MAQAO
Performance Analysis and Optimization Framework

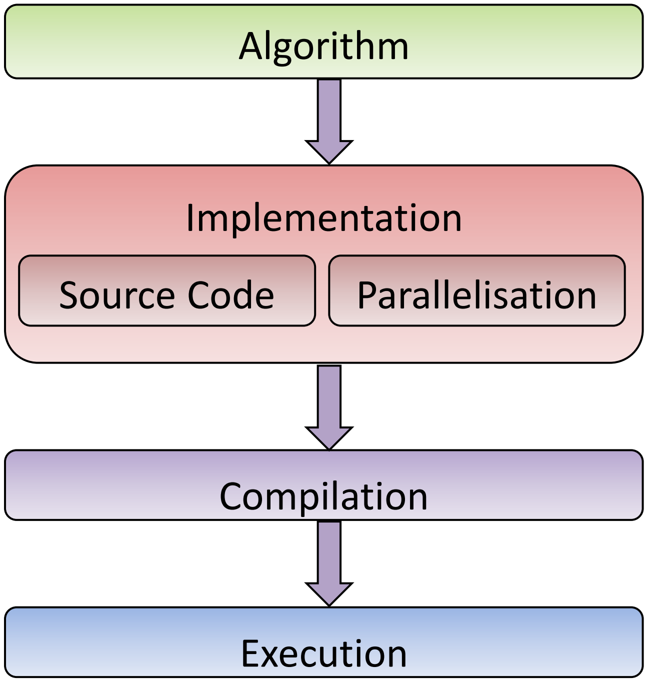
Performance Evaluation Team, University of Versailles Paris-Saclay

http://maqao.exascale-computing.eu

http://maqao.org

Performance Analysis and Optimisation Main Questions

- **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 situation?
  - At which step(s) of the design 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
```

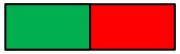
Where are the bottlenecks??

➔ 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**

MAQAO: Modular Assembly Quality Analyzer and Optimizer

- Objectives:
  - Characterizing performance of HPC applications
  - **Guiding users** through optimization process
  - Estimating return of investment (**R.O.I.**)
- Characteristics:
  - Support for **Intel x86-64**, **Xeon Phi** and **AArch64** (beta version)
    - Work in progress on GPU Support
  - **Modular tool** offering complementary views
  - LGPL3 Open Source software
  - Binary release available as **static executable**
- 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

Major Partnerships

- MAQAO is funded by UVSQ, Intel and CEA (French department of energy) through Exascale Computing Research (ECR)

- MAQAO received additional funding through French Ministry of Industry's various European projects (FUI/ITEA: H4H, COLOC, PerfCloud, ELCI, POP2 CoE, TREX CoE, etc...)

- Long term relation/collaboration with
  - CEA DAM
  - CEA Life Science
  - ATOS
  - ……

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
  - Hugo Bolloré , M.Sc.Eng
  - Kévin Camus, Eng.
  - Aurélien Delval, Eng.
  - Max Hoffer, Eng.

- **Collaborators**
  - David J. Kuck, Prof. (Intel Fellow)
  - Andrés S. Charif-Rubial, Ph.D. (CEA)
  - Eric Petit, Ph.D. (Intel)
  - Pablo de Oliveira, Ph.D. (McF UVSQ)
  - David Wong, Ph.D. (Intel)
  - Othman Bouizi, Ph.D. (Intel)
  - AbdelHafid Mazouz Ph.D.(Intel)
  - Jeongnim Kim (Intel)

- **Past Collaborators or Team members**
  - Denis Barthou, Prof. (Univ. Bordeaux)
  - Jean-Thomas Acquaviva, Ph.D. (DDN)
  - Stéphane Zuckerman, Ph.D. (McF Univ Cergy)
  - 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)
  - Vincent Palomarès, Ph.D. (Google)
  - José Noudohouenou, Ph.D. (AMD)
  - Franck Talbart, M. Sc. Eng (DDN)
  - Nicolas Triquenaux, Ph.D. (DDN)
  - Jean-Baptiste Le Reste , M.Sc.Eng
  - Sylvain Henry, Ph.D.
  - Aleksandre Vardoshvili , M.Sc.Eng
  - Romain Pillot, Eng
  - Youenn Lebras, Ph.D.

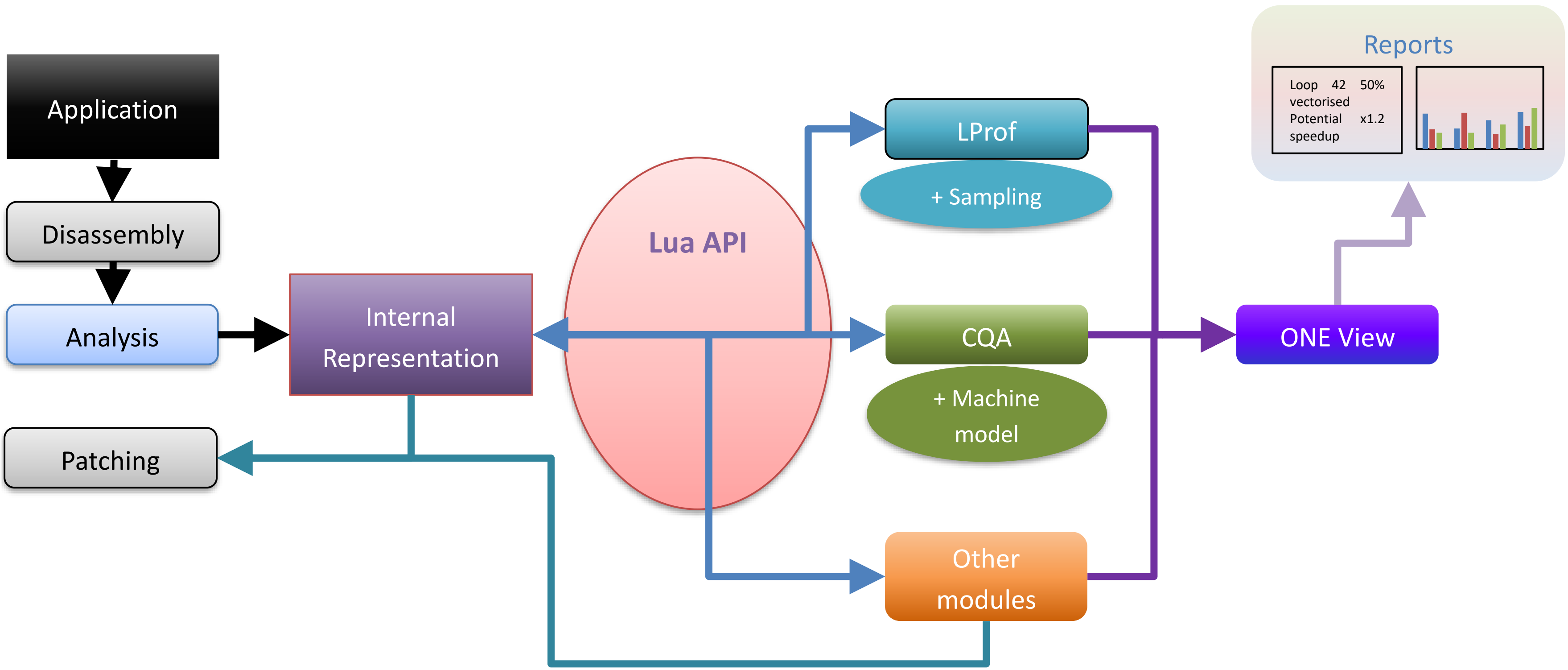
Website & resources

- MAQAO website: www.maqao.org
  - Mirror: maqao.exascale-computing.eu
- Documentation: www.maqao.org/documentation.html
  - Tutorials for ONE View, LProf and CQA
  - Lua API documentation
- Latest release: www.maqao.org/downloads.html
  - Binary releases (2-3 per year)
  - Core sources
- Publications: www.maqao.org/publications.html
- Email: contact@maqao.org

MAQAO Main Features

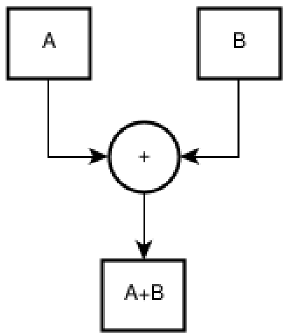
- Binary layer
  - Builds internal representation from binary
  - Allows patching through binary rewriting
- Profiling
  - LProf: Lightweight sampling-based Profiler operating at process, thread, function and loops level
- Static analysis
  - CQA (Code Quality Analyzer): Evaluates the quality of the 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

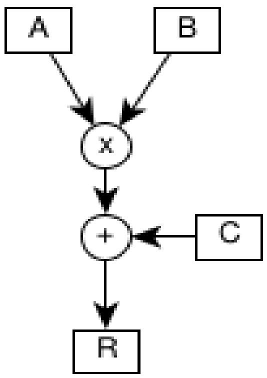


SIMD/Vectorization/Data Parallelism

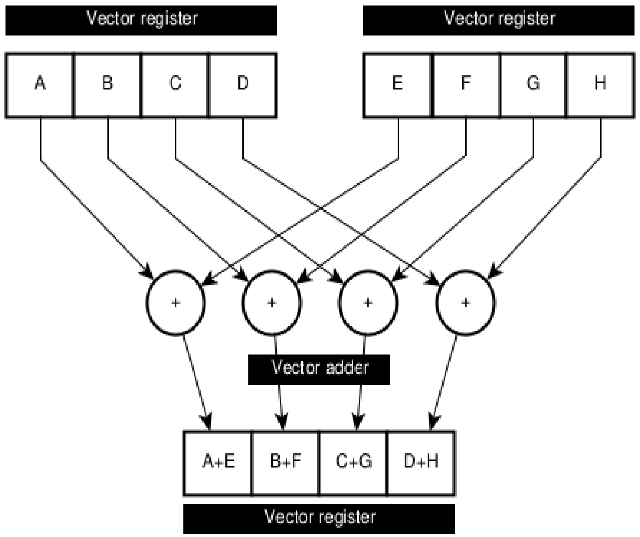
- Scalar pattern (C): a[i] = b[i] + c[i]
- Vector pattern (FORTRAN): a(i, i + 8) = b(i, i + 8) + c(i, i + 8)
- Benefits : increases memory bandwidth and **IPC**
- Implementations:
  - x86 : SSE, AVX, AVX512
  - ARM : Neon, SVE
- FMA/MAC: (the core operation of LinAlg/DSP algorithms)
  - Fused-Multiply-Add
  - Multiply-Accumulate



Scalar addition



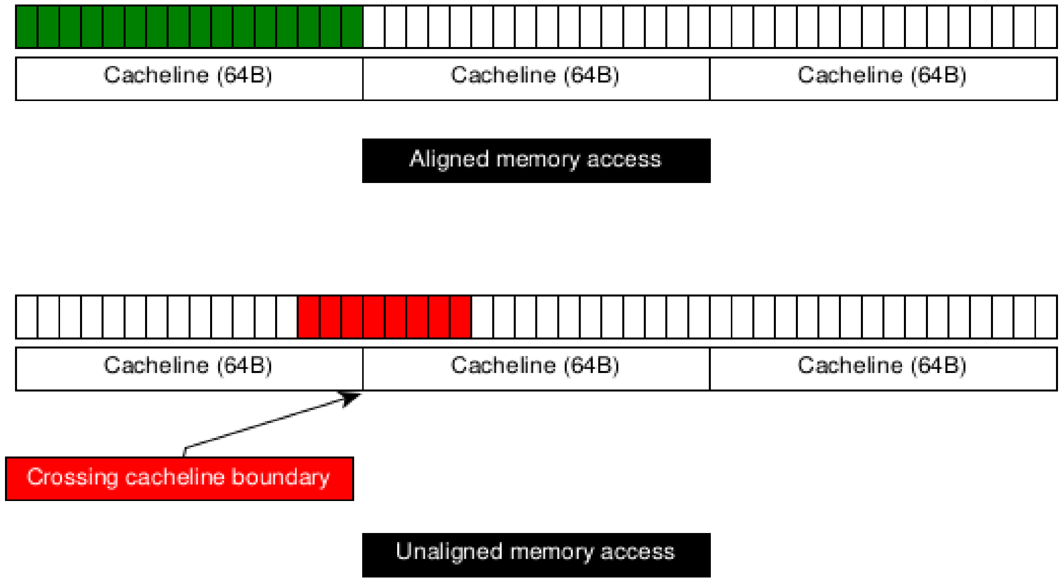
FMA / MAC



Vector addition

Memory and caches

- Computations are, in general, faster than memory accesses
- Alignment/Contiguity of memory (x86) : posix_memalign, aligned_alloc, ...
- Are caches (L1, L2, L3) used properly?
- Memory performance → Maximum bandwidth



Compiler optimisations

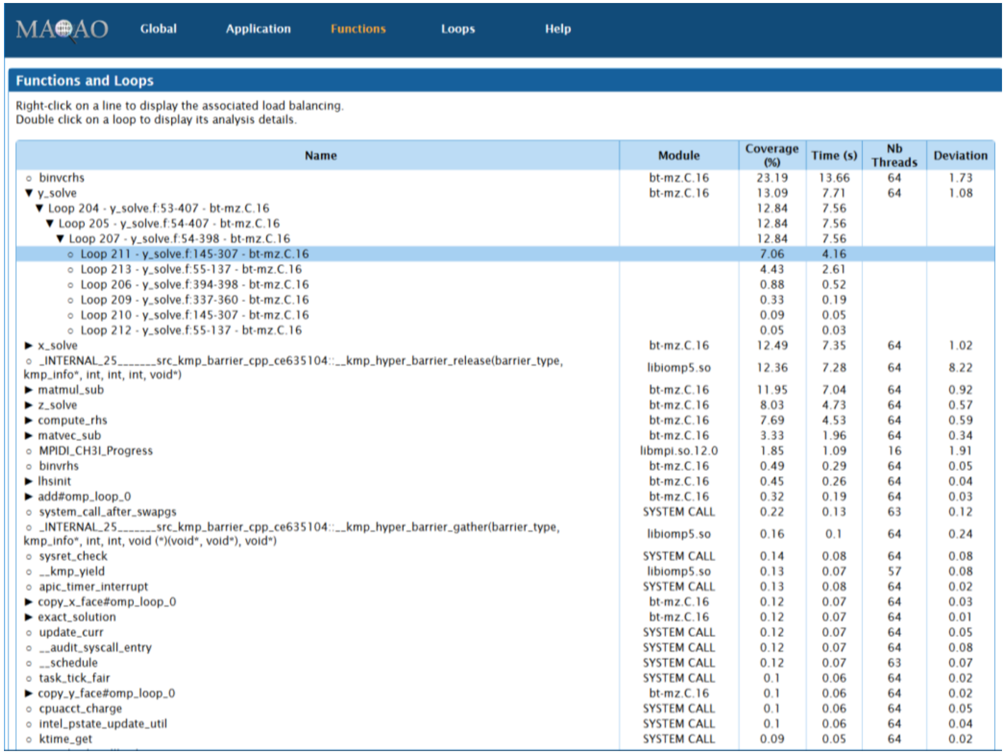
- Compiler flags:
  - Loop unrolling: -funroll-loops
    - Reduce branches
    - Fill the pipeline (more instructions per iteration)
    - Increases memory bandwidth and IPC
  - Function inlining: -finline-functions
  - Vectorization: -ftree-vectorize, -ftree-slp-vectorize, ...
  - Target micro-architectures: -march or -mtune or -xHOST
- Compiler directives:
  - OpenMP directives: #pragma omp simd, #pragma omp parallel for, …
  - Intel compiler specific: #pragma simd, #pragma unroll, #pragma inline, …
- Compiler/language keywords/features:
  - Using restrict for pointers aliasing in C/C++
  - Using inline for function inlining in C
  - Using array sections in FORTRAN

MAQAO LProf: Lightweight Profiler

- **Goal**: 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



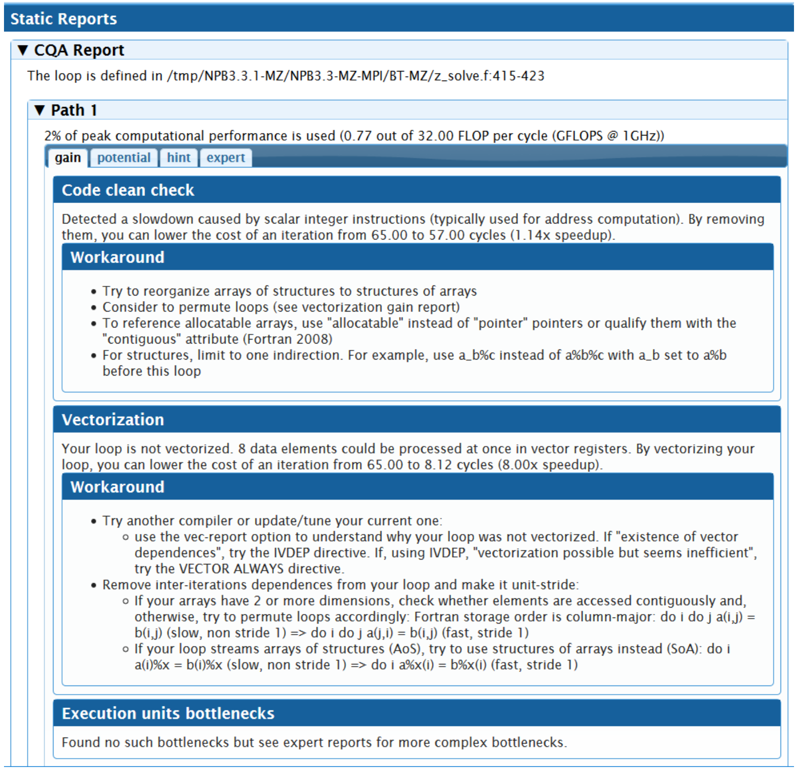
Functions and Loops

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	bt-mz.C.16	8.03	4.73	64	0.57
▶ compute_rhs	bt-mz.C.16	7.69	4.53	64	0.59
▶ matvec_sub	bt-mz.C.16	3.33	1.96	64	0.34
○ MPIDI_CH3I_Progress	libmpi.so.12.0	1.85	1.09	16	1.91
○ binvrhs	bt-mz.C.16	0.49	0.29	64	0.05
▶ lhsinit	bt-mz.C.16	0.45	0.26	64	0.04
▶ add#omp_loop_0	bt-mz.C.16	0.32	0.19	64	0.03
○ system_call_after_swapgs	SYSTEM CALL	0.22	0.13	63	0.12
○ _INTERNAL_25_______src_kmp_barrier_cpp_ce635104::__kmp_hyper_barrier_gather(barrier_type, kmp_info*, int, int, void (*)(void*, void*), void*)	libiomp5.so	0.16	0.1	64	0.24
○ sysret_check	SYSTEM CALL	0.14	0.08	64	0.08
○ __kmp_yield	libiomp5.so	0.13	0.07	57	0.08
○ apic_timer_interrupt	SYSTEM CALL	0.13	0.08	64	0.02
▶ copy_x_face#omp_loop_0	bt-mz.C.16	0.12	0.07	64	0.03
▶ exact_solution	bt-mz.C.16	0.12	0.07	64	0.01
○ update_curr	SYSTEM CALL	0.12	0.07	64	0.05
○ __audit_syscall_entry	SYSTEM CALL	0.12	0.07	64	0.08
○ __schedule	SYSTEM CALL	0.12	0.07	63	0.07
○ task_tick_fair	SYSTEM CALL	0.1	0.06	64	0.02
▶ copy_y_face#omp_loop_0	bt-mz.C.16	0.1	0.06	64	0.02
○ cpuacct_charge	SYSTEM CALL	0.1	0.06	64	0.05
○ intel_pstate_update_util	SYSTEM CALL	0.1	0.06	64	0.04
○ ktime_get	SYSTEM CALL	0.09	0.05	64	0.02

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
  - Evaluate the **quality** of compiler generated code
  - Proposes **hints and workarounds** to improve quality / performance
  - **Loop 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)

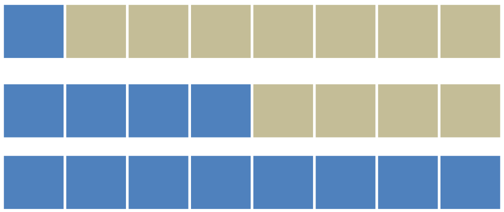
Execution units bottlenecks

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

MAQAO CQA Main Concepts

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

- Main elements of analysis:
  - Peak performance
  - Execution pipeline
  - Resources/Functional units



- Key performance levers for core level efficiency:
  - Vectorisation
  - Avoiding high latency instructions if possible (e.g. DIV/SQRT)
  - Guiding the compiler code optimisation
  - Reorganizing memory and data structures layout

“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 hints

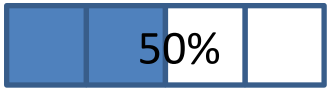
- Compiler can be driven using flags, pragmas and keywords:
  - Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
  - Forcing optimization (unrolling, vectorization, alignment…)
  - Bypassing conservative behaviour when possible (e.g., 1/X precision)

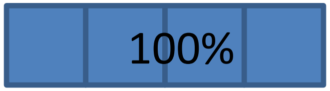
- Hints for implementation changes
  - Improve data access patterns
    - Memory alignment
    - Loop interchange
    - Change loop stride
    - Reshaping arrays of structures
  - Avoid instructions with high latency (SQRT, DIV, GATHER, SCATTER, …)

MAQAO CQA Advanced Features
Vector Efficiency

- Ex: vectorized SSE code on AVX machine
- Compiler: “LOOP WAS VECTORIZED”
- In reality 50% vectorization speedup loss
- CQA:
  - vectorization ratio: 100% (“all instructions vectorized”)
  - vec. efficiency ratio: 50% (“but using only half vector width”)
  - hint: “recompile with –xHost” (on Intel compilers)

128 bits vectorized:	256 bits vectorized:
Vec. ratio = 100%	*Vec. ratio = 100%*
`ADDPD (xmm)`	`VADDPD (ymm)`
`MULPD (xmm)`	`VMULPD (ymm)`
`etc…`	`etc…`





MAQAO CQA Application to Motivating Example

- Issues identified by CQA

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

6) Variable number of iterations

2) Non-unit stride accesses

4) DIV/SQRT

3) Indirect accesses

5) Reductions

2) Non-unit stride accesses

1) High number of statements

7) Vector vs scalar

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

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

MAQAO CQA: Code Quality Analyzer

Application to motivating example

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(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
    do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

Execution units bottlenecks

Performance is limited by:

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

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

Proposed solution(s):

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

Gain | Potential gain | Hints | Experts only

FMA

Detected 48 FMA (fused multiply-add) operations.
Presence of both ADD/SUB and MUL operations.
Proposed solution(s):
Try to change order in which elements are evaluated (using parentheses)
MUL operations to enable your compiler to generate FMA instructions whe
For instance a + b*c is a valid FMA (MUL then ADD).
However (a+b)* c cannot be translated into an FMA (ADD then MUL).

Gain | Potential gain | Hints | Experts only

Slow data structur

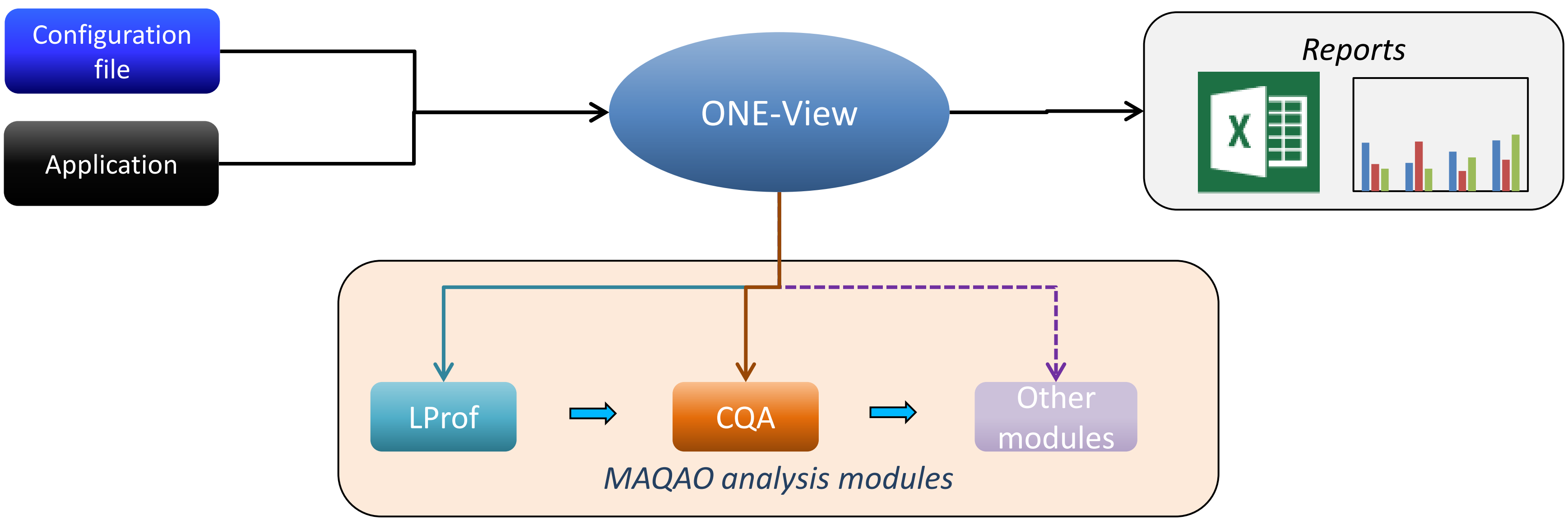
Detected data structures (typically arrays) that cannot be efficiently read/writter

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

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

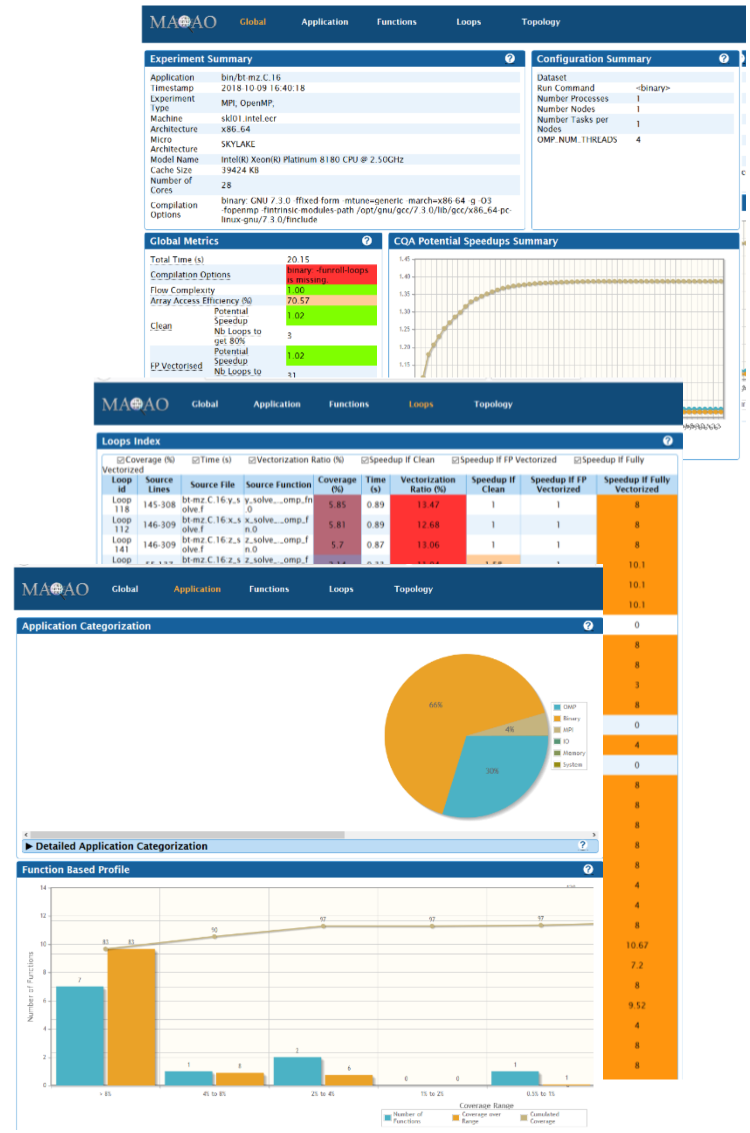
MAQAO ONE View: Performance View Aggregator

- Goal: **Automating** the whole analysis process
  - Invoke multiple MAQAO modules
  - Generate **aggregated performance views**
    - Reports in HTML or XLS format



MAQAO ONE View: Performance View Aggregator

- Main steps:
  - Invokes LProf to **identify hotspots**
  - Invokes CQA and other modules on **loop hotspots**
- Available results:
  - **Speedup** predictions
  - Global **code quality** metrics
  - **Hints** for improving performance
  - Detailed analyses results
  - Parallel **efficiency** analysis



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

Comparative Analysis

- Basic principles: run different "code versions" and compare them on "appropriate levels".
- TRIAL AND ERROR and comparison are fundamental techniques in scientific approach.
- Different "code versions"
  - Different runtime settings (on different number of cores, etc..)
  - Different compilers
  - Different hardware (X86, ARM, …) with same or different ISA
  - Different code versions
- "Appropriate levels":
  - ISOBINARY: the same binary is compared in different settings
  - ISOSOURCE: the same source is compared
  - ISOFUNCTION STRUCTURE: the source code can be different but the function structure is preserved.
  - Generic: much harder to compare
- **Not very sophisticated at first but very useful and implementation is a bit subtle**

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

```
$ maqao oneview -R1 ./myexe
```

```
$ maqao oneview --create-report=one --executable=./myexe --mpi_command="mpirun -n 16"
```

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

- Analyses can be tweaked if necessary
- 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" -- ./myexe
```

  - 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 myexe
```

Online help is available:

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

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

Thanks for your attention

QUESTIONS ?

NAVIGATING ONE VIEW REPORTS

MAQAO ONE View Global Summary

- 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

ONE View Global Metrics

- 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

TYPICAL ONE VIEW GLOBAL TAB

Global Metrics		
Total Time (s)		63.86
Profiled Time (s)		61.31
Time in analyzed loops (%)		61.6
Time in analyzed innermost loops (%)		61.2
Time in user code (%)		61.6
Compilation Options		OK
Perfect Flow Complexity		1.01
Iterations Count		1.00
Array Access Efficiency (%)		88.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	7
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	11
FP Arithmetic Only	Potential Speedup	1.16
	Nb Loops to get 80%	11

FOCUS: on transformations and impact at the application level

Perfect flow complexity: evaluate performance gain if innermost loops had no branches

Iteration count: evaluate the impact of having all loop iteration count over 100

Array Access Efficiency: Percentage of Unit Stride access

TYPICAL ONE VIEW GLOBAL TAB

Global Metrics		
Total Time (s)		63.86
Profiled Time (s)		61.31
Time in analyzed loops (%)		61.6
Time in analyzed innermost loops (%)		61.2
Time in user code (%)		61.6
Compilation Options		OK
Perfect Flow Complexity		1.01
Iterations Count		1.00
Array Access Efficiency (%)		88.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	7
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	11
FP Arithmetic Only	Potential Speedup	1.16
	Nb Loops to get 80%	11

FOCUS: on transformations and impact at the application level

FP vectorized: Performance gain if all the FP arithmetic operations were vectorized

Fully vectorized: Performance gain if all the FP arithmetic operations+ Load/Store instructions were vectorize

MAQAO ONE View: Functions Profiling

Identifying hotspots

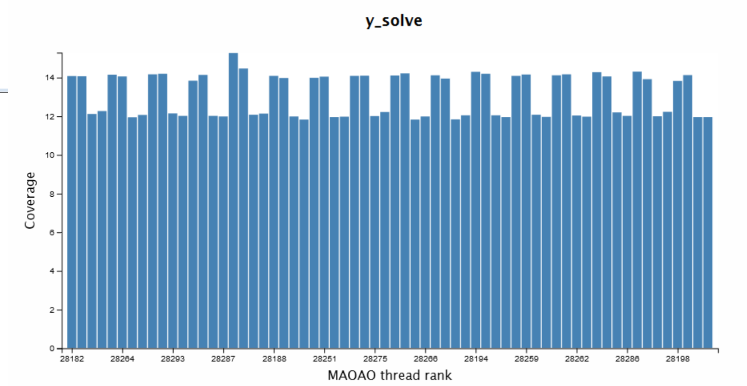
- Exclusive coverage
- Load balancing across threads
- Loops nests by functions

MAQAO | Global | Application | Functions | Loops | Topology

Functions and Loops

▶ Filters

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
○ 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	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
○ binvrhs	bt-mz.C.16	0.33	0.05	64	0.04
○ ompi_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
○ opal_timer_linux_get_cycles_sys_timer	libopen-pal.so.40.10.1	0.15	0.02	16	0.09
○ exact_solution	bt-mz.C.16	0.13	0.02	64	0.03
▶ copy_x_face_._omp_fn.3					
▶ copy_y_face_._omp_fn.1					
○ gomp_team_barrier_wait_final					
○ exact_rhs_._omp_fn.0					
○ opal_progress@plt					
▶ initialize_._omp_fn.0					
○ ompi_request_default_wait_all					
○ gomp_thread_start					
○ Unknown kernel region					



- ▼ 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

MAQAO ONE View Loop Profiling Summary

- 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 Global Application Functions Loops Topology

Show Innermost Profile | Open Expert Summary

Loops Index

▶ Filters

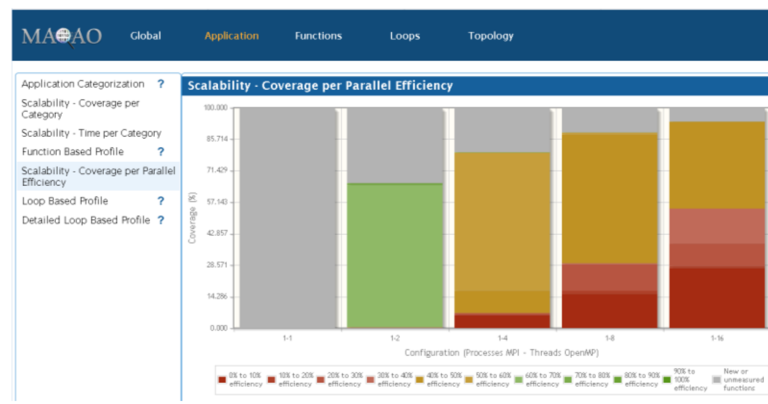
☑Coverage (%) ☑Level ☑Time (s) ☑Vectorization Ratio (%) ☑Speedup If Clean ☑Speedup If FP Vectorized ☑Speedup If Fully Vectorized ☑Speedup If Data in L1 ☐Select none
☐Select All Speed-Ups ☐Select All Efficiencies

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
18403	qmcpack:MultiBsplineValue.hpp:56-57	qmcplusplus::BsplineSet >::evaluate	26.71	Innermost	3.61	100	1	1	1	8.25
26027	qmcpack:cmath:261-464	qmcplusplus::SoaDistanceTableAA::moveOnSphere	12.01	Single	1.62	100	1	1	1	1.03
18424	qmcpack:MultiBsplineVGLH.hpp:187-207	qmcplusplus::BsplineSet >::evaluate	10.81	Innermost	1.46	100	1.06	1	1	4.15
18474	qmcpack:MultiBsplineVGLH.hpp:187-207	qmcplusplus::BsplineSet >::evaluate_notranspose	4.84	Innermost	0.65	100	1.06	1	1	4.52
26026	qmcpack:cmath:261-464	qmcplusplus::SoaDistanceTableAA::evaluate	2.78	Single	0.38	100	1	1	1	1.05
26028	qmcpack:cmath:261-464	qmcplusplus::SoaDistanceTableAA::move	2.64	Single	0.36	100	1	1	1	1.03
8754	qmcpack:CoulombPBCAA.cpp:425-427	qmcplusplus::CoulombPBCAA::evalSR	1.57	Innermost	0.21	0	1.64	2.59	7.67	1.01
12711	qmcpack:BsplineFunctor.h:690-695	qmcplusplus::J2OrbitalSoA >::ratioGrad	1.41	Innermost	0.19	0	1.3	1	16	1.08
18501	qmcpack:SplineC2RAdoptor.h:325-373	void qmcplusplus::SplineC2RSoA::assign_vgl >, qmcplusplus::Vector, std::allocator > > >	1.22	Single	0.16	100	1.01	1	1	3.51

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



ISO BINARY: SCALABILITY RUNS (1)

MINIQMC: Weak Scalability Analysis

r0 : 1 core   r1: 2 cores   r2: 4 cores   r3: 8 cores   r4: 16 cores   r5: 32 cores   r6: 64 Cores

Global Metrics

Metric		r0	r1	r2	r3	r4	r5	r6
Total Time (s)		54.59	56.02	56.89	59.12	67.23	93.17	156.70
Profiled Time (s)		53.81	55.22	56.06	57.98	65.20	89.03	148.10
Time in analyzed loops (%)		51.7	50.7	50.1	49.5	47.5	48.8	46.2
Time in analyzed innermost loops (%)		51.6	50.6	50.0	49.4	47.4	48.7	46.1
Time in user code (%)		52.2	51.3	50.6	49.9	48.0	49.2	46.5
Compilation Options Score (%)		25.0	25.0	25.0	25.0	25.0	25.0	25.0
Perfect Flow Complexity		1.00	1.00	1.00	1.00	1.00	1.00	1.00
Array Access Efficiency (%)		Not Available	Not Available	Not Available	Not Available	Not Available	Not Available	Not Available
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00	1.00	1.00	1.01	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.01	1.01	1.01	1.01	1.01	1.01
No Scalar Integer	Potential Speedup	1.02	1.02	1.02	1.02	1.02	1.01	1.01
	Nb Loops to get 80%	4	4	4	4	4	4	4
FP Vectorised	Potential Speedup	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Nb Loops to get 80%	3	3	3	3	3	3	3
Fully Vectorised	Potential Speedup	1.02	1.02	1.02	1.02	1.02	1.02	1.01
	Nb Loops to get 80%	4	4	4	5	5	5	6
Only FP Arithmetic	Potential Speedup	1.16	1.15	1.15	1.15	1.13	1.12	1.10
	Nb Loops to get 80%	6	6	6	6	7	6	7
Scalability - Gap		1.00	1.03	1.04	1.08	1.23	1.71	2.87

ISO BINARY: SCALABILITY RUNS (2)

MINIQMC: Weak Scalability Analysis

r0 : 1 core r1: 2 cores r2: 4 cores r3: 8 cores r4: 16 cores r5: 32 cores r6: 64 Cores

▼ Colums Filter

☐ Coverage m1o1 (%) ☐ Coverage m1o2 (%) ☐ Coverage m1o4 (%) ☐ Coverage m1o8 (%) ☐ Coverage m1o16 (%) ☐ Coverage m1o32 (%) ☐ Coverage m1o64 (%)
☑ Max Time Over Threads m1o1 (s) ☑ Max Time Over Threads m1o2 (s) ☑ Max Time Over Threads m1o4 (s) ☑ Max Time Over Threads m1o8 (s) ☑ Max Time Over Threads m1o16 (s)
☑ Max Time Over Threads m1o32 (s) ☑ Max Time Over Threads m1o64 (s) ☐ Time w.r.t. Wall Time m1o1 (s) ☐ Time w.r.t. Wall Time m1o2 (s) ☐ Time w.r.t. Wall Time m1o4 (s)
☐ Time w.r.t. Wall Time m1o8 (s) ☐ Time w.r.t. Wall Time m1o16 (s) ☐ Time w.r.t. Wall Time m1o32 (s) ☐ Time w.r.t. Wall Time m1o64 (s) ☐ Nb Threads m1o1 ☐ Nb Threads m1o2
☐ Nb Threads m1o4 ☐ Nb Threads m1o8 ☐ Nb Threads m1o16 ☐ Nb Threads m1o32 ☐ Nb Threads m1o64 ☐ Deviation (coverage) m1o1 ☐ Deviation (coverage) m1o2
☐ Deviation (coverage) m1o4 ☐ Deviation (coverage) m1o8 ☐ Deviation (coverage) m1o16 ☐ Deviation (coverage) m1o32 ☐ Deviation (coverage) m1o64 ☐ Categories m1o1
☐ Categories m1o2 ☐ Categories m1o4 ☐ Categories m1o8 ☐ Categories m1o16 ☐ Categories m1o32 ☐ Categories m1o64 ☐ Compilation Options ☐ (m1o1) Efficiency
☐ (m1o1) Potential Speed-Up (%) ☐ (m1o2) Efficiency ☐ (m1o2) Potential Speed-Up (%) ☐ (m1o4) Efficiency ☐ (m1o4) Potential Speed-Up (%) ☐ (m1o8) Efficiency

Name	Module	Max Time Over Threads m1o1 (s)	Max Time Over Threads m1o2 (s)	Max Time Over Threads m1o4 (s)	Max Time Over Threads m1o8 (s)	Max Time Over Threads m1o16 (s)	Max Time Over Threads m1o32 (s)	Max Time Over Threads m1o64 (s)
○ dgemm_sve_big	libarmpl.so	21.99	22.86	23.6	24.73	29.12	37.66	60.43
▶ void qmcplusplus::DTD_BConds<double, 3u, 39>::computeDistances<qmcplusplus::TinyVector<double, 3u>, qmcplusplus::VectorSoAContainer<double, 3u, 32ul, qmcplusplus::Mallocator<double, 32ul> >, qmcplusplus::VectorSoAContainer<dou...	miniqmc	10.95	11.01	11.15	11.28	11.39	11.6	12.27
▶ void miniqmcreference::MultiBsplineEvalRef::evaluate_v<double>(qmcplusplus::bspline_traits<double, 3u>::SplineType const*, double, double, double, double*, unsigned long)	miniqmc	6.83	6.87	7.01	7.31	8.78	18.58	35.49
▶ void miniqmcreference::MultiBsplineEvalRef::evaluate_vgh<double>(qmcplusplus::bspline_traits<double, 3u>::SplineType const*, double, double, double, double*, double*, double*, unsigned long)	miniqmc	5.44	5.69	5.73	5.98	6.88	8.89	13.9
○ interleave_2vl_sve_kernel_dc	libarmpl.so	1.71	1.92	1.9	2.01	2.21	4.42	11.45
▶ void qmcplusplus::DTD_BConds<double, 3u, 39>::computeDistances<qmcplusplus::TinyVector<double, 3u>, qmcplusplus::VectorSoAContainer<double, 3u, 32ul, q	miniqmc	0.93	0.92	1.05	1.05	1.04	1	1.16

ISO SOURCE: COMPILER COMPARISON

MINIQMC: ARM Clang versus ARM Clang + ARM PL

▼ Compared Reports

- r0: miniqmc_ov1_armclang_o1m1
- r1: miniqmc_ov1_armclang_o1m1_pl

Global Metrics

Metric		r0	r1
Total Time (s)		231.75	53.89
Profiled Time (s)		231.03	53.16
Time in analyzed loops (%)		11.8	51.9
Time in analyzed innermost loops (%)		11.8	51.7
Time in user code (%)		12.0	52.3
Compilation Options Score (%)		25.0	25.0
Perfect Flow Complexity		1.00	1.00
Array Access Efficiency (%)		Not Available	Not Available
Perfect OpenMP + MPI + Pthread		1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00
No Scalar Integer	Potential Speedup	1.00	1.02
	Nb Loops to get 80%	4	6
FP Vectorised	Potential Speedup	1.00	1.00
	Nb Loops to get 80%	3	3
Fully Vectorised	Potential Speedup	1.00	1.02
	Nb Loops to get 80%	5	7
Only FP Arithmetic	Potential Speedup	1.03	1.16
	Nb Loops to get 80%	6	7

ISO FUNCTION STRUCTURE

Global Metrics

Metric		r0	r1	r2	r3	r4
Total Time (s)		29.12	22.53	21.32	19.63	21.80
Profiled Time (s)		28.78	22.18	20.92	19.25	21.49
Time in analyzed loops (%)		87.3	81.1	79.7	79.8	78.7
Time in analyzed innermost loops (%)		37.8	47.1	43.8	51.4	51.7
Time in user code (%)		94.6	90.7	90.0	88.8	88.5
Compilation Options		OK	OK	OK	OK	OK
Perfect Flow Complexity		1.00	1.05	1.00	1.00	1.00
Iterations Count		1.04	1.02	1.03	1.03	1.02
Array Access Efficiency (%)		79.6	81.9	71.8	70.3	71.3
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.23	1.20	1.19	1.17	1.16
	Nb Loops to get 80%	12	13	10	12	11
FP Vectorised	Potential Speedup	1.18	1.27	1.27	1.29	1.27
	Nb Loops to get 80%	14	14	14	17	18
Fully Vectorised	Potential Speedup	3.69	2.86	2.73	2.63	2.58
	Nb Loops to get 80%	41	41	41	41	41
Only FP Arithmetic	Potential Speedup	2.01	1.65	1.65	1.59	1.69
	Nb Loops to get 80%	26	29	28	35	37
Data In L1 Cache	Potential Speedup	1.05	1.06	1.07	1.10	1.08
	Nb Loops to get 80%	5	4	5	6	6

5 successive code versions of CHAMP

Unicore runs SKL

Regular gains except for the last one!!

CHAMP Unicore on SKL

Functions

Name	Module	Time (s)				
		champ_01apr_ov3_energy_15k	champ_26apr_ov3_energy_15k	champ_27apr_ov3_energy_15k	champ_29apr_ov3_energy_15k	champ_11may_ov3_energy_15k
multideterminante	vmc.mov1	7.37	4.6	4.07	3.45	3.55
basis_fns	vmc.mov1	2.19	1.85	2.09	1.96	1.93
compute_ymat	vmc.mov1	6.01	0.13	0.13	0.08	3.6
orbitals	vmc.mov1	1.49	1.56	1.47	1.43	1.48
nonloc	vmc.mov1	1.37	1.28	1.38	1.32	1.44
multideterminante_grad	vmc.mov1	1.09	1.09	1.11	1.11	1.19
multideterminant_hpsi	vmc.mov1	1.29	0.9	0.76	0.7	0.82
orbitalse	vmc.mov1	0.79	0.87	0.8	0.81	0.86
matinv	vmc.mov1	0.85	0.94	0.93	0.56	0.7
__powr8i4	vmc.mov1	0.62	0.76	0.71	0.68	0.7
idiff	vmc.mov1	0.65	0.65	0.7	0.66	0.66
splfit	vmc.mov1	0.56	0.55	0.51	0.58	0.61
detsav	vmc.mov1	1.31	0.56	0.5	0.2	0.21
__intel_avx_rep_memset	vmc.mov1	0.12	0.57	0.47	0.53	0.5
__intel_avx_rep_memcpy	vmc.mov1	0.25	0.14	0.42	0.46	0.82
determinante_psit	vmc.mov1	0.49	0.3	0.36	0.32	0.55
update_ymat	vmc.mov1	0.54	0.3	0.24	0.23	0.31
__libm_log_l9	vmc.mov1	0.24	0.31	0.25	0.23	0.23
psinl	vmc.mov1	0.13	0.16	0.14	0.14	0.17
slm	vmc.mov1	0.15	0.16	0.15	0.12	0.13
multideterminants_define	vmc.mov1	0.11	0.13	0.07	0.11	0.12
__libm_exp_l9	vmc.mov1	0.13	0.1	0.09	0.11	0.09
jastrow4e	vmc.mov1	0.14	0.06	0.07	0.05	0.07
compute_determinante_grad	vmc.mov1	0.07	0.04	0.07	0.05	0.07
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

ISO SOURCE: DIFFERENT HARDWARE

All runs were unicore and used the same compiler GNU 11
Code MAHYCO (Arcane framework)

r0: SKL r1: ZEN_2 r2: ZEN_3

Global Metrics

Metric		r0	r1	r2
Total Time (s)		916.81	738.02	592.37
Profiled Time (s)		915.78	734.50	590.03
Time in analyzed loops (%)		72.8	69.3	68.2
Time in analyzed innermost loops (%)		41.7	40.1	41.4
Time in user code (%)		87.7	86.6	85.9
Compilation Options		OK	OK	OK
Perfect Flow Complexity		1.36	1.26	1.28
Array Access Efficiency (%)		64.0	62.1	61.7
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.26	1.17	1.16
	Nb Loops to get 80%	5	5	5
FP Vectorised	Potential Speedup	1.41	1.31	1.29
	Nb Loops to get 80%	5	7	7
Fully Vectorised	Potential Speedup	2.26	1.91	1.88
	Nb Loops to get 80%	16	14	14
Only FP Arithmetic	Potential Speedup	1.46	1.42	1.33
	Nb Loops to get 80%	7	6	6

BACKUP SLIDES

MAQAO History

- 2004: Begun development
  - Focusing on Intel Itanium architecture
  - Analysis of assembly files
- 2006: Transition to Intel x86-64
- 2009: Binary analysis support
  - 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
- 2018: Scalability mode
- 2020: Comparison mode
- 2022: Support of ARM (beta)