Performance analysis

Florian Stiefel

Research group Scientific Computing
Department of Informatics
Faculty of Mathematics, Informatics and Natural Sciences
University of Hamburg

17.11.2016

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

informatik
die zukunft

Agenda

1 Introduction

2 Methods

3 Tools

4 Conclusion

5 Demo

6 Literature

Agenda

1 Introduction
- Why do we need performance analysis?
- When is performance analysis useful?

2 Methods

3 Tools

4 Conclusion

5 Demo

6 Literature

Program efficiency

What impacts program efficiency?

- Time consumption

Theory time complexity of algorithms (Big O notation)
Practice implementation of algorithms

- Space consumption
  - data input/output size
  - space used during the calculation
- Energy consumption

You will always have the Memory Time Trade-off problem, meaning you have to choose between time and memory consumption.

Why analyzing?

We want to optimize our programs, because optimized programs:

- Save time
- Save money

Before we can optimize, we have to analyze

We have to find bottlenecks in terms of:

- CPU usage
- Memory usage
- Disk usage

When to analyze?

Performance analysis is useful for

- Application software when
  - The software feels unresponsive
  - The software uses a lot more ressources than expected
- High Performance Computing (HPC) when
  - A lot of CPU cores are unused
  - A lot of memory swapping occurs

Generally:
Premature optimization is bad, because of readability and development speed
Thus, write the program first, then optimize.

Optimization Cycle Workflow

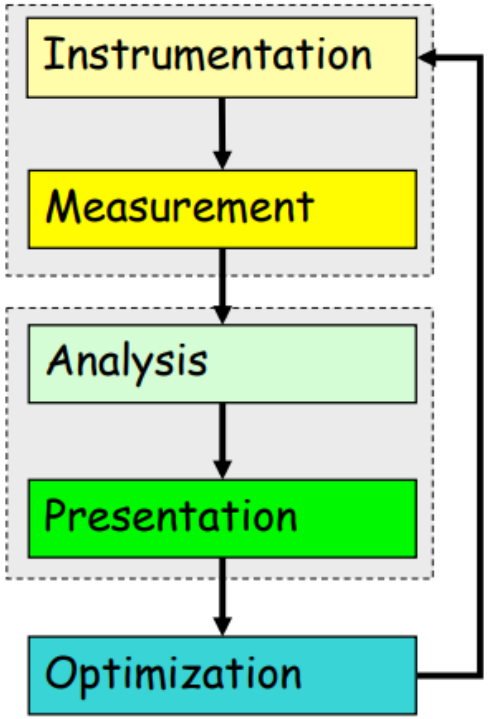


Figure: Source [Moo11]

Agenda

1 Introduction

2 Methods
- Profiling
- Tracing
- Benchmarking

3 Tools

4 Conclusion

5 Demo

What is profiling?

- High-level summary of program performance
- Aggregates statistics at runtime

Data collected:

- Total time spend, total bytes sent/written
- Number of times a method was invoked
- Average time a method takes executing

How can you do profiling?

To profile your program you can:

- Manually add output, aggregate them in a log file
- Use compiler options, e.g. *gcc -pg* with gprof
- Use external profiling tools

Example output with gprof for C++

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 45.79      0.26     0.26                             _fini
 29.94      0.43     0.17 39757296     4.29     4.29  ackermann(unsigned long, unsigned long)
 24.65      0.57     0.14                             __libc_csu_init
```

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     54.4    0.14    0.17                 __libc_csu_init [1]
                0.17    0.00 39757295/39757296     ackermann(unsigned long, unsigned long) [3]
-----------------------------------------------
                                                 <spontaneous>
[2]     45.6    0.26    0.00                 _fini [2]
                0.00    0.00       1/39757296     ackermann(unsigned long, unsigned long) [3]
-----------------------------------------------
                0.00    0.00       1/39757296     _fini [2]
                0.17    0.00 39757295/39757296     __libc_csu_init [1]
[3]     29.8    0.17    0.00 39757296          ackermann(unsigned long, unsigned long) [3]
-----------------------------------------------
```

Pros and Cons

Pro

- Finite data size
  - Small and clear
- Little overhead
- Delivers a overview of the performance (problems)

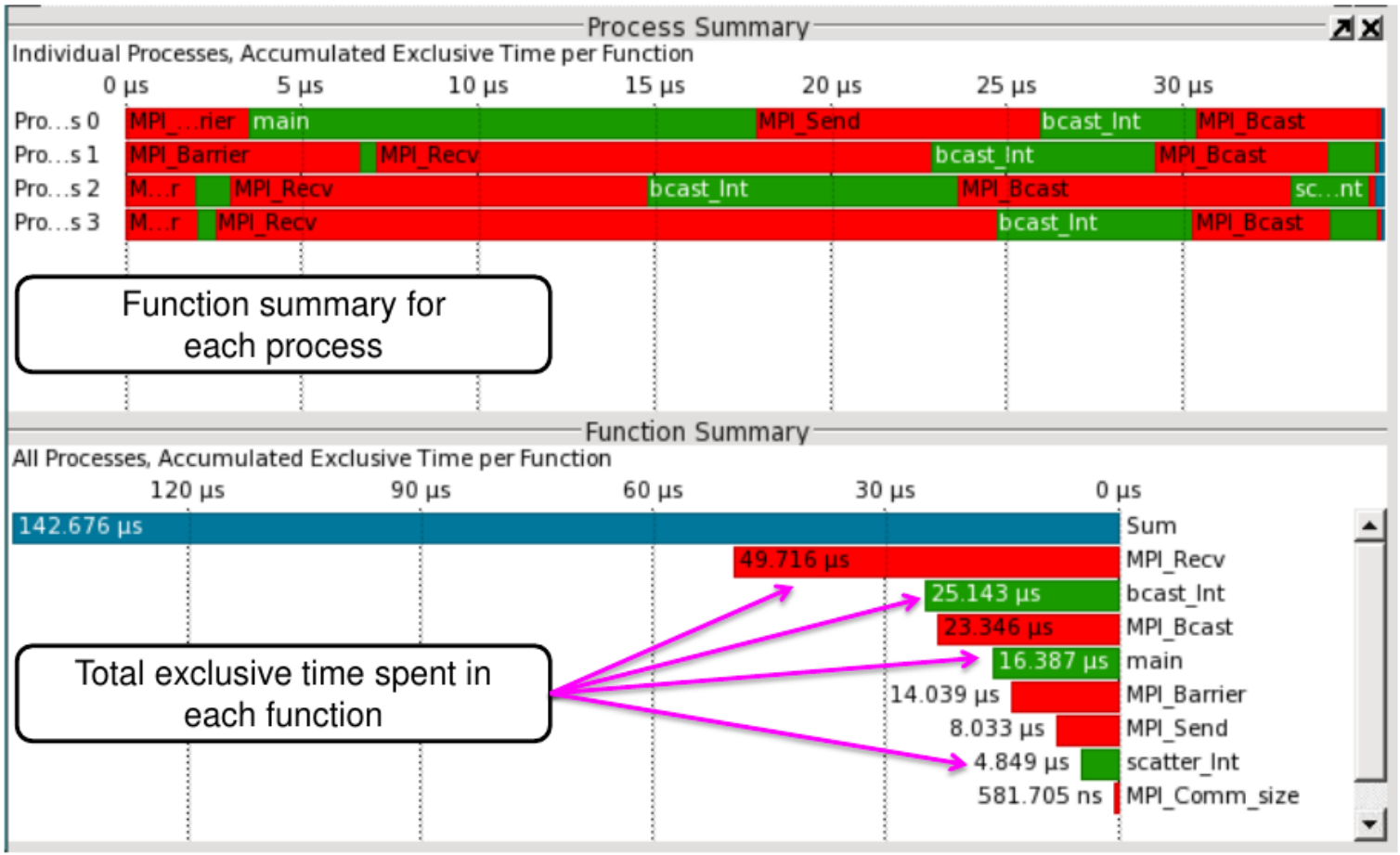
Contra

- Lacks certain information
- Cannot describe process interaction

What is tracing?

- Collects event history
- Generates a trace record with timestamps
- Generates very detailled data
- Common to be displayed on a timeline

Tracing visualized



Pros and Cons

Pro

- Covers spatial temporal dimension
  - Meaning parallel processes and process interaction
- Delivers detailled data which can be great for understanding the programm behavior

Contra

- Can generate huge data sets
- Generates overhead
  - Can change the program behavior

What is benchmarking?

- A way to test and compare your programs performance
- Can be easily repeated
- Generates a score to easily compare programs
  - e.g. Running two different versions of your program with the same data set

What do you need for a benchmark?

A benchmark consists of a

- Scenario
- Criteria
- Metrics
- Score

You can met these requirements if you are using a benchmarking framework.

Agenda

1 Introduction

2 Methods

3 Tools
- Score-P
- Vampir

4 Conclusion

5 Demo

6 Literature

Performance tools

Performance tools will help you:

1. Understand the runtime behavior better
2. Find bottlenecks for you to optimize
3. Visualize your data

They will not:

- Make your program run faster

There are different tools for different approaches or languages (direct/indirect, singlecore/multicore (parallel))

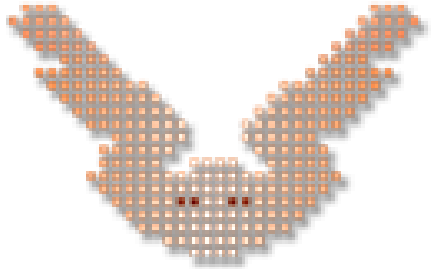
Score-P

- Tool suite for performance analysis of HPC applications
- Supports profiling and event tracing
- Works with multiple analysis tools
- Uses Open Trace Format

How it works

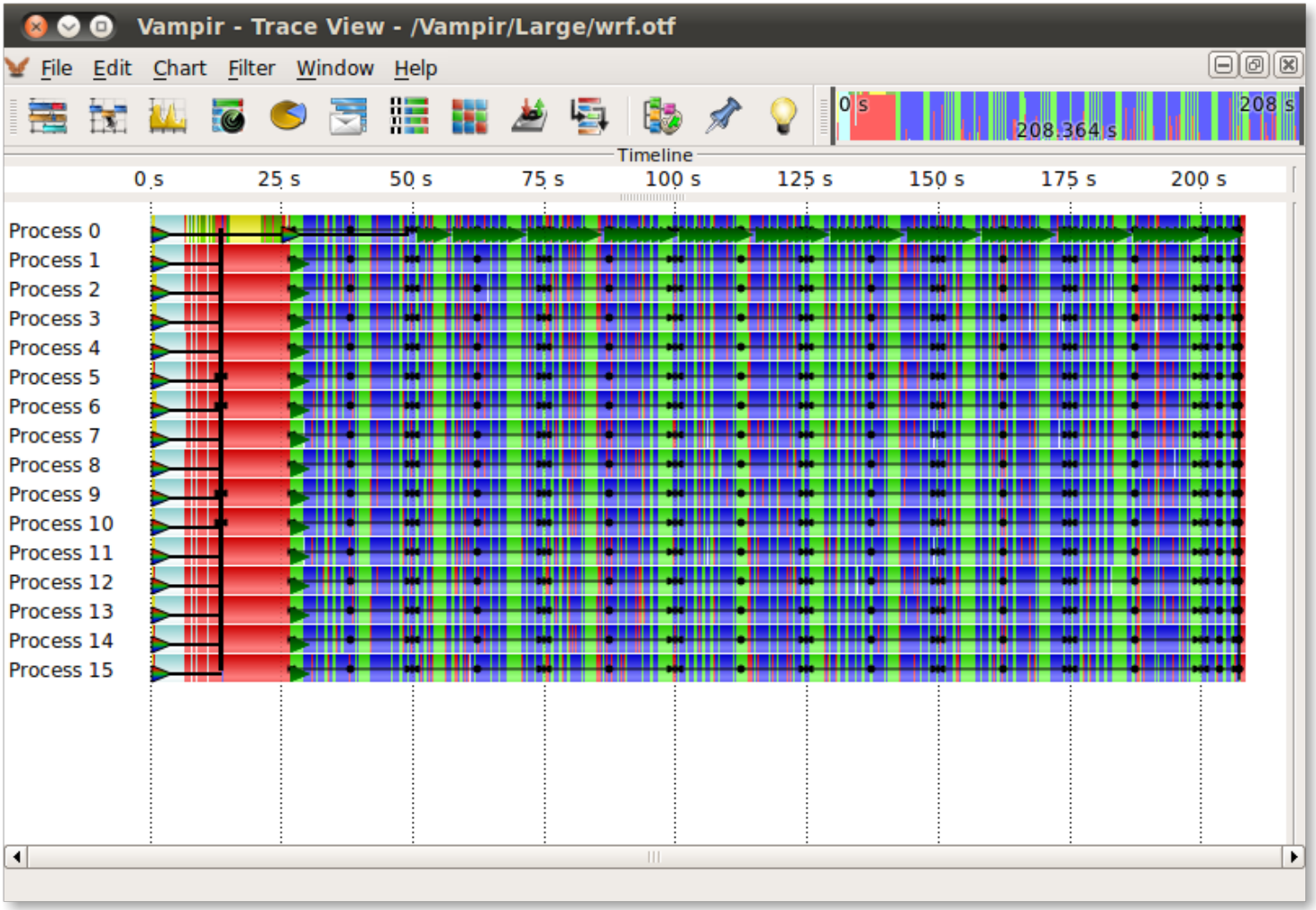
- Acts as a compiler wrapper with e.g. mpicc or mpifort
- Needs libraries like OpenMP
- Inserts measurement probes into C/C++ or Fortran code
  - The data is collected when the probes are triggered

Vampir



- Analysis framework
- Uses visualizes data collected with Score-P
- Has a server edition for very big data sets

Vampir tracing monitor



Agenda

1 Introduction

2 Methods

3 Tools

4 Conclusion

5 Demo

6 Literature

Possible workflow for performance analysis

A good way to analyze a program would be

1. First write the programm
2. Then profile the behavior for a approximate performance analysis
3. When bottlenecks are detected you can do tracing for details
4. After optimization you can benchmark your program to compare it
5. Repeat step 2 to 4 until you reach good performance

Agenda

1 Introduction

2 Methods

3 Tools

4 Conclusion

5 Demo

6 Literature

Live demo

Vampir Live Demo

Agenda

1 Introduction

2 Methods

3 Tools

4 Conclusion

5 Demo

6 Literature

Literatur

[Hsi16] Paul Hsieh. Programming Optimization, February 2016.

[Ili12] Hristo Iliev. Advanced mpi. hybrid programming, profiling and debugging of mpi applications. Technical report, Rechen- und Kommunikationszentrum (RZ), RWTH Aachen University, 2012.

[Jon95] C. Jones. Software benchmarking. *Computer*, 28(10):102–103, Oct 1995.

[KR12] Andreas Knüpfer and Rössel. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*, pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Moo11] Shirley Moore. Introduction to parallel performance analysis and engineering. VI-HPS Tuning Workshop,