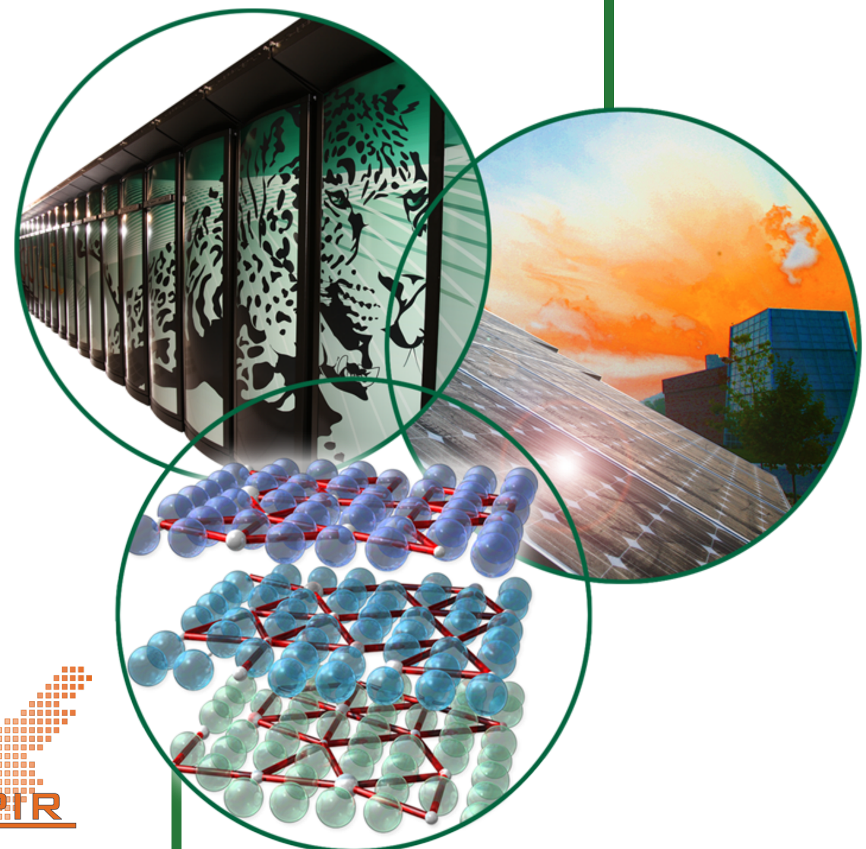
Performance Analysis at Scale:
The Score-P Tools Infrastructure



23 May 2016

Frank Winkler

On-site contractor for
Vampir, CSMD

cube scalasca

VAMPIR

Score-P
Scalable performance measurement
infrastructure for parallel codes

U.S. DEPARTMENT OF
ENERGY

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Disclaimer

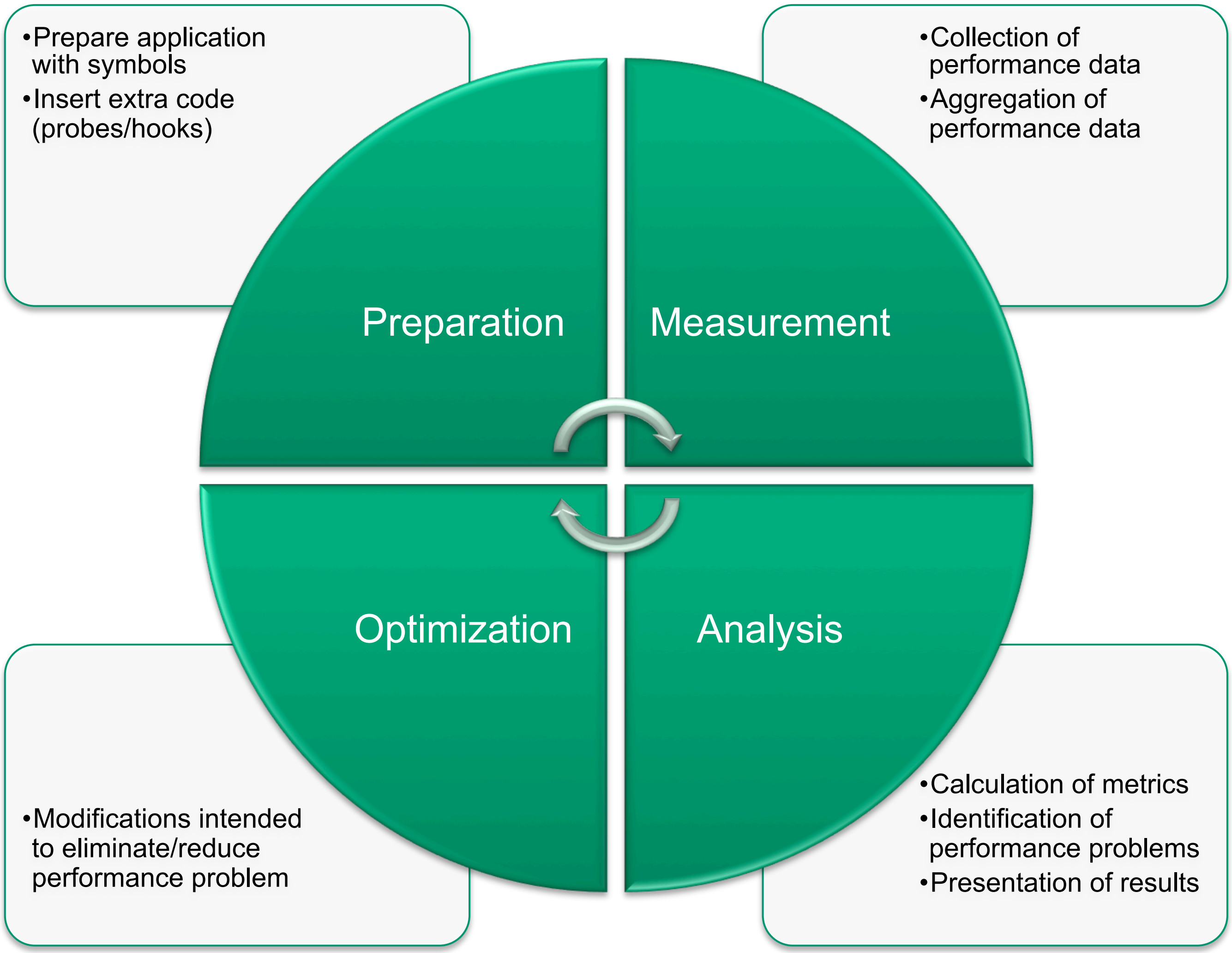
It is extremely easy to waste performance!

- Bad MPI (50-90%)
- No node-level parallelism (94%)
- No vectorization (75%)
- Bad memory access pattern (99%)
- In sum: 0.008% of the peak performance (about 2 teraflops of Titan)

Disclaimer (2)

Performance tools will not automatically make your code run faster. They help you understand, what your code does and where to put in work.

Performance engineering workflow



Agenda

Performance Analysis Approaches

- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes

- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools

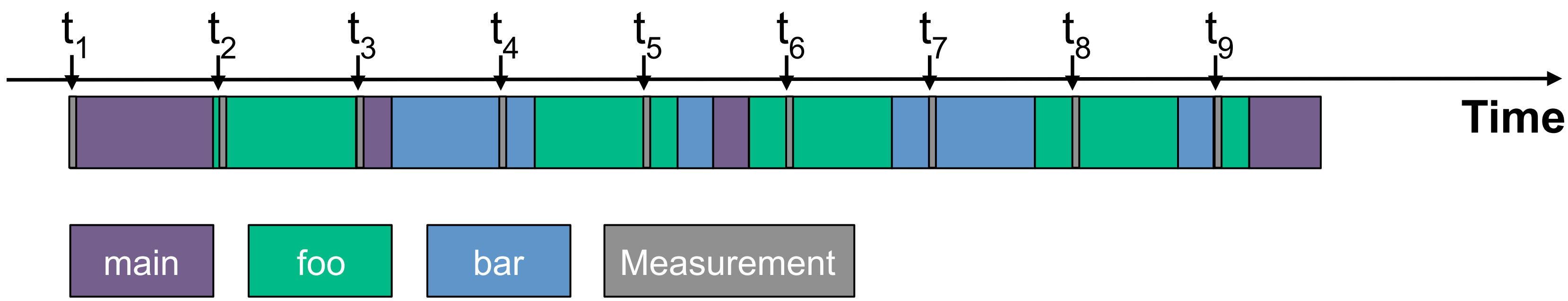
- Cube
- Vampir

Demo

- Performance Analysis of Jacobi Solver on Titan

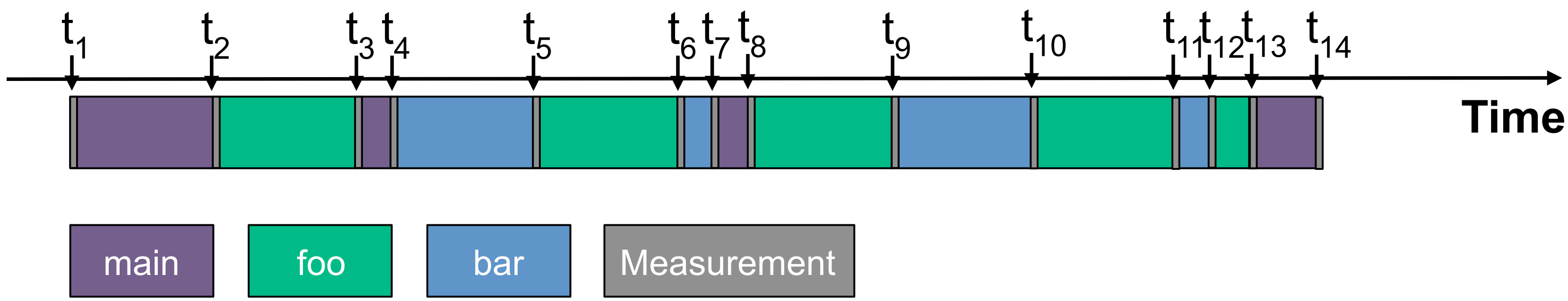
Conclusions

Sampling



- Running program is periodically interrupted to take measurement
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

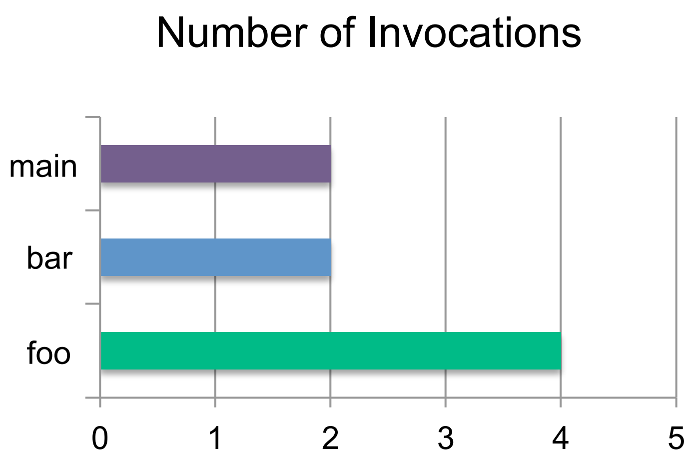
Instrumentation

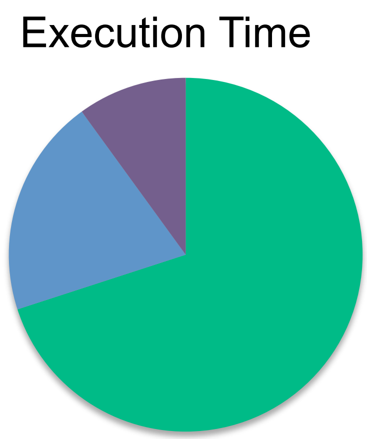


- Measurement code is inserted such that every event of interest is captured directly
  - Can be done in various ways
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions

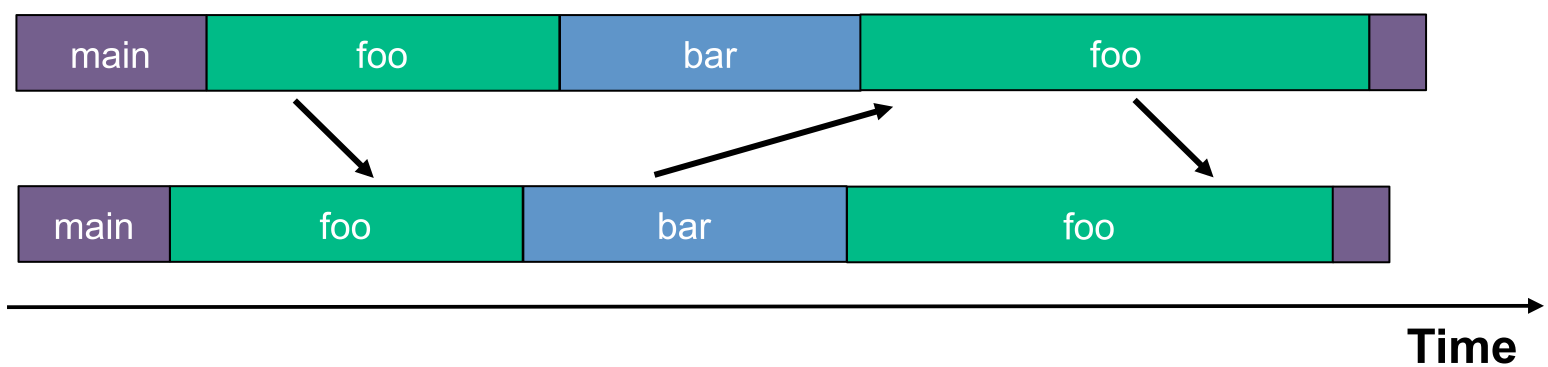
Profiling vs. Tracing

- Statistics

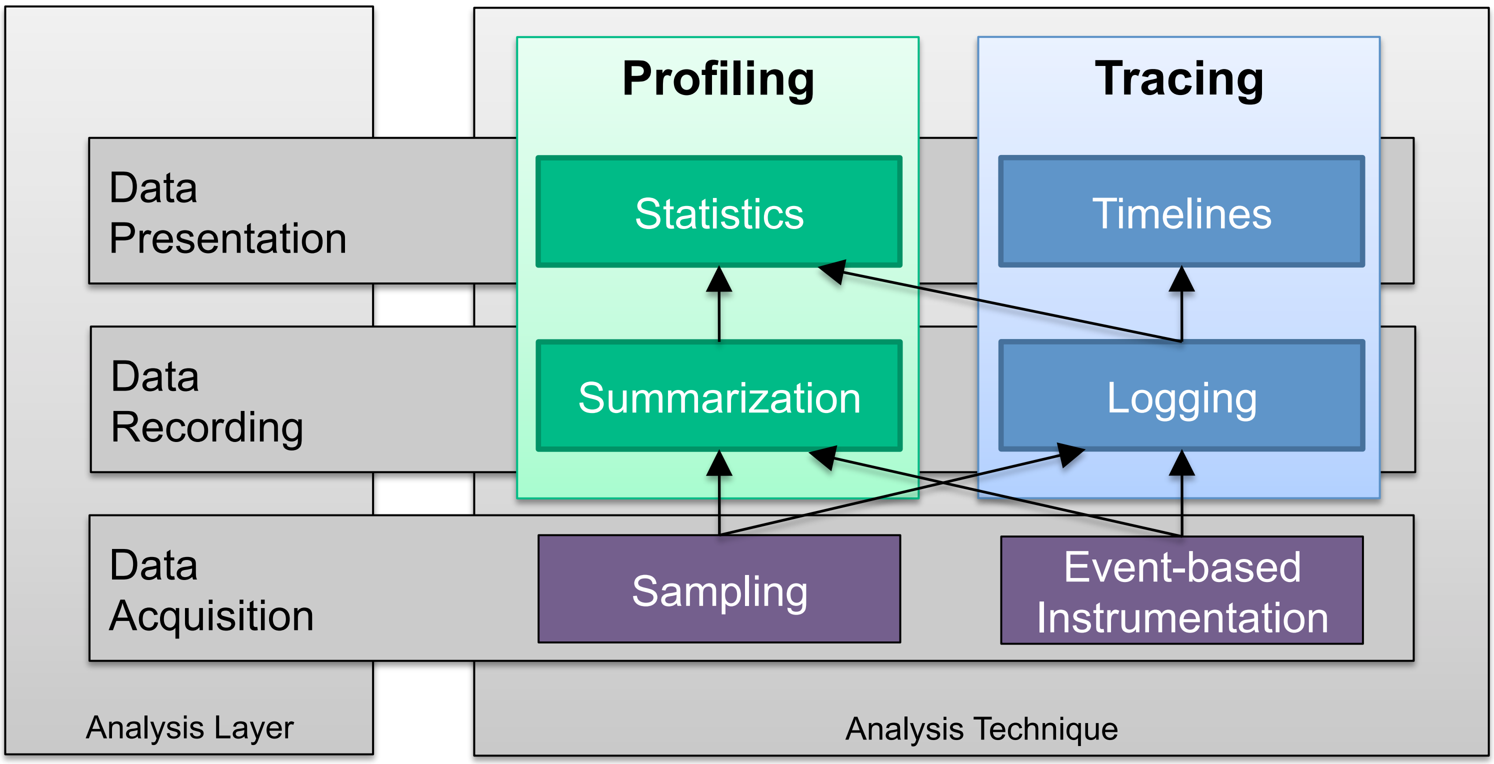




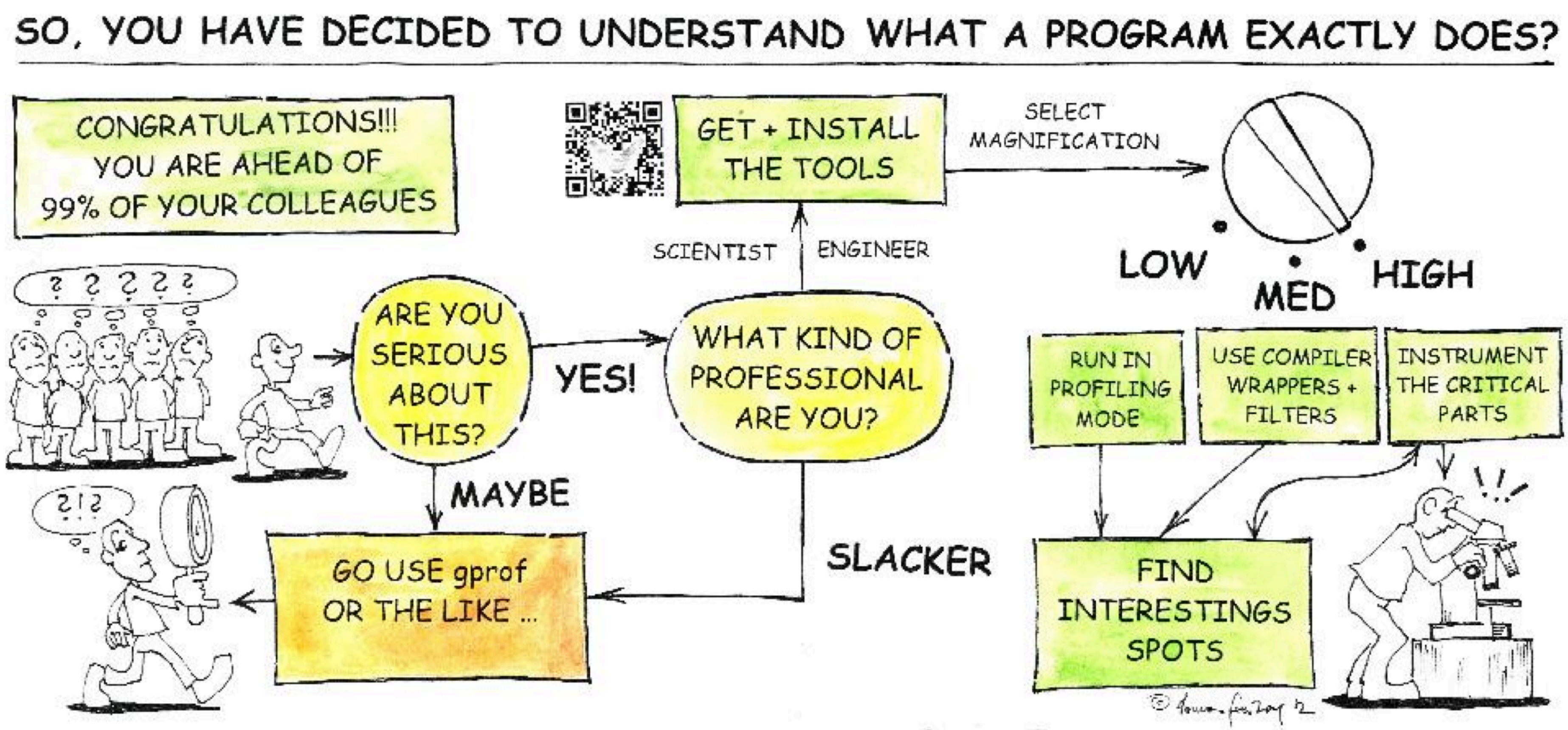
- Timelines



Terms Used and How They Connect



So what is the right choice?



Agenda

Performance Analysis Approaches

- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes

- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools

- Cube
- Vampir

Demo

- Performance Analysis of Jacobi Solver on Titan

Conclusions

Score-P: Motivation

- Several performance tools co-exist
- Separate measurement systems and output formats
- Complementary features and overlapping functionality
- Redundant effort for development and maintenance
- Limited or expensive interoperability
- Complications for user experience, support, training

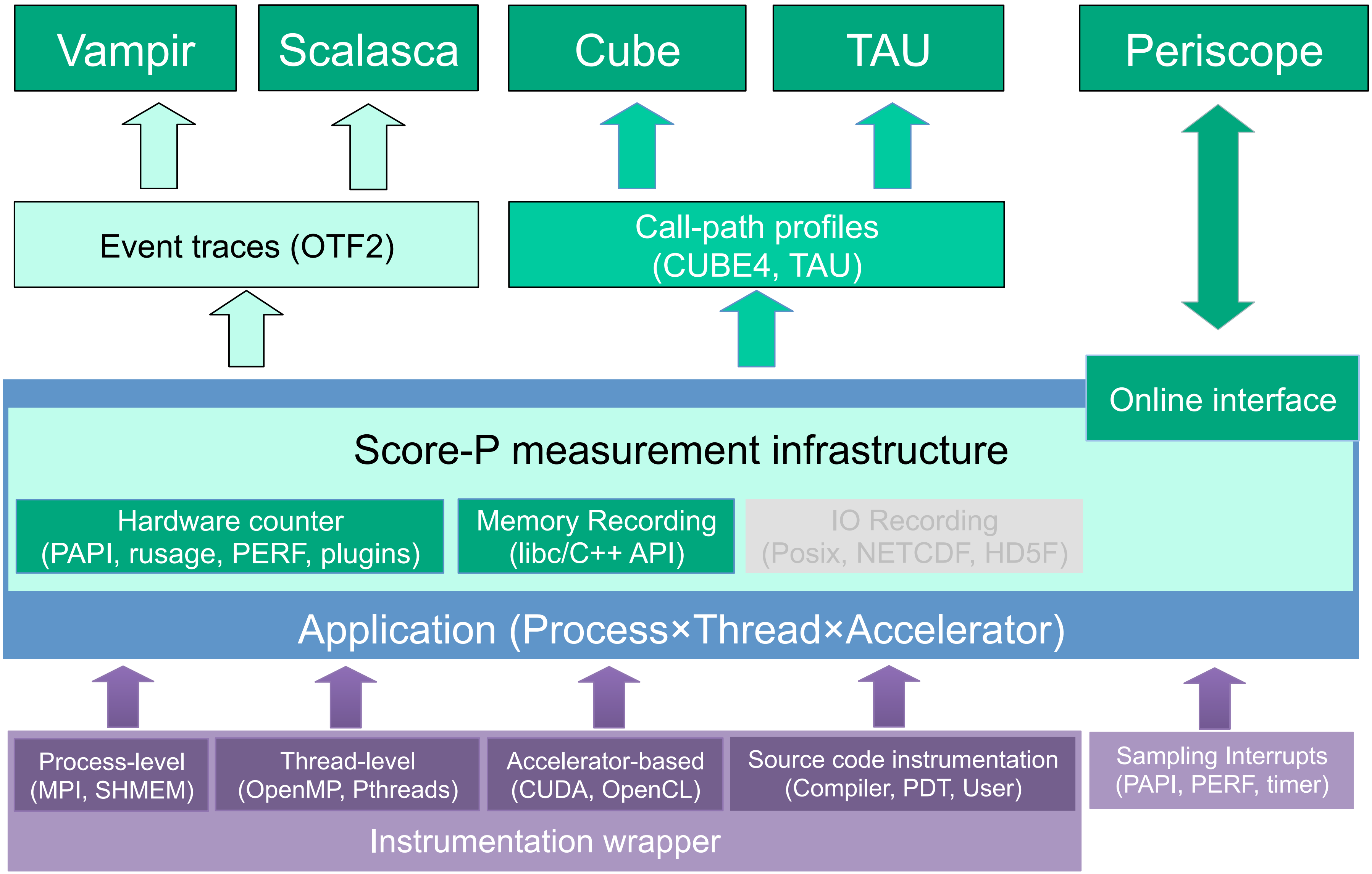
Vampir	Scalasca	TAU	Periscope
VampirTrace OTF	**EPILOG / CUBE**	**TAU native formats**	**Online measurement**

Score-P: Functionality

- Typical functionality for HPC performance tools
  - Instrumentation (various methods)
  - Sampling (experimental)
- Flexible measurement without re-compilation
  - Basic and advanced profile generation
  - Event trace recording
- Programming paradigms:
  - Multi-process
    - MPI, SHMEM
  - Thread-parallel
    - OpenMP, Pthreads
  - Accelerator-based
    - CUDA, OpenCL, OpenACC (Prototype)

Hybrid parallelism

Score-P: Architecture



Score-P: Workflow

0. **Perform a reference run and note the run time to be able to refer to it later**

1. **Instrument your application with Score-P**

CC = cc
CXX = CC
F90 = ftn

CC = scorep <options> cc
CXX = scorep <options> CC
F90 = scorep <options> ftn

- To see all available options for instrumentation:

```
$ scorep --help
This is the Score-P instrumentation tool. The usage is:
scorep <options> <original command>

Common options are:
...
  --nocompiler    Disables compiler instrumentation.
  --user          Enables user instrumentation.
  --cuda          Enables cuda instrumentation.
```

Score-P: Workflow

- For CMake and autotools based build systems it is recommended to use the scorep-wrapper script instances

Disable instrumentation

```
#CMake

SCOREP_WRAPPER=OFF cmake .. \
   -DCMAKE_C_COMPILER=scorep-cc \
   -DCMAKE_CXX_COMPILER=scorep-CC \
   -DCMAKE_Fortran_COMPILER=scorep-ftn
```

```
#Autotools

SCOREP_WRAPPER=OFF ../configure \
     CC=scorep-cc \
     CXX=scorep-CC \
     FC=scorep-ftn \
     --disable-dependency-tracking
```

- Pass instrumentation and compiler flags at make:

```
make SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--cuda" \
     SCOREP_WRAPPER_COMPILER_FLAGS="-g -O2"
```

scorep --cuda <your_compiler> –g –O2

Score-P: Workflow

2. Perform a measurement run with profiling enabled

- Example for generating a profile:

```
$ export SCOREP_ENABLE_PROFILING=true #default
$ export SCOREP_ENABLE_TRACING=false #default
$ export SCOREP_EXPERIMENT_DIRECTORY=profile

$ aprun <instrumented binary>
```

- To see all environment variables for the measurement:

```
$ scorep-info config-vars --full

SCOREP_ENABLE_PROFILING
    [...]
SCOREP_ENABLE_TRACING
    [...]
SCOREP_TOTAL_MEMORY
  Description: Total memory in bytes for the measurement system
    [...]
SCOREP_EXPERIMENT_DIRECTORY
  Description: Name of the experiment directory
    [...]
```

Score-P: Workflow

3. Compare profile runtime with reference runtime

- If overhead is too high:
  - Exclude short frequently called functions from measurement using hints from scorep-score

```
$ scorep-score -r profile/profile.cubex
  [...]
Flt type     max_buf[B]        visits time[s] time[%] time/visit[us]  region
  [...]
     USR  3,421,305,420   522,844,416  137.49    10.7           0.26  matvec_sub
     USR  3,421,305,420   522,844,416  174.16    13.5           0.33  matmul_sub
     USR  3,421,305,420   522,844,416  226.67    17.6           0.43  binvcrhs
     USR    150,937,332    22,692,096    6.73     0.5           0.30  binvrhs
     USR    150,937,332    22,692,096   14.69     1.1           0.65  lhsinit
     USR    112,194,160    17,219,840    4.70     0.4           0.27  exact_solution
     OMP      1,312,128       102,912    0.06     0.0           0.58  !$omp parallel
```

42% of the total time for these 3 regions , however, much of that is very likely measurement overhead due to short frequently called functions!

Score-P: Workflow

4. **Create an optimized profile with filter applied if measurement overhead of full instrumented profile is too high**

- Create a filter file and list functions to be excluded

```
$ vim scorep.filt
SCOREP_REGION_NAMES_BEGIN EXCLUDE
  matmul_sub
  matvec_sub
  binvcrhs
SCOREP_REGION_NAMES_END
```

- Example for generating a profile with filter applied:

```
$ export SCOREP_ENABLE_PROFILING=true
$ export SCOREP_ENABLE_TRACING=false
$ export SCOREP_FILTERING_FILE=scorep.filt
$ export SCOREP_EXPERIMENT_DIRECTORY=profile_with_filter

$ aprun <instrumented binary>
```

Score-P: Workflow

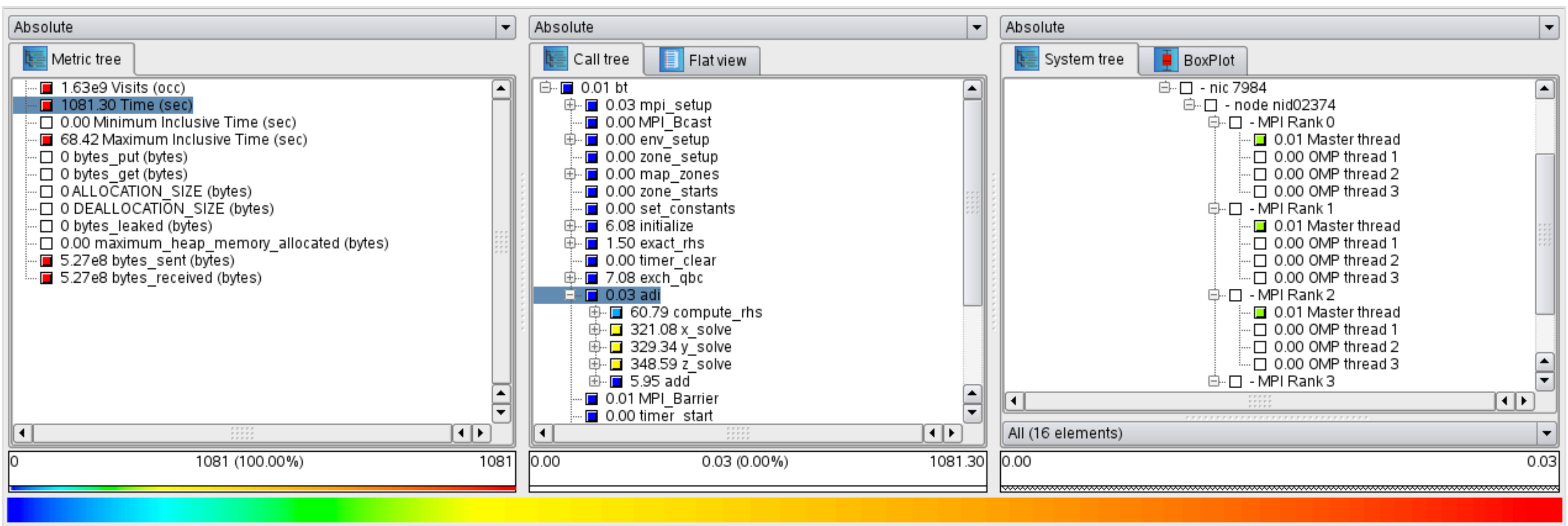
5. Perform analysis on (optimized) profile data

- Flat profile analysis with cube_stat:

```
$ cube_stat -t 3 -p profile_with_filter/profile.cubex
cube::Region              NumberOfCalls        ExclusiveTime  InclusiveTime
!$omp do @z_solve.f:52     51456.000000           131.579771     131.579771
!$omp do @y_solve.f:52     51456.000000           122.818761     122.818761
!$omp do @x_solve.f:54     51456.000000           117.027571     117.027571
```

- Call-path profile analysis with Cube:

```
$ cube profile_with_filter/profile.cubex
```



Score-P: Workflow

6. Define an appropriate filter for a tracing run

- Exclude functions from measurement which require a large trace buffer to reduce total trace size
- Use scorep-score with **full** instrumented profile

```
$ scorep-score -r profile/profile.cubex
Estimated aggregate size of event trace:                    40GB
Estimated requirements for largest trace buffer (max_buf): 10GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):        10GB
  [...]
Flt type      max_buf[B]        visits time[s] time[%] time/visit[us]  region
  [...]
     USR 3,421,305,420   522,844,416  137.49    10.7           0.26  matvec_sub
     USR 3,421,305,420   522,844,416  174.16    13.5           0.33  matmul_sub
     USR 3,421,305,420   522,844,416  226.67    17.6           0.43  binvcrhs
     USR   150,937,332    22,692,096    6.73     0.5           0.30  binvrhs
     USR   150,937,332    22,692,096   14.69     1.1           0.65  lhsinit
     USR   112,194,160    17,219,840    4.70     0.4           0.27  exact_solution
     OMP     1,312,128       102,912    0.06     0.0           0.58  !$omp parallel
```

About 10 GB just for these 6 regions per process!

- Test the effect of your filter on the trace file

```
$ scorep-score -f scorep.filt profile/profile.cubex
```

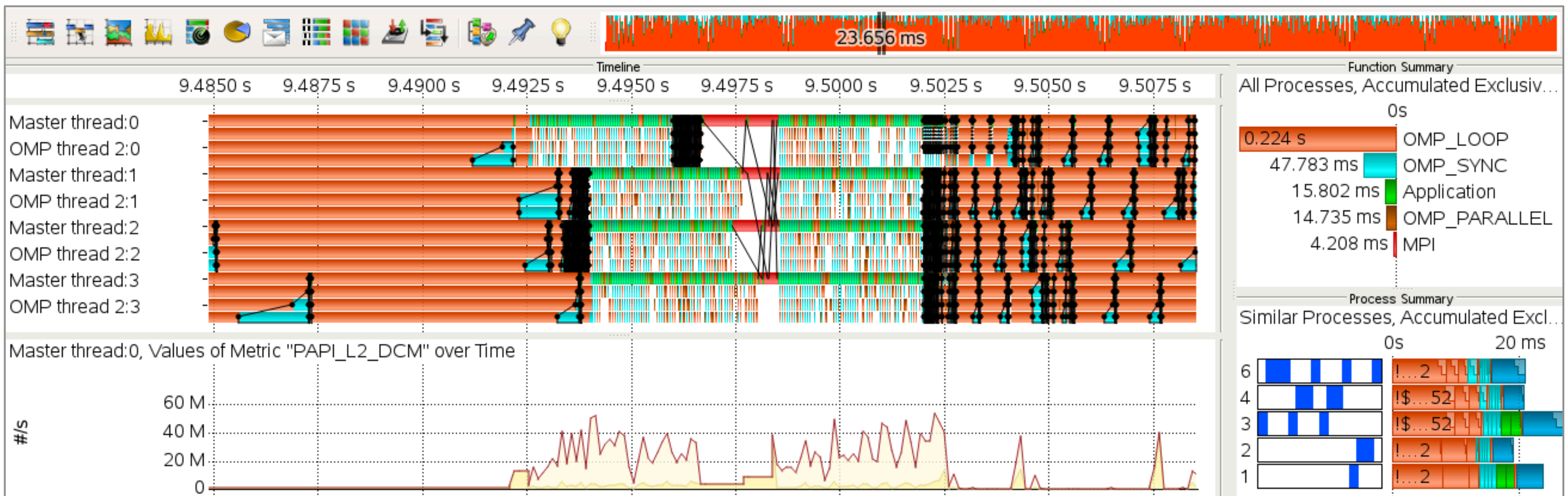
Score-P: Workflow

5. Perform a measurement run with tracing enabled and the filter applied

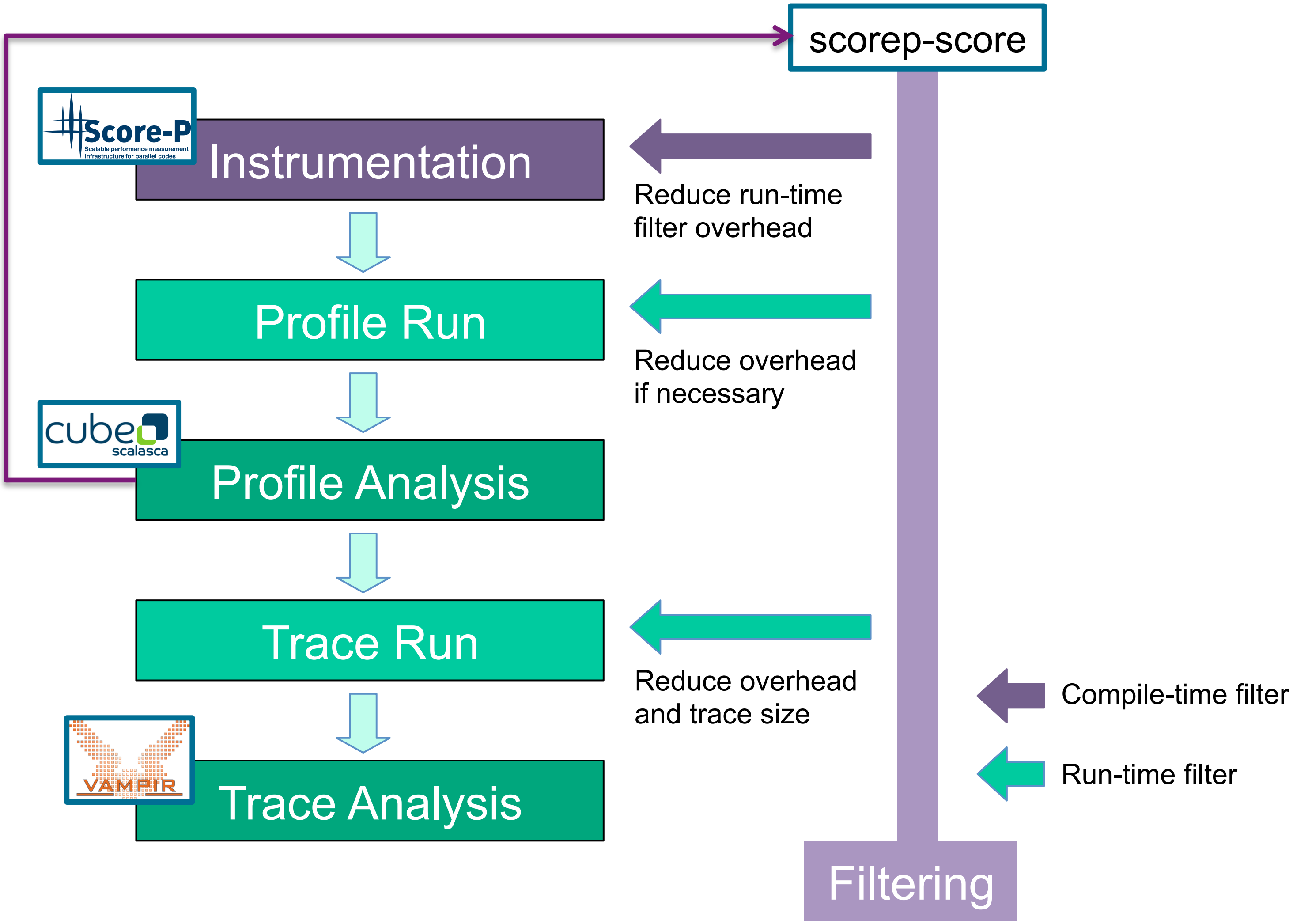
```
$ export SCOREP_ENABLE_PROFILING=false
$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_EXPERIMENT_DIRECTORY=trace
$ export SCOREP_FILTERING_FILE=scorep.filt
$ aprun <instrumented binary>
```

6. Perform analysis on the trace data with Vampir

```
$ vampir trace/traces.otf2
```



Score-P: Workflow Summary



Score-P Advanced Features: Sampling

- Alternative to compiler instrumentation to generate profiles or traces
- Regulate the trade-off between overhead and correctness
- Libunwind/1.1 to capture current stack
- Sampling interrupt sources:
  - Interval timer, PAPI, Perf
- Example for enabling sampling for measurement run:

```
$ export SCOREP_ENABLE_UNWINDING=true
$ export SCOREP_SAMPLING_EVENTS=PAPI_TOT_CYC@1000000
```

- Combination of instrumented and sampled events (not for compiler instrumented events)
- Calling context information for every event

Score-P Advanced Features: Memory Rec.

- Memory (de)allocations are recorded via the libc/C++ API
- Recording of memory location’s call-site in sampling mode
  – Debugging symbols required (-g)
- Interplay of memory usage and application’s execution
  – CUBE: (De)allocation size, maximum heap memory, leaked bytes
  – Vampir: Memory usage in “Counter Timelines”
- Enabling memory recording for measurement run:

```
$ export SCOREP_MEMORY_RECORDING=true
```

Agenda

Performance Analysis Approaches

- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes

- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools

- Cube
- Vampir

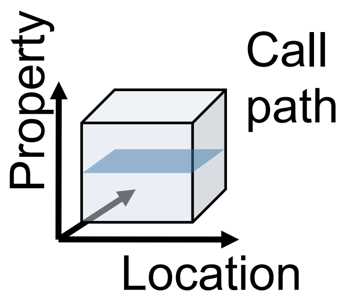
Demo

- Performance Analysis of Jacobi Solver on Titan

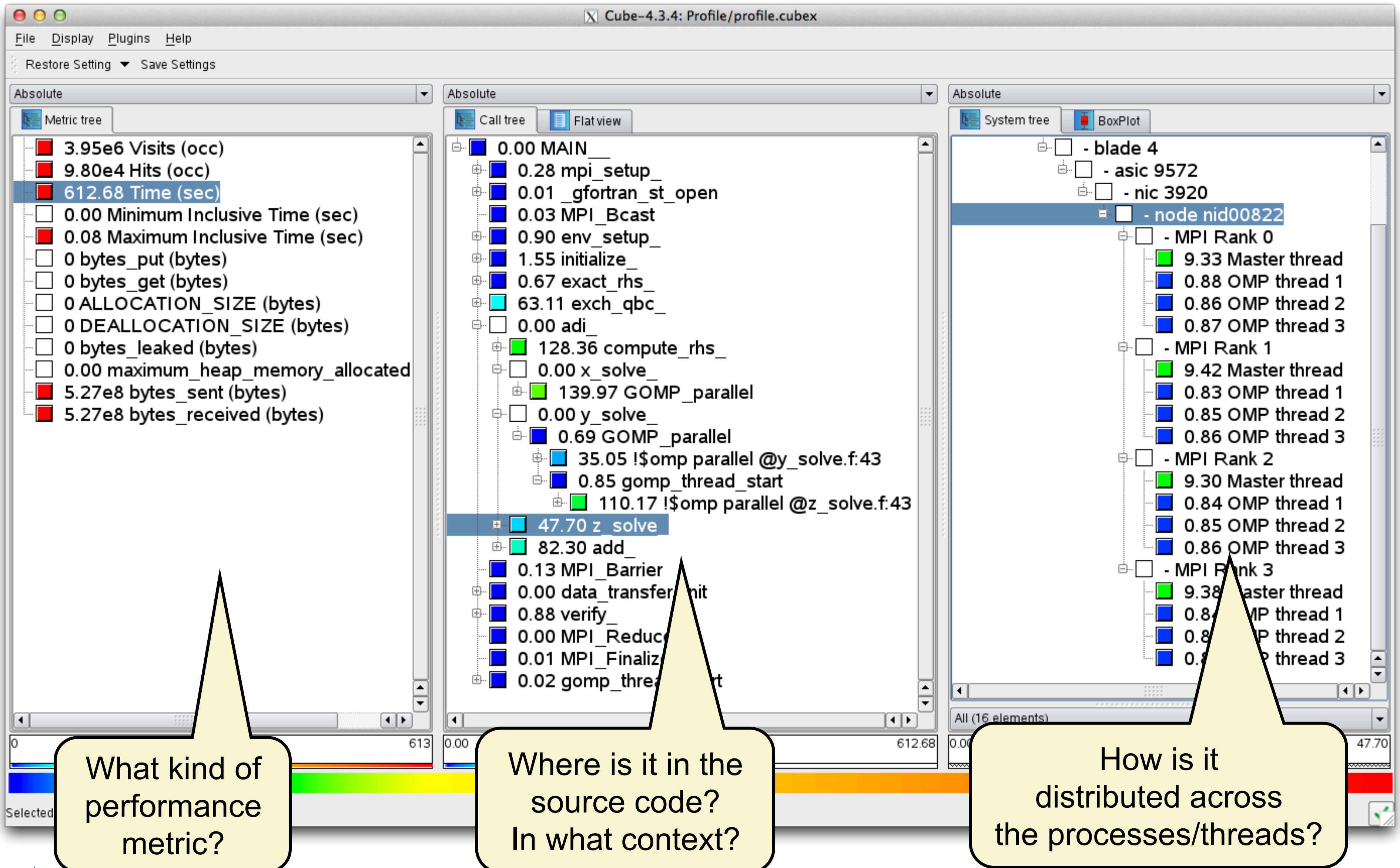
Conclusions

Cube

- Profile analysis tool for displaying performance data of parallel programs
- Originally developed as part of Scalasca toolset
- Available as a separate component of Score-P
- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call-tree path (program location)
  - System location (process/thread)
- Three coupled tree browsers



Cube: Analysis Presentation

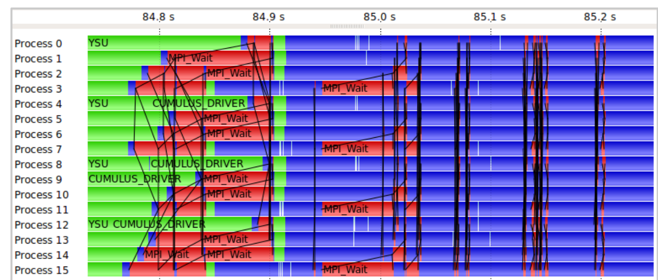


Vampir

- Event trace analysis tool for displaying performance data of complex parallel programs
- Show dynamic run-time behavior graphically at a fine level of detail
- Provide summaries (profiles) on performance metrics

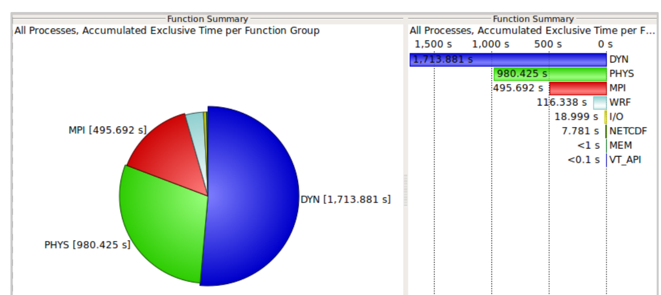
Timeline charts

- Show application activities and communication along a time axis



Summary charts

- Provide quantitative results for the currently selected time interval



Vampir: Performance Charts Overview

Timeline Charts

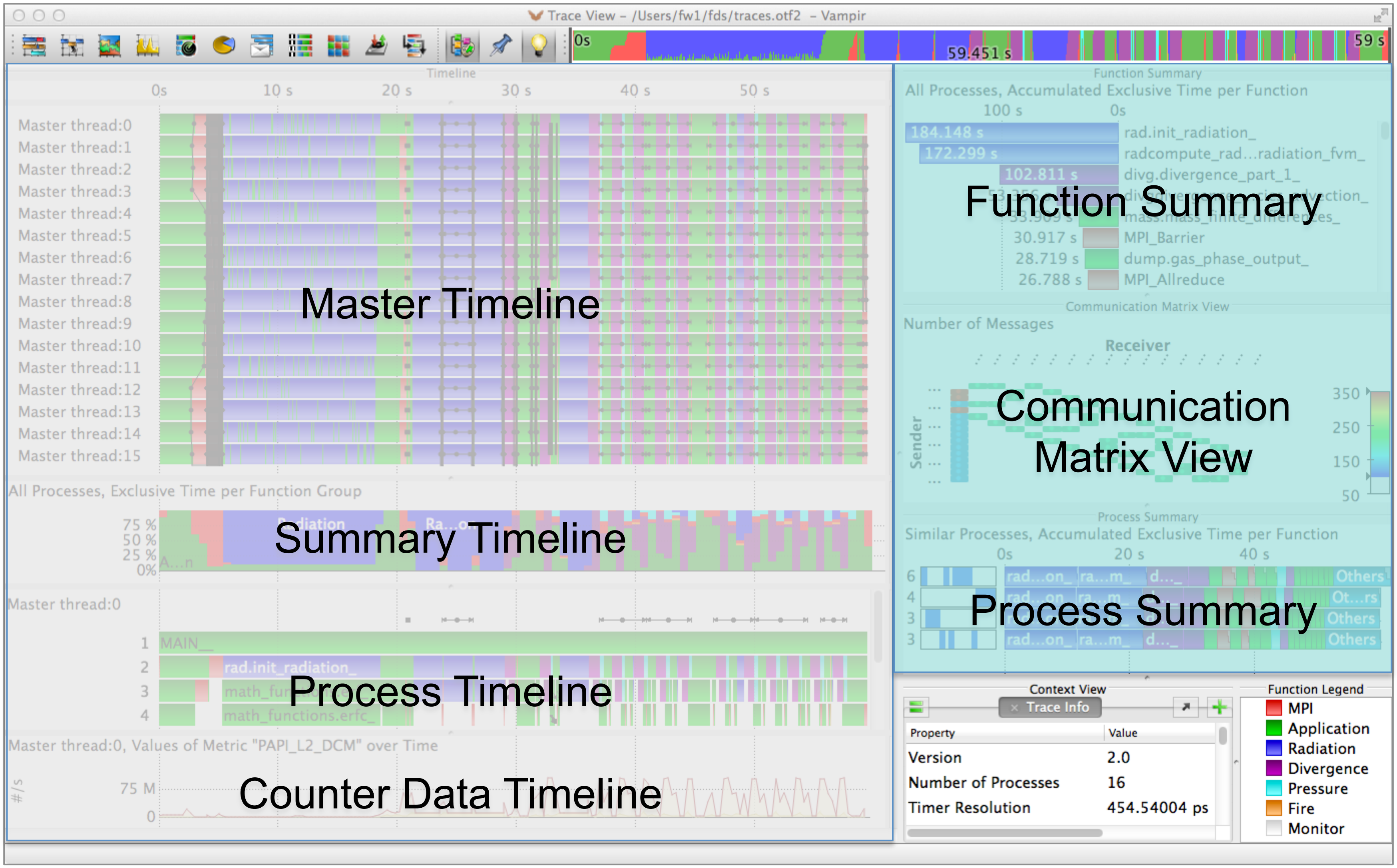
Master Timeline	➡	*all threads' activities over time per thread*
Summary Timeline	➡	*all threads activities over time per activity*
Performance Radar	➡	*all threads' perf-metric over time*
Process Timeline	➡	*single thread's activities over time*
Counter Data Timeline	➡	*single threads perf-metric over time*

Summary Charts

Function Summary	Process Summary
Message Summary	Communication Matrix View

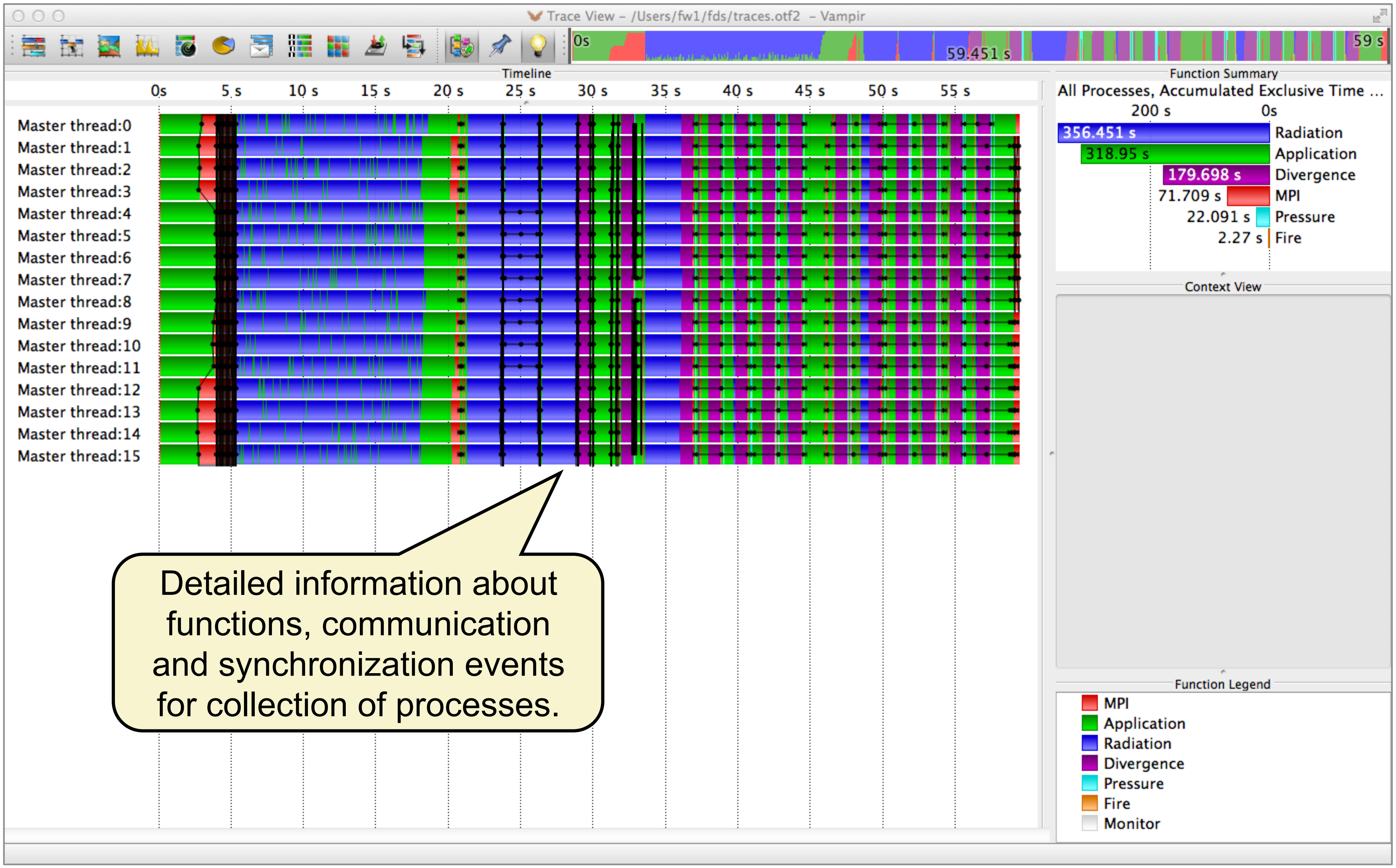
Vampir: Performance Charts

- Trace visualization of FDS (Fire Dynamics Simulator)



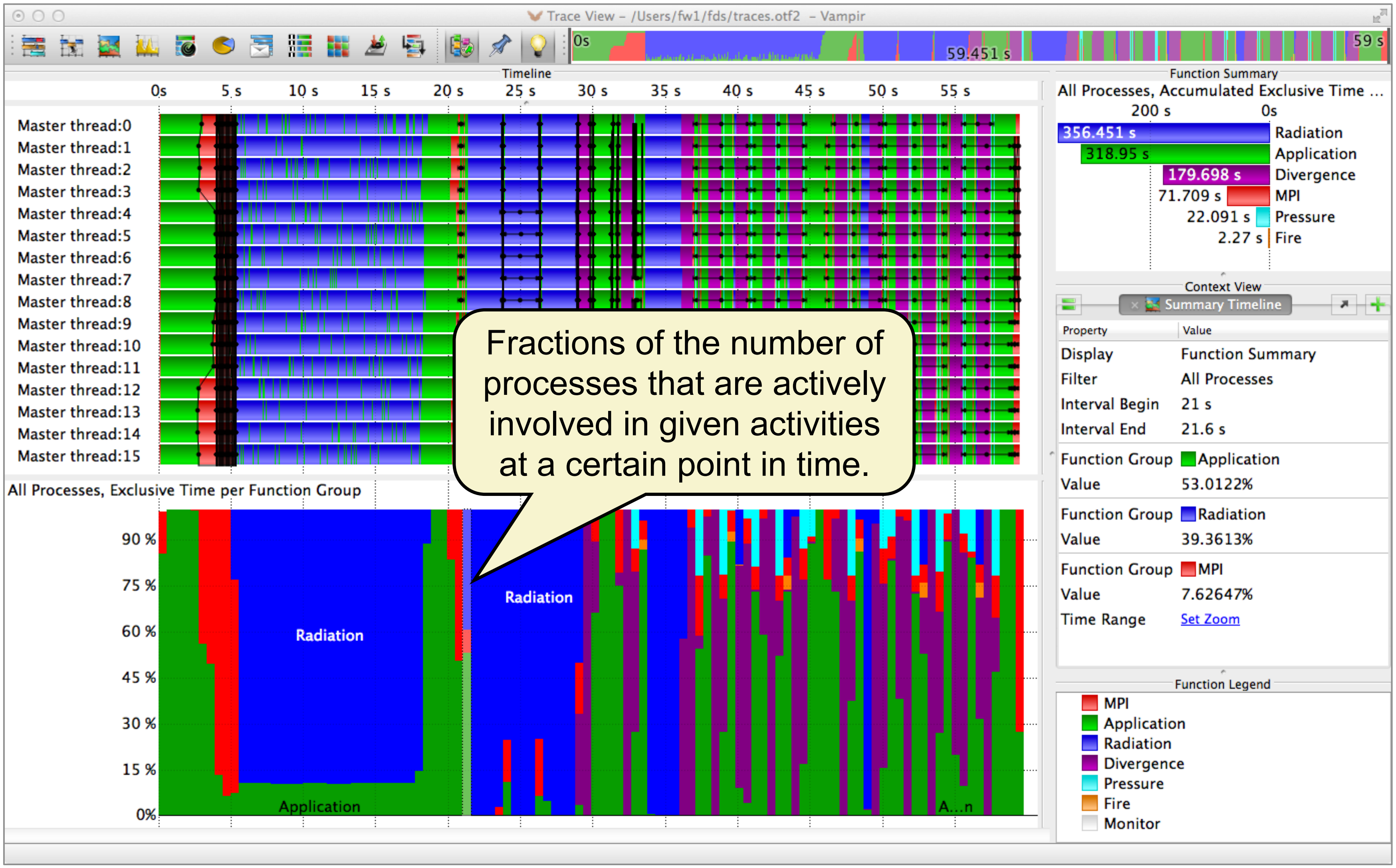
Vampir: Performance Charts

Master Timeline



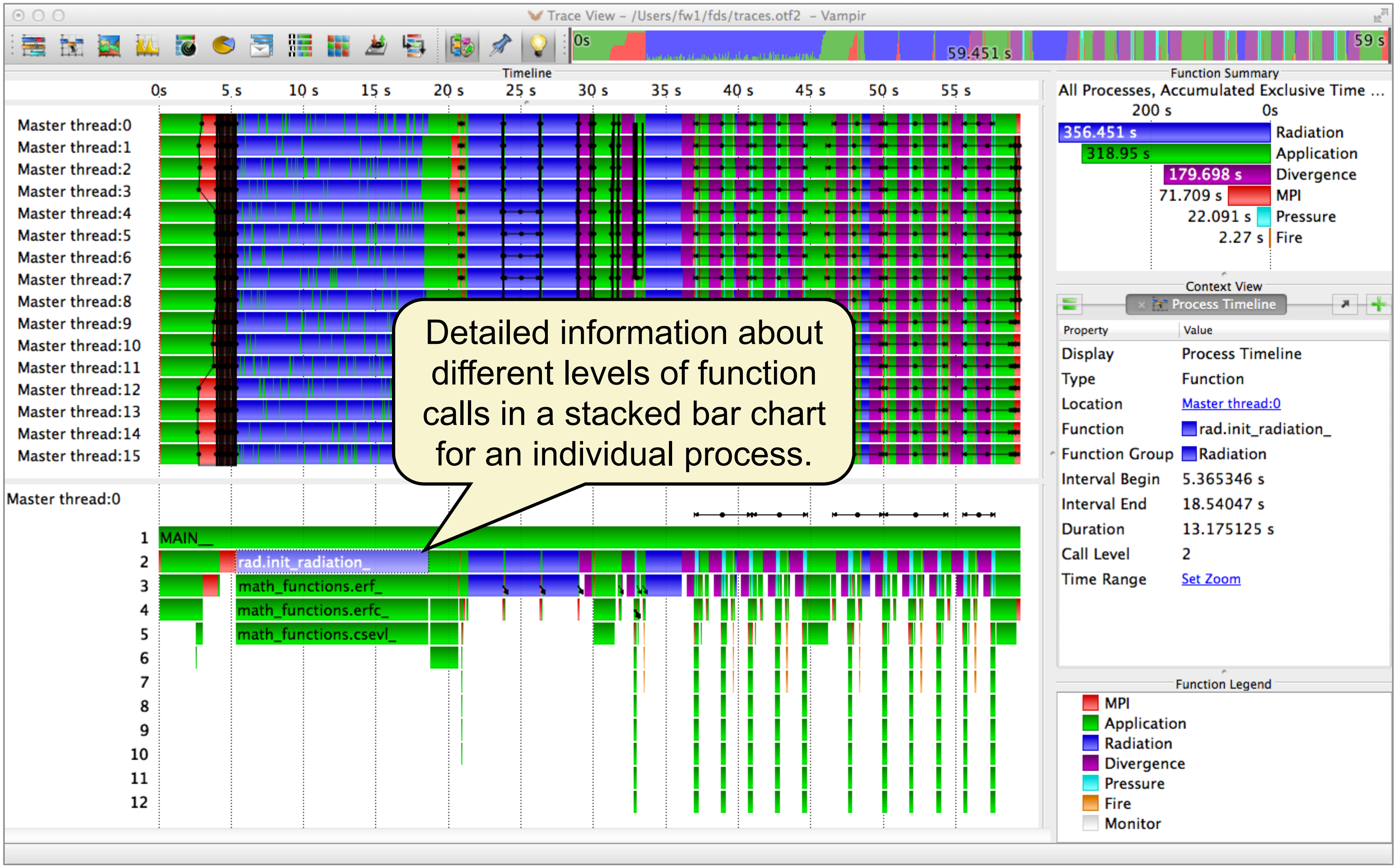
Vampir: Performance Charts

Summary Timeline



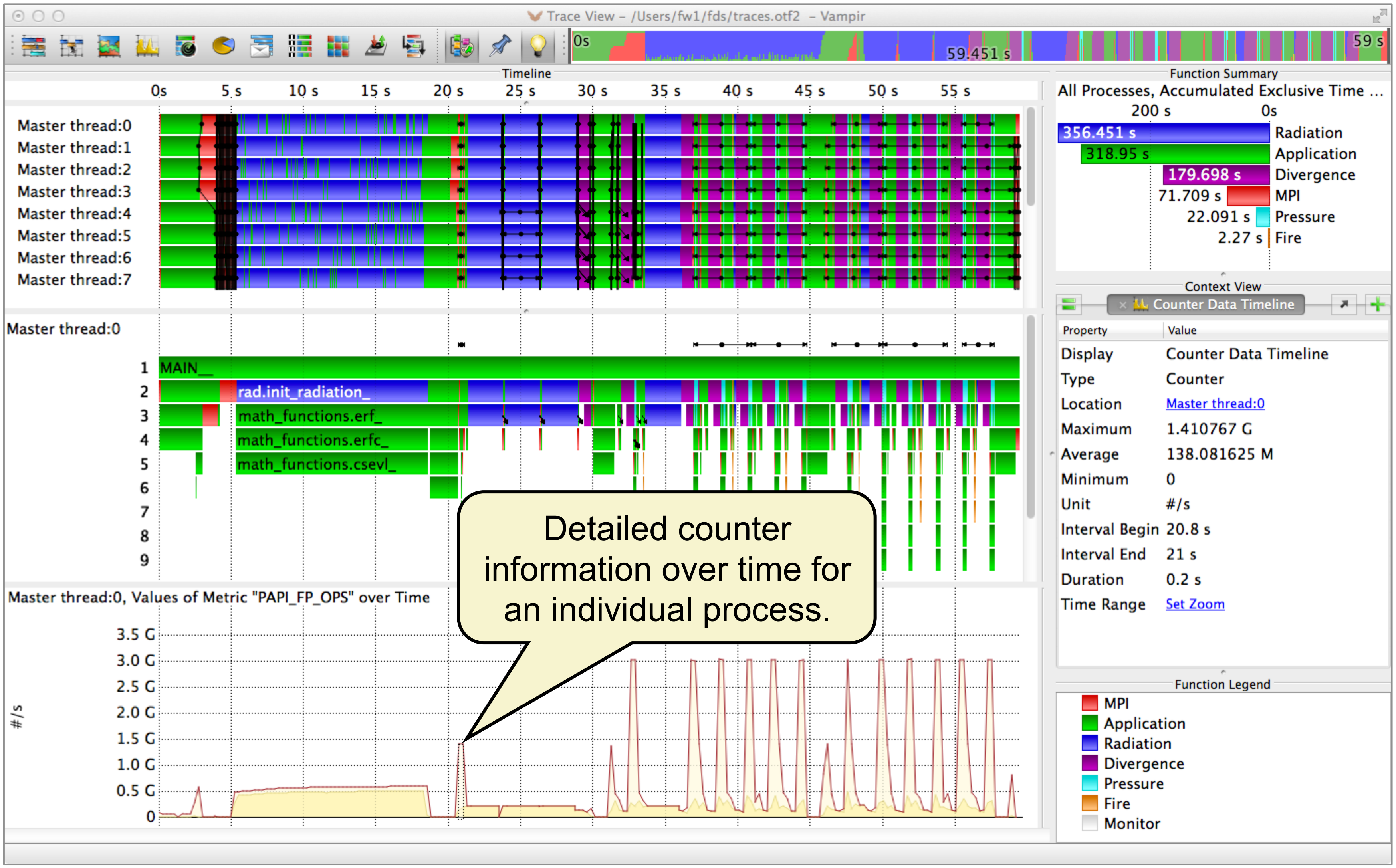
Vampir: Performance Charts

Process Timeline



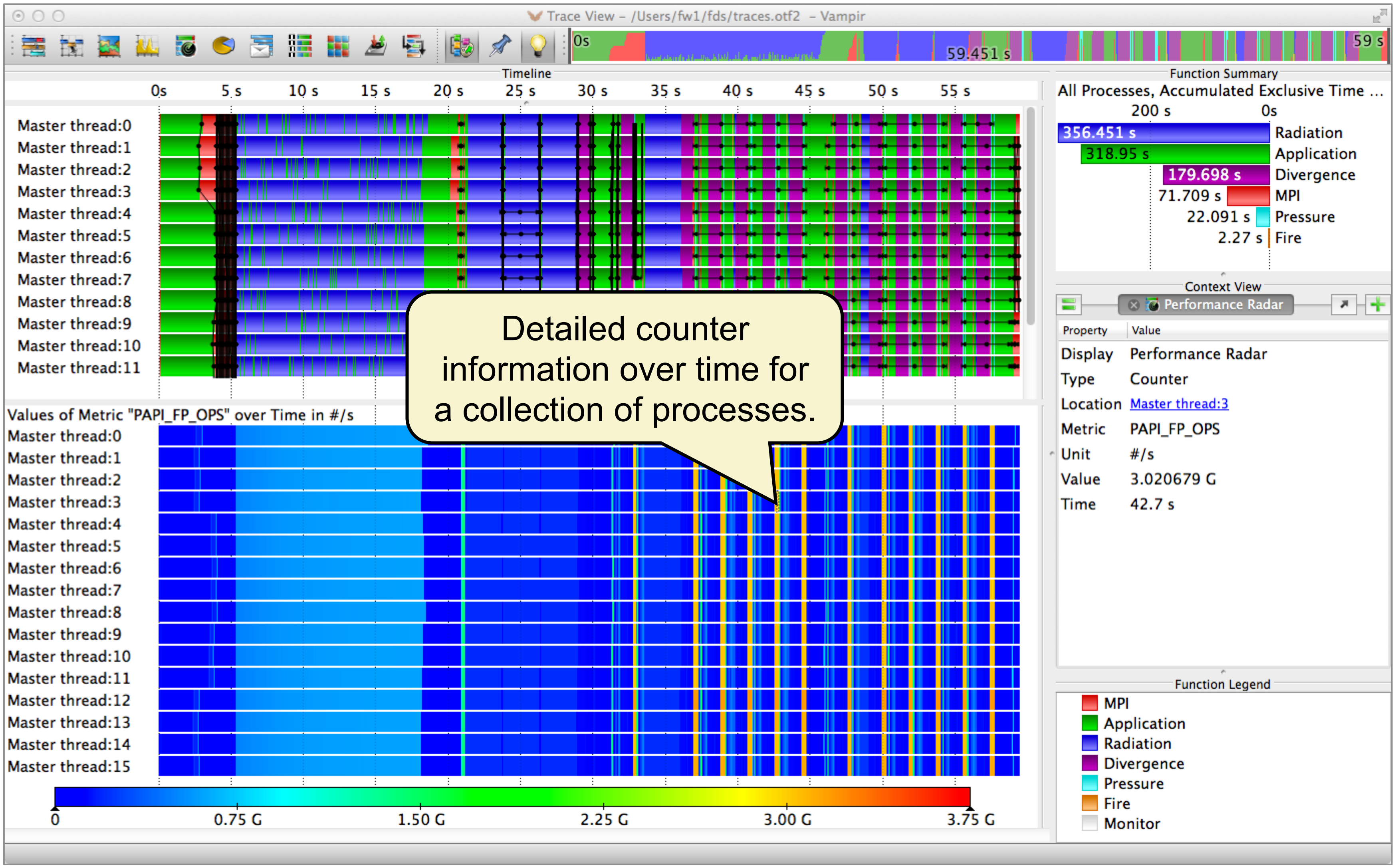
Vampir: Performance Charts

Counter Timeline



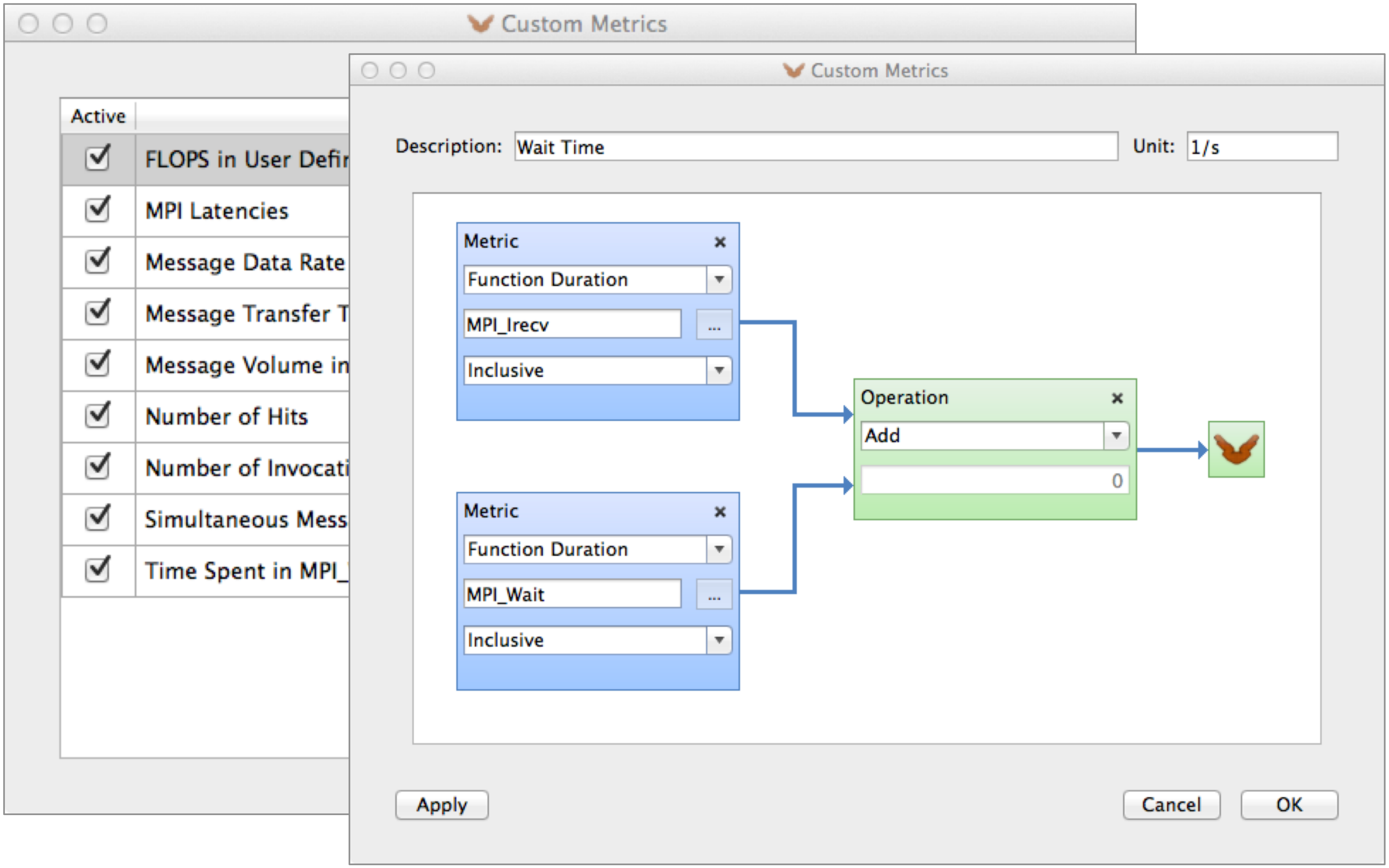
Vampir: Performance Charts

Performance Radar



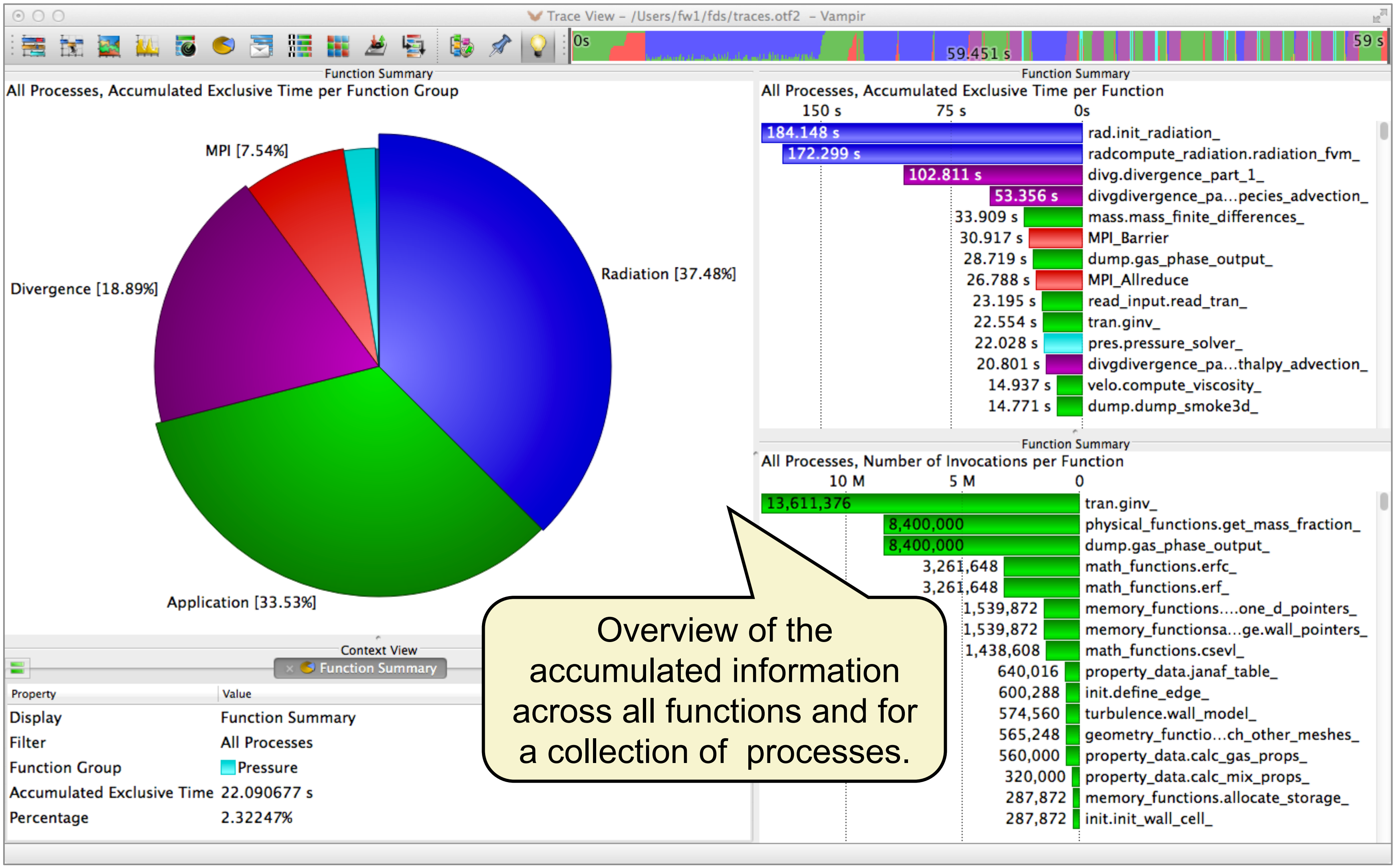
Vampir: Where Do the Metrics Come From?

- Custom Metrics Built-In Editor



Vampir: Performance Charts

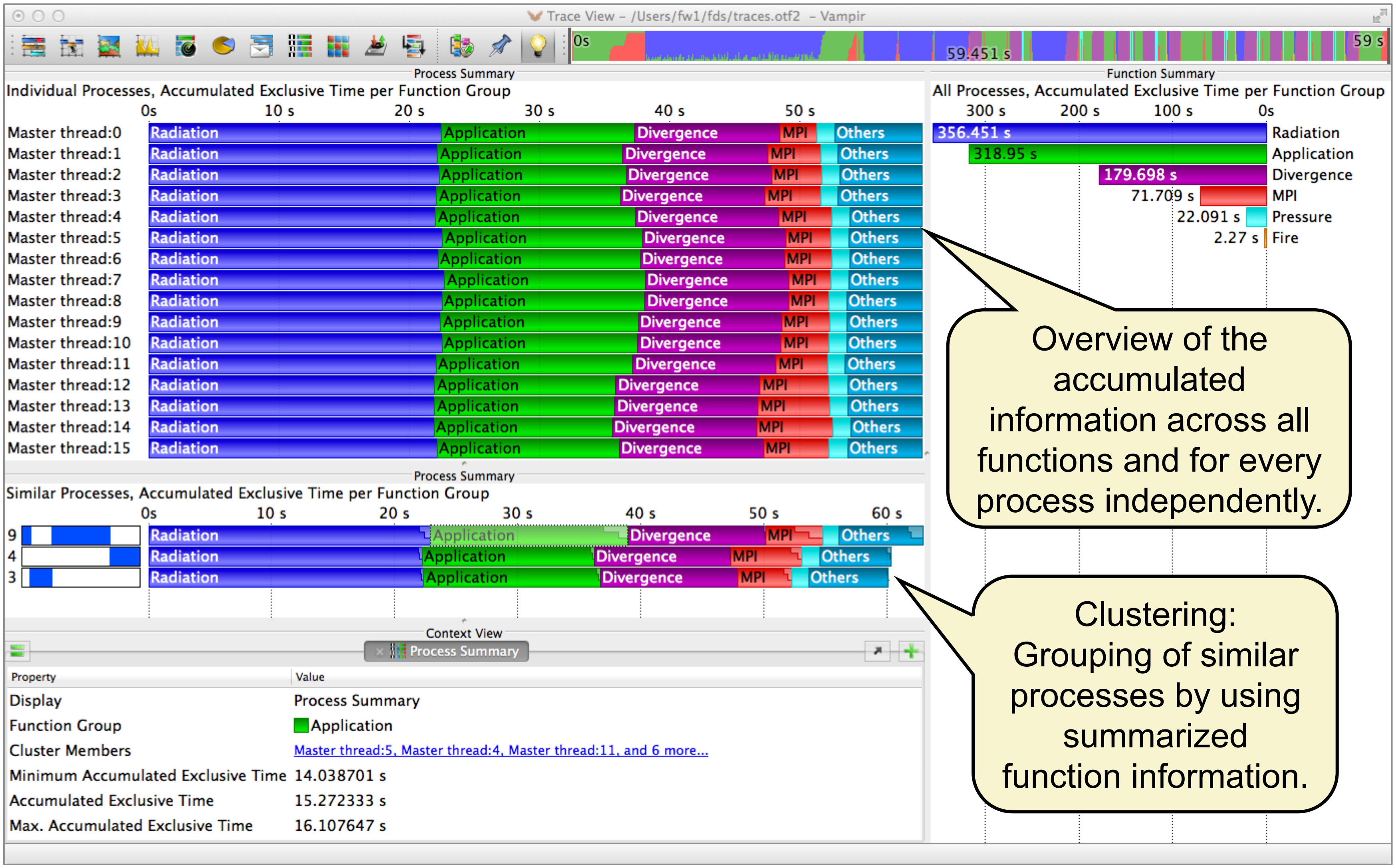
Function Summary



Overview of the accumulated information across all functions and for a collection of processes.

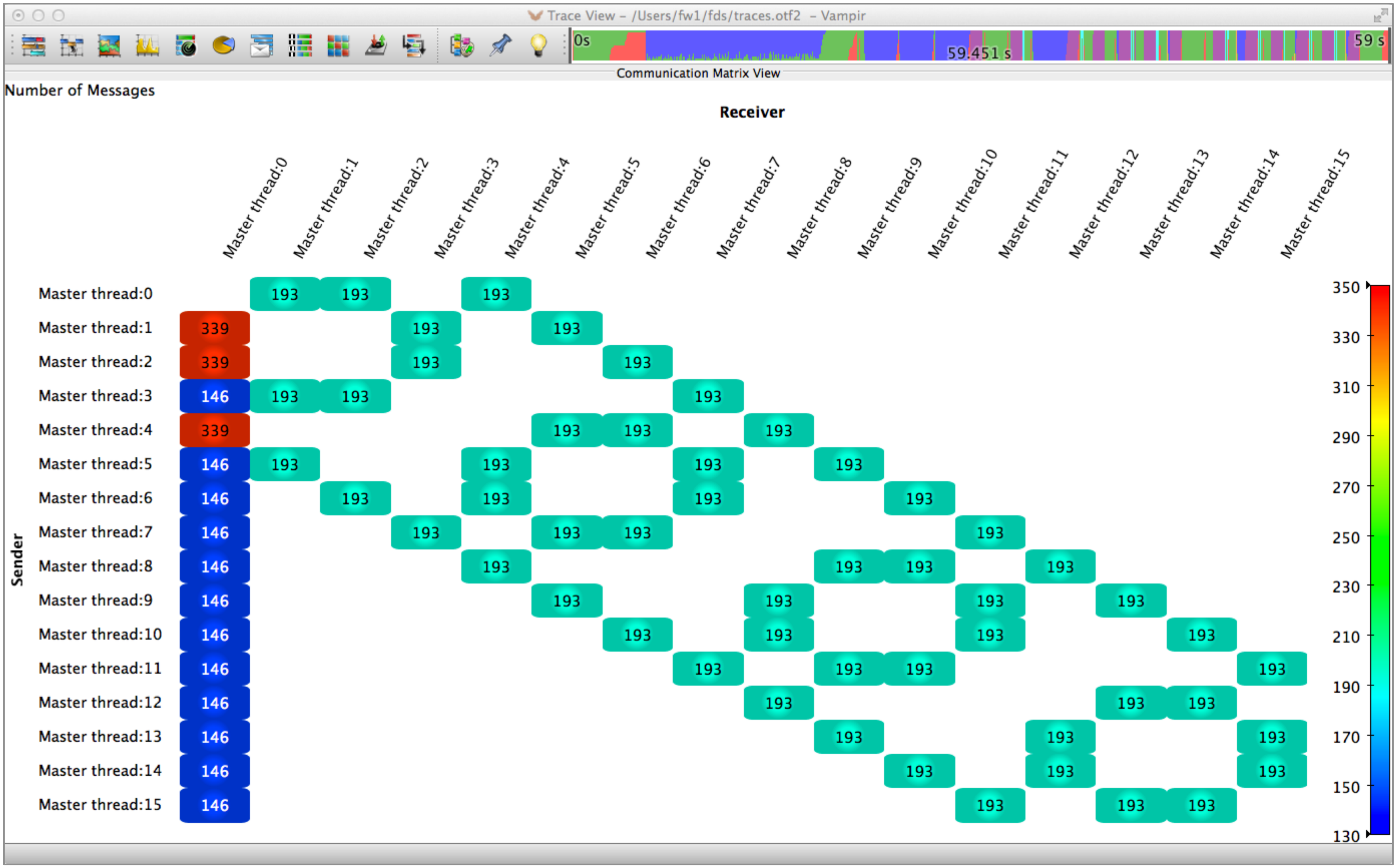
Vampir: Performance Charts

Process Summary



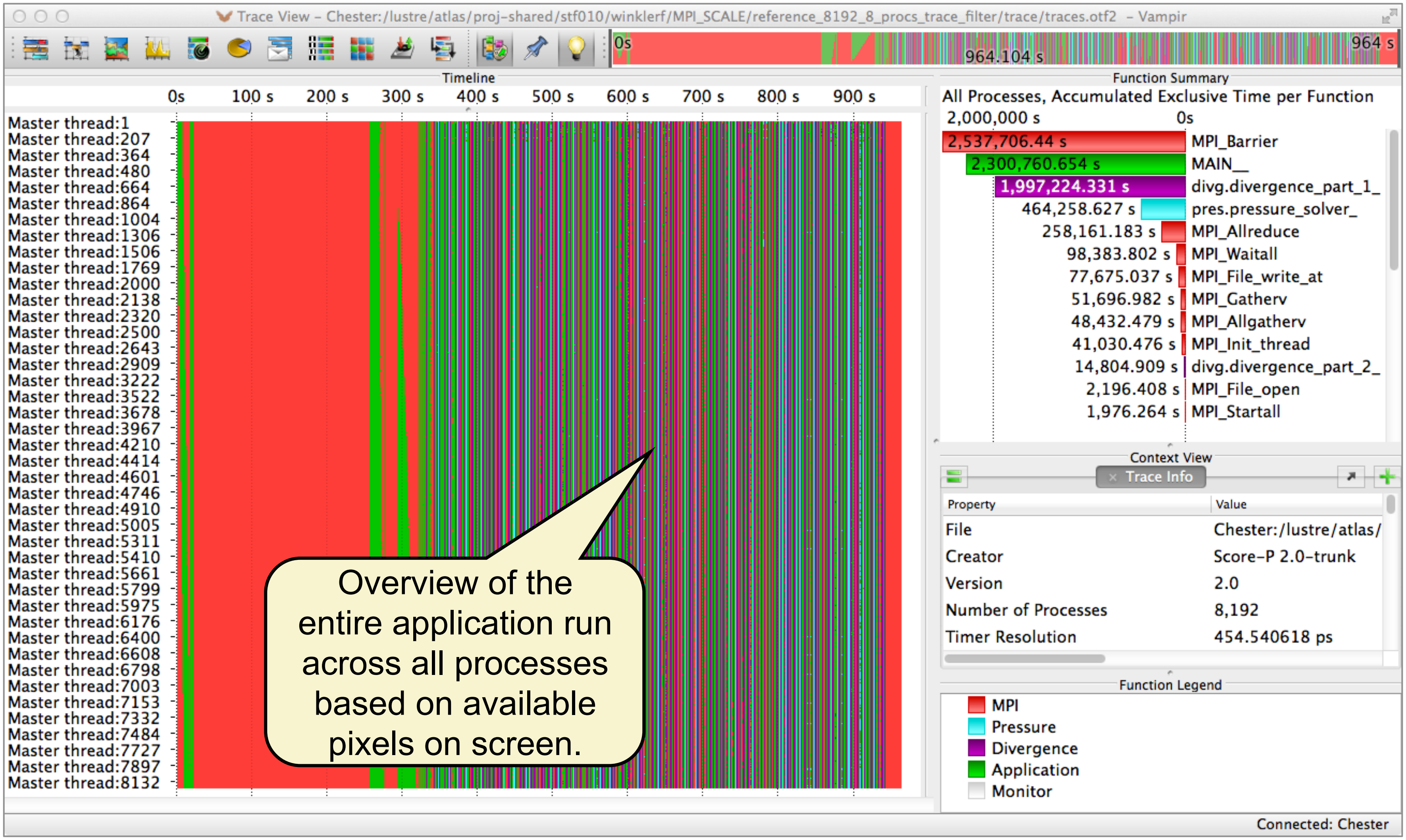
Vampir: Performance Charts

Communication Matrix View



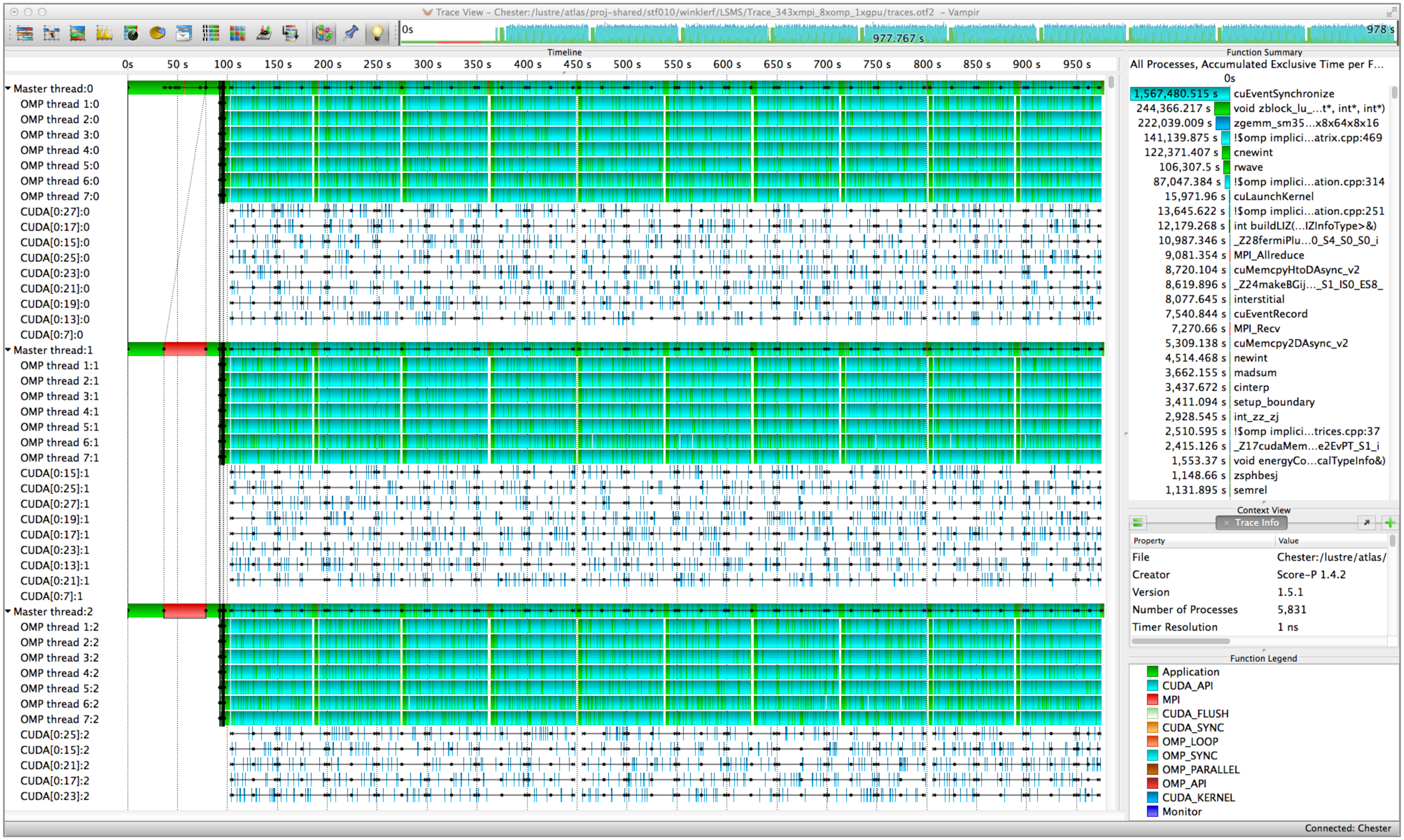
Vampir at Scale: FDS with 8192 cores

- Fit to chart height feature in Master Timeline



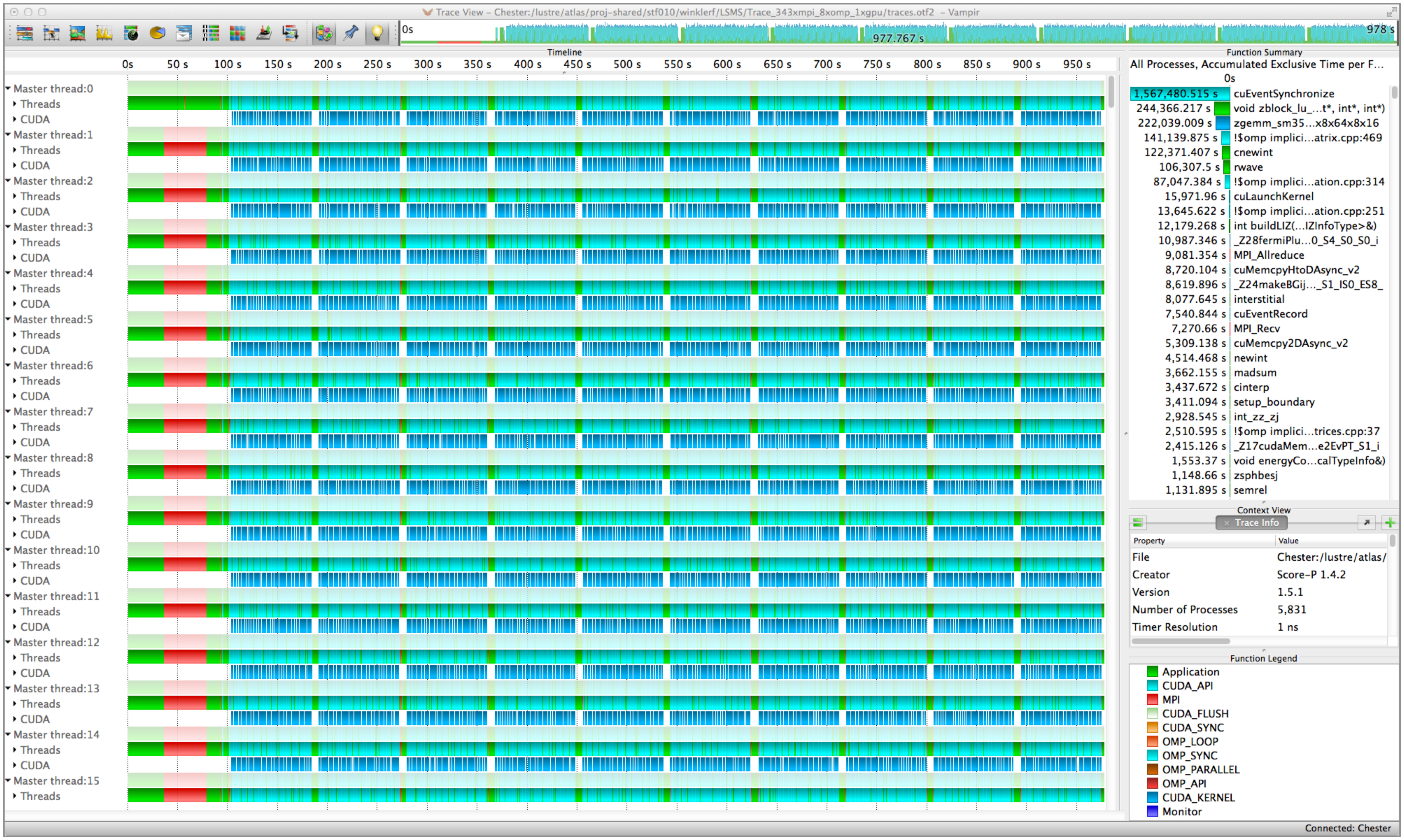
Vampir at Scale: LSMS (hybrid parallelism)

- 5831 processes: 343xMPI with 8xOpenMP and 8xCUDA



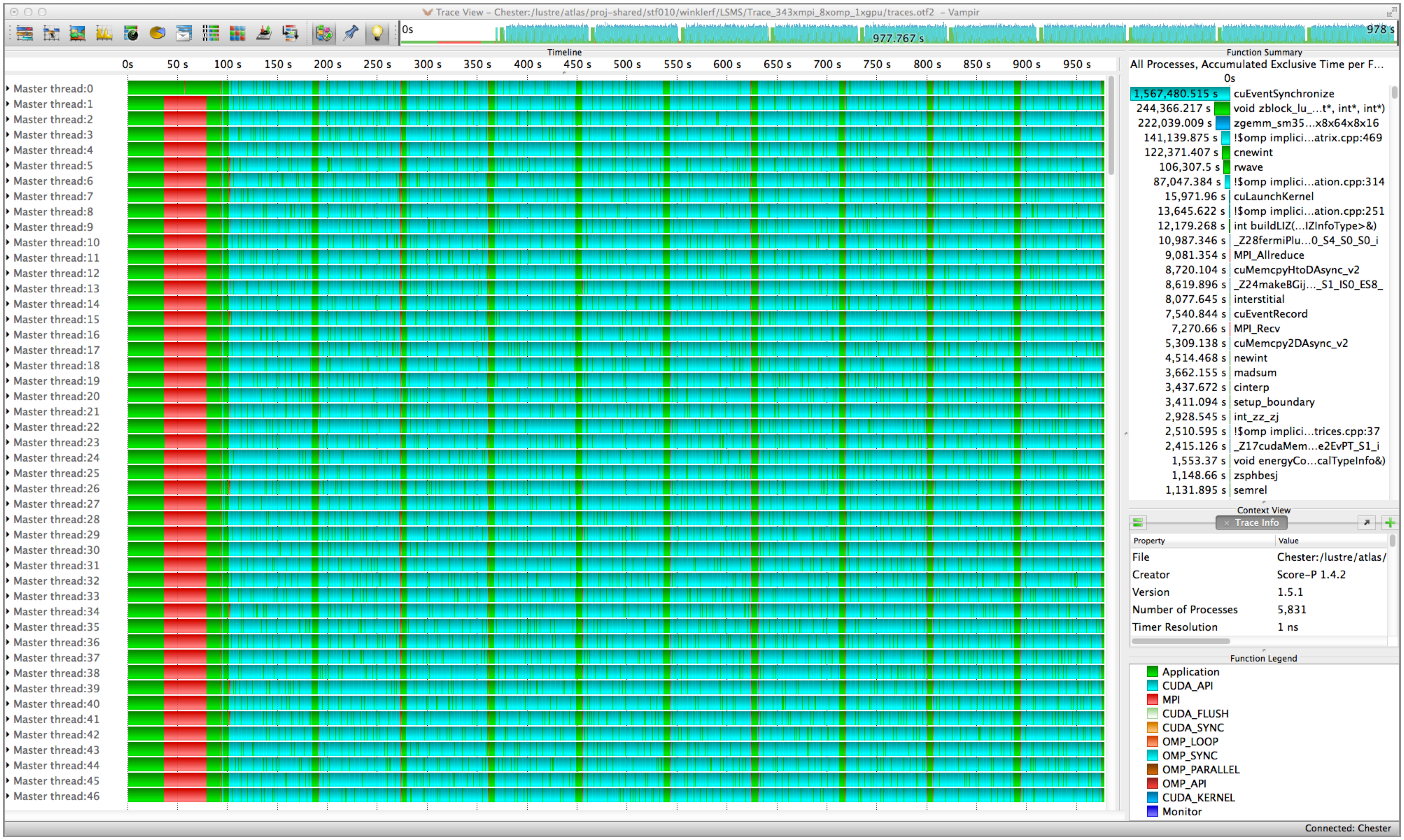
Vampir at Scale: LSMS (hybrid parallelism)

- Group threads and CUDA streams



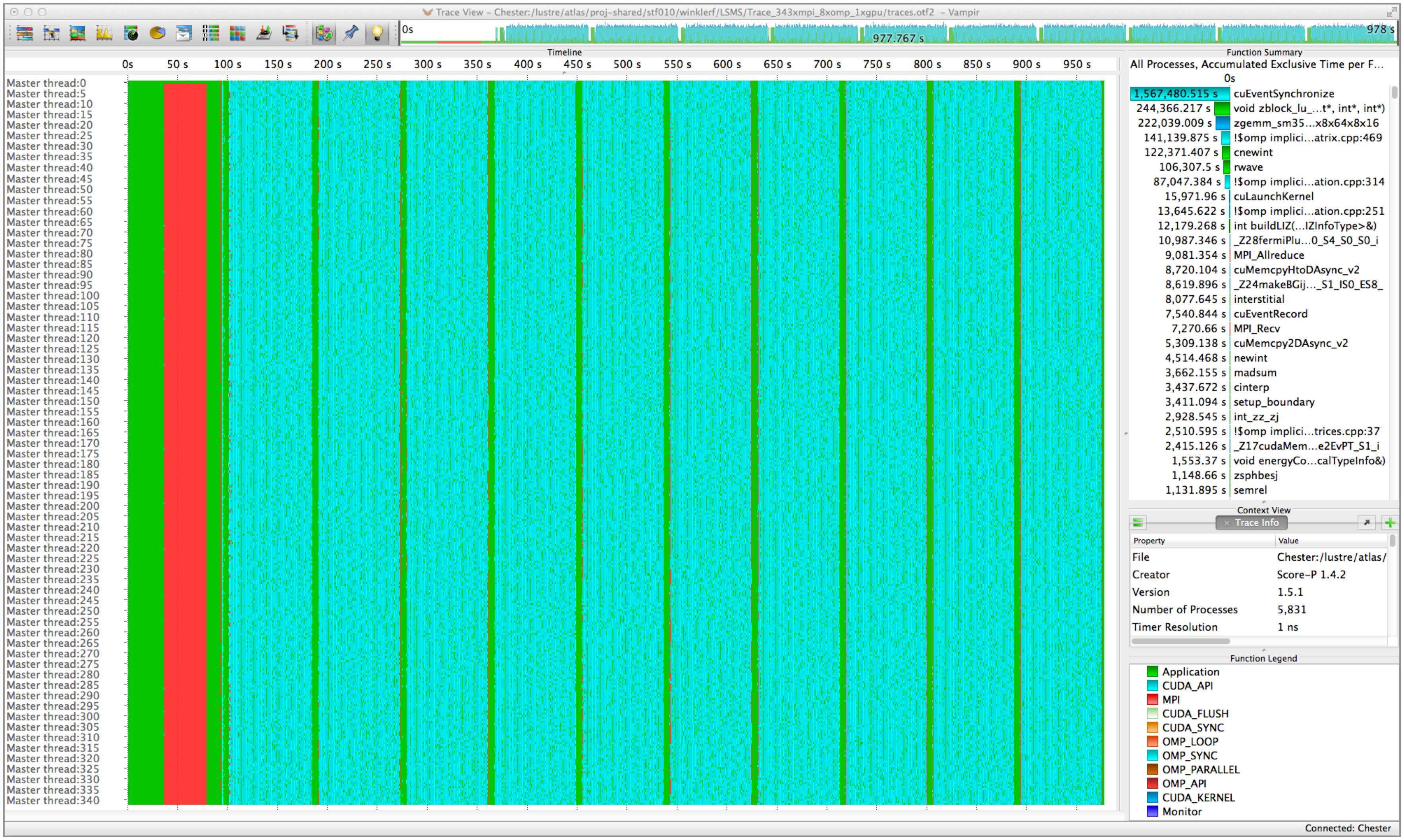
Vampir at Scale: LSMS (hybrid parallelism)

- Collapse all MPI processes



Vampir at Scale: LSMS (hybrid parallelism)

- Fit to chart height for all collapsed MPI processes



Agenda

Performance Analysis Approaches

- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes

- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools

- Cube
- Vampir

Demo

- Performance Analysis of Jacobi Solver on Titan

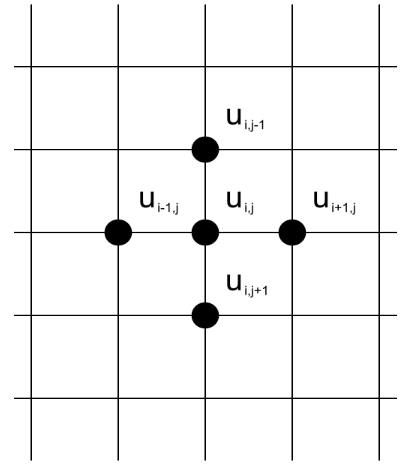
Conclusions

Demo: Jacobi Solver

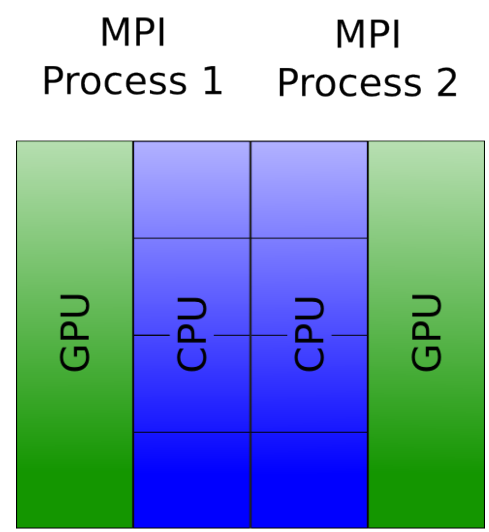
- Jacobi Example
  - Iterative solver for system of equations

$$U_{old} = U$$
$$u_{i,j} = bu_{old,i,j} + a_x(u_{old,i-1,j} + u_{old,i+1,j}) + a_y(u_{old,i,j-1} + u_{old,i,j+1}) - rHs / b$$

  - Code uses OpenMP, CUDA and MPI for parallelization



- Domain decomposition
  - Halo exchange at boundaries:
    - Via MPI between processes
    - Via CUDA between hosts and accelerators



Demo: Jacobi Solver / Setup

- Connect to Titan via X forwarding and copy sources

```
$ cd $MEMBERWORK/[projid]
$ cp /sw/sources/vampir/tutorial/jacobi.tar.gz .
$ tar xzvf jacobi.tar.gz
$ cd jacobi
```

- Change programming environment and load modules

```
$ module swap PrgEnv-{pgi,gnu}
$ module load cudatoolkit
$ module load scorep
```

- Compile benchmark and submit job

```
$ make
$ qsub -A [projid] run.pbs
$ less jacobi.o[JOB_ID]
Jacobi relaxation Calculation: 8192 x 8192 mesh with
 2 processes and 16 threads + one Tesla K20X for each process.
 614 of 2049 local rows are calculated on the CPU to balance the load
 between the CPU and the GPU.
 0, 0.489197
  100, 0.002397
   [...]
 total: 8.425432 s
```

Keep time in mind!

Demo: Jacobi Solver / Profiling

- Build instrumented executable

```
$ make clean
$ make scorep
scorep --cuda cc … -o bin/jacobi_mpi+openmp+cuda
```

- Submit job for profiling run

```
$ less run_profile.pbs
[...]
export SCOREP_ENABLE_PROFILING=true
export SCOREP_ENABLE_TRACING=false
export SCOREP_EXPERIMENT_DIRECTORY=jacobi_mpi+openmp+cuda_profile
export SCOREP_CUDA_ENABLE=yes
export SCOREP_TIMER=clock_gettime
export SCOREP_MEMORY_RECORDING=yes
[...]
aprun -n 2 -d 16 -N 1 ./jacobi_mpi+openmp+cuda 8192 8192 0.15

$ qsub -A [projid] run_profile.pbs
$ less jacobi.o[JOB_ID]
Jacobi relaxation Calculation: 8192 x 8192 mesh with
 2 processes and 16 threads + one Tesla K20X for each process.
 [...]
 total: 9.858350 s
```

15% Overhead!

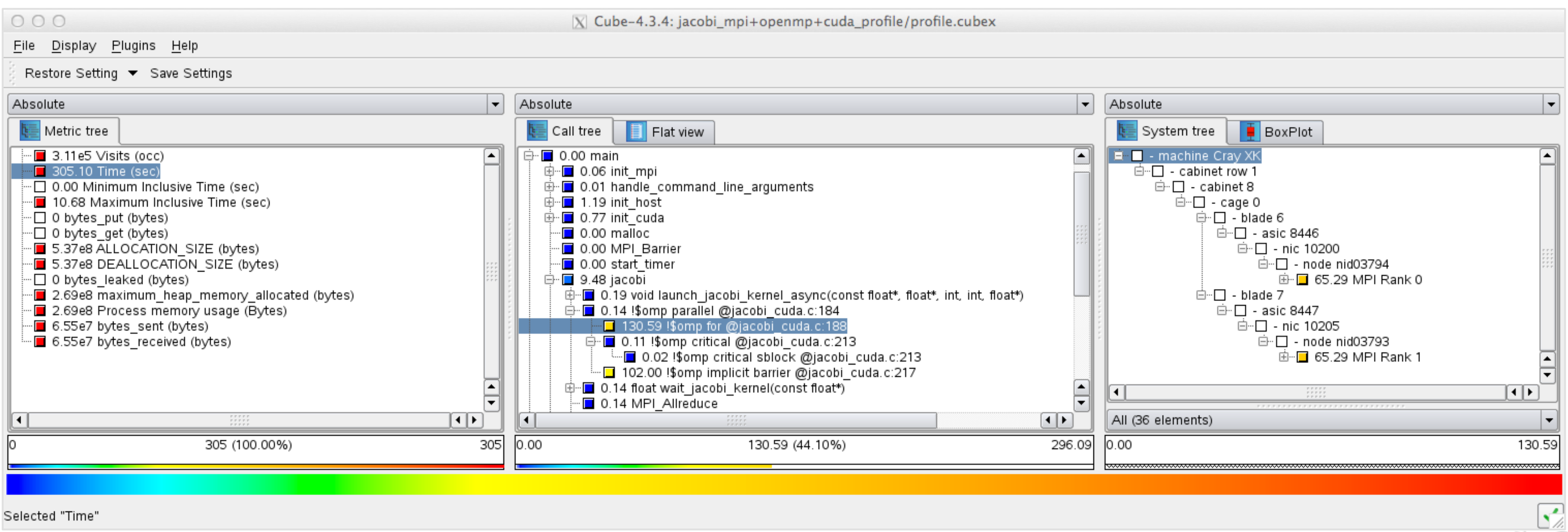
Demo: Jacobi Solver / Profile Analysis

- Perform flat profile analysis with cube_stat

```
$ cd bin.scorep
$ cube_stat -t 10 -p jacobi_mpi+openmp+cuda_profile/profile.cubex
cube::Region                    NumberOfCalls ExclusiveTime InclusiveTime
!$omp for @jacobi_cuda.c:188    32000.000000    131.797289    131.797289
!$omp implicit barrier          32000.000000    104.298683    104.298683
!$omp for @jacobi_cuda.c:258    32000.000000     42.999056     50.568642
[...]
```

- Perform call-path profile analysis with Cube

```
$ cube jacobi_mpi+openmp+cuda_profile/profile.cubex
```



Demo: Jacobi Solver / Scoring

- Do we need a filter? (Overhead and memory footprint)

```
$ scorep-score jacobi_mpi+openmp+cuda_profile/profile.cubex
Estimated aggregate size of event trace:                    10MB
Estimated requirements for largest trace buffer (max_buf): 5MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):        41MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=41MB 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  4,924,060 310,504  308.53   100.0         993.63  ALL
         OMP  4,135,850 256,417  287.31    93.1        1120.46  OMP
        CUDA    494,338  38,025   10.40     3.4         273.53  CUDA
         COM    156,260  12,020   10.46     3.4         870.58  COM
         MPI    137,222   4,012    0.30     0.1          73.96  MPI
      MEMORY        260      20    0.06     0.0        2972.15  MEMORY
         USR        130      10    0.00     0.0          10.26  USR
```

No filtering required.

Demo: Jacobi Solver / Tracing

- Submit job for tracing run

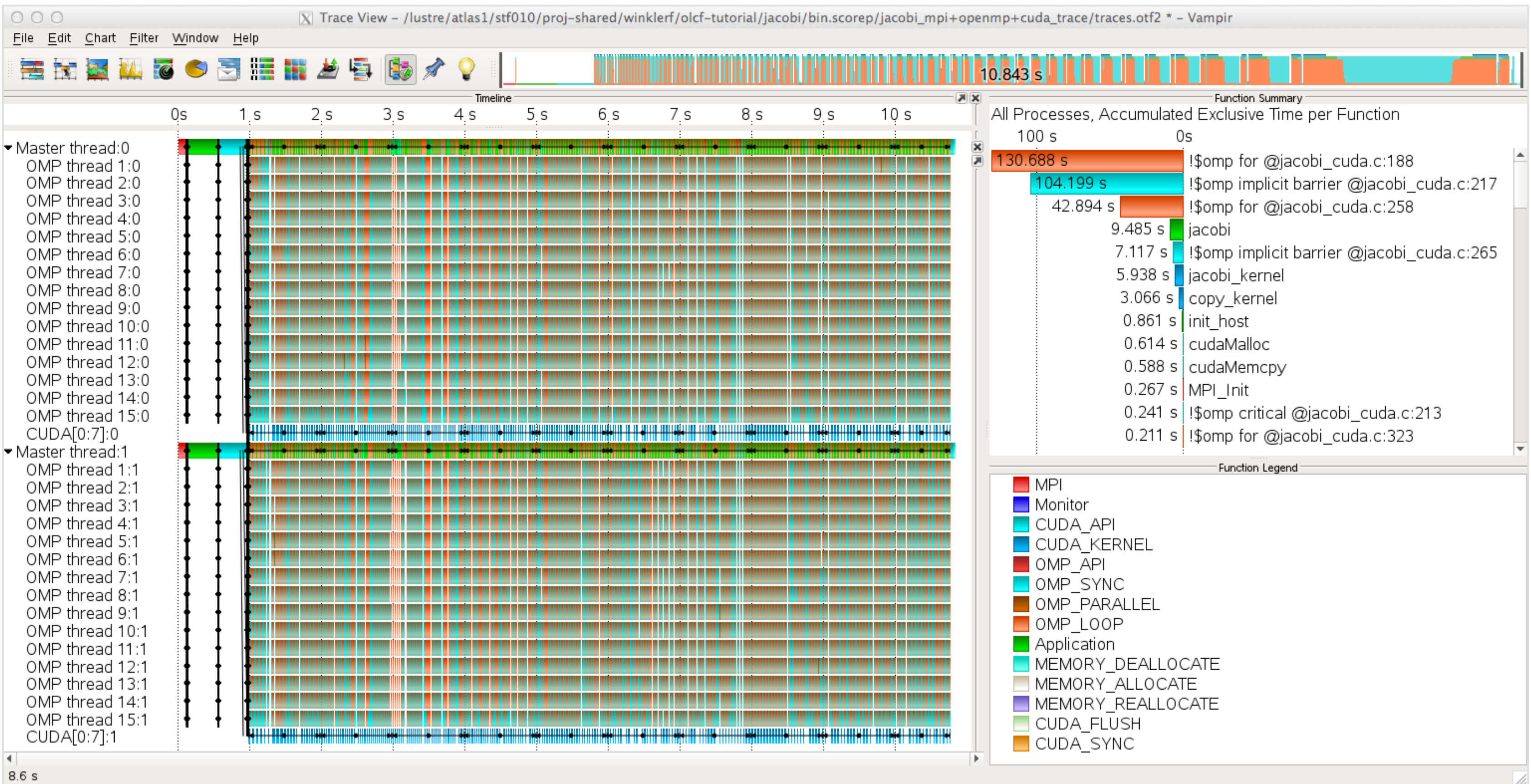
```
$ cd ..
$ less run_trace.pbs
[...]
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_TRACING=true
export SCOREP_EXPERIMENT_DIRECTORY=jacobi_mpi+openmp+cuda_trace
export SCOREP_CUDA_ENABLE=yes
export SCOREP_TIMER=clock_gettime
export SCOREP_MEMORY_RECORDING=yes
export SCOREP_TOTAL_MEMORY=50MB
[...]
aprun -n 2 -d 16 -N 1 ./jacobi_mpi+openmp+cuda 8192 8192 0.15

$ qsub -A [projid] run_trace.pbs
$ less jacobi.o[JOB_ID]
Jacobi relaxation Calculation: 8192 x 8192 mesh with
 2 processes and 16 threads + one Tesla K20X for each process.
 614 of 2049 local rows are calculated on the CPU to balance the load
 between the CPU and the GPU.
    0, 0.489197
  100, 0.002397
  [...]
  900, 0.000269
 total: 9.895828 s
```

Demo: Jacobi Solver / Trace Analysis

- Perform analysis on the trace data with Vampir

```
$ cd bin.scorep
$ module load vampir
$ vampir jacobi_mpi+openmp+cuda_trace/traces.otf2
```



Agenda

Performance Analysis Approaches

- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes

- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools

- Cube
- Vampir

Demo

- Performance Analysis of Jacobi Solver on Titan

Conclusions

Conclusions

Score-P

- Common instrumentation and measurement infrastructure for various analysis tools
- Hides away complicated details
- Provides many options and switches for experts

General Workflow

- Instrument your application with Score-P
- Perform a measurement run with **profiling enabled**
- Perform profile analysis with **Cube**
- Use scorep-score to define an appropriate filter
- Perform a measurement run with **tracing enabled** and the filter applied
- Perform in-depth analysis on the trace data with **Vampir**

If you have any questions or need help, please don't hesitate to contact me under winklerf@ornl.gov**.**

Detailed information under:

http://www.vi-hps.org/projects/score-p or

https://www.olcf.ornl.gov/support/software/

Score-P Advanced Features: Metrics

- Available PAPI metrics
  - Preset events: common set of events deemed relevant and useful for application performance tuning

```
$ papi_avail
```

  - Native events: set of all events that are available on the CPU (platform dependent)

```
$ papi_native_avail
```

- Available resource usage metrics

```
$ man getrusage
    [... Output ...]

  struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    [... More output ...]
```

Score-P Advanced Features: Metrics (2)

- Recording hardware counters via PAPI

```
$ export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_FP_INS
```

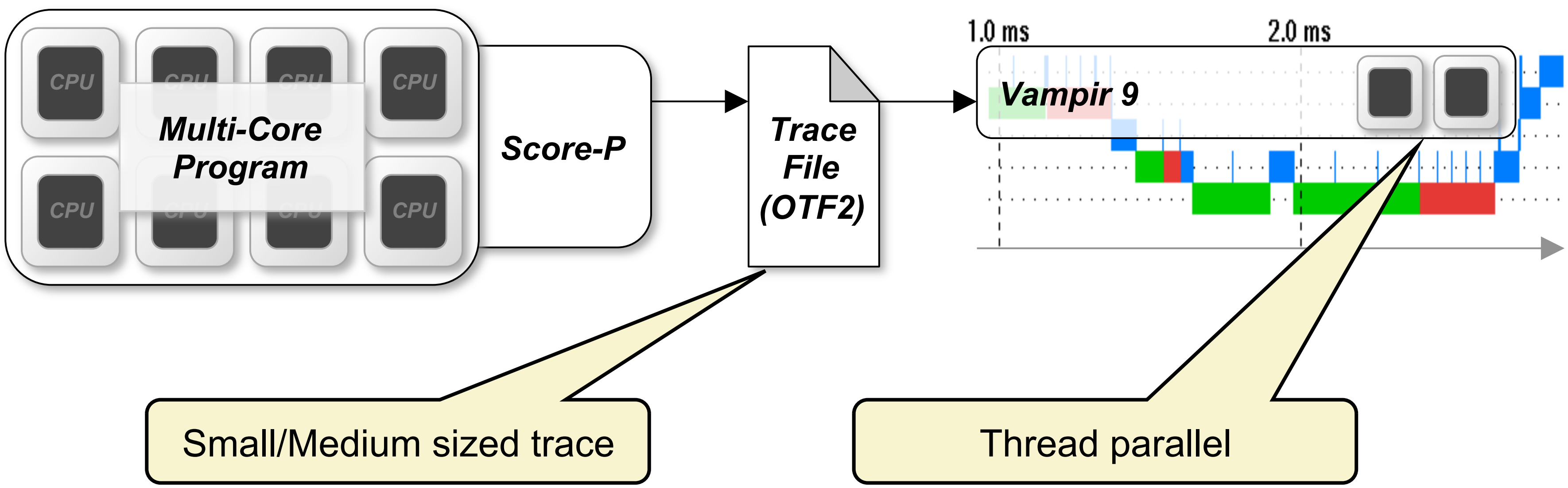
- Recording operating system resource usage

```
$ export SCOREP_METRIC_RUSAGE=ru_maxrss,ru_stime
```

Vampir: Visualization Modes (1)

- Directly on front end or local machine

```
$ vampir
```

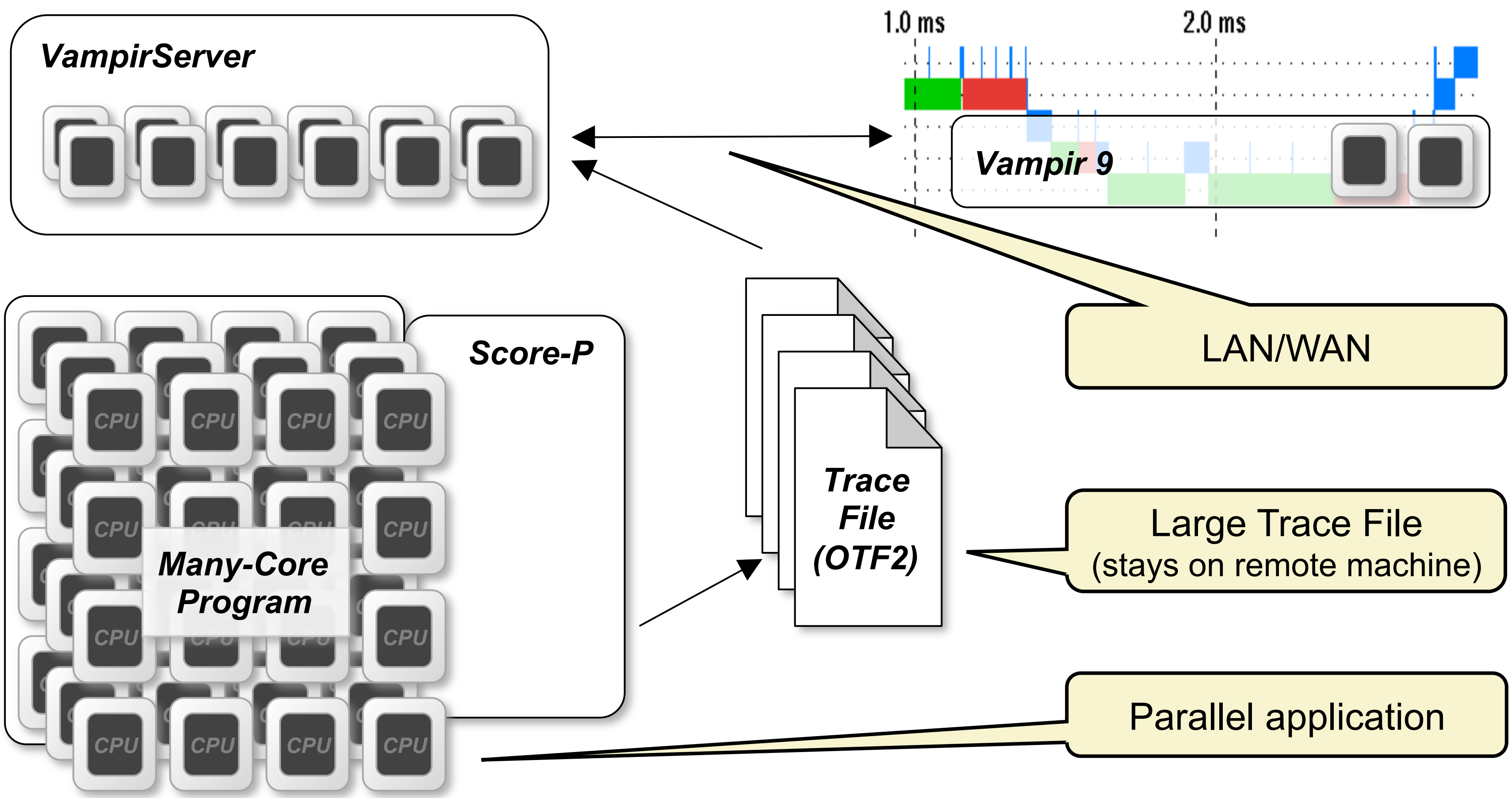


Vampir: Visualization Modes (2)

- On local machine with remote VampirServer

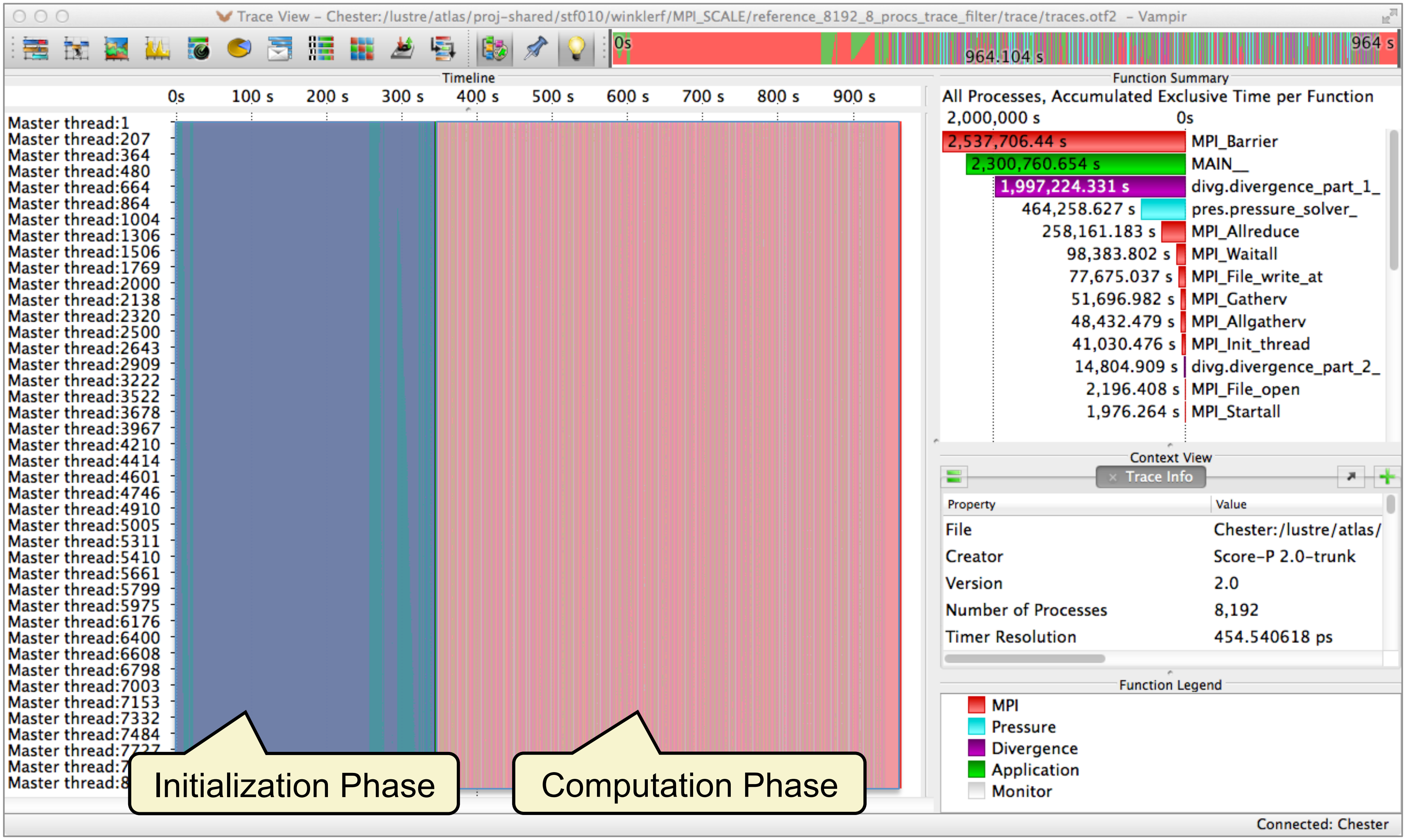
```
$ vampirserver start –n 16
```

```
$ vampir
```



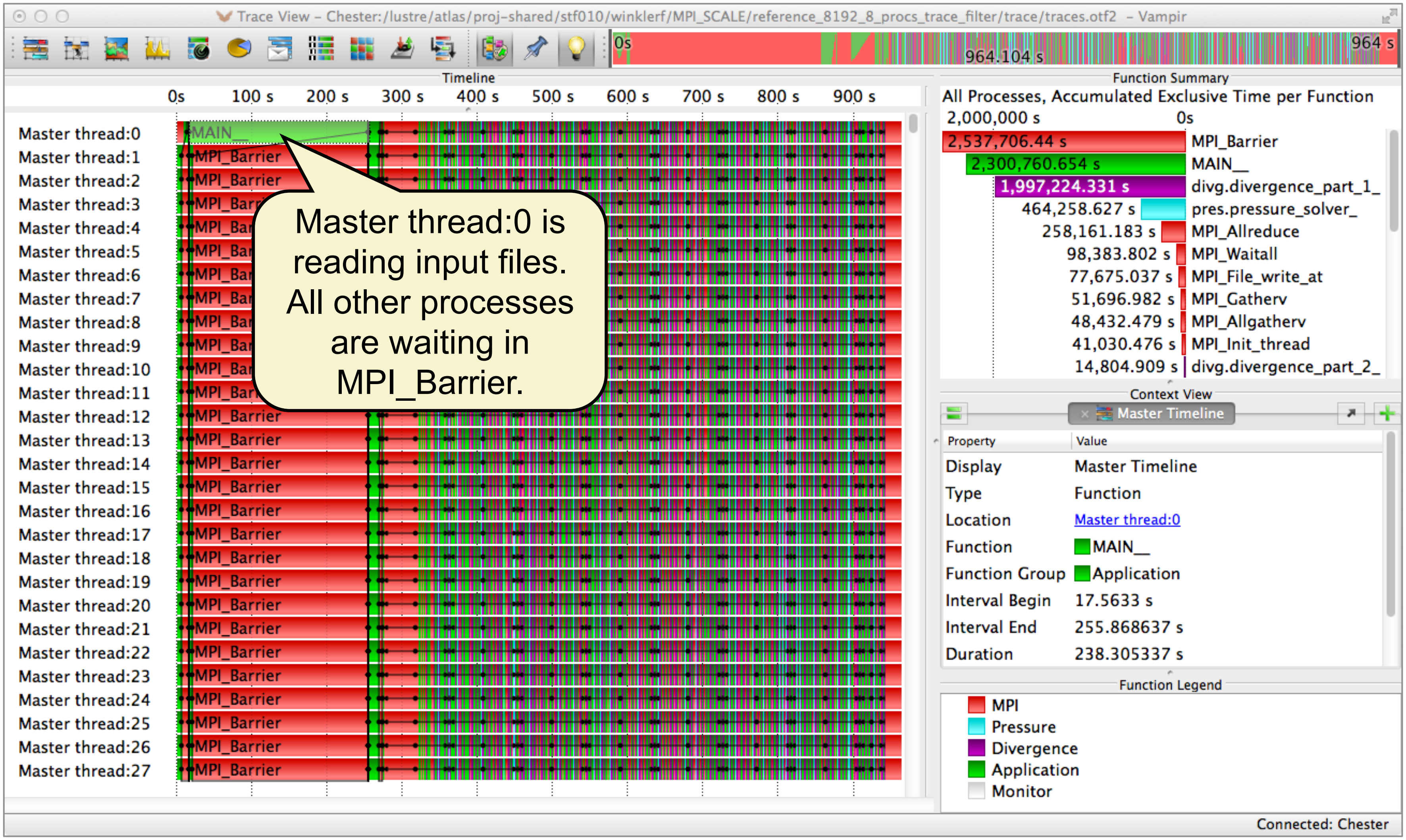
Vampir Bonus: Case Study of FDS

- Indentification of program phases



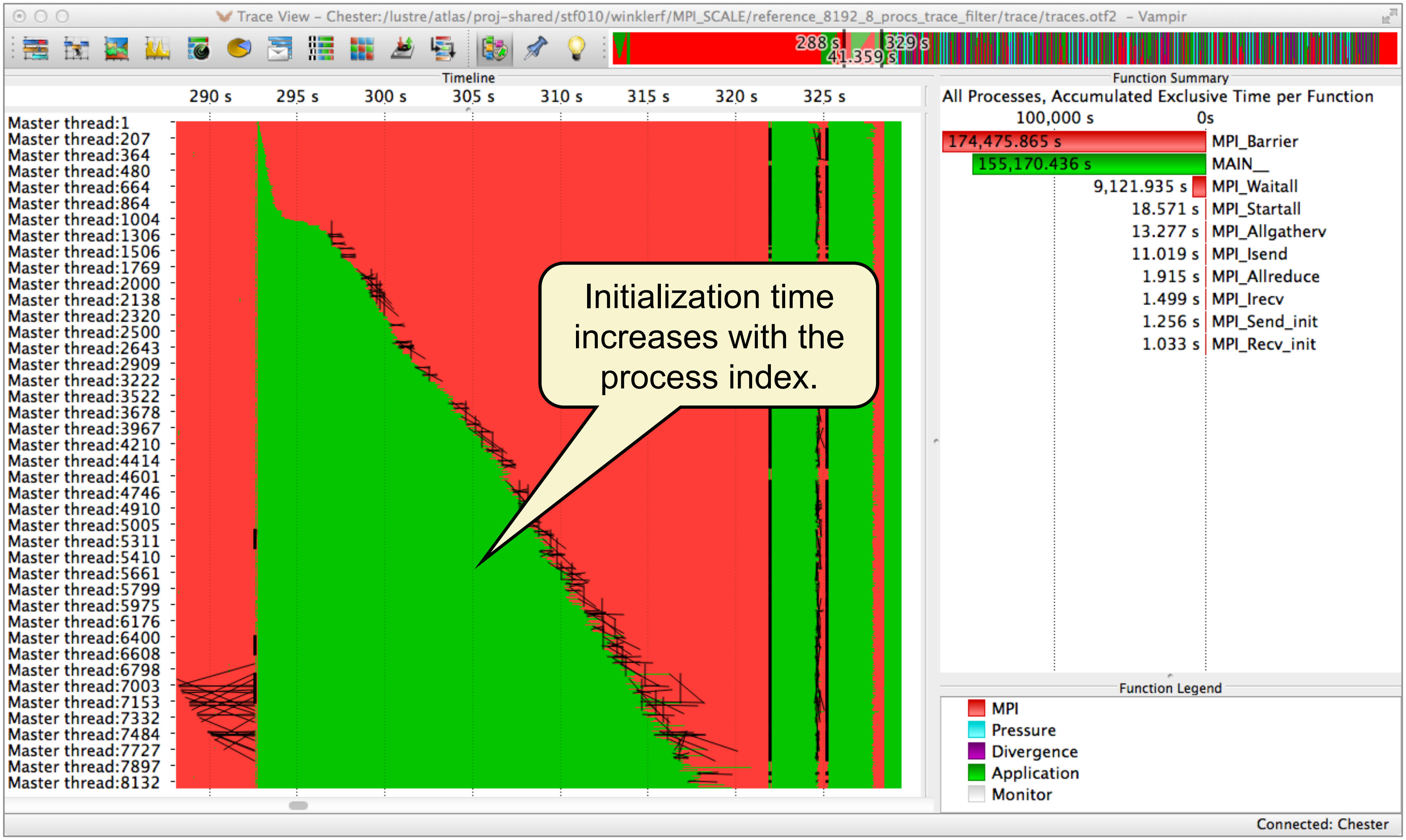
Vampir Bonus: Case Study of FDS

- Load imbalance in initialization phase



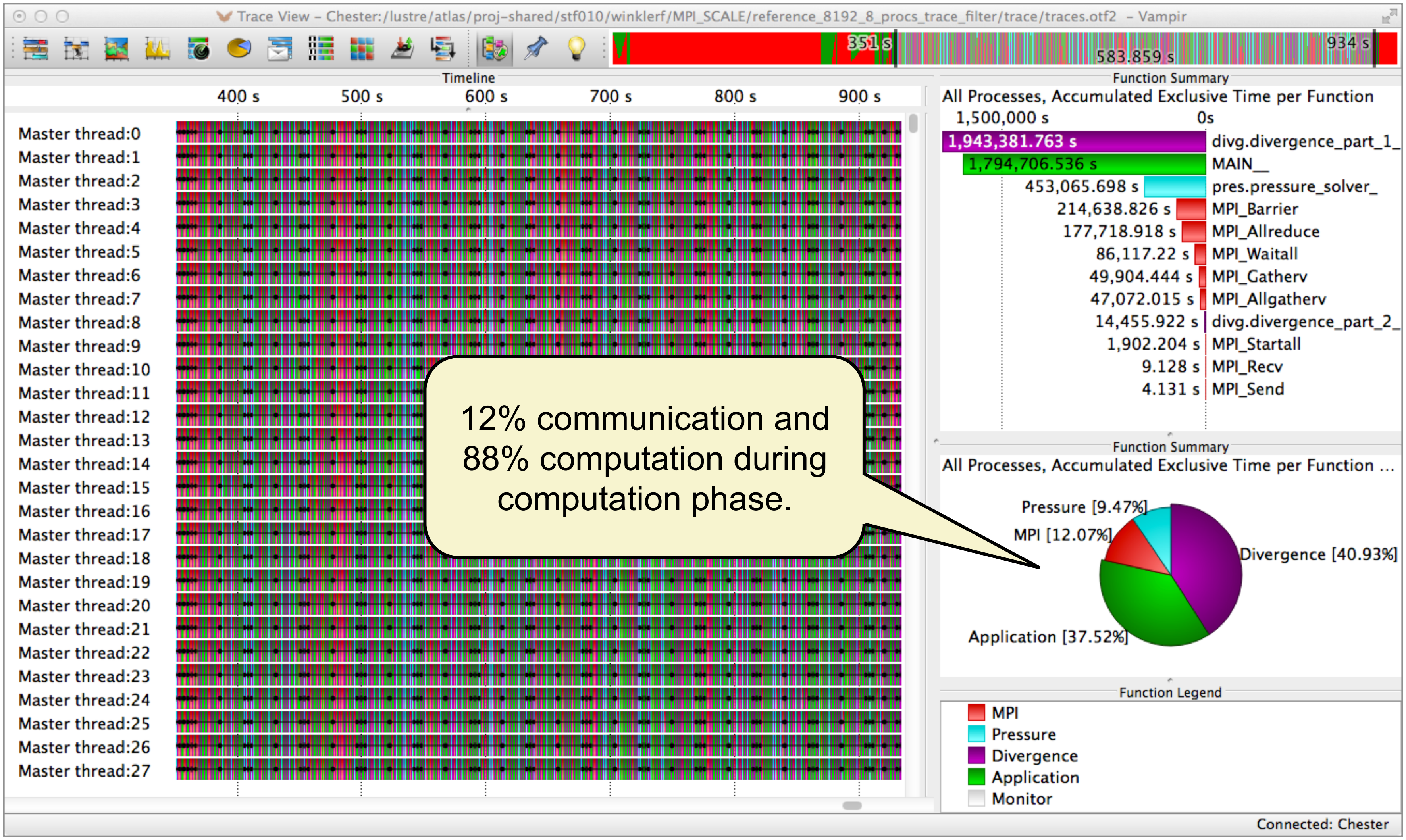
Vampir Bonus: Case Study of FDS

- Load imbalance in initialization phase (2)



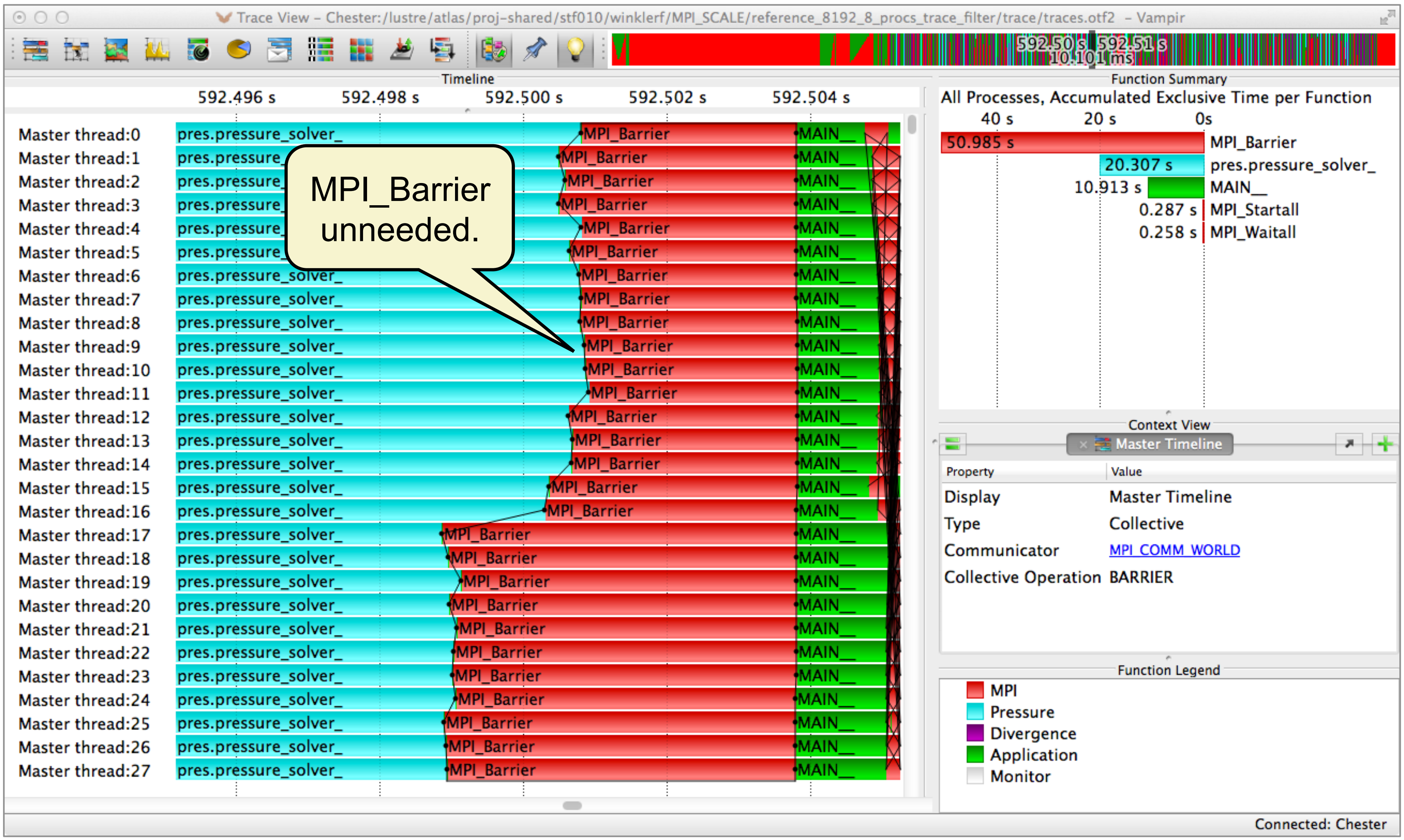
Vampir Bonus: Case Study of FDS

- Computation phase



Vampir Bonus: Case Study of FDS

- Unnecessary synchronization in computation phase



Vampir Bonus: Case Study of FDS

- Inefficient cache usage in computation phase