

HPC Software – Debugger and Performance Tools

May 2018 | [Michael Knobloch](#)

m.knobloch@fz-juelich.de

- 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

- 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
 - 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)

Debuggers

Debugging Tools (**status: May 2018**)

- TotalView
- DDT
- MUST
- Intel Inspector

- 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

The screenshot shows the TotalView debugger interface. At the top is a menu bar (File, Edit, View, Group, Process, Thread, Action Point, Debug, Tools, Window, Help) and a toolbar with icons for Go, Halt, Kill, Restart, Next Step, Out, Run To, Prev, UnStep, Caller, BackTo, and Live. Below the toolbar is a status bar showing 'Rank 0: hm.0 (At Breakpoint 1)' and 'Thread 1 (46997343479520): hm (At Breakpoint 1)'. The main window is divided into several panes. The 'Stack Trace' pane on the left shows a list of frames: 'C main, FP=7fff40759f90', '_libc_start_main, FP=7fff4075a040', and '_start, FP=7fff4075a050'. The 'Stack Frame' pane on the right shows details for the 'main' function, including arguments 'argc: 0x00000001 (1)' and 'argv: 0x7fff4075a068', and local variables 'myrank: 0x00000000 (0)' and 'numprocs: 0x00000004 (4)'. The 'Source code' pane at the bottom shows the code for 'Function main in hello-mpi.c', with line 10 highlighted: 'printf("hello from %d of %d\n", myrank, numprocs);'. A red arrow points to this line, indicating a breakpoint. The 'Action Points' pane at the bottom shows a list of breakpoints, with the first one highlighted: '1 hello-mpi.c#11 main+0x5f'. Callouts with blue boxes and white text point to these features: 'Stack trace' points to the Stack Trace pane; 'Break points' points to the red arrow on line 10; 'Toolbar for common options' points to the toolbar; 'Local variables for selected stack frame' points to the local variables section in the Stack Frame pane; and 'Source code window' points to the source code pane.

Stack trace

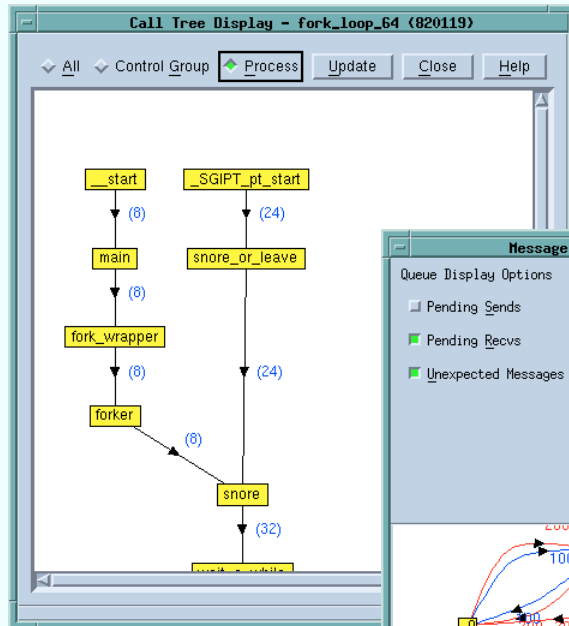
Break points

Toolbar for common options

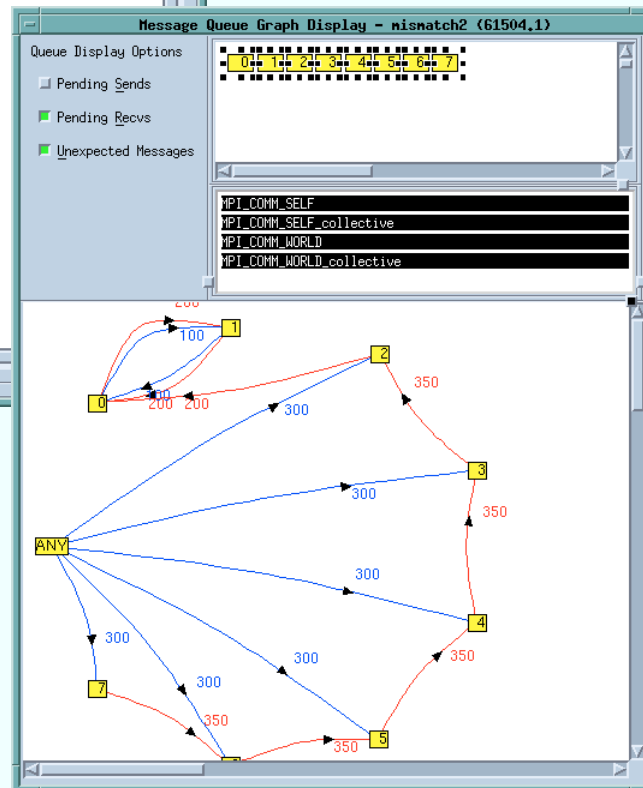
Local variables for selected stack frame

Source code window

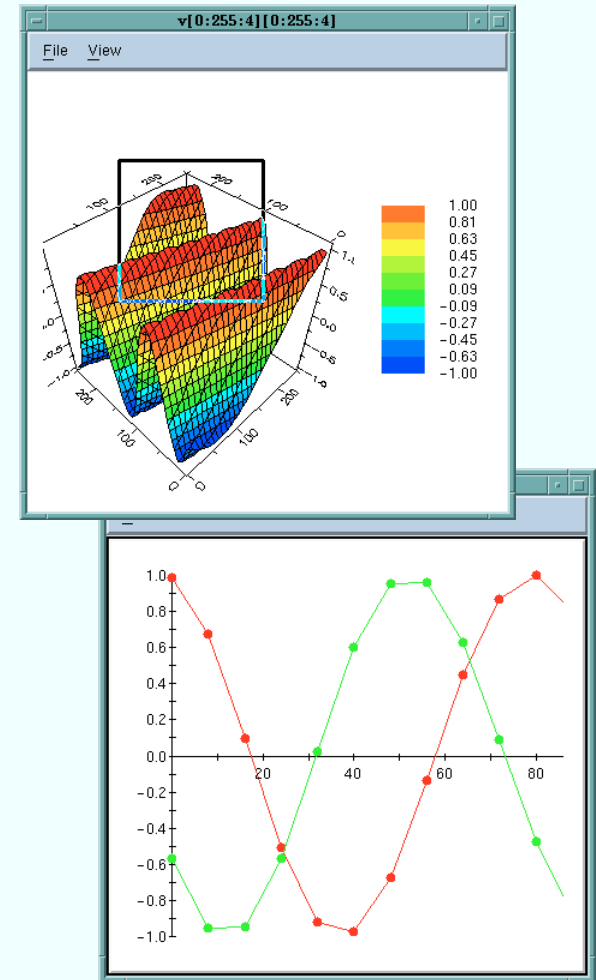
- Call Graph



- Message queue graph

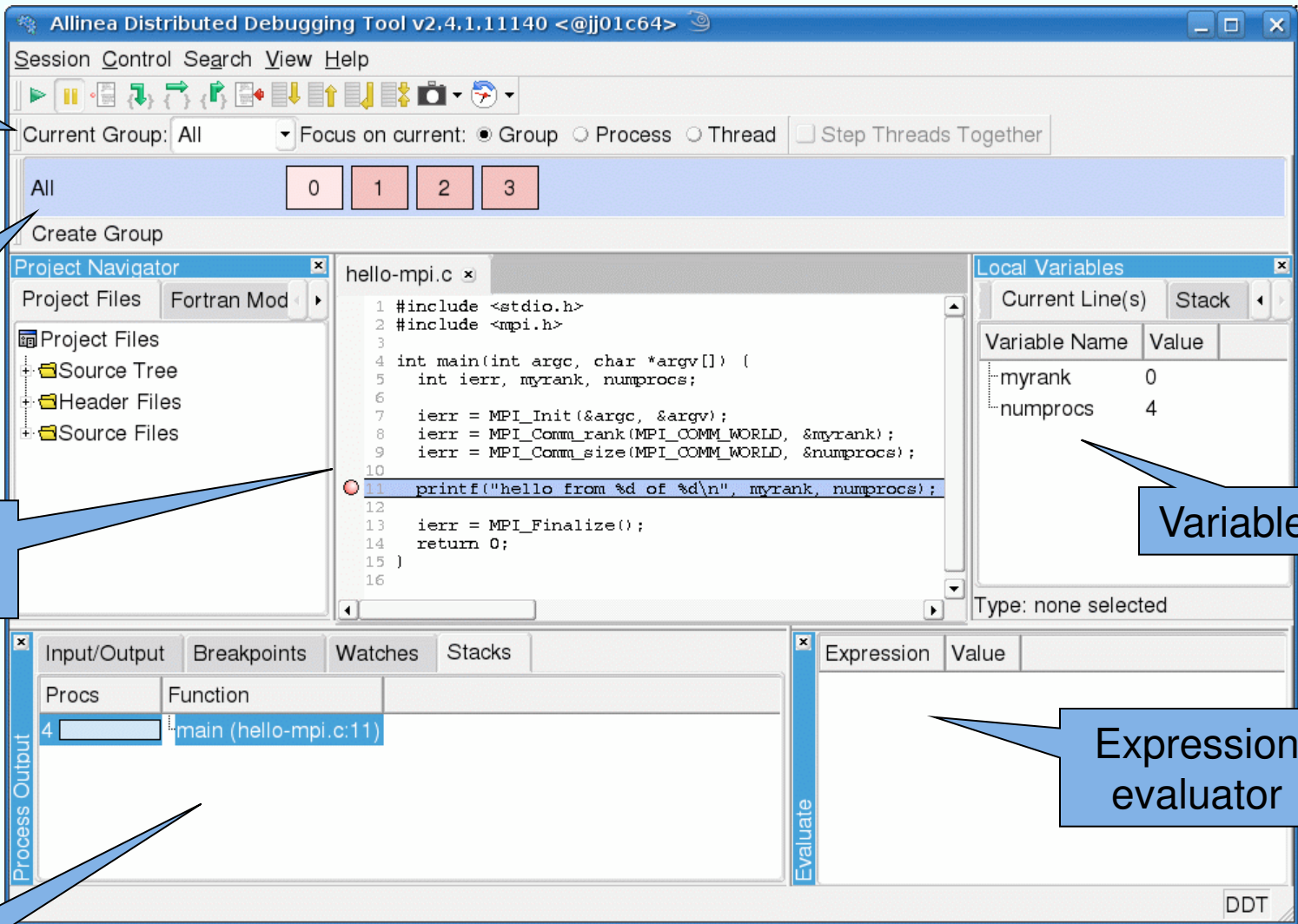


- Data visualization



- 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



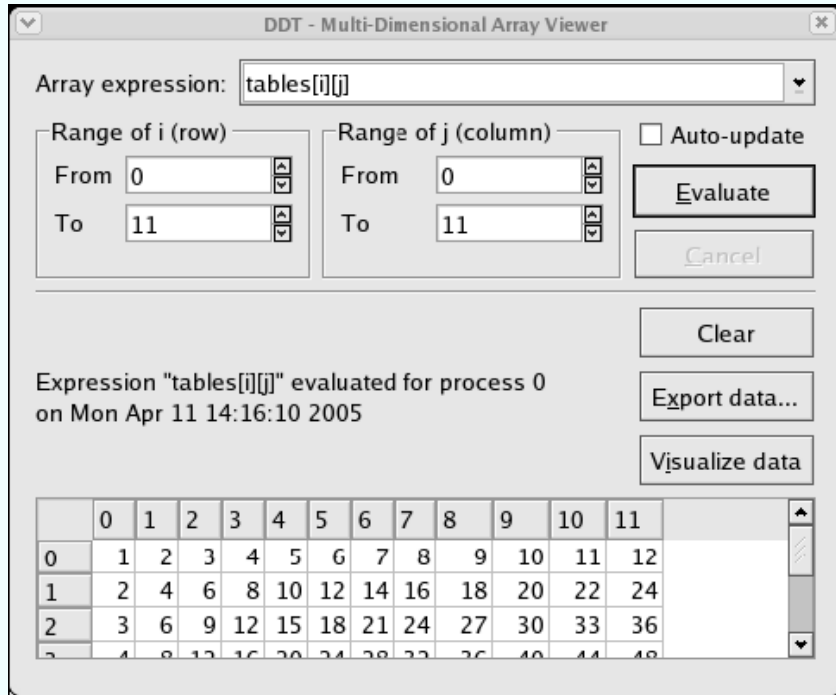
The screenshot shows the Allinea Distributed Debugging Tool (DDT) v2.4.1.11140 interface. The window is divided into several panes:

- Process controls:** Located at the top, it includes a menu bar (Session, Control, Search, View, Help), a toolbar with various debugging icons, and a section for "Current Group: All" with buttons for "0", "1", "2", and "3".
- Process groups:** A "Create Group" button is located below the process controls.
- Source code:** The "Project Navigator" pane on the left shows a tree structure with "Project Files", "Fortran Mod", "Source Tree", "Header Files", and "Source Files". The main pane displays the source code for "hello-mpi.c", with line 11 highlighted:

```
11 printf("hello from %d of %d\n", myrank, numprocs);
```
- Variables:** The "Local Variables" pane on the right shows the current line(s) and a table of variables:

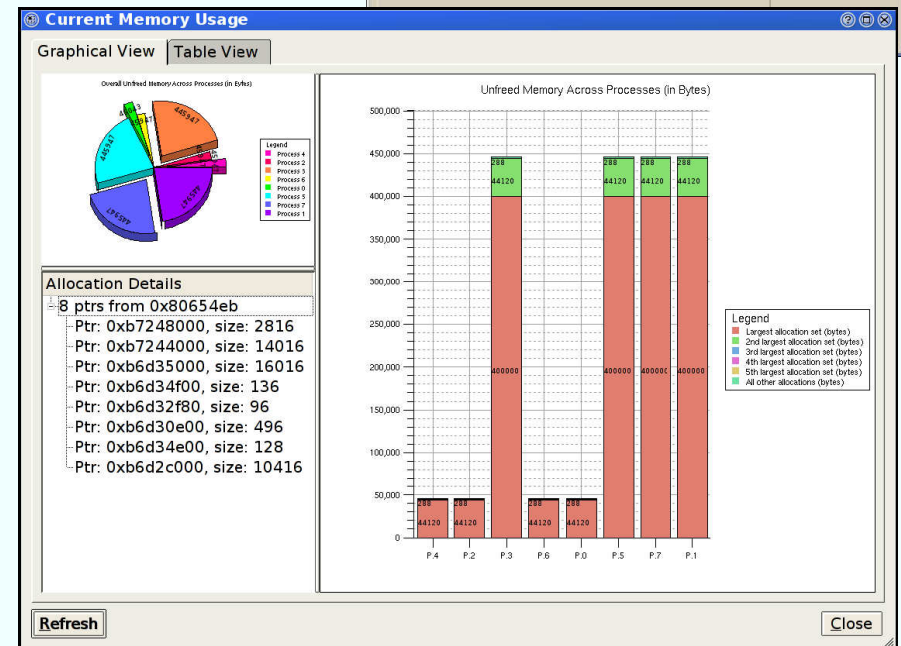
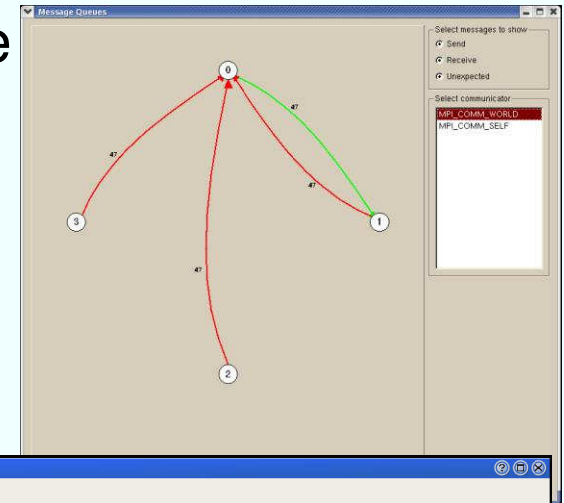
Variable Name	Value
myrank	0
numprocs	4
- Stack trace:** The "Input/Output" pane at the bottom left shows a table of processes and functions:

Procs	Function
4	main (hello-mpi.c:11)
- Expression evaluator:** The "Expression" pane at the bottom right allows for evaluating expressions, with a table showing "Expression" and "Value".



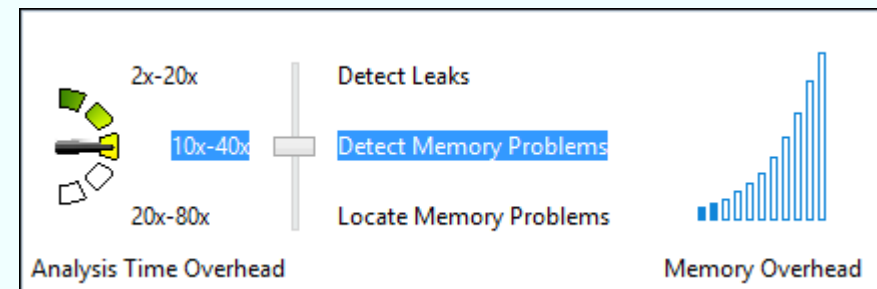
- Multi-Dimensional Array Viewer

- Message queue graph



- Memory Usage

- 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



The screenshot shows the Intel Inspector GUI with the 'Detect Deadlocks and Data Races' analysis. The 'Problems' table lists three data race issues (P1, P2, P3) involving 'find_and_fix_threading_errors.exe'. The 'Filters' panel on the right shows the severity as 'Error' and the type as 'Data race'. The 'Code Locations' panel at the bottom shows the source code for 'render_one_pixel' in 'find_and_fix_threading_errors.cpp'.

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

The screenshot shows the Intel Inspector GUI with the 'Detect Memory Problems' analysis. The 'Problems' table lists four memory issues (P1, P2, P3, P4) involving 'find_and_fix_memory_errors.exe'. The 'Code Locations' panel at the bottom shows the source code for 'operator()' in 'find_and_fix_memory_error...'.

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

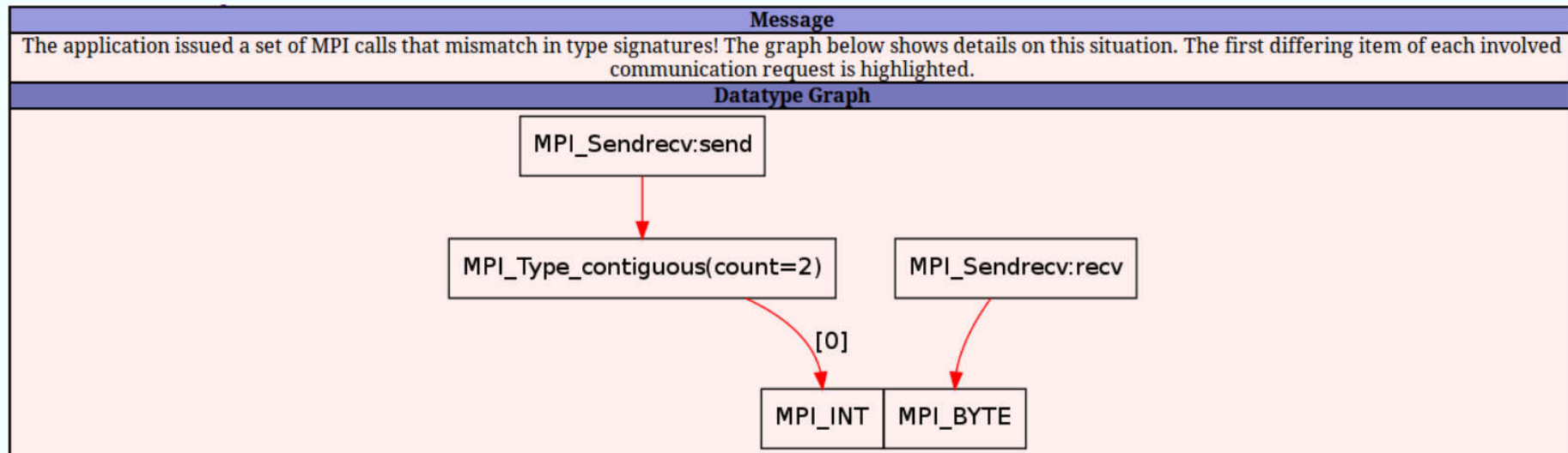
- Next generation MPI correctness and portability checker
- <http://doc.itc.rwth-aachen.de/display/CCP/Project+MUST>



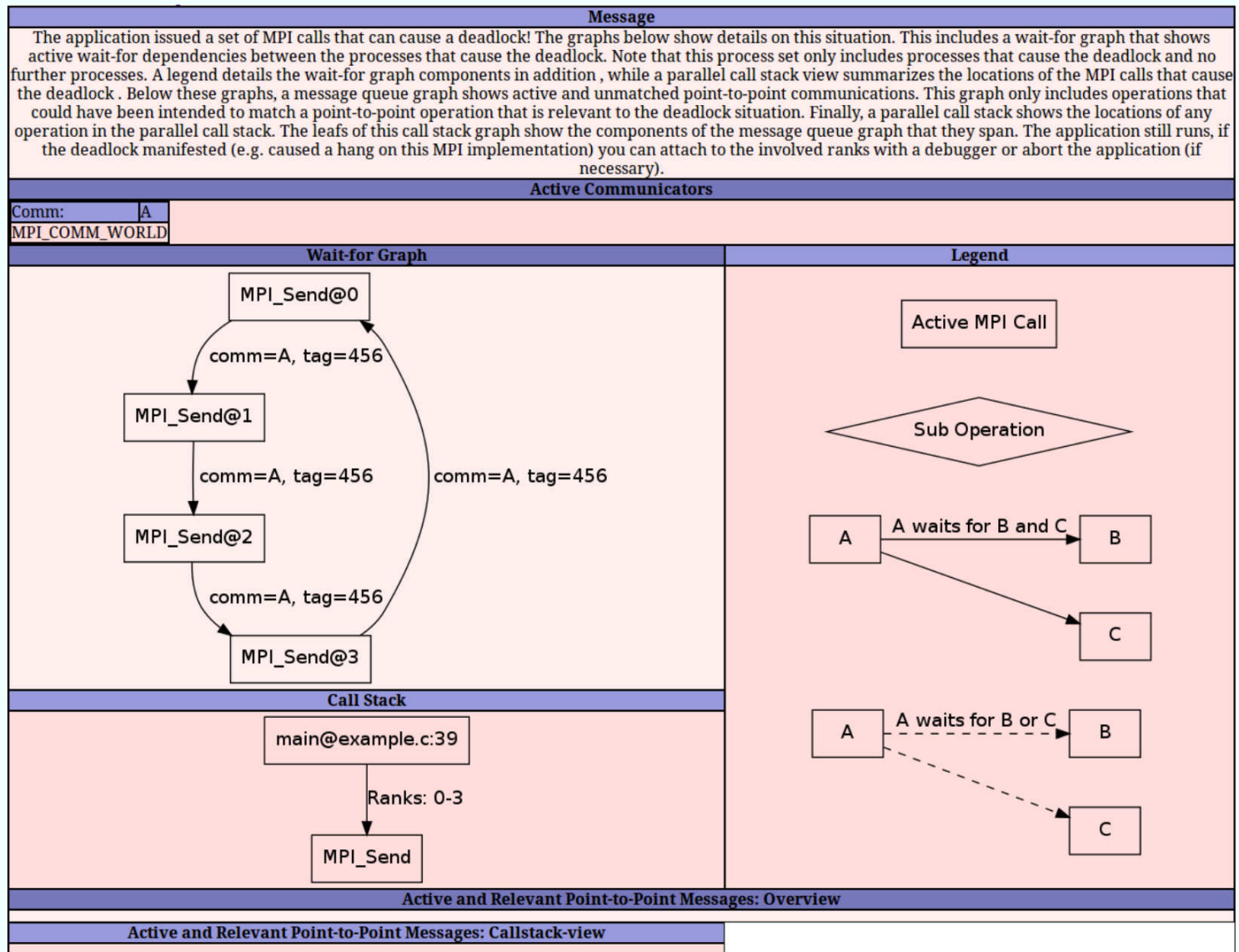
- MUST reports
 - Errors: violations of the MPI-standard
 - Warnings: unusual behavior or possible problems
 - Notes: harmless but remarkable behavior
 - Further: potential deadlock detection
- Usage
 - Relink application with `mustc`, `mustcxx`, `mustf90`, ...
 - Run application under the control of `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, committed at reference 4, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}) (Information on receive of count 8 with type:MPI_BYTE)	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



MUST Deadlock Detection



Performance Analysis Tools

- 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, ...

- Accuracy
 - Intrusion overhead
 - Measurement itself needs time and thus lowers performance
 - Perturbation
 - Measurement alters program behavior, e.g., memory access pattern
 - Might prevent compiler optimization, e.g. function inlining
 - Accuracy of timers & counters
- Granularity
 - How many measurements?
 - How much information / processing during each measurement?

☞ *Tradeoff: Accuracy vs. Expressiveness of data*

- 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*



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



UNIVERSITY OF OREGON

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

- 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
 - 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

```
% module load Toolchain Score-P Scalasca
% mpif90 -O3 -c foo.f90
% mpif90 -O3 -c bar.f90
% scorep --nocompiler \
    mpif90 -O3 -o myprog foo.o bar.o
```

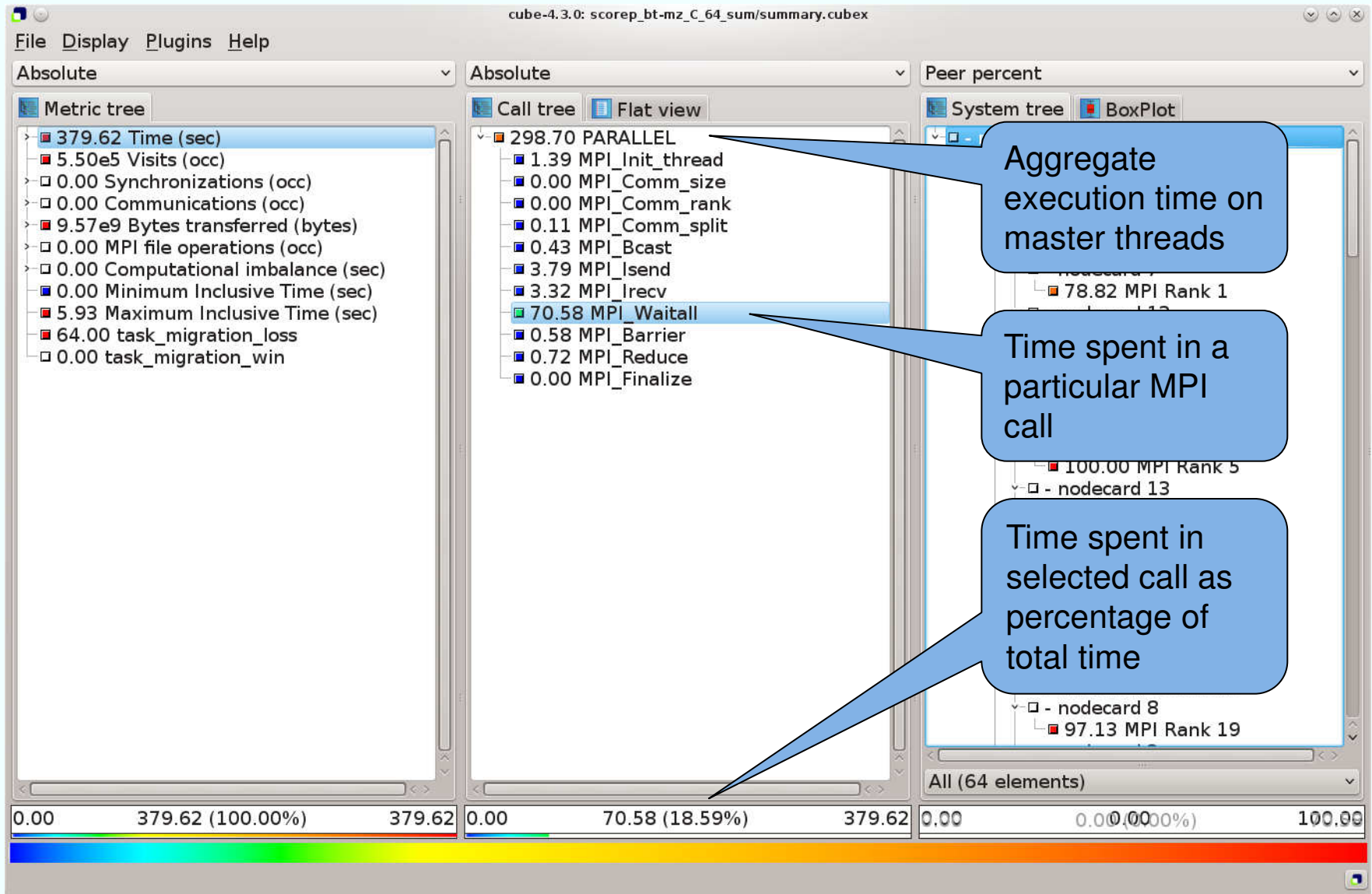
```
#####
##  In the job script:  ##
#####
```

```
module load Toolchain Scalasca
scalasca -analyze \
    srun --tasks-per-node  $P$  --ntasks  $n$  [...] --exe ./myprog
```

```
#####
## After job finished: ##
#####
```

```
% scalasca -examine scorep_myprog_ $P$  $n$  $x$  $t$ _sum
```

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):
Automatic compiler-based instrumentation
- Option 2:
Manual instrumentation of interesting phases, routines, loops

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

```
% module load Toolchain Score-P Scalasca
% scorep mpif90 -O3 -c foo.f90
% scorep mpif90 -O3 -c bar.f90
% scorep \
    mpif90 -O3 -o myprog foo.o bar.o
```

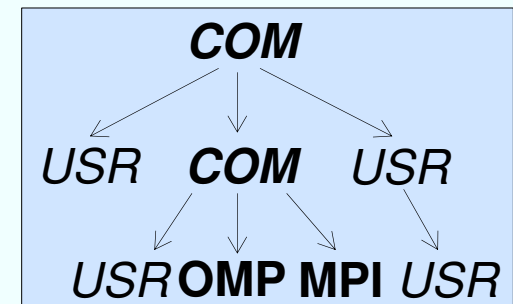
```
#####
##  In the job script:  ##
#####
```

```
module load Toolchain Scalasca
scalasca -analyze \
    srun --tasks-per-node  $P$  --ntasks  $n$  [...] --exe ./myprog
```

Call-path Profile: Example (cont.)

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

- Estimates trace buffer requirements
- Allows to identify candidate 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_Ppnext_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/ visit[us]	region
	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
	[...]						

- In this example, the 6 most frequently 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

```
% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvrhs
    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.)

```
## To verify effect of filter:

% scalasca -examine -s -f filter.txt \
  scorep_myprog_Pp n x t_sum

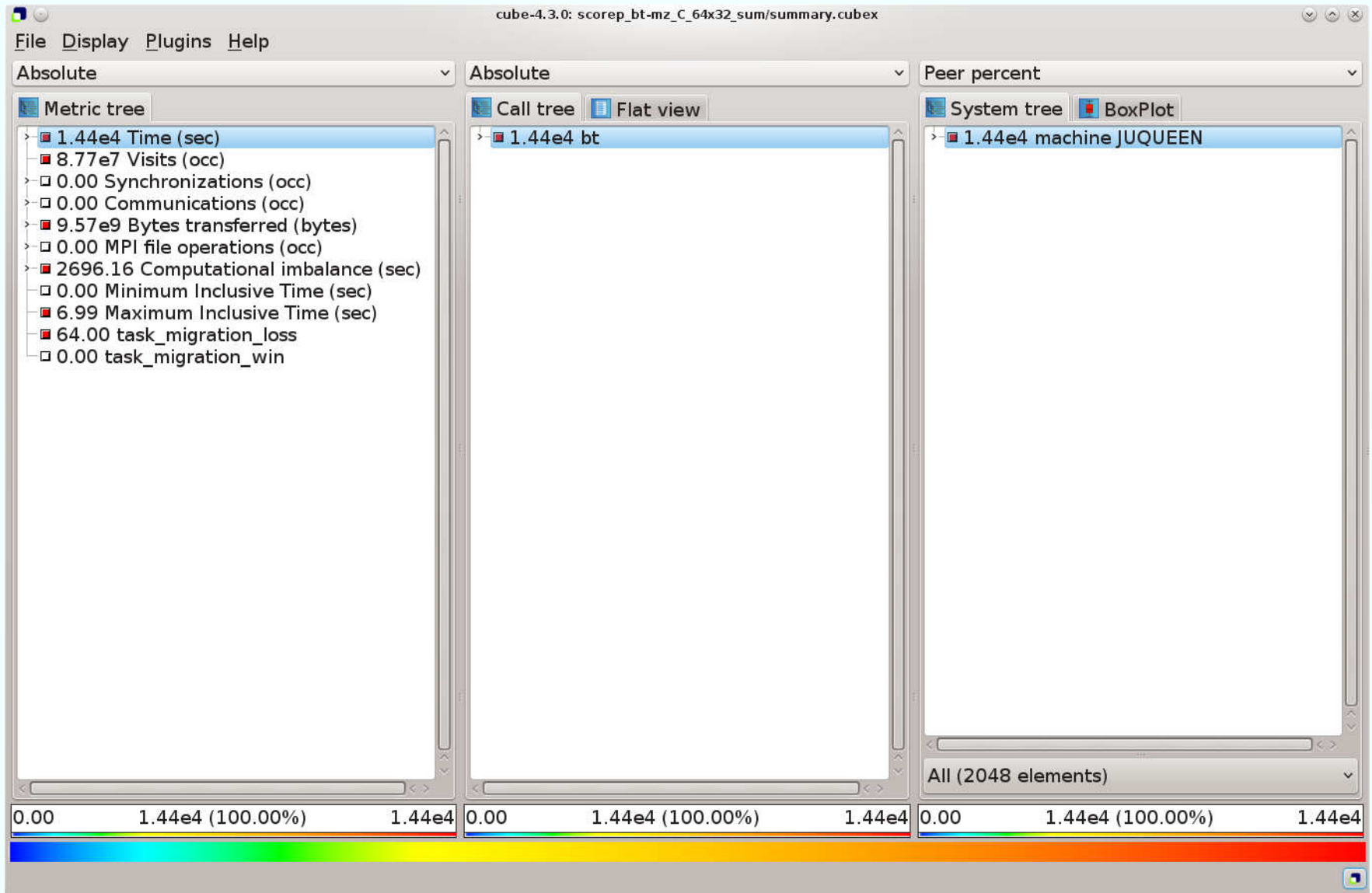
#####
## In the job script: ##
#####

module load UNITE scalasca
scalasca -analyze -f filter.txt \
  runjob --ranks-per-node P --np n [...] --exe ./myprog

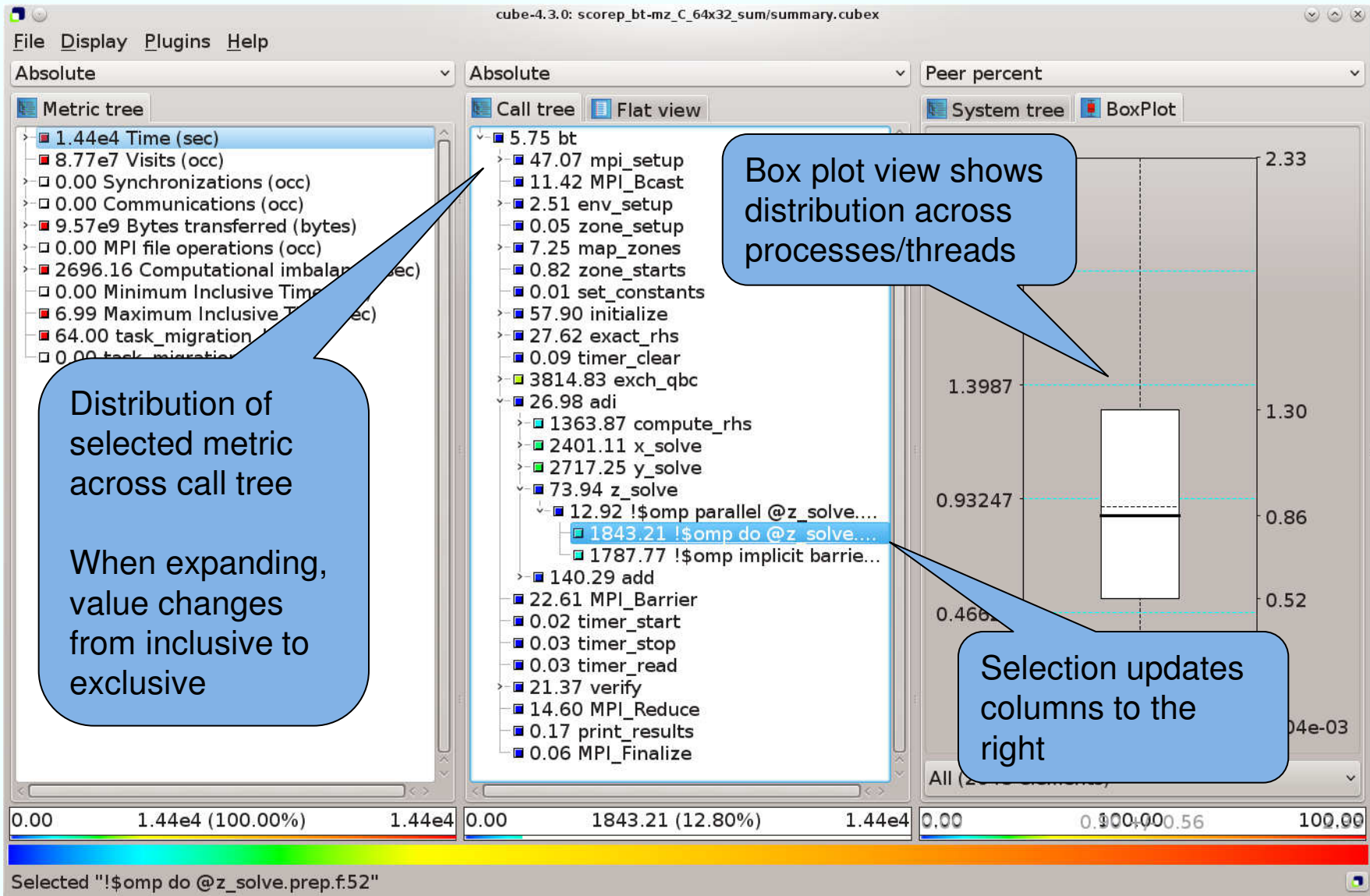
#####
## After job finished: ##
#####

% scalasca -examine scorep_myprog_Pp n x t_sum
```

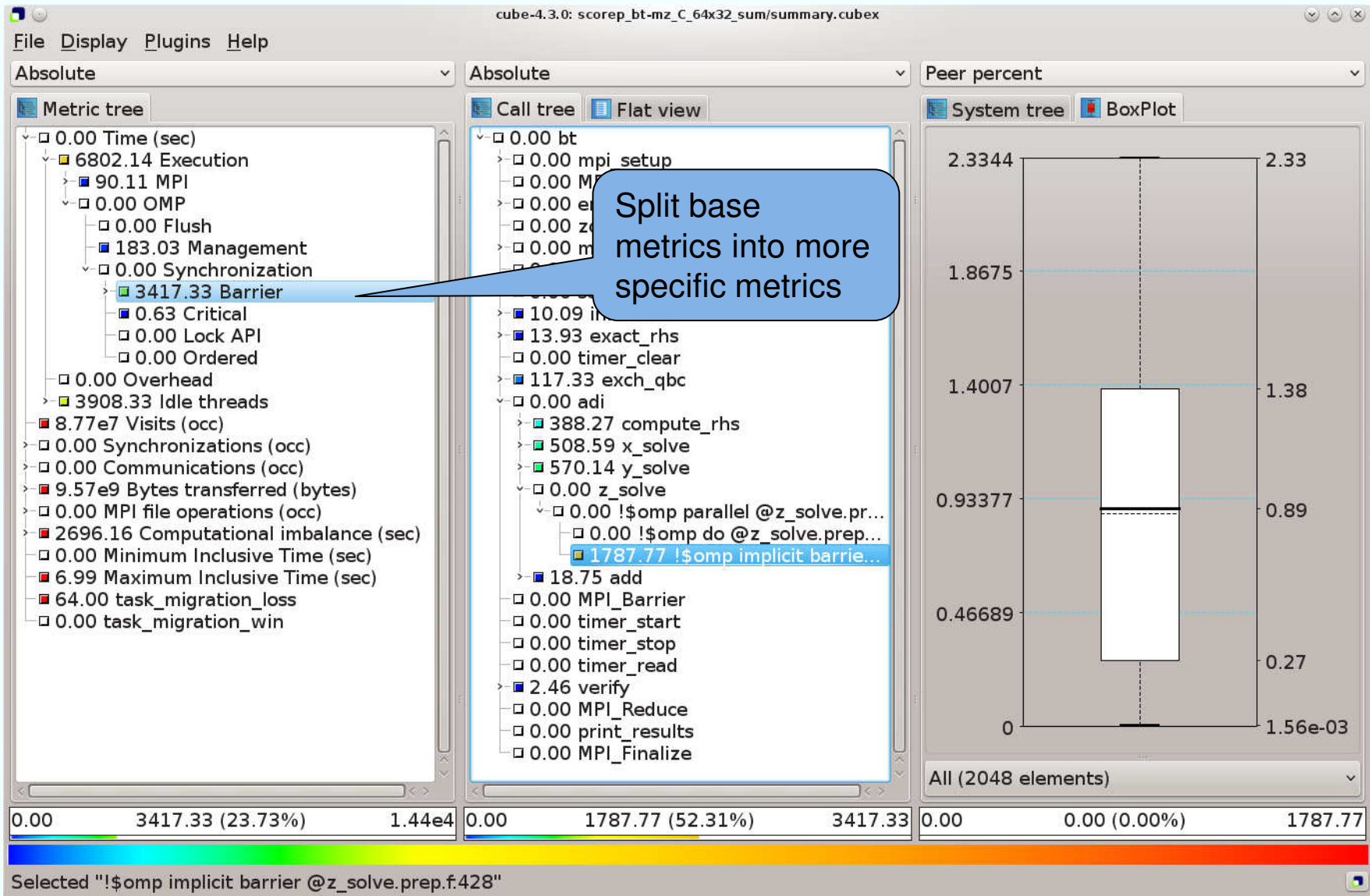
Call-path Profile: Example (cont.)



Call-path Profile: Example (cont.)



Call-path Profile: Example (cont.)



- Sampling support
 - x86 only
- 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

- Record CUDA events using the CUPTI interface

```
% export SCOREP_CUDA_ENABLE=gpu,kernel,idle
```

- Important record types:
 - runtime CUDA runtime API
 - driver CUDA driver API
 - gpu GPU activities
 - kernel CUDA kernels
 - Idle GPU compute idle time
 - memcpy CUDA memory copy
- For all record types consult the Score-P user guide

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

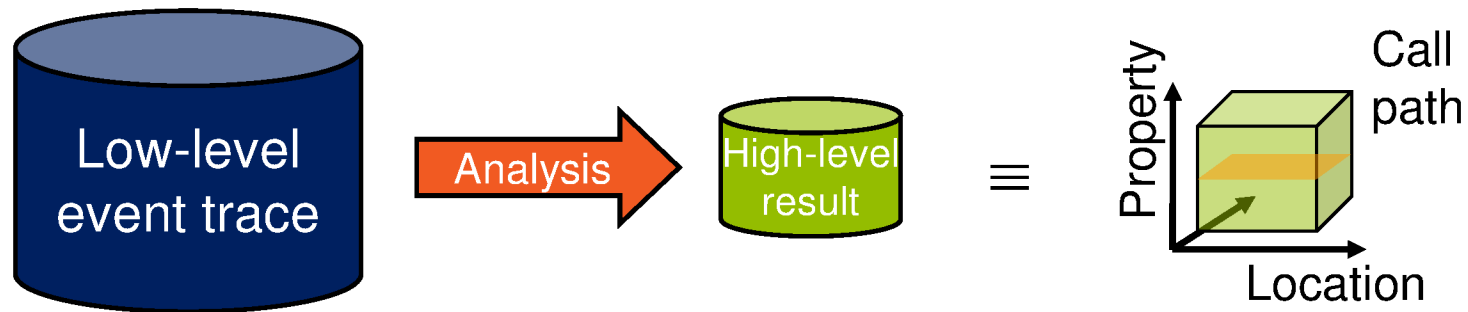
- Score-P supports both PAPI preset and native counters
- Available counters: `papi_avail` or `papi_native_avail`

```
% module load PAPI
```

- Specify using “SCOREP_METRIC_PAPI” environment variable

```
#####  
## In the job script: ##  
#####  
  
module load UNITE scalasca  
export SCOREP_METRIC_PAPI="PAPI_FP_OPS,PAPI_TOT_CYC"  
scalasca -analyze -f filter.txt \  
runjob --ranks-per-node P --np n [...] --exe ./myprog
```

- 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



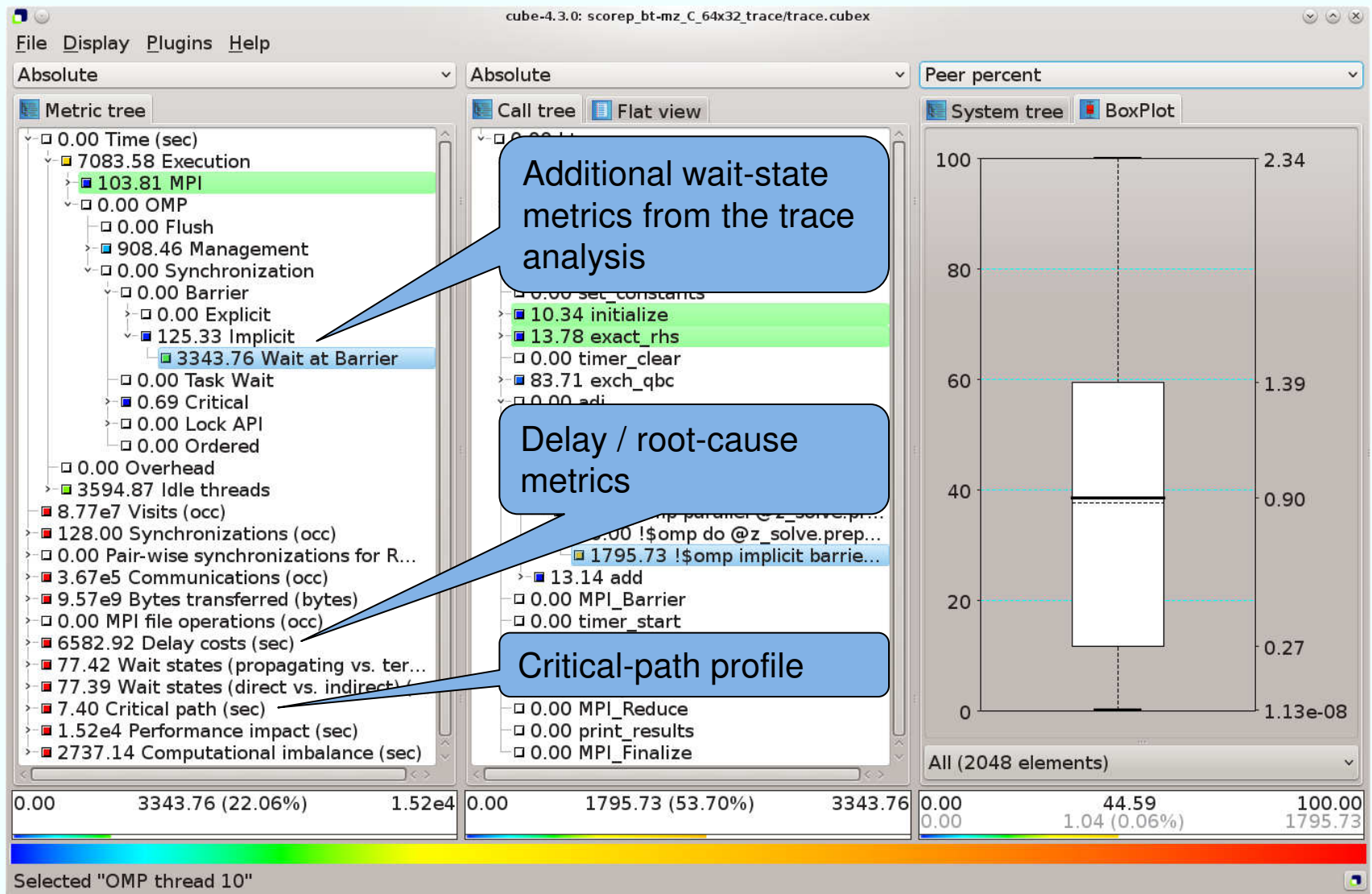
- Advantages
 - Guaranteed to cover the entire event trace
 - Quicker than manual/visual trace analysis
 - Helps to identify hot-spots for in-depth manual analysis

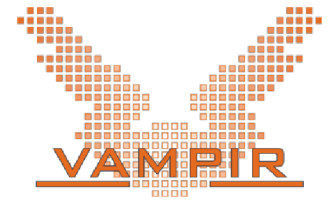
- Enable trace collection & analysis using “-t” option of “scalasca -analyze”:

```
#####  
##  In the job script:  ##  
#####  
  
module load Toolchain Scalasca  
export SCOREP_TOTAL_MEMORY=120MB    # Consult score report  
scalasca -analyze -f filter.txt -t \  
  srun --tasks-per-node P --ntasks n [...] --exe ./myprog
```

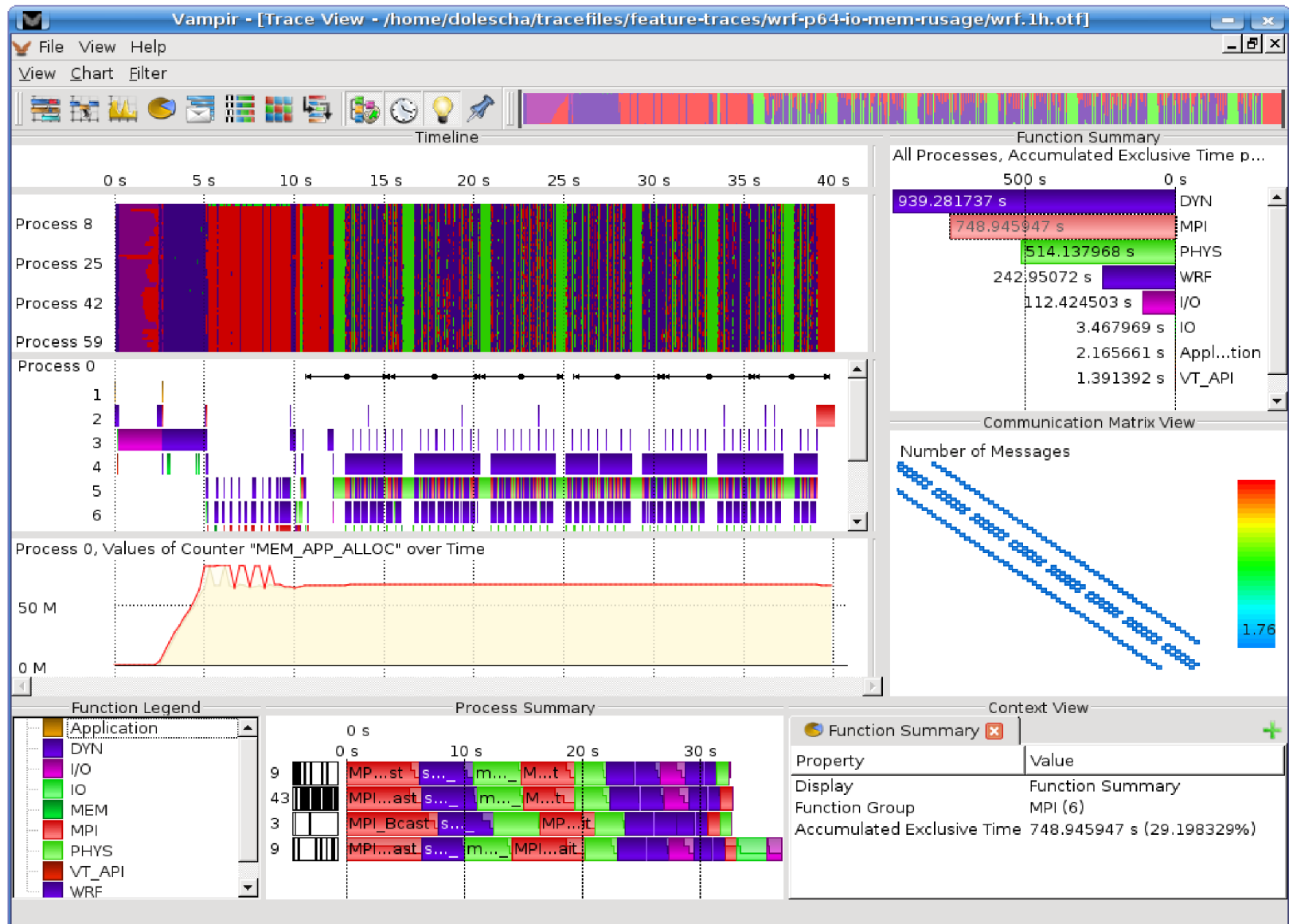
- **ATTENTION:**
 - Traces can quickly become extremely large!
 - Remember to use proper filtering, selective instrumentation, and Score-P memory specification
 - Before flooding the file system, **ask us for assistance!**

Scalasca Trace Analysis Example





- 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

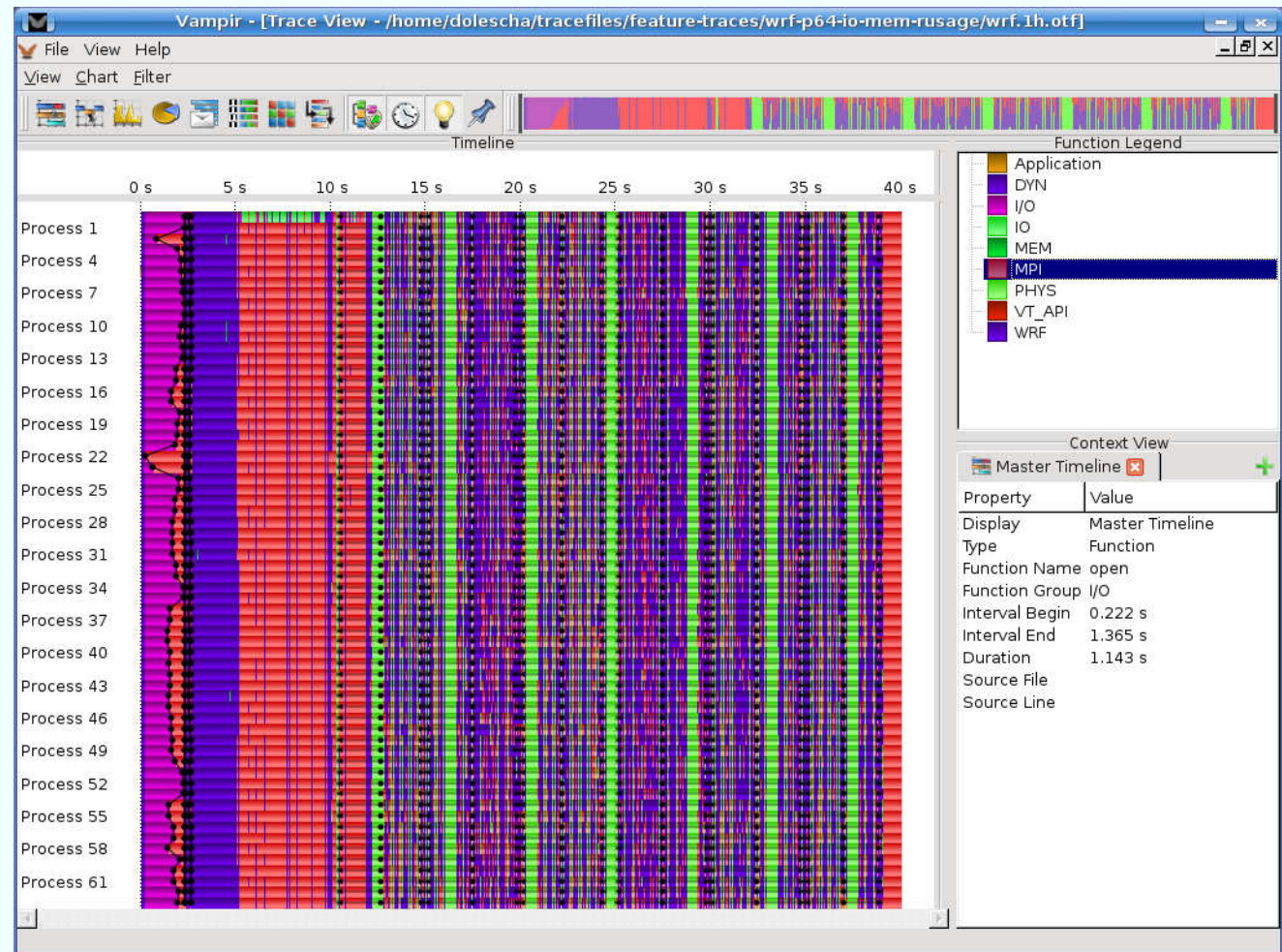


Vampir: Timeline Diagram

- Functions organized into groups

- Coloring by group

- Message lines can be colored by tag or size



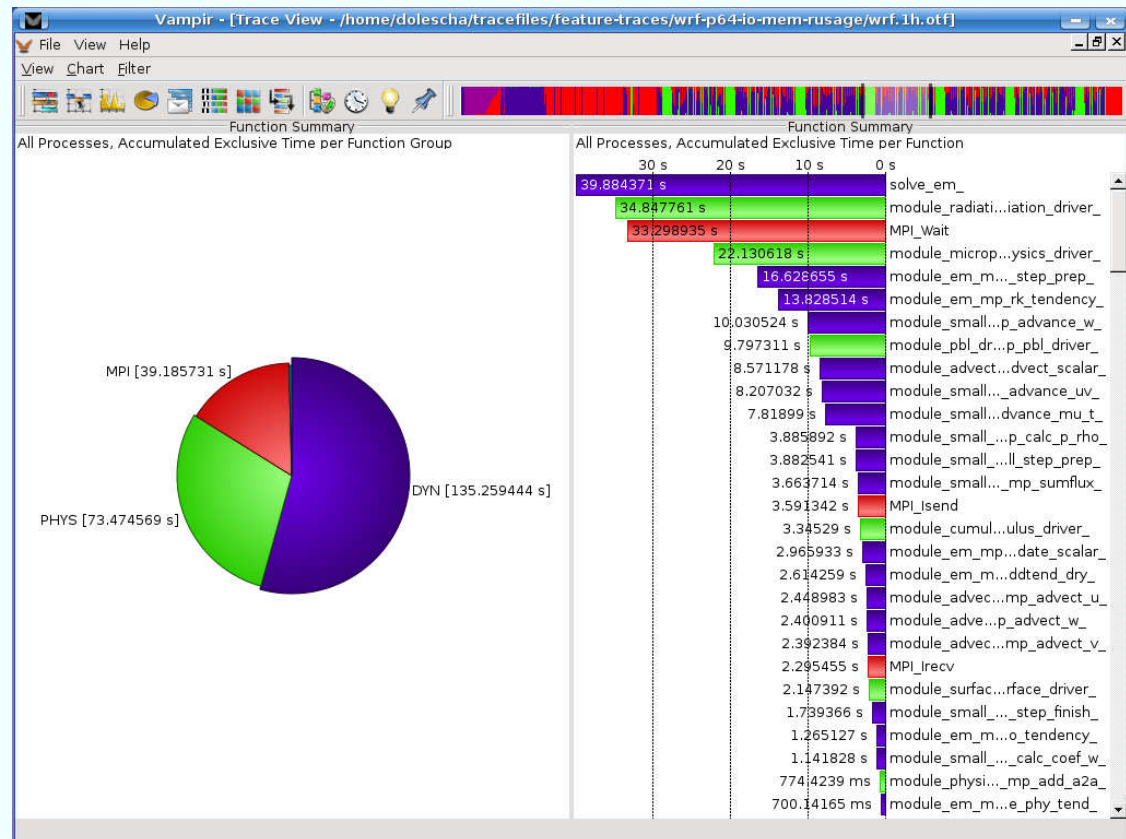
- Information about states, messages, collective and I/O operations available through clicking on the representation

Vampir: Process and Counter Timelines

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

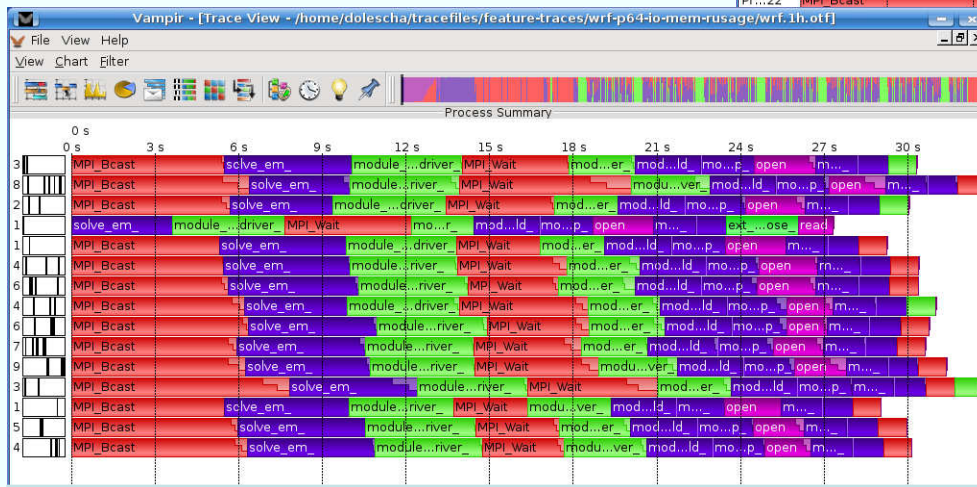
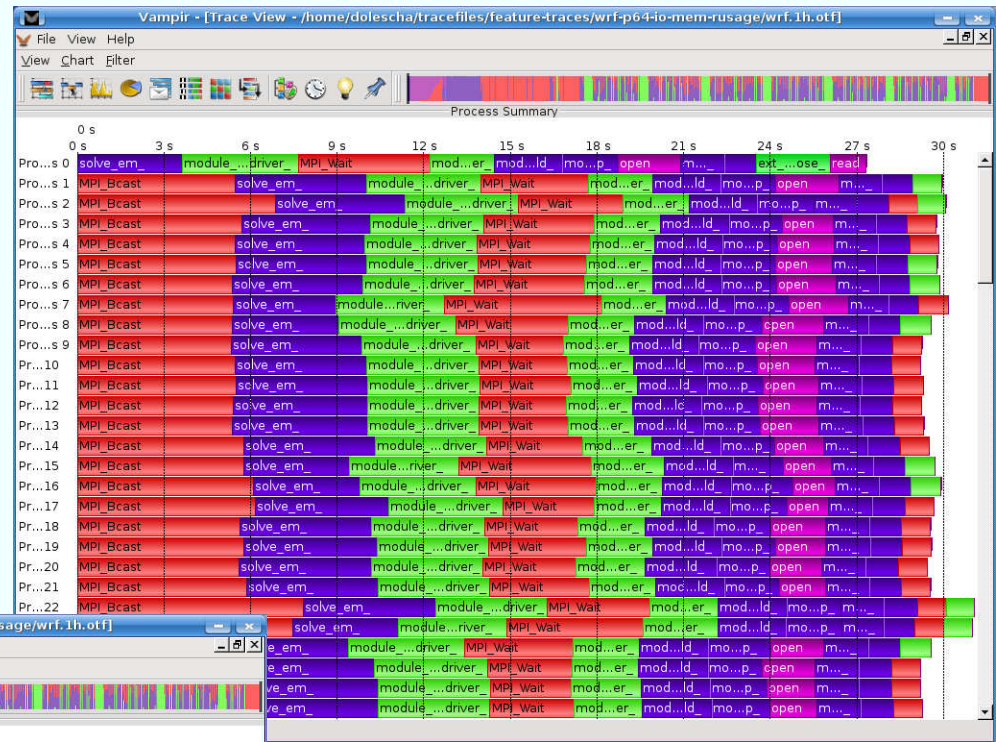


- Available for any part of the trace
⇒ selectable through time line diagram



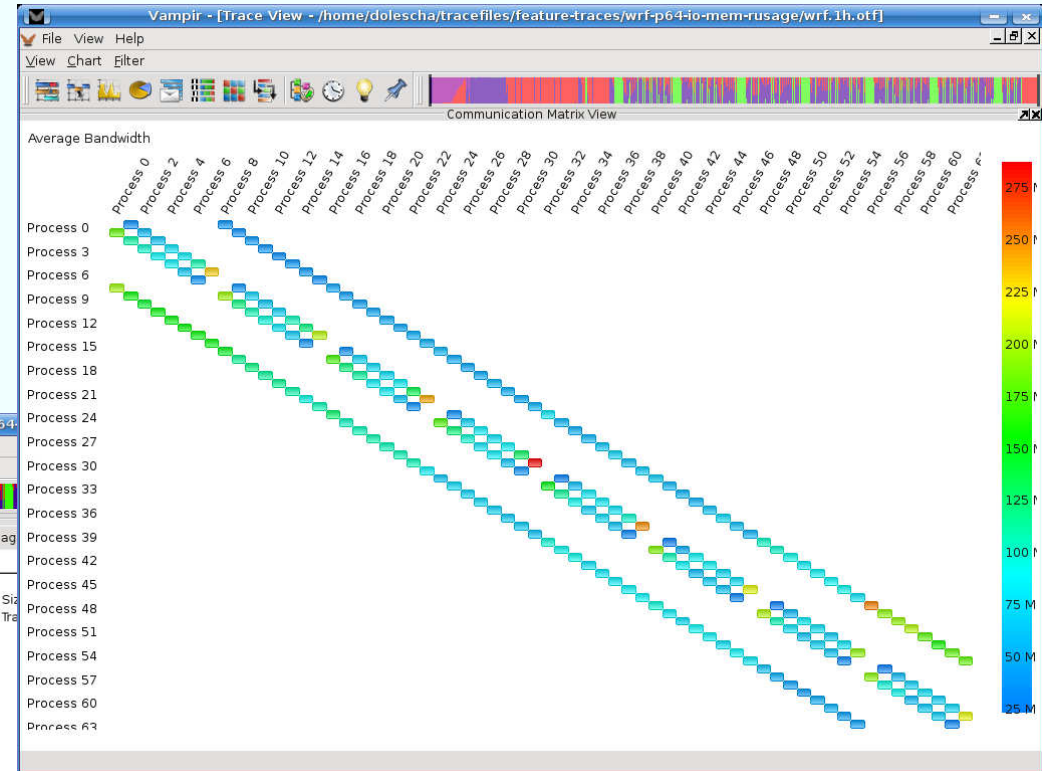
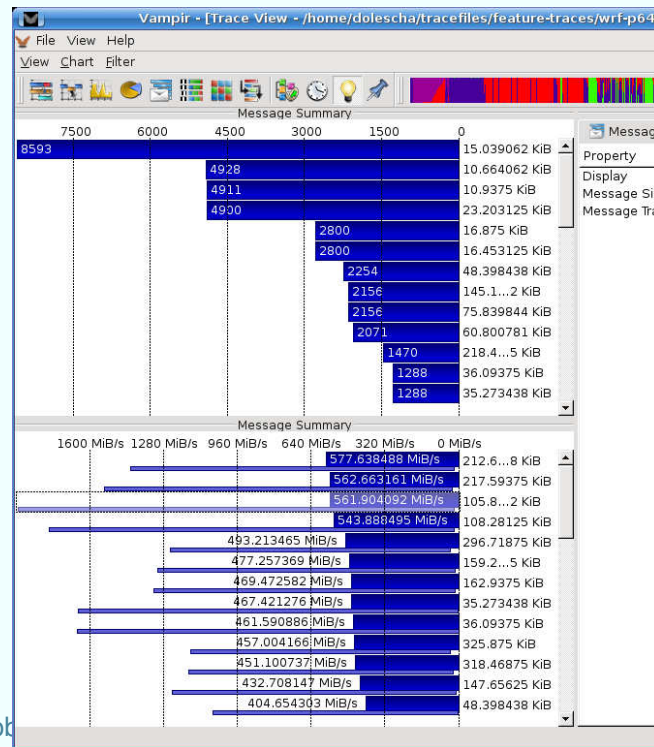
Vampir: Process Summary

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



Vampir: Communication Statistics

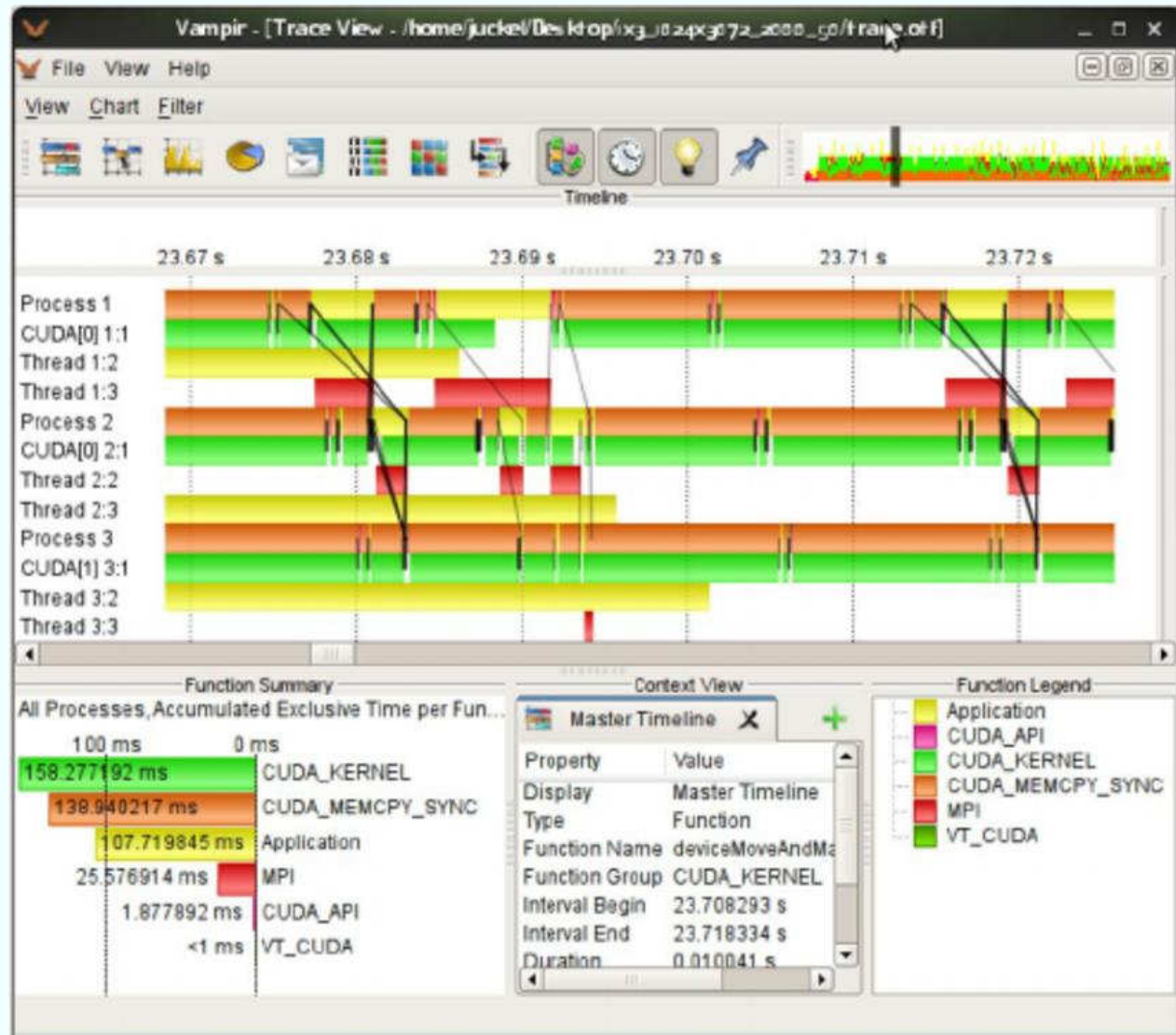
- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics



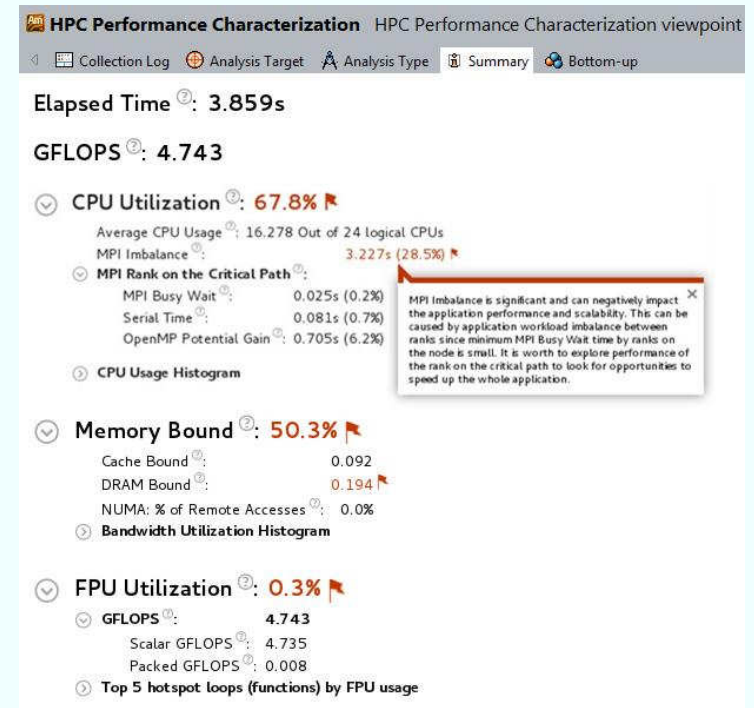
- Available for any part of the trace

Vampir: CUDA Example

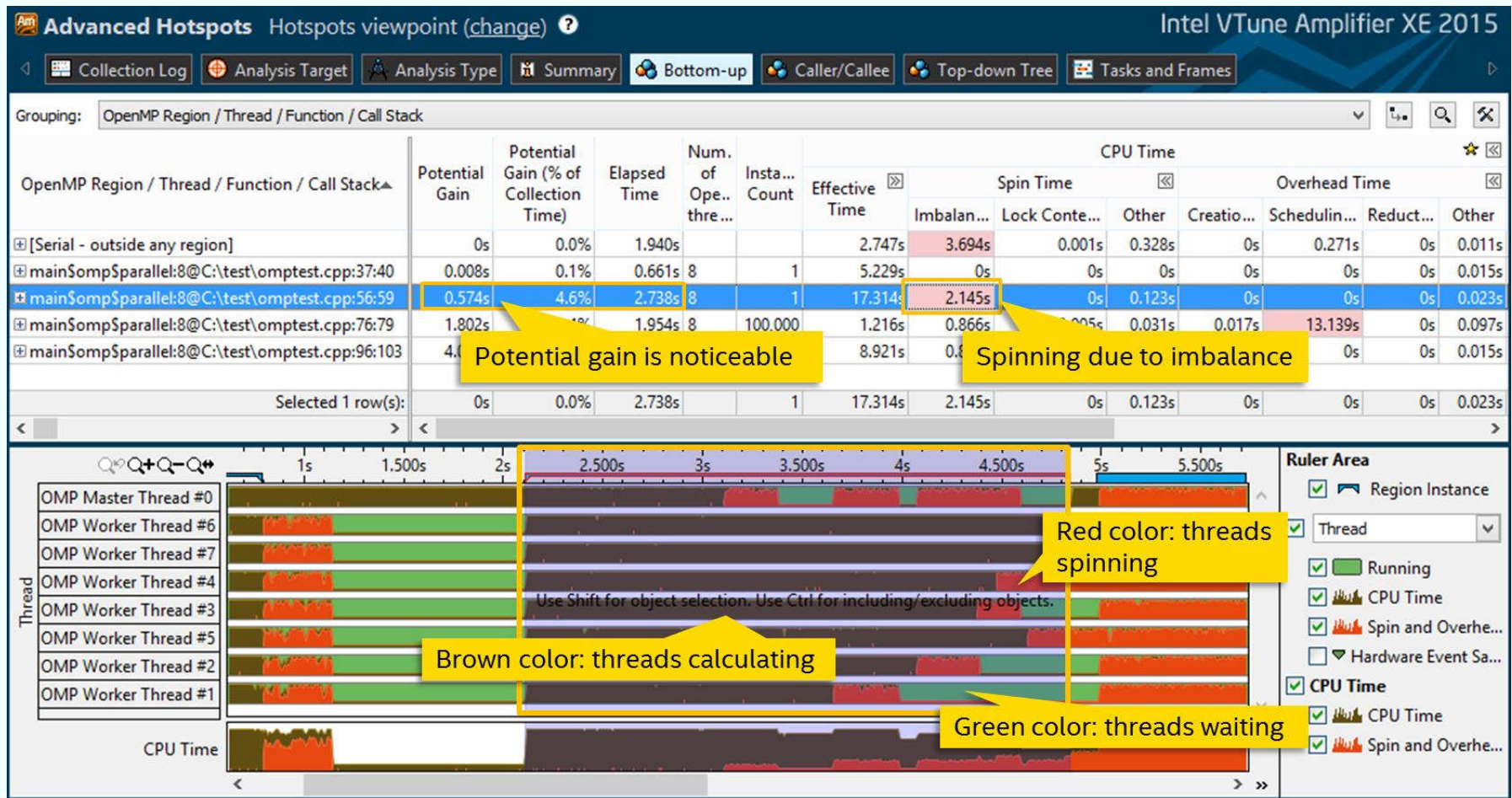
- Detailed information on kernel execution and memory transfers
- All statistics and displays also available for CUDA events



- 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





OpenCL kernels and data transfers

OpenCL kernel invocations

GPU HW metrics

GPU OpenCL command-queue

- 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

» **Ad** Where should I add vectorization and/or threading parallelism? Intel Advisor XE 2016

Summary Survey Report Refinement Reports Annotation Report Suitability Report

Program time: 26.54s Vectorized Not Vectorized FILTER: All Modules All Sources Filters 🔍

Function Call Sites and Loops	Self Time	Total Time	🔥	💡	Loop Type	Why No Vectorization?	Vector... Loops	Instruction Set Analysis	Optimization
⊕ V [loop at loopstl.cpp:3962 in s ...	0.125s	0.125s			Expand	Expand	AVX	Inserts; Maske...	Unrolled
⊖ V [loop at loopstl.cpp:6186 in ...	0.125s	0.125s		1	Collapse	Collapse	AVX		Unrolled
↳ [loop at loopstl.cpp:6186 in s ...	0.062s	0.062s			Remainder				
↳ V [loop at loopstl.cpp:...					Vectorized (Body)		AVX		Unrolled
↳ [loop at loopstl.cpp:5625 in ...					Scalar	loop with early exits cannot be ...			

Scalar Loop. Not vectorized. Loop with early exits cannot be vectorized unless it meets search loop id criteria
No loop transformations were applied

Loop may have several parts or versions

Loop summary

Vectorization and compiler optimization details

Top Down Source Loop Assembly Assistance Recommendations **Compiler Diagnostic Details**

Issue: Ineffective peeled/remainder loop(s) present

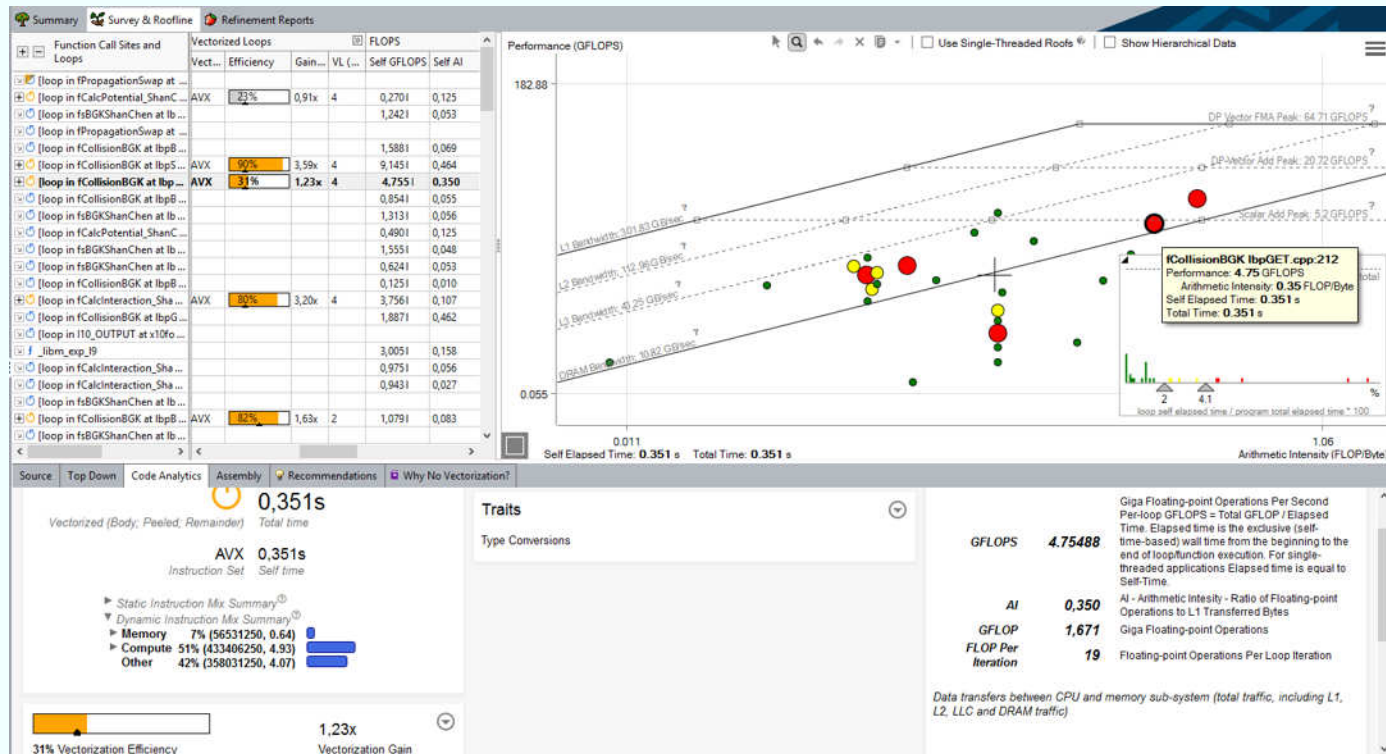
All or some [source loop](#) iterations are not executing in the [loop body](#). Improve performance by moving source loop iterations from [peeled/remainder](#) loops to the loop body.

🔍 **Add data padding**
Potential performance gain: Information not available
Confidence this recommendation applies to your code: Information not available until Beta Update release
The [trip count](#) is not a multiple of [vector length](#). To fix: Do one of the following:

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

Recommendations for optimization

Intel Advisor – Roofline



- Multi-platform sampling-based call-path profiler
- Works on unmodified, optimized executables
- <http://hpctoolkit.org>
- Advantages:
 - Overhead can be easily controlled via sampling interval
 - Advantageous for complex C++ codes with many small functions
 - Loop-level analysis (sometimes even individual source lines)
 - Supports POSIX threads
- Disadvantages:
 - Statistical approach that might miss details
 - MPI/OpenMP time displayed as low-level system calls

- Specified via environment variable `HPCRUN_EVENT_LIST`
- General format:
 `"name@interval [;name@interval ...]"`
- Possible sample sources:
 - `WALLCLOCK`
 - `PAPI counters`
 - `IO` (use w/o interval spec)
 - `MEMLEAK` (use w/o interval spec)
- Interval: given in microseconds
 - E.g., `10000` → 100 samples per second

Example: hpcviewer

hpcviewer: sor <@jj28103>

File Debug Help

sor.c

```
670  fkjm1 = Field[k][j-1];
671  rkj   = Rhs[k][j];
672  akj   = Ans[k][j];
673  for (i=1+mod; i<=nxl; i+=2)
674  {
675      delta = omega*( fkj[i+1] + fkj[i-1] +fkjpl[i] + fkjm1[i]
676                  -4.0*fkj[i] - rkj[i] );
677
678      tmpres += fabs(delta);
679
680      fkj[i] = fkj[i] + delta;
681
682      tmperr += fabs(fkj[i] - akj[i]);
683  }
```

associated source code

Calling Context View Callers View Flat View

Scope WALLCLOCK (us).[0.0] (I) WALLCLOCK (us).[0.0] (E) WALLCLOCK (us).[1.0] (I) WALLCLOCK (us).[1.0] (E)

Experiment Aggregate Metrics	4.79e+06	100 %	4.79e+06	100 %	4.76e+06	100 %	4.76e+06
main	4.79e+06	100 %			4.76e+06	100 %	
sor_iter	4.68e+06	97.7%	4.01e+06	83.7%	4.66e+06	97.7%	3.95e+06
loop at sor.c: 344	2.67e+06	55.7%	2.00e+06	41.7%	2.71e+06	56.8%	2.00e+06
inlined from sor.c: 658	2.00e+06	41.7%	2.00e+06	41.7%	2.00e+06	42.0%	2.00e+06
loop at sor.c: 662	2.00e+06	41.7%			2.00e+06	42.0%	
loop at sor.c: 673	2.00e+06	41.7%	3.55e+04	0.7%	2.00e+06	42.0%	
loop at sor.c: 673	1.96e+06	41.0%	1.96e+06	41.0%	2.00e+06	42.0%	2.00e+06
inlined from sor.c: 331	8.38e+05	17.5%	8.38e+05	17.5%	7.06e+05	14.8%	7.06e+05
sor.c: 675	6.59e+05	13.8%	6.59e+05	13.8%	6.23e+05	13.1%	6.23e+05
sor.c: 682	2.40e+05	5.0%	2.40e+05	5.0%	3.96e+05	8.3%	3.96e+05
sor.c: 678	1.08e+05	2.2%	1.08e+05	2.2%	8.39e+04	1.8%	8.39e+04

Callpath to hotspot

191M of 400M

- **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)

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

The total wallclock time was spent as follows:

CPU 56.5% 

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

MPI 43.5% 

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

I/O 0.0% 

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

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

CPU

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

Scalar numeric ops 27.7% 

Vector numeric ops 11.3% 

Memory accesses 60.9% 

Other 0.0% 

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

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

I/O

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

Time in reads 0.0% 

Time in writes 0.0% 

Estimated read rate 0 bytes/s 

Estimated write rate 0 bytes/s 

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

MPI

Of the 43.5% total time spent in MPI calls:

Time in collective calls 8.2% 

Time in point-to-point calls 91.8% 

Estimated collective rate 169 Mb/s 

Estimated point-to-point rate 50.6 Mb/s 

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

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage 82.5 Mb 

Peak process memory usage 89.3 Mb 

Peak node memory usage 7.4% 

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

Allinea Performance Reports: Example

```
% module load AllineaPerformanceReports

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

perf-report --mpi="slurm" \
  srun --procs-per-node= $P$  --nodes= $n$  [...] ./myprog [args]

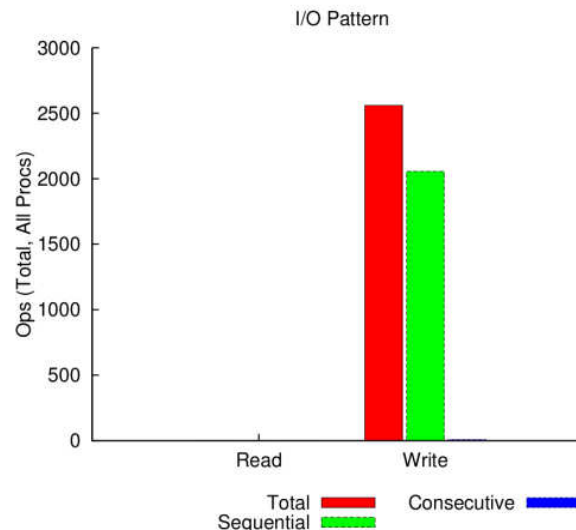
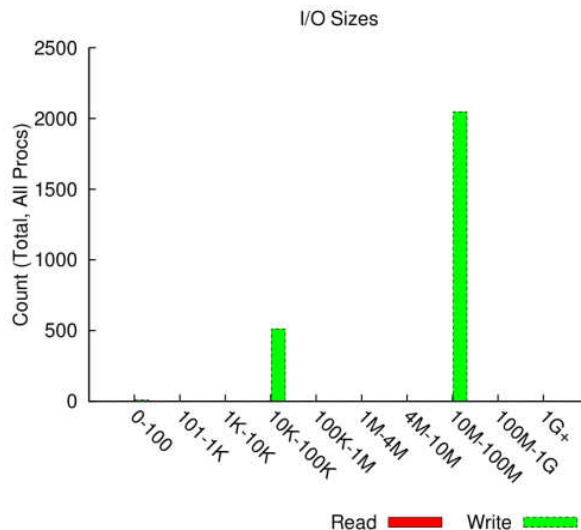
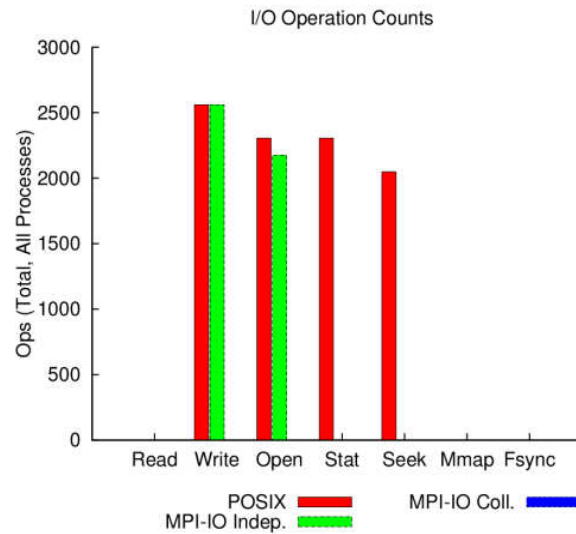
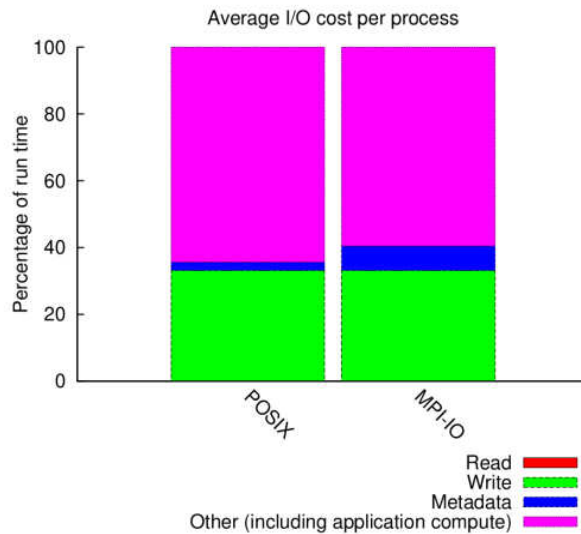
#####
## After job finished: ##
#####

% less myprog_<NP>p_<DATE>.txt
% firefox myprog_<NP>p_<DATE>.html
```

- 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, ...
 - Doesn't support mpif90 on BlueGene systems (use mpif77)
- 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



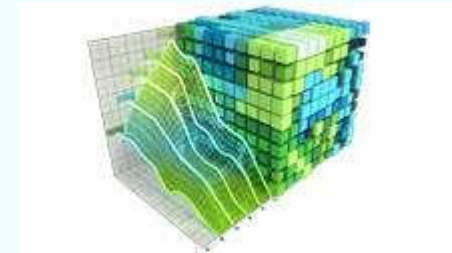
```
% load compiler and MPI module
% module load darshan-runtime darshan-util

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

export LD_PRELOAD=$EBROOTDARSHANMINRUNTIME/lib/libdarshan.so
export DARSHAN_LOG_PATH=$PWD
export DARSHAN_LOGFILE=darshan.log
srun --tasks-per-node P --ntasks n [...] ./myprog [args]

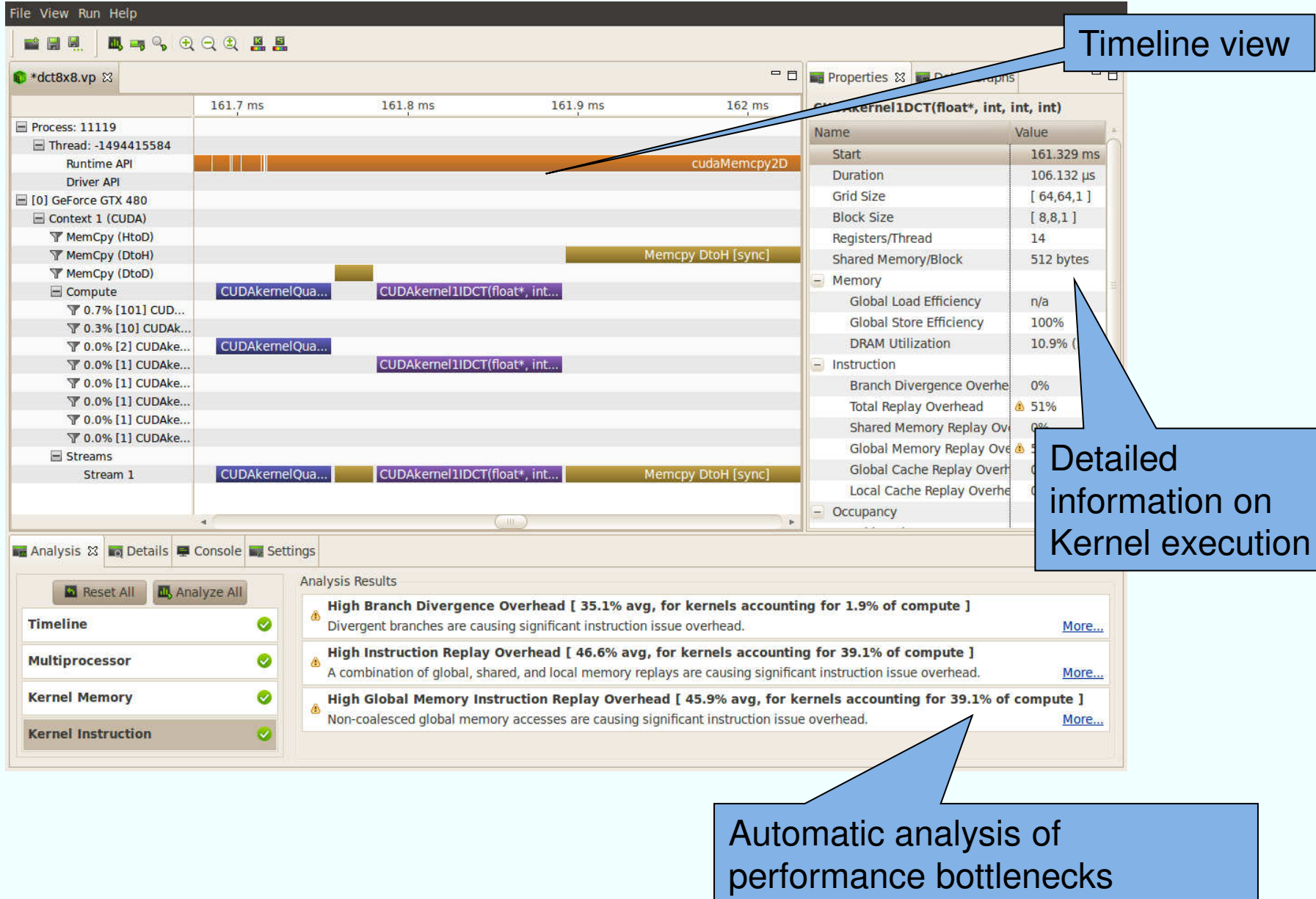
#####
##  After job finished:  ##
#####

% darshan-job-summary.pl darshan.log
% gv darshan.pdf
```

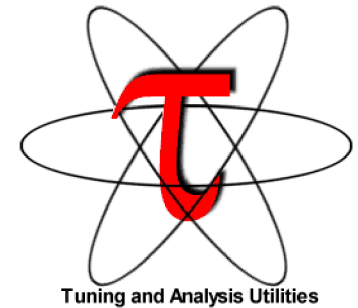


- Part of the CUDA Toolkit
- Supports all CUDA enabled GPUs
- Supports CUDA and OpenACC on Windows, OS X and Linux
- Unified CPU and GPU Timeline
- CUDA API trace
 - Memory transfers, kernel launches, and other API functions
- Automated performance analysis
 - Identify performance bottlenecks and get optimization suggestions
- Guided Application Analysis
- Power, thermal, and clock profiling

NVIDIA Visual Profiler: Example

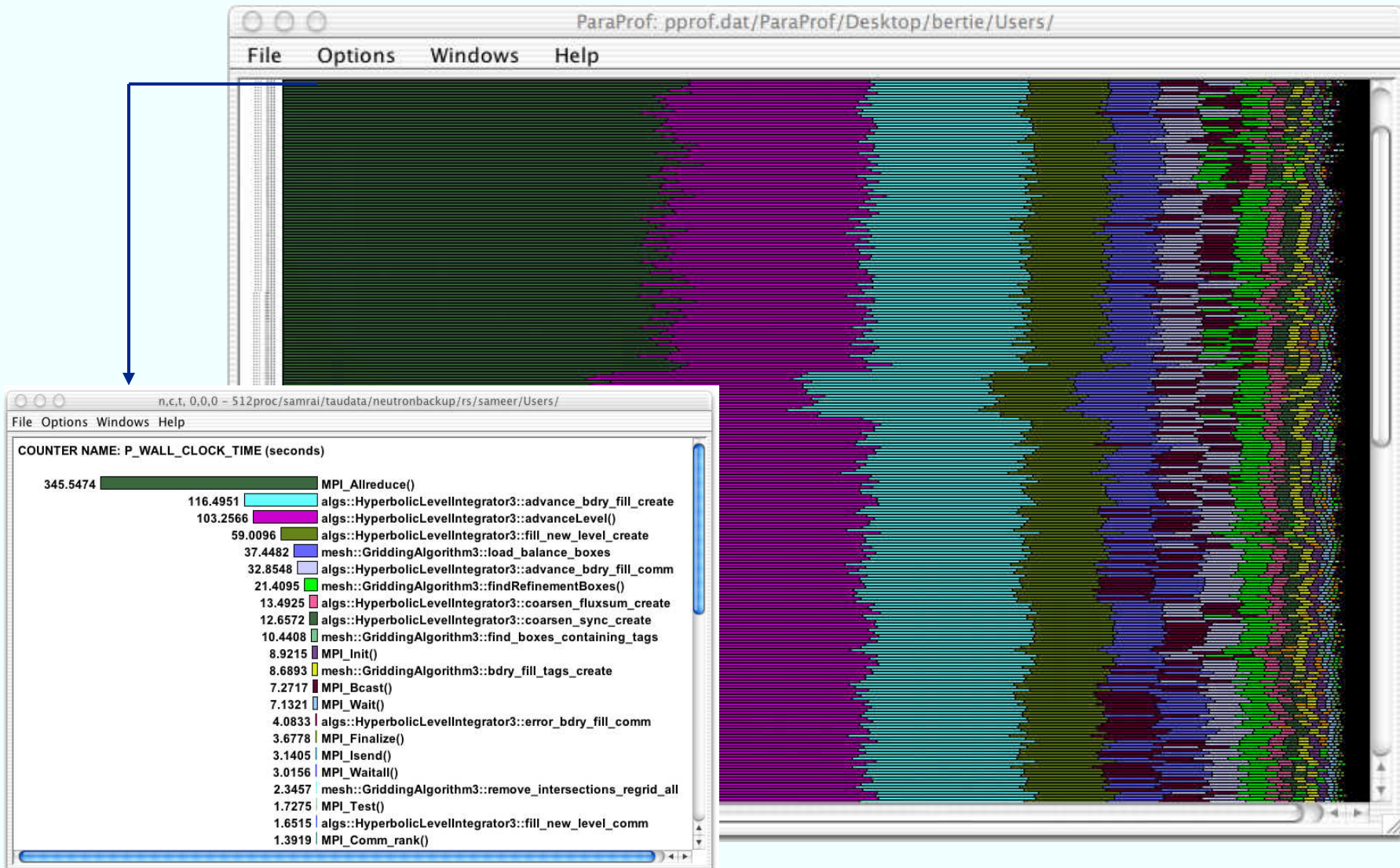


- Very portable tool set for instrumentation, measurement and analysis of parallel multi-threaded applications
- <http://tau.uoregon.edu/>
- Supports
 - Various profiling modes and tracing
 - Various forms of code instrumentation
 - C, C++, Fortran, Java, Python
 - MPI, multi-threading (OpenMP, Pthreads, ...)
 - Accelerators

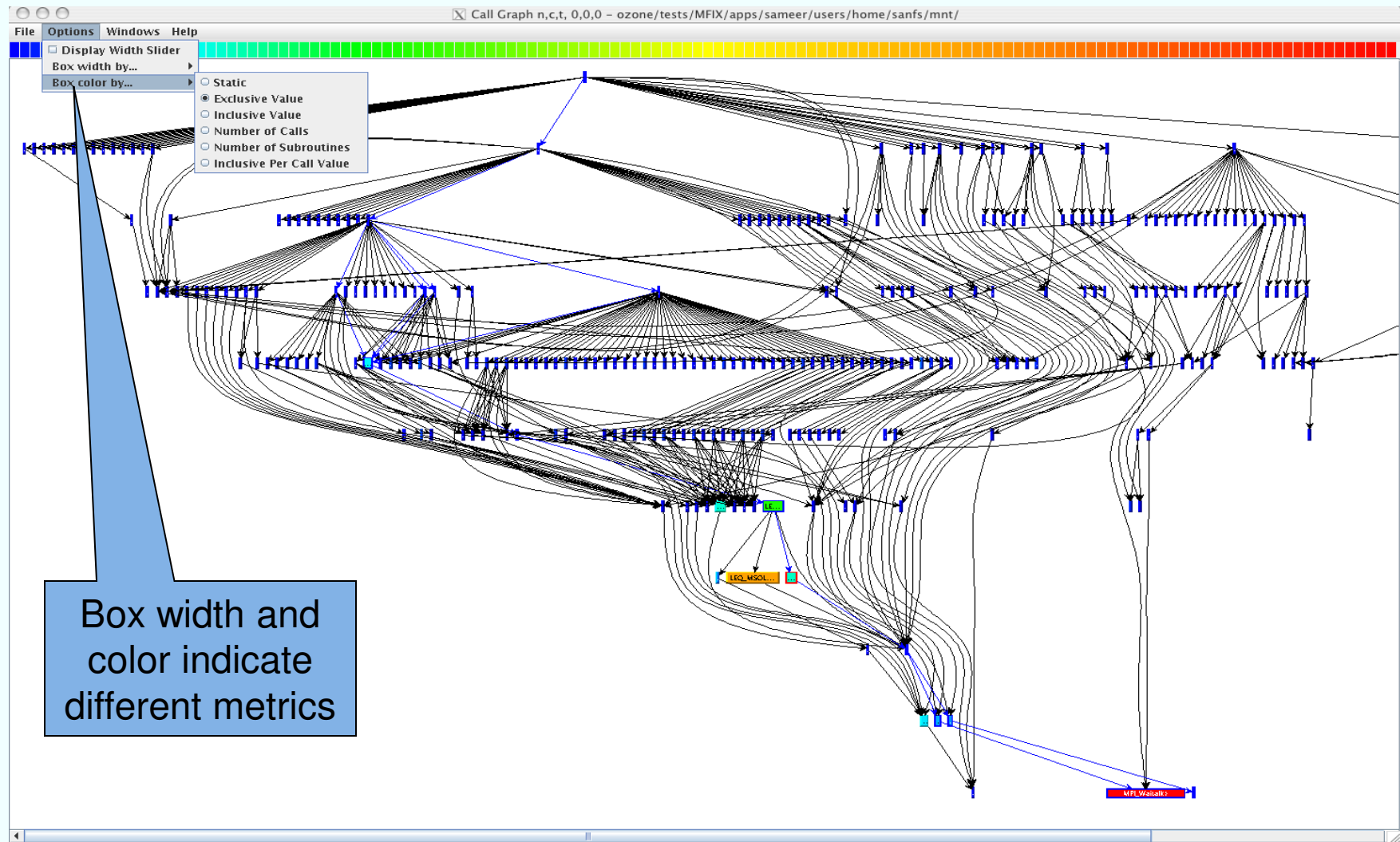


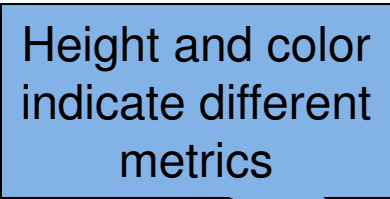
- Flexible instrumentation mechanisms at multiple levels
 - Source code
 - manual
 - automatic
 - C, C++, F77/90/95 (Program Database Toolkit (PDT))
 - OpenMP (directive rewriting with **Opari**)
 - Object code
 - pre-instrumented libraries (e.g., MPI using **PMPI**)
 - statically-linked and dynamically-loaded (e.g., Python)
 - Executable code
 - dynamic instrumentation (pre-execution) (**DynInst**)
 - virtual machine instrumentation (e.g., Java using **JVMPI**)
- Support for **performance mapping**
- Support for **object-oriented** and **generic** programming

TAU: Basic Profile View



TAU: Callgraph Profile View





- To check latest status and versions
 - “module spider TOOL”
- Websites
 - <http://www.fz-juelich.de/ias/jsc/juwels/>
 - <http://www.fz-juelich.de/ias/jsc/jureca/>
 - User Info
 - Parallel Debugging ()
 - Parallel Performance Analysis ()
 - <http://www.vi-hps.org/training/material/>
 - Performance Tools LiveDVD image
 - Links to tool websites and documentation
 - Tutorial slides



- For general support: `sc@fz-juelich.de`
- Tool-specific support via corresponding mailing lists
 - Score-P: `support@score-p.org`
 - Scalasca: `scalasca@fz-juelich.de`
- Workshops and Trainings:
 - Regular VI-HPS Tuning Workshops
 - Several days
 - Multiple tools, e.g. Score-P, Scalasca, Vampir, TAU, ...
 - Bring-your-own-code
 - <http://www.vi-hps.org/training/tws/>
 - JSC Porting and Tuning Workshop Series
 - WS's and trainings at several HPC conferences