



MAQAO

Performance Analysis and Optimization Tool

Andres S. CHARIF-RUBIAL

andres.charif@uvsq.fr

Performance Evaluation Team, University of Versailles S-Q-Y

<http://www.maqao.org>

VI-HPS 17th Stuttgart – 23/27 February 2015

MAQAO Framework and Toolsuite

R&D Team: develop performance evaluation and optimization tools



Open Source software (LGPL 3)

- Currently only binary release (source => ongoing)
- Profilers (generic and MPI) work on any LSB/Most Unix
- Code quality analysis and hardware counters support only available for Intel x86-64 and Xeon Phi



Funded by UVSQ, Intel and CEA (French department of energy)

Establish partnerships:

- Optimize industrial applications
- Provide building blocks (framework services) to other tools:
 - TAU tool *tau_rewrite*: binary rewriting feature (MIL)
 - ATOS/BULL tool *bullxprof* : binary rewriting feature (MIL)

Introduction

Performance analysis (1/2)

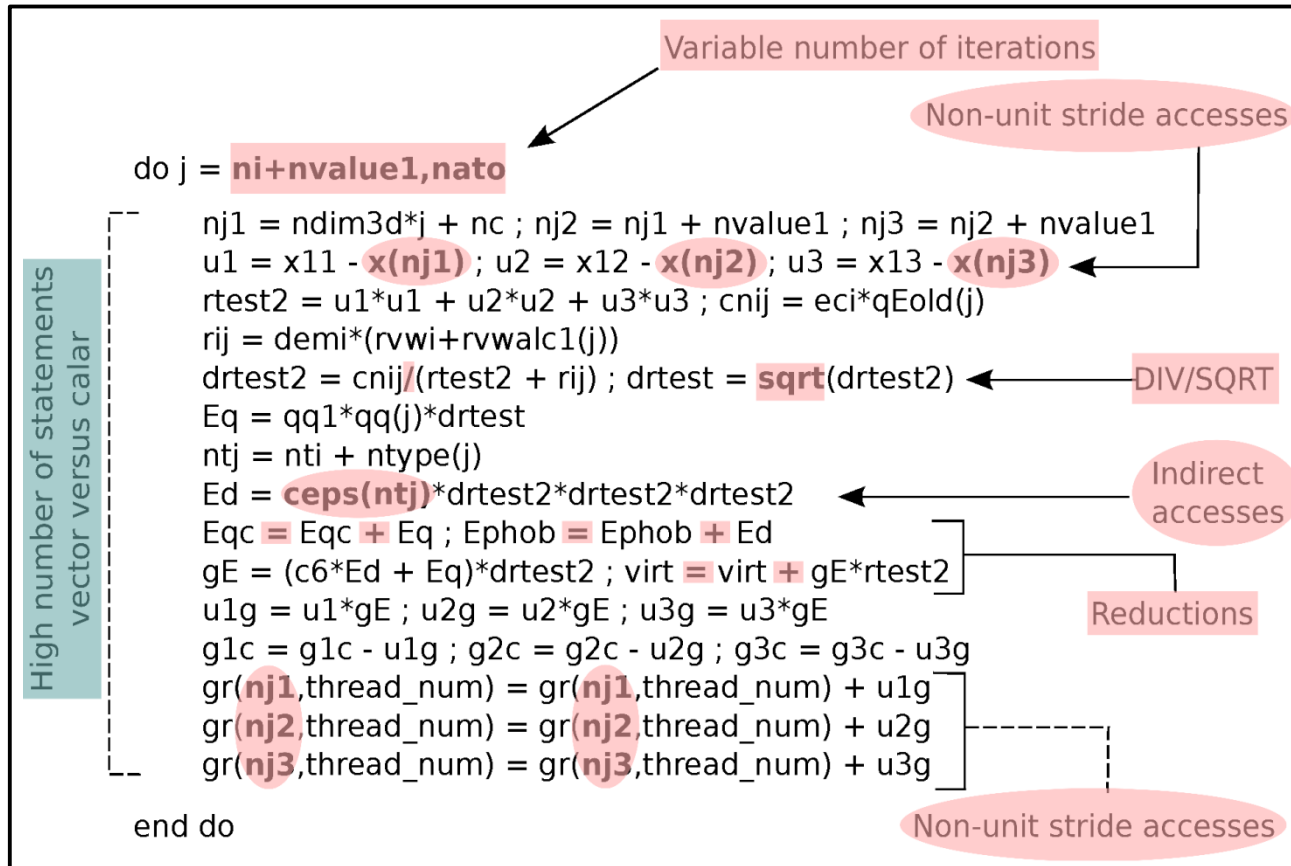
- Characterize the performance of an application
 - Complex multicore CPUs and memory systems
 - How well does it behaves on a given machine
- Generally a multifaceted problem
 - What are the issues (numerous but finite) ?
 - Which one(s) dominates ?
 - Maximizing the number of views
 - => Need for specialized tools
- Three main classes of issues
 - Find/Select relevant algorithms
 - Work sharing/decomposition
 - Exploiting performance available at CPU level



Introduction

Performance analysis (2/2)

Motivating example: loop ~10% walltime



Source code and associated issues:

- 1) High number of statements
- 2) Non-unit stride accesses
- 3) Indirect accesses
- 4) DIV/SQRT
- 5) Reductions
- 6) Vector vs Scalar

Introduction

MAQAO: working at binary level (1/2)

Why ???

Most of the time the compiler changes source code

Some source code instrumentation may prevent the compiler from applying transformation

- i.e.: loop interchange

We want to evaluate the “real” executed code

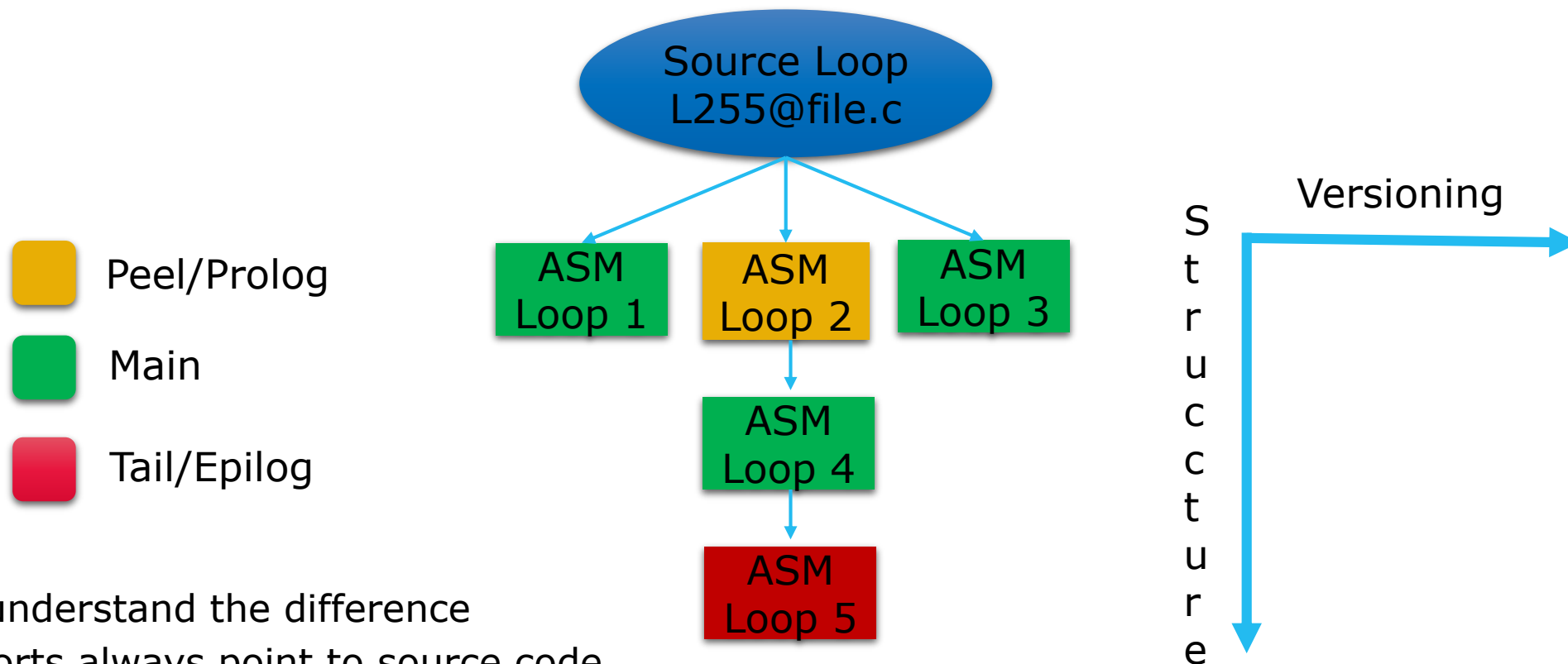
We are able to reconstruct an abstract view with functions and loops in order to be able to correlate with your source code.

One little difference is understanding loops at assembly level

Introduction

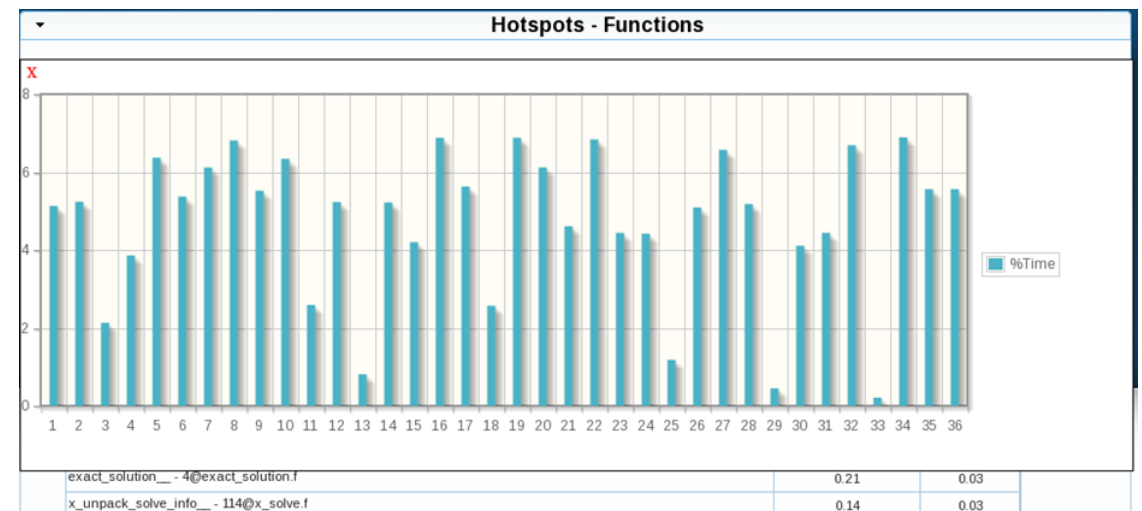
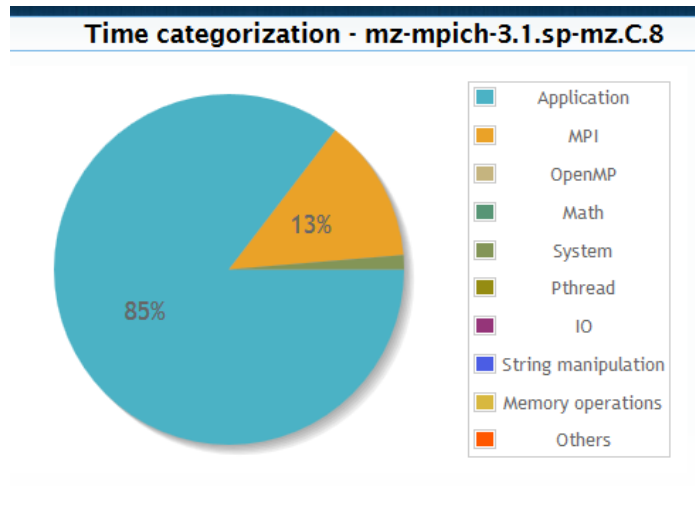
MAQAO: working at binary level (2/2)

Source level V.S. Assembly level



You just need to understand the difference
But our tools' reports always point to source code

MAQAO Perf: locating hotspots



MAQAO Perf: locating hotspots

Introduction

Locating most time consuming hotspots is the first step you want to accomplish.

Multiple measurement methods available:

- Why is it important to know this ?
- Instrumentation
 - Through binary rewriting
 - High overhead / More precision
- Sampling
 - Hardware counters (through `perf_event_open` system call)
 - Linux kernel timers
 - No instrumentation / Very low overhead / less details (i.e. function calls count)
- Default method: Sampling using hardware counters (if available) or timers

Runtime-agnostic: Only system processes and threads are considered

Where is time spent ? Which one(s) should I investigate first ?

MAQAO Perf: locating hotspots

Time categorization

Sadly, executing an application is not just doing the science you are supposed to !

Work sharing/splitting

- Shared: Pthreads, OpenMP, etc ...
- Distributed: MPI, etc...

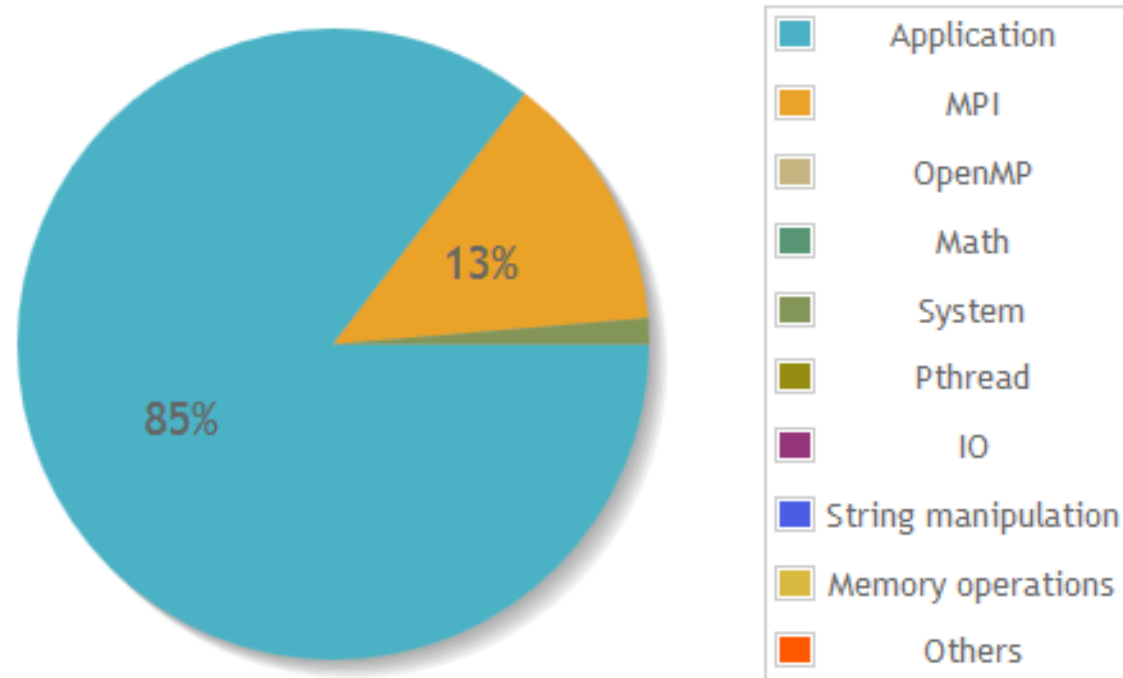
Programming

- IO
- String manipulation
- Memory management
- Math (external libraries)

Doing actual science (Application)

- Functions
- Loops

Time categorization - mz-mpich-3.1.sp-mz.C.8



MAQAO Perf: locating hotspots

Function and loop hotspots (1/3)

Lets focus on science !

First we want to check function hotspots load balancing vue at (multi)node level

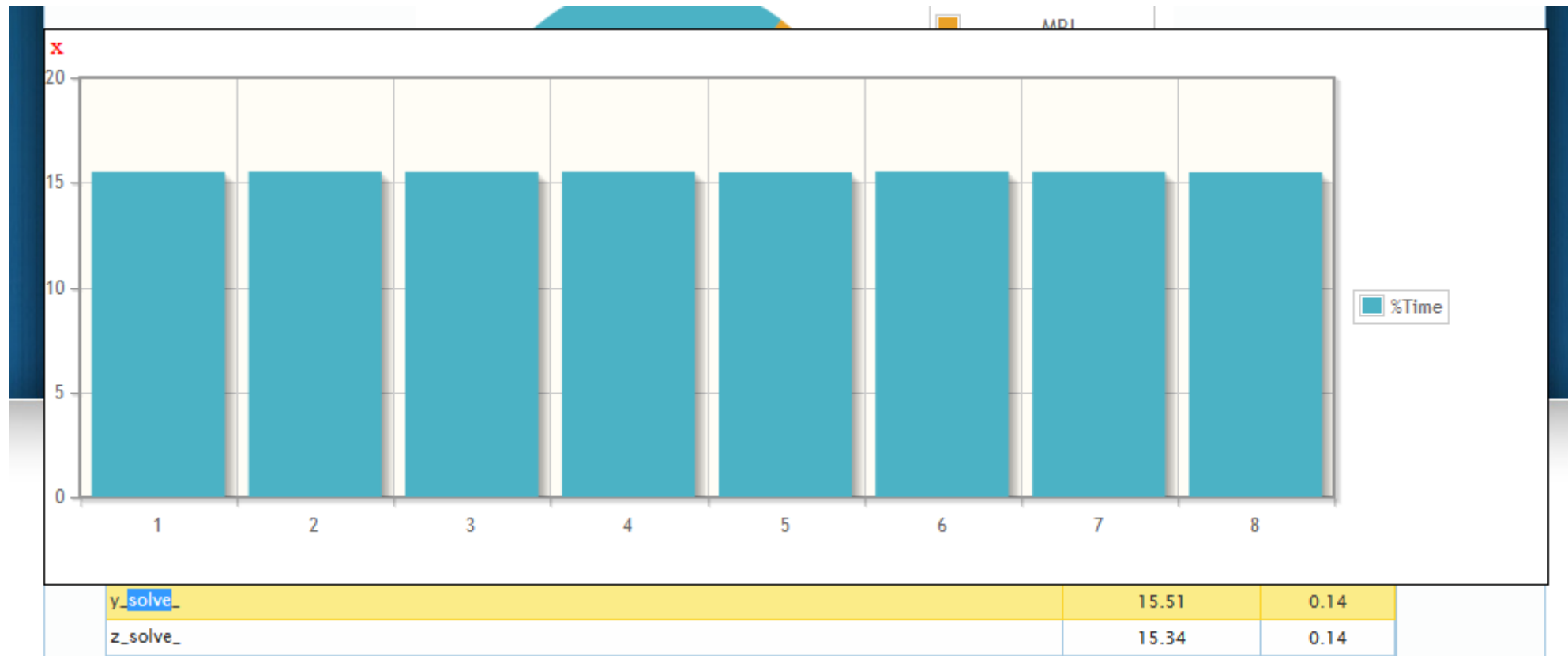
- For the same function
- Does it behave the same way on all the nodes ?

Hotspots - Functions

| Name | Median Excl %Time | Deviation |
|---------------------|-------------------|-----------|
| compute_rhs_ | 30.88 | 0.14 |
| y_solve_ | 15.51 | 0.14 |
| z_solve_ | 15.34 | 0.14 |
| x_solve_ | 15.07 | 0.14 |
| MPIDI_CH3I_Progress | 5.61 | 0.14 |

MAQAO Perf: locating hotspots

Function and loop hotspots (2/3)



MAQAO Perf: locating hotspots

Function and loop hotspots (3/3)

Then analyse time spent in loops:

- Time spent in loop w.r.t. function

- Use MAQAO CQA tool to analyse loops of interest

| dauvergne - Process #14213 - Thread #14201 | | | |
|--|------------|---------------|--|
| Name | Excl %Time | Excl Time (s) | |
| binvcrhs - 206@solve_subs.f | 17.27 | 2.23 | |
| MPIDI_CH3I_Progress | 15.24 | 1.96 | |
| poll_active_fboxes | 13.71 | 1.77 | |
| ▼ y_solve_omp_fn.0 - 45@y_solve.f | 8.47 | 1.09 | |
| ▼ loops | 8.47 | | |
| ▼ Loop 121 - y_solve.f@45 | 0 | | |
| ▼ Loop 122 - y_solve.f@45 | 0.16 | | |
| ○ Loop 124 - y_solve.f@45 | 0.14 | | |
| ○ Loop 125 - y_solve.f@145 | 5.12 | | |
| ○ Loop 126 - y_solve.f@55 | 2.03 | | |
| ○ Loop 123 - y_solve.f@45 | 1.02 | | |
| ▼ x_solve_omp_fn.0 - 48@x_solve.f | 8.23 | 1.06 | |
| ▶ loops | 8.23 | | |

MAQAO Perf/MPI: MPI characterization



MAQAO Perf/MPI: MPI characterization

Introduction (1/2)

The previous profiler module only provided a global figure about time spent in the MPI runtime (X%)

We want the same kind of insight but dealing with MPI primitives

Our methodology:

- Coarse grain: overview, global trends/patterns => cheapest possible cost/overhead
- Fine grain: filtering precise issues => accept to pay higher cost/overhead if worth

Online profiling:

- No traces to void IO wall: no IOs (only one result file with pre-processed data)
- Avoid memory : reduced memory footprint thanks to aggregated metrics
- Scalable on 1000+ MPI processes

MAQAO Perf/MPI: MPI characterization

Introduction (2/2)

Summary: Perf/MPI is a simple MPI profiling tool targeting lightweight metrics which can be reduced online (no trace required).

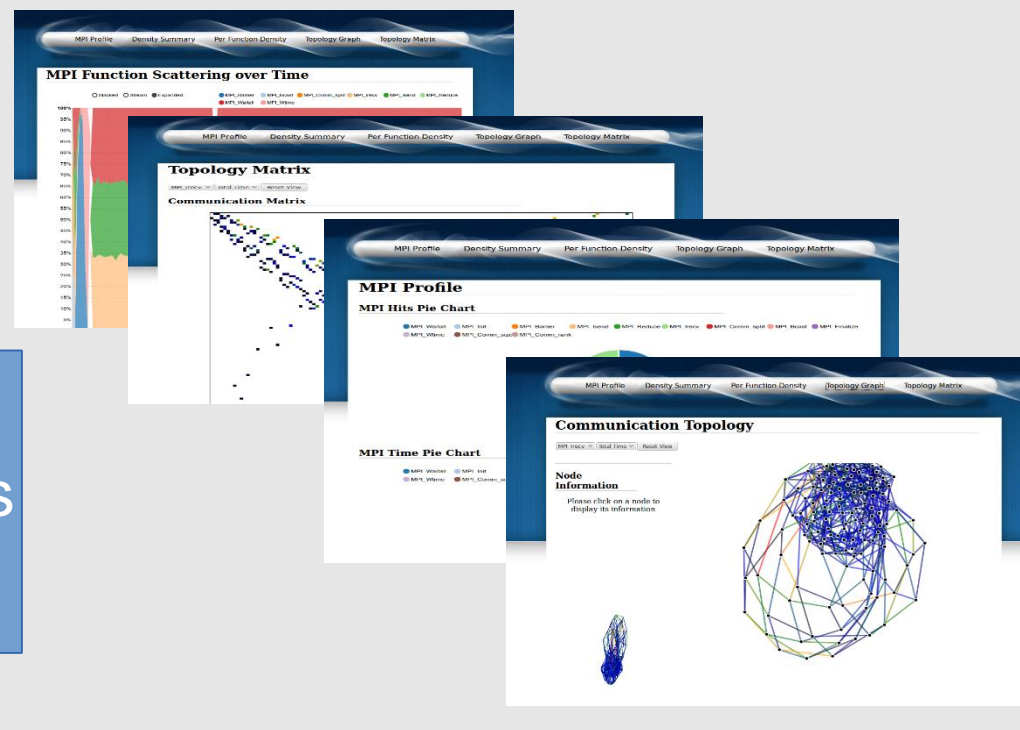
APPLICATION

MAQAO

profile.js

Does not require recompiling

In-browser Visualizer

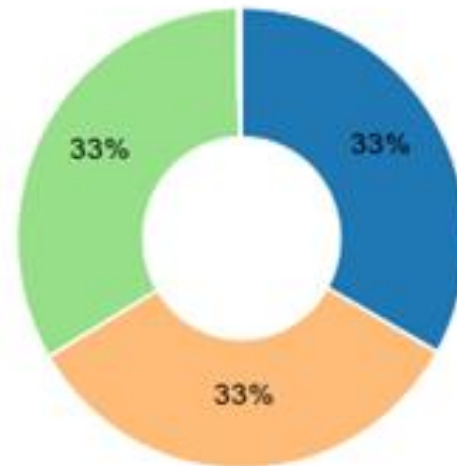
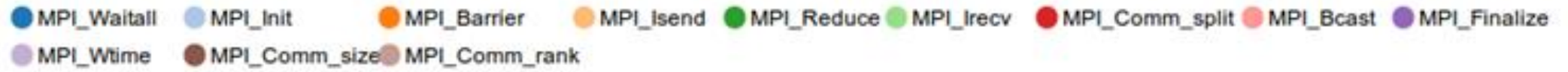


MAQAO Perf/MPI: MPI characterization

Global profile (1/3)

Summary vue: MPI primitives classified by hits (calls), time and size (if applicable)

MPI Hits Pie Chart

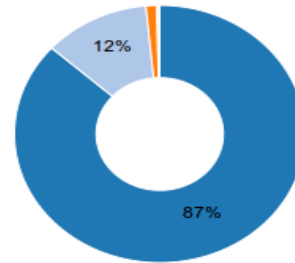


MAQAO Perf/MPI: MPI characterization

Global profile (2/3)

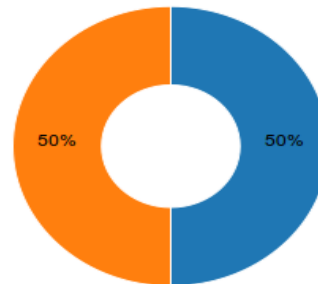
MPI Time Pie Chart

● MPI_Waitall ● MPI_Init ● MPI_Barrier ● MPI_Isend ● MPI_Reduce ● MPI_Irecv ● MPI_Comm_split ● MPI_Bcast ● MPI_Finalize
● MPI_Wtime ● MPI_Comm_size ● MPI_Comm_rank



MPI Size Pie Chart

● MPI_Isend ● MPI_Reduce ● MPI_Irecv ● MPI_Bcast



MAQAO Perf/MPI: MPI characterization

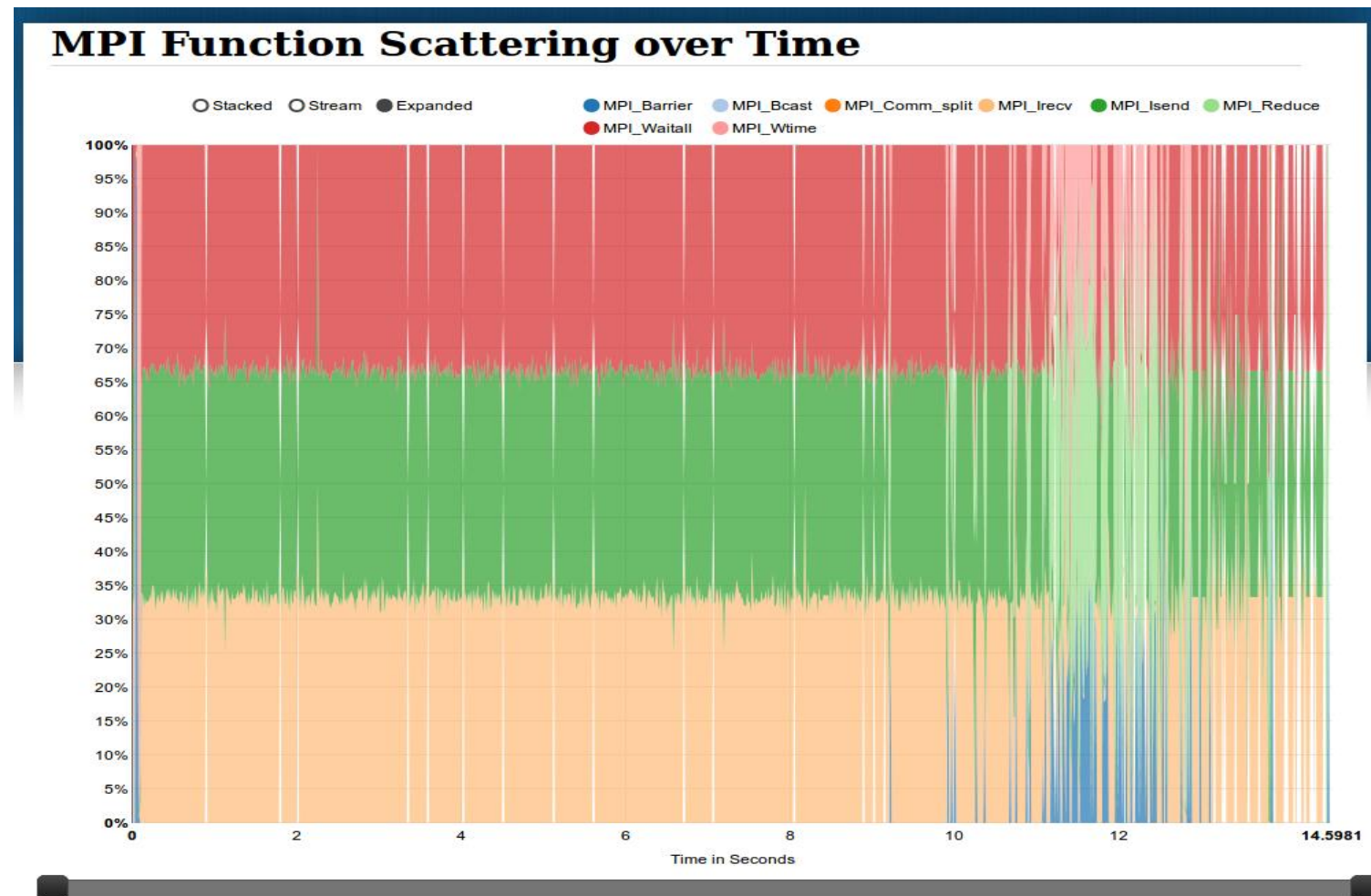
Global profile: flat vue (3/3)

MPI Profile

| Function | Hits | Time | Size | Walltime % |
|----------------|--------|-------------|------------|------------|
| MPI_Waitall | 192960 | 13 m 1.51 s | 0 B | 52.333% |
| MPI_Init | 128 | 1 m 46.60 s | 0 B | 7.138% |
| MPI_Barrier | 256 | 10.88 s | 0 B | 0.729% |
| MPI_Isend | 192960 | 1.47 s | 4.568 GB | 0.098% |
| MPI_Reduce | 384 | 5.36e-1 s | 11.000 KB | 0.036% |
| MPI_Irecv | 192960 | 4.62e-1 s | 4.568 GB | 0.031% |
| MPI_Comm_split | 128 | 4.05e-1 s | 0 B | 0.027% |
| MPI_Bcast | 1152 | 3.12e-2 s | 132.000 KB | 0.002% |
| MPI_Finalize | 128 | 2.07e-3 s | 0 B | 0.000% |
| MPI_Wtime | 256 | 3.53e-4 s | 0 B | 0.000% |
| MPI_Comm_size | 128 | 1.30e-4 s | 0 B | 0.000% |
| MPI_Comm_rank | 256 | 4.28e-5 s | 0 B | 0.000% |

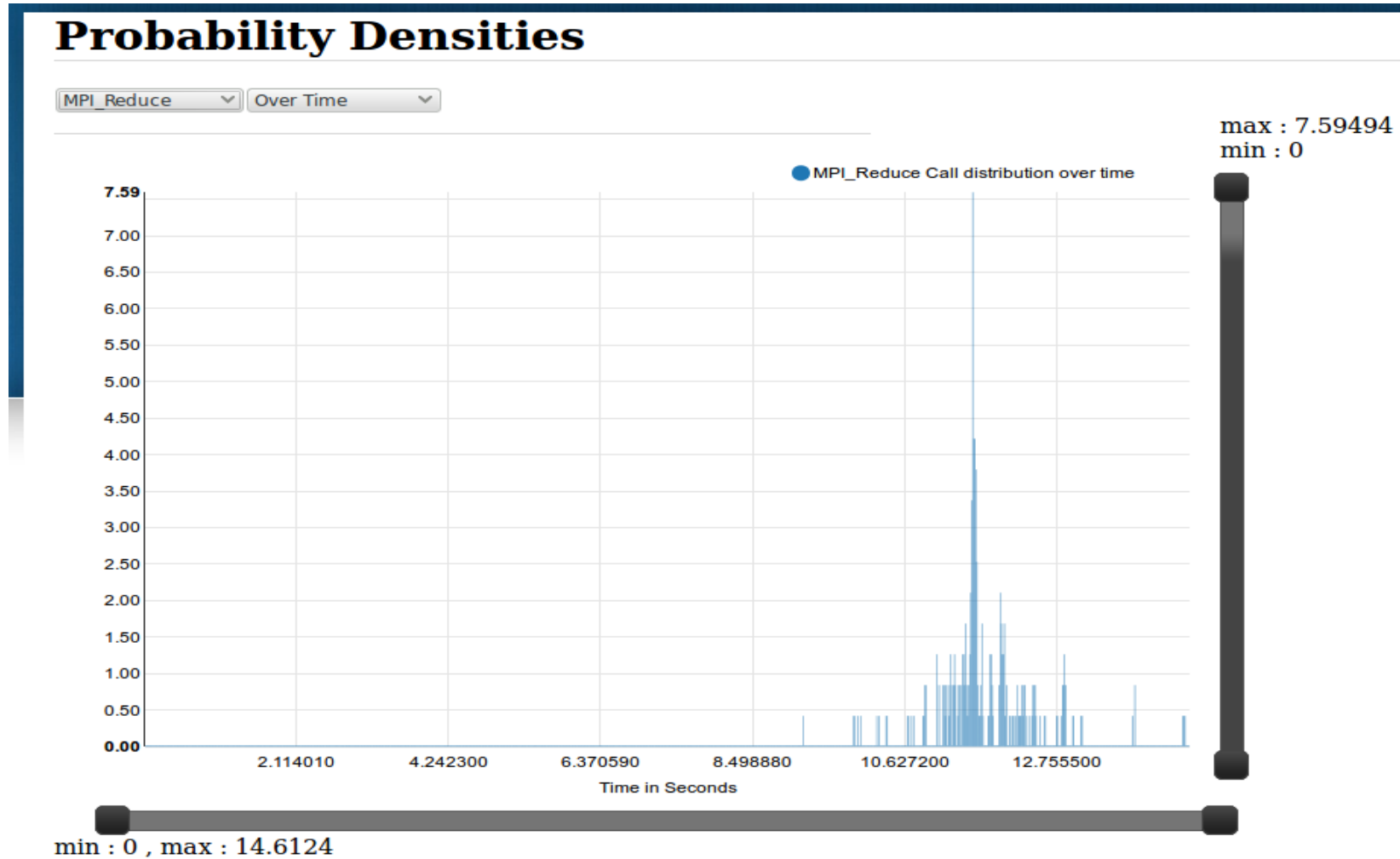
MAQAO Perf/MPI: MPI characterization

Function scattering over time



MAQAO Perf/MPI: MPI characterization

Probability densities: when and how long ?



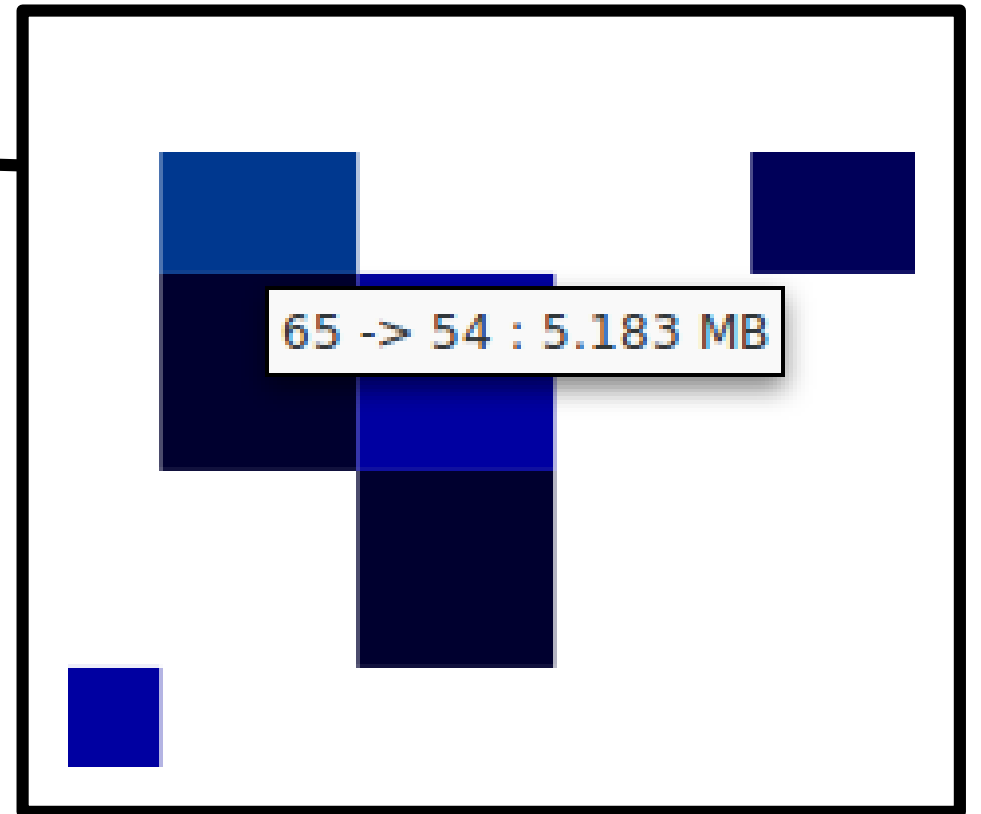
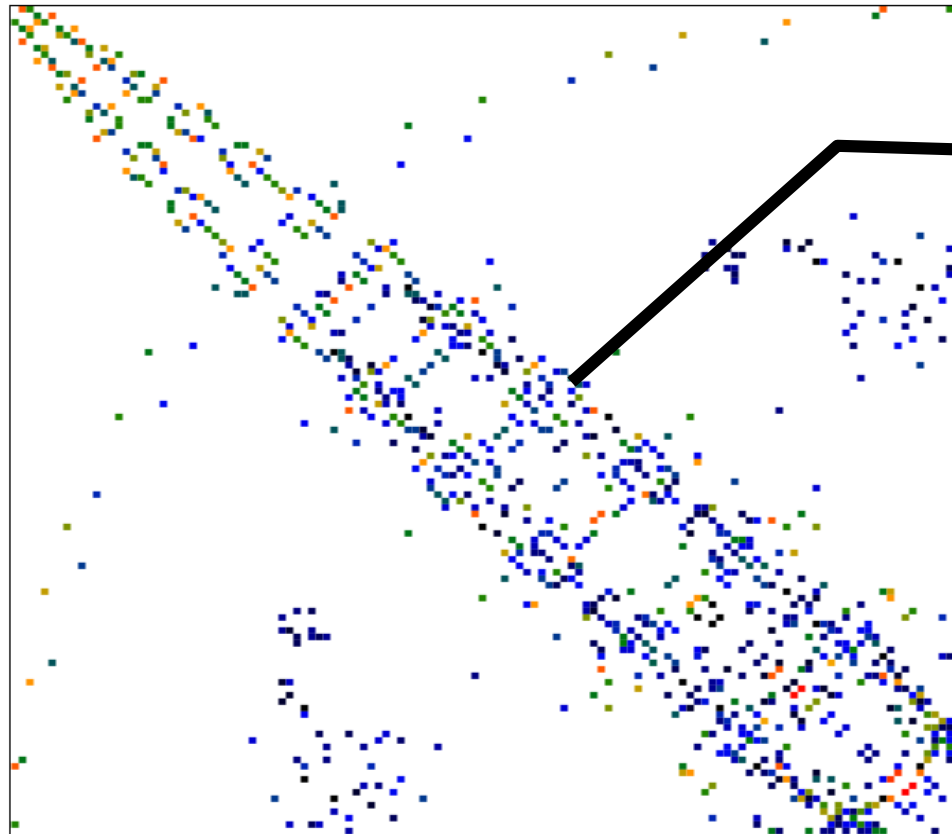
MAQAO Perf/MPI: MPI characterization

2D communication matrix

Hit, time, size

MPI Irecv Total Size Reset View

Communication Matrix



1.196 MB

15.350 MB

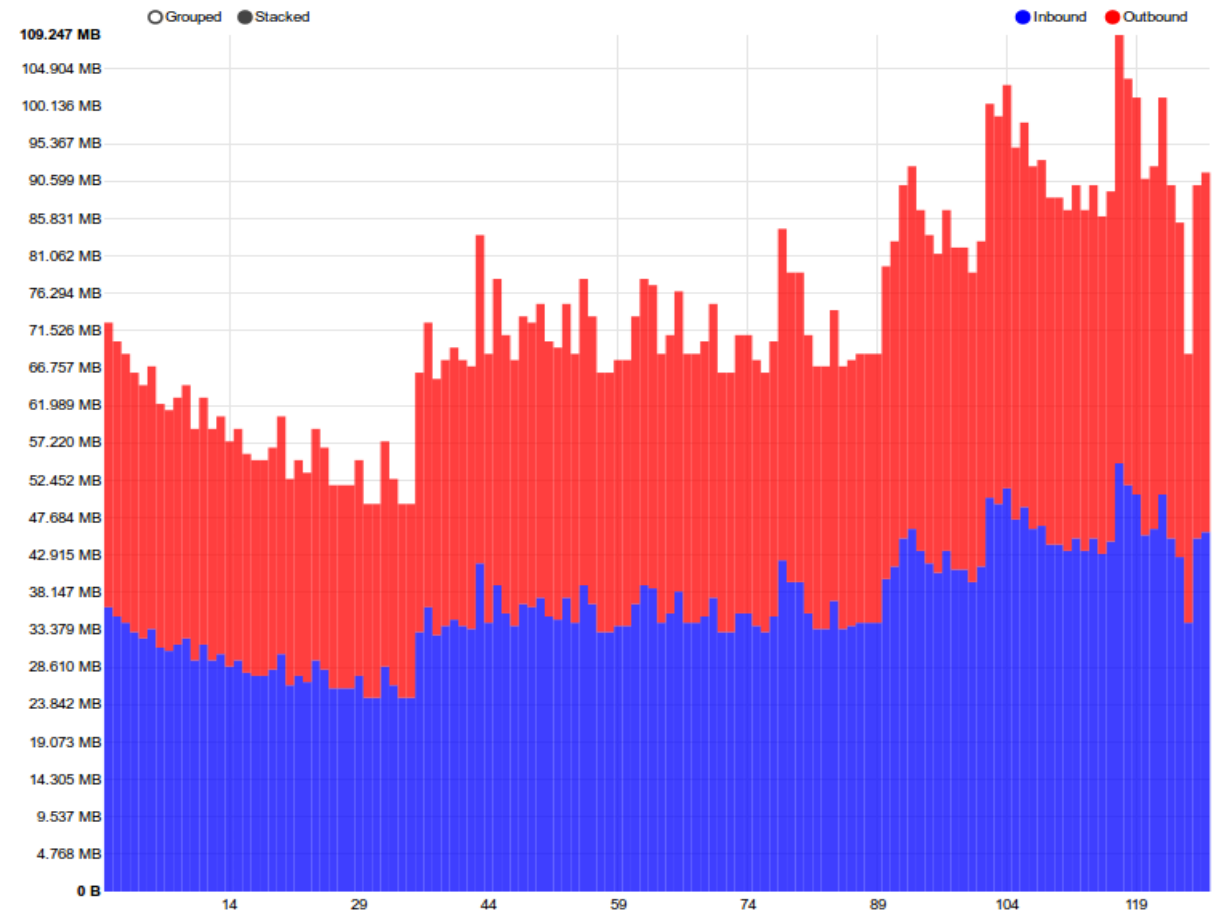
MAQAO Perf/MPI: MPI characterization

Per rank distribution

Hit, time, size

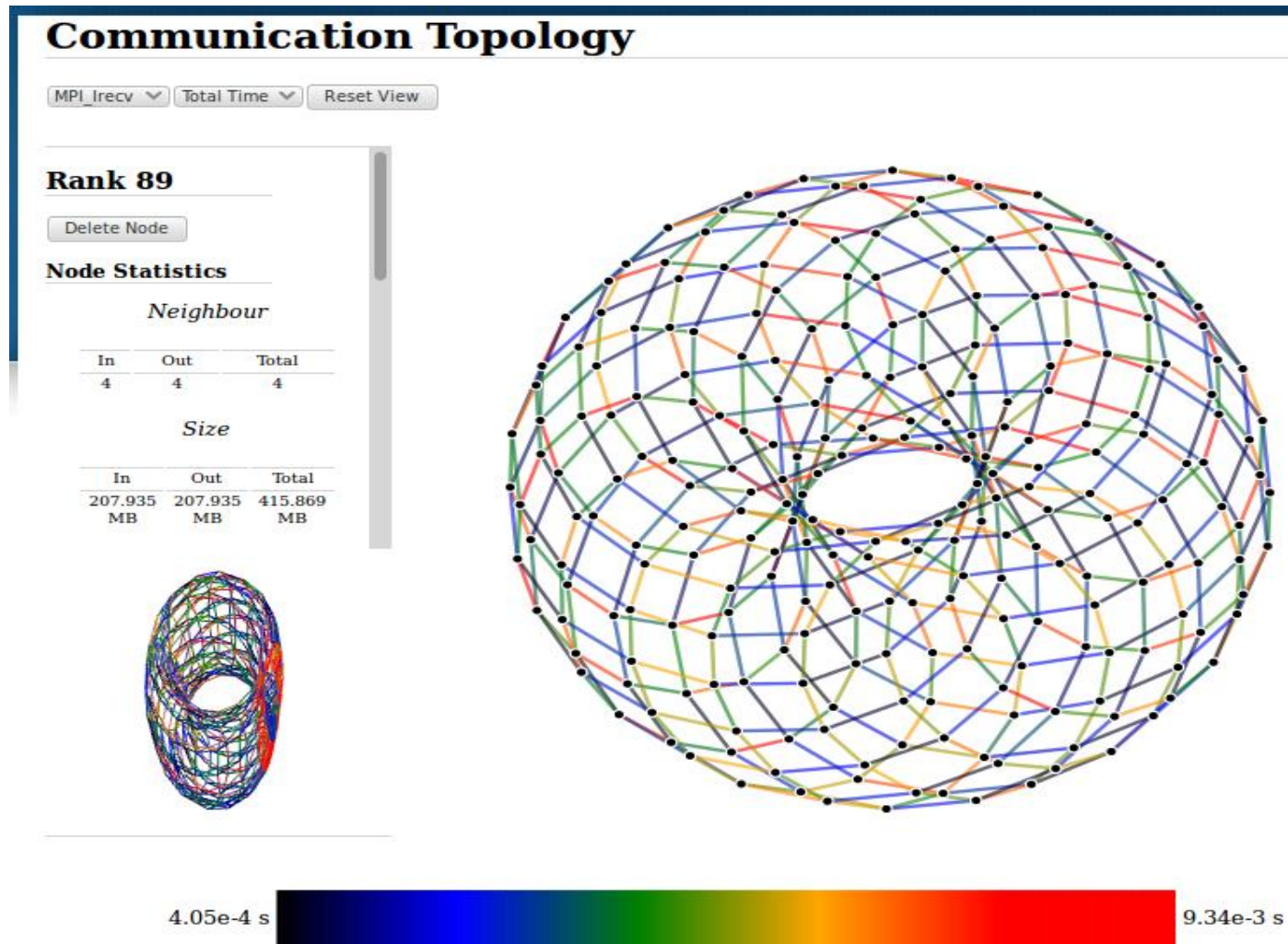
Check load balancing

Per Rank distribution



MAQAO Perf/MPI: MPI characterization

3D Topology



MAQAO CQA: Analysing the code quality of your loops

The image shows a screenshot of the MAQAO (MAQAO Code Quality Analysis) interface. The title bar reads "MAQAO Code quality analysis". Below the title, there is a section for "Source loop ending at line 682". Inside this section, there is a sub-section for "MAQAO binary loop id: 238". The text indicates that the loop is defined in "MPI/BT/x_solve.f:519-682" and that "15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))". There are four tabs: "Gain", "Potential gain", "Hints", and "Experts only". The "Gain" tab is selected, showing a section titled "Vectorization". The text in this section states: "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." Below this, it says "Proposed solution(s):" and lists two propositions: "Try another compiler or update/tune your current one:" and "Remove inter-iterations dependences from your loop and make it unit-stride." There is also a section titled "Bottlenecks" which states: "By removing all these bottlenecks, you can lower the cost of an iteration from 190.00 to 143.00 cycles (1.33x speedup)." At the bottom of the interface, there is a section for "Source loop ending at line 734".

MAQAO CQA: Analysing the code quality of your loops

Introduction

Main performance issues:

- Work sharing / communications / multicore interactions
- Core level

Most of the time core level is forgotten ! But that's where science is computed

CQA works at (assembly) loop level:

- In HPC most of the time is spent in loops (V.S. functions)
- Assess the quality of code generated by the compiler
- Take into account processor's (micro)architecture via simulation
- Hints and workarounds to improve static performance

Compute bound :

- this tool is not meant for optimizing memory issues
- It assumes that you have fixed them

MAQAO CQA: Analysing the code quality of your loops

Goal: how will it help you ?

Produce reports:

- We deal with low level details (assembly, microarchitecture details)

- You get high level reports

Provide high level reports:

- Provide source loop context when available (-g or equivalent)
- Describing a pathology/bottleneck
- Suggesting workarounds to improve static performance
- Reports categorized by confidence level:
 - gain, potential gain, hint and expert

No runtime cost/overhead:

- You don't need to execute your app
- Static analysis

The screenshot displays the MAQAO CQA tool interface. At the top, it says "Source loop ending at line 10". Below that, a dropdown menu shows "MAQAO binary loop id: 2". The main text area states: "The loop is defined in /zhome/academic/HLRS/xhp/xhpeo/TEST/matmul/kernel.c:9-10" and "2% of peak computational performance is used (0.67 out of 32.00 FLOP per cycle (1.67 GFLOPS @ 2.50GHz))". There are four tabs: "Gain", "Potential gain", "Hints", and "Experts only". The "Potential gain" tab is selected, showing a section titled "Vectorization". The text in this section reads: "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 3.00 to 0.38 cycles (8.00x speedup). Since your execution units are vector units, only a fully vectorized loop can use their full power." Below this, it says "Proposed solution(s):" and lists two propositions: "Two propositions: - Try another compiler or update/tune your current one: - Remove inter-iterations dependences from your loop and make it unit-stride." It then provides two examples of code transformations: "C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)" and "If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)".

MAQAO CQA: Analysing the code quality of your loops

Processor Architecture: Core level

Maybe you want an efficient code that gets the best out of available computing resources ?

Concepts:

- Peak performance, TOP500/LINPACK
- Execution pipeline
- Ressources/Functional units

Most of the time applications only exploit at best 5% to 10% of the peak performance

Key performance levers:

- Vectorization
- Get rid of high latency instructions if possible
- Make the compiler generated an efficient code

Same instruction – Same cost



MAQAO CQA: Analysing the code quality of your loops

The compiler

Compiler remains our best friend

Be sure to select proper flags

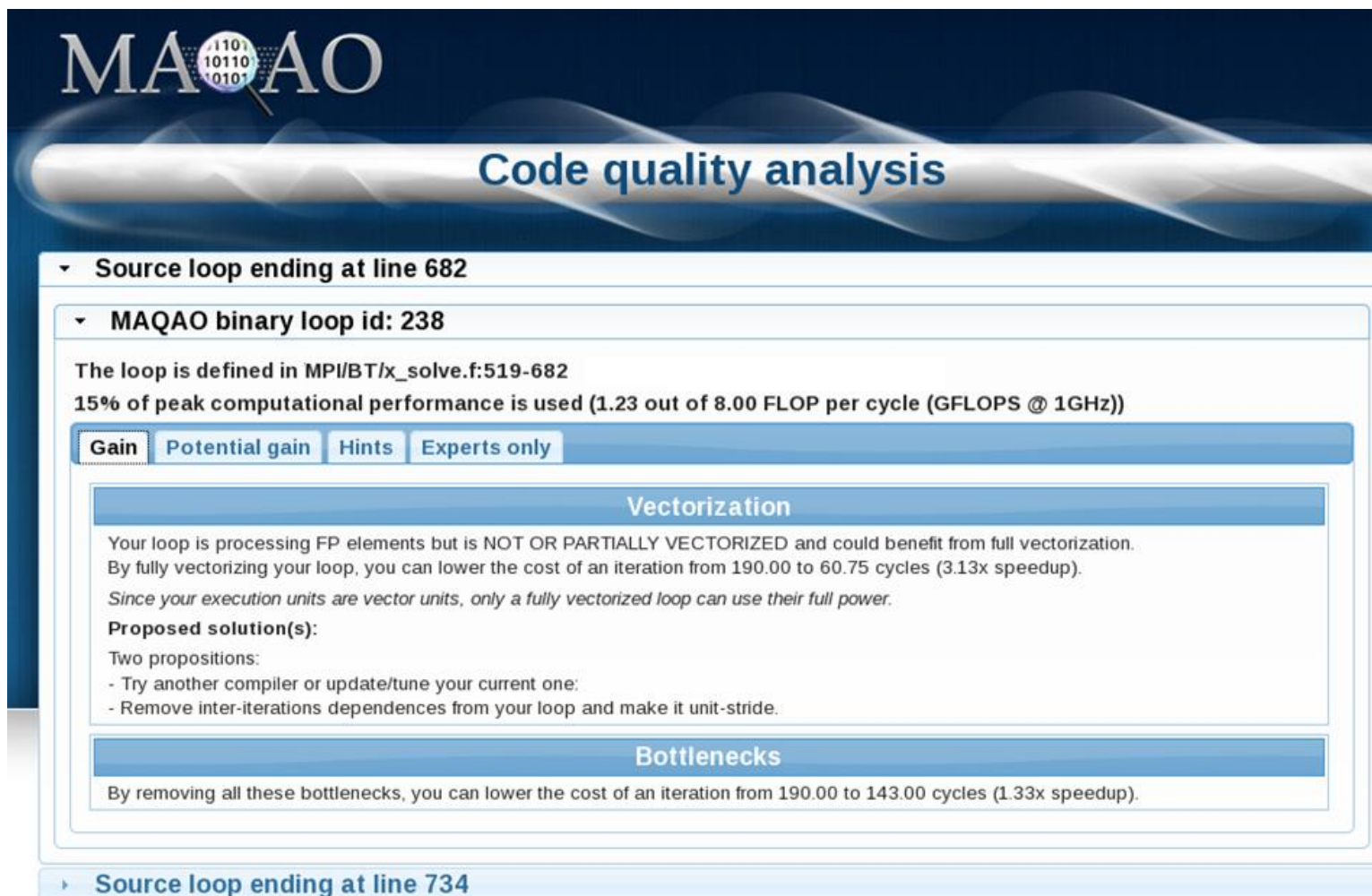
- Know default flags (e.g., -xHost on AVX capable machines)
- Bypass conservative behavior when possible (e.g., 1/X precision)

Pragmas:

- Vectorization, Alignement, Unrolling, etc...
- Portable transformations

MAQAO CQA: Analysing the code quality of your loops

GUI sample (1/2)



The screenshot displays the MAQAO Code quality analysis interface. At the top, the MAQAO logo is shown next to a magnifying glass icon over binary code. Below the logo, the title "Code quality analysis" is centered. The main content area is divided into sections. The first section, "Source loop ending at line 682", is expanded to show details for "MAQAO binary loop id: 238". It states that the loop is defined in MPI/BT/x_solve.f:519-682 and that 15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz)). Below this, there are four tabs: "Gain", "Potential gain", "Hints", and "Experts only". The "Gain" tab is selected, showing a "Vectorization" section. This section explains that the loop is processing FP elements but is not or partially vectorized, and could benefit from full vectorization. It states that by fully vectorizing the loop, the cost of an iteration can be lowered from 190.00 to 60.75 cycles (3.13x speedup). It also notes that since execution units are vector units, only a fully vectorized loop can use their full power. Below this, it lists "Proposed solution(s):" with two propositions: "Try another compiler or update/tune your current one:" and "Remove inter-iterations dependences from your loop and make it unit-stride." Below the vectorization section, there is a "Bottlenecks" section. It states that by removing all these bottlenecks, the cost of an iteration can be lowered from 190.00 to 143.00 cycles (1.33x speedup). The bottom of the interface shows a section for "Source loop ending at line 734".

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.

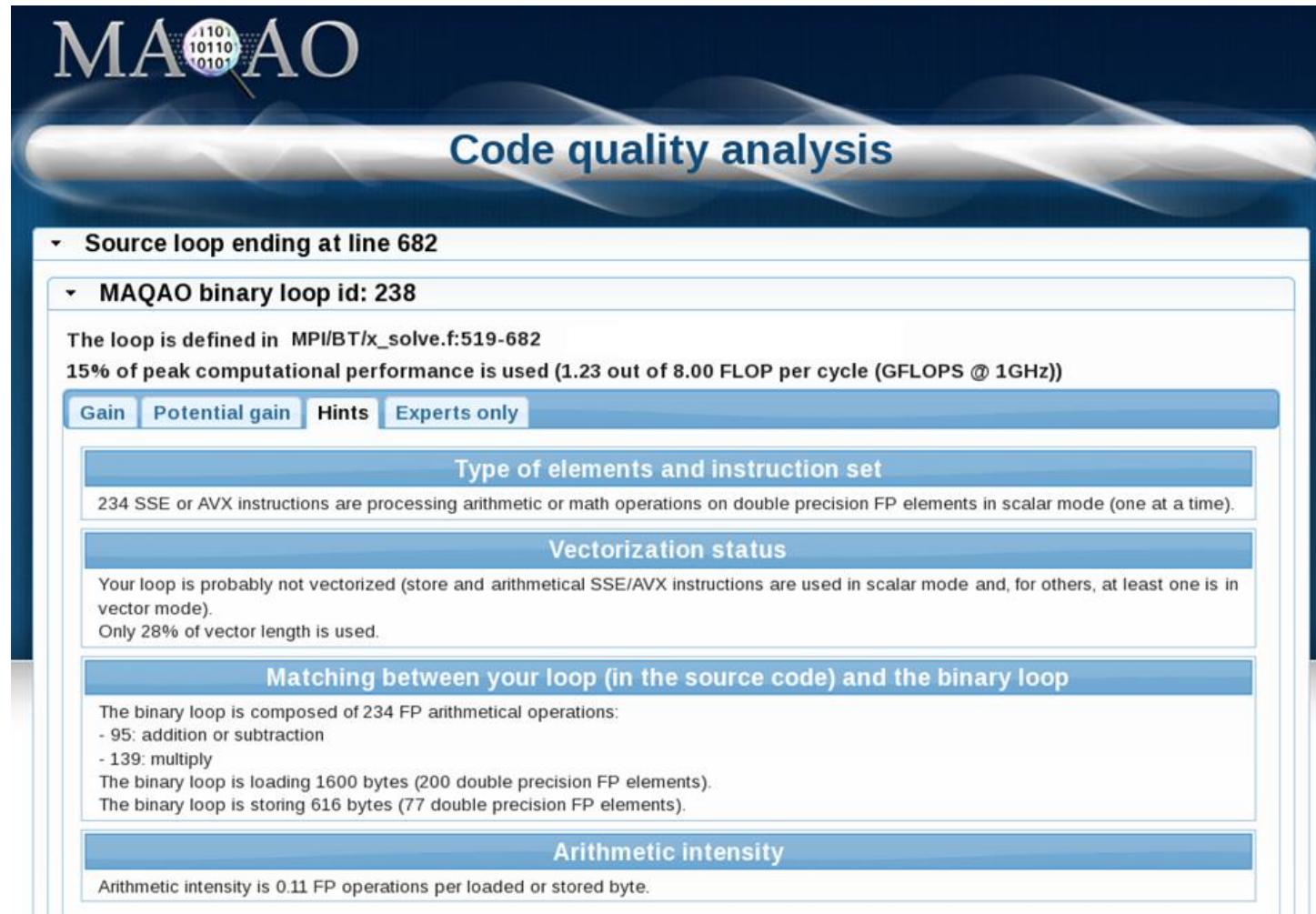
Bottlenecks

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

MAQAO CQA: Analysing the code quality of your loops

GUI sample (2/2)



The screenshot displays the MAQAO Code quality analysis interface. At the top, the MAQAO logo is shown next to a magnifying glass icon over binary code. Below the logo, the title "Code quality analysis" is centered. The main content area is divided into sections. The first section, "Source loop ending at line 682", is expanded. Below it, the "MAQAO binary loop id: 238" section is also expanded. This section contains the text: "The loop is defined in MPI/BT/x_solve.f:519-682" and "15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))". Below this text are four tabs: "Gain", "Potential gain", "Hints", and "Experts only". The "Potential gain" tab is selected. Under this tab, there are three main sections: "Type of elements and instruction set", "Vectorization status", and "Matching between your loop (in the source code) and the binary loop". The "Type of elements and instruction set" section states: "234 SSE or AVX instructions are processing arithmetic or math operations on double precision FP elements in scalar mode (one at a time)". The "Vectorization status" section states: "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." The "Matching between your loop (in the source code) and the binary loop" section states: "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)." Below these sections is the "Arithmetic intensity" section, which states: "Arithmetic intensity is 0.11 FP operations per loaded or stored byte."

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.

Thank you for your attention

Questions