



HPC SOFTWARE – DEBUGGER AND PERFORMANCE ANALYSIS TOOLS

NOVEMBER 13, 2024 | MICHAEL KNOBLOCH | M.KNOBLOCH@FZ-JUELICH.DE

OUTLINE

- Local module setup
- Compilers
- Libraries

Debugger and Correctness Tools

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

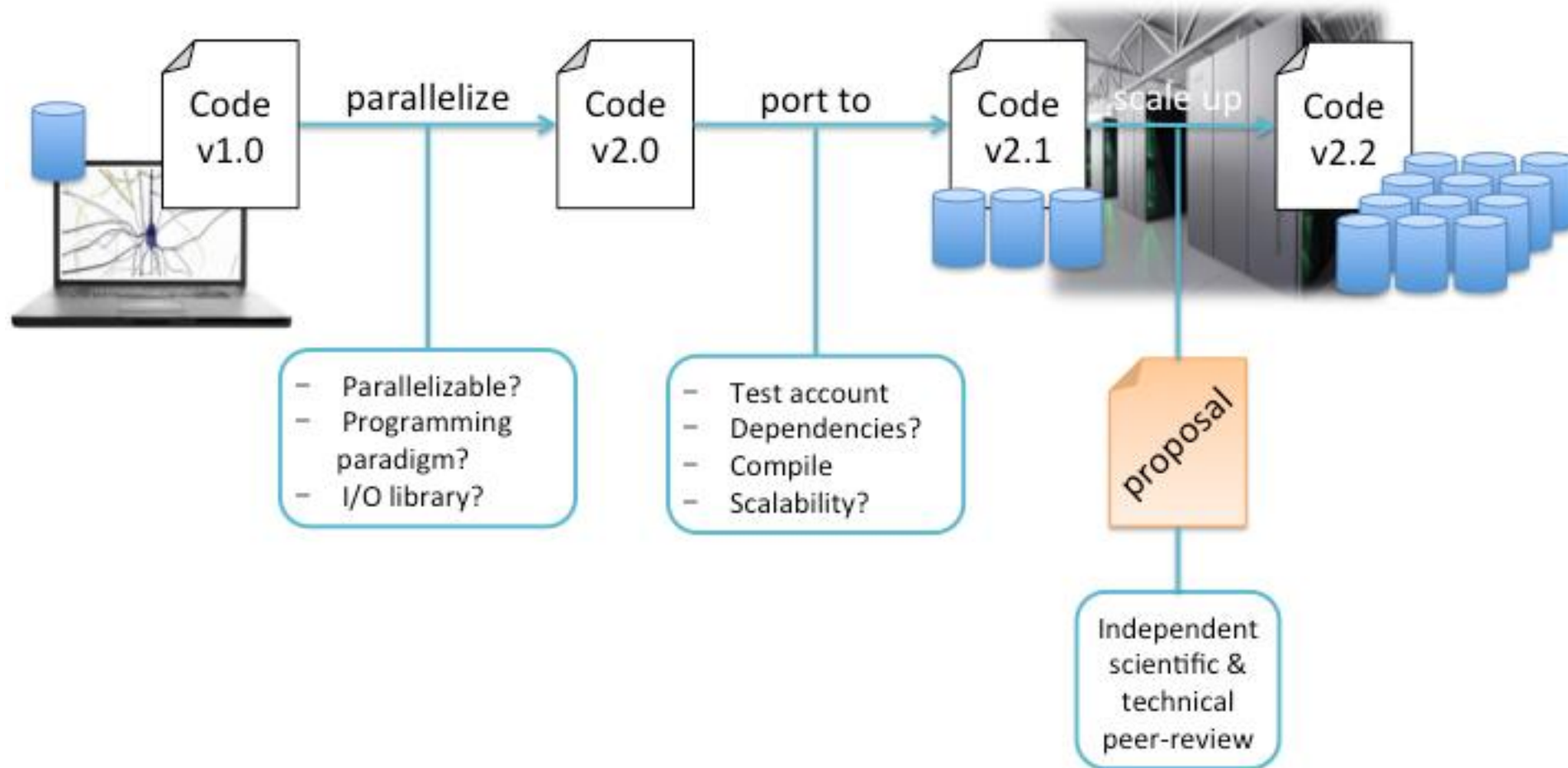
Kent Beck

Performance Analysis Tools

WHY SHOULD YOU CARE ABOUT TOOLS?



NEW APPLICATION?



WORKING WITH LEGACY CODES?



VETERAN HPC USER, BUT NEW TO JSC?



- Assess performance on a JSC machine



- Compare behavior on different machines



- Investigate scaling behavior



The screenshot displays a debugger interface with four main panels:

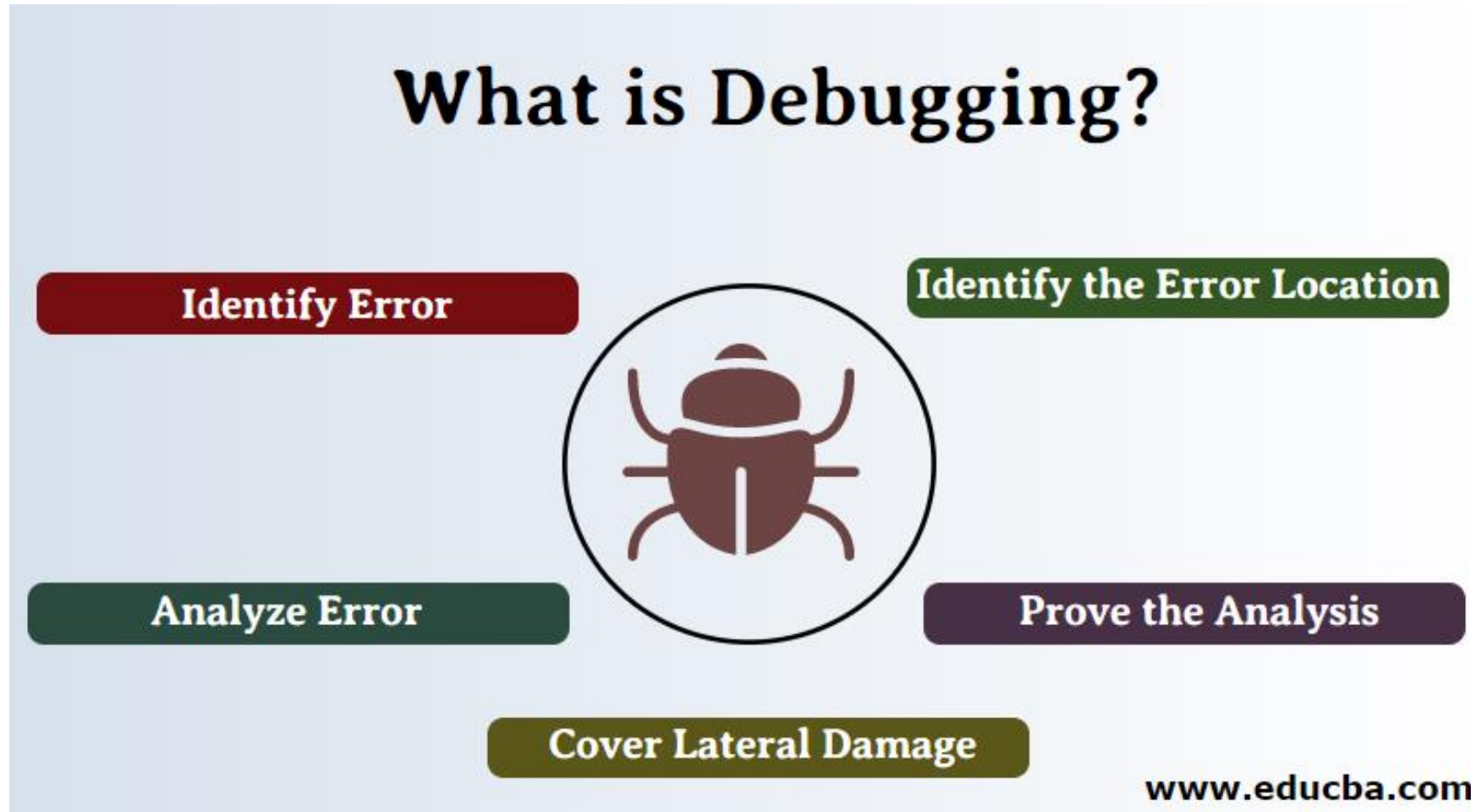
- Thread State Panel:** Shows a list of threads. The first thread is a **Breakpoint** with TID 1.1 and location `tensorflow::SoftmaxXentWith...`. Other threads are in a **Stopped** state with TIDs 1.2, 1.3, and 1.4, all at `pthread_cond_wait`.
- Variable Values Panel:** A table showing memory locations and their values:

Name	Type	Value
<code>_</code>	<code>int</code>	<code>0x0000000000000000...</code>
<code>nstar</code>	<code>int</code>	<code>0x000000006 (6)</code>
<code>grap...</code>	<code>int</code>	<code>0x000000015 (21)</code>
- Source Code Panel:** Displays C++ code from `tensorflow::TF_Run`. Line 619 is highlighted in yellow, showing the function signature:


```
619 TF_Run_Helper(s->session, nullptr, run_options, input_pairs, output_names, c_outputs, target_oper_names, run_metadata, status);
```
- Call Stack Panel:** Lists the sequence of function calls:
 - `tensorflow::FunctionLibraryRuntime...` (C++)
 - `tensorflow::DirectSession::GetOr...` (C++)
 - `std::_Function_handler<tensorflow::...` (C++)
 - `std::function<tensorflow::Status (...` (C++)
 - `tensorflow::Sunnamed_namespa...` (C++)
 - `tensorflow::NewLocalExecutor` (C++)
 - `tensorflow::DirectSession::GetOr...` (C++)
 - `tensorflow::DirectSession::Run` (C++)
 - `TF_Run_Helper` (C++)
 - `TF_Run`** (C++) - currently selected
 - `tensorflow::TF_Run_wrapper_hel...` (C++)
 - `tensorflow::TF_Run_wrapper` (C++)
 - `_run_fn` (Py)
 - `ext_do_call` (C)
 - `_do_call` (Py)
 - `_do_run` (Py)

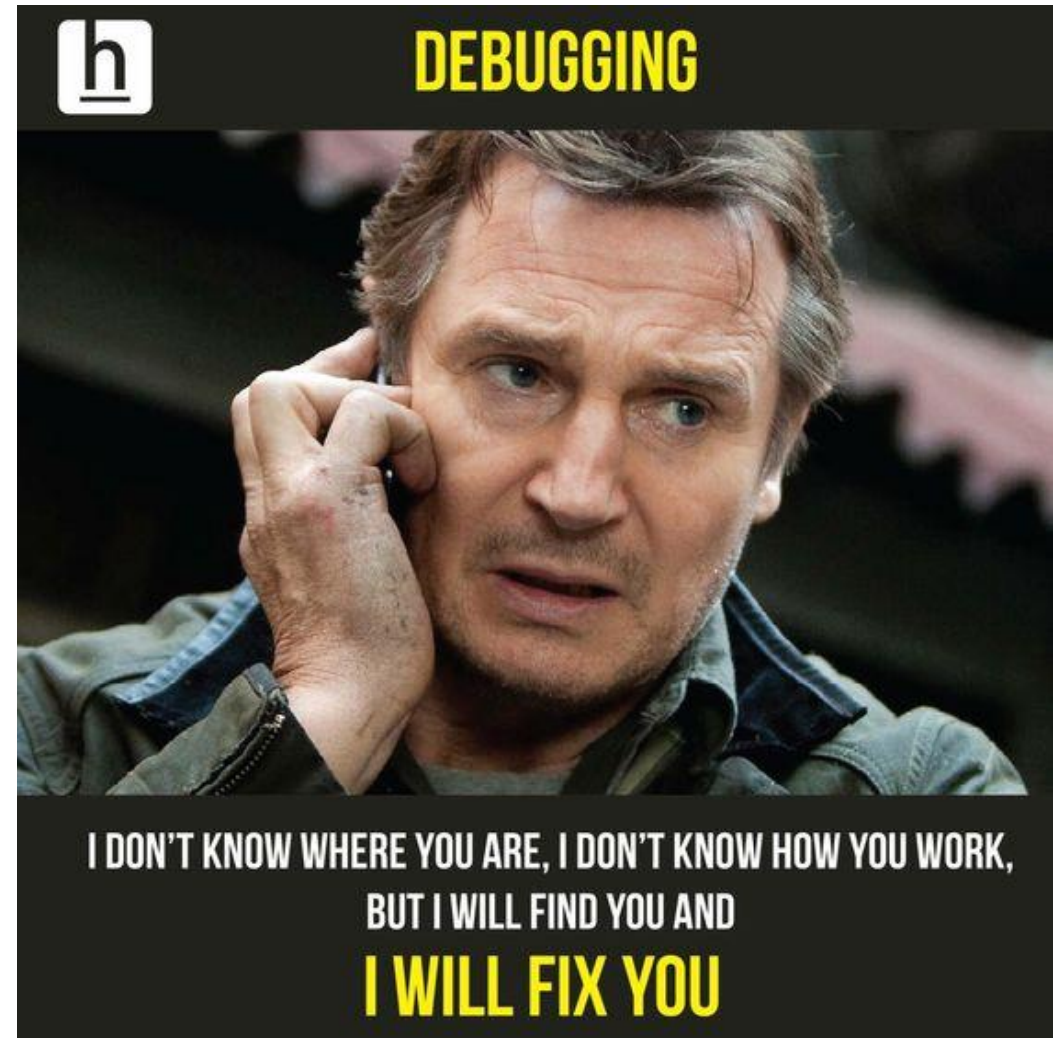
DEBUGGER & CORRECTNESS TOOLS

WHAT IS DEBUGGING?



DEBUGGING TOOLS (STATUS: NOV 2024)

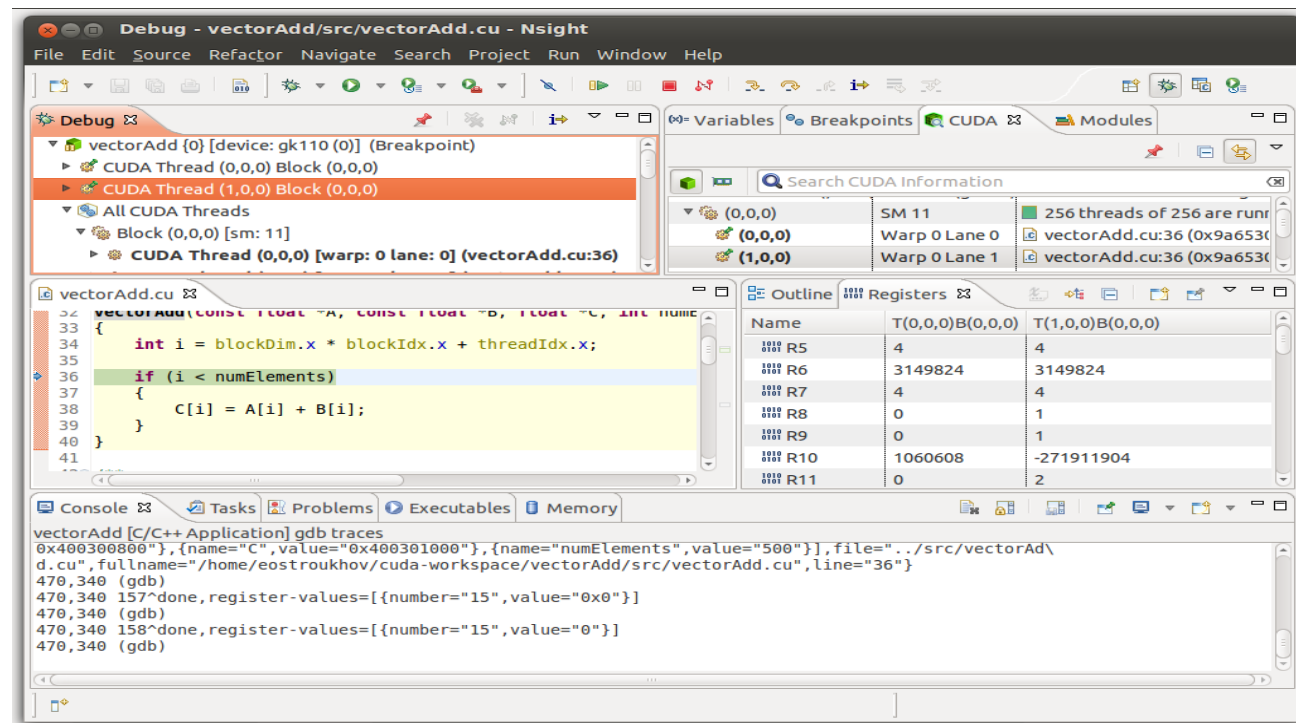
- **Debugger:**
 - CUDA-GDB
 - TotalView
 - LinaroForge - DDT
- **Memory Analyzer:**
 - Intel Inspector
 - Archer
- **Correctness Checker:**
 - MUST



CUDA-GDB



- Part of the CUDA toolkit
- 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
- Examine variables, read/write memory and registers
- Inspect GPU state when the application is suspended
- Identify memory access violations



- 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

The screenshot shows the CodeDynamics 2017 IDE interface. At the top is a menu bar (File, Edit, Group, Process, Thread, Action Points, Debug, Window, Help) and a toolbar with various icons. Below the toolbar is the 'Processes & Threads' panel on the left, showing a tree view of the current process and threads. The central pane is the 'Source code window' displaying C++ code for a CUDA kernel. The 'Call Stack' panel on the right shows the current stack frame. At the bottom, there is an 'Action Points' panel with a table of breakpoints, a 'Command Line' panel, and a 'Data View' panel showing the contents of a local variable 'A'.

Toolbar for common options

Stack trace

Local variables for selected stack frame

Source code window

Thread control

Break points

ID	Type	Stop	File	Line
1	BP	Process	tx_cuda_matmul	91

Name	Type	Value
A	Matrix @local	(Matrix @local)
width	int	0x00000002 (2)
height	int	0x00000002 (2)
stride	int	0x00000002 (2)
elements	float @generic *	0xb03ee0000 -...

LINARO FORGE - DDT

- UNIX Graphical Debugger for C/C++, Fortran, and Python programs
- Modern, easy-to-use debugger
- Advanced features
 - Multi-process and multi-threaded
 - Multi-dimensional 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://linaroforge.com/linaroDdt>
- **NOTE:** JSC license limited to 128 processes (shared between all users)

DDT: MAIN WINDOW

Process controls

CUDA Thread stepping

Variables

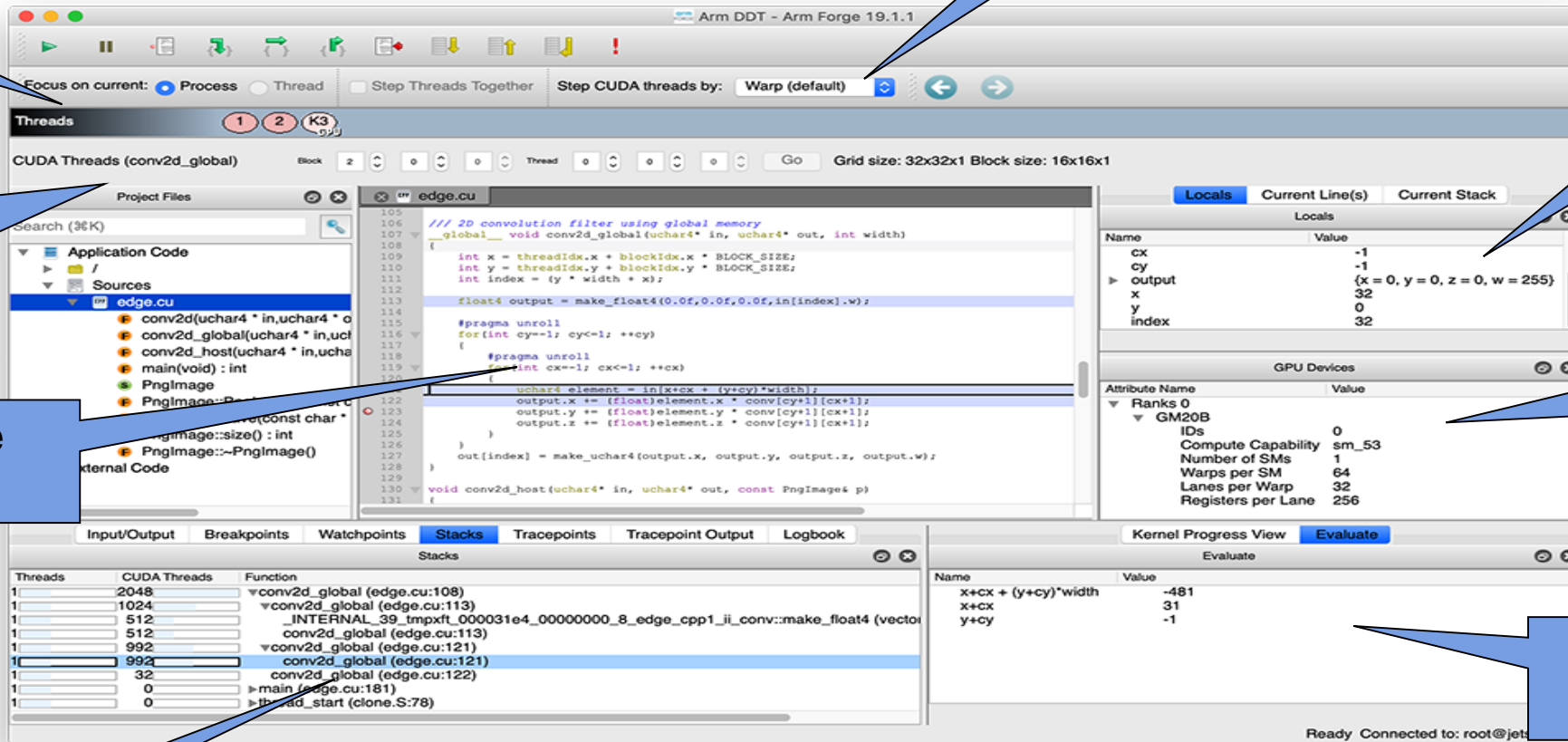
CUDA Thread control

GPU Device information

Source code

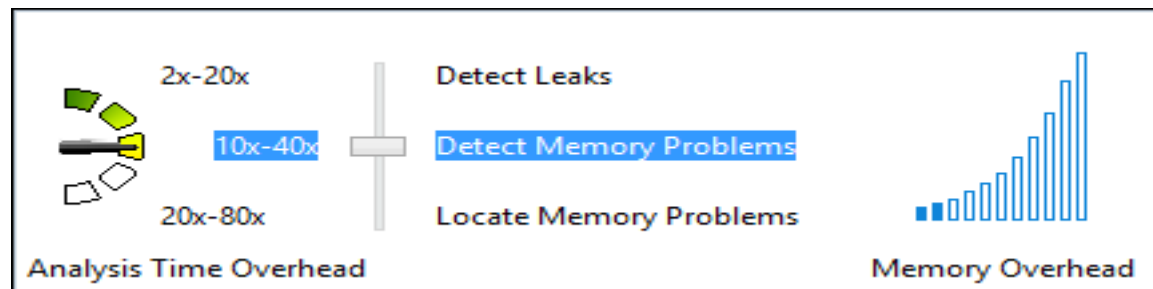
Expression evaluator

Stack trace



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

The screenshot shows the Intel Inspector interface for 'Detect Deadlocks and Data Races'. The main window displays a table of detected problems:

ID	Type	Sources	Modules	State
P1	Data race	find_and_fix_threading_errors.cpp; task_scheduler_init.h	find_and_fix_threading_errors.exe	New
P2	Data race	blocked_range.h; parallel_for.h; partitioner.h; task.h	find_and_fix_threading_errors.exe	New
P3	Data race	winvideo.h	find_and_fix_threading_errors.exe	New

On the right, a 'Filters' pane shows the severity and type of the detected data races, with 3 items listed for each.

This screenshot shows the 'Code Locations: Data race' view. It displays the source code for the function `render_one_pixel` in `find_and_fix_threading_errors.cpp`. The code includes comments indicating a threading error and a hint to use a local variable for synchronization.

```
103 primary.scene = ascene;
104
105 col=trace(secondary); //Threading Error: col is a global variable
106 //2 ways to fix this threading error
107 // 1) Make col a local variable
```

A timeline at the bottom shows a thread named `thread_video (2108)`.

The screenshot shows the Intel Inspector interface for 'Detect Memory Problems'. The main window displays a table of detected memory issues:

ID	Type	Sources	State
P1	Mismatched allocation/deallocation	find_and_fix_memory_errors. ...	New
P2	Invalid memory access	find_and_fix_memory_errors. ...	New
P3	Memory growth	[Unknown]; find_and_fix_me ...	Not fixed
P4	Memory growth	[Unknown]; find_and_fix_me ...	Confirmed

The 'Code Locations: Invalid memory access' view shows the source code for the `operator()` function in `find_and_fix_memory_error...`. The code includes a loop that accesses `local_mbox[i]`, with a comment indicating a memory error.

```
164
165 for (unsigned int i=0; i<=(mboxsize-1); i++)
166     local_mbox[i]=0; //Memory Error
167
168 for (int y = r.begin(); y != r.end(); y++)
```


ARCHER



- Data race detector for large OpenMP programs
- Combination of static and dynamic techniques
 - Low runtime and memory overhead
 - Still high accuracy and precision
- Now part of LLVM
- Compile with `-fsanitize=thread`
- Can be used with GCC, but CLANG OpenMP runtime must be linked
- Creates output in text format

ARCHER EXAMPLE

```
WARNING: ThreadSanitizer: data race (pid=2234)
Write of size 4 at 0x7fff81d209d0 by thread T1:
#0 .omp_outlined._debug__ /p/project/training2410/knobloch1/archer_test.c:11:10 (a.out+0xd1efb)
#1 .omp_outlined. /p/project/training2410/knobloch1/archer_test.c:9:1 (a.out+0xd1f85)
#2 __kmp_invoke_microtask <null> (libomp.so+0xb8782)
#3 main /p/project/training2410/knobloch1/archer_test.c:9:1 (a.out+0xd1d55)

Previous read of size 4 at 0x7fff81d209d0 by main thread:
#0 .omp_outlined._debug__ /p/project/training2410/knobloch1/archer_test.c:11:12 (a.out+0xd1ed6)
#1 .omp_outlined. /p/project/training2410/knobloch1/archer_test.c:9:1 (a.out+0xd1f85)
#2 __kmp_invoke_microtask <null> (libomp.so+0xb8782)
#3 main /p/project/training2410/knobloch1/archer_test.c:9:1 (a.out+0xd1d55)

Location is stack of main thread.

Location is global '??' at 0x7fff81d03000 ([stack]+0x1d9d0)

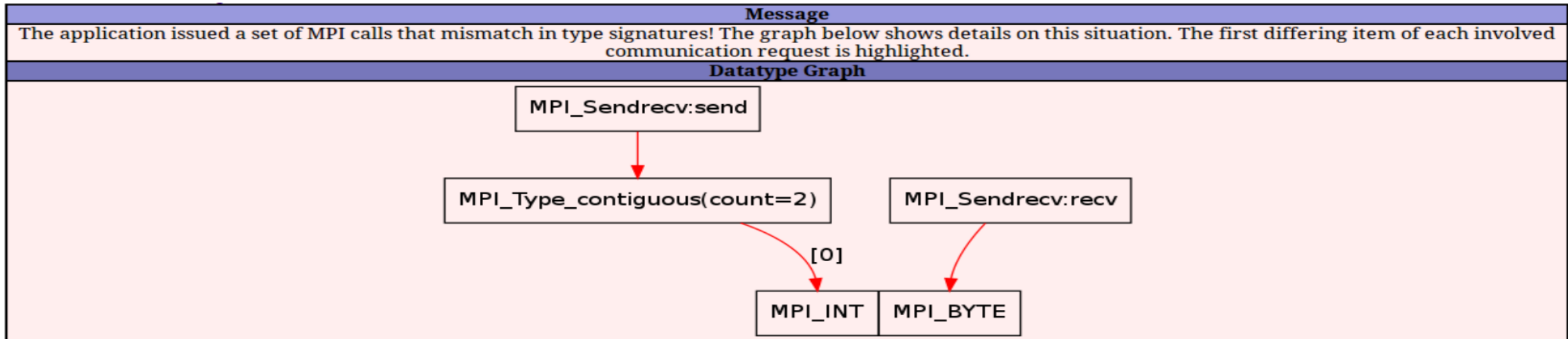
Thread T1 (tid=2237, running) created by main thread at:
#0 pthread_create /dev/shm/swmanage/jusuf/Clang/16.0.6/GCCcore-12.3.0/llvm-project-16.0.6.src/compiler-rt/lib/tsan/rtl/tsan_interceptors_posix.cpp:1048:3 (a.out+0x2678b)
#1 __kmp_create_worker <null> (libomp.so+0x97676)

SUMMARY: ThreadSanitizer: data race /p/project/training2410/knobloch1/archer_test.c:11:10 in .omp_outlined._debug__
=====
ThreadSanitizer: reported 1 warnings
```

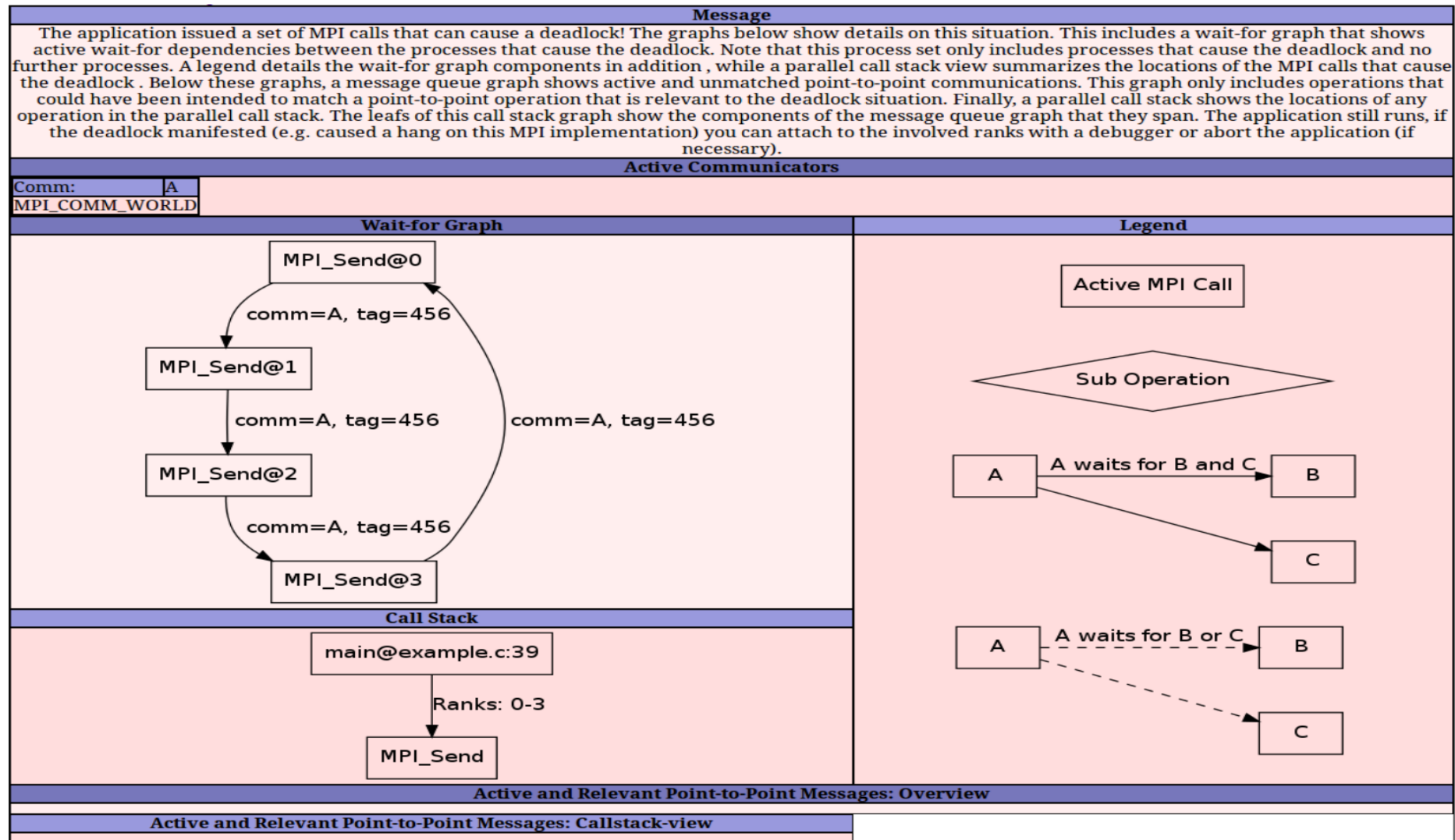
- Next generation MPI correctness and portability checker
- <https://www.i12.rwth-aachen.de/go/id/nrbe>
- MUST reports
 - Errors: violations of the MPI-standard
 - Warnings: unusual behavior or possible problems
 - Notes: harmless but remarkable behavior
 - Potential deadlock detection
- Usage
 - Compile with debug information (i.e. use the -g flag)
 - Run application under the control of `mustrun` (requires (at least) one additional MPI process)
 - E.g. on JUSUF: `mustrun --must:mpiexec srun --must:np -n -n 4 ./app`
 - Open output html report (might need to copy it to your local machine)

MUST DATATYPE MISMATCH

Rank	Type	Message	From	References
0	Error	<p>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, committed 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)</p>	<p>MPI_Sendrecv called from: #0 main@example.c:33</p>	<p>reference 1 rank 0: MPI_Sendrecv called from: #0 main@example.c:33</p> <p>reference 2 rank 1: MPI_Sendrecv called from: #0 main@example.c:33</p> <p>reference 3 rank 0: MPI_Type_contiguous called from: #0 main@example.c:29</p> <p>reference 4 rank 0: MPI_Type_commit called from: #0 main@example.c:30</p>



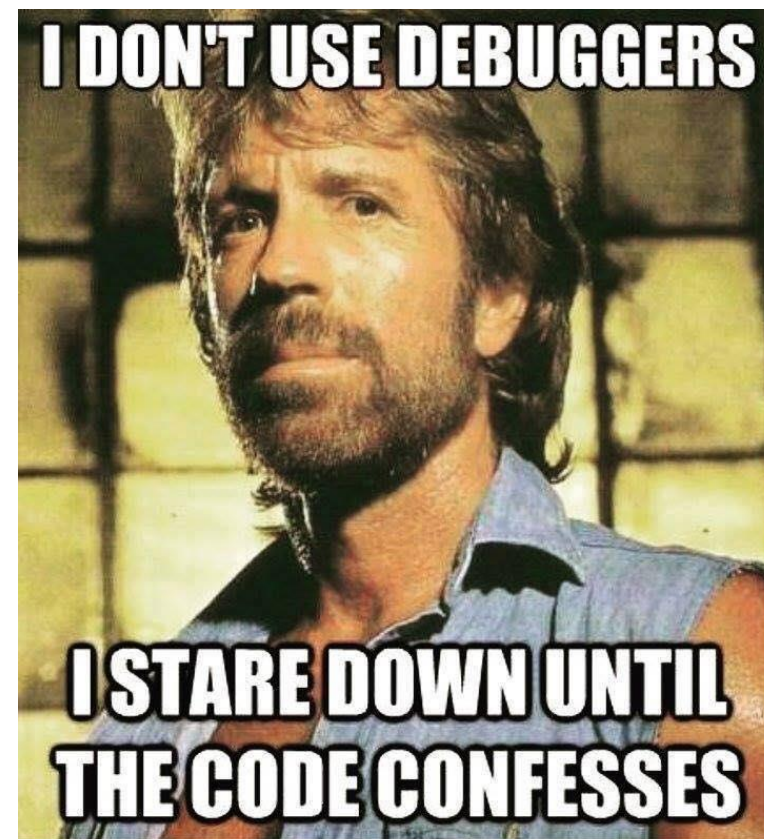
MUST DEADLOCK DETECTION



DEBUGGING RECOMMENDATIONS

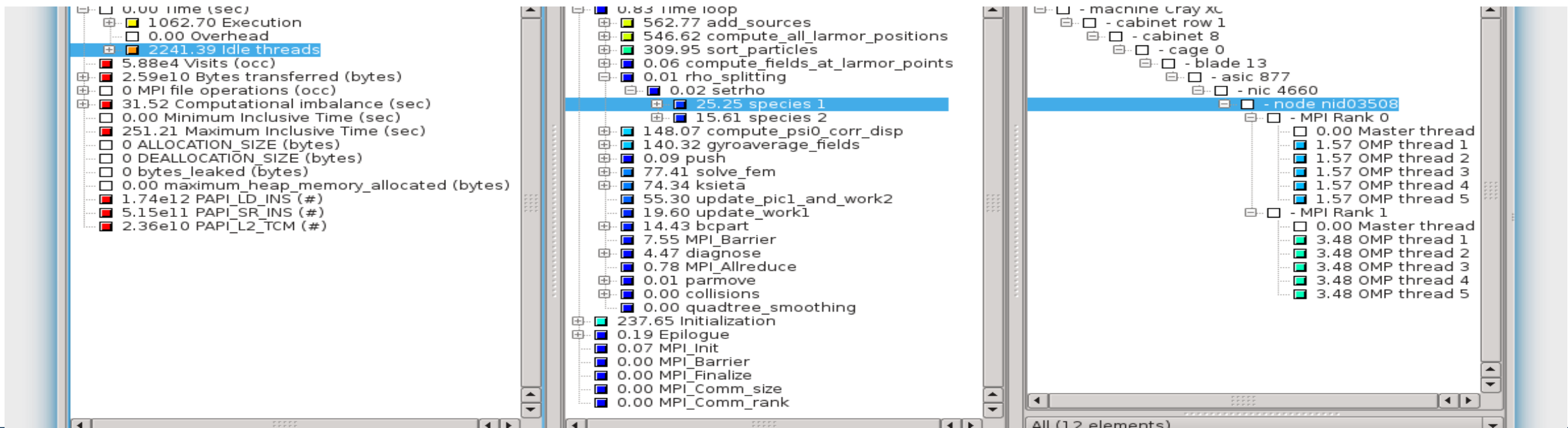
- Always debug at the lowest possible scale!
- GPU Applications:
 - Single Node / Workstation: Use CUDA-GDB
 - Multi-Node / Supercomputer: Use TotalView/DDT
- MPI Applications:
 - Check with MUST at least once
 - Use TotalView/DDT at small scale (if error occurs there), else attach to as few processes as necessary

DON'T BE THESE GUYS



REMINDER: DEBUGGING CAN BE FRUSTRATING





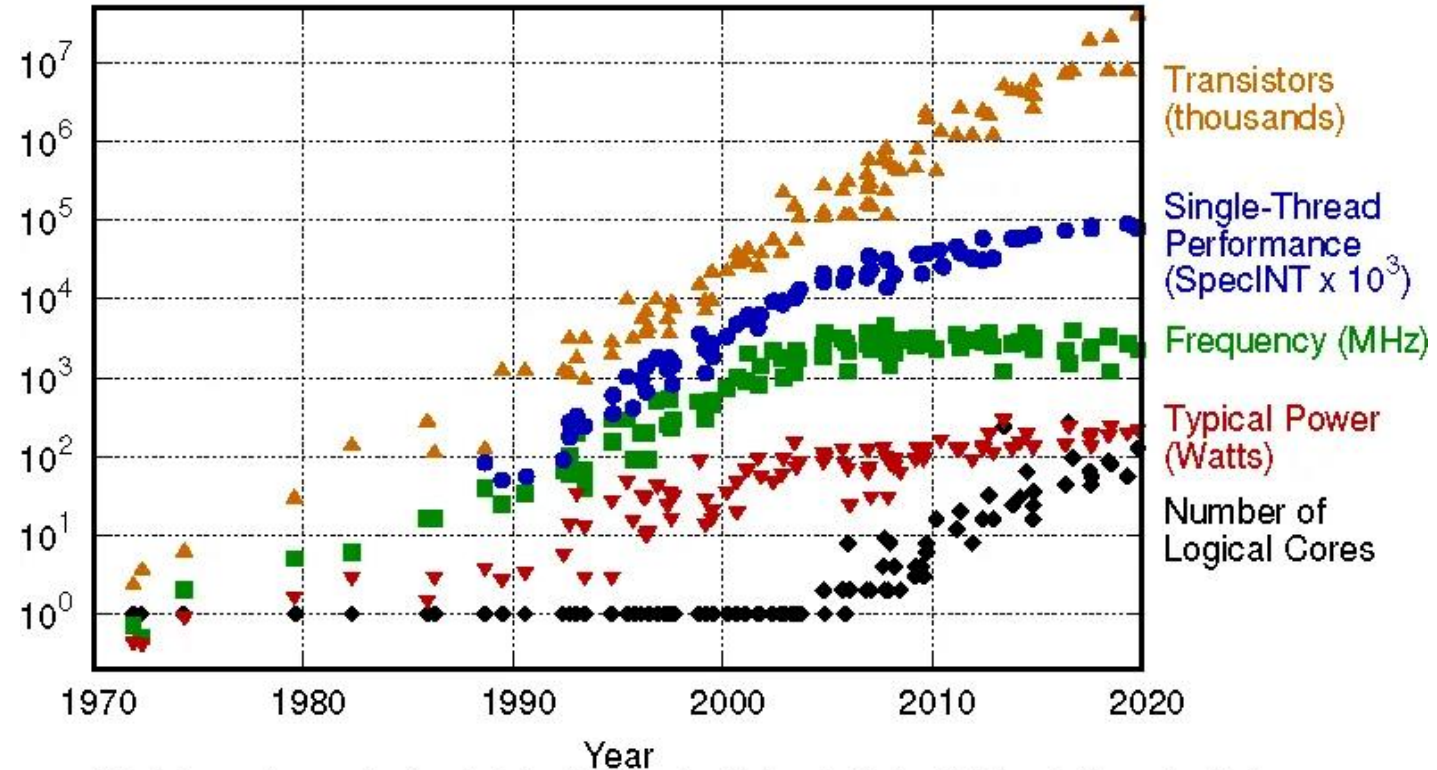
PERFORMANCE ANALYSIS TOOLS

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 Supercomputing Architecture

☞ Every doubling of scale reveals a new bottleneck!

48 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

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

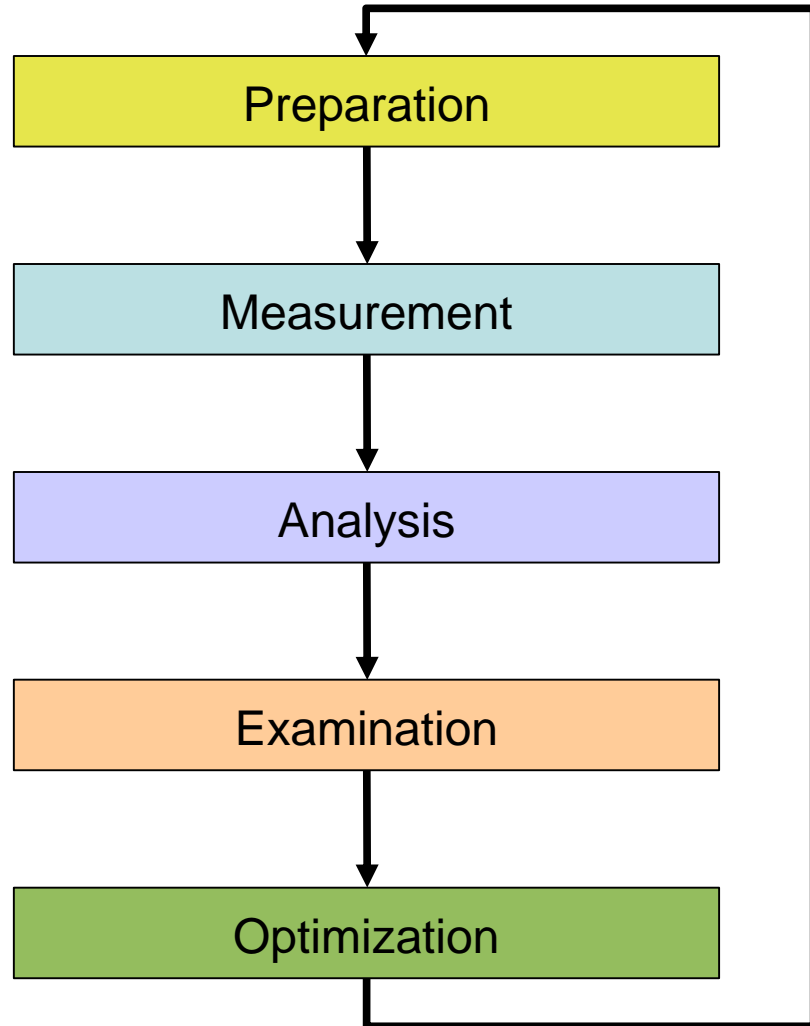
PERFORMANCE FACTORS

- “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

- Carefully set various tuning parameters
 - The right (parallel) algorithms and libraries
 - Compiler flags and directives
 - Correct machine usage (mapping and bindings)
 - 👉 Get the most performance before tuning!
- 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

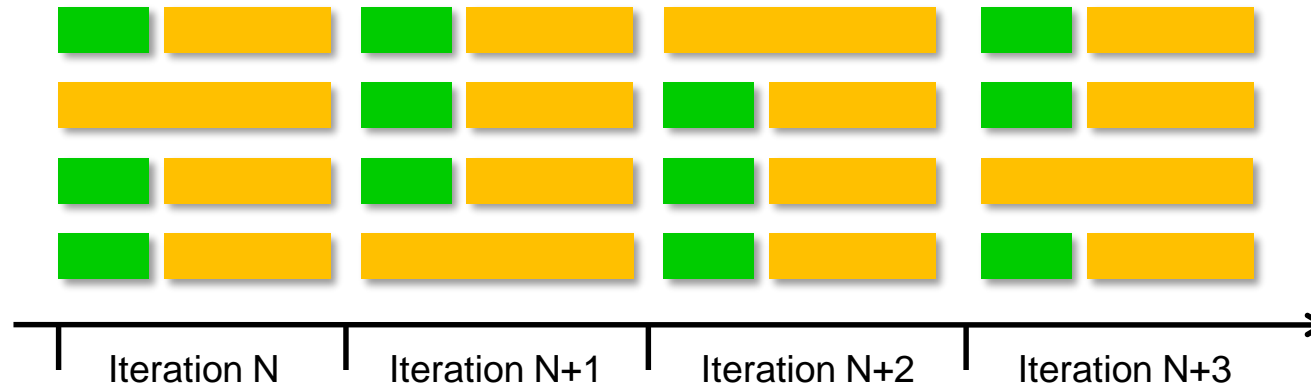
- Recording of **aggregated information**
 - Time
 - Counts
 - Calls
 - Hardware counters
- **Across program and system entities**
 - Functions, call sites, loops, basic blocks, ...
 - Processes, threads
- **Statistical information**
 - Min, max, mean and total number of values

Advantages
+ Works also for
long-running programs

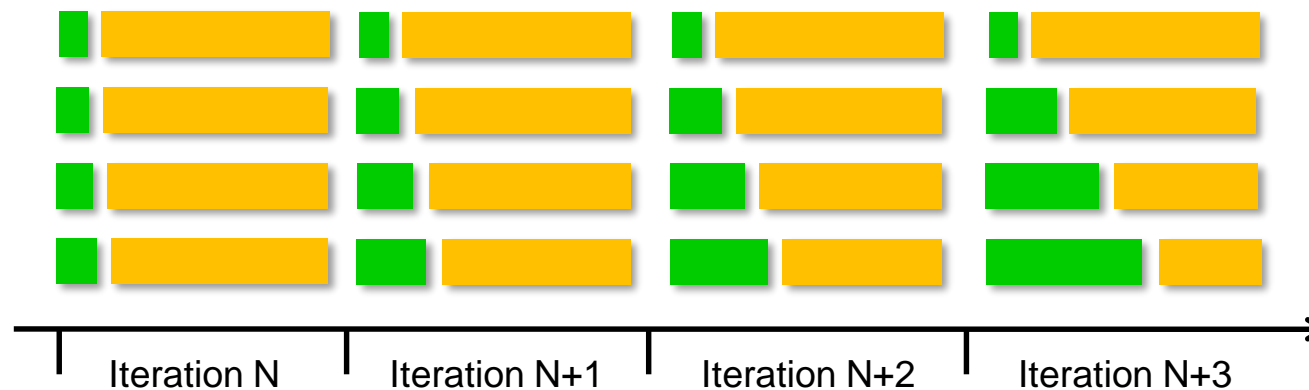
Disadvantages
– Variations over time
get lost

PROFILING: ISSUES RELATED TO "AVERAGING"

- Moving bottleneck across processors can "average out" imbalances



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



MEASUREMENT METHODS: TRACING

- Recording **information about** significant points (**events**) during execution of the program
 - Enter/leave a code region (function, loop, ...)
 - Send/receive a message ...
- Save information in **event record**
 - Timestamp, location ID, event type
 - plus event specific information
- **Event trace** := stream of event records sorted by time

⇒ 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

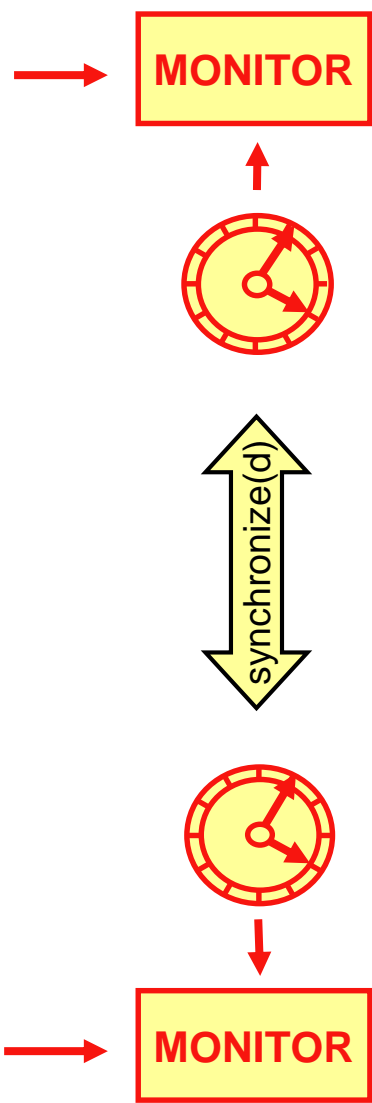
Process A

```
void foo() {
  trc_enter("foo");
  ...
  trc_send(B);
  send(B, tag, buf);
  ...
  trc_exit("foo");
}
```

instrument

Process B

```
void bar() {
  trc_enter("bar");
  ...
  recv(A, tag, buf);
  trc_recv(A);
  ...
  trc_exit("bar");
}
```



Local trace A

...		
58	ENTER	1
62	SEND	B
64	EXIT	1
...		
1	foo	
...		

Local trace B

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

Global trace

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

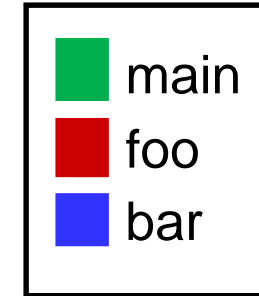
merge

unify

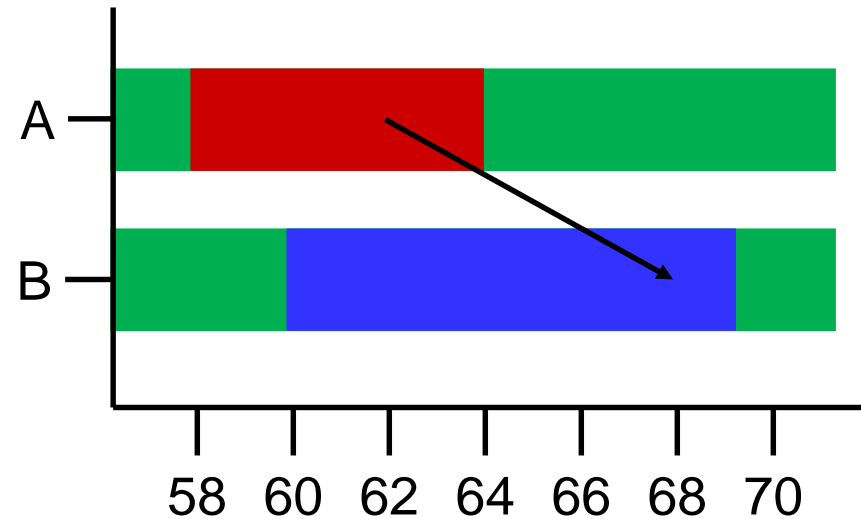
1	foo
2	bar
...	

EVENT TRACING: "TIMELINE" VISUALIZATION

1	foo
2	bar
3	...



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

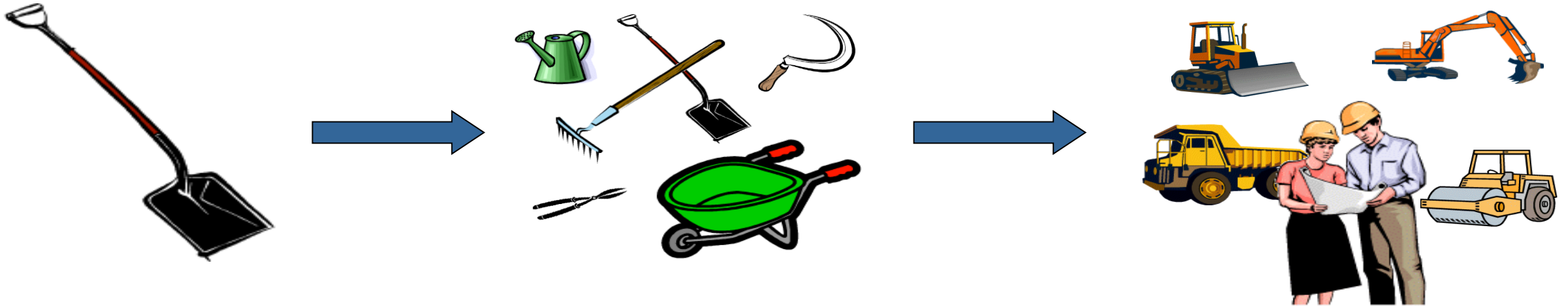


CRITICAL ISSUES

- 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*

REMARK: NO SINGLE SOLUTION IS SUFFICIENT!



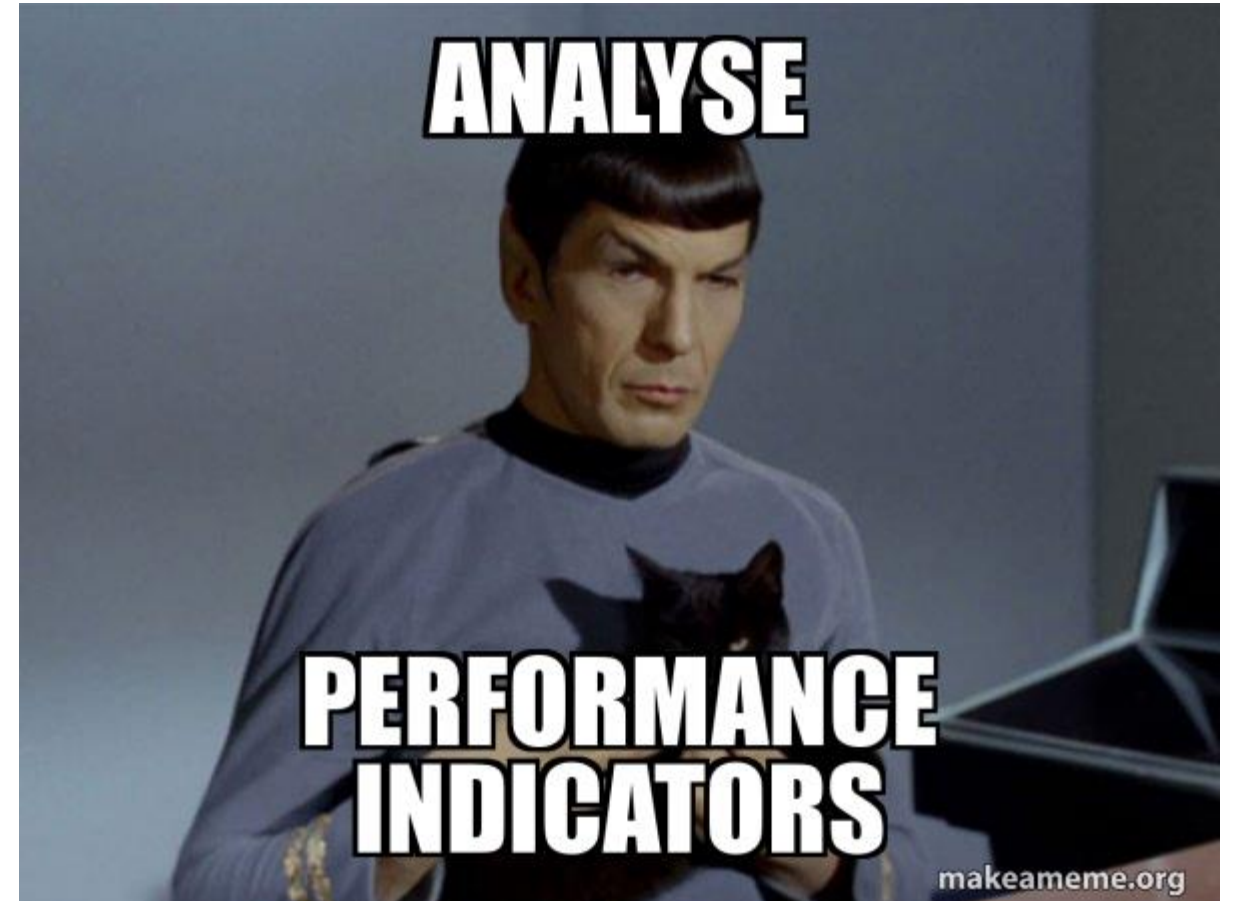
☞ *A combination of different methods, tools and techniques is typically needed!*

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

PERFORMANCE TOOLS (**STATUS: NOV 2024**)

- Score-P
- Scalasca
- Vampir[Server]
- Linaro Forge
 - Performance Reports
 - MAP
- Intel Tools
 - VTune Amplifier XE
 - Intel Advisor
- AMD uProf
- NVIDIA Tools
 - Nsight Systems
 - Nsight Compute
- Darshan
- ...

Mitglied der Helmholtz-Gemeinschaft





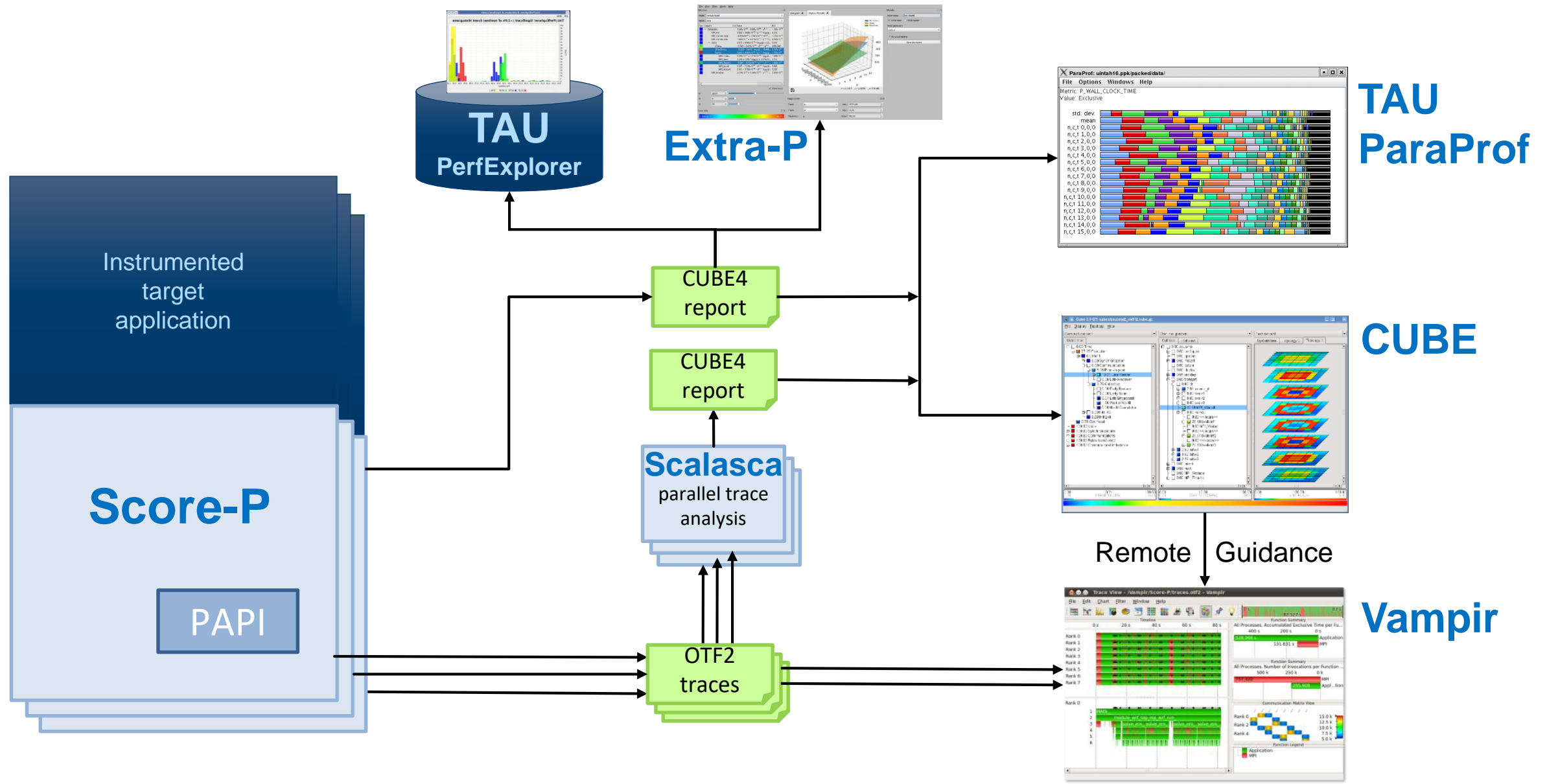
Score-P

Scalable performance measurement
infrastructure for parallel codes

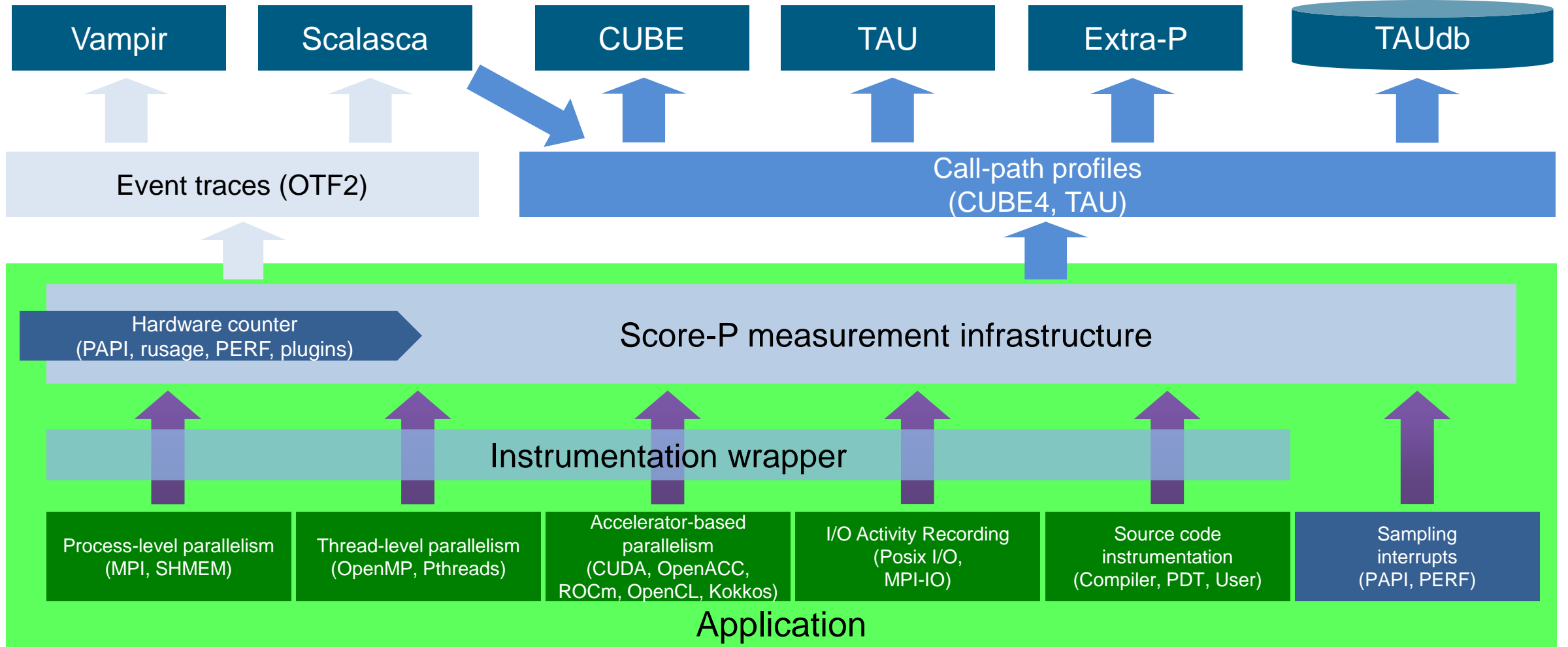
- Community-developed open-source
- Replaced tool-specific instrumentation and measurement components of partners
- <http://www.score-p.org>



Score-P TOOL ECOSYSTEM

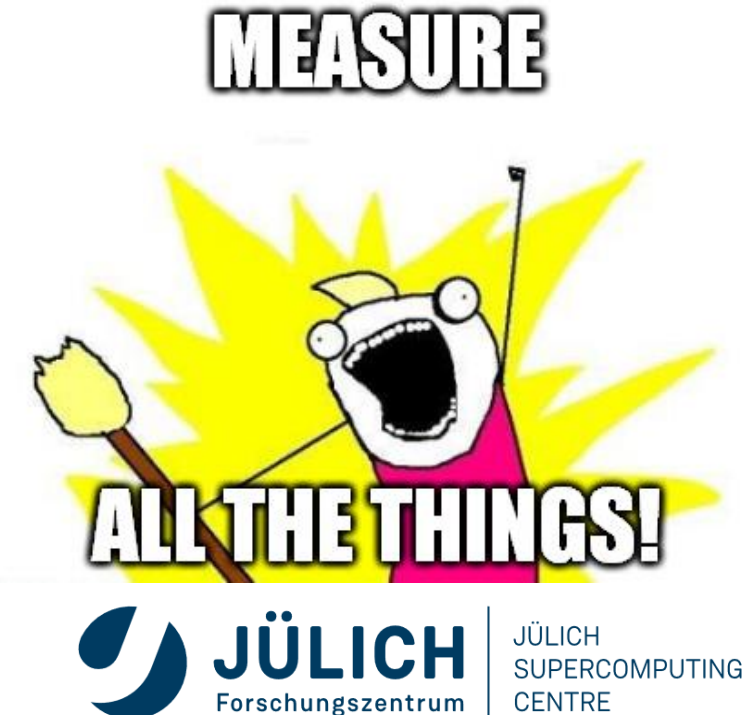


Score-P ARCHITECTURE

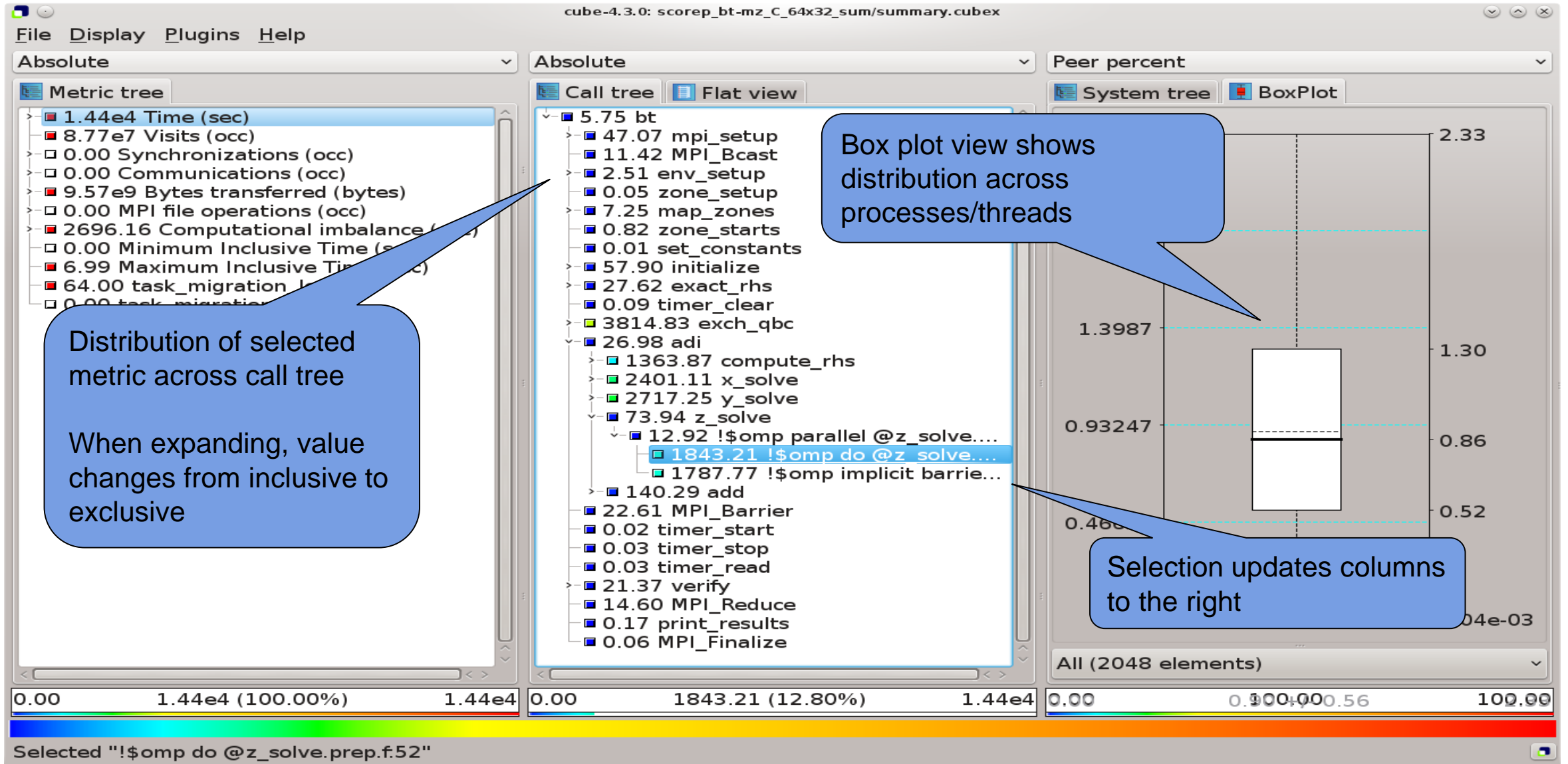


#Score-P FUNCTIONALITY

- 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



CUBE EXAMPLE



SCORE-P: ADVANCED FEATURES

- Measurement can be extensively configured via environment variables
- Allows for targeted measurements:
 - Selective recording
 - Phase profiling
 - Parameter-based profiling
 - ...
- GPU support: CUDA, OpenACC, OpenCL, HIP, Kokkos, ...
- Please ask us or see the user manual for details



SCALASCA

- 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

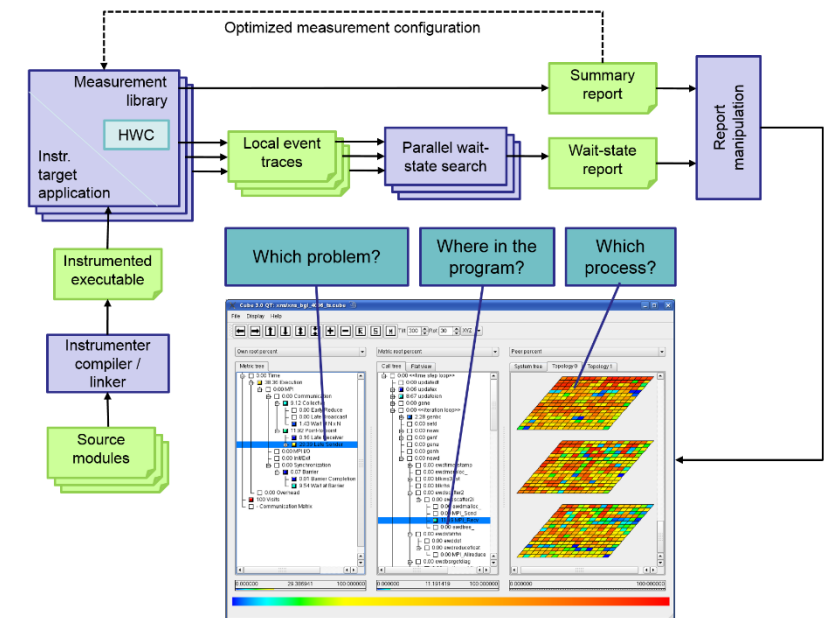
- Process trace in parallel

- Wait-state analysis

- Delay and root-cause analysis

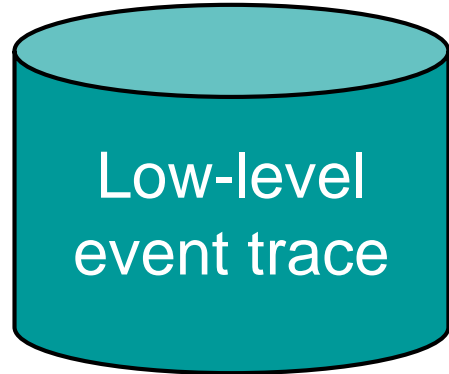
- Critical path analysis

- Categorize and rank results

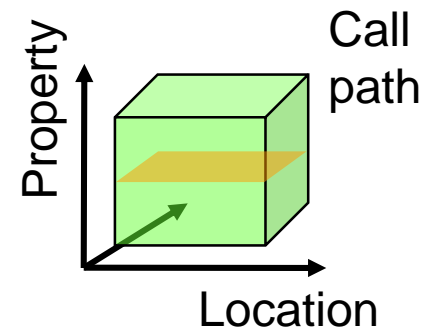


AUTOMATIC TRACE ANALYSIS

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

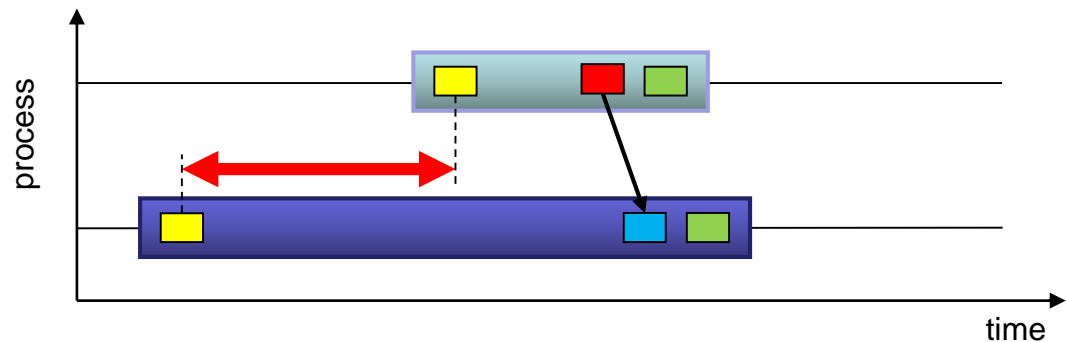


≡

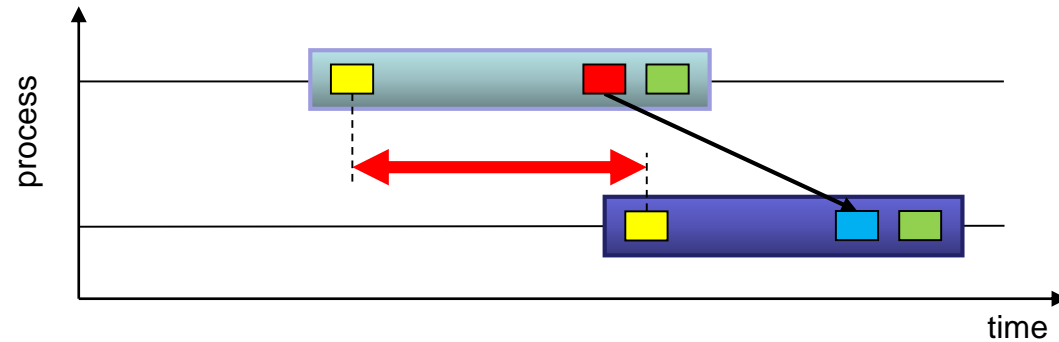


- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability

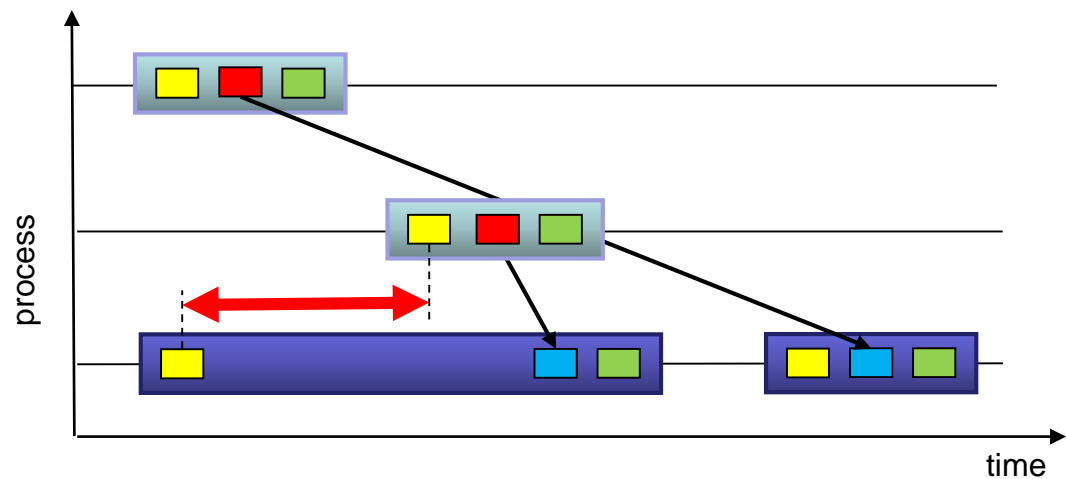
EXAMPLE MPI WAIT STATES



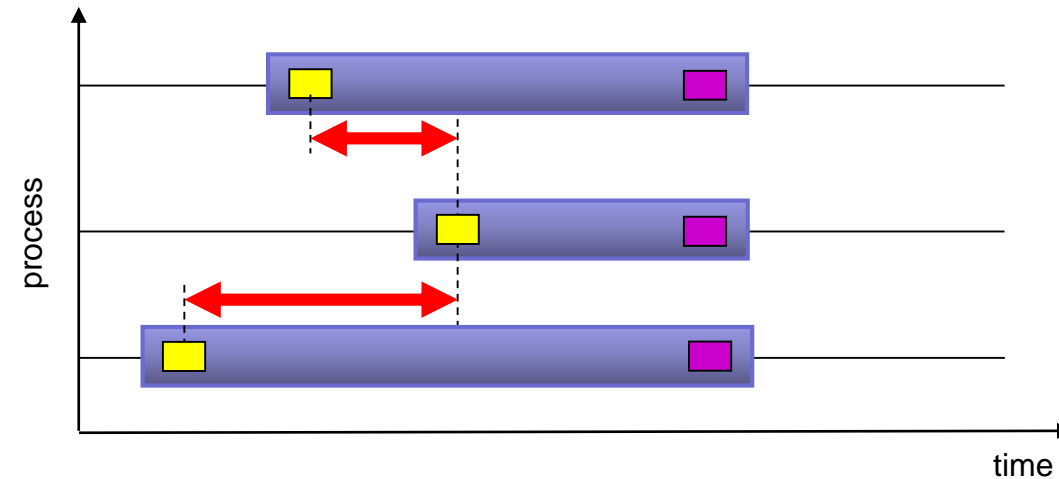
(a) Late Sender



(b) Late Receiver



(c) Late Sender / Wrong Order



(d) Wait at N x N



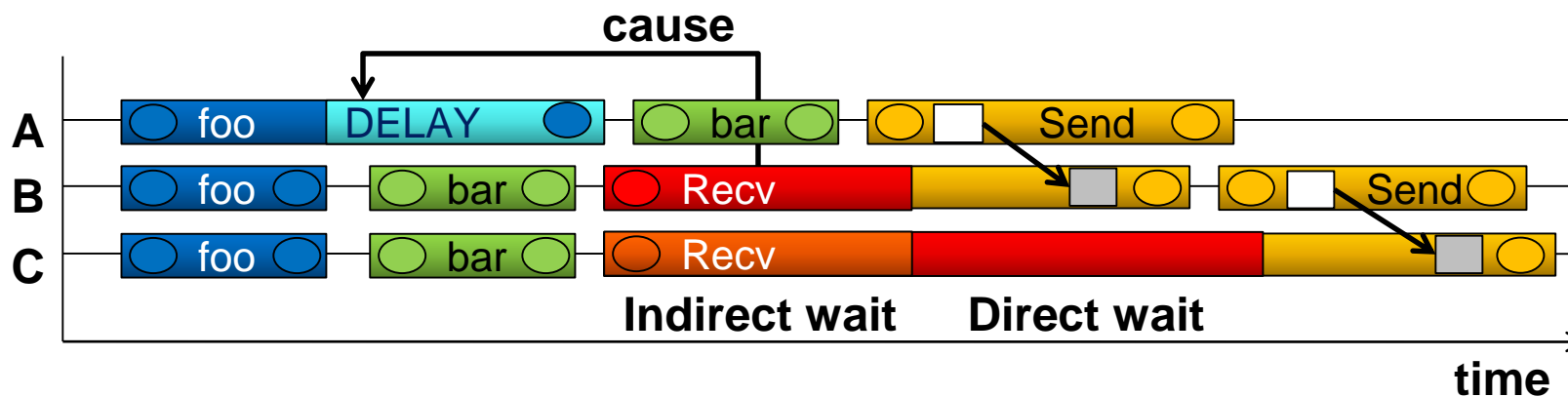
SCALASCA ROOT CAUSE ANALYSIS

- **Root-cause analysis**

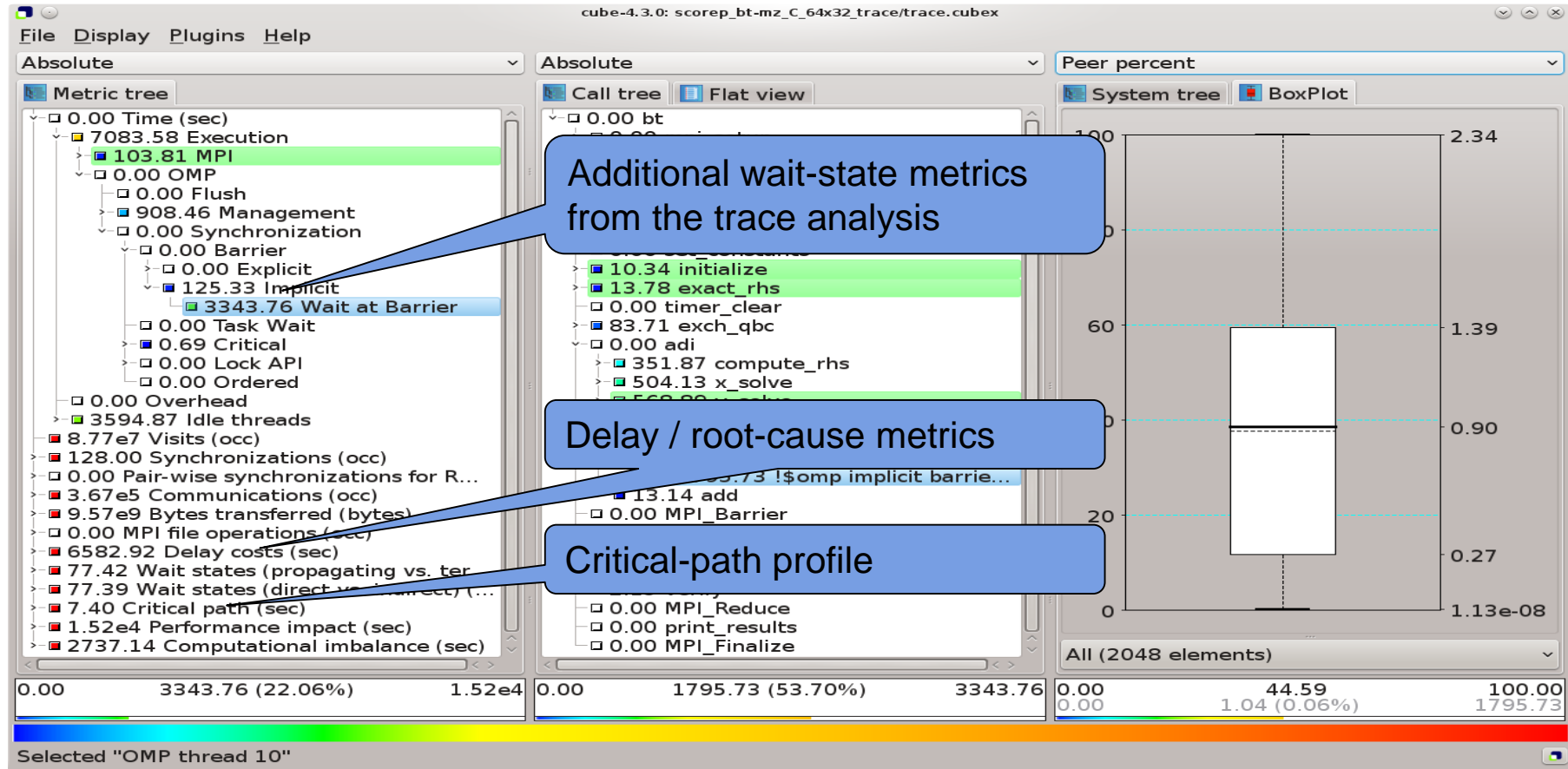
- Wait states typically caused by load or communication imbalances earlier in the program
- Waiting time can also propagate (e.g., indirect waiting time)
- Enhanced performance analysis to find the root cause of wait states

- **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**

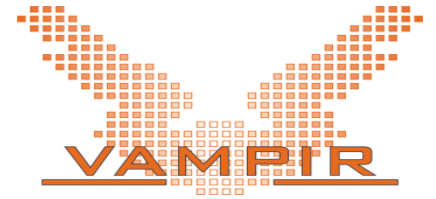


SCALASCA TRACE ANALYSIS EXAMPLE



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

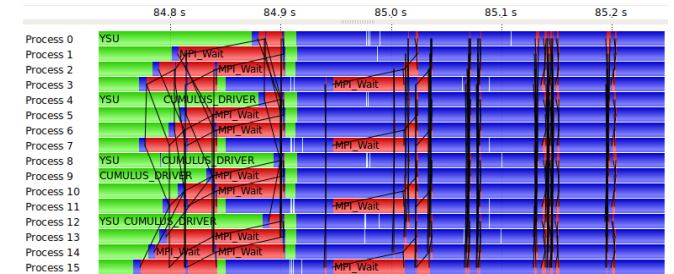


EVENT TRACE VISUALIZATION WITH VAMPIR

- 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?

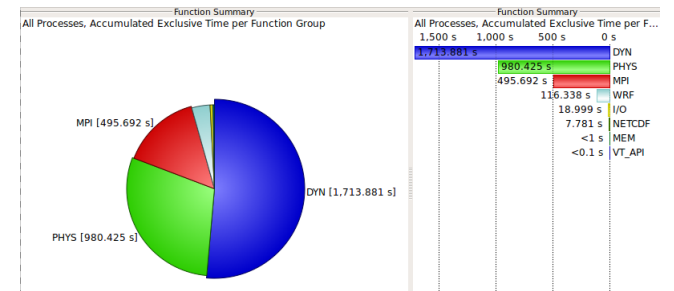
■ Timeline charts

- Application activities and communication along a time axis



■ Summary charts

- Quantitative results for the currently selected time interval



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

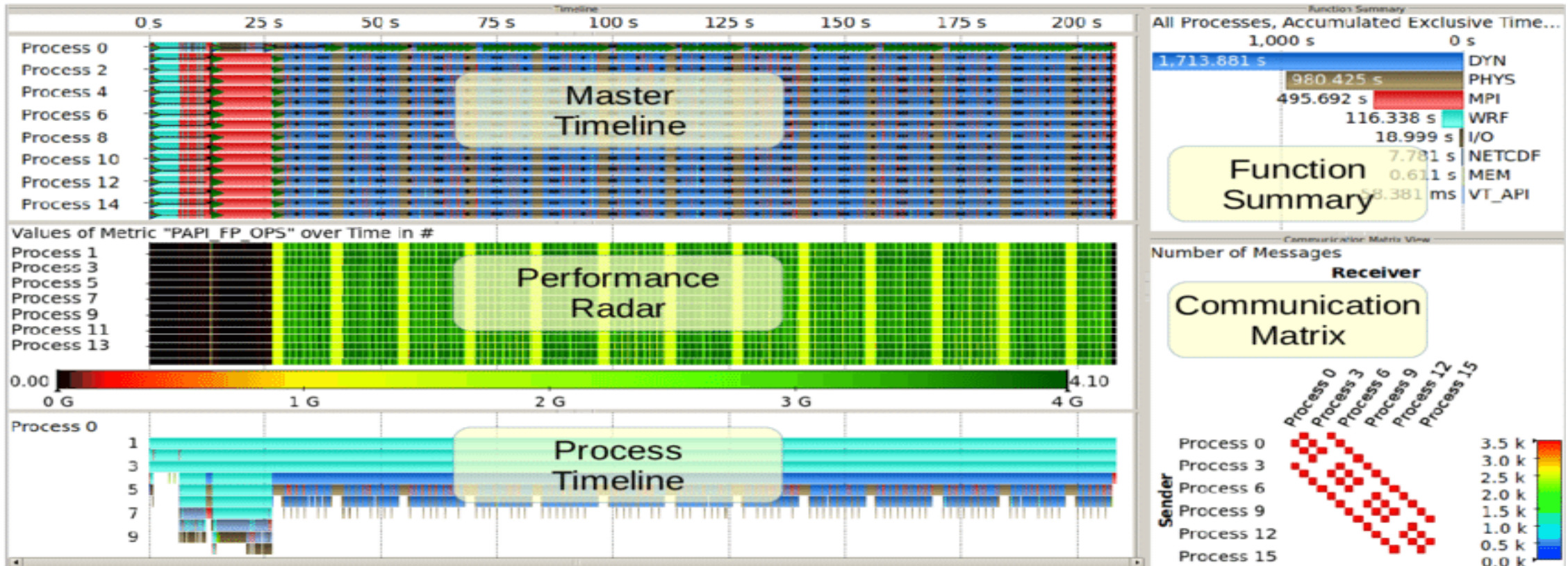


I/O Summary



Call Tree

VAMPIR DISPLAYS



LINARO 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.linaroforge.com/linaroPerformanceReports>
- **Note:** License limited to 128 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.

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!

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.

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



Nsight Systems

Standalone GUI+CLI

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



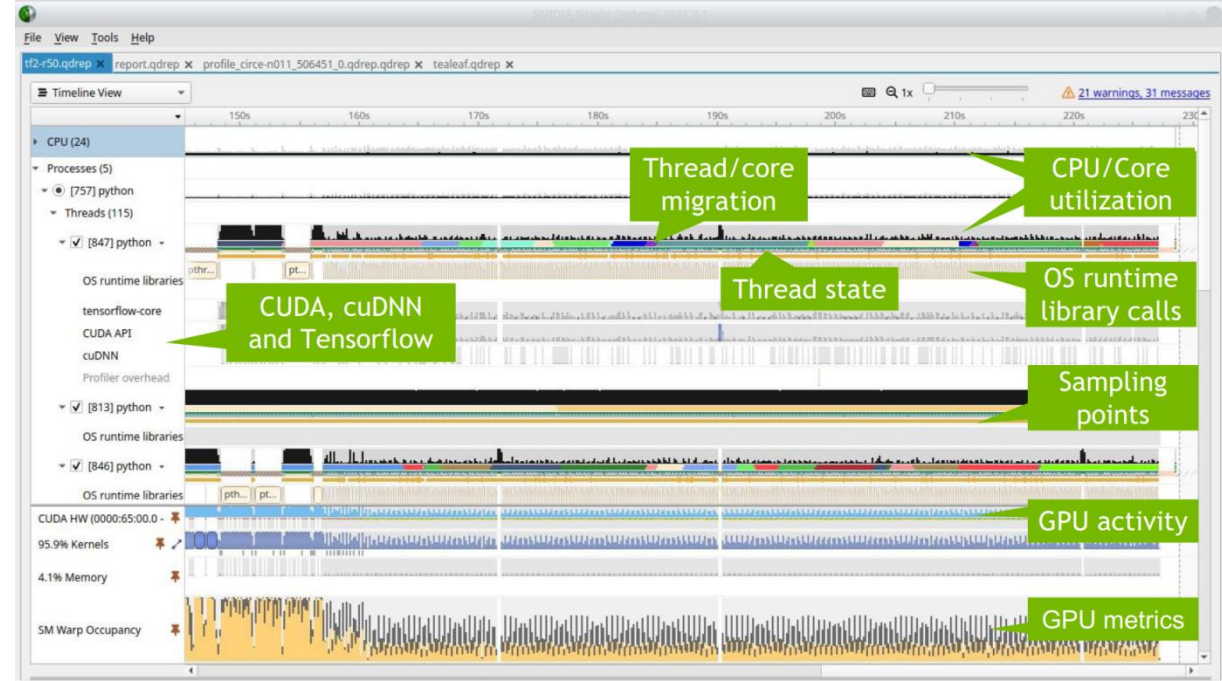
Nsight Compute

Standalone GUI+CLI

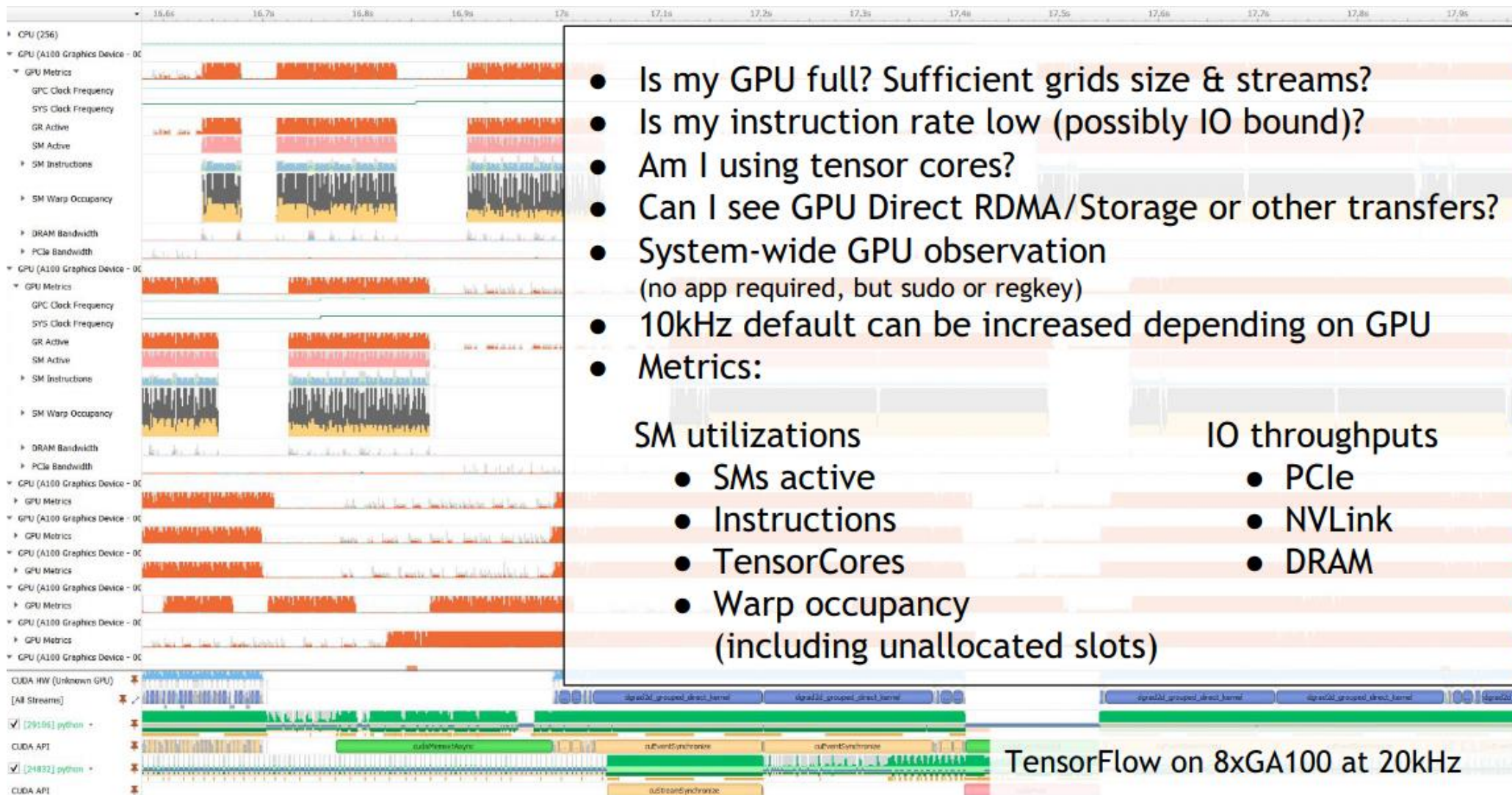
- GPU CUDA kernel analysis & debug
- Very high freq GPU perf counters
- Compare results (diff)
- Incredible statistics & customizable

NSIGHT SYSTEM

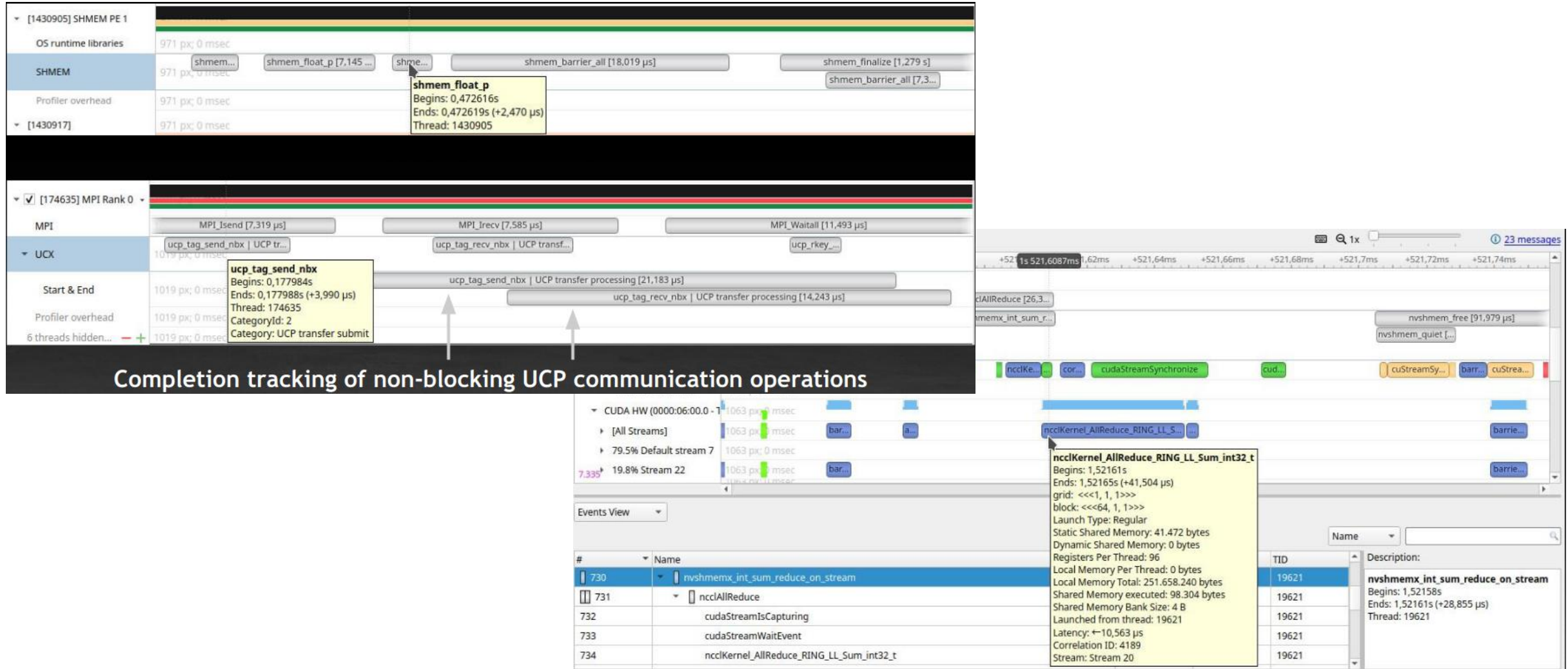
- System-wide application tuning
- Locate optimization opportunities
 - Visualize millions of events on a timeline
 - See gaps of unused CPU and GPU time
- Balance workloads across multiple CPUs and GPUs
 - CPU utilization and thread state
 - GPU streams, kernels, memory transfers, etc.
- Multi-platform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



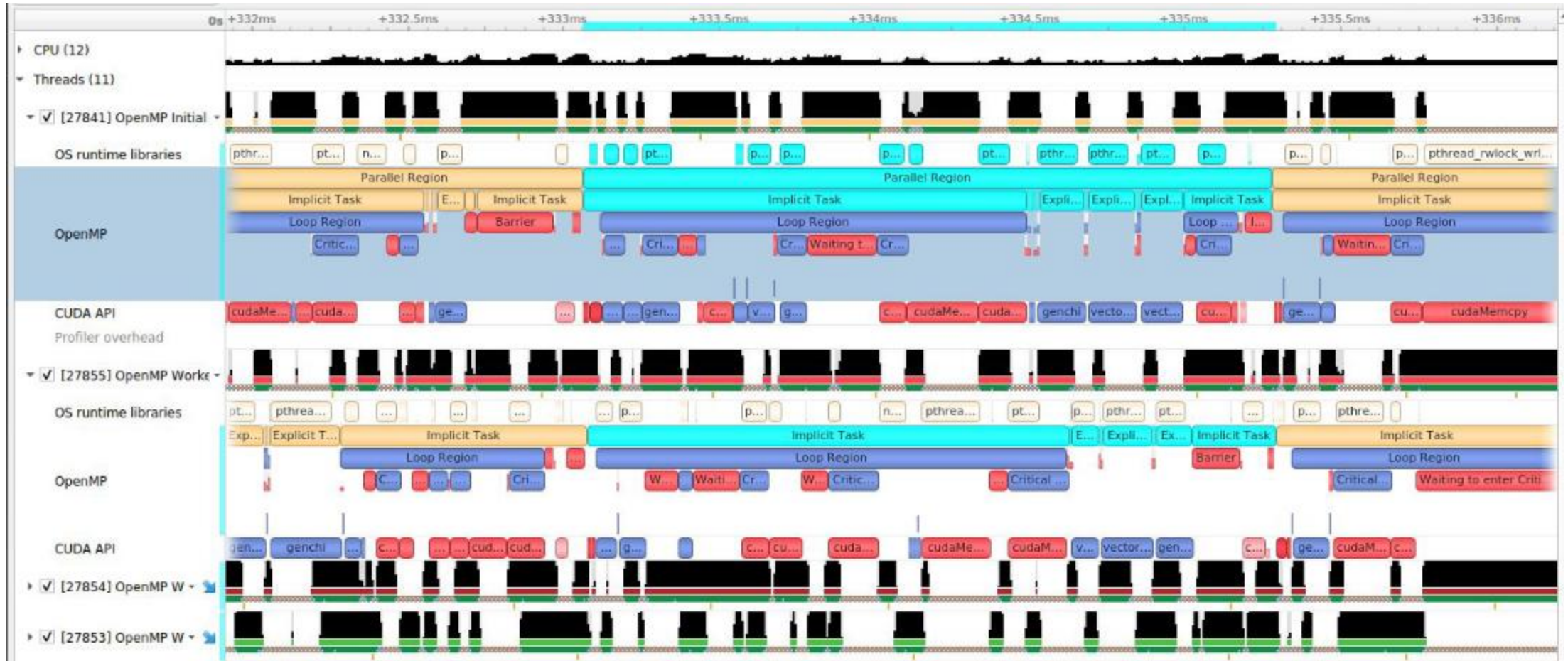
GPU METRIC SAMPLING



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



OPENMP



OMPT-capable OpenMP runtime required

EXPERT SYSTEM

The screenshot displays the NVIDIA Nsight Systems interface. The top section shows a timeline view of various CUDA API calls. A specific call, 'cudaMemc...', is highlighted in cyan. An expert system overlay on the right lists several performance issues, with 'CUDA Async Memcpy with Pageable Memory' selected. Below the timeline, the 'Expert System View' section provides a detailed table of the identified issue.

Duration	Start	Src Kind	Dst Kind	Bytes	PID	Device ID	Context ID	Stream ID	API Name
2,048 µs	6,38792s	Device	Pageable	8 B	75475	0	0	1	7 cudaMemcpy
2,048 µs	6,8334s	Device	Pageable	4 B	75475	0	0	1	7 cudaMemcpy
2,016 µs	2,5394s	Device	Pageable	4 B	75475	0	0	1	7 cudaMemcpy
2,016 µs	3,90617s	Device	Pageable	48 B	75475	0	0	1	7 cudaMemcpy
2,016 µs	4,25257s	Device	Pageable	4 B	75475	0	0	1	7 cudaMemcpy
2,016 µs	5,67617s	Device	Pageable	48 B	75475	0	0	1	7 cudaMemcpy
2,016 µs	5,9572s	Device	Pageable	8 B	75475	0	0	1	7 cudaMemcpy
2,016 µs	5,97088s	Device	Pageable	4 B	75475	0	0	1	7 cudaMemcpy

The expert system also provides the following text:

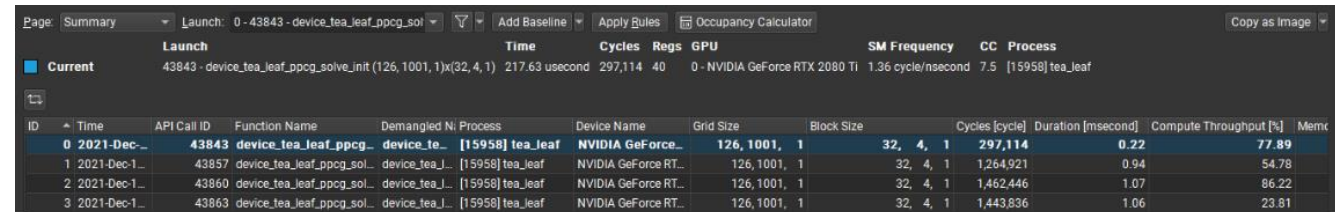
The following APIs use PAGEABLE memory which causes asynchronous CUDA memcpy operations to block and be executed synchronously. This leads to low GPU utilization.

Suggestion: If applicable, use PINNED memory instead.

CLI command:
`nsys analyze -r cuda-async-memcpy /mnt/data/traces/qdrep/ncl/profile_circe-n011_506451_0.sqlite`

NSIGHT COMPUTE

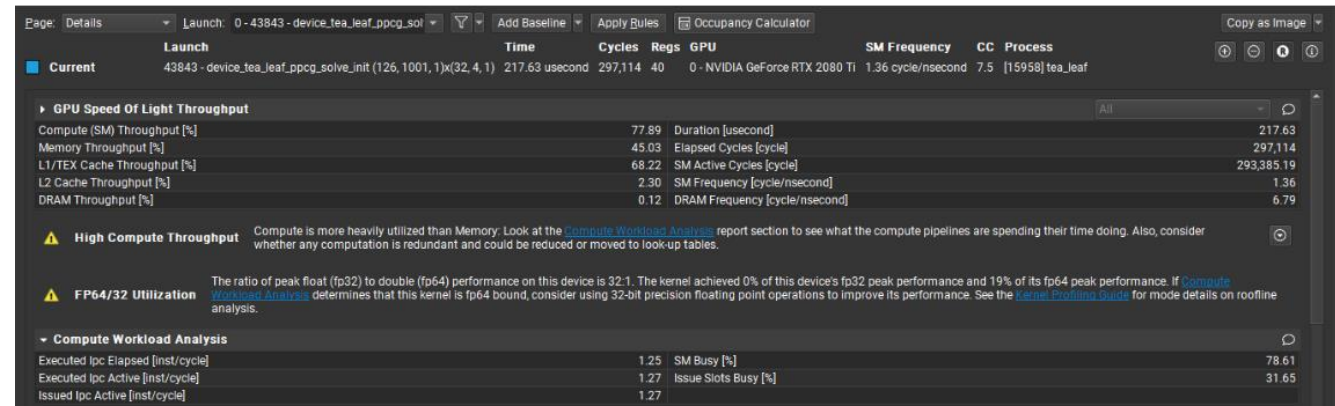
- Interactive CUDA kernel profiler
- Targeted metric sections for various performance aspects
- Customizable data collection and presentation (tables, charts, ...)
- GUI and CLI
- Python-based API for guided analysis and post-processing
- Support for remote profiling across machines and platforms



Page: Summary | Launch: 0 - 43843 - device_tea_leaf_ppcg_sol | Add Baseline | Apply Rules | Occupancy Calculator | Copy as Image

Launch	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	43843 - device_tea_leaf_ppcg_solve_init (126, 1001, 1)x(32, 4, 1)	217.63 usecond	297,114	40	0 - NVIDIA GeForce RTX 2080 Ti	1.36 cycle/nsecond	7.5 [15958] tea_leaf

ID	Time	API Call ID	Function Name	Demangled N:	Process	Device Name	Grid Size	Block Size	Cycles [cycle]	Duration [msecond]	Compute Throughput [%]	Memc
0	2021-Dec-1...	43843	device_tea_leaf_ppcg...	device_te...	[15958] tea_leaf	NVIDIA GeForce...	126, 1001, 1	32, 4, 1	297,114	0.22	77.89	
1	2021-Dec-1...	43857	device_tea_leaf_ppcg_sol...	device_tea_...	[15958] tea_leaf	NVIDIA GeForce RT...	126, 1001, 1	32, 4, 1	1,264,921	0.94	54.78	
2	2021-Dec-1...	43860	device_tea_leaf_ppcg_sol...	device_tea_...	[15958] tea_leaf	NVIDIA GeForce RT...	126, 1001, 1	32, 4, 1	1,462,446	1.07	86.22	
3	2021-Dec-1...	43863	device_tea_leaf_ppcg_sol...	device_tea_...	[15958] tea_leaf	NVIDIA GeForce RT...	126, 1001, 1	32, 4, 1	1,443,836	1.06	23.81	



Page: Details | Launch: 0 - 43843 - device_tea_leaf_ppcg_sol | Add Baseline | Apply Rules | Occupancy Calculator | Copy as Image

Current: 43843 - device_tea_leaf_ppcg_solve_init (126, 1001, 1)x(32, 4, 1) | 217.63 usecond | 297,114 | 40 | 0 - NVIDIA GeForce RTX 2080 Ti | 1.36 cycle/nsecond | 7.5 [15958] tea_leaf

GPU Speed Of Light Throughput

Metric	Value	Duration [usecond]	Elapsed Cycles [cycle]
Compute (SM) Throughput [%]	77.89	217.63	297,114
Memory Throughput [%]	45.03		293,385.19
L1/TEX Cache Throughput [%]	68.22		1.36
L2 Cache Throughput [%]	2.30		6.79
DRAM Throughput [%]	0.12		

High Compute Throughput Compute is more heavily utilized than Memory: Look at the [Compute Workload Analysis](#) report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

FP64/32 Utilization The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 0% of this device's fp32 peak performance and 19% of its fp64 peak performance. If [Compute Workload Analysis](#) determines that this kernel is fp64 bound, consider using 32-bit precision floating point operations to improve its performance. See the [Kernel Profiling Guide](#) for more details on routine analysis.

Compute Workload Analysis

Metric	Value	SM Busy [%]
Executed lpc Elapsed [inst/cycle]	1.25	78.61
Executed lpc Active [inst/cycle]	1.27	31.65
Issued lpc Active [inst/cycle]	1.27	

PROFILER REPORT

Selected result

Metric values

The screenshot shows a GPU profiler report interface. At the top, there is a navigation bar with 'Page: Details', a 'Launch' dropdown menu (highlighted with a green box), and buttons for 'Add Baseline', 'Apply Rules', and 'Occupancy Calculator'. Below this is a table with columns: Launch, Time, Cycles, Regs, GPU, SM Frequency, CC, and Process. The 'Current' launch is highlighted with a blue square. Below the table, there are several sections: 'GPU Speed Of Light Throughput' with a table of metrics; 'High Compute Throughput' warning; 'FP64/32 Utilization' warning; and 'Compute Workload Analysis' section (highlighted with a green box) containing a table of IPC metrics. A green box also highlights the 'Duration [usecond]' and 'Elapsed Cycles [cycle]' rows in the throughput table. A vertical green line on the right side of the interface is labeled 'Metric values'.

Launch	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
43843 - device_tea_leaf_ppcg_solve_init (126, 1001, 1)x(32, 4, 1)	217.63 usecond	297,114	40	0 - NVIDIA GeForce RTX 2080 Ti	1.36 cycle/nsecond	7.5	[15958] tea_leaf

Metric	Value
Compute (SM) Throughput [%]	77.89
Memory Throughput [%]	45.03
L1/TEX Cache Throughput [%]	68.22
L2 Cache Throughput [%]	2.30
DRAM Throughput [%]	0.12
Duration [usecond]	217.63
Elapsed Cycles [cycle]	297,114
SM Active Cycles [cycle]	293,385.19
SM Frequency [cycle/nsecond]	1.36
DRAM Frequency [cycle/nsecond]	6.79

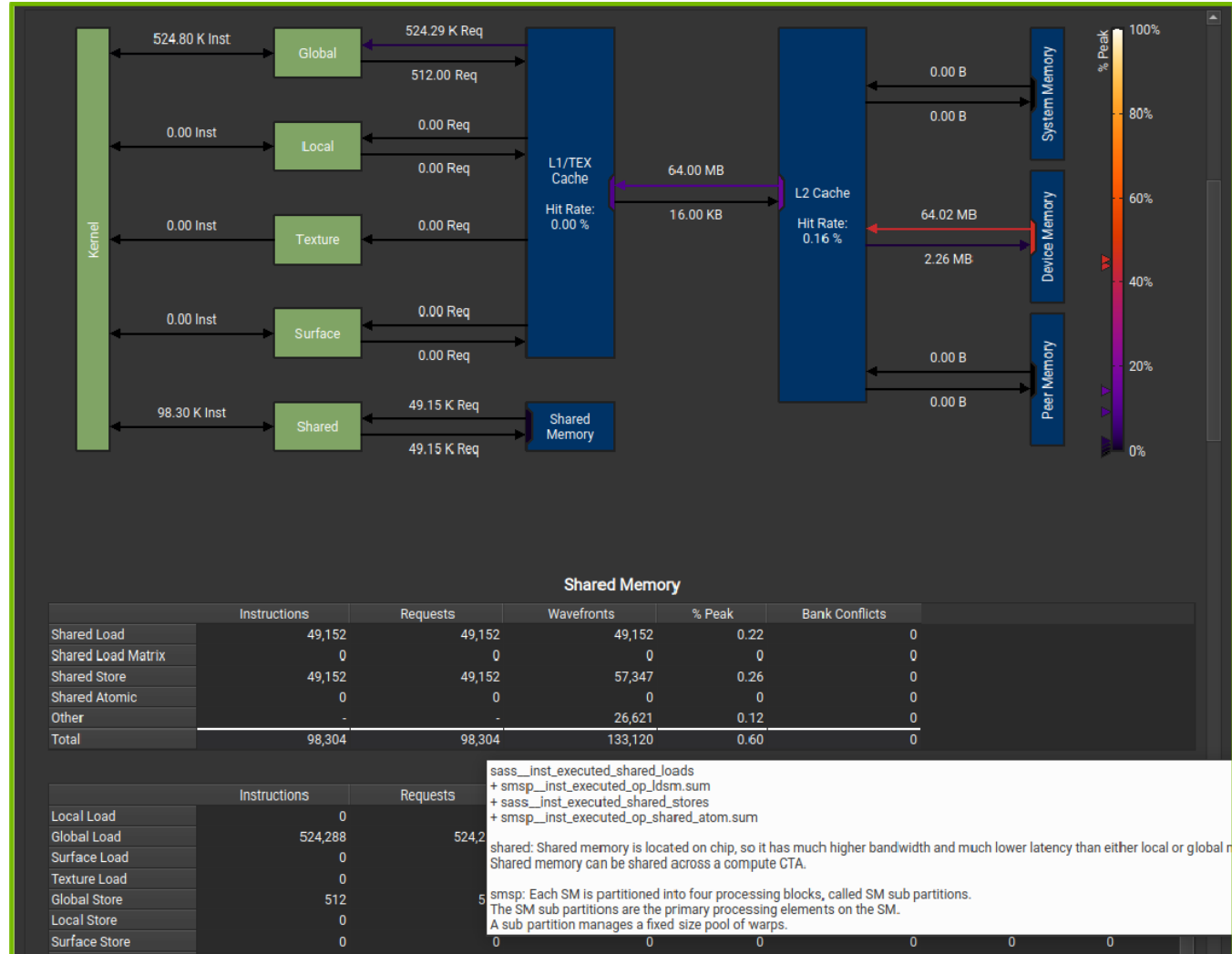
Metric	Value
Executed Ipc Elapsed [inst/cycle]	1.25
Executed Ipc Active [inst/cycle]	1.27
Issued Ipc Active [inst/cycle]	1.27
SM Busy [%]	78.61
Issue Slots Busy [%]	31.65

Expandable Sections

Expert Analysis (Rules)

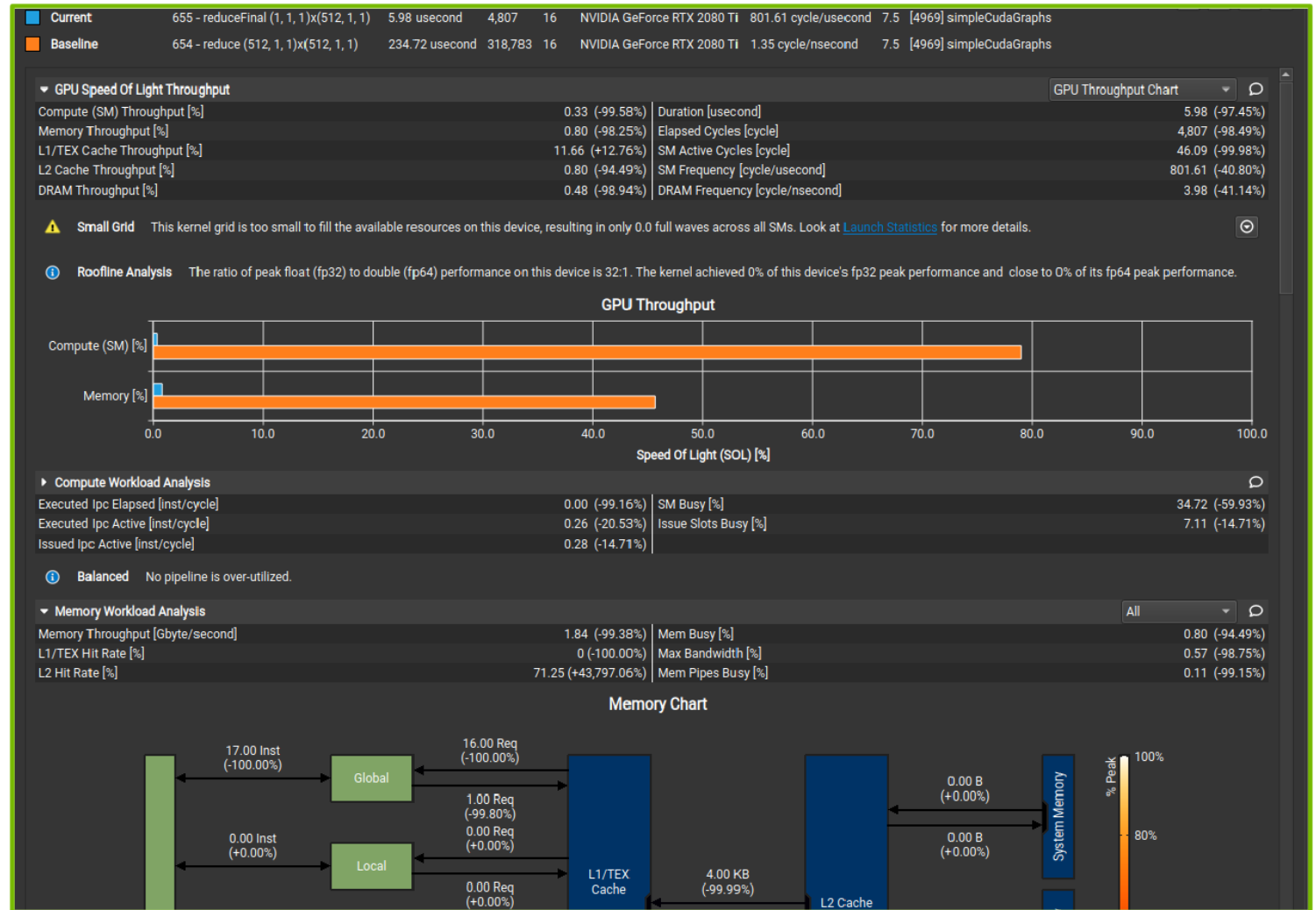
DATA TRANSFER ANALYSIS

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



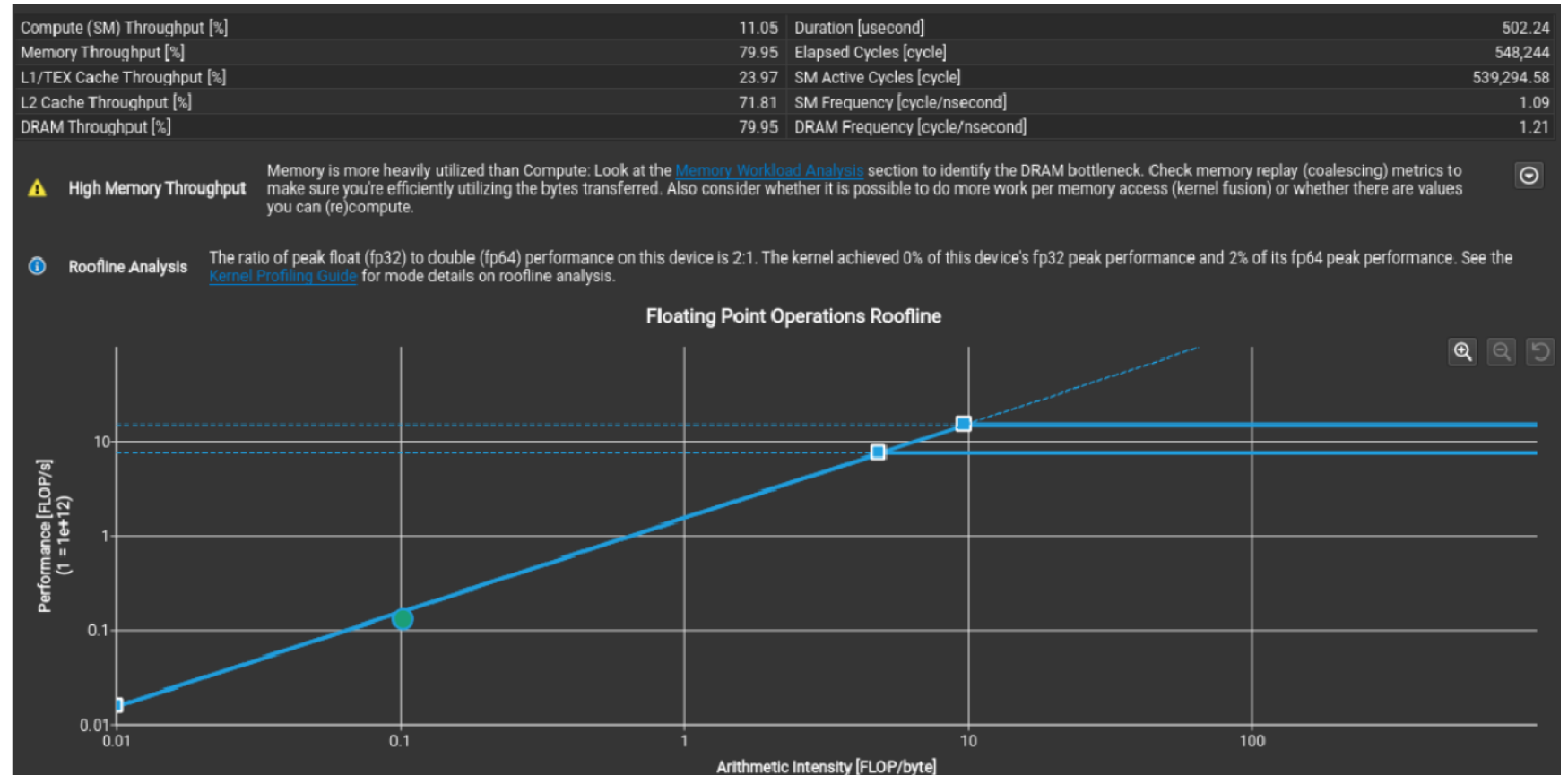
BASELINE COMPARISON

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



ROOFLINE ANALYSIS

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



DARSHAN

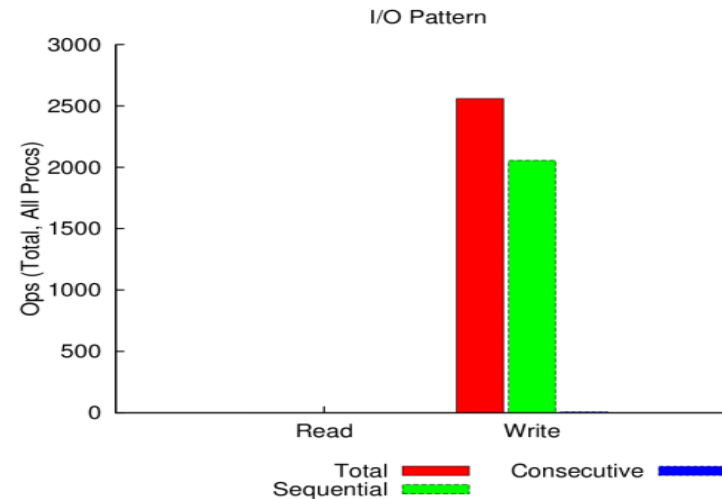
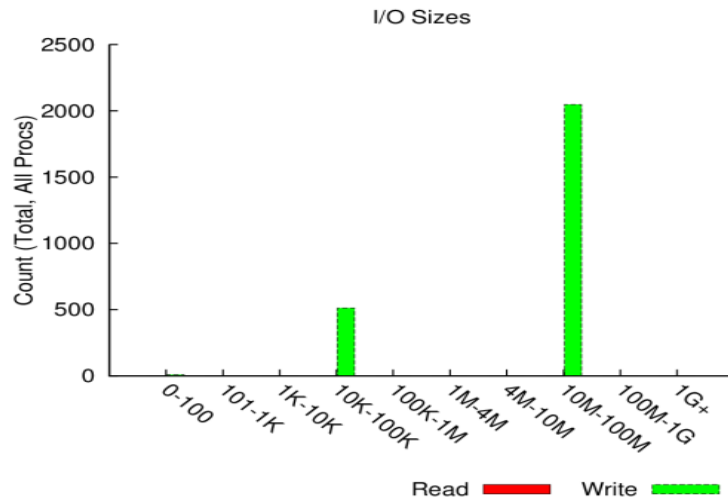
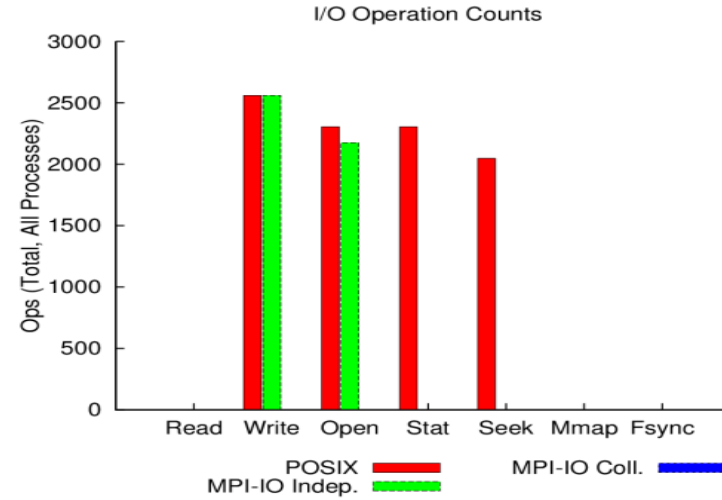
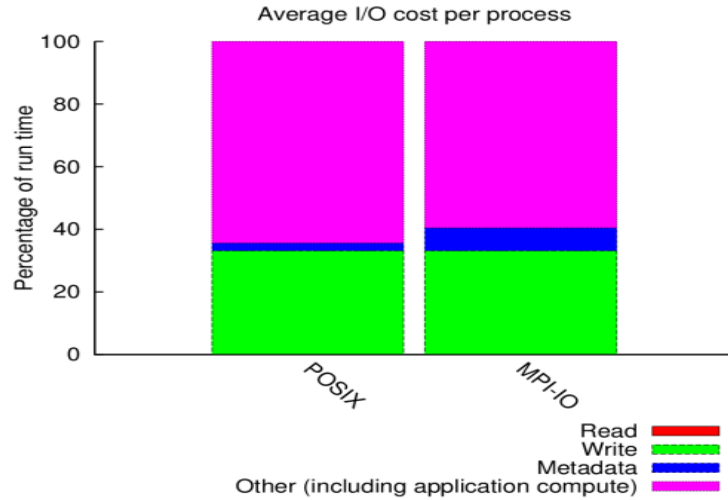
- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

- 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



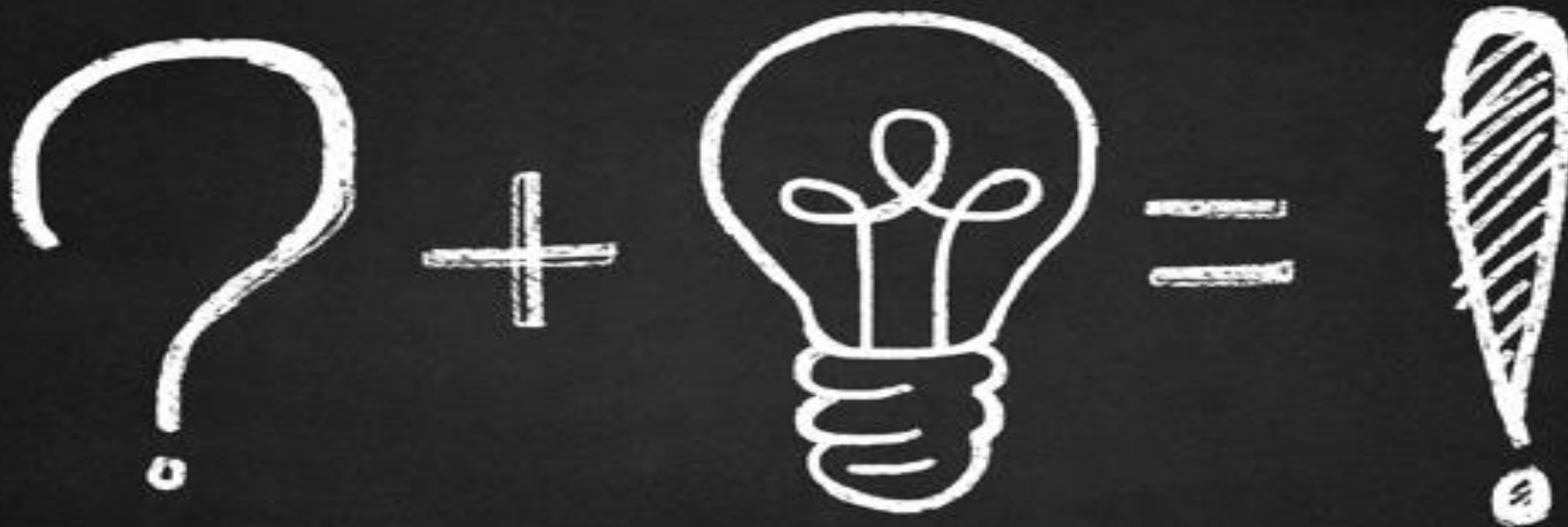
PERFORMANCE ANALYSIS RECOMMENDATIONS

- Measure and analyze at the desired scale (once you have a reasonable measurement setup)
- Get performance overview with Performance Reports
 - CPU Issues:
 - Use MAP, Vtune (on Intel nodes), or uProf (on AMD nodes)
 - Use perf / LIKWID / PAPI
 - 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. SC support, SimLab
 - Use mailing lists
 - JSC/NVIDIA Application Lab
 - ATML Parallel Performance
 - ATML Application Optimization and User Service Tools

👉 Successful performance engineering often is a collaborative effort



QUESTIONS