HPC SOFTWARE – DEBUGGER AND PERFORMANCE ANALYSIS TOOLS

NOV 12, 2025 I MICHAEL KNOBLOCH I M.KNOBLOCH@FZ-JUELICH.DE

OUTLINE

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

Kent Beck

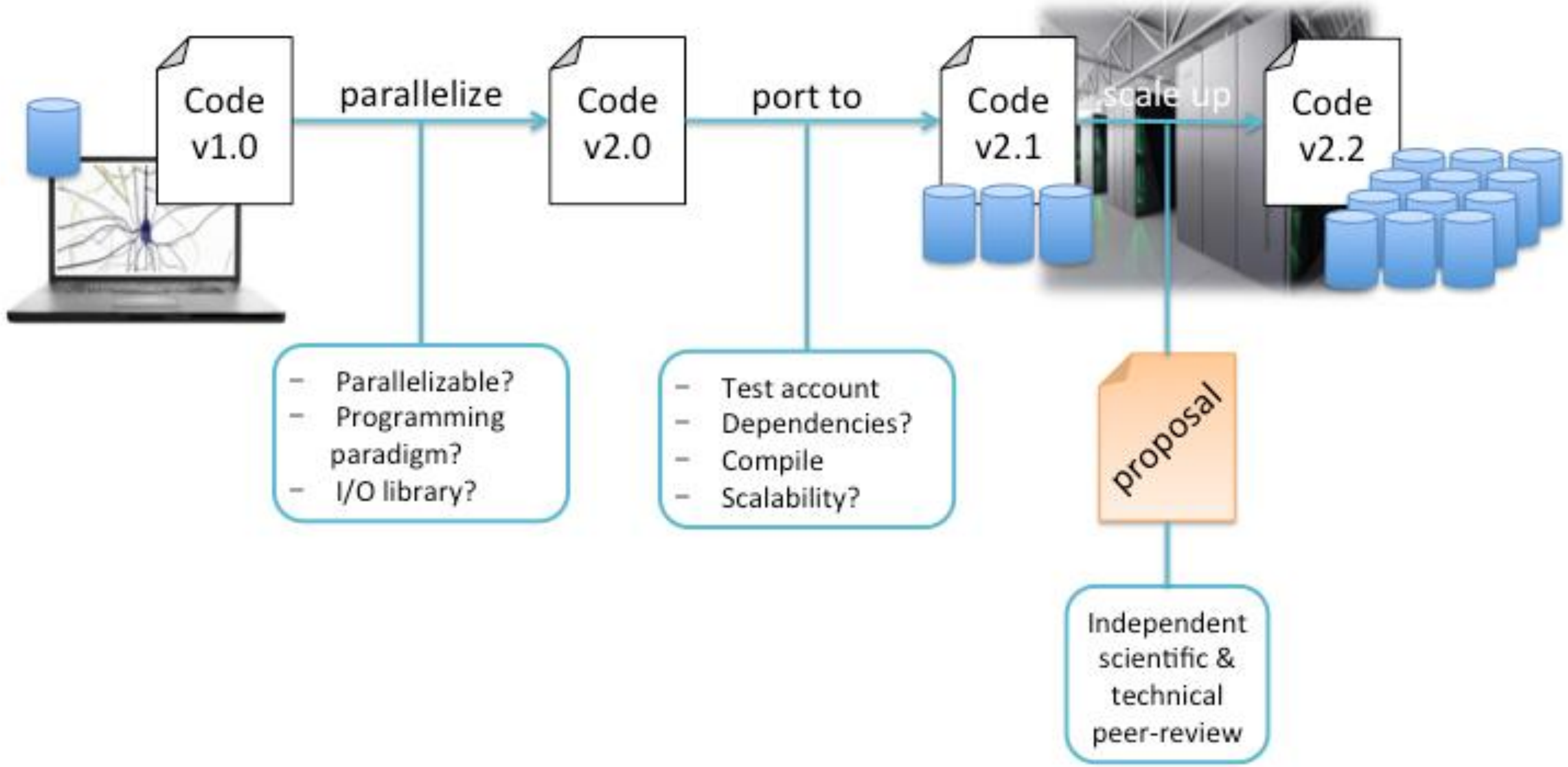
- Local module setup
- Compilers
- Libraries

Debugger and Correctness Tools

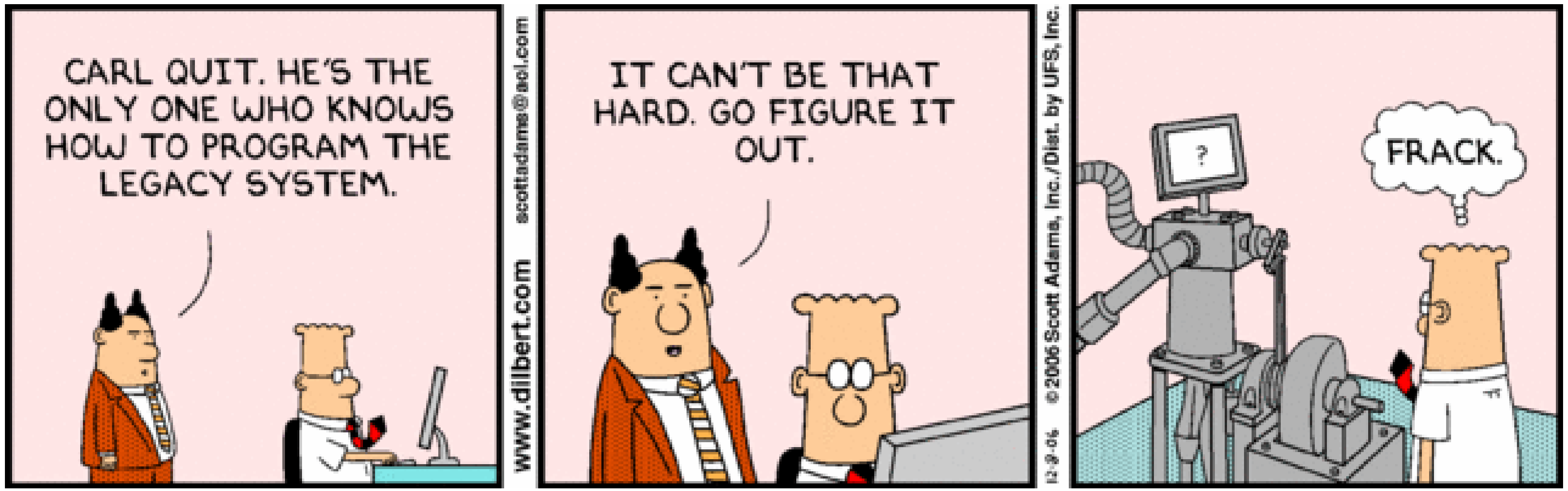
Performance Analysis Tools

WHY SHOULD YOU CARE ABOUT TOOLS?

NEW APPLICATION?



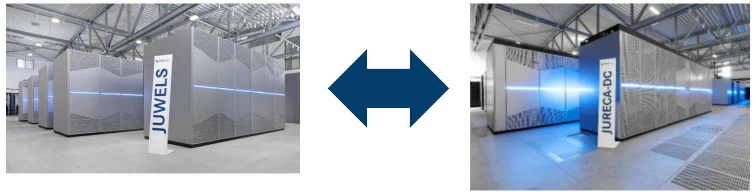
WORKING WITH LEGACY CODES?


CARL QUIT. HE'S THE ONLY ONE WHO KNOWS HOW TO PROGRAM THE LEGACY SYSTEM.
IT CAN'T BE THAT HARD. GO FIGURE IT OUT.
?
FRACK.
www.dilbert.com scottadams@aol.com
12-9-06 © 2006 Scott Adams, Inc./Dist. by UFS, Inc.

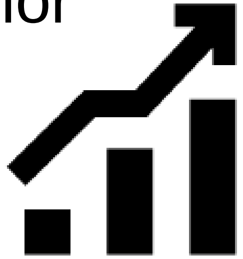
VETERAN HPC USER, BUT NEW TO JSC?

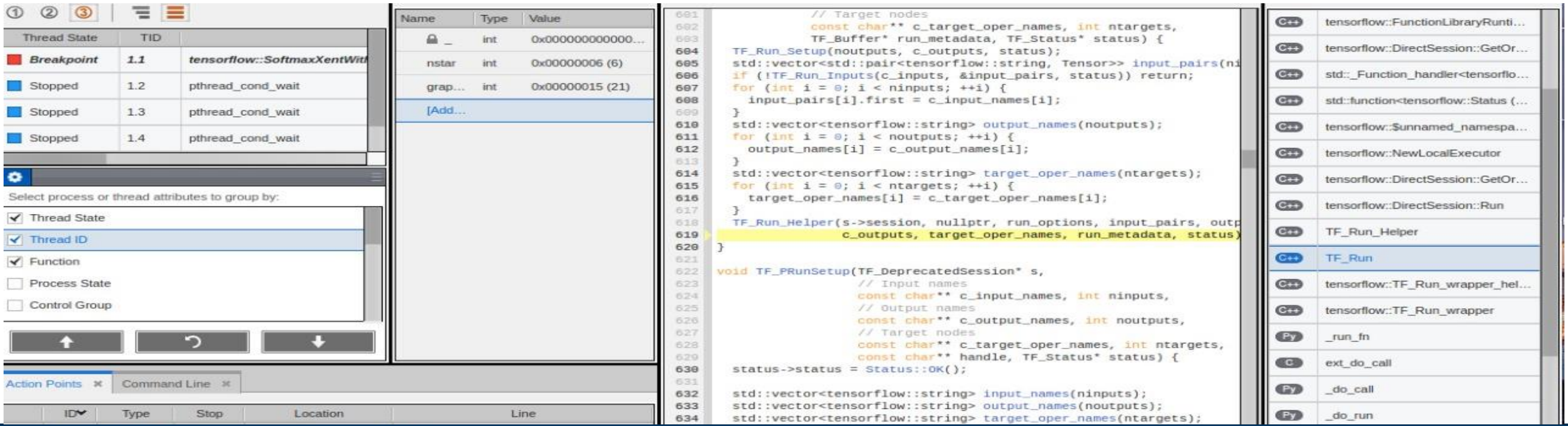
- Assess performance on a JSC machine

- Compare behavior on different machines



- Investigate scaling behavior





Thread State	TID	
🟥 ***Breakpoint***	***1.1***	***tensorflow::SoftmaxXentWit***
🟦 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

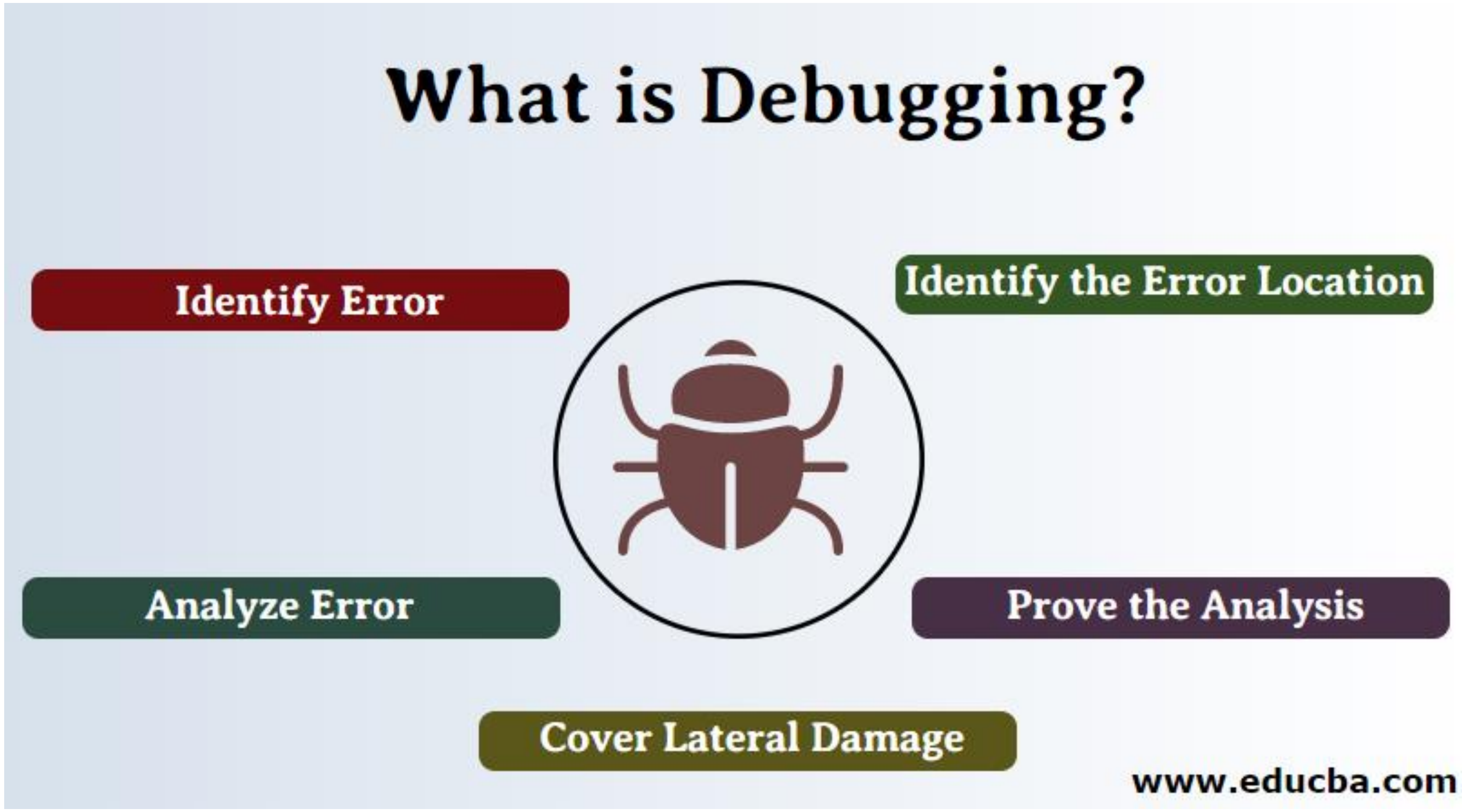
Name	Type	Value
_	int	0x000000000000…
nstar	int	0x00000006 (6)
grap…	int	0x00000015 (21)
[Add…		

```
                    // Target nodes
                    const char** c_target_oper_names, int ntargets,
                    TF_Buffer* run_metadata, TF_Status* status) {
  TF_Run_Setup(noutputs, c_outputs, status);
  std::vector<std::pair<tensorflow::string, Tensor>> input_pairs(ni
  if (!TF_Run_Inputs(c_inputs, &input_pairs, status)) return;
  for (int i = 0; i < ninputs; ++i) {
    input_pairs[i].first = c_input_names[i];
  }
  std::vector<tensorflow::string> output_names(noutputs);
  for (int i = 0; i < noutputs; ++i) {
    output_names[i] = c_output_names[i];
  }
  std::vector<tensorflow::string> target_oper_names(ntargets);
  for (int i = 0; i < ntargets; ++i) {
    target_oper_names[i] = c_target_oper_names[i];
  }
  TF_Run_Helper(s->session, nullptr, run_options, input_pairs, outp
                c_outputs, target_oper_names, run_metadata, status)
}

void TF_PRunSetup(TF_DeprecatedSession* s,
                  // Input names
                  const char** c_input_names, int ninputs,
                  // Output names
                  const char** c_output_names, int noutputs,
                  // Target nodes
                  const char** c_target_oper_names, int ntargets,
                  const char** handle, TF_Status* status) {
  status->status = Status::OK();

  std::vector<tensorflow::string> input_names(ninputs);
  std::vector<tensorflow::string> output_names(noutputs);
  std::vector<tensorflow::string> target_oper_names(ntargets);
```

C++	tensorflow::FunctionLibraryRunti…
C++	tensorflow::DirectSession::GetOr…
C++	std::_Function_handler<tensorflo…
C++	std::function<tensorflow::Status (…
C++	tensorflow::$unnamed_namespa…
C++	tensorflow::NewLocalExecutor
C++	tensorflow::DirectSession::GetOr…
C++	tensorflow::DirectSession::Run
C++	TF_Run_Helper
C++	TF_Run
C++	tensorflow::TF_Run_wrapper_hel…
C++	tensorflow::TF_Run_wrapper
Py	_run_fn
C	ext_do_call
Py	_do_call
Py	_do_run

DEBUGGER & CORRECTNESS TOOLS

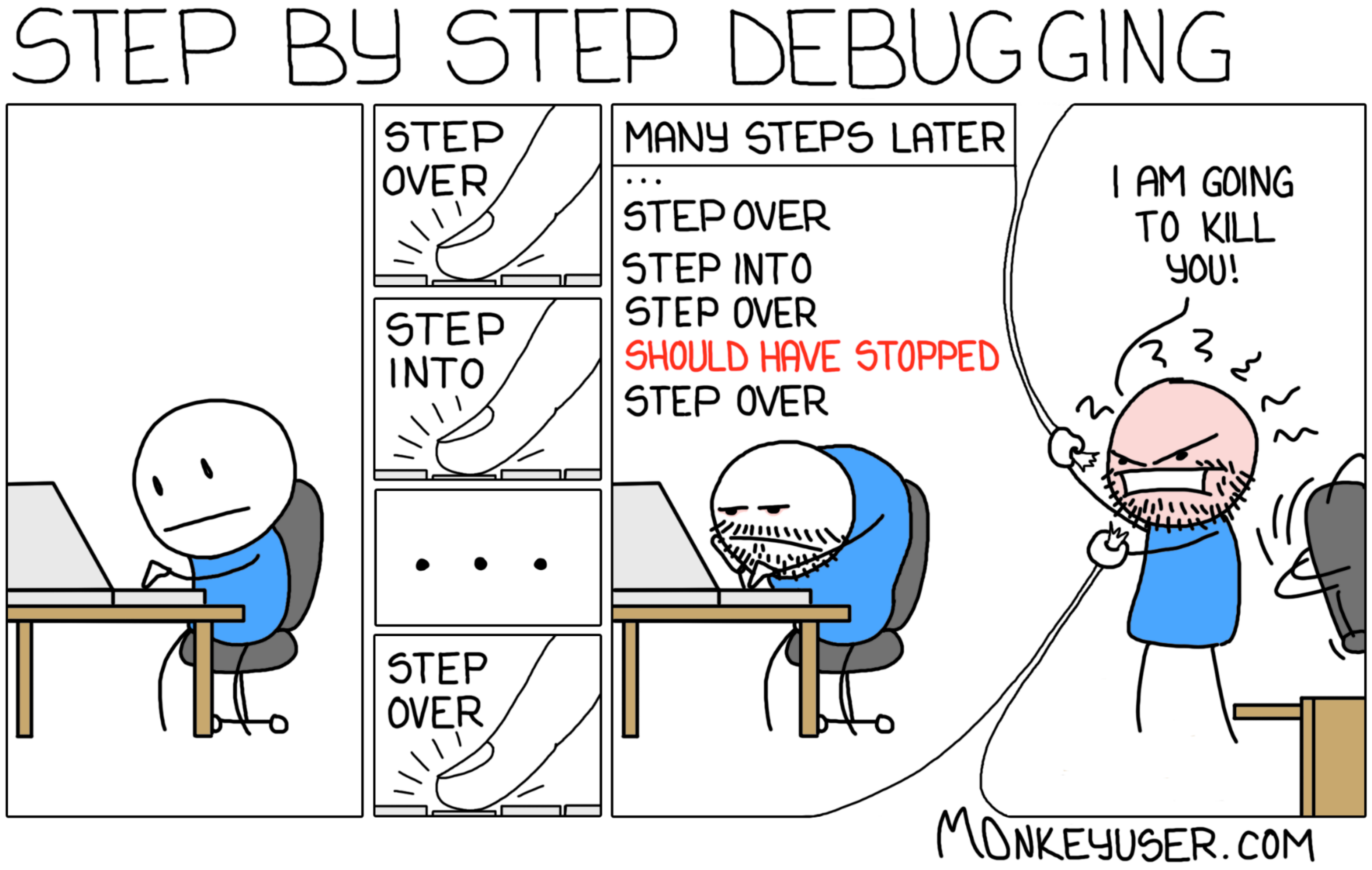
WHAT IS DEBUGGING?



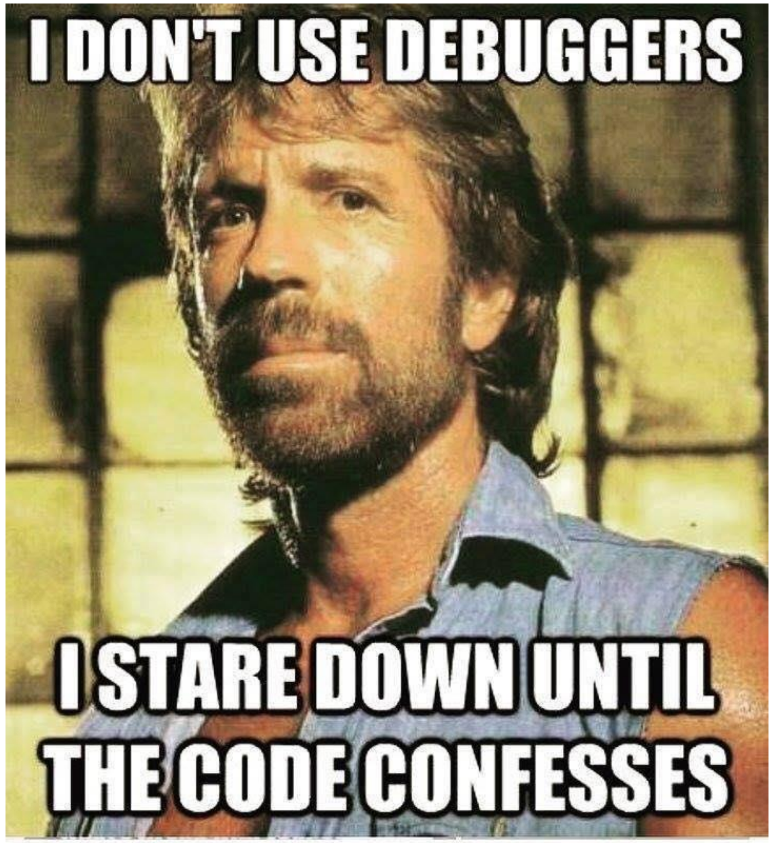
ATTENTION: DEBUGGING CAN BE TIME CONSUMING

99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.
127 little bugs in the code...

ATTENTION: DEBUGGING CAN BE FRUSTRATING



BUT: DON‘T BE THESE GUYS

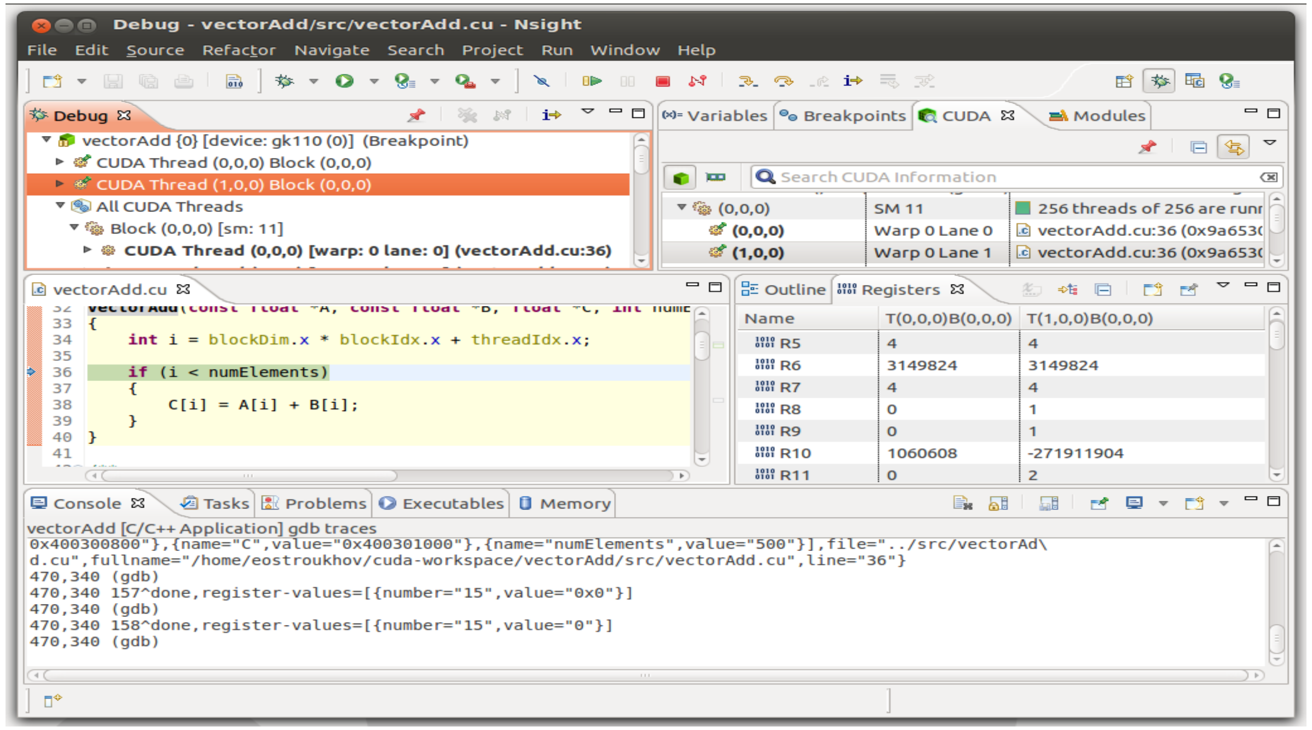


DEBUGGING TOOLS (STATUS: NOV 2025)

- **Debugger:**
  - (GDB)
  - CUDA-GDB
  - TotalView
  - LinaroForge - DDT
- **Correctness and Memory Analyzer:**
  - (Valgrind)
  - MUST
  - Archer
  - CUDA Compute Sanitizer

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



vectorAdd {0} [device: gk110 (0)] (Breakpoint)
CUDA Thread (0,0,0) Block (0,0,0)
CUDA Thread (1,0,0) Block (0,0,0)
All CUDA Threads
Block (0,0,0) [sm: 11]
CUDA Thread (0,0,0) [warp: 0 lane: 0] (vectorAdd.cu:36)

(0,0,0)	SM 11	🟩 256 threads of 256 are runn
(0,0,0)	Warp 0 Lane 0	vectorAdd.cu:36 (0x9a653(
(1,0,0)	Warp 0 Lane 1	vectorAdd.cu:36 (0x9a653(

```
vectorAdd(const float *A, const float *B, float *C, int nume
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

Name	T(0,0,0)B(0,0,0)	T(1,0,0)B(0,0,0)
R5	4	4
R6	3149824	3149824
R7	4	4
R8	0	1
R9	0	1
R10	1060608	-271911904
R11	0	2

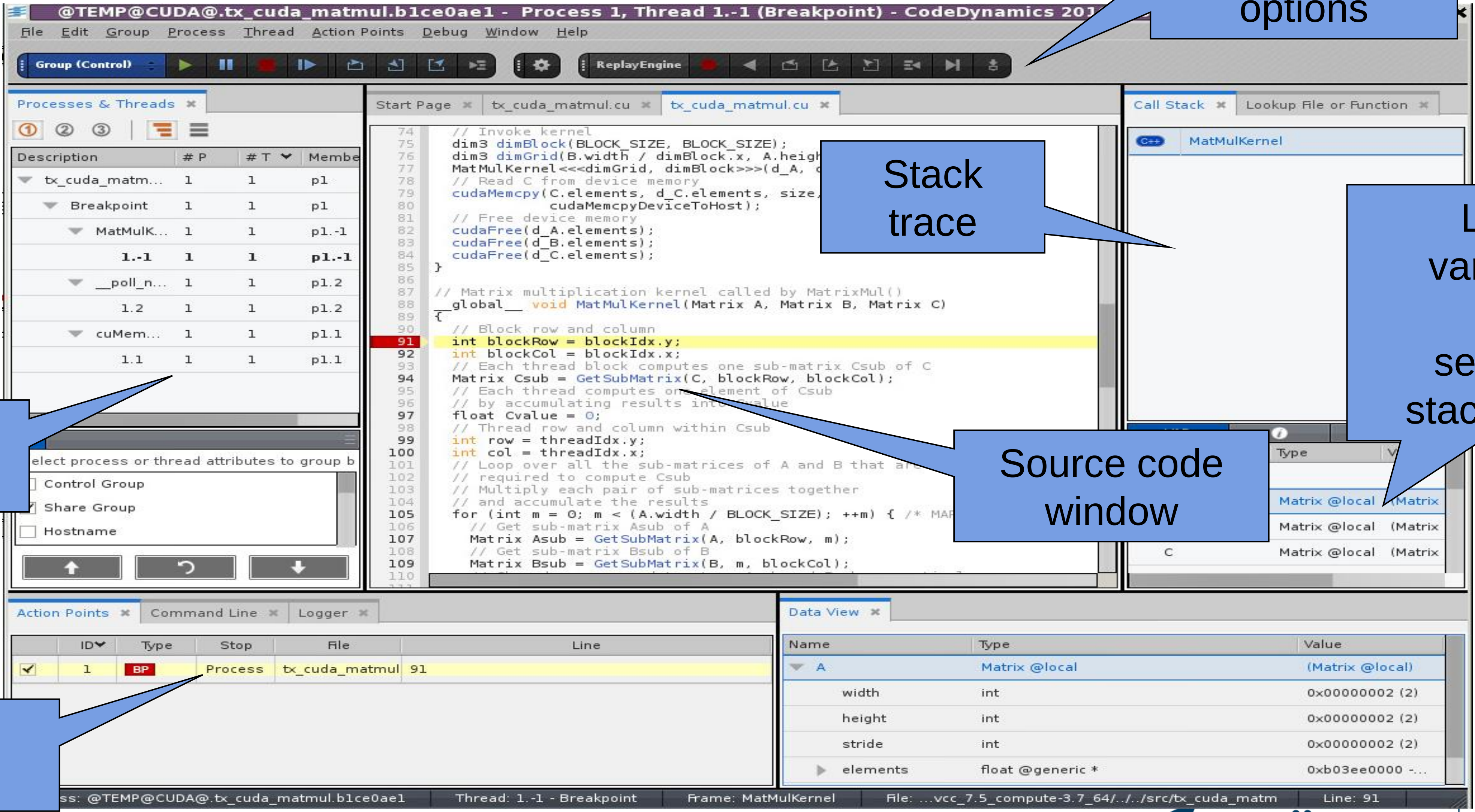
vectorAdd [C/C++ Application] gdb traces

```
0x400300800"},{name="C",value="0x400301000"},{name="numElements",value="500"}],file="../src/vectorAd\
d.cu",fullname="/home/eostroukhov/cuda-workspace/vectorAdd/src/vectorAdd.cu",line="36"}
470,340 (gdb)
470,340 157^done,register-values=[{number="15",value="0x0"}]
470,340 (gdb)
470,340 158^done,register-values=[{number="15",value="0"}]
470,340 (gdb)
```

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



@TEMP@CUDA@.tx_cuda_matmul.b1ce0ae1 - Process 1, Thread 1.-1 (Breakpoint) - CodeDynamics 201

Description	# P	# T	Membe
tx_cuda_matm...	1	1	p1
Breakpoint	1	1	p1
MatMulK...	1	1	p1.-1
1.-1	**1**	**1**	**p1.-1**
__poll_n...	1	1	p1.2
1.2	1	1	p1.2
cuMem...	1	1	p1.1
1.1	1	1	p1.1

elect process or thread attributes to group b

- Control Group
- Share Group
- Hostname

```
  // Invoke kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(B.width / dimBlock.x, A.heigh
  MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d
  // Read C from device memory
  cudaMemcpy(C.elements, d_C.elements, size,
             cudaMemcpyDeviceToHost);
  // Free device memory
  cudaFree(d_A.elements);
  cudaFree(d_B.elements);
  cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;
  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;
  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;
  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) { /* MAF
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
```

	Type	V
	Matrix @local	(Matrix
	Matrix @local	(Matrix
C	Matrix @local	(Matrix

	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 -...

ss: @TEMP@CUDA@.tx_cuda_matmul.b1ce0ae1 Thread: 1.-1 - Breakpoint Frame: MatMulKernel File: ...vcc_7.5_compute-3.7_64/../../src/tx_cuda_matm Line: 91

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-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://linaroforge.com/linaroDdt
- NOTE**:** JSC license limited to 128 processes (shared between all users)

DDT: MAIN WINDOW

Process controls

CUDA Thread stepping

Variables

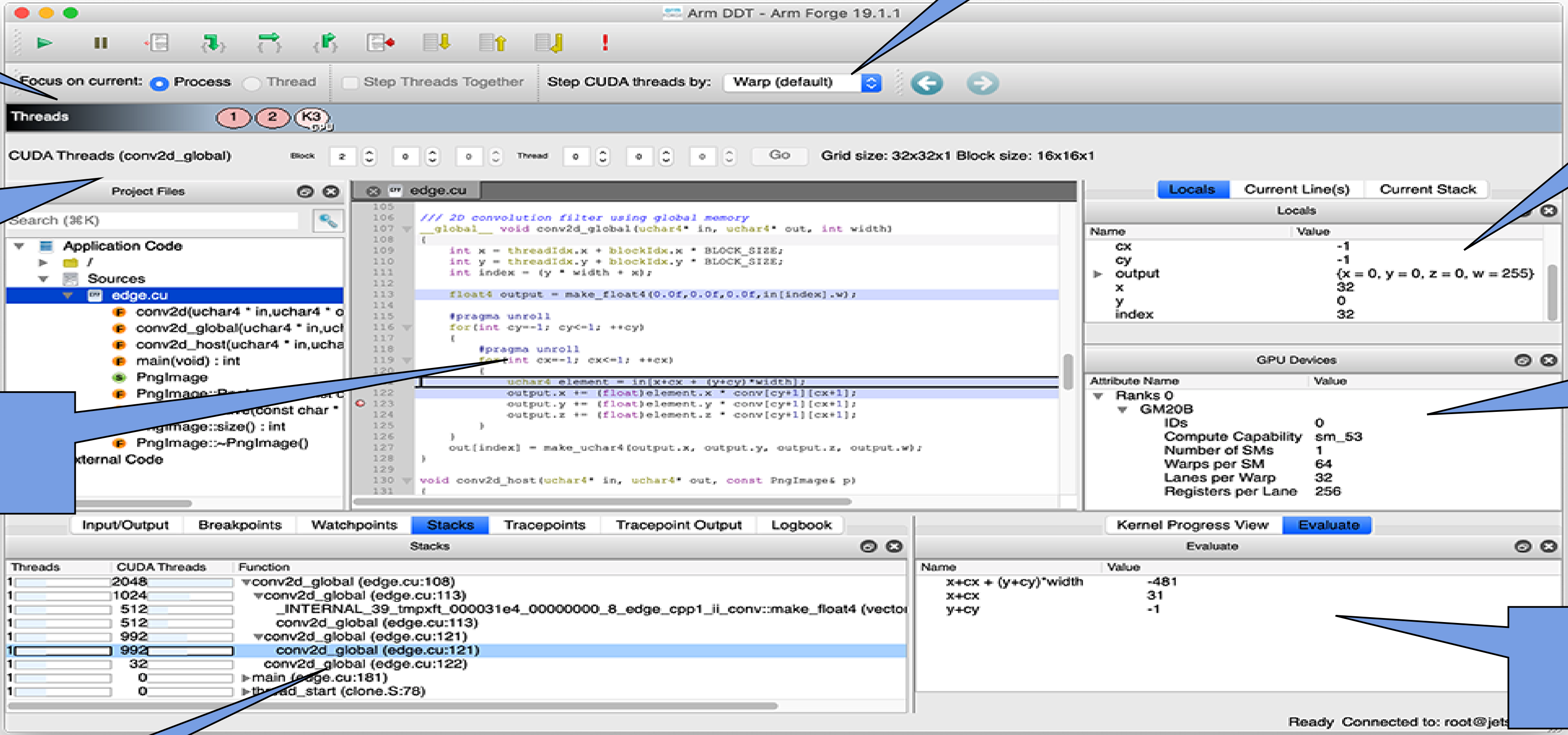
CUDA Thread control

GPU Device information

Source code

Expression evaluator

Stack trace


Arm DDT - Arm Forge 19.1.1
Focus on current: Process Thread
Step Threads Together
Step CUDA threads by: Warp (default)
Threads 1 2 K3
CUDA Threads (conv2d_global)
Block 2 0 0 Thread 0 0 0 Go
Grid size: 32x32x1 Block size: 16x16x1
Project Files
Search (⌘K)
Application Code
/
Sources
edge.cu
conv2d(uchar4 * in,uchar4 * o
conv2d_global(uchar4 * in,ucl
conv2d_host(uchar4 * in,ucha
main(void) : int
PngImage
PngImage::PngImage(const char *
PngImage::size() : int
PngImage::~PngImage()
External Code
Locals
Current Line(s)
Current Stack
GPU Devices
Input/Output
Breakpoints
Watchpoints
Stacks
Tracepoints
Tracepoint Output
Logbook
Kernel Progress View
Evaluate
Ready Connected to: root@jets

```
/// 2D convolution filter using global memory
__global__ void conv2d_global(uchar4* in, uchar4* out, int width)
{
    int x = threadIdx.x + blockIdx.x * BLOCK_SIZE;
    int y = threadIdx.y + blockIdx.y * BLOCK_SIZE;
    int index = (y * width + x);

    float4 output = make_float4(0.0f,0.0f,0.0f,in[index].w);

    #pragma unroll
    for(int cy=-1; cy<=1; ++cy)
    {
        #pragma unroll
        for(int cx=-1; cx<=1; ++cx)
        {
            uchar4 element = in[x+cx + (y+cy)*width];
            output.x += (float)element.x * conv[cy+1][cx+1];
            output.y += (float)element.y * conv[cy+1][cx+1];
            output.z += (float)element.z * conv[cy+1][cx+1];
        }
    }
    out[index] = make_uchar4(output.x, output.y, output.z, output.w);
}

void conv2d_host(uchar4* in, uchar4* out, const PngImage& p)
{
```

Locals

Name	Value
cx	-1
cy	-1
output	{x = 0, y = 0, z = 0, w = 255}
x	32
y	0
index	32

GPU Devices

Attribute Name	Value
Ranks 0	
GM20B	
IDs	0
Compute Capability	sm_53
Number of SMs	1
Warps per SM	64
Lanes per Warp	32
Registers per Lane	256

Stacks

Threads	CUDA Threads	Function
1	2048	conv2d_global (edge.cu:108)
1	1024	conv2d_global (edge.cu:113)
1	512	_INTERNAL_39_tmpxft_000031e4_00000000_8_edge_cpp1_ii_conv::make_float4 (vecto
1	512	conv2d_global (edge.cu:113)
1	992	conv2d_global (edge.cu:121)
1	992	conv2d_global (edge.cu:121)
1	32	conv2d_global (edge.cu:122)
1	0	main (edge.cu:181)
1	0	thread_start (clone.S:78)

Evaluate

Name	Value
x+cx + (y+cy)*width	-481
x+cx	31
y+cy	-1

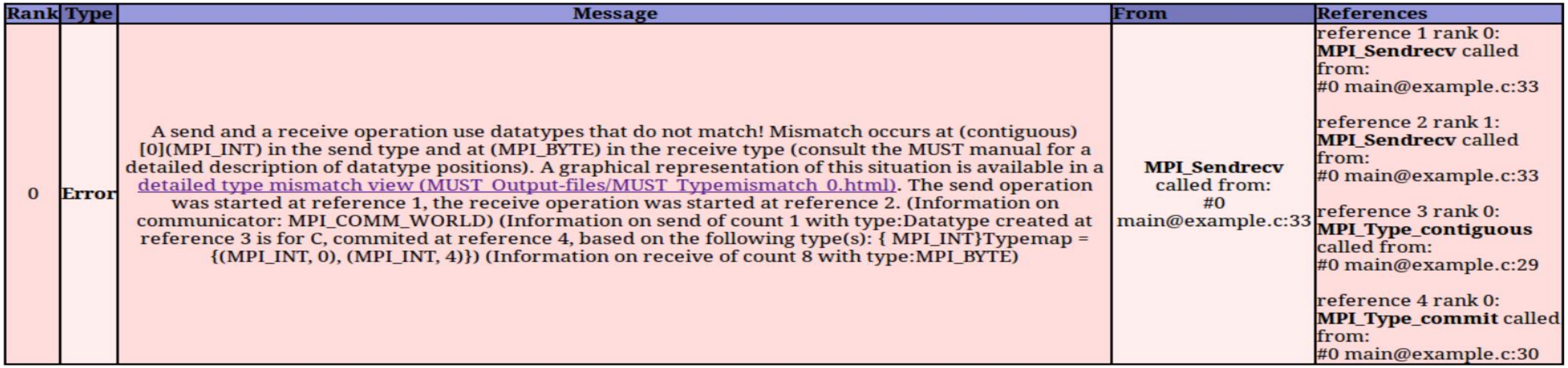
MUST

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

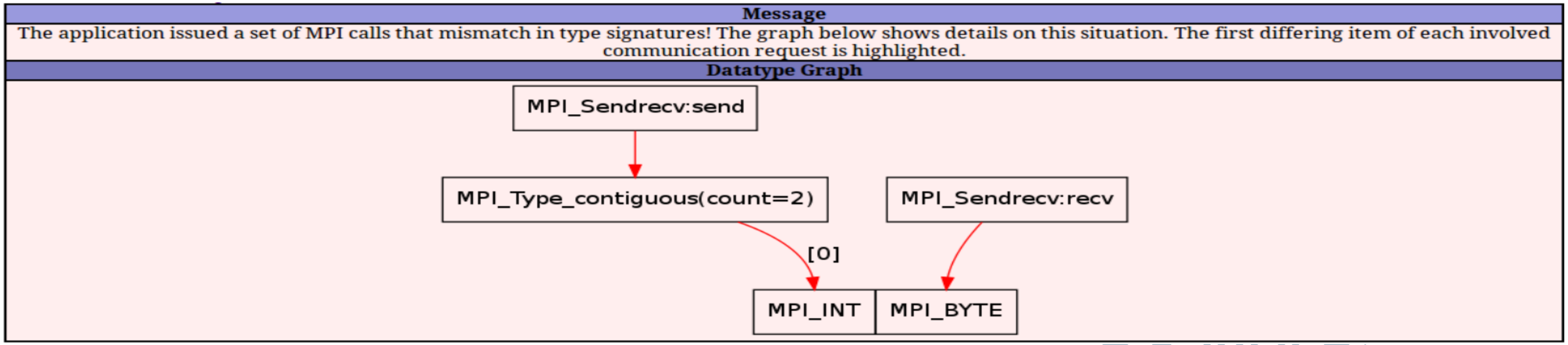
- MUST reports
  - Errors: violations of the MPI-standard
  - Warnings: unusual behavior or possible problems
  - Notes: harmless but remarkable behavior
  - Potential deadlock detection

- Usage
  - 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



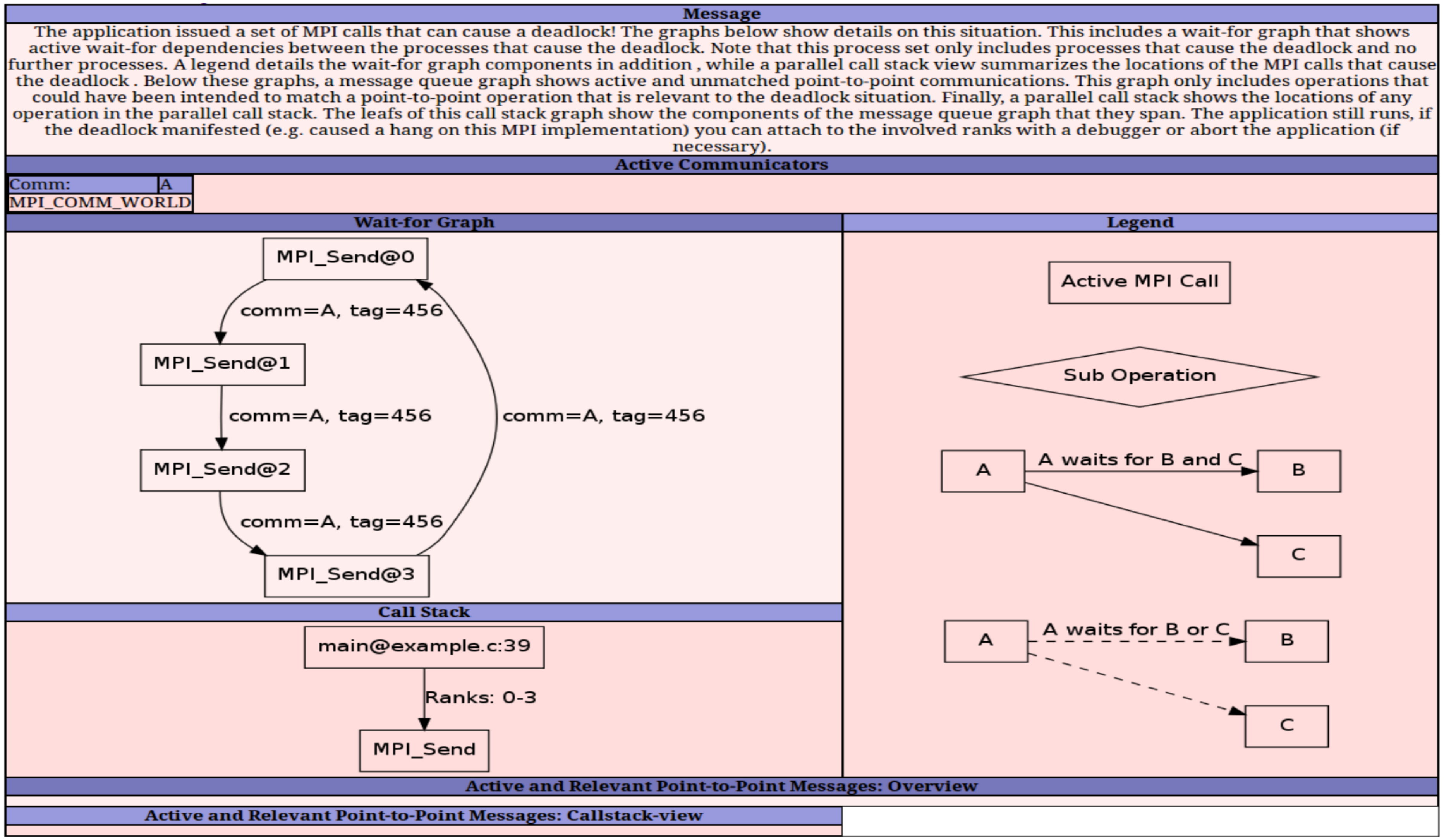
Rank	Type	Message	From	References
0	Error	A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous)[0](MPI_INT) in the send type and at (MPI_BYTE) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation 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

$$\begin{tikzcd}
\boxed{\text{MPI\_Send@0}} \arrow[d, "\text{comm=A, tag=456}"] \\
\boxed{\text{MPI\_Send@1}} \arrow[d, "\text{comm=A, tag=456}"] \\
\boxed{\text{MPI\_Send@2}} \arrow[d, "\text{comm=A, tag=456}"] \\
\boxed{\text{MPI\_Send@3}} \arrow[uuu, bend right, "\text{comm=A, tag=456}"']
\end{tikzcd}$$

Legend

Call Stack

$$\begin{tikzcd}
\boxed{\text{main@example.c:39}} \arrow[d, "\text{Ranks: 0-3}"] \\
\boxed{\text{MPI\_Send}}
\end{tikzcd}$$

Active and Relevant Point-to-Point Messages: Overview

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

ARCHER

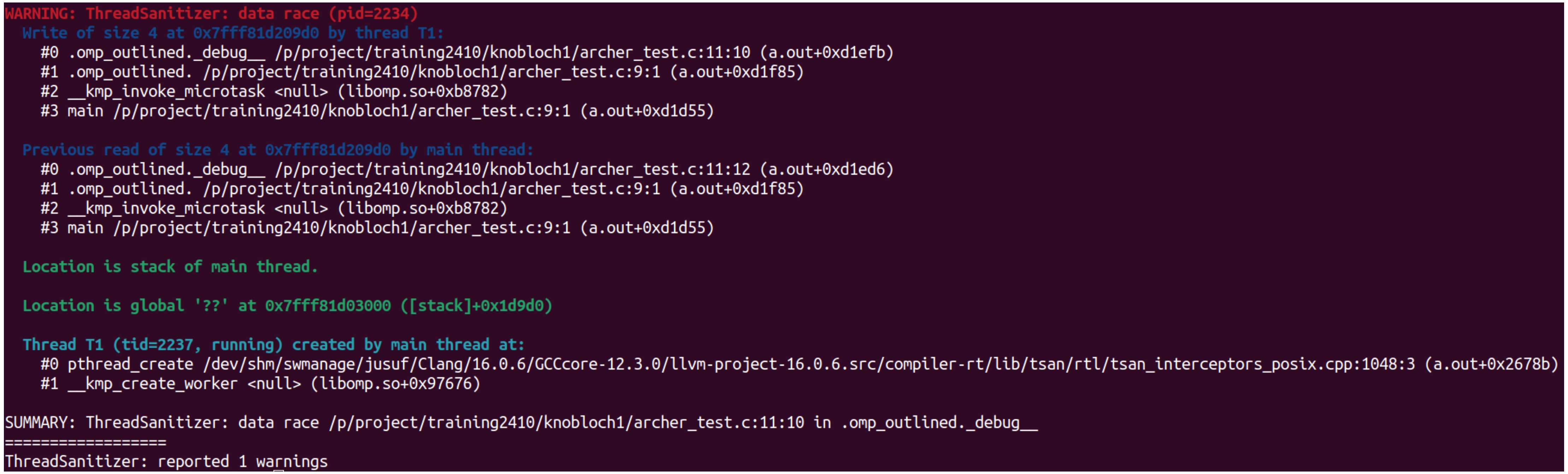
- Data race detector for large OpenMP programs

- 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

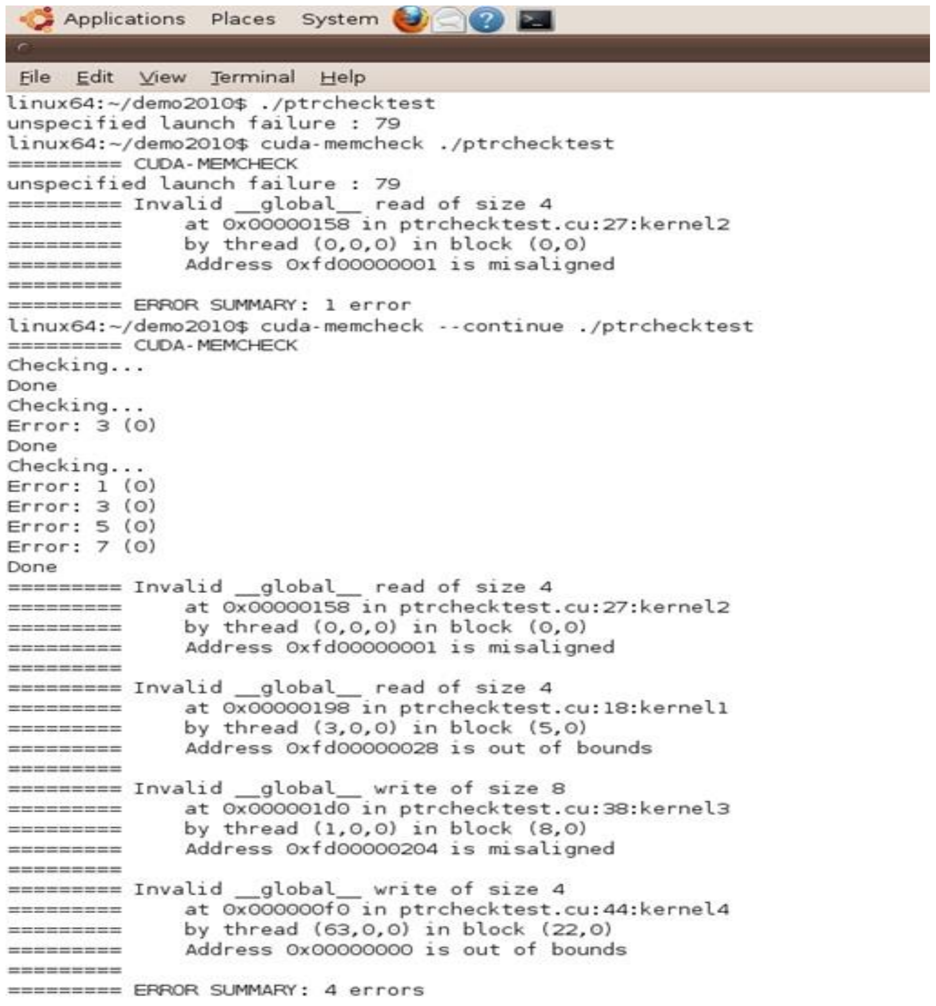


```
WARNING: ThreadSanitizer: data race (pid=2234)
  Write of size 4 at 0x7fff81d209d0 by thread T1:
    #0 .omp_outlined._debug__ /p/project/training2410/knobloch1/archer_test.c:11:10 (a.out+0xd1efb)
    #1 .omp_outlined. /p/project/training2410/knobloch1/archer_test.c:9:1 (a.out+0xd1f85)
    #2 __kmp_invoke_microtask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

CUDA COMPUTE SANITIZER

- Valgrind for GPUs
- Monitors hundreds of thousands of threads running concurrently on each GPU
- Multiple Tools to detect various issues
  - Memcheck – Memory error and leak detection tool
  - Racecheck – Shared memory data access hazard detection tool
  - Initcheck – Uninitialized device global memory access detection tool
  - Synccheck – Thread synchronization hazard detection tool

- Included in the CUDA Toolkit



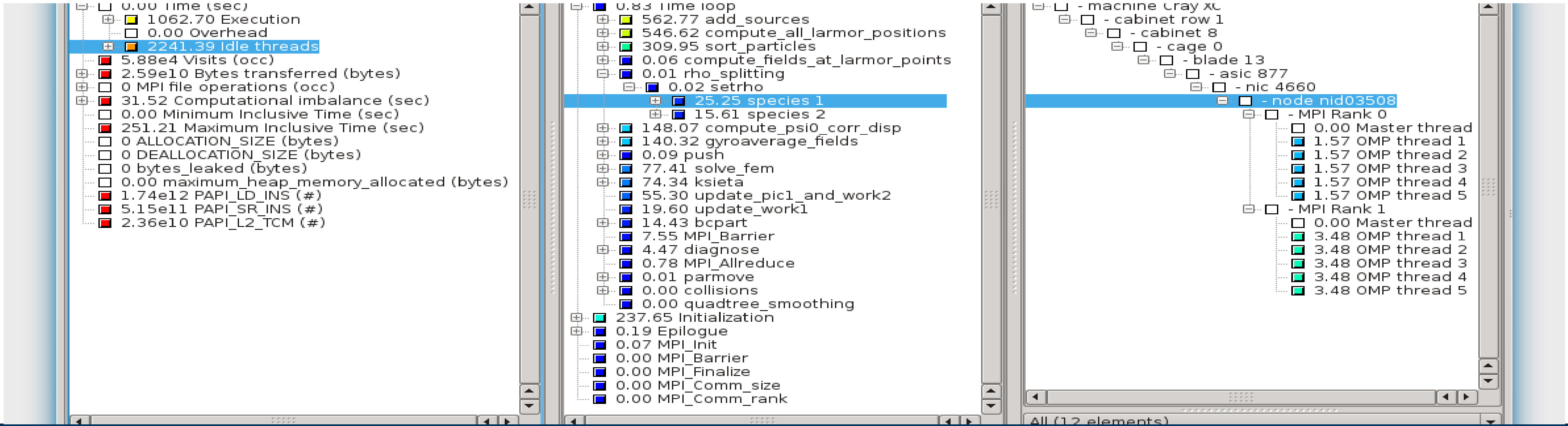
```
linux64:~/demo2010$ ./ptrchecktest
unspecified launch failure : 79
linux64:~/demo2010$ cuda-memcheck ./ptrchecktest
========= CUDA-MEMCHECK
unspecified launch failure : 79
========= Invalid __global__ read of size 4
=========     at 0x00000158 in ptrchecktest.cu:27:kernel2
=========     by thread (0,0,0) in block (0,0)
=========     Address 0xfd00000001 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 0xfd00000001 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
```

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 neccessary



```
0.00 Time (sec)
  1062.70 Execution
  0.00 Overhead
  2241.39 Idle threads
5.88e4 Visits (occ)
2.59e10 Bytes transferred (bytes)
0 MPI file operations (occ)
31.52 Computational imbalance (sec)
0.00 Minimum Inclusive Time (sec)
251.21 Maximum Inclusive Time (sec)
0 ALLOCATION_SIZE (bytes)
0 DEALLOCATION_SIZE (bytes)
0 bytes_leaked (bytes)
0.00 maximum_heap_memory_allocated (bytes)
1.74e12 PAPI_LD_INS (#)
5.15e11 PAPI_SR_INS (#)
2.36e10 PAPI_L2_TCM (#)
```

```
0.83 Time loop
  562.77 add_sources
  546.62 compute_all_larmor_positions
  309.95 sort_particles
  0.06 compute_fields_at_larmor_points
  0.01 rho_splitting
    0.02 setrho
      25.25 species 1
      15.61 species 2
  148.07 compute_psi0_corr_disp
  140.32 gyroaverage_fields
  0.09 push
  77.41 solve_fem
  74.34 ksieta
  55.30 update_pic1_and_work2
  19.60 update_work1
  14.43 bcpart
  7.55 MPI_Barrier
  4.47 diagnose
  0.78 MPI_Allreduce
  0.01 parmove
  0.00 collisions
  0.00 quadtree_smoothing
237.65 Initialization
0.19 Epilogue
0.07 MPI_Init
0.00 MPI_Barrier
0.00 MPI_Finalize
0.00 MPI_Comm_size
0.00 MPI_Comm_rank
```

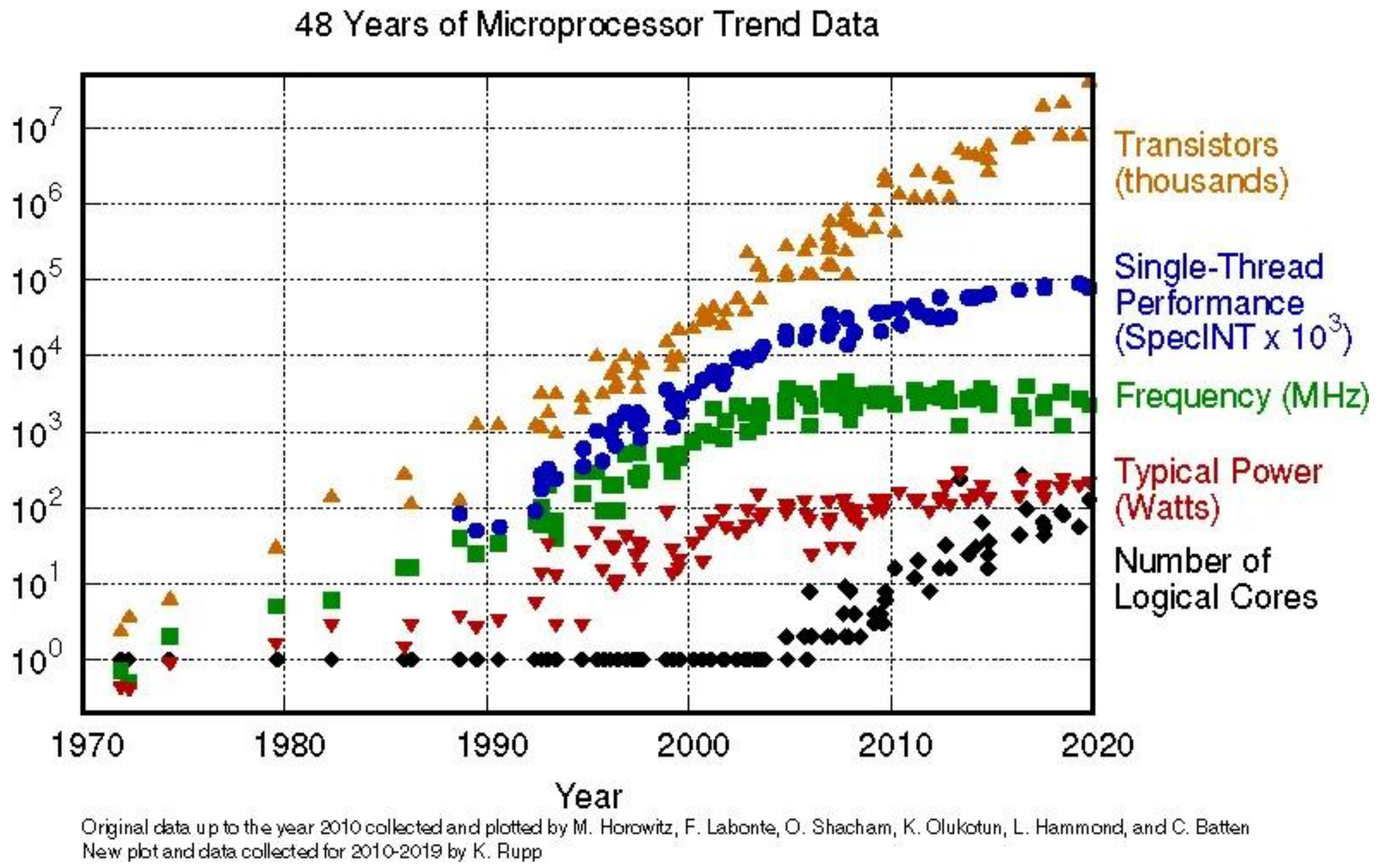
```
- machine Cray XC
  - cabinet row 1
    - cabinet 8
      - cage 0
        - blade 13
          - asic 877
            - nic 4660
              - node nid03508
                - MPI Rank 0
                  0.00 Master thread
                  1.57 OMP thread 1
                  1.57 OMP thread 2
                  1.57 OMP thread 3
                  1.57 OMP thread 4
                  1.57 OMP thread 5
                - MPI Rank 1
                  0.00 Master thread
                  3.48 OMP thread 1
                  3.48 OMP thread 2
                  3.48 OMP thread 3
                  3.48 OMP thread 4
                  3.48 OMP thread 5
All (12 elements)
```

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

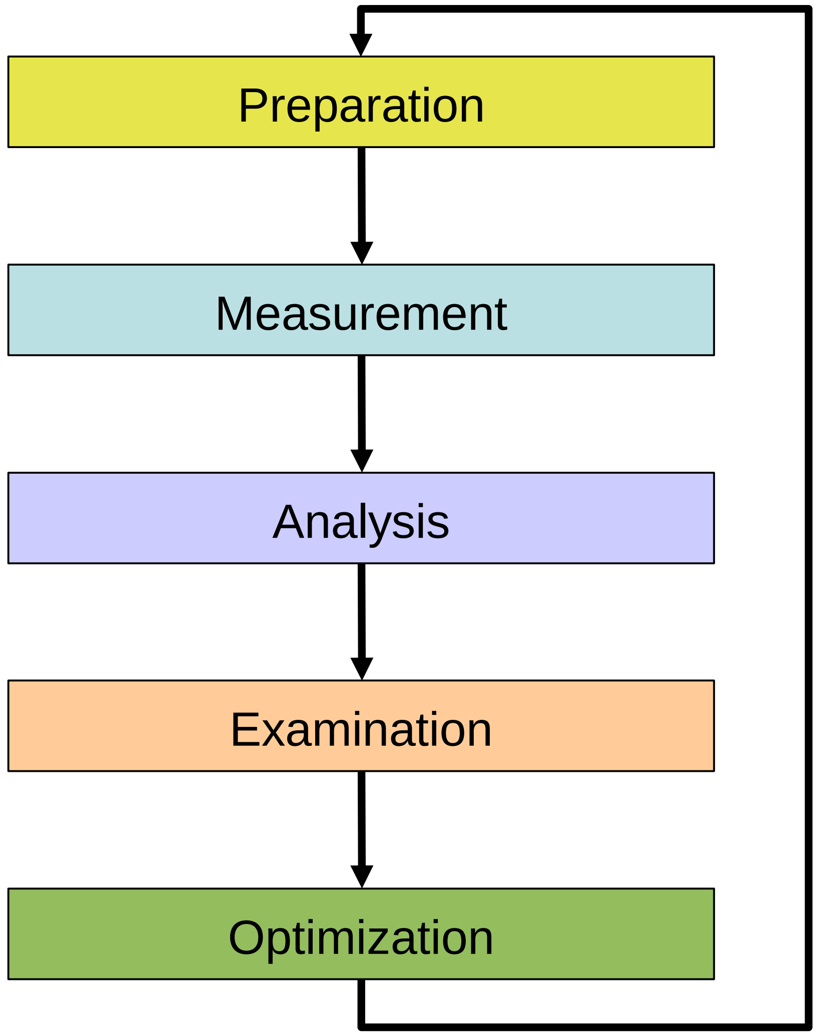
- “Parallel” (multi core/node) factors
  - Partitioning / decomposition
    - ☞ Load balancing
  - Communication (i.e., message passing)
  - Multithreading
  - Core binding / NUMA
  - Synchronization / locking
  - I/O
    - ☞ Often not given enough attention
    - ☞ Parallel I/O matters

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

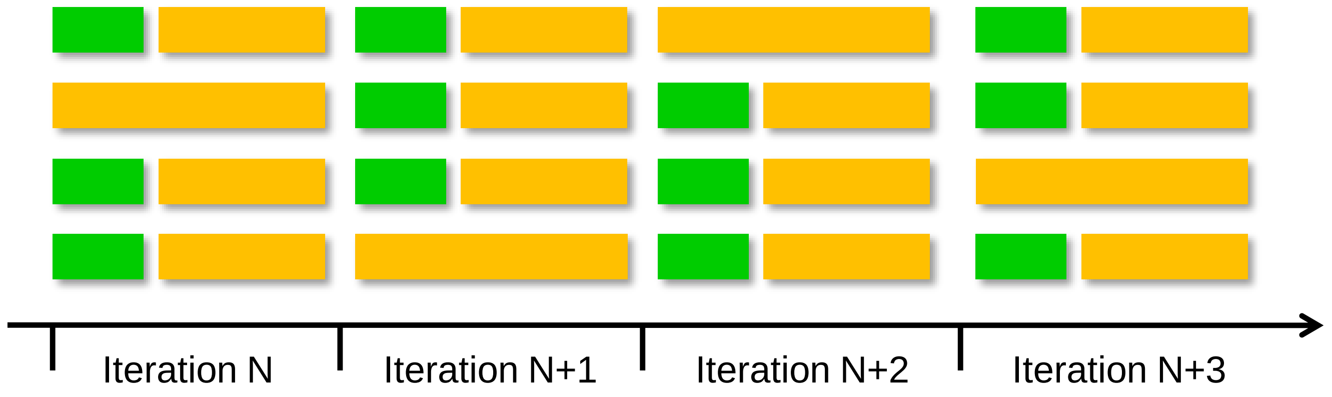
- Recording of **aggregated information**
  - Time
  - Counts
    - Calls
    - Hardware counters
- **Across program and system entities**
  - Functions, call sites, loops, basic blocks, …
  - Processes, threads
- **Statistical information**
  - Min, max, mean and total number of values

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

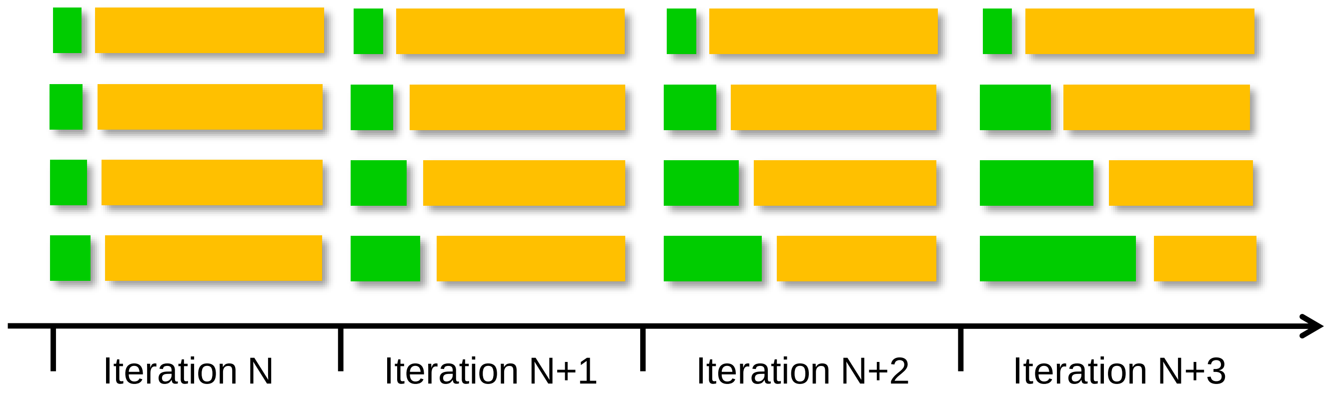
Disadvantages
– **Variations over time get lost**

PROFILING: ISSUES RELATED TO "AVERAGING"

- Moving bottleneck across processors can "average out" imbalances



- Imbalance changes over time ⇨ 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**

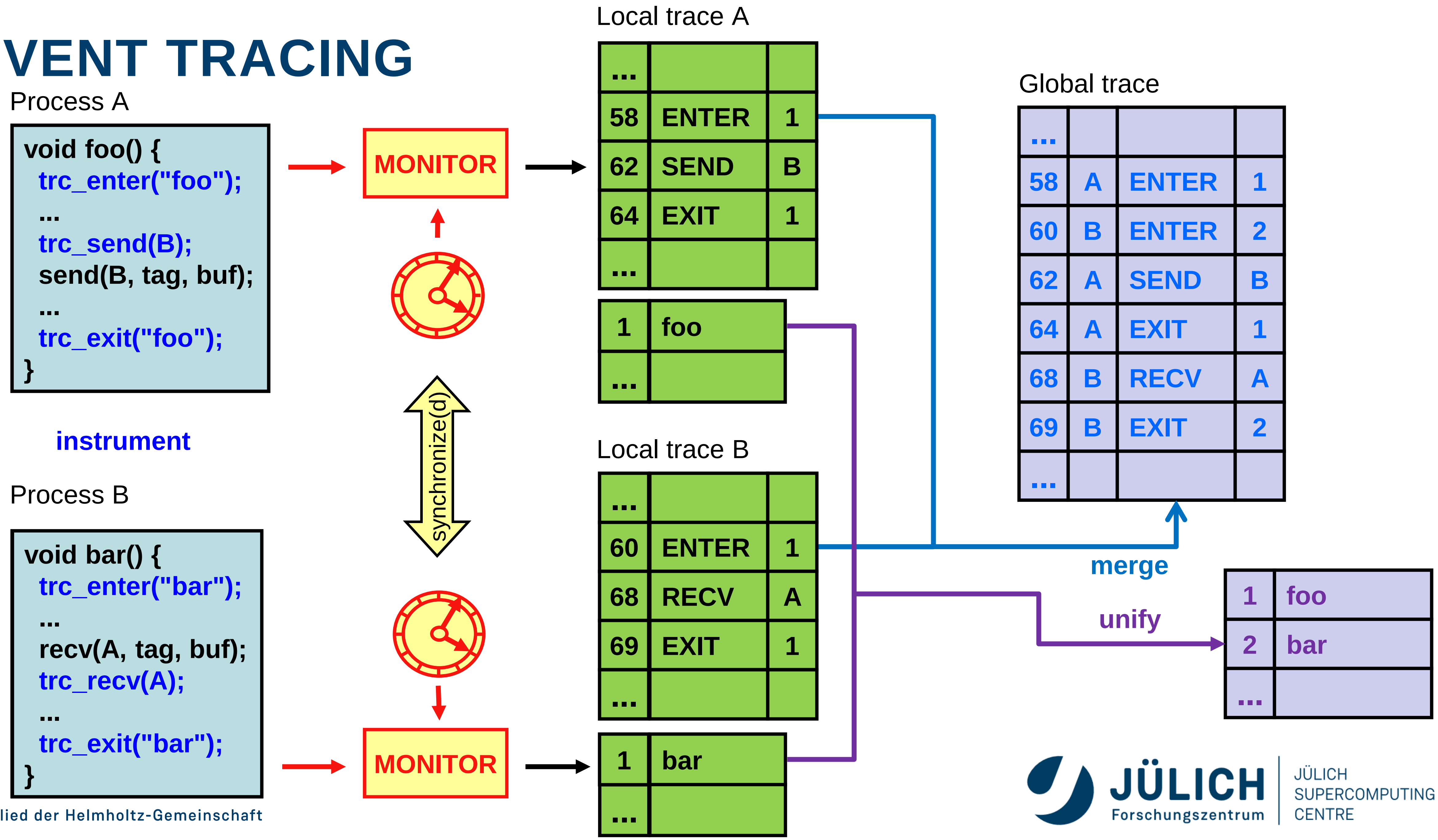
+ **Profiles can be calculated out of trace data**

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");
}
```

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

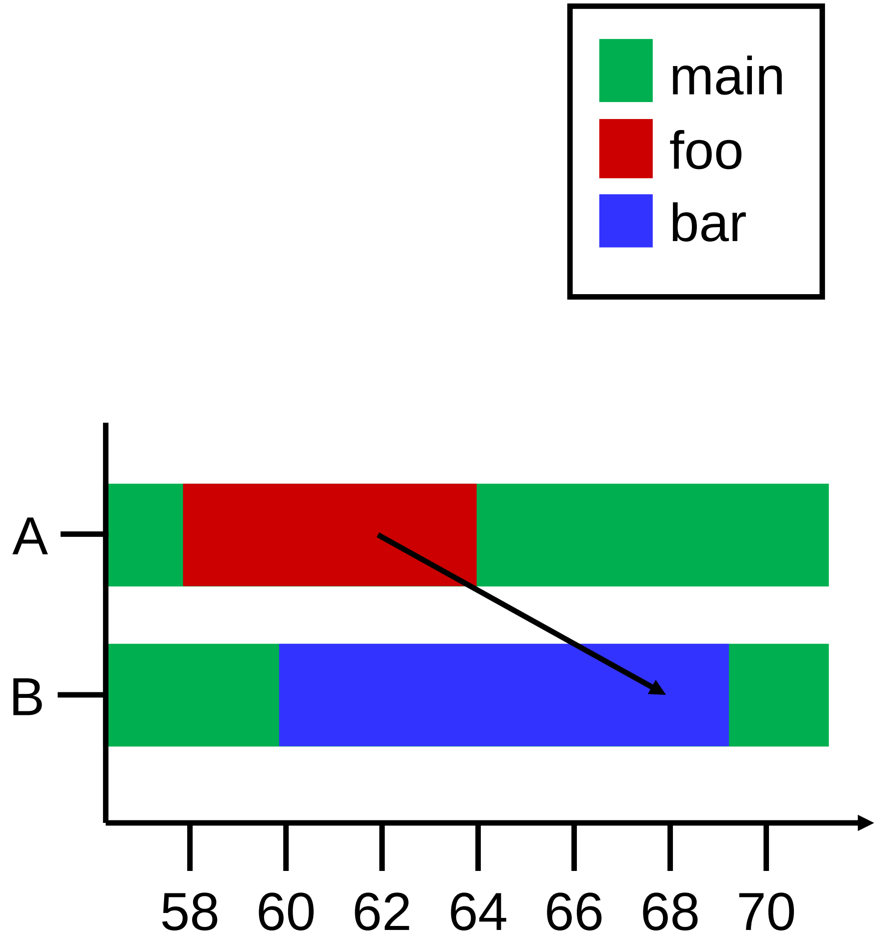
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

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
…			

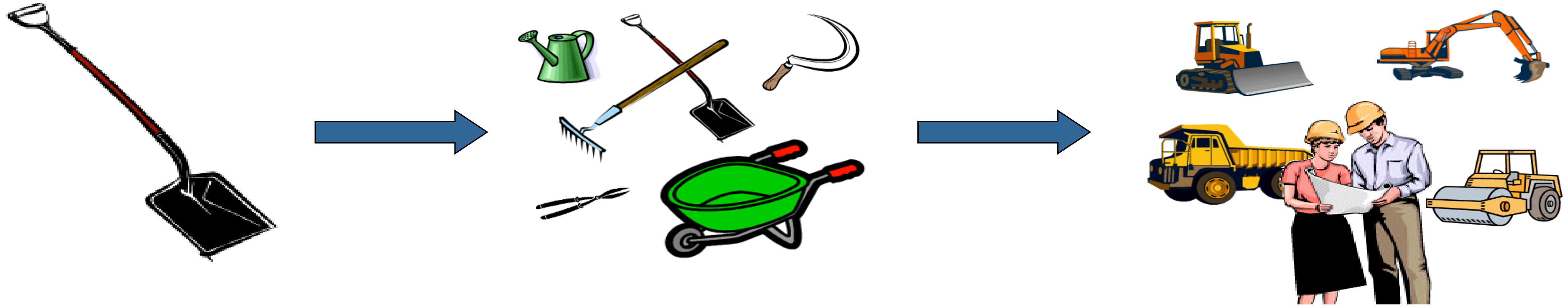


CRITICAL ISSUES

- 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!

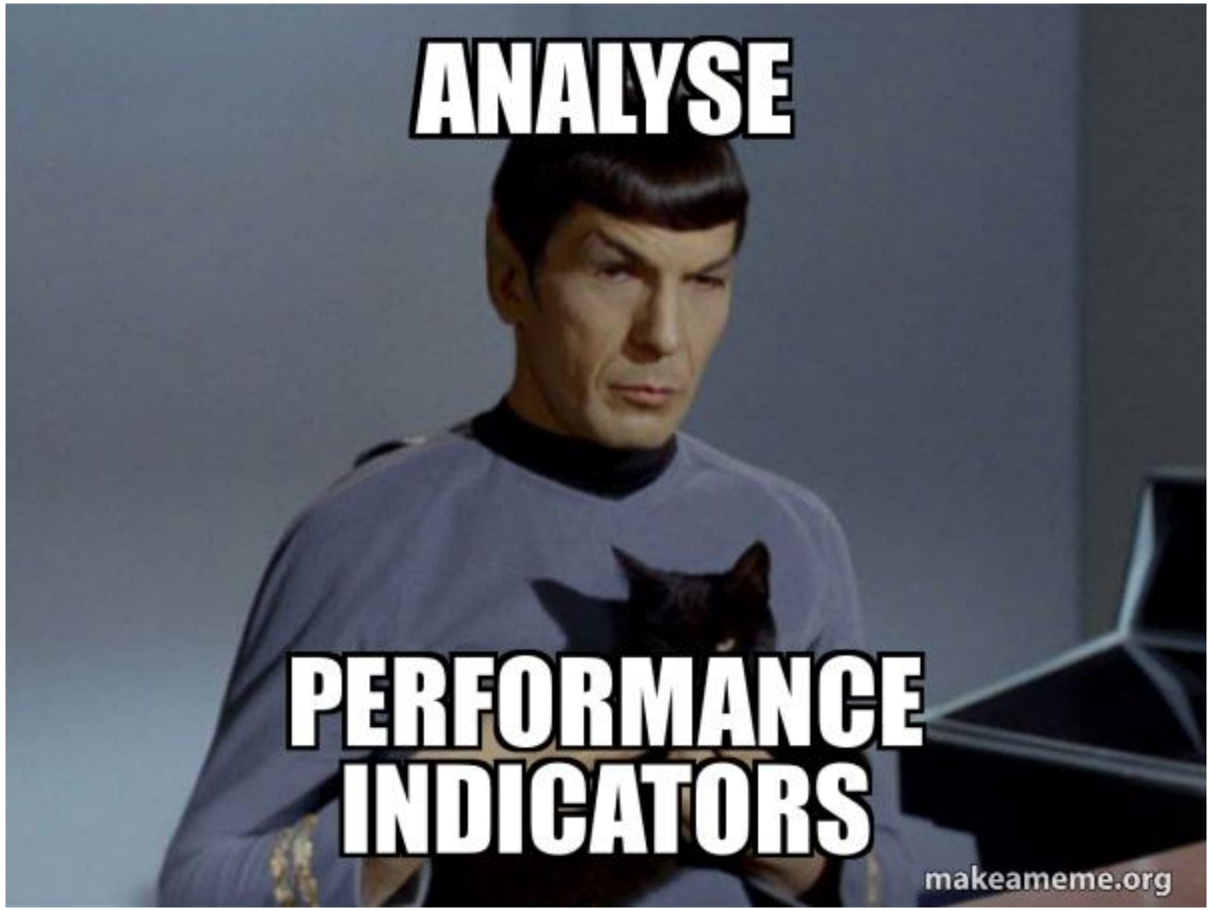


☞ *A combination of different methods, tools and techniques is typically needed!*

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

PERFORMANCE TOOLS (STATUS: NOV 2025)

- Score-P
- Scalasca
- Vampir[Server]
- Linaro Forge
    - Performance Reports
    - MAP
- Intel Tools
    - VTune Amplifier XE
    - Intel Advisor
- 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

TECHNISCHE UNIVERSITÄT DRESDEN

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

German Research School for Simulation Sciences

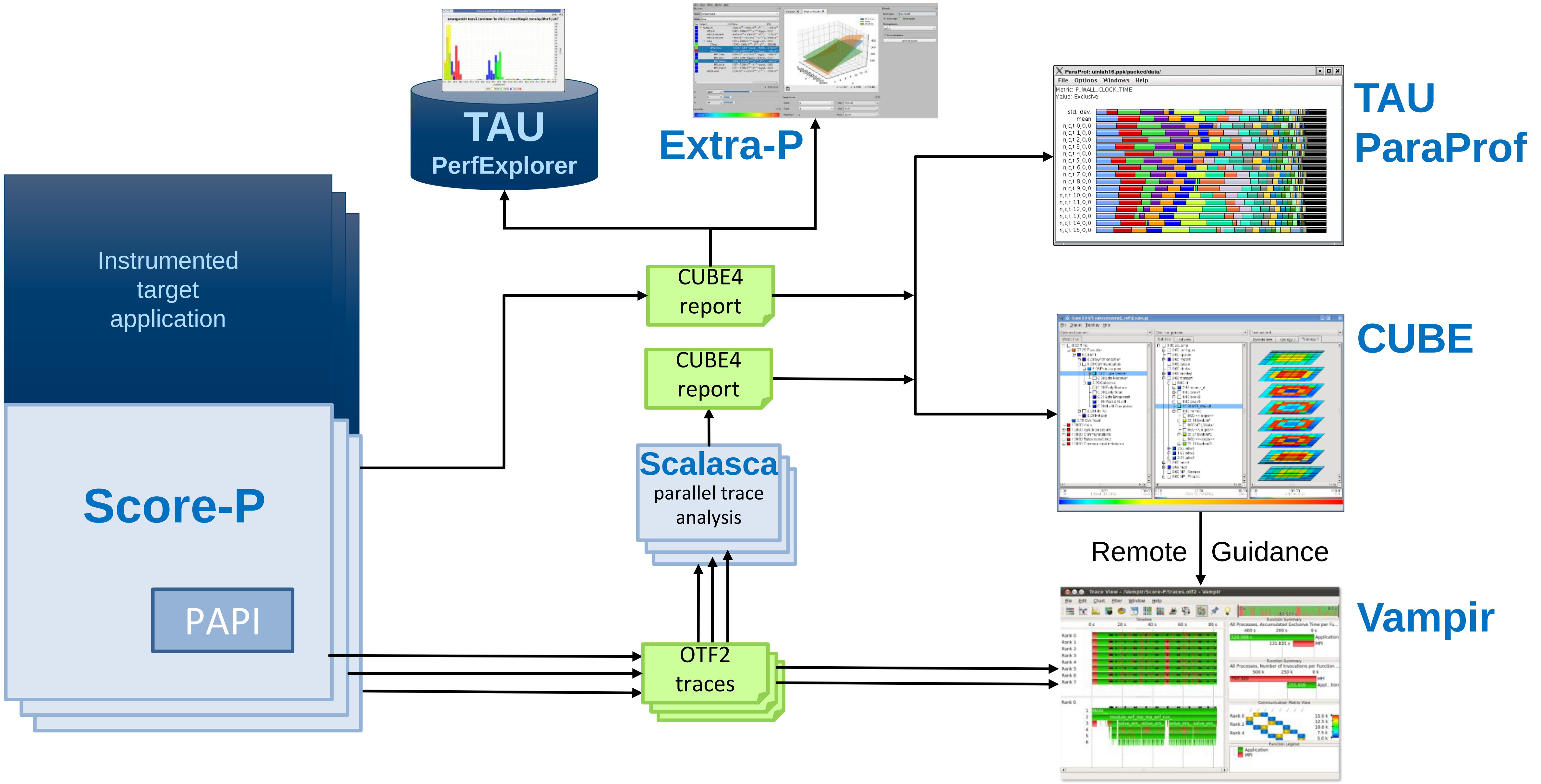
RWTH AACHEN UNIVERSITY

TUM Technische Universität München

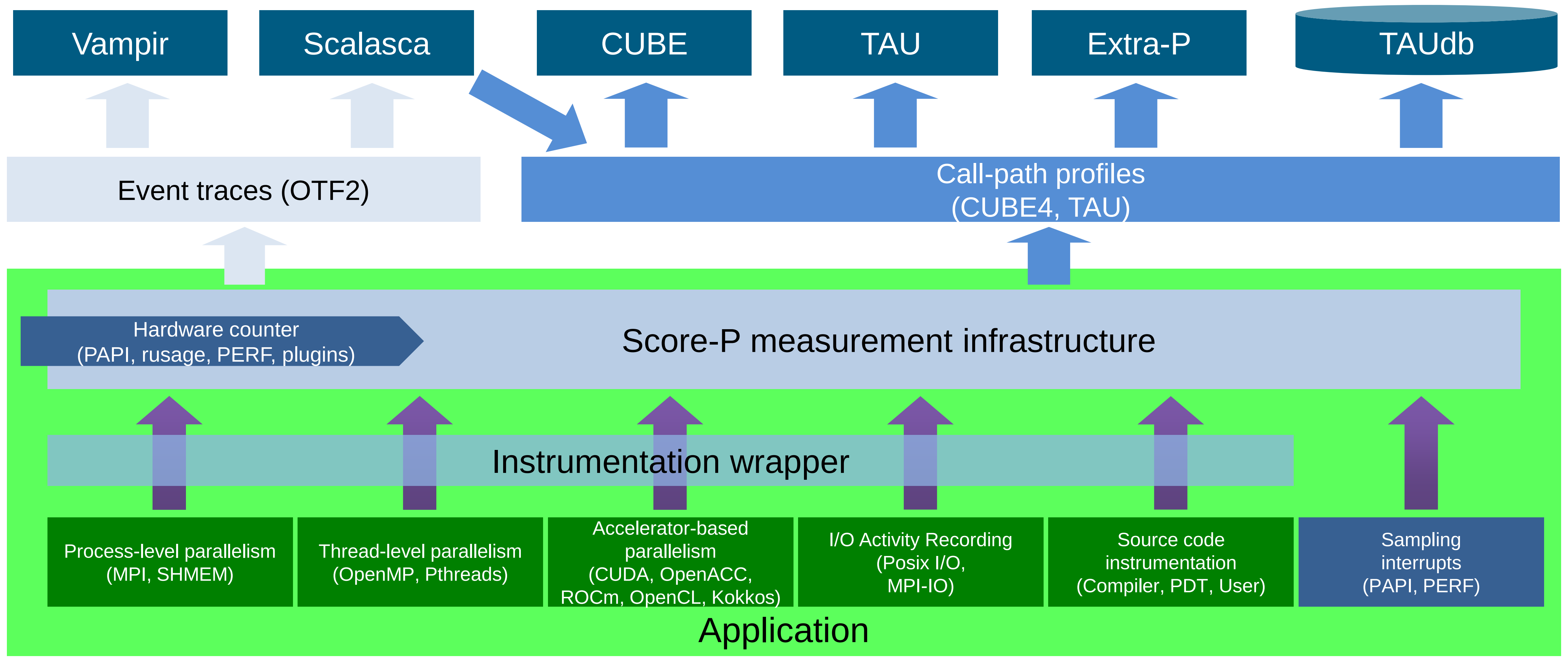
UNIVERSITY OF OREGON

Score-P TOOL ECOSYSTEM

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

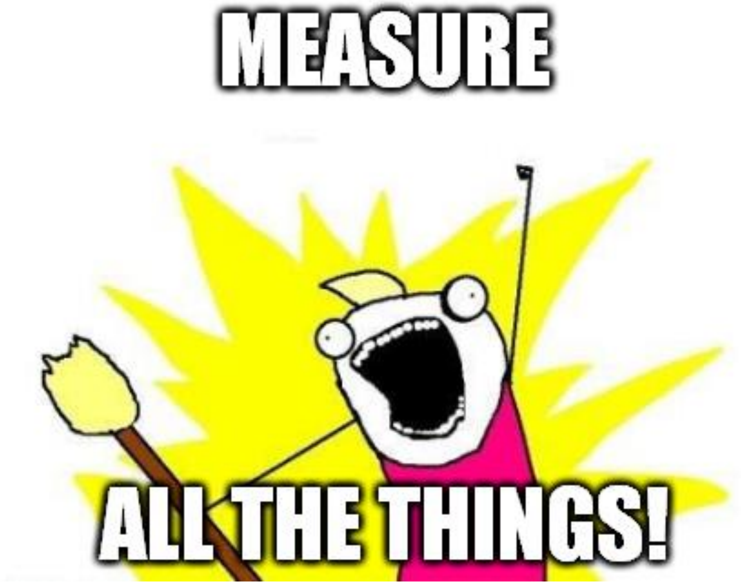


Score-P ARCHITECURE

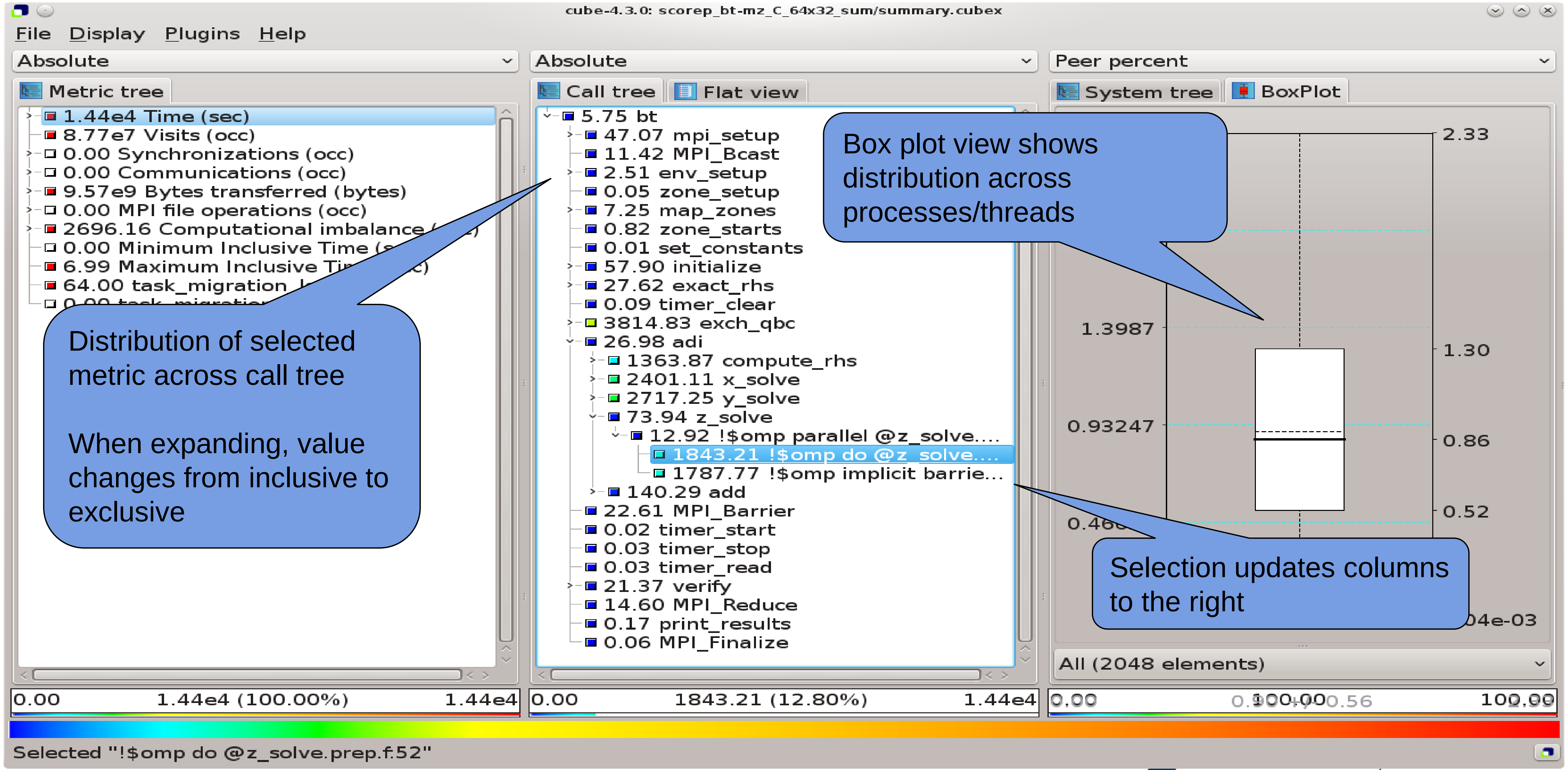

Vampir
Scalasca
CUBE
TAU
Extra-P
TAUdb
Event traces (OTF2)
Call-path profiles (CUBE4, TAU)
Hardware counter (PAPI, rusage, PERF, plugins)
Score-P measurement infrastructure
Instrumentation wrapper
Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenACC, ROCm, OpenCL, Kokkos)
I/O Activity Recording (Posix I/O, MPI-IO)
Source code instrumentation (Compiler, PDT, User)
Sampling interrupts (PAPI, PERF)
Application

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, ROCm, 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



cube-4.3.0: scorep_bt-mz_C_64x32_sum/summary.cubex

File Display Plugins Help

Absolute | Absolute | Peer percent

Metric tree | Call tree | Flat view | System tree | BoxPlot

```
1.44e4 Time (sec)
8.77e7 Visits (occ)
0.00 Synchronizations (occ)
0.00 Communications (occ)
9.57e9 Bytes transferred (bytes)
0.00 MPI file operations (occ)
2696.16 Computational imbalance (
0.00 Minimum Inclusive Time (s
6.99 Maximum Inclusive Ti      c)
64.00 task_migration_l
0.00 task_migration
```

```
5.75 bt
  47.07 mpi_setup
  11.42 MPI_Bcast
  2.51 env_setup
  0.05 zone_setup
  7.25 map_zones
  0.82 zone_starts
  0.01 set_constants
  57.90 initialize
  27.62 exact_rhs
  0.09 timer_clear
  3814.83 exch_qbc
  26.98 adi
    1363.87 compute_rhs
    2401.11 x_solve
    2717.25 y_solve
    73.94 z_solve
      12.92 !$omp parallel @z_solve....
        1843.21 !$omp do @z_solve....
        1787.77 !$omp implicit barrie...
    140.29 add
  22.61 MPI_Barrier
  0.02 timer_start
  0.03 timer_stop
  0.03 timer_read
  21.37 verify
  14.60 MPI_Reduce
  0.17 print_results
  0.06 MPI_Finalize
```

All (2048 elements)

0.00 1.44e4 (100.00%) 1.44e4 | 0.00 1843.21 (12.80%) 1.44e4 | 0.00 100.00 100.00

Selected "!$omp do @z_solve.prep.f:52"

Box plot view shows distribution across processes/threads

Distribution of selected metric across call tree

When expanding, value changes from inclusive to exclusive

Selection updates columns to the right

SCORE-P: ADVANCED FEATURES

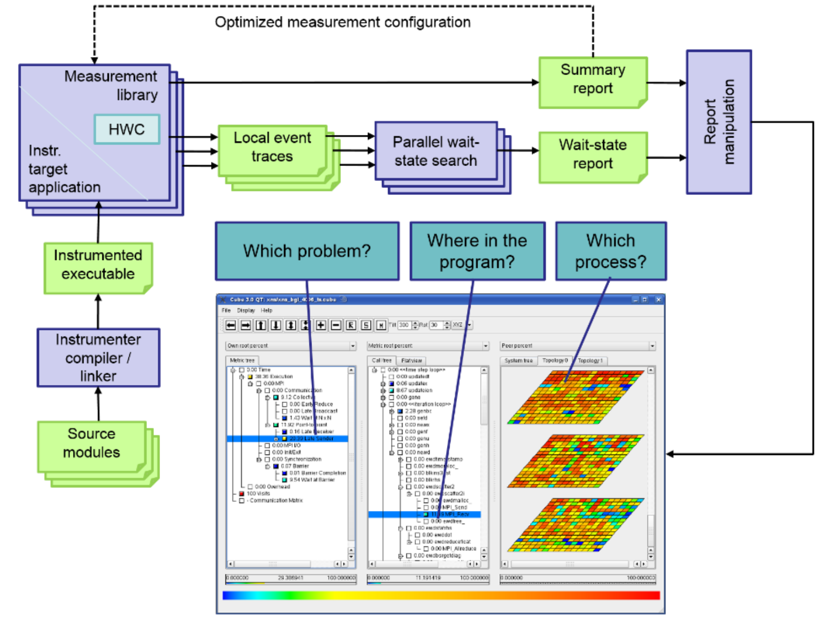
- Measurement can be extensively configured via environment variables
- Allows for targeted measurements:
  - Selective recording
  - Phase profiling
  - Parameter-based profiling
  - …
- GPU support: CUDA, OpenACC, OpenCL, HIP, Kokkos, …
- Please ask us or see the user manual for details

SCALASCA

http://www.scalasca.org/

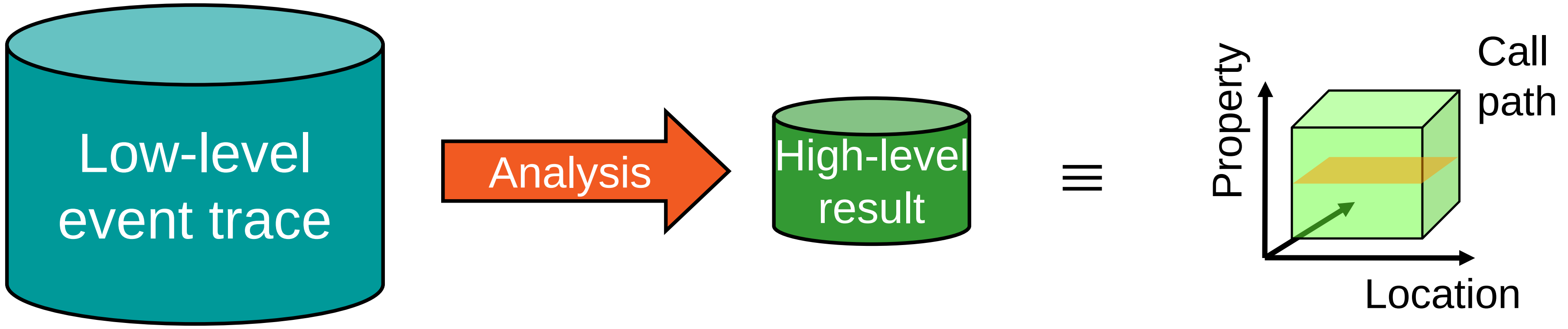
- **Sc**alable **A**nalysis of **La**rge **Sc**ale **A**pplications

- 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



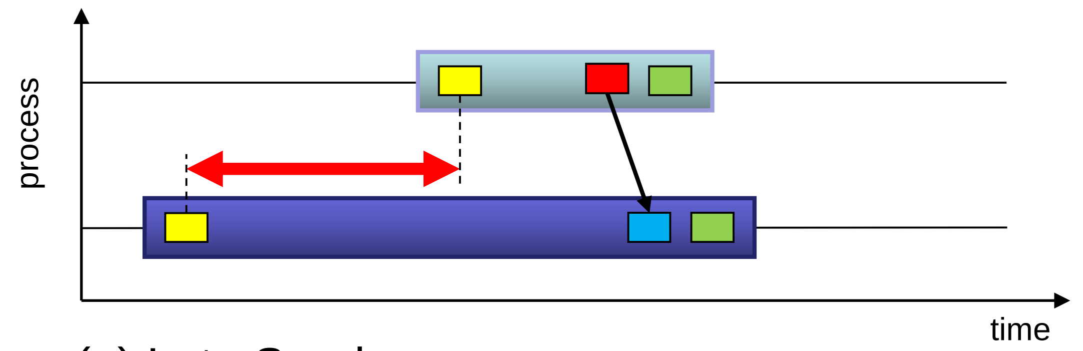
AUTOMATIC TRACE ANALYSIS

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

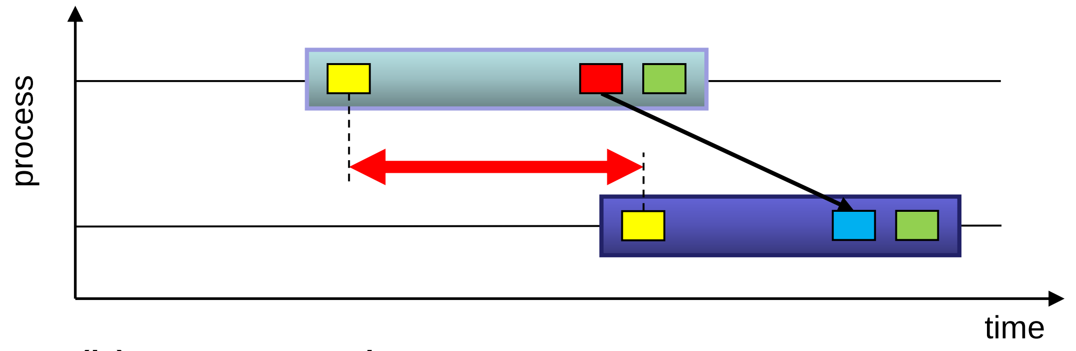


- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability

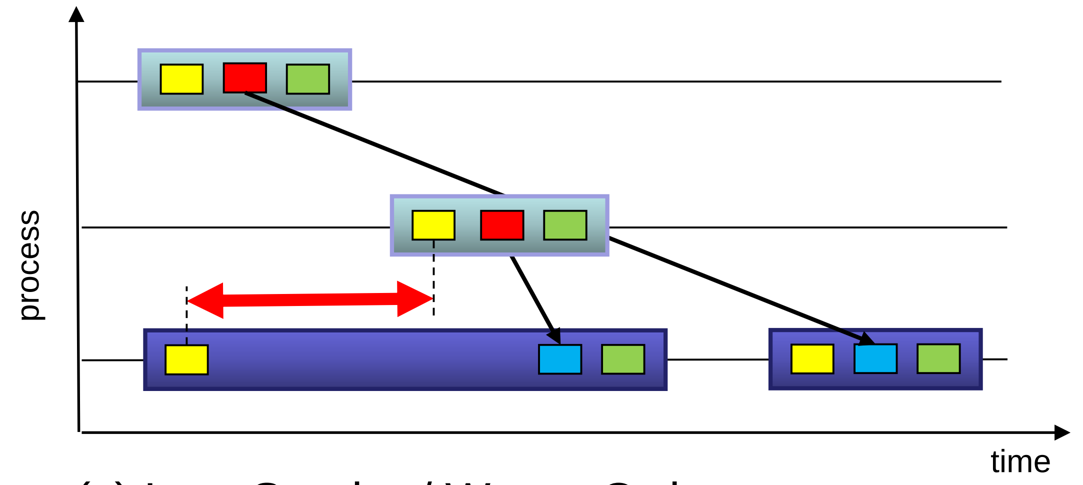
EXAMPLE MPI WAIT STATES



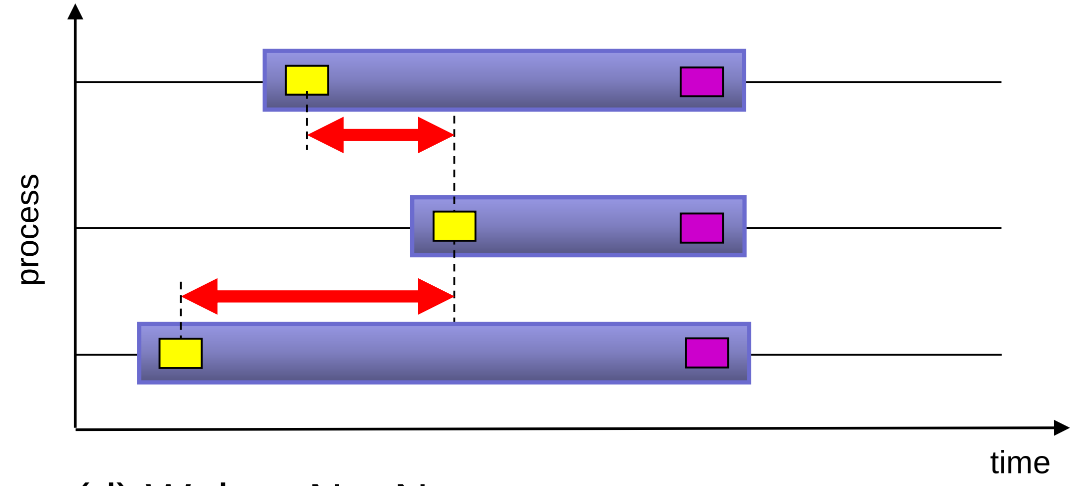
(a) Late Sender



(b) Late Receiver



(c) Late Sender / Wrong Order

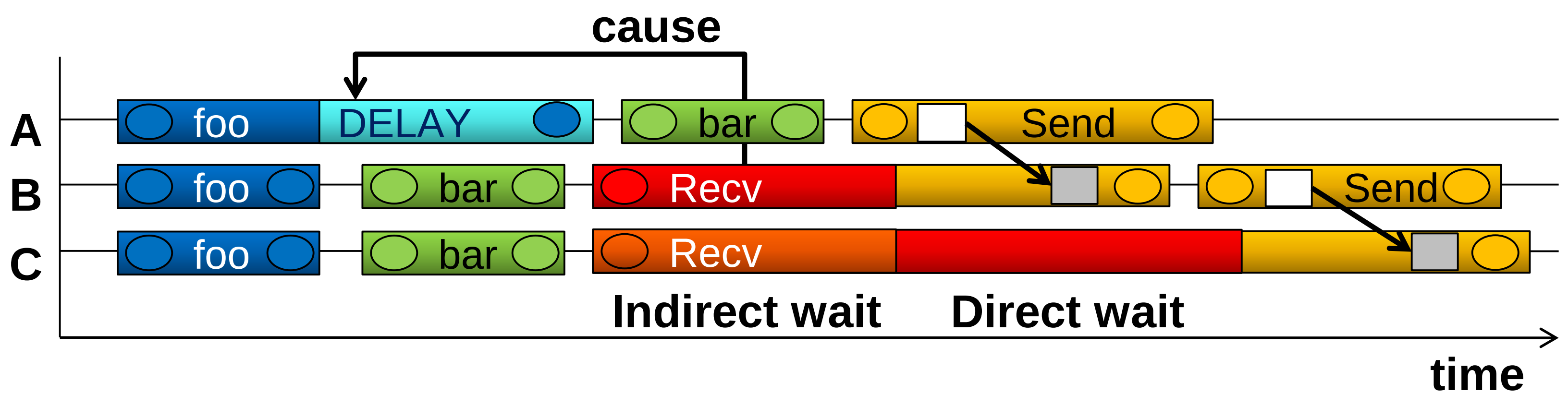


(d) Wait at N x N

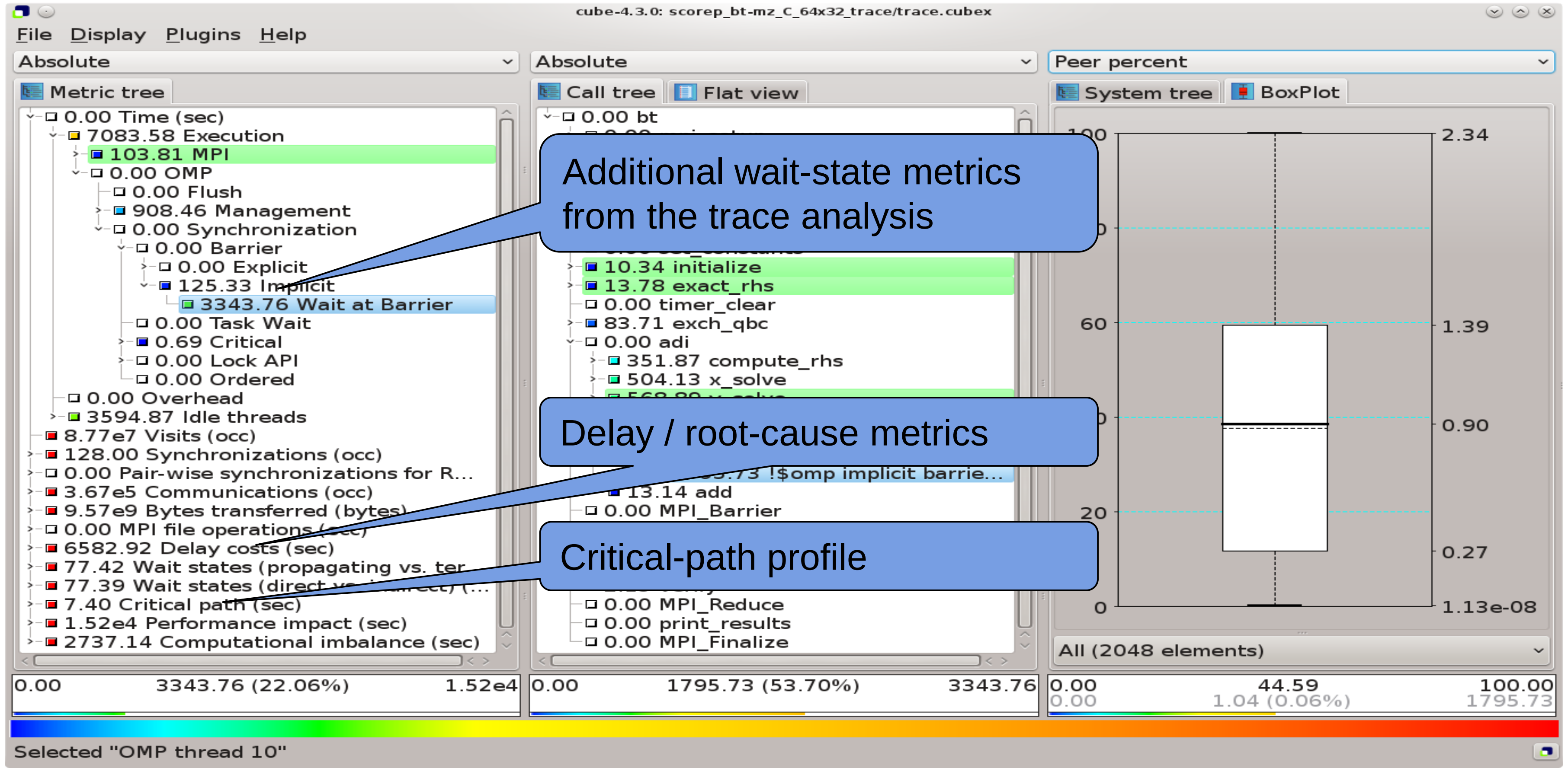
🟨 ENTER 🟩 EXIT 🟥 SEND 🟦 RECV 🟪 COLLEXIT

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



```
cube-4.3.0: scorep_bt-mz_C_64x32_trace/trace.cubex
File  Display  Plugins  Help

Absolute                          Absolute                          Peer percent
Metric tree                       Call tree | Flat view             System tree | BoxPlot

0.00 Time (sec)                   0.00 bt
  7083.58 Execution                 10.34 initialize
    103.81 MPI                      13.78 exact_rhs
    0.00 OMP                        0.00 timer_clear
      0.00 Flush                    83.71 exch_qbc
      908.46 Management             0.00 adi
      0.00 Synchronization            351.87 compute_rhs
        0.00 Barrier                  504.13 x_solve
          0.00 Explicit               !$omp implicit barrie…
          125.33 Implicit             13.14 add
            3343.76 Wait at Barrier 0.00 MPI_Barrier
        0.00 Task Wait              0.00 MPI_Reduce
        0.69 Critical               0.00 print_results
        0.00 Lock API               0.00 MPI_Finalize
        0.00 Ordered
    0.00 Overhead
    3594.87 Idle threads
8.77e7 Visits (occ)
128.00 Synchronizations (occ)
0.00 Pair-wise synchronizations for R…
3.67e5 Communications (occ)
9.57e9 Bytes transferred (bytes)
0.00 MPI file operations (occ)
6582.92 Delay costs (sec)
77.42 Wait states (propagating vs. ter
77.39 Wait states (direct vs. indirect) (…
7.40 Critical path (sec)
1.52e4 Performance impact (sec)
2737.14 Computational imbalance (sec)

0.00   3343.76 (22.06%)   1.52e4 | 0.00   1795.73 (53.70%)   3343.76 | 0.00   44.59   100.00
                                 |                                   | 0.00   1.04 (0.06%)   1795.73

Selected "OMP thread 10"
```

Additional wait-state metrics from the trace analysis

Delay / root-cause metrics

Critical-path profile

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

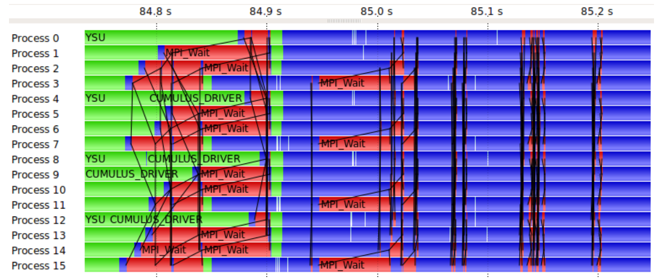
- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Completely manual analysis
  - Too many details can hide the relevant parts

EVENT TRACE VISUALIZATION WITH VAMPIR

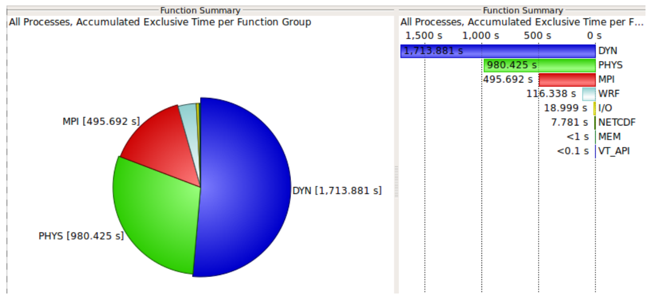
- Visualization of dynamic runtime behaviour at any level of detail along with statistics and performance metrics
- Alternative and supplement to automatic analysis

- **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

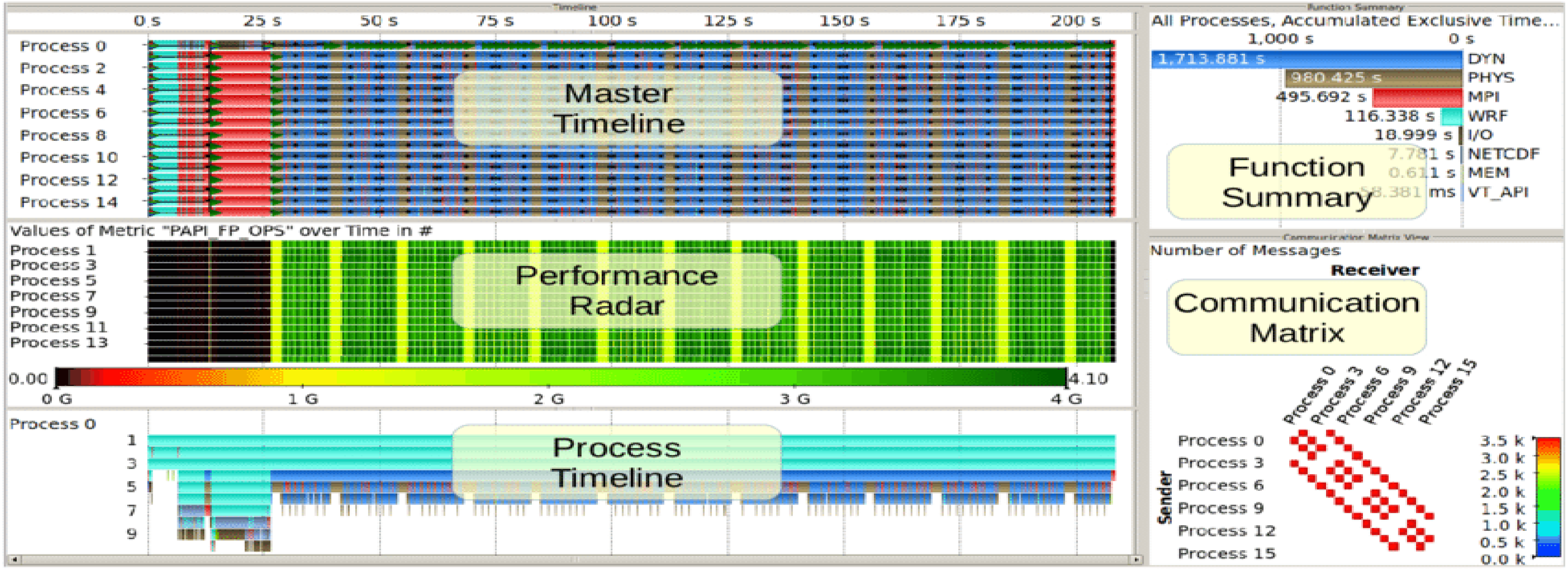
Timeline Charts

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

VAMPIR DISPLAYS



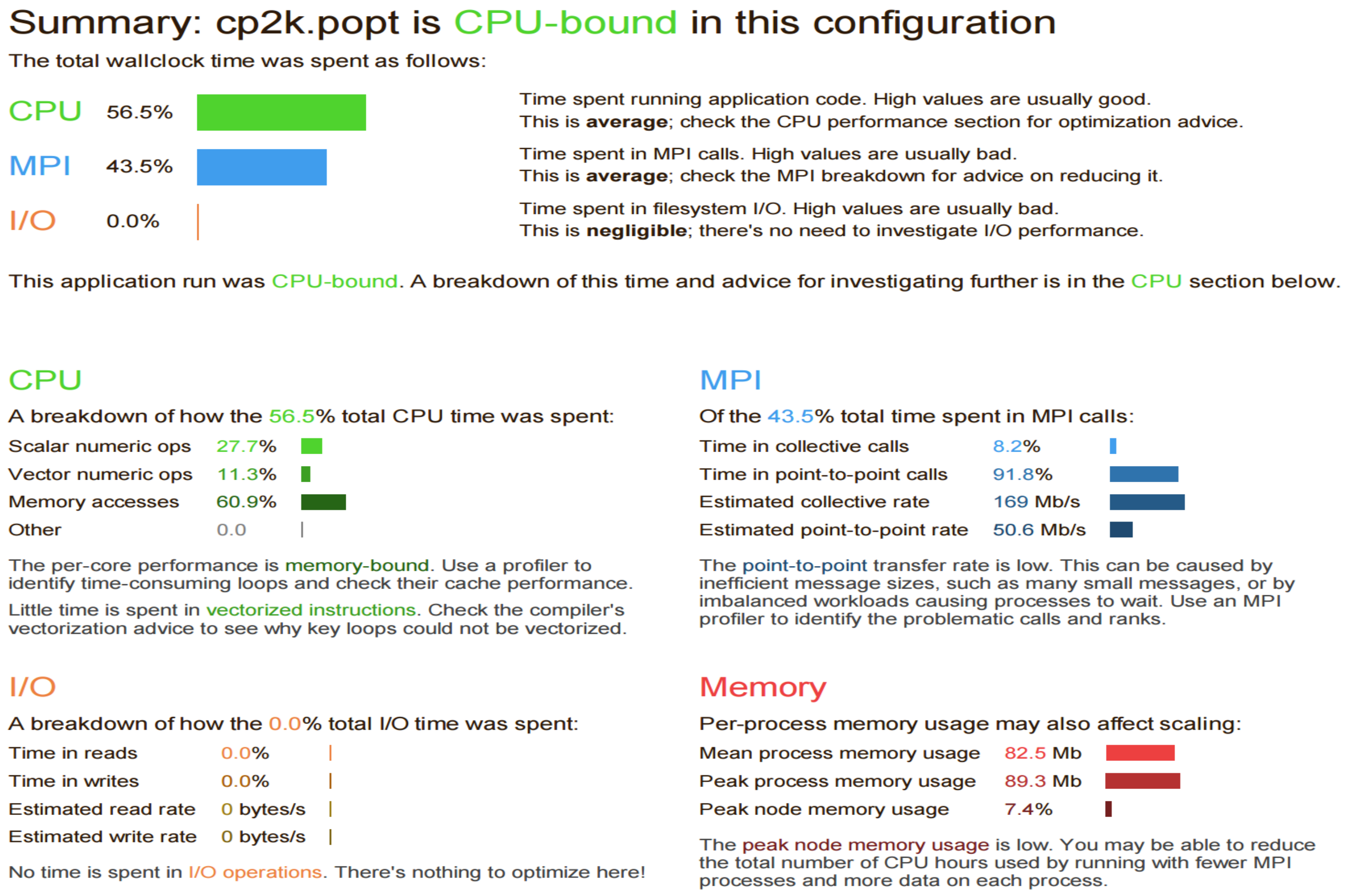
LINARO PERFORMANCE REPORTS

- 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.

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

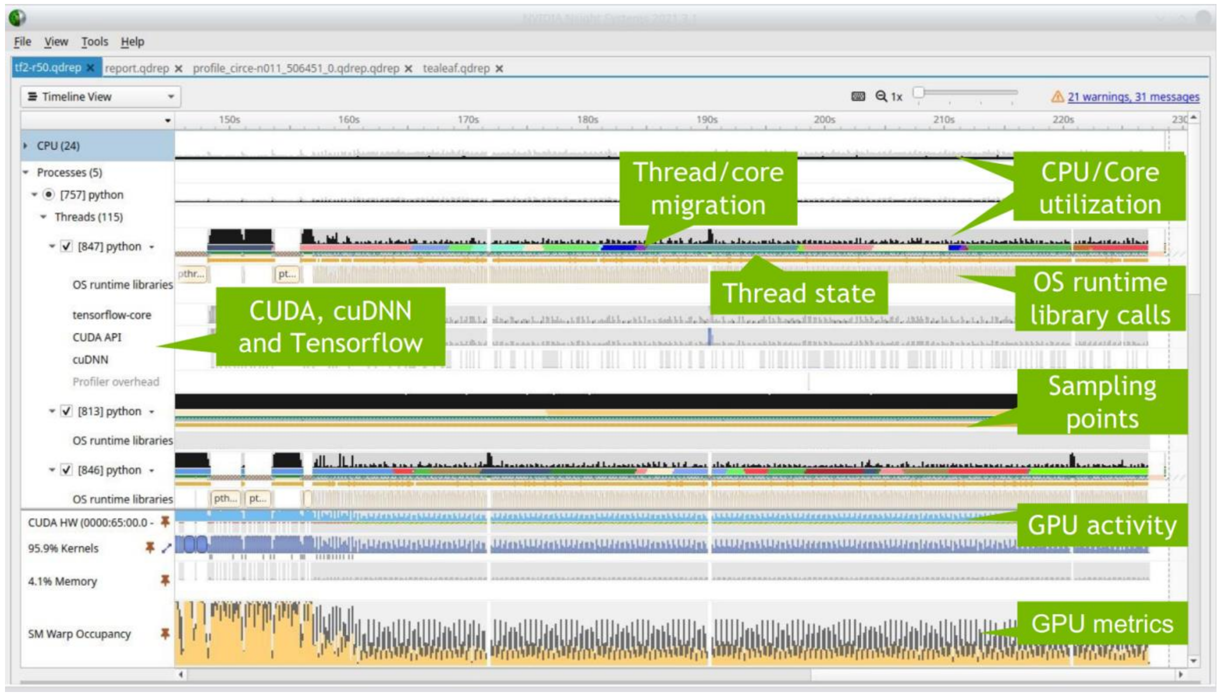
CPU-GPU interactions & triage
Low overhead capture
GPU compute & graphics
Faster GUI + more data

Nsight Compute
Standalone GUI+CLI

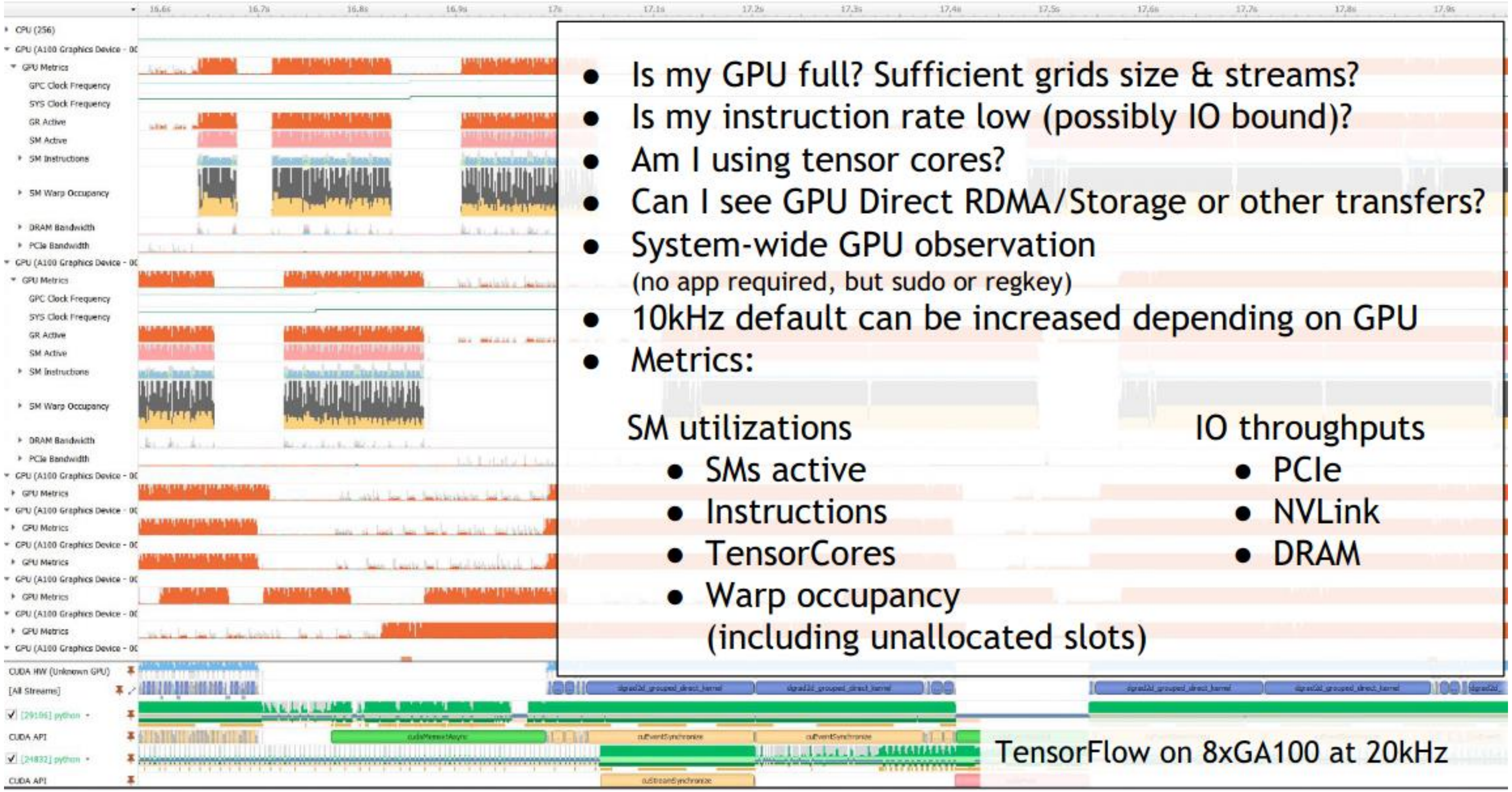
GPU CUDA kernel analysis & debug
Very high freq GPU perf counters
Compare results (diff)
Incredible statistics & customizable

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:

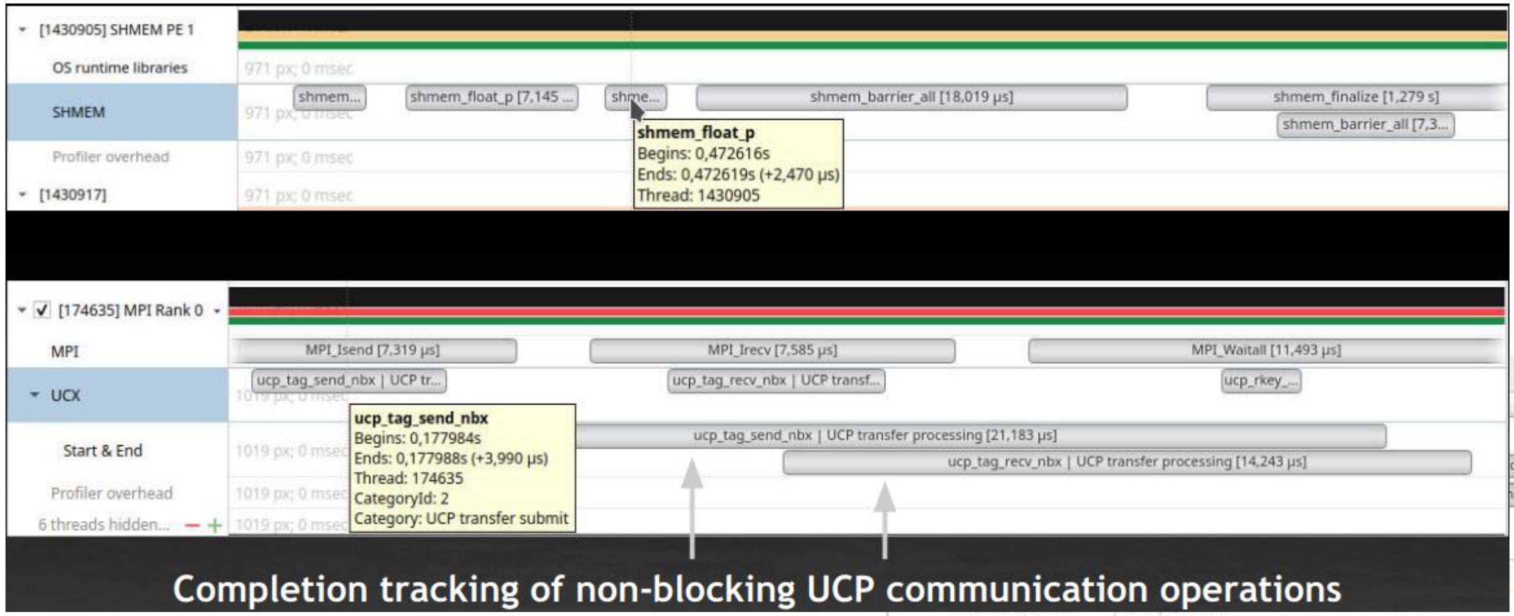
SM utilizations
- SMs active
- Instructions
- TensorCores
- Warp occupancy
  (including unallocated slots)

IO throughputs
- PCIe
- NVLink
- DRAM

TensorFlow on 8xGA100 at 20kHz

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

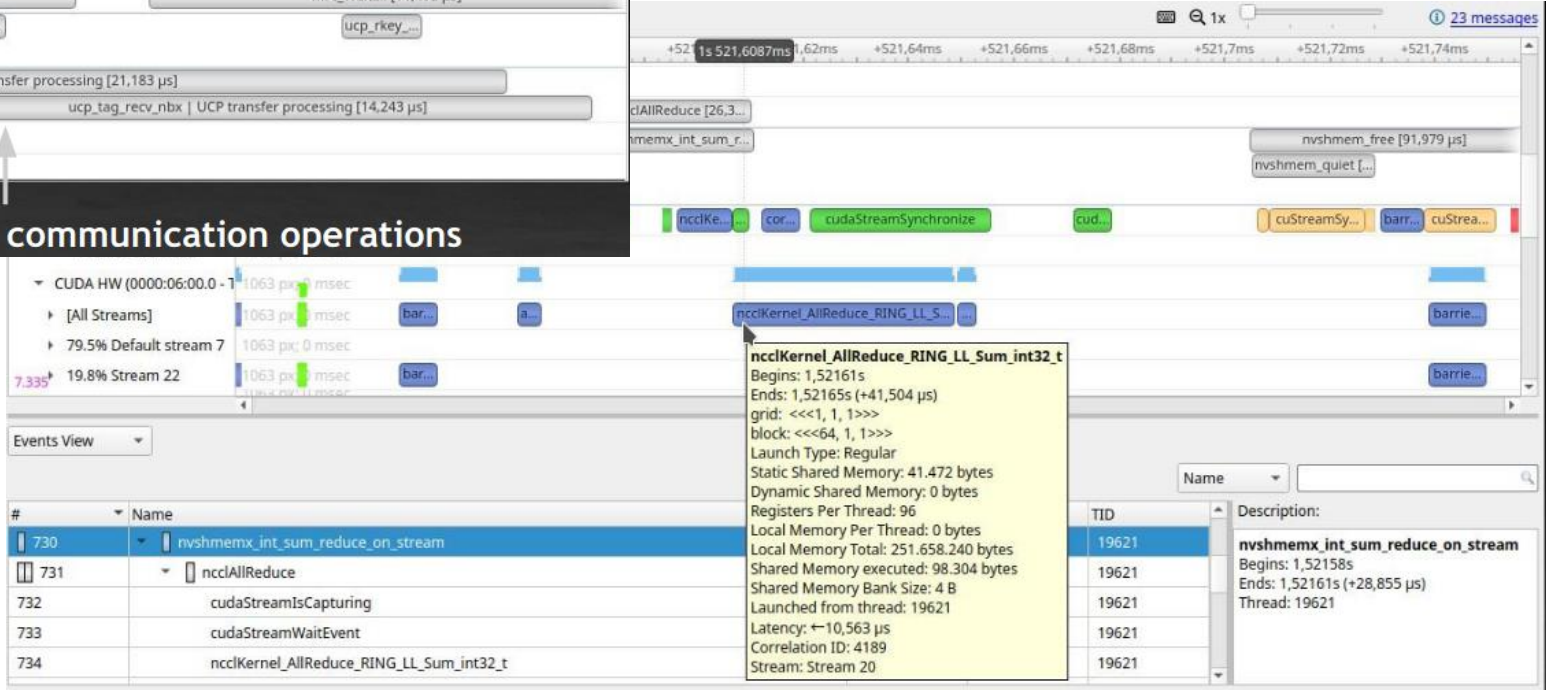
MULTI NODE SUPPORT – SHMEM, MPI, UCX, AND NCCL



shmem_float_p
Begins: 0,472616s
Ends: 0,472619s (+2,470 µs)
Thread: 1430905

ucp_tag_send_nbx
Begins: 0,177984s
Ends: 0,177988s (+3,990 µs)
Thread: 174635
CategoryId: 2
Category: UCP transfer submit

Completion tracking of non-blocking UCP communication operations

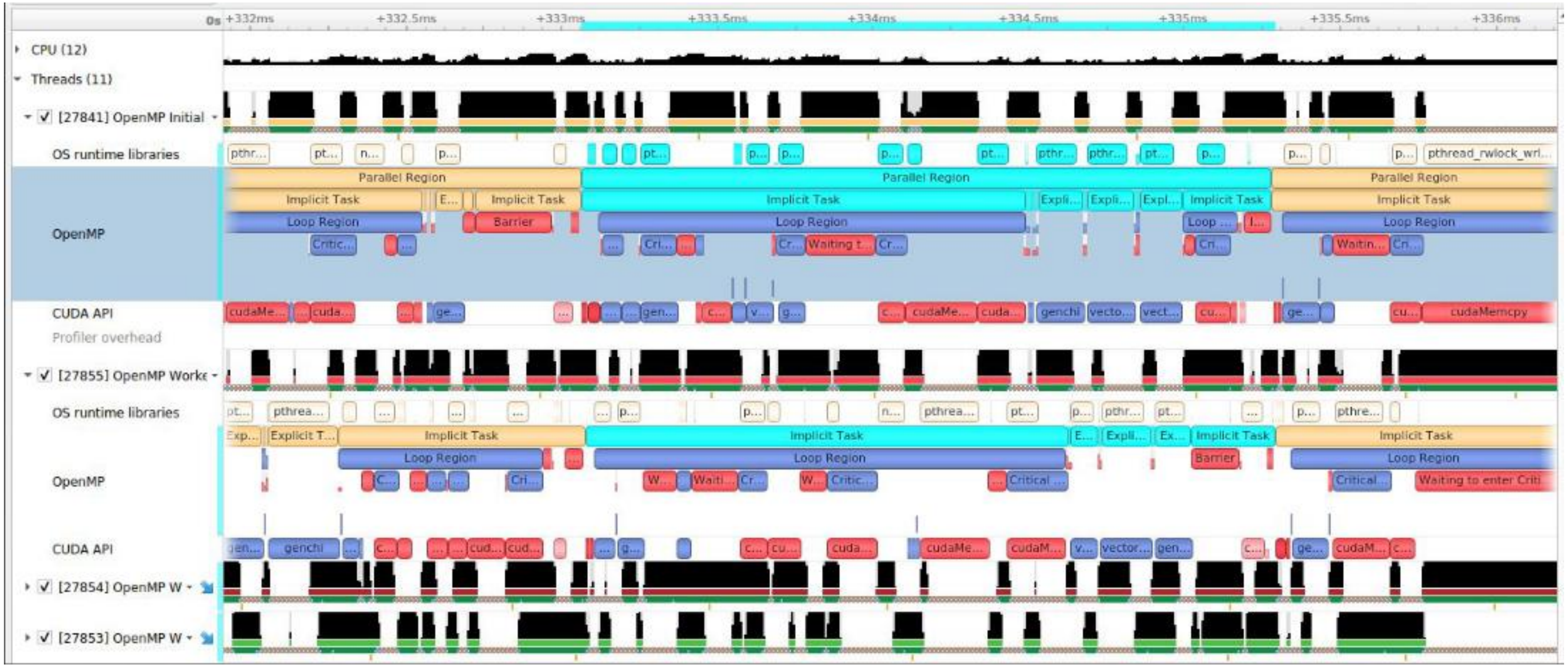


ncclKernel_AllReduce_RING_LL_Sum_int32_t
Begins: 1,52161s
Ends: 1,52165s (+41,504 µs)
grid: <<<1, 1, 1>>>
block: <<<64, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 41.472 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 96
Local Memory Per Thread: 0 bytes
Local Memory Total: 251.658.240 bytes
Shared Memory executed: 98.304 bytes
Shared Memory Bank Size: 4 B
Launched from thread: 19621
Latency: ←10,563 µs
Correlation ID: 4189
Stream: Stream 20

#	Name	TID
730	nvshmemx_int_sum_reduce_on_stream	19621
731	ncclAllReduce	19621
732	cudaStreamIsCapturing	19621
733	cudaStreamWaitEvent	19621
734	ncclKernel_AllReduce_RING_LL_Sum_int32_t	19621

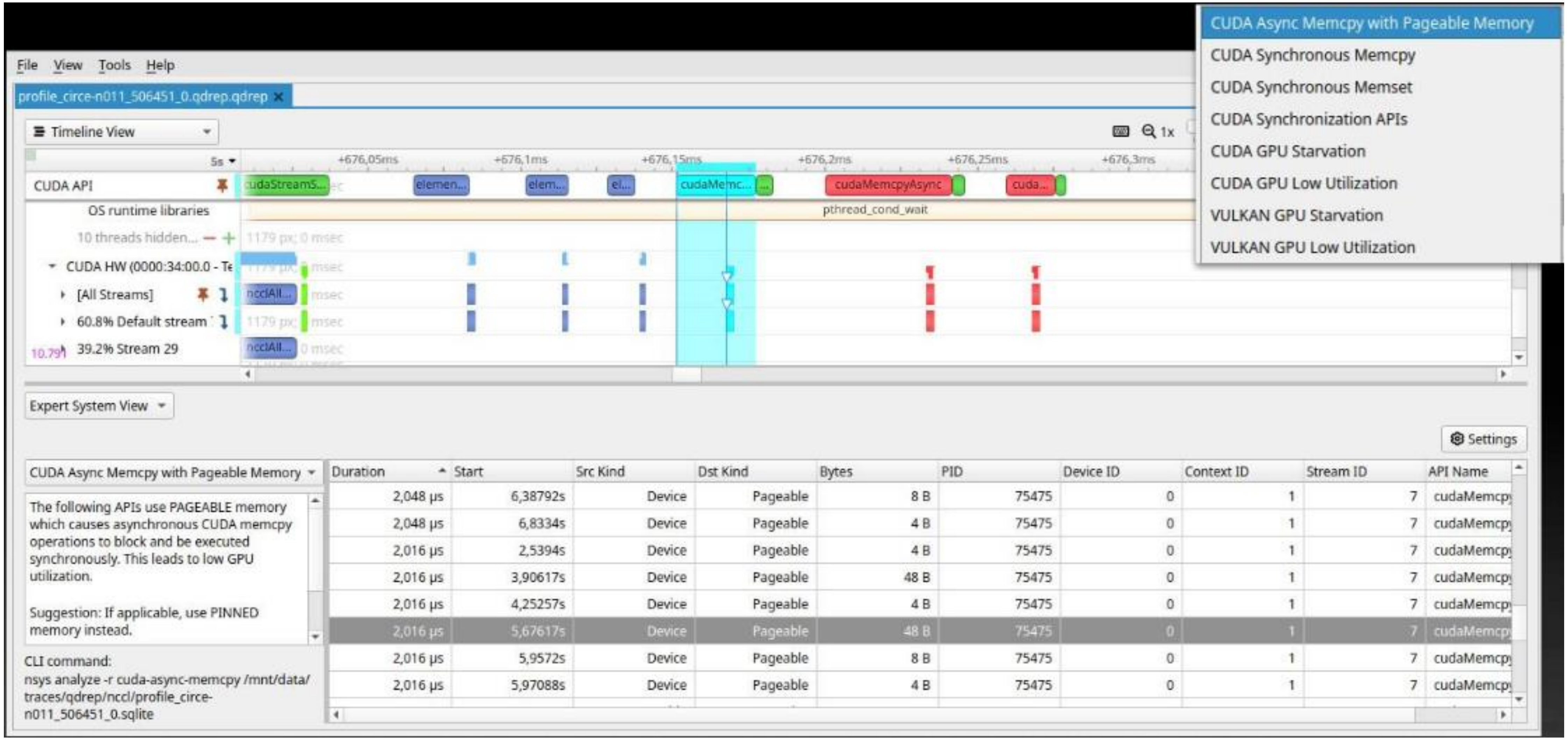
Description:
nvshmemx_int_sum_reduce_on_stream
Begins: 1,52158s
Ends: 1,52161s (+28,855 µs)
Thread: 19621

OPENMP



OMPT-capable OpenMP runtime required

EXPERT SYSTEM



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

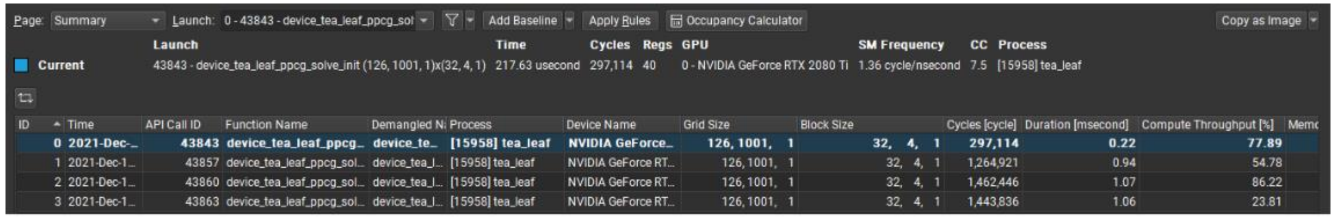
Suggestion: If applicable, use PINNED memory instead.

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

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	1	7	cudaMemcp
2,048 µs	6,8334s	Device	Pageable	4 B	75475	0	1	7	cudaMemcp
2,016 µs	2,5394s	Device	Pageable	4 B	75475	0	1	7	cudaMemcp
2,016 µs	3,90617s	Device	Pageable	48 B	75475	0	1	7	cudaMemcp
2,016 µs	4,25257s	Device	Pageable	4 B	75475	0	1	7	cudaMemcp
2,016 µs	5,67617s	Device	Pageable	48 B	75475	0	1	7	cudaMemcp
2,016 µs	5,9572s	Device	Pageable	8 B	75475	0	1	7	cudaMemcp
2,016 µs	5,97088s	Device	Pageable	4 B	75475	0	1	7	cudaMemcp

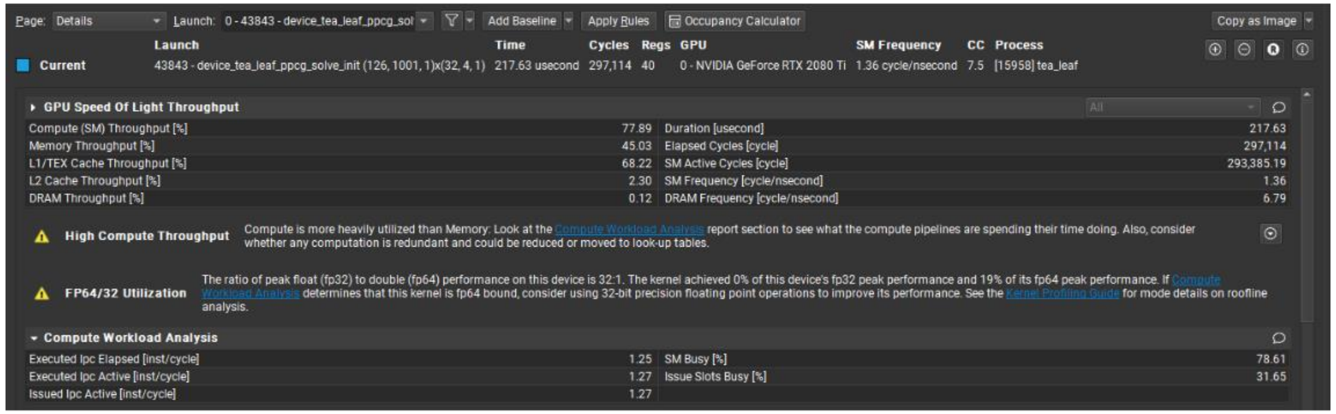
NSIGHT COMPUTE

- 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



	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-…	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_l…	[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_l…	[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_l…	[15958] tea_leaf	NVIDIA GeForce RT…	126, 1001, 1	32, 4, 1	1,443,836	1.06	23.81	



	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

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

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.

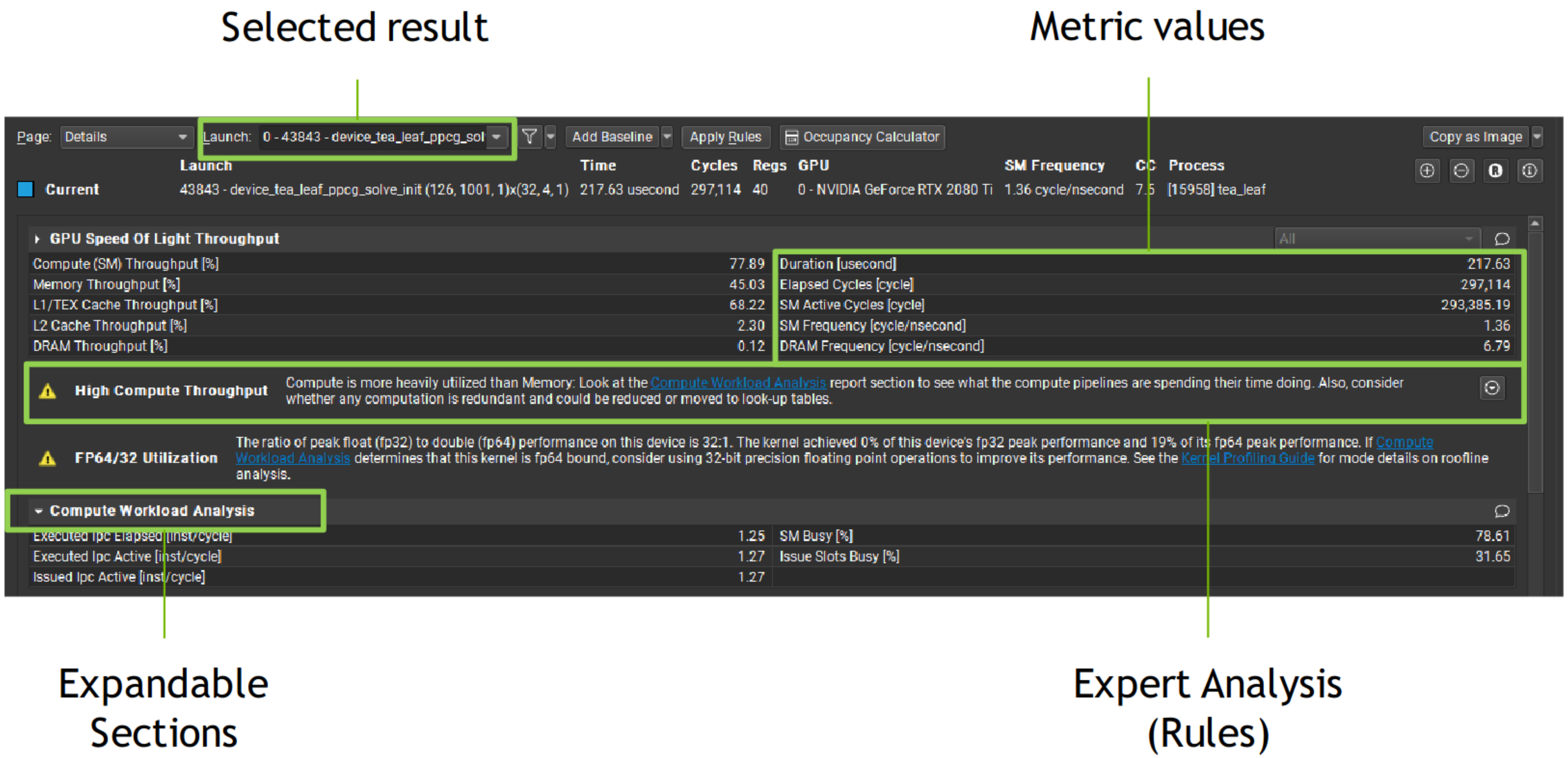
FP64/32 Utilization The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 0% of this device's fp32 peak performance and 19% of its fp64 peak performance. If Compute Workload Analysis determines that this kernel is fp64 bound, consider using 32-bit precision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

Compute Workload Analysis

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

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

PROFILER REPORT



```
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

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

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.

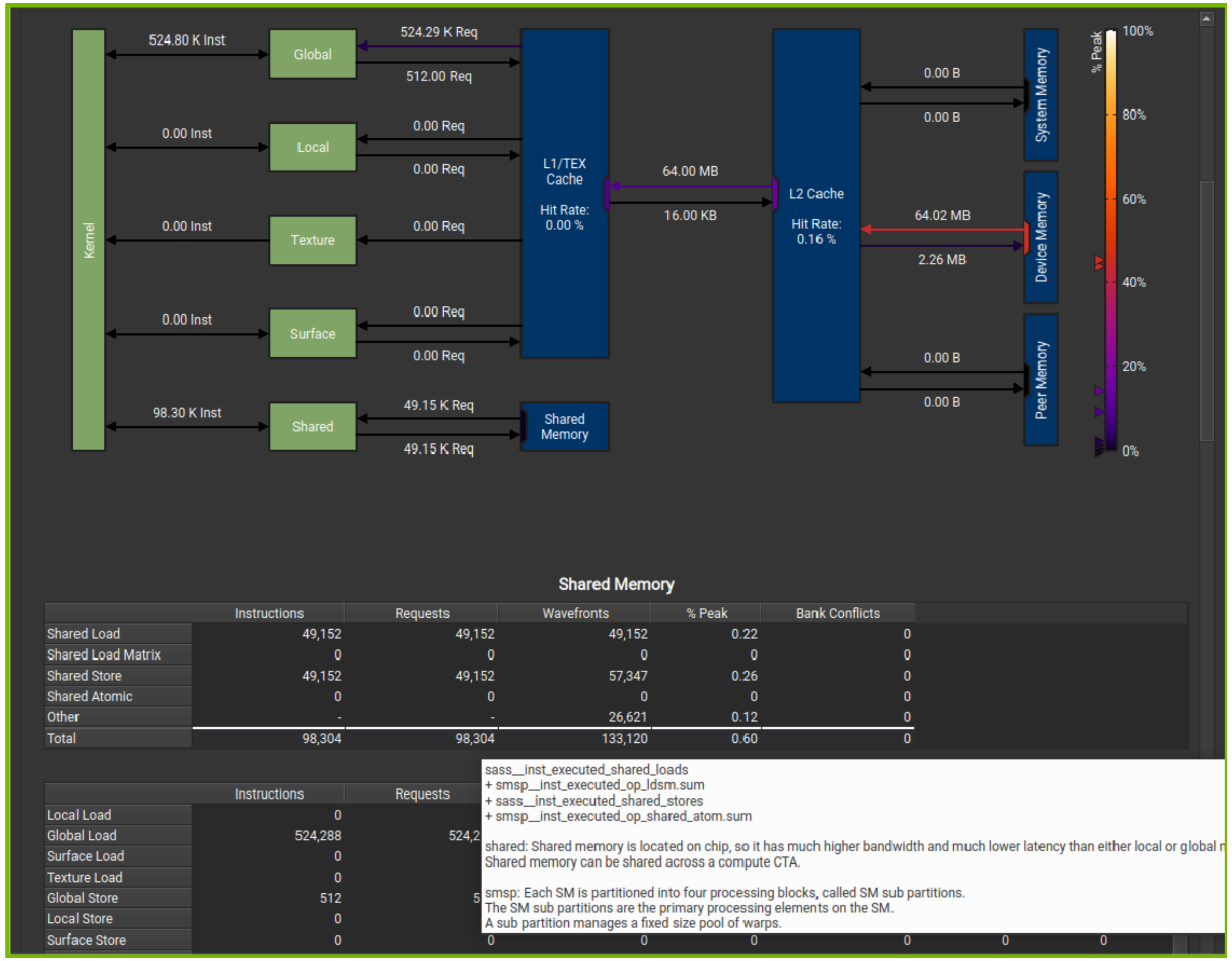
FP64/32 Utilization The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The kernel achieved 0% of this device's fp32 peak performance and 19% of its fp64 peak performance. If Compute Workload Analysis determines that this kernel is fp64 bound, consider using 32-bit precision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

Compute Workload Analysis

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

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



Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	49,152	49,152	49,152	0.22	0
Shared Load Matrix	0	0	0	0	0
Shared Store	49,152	49,152	57,347	0.26	0
Shared Atomic	0	0	0	0	0
Other	-	-	26,621	0.12	0
Total	98,304	98,304	133,120	0.60	0

	Instructions	Requests					
Local Load	0						
Global Load	524,288	524,2					
Surface Load	0						
Texture Load	0						
Global Store	512	5					
Local Store	0						
Surface Store	0	0	0	0	0	0	0

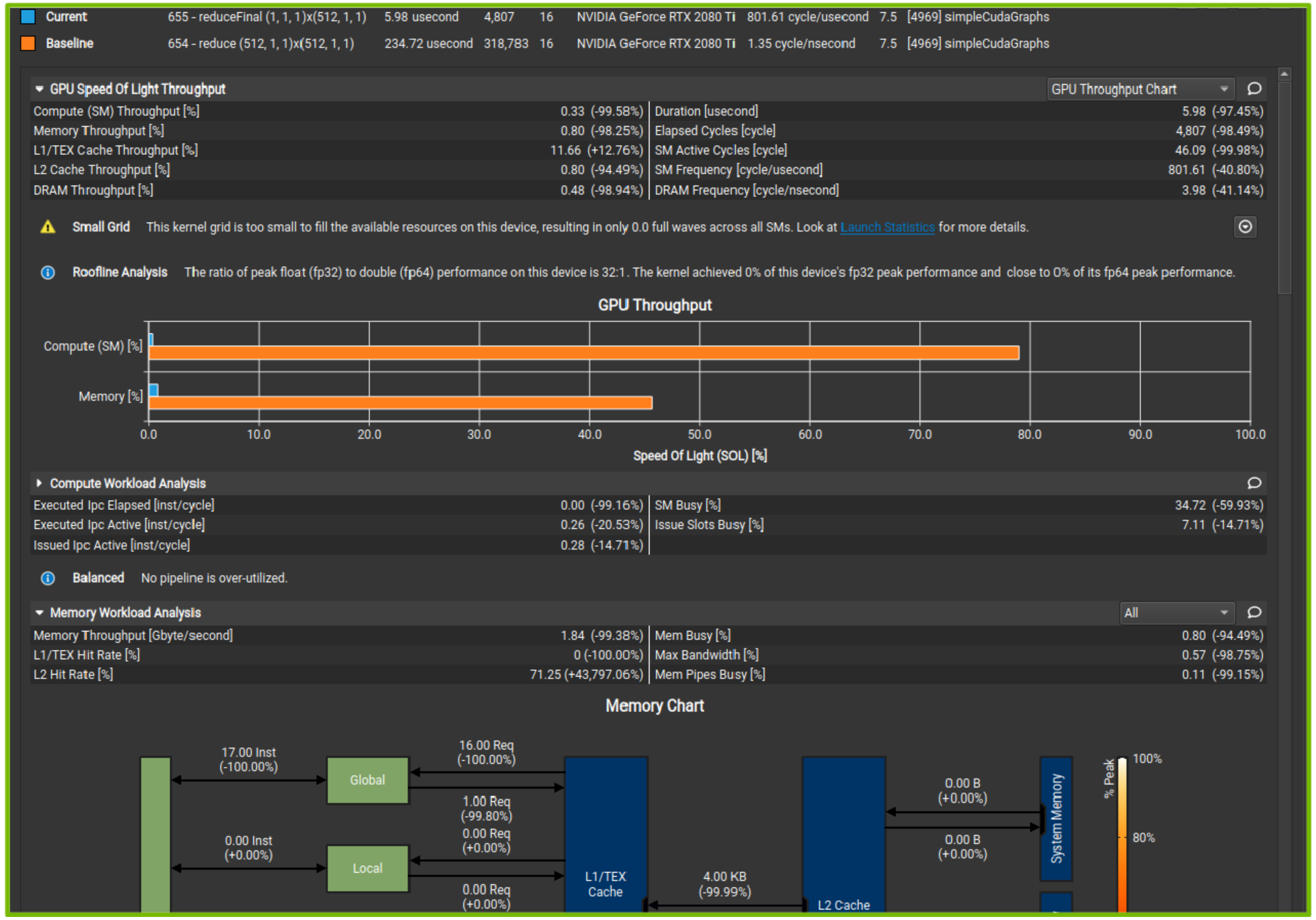
sass__inst_executed_shared_loads
+ smsp__inst_executed_op_ldsm.sum
+ sass__inst_executed_shared_stores
+ smsp__inst_executed_op_shared_atom.sum

shared: Shared memory is located on chip, so it has much higher bandwidth and much lower latency than either local or global r
Shared memory can be shared across a compute CTA.

smsp: Each SM is partitioned into four processing blocks, called SM sub partitions.
The SM sub partitions are the primary processing elements on the SM.
A sub partition manages a fixed size pool of warps.

BASELINE COMPARISON

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



🟦	Current	655 - reduceFinal (1, 1, 1)x(512, 1, 1)	5.98 usecond	4,807	16	NVIDIA GeForce RTX 2080 Ti	801.61 cycle/usecond	7.5	[4969] simpleCudaGraphs
🟧	Baseline	654 - reduce (512, 1, 1)x(512, 1, 1)	234.72 usecond	318,783	16	NVIDIA GeForce RTX 2080 Ti	1.35 cycle/nsecond	7.5	[4969] simpleCudaGraphs

GPU Speed Of Light Throughput — GPU Throughput Chart

Compute (SM) Throughput [%]	0.33 (-99.58%)	Duration [usecond]	5.98 (-97.45%)
Memory Throughput [%]	0.80 (-98.25%)	Elapsed Cycles [cycle]	4,807 (-98.49%)
L1/TEX Cache Throughput [%]	11.66 (+12.76%)	SM Active Cycles [cycle]	46.09 (-99.98%)
L2 Cache Throughput [%]	0.80 (-94.49%)	SM Frequency [cycle/usecond]	801.61 (-40.80%)
DRAM Throughput [%]	0.48 (-98.94%)	DRAM Frequency [cycle/nsecond]	3.98 (-41.14%)

Small Grid This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full waves across all SMs. Look at Launch Statistics for more details.

Roofline Analysis 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 close to 0% of its fp64 peak performance.

Compute Workload Analysis

Executed Ipc Elapsed [inst/cycle]	0.00 (-99.16%)	SM Busy [%]	34.72 (-59.93%)
Executed Ipc Active [inst/cycle]	0.26 (-20.53%)	Issue Slots Busy [%]	7.11 (-14.71%)
Issued Ipc Active [inst/cycle]	0.28 (-14.71%)		

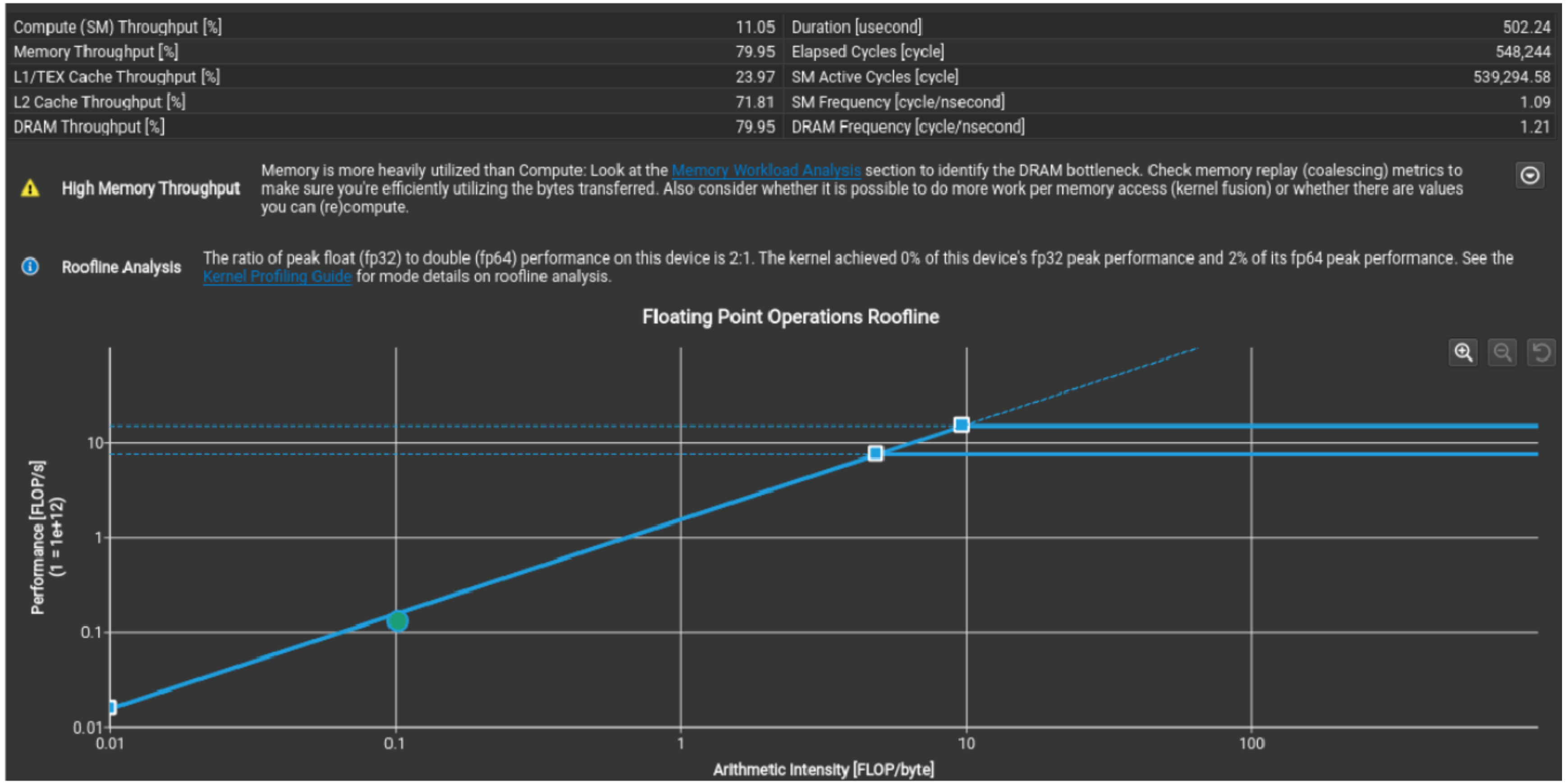
Balanced No pipeline is over-utilized.

Memory Workload Analysis — All

Memory Throughput [Gbyte/second]	1.84 (-99.38%)	Mem Busy [%]	0.80 (-94.49%)
L1/TEX Hit Rate [%]	0 (-100.00%)	Max Bandwidth [%]	0.57 (-98.75%)
L2 Hit Rate [%]	71.25 (+43,797.06%)	Mem Pipes Busy [%]	0.11 (-99.15%)

ROOFLINE ANALYSIS

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



Compute (SM) Throughput [%]	11.05	Duration [usecond]	502.24
Memory Throughput [%]	79.95	Elapsed Cycles [cycle]	548,244
L1/TEX Cache Throughput [%]	23.97	SM Active Cycles [cycle]	539,294.58
L2 Cache Throughput [%]	71.81	SM Frequency [cycle/nsecond]	1.09
DRAM Throughput [%]	79.95	DRAM Frequency [cycle/nsecond]	1.21

High Memory Throughput Memory is more heavily utilized than Compute: Look at the Memory Workload Analysis section to identify the DRAM bottleneck. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can (re)compute.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 2% of its fp64 peak performance. See the Kernel Profiling Guide for mode details on roofline analysis.

Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1e+12)
10
1
0.1
0.01
0.01
0.1
1
10
100
Arithmetic Intensity [FLOP/byte]

DARSHAN

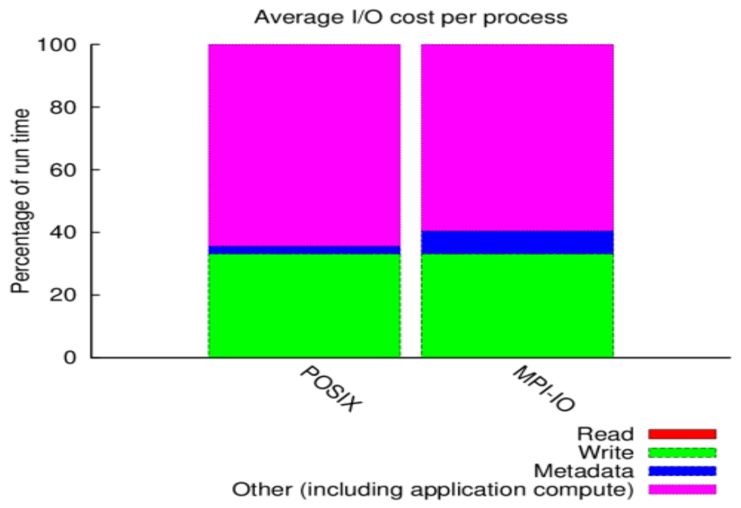
- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

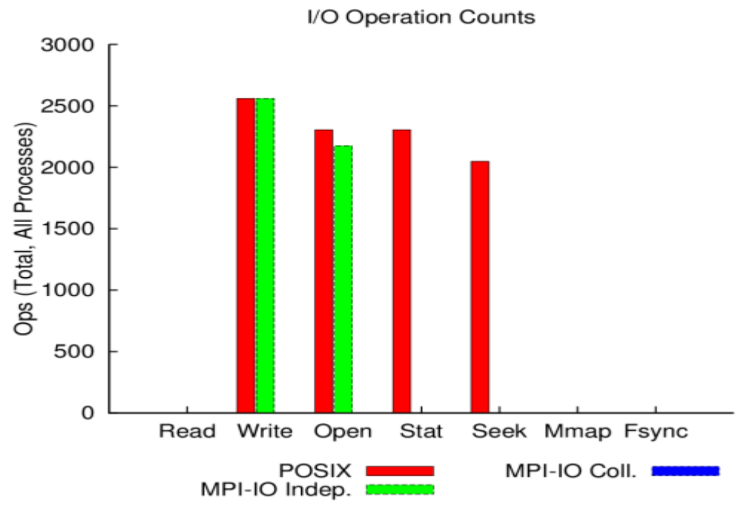
- Supports POSIX, MPI-IO, HDF5, PnetCDF, …
- Binary log file written at exit post-processed into PDF report

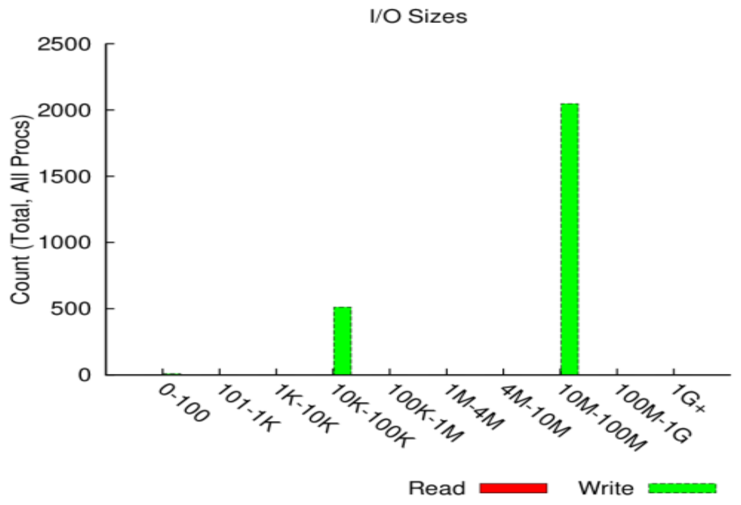
- http://www.mcs.anl.gov/research/projects/darshan/
- Open Source: installed on many HPC systems

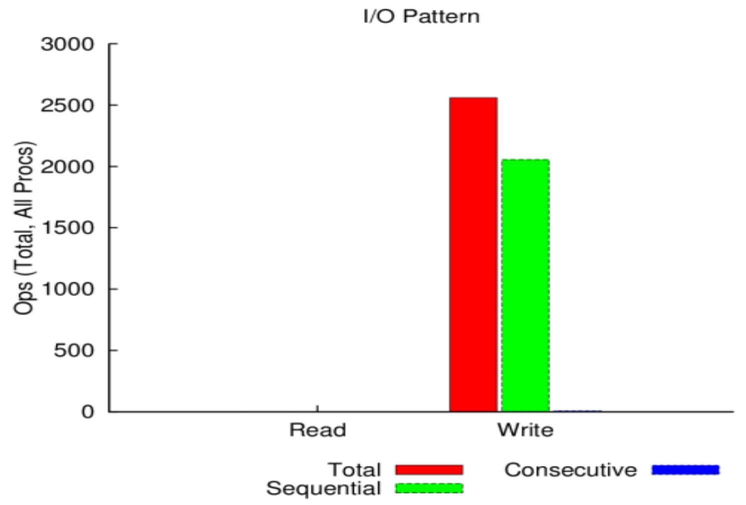
EXAMPLE DARSHAN REPORT EXTRACT

jobid:	uid:	nprocs: 4096	runtime: 175 seconds









PERFORMANCE ANALYSIS RECOMMENDATIONS

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

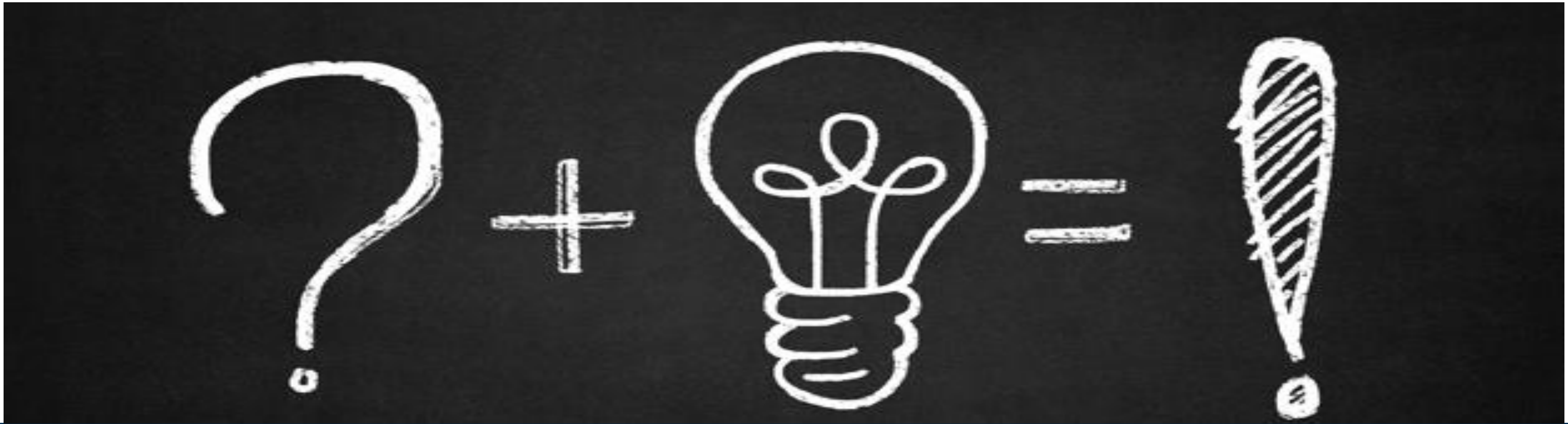
- Get performance overview with Performance Reports
  - CPU Issues:
    - Use MAP, Vtune (on Intel nodes), or uProf (on AMD nodes)
    - Use perf / LIKWID / PAPI
  - MPI Issues: Use Scalasca/Vampir
  - GPU Issues: Use NVIDIA tools
  - I/O Issues: Use DARSHAN

- 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
    - VI-HPS Tuning Workshops
    - POP CoE
  - ATML Application Optimization and User Service Tools
    - EPICURE

☞Successful performance engineering often is a collaborative effort



QUESTIONS