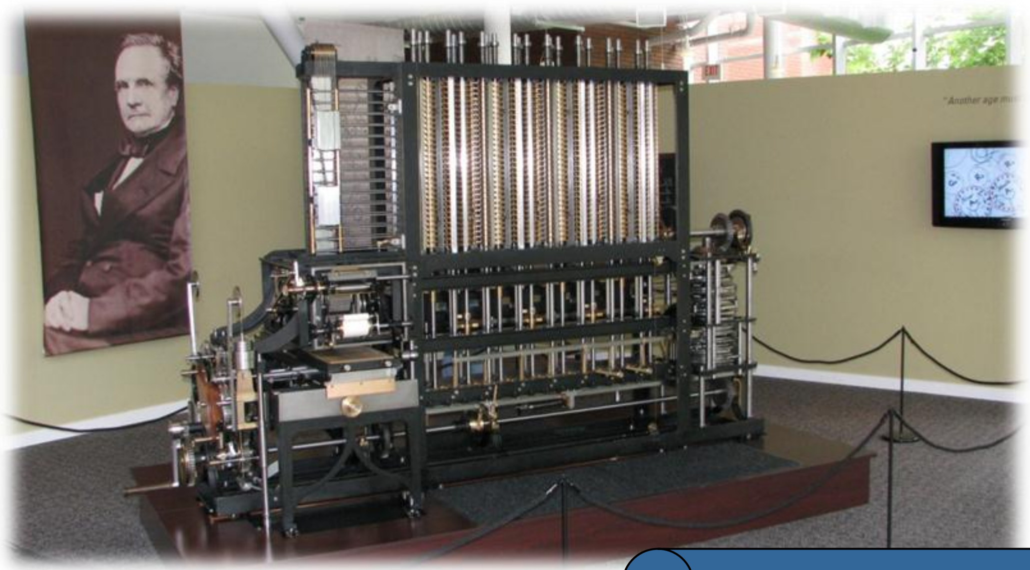
# Introduction to Performance Engineering

Markus Geimer
Jülich Supercomputing Centre

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

# Performance: an old problem

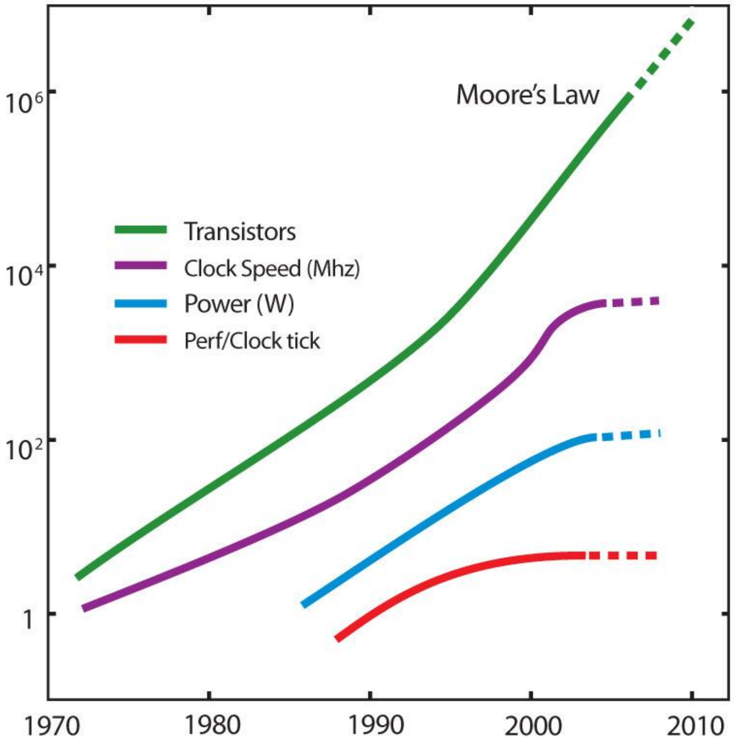


"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

Charles Babbage
1791 – 1871

# Today: the "free lunch" is over

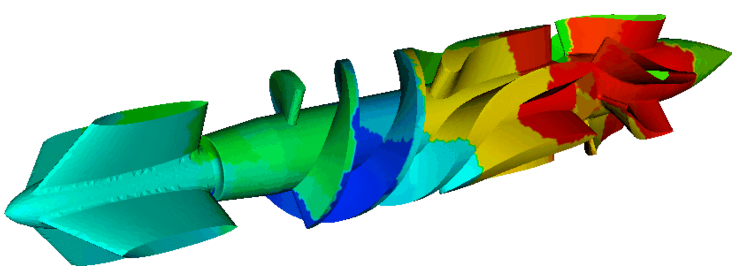
- Moore's law is still in charge, but
  - Clock rates no longer increase
  - Performance gains only through increased parallelism
- Optimizations of applications more difficult
  - Increasing application complexity
    - Multi-physics
    - Multi-scale
  - Increasing machine complexity
    - Hierarchical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



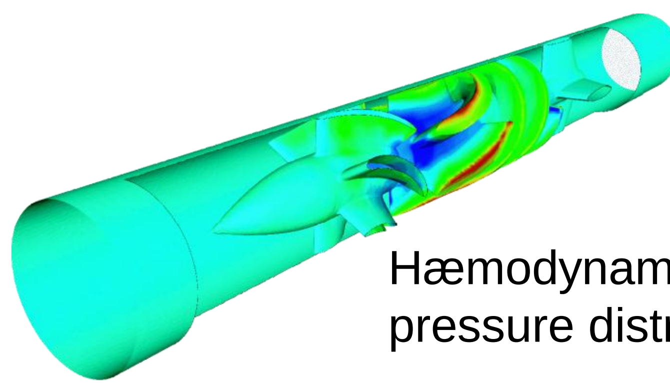

☞ Every doubling of scale reveals a new bottleneck!

# Example: XNS

- CFD simulation of unsteady flows
  - Developed by CATS / RWTH Aachen
  - Exploits finite-element techniques, unstructured 3D meshes, iterative solution strategies
- MPI parallel version
  - >40,000 lines of Fortran & C
  - DeBakey blood-pump data set (3,714,611 elements)

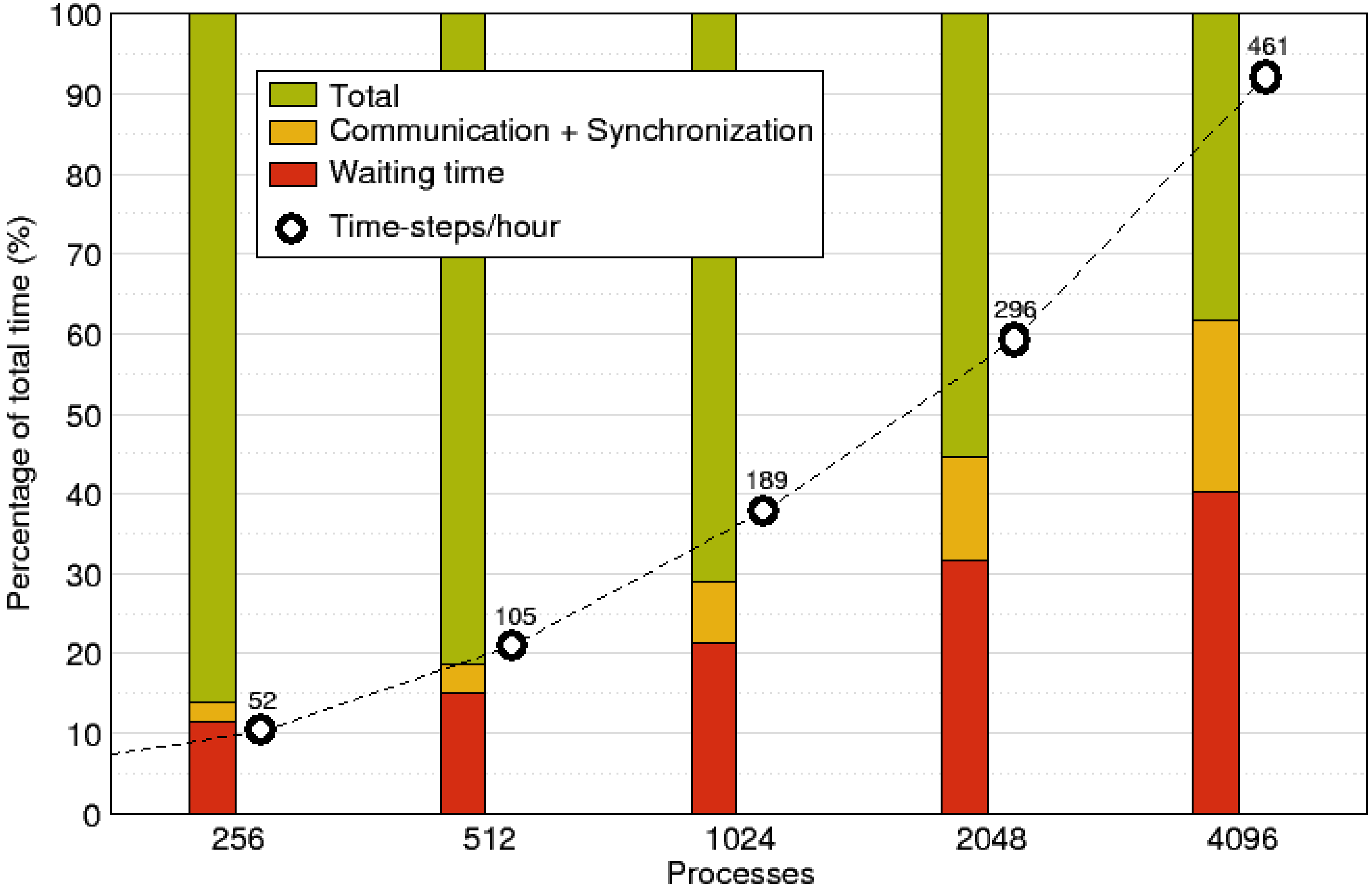


Partitioned finite-element mesh



Hæmodynamic flow pressure distribution

# XNS wait-state analysis on BG/L (2007)



# Performance factors of parallel applications

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

# Tuning basics

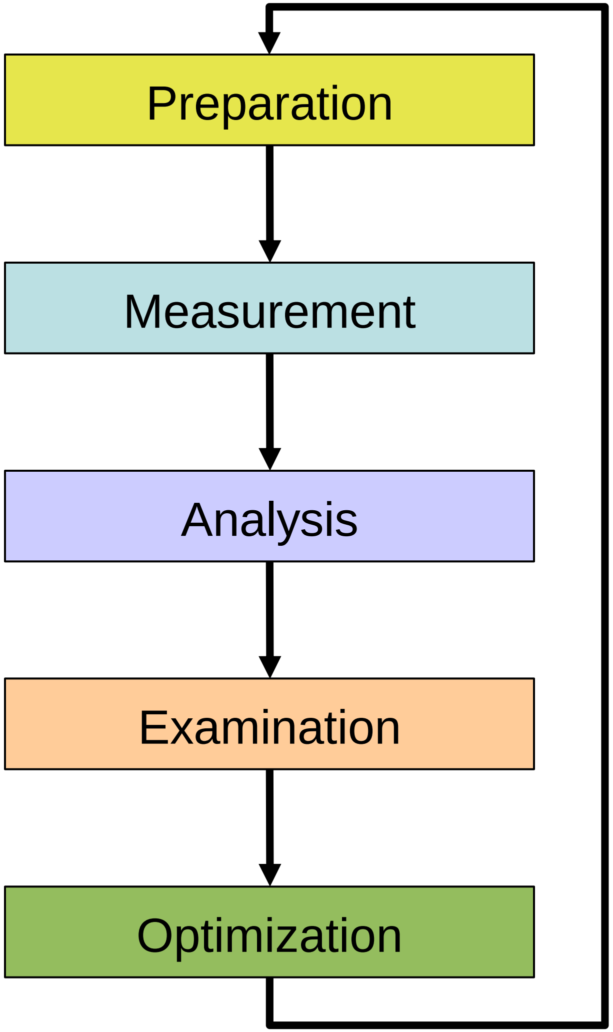
- Successful engineering is a combination of
  - The right algorithms and libraries
  - Compiler flags and directives
  - Thinking !!!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - ☞After each step!

## However…

> "We should forget about small efficiencies, say 97% of the time: premature optimization is the root of all evil."
>
> Charles A. R. Hoare

- It's easier to optimize a slow correct program than to debug a fast incorrect one
  - ☞ *Nobody cares how fast you can compute a wrong answer...*

# 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
- 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

# 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

“math” Operations?
HW Operations?
HW Instructions?
32-/64-bit? …

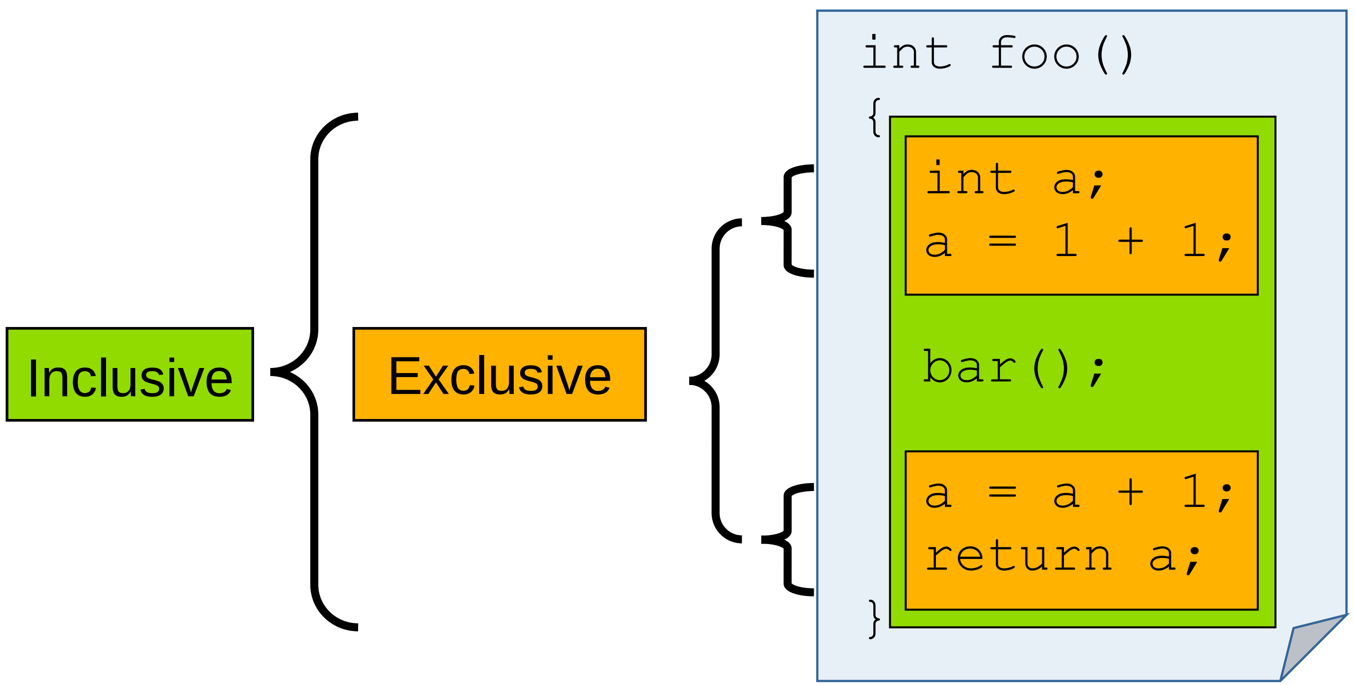
# Execution time

- 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

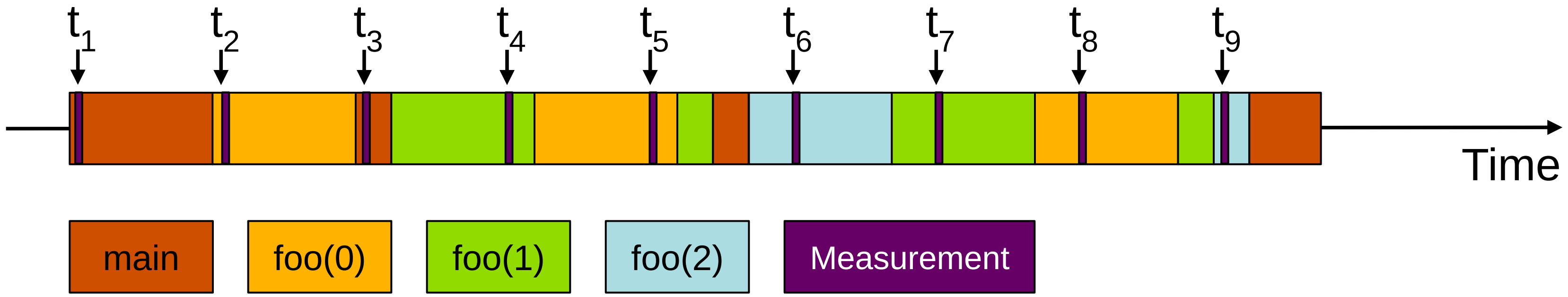


# Classification of measurement techniques

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

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

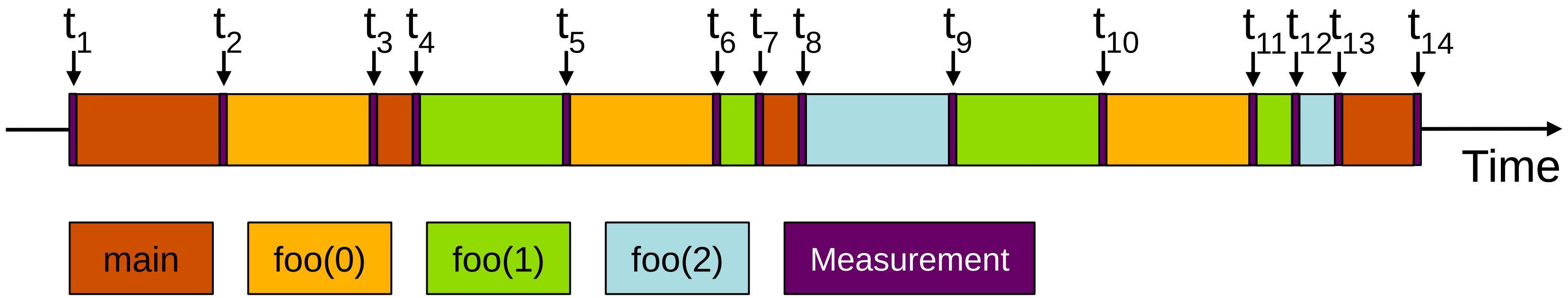
# Sampling




```
int main()
{
  int i;

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

  return 0;
}

void foo(int i)
{

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

}
```

- 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

# Instrumentation




```
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");
}
```

- 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

# 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

# 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

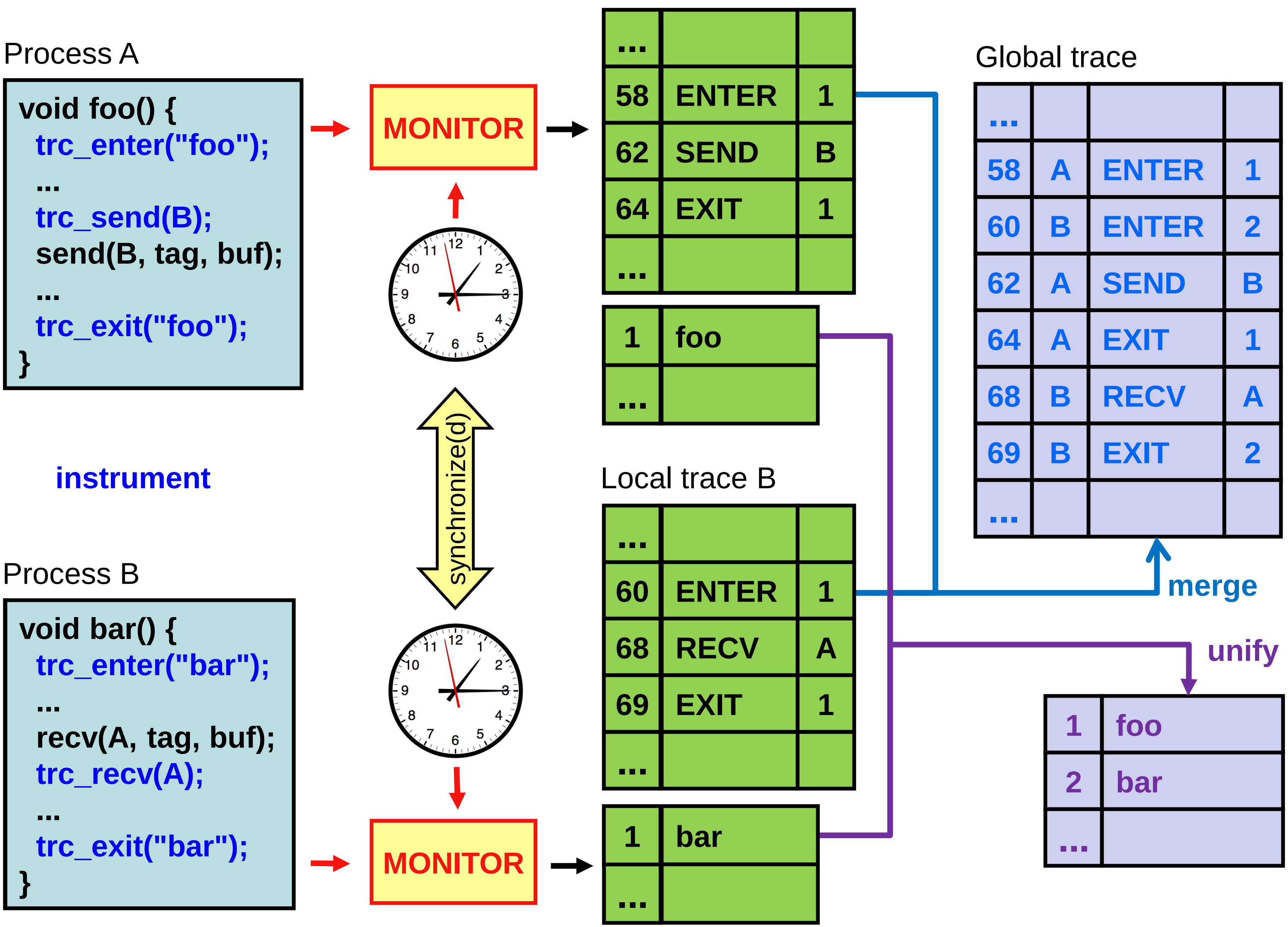
- Flat profile
  - Shows distribution of metrics per routine / instrumented region
  - Calling context is not taken into account
- Call-path profile
  - Shows distribution of metrics per executed call path
  - Sometimes only distinguished by partial calling context (e.g., two levels)
- Special-purpose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

# Tracing

- Recording 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*

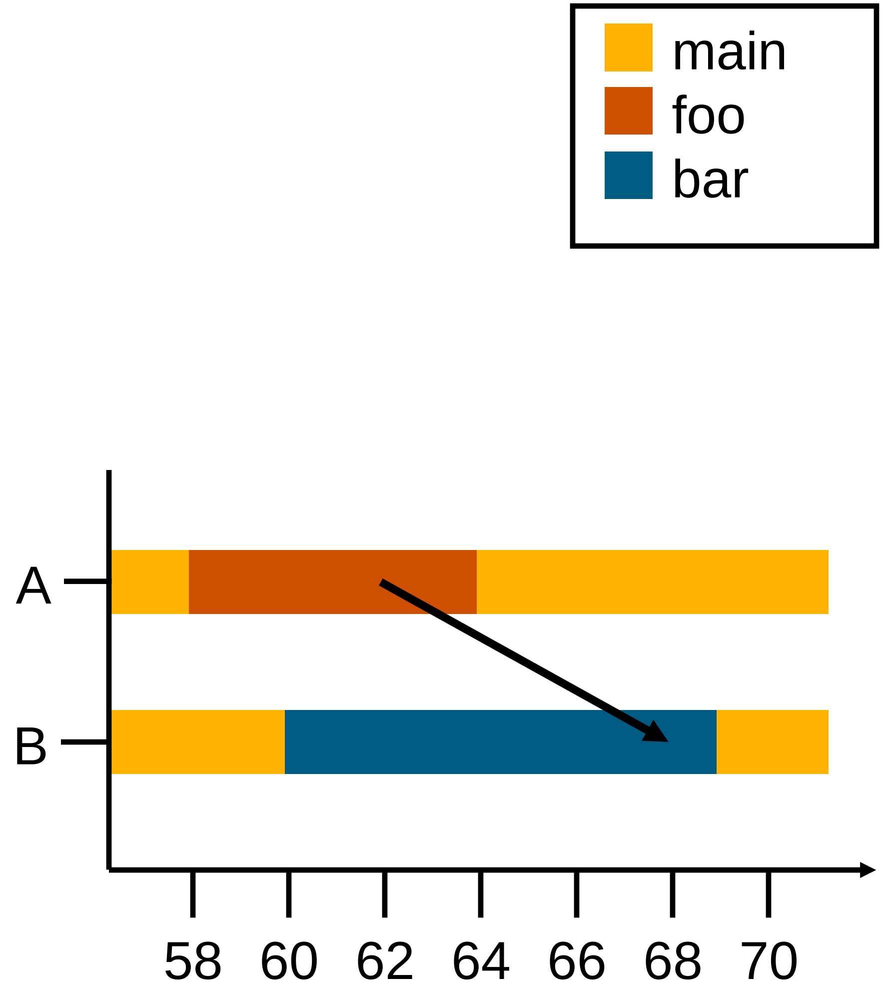
# Event tracing



# Example: Time-line 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 |
| ... | | | |



# Tracing vs. Profiling

- 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
  - Writing tracing software is complicated
    - Event buffering, clock synchronization, ...

# 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

# Performance Analysis Tools Overview

Markus Geimer
Jülich Supercomputing Centre

# Fall-back: “Home-grown performance tool”

- Simple measurements can always be performed
  - C/C++: times(), clock(), gettimeofday(), clock_gettime(), getrusage()
  - Fortran: etime, cpu_time, system_clock
- However, ...
  - Use these functions rarely and only for coarse-grained timings
  - Avoid do-it-yourself solutions for detailed measurements
  - Use dedicated tools instead
    - Typically more powerful
    - Might use platform-specific timers with better resolution and/or lower overhead

# GNU gprof

- Shows where the program spends its time
  - Indicates which functions are candidates for tuning
  - Identifies which functions are being called
  - Can be used on large and complex programs
- Method
  - Compiler inserts code to count each function call (compile with “-pg”)
    - Shows up in profile as “__mcount”
  - Time information is gathered using sampling
    - Current function and and its parents (two levels) are determined
  - Program execution generates “gmon.out” file
  - To dump human-readable profile to stdout, call

```
gprof <executable> <gmon.out file>
```

# GNU gprof: flat profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 82.90     31.55    31.55    27648     1.14     1.14  pc_jac2d_blk3_
  7.54     34.42     2.87        1  2870.00 37910.00  cg3_blk_
  5.96     36.69     2.27    28672     0.08     0.08  matxvec2d_
  2.13     37.50     0.81    55296     0.01     0.01  dot_prod2d_
  0.87     37.83     0.33       54     6.11     6.11  add_exchange2d_
  0.34     37.96     0.13                             main
  0.21     38.04     0.08       27     2.96     2.96  cs_jac2d_blk3_
...
```

Avg. inclusive time

Avg. exclusive time

Number of calls

Exclusive time
(sorting criterion)

Time sum

Percentage of total time

# GNU gprof: (partial) call-path profile

```
                              called/total     parents
index %time   self descendents  called+self    name           index
                              called/total     children

              0.17        2.83      1/1              .__start [2]
[1]     99.3  0.17        2.83      1            .main [1]
              2.83        0.00     10/10             .relax [3]
              0.00        0.00      3/13             .printf [15]
              0.00        0.00      1/1              .saveOutput [27]
              0.00        0.00      2/1523           .malloc [5]
              0.00        0.00      1/1              .readInput [40]
              0.00        0.00     10/20             .update [68]

...
```

Percentage of total time (sorting criterion)

Exclusive time

Time spent in children

Number of calls from this callsite

Total number of calls

Children (Called functions)

Parent (Callsite)

# MPI profiling: mpiP

- Scalable, light-weight MPI profiling library
- Generates detailed text summary of MPI behavior
  - Time spent in each MPI function call site
  - Bytes sent by each MPI function call site (if applicable)
  - MPI I/O statistics
  - Configurable traceback depth for function call sites
- Controllable from program using MPI_Pcontrol()
  - Allows to profile just one code module or a single iteration
- Uses standard PMPI interface: only re-linking of application necessary
- License: new BSD
- Web site: http://mpip.sourceforge.net

# mpiP: text output example

```
@ mpiP
@ Version: 3.1.1
// 10 lines of mpiP and experiment configuration options
// 8192 lines of task assignment to BlueGene topology information

@--- MPI Time (seconds) ---------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       37.7       25.2    66.89
// ...
8191       37.6         26    69.21
   *   3.09e+05   2.04e+05    65.88

@--- Callsites: 26 --------------------------------------------------------
 ID Lev File/Address     Line Parent_Funct             MPI_Call
  1   0 coarsen.c         542 hypre_StructCoarsen      Waitall
// 25 similiar lines

@--- Aggregate Time (top twenty, descending, milliseconds) --------
Call                 Site       Time    App%    MPI%     COV
Waitall                21   1.03e+08   33.27   50.49    0.11
Waitall                 1   2.88e+07    9.34   14.17    0.26
// 18 similiar lines
```

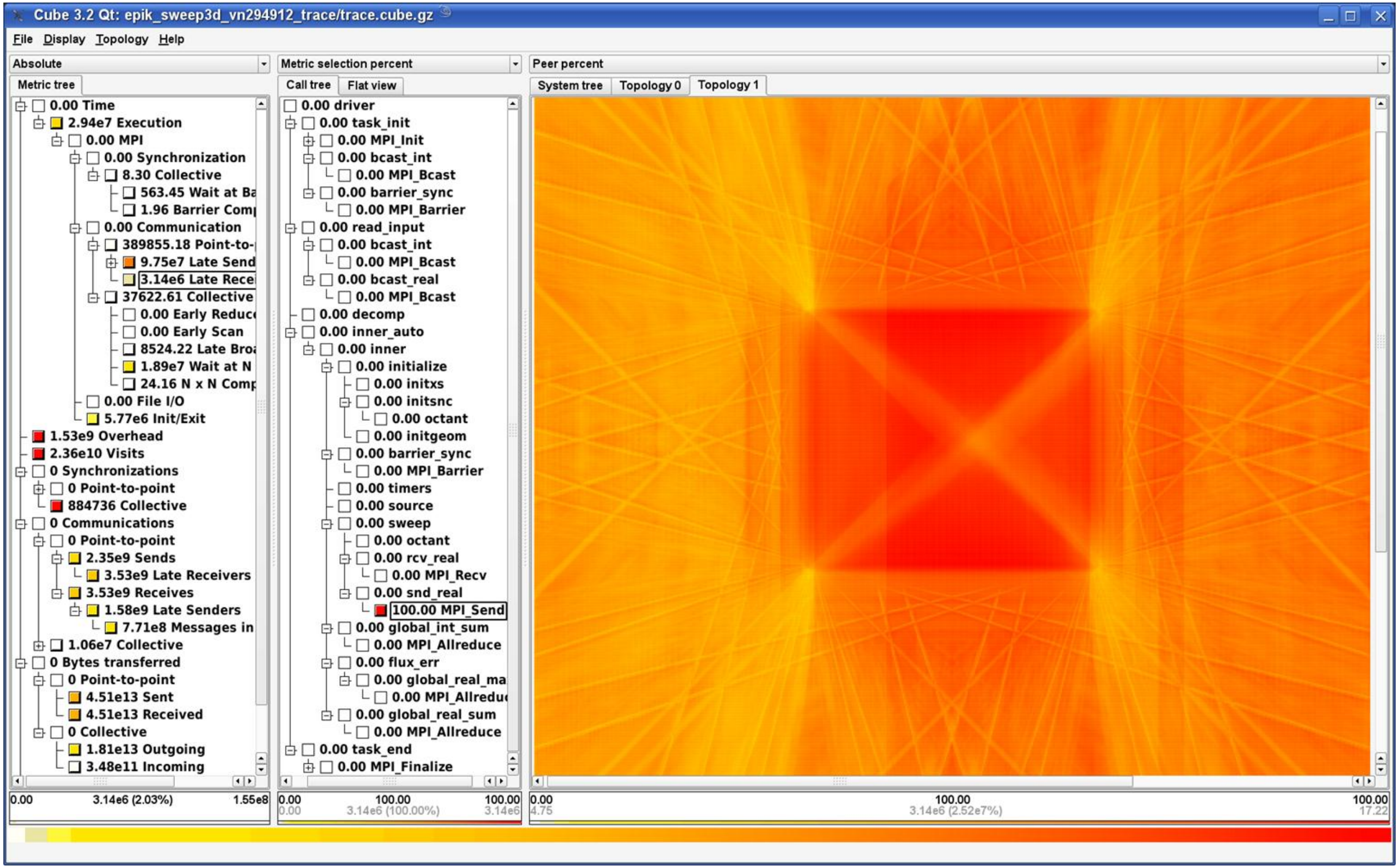
# mpiP: text output example (cont.)

```
@--- Aggregate Sent Message Size (top twenty, descending, bytes) --
Call                     Site      Count        Total        Avrg  Sent%
Isend                      11  845594460     7.71e+11         912  59.92
Allreduce                  10      49152     3.93e+05           8   0.00
// 6 similiar lines

@--- Callsite Time statistics (all, milliseconds): 212992 ---------
Name          Site Rank     Count     Max    Mean       Min   App%    MPI%
Waitall         21    0    111096     275     0.1  0.000707  29.61   44.27
//   ...
Waitall         21 8191     65799     882    0.24  0.000707  41.98   60.66
Waitall         21    * 577806664     882   0.178  0.000703  33.27   50.49
// 213,042 similiar lines

@--- Callsite Message Sent statistics (all, sent bytes) -----------
Name          Site Rank     Count        Max      Mean    Min        Sum
Isend           11    0     72917  2.621e+05     851.1      8  6.206e+07
//...
Isend           11 8191     46651  2.621e+05      1029      8  4.801e+07
Isend           11    * 845594460  2.621e+05     911.5      8  7.708e+11
// 65,550 similiar lines
```

# Scalasca

- Scalable performance-analysis toolkit for parallel codes
  - Specifically targeting large-scale applications running on 10,000s to 100,000s of cores
- Integrated performance-analysis process
  - Performance overview via call-path profiles
  - In-depth study of application behavior via event tracing
    - Automatic trace analysis identifying wait states
  - Switching between both options without re-compilation or re-linking
- Supports MPI 2.2 and basic OpenMP
- License: new BSD
- Website: http://www.scalasca.org

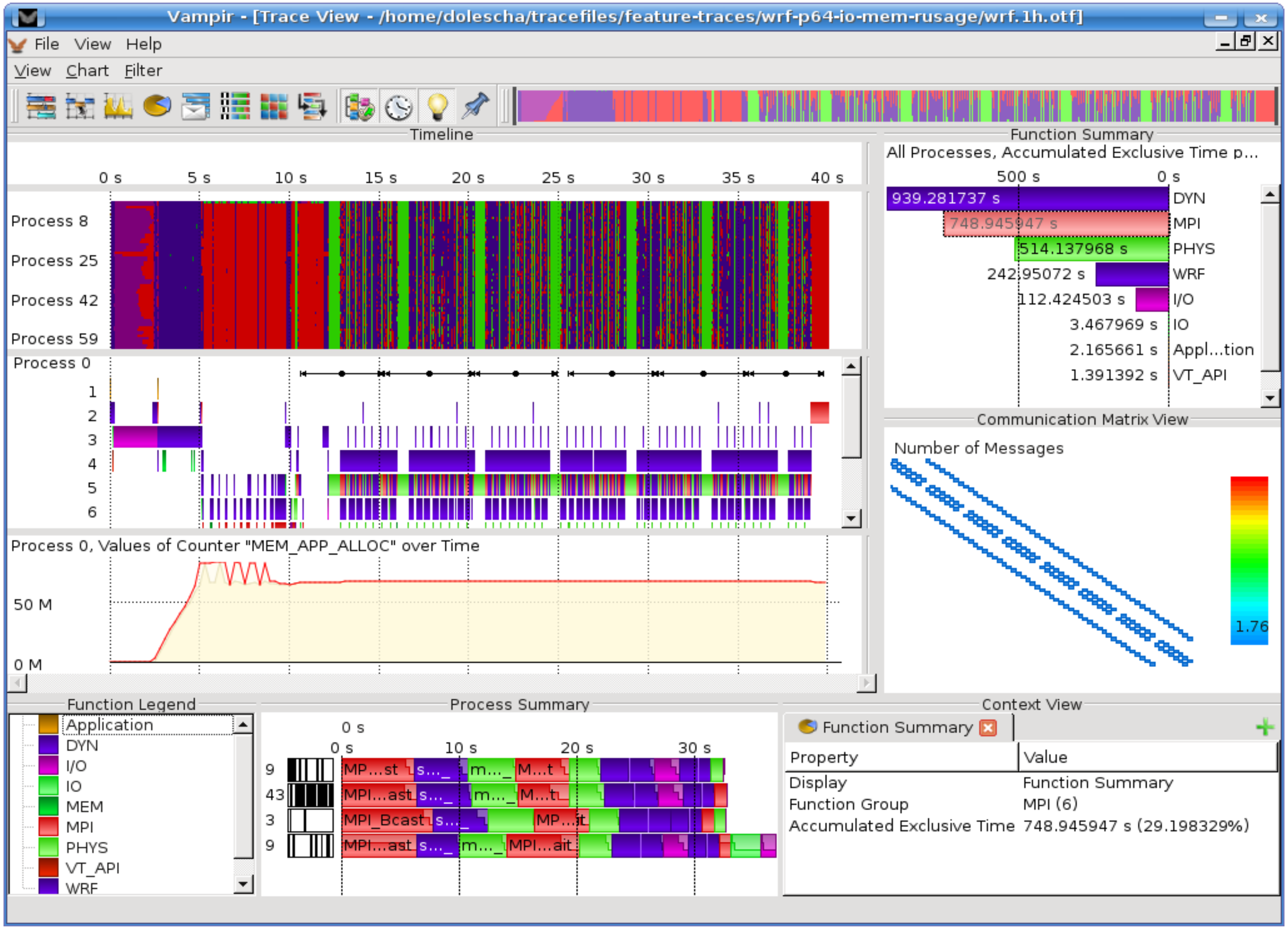
# Scalasca Sweep3D trace analysis on BG/P @ 288k CPUs




**More after the coffee break …**

- VampirTrace
  - Tool set and runtime library to generate OTF traces
  - Supports MPI, OpenMP, POSIX threads, Java, CUDA, ...
  - Also distributed as part of Open MPI since v1.3
  - License: new BSD
  - Web site: http://www.tu-dresden.de/zih/vampirtrace
- Vampir / VampirServer
  - Interactive trace visualizer with powerful statistics
  - License: commercial
  - Web site: http://www.vampir.eu

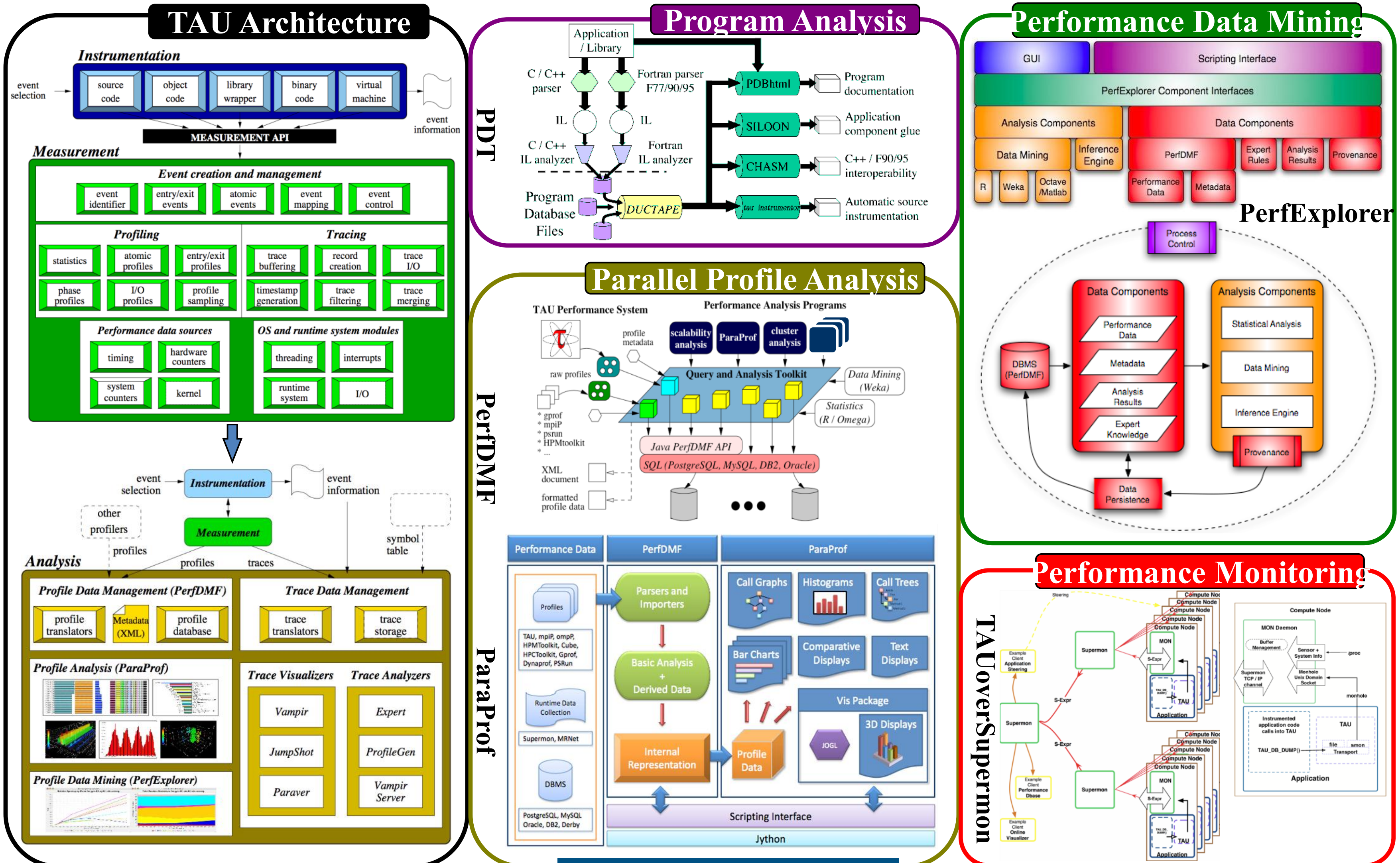
# Vampir displays overview




More after the lunch break …

- Very portable tool set for instrumentation, measurement and analysis of parallel applications
- The “swiss army knife” of performance analysis
- Instrumentation API supports choice
  - between profiling and tracing
  - of metrics (e.g., time, HW counter, ...)
- Supports
  - C, C++, Fortran, HPF, HPC++, Java, Python
  - MPI, OpenMP, POSIX threads, Java, Win32, ...
- License: Open-source (historical permission notice and disclaimer)
- Web site: http://tau.uoregon.edu

# TAU components




More tomorrow …

# Other open-source tools worth mentioning

- Open|SpeedShop
  - Modular and extensible performance analysis tool set
  - Web site: http://www.openspeedshop.org
- HPCToolkit
  - Multi-platform statistical profiling package
  - Web site: http://hpctoolkit.org
- Paraver
  - Trace-based performance-analysis and visualization framework
  - Web site: http://www.bsc.es/paraver
- PerfSuite, Periscope, IPM, ...

# Commercial / vendor-specific tools

- Intel VTune (serial/multi-threaded)
- Intel Trace Analyzer and Collector (MPI)
- Cray Performance Toolkit: CrayPat / Apprentice2
- IBM HPC Toolkit
- Oracle Performance Analyzer
- Acumem ThreadSpotter
- ...