



MAQAO Performance Analysis and Optimization Tool

Cédric Valensi, Emmanuel Oseret

cedric.valensi@uvsq.fr, emmanuel.oseret@uvsq.fr

Performance Evaluation Team, University of Versailles

<http://www.maqao.org>

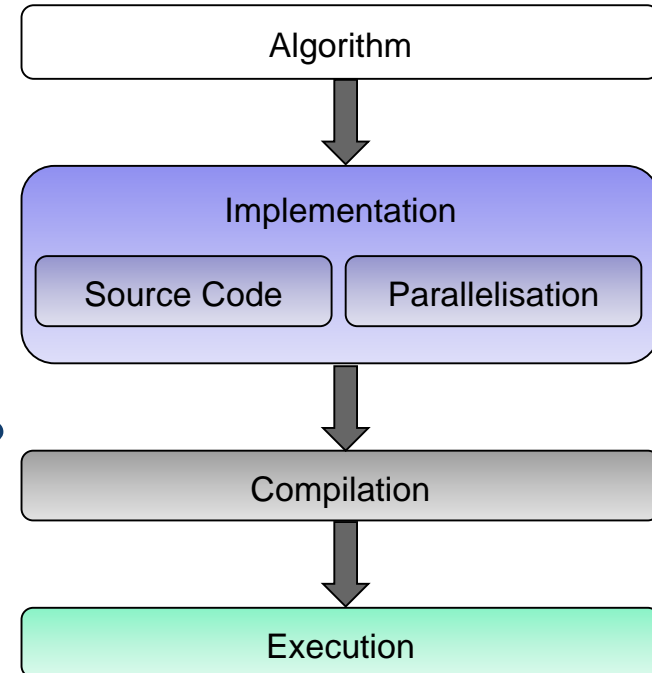


- **How much** can I optimise my application?
 - Can it actually be done?
 - What would the effort/gain ratio be?

- **Where** can I gain time?
 - Where is my application wasting time?

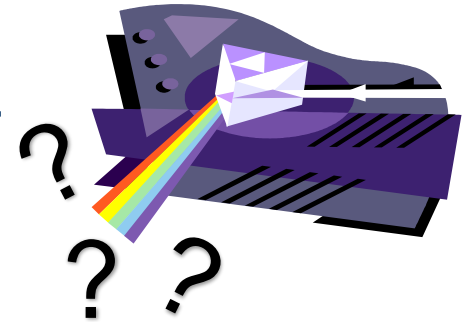
- **Why** is the application spending time there?
 - Algorithm, implementation or hardware?
 - Data access or computation?

- **How** can I improve the situation?
 - In which step(s) of the design process?
 - What additional information do I need?





- **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



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:

- 1) High number of statements (vertical label on the left side of the code block)
- 2) Non-unit stride accesses (points to `ni + nvalue1, nato` and the `gr(nj1, thread_num)` calls)
- 3) Indirect accesses (points to `x(nj1)`, `x(nj2)`, and `x(nj3)`)
- 4) DIV/SQRT (points to `sqrt(drtest2)`)
- 5) Reductions (points to `Eqc = Eqc + Eq` and `virt = virt + gE*rtest2`)
- 6) Variable number of iterations (points to the loop range `ni + nvalue1, nato`)

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



➤ 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.**)

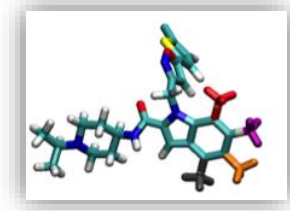
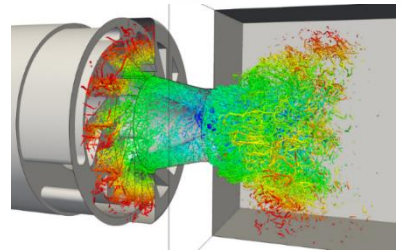
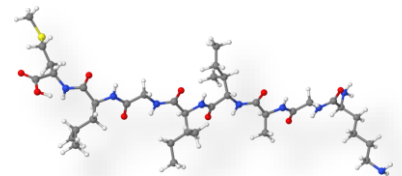


➤ Characteristics:

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



- QMC=CHEM (IRSAMC)
 - Quantum chemistry
 - Speedup: > 3x
 - Moved invocation of function with identical parameters out of loop body
- Yales2 (CORIA)
 - Computational fluid dynamics
 - Speedup: up to 2,8x
 - Removed double structure indirections
- Polaris (CEA)
 - Molecular dynamics
 - Speedup: 1,5x – 1,7x
 - Enforced loop vectorisation through compiler directives
- AVBP (CERFACS)
 - Computational fluid dynamics
 - Speedup: 1,08x – 1,17x
 - Replaced division with multiplication by reciprocal
 - Complete unrolling of loops with small number of iterations





- 2004: Begun development
 - Focusing on Intel Itanium architecture
 - Analysis of assembly files
- 2006: Transition to Intel x86-64
- 2009: Binary analysis support
- 2010: First version of decremental analysis
- 2012: Support of KNC architecture
- 2014 : Profiling features
- 2015: First version of ONE View
- 2017: Prototype support of ARM architecture

The screenshot displays the MAQAO Web Interface with several panels:

- Functions:** A tree view showing the project structure: `Enter Here Project Name`, `L_rbgauss_50_tree_reduce2_2_0`, `L_rbgauss_21_par_region_2_1`, and `Loop SRC L30`.
- Loop's DDG:** A graph showing nodes for instructions like `movl`, `mulsl`, and `addsl` connected by edges representing data dependencies.
- Assembly Code:** A window showing assembly instructions for `Jdemo/recom/recom_original.c`, including `include <omp.h>`, `define _FABS(x) ((x)>0?(x):-x)`, and OpenMP pragmas like `#pragma omp parallel firstprivate(dtpch_hamb;)`.
- Performance Metrics:** A table on the left lists various bounds and ratios:

Loop Bounds:	0
Predecoder bound:	147N
Decoder bound:	147N
Reorder Buffer bound:	157N
Execution ports (qsp) bound:	197N
Execution ports (qgpr) bound:	19700N
Front-end/Back-end ratio:	0.79
Instructions/Cycles ratio:	2.13
Packed Degree:	0.00
Packed LOAD ratio:	0.00
Packed STORE ratio:	0.00
Packed MUL ratio:	0.00
Packed ADD/SUB ratio:	0.00
Execution ports dispatch:	
P0:	8
P1:	10
P2:	19
P3:	1
P4:	1
P5:	5
L1 prediction:	19
L2 prediction min:	32.95
L2 prediction avg:	32.95
RAM prediction:	82.36

The screenshot shows the MAQAO ONE View dashboard with the following sections:

- Experiment Summary:**

Application:	bin/bt.mz.C.16
Timestamp:	2018 10 09 16:40:18
Experiment Type:	MPI, OpenMP
Machine:	skl01.intel.eur
Architecture:	x86_64
Micro Architecture:	SKYLAKE
Model Name:	Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz
Cache Size:	39424 KB
Number of Cores:	28
Compilation Options:	Binary: GNU 7.3.0; fflags: -m tune=generic; march: x86_64; g: 03; -openmp; -funroll-loops; -path /opt/gnu/gcc/7.3.0/lib/gcc/x86_64-pc-linux-gnu/7.3.0/include
- Configuration Summary:**

Dataset:	
Run Command:	<binary>
Number Processes:	1
Number Nodes:	1
Number Tasks per Node:	1
OMP_NUM_THREADS:	4
- Global Metrics:**

Total Time (s):	20.15
Compilation Options:	Binary -funroll-loops is missing
Flow Complexity:	1.00
Array Access Efficiency (%):	70.57
Potential Speedup:	1.02
Clean:	
Nb Loops to get 80%:	3
Potential Speedup:	1.02
FP Vectorised:	
Nb Loops to get 80%:	31
Potential Speedup:	1.39
Fully Vectorised:	
Nb Loops to get 80%:	10
- CQA Potential Speedups Summary:** A line graph showing potential speedup vs. number of processors. The 'Clean' baseline is at 1.0. The 'FP vectorised' series reaches a speedup of approximately 1.39 at 10 processors. The 'Fully vectorised' series reaches a speedup of approximately 1.45 at 10 processors.



- 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, MB3, etc...)



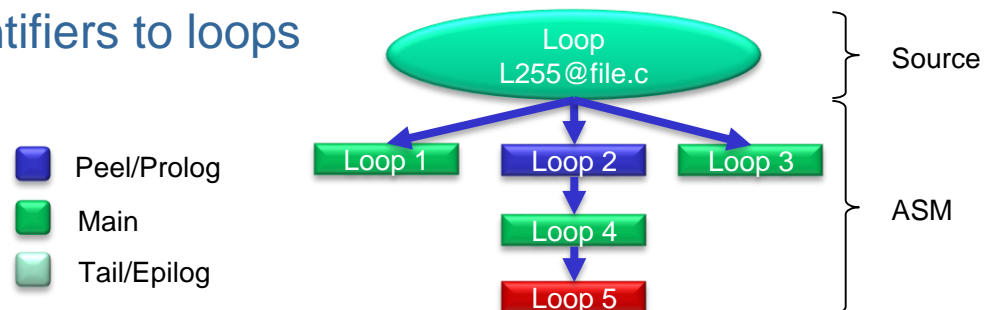
- Provides core technology to be integrated with other tools:
 - TAU performance tools with MADRAS patcher through MIL (MAQAO Instrumentation Language)
 - ATOS bullxprof with MADRAS through MIL
 - Intel Advisor
 - INRIA Bordeaux HWLOC



- PeXL ISV also contributes to MAQAO:
 - Commercial performance optimization expertise
 - Training and software development

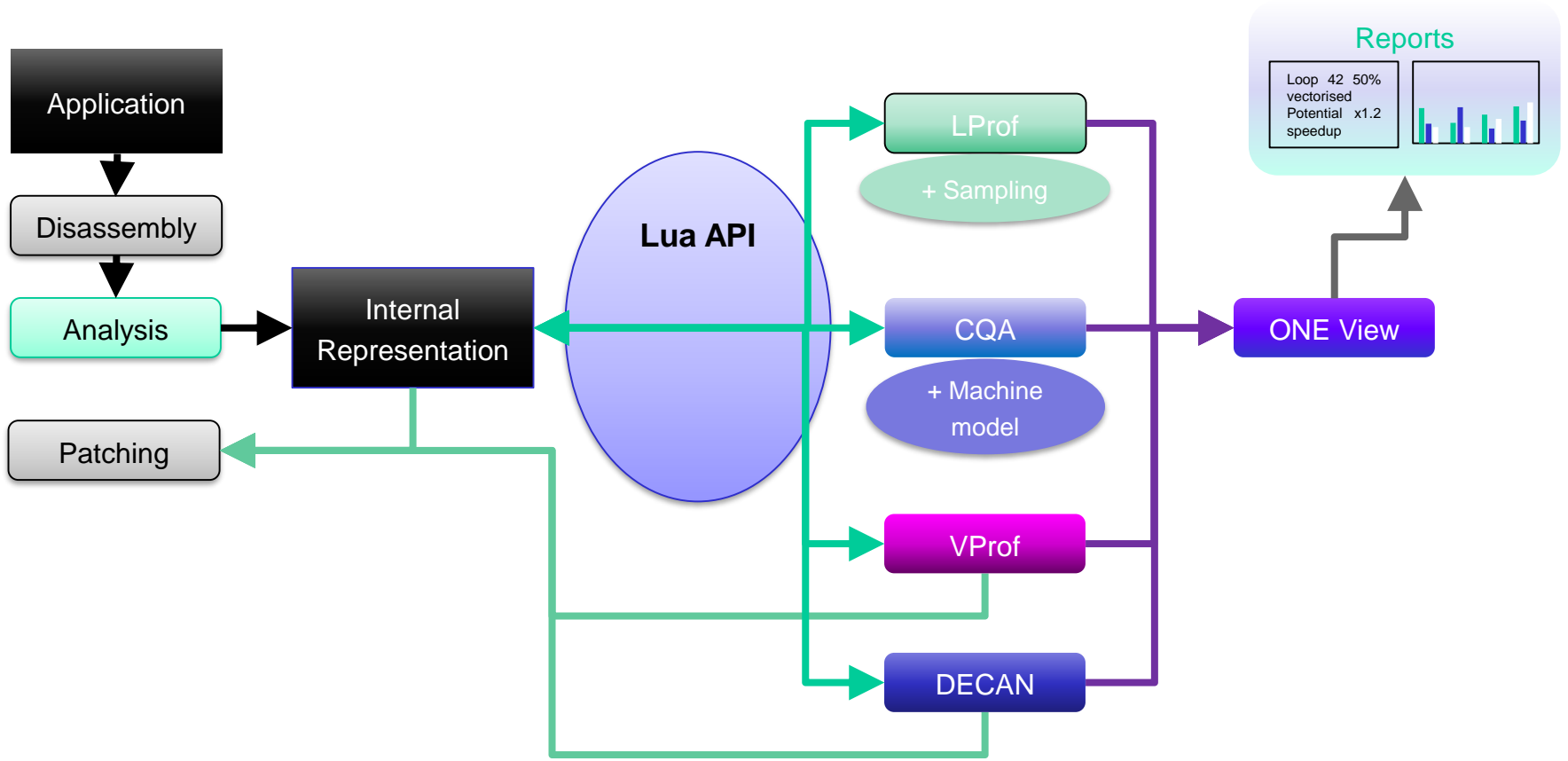


- Advantages of binary analysis:
 - Compiler optimizations increase the distance between the executed code and the source
 - Source code instrumentation may prevent the compiler from applying some transformations
- We want to evaluate the “real” executed code: **What You Analyse Is What You Run**
- Main steps:
 - Reconstruct the program structure
 - Relate the analyses to source code
 - A single source loop can be compiled as multiple assembly loops
 - Affecting unique identifiers to loops





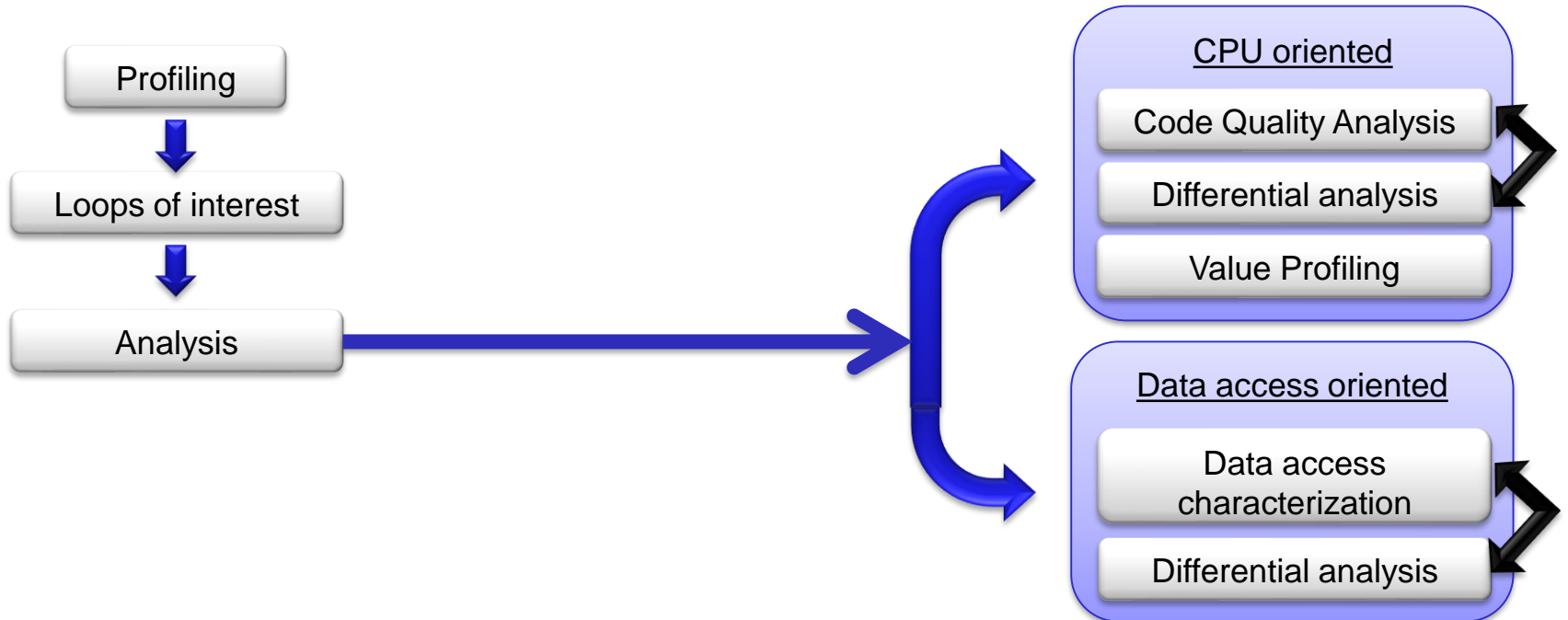
- Binary layer
 - Builds internal representation from binary
 - Allows patching through binary rewriting
- Profiling
 - LProf: Lightweight sampling-based profiler
 - VProf: Instrumentation-based value profiler
- Static analysis
 - CQA (Code Quality Analyzer): Evaluates the quality of the binary code and offers hints for improving it
 - UFS (Uops Flow Simulator): Cycle-accurate CPU engine simulator
- Dynamic analysis
 - DECAN (DECremental Analyzer): Modifies the application to evaluate the impact of groups of instructions on performance
- Performance view aggregation module
 - ONE View: Invokes the modules and produces reports aggregating their results



MAQAO Performance Analysis and Optimization Tool



➤ Decision tree





- **Goal:** Lightweight localization of application hotspots

- **Features:**
 - **Sampling** based
 - Access to hardware counters for additional information
 - Can also access OS timers for unsupported architectures
 - Results at function and loop granularity

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



- Goal: **Assist developers** in improving code performance
- Features:
 - Evaluates the **quality** of the compiler generated code
 - Returns **hints and workarounds** to improve quality
 - Focuses on **loops**
 - In HPC most of the time is spent in loops
 - Targets **compute-bound codes**
- Static analysis:
 - Requires **no execution** of the application
 - Allows **cross-analysis**

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			
<ul style="list-style-type: none"> • 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			
<ul style="list-style-type: none"> • Try another compiler or update/tune your current one: <ul style="list-style-type: none"> ◦ 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: <ul style="list-style-type: none"> ◦ 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.			



- Relies on simplified CPU model
 - Allows faster analyses
 - More precise but slower analyses available with **UFS**
- Machine model:
 - Execution pipeline
 - Port throughput
 - L1 data access
 - Buffers ignored if not UFS
- Key performance levers for core level efficiency:
 - Vectorising
 - Avoiding high latency instructions if possible
 - Having the compiler generate an efficient code
 - Reorganizing memory layout

Same instruction – Same cost



Process up to
8X (SP) data



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

- Implementation changes
 - Improve data access
 - Loop interchange
 - Change loop stride
 - Reshaping arrays of structures
 - Avoid instructions with high latency



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
  
```

Annotations:

- 1) High number of statements (vertical bar on the left)
- 2) Non-unit stride accesses (green boxes pointing to `ni + nvalue1, nato`, `x(nj1)`, `x(nj2)`, `x(nj3)`, and `gr(nj1, thread_num)`, `gr(nj2, thread_num)`, `gr(nj3, thread_num)`)
- 3) Indirect accesses (yellow box pointing to `ceps(ntj)`)
- 4) DIV/SQRT (green box pointing to `sqrt`)
- 5) Reductions (red box pointing to `Eqc = Eqc + Eq`, `Ephob = Ephob + Ed`, and `virt = virt + gE*rtest2`)
- 6) Variable number of iterations (red box pointing to `ni + nvalue1, nato`)
- 7) Vector vs scalar (vertical bar on the left)

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



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 "vector dependences are 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 the arrays are accessed continuously and, otherwise, try to permute loops accordingly.
 - Fortran storage order is column-major: `do i do j a(i,j) = b(i,j)` (slow, non stride 1) => `do i do j a(i,j) = 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)

Execution units bottlenecks

Performance is limited by:

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

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

Proposed solution(s):

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

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 acces

3) Indirect accesses

4) DIV/SQRT

5) Reductions

6) Variable number of iterations

7) Vector vs scalar



- Goal: modify the application to
 - Identify cause of bottlenecks
 - Estimate associated ROI

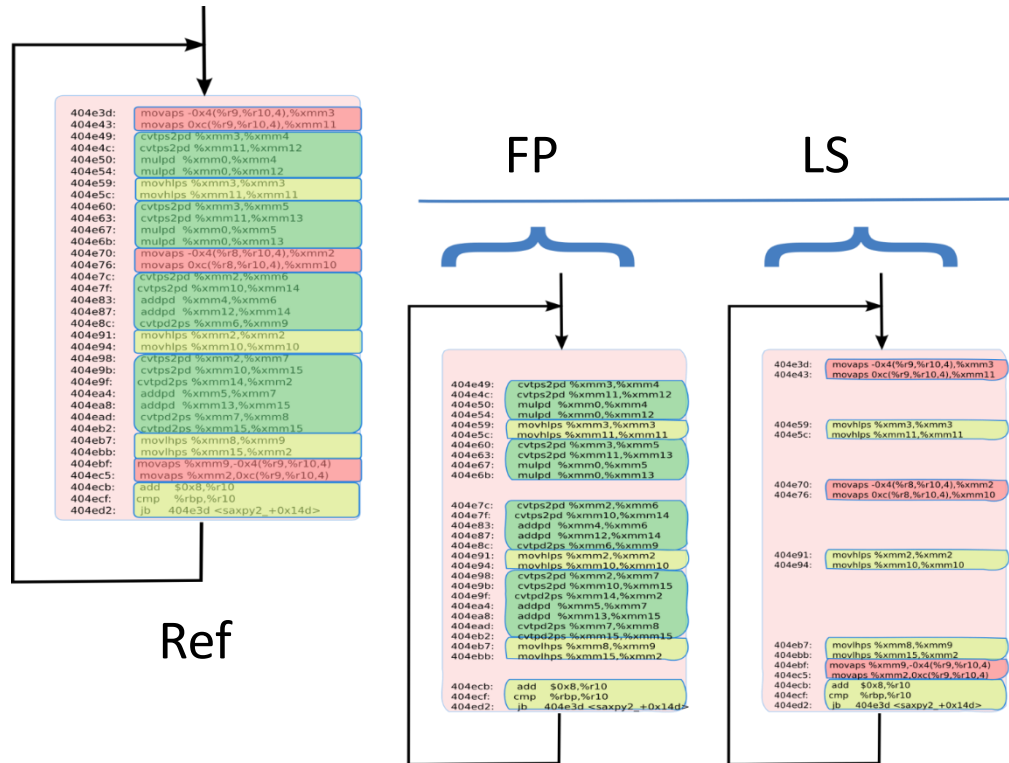
- Differential analysis:
 - Targets innermost loops
 - Transforms loops
 - Measure and compare performance of original and transformed copy

- Transformations
 - Remove or modify groups of instructions
 - Targets memory accesses or computation



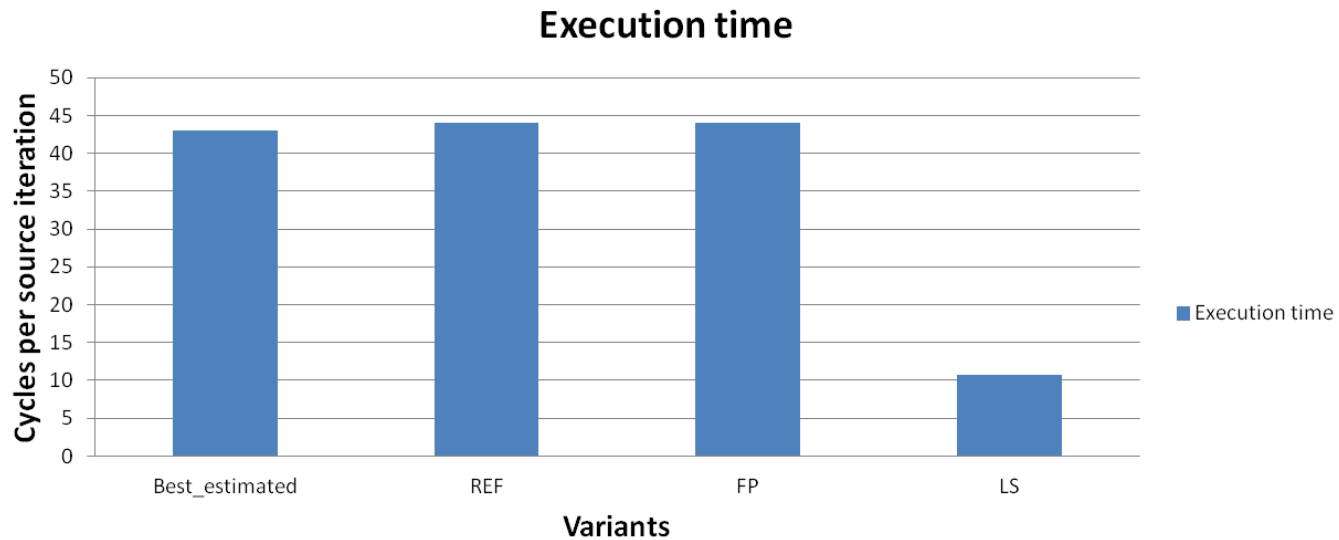
➤ Typical transformations:

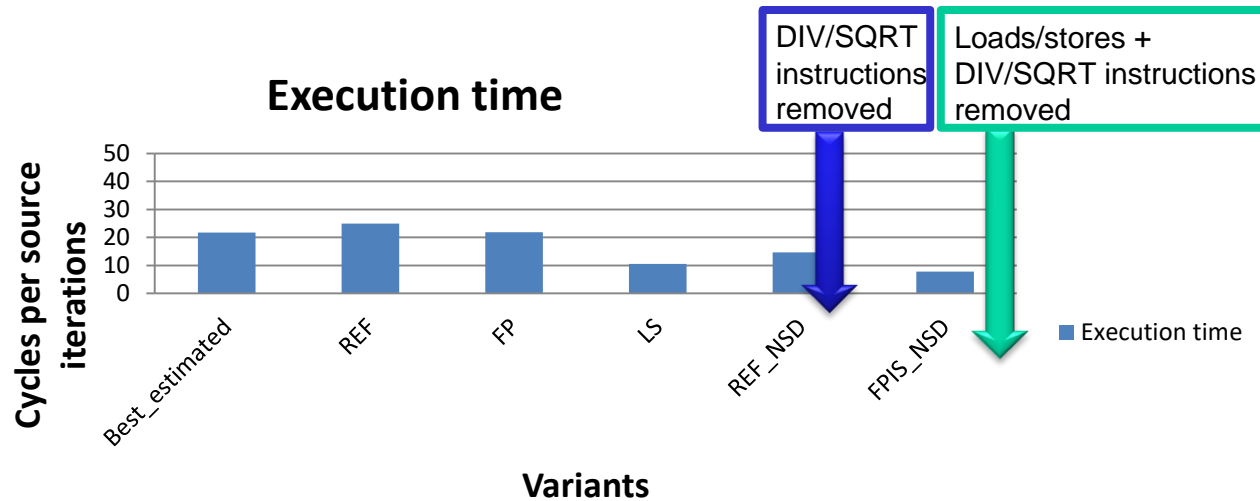
- FP: only FP arithmetic instructions are preserved
 - => loads and stores are removed
- LS: only loads and stores are preserved
 - => compute instructions are removed
- DL1: memory references replaced with global variables ones
 - => data now accessed from L1





- ROI = FP / LS = 4,1
- Imbalance between the two streams
=> Try to consume more elements inside one iteration.





REF_NSD : removing DIV/SQRT instructions provides a 1.5 x speedup

=> the bottleneck is the presence of these DIV/SQRT instructions

FPLS_NSD : removing loads/stores after DIV/SQRT provides a small additional speedup

Conclusion: No room left for improvement here (algorithm bound)



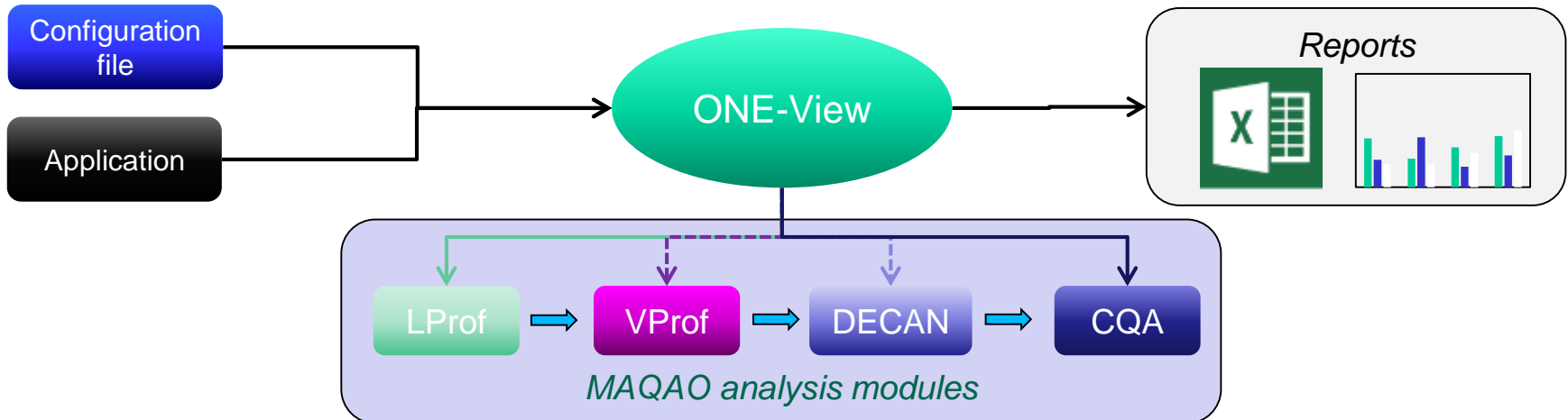
- Value profiling
 - Targets loops or functions
 - Instrumentation
 - Iteration count, loop path uses, function parameters, ...

- Metrics
 - Detection of stable values
 - Loop characterisation through number of iterations

- Provides leads for code specialisation



- Goal: **Automating** the whole analysis process
 - Invocation of the required MAQAO modules
 - Generation of **aggregated performance views** as HTML or XLS files
- Report levels of increasing analysis complexities
 - Each level includes the analyses of the levels below it
 - An experiment directory can be reused for generating a higher level report



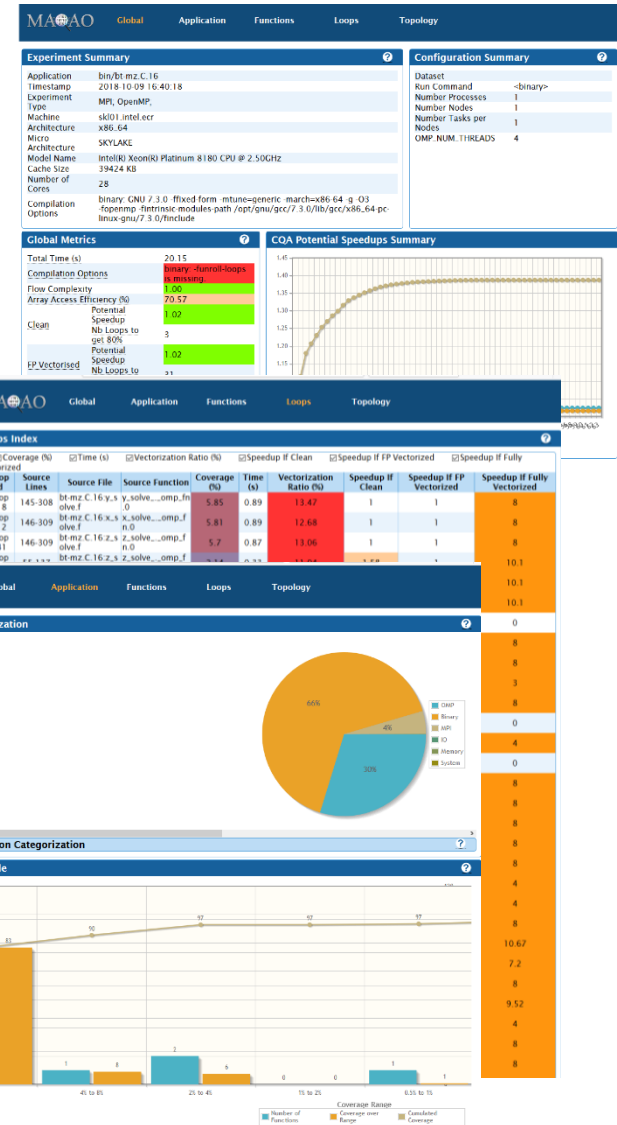


- ONE VIEW **ONE**
 - Requires a single run of the application
 - Profiling of the application using **LProf**
 - Static analysis using **CQA**
- ONE VIEW **TWO** (includes analyses from report **ONE**)
 - Requires **3 or 4 runs** on average
 - Value profiling using **VProf** to identify loop iteration count
 - Decremental analysis for L1 projection using **DECAN**
- ONE VIEW **THREE** (includes analyses from report **TWO**)
 - Requires **20 to 30 runs**
 - Decremental analyses using all **DECAN** variants
 - Collects hardware performance events
- **Scalability**
 - Require as many additional runs as parallel configurations
 - Can be executed in addition of another report
 - Profilings using **LProf** on different parallel configurations



- Main steps:
 - Invokes LProf to **identify hotspots**
 - Invokes CQA, VPROF and DECAN on **loop hotspots**

- Available results:
 - **Speedup** predictions
 - Global code **quality** metrics
 - **Hints** for improving performance
 - Detailed analyses results
 - Parallel efficiency





- Experiment summary
 - Characteristics of the machine where the experiment took place

- 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 in priority





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

$$Global\ Speedup = \sum_{loops} coverage * potential\ speedup$$

- LProf provides coverage of the loops
- CQA and DECAN provide speedup estimation for loops
 - Speedup if loop vectorised or without address computation
 - All data in L1 cache

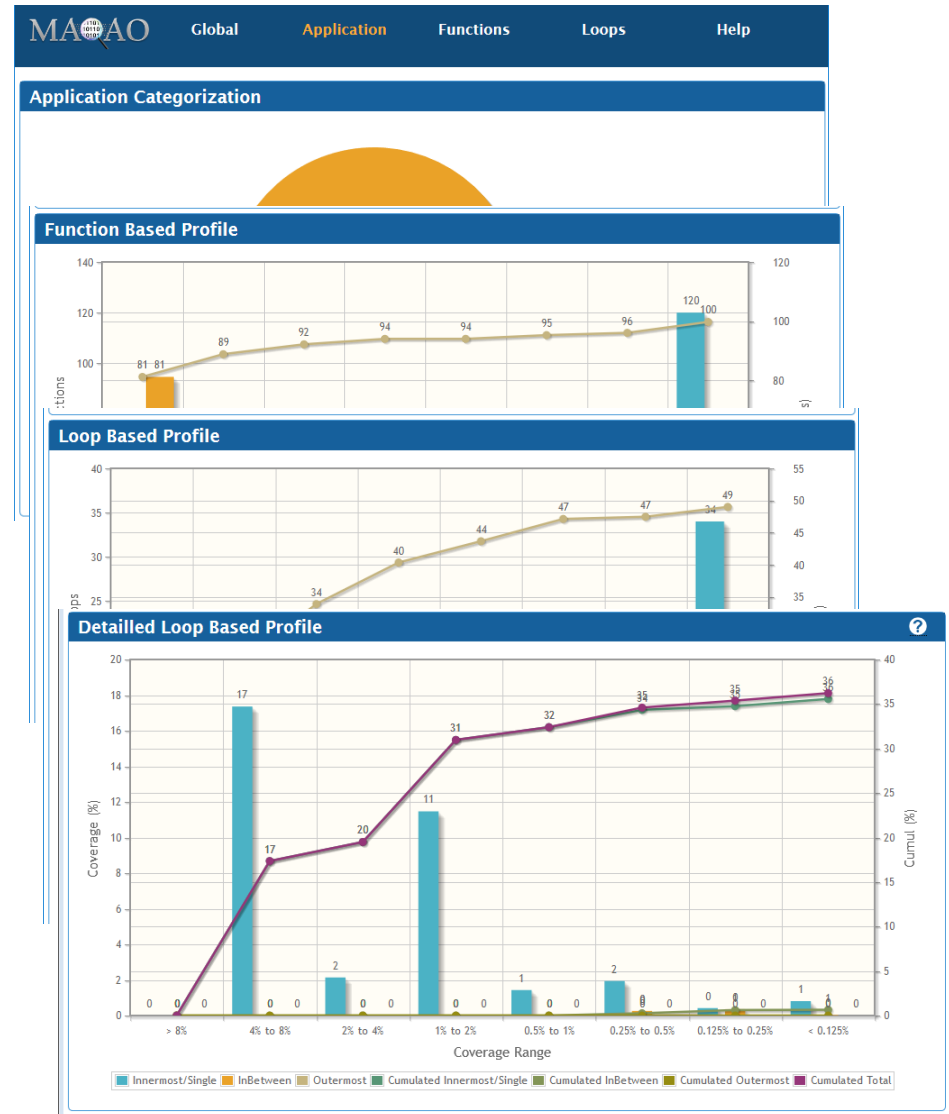


- Application categorisation
 - Time spent in different regions of code

- Function based profile
 - Functions by coverage ranges

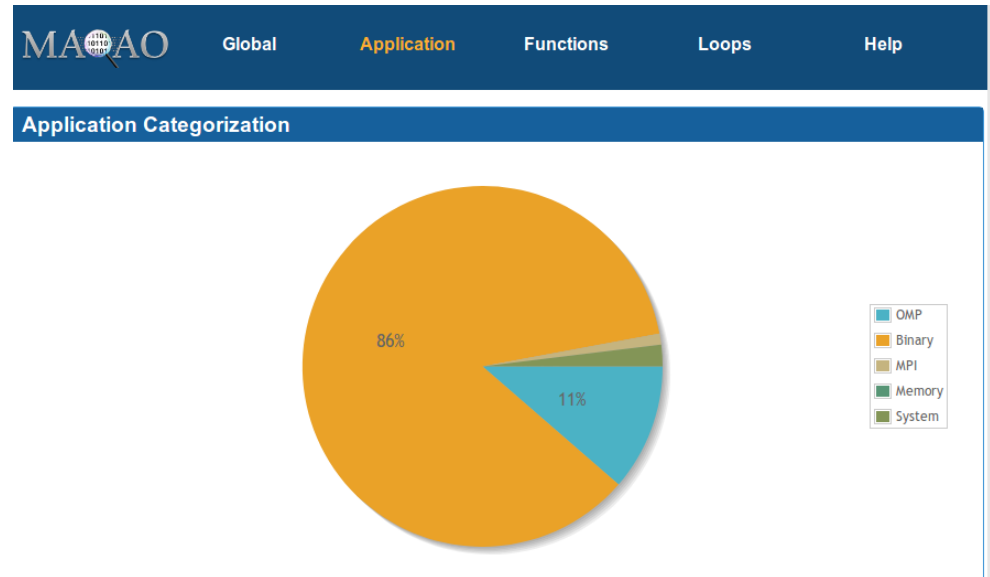
- Loop based profile
 - Loops by coverage ranges

- Detailed loop based profile
 - Loop types by coverage ranges





- Goal: allowing to identify at a glance where time is spent
 - Categories based on functions or libraries names
- Application
 - Main executable
- Parallelization
 - Threads
 - OpenMP
 - MPI
- System libraries
 - I/O operations
 - String operations
 - Memory management functions (allocation, free)
- External libraries
 - Specialised libraries such as libm / libmkl
 - Application code in external libraries





Identifying hotspots

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

MAQAO Global Application Functions Loops Topology					
Functions and Loops					
Filters					
Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
o gomp_team_barrier_wait_end	libgomp.so.1.0.0	21.34	3.26	64	4.47
o binvrchs	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
o 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		
o Loop 118 - y_solve.f:145-308 - bt-mz.C.16		5.85	0.89		
o Loop 119 - y_solve.f:55-137 - bt-mz.C.16		1.77	0.27		
o Loop 116 - y_solve.f:394-398 - bt-mz.C.16		1.08	0.17		
o 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
o 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
o mca_btl_vader_component_progress	mca_btl_vader.so	3.62	0.55	16	1.76
o matvec_sub	bt-mz.C.16	2.73	0.42	64	0.20
o lhsinit	bt-mz.C.16	0.54	0.08	64	0.06
o opal_progress	libopen-pal.so.40.10.1	0.37	0.06	16	0.18
▶ copy_x_face...omp_fn.2	bt-mz.C.16	0.35	0.05	64	0.06
▶ add...omp_fn.0	bt-mz.C.16	0.35	0.05	64	0.05
o binvrchs	bt-mz.C.16	0.33	0.05	64	0.04
o mpi_coll_libnbc_progress	mca_coll_libnbc.so	0.27	0.04	16	0.13
▶ copy_y_face...omp_fn.0	bt-mz.C.16	0.25	0.04	64	0.05
o opal_timer_linux_get_cycles_sys_timer	libopen-pal.so.40.10.1	0.15	0.02	16	0.09
o exact_solution	bt-mz.C.16	0.13	0.02	64	0.03
▶ copy_x_face...omp_fn.3	bt-mz.C.16	0.11	0.02	64	0.03
▶ copy_y_face...omp_fn.1	bt-mz.C.16	0.11	0.02	64	0.03
o gomp_team_barrier_wait_final					
o exact_rhs...omp_fn.0					
o opal_progress@plt					
▶ initialize...omp_fn.0					
o mpi_request_default_wait_all					
o gomp_thread_start					
o Unknown kernel region					

- ▼ matmul_sub
 - o Loop 230 - solve_subs.f:71-175 - bt-mz.C.16
 - o 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
 - o Loop 239 - z_solve.f:146-308 - bt-mz.C.16
 - o Loop 235 - z_solve.f:55-137 - bt-mz.C.16
 - o Loop 234 - z_solve.f:415-423 - bt-mz.C.16

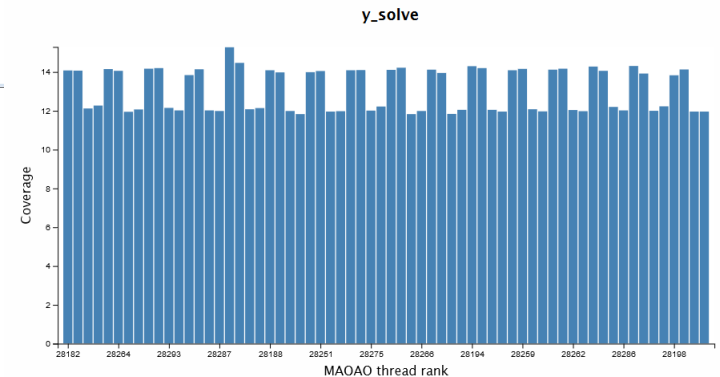
Single

Outermost

Inbetween

Inbetween

Innermost





Identifying loop hotspots

- Vectorisation information
- Potential speedups by optimisation
 - Clean: Removing address computations
 - FP Vectorised: Vectorising floating-point computations
 - Fully Vectorised: Vectorising floating-point computations and memory accesses

MAQAO ONE View Loop Profiling Summary											
Global Application Functions Loops Topology											
Show Full Profile Open Expert Summary											
Loops Index											
<input checked="" type="checkbox"/> Coverage (%)	<input checked="" type="checkbox"/> Level	<input checked="" type="checkbox"/> Time (s)	<input checked="" type="checkbox"/> Vectorization Ratio (%)	<input checked="" type="checkbox"/> Speedup If Clean	<input checked="" type="checkbox"/> Speedup If FP Vectorized	<input checked="" type="checkbox"/> Speedup If Fully Vectorized	<input checked="" type="checkbox"/> Speedup If Data in L1	<input type="checkbox"/> Select none			
Loop id	Source Location	Source Function	Coverage (%)	Level	Time (s)	Vectorization Ratio (%)	Speedup If Clean	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Data in L1	Select none
26831	qmcpack:ParticleBConds3DS oa.h:231-262	qmcplusplus::SoaDistanceTableAA::moveOnSphere	21.06	Single	2.54	100	1	1	1	1.05	
19042	qmcpack:MultiBsplineValue.h pp:56-57	qmcplusplus::BsplineSet >::evaluate	15.42	Innermost	1.86	100	1	1	1	2.6	
19064	qmcpack:MultiBsplineVGLH.h pp:187-207	qmcplusplus::BsplineSet >::evaluate	9.45	Innermost	1.14	100	1.03	1	1	2.4	
13028	qmcpack:BsplineFuncion.h:63-9-643	qmcplusplus::J2OrbitalSoA >::ratio	4.64	Innermost	0.56	0	1.38	1	8	1.08	
19116	qmcpack:MultiBsplineVGLH.h pp:187-207	qmcplusplus::BsplineSet >::evaluate_notranspose	4.31	Innermost	0.52	100	1.03	1	1	3.79	
26829	qmcpack:ParticleBConds3DS oa.h:231-262	qmcplusplus::SoaDistanceTableAA::evaluate	3.98	Single	0.48	100	1	1	1	1.09	
26833	qmcpack:ParticleBConds3DS oa.h:231-262	qmcplusplus::SoaDistanceTableAA::move	3.32	Single	0.4	100	1	1	1	NA	
19142	qmcpack:SplineC2RAdoptor.h:323-373	void qmcplusplus::SplineC2RSoA::assign_vgl >, qmcplusplus::Vector, std::allocator > >>	2.49	Single	0.3	56.36	1	1	1.78	1.51	
19038	qmcpack:SplineC2RAdoptor.h:224-234	qmcplusplus::BsplineSet >::evaluate	1.33	Single	0.16	48.39	1	1	1.69	1.66	
...	qmcpack:ParticleBConds3DS



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 MAQAO ONE View Loop Analysis Report interface. It shows a source code window with a loop analysis report for Loop ID: 224. The report includes a 'Code clean check' section with a 'Workaround' for a slowdown caused by scalar integer instructions. It also features a 'Vectorization' section with a 'Workaround' for extending 'ffast-math' to extend 'take it unit-stride: her elements are accessed ordingly: Fortran storage n stride 1) => do i do j'. The 'FMA' section highlights the 'Presence of both ADD/SUB and MUL operations' and provides a 'Workaround' to 'Recompile with march=skylake-avx512. CQA target is Skylake_SP (Intel(R) Xeon(R) Skodake SPl but specialization flags are march=avx86_64'. The 'Type of elements and instruction set' section states '195 SSE or AVX instructions are processing arithmetic or math operations on double precision FP elements in scalar mode (one at a time)'. The 'Matching between your loop (in the source code) and the binary loop' section notes 'The binary loop is composed of 195 FP arithmetical operations: 70: addition or subtraction, 125: multiply'. The 'Arithmetic intensity' section states 'Arithmetic intensity is 0.06 FP operations per loaded or stored byte'. The 'Unroll opportunity' section notes 'Loop is data access bound' and provides a 'Workaround' to '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.'



- 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	2	0.50
INC %RDI	1	0	0	0	0	1.50	0.50	0	1	1

Source

Assembly

Hide groups analysis

CQA

Advanced

Path 1 / 1

Metric	Value
Coverage (% app. time)	5.85
Time (s)	0.89
CQA speedup if clean	1.00
CQA speedup if FP arith vectorized	1.00
CQA speedup if fully vectorized	8.00
CQA speedup if no inter-iteration dependency	NA
CQA speedup if next bottleneck killed	1.44
Source	y_solve.f:145-308
Source loop unroll info	not unrolled or unrolled with no peel/tail loop
Source loop unroll confidence level	max
Unroll/vectorization loop type	NA
Unroll factor	NA
CQA cycles	204.00
CQA cycles if clean	204.00
CQA cycles if FP arith vectorized	204.00
CQA cycles if fully vectorized	25.50
Front-end cycles	117.00
P0 cycles	97.50
P1 cycles	97.50
P2 cycles	141.33
P3 cycles	141.33
P4 cycles	204.00
P5 cycles	25.00



- Software Topology
 - Nodes list
 - Processes by node
 - Thread by process

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

MAQAO			Global	Application	Functions	Loops	Topology	Help
Software Topology								
ID			Time(s)					
▼ Node skylake01			11.22					
▼ Process 359337			11.22					
○ Thread 359337			11.22					
▶ Process 359338			11.16					
▶ Process 359352			11.22					
▶ Process 359351			11.2					
▶ Process 359353			11.22					
▶ Process 359354			11.22					
▶ Process 359355			11.18					
▶ Process 359356			11.18					
▶ Process 359357			11.22					
▶ Process 359358			11.18					
▶ Process 359359			11.18					
▶ Process 359360			11.16					
▶ Process 359361			11.18					
▶ Process 359362			11.08					
▶ Process 359364			11.22					
▶ Process 359366			11.19					
○ AVERAGE			11.22					

MAQAO				Global	Application	Functions	Loops	Topology	Help
Profiling node skylake01 - process 359337 - thread 359337									
Name	Module	Coverage (%)	Time (s)						
○ MPIDI_CH3I_Progress	libmpi.so.12.0	20.62	2.31						
▶ calc_data_gradient	3D_cylinder	4.95	0.56						
▶ ics_advance_velocity_tfv4a_4th	3D_cylinder	3.75	0.42						
▶ calc_data_tridiag_op_product	3D_cylinder	3.58	0.4						
○ MPIR_Allreduce_group	libmpi.so.12.0	3.22	0.36						
▶ filter_real_data	3D_cylinder	2.43	0.27						
▶ update_int_comm	3D_cylinder	2.42	0.27						
○ system_call_after_swagps	SYSTEM CALL	1.66	0.19						
▶ adv_scalar_w_u_tfv4a_4th	3D_cylinder	1.59	0.18						
▶ solve_linear_system_deflated_pcg	3D_cylinder	1.45	0.16						



- Goal: Provide a view of the application scalability
 - Profiles with different numbers of threads/processes
 - Displays efficiency metrics for application



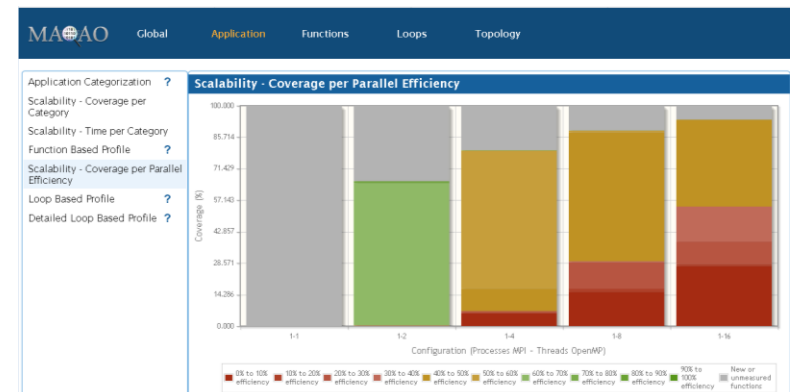
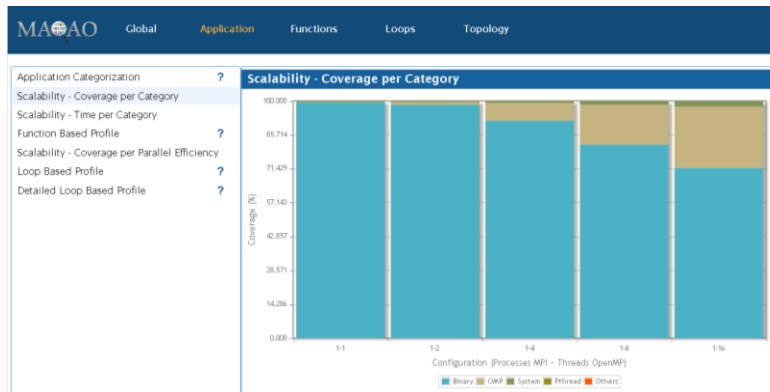
▼ Detailed Speed-Up and Efficiency

Number of threads	Configuration (Processes MPI - Threads OpenMP)	Efficiency (ideal is 1)	Speed Up	Ideal Speed Up	Time (s)
2	1-1	1	1	1	546.3
3	1-2	0.91	1.82	2	300.66
5	1-4	0.79	3.15	4	173.42
9	1-8	0.6	4.78	8	114.18
17	1-16	0.5	8	16	69.26



- 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





Displays metrics for each function/loop

- Efficiency
- Potential speedup if efficiency=1

MAQAO Global Application **Functions** Loops Topology

Functions and Loops

Filters

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)	(1-1) Efficiency	(1-2) Efficiency	(1-2) Potential Speed-Up (%)	(1-4) Efficiency	(1-4) Potential Speed-Up (%)	(1-8) Efficiency	(1-8) Potential Speed-Up (%)	(1-16) Efficiency	(1-16) Potential Speed-Up (%)
o _INTERNAL_25...src_kmp_barrier_cpp_ac7c2c73...kmp_hyper_barrier_release{barrier_type, kmp_info*, int, int, int, void*}	libiomp5.so	24.02	15.38	16	18.62		1	0	0.04	5.49	0.01	14.35	0.01	23
o binvcrhs	bt-mz.C.1	20.71	13.27	16	6.22	1	0.7	6.14	0.55	10.2	0.45	11.58	0.41	11.43
▶ compute_rhs	bt-mz.C.1	10.76	6.9	16	2.45	1	0.63	2.68	0.42	5.39	0.26	8.47	0.25	7.57
▶ matmul_sub	bt-mz.C.1	10.11	6.48	16	2.91	1	0.7	2.91	0.57	4.44	0.44	5.75	0.41	5.45
▶ z_solve	bt-mz.C.1	9.46	6.06	16	2.46	1	0.69	2.69	0.55	4.24	0.42	5.43	0.37	5.61
▶ x_solve	bt-mz.C.1	7.65	4.9	16	2.25	1	0.68	2.48	0.55	3.79	0.46	4.09	0.41	4.18
▶ y_solve	bt-mz.C.1	7.1	4.55	16	2.13	1	0.7	2.06	0.54	3.56	0.45	3.92	0.39	4.11
▶ matvec_sub	bt-mz.C.1	2.88	1.84	16	0.74	1	0.69	0.91	0.57	1.31	0.45	1.62	0.41	1.59
▶ add#omp_loop_0	bt-mz.C.1	1.42	0.91	16	0.40	1	0.64	0.12	0.44	0.22	0.25	0.41	0.09	1.17
o _INTERNAL_25...src_kmp_barrier_cpp_ac7c2c73...kmp_hyper_barrier_gather{barrier_type, kmp_info*, int, int, void*, void*, void*}	libiomp5.so	0.77	0.49	16	1.35		1	0	0.45	0.08	0.17	0.23	0.06	0.62

MAQAO Global Application Functions **Loops** Topology

Loops Index

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_sub.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_sub.f	matmul_sub	0.77	0.23	0.62	0.41	0.48	0.54	0.4	0.62
Loop 151	304-349	bt-mz.C.1.rhs.f	compute_rhs	0.71	0.29	0.65	0.34	0.46	0.56	0.44	0.6



- MAQAO website: www.maqao.org
 - 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)
 - Core sources
 - Publications around MAQAO:
<http://www.maqao.org/publications.html>



➤ MAQAO Team

- Prof. William Jalby
- Cédric Valensi, Ph D
- Emmanuel Oseret, Ph D
- Mathieu Tribalat
- Salah Ibn Amar
- Youenn Lebras
- Kévin Camus

➤ Collaborators

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

➤ Past Collaborators or Team members

- Prof. Denis Barthou
- 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
- Jean-Baptiste Le Reste
- Sylvain Henry, Ph D
- José Noudohouennou, Ph D
- Aleksandre Vardoshvili
- Romain Pillot



Thanks for your attention!

Questions ?