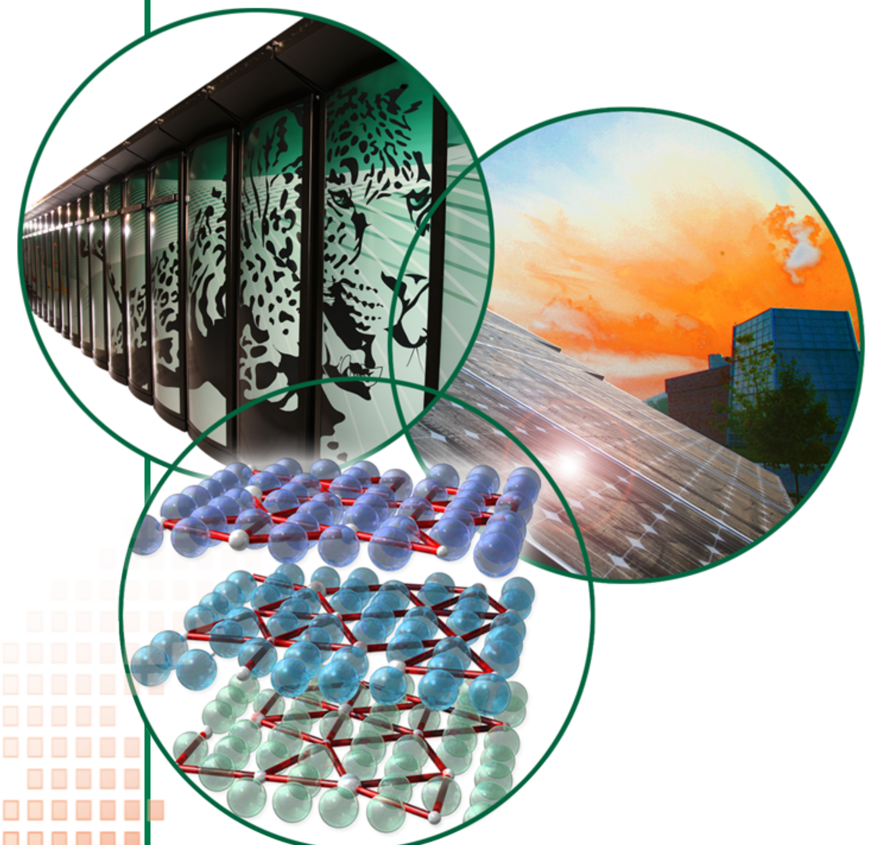
Titan Workshop - Vampir and VampirTrace

Trace Based Performance Analysis at Large Scale on Titan

Jens Domke, JICS, ORNL

January 25, 2012



U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY

MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Disclaimer

Performance tools will not automatically make you code run faster. They help you understand, what your code does and where to put in work.

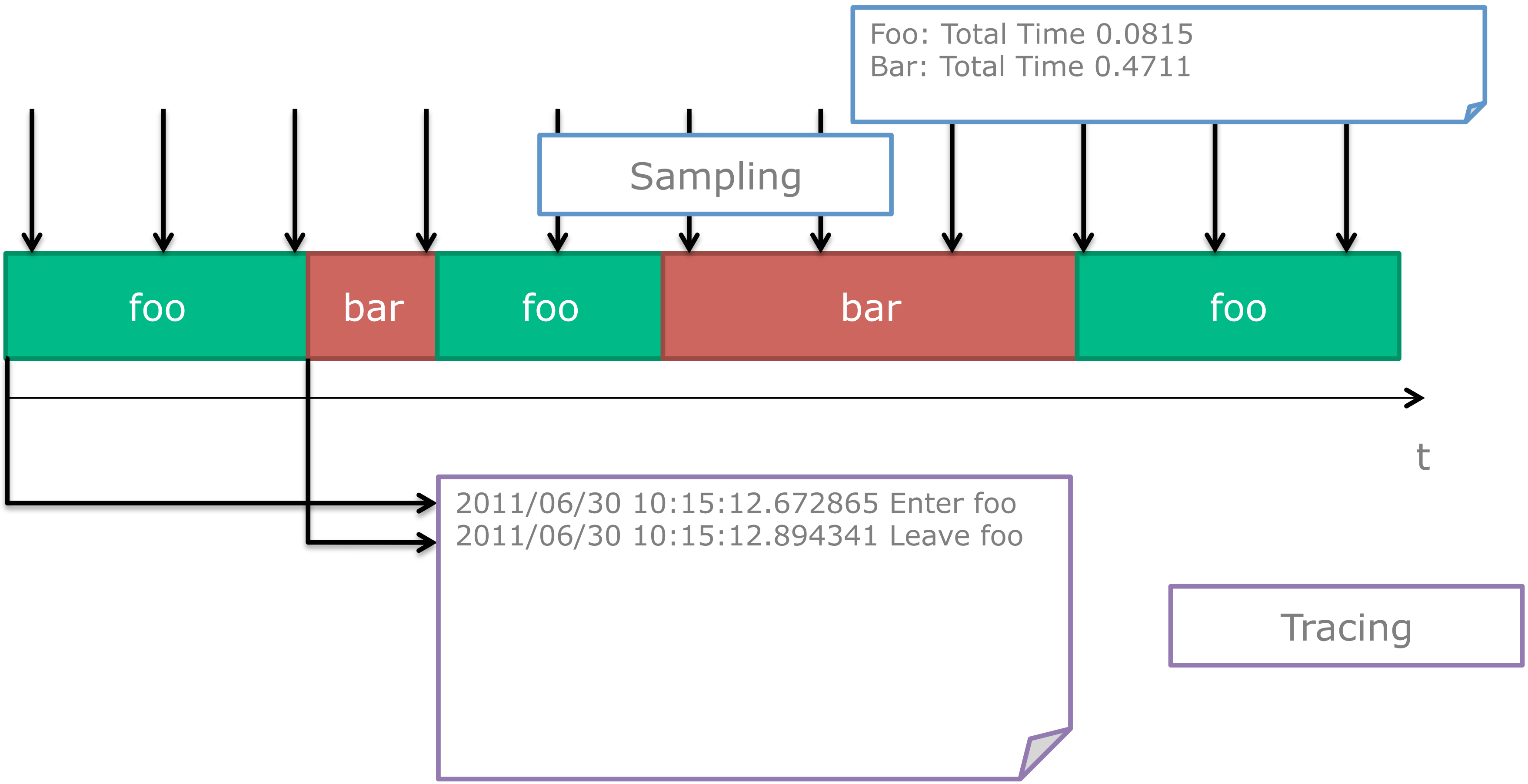
Agenda

1. **Introduction**
   – Sampling vs. Profiling vs. Tracing
2. **The Vampir Framework Workflow**
   – VampirTrace
   – Vampir
3. **Advanced Topics**
   – GPU support & Scalability
4. **Vampir Framework on Titan**
5. **Summary**
6. **Bonus Material**

1. Introduction

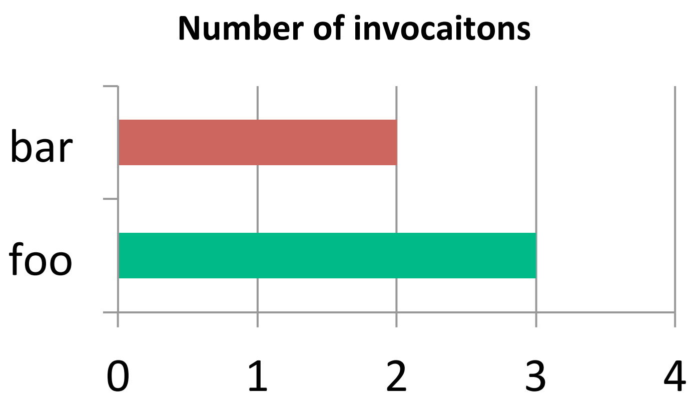
- **Why bother with performance analysis?**
  - Efficient usage of expensive and limited resources
  - Scalability to achieve next bigger simulation
- **Profiling and Tracing**
  - Have an optimization phase
    - just like testing and debugging phase
  - Use tools!
  - Avoid *do-it-yourself-with-printf* solutions, really!

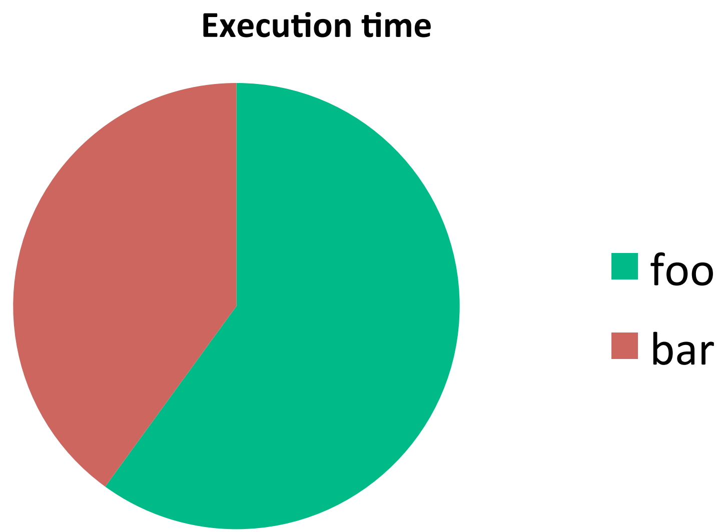
1.1 Sampling vs. Tracing



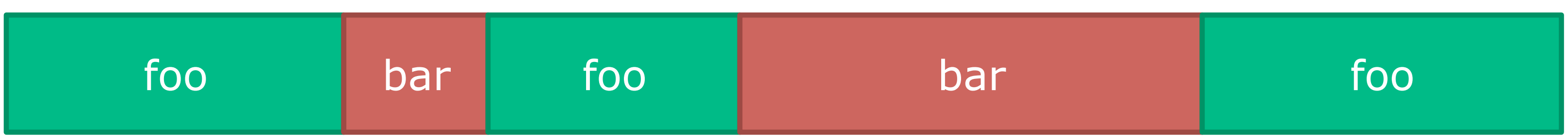
1.2 Profiling vs. Tracing

- Statistics





- Timelines



Agenda

1. Introduction
   – Sampling vs. Profiling vs. Tracing
2. **The Vampir Framework Workflow**
   – VampirTrace
   – Vampir
3. Advanced Topics
   – GPU support & Scalability
4. Vampir Framework on Titan
5. Summary

2.1.1 Instrumentation

What does VampirTrace do in the background?

- **Instrumentation phase (pre-run):**
  - Via compiler wrappers
  - By underlying compiler with specific options
  - MPI instrumentation with replacement lib
  - OpenMP instrumentation with Opari
  - Also binary instrumentation with Dyninst
  - Partial manual instrumentation
  - By an automatic instrumentor (TAU)

2.1.2 Measurement

What does VampirTrace do in the background?

- **Measurement phase (during run):**
  - Event data collection
  - Precise time measurement
  - Parallel timer synchronization
  - Collecting parallel process/thread traces
  - Collecting performance counters (from PAPI, memory usage, POSIX I/O calls and fork/system/exec calls, CUDA, and more …)
  - Monitor accelerator (like GPU, Cell SPE) usage
  - Filtering and grouping of function calls

2.1.3 Common Event Types

- **Enter/leave of function/routine/region**
  - time, process/thread, function ID
- **Send/receive of P2P message (MPI)**
  - time, sender, receiver, length, tag, comm.
- **Collective communication (MPI)**
  - time, process, root, communicator, # bytes
- **Hardware performance counter values**
  - time, process, counter ID, value
- …

2.1.4 Tracing – Pros and Cons

- **Tracing Advantages**
  - Preserve temporal and spatial relationships of events
  - Allow reconstruction of dynamic behavior on any required abstraction level
  - Profiles can be calculated from trace

- **Tracing Disadvantages**
  - Traces can become very large
  - May cause perturbation
  - Instrumentation and tracing is complex
    (event buffering, clock synchronization, …)

2.1.5 Application Adaptations

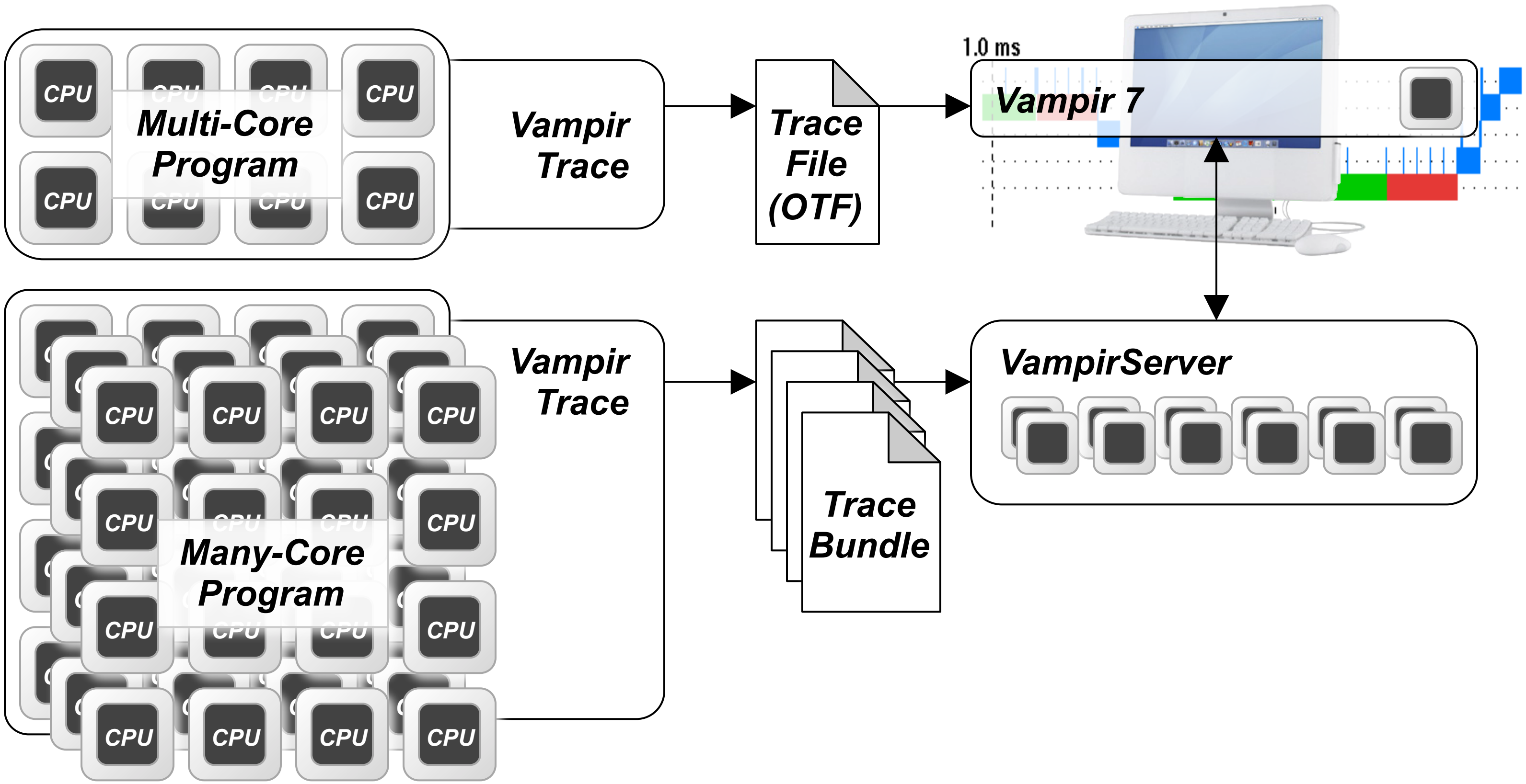
What do you need to do for it?

1. Change the compiler

CC=icc CXX=icpc F90=ifc MPICC=mpicc	CC=vtcc CXX=vtcxx F90=vtf90 MPICC=vtcc -vt:cc mpicc

2. Re-compile & re-link
3. Trace Run (run the application with an appropriate test data set)

2.2 Workflow Overview



2.3.1 Event Trace Visualization

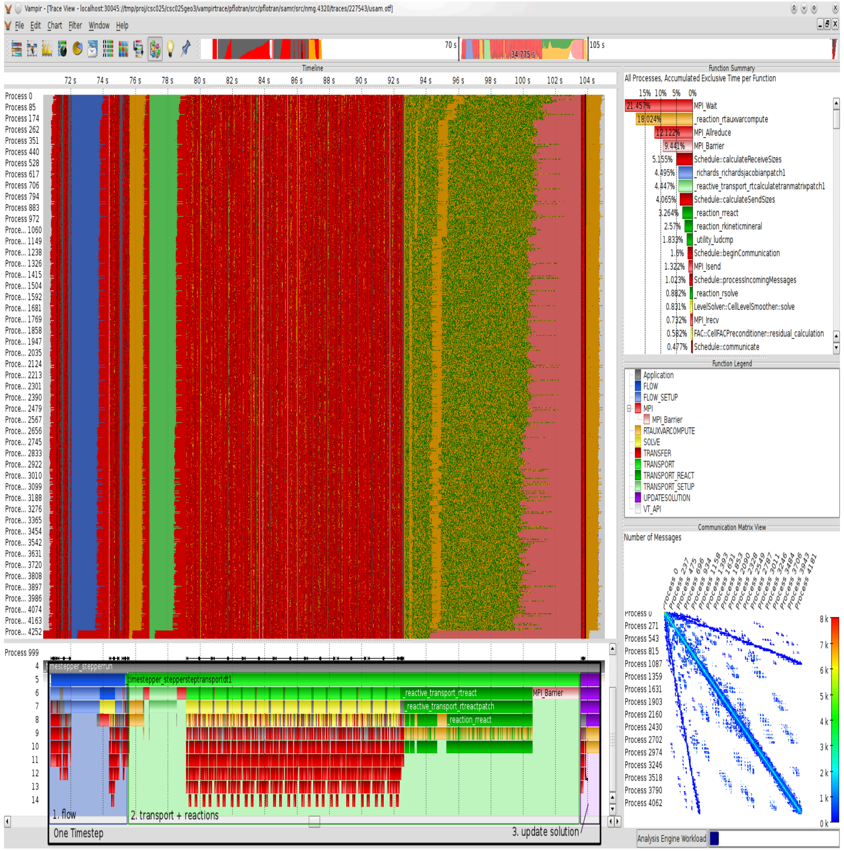
- **Trace Visualization**
  - Alternative and supplement to automatic analysis
  - Show dynamic run-time behavior graphically
  - Provide statistics and performance metrics
    - Master timeline for parallel processes/threads
    - Process timeline plus performance counters
    - Statistics summary display
    - Message statistics
    - and more
  - Interactive browsing, zooming, selecting
    - Adapt statistics to zoom level (time interval)
    - Also for very large and highly parallel traces

2.3.2 Finding Performance Bottlenecks

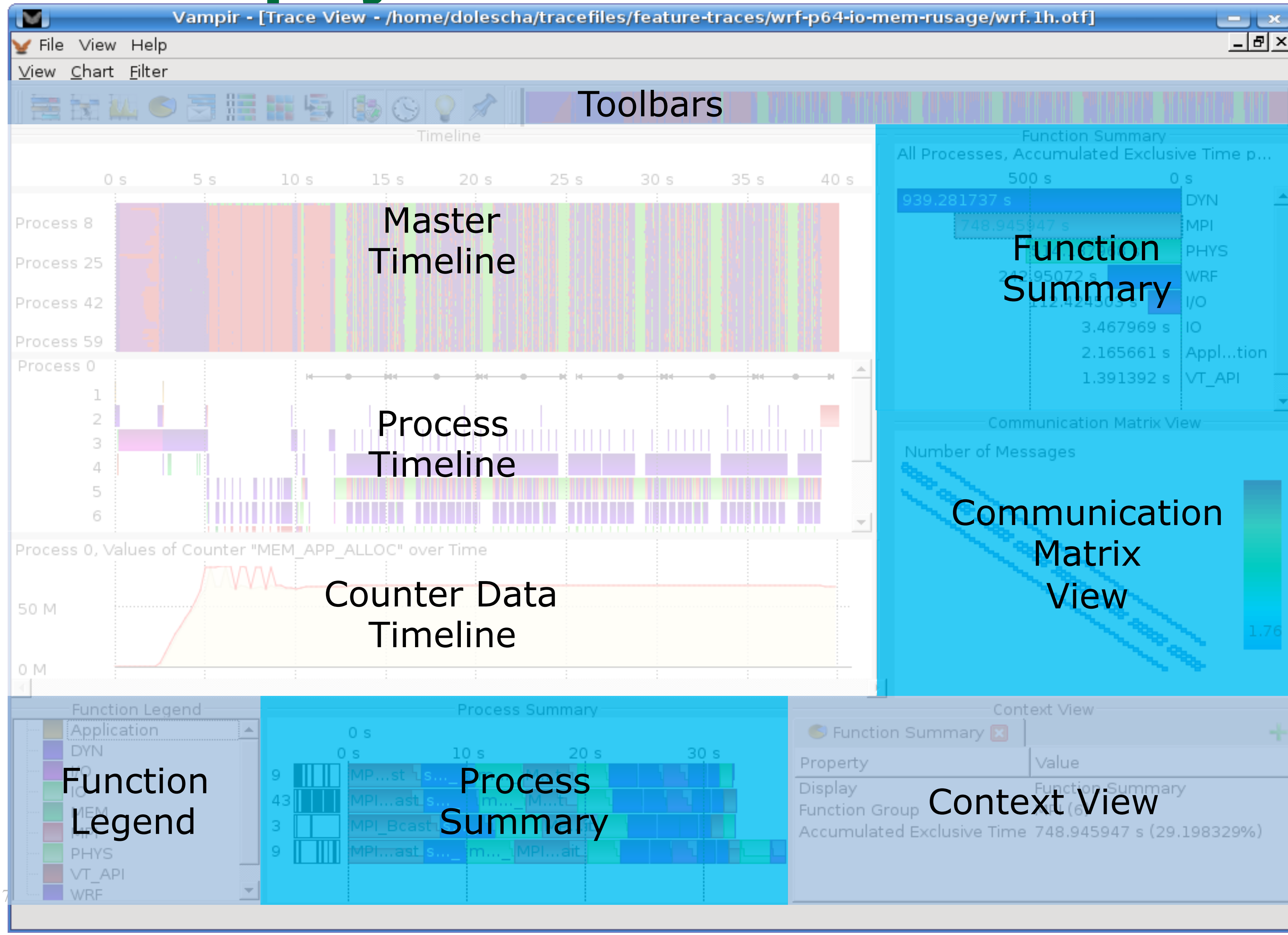
- Inefficient communication patterns
- Load imbalance / serial parts of the application
- Memory bound computation
  - Inefficient cache usage
  - TLB misses
  - Use HW counters (PAPI) to detect
- I/O bottlenecks
- Most time consuming functions
- …

2.3.3 VampirClient (Vampir 7)

- GUI to analyze trace files (OTF)
- Main concept: Timeline + statistics
- GUI is QT based – available on Linux, Mac, Windows



2.3.4 Displays for a trace with 64 cores



2.3.5 VampirServer

- Parallel analysis engine for Vampir
  - MPI parallelized for distributed memory systems
  - Pthread parallelized for shared memory systems
- Scales to > 10,000 analysis processes
- Loads the entire uncompressed trace into memory

Agenda

1. Introduction
   - Sampling vs. Profiling vs. Tracing
2. The Vampir Framework Workflow
   - VampirTrace
   - Vampir
3. **Advanced Topics**
   - GPU support & Scalability
4. Vampir Framework on Titan
5. Summary

3.1 Challenges for VT on Titan

Scalability

- Overcome I/O problems
- Find ways to show that much data

GPU Support

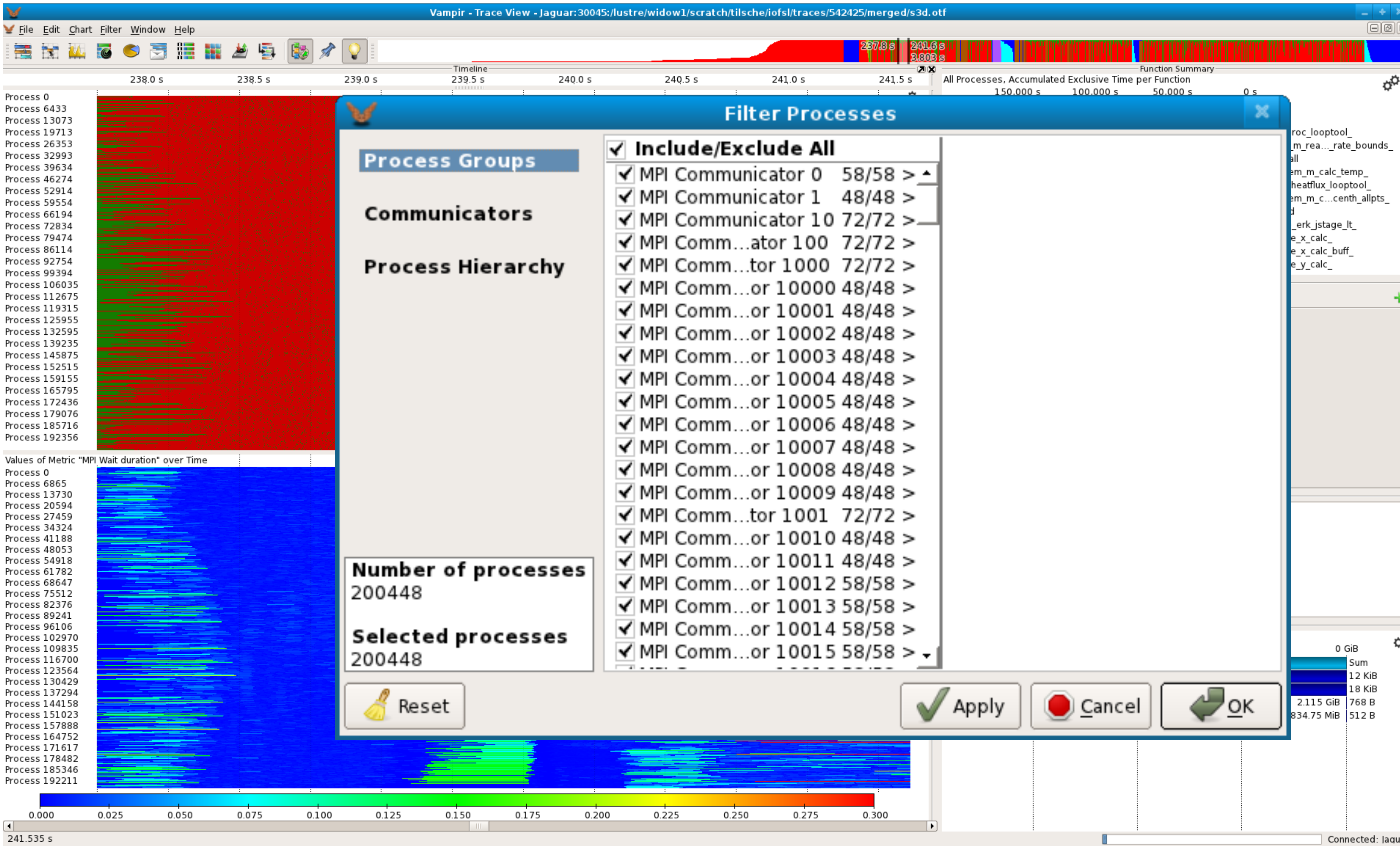
- Hitting a moving target
- Another layer of heterogeneity to display

3.1.1 Limits for Tracing on Titan

- **Current limit** is (based on user experiences):
  - Tracing up to 8,000 processes
  - I/O problem (too many file creates – one per process)

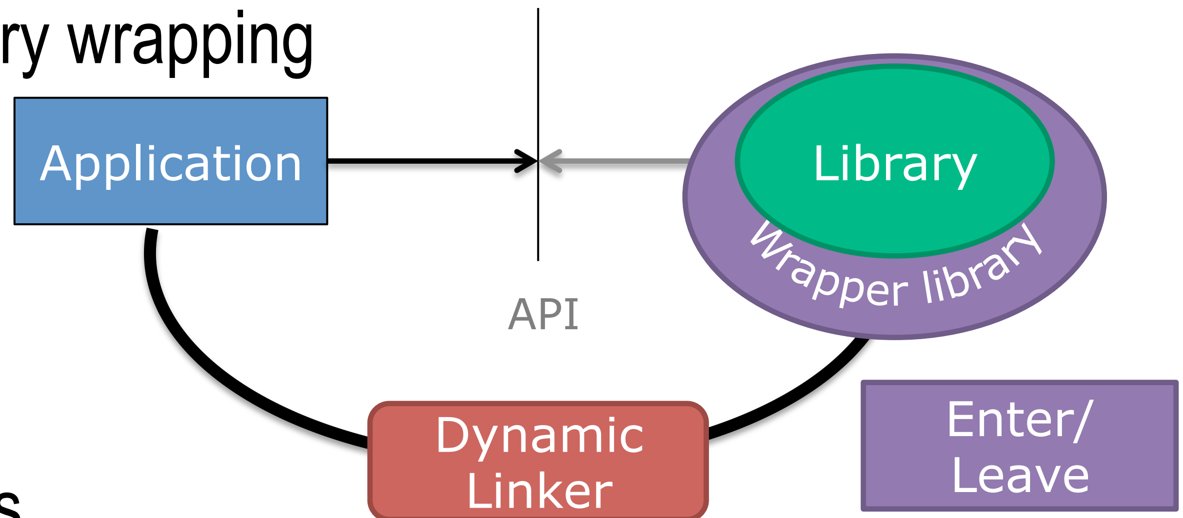
- **Prototype** was already working on Jaguar
  - Tracing 200,000+ processes
  - Based on IOFSL library
  - Opened for visualisation with 20,000 VampirServer processes

3.1.2 200,000+ Processes in Vampir



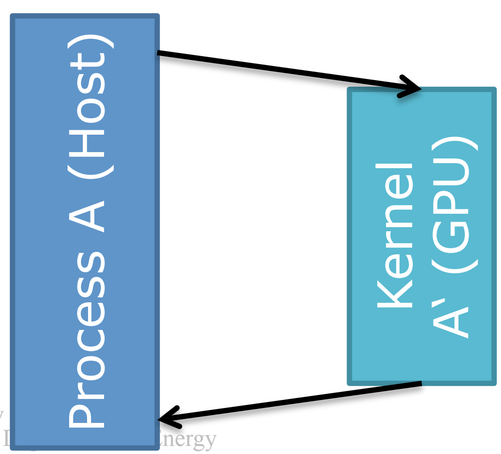
3.1.3 GPU Support: CUDA (& OpenCL)

- Currently done via library wrapping

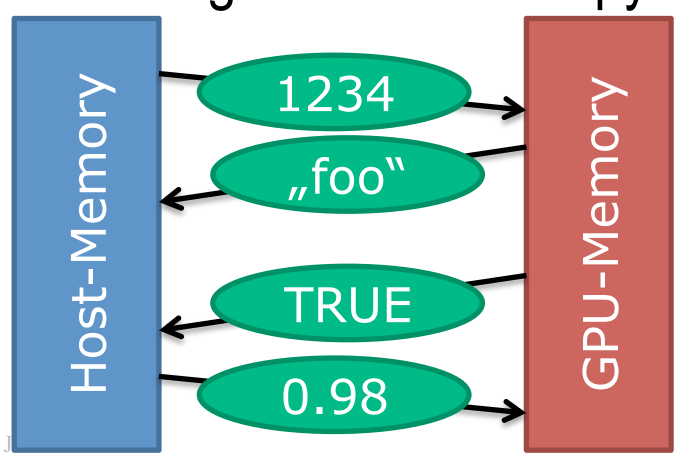


- Reuse of known metrics

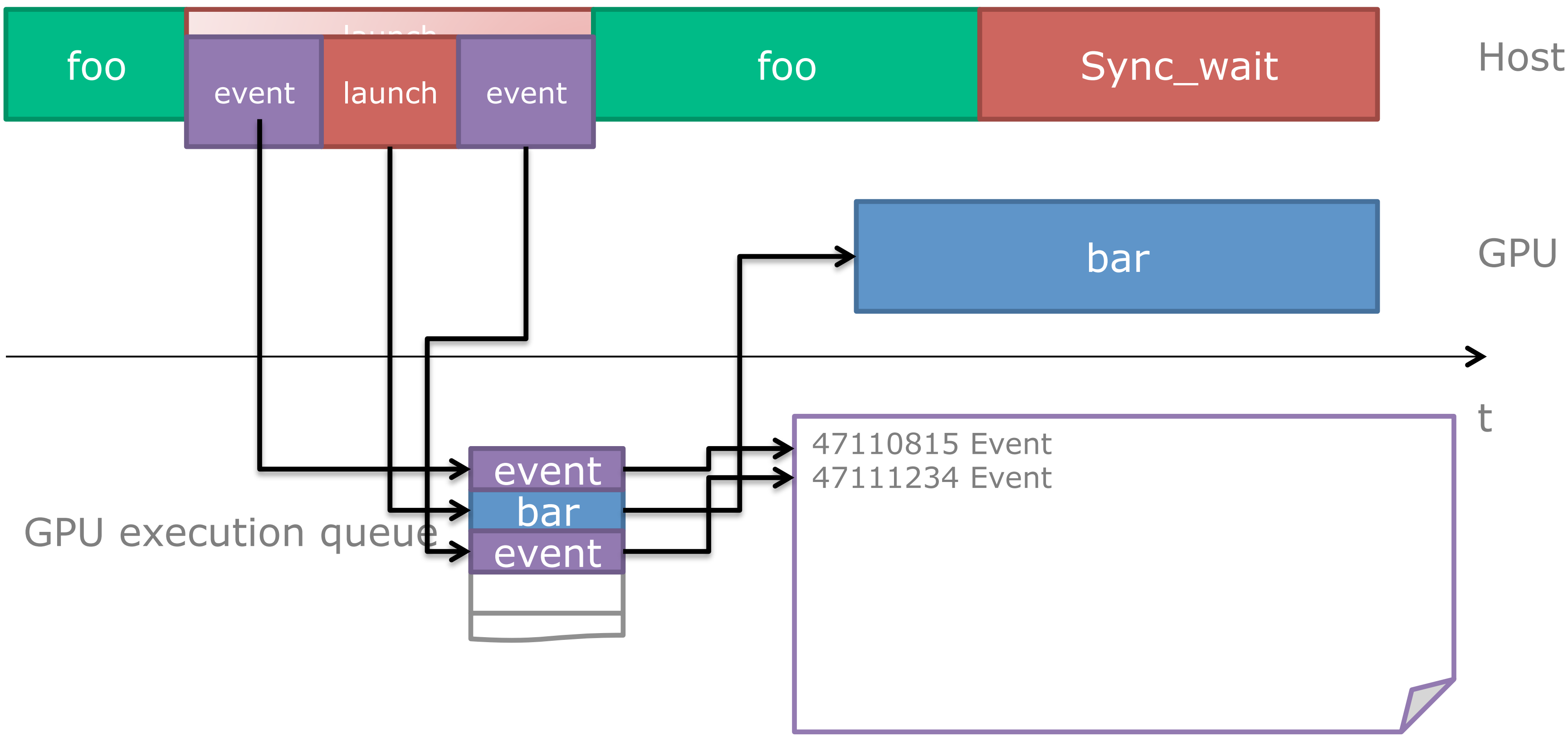
Thread = Kernel



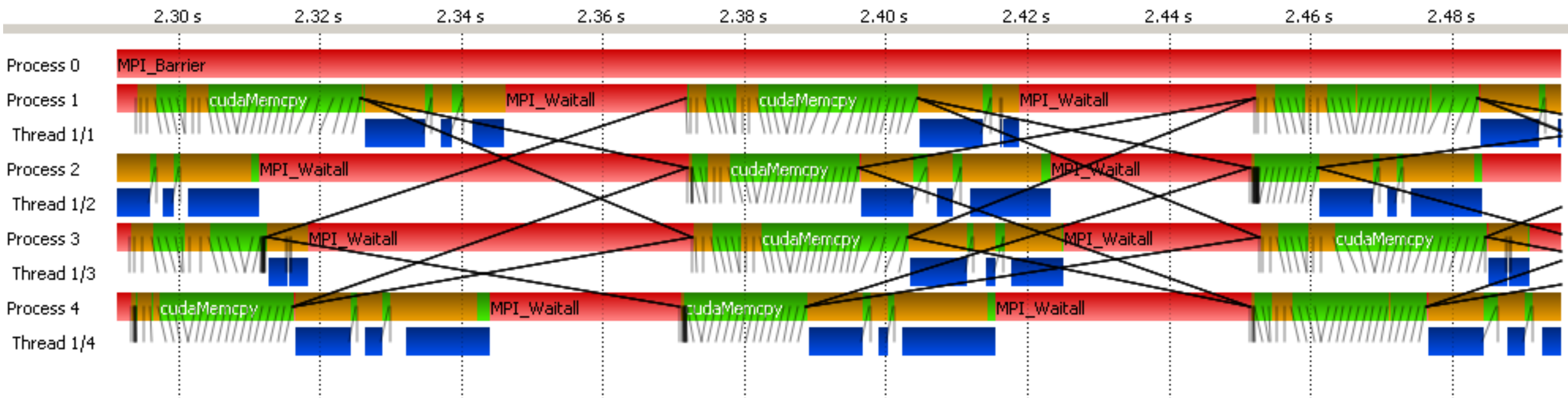
Message = cudaMemCpy



3.1.4 Time stamps for asynchronous events



3.1.5 Visualization of GPU Tracing



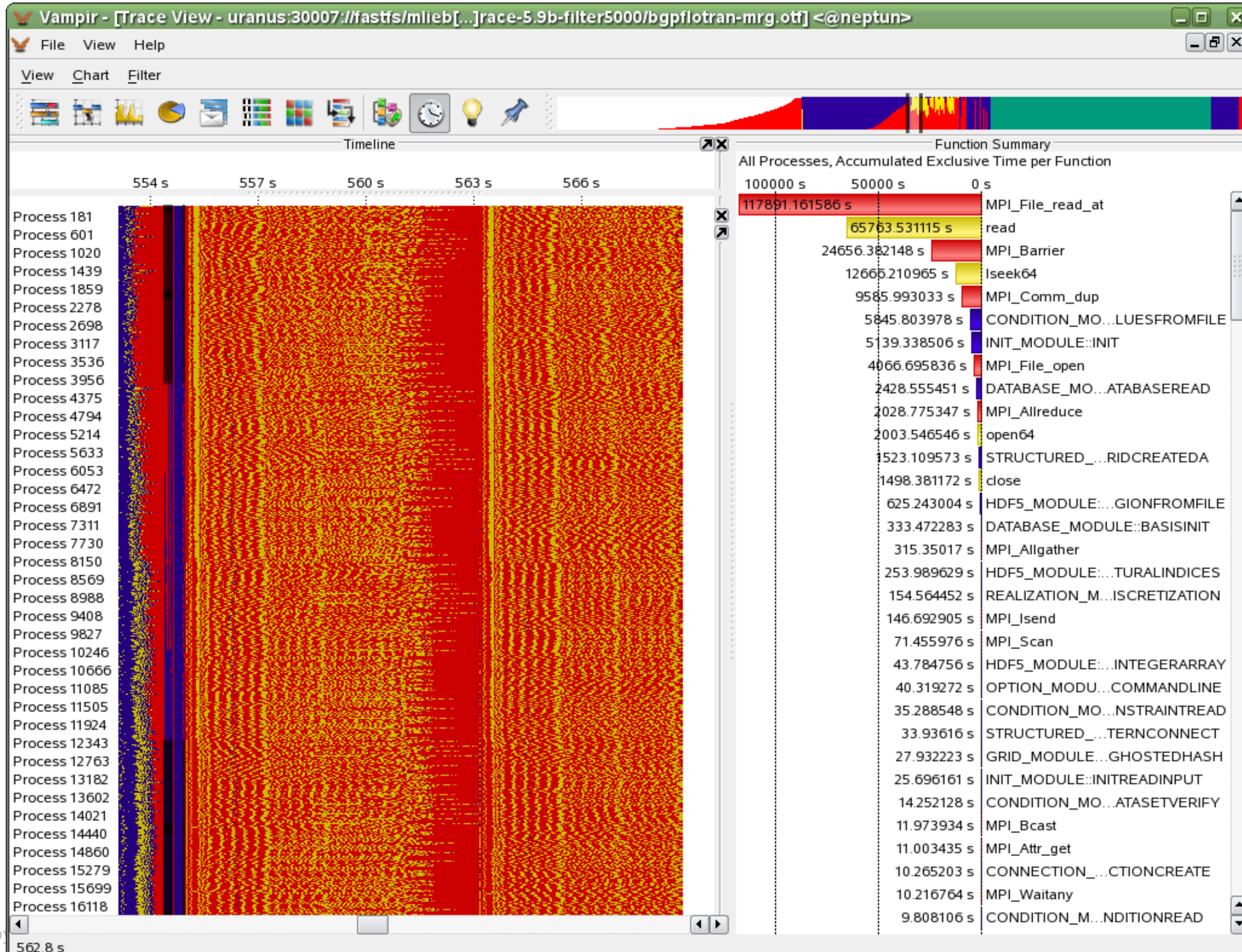
- Support for hybrid modes: GPGPU + MPI + threads
  - Function invocations in host processes (Process X) and threads
  - Kernel invocations in CUDA threads (Thread x/y)
  - Host-GPU interactions via CUDA API (light arrows)
  - Host-Host interactions via MPI (bold arrows)

*) old picture; CUDA Threads were renamed (CUDA[x] y:z)

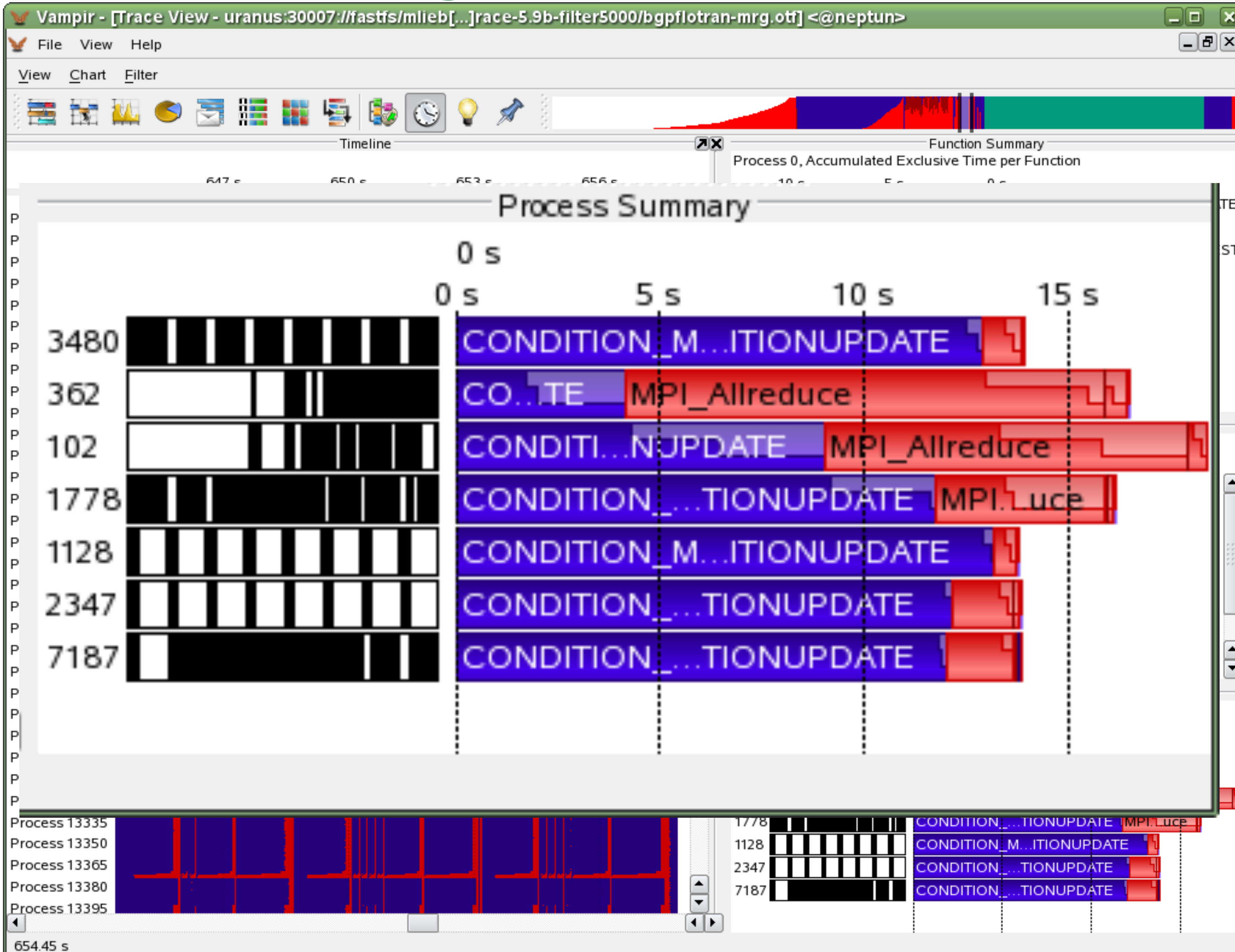
3.2 Scalability Features in Vampir

- Fit to chart height feature of master timeline and performance radar
  - Allows visualization of more processes than pixels of screen are available
- Clustering
  - Allows detection of outlier processes and groups with similar behavior
- Performance radar
  - Highlighting performance conditions of your program in a global timeline

3.2.1 Fit to chart height: Pflotran initialization + I/O

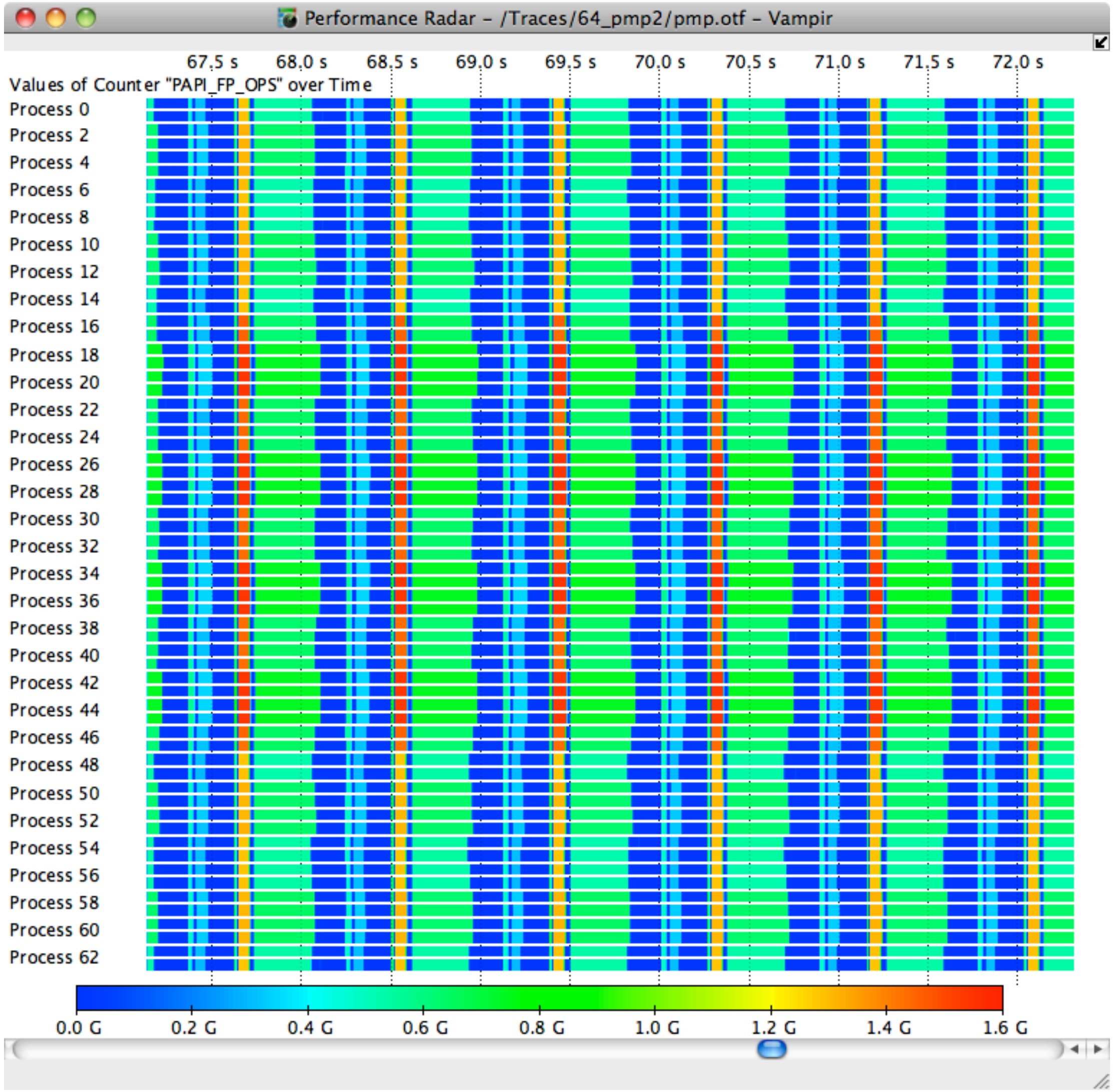


3.2.2 Clustering: Pflotran - first iteration



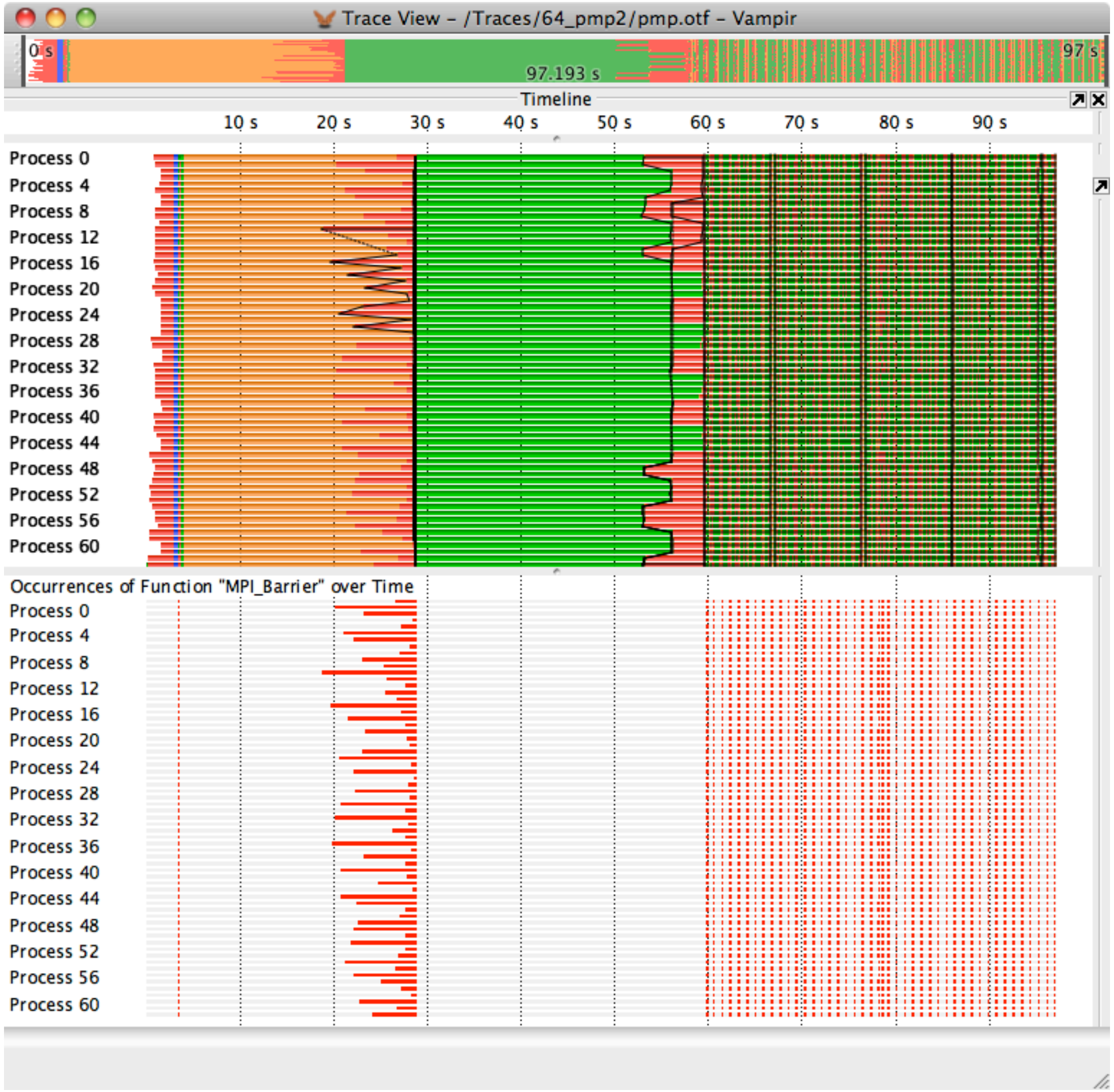
3.2.3 Performance Radar

- Display objectives:
  - Identification of performance relevant trace parts
  - Assistance to users to navigate in trace data and to spot interesting sections
  - Performance of basic arithmetics on counter data



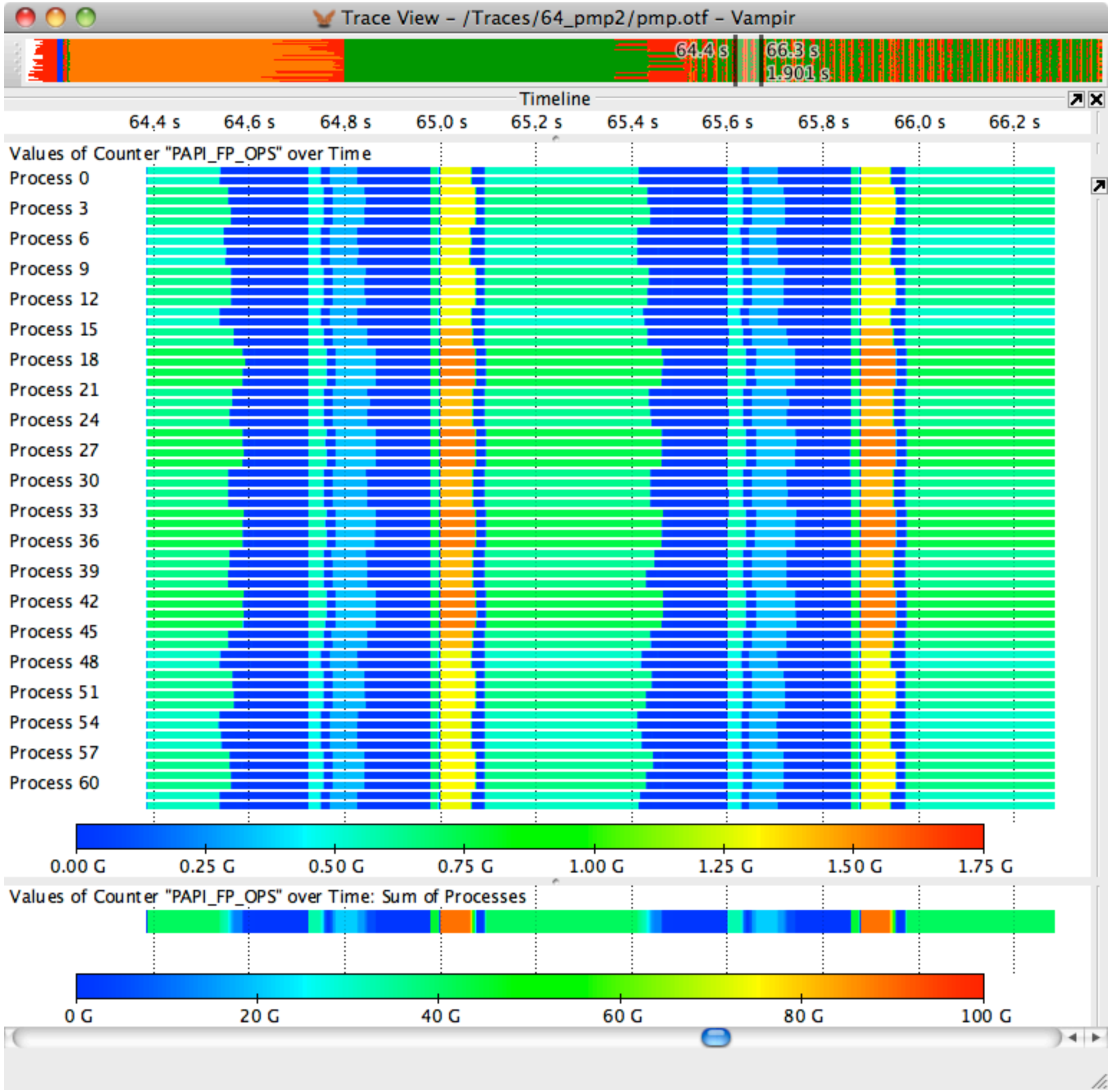
3.2.3 Performance Radar

- Features:
  - Detection of occurrences of functions/function groups
  - Visualization of call density of functions/ function groups to help to find performance relevant candidates
  - Construction of filter based on function occurrences over time for further usage in calculations



3.2.3 Performance Radar

- Features:
  - Performance of arbitrary calculations on counter data, e.g. add up all floating point operations over time, differentiation of performance counter
  - Support for concatenation of several calculations
  - Utilization of filter in calculations, e.g. only add up FLOPS of function x



Agenda

1. Introduction
   – Sampling vs. Profiling vs. Tracing
2. The Vampir Framework Workflow
   – VampirTrace
   – Vampir
3. Advanced Topics
   – GPU support & Scalability
4. **Vampir Framework on Titan**
5. Summary

4.1 VampirTrace

titan$ module load vampirtrace

- Use the appropriate compiler wrappers
  - vtcc, vtCC, vtf77, vtf90
- Pick appropriate library (seq, mpi, mt, hyb)
  - e.g. vtcc –vt:hyb (recommended)
- Pick instrumentation type
  - –vt:inst compinst (**default**, compiler instrumentation)
  - –vt:inst manual (MPI, OpenMP, CUDA and manual inst.)
- Tell your build system to use VampirTrace
  - ./configure –with–CC="vtcc –vt:hyb" …

4.1.1 VampirTrace

- Pitfalls & Hints (especially for Titan)
  - Make sure that the VampirTrace module is loaded
    - Either ‘qsub -V’ or ‘module load vampirtrace’ within the PBS script
  - Save traces to special directory to optimize I/O performance
    - Set VT_PFORM_GDIR=traces
    - lfs setstripe -c 1 $VT_PFORM_GDIR
  - Set VT flag ‘-vt:hyb’ (VT does not identify the Cray wrapper as MPI/OMP compiler)
  - Use ‘-vt:verbose’ to see the commands executed by VT
  - Run with VT_MODE=STAT to create a profile (instead of a trace) for the first overview of you program
  - Set VT_BINDIR=/tmp/work/$USER/.vt/bin if you get something like: ‘/sw/…/vtunify No such file or directory’

4.1.2 VampirTrace

- More Pitfalls & Hints
  - Increase VT_BUFFER_SIZE if necessary
  - Avoid VT_MAX_FLUSHES=0 when you don’t know how large your trace will get (you could write TBytes of data)
  - Some compiler instrument inline functions, some do not
  - Avoid ‘-vt:inst compinst’ (default) for large C++ codes with STL
    - Use TAU (-vt:inst tau) or Dyninst (-vt:inst dyninst) instead
  - Use VT_FILTER_SPEC to limit the number of recorded events
    - Reduces resulting trace size by filtering frequently called functions
  - Function filtering won’t reduce runtime dramatically
    - -finstrument-functions-exclude-function-list=… works for a small number
    - ‘-vt:inst tau’ with a include/exclude file is recommended for large projects

4.2 Vampir

titan$ module load vampir

titan$ vampirserver start –a <PROJ> –n <cores>

titan$ *wait until server is running and follow the instructions*

local1$ ssh –L 30XXX:nidYYYY:30ZZZ <user>@titan

local2$ ./vampir *and choose 'File –> Remote Open'*

titan$ vampirserver stop XXXXX

- Pitfalls & Hints:
  - Use enough cores to load the trace in memory (decompressed, 4x)
  - Avoid running Vampir on the login nodes (X11 is slow)

Agenda

1. Introduction
   – Sampling vs. Profiling vs. Tracing
2. The Vampir Framework Workflow
   – VampirTrace
   – Vampir
3. Advanced Topics
   – GPU support & Scalability
4. Vampir Framework on Titan
5. **Summary**

5. Summary

- **VampirTrace**
  – Convenient instrumentation and measurement infrastructure
  – Hides away complicated details
  – Provides many options and switches for experts
  – Available as
    - Part of Open MPI 1.3 and higher
    - Stand-alone package: www.tu-dresden.de/zih/vampirtrace
  – Further information
    - www.nccs.gov/computing-resources/lens/software/?software=vampirtrace
    - www.tu-dresden.de/zih/vampirtrace
    - chester$ /ccs/proj/trn001/vampir/manual/vampirtrace.pdf

5. Summary

- **VampirClient & VampirServer**
  - Interactive trace visualization and analysis
  - Intuitive browsing and zooming
  - Scalable to “quite large” trace data sizes (1.5 TByte)
  - Scalable to high parallelism (200,000 processes)
  - VampirClient is available for Windows, Linux/Unix and Mac OS X
  - Further information
    - www.nccs.gov/computing-resources/lens/software/?software=vampir
    - www.vampir.eu
    - chester$ /ccs/proj/trn001/vampir/manual/vampirtrace.pdf

Acknowledgements

This work would have been impossible without the dedication of:

- Matthias Lieber (Tracing & Analysis)
- Matthias Jurenz (VampirTrace Software & Support)
- Matthias Weber (Vampir Software & Support)

- The Vampir Team:

  Matthias Jurenz, Andreas Knüpfer, Ronny Brendel, Matthias Lieber, Jens Doleschal, Jens Domke, Holger Mickler, Daniel Hackenberg, Michael Heyde, Thomas Ilsche, Guido Juckeland, Dietrich Robert, Johannes Spazier, Michael Kluge, Matthias Müller, Holger Brunst, Ronald Geisler, Reinhard Neumann, Heide Rohling, Rene Widera, Thomas Ilsche, Matthias Weber, Bert Wesarg, Hartmut Mix, Thomas William, Wolfgang E. Nagel

Thank you!

- Current on-side contact:
  - Jens Domke, JICS, ORNL
    Bldg. 5700, Rm. B206
    E-Mail: domkej@ornl.gov
    Tel: 1-865-241-6293

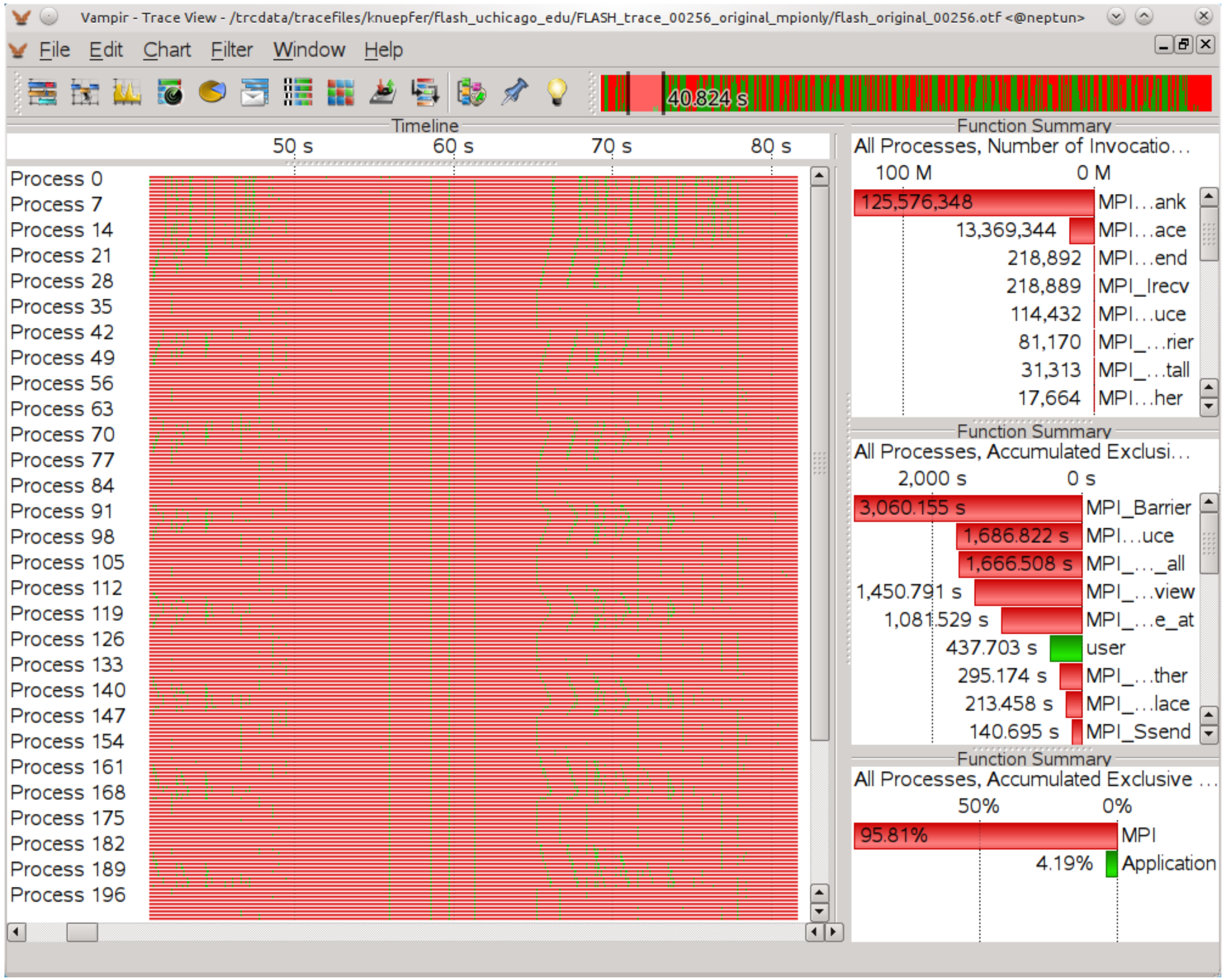
- Vampir & VampirTrace Support:
  - vampirsupport@zih.tu-dresden.de

7. Bonus Material – Bottlenecks

- Trace visualization to identify bottlenecks
  - Several displays with many options
  - Identify essential parts of an application (initialization, main iteration, I/O, finalization)
  - Identify important components of the code (serial computation, MPI P2P, collective MPI, OpenMP)
  - Make a hypothesis about performance problems
  - Consider application's internal workings if known
  - Select the appropriate displays
  - Use statistic displays in conjunction with timelines

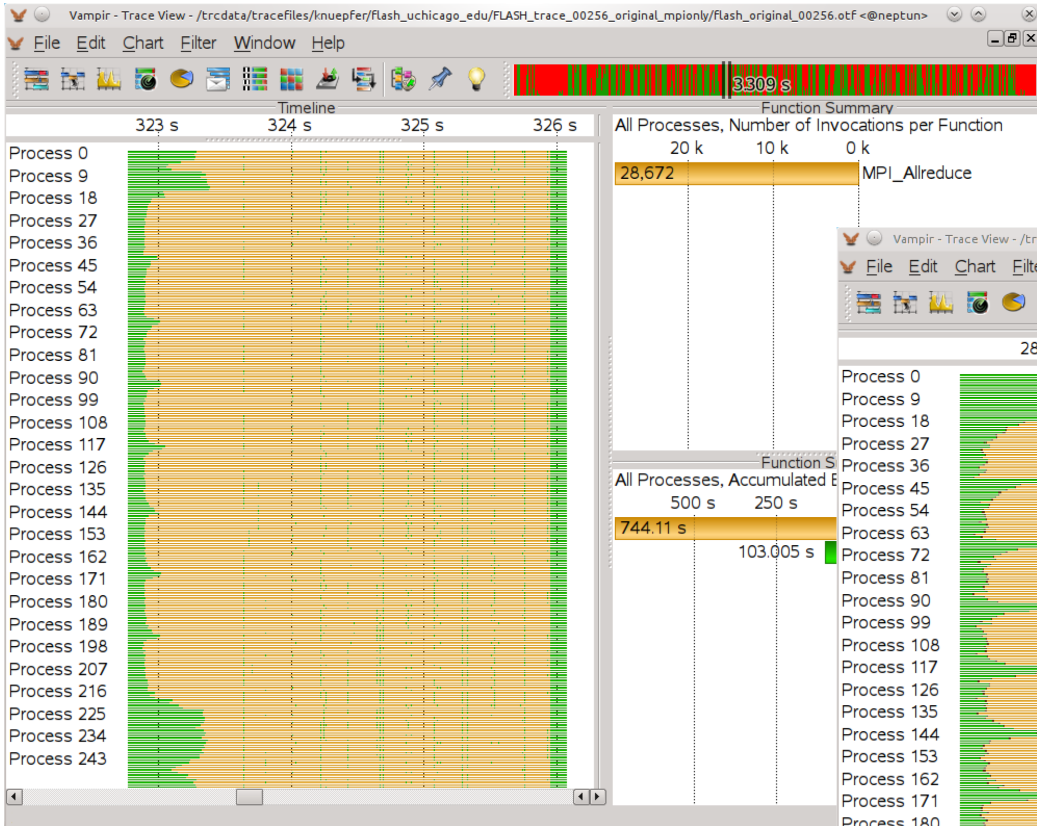
7.1 Communication it-self

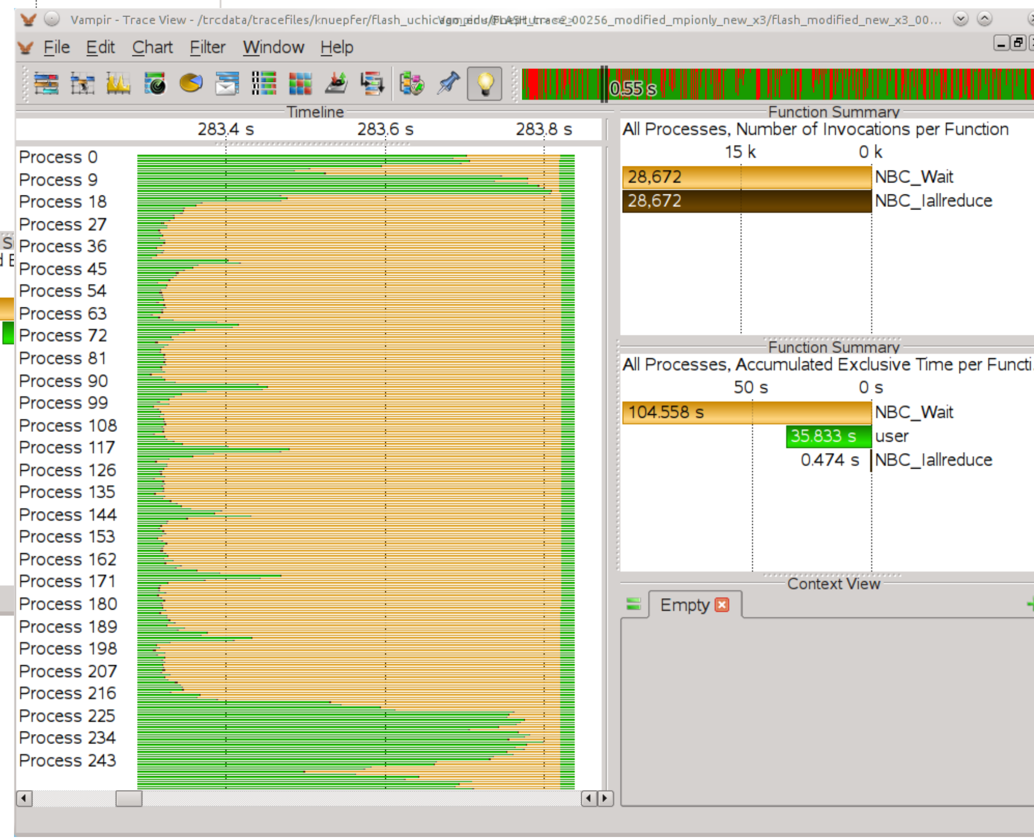
- Too much runtime share for MPI all-together



7.2 Bursts of large messages

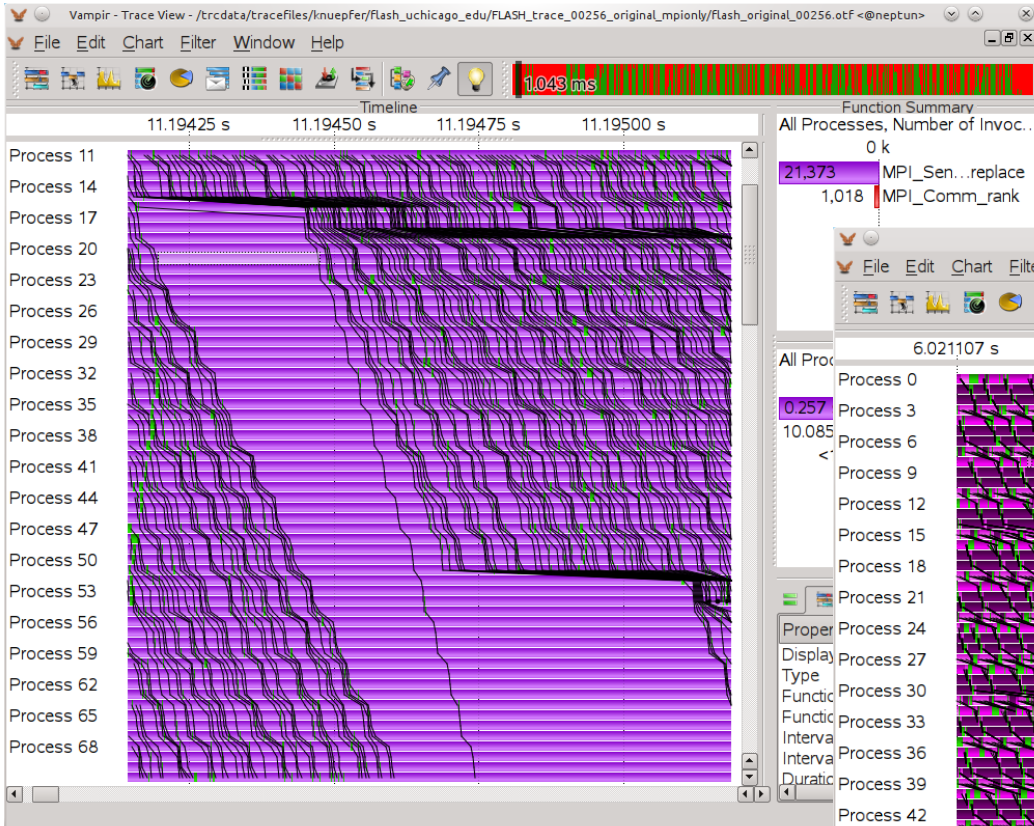
- Long bursts of MPI allreduce operations



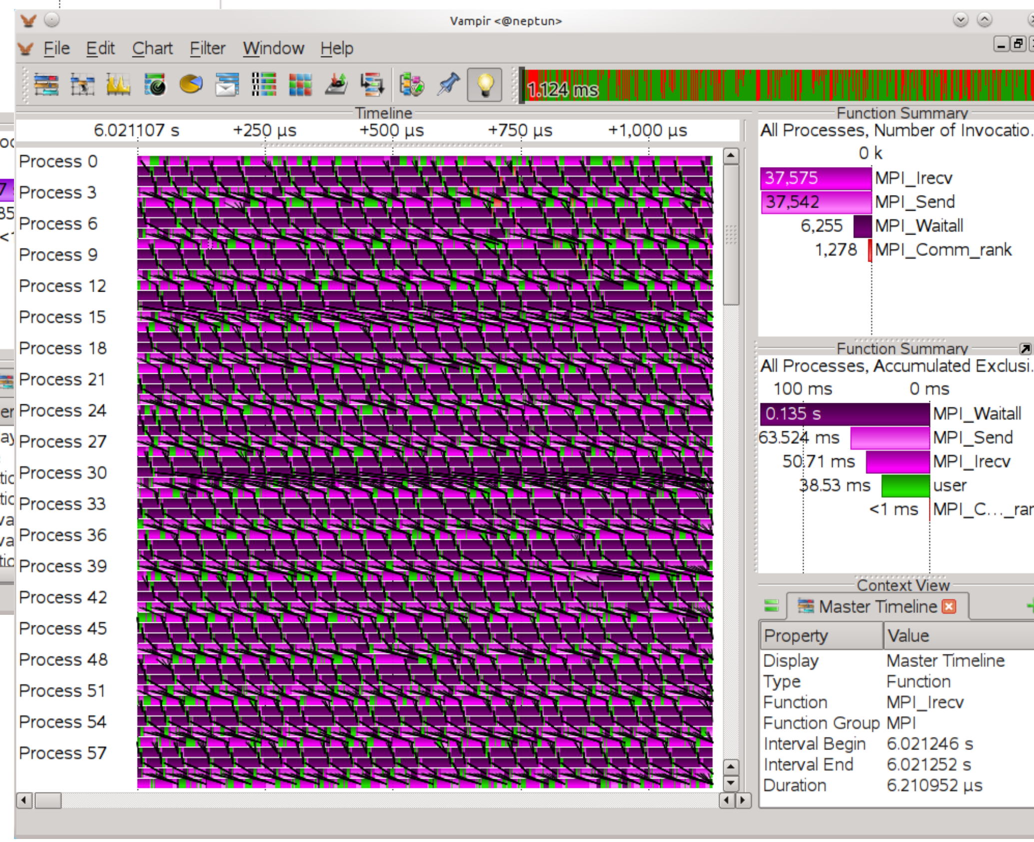


→ Replace by NBC non-blocking Allreduce plus wait

7.3 Sequential point-to-point messages

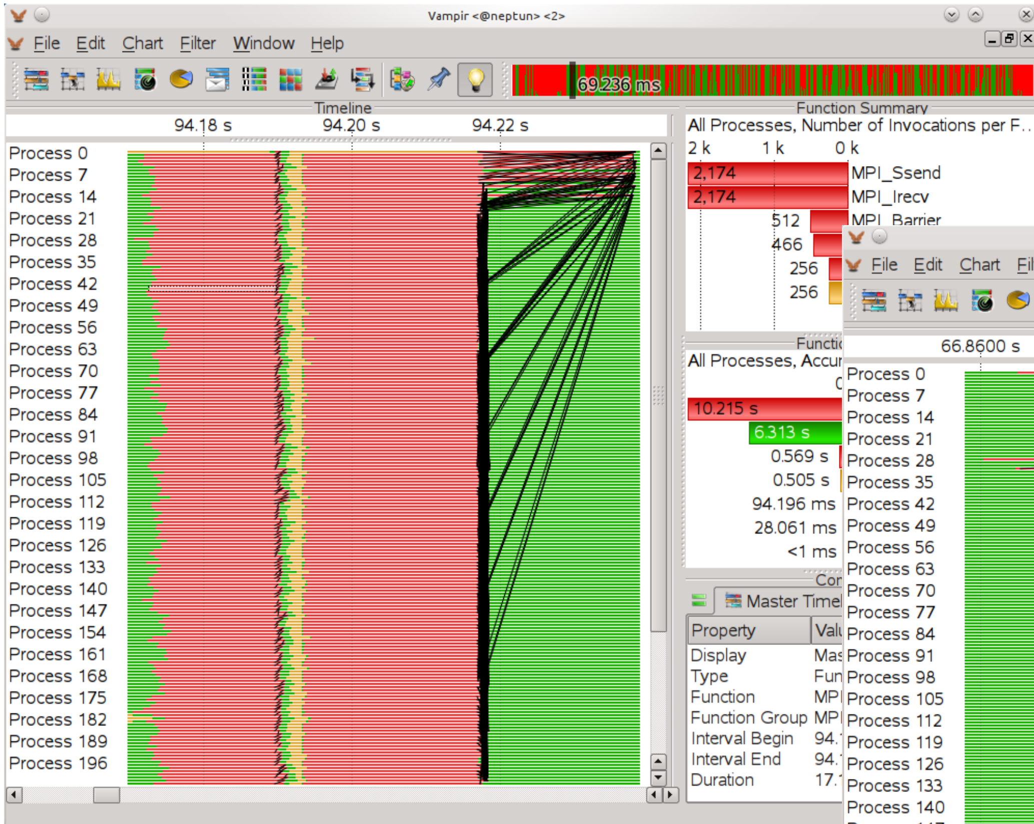


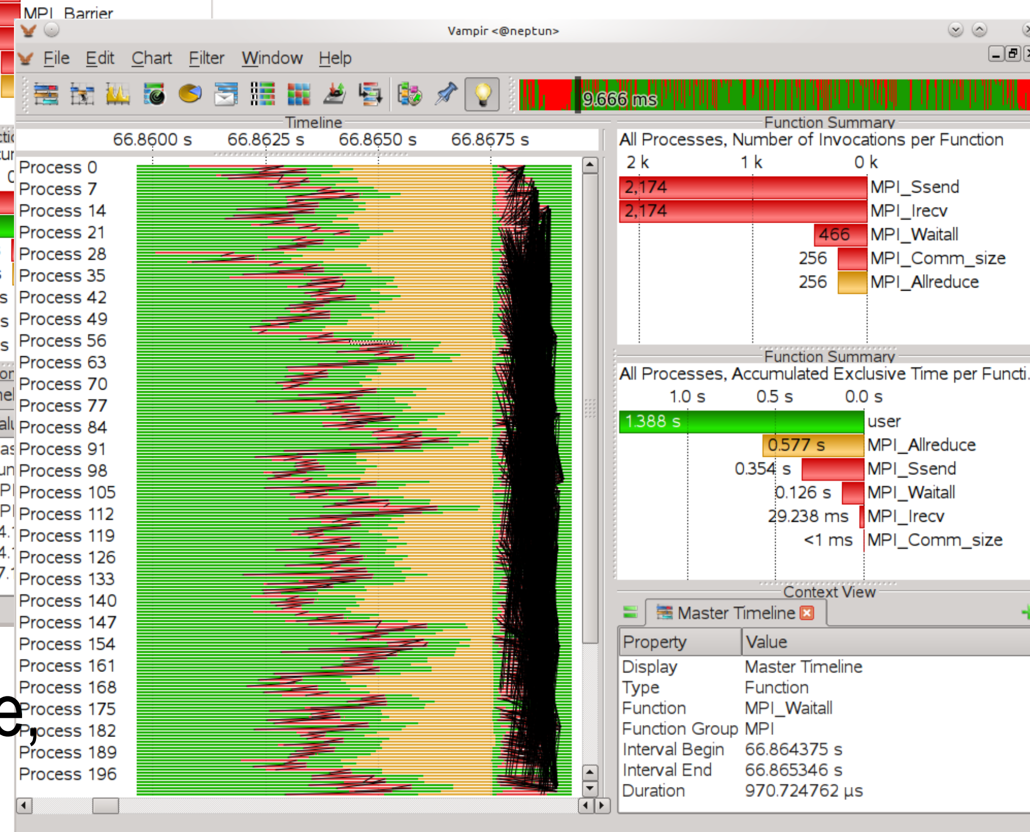
- Chains of MPI Sendrecv



→ Replace by faster over-lapping series of Send, Irecv, and Waitall

7.4 Unnecessary synchronization

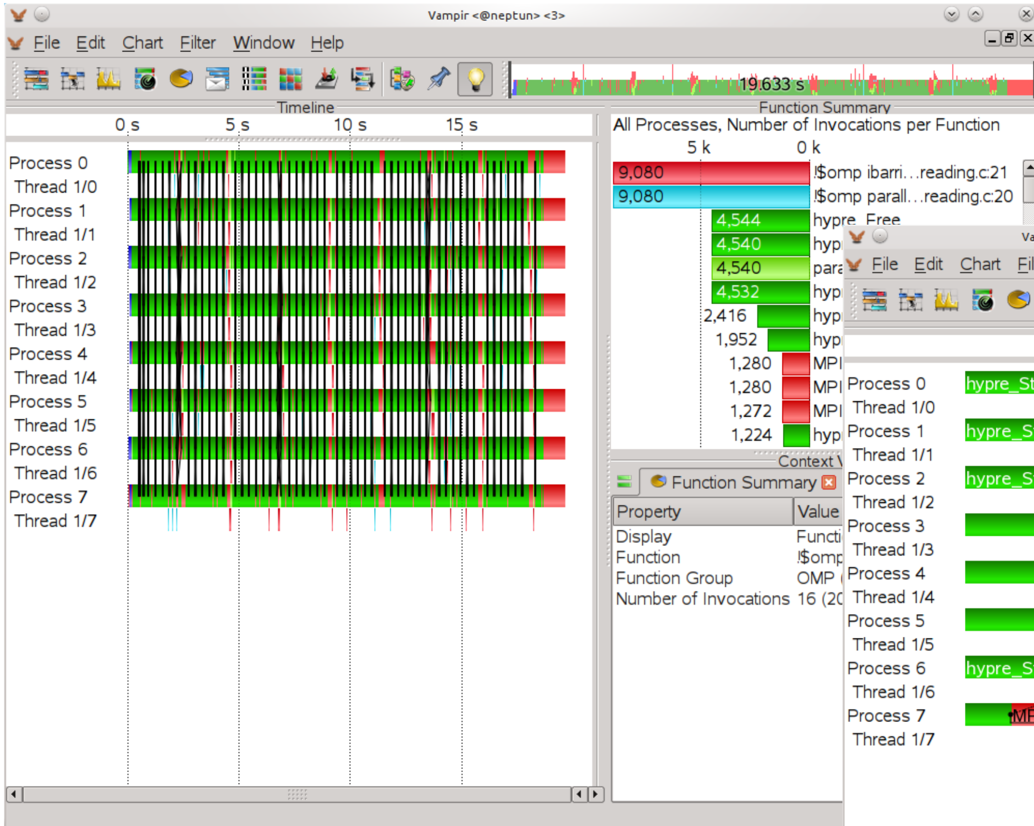




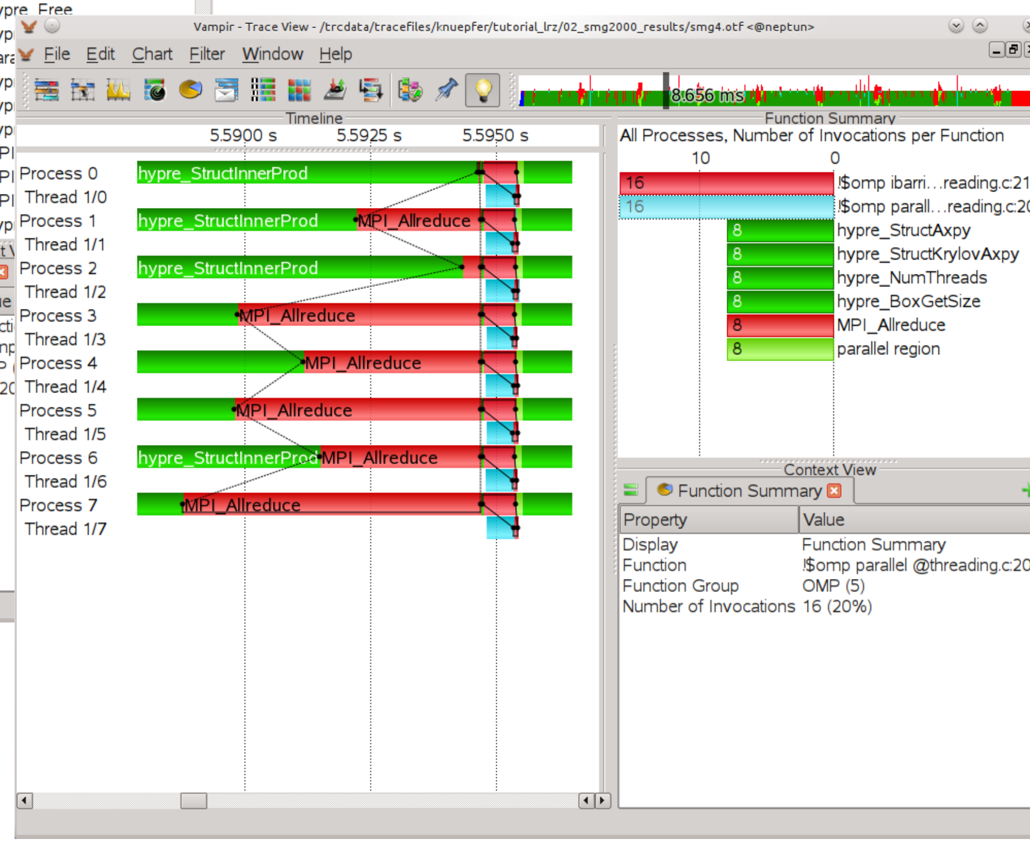
- Many MPI barriers are unneeded

→ Remove barriers if possible, following operations start less regular but are faster.

7.5 Low parallel efficiency

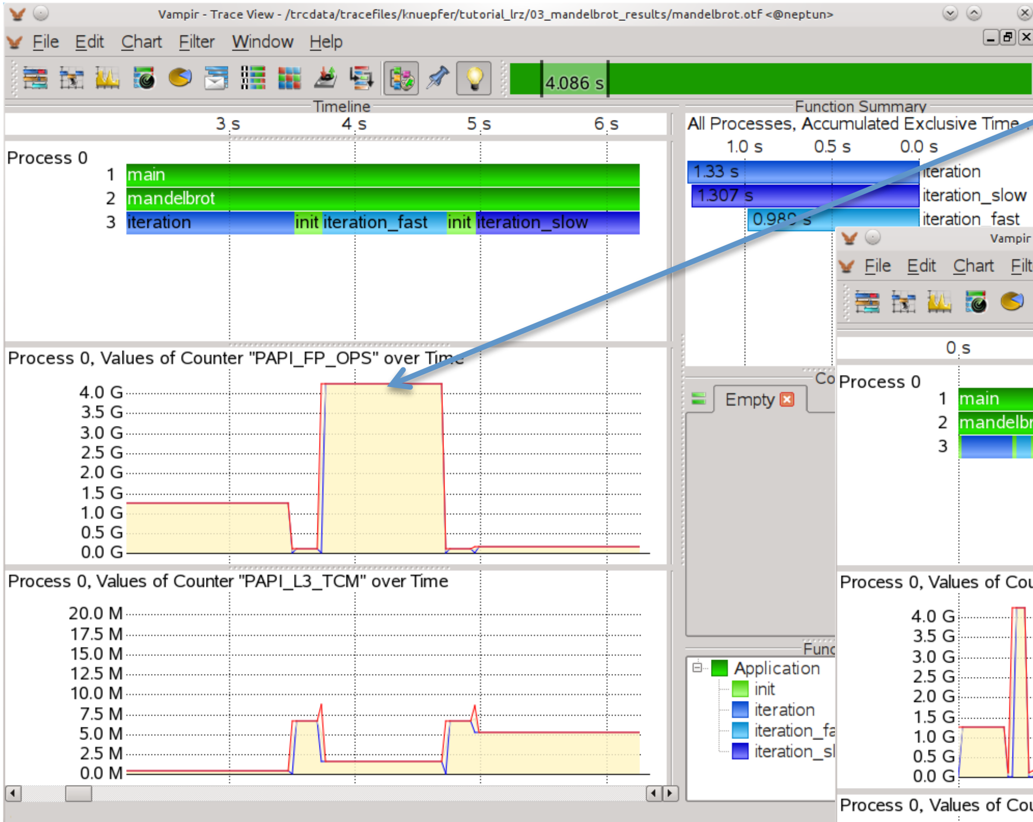


- Mostly idle OpenMP threads

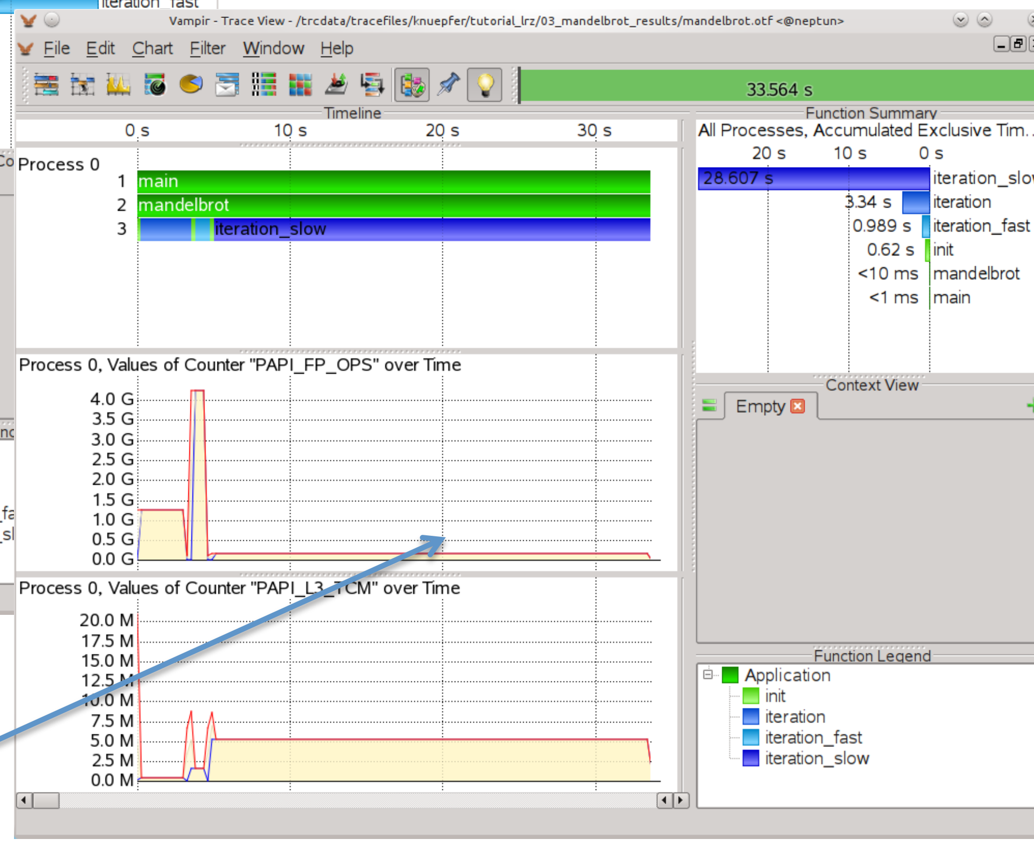


(Zoom)

7.6 Inefficient L1/L2/L3 cache usage



High rate of Flop/s with low rate of L3 cache misses



Low Flop/s rate due to a high L3 miss rate