Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Performance Analysis

An Introduction

July 09, 2014 | Florian Janetzko

Acknowledgements

Slides taken partially from the

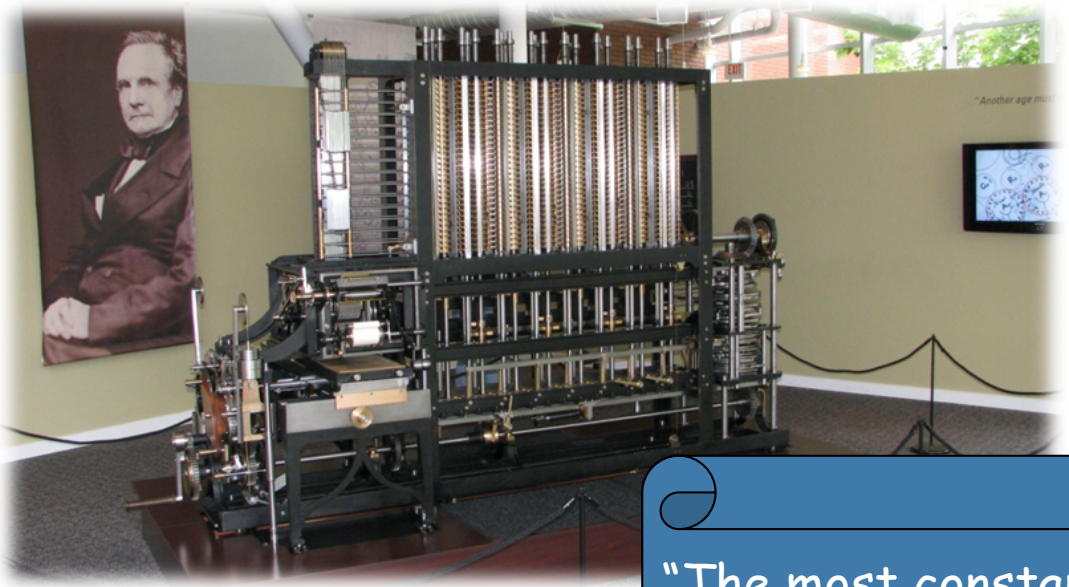
Virtual Institute – High Productivity Supercomputing (VI-HPS)

http://www.vi-hps.org

Outline

- Introduction
  - Hardware development
  - Tuning basics
- Code development
- Performance analysis and tuning
- Summary

Performance: an old problem



Difference Engine

"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

Charles Babbage
1791 – 1871

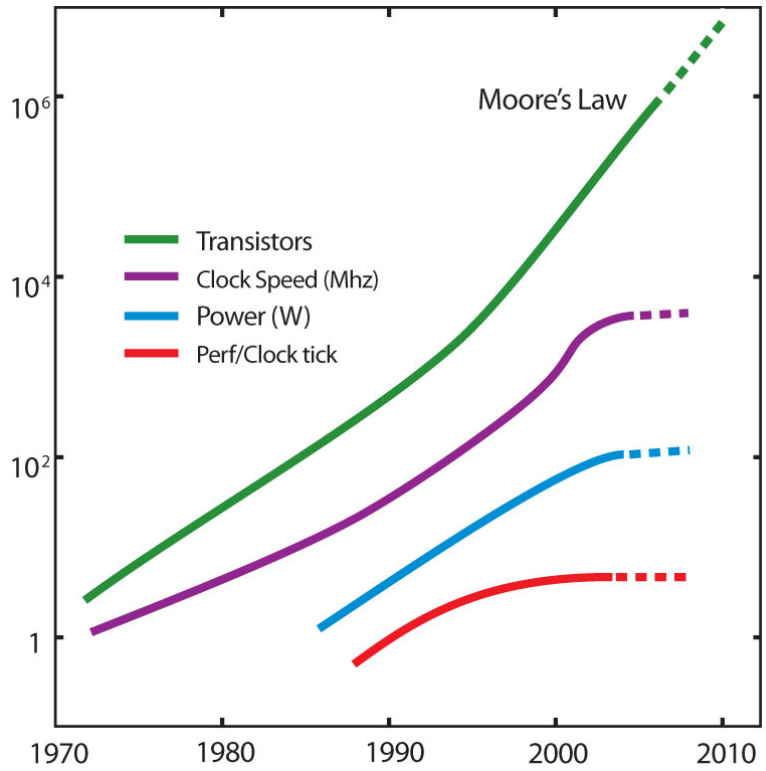
HPC hardware development

Moore's law is still in charge, but

- Clock rates no longer increase
- Performance gains only through increased parallelism

Optimizations of applications more difficult

- Increasing application complexity
  - Multi-physics
  - Multi-scale
- Increasing machine complexity
  - Hierarchical networks / memory
  - More CPUs / multi-core



➢ Challenges for HPC applications!

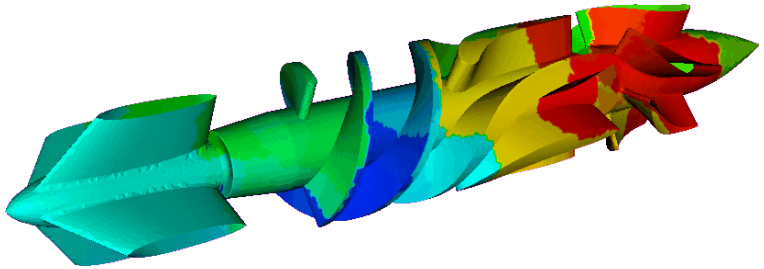
Example: XNS

CFD simulation of unsteady flows

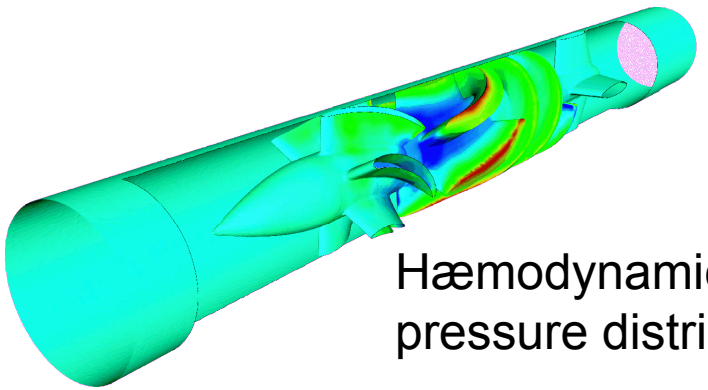
- Developed by CATS / RWTH Aachen
- Exploits finite-element techniques, unstructured 3D meshes, iterative solution strategies

MPI parallel version

- >40,000 lines of Fortran & C
- DeBakey blood-pump data set (3,714,611 elements)

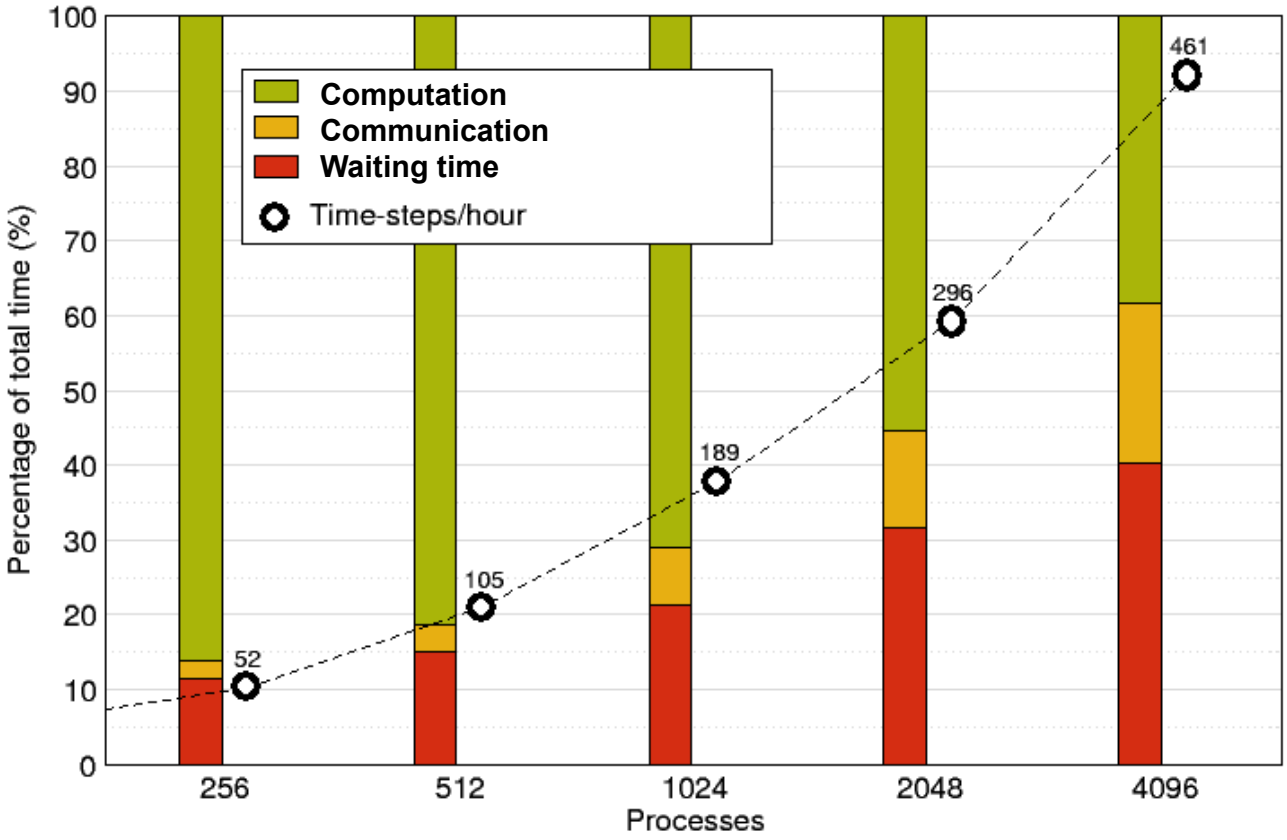


Partitioned finite-element mesh



Hæmodynamic flow pressure distribution

XNS wait-state analysis on BG/L (2007)



Tuning applications

Successful engineering is a combination of

- The right algorithms and libraries
- Compiler flags and directives
- Thinking !!!

Measurement is better than guessing

- To determine performance bottlenecks
- To compare alternatives
- To validate tuning decisions and optimizations
  - After *each* step!

Code development – “Golden rules”

Programmer’s rule of code development:

Nobody cares how fast you can compute a wrong answer!

Performance analyst’s deduction:

It's easier to optimize a slow correct program than to debug a fast incorrect one!

Outline

- Introduction
- Code development
  - Code development stages and tools
    - Marmot
    - MUST
    - Thread Inspector
    - TotalView
- Performance analysis and tuning
- Summary

Code development stages

1. Programming
   - Tools: editors with syntax highlighting (e.g. vim, emacs,…), development tools (e.g. Parallel Tools Platform (PTP), syntax checker (e.g. forcheck)

2. Debugging
   - Tools: write/printf statements, classical debuggers (TotalView, DDT, GDB, …), MARMOT, MUST (for MPI codes), Intel® Inspector (for OpenMP codes)

3. Performance
   - Tools: performance analysis tools (Scalasca, Vampir, TAU, …)

Code development – Programming

Code development – FORCHECK

Selected Features

- Verification of conformance to all levels of Fortran standard
- Full static analysis of separate program units
- Reverse engineering tool
- Generates call trees, callby trees, use trees and module dependencies
- Provides an IDE

http://www.forcheck.nl

Code development – Syntax highlighting

```
      CHECK = '0123456789+-DE.'
      RNUM = 0.0
      INUM = 0
C
C     *** Check STRING for allowed characters and count dots ***
C
      DOT = 0
      DO 10 I=1,STRLEN(STRING)
        IF (INDEX(CHECK,STRING(I:I)).EQ.0) GO TO 3000
        IF (INDEX('.',STRING(I:I)).EQ.1) DOT = DOT + 1
   10 CONTINUE
      IF (DOT.EQ.0) GO TO 1000
      IF (DOT.EQ.1) GO TO 2000
      IF (DOT.GT.1) GO TO 3000
C
C     *** Try to read integer value fr
C
 1000 CONTINUE
      READ (STRING,*,ERR=2000) INUM
```

Syntax highlighting

```
      CHECK = '0123456789+-DE.'
      RNUM = 0.0
      INUM = 0
C
C     *** Check STRING for allowed characters and count dots ***
C
      DOT = 0
      DO 10 I=1,STRLEN(STRING)
        IF (INDEX(CHECK,STRING(I:I)).EQ.0) GO TO 3000
        IF (INDEX('.',STRING(I:I)).EQ.1) DOT = DOT + 1
   10 CONTINUE
      IF (DOT.EQ.0) GO TO 1000
      IF (DOT.EQ.1) GO TO 2000
      IF (DOT.GT.1) GO TO 3000
C
C     *** Try to read integer value from STRING ***
C
 1000 CONTINUE
      READ (STRING,*,ERR=2000) INUM
```

Code development – PTP

What is PTP:

- Integrated development environment (IDE) for parallel application development
- Based on Eclipse
- Open Source
- Developers:
  - *IBM, U.Oregon, UTK, Heidelberg University, NCSA, UIUC, JSC, ...*

http://www.eclipse.org/

Code development – Eclipse

File Edit Source Refactor Navigate Search Project Run Run Window Help

Quick Access C/C++

Project Explor

- HelloWorld
  - Binaries
  - HelloWorld.cpp
  - HelloWorld - [x86_64/le]
  - Includes
  - Makefile

HelloWorld.cpp

```cpp
* HelloWorld.cpp
#include "mpi.h"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {

    int size, rank;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    cout<<"My rank is "<<rank<<endl;
    if(rank == 0){
        cout<<"We are "<<size<<" tasks"<<endl;
    }

    MPI_Finalize();
}
```

Makefile

```makefile
all: HelloWorld

HelloWorld: HelloWorld.cpp
    mpicxx -o HelloWorld HelloWorld.cpp
```

Problems Tasks Console Properties Call Graph Remote Environments

CDT Build Console [HelloWorld]

```
16:13:07 **** Incremental Build of configuration Default_remote for project HelloWorld ****
make all
mpicxx -o HelloWorld HelloWorld.cpp

16:13:09 Build Finished (took 2s.408ms)
```

Writable | Smart Insert | 26 : 1

Code development stages

1. Programming
   - Tools: editors with syntax highlighting (e.g. vim, emacs,…), development tools (e.g. Parallel Tools Platform (PTP), syntax checker (e.g. forcheck)

2. Debugging
   - Tools: write/printf statements, classical debuggers (TotalView, DDT, GDB, …), MARMOT, MUST (for MPI codes), Intel® Inspector (for OpenMP codes)

3. Performance
   - Tools: performance analysis tools (Scalasca, Vampir, TAU, …)

Code development – debugging

Murphy‘s law:

“If the code works the first time it simply means, that the bug is hidden more carefully”

MARMOT is freely available at

http://www.hlrs.de/organization/av/amt/projects/marmot/

Code development – Marmot

Tool for analyzing and checking MPI applications

- Checks usage of MPI calls during runtime
- Supports C and Fortran

Features

- Reports violations of the MPI-standard
- Reports unusual behavior or possible problems
- Displayed when harmless but remarkable behavior occurs
- MPI-calls are traced on each node throughout the whole application
- When detecting a deadlock the last few calls (as configured by the user) can be traced back on each node

Code development – Marmot Usage

Using marmot:

- Compile your application with the corresponding marmot wrapper: marmotcc, marmotcxx, marmotf77, marmotf90
- Set marmot options via environment variables
- Run your application with **n+1** MPI tasks

Some environment variables:

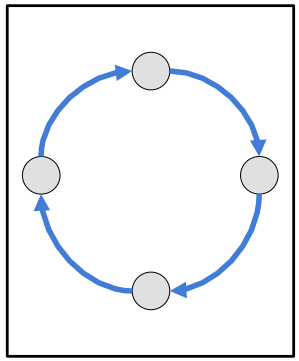
Variable	Possible values
MARMOT_DEBUG_MODE	0: errors 1: errors and warnings 2: errors, warnings and remarks
MARMOT_LOGFILE_TYPE	0: ASCII 1: HTML 2: CUBE

Code development – Marmot example

Example code

- 4 ranks on a ring
- Each rank sends a message to its right neighbor and receives a message from its left neighbor
- Compiled with

  marmotcc –o 7.1.x 7.1c



Marmot example output (HTML)

Code development – Marmot example

```
...
left  = (myrank-1+nranks)%nranks;
right = (myrank+1)%nranks;

for (i=1;i<=nranks;i++)
{
  summe = recvbuf + myrank;
  MPI_Ssend(&summe, 1, MPI_INT, right, myrank,
            MPI_COMM_WORLD);
  MPI_Recv(&recvbuf, 1, MPI_INT, left, left,
            MPI_COMM_WORLD, &status);
  MPI_Wait(&request, &status);
}
...
```

MUST is freely available (BSD license) at

https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

Code development – MUST

Tool for analyzing and checking MPI applications

- Checks usage of MPI calls during runtime
- Supports C and Fortran

MUST

MUST checks for the following classes of errors (among others)

- Communicator usage
- Datatype usage
- Leak checks (MPI resources not freed before calling MPI Finalize)
- Overlapping buffers passed to MPI
- Deadlocks resulting from MPI calls
- Basic checks for thread level usage (MPI_Init_thread)

Code development – MUST Usage

Using MUST:

- Compile your application as usual (e.g. with mpicc, mpif90, etc.)
- Replace the starter (e.g. **mpiexec**) with

```
mustrun --envall -np X application.x
```

Option	Explanation
none	• Very slow (< 32 processes) • Detects errors even if application crashes • Needs one extra process
--must:nodesize Y	• Fast • Detects errors even if application crashes • Needs 1+[X/(Y-1)] extra processes
--must:nocrash	• Fast • Detects errors only if the application does not crash • Needs one extra process

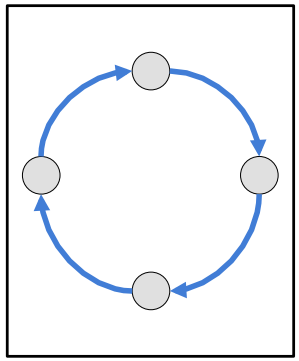
Code development – MUST example

Example code

- 4 ranks on a ring
- Each rank sends a message to its right neighbor and receives a message from its left neighbor
- Compiled with

  mpicc –o 7.1.x 7.1c
- **Started with**

  mustrun --envall -np 4 7.1.x



MUST example output (HTML)

Thread Inspector is a commercial tool

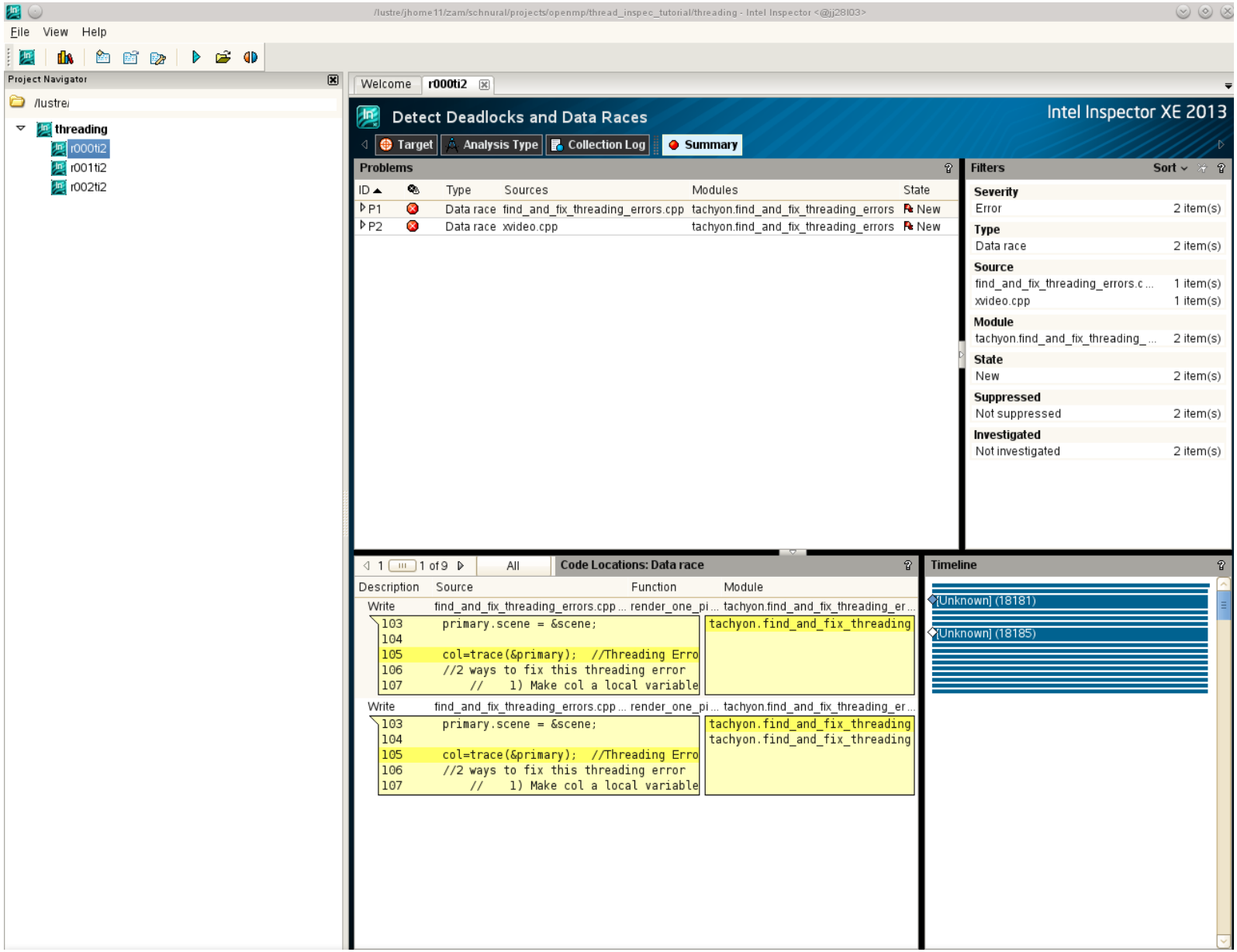
http://software.intel.com/en-us/intel-inspector-xe

Intel® Inspector Memory & Thread Analyzer

- Memory error and thread checker tool
- Supported languages on linux systems
  - C/C++, Fortran
- Maps errors to the source code line and call stack
- Detects problems that are not recognized by the compiler (e.g. race conditions, data dependencies)

Never use an OpenMP parallelized code in production without checking for race conditions

Alternatives: Threadspotter, Coverity Thread Analyzer, Sun Thread Analyzer, Helgrind


/lustre/jhome11/zam/schnural/projects/openmp/thread_inspec_tutorial/threading - Intel Inspector <@jj28l03>
File View Help
Project Navigator
/lustre/
threading
r000ti2
r001ti2
r002ti2
Welcome
r000ti2
Detect Deadlocks and Data Races
Intel Inspector XE 2013
Target
Analysis Type
Collection Log
Summary
Problems
ID Type Sources Modules State
P1 Data race find_and_fix_threading_errors.cpp tachyon.find_and_fix_threading_errors New
P2 Data race xvideo.cpp tachyon.find_and_fix_threading_errors New
Filters
Sort
Severity
Error 2 item(s)
Type
Data race 2 item(s)
Source
find_and_fix_threading_errors.c ... 1 item(s)
xvideo.cpp 1 item(s)
Module
tachyon.find_and_fix_threading_ ... 2 item(s)
State
New 2 item(s)
Suppressed
Not suppressed 2 item(s)
Investigated
Not investigated 2 item(s)
1 of 9
All
Code Locations: Data race
Description Source Function Module
Write find_and_fix_threading_errors.cpp ... render_one_pi ... tachyon.find_and_fix_threading_er ...
103 primary.scene = &scene;
104
105 col=trace(&primary); //Threading Erro
106 //2 ways to fix this threading error
107 // 1) Make col a local variable
tachyon.find_and_fix_threading
Write find_and_fix_threading_errors.cpp ... render_one_pi ... tachyon.find_and_fix_threading_er ...
103 primary.scene = &scene;
104
105 col=trace(&primary); //Threading Erro
106 //2 ways to fix this threading error
107 // 1) Make col a local variable
tachyon.find_and_fix_threading
tachyon.find_and_fix_threading
Timeline
[Unknown] (18181)
[Unknown] (18185)

TotalView is a commercial debugger

http://www.roguewave.com/products/totalview.aspx

Code development – TotalView debugger

Very powerful tool for code debugging

- Supports C, C++, Fortran
- Available for many platforms
- serial, MPI, OpenMP, hybrid MPI/OpenMP supported
- Some features:
  - Memory debugging
  - Breakpoints, evaluations points, barriers, batch debugging
  - Replay engine
  - 2D Array view, call graphs, value manipulations

Code development – TotalView debugger

Compile your code with debug flags

```
mpif90 -o prog.x -debug program.f90  # Fortran, Intel compiler
mpicc  -o prog.x -debug program.c    # C, Intel compiler
mpicxx -o prog.x -debug program.cc   # C++, Intel compiler
```

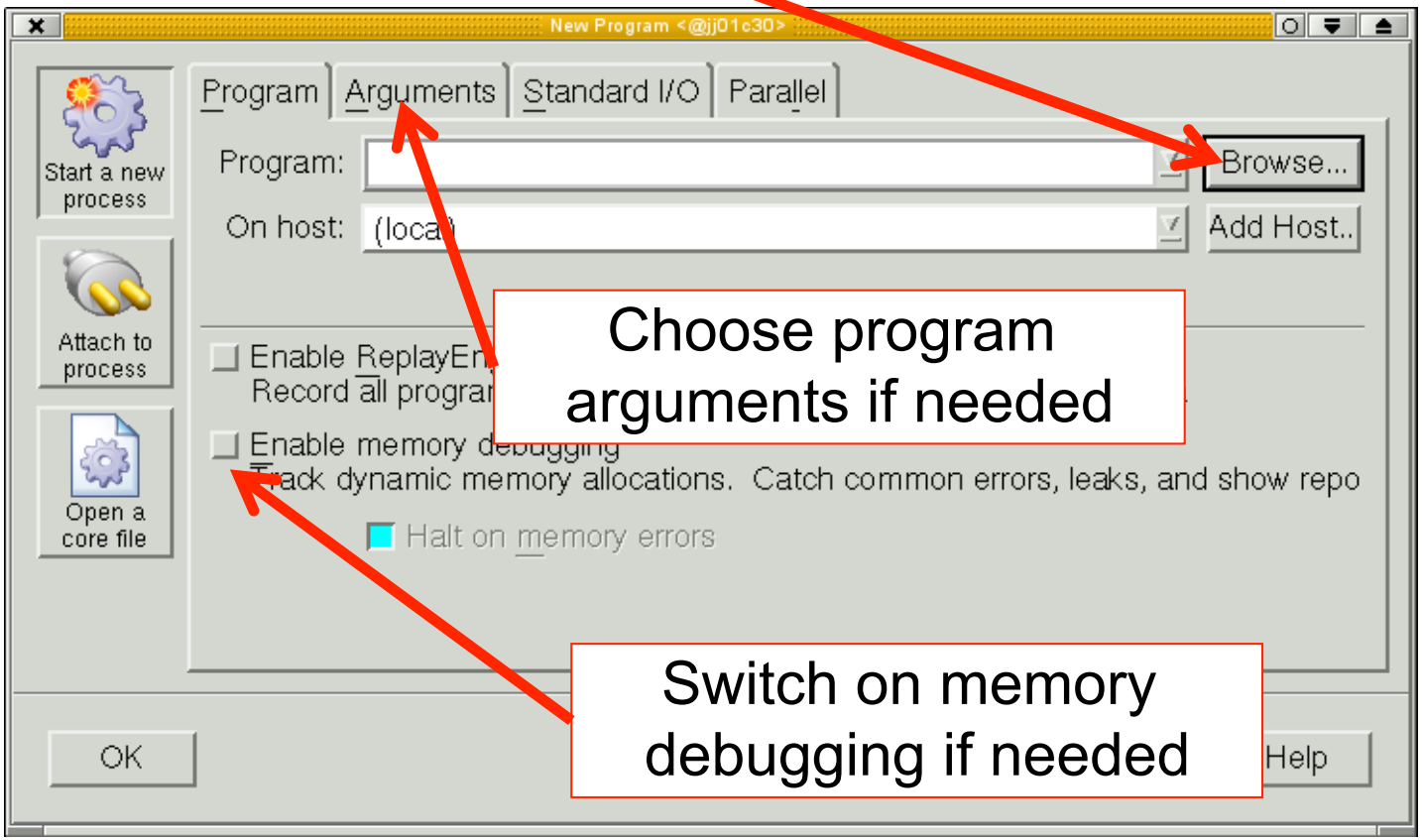
- -g -O0 also possible (as with most compilers)

TotalView execution modes

1. GUI
2. Script (**tvscript**)

Code development – TotalView

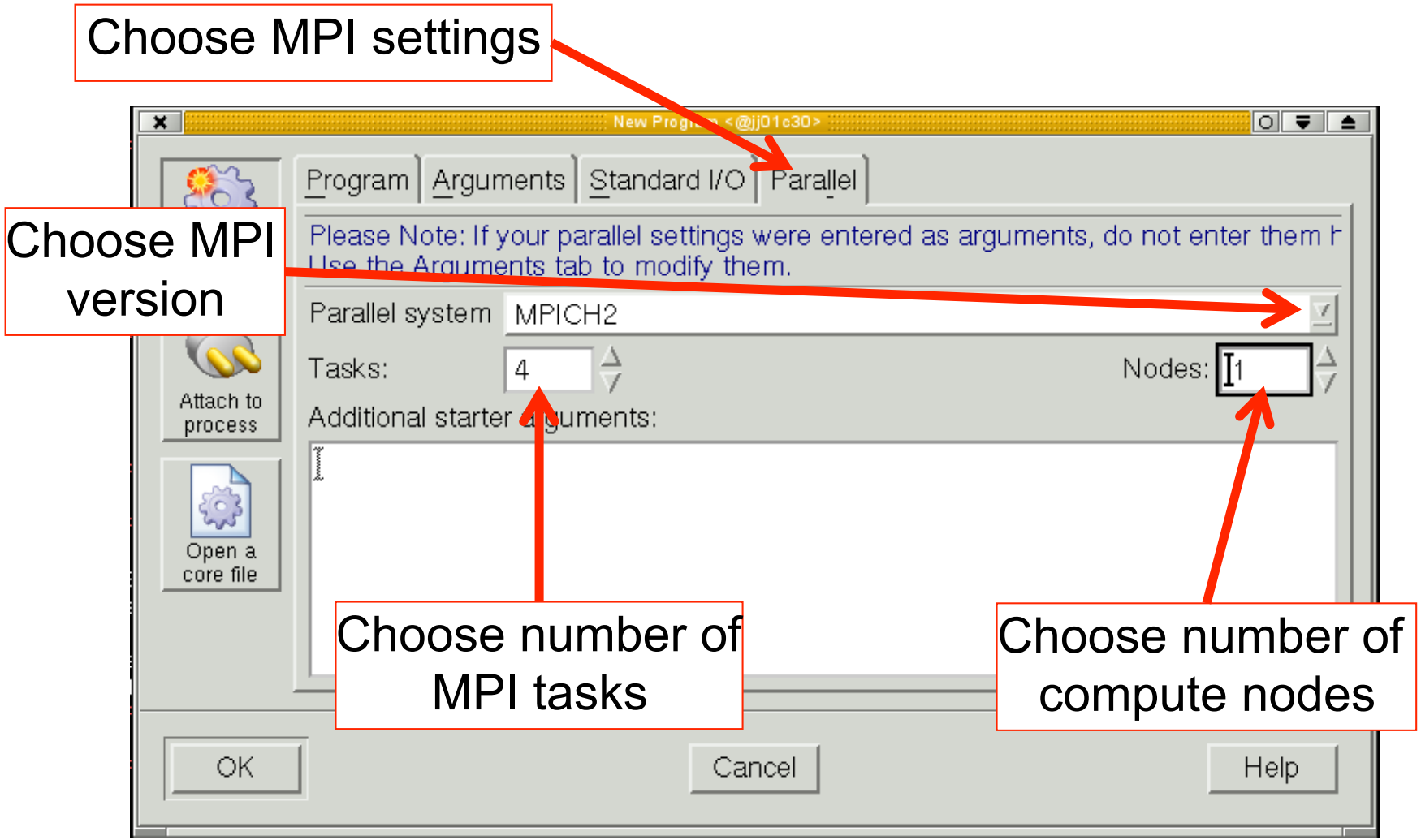
Choose your executable



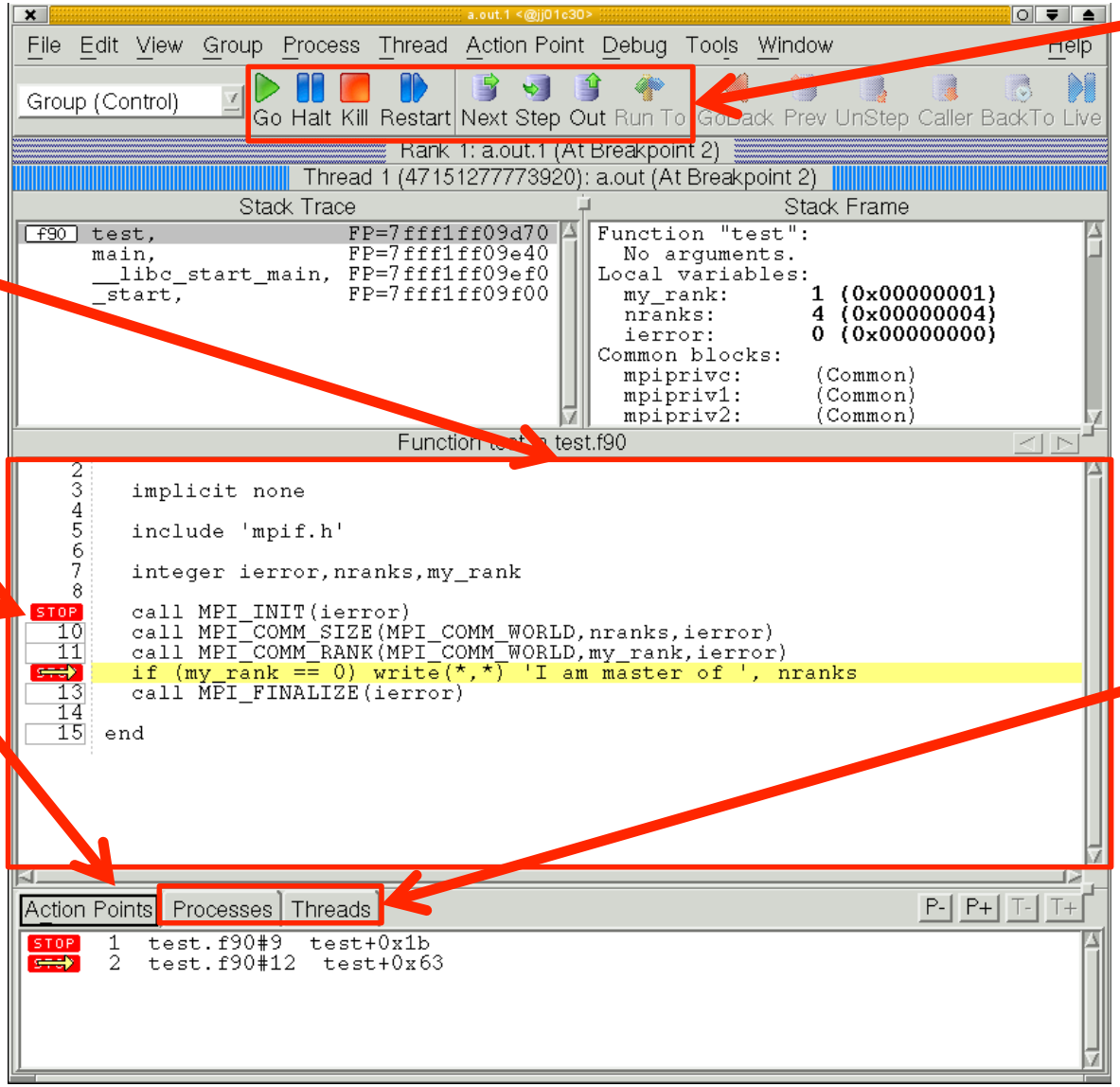
Choose program arguments if needed

Switch on memory debugging if needed

Code development – TotalView



Code development – TotalView



Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
  - Use cases
- Summary

Performance factors of applications

"Sequential" factors

- Computation
  - Choose right algorithm, use compiler to optimize
- Cache and memory
  - Tough, only limited tool support
- Input / output
  - Often not given enough attention

"Parallel" factors

- Partitioning / decomposition
- Communication (i.e., message passing)
- Multithreading
- Synchronization / locking
  - Good tool support

Parallelism: Efficiency and Scalability

Efficiency:

$$E(n) = \frac{t(1)}{n \cdot t(n)} \cdot 100\%$$

E(n): Efficiency on *n* cores/CPUS
t(1) : time on 1 core/CPU
t(n) : time on n cores/CPUs

Speed-up:

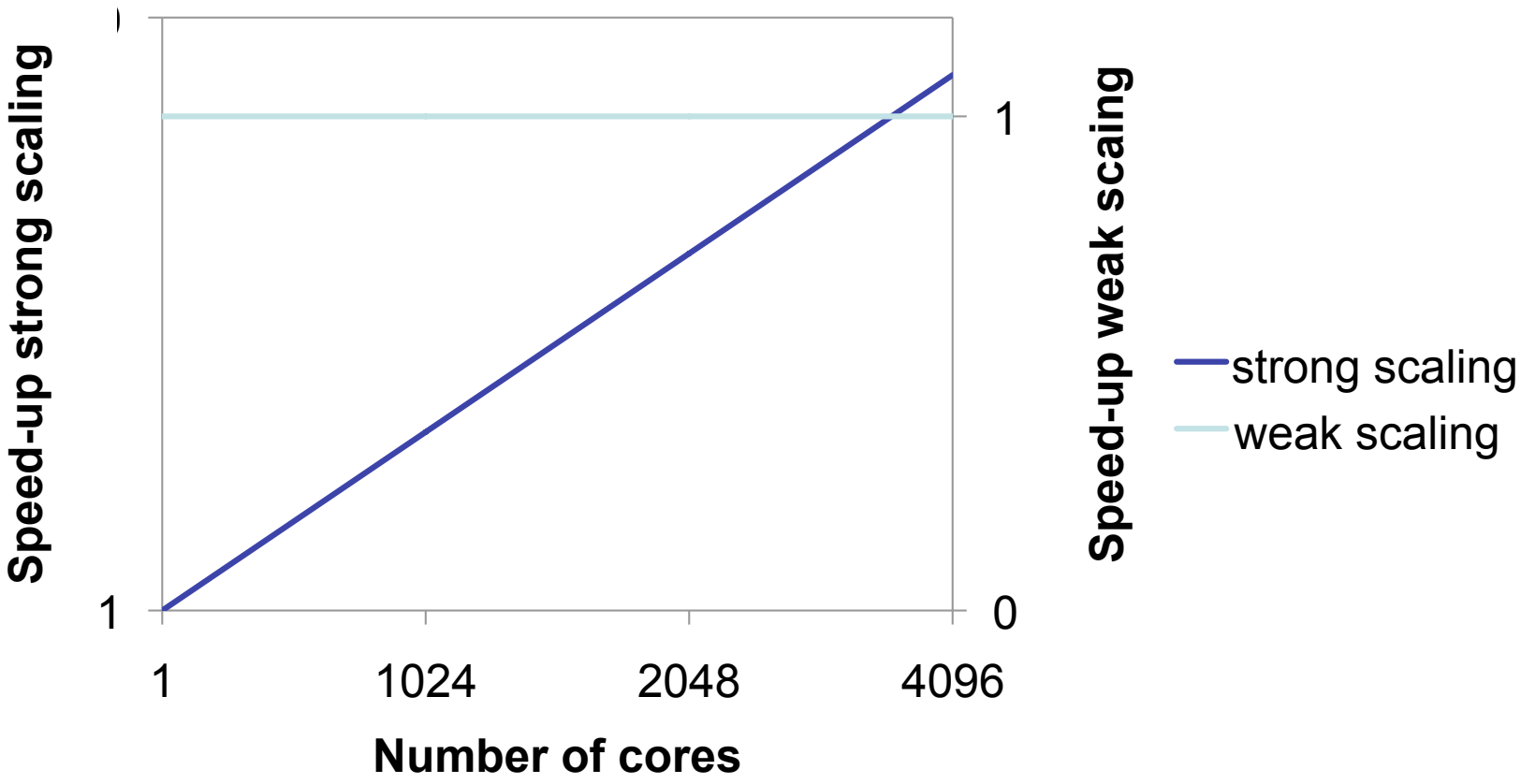
$$S(n) = \frac{t(1)}{t(n)}$$

S(n): Speed-up on *n* cores/CPUS

Scalability:

- Strong scaling (problem size constant, increase *n*)
- Weak scaling (problem-size increase proportional to *n*)

Parallelism: Ideal Scalability



Amdahl's Law

Limit of scalability:

$$S_r = \frac{1}{\alpha + \frac{1-\alpha}{n}}$$

S_r: Real speed-up

α : serial part (cannot be parallelized)

n : number of cores

Example:

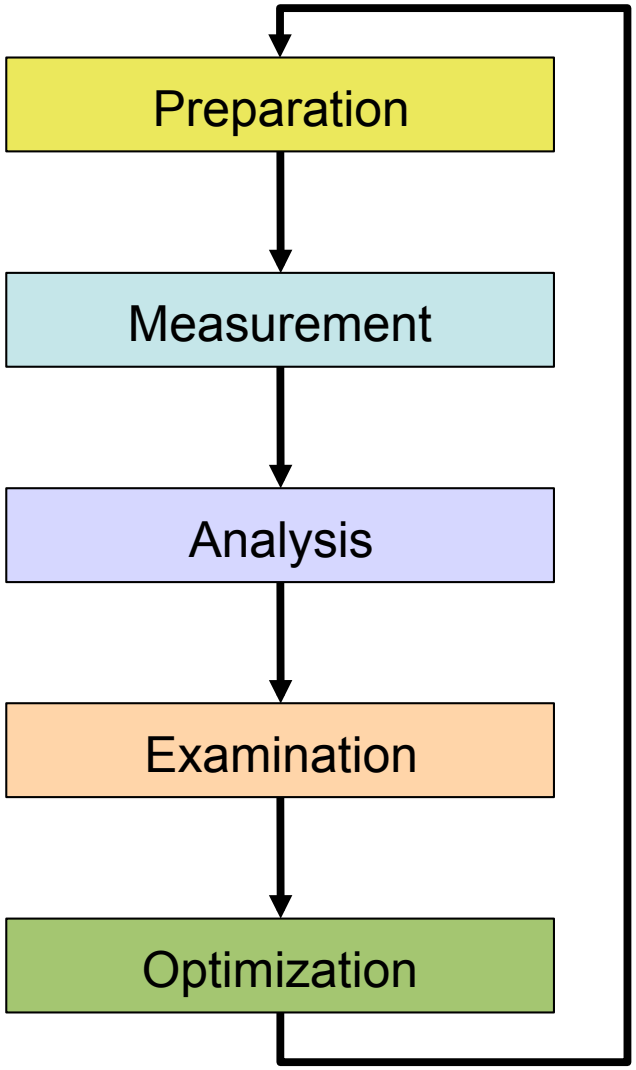
$\alpha = 0.1$

$n = 8$

→ $\mathbf{S_r = 4.7}$

$$\lim_{n\to\infty}\left(\frac{1}{\alpha + \frac{1-\alpha}{n}}\right) = \frac{1}{\alpha}$$

Performance engineering workflow



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

The 80/20 rule

Programs typically spend 80% of their time in 20% of the code

Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application

- Know when to stop!

Don't optimize what does not matter

- Make the common case fast!

"If you optimize everything, you will always be unhappy."

Donald E. Knuth

Metrics of performance

What can be measured?

- A **count** of how often an event occurs
  - E.g., the number of MPI point-to-point messages sent
- The **duration** of some interval
  - E.g., the time spent in these send calls
- The **size** of some parameter
  - E.g., the number of bytes transmitted by these calls

Derived metrics

- E.g., rates / throughput
- Needed for normalization

Example metrics

Following example metrics can be measured

- Execution time
- Number of function calls
- CPI
  - CPU cycles per instruction
- FLOPS
  - Floating-point operations executed per second

Execution time

Wall-clock time

- Includes waiting time: I/O, memory, other system activities
- In time-sharing environments also the time consumed by other applications

CPU time

- Time spent by the CPU to execute the application
- Does not include time the program was context-switched out
  - Problem: Does not include inherent waiting time (e.g., I/O)
  - Problem: Portability? What is user, what is system time?

Problem: Execution time is non-deterministic

- Use mean or minimum of several runs

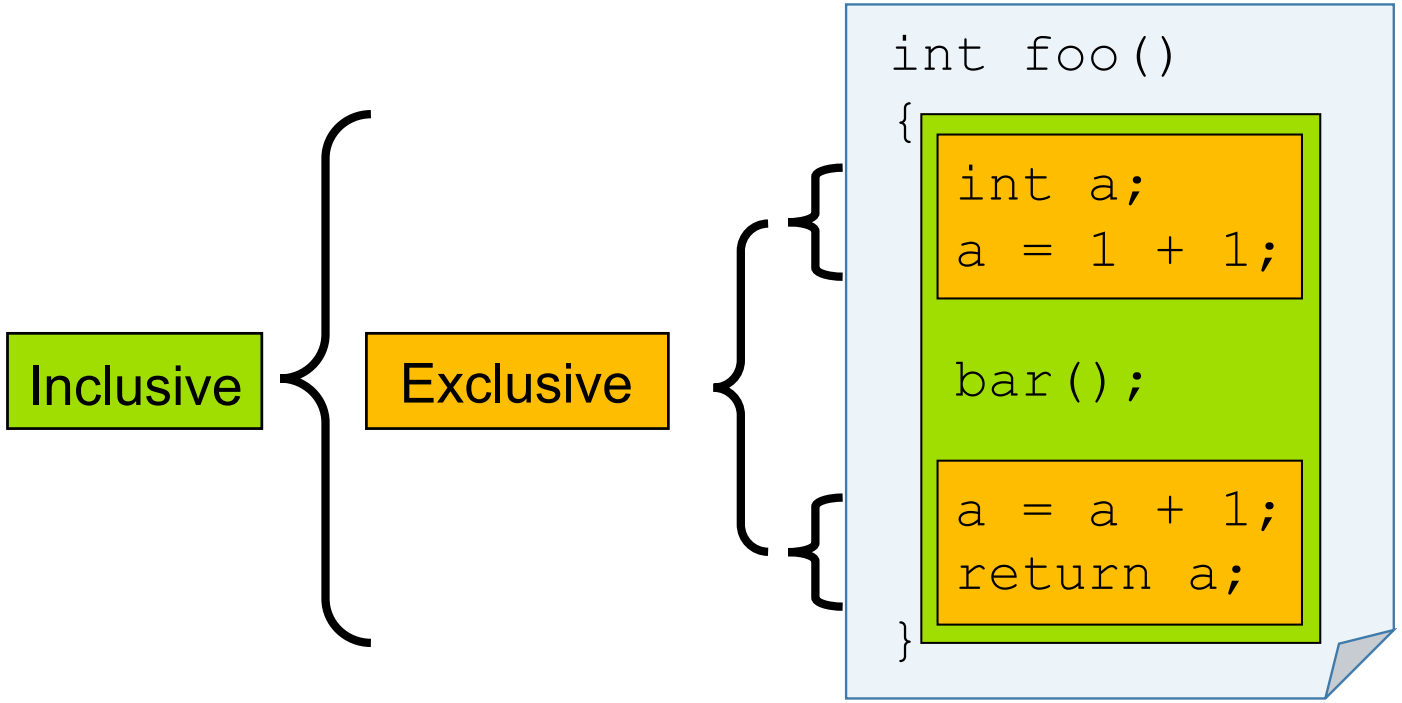
Inclusive vs. exclusive values

Inclusive

- Information of all sub-elements aggregated into single value

Exclusive

- Information cannot be subdivided further



Classification of measurement techniques

How are performance measurements triggered?

- Sampling
- Code instrumentation

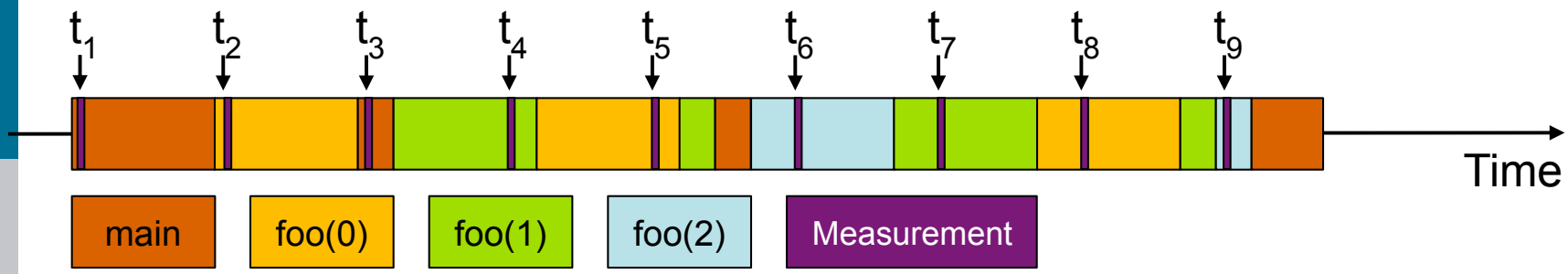
How is performance data recorded?

- Profiling / Runtime summarization
- Tracing

How is performance data analyzed?

- Online
- Post mortem

Sampling




```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i - 1);

}
```

Running program is periodically interrupted to take measurement

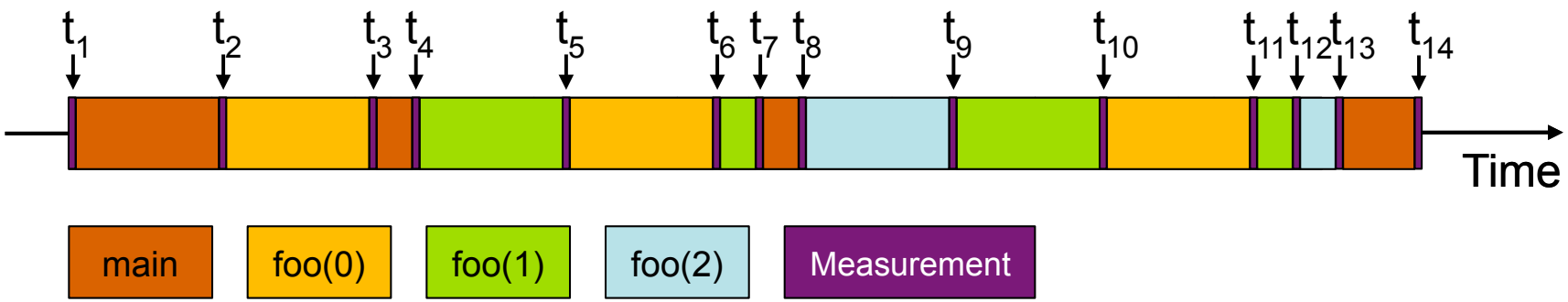
- Timer interrupt, OS signal, or HWC overflow
- Service routine examines return-address stack
- Addresses are mapped to routines using symbol table information

Statistical inference of program behavior

- Not very detailed information on highly volatile metrics
- Requires long-running applications

Works with unmodified executables

Instrumentation




```
int main()
{
  int i;
  Enter("main");
  for (i=0; i < 3; i++)
    foo(i);
  Leave("main");
  return 0;
}

void foo(int i)
{
  Enter("foo");
  if (i > 0)
    foo(i - 1);
  Leave("foo");
}
```

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

Instrumentation techniques

Static instrumentation

- Program is instrumented prior to execution

Dynamic instrumentation

- Program is instrumented at runtime

Code is inserted

- Manually
- Automatically
  - By a preprocessor / source-to-source translation tool
  - By a compiler
  - By linking against a pre-instrumented library / runtime system
  - By binary-rewrite / dynamic instrumentation tool

Critical issues

Accuracy

- Intrusion overhead
  - Measurement itself needs time and thus lowers performance
- Perturbation
  - Measurement alters program behaviour
  - E.g., memory access pattern
- Accuracy of timers & counters

Granularity

- How many measurements?
- How much information / processing during each measurement?

➢ Tradeoff: Accuracy vs. Expressiveness of data

Classification of measurement techniques

How are performance measurements triggered?

- Sampling
- Code instrumentation

How is performance data recorded?

- Profiling / Runtime summarization
- Tracing

How is performance data analyzed?

- Online
- Post mortem

Profiling / Runtime summarization

Recording of aggregated information

- Total, maximum, minimum, …

For measurements

- Time
- Counts
  - Function calls
  - Bytes transferred
  - Hardware counters

Over program and system entities

- Functions, call sites, basic blocks, loops, …
- Processes, threads

➢ Profile = summarization of events over execution interval

Types of profiles

Flat profile

- Shows distribution of metrics per routine / instrumented region
- Calling context is not taken into account

Call-path profile

- Shows distribution of metrics per executed call path
- Sometimes only distinguished by partial calling context (e.g., two levels)

Special-purpose profiles

- Focus on specific aspects, e.g., MPI calls or OpenMP constructs
- Comparing processes/threads

Tracing

Recording information about significant points (events) during execution of the program

- Enter / leave of a region (function, loop, …)
- Send / receive a message, …

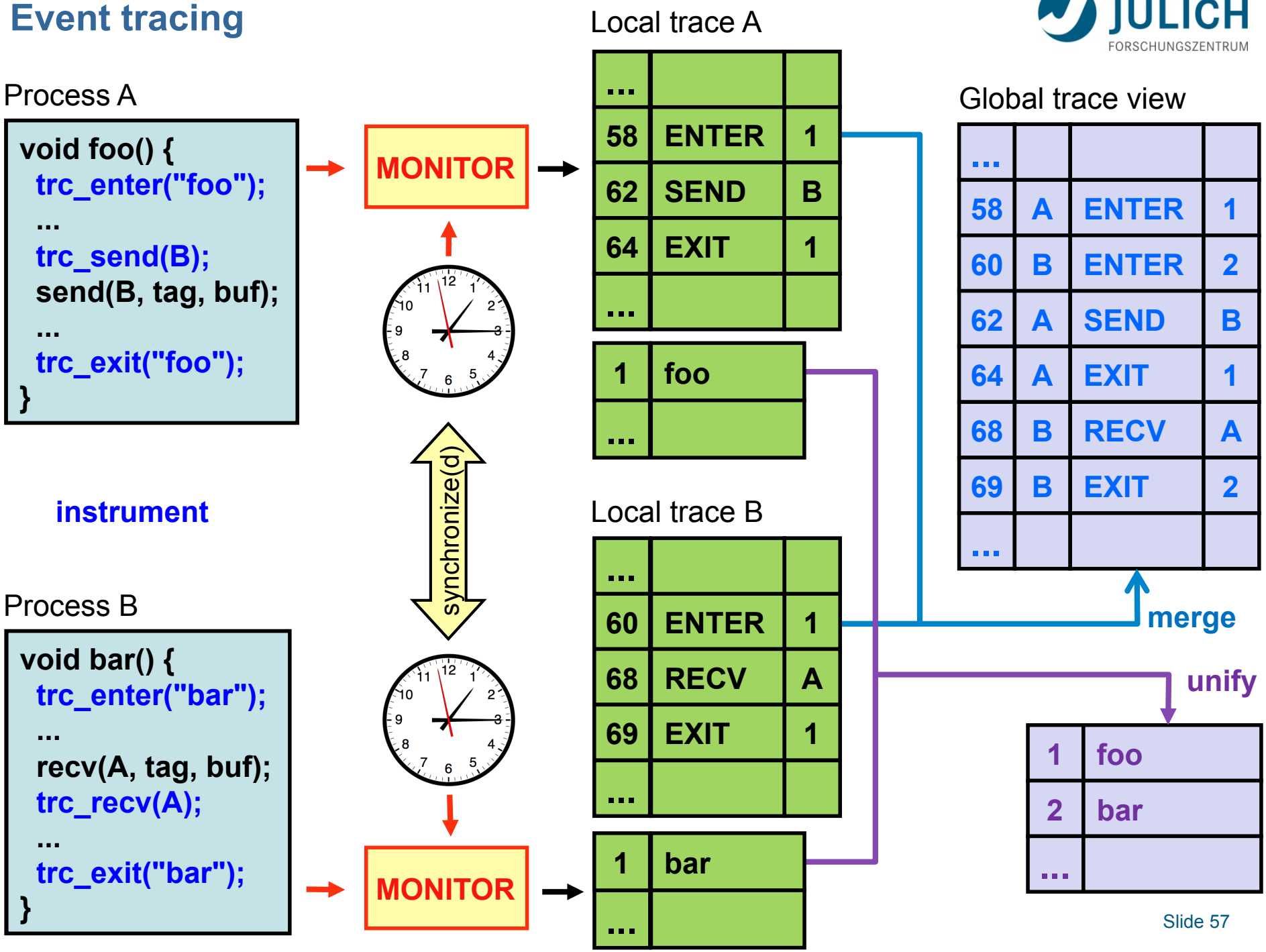
Save information in event record

- Timestamp, location, event type
- Plus event-specific information (e.g., communicator, sender / receiver, …)

Abstract execution model on level of defined events

➢ Event trace = Chronologically ordered sequence of event records

Event tracing



Tracing vs. Profiling

Tracing advantages

- Event traces preserve the temporal and spatial relationships among individual events (→ context)
- Allows reconstruction of dynamic application behavior on any required level of abstraction
- Most general measurement technique
  - Profile data can be reconstructed from event traces

Disadvantages

- Traces can very quickly become extremely large
- Writing events to file at runtime causes perturbation
- Writing tracing software is complicated
  - Event buffering, clock synchronization, ...

Classification of measurement techniques

How are performance measurements triggered?

- Sampling
- Code instrumentation

How is performance data recorded?

- Profiling / Runtime summarization
- Tracing

How is performance data analyzed?

- Online
- Post mortem

Online analysis

Performance data is processed during measurement run

- Process-local profile aggregation
- More sophisticated inter-process analysis using
  - “Piggyback” messages
  - Hierarchical network of analysis agents

Inter-process analysis often involves application steering to interrupt and re-configure the measurement

Post-mortem analysis

Performance data is stored at end of measurement run

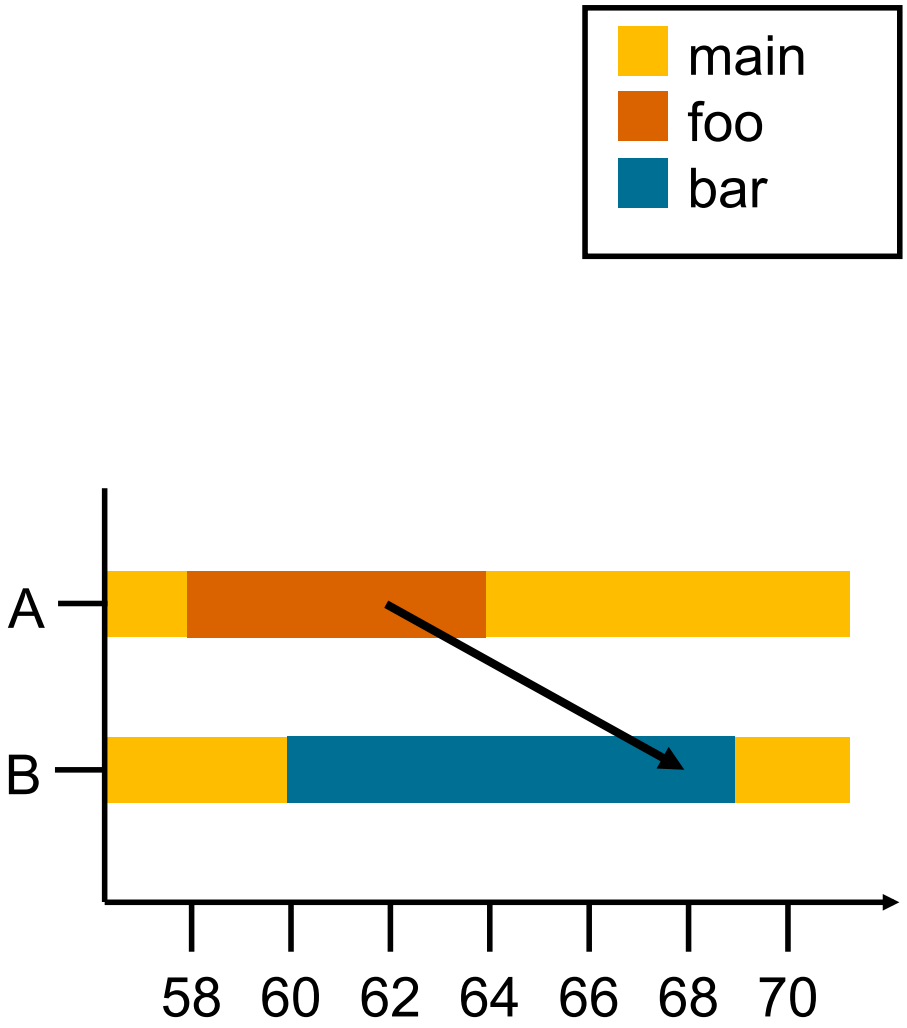
Data analysis is performed afterwards

- Automatic search for bottlenecks
- Visual trace analysis
- Calculation of statistics

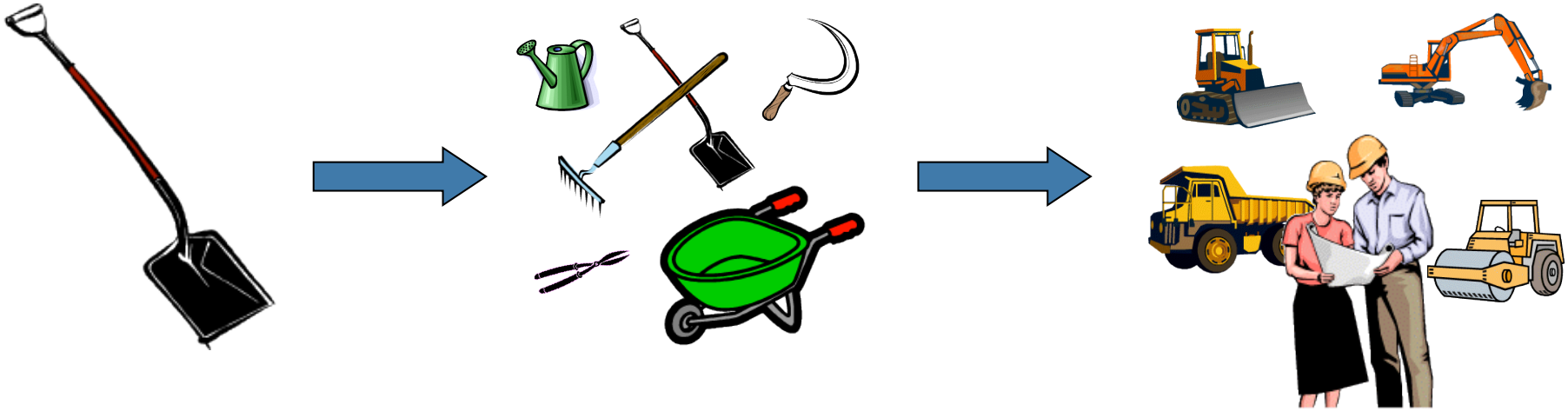
Example: Time-line visualization

1	foo
2	bar
3	...

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



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, ...

Typical performance analysis procedure

Do I have a performance problem at all?

- Time / speedup / scalability measurements

What is the key bottleneck (computation / communication)?

- MPI / OpenMP / flat profiling

Where is the key bottleneck?

- Call-path profiling, detailed basic block profiling

Why is it there?

- Hardware counter analysis, trace selected parts to keep trace size manageable

Does the code have scalability problems?

- Load imbalance analysis, compare profiles at various sizes function-by-function

Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
    - Computational load imbalance
    - MPI patterns
    - OpenMP patterns
  - Selected performance analysis tools
  - Use cases
- Summary

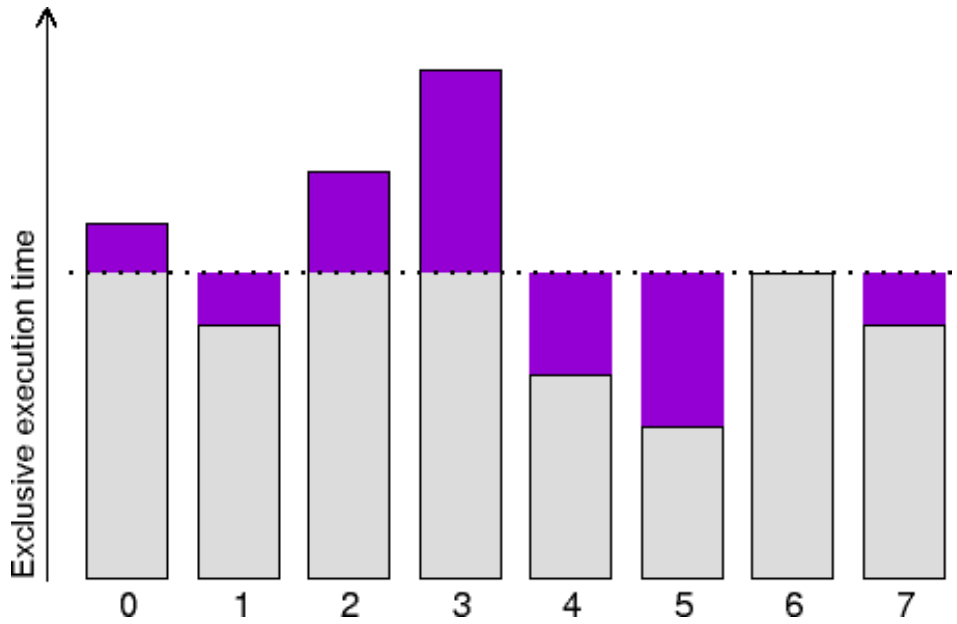
Computational imbalance

Absolute difference to average exclusive execution time

- Focusses only on computational parts

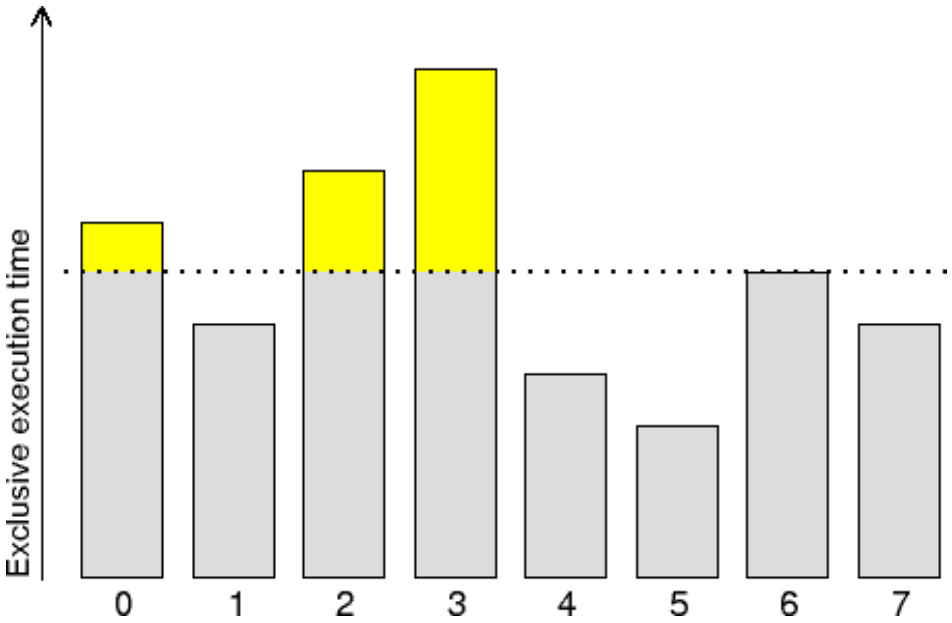
Captures global imbalances

- Based on entire measurement
- Does not compare individual instances of function calls



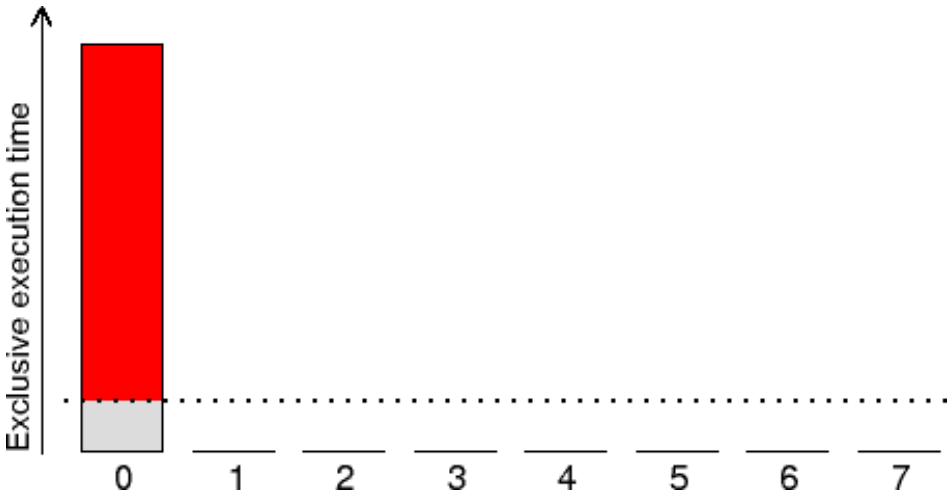
Overload

Processes/threads with exclusive execution time above average



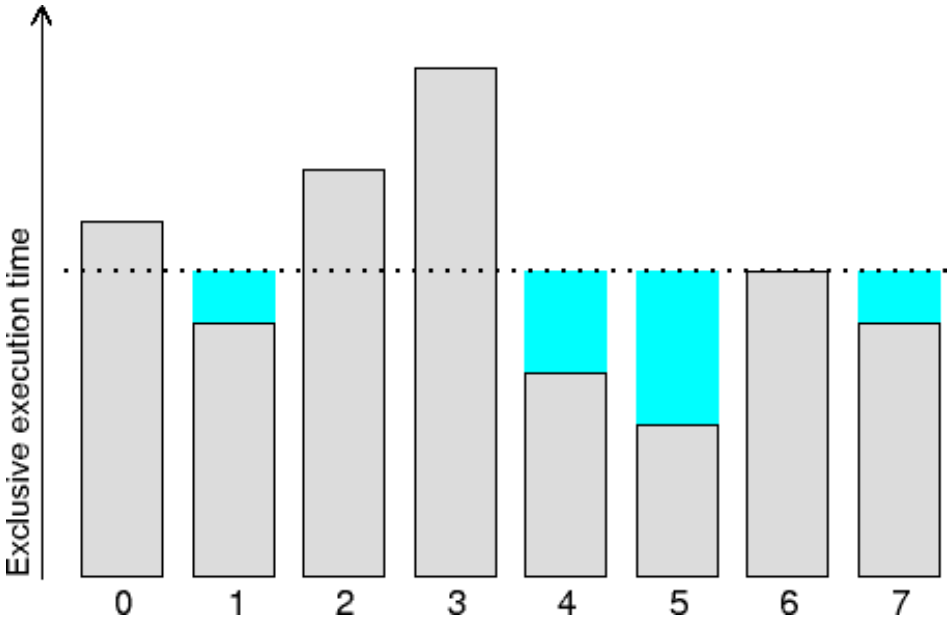
Overload, single participant

Call-paths executed by single process/thread



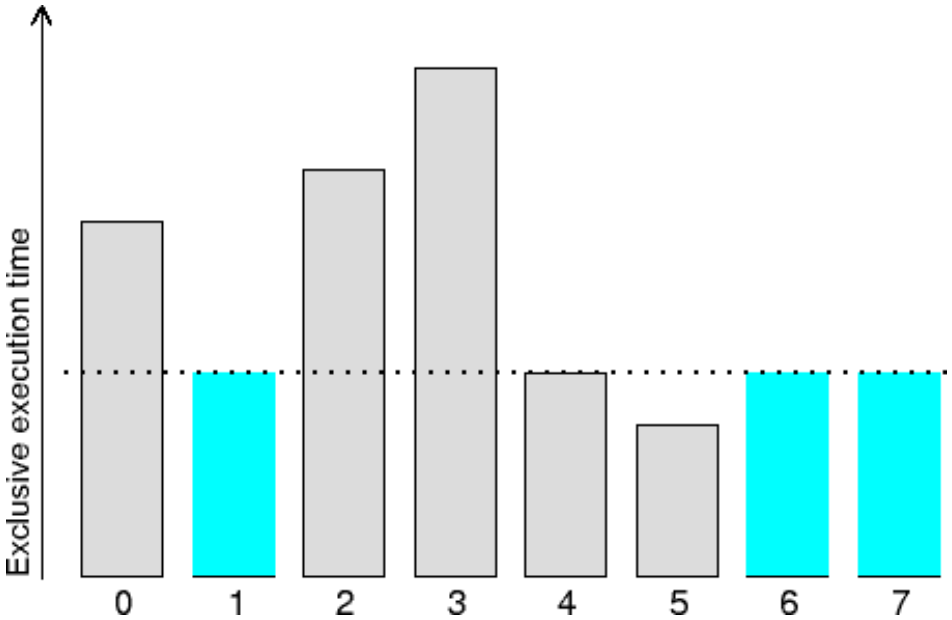
Underload

Processes/threads with exclusive execution time below average



Underload, non-participation

Call-paths not executed by a subset of processes/threads

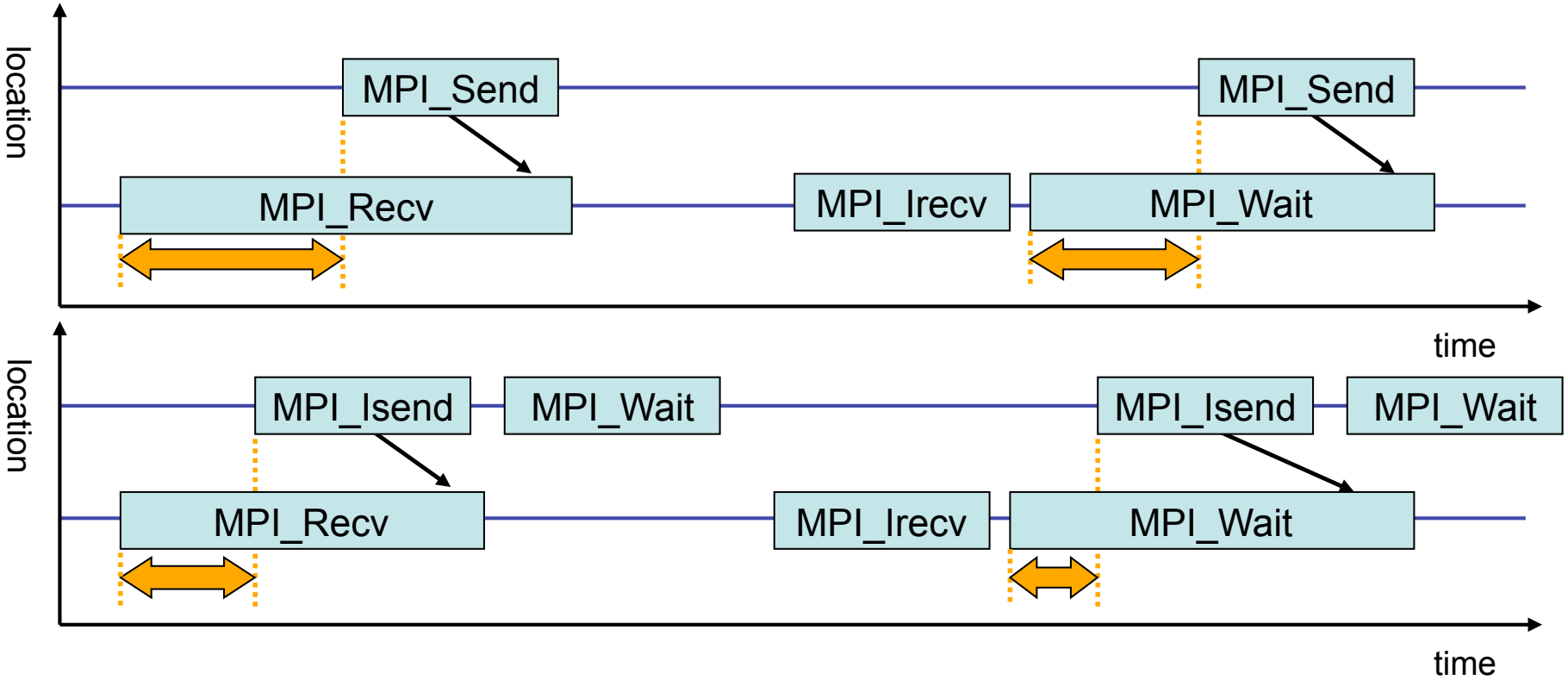


Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
    - Computational load imbalance
    - MPI patterns
    - OpenMP patterns
  - Selected performance analysis tools
  - Use cases
- Summary

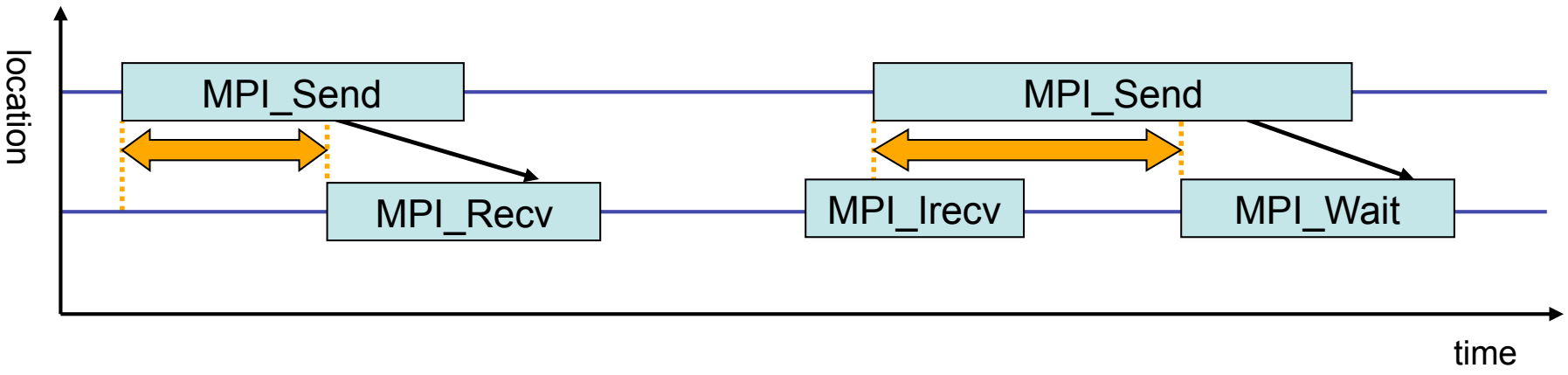
Late sender

- Waiting time caused by a blocking receive operation posted earlier than the corresponding send operation
- Applies to blocking as well as non-blocking communication



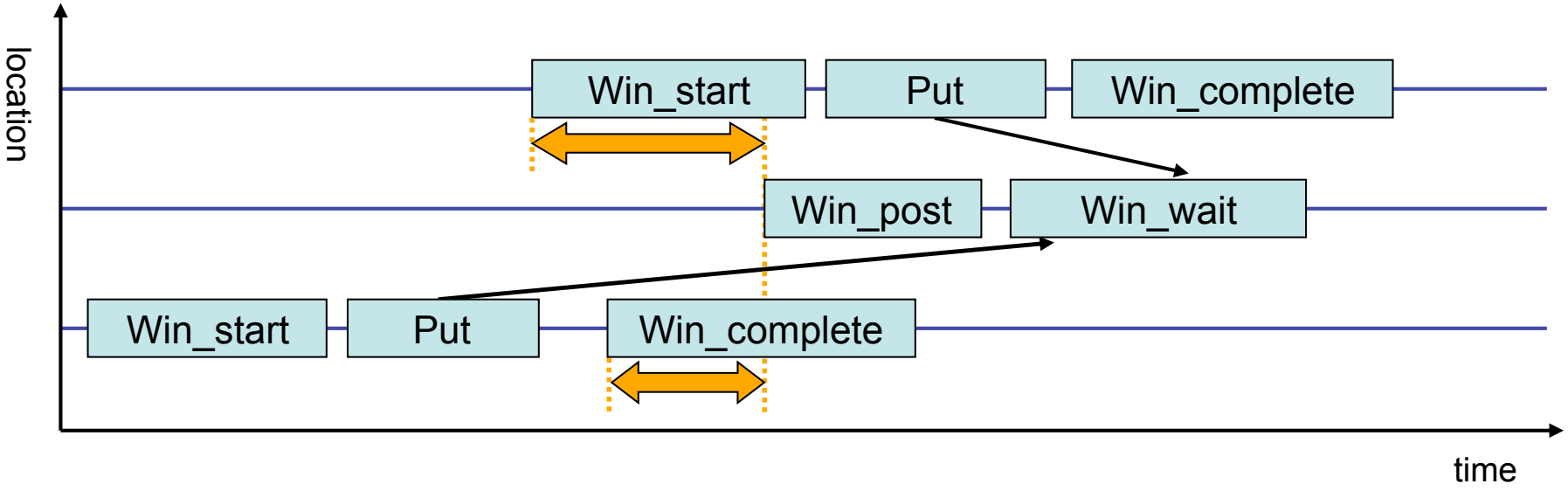
Late receiver

- Waiting time caused by a blocking send operation posted earlier than the corresponding receive operation
- Calculated by receiver but waiting time attributed to sender
- Does currently not apply to non-blocking sends



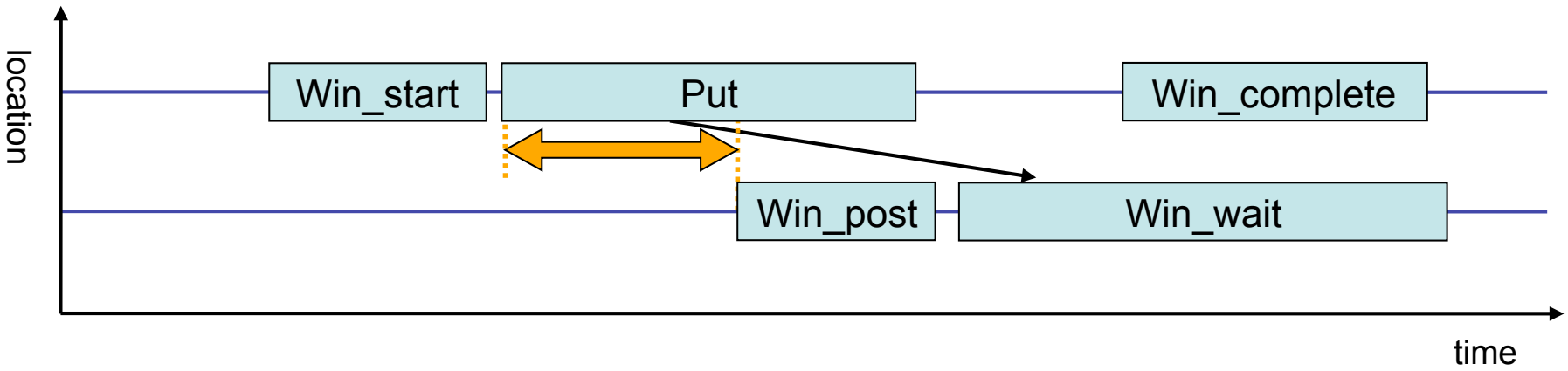
Late post

- **MPI_Win_start** (top) or **MPI_Win_complete** (bottom) wait until exposure epoch is opened by **MPI_Win_post**
- Which of the two calls blocks is implementation dependent



Early transfer

- Time spent waiting in RMA operation on origin(s) started before exposure epoch was opened on target

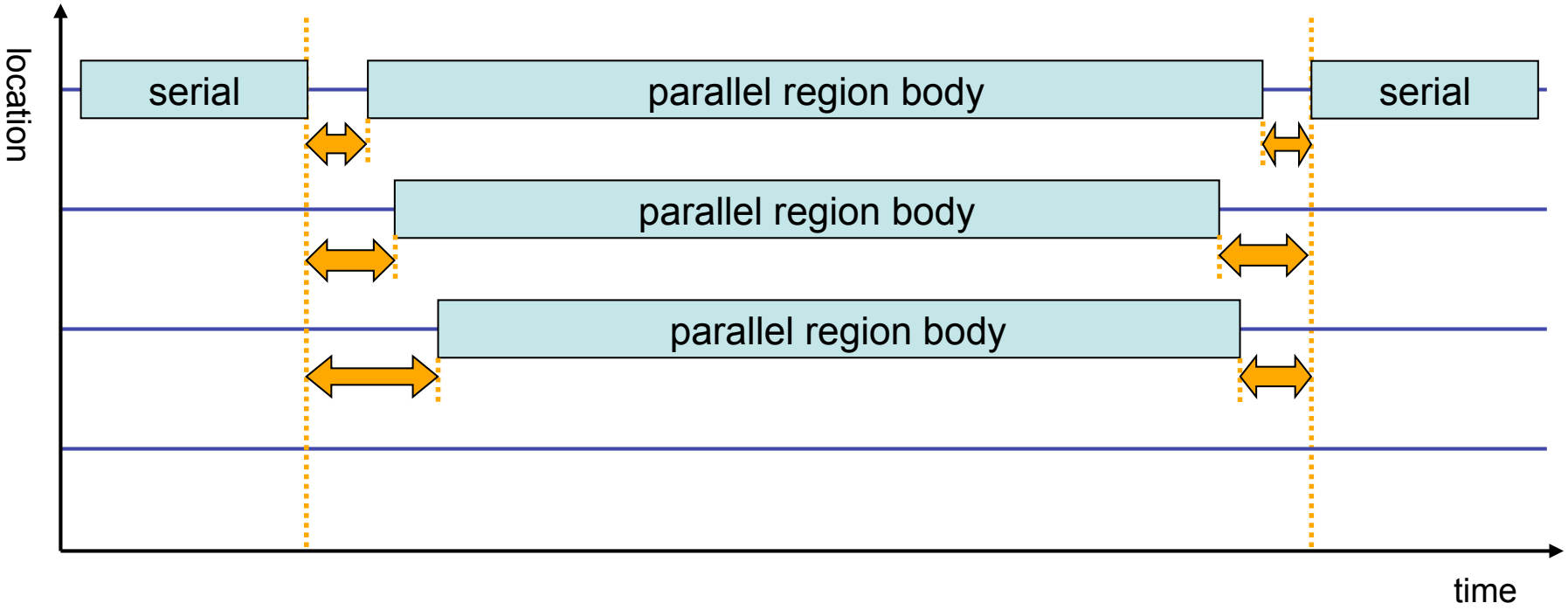


Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
    - Computational load imbalance
    - MPI patterns
    - OpenMP patterns
  - Selected performance analysis tools
  - Use cases
- Summary

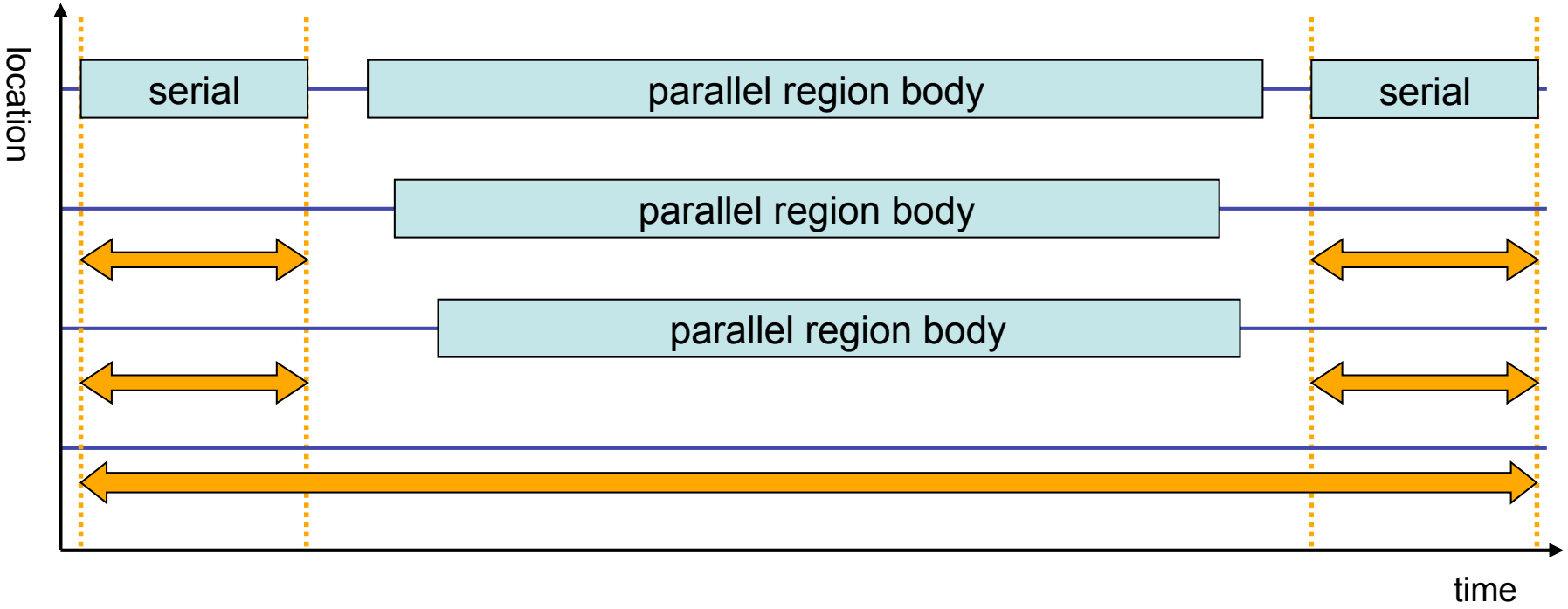
OpenMP management time

- Time spent on master thread for creating/destroying OpenMP thread teams



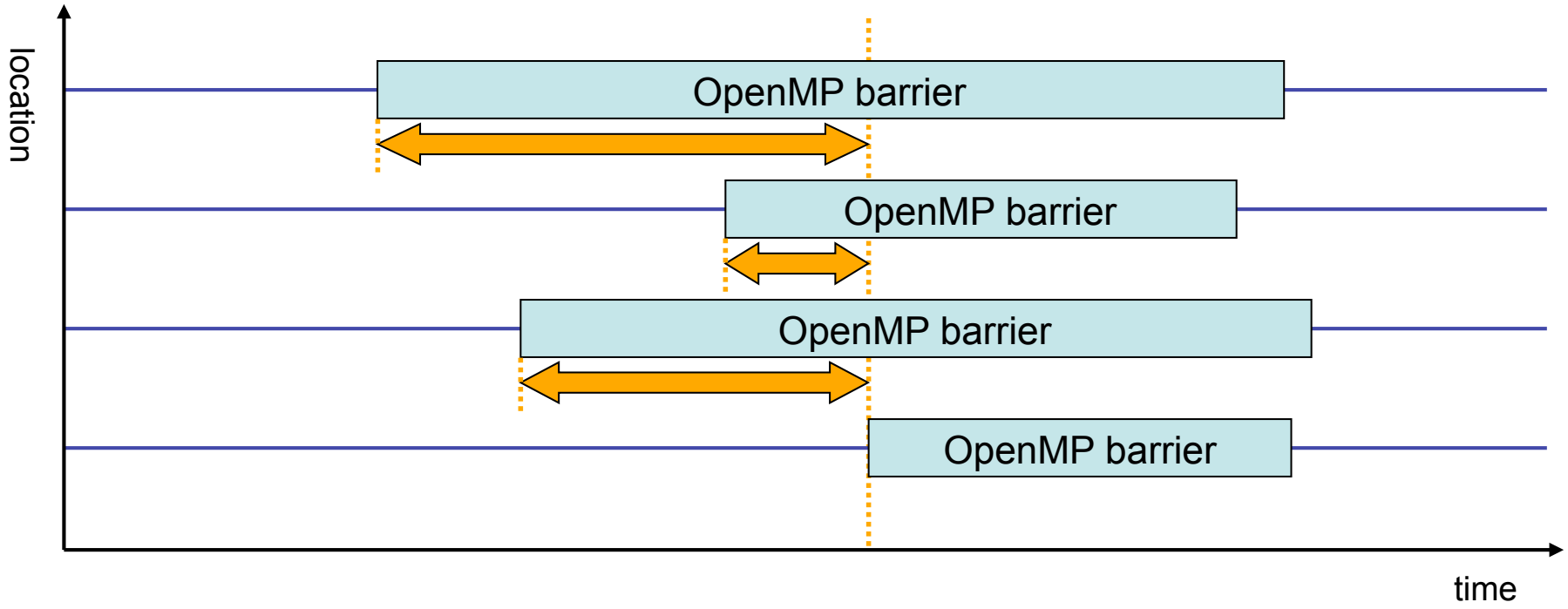
OpenMP idle threads

- Time spent idle on CPUs reserved for worker threads



OpenMP waiting at barrier

- Time spent waiting in front of a barrier call until the last process reaches the barrier operation
- Applies to: Implicit/explicit barriers



Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
    - Score-P
    - Scalasca
    - CUBE
    - Vampir
    - TAU
  - Use cases
- Summary

Fragmentation of tools landscape

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

SILC Project Idea

Start a community effort for a common infrastructure

- Score-P instrumentation and measurement system
- Common data formats OTF2 and CUBE4

Developer perspective:

- Save manpower by sharing development resources
- Invest in new analysis functionality and scalability
- Save efforts for maintenance, testing, porting, support, training

User perspective:

- Single learning curve
- Single installation, fewer version updates
- Interoperability and data exchange

SILC project funded by BMBF

Close collaboration PRIMA project funded by DOE

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Partners

Forschungszentrum Jülich, Germany

German Research School for Simulation Sciences, Aachen, Germany

Gesellschaft für numerische Simulation mbH Braunschweig, Germany

RWTH Aachen, Germany

Technische Universität Dresden, Germany

Technische Universität München, Germany

University of Oregon, Eugene, USA

Score-P Functionality

Provide typical functionality for HPC performance tools

Support all fundamental concepts of partner's tools

Instrumentation (various methods)

Flexible measurement without re-compilation:

- Basic and advanced profile generation
- Event trace recording
- Online access to profiling data

MPI, OpenMP, and hybrid parallelism (and serial)

Enhanced functionality (OpenMP 3.0, CUDA, highly scalable I/O)

Design Goals

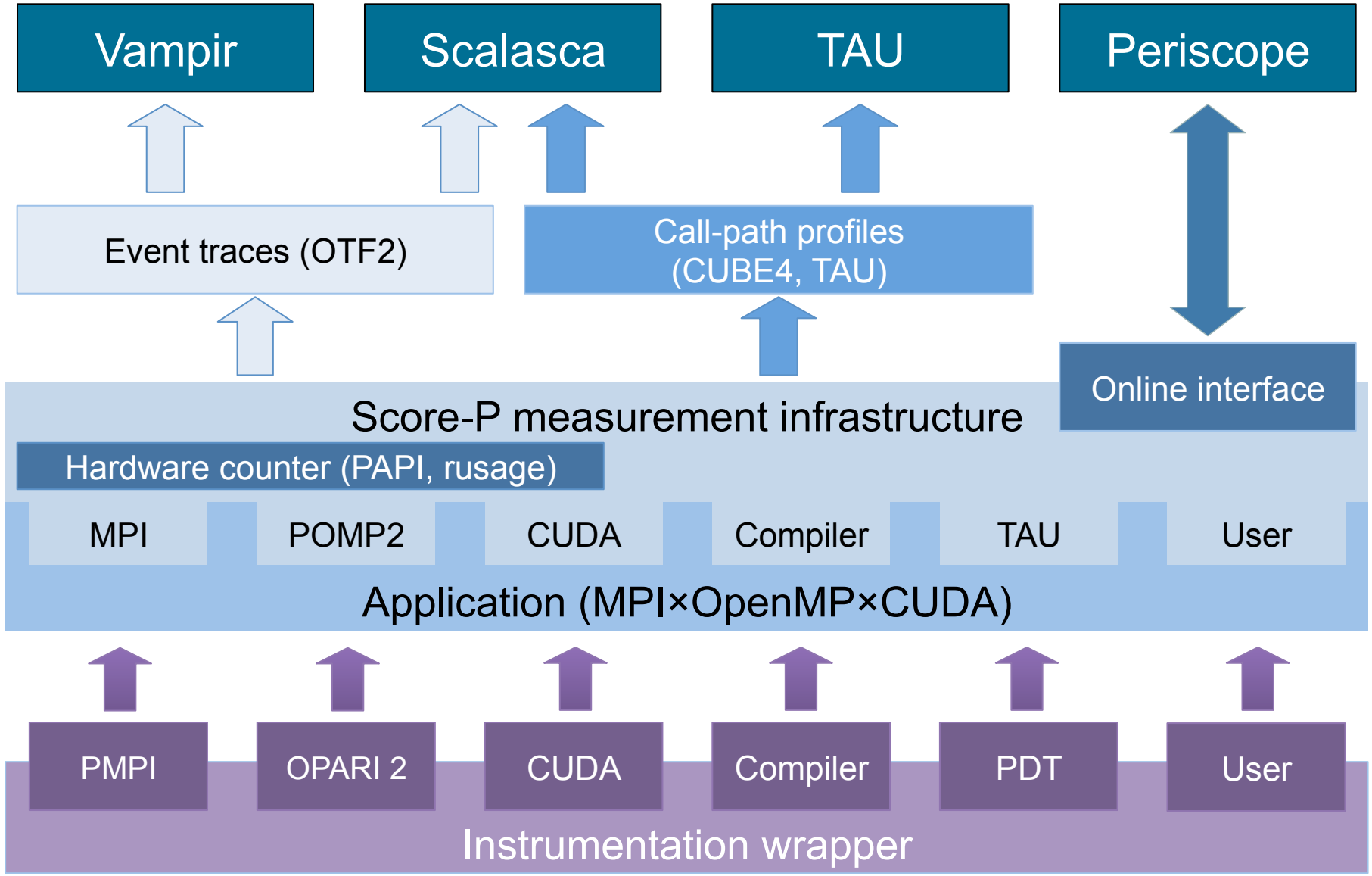
Functional requirements

- Generation of call-path profiles and event traces
- Using direct instrumentation, later also sampling
- Recording time, visits, communication data, hardware counters
- Access and reconfiguration also at runtime
- Support for MPI, OpenMP, basic CUDA, and all combinations
  - Later also OpenCL/HMPP/PTHREAD/…

Non-functional requirements

- Portability: all major HPC platforms
- Scalability: petascale
- Low measurement overhead
- Easy and uniform installation through UNITE framework
- Robustness
- Open Source: New BSD License

Score-P Architecture



Code instrumentation with Score-P

Automatic instrumentation

- Prefix compiler and linker command e.g. in your Makefile

mpicc …	**→ scorep mpicc …**
mpif90 …	**→ scorep mpif90 …**

Manual instrumentation

- Add instructions to your code manually
- Available for Fortran (requires C preprocessor), C, and C++
- Can be used to
  - Add measurements
  - Disable (automatically instrumented) measurements

Score-P User Instrumentation API (Fortran)

```
#include "scorep/SCOREP_User.inc"

subroutine foo(…)
  ! Declarations
  SCOREP_USER_REGION_DEFINE( solve )

  ! Some code…
  SCOREP_USER_REGION_BEGIN( solve, "<solver>", \
                            SCOREP_USER_REGION_TYPE_LOOP )
  do i=1,100
    [...]
  end do
  SCOREP_USER_REGION_END( solve )
  ! Some more code…
end subroutine
```

Requires processing by the C preprocessor

Score-P User Instrumentation API (C/C++)

```
#include "scorep/SCOREP_User.h"

void foo()
{
  /* Declarations */
  SCOREP_USER_REGION_DEFINE( solve )

  /* Some code… */
  SCOREP_USER_REGION_BEGIN( solve, "<solver>", \
                            SCOREP_USER_REGION_TYPE_LOOP )
  for (i = 0; i < 100; i++)
  {
    [...]
  }
  SCOREP_USER_REGION_END( solve )
  /* Some more code… */
}
```

Score-P Measurement Control API

Can be used to temporarily disable measurement for certain intervals

- Annotation macros ignored by default
- Enabled with [**--user**] flag

```
#include "scorep/SCOREP_User.inc"

subroutine foo(…)
  ! Some code…
  SCOREP_RECORDING_OFF()
  ! Loop will not be measured
  do i=1,100
    [...]
  end do
  SCOREP_RECORDING_ON()
  ! Some more code…
end subroutine
```

Fortran (requires C preprocessor)

```
#include "scorep/SCOREP_User.h"

void foo(…) {
  /* Some code… */
  SCOREP_RECORDING_OFF()
  /* Loop will not be measured */
  for (i = 0; i < 100; i++) {
    [...]
  }
  SCOREP_RECORDING_ON()
  /* Some more code… */
}
```

C / C++

Measurement configuration: scorep-info

Score-P measurements are configured via environment

```
% scorep-info config-vars --full
SCOREP_ENABLE_PROFILING
  Description: Enable profiling
           [...]
SCOREP_ENABLE_TRACING
  Description: Enable tracing
           [...]
SCOREP_TOTAL_MEMORY
  Description: Total memory in bytes for the measurement system
           [...]
SCOREP_EXPERIMENT_DIRECTORY
  Description: Name of the experiment directory
           [...]
SCOREP_FILTERING_FILE
  Description: A file name which contain the filter rules
           [...]
SCOREP_METRIC_PAPI
  Description: PAPI metric names to measure
           [...]
SCOREP_METRIC_RUSAGE
  Description: Resource usage metric names to measure
           [... More configuration variables ...]
```

Example summary analysis result scoring

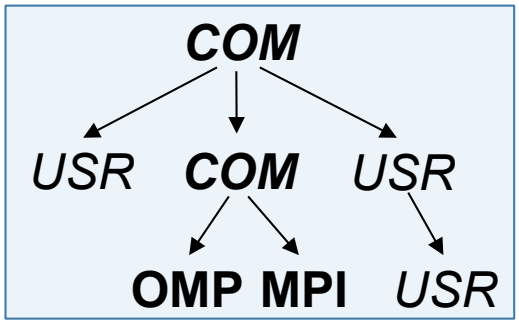
Report scoring as textual output

```
% scorep-score scorep_example_sum/profile.cubex
Estimated aggregate size of event trace (total_tbc):       35955109198 bytes
Estimated requirements for largest trace buffer (max_tbc):  9043348074 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid intermediate flushes
 or reduce requirements using file listing names of USR regions to be filtered.)

flt type         max_tbc         time      % region
     ALL      9043348074       933.55  100.0 ALL
     USR      9025830154       450.52   48.3 USR
     OMP        16431872       480.67   51.5 OMP
     COM          997150         0.67    0.1 COM
     MPI           88898         1.69    0.2 MPI
```

33.5 GB total memory
8.4 GB per rank!

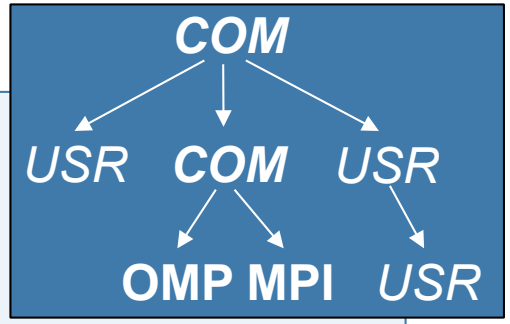
Region/callpath classification

- MPI (pure MPI library functions)
- OMP (pure OpenMP functions/regions)
- USR (user-level source local computation)
- COM ("combined" USR + OpenMP/MPI)
- ANY/ALL (aggregate of all region types)



Example summary analysis report breakdown

Score report breakdown by region




```
% scorep-score -r scorep_example_sum/profile.cubex
  [...]
flt type          max_tbc          time      % region
     ALL       9043348074        933.55  100.0 ALL
     USR       9025830154        450.52   48.3 USR
     OMP         16431872        480.67   51.5 OMP
                   997150          0.67    0.1 COM
                    88898          1.69    0.2 MPI

                2894950740        137.99   14.8 matmul_sub_
                2894950740        119.71   12.8 matvec_sub_
     USR        2894950740        175.59   18.8 binvcrhs_
     USR         127716204          6.08    0.7 binvrhs_
     USR         127716204          7.38    0.8 lhsinit_
     USR          94933520          3.76    0.4 exact_solution
     OMP            771840          0.05    0.0 !$omp parallel @exch_...
     OMP            771840          0.04    0.0 !$omp parallel @exch_...
     OMP            771840          0.05    0.0 !$omp parallel @exch_...
 [...]
```

More than 8 GB just for these 6 regions

Analysis results

Summary measurement analysis score reveals

- Total size of event trace would be ~34 GB
- Maximum trace buffer size would be ~8.5 GB per rank
  - smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
- 99.8% of the trace requirements are for USR regions
  - purely computational routines never found on COM call-paths common to communication routines or OpenMP parallel regions
- These USR regions contribute around 32% of total time
  - however, much of that is very likely to be measurement overhead for frequently-executed small routines

Advisable to tune measurement configuration

- Specify an adequate trace buffer size
- Specify a filter file listing (USR) regions not to be measured

Example Summary Analysis Report Filtering

Report scoring with prospective filter listing 6 USR regions

```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN EXCLUDE
binvcrhs*
matmul_sub*
matvec_sub*
exact_solution*
binvrhs*
lhs*init*
timer_*

% scorep-score -f ../config/scorep.filt scorep_example_sum/profile.cubex
Estimated aggregate size of event trace (total_tbc):       70086838 bytes
Estimated requirements for largest trace buffer (max_tbc): 17521726 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid intermediate flushes
 or reduce requirements using file listing names of USR regions to be filtered.)
```

67 MB of memory in total,
17 MB per rank!

New summary analysis result scoring

Scoring of new analysis report as textual output

```
% scorep-score scorep_example_sum_with_filter/profile.cubex
Estimated aggregate size of event trace (total_tbc):       70086838 bytes
Estimated requirements for largest trace buffer (max_tbc): 17521726 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid intermediate flushes
 or reduce requirements using file listing names of USR regions to be filtered.)

flt type         max_tbc         time      % region
     ALL        17521726       215.07  100.0 ALL
     OMP        16431872       212.86   99.0 OMP
     COM          997150         0.68    0.3 COM
     MPI           88898         1.54    0.7 MPI
     USR            3806         0.00    0.0 USR
```

Significant reduction in runtime (measurement overhead)

- Not only reduced time for USR regions, but MPI/OMP reduced too!

Further measurement tuning (filtering) may be appropriate

- e.g., use “timer_*” to filter timer_start_, timer_read_, etc.

Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
    - Score-P
    - Scalasca
    - CUBE
    - Vampir
    - TAU
  - Use cases
- Summary

scalasca

Scalasca is available at http://www.scalasca.org/,

get support via scalasca@fz-juelich.de

The Scalasca project: Overview

Project started in 2006

- Initial funding by Helmholtz Initiative & Networking Fund
- Many follow-up projects

Follow-up to pioneering KOJAK project (started 1998)

- Automatic pattern-based trace analysis

Now joint development of

- Jülich Supercomputing Centre
- German Research School for Simulation Sciences

Scalasca 2.0 features

Open source, New BSD license

Fairly portable

- IBM Blue Gene, IBM SP & blade clusters, Cray XT, SGI Altix, Solaris & Linux clusters, ...

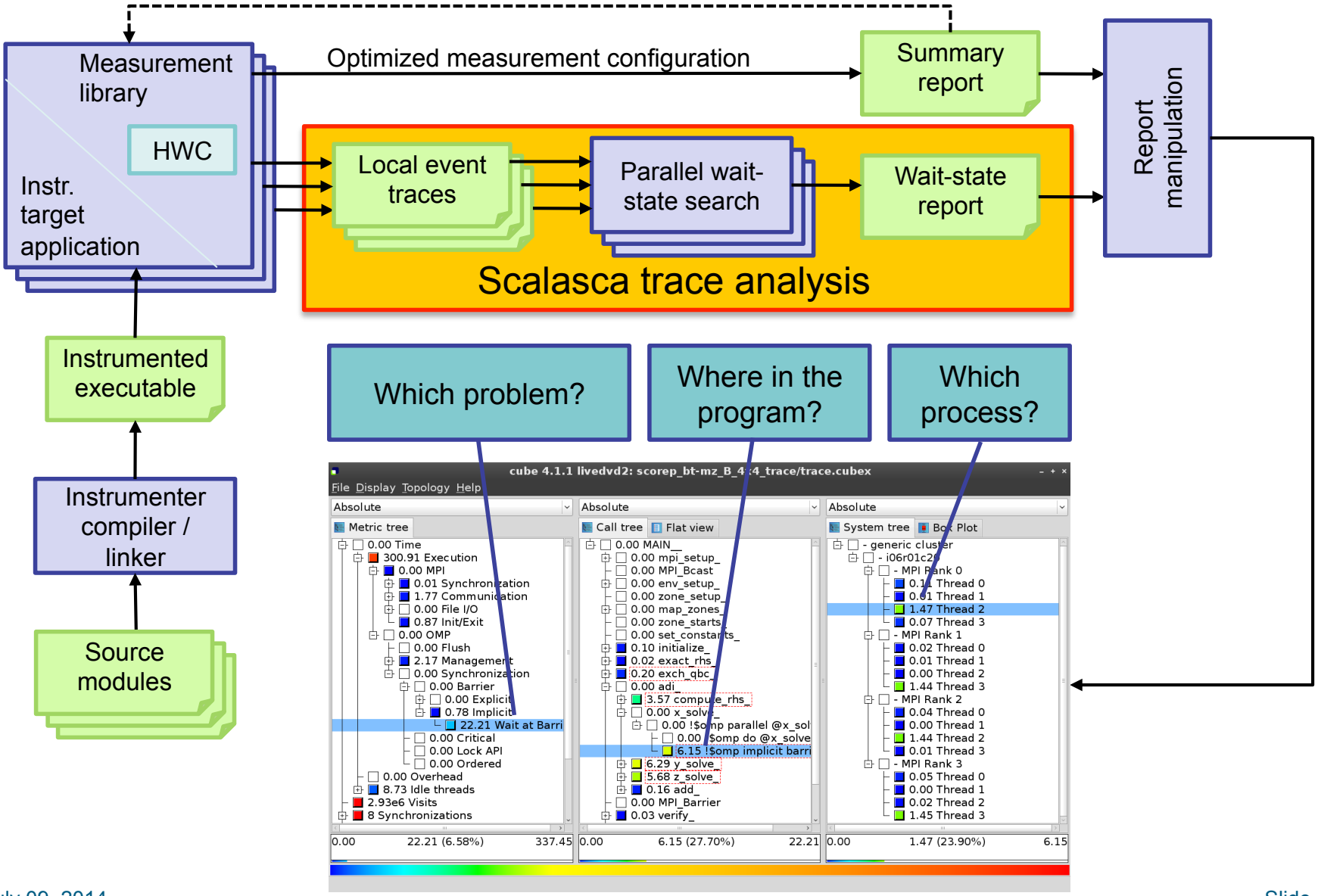
Uses Score-P instrumenter & measurement libraries

- 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



Scalasca command

One command for (almost) everything…

```
% scalasca
Scalasca 2.0
Toolset for scalable performance analysis of large-scale applications
usage: scalasca [-v][-n][c] {action}
    1. prepare application objects and executable for measurement:
       scalasca -instrument <compile-or-link-command> # skin (using scorep)
    2. run application under control of measurement system:
       scalasca -analyze <application-launch-command> # scan
    3. interactively explore measurement analysis report:
       scalasca -examine <experiment-archive|report>  # square

   -v, --verbose      enable verbose commentary
   -n, --dry-run      show actions without taking them
   -c, --show-config  show configuration and exit
```

- The ‘scalasca -instrument’ command is deprecated and only provided for backwards compatibility with Scalasca 1.x.
- Recommended: use Score-P instrumenter directly

Scalasca compatibility command: *skin*

Scalasca application instrumenter

```
% skin
Scalasca 2.0: application instrumenter using scorep
usage: skin [-v] [-comp] [-pdt] [-pomp] [-user] <compile-or-link-cmd>
   -comp={all|none|...}: routines to be instrumented by compiler
           (... custom instrumentation specification for compiler)
   -pdt:  process source files with PDT instrumenter
   -pomp: process source files for POMP directives
   -user: enable EPIK user instrumentation API macros in source code
   -v:    enable verbose commentary when instrumenting

   --*:   options to pass to Score-P instrumenter
```

- Provides compatibility with Scalasca 1.x
- Recommended: use Score-P instrumenter directly

Scalasca convenience command: *scan*

Scalasca measurement collection & analysis nexus

```
% scan
Scalasca 2.0: measurement collection & analysis nexus
usage: scan {options} [launchcmd [launchargs]] target [targetargs]
      where {options} may include:
  -h    Help: show this brief usage message and exit.
  -v    Verbose: increase verbosity.
  -n    Preview: show command(s) to be launched but don't execute.
  -q    Quiescent: execution with neither summarization nor tracing.
  -s    Summary: enable runtime summarization. [Default]
  -t    Tracing: enable trace collection and analysis.
  -a    Analyze: skip measurement to (re-)analyze an existing trace.
  -e exptdir   : Experiment archive to generate and/or analyze.
                 (overrides default experiment archive title)
  -f filtfile  : File specifying measurement filter.
  -l lockfile  : File that blocks start of measurement.
```

Automatic measurement configuration

scan configures Score-P measurement by setting some environment variables automatically

- e.g., experiment title, profiling/tracing mode, filter file, …
- Precedence order:
  - Command-line arguments
  - Environment variables already set
  - Automatically determined values

Also, **scan** includes consistency checks and prevents corrupting existing experiment directories

For tracing experiments, after trace collection completes then automatic parallel trace analysis is initiated

- uses identical launch configuration to that used for measurement (i.e., the same allocated compute resources)

BT-MZ summary measurement

Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_sum
% OMP_NUM_THREADS=4 scan mpiexec -np 4 ./bt-mz_W.4
S=C=A=N: Scalasca 2.0 runtime summarization
S=C=A=N: ./scorep_bt-mz_W_4x4_sum experiment archive
S=C=A=N: Thu Sep 13 18:05:17 2012: Collect start
mpiexec -np 4 ./bt-mz_W.4

 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark

 Number of zones:   8 x   8
 Iterations: 200    dt:   0.000300
 Number of active processes:     4

           [... More application output ...]

S=C=A=N: Thu Sep 13 18:05:39 2012: Collect done (status=0) 22s
S=C=A=N: ./scorep_bt-mz_W_4x4_sum complete.
```

Creates experiment directory **./scorep_bt-mz_W_4x4_sum**

BT-MZ summary analysis report examination

Score summary analysis report

```
% square -s  scorep_bt-mz_W_4x4_sum
INFO: Post-processing runtime summarization result...
INFO: Score report written to ./scorep_bt-mz_W_4x4_sum/scorep.score
```

Post-processing and interactive exploration with CUBE

```
% square  scorep_bt-mz_W_4x4_sum
INFO: Displaying ./scorep_bt-mz_W_4x4_sum/summary.cubex...

                  [GUI showing summary analysis report]
```

The post-processing derives additional metrics and generates a structured metric hierarchy

Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
    - Score-P
    - Scalasca
    - CUBE
    - Vampir
    - TAU
  - Use cases
- Summary

CUBE is available at http://www.scalasca.org/,

get support via scalasca@fz-juelich.de

CUBE

Parallel program analysis report exploration tools

- Libraries for XML report reading & writing
- Algebra utilities for report processing
- GUI for interactive analysis exploration
  - requires Qt4

Originally developed as part of Scalasca toolset

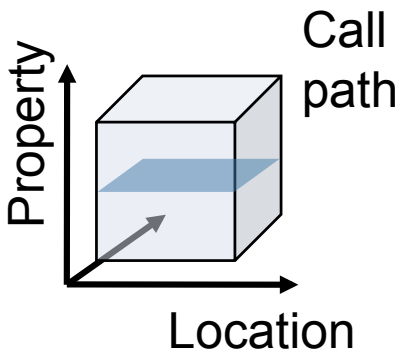
Now available as a separate component

- Can be installed independently of Score-P, e.g., on laptop or desktop
- Latest release: CUBE 4.2.3 (June 2014)

Analysis presentation and exploration

Representation of values (severity matrix) on three hierarchical axes

- Performance property (metric)
- Call path (program location)
- System location (process/thread)

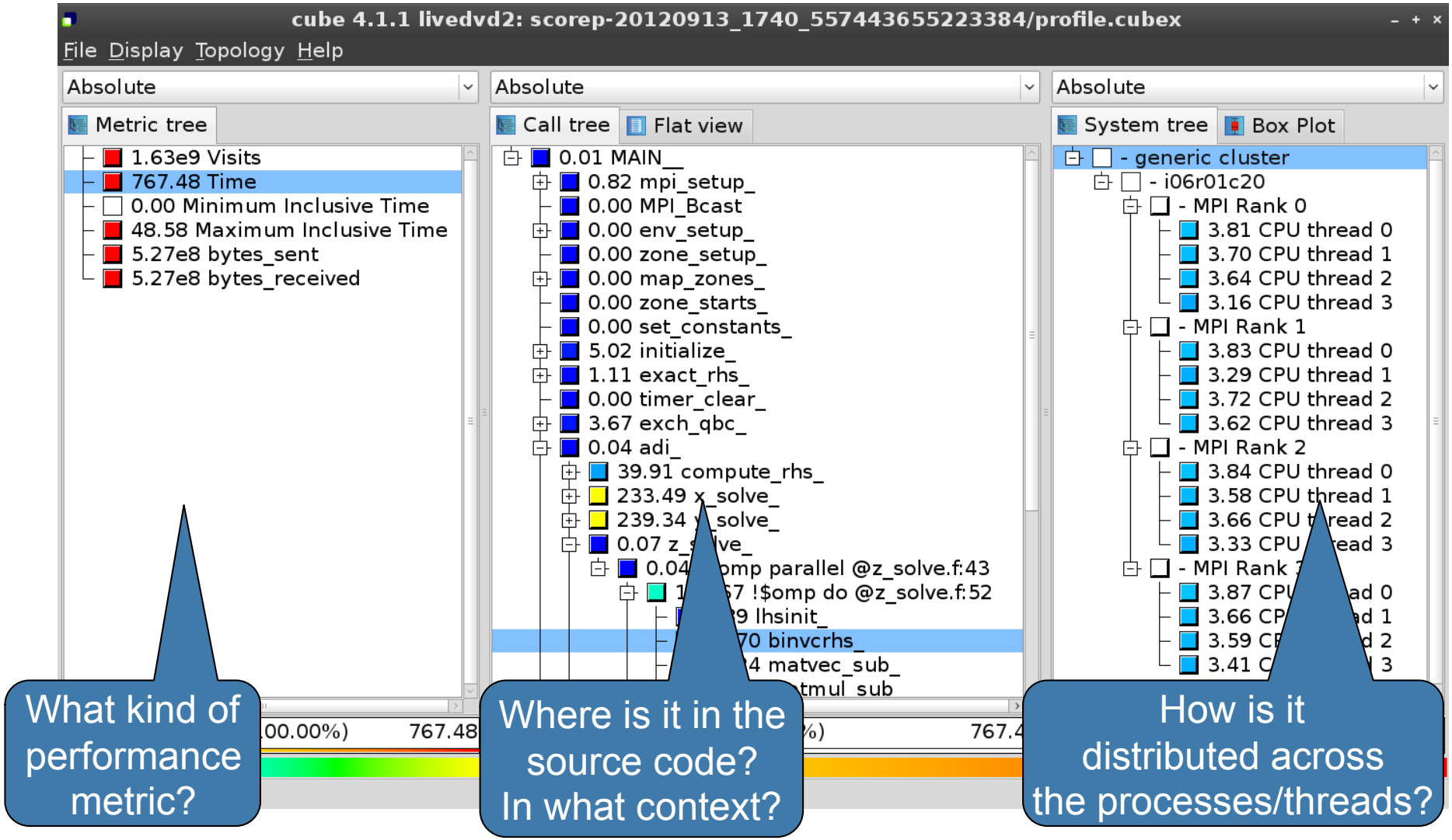


Three coupled tree browsers

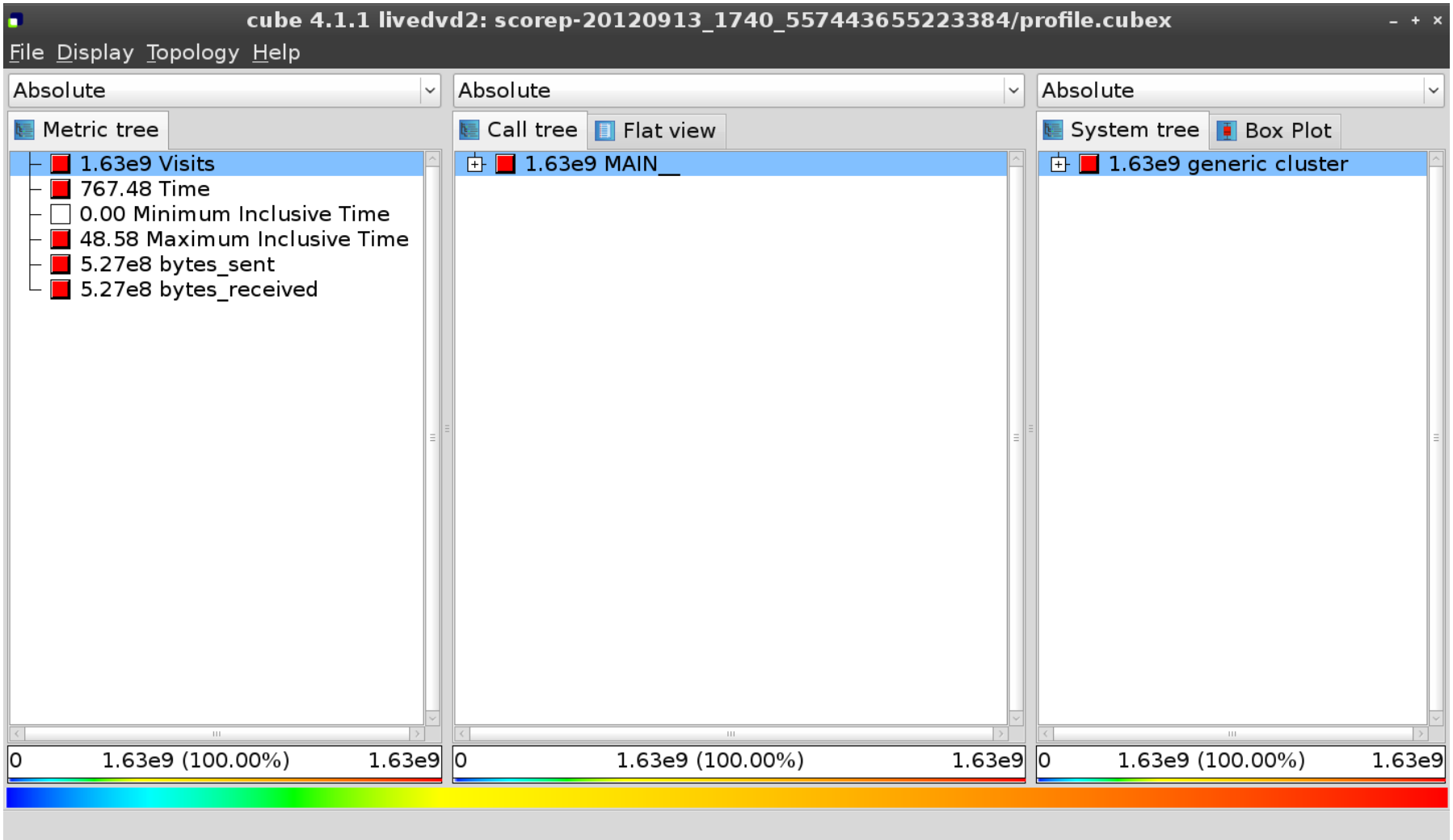
CUBE displays severities

- As value: for precise comparison
- As colour: for easy identification of hotspots
- Inclusive value when closed & exclusive value when expanded
- Customizable via display modes

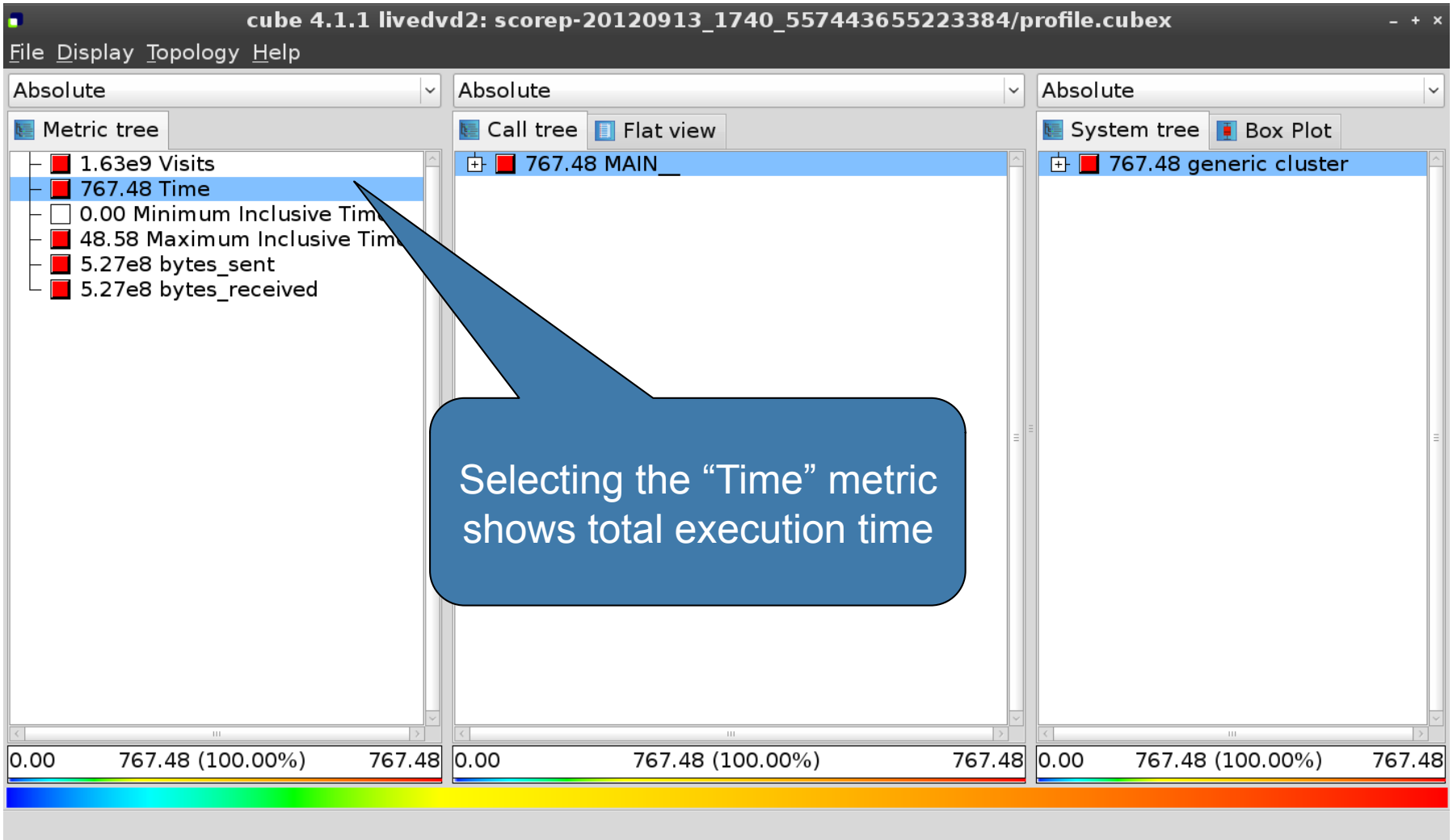
Analysis presentation



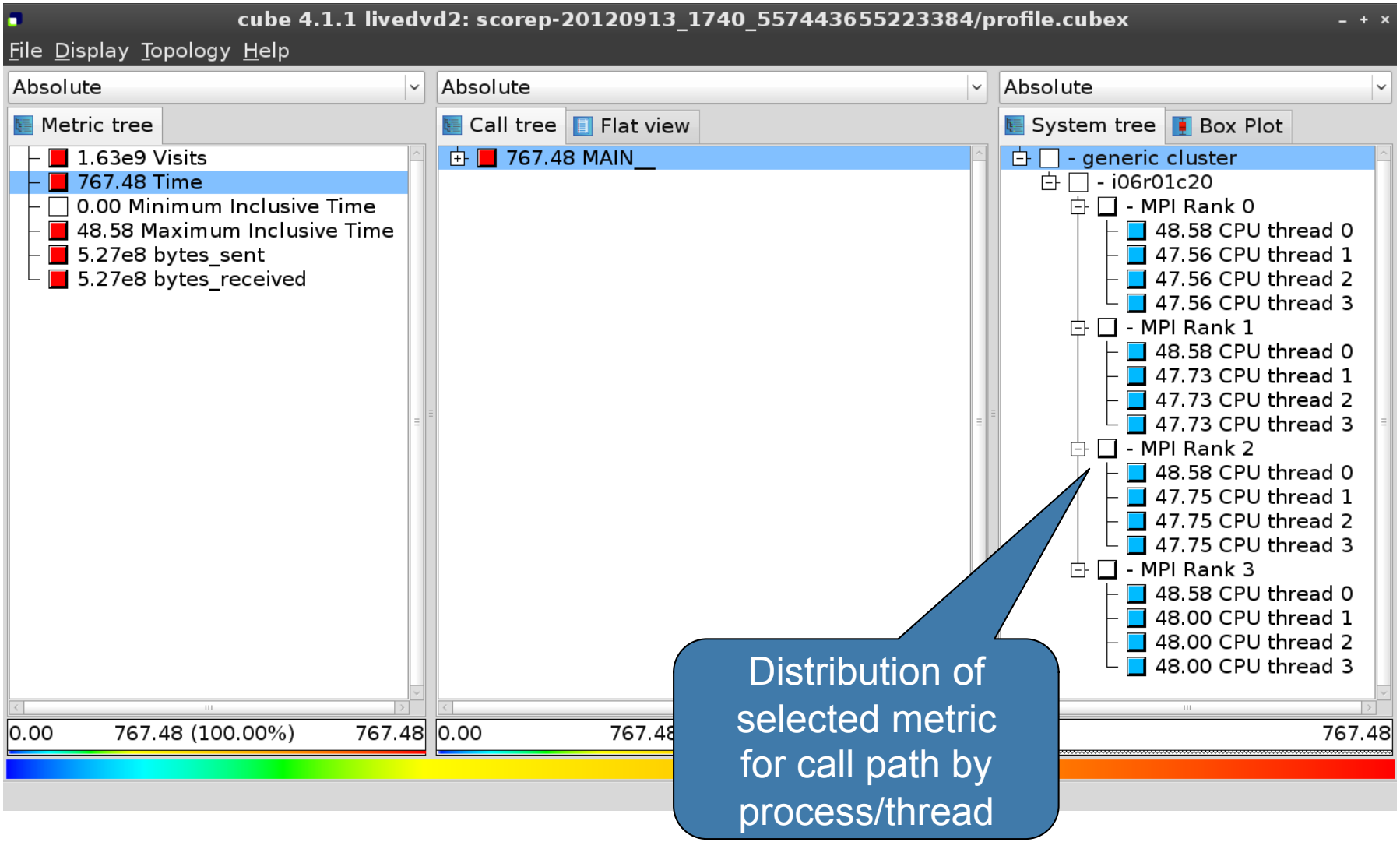
Analysis report exploration (opening view)



Metric selection



Expanding the system tree



Selecting a call path

cube 4.1.1 livedvd2: scorep-20120913_1740_557443655223384/profile.cubex

File Display Topology Help

Absolute | Absolute | Absolute

Metric tree | Call tree | Flat view | System tree | Box Plot

- 1.63e9 Visits
- 767.48 Time
- 0.00 Minimum Inclusive Time
- 48.58 Maximum Inclusive Time
- 5.27e8 bytes_sent
- 5.27e8 bytes_received

0.00 767.48 (100.00%) 767.48

- 0.01 MAIN__
  - 0.82 mpi_setup_
  - 0.00 MPI_Bcast
  - 0.00 env_setup_
  - 0.00 zone_setup_
  - 0.00 map_zones_
  - 0.00 zone_starts_
  - 0.00 set_constants_
  - 5.02 initialize_
  - 1.11 exact_rhs_
  - 0.00 timer_clear_
  - 3.67 exch_qbc_
  - 0.04 adi_
    - 39.91 compute_rhs_
    - 233.49 x_solve_
    - 239.34 y_solve_
    - 0.07 z_solve_
      - 0.04 !$omp parallel @z_solve.f:43
        - 100.67 !$omp do @z_solve.f:52
          - 2.89 lhsinit_
          - 57.70 binvcrhs_
          - 27.24 matvec_sub_
          - 36.11 matmul_sub

0.00 57.70 (7.52%) 57.70

- generic cluster
  - i06r01c20
    - MPI Rank 0
      - 3.81 CPU thread 0
      - 3.70 CPU thread 1
      - 3.64 CPU thread 2
      - 3.16 CPU thread 3
    - MPI Rank 1
      - 3.83 CPU thread 0
      - 3.29 CPU thread 1
      - 3.72 CPU thread 2
      - 3.62 CPU thread 3
    - MPI Rank 2
      - 3.84 CPU thread 0
      - 3.58 CPU thread 1
      - 3.66 CPU thread 2
      - 3.33 CPU thread 3
    - MPI Rank 3
      - 3.87 CPU thread 0
      - 3.66 CPU thread 1
      - 3.59 CPU thread 2
      - 3.41 CPU thread 3

Selection updates metric values shown in columns to right

Source-code view via context menu

cube 4.1.1 livedvd2: scorep-20120913_1740_557443655223384/profile.cubex

File Display Topology Help

Absolute | Absolute | Absolute

Metric tree | Call tree | Flat view | System tree | Box Plot

- 1.63e9 Visits
- 767.48 Time
- 0.00 Minimum Inclusive Time
- 48.58 Maximum Inclusive Time
- 5.27e8 bytes_sent
- 5.27e8 bytes_received

- 0.01 MAIN__
  - 0.82 mpi_setup_
  - 0.00 MPI_Bcast
  - 0.00 env_setup_
  - 0.00 zone_setup_
  - 0.00 map_zones_
  - 0.00 zone_starts_
  - 0.00 set_constants_
  - 5.02 initialize_
  - 1.11 exact_rhs_
  - 0.00 timer_clear_
  - 3.67 exch_qbc_
  - 0.04 adi_
    - 39.91 compute_rhs_
    - 233.49 x_solve_
    - 239.34 y_solve_
    - 0.07 z_solve_
      - 0.04 !$omp paralle
        - 100.67 !$omp d
          - 2.89 lhsinit_
          - 57.70 binvcrhs_
          - matvec_sub_
          - matmul_sub

- generic cluster
  - i06r01c20
    - MPI Rank 0
      - 3.81 CPU thread 0
      - 3.70 CPU thread 1
      - 3.64 CPU thread 2
      - 3.16 CPU thread 3
    - MPI Rank 1
      - U thread 0
      - U thread 1
      - U thread 2
      - U thread 3
      - 2
      - 3.84 CPU thread 0
      - 3.58 CPU thread 1
      - 3.66 CPU thread 2
      - 3.33 CPU thread 3
    - MPI Rank 3
      - 3.87 CPU thread 0
      - 3.66 CPU thread 1
      - 3.59 CPU thread 2
      - 3.41 CPU thread 3

Call site
Called region
Expand/collapse
Hiding
Cut call tree
Find items
Find Next
Clear found items
Copy to clipboard
Min/max values

Info
Online description
Location
Source code

0.00 767.48 (100.00%) 76 | %) 767.48 | 0.00 57.70

Shows the source code of the clicke

Right-click opens context menu

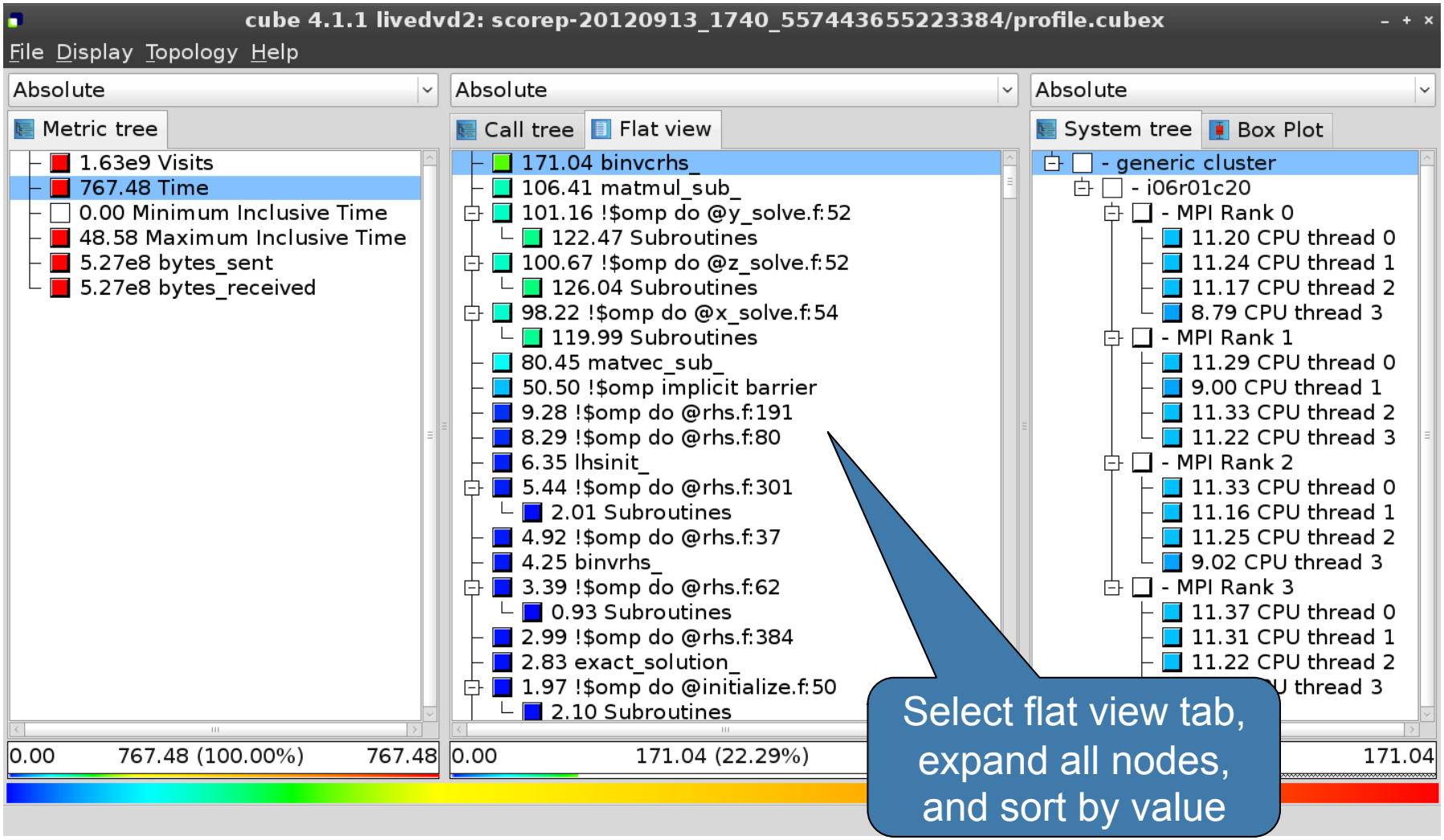
Source-code view

/home/geimer/Projects/Tests/NPB3.3-MZ-MPI/BT-MZ/solve_subs.f

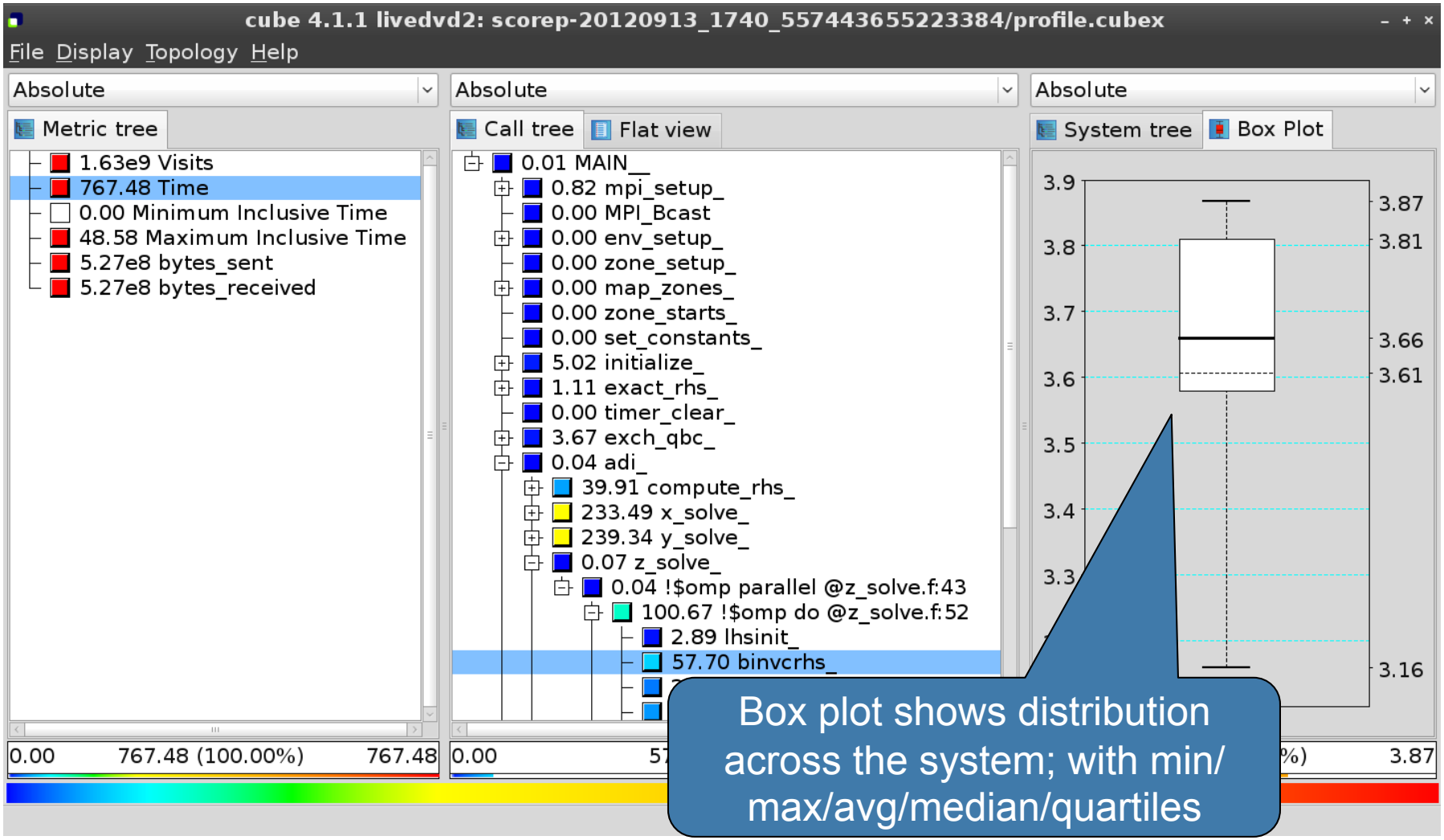
```
      subroutine binvcrhs( lhs,c,r )

c---------------------------------------------------------------------
c---------------------------------------------------------------------

c---------------------------------------------------------------------
c
c---------------------------------------------------------------------

      implicit none

      double precision pivot, coeff, lhs
      dimension lhs(5,5)
      double precision c(5,5), r(5)

c---------------------------------------------------------------------
c
c---------------------------------------------------------------------

      pivot = 1.00d0/lhs(1,1)
      lhs(1,2) = lhs(1,2)*pivot
      lhs(1,3) = lhs(1,3)*pivot
      lhs(1,4) = lhs(1,4)*pivot
      lhs(1,5) = lhs(1,5)*pivot
      c(1,1) = c(1,1)*pivot
      c(1,2) = c(1,2)*pivot
      c(1,3) = c(1,3)*pivot
      c(1,4) = c(1,4)*pivot
```

Read only | Save | Save as | Font... | Close

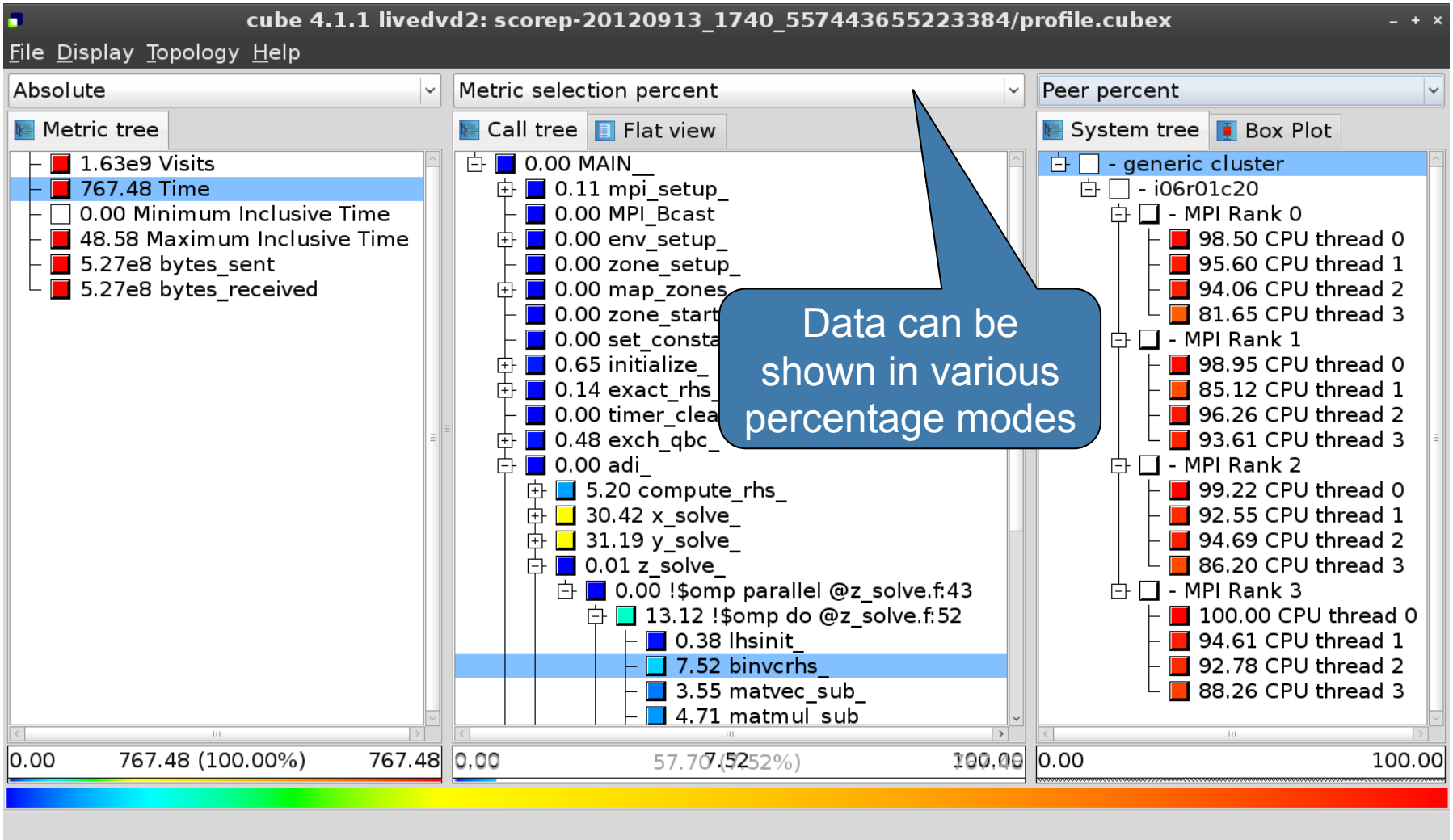
Flat profile view



Box plot view



Alternative display modes



Important display modes

Absolute

- Absolute value shown in seconds/bytes/counts

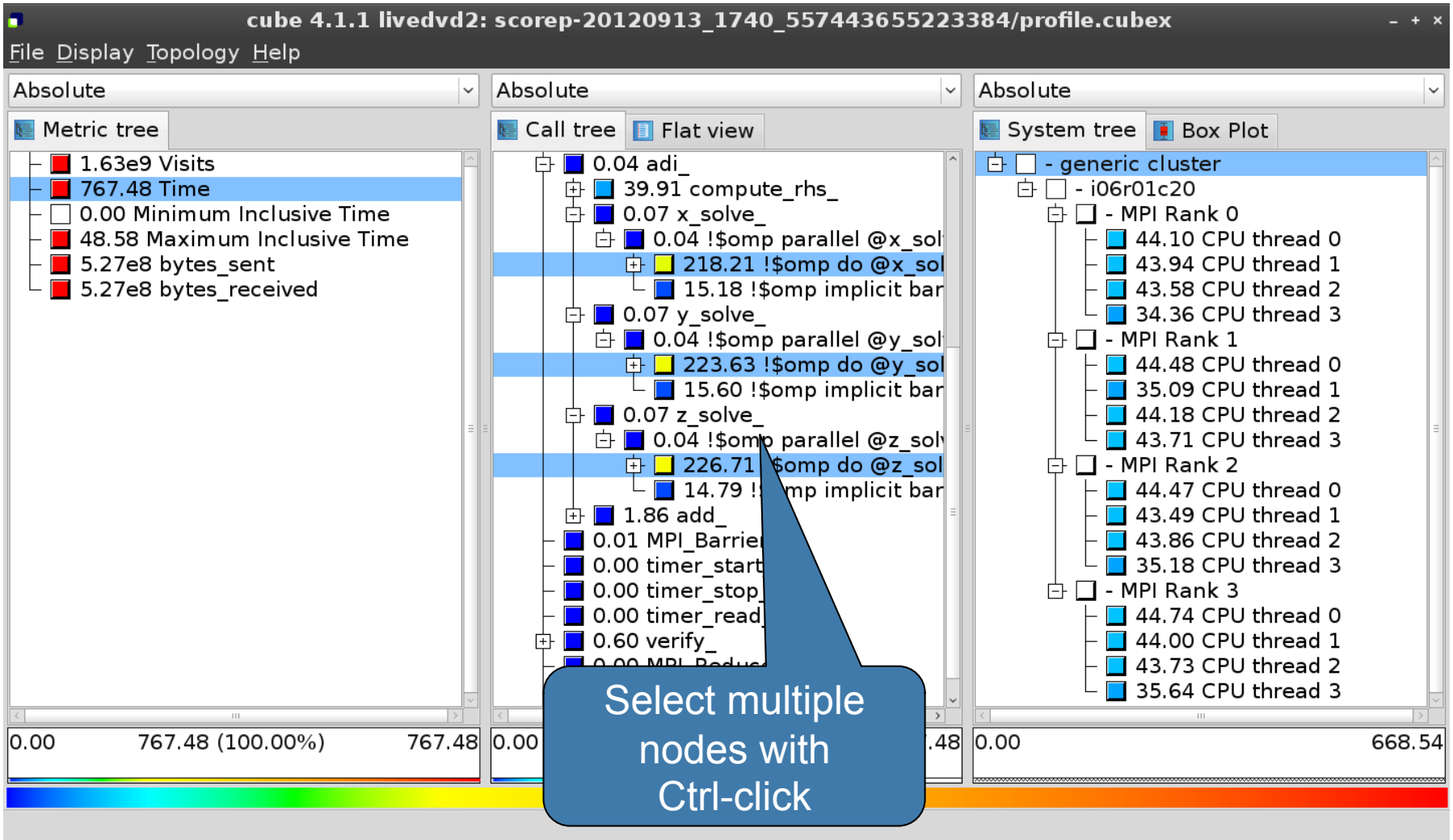
Selection percent

- Value shown as percentage w.r.t. the selected node “on the left“ (metric/call path)

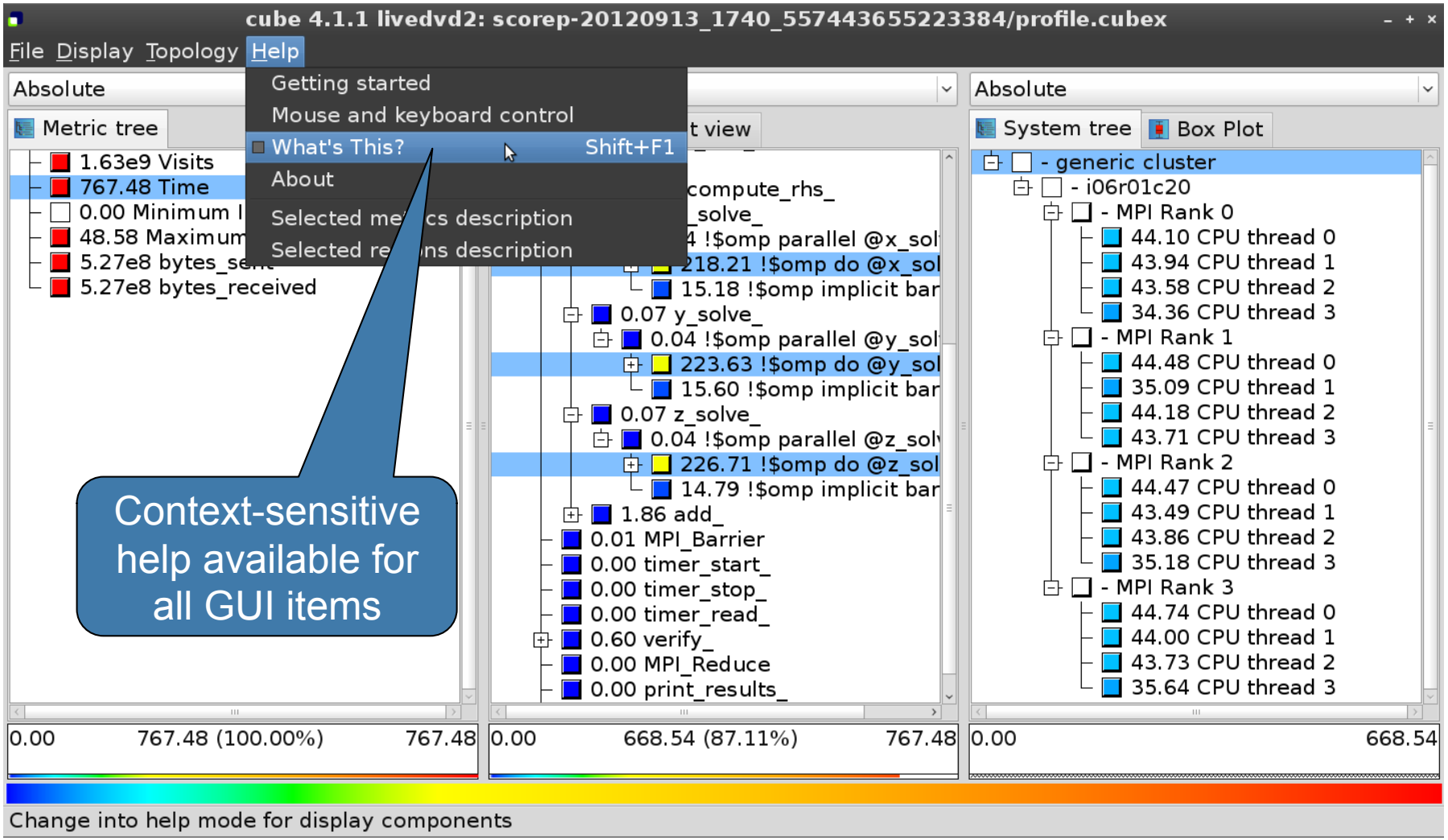
Peer percent (system tree only)

- Value shown as percentage relative to the maximum peer value

Multiple selection



Context-sensitive help



CUBE algebra utilities

Calculating difference of two reports

```
% cube_diff  scorep_bt-mz_W_4x4_sum/profile.cubex  cut.cubex
Writing diff.cubex... done.
```

Additional utilities for merging, calculating mean, etc.

- Default output of cube_utility is a new report utility.cubex

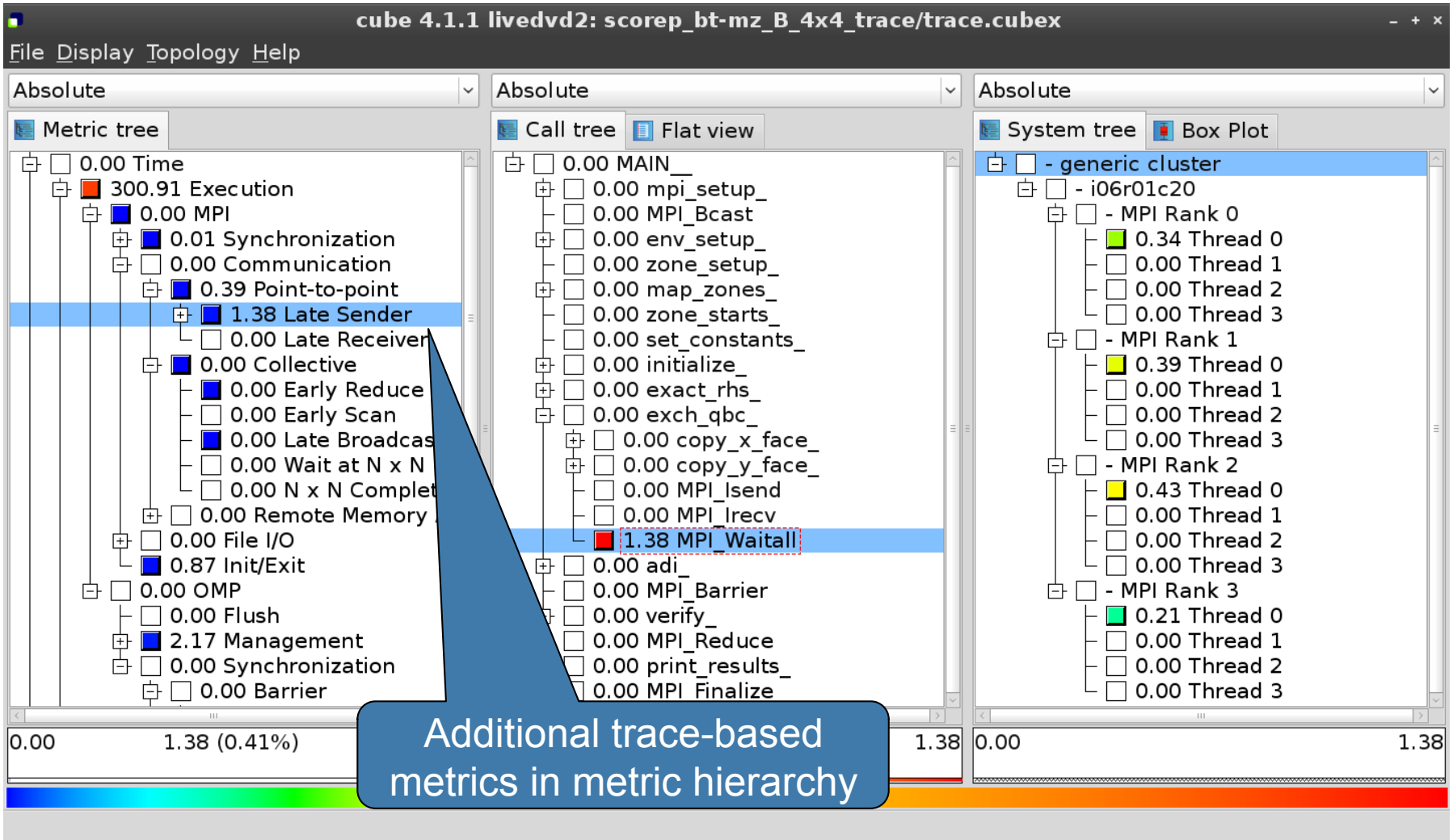
Further utilities for report scoring & statistics

Run utility with “-h” (or no arguments) for brief usage info

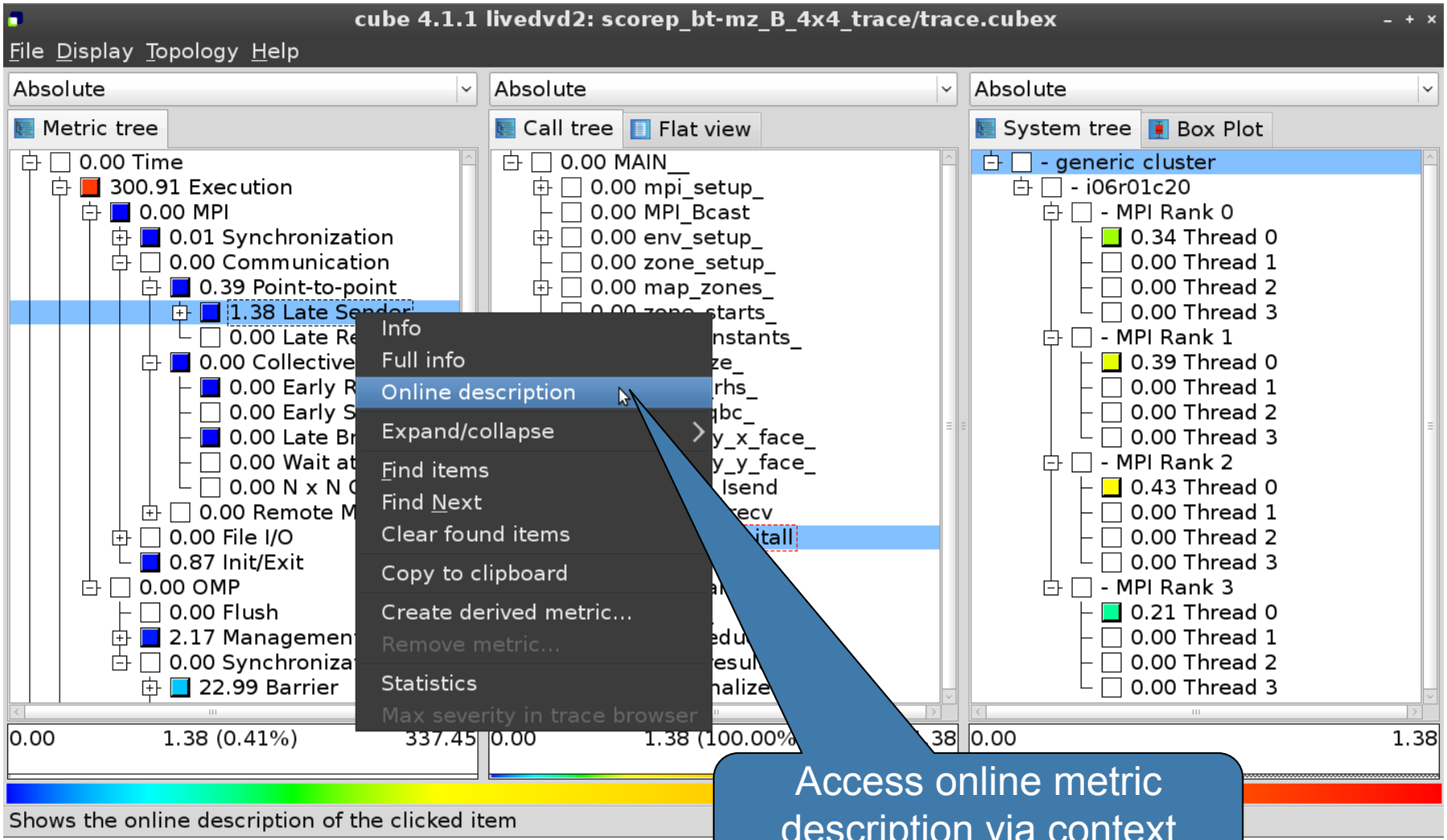
Cube - Demo

Analyzing a performance issue with Scalasca/ CUBE

Post-processed trace analysis report



Online metric description



Access online metric description via context menu

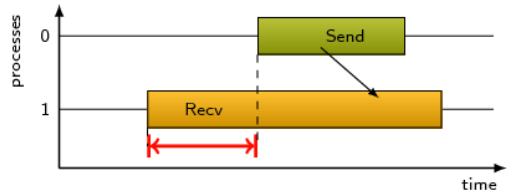
Online metric description

Performance properties

Late Sender Time

Description:

Refers to the time lost waiting caused by a blocking receive operation (e.g., `MPI_Recv` or `MPI_Wait`) that is posted earlier than the corresponding send operation.



If the receiving process is waiting for multiple messages to arrive (e.g., in an call to `MPI_Waitall`), the maximum waiting time is accounted, i.e., the waiting time due to the latest sender.

Unit:

Seconds

Diagnosis:

Try to replace `MPI_Recv` with a non-blocking receive `MPI_Irecv` that can be posted earlier, proceed concurrently with computation, and complete with a wait operation after the message is expected to have been sent. Try to post sends earlier, such that they are available when receivers need them. Note that outstanding messages (i.e., sent before the receiver is ready) will occupy internal message buffers, and that large numbers of posted receive buffers will also introduce message management overhead, therefore moderation is advisable.

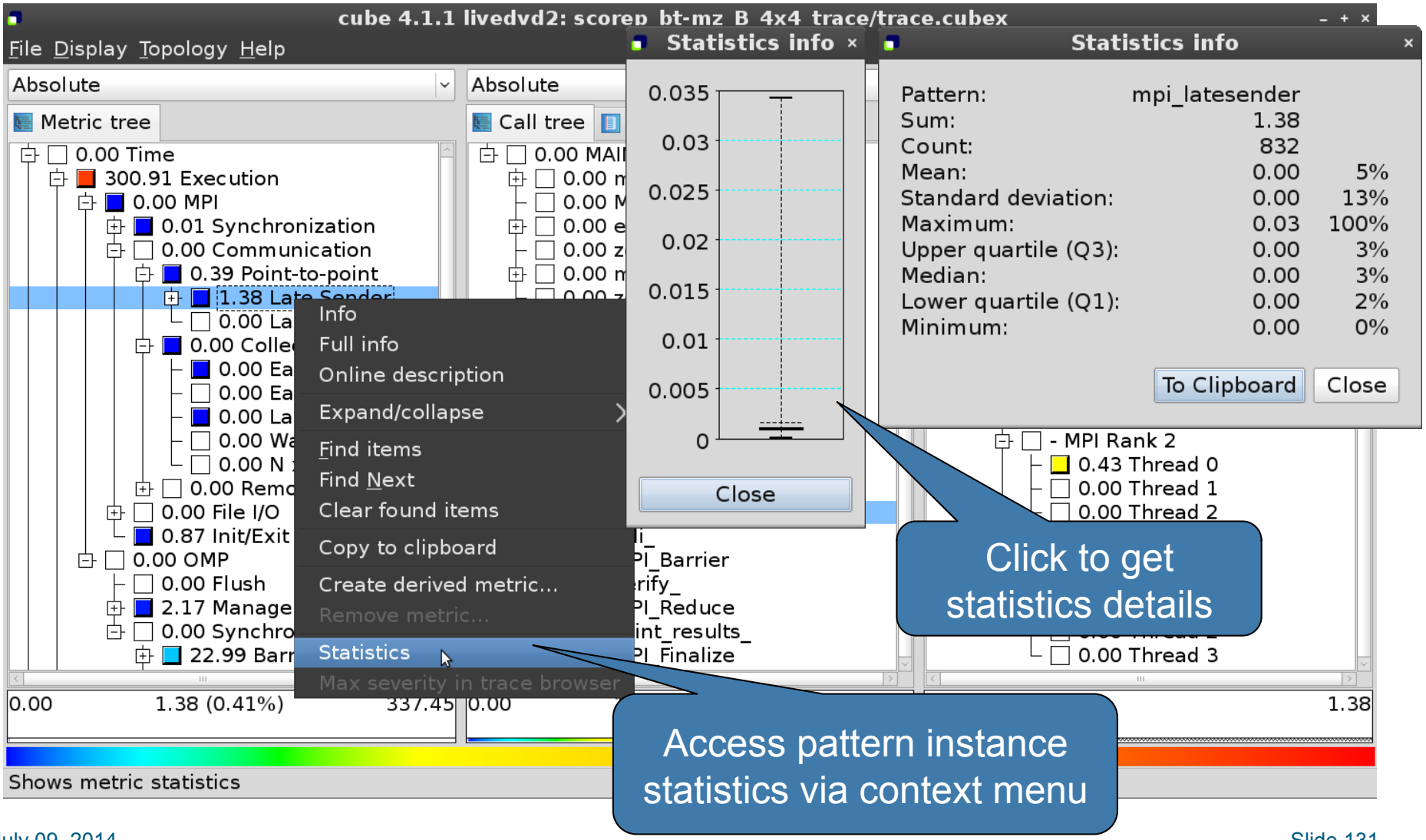
Parent:

MPI Point-to-point Communication Time

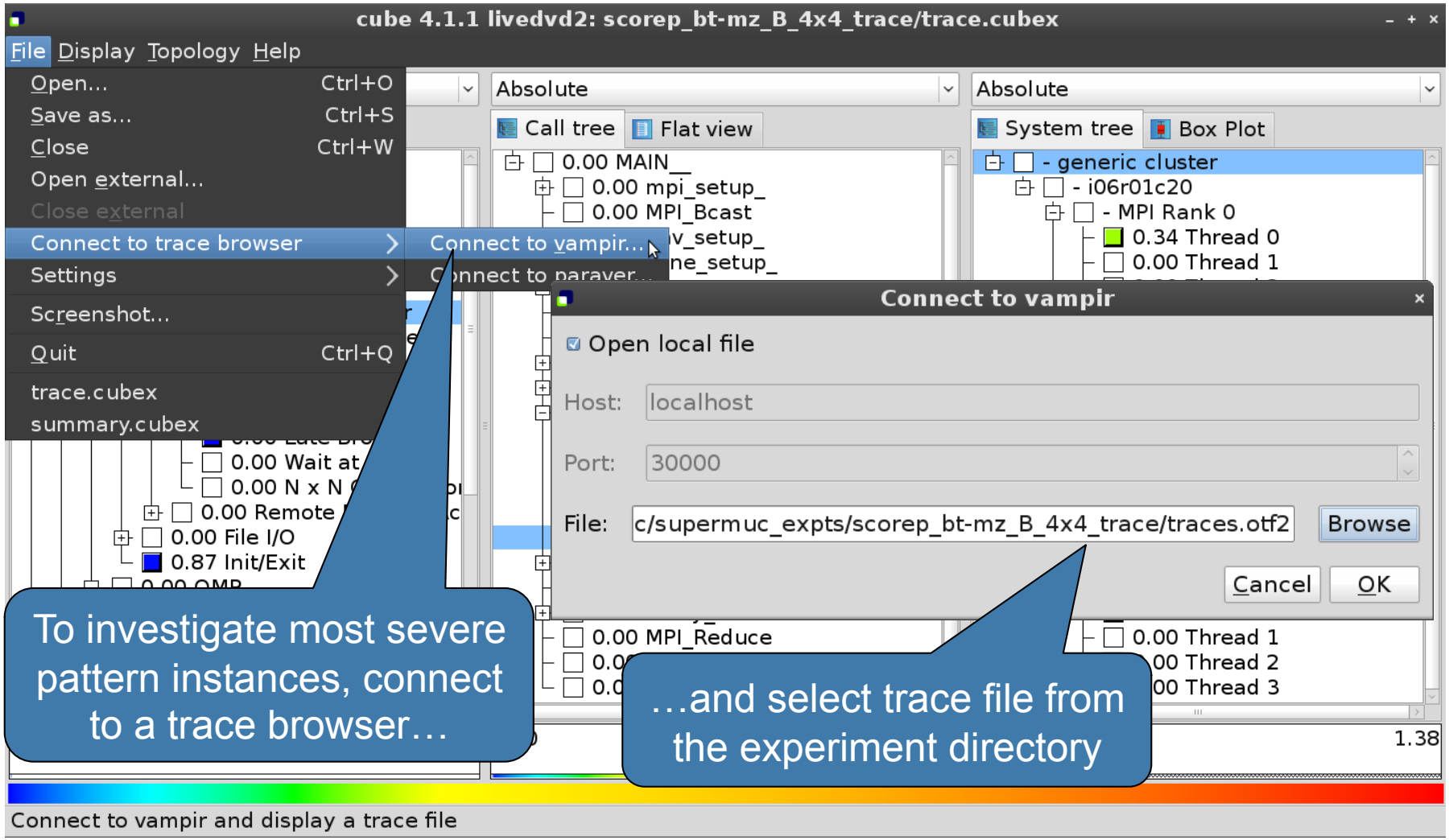
Children:

Close

Pattern instance statistics



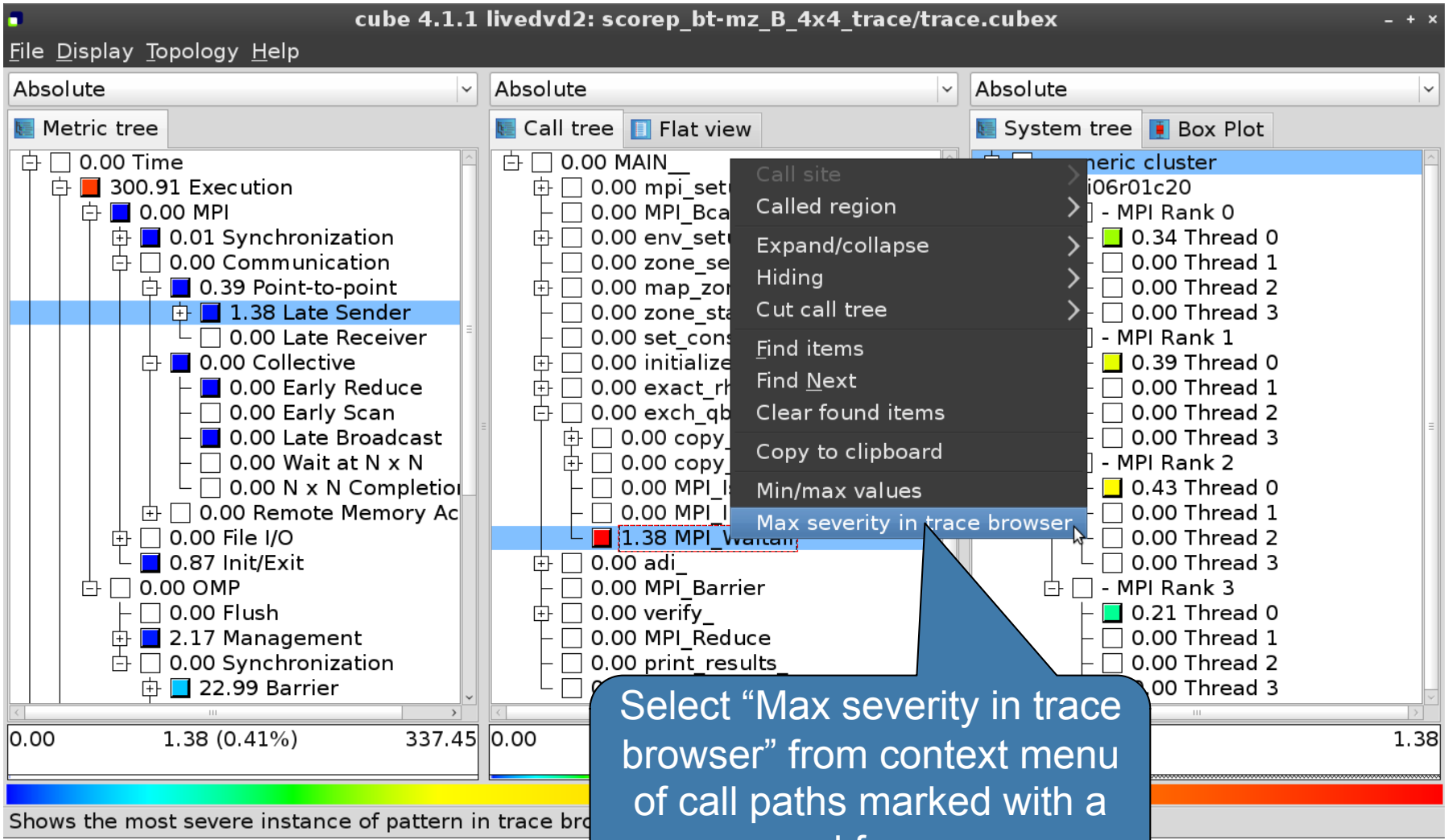
Connect to Vampir trace browser



To investigate most severe pattern instances, connect to a trace browser…

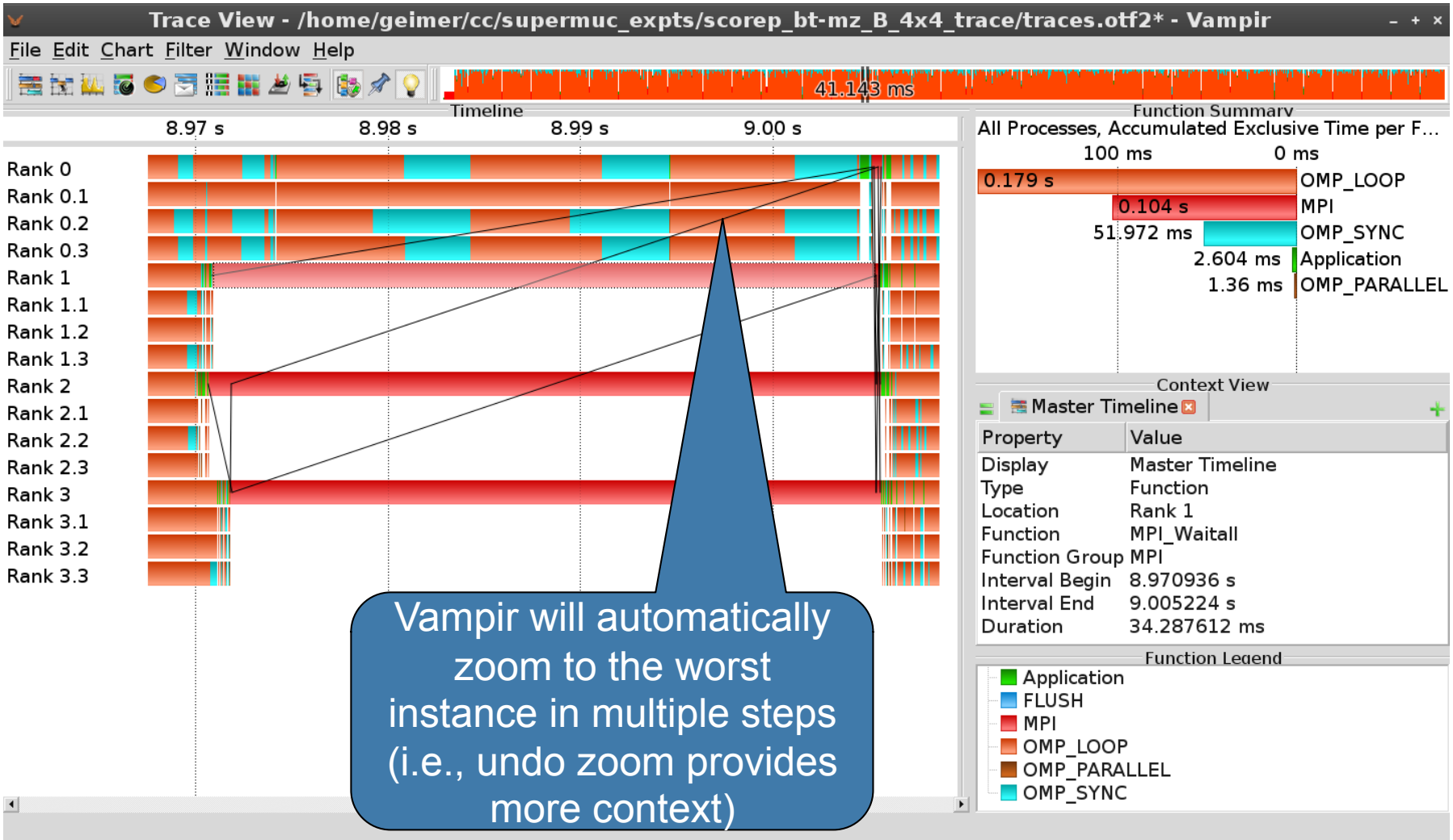
…and select trace file from the experiment directory

Show most severe pattern instances



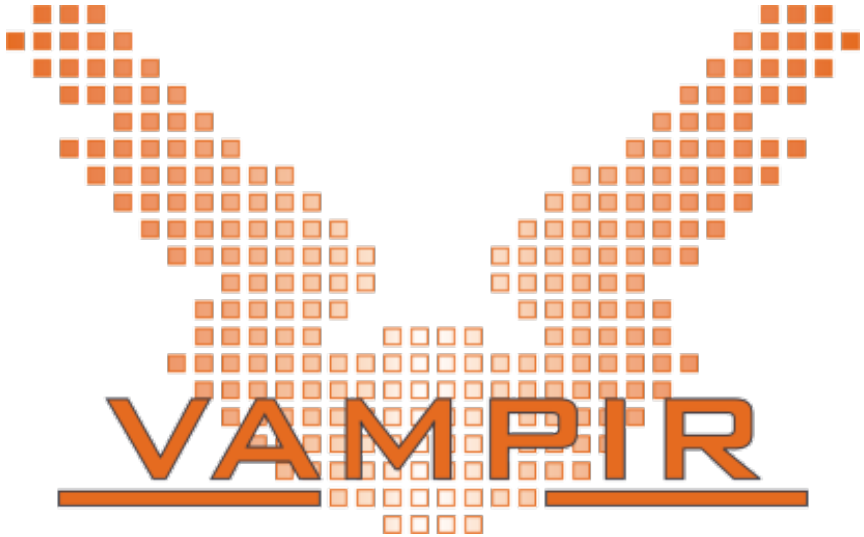
Select “Max severity in trace browser” from context menu of call paths marked with a red frame

Investigate most severe instance in Vampir



Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
    - Score-P
    - Scalasca
    - CUBE
    - Vampir
    - TAU
  - Use cases
- Summary



Vampir is available at http://www.vampir.eu,

get support via vampirsupport@zih.tu-dresden.de

Objectives

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?

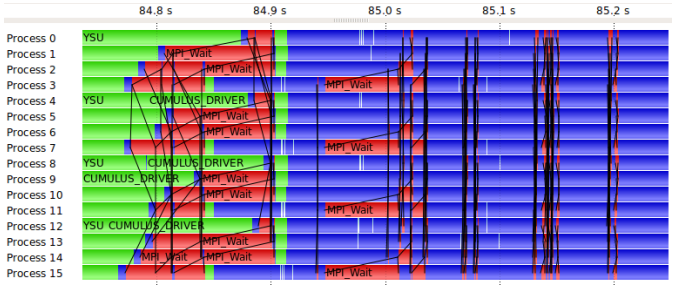
Event trace visualization with Vampir

Alternative and supplement to automatic analysis

- Show dynamic run-time behavior graphically at any level of detail
- Provide statistics and performance metrics

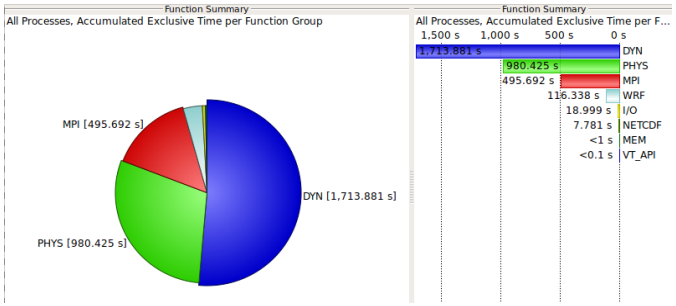
Timeline charts

➢ Show application activities and communication along a time axis



Summary charts

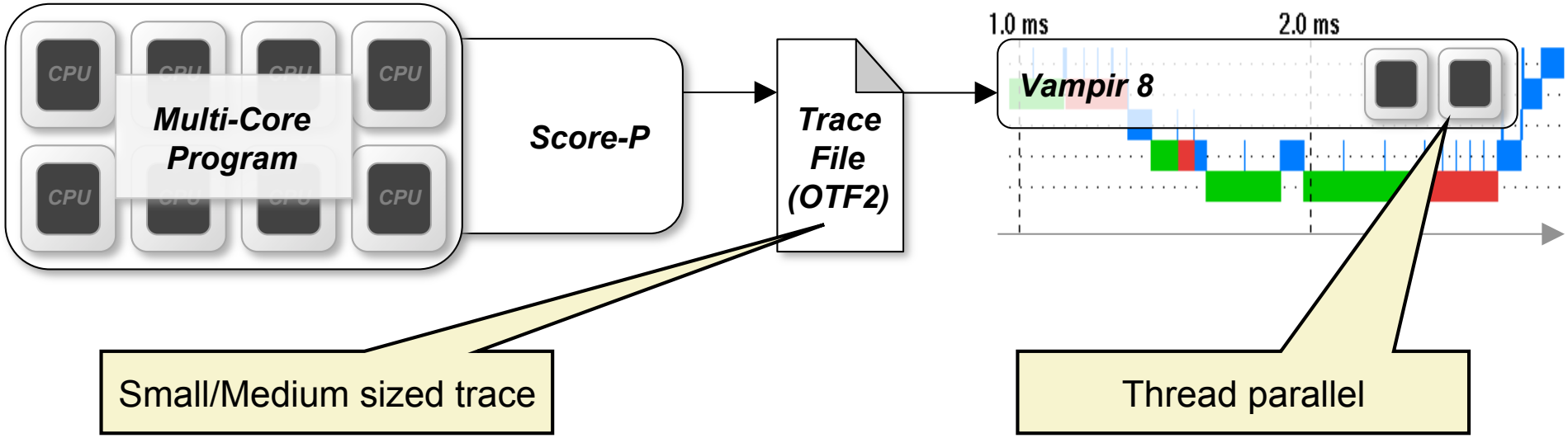
➢ Provide quantitative results for the currently selected time interval



Vampir – Visualization Modes (1)

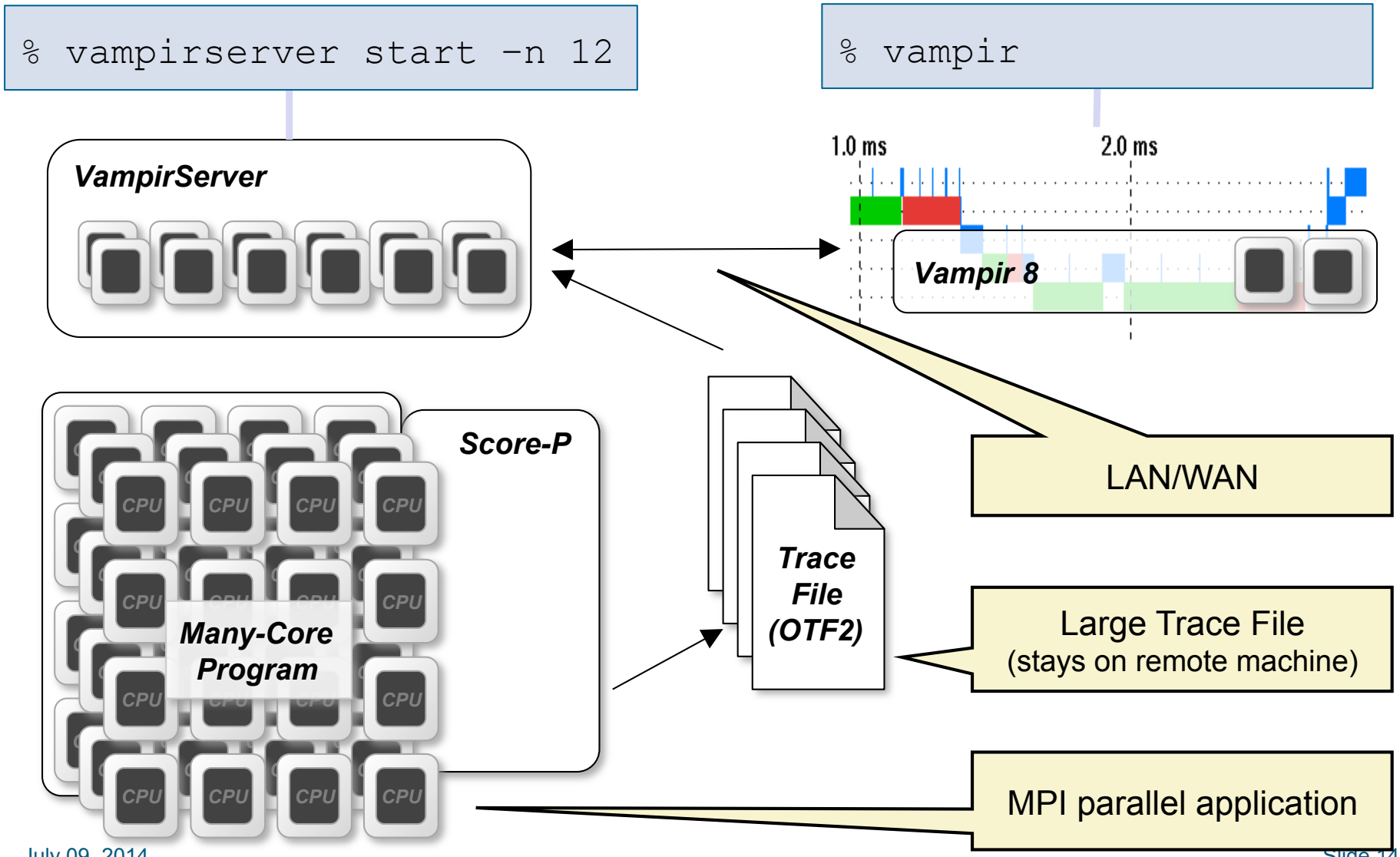
Directly on front end or local machine

```
% vampir
```



Vampir – Visualization Modes (2)

On local machine with remote VampirServer



Usage order of the Vampir Performance Analysis Toolset

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

The main displays of Vampir

Timeline Charts:

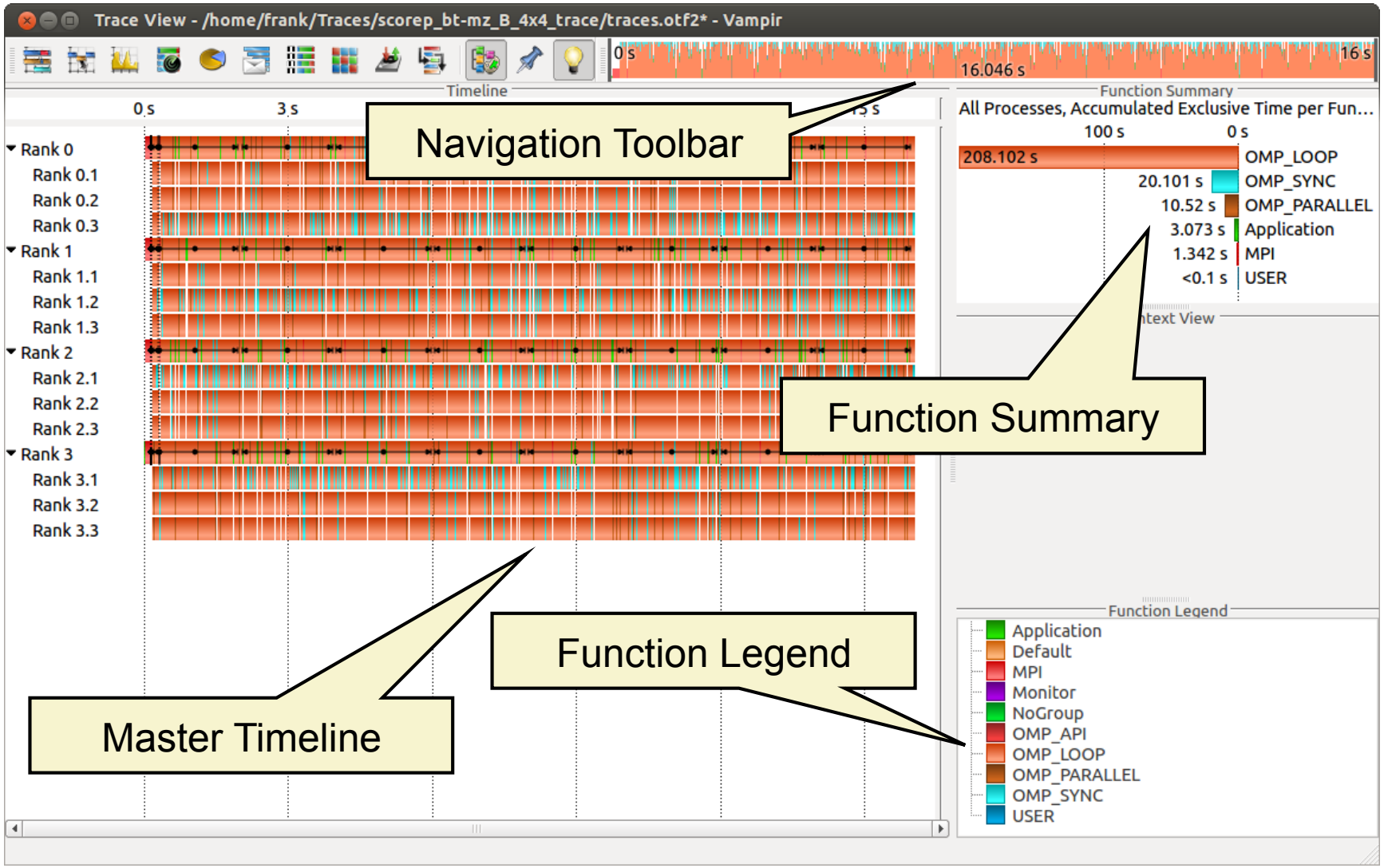
- Master Timeline
- Process Timeline
- Counter Data Timeline
- Performance Radar

Summary Charts:

- Function Summary
- Message Summary
- Process Summary
- Communication Matrix View

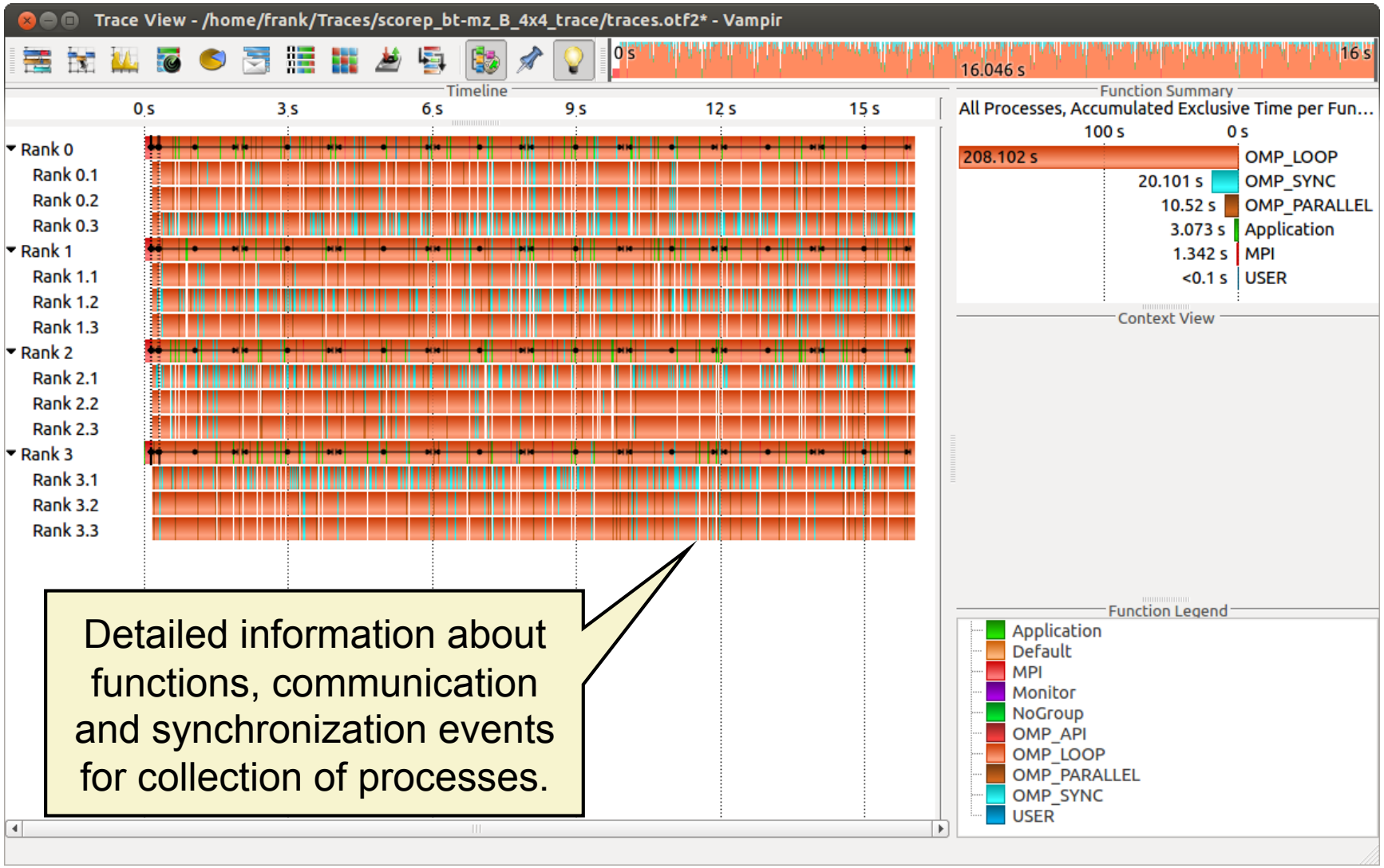
Vampir: example visualization

```
% vampir scorep_bt-mz_B_4x4_trace
```



Vampir: example visualization

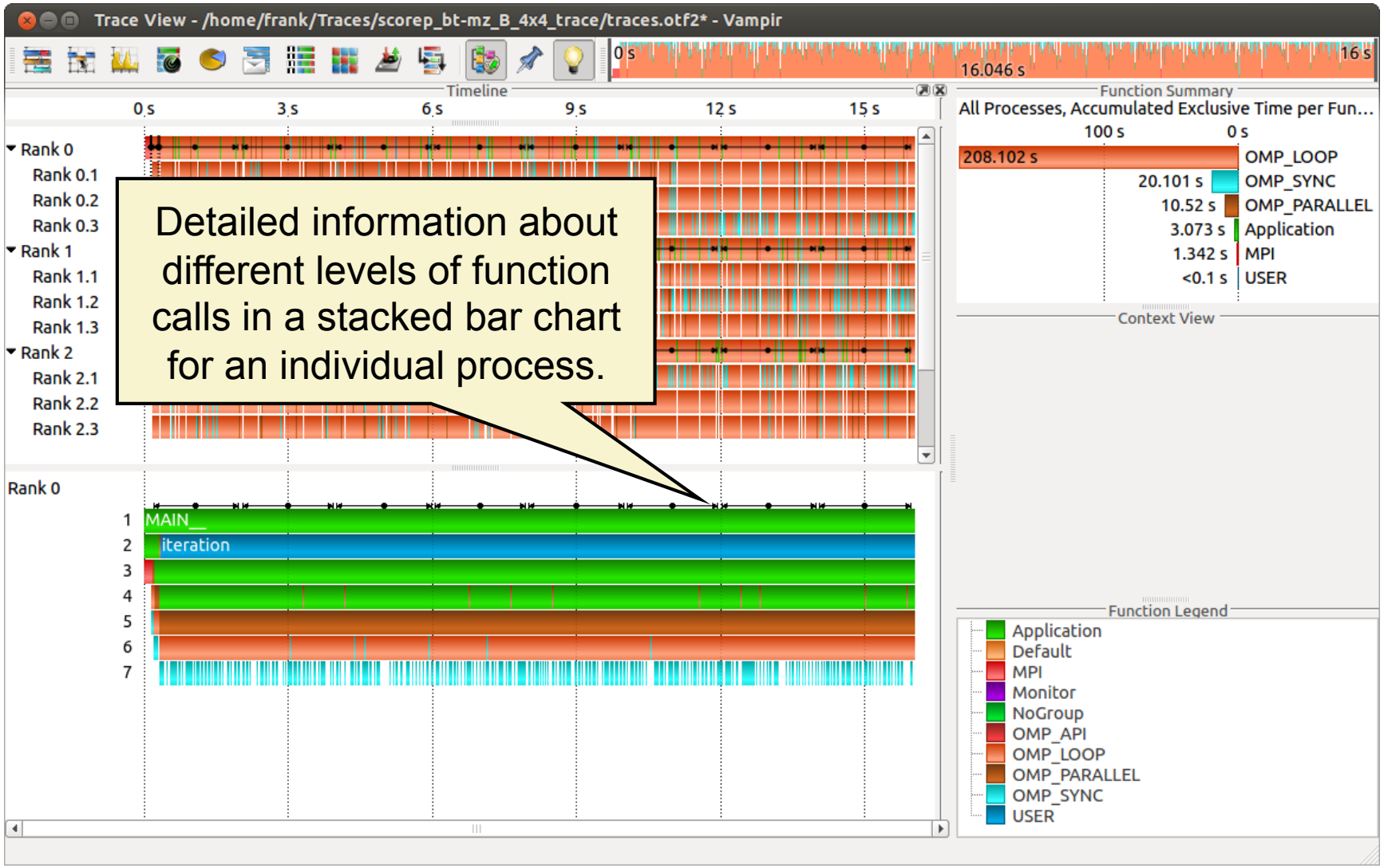
Master Timeline



Detailed information about functions, communication and synchronization events for collection of processes.

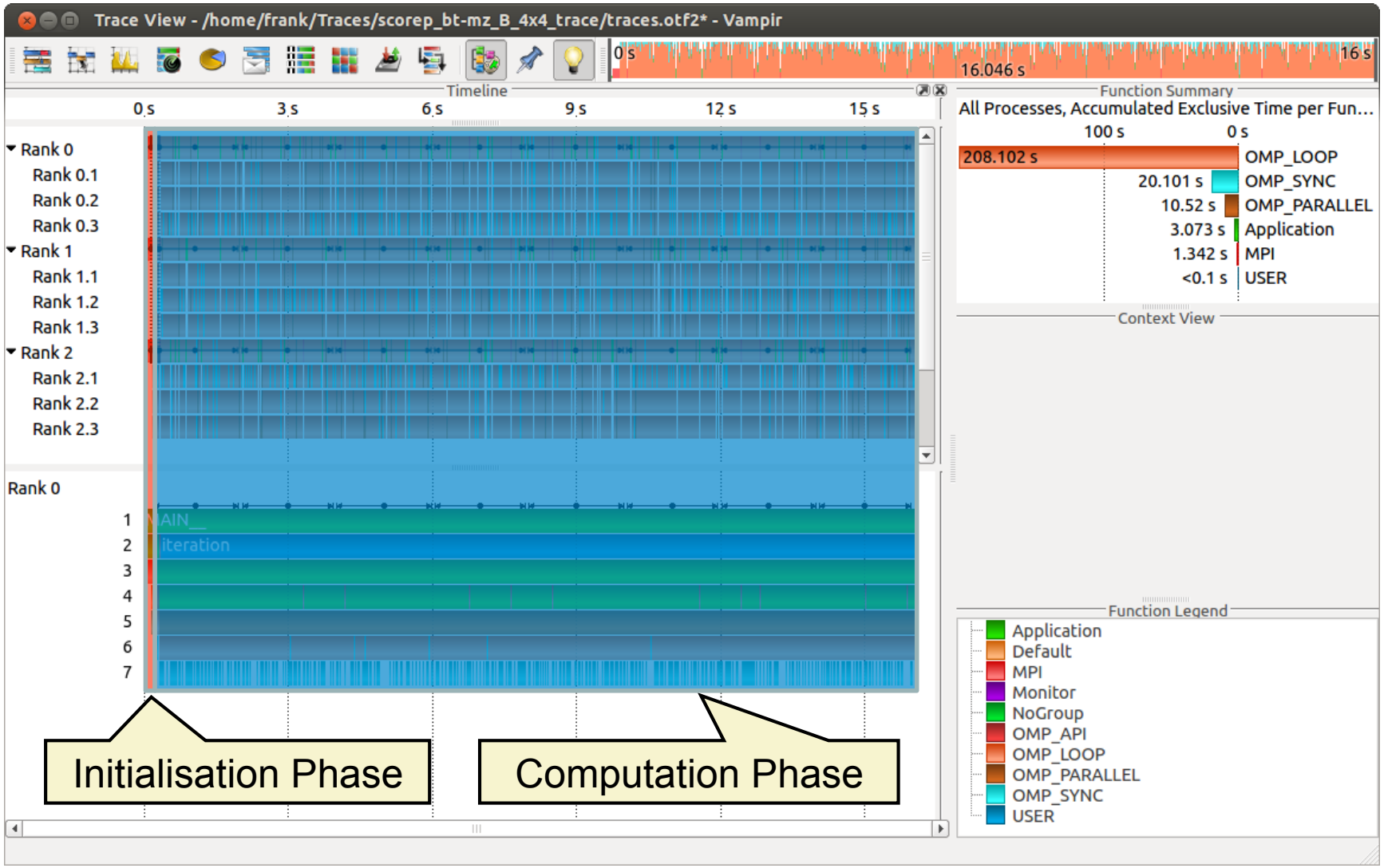
Vampir: example visualization

Process Timeline



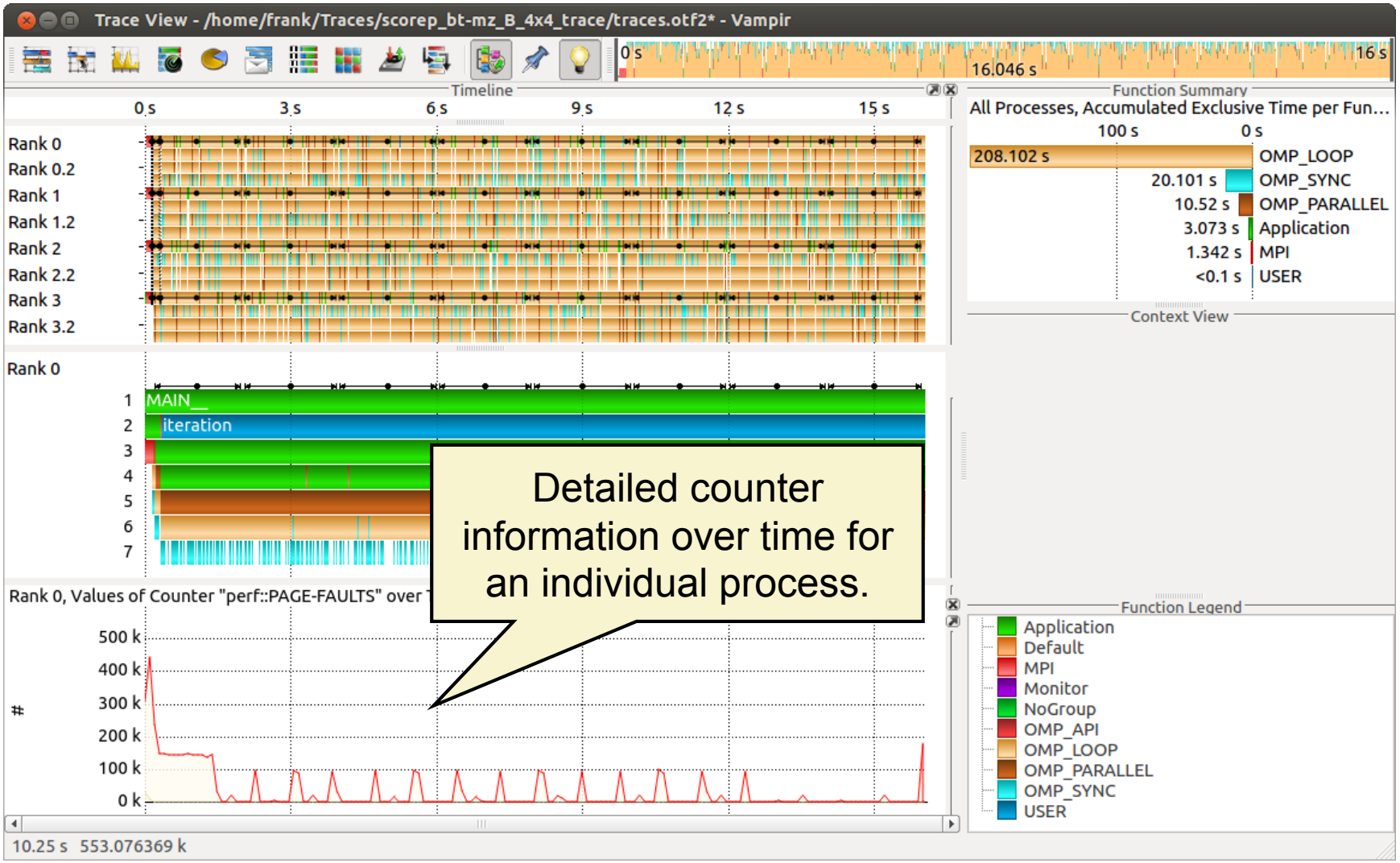
Vampir: example visualization

Typical program phases



Vampir: example visualization

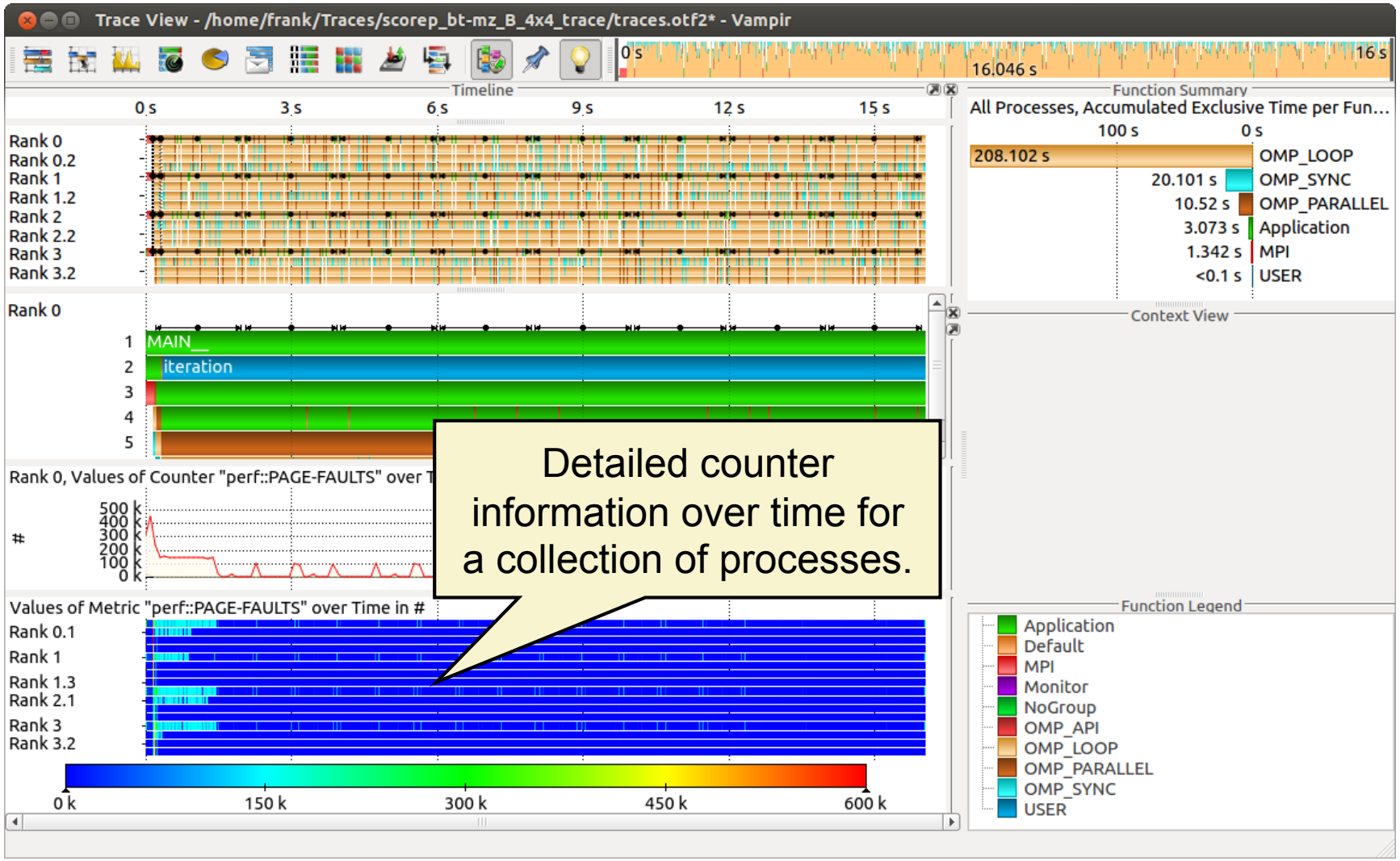
Counter Data Timeline



Detailed counter information over time for an individual process.

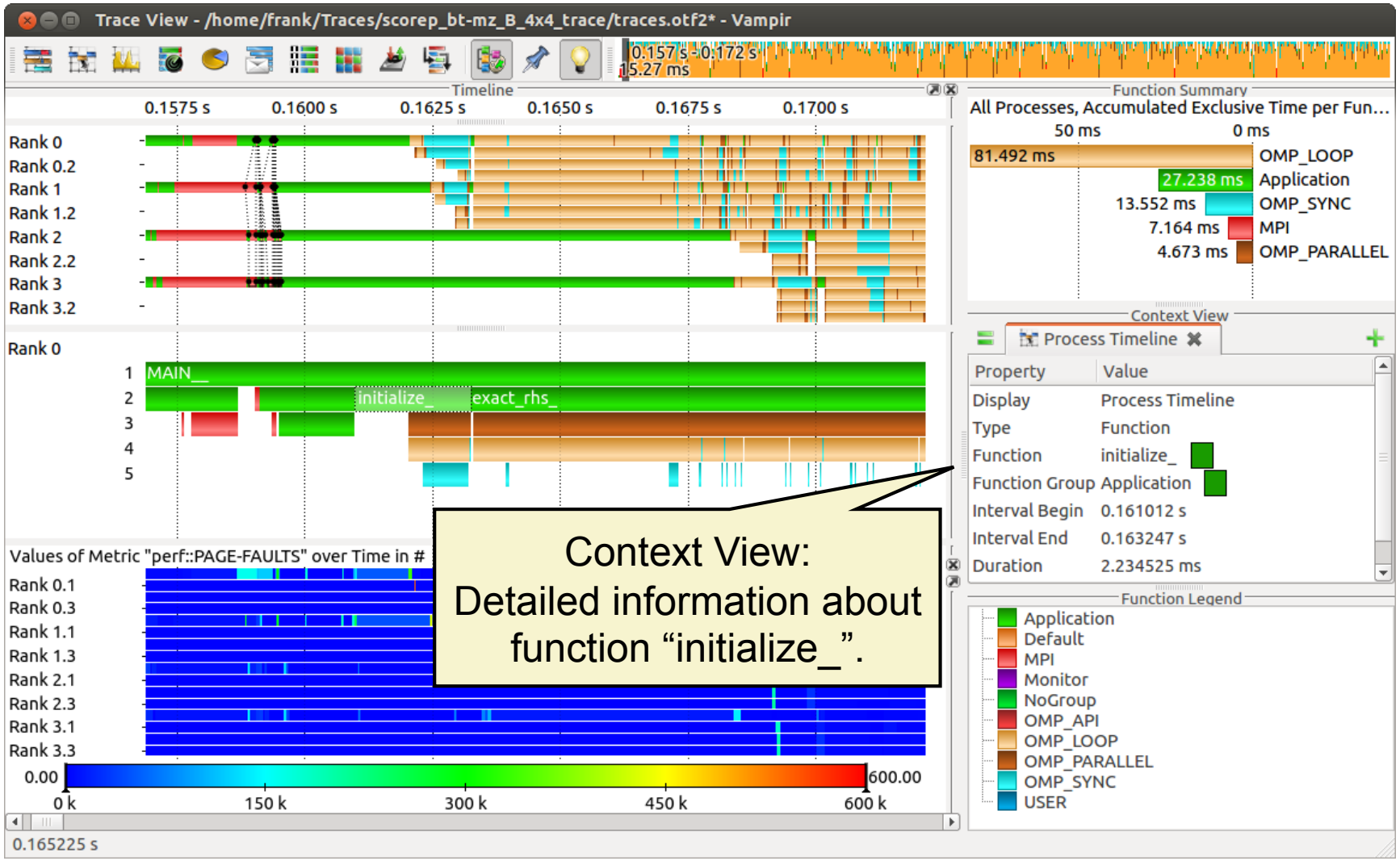
Vampir: example visualization

Performance Radar



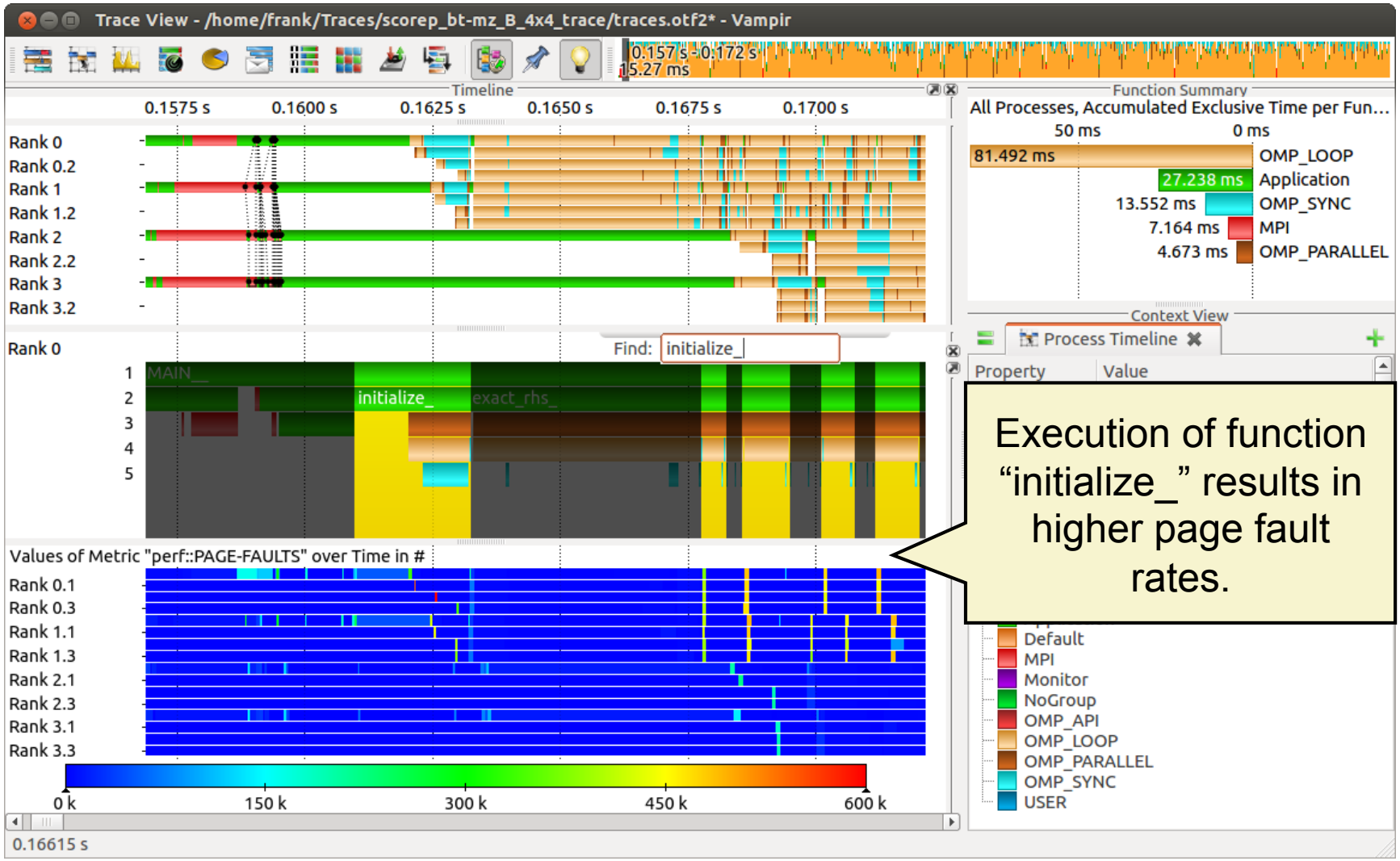
Vampir: example visualization

Zoom in: Inititialisation Phase



Vampir: example visualization

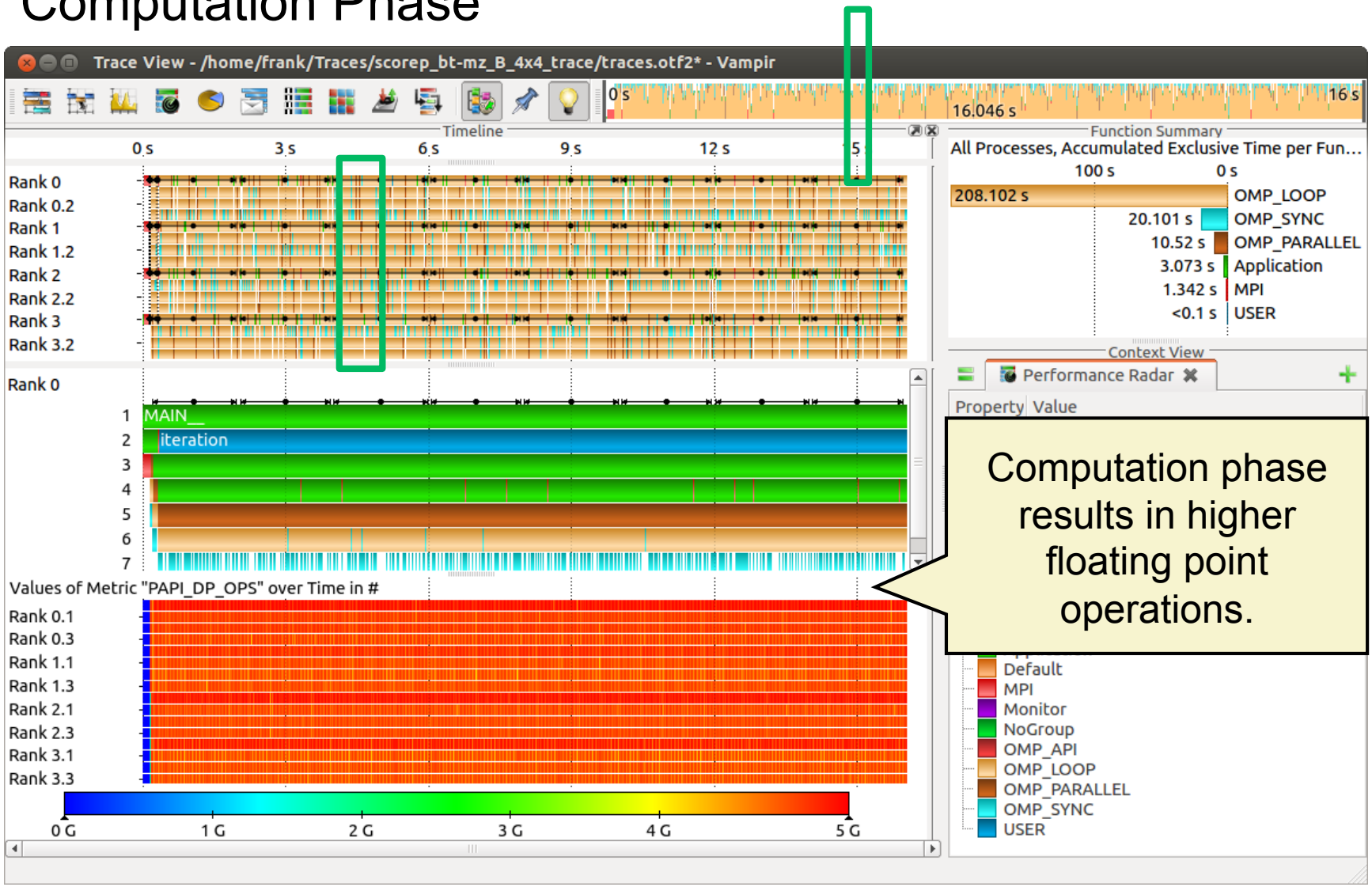
Feature: Find Function



Execution of function “initialize_” results in higher page fault rates.

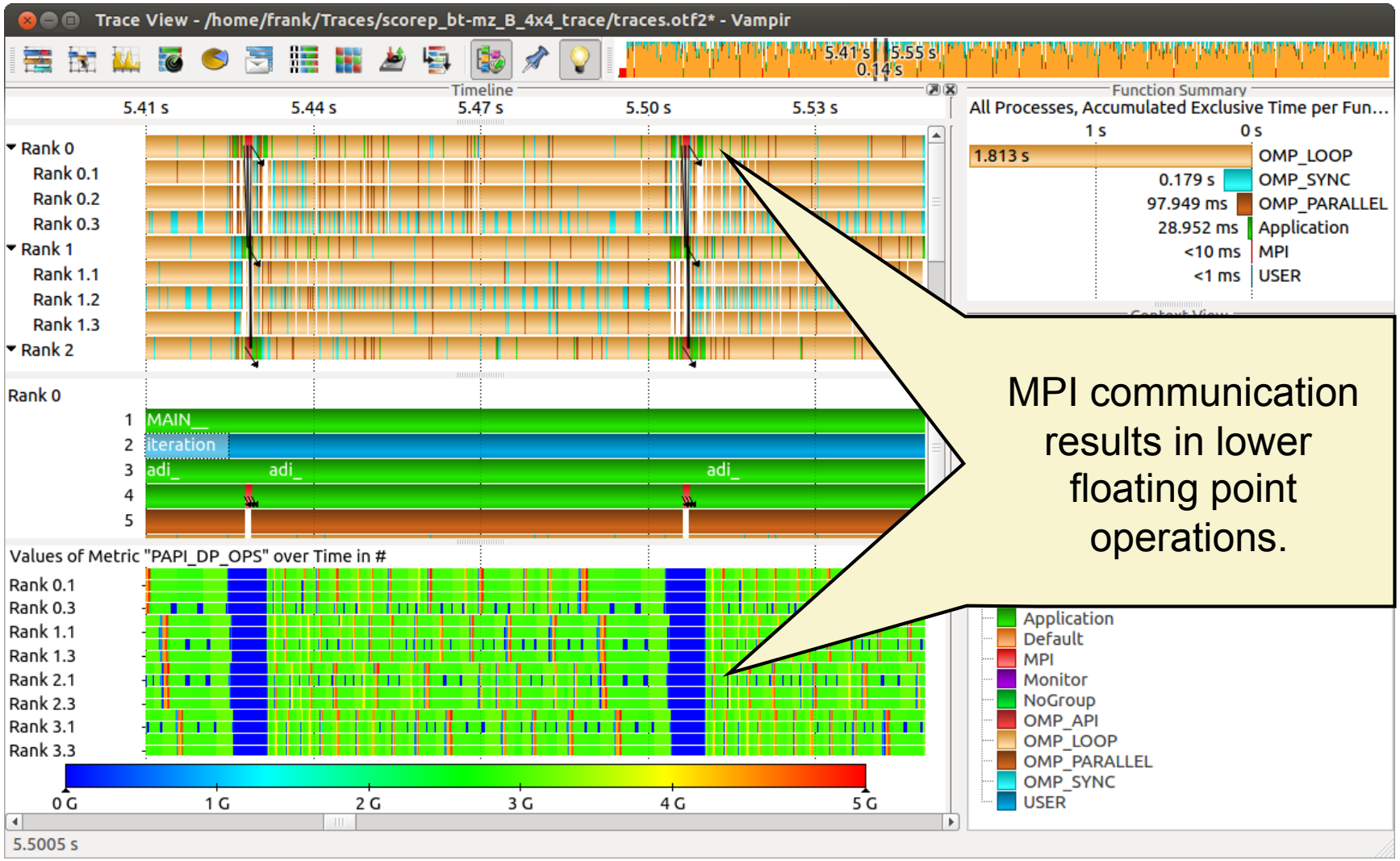
Vampir: example visualization

Computation Phase



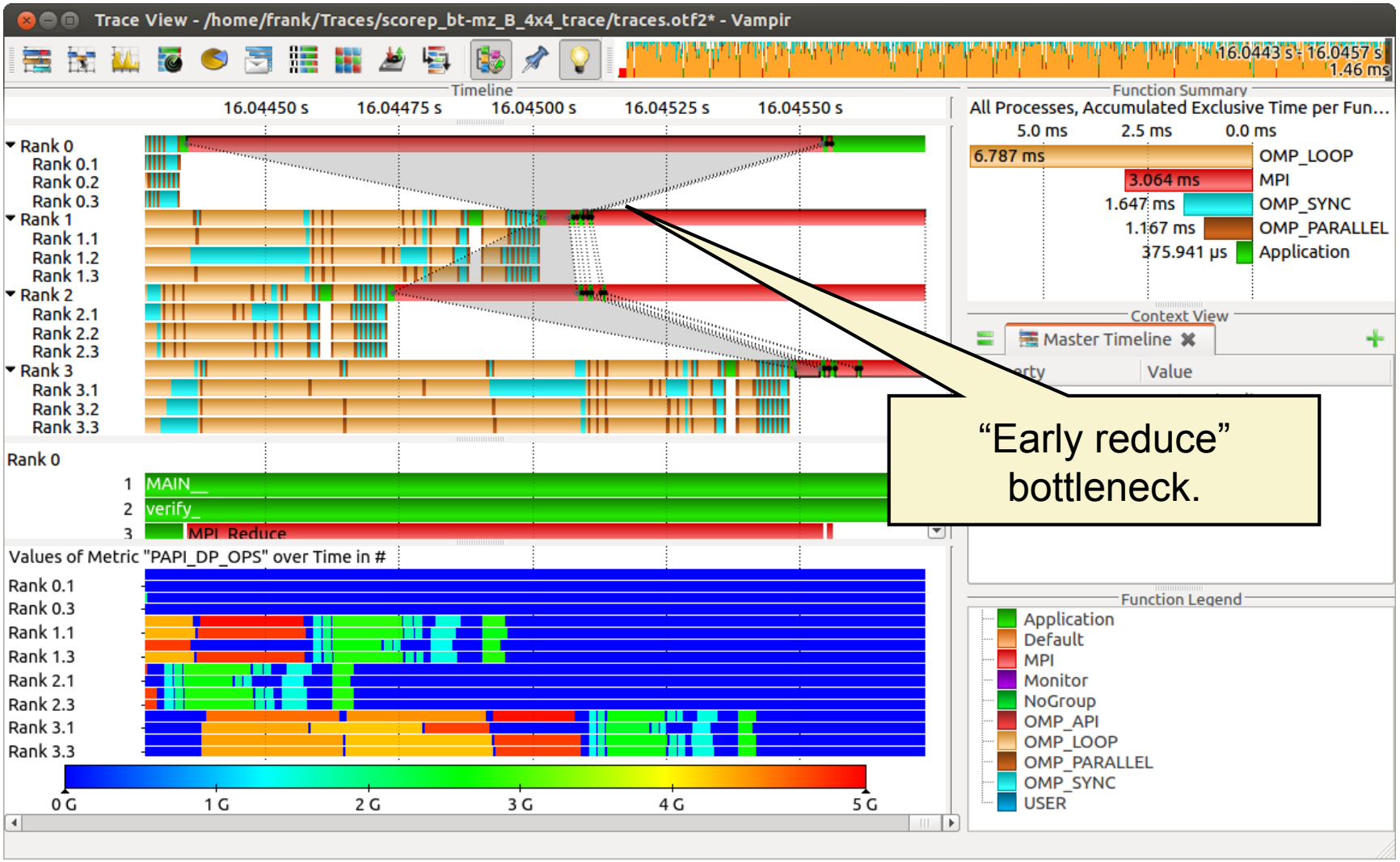
Vampir: example visualization

Zoom in: Computation Phase



Vampir: example visualization

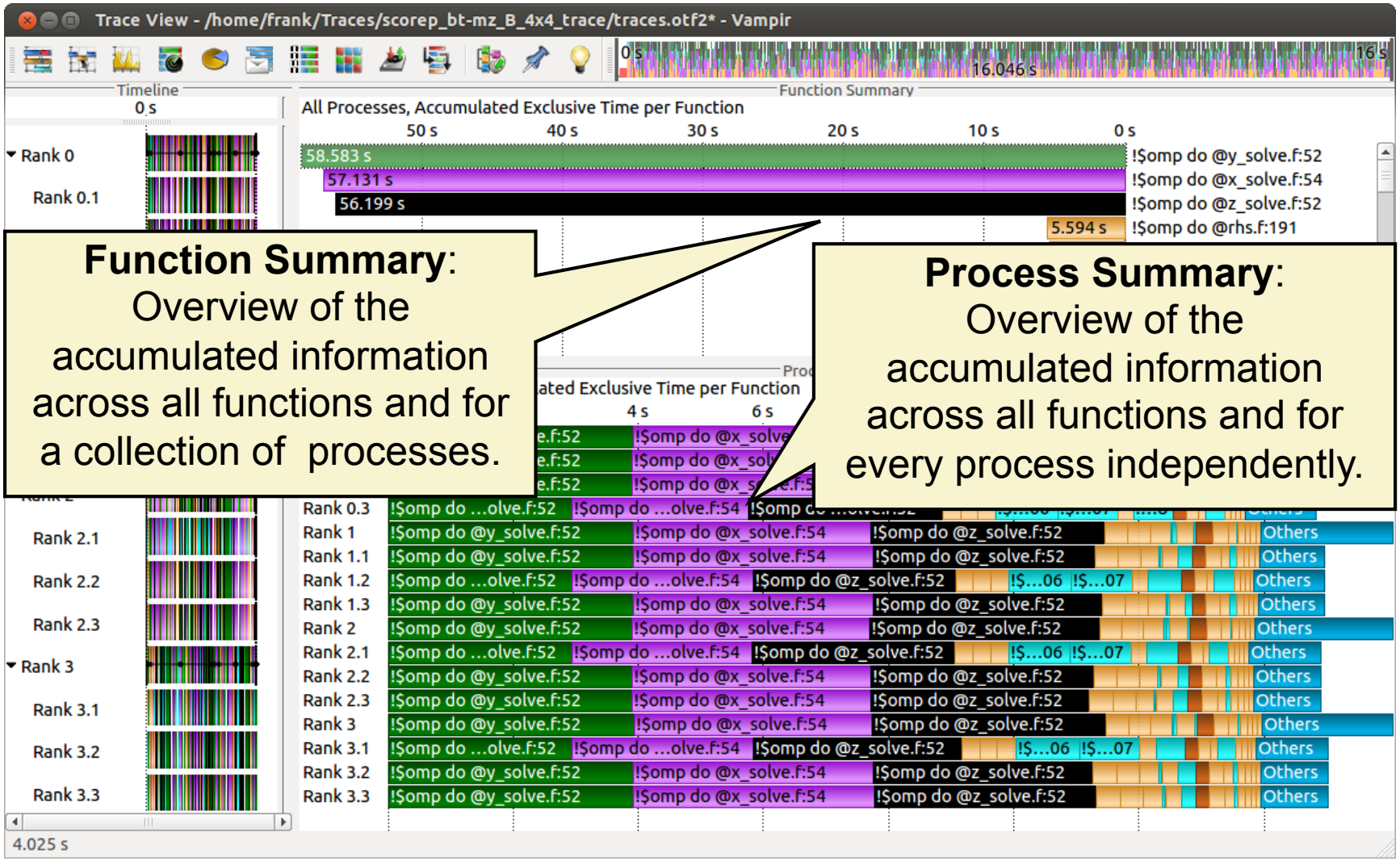
Zoom in: Finalisation Phase



"Early reduce" bottleneck.

Vampir: example visualization

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.

Vampir: example visualization

Process Summary

Trace View - /home/frank/Traces/scorep_bt-mz_B_4x4_trace/traces.otf2* - Vampir

Timeline

Function Summary

All Processes, Accumulated Exclusive Time per Function

58.583 s !$omp do @y_solve.f:52
57.131 s !$omp do @x_solve.f:54
56.199 s !$omp do @z_solve.f:52
5.594 s !$omp do @rhs.f:191
5.391 s !$omp do @rhs.f:80
4.83 s !$omp do @rhs.f:62
4.323 s !$omp implici...y_solve.f:406
4.287 s !$omp implici...x_solve.f:407

Process Summary

Individual Processes, Accumulated Exclusive Time per Function

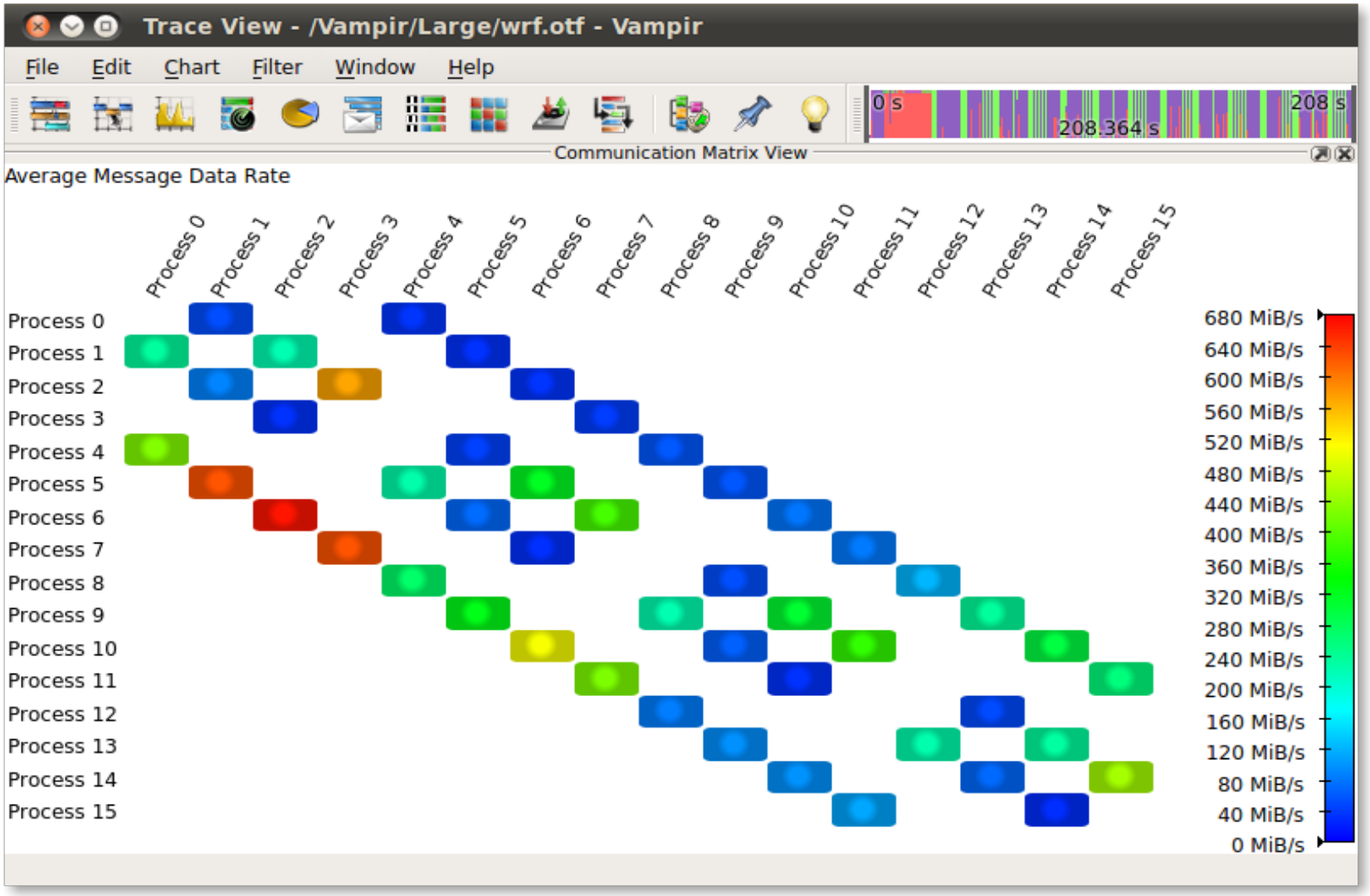
Process Summary

Similar Processes, Accumulated Exclusive Time per Function

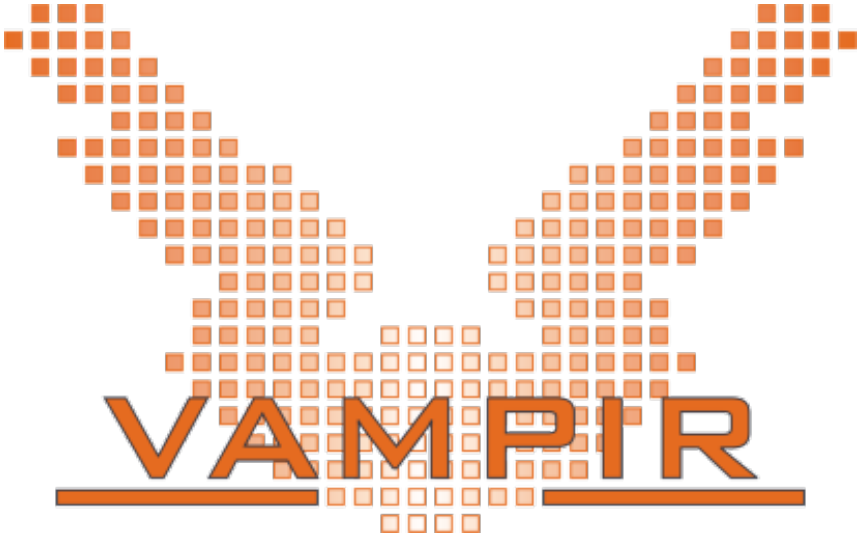
Find groups of similar processes and threads by using summarized function information.

Vampir: example visualization

Communication matrix



Vampir - Demo



Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
    - Score-P
    - Scalasca
    - CUBE
    - Vampir
    - TAU
  - Use cases
- Summary

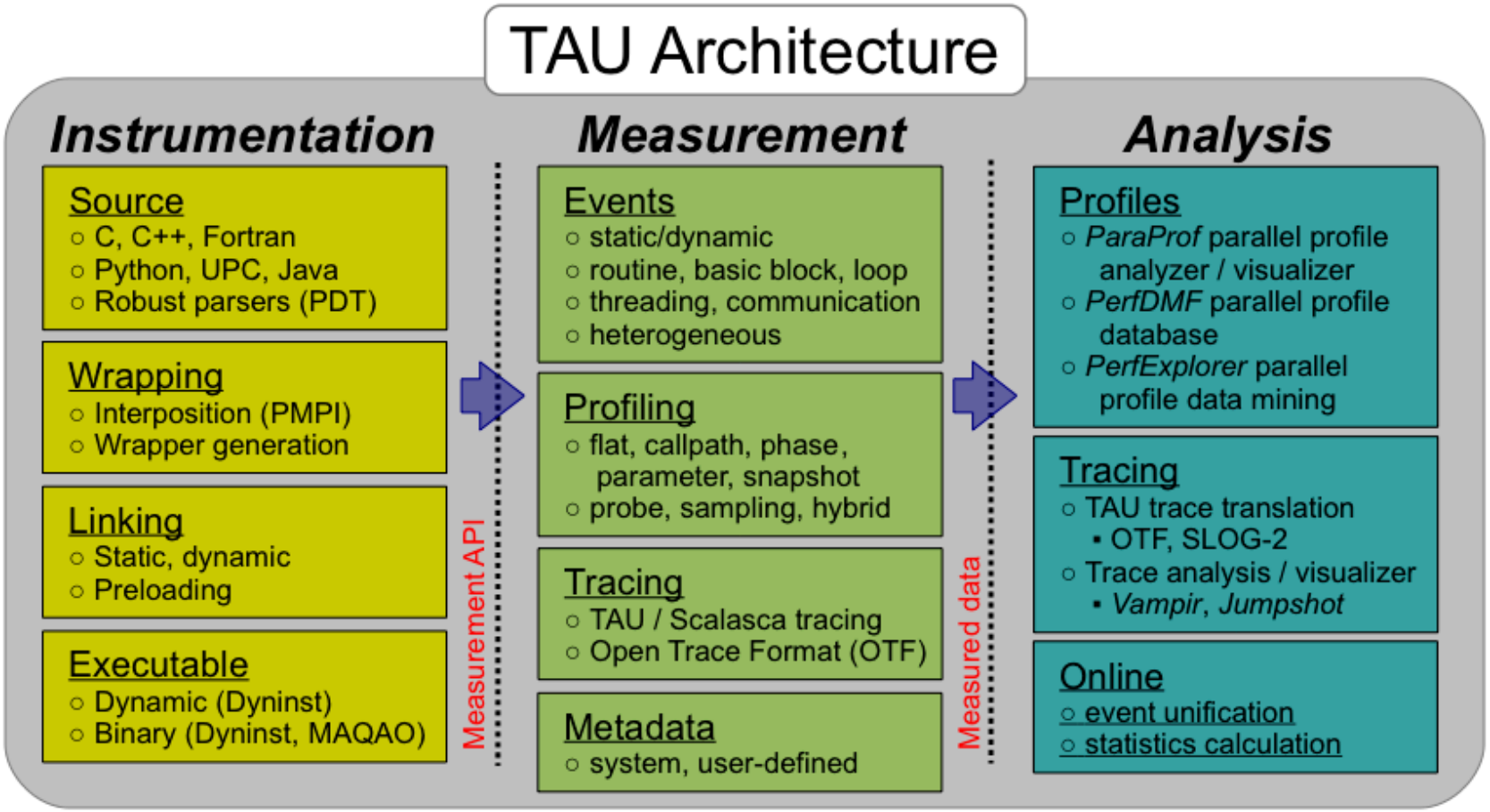
TAU is available at http://tau.uoregon.edu,

Free download, open source, BSD license

TAU Performance System®

Parallel performance framework and toolkit

- Supports all HPC platforms, compilers, runtime system
- Provides portable instrumentation, measurement, analysis



TAU Performance System®

Instrumentation

- Fortran, C++, C, UPC, Java, Python, Chapel
- Automatic instrumentation

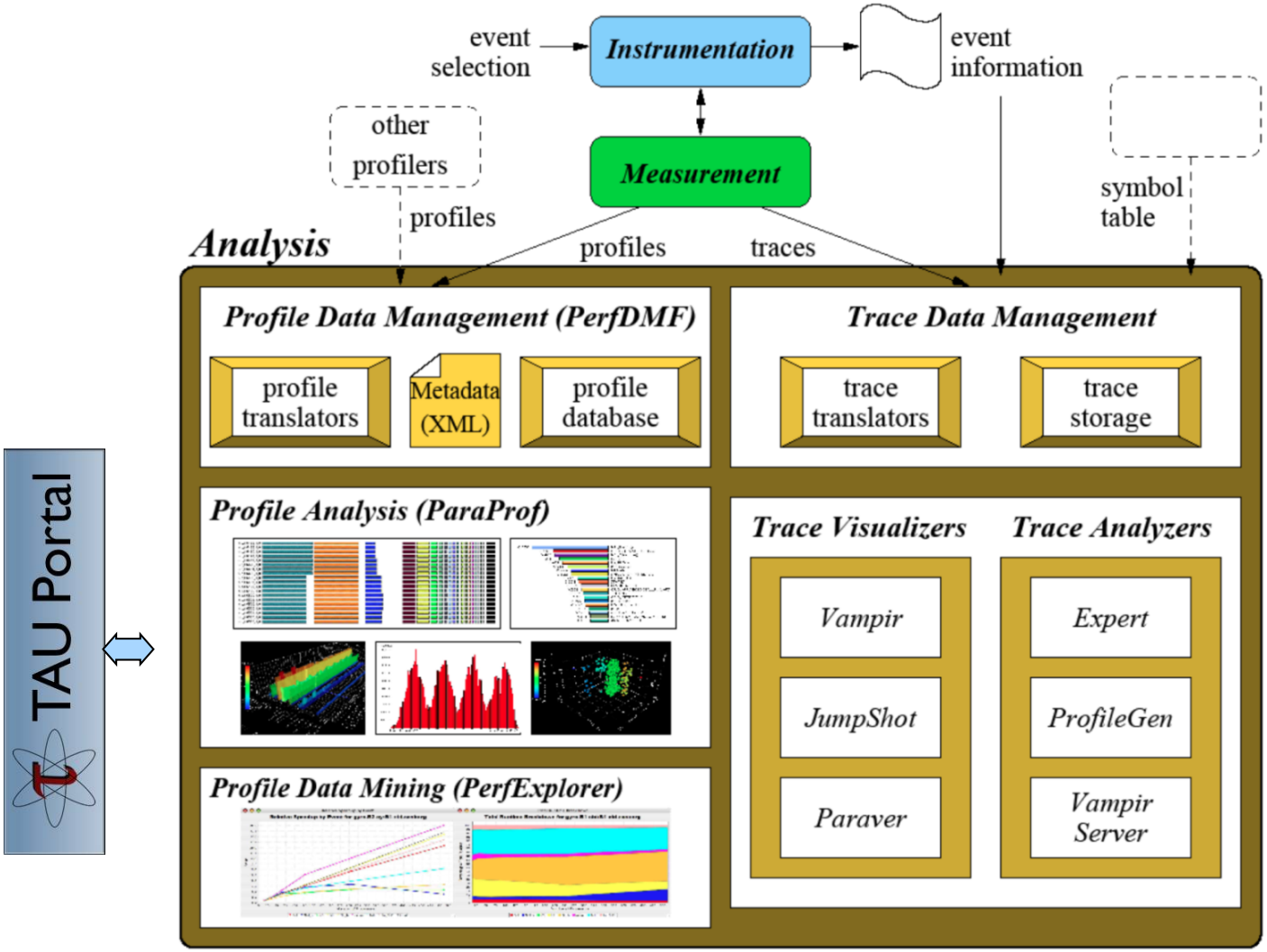
Measurement and analysis support

- MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
- pthreads, OpenMP, hybrid, other thread models
- GPU, CUDA, OpenCL, OpenACC
- Parallel profiling and tracing
- Use of Score-P for native OTF2 and CUBEX generation
- Efficient callpath profiles and trace generation using Score-P

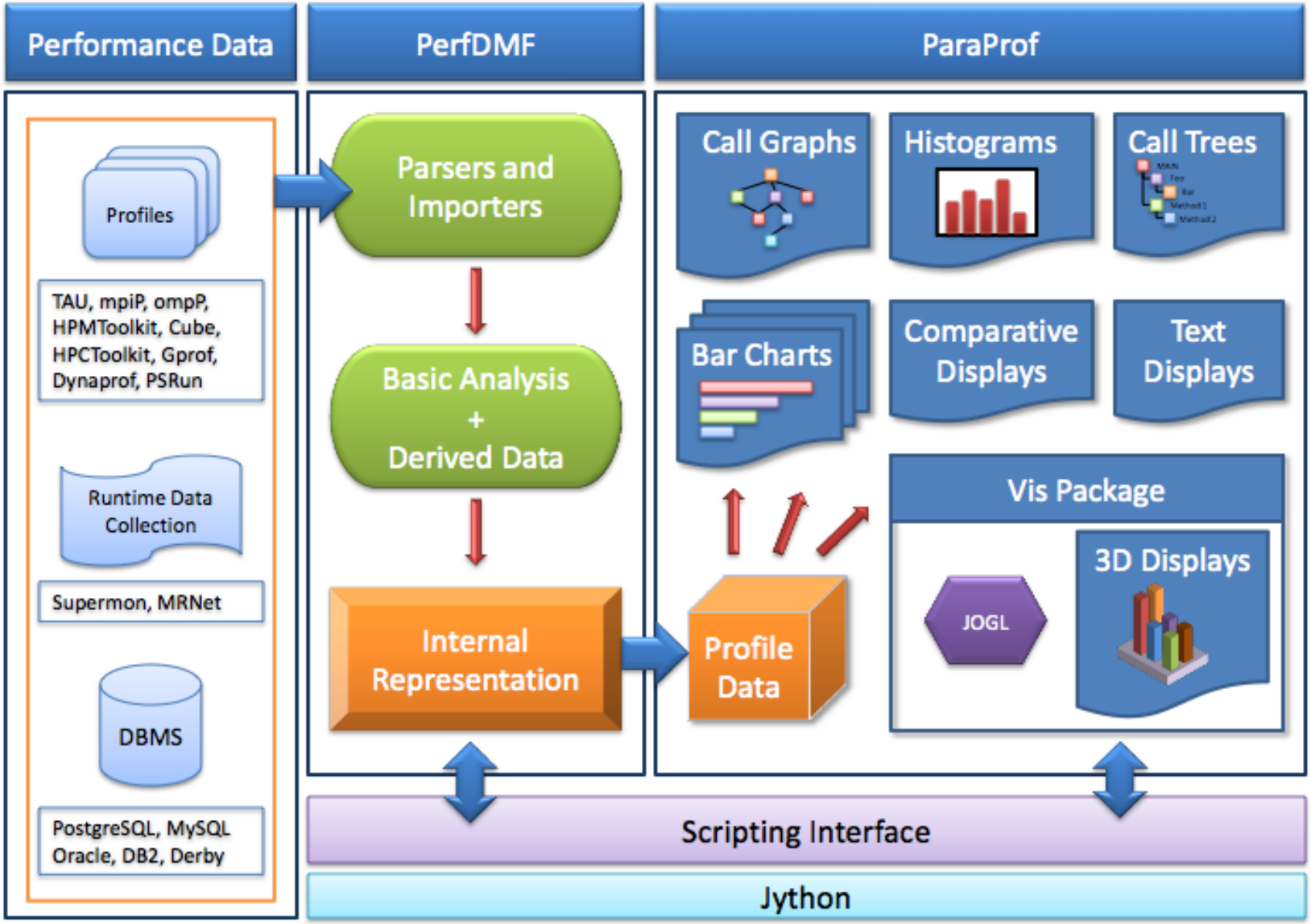
Analysis

- Parallel profile analysis (ParaProf), data mining (PerfExplorer)
- Performance database technology (PerfDMF, TAUdb)
- 3D profile browser

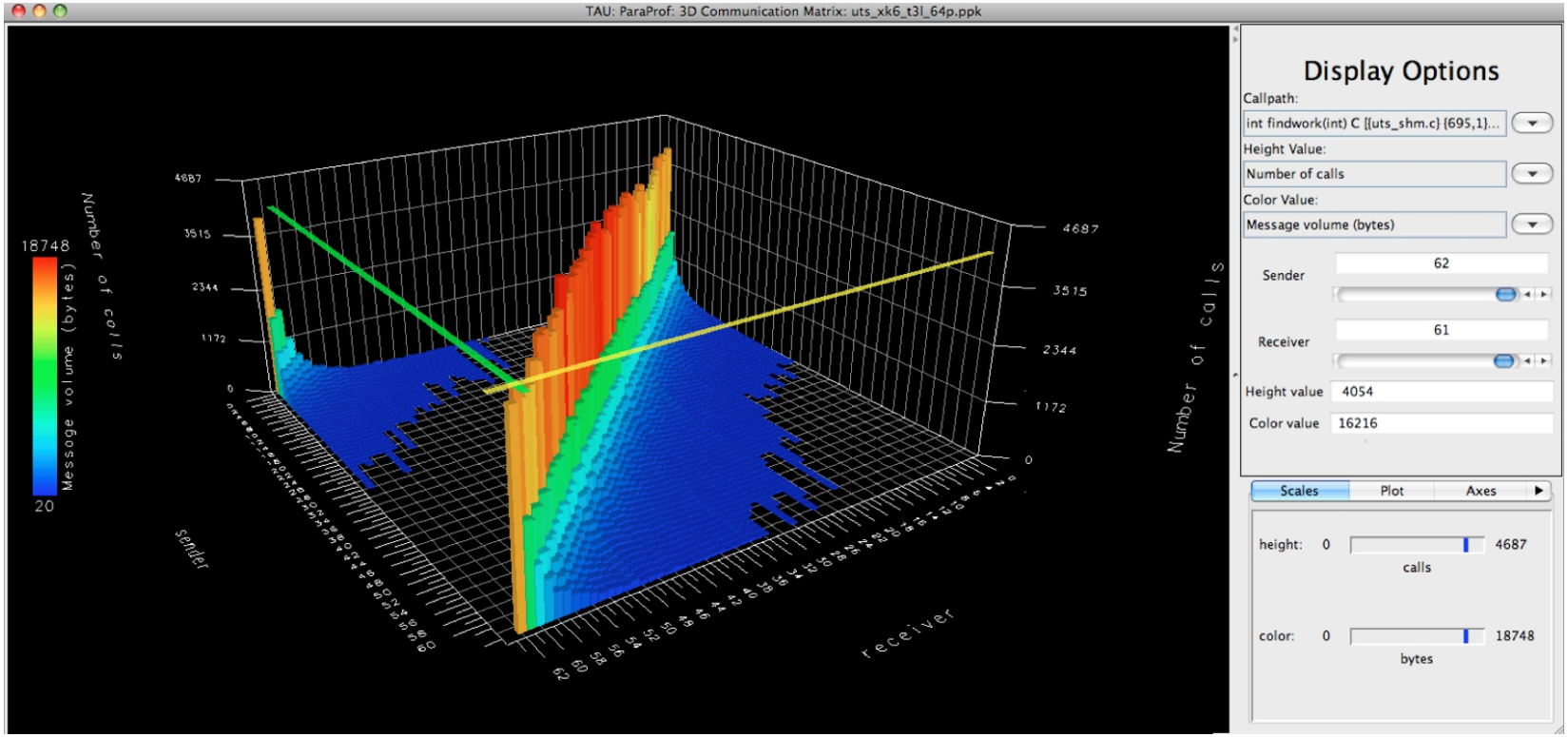
TAU Analysis



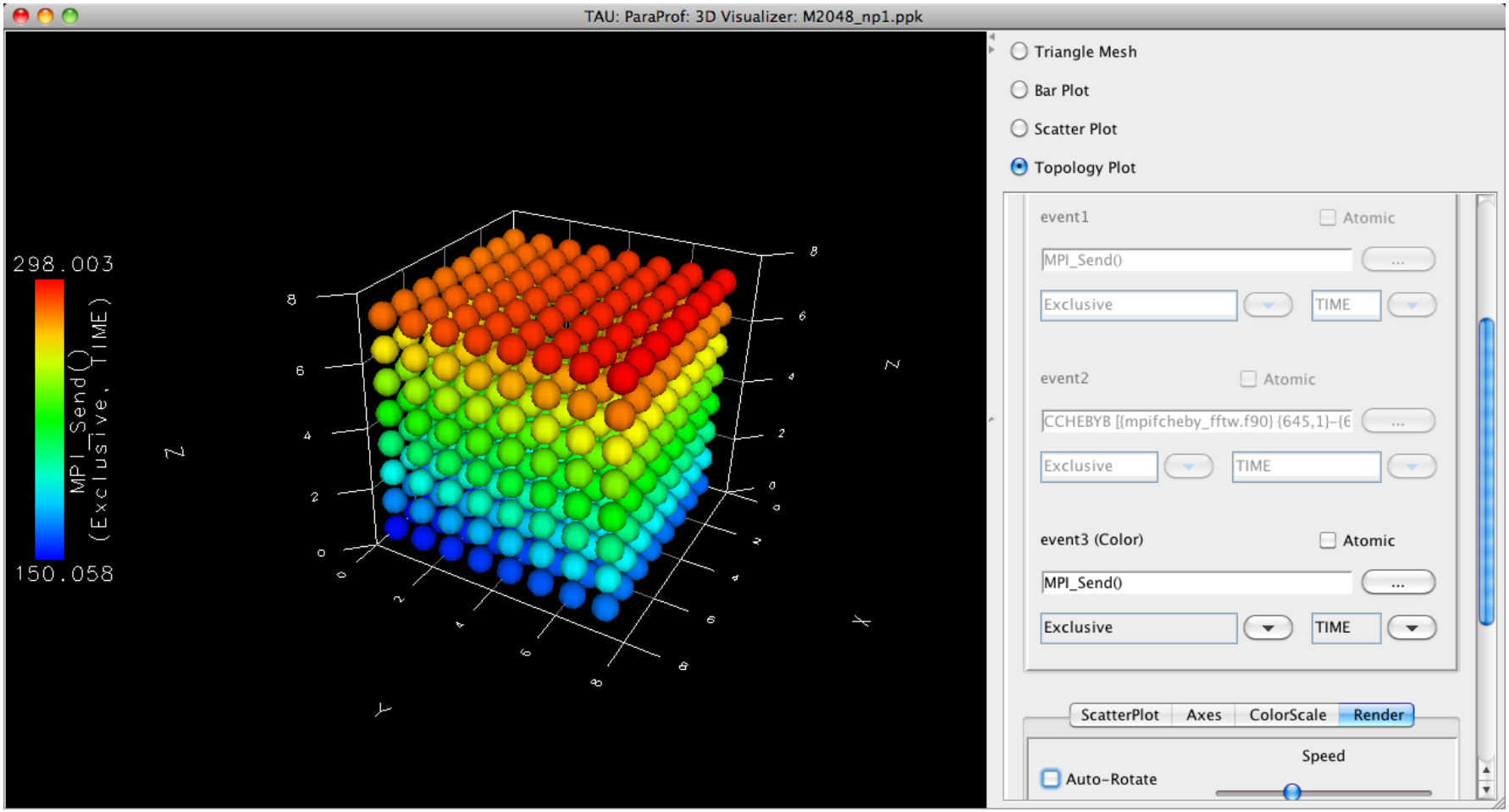
ParaProf Profile Analysis Framework



ParaProf: 3D Communication Matrix



ParaProf: Topology View 3D Torus (IBM BG/P)



Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
  - Use cases
    - Load imbalances (OpenMP)
    - GemsFDTD case study
    - COSMO case study
- Summary

Sparse matrix vector multiplication

$$\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Matrix has significant more zero elements => sparse matrix

Only non-zero elements of a_{ij} are saved efficiently in memory

Algorithm:

```
foreach row r in A
  y[r.x] = 0
  foreach non-zero element e in row
    y[r.x] += e.value * x[e.y]
```

Sparse matrix vector multiplication

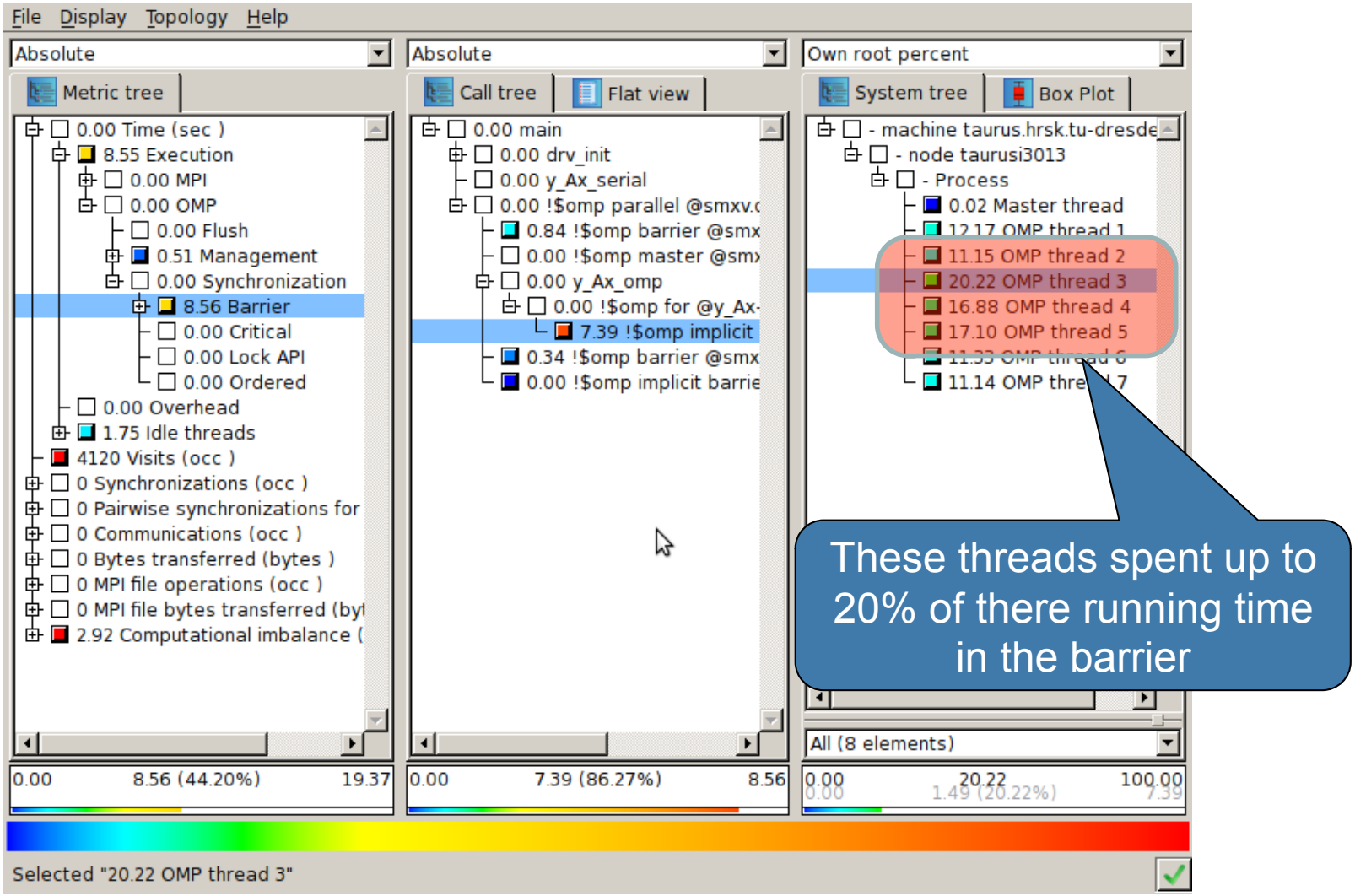
Naïve OpenMP Algorithm:

```
#pragma omp parallel for
foreach row r in A
  y[r.x] = 0
  foreach non-zero element e in row
    y[r.x] += e.value * x[e.y]
```

Distributes the rows of A evenly across the threads in the parallel region

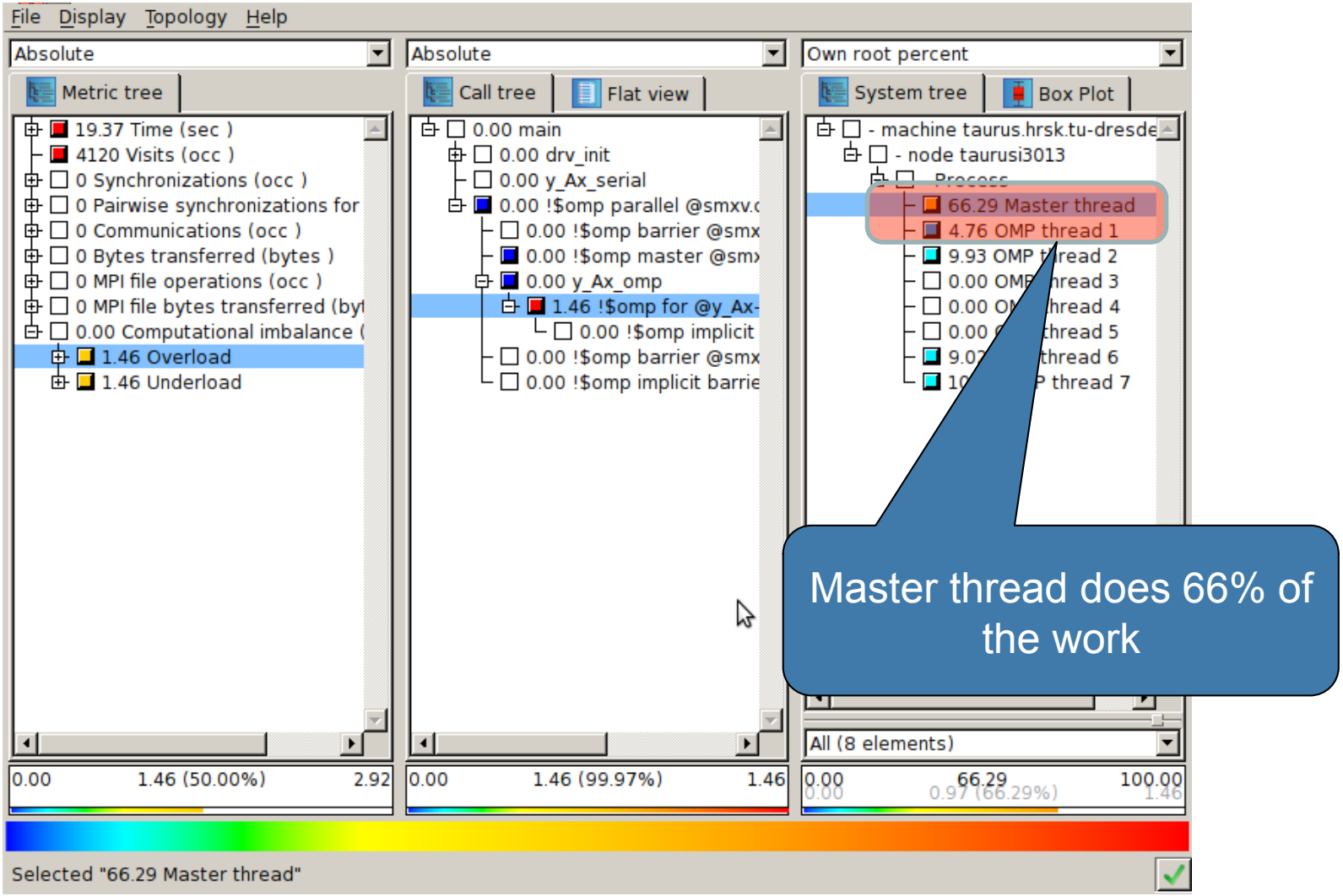
The distribution of the non-zero elements may influence the load balance in the parallel application

Time spent in OpenMP barriers



These threads spent up to 20% of there running time in the barrier

Computational imbalance



Master thread does 66% of the work

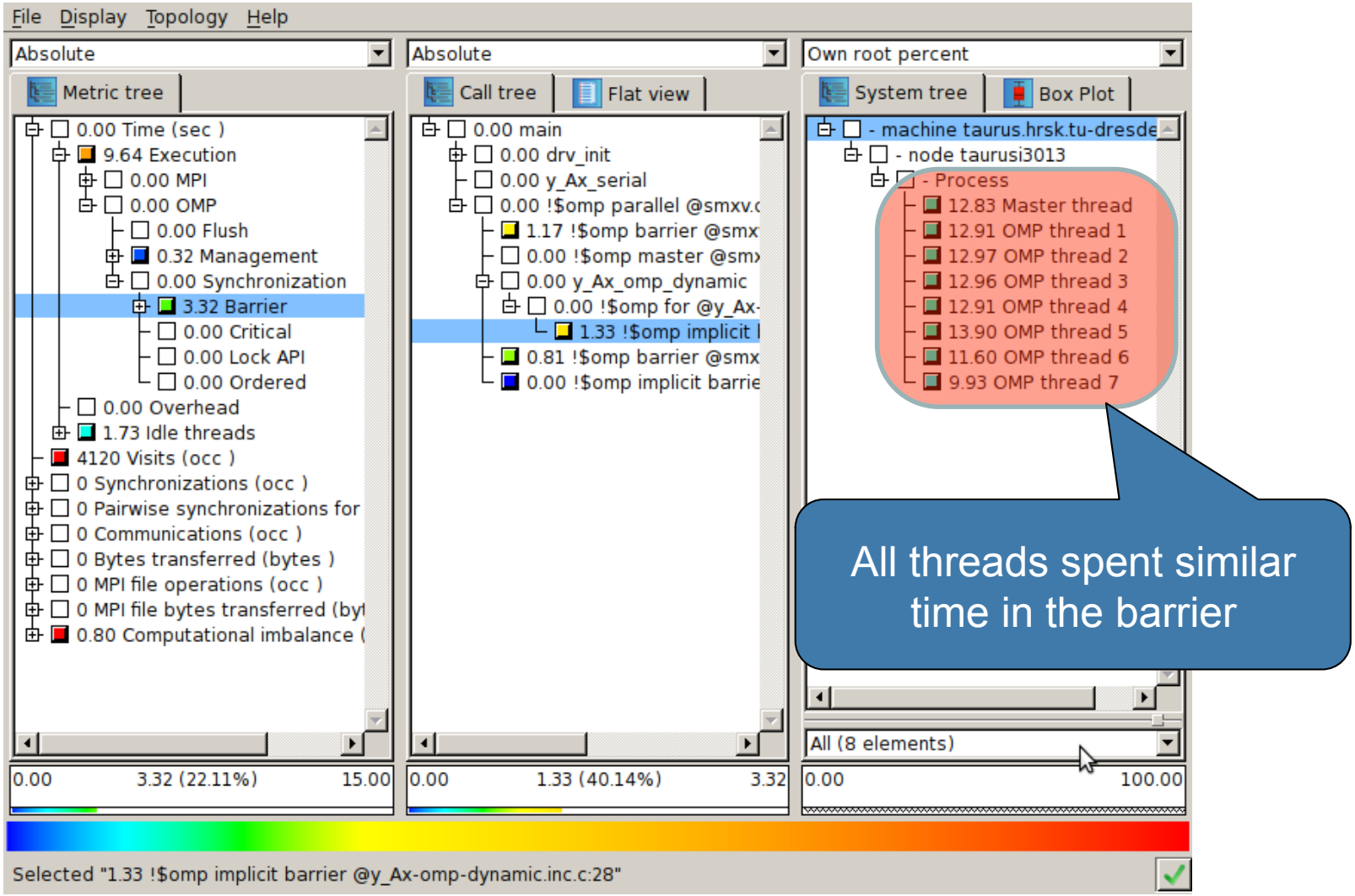
Sparse matrix vector multiplication

Improved OpenMP Algorithm

```
#pragma omp parallel for schedule(dynamic,1000)
foreach row r in A
  y[r.x] = 0
  foreach non-zero element e in row
    y[r.x] += e.value * x[e.y]
```

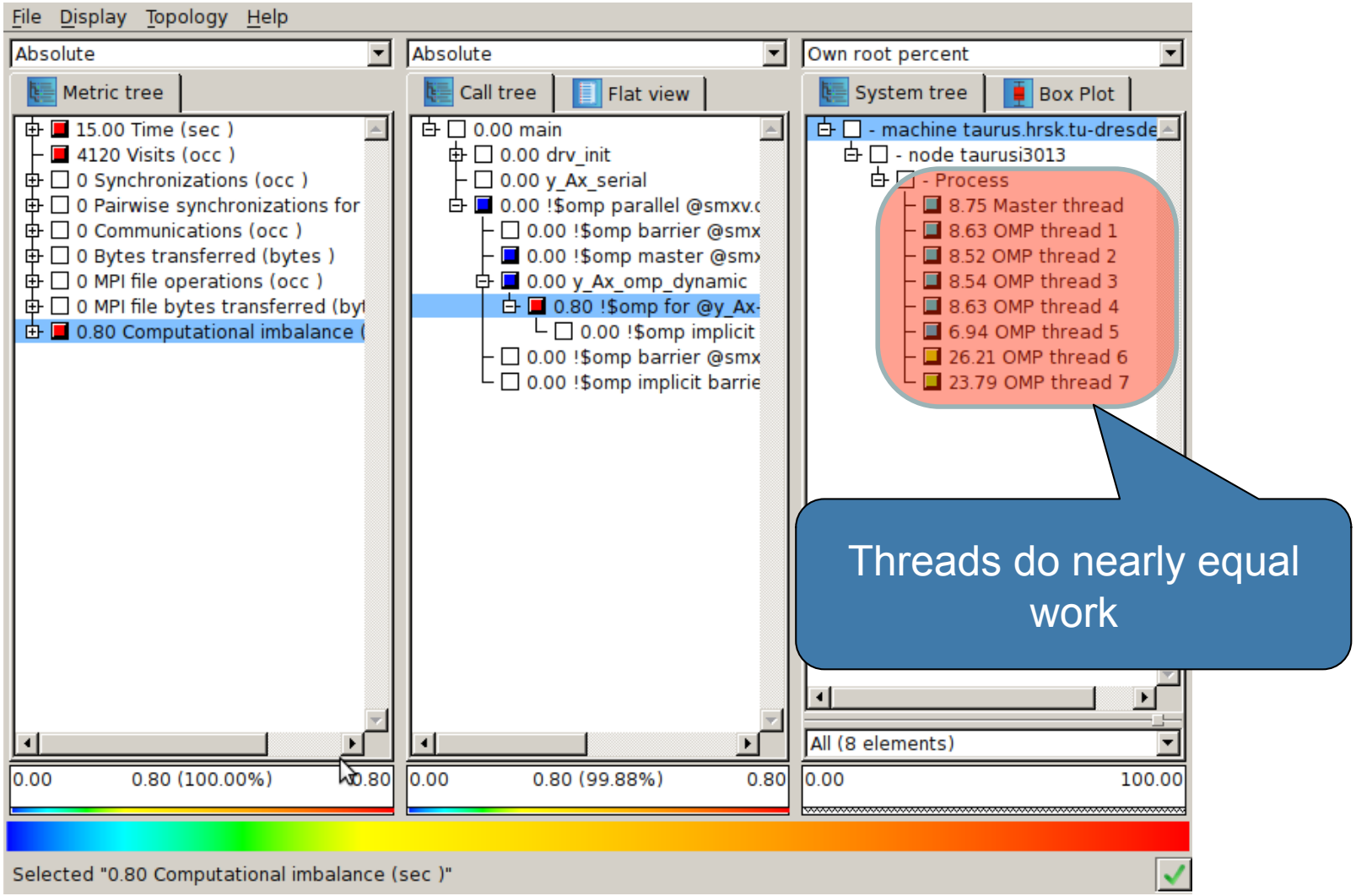
Distributes the rows of A *dynamically* across the threads in the parallel region

Time spent in OpenMP barriers



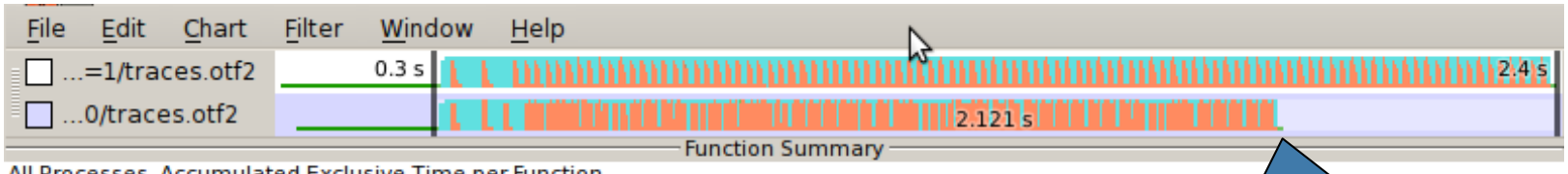
All threads spent similar time in the barrier

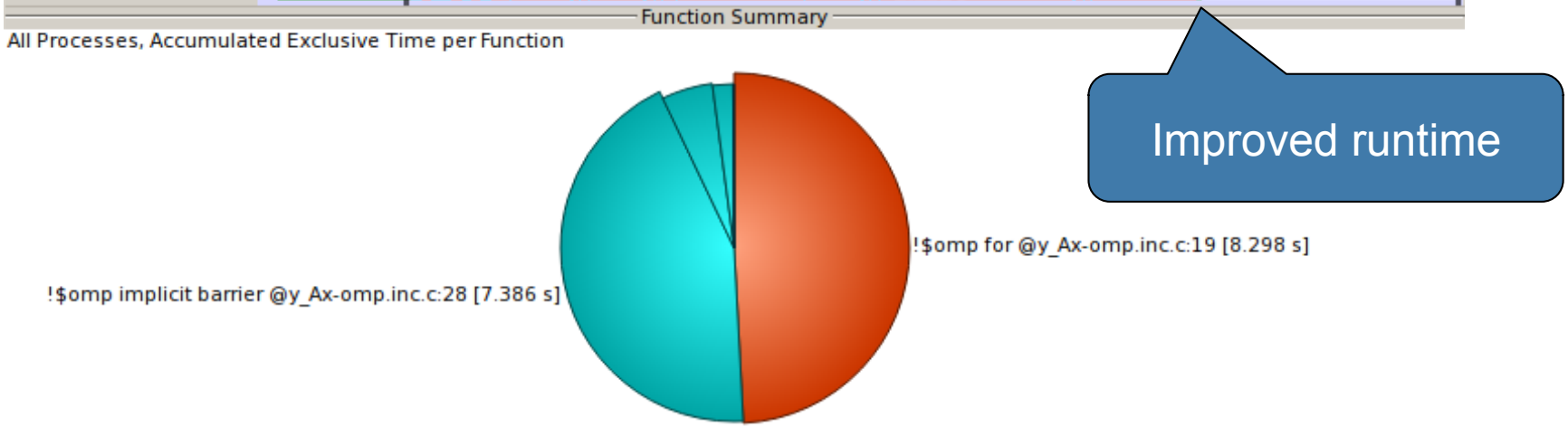
Computational imbalance



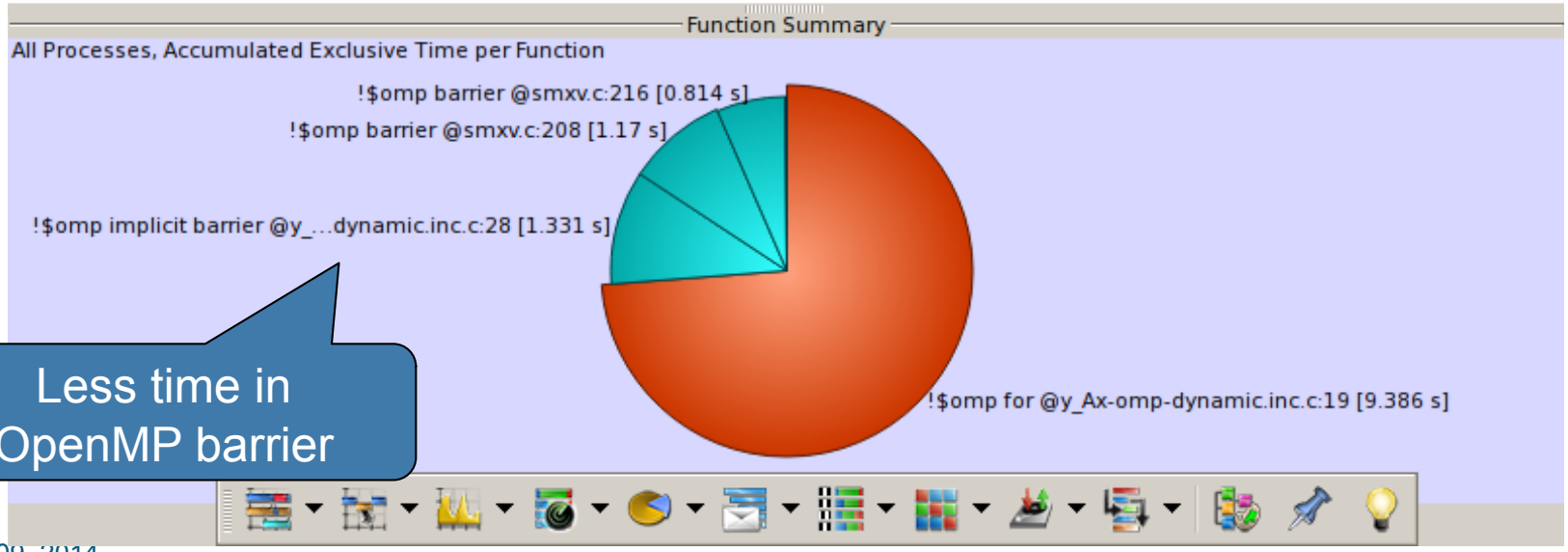
Threads do nearly equal work

Time spent in OpenMP barriers



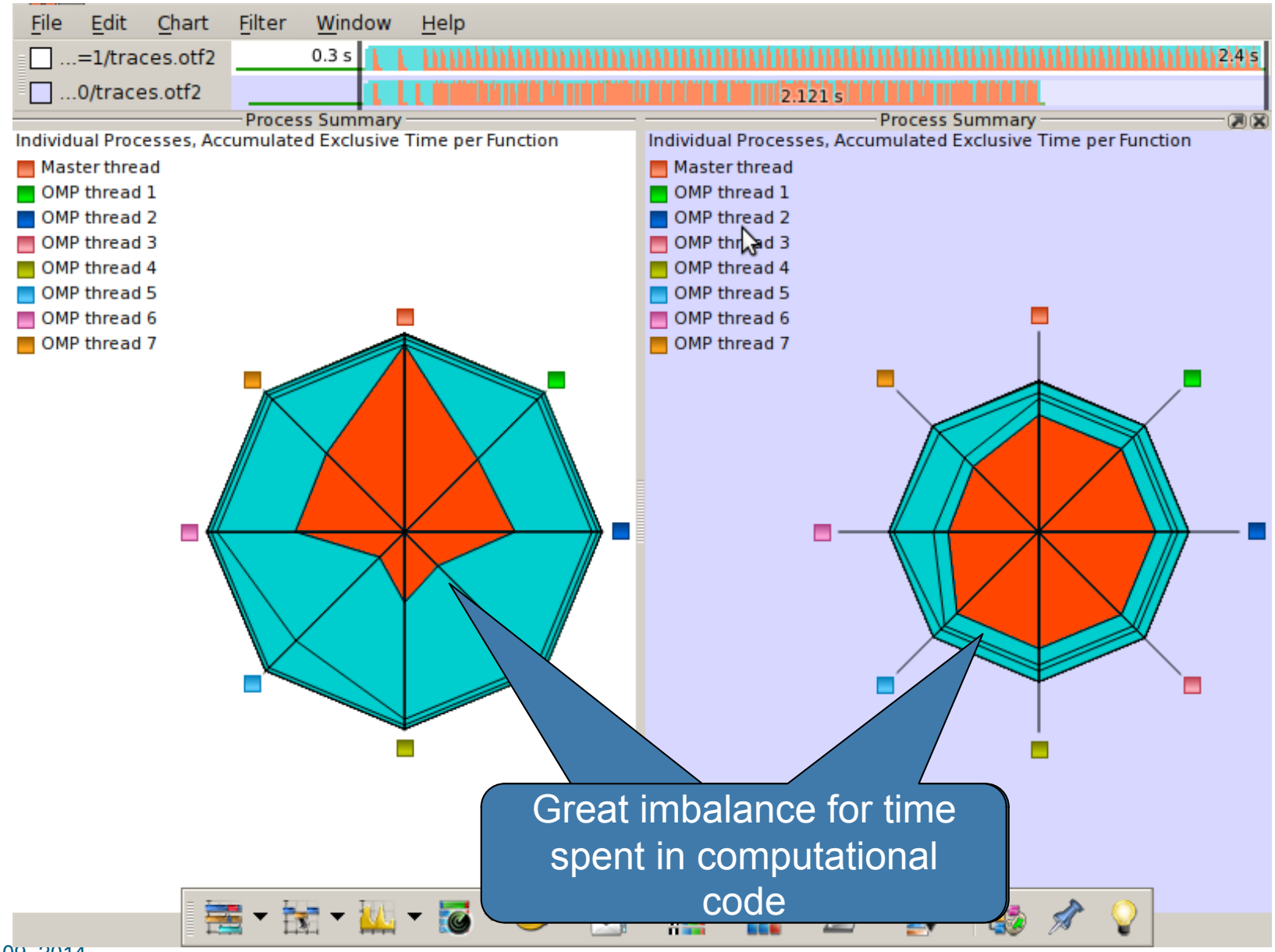


Improved runtime



Less time in OpenMP barrier

Computational imbalance



Great imbalance for time spent in computational code

Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
  - Use cases
    - Load imbalances (OpenMP)
    - GemsFDTD case study
    - COSMO case study
- Summary

GemsFDTD case study

Computational electromagnetics solver

- originates from KTH General ElectroMagnetics Solvers project
- finite-difference time-domain method for Maxwell equations

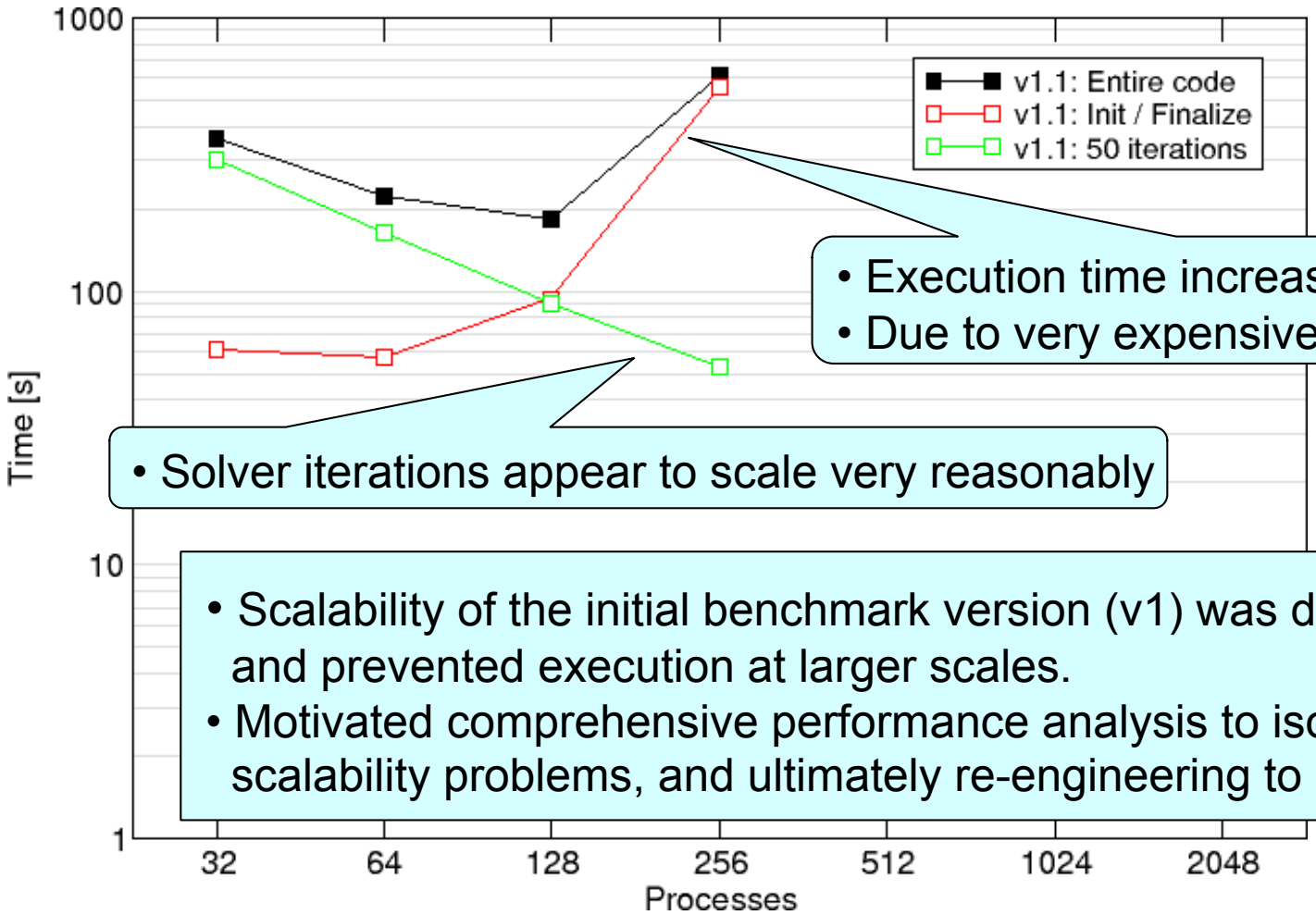
MPI parallel versions in SPEC MPI2007 benchmark suite

- original v1.1 (113.GemsFDTD) “medium” size
- revised v2.0 (145.lGemsFDTD) “large” size
- built with PGI 9.0.4 Fortran90 compiler (21k lines of code)
  - typical benchmark optimization: -fastsse -O3 -Mipa=fast,inline

Results for Cray XT4@EPCC (“HECToR”)

- using “ltrain” dataset from v2.0 benchmark (50 timesteps)
- default Scalasca instrumentation for measurements
  - 9 of 90 application user-level source routines specified in filter determined by scoring initial summary experiment

GemsFDTD v1 scalability on Cray XT4



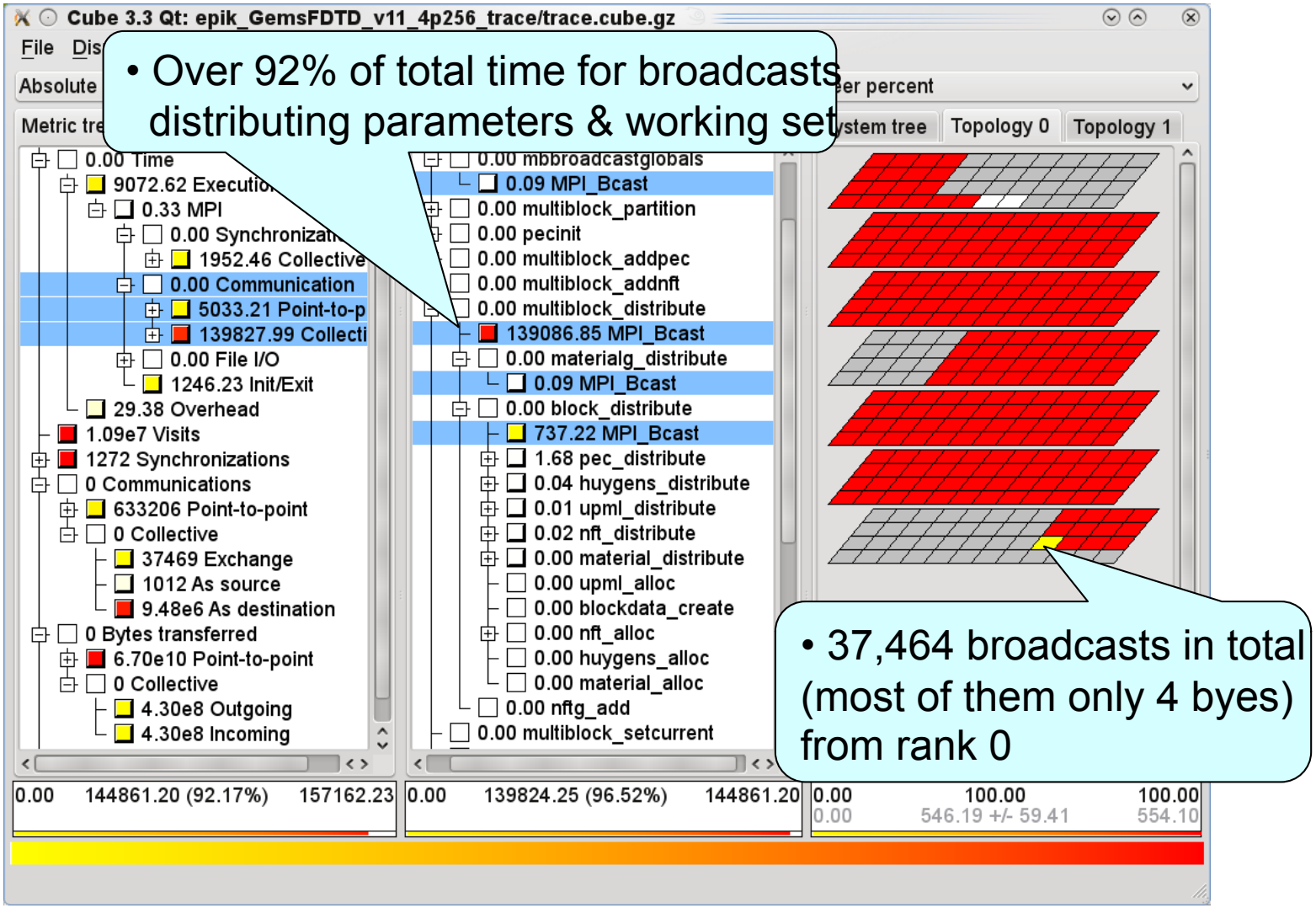
['ltrain' runs on CrayXT4 HECToR]

- Execution time increases exponentially
- Due to very expensive initialization

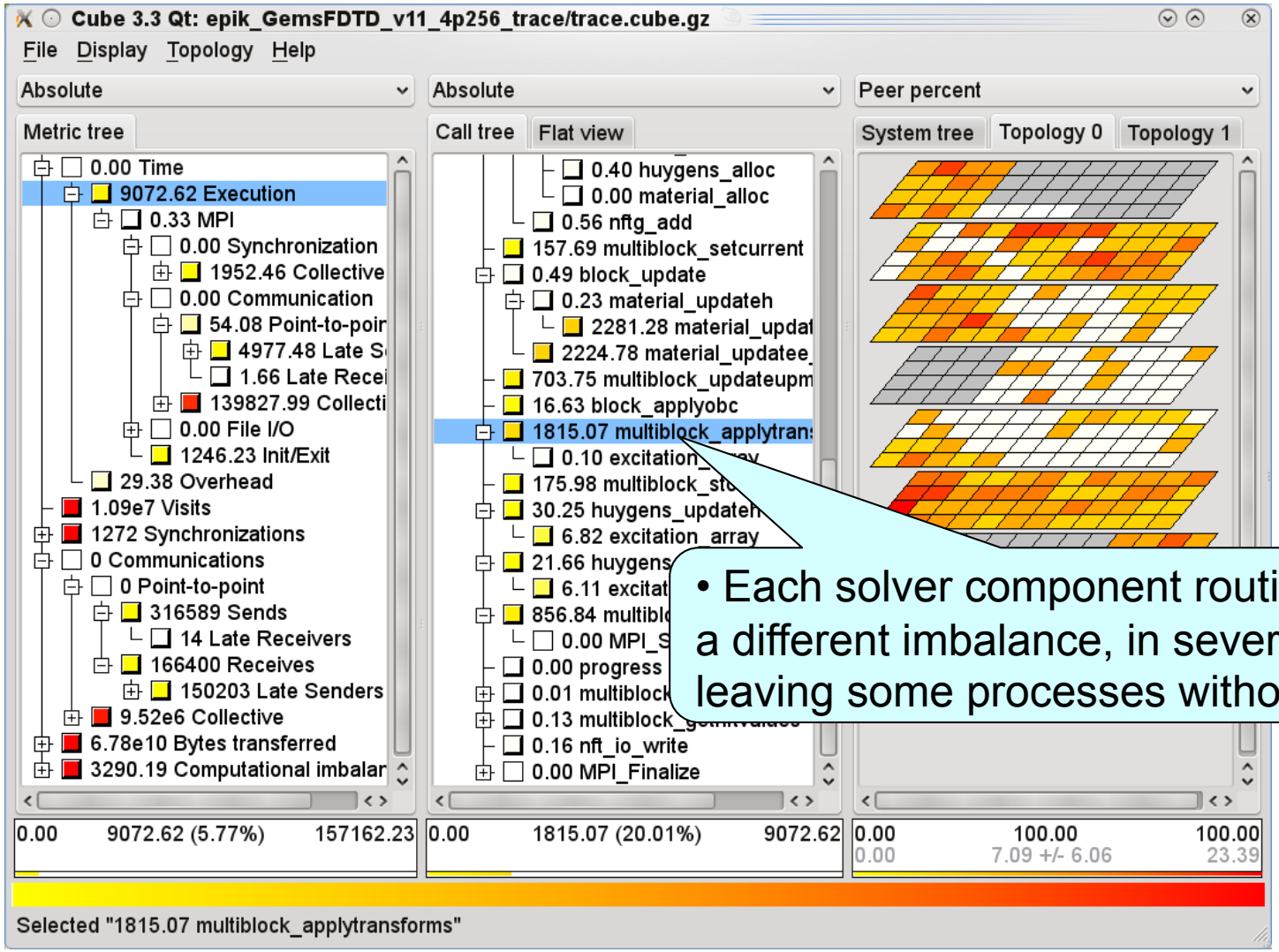
- Solver iterations appear to scale very reasonably

- Scalability of the initial benchmark version (v1) was disappointing and prevented execution at larger scales.
- Motivated comprehensive performance analysis to isolate scalability problems, and ultimately re-engineering to resolve them.

Time for initialization broadcasts (v1.1)



Computation time in solver transforms (v1.1)



- Each solver component routine has a different imbalance, in severe cases leaving some processes without work

GemsFDTD case study

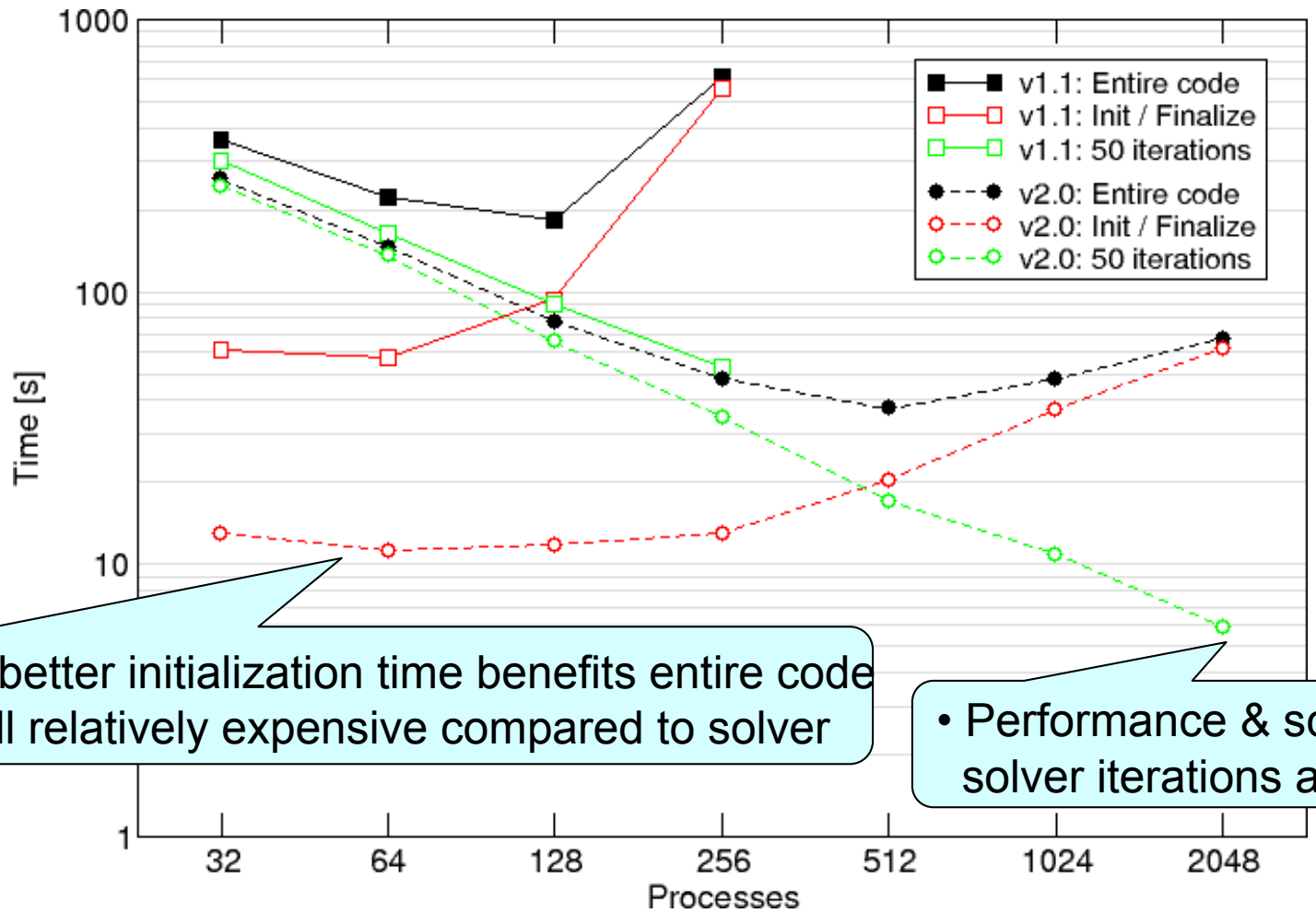
Analysis results

- Initialization dominated by numerous broadcasts
- Expensive serial multi-block partition by rank 0
- Computational imbalance and blocking communication in solver
  - Late sender

Reengineering of the code

- Aggregation of multiple data values into larger messages
- Postpones allocations until all block information in broadcast
- Using nonblocking communication to exchange blocks
- Omitting idle states of 2 processes

GemsFDTD v1 & v2 scalability on Cray XT4



['ltrain' runs on CrayXT4 HECToR]

- Much better initialization time benefits entire code
- but still relatively expensive compared to solver

- Performance & scalability of solver iterations also improved

Outline

- Introduction
- Code development
- Performance analysis and tuning
  - Concepts and basics
  - Selected performance issues
  - Selected performance analysis tools
  - Use cases
    - Load imbalances (OpenMP)
    - GemsFDTD case study
    - COSMO case study
- Summary

COSMO-7/XE6 case study

Regional climate and weather model

- Developed by Consortium for Small-scale Modeling (COSMO)
  - DWD, MeteoSwiss and others
- Non-hydrostatic limited-area atmospheric model (6.6km grid)

MPI parallel version 4.12 (Jan-2011)

- Built with PGI 10.9 Fortran90 compiler (222k lines of code)

MeteoSwiss operational 24-hour forecast of 06-Dec-2010

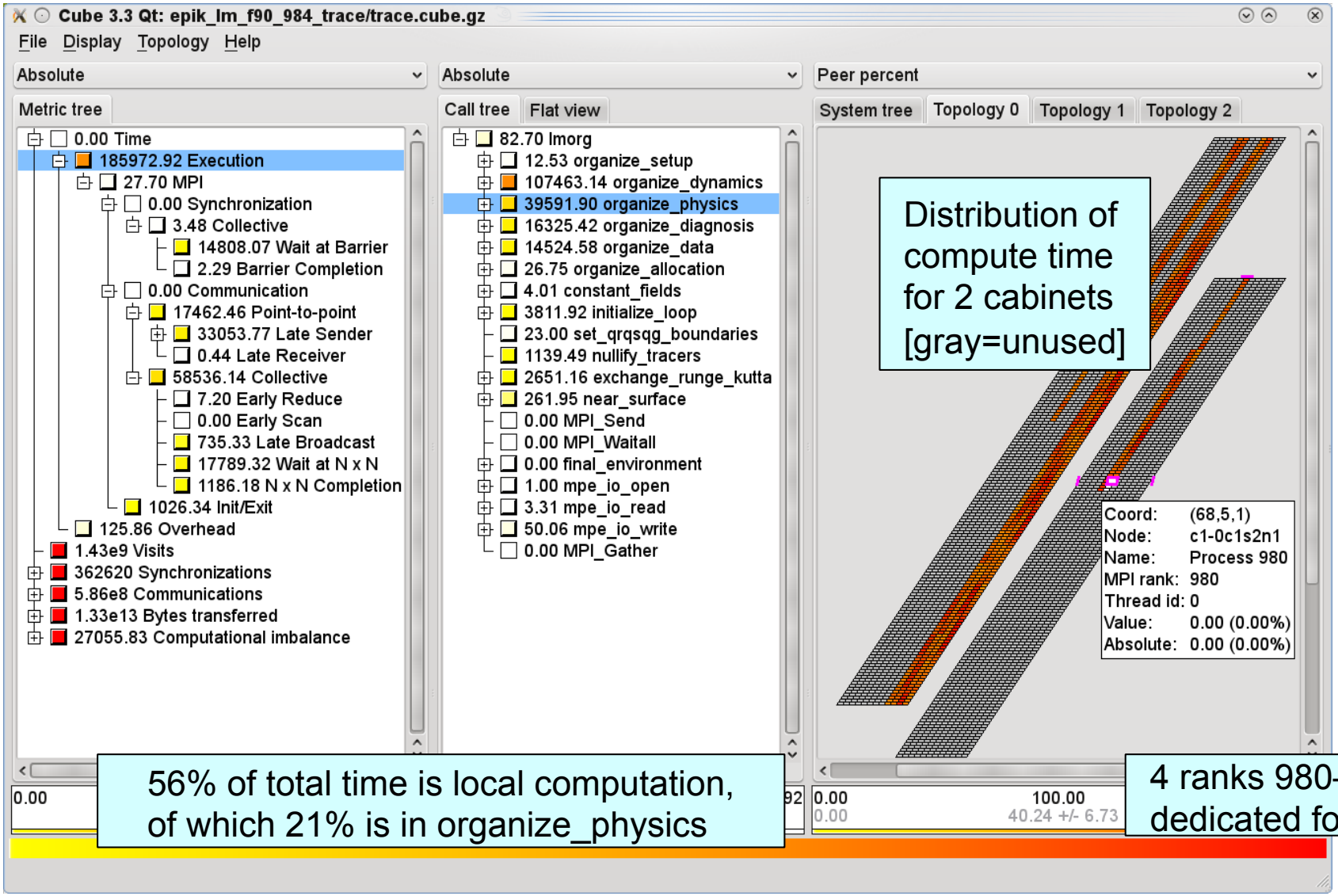
- Western Europe 393x338x60 resolution, 1440 timesteps

Run with 984 processes on 'palu' Cray XE6 at CSCS

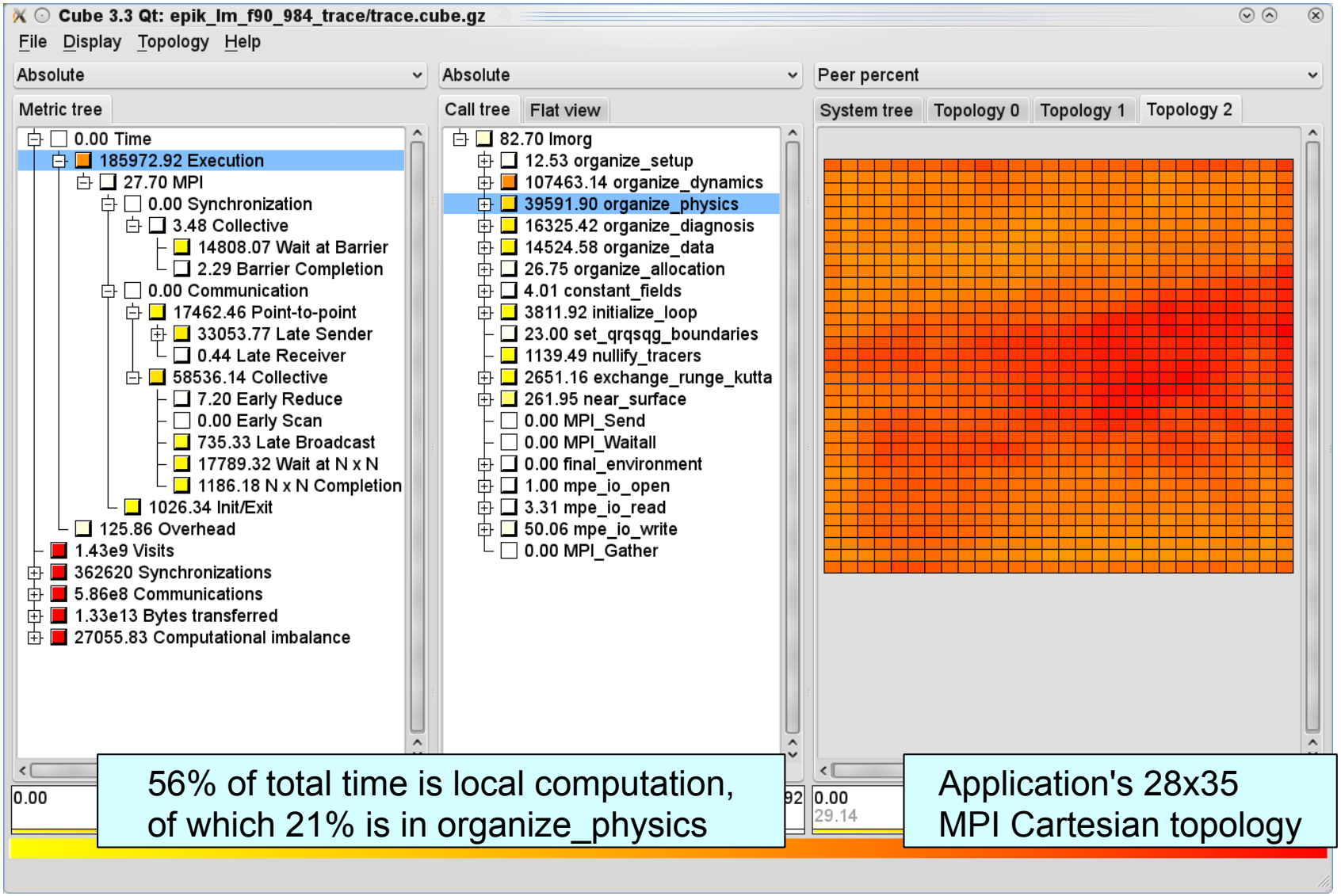
- 28x35 compute grid + 4 dedicated I/O processes
- Used 41 Opteron compute nodes each with 24 cores
- Scalasca trace measurement with 19 of 178 routines filtered
- 44GB trace written in 23s and analyzed in 82s

Courtesy of Oliver Fuhrer (MeteoSwiss) & CSCS

COSMO/XE6 physics computation time



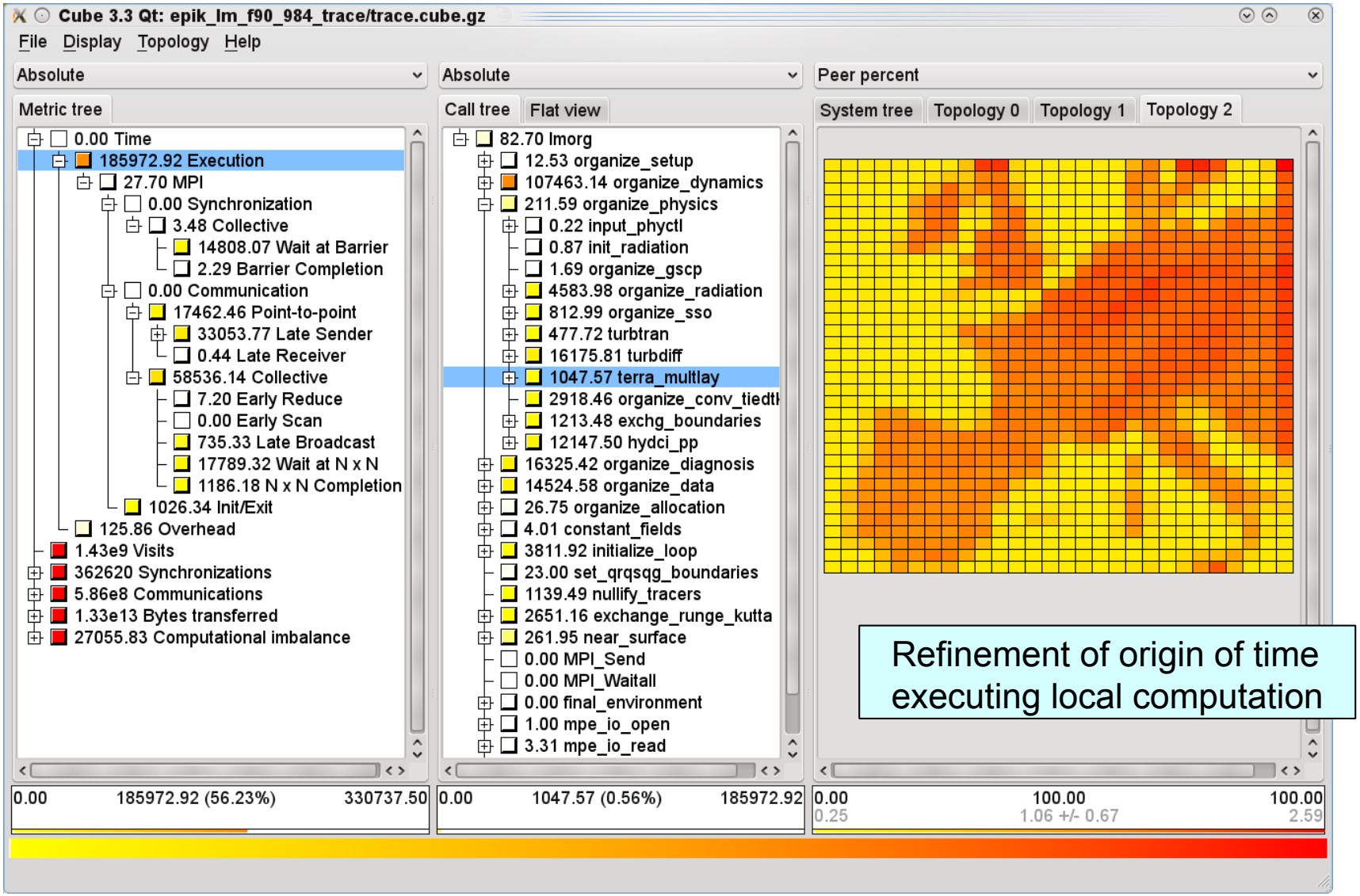
COSMO/XE6 physics computation time



56% of total time is local computation, of which 21% is in organize_physics

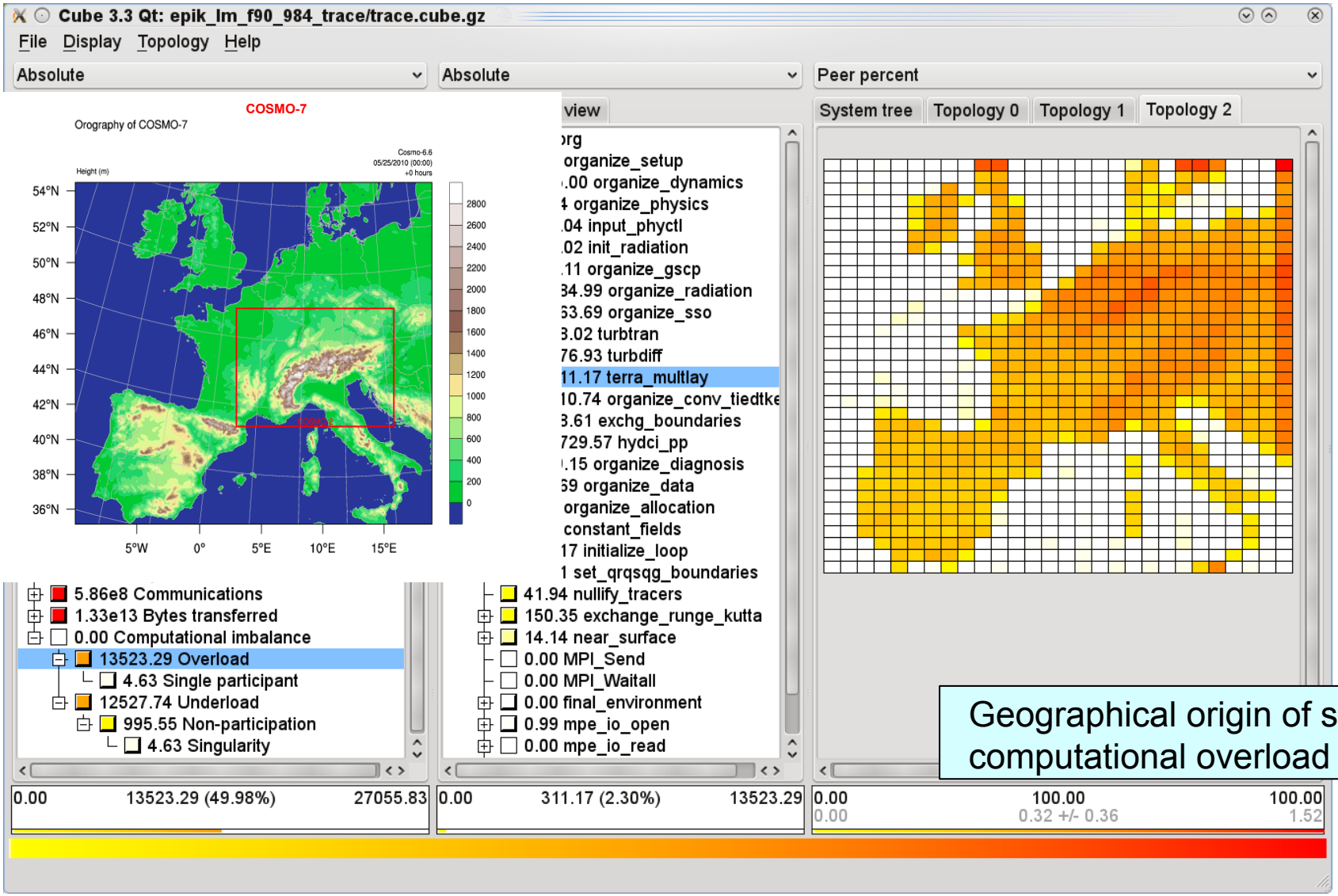
Application's 28x35 MPI Cartesian topology

COSMO/XE6 physics computation imbalance



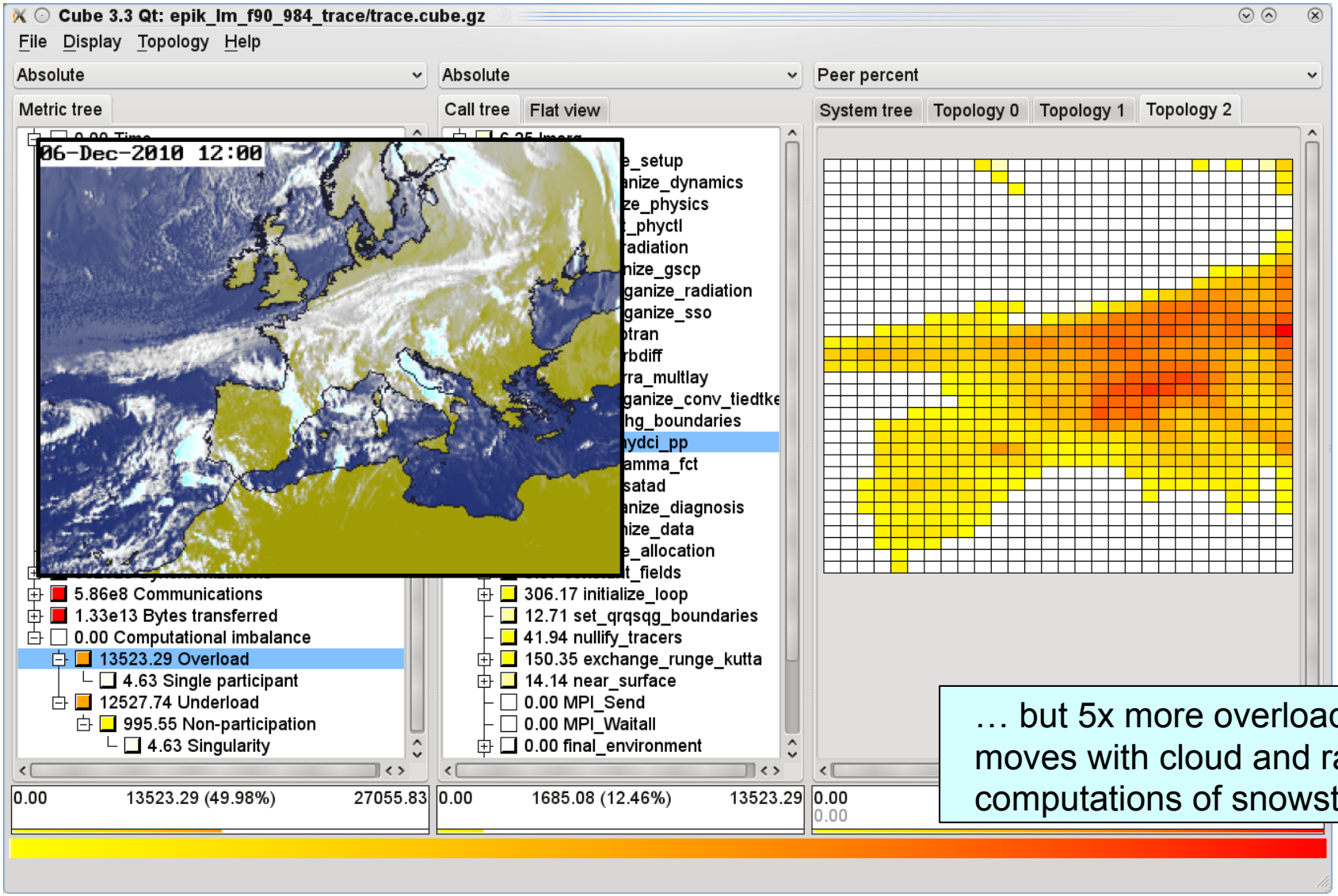
Refinement of origin of time executing local computation

COSMO/XE6 computational overload (geo)



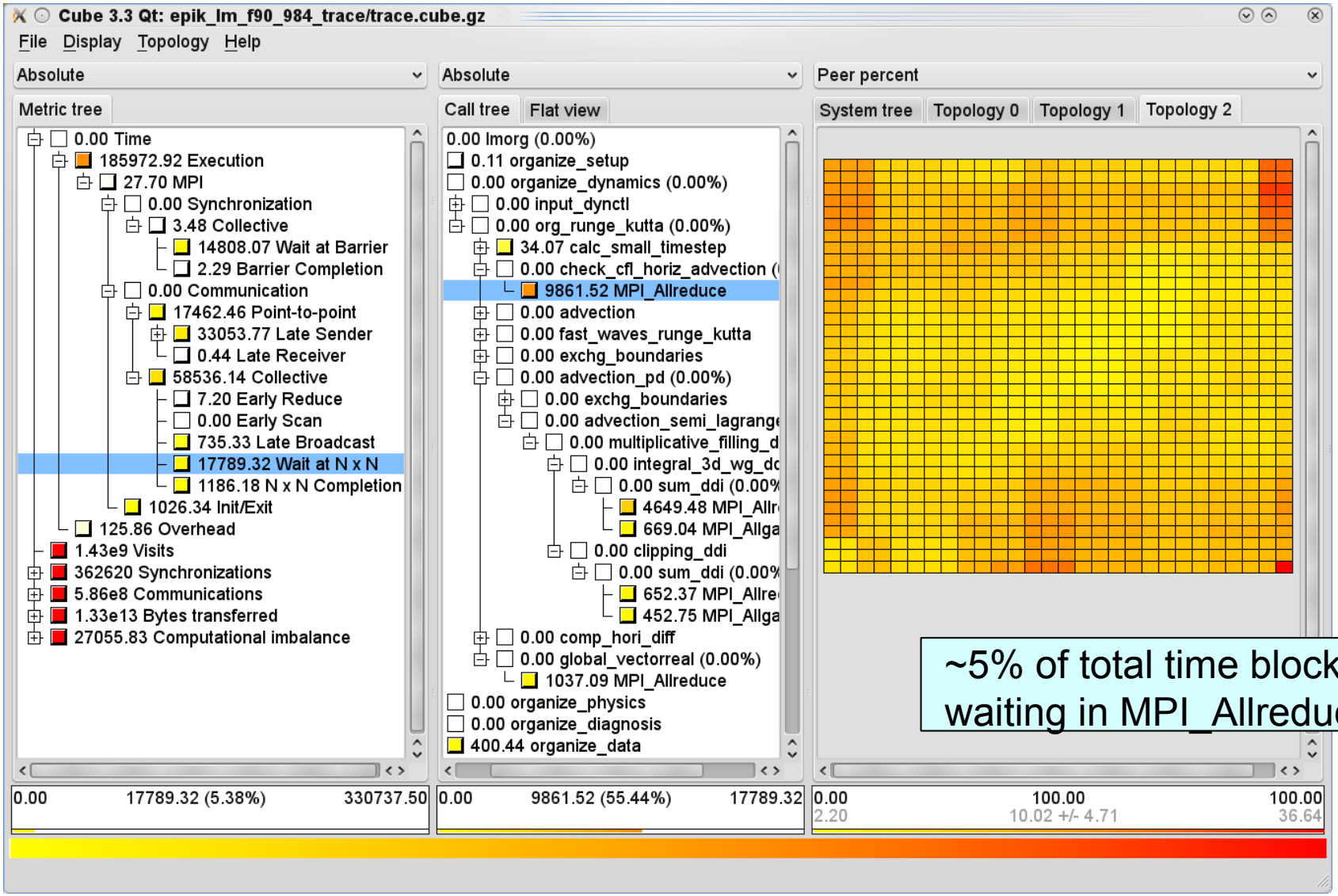
Geographical origin of some computational overload ...

COSMO/XE6 computational overload (hydro)



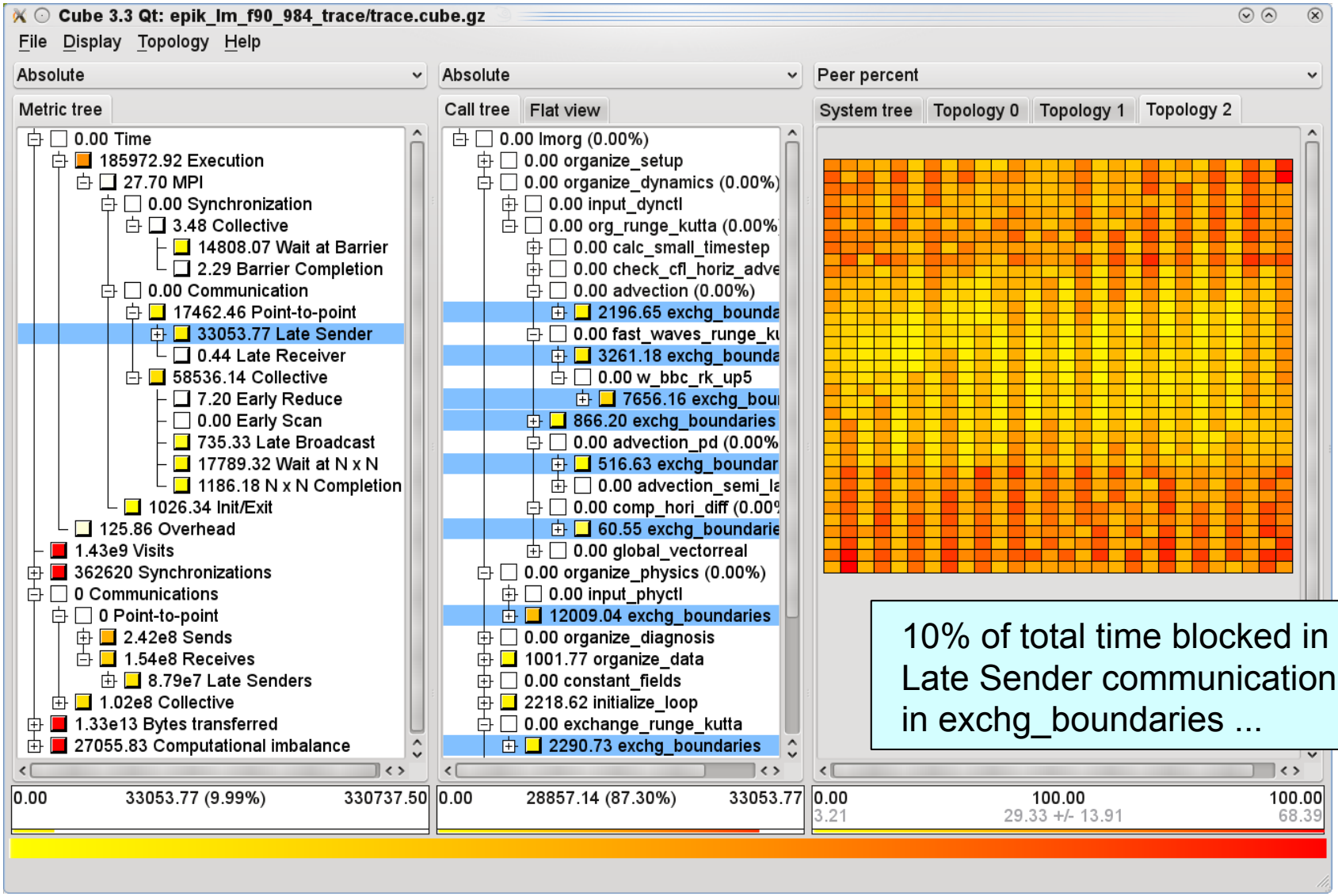
… but 5x more overload moves with cloud and rain computations of snowstorm

COSMO/XE6 collective wait at N x N time



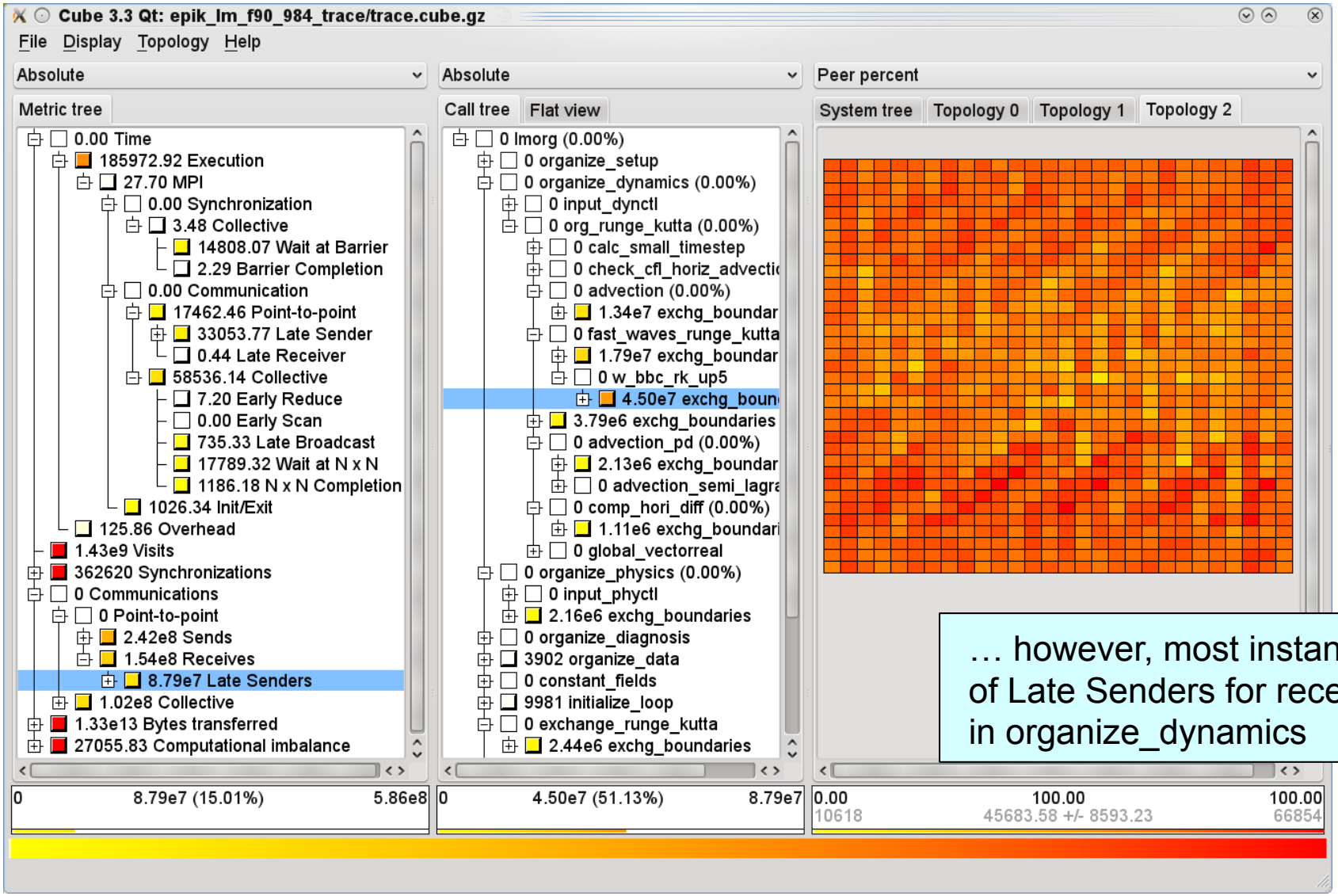
~5% of total time blocked waiting in MPI_Allreduces

COSMO/XE6 late sender waiting time



10% of total time blocked in Late Sender communication in exchg_boundaries ...

COSMO/XE6 late sender communications



… however, most instances of Late Senders for receives in organize_dynamics

Outline

- Introduction
- Code development
- Performance analysis and tuning
- Summary

Summary

You've been introduced to a variety of tools, and had an opportunity to try them with a prepared example code

- with guidance to apply and use the tools most effectively

Tools provide complementary capabilities

- computational kernel & processor analyses
- communication/synchronization analyses
- load-balance, scheduling, scaling, …

Tools are designed with various trade-offs

- general-purpose versus specialized
- platform-specific versus agnostic
- simple/basic versus complex/powerful

Tool selection

Which tools you use and when you use them likely to depend on situation

- which are available on (or for) your computer system
- which support your programming paradigms and languages
- which you are familiar (comfortable) with using

also depends on the type of issue you have or suspect

Awareness of (potentially) available tools can help finding the most appropriate tools

Workflow (getting started)

First ensure that the parallel application runs correctly

- No-one will care how quickly you can get invalid answers or produce a directory full of corefiles
- Parallel debuggers help isolate known problems
- Correctness checking tools can help identify other issues
- (that might not cause problems right now, but will eventually)
  - *e.g., race conditions, invalid/non-compliant usage*

Generally valuable to start with an overview of execution performance

- Fraction of time spent in computation vs comm/synch vs I/O
- Which sections of the application/library code are most costly

and how it changes with scale or different configurations

- Processes vs threads, mappings, bindings

Workflow (communication/synchronization)

Communication/synchronization issues generally apply to every computer system (to different extents) and typically grow with the number of processes/threads

- *Weak scaling*: fixed computation per thread, and perhaps fixed localities, but increasingly distributed
- *Strong scaling*: constant total computation, increasingly divided amongst threads, while communication grows
- Collective communication (particularly of type “all-to-all”) result in increasing data movement
- Synchronizations of larger groups are increasingly costly
- Load-balancing becomes increasingly challenging, and imbalances increasingly expensive
  - *generally manifests as waiting time at following collective ops*

Workflow (wasted waiting time)

Waiting times are difficult to determine in basic profiles

- Part of the time each process/thread spends in communication &
  synchronization operations may be wasted waiting time
- Need to correlate event times between processes/ threads
  - *Post-mortem event trace analysis avoids interference and provides a complete history*
  - *Scalasca automates trace analysis and ensures waiting times are completely quantified*
  - *Vampir allows interactive exploration and detailed examination of reasons for inefficiencies*

Workflow (core computation)

Effective computation within processors/cores is also vital

- Optimized libraries may already be available
- Optimization using compilers can also do a lot
  - *provided the code is clearly written and not too complex*
  - *appropriate directives and other hints can also help*
- Processor hardware counters can also provide insight
  - *although hardware-specific interpretation required*
- Tools available from processor and system vendors help navigate and interpret processor-specific performance issues

Presented tools

Score-P

- community-developed instrumenter & measurement libraries for parallel profiling and event tracing

Scalasca

- automated event-trace analysis

CUBE

- interactive parallel profile analyses

Vampir

- interactive event-trace visualizations and analyses

TAU

- comprehensive performance system