Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

Debuggers and Performance Tools

February 2014 | Markus Geimer, Alexandre Strube

Outline

Make it work,
make it right,
make it fast.

Kent Beck

Local module setup

Debuggers:

- STAT
- TotalView

Performance Tools:

- Scalasca
- Score-P
- TAU
- Vampir
- HPCToolkit

UNITE

- UNiform Integrated Tool Environment
- Standardizes tool access and documentation
  - Currently in use at JSC, RWTH, ZIH
- Based on "`module`" command
  - Standardized tool and version identification
    - <tool>/<version>-<special>
    - <special>: optional indicator if tool is specific for a MPI library, compiler, or 32/64 bit mode
- **Tools only visible after**
  - **`module load UNITE`** **# once per session**
- Basic usage and pointer to tool documentation via
  - `module help <tool>`

Example

```
% module load UNITE
UNITE loaded
% module help scalasca
Module Specific Help for scalasca/1.4.3:

Scalasca: Scalable Performance Analysis of Large-Scale
          Parallel Applications
Version 1.4.3

Basic usage:
1. Instrument application with skin
2. Collect & analyze execution measurement with scan
3. Examine analysis results with square

For more information:
- See ${SCALASCA_ROOT}/doc/manuals/QuickReference.pdf
  or type "scalasca -h"
- http://www.scalasca.org
- mailto:scalasca@fz-juelich.de
```

Documentation

- Use “`module avail`” to check latest status

- Websites
  - http://www.fz-juelich.de/ias/jsc/juqueen/
    - User Info
      - Debugging
      - Performance Analysis ( )
  - http://www.vi-hps.org/training/material/
    - Performance Tools LiveDVD image
    - Links to tool websites and documentation
    - Tutorial slides

Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

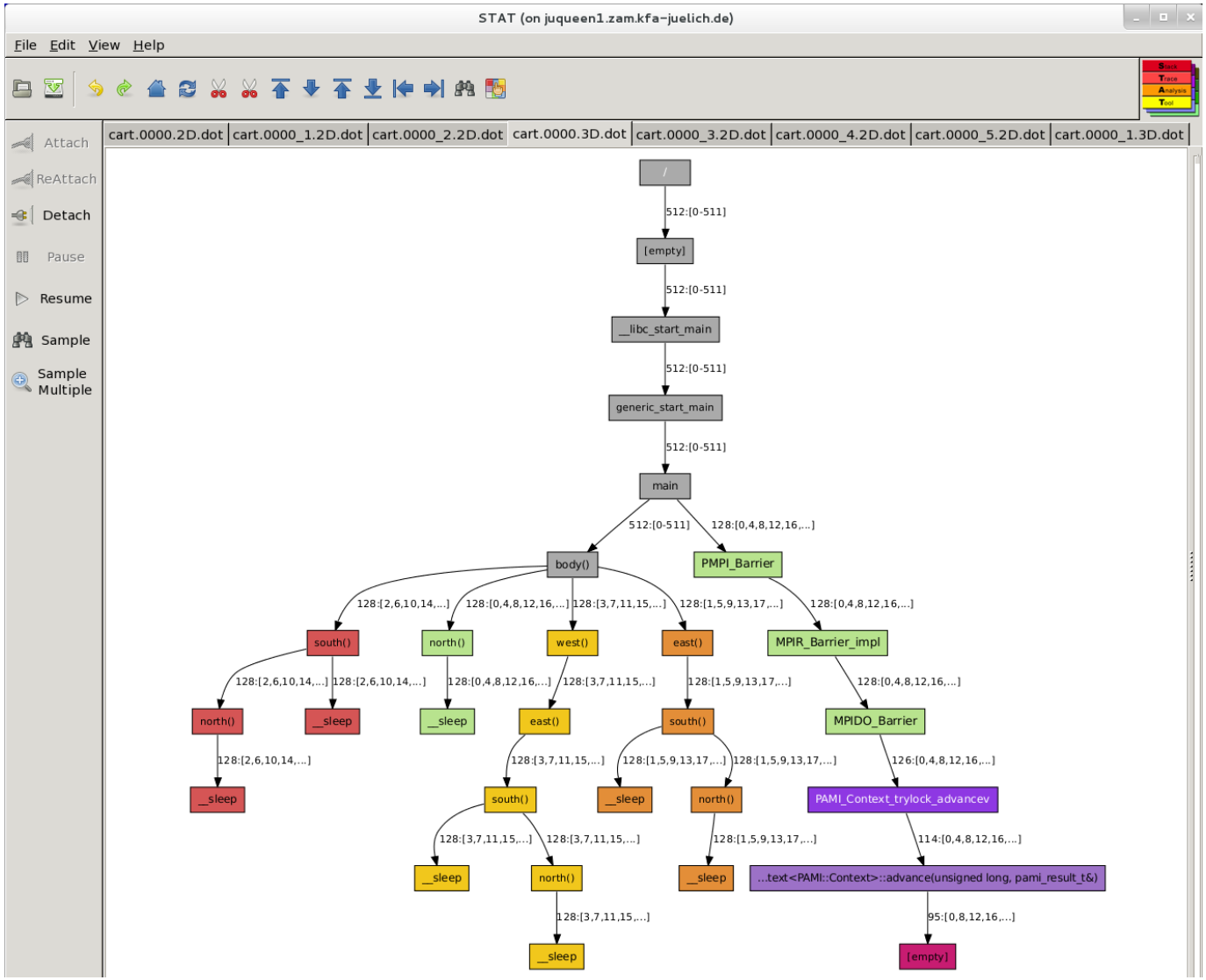
Debugging on JUQUEEN

February 2014 | Alexandre Strube

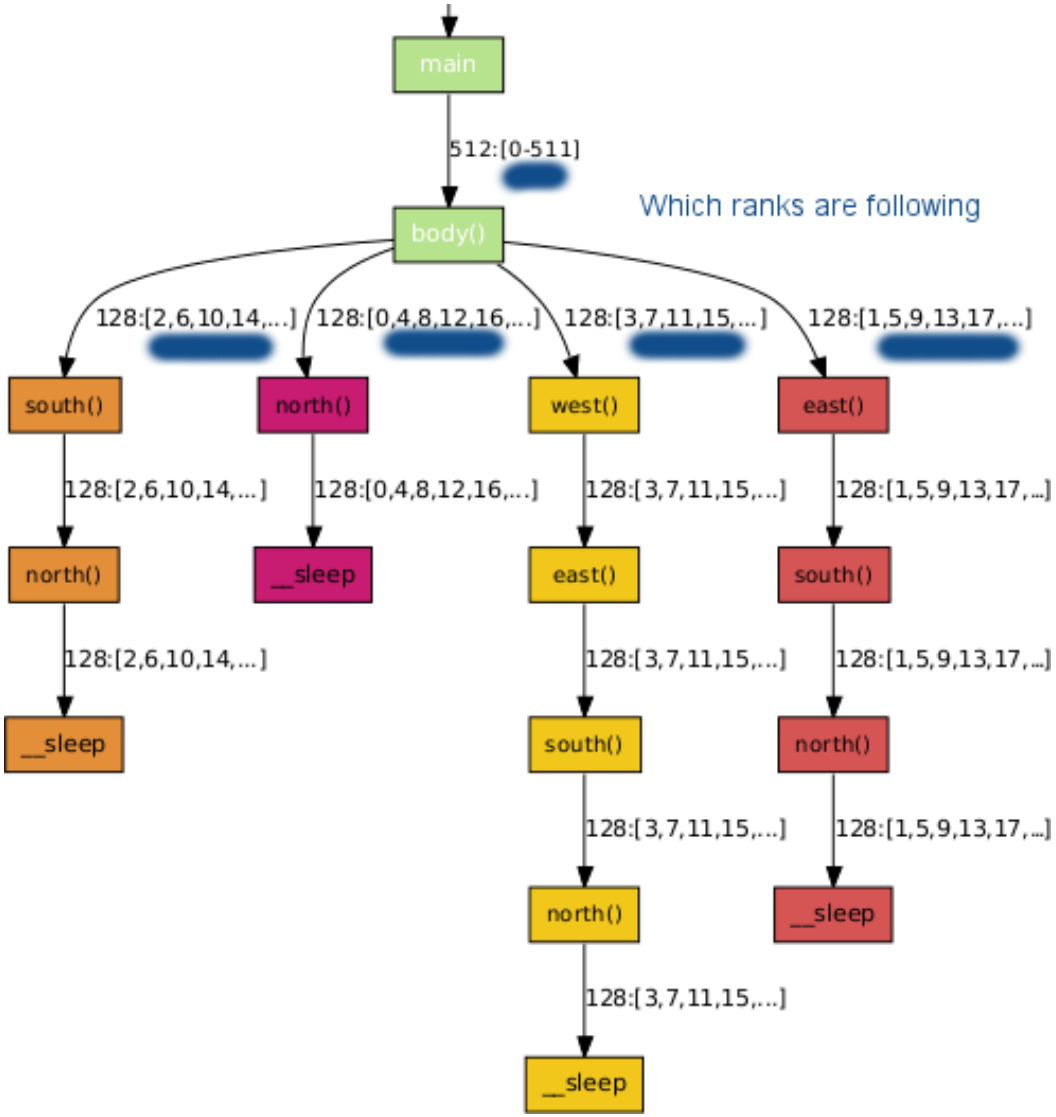
STAT: Stack Trace Analysis Tool

- Very lightweight helper tool
- Shows merged call tree of whole program
- Useful to detect deadlocks
- Scales to millions of processes
  - http://www.hpcwire.com/hpcwire/2012-12-03/bug_repellent_for_supercomputers_proves_effective.html
- Pinpoint individual problems
- **NOT** a real/full debugger
- http://www.paradyn.org/STAT/STAT.html

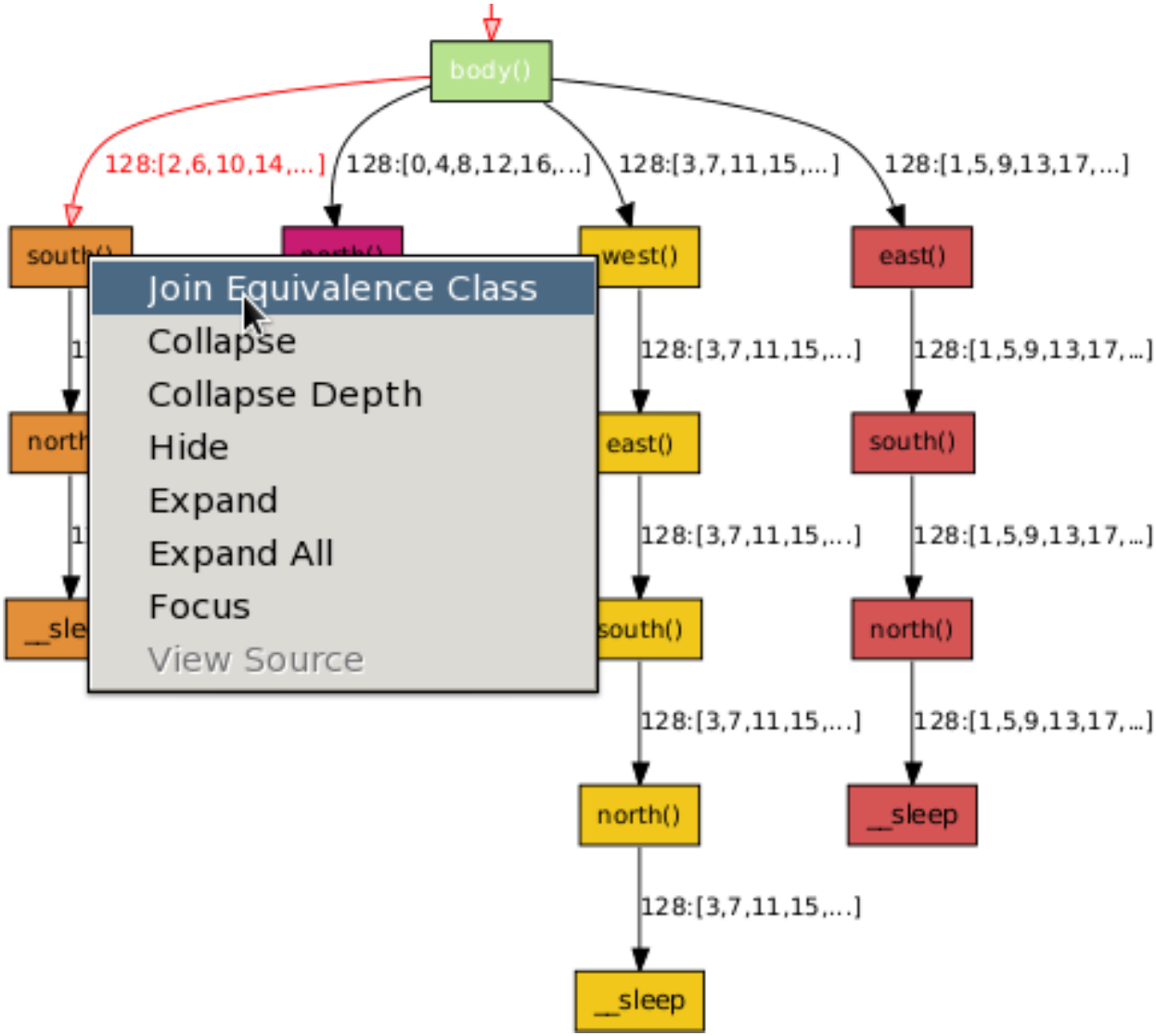
STAT: Main Window



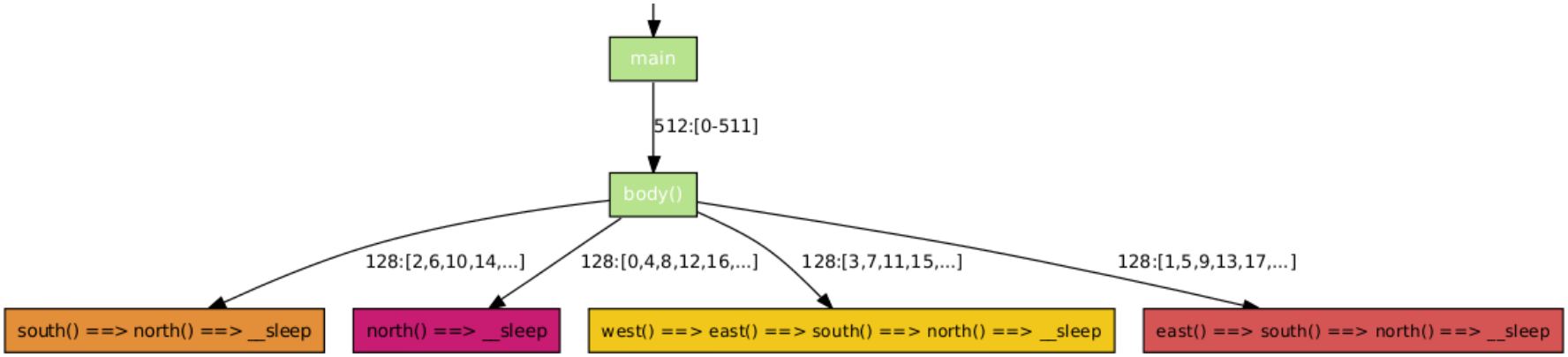
STAT: Zoom


main
512:[0-511]
Which ranks are following
body()
128:[2,6,10,14,...]
128:[0,4,8,12,16,...]
128:[3,7,11,15,...]
128:[1,5,9,13,17,...]
south()
north()
west()
east()
128:[2,6,10,14,...]
128:[0,4,8,12,16,...]
128:[3,7,11,15,...]
128:[1,5,9,13,17,...]
north()
__sleep
east()
south()
128:[2,6,10,14,...]
128:[3,7,11,15,...]
128:[1,5,9,13,17,...]
__sleep
south()
north()
128:[3,7,11,15,...]
128:[1,5,9,13,17,...]
north()
__sleep
128:[3,7,11,15,...]
__sleep

STAT: Equivalence Classes



STAT: Equivalence Classes (cont.)



STAT: Recipe

- Compile and link your program with debug option: `-g`
- Load modules

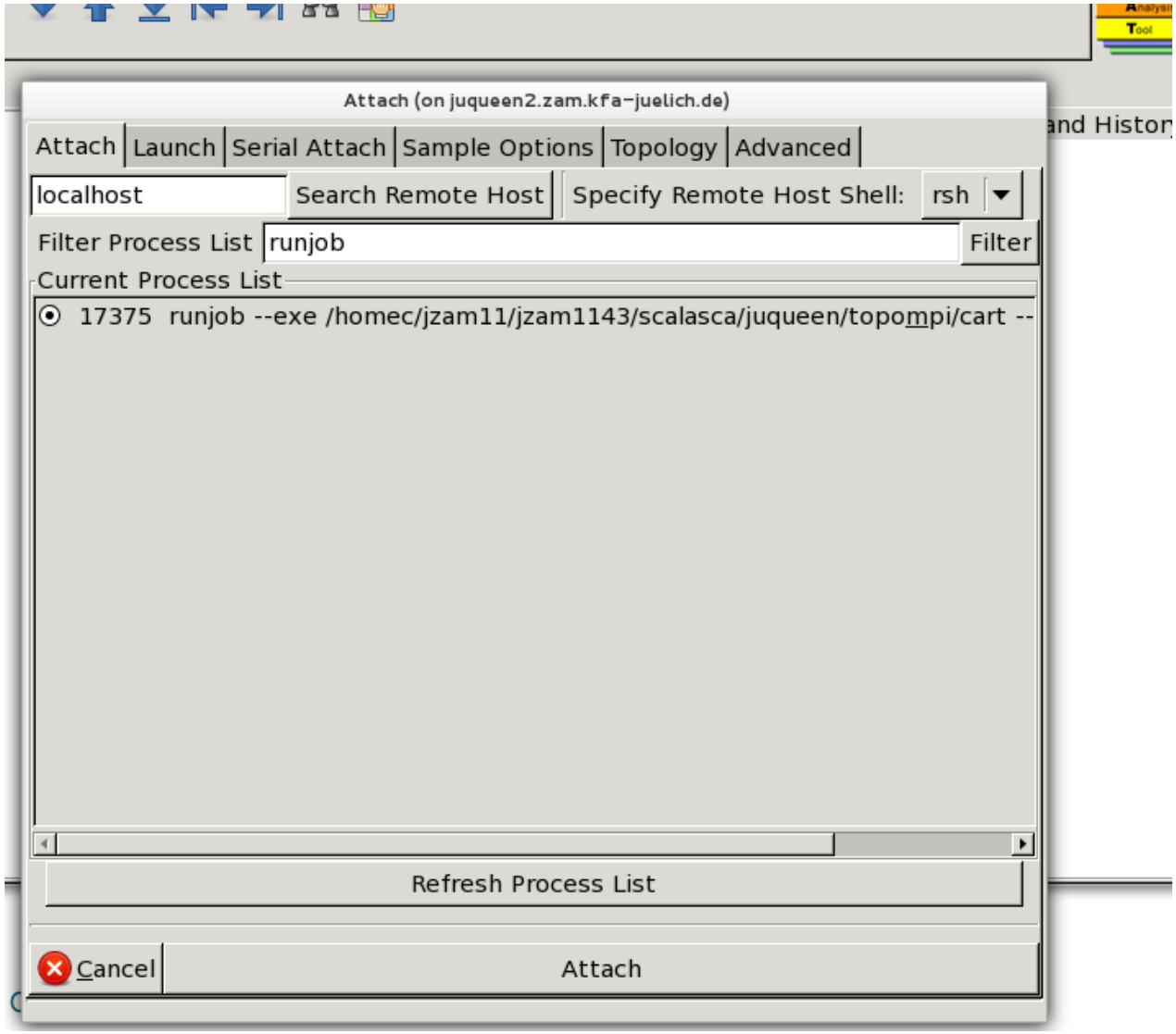
```
% ssh -X user@juqueen
[...]
juqueen% module load UNITE stat
UNITE loaded
stat/2.0 loaded
juqueen% STATGUI
```

- Submit job and attach to it from GUI
- **CAVEAT:**
  - Job needs to be started by login node where GUI is running
  - Add the following entry to the submission script

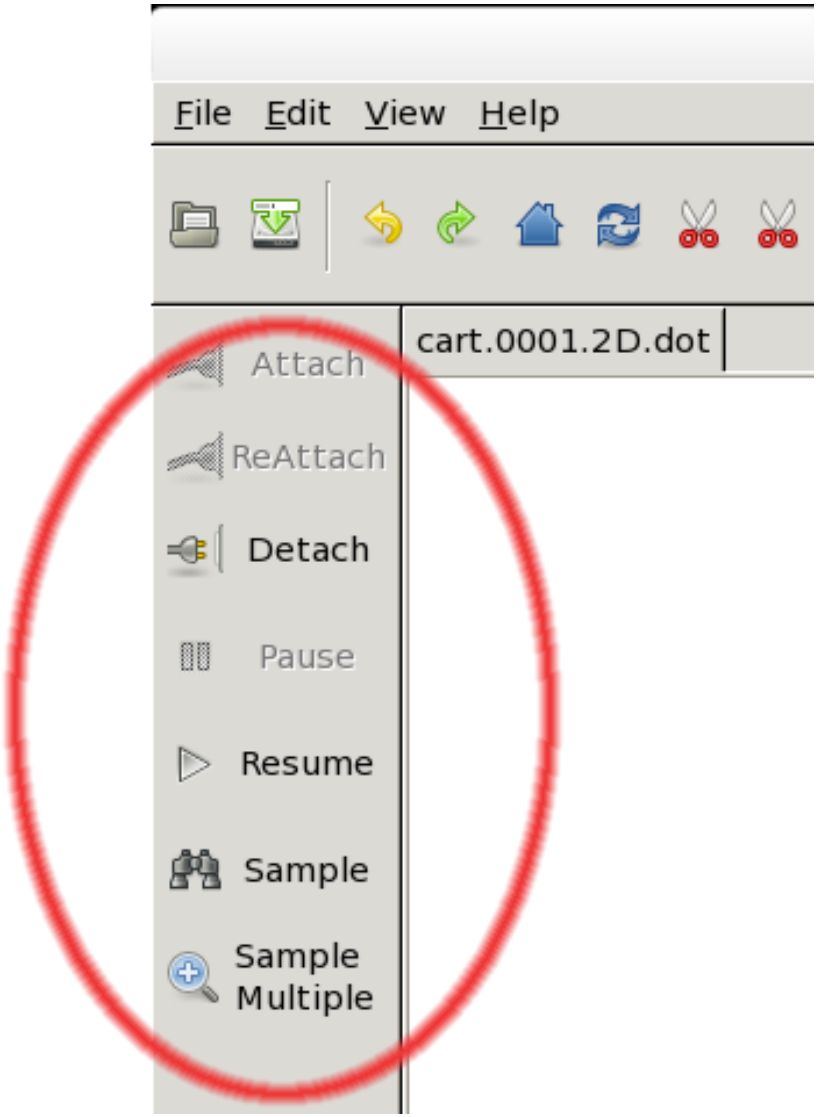
    `#@ requirements = (Machine == "juqueen<n>")`

    with <n> being the login node id

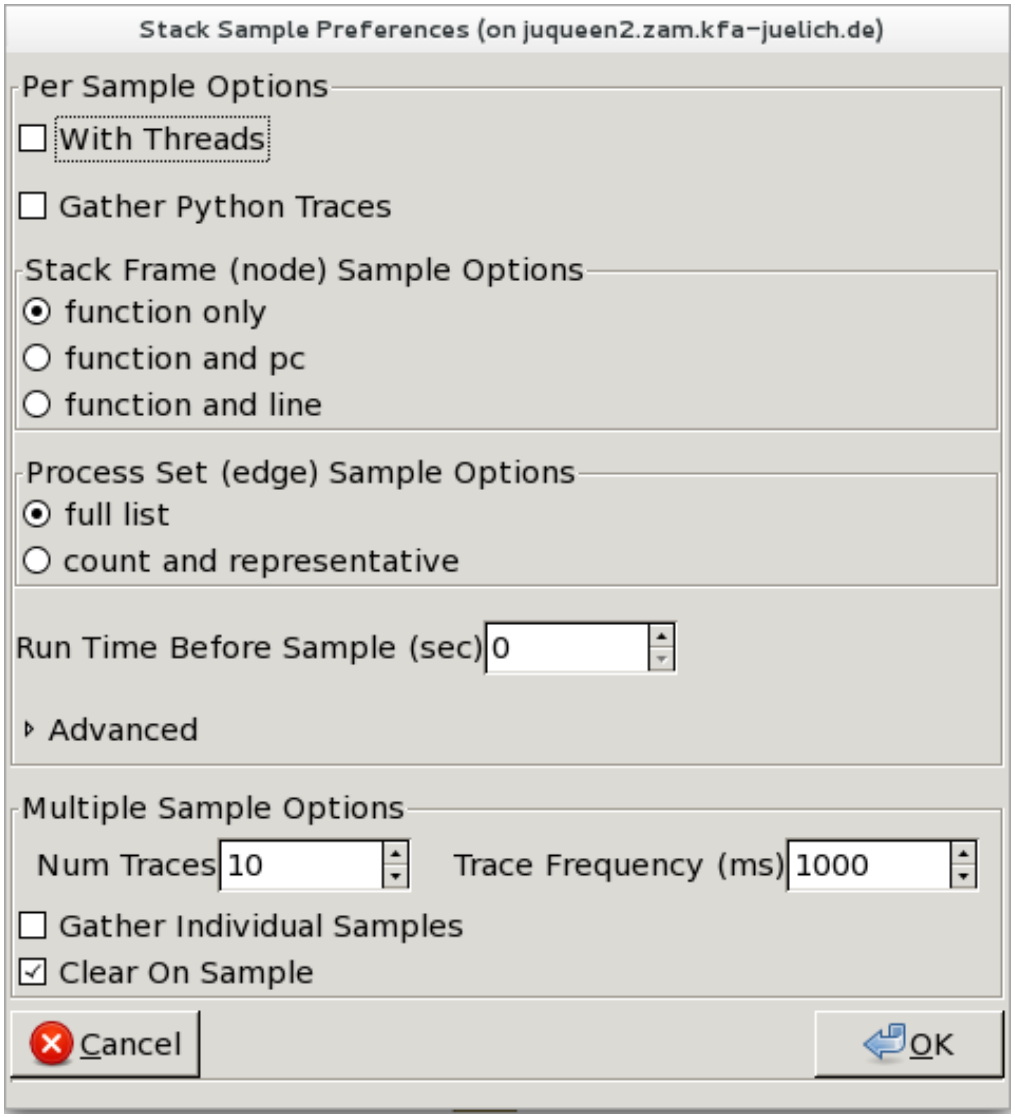
STAT: Attaching to a Job



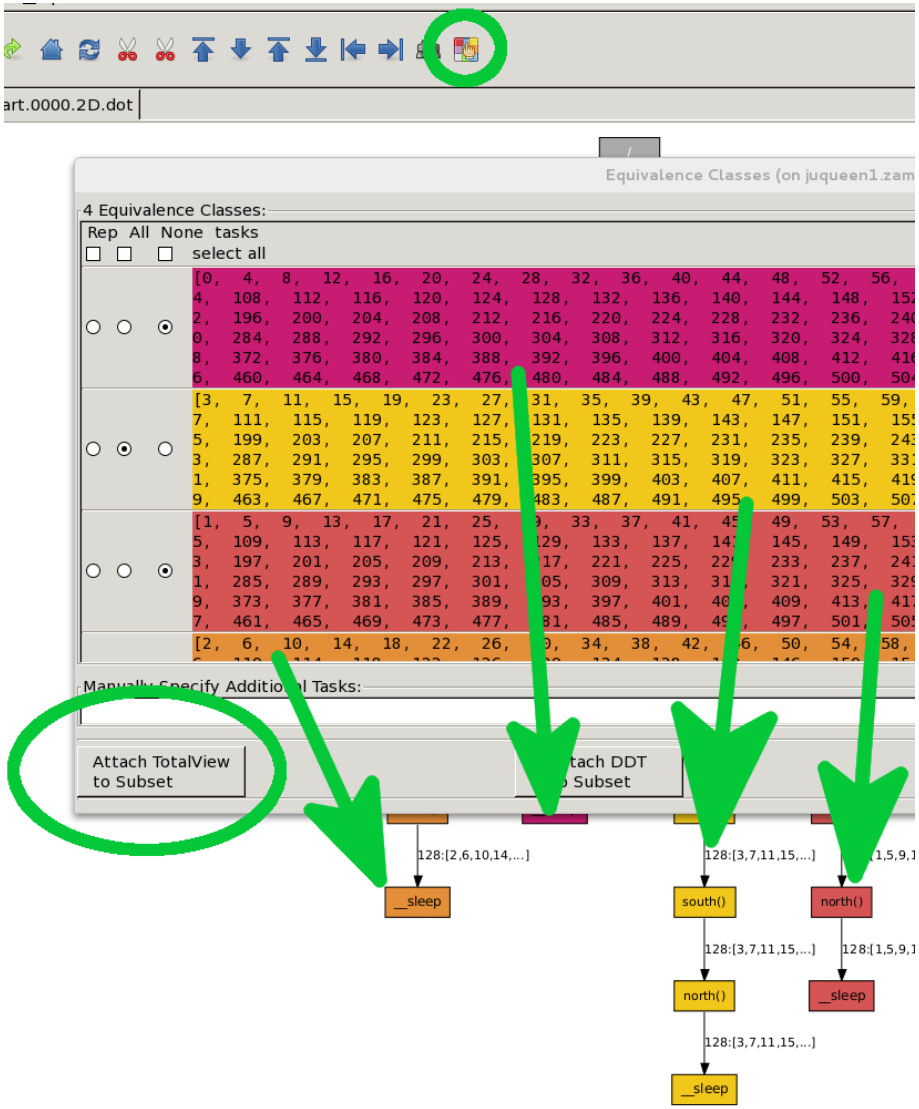
STAT: Collecting Multiple Samples



STAT: Collecting Multiple Samples (cont.)



STAT: Attach TotalView to Subset



Parallel Debugger

- UNIX Symbolic Debugger
  for C, C++, f77, f90, PGI HPF, assembler programs
- “Standard” debugger
- Special, non-traditional features
  - Multi-process and multi-threaded
  - C++ support (templates, inheritance, inline functions)
  - F90 support (user types, pointers, modules)
  - 1D + 2D Array Data visualization
  - Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, pthreads)
  - Scripting and batch debugging
  - Memory Debugging
- http://www.roguewave.com

TotalView: Recipe

- Compile and link your program with debug option: `-g`
- Use absolute paths for source code info: `-qfullpath`
- In case of optimized codes (XL), keep function call parameters: `-qkeepparm`
- Load modules

```
% ssh -X user@juqueen
[...]
juqueen% module load UNITE totalview
UNITE loaded
totalview/8.12.0-1 loaded

juqueen% mpixlcxx hello.cpp -qfullpath -qkeepparm -g -o helloworld
juqueen%
```

TotalView: Interactive Startup

- Interactively: call the `lltv` script
  - Creates a LoadLeveler batch script with required TotalView parameters
- If user cancels the script, it cancels the debugging job (does not eat your computing quota)
- DON'T attach to all ranks! This will be VERY slow.

TotalView: `lltv` Launch Script

```
% lltv -n <nodes> : -default_parallel_attach_subset=\
<rank-range> runjob -a --exe <program> -p <num>
```

- Starts <program> with <nodes> and <num> processes per node, attaches to <rank-range>:
  - Rank: that rank only
  - RankX-RankZ: all ranks, both inclusive
  - RankX-RankZ:stride every *stride*th between RankX and RankZ
- Example:

```
% lltv -n 2 : -default_parallel_attach_subset=\
2-6 runjob -a --exe helloworld -p 64

Creating LoadLeveler Job
Submitting LoadLeveler Interactive Job for Totalview
 Wait for job juqueen1c1.32768.0 to be started:..........
```

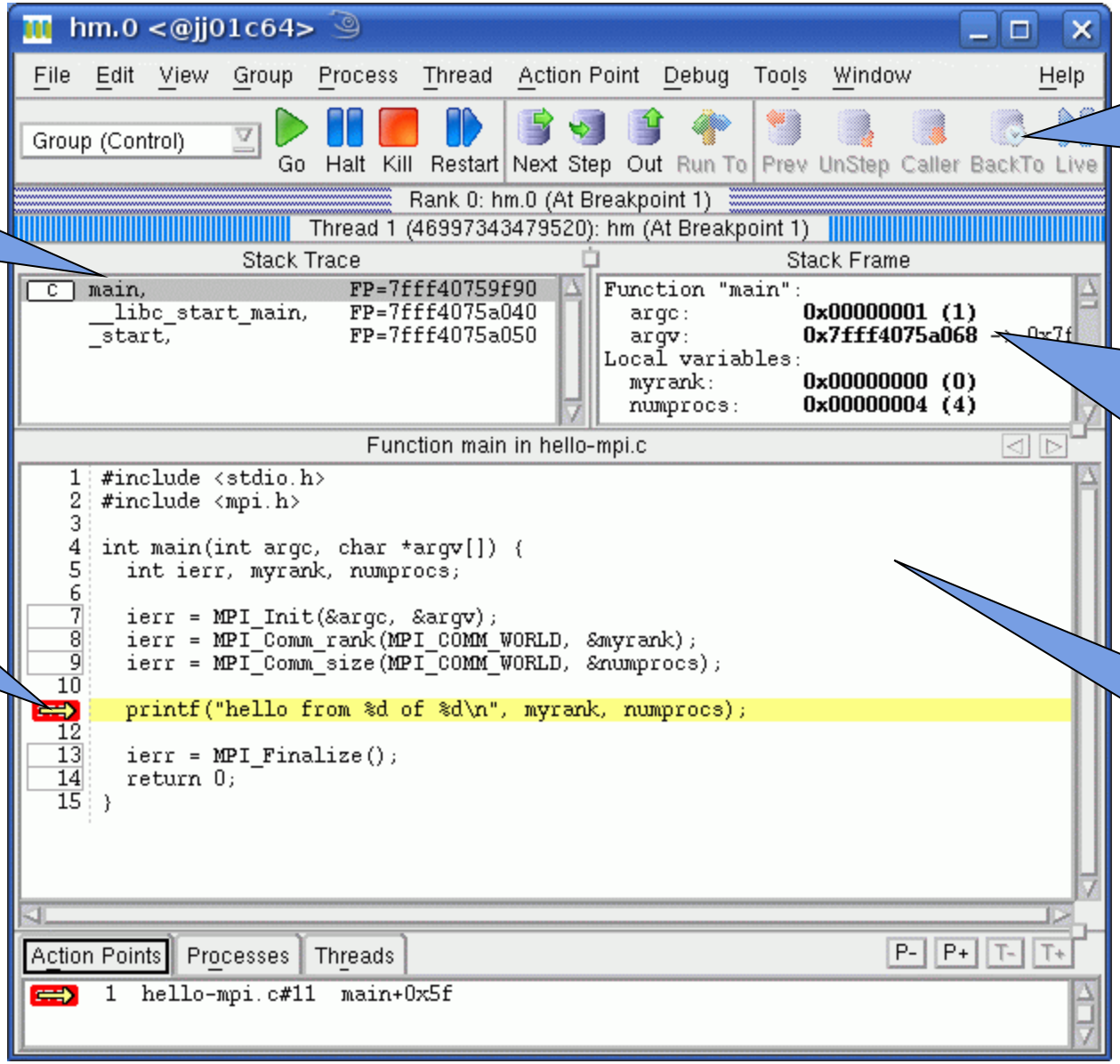
TotalView: Execution Recipe

- TotalView tries to debug “runjob” and shows no source code
  - Ignore it and press “GO”

- After some seconds, TotalView will detect parallel execution and ask if it should stop. Yes, it should stop.
- To find the correct point file/function to debug, use the “File-Open” command.
- Set your breakpoints, and press “GO” again. Debugging session will then start.

- To see a variable’s contents, double click on it in the source.

TotalView: Main Window



Stack trace

Toolbar for common options

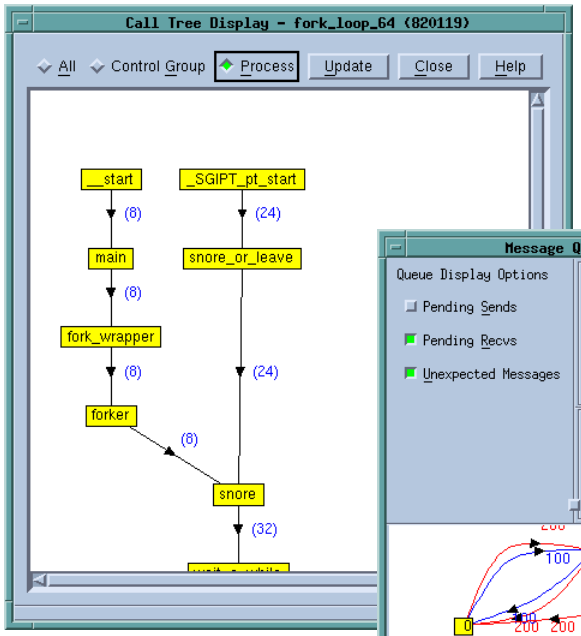
Local variables for selected stack frame

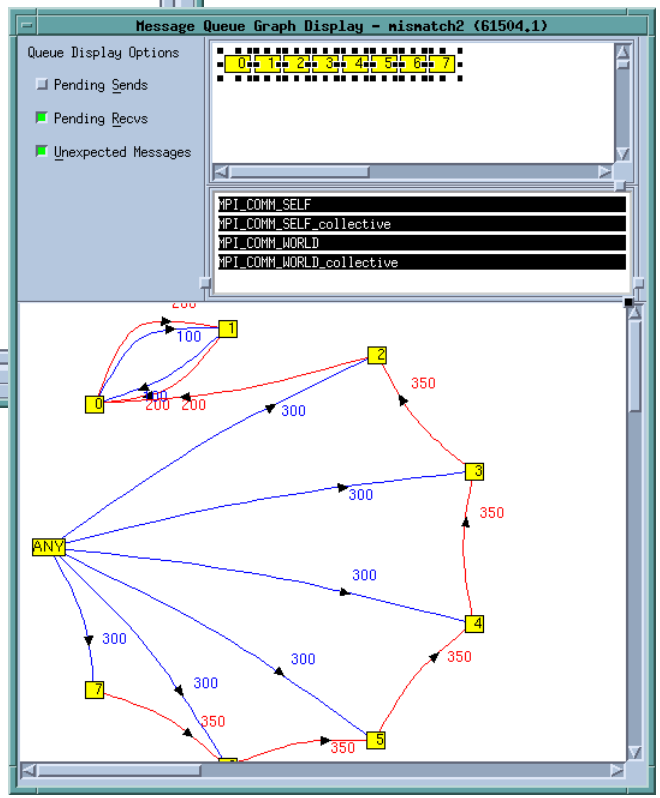
Break points

Source code window

TotalView: Tools Menu

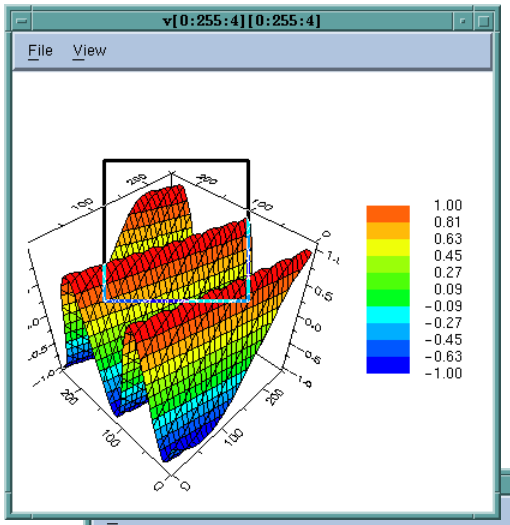
- Call Graph

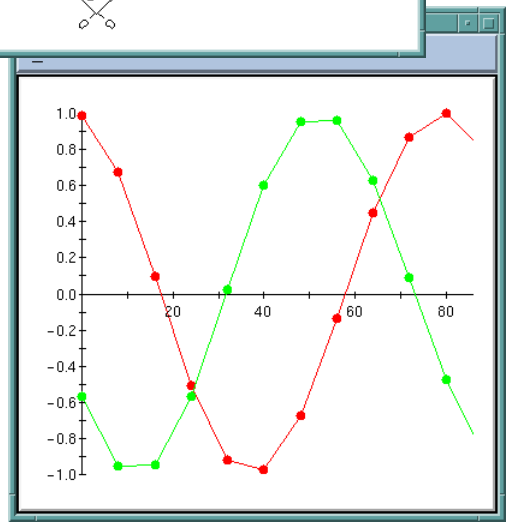




- Message queue graph

- Data visualization





Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

Performance Analysis Tools on JUQUEEN

February 2014 | Markus Geimer

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

Remark: No Single Solution is Sufficient!

☞ *A combination of different methods, tools and techniques is typically needed!*

- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...

Critical Issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behavior
    - 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*

- **Sc**alable **A**nalysis of **La**rge **Sc**ale **A**pplications
- Approach
  - **Instrument** C, C++, and Fortran parallel applications
    - Based on MPI, OpenMP, SHMEM, or hybrid
  - Option 1: **scalable call-path profiling**
  - Option 2: **scalable event trace analysis**
    - **Collect** event traces
    - **Search** trace for event patterns representing inefficiencies
    - **Categorize and rank** inefficiencies found
- http://www.scalasca.org

What is the Key Bottleneck?

- Generate flat MPI profile using Scalasca
  - Only requires re-linking
  - Low runtime overhead

- Provides detailed information on MPI usage
  - How much time is spent in which operation?
  - How often is each operation called?
  - How much data was transferred?

- Limitations:
  - Computation on non-master threads and outside of MPI_Init/MPI_Finalize scope ignored

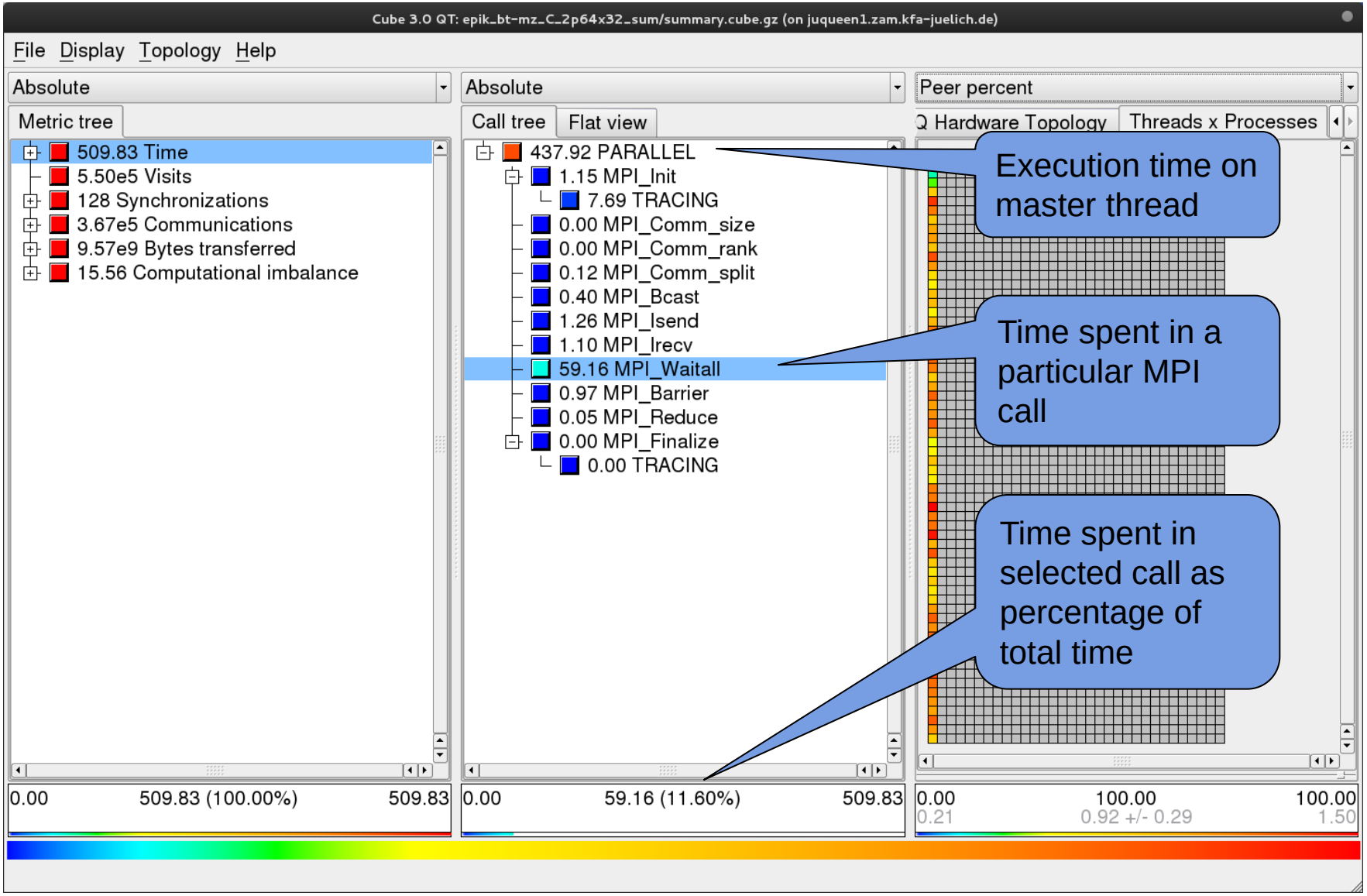
Flat MPI Profile: Recipe

1. Prefix your ***link command*** with
   "`scalasca -instrument -comp=none`"
2. Prefix your MPI ***launch command*** with
   "`scalasca -analyze`"
3. After execution, examine analysis results using
   "`scalasca -examine epik_<title>`"

Flat MPI Profile: Example

```
% module load UNITE scalasca
% mpixlf90 -O3 -qsmp=omp -c foo.f90
% mpixlf90 -O3 -qsmp=omp -c bar.f90
% scalasca -instrument -comp=none \
    mpixlf90 -O3 –qsmp=omp -o myprog foo.o bar.o

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

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

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

% scalasca -examine epik_myprog_Ppnxt_sum
```

Flat MPI Profile: Example (cont.)



Where is the Key Bottleneck?

- Generate call-path profile using Scalasca
  - Requires re-compilation
  - Runtime overhead depends on application characteristics
  - Typically needs some care setting up a good measurement configuration
    - Filtering
    - Selective instrumentation

- Option 1 (recommended):
  Automatic compiler-based instrumentation

- Option 2:
  Manual instrumentation of interesting phases, routines, loops

Call-path Profile: Recipe

1. Prefix your ***compile & link commands*** with
   "`scalasca -instrument`"
2. Prefix your MPI ***launch command*** with
   "`scalasca -analyze`"
3. After execution, compare overall runtime with uninstrumented run to determine overhead
4. If overhead is too high
   1. Score measurement using
      "`scalasca -examine -s epik_<title>`"
   2. Prepare filter file
   3. Re-run measurement with filter applied using prefix
      "`scalasca –analyze –f <filter_file>`"
5. After execution, examine analysis results using
   "`scalasca -examine epik_<title>`"

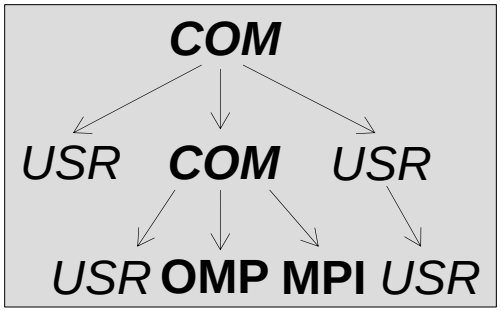
Call-path Profile: Example

```
% module load UNITE scalasca
% scalasca -instrument mpixlf90 -O3 -qsmp=omp -c foo.f90
% scalasca -instrument mpixlf90 -O3 -qsmp=omp -c bar.f90
% scalasca -instrument \
    mpixlf90 -O3 –qsmp=omp -o myprog foo.o bar.o



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

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

Call-path Profile: Example (cont.)

```
% scalasca -examine -s epik_myprog_Ppnxt_sum
cube3_score -r ./epik_myprog_Ppnxt_sum/summary.cube
Reading ./epik_myprog_Ppnxt_sum/summary.cube... done.
Est. aggregate size of event trace (total_tbc): 160,338,400,040 bytes
Est. size of largest thread trace (max_tbc):        133,910,372 bytes
(When tracing set ELG_BUFFER_SIZE to avoid intermediate flushes or
 reduce requirements using filter file listing names of USR regions.)

INFO: Score report written to ./epik_myprog_Ppnxt_sum/epik.score
```

- Region/call-path classification
  - MPI (pure MPI library functions)
  - OMP (pure OpenMP functions/regions)
  - USR (user-level source local computation
  - COM (“combined” USR + OpeMP/MPI)
  - ANY/ALL (aggregate of all region types)



Call-path Profile: Example (cont.)

```
% less epik_myprog_Ppnxt_sum/epik.score
flt type    max_tbc       time        %  region
     ANY  133910372   49656.65   100.00  (summary) ALL
     MPI     483104    6575.16    13.24  (summary) MPI
     OMP    1620780   27541.42    55.46  (summary) OMP
     COM     343320    4442.51     8.95  (summary) COM
     USR  131511360   10628.25    21.40  (summary) USR

     USR   42219648    4664.19     9.39  binvcrhs
     USR   42219648    2994.22     6.03  matmul_sub
     USR   42219648    2519.26     5.07  matvec_sub
     USR    1891008     156.64     0.32  binvrhs
     USR    1891008     210.20     0.42  lhsinit
     USR    1033416      76.88     0.15  exact_solution
     MPI     201000      37.20     0.07  MPI_Isend
     MPI     184920      33.19     0.07  MPI_Irecv
     MPI      96480    6362.37     6.25  MPI_Waitall
     COM      96480    1330.77     2.68  copy_x_face
     COM      96480    1331.61     2.68  copy_y_face
     OMP      88440     533.11     1.07  !$omp parallel @foo.f90
[...]
```

Call-path Profile: Filtering

- In this example, the 6 most fequently called routines are of type USR (max_tbc is proportional to visit count)
- These routines contribute around 21% of total time
  - However, much of that is most likely measurement overhead for a few frequently-executed small routines

☞ Avoid measurements to reduce the overhead

☞ List routines to be filtered in simple text file

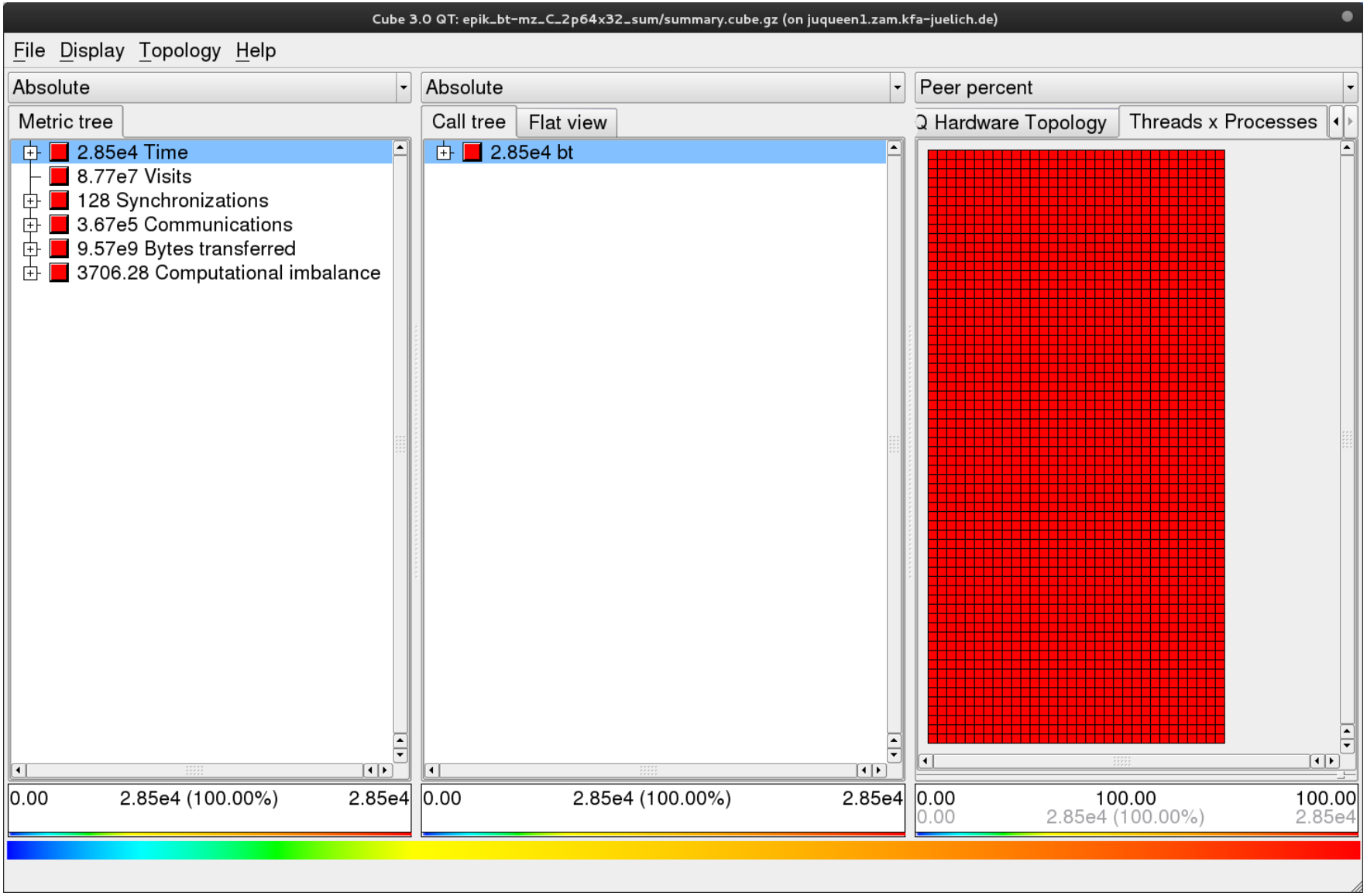
- Supports wildcards (shell globs)

```
% cat filter.txt
binvcrhs
matmul_sub
matvec_sub
binvrhs
lhsinit
exact_solution
```

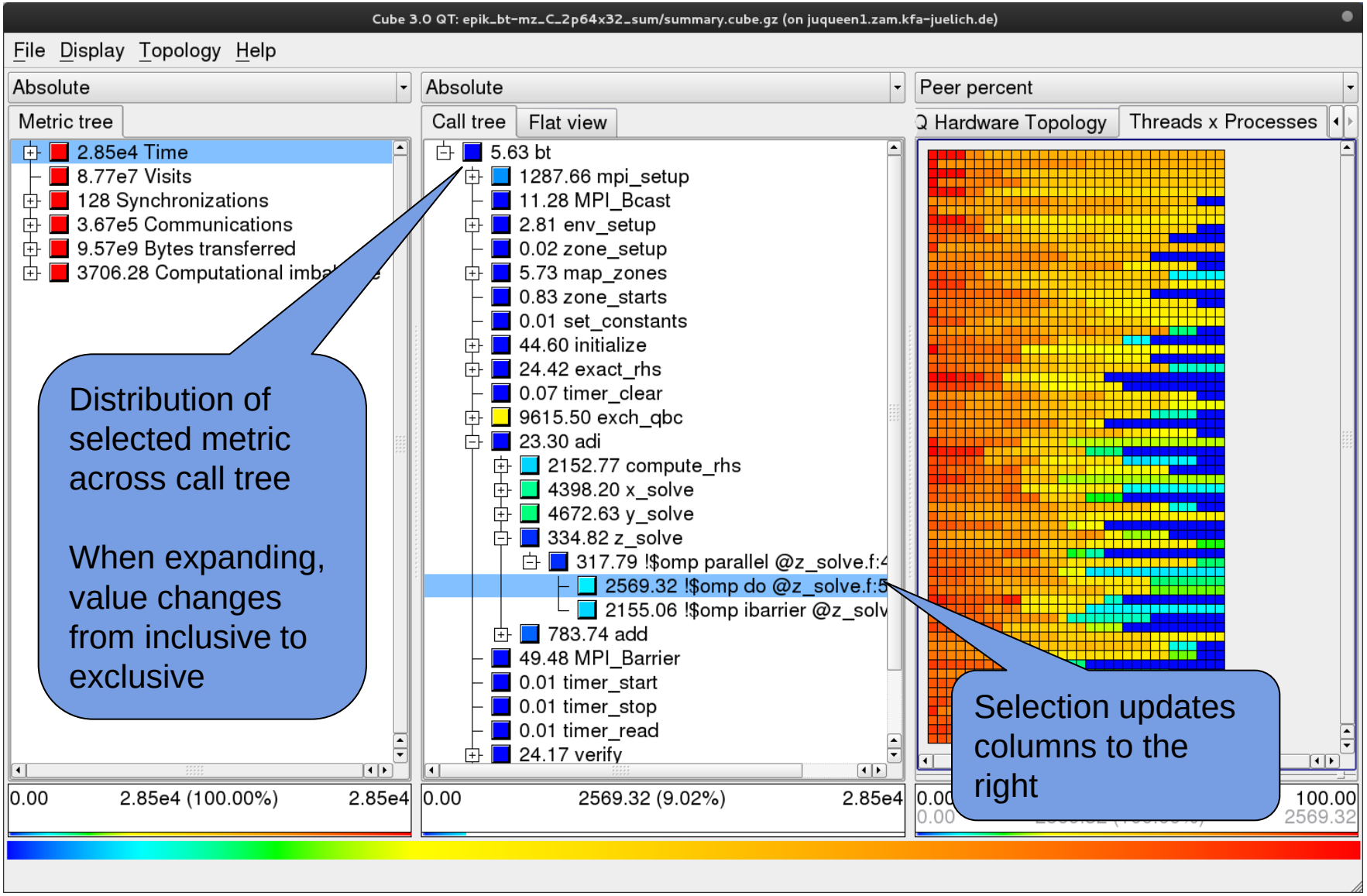
Call-path Profile: Example (cont.)

```
##  To verify effect of filter:

% scalasca -examine -s -f filter.txt \
    epik_myprog_Ppnxt_sum

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

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

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

% scalasca -examine epik_myprog_Ppnxt_sum
```

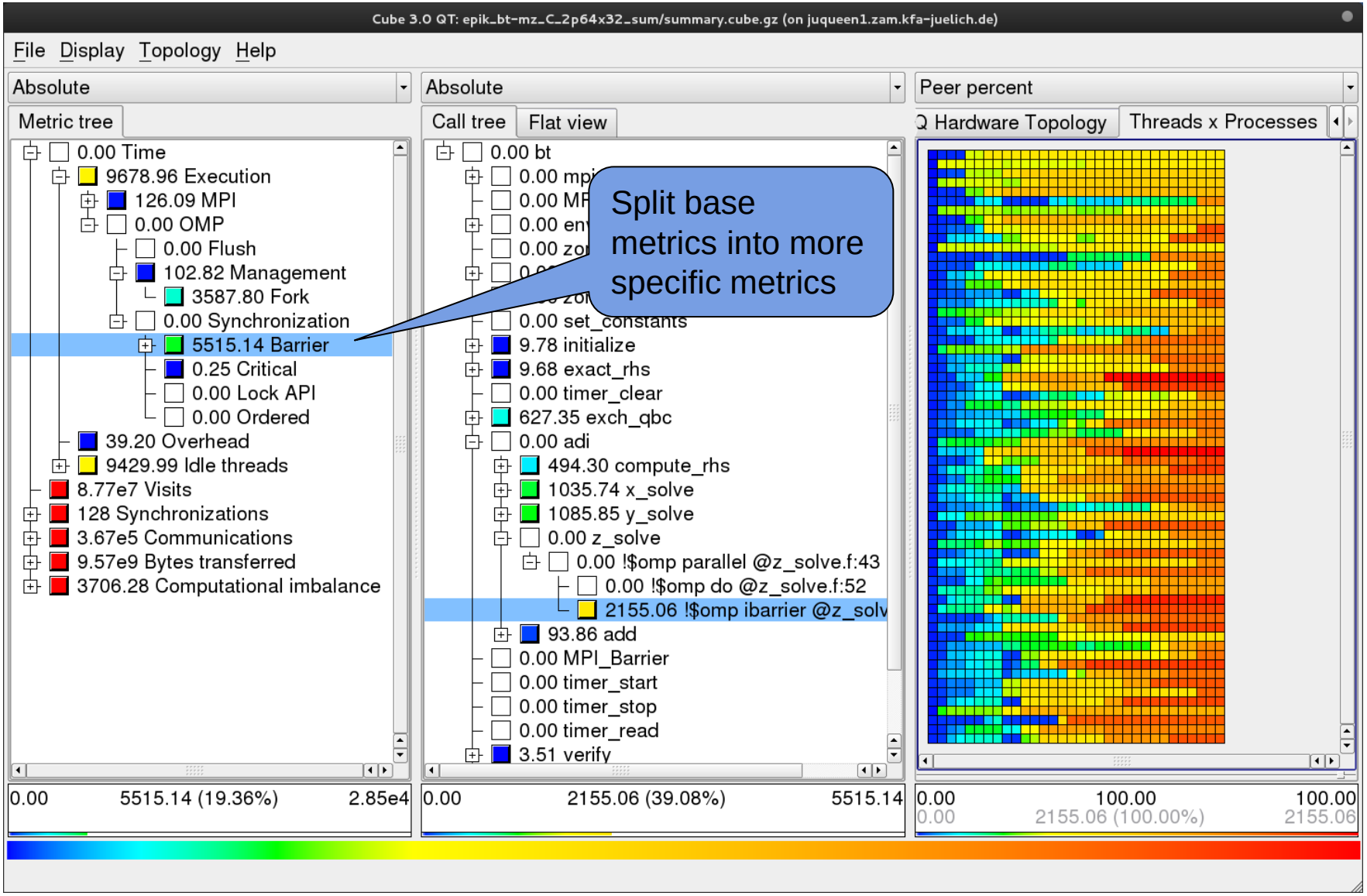
Call-path Profile: Example (cont.)



Call-path Profile: Example (cont.)



Call-path Profile: Example (cont.)



Why is the Bottleneck There?

- This is highly application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Manual/automatic event trace analysis

HW Counter Measurements w/ Scalasca

- Scalasca supports both PAPI and native counters
- Available counters:

```
% module load UNITE papi/5.0.1
% less $PAPI_ROOT/doc/papi-5.0.1-avail.txt
% less $PAPI_ROOT/doc/papi-5.0.1-native_avail.txt
% less $PAPI_ROOT/doc/papi-5.0.1-avail-detail.txt
```

- Specify using "`-m`" option of "`scalasca -analyze`":

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

module load UNITE scalasca
scalasca -analyze -f filter.txt \
    -m PAPI_FP_OPS:PAPI_TOT_INS:PAPI_TOT_CYC \
    runjob --ranks-per-node P --np n [...] --exe ./myprog
```

Manual Instrumentation w/ Scalasca

- Can be used to mark initialization, solver & other phases
  - Annotation macros ignored by default
  - Enabled with "`-user`" flag of "`scalasca -instrument`"
- Appear as additional regions in analyses
  - Distinguishes performance of important phase from rest

```
#include "epik_user.inc"

subroutine foo(…)
  ! Declarations
  EPIK_USER_REG(solve, "<solver>")

  ! Some code…
  EPIK_USER_START(solve)
  do i=1,100
    [...]
  end do
  EPIK_USER_END(solve)
  ! Some more code…
end subroutine
```

Fortran (requires C preprocessor)

```
#include "epik_user.h"

void foo(…) {
  /* Declarations */
  EPIK_USER_REG(solve, "<solver>");

  /* Some code… */
  EPIK_USER_START(solve);
  for (i=0; i<100; i++) {
    [...]
  }
  EPIK_USER_END(solve);
  /* Some more code… */
}
```

C / C++

Selective Measurement w/ Scalasca

- Can be used to temporarily disable measurement for certain intervals
  - Annotation macros ignored by default
  - Enabled with "`-user`" flag of "`scalasca -instrument`"
- Appear as synthetic PAUSED region in analyses

```
#include "epik_user.inc"

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

Fortran (requires C preprocessor)

```
#include "epik_user.h"

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

C / C++

Scalasca API Limitations

- START and END calls must be correctly nested
  - That is, you cannot start a region in one routine and end it in another
- Measurement control API calls are not allowed within OpenMP parallel regions
- Behavior of automatic trace analysis is undefined when PAUSING skips recording MPI events on subsets of processes
  - Examples:
    - A collective operation is excluded only on some ranks involved
    - A send/receive on one rank is excluded w/o the corresponding send/receive on the matching rank
  - Will generally lead to a deadlock

Trace Generation & Analysis w/ Scalasca

- Identifies inefficiency patterns in communication and synchronization operations
- Specify using "`-t`" option of "`scalasca -analyze`":

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

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

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

Score-P

- Community instrumentation and measurement infrastructure
  - Developed by a consortium of performance tool groups
  - Next generation measurement system of
    - Scalasca 2.x
    - Vampir
    - TAU
    - Periscope
- Common data formats improve tool interoperability
- http://www.score-p.org

Score-P: Recipe Overview

- Load module
  - "`module load UNITE scorep`"
- Flat MPI profile
  - Prefix ***link command*** with "`scorep --nocompiler`"
- Call-path profile
  - Prefix ***compile & link commands*** with "`scorep`"
- Run application as usual
  - Measurement can be extensively configured via environment variables
  - Check output of "`scorep-info config-vars`" for details
  - Filtering is slightly more complicated than with Scalasca, but also more powerful

TAU

- Very portable tool set for instrumentation, measurementand analysis of parallel multi-threaded applications
- http://tau.uoregon.edu/

- Supports
  - Various profiling modes and tracing
  - Various forms of code instrumentation
  - C, C++, Fortran, Java, Python
  - MPI, multi-threading (OpenMP, Pthreads, …)

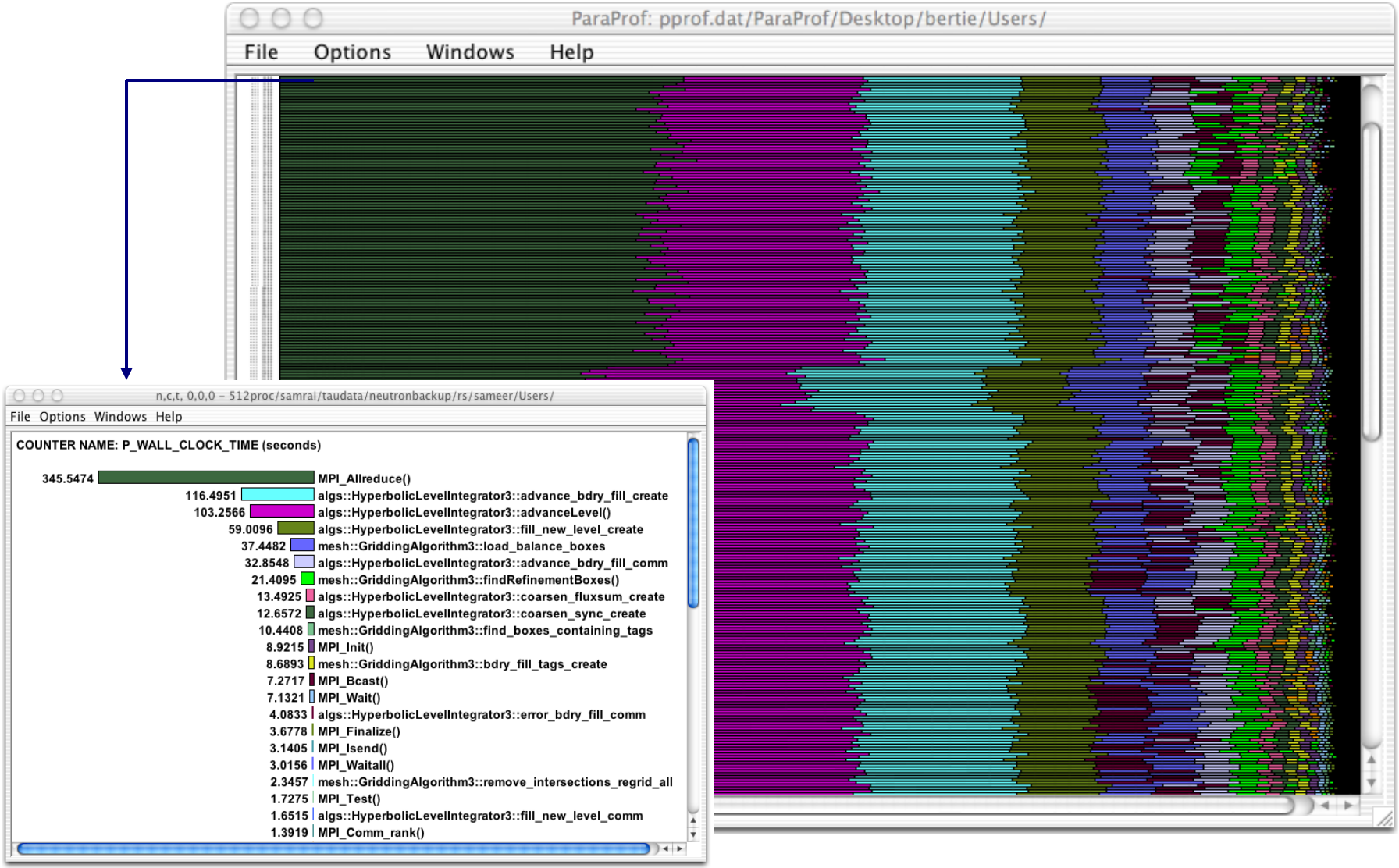
TAU: Instrumentation

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

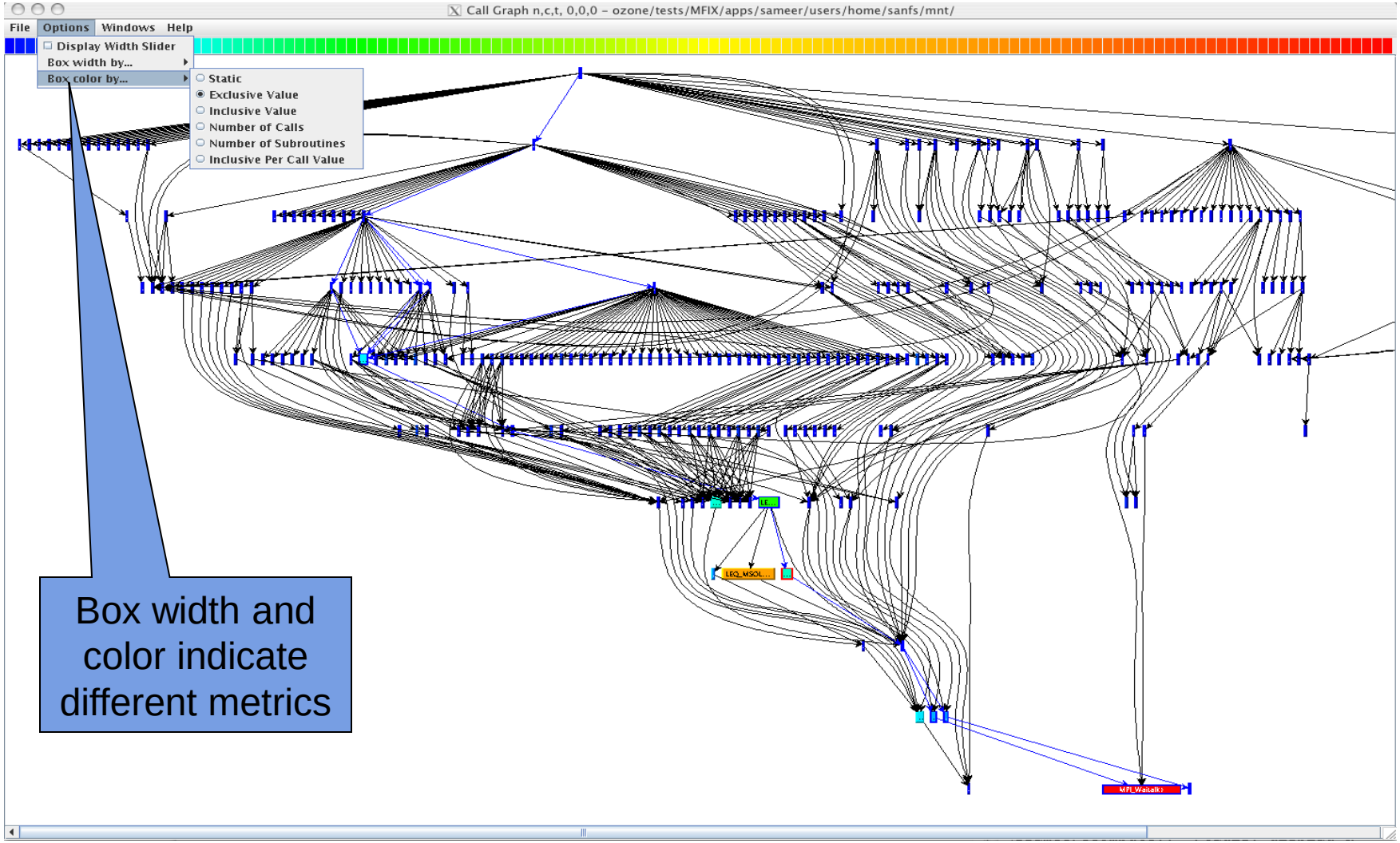
TAU: Recipe

1. `module load UNITE tau` # once per session
2. Specify programming model by setting `TAU_MAKEFILE` to one of `$TAU_MF_DIR/Makefile.tau-*`
   - `MPI: Makefile.tau-bgqtimers-papi-mpi-pdt`
   - `OpenMP/MPI: Makefile.tau-bgqtimers-papi-mpi-pdt-openmp-opari`
3. Compile and link with
   - `tau_cc.sh file.c ...`
   - `tau_cxx.sh file.cxx...`
   - `tau_f90.sh file.f90 ...`
4. Execute with real input data
   Environment variables control measurement mode
   - TAU_PROFILE, TAU_TRACE, TAU_CALLPATH, …
5. Examine results with `paraprof`

TAU: Basic Profile View



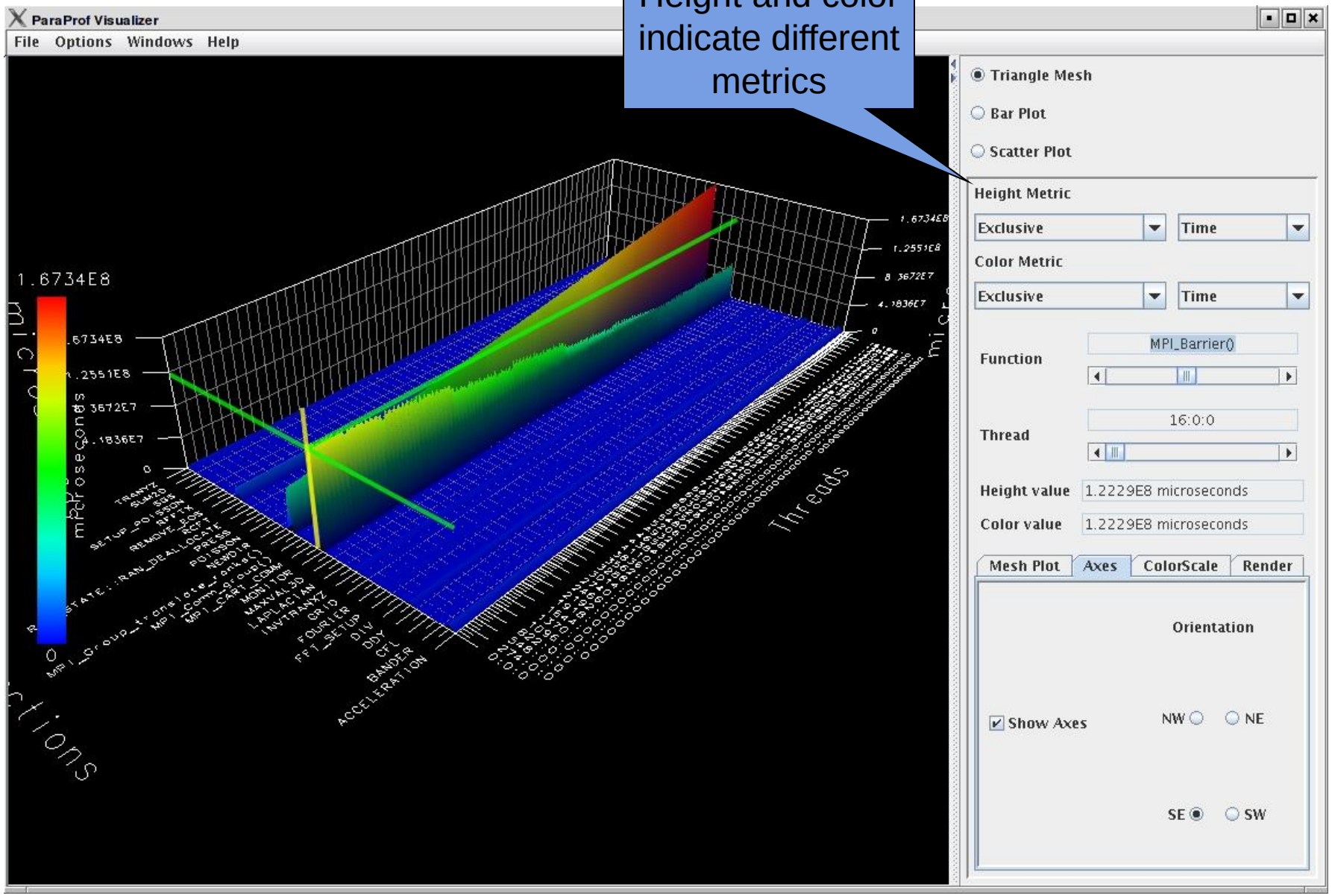
TAU: Callgraph Profile View



Box width and color indicate different metrics

TAU: 3D Profile View

Height and color indicate different metrics

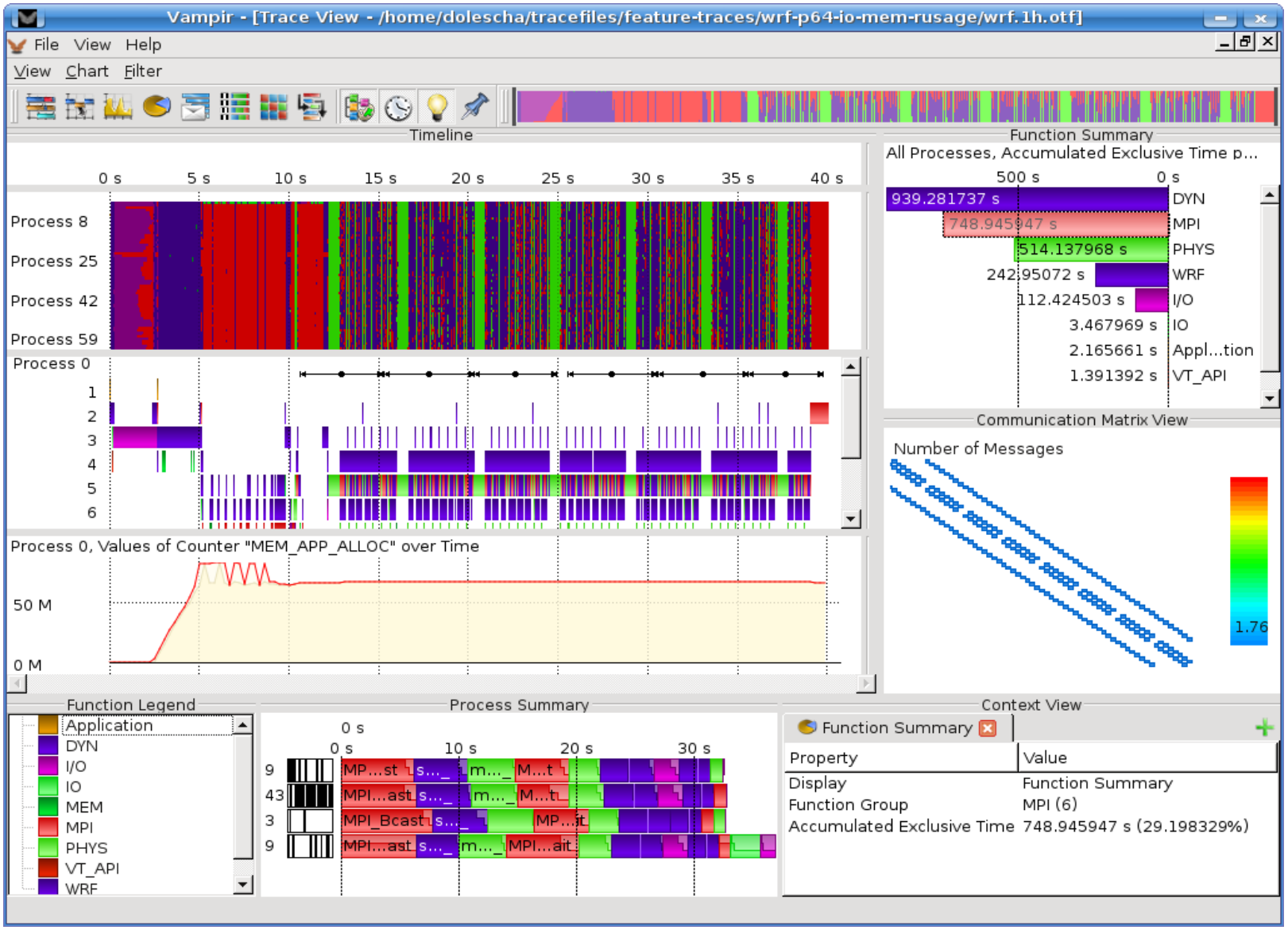


Vampir Event Trace Visualizer

- Offline trace visualization for
  - Score-P OTF2 trace files
  - VampirTrace OTF trace files
  - Scalasca EPILOG trace files
- Visualization of MPI, OpenMP and application events:
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for ANY part of the trace
- http://www.vampir.eu

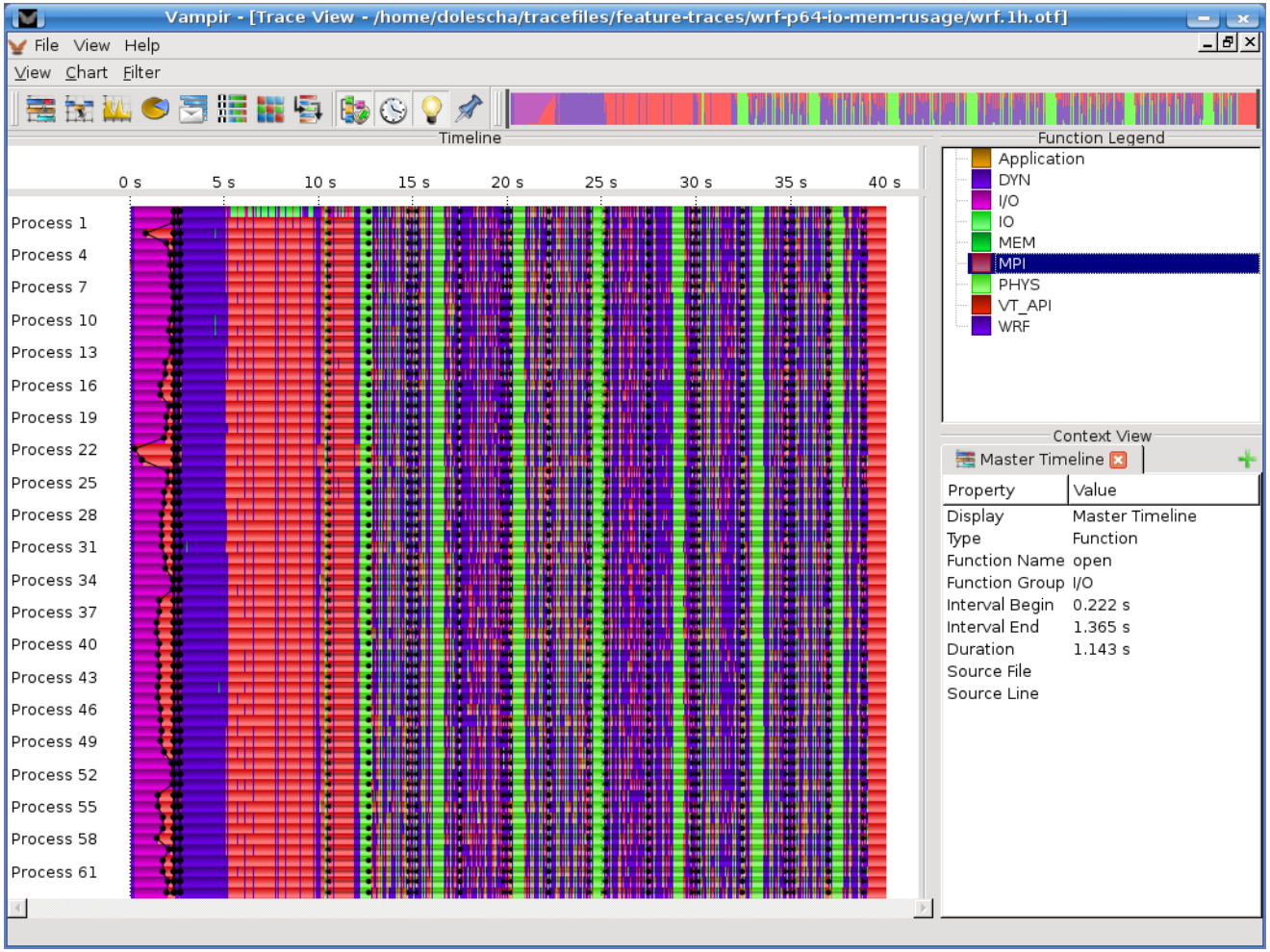
- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis

Vampir Displays



Vampir: Timeline Diagram

- Functions organized into groups
- coloring by group
- Message lines can be colored by tag or size



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

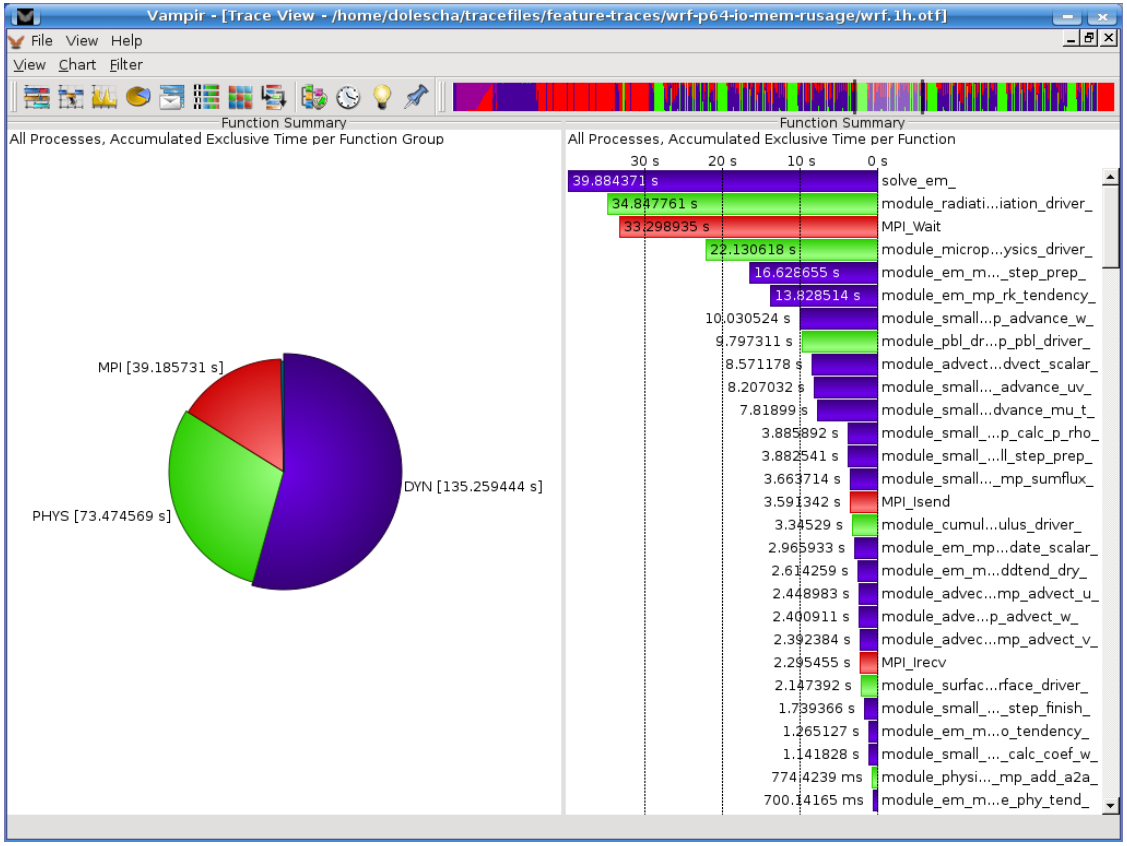
Vampir: Process and Counter Timelines

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

Vampir: Execution Statistics

- Aggregated profiling information: execution time, number of calls, inclusive/exclusive

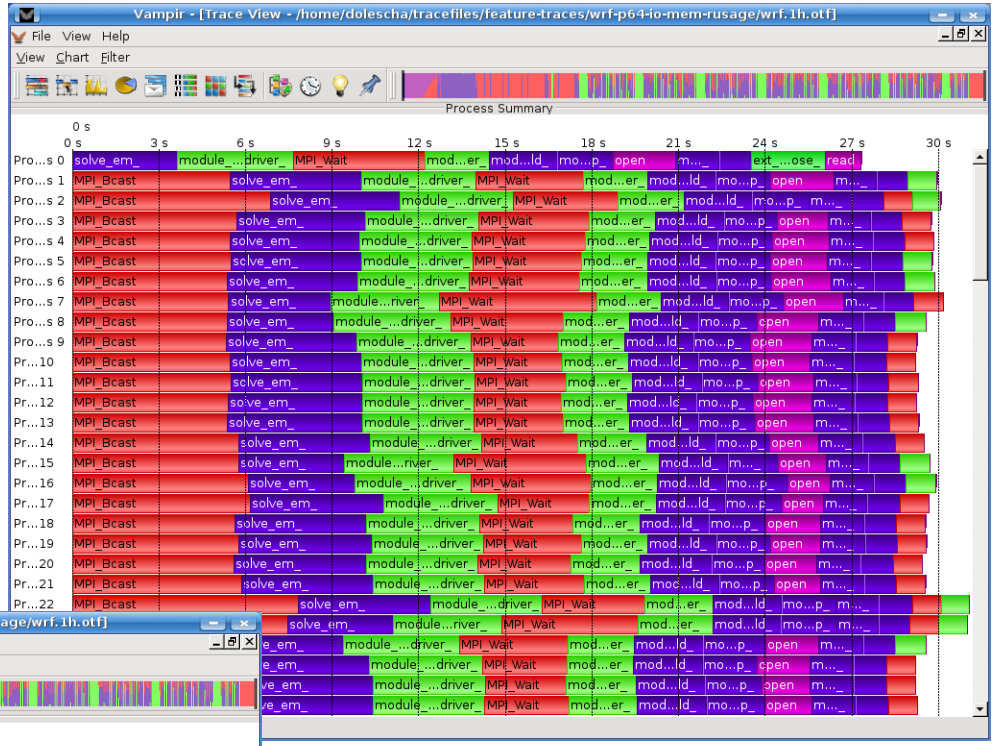
- Available for all / any group (activity) or all routines (symbols)

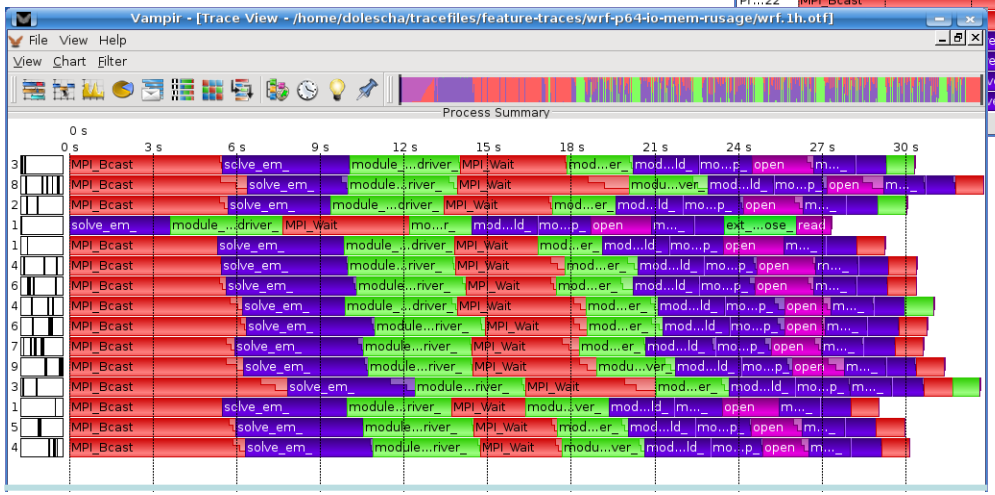


- Available for any part of the trace
  ⇨ selectable through time line diagram

Vampir: Process Summary

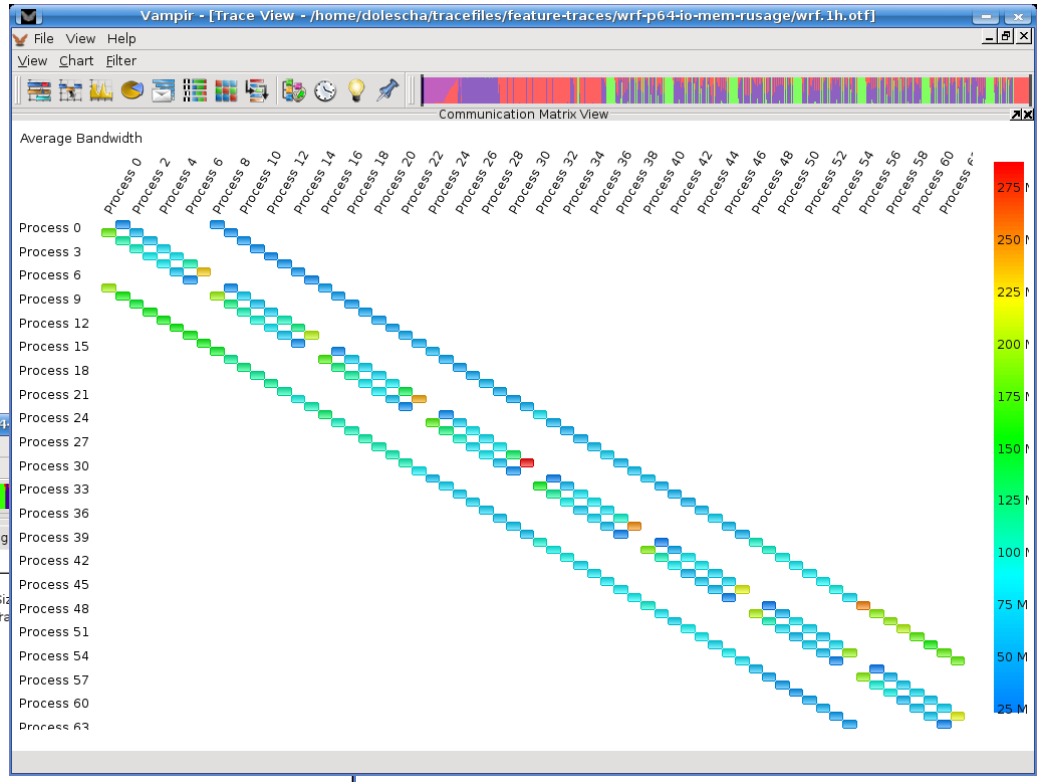
- Execution statistics over all processes for comparison
- Clustering mode available for large process counts

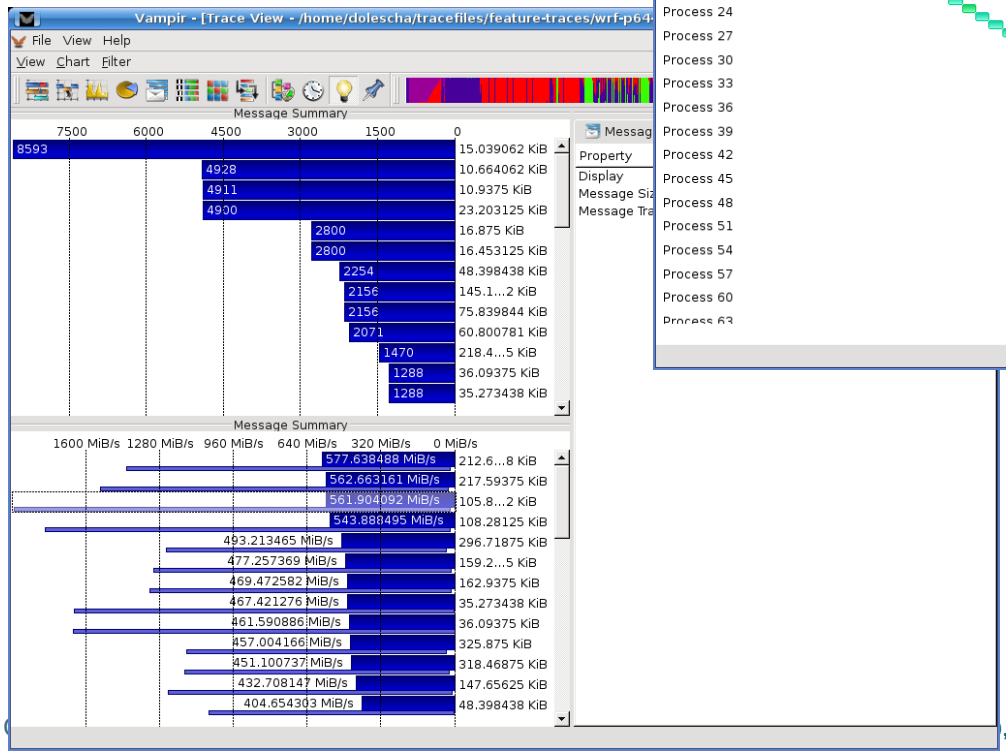




Vampir: Communication Statistics

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





- Available for any part of the trace

Vampir: Recipe (JUQUEEN)

1. `module load UNITE vampirserver`

2. Start Vampir server component using
   "`vampirserver start smp`"
   - Check output for port and pid

3. Connect to server from remote machine (see next slide) and analyze the trace

4. `vampirserver stop <pid>`
   - See above (2.)

Vampir: Recipe (local system)

1. Open SSH tunnel to JUQUEEN using
   "`ssh -L30000:localhost:<port> juqueen<n>`"

2. Start Vampir client component
   - For example: "`/usr/local/zam/unite/bin/vampir`"

3. Select
   1. "Open other…"
   2. "Remote file"
   3. "Connect" (keep defaults)
   4. File "`epik.esd`" from Scalasca trace measurement dir or "`traces.otf2`" from Score-P trace measurement dir

HPCToolkit (Rice University)

- Multi-platform sampling-based call-path profiler
- Works on unmodified, optimized executables
- http://hpctoolkit.org

- Advantages:
  - Overhead can be easily controlled via sampling interval
  - Advantageous for complex C++ codes with many small functions
  - Loop-level analysis (sometimes even individual source lines)
  - Supports POSIX threads
- Disadvantages:
  - Statistical approach that might miss details
  - MPI/OpenMP time displayed as low-level system calls

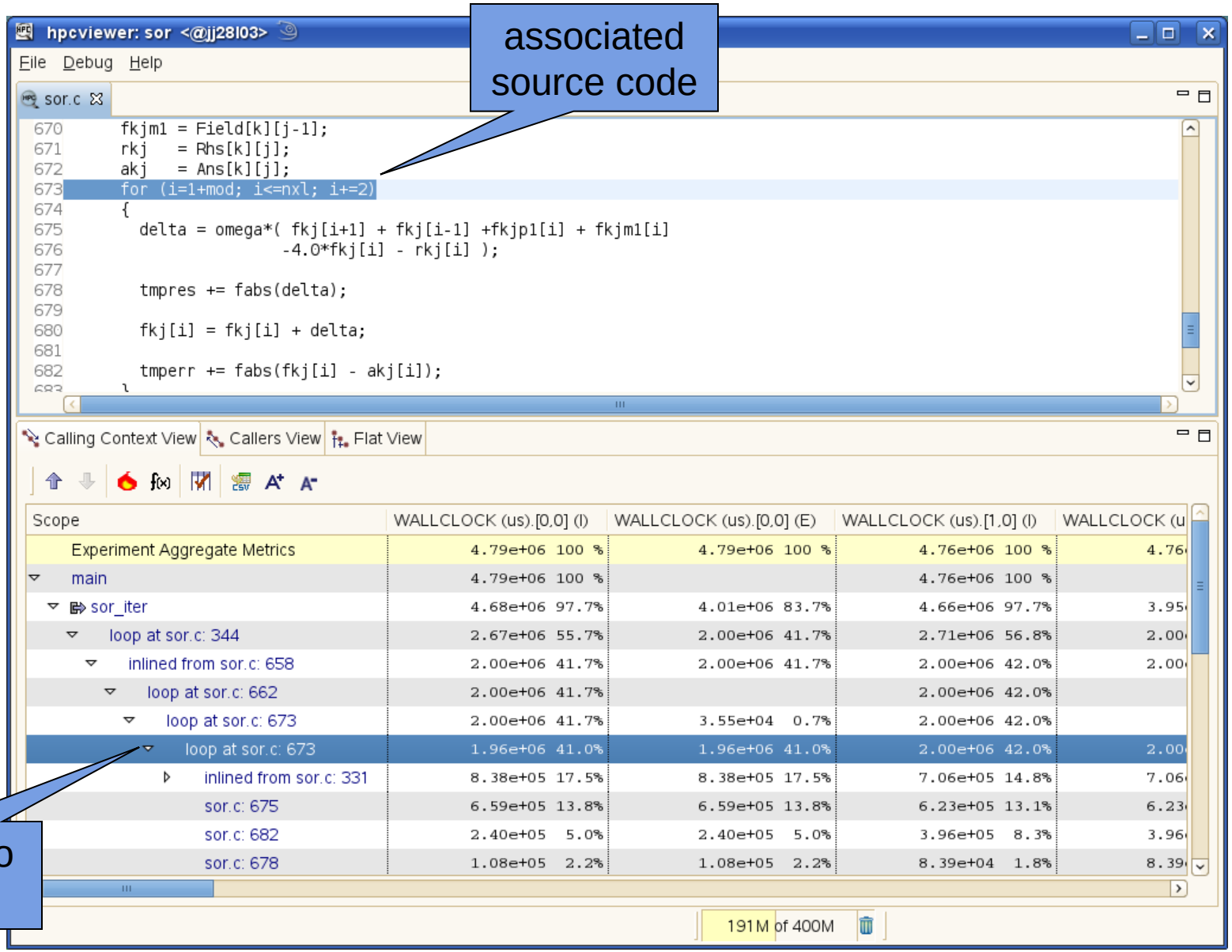
HPCToolkit: Recipe

1. Compile your code with "`-g -qnoipa`"
   - For MPI, also make sure your application calls MPI_Comm_rank first on MPI_COMM_WORLD
2. Prefix your ***link command*** with "`hpclink`"
   - Ignore potential linker warnings ;-)
3. Run your application as usual, specifying requested metrics with sampling intervals in environment variable
   "`HPCRUN_EVENT_LIST`"
4. Perform static binary analysis with
   "`hpcstruct --loop-fwd-subst=no <app>`"
5. Combine measurements with
   "`hpcprof -S <struct file> \`
   `-I "<path_to_src>/*" <measurement_dir>`"
6. View results with
   "`hpcviewer <hpct_database>`"

HPCToolkit: Metric Specification

- General format:
  "`name@interval [;name@interval ...]`"

- Possible sample sources:
  - WALLCLOCK
  - PAPI counters
  - IO (use w/o interval spec)
  - MEMLEAK (use w/o interval spec)

- Interval: given in microseconds
  - E.g., 10000 → 100 samples per second

Example: hpcviewer