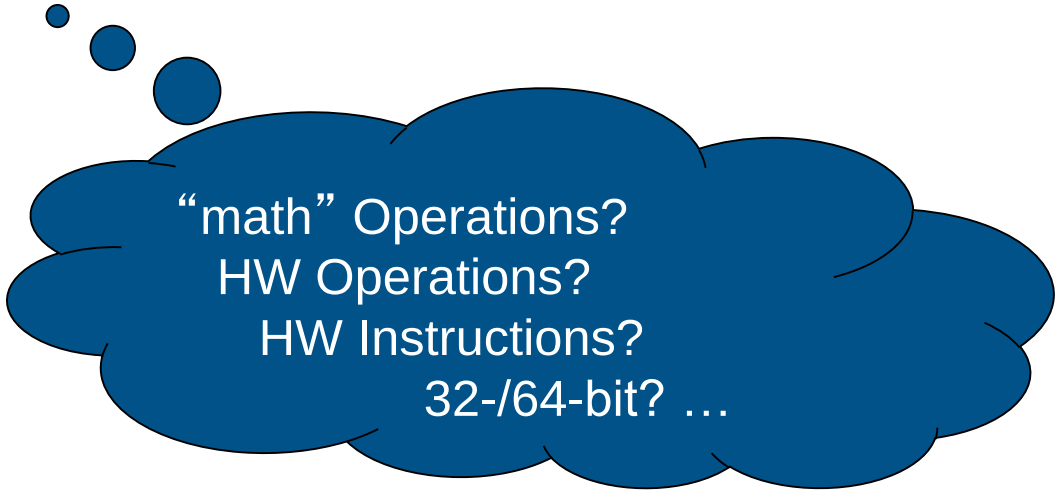


# Metrics of performance

- 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

# Example metrics

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

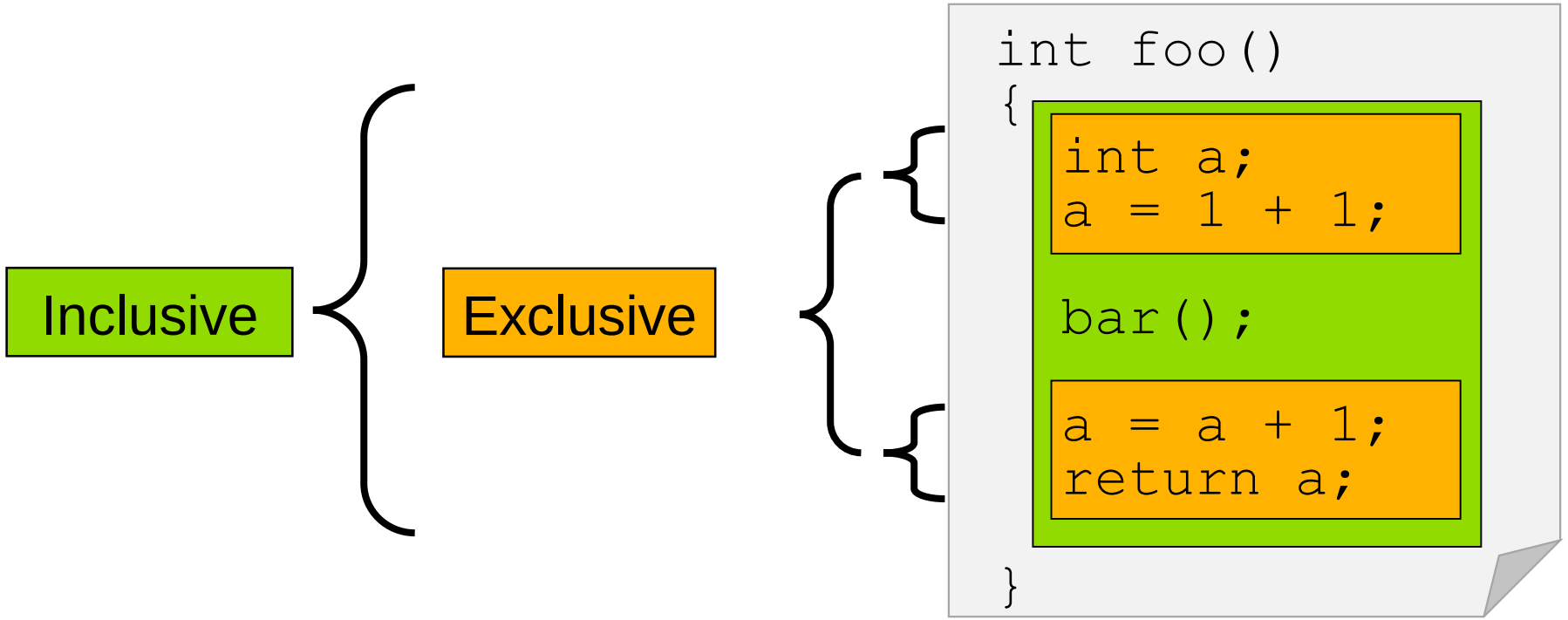


# Execution time

- 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



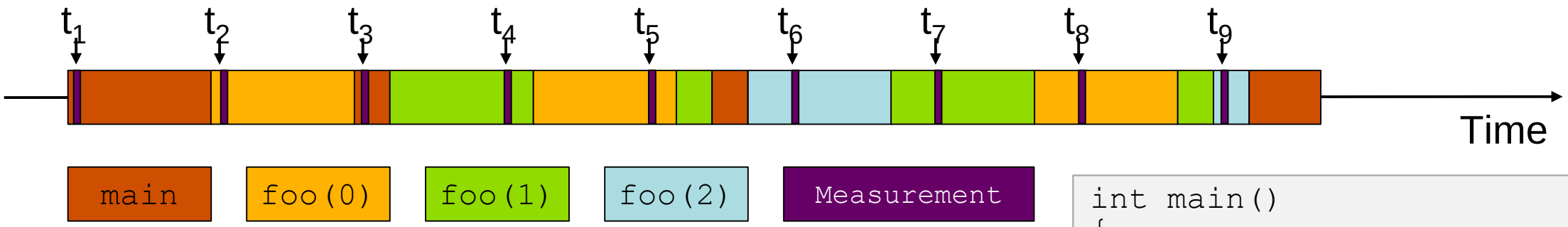
# Classification of measurement techniques

- **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

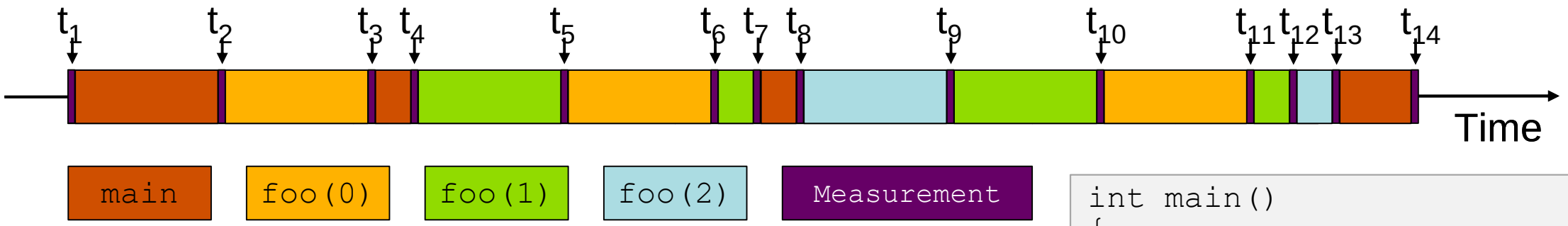
# Sampling




- 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 behaviour
  - 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 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

# 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*

# 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

# 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

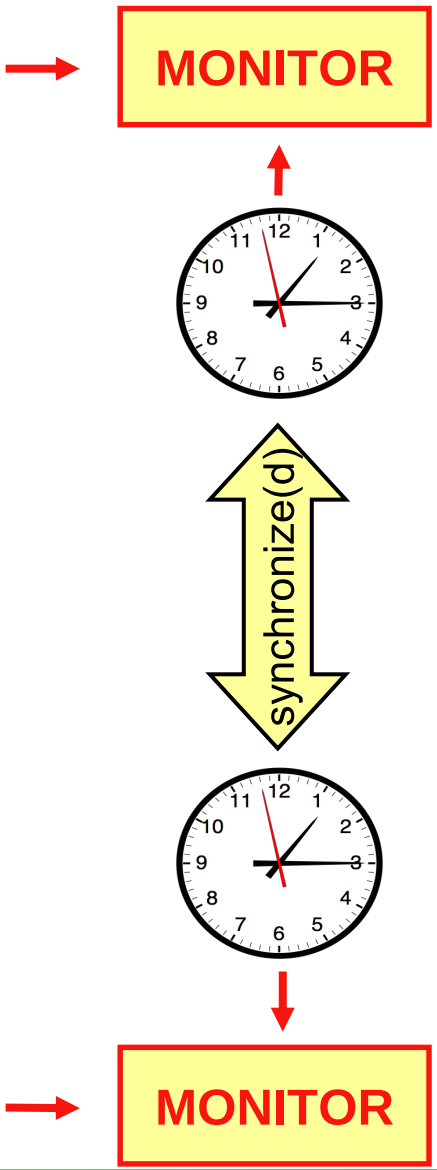
Process A

```
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 behaviour 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 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

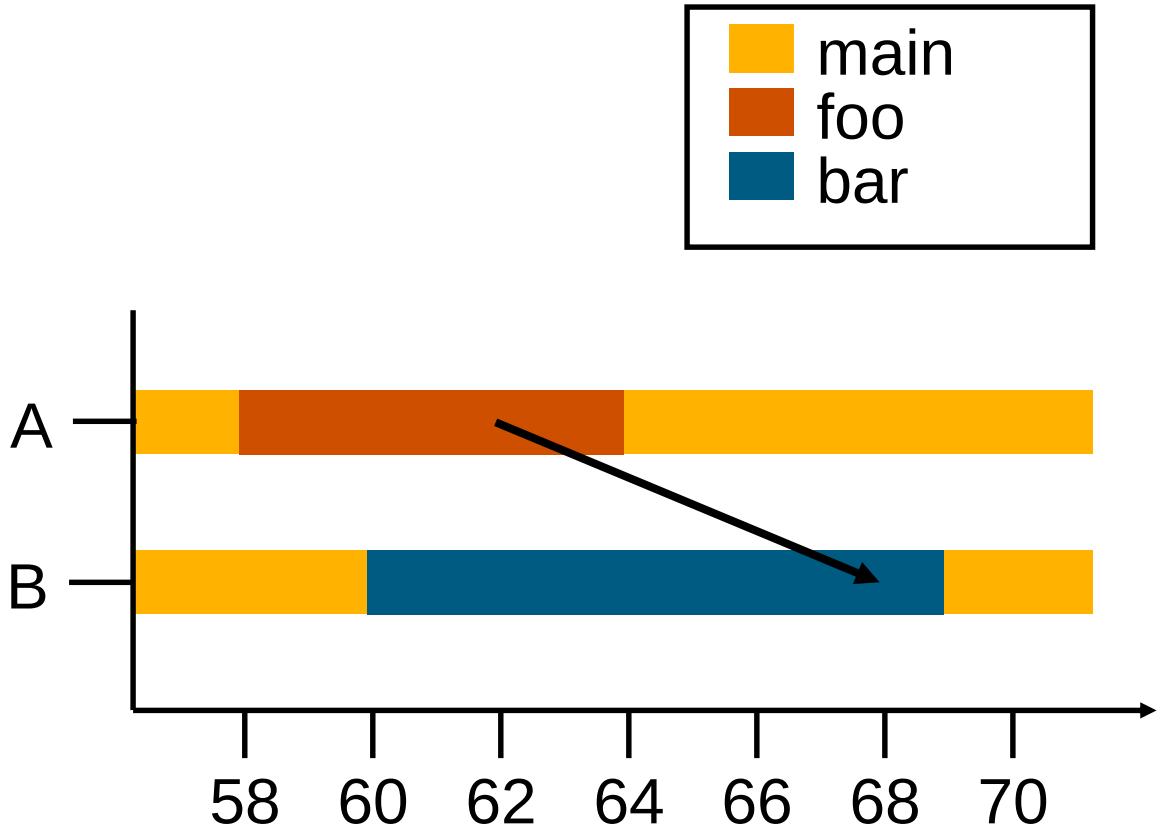
- Performance data is stored at end of measurement run
- Data analysis is performed afterwards
  - Automatic search for bottlenecks
  - Visual trace analysis
  - Calculation of statistics

# Example: Time-line visualization

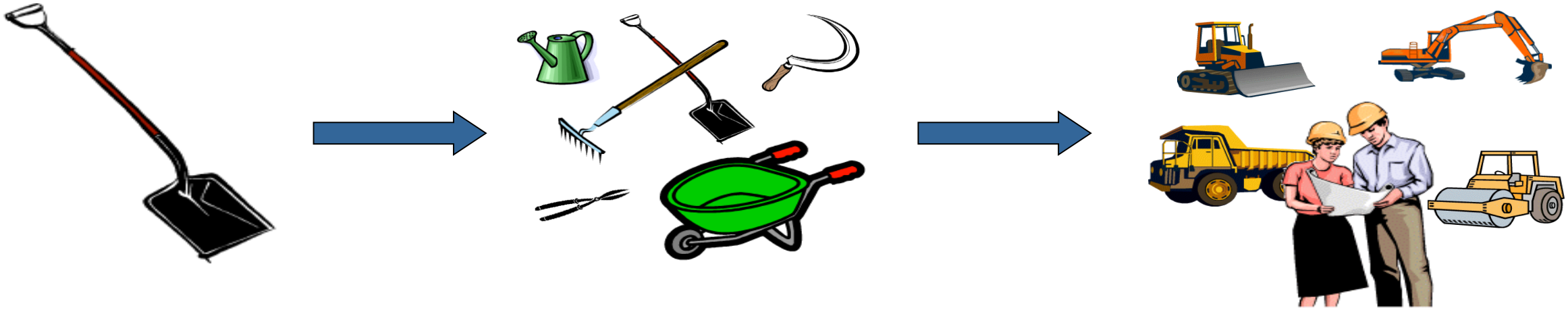
**Global** trace view

| ... | | |
|---|---|---|
| 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

# Virtual Institute – High Productivity Supercomputing

# 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
- Activities
  - Development and integration of HPC programming tools
    - Correctness checking & performance analysis
  - Academic workshops
  - Training workshops
  - Service
    - Support email lists
    - Application engagement

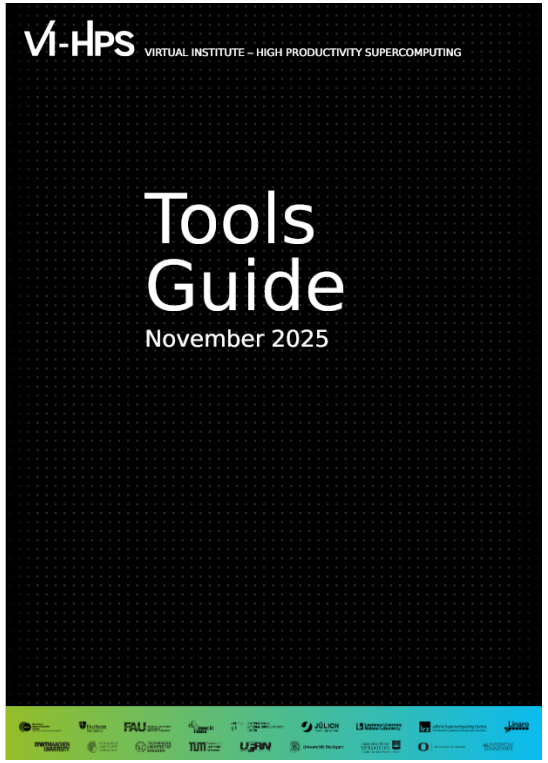
HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

**http://www.vi-hps.org**

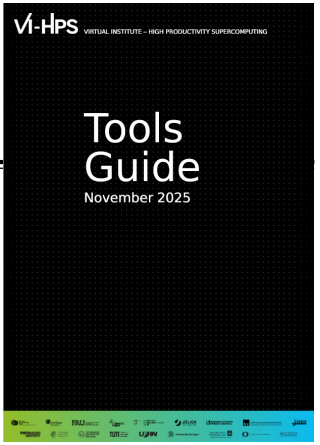
# Productivity tools (selected)

- **MAQAO**: Assembly analysis & optimization
- **Score-P:** Community-developed instrumentation & measurement infrastructure
- **Scalasca**: Large-scale parallel performance analysis
- **Extrae/Paraver/Dimemas**: Trace collection & analysis

For a brief overview of tools consult the VI-HPS Tools Guide:



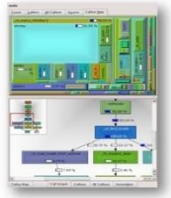
# Productivity tools (cont.)



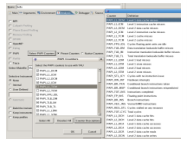

- Extra-P: Automated performance modelling
- JUBE: Workflow execution environment
- Kcachegrind: Callgraph-based cache analysis [x86 only]
- LIKWID: Command-line performance tools suite
- Linaro FORGE DDT/MAP/PR: Parallel debugging, profiling & performance reports
- MUST & Archer: MPI & OpenMP usage correctness checking
- mpiP/mpiPview: MPI profiling tool and analysis viewer
- Open MPI: Integrated memory checking
- Open|SpeedShop: Integrated parallel performance analysis environment
- PAPI: Interfacing to hardware performance counters
- STAT: Stack trace analysis tools
- TAU: Integrated parallel performance system
- Vampir: Interactive graphical trace visualization & analysis

# Technologies and their integration

CALLGRIND

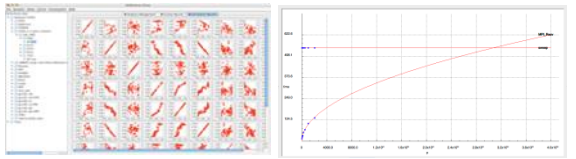


PAPI



MAP/PR / MPIP / O|SS / **MAQAO** / LIKWID

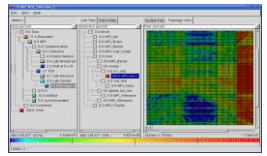
TAU EXTRA-P



Hardware monitoring

Automatic profile & trace analysis

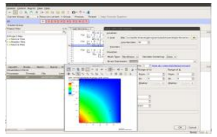
**SCALASCA**



**SCORE-P / EXTRAE**

MUST / ARCHER

DDT

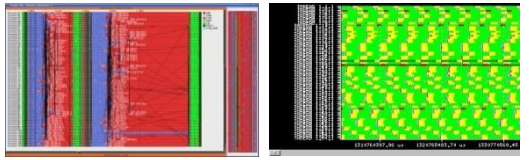


Debugging, error & anomaly detection

JUBE

Visual trace analysis

VAMPIR **PARAVER**



STAT

Execution

Optimization

MEMCHECKER

MAQAO

# Disclaimer

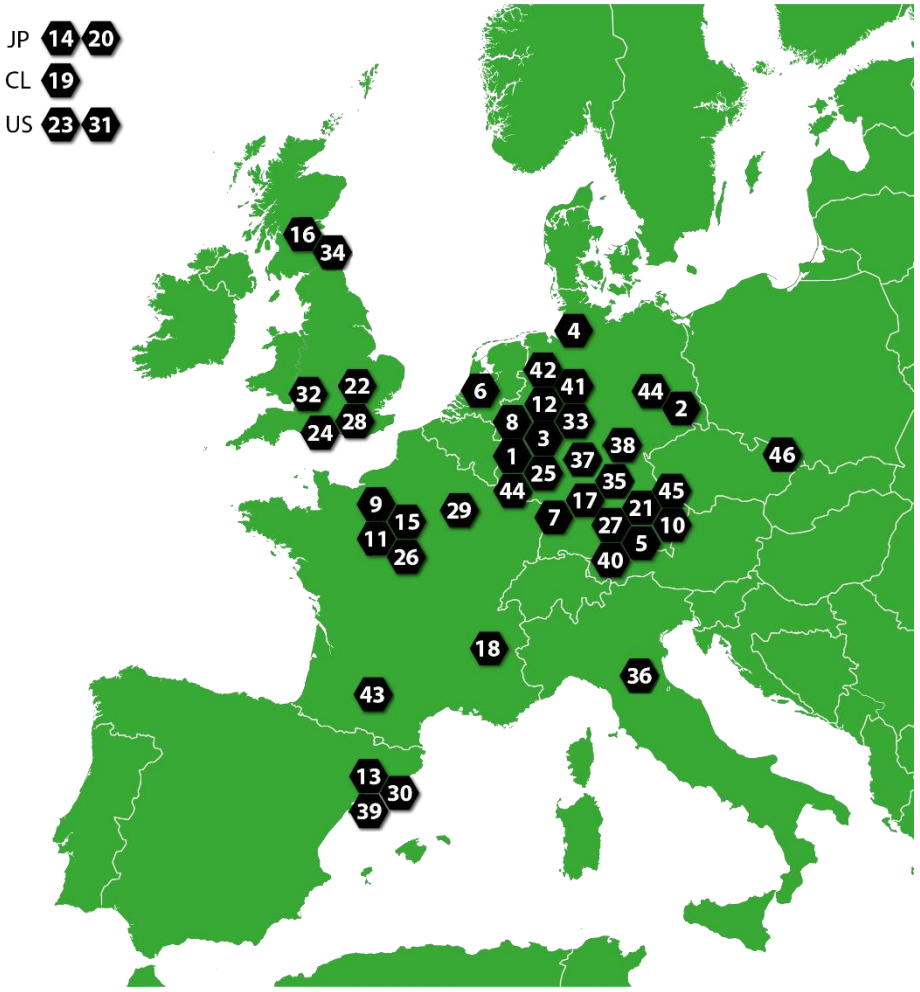
Tools will ***not*** automatically make you, your applications or computer systems more productive.

However, they can help you understand ***how*** your parallel code executes and ***when / where*** it's necessary to work on correctness and performance issues.

# VI-HPS training & Tuning Workshops

- Goals
  - Give an overview of the programming tools suite
  - Explain the functionality of individual tools
  - Teach how to use the tools effectively
  - Offer hands-on experience and expert assistance using tools
  - Receive feedback from users to guide future development
- For best results, bring & analyze/tune your own code(s)!
- VI-HPS Hands-on Tutorial series
  - SC'08-11/13/14/15/16/17/19/20/21/22/23, ICCS'09, Cluster'10, EuroMPI'12/14, XSEDE'13, ISC-HPC'15-19,21/22/23
- VI-HPS Tuning Workshop series
  - 2008 (x2), 2009 (x2), 2010 (x2), 2011 (x2), 2012 (x2), 2013 (x2), 2014(x4), 2015(x3)
  - 2016 (Kobe/Japan, Garching/Germany, Cambridge/UK, Livermore/USA)
  - 2017 (Southampton/UK, Aachen/Germany, Bruyères-le-Châtel/France)
  - 2018 (Garching/Germany, London/UK, Reims/France)
  - 2019 (Barcelona/Spain, Knoxville/USA, Bristol/UK, Jülich/Germany)
  - 2020 (EPCC/Scotland / *Online*, HLRS/Germany / *Online*, CINECA/Italy / *Online*, CSC/Germany / *Online*)
  - 2021 (NHR@FAU/Germany / *Online*, POP CoE / *Online*)
  - 2022 (JSC/RWTH/Germany / *Online*, POP CoE / *Online*)
  - 2024 (Toulouse/France, Aachen&Dresden/Germany / *Hybrid,* Garching b. München/Germany, Ostrava/Czechia / *Online*)
  - 2025 (Essen/Germany)
  - 2026 (BSC/Spain, Durham/UK)

# VI-HPS Tuning Workshop series




1. 2008/03/05+3: RWTH, Aachen, Germany
2. 2008/10/08+3: ZIH, Dresden, Germany
3. 2009/02/16+5: JSC, Jülich, Germany
4. 2009/09/09+3: HLRN, Bremen, Germany
5. 2010/03/08+3: TUM, Garching, Germany
6. 2010/05/26+3: SARA, Amsterdam, Netherlands
7. 2011/03/28+3: HLRS, Stuttgart, Germany
8. 2011/09/05+5: GRS, Aachen, Germany
9. 2012/04/23+5: UVSQ, St-Quentin, France
10. 2012/10/16+4: LRZ, Garching, Germany
11. 2013/04/22+4: MdS, Saclay, France
12. 2013/10/07+5: JSC, Jülich, Germany
13. 2014/02/10+5: BSC, Barcelona, Spain
14. 2014/03/25+3: RIKEN AICS, Kobe, Japan
15. 2014/04/07+4: MdS, Saclay, France
16. 2014/04/29+3: EPCC, Edinburgh, Scotland
17. 2015/02/23+5: HLRS, Stuttgart, Germany
18. 2015/05/18+5: UGA, Grenoble, France
19. 2015/10/27+3: NLHPC, Santiago, Chile
20. 2016/02/24+3: RIKEN AICS, Kobe, Japan
21. 2016/04/18+5: LRZ, Garching, Germany
22. 2016/07/06+3: Uni. Cambridge, England
23. 2016/07/27+3: LLNL, Livermore, California, USA
24. 2017/02/08+3: Uni. Southampton, England
25. 2017/03/27+5: RWTH, Aachen, Germany
26. 2017/10/16+5: Lab. ECR, Ter@tec, France
27. 2018/04/23+5: LRZ, Garching, Germany
28. 2018/06/21+3: UCL, London, England
29. 2018/10/15+5: ROMEO, Reims, France
30. 2019/01/21+5: BSC, Barcelona, Spain
31. 2019/04/09+4: UTK-ICL, Knoxville/TN, USA
32. 2019/04/24+3: Uni. Bristol, England
33. 2019/06/24+5: JSC, Jülich, Germany
34. 2020/07/28+3: EPCC, *Scotland* / ONLINE
35. 2020/09/14+5: HLRS, *Germany* / ONLINE
36. 2020/09/30+3: CINECA, *Italy* / ONLINE
37. 2020/12/07+5: CSC, *Germany* / ONLINE
38. 2021/03/01+3: NHR@FAU, *Germany* / ONLINE
39. 2021/04/19+3: POP CoE / ONLINE
40. 2021/06/14+5: LRZ, *Germany* / ONLINE
41. 2022/02/07+4: JSC/RWTH, *Germany* / ONLINE
42. 2022/05/17+3: POP CoE / ONLINE
43. 2024/01/29+3: CALMIP, Toulouse, France
44. 2024/02/26+3: RWTH Aachen & ZIH, TU Dresden, Germany / Hybrid
45. 2024/06/10+3: LRZ, Garching, Germany
46. 2024/09/04+2: IT4I, Ostrava, Czechia / ONLINE
47. 2025/06/02+3: Uni. Duisburg-Essen, Germany
48. 2026/02/09+5: BSC, Spain
49. 2026/04/27+3: Durham Uni., England

# Upcoming events

- 50th Tuning Workshop (GWDG Academy, Göttingen, Germany, 31 Aug - 04 Sept 2026)
- 51st Tuning Workshop (JSC, Jülich, Germany, Autumn 2026)
- 52nd Tuning Workshop (U. Wuppertal, Germany, 22-26 February 2027)
- 53rd Tuning Workshop (HLRS, Stuttgart, Germany, 05-09 April 2027)

- Future events to be determined
  - (one-day) tutorials: with guided exercises
  - (multi-day) training workshops: with your own applications on actual HPC systems
- Check **www.vi-hps.org/training** for announced events

- Contact us if you might be interested in hosting a training event