



HPC SOFTWARE – DEBUGGER AND PERFORMANCE ANALYSIS TOOLS

JUNE 1, 2023 | 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.

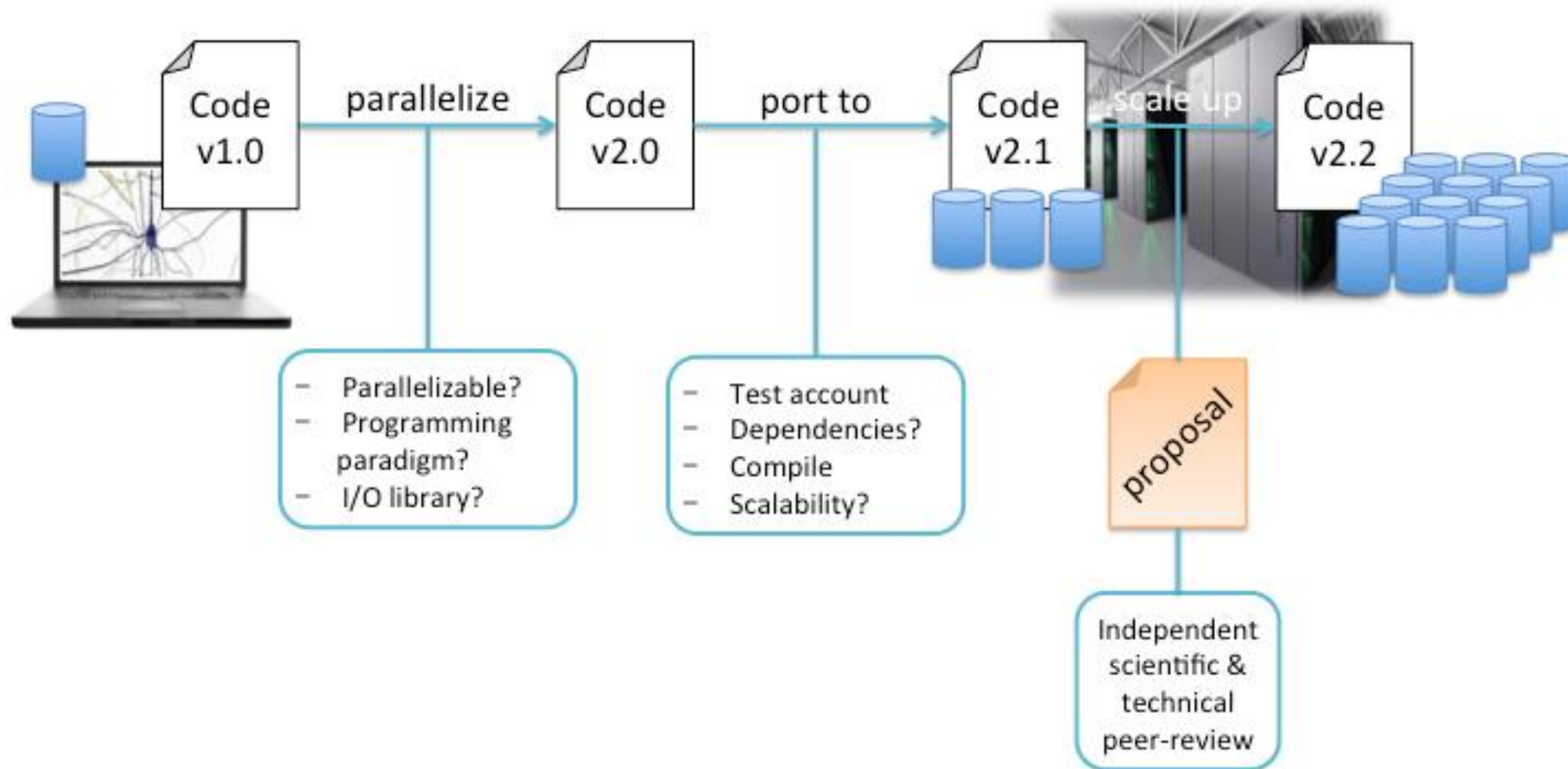
Kent Beck

Performance Analysis Tools

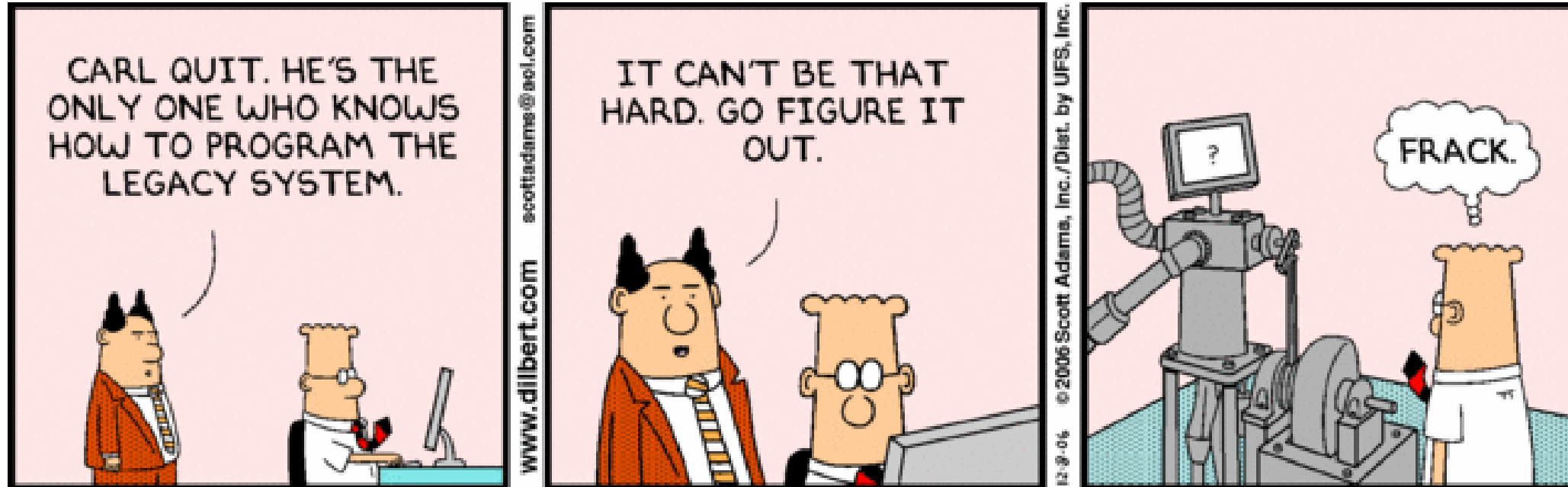
WHY SHOULD YOU CARE ABOUT TOOLS?



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



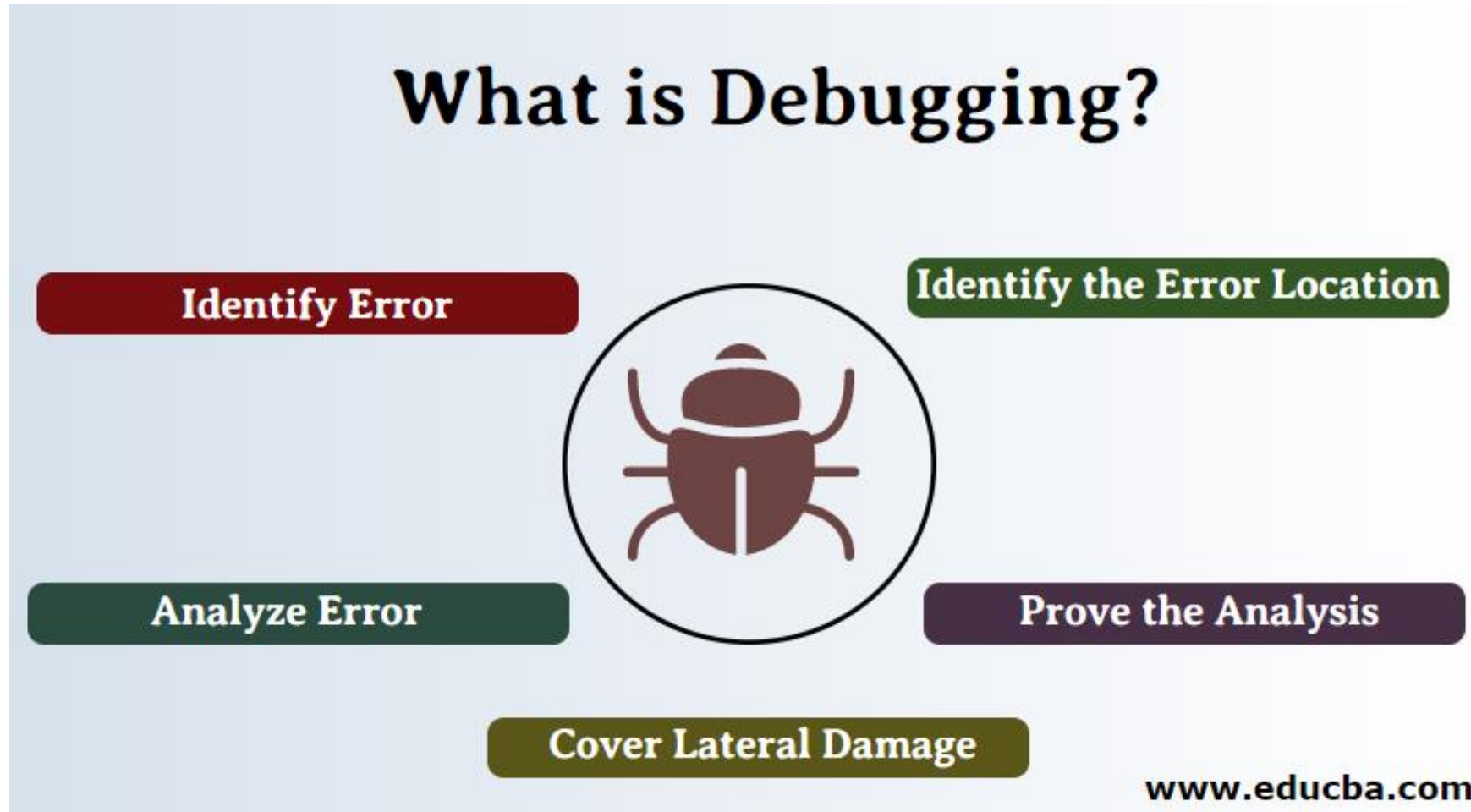
The screenshot displays a debugger interface with four main panels:

- Thread State Panel:** Shows a list of threads. Thread 1.1 is highlighted as a breakpoint, with state 'Stopped' and location 'tensorflow::SoftmaxXentWith...'. Other threads (1.2, 1.3, 1.4) are also 'Stopped' and in 'pthread_cond_wait' state.
- Variable Values Panel:** A table showing memory locations and values:

Name	Type	Value
...	int	0x0000000000000000...
nstar	int	0x000000006 (6)
grap...	int	0x000000015 (21)
- Source Code Panel:** Shows C++ code for TensorFlow's TF_Run function. Line 620 is highlighted in yellow, corresponding to the selected function in the call stack. The code includes comments for input/output names and target nodes, and uses vectors to manage these data structures.
- Call Stack Panel:** A list of functions in the call stack, with 'TF_Run' selected. The stack includes TensorFlow internal functions like 'TF_Run_Helper', 'TF_Run_wrapper', and 'ext_do_call', as well as Python wrappers like '_do_run'.

DEBUGGER & CORRECTNESS TOOLS

WHAT IS DEBUGGING?



REMINDER: DEBUGGING CAN BE FRUSTRATING



DEBUGGING TOOLS (STATUS: JUNE 2023)

- **Debugger:**

- CUDA-GDB
- TotalView
- ARMForge - DDT

- **Memory Analyzer:**

- CUDA-MEMCHECK
- Intel Inspector (older stage only)

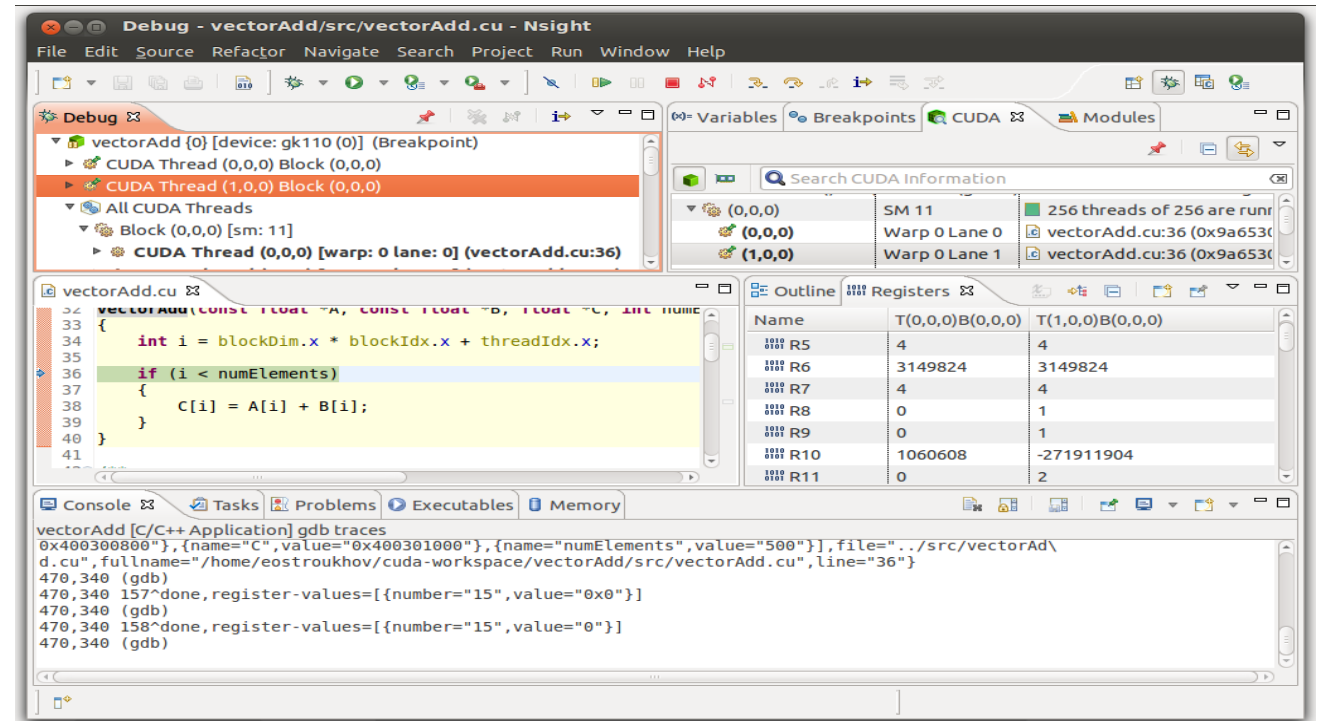
- **Correctness Checker:**

- 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.



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 execution 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

```
Applications Places System
File Edit View Terminal Help
linux64:~/demo2010$ ./ptrchecktest
unspecified launch failure : 79
linux64:~/demo2010$ cuda-memcheck ./ptrchecktest
===== CUDA-MEMCHECK
unspecified launch failure : 79
===== Invalid __global__ read of size 4
===== at 0x00000158 in ptrchecktest.cu:27:kernel2
===== by thread (0,0,0) in block (0,0)
===== Address 0xfd0000001 is misaligned
=====
===== ERROR SUMMARY: 1 error
linux64:~/demo2010$ cuda-memcheck --continue ./ptrchecktest
===== CUDA-MEMCHECK
Checking...
Done
Checking...
Error: 3 (0)
Done
Checking...
Error: 1 (0)
Error: 3 (0)
Error: 5 (0)
Error: 7 (0)
Done
===== Invalid __global__ read of size 4
===== at 0x00000158 in ptrchecktest.cu:27:kernel2
===== by thread (0,0,0) in block (0,0)
===== Address 0xfd0000001 is misaligned
=====
===== Invalid __global__ read of size 4
===== at 0x00000198 in ptrchecktest.cu:18:kernel1
===== by thread (3,0,0) in block (5,0)
===== Address 0xfd00000028 is out of bounds
=====
===== Invalid __global__ write of size 8
===== at 0x000001d0 in ptrchecktest.cu:38:kernel3
===== by thread (1,0,0) in block (8,0)
===== Address 0xfd00000204 is misaligned
=====
===== Invalid __global__ write of size 4
===== at 0x000000f0 in ptrchecktest.cu:44:kernel4
===== by thread (63,0,0) in block (22,0)
===== Address 0x00000000 is out of bounds
=====
===== ERROR SUMMARY: 4 errors
```

- 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 displays the CodeDynamics 2017 IDE interface with several panels and callouts:

- Toolbar for common options:** Located at the top, containing icons for running, pausing, and other debugging actions.
- Processes & Threads:** A tree view on the left showing the execution context, including process 'tx_cuda_matmul', thread '1.-1', and various system threads.
- Source code window:** The central pane showing C++ code for 'MatMulKernel'. Line 91 is highlighted in yellow.
- Stack trace:** A panel on the right showing the current call stack, with 'MatMulKernel' as the active frame.
- Local variables for selected stack frame:** A panel on the right showing variables like 'Matrix @local (Matrix)' and 'Matrix @local (Matrix)'. Below it, a 'Data View' table shows the structure of variable 'A':

Name	Type	Value
A	Matrix @local	(Matrix @local)
width	int	0x00000002 (2)
height	int	0x00000002 (2)
stride	int	0x00000002 (2)
elements	float @generic *	0xb03ee0000 -...
- Thread control:** A panel on the left with options to 'Control Group', 'Share Group', and 'Hostname'.
- Break points:** A table at the bottom left showing a breakpoint set at line 91 of 'tx_cuda_matmul'.

ID	Type	Stop	File	Line
1	BP	Process	tx_cuda_matmul	91

Break points

Thread control

Stack trace

Source code window

Local variables for selected stack frame

Toolbar for common options

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-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://developer.arm.com>
- **NOTE:** JSC license limited to 64 processes (shared between all users)

DDT: MAIN WINDOW

Process controls

CUDA Thread stepping

Variables

CUDA Thread control

GPU Device information

Source code

Expression evaluator

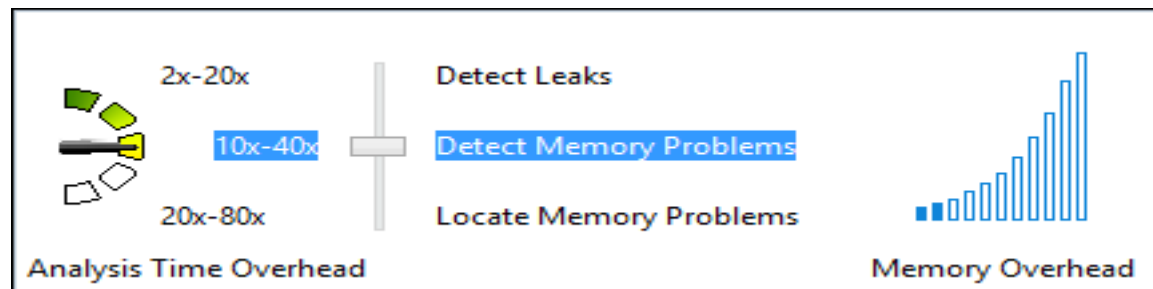
Stack trace

The screenshot displays the Arm DDT main window with the following components:

- Process controls:** Top toolbar with play, pause, and other execution controls.
- CUDA Thread stepping:** 'Step CUDA threads by: Warp (default)' dropdown and navigation arrows.
- Variables:** 'Locals' panel showing variables like 'cx', 'cy', 'output', 'x', 'y', and 'index' with their values.
- CUDA Thread control:** 'Threads' panel showing 'CUDA Threads (conv2d_global)' and 'Block 2'.
- GPU Device information:** 'GPU Devices' panel showing details for 'GM20B' such as 'Compute Capability sm_53', 'Number of SMs 1', 'Warpes per SM 64', 'Lanes per Warp 32', and 'Registers per Lane 256'.
- Source code:** Central editor showing C++ code for a 2D convolution filter.
- Stack trace:** 'Stacks' panel showing a list of threads and their current function locations.
- Expression evaluator:** 'Evaluate' panel showing the evaluation of expressions like 'x+cx + (y+cy)*width'.

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 a target named 'r000t2'. The main window is titled 'Detect Deadlocks and Data Races'. It features a 'Problems' table with the following data:

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' panel shows a list of sources and modules, including 'blocked_range.h', 'find_and_fix_threading_errors.c...', 'parallel_for.h', 'partitioner.h', 'task.h', 'task_scheduler_init.h', and 'winvideo.h'. The 'Module' filter is set to 'Find and fix threading errors.exe'.

This screenshot provides a detailed view of a data race problem. The 'Code Locations: Data race' panel shows the following code snippet:

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

The 'Description' column indicates a 'Write' operation at 'find_and_fix_threading_errors.cpp:105' in the 'render_one_pixel' function. The 'Variable' column shows 'col:r'. A 'HINT' at the bottom suggests a synchronization allocation site at 'task_scheduler_init.h:104'.

The screenshot shows the Intel Inspector XE 2015 interface for a target named 'r000t2'. The main window is titled 'Detect Memory Problems'. It features a 'Problems' table with the following data:

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

Below the table, the 'Code Locations: Invalid memory access' panel shows the following code snippet:

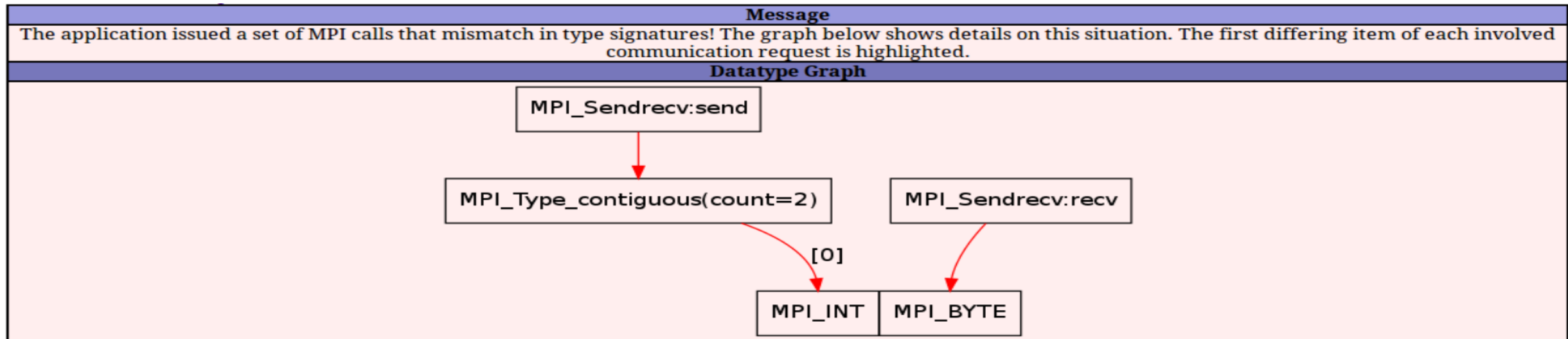
```
164
165 for (unsigned int i=0;i<=(mboxsize
166 local_mbox[i]=0; //Memory Err
167
168 for (int y = r.begin(); y != r.end
```

The 'Description' column indicates a 'Write' operation at 'find_and_fix_memory_error... operator()' in the 'find_and_fix_memory_error...' module. The 'Object Size' and 'Offset' columns are also visible.

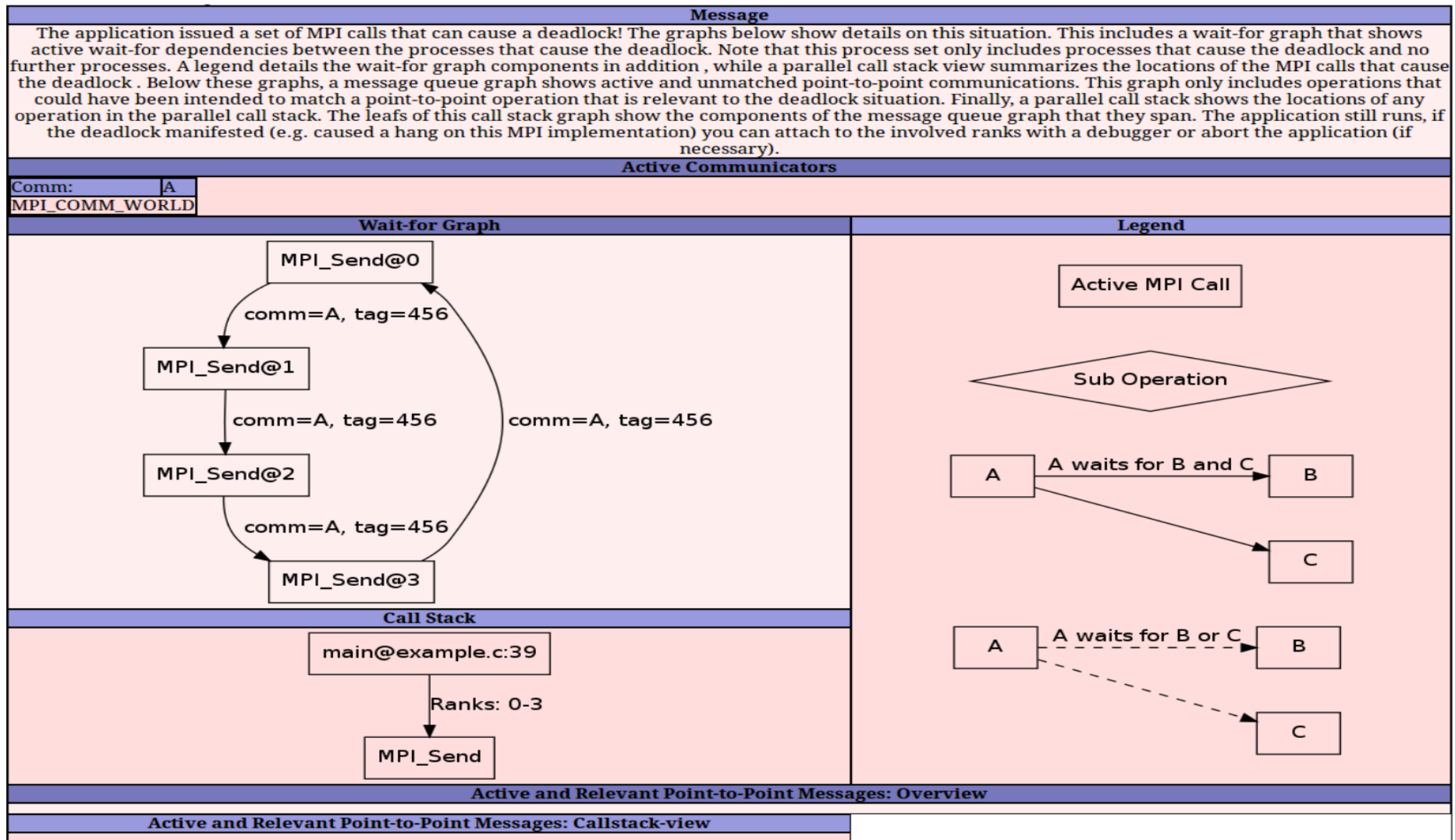
- Next generation MPI correctness and portability checker
- <https://www.i12.rwth-aachen.de/go/id/nrbe>
- MUST reports
 - Errors: violations of the MPI-standard
 - Warnings: unusual behavior or possible problems
 - Notes: harmless but remarkable behavior
 - Potential deadlock detection
- Usage
 - Relink 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	<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>

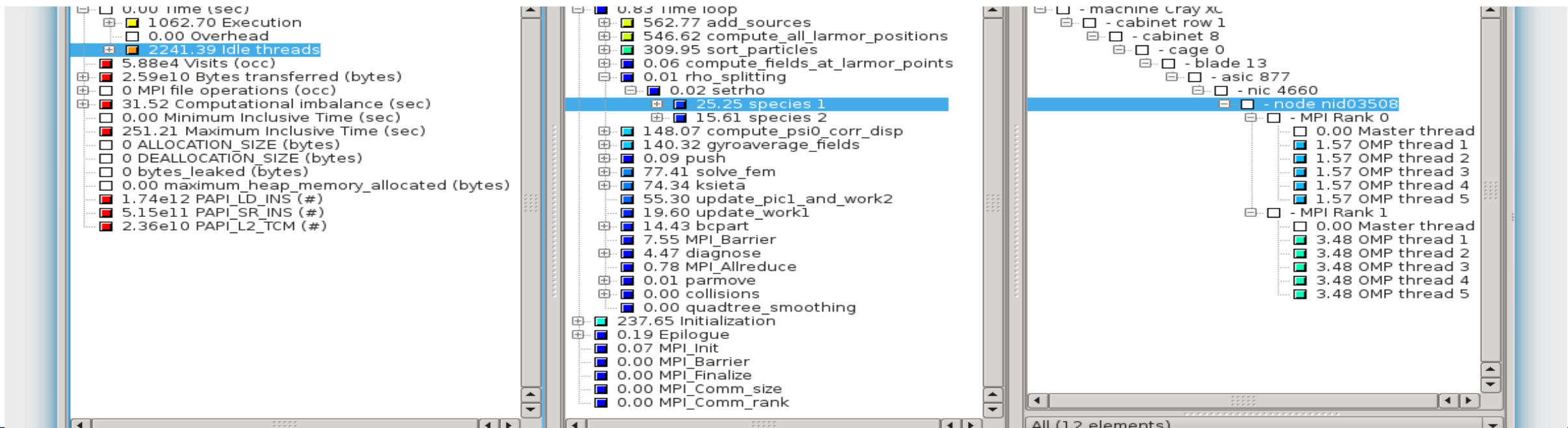


MUST DEADLOCK DETECTION



DEBUGGING RECOMMENDATIONS

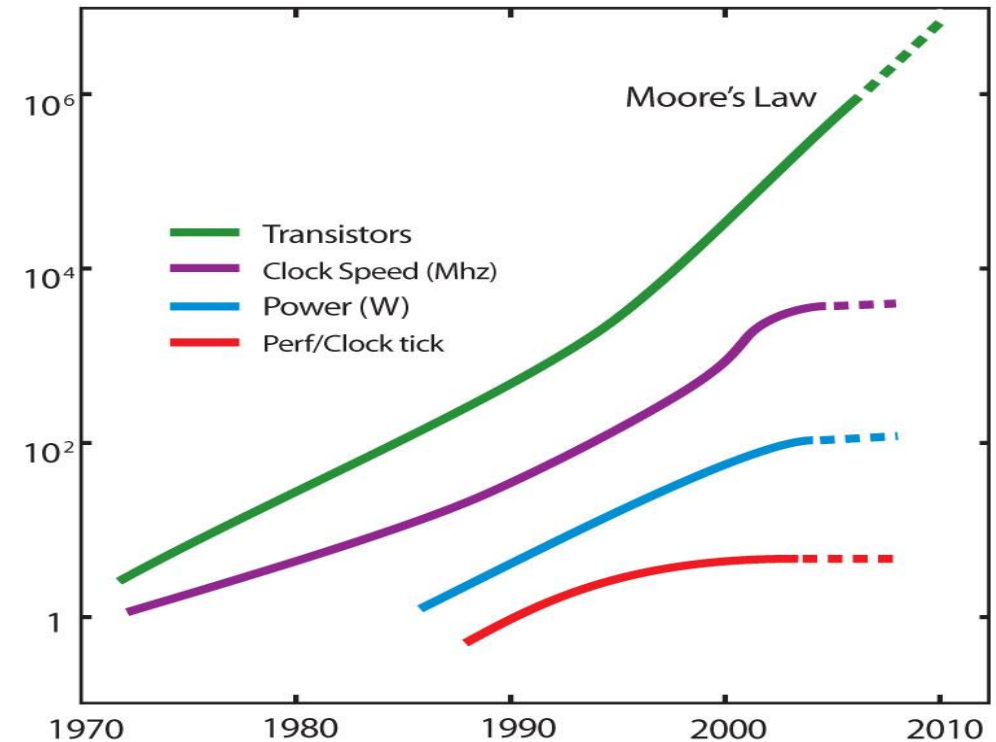
- Always debug at the lowest possible scale!
- GPU Applications:
 - Single Node: Use CUDA-GDB and CUDA-MEMCHECK
 - Multi-Node: Use TotalView/DDT
- MPI Applications:
 - Check with MUST at least once
 - Use TotalView/DDT at small scale (if error occurs there), else attach to as few processes as necessary



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!



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

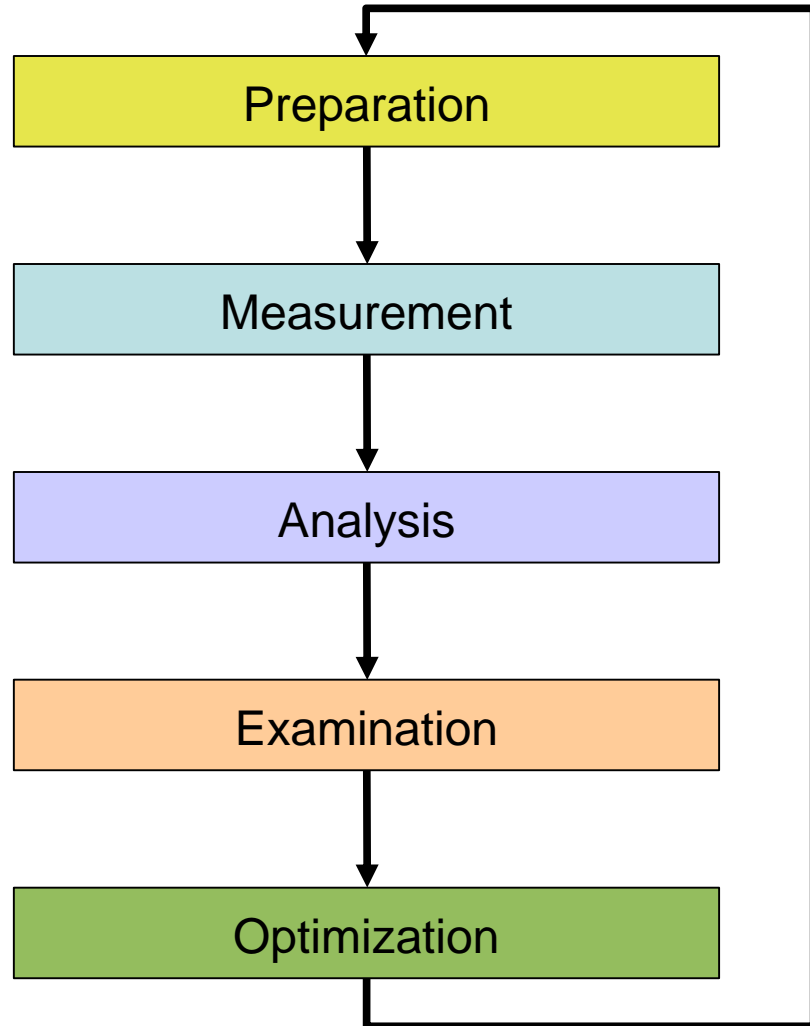
PERFORMANCE FACTORS

- “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

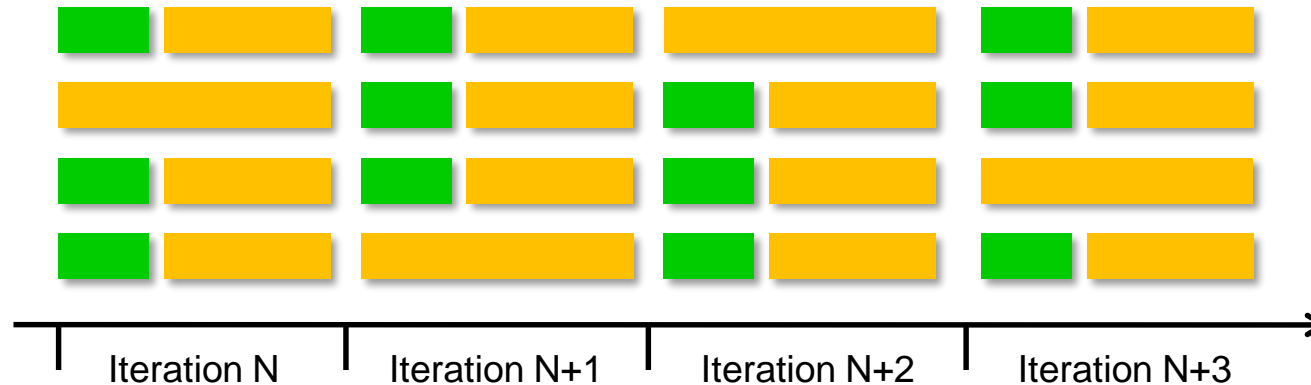
- Recording of **aggregated information**
 - Time
 - Counts
 - Calls
 - Hardware counters
- **Across program and system entities**
 - Functions, call sites, loops, basic blocks, ...
 - Processes, threads
- **Statistical information**
 - Min, max, mean and total number of values

Advantages
+ Works also for
long-running programs

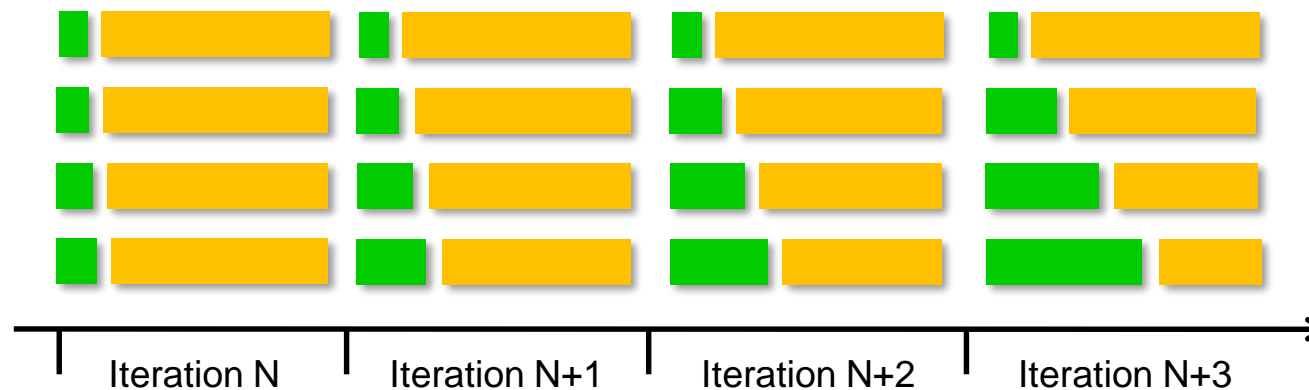
Disadvantages
– Variations over time
get lost

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

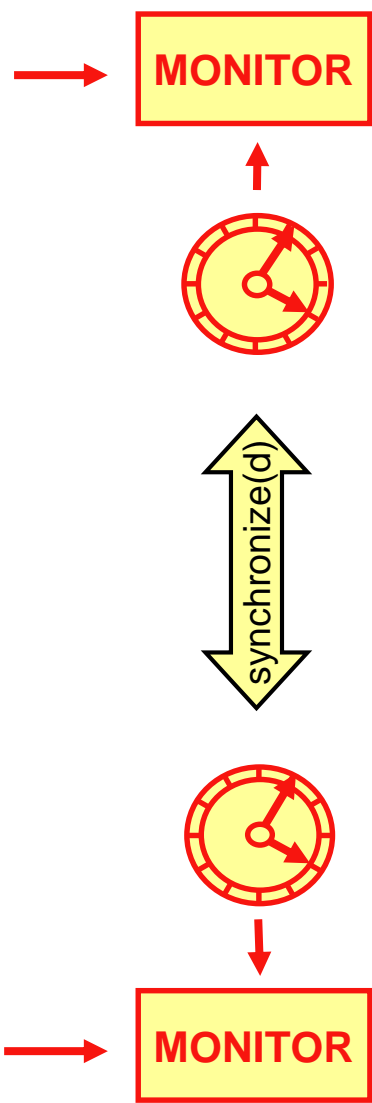
Process A

```
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
...			

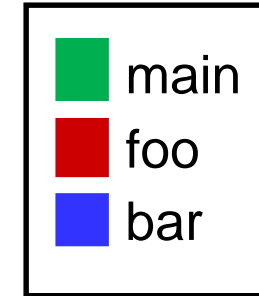
merge

unify

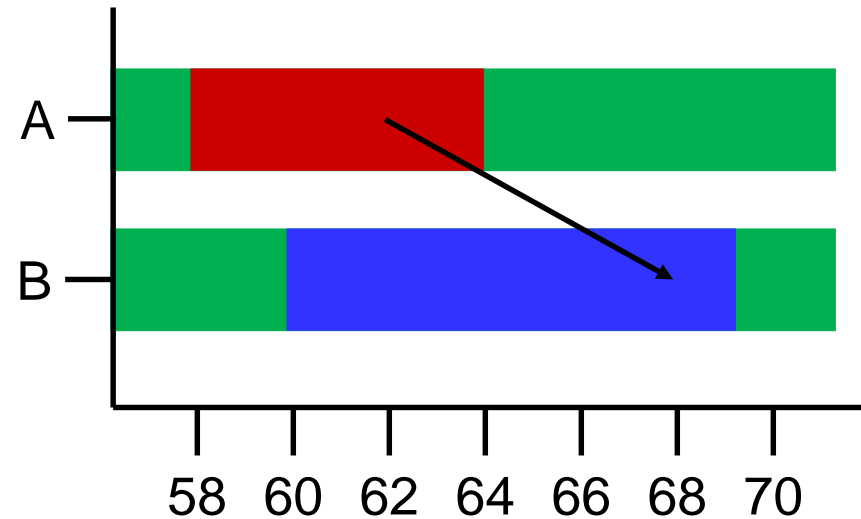
1	foo
2	bar
...	

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	RECV	A
69	B	EXIT	2
...			

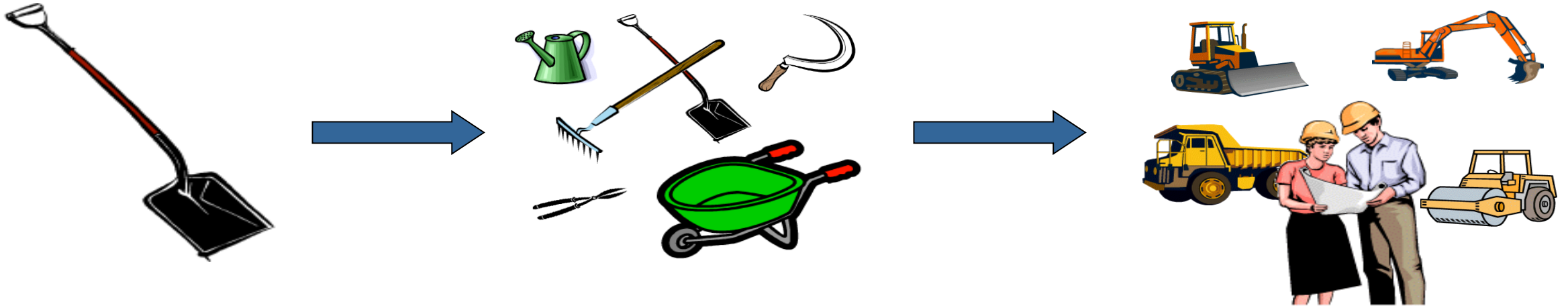


CRITICAL ISSUES

- 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*

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: JUNE 2023**)

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



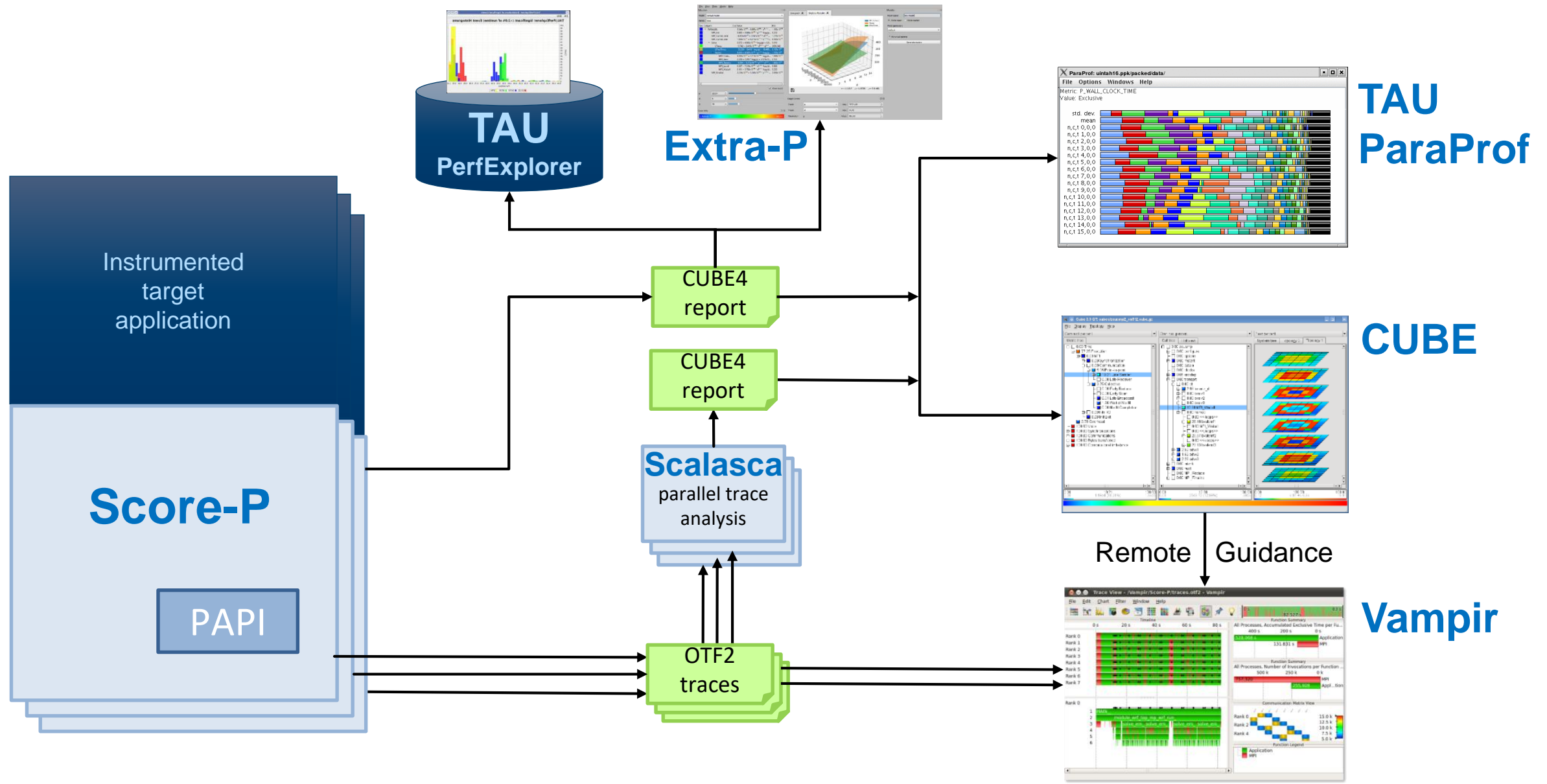
Score-P

Scalable performance measurement
infrastructure for parallel codes

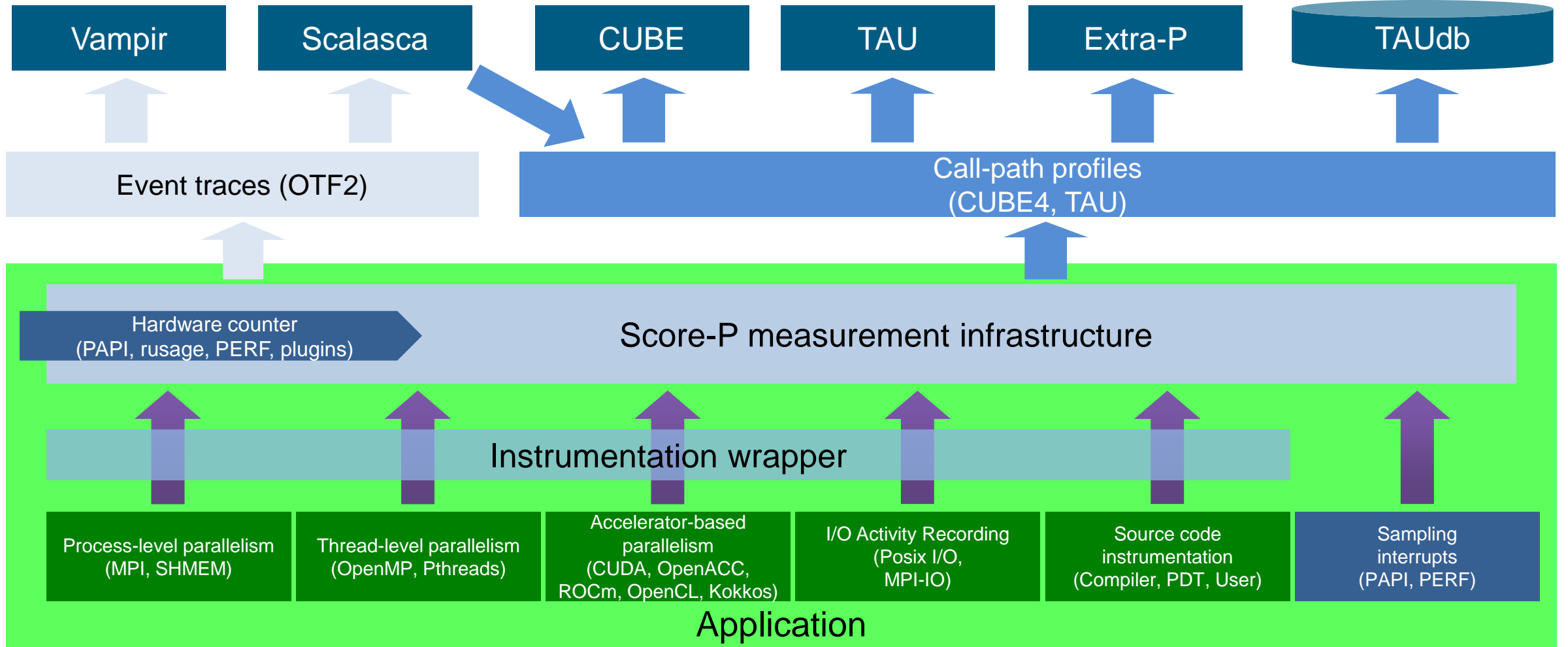
- Community-developed open-source
- Replaced tool-specific instrumentation and measurement components of partners
- <http://www.score-p.org>



Score-P TOOL ECOSYSTEM



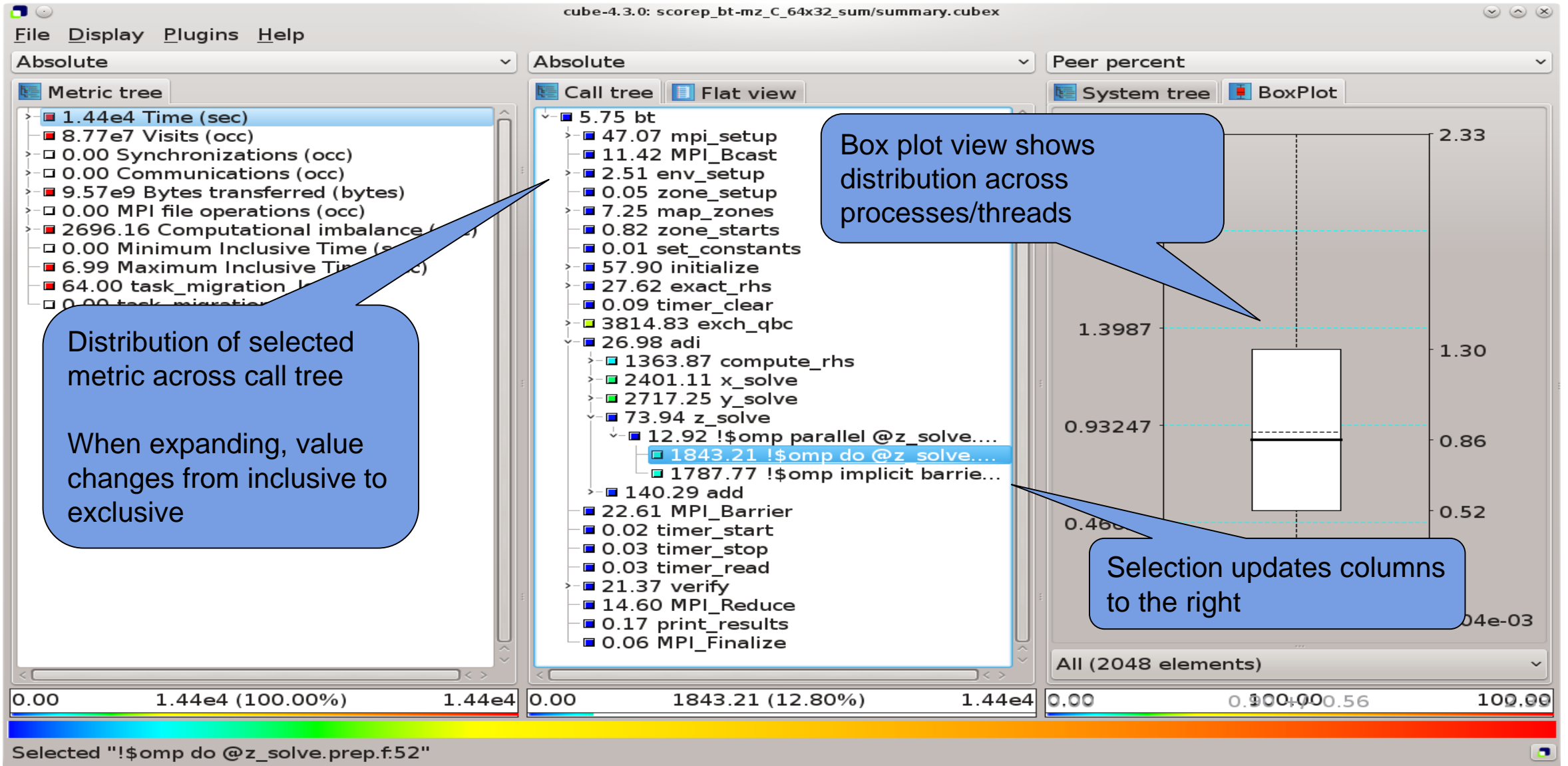
Score-P ARCHITECTURE



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

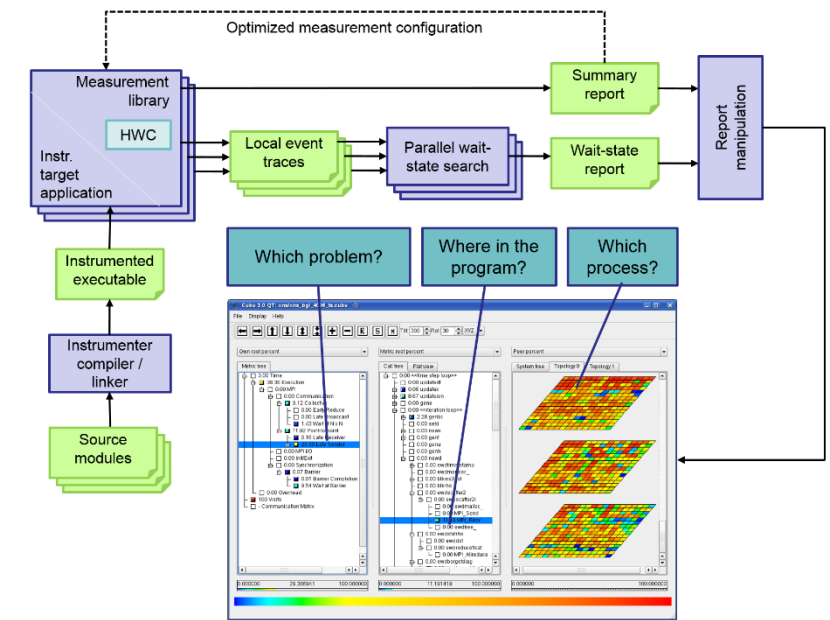
- Process trace in parallel

- Wait-state analysis

- Delay and root-cause analysis

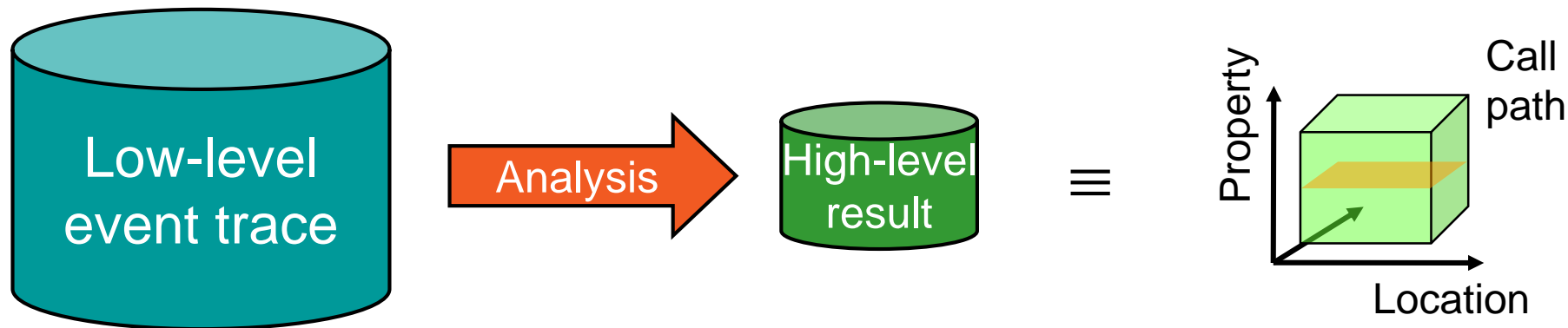
- Critical path analysis

- Categorize and rank results



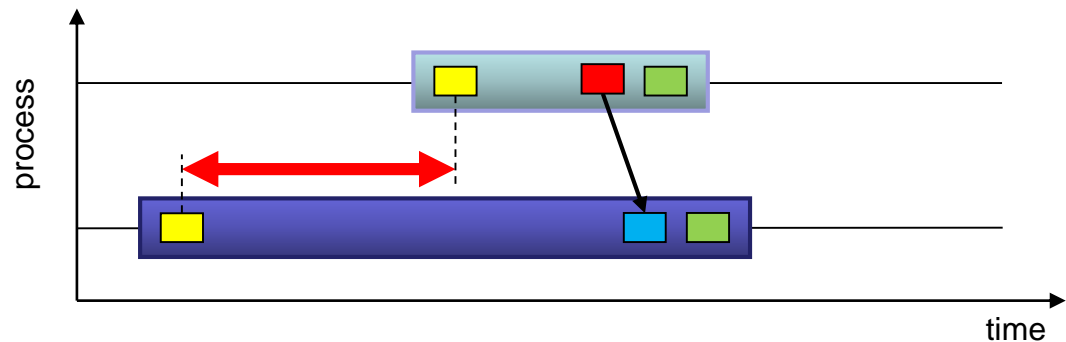
AUTOMATIC TRACE ANALYSIS

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

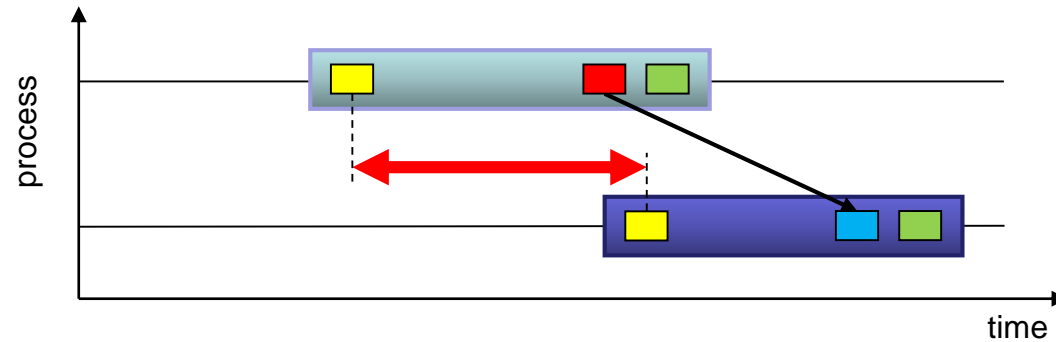


- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability

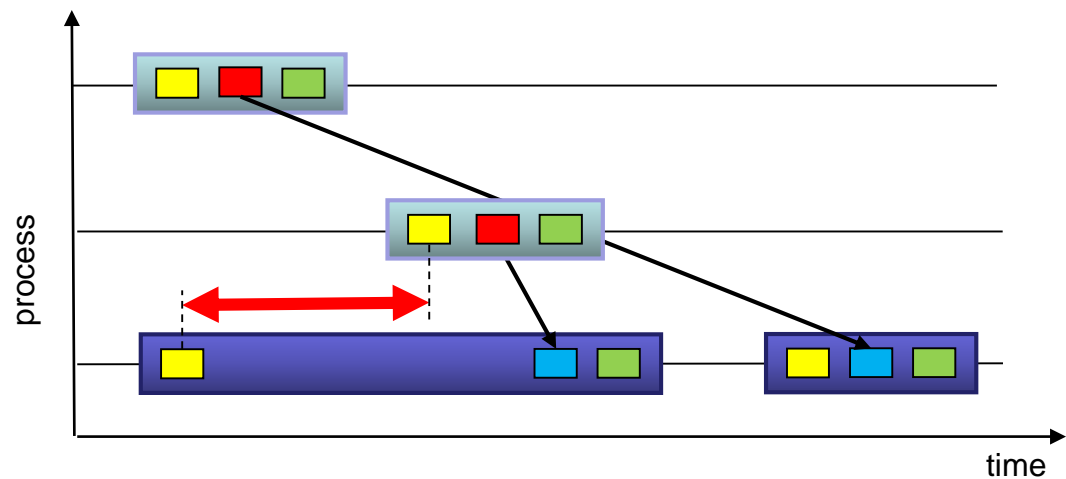
EXAMPLE MPI WAIT STATES



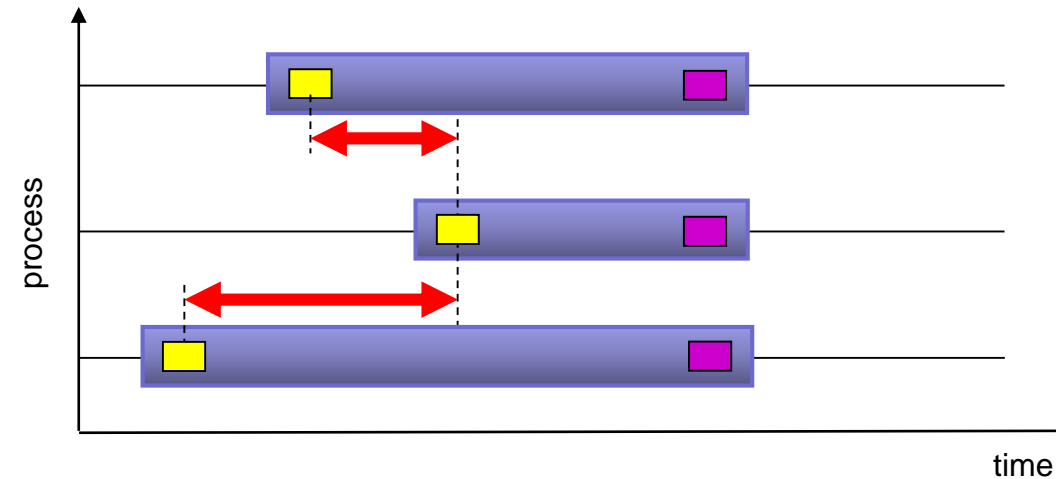
(a) Late Sender



(b) Late Receiver



(c) Late Sender / Wrong Order



(d) Wait at N x N

ENTER
 EXIT
 SEND
 RECV
 COLLEXIT

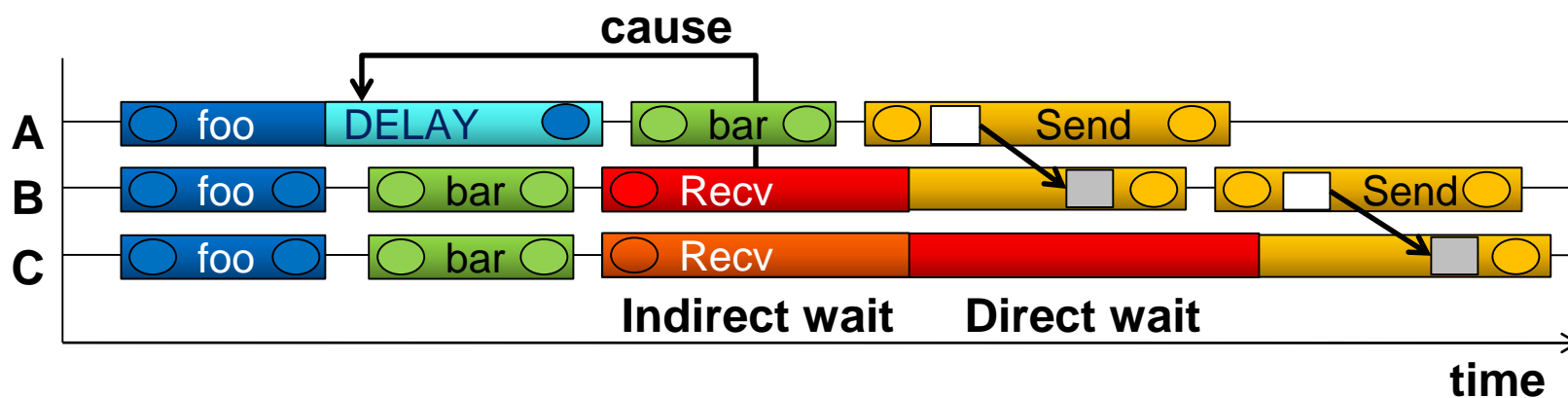
SCALASCA ROOT CAUSE ANALYSIS

- **Root-cause analysis**

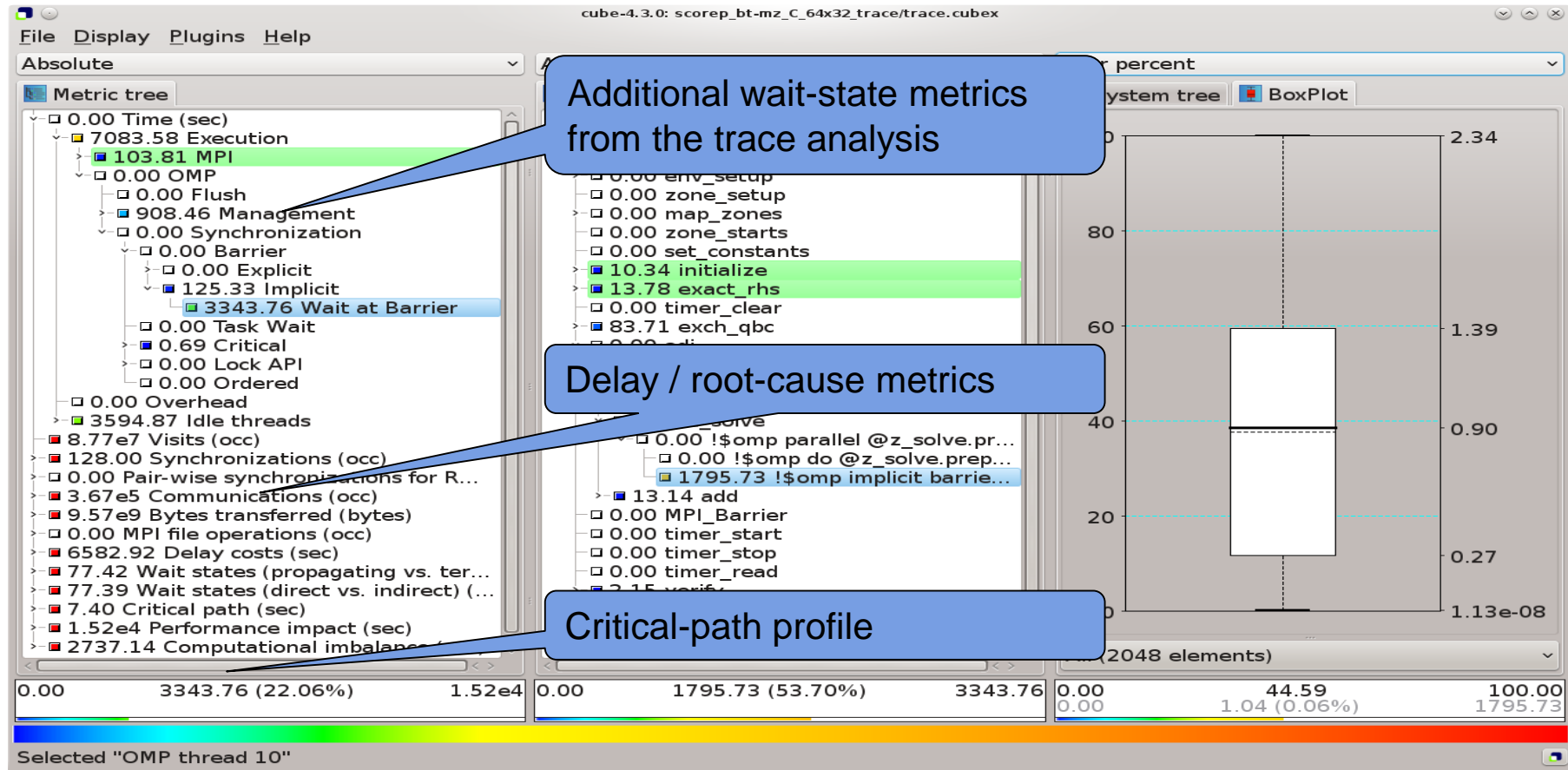
- Wait states typically caused by load or communication imbalances earlier in the program
- Waiting time can also propagate (e.g., indirect waiting time)
- Enhanced performance analysis to find the root cause of wait states

- **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**

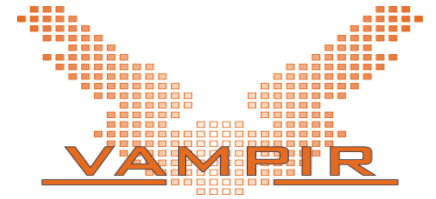


SCALASCA TRACE ANALYSIS EXAMPLE



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

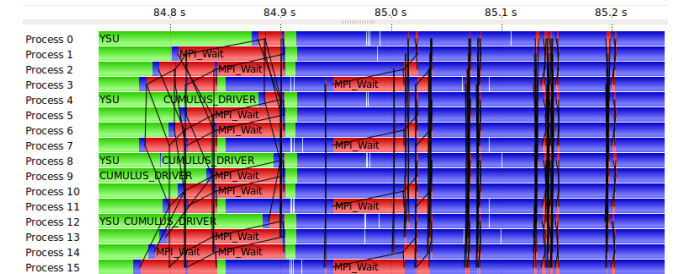


EVENT TRACE VISUALIZATION WITH VAMPIR

- 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?

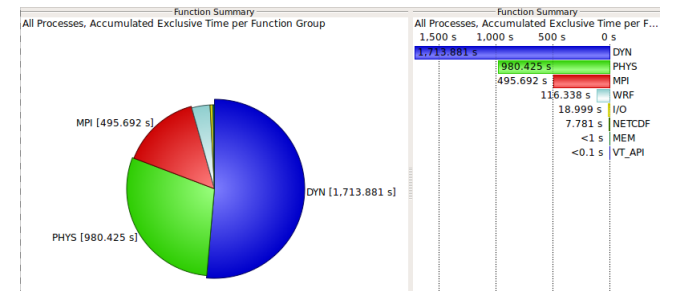
Timeline charts

- Application activities and communication along a time axis









Summary charts

- Quantitative results for the currently selected time interval









VAMPIR PERFORMANCE CHARTS

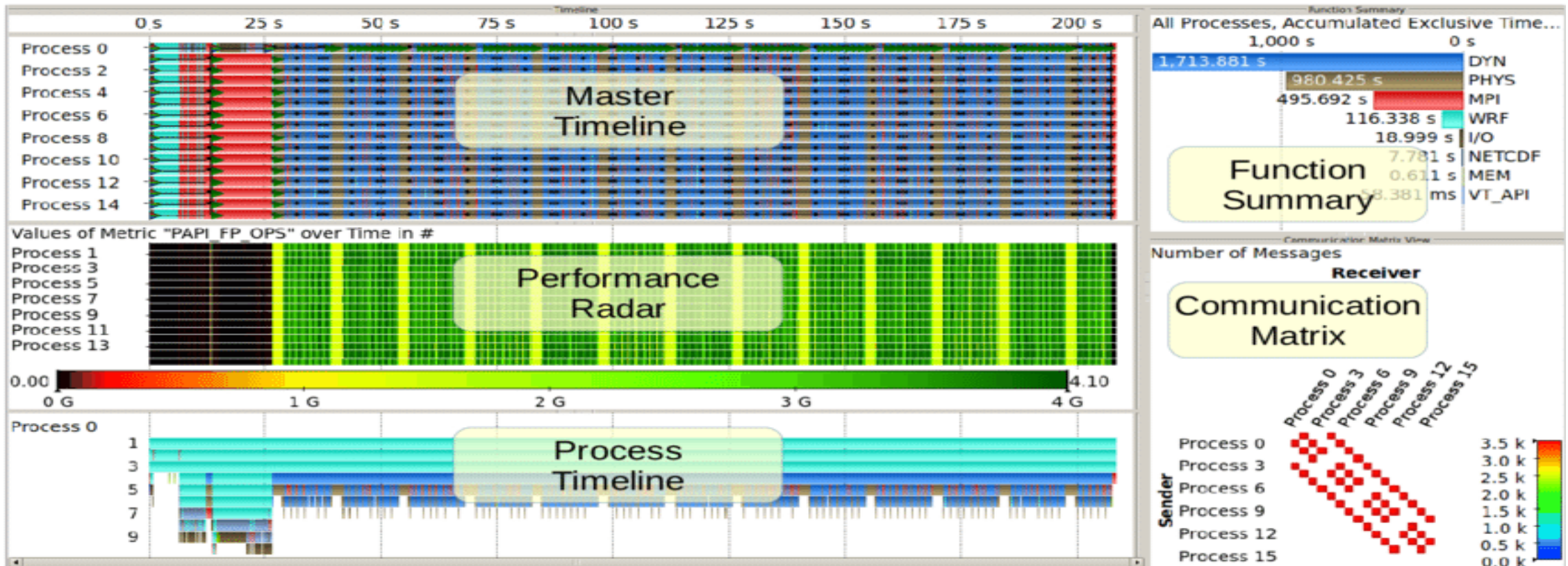
Timeline Charts

	Master Timeline	➔	<i>all threads' activities</i>
	Process Timeline	➔	<i>single thread's activities</i>
	Summary Timeline	➔	<i>all threads' function call statistics</i>
	Performance Radar	➔	<i>all threads' performance metrics</i>
	Counter Data Timeline	➔	<i>single threads' performance metrics</i>
	I/O Timeline	➔	<i>all threads' I/O activities</i>

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.


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



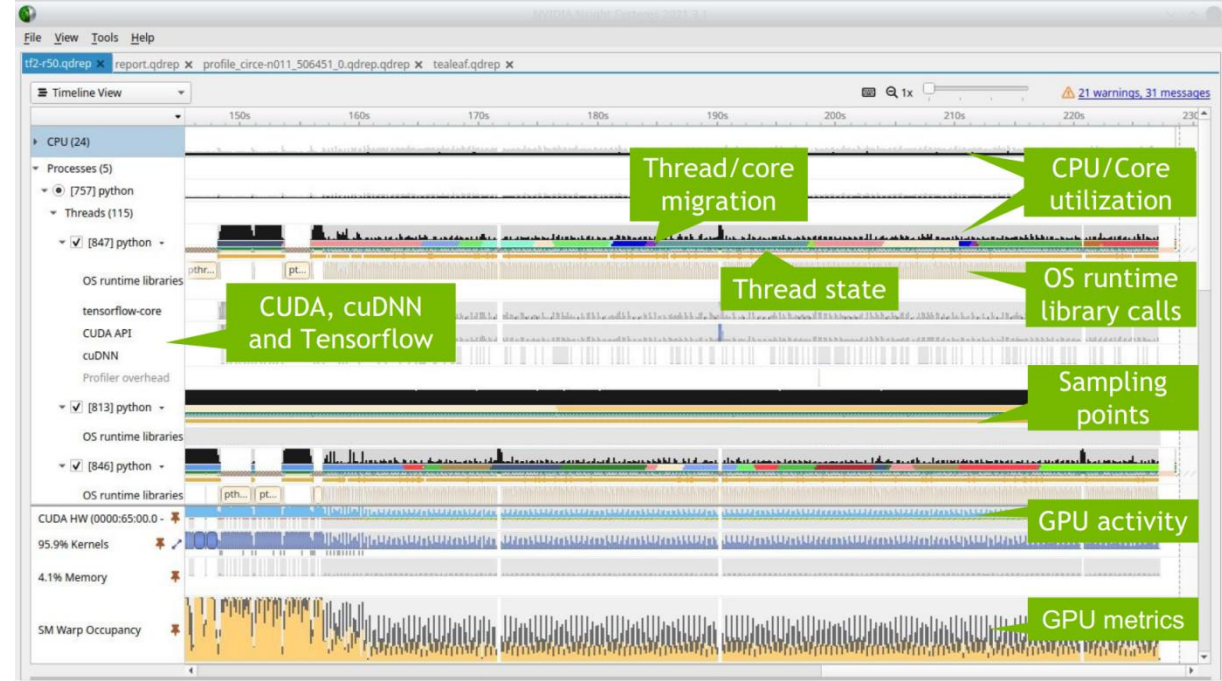
Nsight Compute

Standalone GUI+CLI

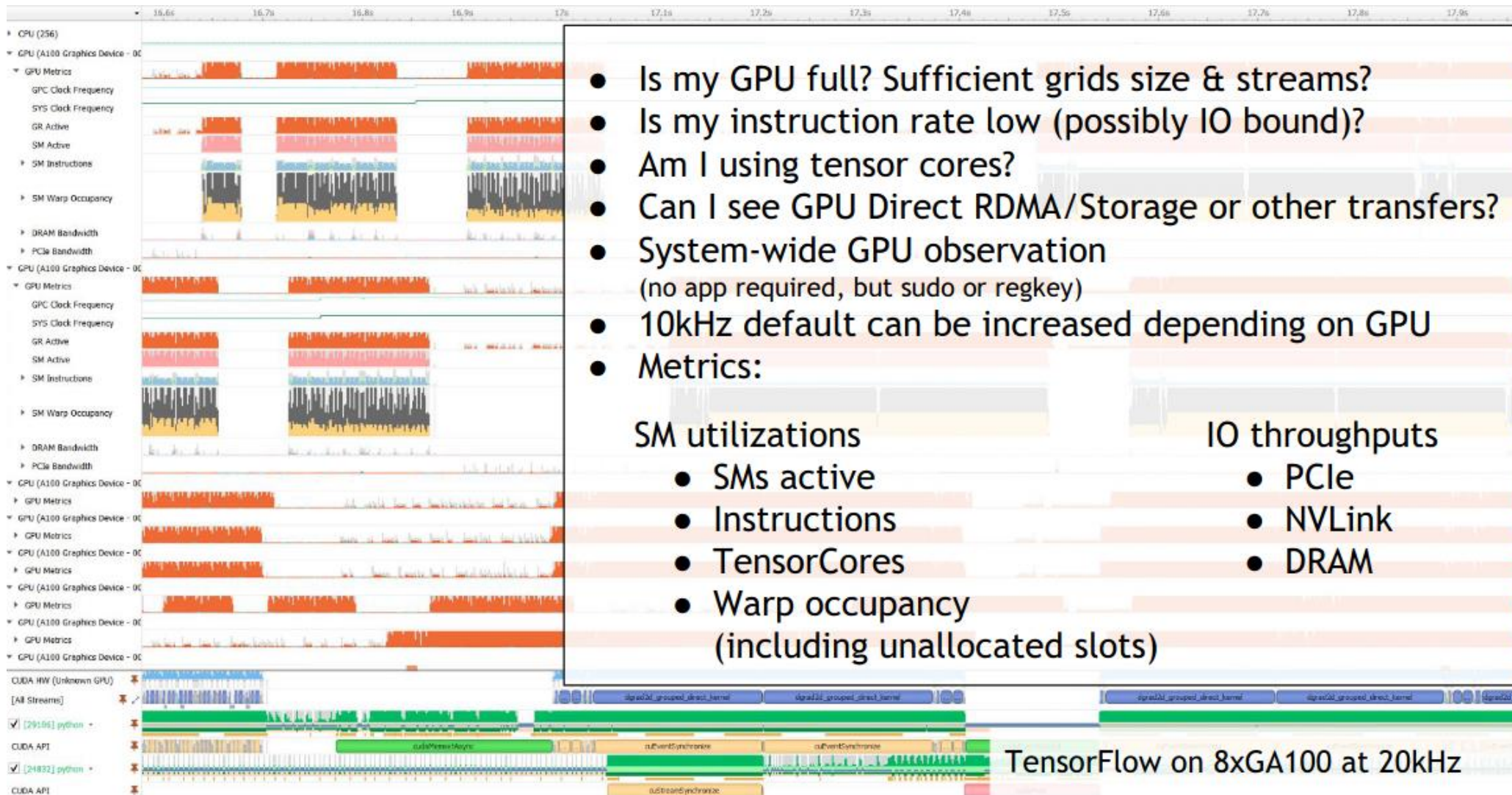
- GPU CUDA kernel analysis & debug
- Very high freq GPU perf counters
- Compare results (diff)
- Incredible statistics & customizable

NSIGHT SYSTEM

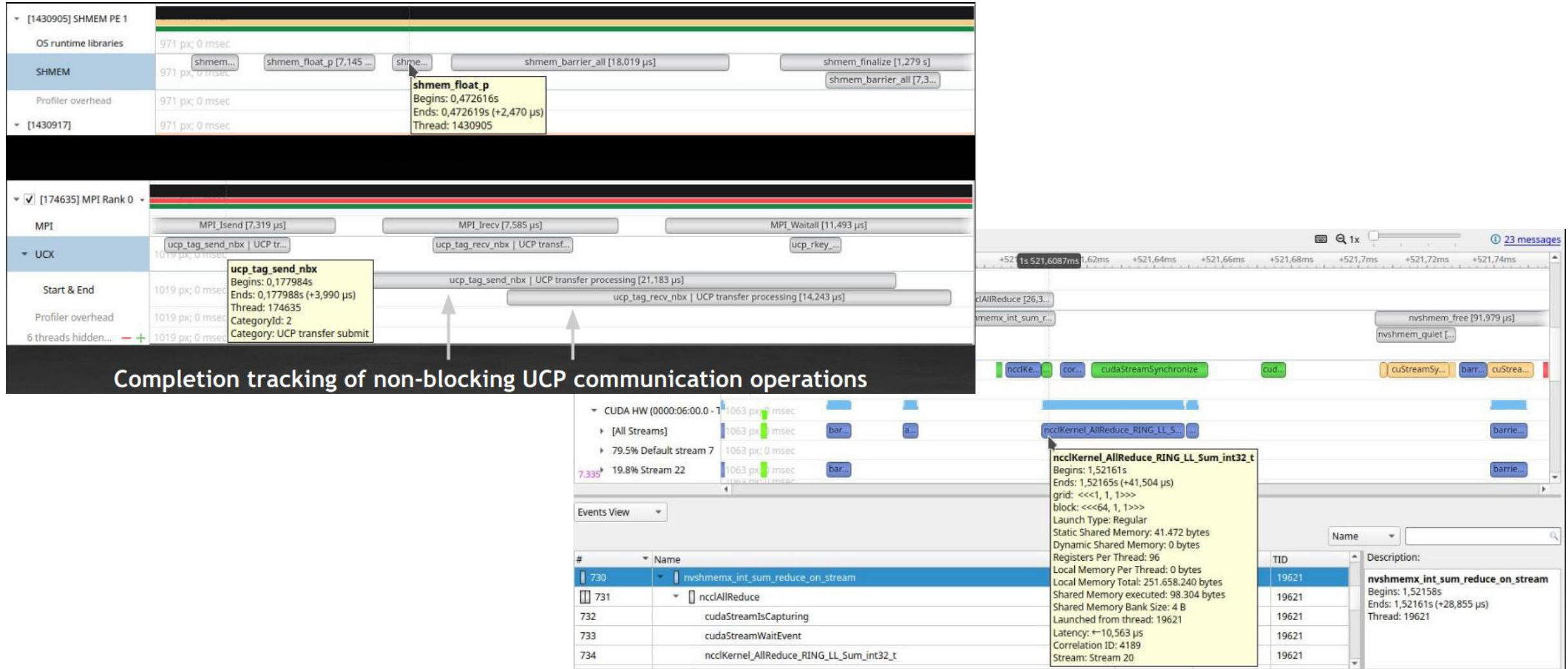
- System-wide application tuning
- Locate optimization opportunities
 - Visualize millions of events on a timeline
 - See gaps of unused CPU and GPU time
- Balance workloads across multiple CPUs and GPUs
 - CPU utilization and thread state
 - GPU streams, kernels, memory transfers, etc.
- Multi-platform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



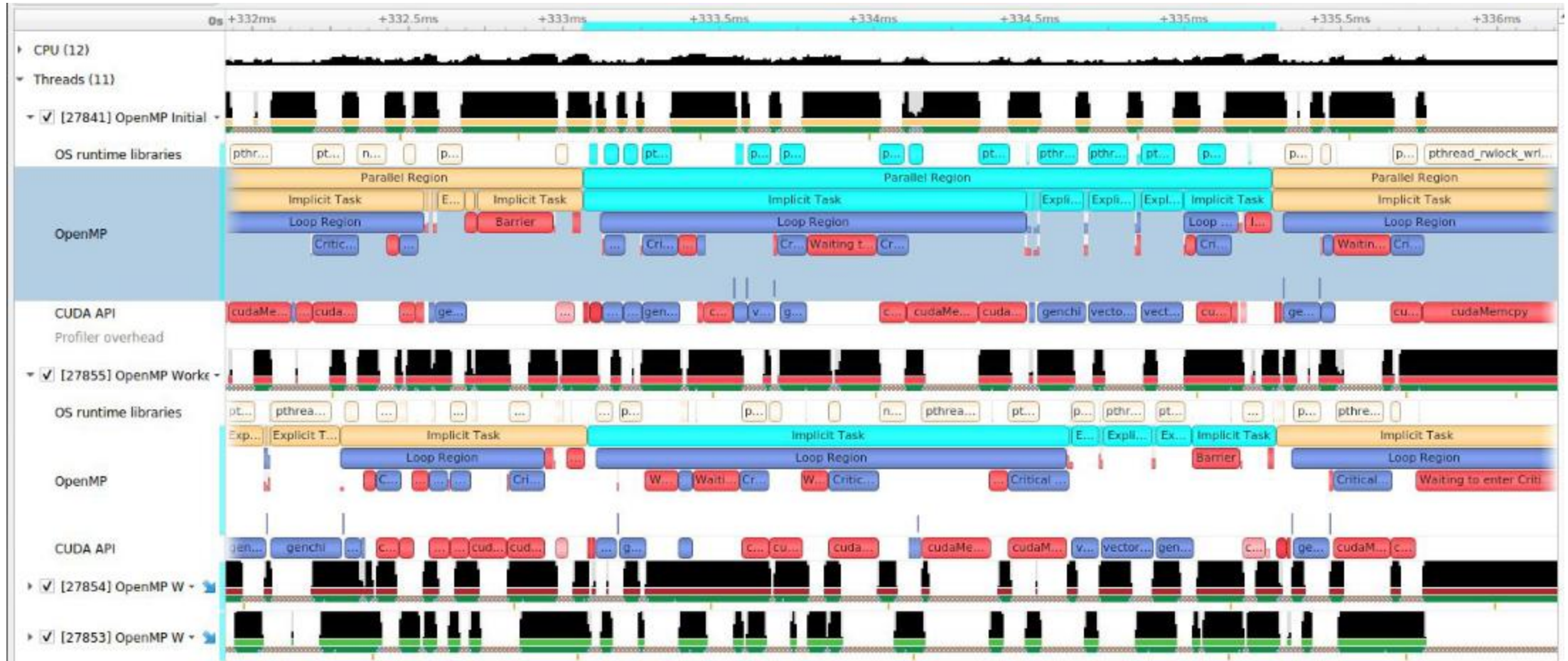
GPU METRIC SAMPLING



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



OPENMP



OMPT-capable OpenMP runtime required

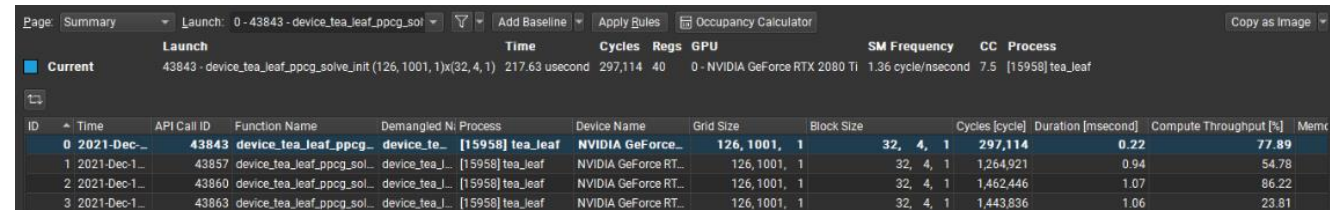
EXPERT SYSTEM

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

CUDA Async Memcpy with Pageable Memory	Duration	Start	Src Kind	Dst Kind	Bytes	PID	Device ID	Context ID	Stream ID	API Name
<p>The following APIs use PAGEABLE memory which causes asynchronous CUDA memcpy operations to block and be executed synchronously. This leads to low GPU utilization.</p> <p>Suggestion: If applicable, use PINNED memory instead.</p> <p>CLI command: <code>nsys analyze -r cuda-async-memcpy /mnt/data/traces/qdrep/ncl/profile_circe-n011_506451_0.sqlite</code></p>	2,048 µs	6,38792s	Device	Pageable	8 B	75475	0	1	7	cudaMemcpy
	2,048 µs	6,8334s	Device	Pageable	4 B	75475	0	1	7	cudaMemcpy
	2,016 µs	2,5394s	Device	Pageable	4 B	75475	0	1	7	cudaMemcpy
	2,016 µs	3,90617s	Device	Pageable	48 B	75475	0	1	7	cudaMemcpy
	2,016 µs	4,25257s	Device	Pageable	4 B	75475	0	1	7	cudaMemcpy
	2,016 µs	5,67617s	Device	Pageable	48 B	75475	0	1	7	cudaMemcpy
	2,016 µs	5,9572s	Device	Pageable	8 B	75475	0	1	7	cudaMemcpy
	2,016 µs	5,97088s	Device	Pageable	4 B	75475	0	1	7	cudaMemcpy

NSIGHT COMPUTE

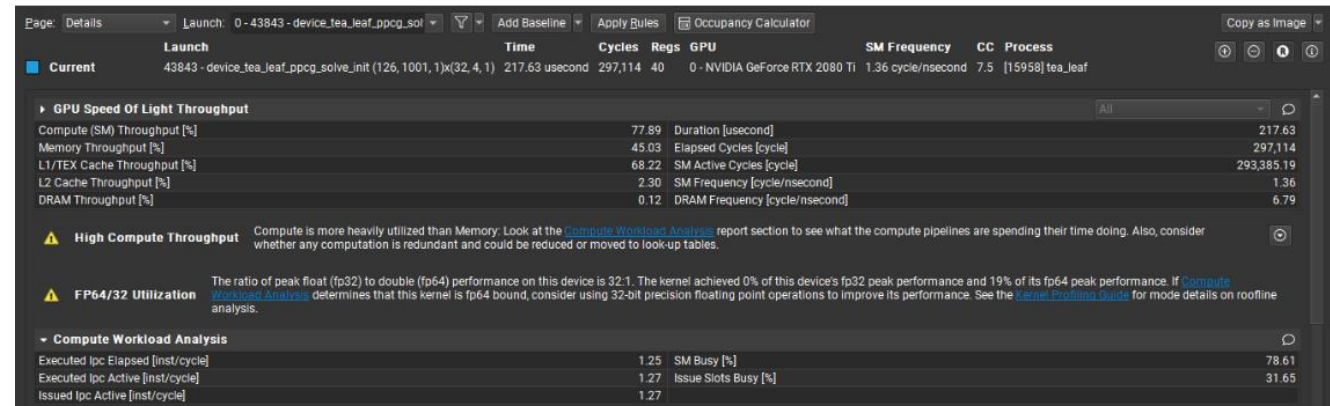
- Interactive CUDA kernel profiler
- Targeted metric sections for various performance aspects
- Customizable data collection and presentation (tables, charts, ...)
- GUI and CLI
- Python-based API for guided analysis and post-processing
- Support for remote profiling across machines and platforms



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

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

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; a warning box 'High Compute Throughput'; a warning box 'FP64/32 Utilization'; and an expandable section 'Compute Workload Analysis' (highlighted with a green box) containing a table of IPC metrics. A green vertical line on the right side of the interface points to the 'Metric values' label.

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	Unit
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	

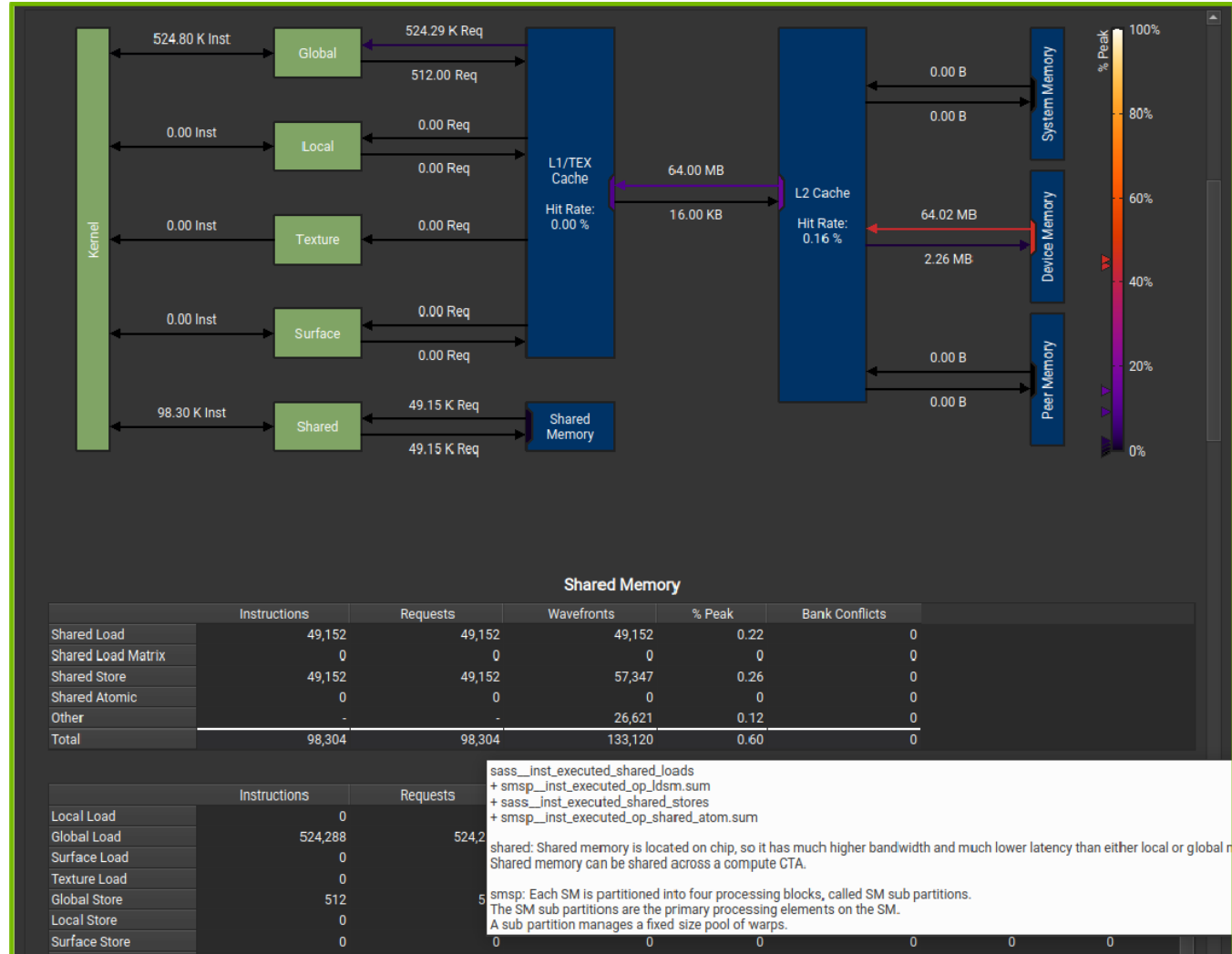
Metric	Value	Unit
Executed Ipc Elapsed [inst/cycle]	1.25	
Executed Ipc Active [inst/cycle]	1.27	
Issued Ipc Active [inst/cycle]	1.27	
SM Busy [%]	78.61	
Issue Slots Busy [%]	31.65	

Expandable Sections

Expert Analysis (Rules)

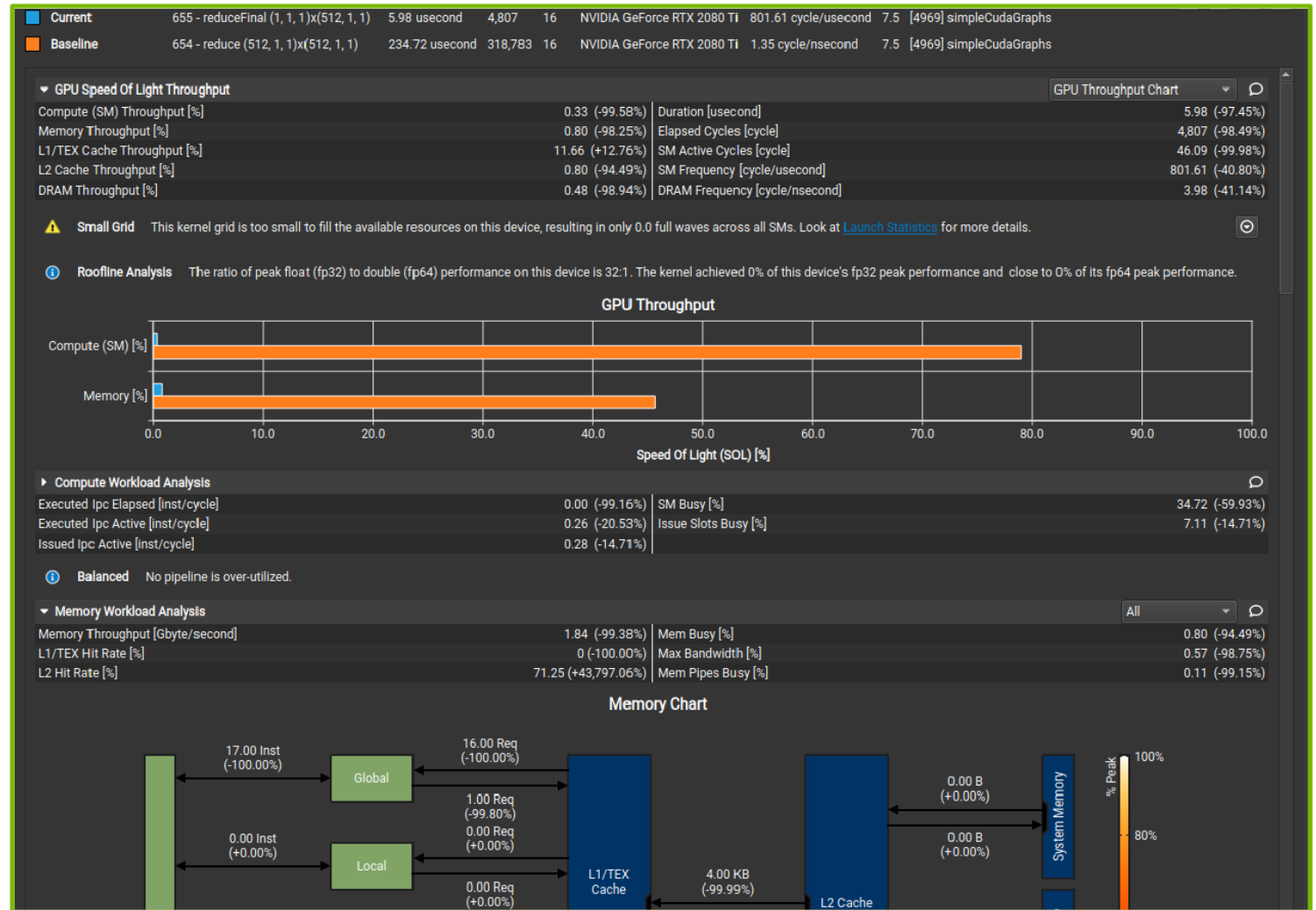
DATA TRANSFER ANALYSIS

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



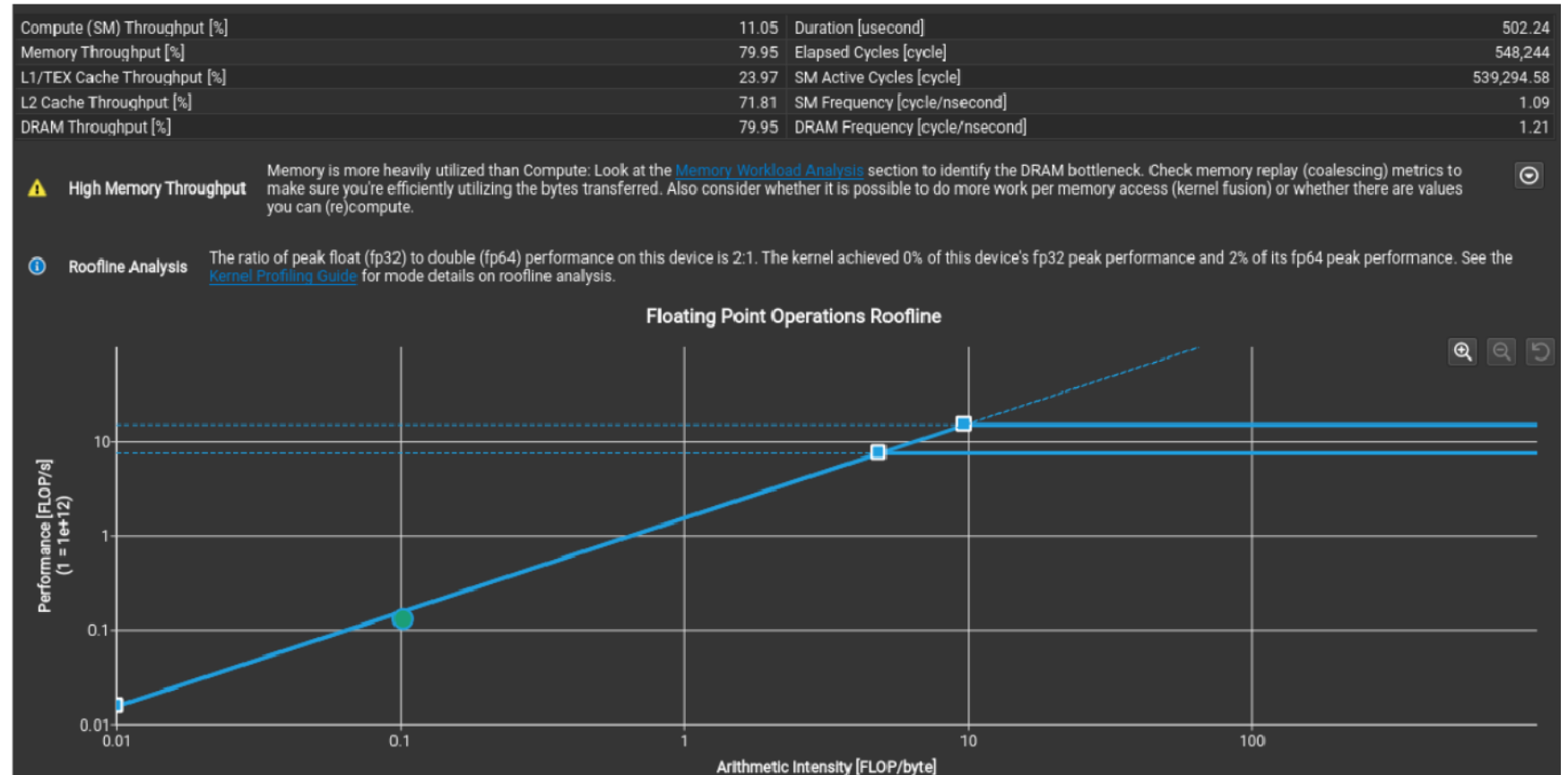
BASELINE COMPARISON

- 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

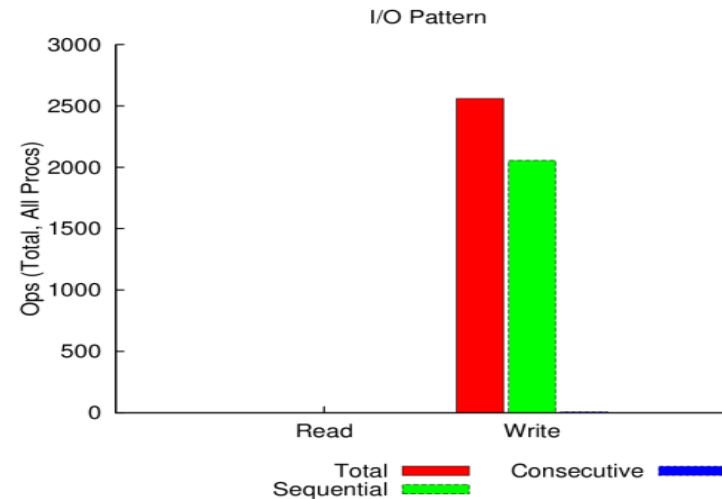
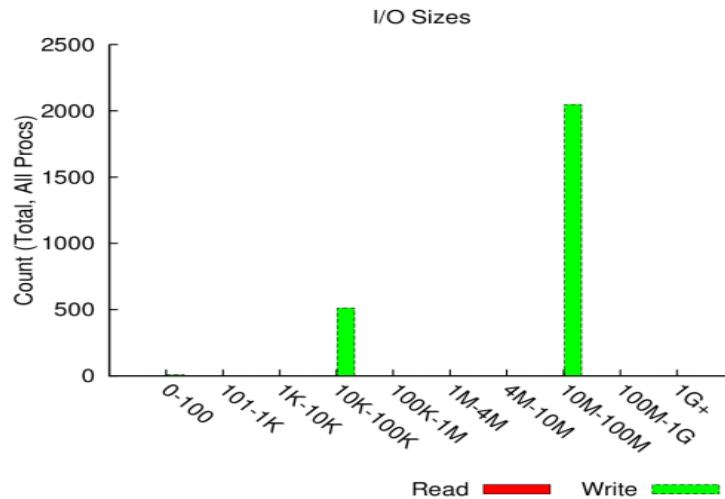
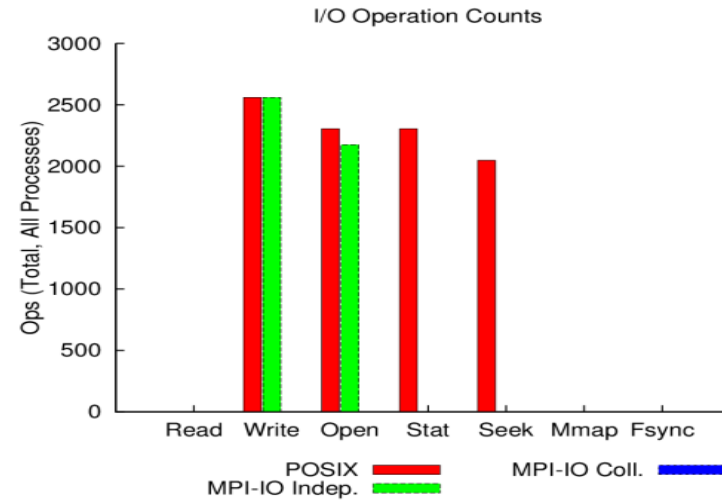
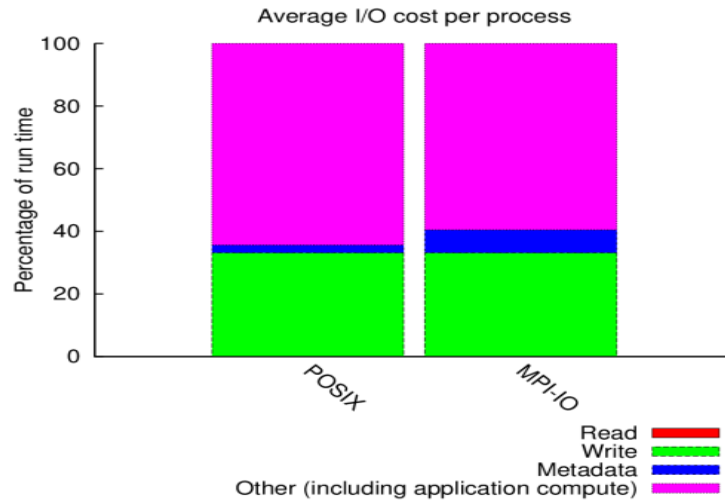
- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

- 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



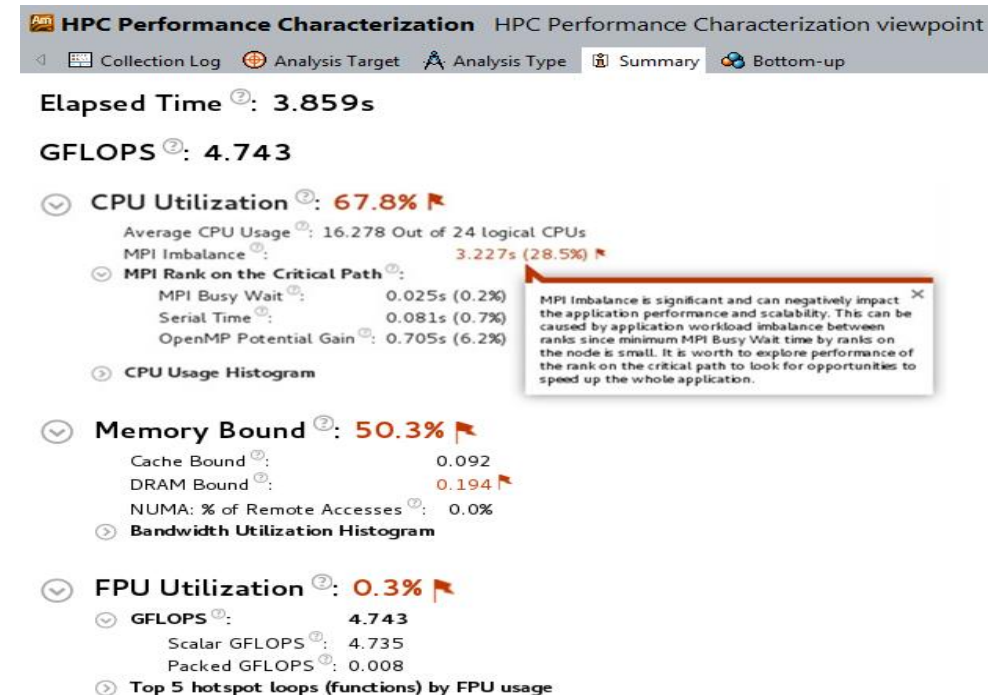
VTUNE AMPLIFIER XE

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

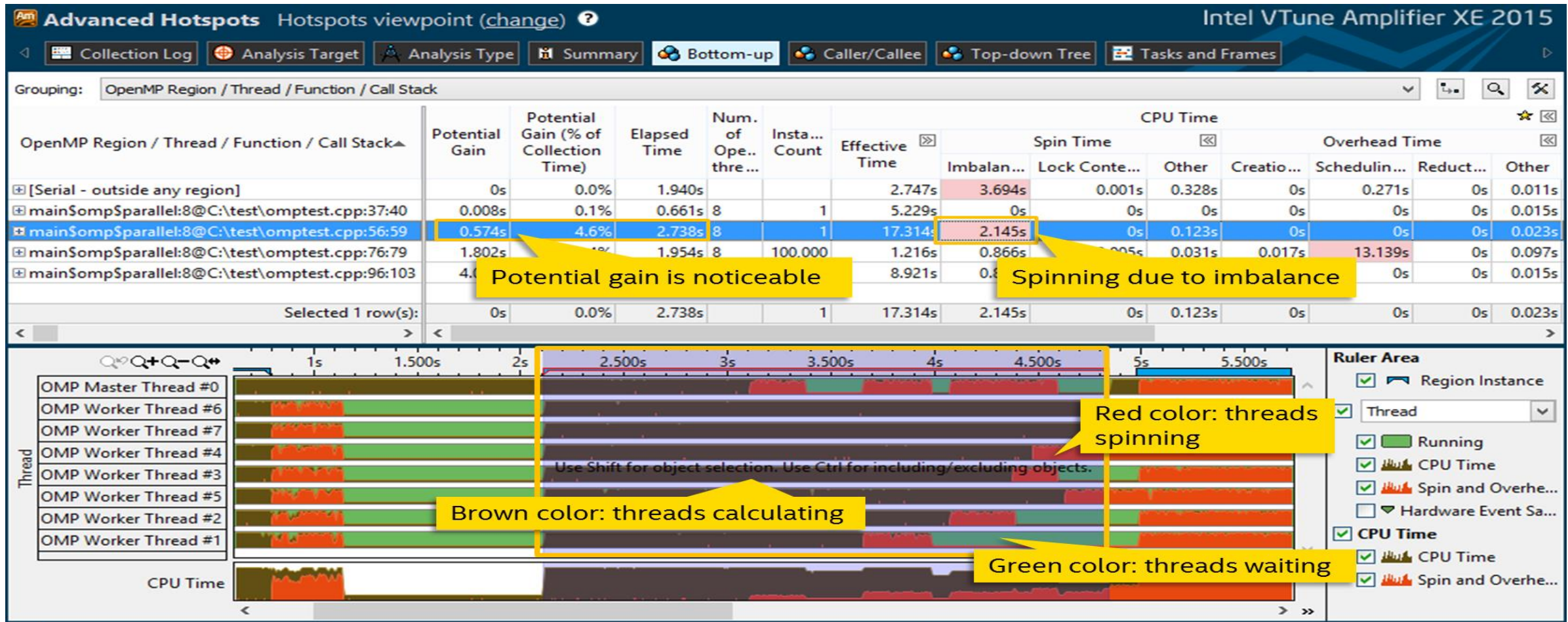
- Lock and Wait analysis for OpenMP and TBB
- HPC analysis for quick overview

- 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

The screenshot displays the Intel Advisor XE 2016 interface. At the top, the title bar reads "Where should I add vectorization and/or threading parallelism?". Below this, there are navigation tabs for "Summary", "Survey Report", "Refinement Reports", "Annotation Report", and "Suitability Report". The main area shows a table of function call sites and loops. The table has columns for "Function Call Sites and Loops", "Self Time", "Total Time", "Loop Type", "Why No Vectorization?", "Vector... Loops", "Instruction Set Analysis", and "Optimization". A yellow box highlights the "Filters" section, which includes "Program time: 26.54s", "Vectorized", "Not Vectorized", and "FILTER: All Modules All Sources". Another yellow box highlights a row in the table, with a callout that says "Loop may have several parts or versions". A third yellow box highlights the "Why No Vectorization?" column, with a callout that says "Vectorization and compiler optimization details". A fourth yellow box highlights the "Recommendations" tab, with a callout that says "Recommendations for optimization". The "Recommendations" tab shows an issue: "Issue: Ineffective peeled/remainder loop(s) present". The issue description is: "All or some **source loop** iterations are not executing in the **loop body**. Improve performance by moving source loop iterations from **peeled/remainder** loops to the loop body." The recommendation is: "Add data padding". The potential performance gain is "Information not available". The confidence is "Information not available until Beta Update release". The recommendation text is: "The **trip count** is not a multiple of **vector length**. To fix: Do one of the following:". The list of recommendations is: "Increase size of objects and add iterations so the trip count is a multiple of vector length." and "Increase the size of static and automatic objects, and use a compiler option to add data padding."

Function Call Sites and Loops	Self Time	Total Time	Loop Type	Why No Vectorization?	Vector... Loops	Instruction Set Analysis	Optimization
[loop at loopstl.cpp:3962 in s...	0.125s	0.125s	Expand	Expand	AVX	Inserts; Maske...	Unrolled
[loop at loopstl.cpp:6186 in ...	0.125s	0.125s	Collapse	Collapse	AVX		Unrolled
i> [loop at loopstl.cpp:6186 in s...	0.062s	0.062s	Remainder				
i> [loop at loopstl.cpp:...			Vectorized (Body)		AVX		Unrolled
i, [loop at loopstl.cpp:5625 in ...			Scalar	loop with early exits cannot be ...			

Issue: Ineffective peeled/remainder loop(s) present

All or some **source loop** iterations are not executing in the **loop body**. Improve performance by moving source loop iterations from **peeled/remainder** loops to the loop body.

➤ Add data padding

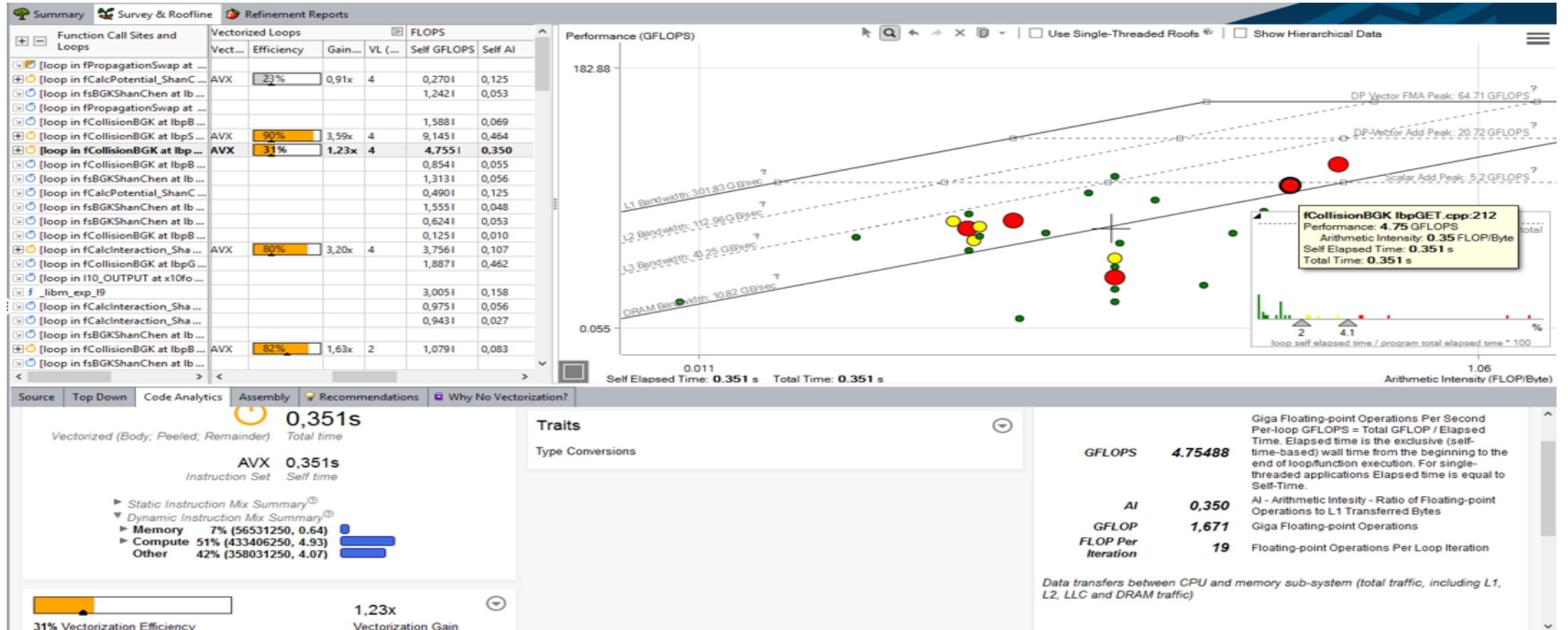
Potential performance gain: Information not available

Confidence this recommendation applies to your code: Information not available until Beta Update release

The **trip count** is not a multiple of **vector length**. To fix: Do one of the following:

- Increase size of objects and add iterations so the trip count is a multiple of vector length.
- Increase the size of static and automatic objects, and use a compiler option to add data padding.

INTEL ADVISOR – ROOFLINE



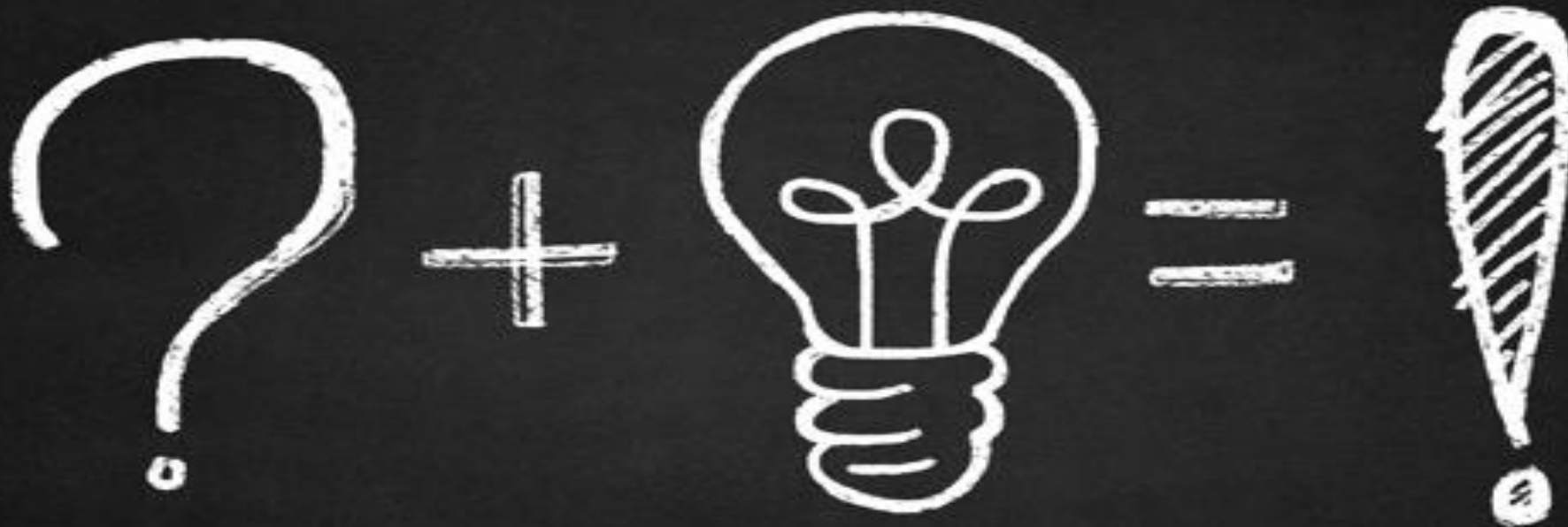
PERFORMANCE ANALYSIS RECOMMENDATIONS

- 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)
 - 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