Parallel Programming
Lecture 12

Prof. Dr. Jonas Posner

jonas.posner@cs.hs-fulda.de

Hochschule Fulda
University of Applied Sciences

Winterterm 2025/2026

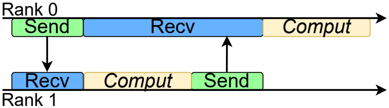
E-Learning: AI5085

30th January 2026

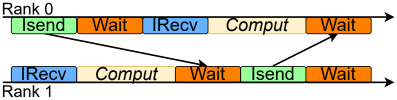
Plan for Today

- **Register for a presentation slot!**
  - Deadline: January 30 at 11:59 p.m.
- Introduction to Performance Analysis
  - Concepts and terminology
  - Tools and workflow
  - Get into practice

Experiment: Ping Pong—Blocking vs. Non-Blocking



VS.



- PING_PONG_LIMIT = 100
- Computation = 10 ms
- Two processes

Variant	**Overlap?**	**Measured Time**	**Speedup**
Blocking	✘ No	$\approx 1.00\,\mathrm{s}$	1
Non-Blocking	✔ Yes	$\approx 0.50\,\mathrm{s}$	2

- **How to "see" this in real programs? (From Lecture 10)**
  Today!

Motivation for Performance Analysis

- Goals for high-quality parallel programs:
  - **Correctness:** no races, deadlocks, or mismatched communication
  - **Efficiency:** high utilization of available resources
  - **Scalability:** efficiency holds as problem size and process count grow
- Performance analysis tools help to:
  - Identify bottlenecks and inefficiencies
  - Validate optimizations with empirical data
  - Understand complex parallel behavior

Instrumentation

- **Instrumentation:** Add code to collect performance data
  - Manually (developer)
  - Automatically (compiler)
  - Via pre-instrumented libraries (e.g., MPI wrappers)
- **Static instrumentation:** Modify code at compile time
  - Analysis after execution (*post-mortem*)
  - Recompile to change instrumentation
- **Dynamic instrumentation:** Modify at runtime
  - Operates directly on the binary
  - No recompilation; flexible but potentially higher overhead

Profiling

- Profiling collects **summary statistics** about program behavior
  - **Event-based:** Metrics per event (e.g., function calls)
  - **Sampling-based:** Periodic interrupts capture program state
- Typical metrics in MPI programs:
  - Number and duration of MPI calls
  - Time in computation vs. communication
- Profile data is **aggregated**
  - Temporal ordering is not preserved
  - Good overview, not a detailed timeline
- Lower overhead than tracing; suitable for long-running applications

Tracing

- Tracing records **timestamped events** during execution
  - Captures *when* and *where* each event happened
  - Preserves temporal ordering and causality
- Trade-offs compared to profiling:
  - Higher storage requirements and runtime overhead
  - More detail; supports root cause analysis
  - Requires specialized tooling for analysis

Example

```
Trace:
[2021-06-12T11:22:09.815479Z] [INFO] [Thread-1] Request started
[2021-06-12T11:22:09.935612Z] [INFO] [Thread-1] Request finished
[2021-06-12T11:22:59.344566Z] [INFO] [Thread-1] Request started
[2021-06-12T11:22:59.425697Z] [INFO] [Thread-1] Request finished
Profile:
2 "Request finished" events
2 "Request started" events
```

Performance Analysis

- Based on data collected during instrumentation and measurement
- Targets for identification:
  - Inefficient code sections
  - Communication bottlenecks
  - Load imbalances
  - Resource contention (e.g., oversubscribed cores)
  - Suboptimal parallelization patterns
- Analysis is often performed *post-mortem*
  - Enables detailed examination with minimal runtime interference

Feedback and Visualization

- Presents analysis results to the developer
- Common feedback mechanisms:
  - Source code annotations highlighting problematic regions
  - Automatically generated reports (text, HTML, PDF)
  - Interactive graphical user interfaces (GUIs)
- Effective visualization enables:
  - Rapid identification of performance issues
  - Correlation between metrics and source code
  - Comparison across different runs or configurations

Debugging Parallel Programs

- Goal: identify and diagnose program errors
- Common errors in parallel programs:
  - **Race conditions:** Non-determinism due to unsynchronized access
  - **Deadlocks:** Circular wait dependencies prevent progress
- MPI-specific issues:
  - **Mismatched send/receive:** message size, datatype, tag, communicator
  - Buffer-related problems (e.g., resource exhaustion)

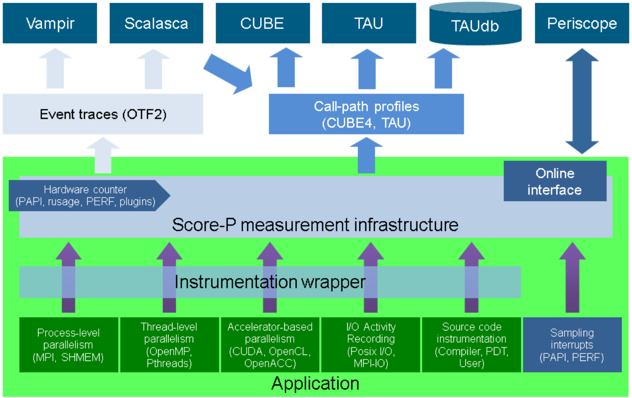
Debugging Tools

- Commercial debuggers for parallel MPI applications:
  - ARM DDT (part of ARM Forge)
  - TotalView (Perforce)
- Both offer comprehensive feature sets:
  - Multi-process and multi-threaded debugging
  - GPU debugging support (CUDA, OpenCL)
  - Memory debugging and leak detection
- Open-source alternatives (with limited features):
  - MUST (MPI correctness checking)
  - STAT (stack trace analysis)
  - ISP (MPI verification)

Instrumentation and Measurement Tools

- Focus on data collection; separate tools needed for analysis and visualization
- **Dyninst**
  - Widely-used API for dynamic binary instrumentation
  - Foundation for tools like TAU and OpenSpeedShop
  - Enables runtime analysis without recompilation
- **Score-P**
  - Popular tool for static instrumentation
  - Unified measurement infrastructure for multiple analysis tools
  - *Details on following slide*

Instrumentation and Measurement Tools



Score-P: `https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/index.html`

Score-P: Scalable Performance Measurement

- *Scalable Performance Measurement Infrastructure for Parallel Codes*
- Output formats:
  - Call-path profiles (`CUBE4` format)
  - Event traces (`OTF2` format)
  - Compatible with Cube, Vampir, Scalasca
- Open Source; portable to "all" HPC systems
- Maintained by Jülich Supercomputing Centre (JSC)
- Usage:
  - Compile: `scorep mpicc -O2 pingpong.c -o pingpong`
  - Run: `mpirun -np 2 ./pingpong`
- `https://www.score-p.org`

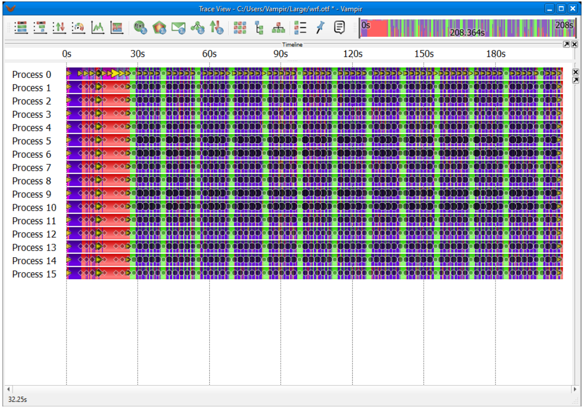
Cube and Vampir: Visualization Tools

- **Cube**
  - Interactive visualization of `CUBE4` profiles
  - Three data views (Metric, Program, and System)
  - Open Source; maintained by JSC
  - `https://www.scalasca.org/software/cube-4.x`
- **Vampir**
  - Interactive visualization of `OTF2` traces
  - Timeline view of application activities and communication
  - Commercial
  - `https://www.vampir.eu`

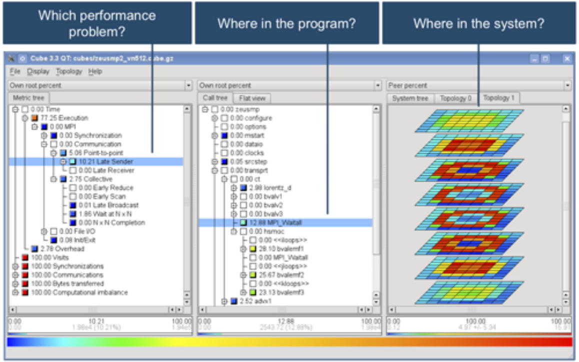
Scalasca: Automated Trace Analysis

- Scalable trace-based performance analysis toolset
- **Key feature:** Automatic pattern detection
  - Identifies common inefficiency patterns (e.g., Late Sender/Receiver)
  - Quantifies wait-state overhead
- Open Source; maintained by JSC
- Workflow (requires Score-P instrumentation):
  - Run: `scalasca -analyze -t mpirun -np 2 ./pingpong`
  - Analyze: `scalasca -examine ./scorep_pingpong_trace`
- `https://www.scalasca.org/scalasca/software/scalasca-2.x`

Vampir and Scalasca

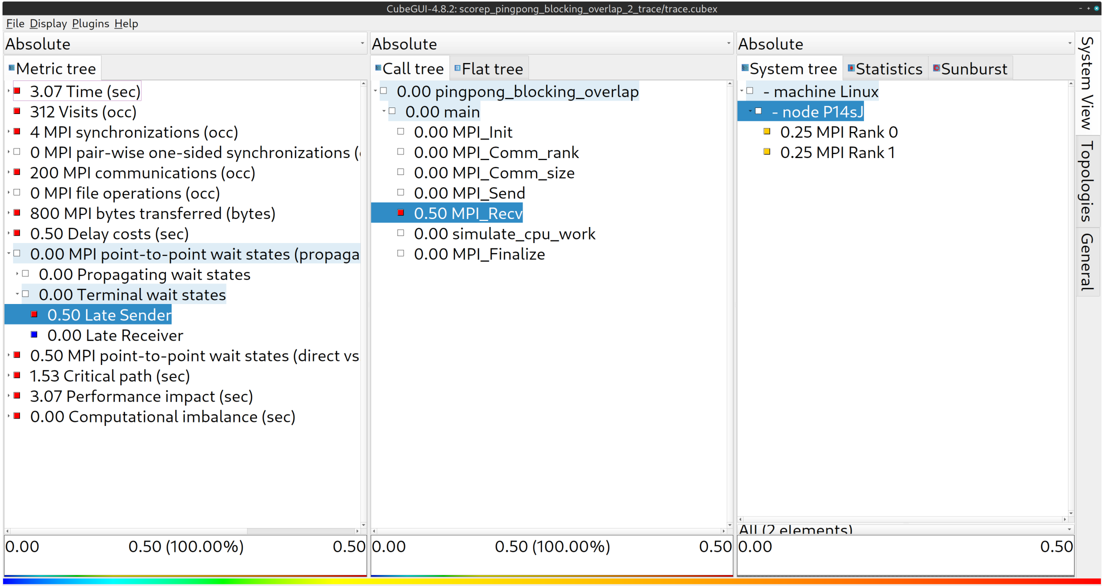


Vampir: `https://vampir.eu`

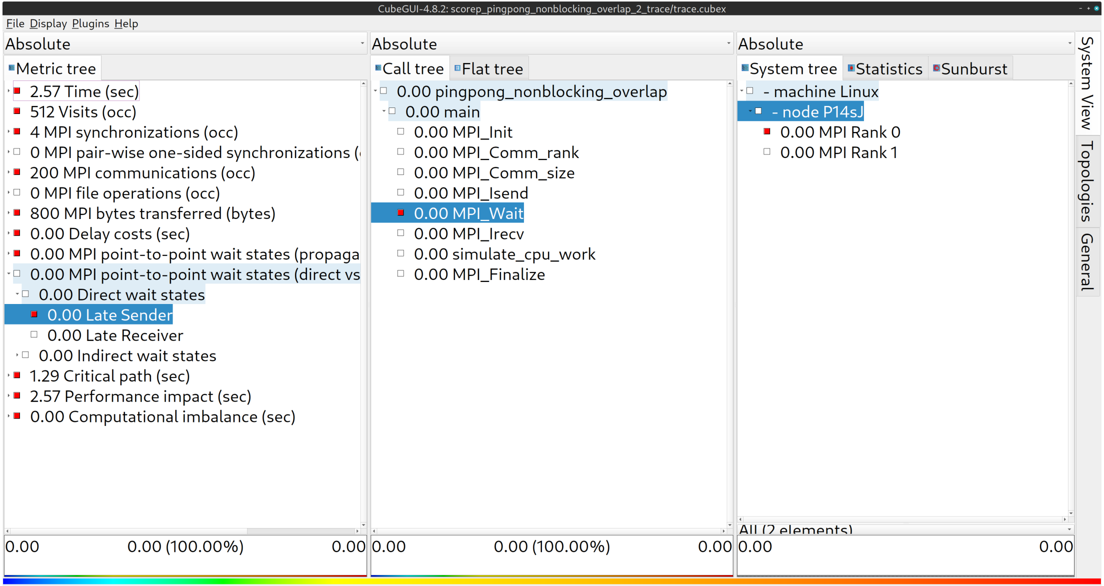


Scalasca: `https://www.scalasca.org/scalasca`

Scalasca: Ping Pong (Blocking)—Late Sender



Scalasca: Ping Pong (Non-Block)—No Late Sender



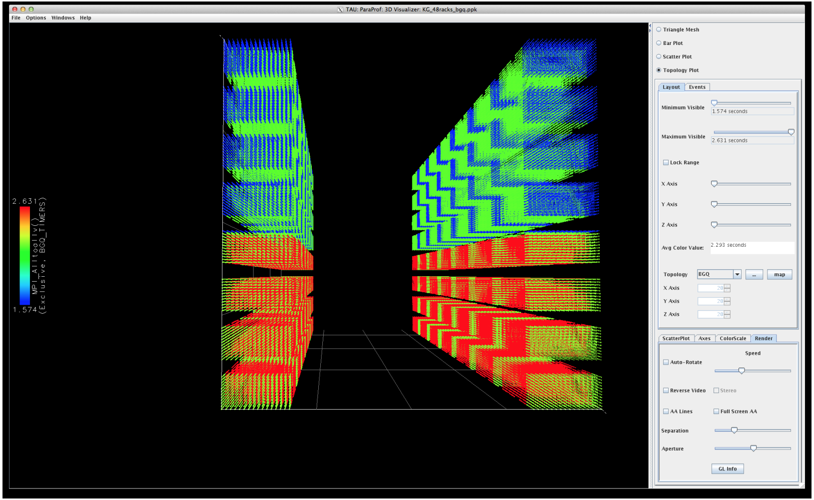
Late Sender and Late Receiver Problems

- Communication-induced idle time due to timing mismatches
- **Late Sender**
  - Receiver blocks waiting for a message not yet sent
  - Common with blocking `MPI_Recv`
  - Causes receiver-side idle time
- **Late Receiver**
  - Sender may block if the receiver has not posted a matching receive
  - Typical with synchronous sends (`MPI_Ssend`) or rendezvous protocol (large messages)
  - Can lead to buffer pressure/exhaustion with buffered sends (`MPI_Bsend`) or heavy eager buffering
- Both patterns are detectable via trace analysis (e.g., Scalasca)

All-In-One Tools

- Combine instrumentation, measurement, analysis, and visualization
- **TAU (Tuning and Analysis Utilities)**
  - Multiple instrumentation options (source, compiler, dynamic)
  - Supports MPI, OpenMP, CUDA, OpenCL, and hybrid models
  - Post-mortem visualization in 2D and 3D
  - Topology view: maps performance data to system architecture
  - Open Source; developed at University of Oregon

TAU



TAU: http://www.cs.uoregon.edu/research/tau/home.php

Summary

- Communication overhead significantly impacts parallel performance
  - Blocking communication introduces synchronization-induced idle time
  - Non-blocking communication enables computation-communication overlap
- Tools like Score-P, Cube, Vampir, and Scalasca help identify performance bottlenecks
  - Visualization of timings, communication patterns, etc.
  - Identification of Late Sender / Late Receiver problems

Summary

- Communication overhead significantly impacts parallel performance
  - Blocking communication introduces synchronization-induced idle time
  - Non-blocking communication enables computation-communication overlap
- Tools like Score-P, Cube, Vampir, and Scalasca help identify performance bottlenecks
  - Visualization of timings, communication patterns, etc.
  - Identification of Late Sender / Late Receiver problems

> Performance optimization is an iterative process:
> → Measure, understand, optimize, repeat