



# MAQAO

## Performance Analysis and Optimization Tool



---

Emmanuel OSERET, Salah IBNAMAR  
emmanuel.oseret@uvsq.fr, mohammed-salah.ibnamar@uvsq.fr  
Performance Evaluation Team, University of Versailles S-Q-Y  
<http://www.maqao.org>  
VI-HPS 34<sup>th</sup> TW (Online) EPCC, Scotland – 28-30 July 2020

---



## Performance analysis and optimisation

**How much** of an application can be optimized?

- What would the effort/gain ratio be?

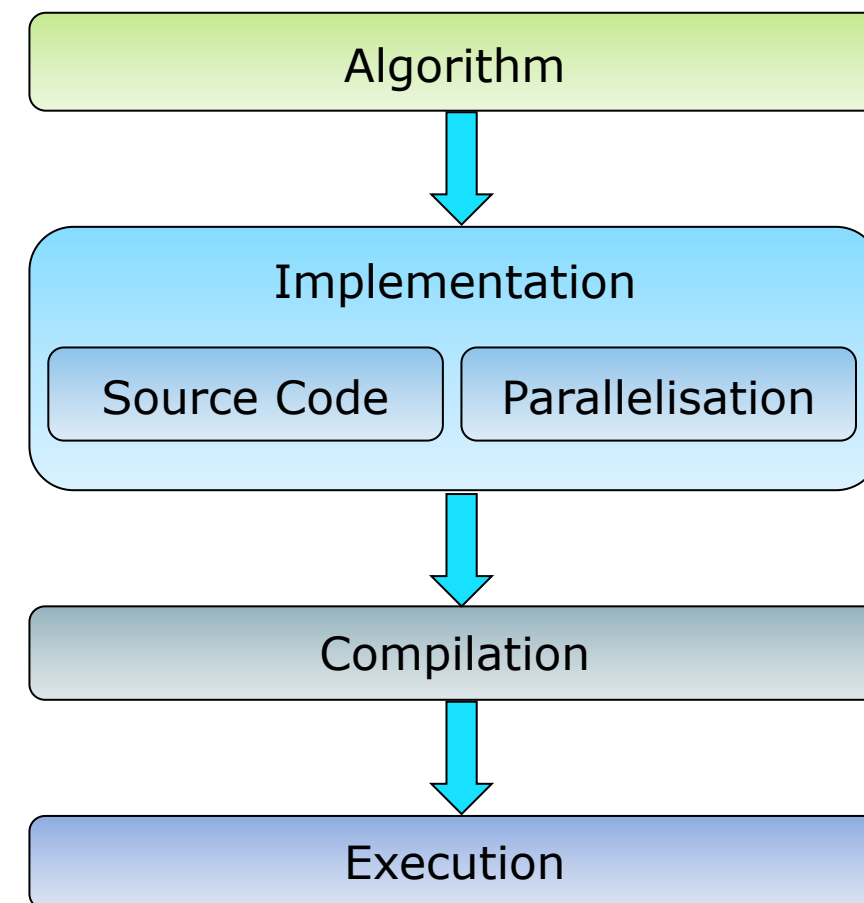
**Where** is the application spending most execution time and resources?

**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?
- What additional information is needed?



## A multifaceted problem

**Pinpointing** the performance bottlenecks

**Identifying** the dominant issues

- Algorithms, implementation, parallelisation, ...

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

- Complex multicore and manycore CPUs
- Complex memory hierarchy

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

- **40%** total time, expected **10%** speedup

- → TOTAL IMPACT: **4%** speedup



- **20%** total time, expected **50%** speedup

- → TOTAL IMPACT: **10%** speedup



→ **Need for dedicated and complementary tools**

## 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

```

Annotations for the code:

- 1) High number of statements (points to the loop body)
- 2) Non-unit stride accesses (points to `ni + nvalue1, nato` and the `gr` calls)
- 3) Indirect accesses (points to `x(nj1)`, `x(nj2)`, `x(nj3)`)
- 4) DIV/SQRT (points to `sqrt`)
- 5) Reductions (points to `Eqc = Eqc + Eq` and `Ephob = Ephob + Ed`)
- 6) Variable number of iterations (points to the loop header)

Source code and associated issues:

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



# MAQAO: Modular Assembly Quality Analyzer and Optimizer

---

## Objectives:

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

## Characteristics:

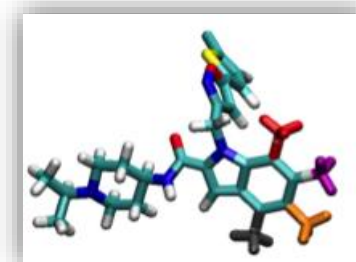
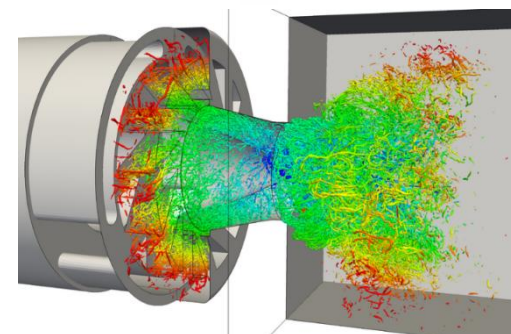
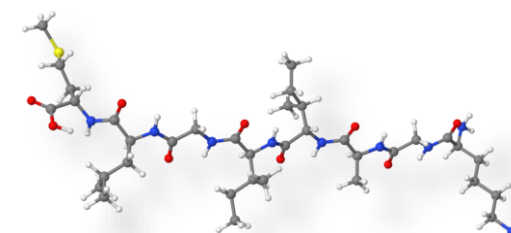
- **Modular tool** offering complementary views
- Support for **Intel x86-64** and **Xeon Phi**
  - ARM on-going development
- LGPL3 Open Source software
- Developed at UVSQ since 2004
- Binary release available as **static executable**



## Success stories

MAQAO was used for optimizing industrial and academic HPC applications:

- QMC=CHEM (IRSAMC)
  - Quantum chemistry
  - Speedup: **> 3x**
    - Optimization: moved invocations of functions with identical parameters out of 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 vectorisation through compiler directives
- AVBP (CERFACS)
  - Computational fluid dynamics
  - Speedup: **1.08x – 1.17x**
    - Replaced divisions with multiplications by reciprocal
    - Complete unrolling of loops with a small number of iterations



## Partnerships

---

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



Provides core capabilities and features for other tools:

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

PeXL ISV also contributes to MAQAO:

- Commercial performance optimization expertise
- Training and software development
- [www.pexl.eu](http://www.pexl.eu)



## MAQAO team and collaborators

---

### MAQAO Team

- William Jalby, Prof.
- Cédric Valensi, Ph.D.
- Emmanuel Oseret, Ph.D.
- Mathieu Tribalat, M.Sc.Eng.
- Salah Ibn Amar, M.Sc.Eng.
- Kévin Camus, Eng.

### Collaborators

- David J. Kuck, Prof
- Andrés S. Charif-Rubial, Ph.D.
- Eric Petit, Ph.D.
- Pablo de Oliveira, Ph.D.
- David C. Wong, Ph.D.
- Othman Bouizi, Ph.D.

### Past Collaborators or Team Members

- Denis Barthou, Prof.
- Jean-Thomas Acquaviva, Ph.D.
- Stéphane Zuckerman, Ph.D.
- Julien Jaeger, Ph.D.
- Souad Koliaï, Ph.D.
- Zakaria Bendifallah, Ph.D.
- Tipp Moseley, Ph.D.
- Jean-Christophe Beyler, Ph.D.
- Hugo Bolloré, M.Sc.Eng.
- Jean-Baptiste Le Reste, M.Sc.Eng.
- Sylvain Henry, Ph.D.
- José Noudouhouenou, Ph.D.
- Aleksandre Vardoshvili, M.Sc.Eng.
- Romain Pillot, Eng
- Youenn Lebras, Ph.D.



## Analysis at binary level

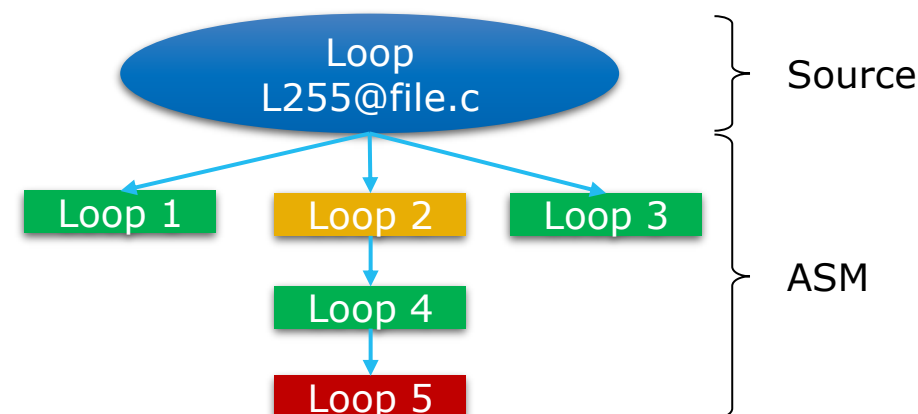
Advantages of binary analysis:

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

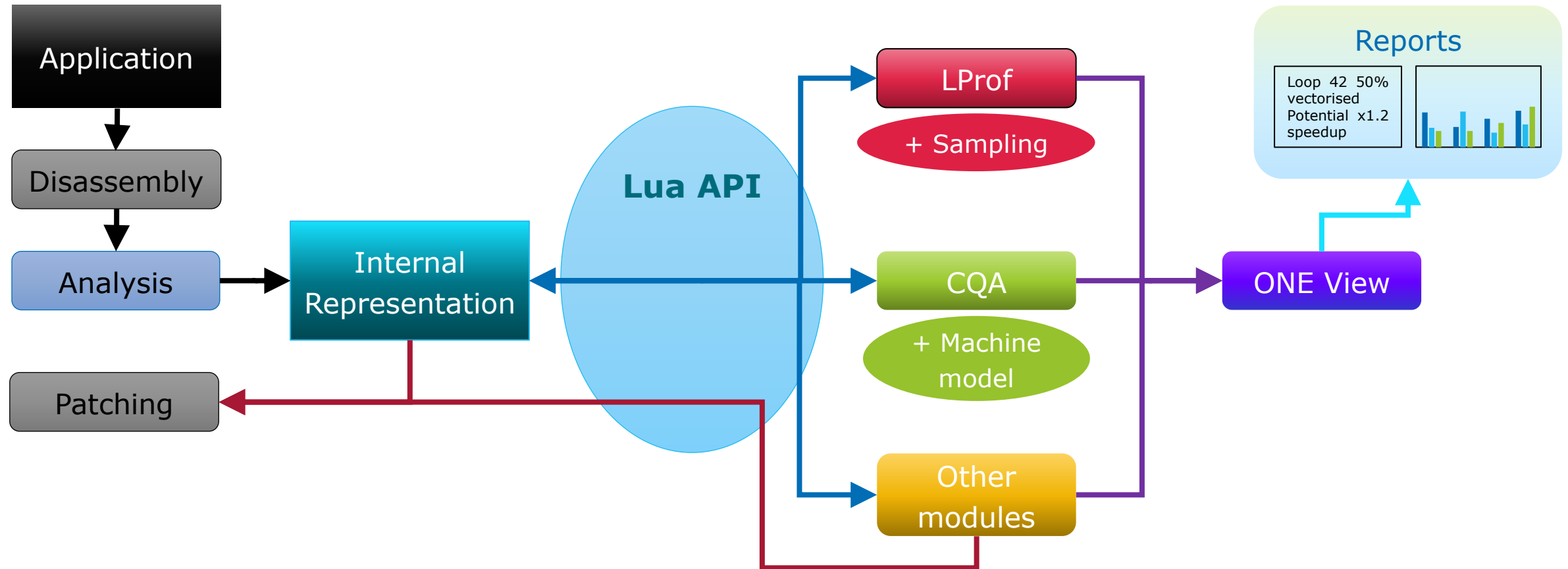
We want to evaluate the “real” executed code: **What You Analyse Is What You Run**

Main steps:

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

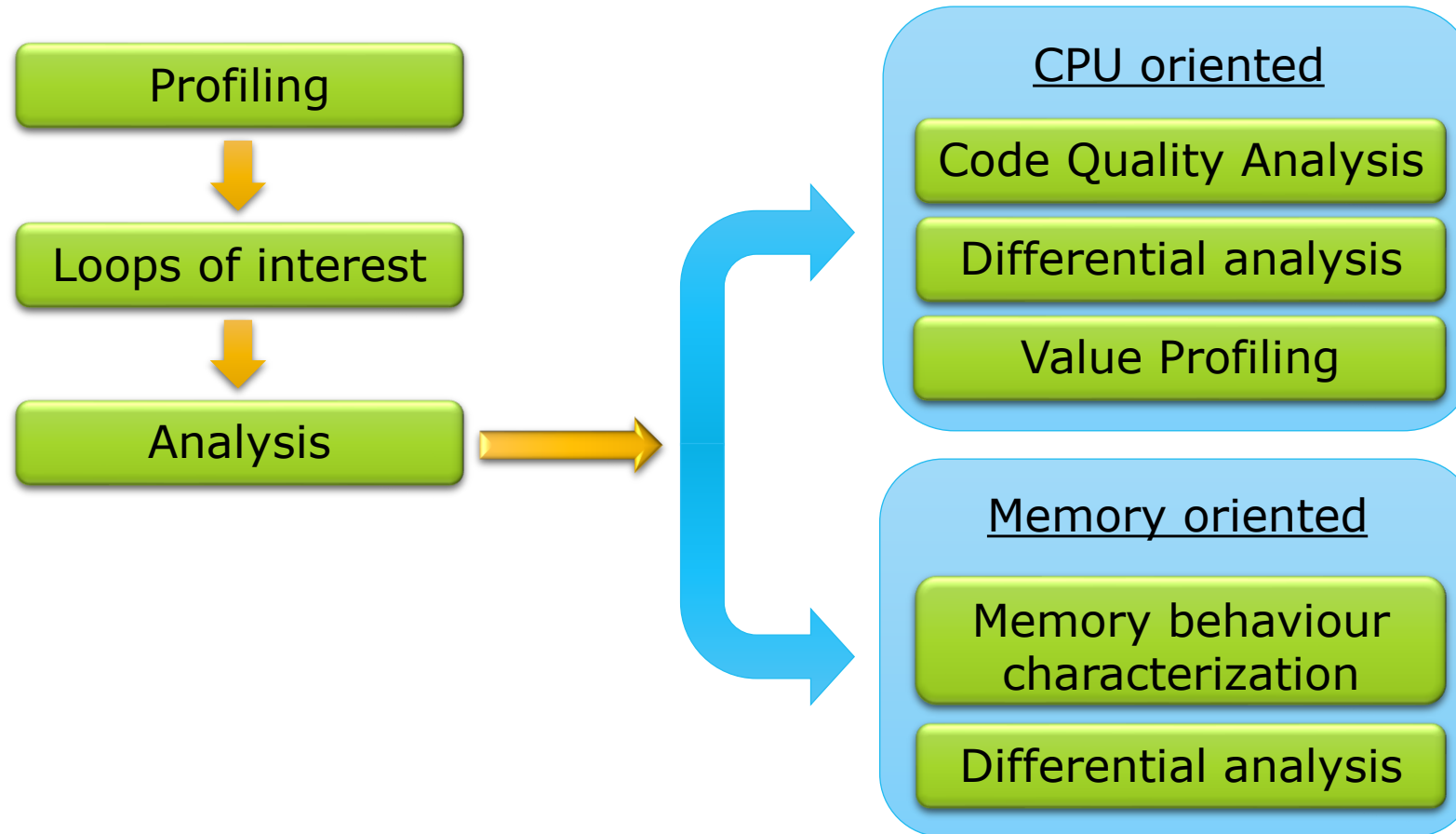


## MAQAO Main structure



## MAQAO Methodology

### Decision tree



# MAQAO LProf: Lightweight Profiler

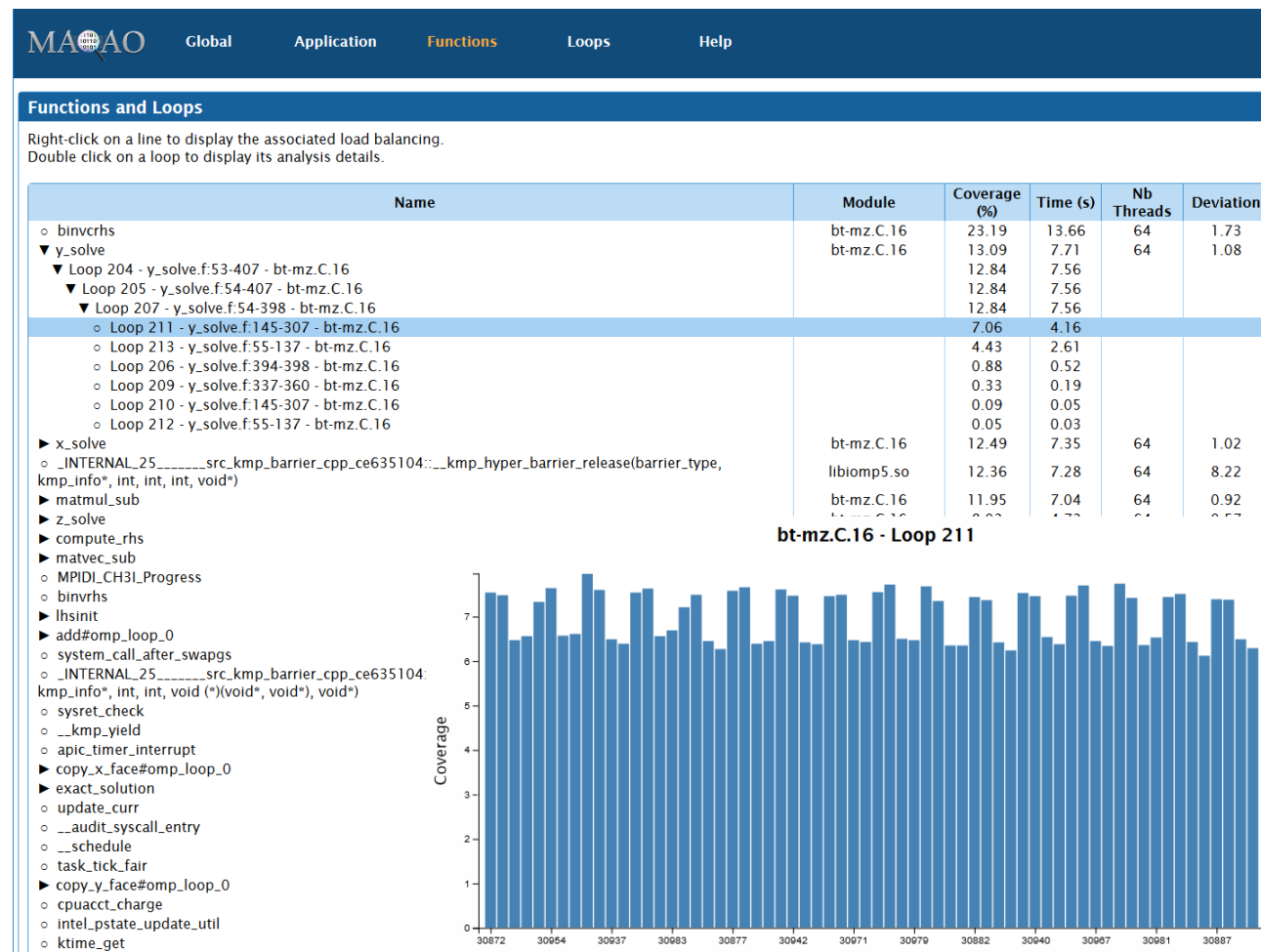
**Goal:** Lightweight localization of application hotspots

Features:

- **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
- Compiler generated code **quality evaluation**
- 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))

gain potential hint expert

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:

- Vectorizing
- 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**

## MAQAO CQA: Guiding the compiler and hints

---

Compiler can be driven using flags and pragmas:

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

Hints for implementation changes:

- Improve data access
  - Memory alignment
  - Loop interchange
  - Changing loop strides
  - Reshaping arrays of structures
- Avoid instructions with high latency

# MAQAO ONE View: Performance View Aggregator

**Automating** the whole analysis process

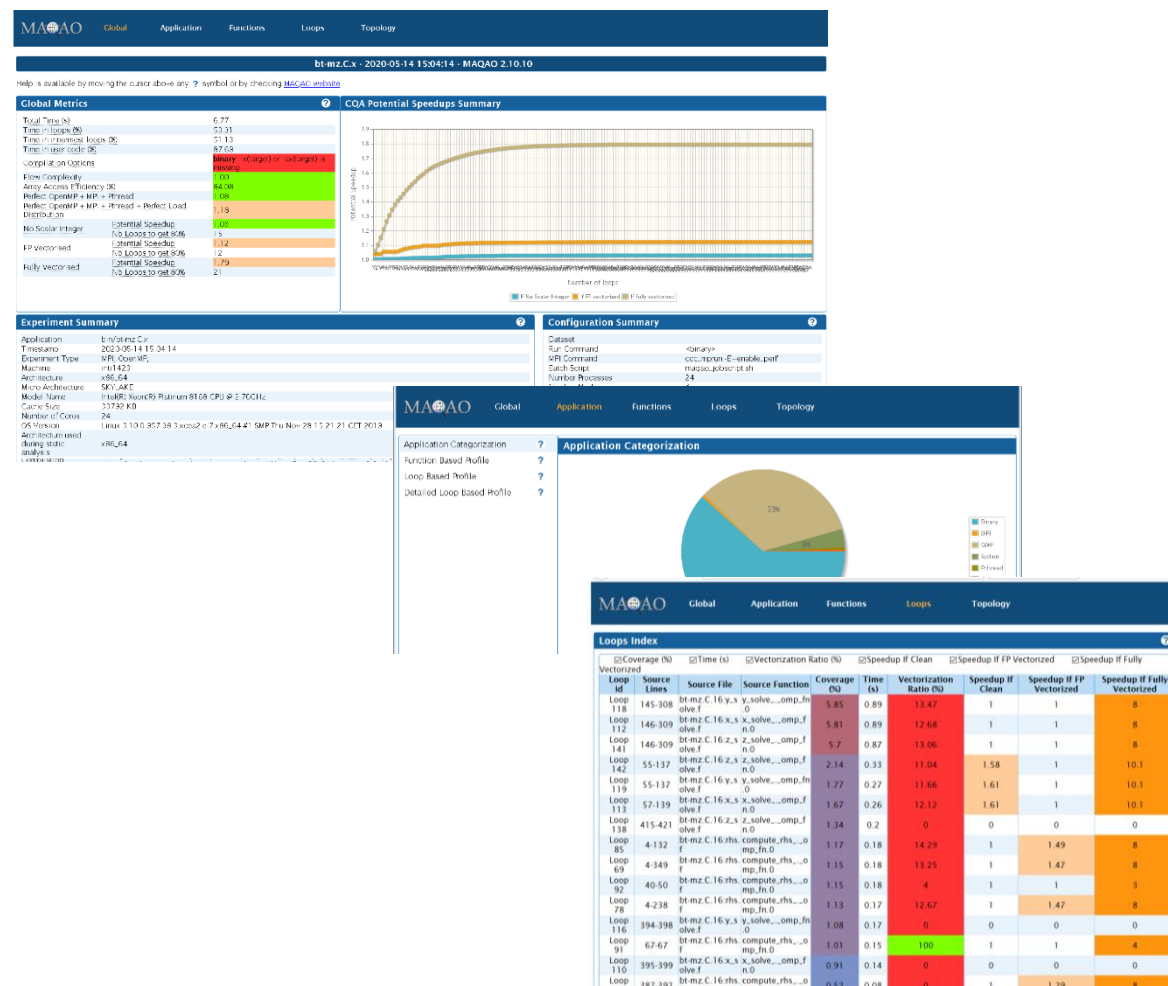
- Invoke multiple MAQAO modules
- Generate **aggregated performance views**
- Available in HTML format

Main steps:

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

Available results:

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



## 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 --binary=bt-mz.C.16 --mpi_command="mpirun -n 16"
```

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

- 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" -- ./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/lprof_npsu -df
```

- CQA

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

Online help is available:

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

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



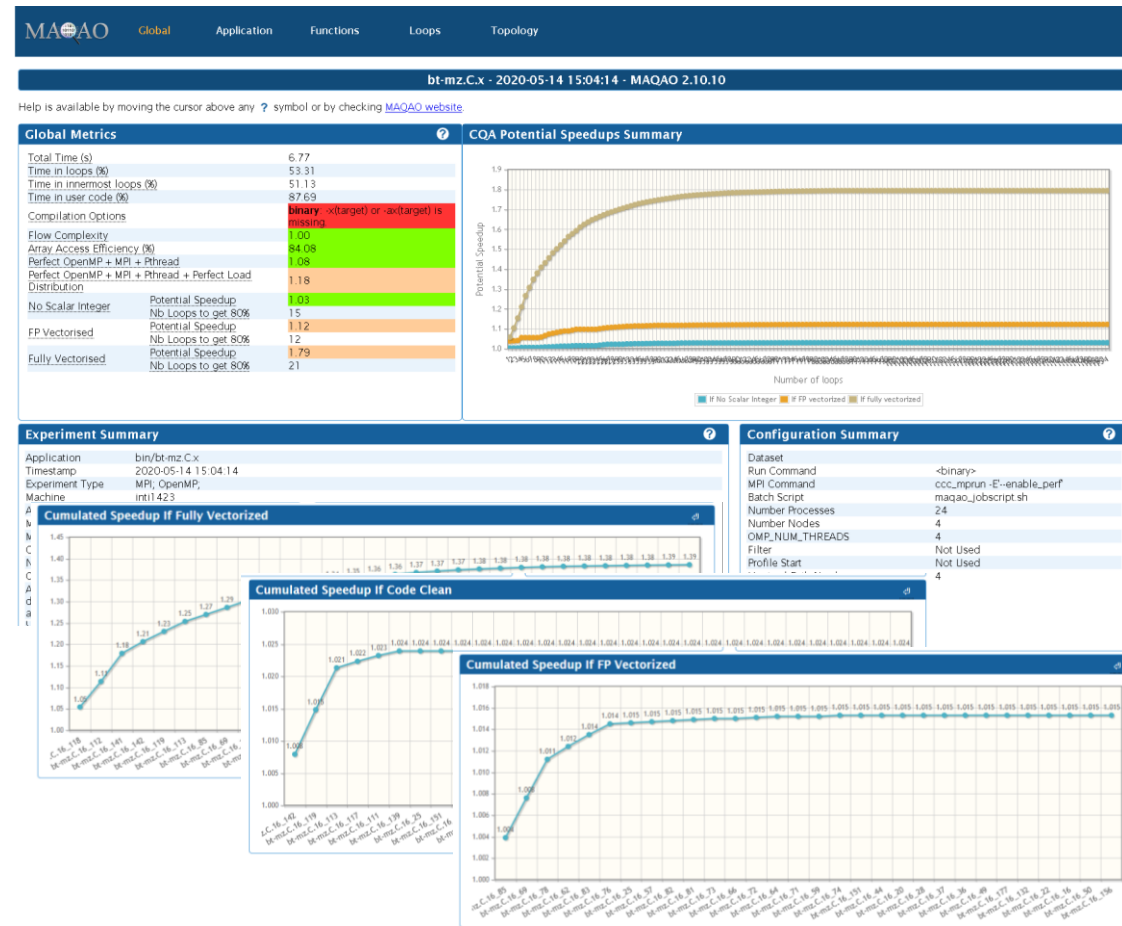
# Global summary

## 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

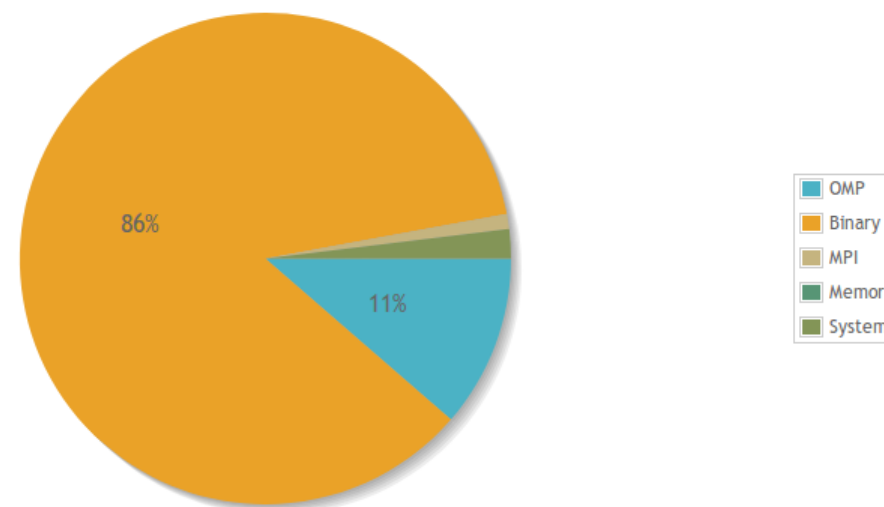


## Time Categorisation

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

Application Categorization



## 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

#### ▼ 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

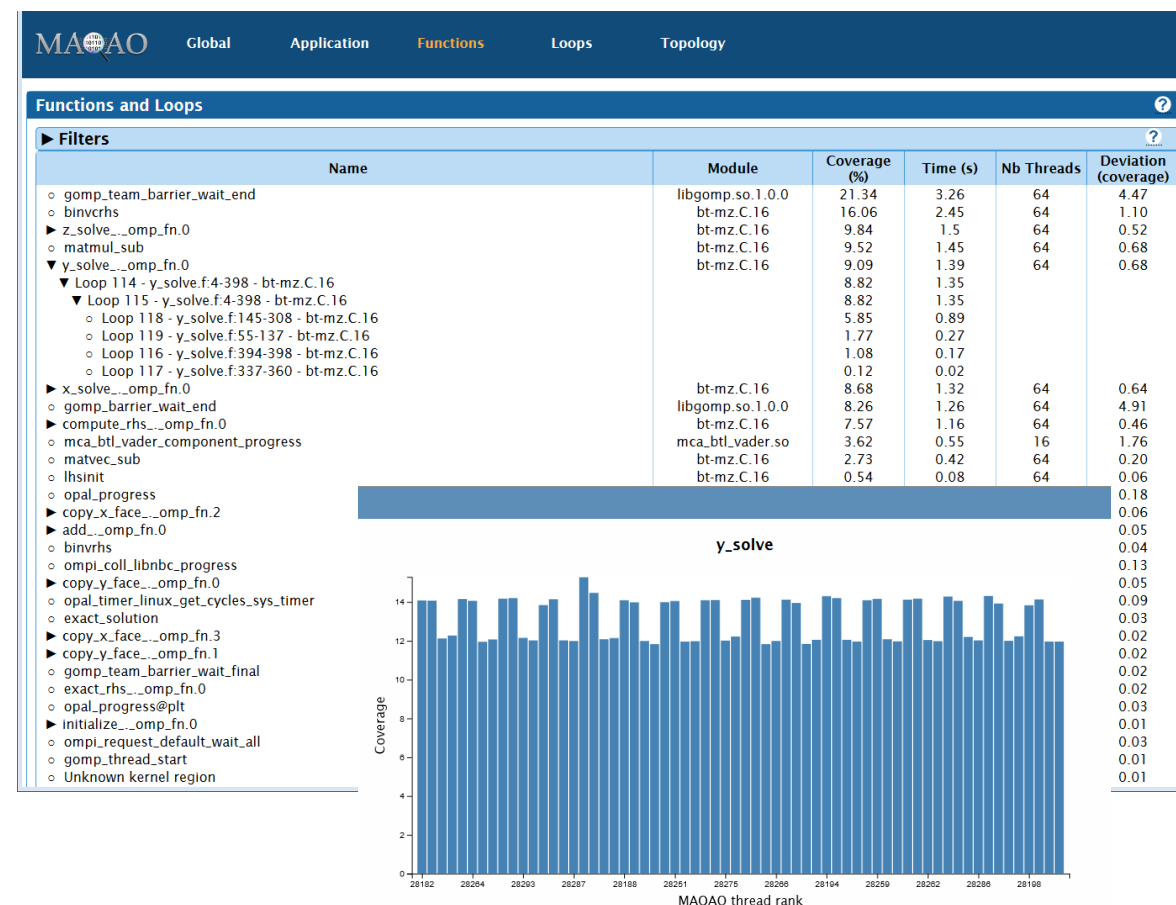
Single

Outermost

Inbetween

Inbetween

Innermost



# 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

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

Select none

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: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

The screenshot displays the MAOAO (Matrix Analysis and Optimization) interface for analyzing a loop. The top navigation bar includes tabs for Global, Application, Functions, Loops, and Topology. The main window shows the source code of a loop (Loop Id: 224, Module: bt-mz.C.16) with line numbers 711 to 1041. The analysis results are displayed on the right, including coverage (4.79%), function (matmul\_sub), and source file/lines (solve\_subs.f:71-175). The report is categorized by probability of gain, with tabs for gain, potential, hint, and expert. The 'gain' tab is selected, showing a 'Code clean check' and a 'Workaround' section. The 'Workaround' section provides detailed information about the loop's performance, including the type of elements and instruction set, matching between the loop and the binary loop, arithmetic intensity, and unroll opportunity. The 'Workaround' section also includes a 'Workaround' subsection with specific recommendations for improving performance.

**Source Code:**

```

711:  cblock(1,1) = cblock(1,1) - mblock(1,1)*mblock(1,1)
712:  > - mblock(1,2)*mblock(2,1)
713:  > - mblock(1,3)*mblock(3,1)
714:  > - mblock(1,4)*mblock(4,1)
715:  > - mblock(1,5)*mblock(5,1)
716:  cblock(2,1) = cblock(2,1) - mblock(2,1)*mblock(1,1)
717:  > - mblock(2,2)*mblock(2,1)
718:  > - mblock(2,3)*mblock(3,1)
719:  > - mblock(2,4)*mblock(4,1)
720:  > - mblock(2,5)*mblock(5,1)
721:  cblock(3,1) = cblock(3,1) - mblock(3,1)*mblock(1,1)
722:  > - mblock(3,2)*mblock(2,1)
723:  > - mblock(3,3)*mblock(3,1)
724:  > - mblock(3,4)*mblock(4,1)
725:  > - mblock(3,5)*mblock(5,1)
726:  cblock(4,1) = cblock(4,1) - mblock(4,1)*mblock(1,1)
727:  > - mblock(4,2)*mblock(2,1)
728:  > - mblock(4,3)*mblock(3,1)
729:  > - mblock(4,4)*mblock(4,1)
730:  > - mblock(4,5)*mblock(5,1)
731:  cblock(5,1) = cblock(5,1) - mblock(5,1)*mblock(1,1)
732:  > - mblock(5,2)*mblock(2,1)
733:  > - mblock(5,3)*mblock(3,1)
734:  > - mblock(5,4)*mblock(4,1)
735:  > - mblock(5,5)*mblock(5,1)
736:  cblock(1,2) = cblock(1,2) - mblock(1,2)*mblock(1,1)
737:  > - mblock(1,3)*mblock(2,1)
738:  > - mblock(1,4)*mblock(3,1)
739:  > - mblock(1,5)*mblock(4,1)
740:  > - mblock(1,6)*mblock(5,1)
741:  cblock(2,2) = cblock(2,2) - mblock(2,2)*mblock(1,1)
742:  > - mblock(2,3)*mblock(2,1)
743:  > - mblock(2,4)*mblock(3,1)
744:  > - mblock(2,5)*mblock(4,1)
745:  > - mblock(2,6)*mblock(5,1)
746:  cblock(3,2) = cblock(3,2) - mblock(3,2)*mblock(1,1)
747:  > - mblock(3,3)*mblock(2,1)
748:  > - mblock(3,4)*mblock(3,1)
749:  > - mblock(3,5)*mblock(4,1)
750:  > - mblock(3,6)*mblock(5,1)
751:  cblock(4,2) = cblock(4,2) - mblock(4,2)*mblock(1,1)
752:  > - mblock(4,3)*mblock(2,1)
753:  > - mblock(4,4)*mblock(3,1)
754:  > - mblock(4,5)*mblock(4,1)
755:  > - mblock(4,6)*mblock(5,1)
756:  cblock(5,2) = cblock(5,2) - mblock(5,2)*mblock(1,1)
757:  > - mblock(5,3)*mblock(2,1)
758:  > - mblock(5,4)*mblock(3,1)
759:  > - mblock(5,5)*mblock(4,1)
760:  > - mblock(5,6)*mblock(5,1)
761:  cblock(1,3) = cblock(1,3) - mblock(1,3)*mblock(1,1)
762:  > - mblock(1,4)*mblock(2,1)
763:  > - mblock(1,5)*mblock(3,1)
764:  > - mblock(1,6)*mblock(4,1)
765:  > - mblock(1,7)*mblock(5,1)
766:  cblock(2,3) = cblock(2,3) - mblock(2,3)*mblock(1,1)
767:  > - mblock(2,4)*mblock(2,1)
768:  > - mblock(2,5)*mblock(3,1)
769:  > - mblock(2,6)*mblock(4,1)
770:  > - mblock(2,7)*mblock(5,1)
771:  cblock(3,3) = cblock(3,3) - mblock(3,3)*mblock(1,1)
772:  > - mblock(3,4)*mblock(2,1)
773:  > - mblock(3,5)*mblock(3,1)
774:  > - mblock(3,6)*mblock(4,1)
775:  > - mblock(3,7)*mblock(5,1)
776:  cblock(4,3) = cblock(4,3) - mblock(4,3)*mblock(1,1)
777:  > - mblock(4,4)*mblock(2,1)
778:  > - mblock(4,5)*mblock(3,1)
779:  > - mblock(4,6)*mblock(4,1)
780:  > - mblock(4,7)*mblock(5,1)
781:  cblock(5,3) = cblock(5,3) - mblock(5,3)*mblock(1,1)
782:  > - mblock(5,4)*mblock(2,1)
783:  > - mblock(5,5)*mblock(3,1)
784:  > - mblock(5,6)*mblock(4,1)
785:  > - mblock(5,7)*mblock(5,1)
786:  cblock(1,4) = cblock(1,4) - mblock(1,4)*mblock(1,1)
787:  > - mblock(1,5)*mblock(2,1)
788:  > - mblock(1,6)*mblock(3,1)
789:  > - mblock(1,7)*mblock(4,1)
790:  > - mblock(1,8)*mblock(5,1)
791:  cblock(2,4) = cblock(2,4) - mblock(2,4)*mblock(1,1)
792:  > - mblock(2,5)*mblock(2,1)
793:  > - mblock(2,6)*mblock(3,1)
794:  > - mblock(2,7)*mblock(4,1)
795:  > - mblock(2,8)*mblock(5,1)
796:  cblock(3,4) = cblock(3,4) - mblock(3,4)*mblock(1,1)
797:  > - mblock(3,5)*mblock(2,1)
798:  > - mblock(3,6)*mblock(3,1)
799:  > - mblock(3,7)*mblock(4,1)
800:  > - mblock(3,8)*mblock(5,1)
801:  cblock(4,4) = cblock(4,4) - mblock(4,4)*mblock(1,1)
802:  > - mblock(4,5)*mblock(2,1)
803:  > - mblock(4,6)*mblock(3,1)
804:  > - mblock(4,7)*mblock(4,1)
805:  > - mblock(4,8)*mblock(5,1)
806:  cblock(5,4) = cblock(5,4) - mblock(5,4)*mblock(1,1)
807:  > - mblock(5,5)*mblock(2,1)
808:  > - mblock(5,6)*mblock(3,1)
809:  > - mblock(5,7)*mblock(4,1)
810:  > - mblock(5,8)*mblock(5,1)
811:  cblock(1,5) = cblock(1,5) - mblock(1,5)*mblock(1,1)
812:  > - mblock(1,6)*mblock(2,1)
813:  > - mblock(1,7)*mblock(3,1)
814:  > - mblock(1,8)*mblock(4,1)
815:  > - mblock(1,9)*mblock(5,1)
816:  cblock(2,5) = cblock(2,5) - mblock(2,5)*mblock(1,1)
817:  > - mblock(2,6)*mblock(2,1)
818:  > - mblock(2,7)*mblock(3,1)
819:  > - mblock(2,8)*mblock(4,1)
820:  > - mblock(2,9)*mblock(5,1)
821:  cblock(3,5) = cblock(3,5) - mblock(3,5)*mblock(1,1)
822:  > - mblock(3,6)*mblock(2,1)
823:  > - mblock(3,7)*mblock(3,1)
824:  > - mblock(3,8)*mblock(4,1)
825:  > - mblock(3,9)*mblock(5,1)
826:  cblock(4,5) = cblock(4,5) - mblock(4,5)*mblock(1,1)
827:  > - mblock(4,6)*mblock(2,1)
828:  > - mblock(4,7)*mblock(3,1)
829:  > - mblock(4,8)*mblock(4,1)
830:  > - mblock(4,9)*mblock(5,1)
831:  cblock(5,5) = cblock(5,5) - mblock(5,5)*mblock(1,1)
832:  > - mblock(5,6)*mblock(2,1)
833:  > - mblock(5,7)*mblock(3,1)
834:  > - mblock(5,8)*mblock(4,1)
835:  > - mblock(5,9)*mblock(5,1)
836:  cblock(1,6) = cblock(1,6) - mblock(1,6)*mblock(1,1)
837:  > - mblock(1,7)*mblock(2,1)
838:  > - mblock(1,8)*mblock(3,1)
839:  > - mblock(1,9)*mblock(4,1)
840:  > - mblock(1,10)*mblock(5,1)
841:  cblock(2,6) = cblock(2,6) - mblock(2,6)*mblock(1,1)
842:  > - mblock(2,7)*mblock(2,1)
843:  > - mblock(2,8)*mblock(3,1)
844:  > - mblock(2,9)*mblock(4,1)
845:  > - mblock(2,10)*mblock(5,1)
846:  cblock(3,6) = cblock(3,6) - mblock(3,6)*mblock(1,1)
847:  > - mblock(3,7)*mblock(2,1)
848:  > - mblock(3,8)*mblock(3,1)
849:  > - mblock(3,9)*mblock(4,1)
850:  > - mblock(3,10)*mblock(5,1)
851:  cblock(4,6) = cblock(4,6) - mblock(4,6)*mblock(1,1)
852:  > - mblock(4,7)*mblock(2,1)
853:  > - mblock(4,8)*mblock(3,1)
854:  > - mblock(4,9)*mblock(4,1)
855:  > - mblock(4,10)*mblock(5,1)
856:  cblock(5,6) = cblock(5,6) - mblock(5,6)*mblock(1,1)
857:  > - mblock(5,7)*mblock(2,1)
858:  > - mblock(5,8)*mblock(3,1)
859:  > - mblock(5,9)*mblock(4,1)
860:  > - mblock(5,10)*mblock(5,1)
861:  cblock(1,7) = cblock(1,7) - mblock(1,7)*mblock(1,1)
862:  > - mblock(1,8)*mblock(2,1)
863:  > - mblock(1,9)*mblock(3,1)
864:  > - mblock(1,10)*mblock(4,1)
865:  > - mblock(1,11)*mblock(5,1)
866:  cblock(2,7) = cblock(2,7) - mblock(2,7)*mblock(1,1)
867:  > - mblock(2,8)*mblock(2,1)
868:  > - mblock(2,9)*mblock(3,1)
869:  > - mblock(2,10)*mblock(4,1)
870:  > - mblock(2,11)*mblock(5,1)
871:  cblock(3,7) = cblock(3,7) - mblock(3,7)*mblock(1,1)
872:  > - mblock(3,8)*mblock(2,1)
873:  > - mblock(3,9)*mblock(3,1)
874:  > - mblock(3,10)*mblock(4,1)
875:  > - mblock(3,11)*mblock(5,1)
876:  cblock(4,7) = cblock(4,7) - mblock(4,7)*mblock(1,1)
877:  > - mblock(4,8)*mblock(2,1)
878:  > - mblock(4,9)*mblock(3,1)
879:  > - mblock(4,10)*mblock(4,1)
880:  > - mblock(4,11)*mblock(5,1)
881:  cblock(5,7) = cblock(5,7) - mblock(5,7)*mblock(1,1)
882:  > - mblock(5,8)*mblock(2,1)
883:  > - mblock(5,9)*mblock(3,1)
884:  > - mblock(5,10)*mblock(4,1)
885:  > - mblock(5,11)*mblock(5,1)
886:  cblock(1,8) = cblock(1,8) - mblock(1,8)*mblock(1,1)
887:  > - mblock(1,9)*mblock(2,1)
888:  > - mblock(1,10)*mblock(3,1)
889:  > - mblock(1,11)*mblock(4,1)
890:  > - mblock(1,12)*mblock(5,1)
891:  cblock(2,8) = cblock(2,8) - mblock(2,8)*mblock(1,1)
892:  > - mblock(2,9)*mblock(2,1)
893:  > - mblock(2,10)*mblock(3,1)
894:  > - mblock(2,11)*mblock(4,1)
895:  > - mblock(2,12)*mblock(5,1)
896:  cblock(3,8) = cblock(3,8) - mblock(3,8)*mblock(1,1)
897:  > - mblock(3,9)*mblock(2,1)
898:  > - mblock(3,10)*mblock(3,1)
899:  > - mblock(3,11)*mblock(4,1)
900:  > - mblock(3,12)*mblock(5,1)
901:  cblock(4,8) = cblock(4,8) - mblock(4,8)*mblock(1,1)
902:  > - mblock(4,9)*mblock(2,1)
903:  > - mblock(4,10)*mblock(3,1)
904:  > - mblock(4,11)*mblock(4,1)
905:  > - mblock(4,12)*mblock(5,1)
906:  cblock(5,8) = cblock(5,8) - mblock(5,8)*mblock(1,1)
907:  > - mblock(5,9)*mblock(2,1)
908:  > - mblock(5,10)*mblock(3,1)
909:  > - mblock(5,11)*mblock(4,1)
910:  > - mblock(5,12)*mblock(5,1)
911:  cblock(1,9) = cblock(1,9) - mblock(1,9)*mblock(1,1)
912:  > - mblock(1,10)*mblock(2,1)
913:  > - mblock(1,11)*mblock(3,1)
914:  > - mblock(1,12)*mblock(4,1)
915:  > - mblock(1,13)*mblock(5,1)
916:  cblock(2,9) = cblock(2,9) - mblock(2,9)*mblock(1,1)
917:  > - mblock(2,10)*mblock(2,1)
918:  > - mblock(2,11)*mblock(3,1)
919:  > - mblock(2,12)*mblock(4,1)
920:  > - mblock(2,13)*mblock(5,1)
921:  cblock(3,9) = cblock(3,9) - mblock(3,9)*mblock(1,1)
922:  > - mblock(3,10)*mblock(2,1)
923:  > - mblock(3,11)*mblock(3,1)
924:  > - mblock(3,12)*mblock(4,1)
925:  > - mblock(3,13)*mblock(5,1)
926:  cblock(4,9) = cblock(4,9) - mblock(4,9)*mblock(1,1)
927:  > - mblock(4,10)*mblock(2,1)
928:  > - mblock(4,11)*mblock(3,1)
929:  > - mblock(4,12)*mblock(4,1)
930:  > - mblock(4,13)*mblock(5,1)
931:  cblock(5,9) = cblock(5,9) - mblock(5,9)*mblock(1,1)
932:  > - mblock(5,10)*mblock(2,1)
933:  > - mblock(5,11)*mblock(3,1)
934:  > - mblock(5,12)*mblock(4,1)
935:  > - mblock(5,13)*mblock(5,1)
936:  cblock(1,10) = cblock(1,10) - mblock(1,10)*mblock(1,1)
937:  > - mblock(1,11)*mblock(2,1)
938:  > - mblock(1,12)*mblock(3,1)
939:  > - mblock(1,13)*mblock(4,1)
940:  > - mblock(1,14)*mblock(5,1)
941:  cblock(2,10) = cblock(2,10) - mblock(2,10)*mblock(1,1)
942:  > - mblock(2,11)*mblock(2,1)
943:  > - mblock(2,12)*mblock(3,1)
944:  > - mblock(2,13)*mblock(4,1)
945:  > - mblock(2,14)*mblock(5,1)
946:  cblock(3,10) = cblock(3,10) - mblock(3,10)*mblock(1,1)
947:  > - mblock(3,11)*mblock(2,1)
948:  > - mblock(3,12)*mblock(3,1)
949:  > - mblock(3,13)*mblock(4,1)
950:  > - mblock(3,14)*mblock(5,1)
951:  cblock(4,10) = cblock(4,10) - mblock(4,10)*mblock(1,1)
952:  > - mblock(4,11)*mblock(2,1)
953:  > - mblock(4,12)*mblock(3,1)
954:  > - mblock(4,13)*mblock(4,1)
955:  > - mblock(4,14)*mblock(5,1)
956:  cblock(5,10) = cblock(5,10) - mblock(5,10)*mblock(1,1)
957:  > - mblock(5,11)*mblock(2,1)
958:  > - mblock(5,12)*mblock(3,1)
959:  > - mblock(5,13)*mblock(4,1)
960:  > - mblock(5,14)*mblock(5,1)
961:  cblock(1,11) = cblock(1,11) - mblock(1,11)*mblock(1,1)
962:  > - mblock(1,12)*mblock(2,1)
963:  > - mblock(1,13)*mblock(3,1)
964:  > - mblock(1,14)*mblock(4,1)
965:  > - mblock(1,15)*mblock(5,1)
966:  cblock(2,11) = cblock(2,11) - mblock(2,11)*mblock(1,1)
967:  > - mblock(2,12)*mblock(2,1)
968:  > - mblock(2,13)*mblock(3,1)
969:  > - mblock(2,14)*mblock(4,1)
970:  > - mblock(2,15)*mblock(5,1)
971:  cblock(3,11) = cblock(3,11) - mblock(3,11)*mblock(1,1)
972:  > - mblock(3,12)*mblock(2,1)
973:  > - mblock(3,13)*mblock(3,1)
974:  > - mblock(3,14)*mblock(4,1)
975:  > - mblock(3,15)*mblock(5,1)
976:  cblock(4,11) = cblock(4,11) - mblock(4,11)*mblock(1,1)
977:  > - mblock(4,12)*mblock(2,1)
978:  > - mblock(4,13)*mblock(3,1)
979:  > - mblock(4,14)*mblock(4,1)
980:  > - mblock(4,15)*mblock(5,1)
981:  cblock(5,11) = cblock(5,11) - mblock(5,11)*mblock(1,1)
982:  > - mblock(5,12)*mblock(2,1)
983:  > - mblock(5,13)*mblock(3,1)
984:  > - mblock(5,14)*mblock(4,1)
985:  > - mblock(5,15)*mblock(5,1)
986:  cblock(1,12) = cblock(1,12) - mblock(1,12)*mblock(1,1)
987:  > - mblock(1,13)*mblock(2,1)
988:  > - mblock(1,14)*mblock(3,1)
989:  > - mblock(1,15)*mblock(4,1)
990:  > - mblock(1,16)*mblock(5,1)
991:  cblock(2,12) = cblock(2,12) - mblock(2,12)*mblock(1,1)
992:  > - mblock(2,13)*mblock(2,1)
993:  > - mblock(2,14)*mblock(3,1)
994:  > - mblock(2,15)*mblock(4,1)
995:  > - mblock(2,16)*mblock(5,1)
996:  cblock(3,12) = cblock(3,12) - mblock(3,12)*mblock(1,1)
997:  > - mblock(3,13)*mblock(2,1)
998:  > - mblock(3,14)*mblock(3,1)
999:  > - mblock(3,15)*mblock(4,1)
1000: > - mblock(3,16)*mblock(5,1)
1001: cblock(4,12) = cblock(4,12) - mblock(4,12)*mblock(1,1)
1002: > - mblock(4,13)*mblock(2,1)
1003: > - mblock(4,14)*mblock(3,1)
1004: > - mblock(4,15)*mblock(4,1)
1005: > - mblock(4,16)*mblock(5,1)
1006: cblock(5,12) = cblock(5,12) - mblock(5,12)*mblock(1,1)
1007: > - mblock(5,13)*mblock(2,1)
1008: > - mblock(5,14)*mblock(3,1)
1009: > - mblock(5,15)*mblock(4,1)
1010: > - mblock(5,16)*mblock(5,1)
1011: cblock(1,13) = cblock(1,13) - mblock(1,13)*mblock(1,1)
1012: > - mblock(1,14)*mblock(2,1)
1013: > - mblock(1,15)*mblock(3,1)
1014: > - mblock(1,16)*mblock(4,1)
1015: > - mblock(1,17)*mblock(5,1)
1016: cblock(2,13) = cblock(2,13) - mblock(2,13)*mblock(1,1)
1017: > - mblock(2,14)*mblock(2,1)
1018: > - mblock(2,15)*mblock(3,1)
1019: > - mblock(2,16)*mblock(4,1)
1020: > - mblock(2,17)*mblock(5,1)
1021: cblock(3,13) = cblock(3,13) - mblock(3,13)*mblock(1,1)
1022: > - mblock(3,14)*mblock(2,1)
1023: > - mblock(3,15)*mblock(3,1)
1024: > - mblock(3,16)*mblock(4,1)
1025: > - mblock(3,17)*mblock(5,1)
1026: cblock(4,13) = cblock(4,13) - mblock(4,13)*mblock(1,1)
1027: > - mblock(4,14)*mblock(2,1)
1028: > - mblock(4,15)*mblock(3,1)
1029: > - mblock(4,16)*mblock(4,1)
1030: > - mblock(4,17)*mblock(5,1)
1031: cblock(5,13) = cblock(5,13) - mblock(5,13)*mblock(1,1)
1032: > - mblock(5,14)*mblock(2,1)
1033: > - mblock(5,15)*mblock(3,1)
1034: > - mblock(5,16)*mblock(4,1)
1035: > - mblock(5,17)*mblock(5,1)
1036: cblock(1,14) = cblock(1,14) - mblock(1,14)*mblock(1,1)
1037: > - mblock(1,15)*mblock(2,1)
1038: > - mblock(1,16)*mblock(3,1)
1039: > - mblock(1,17)*mblock(4,1)
1040: > - mblock(1,18)*mblock(5,1)
1041: cblock(2,14) = cblock(2,14) - mblock(2,14)*mblock(1,1)
1042: > - mblock(2,15)*mblock(2,1)
1043: > - mblock(2,16)*mblock(3,1)
1044: > - mblock(2,17)*mblock(4,1)
1045: > - mblock(2,18)*mblock(5,1)
1046: cblock(3,14) = cblock(3,14) - mblock(3,14)*mblock(1,1)
1047: > - mblock(3,15)*mblock(2,1)
1048: > - mblock(3,16)*mblock(3,1)
1049: > - mblock(3,17)*mblock(4,1)
1050: > - mblock(3,18)*mblock(5,1)
1051: cblock(4,14) = cblock(4,14) - mblock(4,14)*mblock(1,1)
1052: > - mblock(4,15)*mblock(2,1)
1053: > - mblock(4,16)*mblock(3,1)
1054: > - mblock(4,17)*mblock(4,1)
1055: > - mblock(4,18)*mblock(5,1)
1056: cblock(5,14) = cblock(5,14) - mblock(5,14)*mblock(1,1)
1057: > - mblock(5,15)*mblock(2,1)
1058: > - mblock(5,16)*mblock(3,1)
1059: > - mblock(5,17)*mblock(4,1)
1060: > - mblock(5,18)*mblock(5,1)
1061: cblock(1,15) = cblock(1,15) - mblock(1,15)*mblock(1,1)
1062: > - mblock(1,16)*mblock(2,1)
1063: > - mblock(1,17)*mblock(3,1)
1064: > - mblock(1,18)*mblock(4,1)
1065: > - mblock(1,19)*mblock(5,1)
1066: cblock(2,15) = cblock(2,15) - mblock(2,15)*mblock(1,1)
1067: > - mblock(2,16)*mblock(2,1)
1068: > - mblock(2,17)*mblock(3,1)
1069: > - mblock(2,18)*mblock(4,1)
1070: > - mblock(2,19)*mblock(5,1)
1071: cblock(3,15) = cblock(3,15) - mblock(3,15)*mblock(1,1)
1072: > - mblock(3,16)*mblock(2,1)
1073: > - mblock(3,17)*mblock(3,1)
1074: > - mblock(3,18)*mblock(4,1)
1075: > - mblock(3,19)*mblock(5,1)
1076: cblock(4,15) = cblock(4,15) - mblock(4,15)*mblock(1,1)
1077: > - mblock(4,16)*mblock(2,1)
1078: > - mblock(4,17)*mblock(3,1)
1079: > - mblock(4,18)*mblock(4,1)
1080: > - mblock(4,19)*mblock(5,1)
1081: cblock(5,15) = cblock(5,15) - mblock(5,15)*mblock(1,1)
1082: > - mblock(5,16)*mblock(2,1)
1083: > - mblock(5,17)*mblock(3,1)
1084: > - mblock(5,18)*mblock(4,1)
1085: > - mblock(5,19)*mblock(5,1)
1086: cblock(1,16) = cblock(1,16) - mblock(1,16)*mblock(1,1)
1087: > - mblock(1,17)*mblock(2,1)
1088: > - mblock(1,18)*mblock(3,1)
1089: > - mblock(1,19)*mblock(4,1)
1090: > - mblock(1,20)*mblock(5,1)
1091: cblock(2,16) = cblock(2,16) - mblock(2,16)*mblock(1,1)
1092: > - mblock(2,17)*mblock(2,1)
1093: > - mblock(2,18)*mblock(3,1)
1094: > - mblock(2,19)*mblock(4,1)
1095: > - mblock(2,20)*mblock(5,1)
1096: cblock(3,16) = cblock(3,16) - mblock(3,16)*mblock(1,1)
1097: > - mblock(3,17)*mblock(2,1)
1098: > - mblock(3,18)*mblock(3,1)
1099: > - mblock(3,19)*mblock(4,1)
1100: > - mblock(3,20)*mblock(5,1)
1101: cblock(4,16) = cblock(4,16) - mblock(4,16)*mblock(1,1)
1102: > - mblock(4,17)*mblock(2,1)
1103: > - mblock(4,18)*mblock(3,1)
1104: > - mblock(4,19)*mblock(4,1)
1105: > - mblock(4,20)*mblock(5,1)
1106: cblock(5,16) = cblock(5,16) - mblock(5,16)*mblock(1,1)
1107: > - mblock(5,17)*mblock(2,1)
1108: > - mblock(5,18)*mblock(3,1)
1109: > - mblock(5,19)*mblock(4,1)
1110: > - mblock(5,20)*mblock(5,1)
1111: cblock(1,17) = cblock(1,17) - mblock(1,17)*mblock(1,1)
1112: > - mblock(1,18)*mblock(2,1)
1113: > - mblock(1,19)*mblock(3,1)
1114: > - mblock(1,20)*mblock(4,1)
1115: > - mblock(1,21)*mblock(5,1)
1116: cblock(2,17) = cblock(2,17) - mblock(2,17)*mblock(1,1)
1117: > - mblock(2,18)*mblock(2,1)
1118: > - mblock(2,19)*mblock(3,1)
1119: > - mblock(2,20)*mblock(4,1)
1120: > - mblock(2,21)*mblock(5,1)
1121: cblock(3,17) = cblock(3,17) - mblock(3,17)*mblock(1,1)
1122: > - mblock(3,18)*mblock(2,1)
1123: > - mblock(3,19)*mblock(3,1)
1124: > - mblock(3,20)*mblock(4,1)
1125: > - mblock(3,21)*mblock(5,1)
1126: cblock(4,17) = cblock(4,17) - mblock(4,17)*mblock(1,1)
1127: > - mblock(4,18)*mblock(2,1)
1128: > - mblock(4,19)*mblock(3,1)
1129: > - mblock(4,20)*mblock(4,1)
1130: > - mblock(4,21)*mblock(5,1)
1131: cblock(5,17) = cblock(5,17) - mblock(5,17)*mblock(1,1)
1132: > - mblock(5,18)*mblock(2,1)
1133: > - mblock(5,19)*mblock(3,1)
1134: > - mblock(5,20)*mblock(4,1)
1135: > - mblock(5,21)*mblock(5,1)
1136: cblock(1,18) = cblock(1,18) - mblock(1,18)*mblock(1,1)
1137: > - mblock(1,19)*mblock(2,1)
1138: > - mblock(1,20)*mblock(3,1)
1139: > - mblock(1,21)*mblock(4,1)
1140: > - mblock(1,22)*mblock(5,1)
1141: cblock(2,18) = cblock(2,18) - mblock(2,18)*mblock(1,1)
1142: > - mblock(2,19)*mblock(2,1)
1143: > - mblock(2,20)*mblock(3,1)
1144: > - mblock(2,2
```



## 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

**Gain** **Potential gain** **Hints** **Experts only**

**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	0	2	0.50
INC %RDI	1	0	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	0	40-42	12-32
MOVAPS %XMM5,%XMM15	1	0.50	0.50	0	0	0	0	0	0	2	0.50
MULSD %XMM5,%XMM15	1	0.50	0.50	0	0	0	0	0	0	6	0.50
MOVSD %XMM5,0x12890(%R14)	1	0	0	0.50	0.50	0	0	0	1	2	1
MULSD %XMM15,%XMM5	1	0.50	0.50	0	0	0	0	0	0	6	0.50

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

Assembly Code

Hide groups analysis

Source: solve\_subs.f:71-175 Coverage: 4.79%

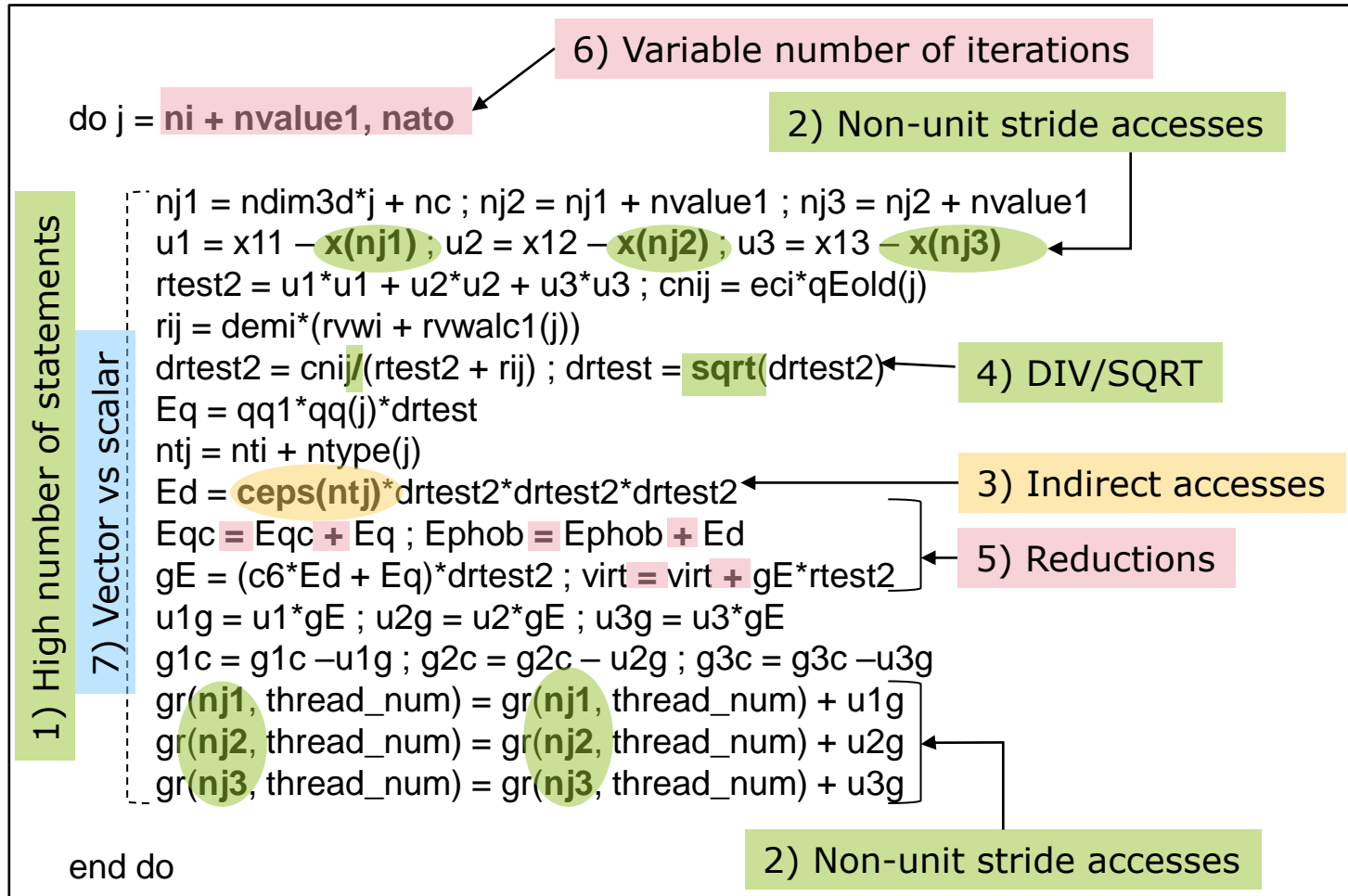
CQA Advanced

Path 1 / 1

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

## 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

# 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 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(j,i)` (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)

**Gain Potential gain Hints Experts only**

### 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**

## 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

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

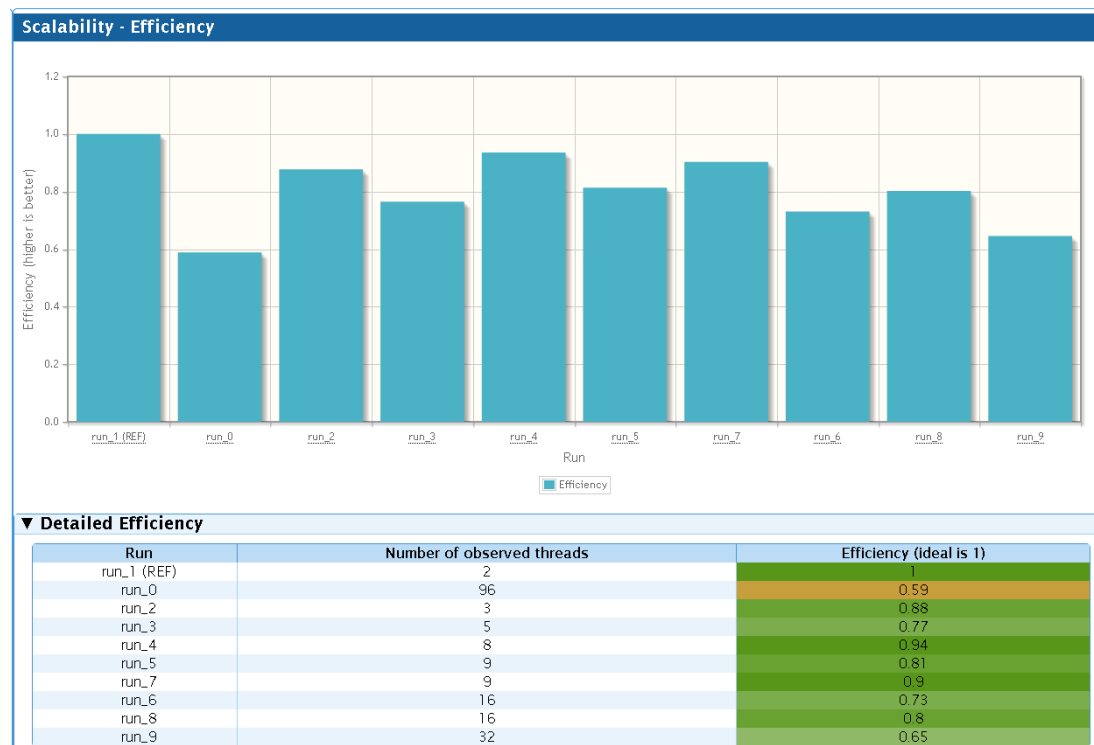
Profiling node c251-109.wrangler.tacc.utexas.edu - process 145897 - thread 145897

Name	Module	Coverage (%)	Time (s)
○ binvcrhs	bt-mz_B.16	24.34	1.3
○ _INTERNAL_25____src_kmp_barrier_cpp_fa608613::__kmp_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





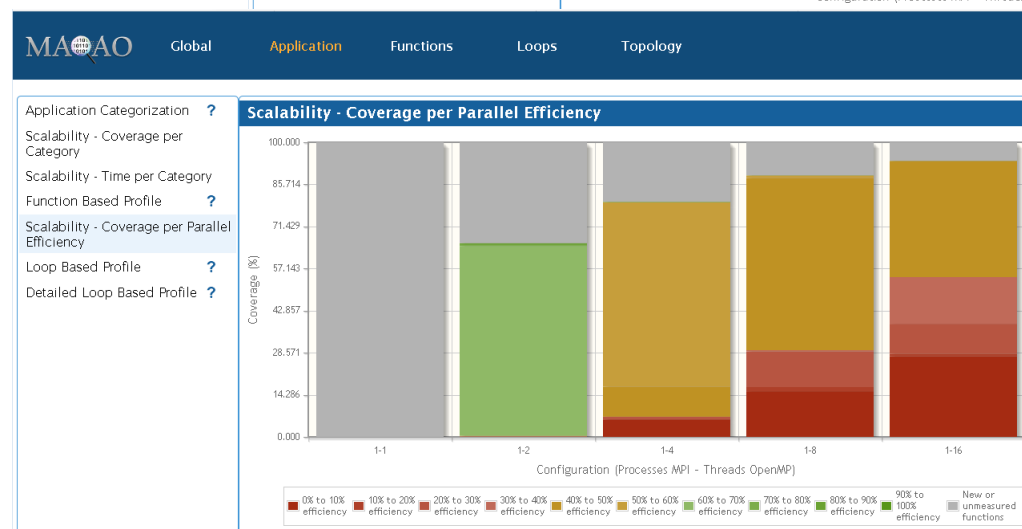
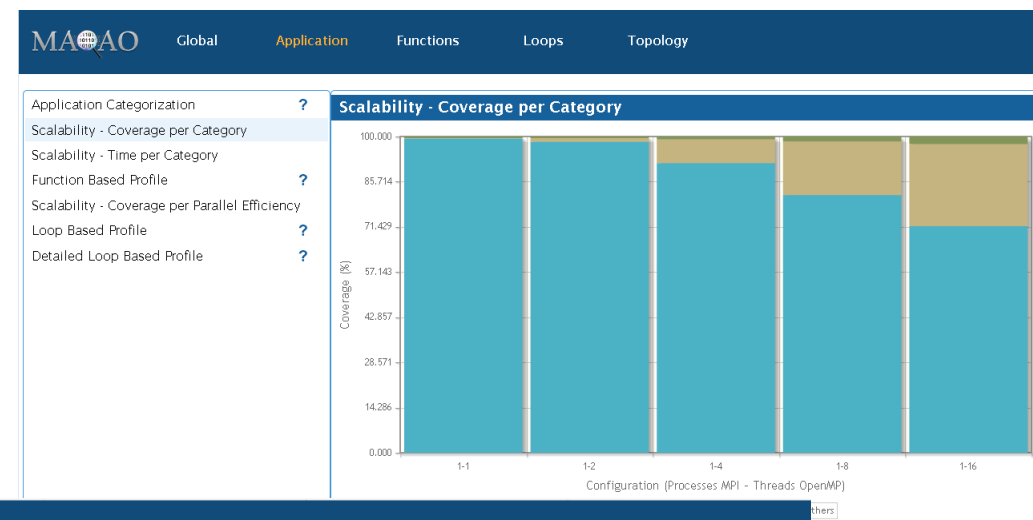
## MAQAO ONE View Scalability Reports – Application View

### 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





## More on MAQAO

---

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

- Documentation: [www.maqao.org/documentation.html](http://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>

# Thank you for your attention !

## Questions ?