VI-HPS

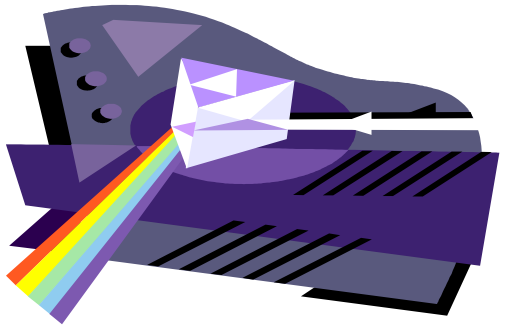
MAQAO

Performance Analysis and Optimization Tool

Andres S. CHARIF-RUBIAL
Emmanuel OSERET
{andres.charif,emmanuel.oseret}@uvsq.fr

Performance Analysis Team, University of Versailles
http://www.maqao.org

- Understand the performance of an application
  - How well it behaves on a given machine
- What are the issues ?
- Generally a multifaceted problem
  - Maximizing the number of views = better understand
- Use techniques and tools to understand issues
- Once understood ⇨ Optimize application

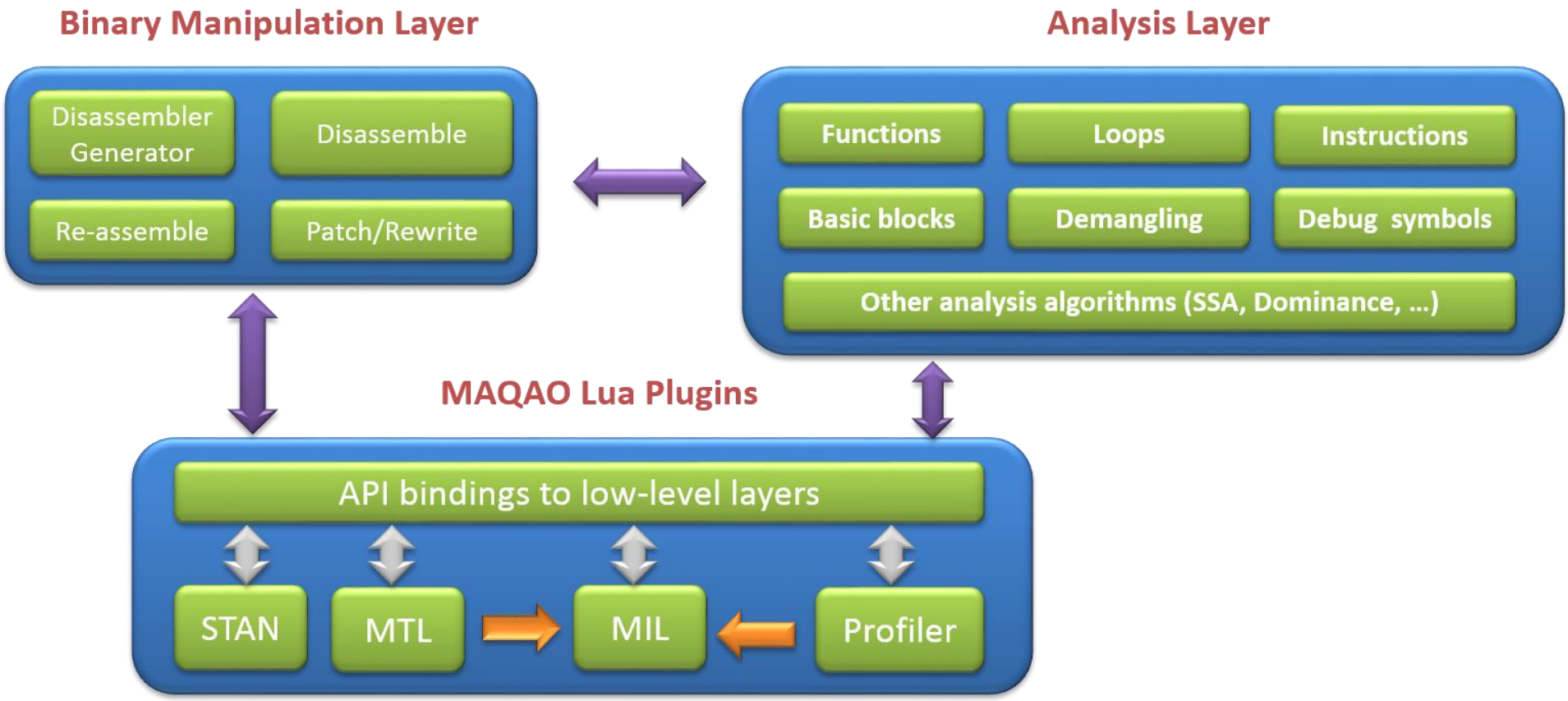


- Compiler remains your best friend
  - Be sure to select proper flags (e.g., -xavx)
  - Pragmas: Unrolling, Vector alignment
  - O2 V.S. O3
  - Vectorisation/optimisation report

- Open source (LGPL 3.0)
  - Currently binary release
  - Source release by mid December
- Available for x86-64 and Xeon Phi
  - Looking forward in porting MAQAO on BlueGene

- Easy install
  - Packaging : ONE (static) standalone binary
    - Easy to embeed
- Audience
  - User/Tool developer: analysis and optimisation tool
  - Performance tool developer: framework services
    - TAU: tau_rewrite (MIL)
    - ScoreP: on-going effort (MIL)

Introduction
MAQAO Tool



- Scripting language
  - Lua language : simplicity and productivity
  - Fast prototyping
  - MAQAO Lua API : Access to services

- Built on top of the Framework
- Loop-centric approach
- Produce reports
  - We deal with low level details
  - You get high level reports

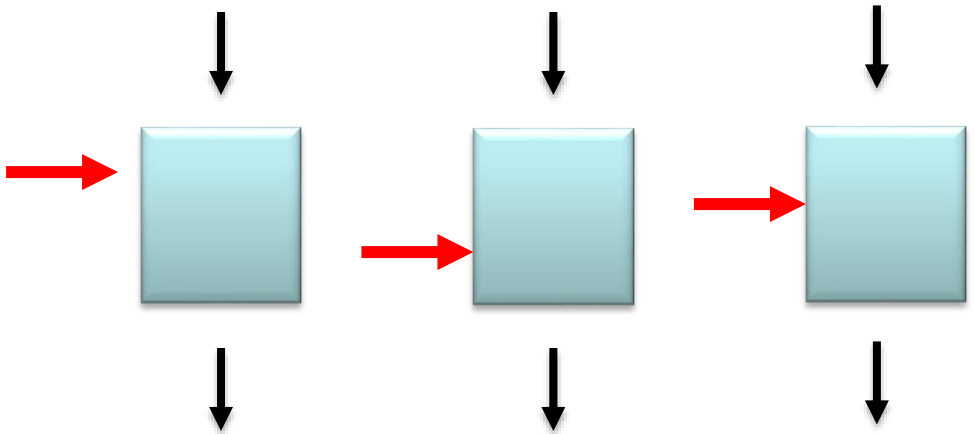
Outline

- Introduction
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules

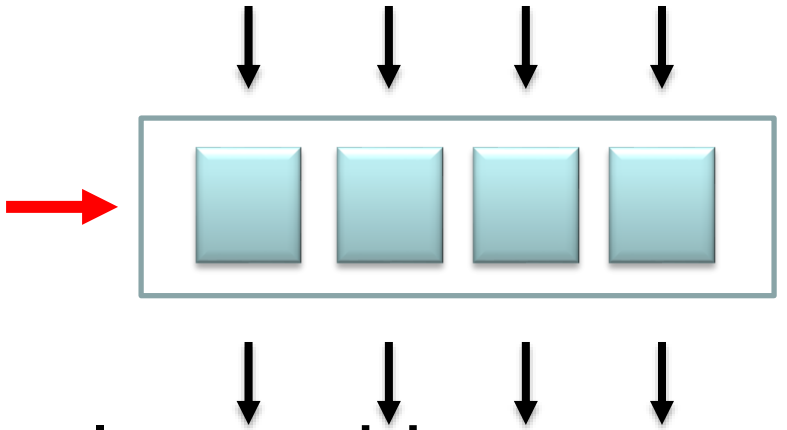
- MAQAO Profiling
  - Instrumentation
    - Through binary rewriting
    - High overhead / More precision
  - Sampling
    - Hardware counter through perf_event_open system call
    - Very low overhead / less details

Pinpointing hotspots
Parallelism level

- SPMD
  - Program level



- SIMD
  - Instruction level

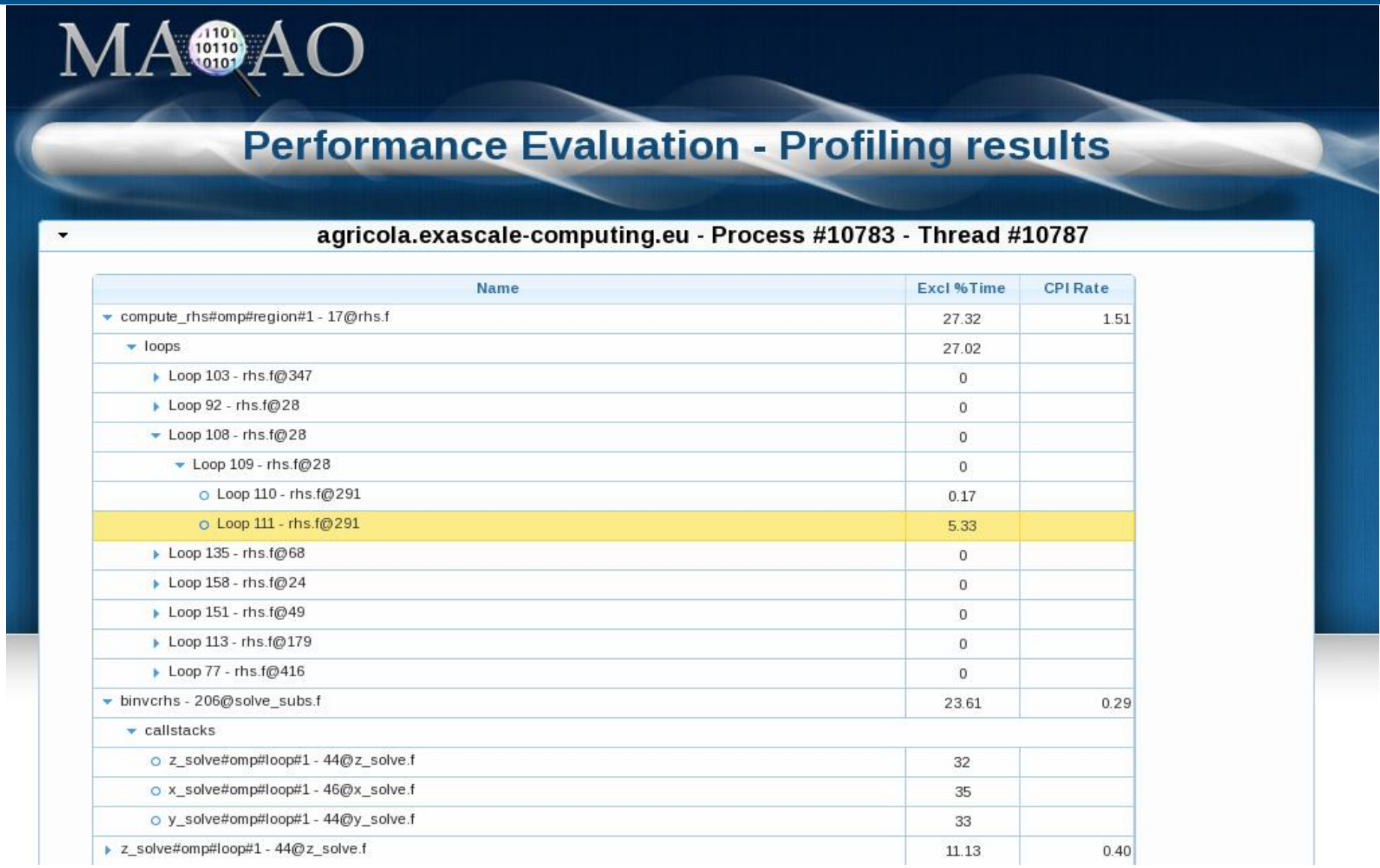


- By default MAQAO only considers system processes and threads

Pinpointing hotspots
Parallelism level

- Display functions and their exclusive time
  - Associated callchains and their contribution
  - Loops
- Innermost loops can then be analyzed by the code quality analyzer module (CQA)
- Command line and GUI (HTML) outputs

Pinpointing hotspots
GUI snapshot



Name	Excl %Time	CPI Rate
compute_rhs#omp#region#1 - 17@rhs.f	27.32	1.51
loops	27.02	
Loop 103 - rhs.f@347	0	
Loop 92 - rhs.f@28	0	
Loop 108 - rhs.f@28	0	
Loop 109 - rhs.f@28	0	
Loop 110 - rhs.f@291	0.17	
Loop 111 - rhs.f@291	5.33	
Loop 135 - rhs.f@68	0	
Loop 158 - rhs.f@24	0	
Loop 151 - rhs.f@49	0	
Loop 113 - rhs.f@179	0	
Loop 77 - rhs.f@416	0	
binvcrhs - 206@solve_subs.f	23.61	0.29
callstacks		
z_solve#omp#loop#1 - 44@z_solve.f	32	
x_solve#omp#loop#1 - 46@x_solve.f	35	
y_solve#omp#loop#1 - 44@y_solve.f	33	
z_solve#omp#loop#1 - 44@z_solve.f	11.13	0.40

Outline

- Introduction
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules

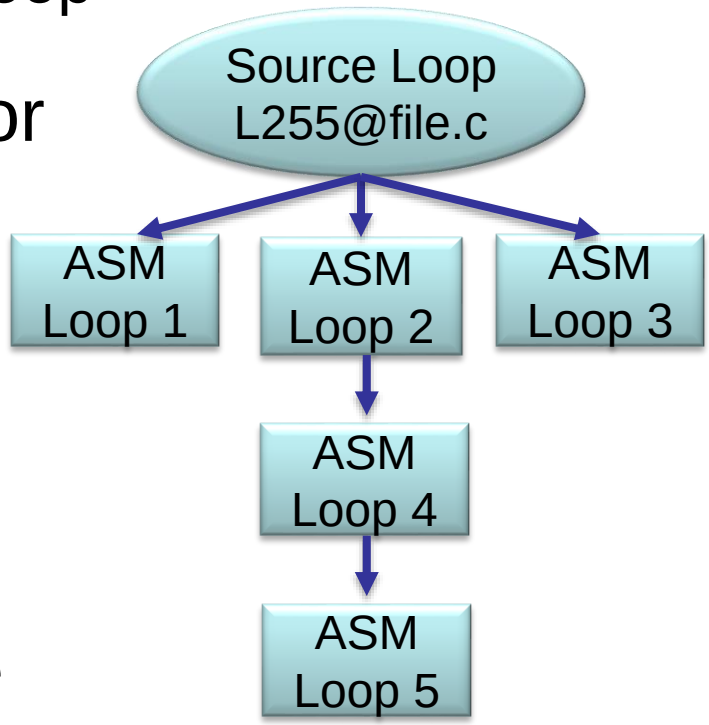
Static performance modeling
Introduction

- Main performance issues:
  - Core level
  - Multicore interactions
  - Communications

- Most of the time core level is forgotten

Static performance modeling
Goals

- Static performance model
  - Targets innermost loops
    - source loop V.S. assembly loop
  - Take into account processor (micro)architecture
  - Assess code quality
    - Estimate performance
    - Degree of vectorization
    - Impact on micro architecture



Static performance modeling
Model

- Simulates the target (micro)architecture
  - Instructions description (latency, uops dispatch...)
  - Machine model
- For a given binary and micro-architecture, provides
  - Quality metrics (how well the binary is fitted to the micro architecture)
  - Static performance (lower bounds on cycles)
  - Hints and workarounds to improve static performance

Static performance modeling
Metrics

- Vectorization (ratio and speedup)
  - Allows to predict vectorization (if possible) speedup and increase vectorization ratio if it's worth
- High latency instructions (division/square root)
  - Allows to use less precise but faster instructions like RCP (1/x) and RSQRT (1/sqrt(x))
- Unrolling (unroll factor detection)
  - Allows to statically predict performance for different unroll factors (find main loops)

Static performance modeling
Report example

Pathological cases

Your loop is processing FP elements but is **NOT OR PARTIALLY VECTORIZED.**
Since your execution units are vector units, only a fully vectorized loop can use their full power.
By fully vectorizing your loop, you can lower the cost of an iteration from 14.00 to 3.50 cycles (4.00x speedup).
Two propositions:
- Try another compiler or update/tune your current one:
  * **gcc: use O3 or Ofast.** If targeting IA32, add mfpmath=sse combined with march=<cputype>, msse or msse2.
  * icc: 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.

WARNING: Fix as many pathological cases as you can before reading the following sections.

Bottlenecks

The divide/square root unit is a bottleneck. Try to reduce the number of division or square root instructions.
If you accept to loose numerical precision, you can speedup your code by passing the following options to your compiler:
gcc: (ffast-math or Ofast) and mrecip
icc: this should be automatically done by default

By removing all these bottlenecks, you can lower the cost of an iteration from 14.00 to 1.50 cycles (9.33x speedup).

Outline

- Introduction
- Pinpointing hotspots
- Code quality analysis
- Upcoming modules

Ongoing work

- Dynamic bottleneck analyzer
  - Differential analysis
- Memory characterization tool
  - Access patterns
  - Data reshaping
  - Cache simulator
- Value profiler
  - Function specialization / memorizing
  - Loops instances (iteration count) variations

Thanks for your attention !

Questions ?