HPC SOFTWARE - TOOLS
Debugger and Performance Analysis

23.11.2018 | MICHAEL KNOBLOCH
OUTLINE

• Local module setup
• Compilers
• Libraries

Debugger:
• TotalView / DDT
• MUST
• Intel Inspector

Performance Tools:
• Score-P
• Scalasca
• Vampir
• Intel Vtune Amplifier
• Intel Advisor
• Performance Reports
• TAU
• NVIDIA Visual Profiler
• Darshan
• PAPI

Make it work, make it right, make it fast.

Kent Beck
MODULE SETUP & COMPILER
THE MODULE SETUP

• Tools are available through “modules”
  • Allows to easily manage different versions of programs
  • Works by dynamic modification of a user's environment

• Module setup based on EasyBuild and lmod
  • Staged, hierarchical setup
  • Automatically manages dependencies via toolchains

• Consistent setup on JURECA (cluster & booster) and JEWELS
MOST IMPORTANT MODULE COMMANDS

module
• spider # show all products
• spider product # show product details
• avail # show all available products
• list # list loaded products

• load product(s) # setup access to product
• unload product(s) # release access
• swap product1 product2 # replace v1 of product with v2

• whatis product(s) # print short description
• help product(s) # print longer description
• show product(s) # show what “settings” are performed
COMPILER AND MPI LIBRARIES

• Compiler
  • Intel C/C++ and Fortran compiler
  • GNU C/C++ and Fortran compiler
  • PGI C/C++ and Fortran compiler
  • Clang C/C++ compiler
  • NVIDIA CUDA compiler

• MPI libraries
  • Intel MPI
  • Parastation MPI
  • MVAPICH MPI (CUDA aware)
  • Open MPI
DEBUGGING TOOLS (STATUS: NOV 2018)

• Debugger:
  • TotalView
  • DDT

• Correctness Checker:
  • Intel Inspector
  • MUST
UNIX Symbolic Debugger
for C, C++, F77, F90, PGI HPF, assembler programs

“Standard” debugger

Special, non-traditional features
- Multi-process and multi-threaded
- C++ support (templates, inheritance, inline functions)
- F90 support (user types, pointers, modules)
- 1D + 2D Array Data visualization
- Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, pthreads)
- Scripting and batch debugging
- Memory Debugging
- CUDA and OpenACC support

http://www.roguewave.com

NOTE: License limited to 2048 processes (shared between all users)
TOTALVIEW: MAIN WINDOW

Toolbar for common options

Local variables for selected stack frame

Source code window

Stack trace

Break points
TOTALVIEW: TOOLS MENU

- Call Graph
- Data visualization
- Message queue graph
UNIX Graphical Debugger for C, C++, F77, F90 programs

Modern, easy-to-use debugger

Special, non-traditional features
- Multi-process and multi-threaded
- 1D + 2D array data visualization
- Support for MPI parallel debugging (automatic attach, message queues)
- Support for OpenMP (Version 2.x and later)
- Support for CUDA and OpenACC
- Job submission from within debugger

http://www.allinea.com

NOTE: License limited to 64 processes (shared between all users)
DDT: NON-STANDARD FEATURES

- Multi-Dimensional Array Viewer
- Message queue graph
- Memory Usage
INTEL INSPECTOR

- Detects memory and threading errors
  - Memory leaks, corruption and illegal accesses
  - Data races and deadlocks

- Dynamic instrumentation requiring no recompilation

- Supports C/C++ and Fortran as well as third party libraries

- Multi-level analysis to adjust overhead and analysis capabilities

- API to limit analysis range to eliminate false positives and speed-up analysis
INTEL INSPECTOR: GUI
Next generation MPI correctness and portability checker

http://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

MUST reports
- Errors: violations of the MPI-standard
- Warnings: unusual behavior or possible problems
- Notes: harmless but remarkable behavior
- Further: potential deadlock detection

Usage
- Relink application with mustc, mustcxx, mustf90, ...
- Run application under the control of mustrun (requires one additional MPI process)
- See MUST_Output.html report
### MUST DATATYPE MISMATCH

<table>
<thead>
<tr>
<th>Rank</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error</td>
<td>A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous) <a href="MPI_INT">0</a> in the send type and at (MPI_BYTE) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation of this situation is available in a detailed type mismatch view (MUST_Output-files/MUST_Typemismatch_0.html). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for C, commited at reference 4, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)} (Information on receive of count 8 with type:MPI_BYTE)</td>
<td></td>
<td>reference 1 rank 0: MPI_Sendrecv called from: #0 <a href="mailto:main@example.c">main@example.c</a>:33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 2 rank 1: MPI_Sendrecv called from: #0 <a href="mailto:main@example.c">main@example.c</a>:33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 3 rank 0: MPI_Type_contiguous called from: #0 <a href="mailto:main@example.c">main@example.c</a>:29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 4 rank 0: MPI_Type_commit called from: #0 <a href="mailto:main@example.c">main@example.c</a>:30</td>
</tr>
</tbody>
</table>

### Message

The application issued a set of MPI calls that mismatch in type signatures! The graph below shows details on this situation. The first differing item of each involved communication request is highlighted.

### Datatype Graph

```
  MPI_Sendrecv:send
   ↓
  MPI_Type_contiguous(count=2)
   ↓
  [0]
  ↓
  MPI_INT   MPI_BYTE
  ↑
  MPI_Sendrecv:recv
```
MUST DEADLOCK DETECTION

The application issued a set of MPI calls that can cause a deadlock! The graphs below show details on this situation. This includes a wait-for graph that shows active wait-for dependencies between the processes that cause the deadlock. Note that this process set only includes processes that cause the deadlock and no further processes. A legend details the wait-for graph components in addition, while a parallel call stack view summarizes the locations of the MPI calls that cause the deadlock. Below those graphs, a message queue graph shows active and unmatched point-to-point communications. This graph only includes operations that could have been intended to match a point-to-point operation that is relevant to the deadlock situation. Finally, a parallel call stack shows the locations of any operation in the parallel call stack. The leaves of this call stack graph show the components of the message queue graph that they span. The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).

Active Communicators

Wait-for Graph

Legend

Call Stack

Active and Relevant Point-to-Point Messages: Overview

Active and Relevant Point-to-Point Messages: Callstack-view
PERFORMANCE ANALYSIS TOOLS
TODAY: THE “FREE LUNCH” IS OVER

- Moore's law is still in charge, but
  - Clock rates no longer increase
  - Performance gains only through increased parallelism
- Optimization of applications more difficult
  - Increasing application complexity
    - Multi-physics
    - Multi-scale
  - Increasing machine complexity
    - Hierarchical networks / memory
    - More CPUs / multi-core

Every doubling of scale reveals a new bottleneck!
PERFORMANCE FACTORS

■ “Sequential” (single core) factors
  ■ Computation
     Choose right algorithm, use optimizing compiler
  ■ Vectorization
     Choose right algorithm, use optimizing compiler
  ■ Cache and memory
     Choose the right data structures and layout
  ■ Input / output
     Often not given enough attention
     Parallel I/O matters
PERFORMANCE FACTORS

- “Parallel” (multi core/node) factors
  - Partitioning / decomposition
    - Load balancing
  - Communication (i.e., message passing)
  - Multithreading
  - Core binding
  - NUMA
  - Synchronization / locking
    - More or less understood, good tool support
TUNING BASICS

- Successful engineering is a combination of
  - The right algorithms and libraries
  - Compiler flags and directives
    - Thinking !!!

- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - After each step!
PERFORMANCE ENGINEERING WORKFLOW

- Prepare application (with symbols), insert extra code (probes/hooks)
- Collection of data relevant to execution performance analysis
- Calculation of metrics, identification of performance metrics
- Presentation of results in an intuitive/understandable form
- Modifications intended to eliminate/reduce performance problems
THE 80/20 RULE

- Programs typically spend 80% of their time in 20% of the code
  ➙ *Know what matters!*

- Developers typically spend 20% of their effort to get 80% of the total speedup possible for the application
  ➙ *Know when to stop!*

- Don't optimize what does not matter
  ➙ *Make the common case fast!*
# Measurement Techniques

## A Classification

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem
PROFILING / RUNTIME SUMMARIZATION

- Recording of aggregated information
  - Total, maximum, minimum, …
- For measurements
  - Time
  - Counts
    - Function calls, Bytes transferred, Hardware counters
- Over program and system entities
  - Functions, call sites, basic blocks, loops, …
  - Processes, threads

Profile = summarization of events over execution interval
TRACING

- Recording information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events

\[ Event \ trace = Chronologically \ ordered \ sequence \ of \ event \ records \]
TRACING VS. PROFILING

- Tracing advantages
  - Event traces preserve the **temporal** and **spatial** relationships among individual events (谢谢你 context)
  - Allows reconstruction of **dynamic** application behaviour on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Writing events to file at runtime causes perturbation
  - Writing tracing software is complicated
    - Event buffering, clock synchronization, ...
CRITICAL ISSUES

- Accuracy
  - Intrusion overhead
    - Measurement takes time and thus lowers performance
  - Perturbation
    - Measurement alters program behaviour
    - E.g., memory access pattern
  - Accuracy of timers & counters
- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

Tradeoff: Accuracy vs. Expressiveness of data
TYPICAL PERFORMANCE ANALYSIS PROCEDURE

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements

- **What** is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling

- **Where** is the key bottleneck?
  - Call-path profiling, detailed basic block profiling

- **Why** is it there?
  - Hardware counter analysis
  - Trace selected parts (to keep trace size manageable)

- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function, performance modeling
REMARK: NO SINGLE SOLUTION IS SUFFICIENT!

A combination of different methods, tools and techniques is typically needed!

- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...
PERFORMANCE TOOLS  (STATUS: NOV 2018)

- Score-P
- Scalasca 2
- Vampir[Server]
- HPCToolkit
- Allinea Performance Reports
- Darshan
- NVIDIA Visual Profiler
- TAU
- Intel VTune Amplifier XE
- Intel Advisor
- mpiP*
- Extrae/Paraver*
- PAPI*
SCORE-P

• Community instrumentation and measurement infrastructure
• Developed by a consortium of performance tool groups
• Next generation measurement system of
  • Scalasca 2.x
  • Vampir
  • TAU
  • Periscope
• Common data formats improve tool interoperability
• http://www.score-p.org
SCORE-P OVERVIEW

Vampir, Scalasca, CUBE, TAU, TAUdb, Periscope

Event traces (OTF2)

Call-path profiles (CUBE4, TAU)

Hardware counter (PAPI, rusage)

Score-P measurement infrastructure

Instrumentation wrapper

Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenCL)
Source code instrumentation
User instrumentation

Online interface

Application
• Collection of trace-based performance analysis tools
  • Specifically designed for large-scale systems
  • Unique features:
    • Scalable, automated search for event patterns representing inefficient behavior
    • Scalable identification of the critical execution path
    • Delay / root-cause analysis
  • Based on Score-P for instrumentation and measurement
    • Includes convenience / post-processing commands providing added value
  • http://www.scalasca.org
WHAT IS THE KEY BOTTLENECK?

• Generate **flat MPI profile** using Score-P/Scalasca (or mpiP)
  • Only requires re-linking
  • Low runtime overhead

• Provides detailed information on MPI usage
  • How much time is spent in which operation?
  • How often is each operation called?
  • How much data was transferred?

• Limitations:
  • Computation on non-master threads and outside of MPI_Init/MPI_Finalize scope ignored
FLAT MPI PROFILE: RECIPE

1. Prefix your *link command* with "scorep --nocompiler"

2. Prefix your MPI *launch command* with "scalasca -analyze"

3. After execution, examine analysis results using "scalasca -examine scorep_<title>"
FLAT MPI PROFILE: EXAMPLE (CONT.)

- Aggregate execution time on master threads:
  - Time spent in a particular MPI call:
    - Time spent in selected call as percentage of total time.

Image shows a graphical representation of an MPI profile with metrics such as time, visits, communications, bytes transferred, and more. The screenshot highlights specific call metrics and their durations.
WHERE IS THE KEY BOTTLENECK?

• Generate call-path profile using Score-P/Scalasca
  • Requires re-compilation
  • Runtime overhead depends on application characteristics
  • Typically needs some care setting up a good measurement configuration
    • Filtering
    • Selective instrumentation

• Option 1 (recommended for beginners):
  Automatic compiler-based instrumentation

• Option 2 (for in-depth analysis):
  Manual instrumentation of interesting phases, routines, loops
CALL-PATH PROFILE: RECIPE

1. Prefix your compile & link commands with “scorep”
2. Prefix your MPI launch command with “scalasca -analyze”
3. After execution, compare overall runtime with uninstrumented run to determine overhead
4. If overhead is too high
   1. Score measurement using “scalasca -examine -s scorep_<title>”
   2. Prepare filter file
   3. Re-run measurement with filter applied using prefix “scalasca -analyze -f <filter_file>”
5. After execution, examine analysis results using “scalasca -examine scorep_<title>”
CALL-PATH PROFILE: EXAMPLE (CONT.)

- Estimates trace buffer requirements
- Allows to identify candidate functions for filtering
  - Computational routines with high visit count and low time-per-visit ratio
- Region/call-path classification
  - MPI (pure MPI library functions)
  - OMP (pure OpenMP functions/regions)
  - USR (user-level source local computation)
  - COM (“combined” USR + OpeMP/MPI)
  - ANY/ALL (aggregate of all region types)

% scalasca -examine -s epik_myprog_Ppntx_sum
scorep-score -r ./epik_myprog_Ppntx_sum/profile.cubex
INFO: Score report written to ./scorep_myprog_Ppntx_sum/scorep.score
CALL-PATH PROFILE: EXAMPLE (CONT.)

% less scorep_myprog_Ppmt_x_sum/scorep.score

Estimated aggregate size of event trace: 162GB
Estimated requirements for largest trace buffer (max_buf): 2758MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 2822MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=2822MB to avoid intermediate flushes or reduce requirements using USR regions filters.)

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>2,891,417,902</td>
<td>6,662,521,083</td>
<td>36581.51</td>
<td>100.0</td>
<td>5.49</td>
<td>ALL</td>
</tr>
<tr>
<td>USR</td>
<td>2,858,189,854</td>
<td>6,574,882,113</td>
<td>13618.14</td>
<td>37.2</td>
<td>2.07</td>
<td>USR</td>
</tr>
<tr>
<td>OMP</td>
<td>54,327,600</td>
<td>86,353,920</td>
<td>22719.78</td>
<td>62.1</td>
<td>263.10</td>
<td>OMP</td>
</tr>
<tr>
<td>MPI</td>
<td>676,342</td>
<td>550,010</td>
<td>208.98</td>
<td>0.6</td>
<td>379.96</td>
<td>MPI</td>
</tr>
<tr>
<td>COM</td>
<td>371,930</td>
<td>735,040</td>
<td>34.61</td>
<td>0.1</td>
<td>47.09</td>
<td>COM</td>
</tr>
<tr>
<td>USR</td>
<td>921,918,660</td>
<td>2,110,313,472</td>
<td>3290.11</td>
<td>9.0</td>
<td>1.56</td>
<td>matmul_sub</td>
</tr>
<tr>
<td>USR</td>
<td>921,918,660</td>
<td>2,110,313,472</td>
<td>5914.98</td>
<td>16.2</td>
<td>2.80</td>
<td>binvcrhs</td>
</tr>
<tr>
<td>USR</td>
<td>921,918,660</td>
<td>2,110,313,472</td>
<td>3822.64</td>
<td>10.4</td>
<td>1.81</td>
<td>matvec_sub</td>
</tr>
<tr>
<td>USR</td>
<td>41,071,134</td>
<td>87,475,200</td>
<td>358.56</td>
<td>1.0</td>
<td>4.10</td>
<td>lhsinit</td>
</tr>
<tr>
<td>USR</td>
<td>41,071,134</td>
<td>87,475,200</td>
<td>145.42</td>
<td>0.4</td>
<td>1.66</td>
<td>binvrhs</td>
</tr>
<tr>
<td>USR</td>
<td>29,194,256</td>
<td>68,892,672</td>
<td>86.15</td>
<td>0.2</td>
<td>1.25</td>
<td>exact_solution</td>
</tr>
</tbody>
</table>
| OMP      | 3,280,320     | 3,293,184 | 15.81 | 0.0 | 4.80 | !$omp parallel  [...]
CALL-PATH PROFILE: FILTERING

• In this example, the 6 most frequently called routines are of type USR
• These routines contribute around 35% of total time
  • However, much of that is most likely measurement overhead
    • Frequently executed
    • Time-per-visit ratio in the order of a few microseconds

☞ Avoid measurements to reduce the overhead
☞ List routines to be filtered in simple text file
FILTERING: EXAMPLE

Score-P filtering files support

- Wildcards (shell globs)
- Blacklisting
- Whitelisting
- Filtering based on filenames
CALL-PATH PROFILE: EXAMPLE (CONT.)
Box plot view shows distribution across processes/threads

Distribution of selected metric across call tree

When expanding, value changes from inclusive to exclusive

Selection updates columns to the right
CALL-PATH PROFILE: EXAMPLE (CONT.)

Split base metrics into more specific metrics.
SCORE-P: ADVANCED FEATURES

- Measurement can be extensively configured via environment variables
  - Check output of “scorep-info config-vars” for details
- Allows for targeted measurements:
  - Selective recording
  - Phase profiling
  - Parameter-based profiling
  - …
- Please ask us or see the user manual for details
WHY IS THE BOTTLENECK THERE?

- This is highly application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Manual/automatic event trace analysis
HARDWARE COUNTERS

- Counters: set of registers that count processor events, e.g. floating point operations or cycles
- Number of registers, counters and simultaneously measurable events vary between platforms
- Can be measured by:
  - perf:
    - Integrated in Linux since Kernel 2.6.31
    - Library and CLI
  - LIKWID:
    - Direct access to MSRs (requires Kernel module)
    - Consists of multiple tools and an API
  - PAPI (Performance API)
PAPI

- Portable API: Uses the same routines to access counters across all supported architectures
- Used by most performance analysis tools

- High-level interface:
  - Predefined standard events, e.g. PAPI_FP_OPS
  - Availability and definition of events varies between platforms
  - List of available counters: papi_avail (-d)

- Low-level interface:
  - Provides access to all machine specific counters
  - Non-portable
  - More flexible
  - List of available counters: papi_native_avail
Automatic Trace Analysis w/ Scalasca

- **Idea:** Automatic search for patterns of inefficient behavior
  - Identification of wait states and their root causes
  - Classification of behavior & quantification of significance
  - Scalable identification of the critical execution path

- **Advantages**
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Helps to identify hot-spots for in-depth manual analysis
TRACE GENERATION & ANALYSIS W/ SCALASCA

• Enable trace collection & analysis using “-t” option of “scalasca -analyze”:

```bash
###
## In the job script: ##
###
module load ENV Score-P Scalasca
export SCOREP_TOTAL_MEMORY=120MB  # Consult score report
scalasca -analyze -f filter.txt -t
   runjob --ranks-per-node P --np n [...] --exe ./myprog
```

• ATTENTION:
  • Traces can quickly become extremely large!
  • Remember to use proper filtering, selective instrumentation, and Score-P memory specification
  • Before flooding the file system, ask us for assistance!
SCALASCA TRACE ANALYSIS EXAMPLE

Additional wait-state metrics from the trace analysis

Delay / root-cause metrics

Critical-path profile
VAMPIR EVENT TRACE VISUALIZER

- Offline trace visualization for Score-P’s OTF2 trace files
- Visualization of MPI, OpenMP and application events:
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for ANY part of the trace
- http://www.vampir.eu

- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis
VAMPIR: TIMELINE DIAGRAM

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size
- Information about states, messages, collective and I/O operations available through clicking on the representation
VAMPIR: PROCESS AND COUNTER TIMELINES

- Process timeline shows call stack nesting
- Counter timelines for hardware and software counters
VAMPIR: EXECUTION STATISTICS

- Aggregated profiling information: execution time, number of calls, inclusive/exclusive
- Available for all / any group (activity) or all routines (symbols)
- Available for any part of the trace ⇒ selectable through time line diagram
VAMPIR: PROCESS SUMMARY

- Execution statistics over all processes for comparison
- Clustering mode available for large process counts
VAMPIR: COMMUNICATION STATISTICS

• Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair

• Message length statistics

▪ Available for any part of the trace
VTUNE AMPLIFIER XE

- Feature-rich profiler for Intel platforms
- Supports Python, C/C++ and Fortran
- MPI support continuously improving
- Lock and Wait analysis for OpenMP and TBB
- HPC analysis for quick overview
- Bandwidth and memory analysis
- I/O analysis
- OpenCL and GPU profiling (no CUDA, Intel iGPU only)
INTEL VTUNE AMPLIFIER GUI

- Potential gain is noticeable
- Spinning due to imbalance
- Red color: threads spinning
- Brown color: threads calculating
- Green color: threads waiting
INTEL VTUNE – GPU ANALYSIS
INTEL ADVISOR

• Vectorization Advisor
  • Loops-based analysis to identify vectorization candidates
  • Finds save spots to enforce compiler vectorization
  • Roofline analysis to explore performance headroom and co-optimize memory and computation

• Threading Advisor
  • Identify issues before parallelization
  • Prototype performance impact of different threading designs
  • Find and eliminate data-sharing issues

• Flow-Graph Analysis
  • Speed up algorithm design and express parallelism efficiently
  • Plan, validate, and model application design

• C/C++ and Fortran with OpenMP and Intel TBB
INTEL ADVISOR GUI

Loop may have several parts or versions

Loop summary

Recommendations for optimization

Vectorization and compiler optimization details
INTEL ADVISOR – ROOFLINE
ALLINEA PERFORMANCE REPORTS

- Single page report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows CPU, memory, network and I/O utilization

- Supports MPI, multi-threading and accelerators
- Saves data in HTML, CVS or text form

- Note: License limited to 512 processes (with unlimited number of threads)
# EXAMPLE PERFORMANCE REPORTS

**Summary:** cp2k.popt is **CPU-bound** in this configuration

The total wallclock time was spent as follows:

<table>
<thead>
<tr>
<th>Component</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>56.5%</td>
</tr>
<tr>
<td>MPI</td>
<td>43.5%</td>
</tr>
<tr>
<td>I/O</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

- **CPU:** 56.5% of the time was spent running application code. High values are usually good. This is an **average**; check the CPU performance section for optimization advice.

- **MPI:** 43.5% of the time was spent in MPI calls. High values are usually bad. This is an **average**; check the MPI breakdown for advice on reducing it.

- **I/O:** 0.0% of the time was spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance.

This application run was **CPU-bound**. A breakdown of this time and advice for investigating further is in the **CPU** section below.

---

## CPU

A breakdown of how the 56.5% total CPU time was spent:

<table>
<thead>
<tr>
<th>Category</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar numeric ops</td>
<td>27.7%</td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td>11.3%</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>60.9%</td>
</tr>
<tr>
<td>Other</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

The per-core performance is **memory-bound**. Use a profiler to identify time-consuming loops and check their cache performance. Little time is spent in **vectorized instructions**. Check the compiler's vectorization advice to see why key loops could not be vectorized.

---

## MPI

Of the 43.5% total time spent in MPI calls:

<table>
<thead>
<tr>
<th>Category</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in collective calls</td>
<td>8.2%</td>
</tr>
<tr>
<td>Time in point-to-point calls</td>
<td>91.8%</td>
</tr>
<tr>
<td>Estimated collective rate</td>
<td>169 Mbs/s</td>
</tr>
<tr>
<td>Estimated point-to-point rate</td>
<td>50.6 Mbs/s</td>
</tr>
</tbody>
</table>

The **point-to-point** transfer rate is low. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait. Use an MPI profiler to identify the problematic calls and ranks.

---

## I/O

A breakdown of how the 0.0% total I/O time was spent:

<table>
<thead>
<tr>
<th>Category</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in reads</td>
<td>0.0%</td>
</tr>
<tr>
<td>Time in writes</td>
<td>0.0%</td>
</tr>
<tr>
<td>Estimated read rate</td>
<td>0 bytes/s</td>
</tr>
<tr>
<td>Estimated write rate</td>
<td>0 bytes/s</td>
</tr>
</tbody>
</table>

No time is spent in **I/O operations**. There's nothing to optimize here!

---

## Memory

Per-process memory usage may also affect scaling:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean process memory usage</td>
<td>82.5 Mb</td>
</tr>
<tr>
<td>Peak process memory usage</td>
<td>89.3 Mb</td>
</tr>
<tr>
<td>Peak node memory usage</td>
<td>7.4%</td>
</tr>
</tbody>
</table>

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.
DARSHAN

- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

- Supports POSIX, MPI-IO, HDF5, PnetCDF, …
- Binary log file written at exit post-processed into PDF report

- Open Source: installed on many HPC systems
EXAMPLE DARSHAN REPORT EXTRACT

jobid: | uid: | nprocs: 4096 | runtime: 175 seconds

Average I/O cost per process

I/O Operation Counts

POSIX
MPI-IO Indep.
MPI-IO Coll.

I/O Sizes

I/O Pattern

Read
Write
Total
Consecutive
NVIDIA VISUAL PROFILER

- Part of the CUDA Toolkit
- Supports all CUDA enabled GPUs
- Supports CUDA and OpenACC on Windows, OS X and Linux

- Unified CPU and GPU Timeline
- CUDA API trace
  - Memory transfers, kernel launches, and other API functions
- Automated performance analysis
  - Identify performance bottlenecks and get optimization suggestions
- Guided Application Analysis
- Power, thermal, and clock profiling
NVIDIA VISUAL PROFILER: EXAMPLE

Timeline view

Detailed information on Kernel execution

Automatic analysis of performance bottlenecks
POP SERVICES

Performance Optimisation and Productivity
POP COE

- A Center of Excellence
  - On Performance Optimization and Productivity
  - Promoting best practices in parallel programming
- Providing Services
  - Precise understanding of application and system behaviour
  - Suggestions/support on how to refactor code in the most productive way
- Horizontal
  - Transversal across application areas, platforms and scales
- For (your?) academic AND industrial codes and users

The best: Currently free of charge
MOTIVATION

• Why?
  • Complexity of machines and codes
    → Frequent lack of quantified understanding of actual behaviour
    → Not clear most productive direction of code refactoring
  • Important to maximize efficiency (performance, power) of compute intensive applications and productivity of the development efforts

• What?
  • Parallel programs, mainly MPI/OpenMP
    • Also CUDA, OpenCL, OpenACC, Python, …
THE PROCESS ...

When?
- POP I: October 2015 – March 2018
- POP II: December 2018 – November 2021

How?
- Apply
- Fill in small questionnaire describing application and needs https://pop-coe.eu/request-service-form
- Questions? Ask pop@bsc.es
- Selection/assignment process
- Install tools @ your production machine (local, PRACE, …)
- Interactively: Gather data → Analysis → Report
SERVICES PROVIDED BY THE COE

? Parallel Application Performance Audit ⇒ Report
• Primary service
• Identify performance issues of customer code (at customer site)
• Small effort (< 1 month)

! Parallel Application Performance Plan ⇒ Report
• Follow-up on the audit service
• Identifies the root causes of the issues found and qualifies and quantifies approaches to address them
• Longer effort (1-3 months)

✓ Proof-of-Concept ⇒ Software Demonstrator
• Experiments and mock-up tests for customer codes
• Kernel extraction, parallelisation, mini-apps experiments to show effect of proposed optimisations
• 6 months effort