Performance Analysis and Optimization Tool

Andres S. CHARIF-RUBIAL
Jean-Baptiste BESNARD
{andres.charif,jean-baptiste.besnard}@uvsq.fr

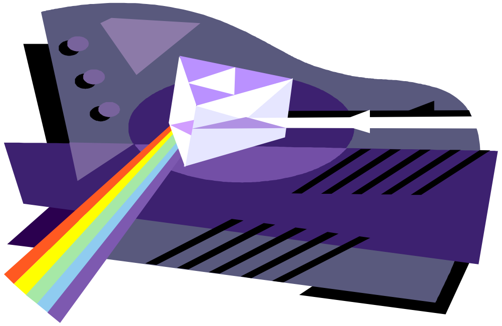
Performance Analysis Team, University of Versailles
http://www.maqao.org

Performance evaluation team

Main activities

- Develop performance analysis and optimisation 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 ?



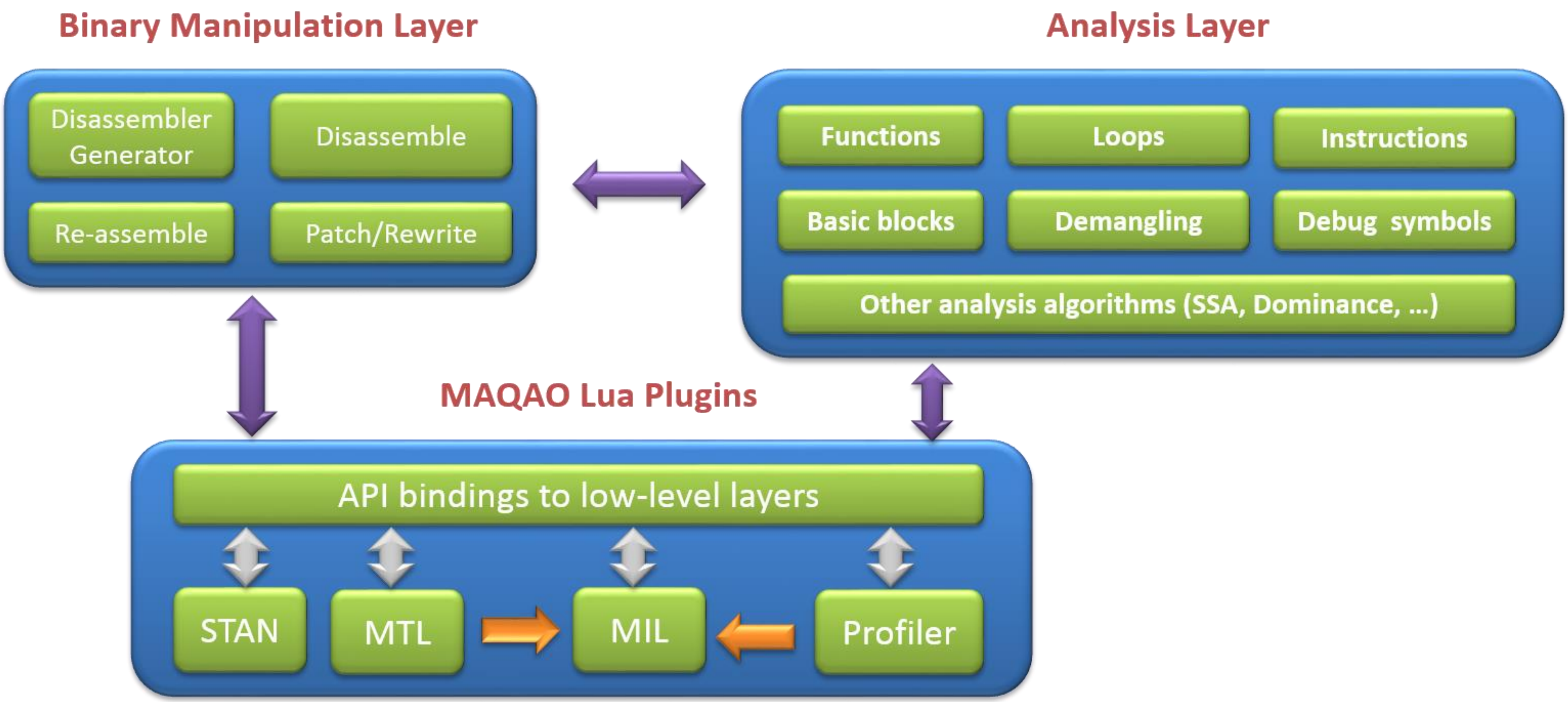
- Generally a multifaceted problem
  - Maximizing the number of views = better understand
- Use techniques and tools to understand issues
- Once understood → Optimize application

- 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

- Open source (LGPL 3.0)
  - Currently binary release
  - Source release by end February
- Special version version for this workshop:
  - Including the MALP tool (see LICENSE file)
  - Special license
- 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)

Introduction
MAQAO Tool



- 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

Introduction

Methodology – Performance analysis and Optimization

- 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/

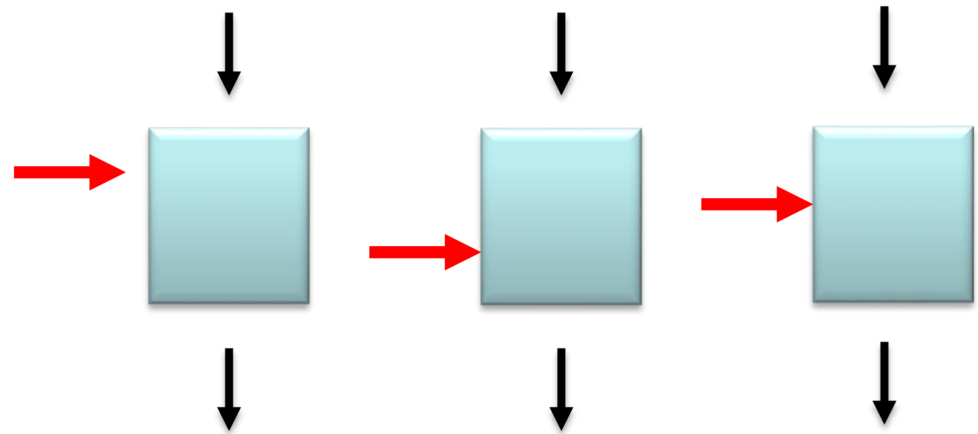
Outline

- Introduction
- Pinpointing hotspots
- Code quality analysis
- MPI Chaterization

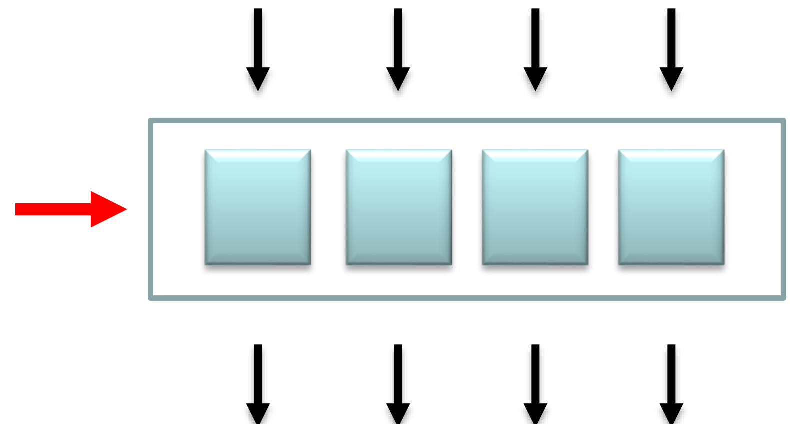
- MAQAO Profiling
  - Instrumentation
    - Through binary rewriting
    - High overhead / More precision
  - Sampling
    - Hardware counter through perf_event_open system call
    - Very low overhead / less details

Pinpointing hotspots
Parallelism level

- SPMD
  - Program level



- SIMD
  - Instruction level



- By default MAQAO only considers system processes and threads

Pinpointing hotspots
Parallelism level

- 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

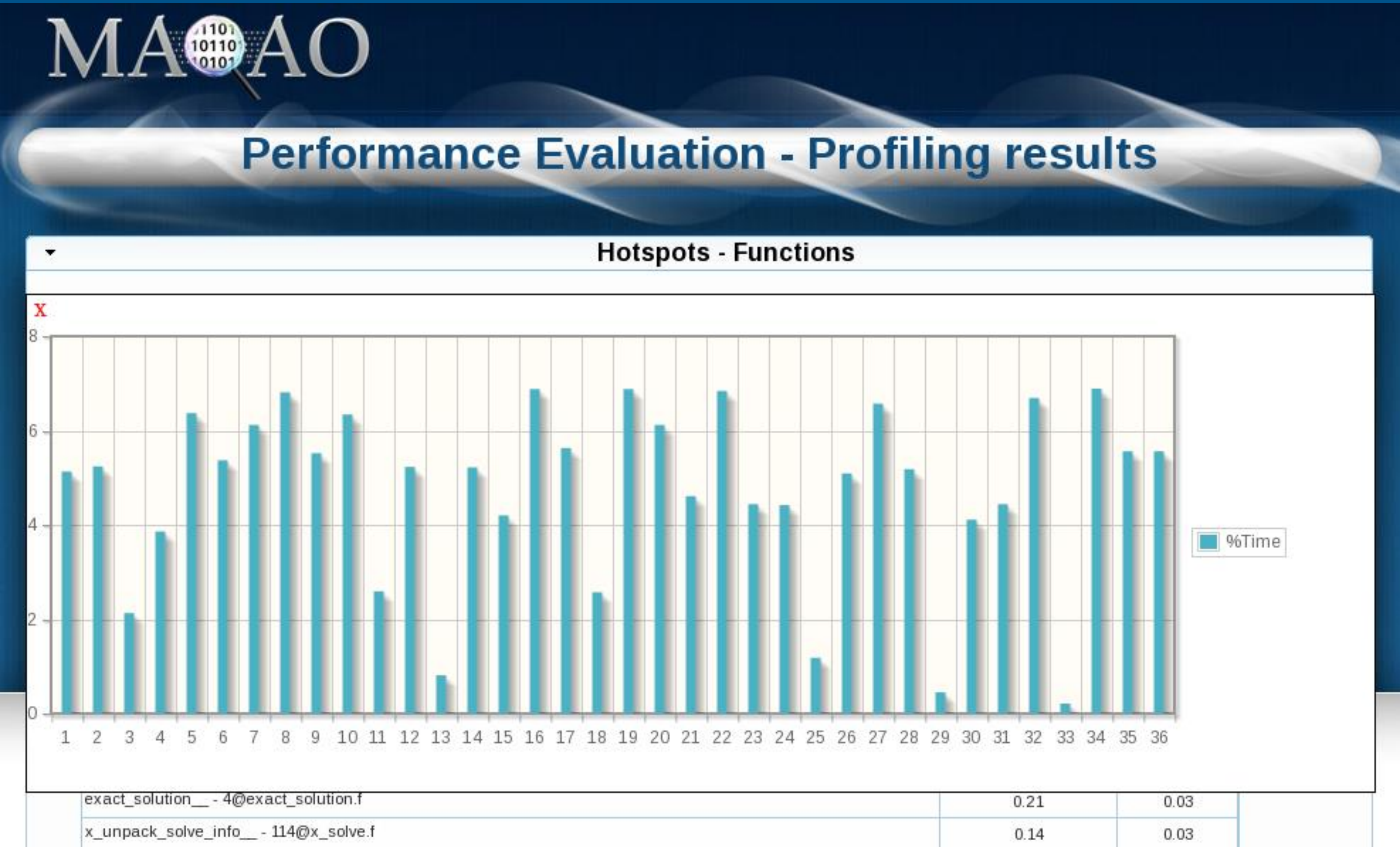
Performance Evaluation - Profiling results

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



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	

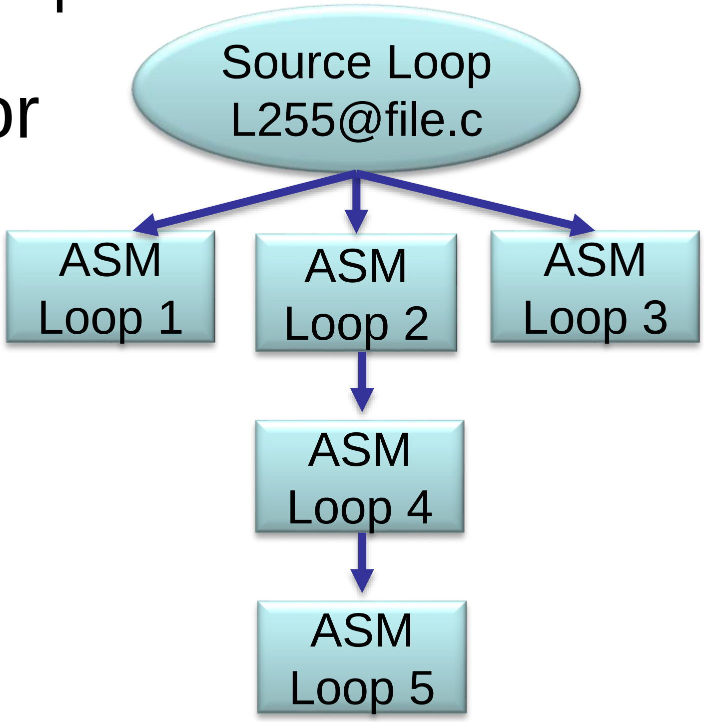
Outline

- Introduction
- Pinpointing hotspots
- Code quality analysis
- MPI Chaterization

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

- Most of the time core level is forgotten

- 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



Code quality analysis
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

Code quality analysis
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)

Code quality analysis

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

Code quality analysis
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.

Outline

- Introduction
- Pinpointing hotspots
- Code quality analysis
- **MPI Chaterization**

- Our methodology
  - Corse 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 ?

Multi-Application Online Profiling (MALP)
is an online MPI oriented profiling tool.

Part of the MAQAO tool-set as a binary module.

Online analysis

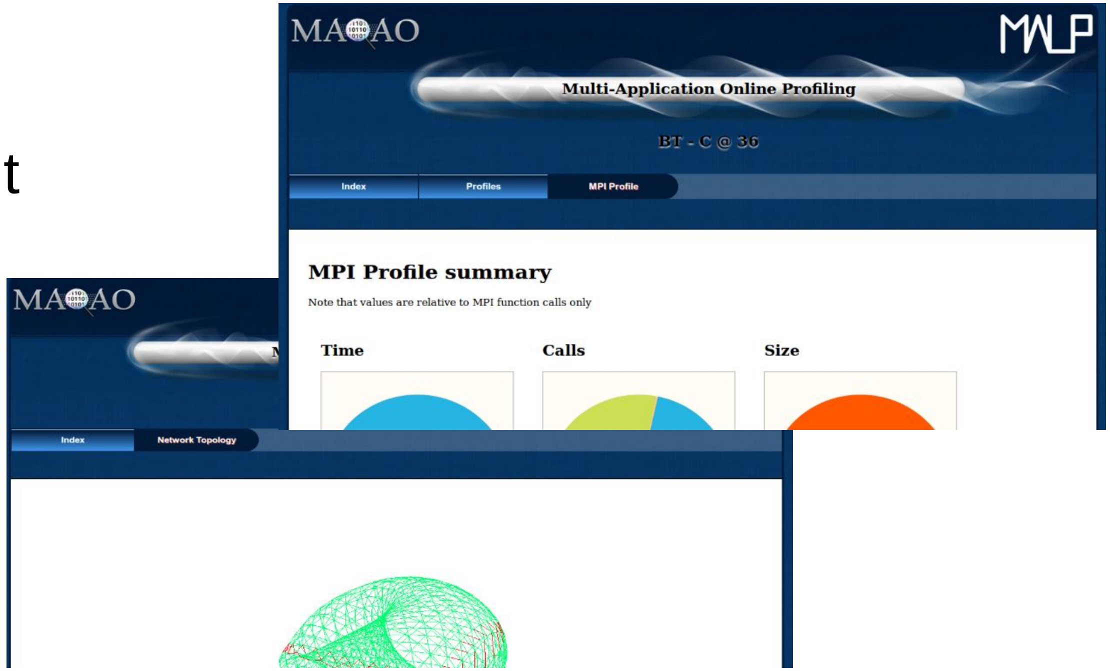
- No *Ios*
- Smaller memory footprint
- Pipelined analysis
- Better scalability

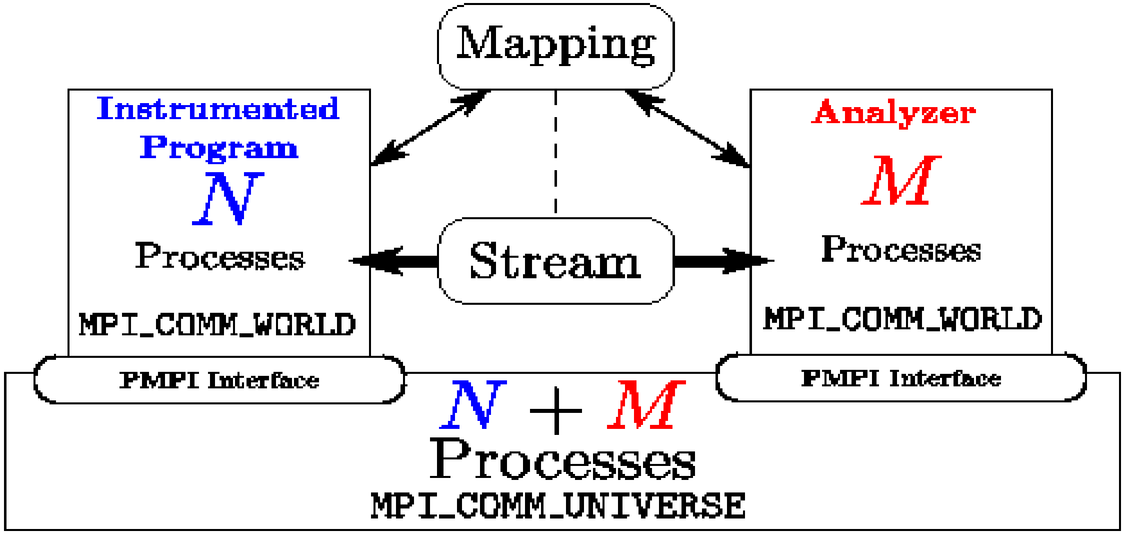
Instrumentation

- Full MPI interface
- Most 'POSIX' functions

Reporting

- Web-based frontend





Runtime coupling through MPI « virtualization »

- Gathers instrumentation and analysis in the same MPI instance.
- Takes advantage of high speed networks.
- Avoids the storage of large traces while preserving event ganularity

Better scalability (no IO contention) tested up to 16k cores

Suitable for very long runs (data are consumed on the fly)

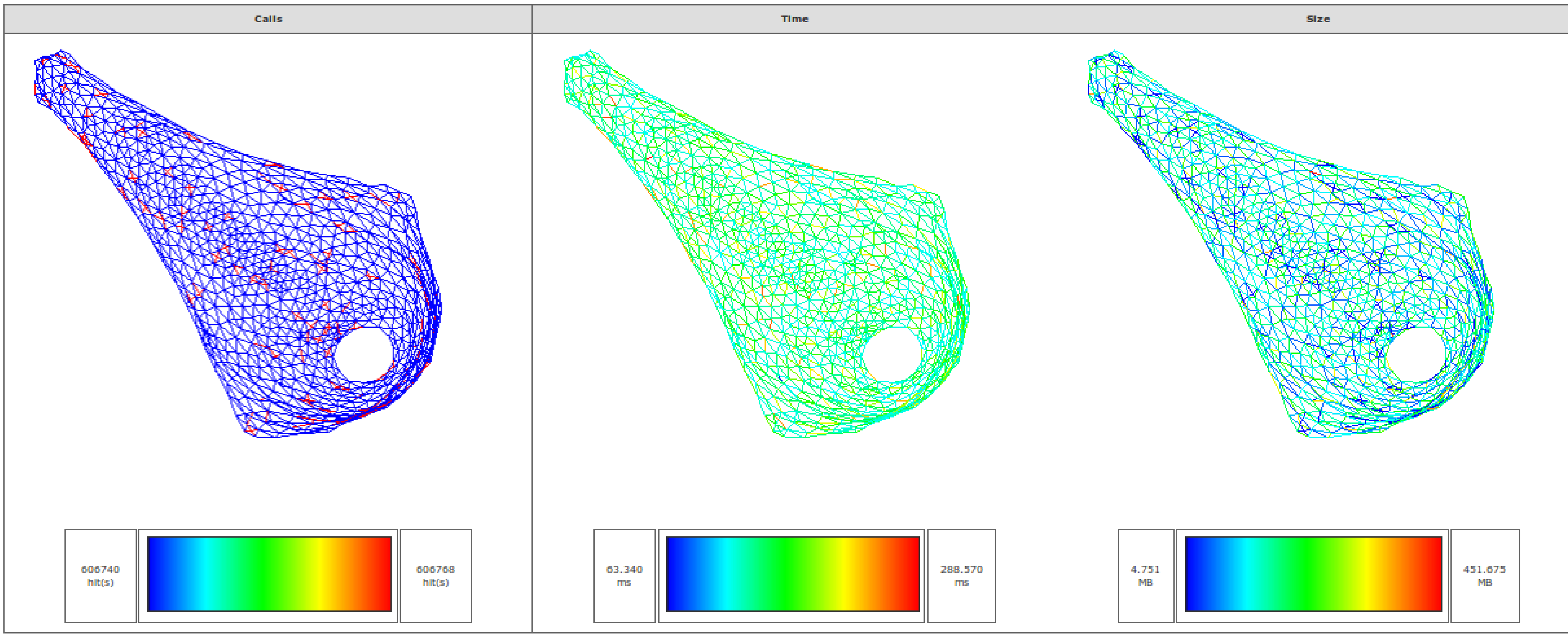
Lowers the time to report

IO delegation (when tracing)

Topological analysis

3D topology viewer

MPI_Irecv Topology

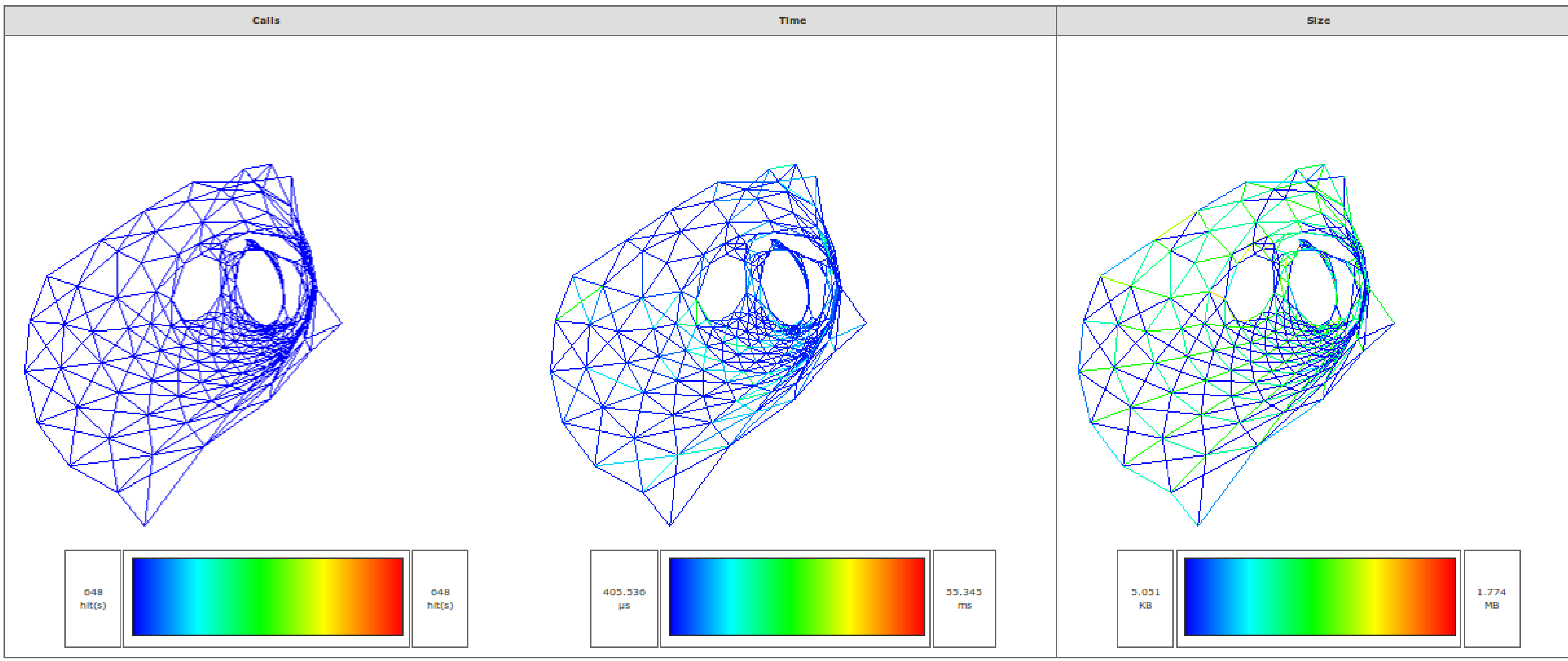


800 cores on a plane engine simulation

MPI Characterization at scale
Measurements

Topological analysis
3D topology viewer

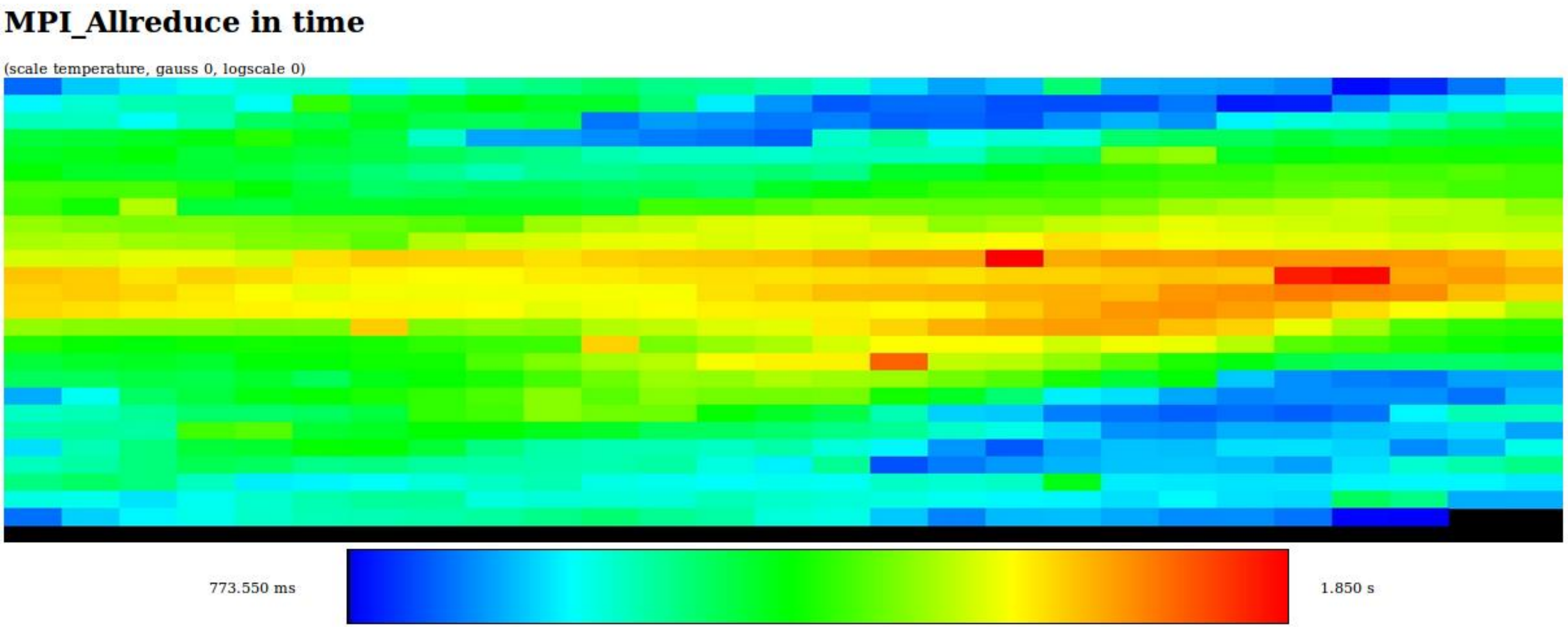
MPI_Isend Topology



CGPOP 230 cores.

Spatial analysis

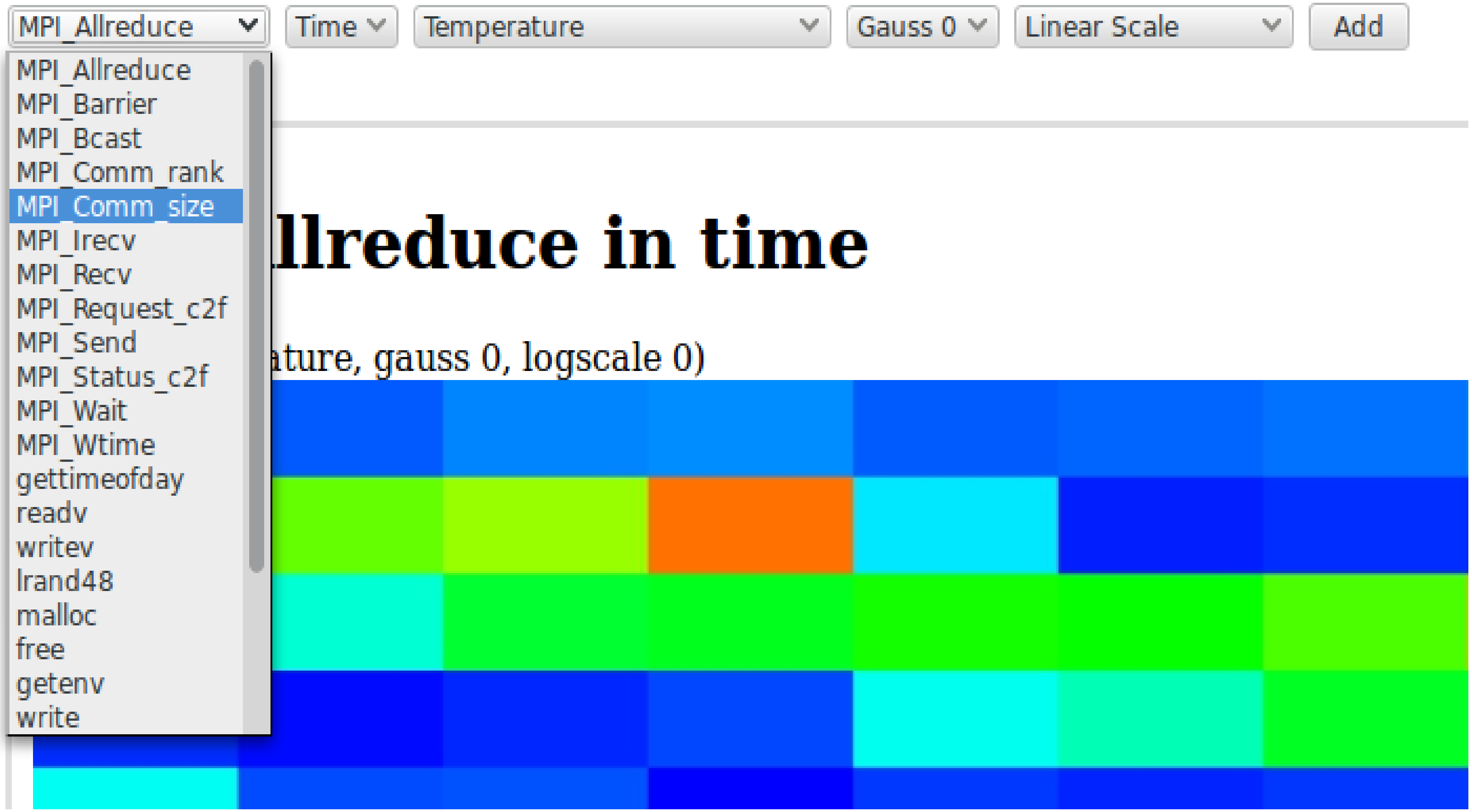
Spatial scattering



700 cores on a Magneto Hydro-Dynamic (MHD) application

Spatial analysis

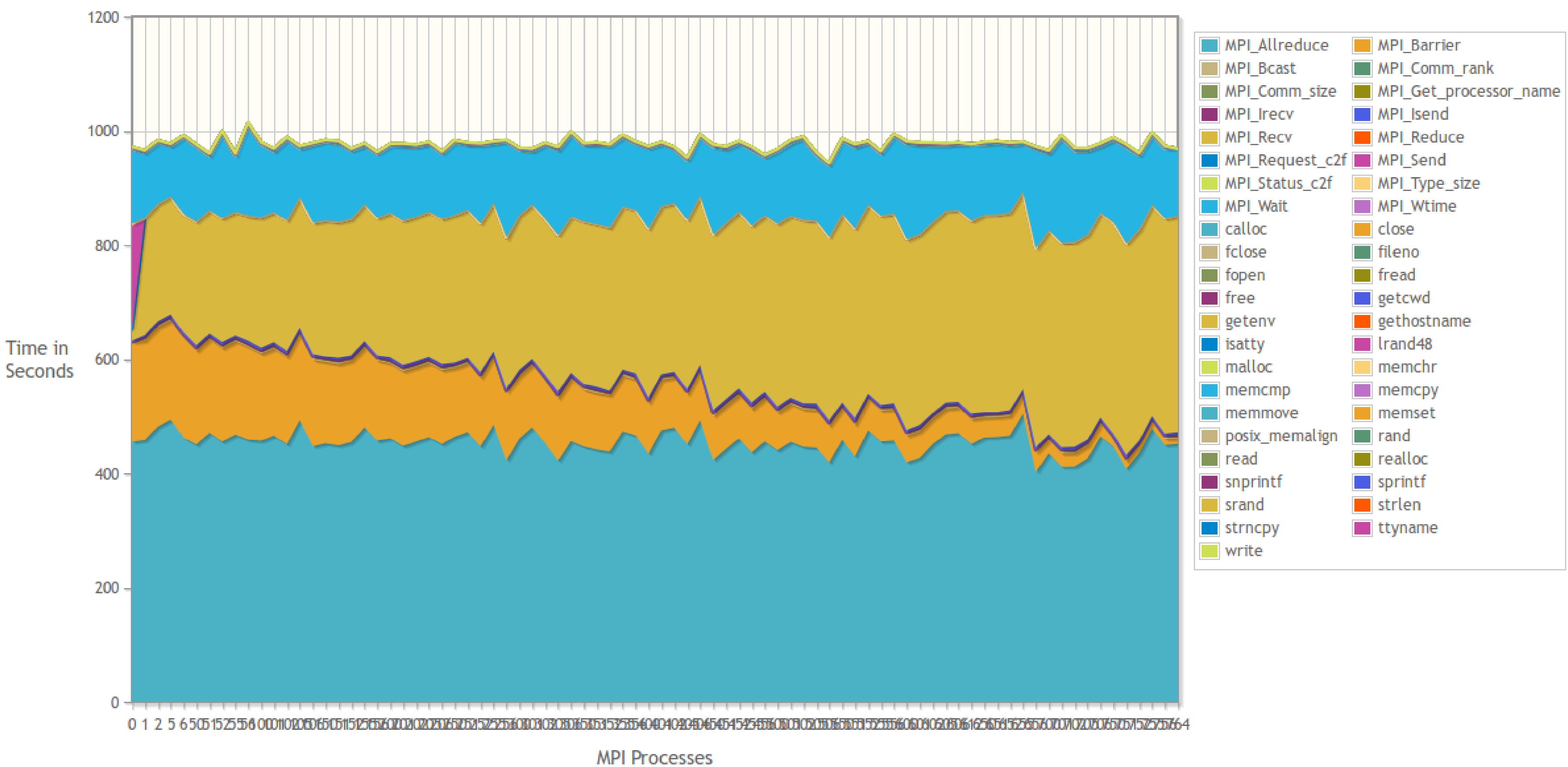
Available for every instrumented calls



Balancing analysis

Function costs over processes (800 cores engine simulation)

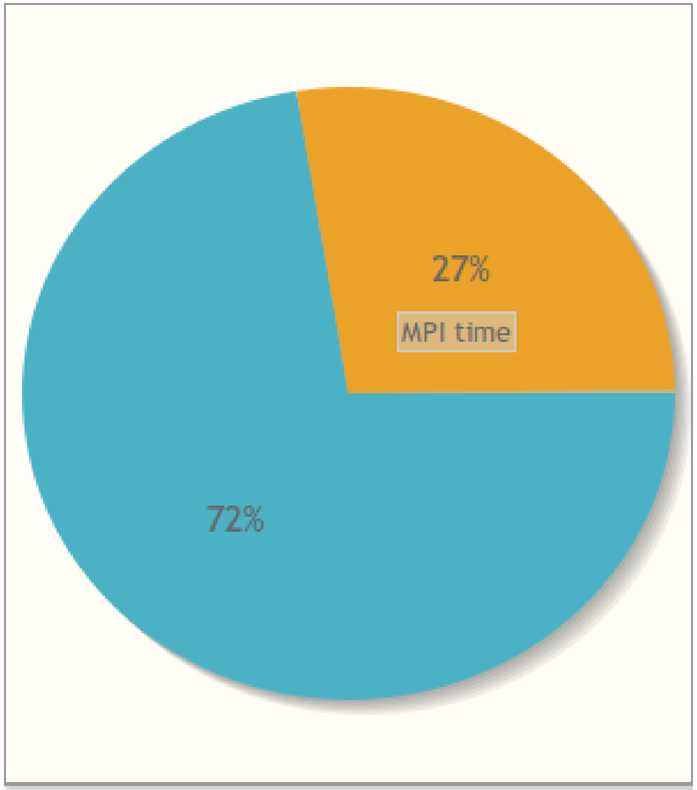
Time Breakdown over Functions



Profiles

Time breakdown (categories)

- MPI
- Posix
- Others (Wall-time - (MPI+POSIX))



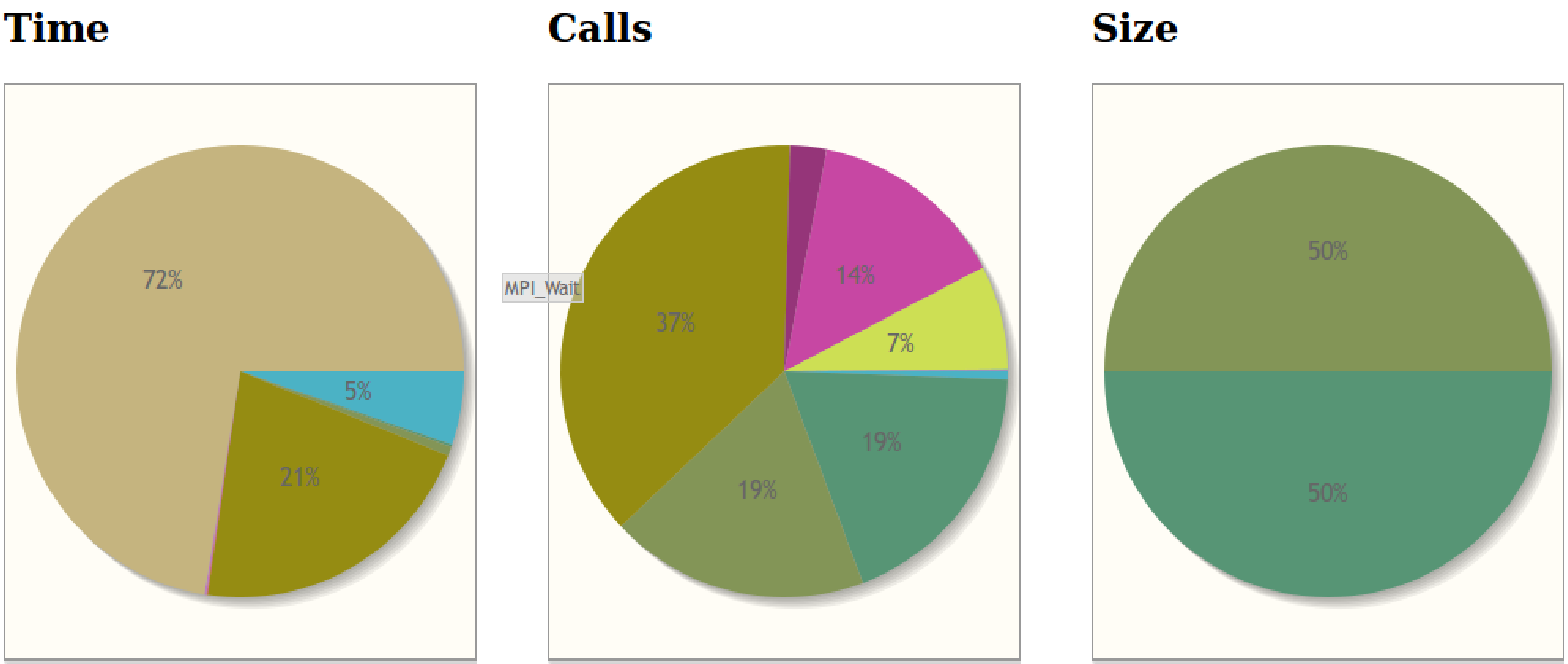
Describes immediatelly what takes time in the application.

MPI bound ?
System bound ?
Other ?

Event Type	Total time	%
Other Calls time	41 m 36.090 s	72.445
MPI time	15 m 41.784 s	27.334
POSIX time	7.604 s	0.221

MHD app (256 cores)

Profiles
Global view (functions)



Function names can be queried by hovering with the mouse.

MHD app (256 cores)

MPI Characterization at scale

Measurements

Profiles

Detailed view

Functions are sorted in decreasing time order.

« Other calls » appears in red.
It includes the actual computation.

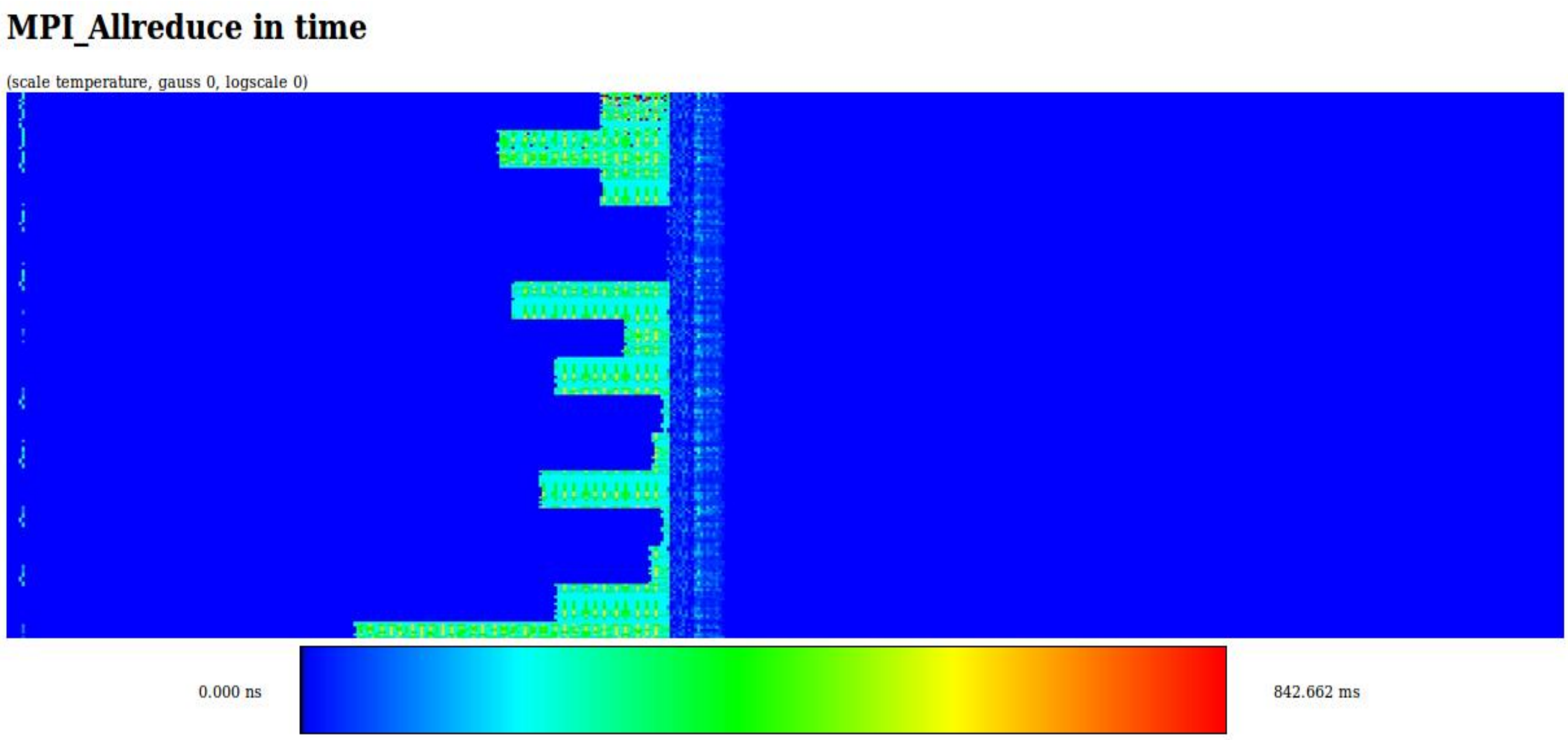
Profiles and pie charts can also be generated for MPI or POSIX calls only.

Function	Calls	Time	%	Size
Other Calls	0	41 m 36.090 s	72.445	0 B
MPI_Wait	20103168	12 m 12.745 s	21.267	0 B
MPI_Allreduce	311040	3 m 1.196 s	5.259	2.386 MB
MPI_Isend	10051584	22.067 s	0.640	113.330 GB
MPI_Irecv	10051584	5.775 s	0.168	113.330 GB
write	34321	3.739 s	0.109	0 B
malloc	7715083	1.998 s	0.058	0 B
mkdir	256	958.680 ms	0.028	0 B
free	4006924	564.599 ms	0.016	0 B
gettimeofday	1414851	320.094 ms	0.009	0 B
writev	505	14.623 ms	0.000	0 B
sbrk	2579	4.906 ms	0.000	0 B
memmove	5120	1.857 ms	0.000	0 B
strlen	34441	1.178 ms	0.000	0 B
readv	814	1.151 ms	0.000	0 B
MPI_Comm_rank	256	1.014 ms	0.000	0 B
sysconf	180	200.309 μs	0.000	0 B
lrand48	240	87.212 μs	0.000	0 B
MPI_Comm_size	256	78.357 μs	0.000	0 B

MHD app (256 cores)

Temporal analysis

Projects metrics over processes and time.



CGPOP (230 cores) overscaled on a 48 core problem

Ongoing work

- Dynamic bottleneck analyzer
  - Differential analysis
- Memory characterization tool
  - Access patterns
  - Data reshaping
  - Memory allocations tracing
  - Cache simulator
- Value profiler
  - Function specialization / memorizing
  - Loops instances (iteration count) variations

Thanks for your attention !

Questions ?