



703650 VO Parallel Systems WS2019/2020

Debugging Parallel Programs

Philipp Gschwandtner

Overview

- ▶ functional debugging

- ▶ generic guidelines
- ▶ serial debugging
- ▶ parallelism-specific debugging

- ▶ performance debugging

- ▶ generic guidelines
- ▶ serial debugging
- ▶ parallelism-specific debugging

Motivation



https://www.youtube.com/watch?v=gp_D8r-2hwk

Motivation

- ▶ Why do we need debugging?
 - ▶ Because we make mistakes!
- ▶ Why do we need a lecture about this?
 - ▶ OpenMPI FAQ “Debugging applications in parallel”, first question:
Q: “How do I debug OpenMPI processes in parallel?”
A: “This is a difficult question. [...] This FAQ section does not provide any definite solutions to debugging in parallel. [...]”



Functional Debugging



Functional Debugging

- ▶ everything that results in not getting the correct program output
 - ▶ program not finishing (freezes, infinite loops)
 - ▶ program crashes
 - ▶ incorrect output
- ▶ errors can be deterministic or non-deterministic
 - ▶ ensure/maximize reproducibility during testing (e.g. fix random seeds, scheduling affinities, ...)
- ▶ all that applies to debugging serial programs is crucial for parallel ones
 - ▶ If you can't trust the serial implementation, why would you in a parallel context?

Coding Guidelines

- ▶ write clean code that prevents bugs or facilitates their detection, e.g.
 - ▶ use meaningful identifiers
 - ▶ minimize vertical distance of variables
 - ▶ don't use OpenMP's `private`
 - ▶ follow the Don't Repeat Yourself (DRY) principle (single component per feature)
 - ▶ ...
- ▶ The toolchain you must use!
 - ▶ read & heed compiler warnings
 - ▶ write and regularly run unit and/or integration tests, especially aimed at (varying degrees of) parallelism
 - ▶ use code coverage tests
 - ▶ use continuous integration
 - ▶ use source version control



“Best of” Commit Messages Encountered in the Past

stuff

Added performance fix for
DataItemManager::get() by caching fragment
result in reference

manager stuff

more manager stuff

Removed debug print statement

refactoring stuff

Fixed a linking issue of the unwrap_tuple function

::w

Redirected runtime system output to error stream

:q

Merge branch 'master'

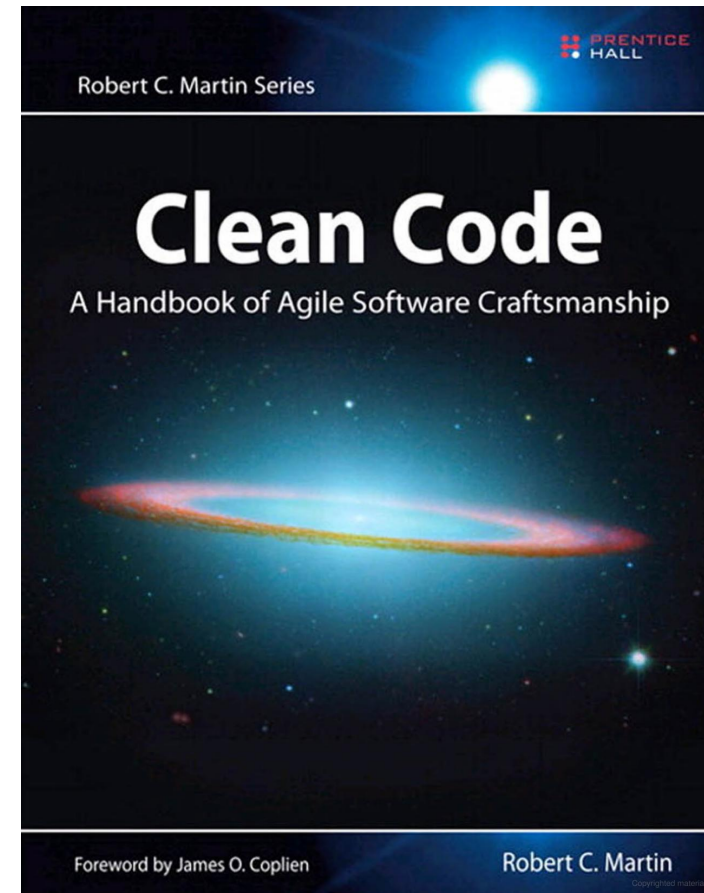
Removed debug print statement

dl;adlwa

fixing typos

Recommended Reading/Reference Material

- ▶ “Clean Code” by Robert Martin, Prentice Hall 2008
 - ▶ ISBN 9780132350884
 - ▶ also available in German
- ▶ naming, functions, commenting, formatting, data structures, error handling, unit tests, classes, concurrency, refinement & refactoring, ...



Generic Debugging Guidelines

- ▶ create a **Minimal Working Example (MWE)**
 - ▶ minimize problem size
 - ▶ minimize software components/features involved
 - ▶ ensure/increase reproducibility
 - ▶ if parallel
 - ▶ minimize machine size (number of threads and/or ranks)
 - ▶ minimize complexity of parallel interaction (e.g. communication patterns, ...)
- ▶ minimizes debugging feedback cycles times, amount of memory to inspect, amount of code to consider, overall degree of complexity of component & parallel interaction
 - ▶ sounds simple, but don't underestimate this
 - ▶ every change along the way to an MWE gives you more information about the problem

Serial Debuggers

▶ gdb

- ▶ useful for inspecting memory contents and getting call stacks
- ▶ can work with multi-threaded programs and also MPI
 - ▶ `mpiexec -n X gdb -ex 'run' -ex 'bt' -ex 'quit' ./a.out`
- ▶ can be used to debug a single MPI process among many
 - ▶ `mpiexec -n 1 gdb ./a.out : -n X-1 ./a.out`
- ▶ can be attached to already-running processes
 - ▶ `gdb -pid 12345`

▶ valgrind

- ▶ mostly used for finding memory leaks (can also simulate cache or generate call graph)
- ▶ can work with multi-threaded programs (but no parallel execution!)
- ▶ can yield some seemingly false positives for OpenMP related to thread-local storage

Sanitizers (Still Mostly Serial)

- ▶ tools that instrument code at compile time to perform checks at runtime
 - ▶ often lower overhead compared to external tools
 - ▶ if in doubt, check same issue with multiple tools (e.g. address sanitizers of multiple compilers and valgrind)
- ▶ depending on compiler, several sanitizers available, e.g.
 - ▶ address: buffer overflows, use-after-free, stack corruption, etc.
 - ▶ undefined behavior: signed integer overflow, float division by zero, negative shift operands, etc.
 - ▶ thread: detects data races
 - ▶ leak: detects memory leaks

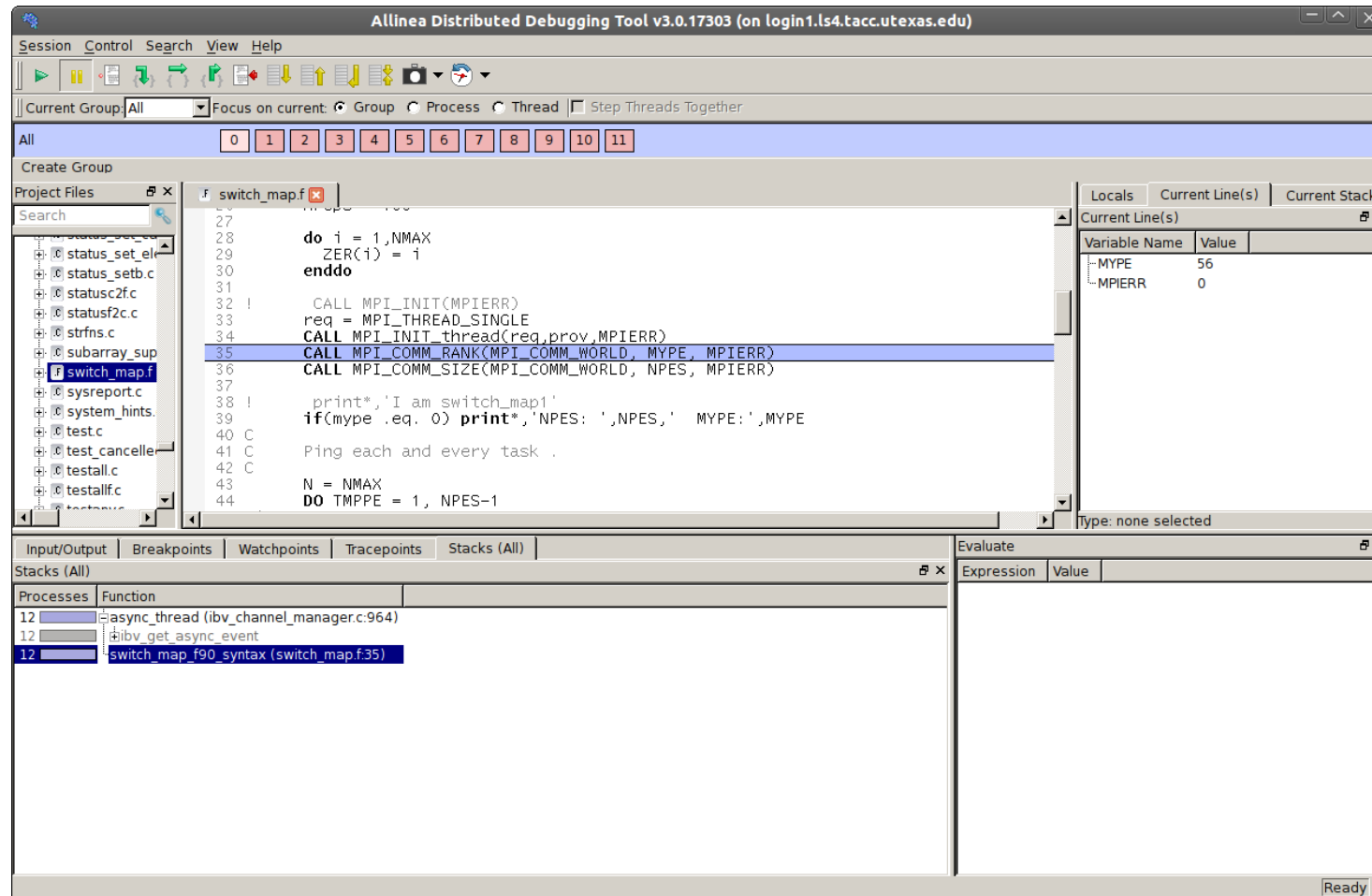
Call Graph Generators

- ▶ many tools available for generating call graphs
- ▶ static (at compile time)
 - ▶ doxygen, opt (llvm), cflow (gcc), etc.
- ▶ dynamic (at runtime)
 - ▶ gprof, callgrind, OpenPAT, pprof, CodeAnalyst, etc.
 - ▶ most performance analysis tools offer some form of call graph generation

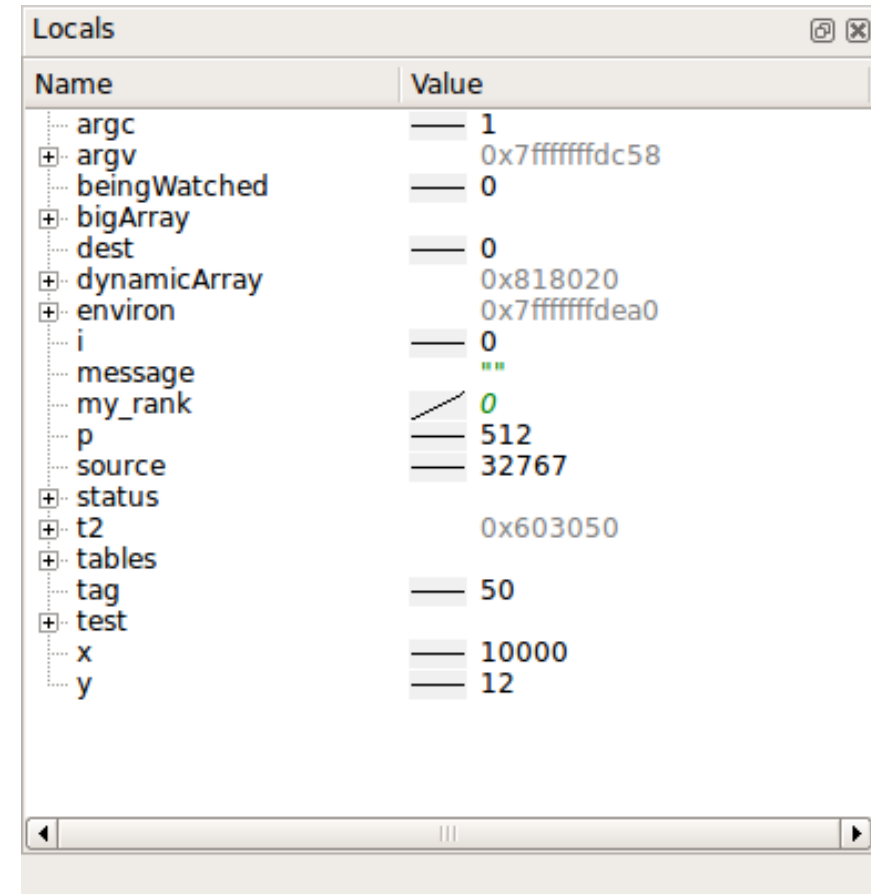
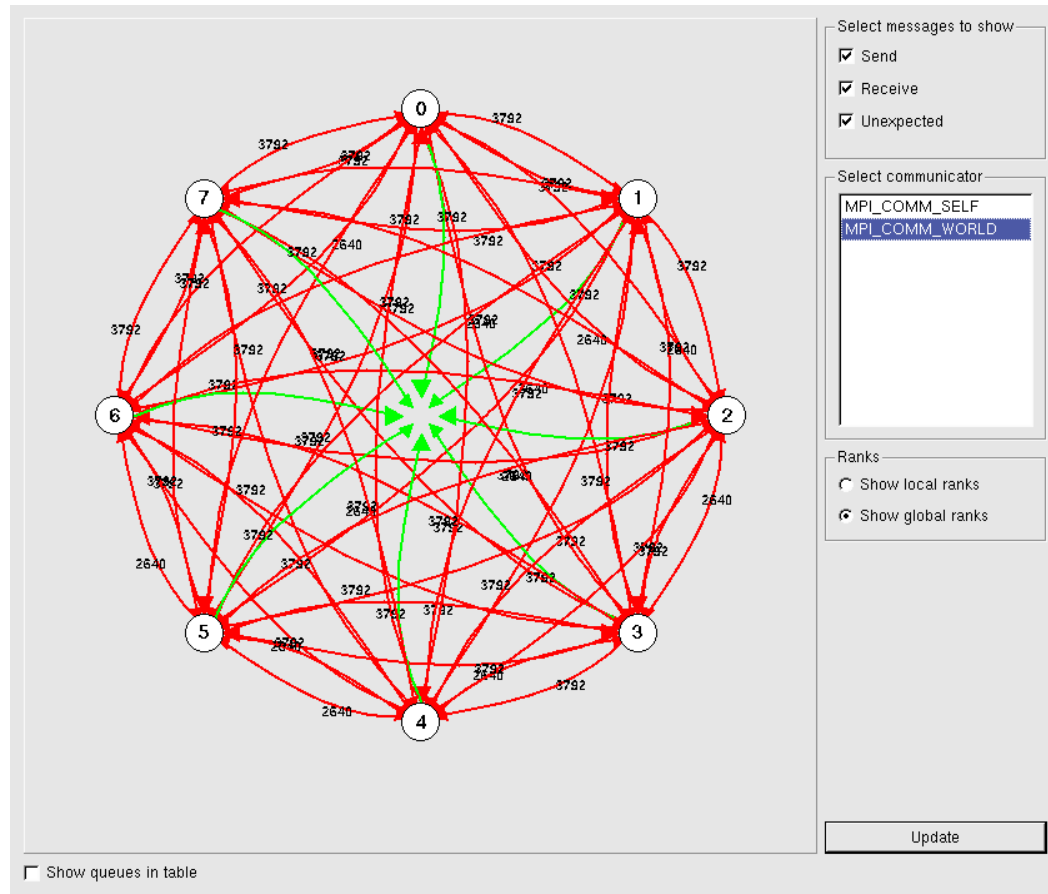
Parallel Debuggers

- ▶ very little free software
- ▶ two commercial top dogs: DDT (ARM) and TotalView (Rogue Wave Software)
- ▶ support OpenMP, MPI, CUDA, etc.
- ▶ several features centered around parallelism
 - ▶ examine variables per rank/thread, examine send/receive queues of MPI libraries, etc.
 - ▶ still, limited usefulness

DDT Screenshot (Overview)



DDT Screenshots (Communication Patterns, Data Across Ranks)



A screenshot of the DDT 'Locals' window showing a list of variables and their values. The window has a title bar with 'Locals' and standard window controls. The table below lists the variables and their corresponding values.

Name	Value
argc	1
argv	0x7fffffffdc58
beingWatched	0
bigArray	
dest	0
dynamicArray	0x818020
environ	0x7fffffffdea0
i	0
message	""
my_rank	0
p	512
source	32767
status	
t2	0x603050
tables	
tag	50
test	
x	10000
y	12

Automatic Race Condition Debugging

- ▶ difficult to do automatically and exactly
 - ▶ statically detecting race conditions is NP-hard
 - ▶ dynamically detecting race conditions incurs large runtime overhead (every memory access and synchronization action must be logged and checked)
- ▶ most solutions resort to heuristics
 - ▶ several experimental tools available in research
 - ▶ many issues: limited scope, only apply to a subset of programming language, etc.
 - ▶ few “mature” tools, e.g. Intel Inspector

Intel Inspector

▶ features

- ▶ free
- ▶ Linux & Windows version
- ▶ automatically finds bugs in multi-threaded programs
 - ▶ deadlocks
 - ▶ memory corruption
 - ▶ race conditions
 - ▶ vulnerabilities
- ▶ supports OpenMP, TBB, Pthreads, Windows threads

▶ limitations & issues

- ▶ slowdown by 1-2 orders of magnitude!
- ▶ explicit support only for Intel OpenMP runtime
- ▶ error detection only at runtime, only in executed control flow branches
- ▶ false positives and negatives possible

OpenMP Data Race Example 1

```
int counter = 0;

#pragma omp parallel for
for(int i = 0; i < 10; ++i) {
    counter++;
}
```

Description ▲	Source	Function	Module
Read	ConsoleApplication1.cpp:9	main	consoleapplication1.exe
7	#pragma omp parallel for		
8	for (int i = 0; i < 10; ++i) {		
9	counter++;		
10	}		
11			
Write	ConsoleApplication1.cpp:9	main	consoleapplication1.exe
7	#pragma omp parallel for		
8	for (int i = 0; i < 10; ++i) {		
9	counter++;		
10	}		
11			

OpenMP Data Race Example 2

```
int sum = 0;

#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    int tmp = sum;
    tmp = tmp + 1;
    sum = tmp;
}
```

Description ▲	Source	Function	Module
Read	ConsoleApplication1.cpp:17	main	consoleapplication1.exe
15	#pragma omp parallel for		
16	for (int i = 0; i < 10; i++) {		
17	int tmp = sum;		
18	tmp = tmp + 1;		
19	sum = tmp;		
Write	ConsoleApplication1.cpp:19	main	consoleapplication1.exe
17	int tmp = sum;		
18	tmp = tmp + 1;		
19	sum = tmp;		
20	}		
21			

OpenMP Data Race Example 2: Wrong Fix

```
int sum = 0;

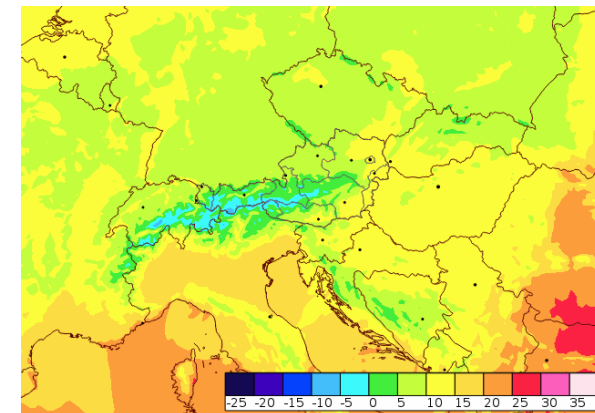
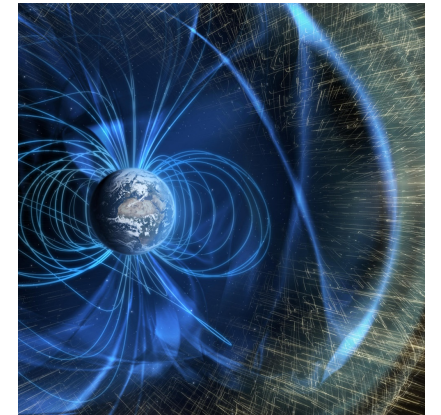
#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    int tmp;
    #pragma omp critical
    tmp = sum;
    tmp = tmp + 1;
    #pragma omp critical
    sum = tmp;
}
```



(not detected by Intel Inspector 2020)

Domain-specific Debugging

- ▶ Visualize the output using appropriate tools
 - ▶ gnuplot
 - ▶ ParaView
 - ▶ ...
- ▶ note that this prohibits automatic checking
 - ▶ whenever feasible, unit and integration tests are preferred



Best Approach to Debugging Parallel Programs

- ▶ know your algorithm and implementation
 - ▶ e.g. “an n-body simulation using Barnes-Hut”
- ▶ know your programming models and languages, and their semantics
 - ▶ “OpenMP threadprivates persist per thread between parallel regions with the same number of threads and affinity policies”
 - ▶ “this C++ object’s destructor will be called at the end of the full-expression”
- ▶ Don’t trust (seemingly) automatic analysis tools too much, read and understand the source code when available!



Performance Debugging



Performance Debugging

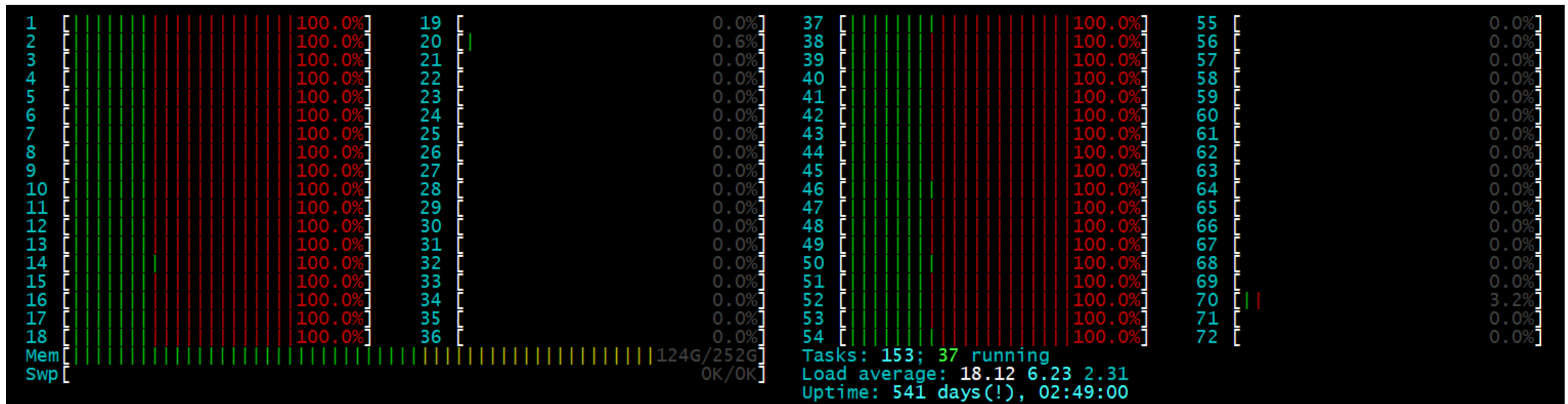
- ▶ also sometimes known as “*non-functional*” debugging (not related to functional output)
 - ▶ short execution time not necessarily but most often the only goal
 - ▶ much more tricky than functional debugging
 - ▶ How do you know the performance bug was fixed? Because it’s “faster” now?
- ▶ most aspects of functional debugging or sequential programs still apply
 - ▶ coding guidelines & best practice
 - ▶ + reproducibility (e.g. fix random seeds, scheduling affinities, ...)
 - ▶ + if required, performance unit tests, performance regression checks
 - ▶ + performance tools (the ones for sequential programs can also be useful)
 - ▶ + a lot more knowledge about hardware required

(h)top

- ▶ Don't underestimate the power of top or htop!
- ▶ Get a high-level overview of the workload on the system (and it's components) and compare to what you expected!
 - ▶ What's the ratio between user time and system time?
 - ▶ high system time could be caused by inefficient I/O, high amount of context switching, etc.
 - ▶ Which CPU cores am I really using?
 - ▶ the only way to verify affinity policies
 - ▶ What is the actual memory footprint vs. what it should be?
 - ▶ detect existence of memory leaks without any analysis tools

htop & affinity

- ▶ 2x Intel E5-2699 v3 (18 cores per CPU) in a single node
- ▶ htop shows cores 1-18 and 37-54 busy, hence 36 cores total – right?



Recap: perf

```
[c703429@login.lcc2 ~]$ perf stat ./heat_stencil_1D_seq
...
28,826,239,136 cycles:u          #    2.471 GHz
35,220,856,783 instructions:u    #    1.22  insn per cycle
 6,711,849,029 branches:u        # 575.356 M/sec
    1,295,209 branch-misses:u    #    0.02% of all branches
      1,044 LLC-load-misses:u
        26 LLC-store-misses:u
    15,312,122 L1-dcache-load-misses:u
    476,440,489 L1-dcache-store-misses:u
```

Terminology

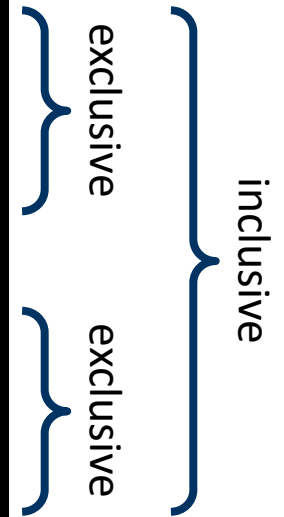
▶ instrumentation

- ▶ add source/machine code that will measure something when executed
- ▶ can happen manually, automatically, during compilation, linking, runtime, ...
- ▶ do not confuse with “measurement”

▶ inclusive/exclusive measurements

- ▶ do measurements include data for nested code regions (e.g. functions)?

```
int outside() {  
    for(int i = 0; i < N; ++i) {  
        // work  
    }  
    inside();  
    for(int j = 0; j < M; ++j) {  
        // more work  
    }  
}
```



More Terminology: Sample- vs. Trace-based Profiling

▶ Sampling

- ▶ gives aggregated information of how much time spent where in the code
- ▶ based on statistics: does not give information on the order of different events, their time interval or exact numbers
- ▶ easy to accomplish, comparatively low overhead, no code changes required
 - ▶ stop program periodically and read program counter of CPU
 - ▶ build histogram at the end

▶ Tracing

- ▶ produces a detailed log of which event happened at what point in time
- ▶ allows to establish order of events, even across processes/nodes if clocks are in sync
- ▶ requires code changes/instrumentation
 - ▶ e.g. wrap every function call with
`start_timer();`
`func_call();`
`end_timer();`

gprof

- ▶ sample-based profiler
 - ▶ also limited code instrumenter for call graph generation and call counts
 - ▶ very simplistic, not always accurate
- ▶ available with every GCC installation
- ▶ very simple in its use
 - ▶ compile with debug symbols (-g) and gprof support (-pg)
 - ▶ run binary as usual
 - ▶ run `gprof binary gmon.out` to view results
 - ▶ use `--line` to get more detailed, line-based results

gprof Example

```
int foo() {  
    long long counter = 0;  
    #pragma omp parallel for  
    for(int i = 0; i < N; ++i) {  
        #pragma omp critical  
        counter++;  
    }  
    return counter;  
}
```

```
int bar() {  
    long long partSum[MAX_NUM_THREADS][8];  
    long long counter = 0;  
    #pragma omp parallel  
    {  
        int tid = omp_get_thread_num();  
        partSum[tid][0] = 0;  
        #pragma omp for  
        for(int i=0; i<N; ++i) partSum[tid][0]++;  
        #pragma omp critical  
        counter += partSum[tid][0];  
    }  
    return counter;  
}
```


gprof Example cont'd

Flat profile:

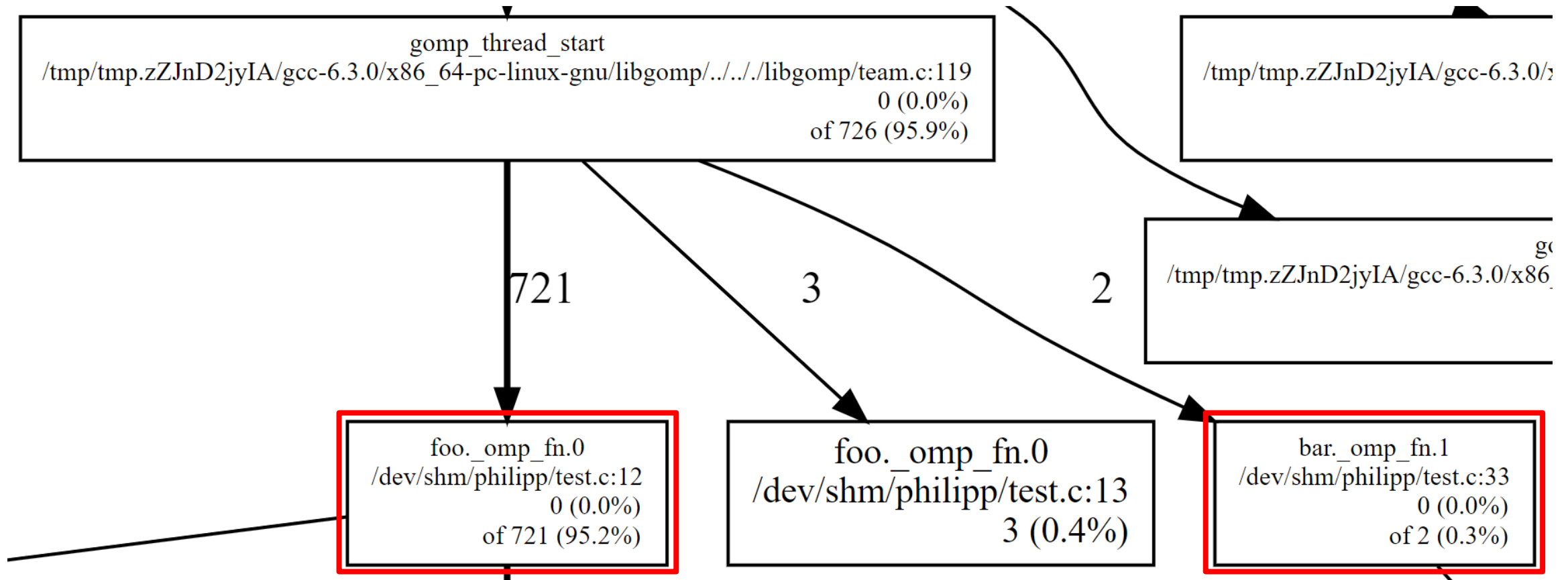
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
100.71	0.02	0.02				foo._omp_fn.0 (test.c:13 @ 400a3d)
0.00	0.02	0.00	1	0.00	0.00	bar (test.c:19 @ 40092c)
0.00	0.02	0.00	1	0.00	0.00	foo (test.c:8 @ 4008e6)

gperftools

- ▶ sample-based profiler
 - ▶ formerly Google Performance Tools
- ▶ actually a collection of performance analysis tools and high-performance multi-threaded memory allocators
- ▶ very simple in its use
 - ▶ install gperftools library
 - ▶ link with `-lprofiler`
 - ▶ run with environment variable `CPUPROFILE=prof.out`
 - ▶ run `pprof binary prof.out` to view results (`--gv` for graphical visualization)

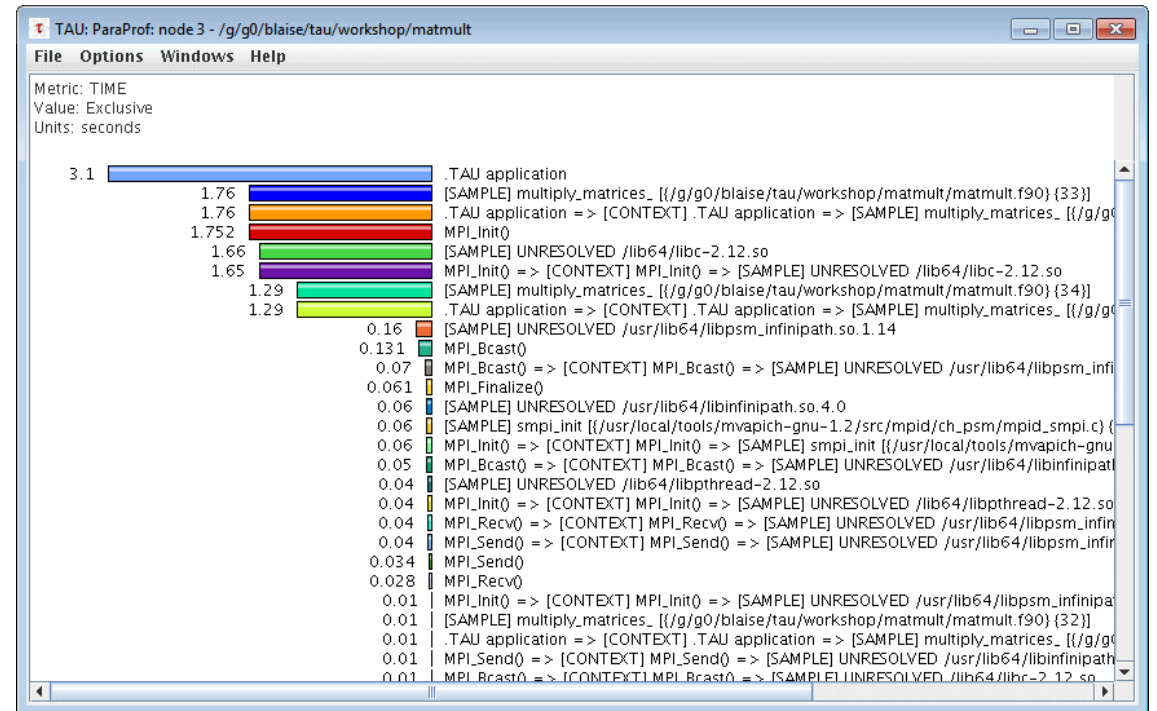
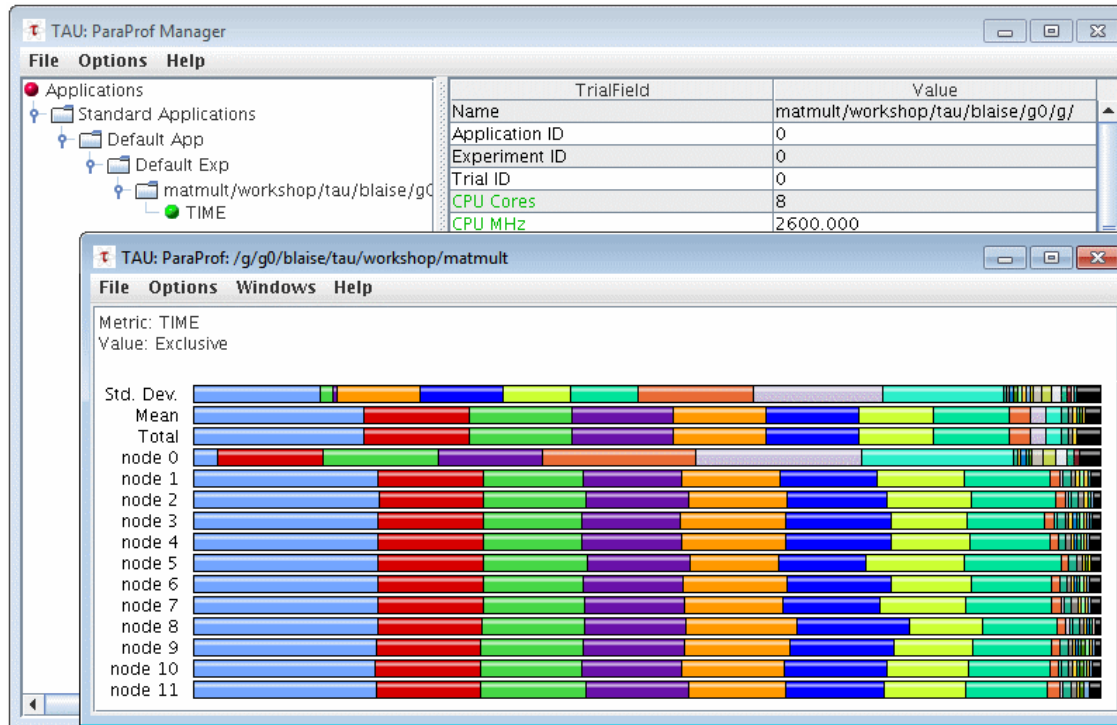
gperftools Example



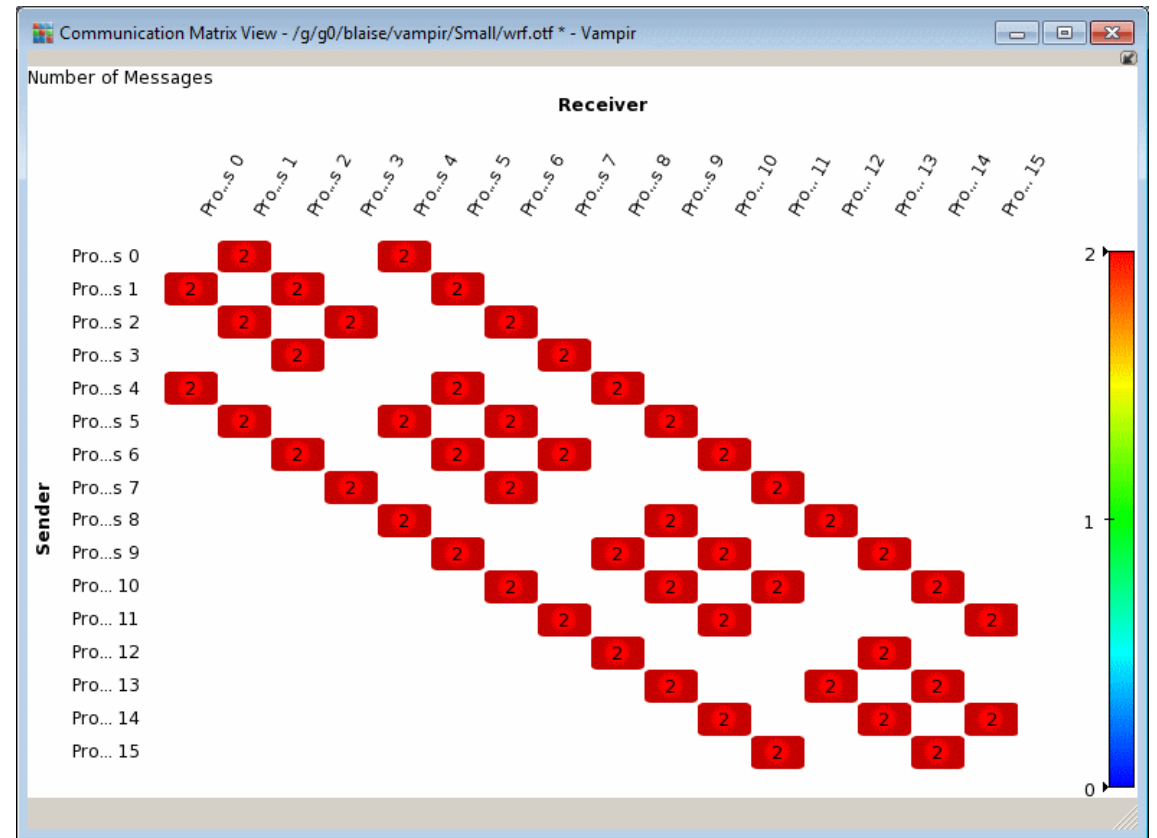
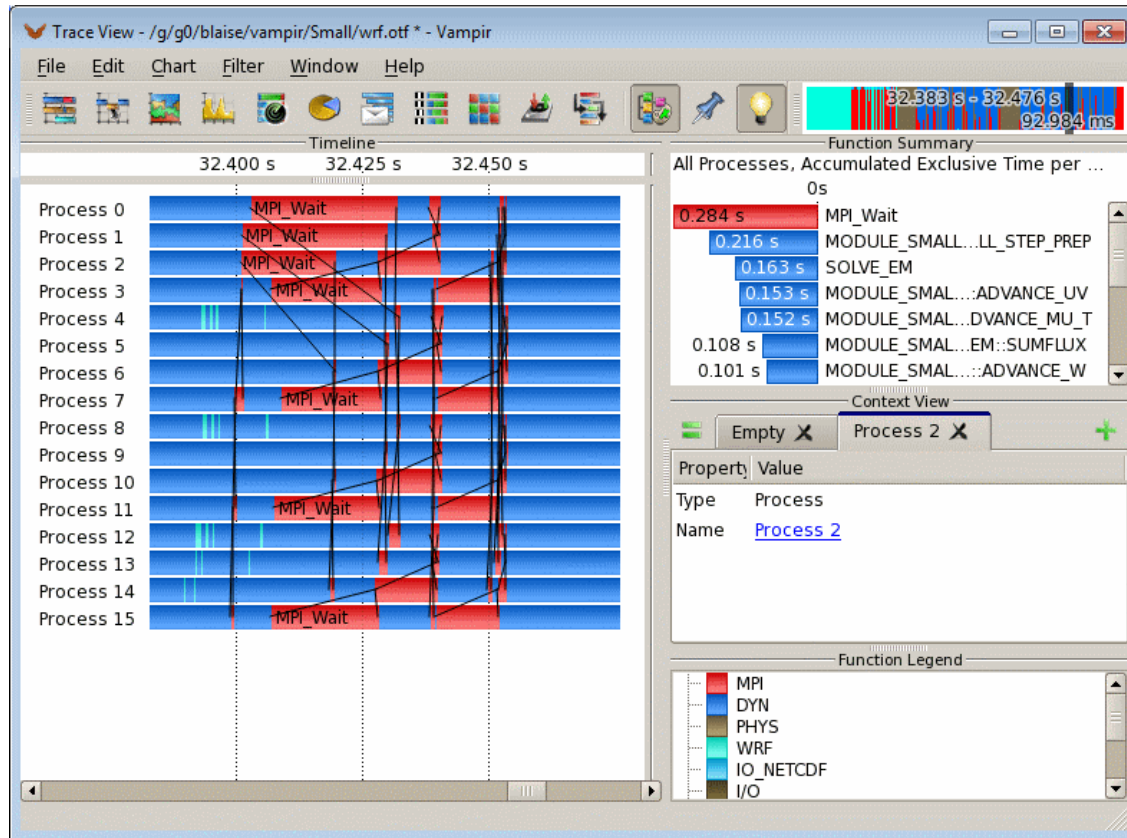
Performance Analysis Tools for Parallel Programs

- ▶ **profiling and analysis software**
 - ▶ Intel Pin: dynamic binary instrumentation
 - ▶ Intel VTune: performance analysis for multi-threaded programs
 - ▶ Intel Advisor: dependency, vectorization and cache analysis tool
 - ▶ AMD CodeXL: profiler and debugger for GPUs
 - ▶ TAU: profiling and tracing toolkit
 - ▶ PAPI: library for access to hardware event counters
 - ▶ OProfile: sampling-based profiler with hardware event counter support
 - ▶ also, some software built into your IDE, e.g. MS Visual Studio
- ▶ **analysis and visualization/reporting tools**
 - ▶ Scalasca, Vampir, Paraver, JumpShot, paraprof, CUBE, etc.
- ▶ These lists are by far not complete!

TAU & ParaProf



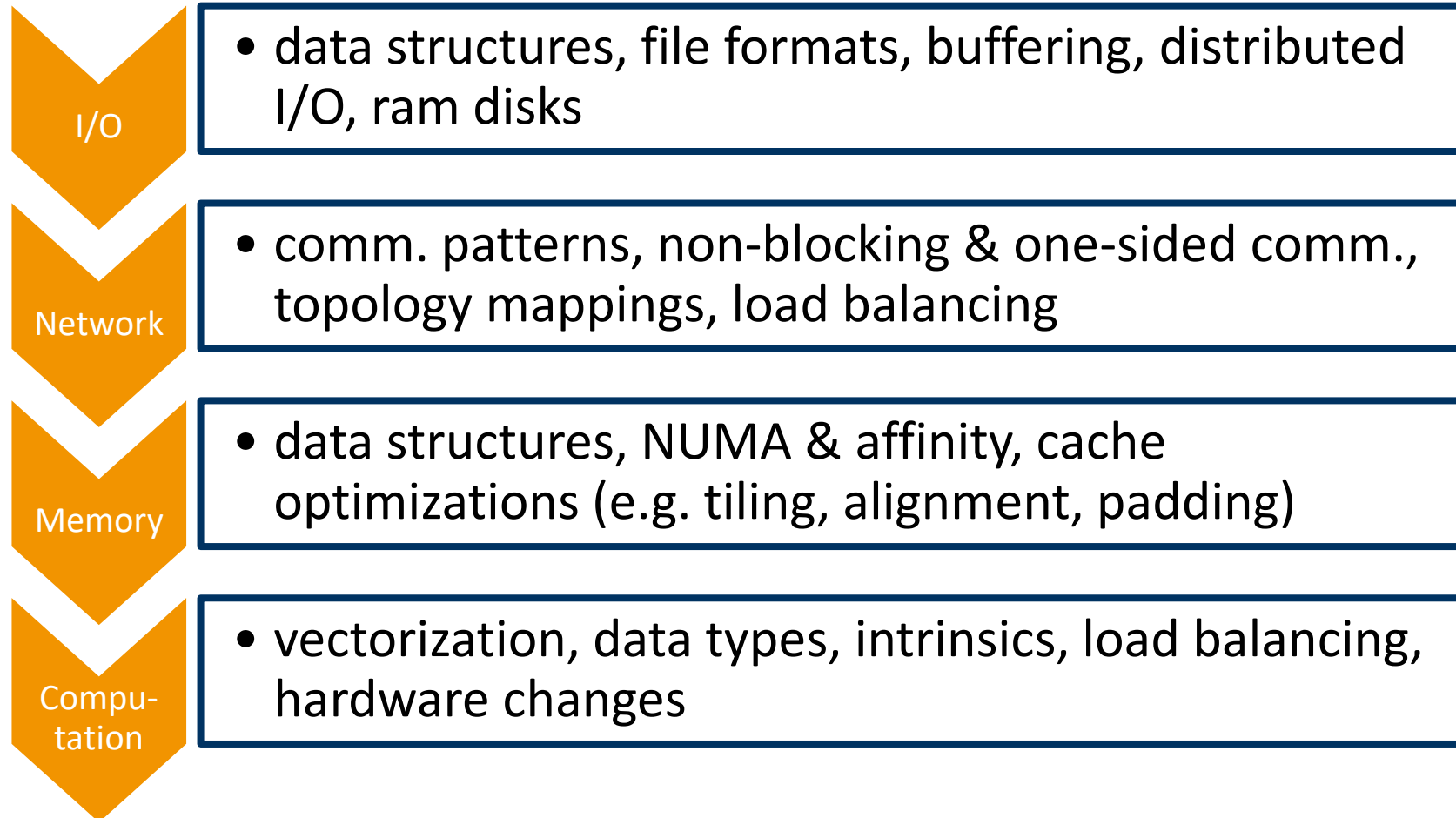
Vampir



General Hints When Working With Debuggers

- ▶ `-g` when compiling if source locations are required
- ▶ careful with optimization flags, especially `-O#`
 - ▶ function inlining, loop fusion/fission, ...
 - ▶ likely to obfuscate source code locations
 - ▶ if feasible, work in `-O0` or temporarily disable conflicting flags
- ▶ check whether child processes are included in analysis/reports
- ▶ check whether threads are included in analysis/reports
- ▶ if tracing or otherwise large-overhead instrumentation required, restrict to code regions of interest

Points of Attack in Order of Benefit



Summary

- ▶ **functional debugging**

- ▶ adhere to coding guideline and best practices of software engineering
- ▶ especially relevant for parallelism: know your programming models and semantics, don't trust automatic tools blindly

- ▶ **performance debugging**

- ▶ don't underestimate the power of simple tools
- ▶ many more advanced tools out there, but not straight-forward to use
- ▶ know your hardware and your program hotspots

Image Sources

- ▶ Yoda: <https://www.deviantart.com/biggiepoppa/art/Master-Yoda-Star-Wars-395511111>
- ▶ DDT: <https://portal.tacc.utexas.edu/software/ddt>, https://www.sharcnet.ca/help/index.php/Parallel_Debugging_with_DDT, <https://developer.arm.com/docs/101136/latest/ddt/viewing-variables-and-data>
- ▶ Domain-specific debugging: <https://twitter.com/maven2mars/status/984440044659159040>, <https://www.nasa.gov/ames/image-feature/nasa-highlights-simulations-at-supercomputing-conference-like-aircraft-landing-gear>, ZAMG Wettervorhersage 03.10.2019 12:00
- ▶ TAU & ParaProf: <https://hpc.llnl.gov/software/development-environment-software/tau-tuning-and-analysis-utilities>
- ▶ Vampir: <https://hpc.llnl.gov/software/development-environment-software/vampir-vampir-server>