

MAQAO Performance Analysis and Optimization Tool

Cédric VALENSI Emmanuel OSERET, Jean-Baptiste LE RESTE

cedric.valensi@uvsq.fr,

emmanuel.oseret@uvsq.fr, jean-baptiste.le-reste@uvsq.fr

Performance Evaluation Team, University of Versailles S-Q-Y

http://www.maqao.org

VI-HPS 21st Garching – Germany – 18-22 April 2016







UNIVERSITÉ D

VERSAILLES

MAQAO Framework and Tool suite

R&D Team: develop performance evaluation and optimization tools

- MAQAO main functionalities:
- Profiling and hardware counters collection
- Code quality analysis
- LGPL3 Open Source software
- Source release of core components planned for 2016
- Binary release available on demand
- Support Intel x86-64 and Xeon Phi

Partnerships:

- Funded by UVSQ, Intel and CEA (French department of energy)
- Optimize industrial and academic HPC applications (Yales2, AVBP, Polaris...)
- Provide building blocks for other tools (TAU performance tools, ATOS bullxprof, Intel AmplifierXE)







Ví-HPS

Introduction Performance analysis (1/2)

Characterize application performance

- Application profiling
- Pinpoint the performance bottlenecks
 - Complex multicore and manycore CPUs
 - Complex memory hierarchy
- Best use of the machine features

Generally a multifaceted problem

- How to determine the dominant issues?
 - Algorithms choice
 - Implementation
 - Parallelization
 - ...
- Maximizing the number of views

=> Need for dedicated and complementary tools



Ví-HPS

Introduction Performance analysis (2/2)

Motivating example: loop ~10% walltime



VI-HPS

Introduction MAQAO: Analysis at the binary level

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 Analyze Is What You Run

We are able to reconstruct the program structure

- Automatically relate the analyses to source code
 - A single source loop can be compiled as multiple assembly loops



VI-HPS

VIRTUAL INSTITUTE - HIGH PRODUCTIVITY SUPERCOMPUTING

Introduction MAQAO methodology





VIRTUAL INSTITUTE -- HIGH PRODUCTIVITY SUPERCOMPUTING

MAQAO LProf: Lightweight Profiler





Ví-HPS

MAQAO LProf: Lightweight Profiler Introduction

Lightweight localization of application hotspots

Multiple measurement methods available:

- Sampling (default)
 - Hardware counters (through perf_event_open system call)
 - Non intrusive, low overhead
- Instrumentation
 - Through binary rewriting
 - Can target specific issues
 - Extra overhead

Runtime-agnostic

Ví-HPS

V VIRTUAL INSTITUTE - HIGH PRODUCTIVITY SUPERCOMPUTING

MAQAO LProf: Lightweight Profiler Time categorization

- Parallelization overhead
- Shared: Pthreads, OpenMP, etc ...
- Distributed: MPI, etc...

Programming

- IO operations
- String operations
- Memory management
- External libraries such as libm / libmkl
- User time breakdown
- Functions
- Loops



MAQAO LProf: Lightweight Profiler Function and loop hotspots (1/3)

Focusing on user time

- Function hotspots
- Load balancing across the nodes

Hotspots - Functions				
	Name	M	edian Excl %Time	Deviation
compute_rhs_			30.88	0.14
y_solve_			15.51	0.14
z_solve_			15.34	0.14
x_solve_			15.07	0.14
MDIDL CH3L Progress			E 61	0.14

MAQAO LProf: Lightweight Profiler Function and loop hotspots (2/3)

Focusing on user time

х 20 Function hotspots Load balancing across the nodes 15 %Time Name compute_rhs_ y_solve_ 2 3 5 6 7 1 4 8 z_solve_ x_solve_ 15.07 0.14 MDIDI CHRI Drogress E 61 0.14

MAQAO LProf: Lightweight Profiler Function and loop hotspots (3/3)

Analyze the time spent at loop level

- Find the most time consuming
- Direct link to MAQAO CQA analyses

Name	Excl %Time	Excl Time (s)
binvcrhs - 206@solve_subs.f	17.27	2.2
MPIDI_CH3I_Progress	15.24	1.9
poll_active_fboxes	13.71	1.7
▼ y_solveomp_fn.0 - 45@y_solve.f	8.47	1.0
▼ loops	8.47	
Loop 121 - y_solve.f@45	0	
Loop 122 - y_solve.f@45	0.16	
 Loop 124 - y_solve.f@45 	0.14	
Loop 125 - y_solve.f@145	5.12	
Loop 126 - y_solve.f@55	2.03	
 Loop 123 - y_solve.f@45 	1.02	
x_solveomp_fn.0 - 48@x_solve.f	8.23	1.00
► loops	8.23	

dauvorgno - Procoss #14212 - Throad #14201

V VIRTUAL INSTITUTE - HIGH PRODUCTIVITY SUPERCOMPUTING

MAQAO LProf/MPI: MPI characterization





MAQAO LProf/MPI: MPI characterization Introduction (1/2)

Objective: profiling MPI runtime use.

- Coarse grain
 - Overview
 - Global trends/patterns
 - Low overhead
- Fine grain
 - Filtering precise issues
 - Higher overhead

Online profiling:

- No IOs: only one result file with pre-processed data
- Reduced memory footprint thanks to aggregated metrics
- Scalable on 1000+ MPI processes



MAQAO LProf/MPI: MPI characterization Introduction (2/2)

LProf/MPI is an MPI profiling tool targeting lightweight metrics that can be deduced online

- No trace required
- Does not require recompiling





MAQAO LProf/MPI: MPI characterization Global profile (1/3)

Summary view: MPI primitives classified by hits (calls), time and size (if applicable)





MAQAO LProf/MPI: MPI characterization Global profile (2/3)



MAQAO LProf/MPI: MPI characterization Global profile: flat view (3/3)

MPI Profile

Function	Hits	Time	Size	Walltime %
MPI_Waitall	192960	13 m 1.51 s	0 B	52.333%
MPI_Init	128	1 m 46.60 s	0 B	7.138%
MPI_Barrier	256	10.88 s	0 B	0.729%
MPI_Isend	192960	1.47 s	4.568 GB	0.098%
MPI_Reduce	384	5.36e-1 s	11.000 KB	0.036%
MPI_Irecv	192960	4.62e-1 s	4.568 GB	0.031%
MPI_Comm_split	128	4.05e-1 s	0 B	0.027%
MPI_Bcast	1152	3.12e-2 s	132.000 KB	0.002%
MPI_Finalize	128	2.07e-3 s	0 B	0.000%
MPI_Wtime	256	3.53e-4 s	0 B	0.000%
MPI_Comm_size	128	1.30e-4 s	0 B	0.000%
MPI_Comm_rank	256	4.28e-5 s	0 B	0.000%

VIRTUAL/INSTITUTE -- HIGH PRODUCTIVITY SUPERCOMPUTING

MAQAO LProf/MPI: MPI characterization Function scattering over time



MAQAO LProf/MPI: MPI characterization Probability densities: when and how long ?



V VIRTUAL INSTITUTE - HIGH PRODUCTIVITY SUPERCOMPUTING

MAQAO LProf/MPI: MPI characterization 2D communication matrix



1.196 MB

15.350 MB

MAQAO LProf/MPI: MPI characterization Per rank distribution

Checking load balancing



VI-HPS

MAQAO LProf/MPI: MPI characterization 3D Topology



MAQAO CQA: Code Quality Analyzer



Ví-HPS

MAQAO CQA: Code Quality Analyzer Introduction

Improving performance of the user code

Focusing on loopsIn HPC most of the time is spent in loops

Static analysis of assembly code

- Evaluate the quality of the compiler generated code
- Analysis based on a microarchitecture model
- Returns hints and workarounds to the developer

Targets compute bound codes

VI-HPS

MAQAO CQA: Code Quality Analyzer **Output**

High level reports

- Reference to the source code
- Bottleneck description
- Hints to improve performance
- Reports categorized by confidence level:
 - gain, potential gain

Low level report for performance experts

No runtime cost/overhead

Source	loop	ending	at	line	10
oouroc	oop.	entaning	***	me	

MAQAO binary loop id: 2

2 The loop is defined in /zhome/academic/HLRS/xhp/xhpeo/TEST/matmul/kernel.c:9-10

2% of peak computational performance is used (0.67 out of 32.00 FLOP per cycle (1.67 GFLOPS @ 2.50GHz))

Potential gain Hints Experts only Gain

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization. By fully vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.38 cycles (8.00x speedup).

Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:

- Try another compiler or update/tune your current one:
- 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:
- C storage order is row-major: for(i) for(j) a[i][i] = b[i][i]; (slow, non stride 1) => for(i) for(j) a[i][i] = b[i][i]; (fast, stride 1)
- * If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
- for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

MAQAO CQA: Code Quality Analyzer Processor Architecture: Core level

Most of the time applications only exploit at best 5% to 10% of the peak performance

Concepts:

- Peak performance
- Execution pipeline
- Resources/Functional units

Key performance levers for core level efficiency:

- Vectorization
- Get rid of high latency instructions if possible
- Have the compiler generate an efficient code

Same instruction – Same cost



MAQAO CQA: Code Quality Analyzer Compiler and programmer hints

Compiler can be driven using flags and pragmas

- Architecture specific flags (e.g. -xHost on AVX capable machines)
- Force optimization (unrolling, vectorization, alignment, ...)
- Bypass conservative behavior when possible (e.g. 1/X precision)

Implementation changes

- Data access
 - Loop interchange
 - Changing loop strides
- Avoid instructions with high latency

VI-HPS

MAQAO CQA: Code Quality Analyzer GUI sample (1/2)

ource loop ending at line 682	And I I I I I I I I I I I I I I I I I I I
MAQAO binary loop id: 238	
of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz)) in Potential gain Hints Experts only	
Vectorization	
our loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization. In fully vectorizing your loop, you can lower the cost of an iteration from 190.00 to 60.75 cycles (3.13x speedup).	
roposed solution(s):	
wo propositions:	



MAQAO CQA: Code Quality Analyzer GUI sample (2/2)

M.	AmaAO
	Code quality analysis
Sour	ce loop ending at line 682
MAG	QAO binary loop id: 238
he loo	p is defined in MPI/BT/x_solve.f:519-682
5% of	peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))
Gain	Potential gain Hints Experts only
	Type of elements and instruction set
234 S	SE or AVX instructions are processing arithmetic or math operations on double precision FP elements in scalar mode (one at a time).
	Vectorization status
Your I vecto Only 2	loop is probably not vectorized (store and arithmetical SSE/AVX instructions are used in scalar mode and, for others, at least one is in ir mode). 28% of vector length is used.
	Matching between your loop (in the source code) and the binary loop
The b - 95: 4 - 139: The b The b	inary loop is composed of 234 FP arithmetical operations: addition or subtraction : multiply binary loop is loading 1600 bytes (200 double precision FP elements). binary loop is storing 616 bytes (77 double precision FP elements).
	Arithmetic intensity



Thank you for your attention !

Questions ?

