MAQAO
Performance Analysis and Optimization Tool

Andres S. CHARIF-RUBIAL
andres.charif@uvsq.fr
Performance Evaluation Team, University of Versailles S-Q-Y
http://www.maqao.org
VI-HPS 17th Stuttgart – 23/27 February 2015
MAQAO Framework and Toolsuite

R&D Team: develop performance evaluation and optimization tools

Open Source software (LGPL 3)
- Currently only binary release (source => ongoing)
- Profilers (generic and MPI) work on any LSB/Most Unix
- Code quality analysis and hardware counters support only available for Intel x86-64 and Xeon Phi

Funded by UVSQ, Intel and CEA (French department of energy)

Establish partnerships:
- Optimize industrial applications
- Provide building blocks (framework services) to other tools:
  - TAU tool tau_rewrite: binary rewriting feature (MIL)
  - ATOS/BULL tool bullxprof: binary rewriting feature (MIL)
Introduction

Performance analysis (1/2)

- Characterize the performance of an application
  - Complex multicore CPUs and memory systems
  - How well does it behave on a given machine

- Generally a multifaceted problem
  - What are the issues (numerous but finite)?
  - Which one(s) dominates?
  - Maximizing the number of views
  - => Need for specialized tools

- Three main classes of issues
  - Find/Select relevant algorithms
  - Work sharing/decomposition
  - Exploiting performance available at CPU level
Introduction

Performance analysis (2/2)

Motivating example: loop ~10% walltime

Source code and associated issues:

1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Vector vs Scalar
Introduction

MAQAO: working at binary level (1/2)

Why ???

Most of the time the compiler changes source code

Some source code instrumentation may prevent the compiler from applying transformation
  • i.e.: loop interchange

We want to evaluate the “real” executed code

We are able to reconstruct an abstract vue with functions and loops in order to be able to correlate with your source code.

One little difference is understanding loops at assembly level
Introduction
MAQAO: working at binary level (2/2)

Source level V.S. Assembly level

You just need to understand the difference
But our tools’ reports always point to source code
MAQAO Perf: locating hotspots
MAQAO Perf: locating hotspots

Introduction

Locating most time consuming hotspots is the first step you want to accomplish.

Multiple measurement methods available:

- **Why is it important to know this?**
- **Instrumentation**
  - Through binary rewriting
  - High overhead / More precision
- **Sampling**
  - Hardware counters (through perf_event_open system call)
  - Linux kernel timers
  - No instrumentation / Very low overhead / less details (i.e. function calls count)
- **Default method:** Sampling using hardware counters (if available) or timers

Runtime-agnostic: Only system processes and threads are considered

Where is time spent? Which one(s) should I investigate first?
MAQAO Perf: locating hotspots

*Time categorization*

Sadly, executing an application is not just doing the science you are supposed to!

**Work sharing/splitting**
- Shared: Pthreads, OpenMP, etc ...
- Distributed: MPI, etc...

**Programming**
- IO
- String manipulation
- Memory management
- Math (external libraries)

**Doing actual science (Application)**
- Functions
- Loops
MAQAO Perf: locating hotspots

*Function and loop hotspots (1/3)*

Lets focus on science!

First we want to check function hotspots load balancing vue at (multi)node level
- For the same function
- Does it behave the same way on all the nodes?

<table>
<thead>
<tr>
<th>Name</th>
<th>Median Excl %Time</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_rhs_</td>
<td>30.88</td>
<td>0.14</td>
</tr>
<tr>
<td>y_solve_</td>
<td>15.51</td>
<td>0.14</td>
</tr>
<tr>
<td>z_solve_</td>
<td>15.34</td>
<td>0.14</td>
</tr>
<tr>
<td>x_solve_</td>
<td>15.07</td>
<td>0.14</td>
</tr>
<tr>
<td>MPI+CH31 Progress</td>
<td>5.61</td>
<td>0.14</td>
</tr>
</tbody>
</table>
MAQAO Perf: locating hotspots

*Function and loop hotspots (2/3)*

<table>
<thead>
<tr>
<th>Function</th>
<th>%Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_solve</td>
<td>15.51</td>
</tr>
<tr>
<td>z_solve</td>
<td>15.34</td>
</tr>
</tbody>
</table>
MAQAO Perf: locating hotspots
*Function and loop hotspots (3/3)*

Then analyse time spent in loops:
- Time spent in loop w.r.t. function

- Use MAQAO CQA tool to analyse loops of interest
MAQAO Perf/MPI: MPI characterization
MAQAO Perf/MPI: MPI characterization

Introduction (1/2)

The previous profiler module only provided a global figure about time spent in the MPI runtime (X%).

We want the same kind of insight but dealing with MPI primitives.

Our methodology:
- Coarse grain: overview, global trends/patterns => cheapest possible cost/overhead
- Fine grain: filtering precise issues => accept to pay higher cost/overhead if worth

Online profiling:
- No traces to void IO wall: no IOs (only one result file with pre-processed data)
- Avoid memory: reduced memory footprint thanks to aggregated metrics
- Scalable on 1000+ MPI processes
MAQAO Perf/MPI: MPI characterization
Introduction (2/2)

Summary: Perf/MPI is a simple MPI profiling tool targeting lightweight metrics which can be reduced online (no trace required).

APPLICATION

MAQAO

profile.js

In-browser Visualizer

Does not require recompiling
MAQAO Perf/MPI: MPI characterization

Global profile (1/3)

Summary vue: MPI primitives classified by hits (calls), time and size (if applicable)
MAQAO Perf/MPI: MPI characterization

Global profile (2/3)

---

**MPI Time Pie Chart**

- MPI_Init
- MPI_Finalize
- MPI_Bcast
- MPI_Comm_split
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Barrier
- MPI_Wtime
- MPI_Waitall

**MPI Size Pie Chart**

- MPI_Recv
- MPI_Recv
- MPI_Bcast

---
MAQAO Perf/MPI: MPI characterization
*Global profile: flat vue (3/3)*

### MPI Profile

<table>
<thead>
<tr>
<th>Function</th>
<th>Hits</th>
<th>Time</th>
<th>Size</th>
<th>Walltime %</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Waitall</td>
<td>192960</td>
<td>13 m 1.51 s</td>
<td>0 B</td>
<td>52.333%</td>
</tr>
<tr>
<td>MPI_Init</td>
<td>128</td>
<td>1 m 46.60 s</td>
<td>0 B</td>
<td>7.138%</td>
</tr>
<tr>
<td>MPI_BARRIER</td>
<td>256</td>
<td>10.88 s</td>
<td>0 B</td>
<td>0.729%</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>192960</td>
<td>1.47 s</td>
<td>4.568 GB</td>
<td>0.098%</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>384</td>
<td>5.36e-1 s</td>
<td>11.000 KB</td>
<td>0.036%</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>192960</td>
<td>4.62e-1 s</td>
<td>4.568 GB</td>
<td>0.031%</td>
</tr>
<tr>
<td>MPI_Comm_split</td>
<td>128</td>
<td>4.05e-1 s</td>
<td>0 B</td>
<td>0.027%</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>1152</td>
<td>3.12e-2 s</td>
<td>132.000 KB</td>
<td>0.002%</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>128</td>
<td>2.07e-3 s</td>
<td>0 B</td>
<td>0.000%</td>
</tr>
<tr>
<td>MPI_Wtime</td>
<td>256</td>
<td>3.53e-4 s</td>
<td>0 B</td>
<td>0.000%</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>128</td>
<td>1.30e-4 s</td>
<td>0 B</td>
<td>0.000%</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>256</td>
<td>4.28e-5 s</td>
<td>0 B</td>
<td>0.000%</td>
</tr>
</tbody>
</table>
MAQAO Perf/MPI: MPI characterization

*Function scattering over time*
MAQAO Perf/MPI: MPI characterization

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

2D communication matrix

Hit, time, size

Communication Matrix

1.196 MB

15.350 MB
MAQAO Perf/MPI: MPI characterization
Per rank distribution

Hit, time, size

Check load balancing
MAQAO Perf/MPI: MPI characterization
3D Topology
MAQAO CQA: Analysing the code quality of your loops

**Code quality analysis**

- Source loop ending at line 682
- MAQAO binary loop id: 238

The loop is defined in MPB/Tk1 solem.5551-882
15% of peak computational performance is used (1.23 out of 9.00 FLOP per cycle (GFLOPS @ 1.00G)

<table>
<thead>
<tr>
<th>Case</th>
<th>Potential gain</th>
<th>Hits</th>
<th>Experts only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Vectorization**

Your loop is processing 176 elements but is NOT ENTIRELY VECTORIZED and could benefit from full vectorization.
By fully vectorizing your loop, you can lower the cost of an iteration from 193.99 to 90.75 cycles (1.3x speedup).
Since your execution units are vector units, only a fully vectorized loop can use their full power.
Proposed vectorization:
- Two propositions:
  - Try another computer or upgrade/disassemble your current one
  - Measure instructions dependencies from your loop and make it work

**Bottlenecks**

By removing all these bottlenecks, you can lower the cost of an iteration from 193.99 to 40.60 cycles (4.0x speedup).
MAQAO CQA: Analysing the code quality of your loops

Introduction

Main performance issues:
- Work sharing / communications / multicore interactions
- Core level

Most of the time core level is forgotten! But that’s where science is computed

CQA works at (assembly) loop level:
- In HPC most of the time is spent in loops (V.S. functions)
- Assess the quality of code generated by the compiler
- Take into account processor’s (micro)architecture via simulation
- Hints and workarounds to improve static performance

Compute bound:
- This tool is not meant for optimizing memory issues
- It assumes that you have fixed them
MAQAO CQA: Analysing the code quality of your loops
Goal: how will it help you?

Produce reports:
- We deal with low level details (assembly, microarchitecture details)
- You get high level reports

Provide high level reports:
- Provide source loop context when available (-g or equivalent)
- Describing a pathology/bottleneck
- Suggesting workarounds to improve static performance
- Reports categorized by confidence level:
  - gain, potential gain, hint and expert

No runtime cost/overhead:
- Your don’t need to execute your app
- Static analysis
MAQAO CQA: Analysing the code quality of your loops

Processor Architecture: Core level

Maybe you want an efficient code that gets the best out of available computing resources?

Concepts:
- Peak performance, TOP500/LINPACK
- Execution pipeline
- Resources/Functional units

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

Key performance levers:
- Vectorization
- Get rid of high latency instructions if possible
- Make the compiler generated an efficient code

<table>
<thead>
<tr>
<th>Same instruction – Same cost</th>
<th>Process up to 8X (SP) data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MAQAO PERFORMANCE ANALYSIS AND OPTIMIZATION TOOL
MAQAO CQA: Analysing the code quality of your loops

*The compiler*

Compiler remains our best friend

Be sure to select proper flags
- Know default flags (e.g., `-xHost` on AVX capable machines)
- Bypass conservative behavior when possible (e.g., 1/X precision)

Pragmas:
- Vectorization, Alignement, Unrolling, etc...
- Portable transformations
MAQAO CQA: Analysing the code quality of your loops

GUI sample (1/2)

Code quality analysis

Source loop ending at line 682

MAQAO binary loop id: 238

The loop is defined in MPI/BTix_solve.f:519-682
15% of peak computational performance is used (1.23 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain Potential gain Hints Experts only

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 190.00 to 50.75 cycles (3.13x 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 dependencies from your loop and make it unit-stride.

Bottlenecks
By removing all these bottlenecks, you can lower the cost of an iteration from 190.00 to 143.00 cycles (1.33x speedup).

Source loop ending at line 734
MAQAO CQA: Analysing the code quality of your loops

GUI sample (2/2)
Thank you for your attention

Questions