

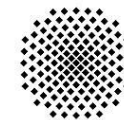
Performance Analysis

An Introduction

October 9, 2013 | Florian Janetzko, [f.janetzko\(at\)fz-juelich.de](mailto:f.janetzko(at)fz-juelich.de)

Acknowledgements

Slides taken partially from the
Virtual Institute – High Productivity Supercomputing (VI-HPS)
<http://www.vi-hps.org>



Universität Stuttgart

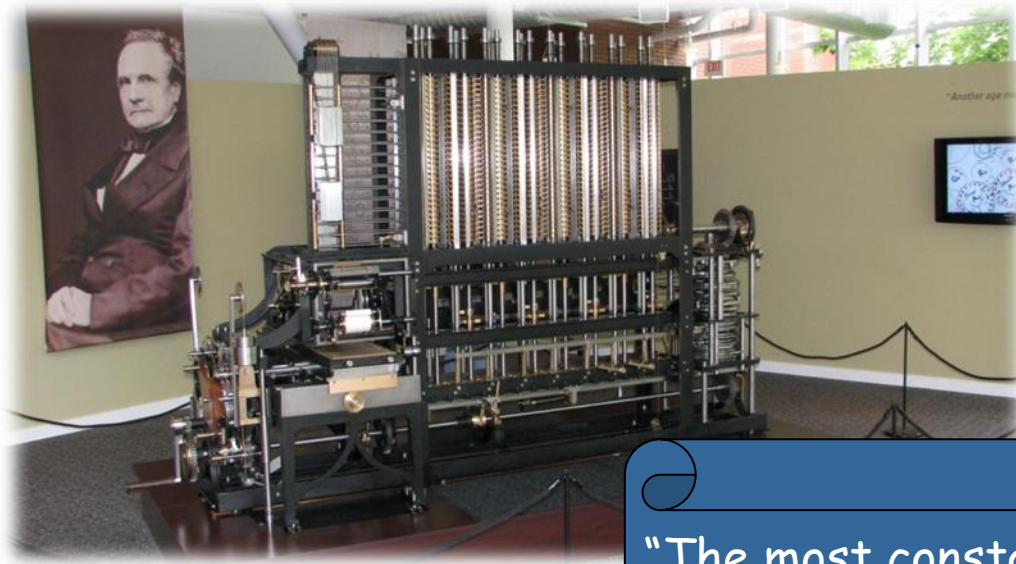


UNIVERSITY OF OREGON

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

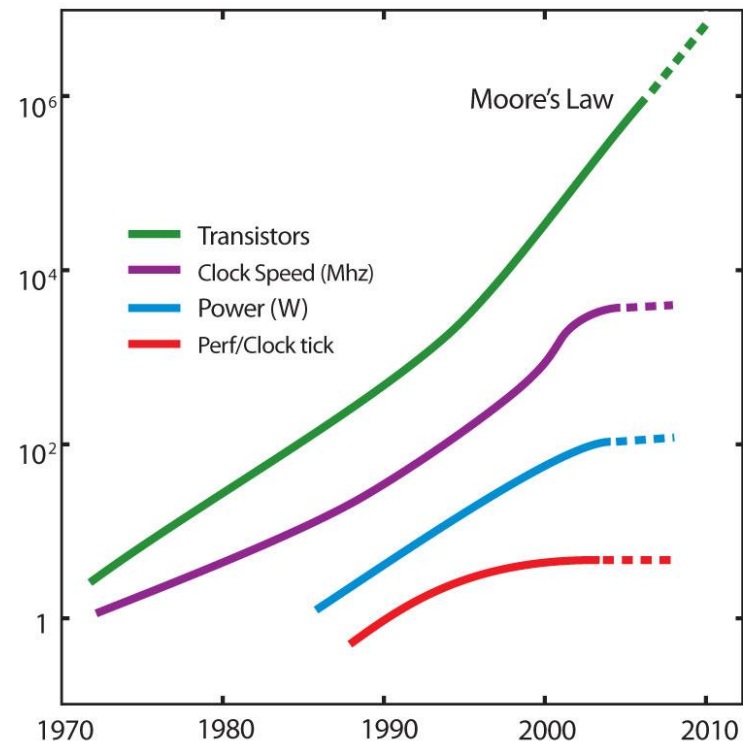
HPC Hardware Development

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!

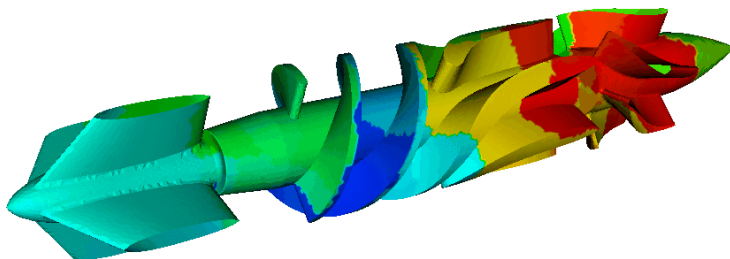
Example: XNS

CFD simulation of unsteady flows

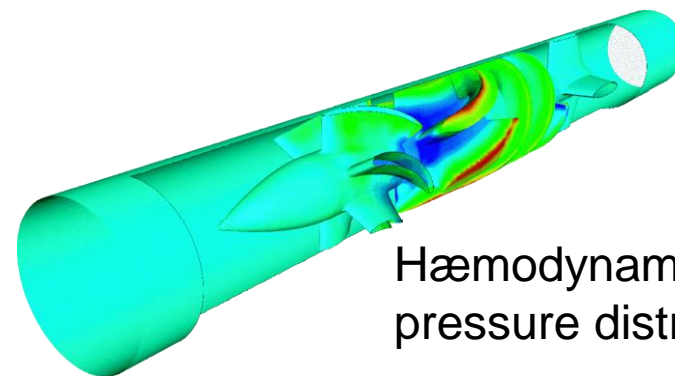
- Developed by CATS / RWTH Aachen
- Exploits finite-element techniques, unstructured 3D meshes, iterative solution strategies

MPI parallel version

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

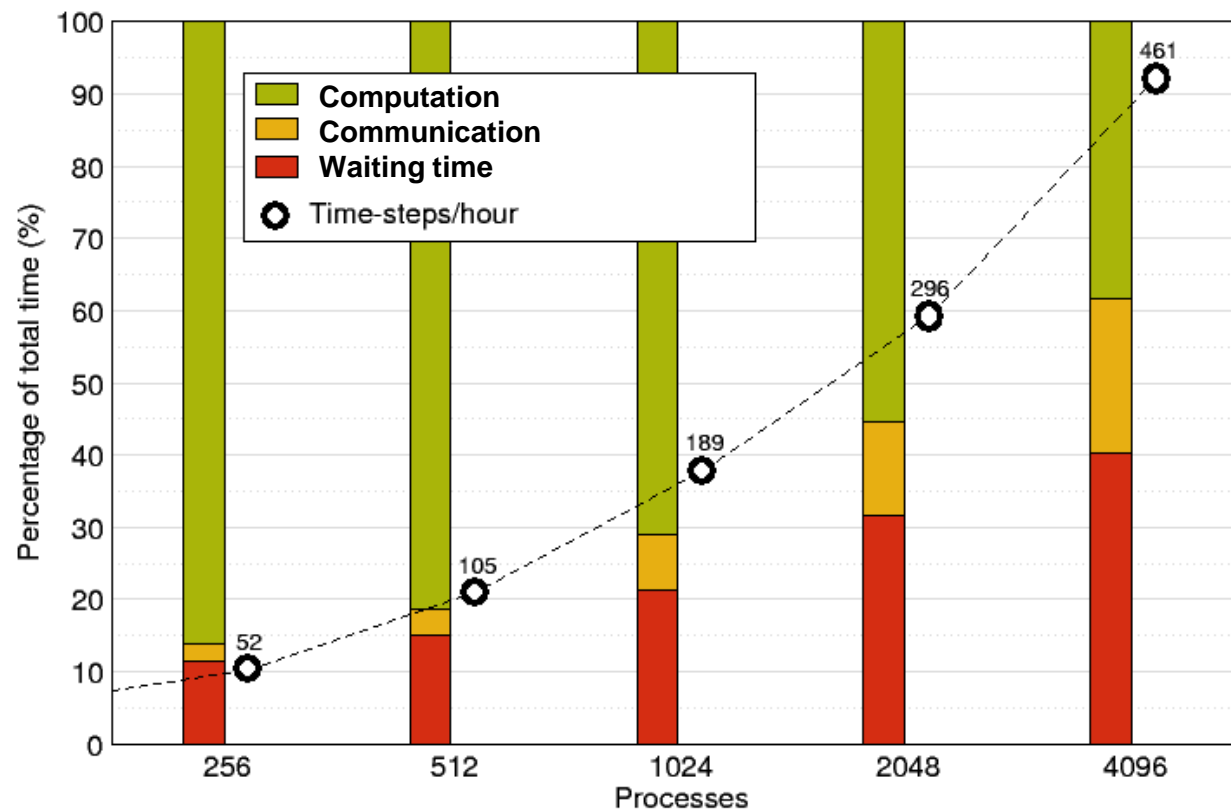


Partitioned finite-element mesh



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
 - 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. eclipse), syntax checker (e.g. forcheck)

2. Debugging

- Tools: write/printf statements, classical debuggers (TotalView, DDT, GDB, ...), MARMOT (for MPI codes), Intel threadchecker (for OpenMP codes)

3. Performance

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



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:

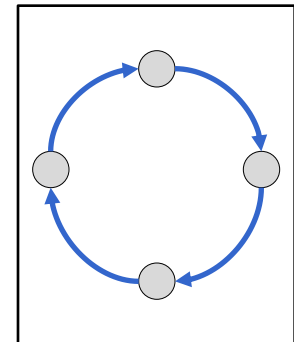
Variable	Possible values
MARMOT_DEBUG_MODE	0: errors 1: errors and warnings 2: errors, warnings and remarks
MARMOT_LOGFILE_TYPE	0: ASCII 1: HTML 2: CUBE

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

```
46    ...
47    left  = (myrank-1+nranks)%nranks;
48    right = (myrank+1)%nranks;
49
50    for (i=1;i<=nranks;i++)
51    {
52        summe = recvbuf + myrank;
53        MPI_Ssend(&summe, 1, MPI_INT, right, myrank,
                   MPI_COMM_WORLD);
54        MPI_Recv(&recvbuf, 1, MPI_INT, left, left,
                  MPI_COMM_WORLD, &status);
55        MPI_Wait(&request, &status);
56    }
57    ...
```



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

Intel Inspector XE 2013

Project Navigator: /lustre/

- threading
 - r000ti2
 - r001ti2
 - r002ti2

Target: r000ti2

Detect Deadlocks and Data Races

Summary

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:

- 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))

Code Locations: Data race

Description	Source	Function	Module
Write	find_and_fix_threading_errors.cpp... render_one_pi...	tachyon.find_and_fix_threading_er...	tachyon.find_and_fix_threading_er...
103	primary.scene = &scene;		
104			
105	col=trace(&primary); //Threading Error		
106	//2 ways to fix this threading error		
107	// 1) Make col a local variable		
Write	find_and_fix_threading_errors.cpp... render_one_pi...	tachyon.find_and_fix_threading_er...	tachyon.find_and_fix_threading_er...
103	primary.scene = &scene;		
104			
105	col=trace(&primary); //Threading Error		
106	//2 ways to fix this threading error		
107	// 1) Make col a local variable		

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
 - 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
```

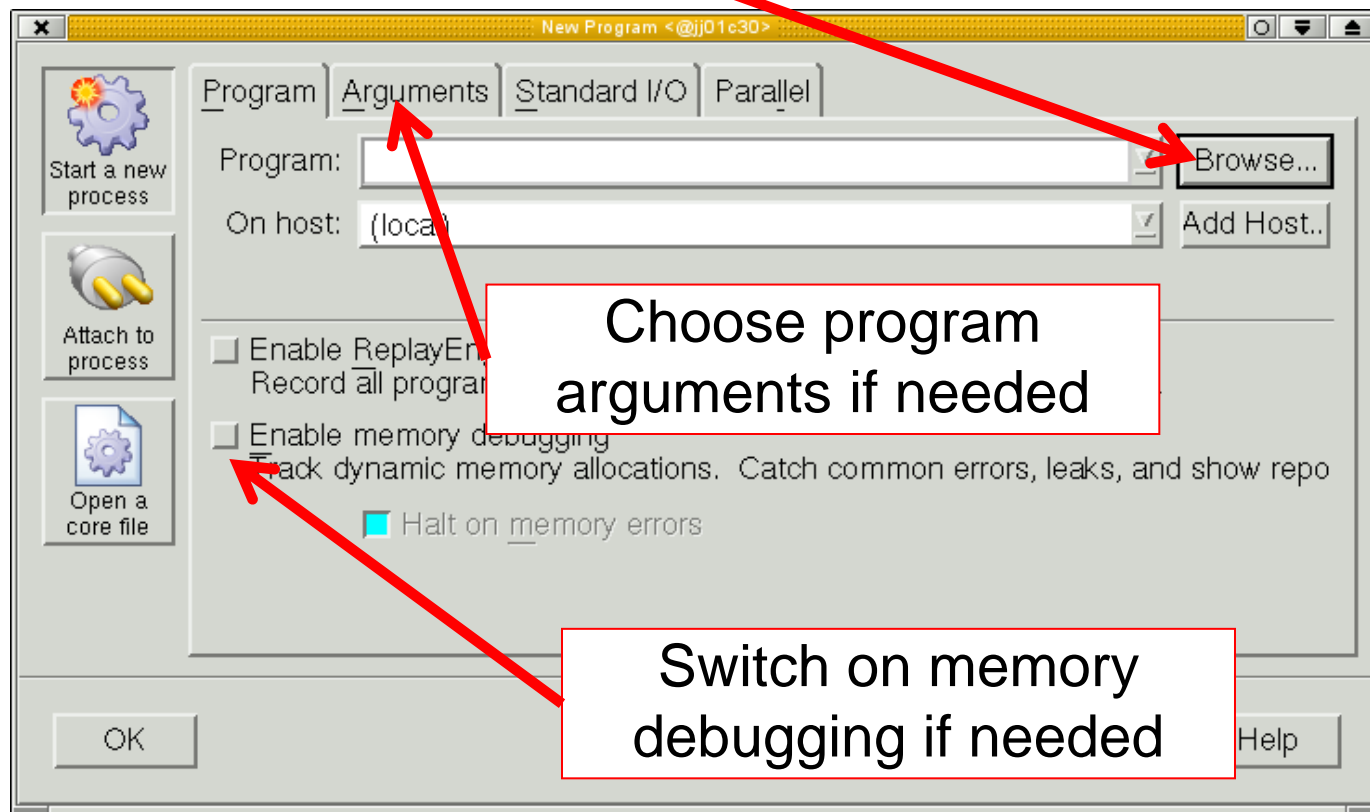
- -g -O0 also possible (as with most compilers)

TotalView execution modes

1. GUI
2. Script (tvscript)

Code Development – TotalView

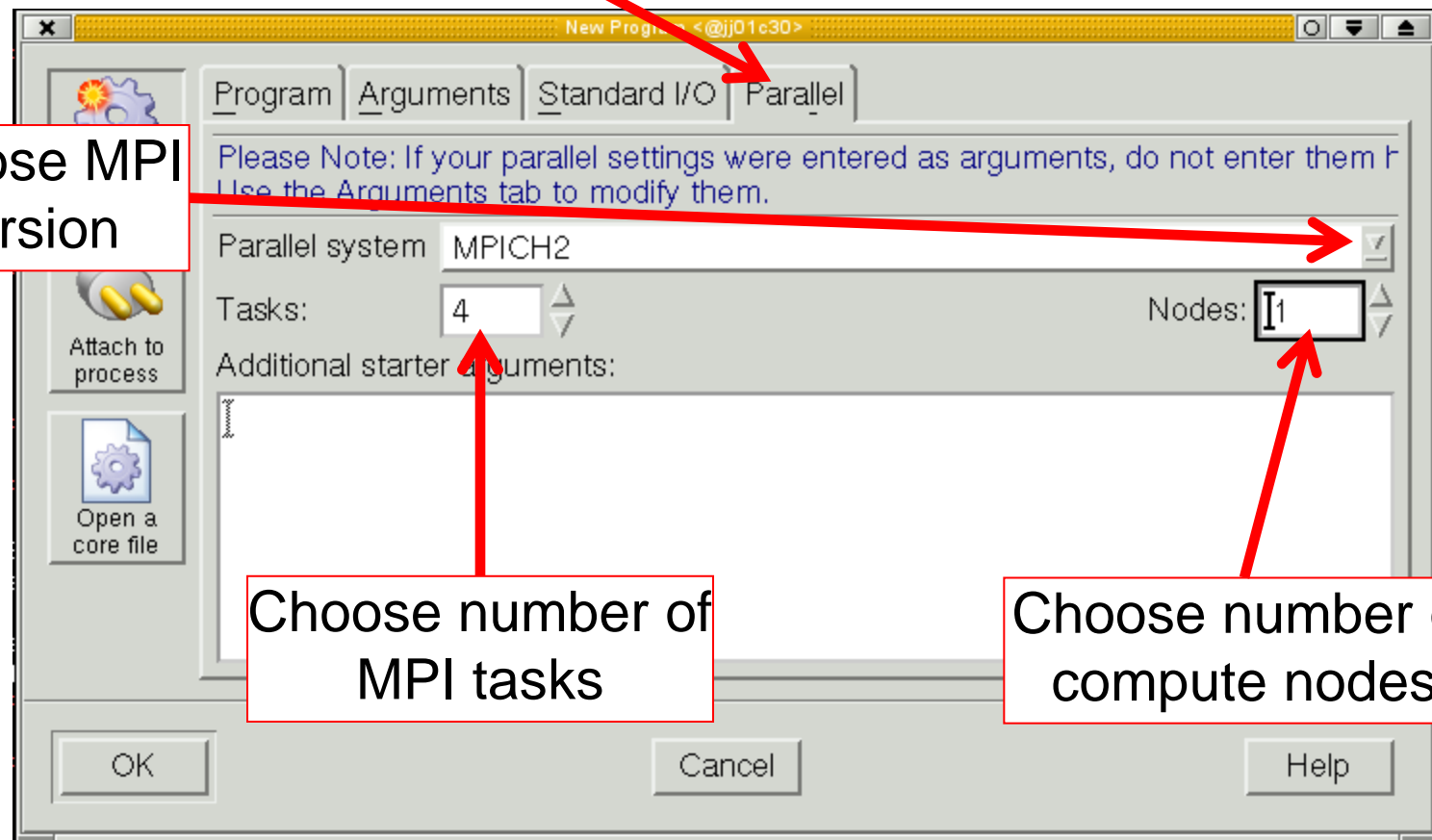
Choose your executable



Code Development – TotalView

Choose MPI settings

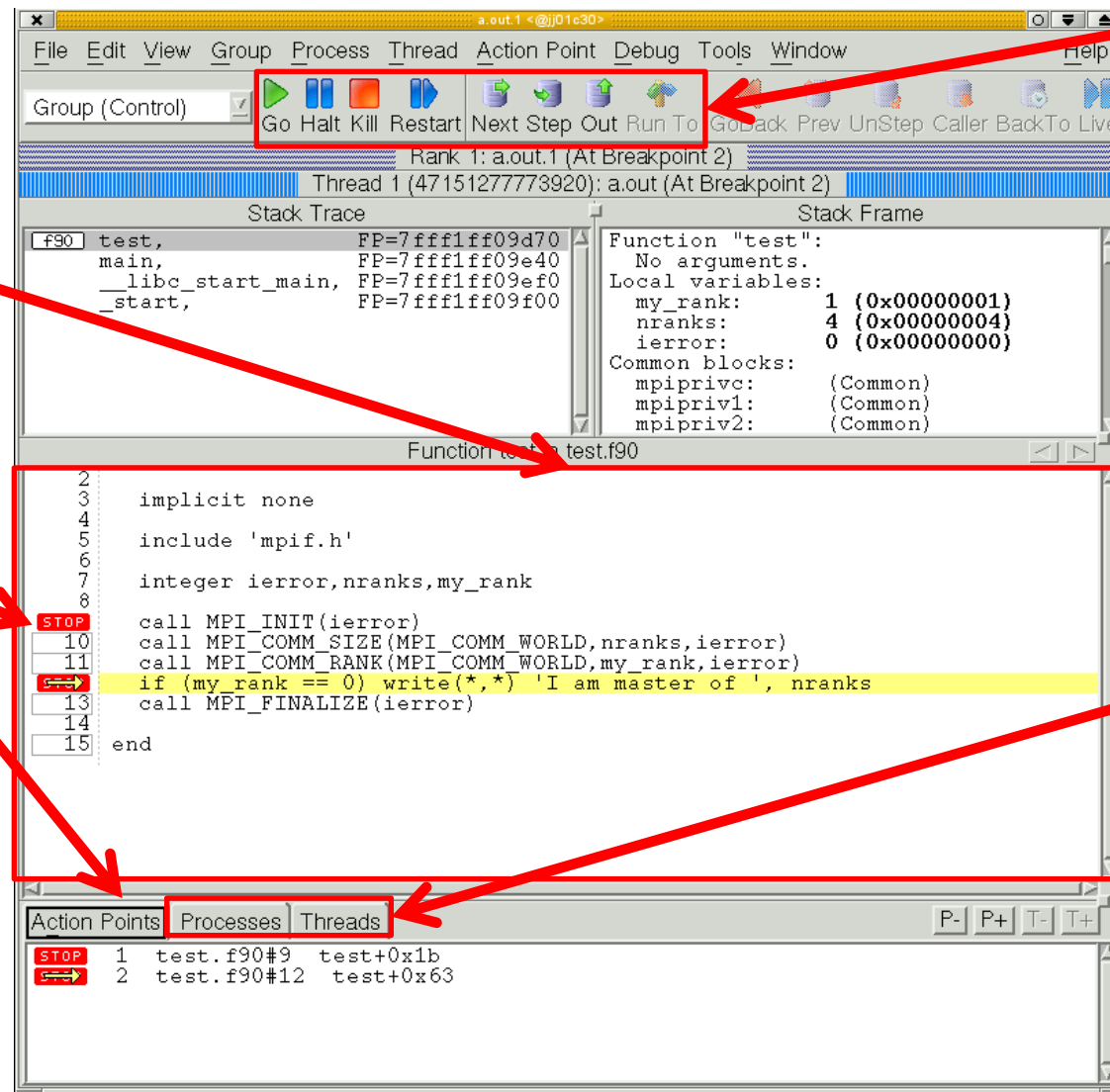
Choose MPI
version



Choose number of
MPI tasks

Choose number of
compute nodes

Code Development – TotalView



The screenshot shows the TotalView IDE interface with several components and annotations:

- Navigation:** A red box highlights the navigation toolbar at the top, containing icons for Go, Halt, Kill, Restart, Next Step, Out, Run To, Go Back, Prev, UnStep, Caller, Back To, and Live. A red arrow points from the "Navigation" label to this toolbar.
- Source code window:** A red box highlights the source code editor in the center, showing the code for `test.f90`. A red arrow points from the "Source code window" label to this area.
- Action points:** A red box highlights the "Action Points" tab at the bottom, showing a list of breakpoints. A red arrow points from the "Action points" label to this tab.
- Process and thread view:** A red box highlights the "Processes" and "Threads" tabs at the bottom, showing a list of processes and threads. A red arrow points from the "Process and thread view" label to this area.

The source code window displays the following code:

```

2
3  implicit none
4
5  include 'mpif.h'
6
7  integer ierror,nranks,my_rank
8
9  call MPI_INIT(ierror)
10 call MPI_COMM_SIZE(MPI_COMM_WORLD,nranks,ierror)
11 call MPI_COMM_RANK(MPI_COMM_WORLD,my_rank,ierror)
12 if (my_rank == 0) write(*,*) 'I am master of ', nranks
13 call MPI_FINALIZE(ierror)
14
15 end
  
```

The "Processes" tab shows the following data:

Process	Thread	Address
1	test.f90#9	test+0x1b
2	test.f90#12	test+0x63

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 optimizing compiler
- 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:

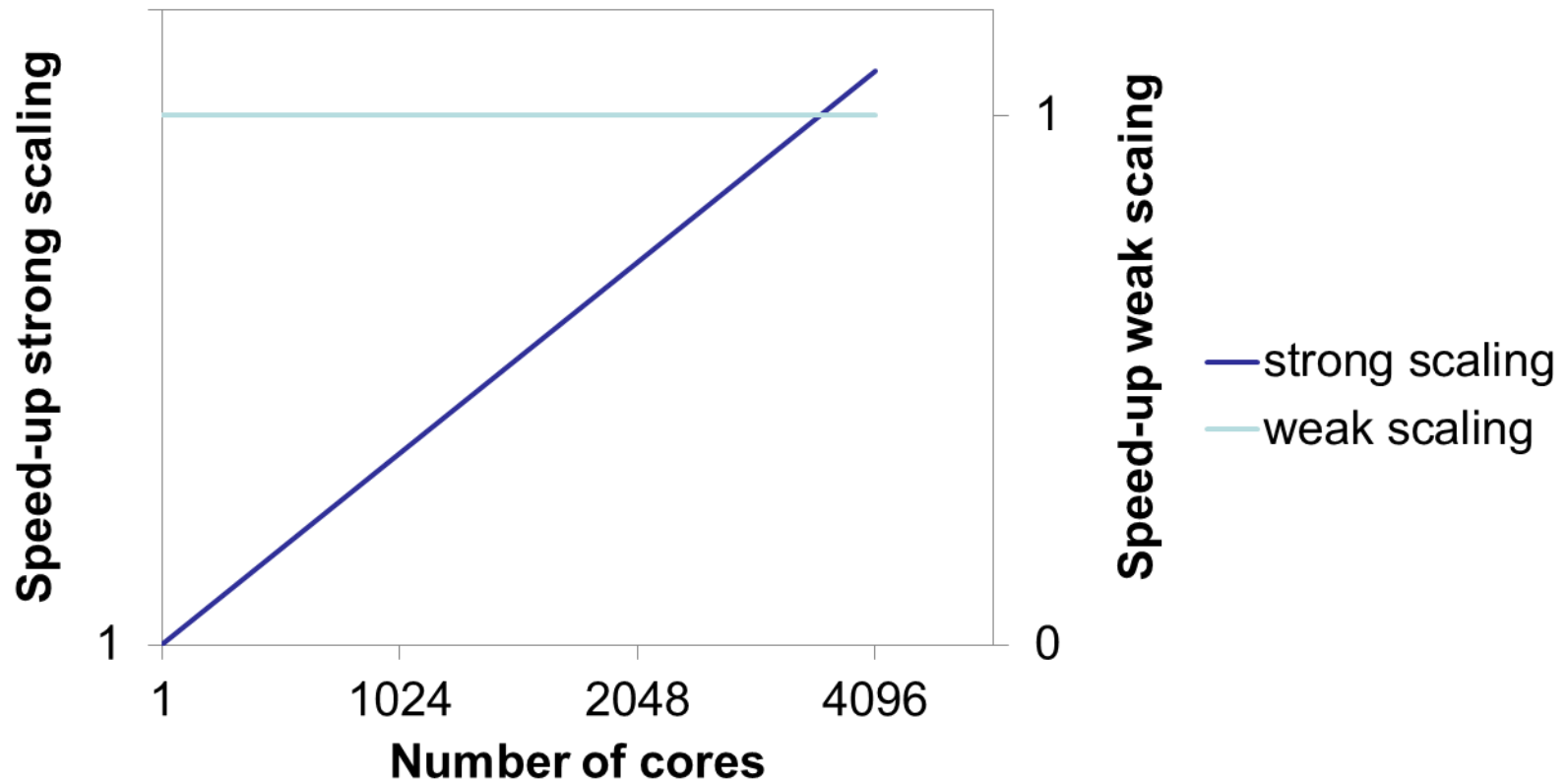
$$S(n) = \frac{t(1)}{t(n)}$$

$S(n)$: 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:

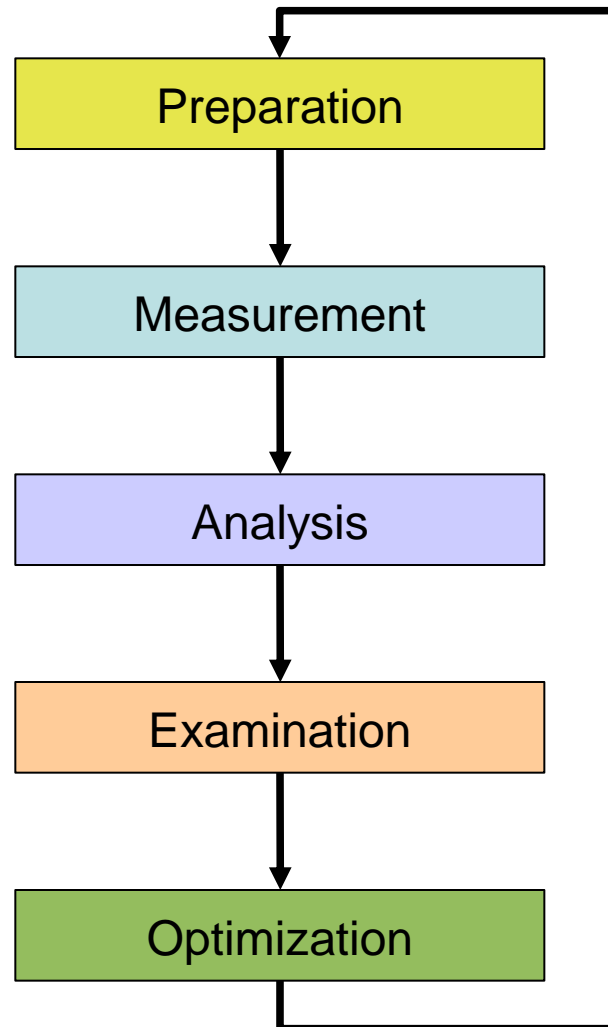
$$\alpha = 0.1$$

$$n = 8$$

$$\rightarrow S_r = 4.7$$

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

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

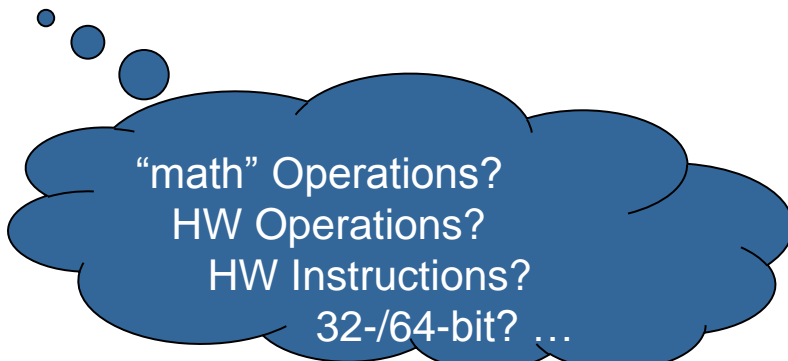
Derived metrics

- 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



“math” Operations?
HW Operations?
HW Instructions?
32-/64-bit? ...

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

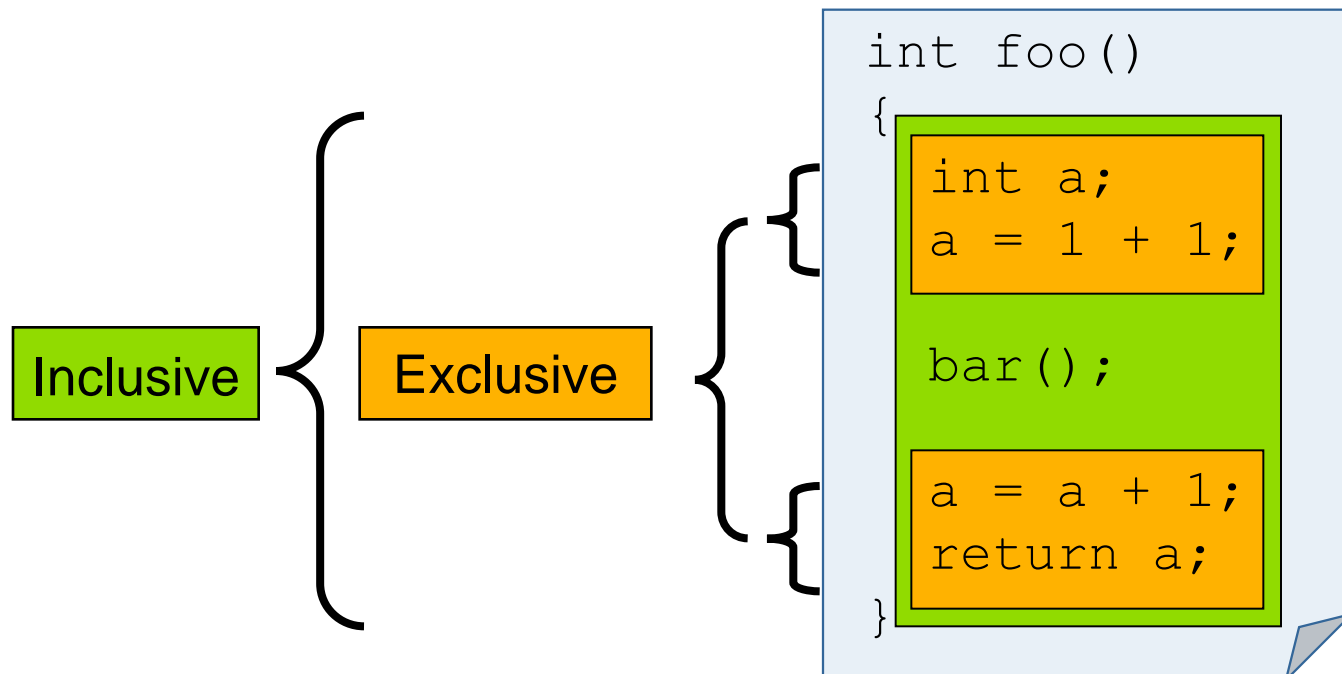
Inclusive vs. Exclusive Values

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

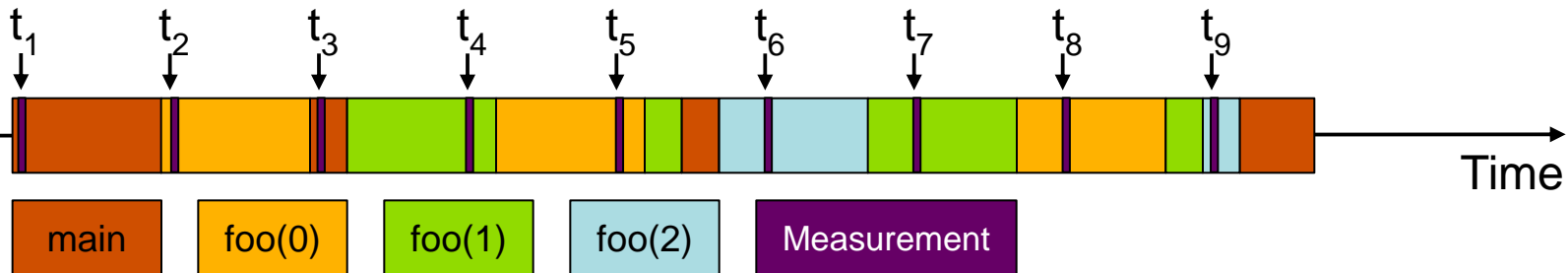
How is performance data recorded?

- 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

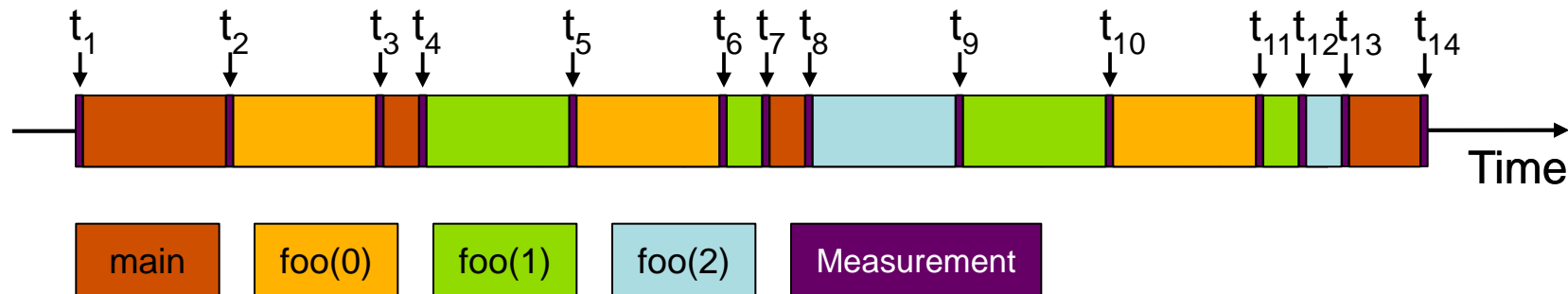
- Timer interrupt, OS signal, or HWC overflow
- Service routine examines return-address stack
- Addresses are mapped to routines using symbol table information

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

Process A

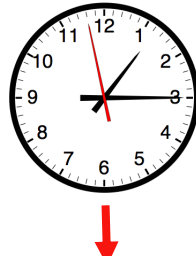
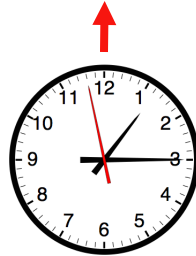
```
void foo() {  
  trc_enter("foo");  
  ...  
  trc_send(B);  
  send(B, tag, buf);  
  ...  
  trc_exit("foo");  
}
```

instrument

Process B

```
void bar() {  
  trc_enter("bar");  
  ...  
  recv(A, tag, buf);  
  trc_recv(A);  
  ...  
  trc_exit("bar");  
}
```

MONITOR



MONITOR

Local trace A

...		
58	ENTER	1
62	SEND	B
64	EXIT	1
...		

1	foo
...	

Local trace B

...		
60	ENTER	1
68	RECV	A
69	EXIT	1
...		

1	bar
...	

Global trace view

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

merge

unify

1	foo
2	bar
...	

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

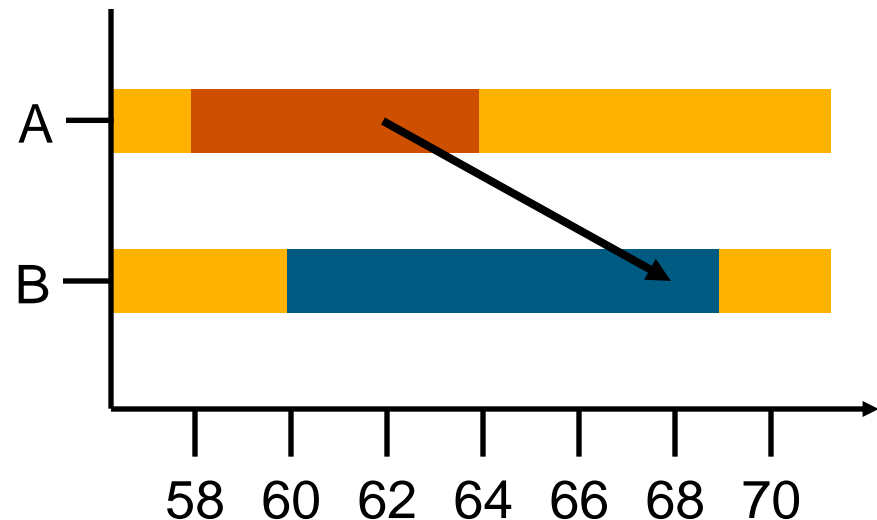
- Automatic search for bottlenecks
- Visual trace analysis
- Calculation of statistics

Example: Time-line Visualization

1	foo
2	bar
3	...

■	main
■	foo
■	bar

...			
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

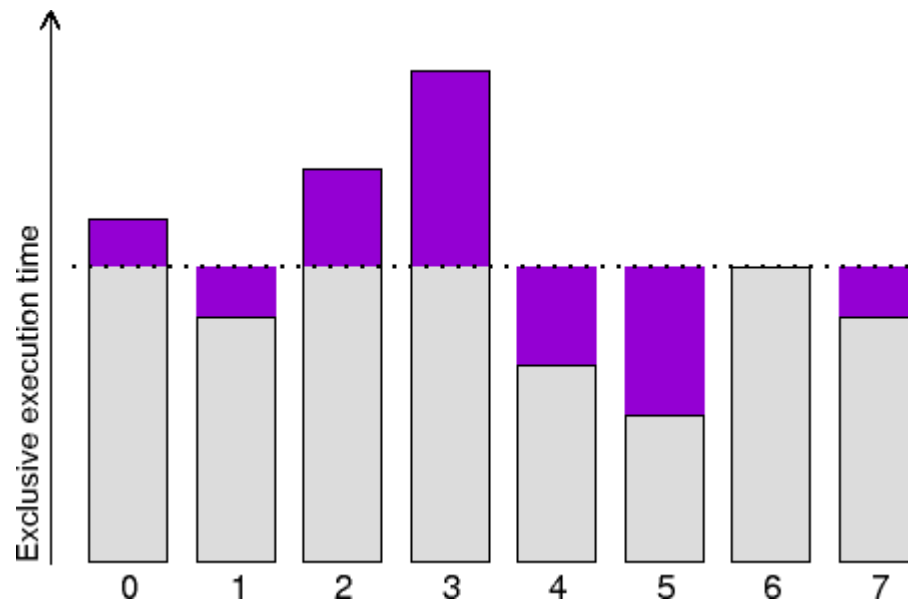
Computational Imbalance

Absolute difference to average exclusive execution time

- Focusses only on computational parts

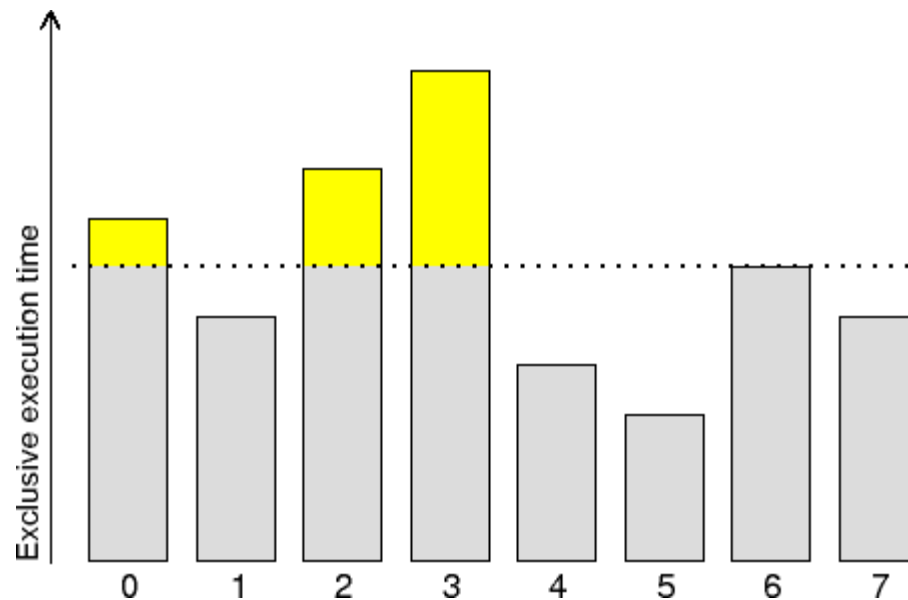
Captures global imbalances

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



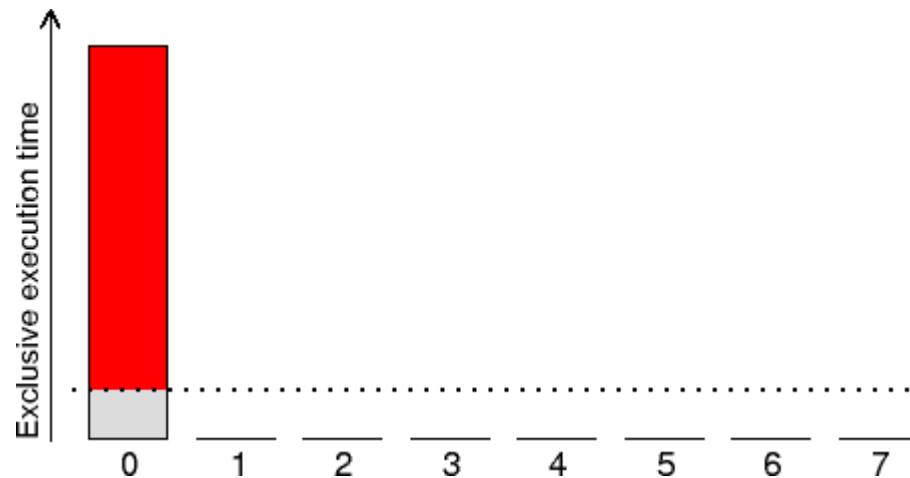
Overload

Processes/threads with exclusive execution time above average



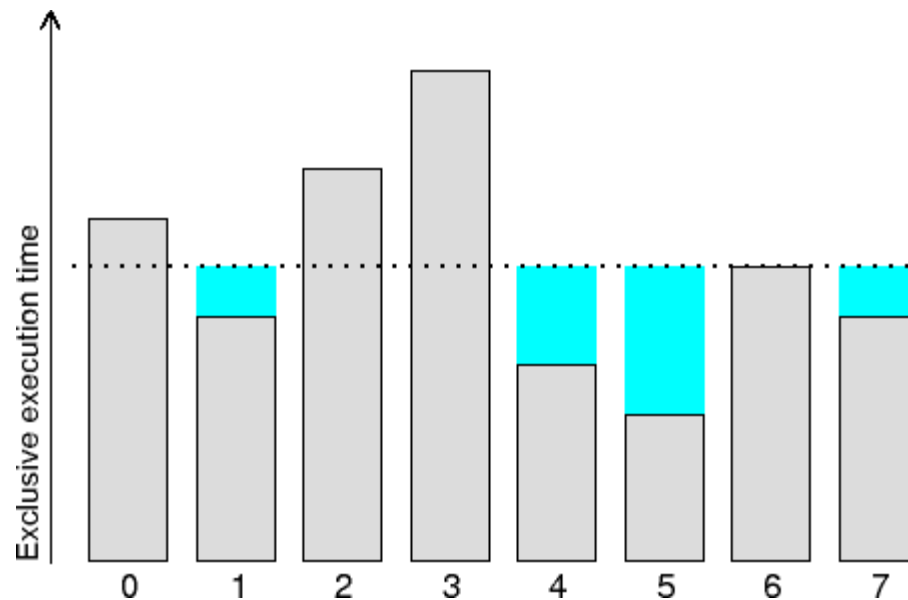
Overload, Single Participant

Call-paths executed by single process/thread



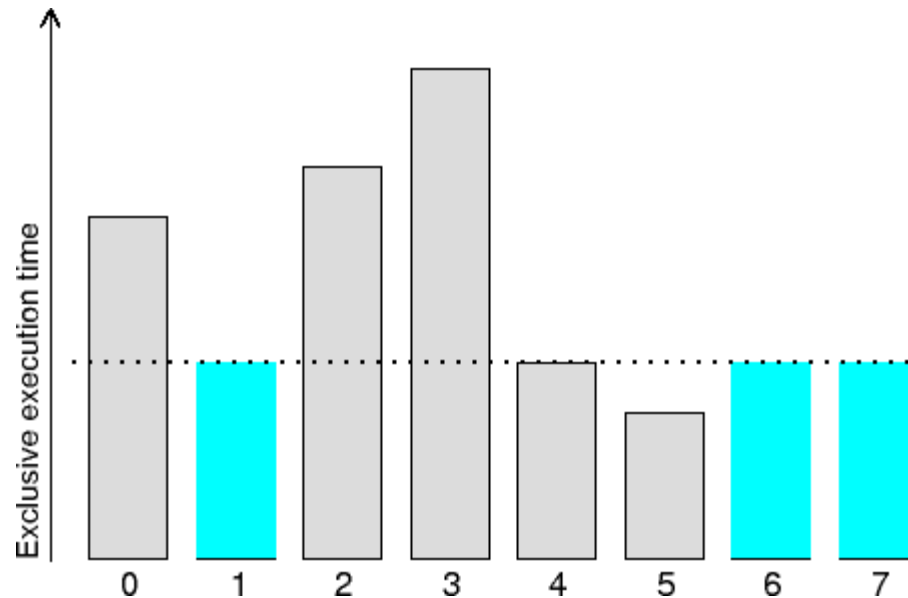
Underload

Processes/threads with exclusive execution time below average



Underload, Non-Participation

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

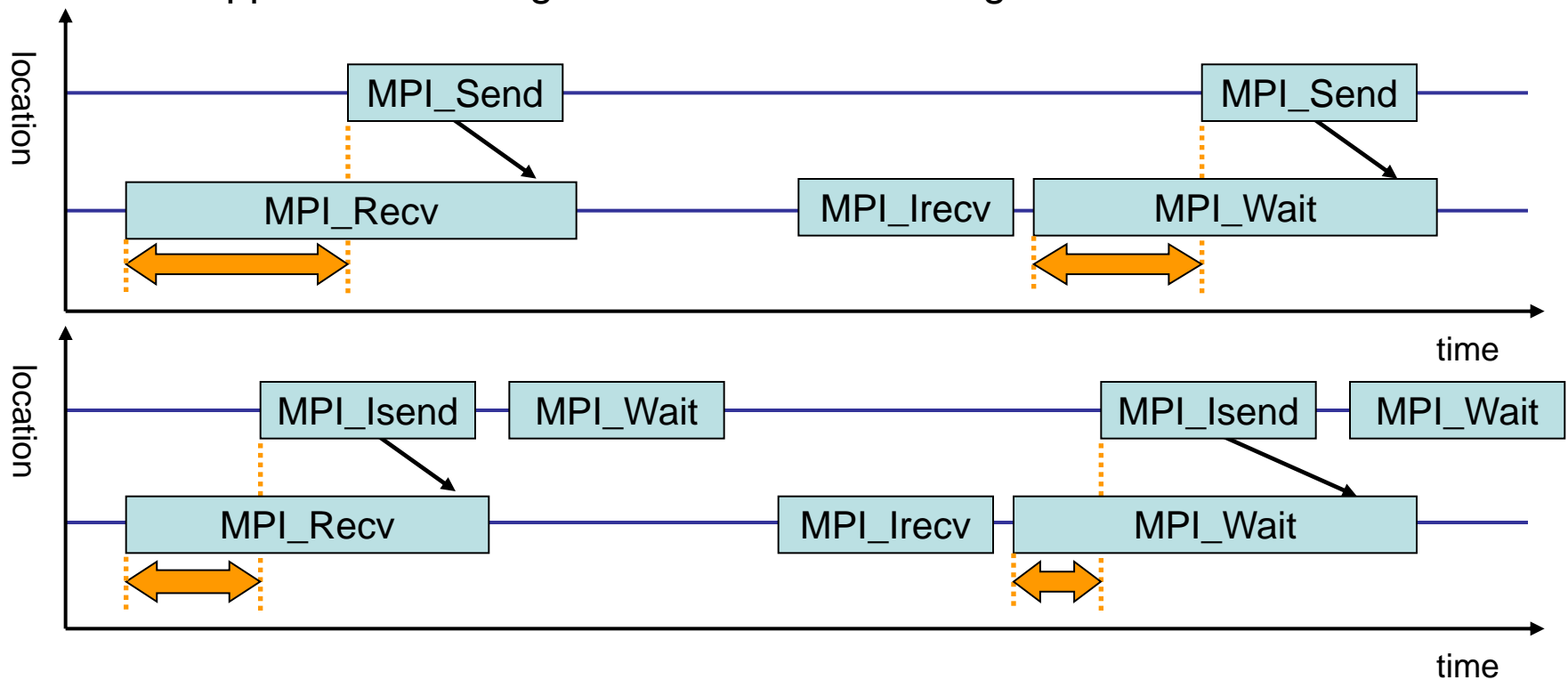


Outline

- 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

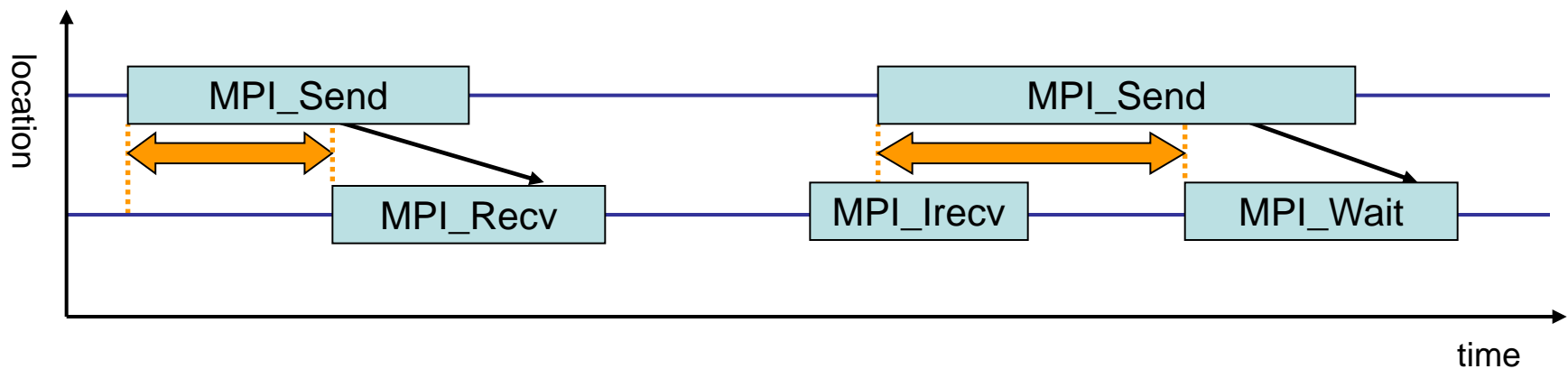
Late Sender

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



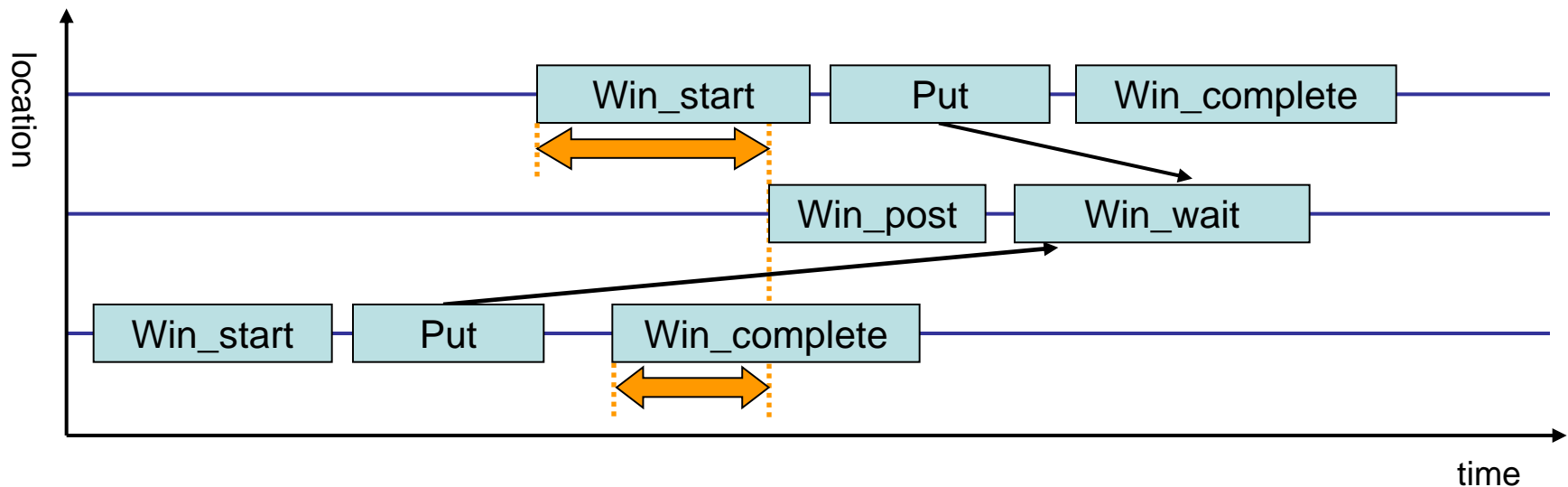
Late Receiver

- 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



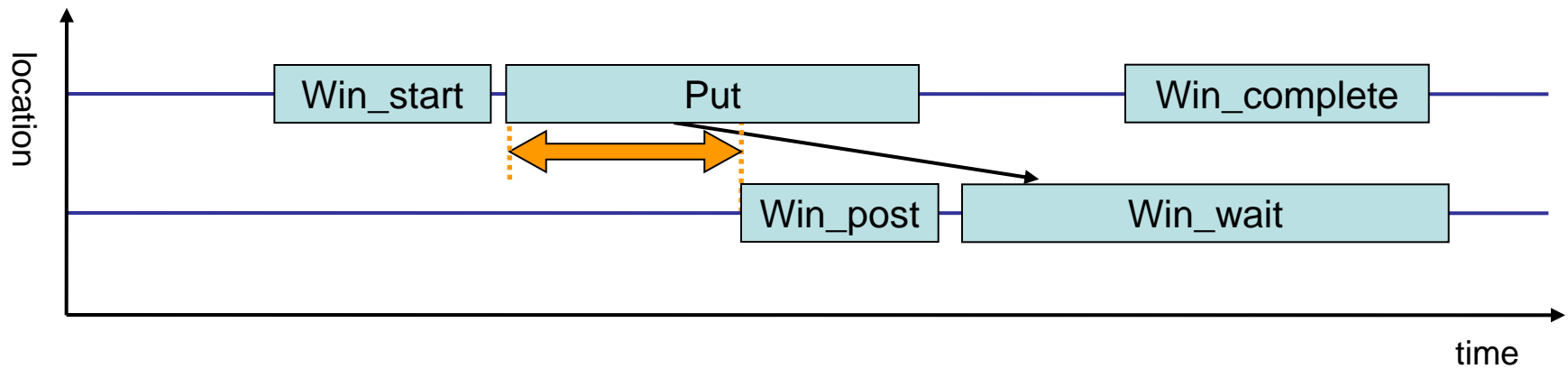
Late Post

- 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

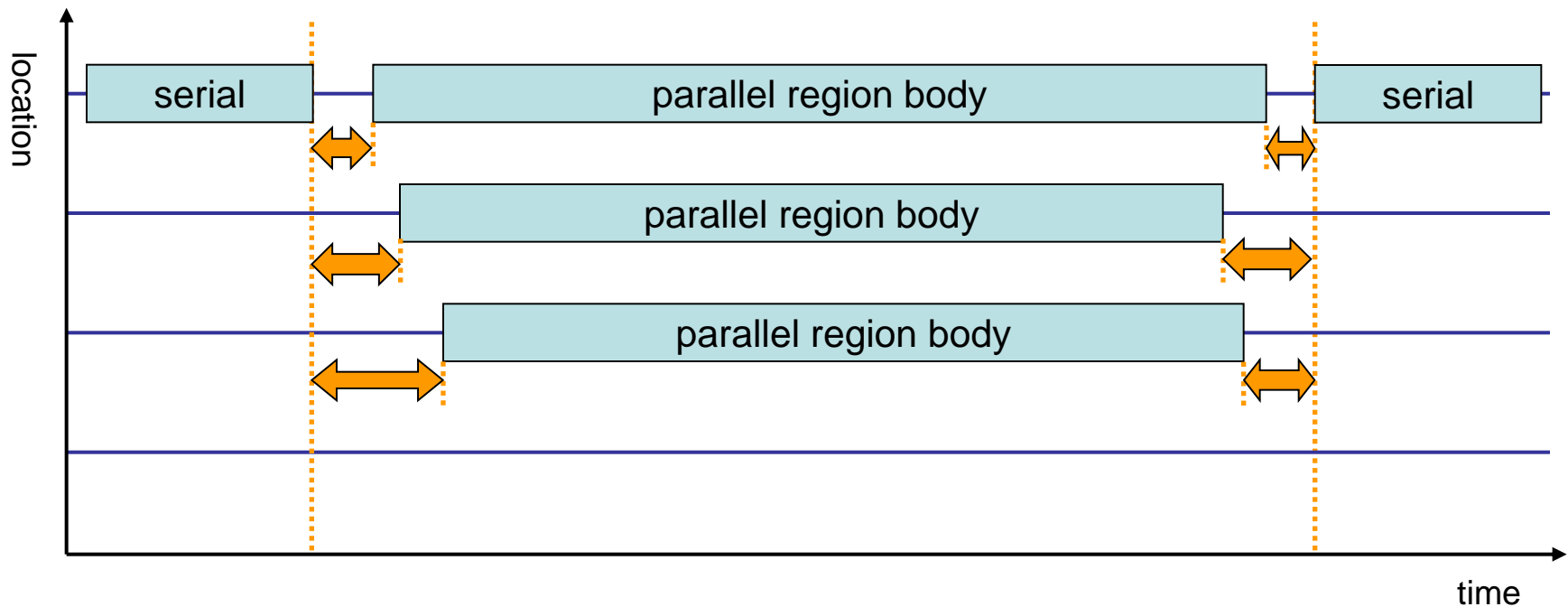


Outline

- 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

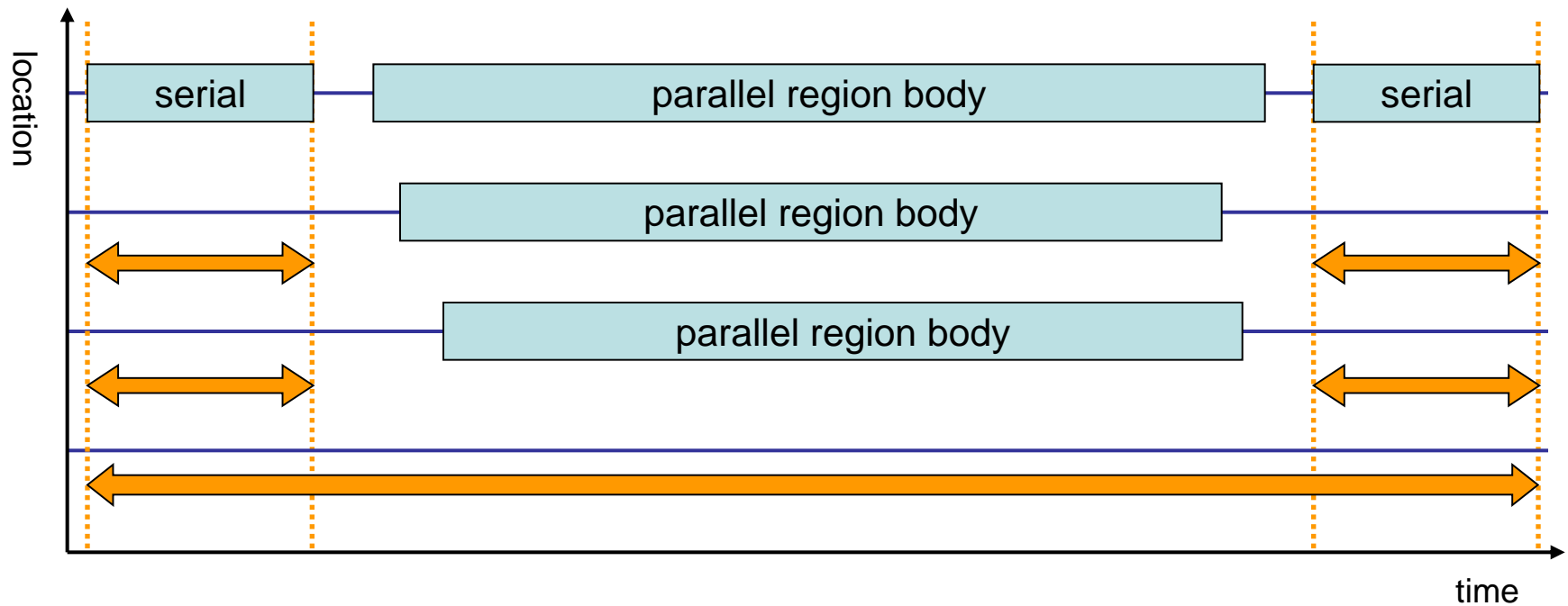
OpenMP Management Time

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



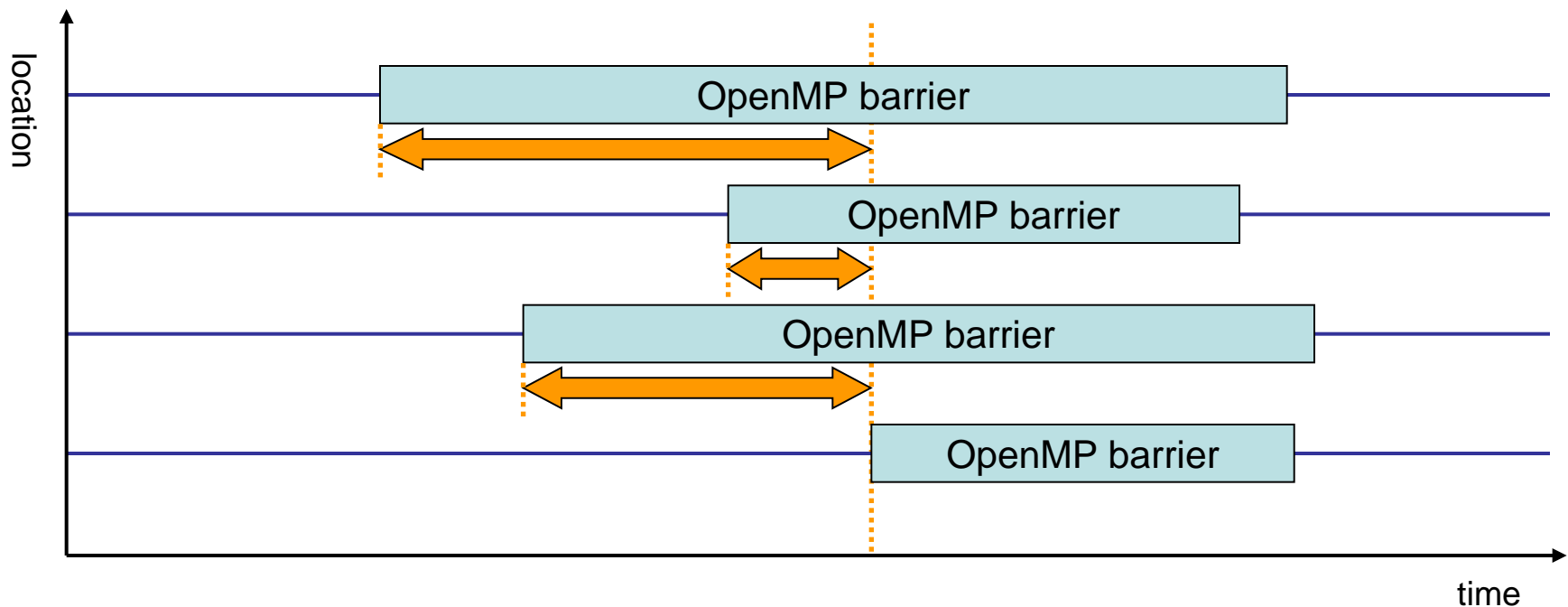
OpenMP Idle Threads

- 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



UNIVERSITY OF OREGON

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

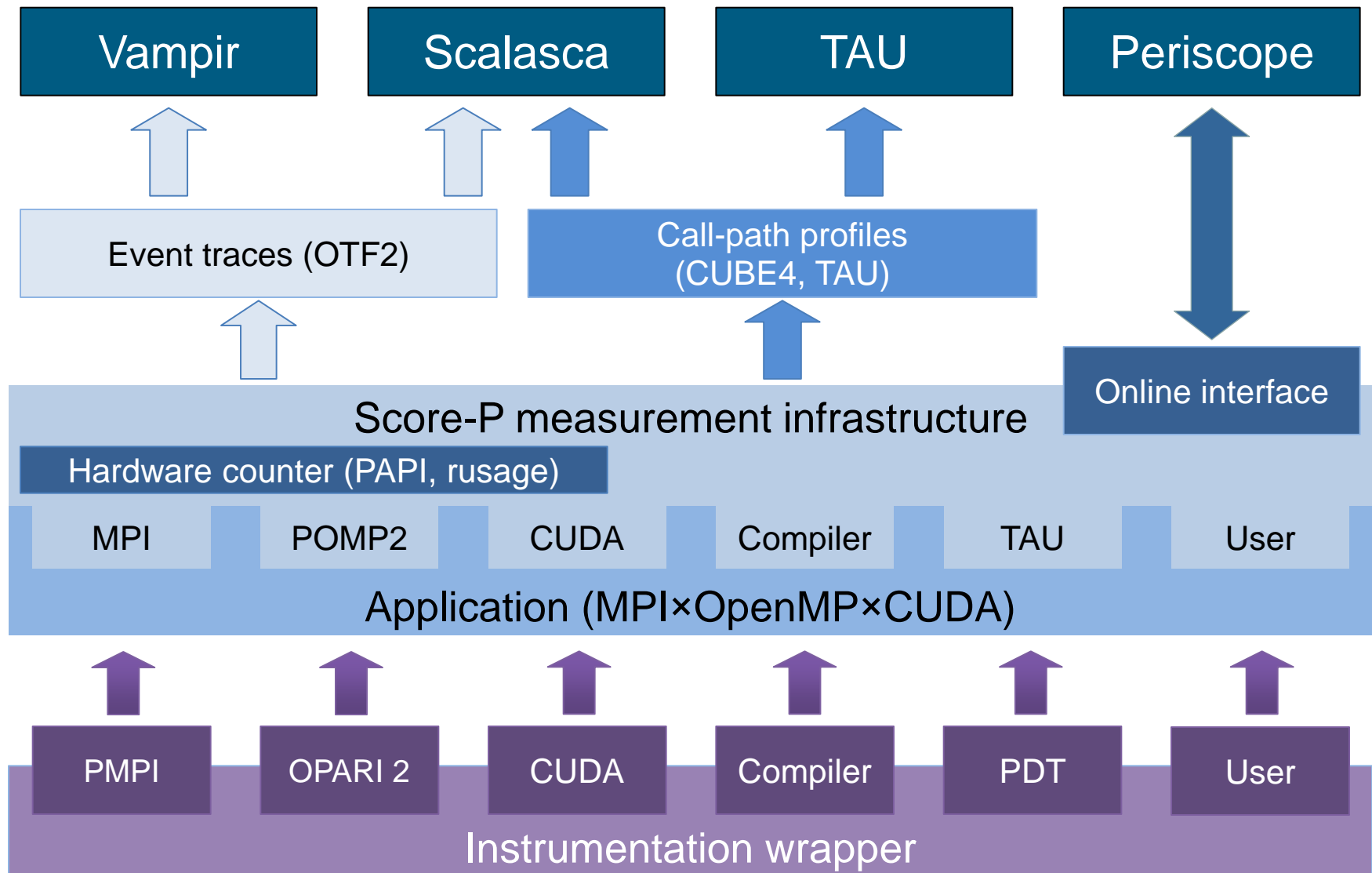
Functional requirements

- 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

<code>mpicc ...</code>	<code>→ scorep mpicc ...</code>
<code>mpif90 ...</code>	<code>→ scorep mpif90 ...</code>

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

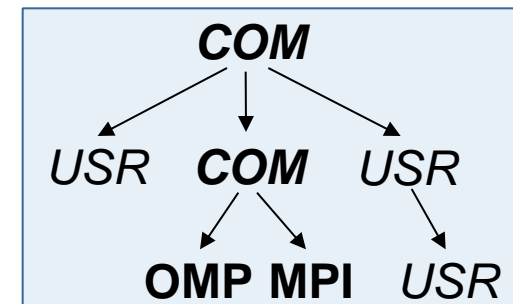
```
% scorep-score scorep_example_sum/profile.cubex
Estimated aggregate size of event trace (total_tbc): 35955109198 bytes
Estimated requirements for largest trace buffer (max_tbc): 9043348074 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid intermediate flushes
or reduce requirements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

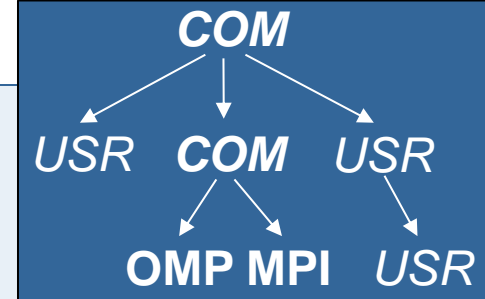
Score report breakdown by region

```
% 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 binvrhs_
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
binvrhs*
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 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, ...

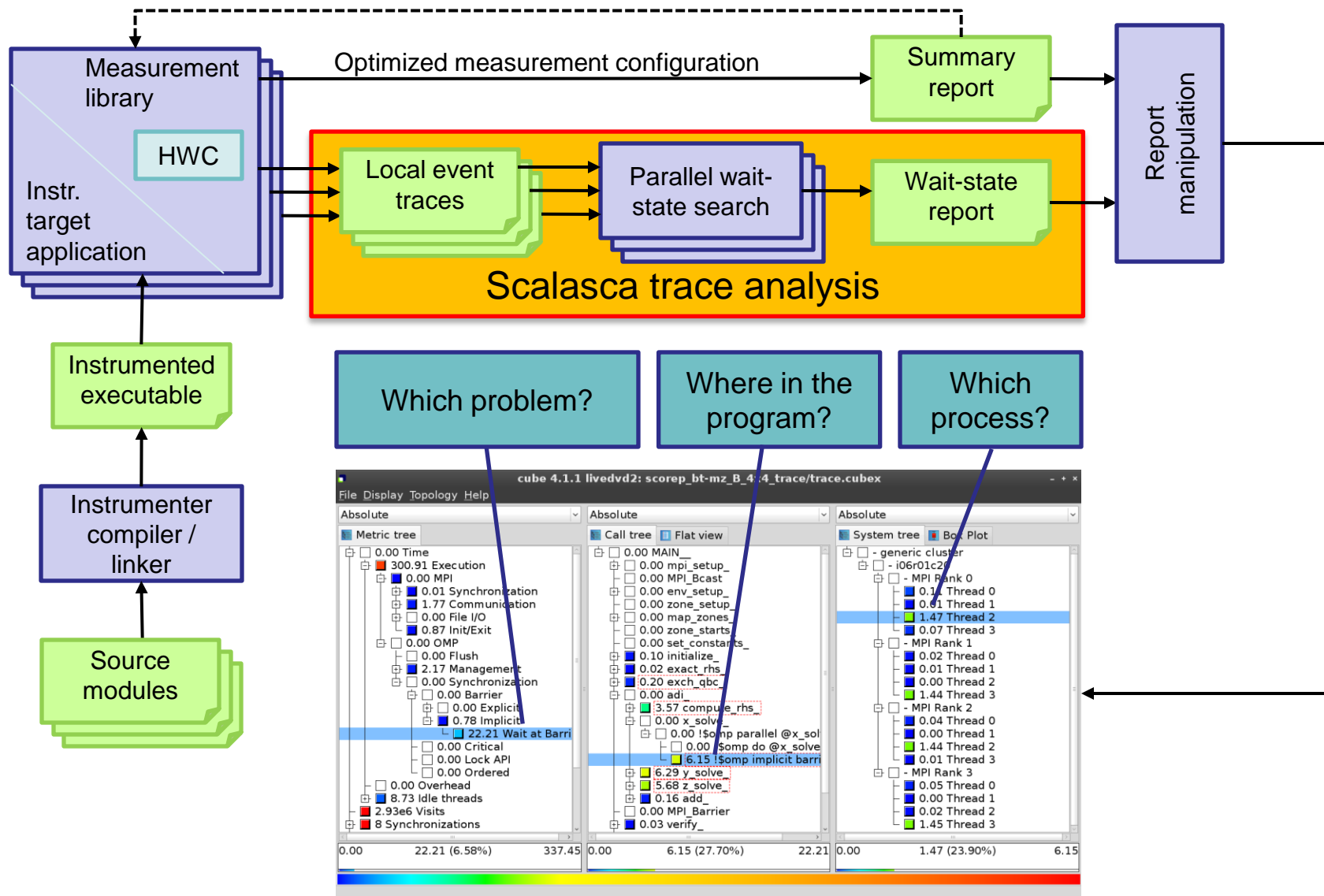
Uses Score-P instrumenter & measurement libraries

- 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 filtfiler  : 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)

Eample 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`

Example 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

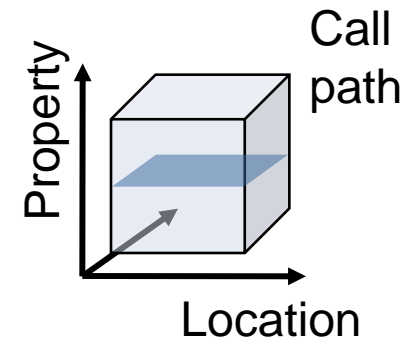
Now available as a separate component

- Can be installed independently of Score-P, e.g., on laptop or desktop
- Latest release: CUBE 4.2 (August 2013)

Analysis Presentation and Exploration

Representation of values (severity matrix) on three hierarchical axes

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

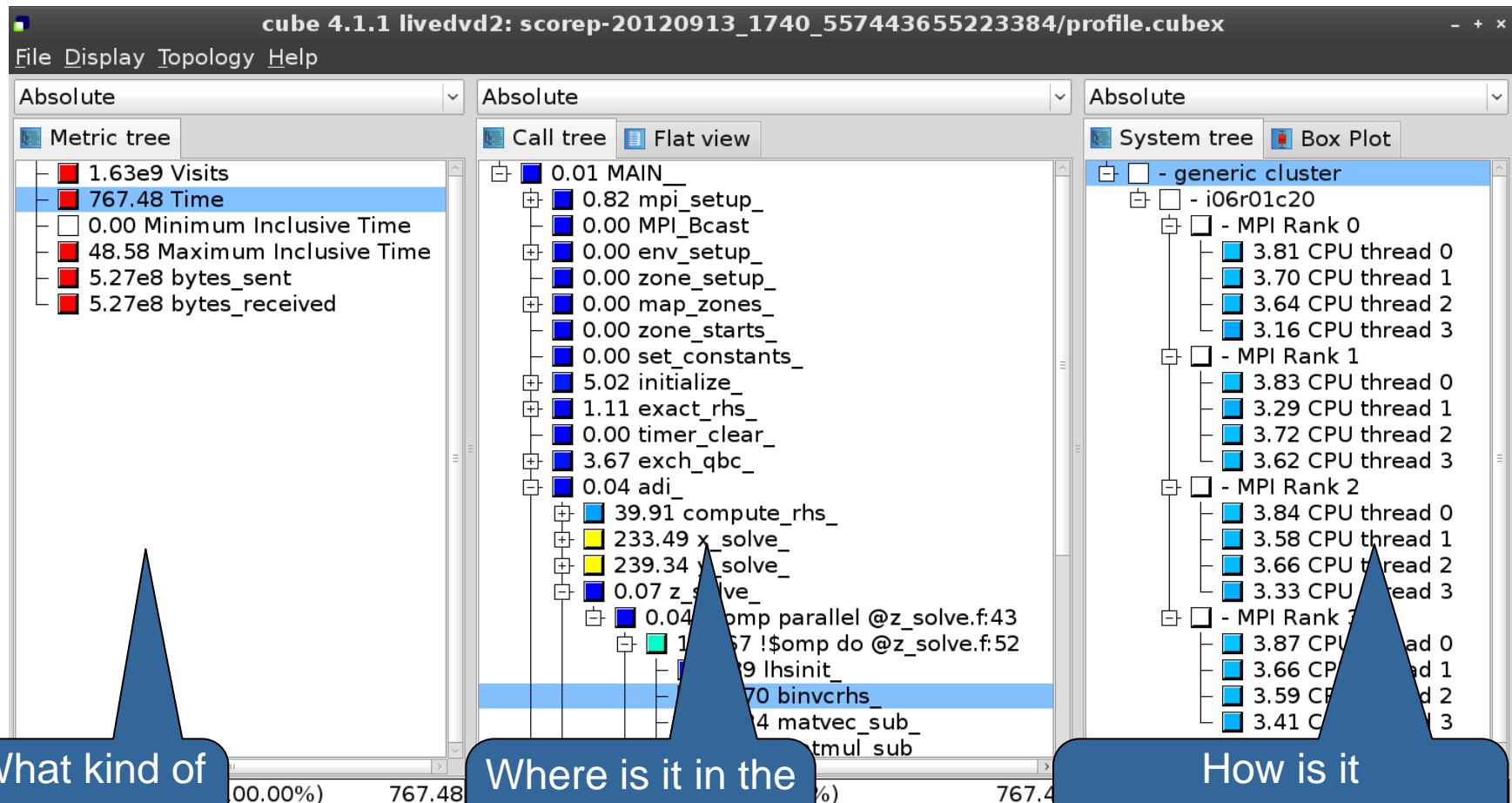


Three coupled tree browsers

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

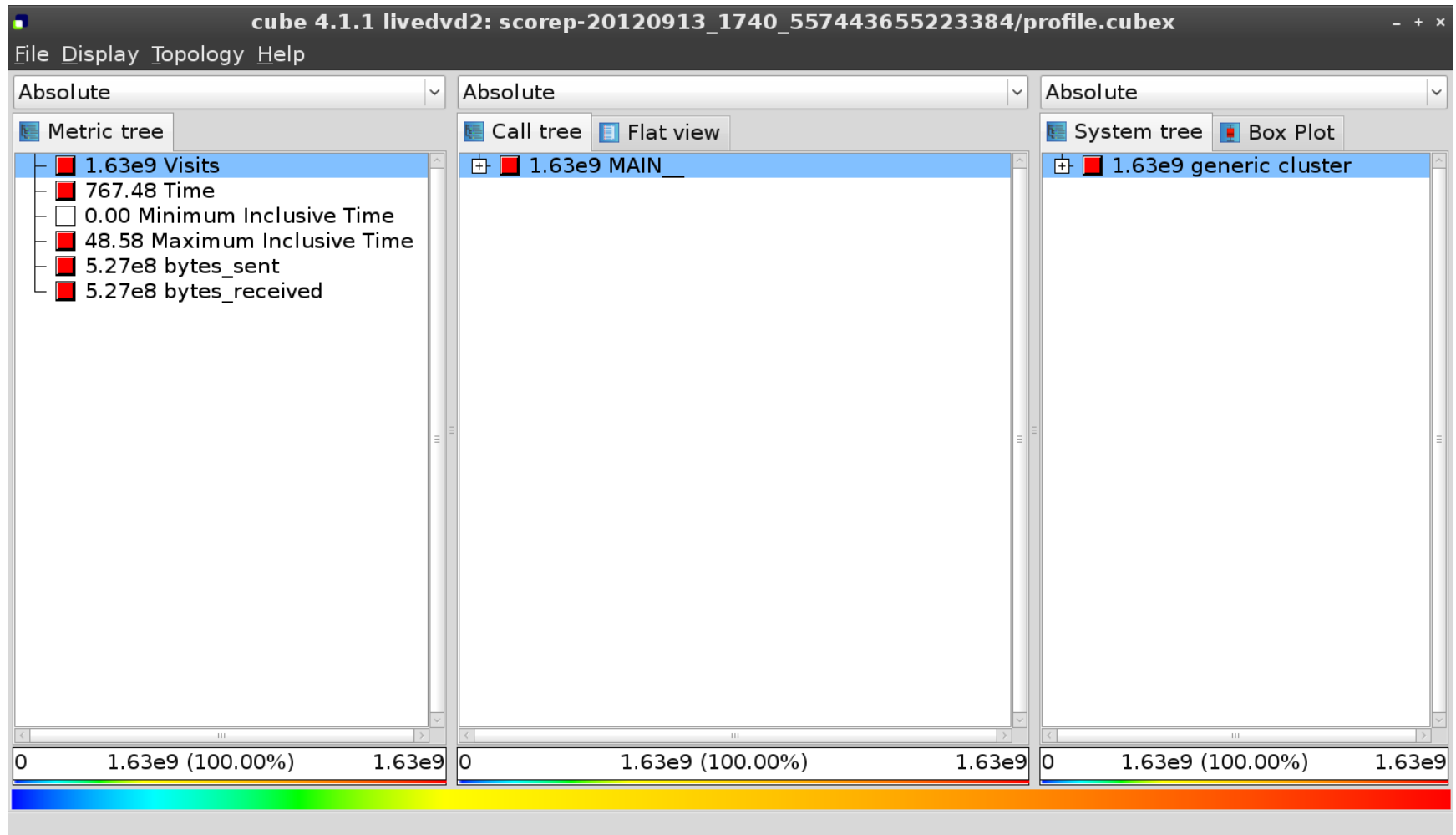


What kind of performance metric?

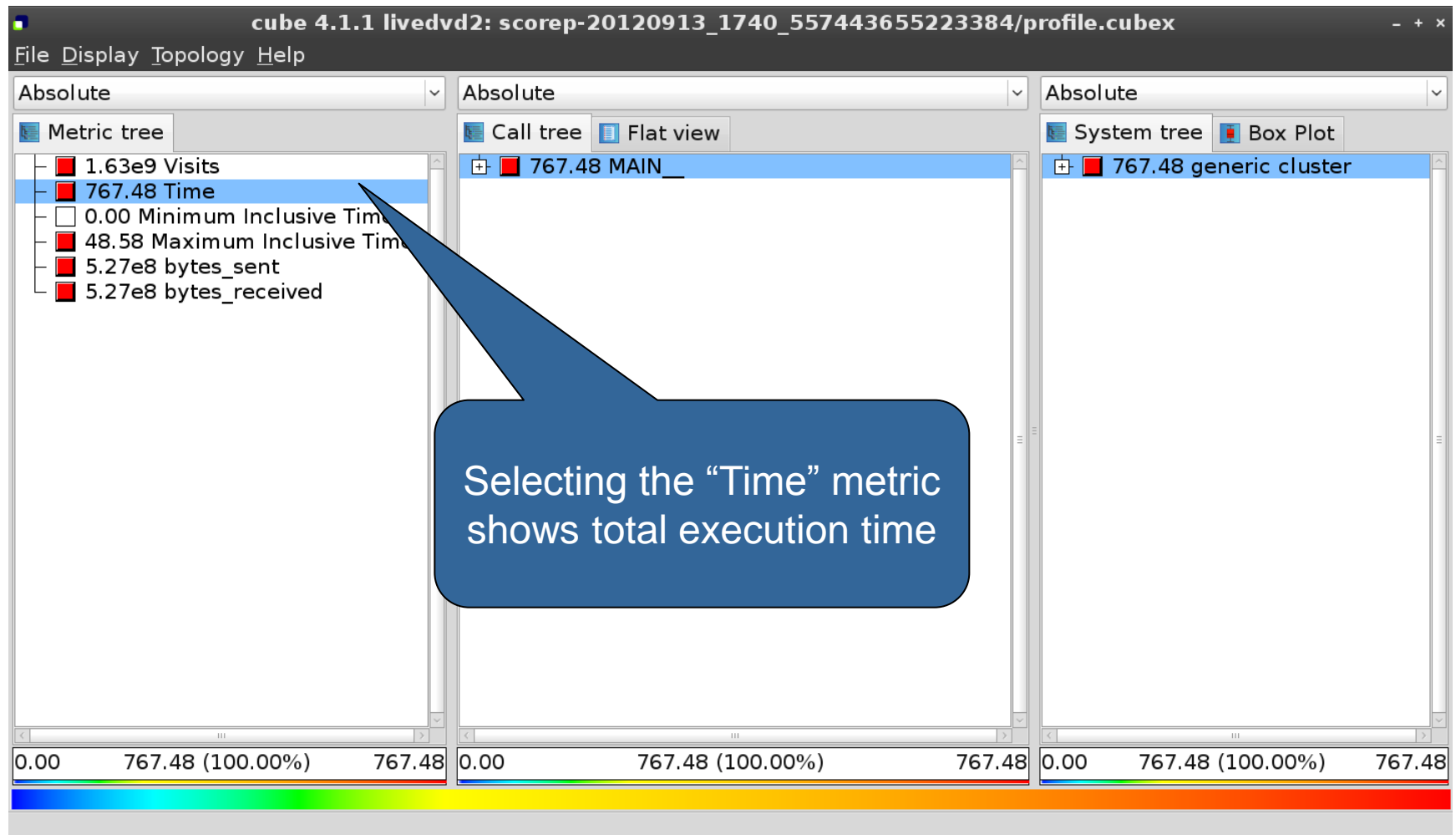
Where is it in the source code? In what context?

How is it distributed across the processes/threads?

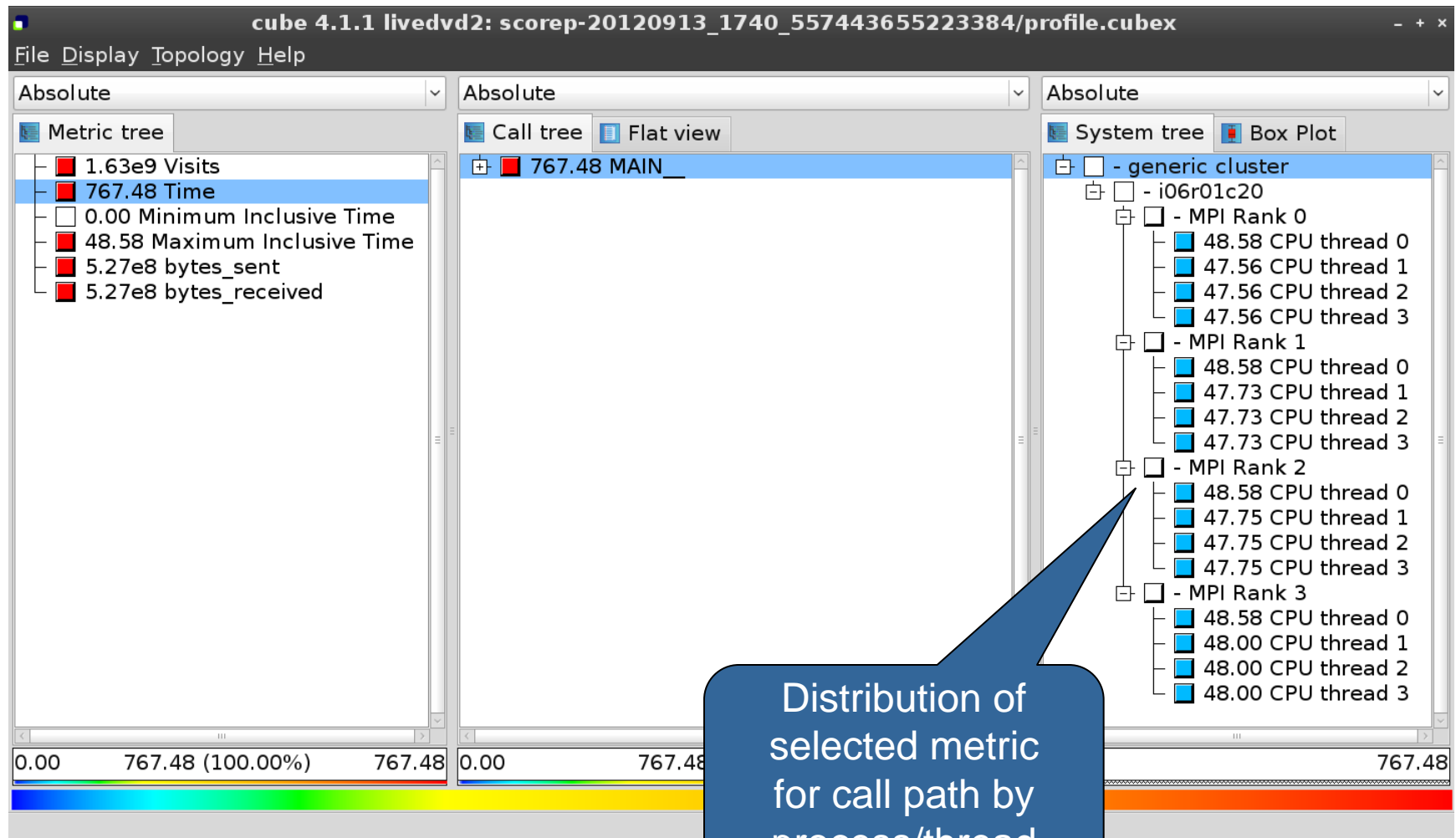
Analysis Report Exploration (Opening View)



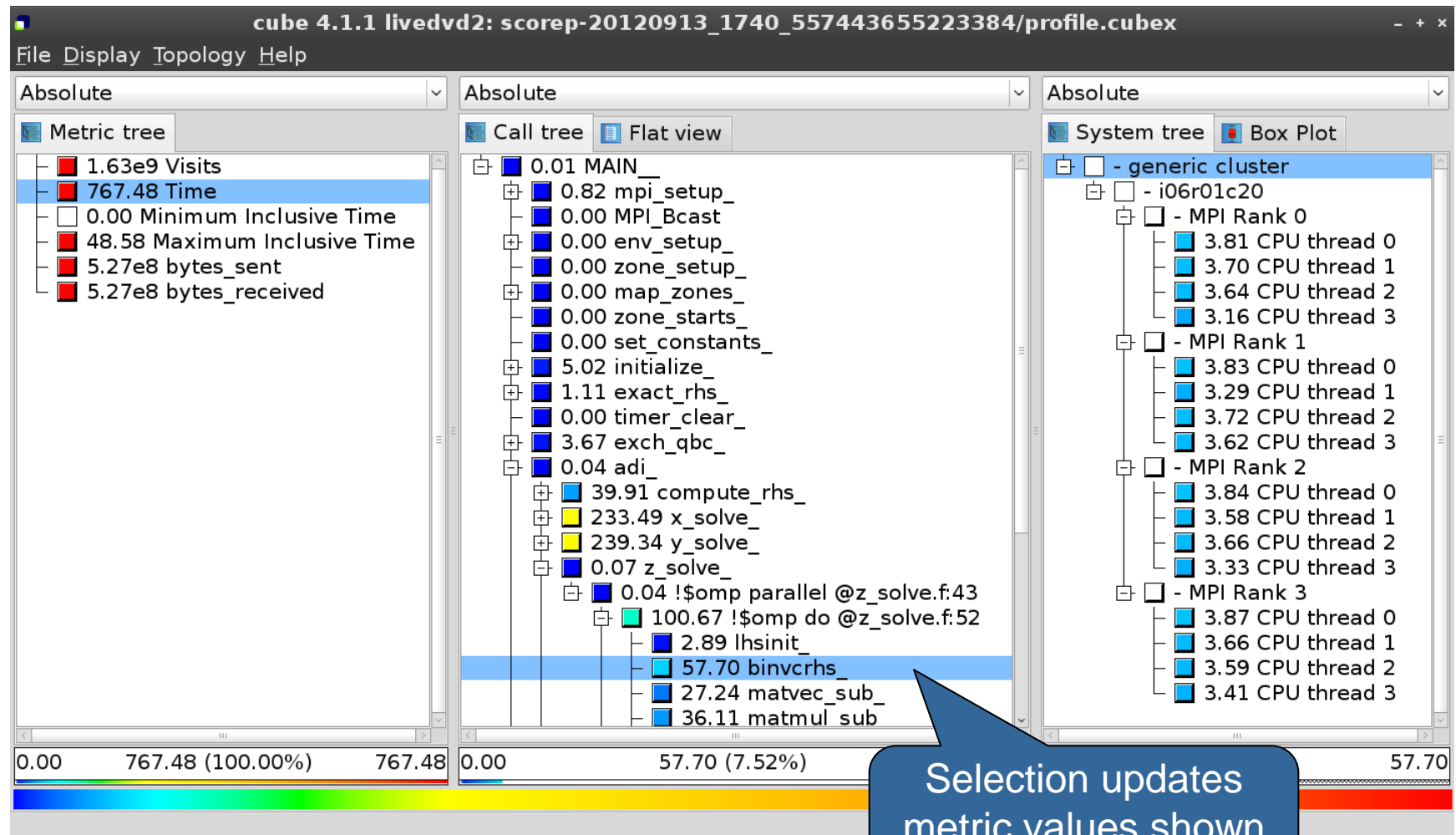
Metric Selection



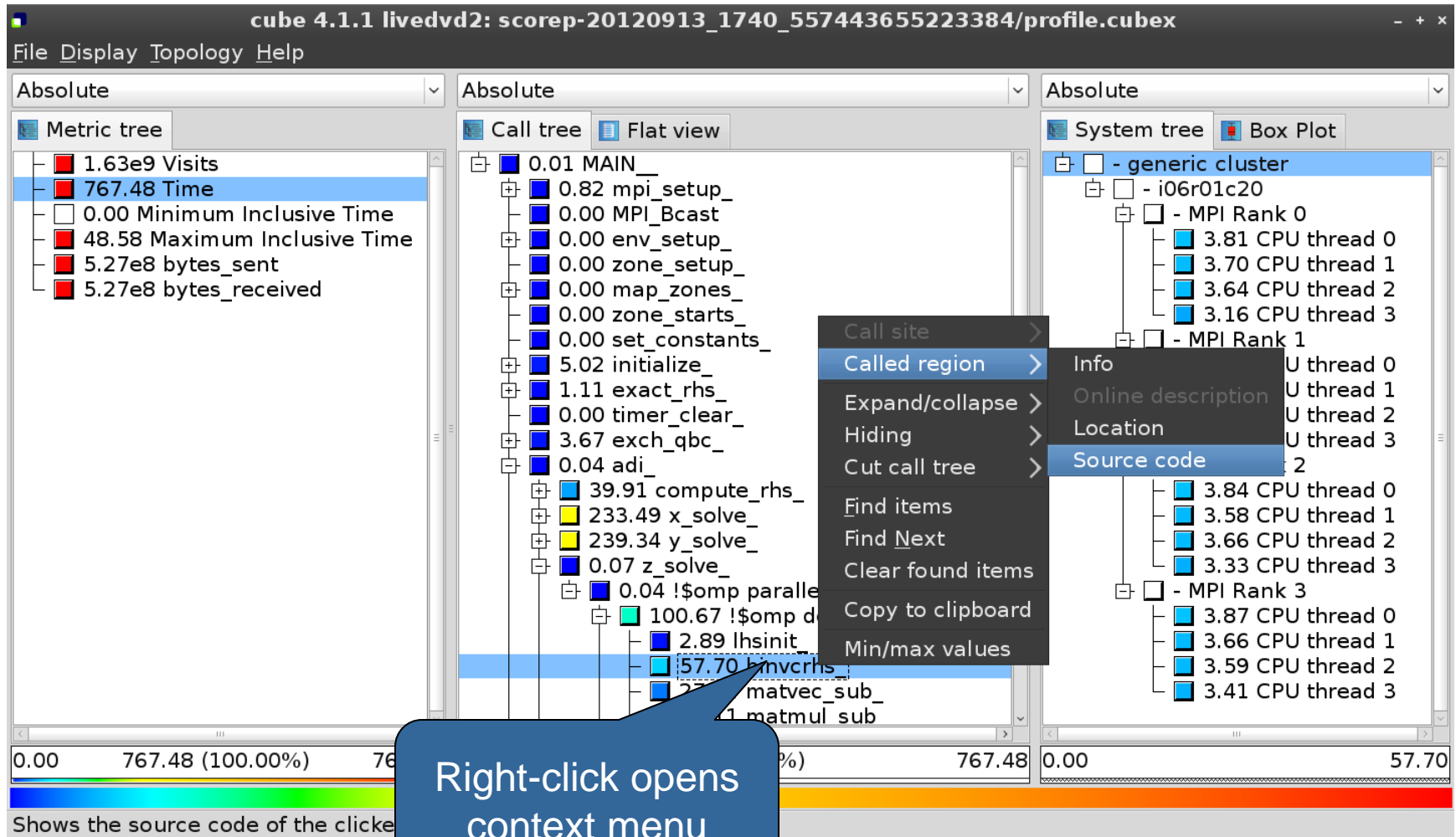
Expanding the System Tree



Selecting a Call Path



Source-code View via Context Menu



The screenshot shows the 'cube 4.1.1' application window with the title 'cube 4.1.1 livedvd2: scorep-20120913_1740_557443655223384/profile.cubex'. The interface is divided into three main panels:

- Metric tree (Left):** Displays various performance metrics. The 'Time' metric is highlighted with a value of 767.48.
- Call tree (Middle):** Shows a hierarchical view of the program's execution. The 'MAIN' function is expanded, showing sub-functions like 'mpi_setup_', 'MPI_Bcast', 'env_setup_', 'zone_setup_', 'map_zones_', 'zone_starts_', 'set_constants_', 'initialize_', 'exact_rhs_', 'timer_clear_', 'exch_qbc_', 'adi_', 'compute_rhs_', 'x_solve_', 'y_solve_', 'z_solve_', and '!\$omp parallel'. The '!\$omp parallel' block is expanded, showing '!\$omp d' and '!\$omp b'. The '!\$omp b' block is further expanded, showing '!\$omp b' and '!\$omp b'.
- System tree (Right):** Displays the system configuration, including the 'generic cluster', 'i06r01c20', and 'MPI Rank 0'.

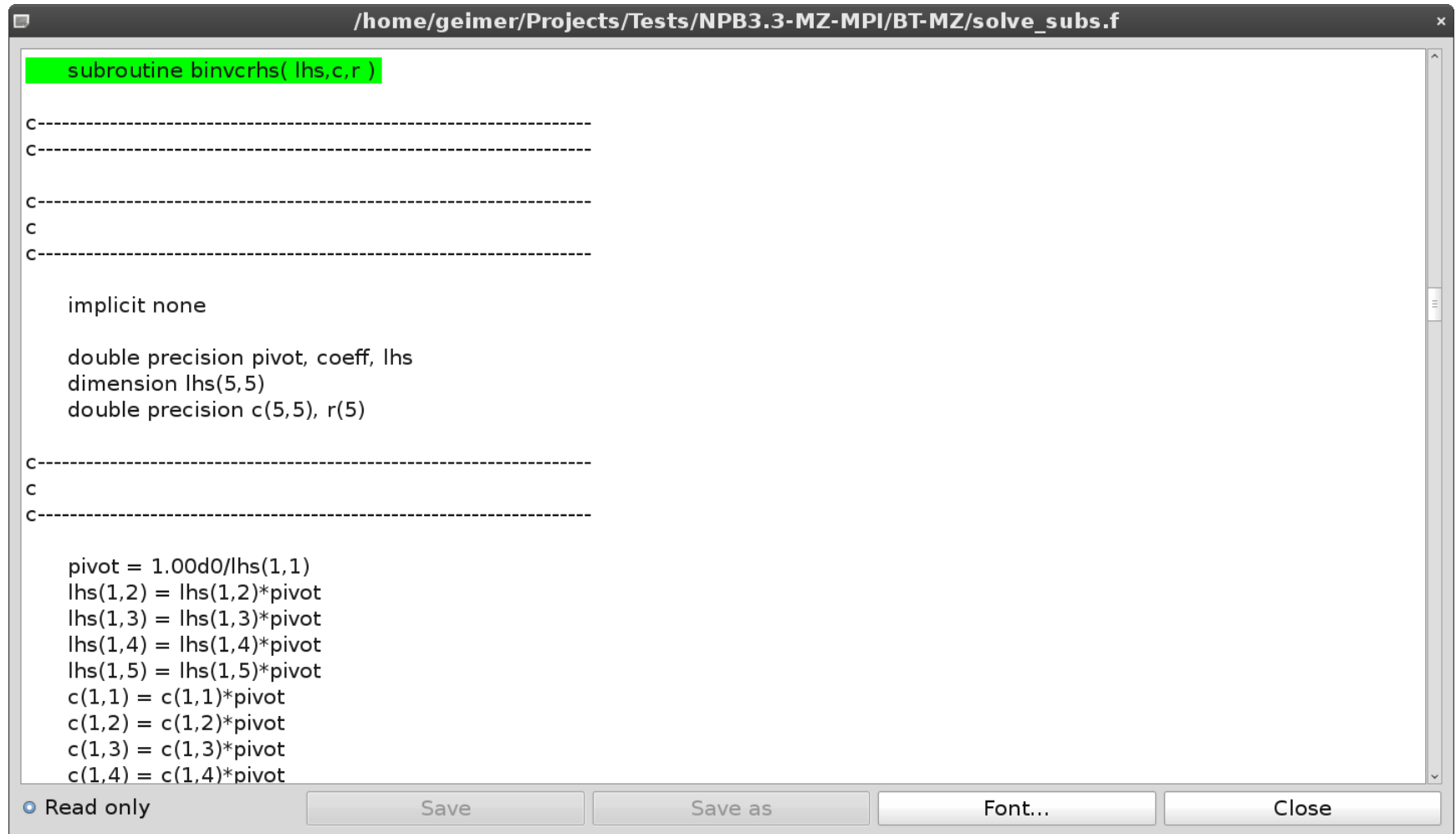
A context menu is open over the '!\$omp b' block in the Call tree. The menu options are:

- 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

A blue callout box with a speech bubble points to the context menu, containing the text: "Right-click opens context menu".

At the bottom of the window, there is a status bar with a progress indicator and a color bar. The progress indicator shows '0.00 767.48 (100.00%) 767.48'. The color bar shows a gradient from blue to red.

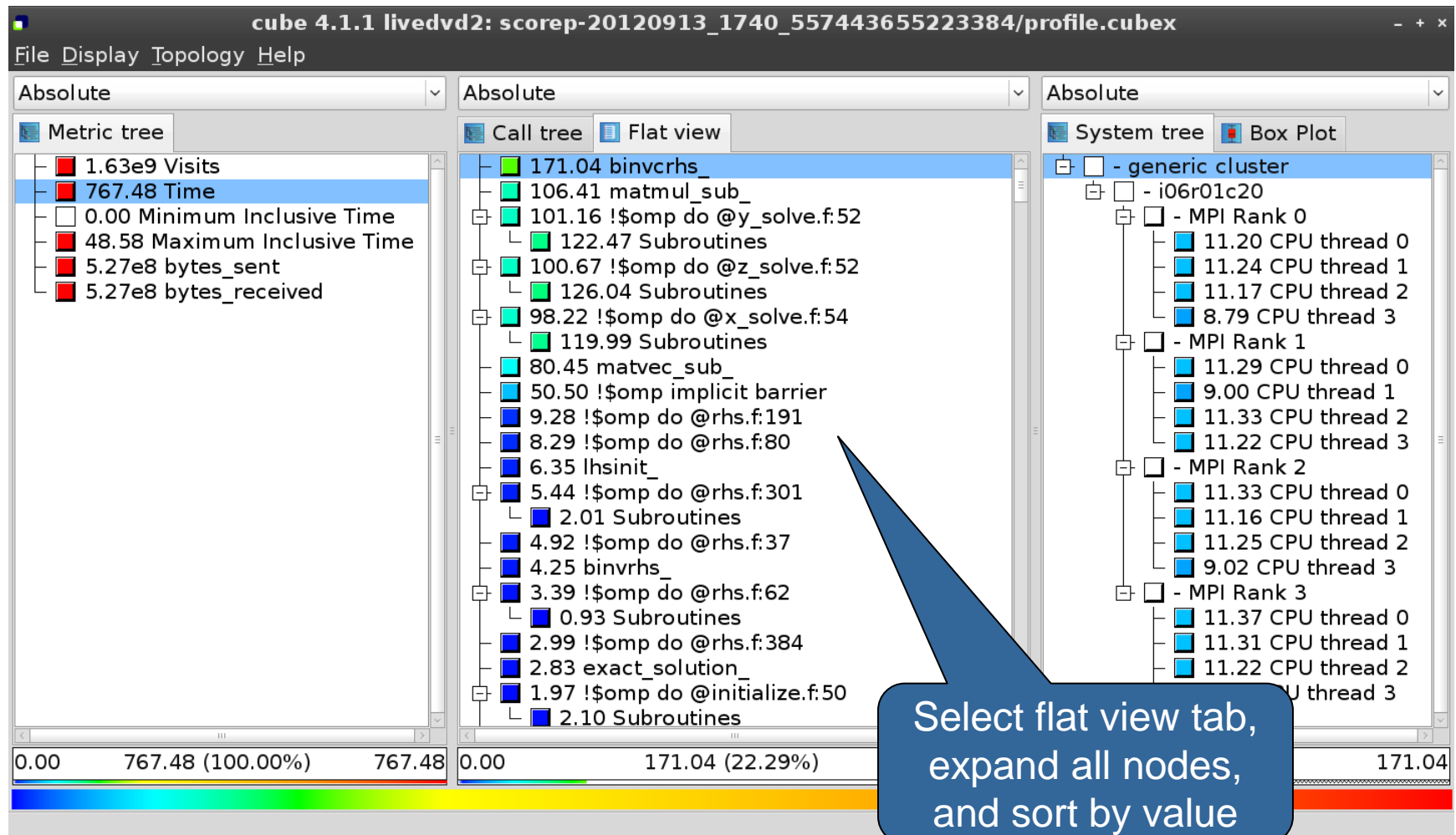
Source-code View



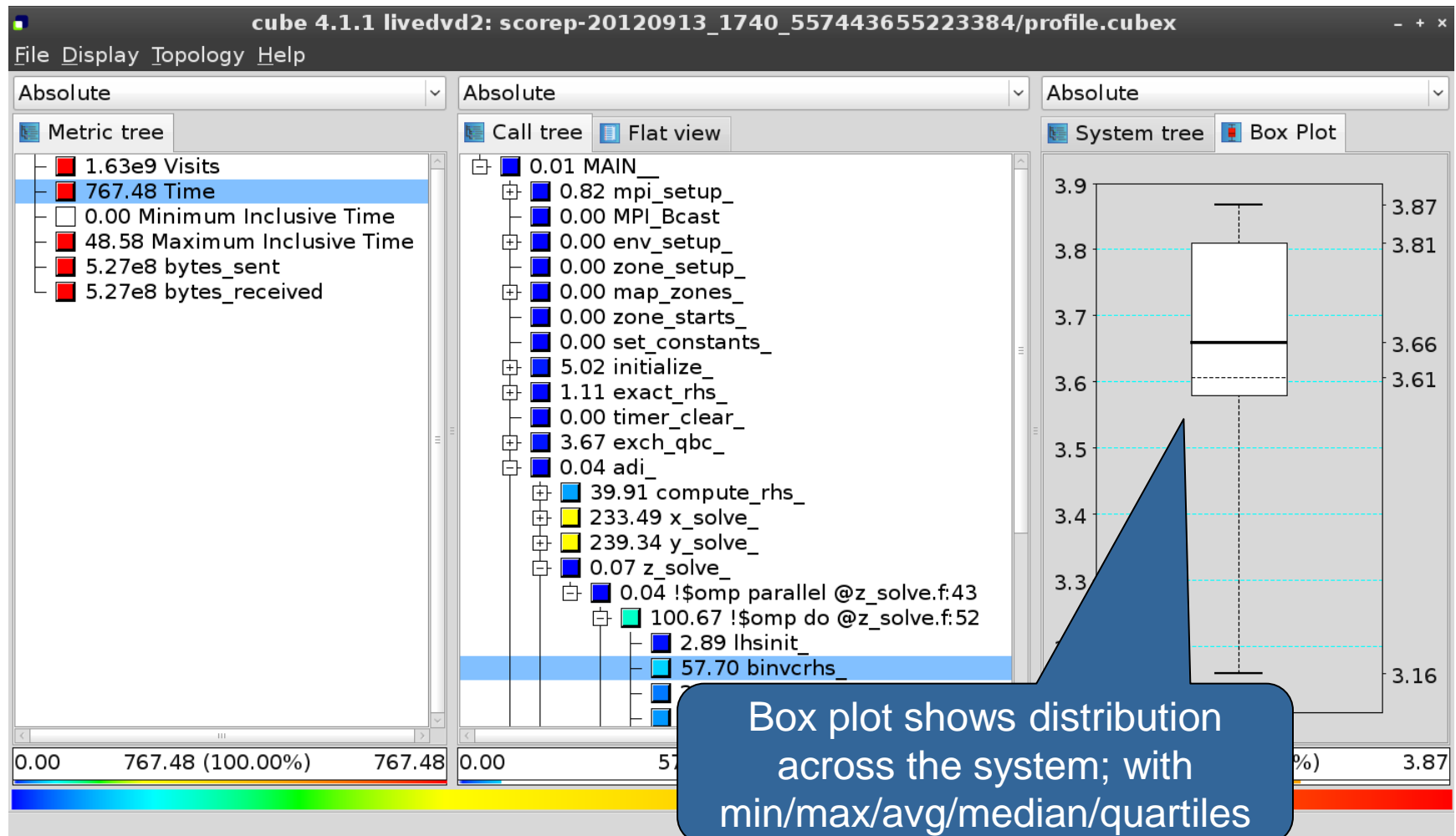
```
subroutine binvcrhs( lhs,c,r )  
  
c-----  
c-----  
  
c-----  
c  
c-----  
  
implicit none  
  
double precision pivot, coeff, lhs  
dimension lhs(5,5)  
double precision c(5,5), r(5)  
  
c-----  
c  
c-----  
  
pivot = 1.00d0/lhs(1,1)  
lhs(1,2) = lhs(1,2)*pivot  
lhs(1,3) = lhs(1,3)*pivot  
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

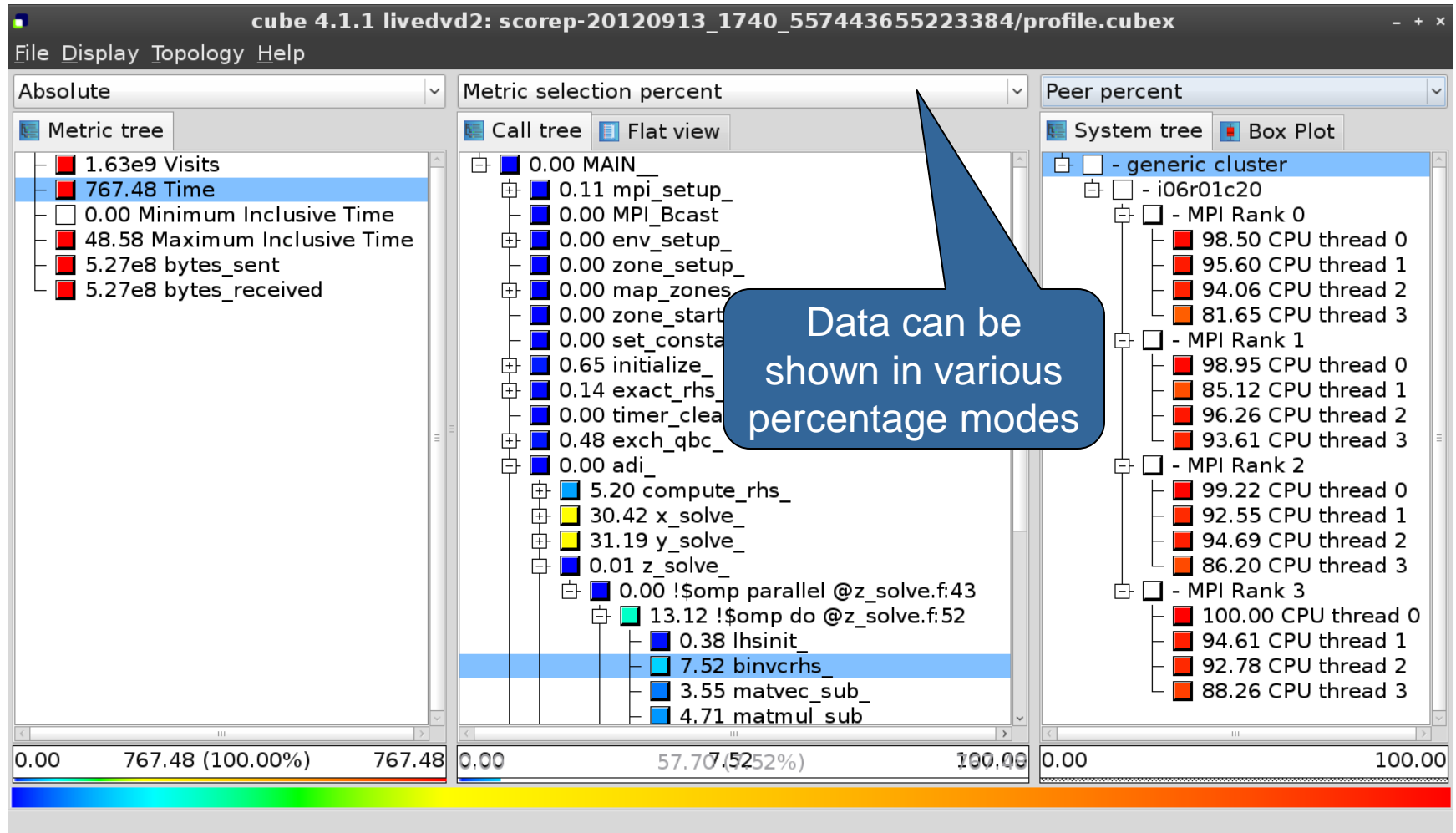
Flat Profile View



Box Plot View



Alternative Display Modes



Important Display Modes

Absolute

- Absolute value shown in seconds/bytes/counts

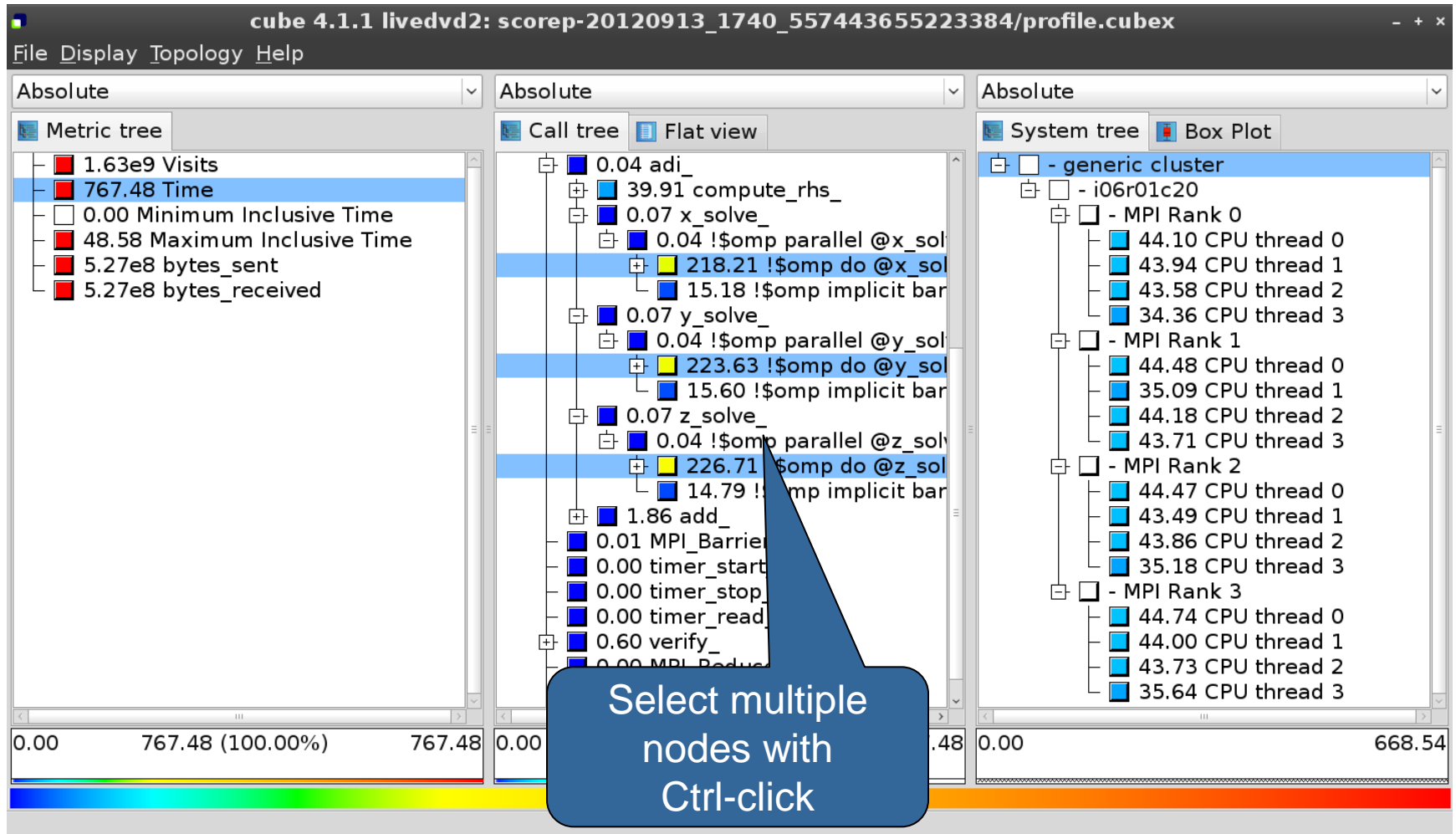
Selection percent

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

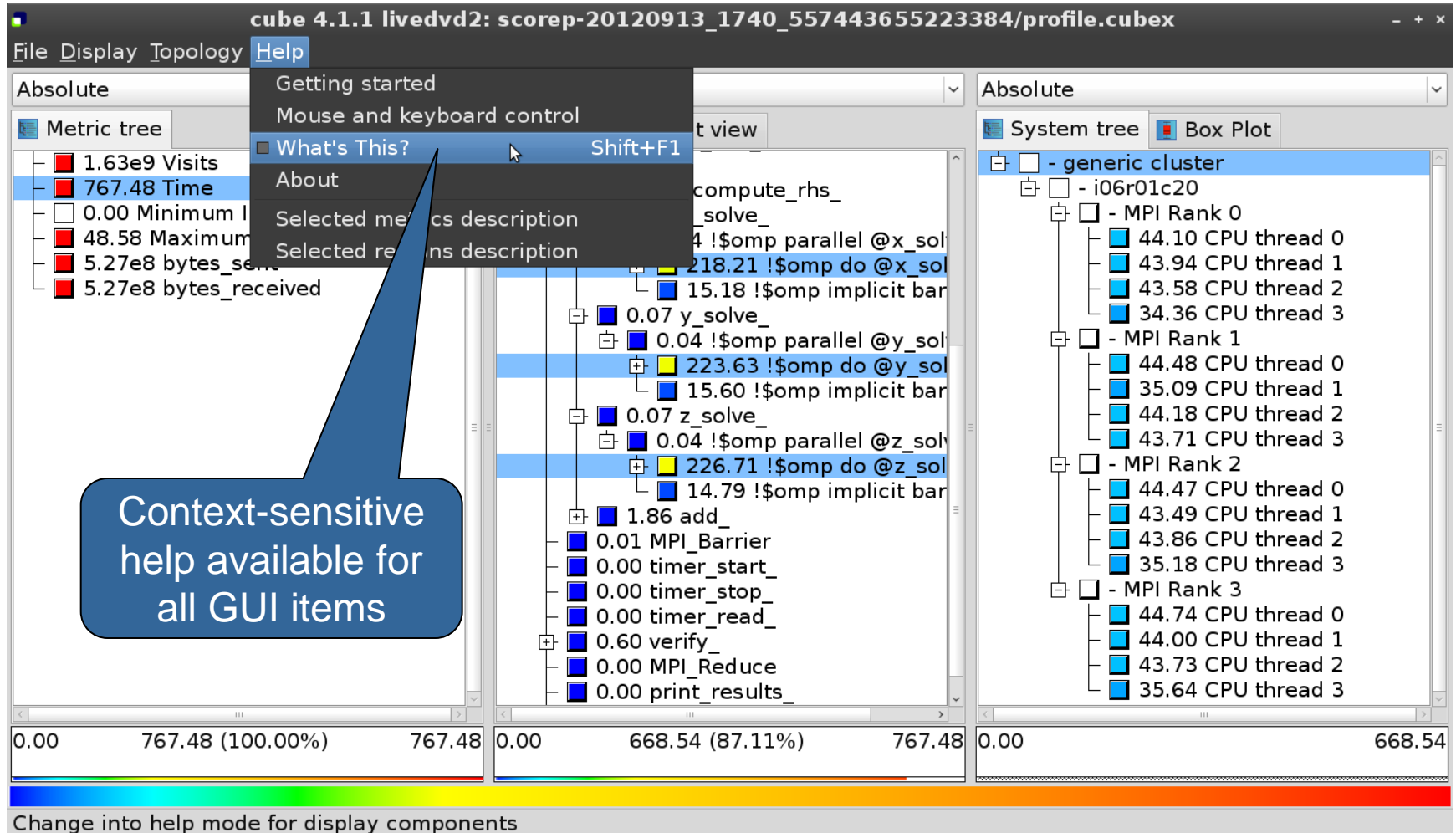
Peer percent (system tree only)

- Value shown as percentage relative to the maximum peer value

Multiple Selection



Context-sensitive Help



The screenshot shows the 'cube 4.1.1' application window with the title bar 'cube 4.1.1 livedvd2: scorep-20120913_1740_557443655223384/profile.cubex'. The 'Help' menu is open, showing options like 'Getting started', 'Mouse and keyboard control', 'What's This? (Shift+F1)', 'About', 'Selected metrics description', and 'Selected regions description'. A blue callout bubble points to the 'What's This?' option, stating: 'Context-sensitive help available for all GUI items'.

The main window displays three panels:

- Metric tree (Left):** Shows a list of metrics with values and units. The 'Time' metric is highlighted with a red square icon.

Metric	Value	Unit
1.63e9 Visits	1.63e9	Visits
767.48 Time	767.48	
0.00 Minimum	0.00	
48.58 Maximum	48.58	
5.27e8 bytes_sent	5.27e8	bytes_sent
5.27e8 bytes_received	5.27e8	bytes_received
- System tree (Right):** Shows a hierarchical view of the system components. The 'generic cluster' is expanded, showing MPI Ranks 0 through 3, each with four CPU threads.

MPI Rank	CPU Thread	Value
- i06r01c20	- MPI Rank 0	44.10 CPU thread 0
	43.94 CPU thread 1	
	43.58 CPU thread 2	
	34.36 CPU thread 3	
- MPI Rank 1	- MPI Rank 1	44.48 CPU thread 0
	35.09 CPU thread 1	
	44.18 CPU thread 2	
	43.71 CPU thread 3	
- MPI Rank 2	- MPI Rank 2	44.47 CPU thread 0
	43.49 CPU thread 1	
	43.86 CPU thread 2	
	35.18 CPU thread 3	
- MPI Rank 3	- MPI Rank 3	44.74 CPU thread 0
	44.00 CPU thread 1	
	43.73 CPU thread 2	
	35.64 CPU thread 3	
- Bottom Panel:** Displays a progress bar and a status bar. The status bar shows '0.00 767.48 (100.00%) 767.48' and '0.00 668.54 (87.11%) 767.48'. The progress bar is a horizontal bar with a color gradient from blue to red.

At the bottom of the window, a text bar reads: 'Change into help mode for display components'.

CUBE Algebra Utilities

Extracting solver sub-tree from analysis report

```
% cube_cut -r '<<ITERATION>>' scorep_bt-mz_W_4x4_sum/profile.cubex  
Writing cut.cubex... done.
```

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.

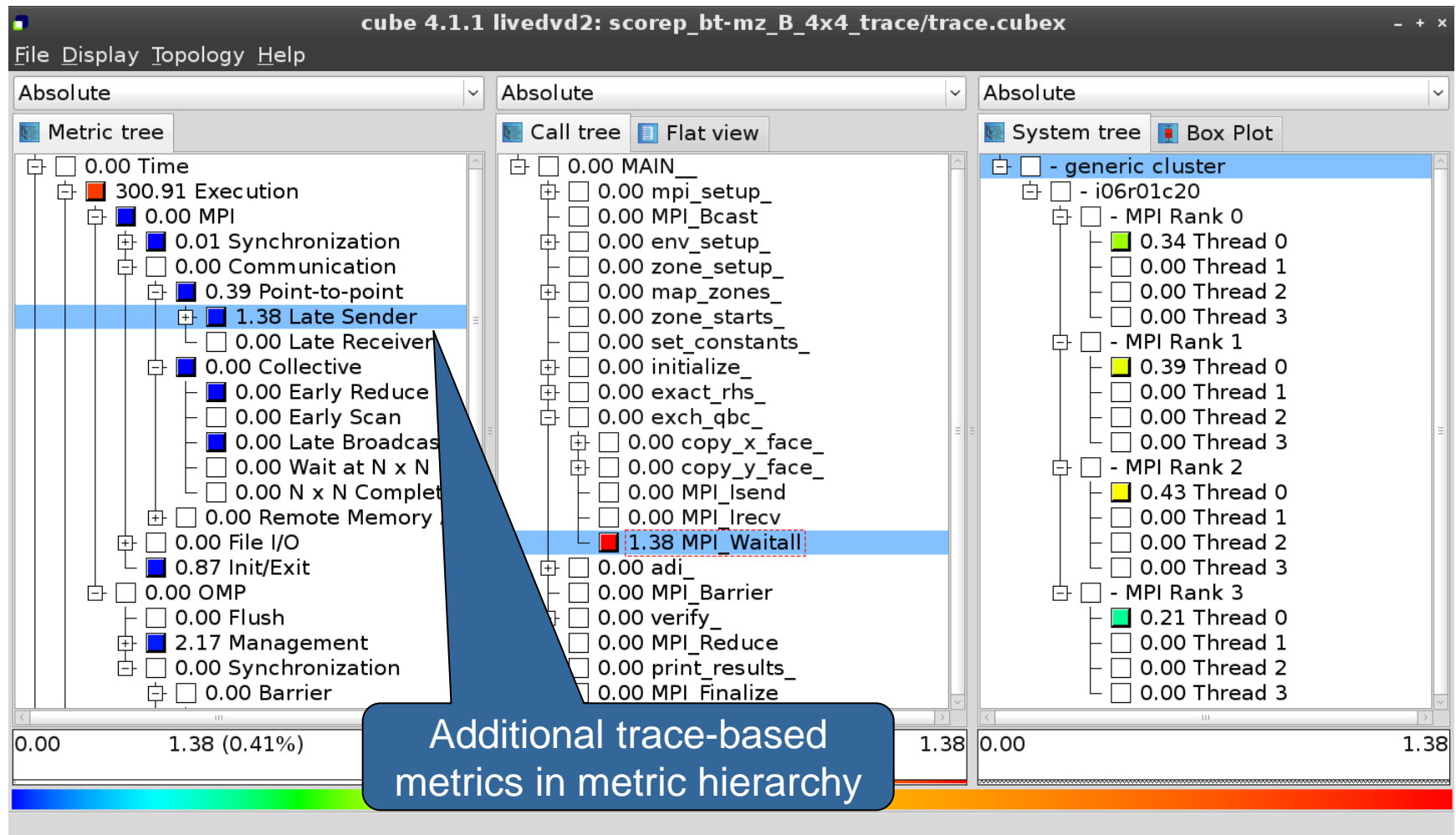
- Default output of cube_utility is a new report utility.cubex

Further utilities for report scoring & statistics

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

Analyzing a Performance Issue with Scalasca/CUBE

Post-processed Trace Analysis Report



Online Metric Description

cube 4.1.1 livedvd2: scorep_bt-mz_B_4x4_trace/trace.cubex

File Display Topology Help

Absolute Absolute Absolute

Metric tree Call tree Flat view System tree Box Plot

0.00 Time

- 300.91 Execution
 - 0.00 MPI
 - 0.01 Synchronization
 - 0.00 Communication
 - 0.39 Point-to-point
 - 1.38 Late Sender** (highlighted)
 - 0.00 Late Receiver
 - 0.00 Collective
 - 0.00 Early Receiver
 - 0.00 Early Sender
 - 0.00 Late Broadcast
 - 0.00 Wait at barrier
 - 0.00 N x N Communication
 - 0.00 Remote Memory Access
 - 0.00 File I/O
 - 0.87 Init/Exit
 - 0.00 OMP
 - 0.00 Flush
 - 2.17 Management
 - 0.00 Synchronization
 - 22.99 Barrier

0.00 1.38 (0.41%) 337.45

0.00 MAIN

- 0.00 mpi_setup
- 0.00 MPI_Bcast
- 0.00 env_setup
- 0.00 zone_setup
- 0.00 map_zones
- 0.00 zone_starts
- 0.00 zone_instants
- 0.00 zone_size
- 0.00 rhs
- 0.00 abc
- 0.00 y_x_face
- 0.00 y_y_face
- 0.00 isend
- 0.00 recv
- 0.00 initial

0.00 1.38 (100.00%) 38

- generic cluster

- i06r01c20
 - MPI Rank 0
 - 0.34 Thread 0
 - 0.00 Thread 1
 - 0.00 Thread 2
 - 0.00 Thread 3
 - MPI Rank 1
 - 0.39 Thread 0
 - 0.00 Thread 1
 - 0.00 Thread 2
 - 0.00 Thread 3
 - MPI Rank 2
 - 0.43 Thread 0
 - 0.00 Thread 1
 - 0.00 Thread 2
 - 0.00 Thread 3
 - MPI Rank 3
 - 0.21 Thread 0
 - 0.00 Thread 1
 - 0.00 Thread 2
 - 0.00 Thread 3

0.00 1.38

Shows the online description of the clicked item

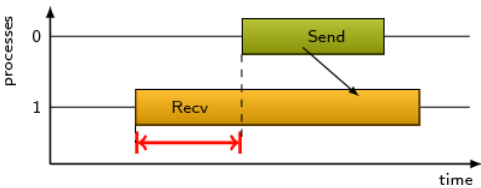
Access online metric description via context menu

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.



The diagram shows two horizontal timelines for processes 0 and 1. Process 0 has a green box labeled 'Send'. Process 1 has a yellow box labeled 'Recv'. The 'Recv' box starts at an earlier time than the 'Send' box. A red double-headed arrow on the timeline for process 1 indicates the duration from the start of the 'Recv' operation to the start of the 'Send' operation, representing the 'Late Sender Time'.

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

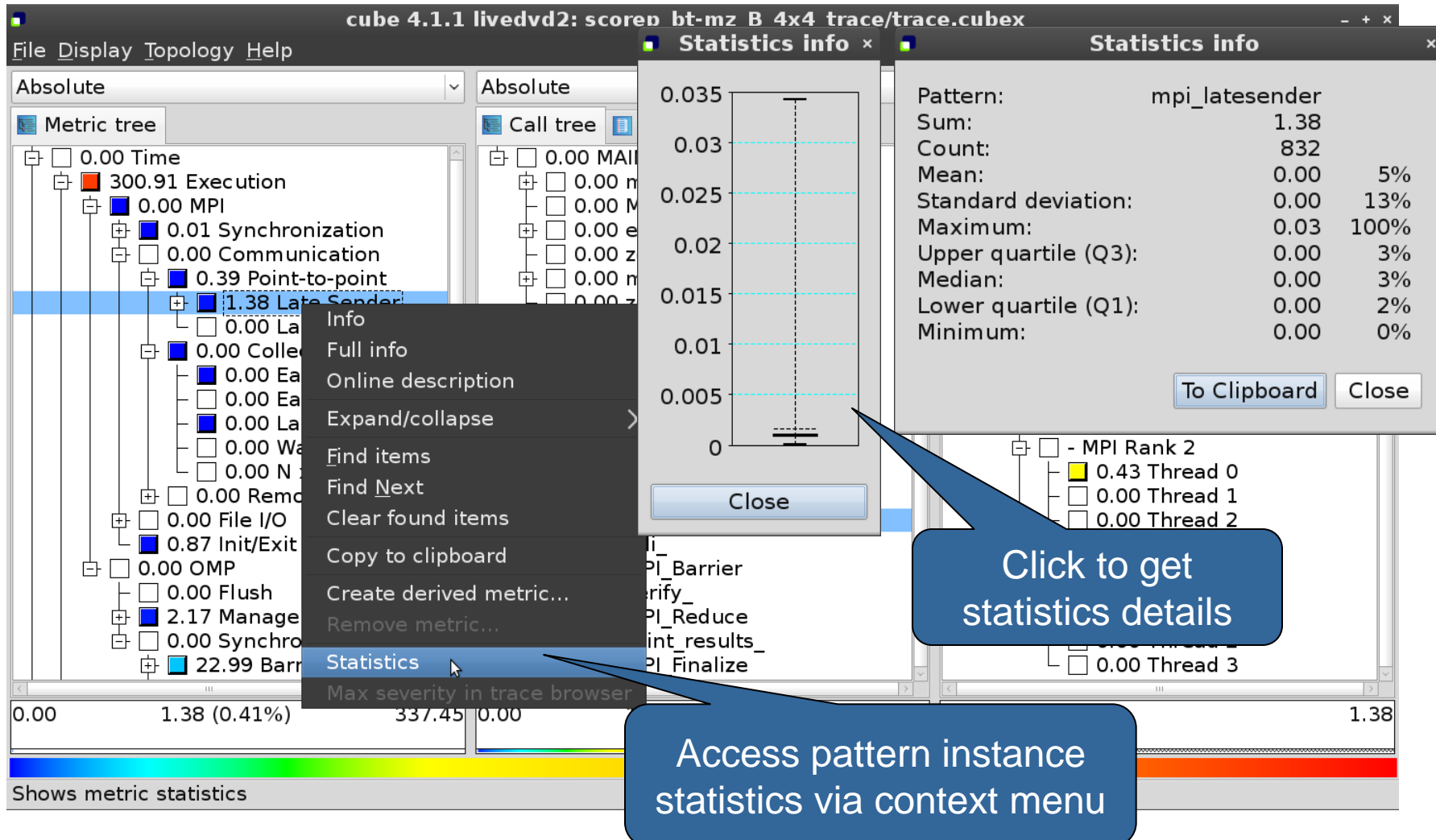
Diagnosis:
 Try to replace `MPI_Recv` with a non-blocking receive `MPI_Irecv` that can be posted earlier, proceed concurrently with computation, and complete with a wait operation after the message is expected to have been sent. Try to post sends earlier, such that they are available when receivers need 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



The screenshot displays the 'cube 4.1.1 livedvd2: scoren bt-mz B 4x4 trace/trace.cubex' application. The 'Metric tree' on the left shows a hierarchy of metrics, with '1.38 Late Sender' selected. A context menu is open over this item, listing options such as 'Info', 'Full info', 'Online description', 'Expand/collapse', 'Find items', 'Find Next', 'Clear found items', 'Copy to clipboard', 'Create derived metric...', 'Remove metric...', 'Statistics', and 'Max severity in trace browser'. The 'Statistics' option is highlighted. A 'Statistics info' dialog box is also open, showing details for the 'mpi_latesender' pattern, including Sum (1.38), Count (832), Mean (0.00), Standard deviation (0.00), Maximum (0.03), Upper quartile (Q3) (0.00), Median (0.00), Lower quartile (Q1) (0.00), and Minimum (0.00). A 'To Clipboard' button and a 'Close' button are at the bottom of the dialog. A 'Statistics info' window is also visible in the background, showing a bar chart and a 'Close' button. A blue callout box points to the 'Statistics' option in the context menu with the text 'Click to get statistics details'. Another blue callout box points to the 'Statistics' option in the context menu with the text 'Access pattern instance statistics via context menu'.

Statistics info

Pattern:	mpi_latesender
Sum:	1.38
Count:	832
Mean:	0.00 5%
Standard deviation:	0.00 13%
Maximum:	0.03 100%
Upper quartile (Q3):	0.00 3%
Median:	0.00 3%
Lower quartile (Q1):	0.00 2%
Minimum:	0.00 0%

Statistics info

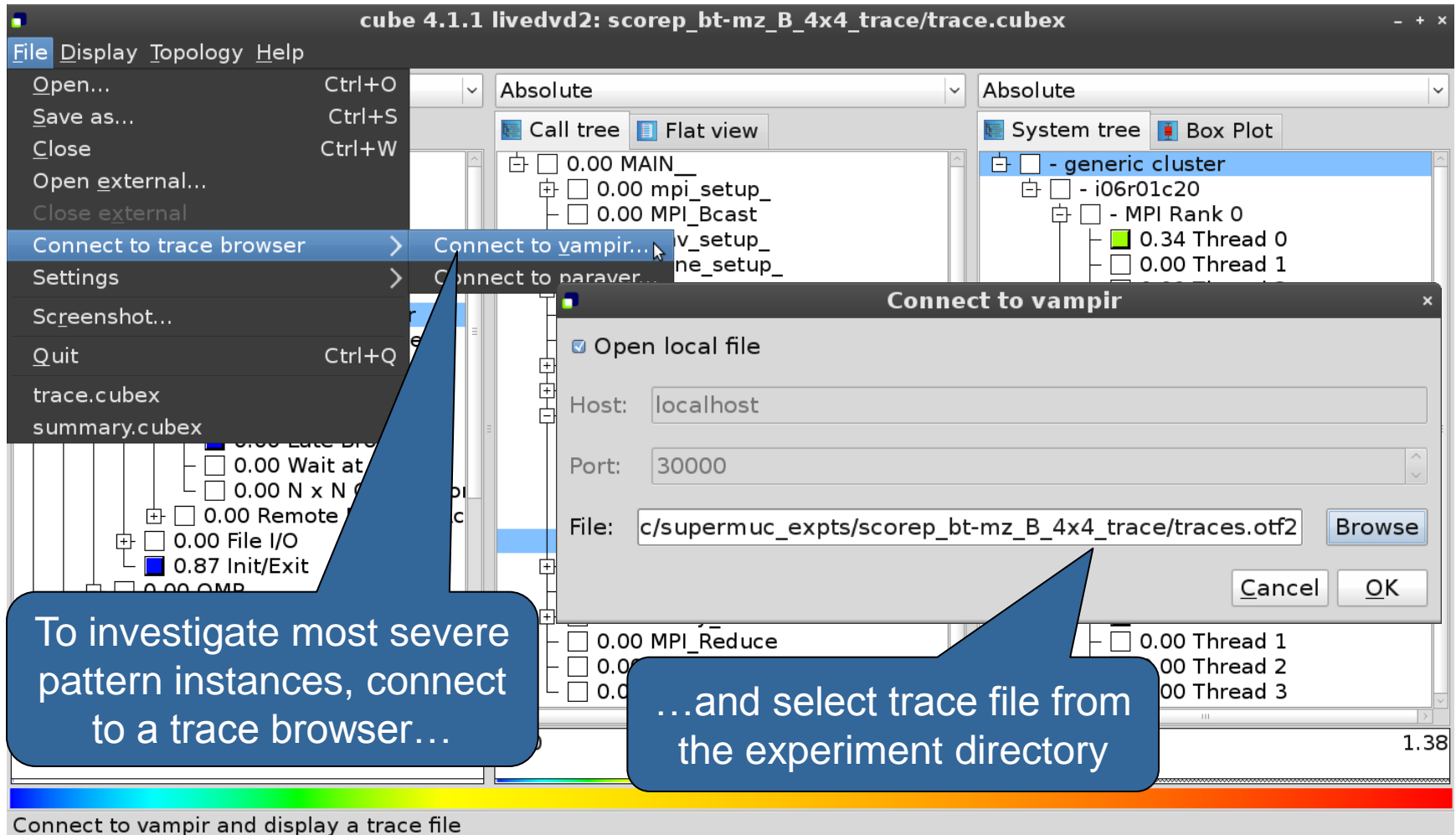
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0

Close

Click to get statistics details

Access pattern instance statistics via context menu

Connect to Vampir Trace Browser



The screenshot shows the Vampir Trace Browser application window titled "cube 4.1.1 livedvd2: scorep_bt-mz_B_4x4_trace/trace.cubex". The "File" menu is open, and the "Connect to trace browser" option is selected. A dialog box titled "Connect to vampir" is displayed, showing the "Open local file" checkbox checked, the "Host" field set to "localhost", the "Port" field set to "30000", and the "File" field set to "c:/supermuc_expts/scorep_bt-mz_B_4x4_trace/traces.otf2". The "Browse" button is highlighted. The background shows a call tree view with various process and thread entries.

Connect to vampir

☒ Open local file

Host: localhost

Port: 30000

File: c:/supermuc_expts/scorep_bt-mz_B_4x4_trace/traces.otf2 **Browse**

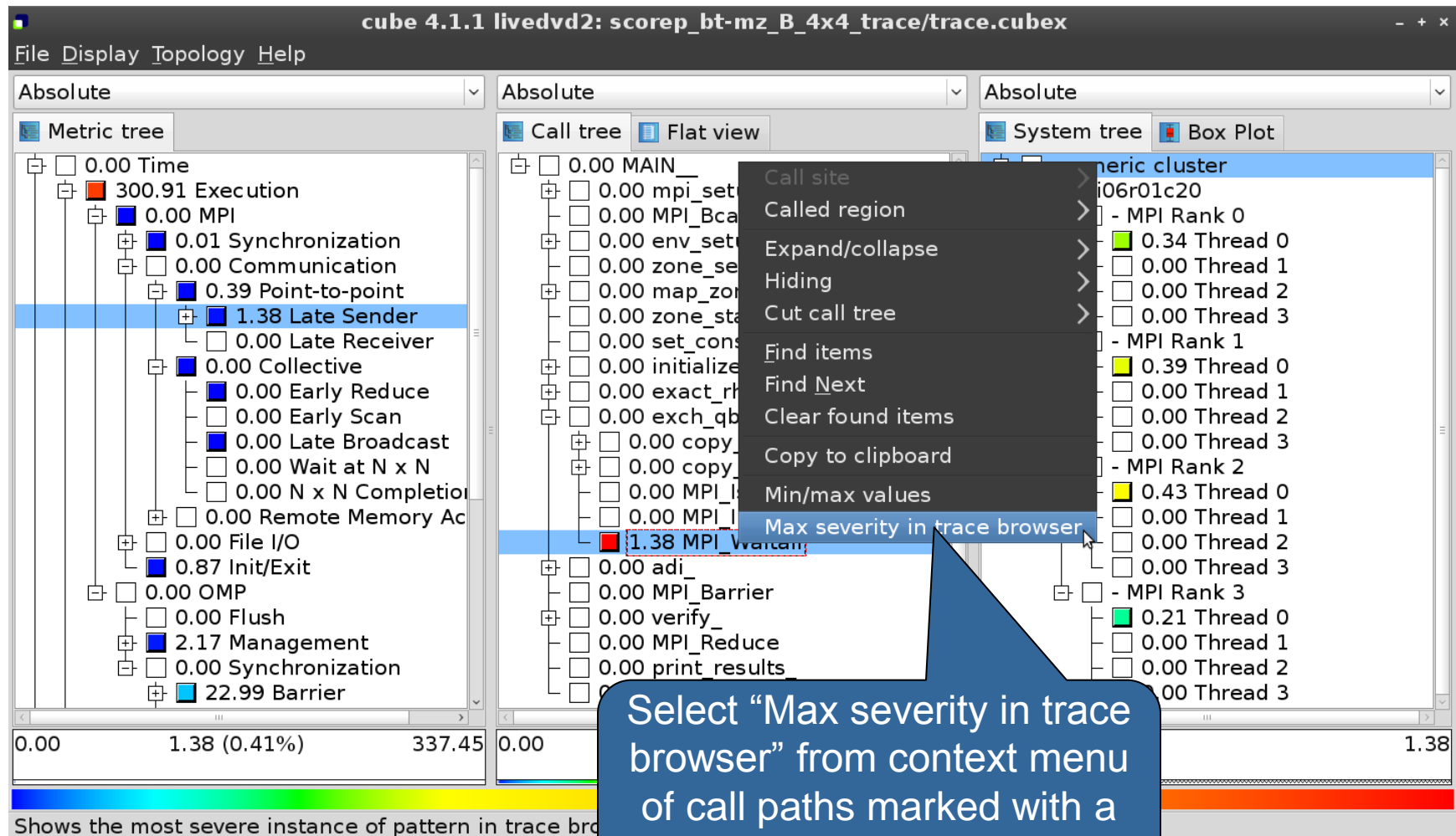
Cancel OK

Connect to vampir and display a trace file

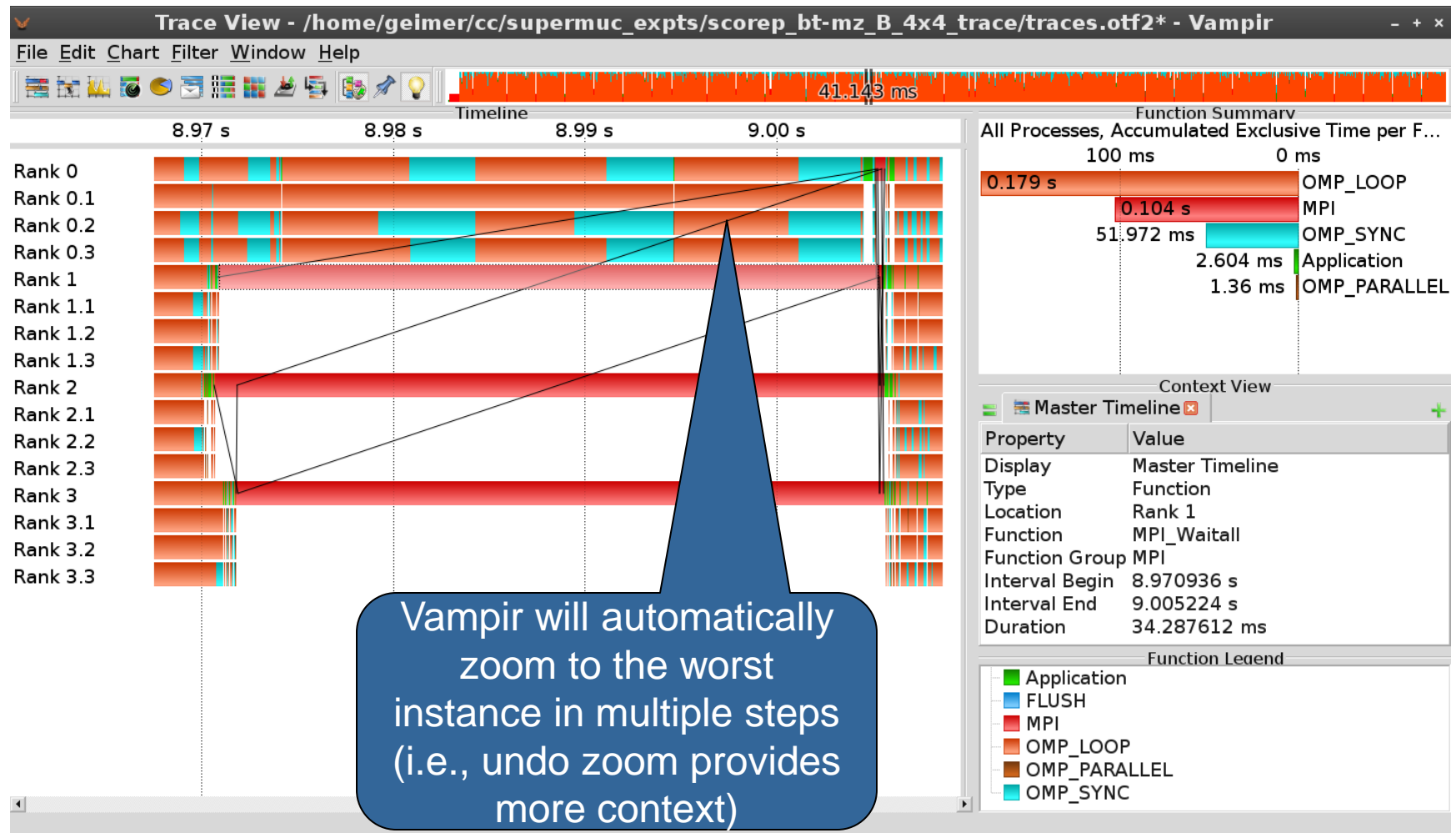
To investigate most severe pattern instances, connect to a trace browser...

...and select trace file from the experiment directory

Show Most Severe Pattern instances

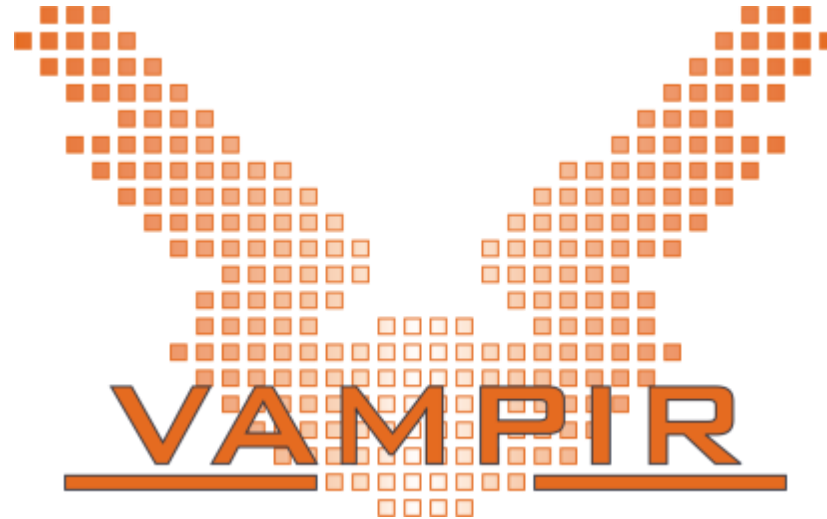


Investigate Most Severe Instance in Vampir



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



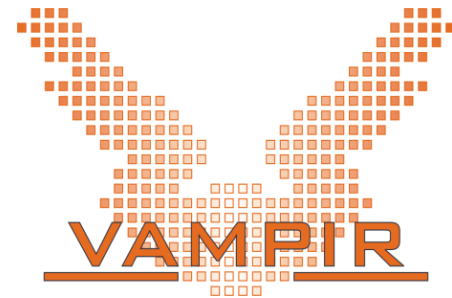
Vampir is available at <http://www.vampir.eu>,
get support via vampirsupport@zih.tu-dresden.de

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?

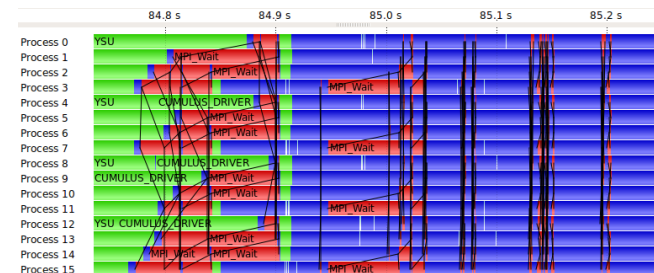
Event Trace Visualization with Vampir

Alternative and supplement to automatic analysis

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

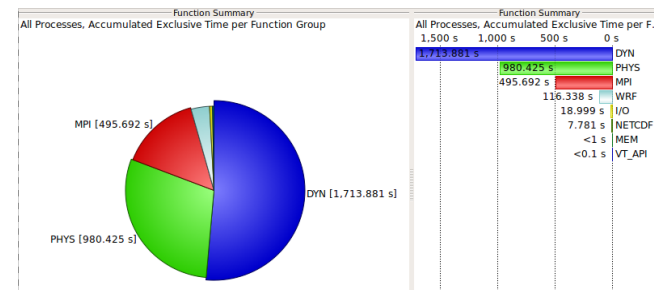
Timeline charts

- Show application activities and communication along a time axis



Summary charts

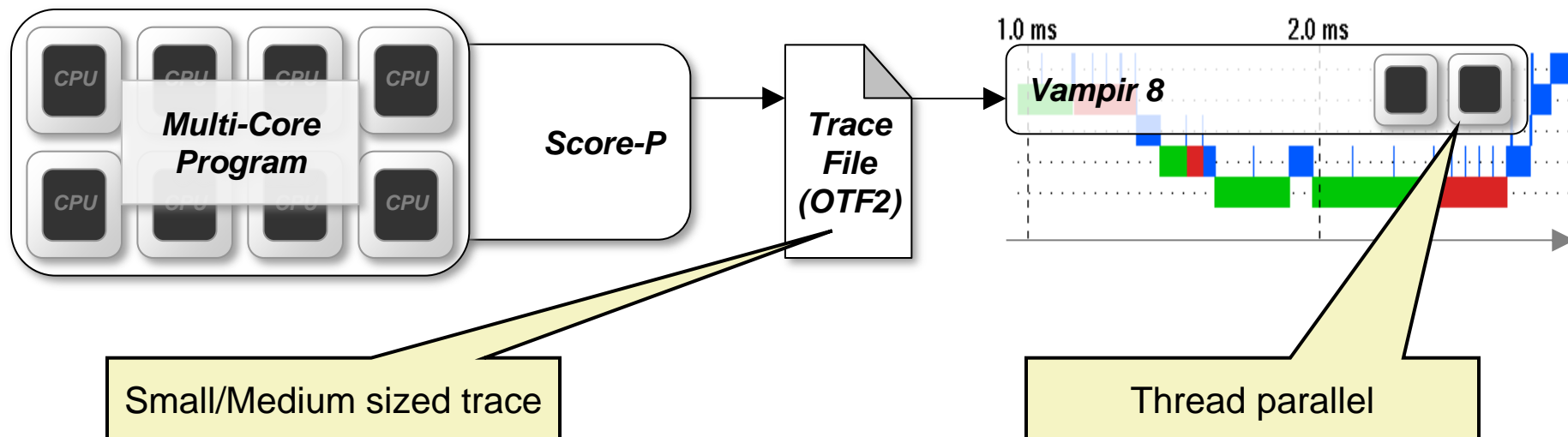
- Provide quantitative results for the currently selected time interval



Vampir – Visualization Modes (1)

Directly on front end or local machine

```
% vampir
```

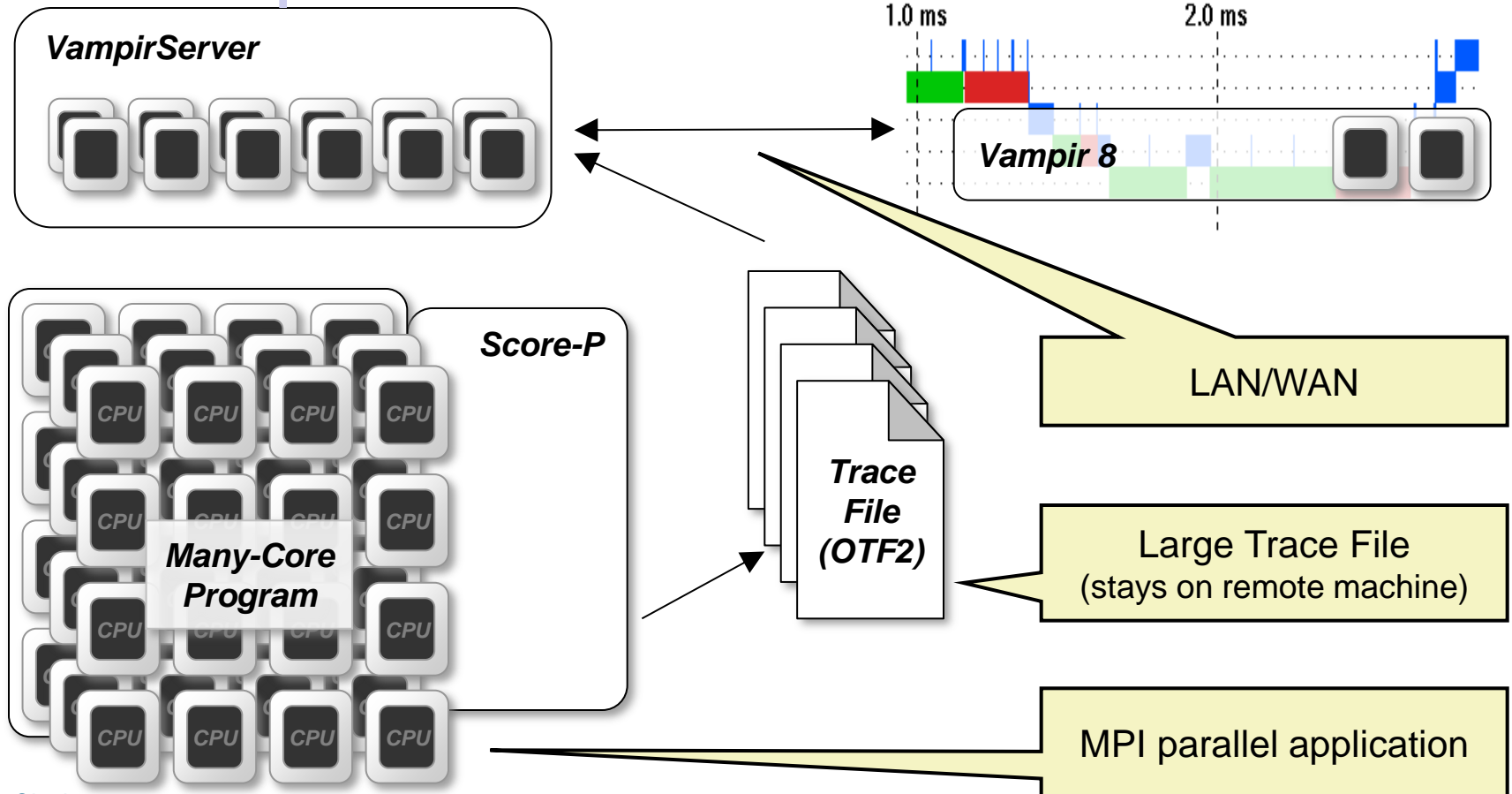


Vampir – Visualization Modes (2)

On local machine with remote VampirServer

```
% vampirserver start -n 12
```

```
% vampir
```







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:

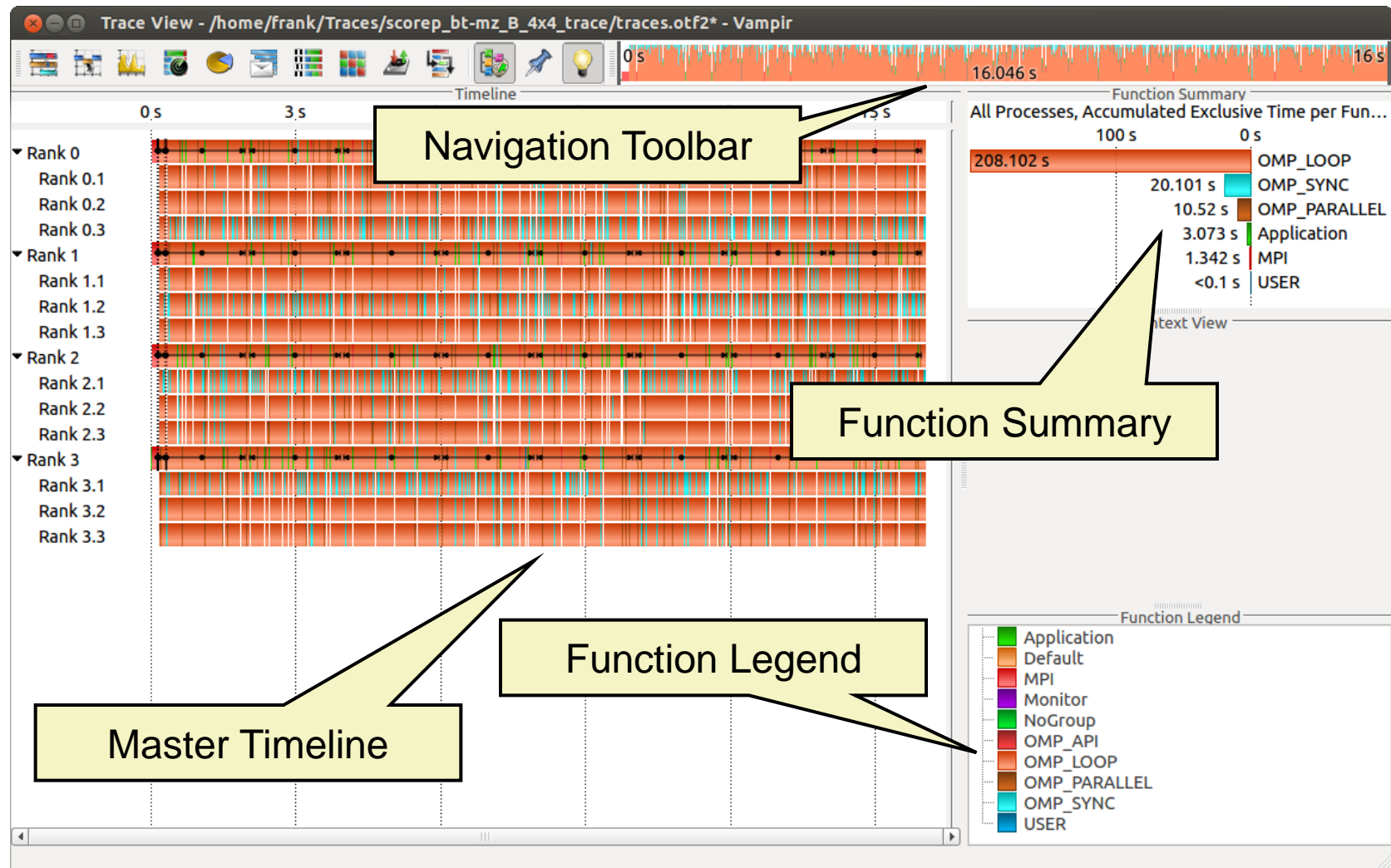
-  Master Timeline
-  Process Timeline
-  Counter Data Timeline
-  Performance Radar

Summary Charts:

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

Vampir: Example Visualization

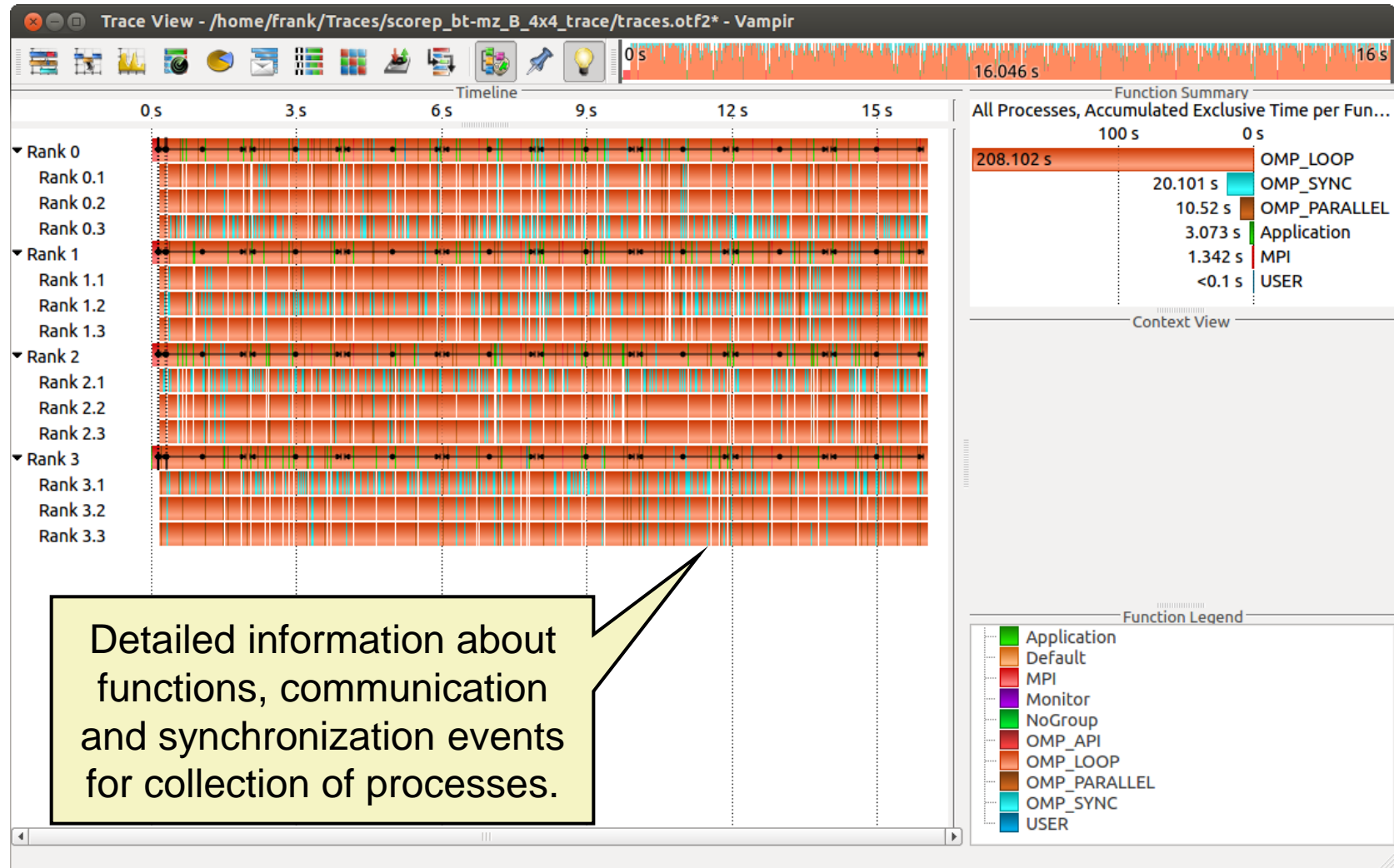
```
% vampir scorep_bt-mz_B_4x4_trace
```



Vampir: Example Visualization



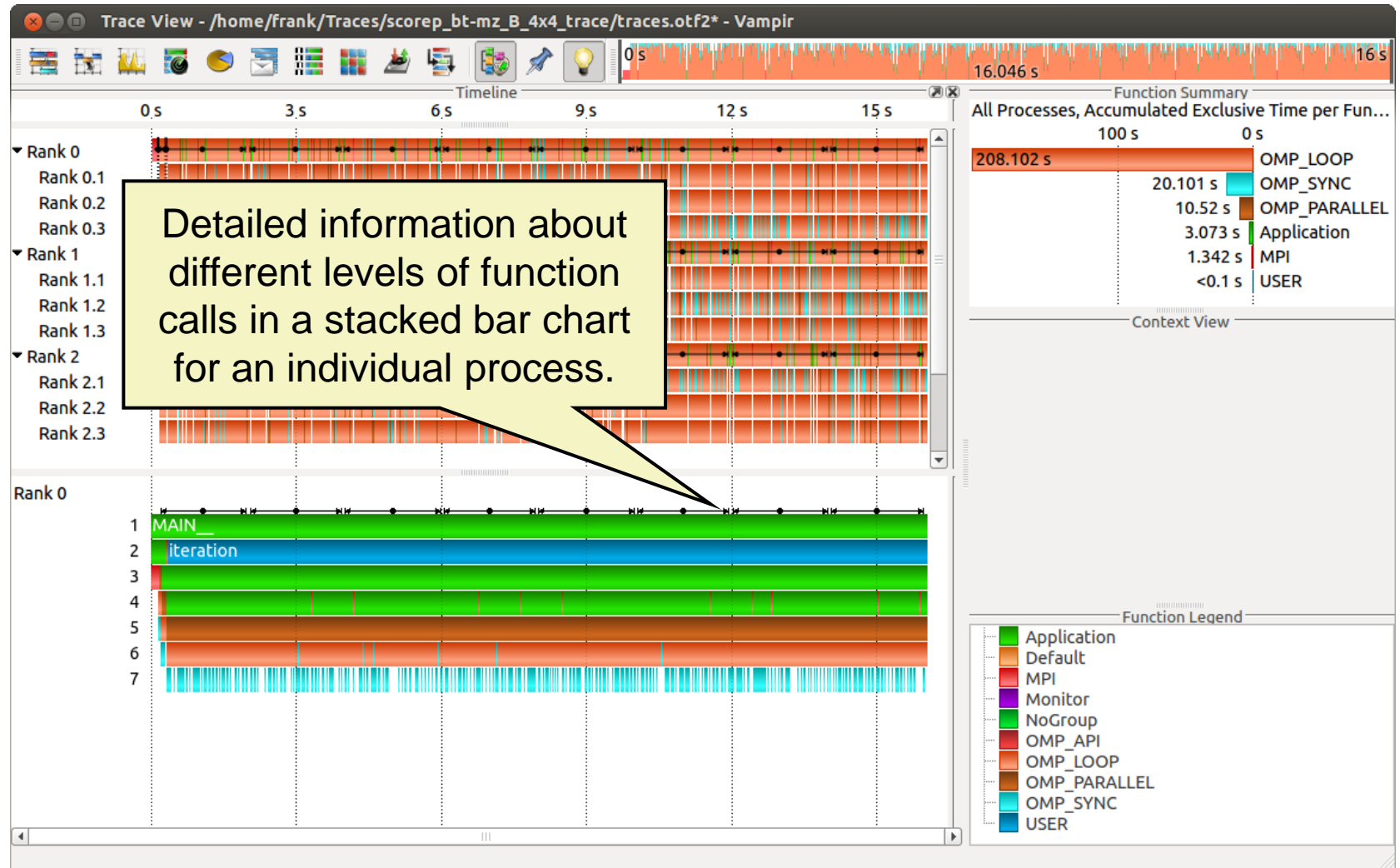
Master Timeline



Vampir: Example Visualization

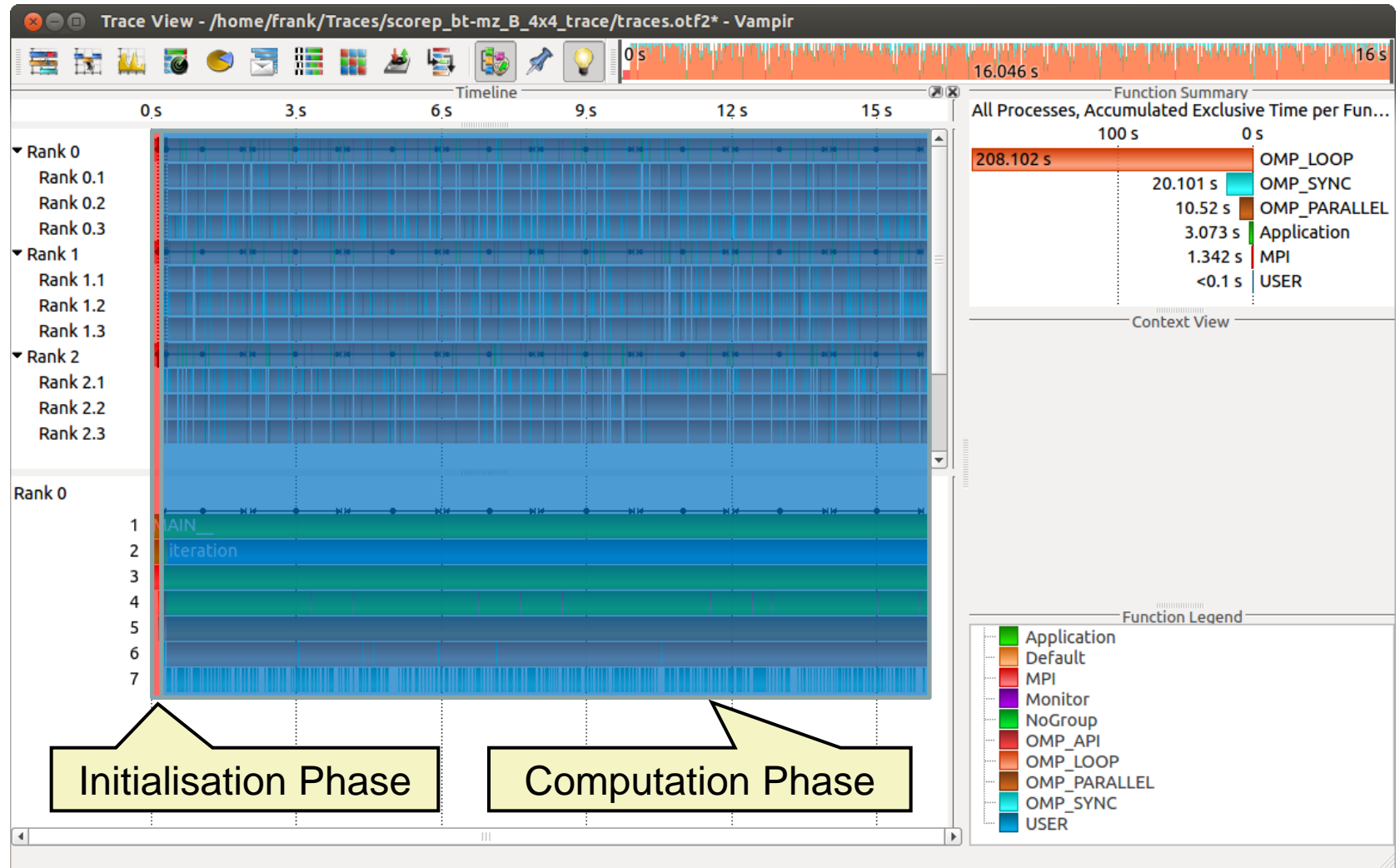


Process Timeline



Vampir: Example Visualization

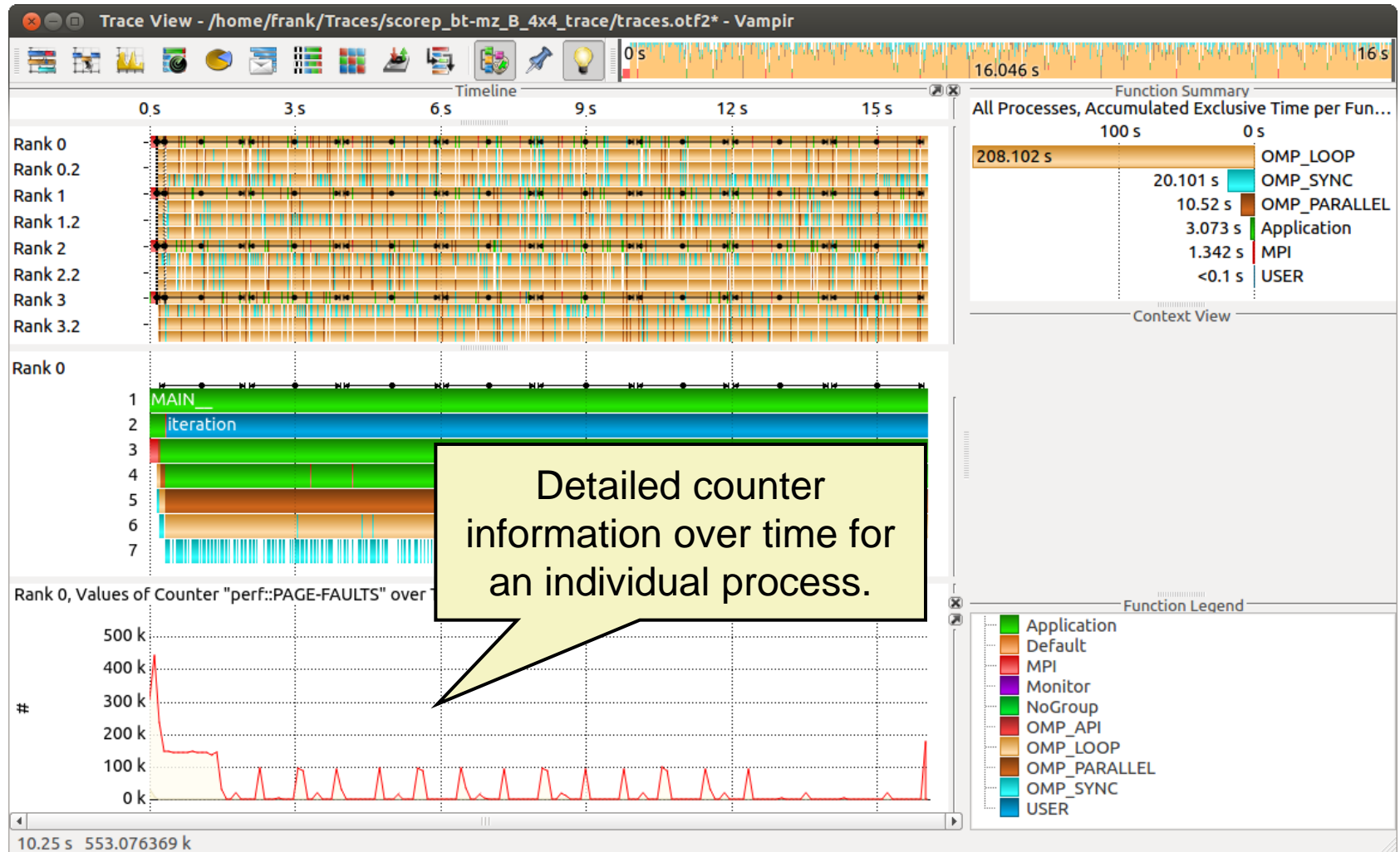
Typical program phases



Vampir: Example Visualization



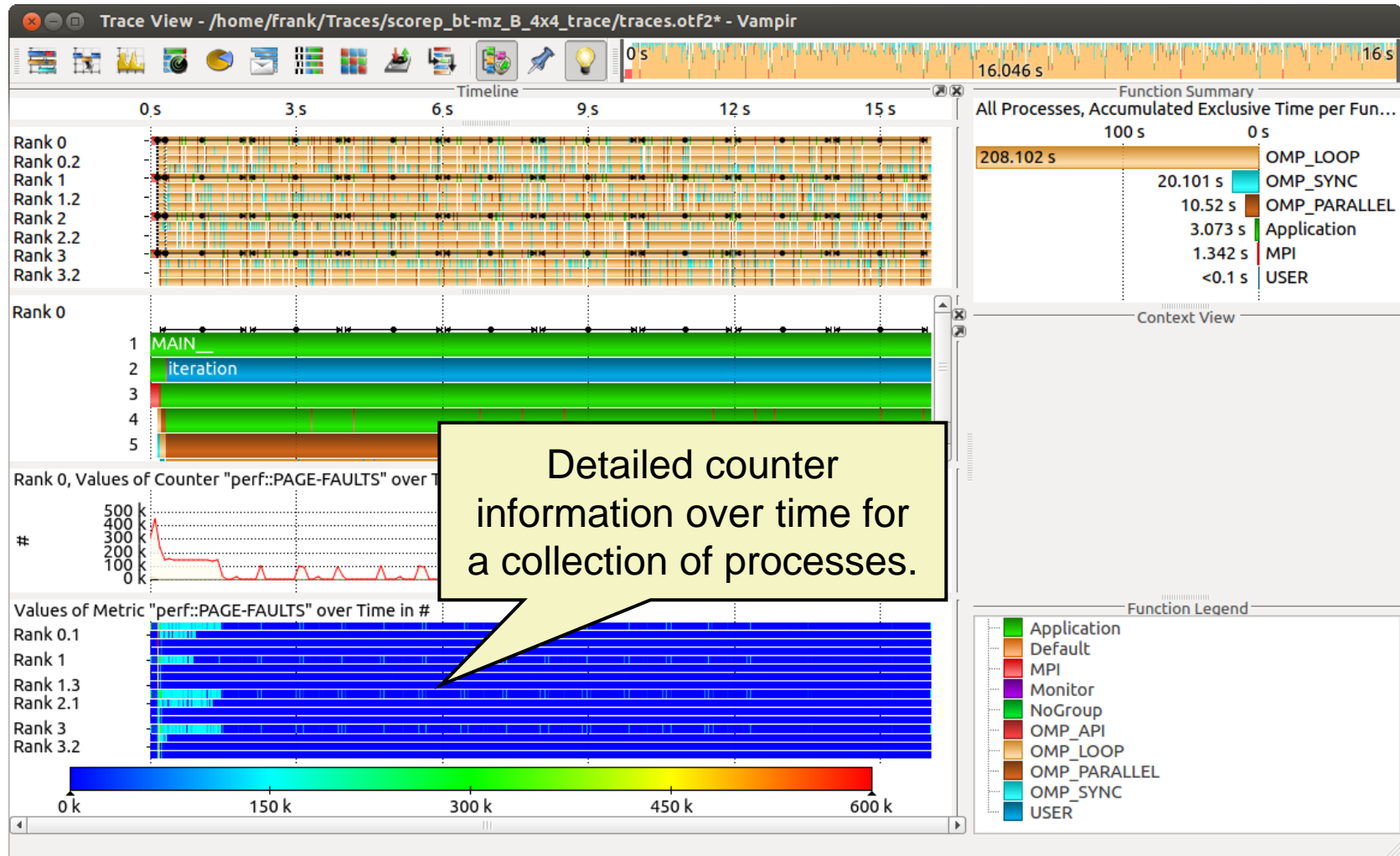
Counter Data Timeline



Vampir: Example Visualization

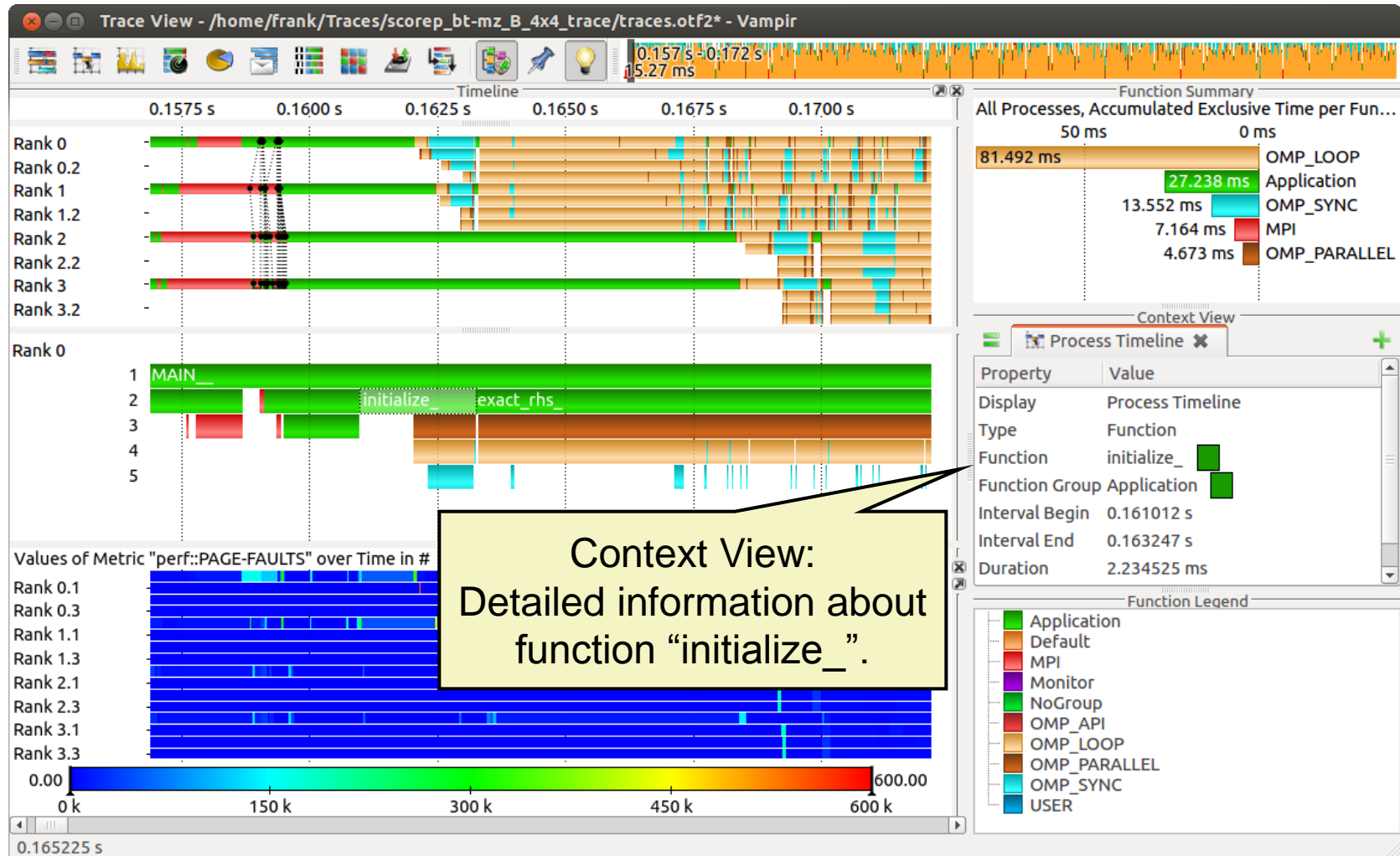


Performance Radar



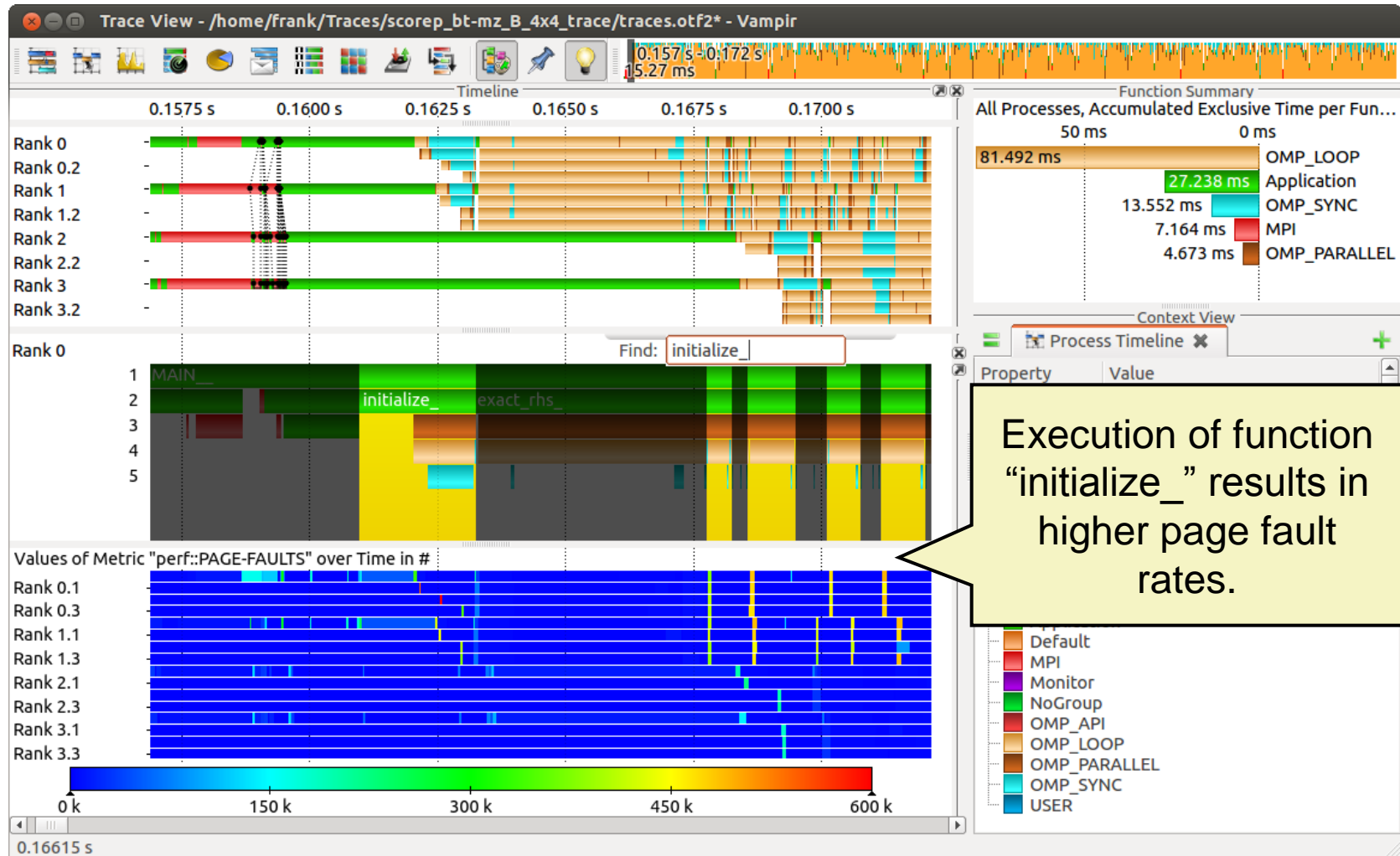
Vampir: Example Visualization

Zoom in: Initialisation Phase



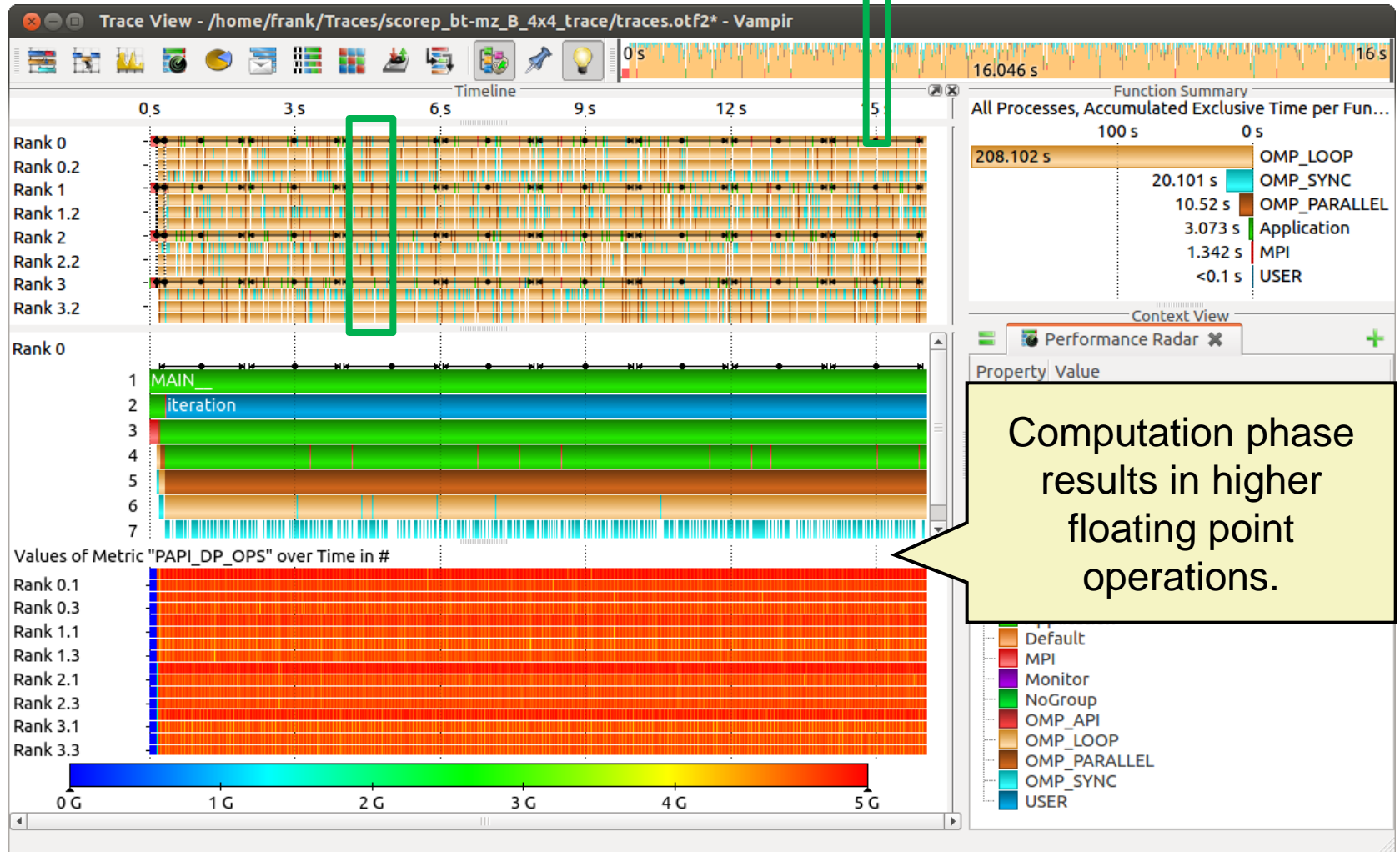
Vampir: Example Visualization

Feature: Find Function



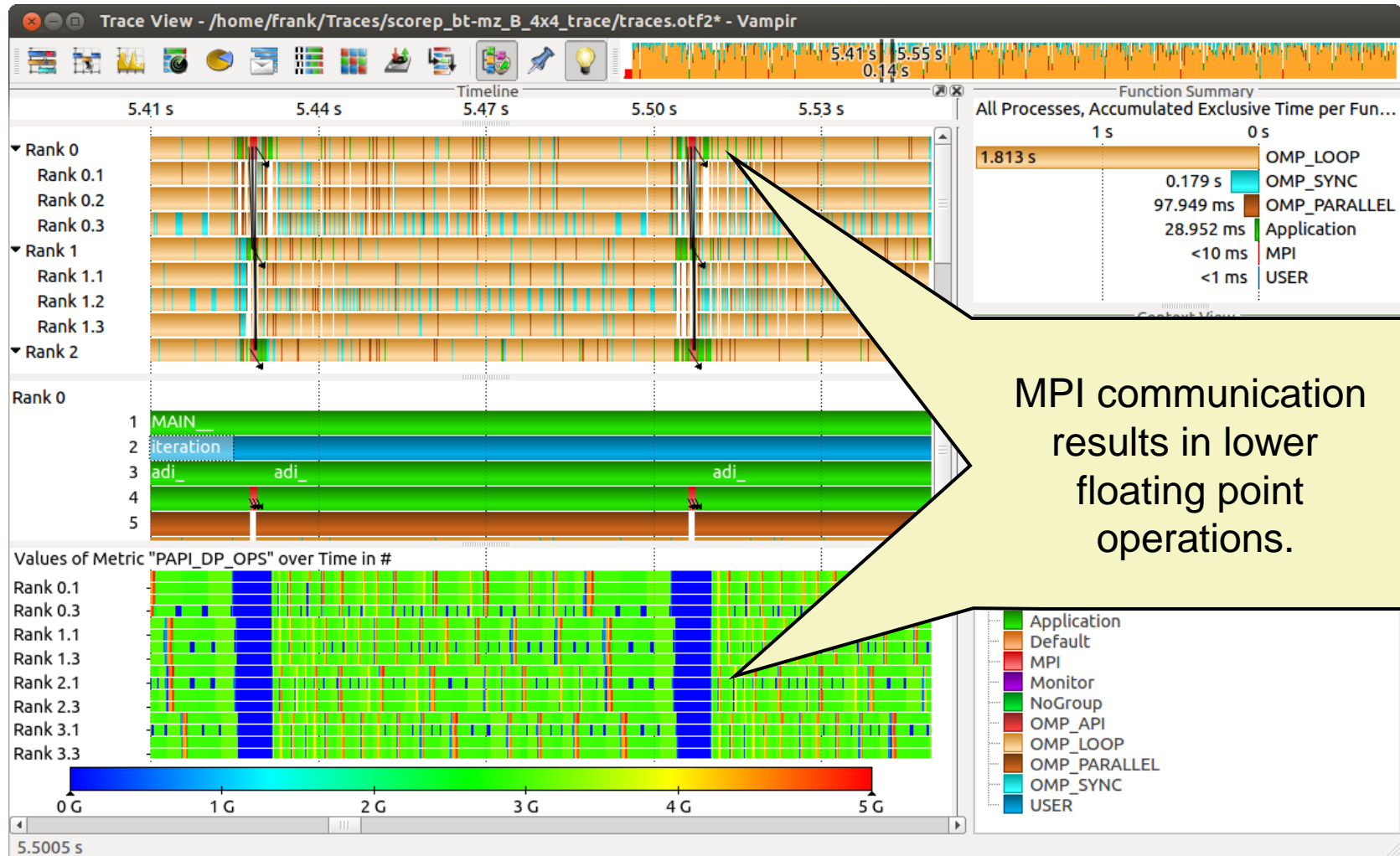
Vampir: Example Visualization

Computation Phase



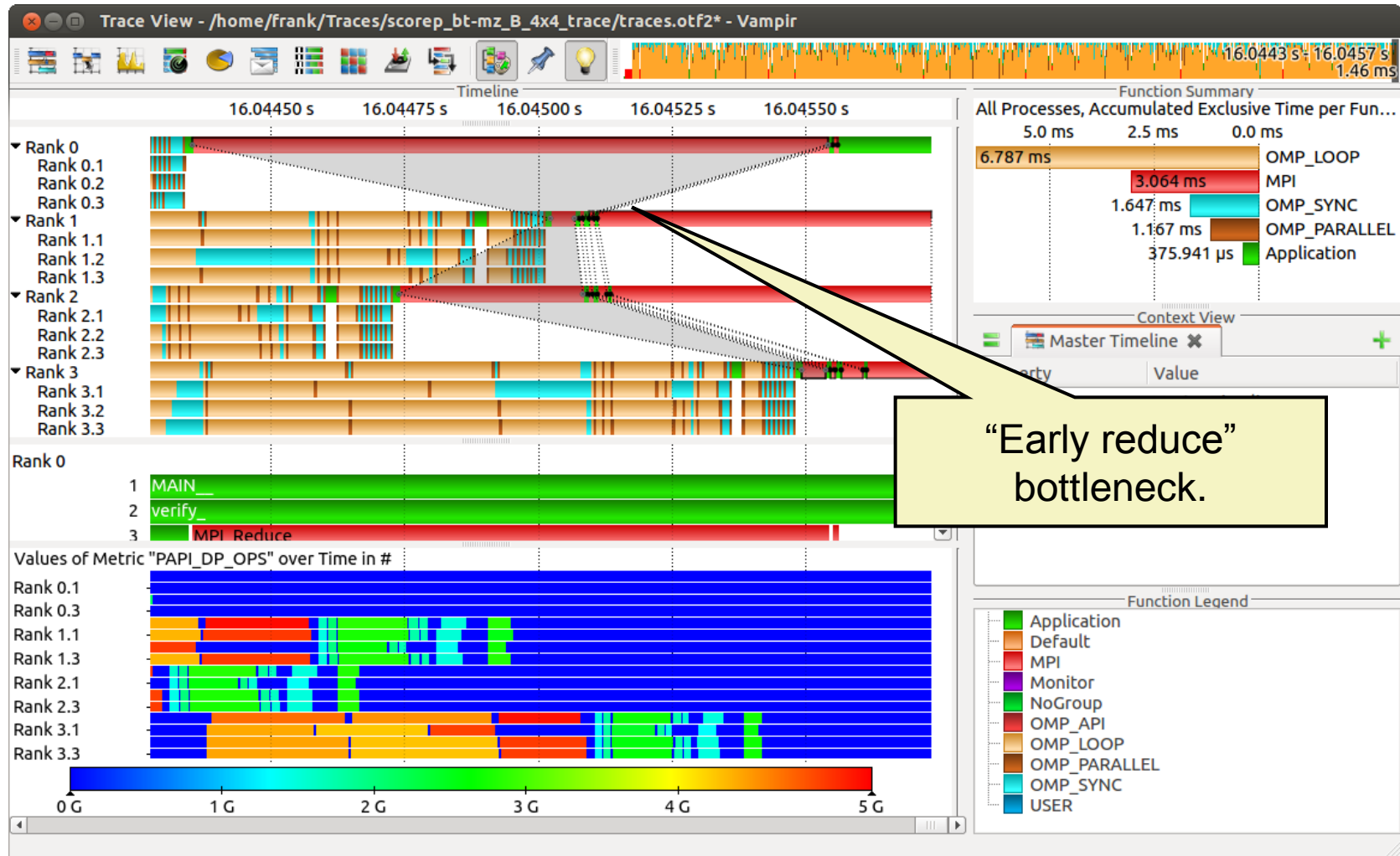
Vampir: Example Visualization

Zoom in: Computation Phase



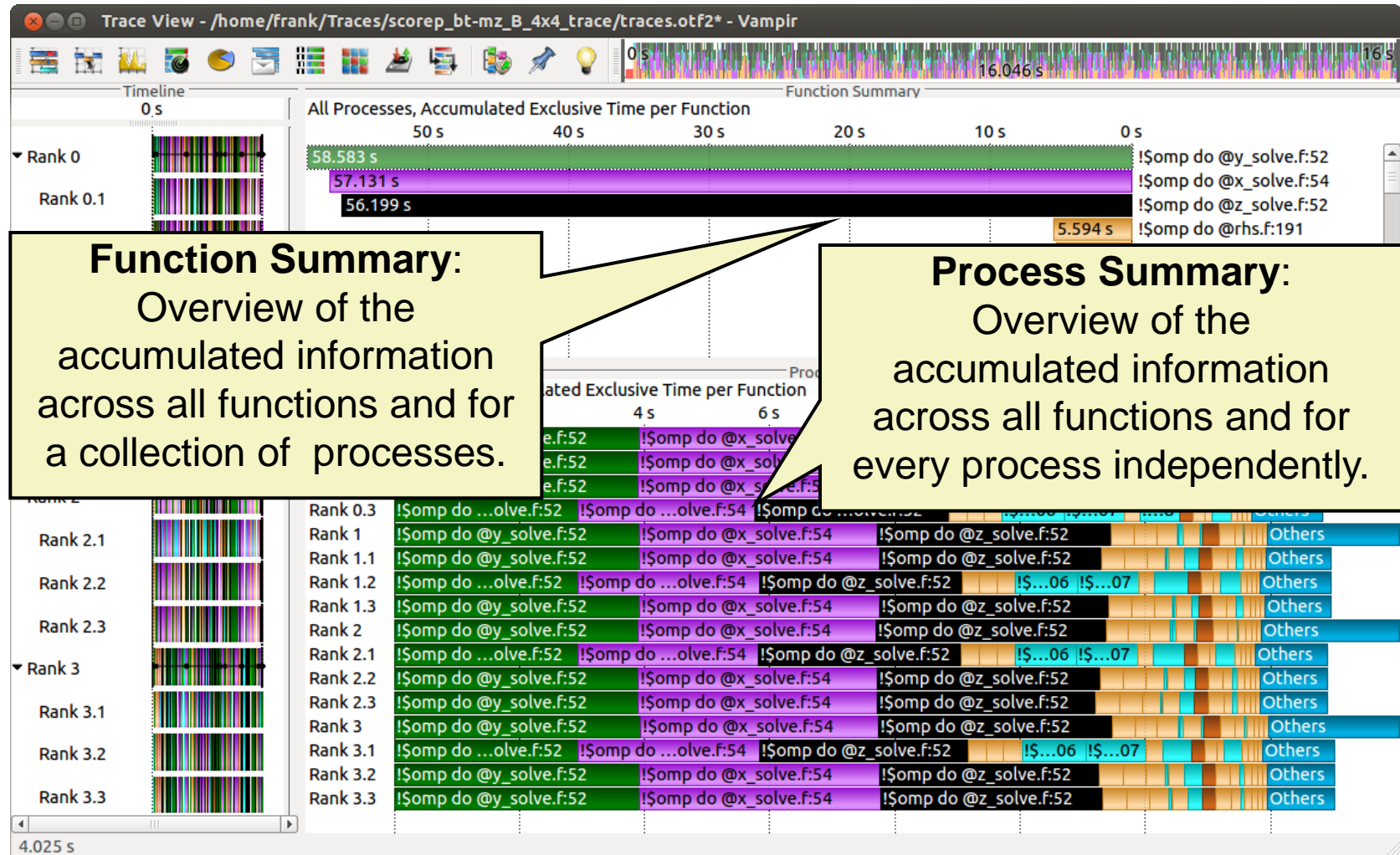
Vampir: Example Visualization

Zoom in: Finalisation Phase



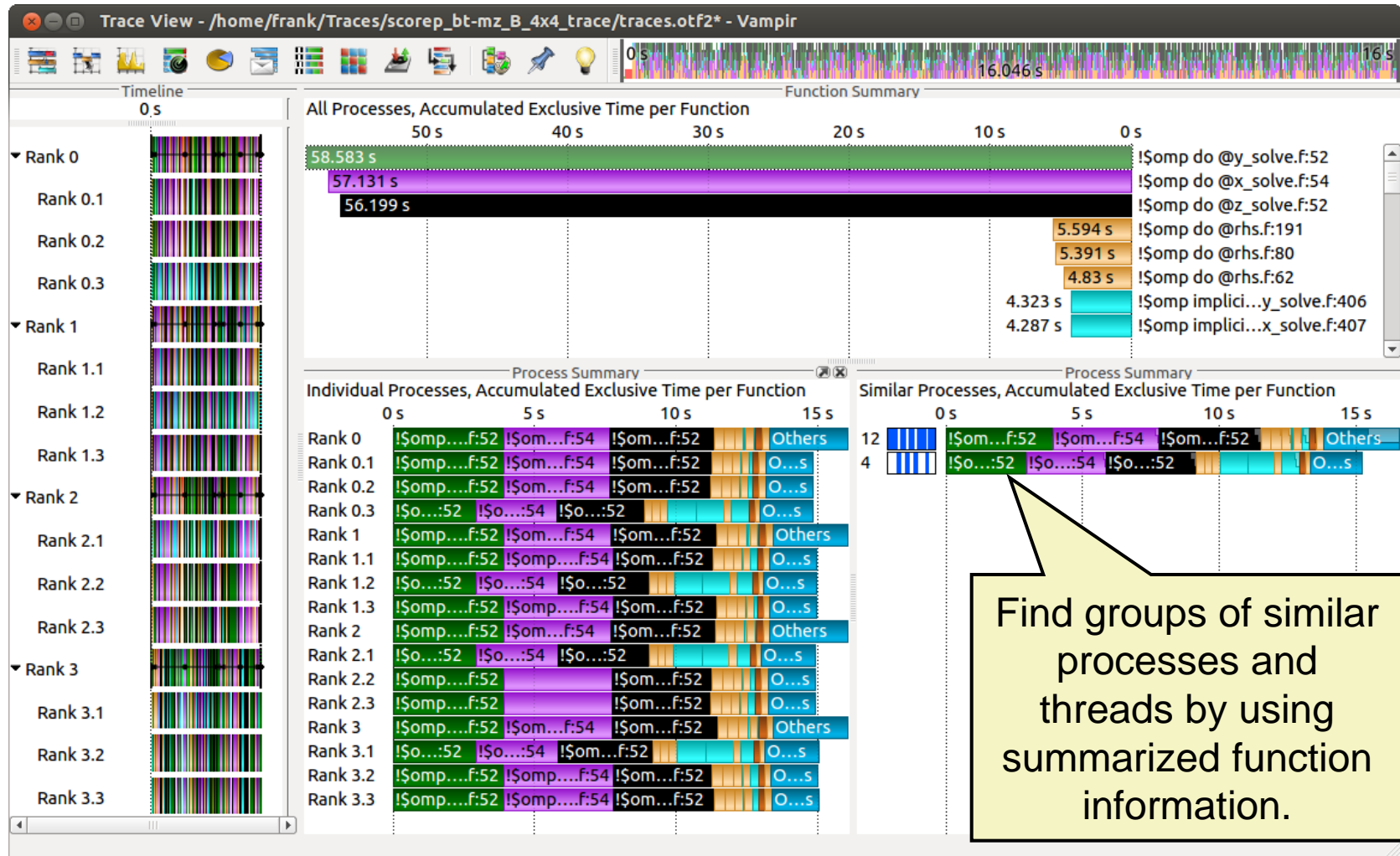


Process Summary

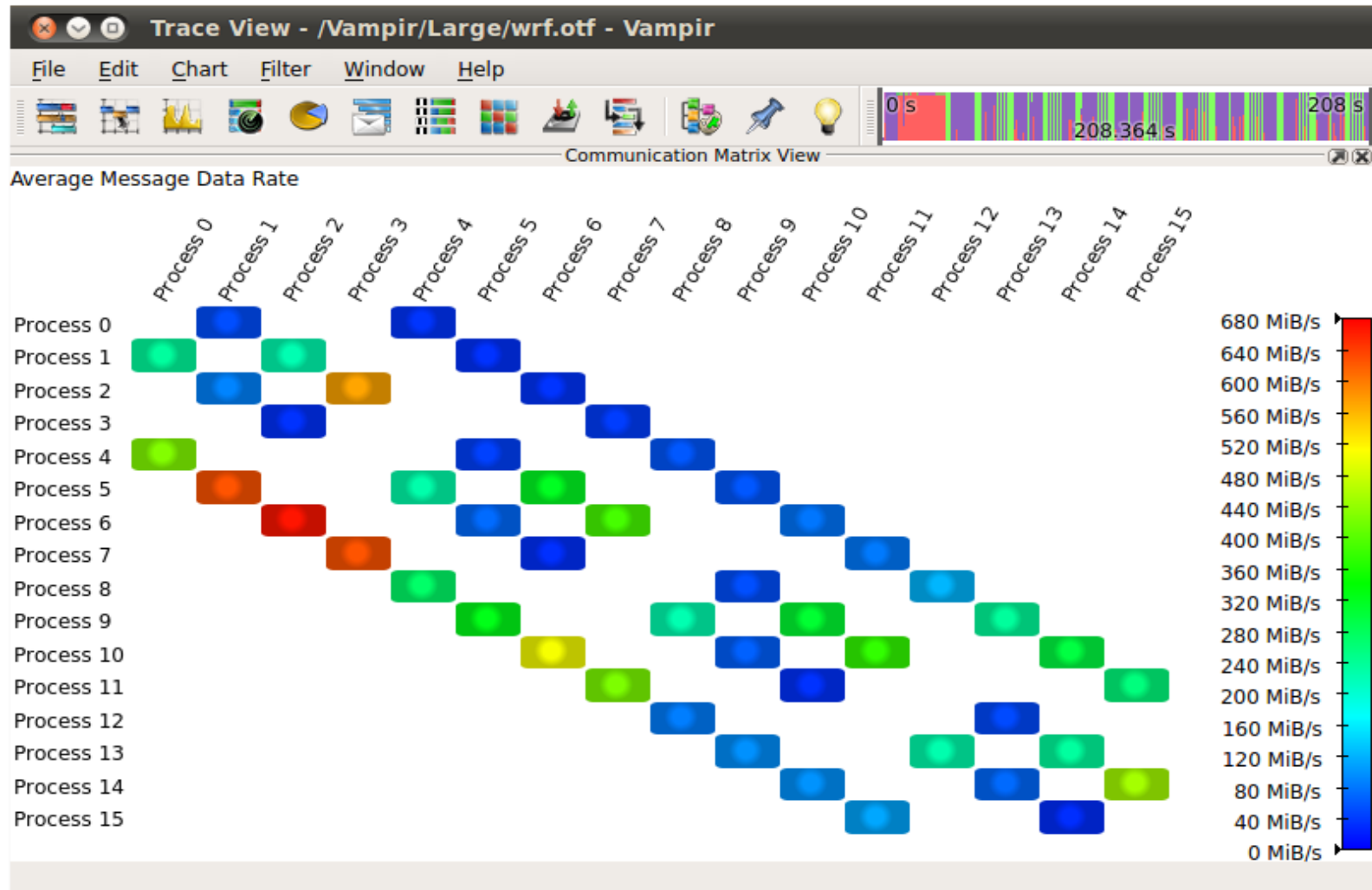




Process Summary

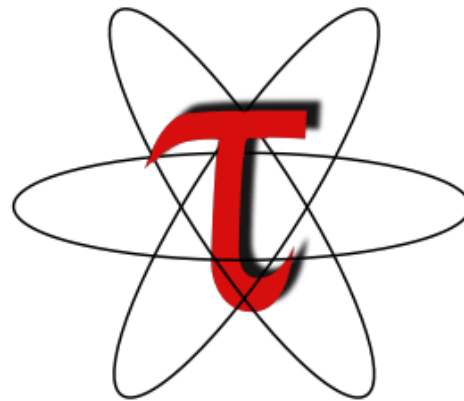


Communication matrix



Outline

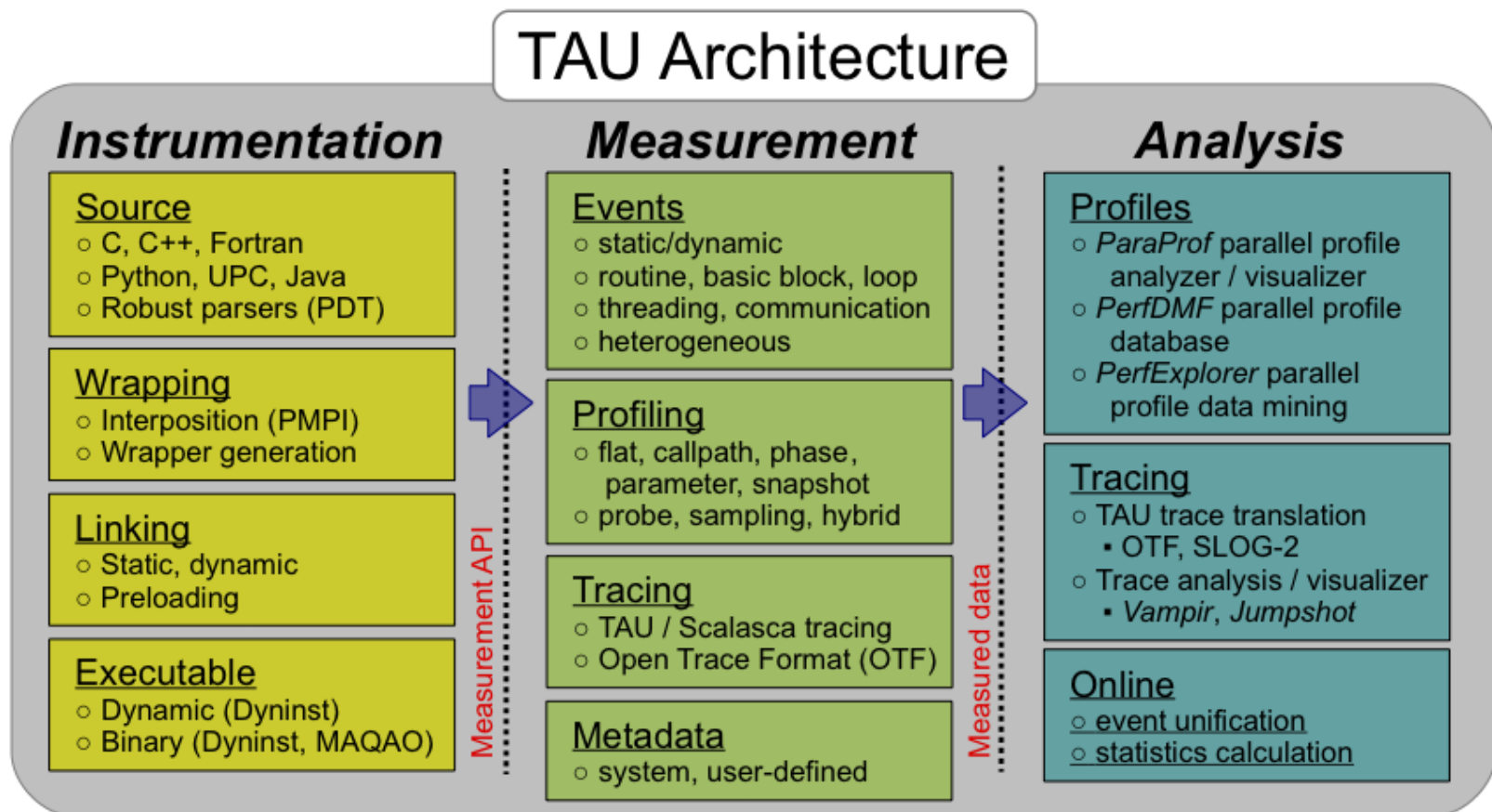
- Introduction
- Code development
- Performance analysis and tuning
 - Concepts and basics
 - Selected performance issues
 - Selected performance analysis tools
 - Score-P
 - Scalasca
 - CUBE
 - Vampir
 - TAU
 - Use cases
- Summary



TAU is available at <http://tau.uoregon.edu>,
Free download, open source, BSD license

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

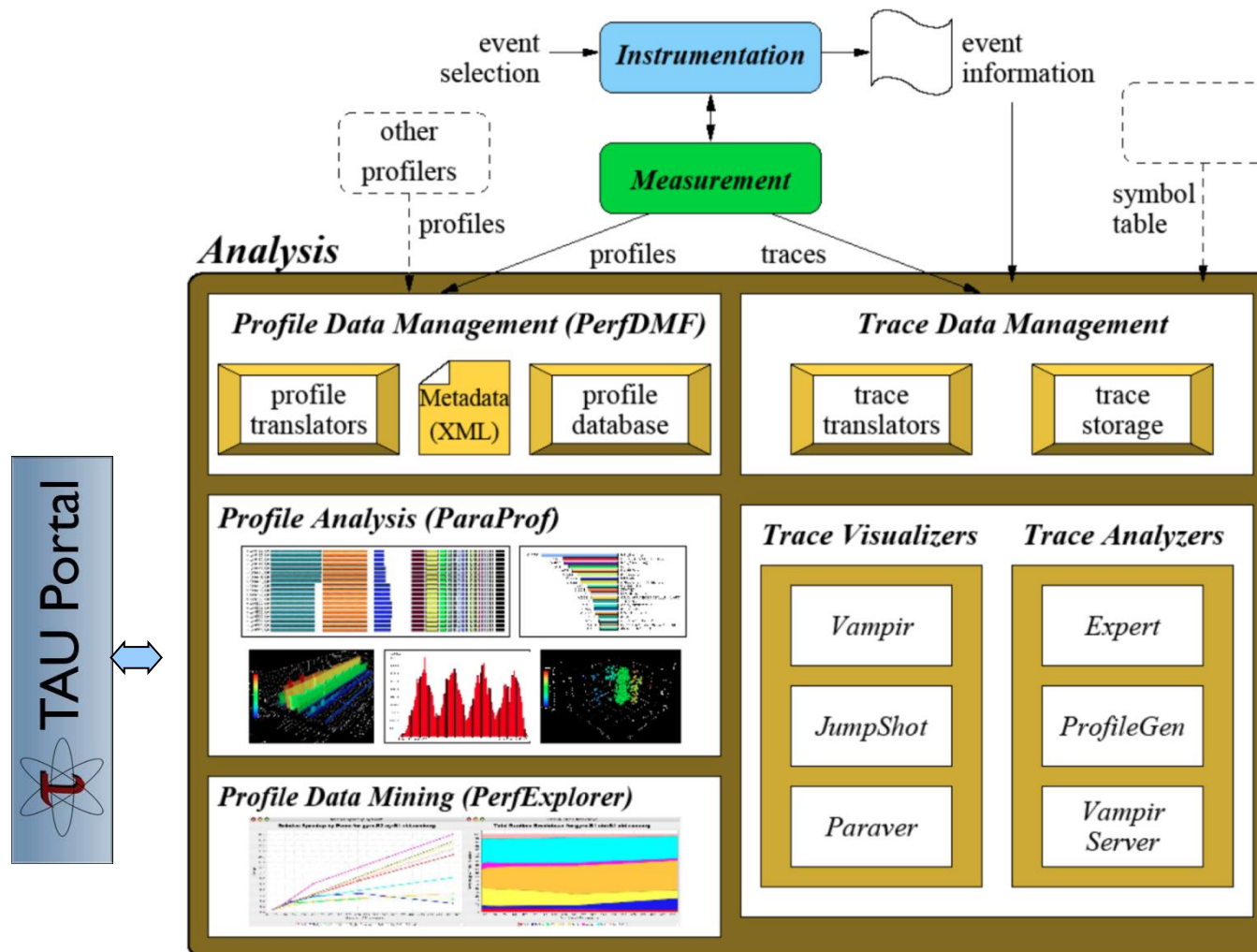
Measurement and analysis support

- 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

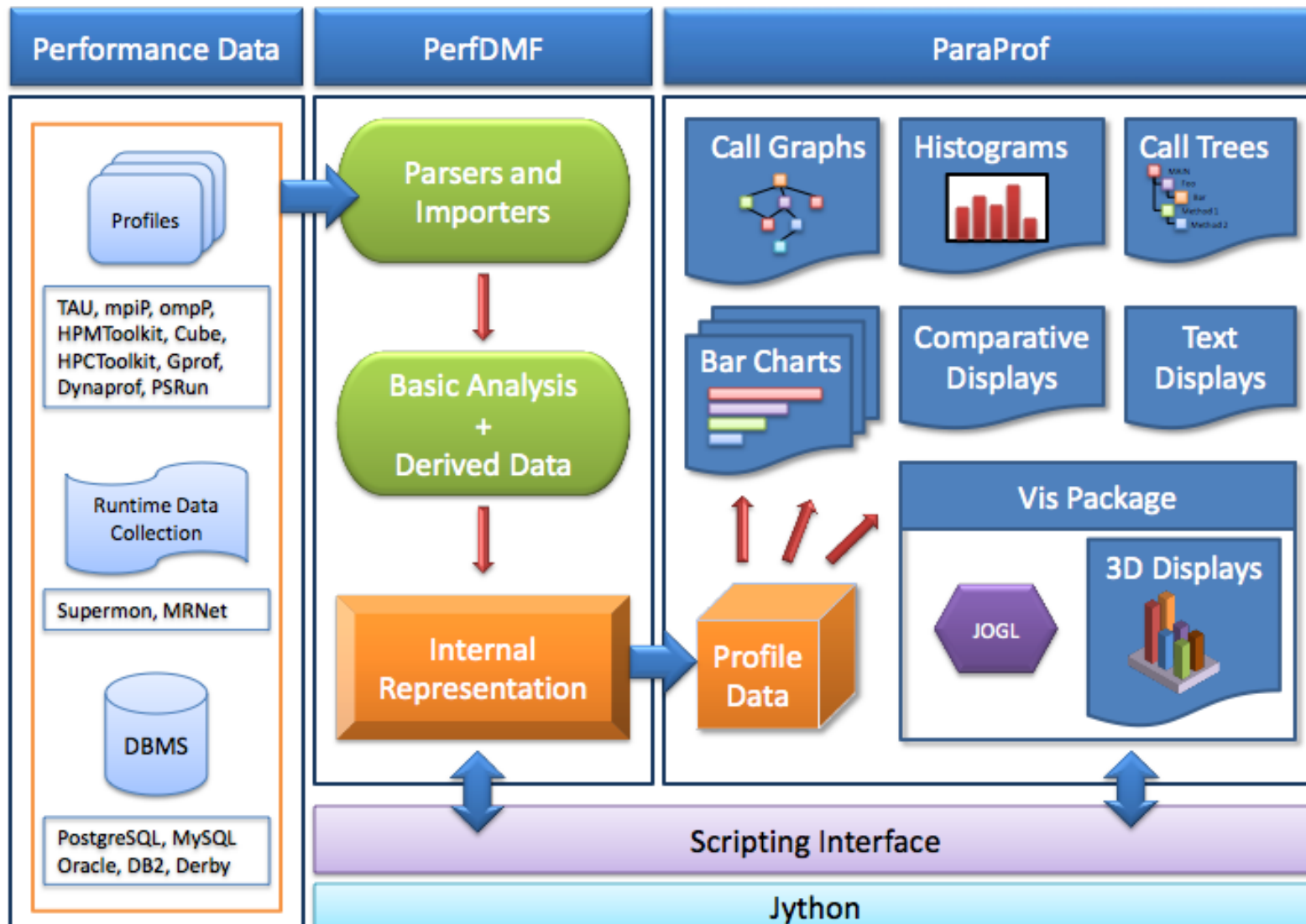
Analysis

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

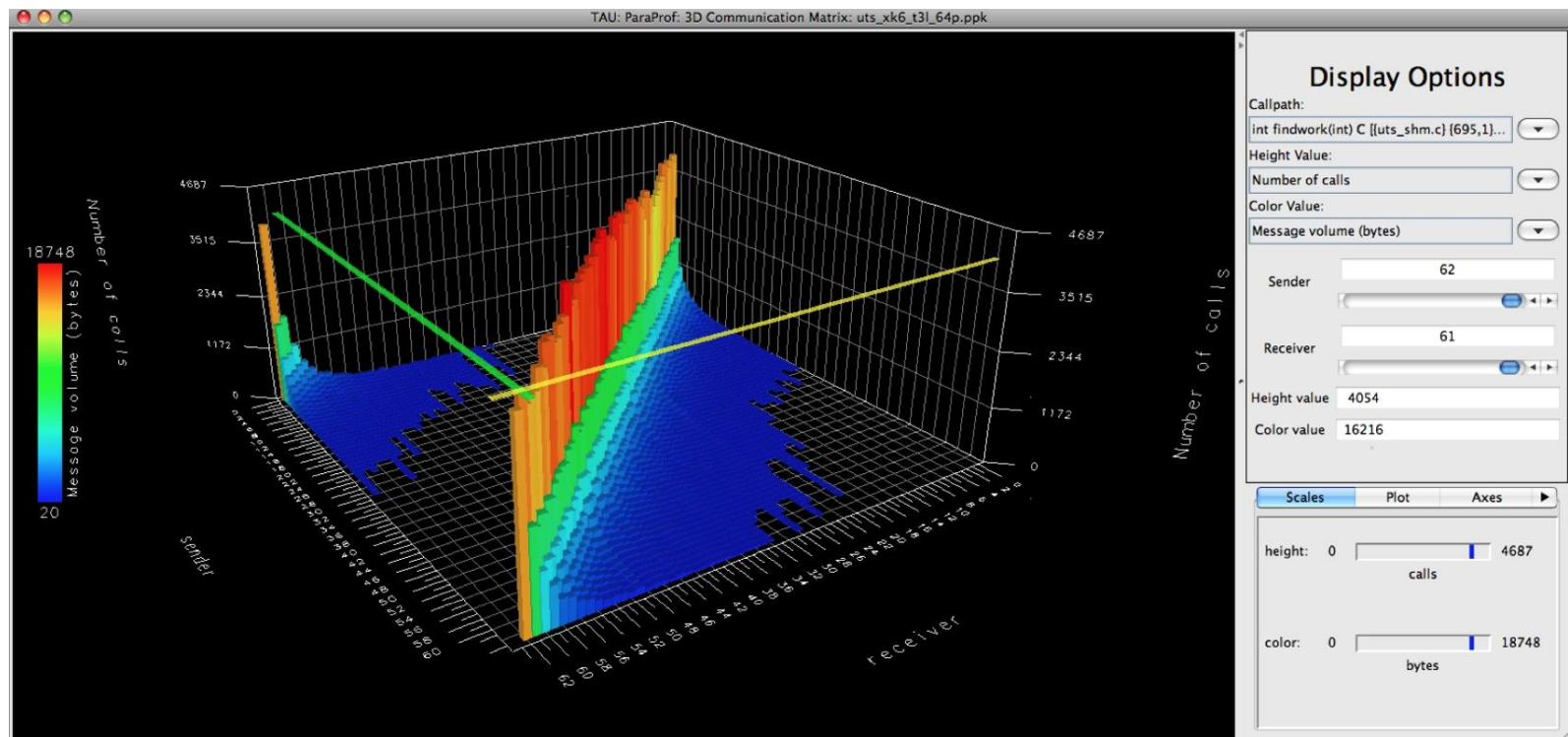
TAU Analysis



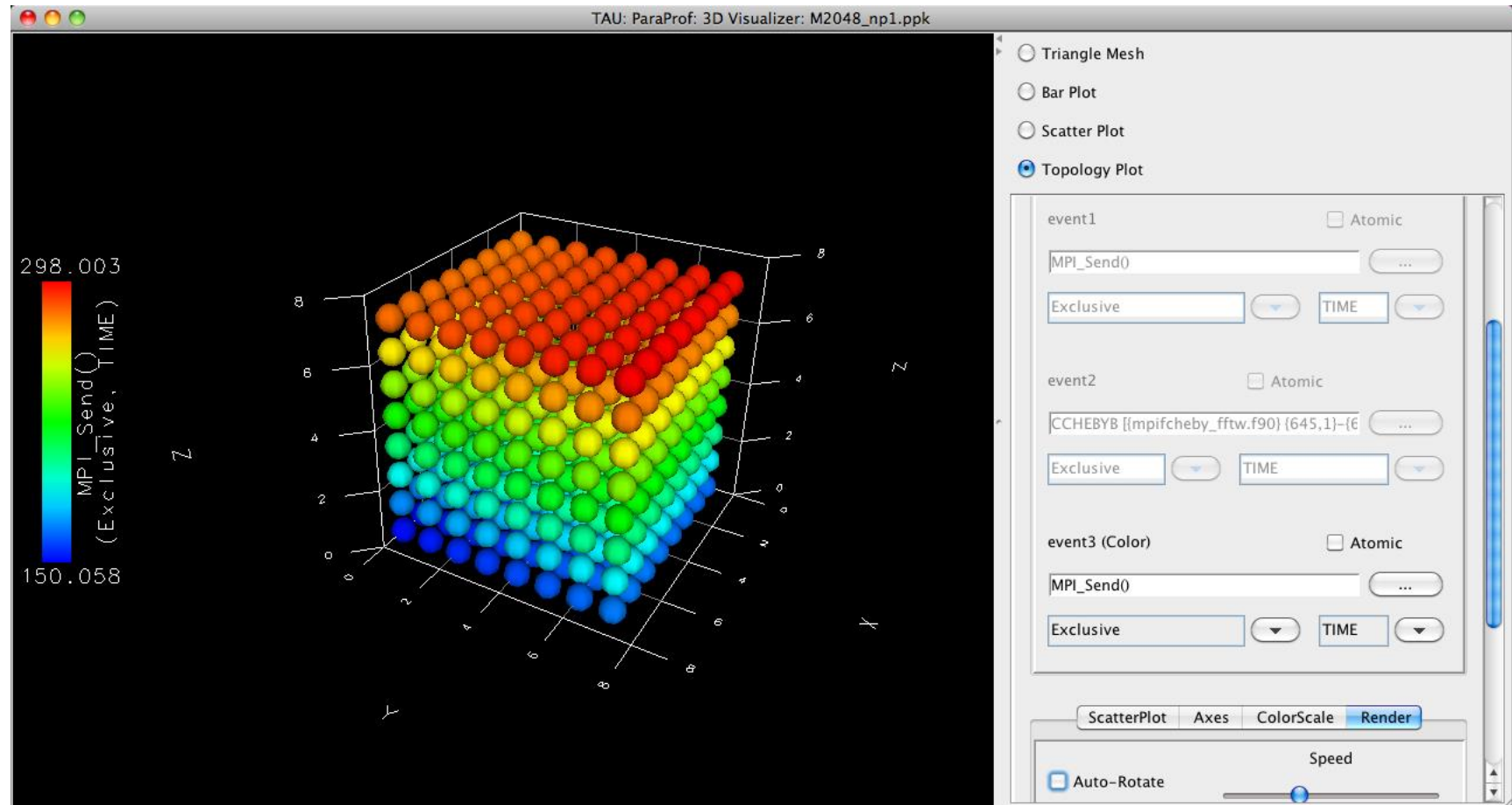
ParaProf Profile Analysis Framework



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

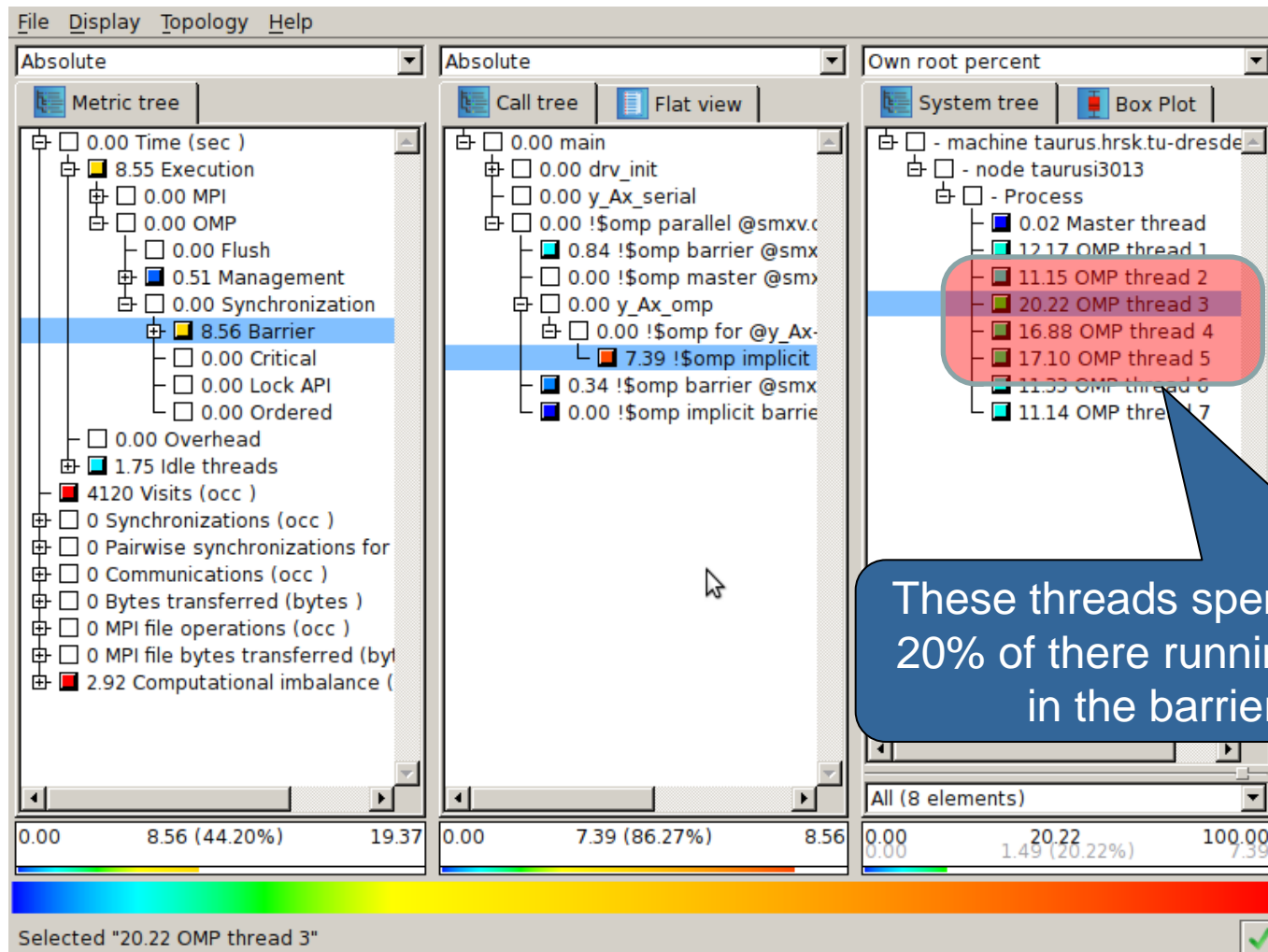
Naïve OpenMP Algorithm:

```
#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]
```

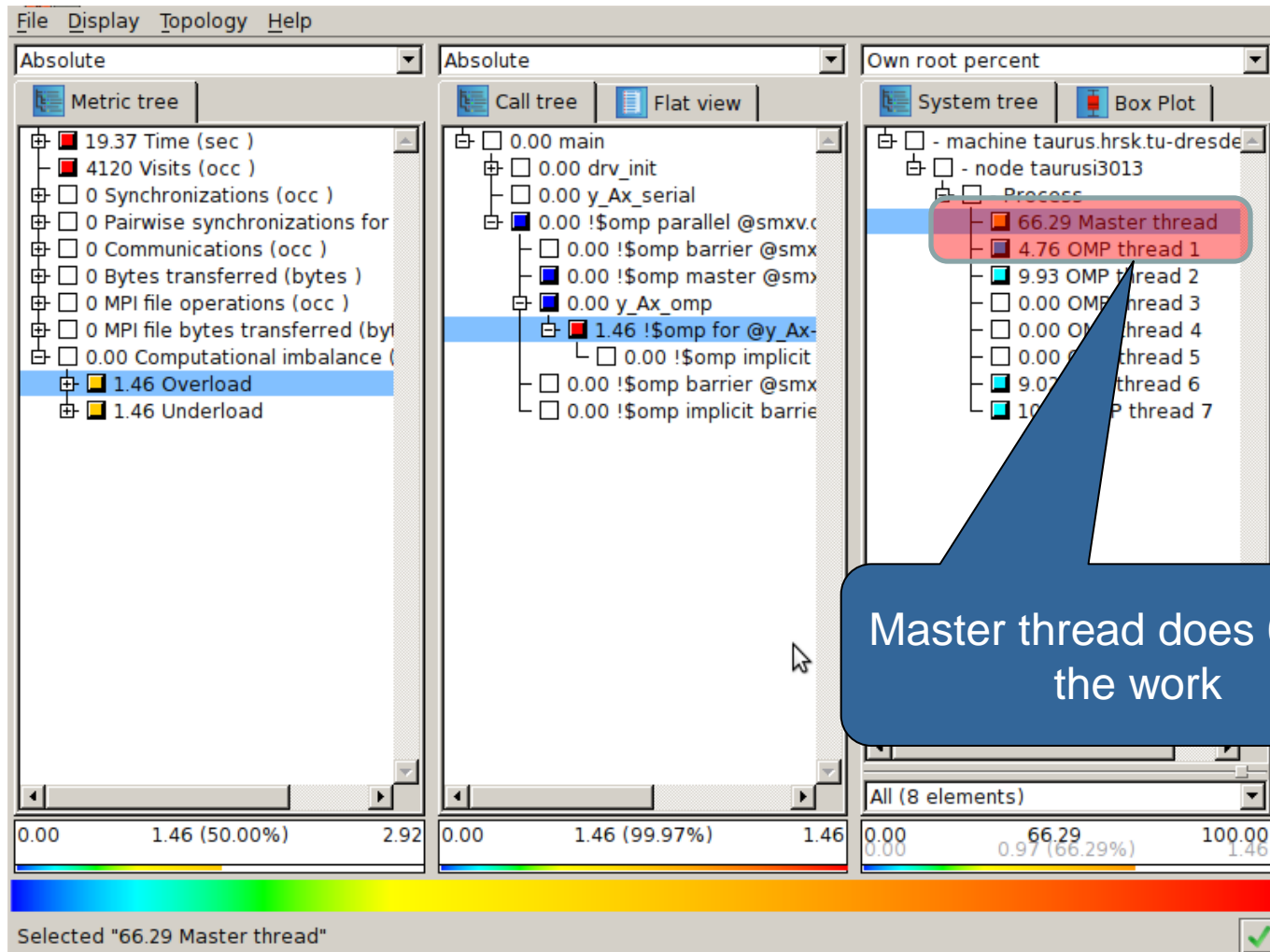
Distributes the rows of A evenly across the threads in the parallel region

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

Time Spent in OpenMP Barriers



Computational Imbalance



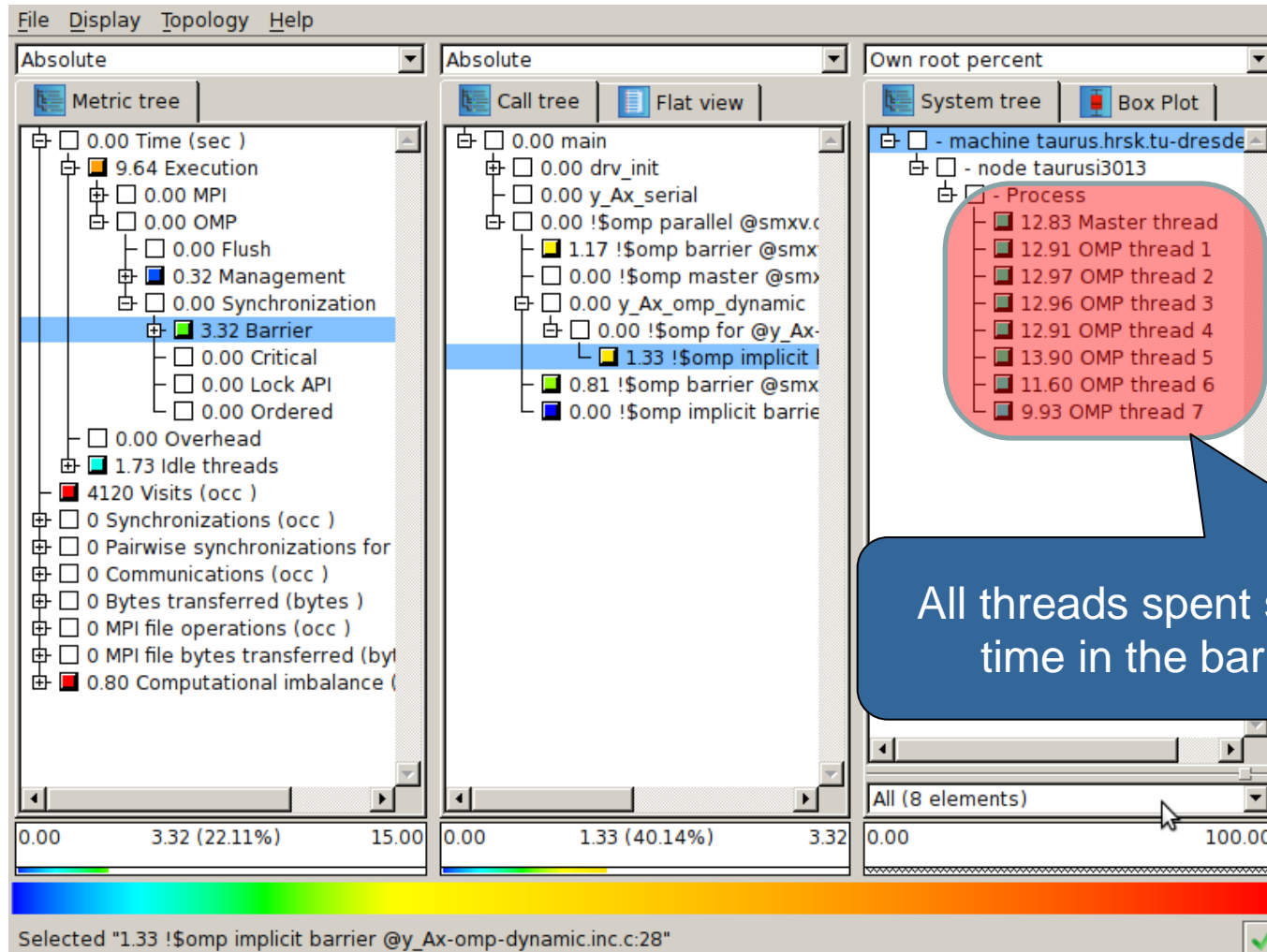
Sparse Matrix Vector Multiplication

Improved OpenMP Algorithm

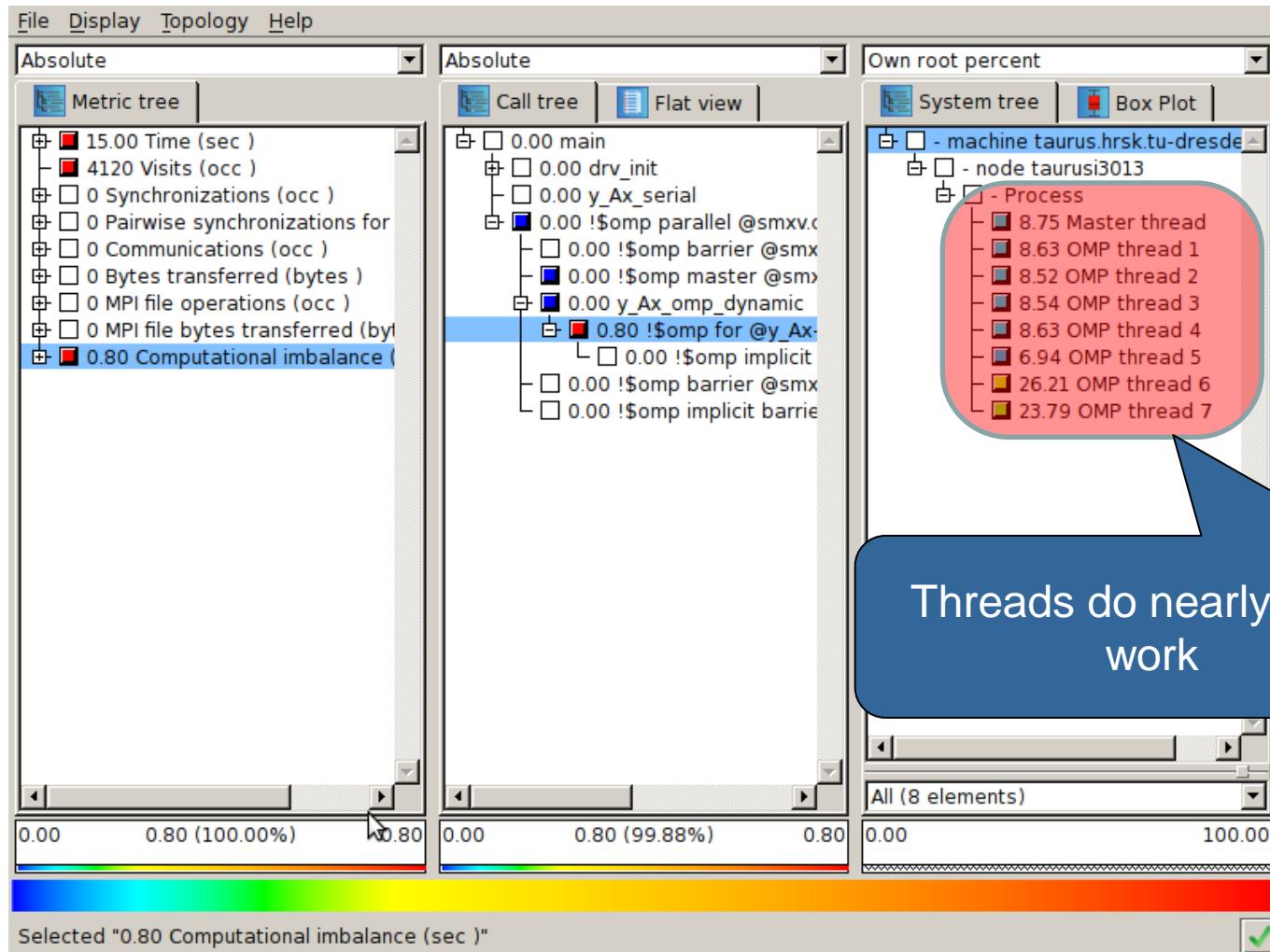
```
#pragma omp parallel for schedule(dynamic,1000)
foreach row r in A
  y[r.x] = 0
  foreach non-zero element e in row
    y[r.x] += e.value * x[e.y]
```

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

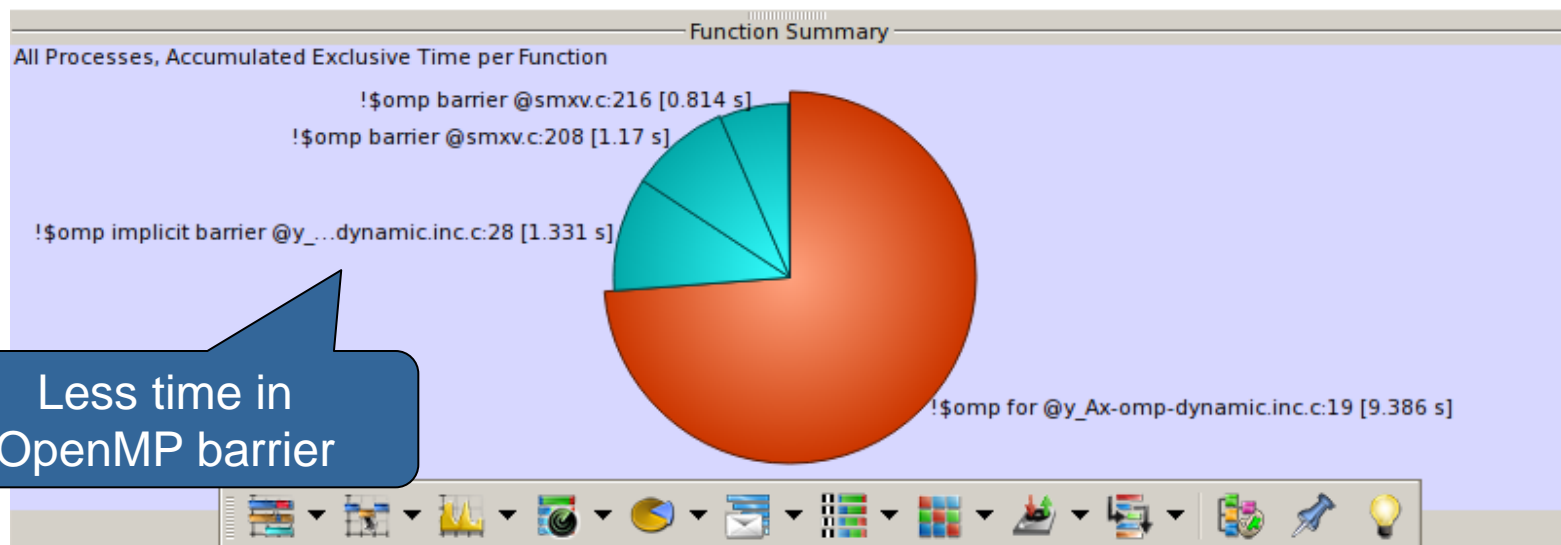
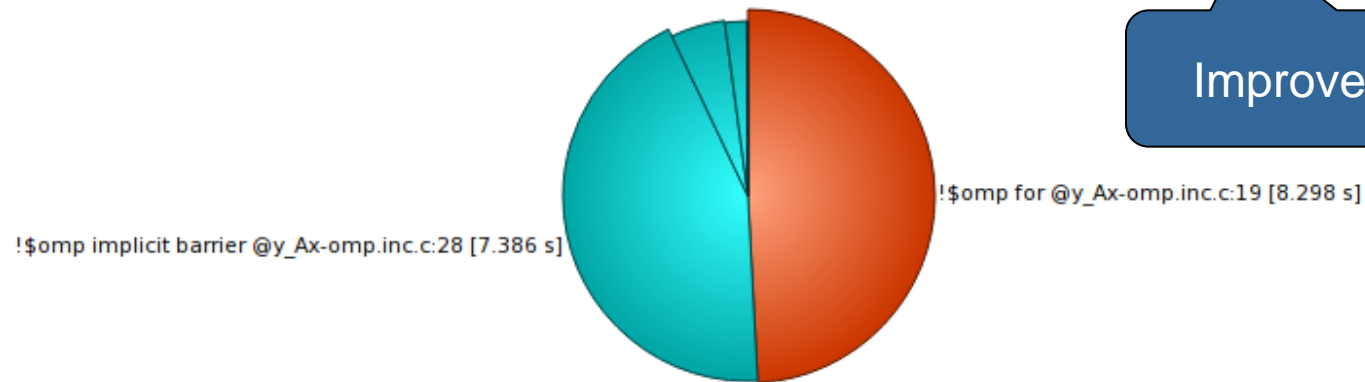
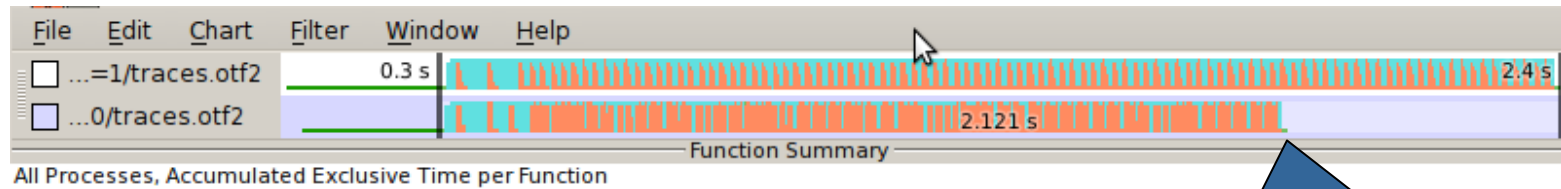
Time Spent in OpenMP Barriers



Computational Imbalance

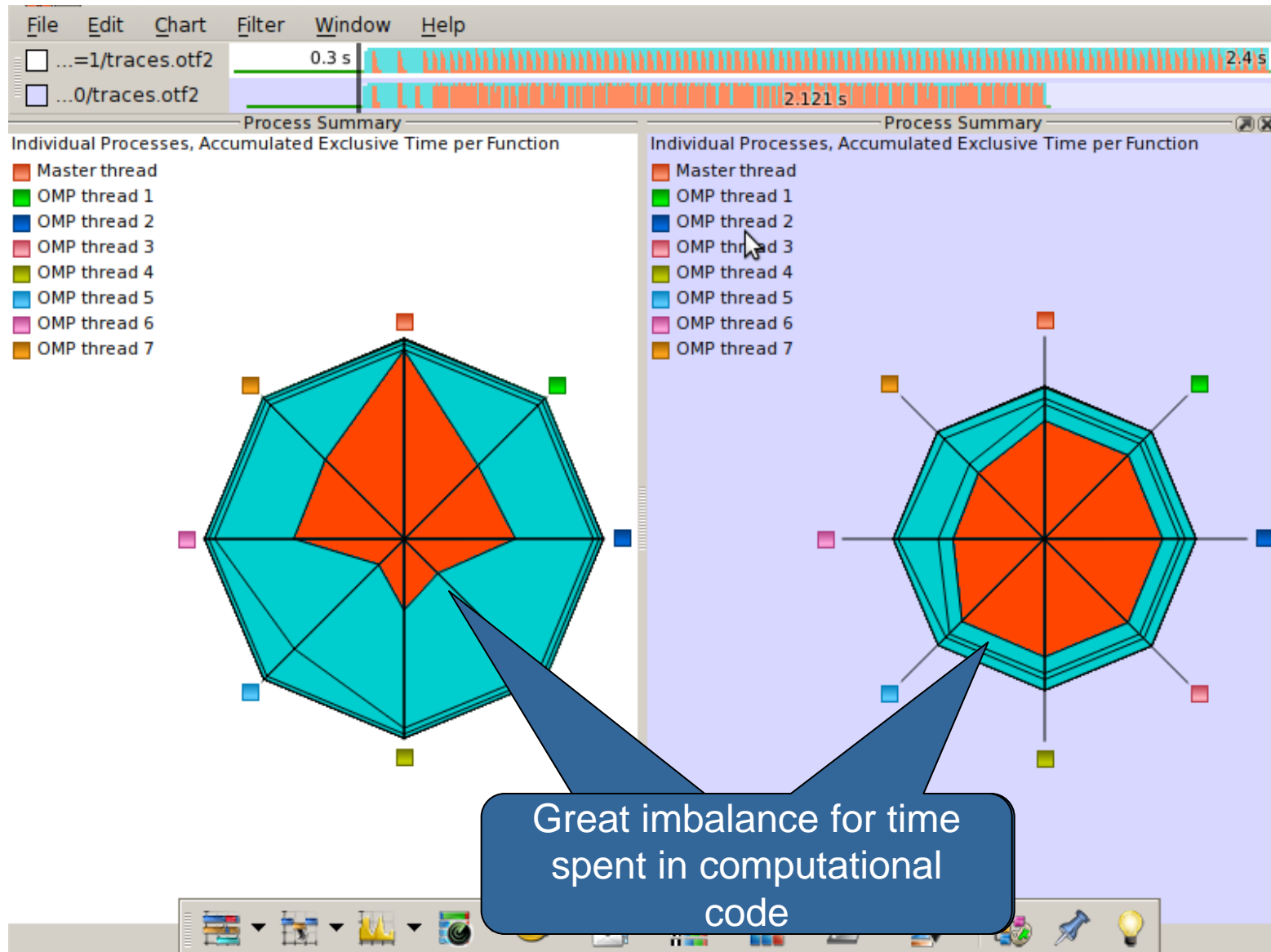


Time Spent in OpenMP Barriers



Less time in
OpenMP barrier

Computational Imbalance



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

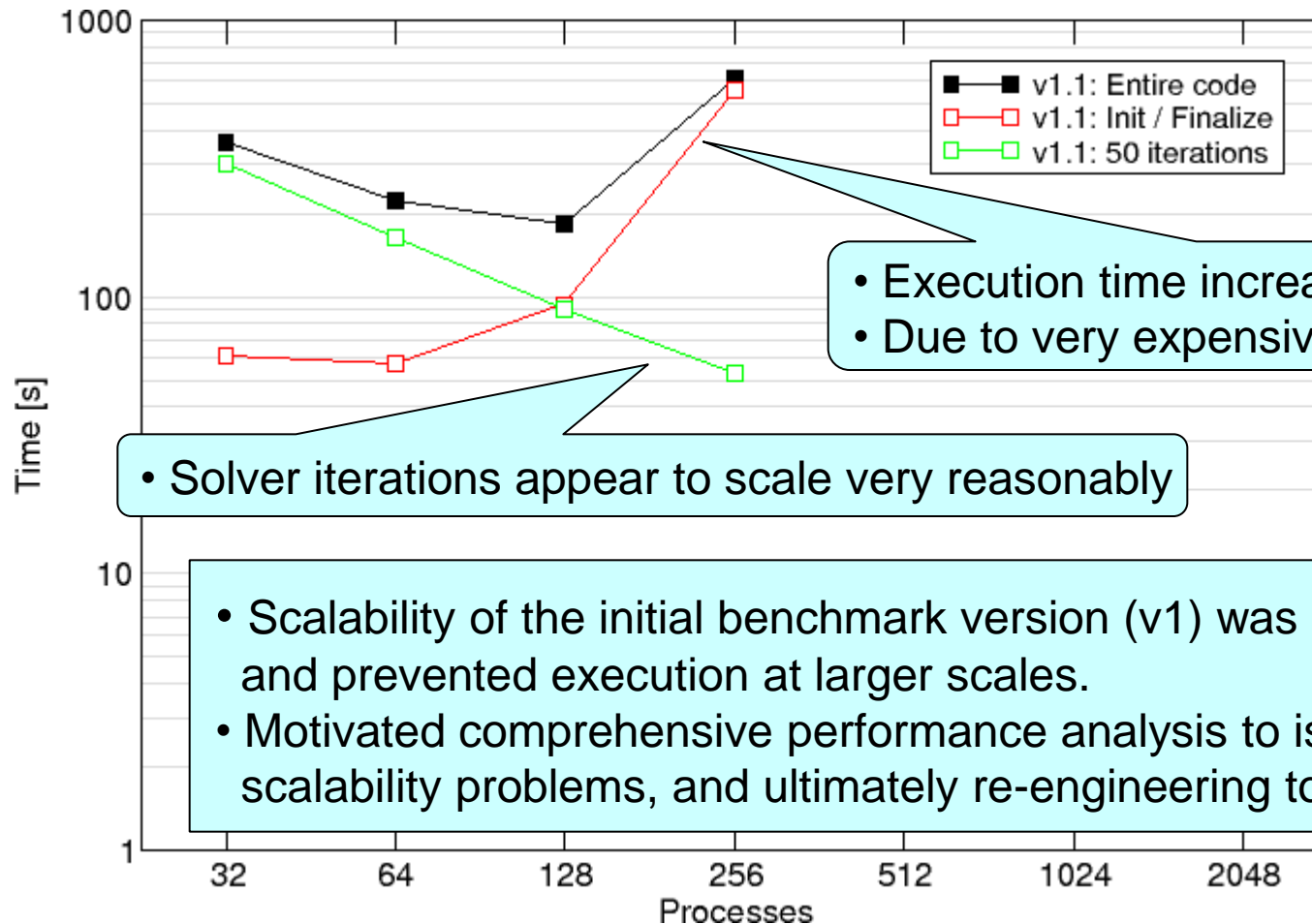
MPI parallel versions in SPEC MPI2007 benchmark suite

- 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



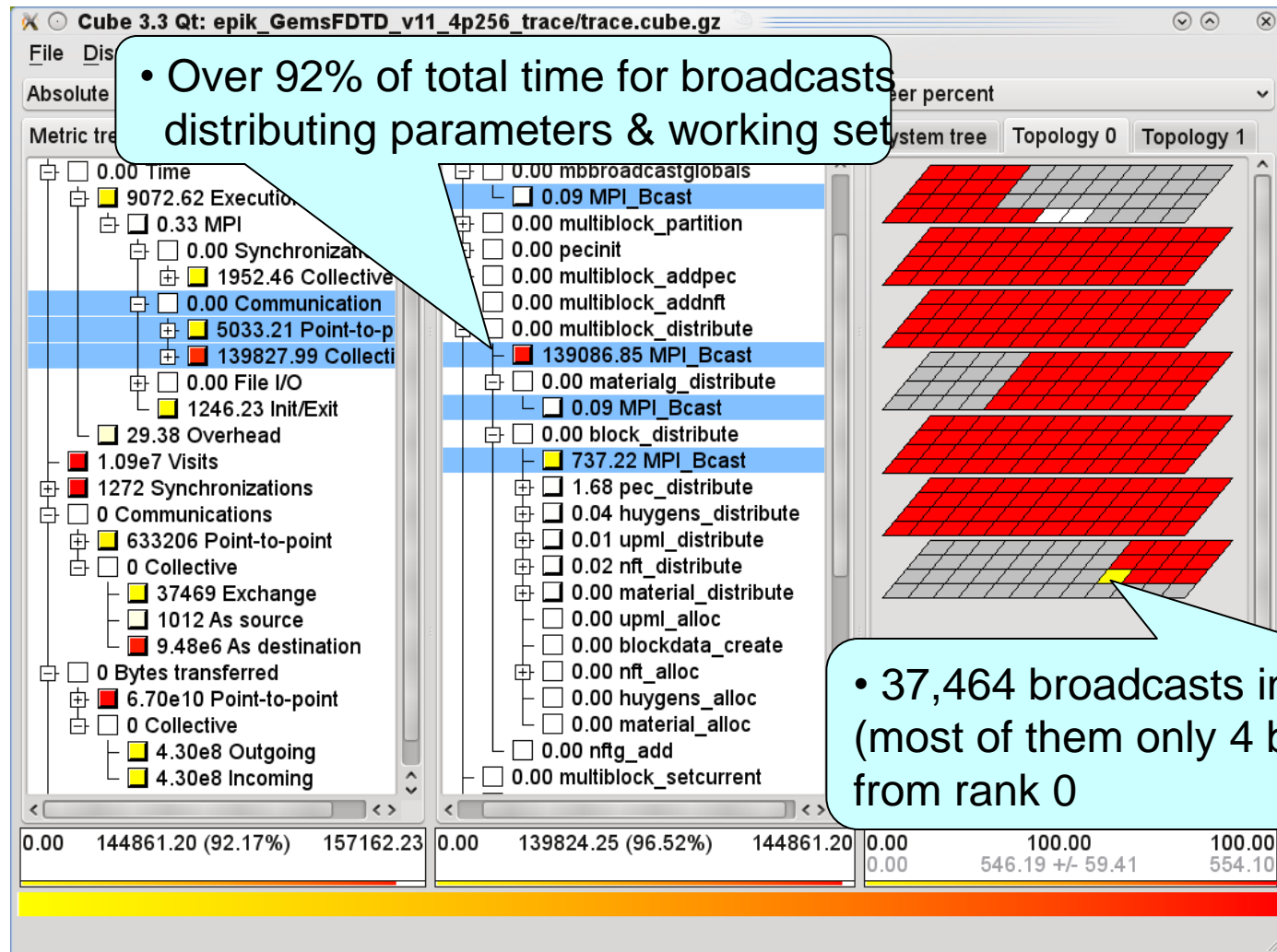
['ltrain' runs
on CrayXT4
HECToR]

- Execution time increases exponentially
- Due to very expensive initialization

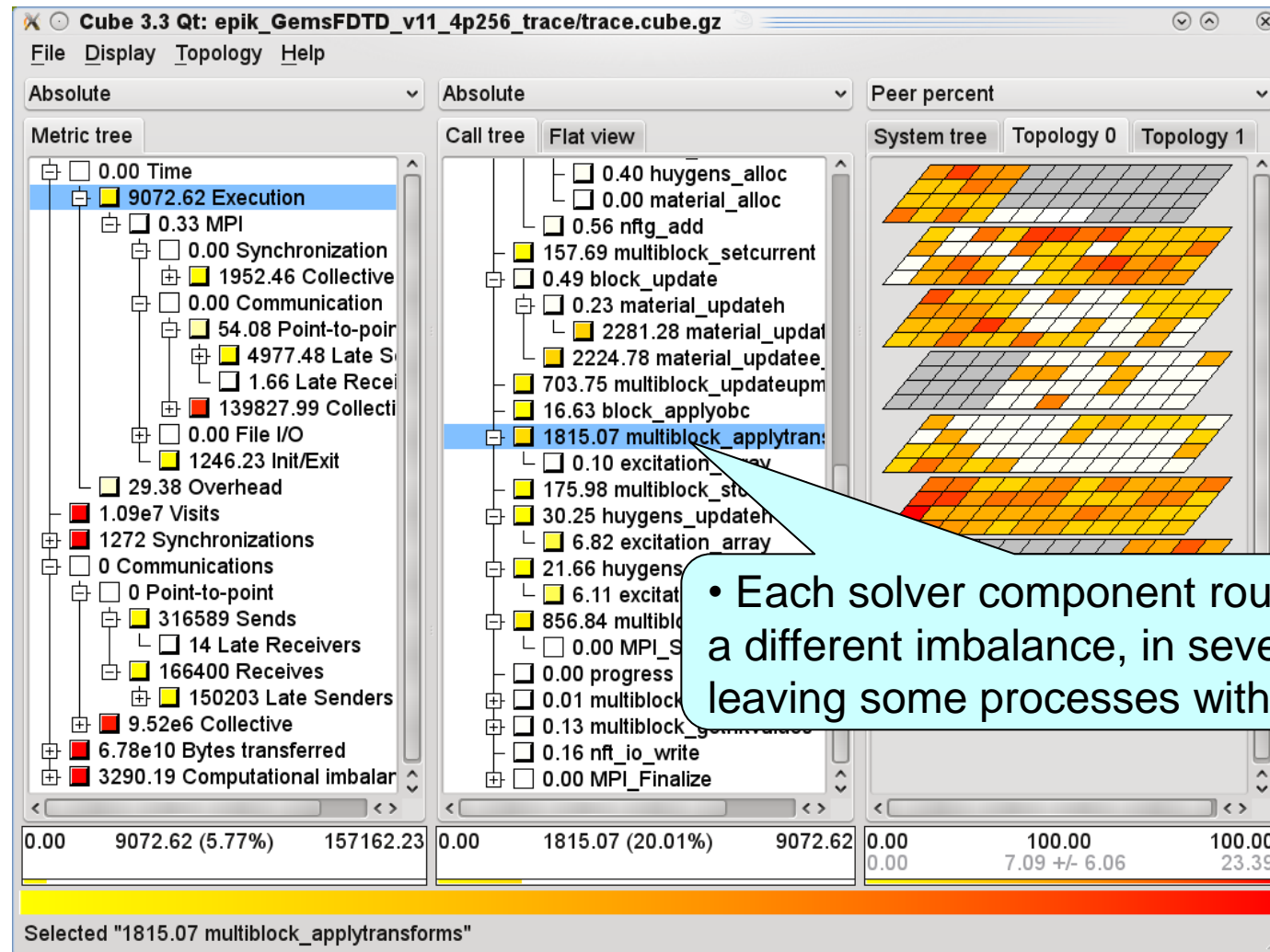
- Solver iterations appear to scale very reasonably

- 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)



GemsFDTD Case Study

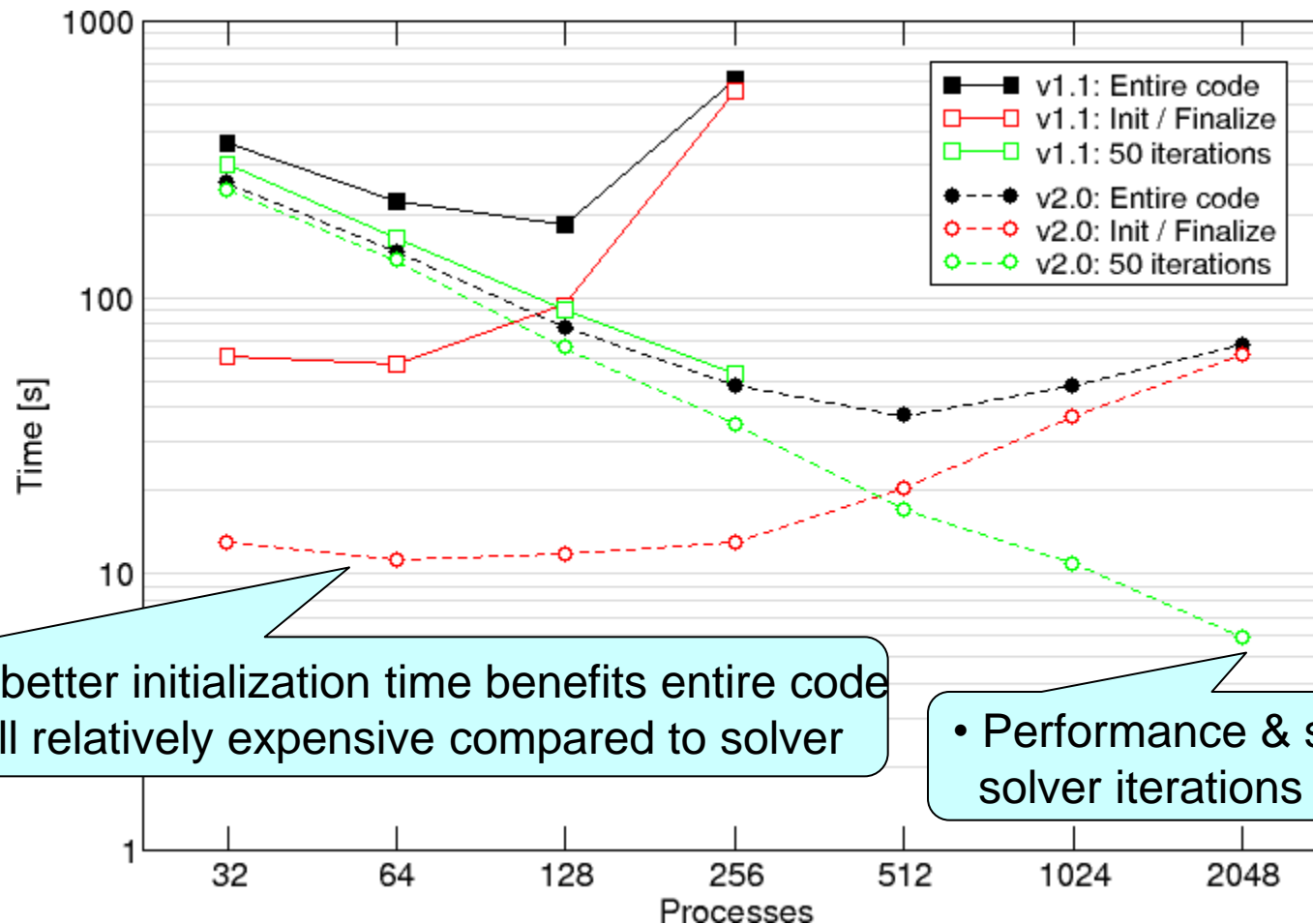
Analysis results

- 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

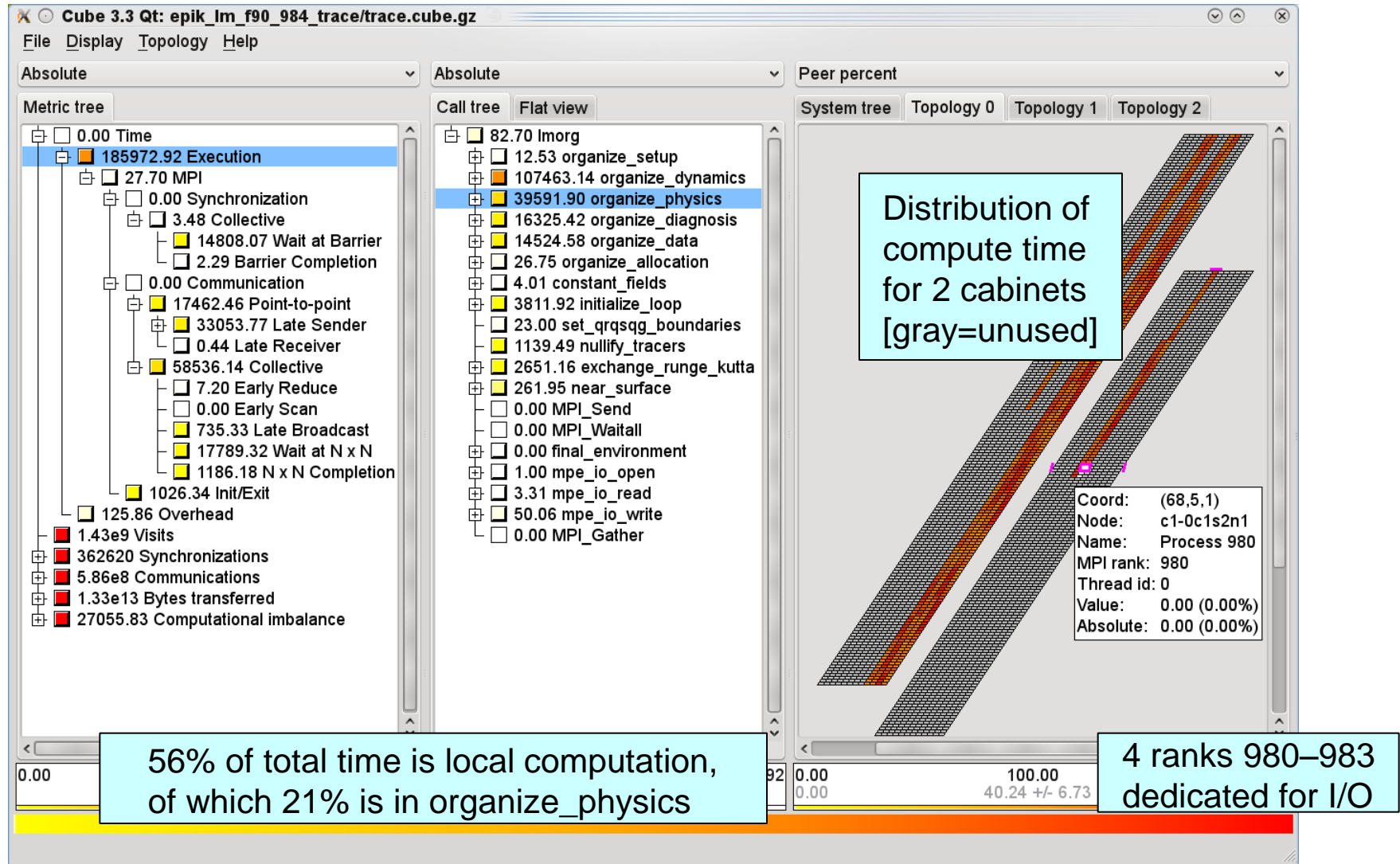
- Western Europe 393x338x60 resolution, 1440 timesteps

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

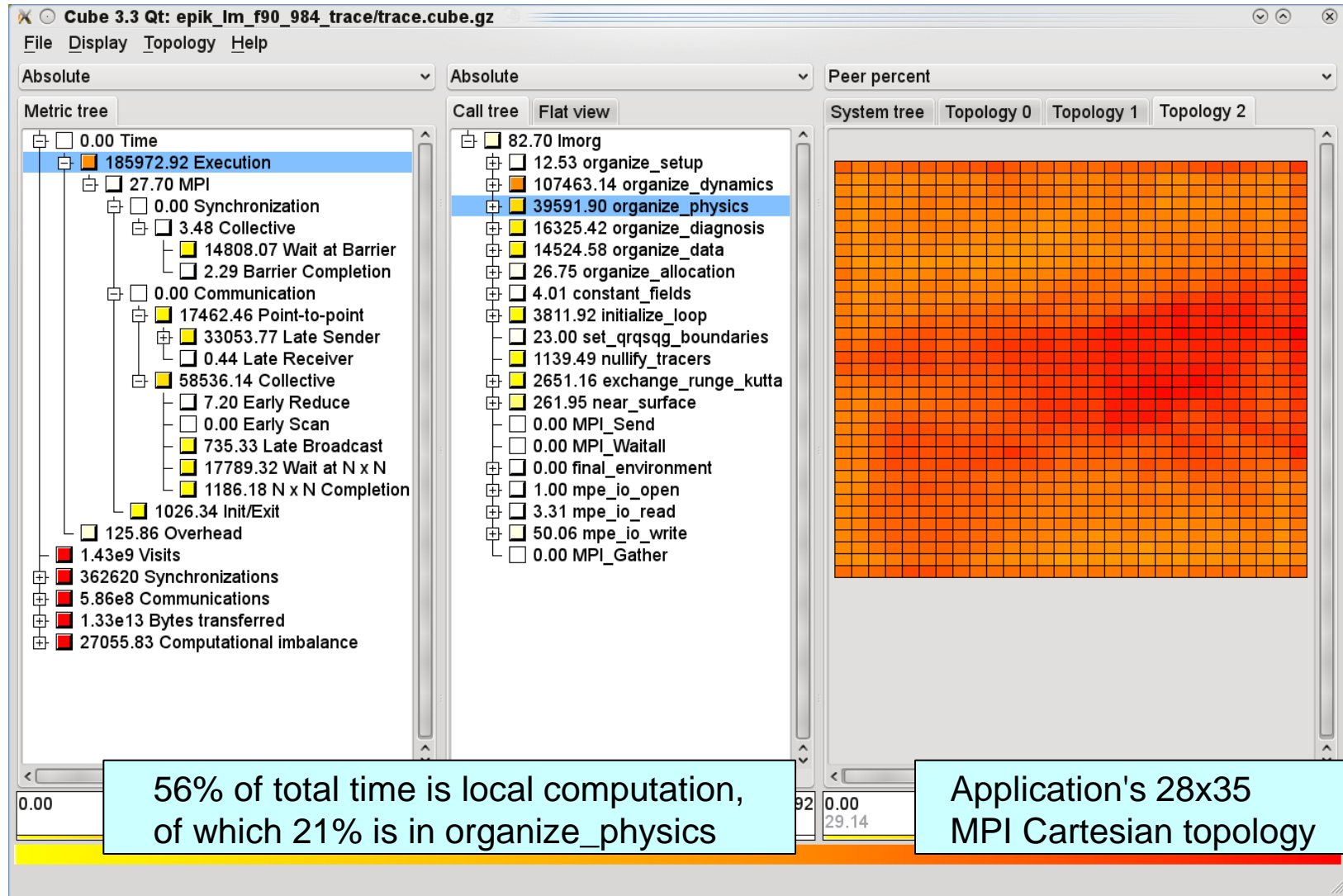
- 28x35 compute grid + 4 dedicated I/O processes
- Used 41 Opteron compute nodes each with 24 cores
- Scalasca trace measurement with 19 of 178 routines filtered
- 44GB trace written in 23s and analyzed in 82s

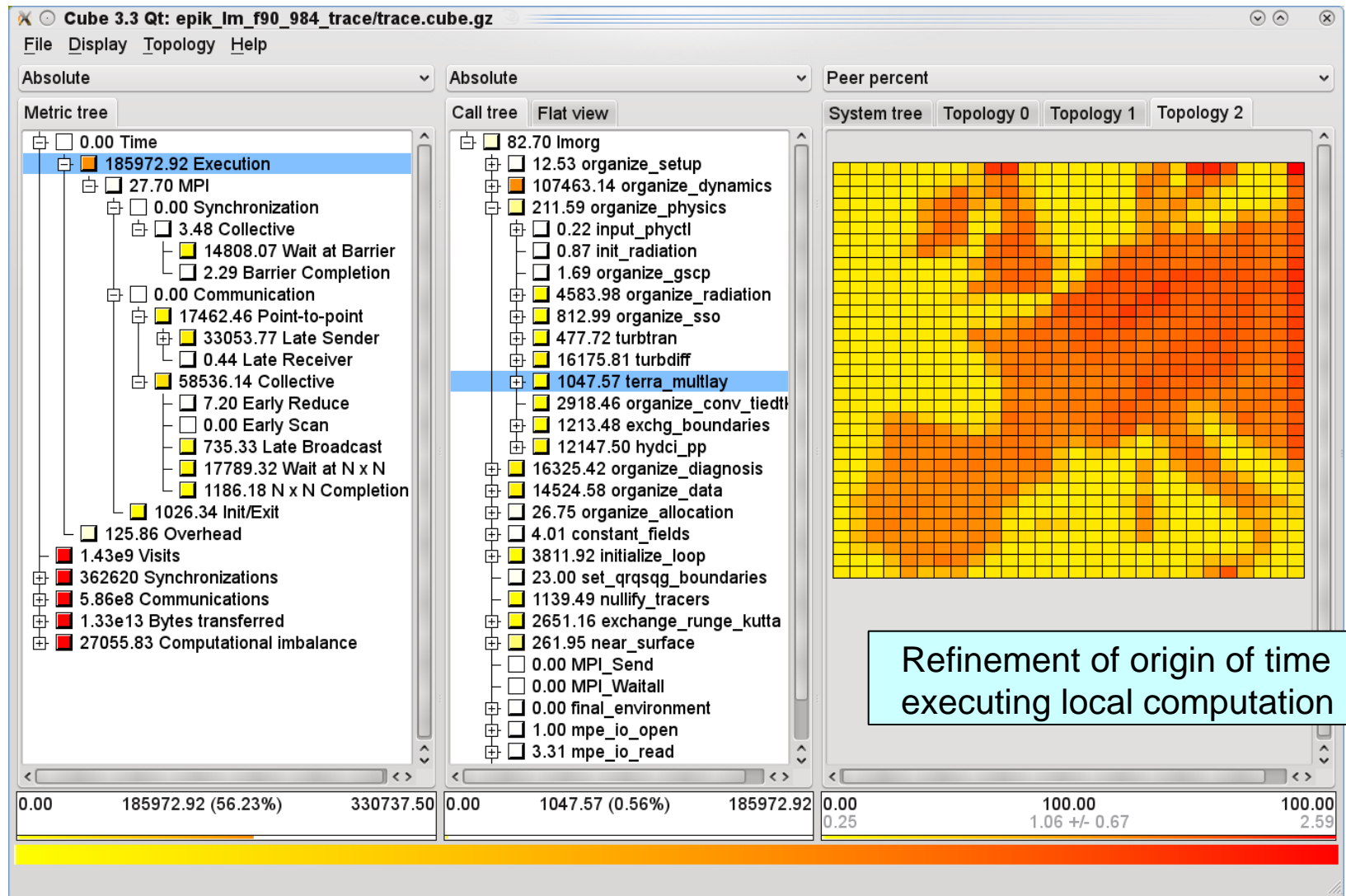
Courtesy of Oliver Fuhrer (MeteoSwiss) & CSCS

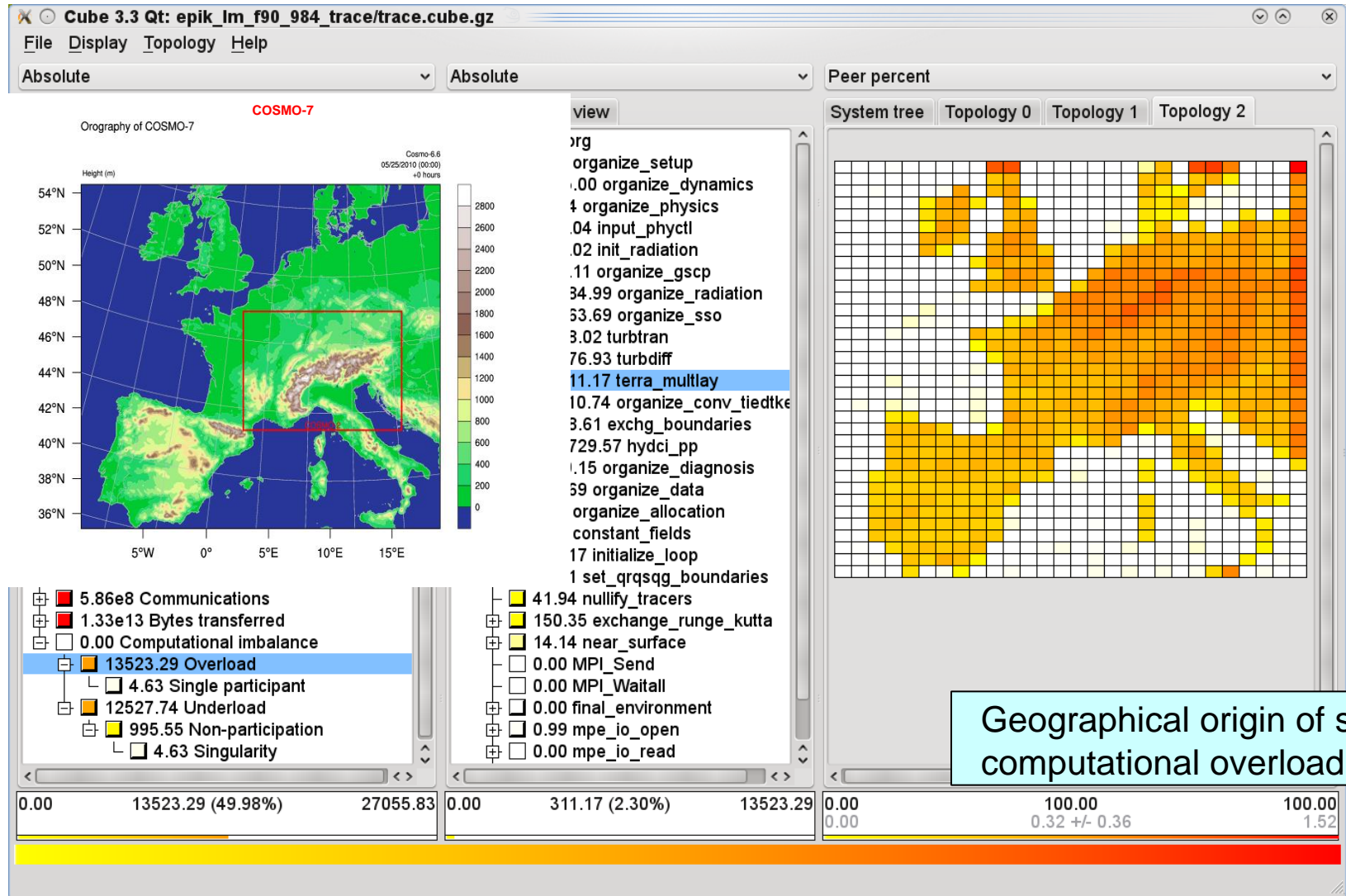
COSMO/XE6 Physics Computation Time

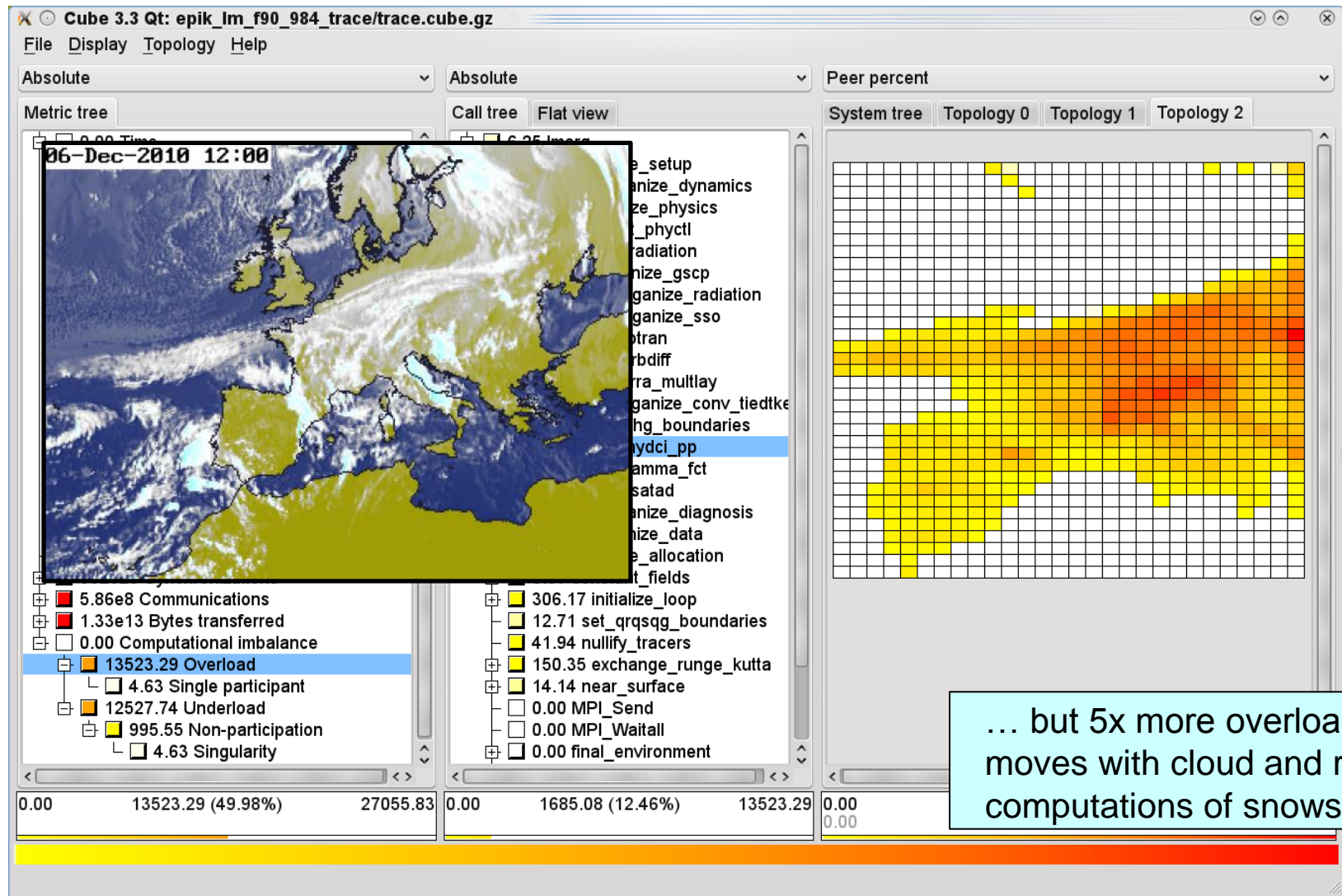


COSMO/XE6 Physics Computation Time

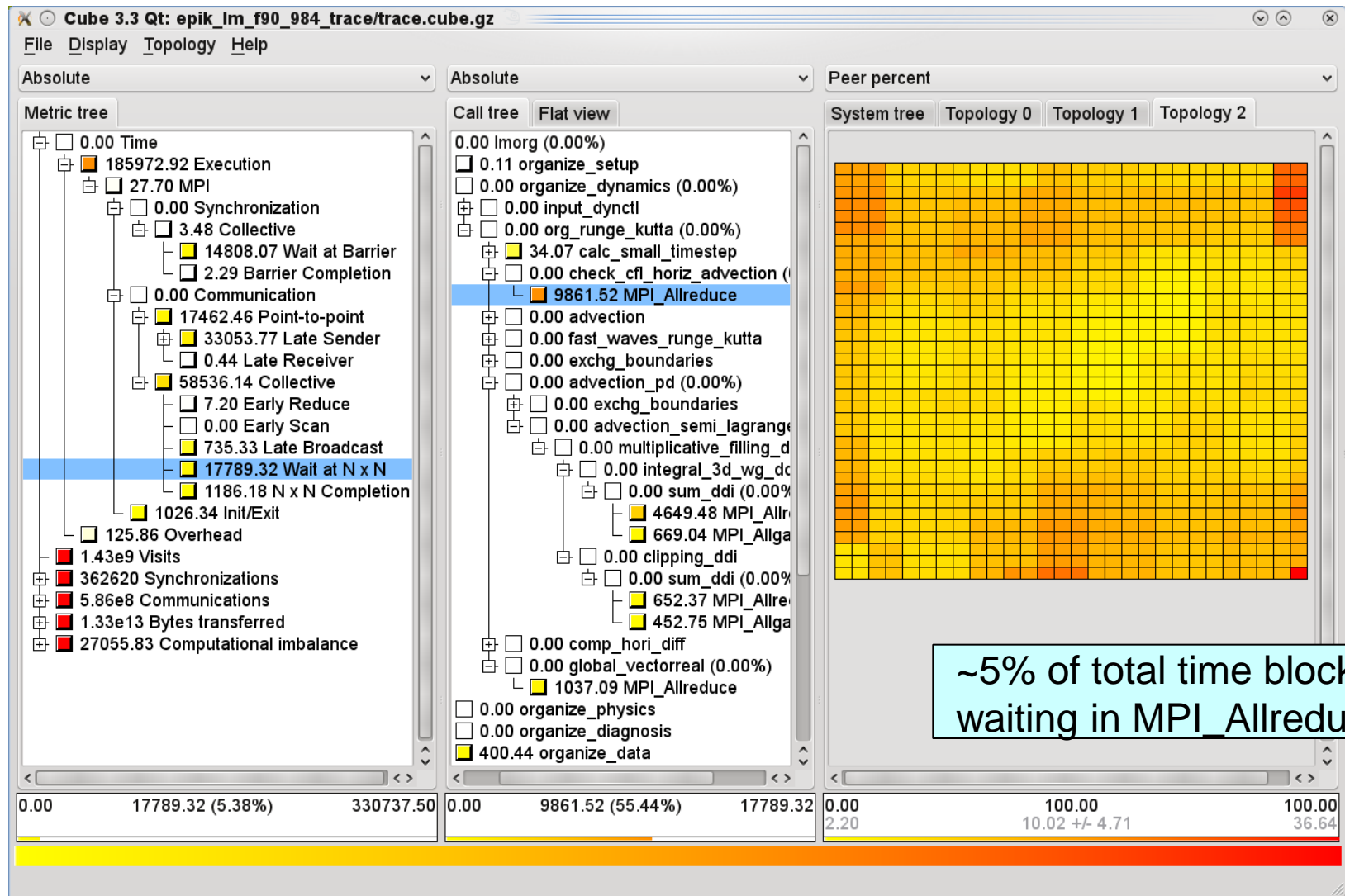




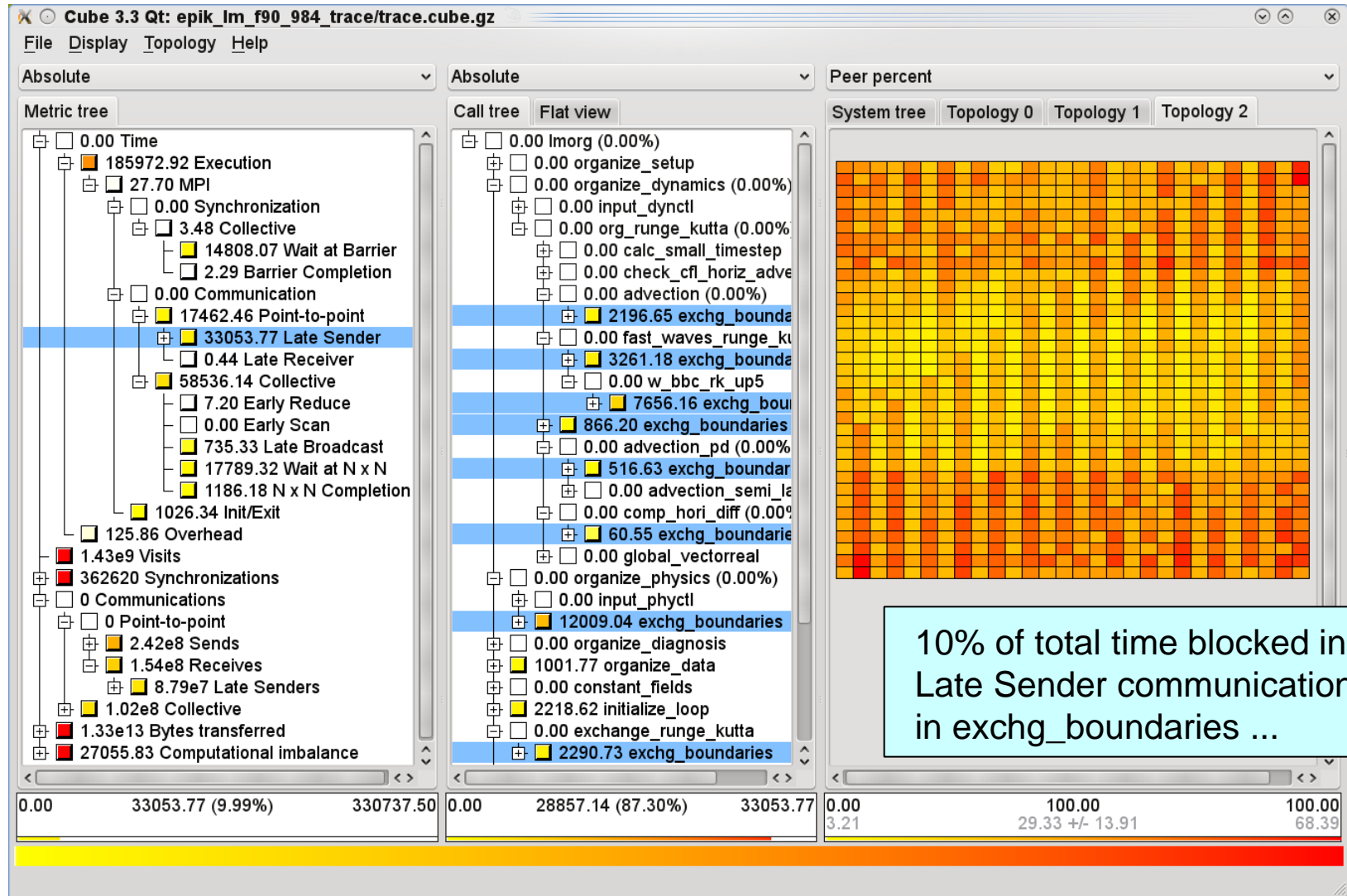


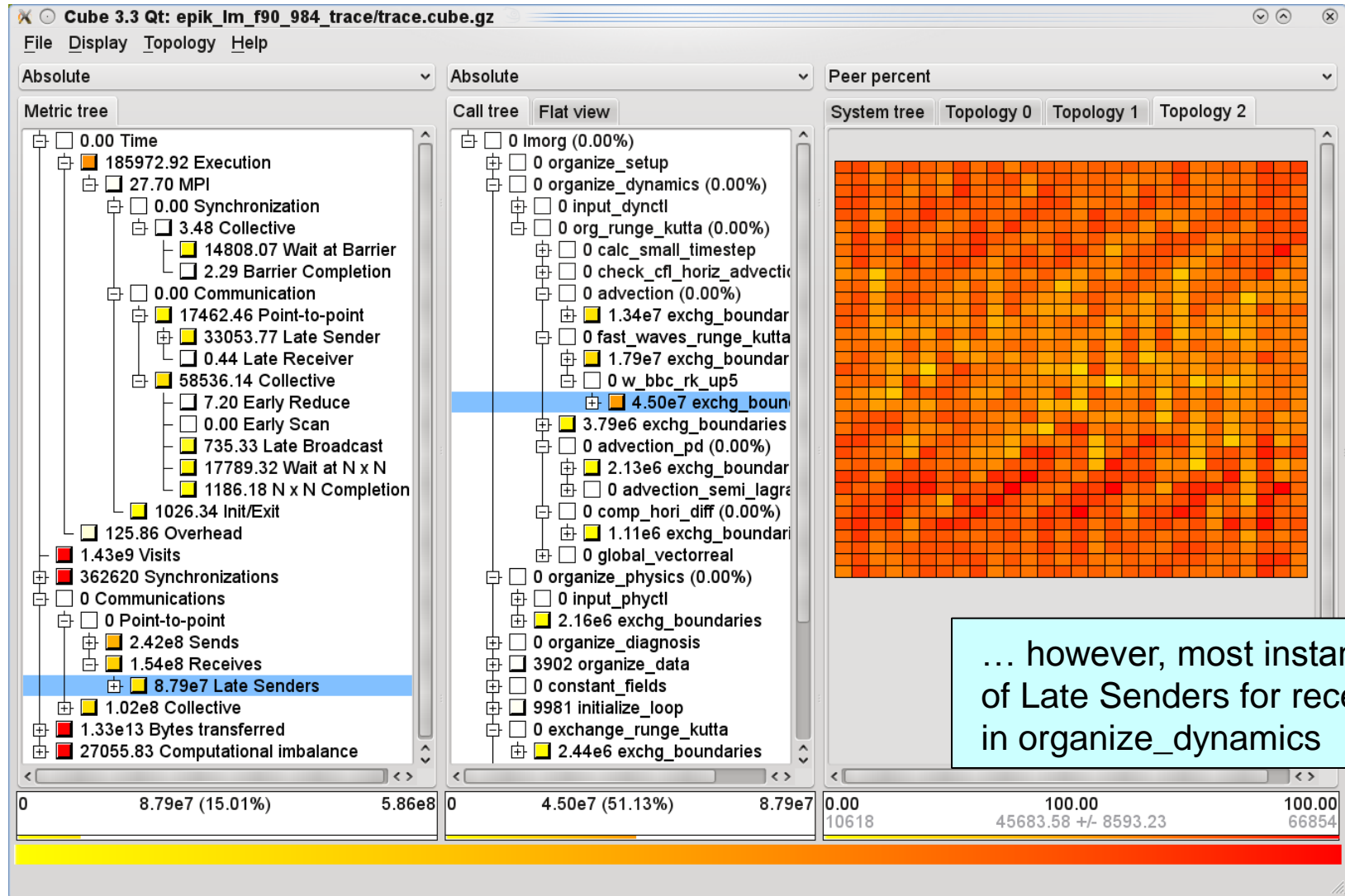


COSMO/XE6 Collective Wait at N x N Time



COSMO/XE6 Late Sender Waiting Time





Outline

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

Summary

You've been introduced to a variety of tools

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