HPC SOFTWARE – DEBUGGER AND PERFORMANCE ANALYSIS TOOLS

MAY 17, 2022 I MICHAEL KNOBLOCH

JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE

OUTLINE

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

Kent Beck

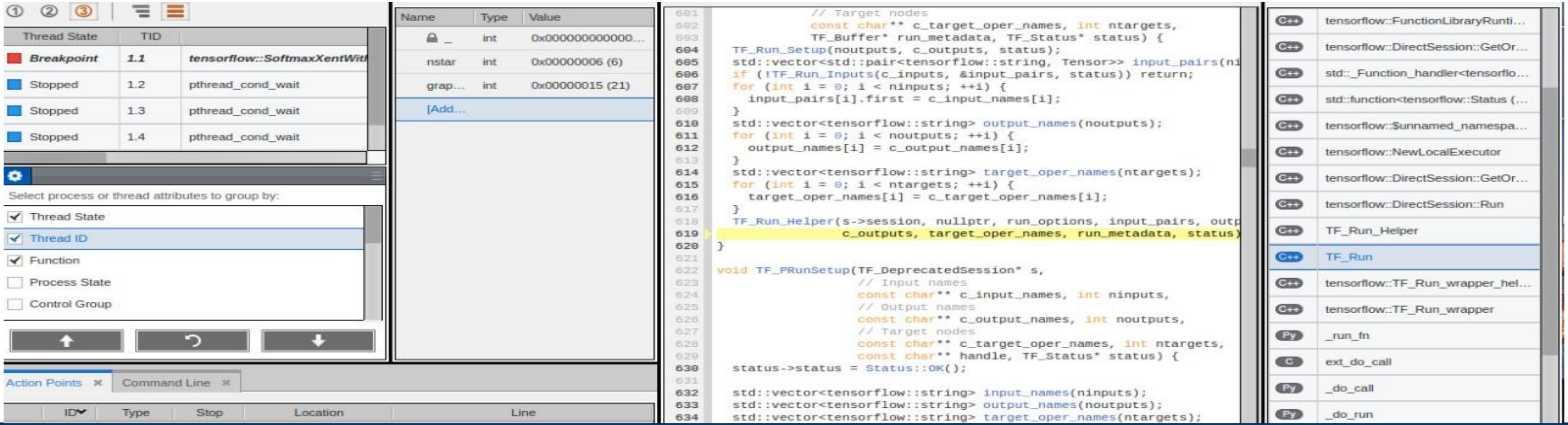
- Local module setup
- Compilers
- Libraries

Debugger:

- NVIDIA Tools
- TotalView
- DDT
- MUST
- Intel Inspector

Performance Tools:

- Score-P
- Scalasca
- Vampir
- Intel + AMD Tools
- ARM Tools
- TAU
- NVIDIA Tools
- Darshan
- PAPI
- And several more


Thread State
TID
Breakpoint
1.1
tensorflow::SoftmaxXentWith
Stopped
1.2
pthread_cond_wait
Stopped
1.3
pthread_cond_wait
Stopped
1.4
pthread_cond_wait
Select process or thread attributes to group by:
Thread State
Thread ID
Function
Process State
Control Group
Action Points
Command Line
ID
Type
Stop
Location
Line
Name
Type
Value
nstar
int
0x00000006 (6)
grap...
int
0x00000015 (21)
[Add...
TF_Run_Setup(noutputs, c_outputs, status);
TF_Run_Helper(s->session, nullptr, run_options, input_pairs,
c_outputs, target_oper_names, run_metadata, status)
void TF_PRunSetup(TF_DeprecatedSession* s,
status->status = Status::OK();
tensorflow::DirectSession::Run
TF_Run_Helper
TF_Run
tensorflow::TF_Run_wrapper
_run_fn
ext_do_call
_do_call
_do_run

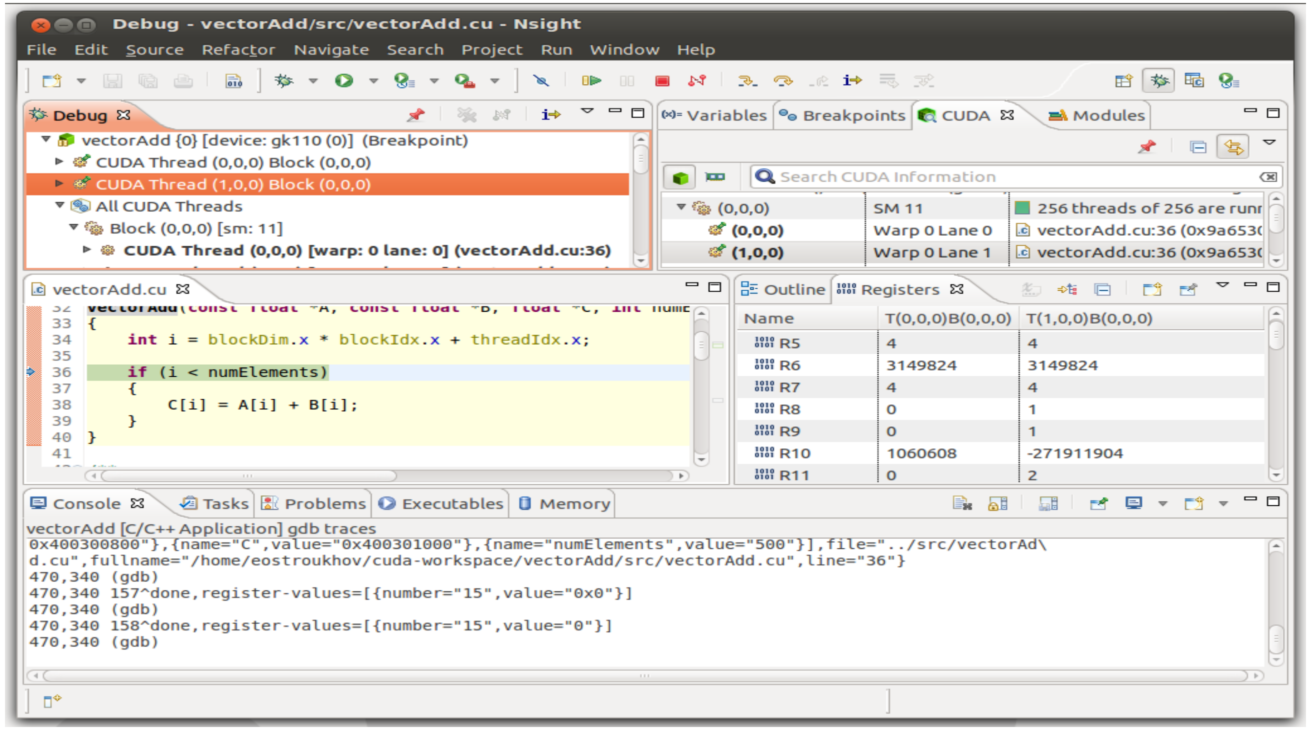
DEBUGGER

DEBUGGING TOOLS (STATUS: MAY 2022)

- **Debugger:**
  - CUDA-GDB
  - TotalView
  - ARMForge - DDT
- **Memory Analyzer:**
  - CUDA-MEMCHECK
  - Intel Inspector
- **Correctness Checker:**
  - MUST

CUDA-GDB

- Extension to gdb
- CLI and GUI (Nsight)
- Simultaneously debug on the CPU and multiple GPUs
- Use conditional breakpoints or break automatically on every kernel launch
- Can examine variables, read/write memory and registers and inspect the GPU state when the application is suspended
- Identify memory access violations
  - Run CUDA-MEMCHECK in integrated mode to detect precise exceptions.

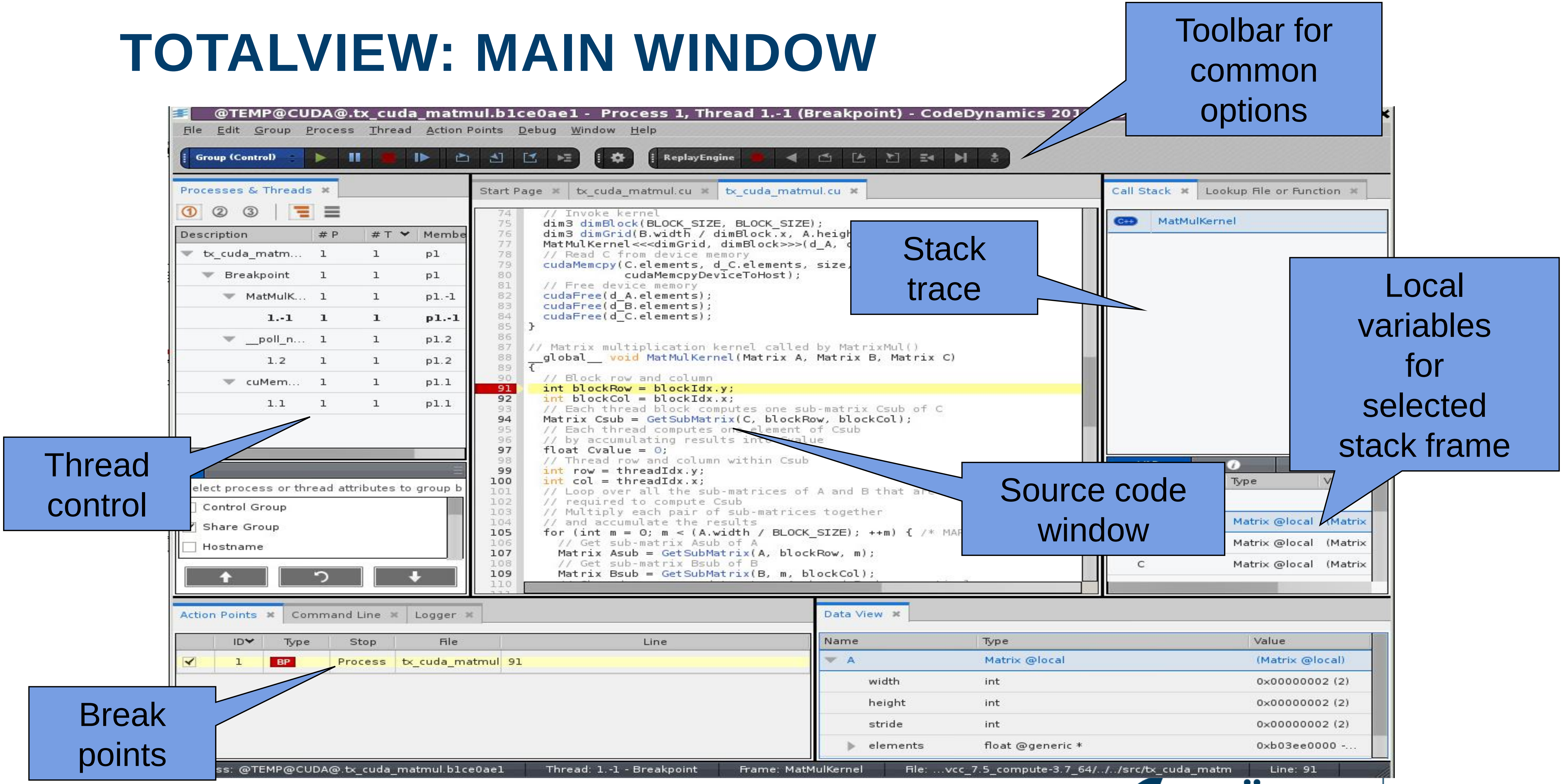


TOTALVIEW

- UNIX Symbolic Debugger for C/C++, Fortran, mixed Python/C++, PGI HPF, assembler programs
- JSC's "standard" debugger
- Advanced features
  - Multi-process and multi-threaded
  - Multi-dimensional array data visualization
  - Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, Pthreads)
  - Scripting and batch debugging
  - Advanced memory debugging
  - Reverse debugging
  - CUDA and OpenACC support
  - Remote debugging

- NOTE: JSC license limited to 2048 processes (shared between all users)

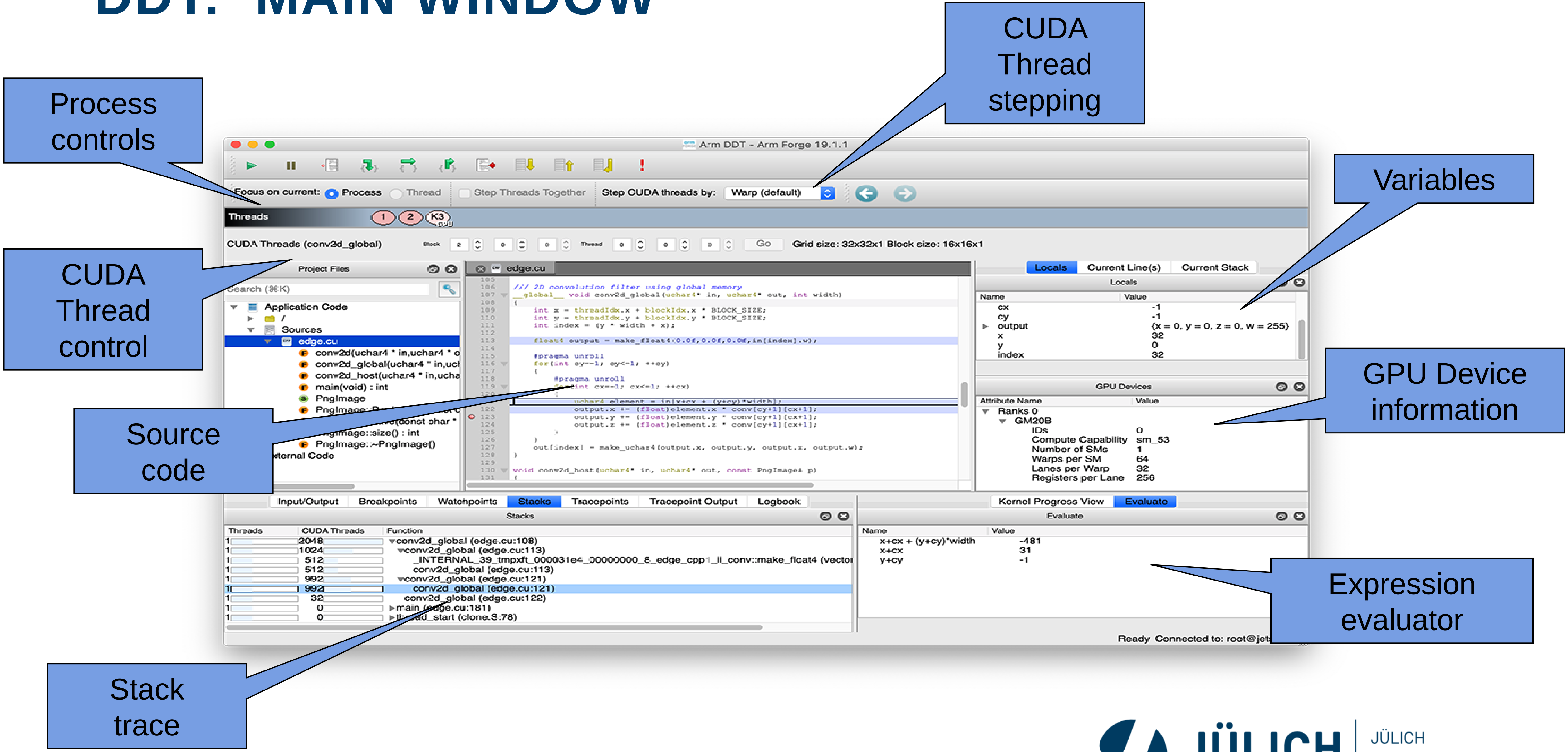
TOTALVIEW: MAIN WINDOW



DDT arm FORGE

- UNIX Graphical Debugger for C/C++, Fortran, and Python programs
- Modern, easy-to-use debugger
- Advanced features
  - Multi-process and multi-threaded
  - Multi-dimesional array data visualization
  - Support for MPI parallel debugging
    (automatic attach, message queues)
  - Support for OpenMP (Version 2.x and later)
  - Support for CUDA and OpenACC
  - Job submission from within debugger
- https://developer.arm.com
- NOTE: JSC license limited to 64 processes (shared between all users)

DDT: MAIN WINDOW



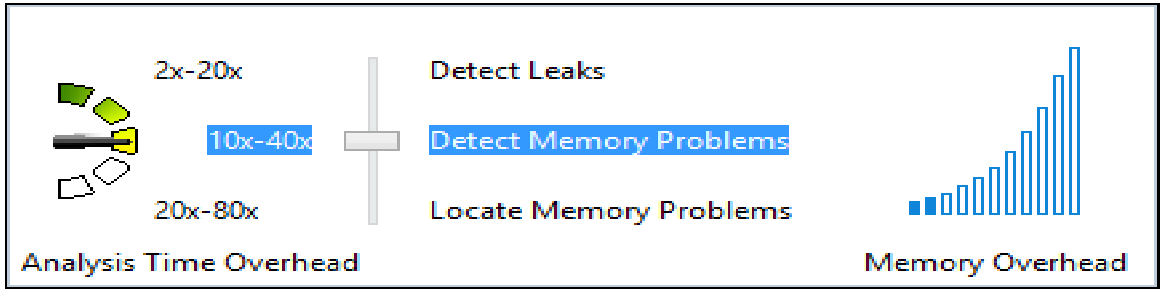
CUDA-MEMCHECK

- Valgrind for GPUs
- Monitors hundreds of thousands of threads running concurrently on each GPU
- Reports detailed information about global, local, and shared memory access errors (e.g. out-of-bounds, misaligned memory accesses)
- Reports runtime executions errors (e.g. stack overflows, illegal instructions)
- Reports detailed information about potential race conditions
- Displays stack back-traces on host and device for errors
- And much more

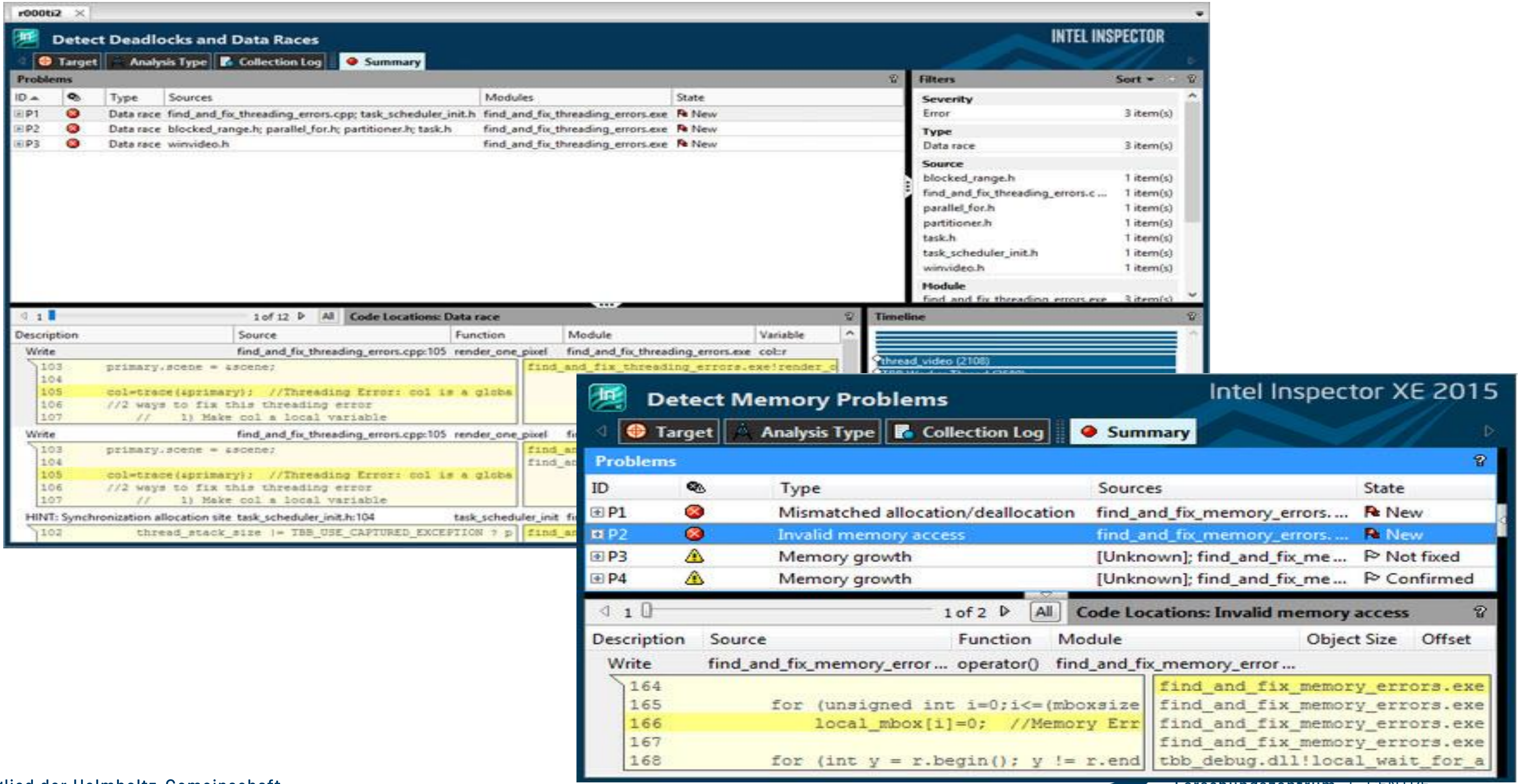
- Included in the CUDA Toolkit

INTEL INSPECTOR

- Detects memory and threading errors
  - Memory leaks, corruption and illegal accesses
  - Data races and deadlocks
- Dynamic instrumentation requiring no recompilation
- Supports C/C++ and Fortran as well as third party libraries
- Multi-level analysis to adjust overhead and analysis capabilities
- API to limit analysis range to eliminate false positives and speed-up analysis



INTEL INSPECTOR: GUI



MUST

- Next generation MPI correctness and portability checker
- https://www.i12.rwth-aachen.de/go/id/nrbe

- MUST reports
  - Errors: violations of the MPI-standard
  - Warnings: unusual behavior or possible problems
  - Notes: harmless but remarkable behavior
  - Potential deadlock detection

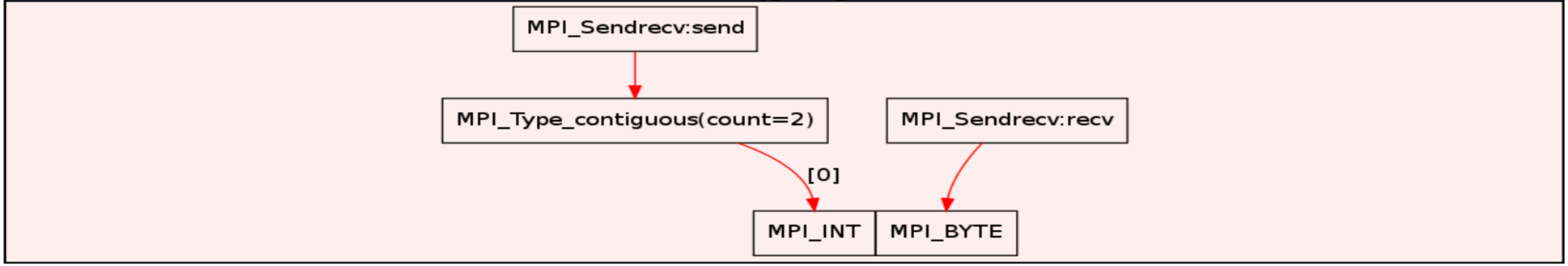
- Usage
  - Relink application with mustc, mustcxx, mustf90, …
  - Run application under the control of mustrun (requires (at least) one additional MPI process)
  - Saves output in html report

MUST DATATYPE MISMATCH

Rank	Type	Message	From	References
0	Error	A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous)[0](MPI_INT) in the send type and at (MPI_BYTE) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation of this situation is available in a detailed type mismatch view (MUST_Output-files/MUST_Typemismatch_0.html). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for C, commited at reference 4, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}) (Information on receive of count 8 with type:MPI_BYTE)	**MPI_Sendrecv** called from: #0 main@example.c:33	reference 1 rank 0: **MPI_Sendrecv** called from: #0 main@example.c:33 reference 2 rank 1: **MPI_Sendrecv** called from: #0 main@example.c:33 reference 3 rank 0: **MPI_Type_contiguous** called from: #0 main@example.c:29 reference 4 rank 0: **MPI_Type_commit** called from: #0 main@example.c:30

Message
The application issued a set of MPI calls that mismatch in type signatures! The graph below shows details on this situation. The first differing item of each involved communication request is highlighted.

Datatype Graph



MUST DEADLOCK DETECTION

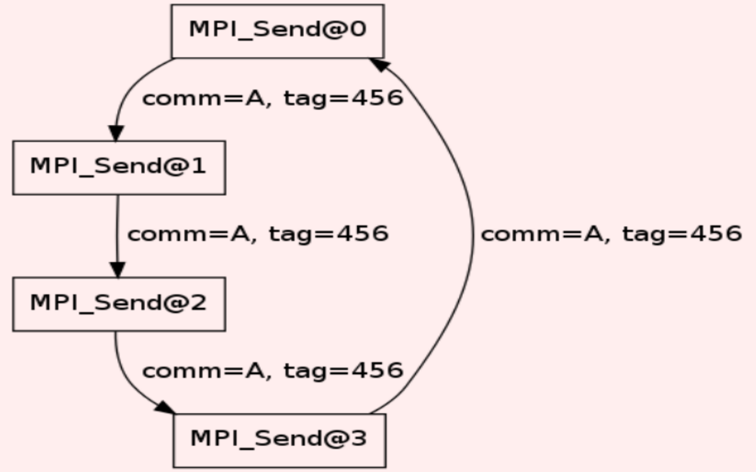
Message

The application issued a set of MPI calls that can cause a deadlock! The graphs below show details on this situation. This includes a wait-for graph that shows active wait-for dependencies between the processes that cause the deadlock. Note that this process set only includes processes that cause the deadlock and no further processes. A legend details the wait-for graph components in addition , while a parallel call stack view summarizes the locations of the MPI calls that cause the deadlock . Below these graphs, a message queue graph shows active and unmatched point-to-point communications. This graph only includes operations that could have been intended to match a point-to-point operation that is relevant to the deadlock situation. Finally, a parallel call stack shows the locations of any operation in the parallel call stack. The leafs of this call stack graph show the components of the message queue graph that they span. The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).

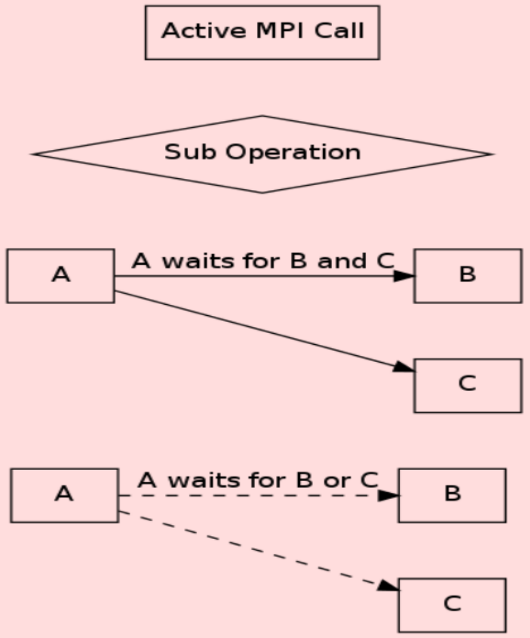
Active Communicators

Comm:	A
MPI_COMM_WORLD	

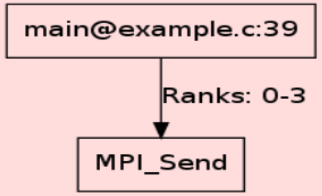
Wait-for Graph



Legend



Call Stack



Active and Relevant Point-to-Point Messages: Overview

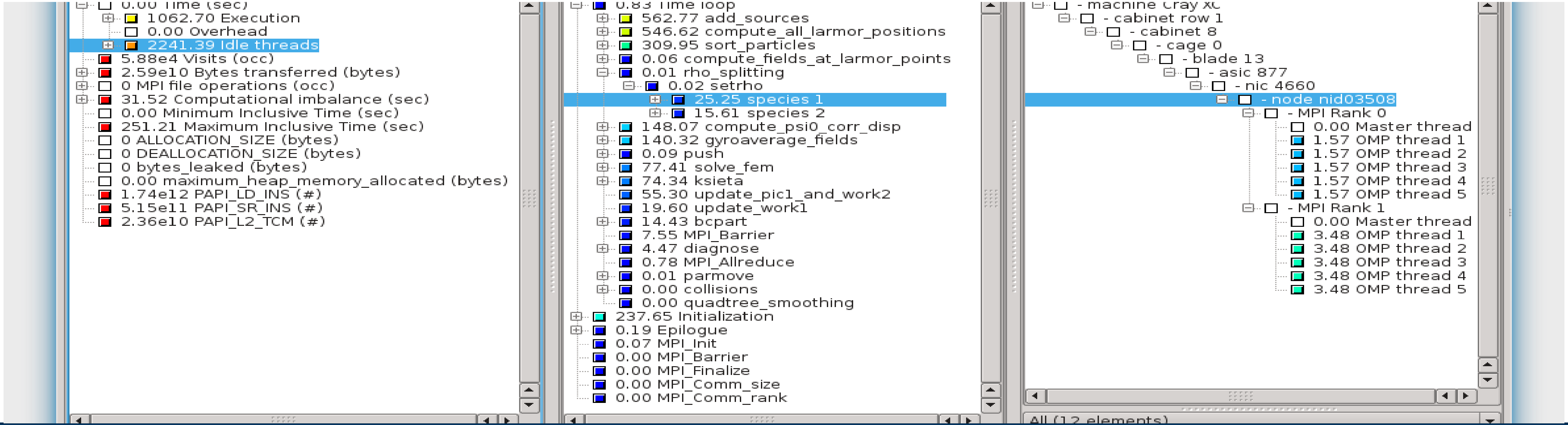
Active and Relevant Point-to-Point Messages: Callstack-view

DEBUGGING RECOMMENDATIONS

- Always debug at the lowest possible scale!

- GPU Applications:
  - Single Node: Use CUDA-MEMCHECK and CUDA-GDB
  - Multi-Node: Use TotalView/DDT

- MPI Applications:
  - Check with MUST at least once
  - Use TotalView/DDT at small scale (if error occurs there), else attach to as few processes as neccessary

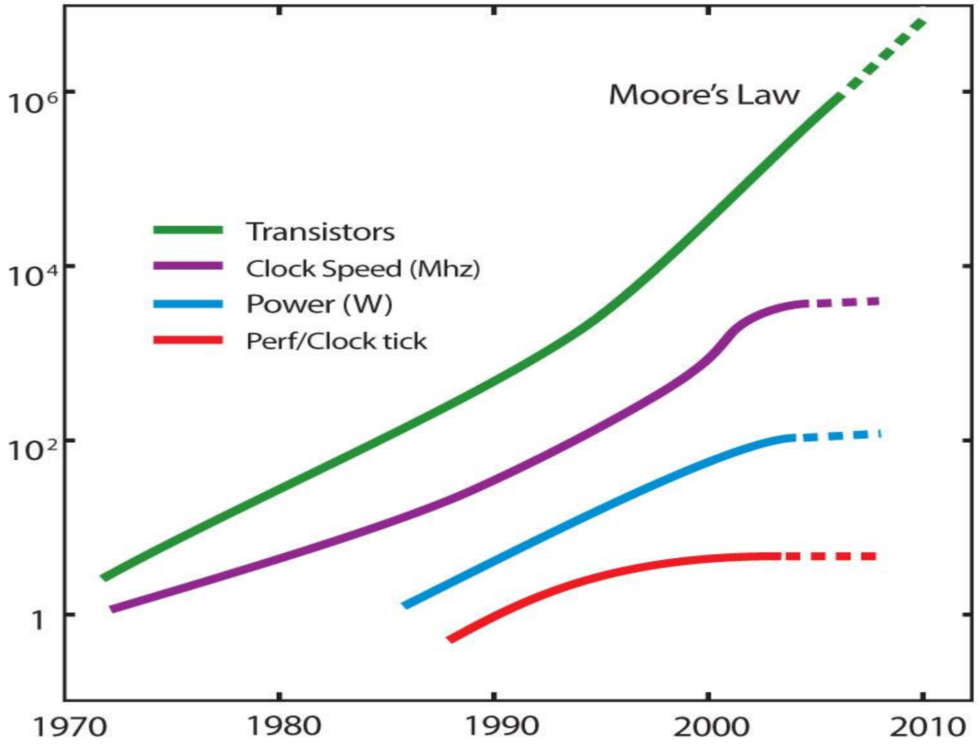


PERFORMANCE ANALYSIS TOOLS

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 Architecture

☞ Every doubling of scale reveals a new bottleneck!



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

PERFORMANCE FACTORS

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

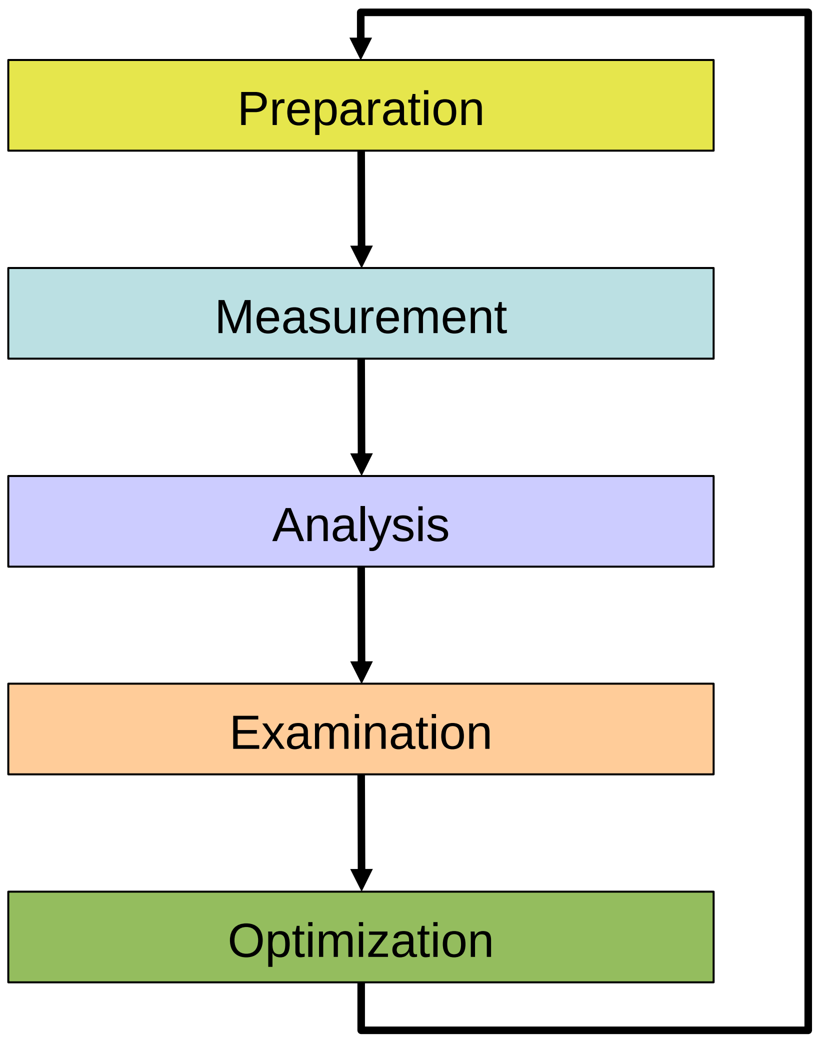
PERFORMANCE FACTORS

- "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

- Successful performance engineering is a combination of
  - The right (parallel) 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!*

PERFORMANCE MEASUREMENT

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

MEASUREMENT METHODS: PROFILING

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

Advantages

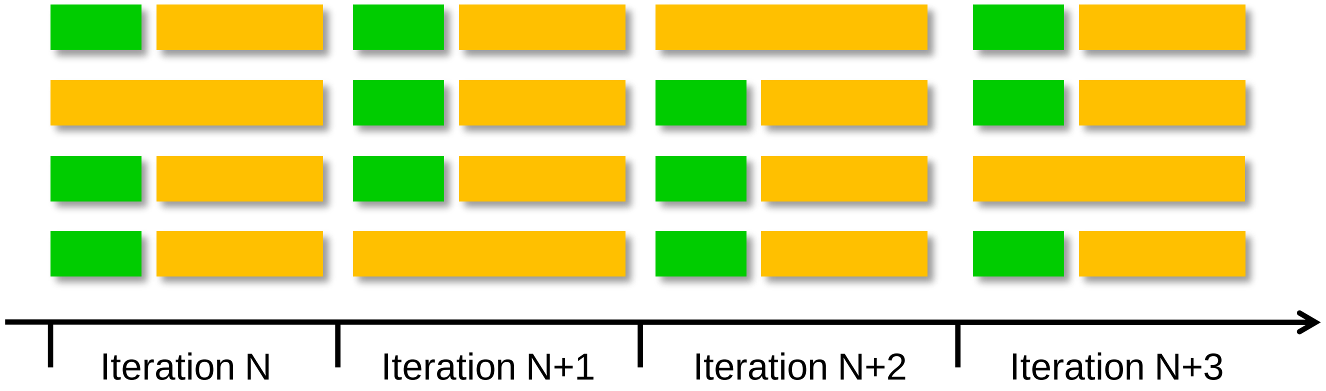
\+ **Works also for long-running programs**

Disadvantages

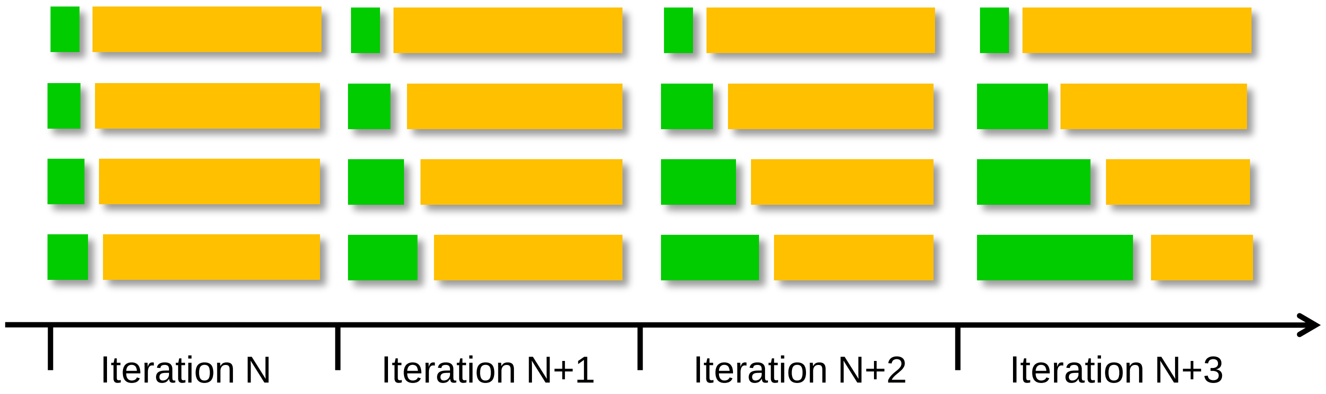
– **Variations over time get lost**

PROFILING: ISSUES RELATED TO "AVERAGING"

- Moving bottleneck across processors can "average out" imbalances



- Imbalance changes over time ⇨ problem worse for 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

⇨ Abstract execution model on level of defined events

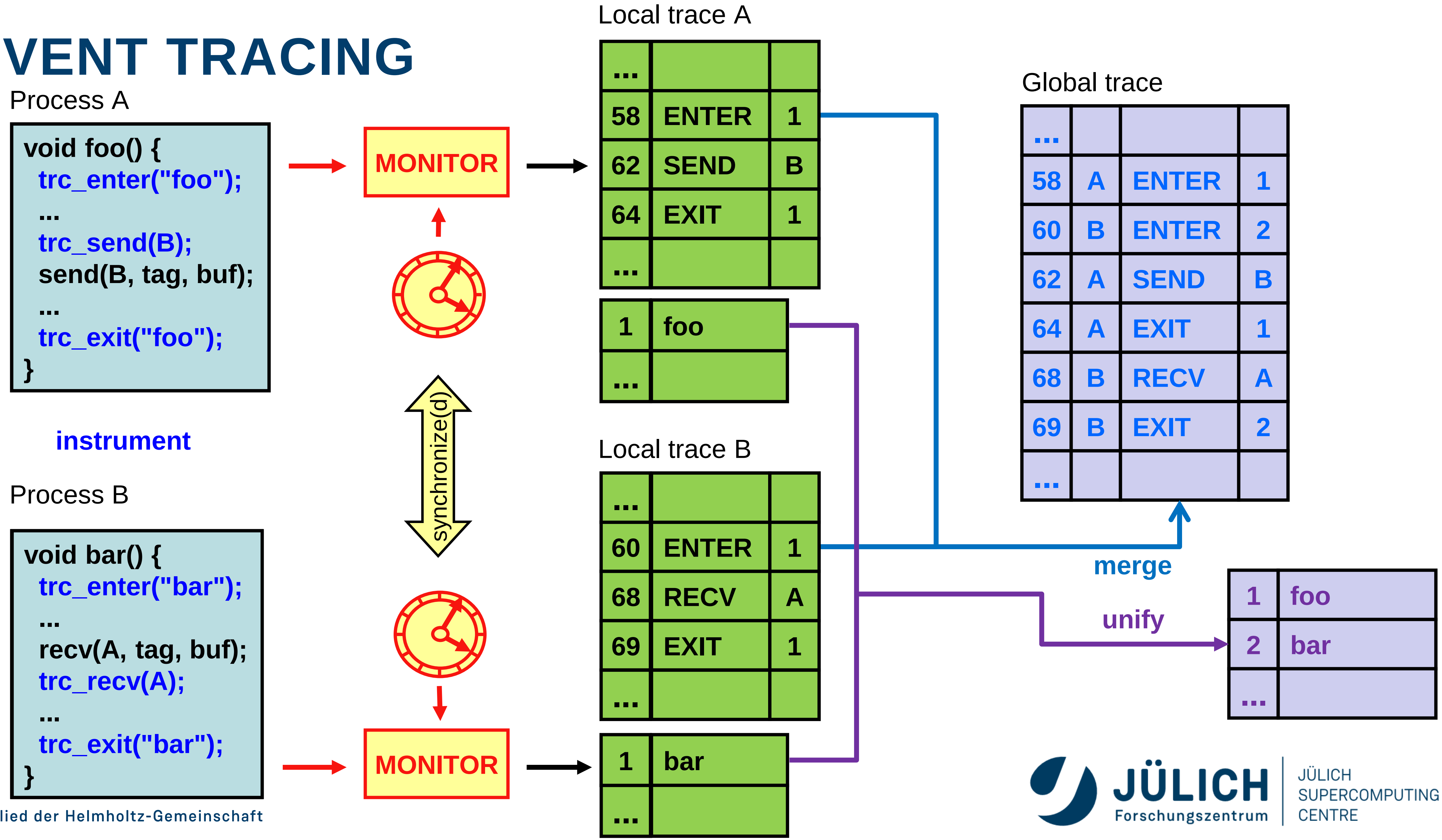
Advantages

+ **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**

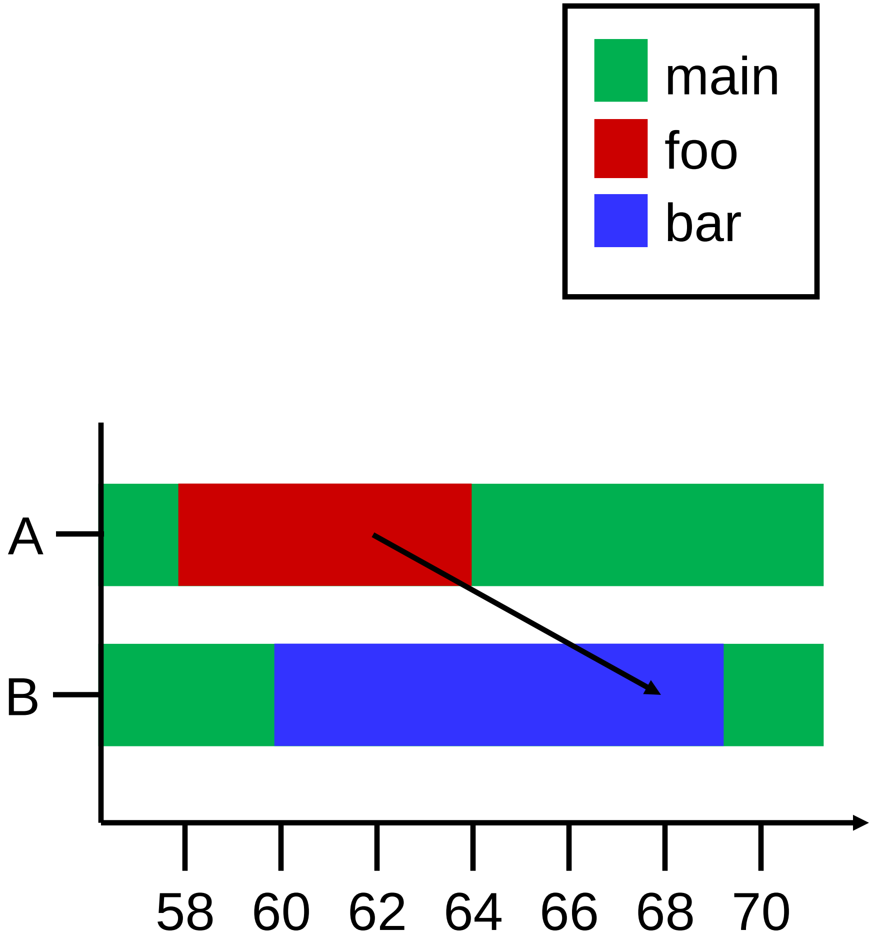
EVENT TRACING



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
...			



CRITICAL ISSUES

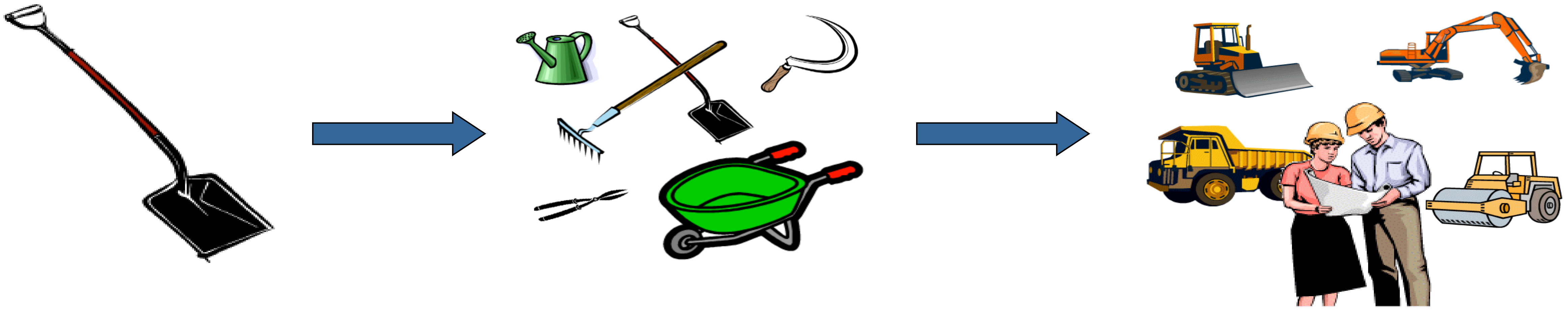
- Accuracy
  - Intrusion overhead
    - Measurement takes time and thus lowers performance
  - Perturbation
    - Measurement alters program behaviour
    - E.g., memory access pattern
  - Accuracy of timers & counters
- Granularity
  - How many measurements?
  - How much information / processing during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

☞ *Tradeoff: Accuracy vs. Expressiveness of data*

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, ...

PERFORMANCE TOOLS (STATUS: MAY 2022)

- Score-P
- Scalasca
- Vampir[Server]
- ARMForge - Performance Reports
- Intel Tools
  - VTune Amplifier XE
  - Intel Advisor
- AMD uProf
- NVIDIA Tools
  - Nsight Systems
  - Nsight Compute
- Darshan
- Extrae

Score-P

Scalable performance measurement infrastructure for parallel codes

- Community-developed open-source
- Replaced tool-specific instrumentation and measurement components of partners
- http://www.score-p.org

TECHNISCHE UNIVERSITÄT DRESDEN

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

German Research School for Simulation Sciences

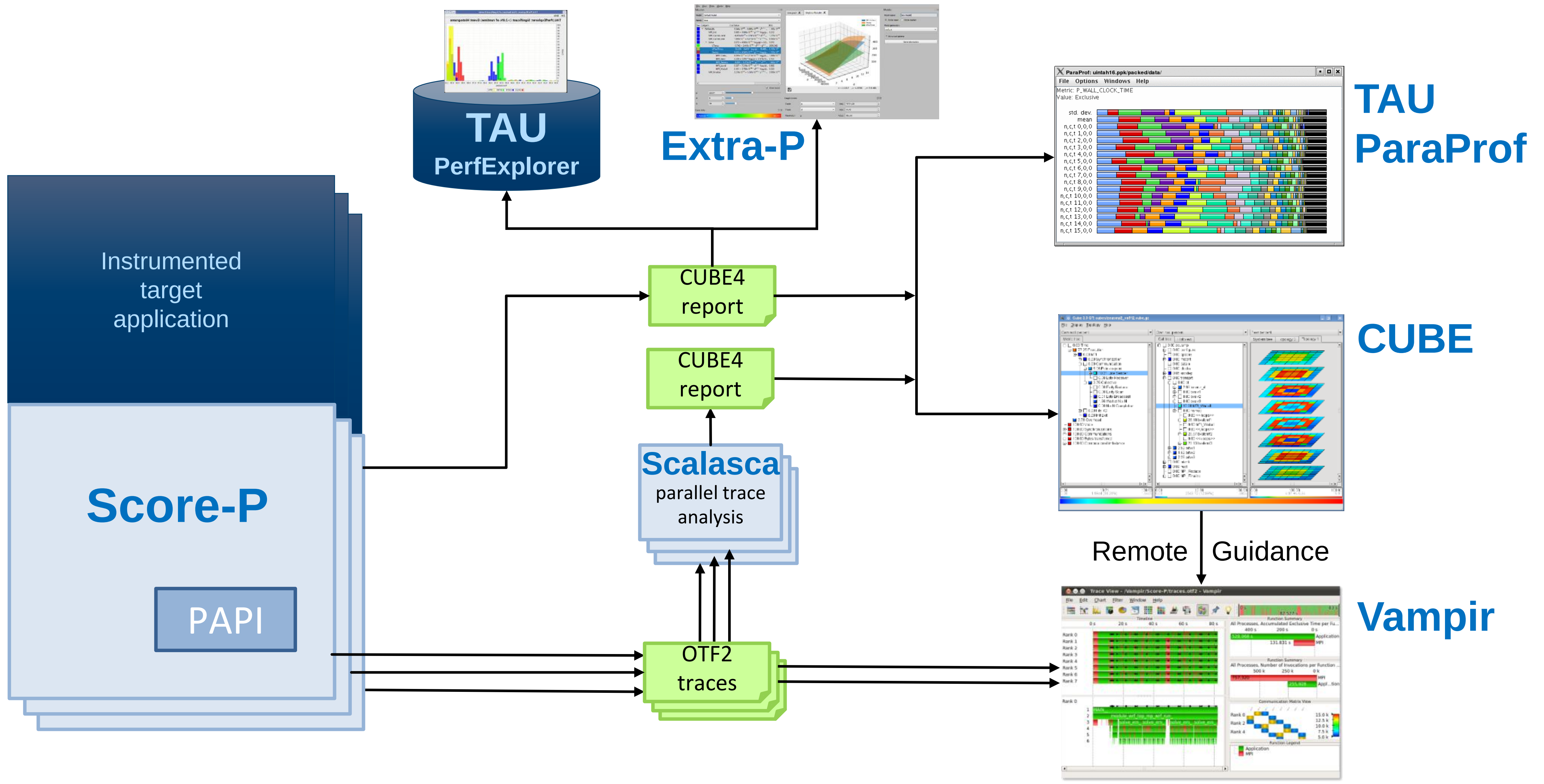
RWTH AACHEN UNIVERSITY

TUM Technische Universität München

UNIVERSITY OF OREGON

Score-P TOOL ECOSYSTEM

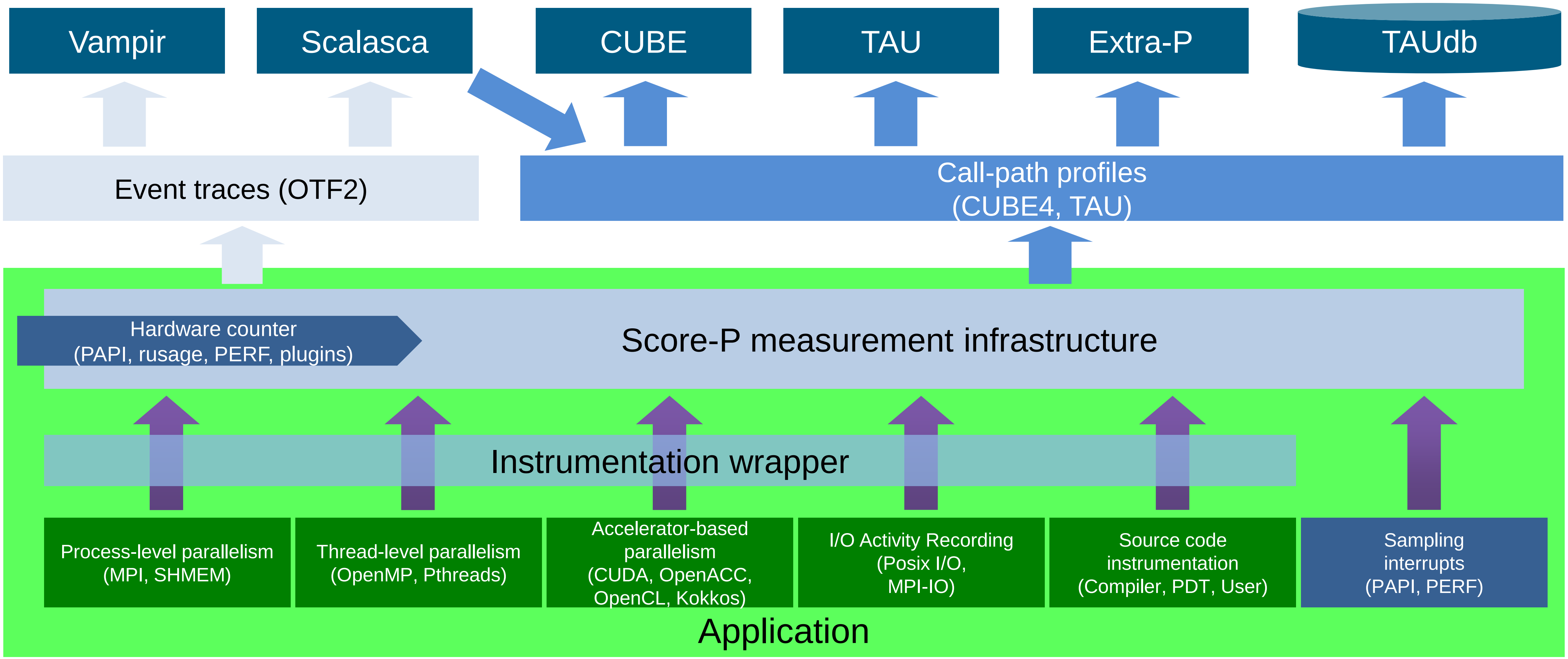
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE



Score-P FUNCTIONALITY

- Provide typical functionality for HPC performance tools
- **Instrumentation** (various methods)
  - Multi-process paradigms (MPI, SHMEM)
  - Thread-parallel paradigms (OpenMP, POSIX threads)
  - Accelerator-based paradigms (OpenACC, CUDA, OpenCL. Kokkos)
  - **In any combination!**
- Flexible **measurement** without re-compilation:
  - Basic and advanced **profile** generation (⇨ CUBE4 format)
  - Event **trace** recording (⇨ OTF2 format)
- Highly scalable I/O functionality
- Support all fundamental concepts of partner’s tools

Score-P ARCHITECURE



WHAT IS THE KEY BOTTLENECK?

- Generate flat MPI profile using Score-P/Scalasca
  - 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

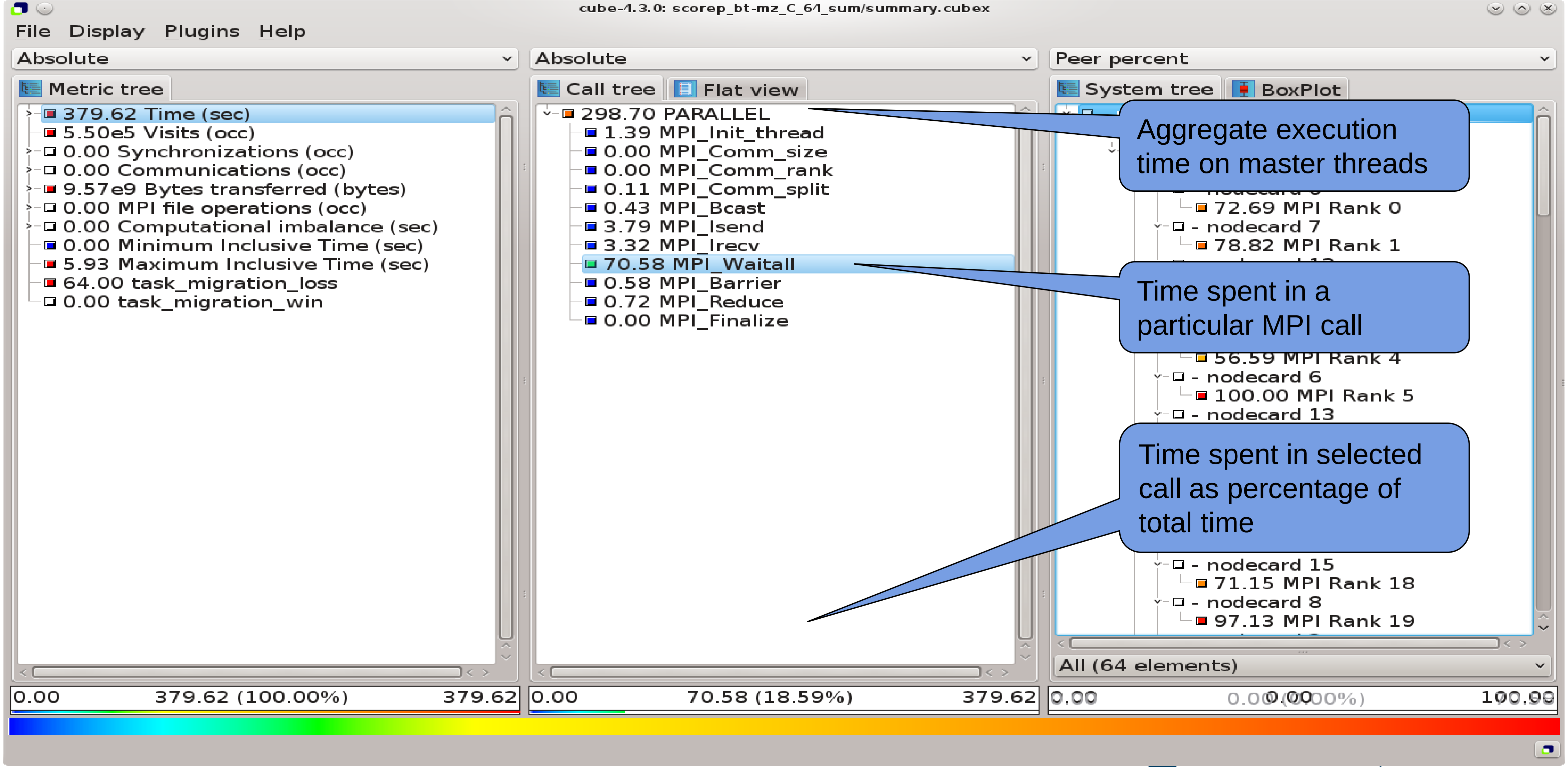
FLAT MPI PROFILE: RECIPE

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 (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 for beginners):
  Automatic compiler-based instrumentation

- Option 2 (for in-depth analysis):
  Manual instrumentation of interesting phases, routines, loops

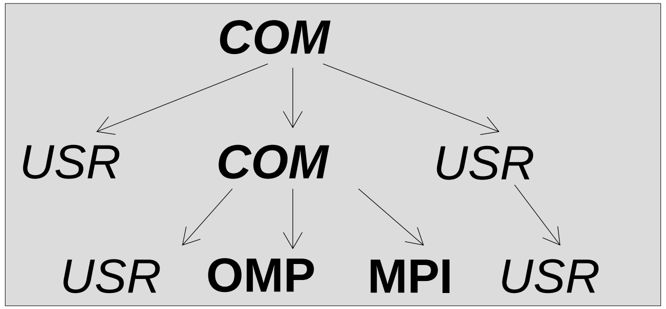
CALL-PATH PROFILE: RECIPE

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 (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_Ppnxt_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

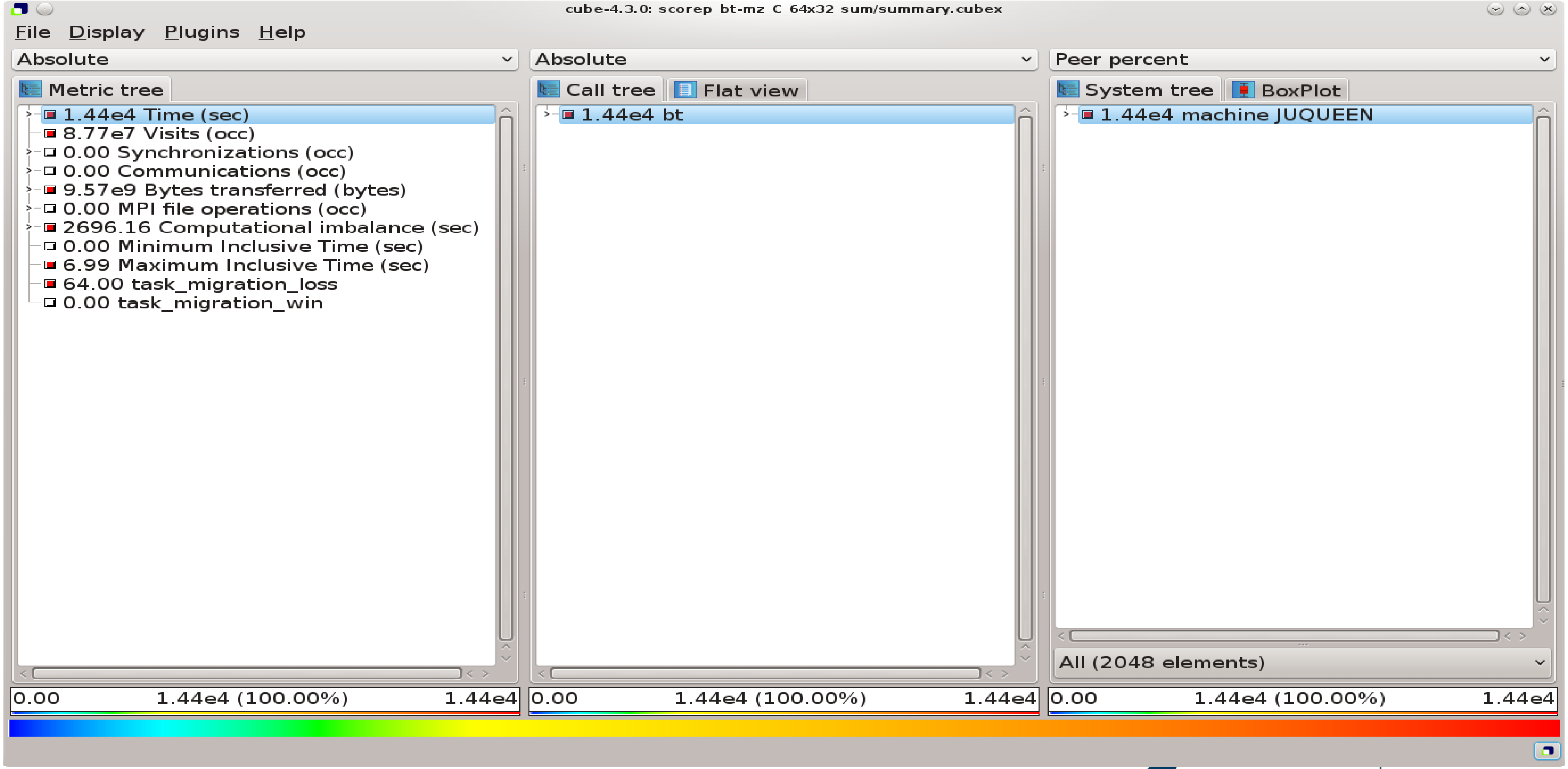
☞List routines to be filtered in simple text file

FILTERING: EXAMPLE

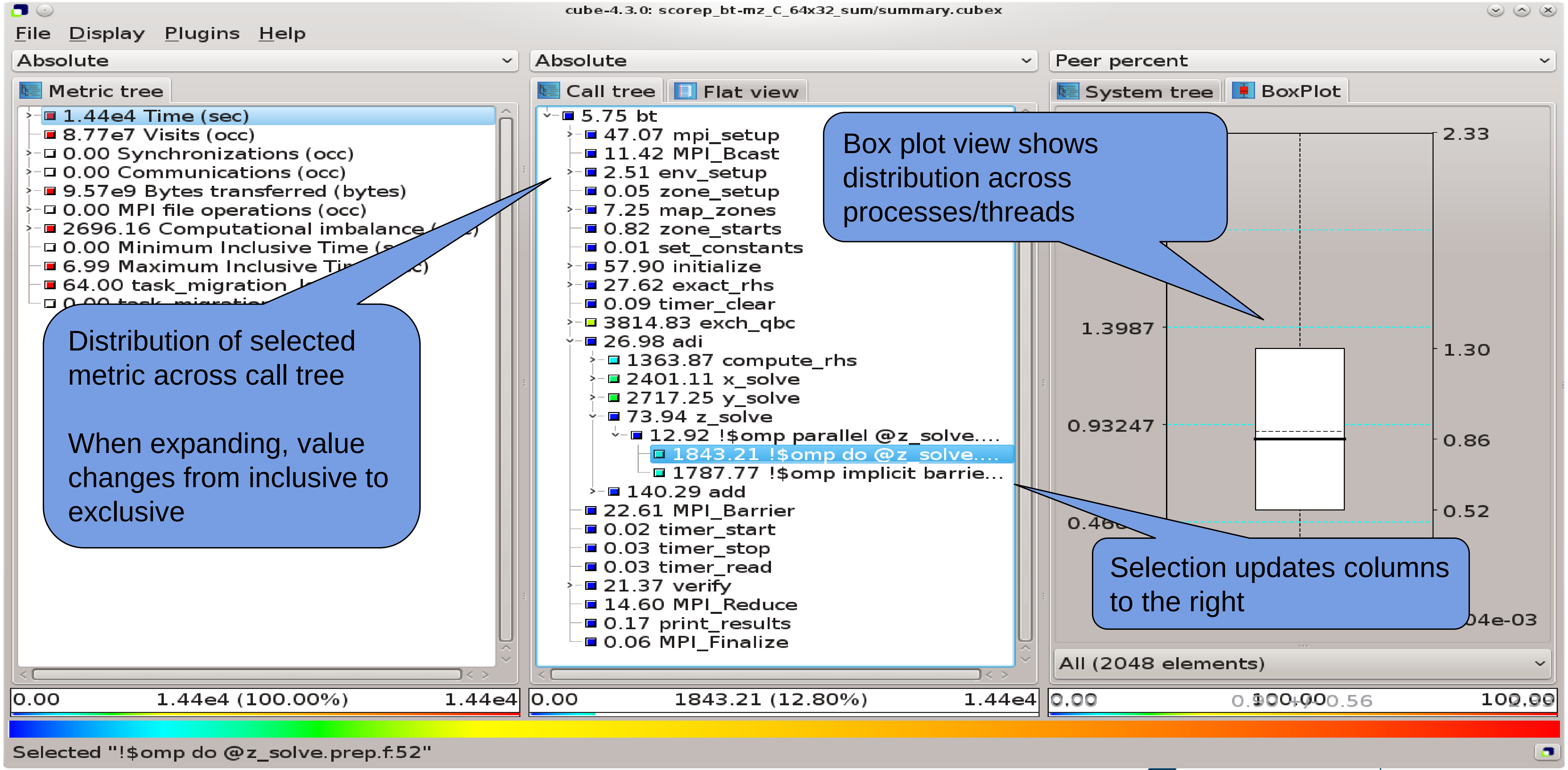
```
% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE
        binvcrhs
        matmul_sub
        matvec_sub
        binvrhs
        lhsinit
        exact_solution
SCOREP_REGION_NAMES_END
```

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

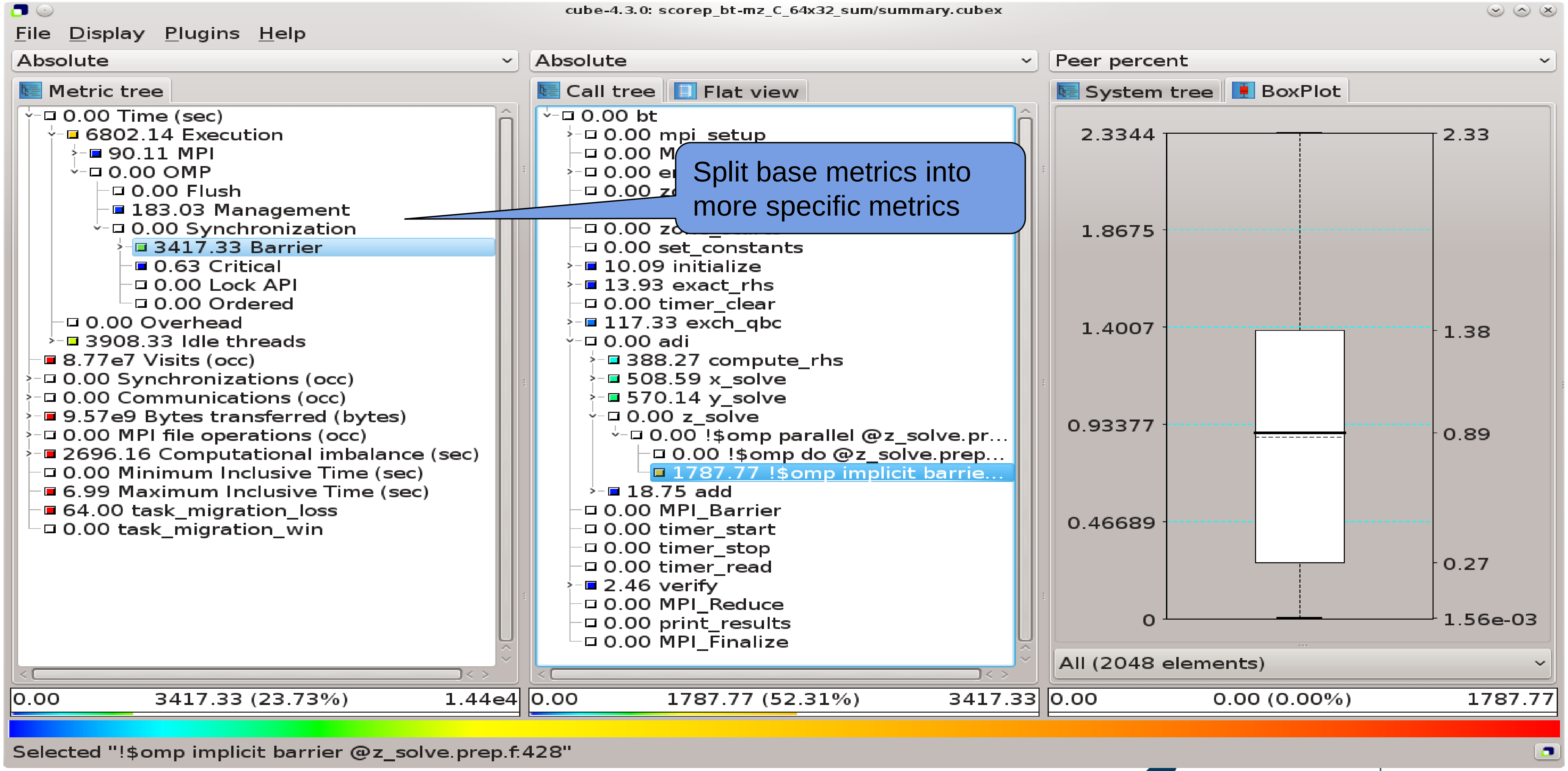
CALL-PATH PROFILE: EXAMPLE (CONT.)



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

SCORE-P GPU MEASUREMENTS

- OpenACC
  - Prefix compiler and linker command with scorep --openacc
  - export ACC_PROFLIB=$SCOREP_ROOT/lib/libscorep_adapter_openacc_event.so
  - export SCOREP_OPENACC_ENABLE=yes
  - yes refers to: regions, wait, enqueue
  - Full list of options in User Guide
- CUDA
  - Prefix compiler and linker command with scorep --cuda
  - export SCOREP_CUDA_ENABLE=yes
  - yes refers to: runtime, kernel, memcpy
  - Full list of options in User Guide

- OpenCL similar (use SCOREP_OPENCL_ENABLE=yes)

WHY IS THE BOTTLENECK THERE?

- This is highly application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Automatic/manual 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
  - PAPI (Performance API)

PAPI

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

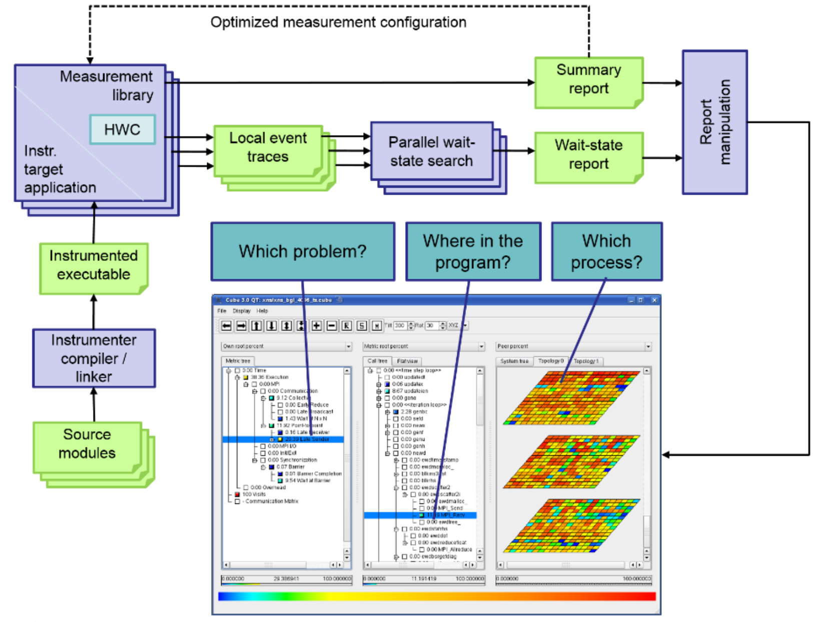
- High-level interface:
  - Predefined standard events, e.g. PAPI_FP_OPS
  - Availability and definition of events varies between platforms
  - List of available counters: `papi_avail (-d)`
- Low-level interface:
  - Provides access to all machine specific counters
  - Non-portable
  - More flexible
  - List of available counters: `papi_native_avail`

SCALASCA

http://www.scalasca.org/

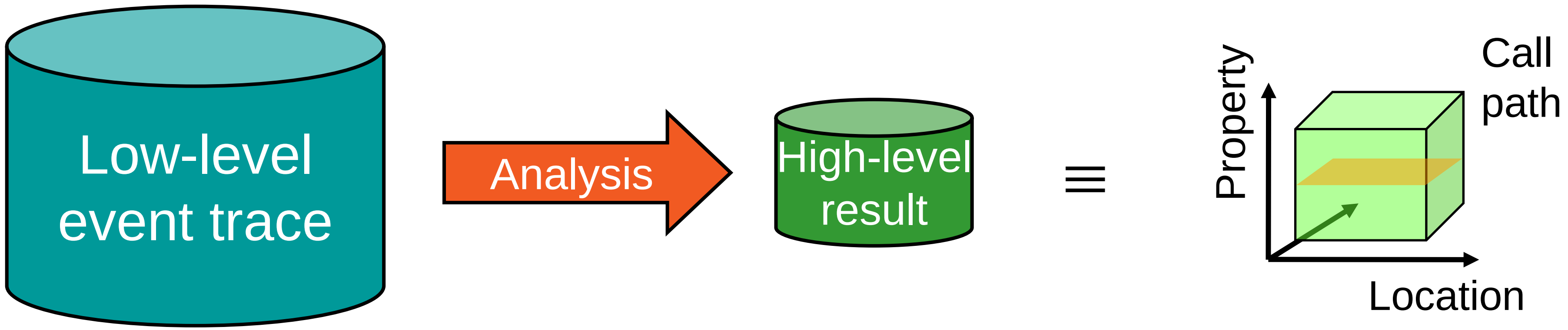
- Scalable Analysis of Large Scale Applications

- Approach
  - **Instrument** C, C++, and Fortran parallel applications (**with Score-P**)
  - Option 1: **scalable call-path profiling**
  - Option 2: **scalable event trace analysis**
    - **Collect** event traces
    - **Process trace in parallel**
      - Wait-state analysis
      - Delay and root-cause analysis
      - Critical path analysis
    - **Categorize and rank** results



AUTOMATIC TRACE ANALYSIS

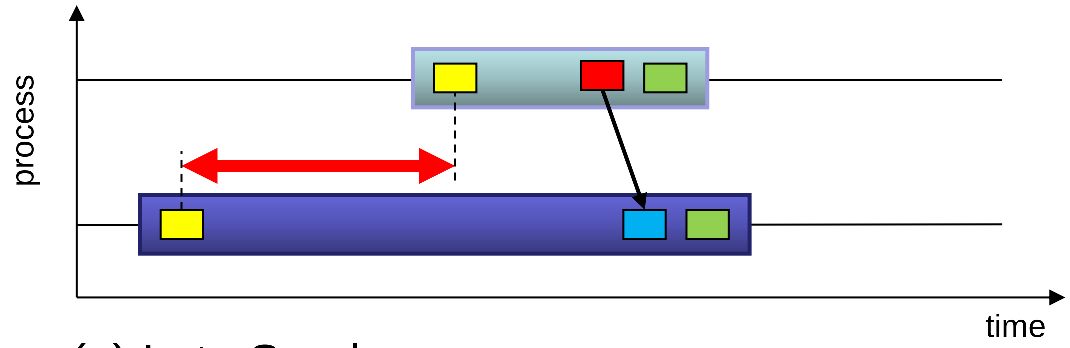
- Automatic search for patterns of inefficient behaviour
- Classification of behaviour & quantification of significance
- Identification of delays as root causes of inefficiencies



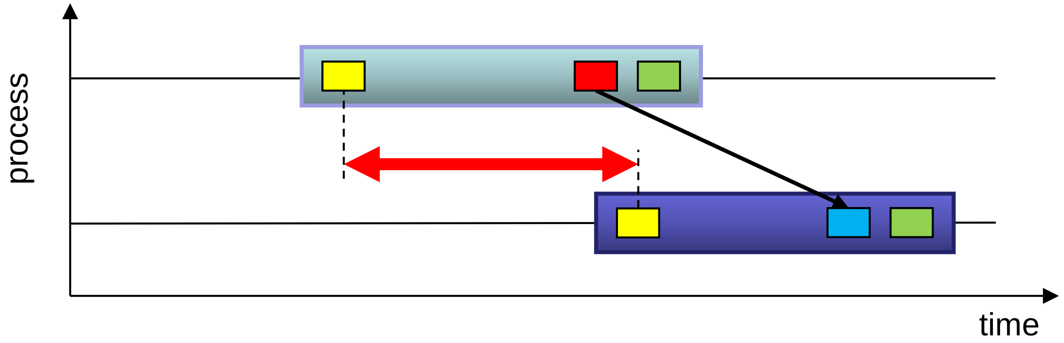
- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability

EXAMPLE MPI WAIT STATES

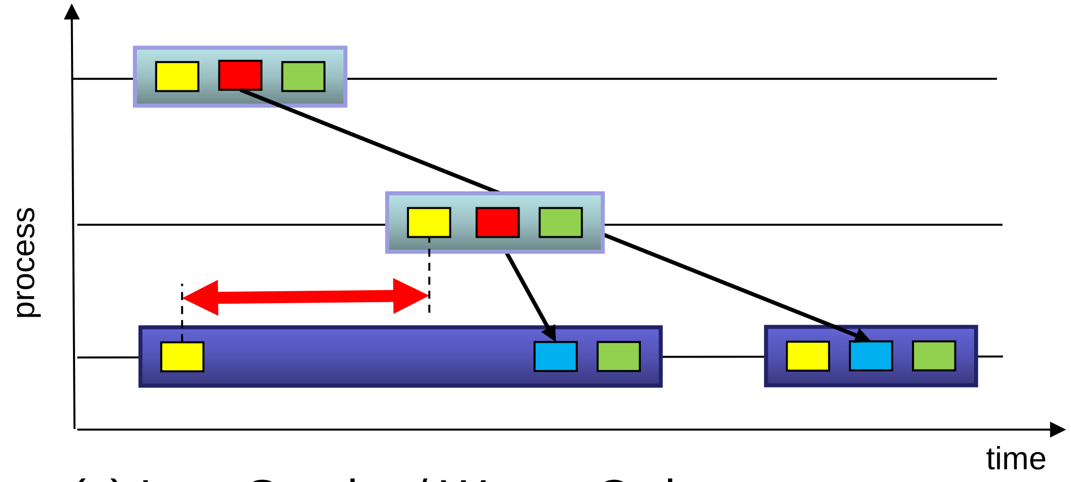
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE



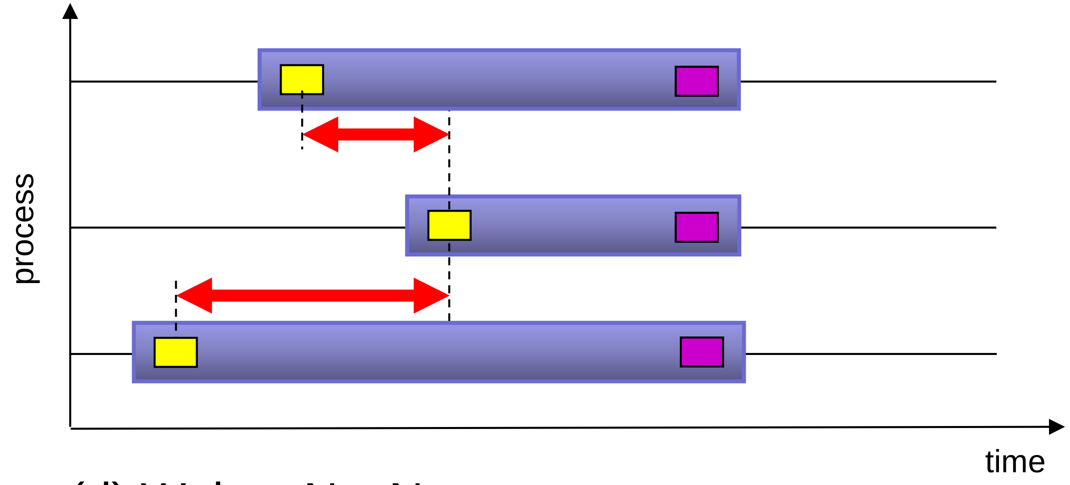
(a) Late Sender



(b) Late Receiver



(c) Late Sender / Wrong Order

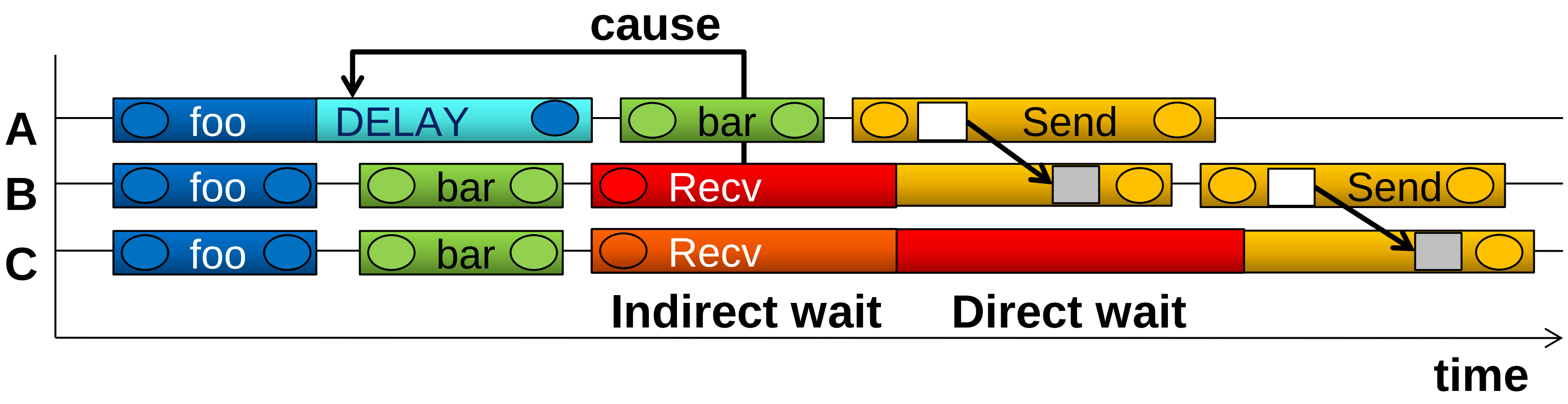


(d) Wait at N x N

ENTER EXIT SEND RECV COLLEXIT

SCALASCA ROOT CAUSE ANALYSIS

- **Root-cause analysis**
  - Wait states typically caused by load or communication imbalances earlier in the program
  - Waiting time can also propagate (e.g., indirect waiting time)
  - Enhanced performance analysis to find the root cause of wait states
- **Approach**
  - Distinguish between direct and indirect waiting time
  - Identify call path/process combinations delaying other processes and causing first order waiting time
  - Identify original **delay**



TRACE GENERATION & ANALYSIS W/ SCALASCA

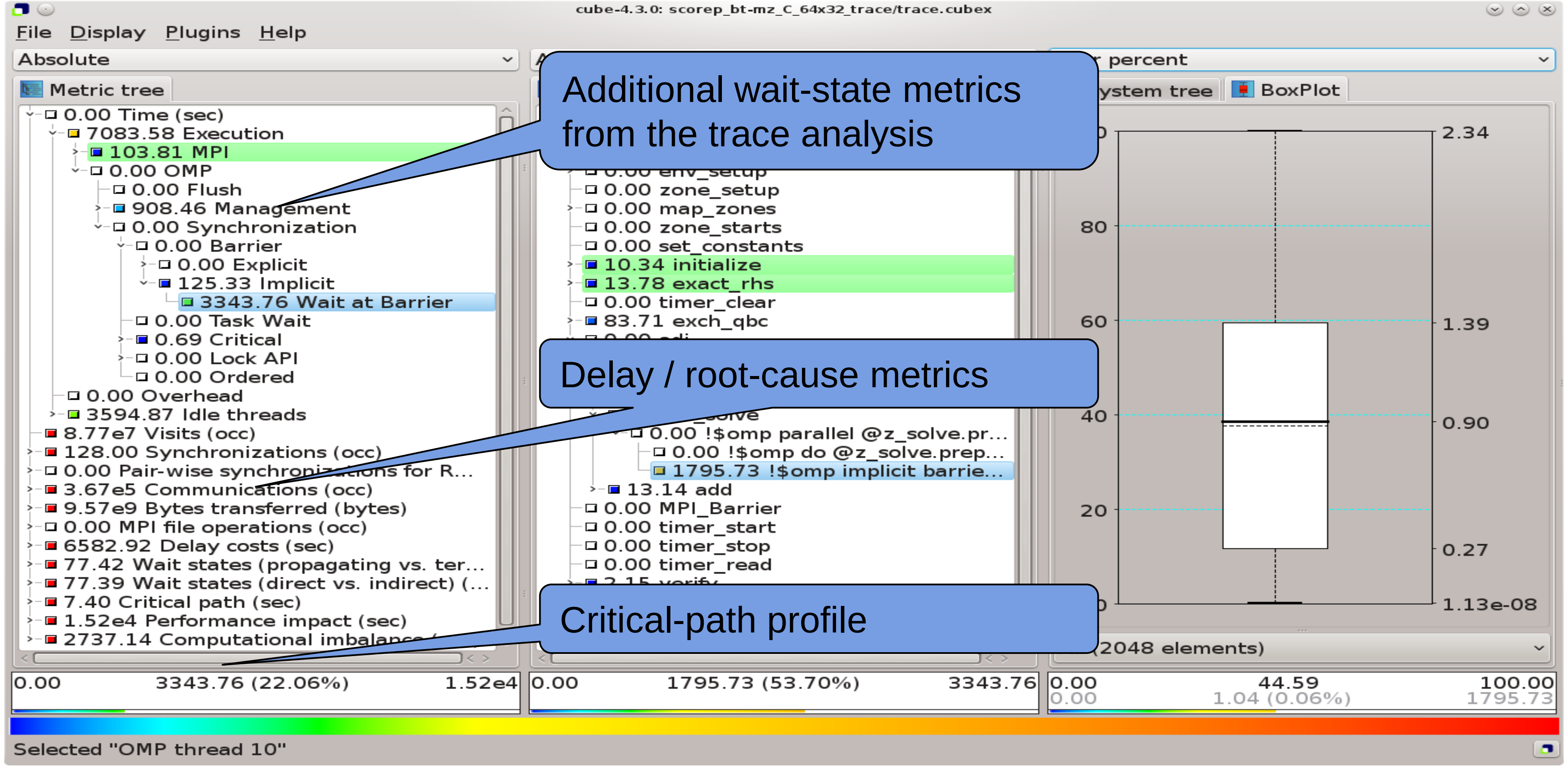
- Enable trace collection & analysis using "`-t`" option of "`scalasca -analyze`":

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

module load ENV Score-P 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-Ps 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

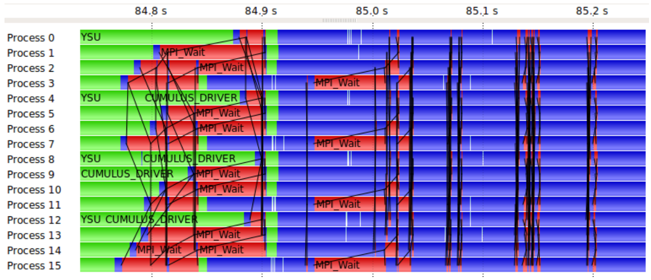
- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Completely manual analysis
  - Too many details can hide the relevant parts

EVENT TRACE VISUALIZATION WITH VAMPIR

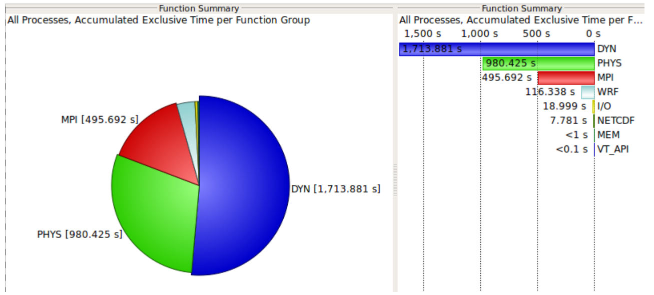
- Visualization of dynamic runtime behaviour at any level of detail along with statistics and performance metrics
- Alternative and supplement to automatic analysis

- **Typical questions that Vampir helps to answer**
  - What happens in my application execution during a given time in a given process or thread?
  - How do the communication patterns of my application execute on a real system?
  - Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

- **Timeline charts**
  - Application activities and communication along a time axis



- **Summary charts**
  - Quantitative results for the currently selected time interval



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

VAMPIR PERFORMANCE CHARTS

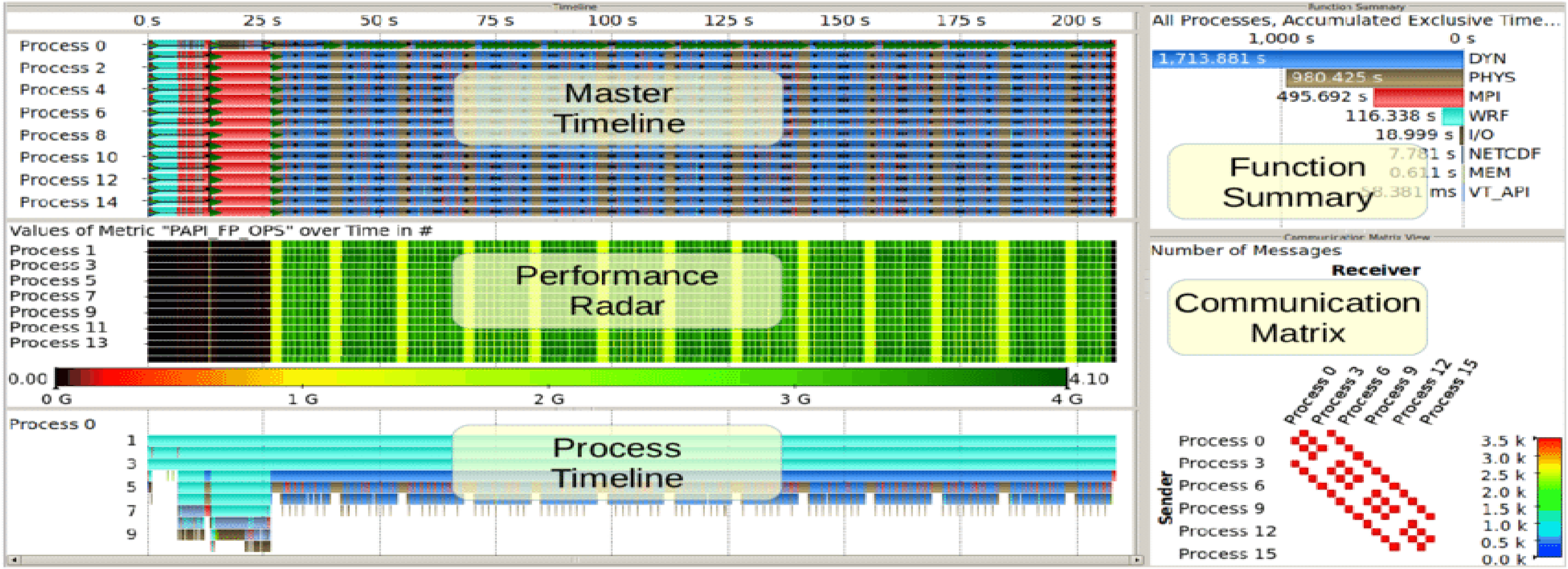
Timeline Charts

Master Timeline	➡	*all threads' activities*
Process Timeline	➡	*single thread's activities*
Summary Timeline	➡	*all threads' function call statistics*
Performance Radar	➡	*all threads' performance metrics*
Counter Data Timeline	➡	*single threads' performance metrics*
I/O Timeline	➡	*all threads' I/O activities*

Summary Charts

Function Summary	Process Summary
Message Summary	Communication Matrix View
I/O Summary	Call Tree

VAMPIR DISPLAYS



ARM 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

- https://www.arm.com/products/development-tools/server-and-hpc/performance-reports
- Note: License limited to 512 processes (with unlimited number of threads)

EXAMPLE PERFORMANCE REPORTS

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

The total wallclock time was spent as follows:

CPU	56.5%	Time spent running application code. High values are usually good. This is **average**; check the CPU performance section for optimization advice.
MPI	43.5%	Time spent in MPI calls. High values are usually bad. This is **average**; check the MPI breakdown for advice on reducing it.
I/O	0.0%	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.

NVIDIA TOOLS -- LEGACY TRANSITION

nvprof
Command-line tool

NVIDIA Visual Profiler
Standalone UI

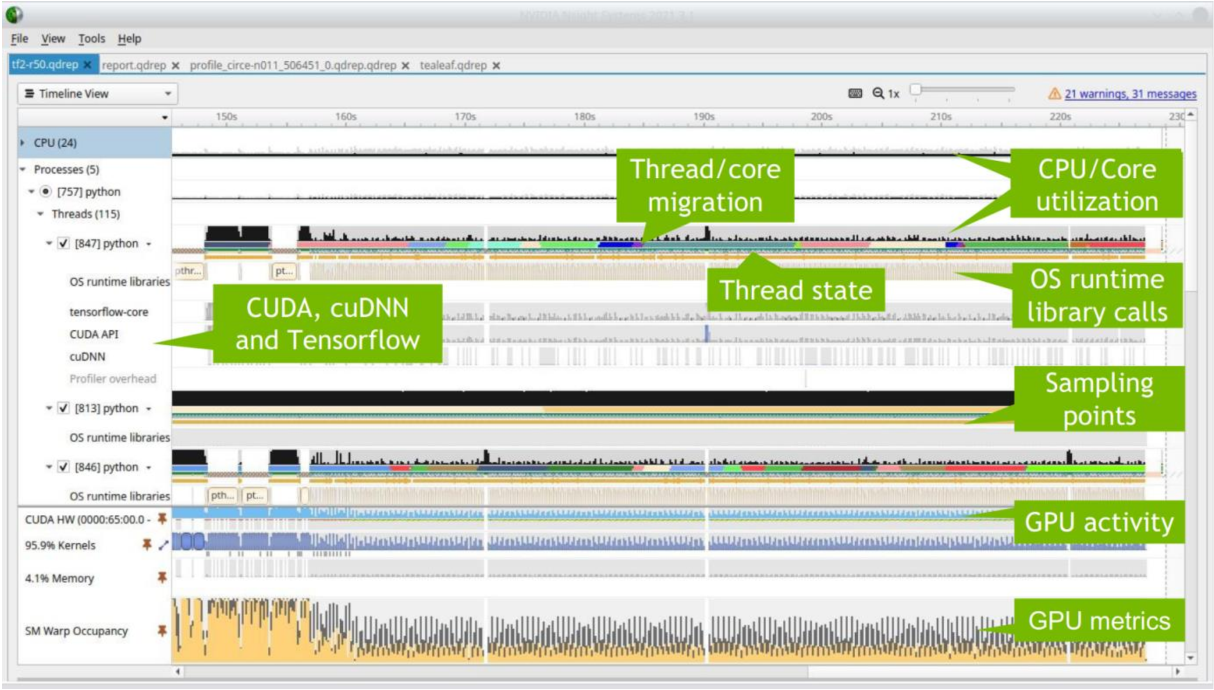
Nsight Systems
Standalone GUI+CLI

CPU-GPU interactions & triage
Low overhead capture
GPU compute & graphics
Faster GUI + more data

Nsight Compute
Standalone GUI+CLI

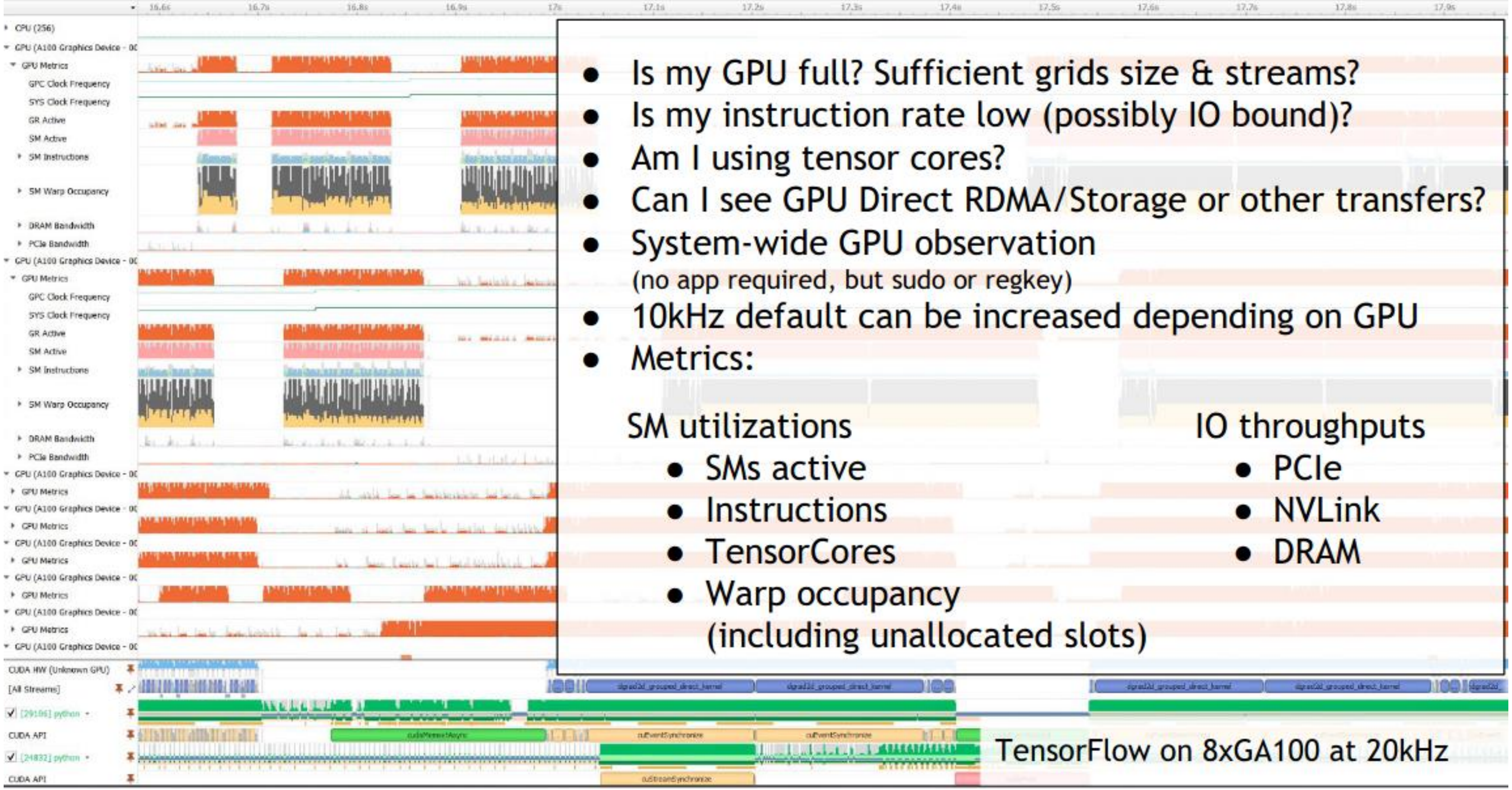
GPU CUDA kernel analysis & debug
Very high freq GPU perf counters
Compare results (diff)
Incredible statistics & customizable

NSIGHT SYSTEM



- System-wide application tuning
- Locate optimization opportunities
  - Visualize millions of events on a timeline
  - See gaps of unused CPU and GPU time
- Balance workloads across multiple CPUs and GPUs
  - CPU utilization and thread state
  - GPU streams, kernels, memory transfers, etc.
- Multi-platform support
  - Linux, Windows and Mac OS X (host-only)
  - x86-64, Power9, ARM server, Tegra (Linux & QNX)

GPU METRIC SAMPLING



- Is my GPU full? Sufficient grids size & streams?
- Is my instruction rate low (possibly IO bound)?
- Am I using tensor cores?
- Can I see GPU Direct RDMA/Storage or other transfers?
- System-wide GPU observation
  (no app required, but sudo or regkey)
- 10kHz default can be increased depending on GPU
- Metrics:

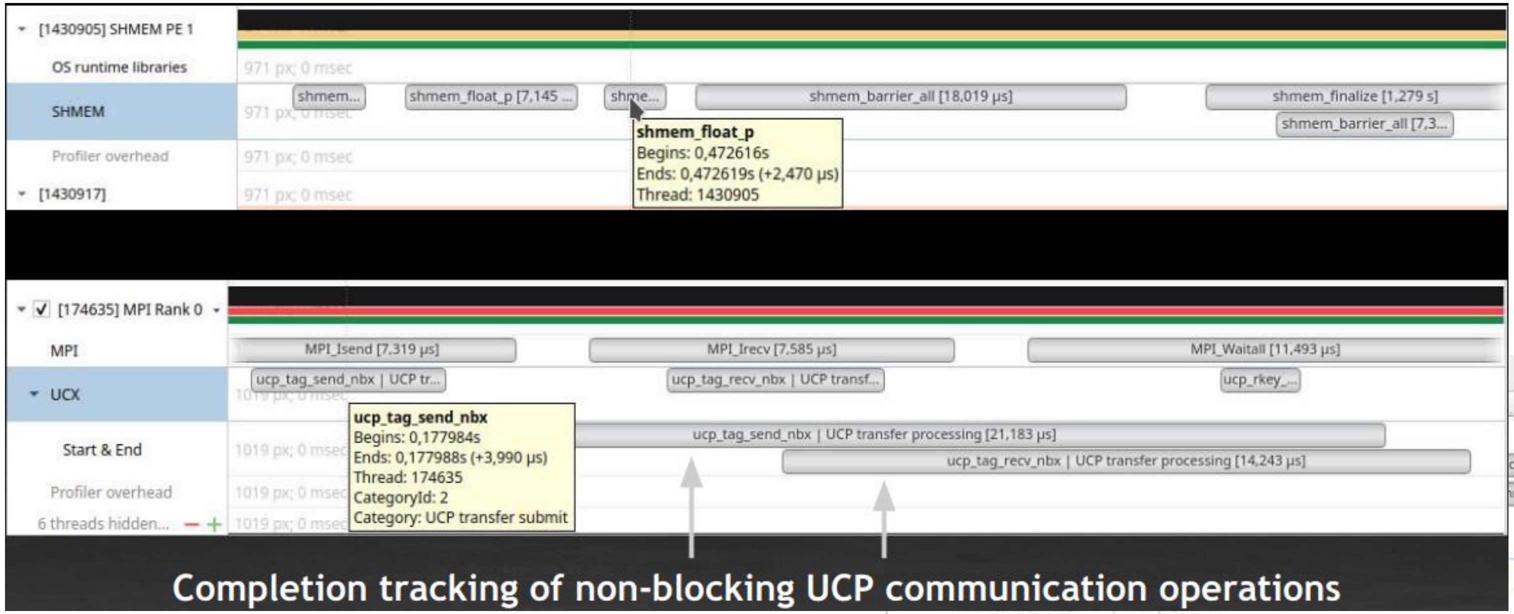
SM utilizations
- SMs active
- Instructions
- TensorCores
- Warp occupancy
  (including unallocated slots)

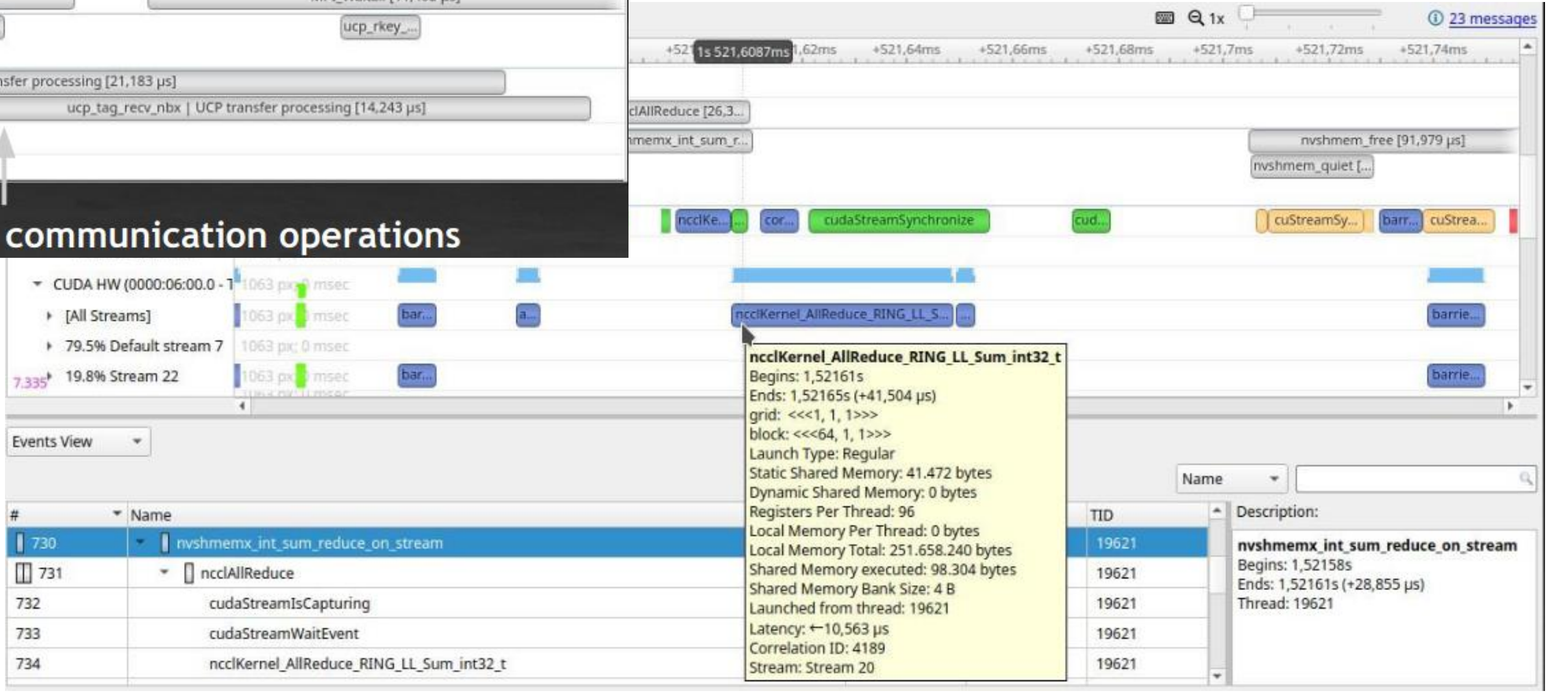
IO throughputs
- PCIe
- NVLink
- DRAM

TensorFlow on 8xGA100 at 20kHz

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

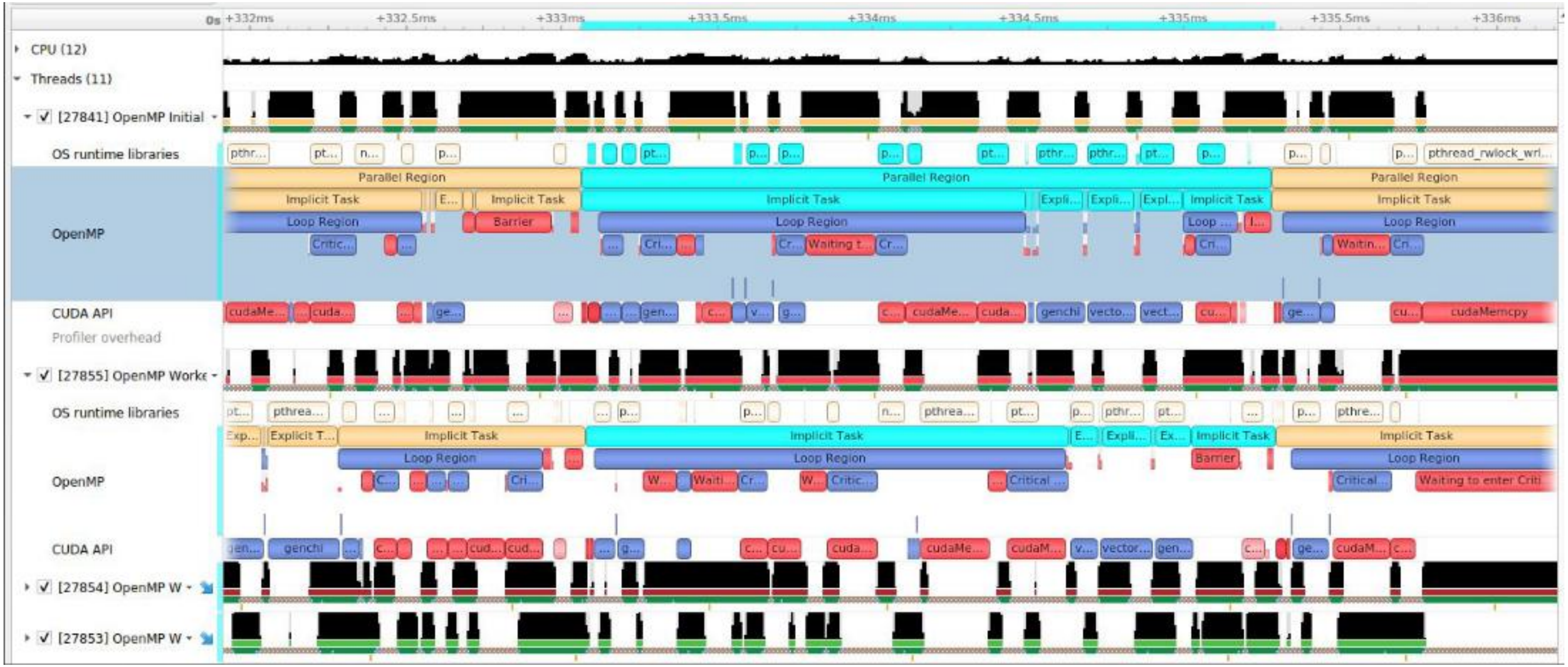
MULTI NODE SUPPORT – SHMEM, MPI, UCX, AND NCCL





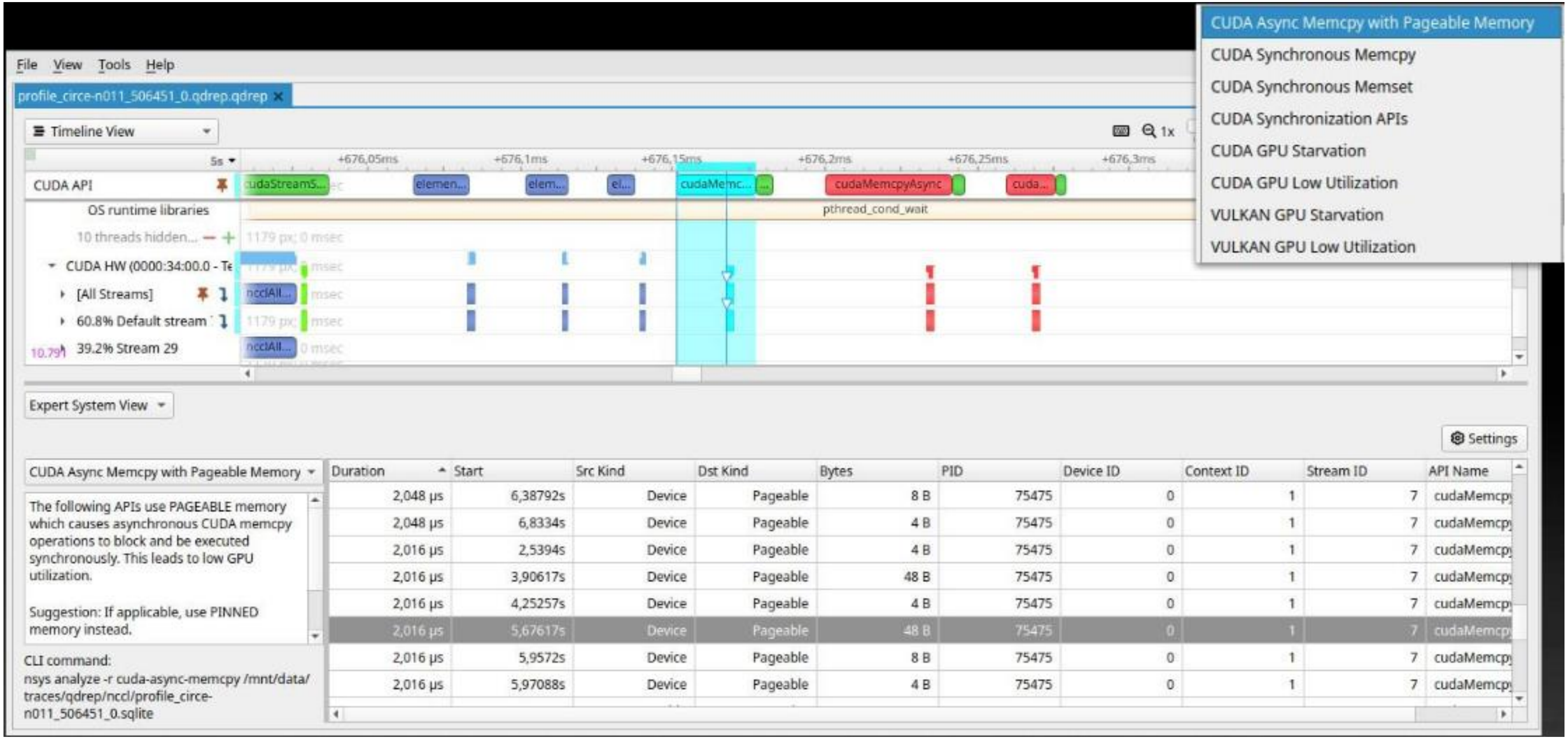
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

OPENMP



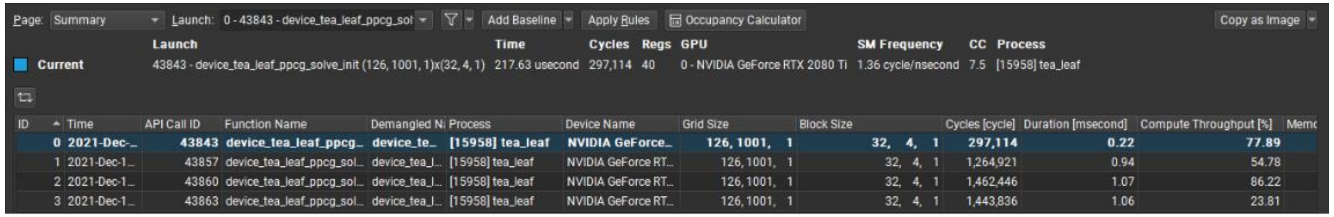
OMPT-capable OpenMP runtime required

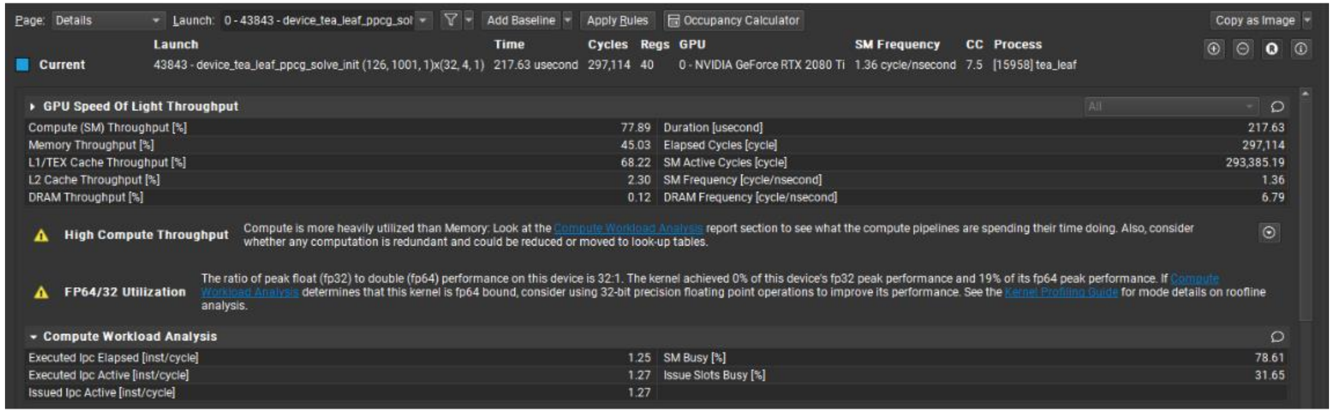
EXPERT SYSTEM



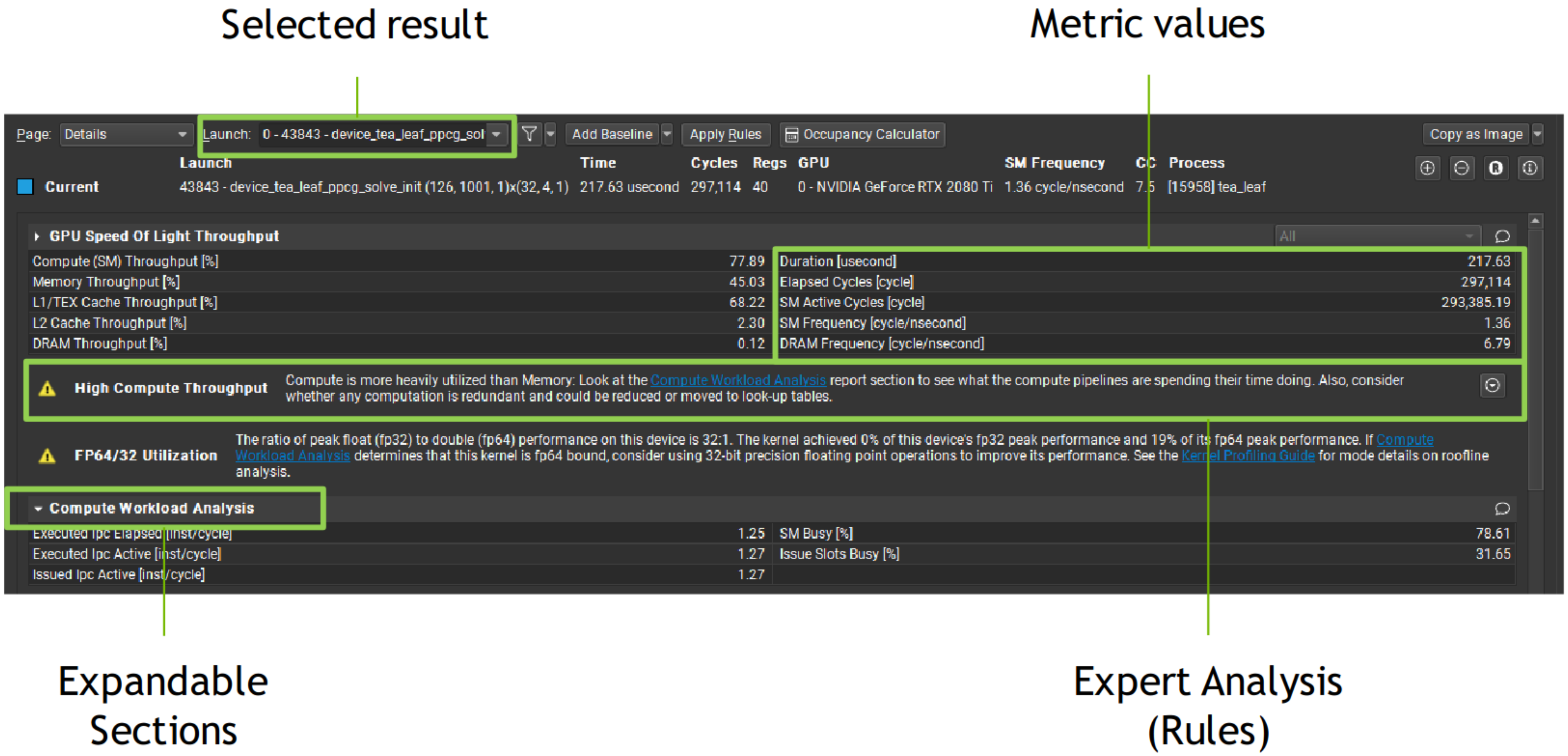
NSIGHT COMPUTE

- Interactive CUDA kernel profiler
- Targeted metric sections for various performance aspects
- Customizable data collection and presentation (tables, charts, ...)
- GUI and CLI
- Python-based API for guided analysis and post-processing
- Support for remote profiling across machines and platforms





PROFILER REPORT

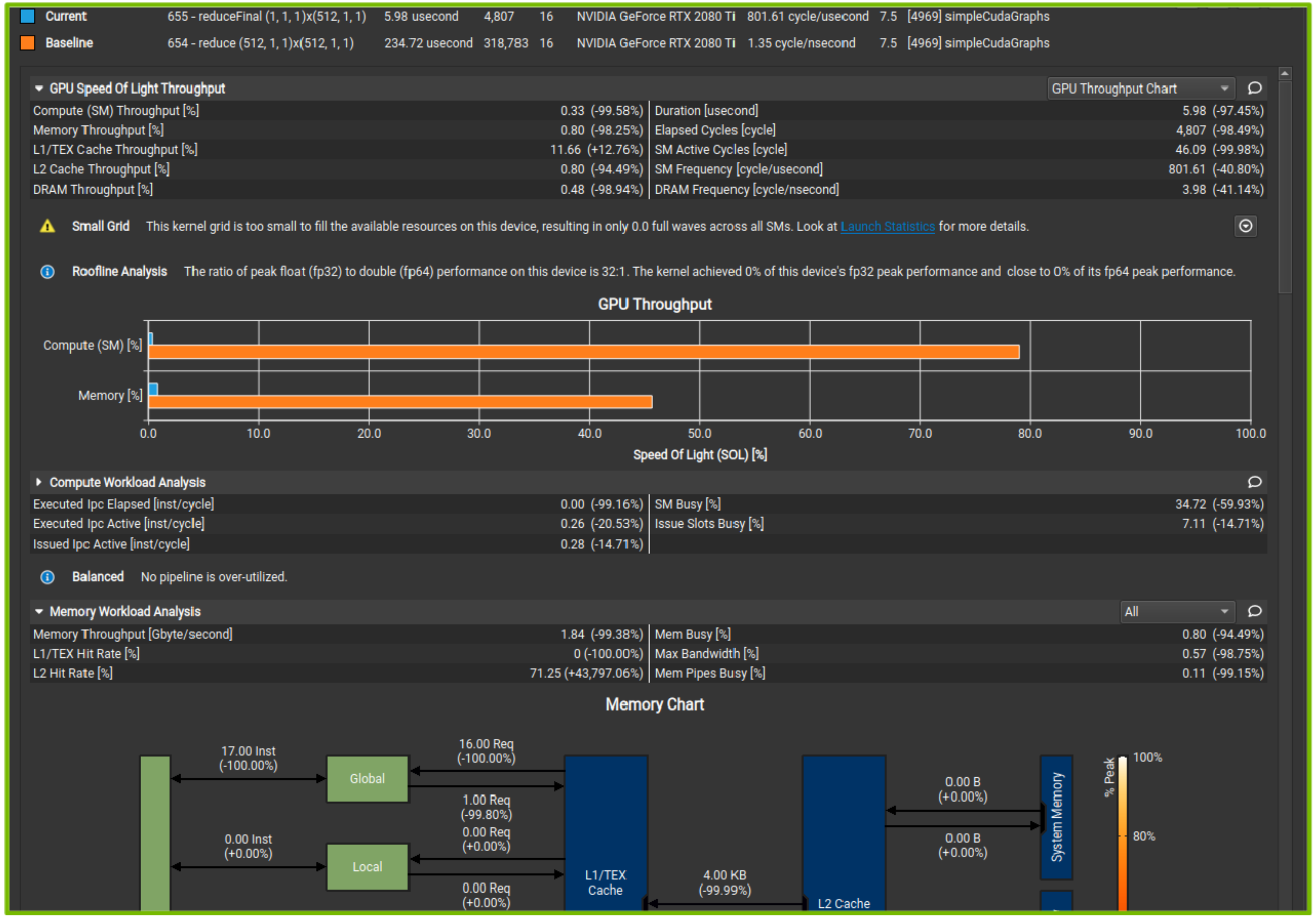


DATA TRANSFER ANALYSIS

- Detailed memory workload analysis chart and tables
- Shows transferred data or throughputs
- Tooltips provide metric names, calculation formulas and detailed background info

BASELINE COMPARISON

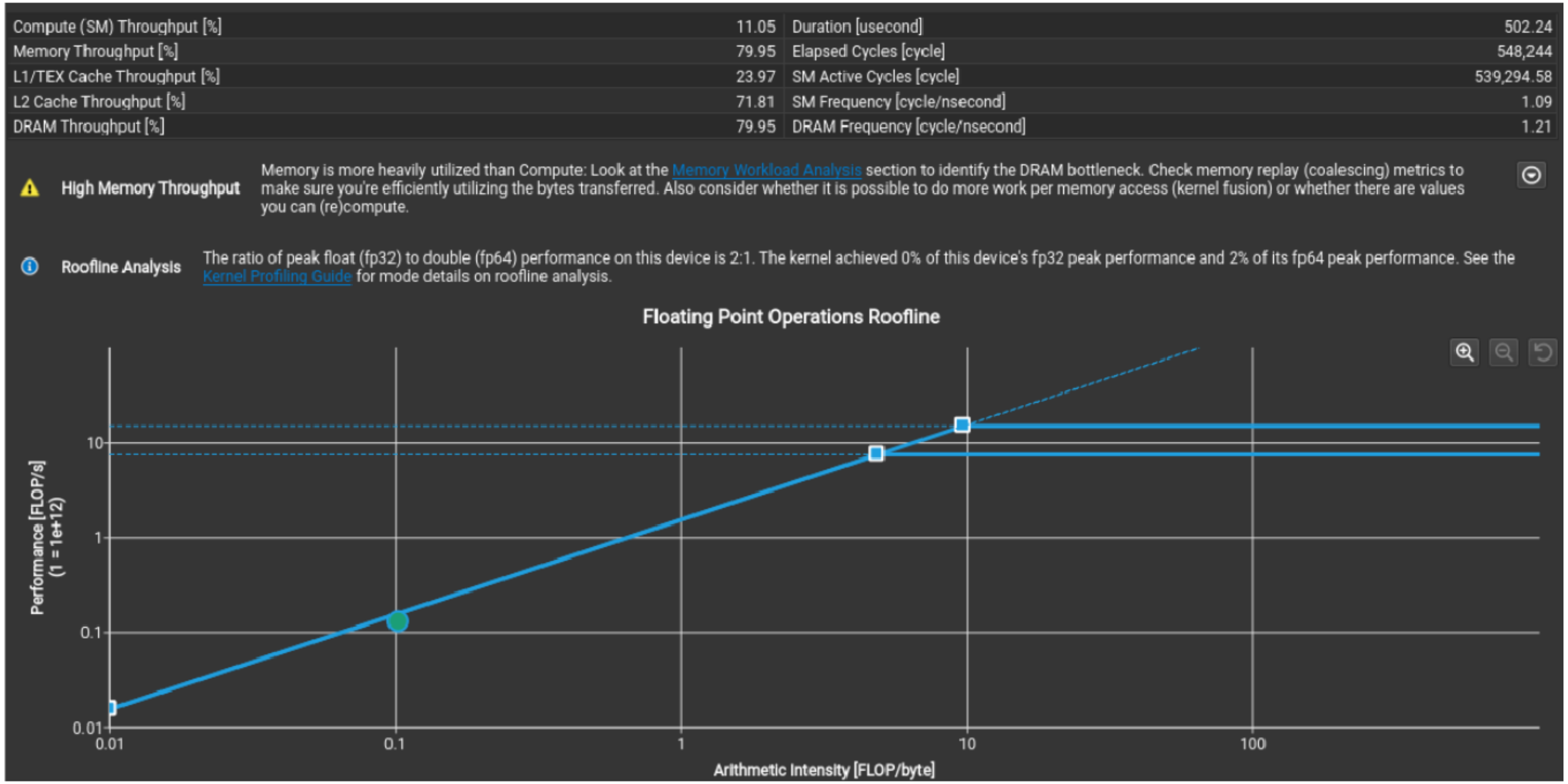
- Comparison of results directly within the tool with "Baselines“
- Supported across kernels, reports, and GPU architectures



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

ROOFLINE ANALYSIS

- Determine whether the application is memory bound or compute bound
- Guided analysis points to detailed analysis of the most severe problem



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

DARSHAN

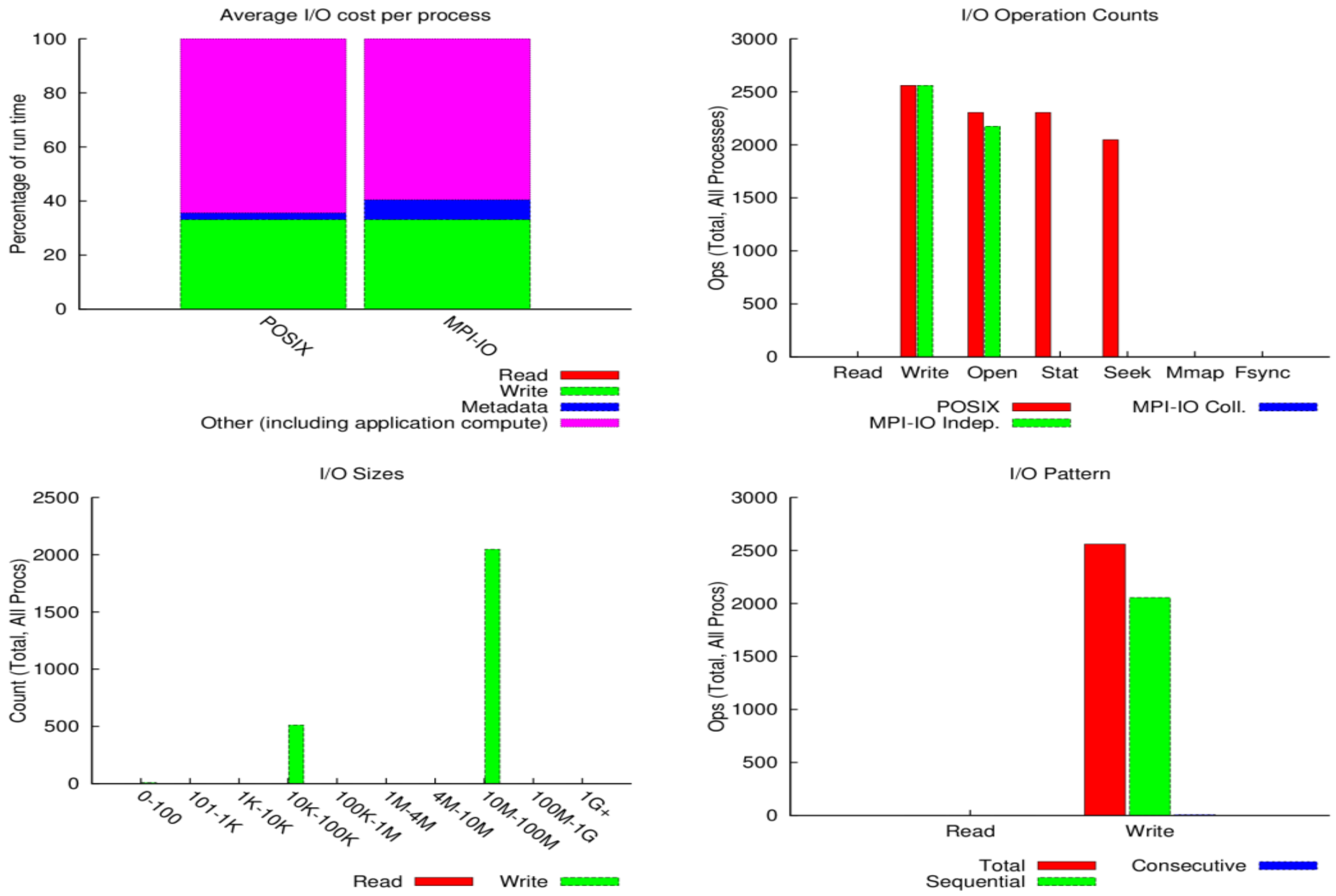
- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

- Supports POSIX, MPI-IO, HDF5, PnetCDF, …
- Binary log file written at exit post-processed into PDF report

- http://www.mcs.anl.gov/research/projects/darshan/
- Open Source: installed on many HPC systems

EXAMPLE DARSHAN REPORT EXTRACT

jobid: | uid: | nprocs: 4096 | runtime: 175 seconds



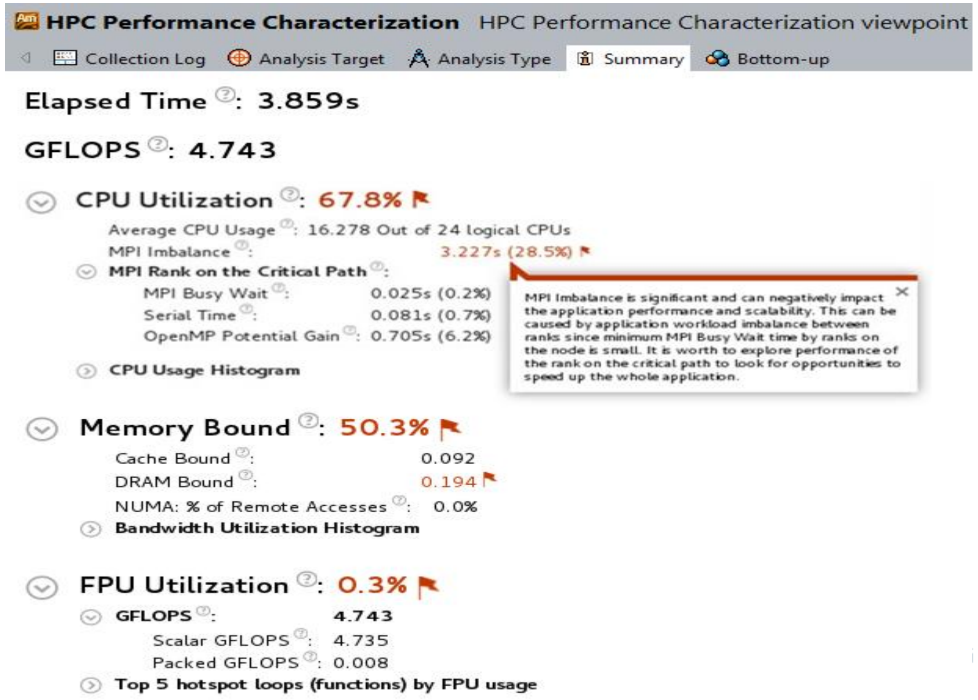
VTUNE AMPLIFIER XE

- Feature-rich profiler for Intel platforms
- Supports Python, C/C++ and Fortran
- MPI support continuously improving

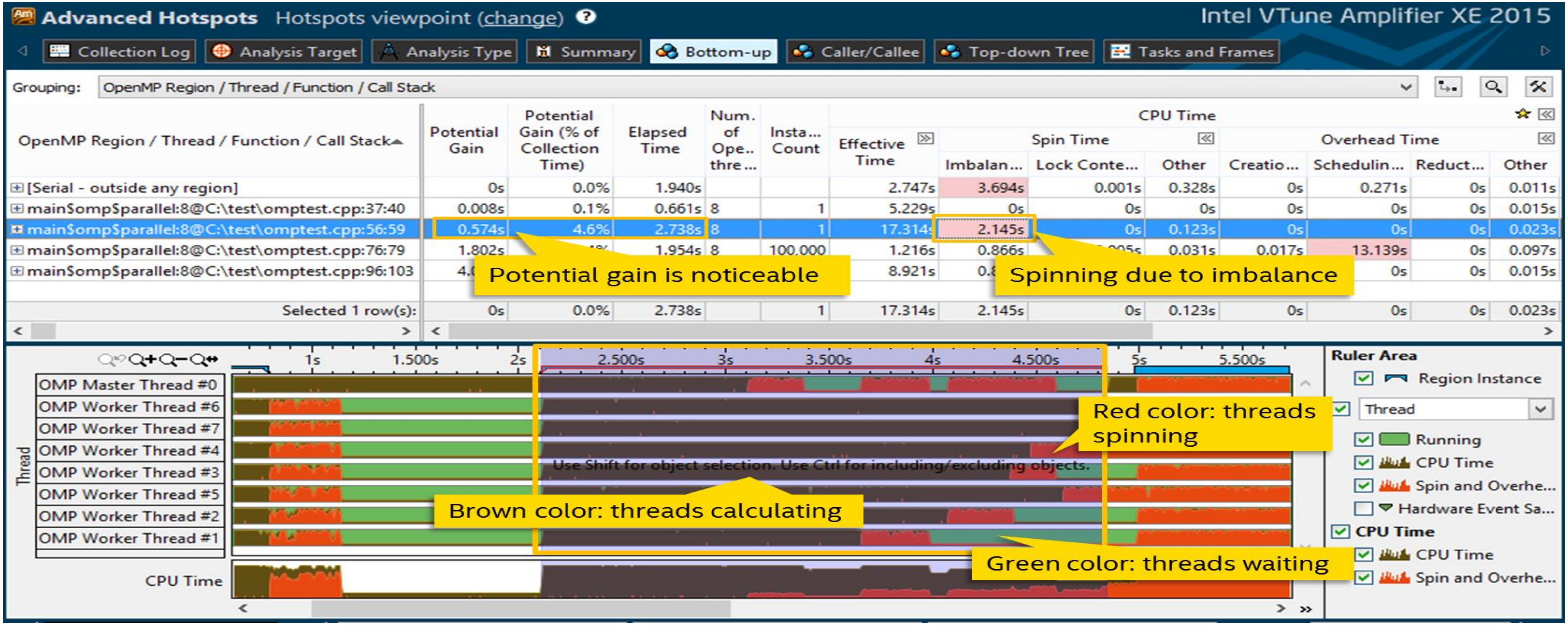
- Lock and Wait analysis for OpenMP and TBB
- HPC analysis for quick overview

- Bandwidth and memory analysis
- I/O analysis

- OpenCL and GPU profiling
  (no CUDA, Intel iGPU only)



INTEL VTUNE AMPLIFIER GUI



INTEL ADVISOR

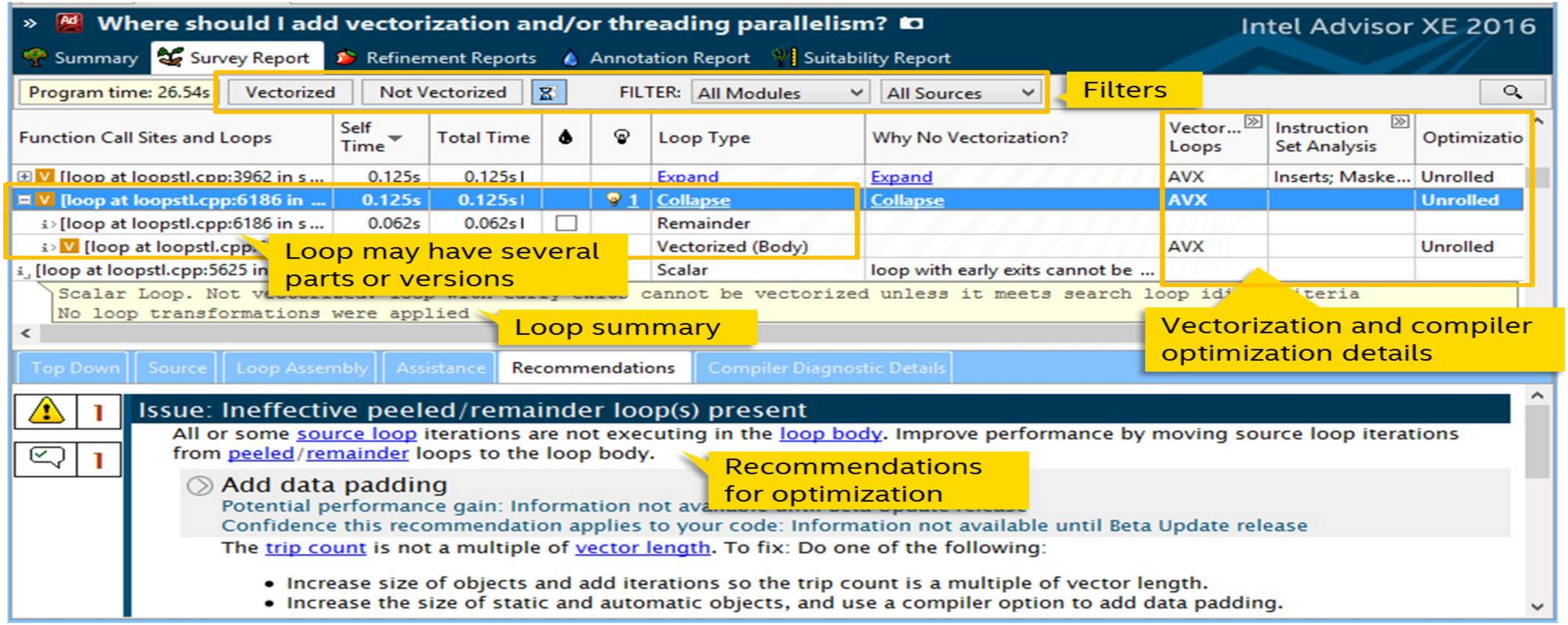
- Vectorization Advisor
  - Loops-based analysis to identify vectorization candidates
  - Finds save spots to enforce compiler vectorization
  - Roofline analysis to explore performance headroom and co-optimize memory and computation

- Threading Advisor
  - Identify issues before parallelization
  - Prototype performance impact of different threading designs
  - Find and eliminate data-sharing issues

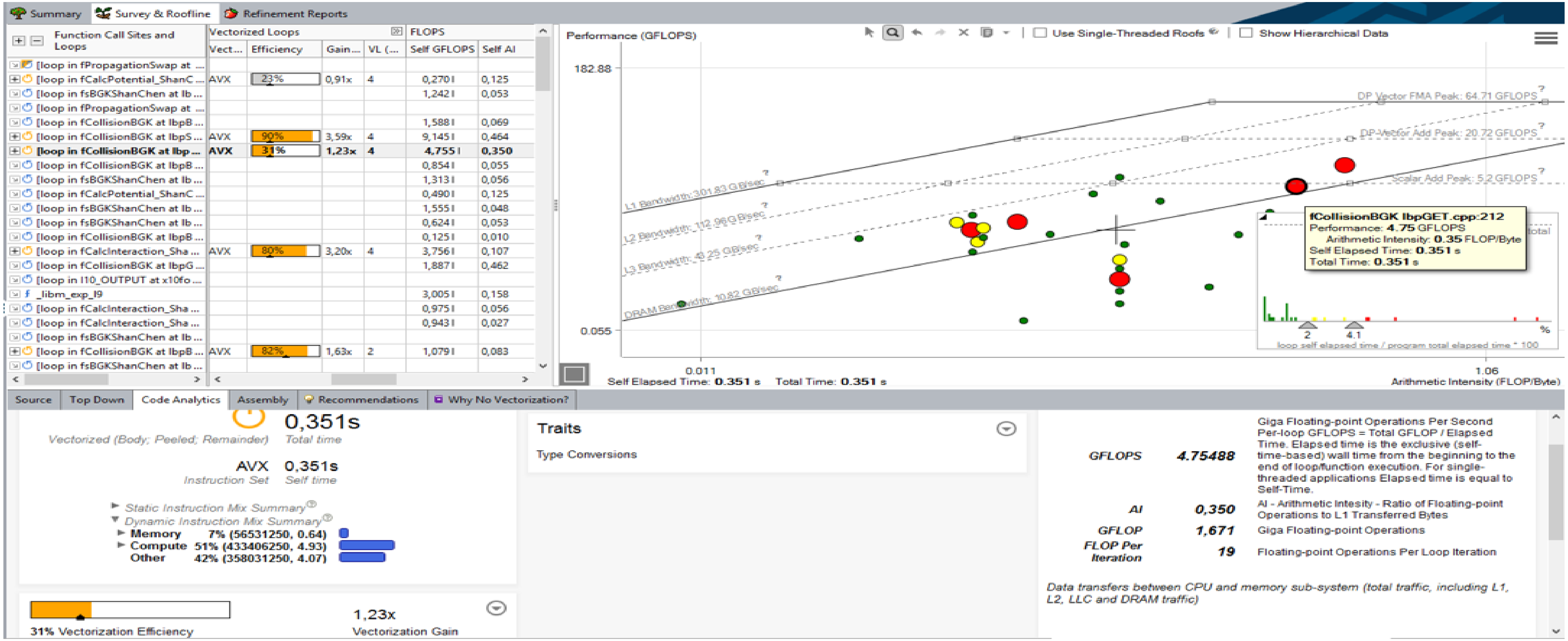
- Flow-Graph Analysis
  - Speed up algorithm design and express parallelism efficiently
  - Plan, validate, and model application design

- C/C++ and Fortran with OpenMP and Intel TBB

INTEL ADVISOR GUI



INTEL ADVISOR – ROOFLINE



PERFORMANCE ANALYSIS RECOMMENDATIONS

- Measure and analyze at the desired scale (once you have a reasonable measurement setup)

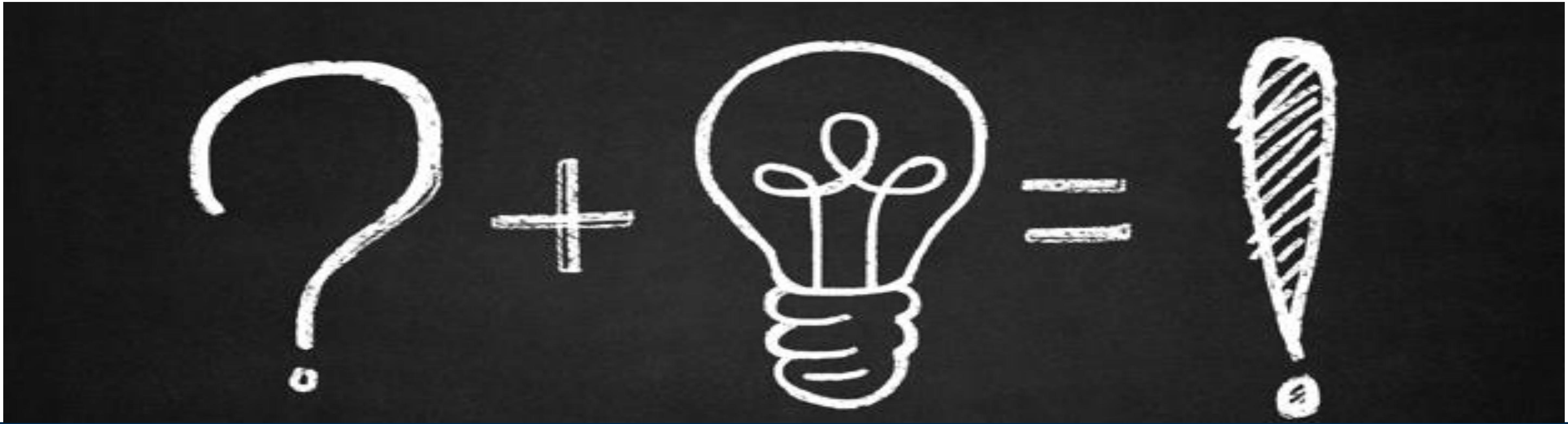
- Get performance overview with Performance Reports or HPC Snapshot
  - CPU Issues: Use Vtune (on Intel nodes) or uProf (on AMD nodes)
  - MPI Issues: Use Scalasca/Vampir
  - GPU Issues: Use NVIDIA tools
  - I/O Issues: Use DARSHAN

- OR: Do it all with Score-P/Scalasca/Vampir

NEED HELP?

- Talk to the experts
  - Use local 1st-level support, e.g. SimLab
  - Use mailing lists
  - JSC/NVIDIA Application Lab
  - ATML Parallel Performance
  - ATML Application Optimization and User Service Tools
  - Apply for a POP audit

☞Successful performance engineering often is a collaborative effort



QUESTIONS