JÜLICH
FORSCHUNGSZENTRUM

Performance Analysis 1

January 2015 | Michael Knobloch

Outline

- This lecture:
  - Basic concepts of performance analysis
  - Sampling & Instrumentation
  - Profiling & Tracing
  - Performance analysis with Score-P
  - Tool overview

- Next lecture (25.01.2016)
  - Trace analysis in detail
  - Automatic analysis with Scalasca
  - Manual analysis with Vampir

Motivation

Make it work,
make it right,
make it fast.

Kent Beck

Premature optimization is the root of all evil.

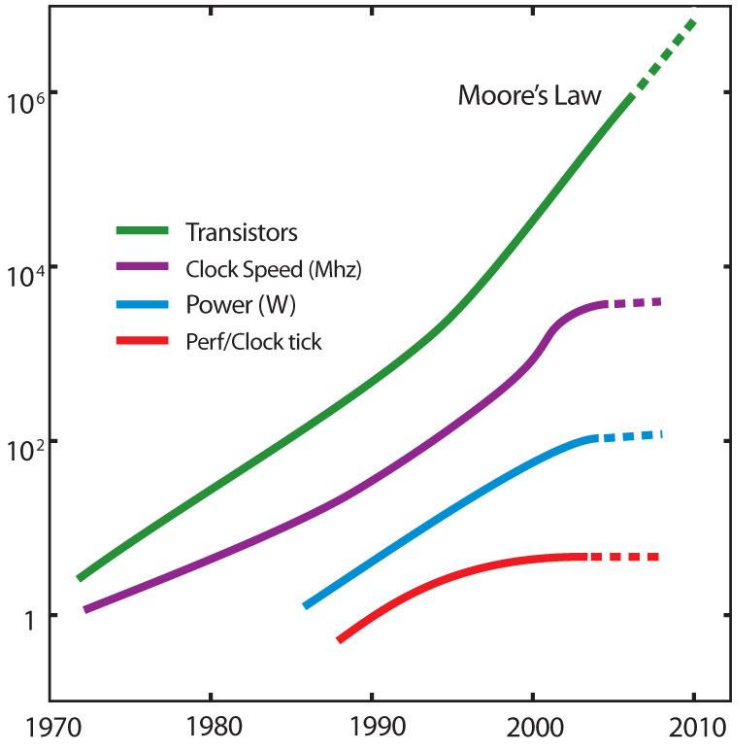
Donald E. Knuth

If you optimize everything, you will always be unhappy.

Donald E. Knuth

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!

Performance factors of parallel applications

- “Sequential” factors
  - Computation
    - ☞Choose right algorithm, use optimizing compiler
  - Vectorization
    - ☞ Especially important on many-core architectures
  - Cache and memory
    - ☞Tough! Only limited tool support, hope compiler gets it right
  - Input / output
    - ☞Often not given enough attention

Performance factors of parallel applications

JÜLICH FORSCHUNGSZENTRUM

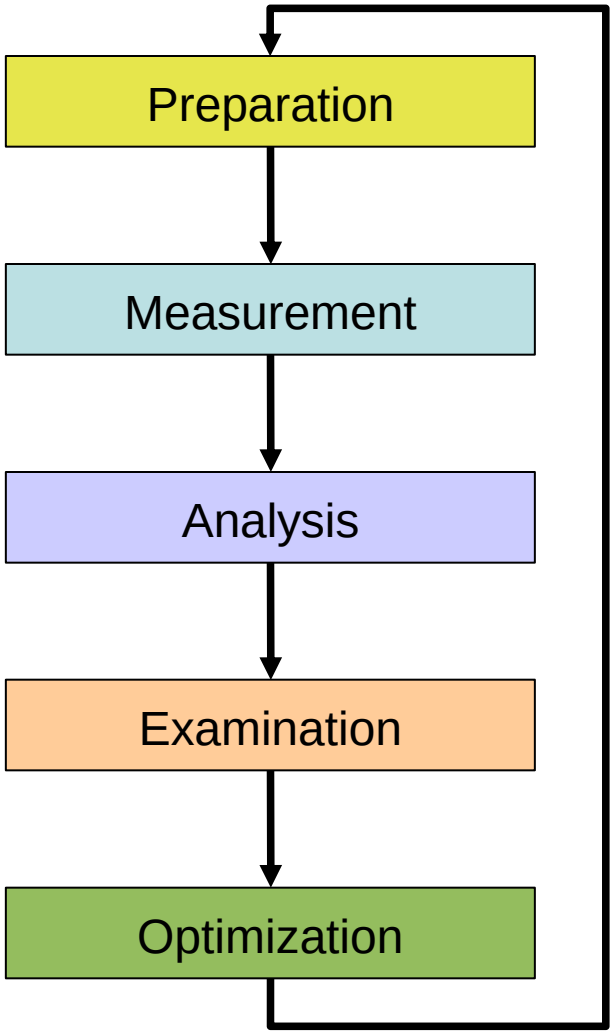
- “Parallel” factors
  - Partitioning / decomposition
    - ☞ Load balancing
  - Communication (i.e., message passing)
  - Multithreading
  - Synchronization / locking
    - ☞ More or less understood, good tool support

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!

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

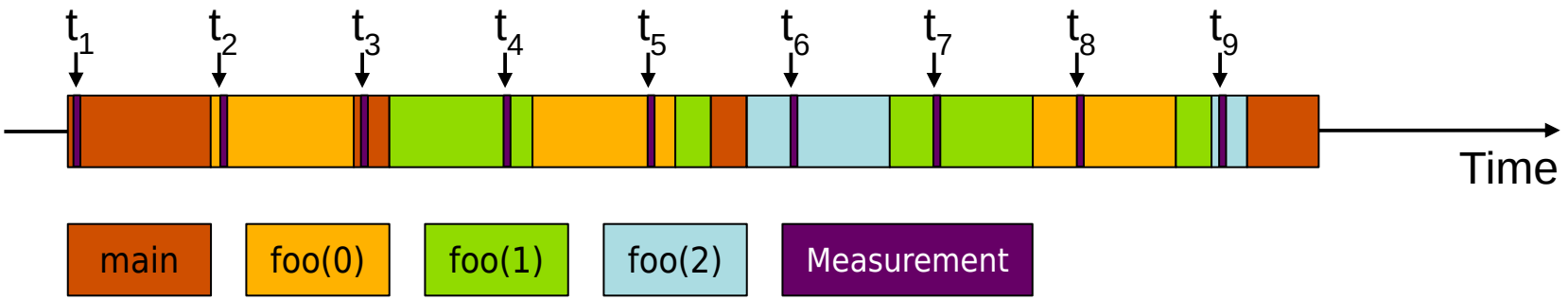
The 80/20 rule

- 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!*

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

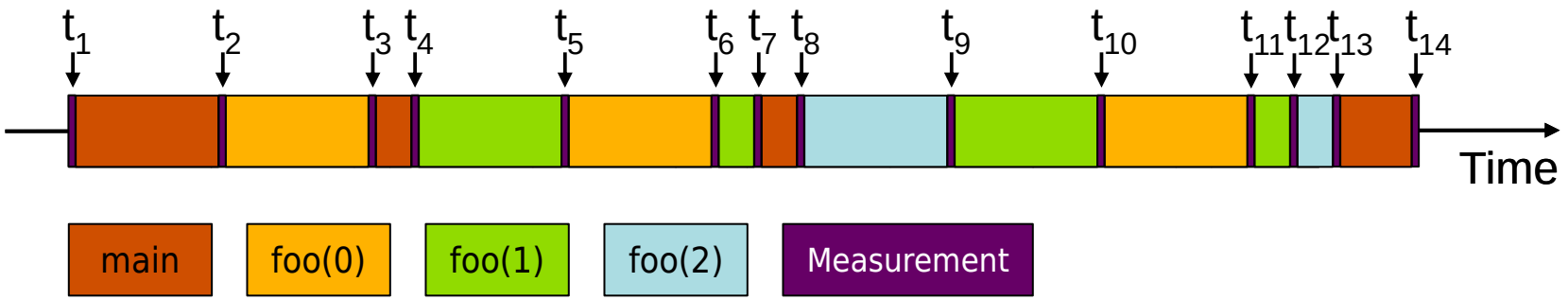
Sampling




```
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 behavior
  - 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*

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

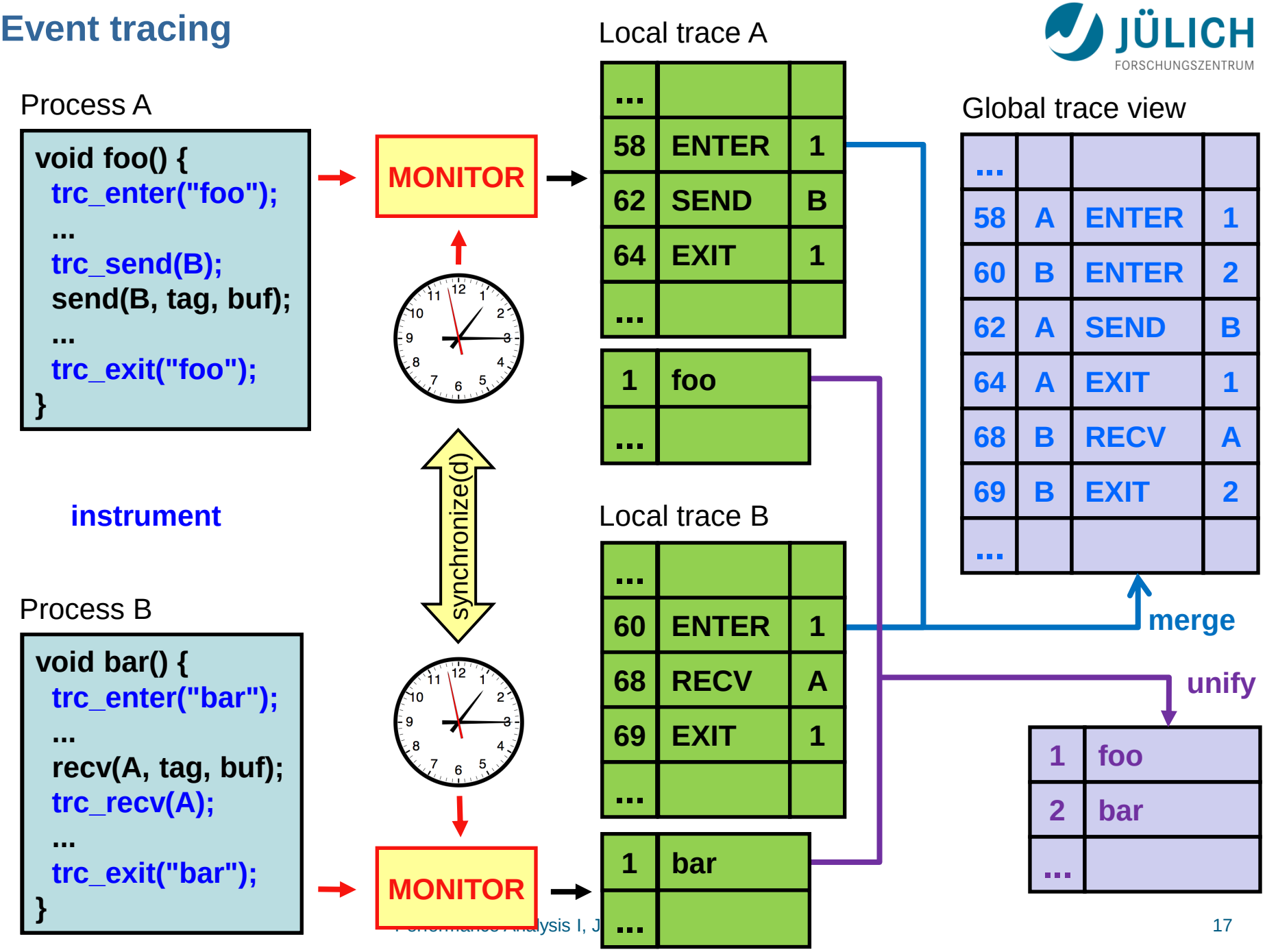
☞ *Profile = summarization of events over execution interval*

Tracing

- Recording 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



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, ...

Online analysis

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

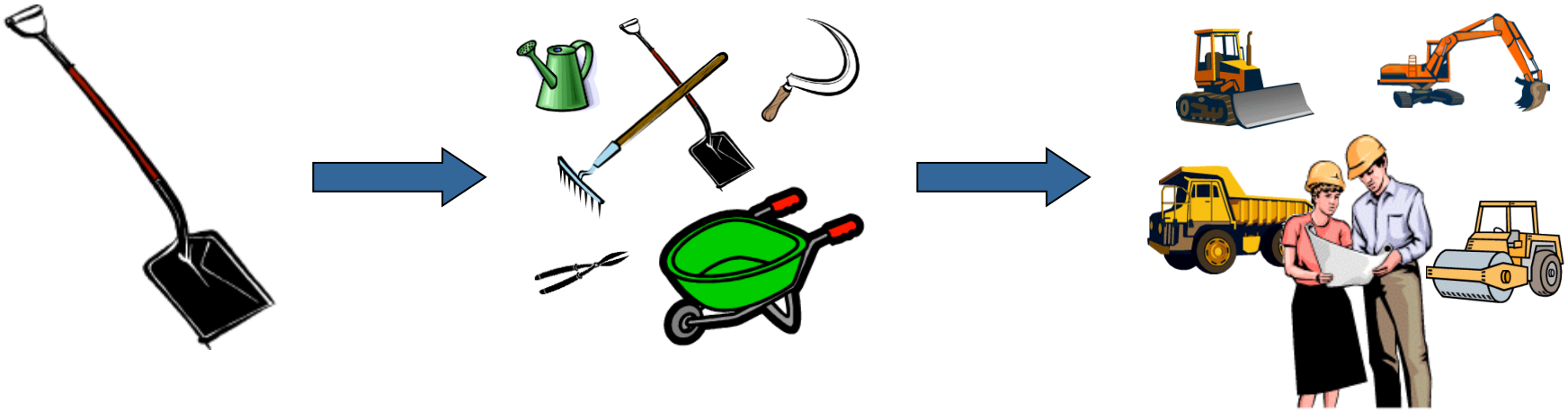
Post-mortem analysis

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

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 modeling

Remark: No Single Solution is Sufficient!



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

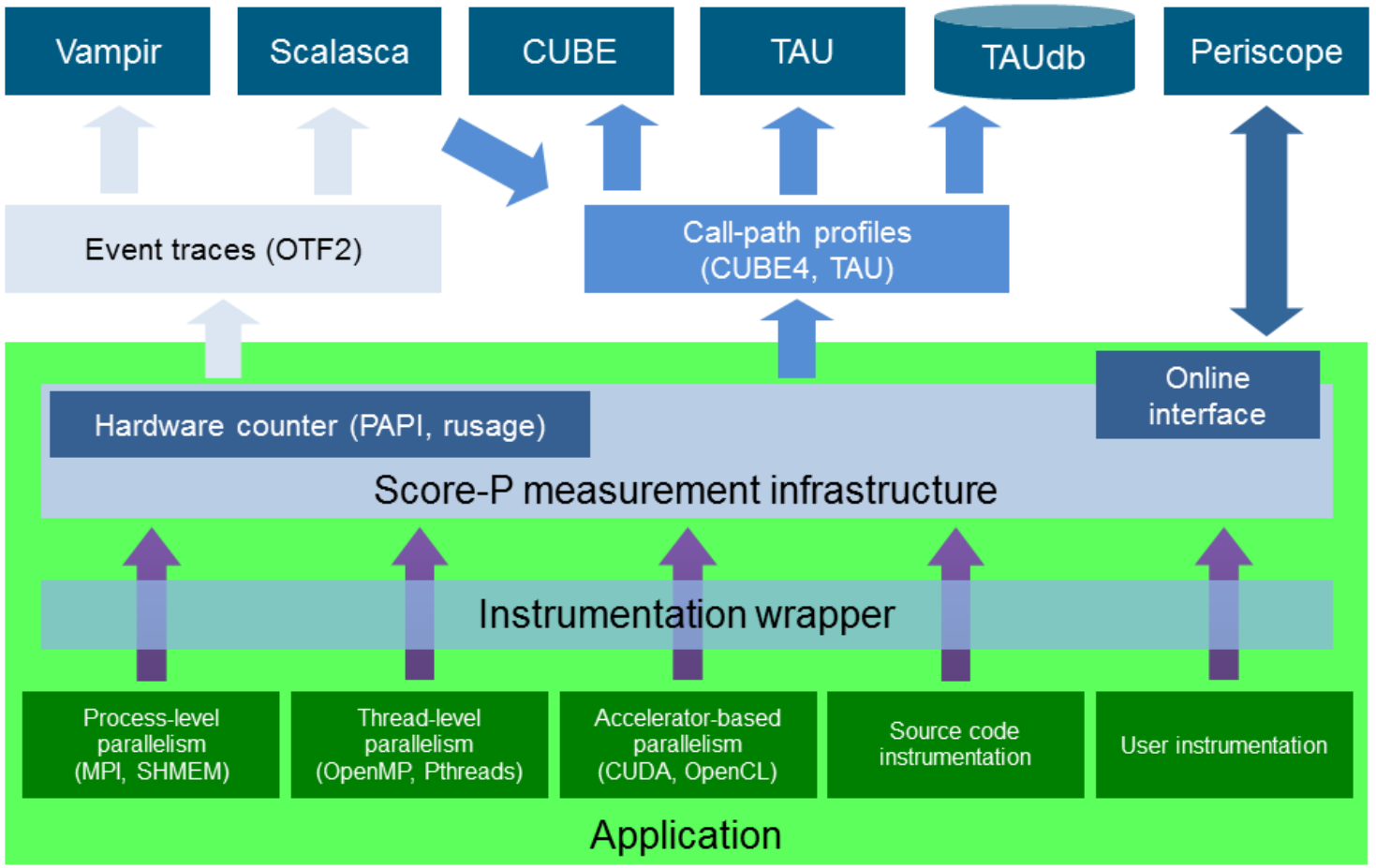
- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...

Score-P

- Community instrumentation and measurement infrastructure
  - Developed by a consortium of performance tool groups

  - Next generation measurement system of
    - Scalasca 2.x
    - Vampir
    - TAU
    - Periscope
- Common data formats improve tool interoperability
- http://www.score-p.org

Score-P Overview



scalasca

JÜLICH FORSCHUNGSZENTRUM

German Research School for Simulation Sciences

- Collection of trace-based performance analysis tools
  - Specifically designed for large-scale systems
  - Unique features:
    - Scalable, automated search for event patterns representing inefficient behavior
    - Scalable identification of the critical execution path
    - Delay / root-cause analysis
- Based on Score-P for instrumentation and measurement
  - Includes convenience / post-processing commands providing added value
- http://www.scalasca.org

What is the Key Bottleneck?

- Generate flat MPI profile using Score-P/Scalasca (or mpiP)
  - Only requires re-linking
  - Low runtime overhead

- Provides detailed information on MPI usage
  - How much time is spent in which operation?
  - How often is each operation called?
  - How much data was transferred?

- Limitations:
  - Computation on non-master threads and outside of MPI_Init/MPI_Finalize scope ignored

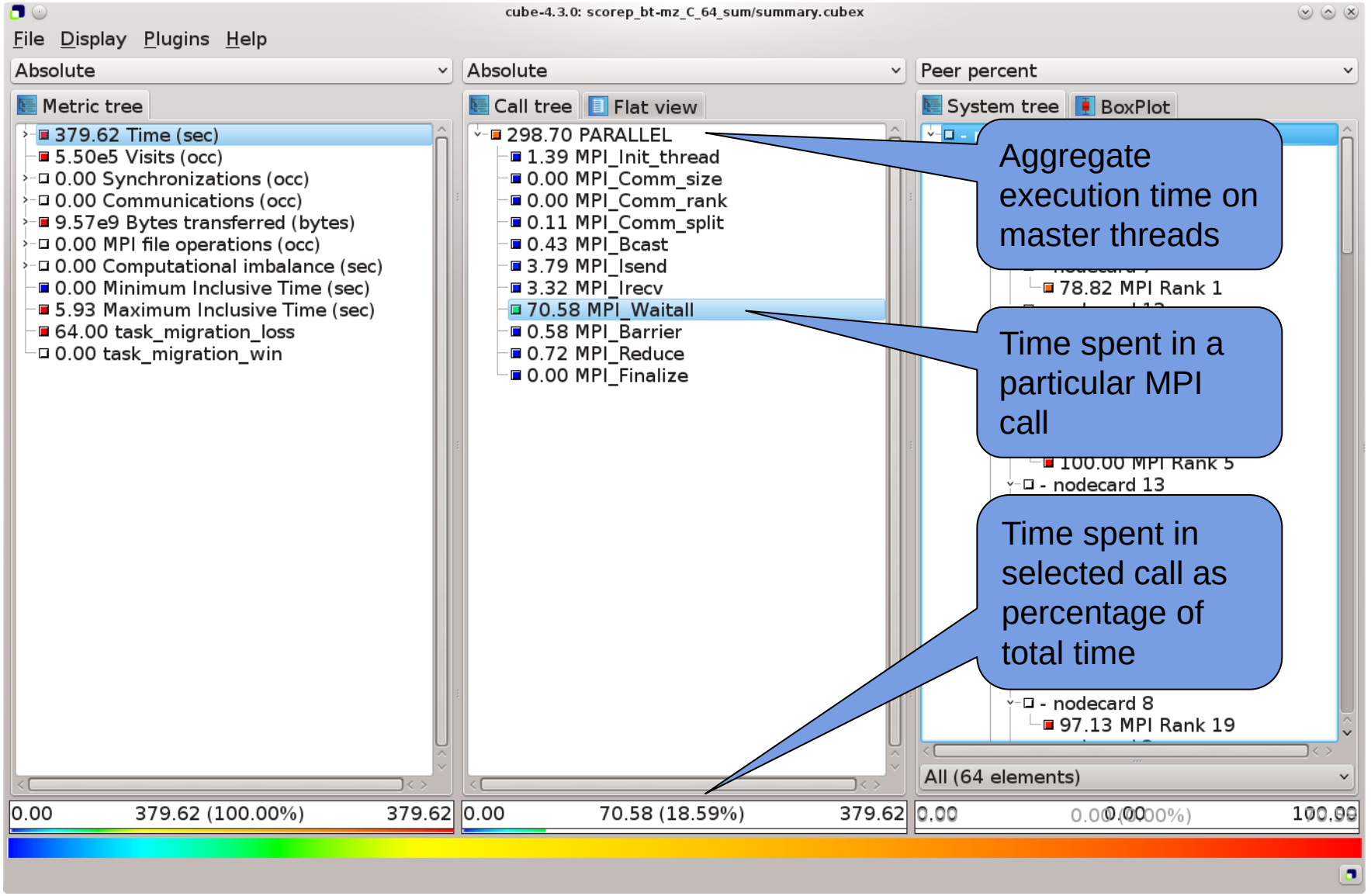
Flat MPI Profile: Recipe

1. Prefix your ***link command*** with
   "`scorep --nocompiler`"
2. Prefix your MPI ***launch command*** with
   "`scalasca -analyze`"
3. After execution, examine analysis results using
   "`scalasca -examine scorep_<title>`"

Flat MPI Profile: Example

```
% module load UNITE scorep scalasca
% mpixlf90 -O3 -qsmp=omp -c foo.f90
% mpixlf90 -O3 -qsmp=omp -c bar.f90
% scorep --nocompiler \
    mpixlf90 -O3 -qsmp=omp -o myprog foo.o bar.o

############################
##  In the job script:  ##
############################

module load UNITE scalasca
scalasca -analyze \
    runjob --ranks-per-node P --np n [...] --exe ./myprog

############################
## After job finished:  ##
############################

% scalasca -examine scorep_myprog_Ppnxt_sum
```

Flat MPI Profile: Example (cont.)



Where is the Key Bottleneck?

- Generate call-path profile using Score-P/Scalasca
  - Requires re-compilation
  - Runtime overhead depends on application characteristics
  - Typically needs some care setting up a good measurement configuration
    - Filtering
    - Selective instrumentation

- Option 1 (recommended):
  Automatic compiler-based instrumentation

- Option 2:
  Manual instrumentation of interesting phases, routines, loops

Call-path Profile: Recipe

1. Prefix your ***compile & link commands*** with
   "`scorep`"
2. Prefix your MPI ***launch command*** with
   "`scalasca -analyze`"
3. After execution, compare overall runtime with uninstrumented run to determine overhead
4. If overhead is too high
   1. Score measurement using
      "`scalasca -examine -s scorep_<title>`"
   2. Prepare filter file
   3. Re-run measurement with filter applied using prefix
      "`scalasca -analyze -f <filter_file>`"
5. After execution, examine analysis results using
   "`scalasca -examine scorep_<title>`"

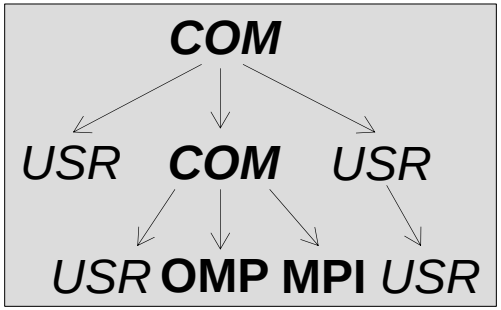
Call-path Profile: Example

```
% module load UNITE scorep scalasca
% scorep mpixlf90 -O3 -qsmp=omp -c foo.f90
% scorep mpixlf90 -O3 -qsmp=omp -c bar.f90
% scorep \
    mpixlf90 -O3 -qsmp=omp -o myprog foo.o bar.o



############################
##  In the job script:  ##
############################

module load UNITE scalasca
scalasca -analyze \
    runjob --ranks-per-node P --np n [...] --exe ./myprog
```

Call-path Profile: Example (cont.)

```
% scalasca -examine -s epik_myprog_Ppnxt_sum
scorep-score -r ./epik_myprog_Ppnxt_sum/profile.cubex
INFO: Score report written to ./scorep_myprog_Ppnxt_sum/scorep.score
```

- Estimates trace buffer requirements
- Allows to identify canditate functions for filtering
  - ☞ Computational routines with high visit count and low time-per-visit ratio
- Region/call-path classification
  - MPI (pure MPI library functions)
  - OMP (pure OpenMP functions/regions)
  - USR (user-level source local computation
  - COM (“combined” USR + OpeMP/MPI)
  - ANY/ALL (aggregate of all region types)



Call-path Profile: Example (cont.)

```
% less scorep_myprog_Ppmxt_sum/scorep.score
Estimated aggregate size of event trace:                   162GB
Estimated requirements for largest trace buffer (max_buf): 2758MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       2822MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=2822MB to avoid
 intermediate flushes or reduce requirements using USR regions
 filters.)

flt type    max_buf[B]         visits  time[s] time[%] time/     region
                                                       visit[us]
     ALL 2,891,417,902 6,662,521,083 36581.51   100.0      5.49  ALL
     USR 2,858,189,854 6,574,882,113 13618.14    37.2      2.07  USR
     OMP    54,327,600    86,353,920 22719.78    62.1    263.10  OMP
     MPI       676,342       550,010   208.98     0.6    379.96  MPI
     COM       371,930       735,040    34.61     0.1     47.09  COM

     USR   921,918,660 2,110,313,472  3290.11     9.0      1.56  matmul_sub
     USR   921,918,660 2,110,313,472  5914.98    16.2      2.80  binvcrhs
     USR   921,918,660 2,110,313,472  3822.64    10.4      1.81  matvec_sub
     USR    41,071,134    87,475,200   358.56     1.0      4.10  lhsinit
     USR    41,071,134    87,475,200   145.42     0.4      1.66  binvrhs
     USR    29,194,256    68,892,672    86.15     0.2      1.25  exact_solution
     OMP     3,280,320     3,293,184    15.81     0.0      4.80  !$omp parallel
     [...]
```

Call-path Profile: Filtering

- In this example, the 6 most fequently called routines are of type USR
- These routines contribute around 35% of total time
  - However, much of that is most likely measurement overhead
    - Frequently executed
    - Time-per-visit ratio in the order of a few microseconds

☞ Avoid measurements to reduce the overhead

☞ List routines to be filtered in simple text file

Filtering: Example

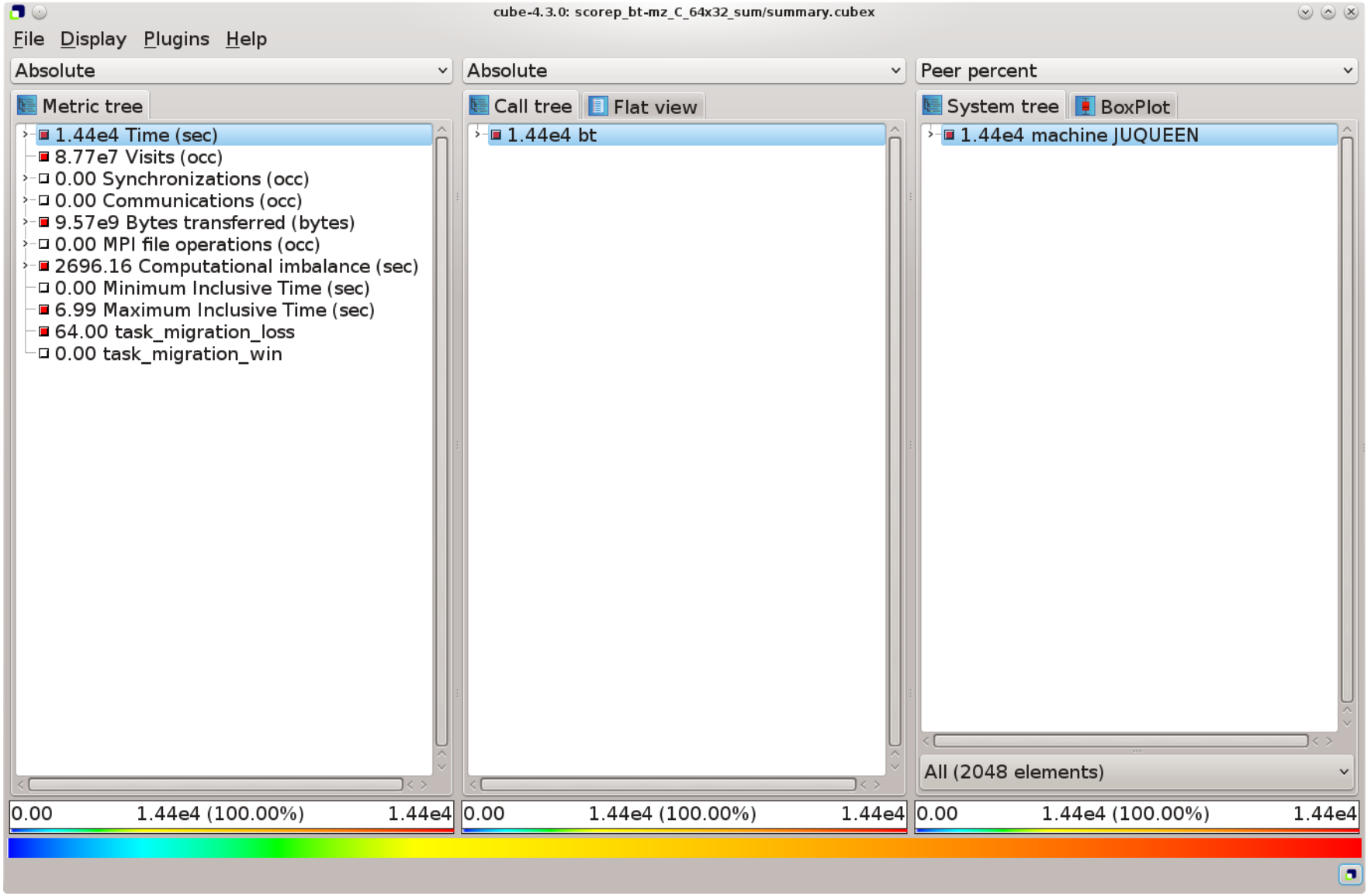
```
% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE
        binvcrhs
        matmul_sub
        matvec_sub
        binvrhs
        lhsinit
        exact_solution
SCOREP_REGION_NAMES_END
```

- Score-P filtering files support
  - Wildcards (shell globs)
  - Blacklisting
  - Whitelisting
  - Filtering based on filenames

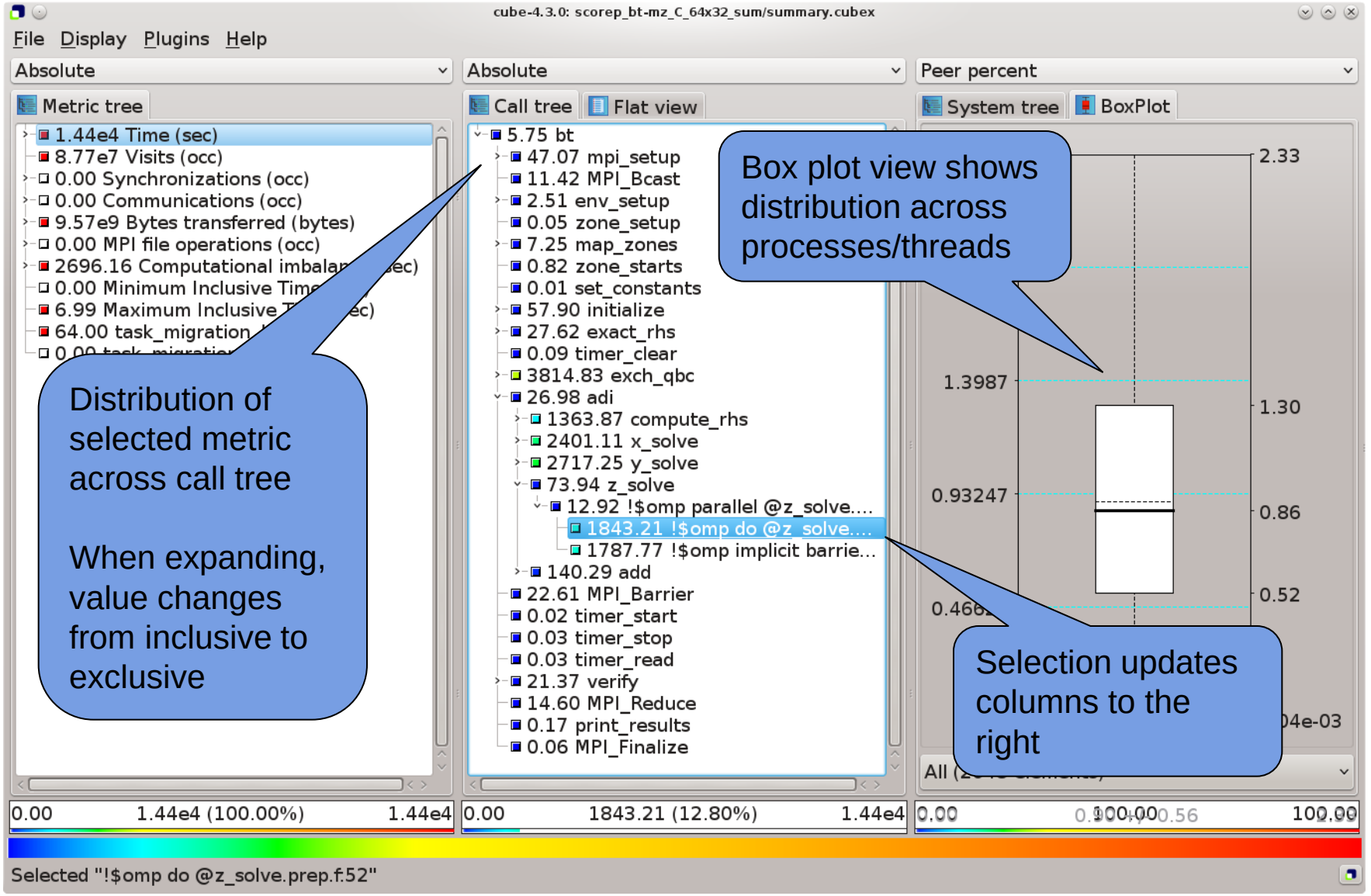
Call-path Profile: Example (cont.)

```
##  To verify effect of filter:

% scalasca -examine -s -f filter.txt \
    scorep_myprog_Ppnxt_sum

############################
##  In the job script:  ##
############################

module load UNITE scalasca
scalasca -analyze -f filter.txt \
    runjob --ranks-per-node P --np n [...] --exe ./myprog

############################
## After job finished:  ##
############################

% scalasca -examine scorep_myprog_Ppnxt_sum
```

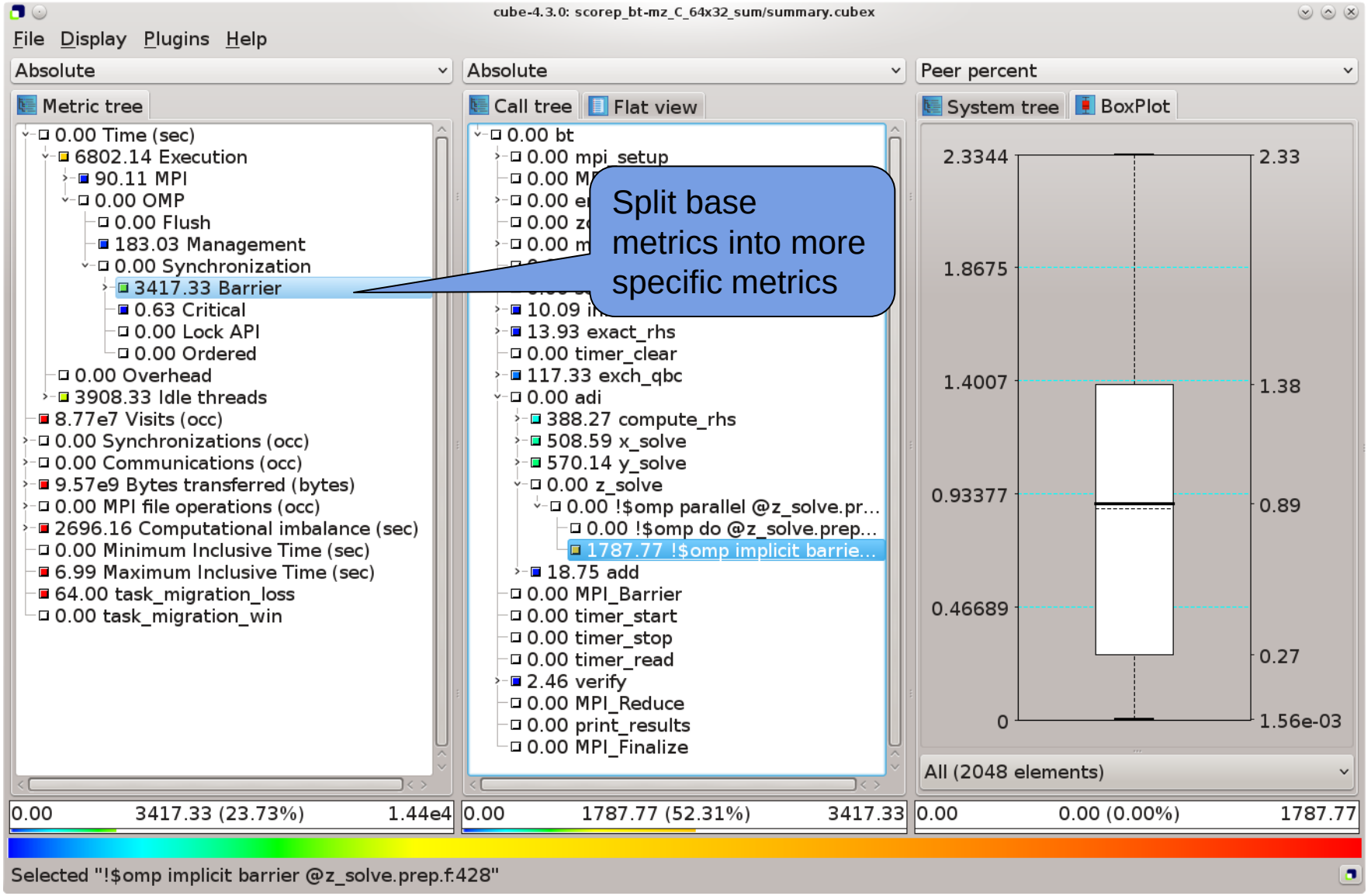
Call-path Profile: Example (cont.)



Call-path Profile: Example (cont.)



Call-path Profile: Example (cont.)



Score-P: Advanced Features

- Measurement can be extensively configured via environment variables
  - Check output of "`scorep-info config-vars`" for details
- Allows for targeted measurements:
  - Selective recording
  - Phase profiling
  - Parameter-based profiling
  - …
- Please ask us or see the user manual for details

Why is the Bottleneck There?

- This is highly application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Manual/automatic event trace analysis

Hardware Counters

- Counters: set of registers that count processor events, e.g. floating point operations or cycles
- Number of registers, counters and simultaneously measurable events vary between platforms
- Can be measured by:
  - perf:
    - Integrated in Linux since Kernel 2.6.31
    - Library and CLI
  - LIKWID:
    - Direct access to MSRs (requires Kernel module)
    - Consists of multiple tools and an API
    - x86 only
  - PAPI (Performance API)

PAPI

- Portable API: Uses the same routines to access counters across all supported architectures
- Used by most performance analysis tools

- High-level interface:
  - Predefined standard events, e.g. PAPI_FP_OPS
  - Availability and definition of events varies between platforms
  - List of available counters: `papi_avail (-d)`
- Low-level interface:
  - Provides access to all machine specific counters
  - Not-portable
  - More flexible
  - List of available counters: `papi_native_avail`

HW Counter Measurements w/ Score-P

- Score-P supports both PAPI preset and native counters
- Available counters: `papi_avail` or `papi_native_avail`

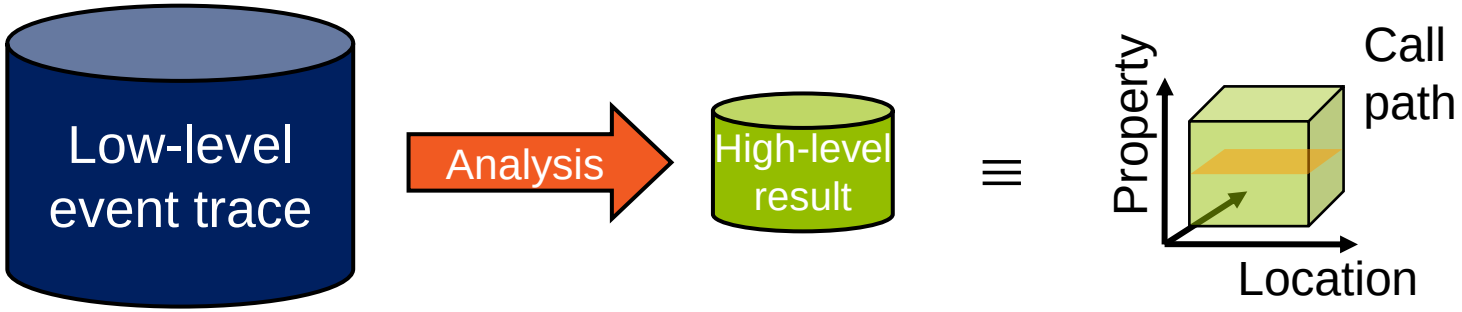
```
% module load UNITE papi/5.0.1
% less $PAPI_ROOT/doc/papi-5.0.1-avail.txt
% less $PAPI_ROOT/doc/papi-5.0.1-native_avail.txt
% less $PAPI_ROOT/doc/papi-5.0.1-avail-detail.txt
```

- Specify using “SCOREP_METRIC_PAPI” environment variable

```
############################
##  In the job script:  ##
############################

module load UNITE scalasca
export SCOREP_METRIC_PAPI=“PAPI_FP_OPS,PAPI_TOT_CYC“
scalasca -analyze -f filter.txt \
runjob --ranks-per-node P --np n [...] --exe ./myprog
```

Automatic Trace Analysis w/ Scalasca

- Idea: Automatic search for patterns of inefficient behavior
  - Identification of wait states and their root causes
  - Classification of behavior & quantification of significance
  - Scalable identification of the critical execution path



- Advantages
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Helps to identify hot-spots for in-depth manual analysis

Trace Generation & Analysis w/ Scalasca

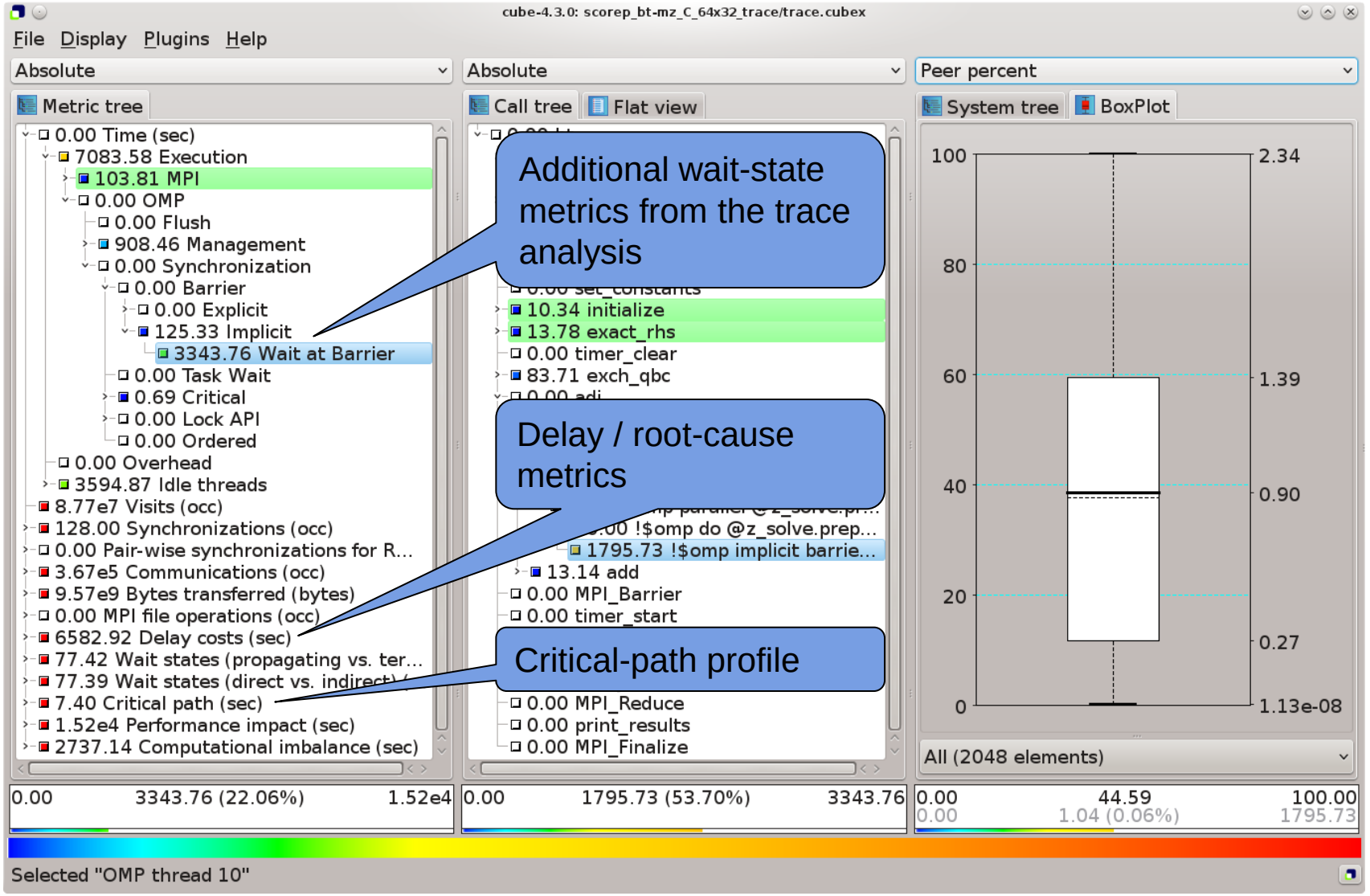
JÜLICH FORSCHUNGSZENTRUM

- Enable trace collection & analysis using "`-t`" option of "`scalasca -analyze`":

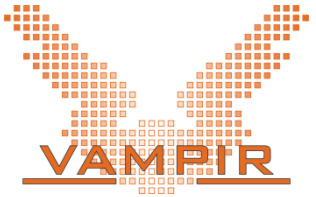
```
############################
##  In the job script:  ##
############################

module load UNITE scalasca
export SCOREP_TOTAL_MEMORY=120MB   # Consult score report
scalasca -analyze -f filter.txt -t \
    runjob --ranks-per-node P --np n [...] --exe ./myprog
```

- **ATTENTION:**
  - Traces can quickly become extremely large!
  - Remember to use proper filtering, selective instrumentation, and Score-P memory specification
  - Before flooding the file system, **ask us for assistance!**

Scalasca Trace Analysis Example



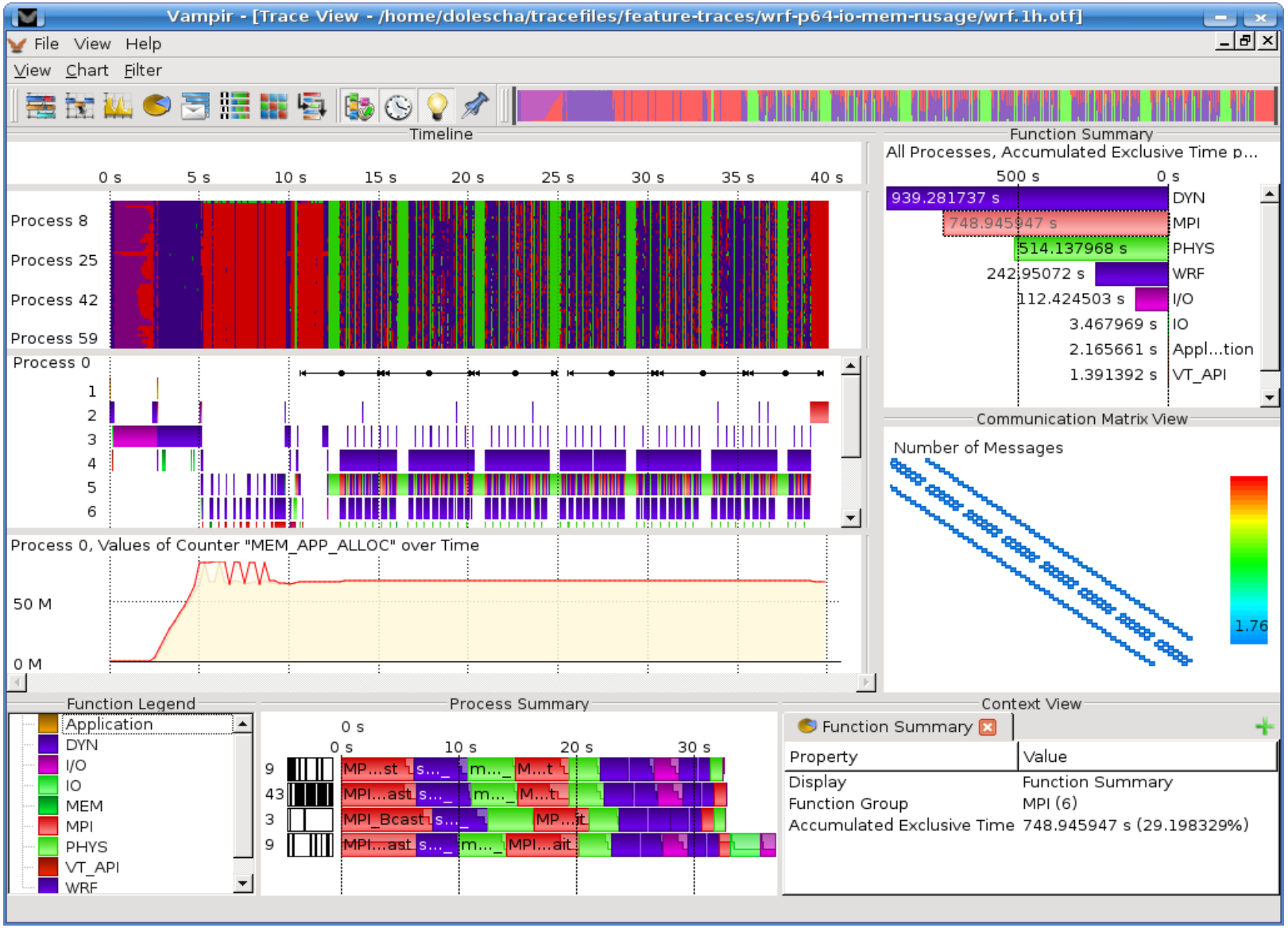
Vampir Event Trace Visualizer



- Offline trace visualization for Score-P's OTF2 trace files
- Visualization of MPI, OpenMP and application events:
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for ANY part of the trace
- http://www.vampir.eu

- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis

Vampir Displays



Allinea Performance Reports

- Single page report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows CPU, memory, network and I/O utilization

- Supports MPI, multi-threading and accelerators
- Saves data in HTML, CVS or text form

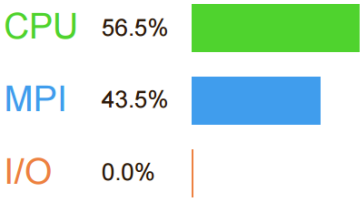
- http://www.allinea.com/products/allinea-performance-reports
- Note: License limited to 512 processes (with unlimited number of threads)

Example Performance Reports

JÜLICH FORSCHUNGSZENTRUM

Summary: cp2k.popt is CPU-bound in this configuration

The total wallclock time was spent as follows:



Time spent running application code. High values are usually good.
This is **average**; check the CPU performance section for optimization advice.

Time spent in MPI calls. High values are usually bad.
This is **average**; check the MPI breakdown for advice on reducing it.

Time spent in filesystem I/O. High values are usually bad.
This is **negligible**; there's no need to investigate I/O performance.

This application run was CPU-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

CPU

A breakdown of how the 56.5% total CPU time was spent:

Scalar numeric ops	27.7%
Vector numeric ops	11.3%
Memory accesses	60.9%
Other	0.0

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

Of the 43.5% total time spent in MPI calls:

Time in collective calls	8.2%
Time in point-to-point calls	91.8%
Estimated collective rate	169 Mb/s
Estimated point-to-point rate	50.6 Mb/s

The point-to-point transfer rate is low. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait. Use an MPI profiler to identify the problematic calls and ranks.

I/O

A breakdown of how the 0.0% total I/O time was spent:

Time in reads	0.0%
Time in writes	0.0%
Estimated read rate	0 bytes/s
Estimated write rate	0 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	82.5 Mb
Peak process memory usage	89.3 Mb
Peak node memory usage	7.4%

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.

HPCToolkit (Rice University)

JÜLICH FORSCHUNGSZENTRUM

- Multi-platform sampling-based call-path profiler
- Works on unmodified, optimized executables
- http://hpctoolkit.org

- Advantages:
  - Overhead can be easily controlled via sampling interval
  - Advantageous for complex C++ codes with many small functions
  - Loop-level analysis (sometimes even individual source lines)
  - Supports POSIX threads
- Disadvantages:
  - Statistical approach that might miss details
  - MPI/OpenMP time displayed as low-level system calls

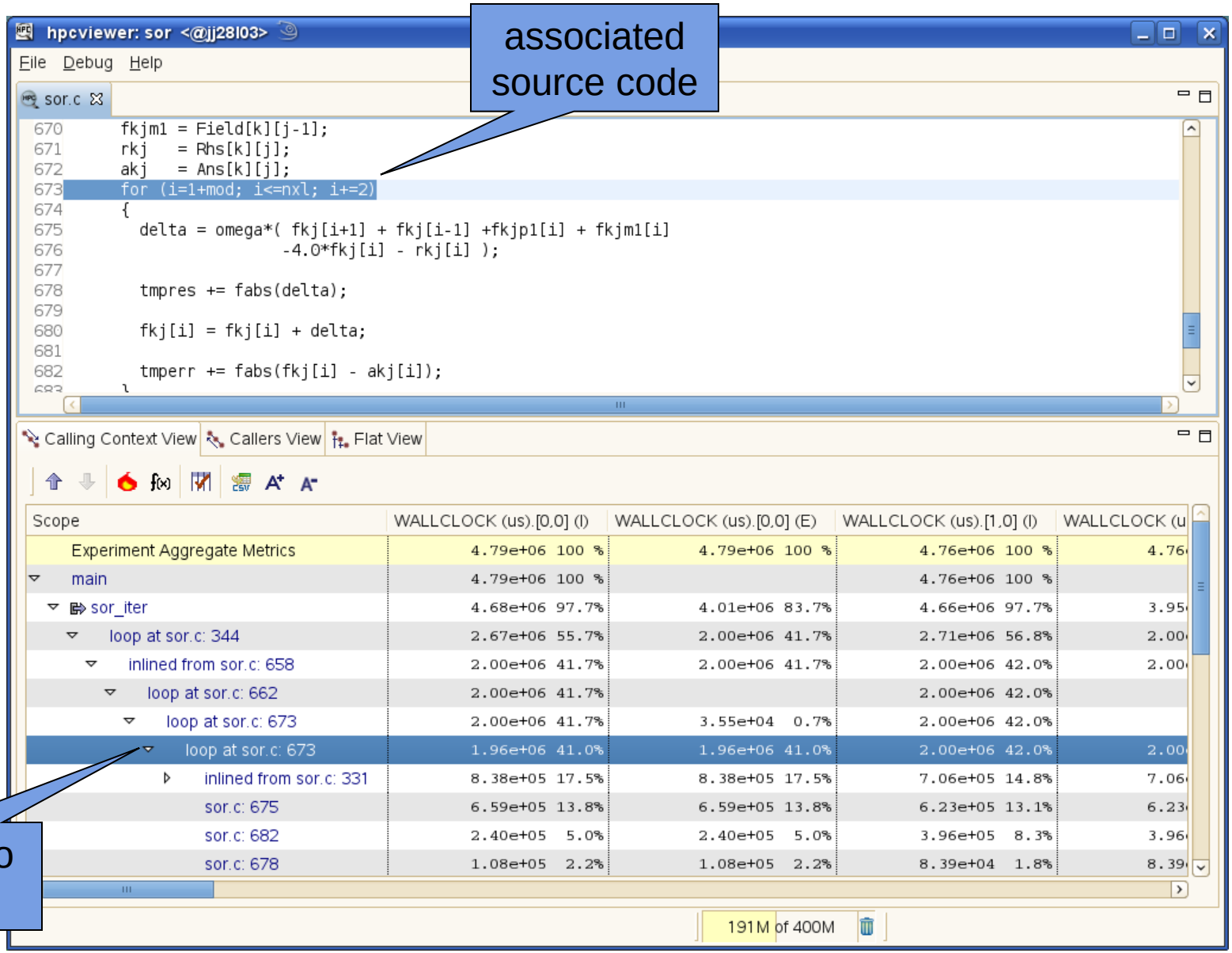
HPCToolkit: Recipe

1. Compile your code with "`-g -qnoipa`"
   - For MPI, also make sure your application calls MPI_Comm_rank first on MPI_COMM_WORLD
2. Prefix your ***link command*** with "`hpclink`"
   - Ignore potential linker warnings ;-)
3. Run your application as usual, specifying requested metrics with sampling intervals in environment variable
   "`HPCRUN_EVENT_LIST`"
4. Perform static binary analysis with
   "`hpcstruct --loop-fwd-subst=no <app>`"
5. Combine measurements with
   "`hpcprof -S <struct file> \`
   `-I "<path_to_src>/*" <measurement_dir>`"
6. View results with
   "`hpcviewer <hpct_database>`"

HPCToolkit: Metric Specification

- Specified via environment variable `HPCRUN_EVENT_LIST`
- General format:
  "`name@interval [;name@interval ...]`"
- Possible sample sources:
  - WALLCLOCK
  - PAPI counters
  - IO (use w/o interval spec)
  - MEMLEAK (use w/o interval spec)
- Interval: given in microseconds
  - E.g., 10000 → 100 samples per second

Example: hpcviewer



TAU

JÜLICH FORSCHUNGSZENTRUM

- Very portable tool set for instrumentation, measurement and analysis of parallel multi-threaded applications
- http://tau.uoregon.edu/

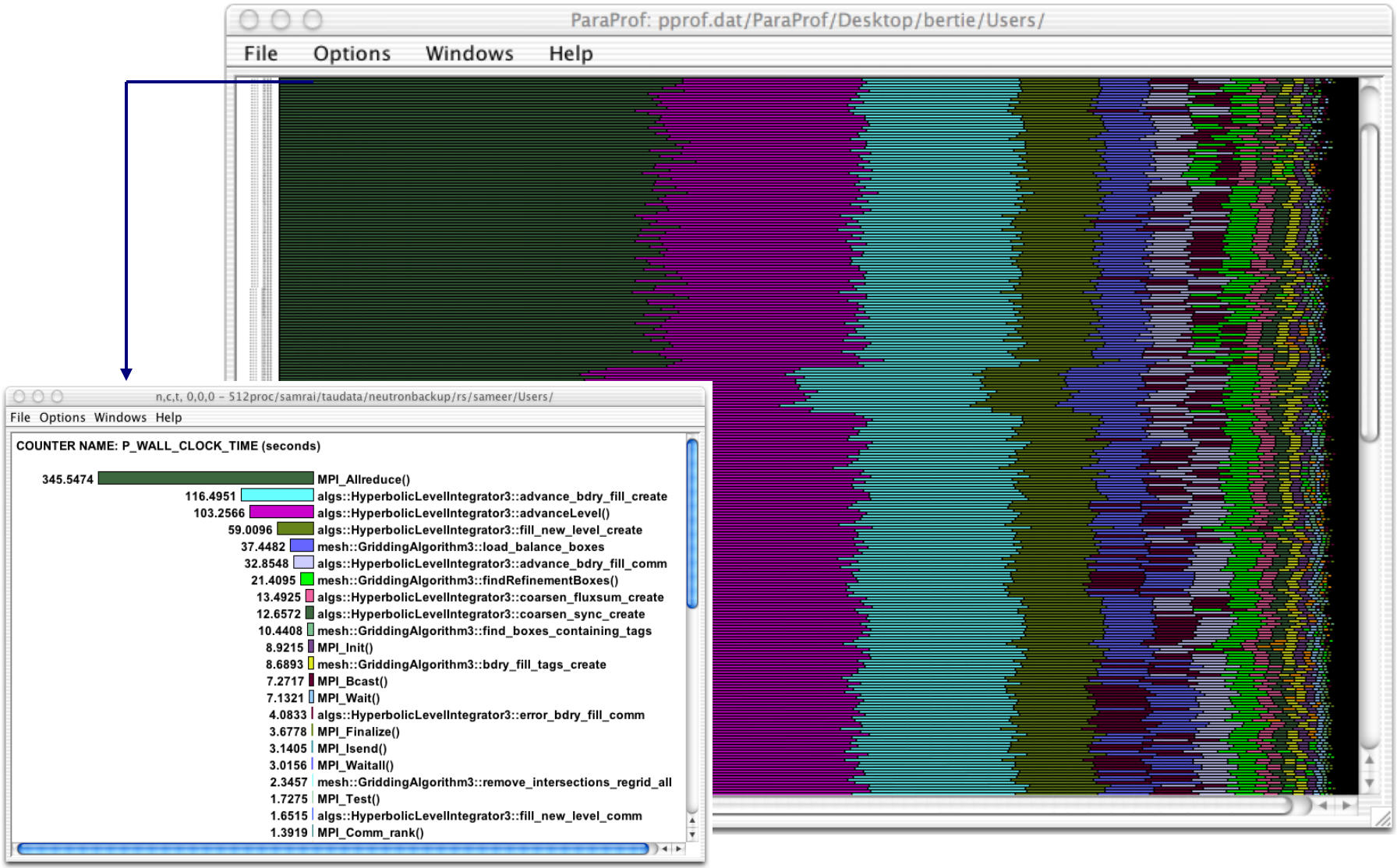
Tuning and Analysis Utilities

- Supports
  - Various profiling modes and tracing
  - Various forms of code instrumentation
  - C, C++, Fortran, Java, Python
  - MPI, multi-threading (OpenMP, Pthreads, …)
  - Accelerators

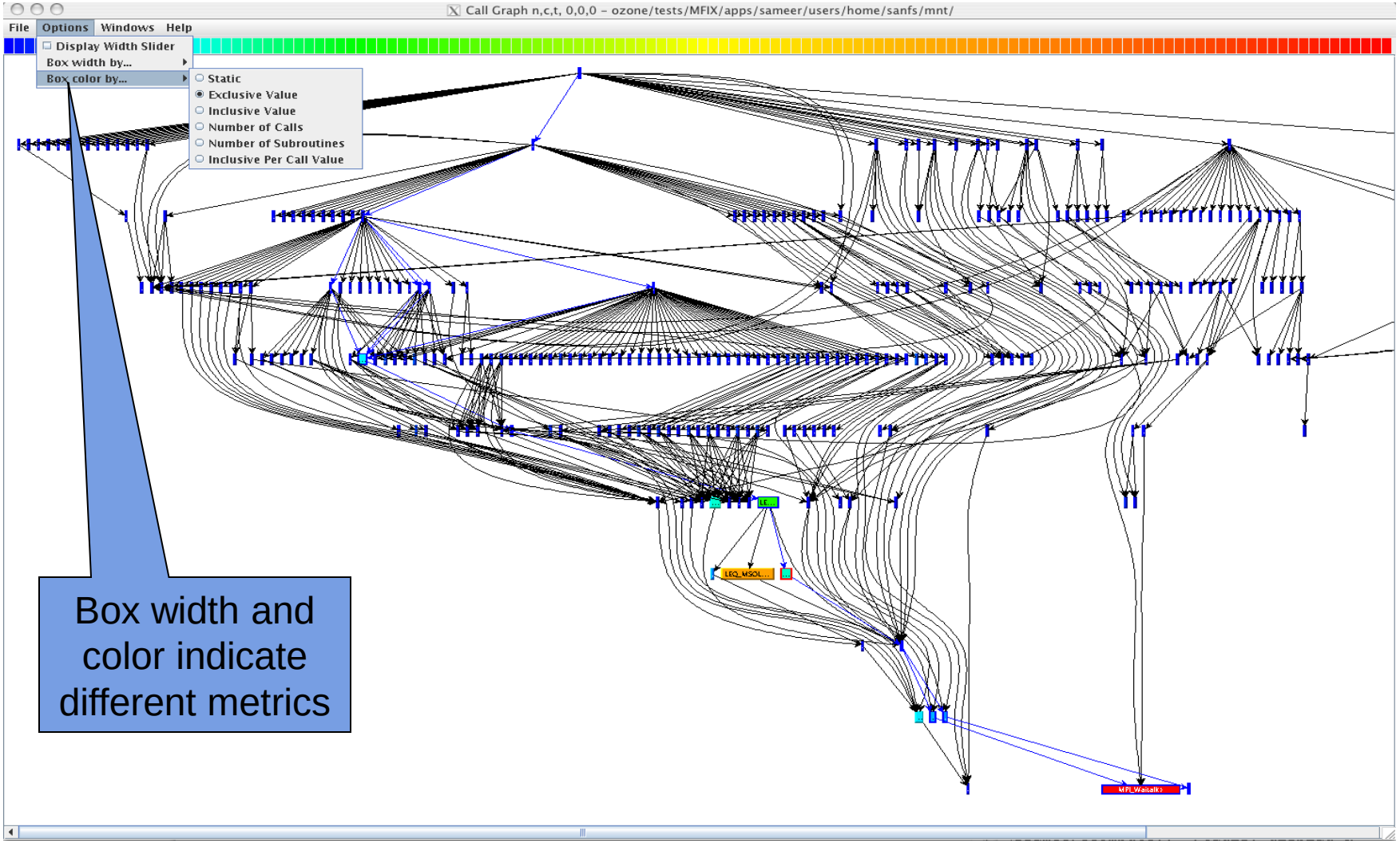
TAU: Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (PDT))
      - OpenMP (directive rewriting with Opari)
  - Object code
    - pre-instrumented libraries (e.g., MPI using PMPI)
    - statically-linked and dynamically-loaded (e.g., Python)
  - Executable code
    - dynamic instrumentation (pre-execution) (DynInst)
    - virtual machine instrumentation (e.g., Java using JVMPI)
- Support for performance mapping
- Support for object-oriented and generic programming

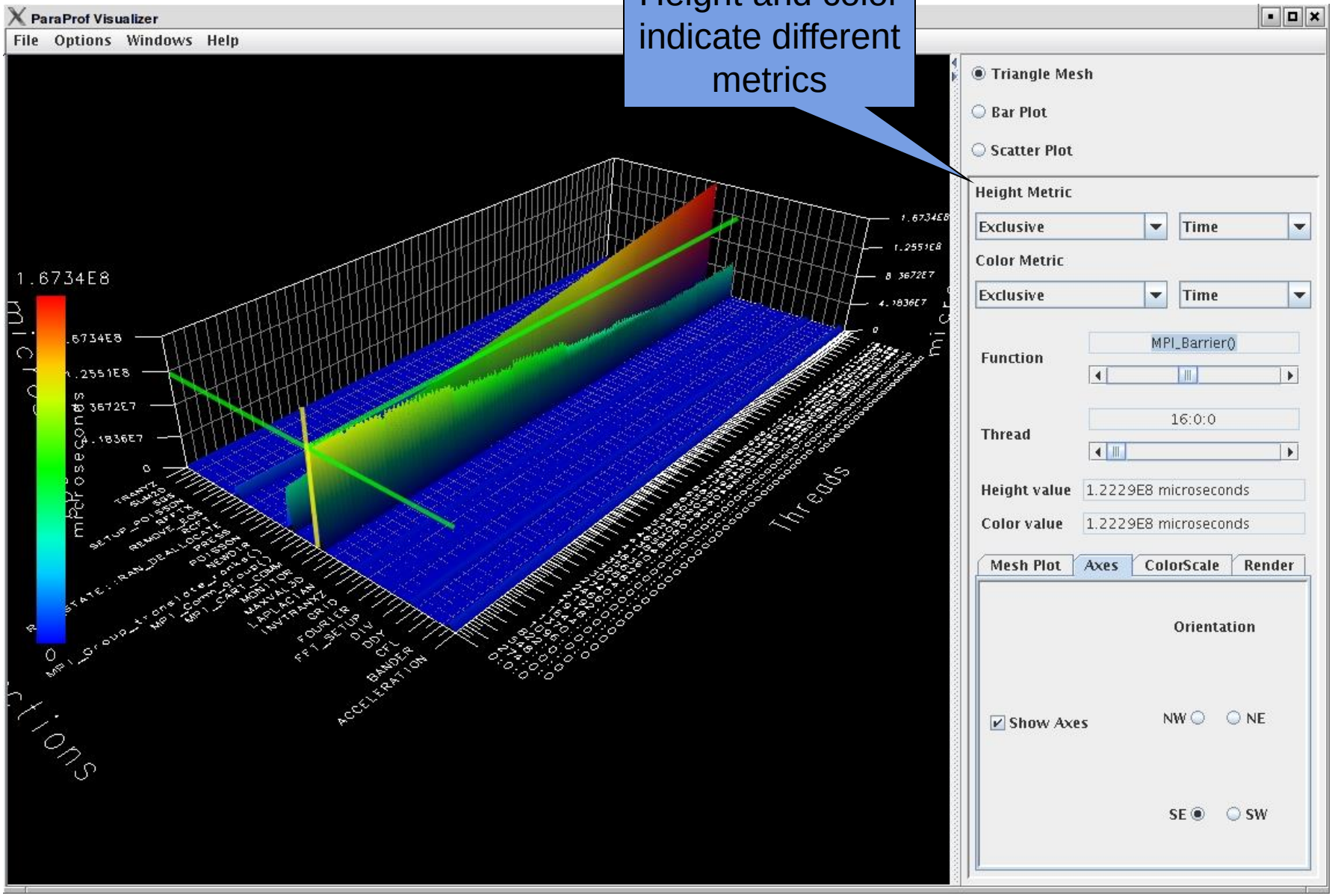
TAU: Basic Profile View



TAU: Callgraph Profile View



TAU: 3D Profile View



Tools not yet mentioned

- Gprof
- Callgrind
- MAQAO
- ompP
- mpiP
- Allinea MAP (commercial)
- Intel VTune (commercial)
- Open|SpeedShop
- Extrae/Paraver
- PerfSuite
- Nvidia visual profiler