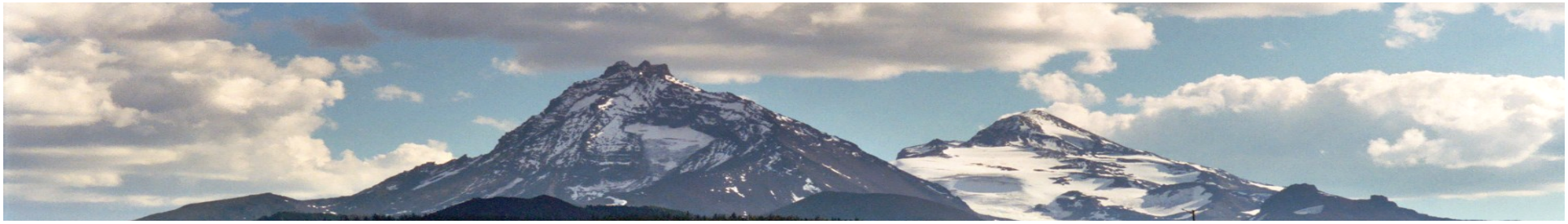


Lecture 4:
Parallel Tools Landscape – Part 1

Allen D. Malony
Department of Computer and Information Science

UNIVERSITY OF OREGON

Short Course Outline

- Lecture 1: Introduction and Fundamentals
- Lecture 2: Methodology
- Lecture 3: Tools Technology
- Lecture 4: Tools Landscape – Part 1
- Lecture 5: Tools Landscape – Part 2
- Lecture 6: TAU Performance System
- Lecture 7: TAU Applications
- Lecture 8: Advances in TAU
- Lecture 9: Future Directions

Tuesday

Wednesday

Thursday

Performance and Debugging Tools

Performance Measurement and Analysis:

- Open|SpeedShop
- HPCToolkit
- Vampir
- Scalasca
- Periscope
- mpiP
- Paraver
- PerfExpert
- TAU

Modeling and prediction

- Prophesy
- MuMMI

Debugging

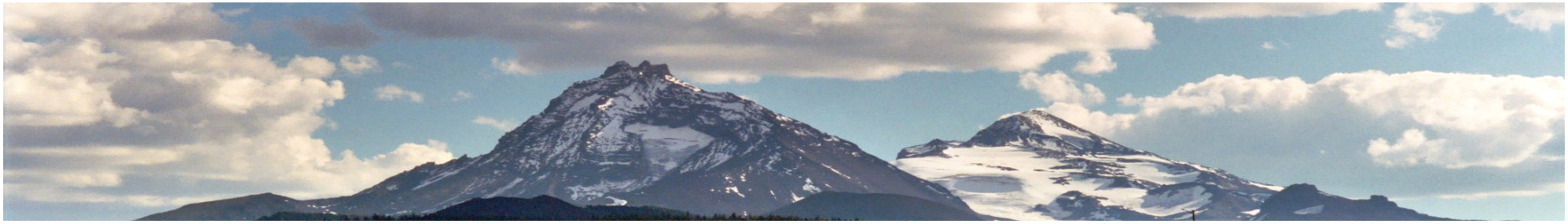
- Stat

Autotuning Frameworks

- Active Harmony

Performance Tools Matrix

TOOL	Profiling	Tracing	Instrumentation	Sampling
Scalasca	X	X	X	X
HPCToolkit	X	X		X
Vampir		X	X	
Open|SpeedShop	X	X	X	X
Periscope	X		X	
mpiP	X	X	X	
Paraver		X	X	X
TAU	X	X	X	X



Open|SpeedShop

Krell Institute (USA)

http://www.openspeedshop.org

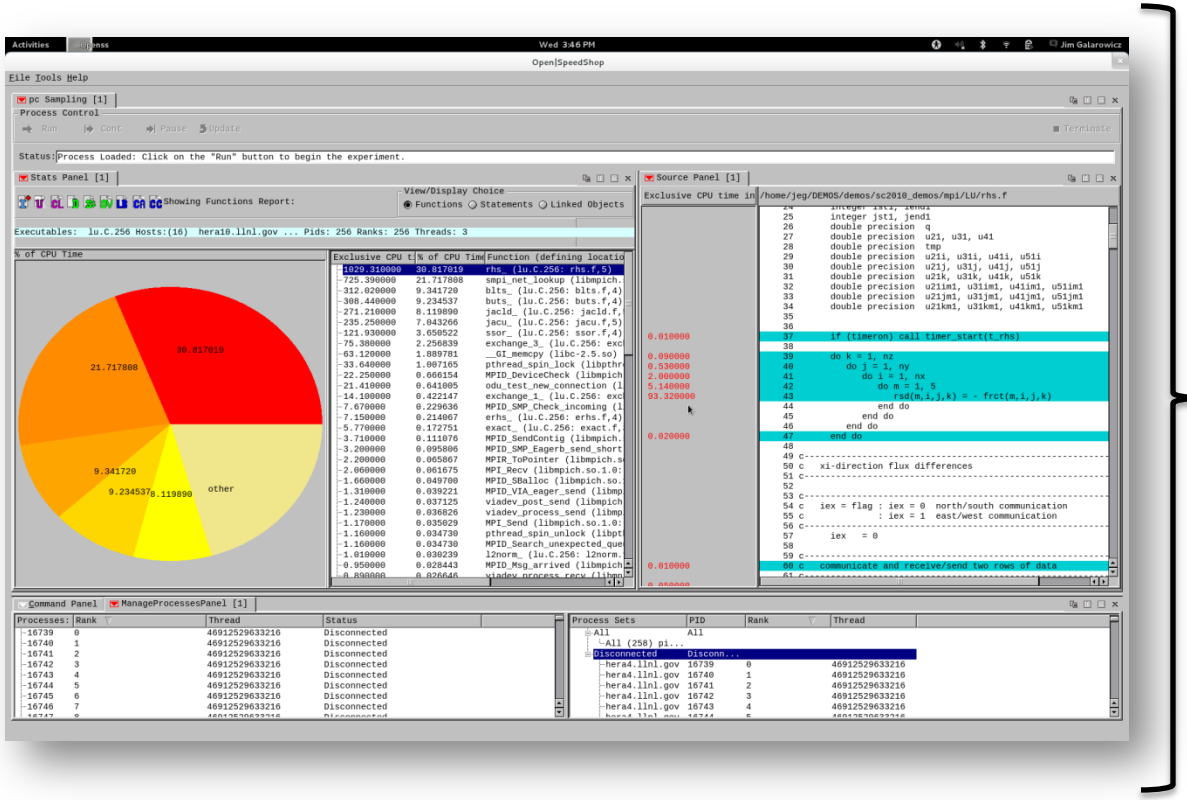
Open|SpeedShop

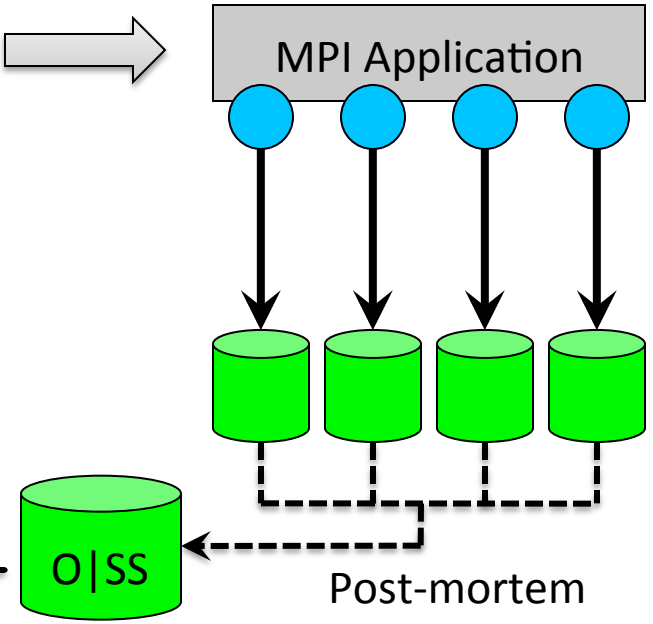
Open|SpeedShop Tool Set

- Open Source Performance Analysis Tool Framework
  - Most common performance analysis steps ***all in one tool***
  - Combines ***tracing*** and ***sampling*** techniques
  - ***Extensible*** by plugins for data collection and representation
  - Gathers and displays several types of performance information
- Flexible and Easy to use
  - User access through:
    GUI, ***Command Line***, ***Python Scripting, convenience scripts***
- Scalable Data Collection
  - Instrumentation of ***unmodified application binaries***
  - New option for ***hierarchical online data aggregation***
- Supports a wide range of systems
  - Extensively used and tested on a variety of ***Linux clusters***
  - ***Cray XT/XE/XK*** and ***Blue Gene L/P/Q*** support

Open|SpeedShop Workflow

osspcsamp “srun –n4 –N1 smg2000 –n 65 65 65”





Alternative Interfaces

- Scripting language
  - Immediate command interface
  - O|SS interactive command line (CLI)
- Python module

```
Experiment Commands
   expAttach
   expCreate
   expDetach
   expGo
   expView

List Commands
   list –v exp
```

```
import openss

my_filename=openss.FileList("myprog.a.out")
my_exptype=openss.ExpTypeList("pcsamp")
my_id=openss.expCreate(my_filename,my_exptype)

openss.expGo()

My_metric_list = openss.MetricList("exclusive")
my_viewtype = openss.ViewTypeList("pcsamp”)
result = openss.expView(my_id,my_viewtype,my_metric_list)
```

Central Concept: Experiments

- Users pick experiments:
  - What to measure and from which sources?
  - How to select, view, and analyze the resulting data?
- Two main classes:
  - Statistical Sampling
    - periodically interrupt execution and record location
    - useful to get an overview
    - low and uniform overhead
  - Event Tracing (DyninstAPI)
    - gather and store individual application events
    - provides detailed per event information
    - can lead to huge data volumes
- O|SS can be extended with additional experiments

Sampling Experiments in O|SS

- PC Sampling (pcsamp)
  - Record PC repeatedly at user defined time interval
  - Low overhead overview of time distribution
  - Good first step, lightweight overview
- Callpath Profiling (usertime)
  - PC sampling and call stacks for each sample
  - Provides inclusive and exclusive timing data
  - Use to find hot call paths, whom is calling who
- Hardware Counters (hwc, hwctime, hwcsamp)
  - Access to data like cache and TLB misses
  - hwc, hwctime:
    - sample a HWC event based on an event threshold
    - default event is PAPI_TOT_CYC overflows
  - hwcsamp:
    - periodically sample up to 6 counter events based (hwcsamp)
    - default events are PAPI_FP_OPS and PAPI_TOT_CYC

Tracing Experiments in O|SS

- Input/Output Tracing (io, iop, iot)
  - Record invocation of all POSIX I/O events
  - Provides aggregate and individual timings
  - Lightweight I/O profiling (iop)
  - Store function arguments and return code for each call (iot)
- MPI Tracing (mpi, mpit, mpiotf)
  - Record invocation of all MPI routines
  - Provides aggregate and individual timings
  - Store function arguments and return code for each call (mpit)
  - Create Open Trace Format (OTF) output (mpiotf)
- Floating Point Exception Tracing (fpe)
  - Triggered by any FPE caused by the application
  - Helps pinpoint numerical problem areas

Performance Analysis in Parallel

- How to deal with concurrency?
  - Any experiment can be applied to parallel application
    - Important step: aggregation or selection of data
  - Special experiments targeting parallelism/synchronization
- O|SS supports MPI and threaded codes
  - Automatically applied to all tasks/threads
  - Default views aggregate across all tasks/threads
  - Data from individual tasks/threads available
  - Thread support (incl. OpenMP) based on POSIX threads
- Specific parallel experiments (e.g., MPI)
  - Wraps MPI calls and reports
    - MPI routine time
    - MPI routine parameter information
  - The mpit experiment also store function arguments and return code for each call

How to Run a First Experiment in O|SS?

1. Picking the experiment
   - What do I want to measure?
   - *We will start with pcsamp to get a first overview*
2. Launching the application
   - How do I control my application under O|SS?
   - *Enclose how you normally run your application in quotes*
   - ***osspcsamp*** *"mpirun –np 256 smg2000 –n 65 65 65"*
3. Storing the results
   - *O|SS will create a database*
   - *Name: smg2000-pcsamp.openss*
4. Exploring the gathered data
   - How do I interpret the data?
   - *O|SS will print a default report*
   - *Open the GUI to analyze data in detail (run: "**openss**")*

Example Run with Output

- osspcsamp “mpirun –np 2 smg2000 –n 65 65 65” (1/2)

```
Bash> osspcsamp "mpirun -np 2 ./smg2000 -n 65 65 65"
[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".
[openss]: Using OPENSS_PREFIX installed in /opt/OSS-mrnet
[openss]: Setting up offline raw data directory in /tmp/jeg/offline-oss
[openss]: Running offline pcsamp experiment using the command:
"mpirun -np 2 /opt/OSS-mrnet/bin/ossrun "./smg2000 -n 65 65 65" pcsamp"

Running with these driver parameters:
 (nx, ny, nz)    = (65, 65, 65)
 …
        <SMG native output>
…
Final Relative Residual Norm = 1.774415e-07
[openss]: Converting raw data from /tmp/jeg/offline-oss into temp file X.0.openss

Processing raw data for smg2000
Processing processes and threads ...
Processing performance data ...
Processing functions and statements ...
```

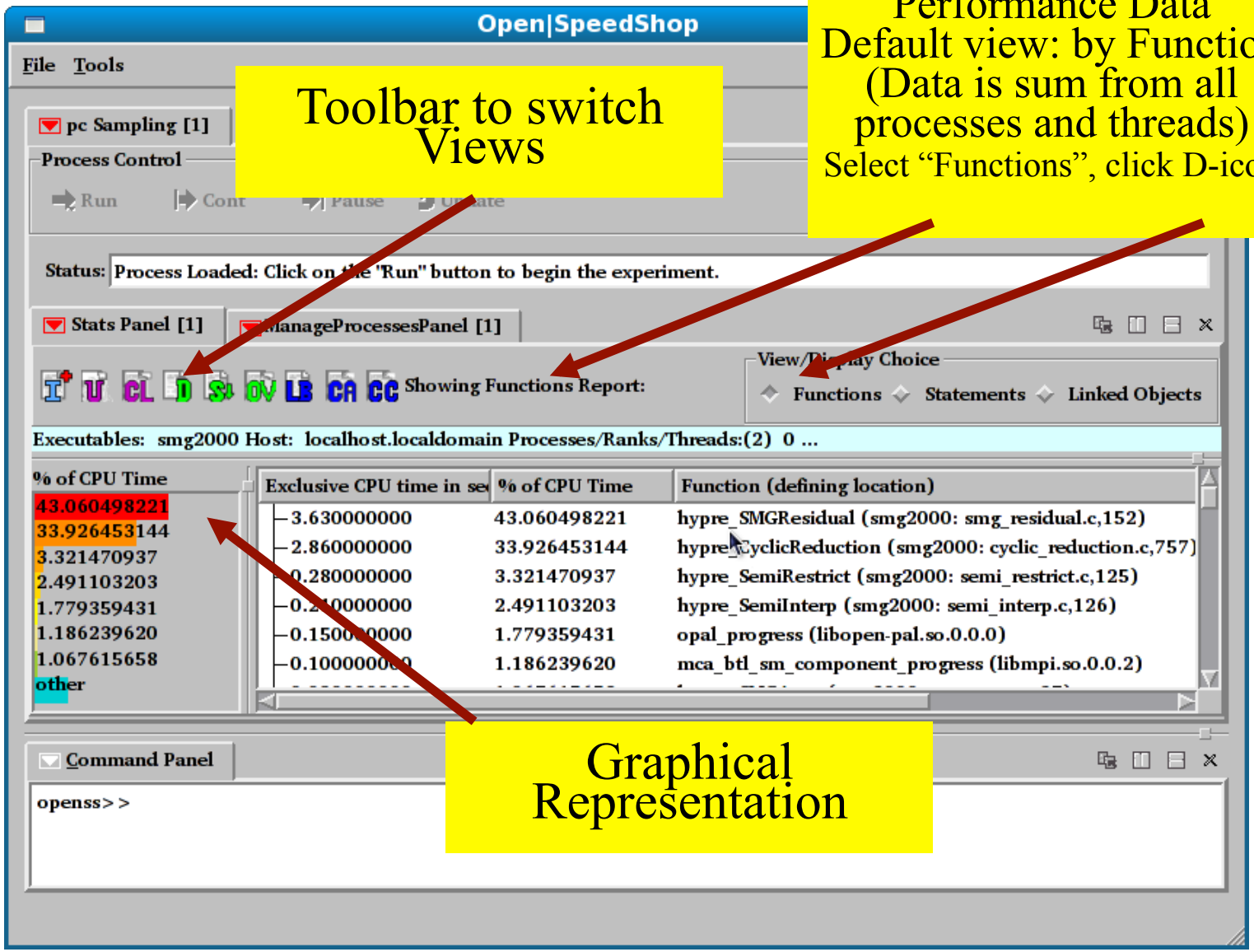
Example Run with Output

- osspcsamp "mpirun –np 2 smg2000 –n 65 65 65" (2/2)

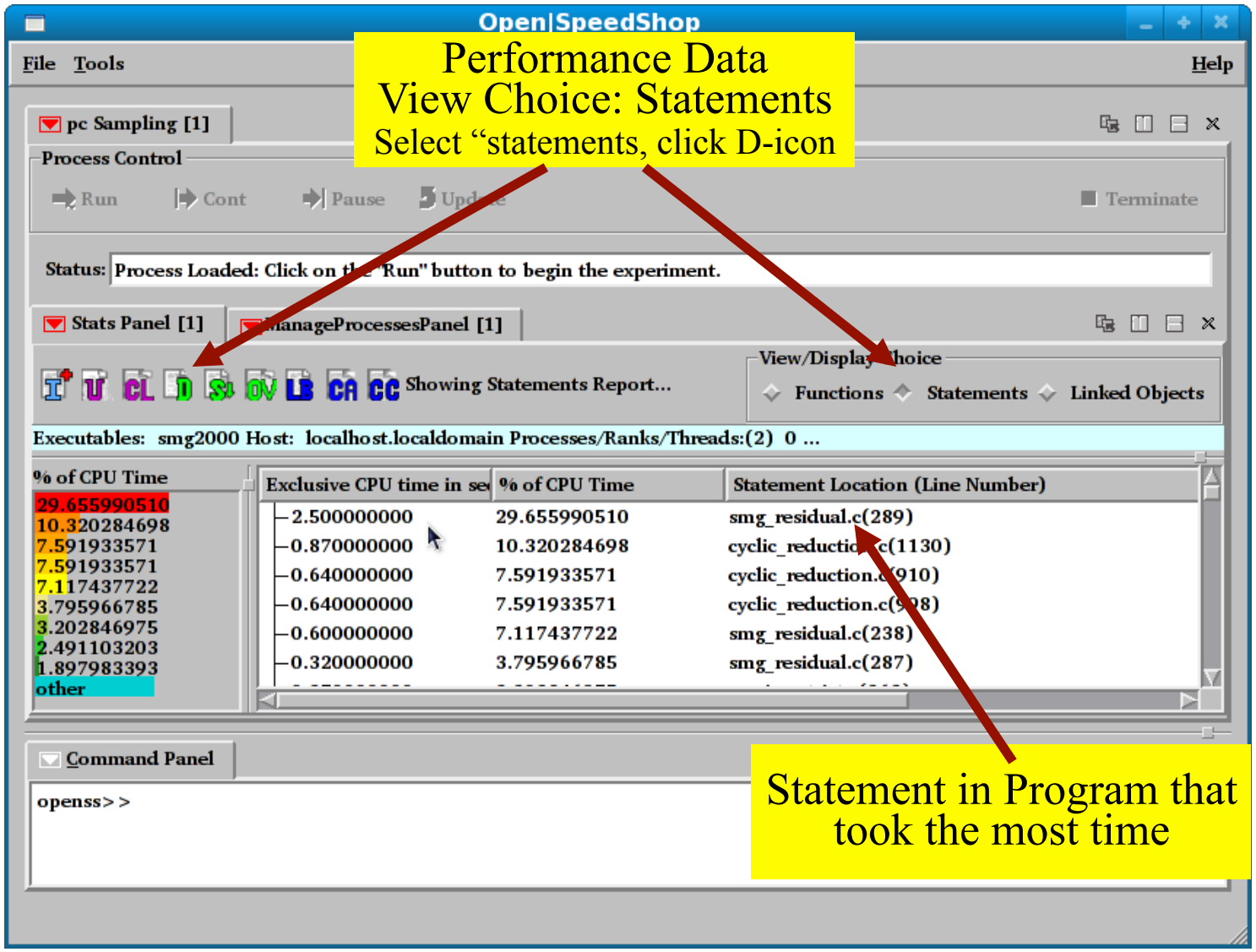
```
[openss]: Restoring and displaying default view for:
      /home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/smg2000/test/smg2000-pcsamp-1.openss
[openss]: The restored experiment identifier is:  -x 1

 Exclusive CPU time      % of CPU Time  Function (defining location)
      in seconds.
      3.630000000       43.060498221  hypre_SMGResidual (smg2000: smg_residual.c,152)
      2.860000000       33.926453144  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
      0.280000000        3.321470937  hypre_SemiRestrict (smg2000: semi_restrict.c,125)
      0.210000000        2.491103203  hypre_SemiInterp (smg2000: semi_interp.c,126)
      0.150000000        1.779359431  opal_progress (libopen-pal.so.0.0.0)
      0.100000000        1.186239620 mca_btl_sm_component_progress (libmpi.so.0.0.2)
      0.090000000        1.067615658  hypre_SMGAxpy (smg2000: smg_axpy.c,27)
      0.080000000        0.948991696  ompi_generic_simple_pack (libmpi.so.0.0.2)
      0.070000000        0.830367734  __GI_memcpy (libc-2.10.2.so)
      0.070000000        0.830367734 hypre_StructVectorSetConstantValues (smg2000:
struct_vector.c,537)
      0.060000000        0.711743772  hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233)
```

View with GUI: openss –f smg2000-pcsamp-1.openss

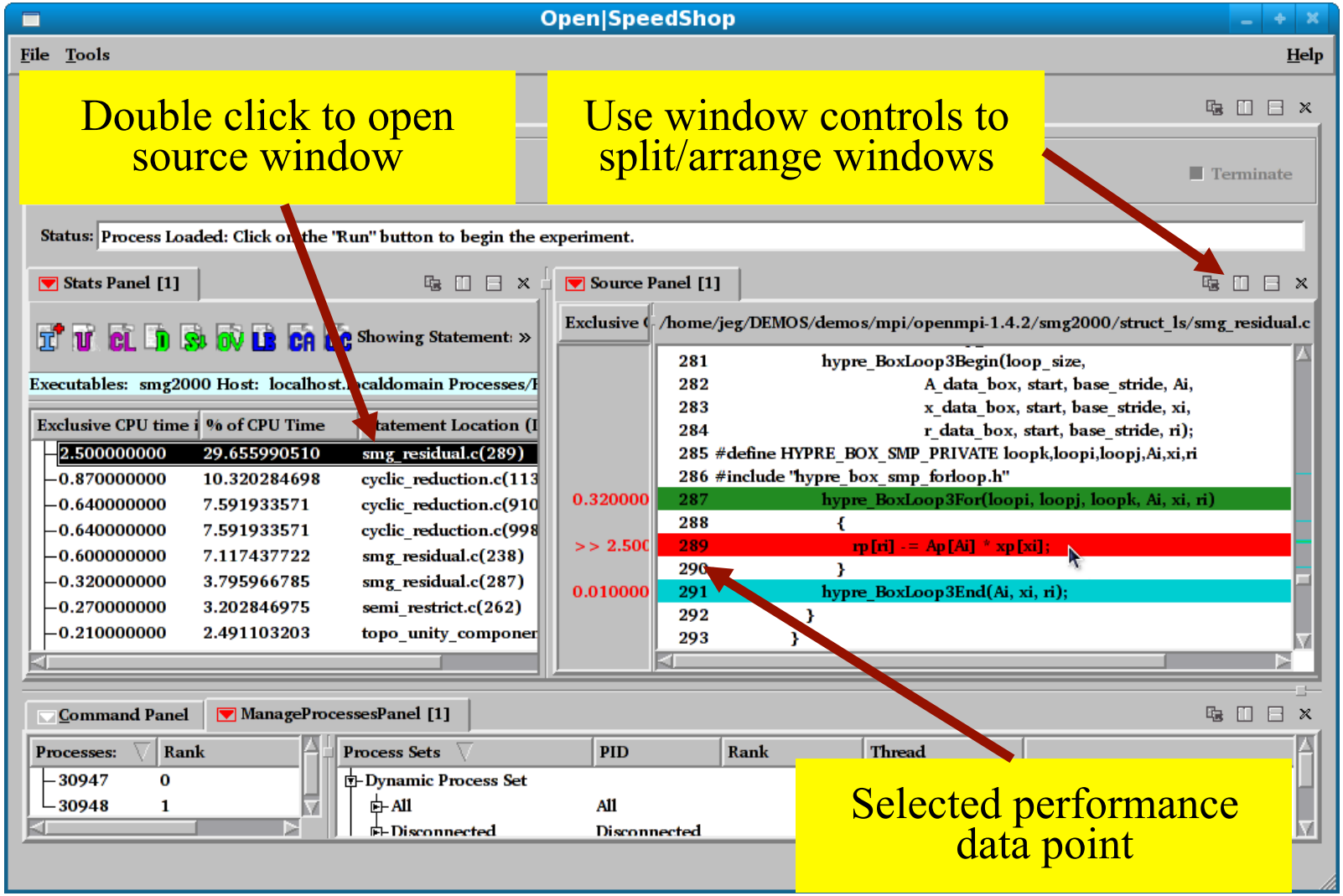
Default Output Report View



Statement Report Output View

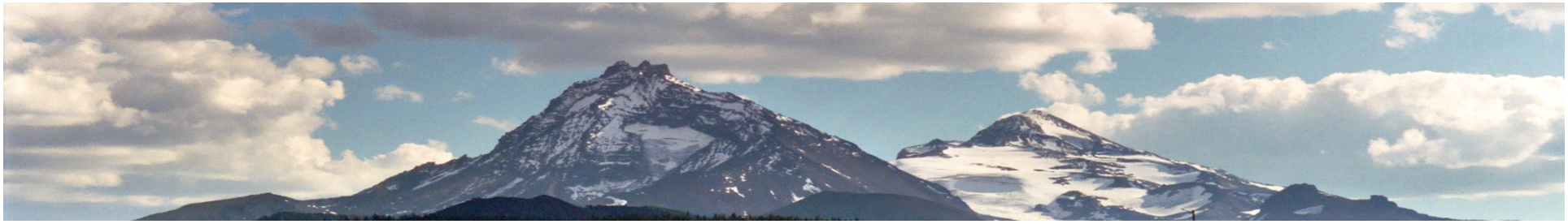


Associate Source & Performance Data



Open|SpeedShop Summary

- Place **the way you run your application normally** in quotes and pass it as an argument to osspcsamp, or any of the other experiment convenience scripts: ossio, ossmpi, etc.
  - osspcsamp "srun –N 8 –n 64 ./mpi_application app_args"
- Open|SpeedShop sends a summary profile to stdout
- Open|SpeedShop creates a database file
- Display alternative views of the data with the GUI via:
  - openss –f <database file>
- Display alternative views of the data with the CLI via:
  - openss –cli –f <database file>
- On clusters, need to set OPENSS_RAWDATA_DIR
  - Should point to a directory in a shared file system
  - More on this later – usually done in a module or dotkit file.
- Start with pcsamp for overview of performance
- Then look into performance issues with other experiments



HPCToolkit

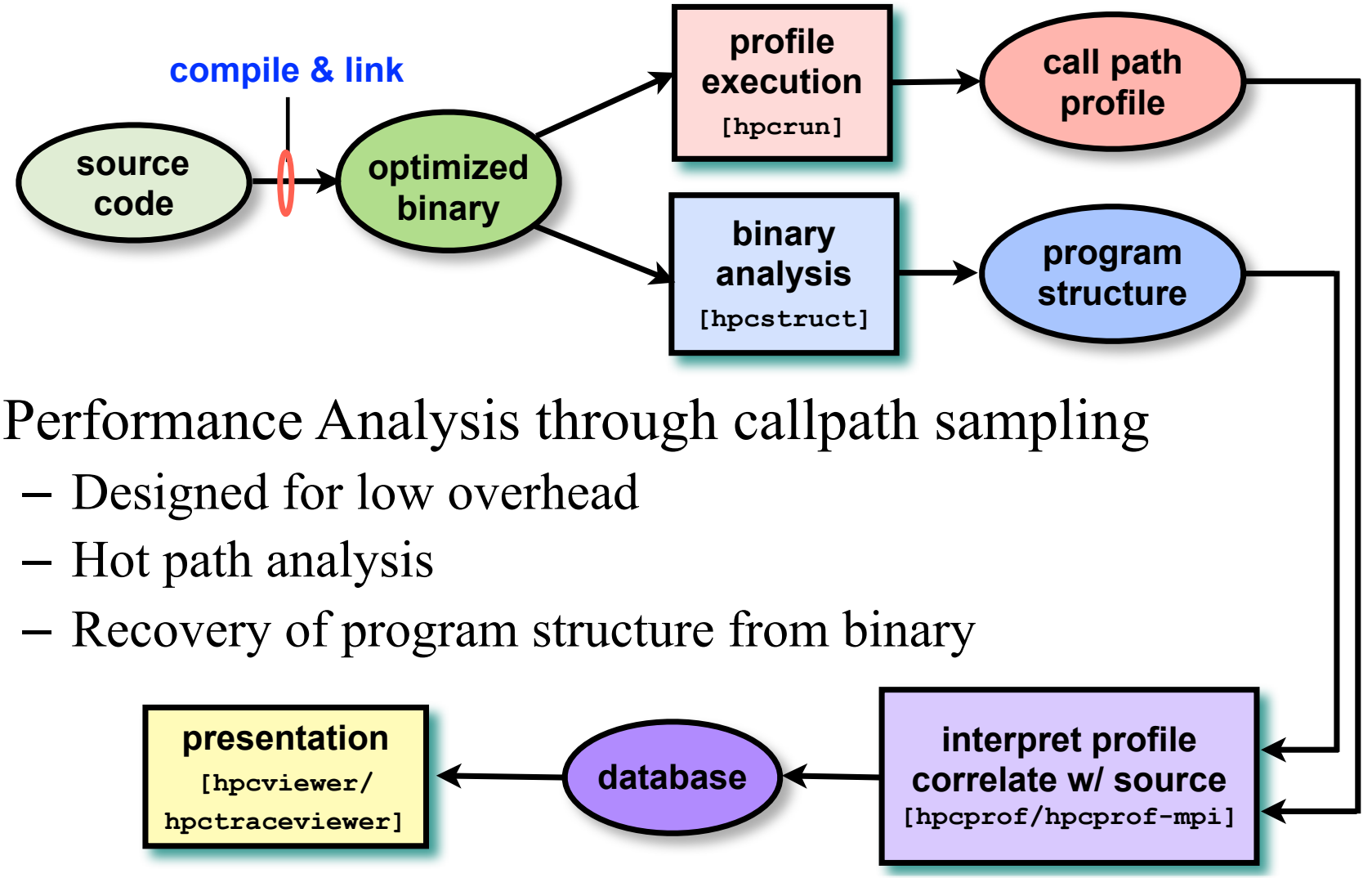
John Mellor-Crummey
Rice University (USA)

http://hpctoolkit.org

HPCToolkit

- Integrated suite of tools for measurement and analysis of program performance
- Works with multilingual, fully optimized applications that are statically or dynamically linked
- Sampling-based measurement methodology
- Serial, multiprocess, multithread applications

HPCToolkit



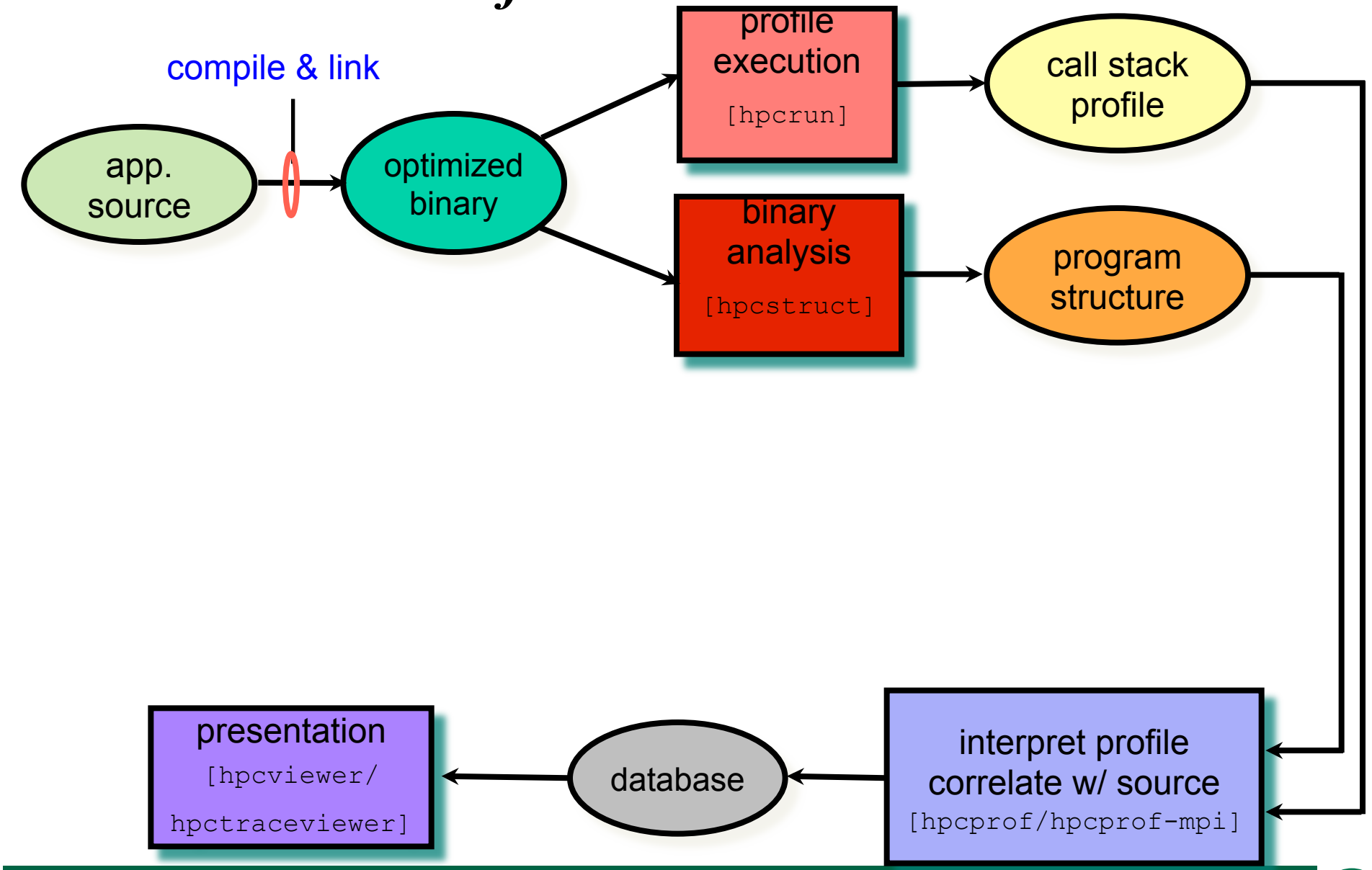
- Performance Analysis through callpath sampling
  - Designed for low overhead
  - Hot path analysis
  - Recovery of program structure from binary

Image by John Mellor-Crummey

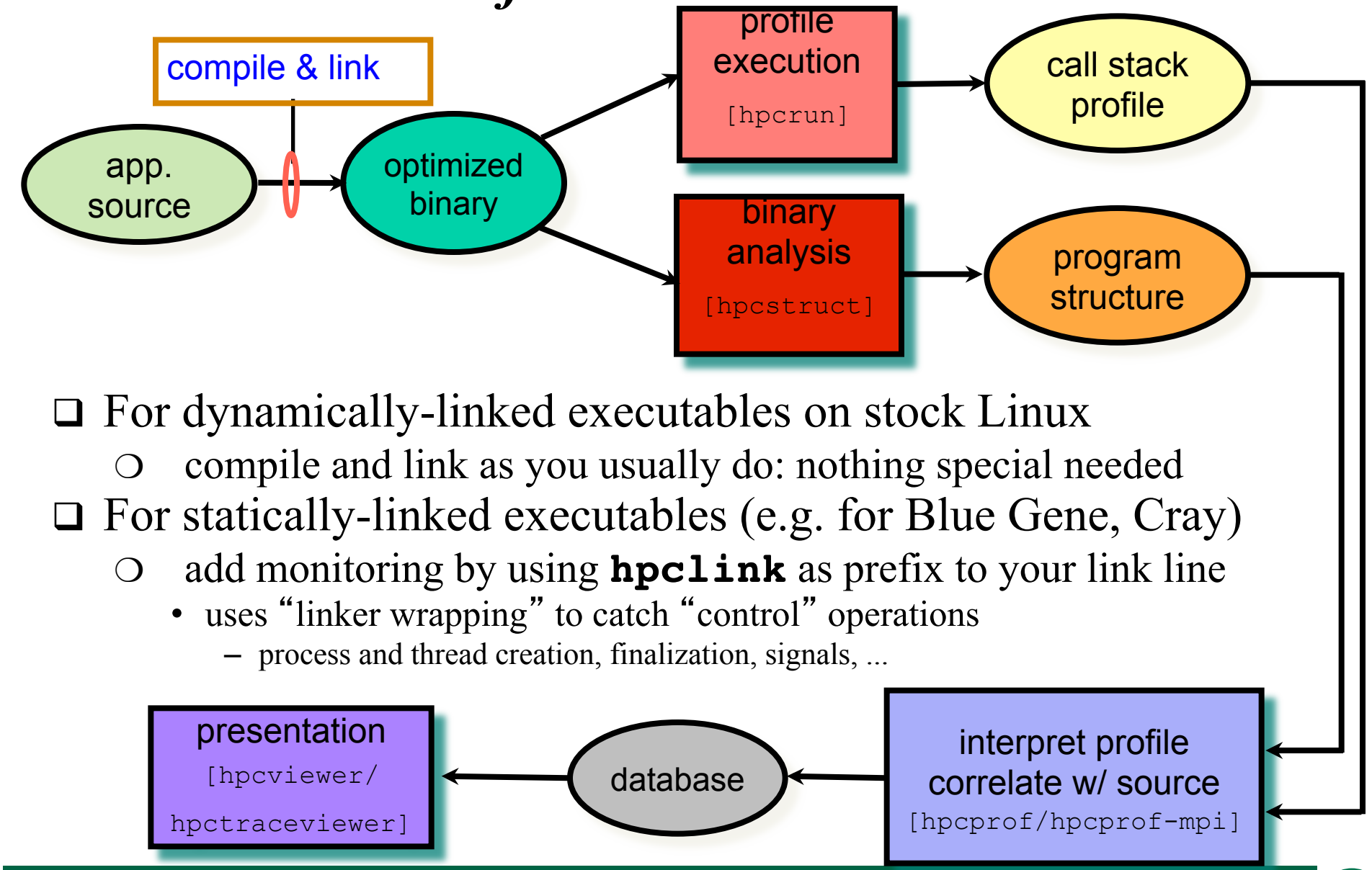
HPCToolkit DESIGN PRINCIPLES

- Employ binary-level measurement and analysis
  - observe *fully optimized*, dynamically linked executions
  - support *multi-lingual codes* with external binary-only libraries
- Use sampling-based measurement (avoid instrumentation)
  - controllable overhead
  - minimize systematic error and avoid blind spots
  - enable data collection for large-scale parallelism
- Collect and correlate multiple derived performance metrics
  - diagnosis typically requires more than one species of metric
- Associate metrics with both static and dynamic context
  - loop nests, procedures, inlined code, calling context
- Support top-down performance analysis
  - natural approach that minimizes burden on developers

HPCToolkit Workflow

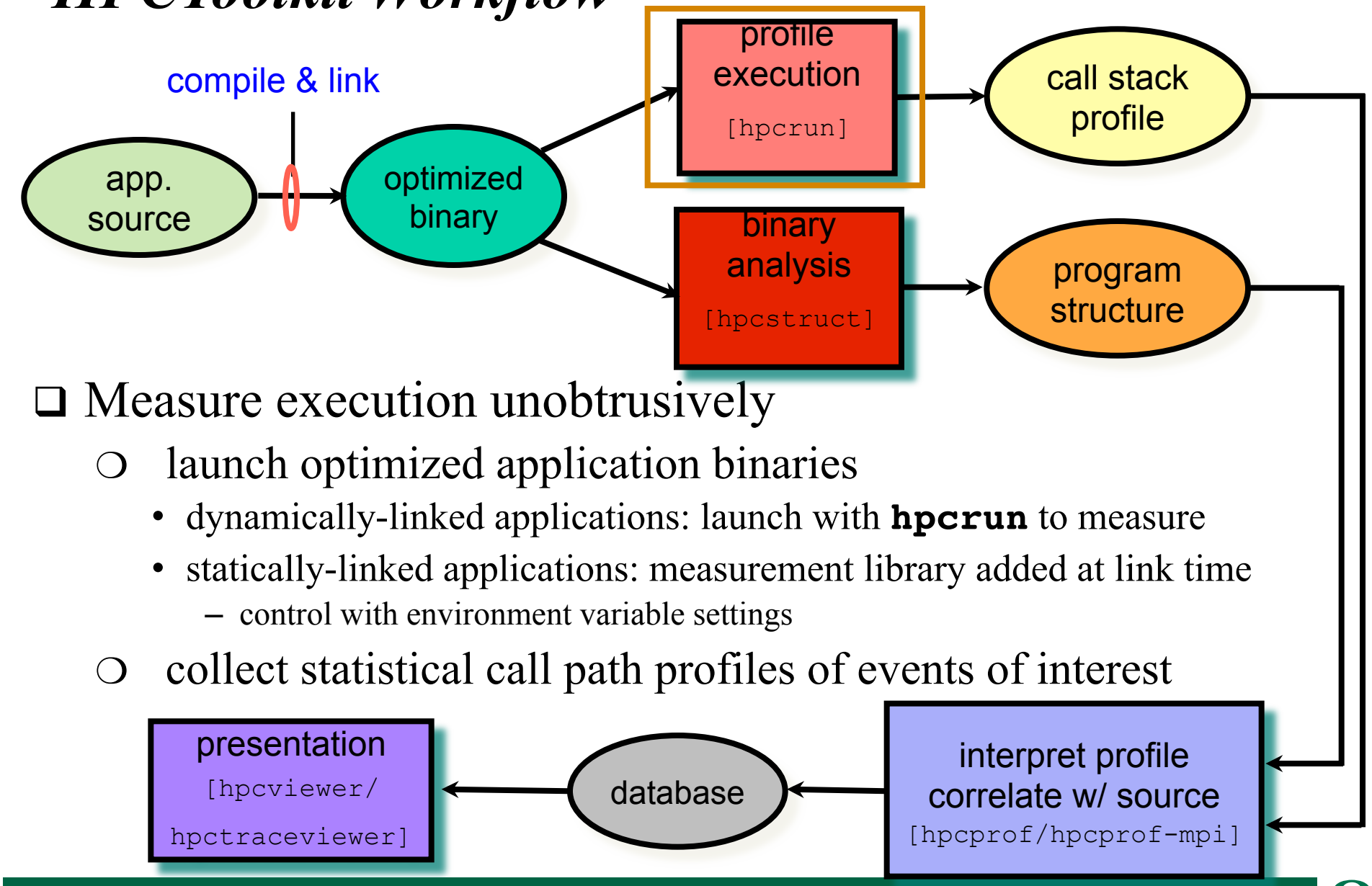


HPCToolkit Workflow



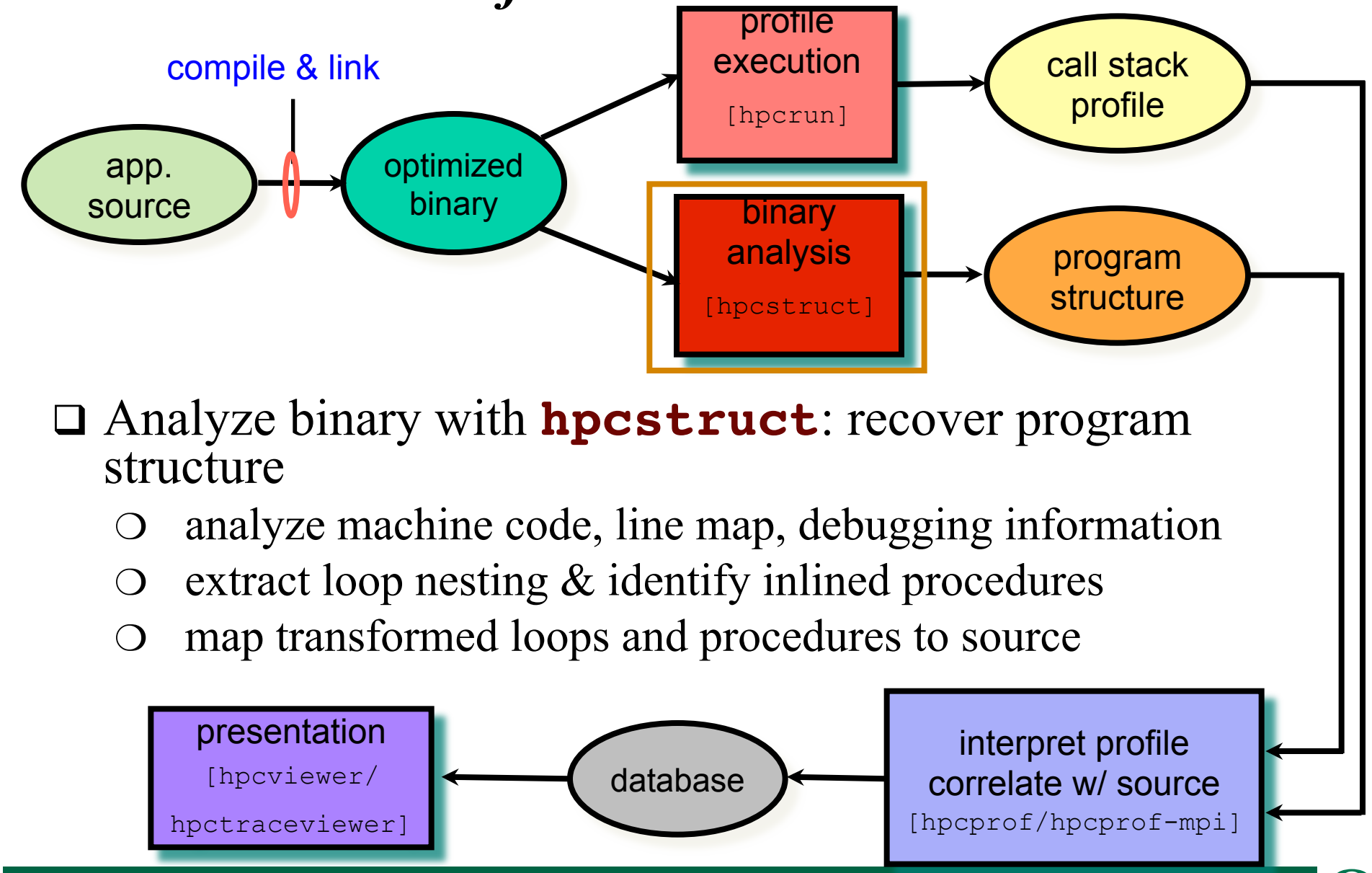
- For dynamically-linked executables on stock Linux
  - compile and link as you usually do: nothing special needed
- For statically-linked executables (e.g. for Blue Gene, Cray)
  - add monitoring by using **`hpclink`** as prefix to your link line
    - uses "linker wrapping" to catch "control" operations
      - process and thread creation, finalization, signals, ...

HPCToolkit Workflow



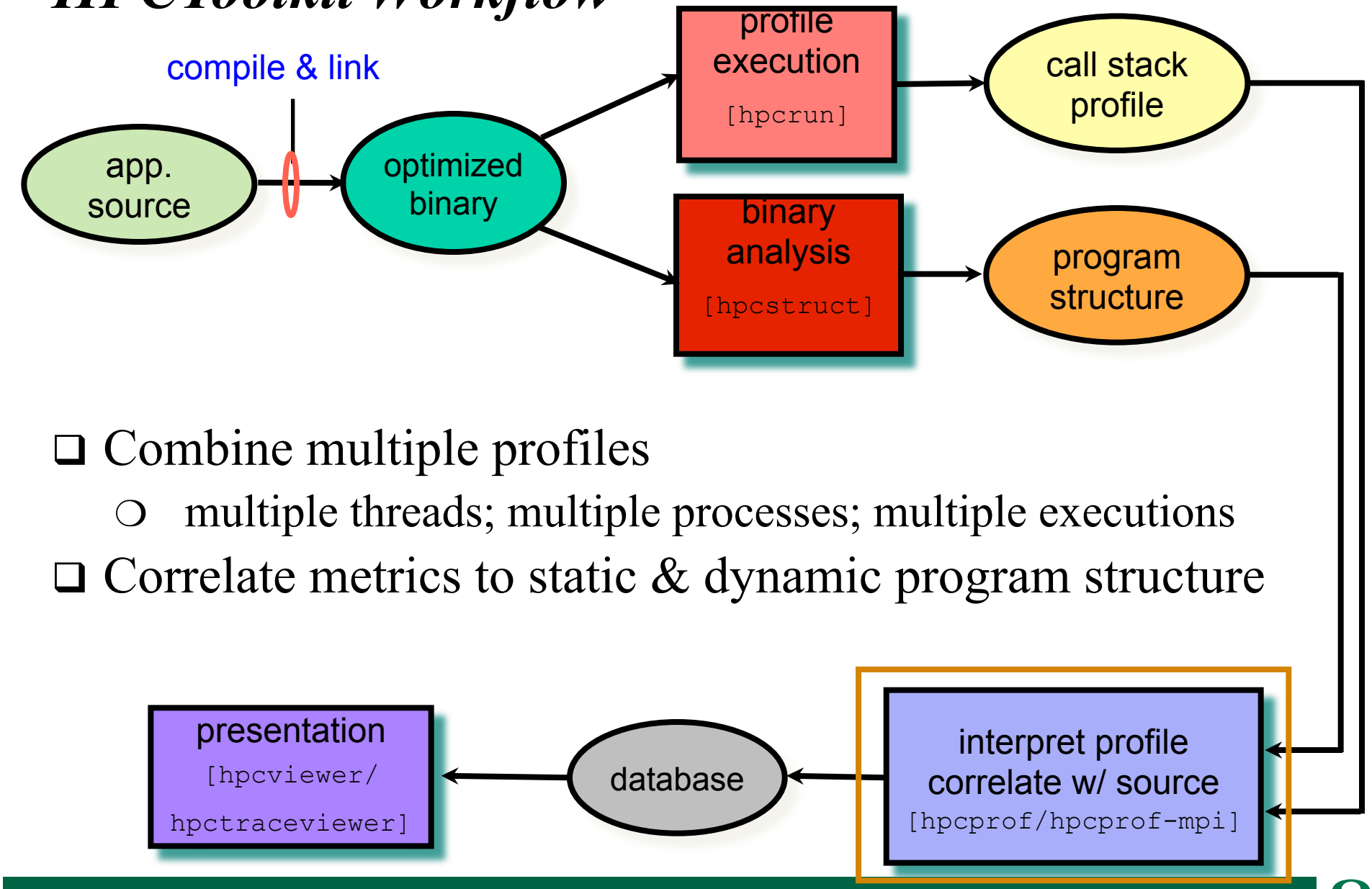
- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with **`hpcrun`** to measure
    - statically-linked applications: measurement library added at link time
      - control with environment variable settings
  - collect statistical call path profiles of events of interest

HPCToolkit Workflow



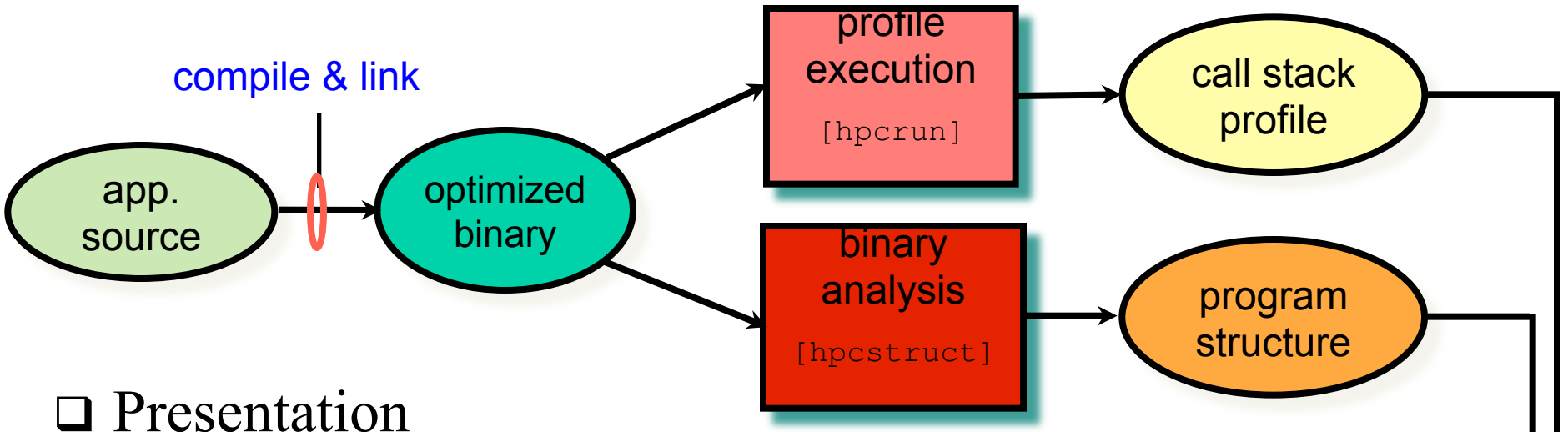
- Analyze binary with **`hpcstruct`**: recover program structure
  - analyze machine code, line map, debugging information
  - extract loop nesting & identify inlined procedures
  - map transformed loops and procedures to source

HPCToolkit Workflow

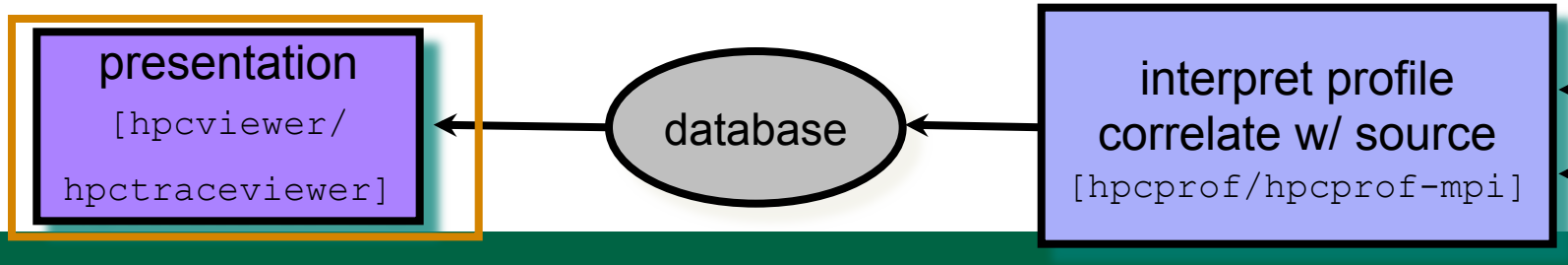


- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate metrics to static & dynamic program structure

HPCToolkit Workflow



- Presentation
  - explore performance data from multiple perspectives
    - rank order by metrics to focus on what's important
    - compute derived metrics to help gain insight
      - e.g. scalability losses, waste, CPI, bandwidth
  - graph thread-level metrics for contexts
  - explore evolution of behavior over time



Analyzing results with hpcviewer

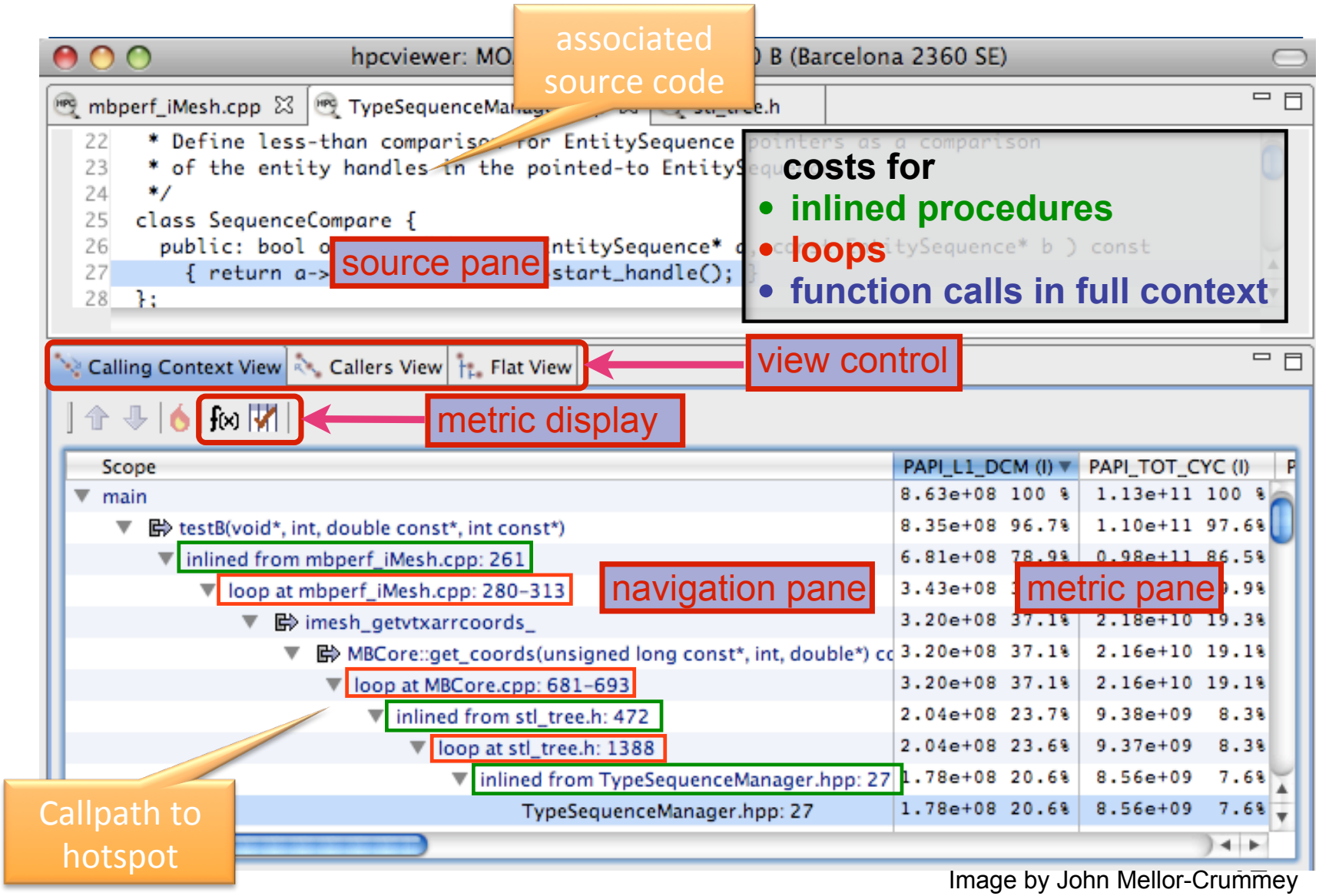


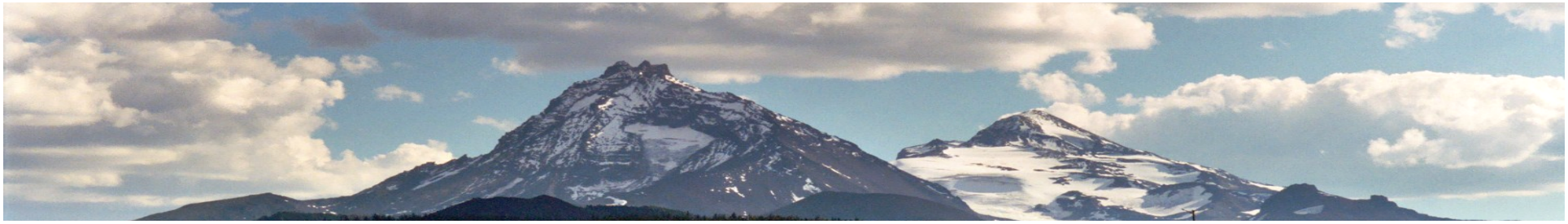
Image by John Mellor-Crummey

Principal Views

- Calling context tree view - "top-down" (down the call chain)
  - Associate metrics with each dynamic calling context
  - High-level, hierarchical view of distribution of costs
  - Example: quantify initialization, solve, post-processing
- Caller's view - "bottom-up" (up the call chain)
  - Apportion a procedure's metrics to its dynamic calling contexts
  - Understand costs of a procedure called in many places
  - Example: see where PGAS put traffic is originating
- Flat view - ignores the calling context of each sample point
  - Aggregate all metrics for a procedure, from any context
  - Attribute costs to loop nests and lines within a procedure
  - Example: assess the overall memory hierarchy performance within a critical procedure

Using HPCToolkit

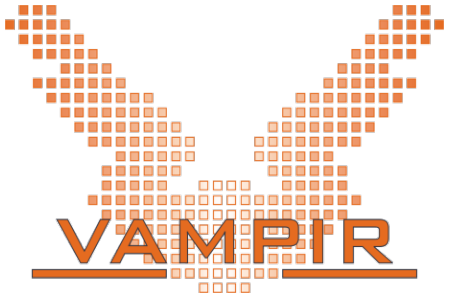
- Add hpctoolkit's bin directory to your path
- Adjust your compiler flags for your application
  - Add –g to retain line map information
- Decide what hardware counters to monitor
- Profile execution:
  - Produces one .hpcrun results file per thread
- Recover program structure
  - hpcstruct <command>
  - Produces one .hpcstruct file containing the loop structure of the binary
- Interpret profile / correlate measurements with source code
  - hpcprof [–S <hpcstruct_file>] [-M thread] [–o <output_db_name>] <hpcrun_files>
  - Creates performance database
- Use hpcviewer to visualize the performance database



Vampir

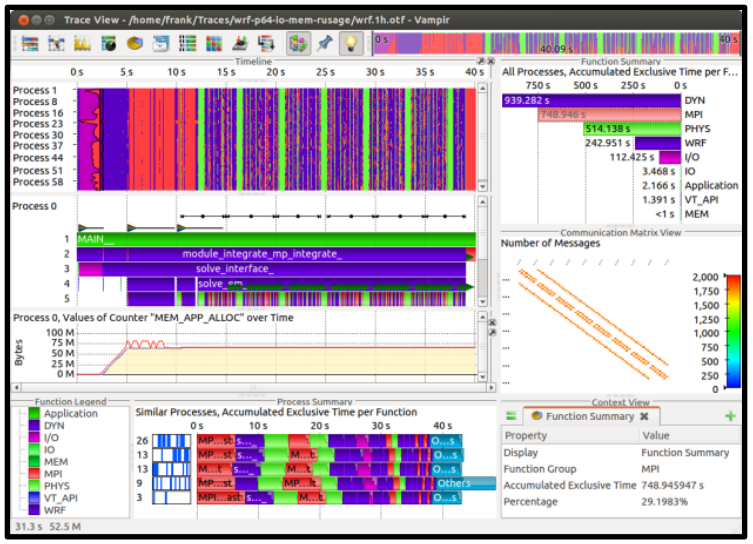
Wolfgang Nagel
ZIH, Technische Universität Dresden (Germany)

http://www.vampir.eu



Mission

- Visualization of dynamics of complex parallel processes
- Requires two components
  - Monitor/Collector (Score-P)
  - Charts/Browser (Vampir)



Typical questions that Vampir helps to answer:

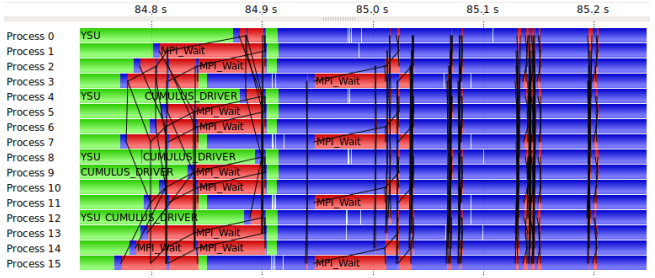
- What happens in my application execution during a given time in a given process or thread?
- How do the communication patterns of my application execute on a real system?
- Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

Event Trace Visualization with Vampir

- Alternative and supplement to automatic analysis
- Show dynamic run-time behavior graphically at any level of detail
- Provide statistics and performance metrics

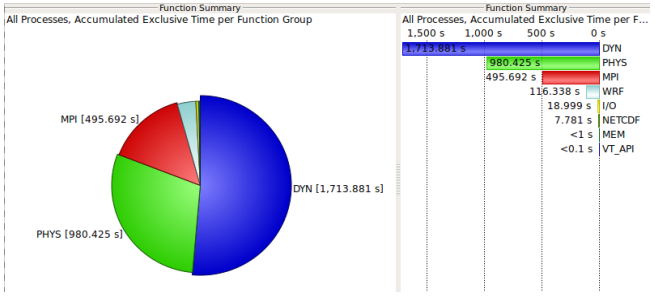
Timeline charts

– Show application activities and communication along a time axis



Summary charts

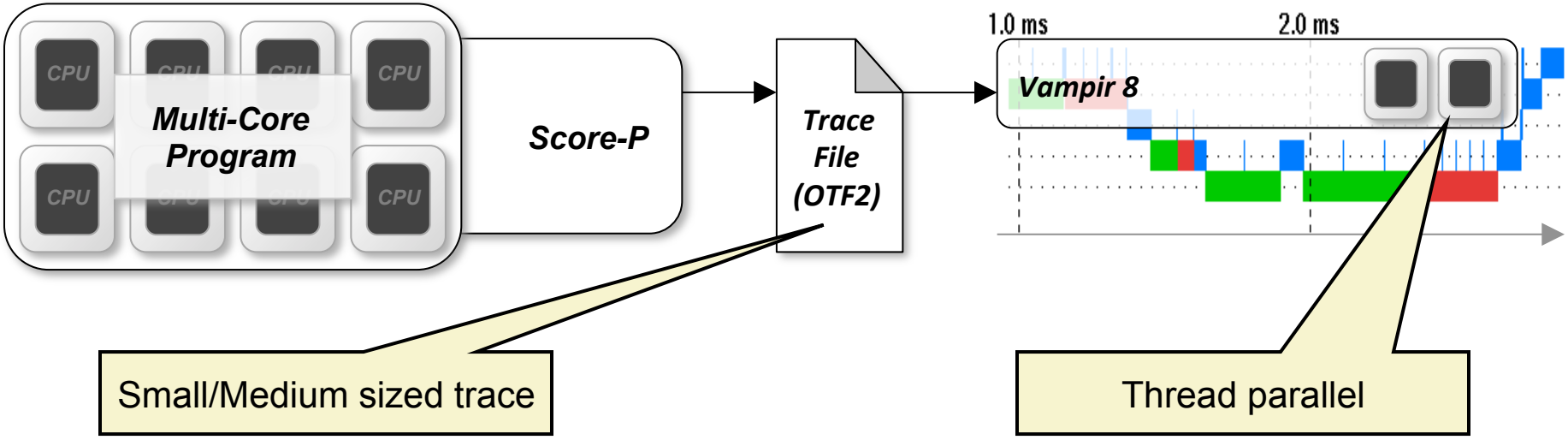
– Provide quantitative results for the currently selected time interval



Vampir – Visualization Modes (1)

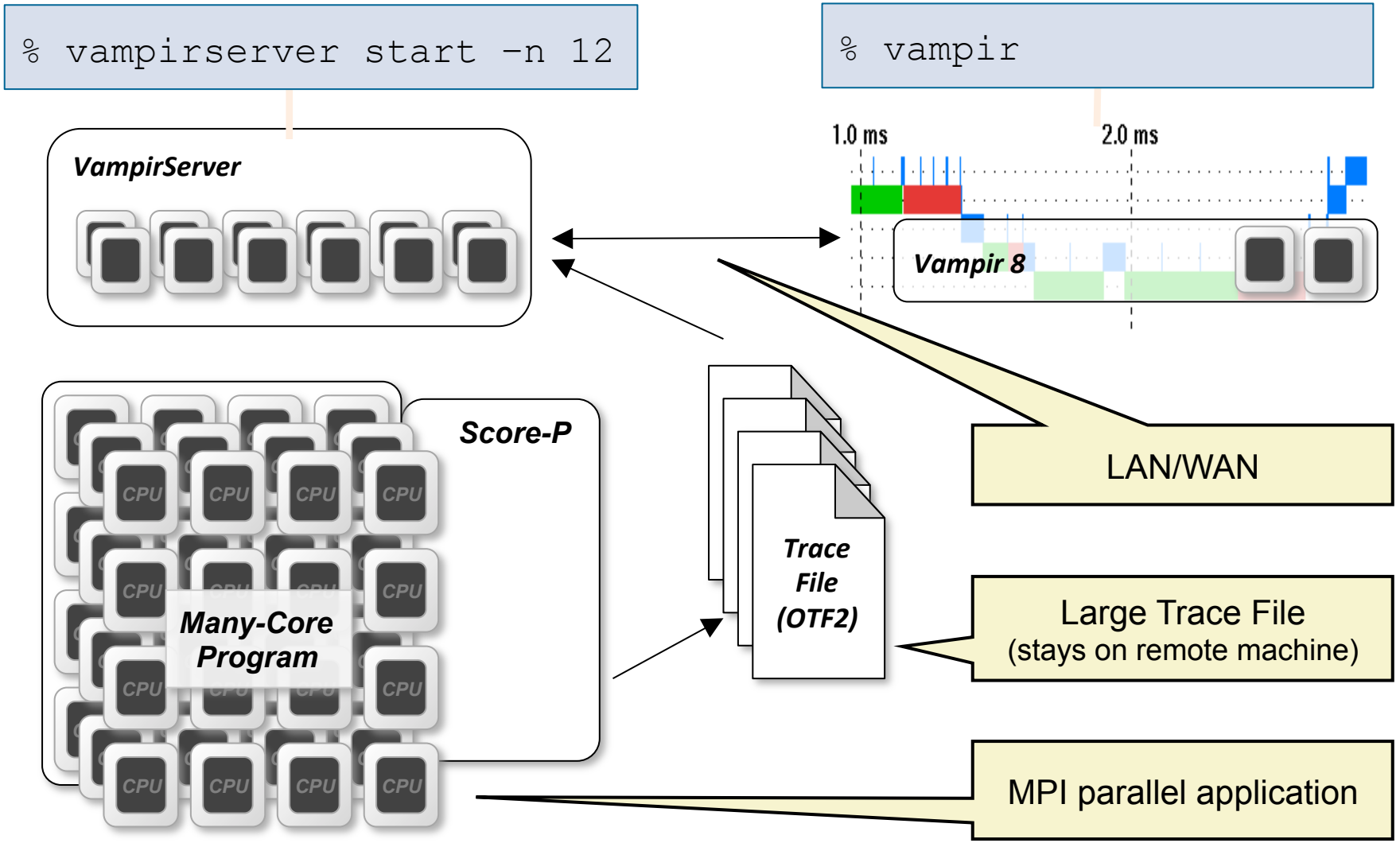
Directly on front end or local machine

```
% vampir
```



Vampir – Visualization Modes (2)

On local machine with remote VampirServer



Vampir Performance Analysis Toolset: Usage

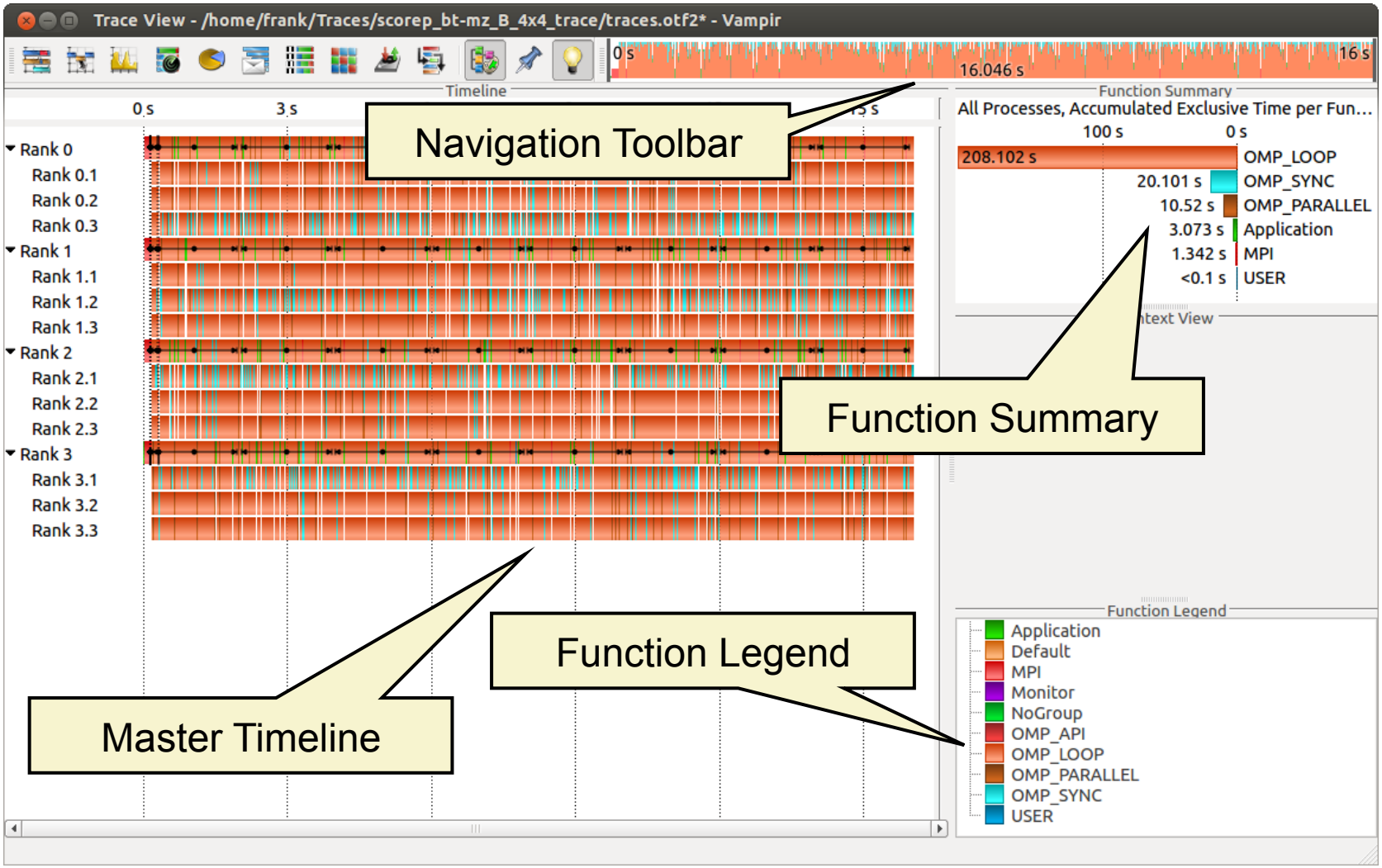
1. Instrument your application with Score-P
2. Run your application with an appropriate test set
3. Analyze your trace file with Vampir
   - Small trace files can be analyzed on your local workstation
     1. Start your local Vampir
     2. Load trace file from your local disk
   - Large trace files should be stored on the HPC file system
     1. Start VampirServer on your HPC system
     2. Start your local Vampir
     3. Connect local Vampir with the VampirServer on the HPC system
     4. Load trace file from the HPC file system

Main Displays of Vampir

- **Timeline Charts:**
  -  Master Timeline
  -  Process Timeline
  -  Counter Data Timeline
  -  Performance Radar
- **Summary Charts:**
  -  Function Summary
  -  Message Summary
  -  Process Summary
  -  Communication Matrix View

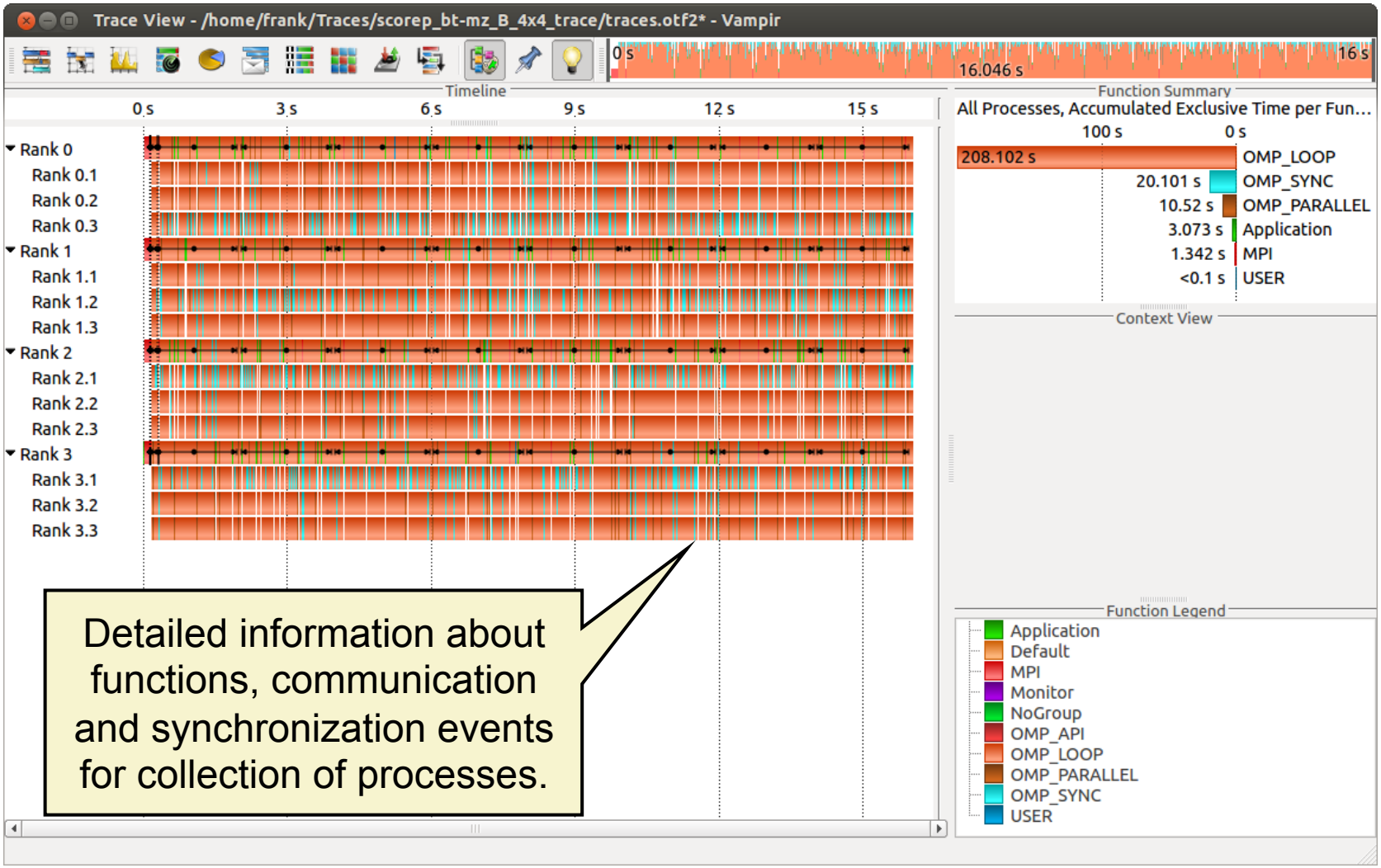
Visualization of the NPB-MZ-MPI / BT trace

```
% vampir scorep_bt-mz_B_4x4_trace
```



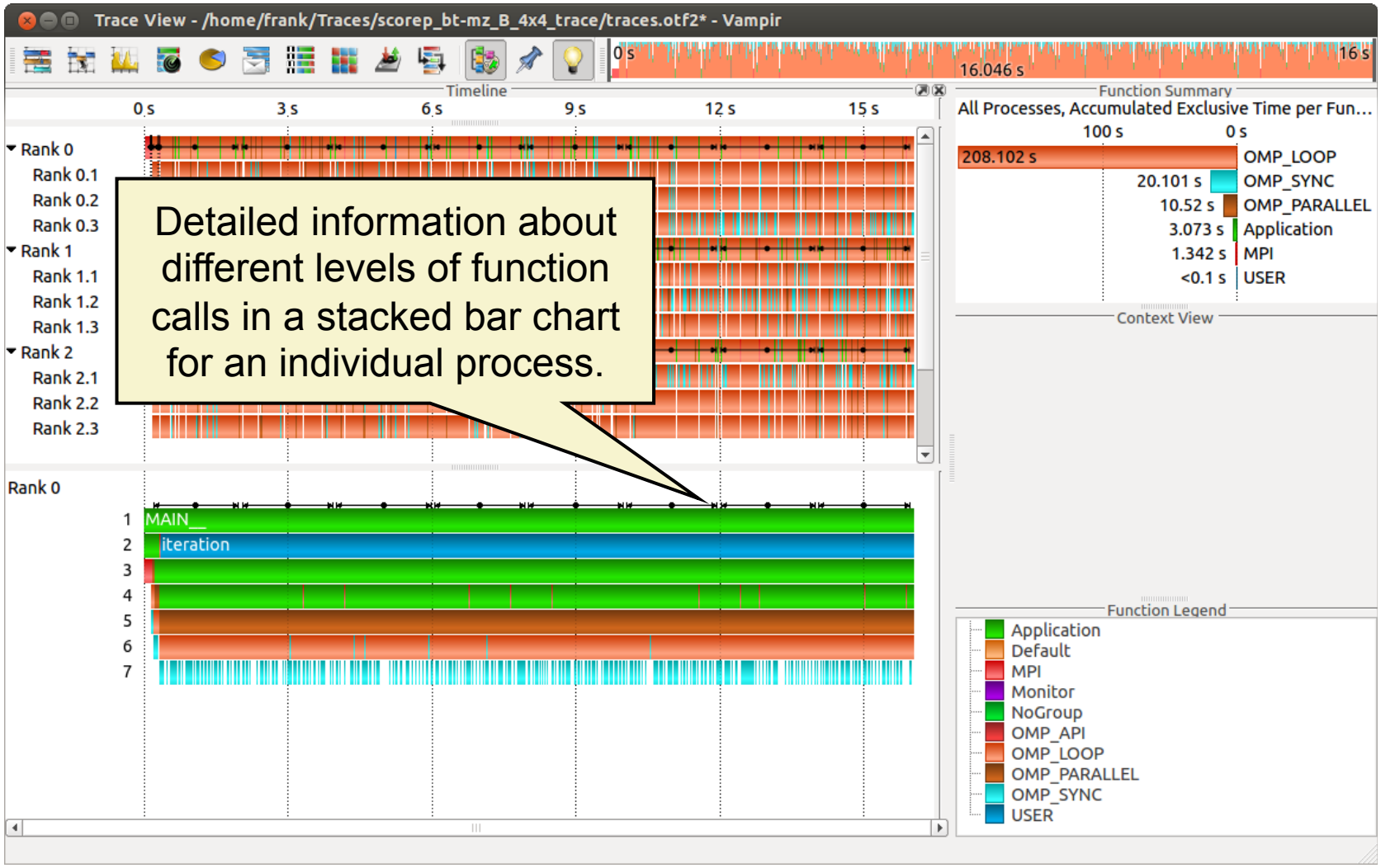
Visualization of the NPB-MZ-MPI / BT trace

Master Timeline



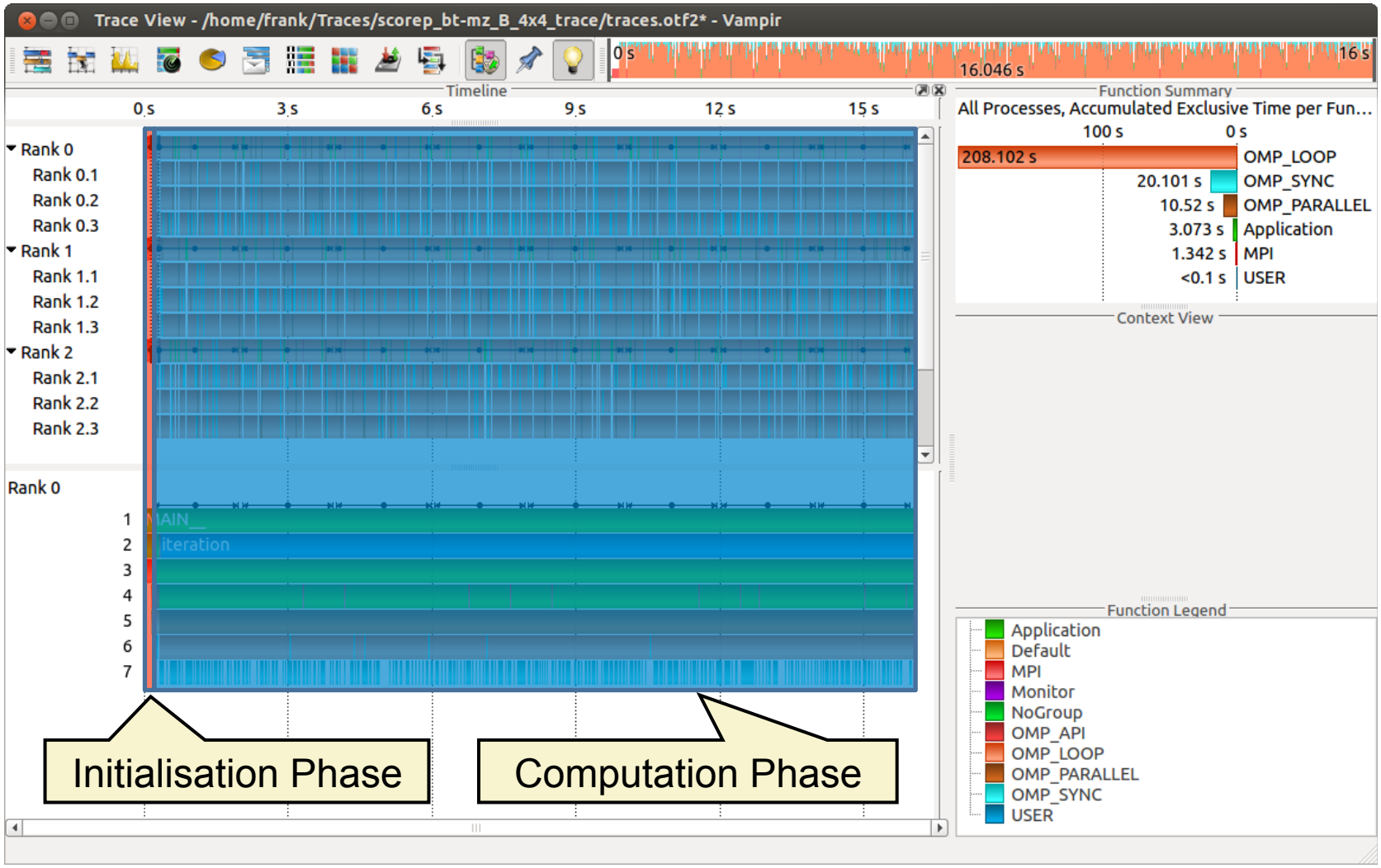
Visualization of the NPB-MZ-MPI / BT trace

Process Timeline



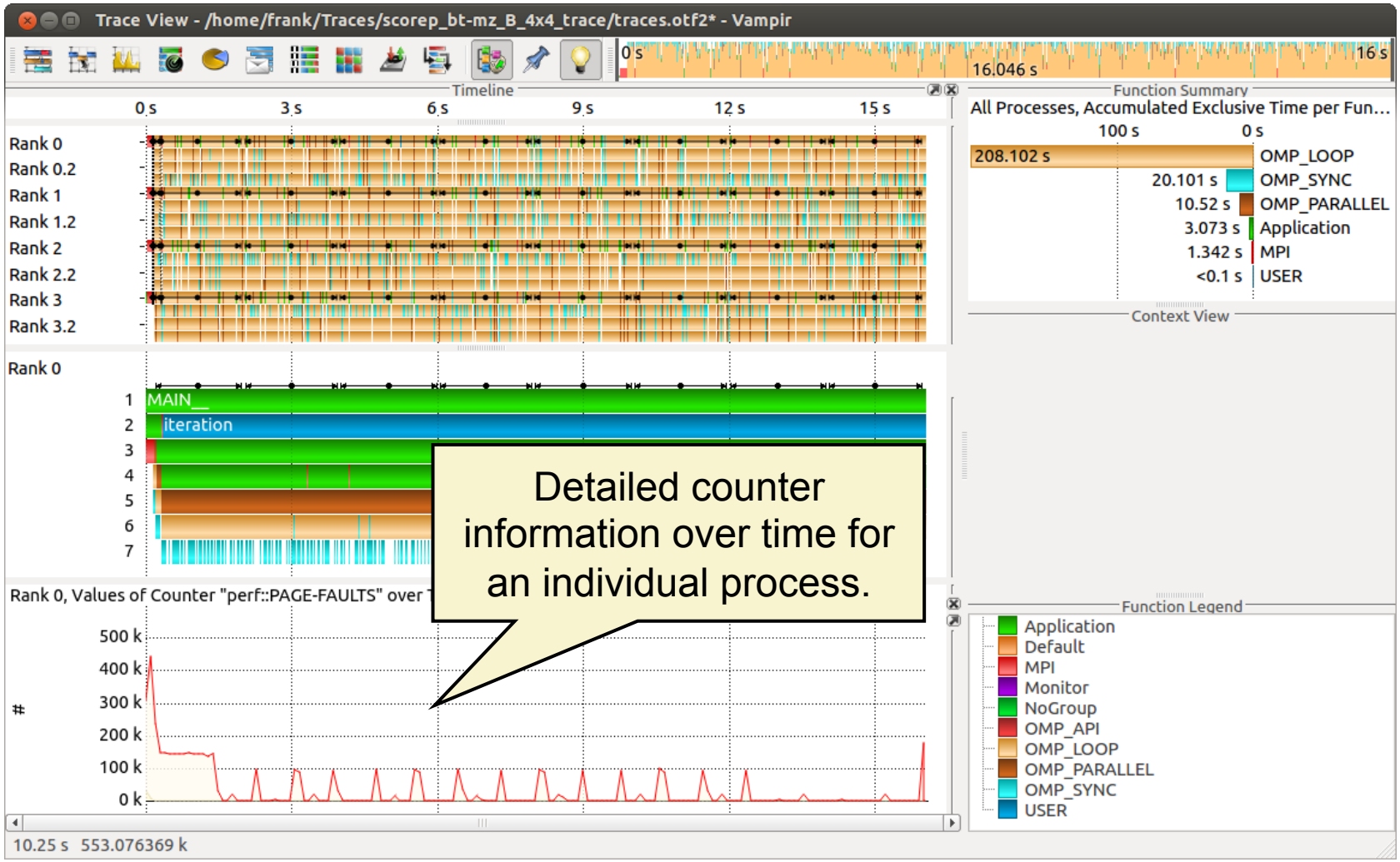
Visualization of the NPB-MZ-MPI / BT trace

Typical program phases



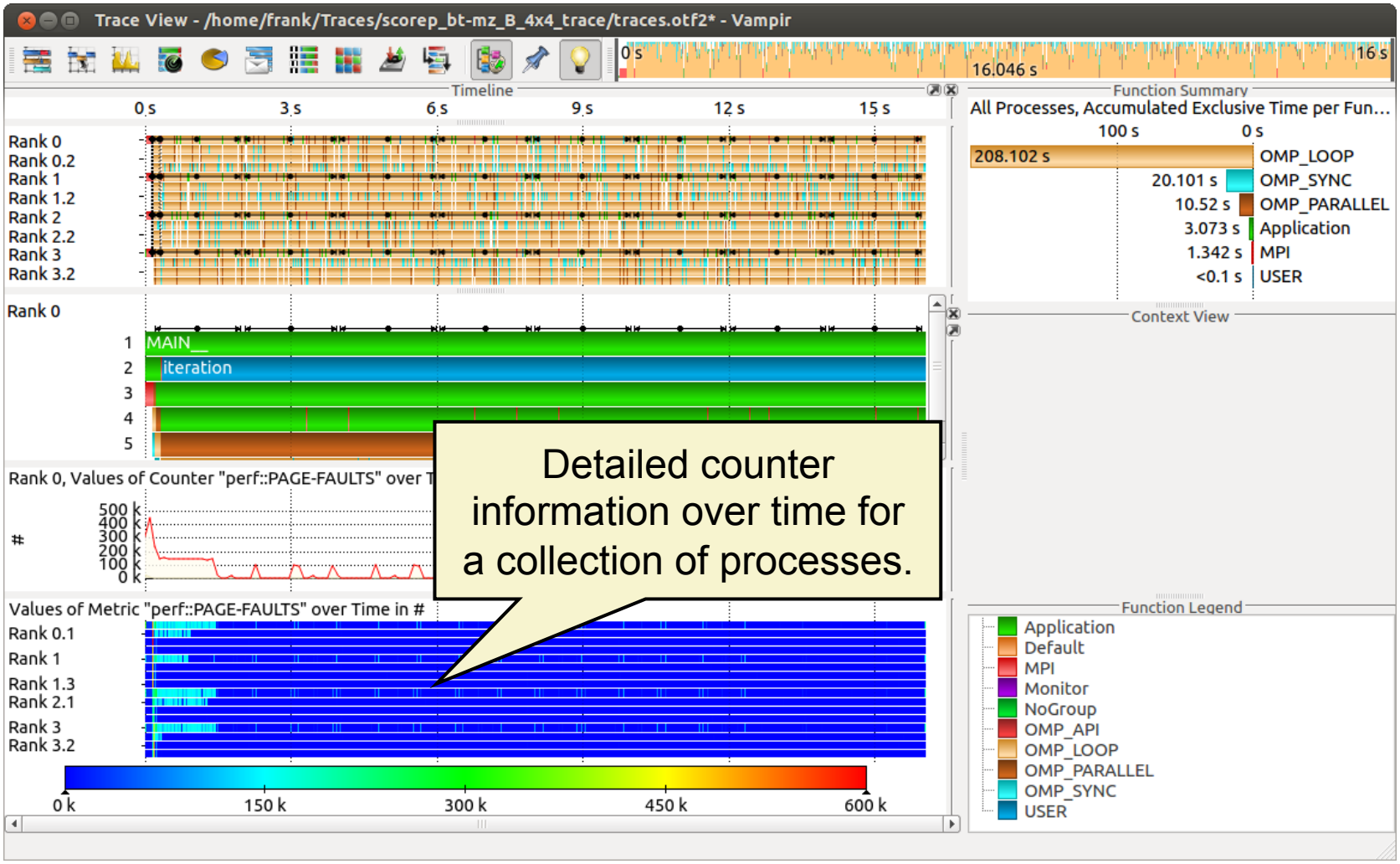
Visualization of the NPB-MZ-MPI / BT trace

Counter Data Timeline



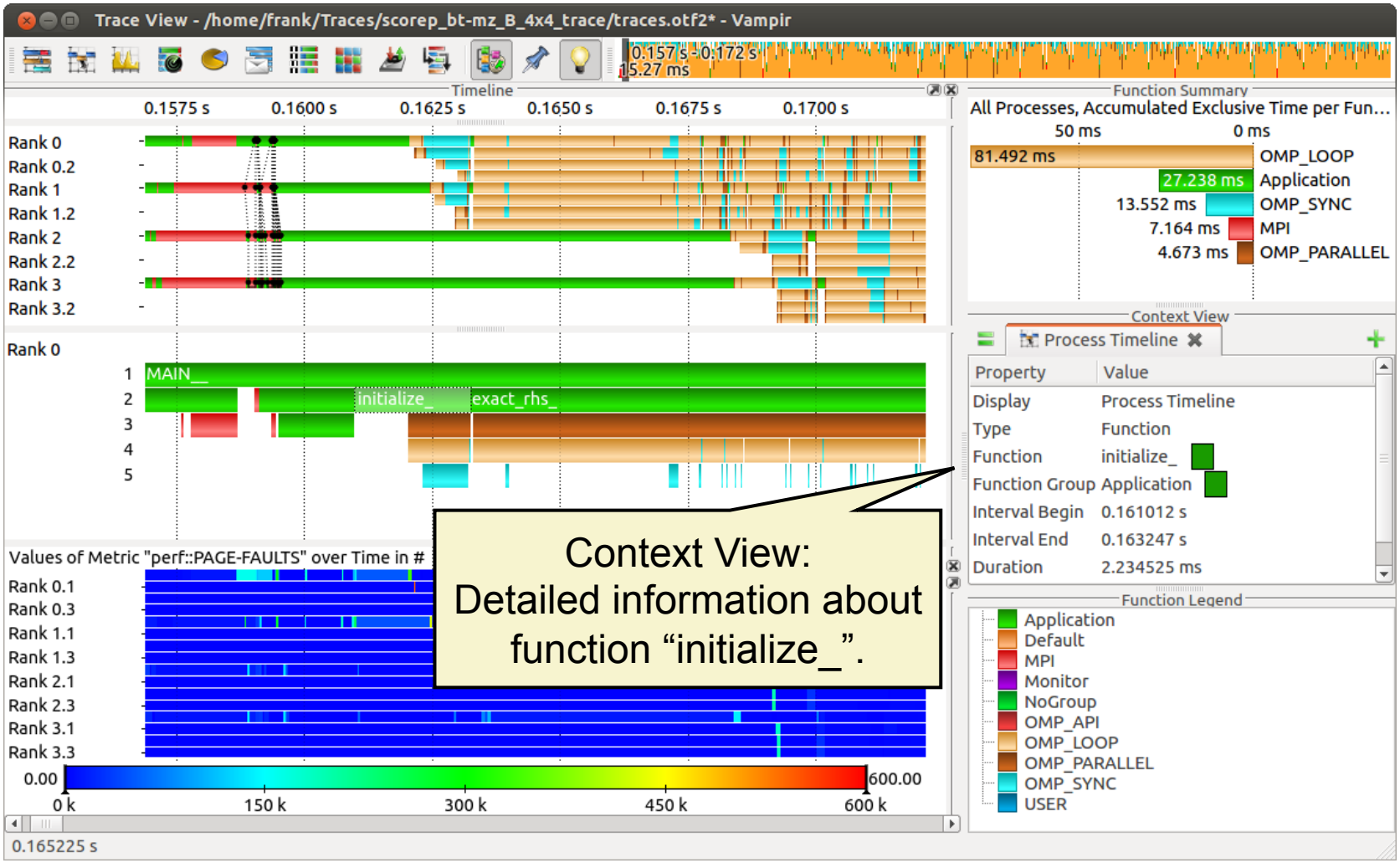
Visualization of the NPB-MZ-MPI / BT trace

Performance Radar



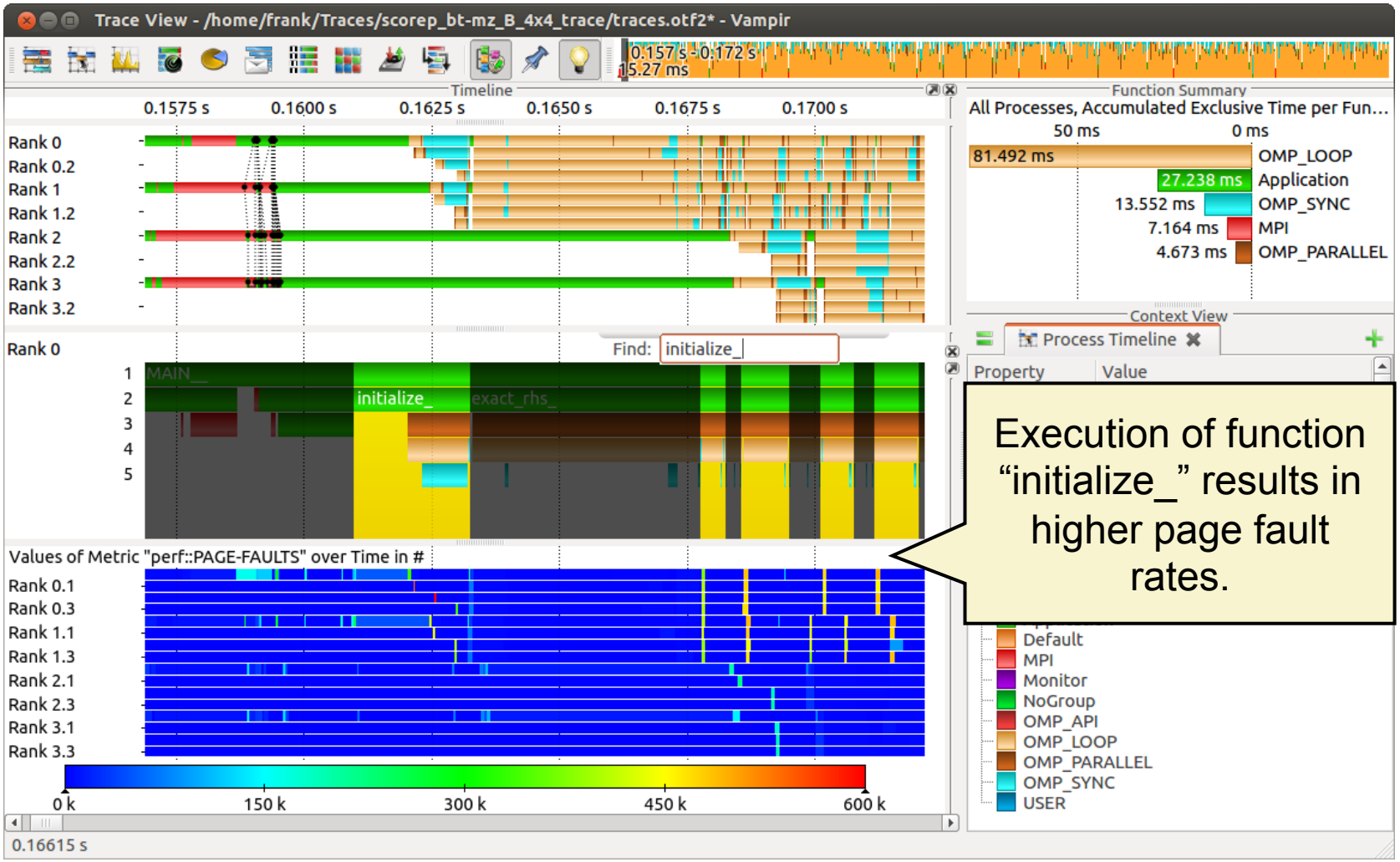
Visualization of the NPB-MZ-MPI / BT trace

Zoom in: Inititialisation Phase



Visualization of the NPB-MZ-MPI / BT trace

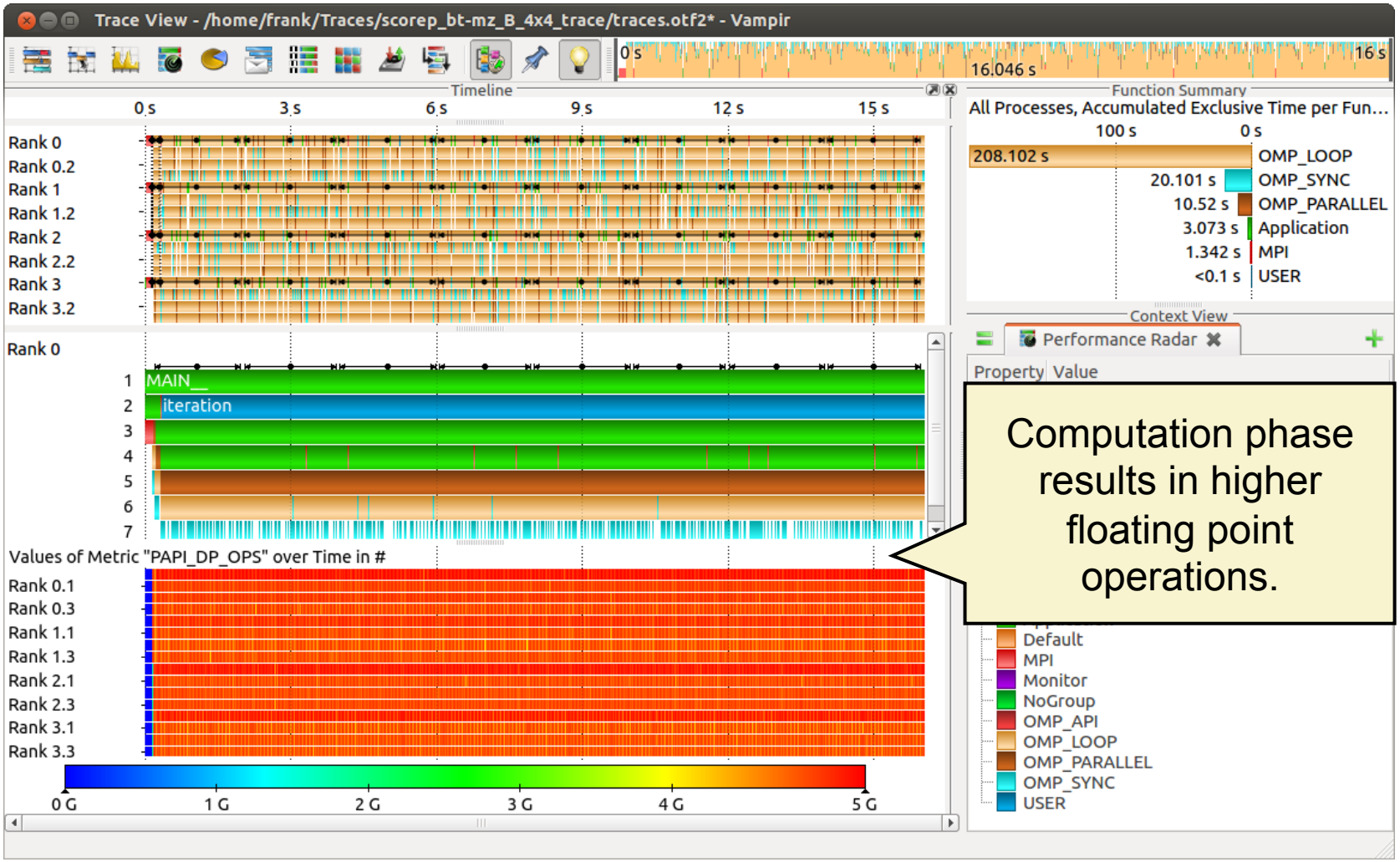
Feature: Find Function



Execution of function "initialize_" results in higher page fault rates.

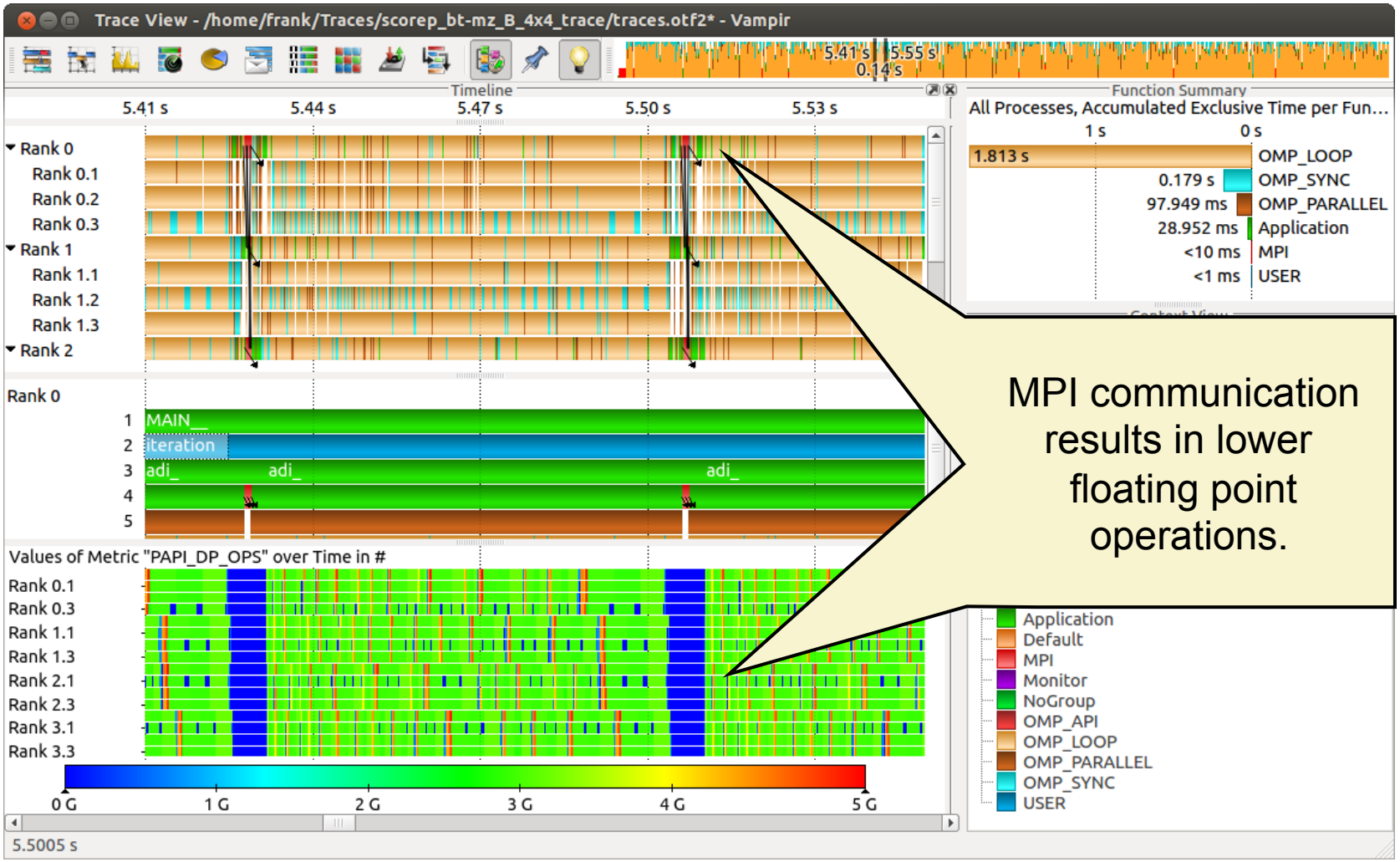
Visualization of the NPB-MZ-MPI / BT trace

Computation Phase



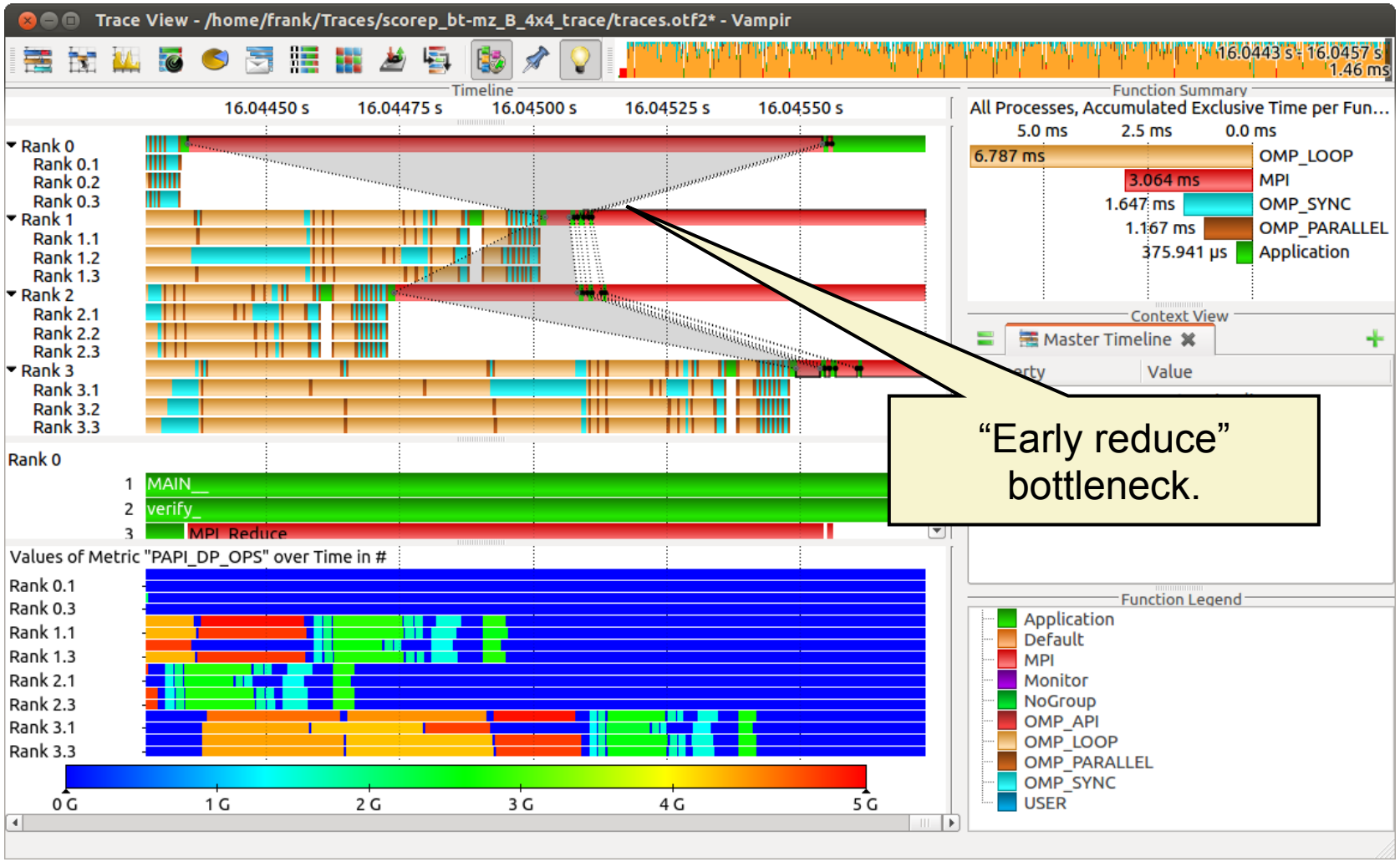
Visualization of the NPB-MZ-MPI / BT trace

Zoom in: Computation Phase



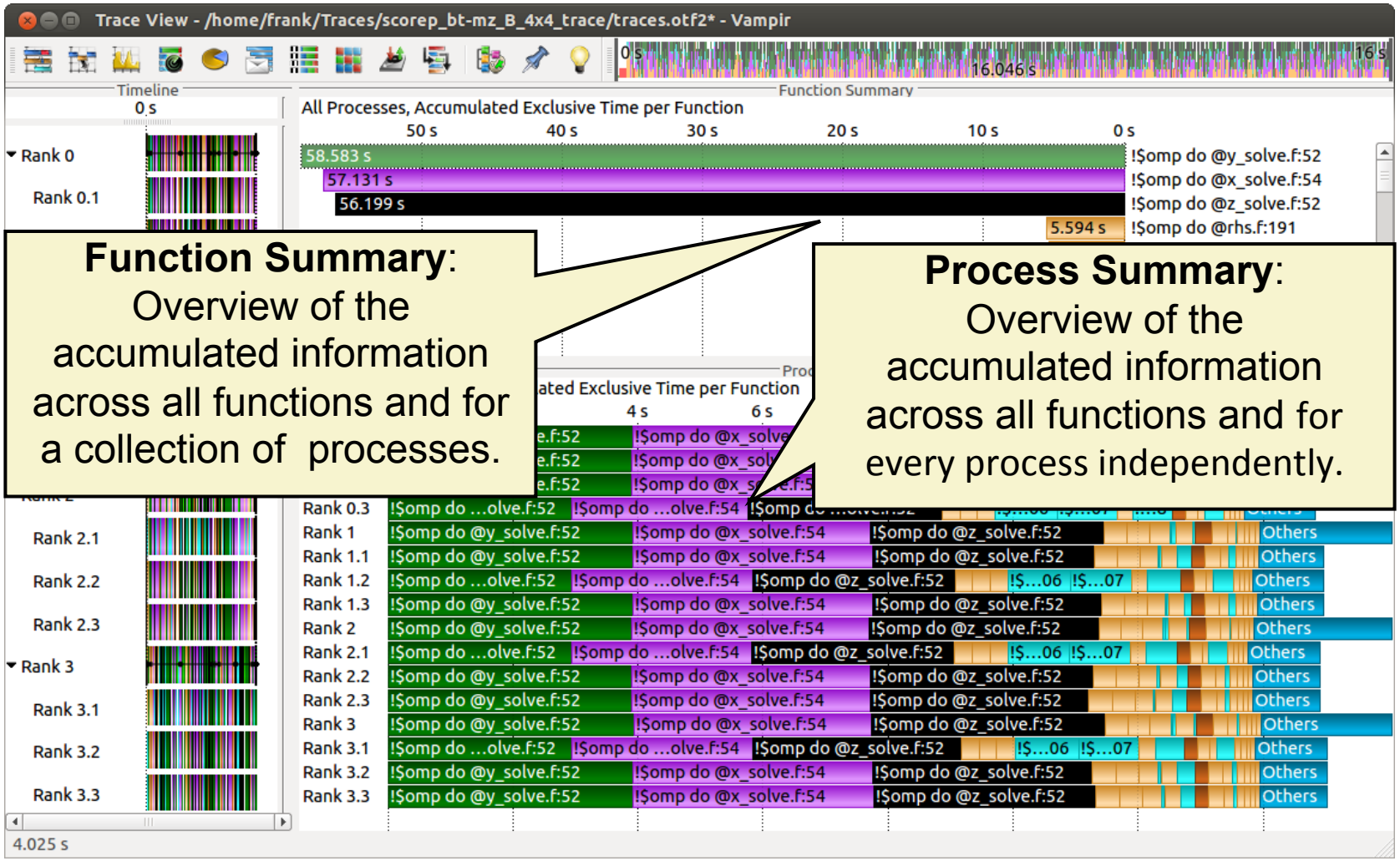
Visualization of the NPB-MZ-MPI / BT trace

Zoom in: Finalisation Phase



Visualization of the NPB-MZ-MPI / BT trace

Process Summary

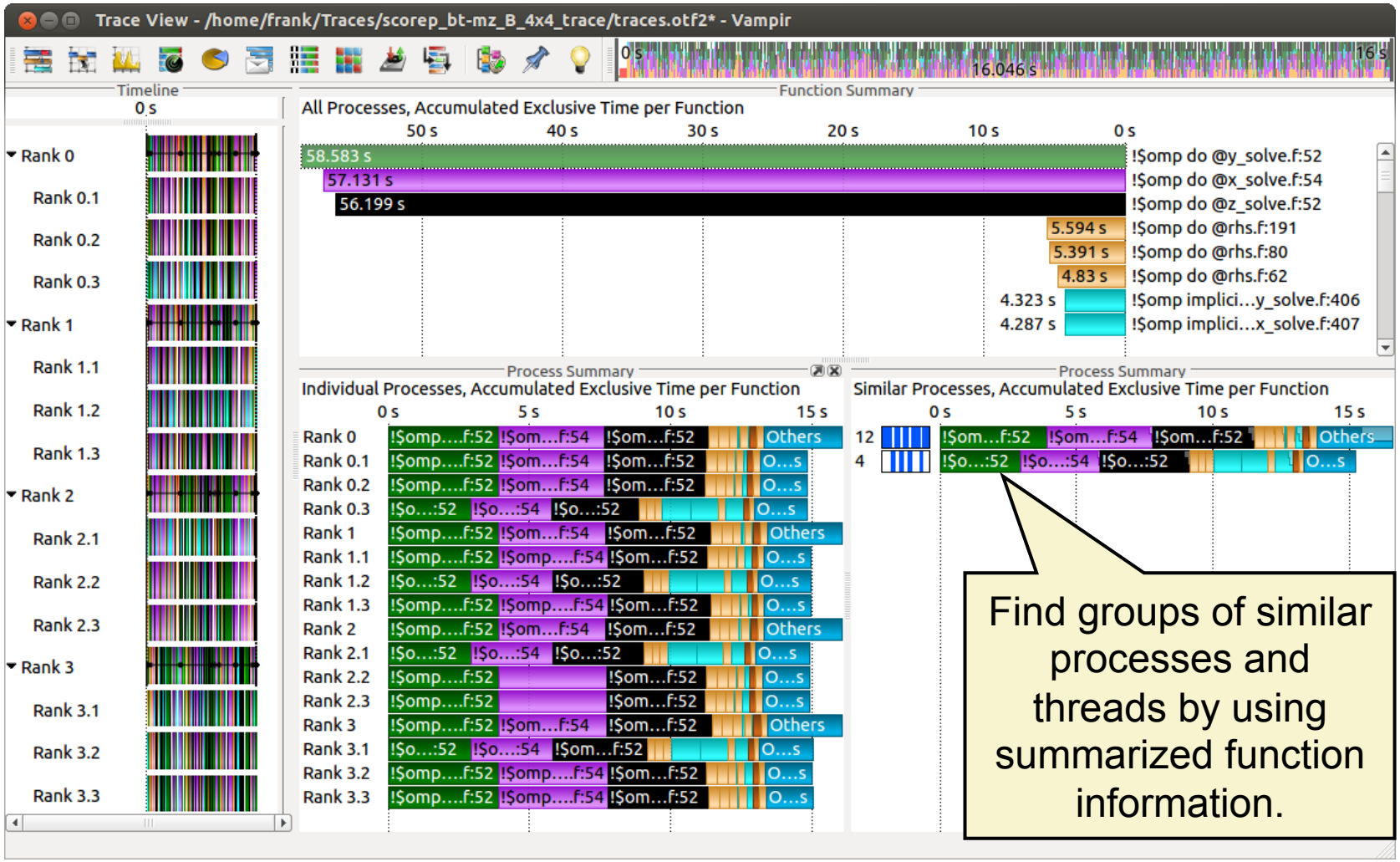


Function Summary: Overview of the accumulated information across all functions and for a collection of processes.

Process Summary: Overview of the accumulated information across all functions and for every process independently.

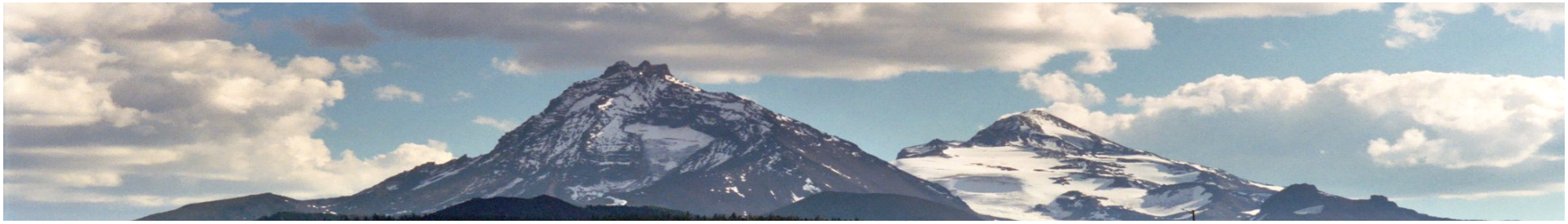
Visualization of the NPB-MZ-MPI / BT trace

Process Summary



Vampir Summary

- Vampir & VampirServer
  - Interactive trace visualization and analysis
  - Intuitive browsing and zooming
  - Scalable to large trace data sizes (20 TByte)
  - Scalable to high parallelism (200000 processes)
- Vampir for Linux, Windows and Mac OS X
- Vampir does neither solve your problems automatically nor point you directly at them
- Rather it gives you FULL insight into the execution of your application



Scalasca

Bernd Mohr and Felix Wolf

Jülich Supercomputing Centre (Germany)
German Research School for Simulation Sciences

http://www.scalasca.org

scalasca

- Scalable parallel performance-analysis toolset
  - Focus on communication and synchronization
- Integrated performance analysis process
  - Callpath profiling
    - performance overview on callpath level
  - Event tracing
    - in-depth study of application behavior
- Supported programming models
  - MPI-1, MPI-2 one-sided communication
  - OpenMP (basic features)
- Available for all major HPC platforms

Scalasca Project: Overview

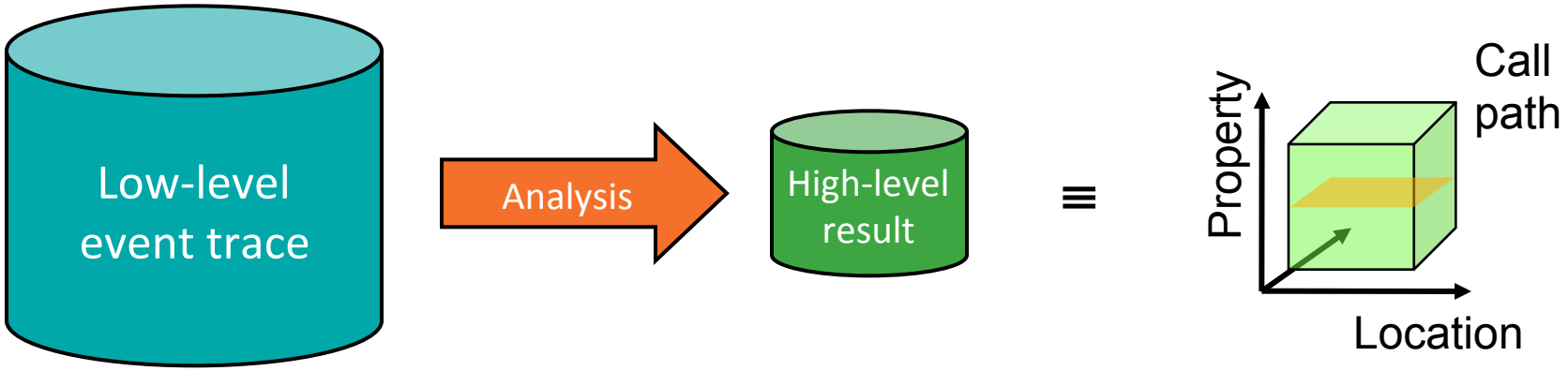
- Project started in 2006
  - Funded by Helmholtz Initiative and Networking Fund
  - Many research projects to present
- Successor to pioneering KOJAK project (1998)
  - Automatic pattern-based trace analysis
- Now joint development of
  - Jülich Supercomputing Centre
  - German Research School for Simulation Sciences

Scalasca Project: Objective

- Development of a *scalable* performance analysis toolset for most popular parallel programming paradigms
- Specifically targeting *large-scale* parallel applications
  - 100,000 – 1,000,000 processes / thread
  - IBM BlueGene or Cray XT systems
- Latest release:
  - Scalasca v2.0 with Score-P support (August 2013)

Scalasca: Automatic Trace Analysis

- Idea
  - Automatic search for patterns of inefficient behavior
  - Classification of behavior and quantification of significance

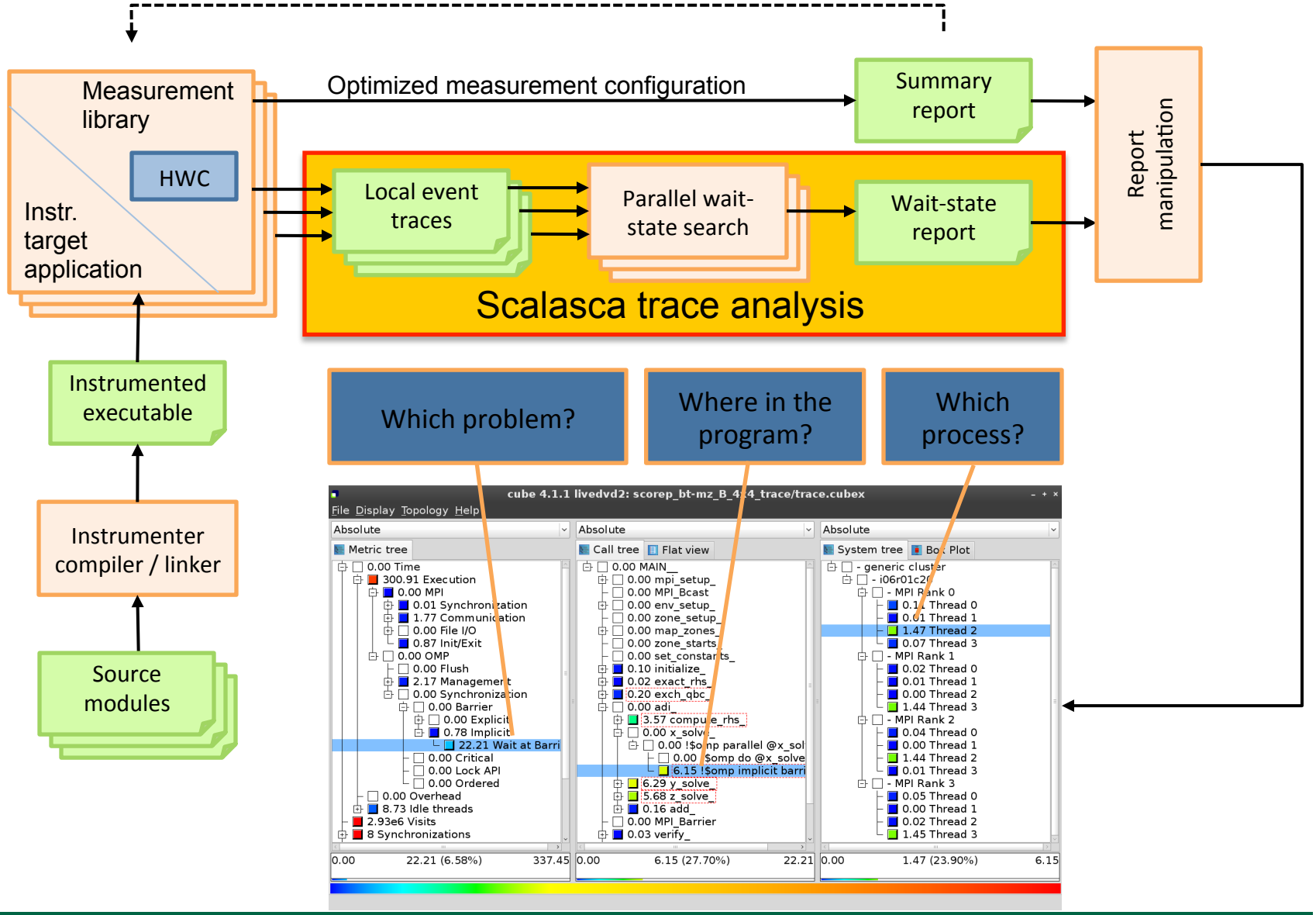


  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Parallel replay analysis online

Scalasca 2.0 Features

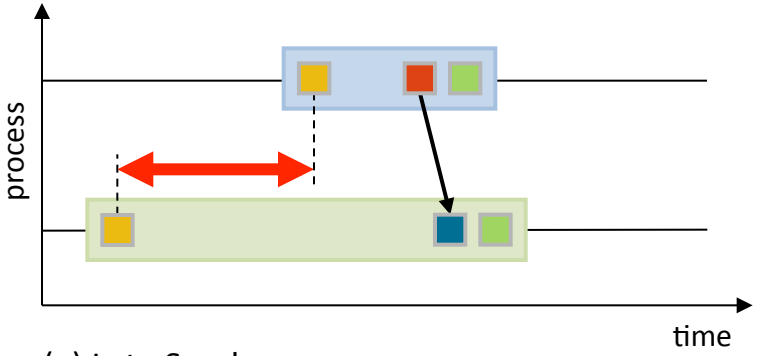
- Open source, New BSD license
- Fairly portable
  - IBM Blue Gene, IBM SP and blade clusters, Cray XT, SGI Altix, Solaris & Linux clusters, ...
- Uses Score-P instrumenter and measurement libraries
  - Scalasca 2.0 core package focuses on trace-based analyses
  - Supports common data formats
    - reads event traces in OTF2 format
    - writes analysis reports in CUBE4 format
- Current limitations:
  - No support for nested OpenMP parallelism and tasking
  - Unable to handle OTF2 traces containing CUDA events

Scalasca Workflow

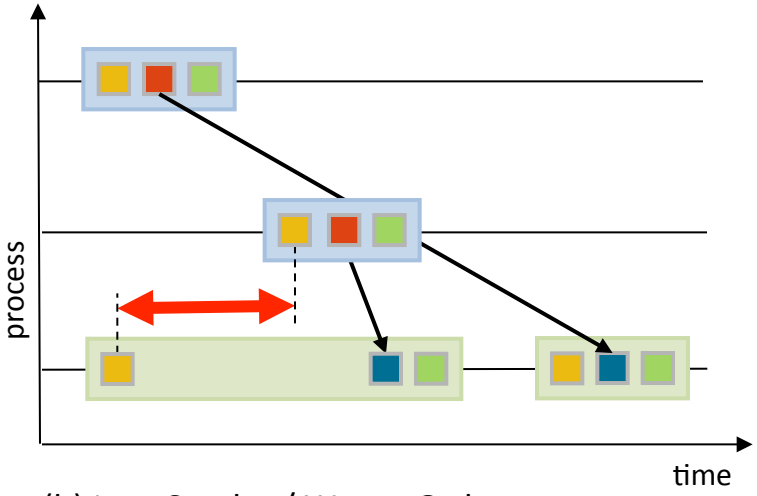


Wait-state Analysis

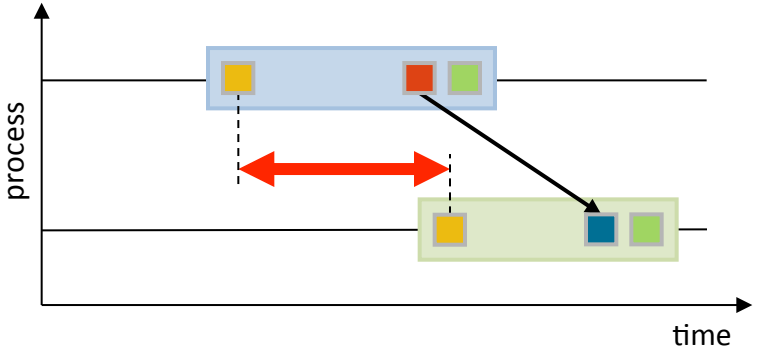
- Classification
- Quantification



(a) Late Sender



(b) Late Sender / Wrong Order

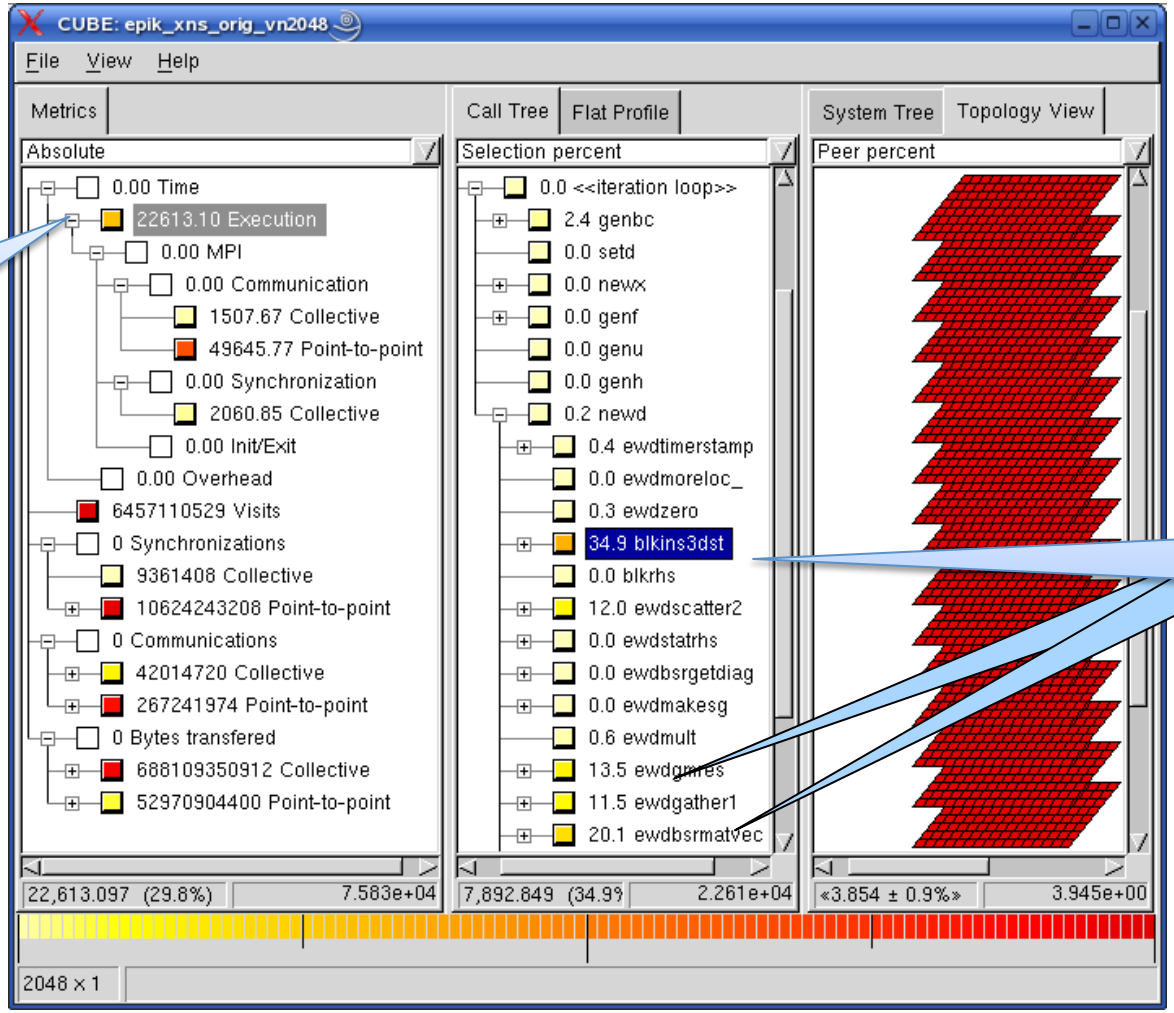


(c) Late Receiver

Callpath Profile: Computation

Execution time excl. MPI comm

Just 30% of simulation

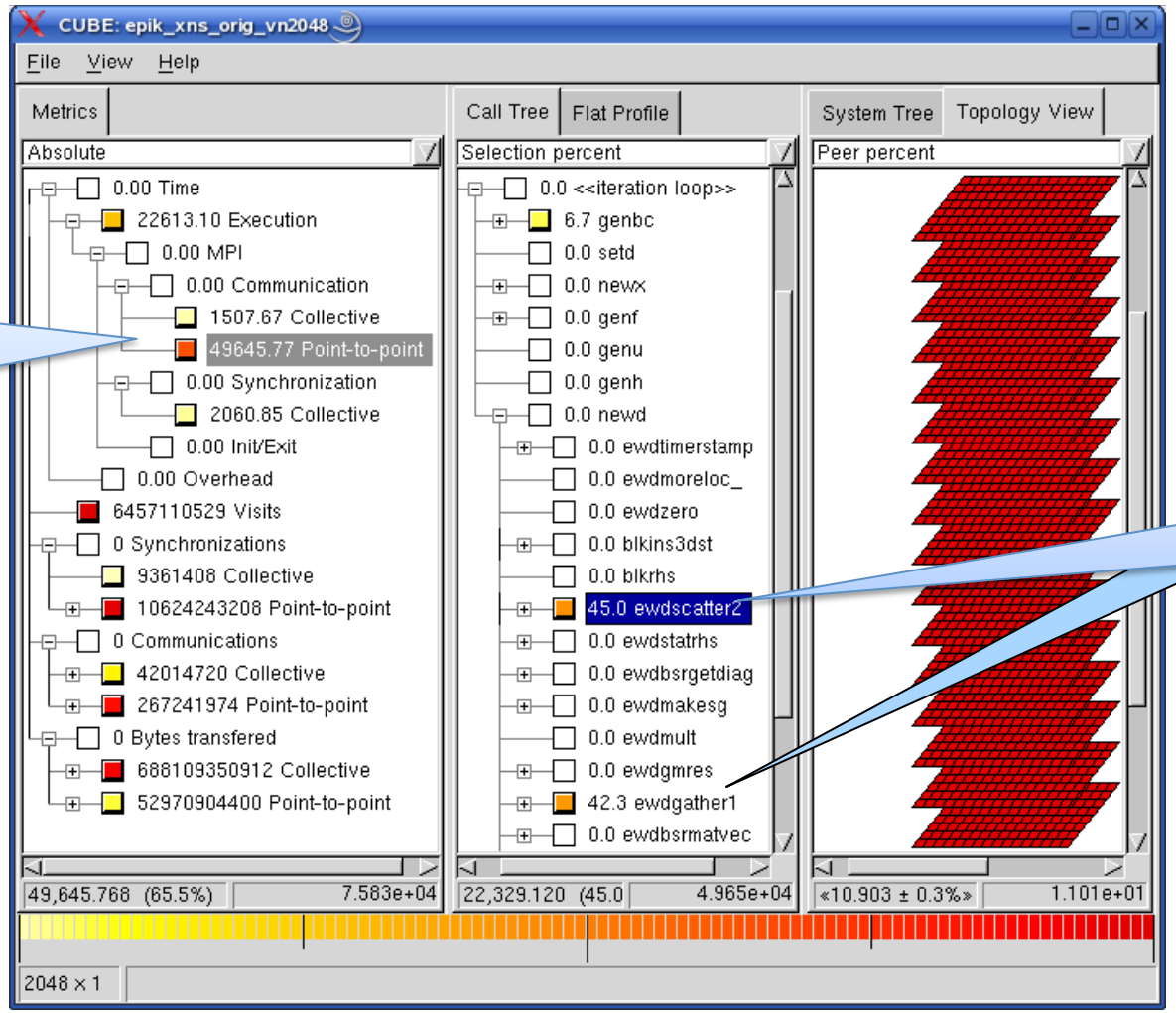


Widely spread in code

Callpath Profile: P2P Messaging

MPI point-to-point communic-ation time

P2P comm 66% of simulation

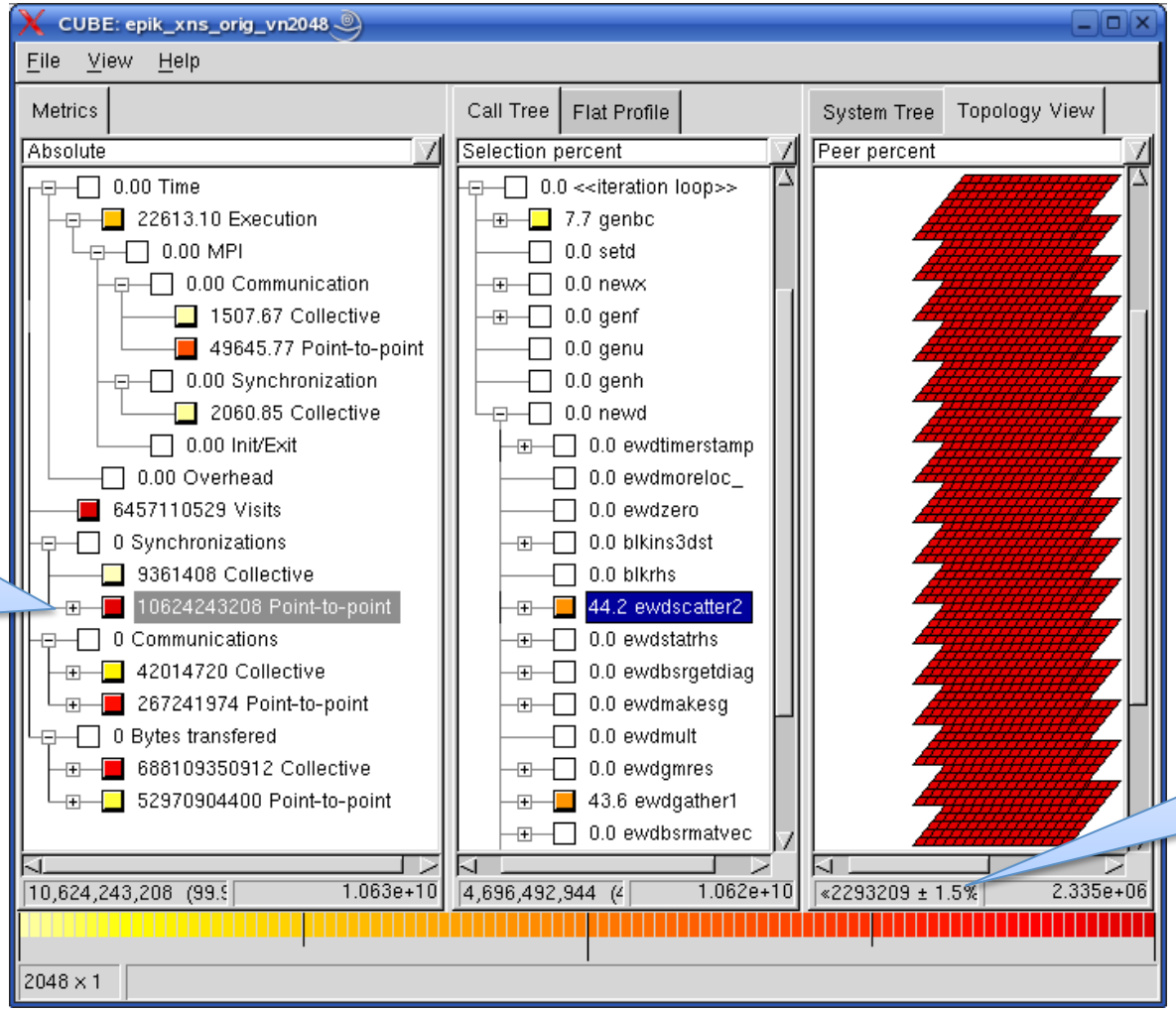


Primarily in scatter & gather

Callpath Profile: P2P Synchronization

Point-to-point msgs w/o data

Masses of P2P sync. operations

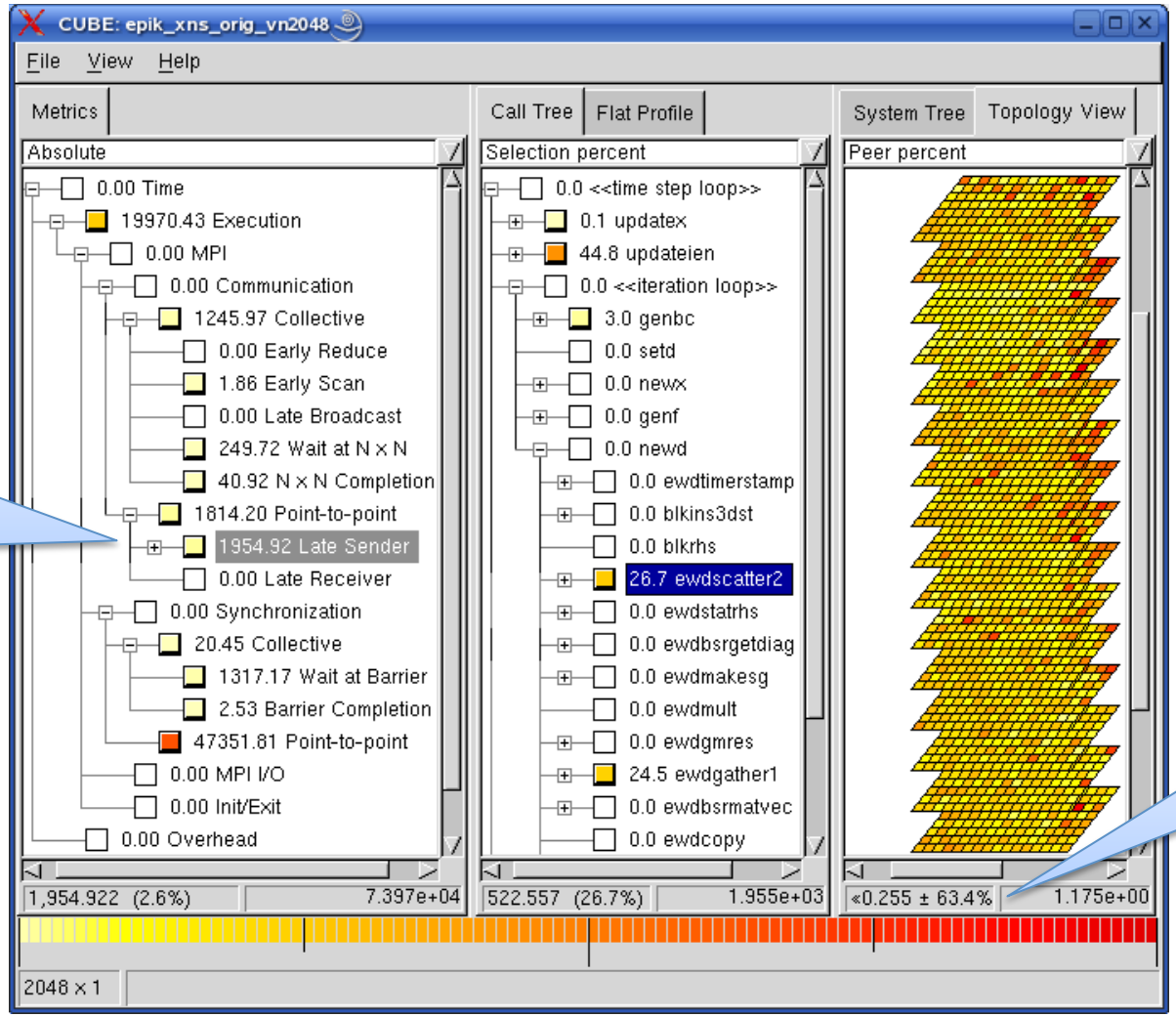


Processes all equally responsible

Trace Analysis: Late Sender

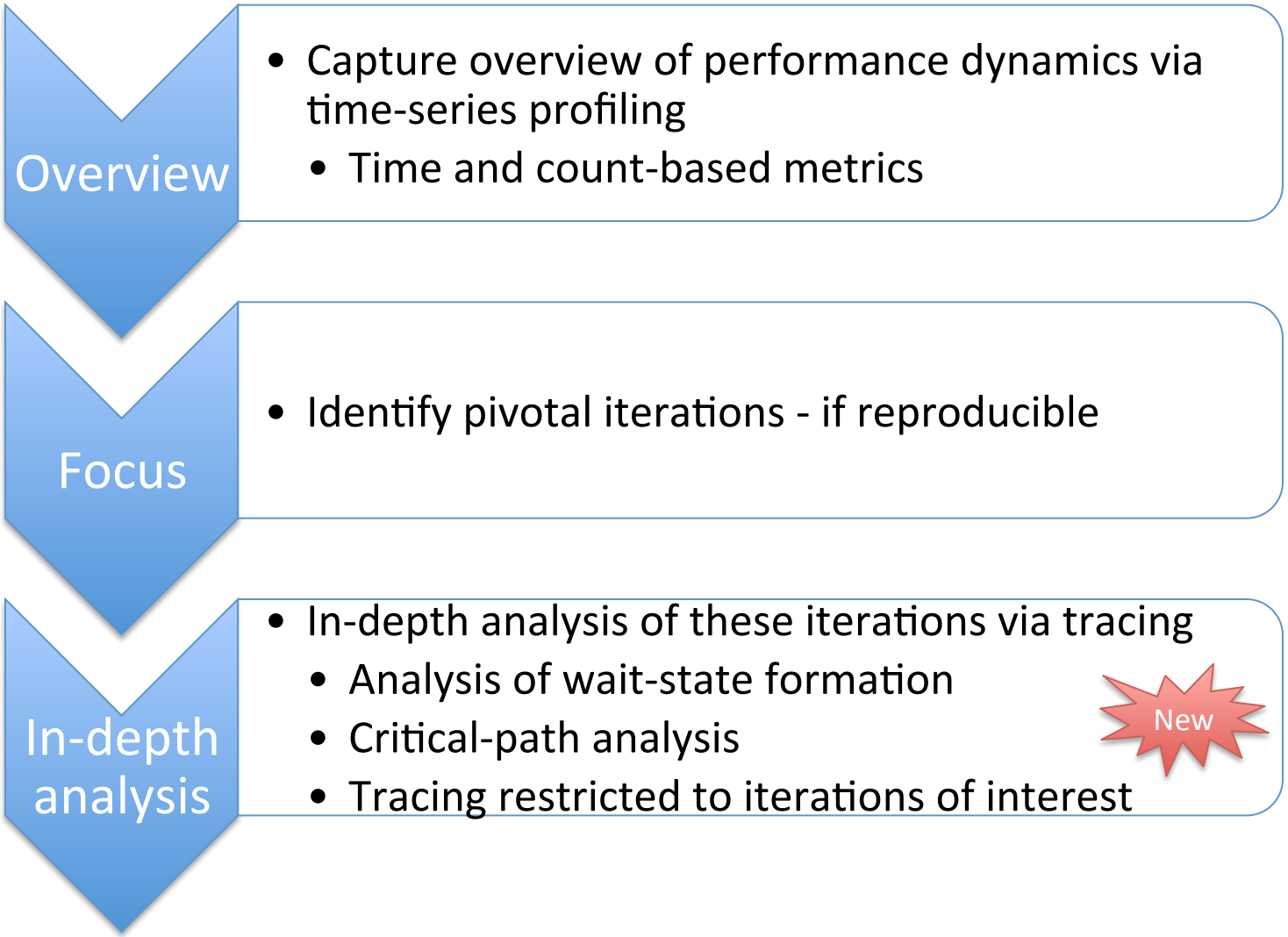
Wait time of receivers blocked for late sender

Half of the send time is waiting



Significant process imbalance

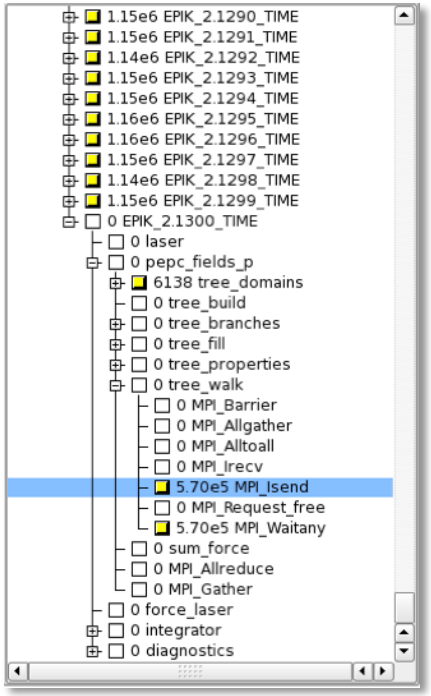
Scalasca Approach to Performance Dynamics



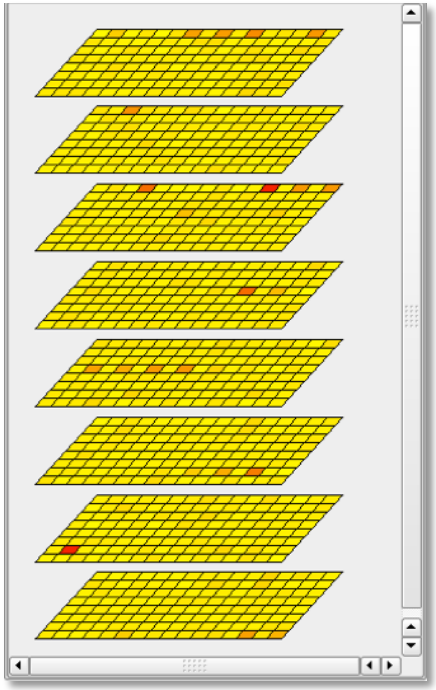
Time-series Callpath Profiling

- Instrument main loop to distinguish individual iterations
  - Complete call tree with multiple metrics recorded for each iteration
  - Challenge: storage requirements proportional to #iterations

```
#include "epik_user.h"

void initialize() {}
void read_input() {}
void do_work() {}
void do_additional_work() {}
void finish_iteration() {}
void write_output() {}

int main() {
  int iter;
  PHASE_REGISTER(iter,"ITER");
  int t;
  initialize();
  read_input();
  for(t=0; t<5; t++) {
    PHASE_START(iter);
    do_work();
    do_additional_work();
    finish_iteration();
    PHASE_END(iter);
  }
  write_output();

  return 0;
}
```



Call tree



Process topology