Performance Analysis with Scalasca

George S. Markomanolis

7 August 2019

ORNL is managed by UT-Battelle, LLC for the US Department of Energy

Outline

- Introduction to Scalasca
- How to compile (using Score-P)
- Explaining functionalities of Scalasca/CUBE4 on two applications
- Testing a case with a large trace

Scalasca

- Scalasca is a software tool that supports the performance analysis of parallel applications
- The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes.
- Installed version on Summit: v2.25
- Module: scalasca
- For instrumentation is used Score-P
- Web site: https://www.scalasca.org
- Email: scalasca@fz-juelich.de

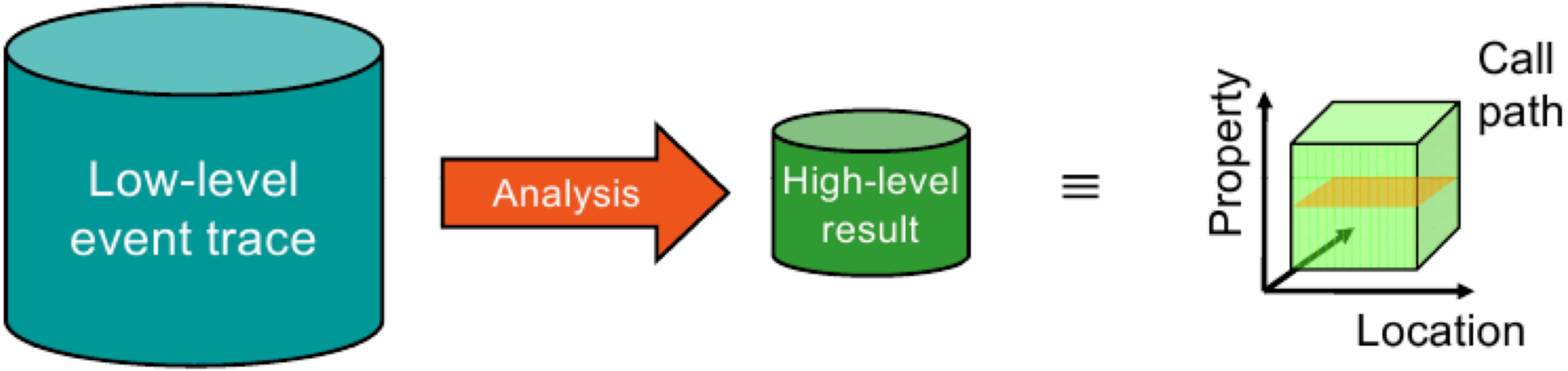
Capability Matrix - Scalasca

Capability	Profiling	Tracing	Notes/Limitations
MPI, MPI-IO	Yes	Yes	
OpenMP CPU	Yes	Yes	
OpenMP GPU	No	No	
OpenACC	No	No	Score-P does instrument but CUBE does not provide information
CUDA	No	No	Score-P does instrument but CUBE does not provide information
POSIX I/O	Yes	Yes	
POSIX threads	Yes	Yes	
Memory – app-level	Yes	Yes	
Memory – func-level	Yes	Yes	
Hotspot Detection	Yes	Yes	
Variance Detection	Yes	Yes	
Hardware Counters	Yes	Yes	

Techniques

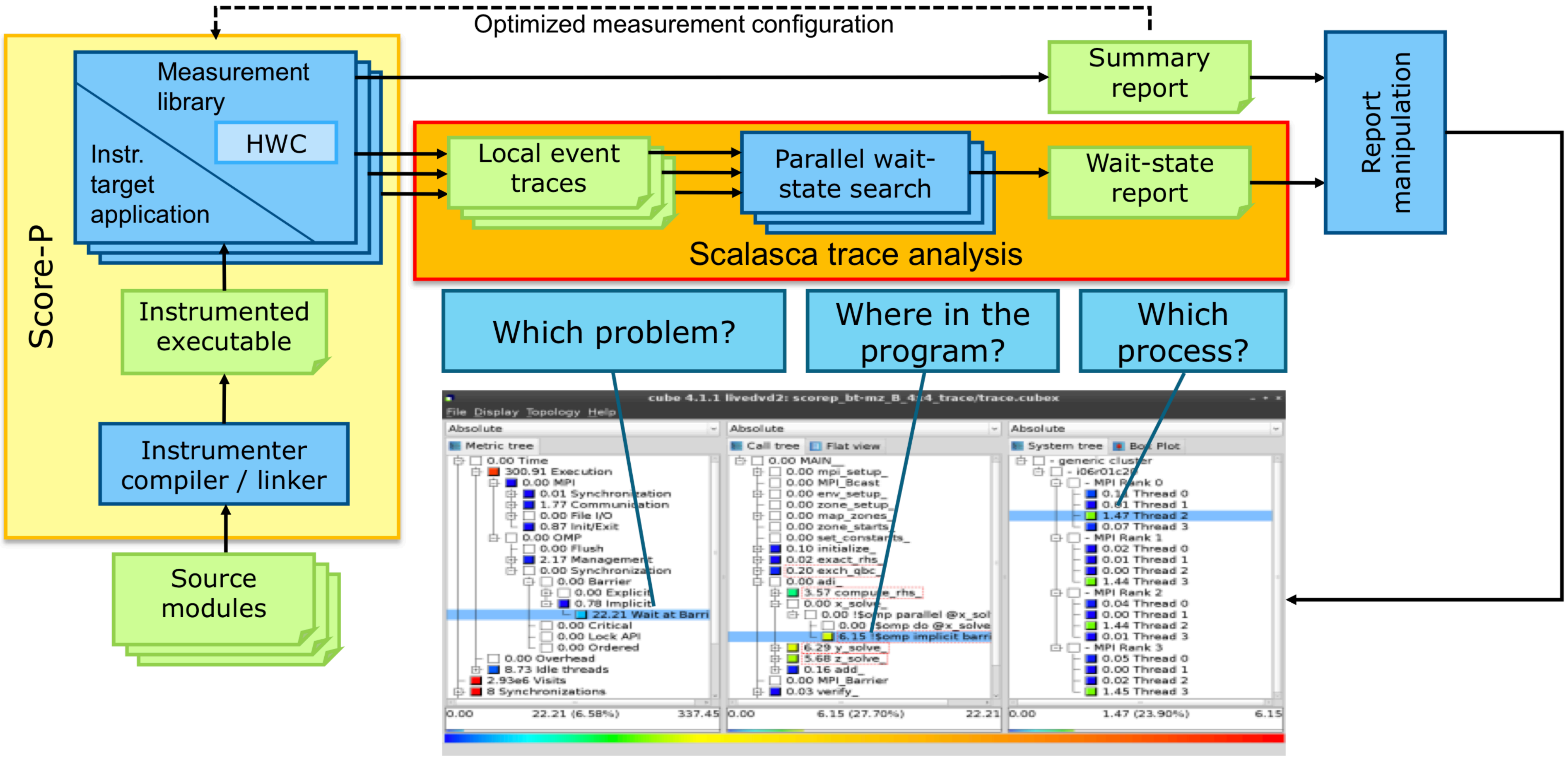
- Profile analysis:
  - Summary of aggregated metrics
    - Per function/call-path and/or per process/thread
  - mpiP, TAU, PerfSuite, Vampir
- Time-line analysis
- Pattern analysis

Automatic trace analysis



- Apply tracing
- Automatic search for patterns on inefficient behavior
- Classificaiton of behavior
- Much faster than manual trace analysis
- Scalability

Workflow



OAK RIDGE National Laboratory

CUBE4

- Parallel program analysis report exploration tools
  - Libraries for XML report
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
- Three coupled tree browsers
  - Performance property
  - Call-tree path
  - System location
- CUBE4 displays severities
  - Value for precise comparison
  - Colour for easy identification of hostpots
  - Inclusive valye when closed and exclusive when expanded

Scalasca on Summit

module load scalasca

scalasca

```
Scalasca 2.5
Toolset for scalable performance analysis of large-scale parallel applications
usage: scalasca [OPTION]... ACTION <argument>...

  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

Options:
  -c, --show-config     show configuration summary and exit
  -h, --help            show this help and exit
  -n, --dry-run         show actions without taking them
      --quickref        show quick reference guide and exit
      --remap-specfile  show path to remapper specification file and exit
  -v, --verbose         enable verbose commentary
  -V, --version         show version information and exit
```

Scalasca Workflow

- Compilation: use Score-P
- Execution of the binary for profiling (it will create an output folder):

  scalasca -analyze jsrun …

- Examine of the data (GUI is loaded)

  scalasca -examine output_folder

MiniWeather – MPI – Tools parameters

- Parameters for Scalasca/Score-P

```
module load scorep/6.0_r14595
module load scalasca/2.5

export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_FP_OPS
export SCOREP_MPI_ENABLE_GROUPS=ALL
export SCAN_MPI_LAUNCHER=jsrun
```

Instrumentation

Type of instrumentation	Instrumenter switch	Default value	Instrumented routines	Runtime measurement control
MPI	--mpp=mpi/ --mpp=none	(auto)	configured by install	'Selection of MPI Groups'
SHMEM	--mpp=shmem/ --mpp=none	(auto)	configured by install	—
CUDA	--[no]cuda	enabled	all	'CUDA Performance Measurement'
OpenCL	--[no]opencl	enabled	configured by install	'OpenCL Performance Measurement'
OpenACC	--[no]openacc	enabled	configured by install	'OpenACC Performance Measurement'
OpenMP	--thread=omp / --[no]openmp	(auto)	all parallel constructs, see Note below	—
Pthread	--thread=pthread	(auto)	Basic Pthread library calls	—
'Automatic Compiler Instrumentation'	--[no]compiler	enabled	all	'Filtering'
'Recording of I/O activities'	--[no]io[=...]	disabled	configured by install	'Recording of I/O activities'
'Source-Code Instrumentation Using PDT'	--[no]pdt	disabled	all	'Filtering'
'Semi-Automatic Instrumentation of POMP2 User Regions'	--[no]pomp	disabled	manually annotated	'Filtering'
'Manual Region Instrumentation'	--[no]user	disabled	manually annotated	'Filtering' and 'Selective Recording'
'Score-P User Library Wrapping'	--libwrap=[...]	disabled	all by library wrapper	'Filtering'

MiniWeather - MPI

- Edit the Makefile and add the $PREP before the compiler name
- Compile:

MPI: make PREP="scorep --mpp=mpi --pdt" mpi

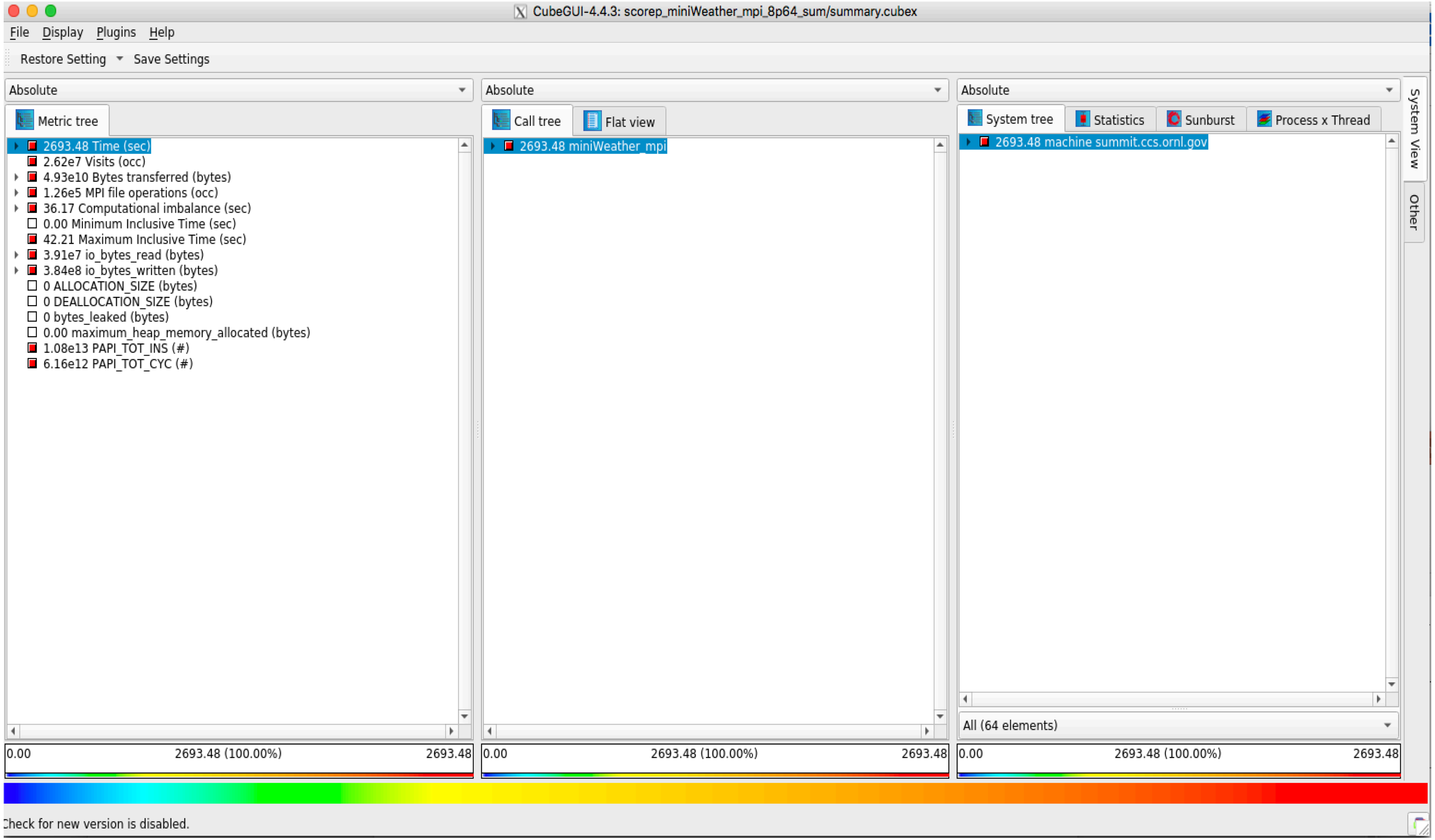
- Execution (submission script):

scalasca -analyze jsrun -n 64 -r 8 ./miniWeather_mpi

- Visualize:

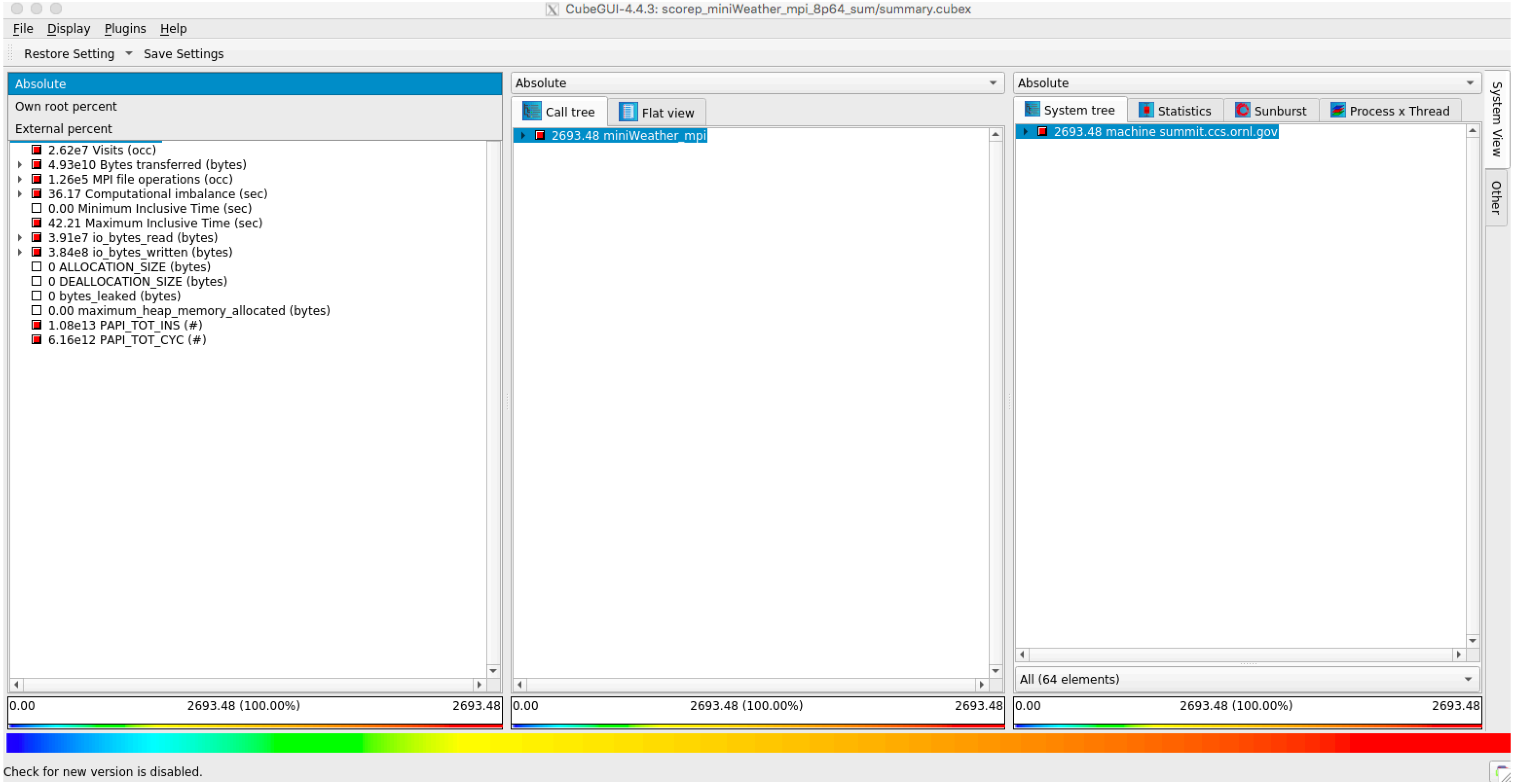
scalasca -examine /gpfs/…/scorep_miniWeather_mpi_8p64_sum

CUBE4 – Central View

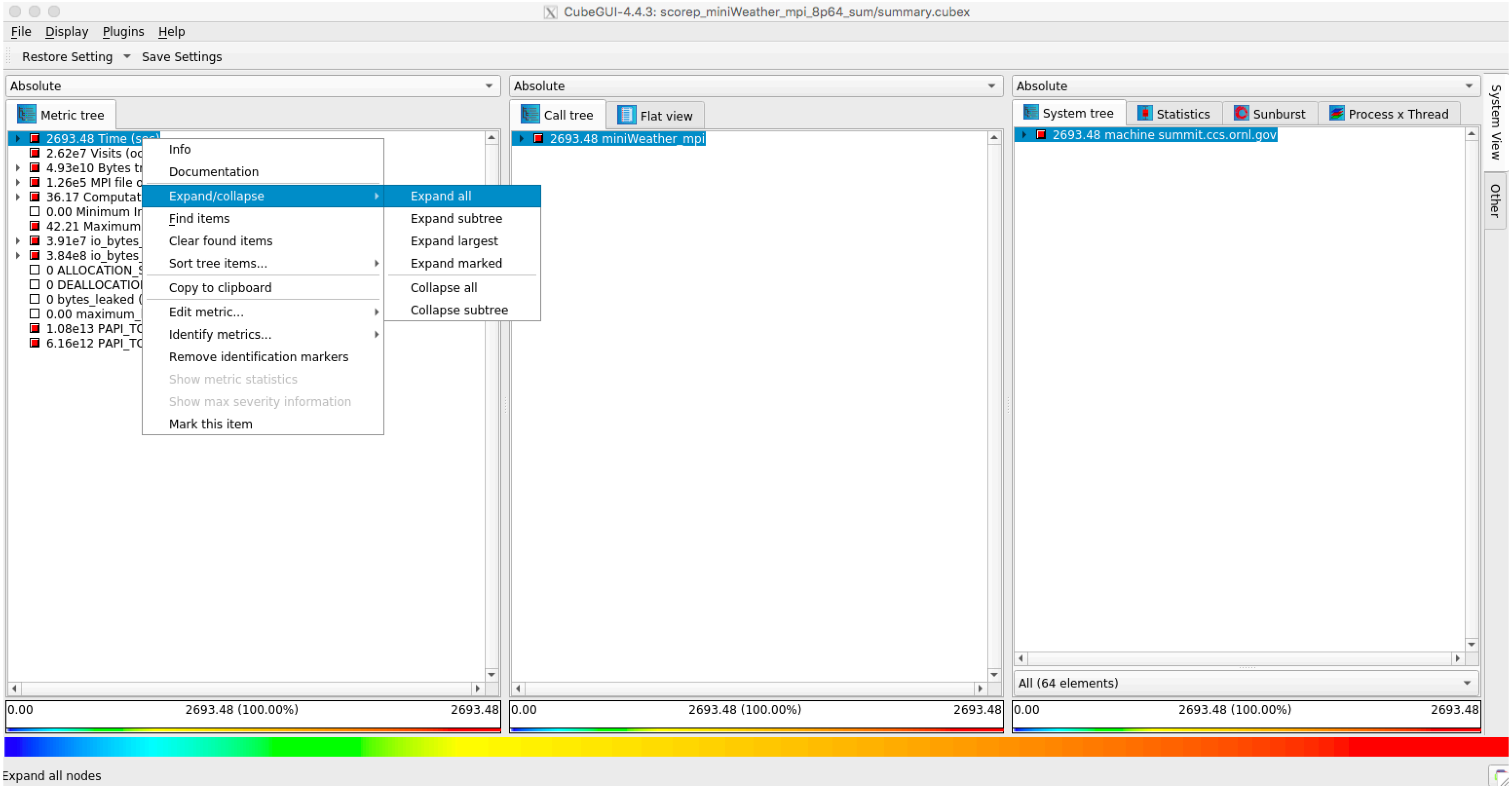


3 Windows:
Metric tree
Call tree
System tree

Exploring the menus

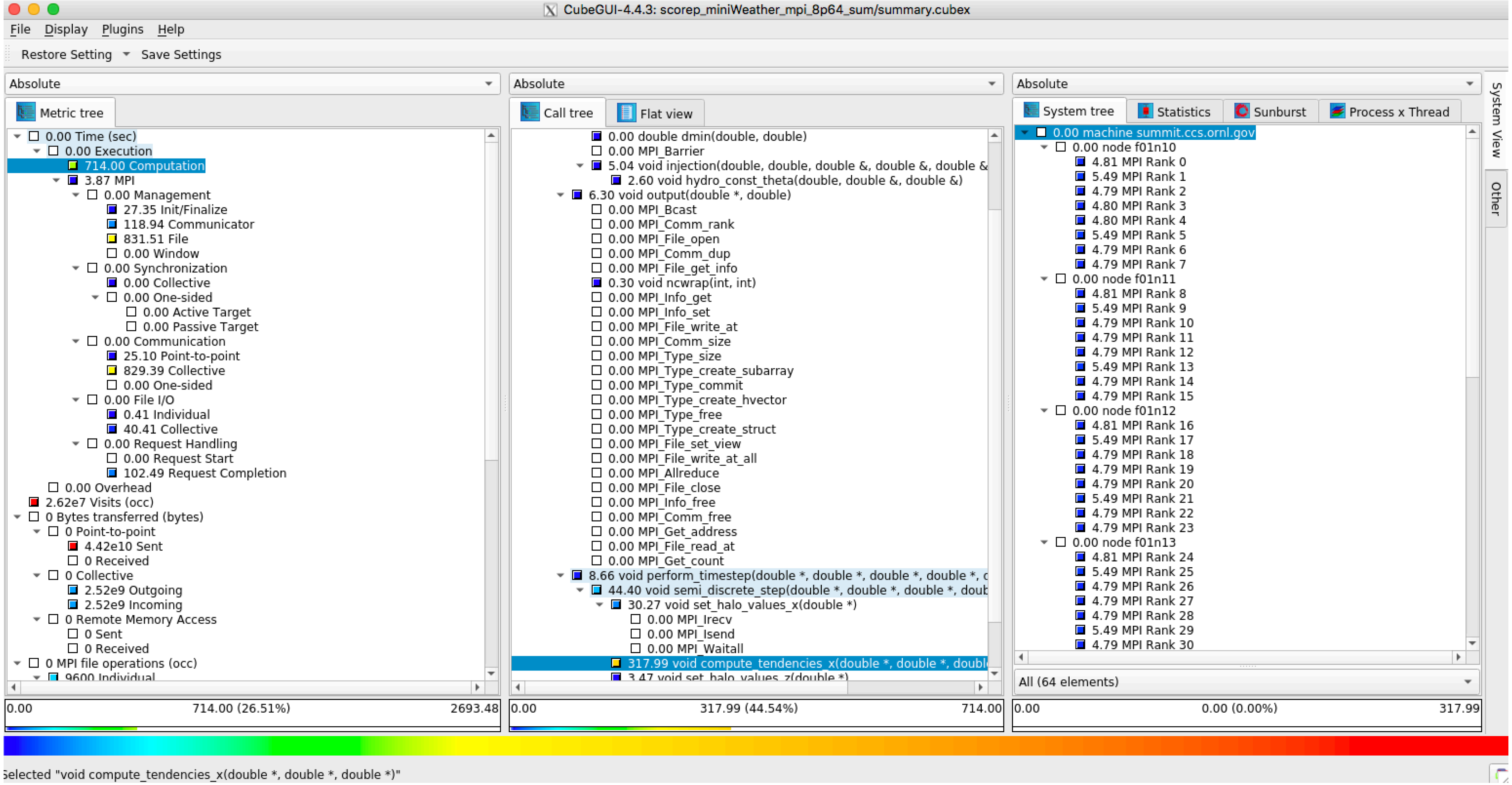


How to expand the trees

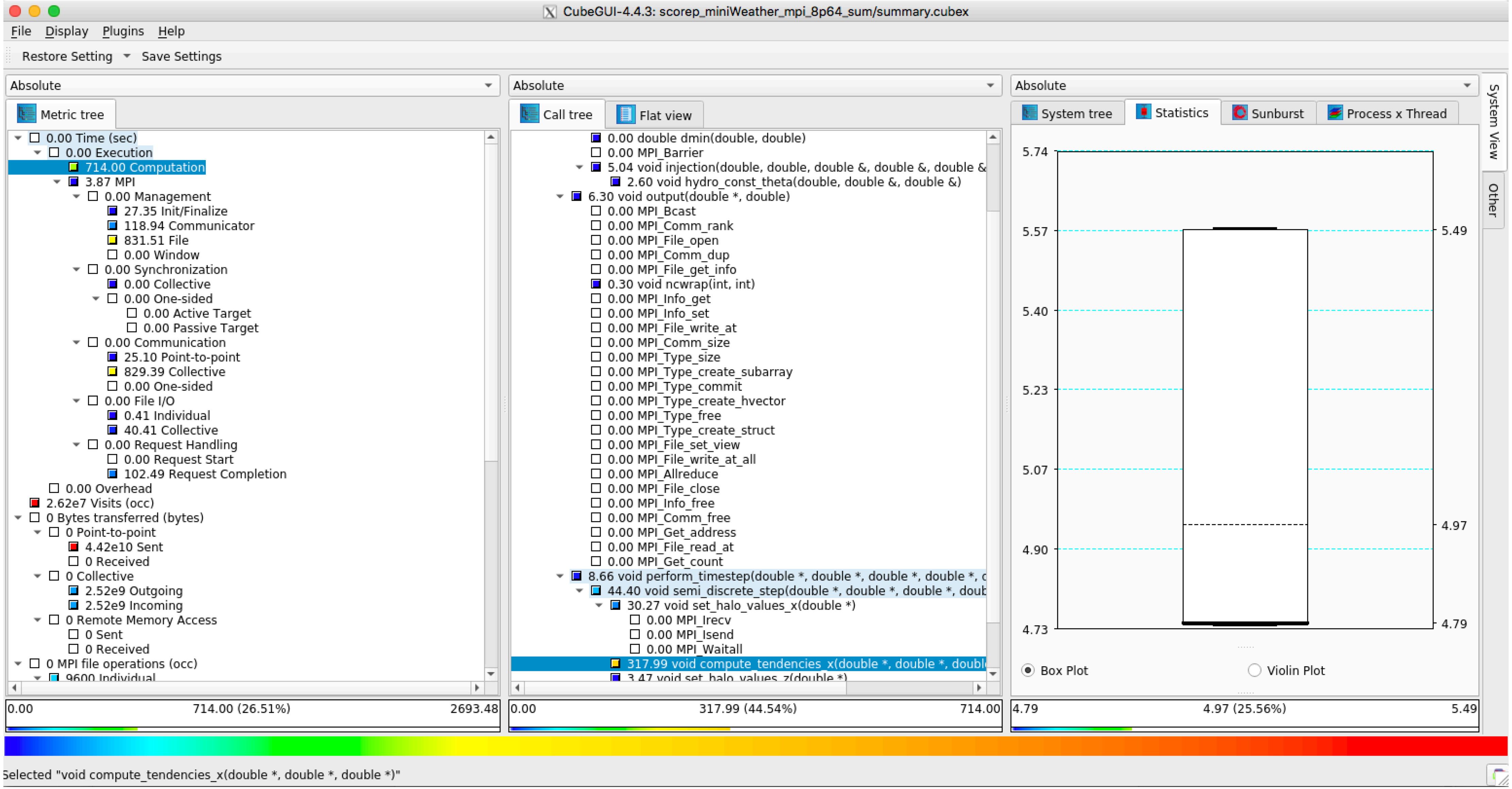


Oak Ridge National Laboratory

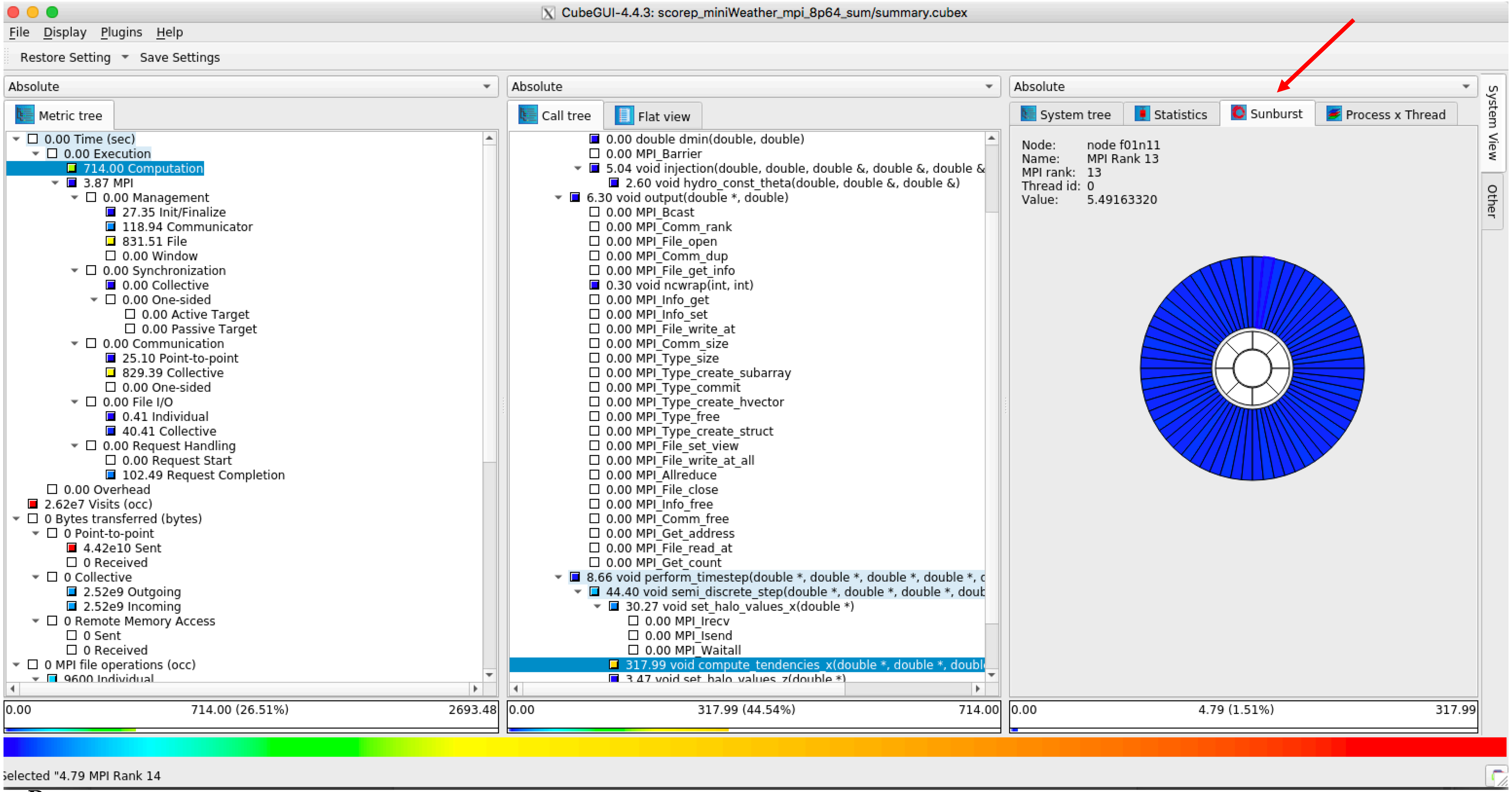
Computation – System tre



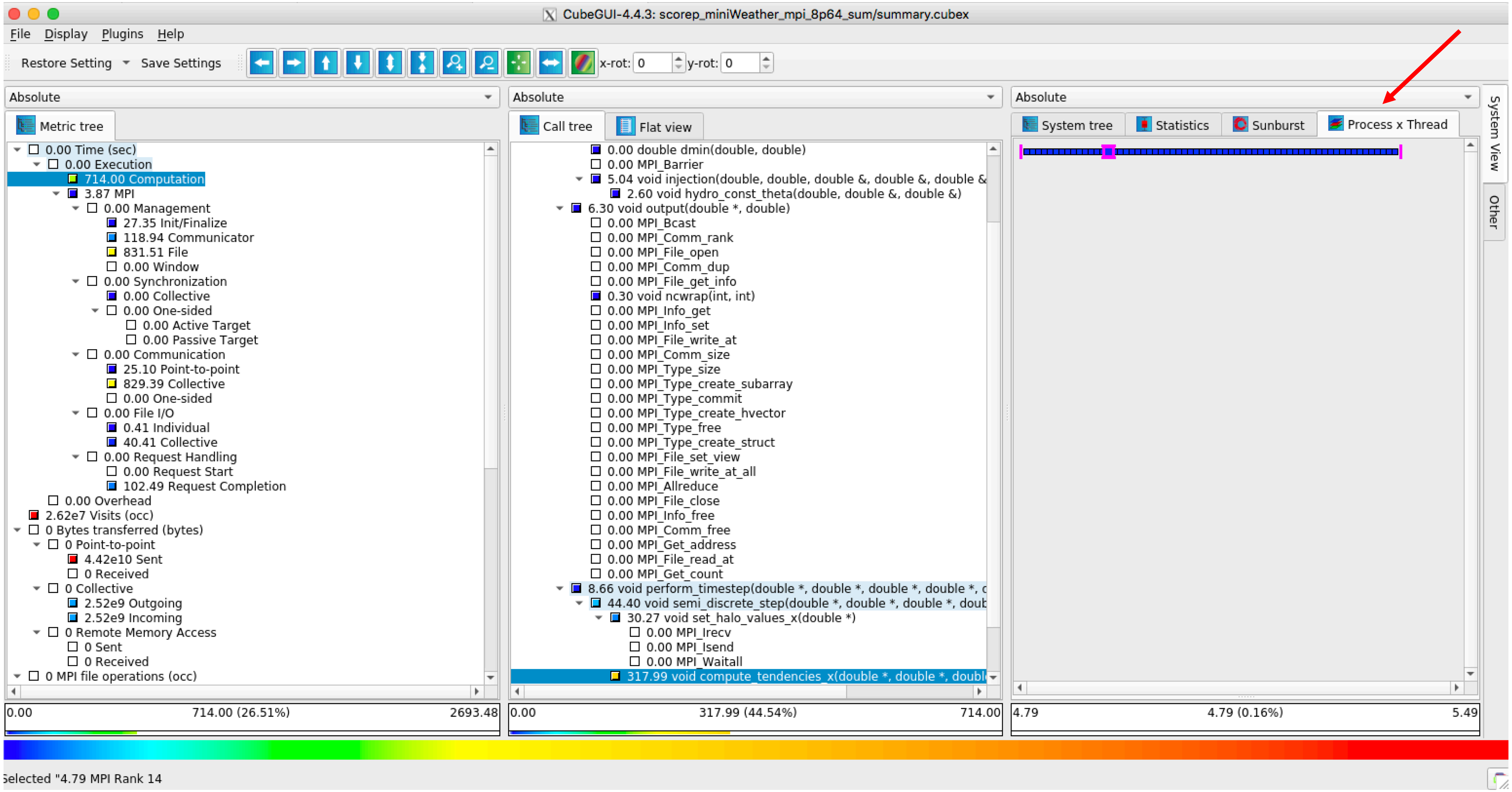
Computation – Blox plot



Computation Sunburst

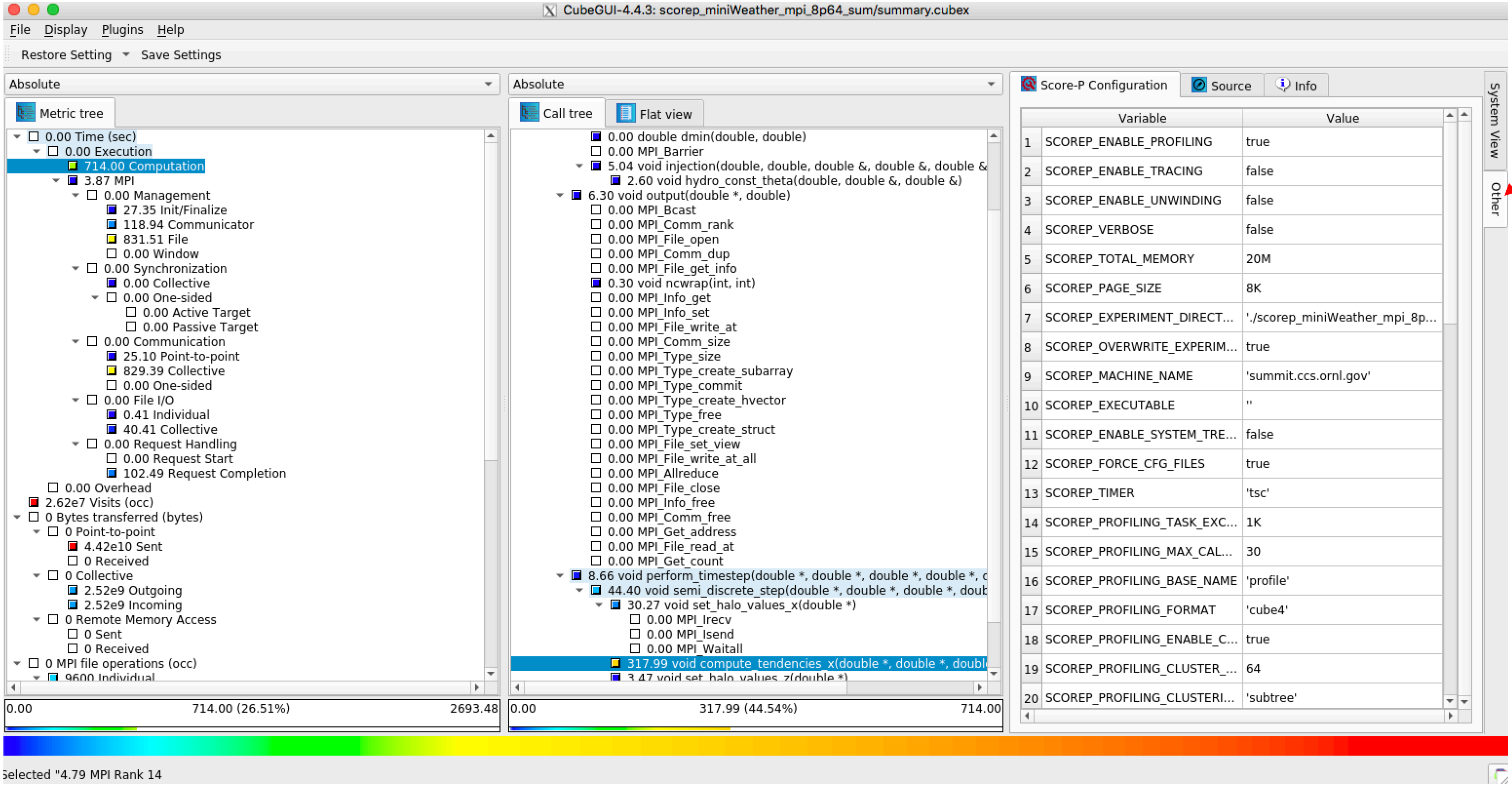


Computation – Process x Thread

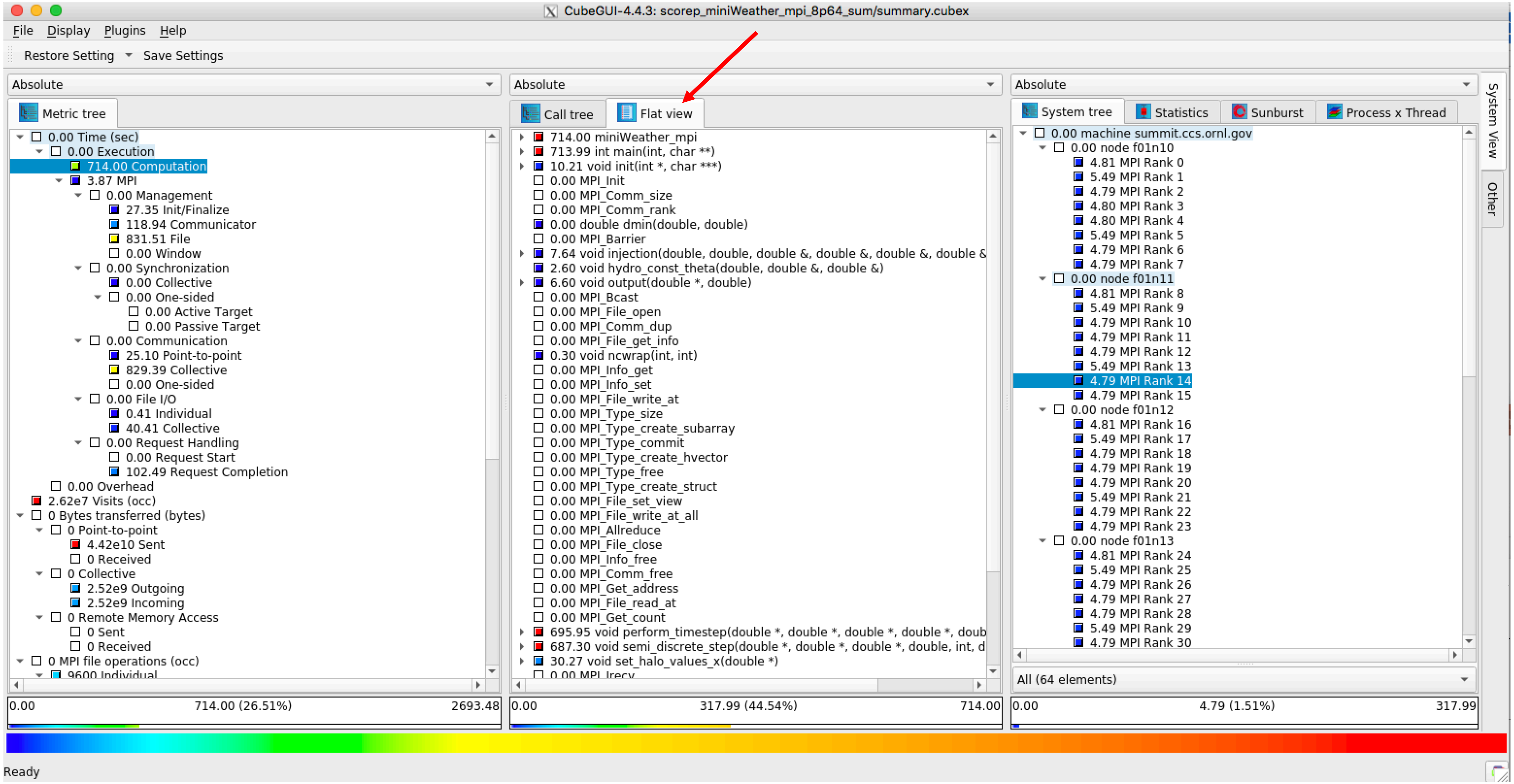


OAK RIDGE National Laboratory

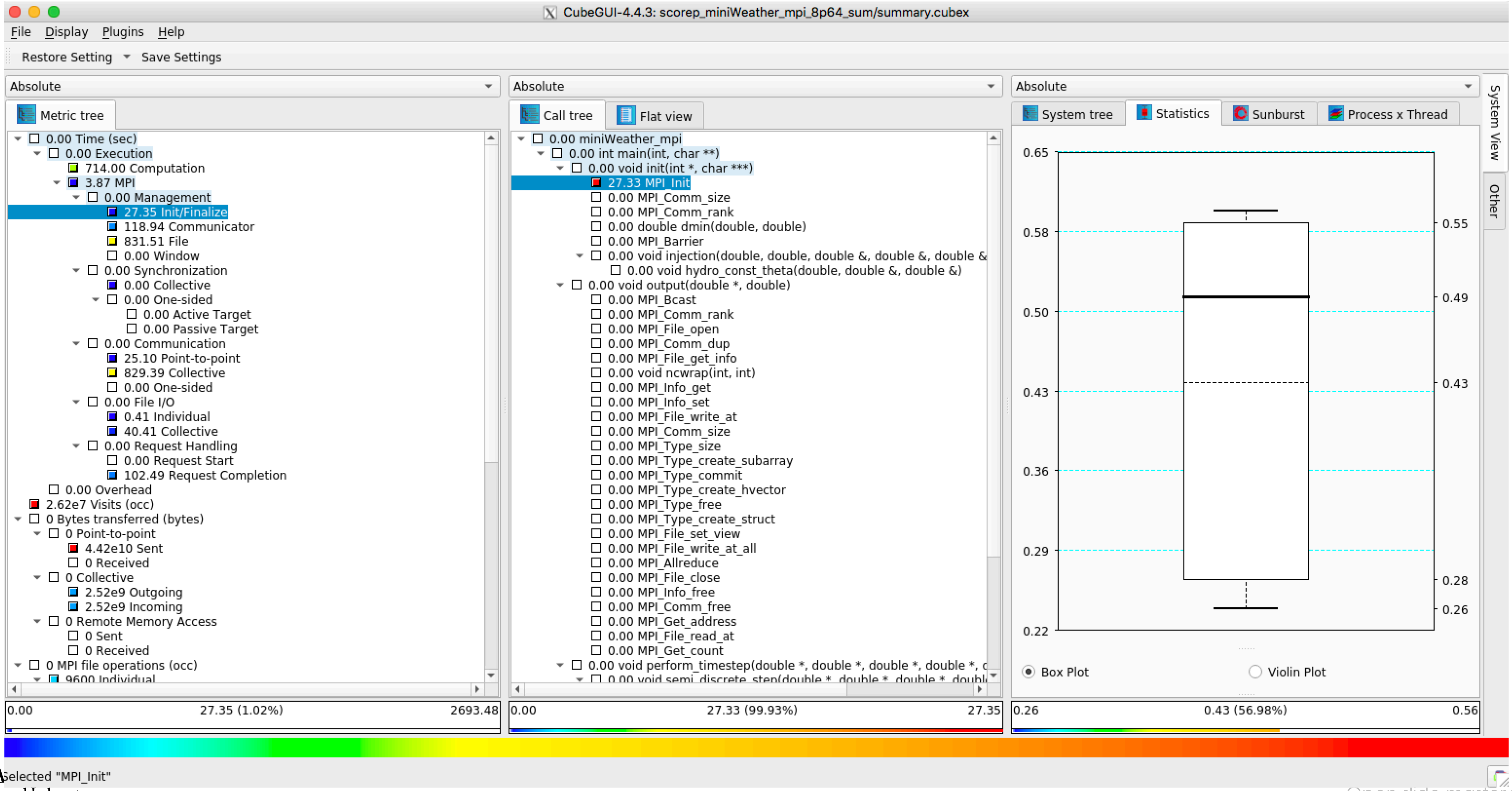
Score-P configuration parameters



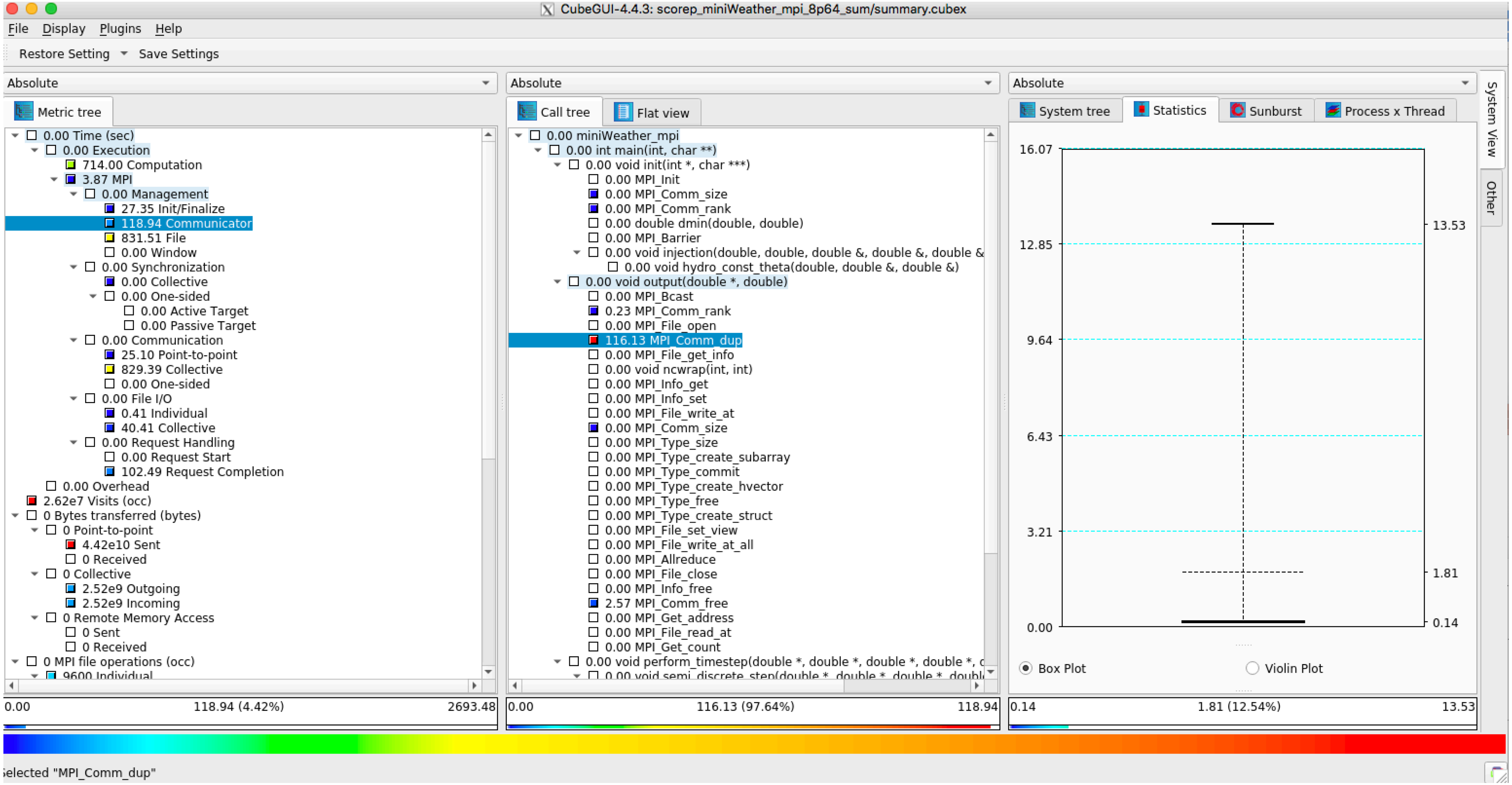
CUBE – Flat view



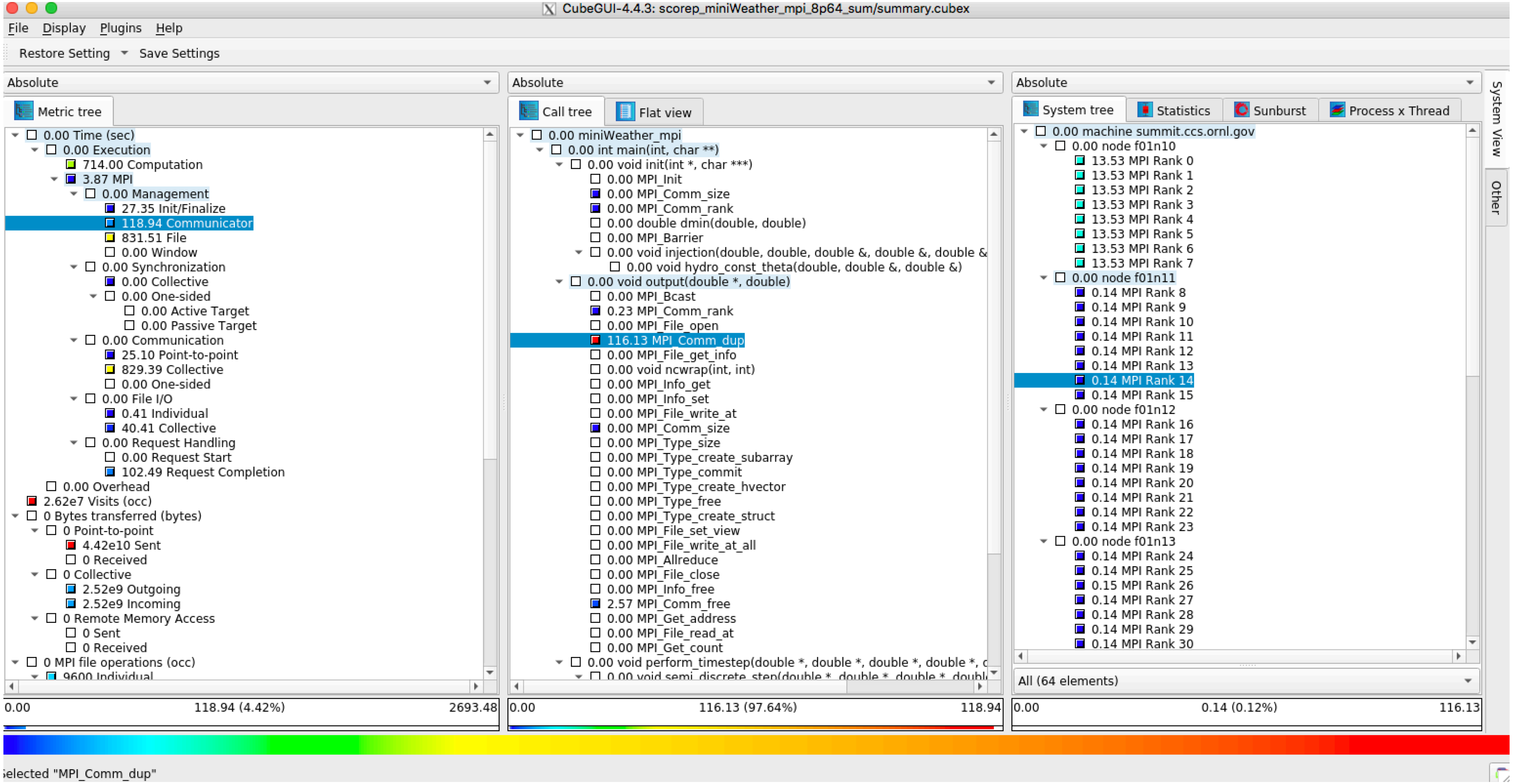
Initialization variation



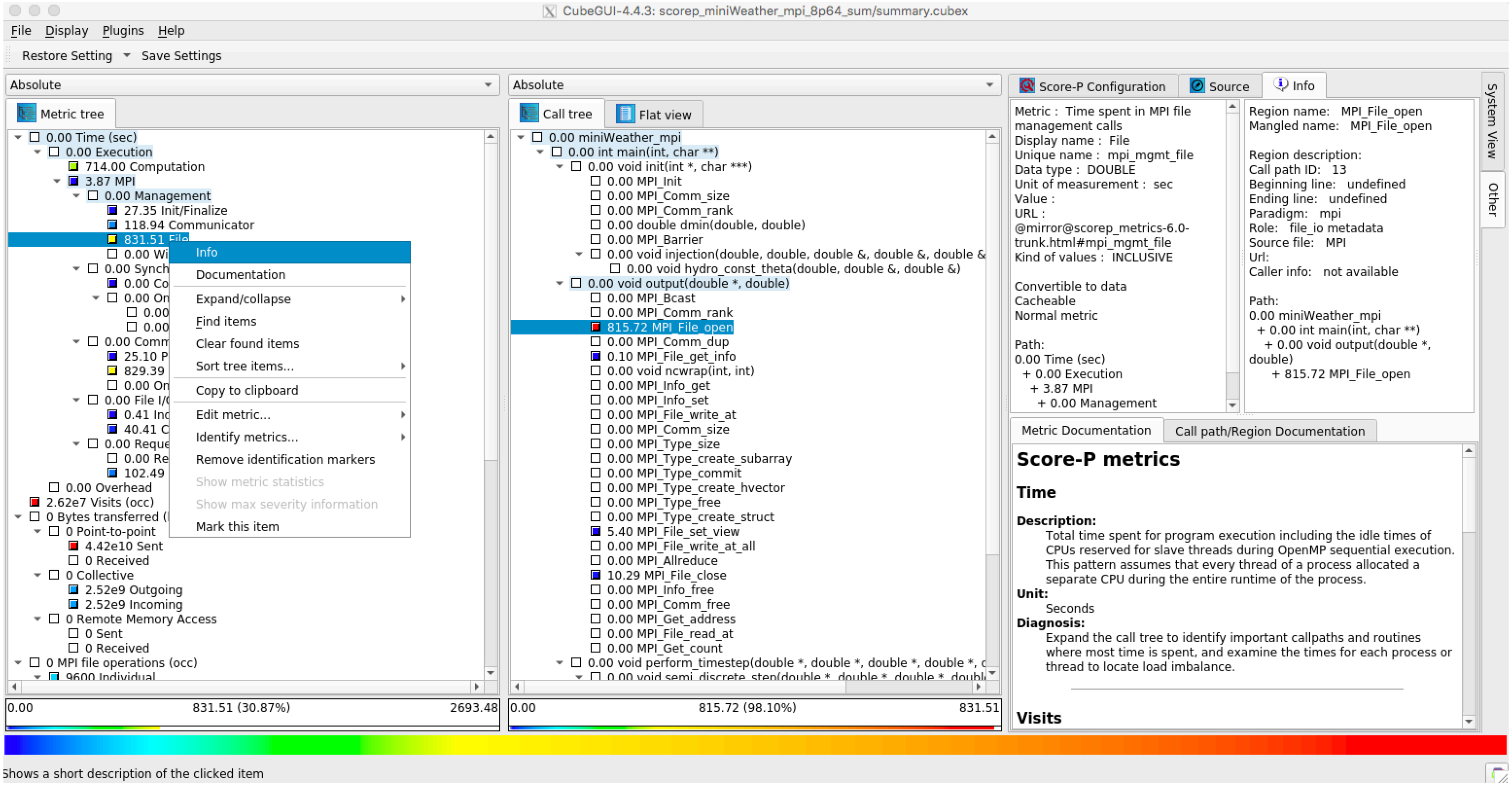
MPI_Comm_dup variation



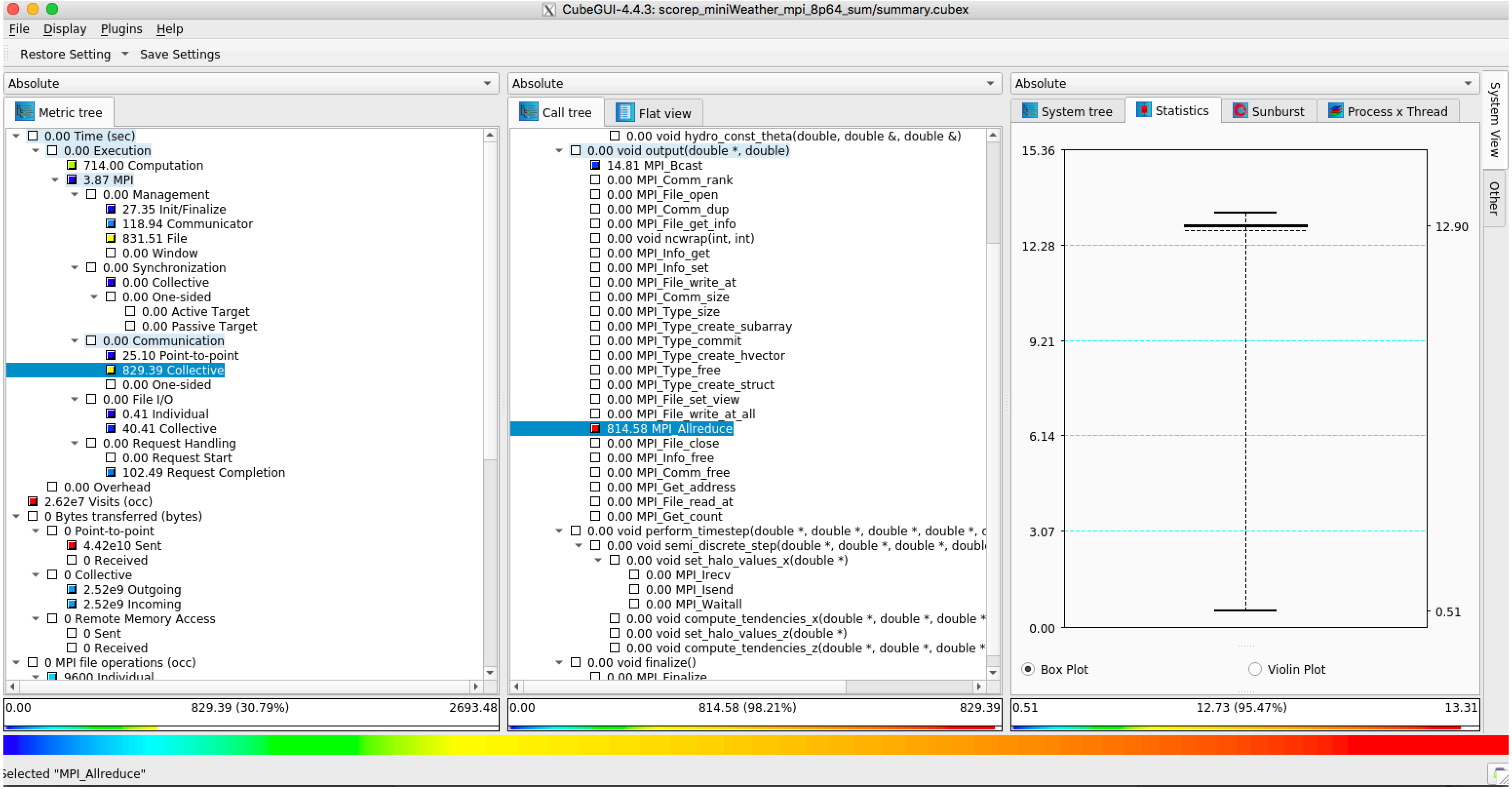
MPI_Comm_dup variation II



Getting information about metrics



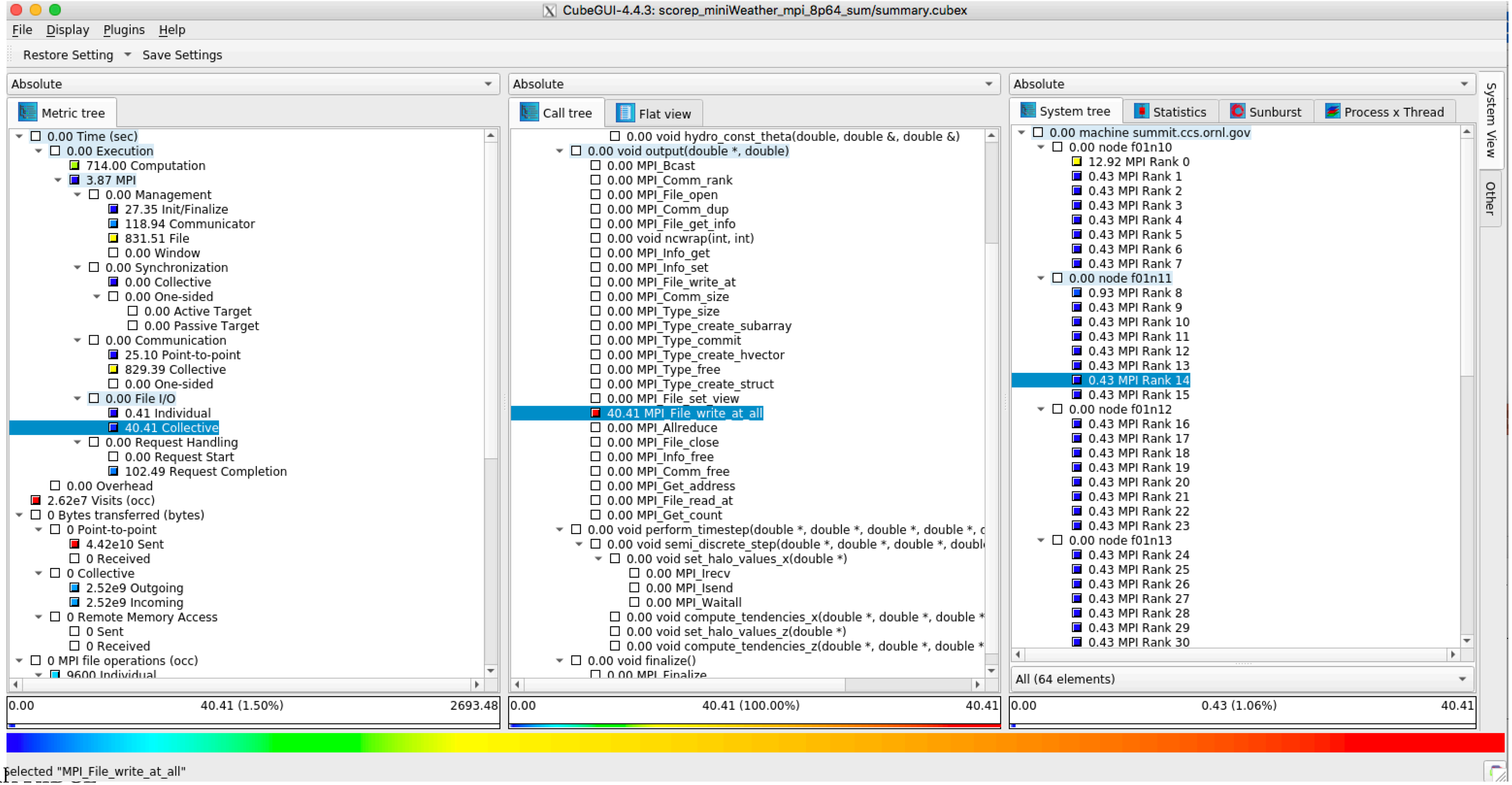
MPI_Allreduce variation



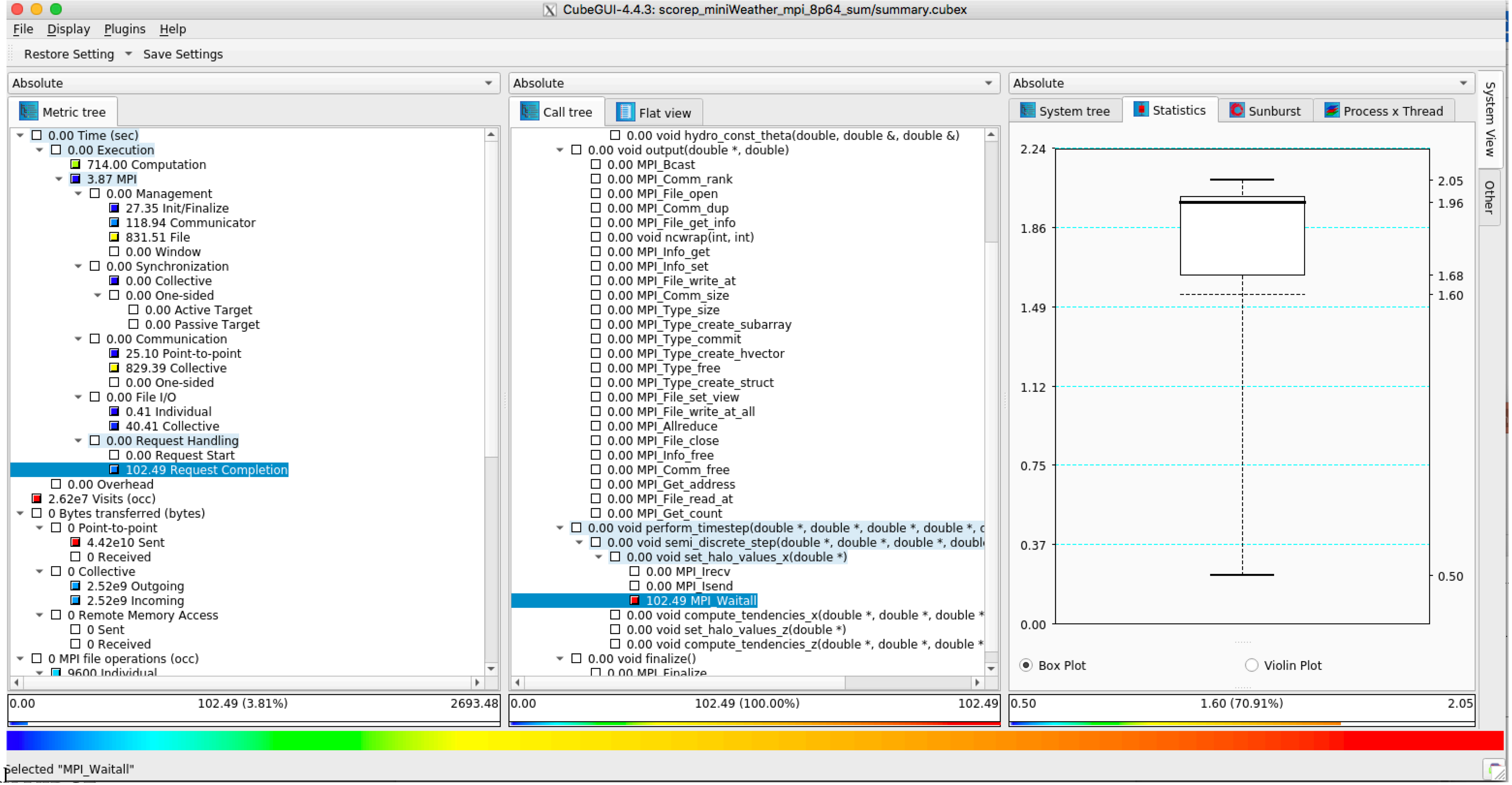
OAK RIDGE National Laboratory

Open slide master to edit

Collective I/O

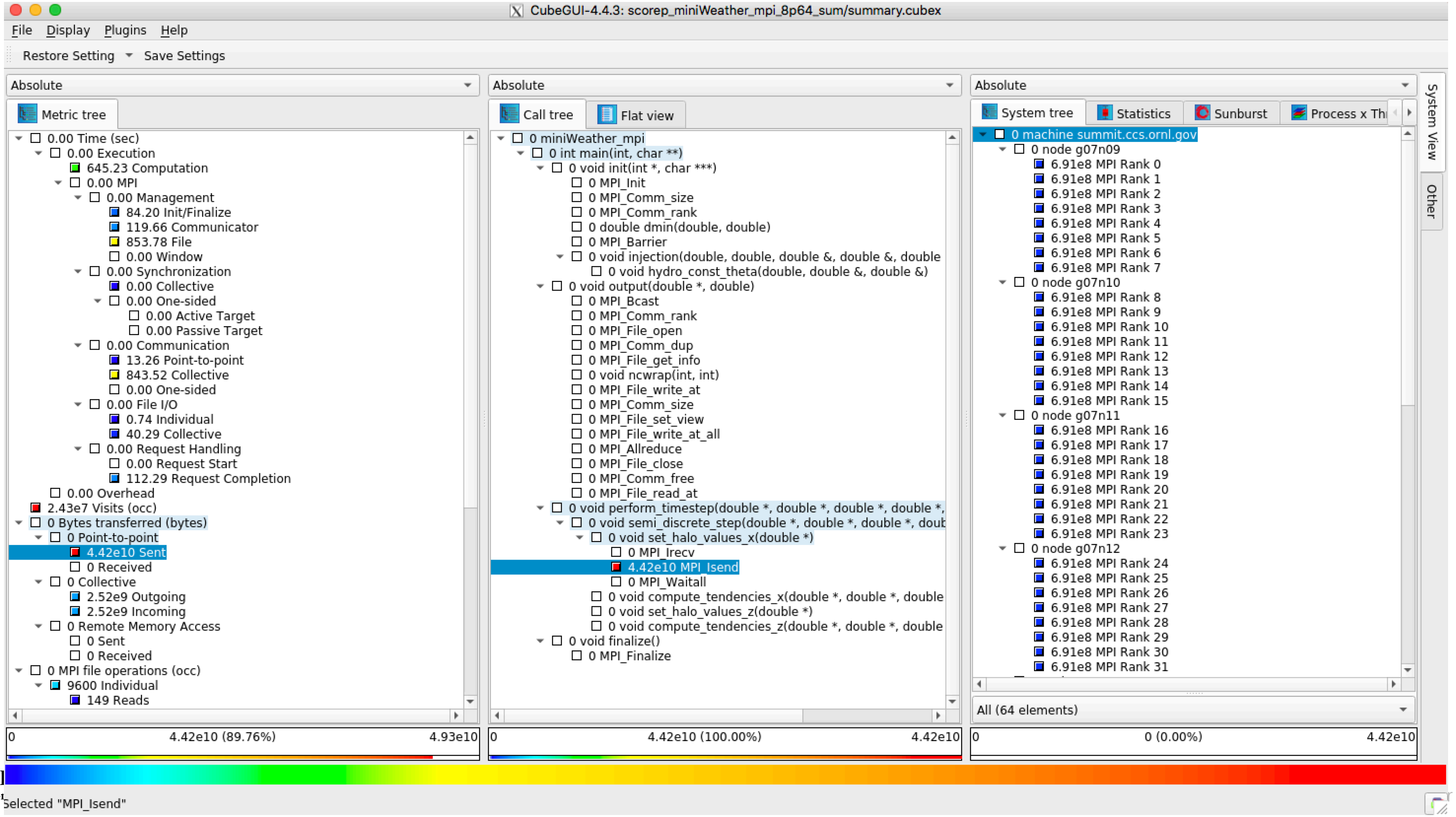


MPI_Waitall variation

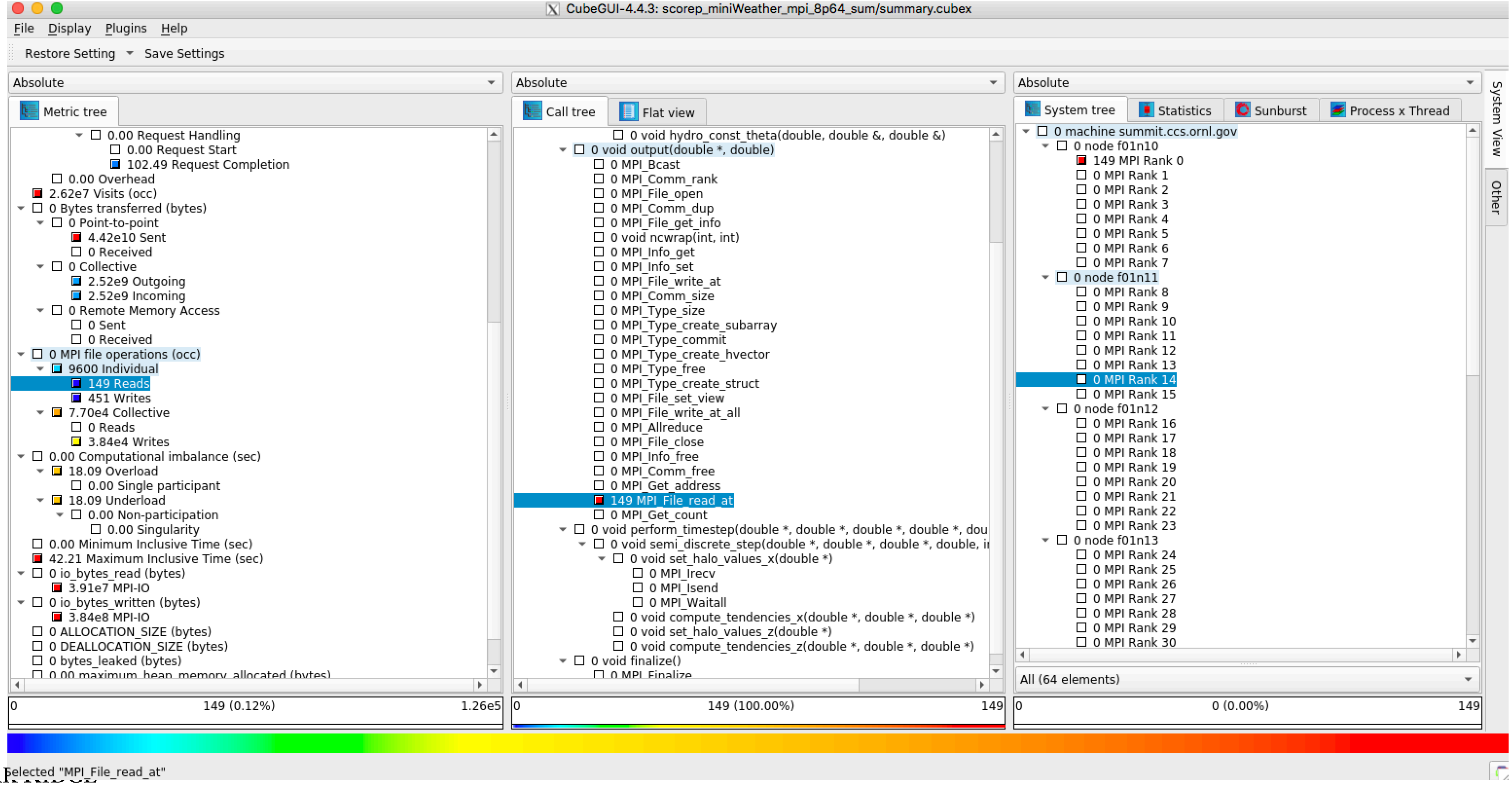


OAK RIDGE National Laboratory

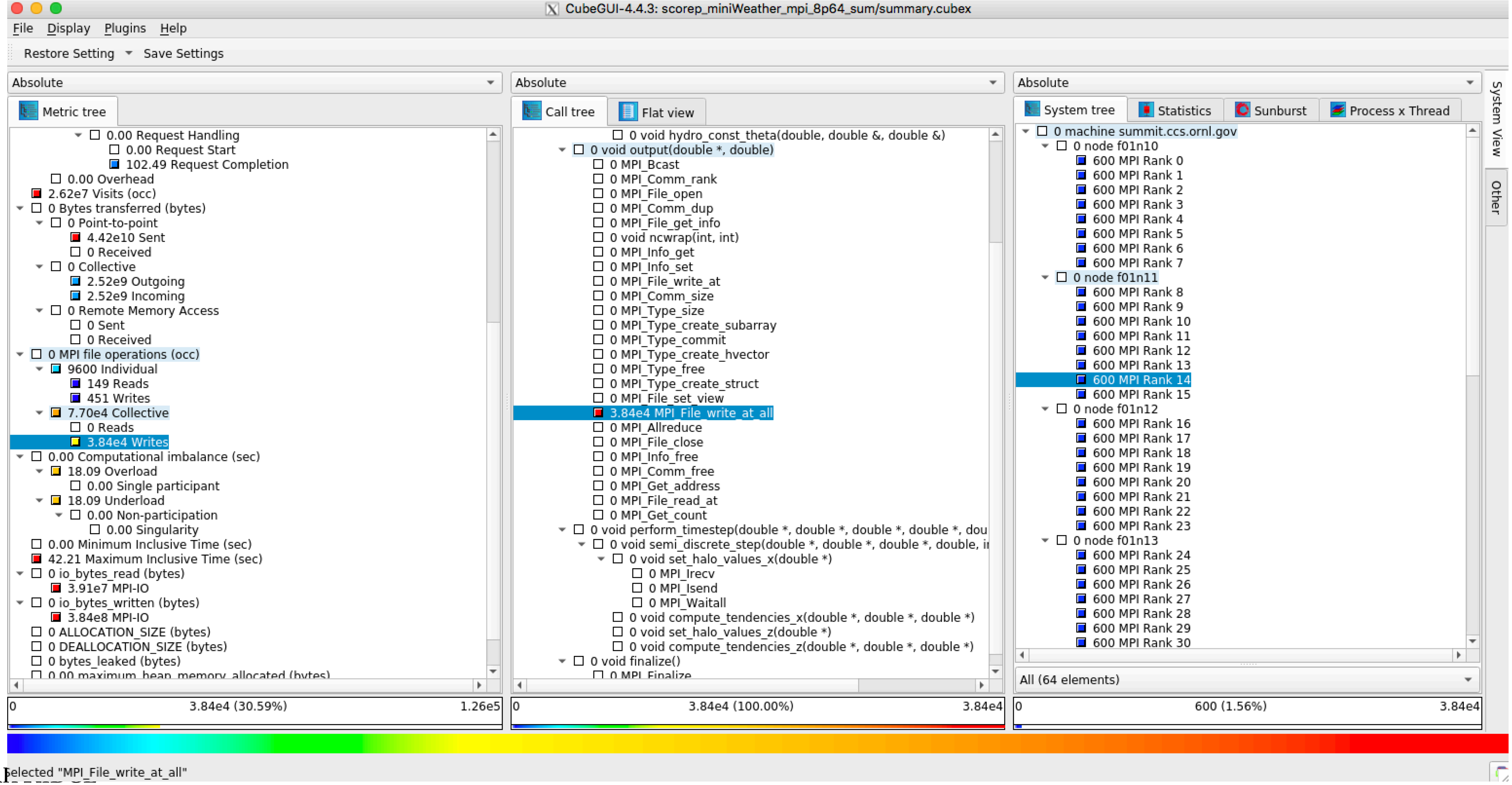
Information about transferred data



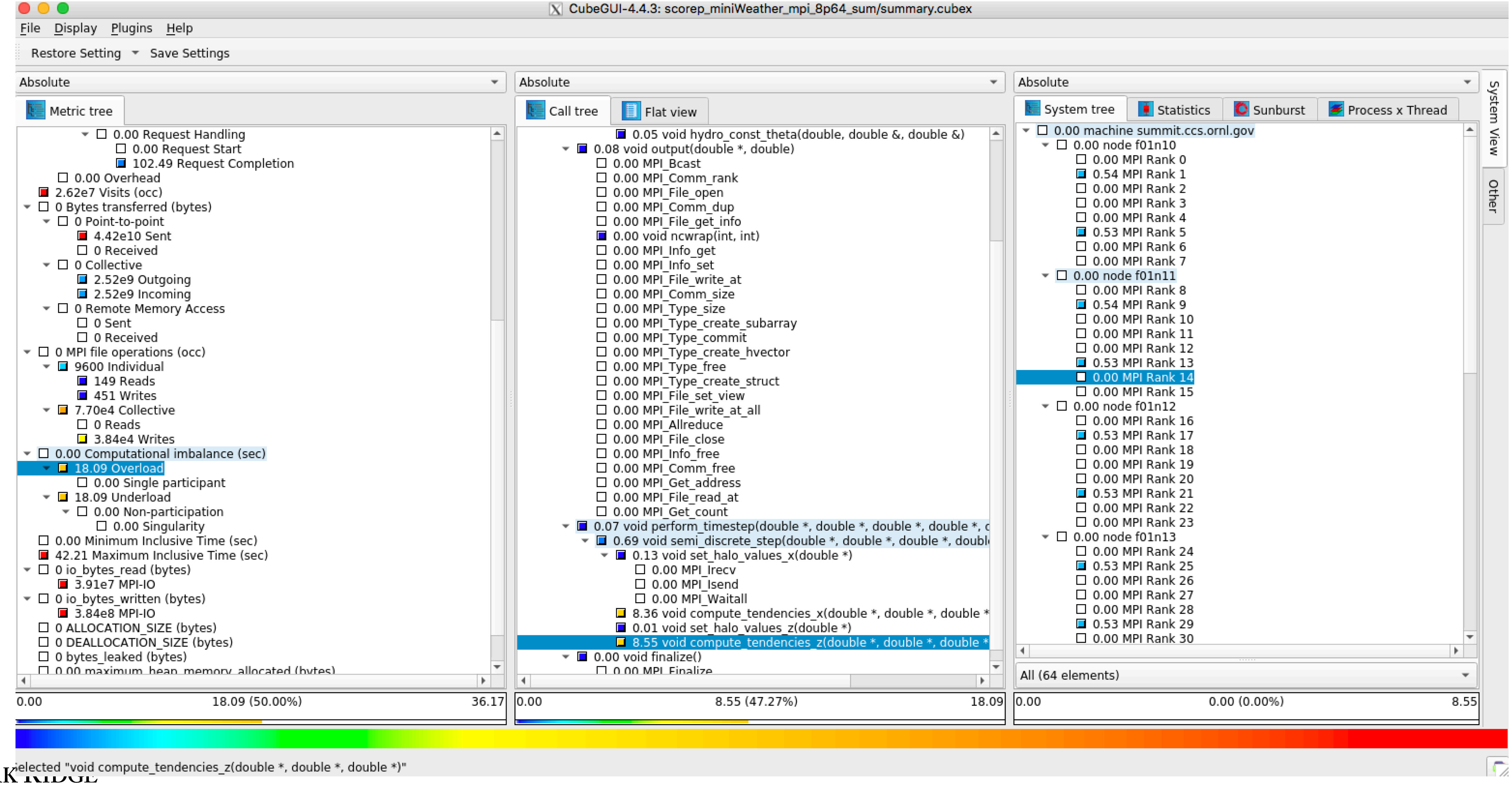
Read-Individual operations



Write-Collective operations

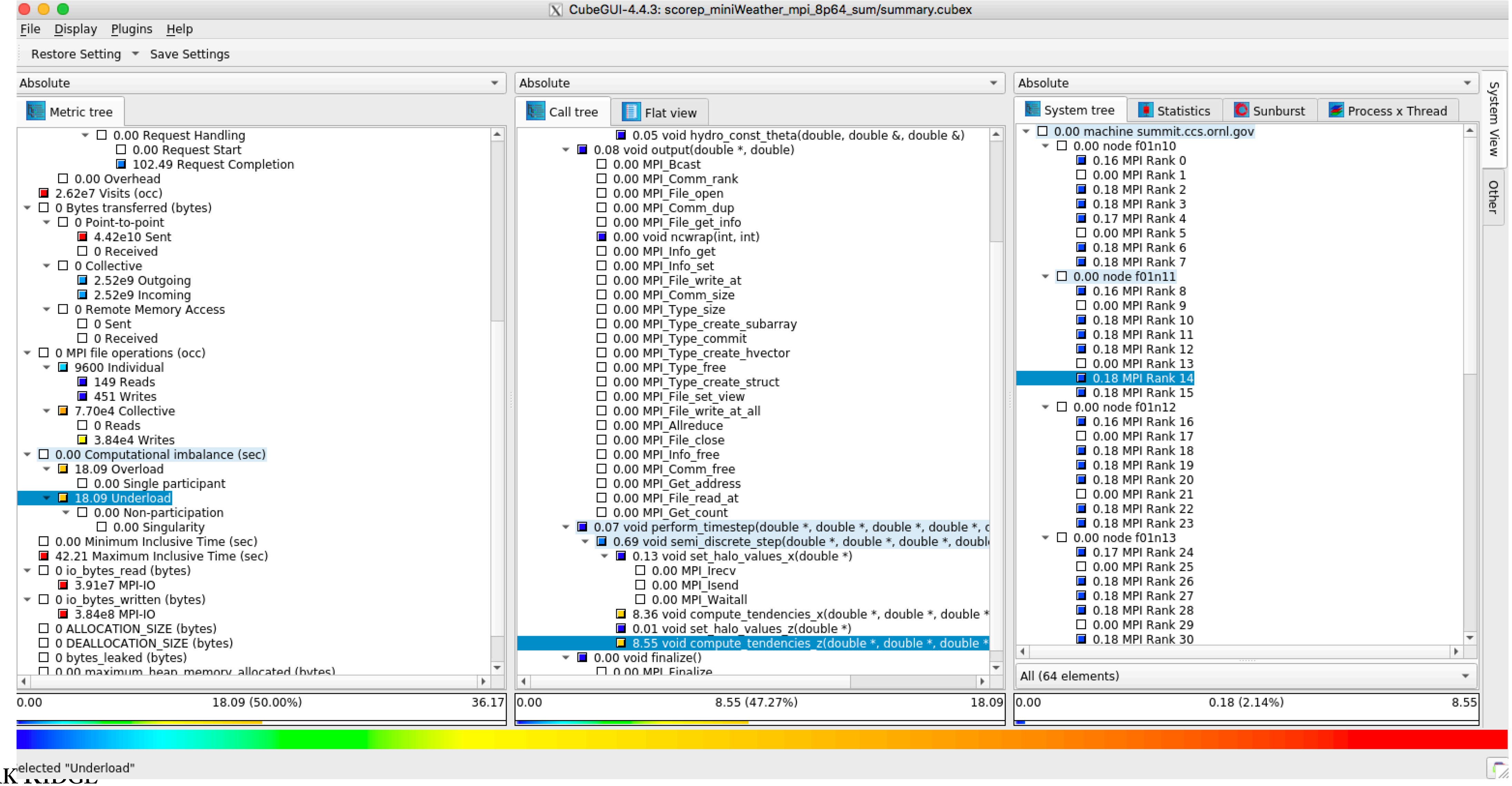


Computational imbalance - Overload



OAK RIDGE National Laboratory

Computational imbalance - Underload

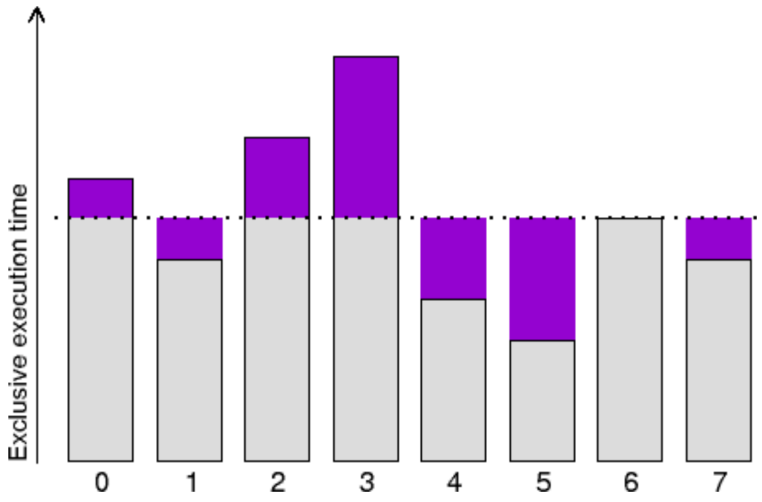


Computational imbalace

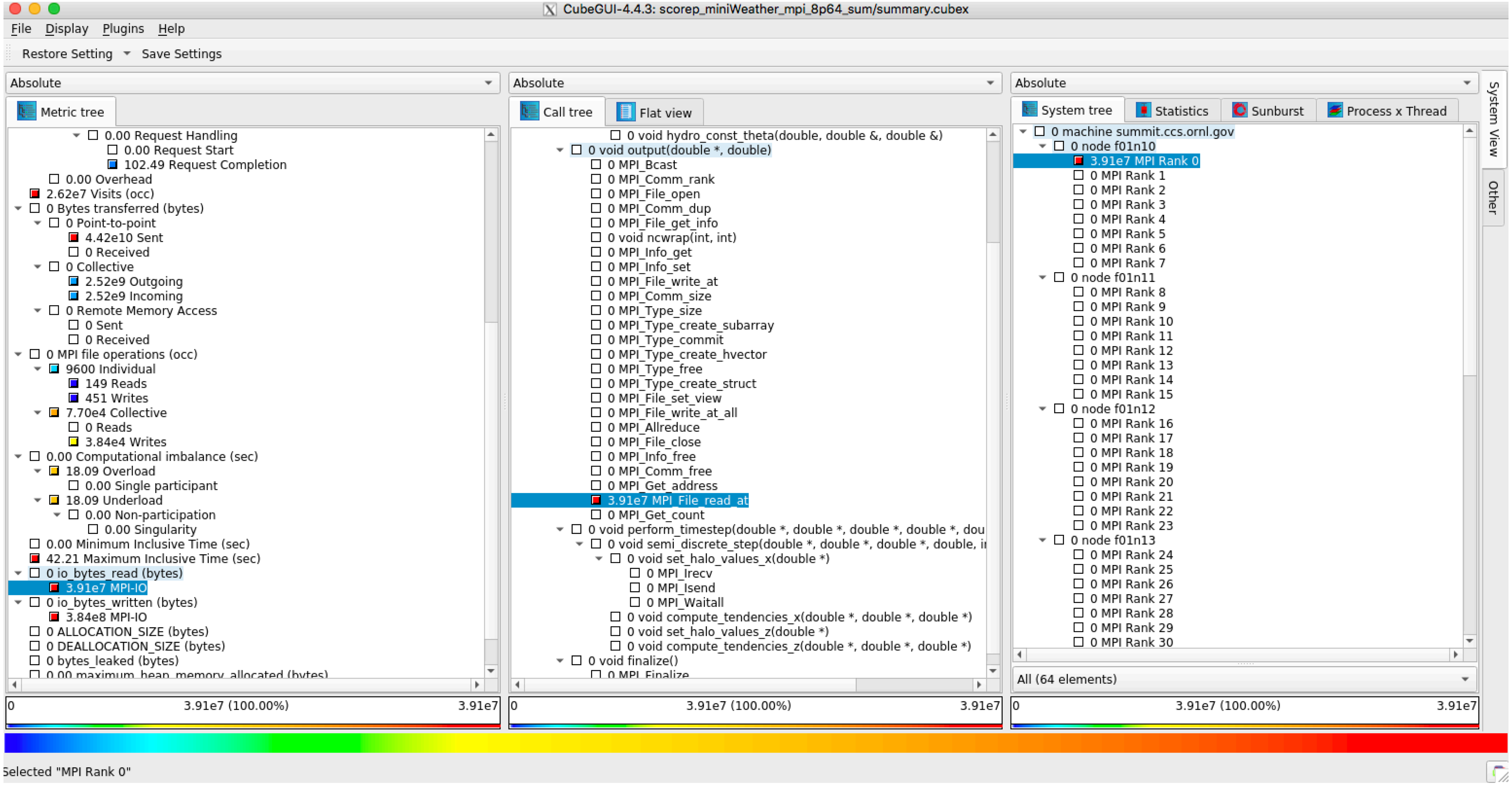
Computational load imbalance heuristic

Description:

This simple heuristic allows to identify computational load imbalances and is calculated for each (call-path, process/thread) pair. Its value represents the absolute difference to the average exclusive execution time. This average value is the aggregated exclusive time spent by all processes/threads in this call-path, divided by the number of processes/threads visiting it.

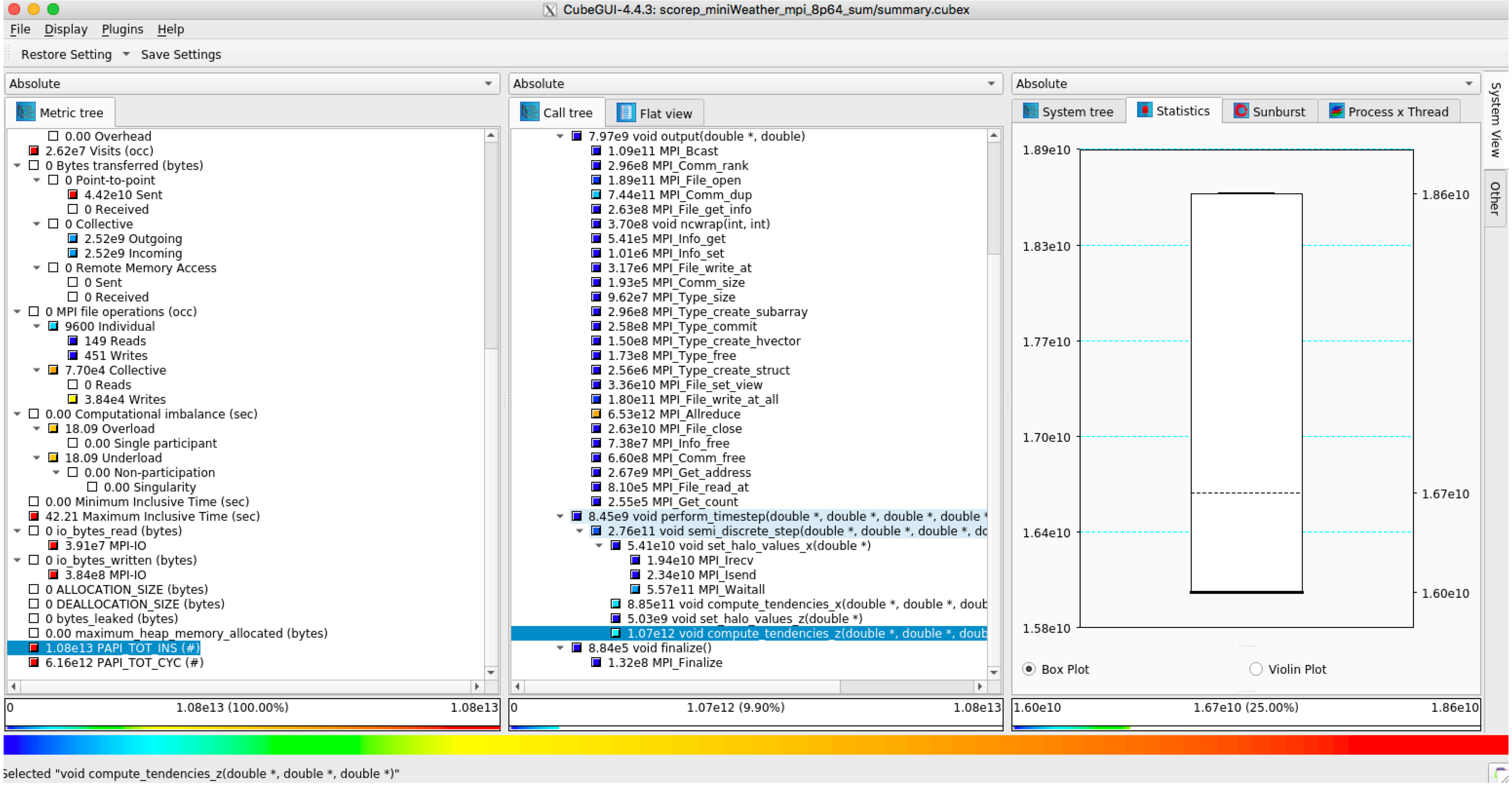


Bytes read



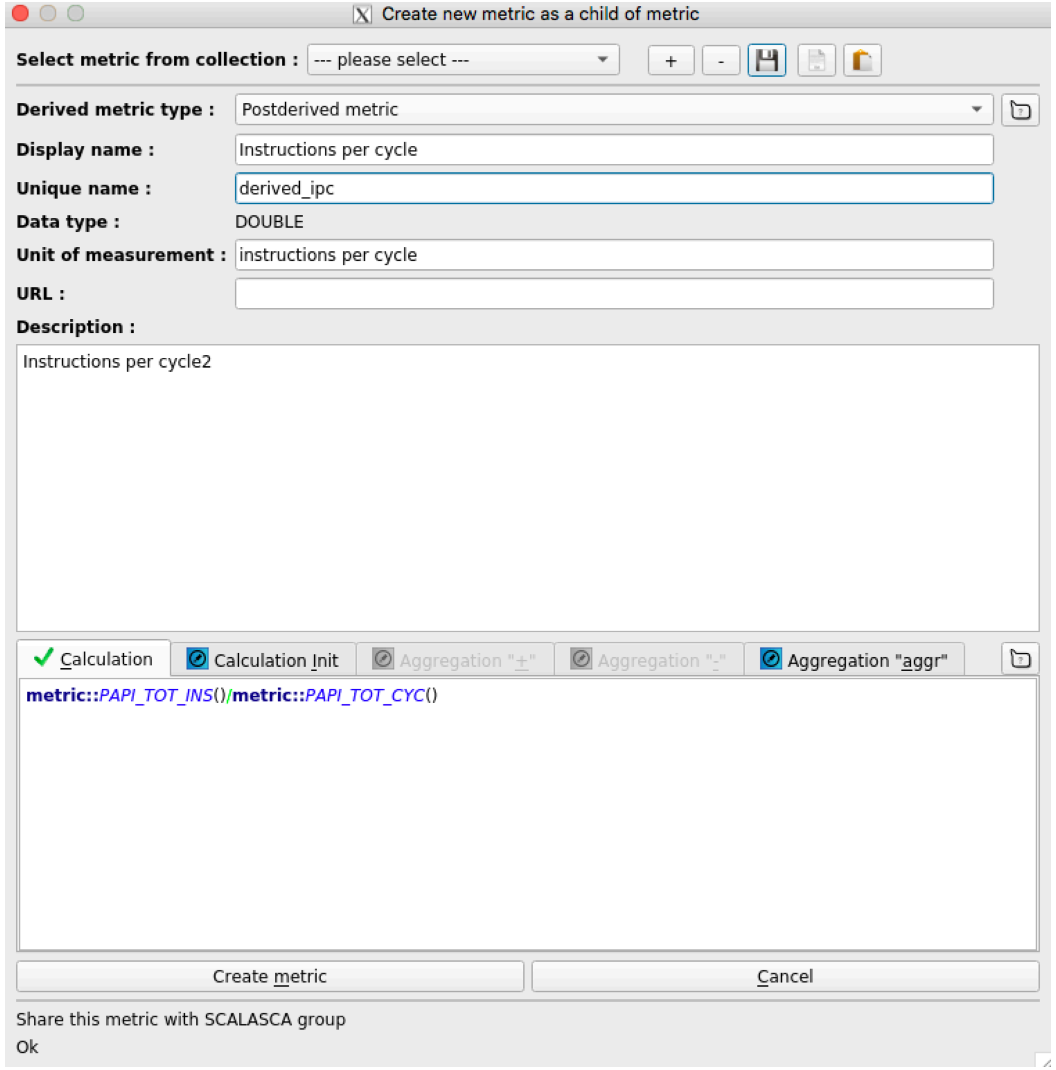
CubeGUI-4.4.3: scorep_miniWeather_mpi_8p64_sum/summary.cubex
File Display Plugins Help
Restore Setting Save Settings
Absolute
Metric tree
0.00 Request Handling
0.00 Request Start
102.49 Request Completion
0.00 Overhead
2.62e7 Visits (occ)
0 Bytes transferred (bytes)
0 Point-to-point
4.42e10 Sent
0 Received
0 Collective
2.52e9 Outgoing
2.52e9 Incoming
0 Remote Memory Access
0 Sent
0 Received
0 MPI file operations (occ)
9600 Individual
149 Reads
451 Writes
7.70e4 Collective
0 Reads
3.84e4 Writes
0.00 Computational imbalance (sec)
18.09 Overload
0.00 Single participant
18.09 Underload
0.00 Non-participation
0.00 Singularity
0.00 Minimum Inclusive Time (sec)
42.21 Maximum Inclusive Time (sec)
0 io_bytes_read (bytes)
3.91e7 MPI-IO
0 io_bytes_written (bytes)
3.84e8 MPI-IO
0 ALLOCATION_SIZE (bytes)
0 DEALLOCATION_SIZE (bytes)
0 bytes_leaked (bytes)
Call tree
Flat view
0 void hydro_const_theta(double, double &, double &)
0 void output(double *, double)
0 MPI_Bcast
0 MPI_Comm_rank
0 MPI_File_open
0 MPI_Comm_dup
0 MPI_File_get_info
0 void ncwrap(int, int)
0 MPI_Info_get
0 MPI_Info_set
0 MPI_File_write_at
0 MPI_Comm_size
0 MPI_Type_size
0 MPI_Type_create_subarray
0 MPI_Type_commit
0 MPI_Type_create_hvector
0 MPI_Type_free
0 MPI_Type_create_struct
0 MPI_File_set_view
0 MPI_File_write_at_all
0 MPI_Allreduce
0 MPI_File_close
0 MPI_Info_free
0 MPI_Comm_free
0 MPI_Get_address
3.91e7 MPI_File_read_at
0 MPI_Get_count
0 void perform_timestep(double *, double *, double *, double *, dou
0 void semi_discrete_step(double *, double *, double *, double, i
0 void set_halo_values_x(double *)
0 MPI_Irecv
0 MPI_Isend
0 MPI_Waitall
0 void compute_tendencies_x(double *, double *, double *)
0 void set_halo_values_z(double *)
0 void compute_tendencies_z(double *, double *, double *)
0 void finalize()
System tree
Statistics
Sunburst
Process x Thread
0 machine summit.ccs.ornl.gov
0 node f01n10
3.91e7 MPI Rank 0
0 MPI Rank 1
0 MPI Rank 2
0 MPI Rank 3
0 MPI Rank 4
0 MPI Rank 5
0 MPI Rank 6
0 MPI Rank 7
0 node f01n11
0 MPI Rank 8
0 MPI Rank 9
0 MPI Rank 10
0 MPI Rank 11
0 MPI Rank 12
0 MPI Rank 13
0 MPI Rank 14
0 MPI Rank 15
0 node f01n12
0 MPI Rank 16
0 MPI Rank 17
0 MPI Rank 18
0 MPI Rank 19
0 MPI Rank 20
0 MPI Rank 21
0 MPI Rank 22
0 MPI Rank 23
0 node f01n13
0 MPI Rank 24
0 MPI Rank 25
0 MPI Rank 26
0 MPI Rank 27
0 MPI Rank 28
0 MPI Rank 29
0 MPI Rank 30
All (64 elements)
System View
Other
0
3.91e7 (100.00%)
3.91e7
Selected "MPI Rank 0"

Instructions

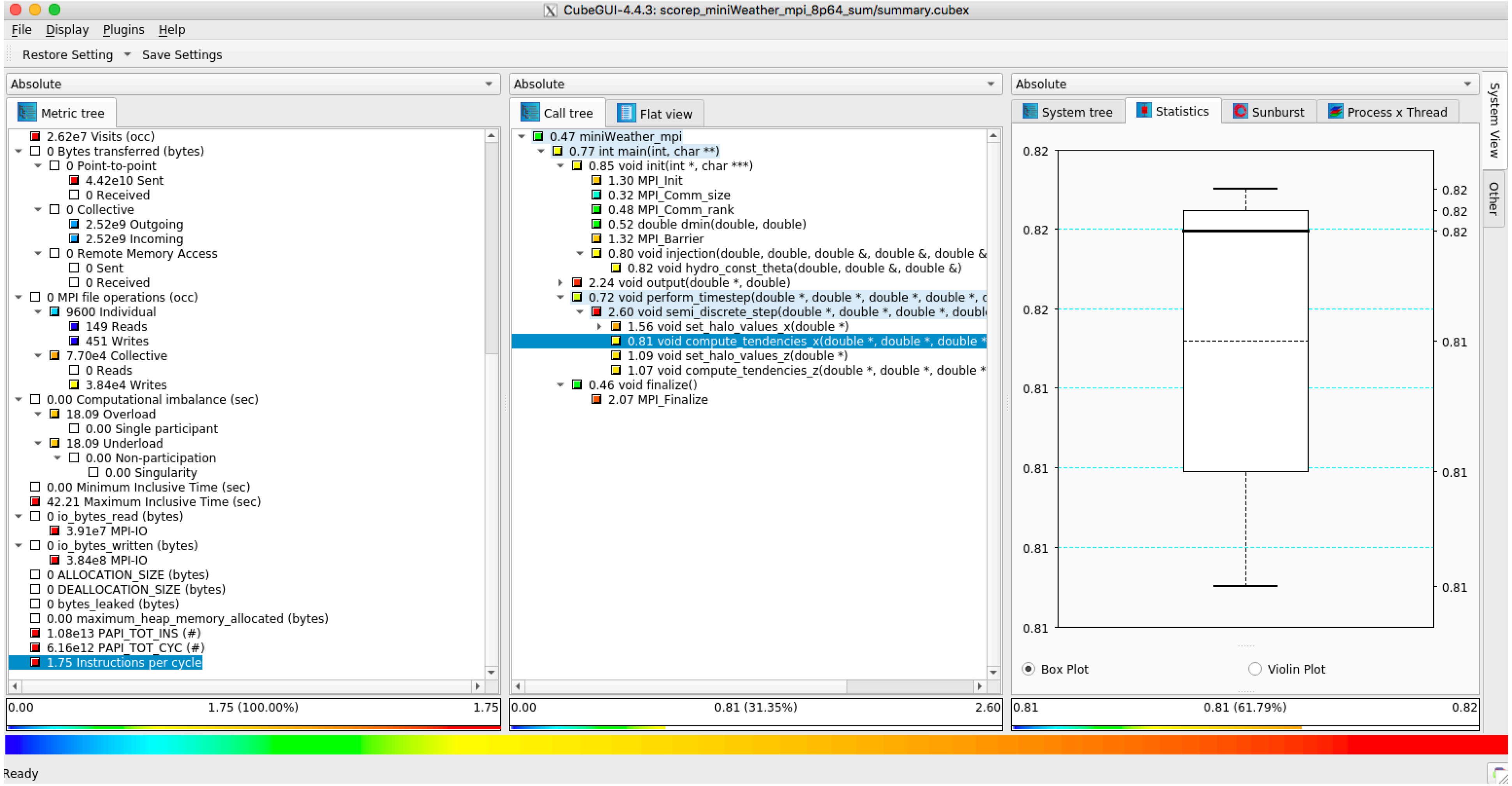


CUBE4 – Derived metrics – Instructions per Cycle

- Right click on any metric of the metric tree, and select “Edit metric” -> Create derived metric -> “as a root”



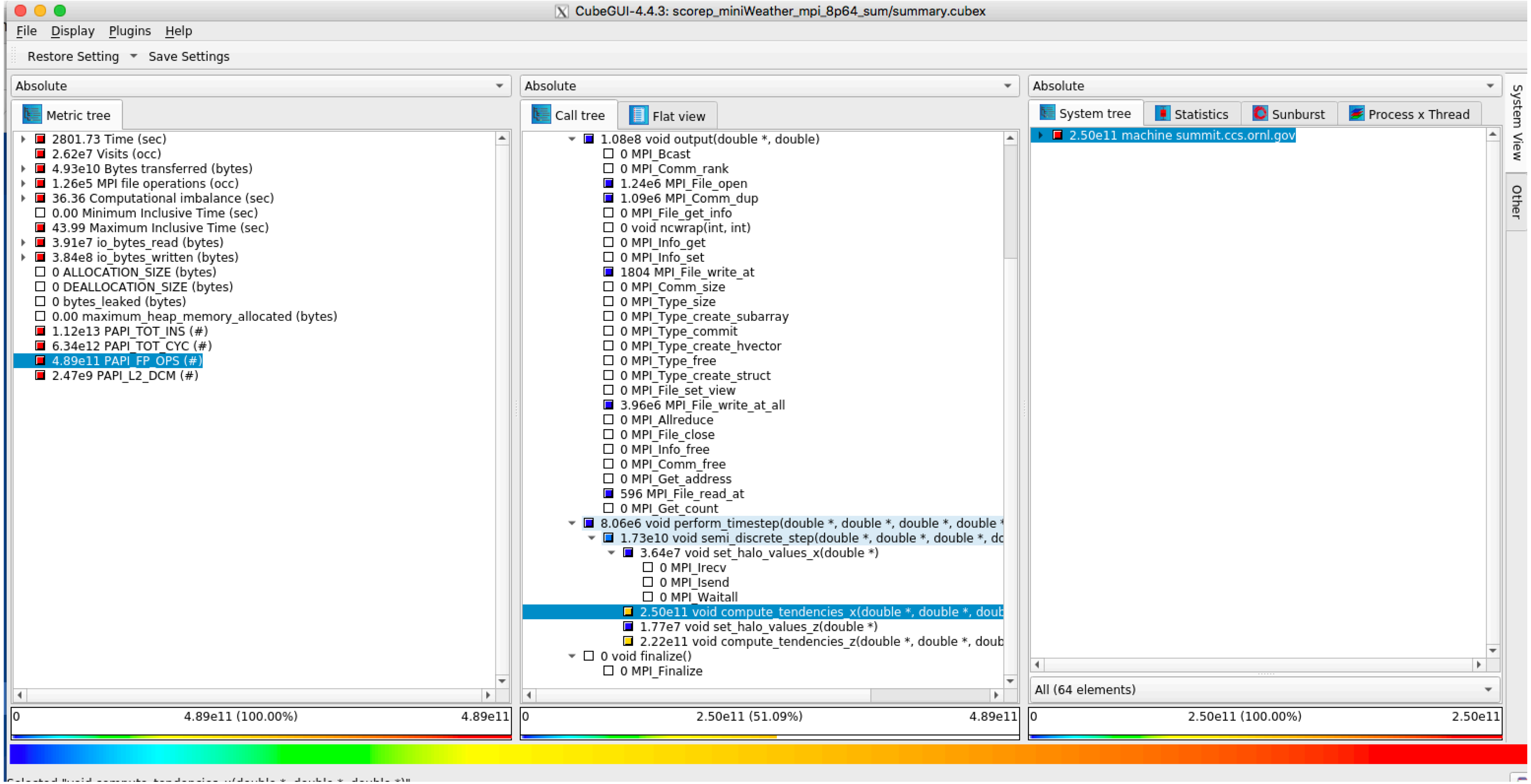
Instructions per cycle (IPC) – Useful computational workload



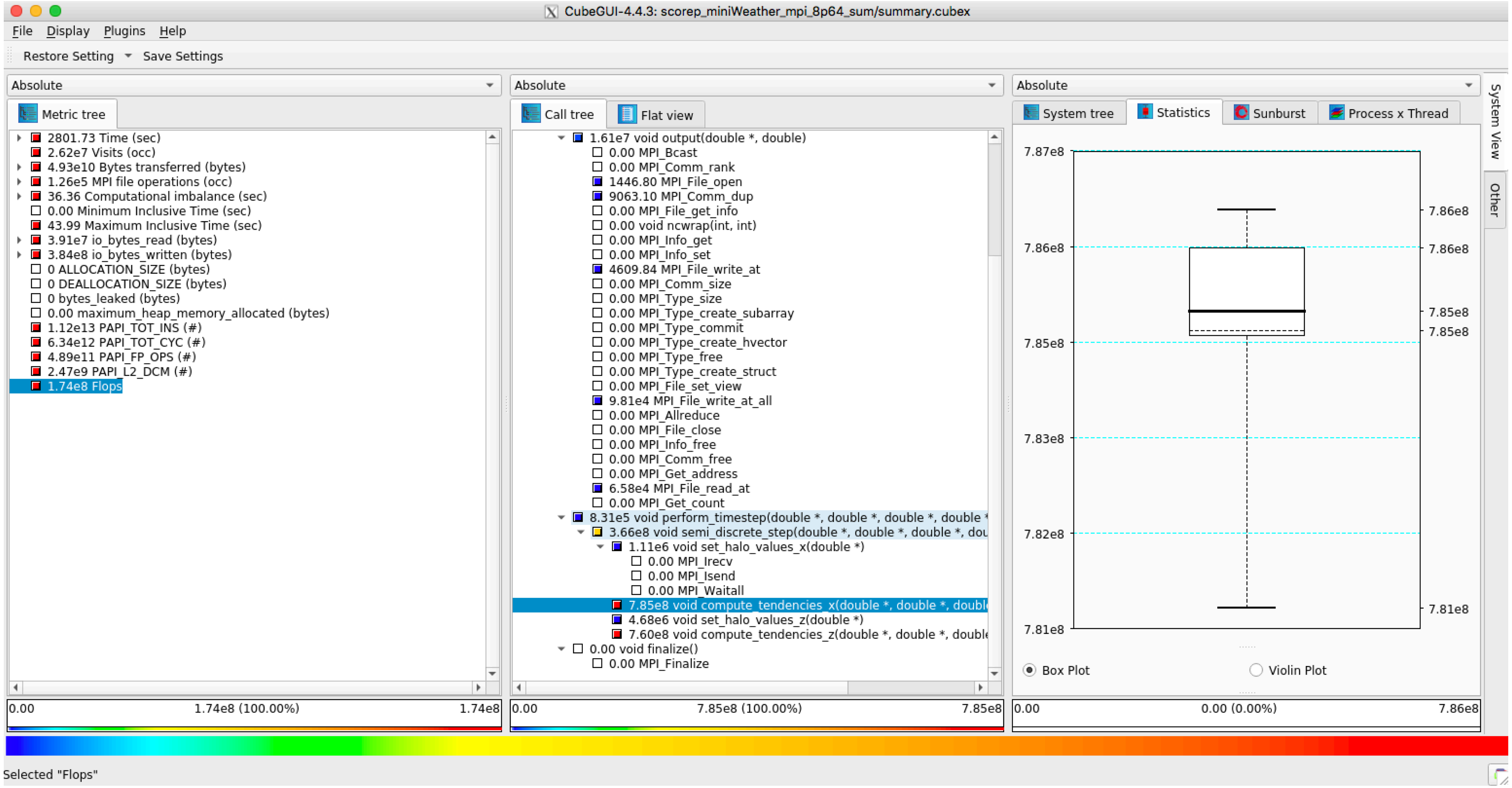
There is no specific rule but codes with IPC less than 1.5, can be improved

OAK RIDGE National Laboratory

Floating operations NOT per second



Derived metric of Floating operations to create the metric Flops



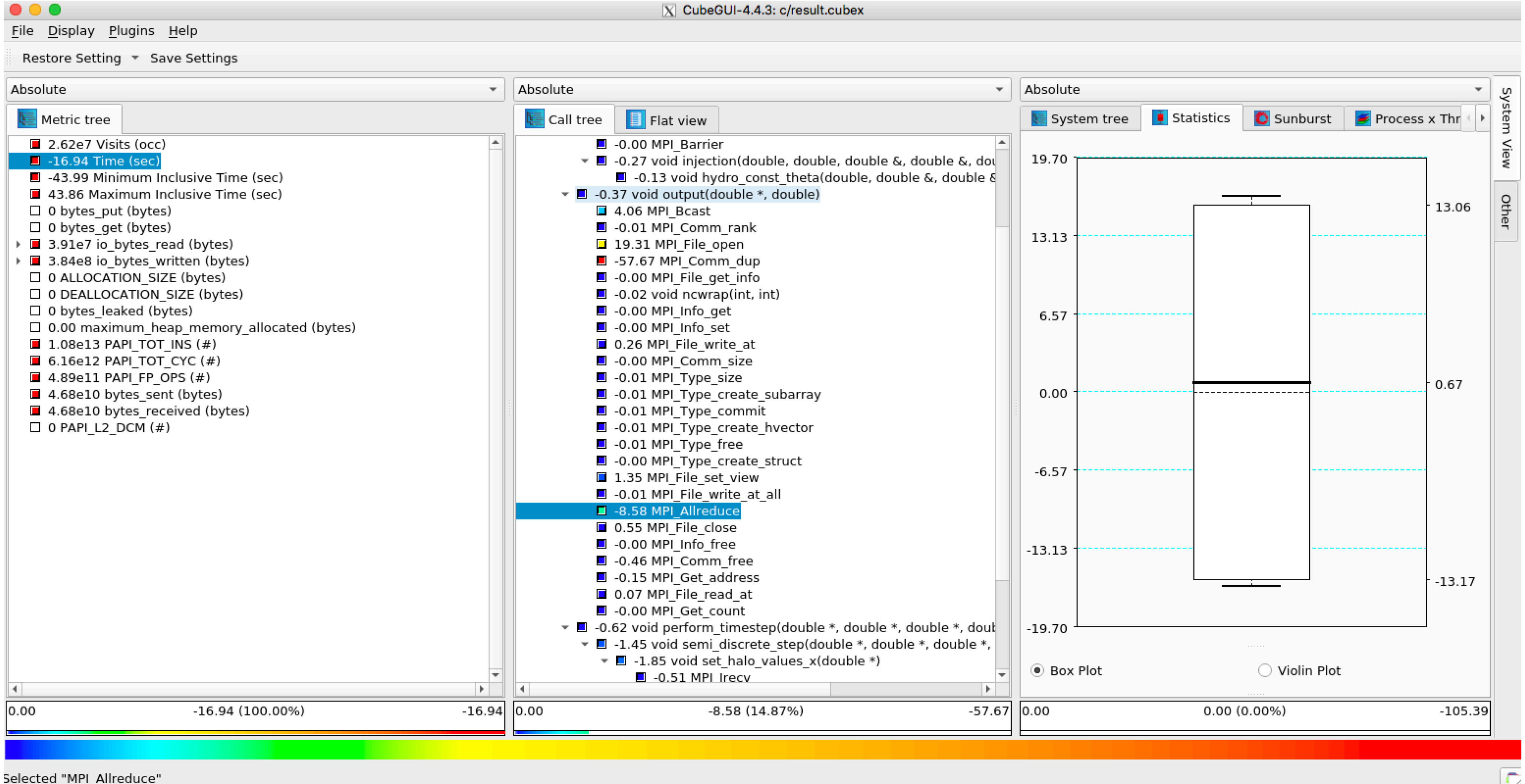
Oak Ridge National Laboratory

Open slide master to edit

Difference between two executions

cube_diff scorep_miniWeather_mpi_4p64_sum1/profile.cubex scorep_miniWeather_mpi_8p64_sum2/profile.cubex -c -o result.cubex

cube result.cubex



Tracing with Scalasca

Tracing with Scalasca

- Tracing can/will cause bigger overhead during the execution of the application
- More information are recorded including timeline
- Scalasca will analyze the trace according to various patterns and it will identify the bottlenecks

How much memory buffer to use for tracing?

- Examine the profiling data

scalasca -examine **-s** /gpfs/alpine/.../scorep_miniWeather_mpi_8p64_sum

INFO: Score report written to /gpfs/alpine/…/scorep_miniWeather_mpi_8p64_sum/**scorep.score**

- head /gpfs/alpine/…/scorep_miniWeather_mpi_8p64_sum/**scorep.score**

Estimated aggregate size of event trace: 978MB

Estimated requirements for largest trace buffer (max_buf): 16MB

Estimated memory requirements (SCOREP_TOTAL_MEMORY): **18MB**

- Add in your submission script (include ~10% extra):

export SCOREP_TOTAL_MEMORY=20MB

Overhead

- You need to declare enough size of SCOREP_TOTAL_MEMORY to avoid flushing of the performance files.
- For our application, non instrumented execution on 1 node takes ~30 seconds, while for profiling and tracing is 45 and 53 seconds respectively, so 50% and 76% overhead.
- The overhead always depends on the application and what you instrument, OpenMP etc.
- We have choice of selective instrumentation or manually profiling filter

How to use Scalasca/Score-P with tracing?

- In your submission script, replace:

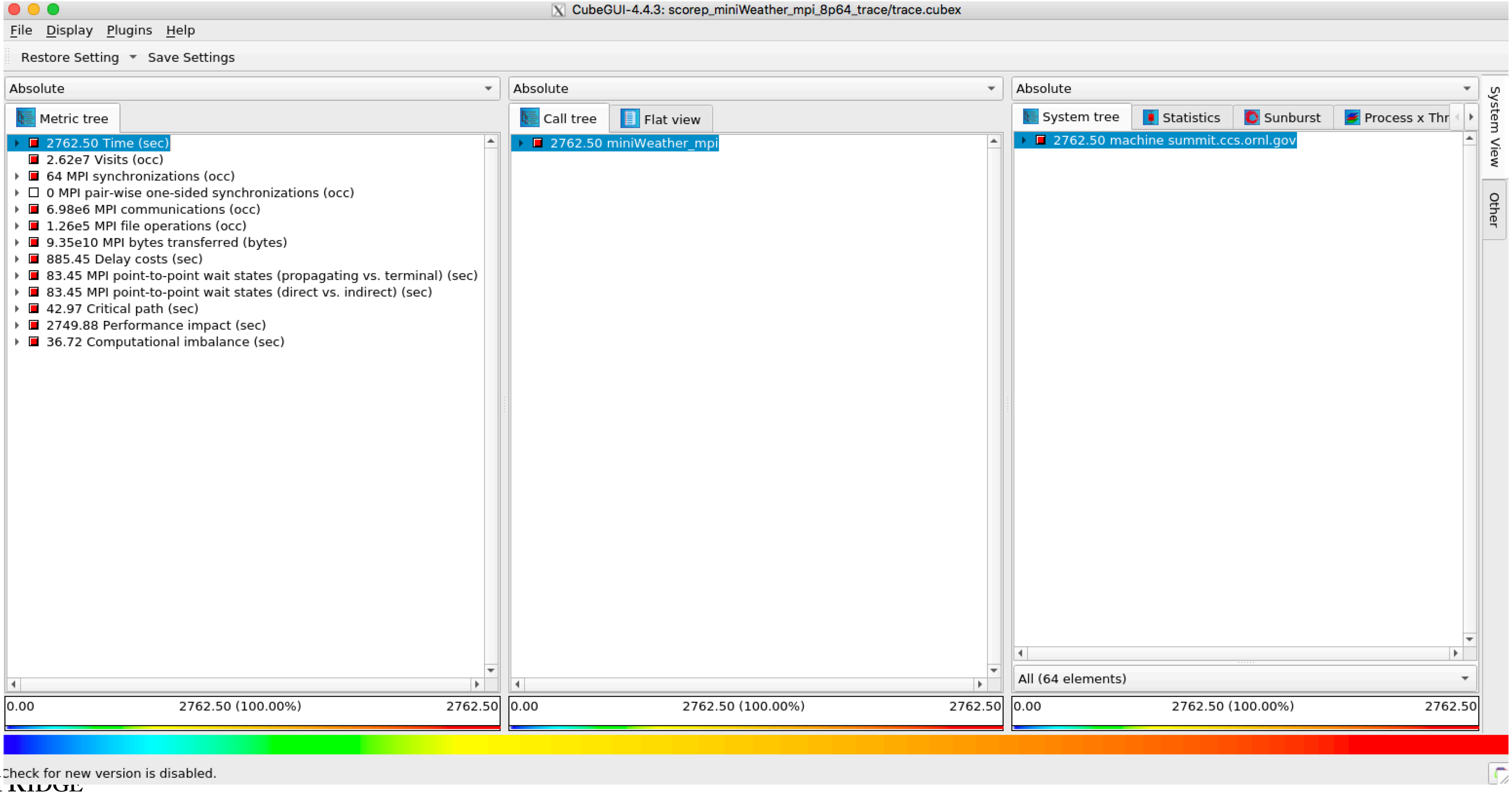
scalasca -analyze jsrun …

with

scalasca -analyze -q -t jsrun

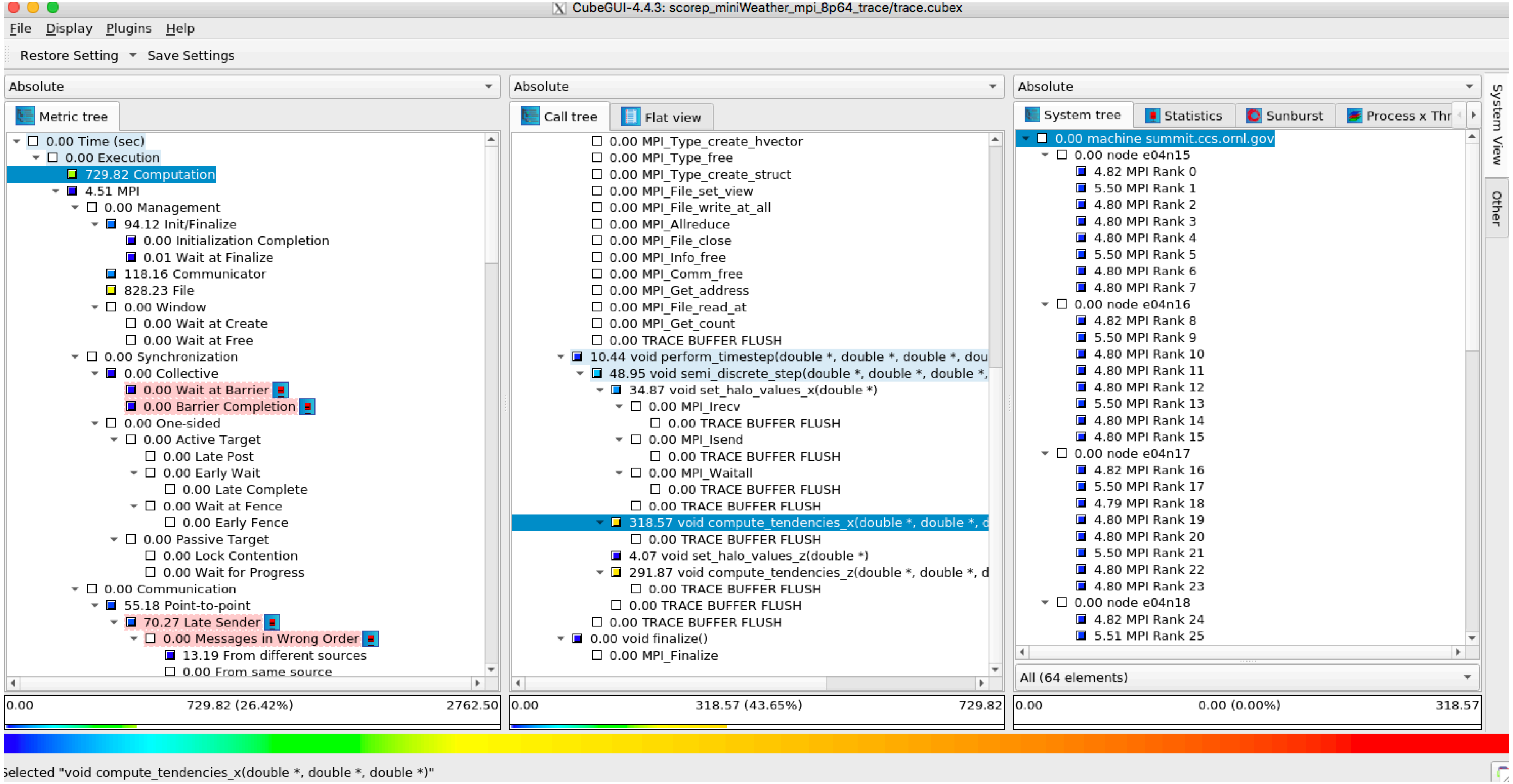
- The –q disables the profiling.

Initial view with tracing

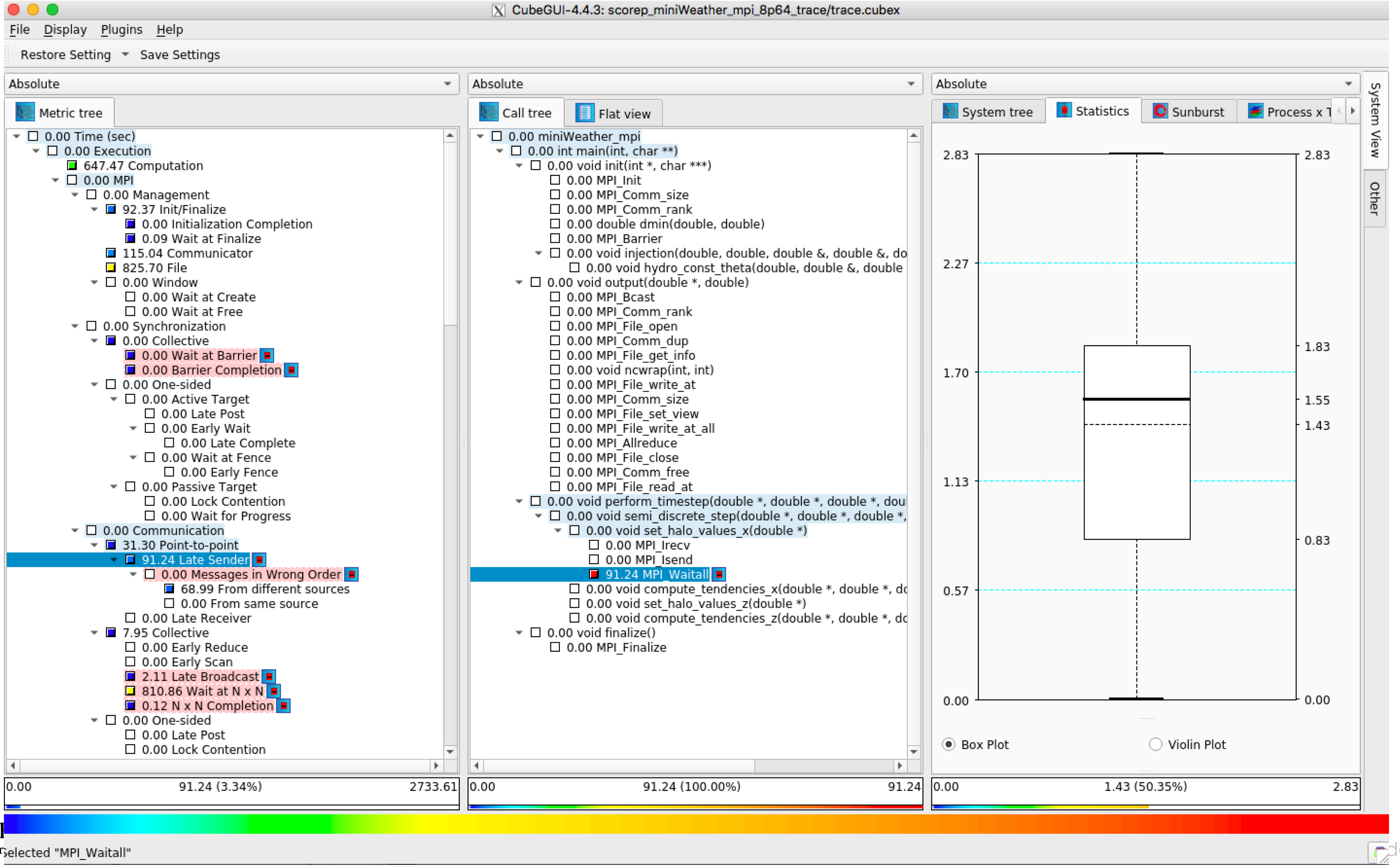


OAK RIDGE National Laboratory

Computation with tracing and expand trees



Late Sender for Point-to-Point communication

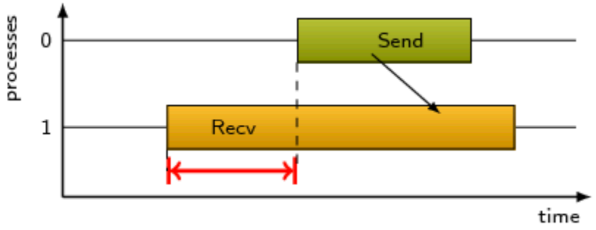


OAK RIDGE National Laboratory

Late Sender - Time

Late Sender Time

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



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

Unit:
Seconds

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

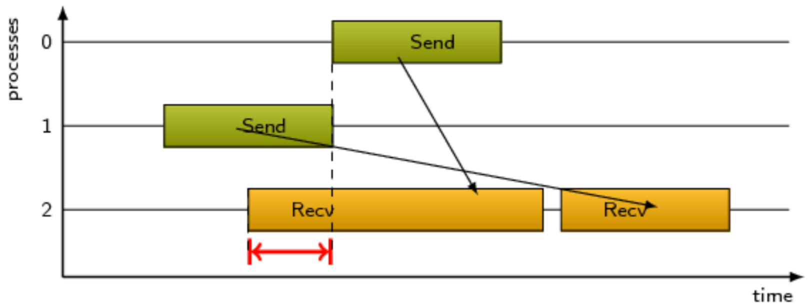
Documentation:
https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/help/scalasca_patterns.html

Late Sender – Wrong Order Time/Different Sources

Late Sender, Wrong Order Time / Different Sources

Description:

This specialization of the *Late Sender, Wrong Order* pattern refers to wrong order situations due to messages received from different source locations.



Unit:

Seconds

Diagnosis:

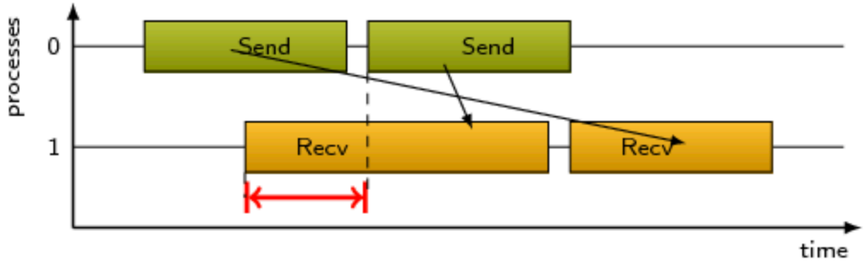
Check the proportion of Point-to-point Receive Communications that are Late Sender, Wrong Order Instances (Communications). Swap the order of receiving from different sources to match the most common ordering. Consider using the wildcard `MPI_ANY_SOURCE` to receive (and process) messages as they arrive from any source rank.

Late Sender – Wrong Order Time/Same source

Late Sender, Wrong Order Time / Same Source

Description:

This specialization of the *Late Sender, Wrong Order* pattern refers to wrong order situations due to messages received from the same source location.



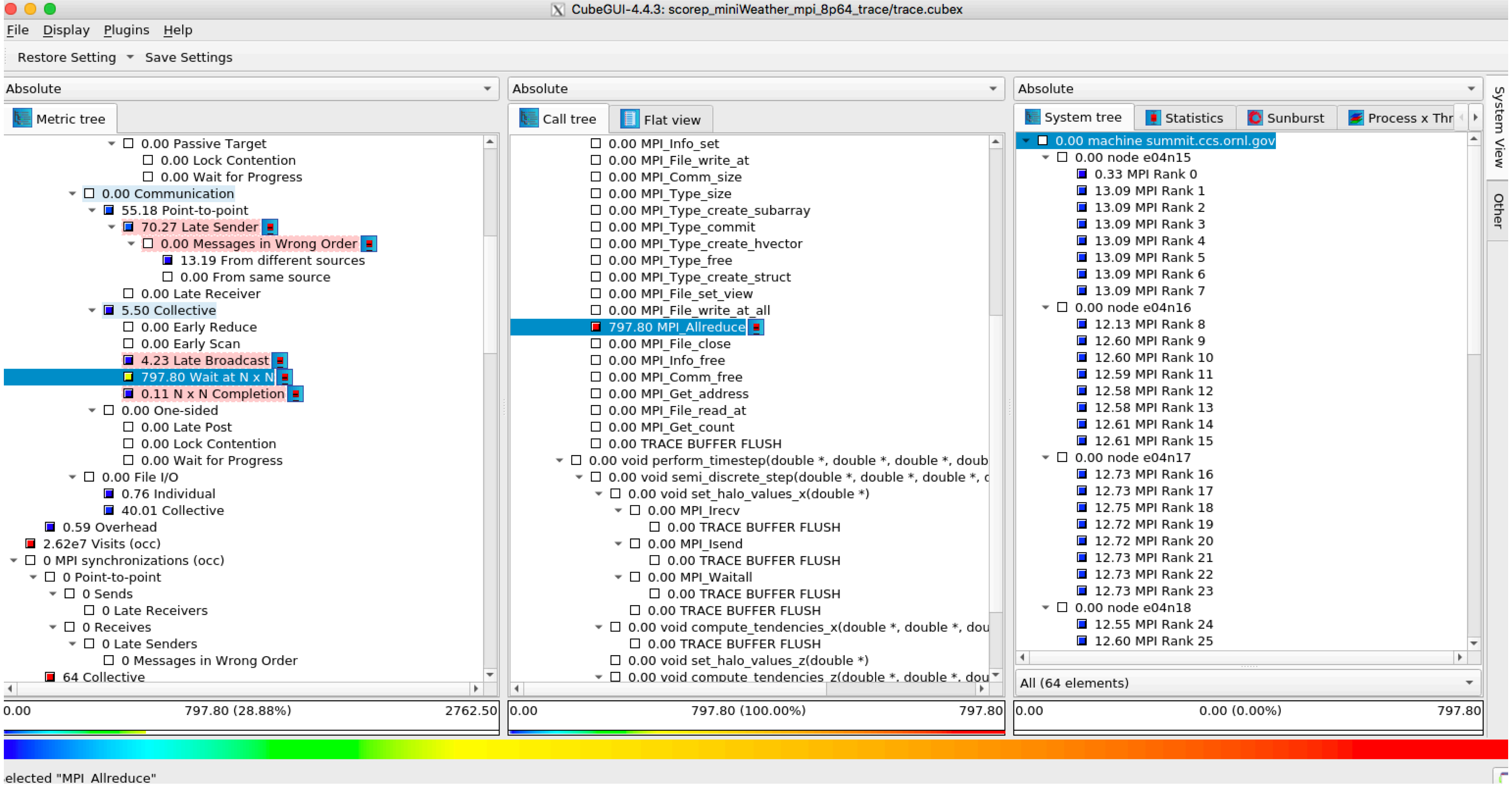
Unit:

Seconds

Diagnosis:

Swap the order of receiving to match the order messages are sent, or swap the order of sending to match the order they are expected to be received. Consider using the wildcard `MPI_ANY_TAG` to receive (and process) messages in the order they arrive from the source.

MPI Wait at N x N Time



Oak Ridge National Laboratory

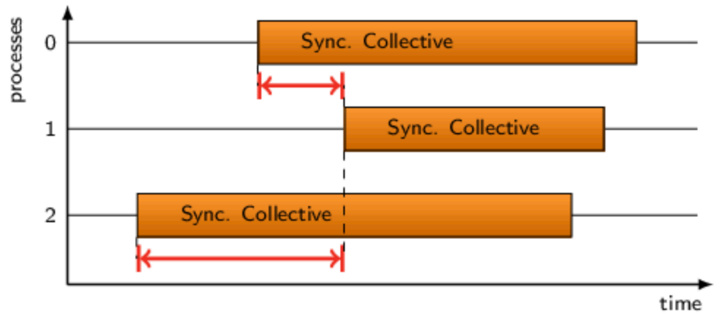
Open slide master to edit

MPI Wait at N x N Time - Explanation

MPI Wait at N x N Time

Description:

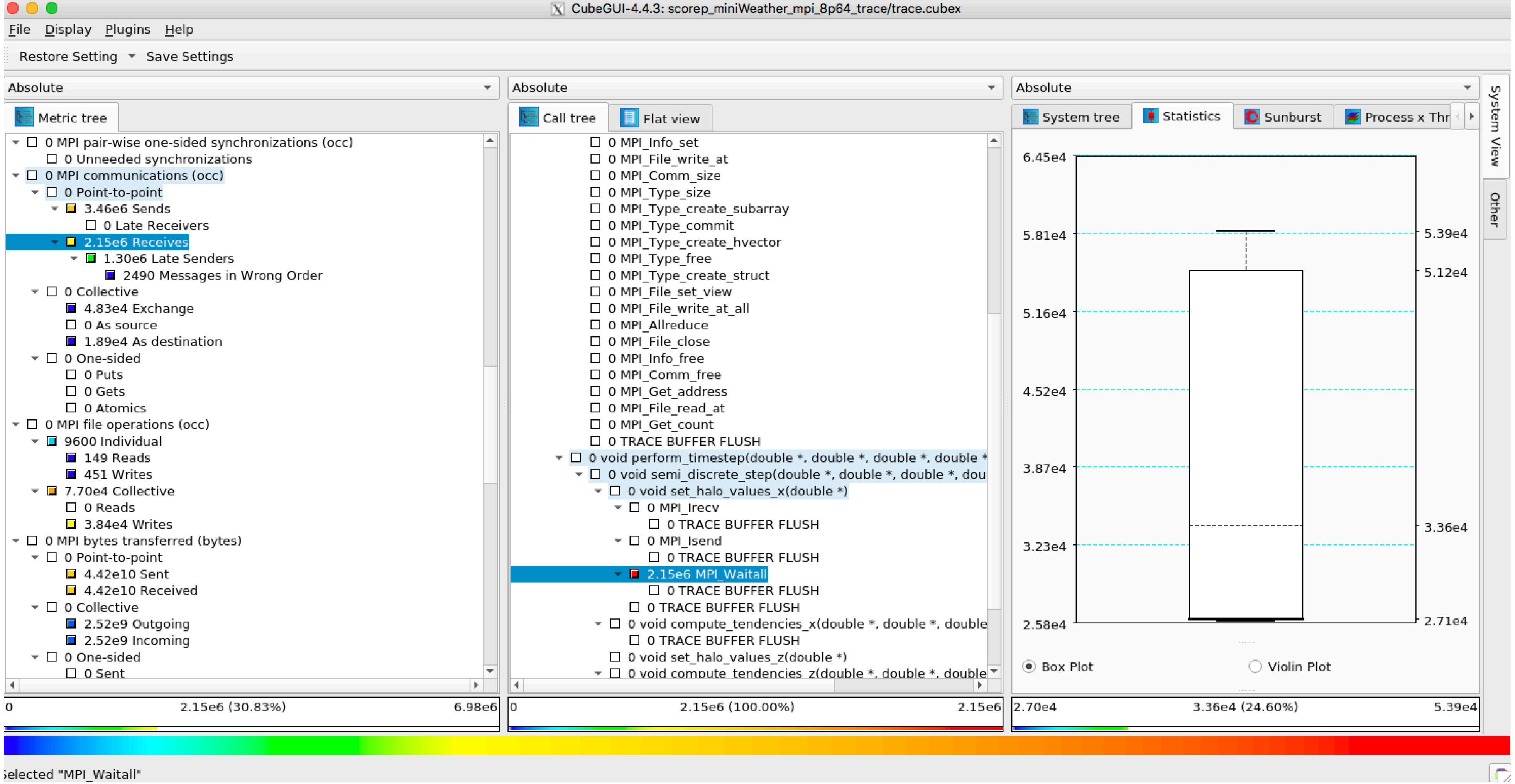
Collective communication operations that send data from all processes to all processes (i.e., n-to-n) exhibit an inherent synchronization among all participants, that is, no process can finish the operation until the last process has started it. This pattern covers the time spent in n-to-n operations until all processes have reached it. It applies to the MPI calls `MPI_Reduce_scatter`, `MPI_Reduce_scatter_block`, `MPI_Allgather`, `MPI_Allgatherv`, `MPI_Allreduce` and `MPI_Alltoall`.



Note that the time reported by this pattern is not necessarily completely waiting time since some processes could – at least theoretically – already communicate with each other while others have not yet entered the operation.

Also note that Scalasca does not yet analyze non-blocking and neighborhood collectives introduced with MPI v3.0.

Number of MPI communications



OAK RIDGE National Laboratory

Open slide master to edit

Short and Long-term delay

Short-term MPI Late Sender Delay Costs

Description:
Short-term delay costs reflect the direct effect of load or communication imbalance on MPI Late Sender wait states.
Unit:
Seconds
Diagnosis:
High short-term delay costs indicate a computation or communication overload in/on the affected call paths and processes/threads. Because of this overload, the affected processes/threads arrive late at subsequent MPI send operations, thus causing Late Sender wait states on the remote processes.

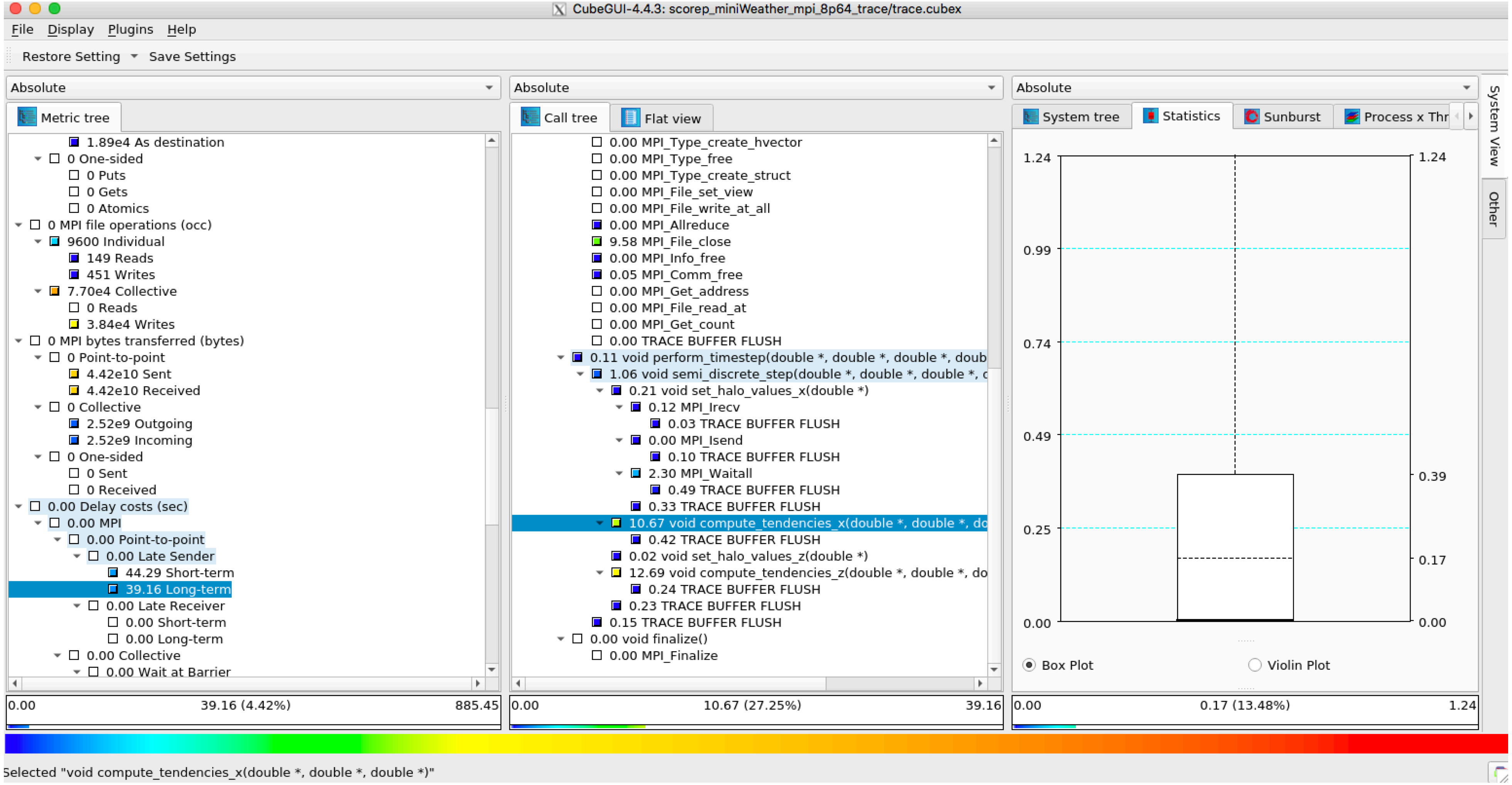
Compare with MPI Late Sender Time to identify an imbalance pattern. Try to reduce workload in the affected call paths. Alternatively, shift workload in the affected call paths from processes/threads with delay costs to processes/threads that exhibit late-sender wait states.

Long-term MPI Late Sender Delay Costs

Description:
Long-term delay costs reflect indirect effects of load or communication imbalance on wait states. That is, they cover waiting time that was caused indirectly by wait states which themselves delay subsequent communication operations.
Unit:
Seconds
Diagnosis:
High long-term delay costs indicate that computation or communication overload in/on the affected call paths and processes/threads has far-reaching effects. That is, the wait states caused by the original computational overload spread along the communication chain to remote locations.

Try to reduce workload in the affected call paths, or shift workload from processes/threads with delay costs to processes/threads that exhibit Late Sender wait states. Try to implement a more asynchronous communication pattern that can compensate for small imbalances, e.g., by using non-blocking instead of blocking communication.

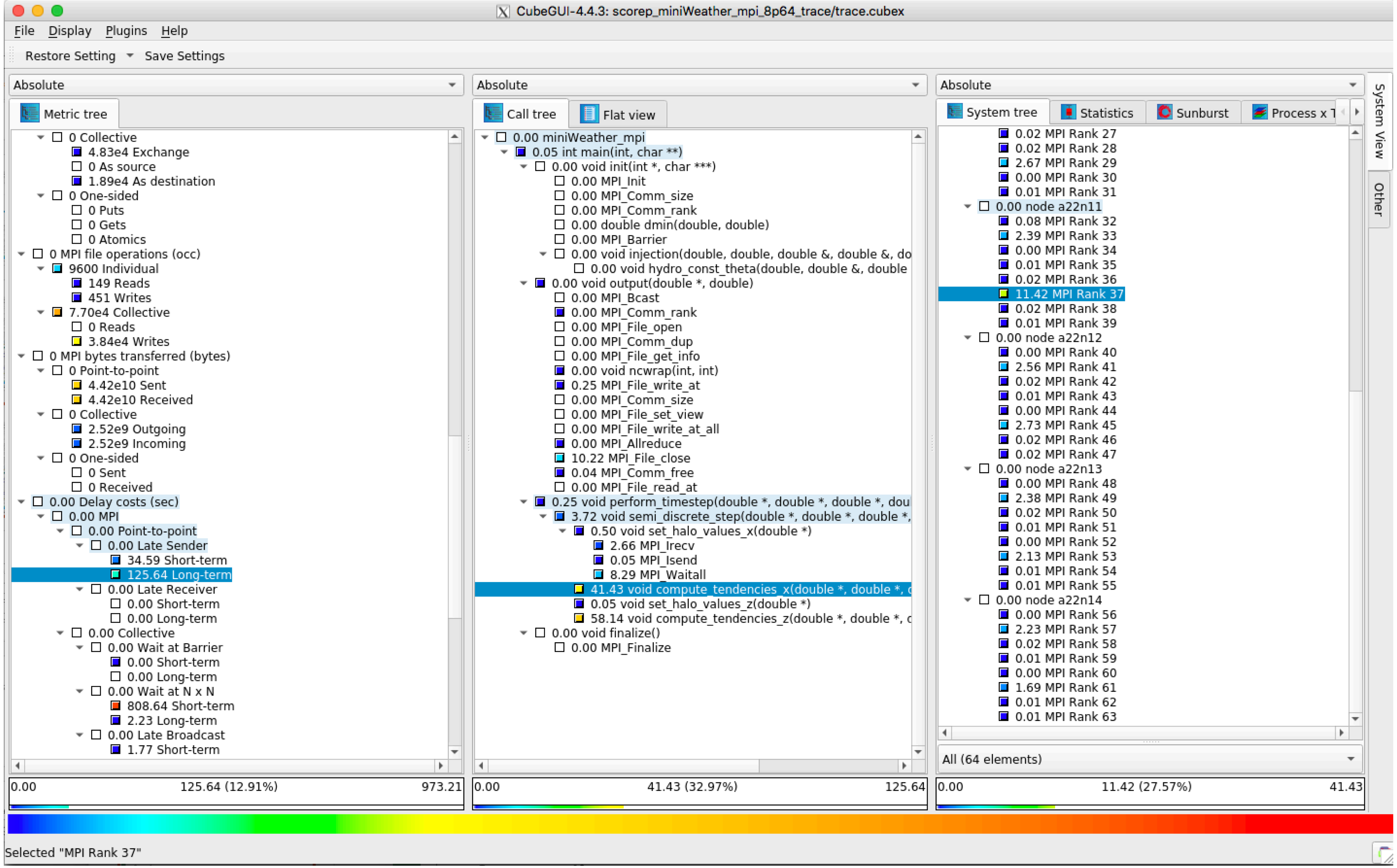
Long-term delay sender



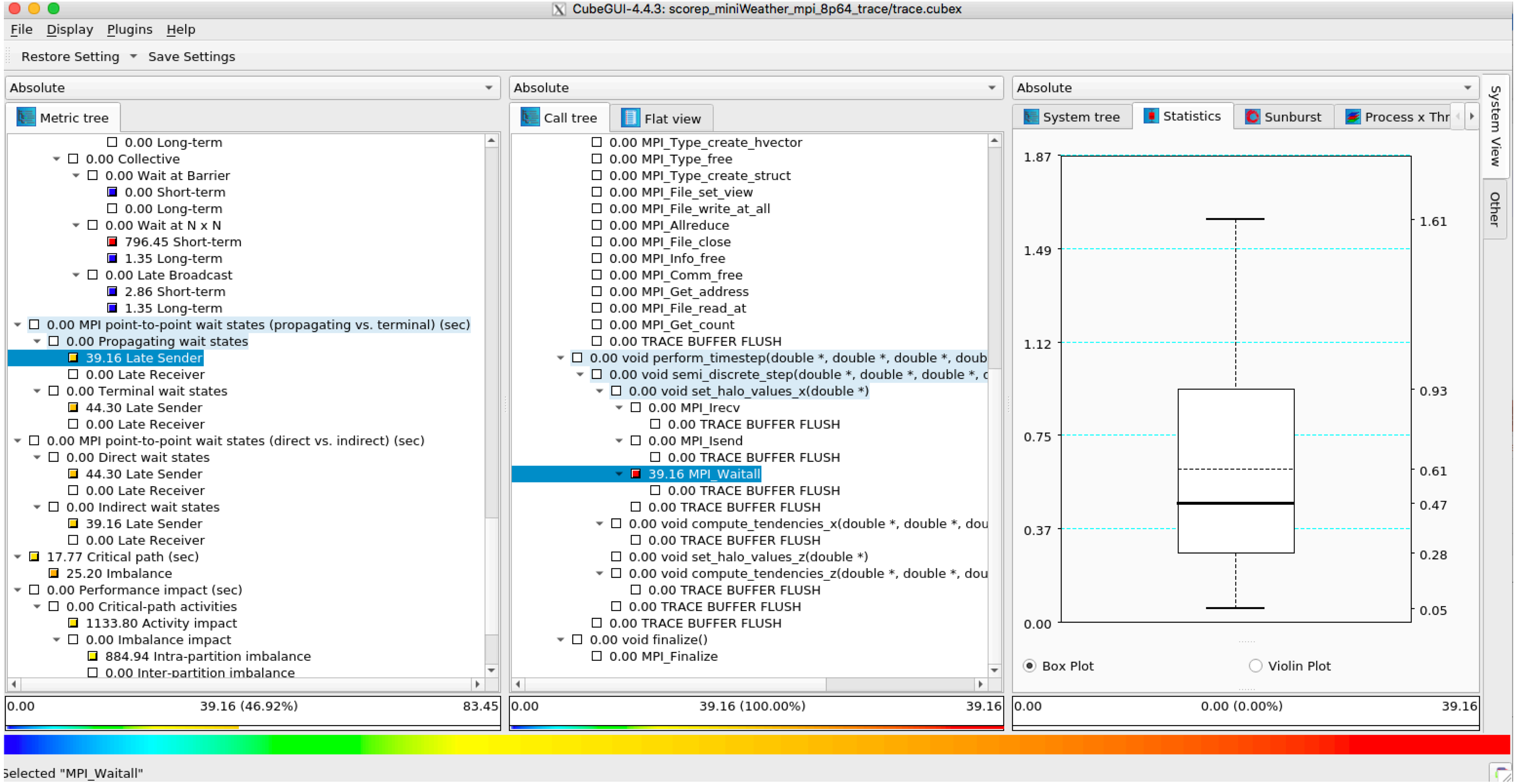
OAK RIDGE National Laboratory

Open slide master to edit

Long-term delay sender



Propagating wait states



Oak Ridge National Laboratory

Tracing start

Propagating MPI Point-to-point Wait States

(only available after remapping)

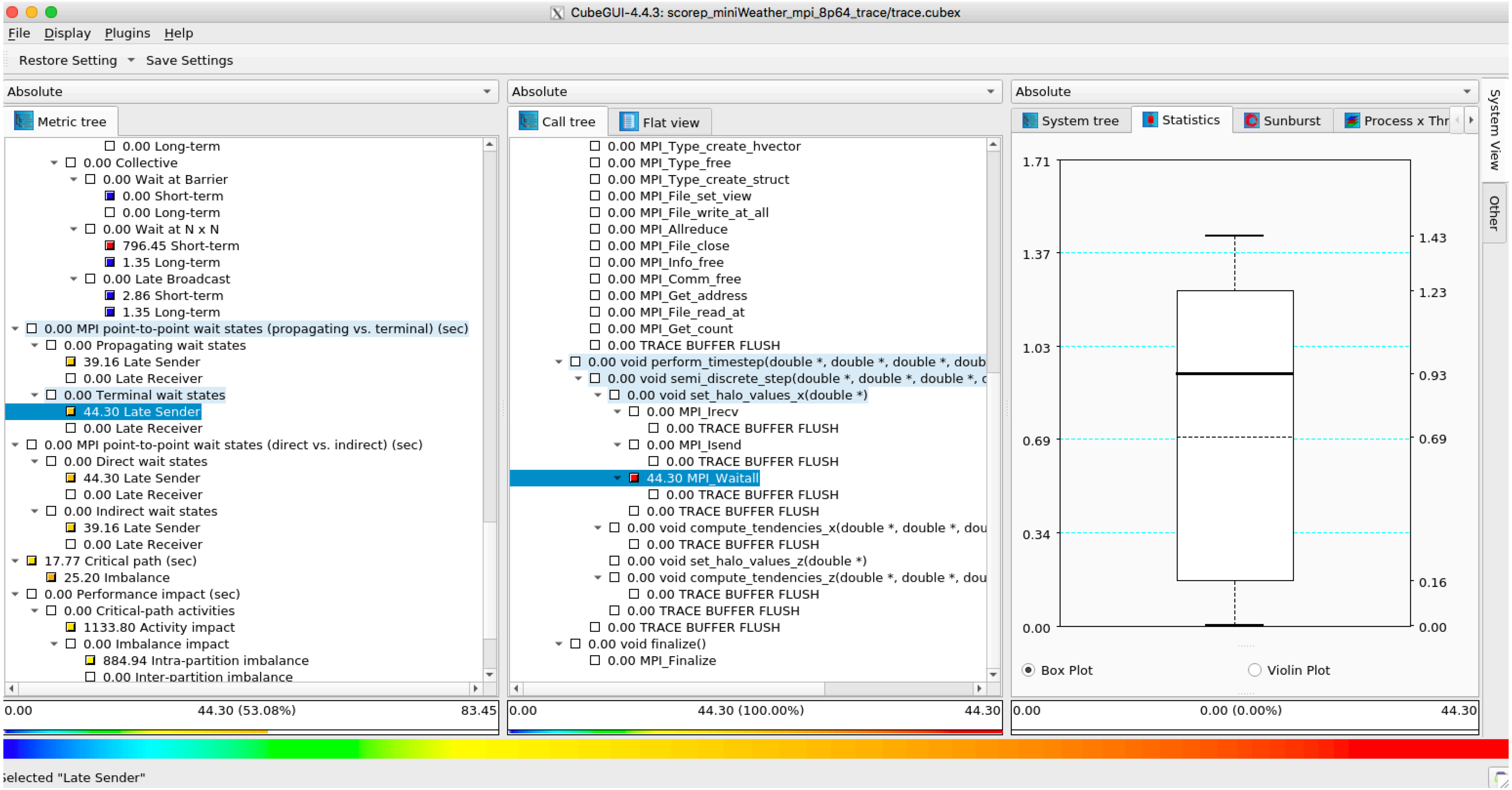
Description:
Waiting time in MPI point-to-point operations that propagates further and causes additional waiting time on other processes.
Unit:
Seconds

Terminal MPI Point-to-point Wait States

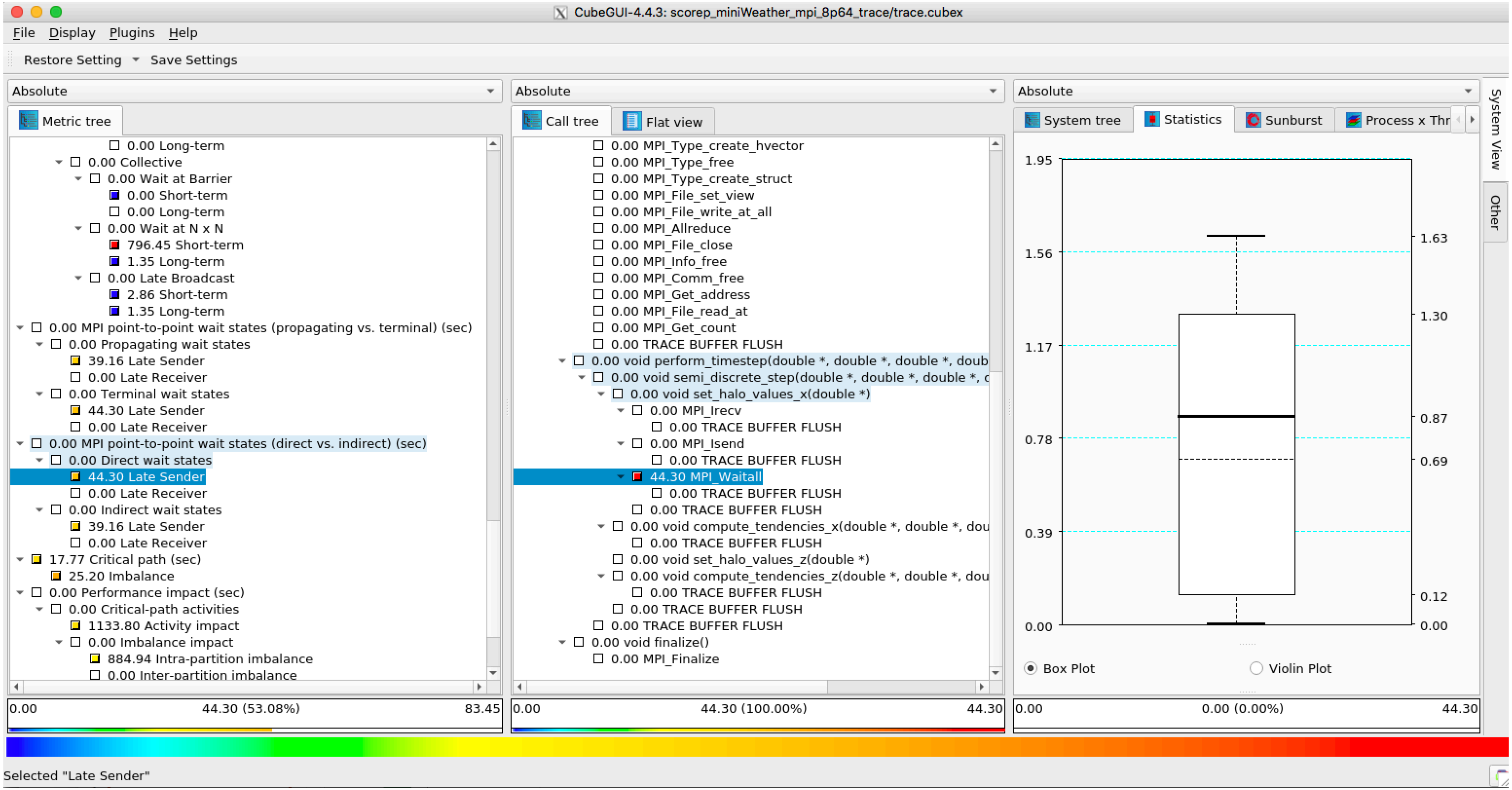
(only available after remapping)

Description:
Waiting time in MPI point-to-point operations that does not propagate further.

Terminal wait states



Direct wait stats



Wait States

MPI Point-to-point Wait State Classification: Direct vs. Indirect

(only available after remapping)

Description:
Partitions MPI point-to-point wait states into waiting time directly caused by delays and waiting time caused by propagation.
Unit:
Seconds

Direct MPI Point-to-point Wait States

(only available after remapping)

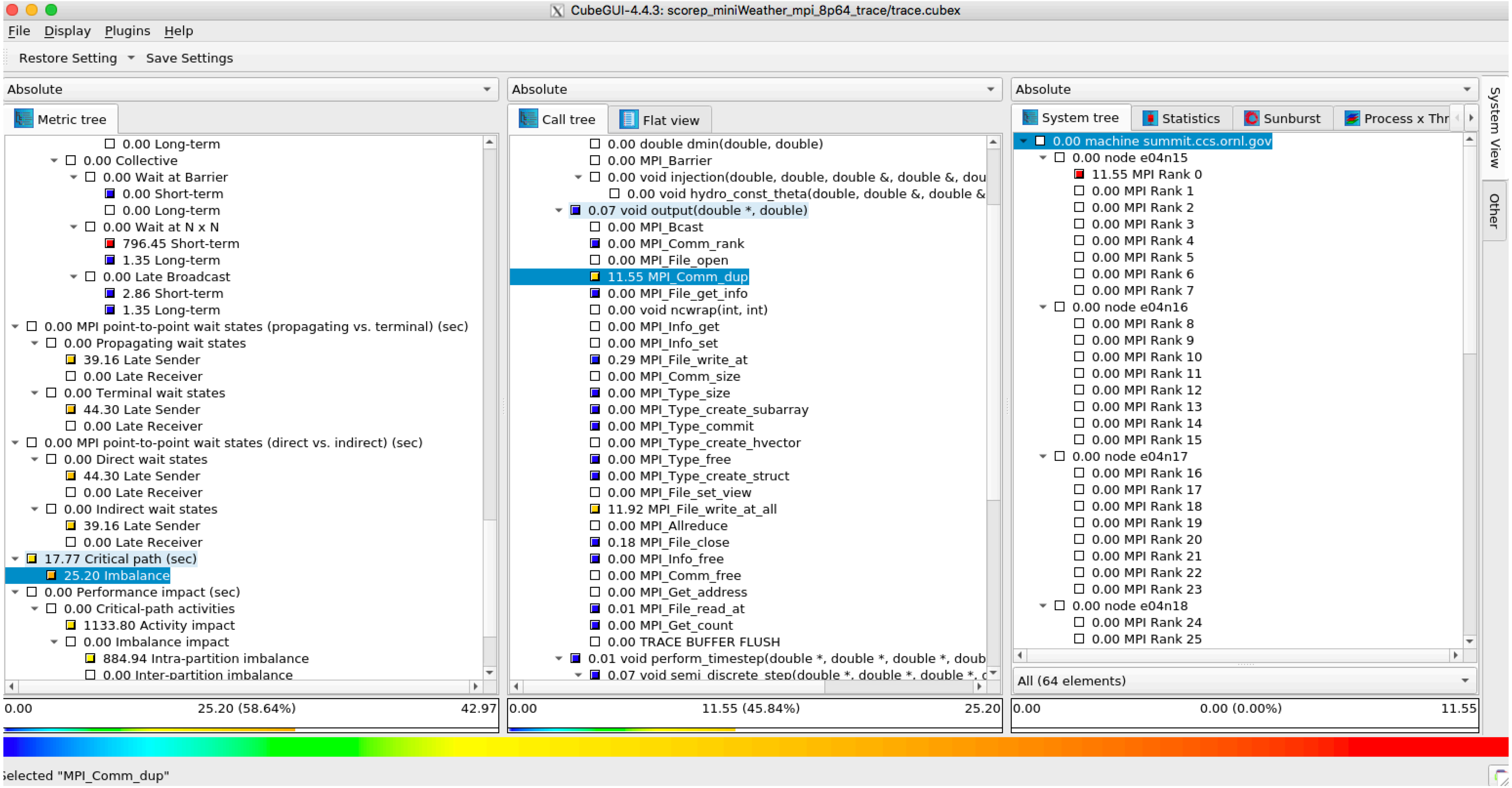
Description:
Waiting time in MPI point-to-point operations that results from direct delay, i.e., is directly caused by a load- or communication imbalance.
Unit:
Seconds

Indirect MPI Point-to-point Wait States

(only available after remapping)

Description:
Waiting time in MPI point-to-point operations that results from indirect delay, i.e., is caused indirectly by wait-state propagation.
Unit:
Seconds

Imbalance in the critical path

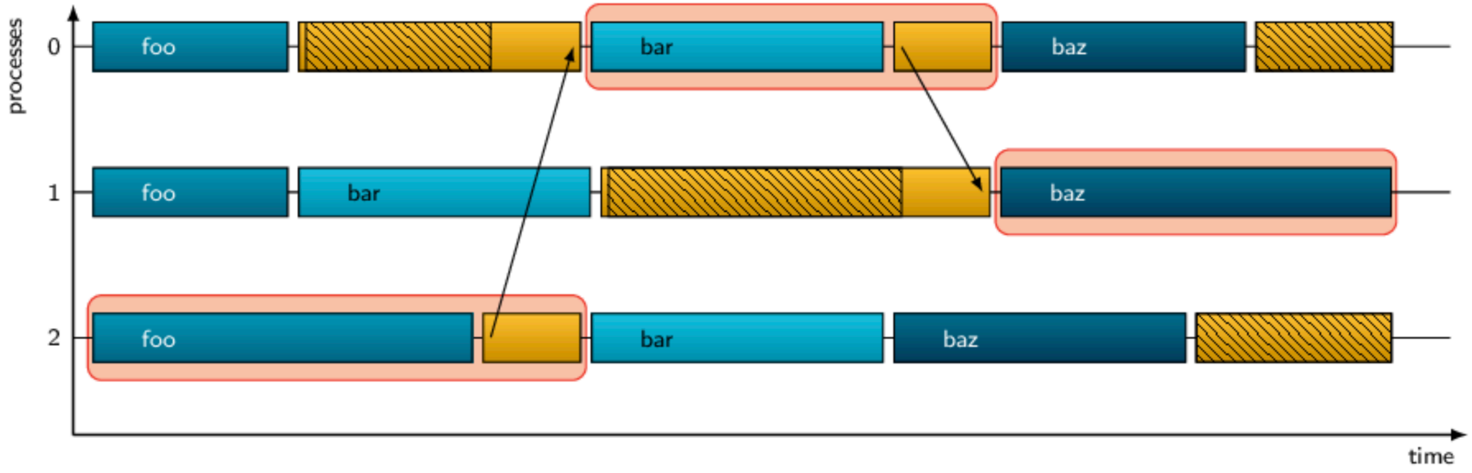


Critical Path Profile

Critical Path Profile

Description:

This metric provides a profile of the application's critical path. Following the causality chain from the last active program process/thread back to the program start, the critical path shows the call paths and processes/threads that are responsible for the program's wall-clock runtime.



Note that Scalasca does not yet consider POSIX threads when determining the critical path. Thus, the critical-path profile is currently incorrect if POSIX threads are being used, as only the master thread of each process is taken into account. However, it may still provide useful insights across processes for hybrid MPI+Pthreads applications.

Unit:

Seconds

Diagnosis:

Call paths that occupy a lot of time on the critical path are good optimization candidates. In contrast, optimizing call paths that do not appear on the critical path will not improve program runtime.

Call paths that spend a disproportionately large amount of time on the critical path with respect to their total execution time indicate parallel bottlenecks, such as load imbalance or serial execution. Use the percentage view modes and compare execution time and critical path profiles to identify such call paths.

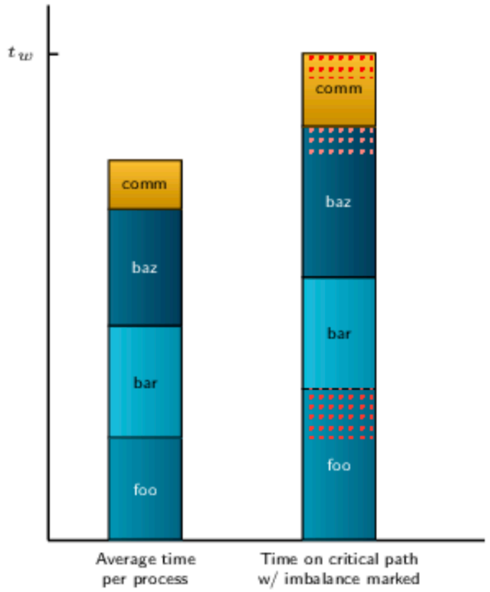
The system tree pane shows the contribution of individual processes/threads to the critical path. However, note that the critical path runs only on one process at a time. In a well-balanced program, the critical path follows a more-or-less random course across processes and may not visit many processes at all. Therefore, a high critical-path time on individual processes does not necessarily indicate a performance problem. Exceptions are significant load imbalances or serial execution on single processes. Use the critical-path imbalance metric or compare with the distribution of execution time across processes to identify such cases.

Critical Path Imbalance

Description:

This metric highlights parallel performance bottlenecks.

In essence, the critical-path imbalance is the positive difference of the time a call path occupies on the critical path and the call path's average runtime across all CPU locations. Thus, a high critical-path imbalance identifies call paths which spend a disproportionate amount of time on the critical path.



The image above illustrates the critical-path profile and the critical-path imbalance for the example in the [Critical Path Profile] metric description. Note that the excess time of regions `foo` and `baz` on the critical path compared to their average execution time is marked as imbalance. While also on the critical path, region `bar` is perfectly balanced between the processes and therefore has no contribution to critical-path imbalance.

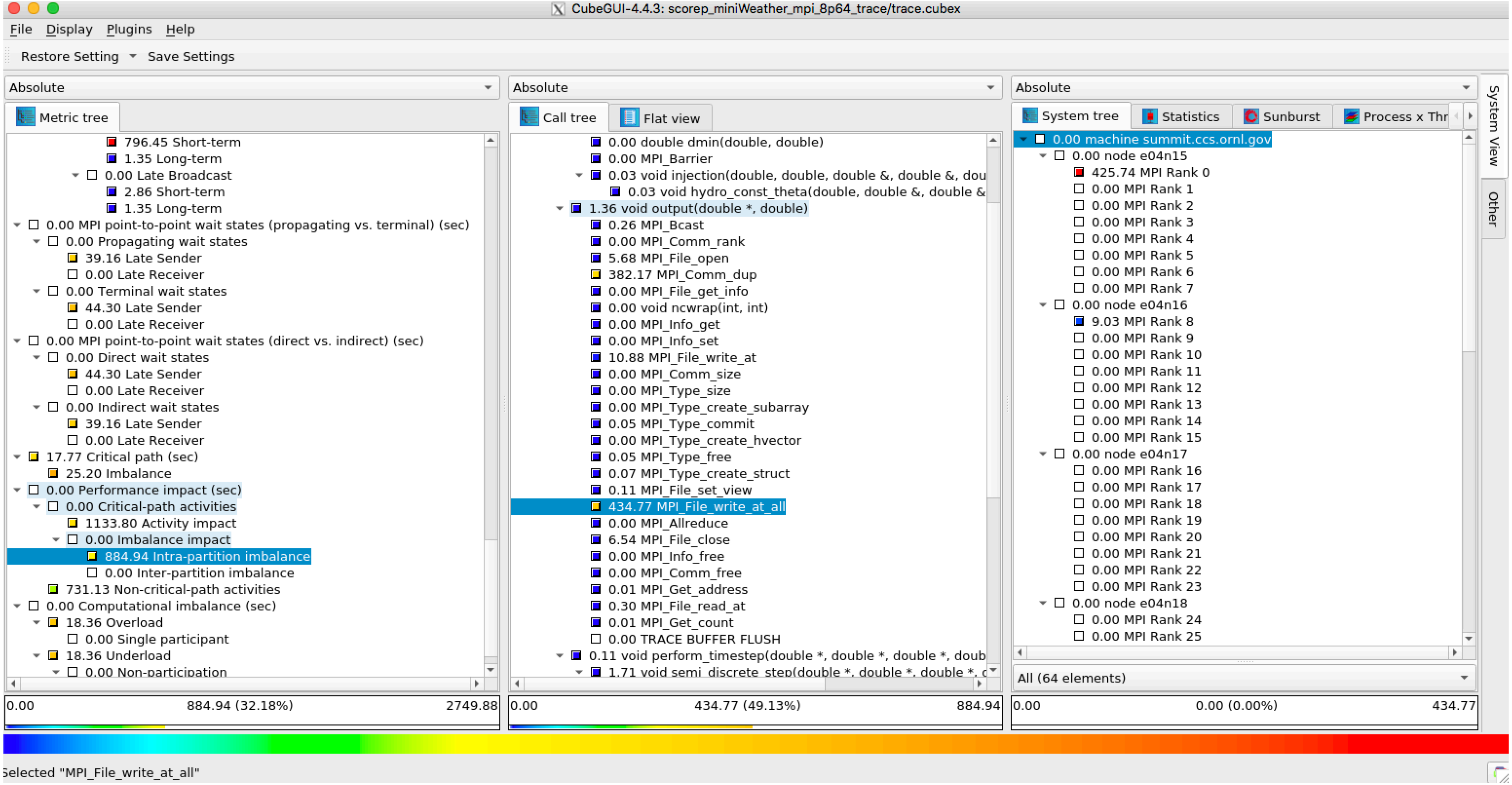
Unit:

Seconds

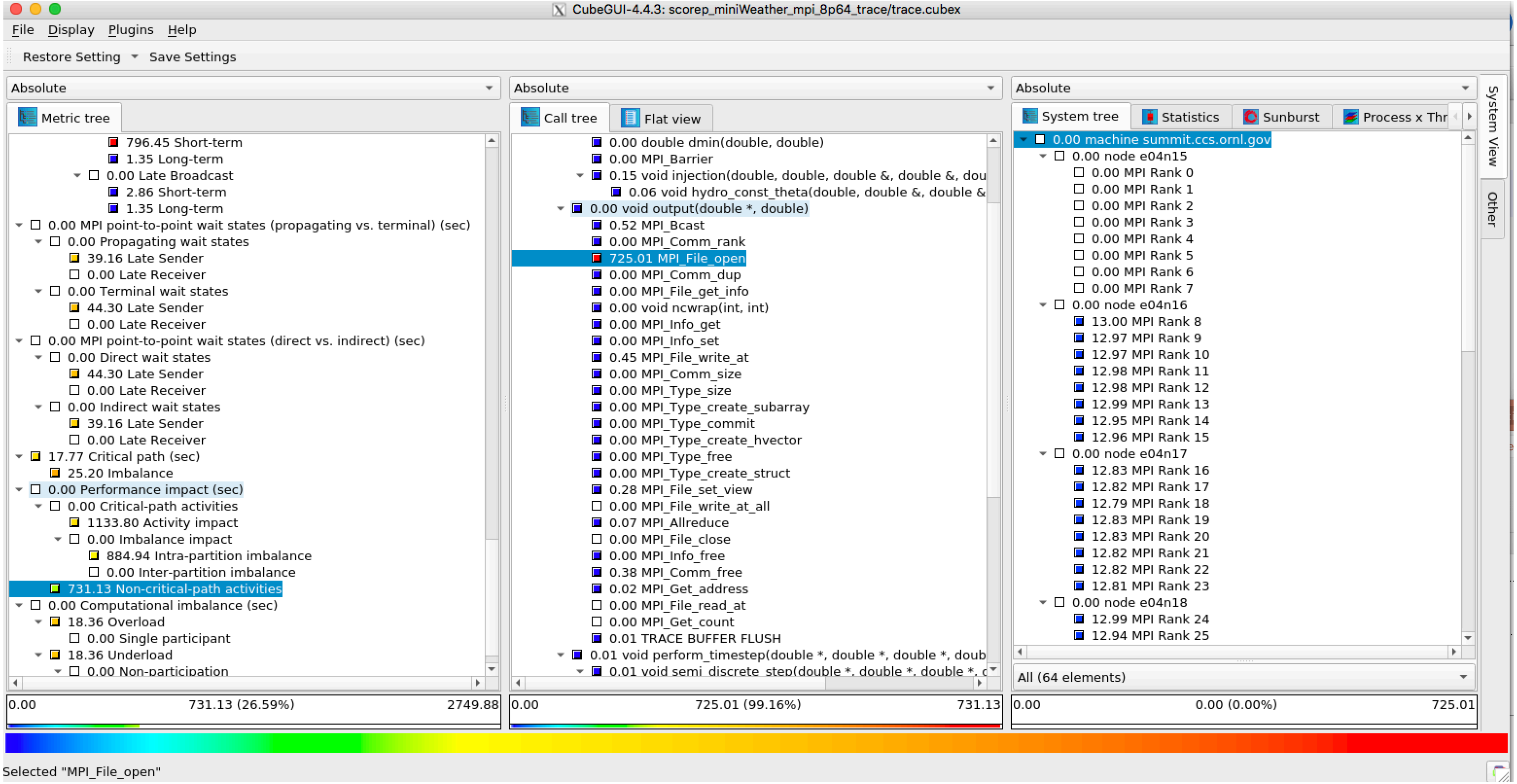
Diagnosis:

A high critical-path imbalance indicates a parallel bottleneck, such as load imbalance or serial execution. Cross-analyze with other metrics, such as the distribution of execution time across CPU locations, to identify the type and causes of the parallel bottleneck.

Intra-partition Imbalance



Non Critical Path Activities



Scalasca with MPI+OpenMP

MiniWeather – MPI+OMP

- Compile:

  MPI: make PREP="scorep --mpp=mpi –openmp --pdt" openmp

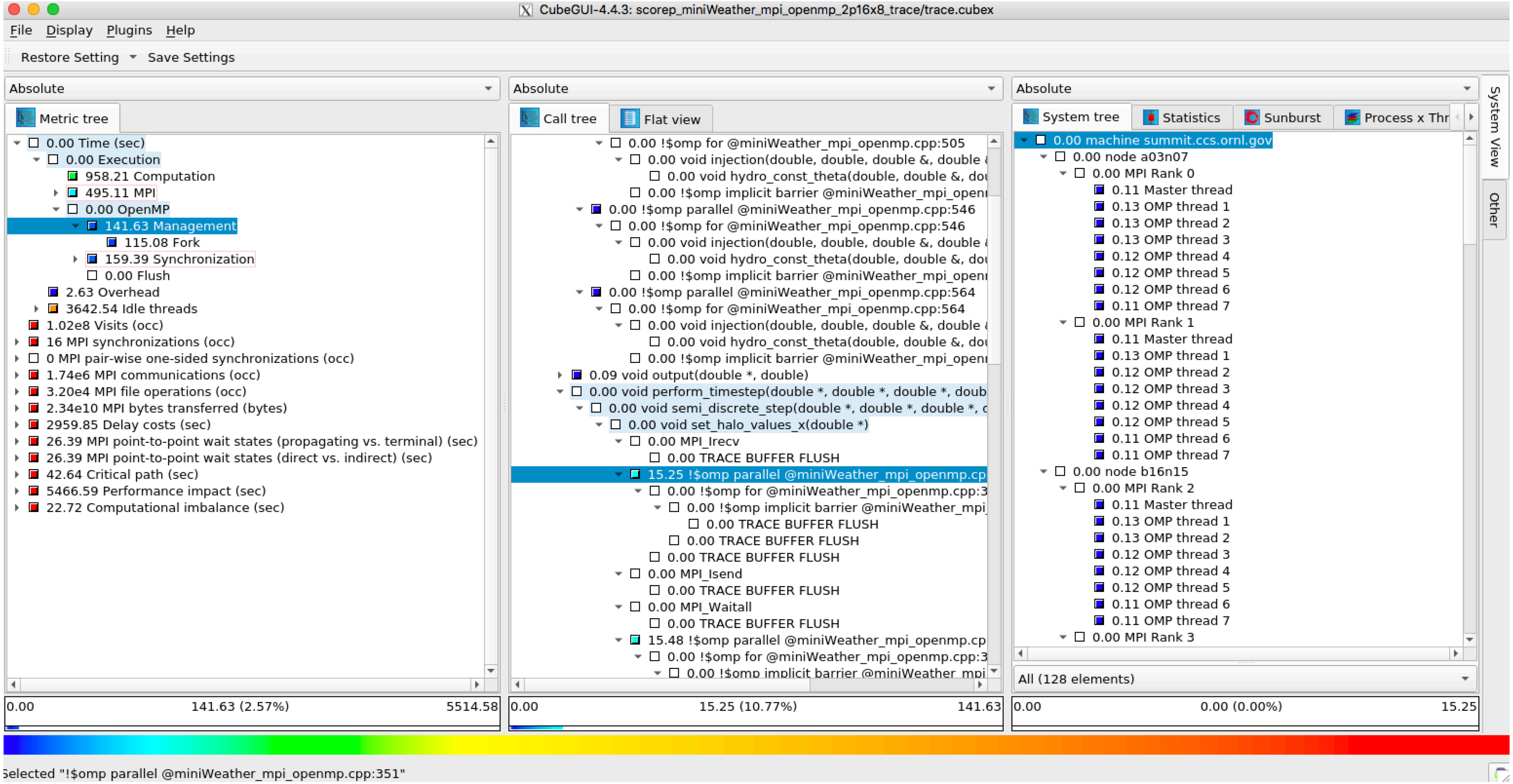
- Execution (submission script):

  scalasca -analyze -q -t jsrun -n 16 -r 2 -a 1 -c 8 "-b packed:8" ./miniWeather_mpi_openmp

- Visualize:

  scalasca -examine /gpfs/…/ scorep_miniWeather_mpi_openmp_2p16x8_trace

OpenMP Views

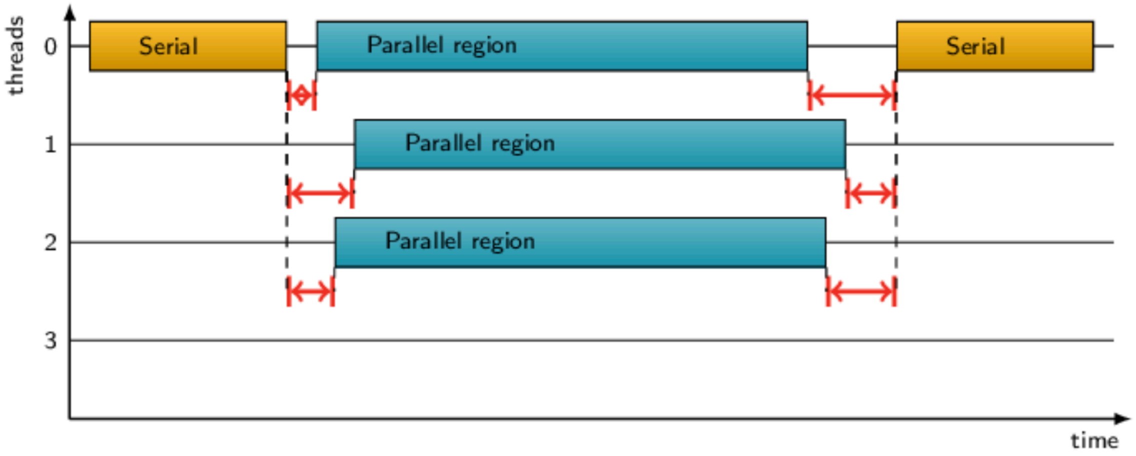


OpenMP Thread Management Time

OpenMP Thread Management Time

Description:

Time spent managing teams of threads, creating and initializing them when forking a new parallel region and clearing up afterwards when joining.



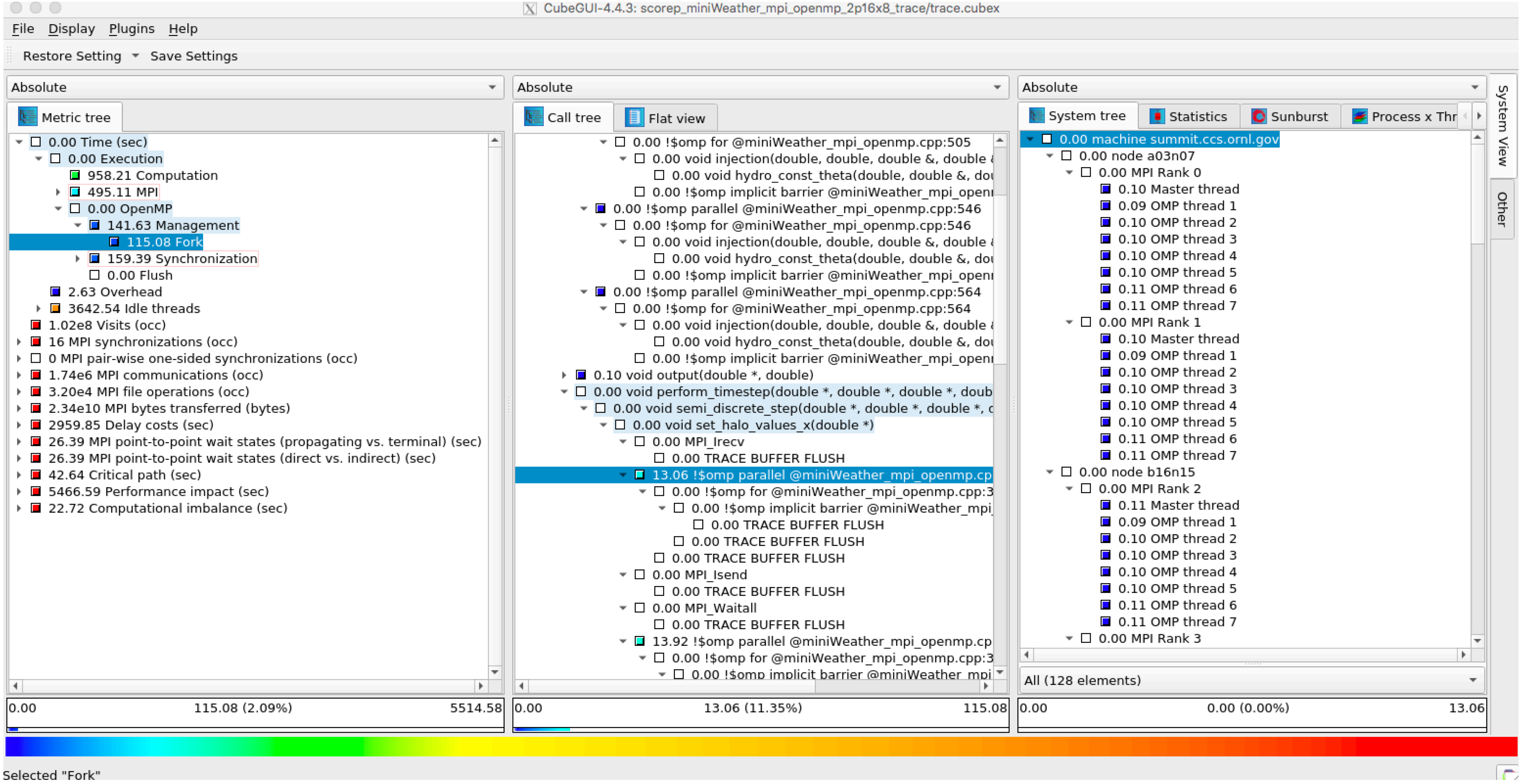
Unit:

Seconds

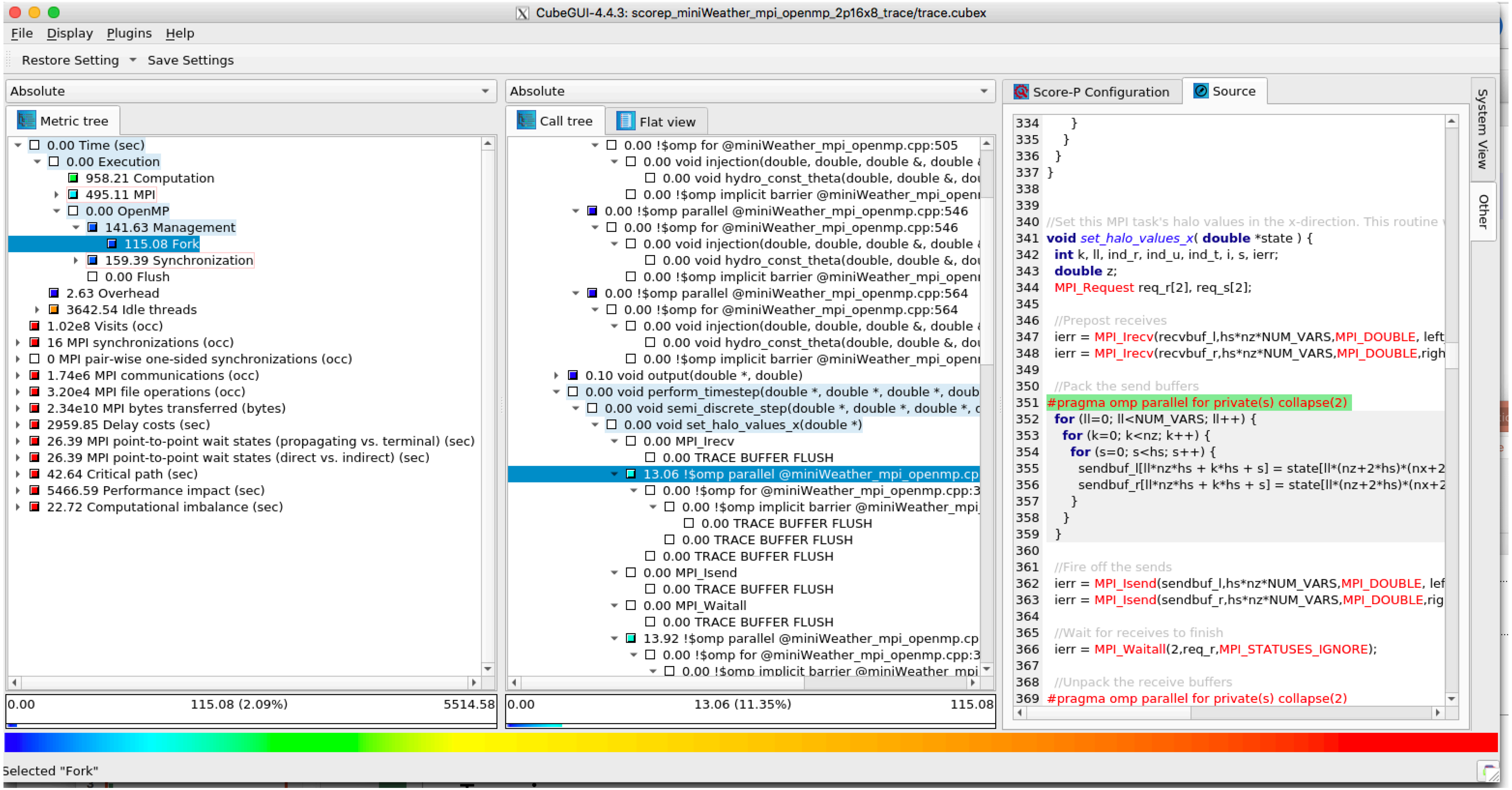
Diagnosis:

Management overhead for an OpenMP parallel region depends on the number of threads to be employed and the number of variables to be initialized and saved for each thread, each time the parallel region is executed. Typically a pool of threads is used by the OpenMP runtime system to avoid forking and joining threads in each parallel region, however, threads from the pool still need to be added to the team and assigned tasks to perform according to the specified schedule. When the overhead is a significant proportion of the time for executing the parallel region, it is worth investigating whether several parallel regions can be combined to amortize thread management overheads. Alternatively, it may be appropriate to reduce the number of threads either for the entire execution or only for this parallel region (e.g., via `num_threads` or `if` clauses).

Duration of Fork



Source code of corresponding OpenMP call

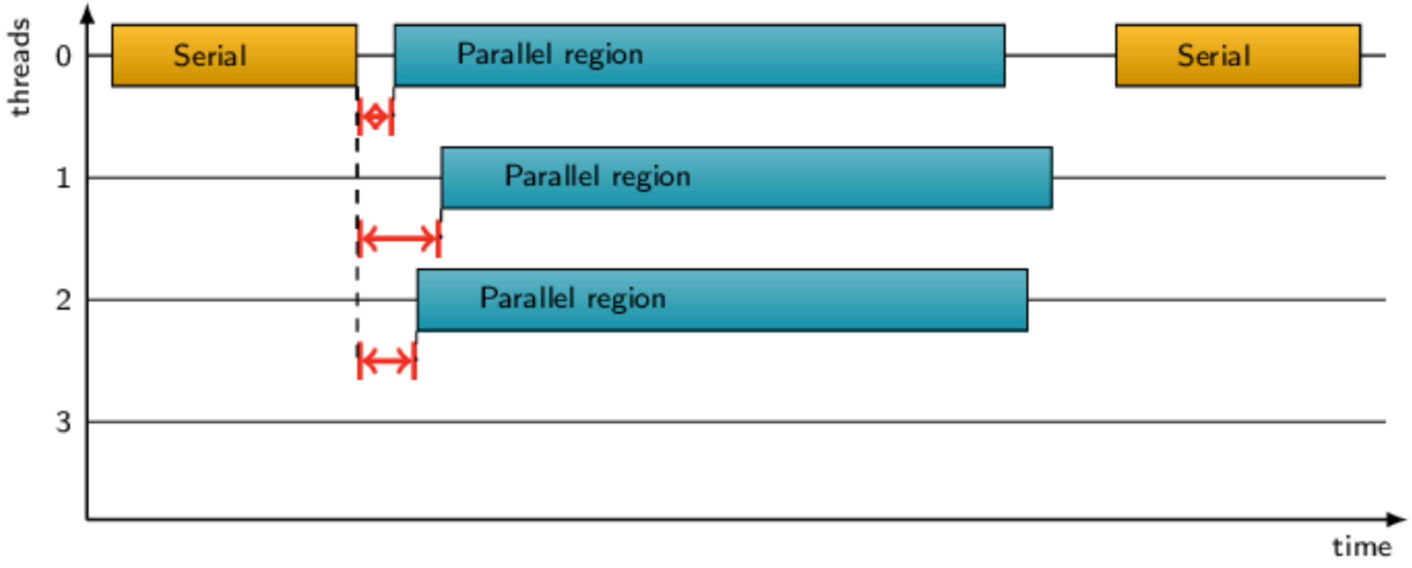


OpenMP Thread Team Fork

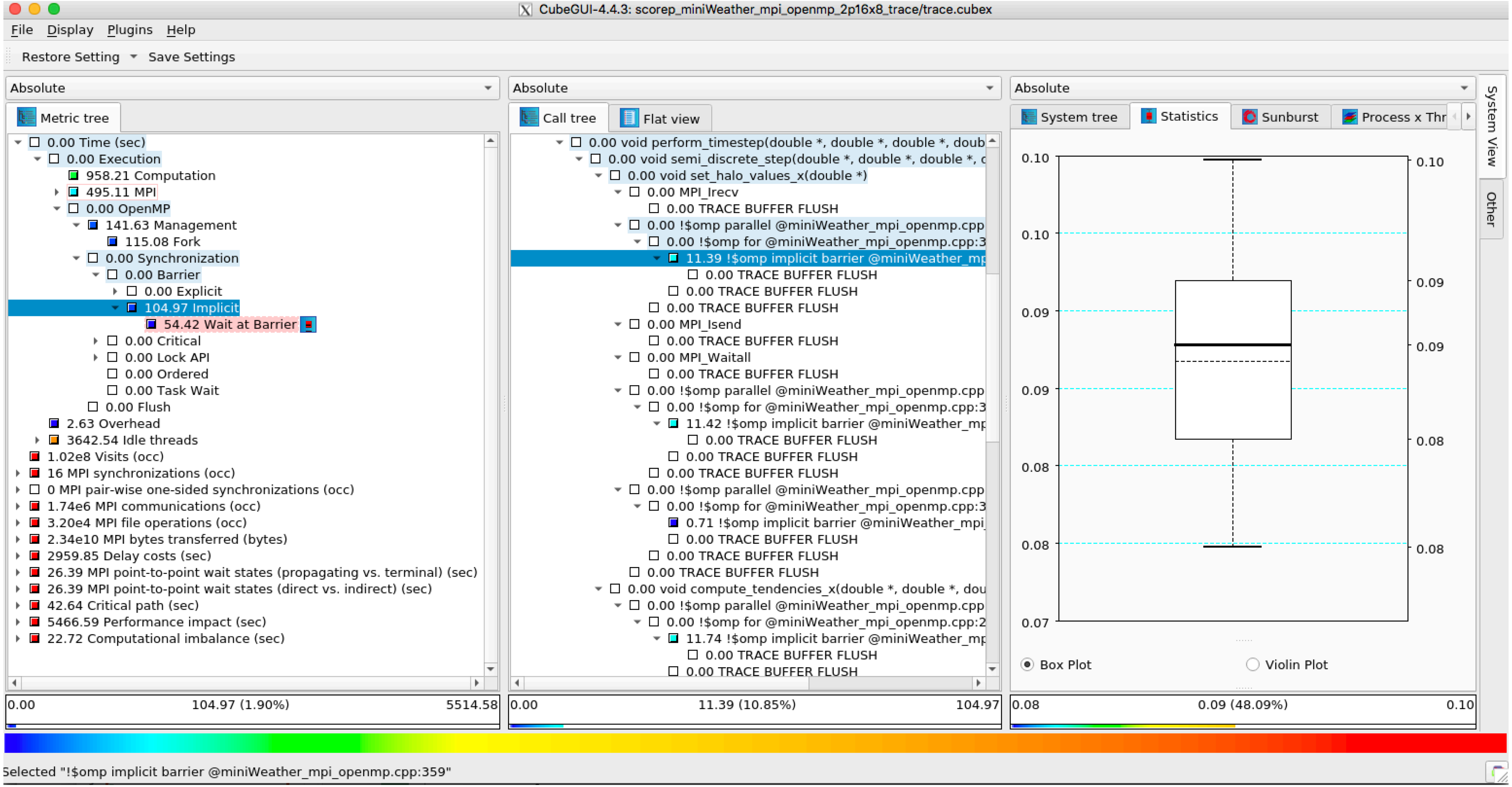
OpenMP Thread Team Fork Time

Description:

Time spent creating and initializing teams of threads.



Implicit Synchronization



Oak Ridge National Laboratory

Implicit Synchronization - Explanation

OpenMP Implicit Barrier Synchronization Time

(only available after remapping)

Description:

Time spent in implicit (i.e., compiler-generated) OpenMP barrier synchronization, both waiting for other threads Wait at Implicit OpenMP Barrier Time and inherent barrier processing overhead.

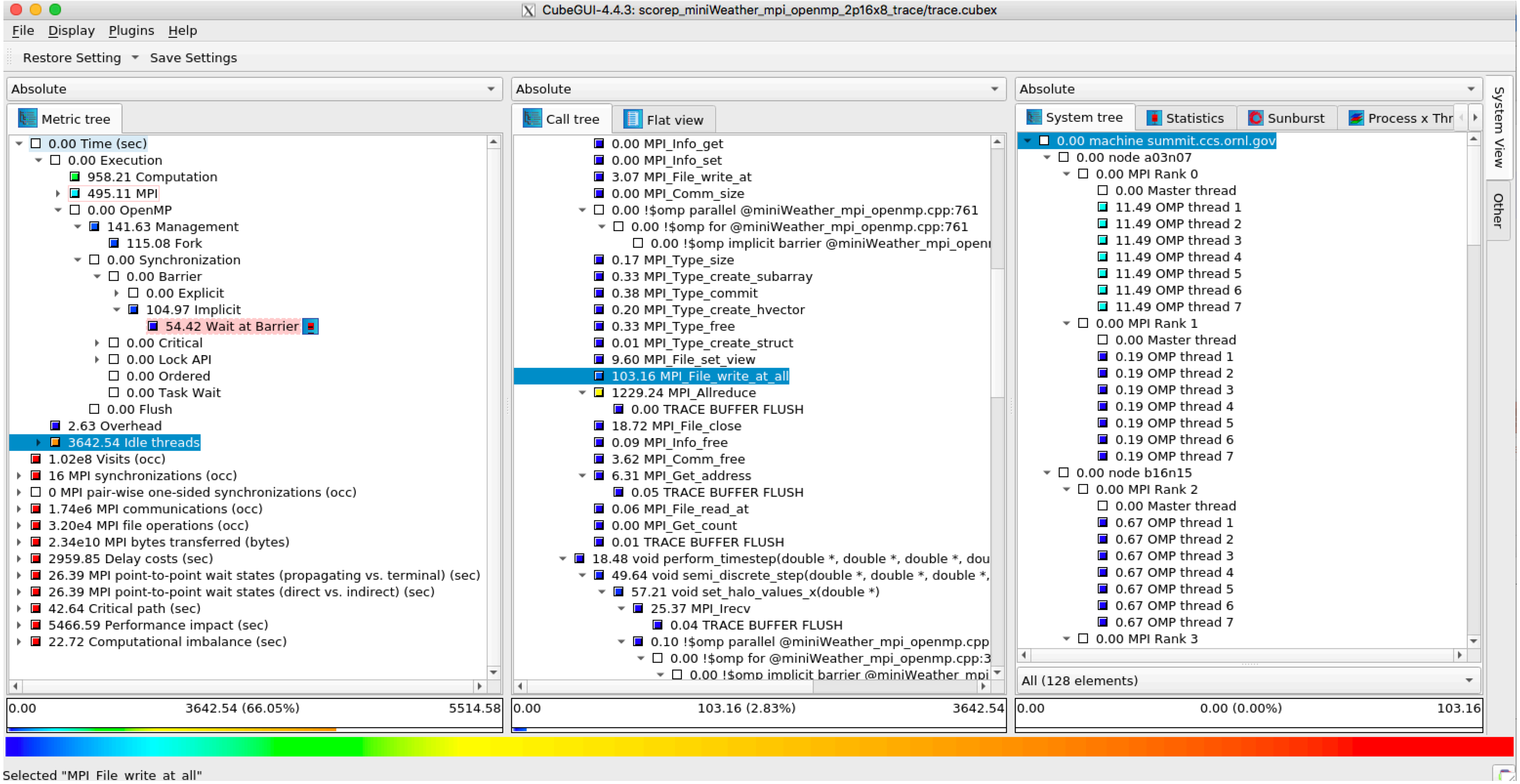
Unit:

Seconds

Diagnosis:

Examine the time that each thread spends waiting at each implicit barrier, and if there is a significant imbalance then investigate whether a `schedule` clause is appropriate. Note that `dynamic` and `guided` schedules may require more OpenMP Thread Management Time than `static` schedules. Consider whether it is possible to employ the `nowait` clause to reduce the number of implicit barrier synchronizations.

Idle threads



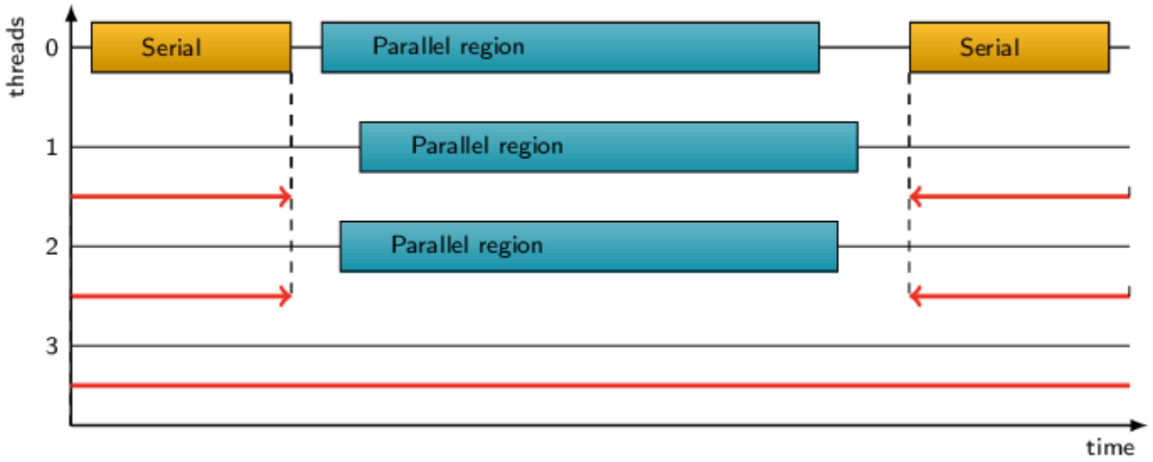
Idle threads - Explanation

OpenMP Idle Threads Time

(only available after remapping)

Description:

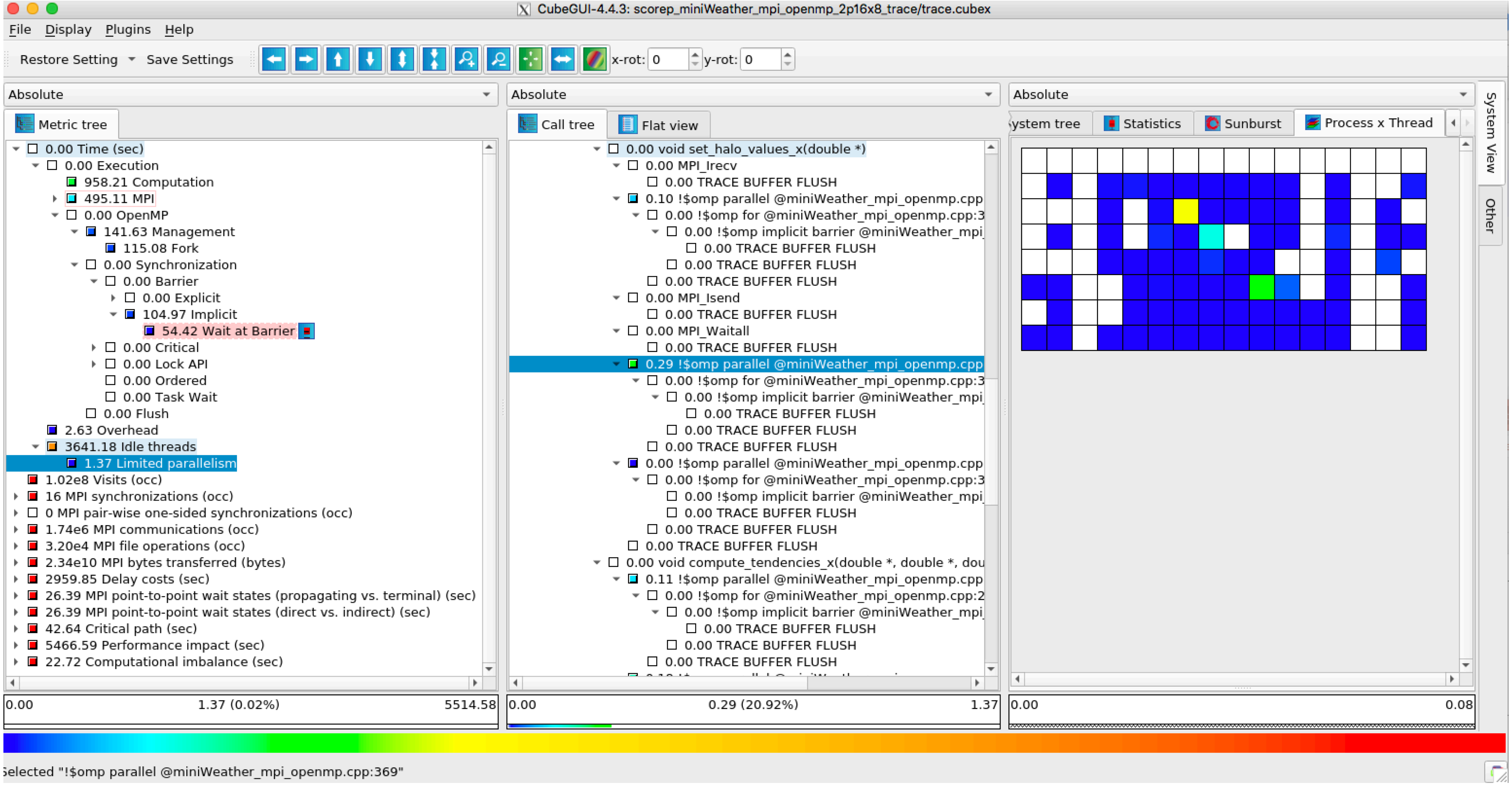
Idle time on CPUs that may be reserved for teams of threads when the process is executing sequentially before and after OpenMP parallel regions, or with less than the full team within OpenMP parallel regions.



Oak Ridge National Laboratory

Open slide master to edit

Limited Parallelism – Process x Thread



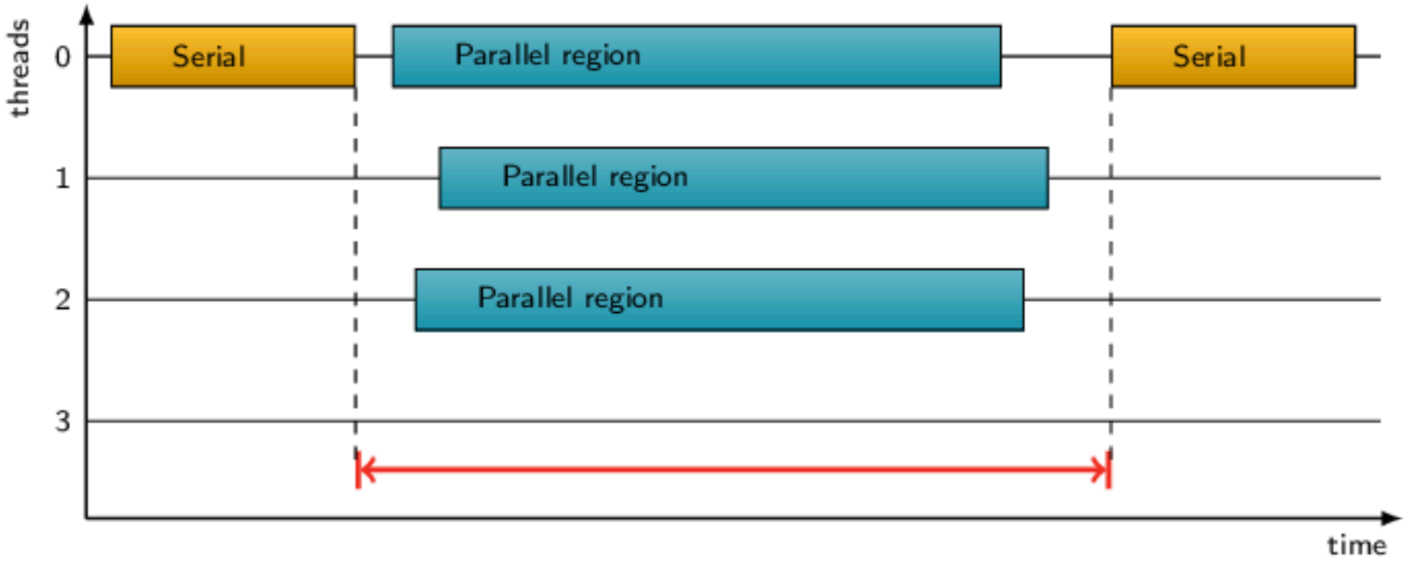
Limited Parallelism - Explanation

OpenMP Limited Parallelism Time

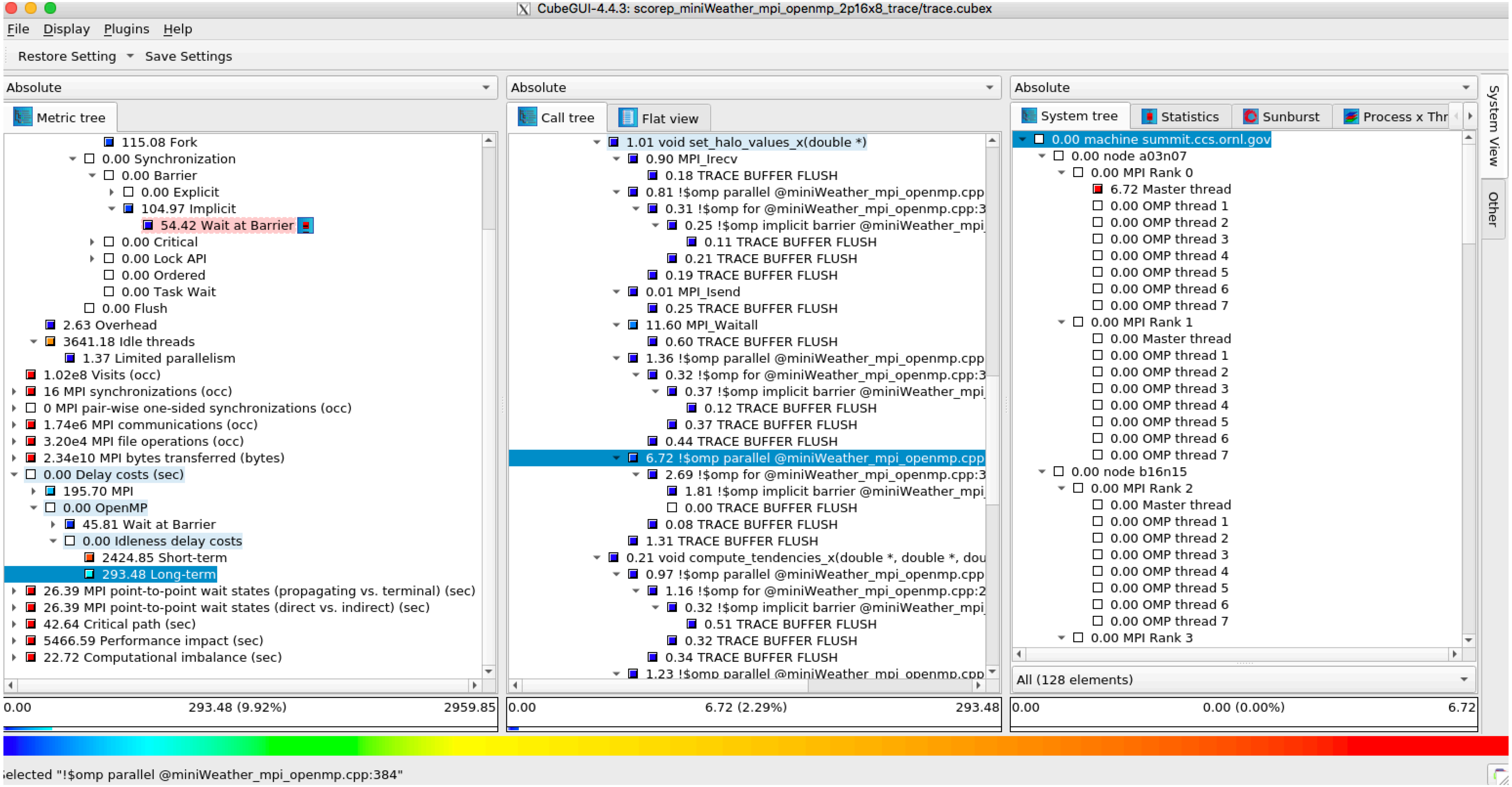
(only available after remapping)

Description:

Idle time on CPUs that may be reserved for threads within OpenMP parallel regions where not all of the thread team participates.



Long-term delay costs



Long/Short-term OpenMP Thread Idleness delay costs

Long-term OpenMP Thread Idleness Delay Costs

Description:

Long-term delay costs reflect indirect effects of load or communication imbalance on wait states. That is, they cover waiting time that was caused indirectly by wait states which themselves delay subsequent communication operations. Here, they identify costs and locations of delays that indirectly leave OpenMP worker threads idle due to wait-state propagation. In particular, long-term idle thread delay costs indicate call paths and processes/threads that increase the time worker threads are idling because of MPI wait states outside of OpenMP parallel regions.

Unit:

Seconds

Diagnosis:

High long-term delay costs indicate that computation or communication overload in/on the affected call paths and processes/threads has far-reaching effects. That is, the wait states caused by the original computational overload spread along the communication chain to remote locations.

Short-term OpenMP Thread Idleness Delay Costs

Description:

Short-term costs reflect the direct effect of sections outside of OpenMP parallel regions on thread idleness.

Unit:

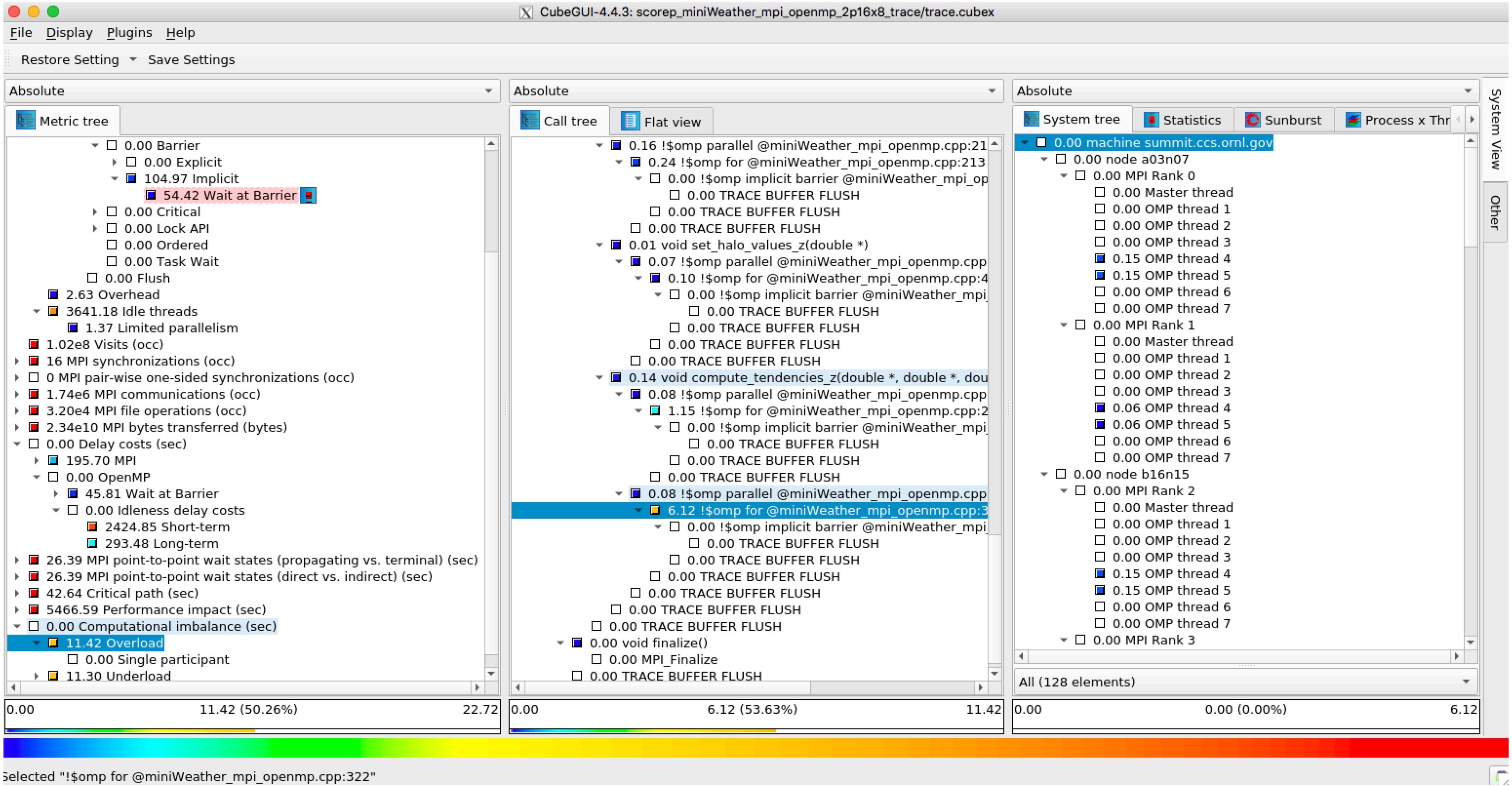
Seconds

Diagnosis:

High short-term delay costs for thread idleness indicates that much time is spent outside of OpenMP parallel regions in the affected call paths.

Try to reduce workload in the affected call paths. Alternatively, apply OpenMP parallelism to more sections of the code.

Computational Imbalance overload



OAK RIDGE National Laboratory

Computational Imbalance overload – Process x Thread