Performance Analysis and Optimization Tool

Andres S. CHARIF-RUBIAL

andres.charif@uvsq.fr

Performance Analysis Team, University of Versailles

http://www.maqao.org

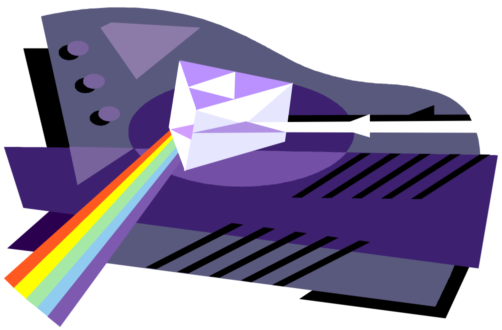
Introduction
Performance Analysis

- Develop performance analysis and optimization tools: MAQAO Framework and Toolsuite

- Establish partnerships

- Optimize industrial applications

- Understand the performance of an application
  - How well it behaves on a given machine
- What are the issues ?



- Generally a multifaceted problem
  - Maximizing the number of views = better understand
- Use techniques and tools to understand issues
- Once understood ⇨ Optimize application

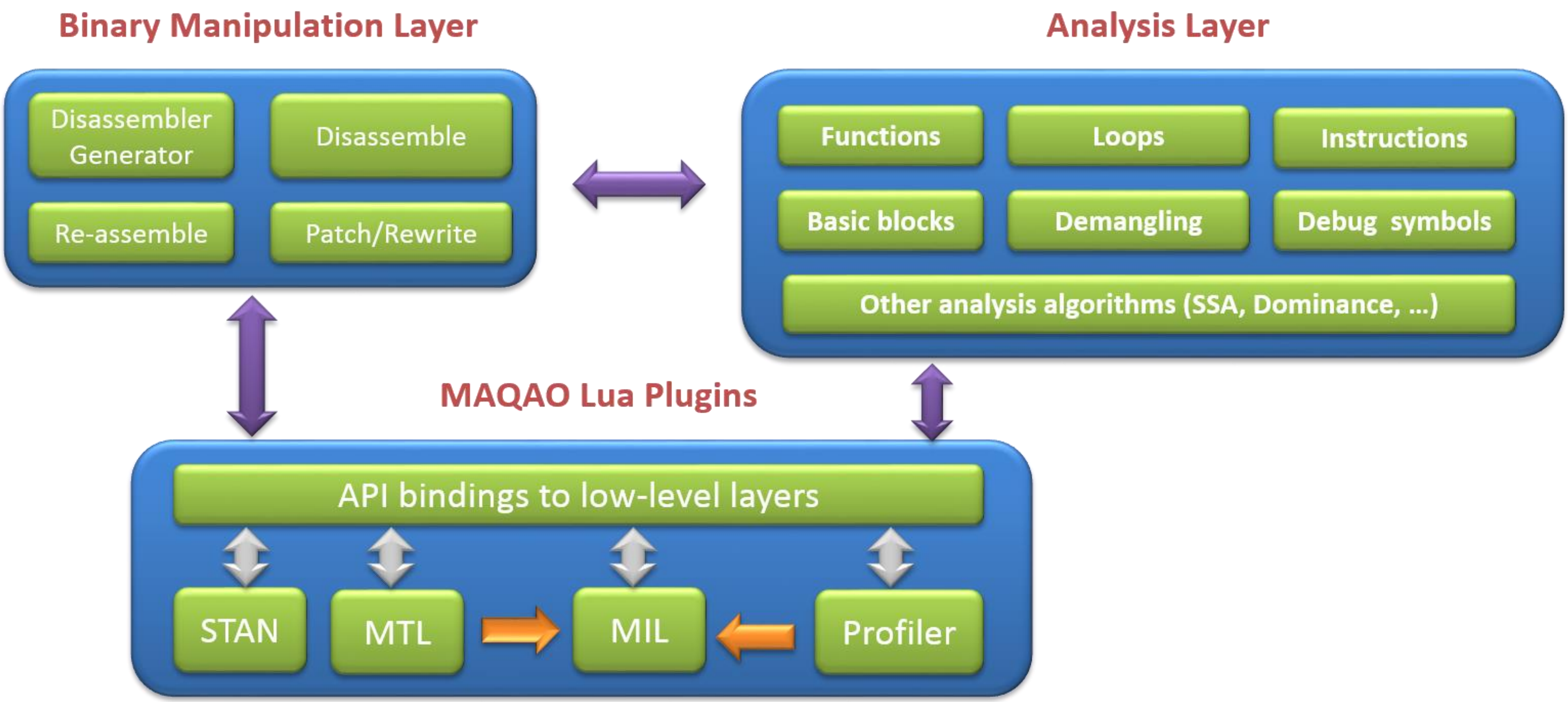
- Compiler remains your best friend
  - Be sure to select proper flags (e.g., -xavx)
  - Pragmas: Unrolling, Vector alignment
  - O2 V.S. O3
  - Vectorisation/optimisation report

Introduction
MAQAO Tool

- Open source (LGPL 3.0)
  - Currently binary release

- Available for x86-64 and Xeon Phi
  - Looking forward in porting MAQAO on BlueGene

- Easy install
  - Packaging : ONE (static) standalone binary
    - Easy to embeed
- Audience
  - User/Tool developer: analysis and optimisation tool
  - Performance tool developer: framework services
    - TAU: tau_rewrite (MIL)
    - ScoreP: on-going effort (MIL)



Binary Manipulation Layer
Disassembler Generator
Disassemble
Re-assemble
Patch/Rewrite
Analysis Layer
Functions
Loops
Instructions
Basic blocks
Demangling
Debug symbols
Other analysis algorithms (SSA, Dominance, ...)
MAQAO Lua Plugins
API bindings to low-level layers
STAN
MTL
MIL
Profiler

- Scripting language
  - Lua language : simplicity and productivity
  - Fast prototyping
  - MAQAO Lua API : Access to services

- Built on top of the Framework
- Loop-centric approach
- Produce reports
  - We deal with low level details
  - You get high level reports

- A lot of tools ! Which one to use ? When
- Our approach/experience: decision tree
  - Currently working on HPC
    - Multi-node > Node > Socket > Core
    - Classify IO/Memory/MPI/OpenMP/Application
- PAMDA methodology
  - to be published: 7th Parallel Tools Workshop
  - https://tools.zih.tu-dresden.de/2013/

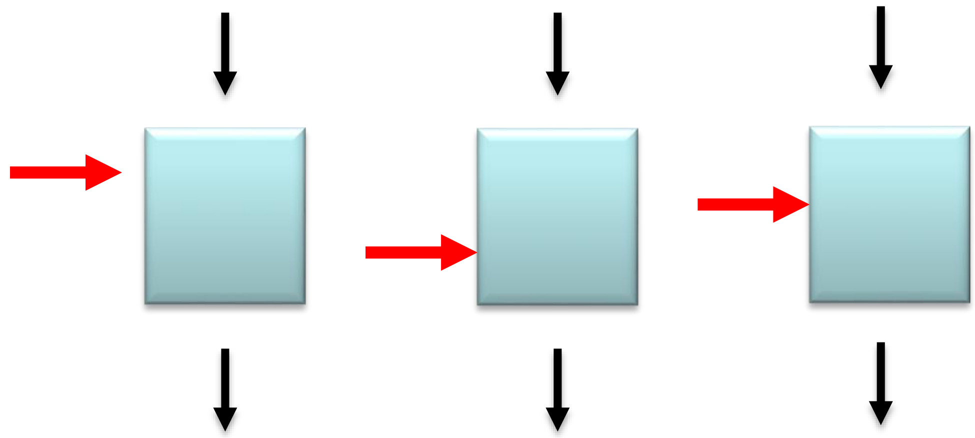
Outline

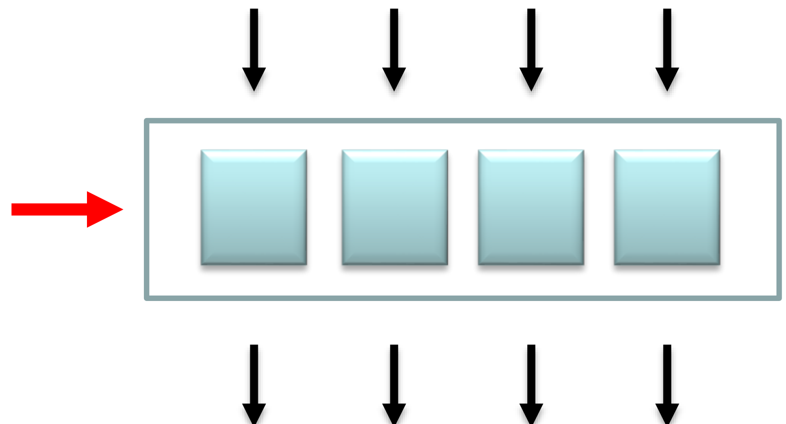
- Introduction
- Pinpointing hotspots
  - Functions, loops
  - MPI characterization
- Code quality analysis

- MAQAO Profiling
  - Instrumentation
    - Through binary rewriting
    - High overhead / More precision
  - Sampling
    - Hardware counter through perf_event_open system call
    - Very low overhead / less details

Pinpointing hotspots
Parallelism level

- SPMD
  - Program level
- SIMD
  - Instruction level
- By default MAQAO only considers system processes and threads





Pinpointing hotspots
Display

- Display functions and their exclusive time
  - Associated callchains and their contribution
  - Loops
- Innermost loops can then be analyzed by the code quality analyzer module (CQA)
- Command line and GUI (HTML) outputs

Pinpointing hotspots

GUI snapshot (1/4)

MAQAO

Performance Evaluation - Profiling results

Hotspots - Functions

Name	Median Excl %Time	Deviation
matmul_sub__ - 56@solve_subs.f	17.16	0.26
compute_rhs__ - 4@rhs.f	10	0.03
y_solve_cell__ - 385@y_solve.f	9.32	0.54
z_solve_cell__ - 385@z_solve.f	8.96	0.14
x_solve_cell__ - 391@x_solve.f	8.68	0.17
MPIDI_CH3I_Progress	5.22	3.66
matvec_sub__ - 5@solve_subs.f	3.92	0.11
x_backsubstitute__ - 330@x_solve.f	3.09	0.14
y_backsubstitute__ - 329@y_solve.f	2.05	0.03
z_backsubstitute__ - 329@z_solve.f	1.98	0.06
copy_faces__ - 4@copy_faces.f	0.88	0.06
MPID_nem_dapl_rc_poll_dyn_opt_	0.74	0.62
MPID_nem_lmt_shm_start_send	0.68	0.06

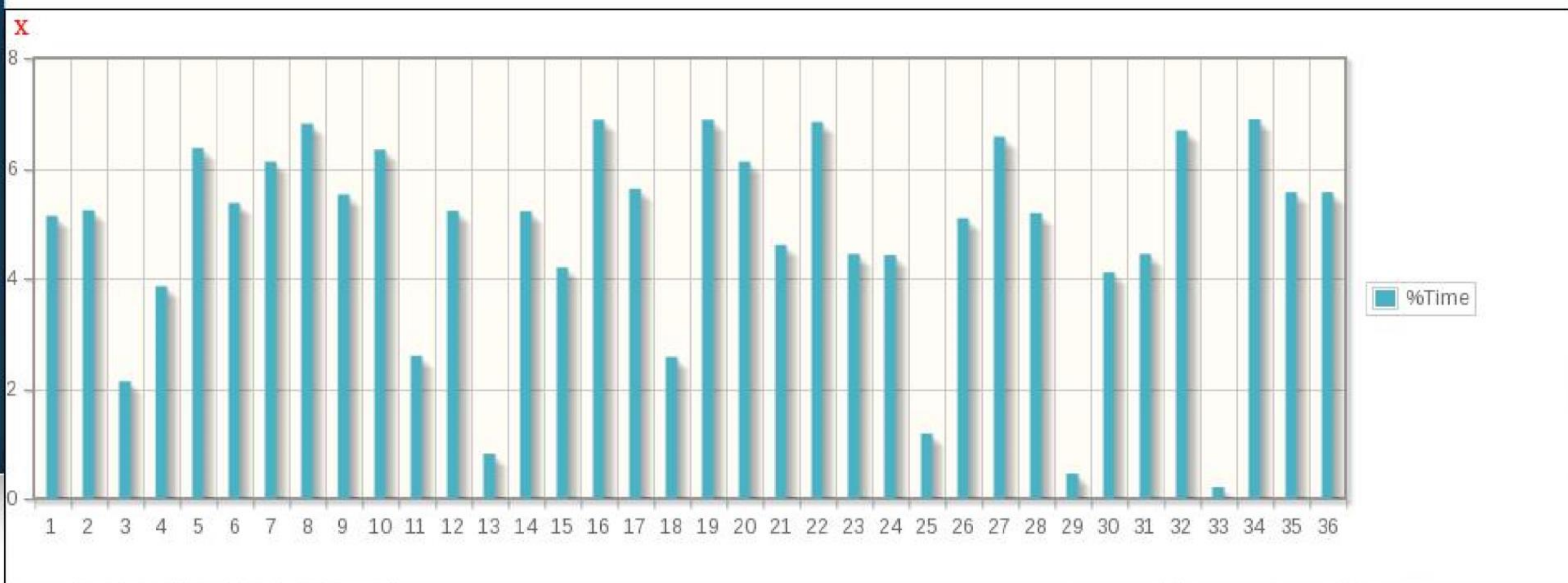
Pinpointing hotspots

GUI snapshot (2/4)

MAQAO

Performance Evaluation - Profiling results

Hotspots - Functions



exact_solution__ - 4@exact_solution.f	0.21	0.03
x_unpack_solve_info__ - 114@x_solve.f	0.14	0.03

Pinpointing hotspots
GUI snapshot (3/4)

cirrus5003 - Process #53572 - Thread #1

Name	Excl %Time	Excl Time (s)
matmul_sub__ - 56@solve_subs.f	16.92	16.48
compute_rhs__ - 4@rhs.f	9.92	9.66
y_solve_cell__ - 385@y_solve.f	9.08	8.84
loops	9.08	
Loop 267 - y_solve.f@415	0	
Loop 268 - y_solve.f@425	0	
Loop 272 - y_solve.f@426	0.25	
Loop 270 - y_solve.f@524	6.57	
Loop 271 - y_solve.f@436	2.22	
Loop 269 - y_solve.f@716	0.04	
x_solve_cell__ - 391@x_solve.f	9.01	8.78
loops	9.01	
Loop 235 - x_solve.f@420	0	
Loop 236 - x_solve.f@429	0	
Loop 237 - x_solve.f@709	0.06	
Loop 239 - x_solve.f@431	2.71	
Loop 238 - x_solve.f@519	6.24	

Pinpointing hotspots
GUI snapshot (4/4)

cirrus5003 - Process #53572 - Thread #1

Name	Excl %Time	Excl Time (s)
matmul_sub__ - 56@solve_subs.f	16.92	16.48
compute_rhs__ - 4@rhs.f	9.92	9.66
y_solve_cell__ - 385@y_solve.f	9.08	8.84
loops	9.08	
Loop 267 - y_solve.f@415	0	
Loop 268 - y_solve.f@425	0	
Loop 272 - y_solve.f@426	0.25	
Loop 270 - y_solve.f@524	6.57	
Loop 271 - y_solve.f@436	2.22	
Loop 269 - y_solve.f@716	0.04	
x_solve_cell__ - 391@x_solve.f	9.01	8.78
loops	9.01	
Loop 235 - x_solve.f@420	0	
Loop 236 - x_solve.f@429	0	
Loop 237 - x_solve.f@709	0.06	
Loop 239 - x_solve.f@431	2.71	
Loop 238 - x_solve.f@519	6.24	

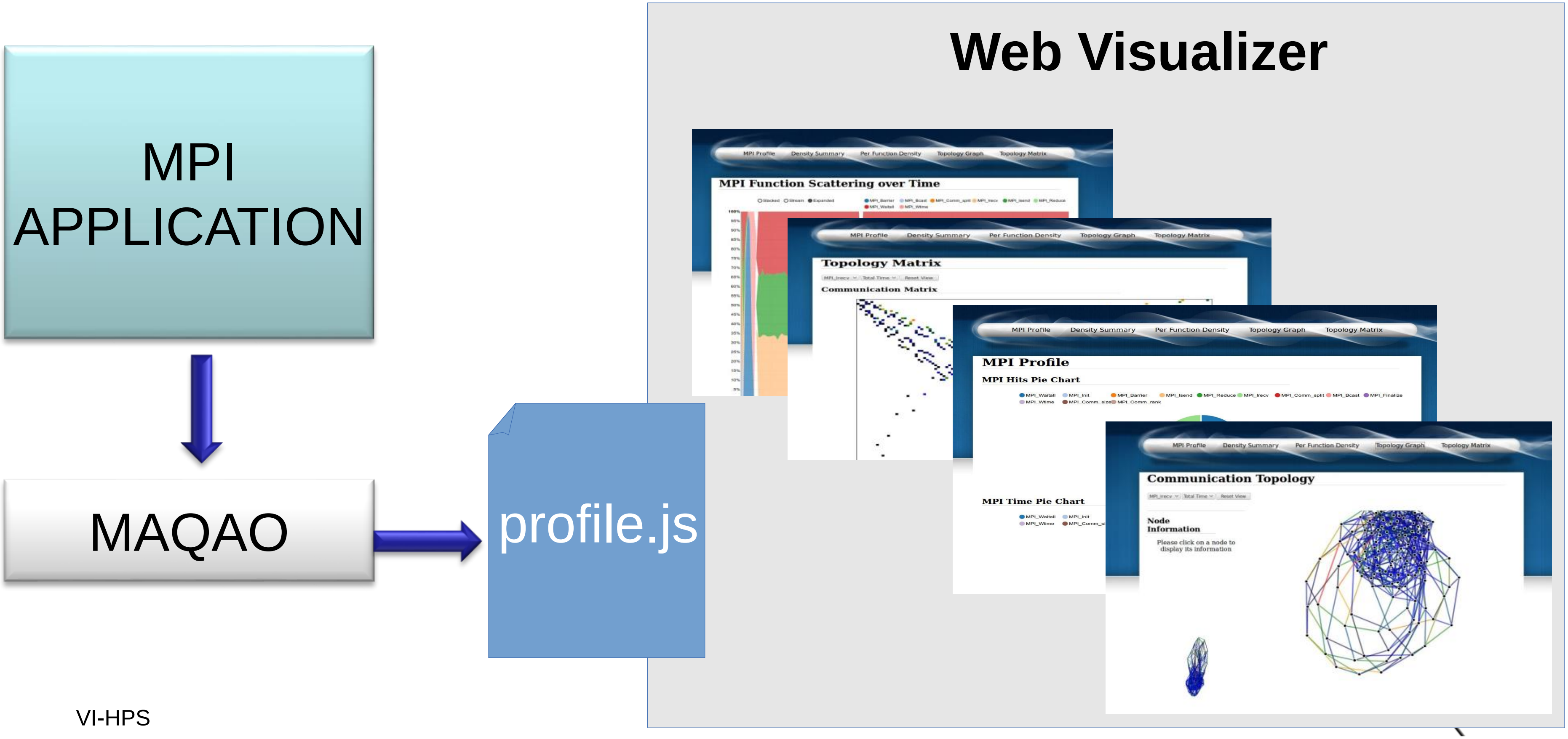
Outline

- Introduction
- Pinpointing hotspots
  - Functions, loops
  - MPI characterization
- Code quality analysis

- Our methodology
  - Coarse grain: overview, global trends/patterns
  - Fine grain: filtering precise issues
- Tracing issues
  - Scalability
  - Memory size: can we reduce it ?
  - Trace size: can we reduce it ?
  - IO’s wall: remove it ?

Pinpointing hotspots
MPI characterization: overview

- Scalable coarse grain analysis

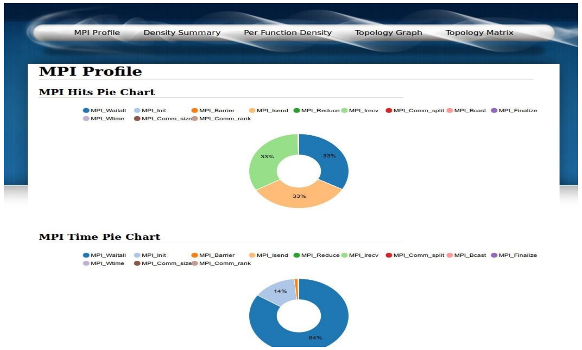


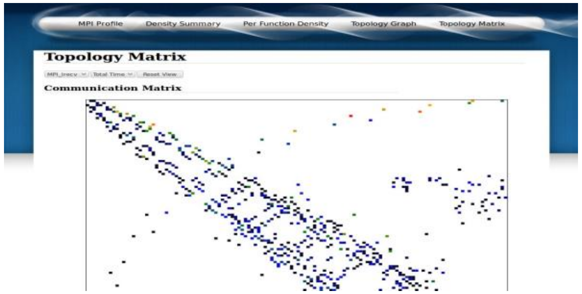
MPI characterization: key concepts

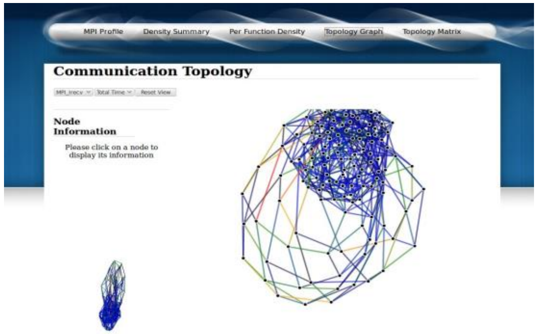
- Online profiling
  - Aggregated metrics
  - No traces
  - No IOs (only one result file)
  - Reduced memory footprint
  - Scalable on 100+ procs

Pinpointing hotspots
MPI characterization: visualization

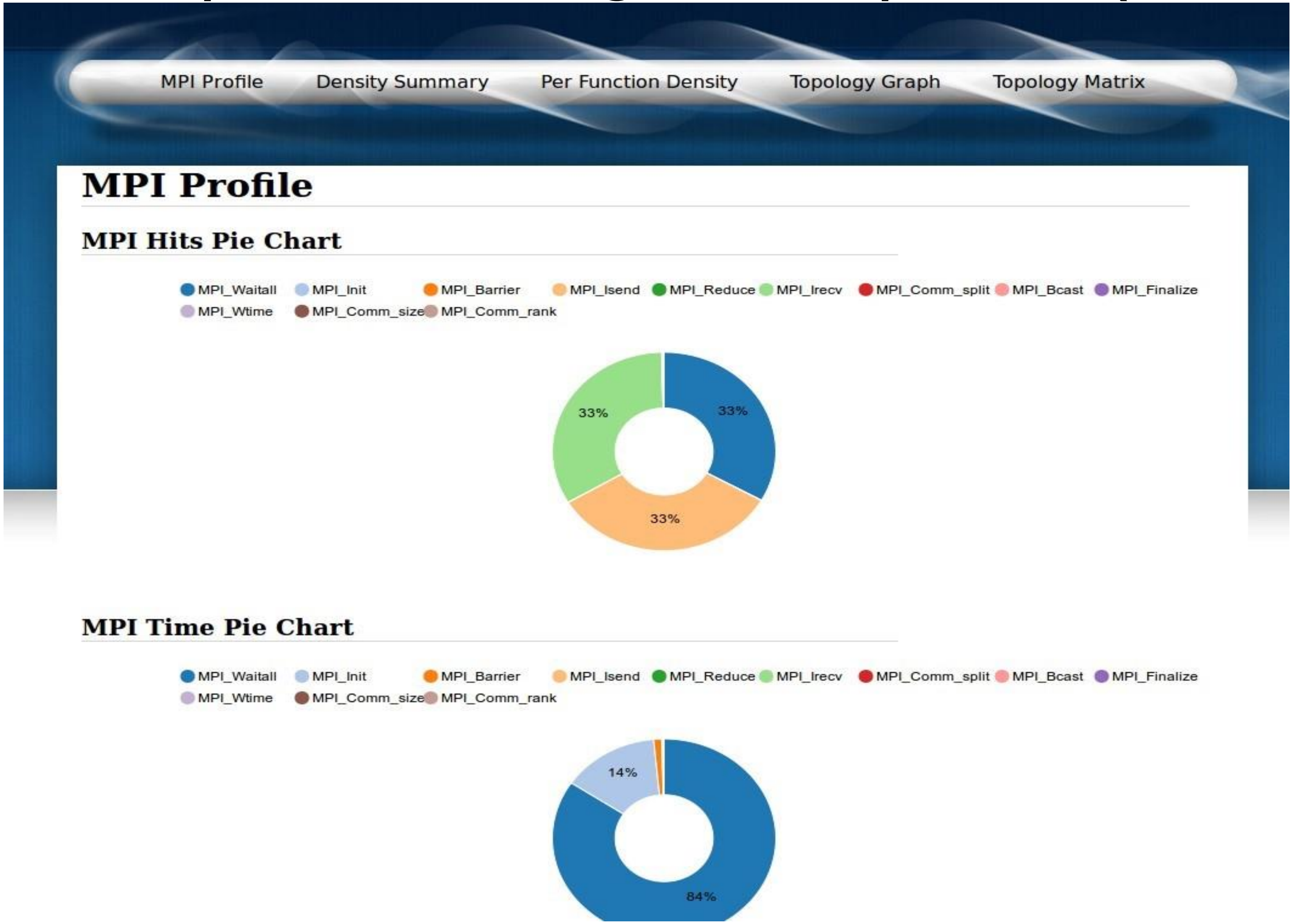
- Web based visualizer
  - Only requires a web browser





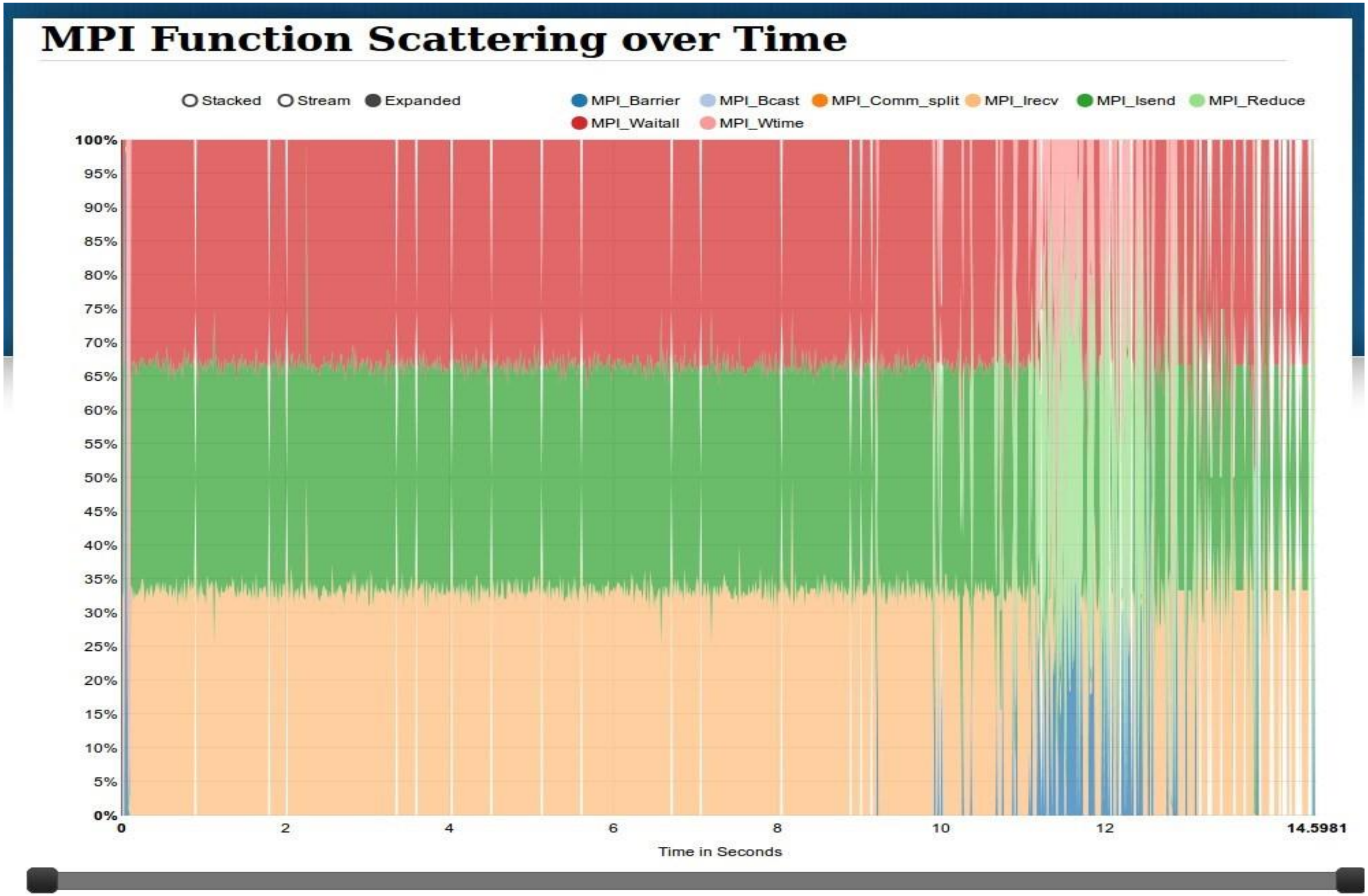


- MPI primitives high level profile (hits,time,size)



Pinpointing hotspots
MPI characterization: analyses (2/5)

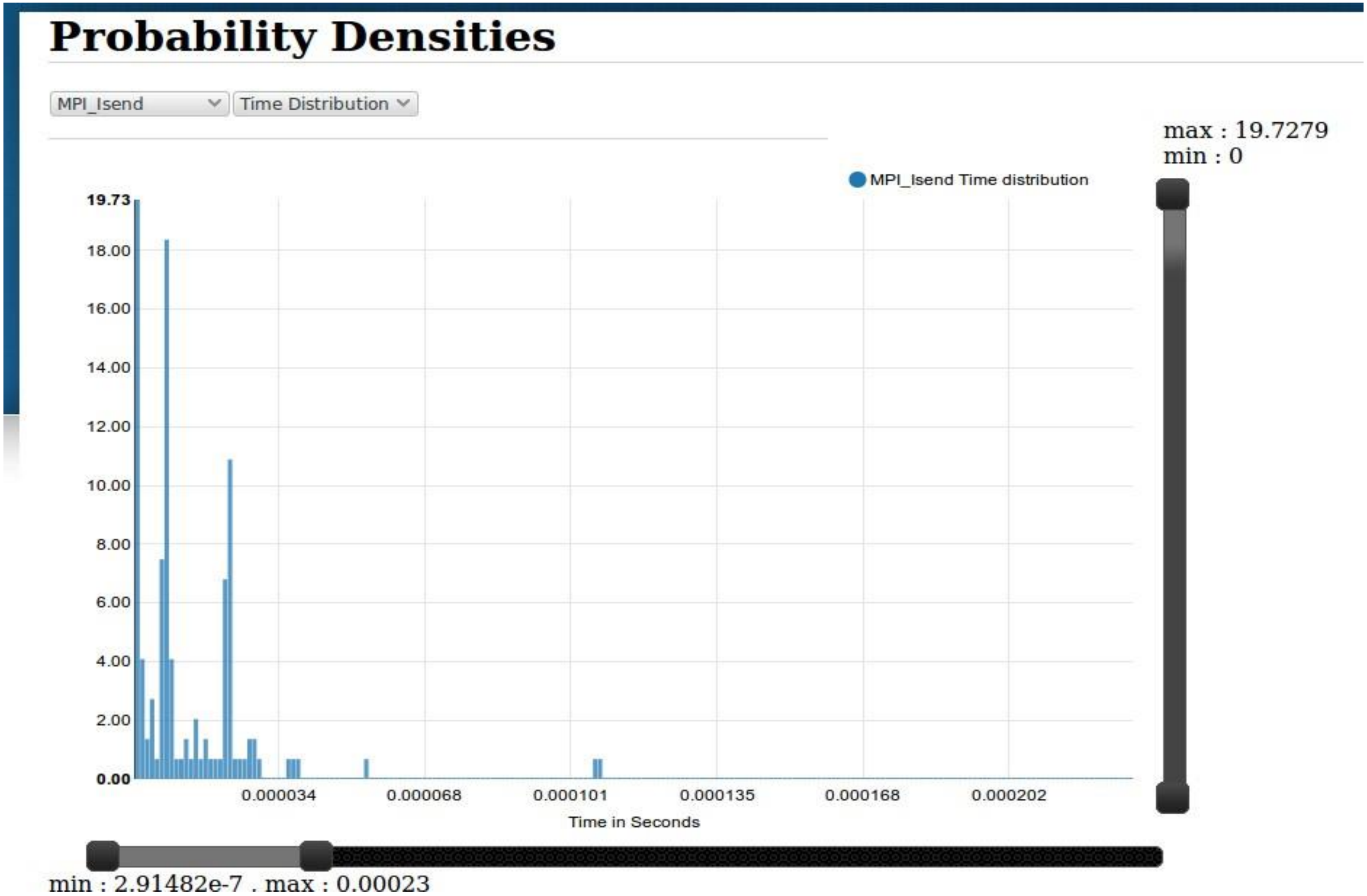
- Function scattering over time



Pinpointing hotspots

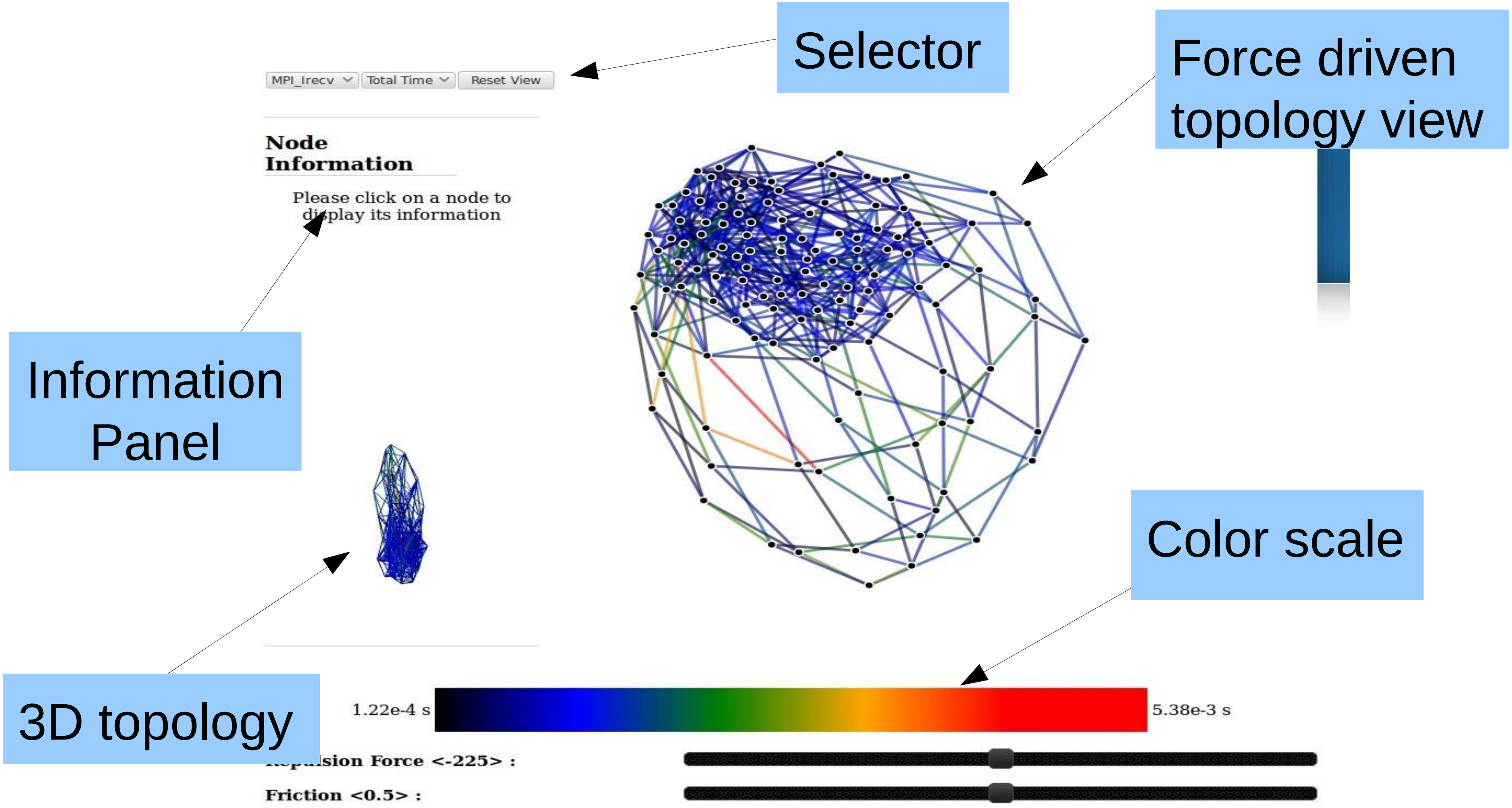
MPI characterization: analyses (3/5)

- Probability densities



Pinpointing hotspots
MPI characterization: analyses (4/5)

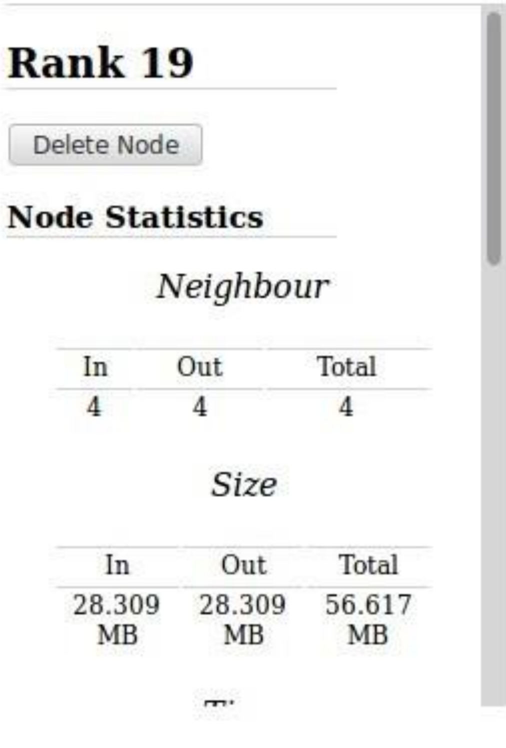
- Topology view (1/2)



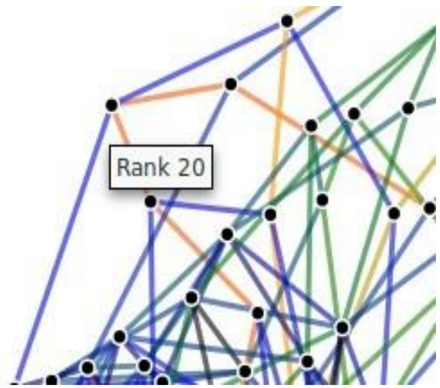
Pinpointing hotspots

MPI characterization: analyses (4/5)

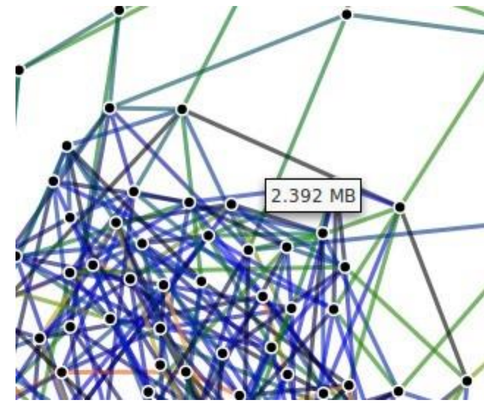
- Topology view (2/2)



Click on a node in the force layout to display its information

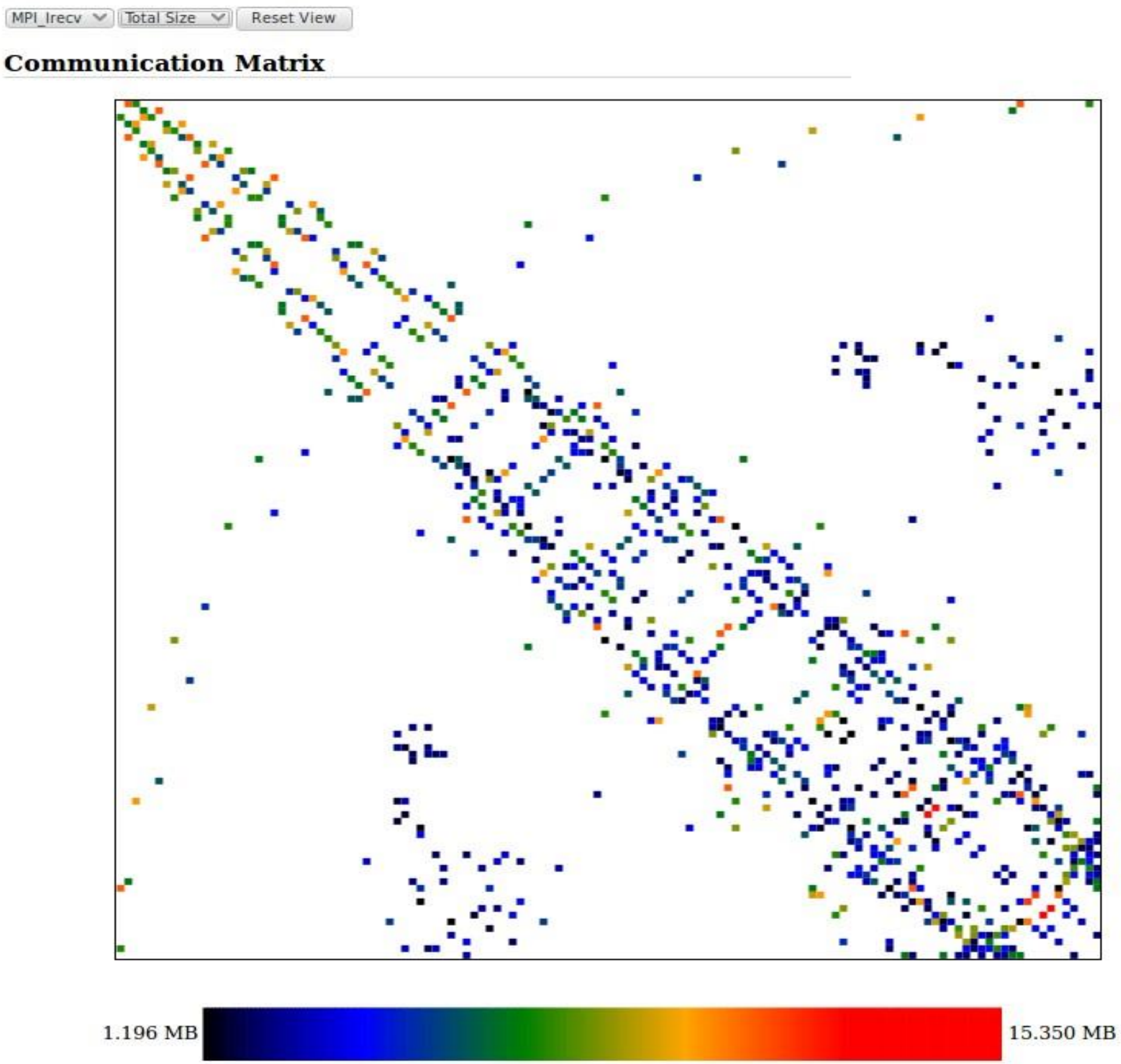


Hover a node to see its MPI rank

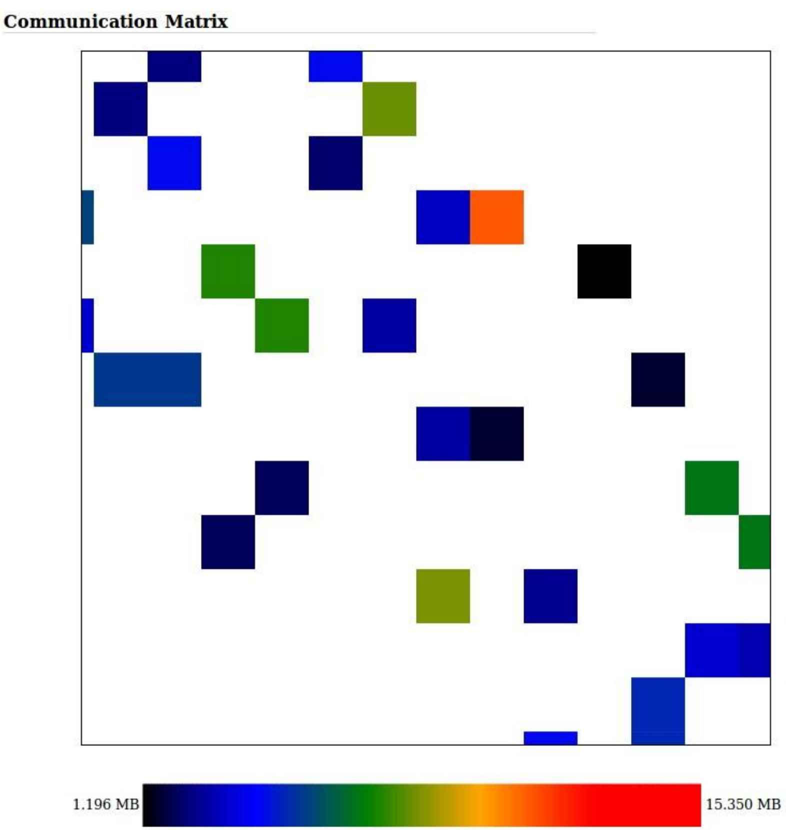


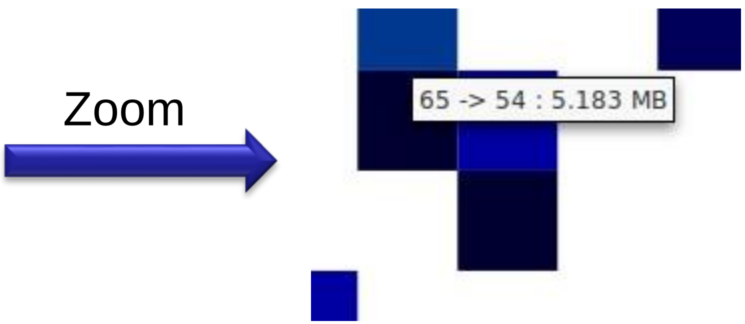
Hover an edge to see its value

- Communication matrix (1/3)



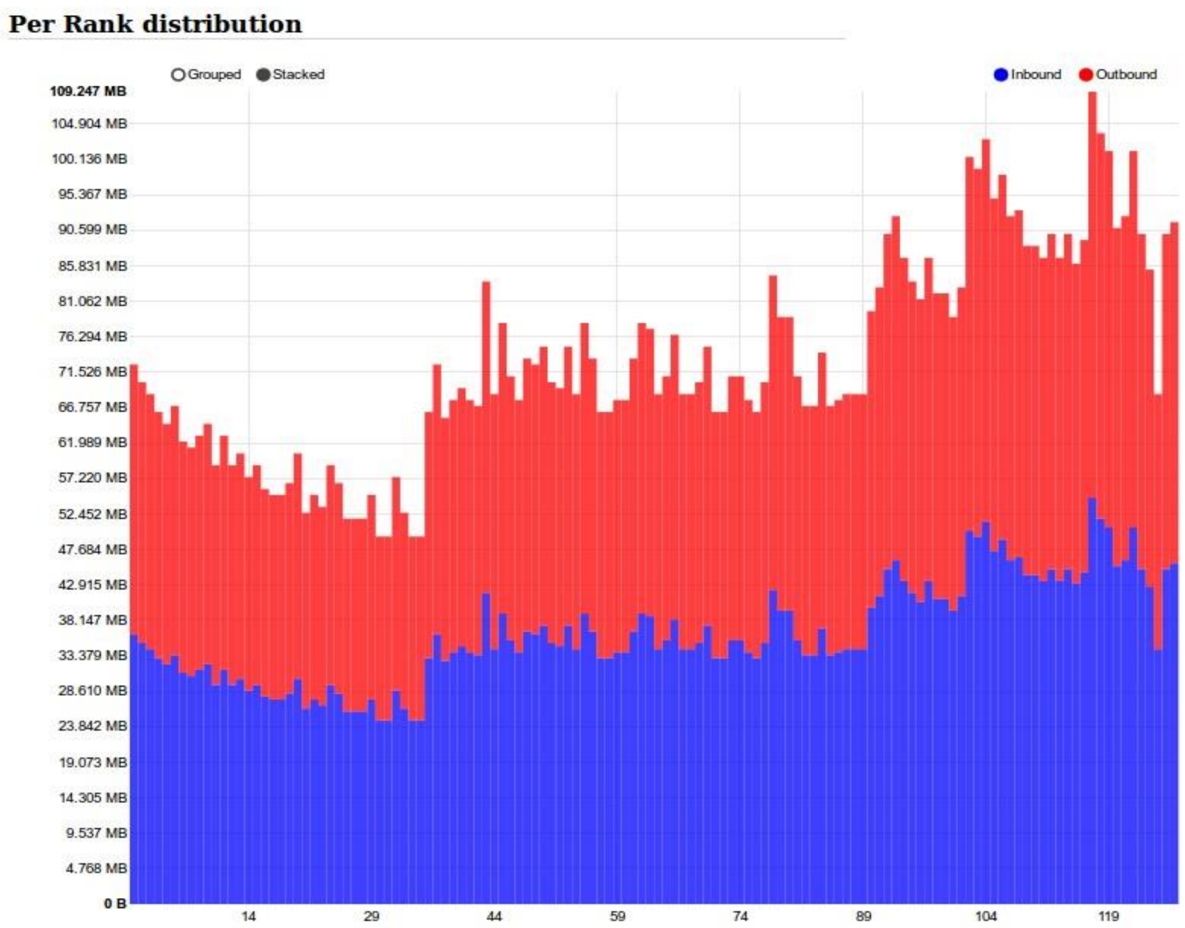
➢ Communication matrix (2/3)





Pinpointing hotspots
MPI characterization: analyses (5/5)

- Communication matrix (3/3)



Pinpointing hotspots
MPI characterization: next steps

- Fine grained analyses should be:
  - Investigated using (MPI) tracing tools
  - And filtering on specific processes/events of interest detected thanks to this tool

Outline

- Introduction
- Pinpointing hotspots
  - Functions, loops
  - MPI characterization
- Code quality analysis

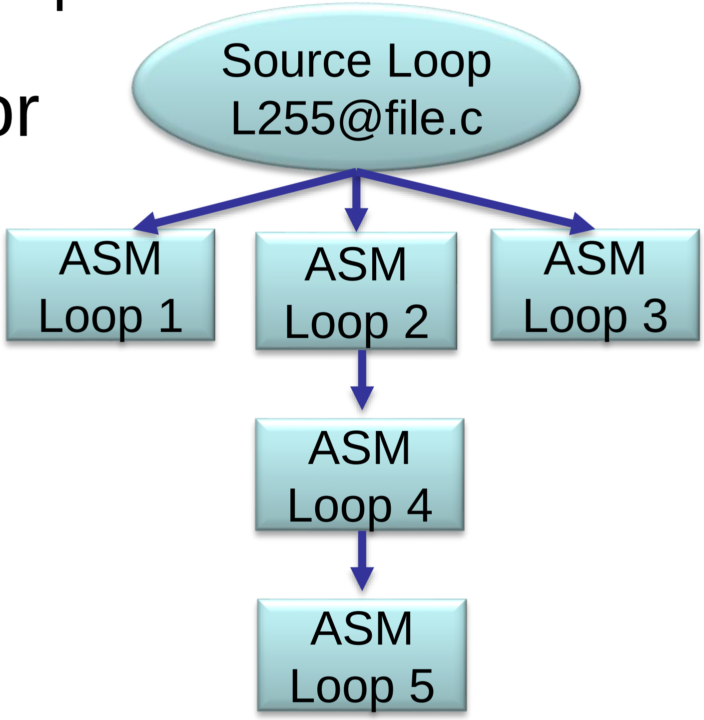
Static performance modeling
Introduction

- Main performance issues:
  - Core level
  - Multicore interactions
  - Communications

- Most of the time core level is forgotten

Static performance modeling
Goals

- Static performance model
  - Targets innermost loops
    - source loop V.S. assembly loop
  - Take into account processor (micro)architecture
  - Assess code quality
    - Estimate performance
    - Degree of vectorization
    - Impact on micro architecture



Static performance modeling

Model

- Simulates the target (micro)architecture
  - Instructions description (latency, uops dispatch...)
  - Machine model
- For a given binary and micro-architecture, provides
  - Quality metrics (how well the binary is fitted to the micro architecture)
  - Static performance (lower bounds on cycles)
  - Hints and workarounds to improve static performance

Static performance modeling

Metrics

- Vectorization (ratio and speedup)
  - Allows to predict vectorization (if possible) speedup and increase vectorization ratio if it’s worth
- High latency instructions (division/square root)
  - Allows to use less precise but faster instructions like RCP (1/x) and RSQRT (1/sqrt(x))
- Unrolling (unroll factor detection)
  - Allows to statically predict performance for different unroll factors (find main loops)

Static performance modeling

Output example (1/2)

MAQAO

Code quality analysis

▾ Source loop ending at line 682

▾ MAQAO binary loop id: 238

The loop is defined in MPI/BT/x_solve.f:519-682

15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain | Potential gain | Hints | Experts only

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization.
By fully vectorizing your loop, you can lower the cost of an iteration from 190.00 to 60.75 cycles (3.13x speedup).

Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:
- Try another compiler or update/tune your current one:
- Remove inter-iterations dependences from your loop and make it unit-stride.

Bottlenecks

By removing all these bottlenecks, you can lower the cost of an iteration from 190.00 to 143.00 cycles (1.33x speedup).

▸ Source loop ending at line 734

Static performance modeling
Output example (2/2)

MAQAO

Code quality analysis

Source loop ending at line 682

MAQAO binary loop id: 238

The loop is defined in MPI/BT/x_solve.f:519-682

15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain | Potential gain | **Hints** | Experts only

Type of elements and instruction set

234 SSE or AVX instructions are processing arithmetic or math operations on double precision FP elements in scalar mode (one at a time).

Vectorization status

Your loop is probably not vectorized (store and arithmetical SSE/AVX instructions are used in scalar mode and, for others, at least one is in vector mode).
Only 28% of vector length is used.

Matching between your loop (in the source code) and the binary loop

The binary loop is composed of 234 FP arithmetical operations:
- 95: addition or subtraction
- 139: multiply

The binary loop is loading 1600 bytes (200 double precision FP elements).
The binary loop is storing 616 bytes (77 double precision FP elements).

Arithmetic intensity

Arithmetic intensity is 0.11 FP operations per loaded or stored byte.

Thanks for your attention !

Questions ?