ONE View: a fully automatic method for aggregating key performance metrics and providing users with a synthetic view of HPC applications

William Jalby, Cédric Valensi, Mathieu Tribalat, Kevin Camus, Youhenn Lebras, Emmanuel Oseret, Salah Ibnamar

Abstract One of the major issues in the performance analysis of HPC codes is the difficulty to fully and accurately characterize the behavior of an application. In particular, it is essential to precisely pinpoint bottlenecks and their true causes. Additionally, providing an estimation of the possible gain obtained after fixing a particular bottleneck would surely allow for a more thorough choice of which optimizations to apply or avoid. In this paper, we present ONE View, a MAQAO module harnessing different techniques (sampling/tracing, static/dynamic analyses) to provide a comprehensive human-friendly view of performance issues and also guide the user's optimization efforts on the most promising performance bottlenecks.

1 Introduction

The evolution of the recent HPC processors has shown an increase in both, the number of components (larger multi-core/many-cores), and in terms of mechanism complexity (advanced out of order, multilevel memory hierarchies). These trends make the task of application optimization more and more complex: not only the sources of potential performance loss have become more diverse, but several of them can occur simultaneously and with different impacts. Additionally, sorting out the sources from the consequences of performance losses, therefore identifying the right issue to be addressed, becomes increasingly difficult.

To face such challenges, the most advanced performance tools rely heavily on hardware performance counters to locate and identify performance issues. Although the recent generation of microprocessors have increased the number (up to several

W. Jalby e-mail: william.jalby@uvsq.fr · C. Valensi e-mail: cedric.valensi@uvsq.fr · M. Tribalat e-mail: mathieu.tribalat@uvsq.fr · K. Camus e-mail: kevin.camus@uvsq.fr · Y. Lebras e-mail: youenn.lebras@uvsq.fr · E. Oseret e-mail: emmanuel.oseret@uvsq.fr · S. Ibnamar e-mail: mohammed-salah.ibnamar@uvsq.fr
Exascale Computing Research and Université de Versailles St-Quentin-en-Yvelines

thousands) of performance events which can be monitored, very often they fail in delivering to the code developers the type of information needed. For example, when looking for potential improvements to an array access (through blocking or array restructuring), a code developer first needs to know whether it is really the critical performance issue to be tackled, and second, to know how much performance gain can be obtained after applying a specific optimization. If performance counters can help at identifying critical issues (although with some strong limitations), they are completely unable to evaluate the performance impact of a code change. In fact, performance counters are great in providing information on hardware resource usage but not in guiding the code developer through optimization choices. Additionally, code developers need to get an idea of the confidence they can have in the results provided by performance tools. Very few tools provide even a basic estimation of the quality of the measurements carried out. Therefore, the code developer can be completely misled and waste substantial time and efforts on a non-existing issue. Finally, the code developers will mostly be interested in optimizing the code for different data sets and for different configurations (number of cores, nodes, ...), thus needing to aggregate performance views across different cases to study the performance impact and select the right trade off. Unfortunately, this simple aggregation capacity is missing in most of today's performance tools.

In this paper, we will present the ONE View module, an element of the MAQAO performance analysis framework, which aims at precisely helping the code developer in selecting, with some reasonable confidence, the most profitable optimizations. The main contributions of ONE View (presented in this paper) are:

- Provide a methodology and tools capable of projecting/evaluating the potential performance gain of important code optimizations such as: vectorization (full and partial), blocking, array restructuring, prefetching, etc.
- Aggregate static analysis and dynamic measurements, and combine sampling and tracing to provide the user with a full assessment of the application performance behavior.
- Present quality estimates on the measurements carried out allowing the user to get a precise degree of "confidence" in the results provided.
- Provide a framework for automatically generating performance views across multiple configurations, data sets or code variants.

Section 2 briefly presents tools with similar approaches. Section 3 presents the experimental setup used to demonstrate ONE View capabilities on a real full strength application, QMCPACK. Section 4 briefly describes the two major modules in charge of the "what if scenarios" (CQA for static evaluation and DECAN for dynamic evaluation). Section 5 gives an overview of ONE View's organization and Section 6 presents a complete set of results obtained on QMCPACK. Section 7 describes how these results could be used to improve the performance of QMCPACK. Finally, Section 8 covers our conclusion and future work directions.

2 State of the Art

Performance optimization has long been dominated by the iterative process of developing the code, measuring its performance on a target platform, analyzing the measured data to identify inefficiencies, and modifying the code to improve performance. Significant advancements have been achieved by the performance tools community in the domain of probe-based and sample-based instrumentation [5] [9], access to high-resolution timers and hardware counters [12] [10], and parallel profiling and tracing measurement [7] [6] [8] [5] that can scale to fit large HPC machines. Other tools [15] [14] profile the application to generate a synthetic distribution (MPI, OpenMP, CPU, IO...), or combine sampling, loop trip count instrumentation and code static analysis to report vectorization metrics and other code patterns [13] but do not provide an estimation of the projected gain after optimization.

Presently, the state-of-the-art performance analysis tools can process large parallel profiles and trace data [5] [10] [6] [7] [11] generated from performance experiments as well as produce results that generally reflect basic properties of HPC application execution (e.g., time distribution, hardware behavior, load imbalance, synchronization barriers, ...) with a strong focus on parallelism issues such as the use of MPI and OpenMP. However, there is much less support for automated reasoning about performance problems and guidance for performance improvement. Similarly, the reliability of the results is seldom evaluated by the tools themselves.

3 Experimental Setup

To demonstrate ONE View's capabilities, we will present in the next sections measurements performed on a Skylake, Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10 GHz with 186 GB 6-channel 2666 MHz DDR4 RAM. The target reference application is QMCPACK [16]: an open-source, C++, high-performance electronic structure code that implements numerous Quantum Monte Carlo algorithms. In this paper, we used the QMCPACK version from NiO ECP Benchmark Suite, the INTEL compiler 19.0.1.144 and the INTEL MKL Library 2019.1.144

4 Evaluating performance gains of code transformations

To evaluate the potential gain of a transformation on a loop, we rely on two different tools (CQA and DECAN) using different evaluation methods but operating along the same principles. Starting from the original assembly code, we first generate an assembly code variant which corresponds to the code after transformation. Then, the performance of this variant is either computed using static methods (CQA) or directly measured by running the variants (DECAN).

4.1 CQA: Code Quality Analyzer

CQA [1] is a static analysis module which computes various code quality metrics (characteristics of the Control Flow Graph, length of the critical data path, etc, ...) on a segment of a binary code. In particular, for a sequence of basic blocks, CQA produces a timing estimate (number of cycles). This performance estimate relies on a simple hardware model assuming infinite buffer sizes but using an exact functional unit configuration and exact instruction timings: latency and throughput (see [4] which provide detailed information on instruction behavior for all of the x86 family of processors). Since CQA is operating statically, no information is available to determine operand location in the memory hierarchy. By default, CQA will assume that all of the data accesses are serviced from L1. In addition to this simple L1 estimate, CQA will produce L2 (resp. L3, RAM) estimates corresponding to data accessed serviced from L2 (resp. L3, RAM).

CQA basic capabilities have been augmented to generate variants obtained by modifying the original assembly. Since these variants will not be executed but simply evaluated using the CQA performance model, there is no constraint on the modifications performed. Three main variants are used:

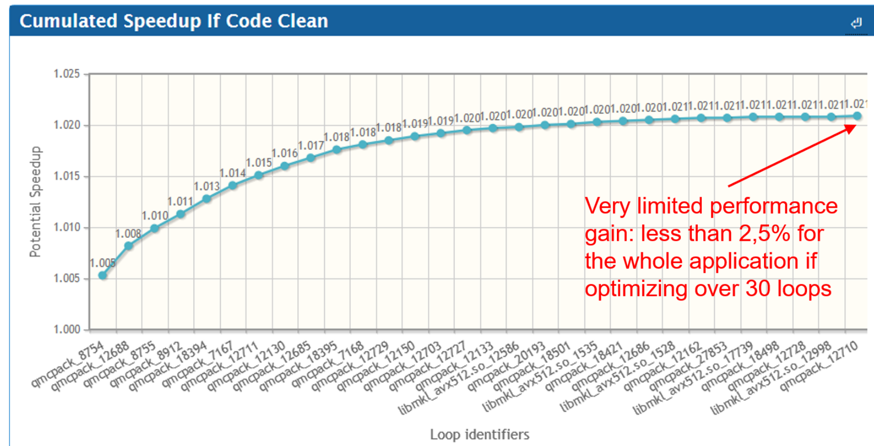


Fig. 1 The vertical y-axis displays the cumulative speedup (on the whole QMCPACK application) which could be obtained by cleaning the loops (removing potential inefficiencies). The horizontal x-axis lists the loops by their decreasing impact in terms of performance gains.

- CODE CLEAN: in this variant, all of the non FP operations are suppressed. The main goal of this variant is to detect cases where the compiler has generated potentially inefficient code. This inefficiency will be quantitatively assessed by running CQA on the “Clean Variant” and comparing the obtained timings with the original ones. Typically, these inefficiencies can be eliminated by using proper compiler switches or permuting loops.

- FP VECTOR: in this variant, first, all of the scalar FP arithmetic instructions are replaced by their vector counterparts. Correspondingly, the load and store instructions which, by the variant definition, have to remain scalar, are replicated and adapted to fill in and use the vector register content.

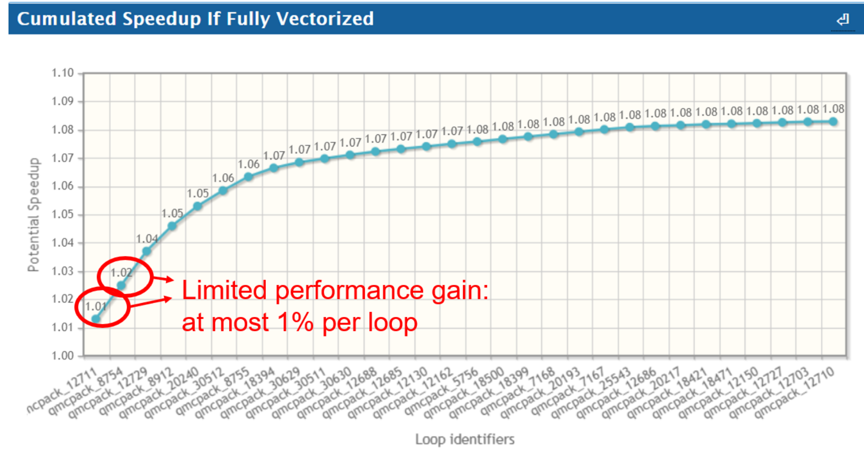


Fig. 2 The vertical y-axis displays the cumulative speedup (on the whole QMCPACK application) which could be obtained by fully vectorizing the loops. The horizontal x-axis lists the loops by their decreasing impact in terms of performance gains.

- FULL VECTOR: in this variant, both, scalar arithmetic, and scalar memory (load/store) instructions are replaced by their vector counterparts. However, non unit stride data accesses (which have no direct vector equivalent) are left scalar or replaced by scatter/gather instructions on the most recent CPU versions. This code variant is essentially equivalent to the code which would be generated by using the SIMD directive which forces the compiler to produce vector code ignoring potential data dependencies.

Figure 1 (resp 2) displays the performance of the Code Clean (Full Vector) variants in QMCPACK. It can be clearly seen that globally, the compiler has generated very efficient code. The maximum potential gain of cleaning (fine tuning) the code would be at most 2,5% and this would require an effort on 25 loops (see Figure 1) which represent quite a large effort for a limited potential gain. As it can be seen on Figure 2, the potential of full vectorization is higher, up to 8% in total with two loops which can offer a performance gain of 1% each.

4.2 DECAN: Differential Analysis

The main goal of Differential Analysis is to precisely identify delinquent instructions (carrying a high performance penalty) and provide a quantitative assessment of their impact. This is performed using DECAN [2], a MAQAO module capable of modifying a loop in the binary file by removing or transforming a subset of instructions through binary rewriting. DECAN can also run and time the modified binary (called a DECAN variant) in order to compare its time with the original unmodified binary time. Given that these transformations can significantly alter the execution, the final application's output will be similarly impacted and its results will most likely be erroneous. In order to limit this impact, extra steps are added to restore the application's context after the measurements are performed. In any case, these variants are not expected to produce meaningful results, since their purpose is the gathering of performance data.

DECAN applies different binary transformations to generate multiple variants. Then, by comparing DECAN variants timings with the original timing, the tool determines the impact on performance of removing/transforming the target subset of instructions. The values of useful hardware event counters are also collected.

In the context of this article, we will focus on the DECAN transformations used in the most significant analyses displayed in ONE View:

- LS Stream: This transformation removes all instructions in target loops except data accesses (loads and stores) and loop control. Observing a large speedup on this variant compared to the original version indicates that the Load/Store instructions are not the limiting factor and that the loop is computation bound.
- FP Stream: This transformation removes all instructions in target loops except those performing FP arithmetic and loop control. A large speedup on this variant indicates that FP arithmetic instructions are not the limiting factor and that the loop is memory (data access) bound.
- DL1: In this variant, all load and store instructions of a target loop are set so that the same address is accessed across different iterations. This ensures that all data accesses are serviced from the L1 cache level. A speedup on this variant means that the corresponding loop suffers from L1 cache misses.

Additional transformations allow to evaluate the front-end stress by replacing all instructions with no-operation instructions (FES transformation), or the correct operation of the prefetcher by inserting prefetch instructions.

In the next section, the use of FP and LS variants will be demonstrated. Figure 3 presents the impact of DL1. At the opposite of the previous transformations (Code Clean and Full Vectorization) which showed limited performance gains, DL1 shows potential large benefit, a single loop transformation bringing about 30% of performance improvement.

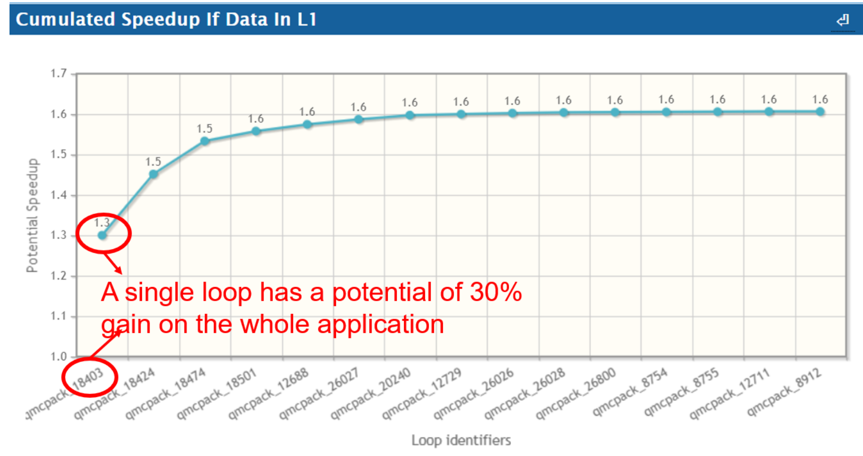


Fig. 3 The vertical y-axis displays the cumulative speedup (on the whole QMCPACK application) which could be obtained by perfect blocking. The horizontal x-axis lists the loops by their decreasing impact in terms of performance gains

5 ONE View: Automated characterization of applications and reporting

In this section, we will describe the overall organization of the ONE View tool. We will start by first describing the profiling tools.

5.1 Profiling

The primary goal of Profiling is to identify and locate the key contributors (functions, loops) to total execution time. Within the MAQAO framework, two profiling techniques are used: sampling and tracing.

The MAQAO LProf module is a lightweight profiler relying on hardware counters sampling to ensure minimal overhead with respect to time and memory usage. LProf provides performance data on functions and loops, and also identifies which should be further investigated.

The MAQAO VProf (Value profiling) module inserts timing probes in the binary file to perform standard tracing measurements. VProf goes further by analyzing each loop instance execution and building a loop behavior summary across the whole application. Loops are known to be executed multiple times (millions, even billions of times) within a single application run. For each loop execution, the Cycles Per Iteration (CPI) value and loop instance number are recorded in “buckets” and sorted according to the CPI. Recording the loop instance number is essential in order to reproduce or track a loop’s behavior. This analysis is critical because loops can

exhibit very different behaviors depending upon the iteration count or cache states. With VProf, the user is capable of not only locating performance issues at loop level, but also at the loop instance level. This allows to rebuild the call chain and precisely locate the issue. In a first pass, 31 instances - representative of a bucket - are identified and will then be used in all further measurements. Using these multiple results, standard statistical metrics (mean, standard deviation, etc...) are calculated, providing an assessment of the quality of the measurement performed.

5.2 Overall ONE View organization

ONE View is a MAQAO module in charge of: **a)** launching all of the other performance modules, **b)** formatting their output, and **c)** aggregating the various performance views in an HTML report, a spreadsheet in the XLSX format, or as formatted text. ONE View offers three levels of reporting:

1. **REPORT ONE**: only LProf and CQA are invoked in order to generate a light application profile and to statically analyze every loop. Generating this report entails a $\sim$10% estimated overhead.
2. **REPORT TWO**: this report includes analyses from REPORT ONE and adds results from a VProf analysis and the DECAN DL1 transformation on the hottest loops of the application. This level requires running the application multiple times thus the resulting overhead is higher (x3). This report provides a full static analysis of vectorization and the Locality analysis performed by DECAN through DL1.
3. **REPORT THREE**: this report includes analyses from REPORT TWO with additional DECAN analyses of all variants as well as the collection of hardware performance events. This level requires to run all DECAN variants and the resulting overhead is much higher, between 2x and x10 depending upon the number of hardware events to be monitored.

ONE View manages the invocation of the various modules with the adequate configurations and options (list of hardware events, ...). The DECAN variants and the various measurements are performed in a single run, heavily using the large number of instances of the same loop. Because the measured loops represent a very small fraction of the overall loop instances, the overhead for a given run is very limited and the corresponding slowdown compared to the original execution time is under x2. To limit the time spent profiling, the user can first run ONE View One (low overhead reporting) and from the obtained results select the hot loops to further investigate. Reducing the number of target loops drastically reduces the overall profiling time.

The concepts presented above can be easily extended to multi-core and multi-node applications. For this, an additional ONE View mode focuses on the scalability properties of loops and/or parallel regions. The ONE View Scalability mode performs multiple invocations of a parallel application with different numbers of threads, processes and nodes defined by the user. The tool then aggregates the re-

sults to compute the efficiency (defined as the ratio between the observed speedup and the expected ideal speedup considering the number of threads) at the application, function, and loop levels.

Figure 4 below shows how all of the modules are combined to provide a detailed performance analysis [3].

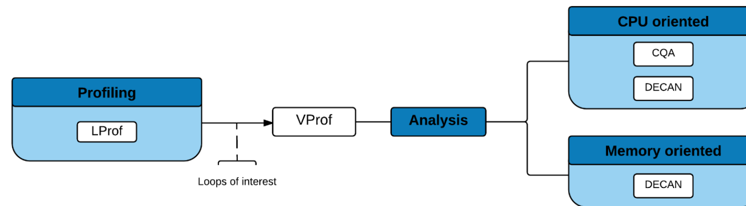


Fig. 4 Methodology outline

6 ONE View Results

In this section, we will present the results produced by ONE View, focusing on a comprehensive set of results particularly useful for the analysis of QMCPACK.

ONE View results are organised along views corresponding to different levels of analyses.

6.1 Global view

This view presents an estimation of the overall quality of the program with regard to performance and the possible improvements to be expected. It includes a set of global metrics, the graphs presenting “what if” scenarios derived from CQA and DECAN analyses (see figures 1, 2 and 3), and a summary of the experiment.

The global metrics aim at giving an overall view of the quality of the code. They include the following values:

- **Timing**: Total execution time of the application; and the percentage spent in loops and innermost loops.
- **Compilation Options**: List of standard optimisation options that were not used when compiling the application. These options include optimisation and architecture specialization flags.
- **Flow Complexity**: Average number of paths in loops. Values closer to 1 are better, since a complex flow makes it harder for the compiler to optimize.

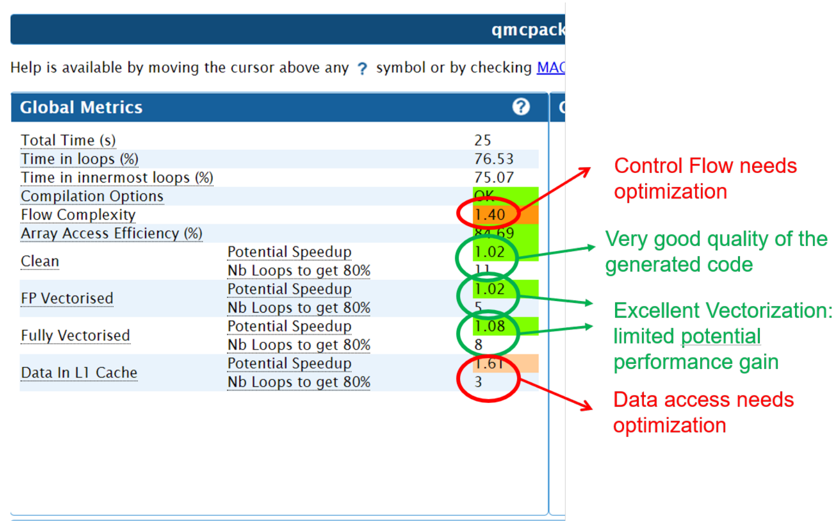


Fig. 5 Global Metrics for QMCPACK. Values are colored from green to red depending on how they influence the program performance (from good to bad).

- **Array Access Efficiency**: Estimation of the regularity of accesses to array elements across the whole application. Higher values mean that most arrays are accessed regularly or at a fixed stride.
- **What-if scenarios**: These metrics are derived from the "what if" scenarios produced by CQA and DECAN. They include for each of them the potential speedup to be expected over the whole application if the optimisation could be applied to every loop in the file, and the number of loops to optimise to obtain 80% of this speedup.

Figure 5 presents an example of these metrics for the initial version of QMCPACK. In this case, the expected speedups if cleaning or vectorizing the code are low. Conversely, the speedup expected for improving data caching is significantly higher, and would require only 3 loops to reach 80%. The average number of paths by loop is 1.4, which means that some improvements could also be expected by simplifying the control flow.

6.2 *Profiling results*

These views focus on the results gathered from the LProf profiling module.

A first view summarises them to provide information on the general profile of the application. It includes a categorization view showing where time is spent in the application or its external dependencies: main application, MPI or OpenMP runtime, memory management, I/O, specialized libraries, etc. It also contains a breakdown

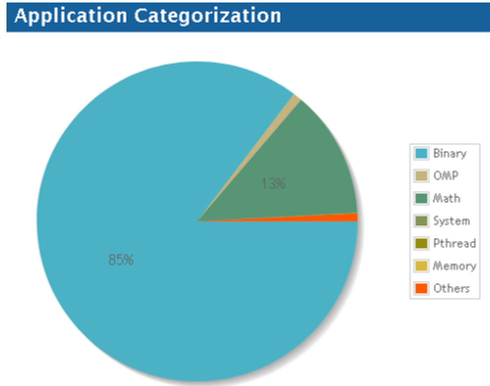


Fig. 6 Code Categorisation for QMCPACK, displaying the percentage of time spent in various code categories involved when executing the application. The three top categories are Binary, which corresponds to the application itself, Maths, which corresponds to functions defined in specialised libraries such as the MKL, and OMP, which corresponds to the functions of the OpenMP runtime specifically

Functions and Loops

▶ Filters

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
▼ qmcplusplus::BsplineSet >::evaluate(qmcplusplus::ParticleSet const&, int, qmcplusplus::Vector >&)	qmcpack	27.86	3.76	1	0.00
∘ Loop 18400 - BsplineSet.h:226-234 - qmcpack		0.21	0.03		
∘ Loop 18399 - BsplineSet.h:226-234 - qmcpack		0.07	0.01		
▼ Loop 18395 - BsplineSet.h:47-57 - qmcpack		0.04	0.01		
▼ Loop 18394 - BsplineSet.h:48-57 - qmcpack		0.18	0.02		
∘ Loop 18403 - BsplineSet.h:56-57 - qmcpack		26.71	3.61		
∘ Loop 18402 - BsplineSet.h:56-57 - qmcpack		0.39	0.05		
∘ Loop 18397 - BsplineSet.h:240-245 - qmcpack		0.02	0		
∘ Loop 18406 - BsplineSet.h:220-224 - qmcpack		0.01	0		
∘ Loop 18404 - BsplineSet.h:695-696 - qmcpack		0	0		
∘ Loop 18405 - BsplineSet.h:695-696 - qmcpack		0	0		
∘ Loop 18398 - BsplineSet.h:240-245 - qmcpack		0	0		
∘ Loop 18396 - BsplineSet.h:240-245 - qmcpack		0	0		
∘ Loop 18401 - BsplineSet.h:226-234 - qmcpack		0	0		
▶ qmcplusplus::SoaDistanceTableAA::moveOnSphere(qmcplusplus::ParticleSet const&, qmcplusplus::TinyVector const&)	qmcpack	12.14	1.64	1	0.00
▶ qmcplusplus::BsplineSet >::evaluate(qmcplusplus::ParticleSet const&, int, qmcplusplus::Vector >&, qmcplusplus::Vector, std::allocator > >&, qmcplusplus::Vector >&)	qmcpack	11.12	1.5	1	0.00

Fig. 7 Function List for QMCPACK. This view lists the functions identified in the application in decreasing order of their coverage. Functions can be expanded to display the loop nests they contain. Functions or loops with a too short execution time are highlighted in orange or red to signal an unreliable value. The Deviation column displays the variation between the coverage of the given function or loop across the different threads of the application.

of the relative coverage of each loop and function of the application allowing to identify how many loops and functions are worth investigating/optimizing. Figure 6 presents an example of the categorization view for QMCPACK. A second more detailed view displays the coverage of each function and of the loops they contain, along with their load distribution across the threads on which the application was ex-

ecuted and the call chains leading to their invocation. Figure 7 presents an example of this view for QMCPACK.

6.3 Loop summary

This view presents all information available on loops, regrouping results from every MAQAO modules involved in the analysis.

Expert Summary

☑Analysis ☐CQA speedup if clean ☐CQA speedup if FP arith vectorized ☐CQA speedup if fully vectorized
☑Number of paths ☑ORIG / DL1 ☐Saturation ratio (MAX(DL1,LS)/REF) ☐Saturation ☐FP/CQA(FP) ☐DL1/CQA(DL1)
☐FP/LS ☐Frequency Impact ☑ORIG (cycles per iteration) ☐STA (ORIG) ☐REF (cycles per iteration) ☐STA (REF)
☑FP (cycles per iteration) ☐STA (FP) ☑LS (cycles per iteration) ☐STA (LS) ☑DL1 (cycles per iteration) ☐STA (DL1)
☑FES (cycles per iteration) ☐STA (FES) ☐CQA cycles ☐CQA cycles if clean ☐CQA cycles if FP arith vectorized
☐CQA cycles if fully vectorized ☐Iteration count ☐Function ☐Source ☐Nb FP_ADD / CPI ☐Nb FP_MUL / CPI
☐CAP(FP) ☐BW(FP) ☐SAT(FP) ☐CAP(L1R) ☐BW(L1R) ☐SAT(L1R) ☐CAP(L1W) ☐BW(L1W) ☐SAT(L1W)
☐CAP(L2) ☐BW(L2) ☐SAT(L2) ☐CAP(L3) ☐BW(L3) ☐SAT(L3) ☐CAP(RAM_R) ☐CAP(RAM_W) ☑Select all

ID	Module	Coverage (% app. time)	Analysis	Number of paths	ORIG / DL1	ORIG (cycles per iteration)	FP (cycles per iteration)	LS (cycles per iteration)	DL1 (cycles per iteration)	FES (cycles per iteration)
▶ Loop 18403	binary	26.71	RAM bound	1	8.49	82.88	6.40	73.20	9.76	8.72
▶ Loop 26027	binary	12.01	Balanced workload (back-end starvation)	128	1.01	150.69	155.04	153.48	148.98	137.44
▶ Loop 18424	binary	10.81	RAM bound	1	3.53	66.94	32.12	75.73	18.98	17.41
▶ Loop 18474	binary	4.84	RAM bound	1	4.18	78.98	32.24	88.04	18.90	17.25
▶ Loop 26026	binary	2.78	L1 bound	128	0.98	173.54	175.29	246.79	177.54	163.29
▶ Loop 26028	binary	2.64	Balanced workload (back-end starvation)	128	0.98	171.90	174.58	168.48	175.27	165.73
▶ Loop 8754	binary	1.57	Balanced workload (fast front-end supply)	1	5.37	47.72	4.35	5.65	9.09	4.84
▶ Loop 12711	binary	1.41	L1 bound	9	1.00	9.86	10.68	10.99	9.88	10.38
▶ Loop 18501	binary	1.22	RAM bound	1	4.40	325.64	79.09	248.73	74.09	63.73
▶ Loop 12729	binary	1.12	Balanced workload (back-end starvation)	9	1.04	8.94	10.36	8.47	8.61	8.91
▶ Loop 12688	binary	1.09	RAM bound	4	2.28	31.92	33.17	35.67	14.00	11.00
▶ Loop 8912	binary	0.97	Balanced workload (fast front-end supply)	6	5.01	49.52	5.75	6.16	9.88	6.23
▶ Loop 26800	binary	0.85	Balanced workload (fast front-end supply)	128	0.83	119.00	122.50	106.00	143.25	107.00
▶ Loop 20240	binary	0.78	RAM bound	1	1.28	10.60	10.00	17.50	8.30	6.05
▶ Loop 8755	binary	0.47	Balanced workload (fast front-end supply)	1	5.02	52.47	4.92	4.20	9.46	4.16

Fig. 8 Loops Expert Summary for QMCPACK. Column ORIG corresponds to the original version of the code, the others to the DECAN transformations with the same name (see 4.2). Values highlighted in red signal a highly unreliable result (execution time below 250 cycles), orange a weakly reliable result (time between 250 and 500), and not highlighted a reliable result (time above 500).

The metrics available include the CQA speedup predictions if the loop can be vectorized or cleaned, the timing of the DECAN variants, and their stability. The stability of a given measure is computed as $(T_{median} - T_{min}) \div T_{min}$, where T_{median} and T_{min} are respectively the median and minimal values across all 31 measurements of the buckets. It can be computed globally for a loop and by buckets.

Figure 8 presents an example of this summary for the initial version of QMCPACK limited to the DECAN variants timings. The loops bound by computation (resp. memory access) can be detected by the timing of the FP variant (resp. LS variant) being close to the timing of the original (ORIG variant). The DECAN timings offer a quantitative estimate of the difference between computation and memory.

Figure 9 adds the stability metrics and iteration counts of the loops. The stability metrics allows to estimate the reliability of a measure. A higher metric means that the value has been varying more between measurements. The relative instabilities of the hottest loops are due to memory accesses.

Expert Summary

☐Analysis ☐CQA speedup if clean ☐CQA speedup if FP arith vectorized ☐CQA speedup if fully vectorized
☐Number of paths ☑ORIG / DL1 ☐Saturation ratio (MAX(DL1,LS)/REF) ☐Saturation ☐FP/CQA(FP) ☐DL1/CQA(DL1)
☐FP/LS ☐Frequency Impact ☑ORIG (cycles per iteration) ☑STA (ORIG) ☐REF (cycles per iteration) ☐STA (REF)
☑FP (cycles per iteration) ☑STA (FP) ☑LS (cycles per iteration) ☑STA (LS) ☑DL1 (cycles per iteration) ☑STA (DL1)
☑FES (cycles per iteration) ☑STA (FES) ☐CQA cycles ☐CQA cycles if clean ☐CQA cycles if FP arith vectorized
☐CQA cycles if fully vectorized ☑Iteration count ☐Function ☐Source ☐Nb FP_ADD / CPI ☐Nb FP_MUL / CPI
☐CAP(FP) ☐BW(FP) ☐SAT(FP) ☐CAP(L1R) ☐BW(L1R) ☐SAT(L1R) ☐CAP(L1W) ☐BW(L1W) ☐SAT(L1W)
☐CAP(L2) ☐BW(L2) ☐SAT(L2) ☐CAP(L3) ☐BW(L3) ☐SAT(L3) ☐CAP(RAM_R) ☐CAP(RAM_W) ☑Select all

ID	Module	Coverage (% app. time)	ORIG / DL1	ORIG (cycles per iteration)	STA (ORIG)	FP (cycles per iteration)	STA (FP)	LS (cycles per iteration)	STA (LS)	DL1 (cycles per iteration)	STA (DL1)	FES (cycles per iteration)	STA (FES)	Iteration count
▶ Loop 18403	binary	26.71	8.49	82.88	0.80	6.40	0.21	73.20	0.34	9.76	0.31	8.72	0.12	25
▶ Loop 26027	binary	12.01	1.01	150.69	0.15	155.04	0.11	153.48	0.19	148.98	0.10	137.44	0.13	96
▶ Loop 18424	binary	10.81	3.53	66.94	0.21	32.12	0.03	75.73	0.27	18.98	0.03	17.41	0.02	51
▶ Loop 18474	binary	4.84	4.18	78.98	0.40	32.24	0.02	88.04	0.46	18.90	0.07	17.25	0.01	51
▶ Loop 26026	binary	2.78	0.98	173.54	0.18	175.29	0.07	246.79	0.25	177.54	0.12	163.29	0.07	96
▶ Loop 26028	binary	2.64	0.98	171.90	0.24	174.58	0.05	168.48	0.06	175.27	0.07	165.73	0.15	96
▶ Loop 8754	binary	1.57	5.37	47.72	0.03	4.35	0.01	5.65	0.06	9.09	0.01	4.84	0.01	1489
▶ Loop 12711	binary	1.41	1.00	9.86	0.16	10.68	0.14	10.99	0.25	9.88	0.15	10.38	0.13	384
▼ Loop 18501	binary	1.22	4.40	325.64	0.08	79.09	0.01	248.73	0.19	74.09	0.02	63.73	0.00	22
▼ Bucket 9		98.86	4.40	325.64	0.08	79.09	0.01	248.73	0.19	74.09	0.02	63.73	0.00	22
▼ Bucket 10		1.08	6.56	488.18	0.53	79.00	0.01	435.73	0.63	74.36	0.01	64.27	0.01	22
▶ Loop 12729	binary	1.12	1.04	8.94	0.19	10.36	0.08	8.47	0.18	8.61	0.22	8.91	0.12	384
▶ Loop 12688	binary	1.09	2.28	31.92	0.24	33.17	0.06	35.67	0.92	14.00	0.14	11.00	0.07	24
▶ Loop 8912	binary	0.97	5.01	49.52	0.07	5.75	0.03	6.16	0.10	9.88	0.05	6.23	0.04	128
▶ Loop 26800	binary	0.85	0.83	119.00	0.14	122.50	0.38	106.00	0.21	143.25	0.13	107.00	0.24	8
▶ Loop 20240	binary	0.78	1.28	10.60	0.21	10.00	0.05	17.50	0.04	8.30	0.03	6.05	0.02	384
▶ Loop 8755	binary	0.47	5.02	52.47	0.13	4.92	0.04	4.20	0.02	9.46	0.00	4.16	0.01	372

Fig. 9 Extended Loop Expert Summary for QMCPACK. Values highlighted in red signal a highly unreliable result (execution time below 250 cycles), orange a weakly reliable result (time between 250 and 500), and not highlighted a reliable result (time above 500). For each variant, the columns STA contain the stability metric of the results (lower is better).

6.4 Scalability results

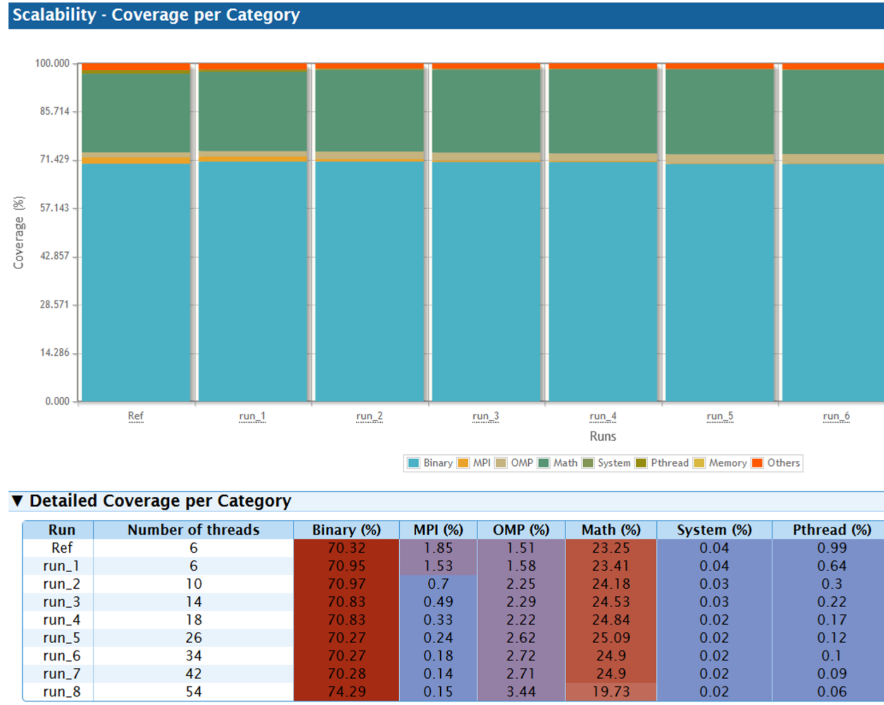
This view presents the metrics related to the application scalability. The main metrics computed during a scalability run are the speedup and efficiency (as described in 5.2). In the case of a weak scaling application, the efficiency does not take into account the number of threads.

▼ Scalability Runs Description

Run run_1	NB processes: 2	NB threads: 2	NB nodes: 1	NB tasks per node: 1
Run run_2	NB processes: 2	NB threads: 4	NB nodes: 1	NB tasks per node: 1
Run run_3	NB processes: 2	NB threads: 6	NB nodes: 1	NB tasks per node: 1
Run run_4	NB processes: 2	NB threads: 8	NB nodes: 1	NB tasks per node: 1
Run run_5	NB processes: 2	NB threads: 12	NB nodes: 1	NB tasks per node: 1
Run run_6	NB processes: 2	NB threads: 16	NB nodes: 1	NB tasks per node: 1
Run run_7	NB processes: 2	NB threads: 20	NB nodes: 1	NB tasks per node: 1
Run run_8	NB processes: 2	NB threads: 26	NB nodes: 1	NB tasks per node: 1

Fig. 10 Weak Scalability Runs Description for QMCPACK. This presents the various parameters used for each run, such as the number of processes, thread, nodes, ...

Figure 10 presents a description of the scalability runs, including the various parameters varying from one run to another. Figure 11 displays the coverage of the various code categories of code (such as MPI, OpenMP, memory handling, or the application itself) across the different runs involved in the scalability analysis.



Run	Number of threads	Binary (%)	MPI (%)	OMP (%)	Math (%)	System (%)	Pthread (%)
Ref	6	70.32	1.85	1.51	23.25	0.04	0.99
run_1	6	70.95	1.53	1.58	23.41	0.04	0.64
run_2	10	70.97	0.7	2.25	24.18	0.03	0.3
run_3	14	70.83	0.49	2.29	24.53	0.03	0.22
run_4	18	70.83	0.33	2.22	24.84	0.02	0.17
run_5	26	70.27	0.24	2.62	25.09	0.02	0.12
run_6	34	70.27	0.18	2.72	24.9	0.02	0.1
run_7	42	70.28	0.14	2.71	24.9	0.02	0.09
run_8	54	74.29	0.15	3.44	19.73	0.02	0.06

Fig. 11 Weak Scalability Coverage by Category for QMCPACK. Coverages are expressed in percentages. The x-axis references the runs by the names used in figure 10

Figure 12 displays the efficiencies of the hottest application loops. Since this is a weak scalability application, most of them are close to 1.

7 Application to QMCPACK

The vectorization of QMCPACK was already quite satisfactory: CQA analyses showed that only a 8% speedup at most could be expected if achieving full vectorization on all loops (as shown in figures 5 and 2). However, it was possible to obtain significant speedups by focusing on other performance issues.

One such issue was the detection by CQA of a large number of paths in a few loops. These loops were perfectly vectorized but the compiler generated a very complex control flow around the vector instructions. The source code contained a loop nest (7 iterations) annotated with a full unroll directive, ignored by the compiler. This was fixed by fully unrolling by hand the problematic loop nest, yielding a speedup between 7 and 9% at application level.

Loop id	Source Location	Source Function	Coverage (%)	(Ref) Efficiency	(run_1) Efficiency	(run_2) Efficiency	(run_3) Efficiency	(run_4) Efficiency	(run_5) Efficiency	(run_6) Efficiency	(run_7) Efficiency	(run_8) Efficiency
18403	qmcpack:MultiBsplineValue.hpp:56-57	qmcplusplus::BsplineSet >::evaluate	16.64	1	1.01	1.02	1.01	1.01	1.01	1.01	0.98	0.37
26027	qmcpack:cmath:261-464	qmcplusplus::SoaDistanceTableAA::moveOnSphere	10.32	1	1.01	1	0.99	0.99	0.99	0.99	0.99	0.94
18424	qmcpack:MultiBsplineVGLH.hpp:187-207	qmcplusplus::BsplineSet >::evaluate	8.03	1	1.02	1.01	0.99	0.99	0.99	1.02	1.05	0.85
26026	qmcpack:cmath:261-464	qmcplusplus::SoaDistanceTableAA::evaluate	4.79	1	1.01	0.99	1	0.99	0.99	0.99	0.99	0.95
26028	qmcpack:cmath:261-464	qmcplusplus::SoaDistanceTableAA::move	4.73	1	1.01	1	0.99	0.99	0.99	0.99	0.99	0.95
18474	qmcpack:MultiBsplineVGLH.hpp:187-207	qmcplusplus::BsplineSet >::evaluate_notranspose	3.74	1	1.02	1.01	1.01	1.02	1.02	1.02	1	0.65
18501	qmcpack:SplineC2RAdoptor.h:325-373	void qmcplusplus::SplineC2RSoA::assign_vgl >, qmcplusplus::Vector, std::allocator > > >	1.37	1	1.01	0.98	0.97	0.97	0.97	0.97	0.98	0.91
30512	qmcpack::	__intel_avx_rep_memset	1.33	1	0.99	0.96	0.94	0.93	0.94	0.98	0.99	0.89
8754	qmcpack:CoulombPBCAA.cpp:425-427	qmcplusplus::CoulombPBCAA::evalSR	1.26	1	1.03	1.02	1.01	1.01	1.01	1.01	1	0.97
12711	qmcpack:BsplineFunctor.h:690-695	qmcplusplus::J2OrbitalSoA >::ratioGrad	0.99	1	1.02	1.01	1	1	1	1	1	0.96
12688	qmcpack:BsplineFunctor.h:639-643	qmcplusplus::J2OrbitalSoA >::ratio	0.94	1	1.01	1.01	0.99	0.98	0.99	0.99	0.98	0.94
	qmcpack:Cou											

Fig. 12 Loop weak scalability report for QMCPACK. The runs are referenced by the names used in figure 10. Efficiency values are highlighted from green (satisfactory) to red (can be improved).

Another issue detected by CQA was a loop containing a large number of stack accesses, unbalanced port usage due to the presence of "special" instructions, and partial vectorization. This was due to a large loop body that overwhelmed compiler optimization capacities. This was addressed by splitting the loop in order to reduce its complexity to a level manageable by the compiler, yielding a speedup of 1% at application level.

It was also possible to detect from DECAN analyses that reducing L1 traffic held a strong potential benefit (as seen on figure 3). This was addressed by adding for some loops a surrounding loop providing some data reuse (blocking) which could be exploited by Unroll and Merge, yielding a 20% speedup at application level.

The cumulative speedup of these optimisations reached 30% at application level.

8 Conclusion

ONE View allows automating the launching of several tools, formatting their outputs and providing the end user with aggregated views of performance metrics. In addition, **ONE View** provides detailed performance analyses of optimizations such as vectorization (full or partial) and loop blocking. Such a tool is of critical importance in the HPC world where architectures are becoming increasingly complex making the code optimization task extremely tedious.

ONE View has been successfully used to optimise industrial and academic applications such as Yales 2 or QMCPACK.

Future works will focus on following the evolution of architectures to provide up-to-date information, expanding the analysis capabilities of ONE View by adding

new modules focusing on other aspects of the performance analysis process and further increasing its usability for non performance optimisation experts.

9 Acknowledgements

This work was funded by the **CEA**, **GENCI**, **INTEL** and **UVSQ** in the framework of the Exascale Computing Research collaboration, and also by the French Ministry of Industry in the framework of PERFCLOUD, ELCI and COLOC European projects.

The authors also wish to thank D. Kuck, V. Lee, J. Kim and D. Wong from INTEL for their help with the QMCPACK application.

References

1. Oseret, E., Charif-Rubial, A, Noudohouenou, J., Jalby, W., Lartigue, G.: CQA: A code quality analyzer tool at binary level. 21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014.
2. Koliaï, S., Bendifallah, Z., Tribalat, M., Valensi, C., Acquaviva, J., Jalby, W.: Quantifying Performance Bottleneck Cost Through Differential Analysis. 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, Eugene, Oregon, USA.
3. Bendifallah, Z., Jalby, W., Noudohouenou, J., Oseret, E., Palomares, V., Charif-Rubial, A.: PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis. 7th International Workshop on Parallel Tools for High Performance Computing, September 2013, ZIH, Dresden, Germany.
4. Fog, A., https://www.agner.org/optimize/instruction_tables.pdf
5. Shende, S. S., Malony, A. D., The Tau Parallel Performance System, Int. J. High Perform. Comput. Appl.
6. Geimer M., Wolf F., Wylie B. J., Ábrahám E., Becker D., and Mohr B., “The scalasca performance toolset architecture,” Concurrency and Com- putation: Practice and Experience
7. Adhianto L., Banerjee S., Fagan M., Krentel M., Marin G., Mellor- Crummey J., and Tallent N. R., “HPCToolkit: Tools for performance anal- ysis of optimized parallel programs,” Concurrency and Computation: Practice and Experience
8. Knüpfer A., Brunst H., Doleschal J., Jurenz M., Lieber M., Mickler H., Müller M. S., Nagel W. E., “The vampir performance analysis tool- set,” Tools for High Performance Computing
9. Hollingsworth J., Buck B., An API for Runtime Code Patching, Winter 2000 Journal of High Performance Computing Applications
10. “INTEL VTune^{TM}Amplifier,” https://software.intel.com/en-us/intel-vtune-amplifier-xe.
11. Arm Forge https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-forge
12. Treibig J., Hager G., Wellein G., Likwid: A lightweight performance-oriented tool suite for x86 multicore environments, Parallel Processing Workshops (ICPPW) 2010
13. INTEL Advisor, https://software.intel.com/en-us/advisor
14. Intel Application Performance Snapshot https://software.intel.com/sites/products/snapshots/application-snapshot/
15. ARM Forge https://www.arm.com/products/development-tools/server-and-hpc/forge
16. Kim J. et al, QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids