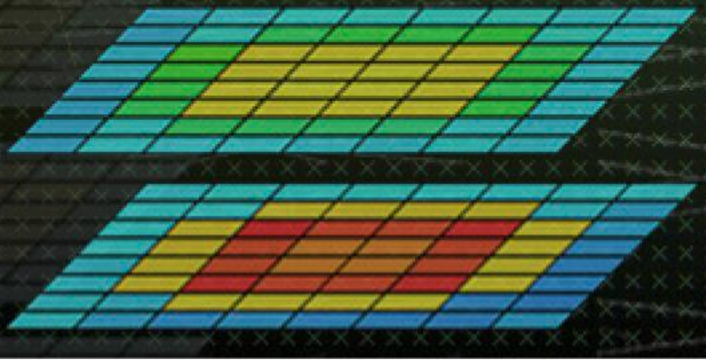
VIRTUAL INSTITUTE – HIGH PRODUCTIVITY SUPERCOMPUTING



MAQAO
Performance Analysis and Optimization Tool

Cédric VALENSI, Emmanuel OSERET, Salah IBN AMAR
{cedric.valensi, emmanuel.oseret, mohammed-salah.ibnamar}@uvsq.fr
Performance Evaluation Team, University of Versailles S-Q-Y
Andres S. CHARIF RUBIAL - ascr@pexl.eu - PeXL

http://www.maqao.org

VI-HPS 27th Garching – Germany – 23-27 April 2018

Introduction
Performance analysis and optimisation

How much can I optimise my application?

- Can it actually be done?
- What would the effort/gain ratio be?

Where can I gain time?

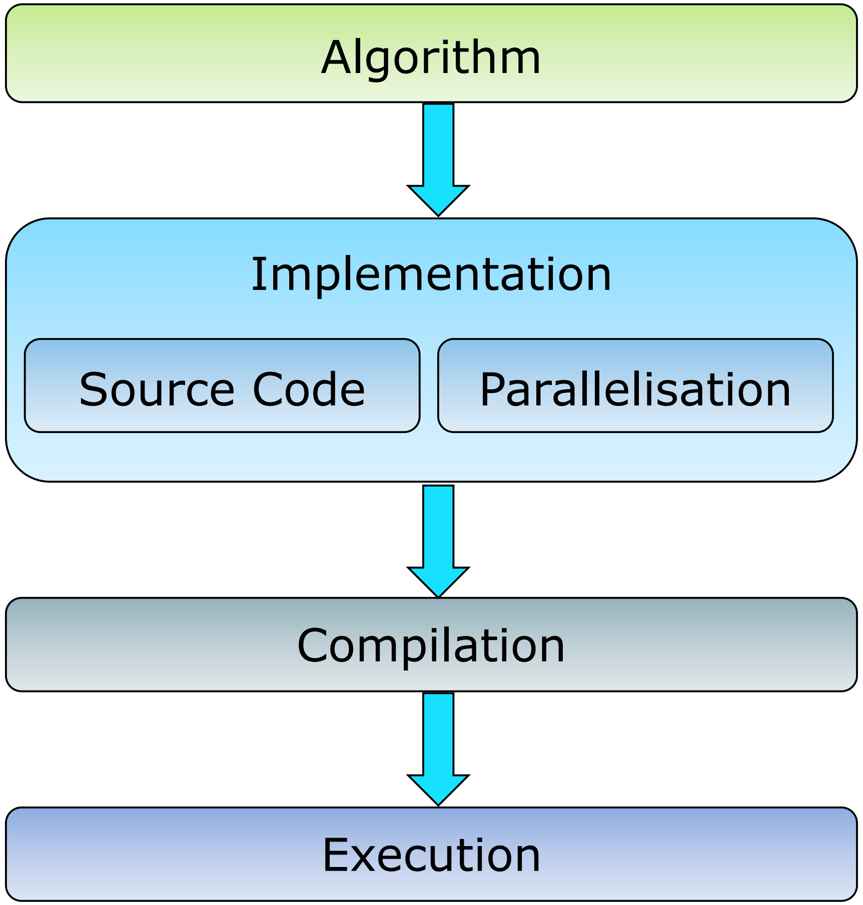
- Where is my application wasting time?

Why is the application spending time there?

- Algorithm, implementation or hardware?
- Data access or computation?

How can I improve the situation?

- In which step(s) of the design process?
- What additional information do I need?


Algorithm

Implementation

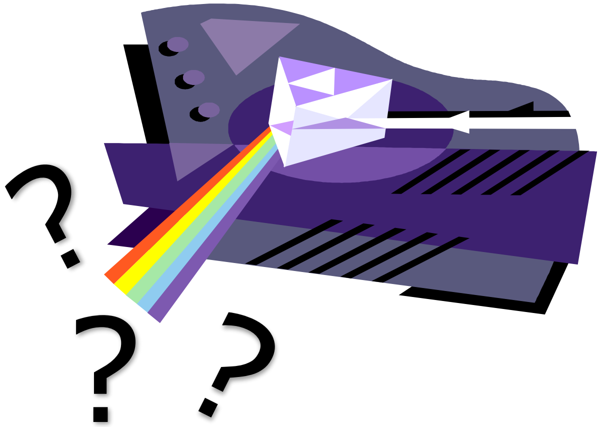
Source Code

Parallelisation

Compilation

Execution

Introduction
A multifaceted problem



Pinpointing the performance bottlenecks

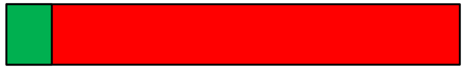
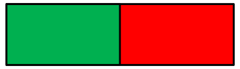
Identifying the dominant issues

- Algorithms, implementation, parallelisation, …

Making the **best use** of the machine features

- Complex multicore and manycore CPUs
- Complex memory hierarchy

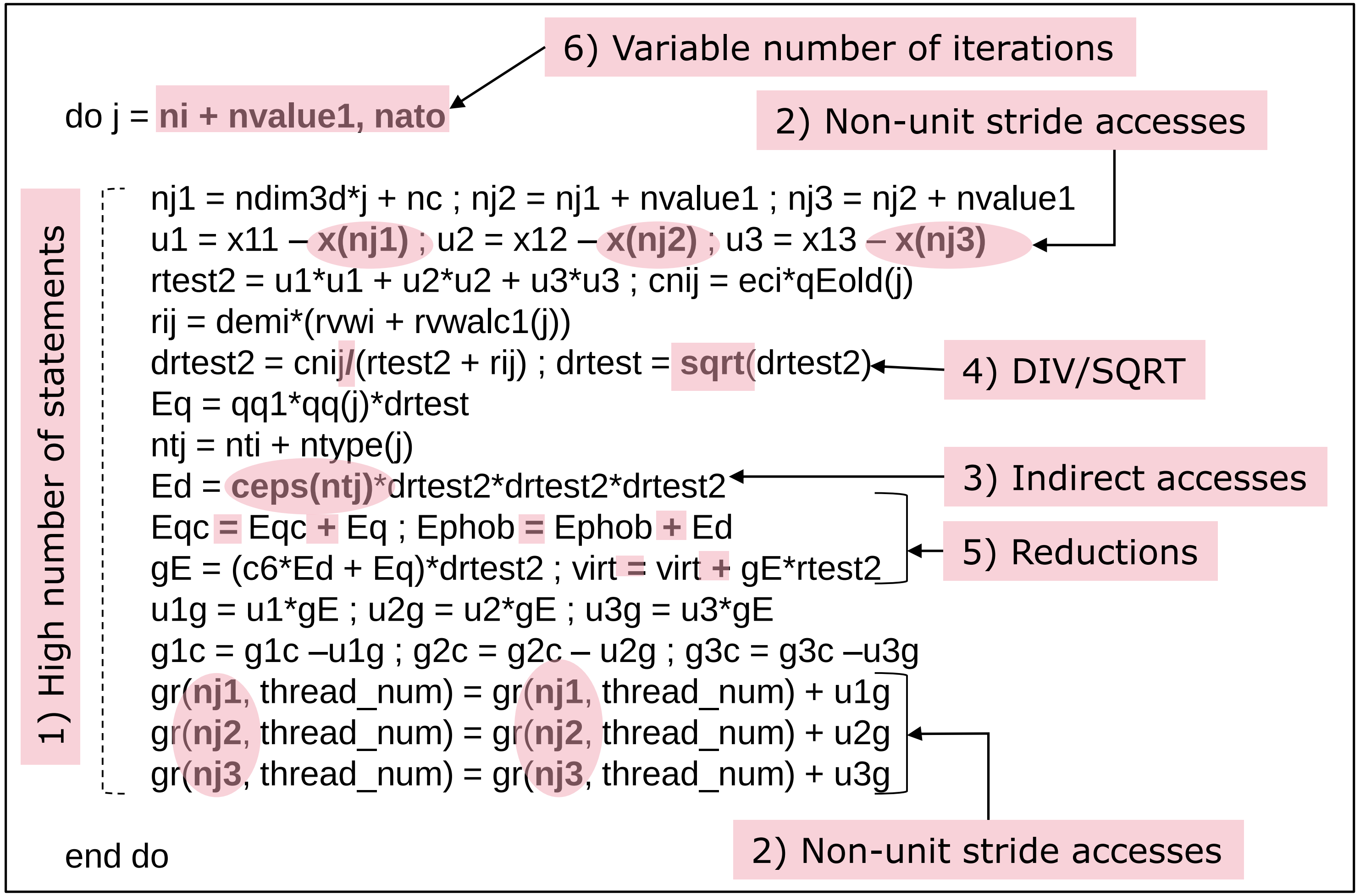
Finding the **most rewarding** issues to be fixed

- **40%** total time, expected **10%** speedup
  - ➔ TOTAL IMPACT: **4%** speedup 
- **20%** total time, expected **50%** speedup
  - ➔ TOTAL IMPACT: **10%** speedup 

=> Need for dedicated and complementary tools

Introduction
Motivating example

Code of a loop representing ~10% walltime



```
do j = ni + nvalue1, nato

    nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
    u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
    rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
    rij = demi*(rvwi + rvwalc1(j))
    drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
    Eq = qq1*qq(j)*drtest
    ntj = nti + ntype(j)
    Ed = ceps(ntj)*drtest2*drtest2*drtest2
    Eqc = Eqc + Eq ; Ephob = Ephob + Ed
    gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
    u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
    g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
    gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
    gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
    gr(nj3, thread_num) = gr(nj3, thread_num) + u3g

end do
```

6) Variable number of iterations

2) Non-unit stride accesses

1) High number of statements

4) DIV/SQRT

3) Indirect accesses

5) Reductions

2) Non-unit stride accesses

Source code and associated issues:

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

Introduction
MAQAO: Modular Assembly Quality Analyzer and Optimizer

Objectives:

- Characterizing performance of HPC applications
- Guiding users through optimization process
- Offering complementary views
- Estimating R.O.I.

Main features:

- Profiling
- Code quality analysis

Characteristics:

- Modular tool
- Support for Intel x86-64 and Xeon Phi
- LGPL3 Open Source software
- Developed at UVSQ since 2004

Introduction
Partnerships

MAQAO was funded by UVSQ, Intel and CEA (French department of energy) through Exascale Computing Research (ECR) and the French Ministry of Industry through various FUI/ITEA projects (H4H, COLOC, PerfCloud, ELCI, etc...)

Provides core technology to be integrated with other tools:

- TAU performance tools with MADRAS patcher through MIL (MAQAO Instrumentation Language)
- ATOS bullxprof with MADRAS through MIL
- Intel Advisor
- INRIA Bordeaux HWLOC

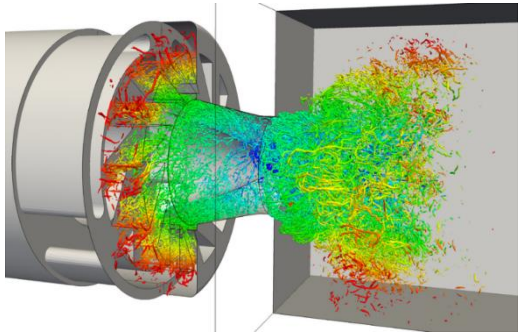
PeXL ISV also contributes to MAQAO:

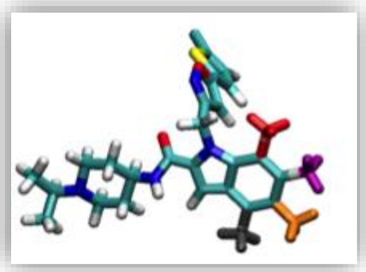
- Commercial performance optimization expertise
- Training and software development

Introduction
Success stories

MAQAO was used for optimizing industrial and academic HPC applications:

- QMC=CHEM (IRSAMC)
  - Quantum chemistry
  - Speedup: **> 3x**
    - Moved invocation of function with identical parameters out of loop body
- Yales2 (CORIA)
  - Computational fluid dynamics
  - Speedup: **up to 2.8x**
    - Removed double structure indirections
- Polaris (CEA)
  - Molecular dynamics
  - Speedup: **1.5x – 1.7x**
    - Enforced loop vectorisation through compiler directives
- AVBP (CERFACS)
  - Computational fluid dynamics
  - Speedup: **1.08x – 1.17x**
    - Replaced division with multiplication by reciprocal
    - Complete unrolling of loops with small number of iterations





Introduction
Some MAQAO Collaborators

- Prof. William Jalby
- Prof. Denis Barthou
- Prof. David J. Kuck
- Andrés S. Charif-Rubial, Ph D
- Jean-Thomas Acquaviva, Ph D
- Stéphane Zuckerman, Ph D
- Julien Jaeger, Ph D
- Souad Koliaï, Ph D
- Cédric Valensi, Ph D
- Eric Petit, Ph D
- Zakaria Bendifallah, Ph D
- Emmanuel Oseret, Ph D
- Pablo de Oliveira, Ph D
- Tipp Moseley, Ph D
- David C. Wong, Ph D
- Jean-Christophe Beyler, Ph D
- Mathieu Tribalat
- Hugo Bolloré
- Jean-Baptiste Le Reste
- Sylvain Henry, Ph D
- Salah Ibn Amar
- Youenn Lebras
- Othman Bouizi, Ph D
- José Noudohouenou, Ph D
- Aleksandre Vardoshvili
- Romain Pillot

Introduction
MAQAO: Analysis at binary level

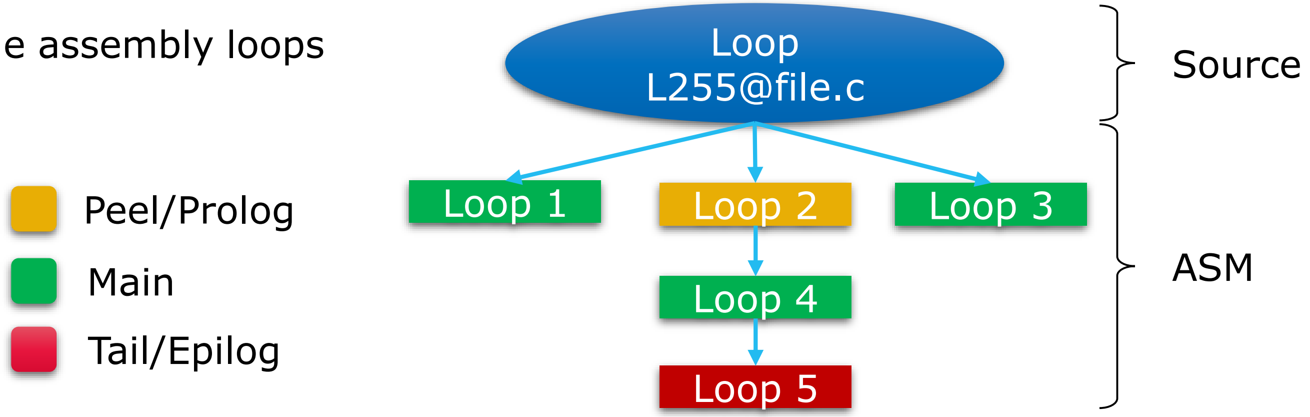
Advantages of binary analysis:

- Compiler optimizations increase the distance between the executed code and the source
- Source code instrumentation may prevent the compiler from applying some transformations

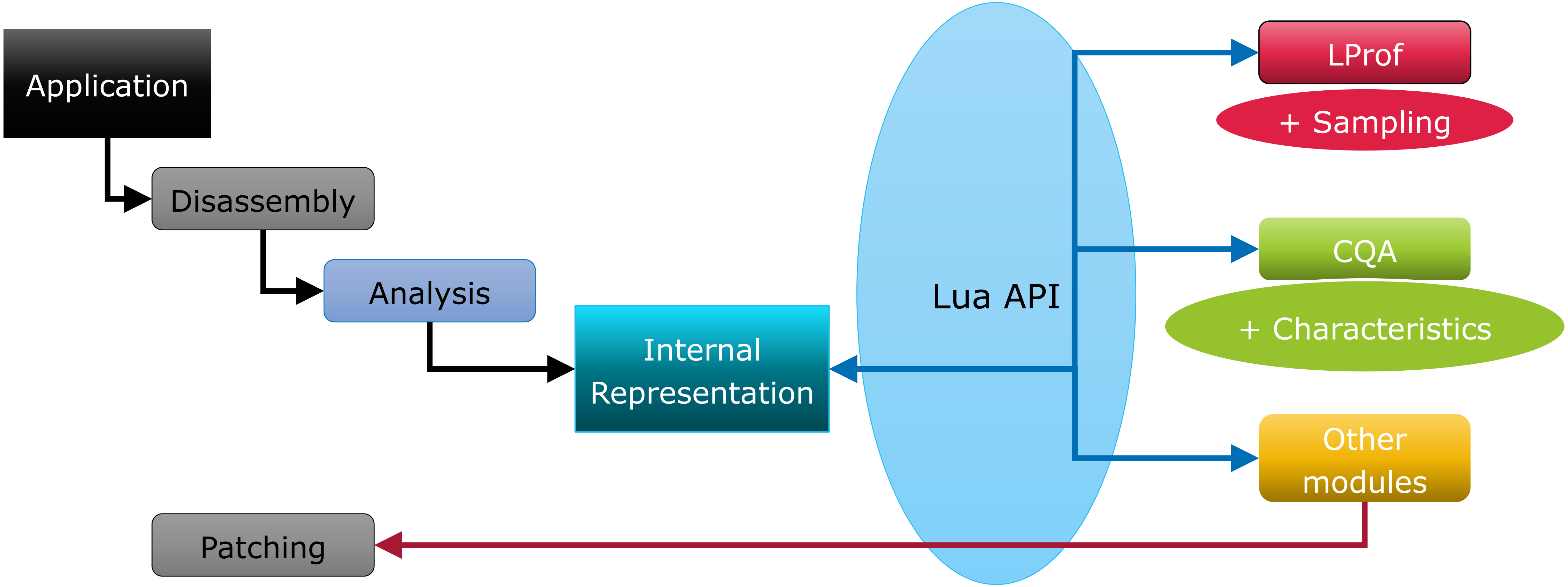
We want to evaluate the “real” executed code: **What You Analyse Is What You Run**

Main steps:

- Reconstruct the program structure
- Relate the analyses to source code
  - A single source loop can be compiled as multiple assembly loops
  - Affecting unique identifiers to loops

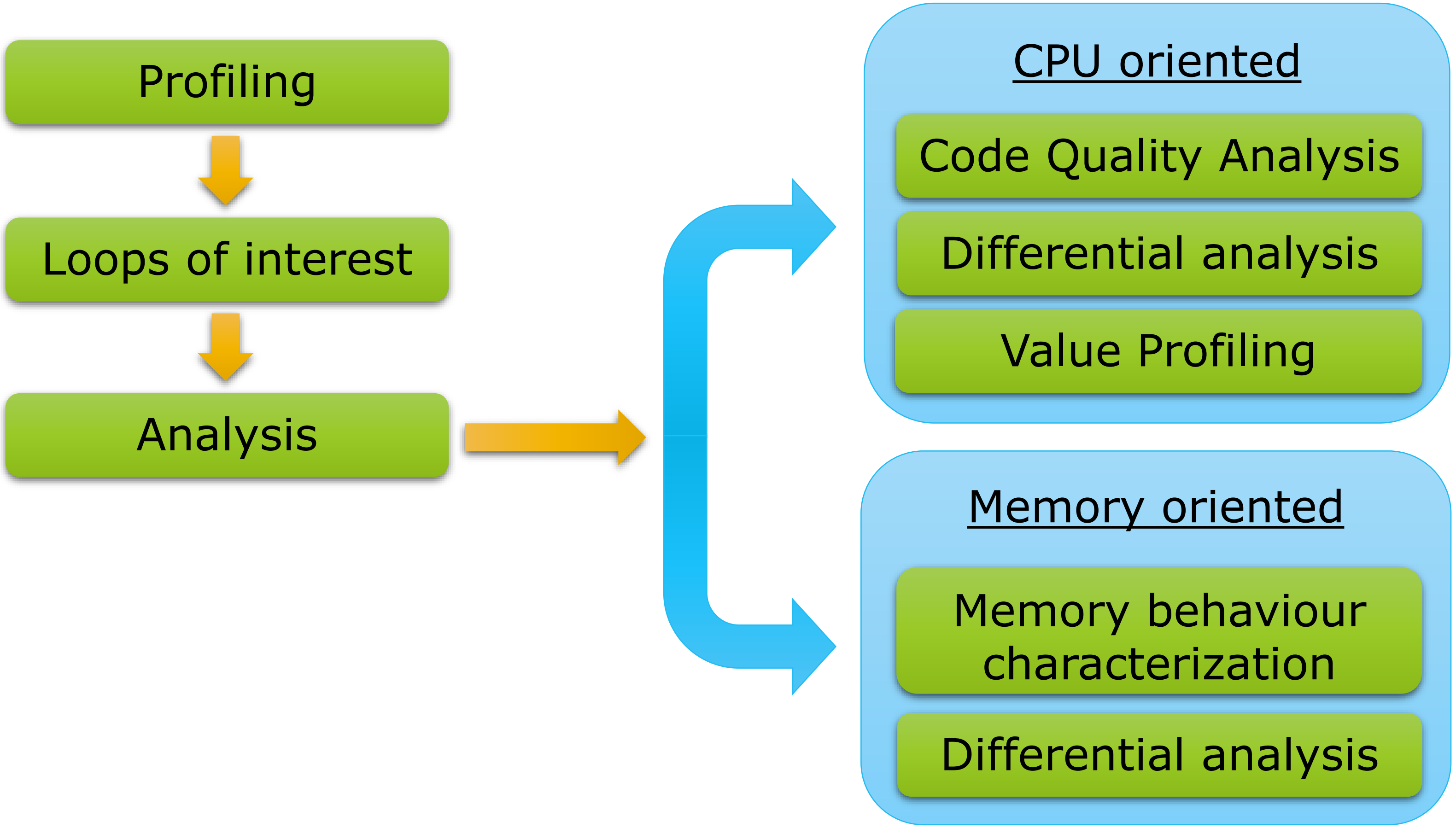


Introduction
MAQAO Main structure

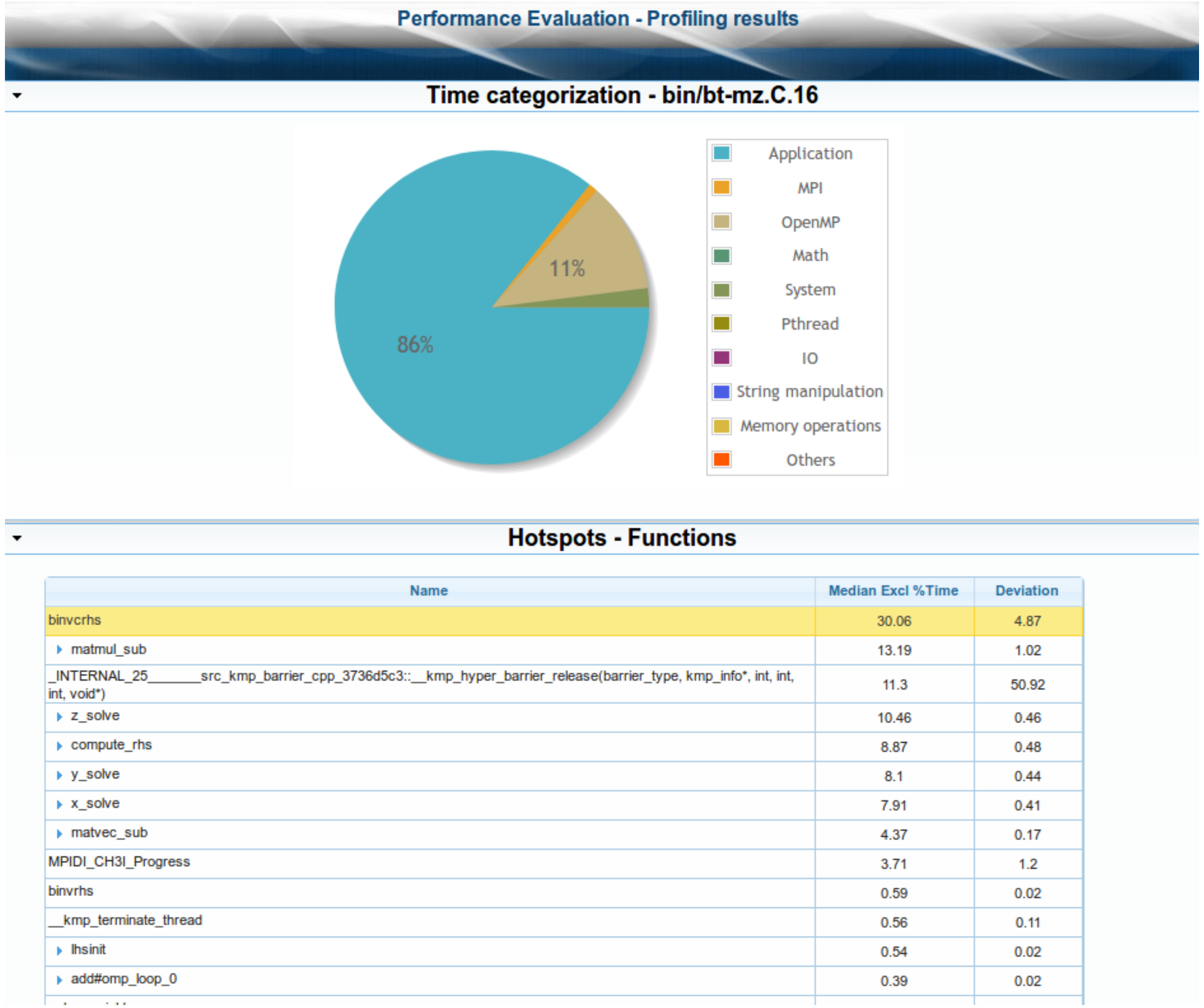


Introduction
MAQAO Methodology

Decision tree



MAQAO LProf: Lightweight Profiler

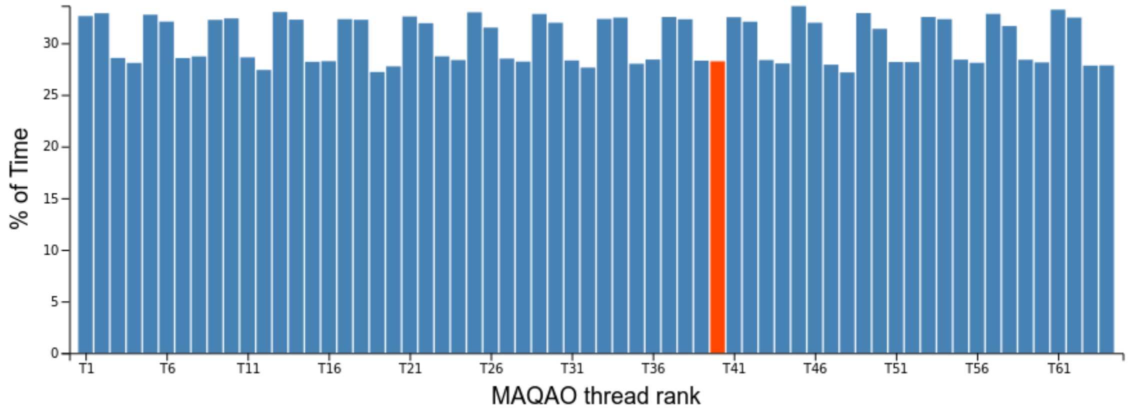


Performance Evaluation - Profiling results

Time categorization - bin/bt-mz.C.16

Hotspots - Functions

Name	Median Excl %Time	Deviation
binvcrhs	30.06	4.87
matmul_sub	13.19	1.02
_INTERNAL_25________src_kmp_barrier_cpp_3736d5c3::__kmp_hyper_barrier_release(barrier_type, kmp_info*, int, int, int, void*)	11.3	50.92
z_solve	10.46	0.46
compute_rhs	8.87	0.48
y_solve	8.1	0.44
x_solve	7.91	0.41
matvec_sub	4.37	0.17
MPIDI_CH3I_Progress	3.71	1.2
binvrhs	0.59	0.02
__kmp_terminate_thread	0.56	0.11
lhsinit	0.54	0.02
add#omp_loop_0	0.39	0.02



MAQAO LProf: Lightweight Profiler
Introduction

Goal: Lightweight localization of application hotspots

Features:

- **Sampling** based
- Access to hardware counters for additional information
- Results at function and loop granularity

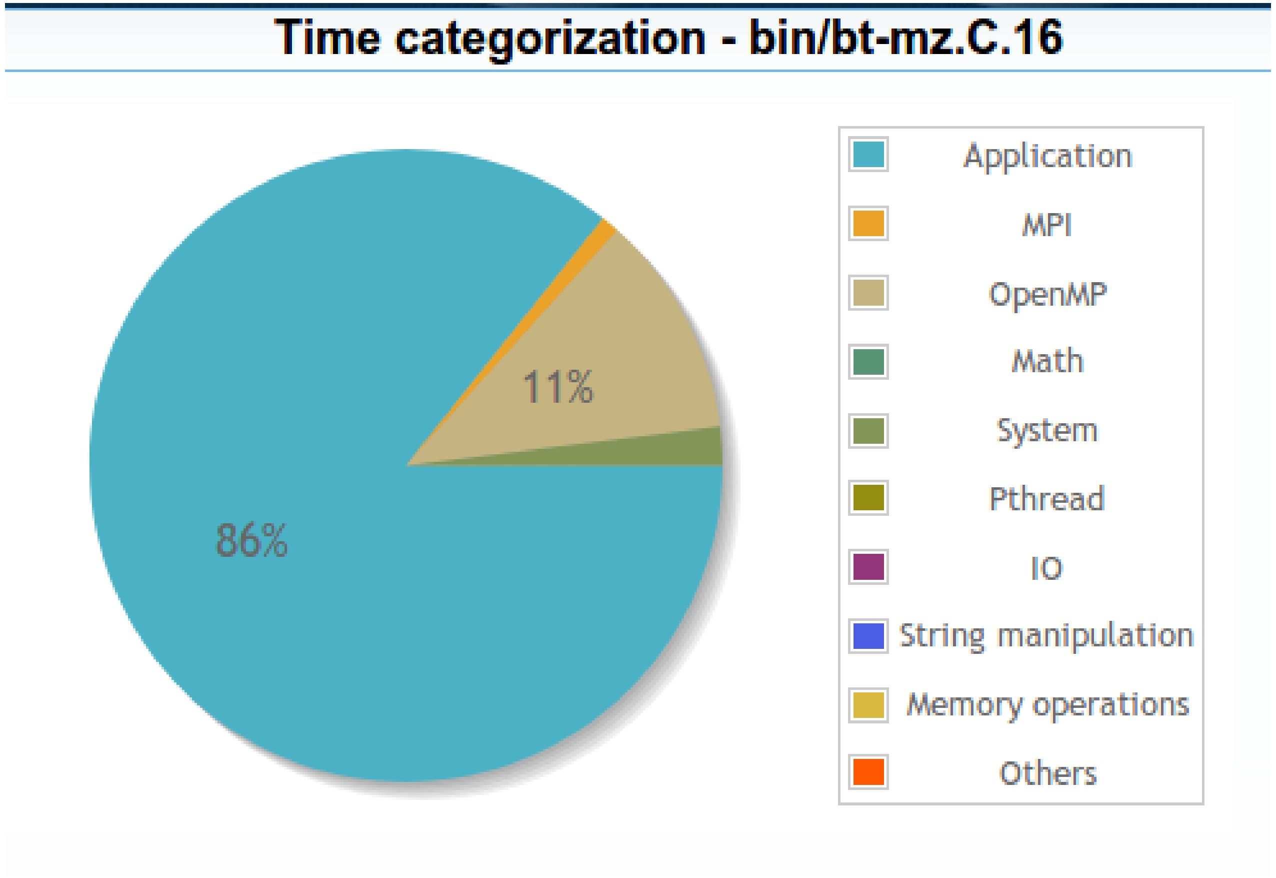
Strengths:

- **Non intrusive**: No recompilation necessary
- **Low overhead**
- Agnostic with regard to parallel runtime

MAQAO LProf: Lightweight Profiler
Time categorization

Identifying at a glance where time is spent

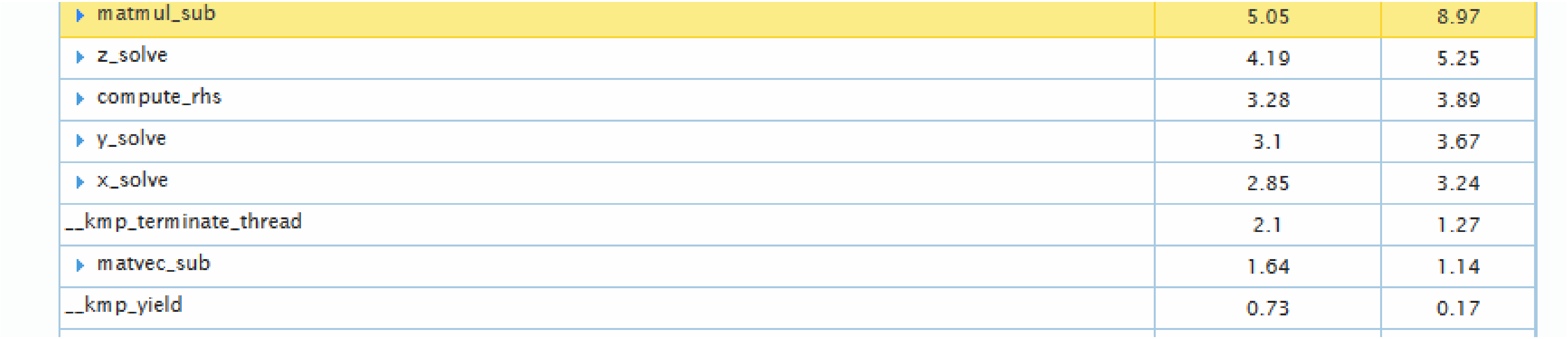
- Application
  - Main executable
- Parallelization
  - Threads
  - OpenMP
  - MPI
- System libraries
  - I/O operations
  - String operations
  - Memory management functions
- External libraries
  - Specialised libraries such as libm / libmkl
  - Application code in external libraries



MAQAO LProf: Lightweight Profiler
Functions hotspots

Focusing on user time:

- Function hotspots

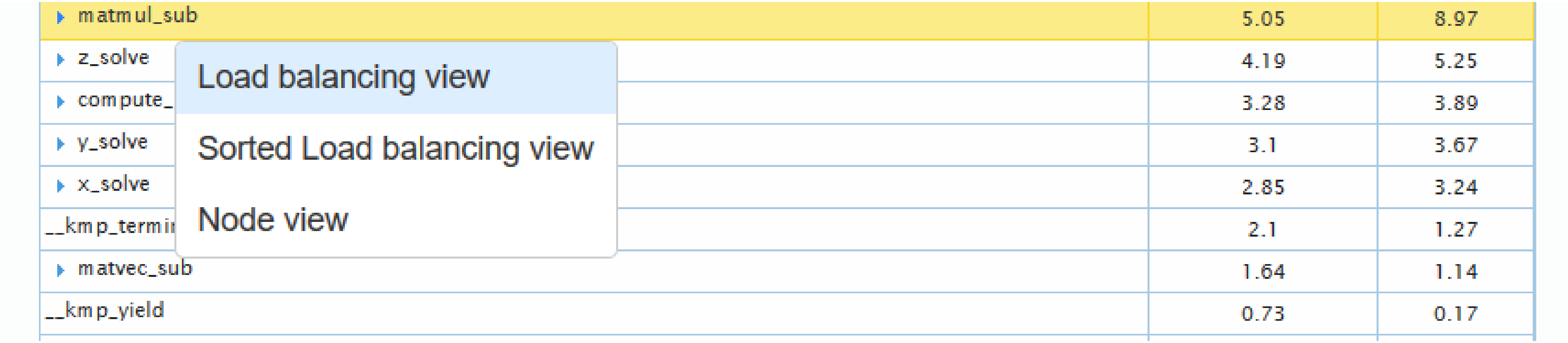


matmul_sub	5.05	8.97
z_solve	4.19	5.25
compute_rhs	3.28	3.89
y_solve	3.1	3.67
x_solve	2.85	3.24
__kmp_terminate_thread	2.1	1.27
matvec_sub	1.64	1.14
__kmp_yield	0.73	0.17

MAQAO LProf: Lightweight Profiler
Functions hotspots

Focusing on user time:

- Function hotspots
- Load balancing across the threads/processes/nodes

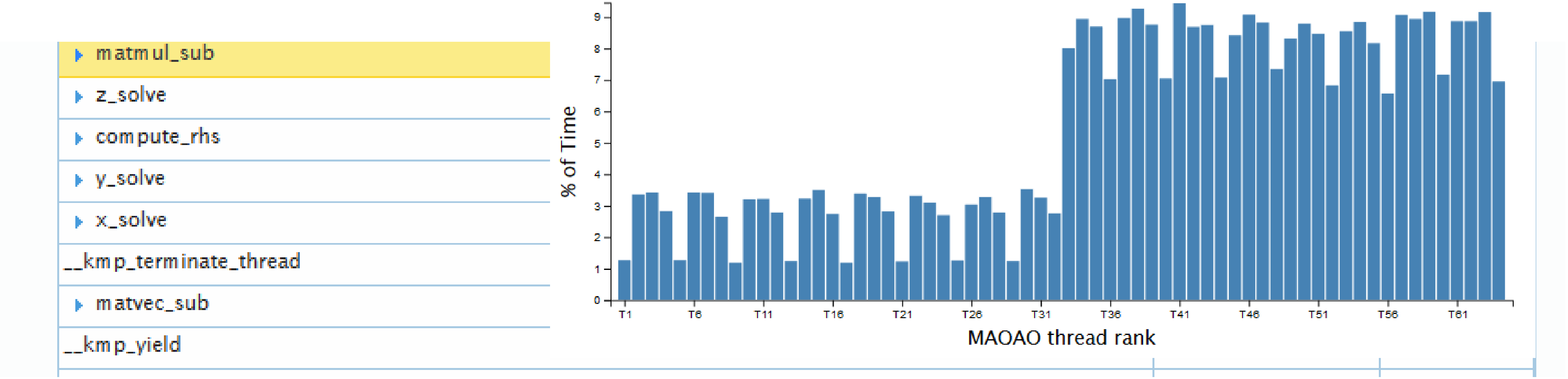


matmul_sub	5.05	8.97
z_solve	4.19	5.25
compute_	3.28	3.89
y_solve	3.1	3.67
x_solve	2.85	3.24
__kmp_termi	2.1	1.27
matvec_sub	1.64	1.14
__kmp_yield	0.73	0.17

MAQAO LProf: Lightweight Profiler
Functions load balancing

Focusing on user time:

- Function hotspots
- Load balancing across the threads/processes/nodes

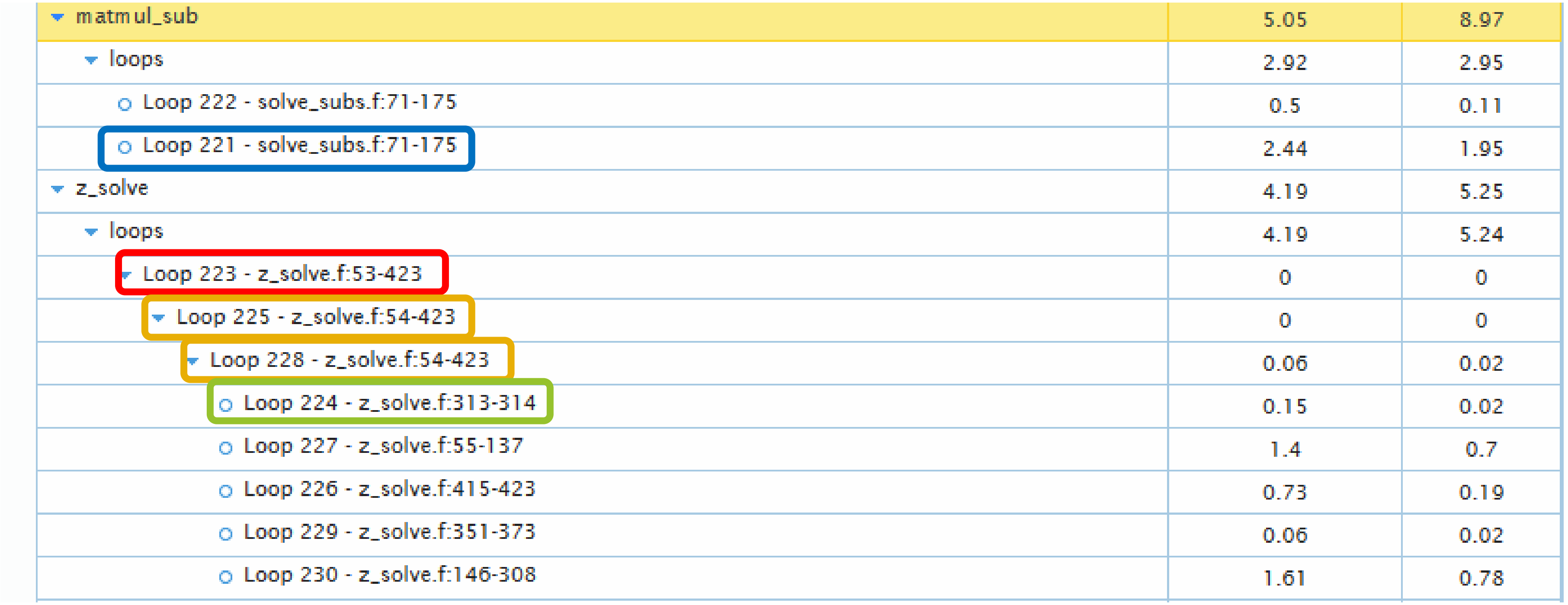


- matmul_sub
- z_solve
- compute_rhs
- y_solve
- x_solve
- __kmp_terminate_thread
- matvec_sub
- __kmp_yield

MAQAO LProf: Lightweight Profiler
Loops hotspots

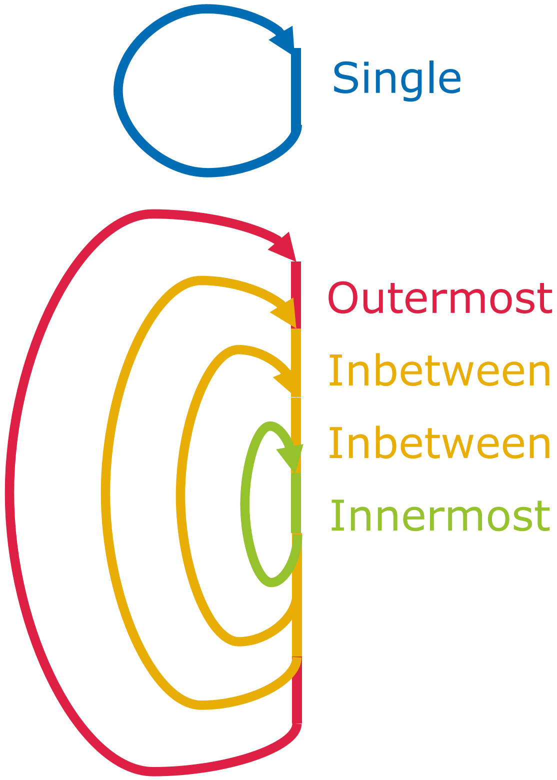
Analysing the time spent at loop level:

- Finding the most time consuming
- Providing loop id for further MAQAO analyses

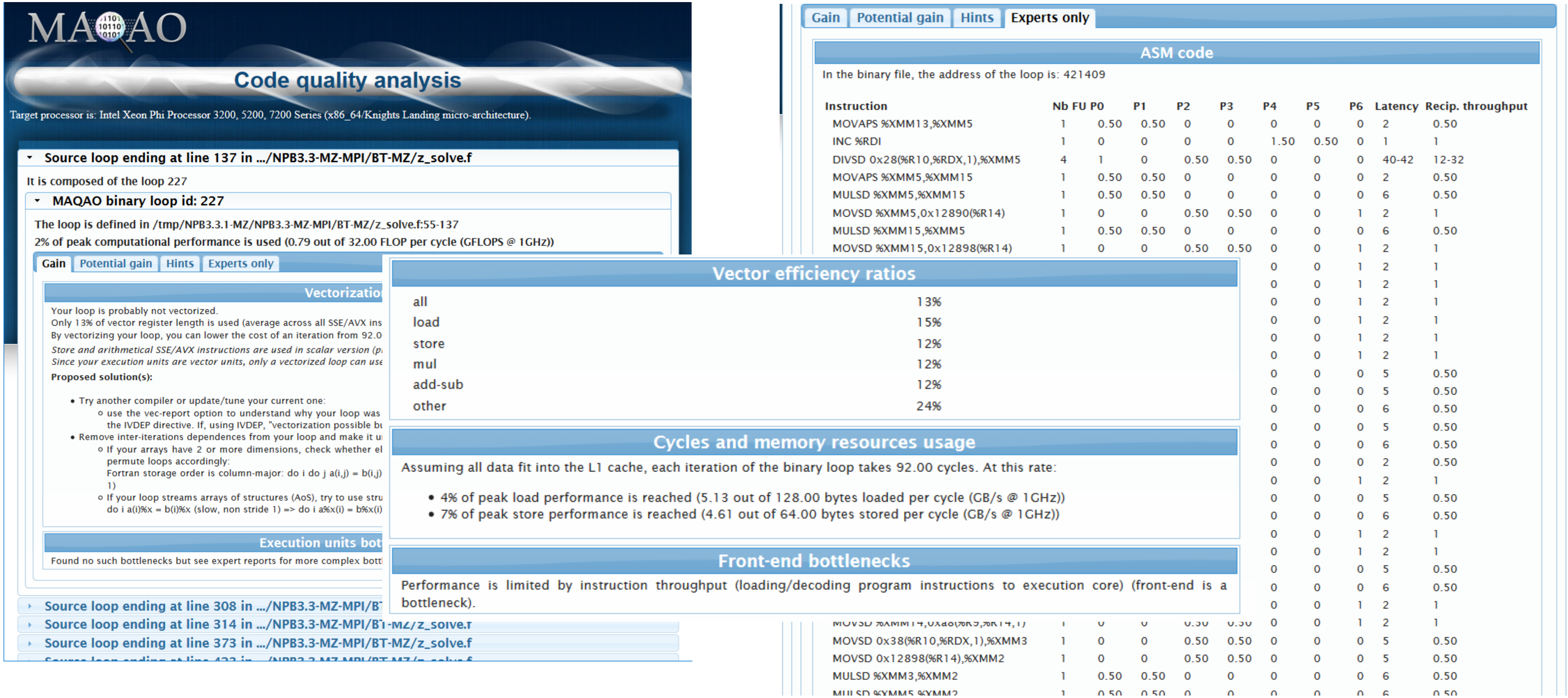


matmul_sub	5.05	8.97
loops	2.92	2.95
Loop 222 - solve_subs.f:71-175	0.5	0.11
Loop 221 - solve_subs.f:71-175	2.44	1.95
z_solve	4.19	5.25
loops	4.19	5.24
Loop 223 - z_solve.f:53-423	0	0
Loop 225 - z_solve.f:54-423	0	0
Loop 228 - z_solve.f:54-423	0.06	0.02
Loop 224 - z_solve.f:313-314	0.15	0.02
Loop 227 - z_solve.f:55-137	1.4	0.7
Loop 226 - z_solve.f:415-423	0.73	0.19
Loop 229 - z_solve.f:351-373	0.06	0.02
Loop 230 - z_solve.f:146-308	1.61	0.78

Loop hierarchy



MAQAO CQA: Code Quality Analyzer



Code quality analysis

Target processor is: Intel Xeon Phi Processor 3200, 5200, 7200 Series (x86_64/Knights Landing micro-architecture).

▾ Source loop ending at line 137 in .../NPB3.3-MZ-MPI/BT-MZ/z_solve.f

It is composed of the loop 227

▾ MAQAO binary loop id: 227

The loop is defined in /tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ/z_solve.f:55-137

2% of peak computational performance is used (0.79 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

Vectorizatio

Your loop is probably not vectorized.
Only 13% of vector register length is used (average across all SSE/AVX ins
By vectorizing your loop, you can lower the cost of an iteration from 92.0

Store and arithmetical SSE/AVX instructions are used in scalar version (p
Since your execution units are vector units, only a vectorized loop can use

Proposed solution(s):

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was the IVDEP directive. If, using IVDEP, "vectorization possible bu
- Remove inter-iterations dependences from your loop and make it u
  - If your arrays have 2 or more dimensions, check whether el permute loops accordingly:
    Fortran storage order is column-major: do i do j a(i,j) = b(i,j) 1)
  - If your loop streams arrays of structures (AoS), try to use stru do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i)

Execution units bot

Found no such bottlenecks but see expert reports for more complex bott

▸ Source loop ending at line 308 in .../NPB3.3-MZ-MPI/BT

▸ Source loop ending at line 314 in .../NPB3.3-MZ-MPI/BT-MZ/z_solve.f

▸ Source loop ending at line 373 in .../NPB3.3-MZ-MPI/BT-MZ/z_solve.f

Vector efficiency ratios

all	13%
load	15%
store	12%
mul	12%
add-sub	12%
other	24%

Cycles and memory resources usage

Assuming all data fit into the L1 cache, each iteration of the binary loop takes 92.00 cycles. At this rate:

- 4% of peak load performance is reached (5.13 out of 128.00 bytes loaded per cycle (GB/s @ 1GHz))
- 7% of peak store performance is reached (4.61 out of 64.00 bytes stored per cycle (GB/s @ 1GHz))

Front-end bottlenecks

Performance is limited by instruction throughput (loading/decoding program instructions to execution core) (front-end is a bottleneck).

ASM code

In the binary file, the address of the loop is: 421409

Instruction	Nb FU	P0	P1	P2	P3	P4	P5	P6	Latency	Recip. throughput
MOVAPS %XMM13,%XMM5	1	0.50	0.50	0	0	0	0	0	2	0.50
INC %RDI	1	0	0	0	0	1.50	0.50	0	1	1
DIVSD 0x28(%R10,%RDX,1),%XMM5	4	1	0	0.50	0.50	0	0	0	40-42	12-32
MOVAPS %XMM5,%XMM15	1	0.50	0.50	0	0	0	0	0	2	0.50
MULSD %XMM5,%XMM15	1	0.50	0.50	0	0	0	0	0	6	0.50
MOVSD %XMM5,0x12890(%R14)	1	0	0	0.50	0.50	0	0	1	2	1
MULSD %XMM15,%XMM5	1	0.50	0.50	0	0	0	0	0	6	0.50
MOVSD %XMM15,0x12898(%R14)	1	0	0	0.50	0.50	0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
						0	0	0	5	0.50
						0	0	0	5	0.50
						0	0	0	6	0.50
						0	0	0	5	0.50
						0	0	0	6	0.50
						0	0	0	2	0.50
						0	0	1	2	1
						0	0	0	5	0.50
						0	0	0	6	0.50
						0	0	1	2	1
						0	0	1	2	1
						0	0	0	5	0.50
						0	0	0	6	0.50
						0	0	1	2	1
MOVSD %XMM14,0xa8(%R9,%R14,1)	1	0	0	0.50	0.50	0	0	1	2	1
MOVSD 0x38(%R10,%RDX,1),%XMM3	1	0	0	0.50	0.50	0	0	0	5	0.50
MOVSD 0x12898(%R14),%XMM2	1	0	0	0.50	0.50	0	0	0	5	0.50
MULSD %XMM3,%XMM2	1	0.50	0.50	0	0	0	0	0	6	0.50
MULSD %XMM5,%XMM2	1	0.50	0.50	0	0	0	0	0	6	0.50

MAQAO CQA: Code Quality Analyzer
Introduction

Goal: **Assist developers** in improving code performance

Features:

- Evaluates the **quality** of the compiler generated code
- Returns **hints and workarounds** to improve quality
- Focuses on **loops**
  - In HPC most of the time is spent in loops
- Targets **compute-bound** codes

Static analysis:

- Requires **no execution** of the application
- Allows **cross-analysis**

MAQAO CQA: Code Quality Analyzer
Processor Architecture: Core level

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

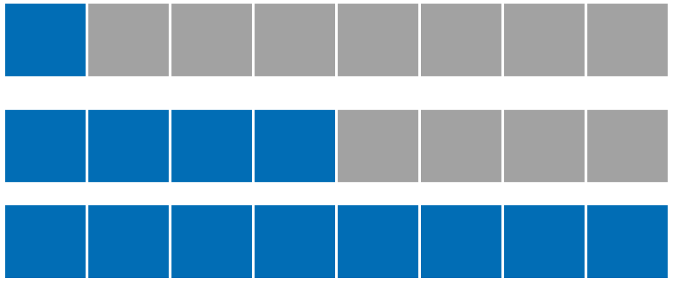
Main elements of analysis:

- Peak performance
- Execution pipeline
- Resources/Functional units

Key performance levers for core level efficiency:

- Vectorizing
- Avoiding high latency instructions if possible
- Having the compiler generate an efficient code

Same instruction – Same cost



Process up to 8X data

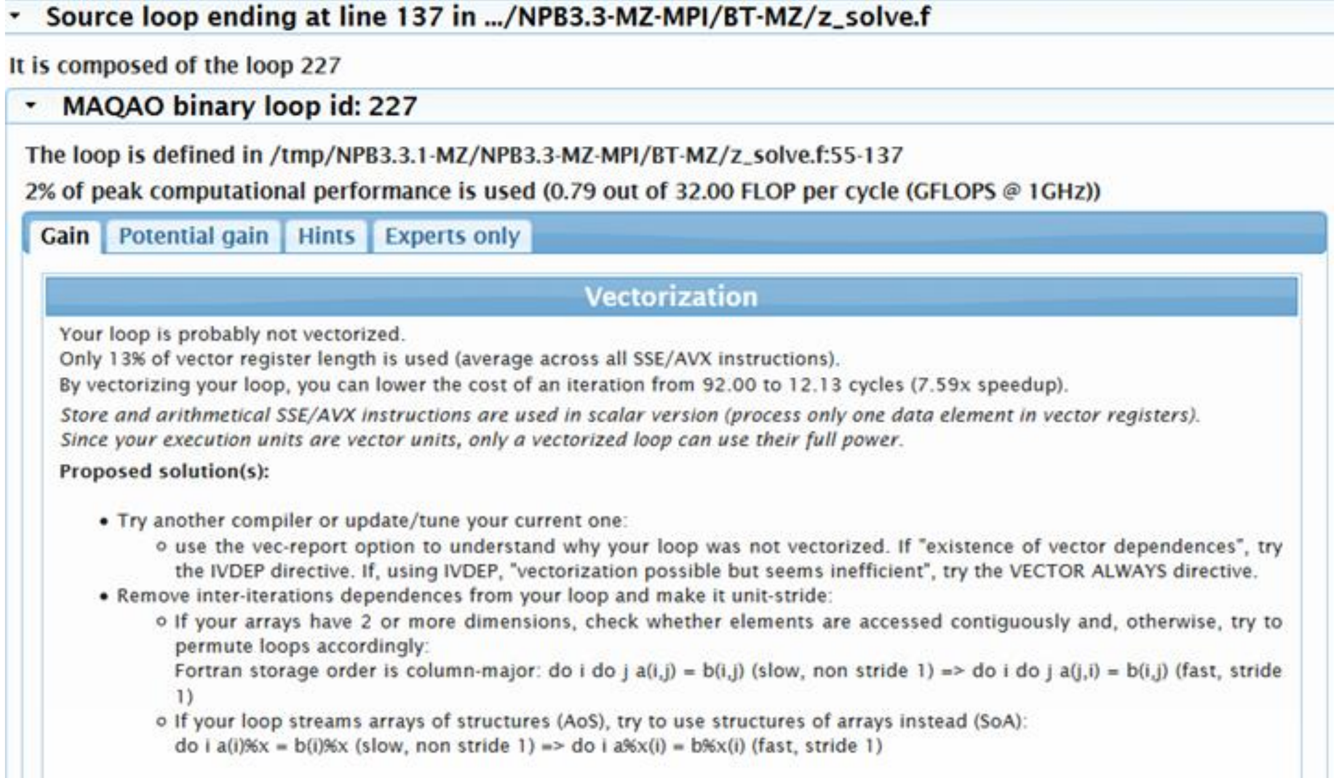
MAQAO CQA: Code Quality Analyzer *Output*

High level reports:

- Reference to the source code
- Bottleneck description
- Hints to improve performance
- Reports categorized by confidence level
  - gain, potential gain

Low level reports for performance experts

- Assembly-level
- Instructions cycles costs
- Instructions dispatch predictions



Source loop ending at line 137 in .../NPB3.3-MZ-MPI/BT-MZ/z_solve.f

It is composed of the loop 227

MAQAO binary loop id: 227

The loop is defined in /tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ/z_solve.f:55-137

2% of peak computational performance is used (0.79 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

Vectorization

Your loop is probably not vectorized.
Only 13% of vector register length is used (average across all SSE/AVX instructions).
By vectorizing your loop, you can lower the cost of an iteration from 92.00 to 12.13 cycles (7.59x speedup).

Store and arithmetical SSE/AVX instructions are used in scalar version (process only one data element in vector registers).
Since your execution units are vector units, only a vectorized loop can use their full power.

Proposed solution(s):

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:
    Fortran storage order is column-major: do i do j a(i,j) = b(i,j) (slow, non stride 1) => do i do j a(j,i) = b(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
    do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

MAQAO CQA: Code Quality Analyzer *Compiler and programmer hints*

Compiler can be driven using flags and pragmas:

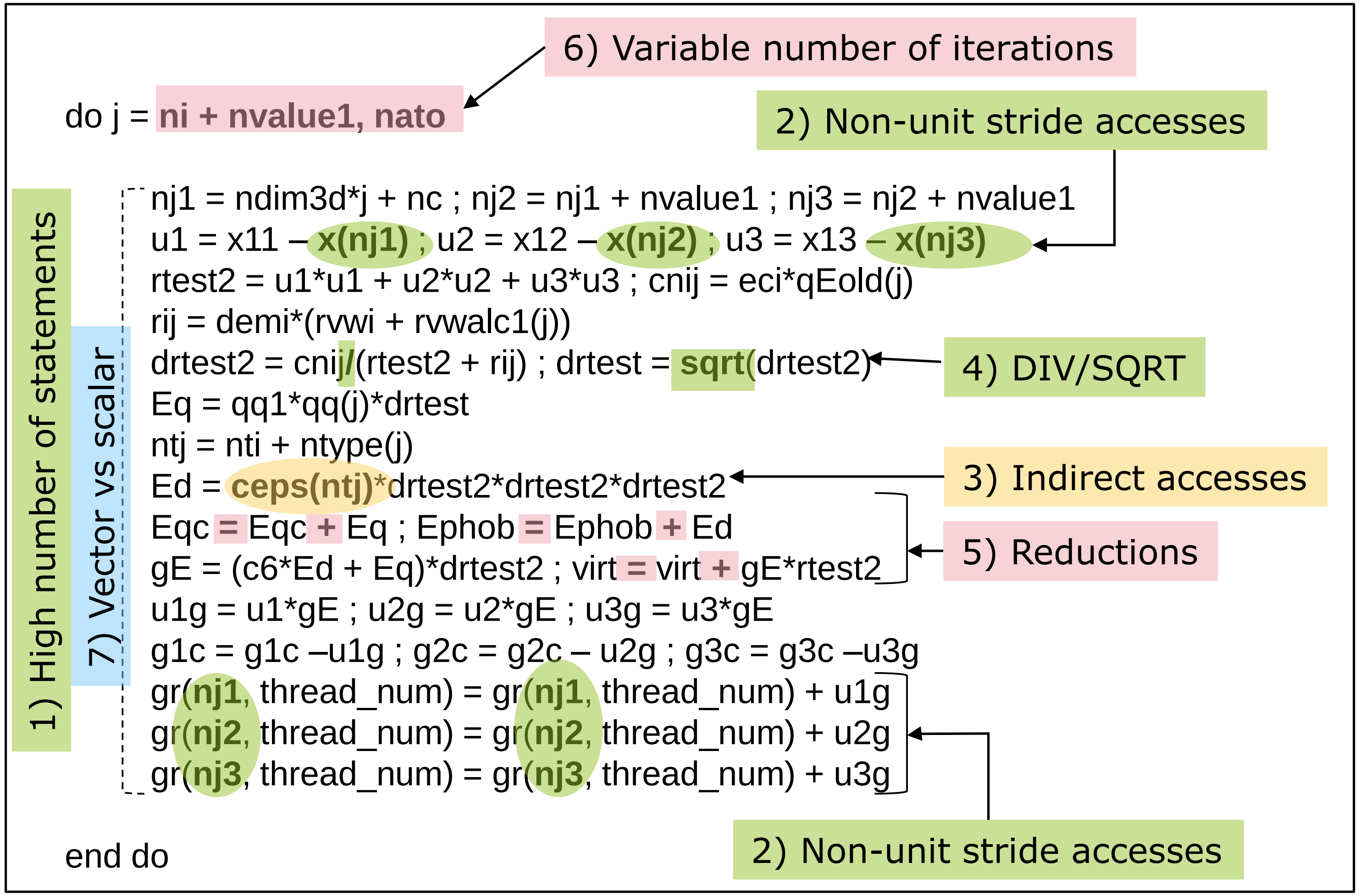
- Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
- Forcing optimization (unrolling, vectorization, alignment, …)
- Bypassing conservative behaviour when possible (e.g. 1/X precision)

Implementation changes:

- Improve data access
  - Loop interchange
  - Changing loop strides
- Avoid instructions with high latency

MAQAO CQA: Code Quality Analyzer
Application to motivating example

Issues identified by CQA



```
do j = ni + nvalue1, nato
    nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
    u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
    rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
    rij = demi*(rvwi + rvwalc1(j))
    drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
    Eq = qq1*qq(j)*drtest
    ntj = nti + ntype(j)
    Ed = ceps(ntj)*drtest2*drtest2*drtest2
    Eqc = Eqc + Eq ; Ephob = Ephob + Ed
    gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
    u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
    g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
    gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
    gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
    gr(nj3, thread_num) = gr(nj3, thread_num) + u3g

end do
```

6) Variable number of iterations

2) Non-unit stride accesses

4) DIV/SQRT

3) Indirect accesses

5) Reductions

2) Non-unit stride accesses

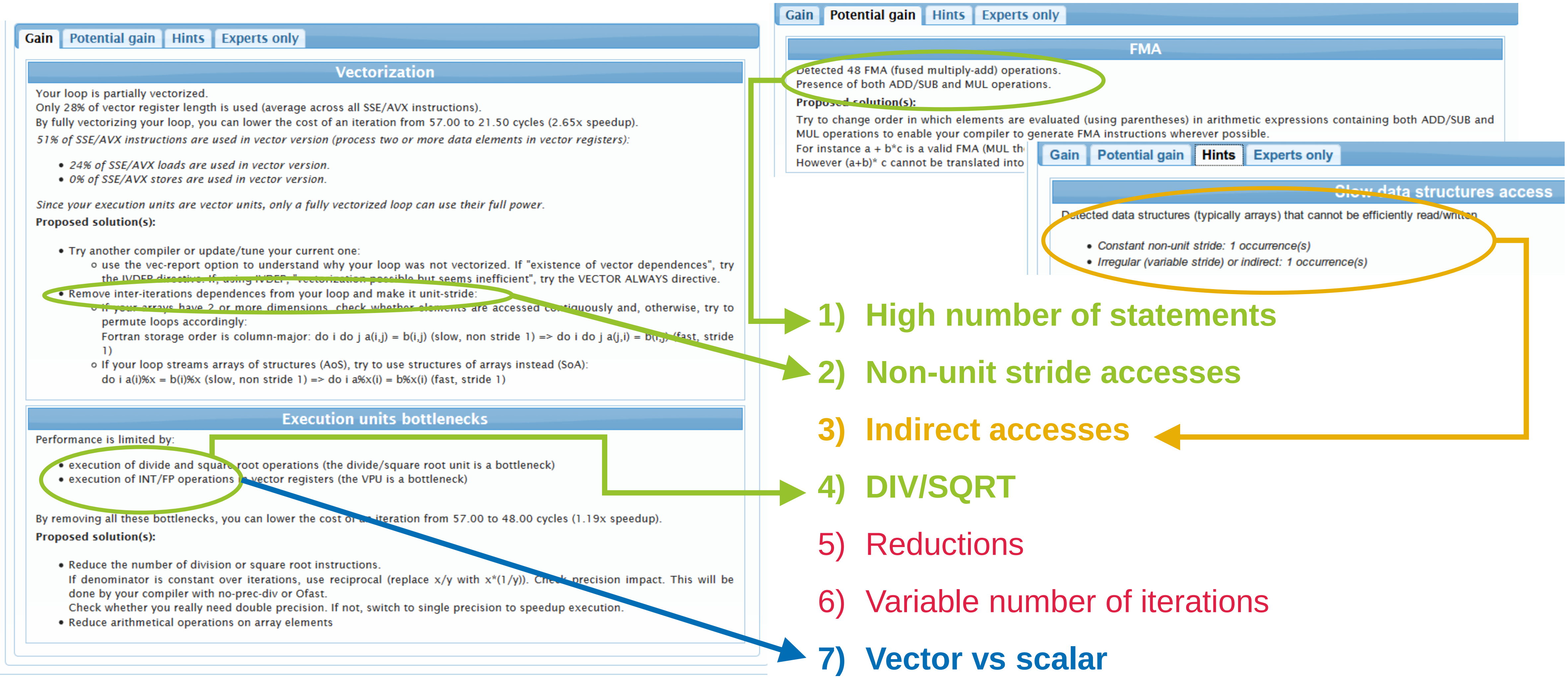
1) High number of statements

7) Vector vs scalar

CQA can detect and provide hints to resolve most of the identified issues:

1) **High number of statements**
2) **Non-unit stride accesses**
3) **Indirect accesses**
4) **DIV/SQRT**
5) Reductions
6) Variable number of iterations
7) **Vector vs scalar**

MAQAO CQA: Code Quality Analyzer
Application to motivating example



Vectorization

Your loop is partially vectorized.
Only 28% of vector register length is used (average across all SSE/AVX instructions).
By fully vectorizing your loop, you can lower the cost of an iteration from 57.00 to 21.50 cycles (2.65x speedup).
51% of SSE/AVX instructions are used in vector version (process two or more data elements in vector registers):

- *24% of SSE/AVX loads are used in vector version.*
- *0% of SSE/AVX stores are used in vector version.*

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

Proposed solution(s):

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:
    Fortran storage order is column-major: do i do j a(i,j) = b(i,j) (slow, non stride 1) => do i do j a(j,i) = b(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
    do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

Execution units bottlenecks

Performance is limited by:

- execution of divide and square root operations (the divide/square root unit is a bottleneck)
- execution of INT/FP operations on vector registers (the VPU is a bottleneck)

By removing all these bottlenecks, you can lower the cost of an iteration from 57.00 to 48.00 cycles (1.19x speedup).

Proposed solution(s):

- Reduce the number of division or square root instructions.
  If denominator is constant over iterations, use reciprocal (replace x/y with x*(1/y)). Check precision impact. This will be done by your compiler with no-prec-div or Ofast.
  Check whether you really need double precision. If not, switch to single precision to speedup execution.
- Reduce arithmetical operations on array elements

FMA

Detected 48 FMA (fused multiply-add) operations.
Presence of both ADD/SUB and MUL operations.

Proposed solution(s):

Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible.
For instance a + b*c is a valid FMA (MUL th
However (a+b)* c cannot be translated into

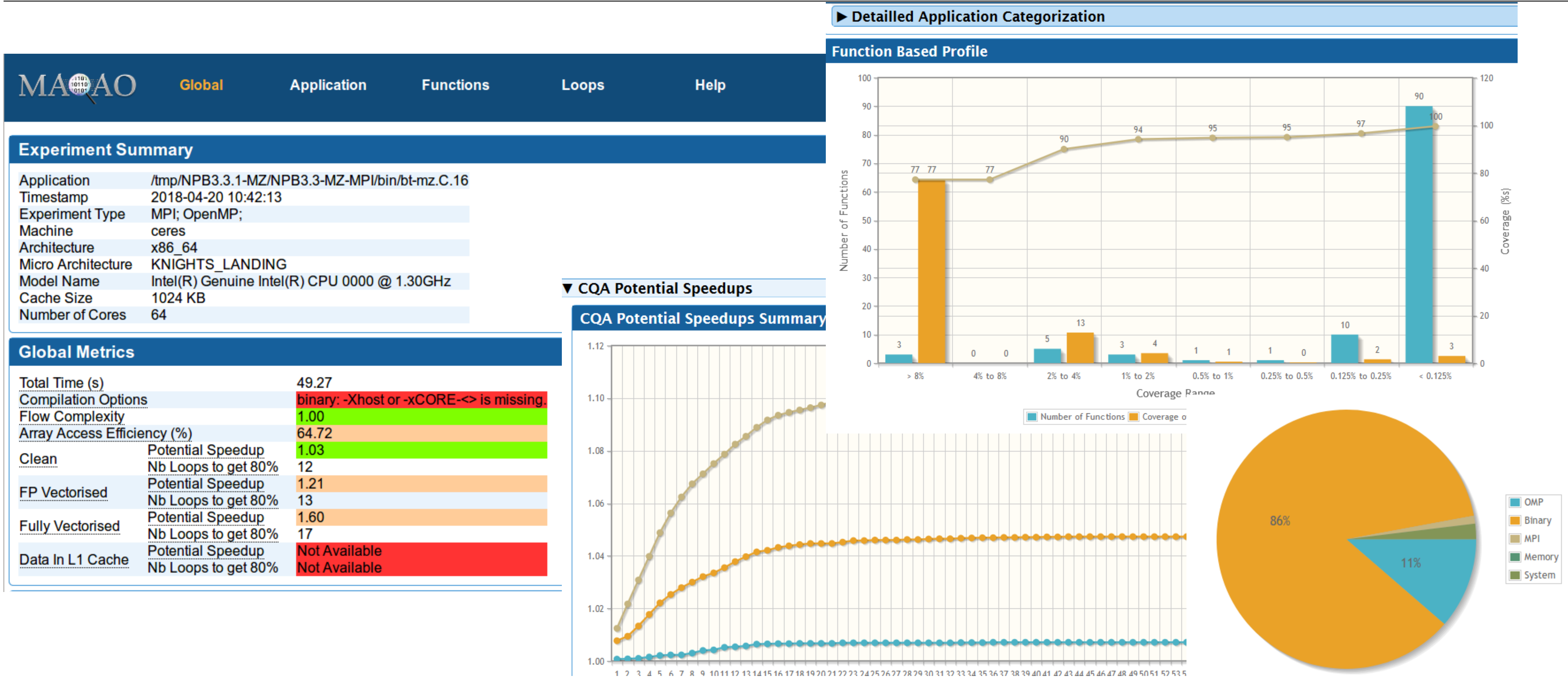
Slow data structures access

Detected data structures (typically arrays) that cannot be efficiently read/written

- *Constant non-unit stride: 1 occurrence(s)*
- *Irregular (variable stride) or indirect: 1 occurrence(s)*

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

MAQAO ONE View: Performance View Aggregator



Experiment Summary

Application	/tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/bin/bt-mz.C.16
Timestamp	2018-04-20 10:42:13
Experiment Type	MPI; OpenMP;
Machine	ceres
Architecture	x86_64
Micro Architecture	KNIGHTS_LANDING
Model Name	Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
Cache Size	1024 KB
Number of Cores	64

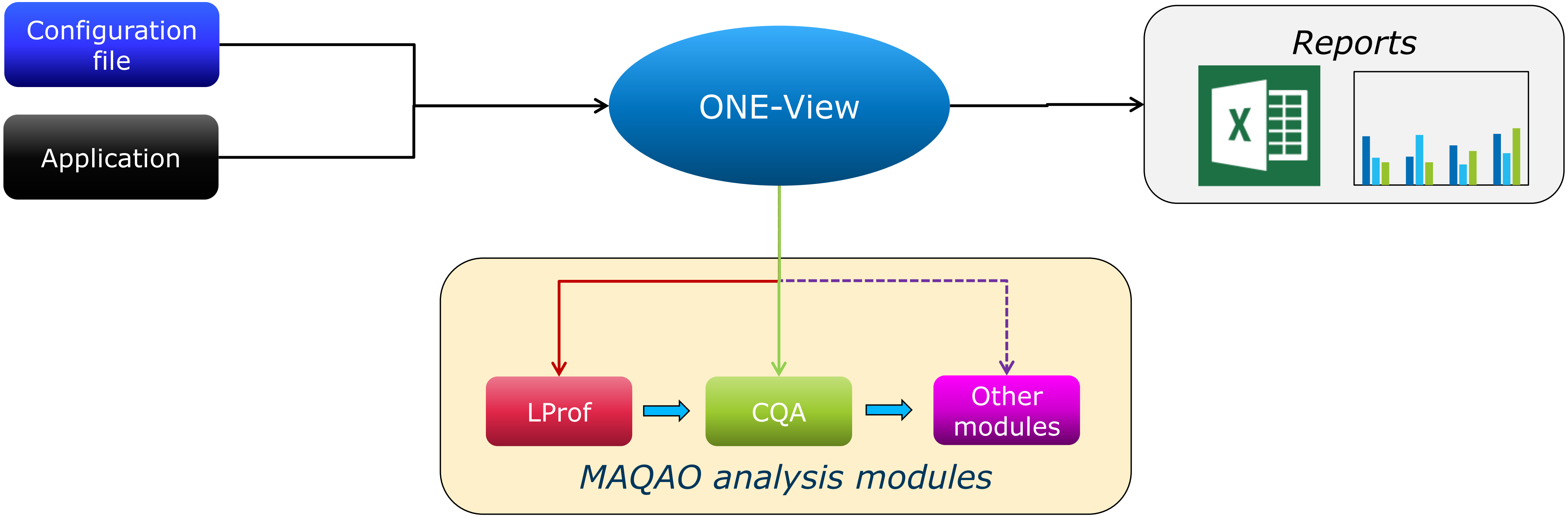
Global Metrics

Total Time (s)		49.27
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		64.72
Clean	Potential Speedup	1.03
	Nb Loops to get 80%	12
FP Vectorised	Potential Speedup	1.21
	Nb Loops to get 80%	13
Fully Vectorised	Potential Speedup	1.60
	Nb Loops to get 80%	17
Data In L1 Cache	Potential Speedup	Not Available
	Nb Loops to get 80%	Not Available

MAQAO ONE View: Performance View Aggregator
Introduction

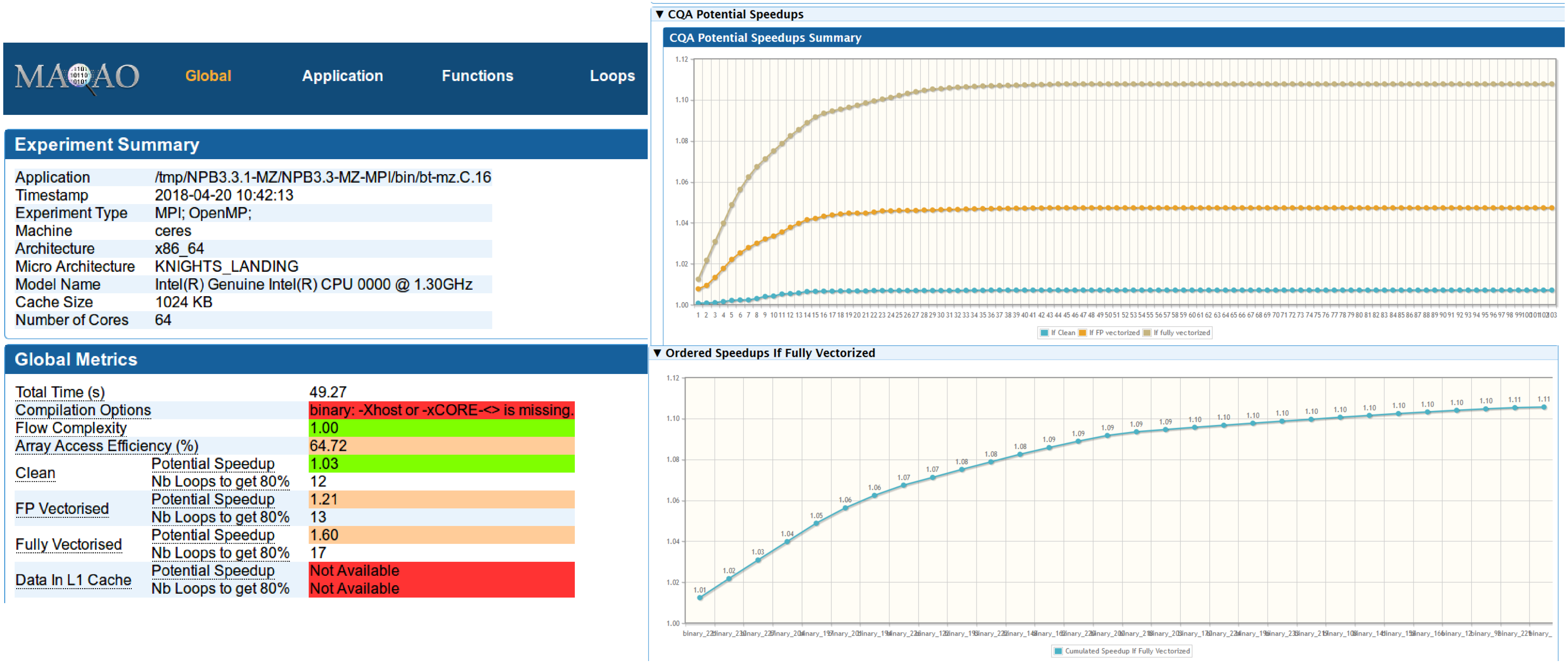
Automating the full analysis process

- Invocation of the MAQAO modules
- Generation of aggregated performance views as HTML or XLS graphs



MAQAO ONE View: Performance View Aggregator

GUI sample: Global View



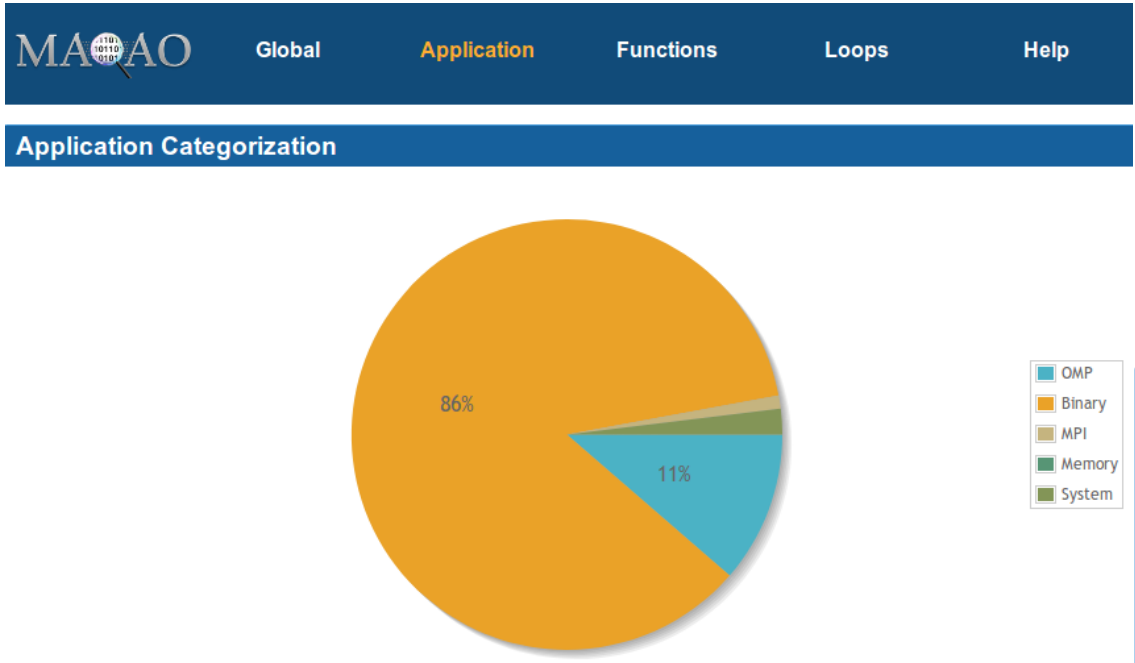
Experiment Summary

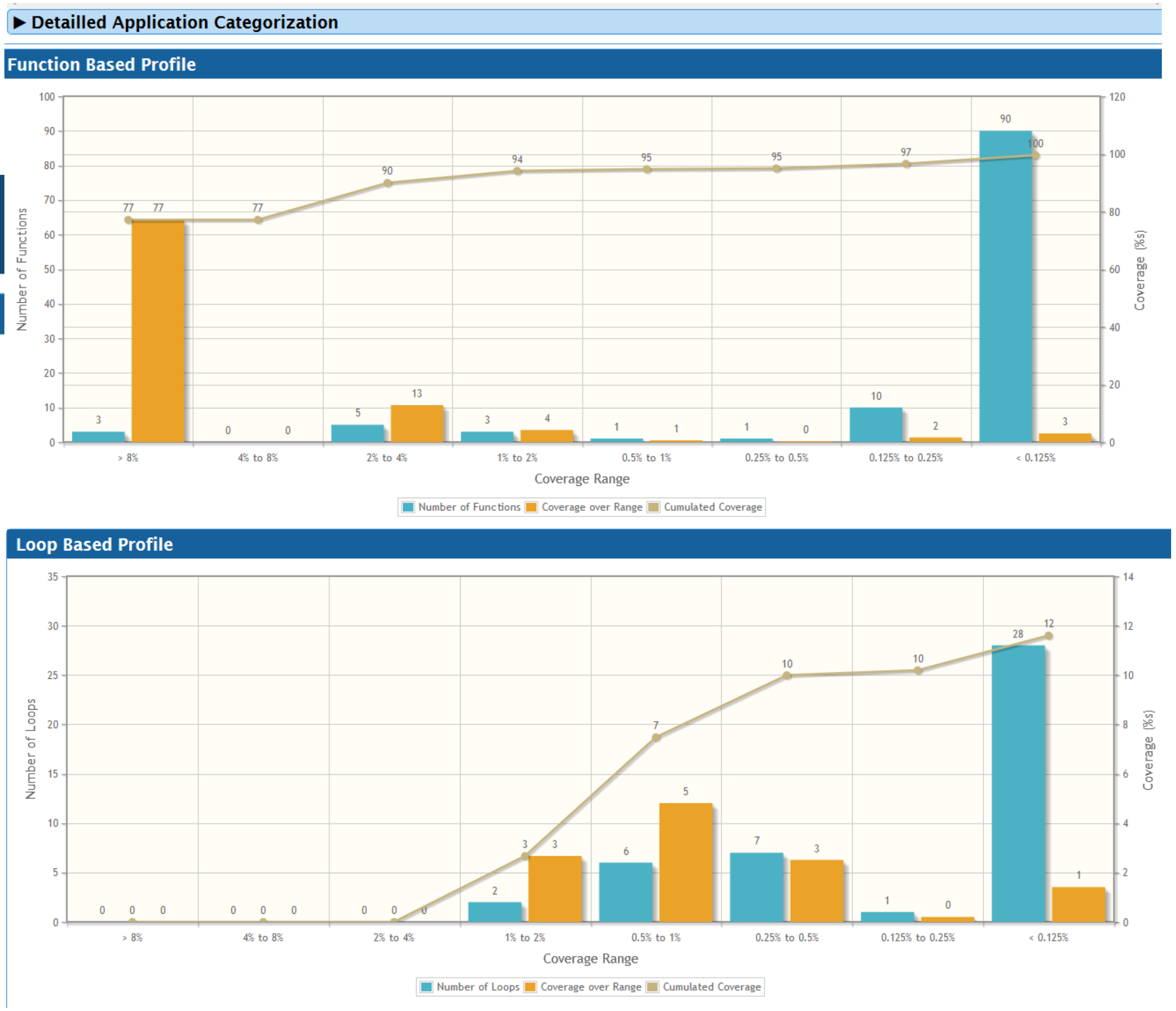
Application	/tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/bin/bt-mz.C.16
Timestamp	2018-04-20 10:42:13
Experiment Type	MPI; OpenMP;
Machine	ceres
Architecture	x86_64
Micro Architecture	KNIGHTS_LANDING
Model Name	Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
Cache Size	1024 KB
Number of Cores	64

Global Metrics

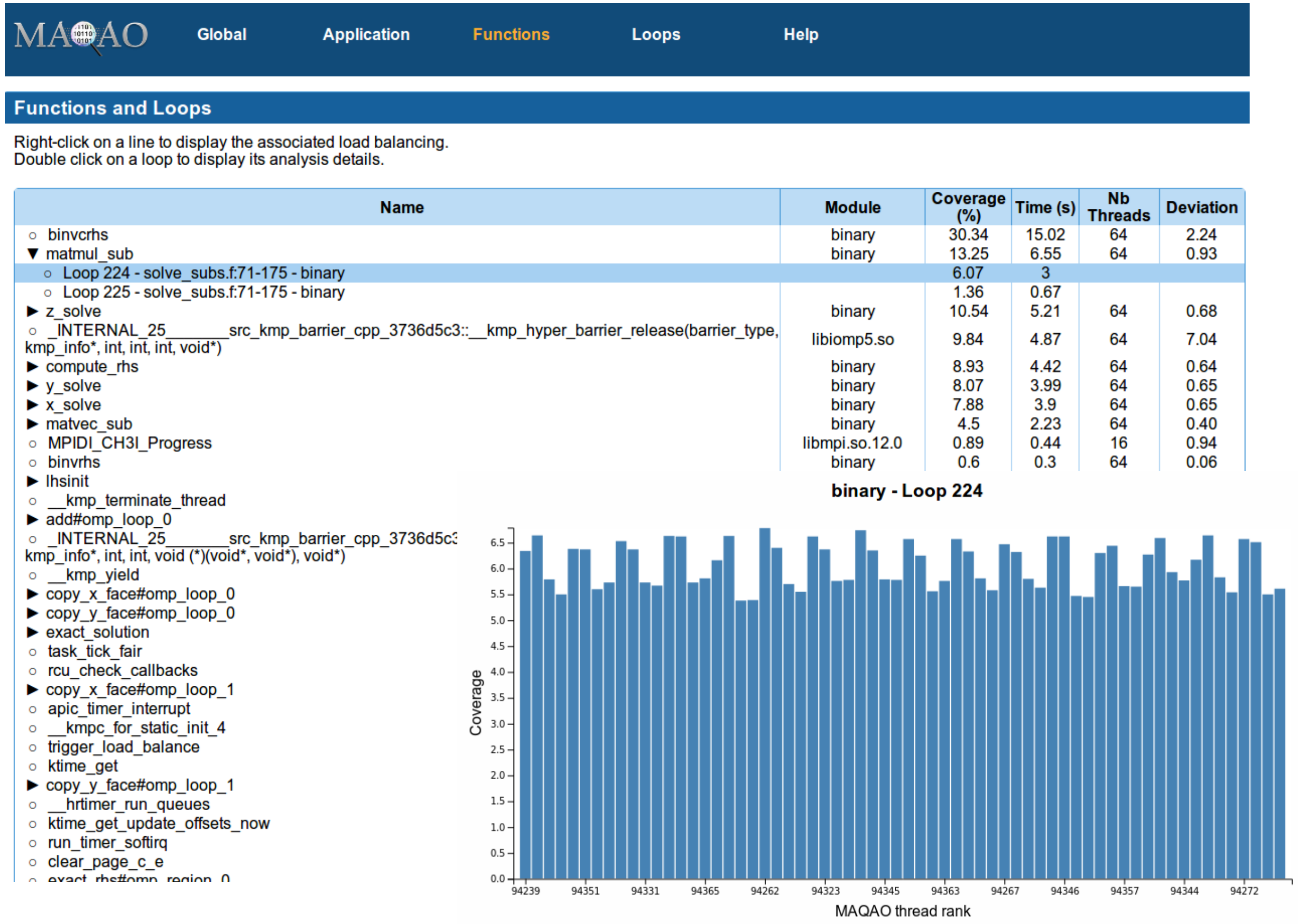
Total Time (s)		49.27
Compilation Options		binary: -Xhost or -xCORE-<> is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		64.72
Clean	Potential Speedup	1.03
	Nb Loops to get 80%	12
FP Vectorised	Potential Speedup	1.21
	Nb Loops to get 80%	13
Fully Vectorised	Potential Speedup	1.60
	Nb Loops to get 80%	17
Data In L1 Cache	Potential Speedup	Not Available
	Nb Loops to get 80%	Not Available

MAQAO ONE View: Performance View Aggregator
GUI sample: Application Characteristics





MAQAO ONE View: Performance View Aggregator
GUI sample: Functions and Loops Views

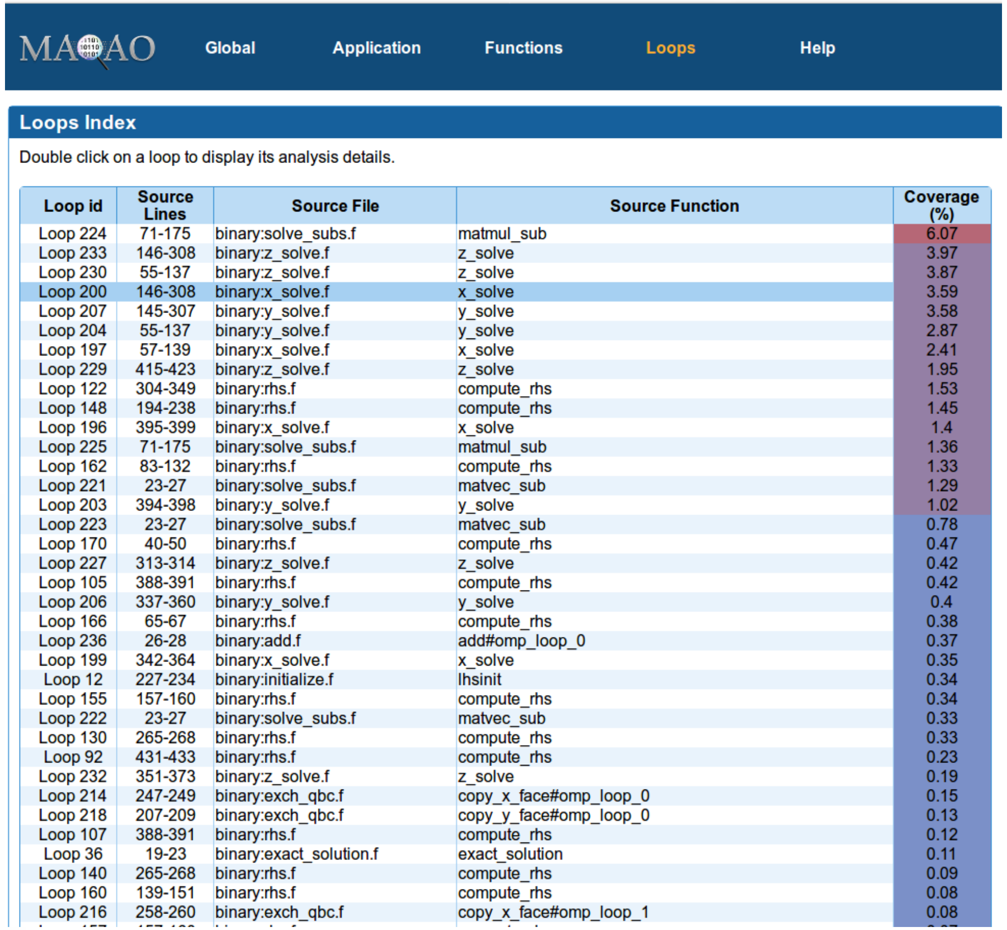


Functions and Loops

Right-click on a line to display the associated load balancing.
Double click on a loop to display its analysis details.

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation
○ binvcrhs	binary	30.34	15.02	64	2.24
▼ matmul_sub	binary	13.25	6.55	64	0.93
○ Loop 224 - solve_subs.f:71-175 - binary		6.07	3		
○ Loop 225 - solve_subs.f:71-175 - binary		1.36	0.67		
▶ z_solve	binary	10.54	5.21	64	0.68
○ _INTERNAL_25_______src_kmp_barrier_cpp_3736d5c3::__kmp_hyper_barrier_release(barrier_type, kmp_info*, int, int, int, void*)	libiomp5.so	9.84	4.87	64	7.04
▶ compute_rhs	binary	8.93	4.42	64	0.64
▶ y_solve	binary	8.07	3.99	64	0.65
▶ x_solve	binary	7.88	3.9	64	0.65
▶ matvec_sub	binary	4.5	2.23	64	0.40
○ MPIDI_CH3I_Progress	libmpi.so.12.0	0.89	0.44	16	0.94
○ binvrhs	binary	0.6	0.3	64	0.06
▶ lhsinit					
○ __kmp_terminate_thread					
▶ add#omp_loop_0					
○ _INTERNAL_25_______src_kmp_barrier_cpp_3736d5c3 kmp_info*, int, int, void (*)(void*, void*), void*)					
○ __kmp_yield					
▶ copy_x_face#omp_loop_0					
▶ copy_y_face#omp_loop_0					
▶ exact_solution					
○ task_tick_fair					
○ rcu_check_callbacks					
▶ copy_x_face#omp_loop_1					
○ apic_timer_interrupt					
○ __kmpc_for_static_init_4					
○ trigger_load_balance					
○ ktime_get					
▶ copy_y_face#omp_loop_1					
○ __hrtimer_run_queues					
○ ktime_get_update_offsets_now					
○ run_timer_softirq					
○ clear_page_c_e					
○ exact_rhs#omp_region_0					

binary - Loop 224

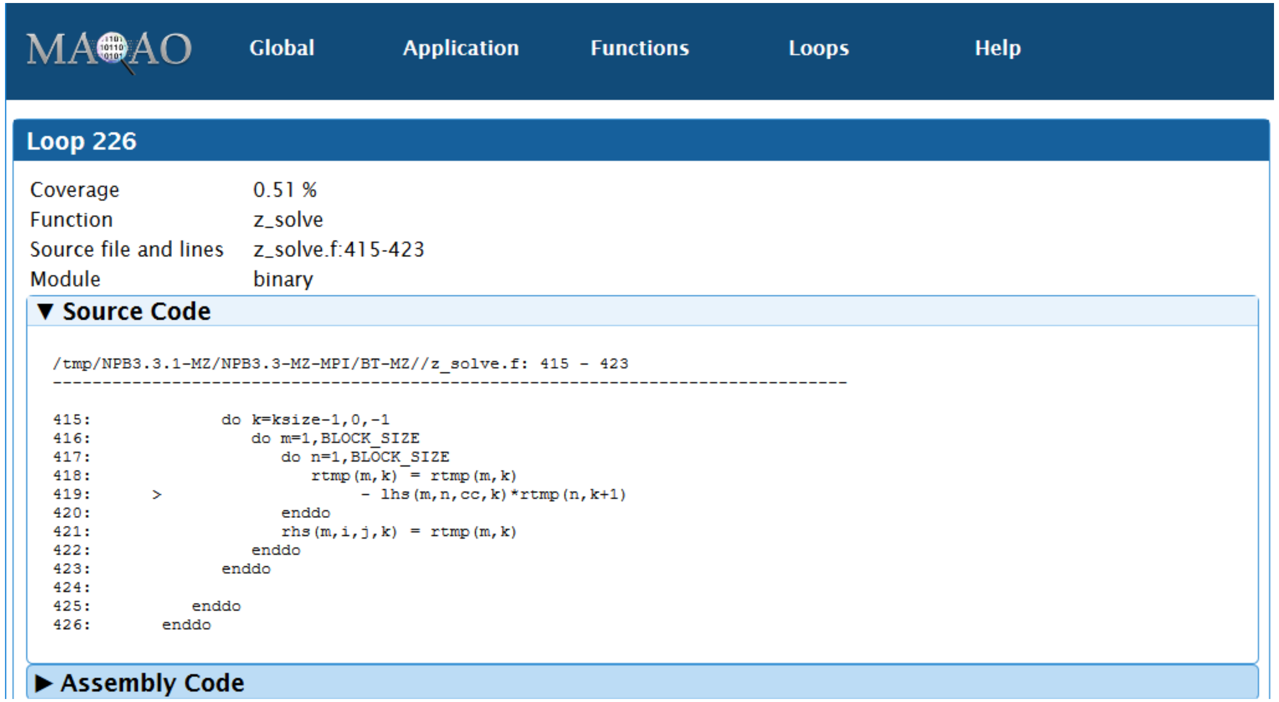


Loops Index

Double click on a loop to display its analysis details.

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 224	71-175	binary:solve_subs.f	matmul_sub	6.07
Loop 233	146-308	binary:z_solve.f	z_solve	3.97
Loop 230	55-137	binary:z_solve.f	z_solve	3.87
Loop 200	146-308	binary:x_solve.f	x_solve	3.59
Loop 207	145-307	binary:y_solve.f	y_solve	3.58
Loop 204	55-137	binary:y_solve.f	y_solve	2.87
Loop 197	57-139	binary:x_solve.f	x_solve	2.41
Loop 229	415-423	binary:z_solve.f	z_solve	1.95
Loop 122	304-349	binary:rhs.f	compute_rhs	1.53
Loop 148	194-238	binary:rhs.f	compute_rhs	1.45
Loop 196	395-399	binary:x_solve.f	x_solve	1.4
Loop 225	71-175	binary:solve_subs.f	matmul_sub	1.36
Loop 162	83-132	binary:rhs.f	compute_rhs	1.33
Loop 221	23-27	binary:solve_subs.f	matvec_sub	1.29
Loop 203	394-398	binary:y_solve.f	y_solve	1.02
Loop 223	23-27	binary:solve_subs.f	matvec_sub	0.78
Loop 170	40-50	binary:rhs.f	compute_rhs	0.47
Loop 227	313-314	binary:z_solve.f	z_solve	0.42
Loop 105	388-391	binary:rhs.f	compute_rhs	0.42
Loop 206	337-360	binary:y_solve.f	y_solve	0.4
Loop 166	65-67	binary:rhs.f	compute_rhs	0.38
Loop 236	26-28	binary:add.f	add#omp_loop_0	0.37
Loop 199	342-364	binary:x_solve.f	x_solve	0.35
Loop 12	227-234	binary:initialize.f	lhsinit	0.34
Loop 155	157-160	binary:rhs.f	compute_rhs	0.34
Loop 222	23-27	binary:solve_subs.f	matvec_sub	0.33
Loop 130	265-268	binary:rhs.f	compute_rhs	0.33
Loop 92	431-433	binary:rhs.f	compute_rhs	0.23
Loop 232	351-373	binary:z_solve.f	z_solve	0.19
Loop 214	247-249	binary:exch_qbc.f	copy_x_face#omp_loop_0	0.15
Loop 218	207-209	binary:exch_qbc.f	copy_y_face#omp_loop_0	0.13
Loop 107	388-391	binary:rhs.f	compute_rhs	0.12
Loop 36	19-23	binary:exact_solution.f	exact_solution	0.11
Loop 140	265-268	binary:rhs.f	compute_rhs	0.09
Loop 160	139-151	binary:rhs.f	compute_rhs	0.08
Loop 216	258-260	binary:exch_qbc.f	copy_x_face#omp_loop_1	0.08

MAQAO ONE View: Performance View Aggregator
GUI sample: CQA Output



Loop 226

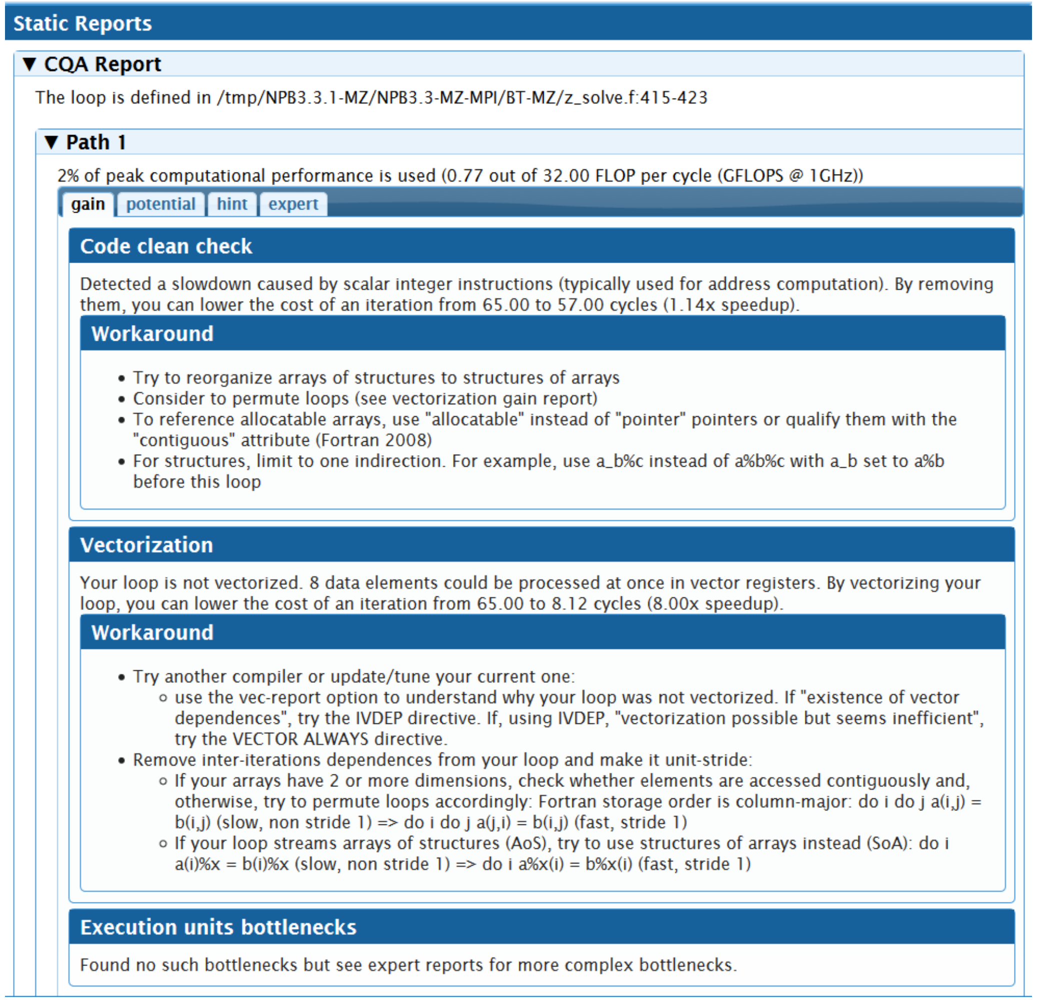
Coverage	0.51 %
Function	z_solve
Source file and lines	z_solve.f:415-423
Module	binary

▼ Source Code

```
/tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ//z_solve.f: 415 - 423
--------------------------------------------------------------------------------

415:              do k=ksize-1,0,-1
416:                 do m=1,BLOCK_SIZE
417:                    do n=1,BLOCK_SIZE
418:                       rtmp(m,k) = rtmp(m,k)
419:      >                     - lhs(m,n,cc,k)*rtmp(n,k+1)
420:                    enddo
421:                    rhs(m,i,j,k) = rtmp(m,k)
422:                 enddo
423:              enddo
424:
425:           enddo
426:        enddo
```

▶ Assembly Code



Static Reports

▼ CQA Report

The loop is defined in /tmp/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ/z_solve.f:415-423

▼ Path 1

2% of peak computational performance is used (0.77 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 65.00 to 57.00 cycles (1.14x speedup).

Workaround

- Try to reorganize arrays of structures to structures of arrays
- Consider to permute loops (see vectorization gain report)
- To reference allocatable arrays, use "allocatable" instead of "pointer" pointers or qualify them with the "contiguous" attribute (Fortran 2008)
- For structures, limit to one indirection. For example, use a_b%c instead of a%b%c with a_b set to a%b before this loop

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 65.00 to 8.12 cycles (8.00x speedup).

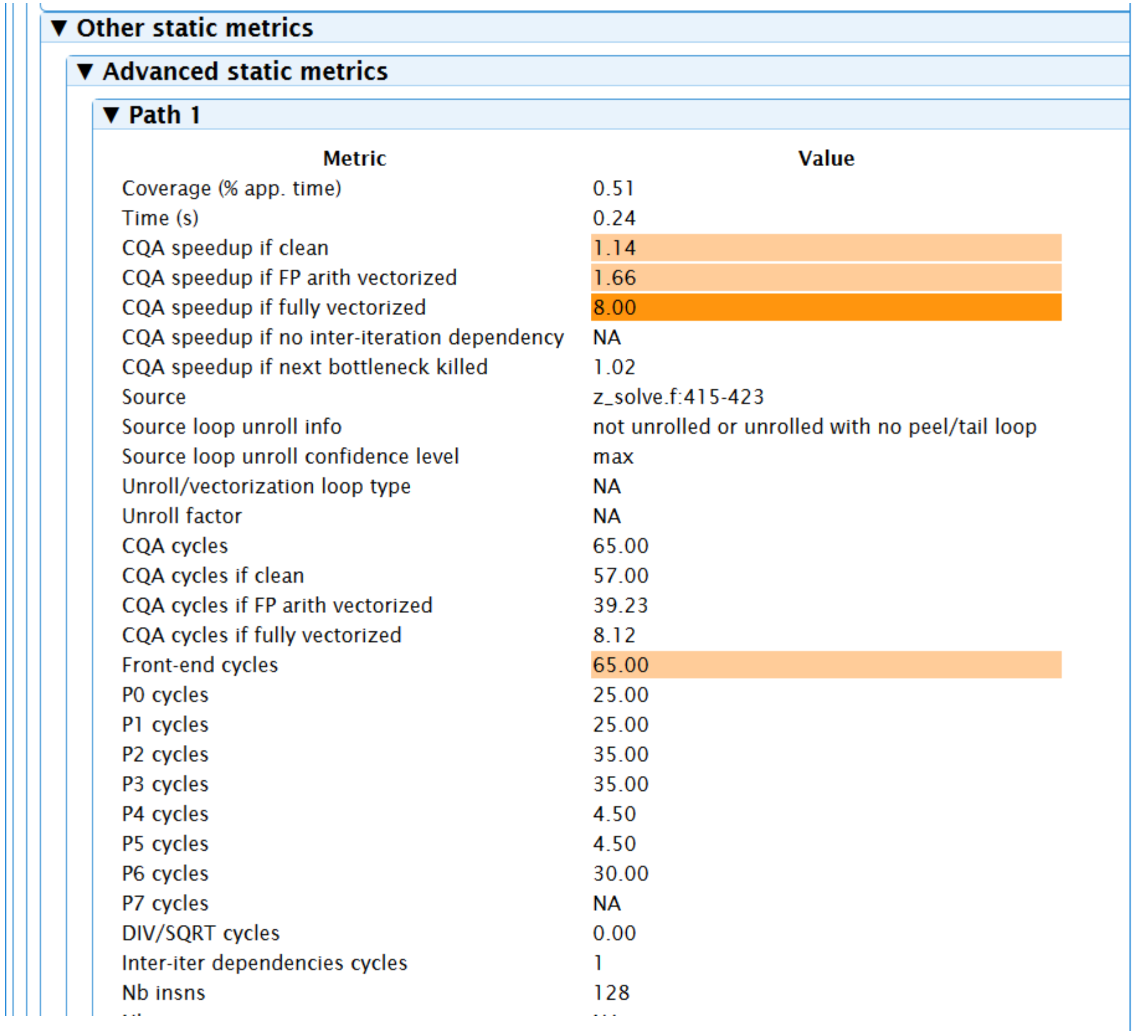
Workaround

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: Fortran storage order is column-major: do i do j a(i,j) = b(i,j) (slow, non stride 1) => do i do j a(j,i) = b(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

Execution units bottlenecks

Found no such bottlenecks but see expert reports for more complex bottlenecks.

MAQAO ONE View: Performance View Aggregator
GUI sample: Advanced Loop Metrics

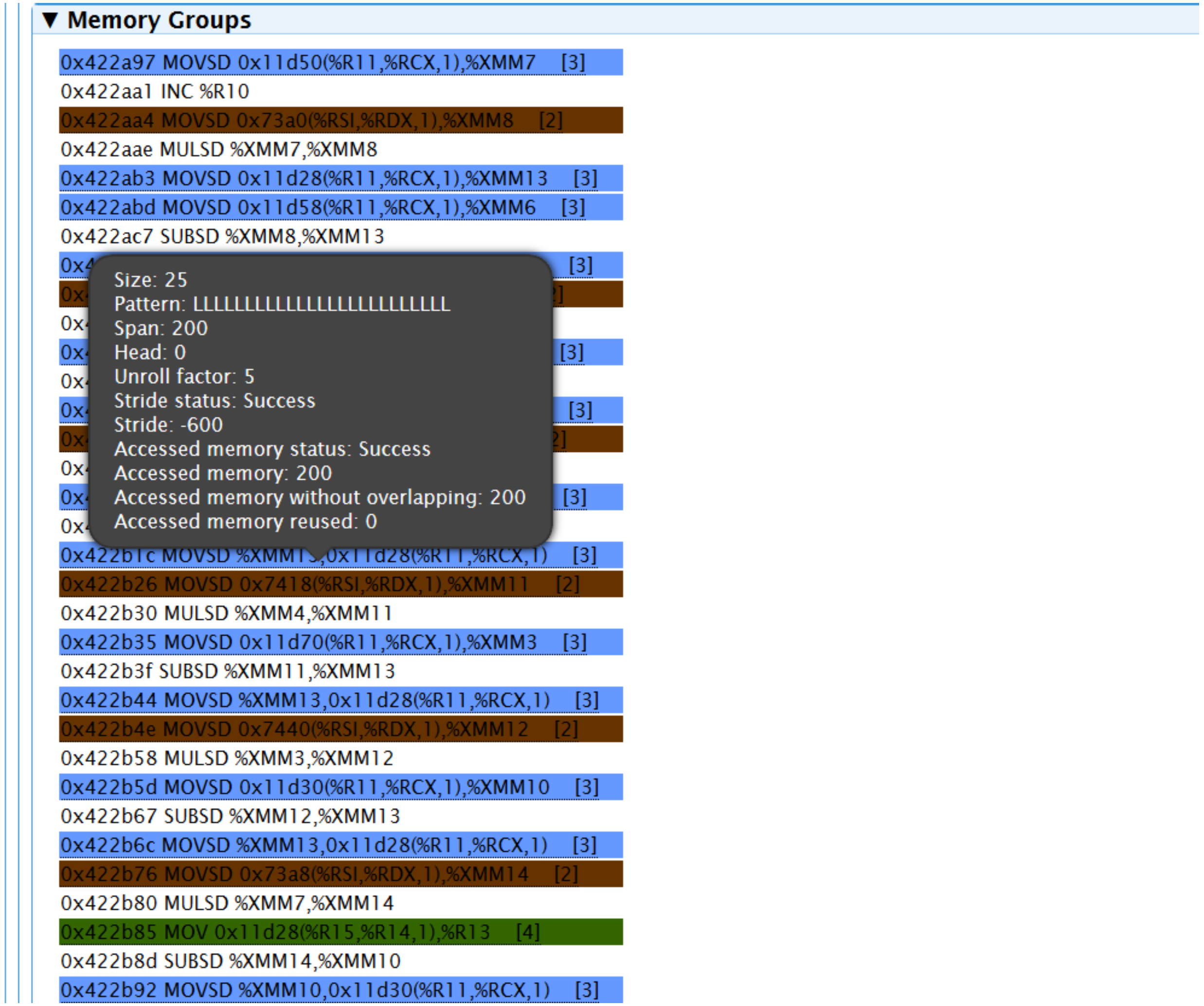


▼ Other static metrics

▼ Advanced static metrics

▼ Path 1

Metric	Value
Coverage (% app. time)	0.51
Time (s)	0.24
CQA speedup if clean	1.14
CQA speedup if FP arith vectorized	1.66
CQA speedup if fully vectorized	8.00
CQA speedup if no inter-iteration dependency	NA
CQA speedup if next bottleneck killed	1.02
Source	z_solve.f:415-423
Source loop unroll info	not unrolled or unrolled with no peel/tail loop
Source loop unroll confidence level	max
Unroll/vectorization loop type	NA
Unroll factor	NA
CQA cycles	65.00
CQA cycles if clean	57.00
CQA cycles if FP arith vectorized	39.23
CQA cycles if fully vectorized	8.12
Front-end cycles	65.00
P0 cycles	25.00
P1 cycles	25.00
P2 cycles	35.00
P3 cycles	35.00
P4 cycles	4.50
P5 cycles	4.50
P6 cycles	30.00
P7 cycles	NA
DIV/SQRT cycles	0.00
Inter-iter dependencies cycles	1
Nb insns	128

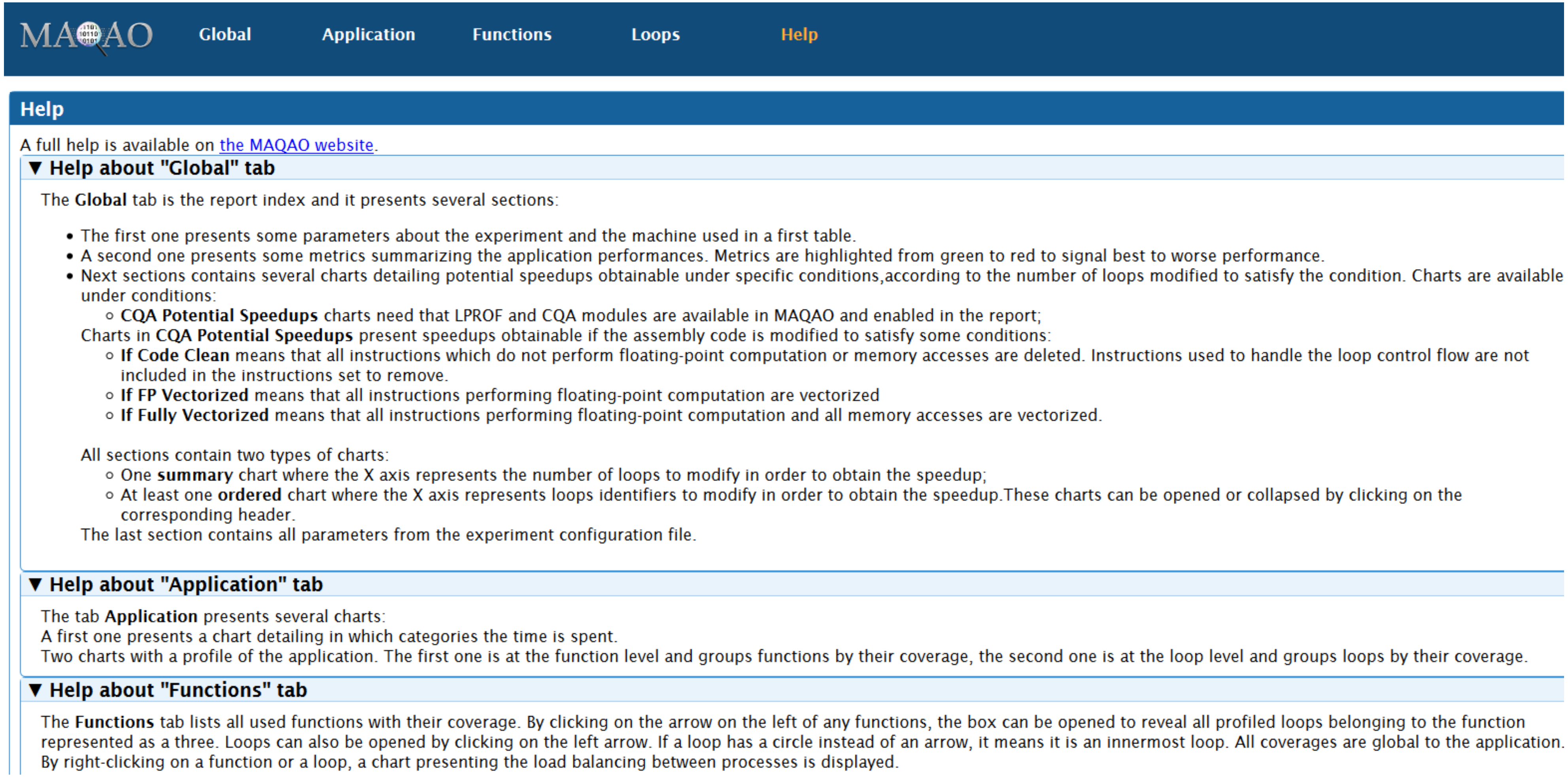


▼ Memory Groups

```
0x422a97 MOVSD 0x11d50(%R11,%RCX,1),%XMM7   [3]
0x422aa1 INC %R10
0x422aa4 MOVSD 0x73a0(%RSI,%RDX,1),%XMM8   [2]
0x422aae MULSD %XMM7,%XMM8
0x422ab3 MOVSD 0x11d28(%R11,%RCX,1),%XMM13   [3]
0x422abd MOVSD 0x11d58(%R11,%RCX,1),%XMM6   [3]
0x422ac7 SUBSD %XMM8,%XMM13
0x4                                          [3]
0x                                          ]
0x4
0x4                                          [3]
0x4
0x4                                          [3]
0x4                                          ]
0x4
0x4                                          [3]
0x4
0x422b1c MOVSD %XMM13,0x11d28(%R11,%RCX,1)   [3]
0x422b26 MOVSD 0x7418(%RSI,%RDX,1),%XMM11   [2]
0x422b30 MULSD %XMM4,%XMM11
0x422b35 MOVSD 0x11d70(%R11,%RCX,1),%XMM3   [3]
0x422b3f SUBSD %XMM11,%XMM13
0x422b44 MOVSD %XMM13,0x11d28(%R11,%RCX,1)   [3]
0x422b4e MOVSD 0x7440(%RSI,%RDX,1),%XMM12   [2]
0x422b58 MULSD %XMM3,%XMM12
0x422b5d MOVSD 0x11d30(%R11,%RCX,1),%XMM10   [3]
0x422b67 SUBSD %XMM12,%XMM13
0x422b6c MOVSD %XMM13,0x11d28(%R11,%RCX,1)   [3]
0x422b76 MOVSD 0x73a8(%RSI,%RDX,1),%XMM14   [2]
0x422b80 MULSD %XMM7,%XMM14
0x422b85 MOV 0x11d28(%R15,%R14,1),%R13   [4]
0x422b8d SUBSD %XMM14,%XMM10
0x422b92 MOVSD %XMM10,0x11d30(%R11,%RCX,1)   [3]
```

```
Size: 25
Pattern: LLLLLLLLLLLLLLLLLLLLLLLLL
Span: 200
Head: 0
Unroll factor: 5
Stride status: Success
Stride: -600
Accessed memory status: Success
Accessed memory: 200
Accessed memory without overlapping: 200
Accessed memory reused: 0
```

MAQAO ONE View: Performance View Aggregator
GUI sample: Help



Help

A full help is available on the MAQAO website.

▼ Help about "Global" tab

The **Global** tab is the report index and it presents several sections:

- The first one presents some parameters about the experiment and the machine used in a first table.
- A second one presents some metrics summarizing the application performances. Metrics are highlighted from green to red to signal best to worse performance.
- Next sections contains several charts detailing potential speedups obtainable under specific conditions,according to the number of loops modified to satisfy the condition. Charts are available under conditions:
  - **CQA Potential Speedups** charts need that LPROF and CQA modules are available in MAQAO and enabled in the report;

  Charts in **CQA Potential Speedups** present speedups obtainable if the assembly code is modified to satisfy some conditions:
  - **If Code Clean** means that all instructions which do not perform floating-point computation or memory accesses are deleted. Instructions used to handle the loop control flow are not included in the instructions set to remove.
  - **If FP Vectorized** means that all instructions performing floating-point computation are vectorized
  - **If Fully Vectorized** means that all instructions performing floating-point computation and all memory accesses are vectorized.

  All sections contain two types of charts:
  - One **summary** chart where the X axis represents the number of loops to modify in order to obtain the speedup;
  - At least one **ordered** chart where the X axis represents loops identifiers to modify in order to obtain the speedup.These charts can be opened or collapsed by clicking on the corresponding header.

  The last section contains all parameters from the experiment configuration file.

▼ Help about "Application" tab

The tab **Application** presents several charts:
A first one presents a chart detailing in which categories the time is spent.
Two charts with a profile of the application. The first one is at the function level and groups functions by their coverage, the second one is at the loop level and groups loops by their coverage.

▼ Help about "Functions" tab

The **Functions** tab lists all used functions with their coverage. By clicking on the arrow on the left of any functions, the box can be opened to reveal all profiled loops belonging to the function represented as a three. Loops can also be opened by clicking on the left arrow. If a loop has a circle instead of an arrow, it means it is an innermost loop. All coverages are global to the application. By right-clicking on a function or a loop, a chart presenting the load balancing between processes is displayed.

Thank you for your attention !

Questions ?