Performance Engineering Basics

Markus Geimer, Ilya Zhukov
Jülich Supercomputing Centre
Performance: an old problem

“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”

Charles Babbage
1791 – 1871
Tuning basics

- Successful performance engineering is a combination of
  - Careful setting of various tuning parameters
  - 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!

- Avoid reinventing the wheel
  
  Use existing performance analysis tools!
Performance analysis tools available on JURECA/JUWELS

- **Score-P**
  - Community-maintained instrumentation and measurement system
    - Supports the generation of profiles & event traces
    - MPI, SHMEM, OpenMP, POSIX threads, CUDA, OpenACC, OpenCL, ...

- **Scalasca Trace Tools**
  - Automatic event trace analyzer
    - Detects communication/synchronization wait states & their root causes
    - Identifies the critical path

- **Cube**
  - Graphical performance report explorer + command-line tools
    - To analyze reports generated by Score-P & Scalasca

- **Vampir**
  - Interactive event trace visualizer
Performance analysis tools available on JURECA/JUWELS (cont.)

- ARM Performance Reports
  - High-level performance overview

- Intel Advisor
  - Node-level vectorization and threading analysis

- Intel VTune Amplifier XE
  - Feature-rich profiler for Intel platforms

- Darshan
  - I/O profiler

- NVIDIA nvprof
  - Profiler for GPU activities (comes as part of CUDA toolkit)
Step 0: Prepare a reference run (baseline)

- Select a representative input data set
  - Make it as small as possible, but not smaller

- Select a reasonable run configuration
  - Avoid burning CPU cycles for nothing
  - For example, start with only a few iterations/timesteps (unless you already know/discovered that your performance problem is iteration-dependent)

- Make sure your code runs & produces correct results
  - Nobody cares how fast you can compute the wrong answer...

- Determine a reference run time
  - Preferably excluding non-deterministic parts (e.g., I/O)

☞ *Keep this setup constant during your experiments!* (unless when carrying out scaling experiments)
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results

- Optimization
  - Modifications intended to eliminate/reduce performance problem
The 80/20 rule

- Programs typically spend 80% of their time in 20% of the code
- Programmers 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!

“If you optimize everything, you will always be unhappy.”

Donald E. Knuth
### 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
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, ...
Case study: CIAO
CIAO – RWTH AACHEN, ITV

- Multiscale Navier-Stokes solver for turbulent reacting flows in complex geometries
- Fortran code parallelised with MPI
- Combustion flame simulation testcase (10 steps)
- Executed on JUWELS
- Measurements done with Intel APS and Score-P and Scalasca

**Key findings:**

- Application spends considerable amount of time in communication
- Domain decomposition suffered from load imbalance

**Using a better workload reduced communication time by 9% and total runtime by 4%**
Baseline measurement with application performance snapshot

- Application is memory bound
- 37% of elapsed time is MPI time
- I/O is negligible
Initial run with Score-P and Scalasca

- 70% runtime dilation with initial measurement
- Filtering reduced runtime dilation to 2%
- Scalasca identified potential cause of large MPI time, i.e. 'combustion_step_finite_'
- 'combustion_step_finite_' has significant load imbalance, i.e. 98%
Optimized run with Score-P and Scalasca

- Change input parameter to optimize workload
- Optimized version reduced
  - Load imbalance in 'combustion_step_finite' by 33%
  - Communication by 9%
  - Total runtime by 4%
Shameless plugs... 😊
VI-HPS Tuning Workshops

- **VI-HPS?**
  - Virtual Institute – High-Productivity Supercomputing
  - Collaboration of renowned tool developers

- **Multi-day “bring-your-own-code” workshop**
  - Duration: 3-5 days
  - Overview of VI-HPS tools suite
  - Introduction and hands-on exercises with individual tools
  - Time to apply tools to your own code
  - Expert assistance to help & discuss results

- **Save the date!**
  - Next workshop: 24.–28.06.2019 @ JSC
POP COE (HTTPS://POP-COE.EU)

- A Centre of Excellence
  - On Performance Optimisation and Productivity
  - Promoting best practices in parallel programming

- Providing FREE Services
  - Precise understanding of application and system behaviour
  - Suggestion/support on how to refactor code in the most productive way

- Horizontal
  - Transversal across application areas, platforms, scales

- For (EU) academic AND industrial codes and users!
FREE SERVICES PROVIDED BY THE POP COE

Parallel Application Performance Audit

• Primary service
• Identify performance issues of customer code (at customer site)
• Small effort (< 1 month)

Parallel Application Performance Plan

• Follow-up on the audit service
• Identifies the root causes of the issues found and qualifies and quantifies approaches to address them (recommendations)
• Longer effort (1-3 months)

Proof of Concept

• Experiments and mock-up tests for customer codes
• Kernel extraction, parallelisation, mini-apps experiments to show effect of proposed optimisations
• 6 months effort
Thank you!
Appendix
Classification of measurement & analysis techniques

- 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
**Sampling**

- Running program is periodically interrupted to take measurement
  - Timer interrupt, OS signal, or HWC overflow
  - Service routine examines return-address stack
  - Addresses are mapped to routines using symbol table information
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

```c
int main()
{
    int i;
    for (i=0; i < 3; i++)
        foo(i);
    return 0;
}

void foo(int i)
{
    if (i > 0)
        foo(i - 1);
}
```
Instrumentation

- Measurement code is inserted such that every event of interest is captured directly
  - Can be done in various ways
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions

```c
int main()
{
  int i;
  Enter("main");
  for (i=0; i < 3; i++)
    foo(i);
  Leave("main");
  return 0;
}

void foo(int i)
{
  Enter("foo");
  if (i > 0)
    foo(i - 1);
  Leave("foo");
}
```
Critical issues

- **Accuracy**
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behavior
    - E.g., memory access pattern
  - Accuracy of timers & counters

- **Granularity**
  - How many measurements?
  - How much information / processing during each measurement?

antiago: accuracy vs. expressiveness of data

Compare with run time of your reference run to determine measurement overhead!
Classification of measurement & analysis techniques

- 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
Types of profiles

- Flat profile
  - Shows distribution of metrics per routine / instrumented region
  - Calling context is not taken into account

- Call-path profile
  - Shows distribution of metrics per executed call path
  - Sometimes only distinguished by partial calling context (e.g., two levels)

- Special-purpose profiles
  - Focus on specific aspects, e.g., MPI calls, OpenMP constructs, or I/O activities
  - Comparing processes/threads
Tracing

- Recording detailed 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: Pros & Cons

■ Tracing advantages
  ■ Event traces preserve the temporal and spatial relationships among individual events (context)
  ■ Allows reconstruction of dynamic application behavior 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 may causes perturbation
Classification of measurement & analysis techniques

- 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
Online analysis

- Performance data is processed during measurement run
  - Process-local profile aggregation
  - Requires formalized knowledge about performance bottlenecks
  - More sophisticated inter-process analysis using
    - “Piggyback” messages
    - Hierarchical network of analysis agents

- Online analysis often involves application steering to interrupt and re-configure the measurement
Post-mortem analysis

- Performance data is stored at end of measurement run
- Data analysis is performed afterwards
  - Automatic search for bottlenecks
  - Visual trace analysis
  - Calculation of statistics