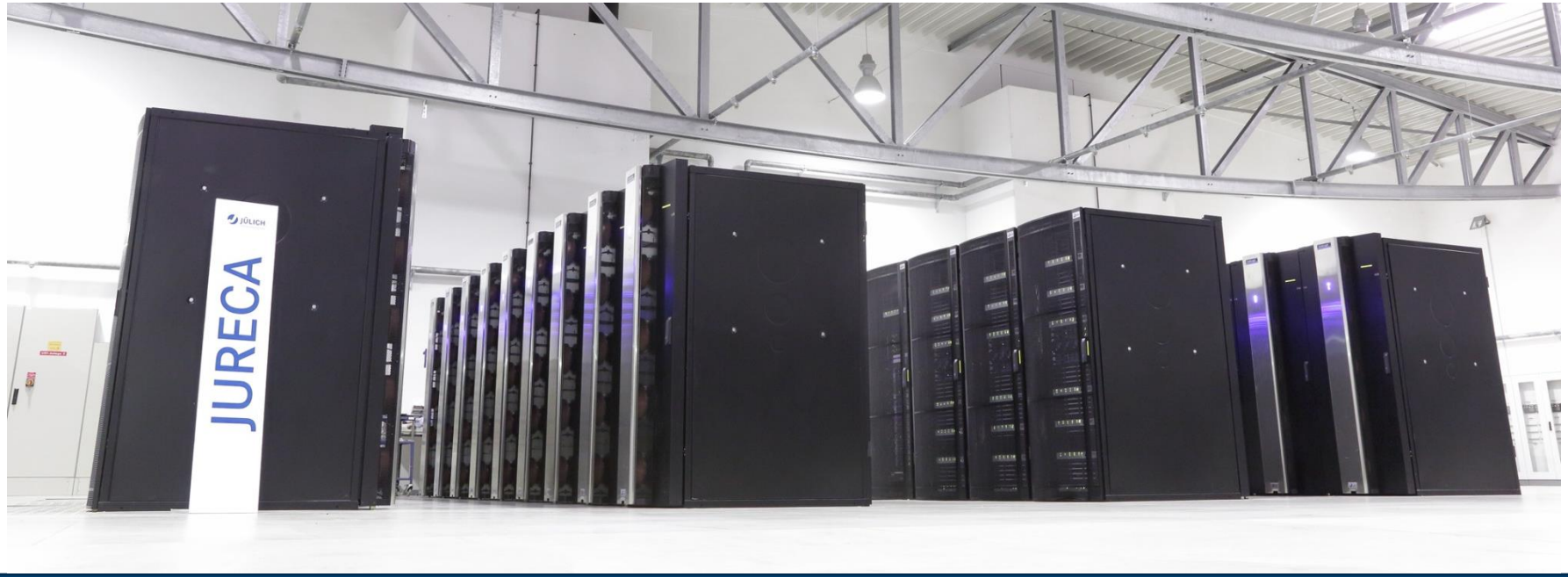


HPC SOFTWARE - TOOLS

Debugger and Performance Analysis

23.11.2018 | MICHAEL KNOBLOCH

OUTLINE

- Local module setup
- Compilers
- Libraries

Debugger:

- TotalView / DDT
- MUST
- Intel Inspector

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

Kent Beck

Performance Tools:

- Score-P
- Scalasca
- Vampir
- Intel Vtune Amplifier
- Intel Advisor
- Performance Reports
- TAU
- NVIDIA Visual Profiler
- Darshan
- PAPI

MODULE SETUP & COMPILER

THE MODULE SETUP

- Tools are available through “modules”
  - Allows to easily manage different versions of programs
  - Works by dynamic modification of a user's environment

- Module setup based on EasyBuild and lmod
  - Staged, hierarchical setup
  - Automatically manages dependencies via toolchains

- Consistent setup on JURECA (cluster & booster) and JEWELS

MOST IMPORTANT MODULE COMMANDS

`module`

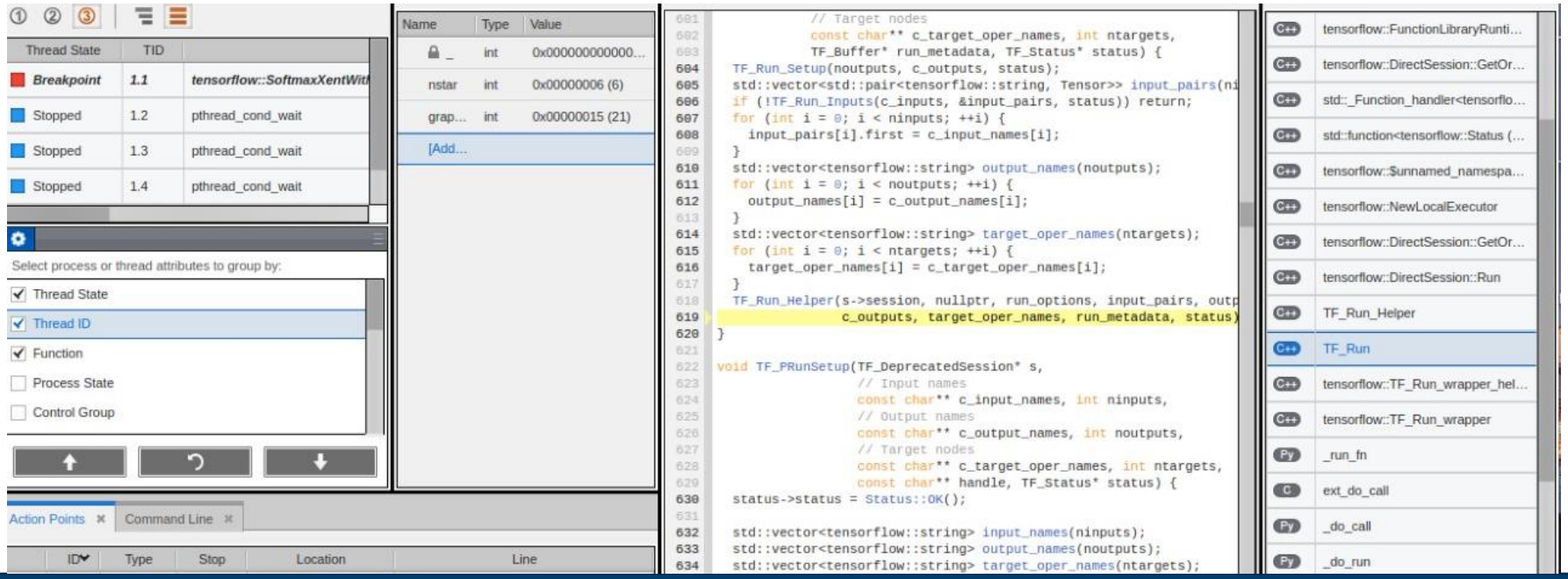
- `spider` # show all products
- `spider` *`product`* # show product details
- `avail` # show all available products
- `list` # list loaded products

- `load` *`product(s)`* # setup access to product
- `unload` *`product(s)`* # release access
- `swap` *`product1 product2`* # replace v1 of product with v2

- `whatis` *`product(s)`* # print short description
- `help` *`product(s)`* # print longer description
- `show` *`product(s)`* # show what “settings” are performed

COMPILER AND MPI LIBRARIES

- Compiler
  - Intel C/C++ and Fortran compiler
  - GNU C/C++ and Fortran compiler
  - PGI C/C++ and Fortran compiler
  - Clang C/C++ compiler
  - NVIDIA CUDA compiler
- MPI libraries
  - Intel MPI
  - Parastation MPI
  - MVAPICH MPI (CUDA aware)
  - Open MPI


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
nstar
int
0x00000006 (6)
grap...
int
0x00000015 (21)
[Add...
Action Points
Command Line
ID
Type
Stop
Location
Line
// 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);
tensorflow::FunctionLibraryRunti...
tensorflow::DirectSession::GetOr...
std::_Function_handler<tensorflo...
std::function<tensorflow::Status (...
tensorflow::$unnamed_namespa...
tensorflow::NewLocalExecutor
tensorflow::DirectSession::GetOr...
tensorflow::DirectSession::Run
TF_Run_Helper
TF_Run
tensorflow::TF_Run_wrapper_hel...
tensorflow::TF_Run_wrapper
_run_fn
ext_do_call
_do_call
_do_run

DEBUGGER

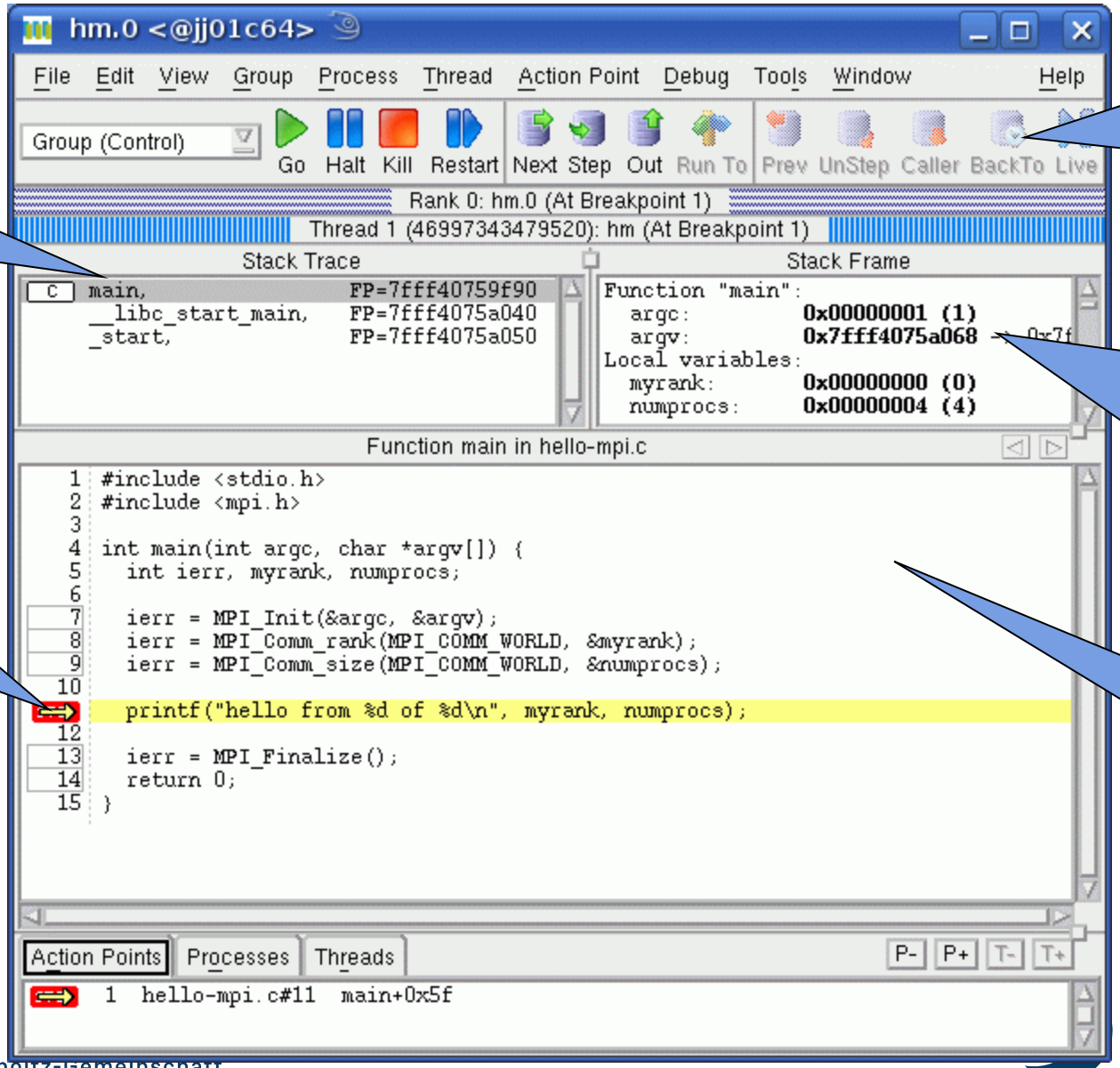
DEBUGGING TOOLS (STATUS: NOV 2018)

- **Debugger:**
  - TotalView
  - DDT

- **Correctness Checker:**
- Intel Inspector
- MUST

- UNIX Symbolic Debugger
  for C, C++, F77, F90, PGI HPF, assembler programs
- “Standard” debugger
- Special, non-traditional features
  - Multi-process and multi-threaded
  - C++ support (templates, inheritance, inline functions)
  - F90 support (user types, pointers, modules)
  - 1D + 2D Array Data visualization
  - Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, pthreads)
  - Scripting and batch debugging
  - Memory Debugging
  - CUDA and OpenACC support
- http://www.roguewave.com
- NOTE: License limited to 2048 processes (shared between all users)

TOTALVIEW: MAIN WINDOW



Stack trace

Toolbar for common options

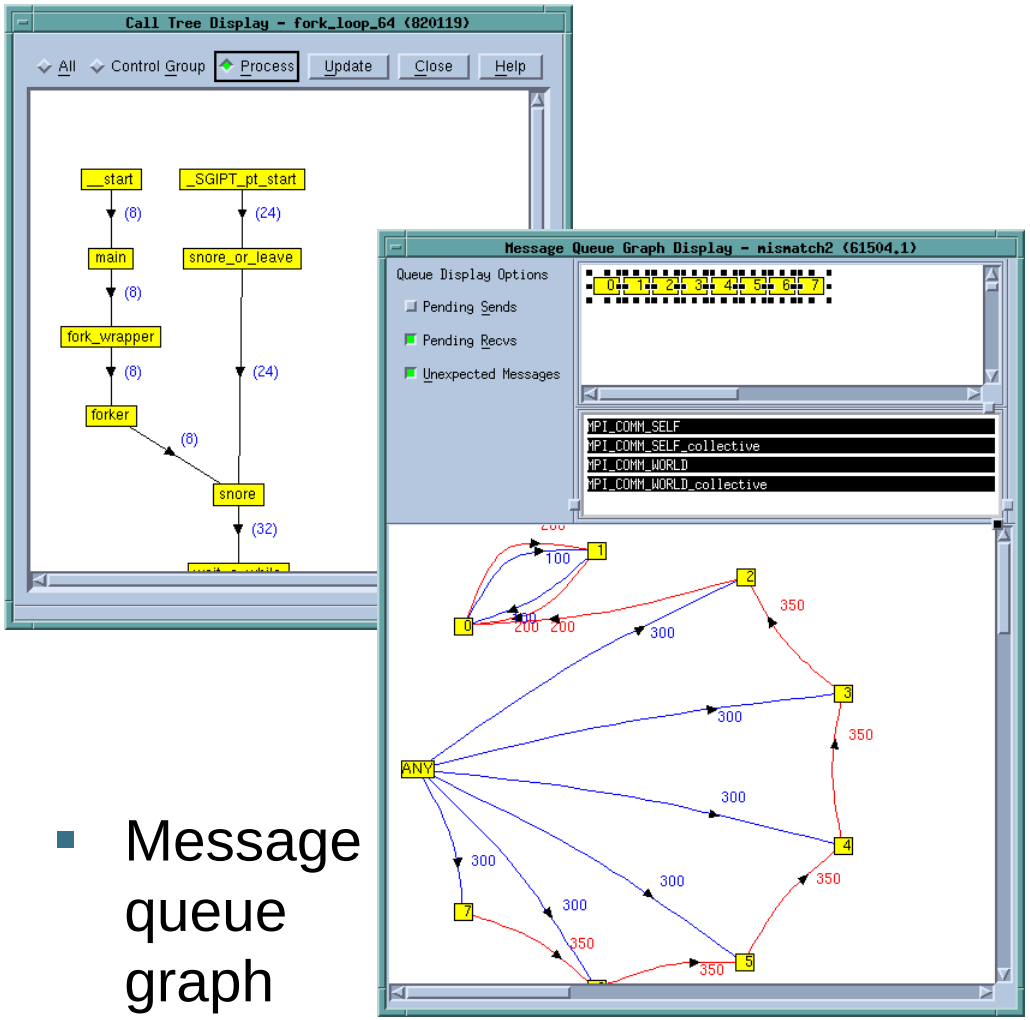
Local variables for selected stack frame

Break points

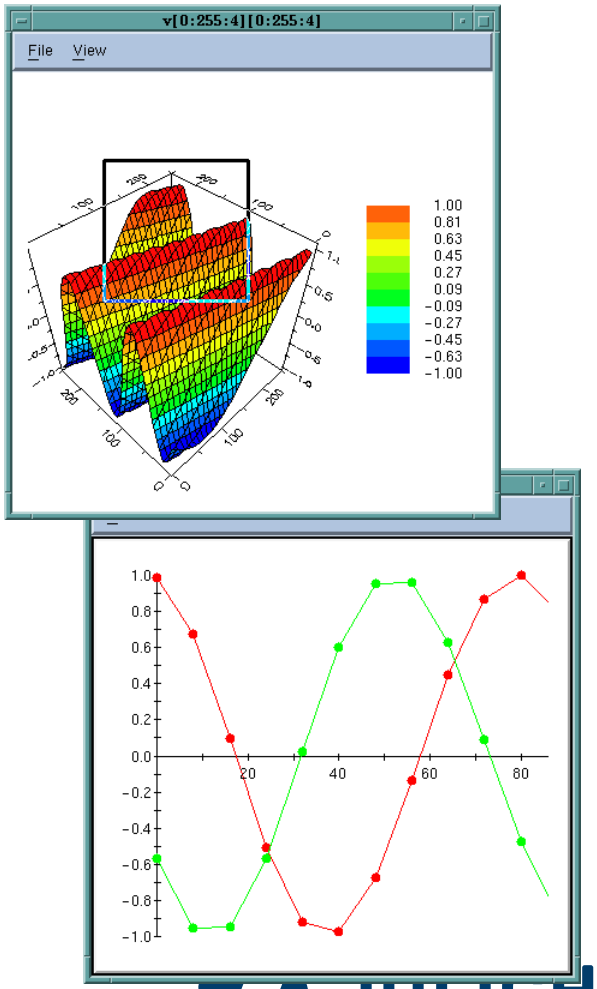
Source code window

TOTALVIEW: TOOLS MENU

- Call Graph



- Data visualization

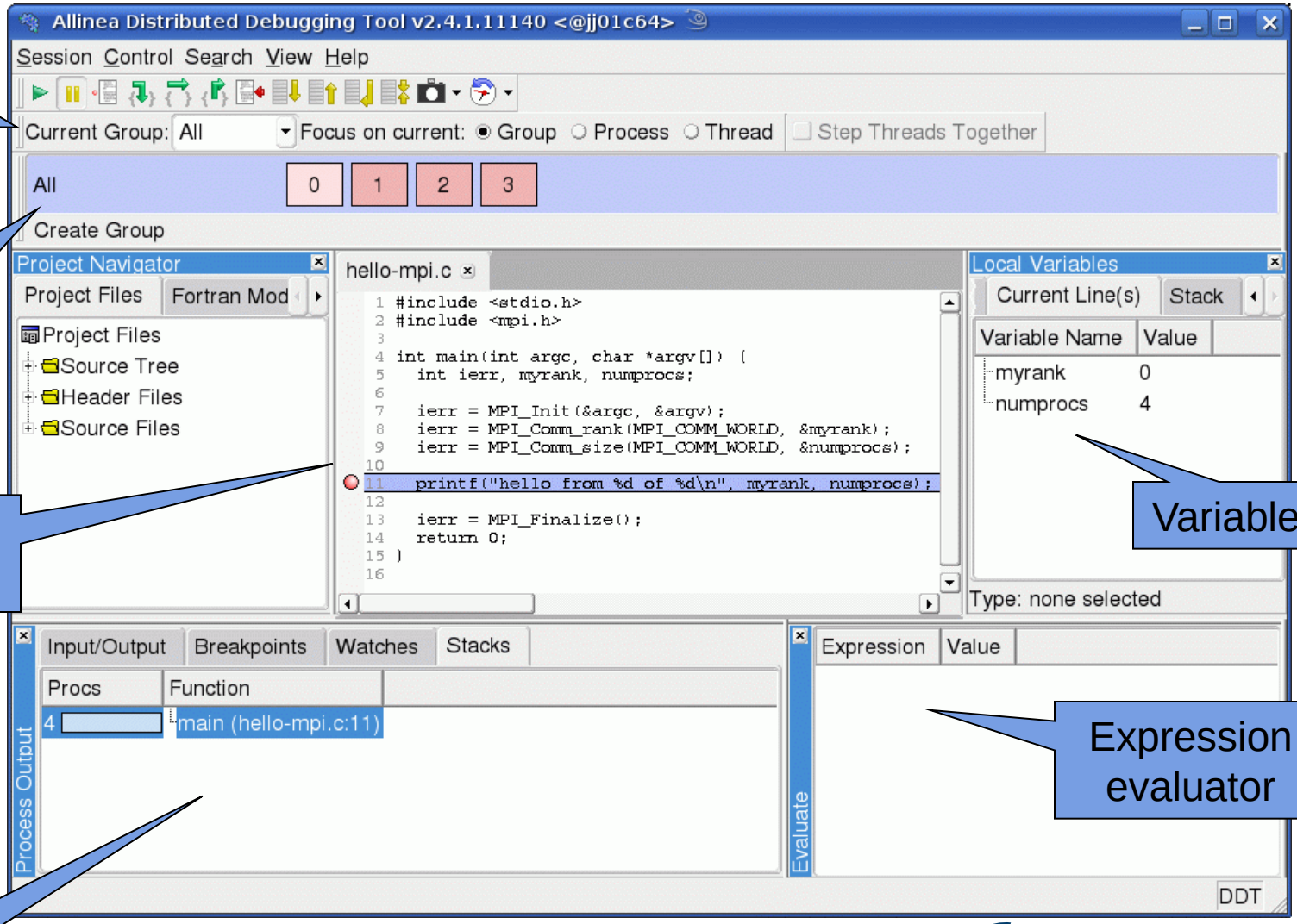


- Message queue graph

allinea DDT

- UNIX Graphical Debugger for C, C++, F77, F90 programs
- Modern, easy-to-use debugger
- Special, non-traditional features
  - Multi-process and multi-threaded
  - 1D + 2D 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
- http://www.allinea.com
- NOTE: License limited to 64 processes (shared between all users)

DDT: MAIN WINDOW



Process controls

Process groups

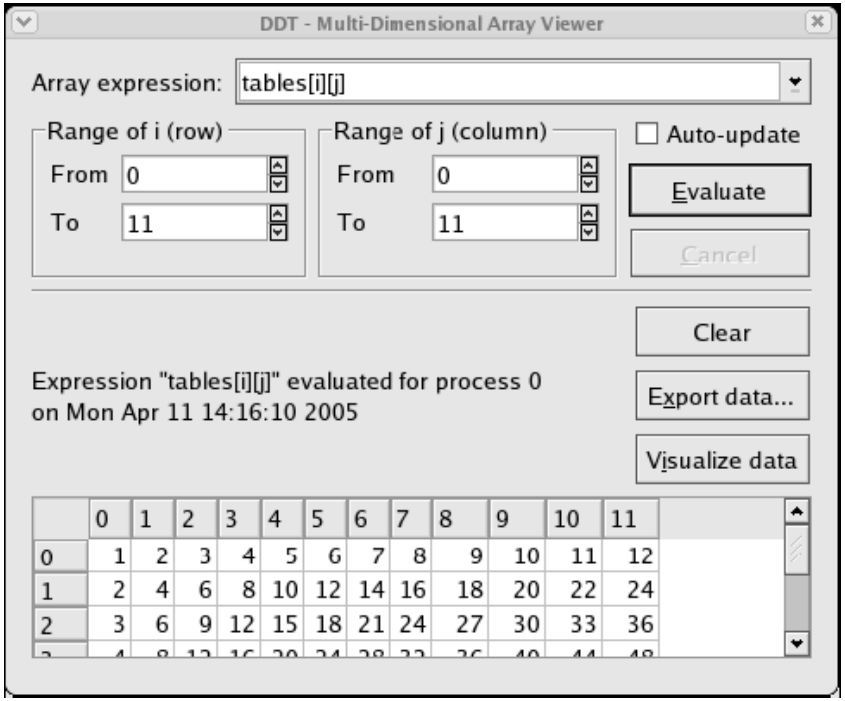
Source code

Variables

Expression evaluator

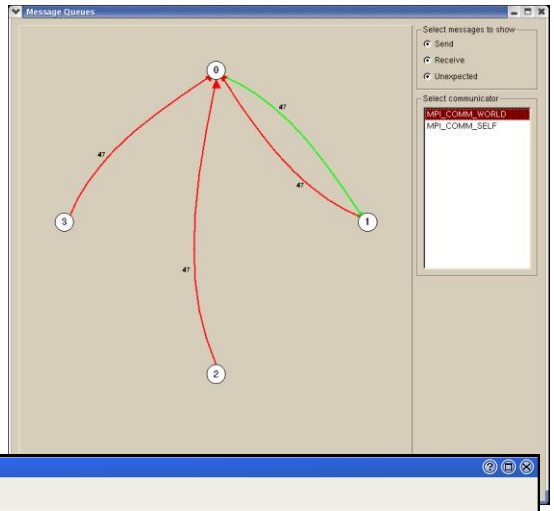
Stack trace

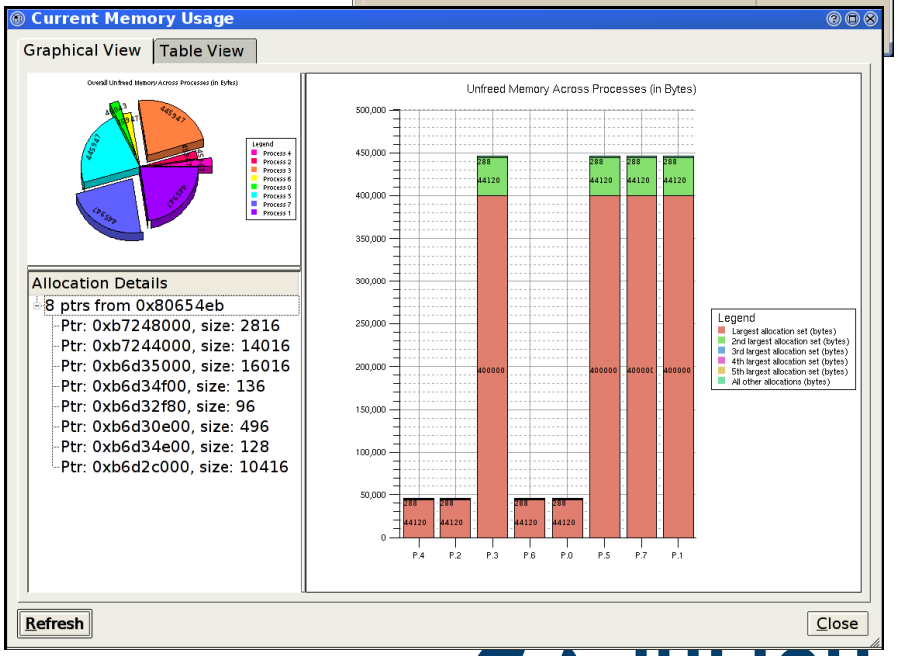
DDT: NON-STANDARD FEATURES



- Multi-Dimensional Array Viewer

- Message queue graph

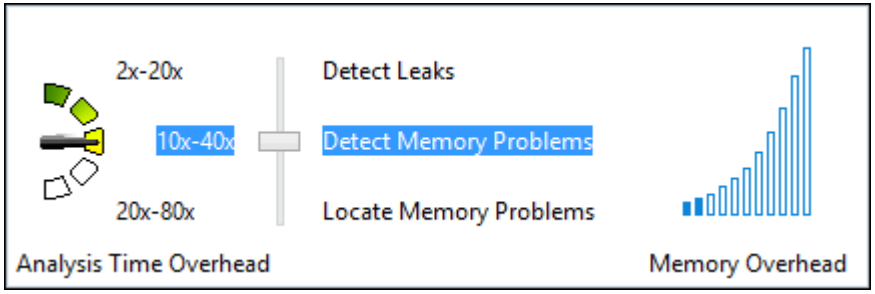




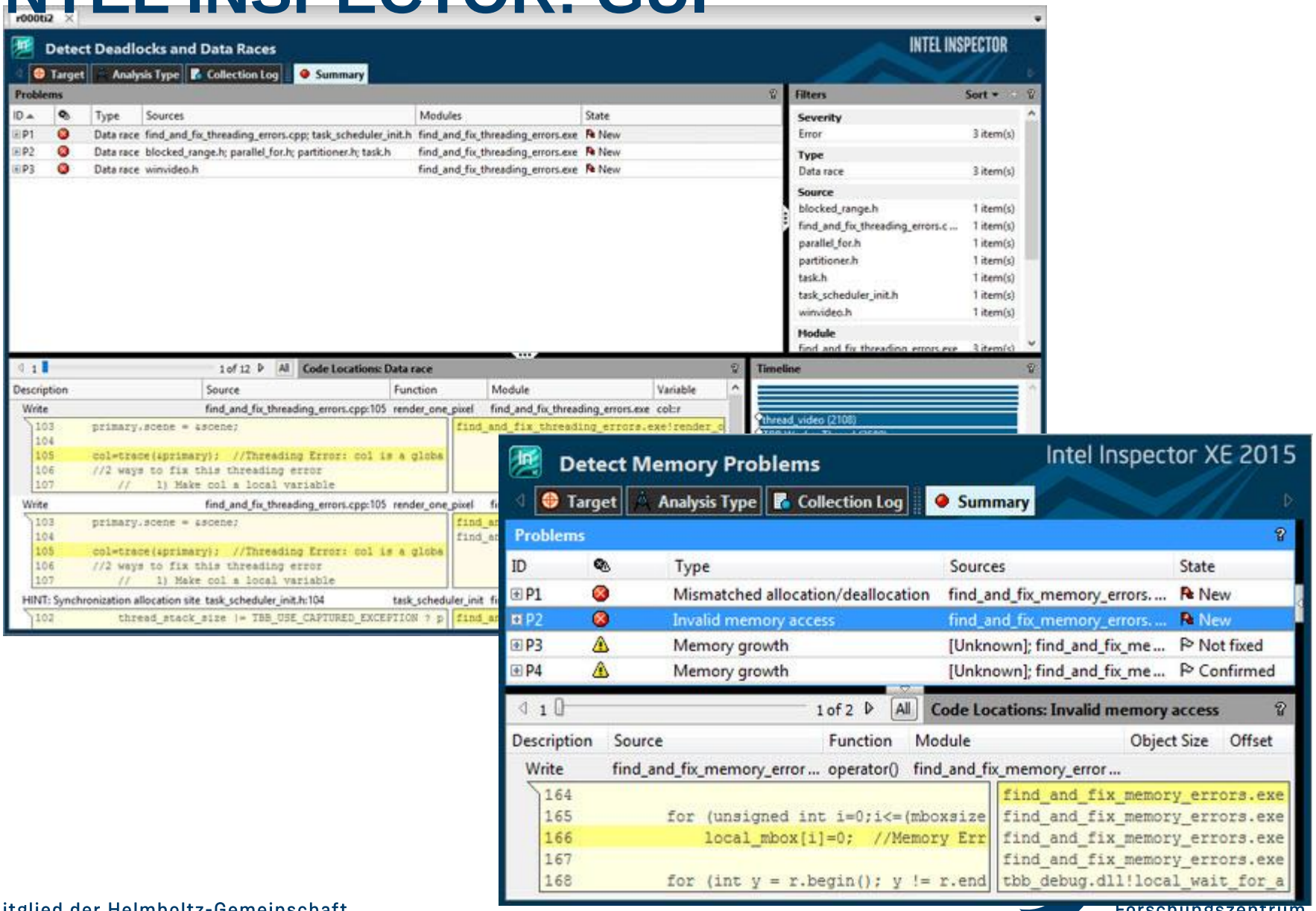
- Memory Usage

INTEL INSPECTOR

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



INTEL INSPECTOR: GUI



MUST

- Next generation MPI correctness and portability checker
- http://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

- MUST reports
  - Errors: violations of the MPI-standard
  - Warnings: unusual behavior or possible problems
  - Notes: harmless but remarkable behavior
  - Further: potential deadlock detection
- Usage
  - Relink application with mustc, mustcxx, mustf90, …
  - Run application under the control of mustrun (requires one additional MPI process)
  - See MUST_Output.html report

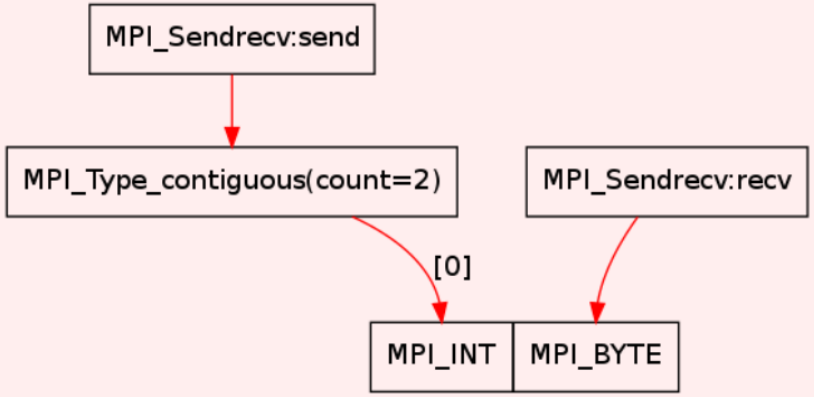
MUST DATATYPE MISMATCH

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

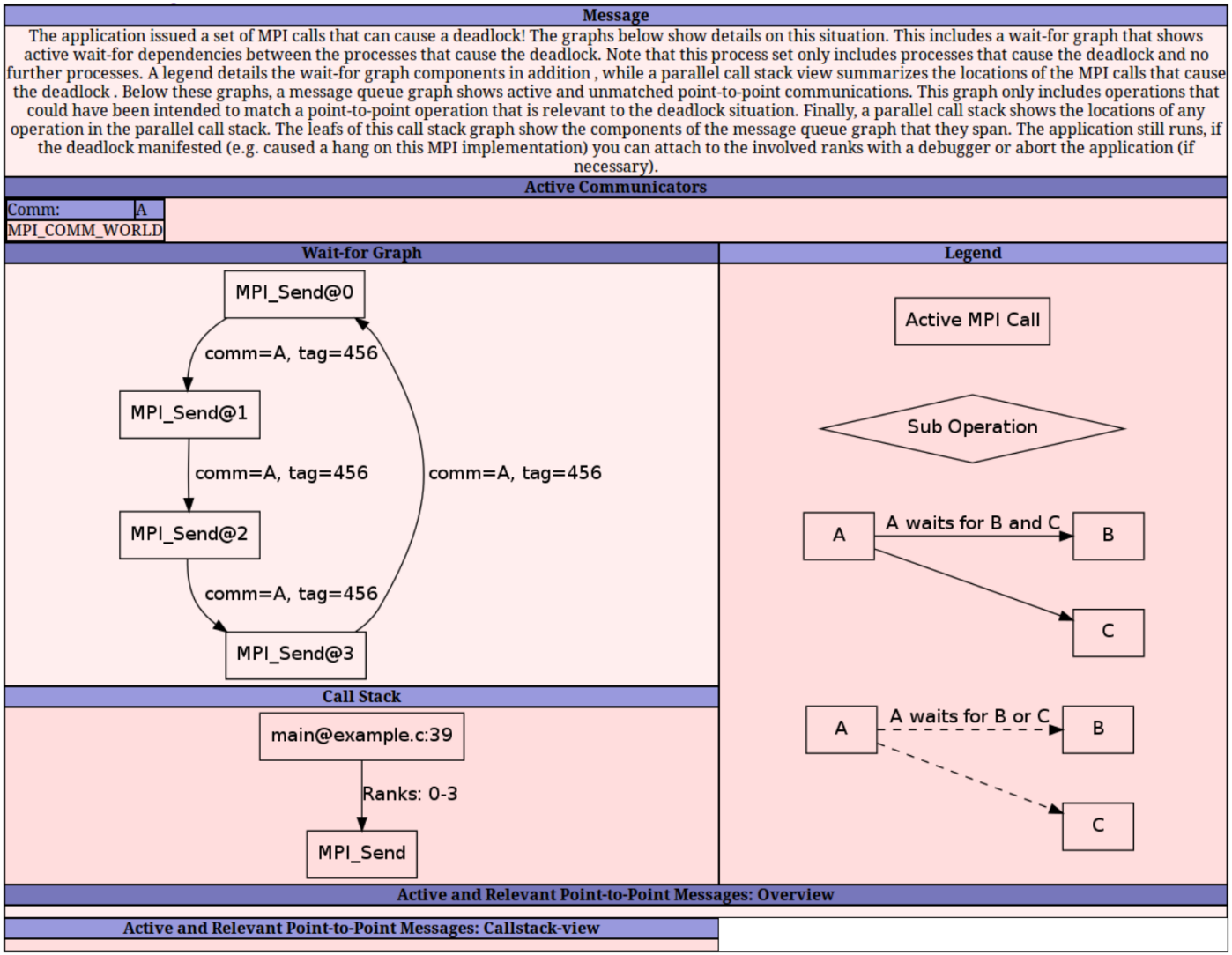
Message

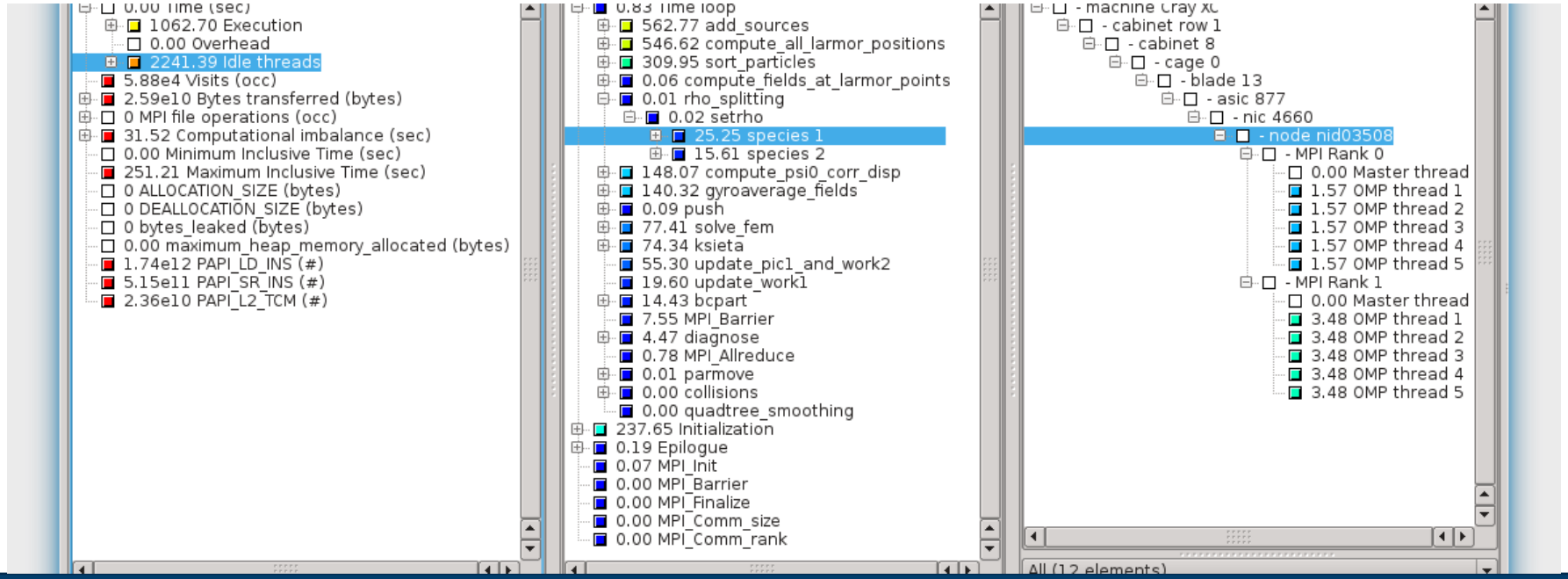
The application issued a set of MPI calls that mismatch in type signatures! The graph below shows details on this situation. The first differing item of each involved communication request is highlighted.

Datatype Graph



MUST DEADLOCK DETECTION



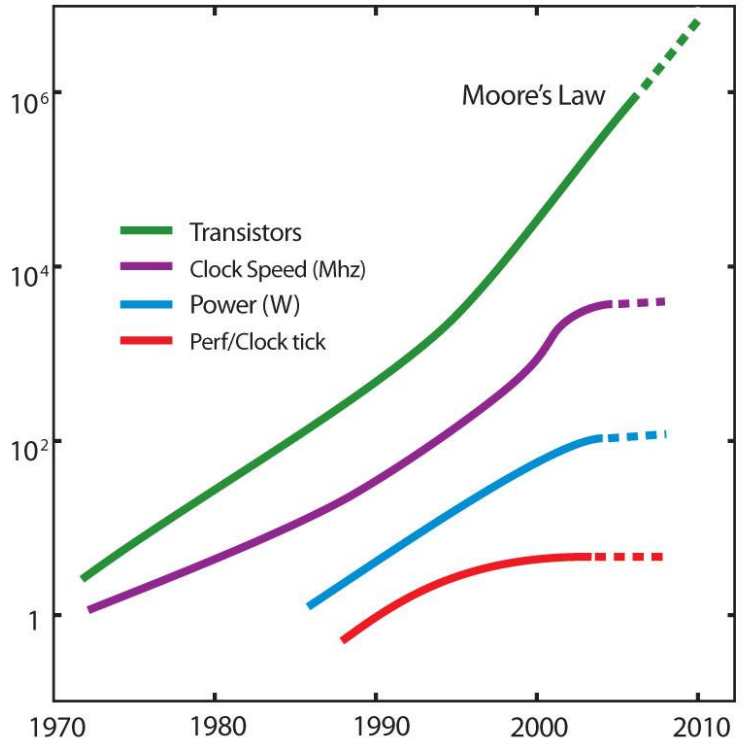


PERFORMANCE ANALYSIS TOOLS

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
    - More CPUs / multi-core

☞ 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 layout
  - Input / output
    - ☞Often not given enough attention
    - ☞Parallel I/O matters

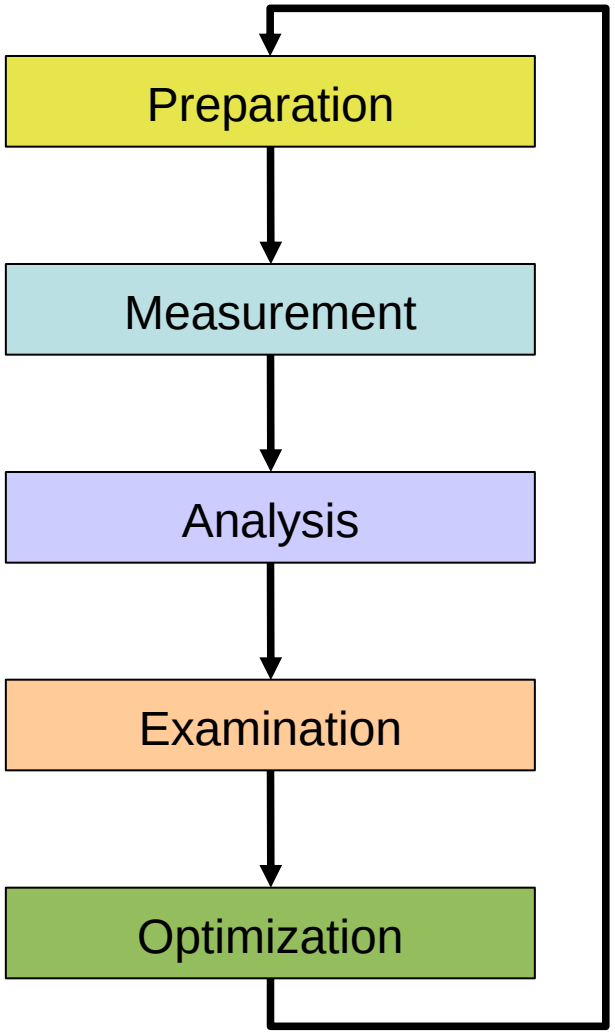
PERFORMANCE FACTORS

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

TUNING BASICS

- Successful engineering is a combination of
  - The right algorithms and libraries
  - Compiler flags and directives
    - ☞Thinking !!!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - ☞After each step!

PERFORMANCE ENGINEERING WORKFLOW



- Prepare application (with symbols), insert extra code (probes/hooks)
- Collection of data relevant to execution performance analysis
- Calculation of metrics, identification of performance metrics
- Presentation of results in an intuitive/understandable form
- Modifications intended to eliminate/reduce performance problems

THE 80/20 RULE

- Programs typically spend 80% of their time in 20% of the code
  - ☞ *Know what matters!*
- Developers typically spend 20% of their effort to get 80% of the total speedup possible for the application
  - ☞ *Know when to stop!*
- Don't optimize what does not matter
  - ☞ *Make the common case fast!*

MEASUREMENT TECHNIQUES

A Classification

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation
- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing
- How is performance data analyzed?
  - Online
  - Post mortem

PROFILING / RUNTIME SUMMARIZATION

- Recording of aggregated information
  - Total, maximum, minimum, …
- For measurements
  - Time
  - Counts
    - Function calls, Bytes transferred, Hardware counters
- Over program and system entities
  - Functions, call sites, basic blocks, loops, …
  - Processes, threads

☞ *Profile = summarization of events over execution interval*

TRACING

- Recording information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events

☞ *Event trace = Chronologically ordered sequence of event records*

TRACING VS. PROFILING

- Tracing advantages
  - Event traces preserve the **temporal** and **spatial** relationships among individual events (☞ context)
  - Allows reconstruction of **dynamic** application behaviour on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Writing events to file at runtime causes perturbation
  - Writing tracing software is complicated
    - Event buffering, clock synchronization, ...

CRITICAL ISSUES

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

☞ *Tradeoff: Accuracy vs. Expressiveness of data*

TYPICAL PERFORMANCE ANALYSIS PROCEDURE

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements
- ***What*** is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling
- ***Where*** is the key bottleneck?
  - Call-path profiling, detailed basic block profiling
- ***Why*** is it there?
  - Hardware counter analysis
  - Trace selected parts (to keep trace size manageable)
- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function, performance modeling

REMARK: NO SINGLE SOLUTION IS SUFFICIENT!

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

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

PERFORMANCE TOOLS (STATUS: NOV 2018)

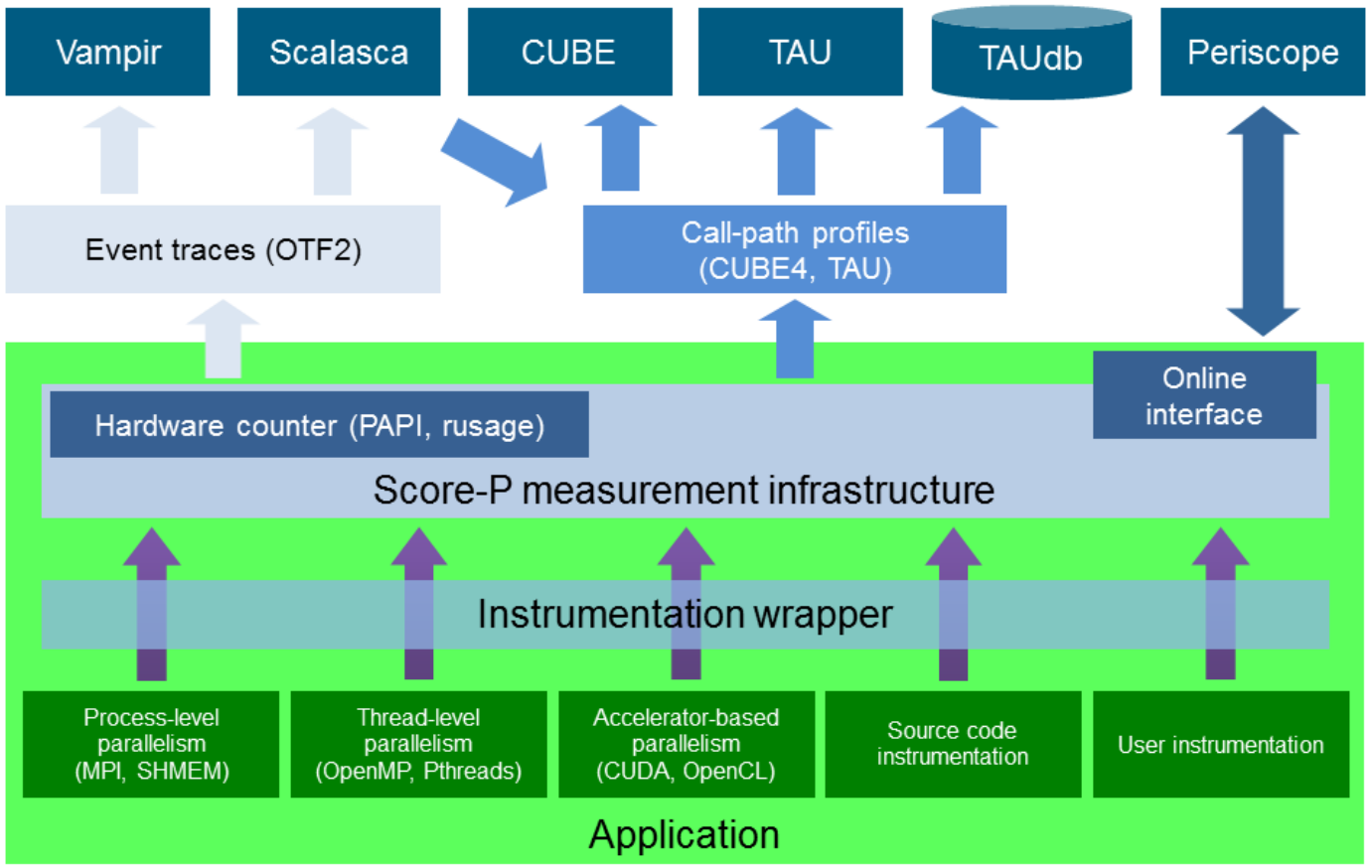
- Score-P
- Scalasca 2
- Vampir[Server]
- HPCToolkit
- Allinea Performance Reports
- Darshan
- NVIDIA Visual Profiler
- TAU
- Intel VTune Amplifier XE
- Intel Advisor
- mpiP*
- Extrae/Paraver*
- PAPI*

SCORE-P

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

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

SCORE-P OVERVIEW



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

WHAT IS THE KEY BOTTLENECK?

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

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

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

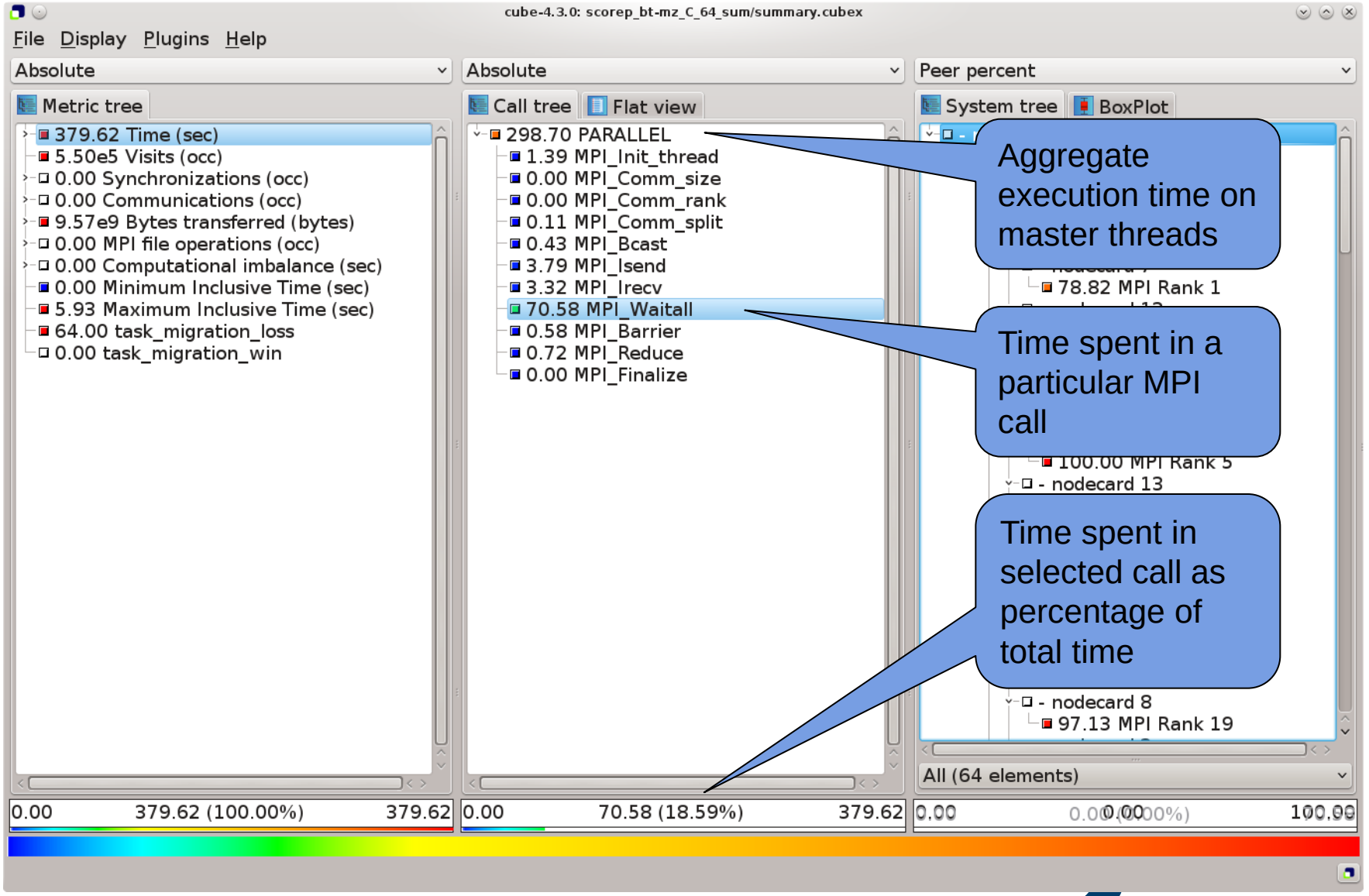
FLAT MPI PROFILE: RECIPE

1. Prefix your ***link command*** with
   "`scorep --nocompiler`"

2. Prefix your MPI ***launch command*** with
   "`scalasca -analyze`"

3. After execution, examine analysis results using
   "`scalasca -examine scorep_<title>`"

FLAT MPI PROFILE: EXAMPLE (CONT.)



WHERE IS THE KEY BOTTLENECK?

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

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

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

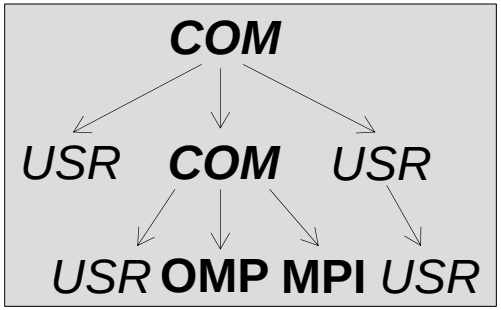
CALL-PATH PROFILE: RECIPE

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

CALL-PATH PROFILE: EXAMPLE (CONT.)

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

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



CALL-PATH PROFILE: EXAMPLE (CONT.)

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

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

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

CALL-PATH PROFILE: FILTERING

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

☞Avoid measurements to reduce the overhead

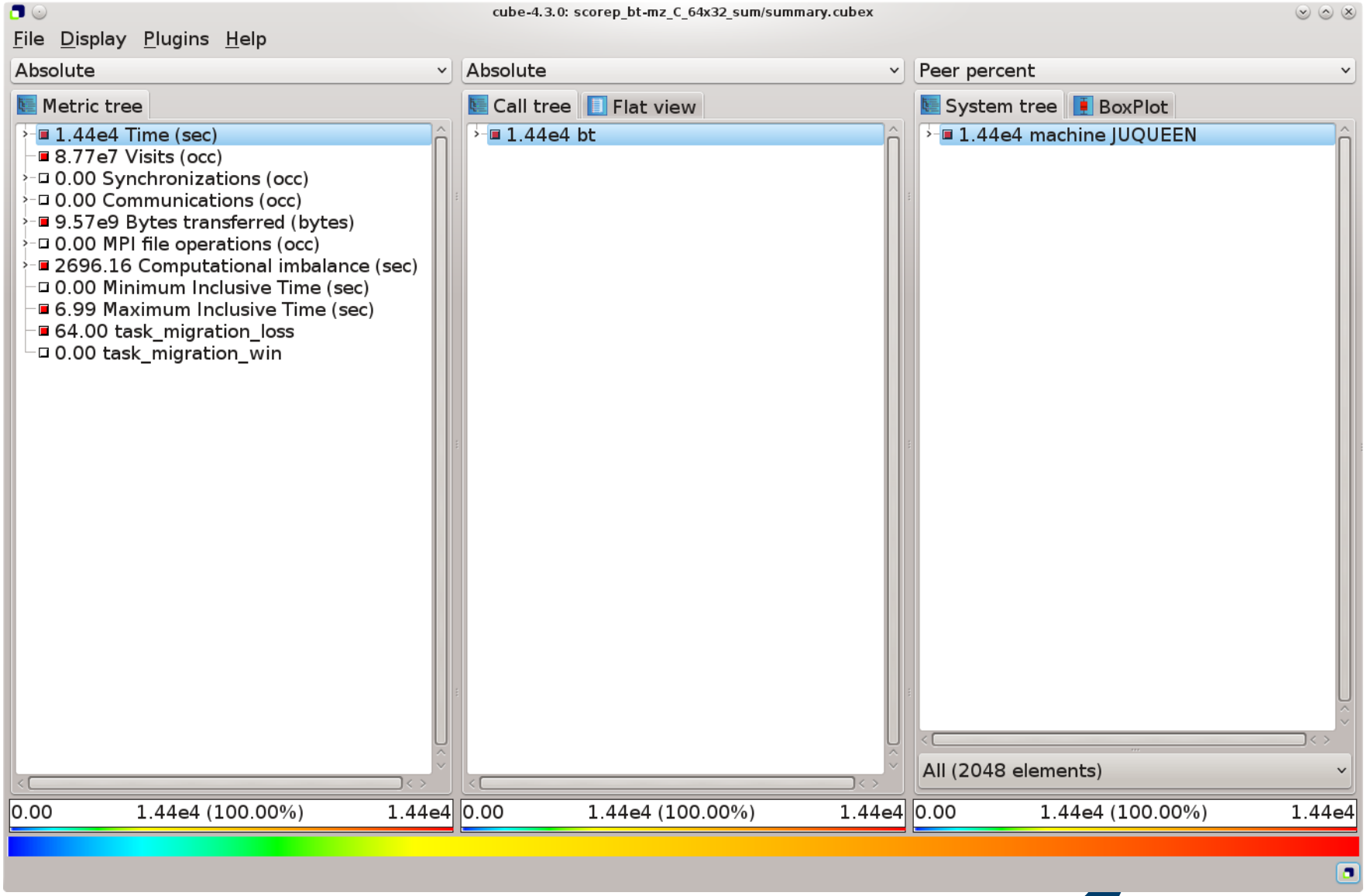
☞List routines to be filtered in simple text file

FILTERING: EXAMPLE

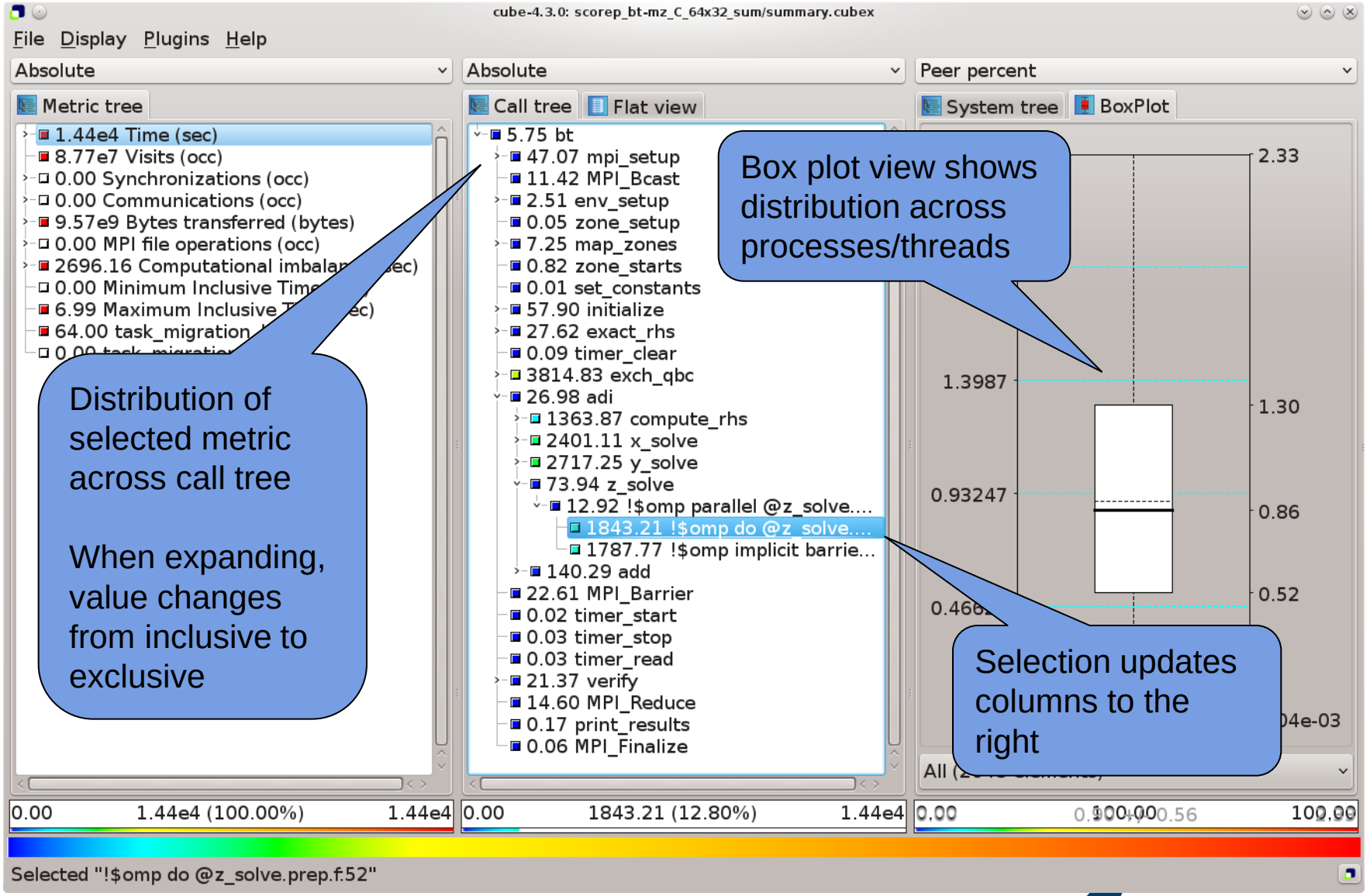
```
% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE
        binvcrhs
        matmul_sub
        matvec_sub
        binvrhs
        lhsinit
        exact_solution
SCOREP_REGION_NAMES_END
```

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

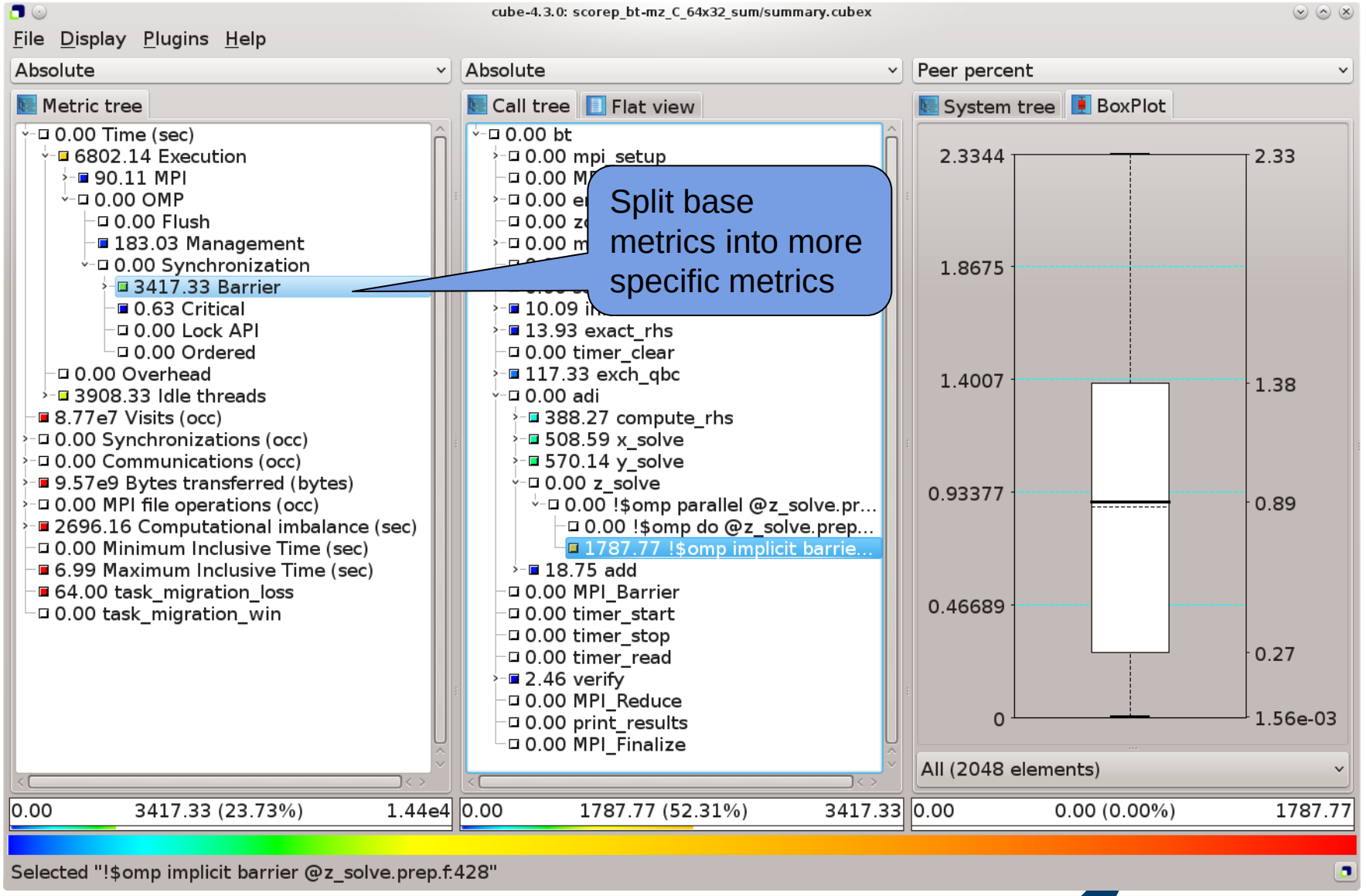
CALL-PATH PROFILE: EXAMPLE (CONT.)



CALL-PATH PROFILE: EXAMPLE (CONT.)



CALL-PATH PROFILE: EXAMPLE (CONT.)



SCORE-P: ADVANCED FEATURES

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

WHY IS THE BOTTLENECK THERE?

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

HARDWARE COUNTERS

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

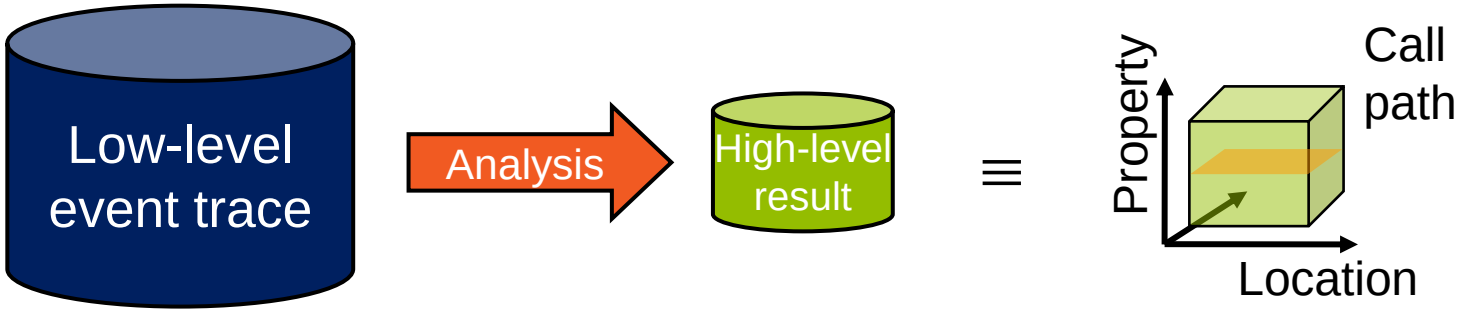
PAPI

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

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

Automatic Trace Analysis w/ Scalasca

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



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

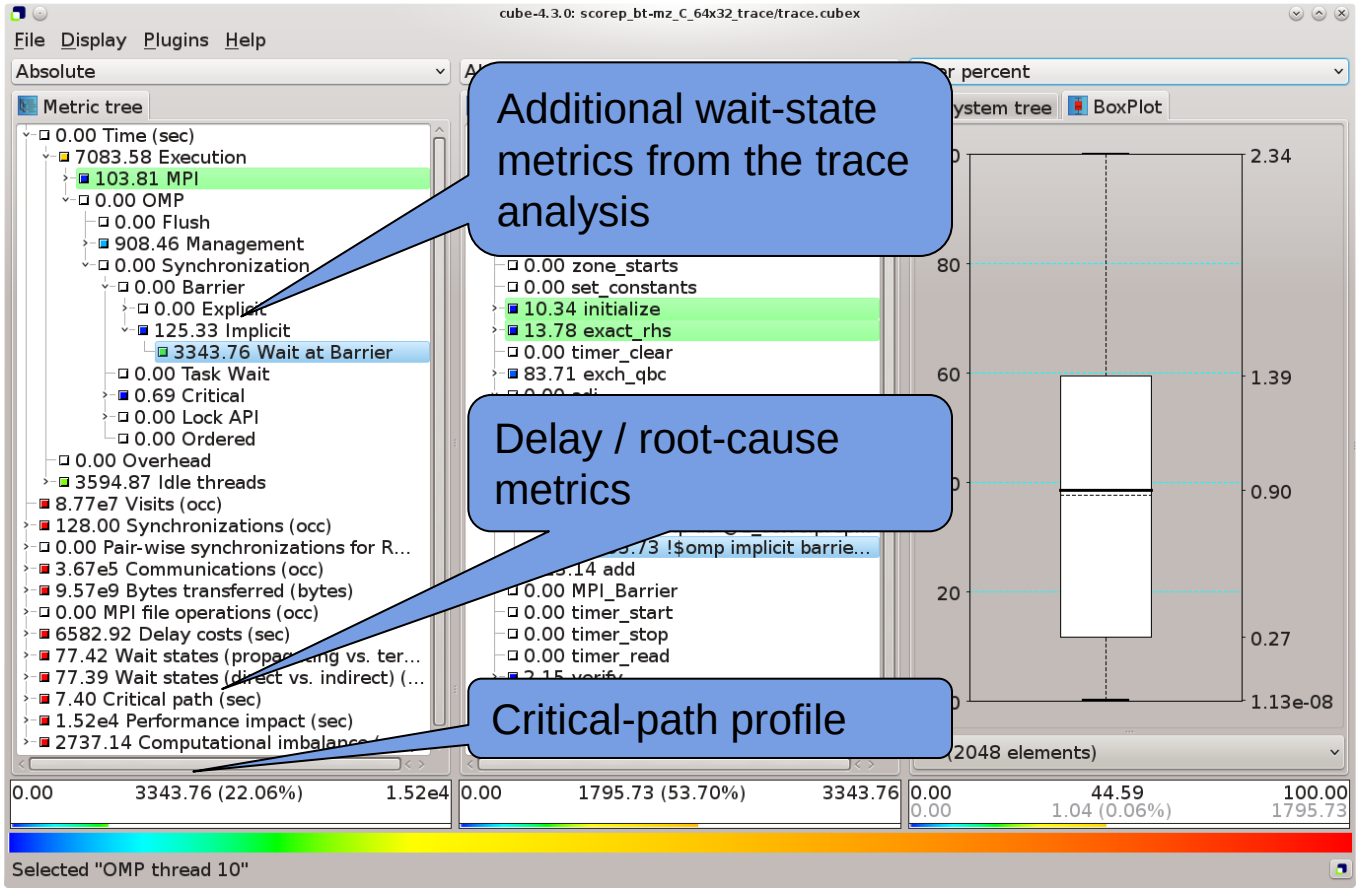
TRACE GENERATION & ANALYSIS W/ SCALASCA

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

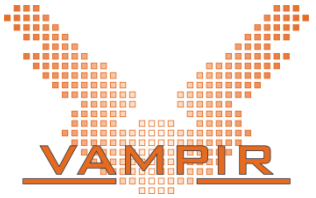
```
############################
##  In the job script:  ##
############################

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

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

SCALASCA TRACE ANALYSIS EXAMPLE



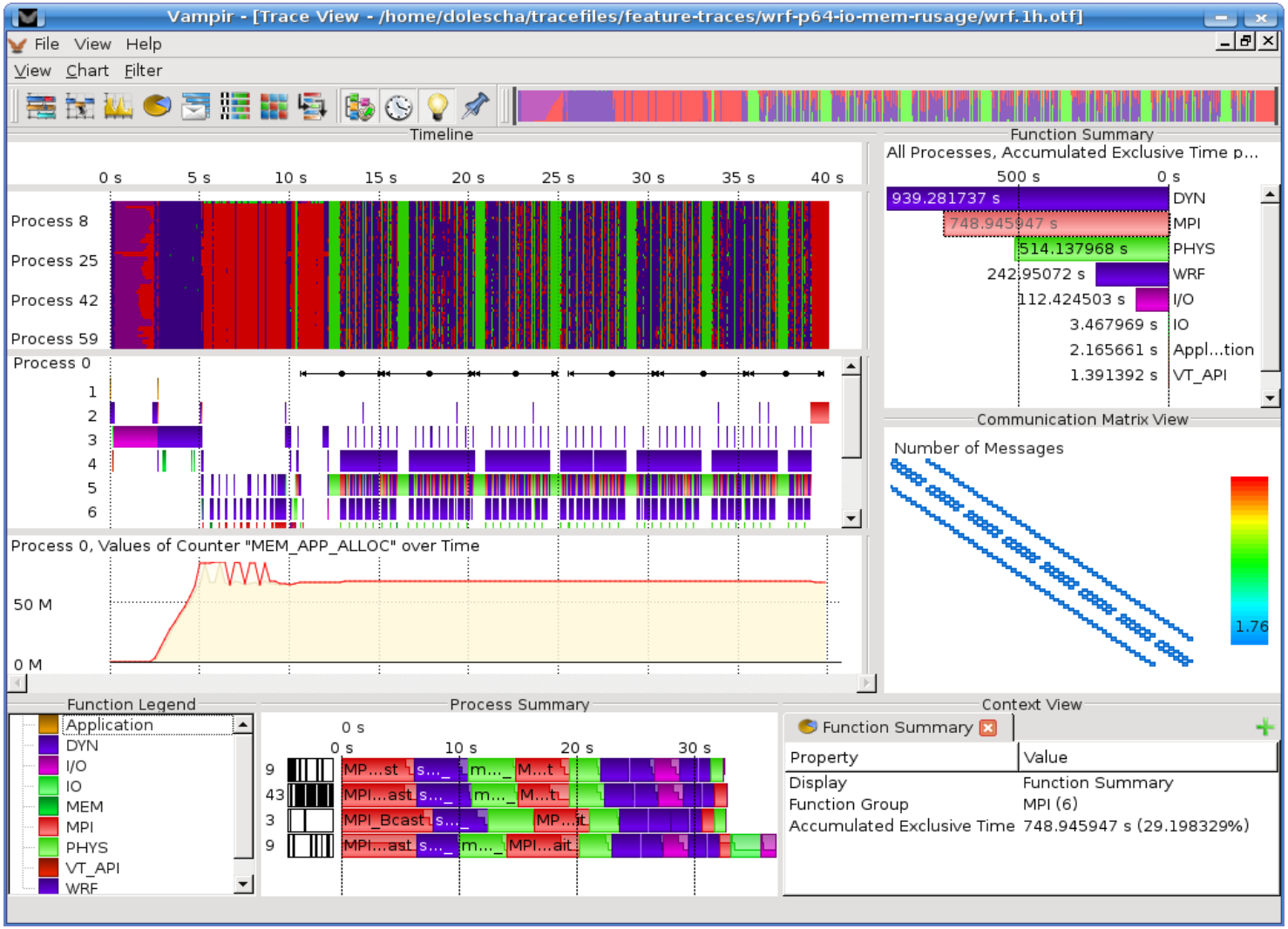
VAMPIR EVENT TRACE VISUALIZER



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

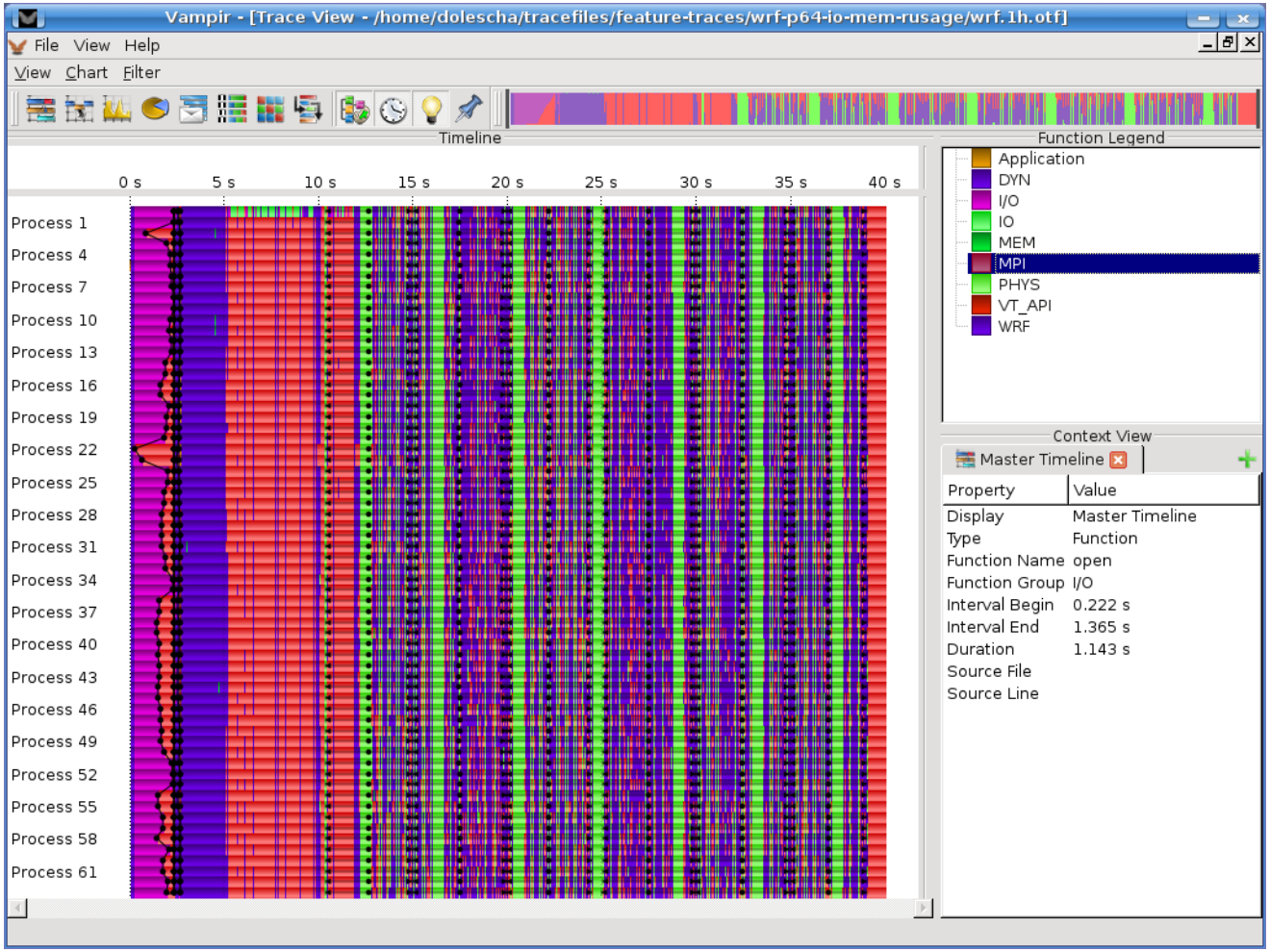
- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis

VAMPIR DISPLAYS



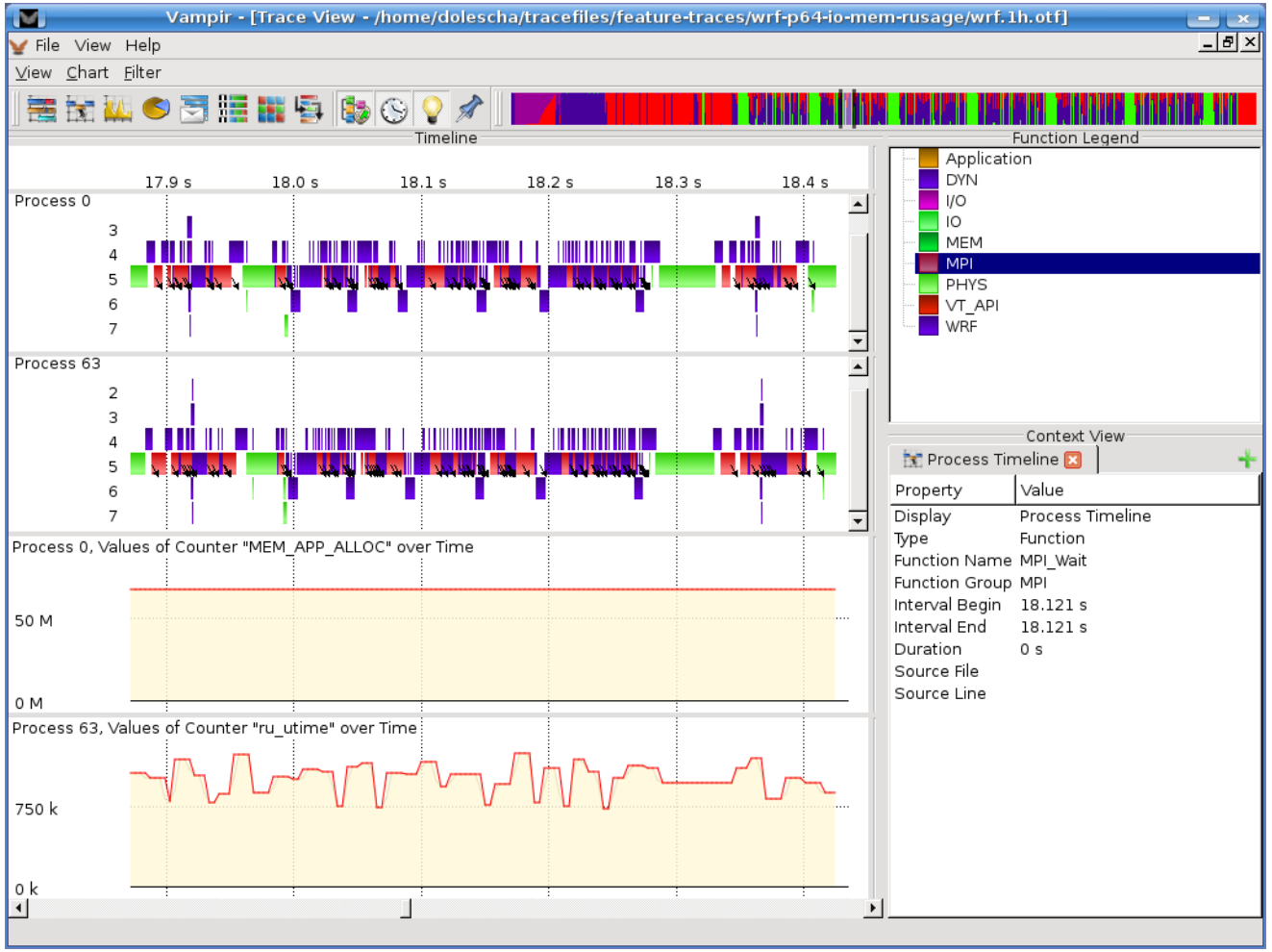
VAMPIR: TIMELINE DIAGRAM

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size
- Information about states, messages, collective and I/O operations available through clicking on the representation



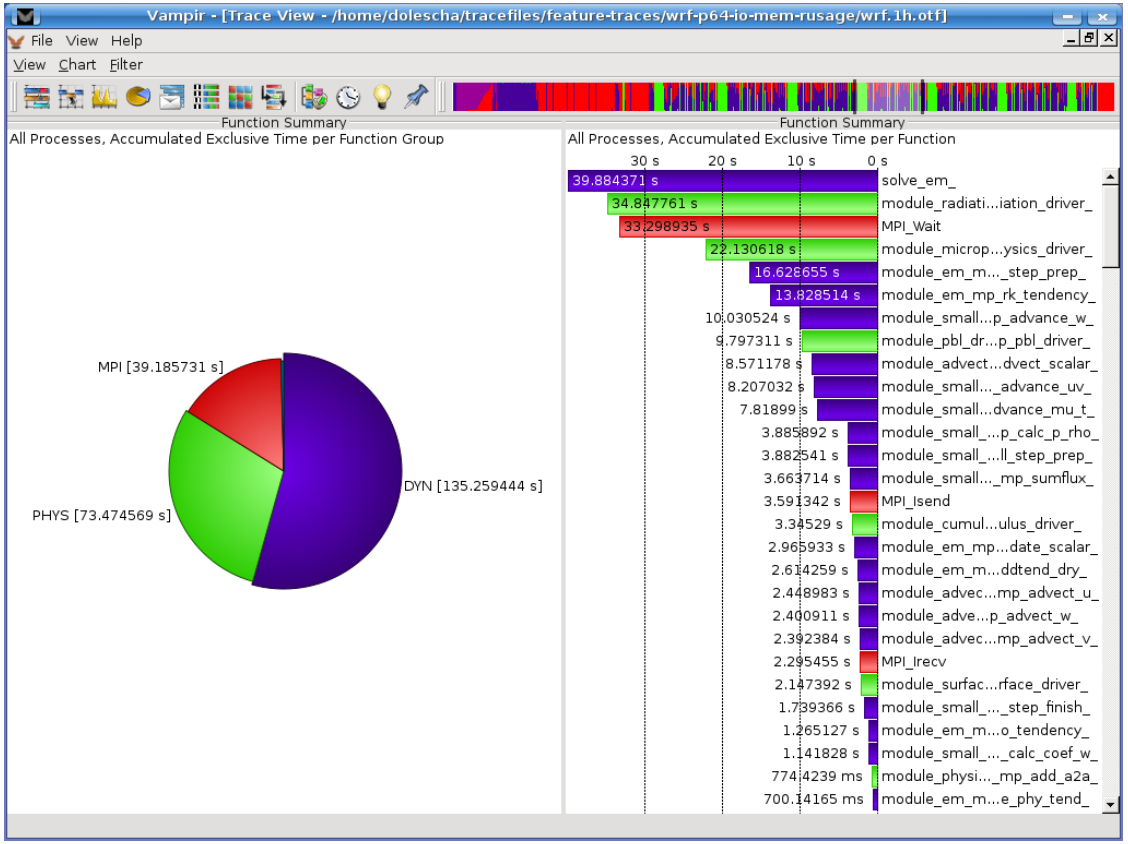
VAMPIR: PROCESS AND COUNTER TIMELINES

- Process timeline show call stack nesting
- Counter timelines for hardware and software counters



VAMPIR: EXECUTION STATISTICS

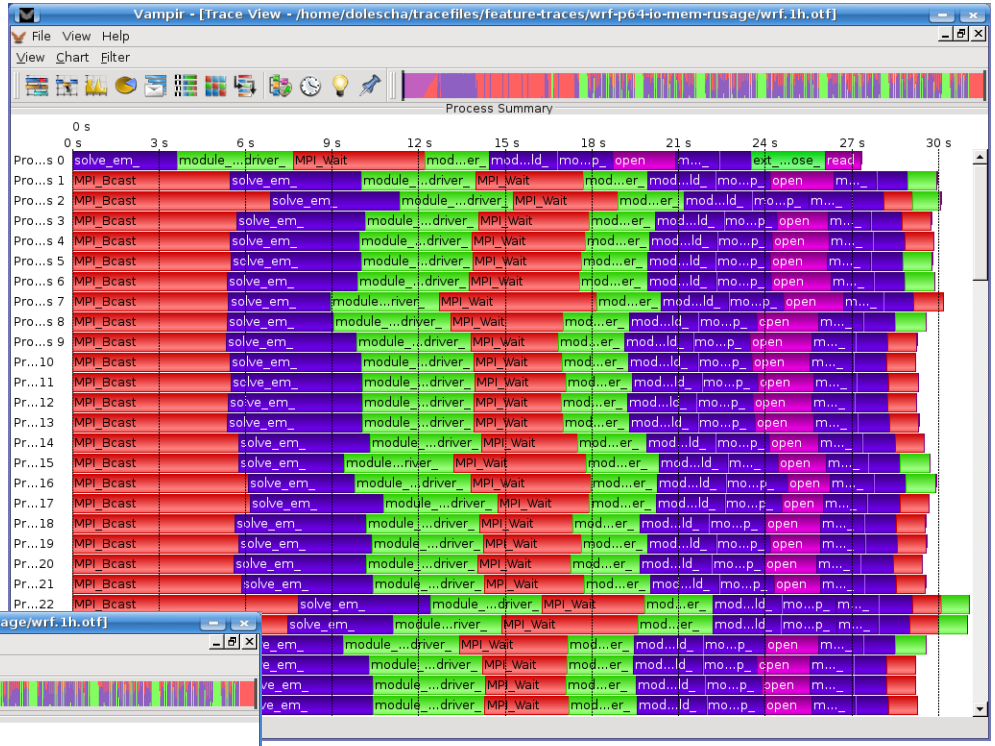
- Aggregated profiling information: execution time, number of calls, inclusive/exclusive



- Aggregated profiling information: execution time, number of calls, inclusive/exclusive
- Available for all / any group (activity) or all routines (symbols)
- Available for any part of the trace
  ⇨ selectable through time line diagram

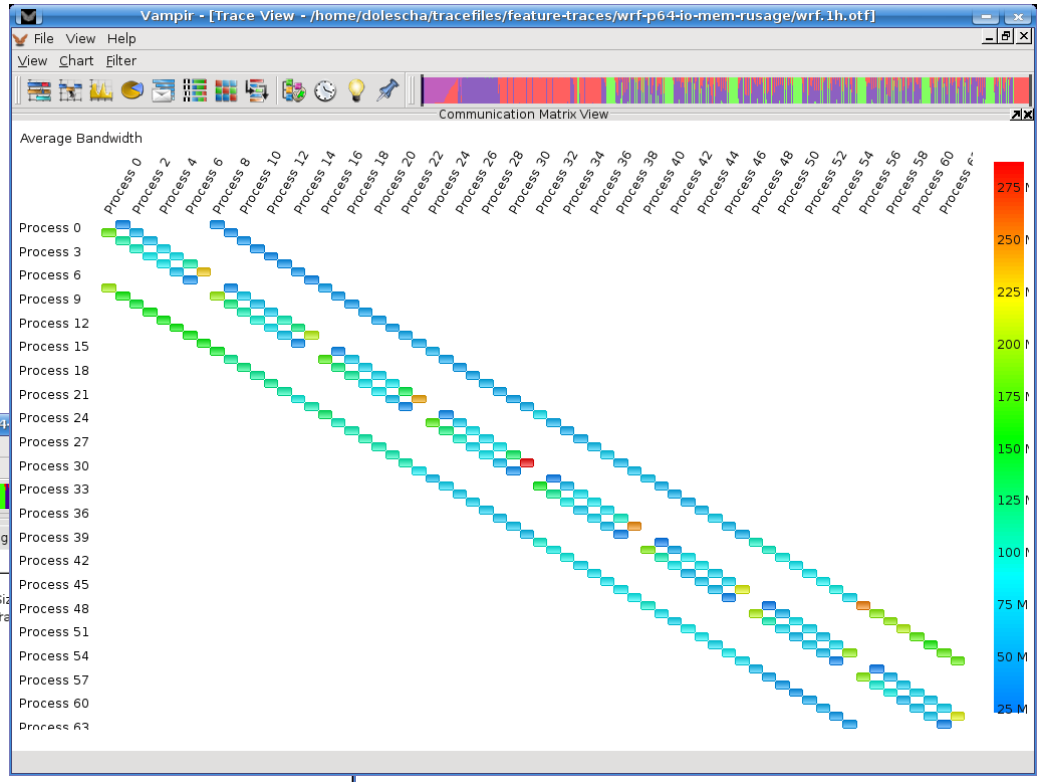
VAMPIR: PROCESS SUMMARY

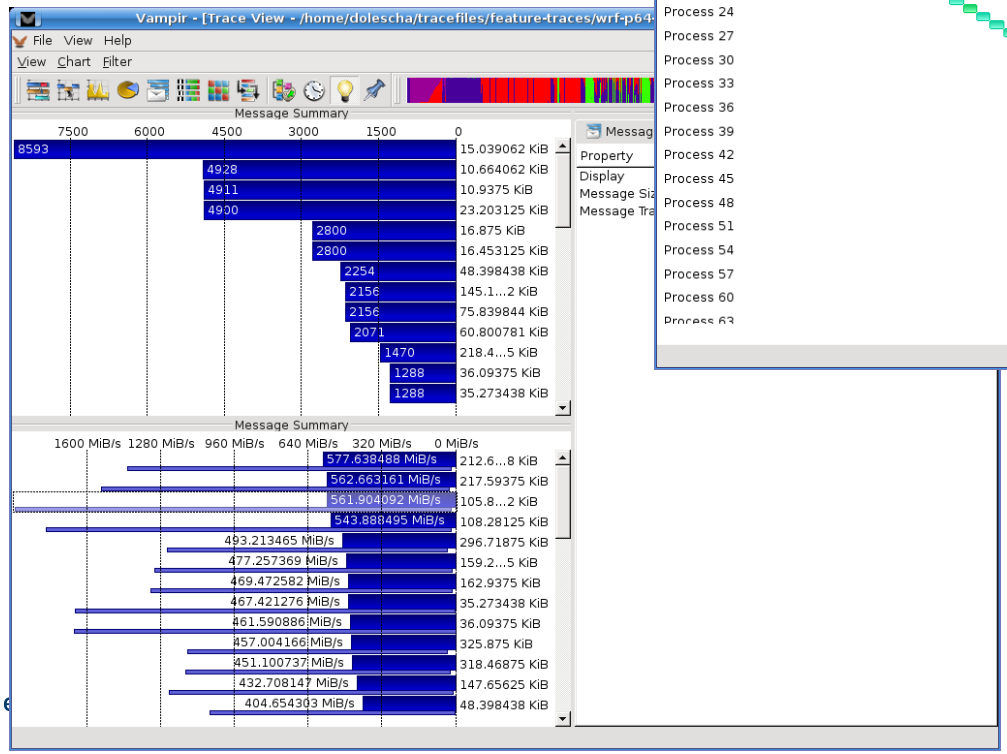
- Execution statistics over all processes for comparison
- Clustering mode available for large process counts



VAMPIR: COMMUNICATION STATISTICS

- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics





- Available for any part of the trace

JÜLICH Forschungszentrum

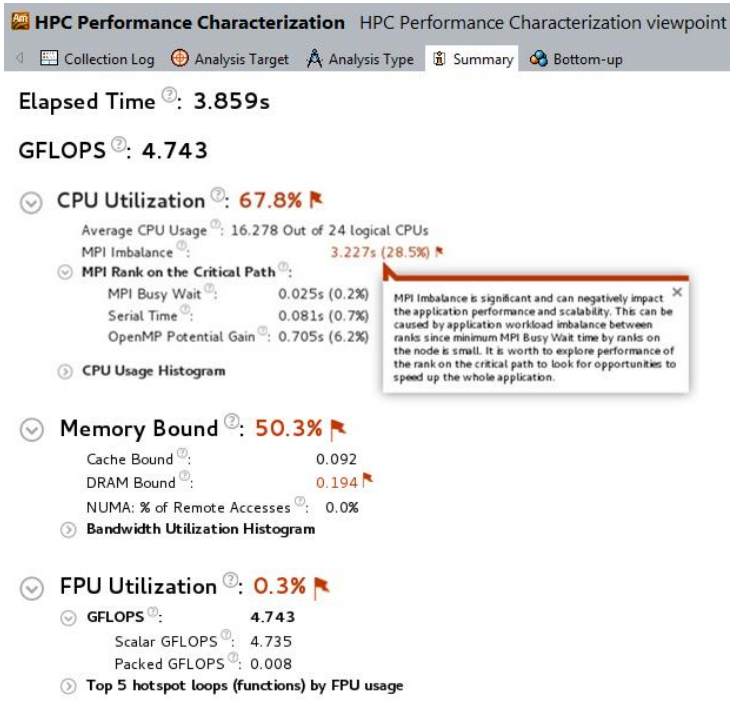
VTUNE AMPLIFIER XE

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

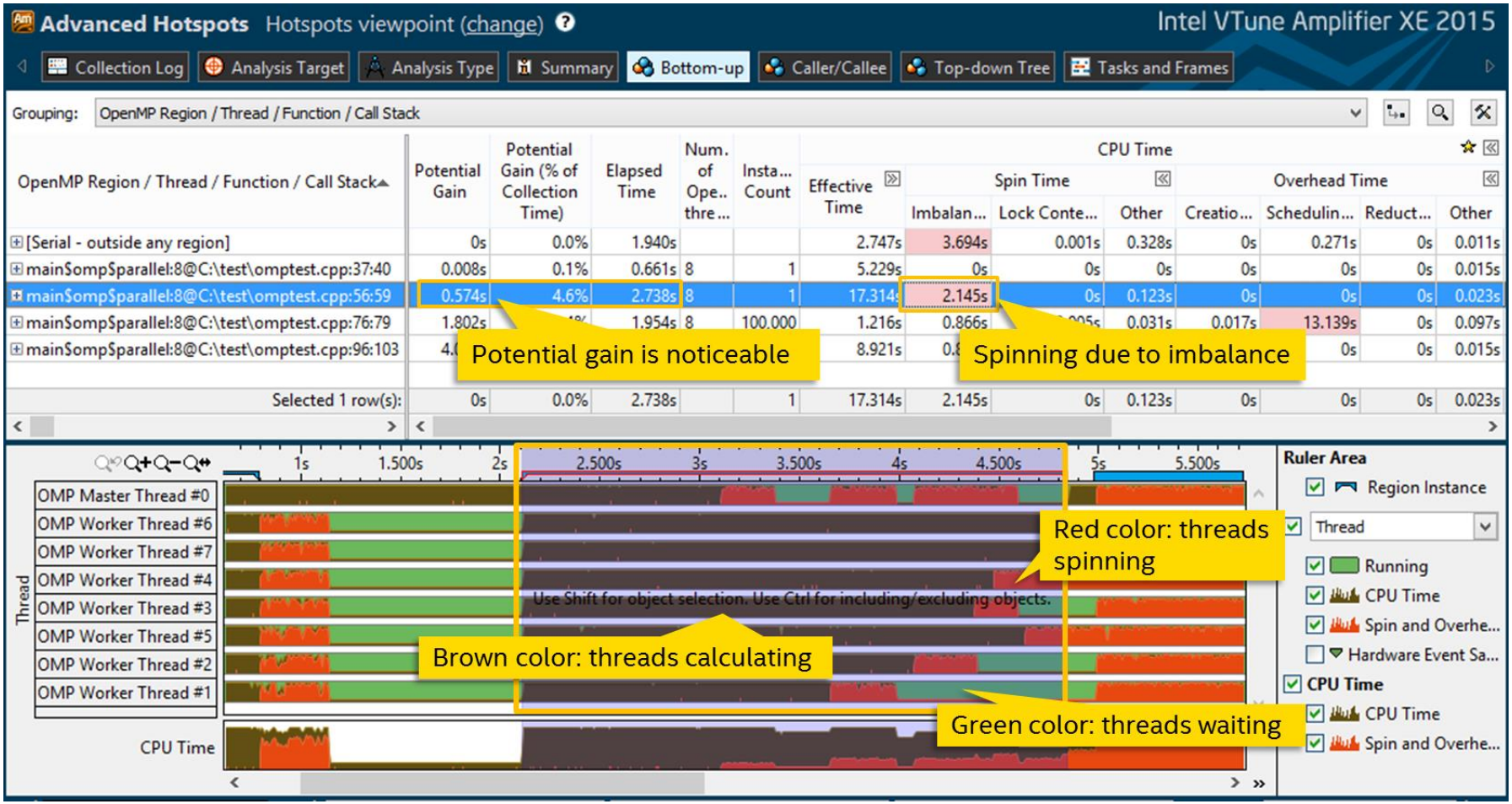
- Lock and Wait analysis for OpenMP and TBB
- HPC analysis for quick overview

- Bandwidth and memory analysis
- I/O analysis

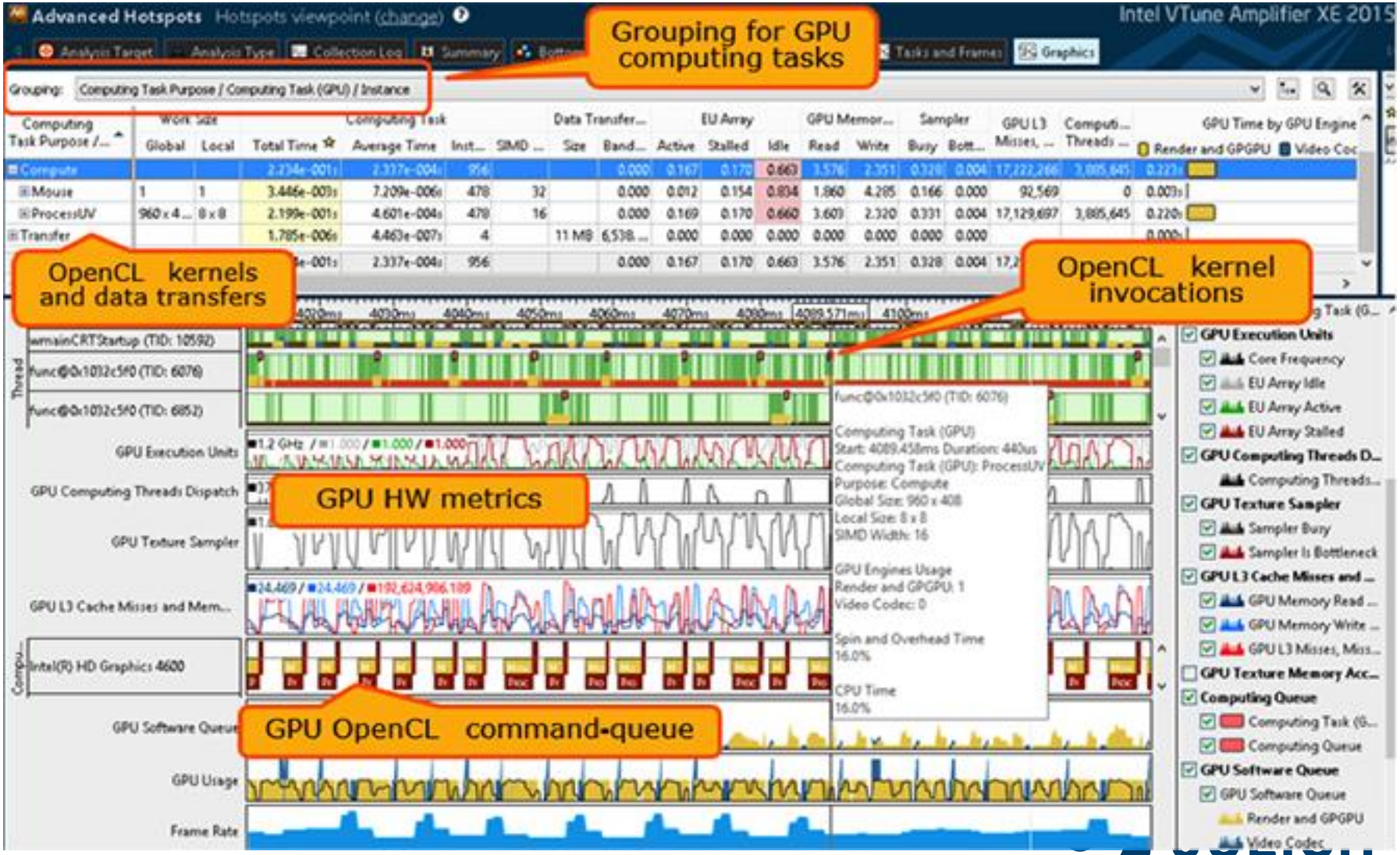
- OpenCL and GPU profiling
  (no CUDA, Intel iGPU only)



INTEL VTUNE AMPLIFIER GUI



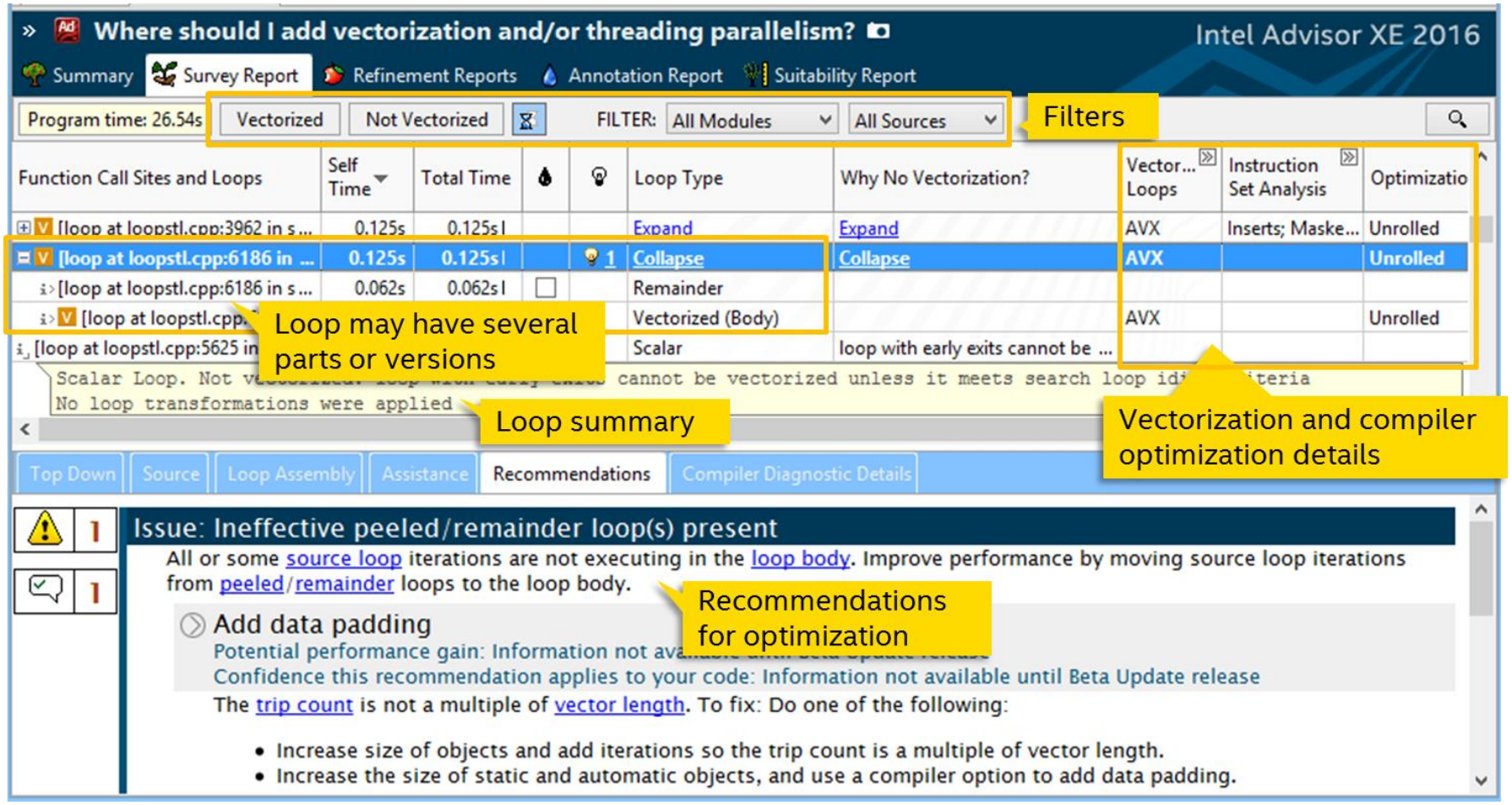
INTEL VTUNE – GPU ANALYSIS



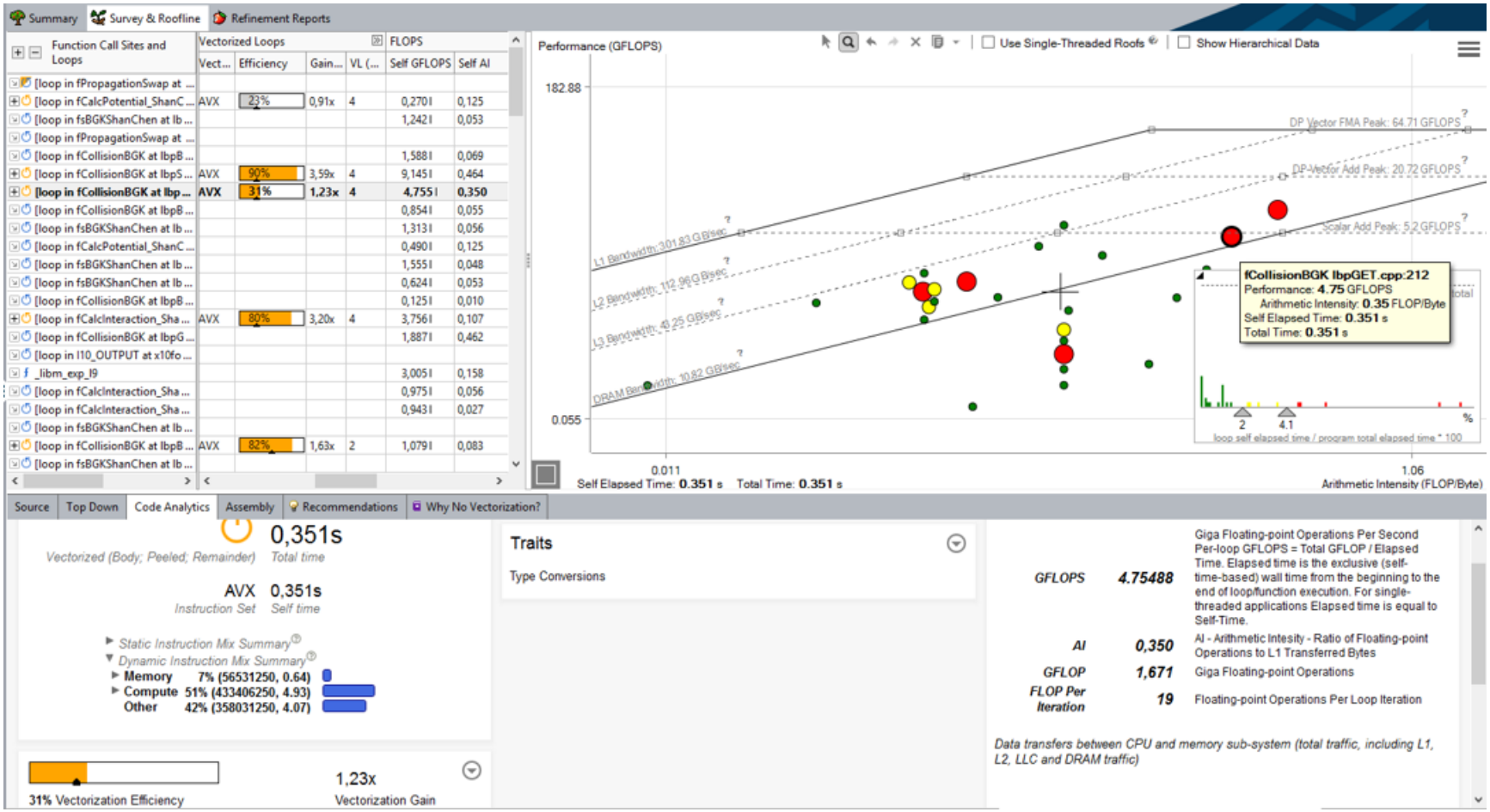
INTEL ADVISOR

- Vectorization Advisor
  - Loops-based analysis to identify vectorization candidates
  - Finds save spots to enforce compiler vectorization
  - Roofline analysis to explore performance headroom and co-optimize memory and computation
- Threading Advisor
  - Identify issues before parallelization
  - Prototype performance impact of different threading designs
  - Find and eliminate data-sharing issues
- Flow-Graph Analysis
  - Speed up algorithm design and express parallelism efficiently
  - Plan, validate, and model application design
- C/C++ and Fortran with OpenMP and Intel TBB

INTEL ADVISOR GUI



INTEL ADVISOR – ROOFLINE



ALLINEA PERFORMANCE REPORTS

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

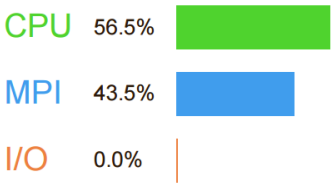
- Supports MPI, multi-threading and accelerators
- Saves data in HTML, CVS or text form

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

EXAMPLE PERFORMANCE REPORTS

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

The total wallclock time was spent as follows:



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

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

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

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

CPU

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

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

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

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

MPI

Of the 43.5% total time spent in MPI calls:

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

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

I/O

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

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

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

Memory

Per-process memory usage may also affect scaling:

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

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

DARSHAN

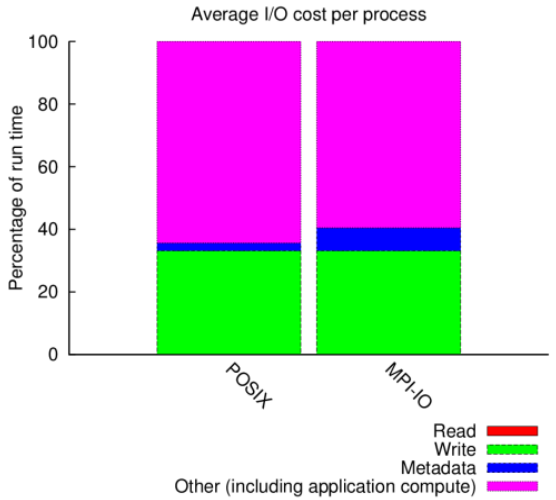
- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

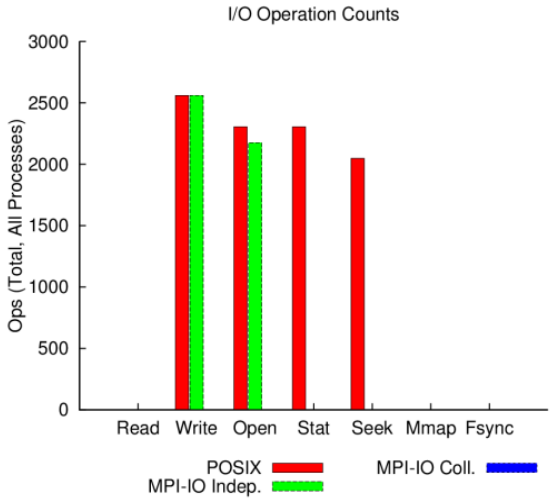
- Supports POSIX, MPI-IO, HDF5, PnetCDF, …
- Binary log file written at exit post-processed into PDF report

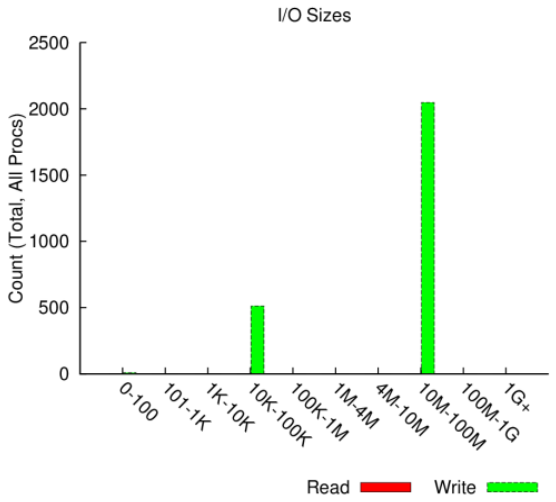
- http://www.mcs.anl.gov/research/projects/darshan/
- Open Source: installed on many HPC systems

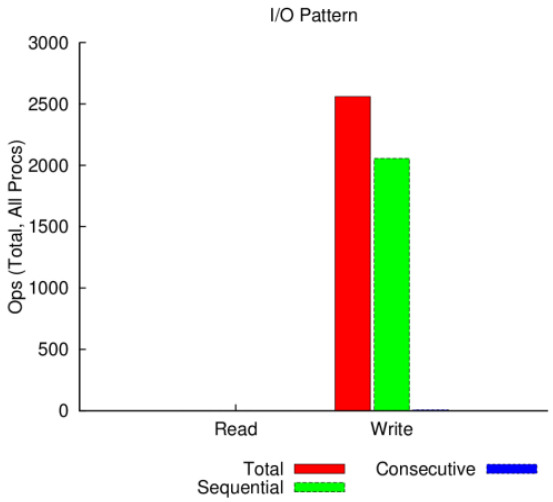
EXAMPLE DARSHAN REPORT EXTRACT

jobid:	uid:	nprocs: 4096	runtime: 175 seconds

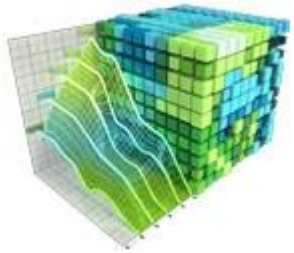








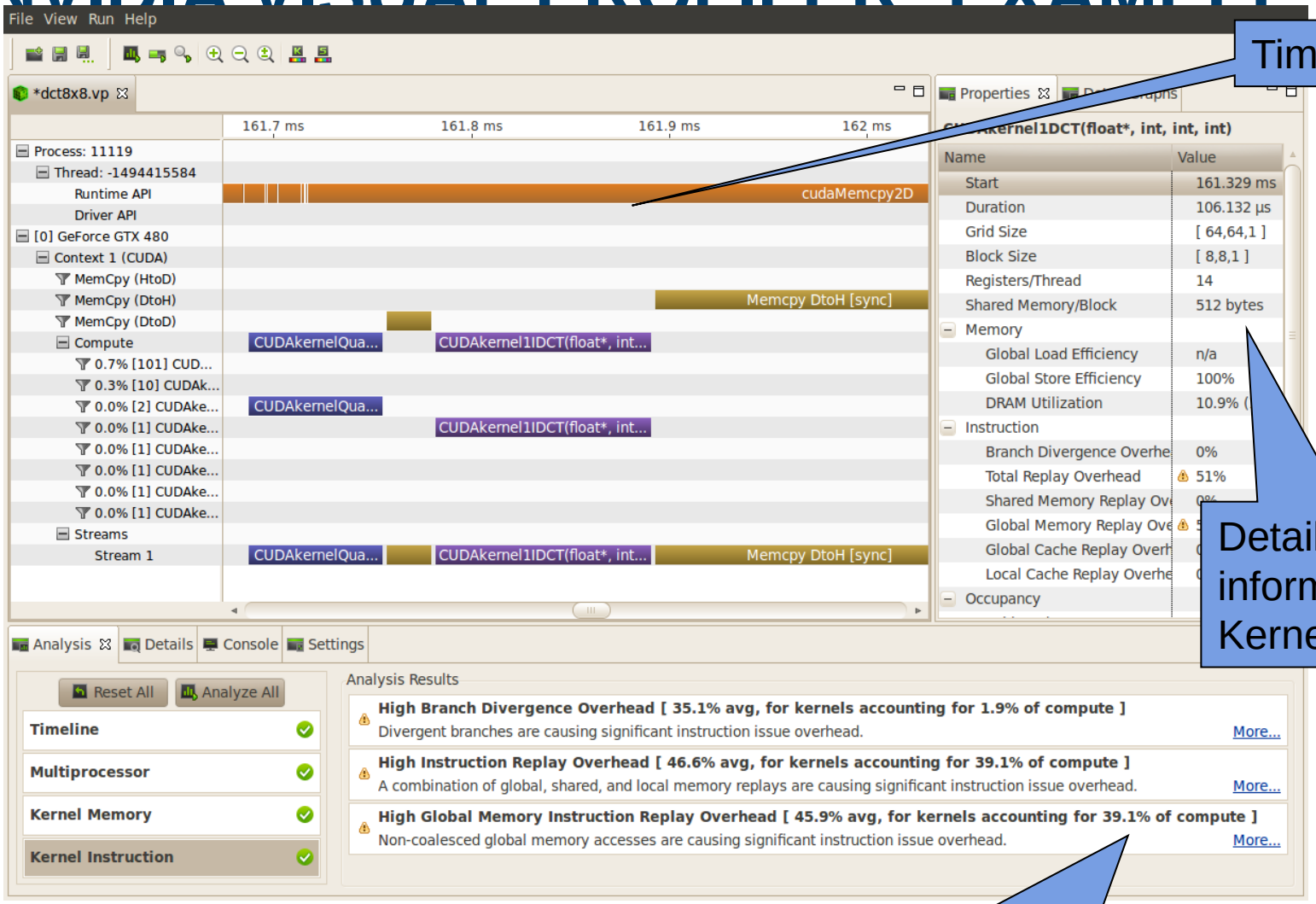
NVIDIA VISUAL PROFILER



- Part of the CUDA Toolkit
- Supports all CUDA enabled GPUs
- Supports CUDA and OpenACC on Windows, OS X and Linux

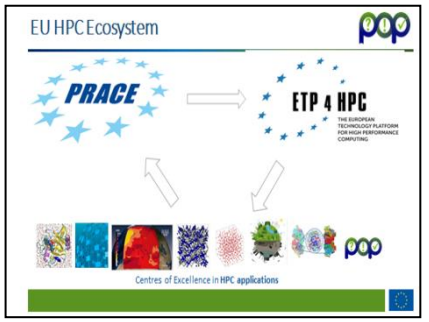
- Unified CPU and GPU Timeline
- CUDA API trace
  - Memory transfers, kernel launches, and other API functions
- Automated performance analysis
  - Identify performance bottlenecks and get optimization suggestions
- Guided Application Analysis
- Power, thermal, and clock profiling

NVIDIA VISUAL PROFILER: EXAMPLE



POP SERVICES

POP COE



- A **Center of Excellence**
  - On Performance Optimization and Productivity
  - Promoting best practices in parallel programming
- Providing Services
  - Precise understanding of application and system behaviour
  - Suggestions/support on how to refactor code in the most productive way
- Horizontal
  - Transversal across application areas, platforms and scales
- For (your?) academic AND industrial codes and users

The best: Currently free of charge

MOTIVATION

- **Why?**
  - Complexity of machines and codes

    → Frequent lack of quantified understanding of actual behaviour

    → Not clear most productive direction of code refactoring
  - Important to maximize efficiency (performance, power) of compute intensive applications and productivity of the development efforts

- **What?**
  - Parallel programs, mainly MPI/OpenMP
    - Also CUDA, OpenCL, OpenACC, Python, …

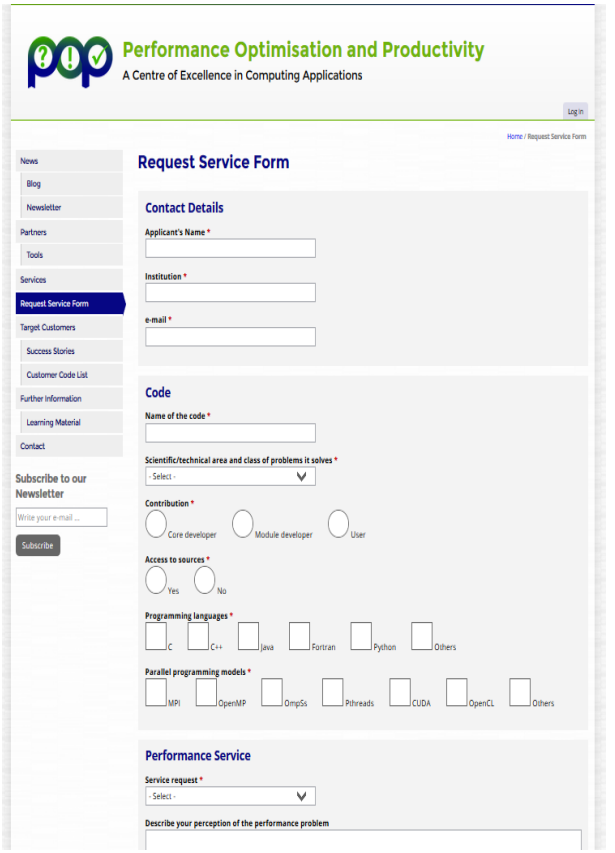
THE PROCESS …

When?

- POP I: October 2015 – March 2018
- POP II: December 2018 – November 2021

How?

- Apply
  - Fill in small questionnaire describing application and needs https://pop-coe.eu/request-service-form
    - Questions? Ask pop@bsc.es
- Selection/assignment process
- Install tools @ your production machine (local, PRACE, …)
- Interactively: Gather data → Analysis → Report



SERVICES PROVIDED BY THE COE

? Parallel Application Performance Audit ⇨ Report

- Primary service
- Identify performance issues of customer code (at customer site)
- Small effort (< 1 month)

! Parallel Application Performance Plan ⇨ Report

- Follow-up on the audit service
- Identifies the root causes of the issues found and qualifies and quantifies approaches to address them
- Longer effort (1-3 months)

✓ Proof-of-Concept ⇨ Software Demonstrator

- Experiments and mock-up tests for customer codes
- Kernel extraction, parallelisation, mini-apps experiments to show effect of proposed optimisations
- 6 months effort