

Maximilian Knespel¹, **Bert Wesarg**¹⁺²

¹TU Dresden / CIDS / ZIH

²GWT-TUD GmbH

Extended Visualization and Analysis of Non-Blocking Multi-Threaded Communication of a Large-Scale CFD Solver with Vampir

17th International Parallel Tools Workshop 2026, Dresden

Outline

Motivation and Research Questions

Extended Messages Event Model

Function Summary

Experiments

Conclusion and Outlook

Motivation

Non-blocking communication, as implemented through MPI_Isend and MPI_Irecv, is a well-established mode of operation in a wide range of high-performance computing (HPC) applications

Naturally, the wait-state analysis has been extended from the blocking pattern of inefficiencies for these operations

Non-blocking communication can also be completed wait-free by testing

There is no tool available to analyze the optimal test frequency

Research Questions

Can Vampir assess whether an application performs too many or too few MPI_Test calls?

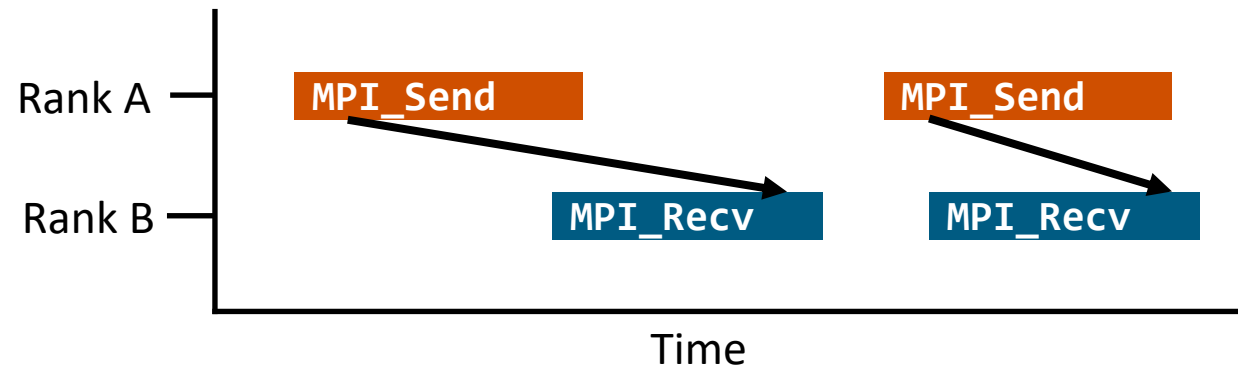
→ Requires an extended message event model capturing per-test timestamps and thread context

Can Vampir calculate and present well-known wait-time patterns attributed to root-cause functions, such as late sender/receiver?

Messages in Vampir

Originates from the blocking MPI event model

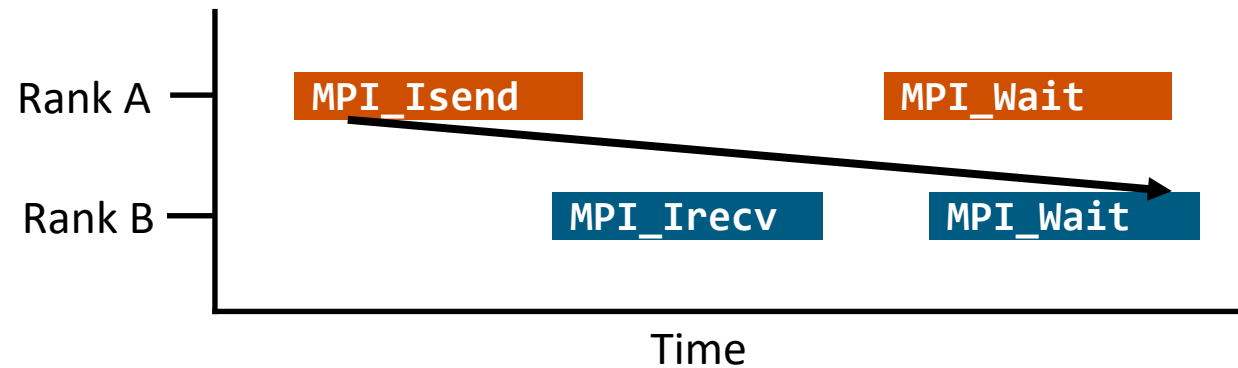
- Origin thread + timestamp
- Destination thread + timestamp



Messages in Vampir

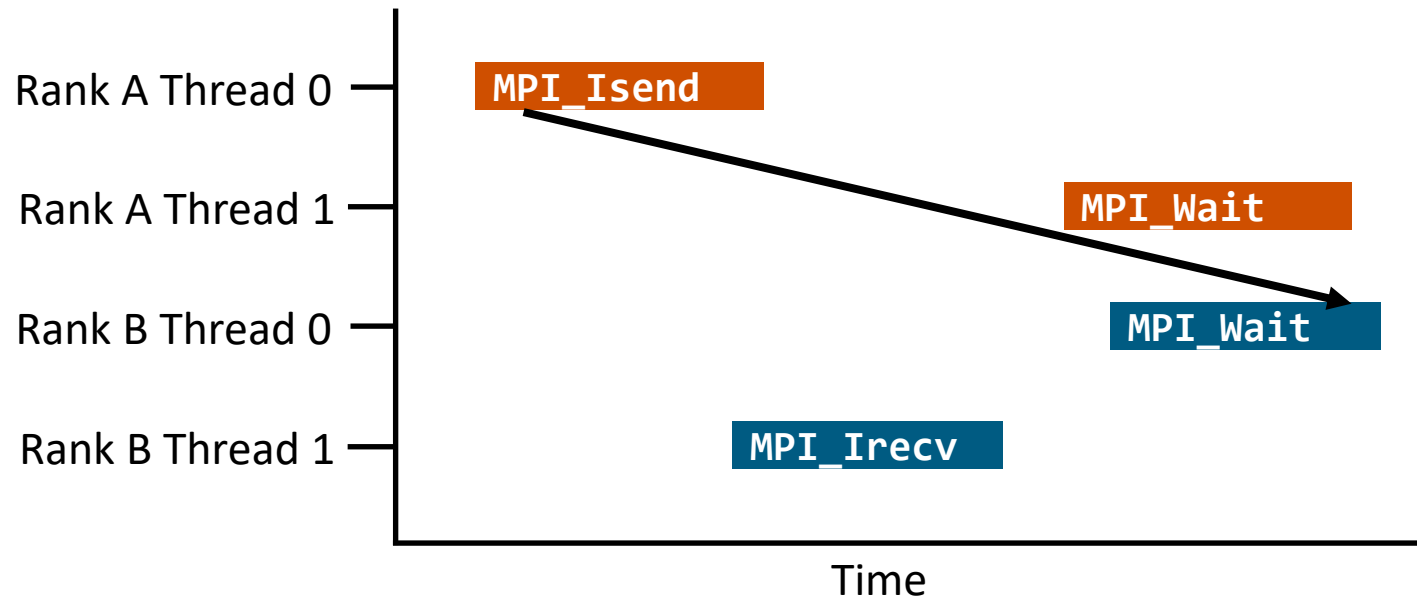
Non-blocking decouples message initiation from completion, on either side

- **Trigger** & completion (thread, timestamp) tuple at origin
- Trigger & **completion** (thread, timestamp) tuple at destination



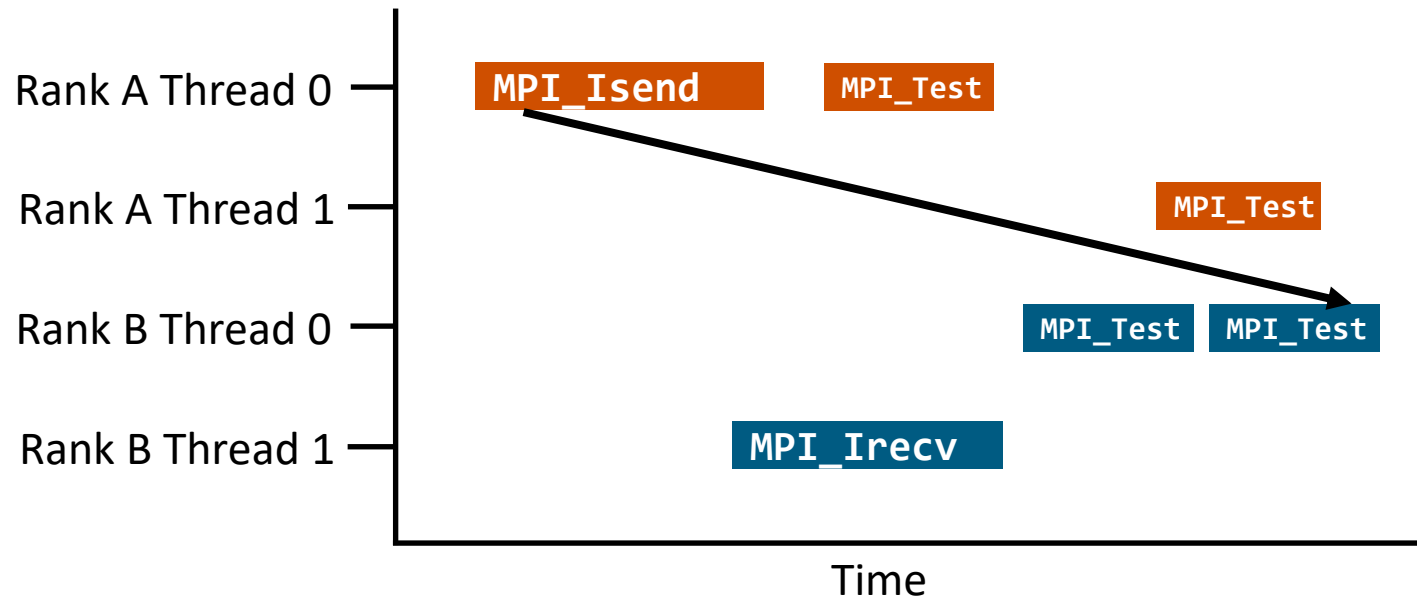
Messages in Vampir

Exacerbated by using multiple threads



Messages in Vampir

Test-based completion

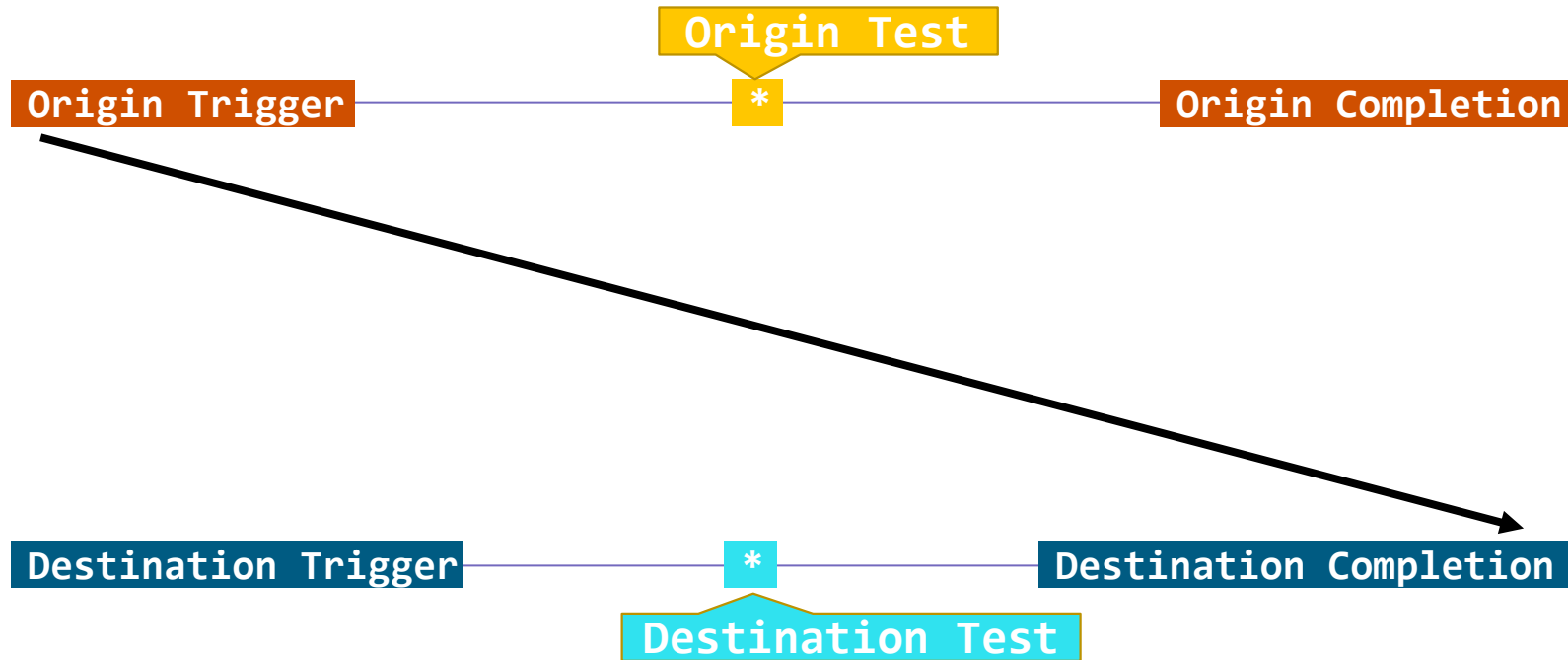


Messages in Vampir

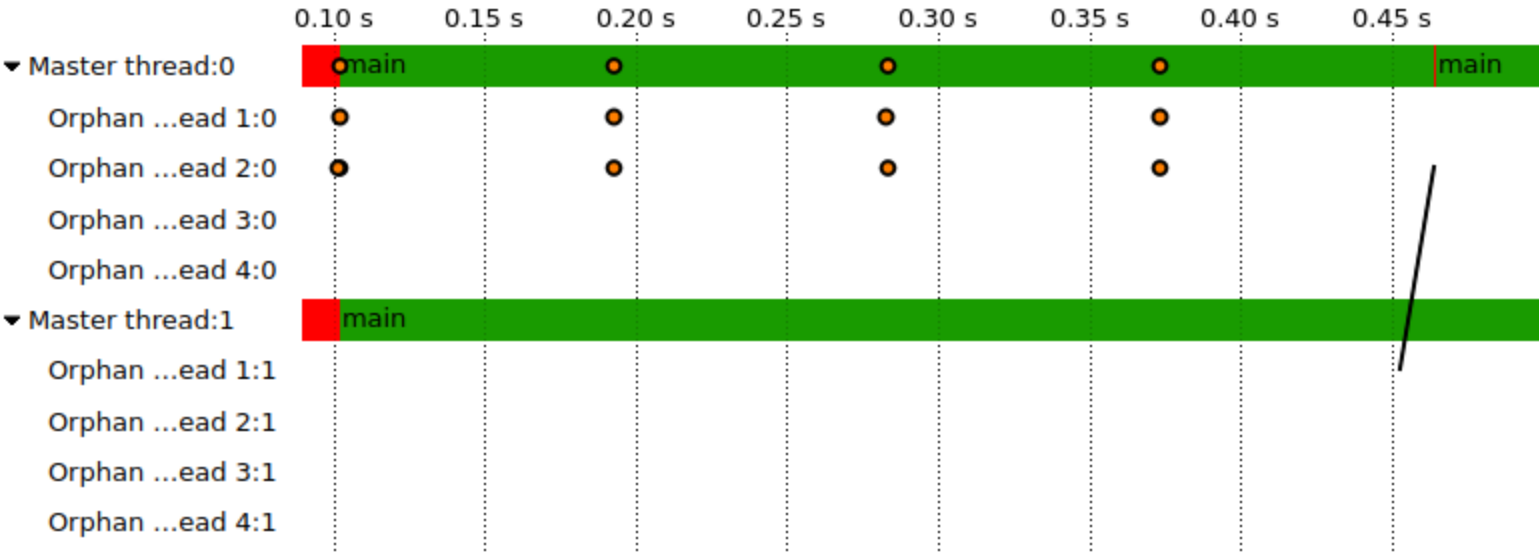
Complete message with durations for trigger and completions (aka. function calls)

- **Trigger** & completion (thread, start, end) tuple at origin
- Arbitrary many (thread, timestamp) tuples for unsuccessful tests
- Trigger & **completion** (thread, timestamp) tuple at destination
- Arbitrary many (thread, start, end) tuples for unsuccessful tests

Messages in Vampir



Messages in Vampir



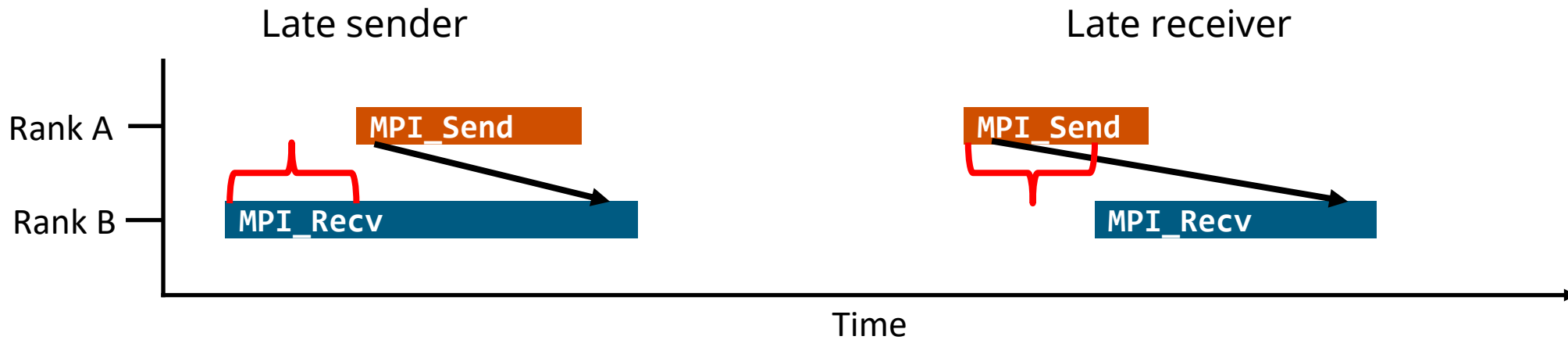
Extended event model only visible for the selected message

Message details lists all trigger & completion function calls

Master Timeline	
Property	Value
Display	Master Timeline
Type	Message
Message Type	Point to point
Origin	Orphan thread 1:1
Destination	Orphan thread 2:0
Communicator	MPI_COMM_WORLD
Tag	0
Start Time	0.451758 s
Arrival Time	0.462688 s
Duration	0.0109297 s
Time Range	Set Zoom
Size	4 B
Data Rate	365.976768 B/s
Receive Request Tests	12
Type	Function
Location	Master thread:0
Function	MPI_Irecv
Function Group	MPI Point-to-point Communication
Interval Begin	0.101557 s
Interval End	0.101573 s
Duration	1.5379e-05 s
Time Range	Set Zoom
Call Level	2
Type	Function
Location	Master thread:1
Function	MPI_Isend
Function Group	MPI Point-to-point Communication
Interval Begin	0.451738 s
Interval End	0.451813 s
Duration	7.4311e-05 s
Time Range	Set Zoom
Call Level	2
Type	Function
Location	Orphan thread 1:1
Function	MPI_Test
Function Group	MPI Request Handling
Interval Begin	0.452786 s
Interval End	0.452814 s
Duration	2.8404e-05 s
Time Range	Set Zoom
Call Level	1
Type	Function
Location	Orphan thread 2:0
Function	MPI_Test
Function Group	MPI Request Handling
Interval Begin	0.462622 s
Interval End	0.462702 s
Duration	7.9701e-05 s
Time Range	Set Zoom
Call Level	1

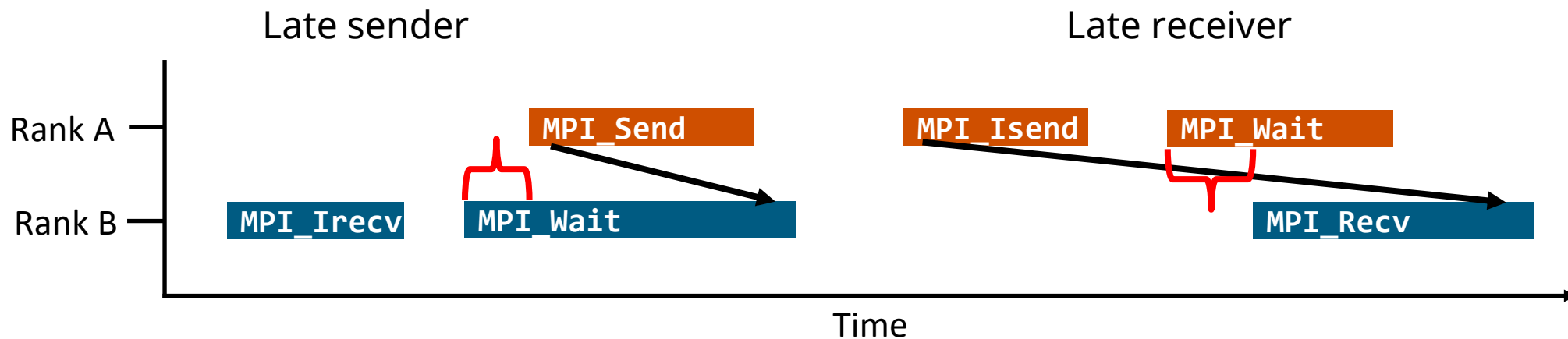
Late Sender/Receiver

Well-known pattern of inefficiency

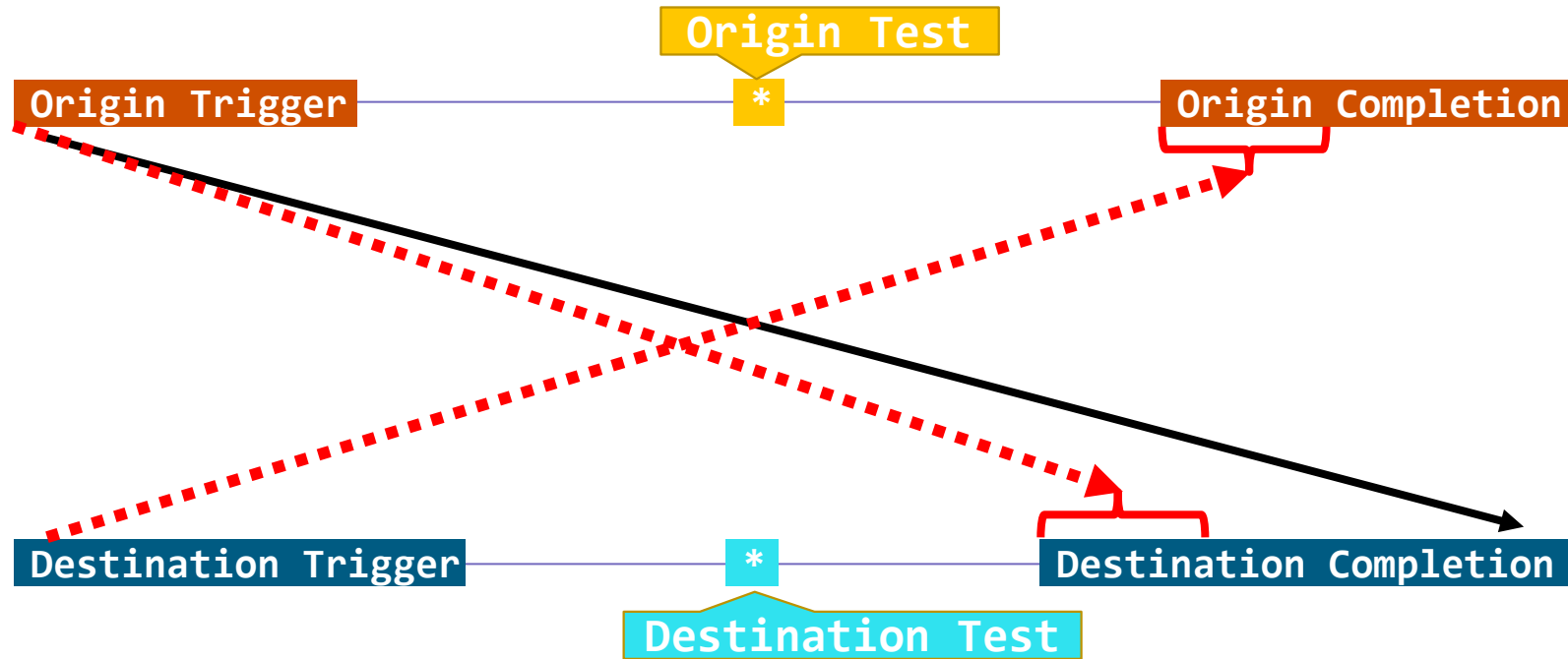


Late Sender/Receiver

Wait-time induced in blocking completion calls



Late Sender/Receiver in Vampir



Late Sender/Receiver in Vampir

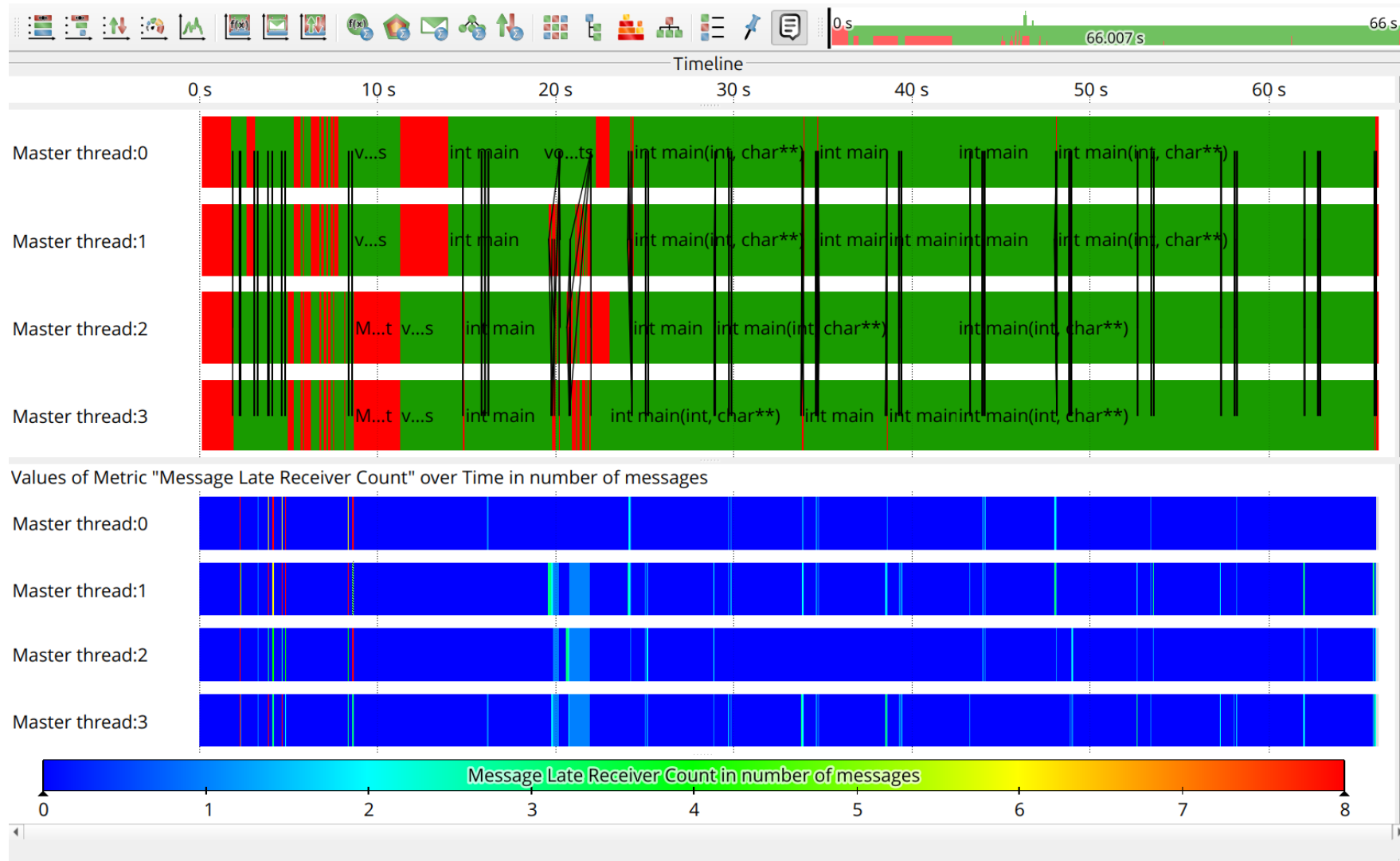
All events from one location are stored in one worker MPI rank

Late sender wait-time should be attribute to the function call in the receiver

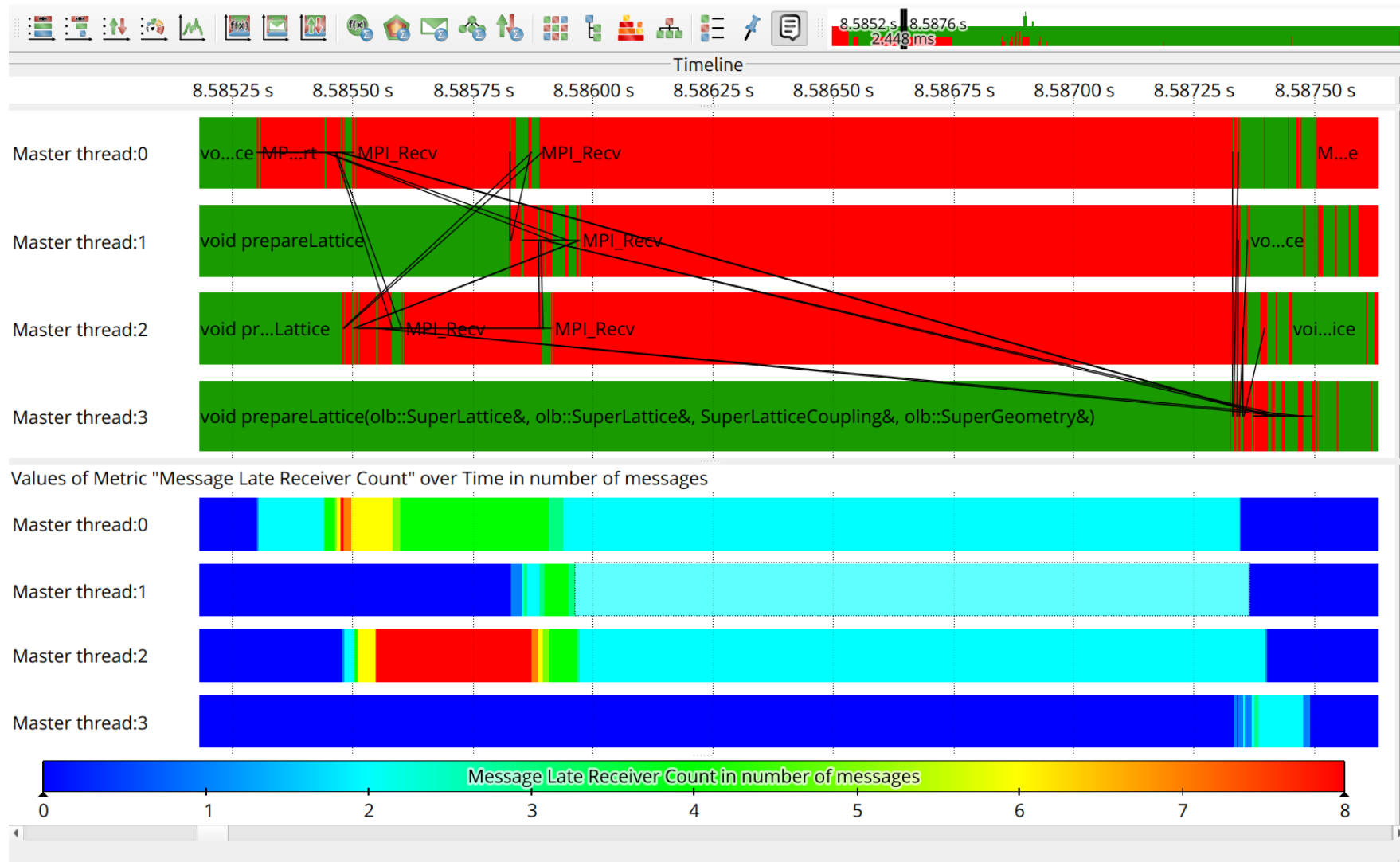
Message is stored in the sender's worker, completion call is stored in the destination's worker

⇒ Duplicate messages in the origin's and destination's worker

Late Sender/Receiver in Vampir



Late Sender/Receiver in Vampir



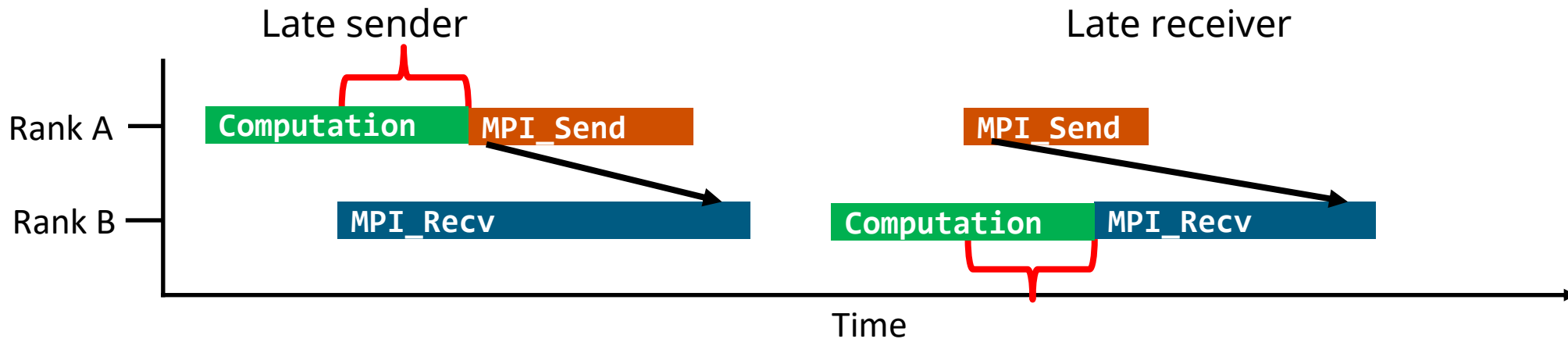
Late Sender/Receiver in Vampir: “Root Cause”

Wait-time metrics form the basis for the critical-path analysis

Critical-path analysis forms the basis for the root-cause analysis

Not possible with Vampir

Attributing the induced wait-time to the thread activity prior the trigger calls



Function Summary in Vampir

Which function caused the most wait-time?

Flat profile of exclusive/inclusive execution time per function

Time is just a monotonic increasing metric which is attributed to the profile if the function is “active” (depends on exclusive /inclusive)

Any monotonic metric should be usable as the “time” metric in the Function Summary

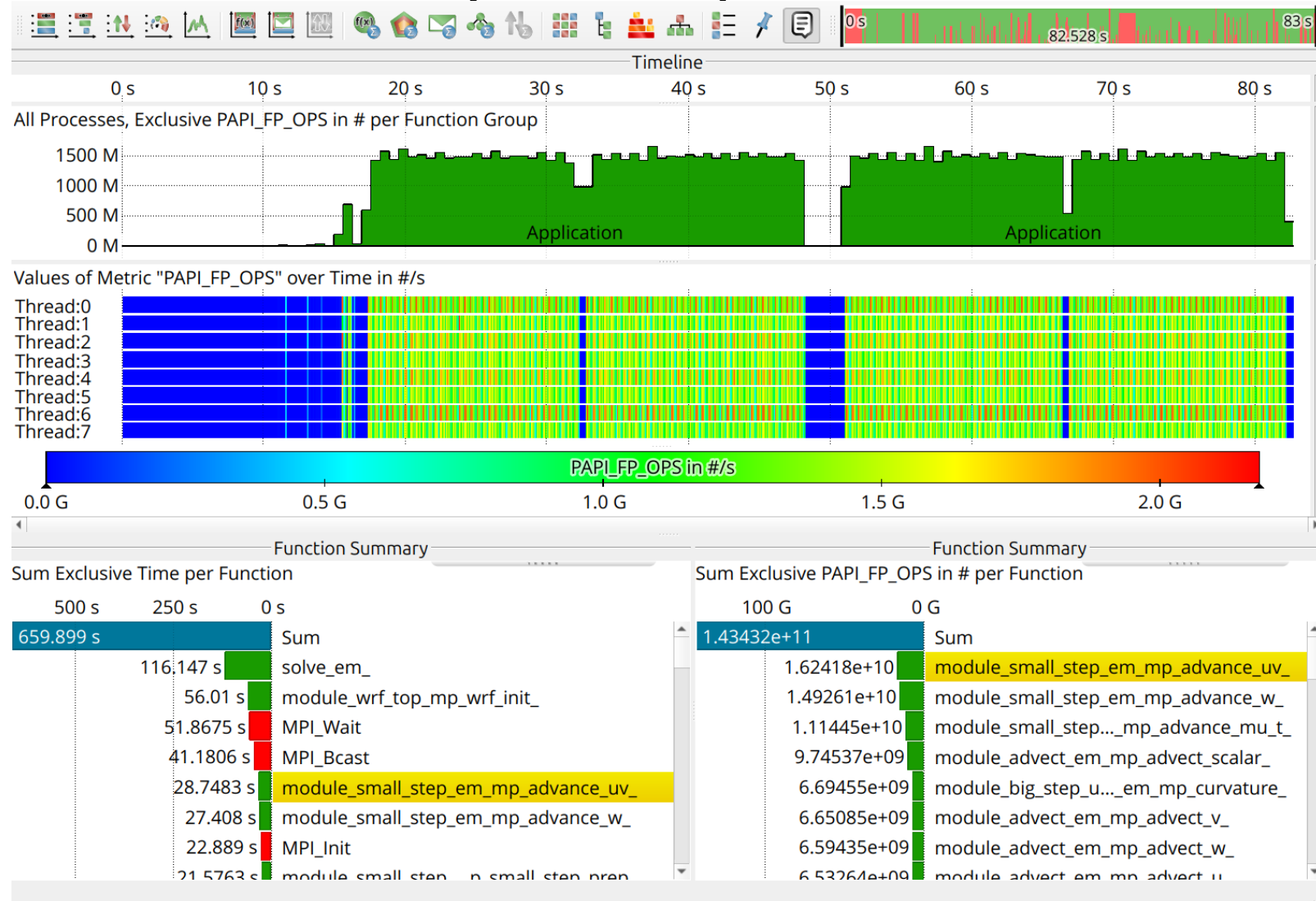
None-Time Metric Function Summary in Vampir

Example: Number of floating operations

⇒ monotonic increasing,
assumed to be linear

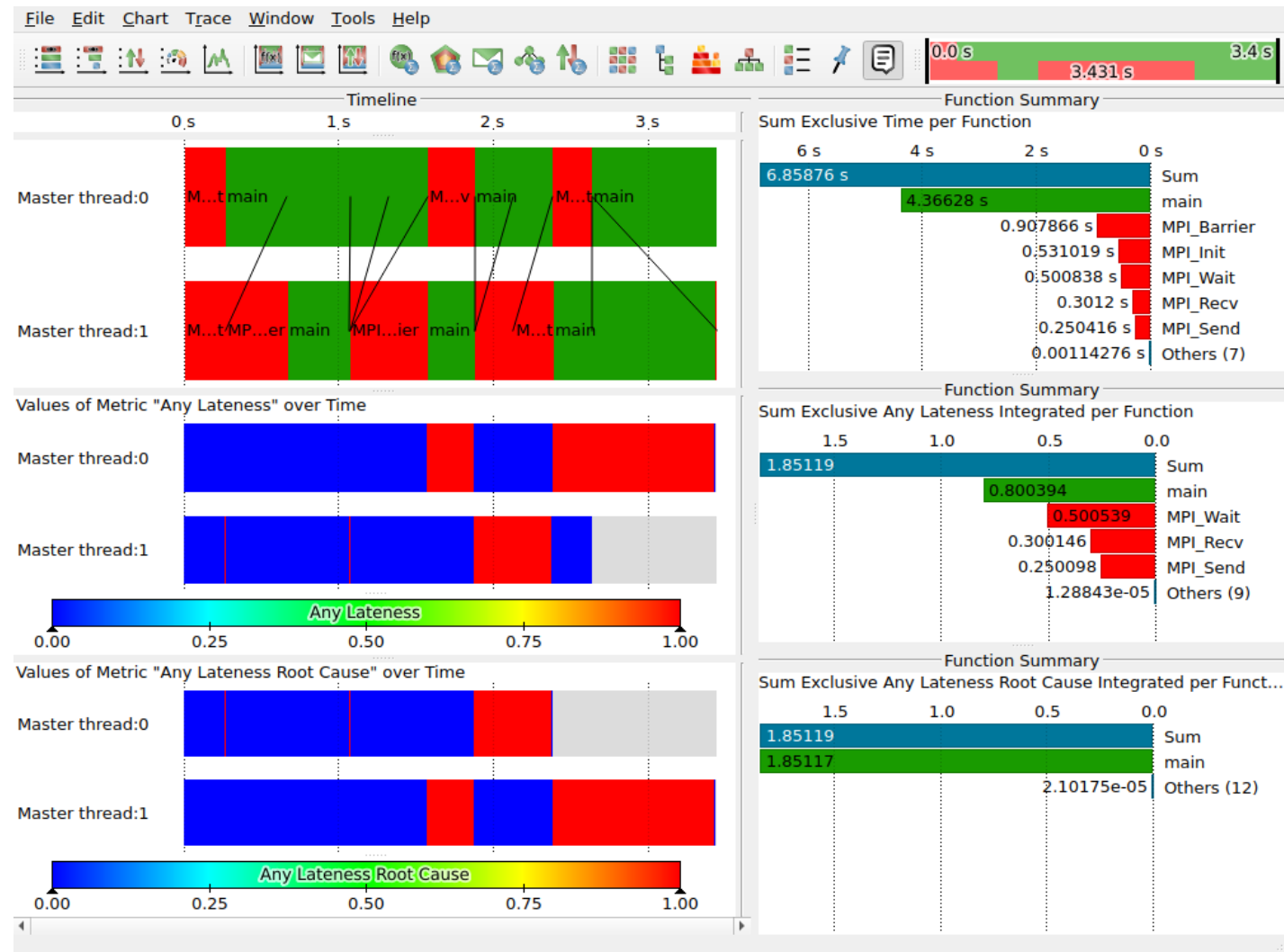
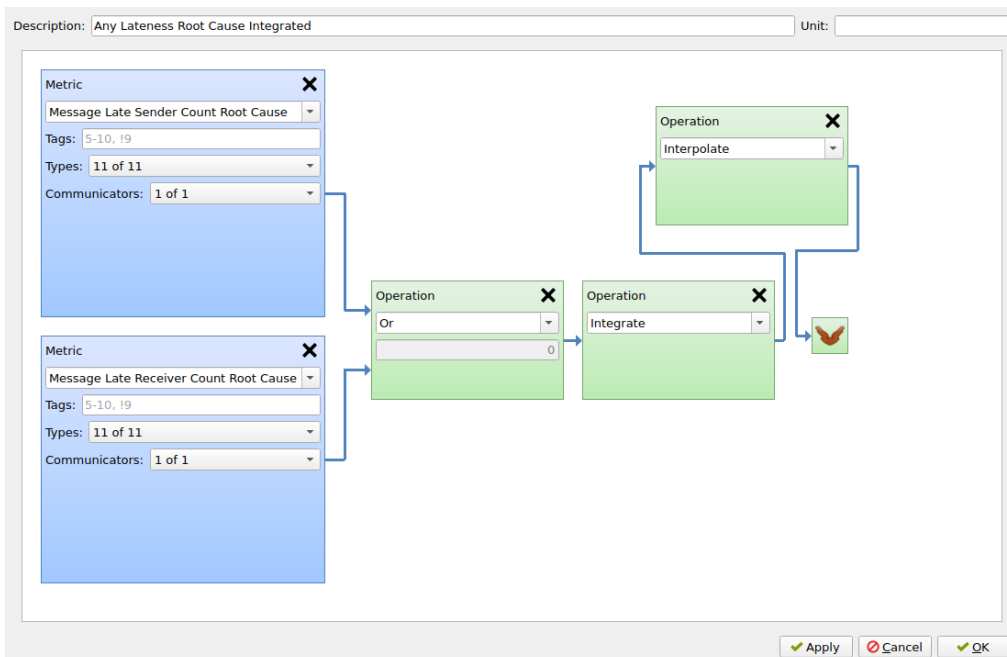
Linear interpolation of metric
values allow misalign
measurement points between
function events and metric

To do: Provide the rate of
change



Late Sender/Receiver Root Cause per Function

Integrating any 0/1 metric results in a "attribute duration to this function"-metric



Wait-time and Request Test Metrics

Message Late Sender/Receiver Count

- Number of messages inducing wait-time at any given time

Message Late Sender/Receiver Count Root Cause

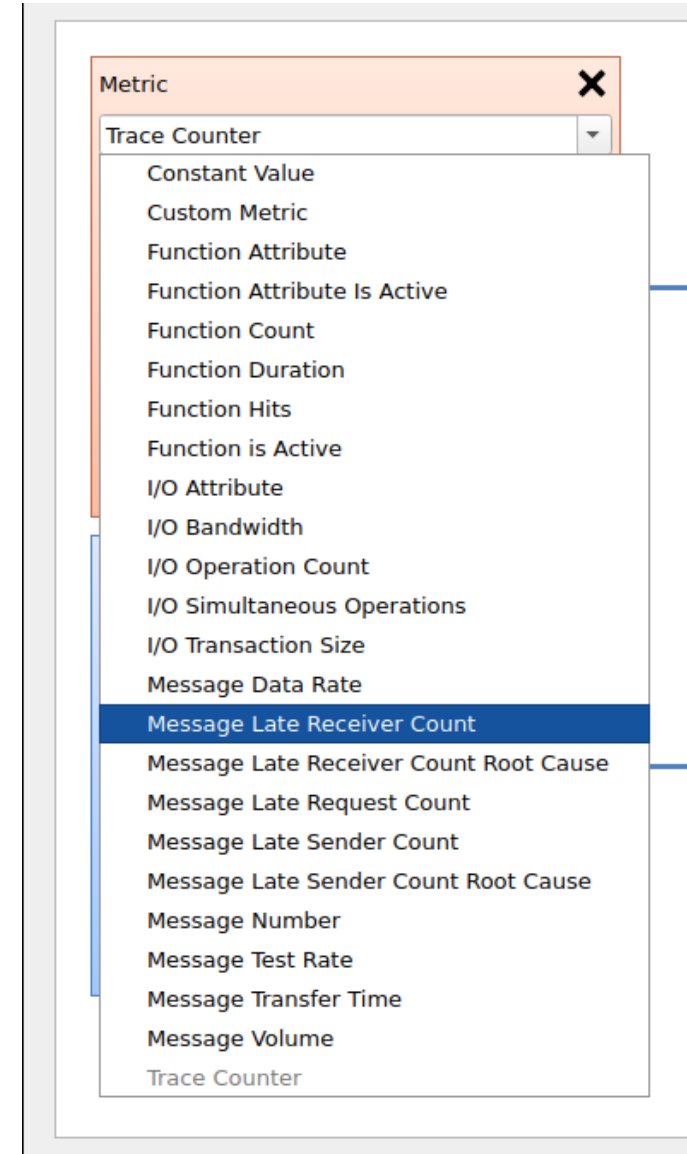
- Number of messages inducing wait-time at any given time attributed to the peer thread

Message Late Request Count

- Number of messages at any given time, which are between the last failed test and the beginning of the completion

Message Test Rate

- Rate of unsuccessful tests between first and last test event



Measurement Overhead Experiments

OpenLB 1.7r0 and rayleighTayler3D example, $n_x, n_y, n_z \in \{200, 500\}$, *maxIters* = 10, 8 and 3 repeats

Score-P 11.0-dev (92f72cede8fbfc844efb1d7feace98d375190c5e)

GCC/14.3.0, OpenMPI/5.0.8, -O3 -march=native -mtune=native

Compile-time filtering:

```
SCOREP_REGION_NAMES_BEGIN
```

```
EXCLUDE MANGLED _ZN3o1b* _ZNK3o1b* pqdownheap bi_reverse longest_match
```

Barnard cluster of TU Dresden, ZIH

2 x Intel Xeon Platinum 8470 (52 cores) @ 2.00 GHz

512 GiB RAM

4 Nodes n[1050-1052,1055] (exclusively)

Running 2 ranks per node, 1 rank on each socket

Measurement Overhead Results: Baseline

$n_x, n_y, n_z = 200$, 8 repeats

	Base	Count Failed Test Tests/s				Request_get_status
Minimum	10.203 s	10.290 s	12.6 M/s	119,987,006	11.7 M/s	10.401 s
Maximum	10.942 s	11.367 s	19.5 M/s	222,058,830	19.9 M/s	10.982 s
Mean/Median	10.545 s	10.789 s		180,973,541		10.660 s

Count failed test

- Mini LD_PRELOAD wrapper which counts all unsuccessful `MPI_Test()`

Request_get_status

- Mini LD_PRELOAD wrapper which “replaces” `MPI_Test()` with `PMPI_Request_get_status()`

Measurement Overhead Results: Baseline

$n_x, n_y, n_z = 500$, 3 repeats

	Base	Count Failed Test Tests/s				Request_get_status
Minimum	176.775 s	177.282 s	11.6 M/s	2,058,175,166	11.6 M/s	177.180 s
Maximum	177.752 s	179.975 s	13.7 M/s	2,481,510,808	13.7 M/s	177.576 s
Mean/Median	177.308 s	178.587 s		2,186,553,174		177.405 s

Large range of variability

Test rate and run time directly correlate

Measurement Overhead Results: Score-P

nx, ny, nz = 200, 8 repeats

Measurement configurations:

default: Only ENTER/LEAVE events for MPI_Test, combines successful and unsuccessful calls

xreqtest: Additional MPI_REQUEST_TEST events for each unsuccessful MPI_Test call

xnobusywait: Score-P 10.0 replaces MPI_Test with PMPI_Request_get_status, neither ENTER/LEAVE nor MPI_REQUEST_TEST events for each unsuccessful MPI_Test call

	Base	default MPI_Test ENTER/s				xreqtest Tests/s				xnobusywait
Minimum	10.203 s	10.240 s	2.2 M/s	20,990,348	2.0 M/s	10.244 s	1.5 M/s	15,545,756	1.5 M/s	10.288
Maximum	10.942 s	10.930 s	3.4 M/s	37,567,991	3.4 M/s	10.850 s	1.6 M/s	22,992,790	2.2 M/s	12.484
Mean/Median	10.545 s	10.507 s		26,797,339		10.538 s		18,503,347		11.196
Trace Size		0.7-1.5 GiB				0.7-1.5 GiB				~ 2.0 MiB

Measurement Overhead Results: Score-P

$n_x, n_y, n_z = 500$, 3 repeats

- 2 Billion unsuccessful MPI_Test calls (1.4-1.5 M/s)
- No impact on runtime
- Up to 8 GiB trace size

Conclusions

Low overhead measurement

Score-P reduces test frequency but does not increase runtime

⇒ Need for new measurement mode which captures test counts and frequencies

Measurement Overhead Results: New Score-P Mode

$n_x, n_y, n_z = 200$, 8 repeats

New measurement configurations

xreqtestsum: ENTER/LEAVE, no MPI_REQUEST_TEST, counts per message, timestamp of first and last unsuccessful MPI_Test call

xnobywait + xreqtestsum: Only events for completed messages

	Base	xnobywait + xreqtestsum MPI_Test			
		ENTER/s			
Minimum	10.203 s	10.257 s	4.1 G/s	36,761,624	3.5 M/s
Maximum	10.942 s	10.613 s	4.4 G/s	46,575,231	4.4 M/s
Mean/Median	10.545 s	10.403 s		43,209,321	
Trace Size		~ 2.0 MiB			

⇒ Trace size bound by number of messages

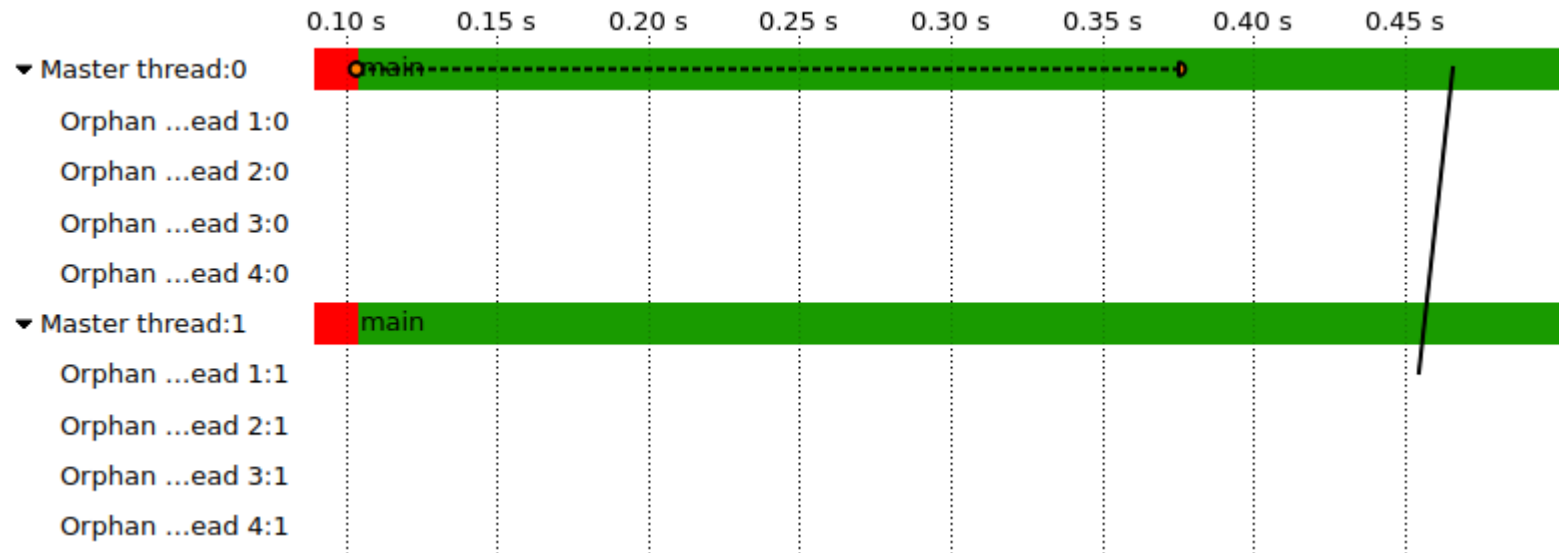
Reduced Request Test Trace Data in Vampir

Instead of recording any unsuccessful MPI_Test event, just count the number and their first and last timestamp

No more thread information

No reduction in expressiveness of request test metrics

But reduced visualization



Conclusion

Can Vampir assess whether an application performs too many or too few MPI_Test calls?

Applications do not need to optimize MPI_Test frequency—runtime is insensitive to test count, but fewer tests reduce overhead without performance loss.

Can Vampir calculate and present well-known wait-time patterns (late sender/receiver) attributed to root-cause functions?

Vampir can visualize basic wait-time patterns (late sender/receiver) and attribute them to root-cause functions via duplicated message records.

Summary

Vampir Enhancements

- Extended event model for non-blocking messages with per-test timestamps and thread context
- None-time metrics (floating point operations, root-cause) integrated into Function and Location Summary and Function Summary Timeline

Score-P Improvements

- Extended recording mode captures non-blocking MPI events (including unsuccessful tests) with negligible size and time overhead
- Measurements do not alter application behavior

Outlook

Apply to non-blocking collectives

Apply to host/device activities

Apply to asynchronous I/O operations

Apply to partitioned point-to-point communication?

Apply none-time metrics to Call Tree and Flame Graph