



# MAQAO

## Performance Analysis and Optimization Framework

Cédric Valensi (Université de Versailles Saint Quentin)

# 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
```

**Where are the bottlenecks?**

# 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 + rwalc1(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
```

1) High number of statements

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

***Which is the dominant one?***

**➔ Need analysis tools to identify performance issues**

# A multifaceted problem

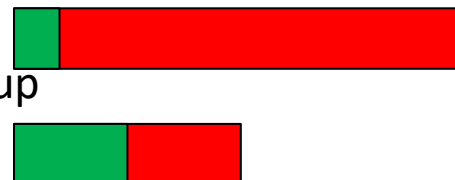


- What type of problems are we facing?
  - Identifying the dominant issues: Algorithm, implementation, parallelisation, compilation, ...
    - CPU or data access problems
  - Making the best use of the machine features
- What levers do we have to address them?
  - Compiler switches, Partial/full vectorization
  - Loop blocking/array restructuring, If removal, Full unroll
  - Binary transforms (prefetch)
  - ...



➔ **Need for dedicated and complementary tools**

- Which issues will be the most rewarding to fix?
  - 40% total time, expected 10% speedup
    - ➔ TOTAL IMPACT: 4% speedup
  - 20% total time, expected 50% speedup
    - ➔ TOTAL IMPACT: 10% speedup





## Nobody wants problems everybody wants solutions 😊

- Focusing on the knobs that code developers can operate:
  - Compiler flags and runtime settings
  - Code restructuring
  - Data restructuring
- Assisting the user in using these knobs
- ➔ In addition to pinpointing problems, guiding the user towards a way to address them.
- Philosophy: Analysis at Binary Level
  - Compiler optimizations increase the distance between the executed code and the source code
  - Source code instrumentation may prevent the compiler from applying certain transformations
  - Allows to be agnostic with regard to compiled source code language

➔ **What You Analyse Is What You Run**

# MAQAO: Modular Assembly Quality Analyzer and Optimizer



## Objectives:

- Characterizing performance of HPC applications
- Focusing on performance at the **core/node level**
- **Guiding users** through the optimization process
- Estimating return on investment (**R.O.I.**)

## Characteristics:

- **Modular tool** offering complementary views
- Support for **x86-64** and **aarch64** (beta version)
  - Working prototype for AMD GPU
- LGPL3 Open Source software
- Developed at UVSQ since 2004
- Binary release available as a **static executable**

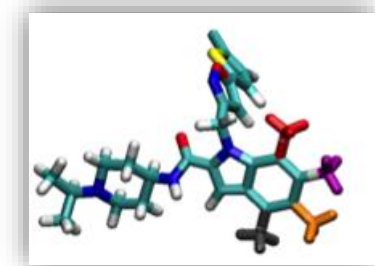
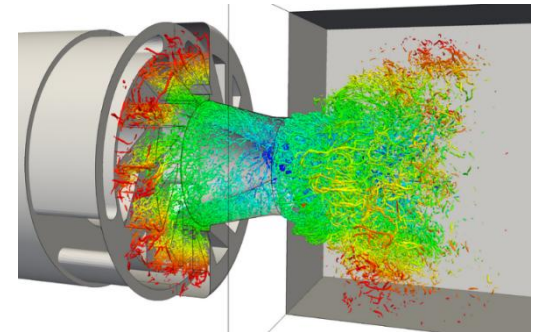
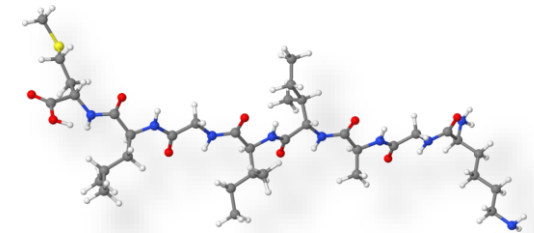


# Success stories



Optimizing industrial and academic HPC applications:

- QMC=CHEM (IRSAMC)
  - Quantum chemistry
  - Speedup: **> 3x**
    - Optimization: moved invocations of functions with identical parameters out of the loop body
- Yales2 (CORIA)
  - Computational fluid dynamics
  - Speedup: **up to 2.8x**
    - Optimization: removing double structure indirections
- Polaris (CEA)
  - Molecular dynamics
  - Speedup: **1.5x – 1.7x**
    - Optimization: enforcing loop vectorization through compiler directives
- AVBP (CERFACS)
  - Computational fluid dynamics
  - Speedup: **1.08x – 1.17x**
    - Replaced divisions by reciprocal multiplications
    - Complete unrolling of loops with a small number of iterations



# MAQAO team and collaborators



- MAQAO Team

- William Jalby, Prof.
- Cédric Valensi, Ph.D.
- Emmanuel Oseret, Ph.D.
- Mathieu Tribalat, M.Sc.Eng.
- Hugo Bolloré, M.Sc.Eng
- Kévin Camus, Eng.
- Lucas Neto, Eng.

- Collaborators

- David J. Kuck, Prof. (Intel US)
- Pablo de Oliveira, Prof. (UVSQ)
- Eric Petit, Ph.D. (Intel US)
- David C. Wong, Ph.D. (ARM US)
- Othman Bouizi, Ph.D. (Eviden)
- AbdelHafid Mazouz Ph.D.(Intel)
- Jeongnim Kim (Intel US)
- Aurélien Delval, Ph.D. Student (SiPearl)
- Nicolas Fond-Massany, Ph.D. Student (Safran)

- Past Team Members

- Denis Barthou, Prof. (Univ. Bordeaux)
- Andrés S. Charif-Rubial, Ph.D. (†)
- Jean-Thomas Acquaviva, Ph.D. (DDN)
- Souad Koliaï, Ph.D. (South Pole)
- Zakaria Bendifallah, Ph.D. (Eviden)
- Jean-Baptiste Le Reste, M.Sc.Eng. (AnotherBrain)
- Sylvain Henry, Ph.D. (InputOutput)
- Aleksandre Vardoshvili, M.Sc.Eng.
- Romain Pillot, Eng
- Youenn Lebras, Ph.D. (Noxant)
- Jäsper Salah Ibnamar, M.Sc.Eng.
- Max Hoffer, Eng. (Worldgrid)

- Past Collaborators

- Stéphane Zuckerman, Ph.D. (ENSEA)
- Julien Jaeger, Ph.D. (CEA DAM)
- Tipp Moseley, Ph.D. (Google)
- Jean-Christophe Beyler, Ph.D. (Google)
- José Noudohouenou, Ph.D. (AMD)

# More on MAQAO



**MAQAO website:** [www.maqao.org](http://www.maqao.org)

- Mirrors: [maqao.liparad.uvsq.fr](http://maqao.liparad.uvsq.fr), [maqao.exascale-computing.eu](http://maqao.exascale-computing.eu)
- Documentation: [www.maqao.org/documentation.html](http://www.maqao.org/documentation.html)
  - Tutorials for ONE View, LProf and CQA
  - Lua API documentation
- Latest release: [www.maqao.org/download.html](http://www.maqao.org/download.html)
  - Binary releases (2-3 per year)
  - Source code
- Publications around MAQAO: [www.maqao.org/publications.html](http://www.maqao.org/publications.html)
- Repository of MAQAO analyses: [datafront.maqao.org/public/](http://datafront.maqao.org/public/)
  - Mirrors: [datafront.liparad.uvsq.fr/public/](http://datafront.liparad.uvsq.fr/public/), [datafront.exascale-computing.eu/public/](http://datafront.exascale-computing.eu/public/)
- Email: [contact@maqao.org](mailto:contact@maqao.org)

# Useful notions



## SIMD/Vectorization/Data Parallelism

- Scalar pattern:  $a[i] = b[i] + c[i]$
- Vector pattern:  $a(i, i + 8) = b(i, i + 8) + c(i, i + 8)$
- Benefits : increases memory bandwidth and **IPC**
- Example implementations :
  - ARM : Neon, SVE
  - x86 : SSE, AVX, AVX512

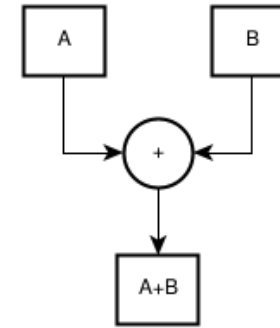
## FMA/MAC

- Fused-Multiply-Add
- Multiply-Accumulate

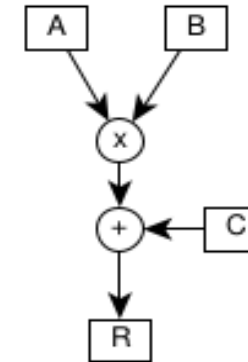
## Memory and caches

- Computations are in general faster than memory accesses
- Alignment/Contiguity of memory (x86) : **posix\_memalign, aligned\_alloc, ...**
- Caches: L1, L2, L3, ...

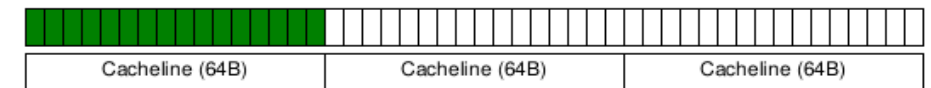
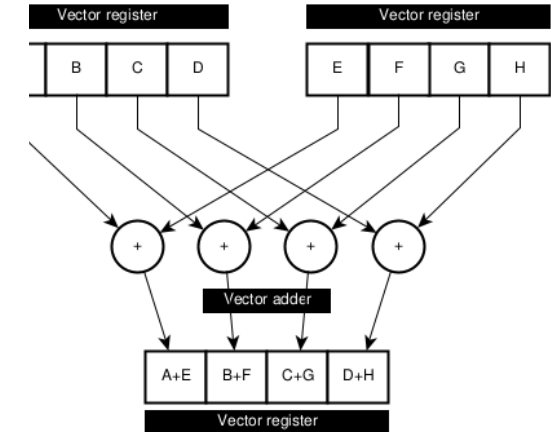
Scalar addition



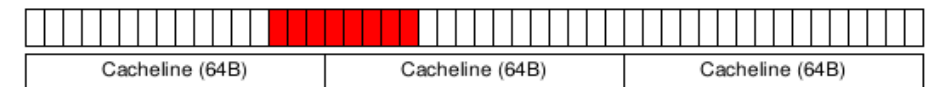
FMA



Vector addition



Aligned memory access



Crossing cacheline boundary

Unaligned memory access

# MAQAO Main Features



- Binary layer
  - Builds internal representation from binary
    - Construct high level structures (CFG, DDG, SSA, ...)
    - Links binary instructions to source code
    - ⚠ A single source loop can be compiled as multiple assembly loops → Affecting unique identifiers to loops
  - Allows patching through binary rewriting
- Profiling
  - **LProf**: Lightweight sampling-based Profiler operating at process, thread, function and loops level
- Static analysis
  - **CQA** (Code Quality Analyzer): Evaluates the quality of the binary code and offers hints for improving it
- Performance view aggregation module: **ONE View**
  - Goal: Guiding the user through the analysis & optimization process.
  - Synthesizes information provided by different MAQAO modules
  - Automates execution of experiments invoking other MAQAO modules and aggregates their results to produce high-level reports in HTML or XLSX format

# MAQAO LProf: Lightweight Profiler



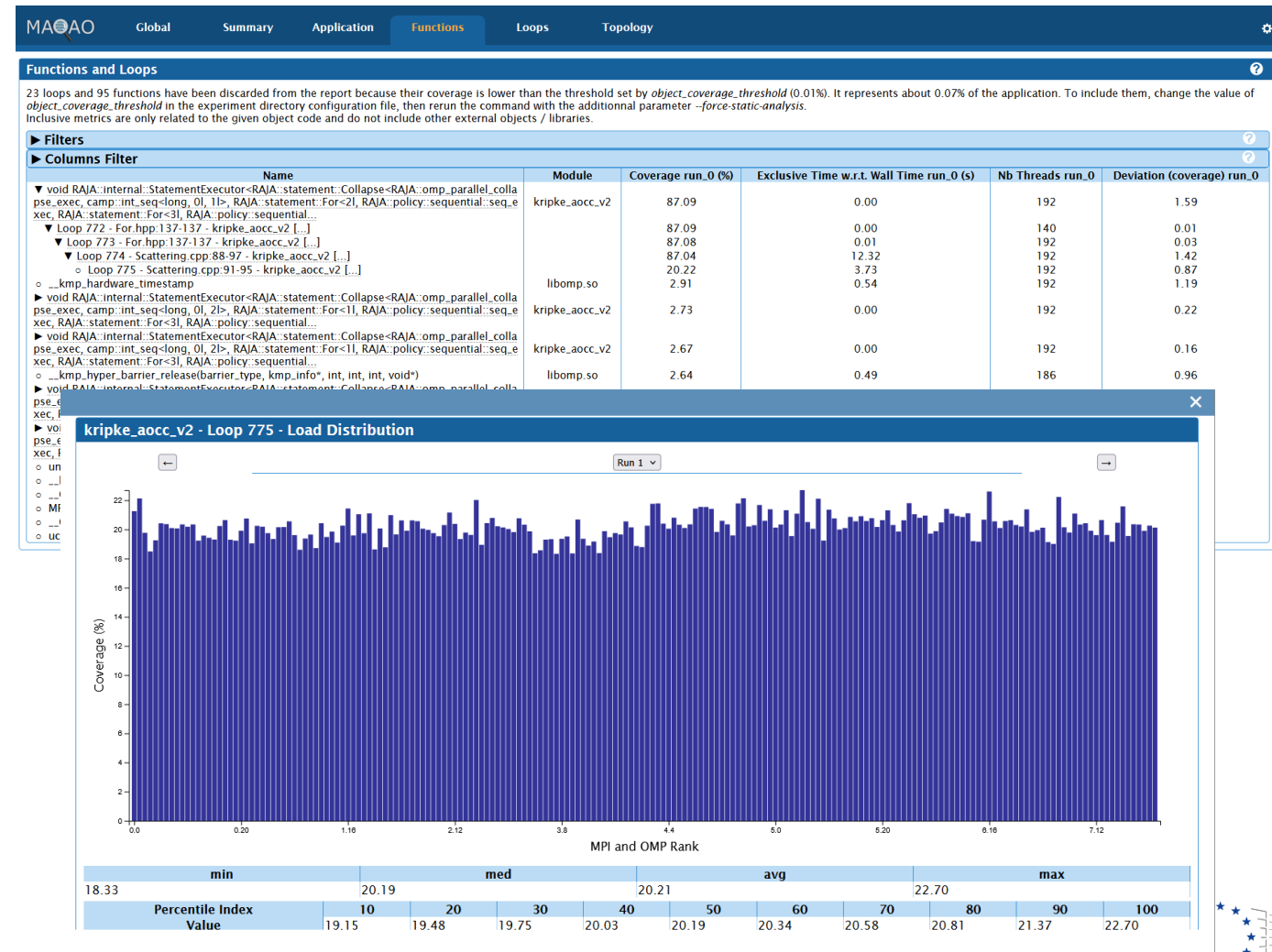
**Goal:** Lightweight localization of application hotspots

## Features:

- Lightweight
- Sampling based
- Access to hardware counters
- Analysis at function and loop granularity

## Strengths:

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



# MAQAO CQA: Code Quality Analyzer



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

## Features:

- Static analysis: no execution of the application
- Allows cross-analysis of/on multiple architectures
- Evaluates the quality of compiler generated code
- Proposes hints and workarounds to improve quality/performance
- Loops centric
  - In HPC, loops cover most of the processing time
- Targets compute-bound codes

### 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 40.00 to 5.00 cycles (8.00x speedup).

### Details

All 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.

### Workaround

- Try another compiler or update/tune your current one:
  - recompile with `fassociative-math` (included in `Ofast` or `ffast-math`) to extend loop vectorization to FP reductions.
- 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: 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)

Logical mapping

```
j=1, 2, 3
i=1  a b c
i=2  d e f
i=3  g h i
```

Physical mapping

(C/C++ storage order: row-major)

```
a b c d e f g h i
```

slow: `for(j=0...) for(i=0...) f(a[i][j]);` OR `for(i=0...) for(j=0...) f(a[j][i]);`

```
fast: for(i=0...) for(j=0...) f(a[i][j]); OR for(j=0...) for(i=0...) f(a[j][i]);

Efficient vectorization + prefetching



```
a b c d e f g h i
```


```

- 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)

# CQA Performance Predictions: “What If” Scenarios

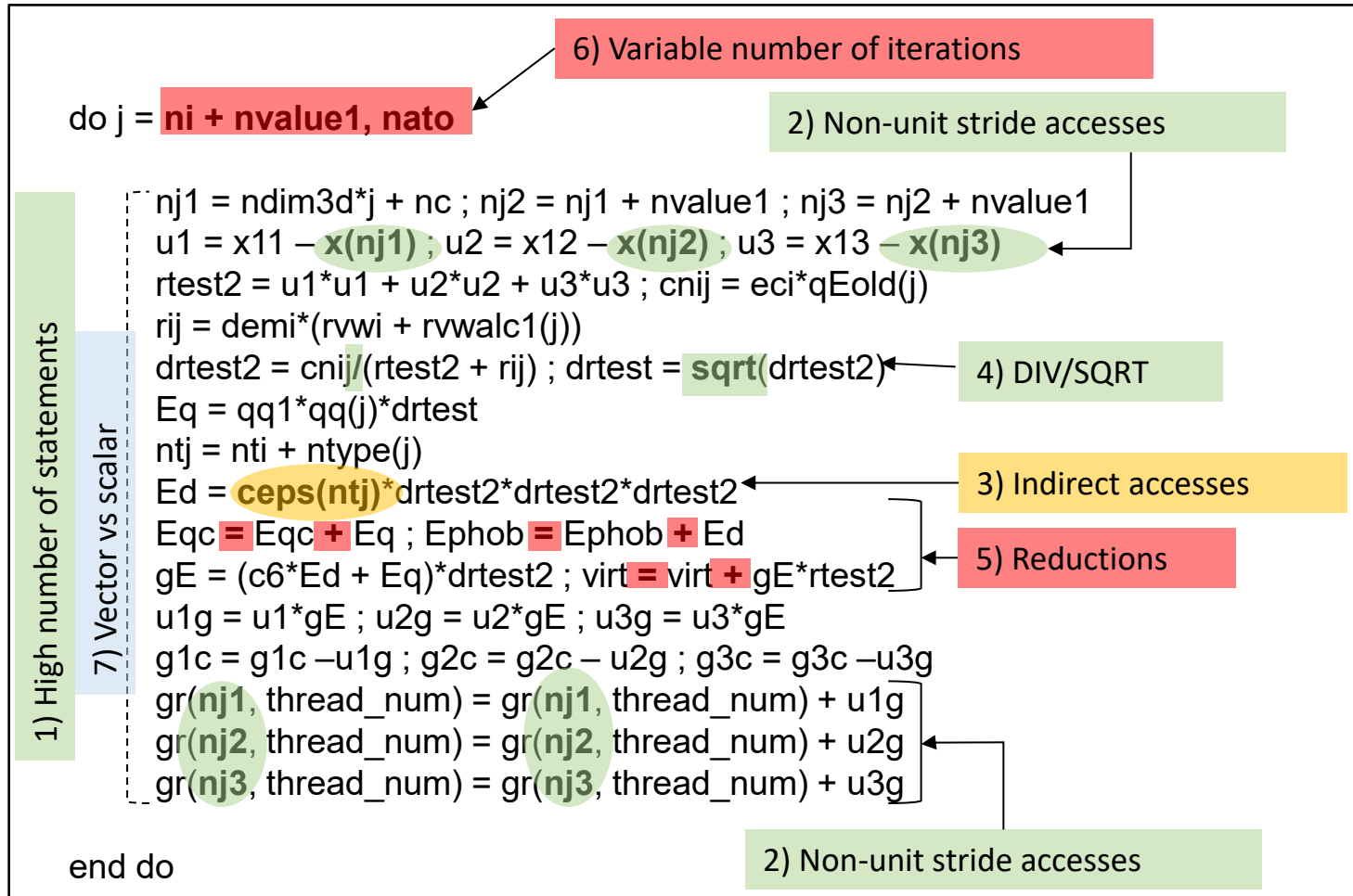


- **Objective: Provide optimistic speedups if a given optimization was applied to a loop**
  - For each optimization, CQA will generate the corresponding ideal assembly code and compute its speedup compared to the original
  - These “What If Scenarios” are generated in a fully static manner.
- **No Scalar Integer:** keep only FP Arithmetic and Memory operations, suppress all others
  - Scenario: Perfect data access (no address computations)
- **FP Vectorised:** only replace scalar FP Arithmetic by Vector FP Arithmetic equivalent. Generate additional instructions to fill in Vector Registers.
  - Scenario: All FP operations successfully vectorised
- **Fully Vectorised:** replace both scalar FP Arithmetic and FP Load/Store by their Vector equivalent.
  - Scenario: All operations successfully vectorised

# Application to Motivating Example



## Issues identified by CQA



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



# ONE View Reports Levels



- ONE View ONE
  - Requires a single run of the application
  - Profiling of the application using LProf
  - Static analysis using CQA
- Scalability mode
  - Multiple executions with varying parallel configurations
  - Allows to evaluate scalability or parallel behaviour of applications
- Comparison mode
  - Comparison of multiple runs (iso-binary or iso-source)
  - Allows to compare performance across different datasets, compilers, or hardware platforms
- Stability mode
  - Multiple runs with identical parameters
  - Allows to assess the stability of execution time

# Analysing an application with MAQAO



## ONE View execution

- Provide all parameters necessary for executing the application
  - Parameters can be passed on the command line or as a configuration file
  - Parameters include executable name, MPI commands, dataset directory, ...

```
$ maqao oneview --create-report=one --mpi_command="mpirun -n 16" -- bt-mz.C.x
```

OR

```
$ maqao oneview --create-report=one --config=my_config.json"
```

- ONE View can reuse an existing experiment directory to perform further analyses
- Results available in HTML format by default
  - XLS spreadsheets and textual output generation are also available

Command line help is available:

```
$ maqao oneview --help
```



# Questions ?



# Navigating ONE View Reports

# ONE View main tabs



- ONE View reports are organised among tabs, each regrouping different sets of information

Global metrics characterizing the application and estimating the impact of standard optimisations

High-level summary evaluating the quality of the experiment and complexity for solving the main optimisation issues

Detailed metrics on the application behaviour

Loops profile and analyses

Reports on application topology and affinity

Functions profile

MAQAO Global Summary Application Functions Loops Topology

kripke\_aocc\_v2 - 2025-03-27 14:40:57 - MAQAO 2.21.4

Help is available by moving the cursor above any ? symbol or by checking [MAQAO website](#).

Style selection

► Filter Information

Global Metrics	
Total Time (s)	19.87
Max (Thread Active Time) (s)	18.44
Average Active Time (s)	18.40
Activity Ratio (%)	100
Average number of active threads	101.560

CQA Potential Speedups Summary

# Global tab



Global metrics on the application general behaviour

Maximal speed-ups at application level for various ideal optimisations

Experimental condition: system characteristics, parallelisation, application compilation flags, ...

kripke\_aocc\_v2 - 2025-03-27 14:40:57 - MAQAO 2.21.4

Help is available by moving the cursor above any [?](#) symbol or by checking [MAQAO website](#).

**Filter Information**

**Global Metrics**

Total Time (s)	19.87
Max (Thread Active Time) (s)	18.44
Average Active Time (s)	18.40
Activity Ratio (%)	100
Average number of active threads	191.569
Affinity Stability (%)	100
GFLOPS	635.617
Time in analyzed loops (%)	94.1
Time in analyzed innermost loops (%)	27.1
Time in user code (%)	94.1
Compilation Options Score (%)	66.7
Array Access Efficiency (%)	59.7

**Potential Speedups**

Perfect Flow Complexity	1.00
Perfect OpenMP + MPI + Pthread	1.02
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.06
No Scalar Integer	Potential Speedup 1.44
FP Vectorised	Potential Speedup 2.01
Fully Vectorised	Potential Speedup 6.18
FP Arithmetic Only	Potential Speedup 1.76
OpenMP perfectly balanced	Potential Speedup 1.02

**CQA Potential Speedups Summary**

**Experiment Summary**

Experiment Name	./kripke_aocc_v2	
Application	./kripke_aocc_v2	
Timestamp	2025-03-27 14:40:57	Universal Timestamp 1743082857
Number of processes observed	8	Number of threads observed 192
Experiment Type	MPI; OpenMP;	
Machine	gmz12.benchmarkcenter.megware.com	
Model Name	AMD EPYC.9655 96-Core Processor	
Architecture	x86_64	Micro Architecture ZEN_V5
Cache Size	1024 KB	Number of Cores 96
OS Version	Linux 5.14.0-503.31.1.el9_5.x86_64 #1 SMP PREEMPT_DYNAMIC Thu Mar 13 06:50:51 EDT 2025	
Architecture used during static analysis	x86_64	Micro Architecture used during static analysis ZEN_V5
Frequency Driver	acpi-cpufreq	Frequency Governor ondemand
Huge Pages	always	Hypertexting on
Number of sockets	2	Number of cores per socket 96
Compilation Options	kripke_aocc_v2: AMD clang version 17.0.6 (CLANG: AOCC_5.0.0-Build#1377 2024_09_24) /home/eoseret/aocc-compiler-5.0.0/bin/clang-17 --driver-mode=g++ -I /beegfs/hackathon/users/eoseret/Kripke/src -I /beegfs/hackathon/users/eoseret/Kripke/build/include -I /beegfs/hackathon/users/eoseret/Kripke/tpl/raja/include -I /beegfs/hackathon/users/eoseret/Kripke/build/tpl/raja/include -I /beegfs/hackathon/users/eoseret/Kripke/tpl/camp/include -I /beegfs/hackathon/users/eoseret/Kripke/build/tpl/raja/tpl/camp/include -I /beegfs/hackathon/users/eoseret/Kripke/build/tpl/raja/tpl/camp/include -isystem /cluster/intel/oneapi/2024.0.0/mpl/2021.11/include -g -grecord-com mand-line -fno-omit-frame-pointer -O3 -D NDEBUG -std=c++14 -fPIC -fopenmp-libomp -MD -MT CMakeFiles/kripke.dir/src/Kripke/Kernel/SweepSubdomain.cpp.o -MF CMakeFiles/kripke.dir/src/Kripke/Kernel/SweepSubdomain.cpp.o.d -o CMakeFiles/kripke.dir/src/Kripke/Kernel/SweepSubdomain.cpp.o -c /beegfs/hackathon/users/eoseret/Kripke/src/Kripke/Kernel/SweepSubdomain.cpp	

**Configuration Summary**

Dataset	
Run Command	<executable> --groups 1024 --zones 24,16,16 --procs 2,2,2
MPI Command	mpirun -n 8
Number Processes	1
Number Nodes	1
Number Processes per Nodes	1
Filter	Not Used
Profile Start	Not Used
Maximal Path Number	4

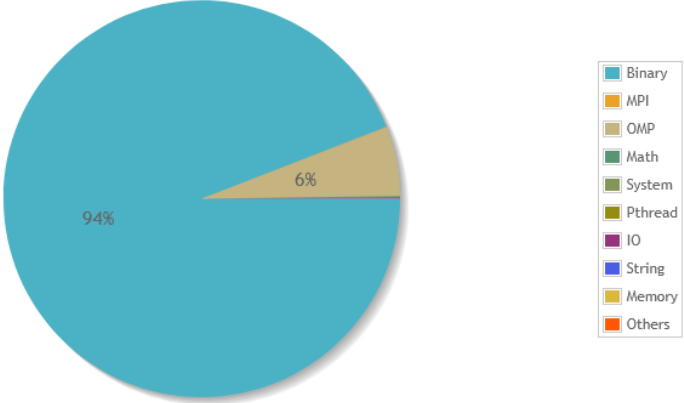
Graphs associated to some of the metrics or speed-ups from the two leftmost panels.

Recap of ONE View configuration

# Navigating Global Tab



Hovering over a question mark in a panel header displays a tooltip describing the content of the panel

Global Metrics		Application Categorization	
Total Time (s)	19.87		
Max (Thread Active Time) (s)	18.44		
Average Active Time (s)	18.40		
Activity Ratio (%)	100		
Average number of active threads	191 569		
Percentage of the application time spent in user code (meaning the time spent in the binary and external libraries specified in custom_categories).			
Time in analyzed innermost loops (%)	27.1		
Time in user code (%)	94.1		
Compilation Options Score (%)	66.7		
Array Access Efficiency (%)	59.7		
Potential Speedups			
Perfect Flow Complexity	1.00		
Perfect OpenMP + MPI + Pthread	1.02		
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.06		
No Scalar Integer	Potential Speedup	1.44	
	Nb Loops to get 80%	1	
FP Vectorised	Potential Speedup	2.01	
	Nb Loops to get 80%	2	
Fully Vectorised	Potential Speedup	6.18	
	Nb Loops to get 80%	4	
FP Arithmetic Only	Potential Speedup	1.76	
	Nb Loops to get 80%	1	
OpenMP perfectly balanced	Potential Speedup	1.02	
	Nb Loops to get 80%	3	

Hovering over a metric name displays a tooltip describing the metric

When hovering over a metric, a small blue bar appearing on the left means that clicking on the metric will cause a graph to appear on the rightmost panel

# Global Tab Metrics

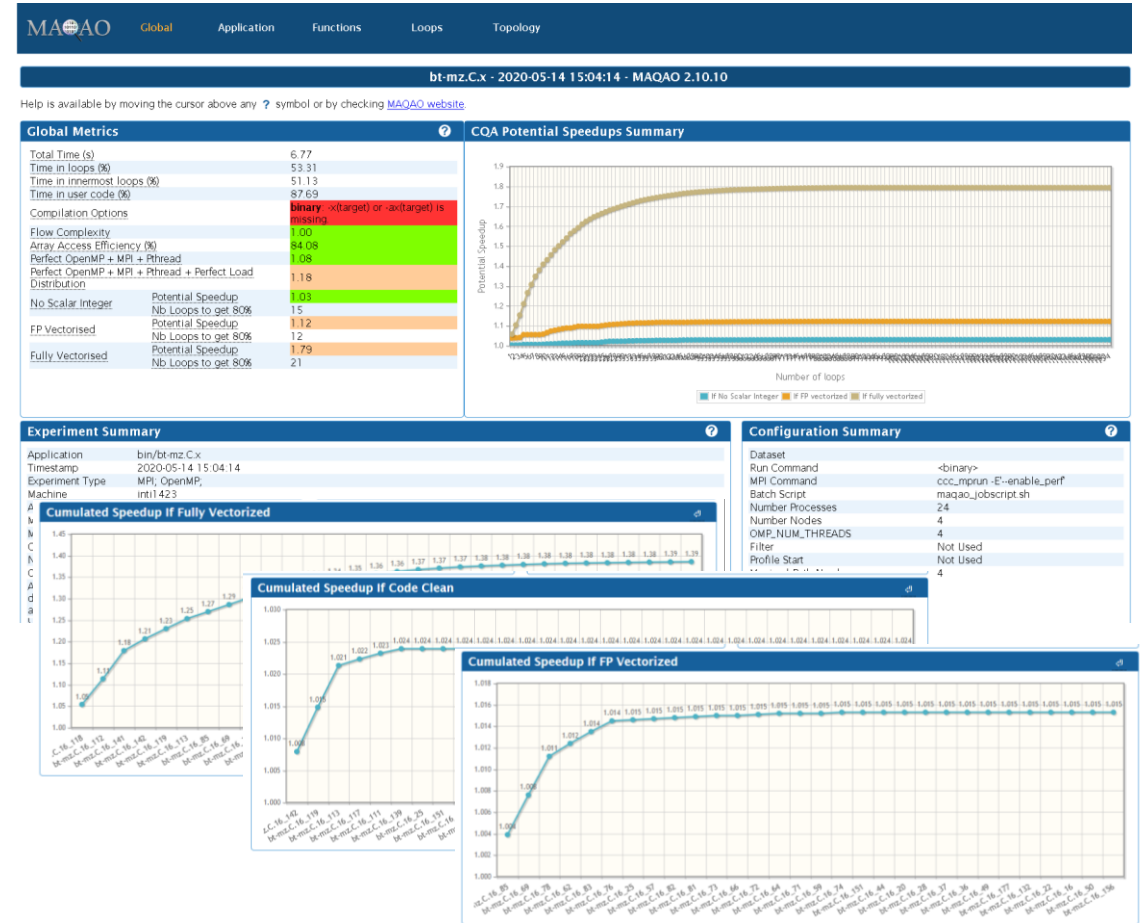


## Experiment summary

- Machine characteristics and configuration

## Global metrics

- General quality metrics derived from MAQAO analyses
- Global speedup predictions
  - Speedup prediction depending on the number of vectorised loops
  - Ordered speedups to identify the loops to optimise first



# Time Categorisation

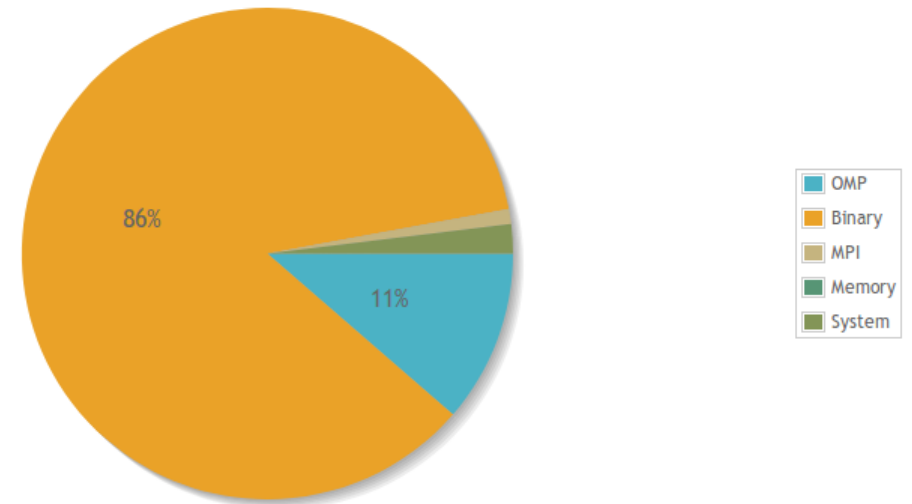


Available from the Global Tab and Application Tab

Objective: 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

Application Categorization



# Summary Tab



Displays an evaluation of the quality of the analysis performed by ONE View. Red items signal issues that could require a rerun of the analysis with the suggested new parameters

Displays an evaluation of the overall optimisation of the code and the expected pitfalls that will be encountered when trying to optimise it.

- The first two tabs will be open by default only if they contain at least one red item

Displays for the hottest loops a list of performance issues and the estimated complexity of their resolutions

The screenshot shows the Summary Tab interface with three main sections: Styler, Strategizer, and Optimizer. Red boxes highlight specific items in each section that correspond to the explanatory text on the left.

**Styler**

- [ 4 / 4 ] Application profile is long enough (18.44 s). To have good quality measurements, it is advised that the application profiling time is greater than 10 seconds.
- [ 3 / 3 ] Optimization level option is correctly used
- [ 3.00 / 3 ] Most of time spent in analyzed modules comes from functions compiled with -g and -fno-omit-frame-pointer. -g option gives access to debugging informations, such as source locations. -fno-omit-frame-pointer improve the accuracy of callchains found during the application profiling.
- [ 3 / 3 ] Host configuration allows retrieval of all necessary metrics.
- [ 0 / 3 ] Compilation of some functions is not optimized for the target processor. Architecture specific options are needed to produce efficient code for a specific processor (-x(target) or -ax(target)).
- [ 2 / 2 ] Application is correctly profiled ("Others" category represents 0.02 % of the execution time). To have a representative profiling, it is advised that the category "Others" represents less than 20% of the execution time in order to analyze as much as possible of the user code
- [ 1 / 1 ] Lstopo present. The Topology Istopo report will be generated.
- [ 0 / 0 ] Fastmath not used. Consider to add fast-math to compilation flags (or replace -O3 with -Ofast) to unlock potential extra speedup by relaxing floating-point computation consistency. Warning: floating-point accuracy may be reduced and the compliance to IEEE/ISO rules/specifications for math functions will be relaxed, typically 'errno' will no longer be set after calling some math functions.

**Strategizer**

- [ 4 / 4 ] Enough time of the experiment time spent in analyzed loops (94.10%). If the time spent in analyzed loops is less than 30%, standard loop optimizations will have a limited impact on application performances.
- [ 4 / 4 ] CPU activity is good. CPU cores are active 100.00% of time
- [ 4 / 4 ] Threads activity is good. On average, more than 99.78% of observed threads are actually active
- [ 4 / 4 ] Affinity is good (100.00%). Threads are not migrating to CPU cores: probably successfully pinned
- [ 4 / 4 ] Loop profile is not flat. At least one loop coverage is greater than 4% (66.79%), representing an hotspot for the application
- [ 4 / 4 ] Enough time of the experiment time spent in analyzed innermost loops (27.12%). If the time spent in analyzed innermost loops is less than 15%, standard innermost loop optimizations such as vectorisation will have a limited impact on application performances.
- [ 0 / 3 ] Cumulative Outermost/In between loops coverage (66.98%) greater than cumulative innermost loop coverage (27.12%). Having cumulative Outermost/In between loops coverage greater than cumulative innermost loop coverage will make loop optimization more complex
- [ 3 / 3 ] Less than 10% (0.00%) is spend in BLAS1 operations. It could be more efficient to inline by hand BLAS1 operations
- [ 2 / 2 ] Less than 10% (0.00%) is spend in BLAS2 operations. BLAS2 calls usually could make a poor cache usage and could benefit from inlining.
- [ 2 / 2 ] Less than 10% (0.00%) is spend in Libm/SVML (special functions)

**Optimizer**

Loop ID	Analysis	Penalty Score
▶ Loop 774 - kripke_aocc_v2	Execution Time: 66 % - Vectorization Ratio: 16.25 % - Vector Length Use: 14.14 %	
▶ Loop 775 - kripke_aocc_v2	Execution Time: 20 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 983 - kripke_aocc_v2	Execution Time: 2 % - Vectorization Ratio: 100.00 % - Vector Length Use: 25.00 %	
▶ Loop 659 - kripke_aocc_v2	Execution Time: 2 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 1309 - kripke_aocc_v2	Execution Time: 1 % - Vectorization Ratio: 14.71 % - Vector Length Use: 14.34 %	
▶ Loop 1088 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 981 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 773 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 658 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 11.89 %	

# Summary Tab: Loop optimisation



Description of a performance issue with associated hint to fix

Estimation of the complexity of fixing this issue (lower is easier)

Loop ID	Analysis	Penalty Score
▼ Loop 774 - kripke_aocc_v2	Execution Time: 66 % - <b>Vectorization Ratio: 16.25 %</b> - <b>Vector Length Use: 14.14 %</b>	6
▼ Loop Computation Issues		4
○	[SA] Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA. Reorganize arithmetic expressions to exhibit potential for FMA. This issue costs 4 points.	4
○	[SA] Presence of a large number of scalar integer instructions - Simplify loop structure, perform loop splitting or perform unroll and jam. This issue costs 2 points.	2
▶ Control Flow Issues		4
▶ Data Access Issues		8
▶ Vectorization Roadblocks		10
▶ Loop 775 - kripke_aocc_v2	Execution Time: 20 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 12.50 %</b>	
▶ Loop 983 - kripke_aocc_v2	Execution Time: 2 % - <b>Vectorization Ratio: 100.00 %</b> - <b>Vector Length Use: 25.00 %</b>	
▶ Loop 659 - kripke_aocc_v2	Execution Time: 2 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 12.50 %</b>	
▶ Loop 1309 - kripke_aocc_v2	Execution Time: 1 % - <b>Vectorization Ratio: 14.71 %</b> - <b>Vector Length Use: 14.34 %</b>	
▶ Loop 1088 - kripke_aocc_v2	Execution Time: 0 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 12.50 %</b>	
▶ Loop 981 - kripke_aocc_v2	Execution Time: 0 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 12.50 %</b>	
▶ Loop 773 - kripke_aocc_v2	Execution Time: 0 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 12.50 %</b>	
▶ Loop 658 - kripke_aocc_v2	Execution Time: 0 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 11.89 %</b>	
▶ Loop 1308 - kripke_aocc_v2	Execution Time: 0 % - <b>Vectorization Ratio: 0.00 %</b> - <b>Vector Length Use: 12.28 %</b>	

Click the triangle to expand and show all performance issue for a loop

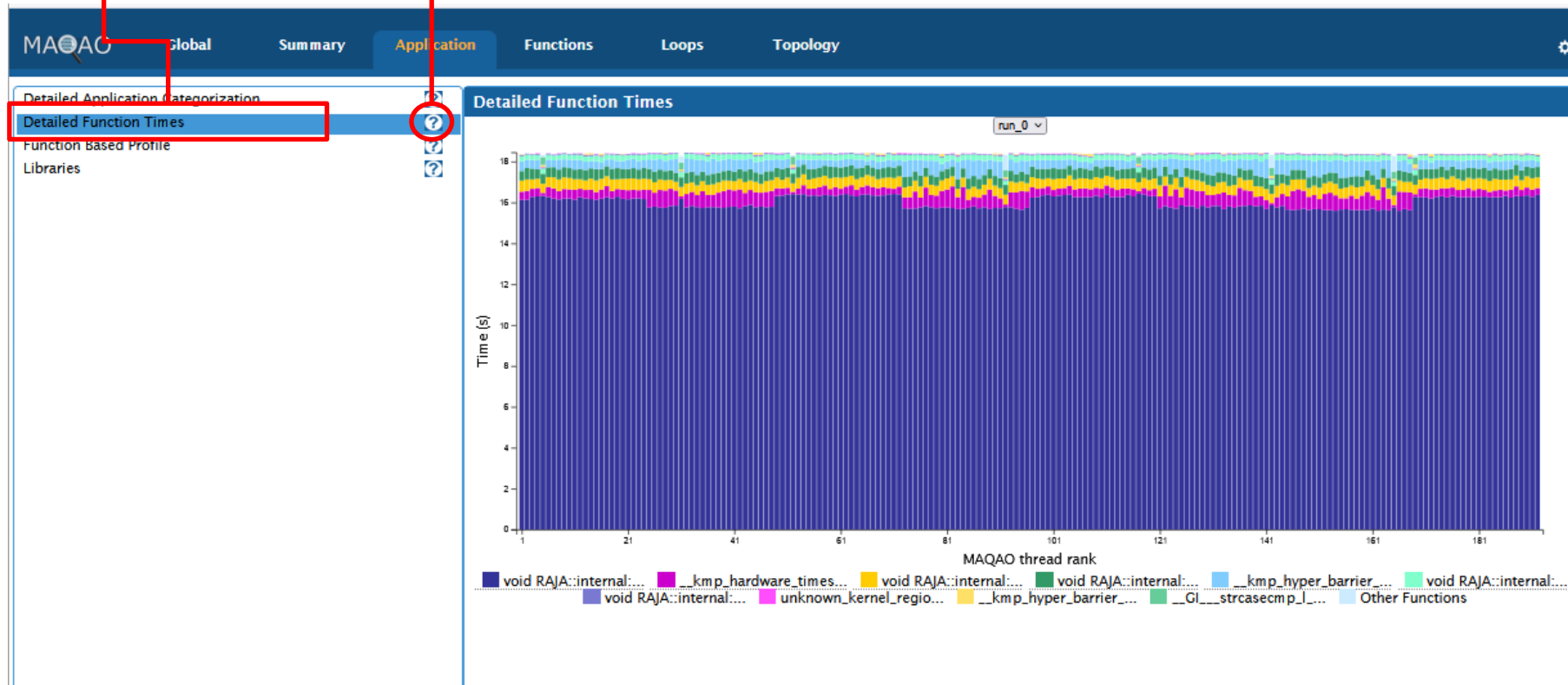
A "+" sign appears when hovering over a line, which allows to expand all at once

# Application Tab



Click on an option to display the corresponding graph on the rightmost panel

Hovering over a question mark near an option displays a tooltip describing the associated information





# Functions Tab: Functions and Loops Profiling



## Identifying hotspots

- Exclusive coverage
- Load balancing across threads
- Loops nests by functions

### ▼ matmul\_sub

- Loop 230 - solve\_subs.f:71-175 - bt-mz.C.16
- Loop 231 - solve\_subs.f:71-175 - bt-mz.C.16

Single

### ▼ z\_solve

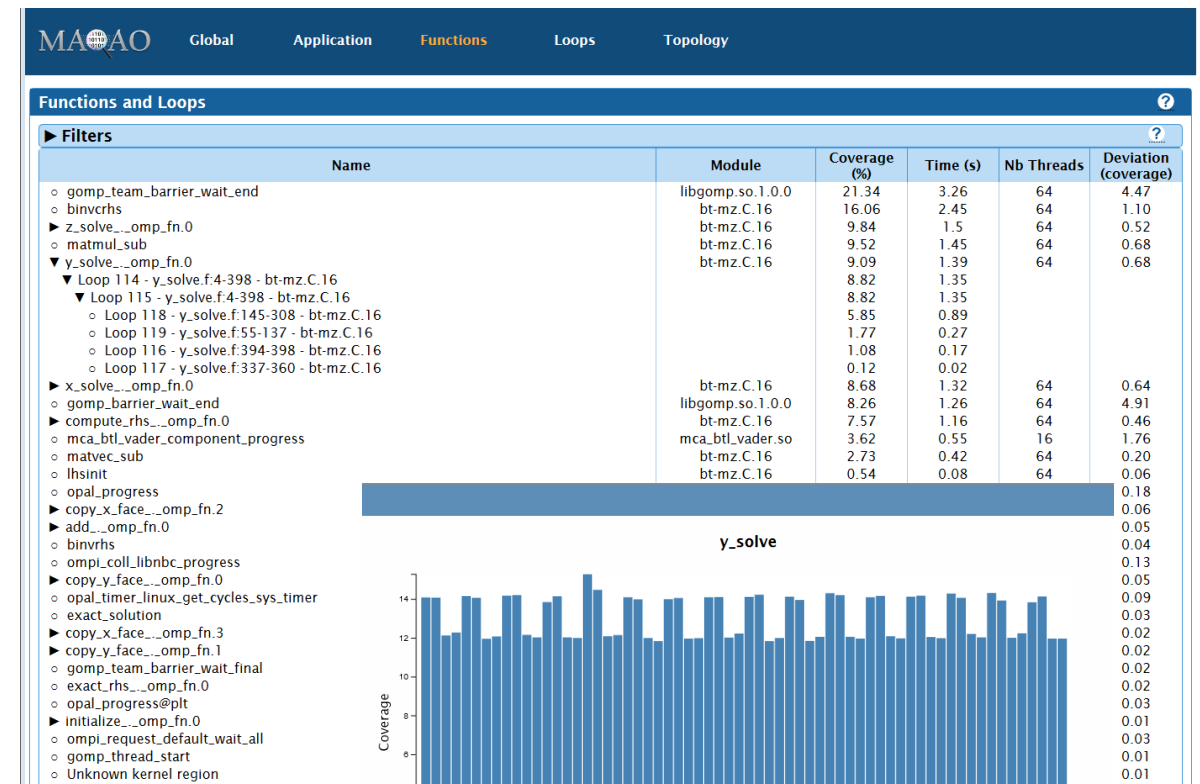
- ▼ Loop 232 - z\_solve.f:53-423 - bt-mz.C.16
- ▼ Loop 233 - z\_solve.f:54-423 - bt-mz.C.16
- ▼ Loop 236 - z\_solve.f:54-423 - bt-mz.C.16
- Loop 239 - z\_solve.f:146-308 - bt-mz.C.16
- Loop 235 - z\_solve.f:55-137 - bt-mz.C.16
- Loop 234 - z\_solve.f:415-423 - bt-mz.C.16

Outermost

Inbetween

Inbetween

Innermost



# Loops Tab



Click this button to restrict the display to innermost/single loops

Expand this tab to view options for restricting the displayed loops by the library containing them

Check the boxes to select which columns to display. The buttons allow to select/deselect groups of similar boxes at once

Hover over a loop Source Location (resp. Source Function) to display in a tooltip the full list of its sources (resp. full name of the function containing it)

Show Innermost Profile Open Expert Summary

### Loops Index

23 loops have been discarded from the report because their coverage is lower than the threshold set by *object\_coverage\_threshold* (0.01%). It represents about 0.03% of the application. To include them, change the value of *object\_coverage\_threshold* in the experiment directory configuration file, then rerun the command with the additional parameter *--force-static-analysis*

**Filters**

**Columns Filter**

Level  
  Exclusive Coverage run\_0 (%)  
  Inclusive Coverage run\_0 (%)  
  Max Exclusive Time Over Threads run\_0 (s)  
  Max Inclusive Time Over Threads run\_0 (s)  
  Exclusive Time w.r.t. Wall Time run\_0 (s)  
  Inclusive Time w.r.t. Wall Time run\_0 (s)  
  Nb Threads run\_0  
  GFLOPS run\_0  
  Vectorization Ratio (%)  
  Vector Length Use (%)  
  Speedup If No Scalar Integer  
  Speedup If FP Vectorized  
  Speedup If Fully Vectorized  
  Speedup If Perfect Load Balancing run\_0  
  Stride 0  
  Stride 1  
  Stride n  
  Stride Unknown  
  Stride Indirect  
  Array Access Efficiency  
   
  
  
  

Loop id	Source Location	Source Function	Level	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
774	kripke_aocc_v2 - Scattering.cpp:88-97 [...]	void RAJA::internal::StatementExecutor<RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec, cam...	InBetween	66.79	16.25	14.14
775	kripke_aocc_v2 - Scattering.cpp:91-95 [...]	void RAJA::internal::StatementExecutor<RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec, cam...	Innermost	20.21	0	12.5
983	kripke_aocc_v2 - For.h pp:137-137 [...]	void RAJA::internal::StatementExecutor<RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec, cam...	Innermost	2.66	100	25
659	kripke_aocc_v2 - For.h pp:137-137 [...]	void RAJA::internal::StatementExecutor<RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec, cam...	Innermost	2.62	0	12.5
1309	kripke_aocc_v2 - For.h pp:137-137 [...]	void RAJA::internal::StatementExecutor<RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec, cam...	Innermost	1.37	14.71	14.34
1088	kripke_aocc_v2 - For.h pp:137-137 [...]	void RAJA::internal::StatementExecutor<RAJA::statement::Collapse<RAJA::omp_parallel_collapse_exec, cam...	Innermost	0.26	0	12.5

# Loops Tab: Profiling Summary



## Identifying loop hotspots

- Vectorisation information
- Potential speedup by optimisation
  - No scalar integer: Removing address computations
  - FP Vectorised: Vectorising floating-point computations
  - Fully Vectorised: Vectorising floating-point computations and memory accesses
  - Perfect Load Balancing: Optimal balance across all threads

MAAO Global Application Functions **Loops** Topology

Show Full Profile Open Expert Summary

### Loops Index

73 loops have been discarded from the report because their coverage is lower than the threshold set by *object\_coverage\_threshold* (0.01%). It represents about 0% of the application. To include them, change the value of *object\_coverage\_threshold* in the experiment directory configuration file, then rerun the command with the additional parameter *--force-static-analysis*

Filters:  Coverage (%)  Level  Time (s)  Vectorization Ratio (%)  Speedup If No Scalar Integer  Speedup If FP Vectorized  Speedup If Fully Vectorized  Speedup If Perfect Load Balancing

Loop id	Source Location	Source Function	Coverage (%)	Level	Time (s)	Vectorization Ratio (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing
179	bt-mz_C.8 - x_solve_e.f:146-309	x_solve_omp_fn.0	7.67	Innermost	1.29	5.02	1.04	1	2.06	1.22
207	bt-mz_C.8 - z_solve_f:146-309	z_solve_omp_fn.0	7.67	Innermost	1.29	5.31	1.02	1	2.06	1.15
185	bt-mz_C.8 - y_solve_f:145-308	y_solve_omp_fn.0	7.35	Innermost	1.24	5.17	1.03	1	2.06	1.22
208	bt-mz_C.8 - z_solve_f:55-137	z_solve_omp_fn.0	3.48	Innermost	0.59	7.09	1	1.13	2.26	1.17
180	bt-mz_C.8 - x_solve_e.f:57-139	x_solve_omp_fn.0	3.09	Innermost	0.52	7.04	1	1.11	2.23	1.25
186	bt-mz_C.8 - y_solve_f:55-137	y_solve_omp_fn.0	3.06	Innermost	0.52	7.09	1	1.11	2.23	1.21
156	bt-mz_C.8 - rhs.f:40-50	compute_rhs_omp_fn.0	2.41	Innermost	0.41	0	1	2	2	1.15
133	bt-mz_C.8 - rhs.f:4-349	compute_rhs_omp_fn.0	1.84	Innermost	0.31	0	1	1.65	3.41	1.29
150	bt-mz_C.8 - rhs.f:4-132	compute_rhs_omp_fn.0	1.77	Innermost	0.3	0	1	1.71	3.68	1.27
142	bt-mz_C.8 - rhs.f:4-238	compute_rhs_omp_fn.0	1.76	Innermost	0.3	0	1	1.65	3.41	1.27
204	bt-mz_C.8 - z_solve_f:115-121	z_solve_omp_fn.0	1.7	Innermost	0.29	0	1	1	2.83	1.17



# Navigating a Loop Report



Source code of the loop. If Assembly Code is selected is the other panel, selecting a line will highlight the corresponding assembly lines

Assembly code of the loop. If Source Code is selected in the other panel, selecting a line will highlight the corresponding source lines

Use the scrolling lists on both panels to choose the content of the panel

Call stacks leading to this loop, with associated % of occurrence

Percentage of time spent in this loop across the threads

High-level static analysis report with hints for improving performance

Detailed metrics generated by the static analysis

The screenshot shows the MAQAO interface for Loop 774. The top navigation bar includes 'Global', 'Summary', 'Application', 'Functions', 'Loops', and 'Topology'. The 'Loops' tab is active, showing 'Loop Id: 774', 'Module: kripke\_aocc\_v2', 'Source: Scattering.cpp:88-97 [...]', and 'Coverage: 66.79%'. A dropdown menu is open over the 'Source Code' tab, showing options: 'Source Code', 'Assembly Code', 'Callchains', 'Load Distribution', 'CQA', and 'CQA Advanced'. The main panel displays source code for lines 137-137, 87-221, and 190-190. The right sidebar shows 'CQA' analysis results under 'gain', 'potential', 'hint', and 'expert' tabs. The 'Vectorization' section states: 'Your loop is probably not vectorized. Only 14% of vector register length is used (average across all SSE/AVX instructions, your loop, you can lower the cost of an iteration from 2.75 to 0.27 cycles (10.35x speedup)'. The 'Workaround' section lists: 'Try another compiler or update/tune your current one: recompile with ffast-math (included in Ofast) to extend loop vectorization to FP reductions.' and '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 loops accordingly: 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)'. A 'Logical mapping' diagram shows a 2x3 grid with 'i=1' on the left and 'j=1, 2, 3' on top, with cells containing 'a', 'b', and 'c'.

# Loop Report: Static Analysis



## High level reports

- Reference to the source code
- Bottleneck description
- Hints for improving performance
- Reports categorized by probability that applying hints will yield predicted gain
  - Gain: Good probability
  - Potential gain: Average probability
  - Hints: Lower probability

The screenshot displays the MAAO (Memory Access Analysis) tool interface. The main window shows the source code for a loop (Loop Id: 224) with line numbers 711 to 1044. The code consists of nested loops (cblock(1,1) to cblock(5,1)) and arithmetic operations. The right-hand side of the interface provides a detailed report for the selected loop, including:

- Coverage:** 4.79%
- Function:** `matmul_sub`
- Source file and lines:** `solve_subs.f:71-175`
- Module:** `bt-mz.C.16`
- Description:** "The loop is defined in /ccc/dsku/nfs-server/user/cont001/ocre/valensic/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ/solve\_subs.f:71-175. It is main loop of related source loop which is unrolled by 2 (including vectorization)."
- 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 27.00 to 25.00 cycles (1.08x speedup)."
- Workaround:** "Remove scalar integer instructions (e.g., `do i = 1, n`) and use vectorized operations (e.g., `do i = 1, n, 4`) to extend the stride of the loop. This will allow the compiler to use AVX2 processors. By fully unrolling the loop (e.g., `do i = 1, n, 16`), you can achieve a speedup of 1.08x." (Note: the text in the image is partially obscured and some details are hard to read).
- Workaround:** "Remove scalar integer instructions (e.g., `do i = 1, n`) and use vectorized operations (e.g., `do i = 1, n, 4`) to extend the stride of the loop. This will allow the compiler to use AVX2 processors. By fully unrolling the loop (e.g., `do i = 1, n, 16`), you can achieve a speedup of 1.08x." (Note: the text in the image is partially obscured and some details are hard to read).
- Workaround:** "Remove scalar integer instructions (e.g., `do i = 1, n`) and use vectorized operations (e.g., `do i = 1, n, 4`) to extend the stride of the loop. This will allow the compiler to use AVX2 processors. By fully unrolling the loop (e.g., `do i = 1, n, 16`), you can achieve a speedup of 1.08x." (Note: the text in the image is partially obscured and some details are hard to read).

# Loop Report – Expert View



Low level reports for performance experts

- Assembly-level
- Instructions cycles costs
- Instructions dispatch predictions
- Memory access analysis

Assembly code

- Highlights groups of instructions accessing the same memory addresses

CQA low level metrics

The screenshot displays the 'Expert View' of a loop report. At the top, there are tabs for 'Gain', 'Potential gain', 'Hints', and 'Experts only'. The main section is titled 'ASM code' and shows the address of the loop as 421409. Below this is a table of instructions with columns for 'Instruction', 'Nb FU', 'P0', 'P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'Latency', and 'Recip. throughput'. The instructions listed include MOVAPS, INC, DIVSD, MOVAPS, MULSD, and MOVSD. Below the table, there are sections for 'Loop Id: 224', 'Module: bt-mz.C.16', and 'Assembly Code'. A 'Hide groups analysis' button is visible. The bottom part of the screenshot shows a 'CQA Advanced' window with a table of metrics and their values. The metrics include Coverage (% app. time), Time (s), CQA speedup if clean, CQA speedup if FP arith vectorized, CQA speedup if fully vectorized, CQA speedup if no inter-iteration dependency, CQA speedup if next bottleneck killed, Source, Source loop unroll info, Source loop unroll confidence level, Unroll/vectorization loop type, Unroll factor, CQA cycles, CQA cycles if clean, CQA cycles if FP arith vectorized, CQA cycles if fully vectorized, Front-end cycles, P0 cycles, P1 cycles, P2 cycles, and P3 cycles. The 'CQA cycles if fully vectorized' metric is highlighted in orange with a value of 2.00.

Metric	Value
Coverage (% app. time)	4.79
Time (s)	0.23
CQA speedup if clean	1.08
CQA speedup if FP arith vectorized	1.65
CQA speedup if fully vectorized	2.00
CQA speedup if no inter-iteration dependency	NA
CQA speedup if next bottleneck killed	1.08
Source	solve_subs.f:71-175
Source loop unroll info	unrolled by 2
Source loop unroll confidence level	max
Unroll/vectorization loop type	main
Unroll factor	2
CQA cycles	27.00
CQA cycles if clean	25.00
CQA cycles if FP arith vectorized	16.32
CQA cycles if fully vectorized	13.50
Front-end cycles	22.50
P0 cycles	25.00
P1 cycles	27.00
P2 cycles	13.00
P3 cycles	13.00

# Topology Tab



**Software Topology** ?

Number processes: 1    Number nodes: 1    Number processes per node: 1  
 Run Command: <executable> --groups 1024 --zones 24,16,16 --procs 2,2,2    MPI Command: mpirun -n 8    Run Directory: .  
 OMP\_NUM\_THREADS: 24    OMP\_PROC\_BIND: spread    I\_MPI\_PIN\_DOMAIN: auto    OMP\_PLACES: threads

ID	Observed Processes	Observed Threads	Time(s)	Elapsed Time (s)	Active Time (%)	Start (after process) (s)	End (before process) (s)	Maximum Time on the Same CPU (s)
▼ Node gmz12.benchmarkcenter.megware.com	8	192	18.44					
▶ MPI # 0 (PID 19442)	+	24	18.44					
▼ MPI # 1 (PID 19458)		24	18.41					
○ MPI # 1 - OMP # 1.0 (TID 19458)			18.33	19.83	92.42	0.00	0.00	18.62
○ MPI # 1 - OMP # 1.1 (TID 19512)			18.41	18.67	98.58	1.15	0.01	18.63
○ MPI # 1 - OMP # 1.2 (TID 19515)			18.40	18.67	98.56	1.15	0.01	18.62
○ MPI # 1 - OMP # 1.3 (TID 19523)			18.41	18.68	98.57	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.4 (TID 19536)			18.40	18.68	98.52	1.15	0.00	18.62
○ MPI # 1 - OMP # 1.5 (TID 19547)			18.41	18.68	98.61	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.6 (TID 19565)			18.41	18.67	98.58	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.7 (TID 19580)			18.41	18.67	98.58	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.8 (TID 19595)			18.40	18.67	98.53	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.9 (TID 19679)			18.40	18.66	98.61	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.10 (TID 19695)			18.40	18.66	98.64	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.11 (TID 19709)			18.41	18.66	98.67	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.12 (TID 19725)			18.40	18.66	98.64	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.13 (TID 19741)			18.41	18.66	98.70	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.14 (TID 19757)			18.40	18.66	98.64	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.15 (TID 19769)			18.41	18.66	98.67	1.17	0.01	18.63
○ MPI # 1 - OMP # 1.16 (TID 19776)			18.40	18.66	98.65	1.17	0.01	18.63

Click the triangle to show the processes (resp. threads) depending on a node (resp. process)

Double click on a thread to display the Functions Profile (similar to the Functions tab) restricted to that thread

A "+" sign appears when hovering over a line, which allows to expand the item and all its children

# MAQAO ONE View Thread/Process View



## Software Topology

- List of nodes
- Processes by node
- Thread by process

## View by thread

- Function profile at the

MAQAO Global Application Functions Loops Topology			
Software Topology ?			
ID	Processes	Threads	Time(s)
▼ Node c251-109.wrangler.tacc.utexas.edu	8	32	5.34
▼ Process 145897		4	5.34
○ Thread 145897			5.34
○ Thread 145933			5.32
○ Thread 145952			5.32
○ Thread 145969			5.3
▶ Process 145899		4	5.34
▶ Process 145901		4	5.34
▶ Process 145903		4	5.34
▶ Process 145898		4	5.34
▶ Process 145900		4	5.34
▶ Process 145895		4	5.34
▶ Process 145896		4	5.34
▶ Node c251-110.wrangler.tacc.utexas.edu	8	32	5.36
○ AVERAGE			5.36

MAQAO Global	
--------------	--

Profiling node c251-109.wrangler.tacc.utexas.edu - process 145897 - thread 145897			
Name	Module	Coverage (%)	Time (s)
○ binvcrhs	bt-mz_B.16	24.34	1.3
○ _INTERNAL_25_____src_kmp_barrier_cpp_fa608613::__kmp_hy_per_barrier_gather(barrier_type, kmp_info*, int, int, void (*)(void*, void*), void*)	libiomp5.so	17.6	0.94
▶ matmul_sub	bt-mz_B.16	12.73	0.68
▶ y_solve	bt-mz_B.16	7.87	0.42
▶ compute_rhs	bt-mz_B.16	7.49	0.4
▶ x_solve	bt-mz_B.16	7.12	0.38
▶ z_solve	bt-mz_B.16	6.74	0.36

# Topology Tab: Istopo View



- Hover above the « Topology » tab name to see the scrolling list allowing to select additional views to display

The screenshot shows the 'TOPOLOGY' tab in the Istopo interface. On the left, a 'Loops' menu has 'Topology' selected. Below it, a 'Command:' field shows 'OMP\_PLACES:'. Three red boxes highlight 'Istopo', 'Istopo\_PU', and 'Istopo\_threads' in the menu. Three callout boxes explain these options:

- Istopo:** Displays a graphical representation of the threads affinities and the cores on which they ran
- Istopo\_PU:** Displays for each core a weighted list of threads that executed on this core
- Istopo\_threads:** Displays for each thread a weighted list of the cores on which the thread executed

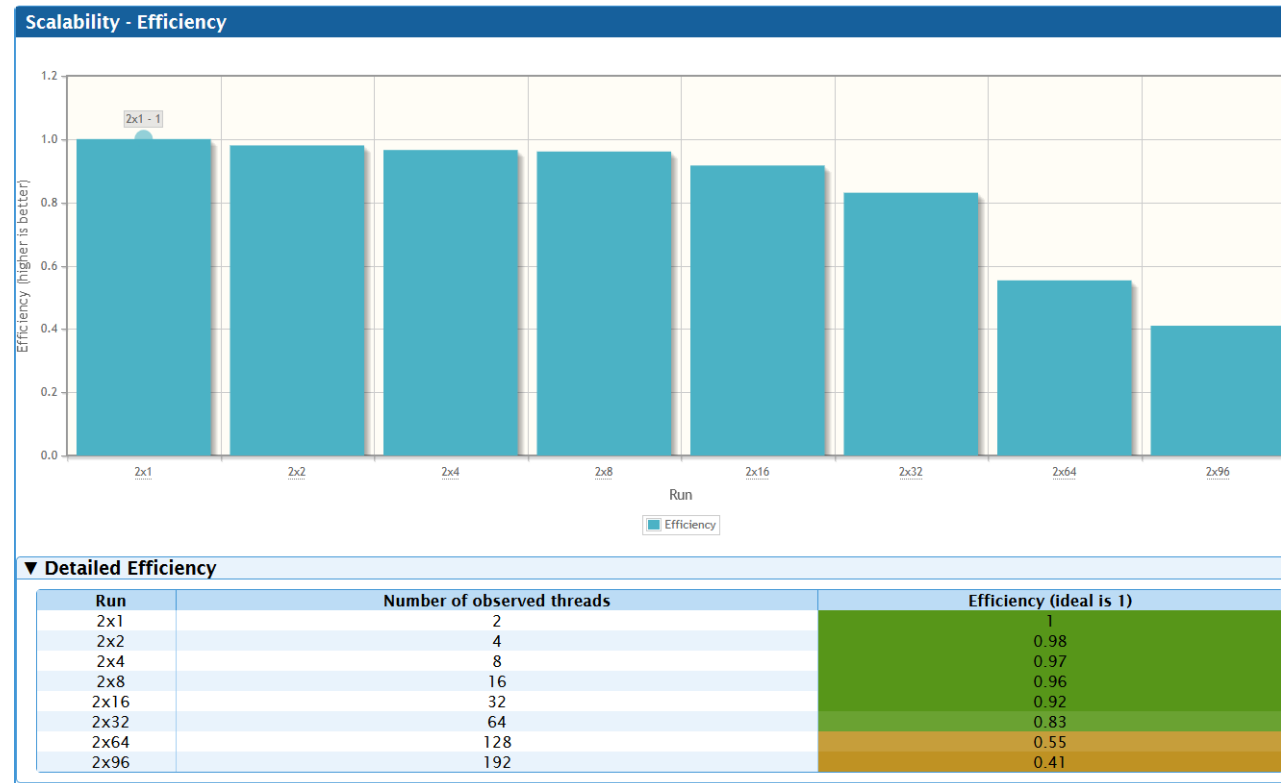
The main view shows a grid of 'Nodes' (Machine: 75568 total) with columns for 'Nodes' and 'PIDs'. A tooltip for 'gmz12.benchmarkcenter.megware.com - P#13: 100.00%' is visible. A list of threads is shown on the right, with 'MPI #0 - OMP #0.13 - Thread 19693' highlighted in red. A callout box at the bottom states: 'Hover over a thread to highlight the core(s) on which it ran'.

# MAQAO ONE View Scalability Reports



Goal: Provide a view of the application scalability

- Profiles with different numbers of threads/processes
- Displays efficiency metrics for application



# MAQAO ONE View Scalability Reports

## Application View

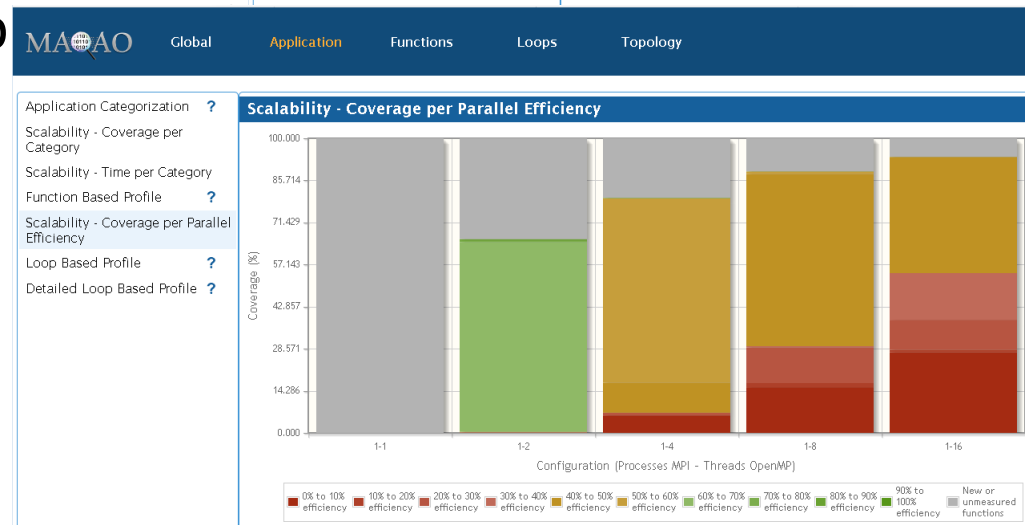
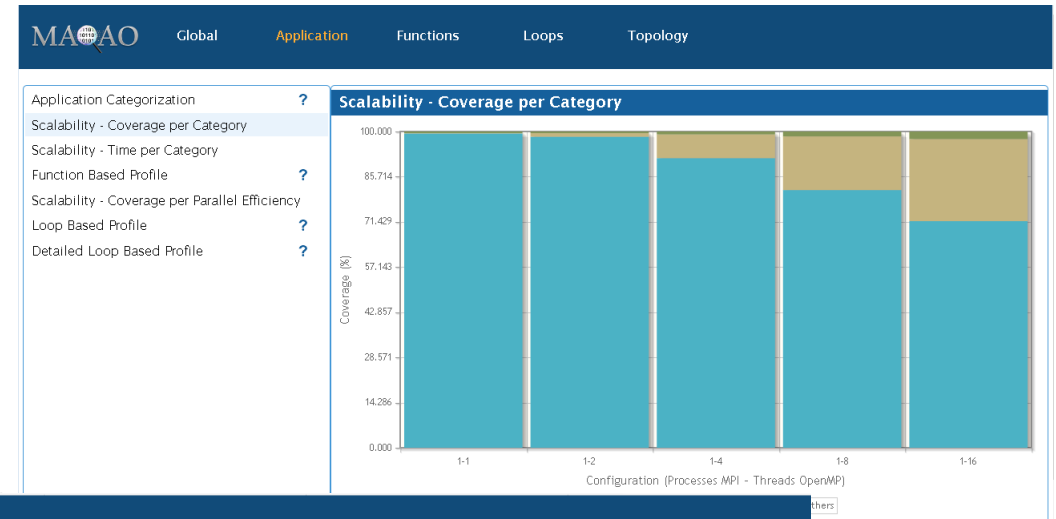


### Coverage per category

- Comparison of categories for each run

### Coverage per parallel efficiency

- $Efficiency = \frac{T_{sequential}}{T_{parallel} * N_{threads}}$ 
  - Distinguishing functions only rep
- Displays efficiency by coverage



# MAQAO ONE View Scalability Reports Functions and Loops Views



Displays metrics for each function/loop

- Efficiency
- Potential speedup if efficiency=1

MAQAO
Global
Application
Functions
Loops
Topology

**Functions and Loops**

**Filters**
 (1-1) Efficiency
  (1-1) Potential Speed-Up (%)
  (1-2) Efficiency
  (1-2) Potential Speed-Up (%)
  (1-4) Efficiency
  (1-4) Potential Speed-Up (%)
  (1-8) Efficiency
  (1-8) Potential Speed-Up (%)
  Select none

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)	(1-1) Efficiency	(1-2) Efficiency	(1-2) Potential Speed-Up (%)	(1-4) Efficiency	(1-4) Potential Speed-Up (%)	(1-8) Efficiency	(1-8) Potential Speed-Up (%)	(1-16) Efficiency	(1-16) Potential Speed-Up (%)
o _INTERNAL_25.....src_kmp_barrier_cpp_ac7c2c73:....kmp_hyper_barrier_release(barrier_type, kmp_info*, int, int, void*)	libiomp5.so	24.02	15.38	16	18.62		1	0	0.04	5.49	0.01	14.35	0.01	23
o binvcrhs	bt-mz.C.1	20.71	13.27	16	6.22	1	0.7	6.14	0.55	10.2	0.45	11.58	0.41	11.43
▶ compute_rhs	bt-mz.C.1	10.76	6.9	16	2.45	1	0.63	2.68	0.42	5.39	0.26	8.47	0.25	7.57

MAQAO
Global
Application
Loops
Topology

**Loops Index**

Coverage (%)
  Time (s)
  Vectorization Ratio (%)
  Speedup If Clean
  Speedup If FP Vectorized
  Speedup If Fully Vectorized
  (1-1) Efficiency
  (1-1) Potential Speed-Up (%)
  (1-2) Efficiency
  (1-2) Potential Speed-Up (%)
  (1-4) Efficiency
  (1-4) Potential Speed-Up (%)
  (1-8) Efficiency
  (1-8) Potential Speed-Up (%)
  (1-16) Efficiency
  (1-16) Potential Speed-Up (%)
  Select none

Loop id	Source Lines	Source File	Source Function	(1-2) Efficiency	(1-2) Potential Speed-Up (%)	(1-4) Efficiency	(1-4) Potential Speed-Up (%)	(1-8) Efficiency	(1-8) Potential Speed-Up (%)	(1-16) Efficiency	(1-16) Potential Speed-Up (%)
Loop 215	71-175	bt-mz.C.1:solve_subsf	matmul_sub	0.71	1.51	0.56	2.49	0.45	2.99	0.41	2.96
Loop 224	146-308	bt-mz.C.1:z_solve.f	z_solve	0.7	1.34	0.57	2.07	0.43	2.73	0.4	2.62
Loop 192	146-308	bt-mz.C.1:x_solve.f	x_solve	0.66	1.22	0.52	1.91	0.45	1.92	0.39	2.04
Loop 199	145-307	bt-mz.C.1:y_solve.f	y_solve	0.69	1.09	0.54	1.81	0.45	1.99	0.39	2.11
Loop 169	40-50	bt-mz.C.1:rhs.f	compute_rhs	0.52	0.49	0.23	1.59	0.11	2.95	0.11	2.3
Loop 221	55-137	bt-mz.C.1:z_solve.f	z_solve	0.66	0.92	0.54	1.32	0.43	1.56	0.37	1.66
Loop 189	57-139	bt-mz.C.1:x_solve.f	x_solve	0.71	0.7	0.57	1.14	0.47	1.28	0.43	1.26
Loop 196	55-137	bt-mz.C.1:y_solve.f	y_solve	0.73	0.52	0.55	1.01	0.44	1.18	0.41	1.12
Loop 165	65-67	bt-mz.C.1:rhs.f	compute_rhs	0.45	0.55	0.24	1.22	0.11	2.31	0.13	1.64
Loop 227	26-28	bt-mz.C.1:add.f	add#omp_loop_0	0.64	0.12	0.44	0.22	0.25	0.4	0.09	1.14
Loop 220	415-423	bt-mz.C.1:z_solve.f	z_solve	0.67	0.34	0.49	0.62	0.34	0.87	0.3	0.88
Loop 188	395-399	bt-mz.C.1:x_solve.f	x_solve	0.62	0.5	0.56	0.57	0.44	0.69	0.41	0.65
Loop 216	71-175	bt-mz.C.1:solve_subsf	matmul_sub	0.77	0.23	0.62	0.41	0.48	0.54	0.4	0.62
Loop 171	304-349	bt-mz.C.1:rhs.f	compute_rhs	0.71	0.29	0.65	0.34	0.46	0.56	0.44	0.5



# Performance Optimisation and Productivity 3

A Centre of Excellence in HPC

## Contact:

 <https://www.pop-coe.eu>

 [pop@bsc.es](mailto:pop@bsc.es)

 [@POP\\_HPC](#)

 [youtube.com/POPHPC](https://www.youtube.com/POPHPC)





# Backup Slides

# Performance analysis and optimisation



**Where** is the application spending most execution time and resources?

**Why** is the application spending time there?

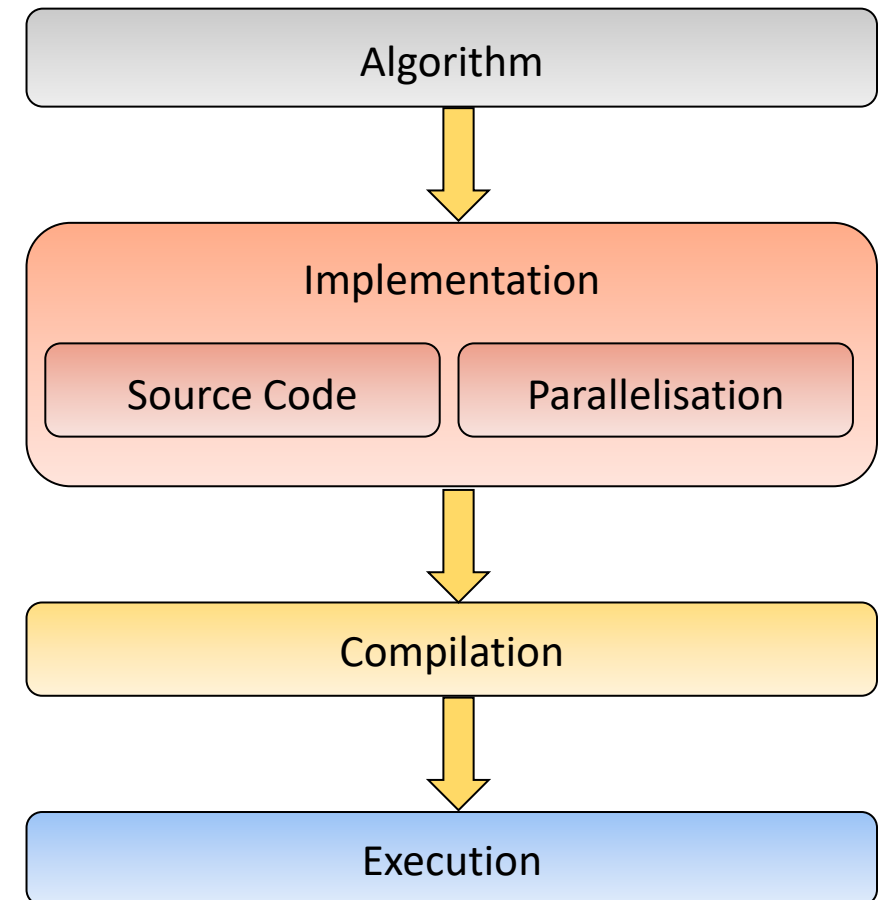
- Algorithm, implementation, runtime or hardware?
- Data access or computation?

**How** to improve the application?

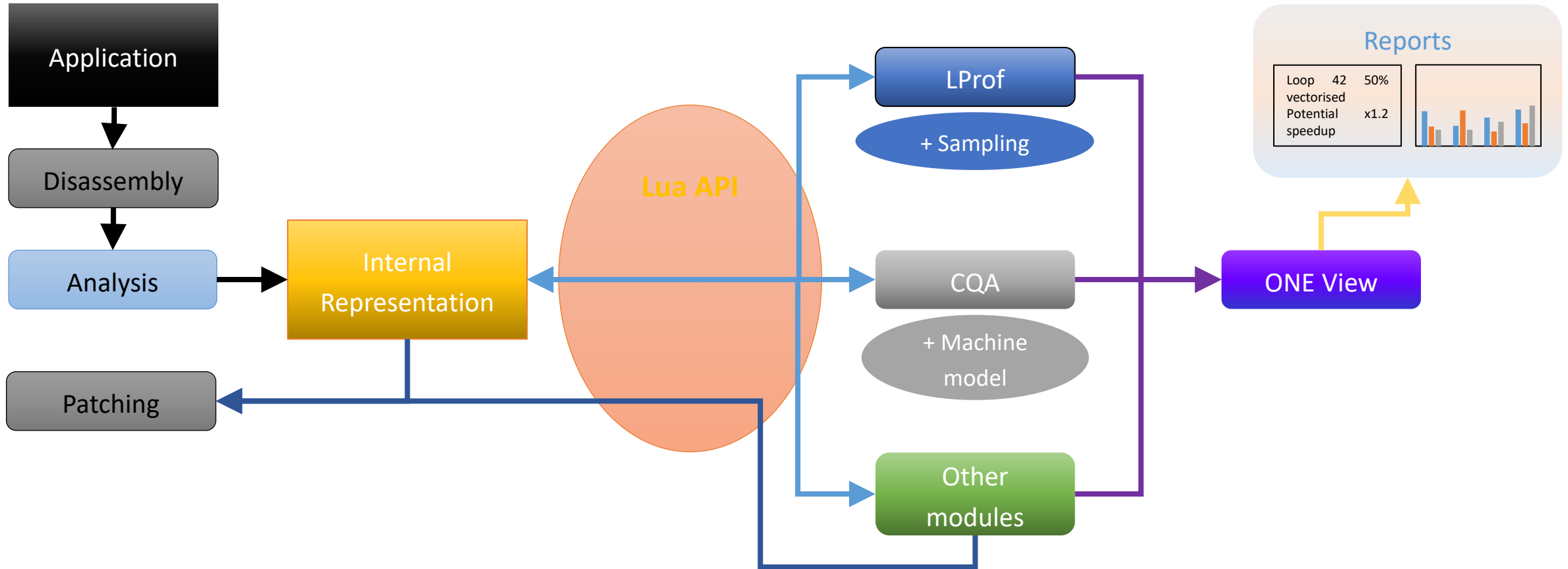
- At which step(s) of the workflow or dev process?
- What additional information is needed?

**How much** gain can be expected?

- At what cost?



# MAQAO Main structure



# Partnerships



- MAQAO is part of the POP Centre of Excellence
- Provides performance optimisation and productivity services for academic and industry
- <https://pop-coe.eu/>



- MAQAO has been funded by UVSQ, Intel and CEA (French department of energy) through Exascale Computing Research (ECR) and through various European projects (FUI/ITEA: H4H, COLOC, PerfCloud, ELCI, POP2 CoE, TREX CoE, etc...)
- Provided core binary analysis and instrumentation capabilities and features for other tools:
- TAU performance tools with MADRAS patcher through MIL (MAQAO Instrumentation Language)
  - X86\_64 only, aarch64 under development
- Intel Advisor

# MAQAO CQA: Main Concepts



Applications exploit at best 5 to 10% of the peak performance.

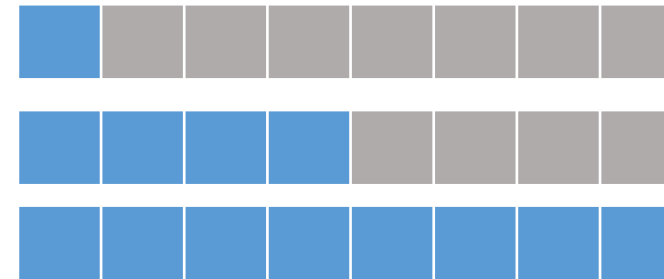
Main elements of analysis:

- Peak performance
- Execution pipeline
- Resources/Functional units

Key performance levers for core level efficiency:

- Vectorization
- Avoiding high latency instructions if possible (DIV/SQRT)
- Guiding the compiler code optimization
- Reorganizing memory and data structures layout

Same instruction – Same cost



Process up to  
8X data



Compilers can be driven using flags, pragmas, and keywords:

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

Hints for implementation changes:

- Improve data access patterns
  - Memory alignment
  - Loop interchange
  - Changing loop strides
  - Reshaping arrays of structures
- Avoid instructions with high latency (SQRT, DIV, GATHER, SCATTER, ...)

# Analysing an application with MAQAO



MAQAO modules can be invoked separately for advanced analyses

- LProf

- Profiling

```
$ maqao lprof xp=exp_dir --mpi-command="mpirun -n 16 -ppn 4" ppn=4 -- ./bt-mz.C.16
```

- Display functions profile

```
$ maqao lprof xp=exp_dir -df
```

- Displaying the results from a ONE View run

```
$ maqao lprof xp=oneview_xp_dir/tools/lprof_npsu -df
```

- CQA

```
$ maqao cqa loop=42 bt-mz.C.16
```

Command line help is available:

```
$ maqao lprof --help
```

```
$ maqao cqa --help
```

# Application to Motivating Example



### Gain Potential gain Hints Experts only

#### 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 continuously 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 in 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

### Gain Potential gain Hints Experts only

#### 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 then ADD).  
However `(a+b)*c` cannot be translated into FMA.

### Gain Potential gain Hints Experts only

#### 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