Reconstructing Execution States and Message Causality for an MPI Token Ring in Eclipse Trace Compass

Author's Personal Copy Technical Report

AHMED EZ-ZOUINE, M.Sc.

September 2026

Abstract

Low-overhead tracing with LTTng-UST records distributed MPI execution dynamics with minimal perturbation, yet generic trace viewers lack native semantic knowledge of custom application-level tracepoints. Consequently, default ingestion yields raw event streams rather than high-level execution states or inter-process causal relationships. In this work, we present an automated, scripted trace analysis framework for an MPI token ring executed in Eclipse Trace Compass via the Eclipse Advanced Scripting Environment (EASE). The pipeline maps operating system thread IDs to logical MPI ranks, transforms entry/exit tracepoint pairs into state intervals indexed within the disk-backed State History Tree (SHT), and resolves message exchanges into directed communication arrows following Lamport's happened-before relation. From the reconstructed timeline, we quantify a 63.865 μs startup skew across ranks, a 118.762 μs blocking wait on Rank 2 (108.715 μs under interactive selection), and an end-to-end ring round-trip latency of 119.001 μs comprising 66.435 μs network transit across four hops and 52.566 μs relay turnaround overhead. Furthermore, we address practical runtime challenges including JavaScript engine resolution under Rhino versus Nashorn, platform stability on Apple Silicon (macOS ARM64), and index completion requirements via `closeHistory()`.

Keywords: MPI Tracing, High-Performance Computing (HPC), LTTng-UST, Eclipse Trace Compass, State History Tree, Causality Analysis, EASE Scripting, Distributed Systems, Performance Analysis.

1. Introduction

1.1 Motivation for Tracing MPI Programs

Parallel programs communicating via the Message Passing Interface (MPI) present significant performance debugging challenges that sampling profilers cannot adequately address. While statistical profilers capture aggregate CPU consumption across functions, they often miss microsecond-scale blocking periods and cannot capture inter-process causal dependencies. Tracing addresses this limitation: frameworks like LTTng record timestamped events from user-space applications (LTTng-UST) into per-CPU lockless ring buffers with minimal runtime perturbation [1]. System trace viewers such as Eclipse Trace Compass subsequently reconstruct, query, and visualize this temporal execution data [5].

However, while Trace Compass natively understands standard Linux kernel and POSIX events, it lacks intrinsic semantic models for custom user-space tracepoints. When ingesting an instrumented MPI application without a specialized state provider, the tool treats tracepoints merely as opaque entries in a flat event list.

1.2 Limitations of Default Ingestion

Opening the raw trace in Trace Compass reveals only aggregate event counts in the Statistics view and uninterpreted tables, as illustrated in Figure 1. Out of the box, the tool cannot infer that:

1. each operating system thread ID (`_vtid`) corresponds to a distinct logical MPI rank;
2. paired `entry` and `exit` tracepoints bound continuous operational states (e.g., blocking communication waits);
3. a send event on rank i causally precedes and pairs with a matching receive event on rank j.

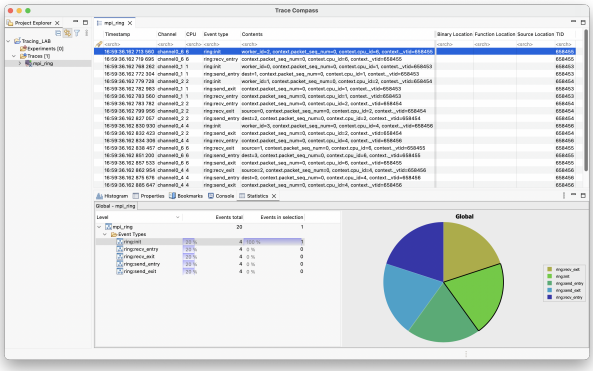


Figure 1: Default ingestion of the MPI trace in Trace Compass. Without a custom state provider, only aggregate event counts (Statistics view) and raw event tables are available; execution states and message flow remain uninterpreted.

1.3 Scripted Analysis Overview

Trace Compass provides extensibility without requiring compiled Java OSGi plugins through the Eclipse Advanced Scripting Environment (EASE), which executes JavaScript or Python scripts against the internal analysis APIs. In this work, we develop an automated analysis pipeline using five modular JavaScript scripts that:

- inspect raw event records and validate payload schemas (`readtrace.js`);
- map operating system thread IDs to logical MPI ranks using initialization events (`readEvents.js`);
- construct persistent execution states within the State History Tree (`stateSystem.js`);
- project reconstructed states onto a graphical time graph (`timeLine.js`);
- correlate send/receive event pairs and render causality arrows (`timeGraphArrow.js`).

From this reconstructed view, we extract communication latencies, verify temporal causality, and resolve practical deployment bottlenecks.

2. Background

2.1 Trace Events and CTF

An execution trace $\mathcal{T}$ is an ordered sequence of timestamped events:

$$\mathcal{T} = \langle e_1, e_2, \ldots, e_M \rangle, \quad t(e_k) \leq t(e_{k+1}) \tag{1}$$

where each event is modeled as a tuple:

$$e_k = \langle t_k, c_k, \text{name}_k, \mathbf{X}_k \rangle \tag{2}$$

with t_k representing the nanosecond timestamp, c_k the CPU core, name_k the tracepoint identifier, and $\mathbf{X}_k$ the contextual payload (process ID, virtual thread ID `_vtid`, and application fields such as `worker_id`, `source`, and `dest`).

LTTng records these records into per-CPU circular buffers formatted in Common Trace Format (CTF) [3]. Upon ingestion by Trace Compass, individual streams are merged into a unified, chronologically sorted event stream.

2.2 The State System and the State History Tree

Trace events represent instantaneous points in time. To reconstruct continuous operational states (e.g., waiting, computing, communicating), Trace Compass employs the State System [2]. Attributes are organized hierarchically as tree paths identified by integer quarks. For each quark q, the state system maintains a sequence of non-overlapping intervals:

$$\mathcal{I}(q) = \left\{ [t^{(i)}_{\text{start}}, t^{(i)}_{\text{end}}) \mapsto v^{(i)} \right\}, \quad t^{(i)}_{\text{start}} < t^{(i)}_{\text{end}} \leq t^{(i+1)}_{\text{start}} \tag{3}$$

where $v^{(i)}$ denotes the state value. Consecutive intervals on a quark touch end-to-start when the attribute is updated; when an attribute is explicitly removed upon an exit event, the quark holds no value, producing idle gaps on the timeline.

These intervals are indexed in the State History Tree (SHT), a disk-backed, self-balancing tree structure designed to answer state queries at arbitrary timestamps in $O(\log N)$ time, ensuring responsive navigation across multi-gigabyte traces [2].

Because the SHT buffers intervals in memory until their closing timestamp is determined, state providers must explicitly finalize the index:

$$\texttt{closeHistory}(t_{\text{final}}) \tag{4}$$

This seals pending intervals and writes trailing tree nodes to disk. Omitting this step leaves the index unclosed, resulting in visualization freezes (Section 5.3).

2.3 Lamport's Happened-Before Relation

Physical timestamps across distributed processes do not inherently prove causality. Our analysis relies on Lamport's happened-before relation $\prec$ [4]:

1. Within a single process, events follow local sequential order: if $t(a) < t(b)$, then $a \prec b$.
2. A message transmission causally precedes its reception: $e_{\text{send}} \prec e_{\text{recv}}$.
3. The relation is transitive: if $a \prec b$ and $b \prec c$, then $a \prec c$.

Each matched communication pair is projected as a directed arrow from e_{send} to e_{recv} on the timeline.

3. Implementation

The analysis is implemented modularly across five JavaScript scripts executed inside EASE.

3.1 Step 1: Reading Trace Events (*readtrace.js*)

The initial script establishes programmatic access to the active trace and iterates through its events, extracting names and payload schemas:

```
loadModule("/TraceCompass/Trace");
var trace = getActiveTrace();
var iter = getEventIterator(trace);

while (iter.hasNext()) {
    var event = iter.next();
    var fields = event.getContent().getFieldNames();
    // Print event name and payload fields
}
```

As illustrated in Figure 2a, the trace contains 20 events, and all required payload fields (`context._vtid`, `worker_id`, `source`, and `dest`) are present and validated.

3.2 Step 2: Mapping Threads to MPI Ranks (*readEvents.js*)

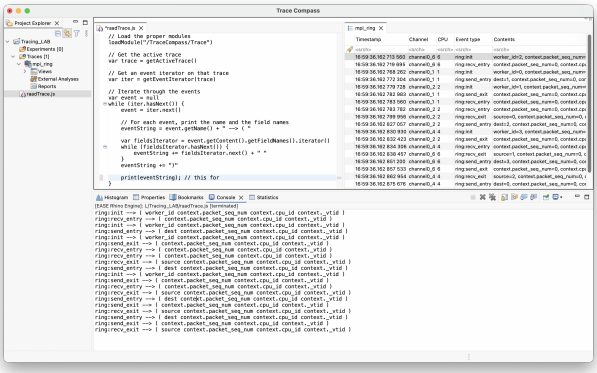
MPI processes manifest as distinct OS threads whose kernel thread IDs vary across executions. To associate thread IDs with logical ranks, each process logs its identifier via a `ring:init` tracepoint during initialization:

```
var tidToWorkerMap = {};
if (event.getName().equals("ring:init")) {
    var wid = event.getContent().getValue("worker_id");
    var tid = ↪
        ↪ event.getContent().getValue("context._vtid");
    tidToWorkerMap[tid] = wid;
}
```

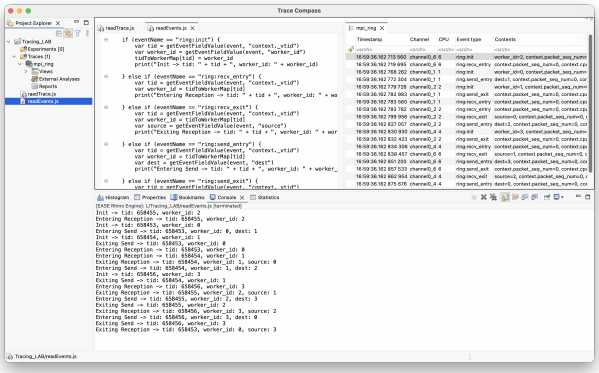
This establishes a deterministic mapping for this execution:

- Thread 658453 ↦ Rank 0 (token origin)
- Thread 658454 ↦ Rank 1
- Thread 658455 ↦ Rank 2
- Thread 658456 ↦ Rank 3

Figure 2b displays the console output: every event is attributed to its originating rank alongside peer communication targets.



(a) `readtrace.js`: inspecting raw tracepoints and payload fields.



(b) `readEvents.js`: mapping OS thread IDs to MPI ranks.

Figure 2: Console outputs for Steps 1 and 2: verifying event fields (a) and attributing events to logical MPI ranks (b).

3.3 Step 3: Building the State System (*stateSystem.js*)

The subsequent script translates entry/exit tracepoints into persistent state intervals. The reconstructed states per rank are defined as:

$$\begin{aligned} \texttt{recv\_entry} &\implies \text{state "Waiting for reception"} \\ \texttt{recv\_exit} &\implies \text{clear state} \\ \texttt{send\_entry} &\implies \text{state "Sending"} \\ \texttt{send\_exit} &\implies \text{clear state} \end{aligned}$$

In the Trace Compass EASE scripting API:

```
var quark = ↪
    ↪ ss.getQuarkAbsoluteAndAdd(worker_id.toString());
if (name.equals("ring:recv_entry")) {
    ss.modifyAttribute(t, "Waiting for reception", ↪
        ↪ quark);
} else if (name.equals("ring:recv_exit")) {
    ss.removeAttribute(t, quark);
}
```

where `ss` represents the custom state system obtained from `createScriptedAnalysis()`. Following event stream exhaustion, the script invokes `ss.closeHistory(t)` to seal the tree (Section 5.3).

Figure 3a shows the resulting attribute tree inside the State System Explorer: distinct quarks track each rank, housing contiguous state intervals.

3.4 Step 4: Displaying the Timeline (*timeLine.js*)

To render state intervals graphically, the script instantiates a time graph data provider and activates the timeline view:

```
loadModule("/TraceCompass/DataProvider");
loadModule("/TraceCompass/View");

var map = new java.util.HashMap();
map.put(ENTRY_PATH, '*');
var provider = createTimeGraphProvider(analysis, map);
openTimeGraphView(provider);
```

While states are visualized correctly (Figure 3b), rows initially appear ordered by first event timestamp $(2, 0, 1, 3)$ rather than numerical rank, as Rank 2 was scheduled prior to Rank 0.

3.5 Step 5: Drawing Communication Arrows (*timeGraphArrow.js*)

The final script enforces rank sorting and introduces message causality arrows. During event iteration, encountering a `ring:send_entry` on rank i targeting destination $d = (i + 1) \pmod N$ registers an in-flight message:

$$\text{pending}[d] = \langle t_{\text{send}}, \text{source} : i, \text{dest} : d \rangle \tag{5}$$

When destination rank d exits its receive phase, the pending record is resolved with $t_{\text{recv}} = t(\texttt{recv\_exit})$:

```
pending = pendingArrows[worker_id];
if (pending != null) {
    pendingArrows[worker_id] = null;
    pending["endTime"] = event.getTimestamp().toNanos();
    arrows.push(pending);
}
```

An essential architectural consideration arises here: arrow primitives cannot be instantiated during single-pass event iteration because time graph entries and their associated internal IDs are only assigned after all entries are populated. The script therefore collects entries, sorts them numerically by rank $(0, 1, 2, 3)$, and subsequently builds the arrow collection in a secondary pass:

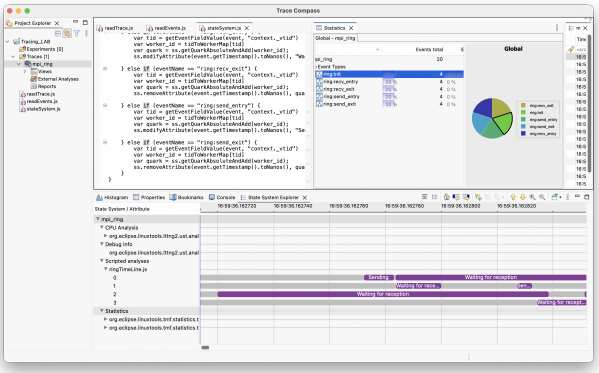
```
mpiEntries.sort(function(a,b){
    return Number(a.getName()) - Number(b.getName());
});
for (var i = 0; i < arrows.length; i++) {
    var arrow = arrows[i];
    var srcId = mpiWorkerToId[String(arrow["source"])];
    var dstId = mpiWorkerToId[String(arrow["dest"])];
    var duration = arrow["endTime"] - arrow["time"];
    if (srcId != null && dstId != null) {
        tgArrows.getList().add(
            createArrow(srcId, dstId, arrow["time"], ↪
                ↪ duration, 1)
        );
    }
}
```

Figure 4 displays the complete reconstructed view: ranks are aligned numerically, with arrows tracing the token around the ring.

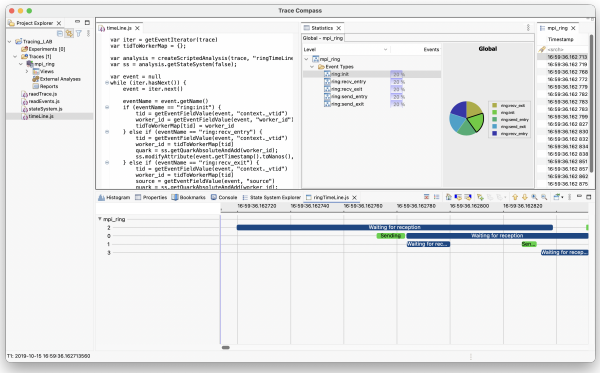
4. Measurements and Causality Analysis

4.1 Timing Data

Table 1 presents the exact timestamps and wait intervals derived from the reconstructed trace. All absolute times-



(a) State System Explorer: states stored per rank in the SHT.



(b) Initial timeline: rows appear in arrival order $(2, 0, 1, 3)$.

Figure 3: Steps 3 and 4: saving state intervals in the State History Tree (a) and drawing the first unsorted timeline (b).

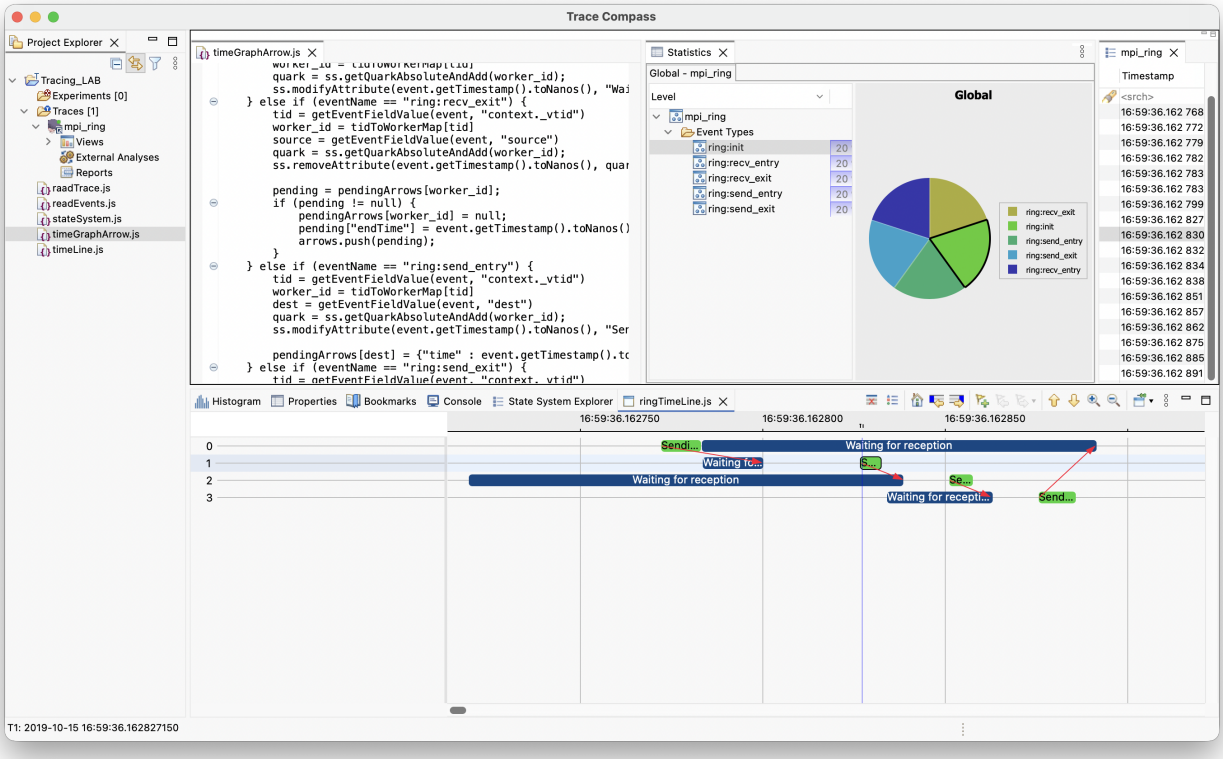


Figure 4: Final reconstructed Time Graph from `timeGraphArrow.js`. Rows are sorted by rank (0 to 3), colored bands mark the "Waiting for reception" and "Sending" states, and directed arrows trace the token around the ring: Rank $0 \to 1 \to 2 \to 3 \to 0$.

tamps represent nanosecond readings within the second starting at 11:59:36 (162,783,560 ns = 162.783560 ms). Event boundaries were cross-verified using an independent binary CTF parser (`ctf_ring_metrics.py`) and validated against interactive timeline selections in Trace Compass.

4.2 Quantitative Observations

Analysis of Table 1 yields four principal findings:

1. **Startup skew (63.865 μs):** Rank 2 entered its receive wait at 162.719695 ms, whereas Rank 0 entered its wait (subsequent to initiating token transfer) at 162.783560 ms, representing an initial skew of 63.865 μs. (Across process initialization points, tracepoints span 117.370 μs from Rank 2 to Rank 3). The operating system scheduled Rank 2 earlier, causing it to block passively while the token traversed Ranks 0 and 1. This startup disparity explains why Rank 2 exhibits the highest cumulative wait (118.762 μs, measured as 108.715 μs under the visual selection bracket in Figure 4) despite executing identical algorithmic work.

2. **Hop-by-hop transmission (66.435 μs):** Network transit delays between consecutive ranks show clear latency asymmetry:

- **Hop 1 ($0 \to 1$):** Rank 0 begins sending at 162.772304 ms (`send_entry`) and Rank 1 completes reception at 162.799956 ms (`recv_exit`), requiring 27.652 μs. Rank 1 spent 16.174 μs blocked. This duration reflects that Rank 1 posted its receive at

Table 1: Measured Wait Intervals for Each MPI Rank (durations computed from exact trace timestamps).

MPI Rank	Thread ID	Wait Start	Wait End	Wait Duration	Role in the Ring
Rank 0	658453	11:59:36.162 783 560	11:59:36.162 891 305	**107.745** μs	Sends token, waits for full loop
Rank 1	658454	11:59:36.162 783 782	11:59:36.162 799 956	**16.174** μs	Receives from Rank 0, relays to Rank 2
Rank 2	658455	11:59:36.162 719 695	11:59:36.162 838 457	**118.762** μs	Posted its receive before Rank 0 started
Rank 3	658456	11:59:36.162 834 306	11:59:36.162 862 954	**28.648** μs	Receives from Rank 2, returns token to Rank 0

162.783782 ms (11.478 μs after Rank 0 initiated transmission).

- **Hop 2 (1 → 2):** Rank 1 initiates sending at 162.827057 ms and Rank 2 completes reception at 162.838457 ms, taking 11.400 μs.
- **Hop 3 (2 → 3):** Rank 2 initiates sending at 162.851200 ms and Rank 3 completes reception at 162.862954 ms, taking 11.754 μs. Rank 3 was already waiting for 28.648 μs.
- **Hop 4 (3 → 0):** Rank 3 initiates sending at 162.875676 ms and Rank 0 completes reception at 162.891305 ms, taking 15.629 μs.

Summing these intervals yields total in-flight network transit latency: $T_{\text{hops}} = 27.652 + 11.400 + 11.754 + 15.629 = 66.435$ μs.

3. **Intermediate relay turnaround (52.566 μs):** Between message reception and subsequent forwarding, each relay rank incurs local CPU turnaround overhead (MPI protocol processing, loop logic, and context switching):

- **Rank 1 Turnaround:** $162.827057 - 162.799956 = 27.101$ μs.
- **Rank 2 Turnaround:** $162.851200 - 162.838457 = 12.743$ μs.
- **Rank 3 Turnaround:** $162.875676 - 162.862954 = 12.722$ μs.

Total intermediate relay turnaround is $T_{\text{turnaround}} = 27.101 + 12.743 + 12.722 = 52.566$ μs. Rank 1 incurs more than double the turnaround of Ranks 2 and 3, consistent with cold cache penalties or OS scheduling quantum boundaries.

4. **Total ring latency (119.001 μs):** Rank 0 initiated transmission at 162.772304 ms and received the returned token at 162.891305 ms. The total elapsed round-trip duration is:

$$T_{\text{ring}} = t^{(0)}_{\text{recv\_exit}} - t^{(0)}_{\text{send\_entry}} = 119.001\ \mu\text{s} \quad (6)$$

$$= T_{\text{hops}} + T_{\text{turnaround}} = 66.435 + 52.566\ \mu\text{s} \quad (7)$$

(or 107.745 μs when restricted strictly to Rank 0's blocking wait state). The exact convergence of both formulations validates the temporal precision of the instrumentation. Lamport's happened-before relation $\prec$ directly explains why Rank 2's wait (118.762 μs) exceeds Rank 0's wait (107.745 μs): Rank 2 posted its receive prior to the causal wavefront reaching it, remaining blocked concurrently as shown in Figure 5.

5. Engineering Considerations and Troubleshooting

Deploying scripted analyses in Trace Compass reveals several practical engineering considerations.

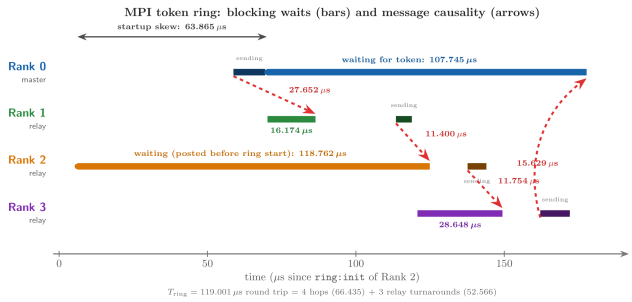


Figure 5: Causality model of the token ring: timeline of rank wait states, message hops, and startup skew.

5.1 JavaScript Engines: Rhino vs. Nashorn

Initial execution of the arrow script triggered a runtime exception: `ScriptExecutionException: SyntaxError (Cannot convert * to java.lang.Integer)`.

EASE supports multiple JavaScript engines. Mozilla Rhino, the legacy engine, coerces untyped numeric values into boxed Java Integer types during method overload resolution. This breaks 64-bit nanosecond timestamps and string wildcards (e.g., `'*'`). Conversely, Oracle Nashorn leverages Java 7+ `invokedynamic` and correctly resolves overloaded methods accepting native `long` values. Selecting Nashorn in the EASE runtime configuration resolves these type coercion errors.

Furthermore, legacy examples often rely on Rhino-specific adapters:

```
// Rhino-only syntax, rejected under Nashorn
new JavaAdapter(Function, { ... });
```

This is replaced with standard JavaScript functions, which Nashorn binds seamlessly to Java functional interfaces:

```
function getEntriesFunction(params) { return ↪
    ↪ tgEntries.getList(); }
function getArrowsFunction(params) { return ↪
    ↪ tgArrows.getList(); }
var provider = createScriptedTimeGraphProvider(
   analysis, getEntriesFunction, null, ↪
       ↪ getArrowsFunction
);
```

This syntax is portable and eliminates runtime adapter overhead.

5.2 Platform Stability on Apple Silicon (macOS ARM64)

When executing on Apple Silicon (macOS ARM64), Trace Compass exhibited occasional JVM crashes during scripted time graph rendering, in contrast to stable execution under x86_64 Linux. This behavior appears related to platform-specific JNI/SWT bindings within Eclipse RCP on ARM64 architectures. The Trace Compass source code[1] was examined to isolate minimal

[1]Trace Compass repository: https://github.com/eclipse-tracecompass/org.eclipse.tracecompass

reproduction steps for upstream issue tracking. Resolving such architectural edge cases is increasingly vital as ARM64 workstations become ubiquitous in research environments.

5.3 Index Finalization via `closeHistory()`

Omitting `ss.closeHistory()` causes Trace Compass to freeze when loading timeline views. In the State History Tree, state intervals remain buffered in memory until their terminating timestamp is recorded. Finalizing the index via:

```
ss.closeHistory(event.getTimestamp().toNanos());
```

writes out the remaining leaf and internal tree nodes, allowing the viewer to query intervals bounding the final timestamp without hanging.

6. Discussion and Future Work

While demonstrated on a canonical 4-process token ring, this scripted causality analysis provides a foundation for addressing key challenges in large-scale parallel tracing.

6.1 Extending to Non-Blocking MPI Communications

The single pending slot per destination utilized here suits deterministic blocking communication. Real-world MPI applications rely heavily on non-blocking primitives (`MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall`), where multiple messages remain concurrently in flight. Generalizing the matching logic requires indexing pending transfers by a 3-tuple: $\langle$`communicator`, `tag`, `source`/`dest`$\rangle$. Validating this matching against instrumented MPI libraries such as LTTng-UST MPI wrappers, alongside validation with network simulators such as LogGOPSim [6], represents an immediate next step.

6.2 Scalability for Large-Scale Traces

While a 4-rank trace processes instantaneously in memory, scaling to 10^4+ ranks with millions of messages necessitates streaming event processing. In-memory associative tables become unsustainable at scale, shifting the performance bottleneck to SHT disk I/O. Investigating parallelized SHT construction and hybridizing declarative XML state providers for low-level parsing with scripted providers for complex matching represents a promising architectural path.

6.3 Automated Critical-Path Extraction

Directed message arrows allow visual inspection of causal chains. Because matched communication pairs form a directed acyclic graph (DAG) governed by Lamport's happened-before relation, the critical path determining total execution time can be extracted algorithmically:

$$\mathcal{P}_{\text{crit}} = \arg \max_{\mathcal{P} \in \text{Paths}} \sum_{e \in \mathcal{P}} \text{weight}(e) \tag{8}$$

Integrating automated critical-path identification directly into Trace Compass would transform the time graph from an observational display into an actionable performance diagnosis tool.

6.4 Hybrid MPI and OpenMP Multi-Paradigm Tracing

Modern scientific workloads increasingly rely on hybrid programming models combining distributed MPI communication with shared-memory OpenMP multithreading within compute nodes. While inter-node token propagation is governed by MPI messaging, intra-node execution manifests as fine-grained parallel regions and task graphs. Reconstructing unified causal dependency graphs across both paradigms requires correlating LTTng-UST MPI wrapper events with OpenMP runtime tracepoints on identical OS threads. In such hybrid models, MPI communications represent macroscopic synchronization edges, whereas nested OpenMP barriers and parallel sections form chains of fine-grained intra-process dependencies. Extending the state-tracking and arrow-matching algorithms developed here to hybrid MPI+OpenMP runtimes represents a promising avenue for automated critical-path analysis on multi-node clusters.

7. Conclusion

Starting from an uninterpreted LTTng-UST trace, this paper demonstrates an automated, scripted methodology in Eclipse Trace Compass to map OS threads to MPI ranks, reconstruct continuous execution states in the State History Tree, and visualize communication causality arrows. The reconstructed timeline enables precise quantitative analysis: resolving a 63.865 μs startup skew, explaining Rank 2's 118.762 μs blocking wait, and decomposing the 119.001 μs round trip into 66.435 μs in-flight network transit and 52.566 μs relay turnaround overhead. Furthermore, addressing runtime engine compatibility (Rhino vs. Nashorn), platform stability on ARM64, and index closure requirements provides practical engineering guidance for researchers developing extensible tracing workflows.

All scripts, trace data, and evaluation tooling are open-source and available at: https://github.com/ahmedez-zouine/LAB_MPI-Tracing-Causality-Analysis.

References

[1] M. Desfossez, J. Boucher, and M. R. Dagenais, "LTTng: High-performance tracing engine for Linux," in *Proceedings of the Linux Symposium*, Ottawa, Canada, 2012, pp. 63–74.

[2] A. Montplaisir-Gagné, M. Eichelberger, and M. R. Dagenais, "State history tree: An index for efficient state reconstruction in execution traces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2404–2417, 2013.

[3] M. R. Dagenais and K. Yaghmour, "Measuring and analyzing system metrics with the Linux Trace Toolkit," *IEEE Software*, vol. 21, no. 1, pp. 64–70, 2004.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[5] M. Eichelberger and M. R. Dagenais, "Trace Compass: An extensible system trace analysis and visualization tool," in *EclipseCon North America*, Reston, VA, 2012.

[6] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: Simulating large-scale MPI applications with non-blocking communications," *Simulation Modelling Practice and Theory*, vol. 18, no. 8, pp. 1109–1122, 2010.

[7] DORSAL Research Group, "Scripted analysis for custom instrumentation," in *Tracevizlab Series*, 2024. [Online]. Available: https://github.com/dorsal-lab/Tracevizlab