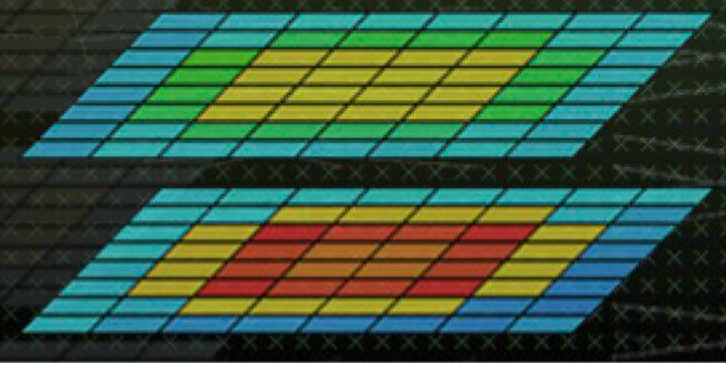


MAQAO
Performance Analysis and Optimization Framework

Cédric VALENSI, Emmanuel OSERET, Hugo BOLLORÉ
{cedric.valensi, emmanuel.oseret, hugo.bollore}@uvsq.fr
Performance Evaluation Team, University of Versailles S-Q-Y

http://www.maqao.org

VI-HPS 45th TW Garching near München, Germany
10–13 June 2024

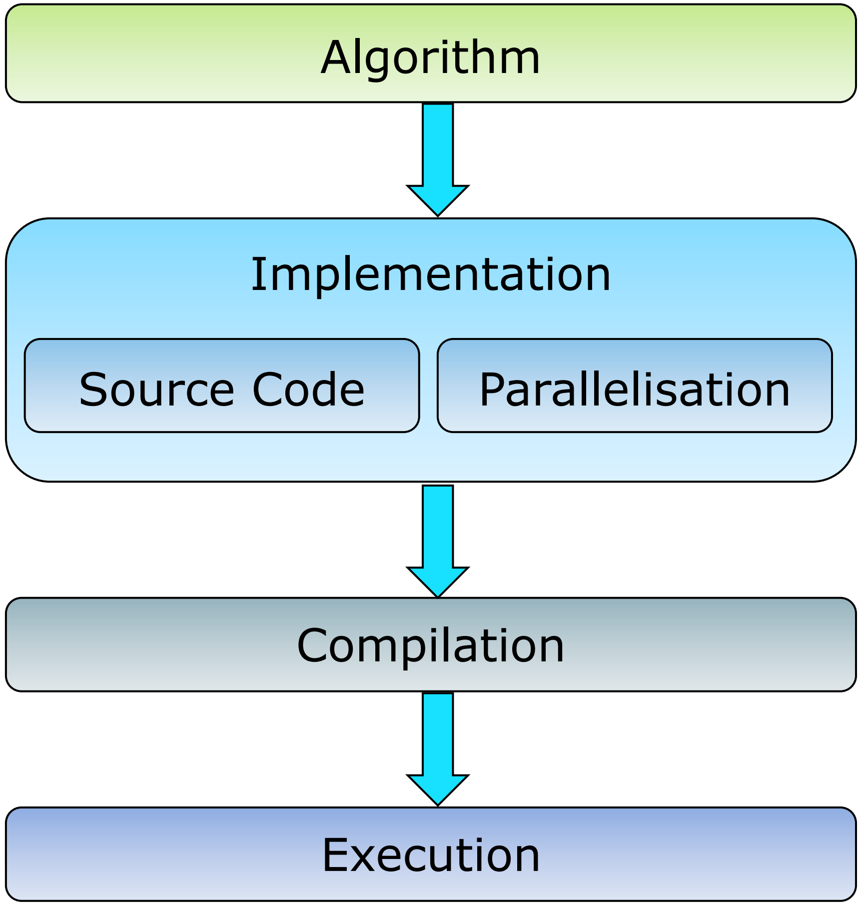
Performance analysis and optimisation

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?



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

6) Variable number of iterations

2) Non-unit stride accesses

1) High number of statements

```
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
```

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?

- CPU or data access problems
- Identifying the dominant issues: Algorithms, implementation, parallelisation, …

What transformations to apply?

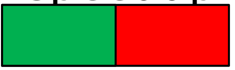
- Compiler switches, Partial/full vectorization
- Loop blocking/array restructuring, If removal, Full unroll
- Binary tranforms (prefetch),
- …

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

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**

Our Approach

Nobody wants problems everybody wants solutions ☺

- Focusing on the knobs that code developers can operate:
  - Compiler flags and runtime settings
  - Code restructuring
  - Data restructuring
- Helping the user in using these knobs

➔ Instead of 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

➔ 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 level**
- **Guiding users** through the optimization process
- Estimating return on investment (**R.O.I.**)

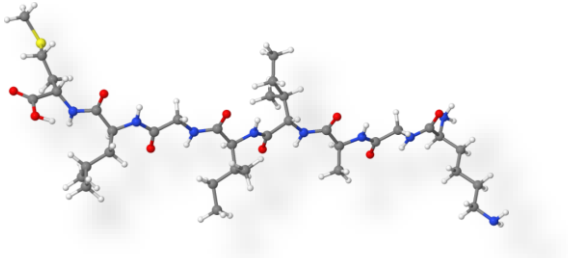
Characteristics:

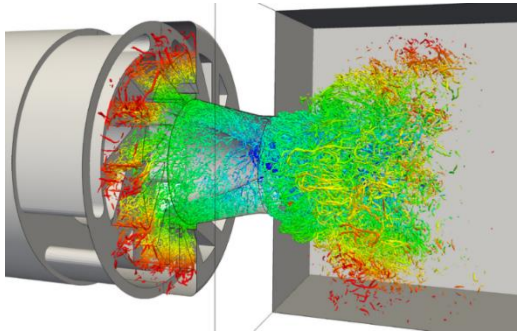
- **Modular tool** offering complementary views
- Support for **x86-64** and **Aarch64** (beta version)
  - Work in progress on GPU support
- LGPL3 Open Source software
- Developed at UVSQ since 2004
- Binary release available as a **static executable**

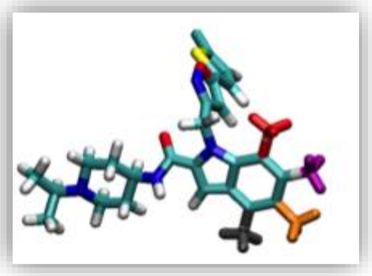
Success stories

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







Partnerships

MAQAO is part of the POP Centre of Excellence

- Provides performance optimisation and productivity services for academic and industrial codes
- 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 team and collaborators

MAQAO Team

- William Jalby, Prof.
- Cédric Valensi, Ph.D.
- Emmanuel Oseret, Ph.D.
- Mathieu Tribalat, M.Sc.Eng.
- Jäsper Salah Ibnamar, M.Sc.Eng.
- Hugo Bolloré , M.Sc.Eng
- Kévin Camus, Eng.
- Aurélien Delval, Eng.
- Max Hoffer, Eng.

Collaborators

- David J. Kuck, Prof. (Intel US)
- Pablo de Oliveira, Prof. (UVSQ)
- Eric Petit, Ph.D. (Intel US)
- David C. Wong, Ph.D. (Intel US)
- Othman Bouizi, Ph.D. (Intel US)
- AbdelHafid Mazouz Ph.D.(Intel)
- Jeongnim Kim (Intel)

Past Collaborators or Team Members

- *Andrés S. Charif-Rubial, Ph.D.*
- Denis Barthou, Prof. (Univ. Bordeaux)
- Jean-Thomas Acquaviva, Ph.D. (DDN)
- Stéphane Zuckerman, Ph.D. (ENSEA)
- Julien Jaeger, Ph.D. (CEA DAM)
- Souad Koliaï, Ph.D. (CELOXICA)
- Zakaria Bendifallah, Ph.D. (ATOS)
- Tipp Moseley, Ph.D. (Google)
- Jean-Christophe Beyler, Ph.D. (Google)
- Jean-Baptiste Le Reste, M.Sc.Eng. (start-up)
- Sylvain Henry, Ph.D. (start-up)
- José Noudohouenou, Ph.D. (Intel US)
- Aleksandre Vardoshvili, M.Sc.Eng.
- Romain Pillot, Eng
- Youenn Lebras, Ph.D. (start-up)

More on MAQAO

MAQAO website: www.maqao.org

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

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

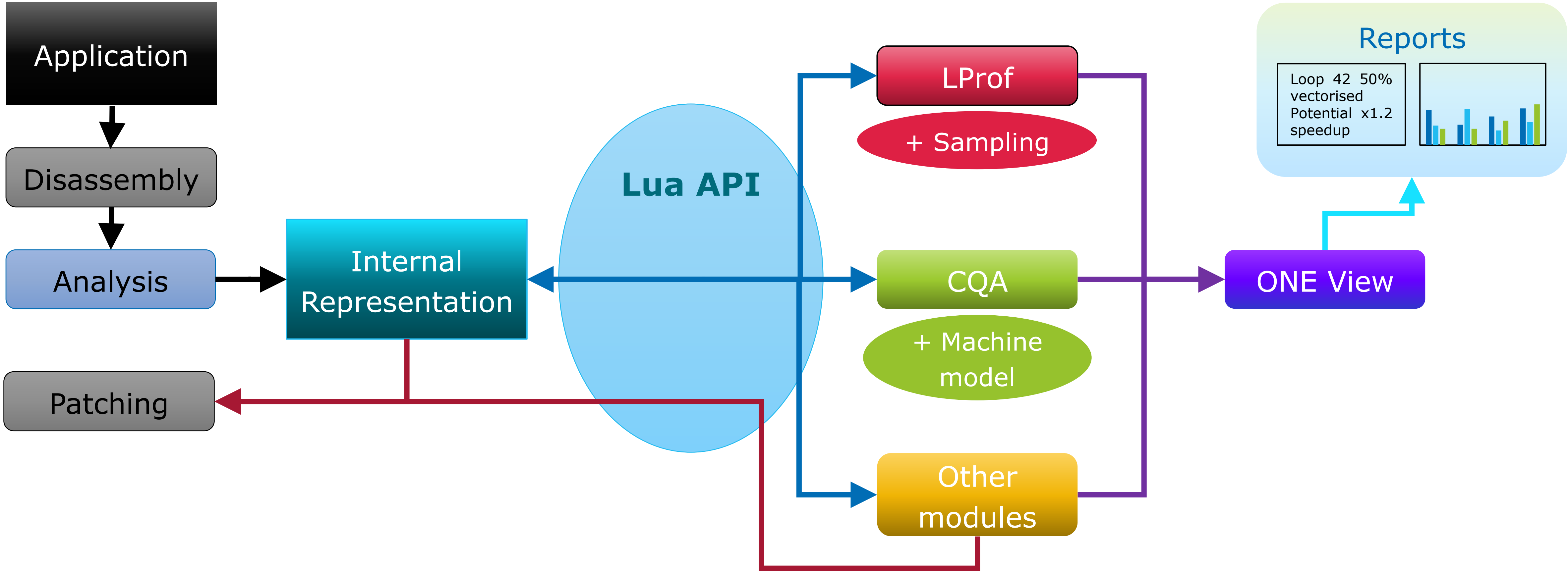
Static analysis

- 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
- Automatizes execution of experiments invoking other MAQAO modules and aggregates their results to produce high-level reports in HTML or XLSX format

MAQAO Main structure



Application

Disassembly

Analysis

Internal Representation

Patching

Lua API

LProf

+ Sampling

CQA

+ Machine model

Other modules

ONE View

Reports

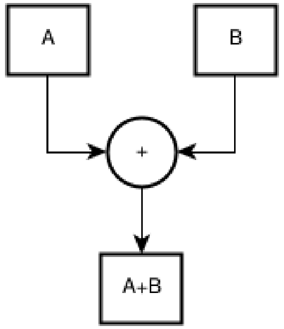
Loop 42 50% vectorised
Potential x1.2 speedup

Useful notions

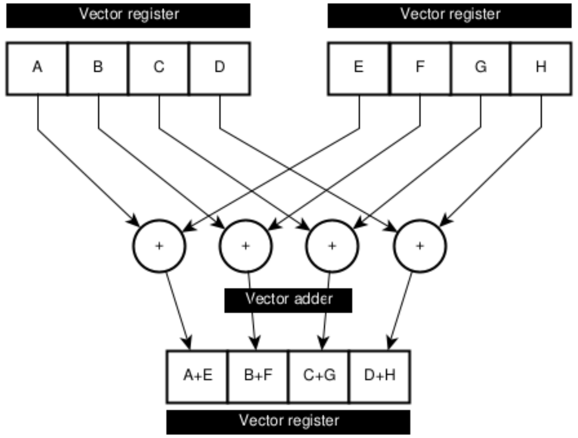
SIMD/Vectorization/Data Parallelism

- 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

Scalar addition

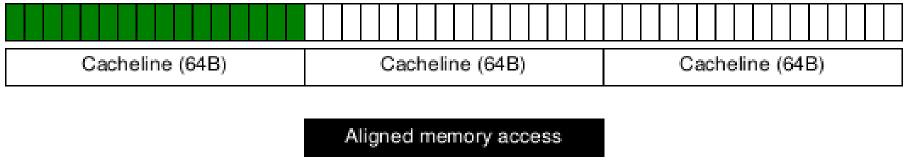


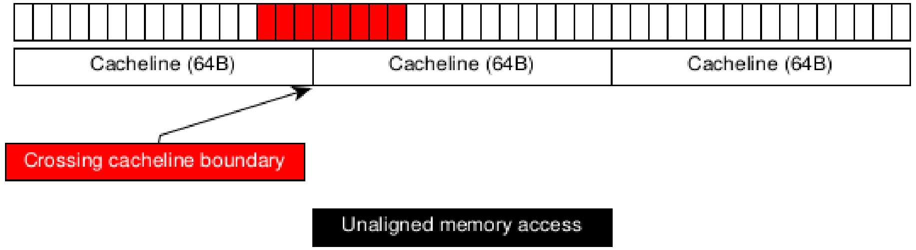
Vector addition



Memory and caches

- Computations are in general faster than memory accesses
- Alignment/Contiguity of memory (x86) : **posix_memalign**, **aligned_alloc**, …
- Caches: L1, L2, L3, …





MAQAO LProf: Lightweight Profiler

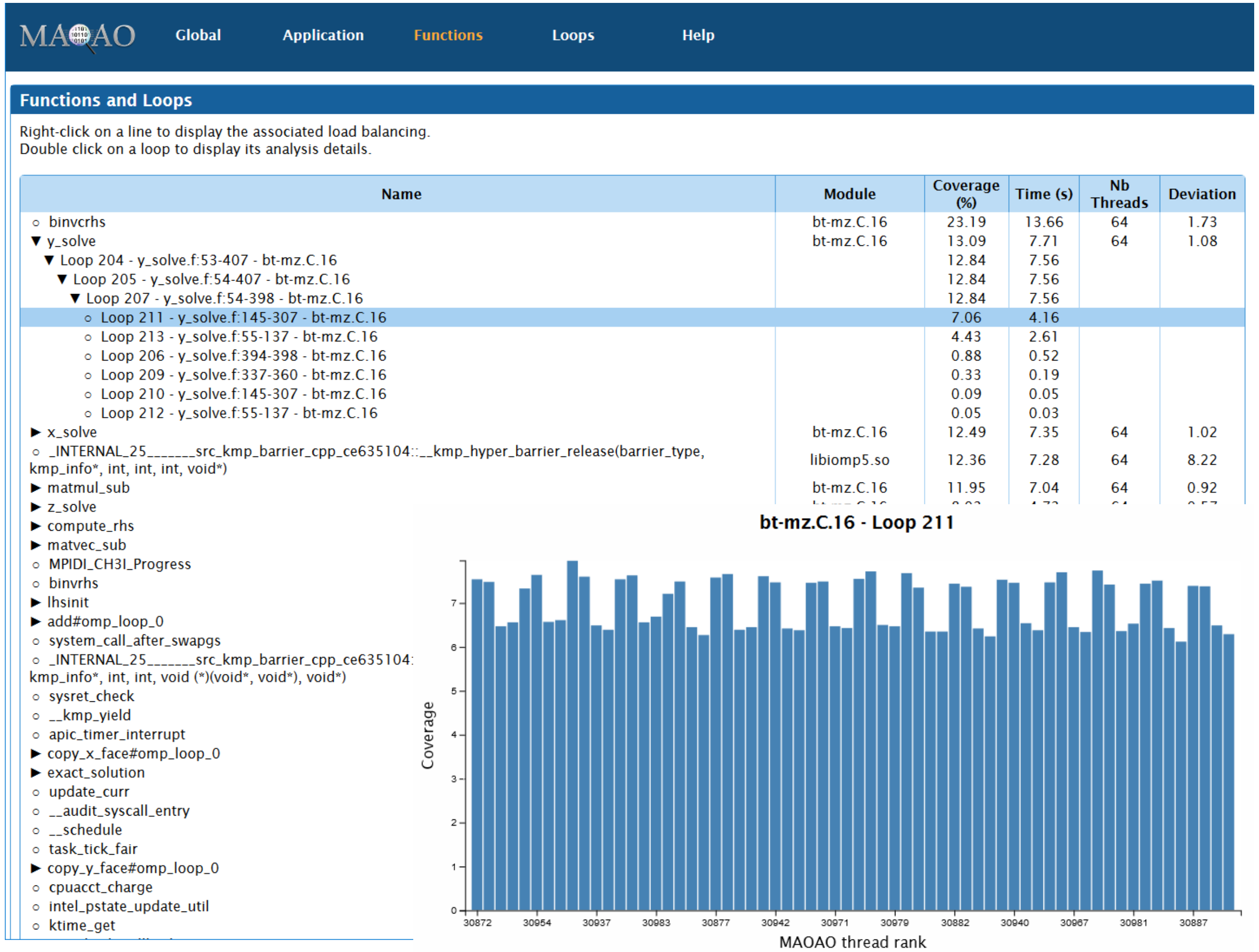
Goal: Lightweight localization of application hotspots

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



Functions and Loops

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

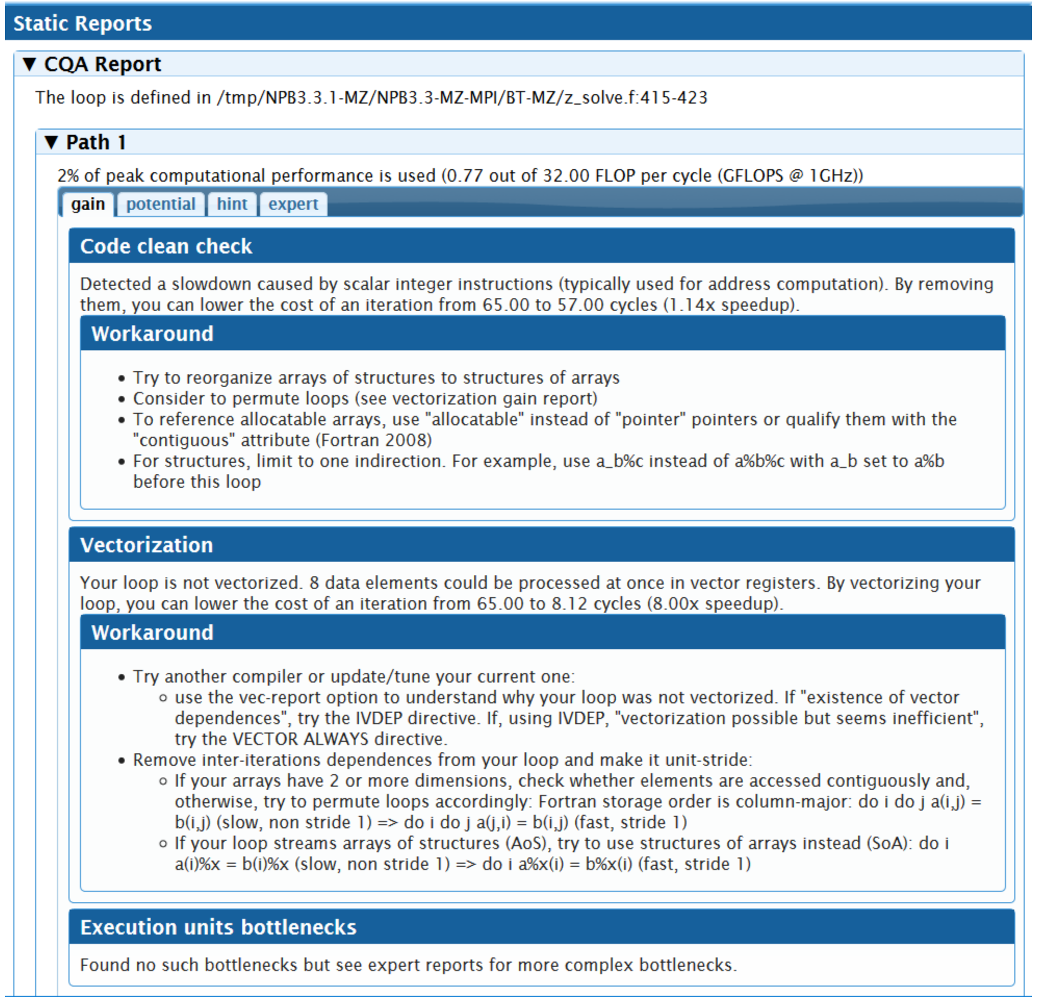
Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation
○ binvcrhs	bt-mz.C.16	23.19	13.66	64	1.73
▼ y_solve	bt-mz.C.16	13.09	7.71	64	1.08
▼ Loop 204 - y_solve.f:53-407 - bt-mz.C.16		12.84	7.56		
▼ Loop 205 - y_solve.f:54-407 - bt-mz.C.16		12.84	7.56		
▼ Loop 207 - y_solve.f:54-398 - bt-mz.C.16		12.84	7.56		
○ Loop 211 - y_solve.f:145-307 - bt-mz.C.16		7.06	4.16		
○ Loop 213 - y_solve.f:55-137 - bt-mz.C.16		4.43	2.61		
○ Loop 206 - y_solve.f:394-398 - bt-mz.C.16		0.88	0.52		
○ Loop 209 - y_solve.f:337-360 - bt-mz.C.16		0.33	0.19		
○ Loop 210 - y_solve.f:145-307 - bt-mz.C.16		0.09	0.05		
○ Loop 212 - y_solve.f:55-137 - bt-mz.C.16		0.05	0.03		
▶ x_solve	bt-mz.C.16	12.49	7.35	64	1.02
○ _INTERNAL_25_______src_kmp_barrier_cpp_ce635104::__kmp_hyper_barrier_release(barrier_type, kmp_info*, int, int, int, void*)	libiomp5.so	12.36	7.28	64	8.22
▶ matmul_sub	bt-mz.C.16	11.95	7.04	64	0.92
▶ z_solve					
▶ compute_rhs					
▶ matvec_sub					
○ MPIDI_CH3I_Progress					
○ binvrhs					
▶ lhsinit					
▶ add#omp_loop_0					
○ system_call_after_swapgs					
○ _INTERNAL_25_______src_kmp_barrier_cpp_ce635104: kmp_info*, int, int, void (*)(void*, void*), void*)					
○ sysret_check					
○ __kmp_yield					
○ apic_timer_interrupt					
▶ copy_x_face#omp_loop_0					
▶ exact_solution					
○ update_curr					
○ __audit_syscall_entry					
○ __schedule					
○ task_tick_fair					
▶ copy_y_face#omp_loop_0					
○ cpuacct_charge					
○ intel_pstate_update_util					
○ ktime_get					

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



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 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

“What If” Scenarios: Vectorization

Code “Clean”

- Generate an Assembly “Clean” variant : **keep only FP Arithmetic and Memory operations, suppress all other**
- Generate a CQA Performance estimate on the “Clean” Variant

Code “FP Vector”

- Generate an Assembly “FP Vector” variant : **only replace scalar FP Arithmetic by Vector FP Arithmetic equivalent**. Generate additional instructions to fill in Vector Registers.
- Generate a CQA Performance estimate

Code “Full Vector”

- Generate an Assembly “Full Vector” variant **: replace both scalar FP Arithmetic and FP Load/Store by their Vector equivalent**.
- Generate a CQA Performance estimate

All of these “What If Scenarios” are generated in a fully static manner.

MAQAO CQA: Guiding the compiler and implementation hints

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

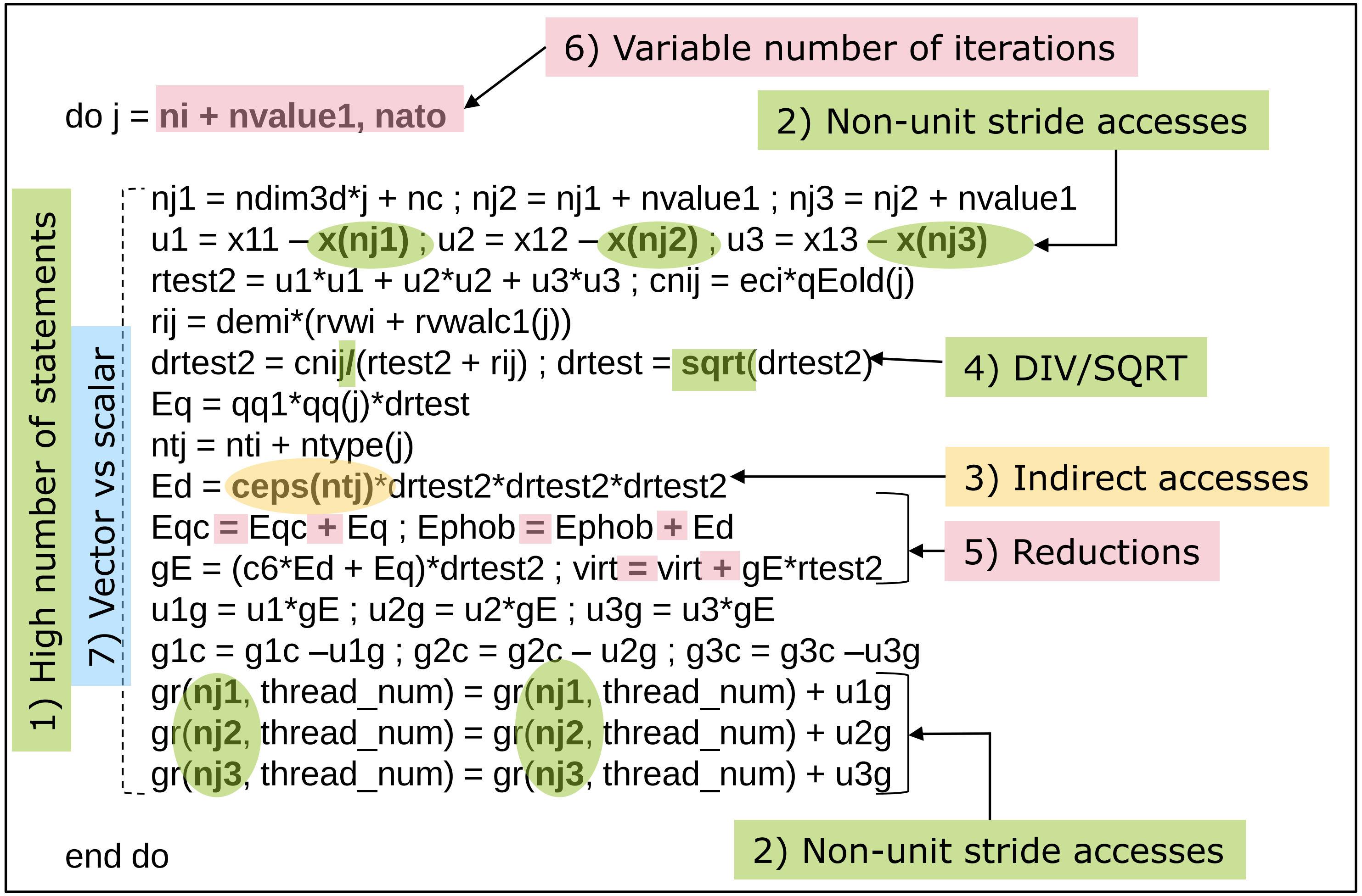
- Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
- Forcing optimizations (unrolling, vectorization, alignment, …)
- Bypassing conservative behaviour when possible (e.g. 1/X precision)

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, …)

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
```

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 ONE View: Performance View Aggregator

Automating the whole analysis process

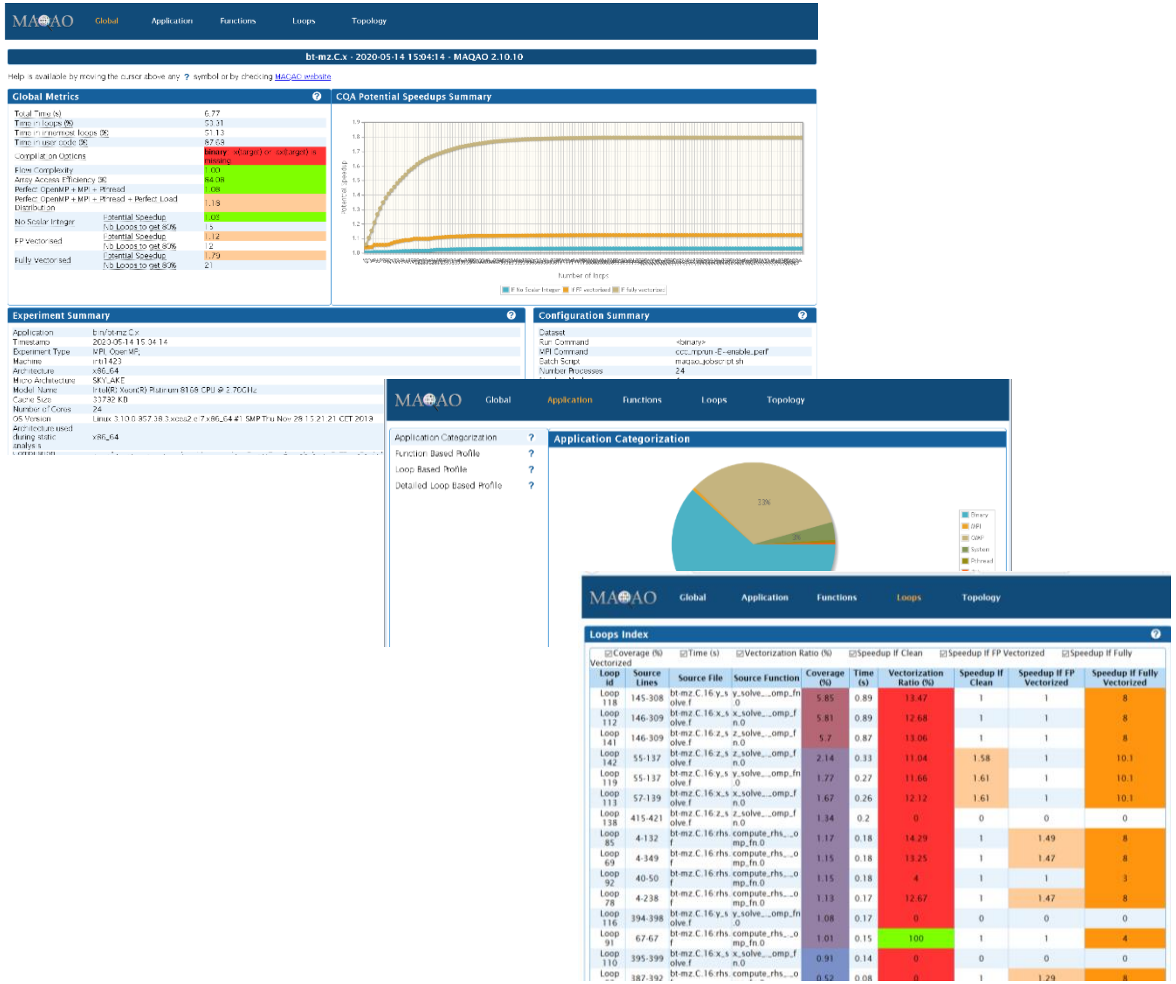
- Invoke multiple MAQAO modules
- Generate **aggregated performance views**
- Generate a report in HTML format

Main steps:

- Invokes LProf to **identify hotspots**
- Invokes CQA on **hotspots**

Available results:

- **Speedup** predictions
- **High-level** summary
- Global **code quality** metrics
- **Hints** for improving performance
- Parallel **efficiency** analysis



bt-mz.C.x - 2020-05-14 15:04:14 - MAQAO 2.10.10

Help is available by moving the cursor above any ? symbol or by checking MAQAO website

Global Metrics

Total Time (s)		6.77
Time in loops (%)		53.31
Time in innermost loops (%)		51.13
Time in user code (%)		87.68
Compilation Options		binary: -x(target) or -ax(target) is missing
Flow Complexity		1.00
Array Access Efficiency (%)		84.08
Perfect OpenMP + MPI + Pthread		1.08
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.18
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	15
FP Vectorised	Potential Speedup	1.12
	Nb Loops to get 80%	12
Fully Vectorised	Potential Speedup	1.79
	Nb Loops to get 80%	21

Experiment Summary

Application	bin/bt-mz.C.x
Timestamp	2020-05-14 15:04:14
Experiment Type	MPI, OpenMP;
Machine	irtu1423
Architecture	x86_64
Micro Architecture	SKYLAKE
Model Name	Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz
Cache Size	33792 KB
Number of Cores	24
OS Version	Linux 3.10.0-957.38.3.xcca2.el7.x86_64 #1 SMP Thu Nov 28 15:21:21 CET 2019
Architecture used during static analysis	x86_64

Configuration Summary

Dataset	
Run Command	<binary>
MPI Command	ccc_mprun -E--enable_perf'
Batch Script	maqao_jobscript.sh
Number Processes	24

Application Categorization ?
Function Based Profile ?
Loop Based Profile ?
Detailed Loop Based Profile ?

Application Categorization

Loops Index

☑Coverage (%) ☑Time (s) ☑Vectorization Ratio (%) ☑Speedup If Clean ☑Speedup If FP Vectorized ☑Speedup If Fully Vectorized

Loop id	Source Lines	Source File	Source Function	Coverage (%)	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized
Loop 118	145-308	bt-mz.C.16:y_solve.f	y_solve_._omp_fn.0	5.85	0.89	13.47	1	1	8
Loop 112	146-309	bt-mz.C.16:x_solve.f	x_solve_._omp_fn.0	5.81	0.89	12.68	1	1	8
Loop 141	146-309	bt-mz.C.16:z_solve.f	z_solve_._omp_fn.0	5.7	0.87	13.06	1	1	8
Loop 142	55-137	bt-mz.C.16:z_solve.f	z_solve_._omp_fn.0	2.14	0.33	11.04	1.58	1	10.1
Loop 119	55-137	bt-mz.C.16:y_solve.f	y_solve_._omp_fn.0	1.77	0.27	11.66	1.61	1	10.1
Loop 113	57-139	bt-mz.C.16:x_solve.f	x_solve_._omp_fn.0	1.67	0.26	12.12	1.61	1	10.1
Loop 138	415-421	bt-mz.C.16:z_solve.f	z_solve_._omp_fn.0	1.34	0.2	0	0	0	0
Loop 85	4-132	bt-mz.C.16:rhs.f	compute_rhs_._omp_fn.0	1.17	0.18	14.29	1	1.49	8
Loop 69	4-349	bt-mz.C.16:rhs.f	compute_rhs_._omp_fn.0	1.15	0.18	13.25	1	1.47	8
Loop 92	40-50	bt-mz.C.16:rhs.f	compute_rhs_._omp_fn.0	1.15	0.18	4	1	1	3
Loop 78	4-238	bt-mz.C.16:rhs.f	compute_rhs_._omp_fn.0	1.13	0.17	12.67	1	1.47	8
Loop 116	394-398	bt-mz.C.16:y_solve.f	y_solve_._omp_fn.0	1.08	0.17	0	0	0	0
Loop 91	67-67	bt-mz.C.16:rhs.f	compute_rhs_._omp_fn.0	1.01	0.15	100	1	1	4
Loop 110	395-399	bt-mz.C.16:x_solve.f	x_solve_._omp_fn.0	0.91	0.14	0	0	0	0
Loop	387-392	bt-mz.C.16:rhs.	compute_rhs_._o	0.52	0.08	0	1	1.29	8

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 binary name, MPI commands, dataset directory, …

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

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

- Analyses can be tweaked if necessary
  - Report level **one** corresponds to lightweight profiling (**LProf**) and code quality analysis (**CQA**)
- 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

Online help is available:

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

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
```

Online help is available:

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

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

Navigating ONE View Reports

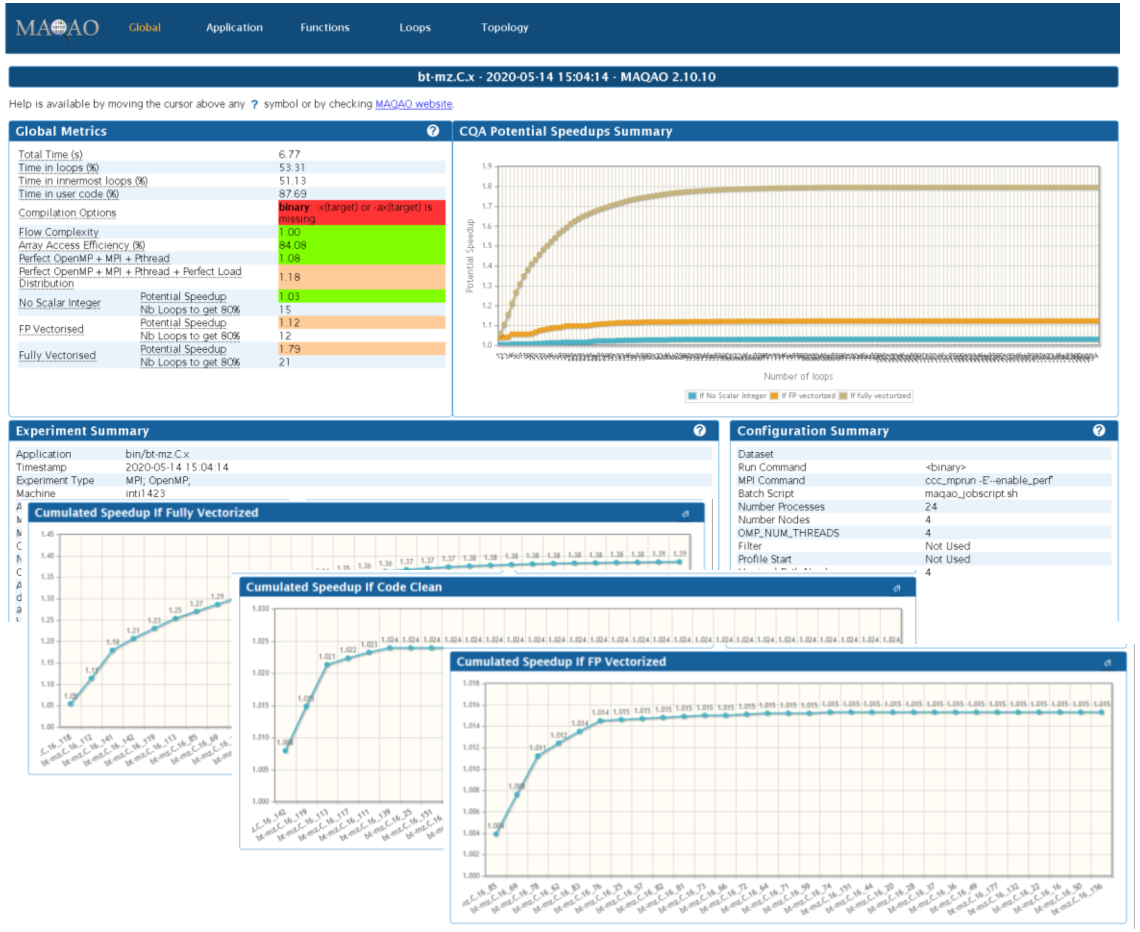
Global summary

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



bt-mz.C.x - 2020-05-14 15:04:14 - MAQAO 2.10.10

Help is available by moving the cursor above any ? symbol or by checking MAQAO website.

Global Metrics

Total Time (s)		6.77
Time in loops (%)		53.31
Time in innermost loops (%)		51.13
Time in user code (%)		87.69
Compilation Options		**binary**: -x(target) or -ax(target) is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		84.08
Perfect OpenMP + MPI + Pthread		1.08
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.18
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	15
FP Vectorised	Potential Speedup	1.12
	Nb Loops to get 80%	12
Fully Vectorised	Potential Speedup	1.79
	Nb Loops to get 80%	21

Experiment Summary

Application	bin/bt-mz.C.x
Timestamp	2020-05-14 15:04:14
Experiment Type	MPI; OpenMP;
Machine	inti1423

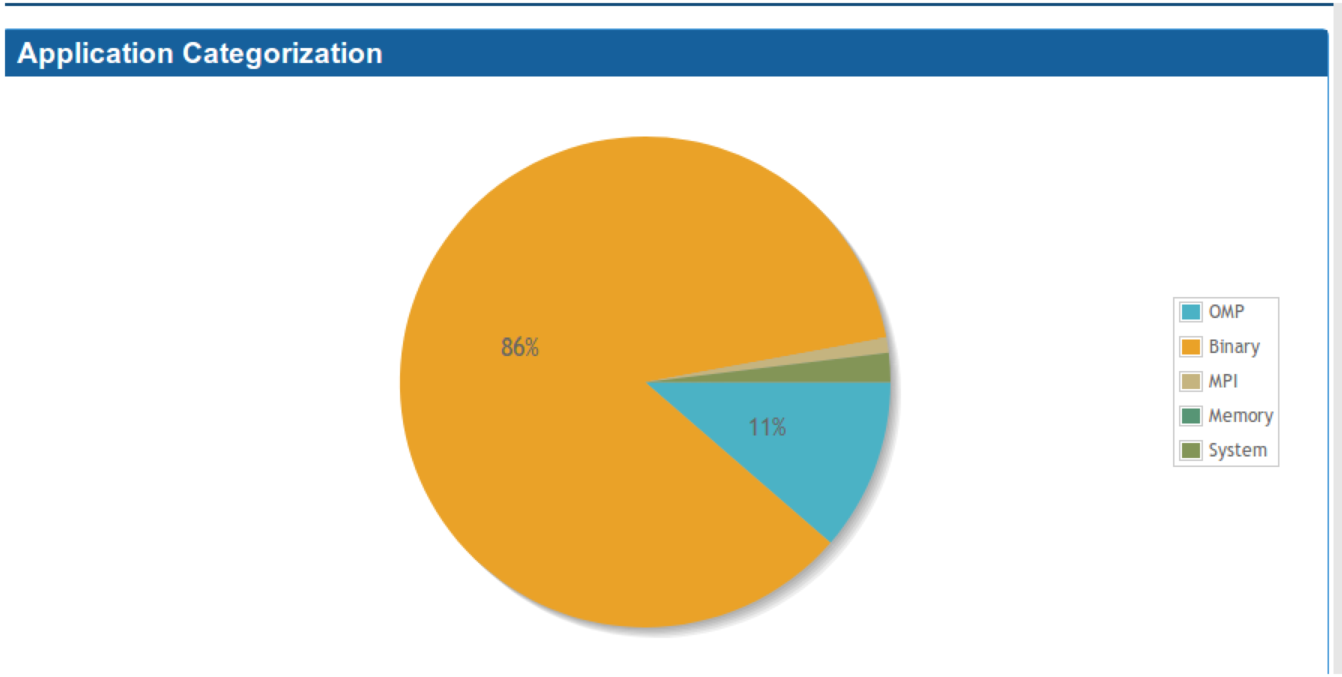
Configuration Summary

Dataset	
Run Command	<binary>
MPI Command	ccc_mprun -E'--enable_perf'
Batch Script	maqao_jobscript.sh
Number Processes	24
Number Nodes	4
OMP_NUM_THREADS	4
Filter	Not Used
Profile Start	Not Used
	4

Time Categorisation

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



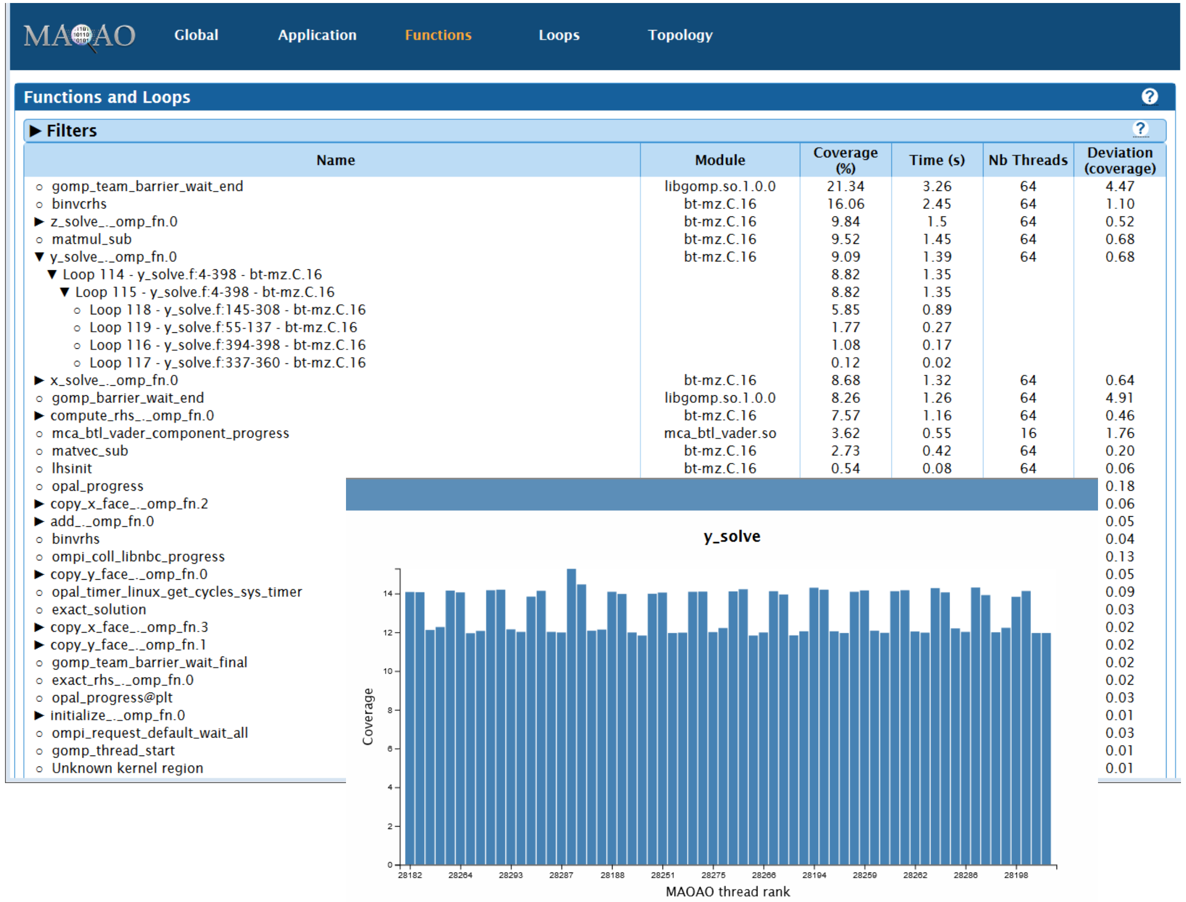
Functions Profiling

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 Outermost
  - ▼ Loop 233 - z_solve.f:54-423 - bt-mz.C.16 Inbetween
    - ▼ Loop 236 - z_solve.f:54-423 - bt-mz.C.16 Inbetween
      - ○ Loop 239 - z_solve.f:146-308 - bt-mz.C.16 Innermost
      - ○ Loop 235 - z_solve.f:55-137 - bt-mz.C.16
      - ○ Loop 234 - z_solve.f:415-423 - bt-mz.C.16

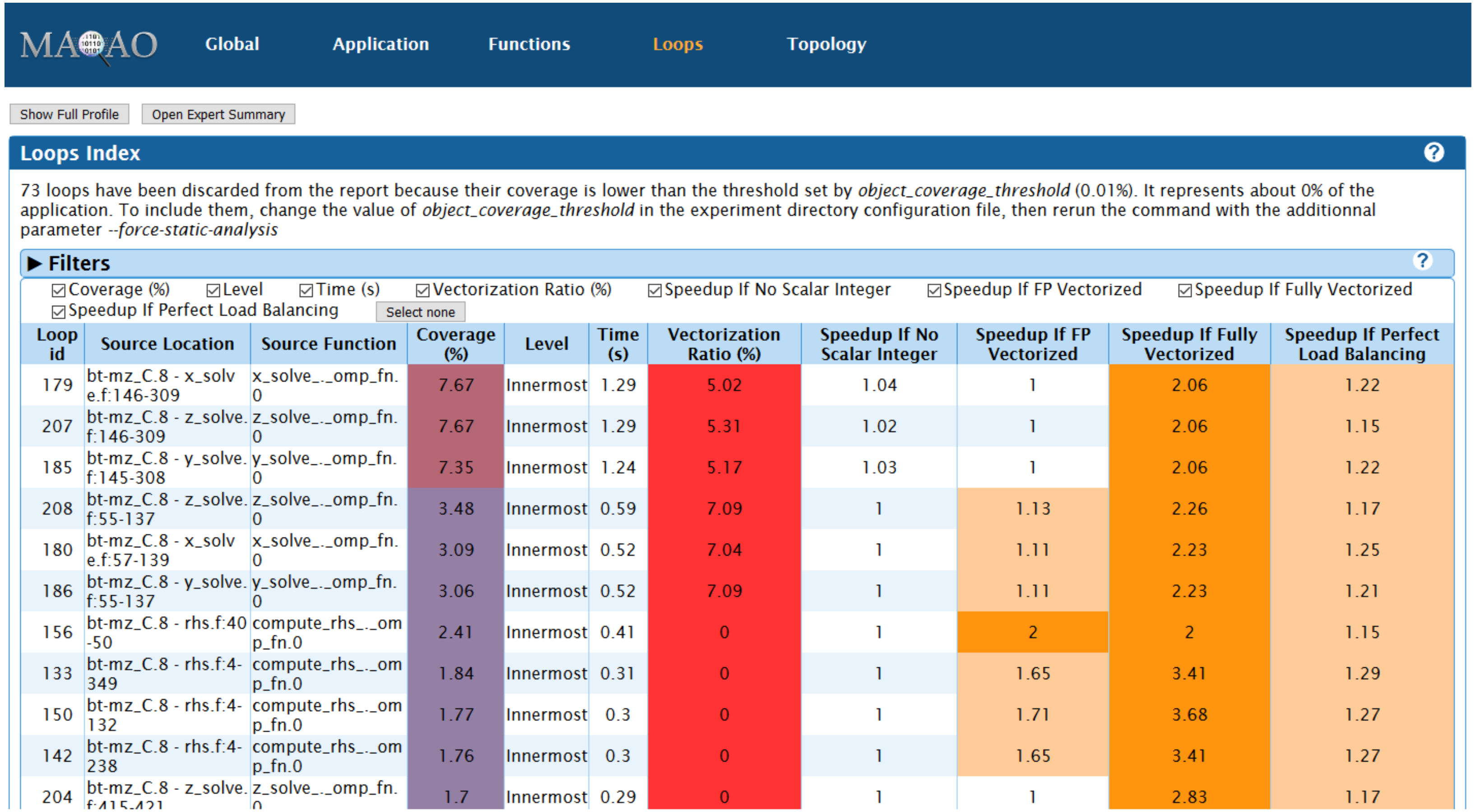


Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
○ gomp_team_barrier_wait_end	libgomp.so.1.0.0	21.34	3.26	64	4.47
○ binvcrhs	bt-mz.C.16	16.06	2.45	64	1.10
▶ z_solve_._omp_fn.0	bt-mz.C.16	9.84	1.5	64	0.52
○ matmul_sub	bt-mz.C.16	9.52	1.45	64	0.68
▼ y_solve_._omp_fn.0	bt-mz.C.16	9.09	1.39	64	0.68
▼ Loop 114 - y_solve.f:4-398 - bt-mz.C.16		8.82	1.35		
▼ Loop 115 - y_solve.f:4-398 - bt-mz.C.16		8.82	1.35		
○ Loop 118 - y_solve.f:145-308 - bt-mz.C.16		5.85	0.89		
○ Loop 119 - y_solve.f:55-137 - bt-mz.C.16		1.77	0.27		
○ Loop 116 - y_solve.f:394-398 - bt-mz.C.16		1.08	0.17		
○ Loop 117 - y_solve.f:337-360 - bt-mz.C.16		0.12	0.02		
▶ x_solve_._omp_fn.0	bt-mz.C.16	8.68	1.32	64	0.64
○ gomp_barrier_wait_end	libgomp.so.1.0.0	8.26	1.26	64	4.91
▶ compute_rhs_._omp_fn.0	bt-mz.C.16	7.57	1.16	64	0.46
○ mca_btl_vader_component_progress	mca_btl_vader.so	3.62	0.55	16	1.76
○ matvec_sub	bt-mz.C.16	2.73	0.42	64	0.20
○ lhsinit	bt-mz.C.16	0.54	0.08	64	0.06
○ opal_progress					0.18
▶ copy_x_face_._omp_fn.2					0.06
▶ add_._omp_fn.0					0.05
○ binvrhs					0.04
○ ompi_coll_libnbc_progress					0.13
▶ copy_y_face_._omp_fn.0					0.05
○ opal_timer_linux_get_cycles_sys_timer					0.09
○ exact_solution					0.03
▶ copy_x_face_._omp_fn.3					0.02
▶ copy_y_face_._omp_fn.1					0.02
○ gomp_team_barrier_wait_final					0.02
○ exact_rhs_._omp_fn.0					0.02
○ opal_progress@plt					0.03
▶ initialize_._omp_fn.0					0.01
○ ompi_request_default_wait_all					0.03
○ gomp_thread_start					0.01
○ Unknown kernel region					0.01

Loops 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



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 additionnal parameter *--force-static-analysis*

▶ **Filters**

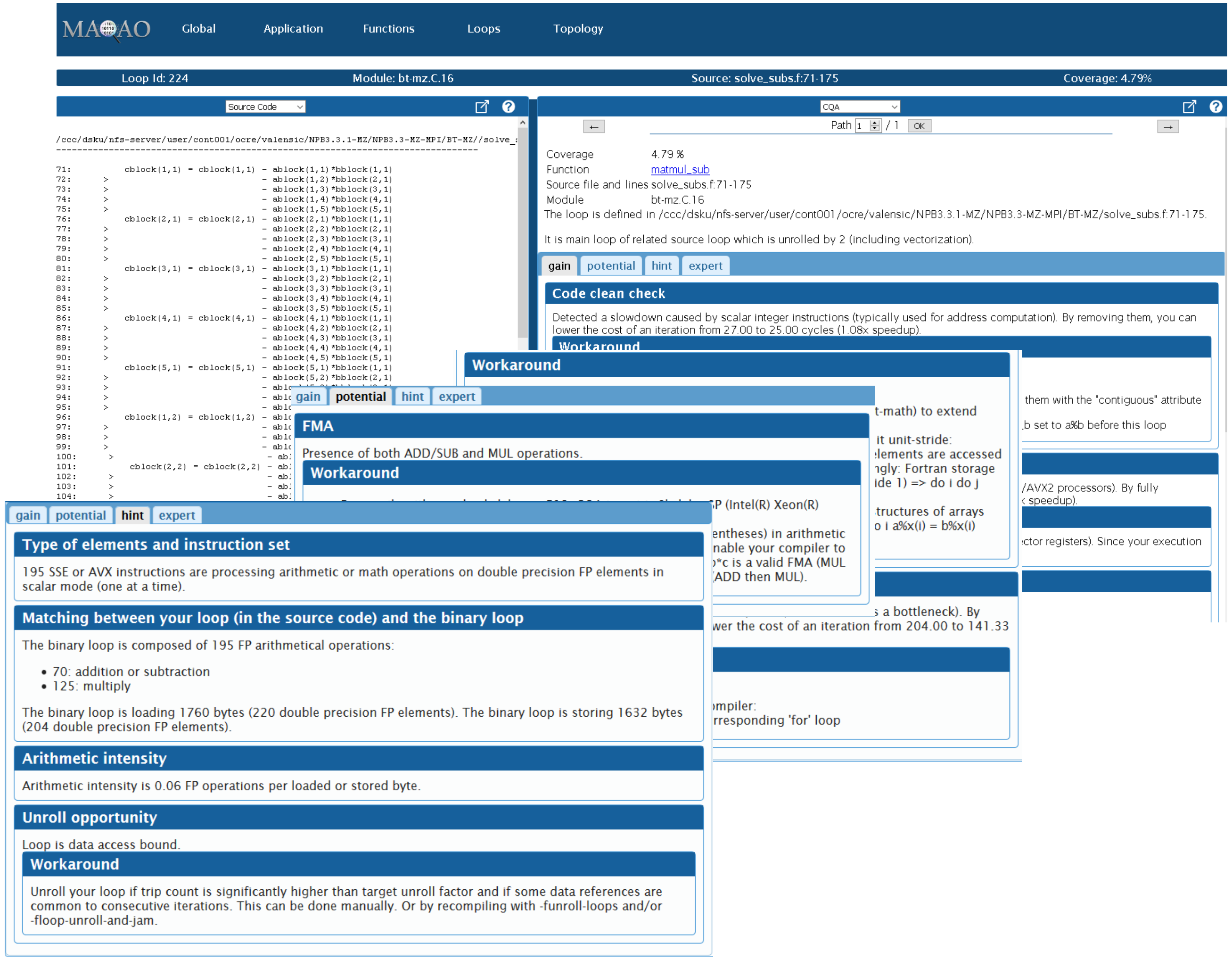
☑ Coverage (%) ☑ Level ☑ Time (s) ☑ Vectorization Ratio (%) ☑ Speedup If No Scalar Integer ☑ Speedup If FP Vectorized ☑ Speedup If Fully Vectorized ☑ Speedup If Perfect Load Balancing

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.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.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:415-421	z_solve_._omp_fn.0	1.7	Innermost	0.29	0	1	1	2.83	1.17

Loop Analysis Reports

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



Loop Id: 224 Module: bt-mz.C.16 Source: solve_subs.f:71-175 Coverage: 4.79%

```
/ccc/dsku/nfs-server/user/cont001/ocre/valensic/NPB3.3.1-MZ/NPB3.3-MZ-MPI/BT-MZ//solve_
--------------------------------------------------------------------------------------

         cblock(1,1) = cblock(1,1) - ablock(1,1)*bblock(1,1)
     >                             - ablock(1,2)*bblock(2,1)
     >                             - ablock(1,3)*bblock(3,1)
     >                             - ablock(1,4)*bblock(4,1)
     >                             - ablock(1,5)*bblock(5,1)
         cblock(2,1) = cblock(2,1) - ablock(2,1)*bblock(1,1)
     >                             - ablock(2,2)*bblock(2,1)
     >                             - ablock(2,3)*bblock(3,1)
     >                             - ablock(2,4)*bblock(4,1)
     >                             - ablock(2,5)*bblock(5,1)
         cblock(3,1) = cblock(3,1) - ablock(3,1)*bblock(1,1)
     >                             - ablock(3,2)*bblock(2,1)
     >                             - ablock(3,3)*bblock(3,1)
     >                             - ablock(3,4)*bblock(4,1)
     >                             - ablock(3,5)*bblock(5,1)
         cblock(4,1) = cblock(4,1) - ablock(4,1)*bblock(1,1)
     >                             - ablock(4,2)*bblock(2,1)
     >                             - ablock(4,3)*bblock(3,1)
     >                             - ablock(4,4)*bblock(4,1)
     >                             - ablock(4,5)*bblock(5,1)
         cblock(5,1) = cblock(5,1) - ablock(5,1)*bblock(1,1)
     >                             - ablock(5,2)*bblock(2,1)
```

Coverage 4.79 %
Function matmul_sub
Source file and lines solve_subs.f:71-175
Module bt-mz.C.16
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

FMA

Presence of both ADD/SUB and MUL operations.

Workaround

Type of elements and instruction set

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

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

The binary loop is composed of 195 FP arithmetical operations:

- 70: addition or subtraction
- 125: multiply

The binary loop is loading 1760 bytes (220 double precision FP elements). The binary loop is storing 1632 bytes (204 double precision FP elements).

Arithmetic intensity

Arithmetic intensity is 0.06 FP operations per loaded or stored byte.

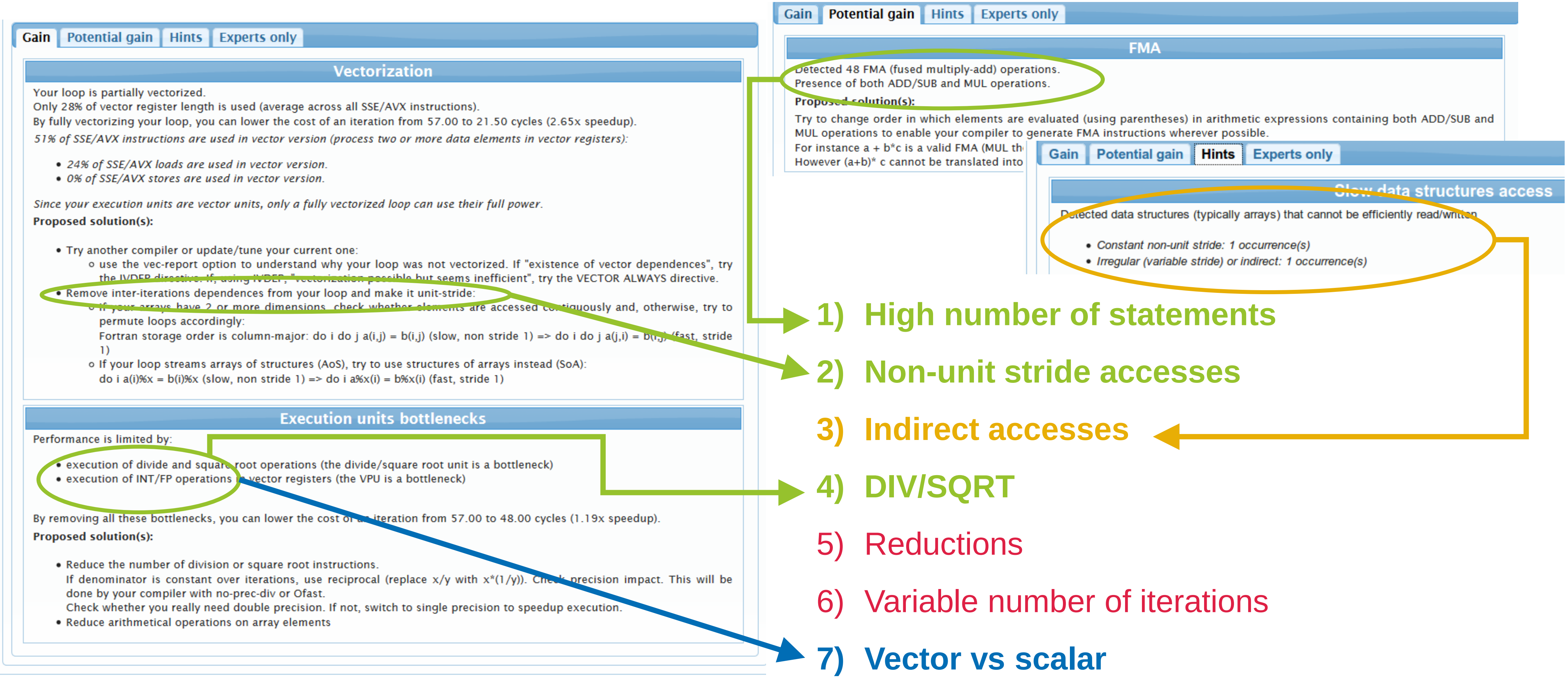
Unroll opportunity

Loop is data access bound.

Workaround

Unroll your loop if trip count is significantly higher than target unroll factor and if some data references are common to consecutive iterations. This can be done manually. Or by recompiling with -funroll-loops and/or -floop-unroll-and-jam.

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 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

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

Loop Analysis Reports – Expert View

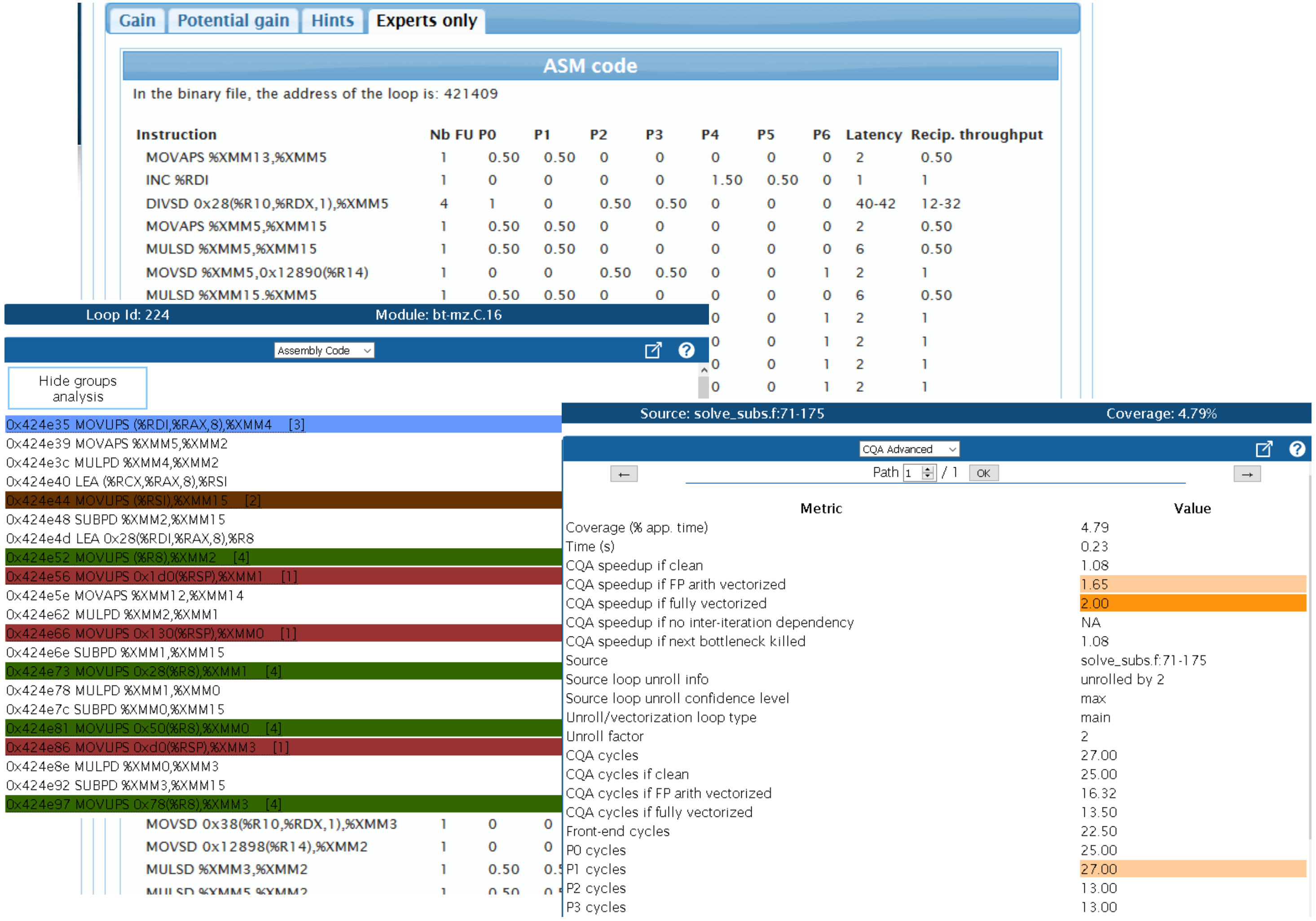
Low level reports for performance experts

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

Assembly code

- Highlights groups of instructions accessing the same memory addresses

CQA internal metrics



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
						0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
						0	0	1	2	1
MOVSD 0x38(%R10,%RDX,1),%XMM3	1	0	0							
MOVSD 0x12898(%R14),%XMM2	1	0	0							
MULSD %XMM3,%XMM2	1	0.50	0.5							

Loop Id: 224 Module: bt-mz.C.16

```
0x424e35 MOVUPS (%RDI,%RAX,8),%XMM4   [3]
0x424e39 MOVAPS %XMM5,%XMM2
0x424e3c MULPD %XMM4,%XMM2
0x424e40 LEA (%RCX,%RAX,8),%RSI
0x424e44 MOVUPS (%RSI),%XMM15   [2]
0x424e48 SUBPD %XMM2,%XMM15
0x424e4d LEA 0x28(%RDI,%RAX,8),%R8
0x424e52 MOVUPS (%R8),%XMM2   [4]
0x424e56 MOVUPS 0x1d0(%RSP),%XMM1   [1]
0x424e5e MOVAPS %XMM12,%XMM14
0x424e62 MULPD %XMM2,%XMM1
0x424e66 MOVUPS 0x130(%RSP),%XMM0   [1]
0x424e6e SUBPD %XMM1,%XMM15
0x424e73 MOVUPS 0x28(%R8),%XMM1   [4]
0x424e78 MULPD %XMM1,%XMM0
0x424e7c SUBPD %XMM0,%XMM15
0x424e81 MOVUPS 0x50(%R8),%XMM0   [4]
0x424e86 MOVUPS 0xd0(%RSP),%XMM3   [1]
0x424e8e MULPD %XMM0,%XMM3
0x424e92 SUBPD %XMM3,%XMM15
0x424e97 MOVUPS 0x78(%R8),%XMM3   [4]
```

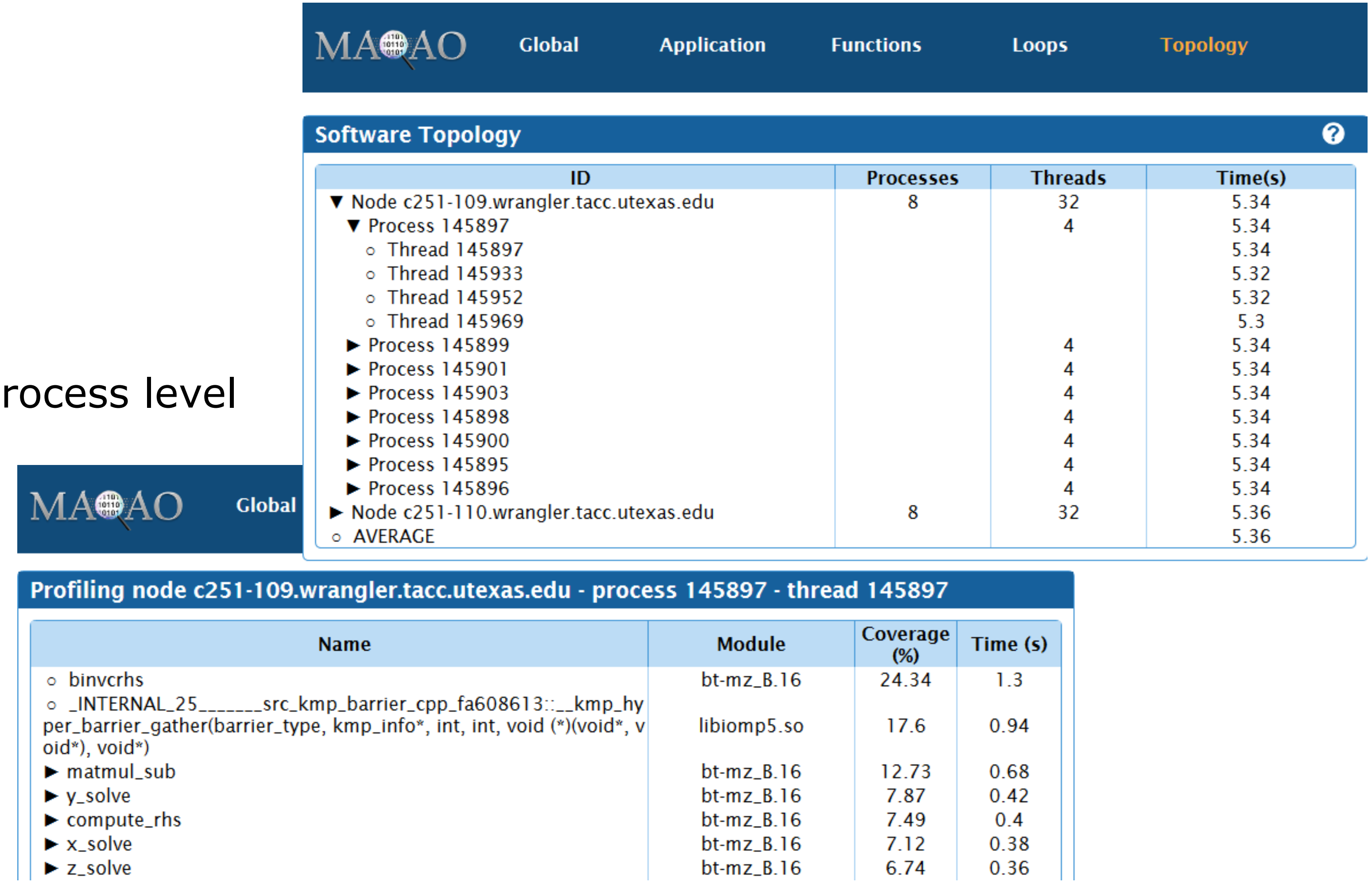
Source: solve_subs.f:71-175 Coverage: 4.79%

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

MAQAO ONE View Thread/Process View

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

View by thread
- Function profile at the thread or process level



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

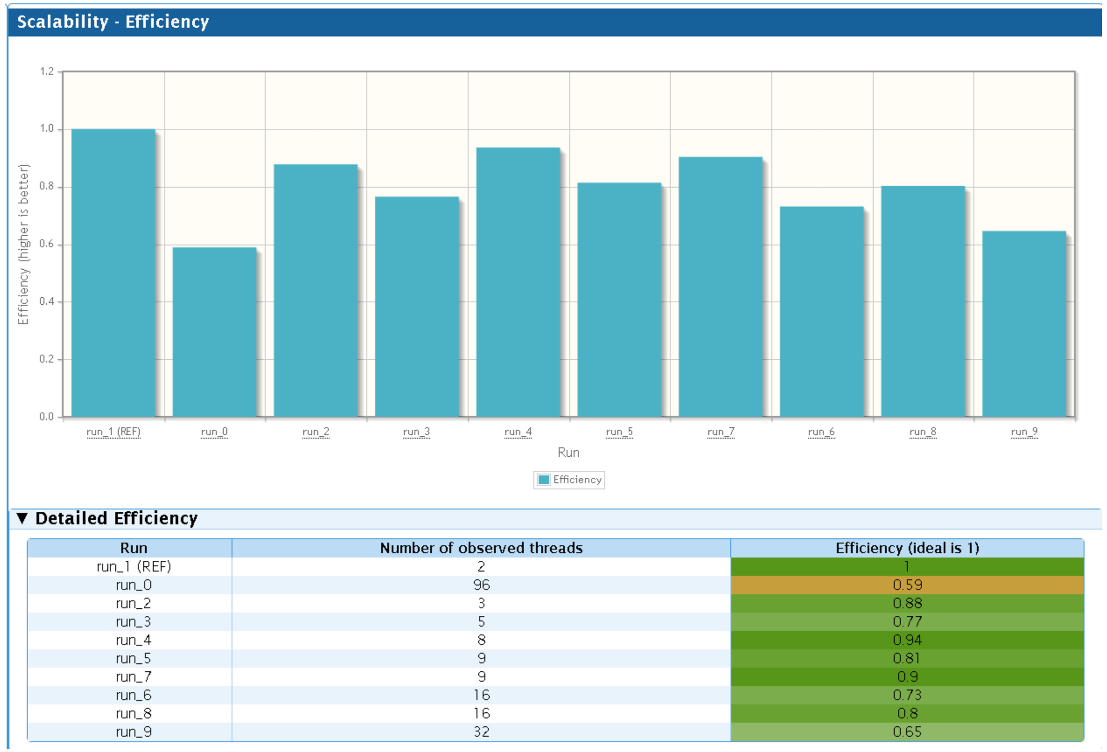
Profiling node c251-109.wrangler.tacc.utexas.edu - process 145897 - thread 145897

Name	Module	Coverage (%)	Time (s)
○ binvcrhs	bt-mz_B.16	24.34	1.3
○ _INTERNAL_25_______src_kmp_barrier_cpp_fa608613::__kmp_hyper_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

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



Scalability - Efficiency

▼ Detailed Efficiency

Run	Number of observed threads	Efficiency (ideal is 1)
run_1 (REF)	2	1
run_0	96	0.59
run_2	3	0.88
run_3	5	0.77
run_4	8	0.94
run_5	9	0.81
run_7	9	0.9
run_6	16	0.73
run_8	16	0.8
run_9	32	0.65

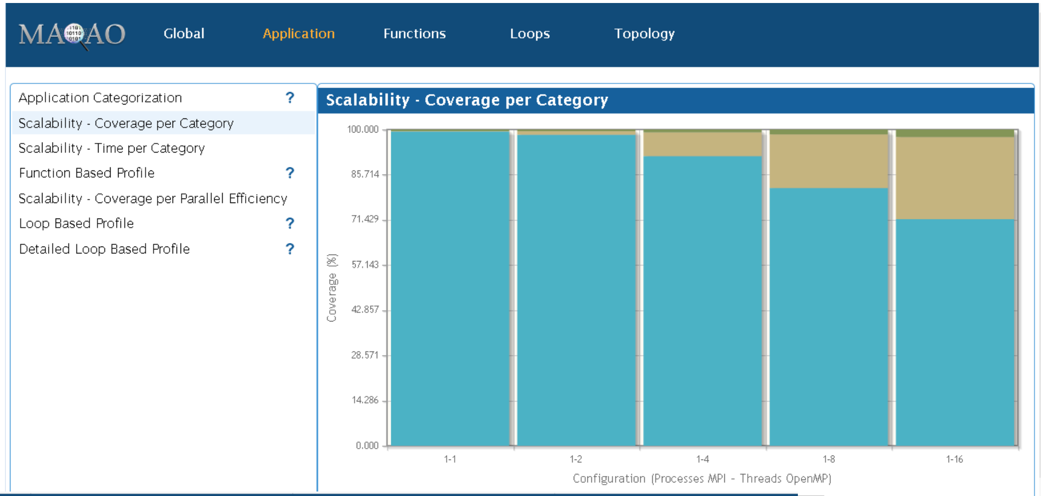
MAQAO ONE View Scalability Reports – Application View

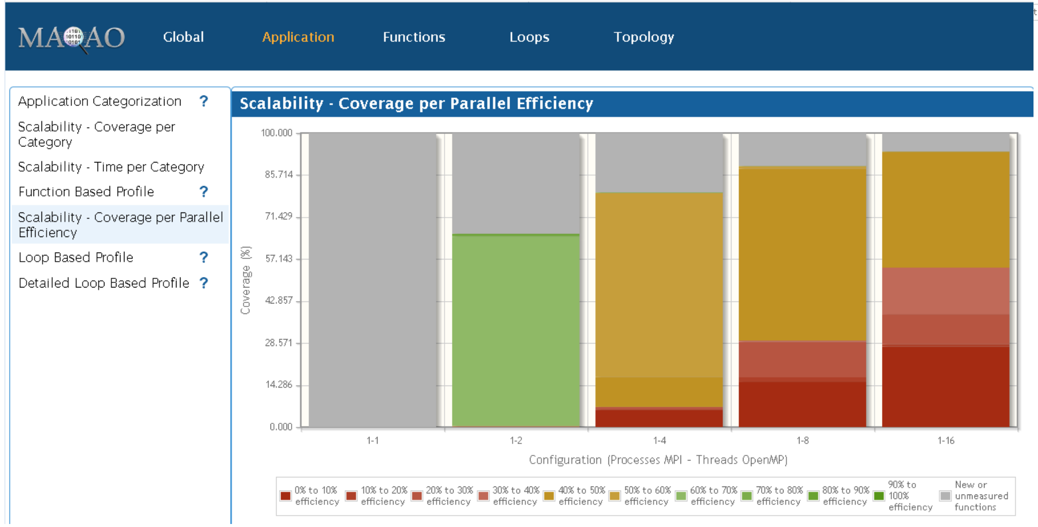
Coverage per category

- Comparison of categories for each run

Coverage per parallel efficiency

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

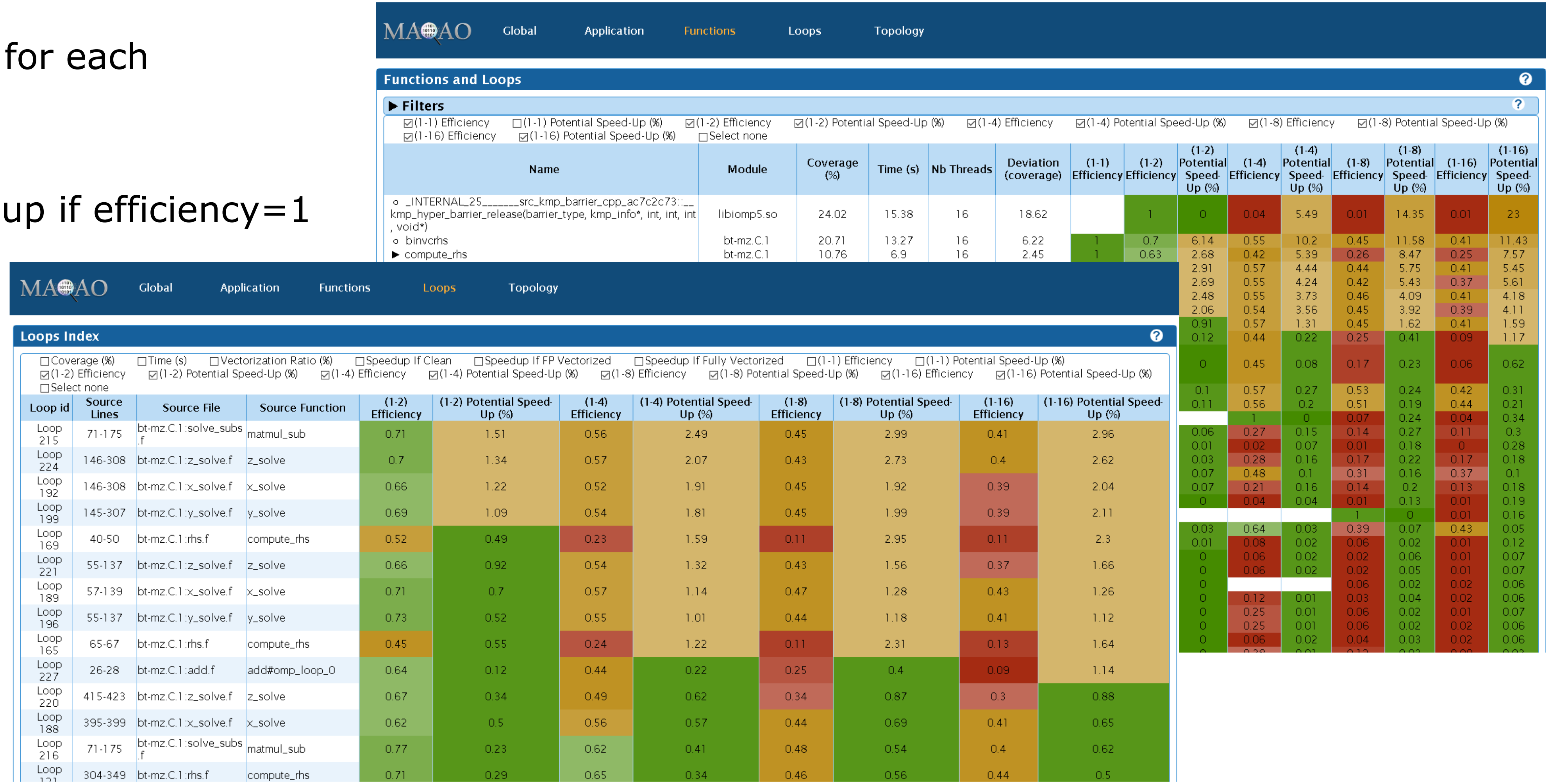




MAQAO ONE View Scalability Reports – Functions and Loops Views

Displays metrics for each function/loop

- Efficiency
- Potential speedup if efficiency=1



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 (%)
☑(1-16) Efficiency ☑(1-16) 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 (%)
○ _INTERNAL_25_______src_kmp_barrier_cpp_ac7c2c73::__kmp_hyper_barrier_release(barrier_type, kmp_info*, int, 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
○ 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
								2.91	0.57	4.44	0.44	5.75	0.41	5.45
								2.69	0.55	4.24	0.42	5.43	0.37	5.61
								2.48	0.55	3.73	0.46	4.09	0.41	4.18
								2.06	0.54	3.56	0.45	3.92	0.39	4.11
								0.91	0.57	1.31	0.45	1.62	0.41	1.59
								0.12	0.44	0.22	0.25	0.41	0.09	1.17
								0	0.45	0.08	0.17	0.23	0.06	0.62
								0.1	0.57	0.27	0.53	0.24	0.42	0.31
								0.11	0.56	0.2	0.51	0.19	0.44	0.21
									1	0	0.07	0.24	0.04	0.34
								0.06	0.27	0.15	0.14	0.27	0.11	0.3
								0.01	0.02	0.07	0.01	0.18	0	0.28
								0.03	0.28	0.16	0.17	0.22	0.17	0.18
								0.07	0.48	0.1	0.31	0.16	0.37	0.1
								0.07	0.21	0.16	0.14	0.2	0.13	0.18
								0	0.04	0.04	0.01	0.13	0.01	0.19
											1	0	0.01	0.16
								0.03	0.64	0.03	0.39	0.07	0.43	0.05
								0.01	0.08	0.02	0.06	0.02	0.01	0.12
								0	0.06	0.02	0.02	0.06	0.01	0.07
								0	0.06	0.02	0.02	0.05	0.01	0.07
								0			0.06	0.02	0.02	0.06
								0	0.12	0.01	0.03	0.04	0.02	0.06
								0	0.25	0.01	0.06	0.02	0.01	0.07
								0	0.25	0.01	0.06	0.02	0.02	0.06
								0	0.06	0.02	0.04	0.03	0.02	0.06

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_subs.f	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_subs.f	matmul_sub	0.77	0.23	0.62	0.41	0.48	0.54	0.4	0.62
Loop	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

Thank you for your attention !

Questions ?

Barcelona Supercomputing Center Centro Nacional de Supercomputación
German Research School for Simulation Sciences
RWTH AACHEN UNIVERSITY
Technische Universität München
Universität Stuttgart
UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
JÜLICH FORSCHUNGSZENTRUM
Lawrence Livermore National Laboratory
TECHNISCHE UNIVERSITÄT DRESDEN
UNIVERSITY OF OREGON
THE UNIVERSITY of TENNESSEE