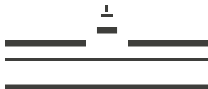


Applied
Mathematics
Münster

Performance Testing

Seminar zu Software-Tools für die Numerische Mathematik

living.knowledge
WWU Münster

Andreas Buhr

15. Oktober 2014

Mein Programm soll schneller werden!

Wie geht das?

living.knowledge
WWU Münster

Mein Programm soll schneller werden!

Wie geht das?

1. Messen

Mein Programm soll schneller werden!

Wie geht das?

1. Messen
2. Ändern

living.knowledge
WWU Münster

Mein Programm soll schneller werden!

Wie geht das?

1. Messen
2. Ändern
3. Messen

living.knowledge
WWU Münster

Mein Programm soll schneller werden!

Wie geht das?

1. Messen
2. Ändern
3. Messen
4. Ändern

living.knowledge
WWU Münster

Mein Programm soll schneller werden!

Wie geht das?

1. Messen
2. Ändern
3. Messen
4. Ändern
5. Messen
6. Ändern
7. ...

living.knowledge
WWU Münster

Messen

- Grundregel Nr. 1:
  Messen!

Messen

- Grundregel Nr. 1:
  Messen!
- ... nicht vermuten
- ... nicht annehmen
- ... nicht raten

Messen

- Grundregel Nr. 1:
  Messen!
- ... nicht vermuten
- ... nicht annehmen
- ... nicht raten

... sonst kann es sein, dass man am Ende Stunden damit verbringt, ein Promille der Laufzeit zu entfernen.

living.knowledge WWU Münster

Messen in welchem Umfeld?

Zwei Arten des Messens:

1. Zeitmessung
2. Profiling

Messen in welchem Umfeld?

Zwei Arten des Messens:

1. Zeitmessung
2. Profiling

Gängige Programmiersprachen:

1. C++
2. Python
3. Matlab

Messen in welchem Umfeld?

Zwei Arten des Messens:

1. Zeitmessung
2. Profiling

Gängige Programmiersprachen:

1. C++
2. Python
3. Matlab

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

living.knowledge
WWU Münster

First things first

Das richtige Messen

- Messungen immer mit voller Optimierung (“-O3 -DNDEBUG”)

living.knowledge
WWU Münster

First things first

Das richtige Messen

- Messungen immer mit voller Optimierung ("-O3 -DNDEBUG")
- Turbo-Boost im BIOS deaktivieren

First things first

Das richtige Messen

- Messungen immer mit voller Optimierung (“-O3 -DNDEBUG”)
- Turbo-Boost im BIOS deaktivieren
- Bei Messungen alle Programme schließen und PC nicht berühren

living.knowledge
WWU Münster

First things first

Das richtige Messen

- Messungen immer mit voller Optimierung (“-O3 -DNDEBUG”)
- Turbo-Boost im BIOS deaktivieren
- Bei Messungen alle Programme schließen und PC nicht berühren
- Threads auf NUMA-Systemen festpinnen

First things first

Das richtige Messen

- Messungen immer mit voller Optimierung (“-O3 -DNDEBUG”)
- Turbo-Boost im BIOS deaktivieren
- Bei Messungen alle Programme schließen und PC nicht berühren
- Threads auf NUMA-Systemen festpinnen
- Auf Uni-Rechnern: sicherstellen, dass sonst keiner den PC verwendet (“users” / “top”)

First things first

Das richtige Messen

- Messungen immer mit voller Optimierung (“-O3 -DNDEBUG”)
- Turbo-Boost im BIOS deaktivieren
- Bei Messungen alle Programme schließen und PC nicht berühren
- Threads auf NUMA-Systemen festpinnen
- Auf Uni-Rechnern: sicherstellen, dass sonst keiner den PC verwendet (“users” / “top”)
- Alle print-statements aus dem Code raus

living.knowledge
WWU Münster

Vokabeln

- wall-time / real-time : Echtzeit

Vokabeln

- wall-time / real-time : Echtzeit
- user-time: Prozesszeit in User-Mode (Berechnungen)

living.knowledge WWU Münster

Vokabeln

- wall-time / real-time : Echtzeit
- user-time: Prozesszeit in User-Mode (Berechnungen)
- sys-time: Prozesszeit in Kernel-Mode (Festplattenzugriffe, Synchronisationen o.ä.)

Zeitmessungen in C++

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

living.knowledge
WWU Münster

Zeitmessung mit ctime

```
#include <ctime>

void f() {
  using namespace std;
  clock_t begin = clock();
  somefunction();
  clock_t end = clock();
  double elapsed = double(end - begin) / CLOCKS_PER_SEC;
}
```

Performance: 280 ns/Measurement (test for your system)
Resolution: 1000 ns

Zeitmessung mit std::chrono

```
#include <chrono>

void f() {
  auto begin = std::chrono::system_clock::now();
  somefunction();
  auto end = std::chrono::system_clock::now();
  auto elapsed_ns =
    std::chrono::duration<long long, std::nano>(end-begin)
      .count();
}
```

Performance: 59 ns/Measurement (test for your system)
compile with “-std=c++11”

living.knowledge
WWU Münster

Zeitmessung mit OpenMP

```
#include "omp.h"
void f() {
  double begin = omp_get_wtime();
  somefunction();
  double end = omp_get_wtime();
  auto elapsed_s = end - begin;
}
```

Performance: 75 ns/Measurement (test for your system)
compile with “-fopenmp”

living.knowledge WWU Münster

Zeitmessungen in Python

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

Python: das time-Modul

```
import time
start = time.time()
somefunction()
end = time.time()
print(end-start)
```

Performance: 400 ns/Measurement (test for your system)
Vorsicht: Garbage-Collection kann zu zufälligen Resultaten führen
Vorsicht: Auflösung unter Windows viel schlechter (16 ms)

living.knowledge
WWU Münster

Python: das time-Modul

```
import time
start = time.clock()
somefunction()
end = time.clock()
print(end-start)
```

Achtung: processing-time unter Linux, wall-time unter Windows

living.knowledge
WWU Münster

Python: das time-Modul

Empfehlung:

- Unter Linux: time.time() für wall-time verwenden (ungenau unter Win)
- Unter Windows: time.clock() für wall-time verwenden (processing time unter Linux)

living.knowledge
WWU Münster

Python: das time-Modul

Empfehlung:

- Unter Linux: time.time() für wall-time verwenden (ungenau unter Win)
- Unter Windows: time.clock() für wall-time verwenden (processing time unter Linux)

timeit-Sourcecode:

```
import sys
if sys.platform == 'win32':
    # On Windows, the best timer is time.clock
    default_timer = time.clock
else:
    # On most other platforms the best timer is time.time
    default_timer = time.time
```

living.knowledge
WWU Münster

das time-Modul SUXX!?

Das ist Mist, deshalb neu in Python 3.3:

- time.perf_counter():
  wall-time mit 400 ns Messdauer
- time.process_time():
  processing-time mit 650 ns Messdauer

Python: das timeit-Modul

- Macht automatisch Performance-Tests für kleine Codesegmente
- Deaktiviert die Garbage-Collection
- Führt Codesegment (default) 1 mio mal aus und gibt Zeit zurück

Python: das timeit-Modul

- Macht automatisch Performance-Tests für kleine Codesegmente
- Deaktiviert die Garbage-Collection
- Führt Codesegment (default) 1 mio mal aus und gibt Zeit zurück

```python
import timeit
def myfunction():
    pass

statement = "myfunction()"
initcode = "from __main__ import myfunction"
result = timeit.timeit(statement, initcode)
print(result)
```

living.knowledge
WWU Münster

Zeitmessungen in Matlab

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

Zeitmessungen mit tic/toc

Syntax 1:

```
tic()
somefunction()
time = toc()
```

Octave-Performance: 5300 ns/Measurement
Matlab-Performance: 530 ns/Measurement

living.knowledge WWU Münster

Zeitmessungen mit tic/toc

Syntax 1:

```
tic()
somefunction()
time = toc()
```

Octave-Performance: 5300 ns/Measurement
Matlab-Performance: 530 ns/Measurement
Syntax 2, erlaubt mehrere gleichzeitig:

```
timer = tic()
somefunction()
time = toc(timer)
```

Octave-Performance: 8500 ns/Measurement
Matlab-Performance: 5300 ns/Measurement

living.knowledge
WWU Münster

Zeitmessungen mit timeit (Matlab only)

```
myfunction = @somefunction
time = timeit(myfunction)
```

Führt angegebene Funktion häufig aus und gibt typische Laufzeit zurück.

Was ist Profiling?

Was ist Profiling?

In der Regel:

- Programmcode unverändert.

living.knowledge
WWU Münster

Was ist Profiling?

In der Regel:

- Programmcode unverändert.
- Anderes Programm (“Profiler”) läuft parallel und beobachtet das getestete Programm

Was ist Profiling?

In der Regel:

- Programmcode unverändert.
- Anderes Programm ("Profiler") läuft parallel und beobachtet das getestete Programm
- Daten über den gesamten Programmablauf werden gesammelt

living.knowledge
WWU Münster

Welche Arten des Profiling gibt es?

Welche Arten des Profiling gibt es?

1. Sampling:
   Programm wird oft angehalten, Profiler notiert den callstack

living.knowledge WWU Münster

Welche Arten des Profiling gibt es?

1. Sampling:
   Programm wird oft angehalten, Profiler notiert den callstack
2. Instrumentierung:
   Das Executable wird geändert, so dass jeder Funktionsaufruf mitgeschrieben wird

Welche Arten des Profiling gibt es?

1. Sampling:
   Programm wird oft angehalten, Profiler notiert den callstack
2. Instrumentierung:
   Das Executable wird geändert, so dass jeder Funktionsaufruf mitgeschrieben wird
3. Simulation:
   Ein PC wird simuliert, und in diesem simulierten PC wird das Programm ausgeführt.

living.knowledge
WWU Münster

Typische Profiling-Ergebnisse

Die üblichen Ergebnisse des Profiling sind

1. Für alle Funktionen:
   - “Total”/ “Cummulative” Time
   - “Self” Time
   - ggf. Anzahl der Aufrufe
2. Callgraph
3. ggf. Verteilung der Laufzeit auf Quellcode-Zeilen

living.knowledge
WWU Münster

Profiling in C++

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

living.knowledge
WWU Münster

Programm für Profiling vorbereiten

- Immer mit Optimierung übersetzen.

Programm für Profiling vorbereiten

- Immer mit Optimierung übersetzen.
- Zusätzlich immer mit Debug-Information “-g”.

living.knowledge
WWU Münster

Programm für Profiling vorbereiten

- Immer mit Optimierung übersetzen.
- Zusätzlich immer mit Debug-Information “-g”.
- Für Instrumentierung zusätzlich mit “-pg”

Inlining abschalten

Problem: Optimierung entfernt Funktionsaufrufe.

Inlining abschalten

Problem: Optimierung entfernt Funktionsaufrufe.
Lösung: GCC-Kommandozeilenoption “-fno-inline” verhindert Inlining

living.knowledge
WWU Münster

Inlining abschalten

Problem: Optimierung entfernt Funktionsaufrufe.
Lösung: GCC-Kommandozeilenoption "-fno-inline" verhindert Inlining
Vorgehen dazu:

1. Gesamtlaufzeit messen
2. "-fno-inline" einbauen
3. Gesamtlaufzeit erneut messen
   → Effekt des nicht-Inlinens abschätzen
4. Profiling
5. Ergebnisse mit Vorsicht genießen

C++ Profiler Nr. 1: Intel VTune Amplifier

- Kommerzielles Produkt, wird von der ZIV für Windows und Linux bereitgestellt.

  https://zivdav.uni-muenster.de/ddfs/Soft.ZIV/Intel/Linux/VTune_Amplifier_XE/Version_2015/Initial_Release/

living.knowledge WWU Münster

C++ Profiler Nr. 1: Intel VTune Amplifier

- Kommerzielles Produkt, wird von der ZIV für Windows und Linux bereitgestellt.

  https://zivdav.uni-muenster.de/ddfs/Soft.ZIV/Intel/Linux/VTune_Amplifier_XE/Version_2015/Initial_Release/

- Kann vieles, vor allem “Hotspot”-Analyse (Sampling)

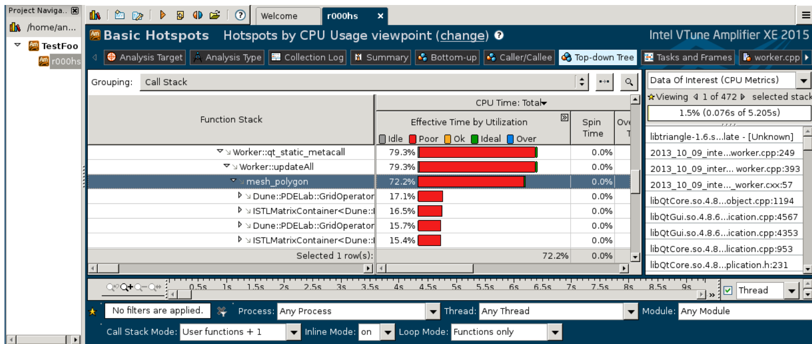
C++ Profiler Nr. 1: Intel VTune Amplifier

- Kommerzielles Produkt, wird von der ZIV für Windows und Linux bereitgestellt.

  https://zivdav.uni-muenster.de/ddfs/Soft.ZIV/Intel/Linux/VTune_Amplifier_XE/Version_2015/Initial_Release/
- Kann vieles, vor allem “Hotspot”-Analyse (Sampling)
- Schnell und einfach zu bedienen

living.knowledge
WWU Münster

C++ Profiler Nr. 1: Intel VTune Amplifier



living.knowledge
WWU Münster

C++ Profiler Nr. 1: Intel VTune Amplifier

Project Naviga.. /home/an... TestFoo r000hs

Welcome | r000hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames | worker.cpp

Source | Assembly | Assembly grouping: Address

S. Li.	Source	Effective Time (Idle / Poor / Ok)
244		0.0%
245	triangulate_output.edgelist = NULL;	0.0%
246	triangulate_output.edgemarkerlist = NULL;	0.0%
247		0.0%
248	char mystring[200] = "pa200qezYQ";	0.0%
249	triangulate(mystring, &triangulate_input, &triangulate_output, NULL);	1.1%
250		0.0%
251	edges.resize(triangulate_output.numberofedges);	0.0%
252	for (int i = 0; i < triangulate_output.numberofedges; ++i) {	0.0%
253	int firstpoint = triangulate_output.edgelist[2 * i];	0.0%
254	int secondpoint = triangulate_output.edgelist[2 * i + 1];	0.0%
255	edges[i] =	0.0%

Selected 1 row(s):

Data Of Interest (CPU Metrics)
Viewing 1 of 472 selected stack
1.5% (0.076s of 5.205s)

- libtriangle-1.6.s...late - [Unknown]
- 2013_10_09_inte...worker.cpp:249
- 2013_10_09_inter... worker.cpp:393
- 2013_10_09_inte..._worker.cxx:57
- libQtCore.so.4.8...object.cpp:1194
- libQtGui.so.4.8.6...ication.cpp:4567
- libQtGui.so.4.8.6...ication.cpp:4353
- libQtCore.so.4.8...lication.cpp:953
- libQtCore.so.4.8...plication.h:231
- libQtCore.so.4.8...cation.cpp:1533
- libQtCore.so.4.8...plication.h:236
- libQtCore.so.4.8...er_glib.cpp:287
- libglib-2.0.so....nown]:[Unknown]

living.knowledge
WWU Münster

C++ Profiler Nr. 2: gprof

- Freies Tool

C++ Profiler Nr. 2: gprof

- Freies Tool
- Basierend auf Instrumentierung

living.knowledge
WWU Münster

C++ Profiler Nr. 2: gprof

- Freies Tool
- Basierend auf Instrumentierung
- Schwäche: Visualisierung

C++ Profiler Nr. 2: gprof

Wie funktioniert es?

- Übersetzen mit "-pg" Option (und natürlich "-O3")

C++ Profiler Nr. 2: gprof

Wie funktioniert es?

- Übersetzen mit “-pg” Option (und natürlich “-O3”)
- Laufen lassen → “gmon.out” Datei entsteht

C++ Profiler Nr. 2: gprof

Wie funktioniert es?

- Übersetzen mit “-pg” Option (und natürlich “-O3”)
- Laufen lassen → “gmon.out” Datei entsteht
- Ergebnisse mit “gprof meinprogramm” anschauen

C++ Profiler Nr. 2: gprof

Wie sehen die Ergebnisse aus?

Flat Profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
```

living.knowledge
WWU Münster

C++ Profiler Nr. 2: gprof

Wie sehen die Ergebnisse aus?

Call Graph:

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     100.0    0.00    0.05                 start [1]
                 0.00    0.05       1/1           main [2]
                 0.00    0.00       1/2           on_exit [28]
                 0.00    0.00       1/1           exit [59]
-----------------------------------------------
                 0.00    0.05       1/1           start [1]
[2]     100.0    0.00    0.05       1         main [2]
                 0.00    0.05       1/1           report [3]
-----------------------------------------------
                 0.00    0.05       1/1           main [2]
[3]     100.0    0.00    0.05       1         report [3]
                 0.00    0.03       8/8           timelocal [6]
```

C++ Profiler Nr. 3: valgrind

- Simuliert eine CPU

living.knowledge
WWU Münster

C++ Profiler Nr. 3: valgrind

- Simuliert eine CPU
- Vorteil: 100% reproduzierbar, PC kann weiter benutzt werden

C++ Profiler Nr. 3: valgrind

- Simuliert eine CPU
- Vorteil: 100% reproduzierbar, PC kann weiter benutzt werden
- Nachteil: ca. 10x langsamer

C++ Profiler Nr. 3: valgrind

- Simuliert eine CPU
- Vorteil: 100% reproduzierbar, PC kann weiter benutzt werden
- Nachteil: ca. 10x langsamer
- Visualisierung mit “kcachegrind”

C++ Profiler Nr. 3: valgrind

So geht's:

1. Programm normal mit "-O3 -g" übersetzen

living.knowledge
WWU Münster

C++ Profiler Nr. 3: valgrind

So geht’s:

1. Programm normal mit “-O3 -g” übersetzen
2. “valgrind –tool=callgrind meinprogramm” laufen lassen

living.knowledge
WWU Münster

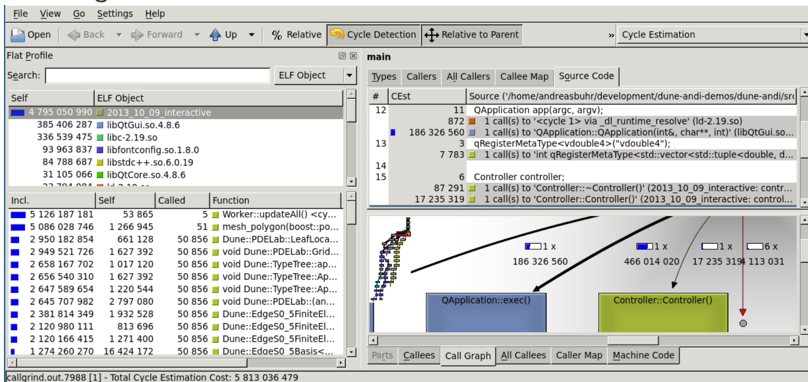
C++ Profiler Nr. 3: valgrind

So geht's:

1. Programm normal mit "-O3 -g" übersetzen
2. "valgrind –tool=callgrind meinprogramm" laufen lassen
3. Ergebnisse mit "kcachegrind" anschauen

C++ Profiler Nr. 3: valgrind

KCachegrind:



living.knowledge WWU Münster

Profiling in Python

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

Profiling in Python: Das cProfile Modul

- Arbeitsweise von cProfile entspricht am ehesten der Instrumentierung

Profiling in Python: Das cProfile Modul

- Arbeitsweise von cProfile entspricht am ehesten der Instrumentierung
- Profiling ist einfach: “python -m cProfile -o myoutput myscript.py”

living.knowledge
WWU Münster

Profiling in Python: Das cProfile Modul

- Arbeitsweise von cProfile entspricht am ehesten der Instrumentierung
- Profiling ist einfach: “python -m cProfile -o myoutput myscript.py”
- Interpretieren der Ergebnisse ist anstrengend:
  1. Möglichkeit: pstat-Modul
  2. Möglichkeit: RunSnakeRun
  3. Möglichkeit: gprof2dot
  4. Möglichkeit: snakeviz

living.knowledge WWU Münster

Profiling in Python

- Einfach: “python -m cProfile -o myoutput myscript.py”

living.knowledge
WWU Münster

Profiling in Python

- Einfach: “python -m cProfile -o myoutput myscript.py”
- Auch möglich: per API:

```
import cProfile
pr = cProfile.Profile()
pr.enable()
somefunction()
pr.disable()
pr.dump_stats("myoutput")
```

living.knowledge
WWU Münster

Profiling in Python: Visualisierung mit RunSnakeRun

- RunSnakeRun für Alltagseinsatz geeignet

living.knowledge WWU Münster

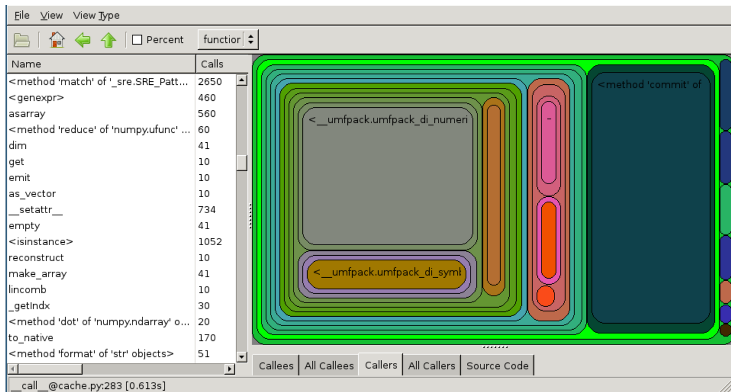
Profiling in Python: Visualisierung mit RunSnakeRun

- RunSnakeRun für Alltagseinsatz geeignet
- Installation unter Ubuntu: “sudo apt-get install runsnakerun”

living.knowledge
WWU Münster

Profiling in Python: Visualisierung mit RunSnakeRun

- RunSnakeRun für Alltagseinsatz geeignet
- Installation unter Ubuntu: “sudo apt-get install runsnakerun”



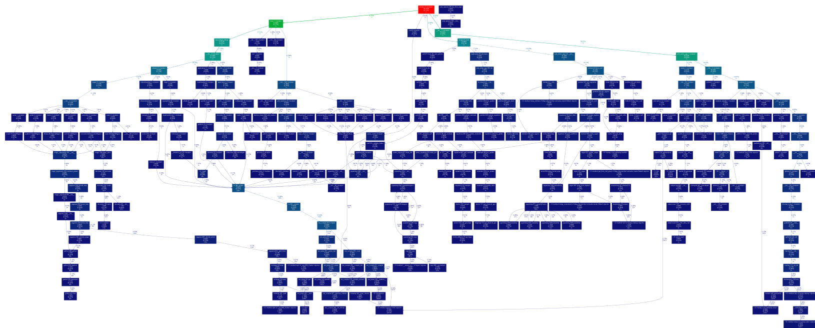
living.knowledge WWU Münster

Profiling in Python: Visualisierung mit gprof2dot

- Python-Script “gprof2dot” kann auch cProfile-output lesen
- Download: https://code.google.com/p/jrfonseca/wiki/Gprof2Dot
- Benutzung:
  gprof2dot.py -f pstats myoutput | dot -Tpng -o output.png
- Generiert ein riesiges Bild

living.knowledge
WWU Münster

gprof2dot Ausgabe



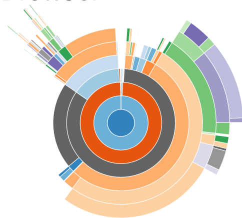
Profiling in Python: Visualisierung mit Snakeviz

- Snakeviz verfügbar über “pip install snakeviz”

living.knowledge
WWU Münster

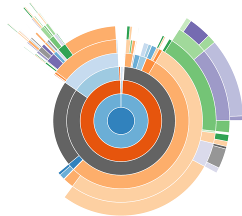
Profiling in Python: Visualisierung mit Snakeviz

- Snakeviz verfügbar über “pip install snakeviz”
- “snakeviz myoutput” startet Webserver, Ergebnisse im Browser



Profiling in Python: Visualisierung mit Snakeviz

- Snakeviz verfügbar über “pip install snakeviz”
- “snakeviz myoutput” startet Webserver, Ergebnisse im Browser



- Scheint leider nur für triviale Probleme performant genug zu sein (kapituliert vor pyMOR).

Profiling in Matlab

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

Profiling in Matlab

Matlab hat eingebauten Profiler:

- Start mit "profile on"

living.knowledge
WWU Münster

Profiling in Matlab

Matlab hat eingebauten Profiler:

- Start mit “profile on”
- Ende mit “profile off”

living.knowledge
WWU Münster

Profiling in Matlab

Matlab hat eingebauten Profiler:

- Start mit “profile on”
- Ende mit “profile off”
- Ergebnisse sehen mit “profile viewer”

living.knowledge
WWU Münster

Ein paar Empfehlungen

- Die beste Optimierung: Weniger tun!

Ein paar Empfehlungen

- Die beste Optimierung: Weniger tun!
- Unit-Tests schreiben

living.knowledge
WWU Münster

Ein paar Empfehlungen

- Die beste Optimierung: Weniger tun!
- Unit-Tests schreiben
- Problem reduzieren

Zusammenfassung

Agenda:

	C++	Python	Matlab
Zeitmessung	1.	2.	3.
Profiling	4.	5.	6.

- Drei Möglichkeiten zur Zeitmessung in C++
- Drei Möglichkeiten zur Zeitmessung in Python (+ timeit)
- Eine Möglichkeit zur Zeitmessung in Matlab (+ timeit)
- Drei Möglichkeiten zum Profiling in C++
- Drei Möglichkeiten zur Profiling-Visualisierung in Python
- Eine Möglichkeit zum Profiling in Matlab

living.knowledge
WWU Münster