

Framework zur grundlegenden Performanceanalyse

— Projektbericht —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Tim Jammer
E-Mail-Adresse:	3jammer@informatik.uni-hamburg.de
Matrikelnummer:	6527284
Studiengang:	Informatik
Betreuer:	Jannek Squar

Hamburg, den 27. Juni 2018

Abstract

Die Performance eines Systems erfolgreich einzuschätzen ist eine hohe Kunst. Daher kann dieser Prozess nicht automatisiert werden [Jai91]. Insbesondere ist es möglich aus den gleichen Daten verschiedene Schlussfolgerungen zu ziehen [Jai91].

Im Kontext der Performanceanalyse soll das entwickelte Framework eine Unterstützung für den ersten Schritt bieten und den Benutzer in seiner Entscheidung unterstützen, welche weiteren Schritte sinnvoll für eine effektive verbesserung der Performance sind.

Das Kapitel 2 bietet daher einen Überblick über verschiedene Tools, welche die Performance eines Systems in verschiedenen Bereichen messen.

Kapitel 3 geht auf die Herausforderungen der Performanceanalyse besonders im HPC-Bereich ein.

In Kapitel 4 wird das Konzept des entwickelten Frameworks zur grundlegenden Performanceanalyse dargestellt.

Kapitel 5 beschreibt daran anschließend die Benutzung des Frameworks.

Inhaltsverzeichnis

1	Einleitung	5
2	Übersicht über verschiedene Tools	6
2.1	Kategorisierung	6
2.2	Überblick	7
2.3	Einige ausgewählte Tools	7
2.3.1	/proc/cpuinfo und andere statische Informationen in /proc	7
2.3.2	top	8
2.3.3	df	8
2.3.4	iostat	10
2.3.5	nstat	10
2.3.6	perf	10
2.3.7	scoreP	10
3	Herausforderungen bei der Performance-Evaluation	12
3.1	Systematische Fehler	12
3.2	Zufällige Fehler	13
4	Überblick über das Framework-Konzept	14
4.1	Auswahl von Performance-Metriken	14
4.1.1	Datenformat	16
4.1.2	Beschreibung der Metriken	16
4.1.3	Einfügen neuer Metriken	18
4.1.4	Verschiedene Arten Metriken zu messen	19
4.1.5	Wrapper, um die Metriken zu messen	20
4.2	Analyse der gesammelten Daten	22
4.2.1	Analysemethodik	22
5	Benutzung des Frameworks	25
5.1	Aufruf und Funktionen des Programms zum Erstellen einer Konfigurationsdatei	25
5.2	Aufruf und Funktionen des Programms zur Messung	26
5.3	Aufruf und Funktionen des Programms zur Analyse	27
6	Zusammenfassung und Future Work	29
A	Messung des Overheads des Frameworks	30

Abbildungsverzeichnis	31
Listingverzeichnis	31
Literaturverzeichnis	32

1. Einleitung

"Performance can be bad, but can it ever be wrong?"

Jim Kohn, SGI/Cray Research, Inc.

Die Performance eines Systems erfolgreich einzuschätzen ist eine hohe Kunst. Daher kann dieser Prozess nicht automatisiert werden [Jai91]. Insbesondere ist es möglich aus den gleichen Daten verschiedene Schlussfolgerungen zu ziehen [Jai91].

Die Performance eines Systems hängt nicht nur von einem einzigen Parameter ab, vielmehr ist das Zusammenspiel zwischen den verschiedenen Komponenten des Systems entscheidend für die Performance. Daher ergibt sich die Performance eines Systems aus vielen verschiedenen Bereichen [Wik].

Im Kontext der Performanceanalyse soll das entwickelte Framework eine Unterstützung für die ersten Schritte bieten.

Ziel ist es, dass der Benutzer einen ersten Eindruck von der Performance des Systems bekommt. Mit diesem Eindruck ist es möglich andere weiterführende Tools wie z.B. Score-P (Abschnitt 2.3.7) oder Intel VTune gezielter zu nutzen. Dies ist sinnvoll, da diese Tools den Ablauf des Programms stärker beeinflussen (vgl. Kapitel 3). Intel Vtunes hat beispielsweise einen Overhead von bis zu 1,6. [Int17] Für ein möglichst gutes Ergebnis ist daher eine gezielte Nutzung solch umfangreicher Tools vorzuziehen, da eine solche Nutzung das System weniger beeinflusst. Beispielsweise ist es bei einer Anwendung mit sehr schlechter Cache-Performance nach dem Pareto Prinzip¹ viel zielführender die Cache-Performance zu verbessern als zunächst hohen Aufwand in die Analyse des Kommunikationsschemas zu stecken. Die in Abschnitt 4.2.1 dargestellte Analyse soll daher einen schnellen ersten Eindruck geben, wo sich mögliche Performance-Engpässe befinden und sich eine weitere Analyse daher am meisten lohnt.

Das Kapitel 2 bietet daher einen Überblick über verschiedene Tools, welche die Performance eines Systems in verschiedenen Bereichen messen.

Kapitel 3 geht auf die Herausforderungen der Performanceanalyse besonders im HPC-Bereich ein.

In Kapitel 4 wird das Konzept des entwickelten Frameworks zur grundlegenden Performanceanalyse dargestellt.

Kapitel 5 beschreibt daran anschließend die Benutzung des Frameworks.

¹auch 80/20 Regel

2. Übersicht über verschiedene Tools

Dieses Kapitel bietet einen Einblick in eine Auswahl häufig benutzter Tools und deren Fähigkeiten, um die Performance eines Systems besser verstehen zu können. Zunächst wird die vorgenommene Kategorisierung der Tools vorgestellt. Abschnitt 2.2 gibt einen Überblick über verschiedene Tools und deren Einordnung in die vorgestellten Kategorien, bevor die Tools in Abschnitt 2.3 kurz einzeln vorgestellt werden.

2.1. Kategorisierung

Man kann Tools, um die Performance eines Systems besser zu verstehen, in verschiedene Kategorien einteilen.

Zunächst bietet sich selbstverständlich eine Einteilung anhand der gemessenen Leistungscharakteristika an. Eine andere Möglichkeit der Kategorisierung ist der Einsatzzweck der jeweiligen Tools. [Gre15] unterscheidet Tools, die statische Informationen über ein System und dessen Leistungsfähigkeit sammeln, von Tools, die dem Monitoring des Systems, dem Profilen oder Tracen einer Anwendung dienen.

Statische Informationen geben Auskunft darüber, wie leistungsfähig ein System in der Theorie ist. Sie enthalten z.B. die verfügbare Anzahl an CPU-Kernen und deren maximale Taktfrequenz. Statische Informationen können von den Tools ermittelt werden auch ohne, dass Last auf den analysierten Ressourcen liegt. Statische Informationen allein sind oft nur wenig aufschlussreich. Man kann mit ihnen theoretische Abschätzungen der Performance durchführen. Erst in Kombination mit dynamisch gewonnenen Informationen über das tatsächliche Verhalten des Systems können sie dafür benutzt werden zu analysieren, welche Komponenten ihre Leistungsgrenze erreicht haben und für das Gesamtsystem möglicherweise einen Flaschenhals darstellen.

Monitoring-Tools dienen dazu, die aktuelle Auslastung von Systemressourcen zu überwachen. Man kann mit ihnen feststellen, welche Ressourcen besonders stark genutzt werden. Mit ihnen lässt sich häufig auch feststellen, welche Prozesse für die übermäßige Benutzung der Ressourcen verantwortlich sind.

Profiling-Tools halten üblicherweise die Performance des Systems für eine bestimmte Anwendung fest. Man kann anhand der gesammelten Daten die Performanz des Programmes analysieren. Im Gegensatz zu Tracing-Tools halten sie nicht den zeitlichen Ablauf der Ressourcenbenutzung fest. Sie geben daher einen Überblick über die Performance der Anwendung.

Tracing-Tools werden für genauere Analysen des Verhaltens einer Anwendung genutzt. Sie halten den zeitlichen Ablauf der Ressourcennutzung während des Programmlaufes fest, sodass später die einzelnen Phasen einer Anwendung analysiert werden können.

Üblicherweise belastet das Auslesen statischer Informationen das System nicht weiter. Die Nutzung von Tracing-Tools führt jedoch häufig zu einer erheblichen Mehrlast für das System, sodass man diese Kategorisierung auch als Einteilung ansehen kann, wie sehr die einzelnen Tools das System zusätzlich zu der zu analysierenden Anwendung(en) belasten.

Der nächste Abschnitt teilt einige bekannte Tools in diese Kategorien ein.

2.2. Überblick

	Statisch	Monitoring	Profiling	Tracing
CPU	/proc/cpuinfo	top	perf	scoreP
RAM	/proc/meminfo	top		scoreP
IO(Datensystem)	(df)	iostat		scoreP
Netzwerk	/proc/net	nstat		scoreP

Tabelle 2.1.: Kategorisierung verschiedener Tools zur Performanceanalyse

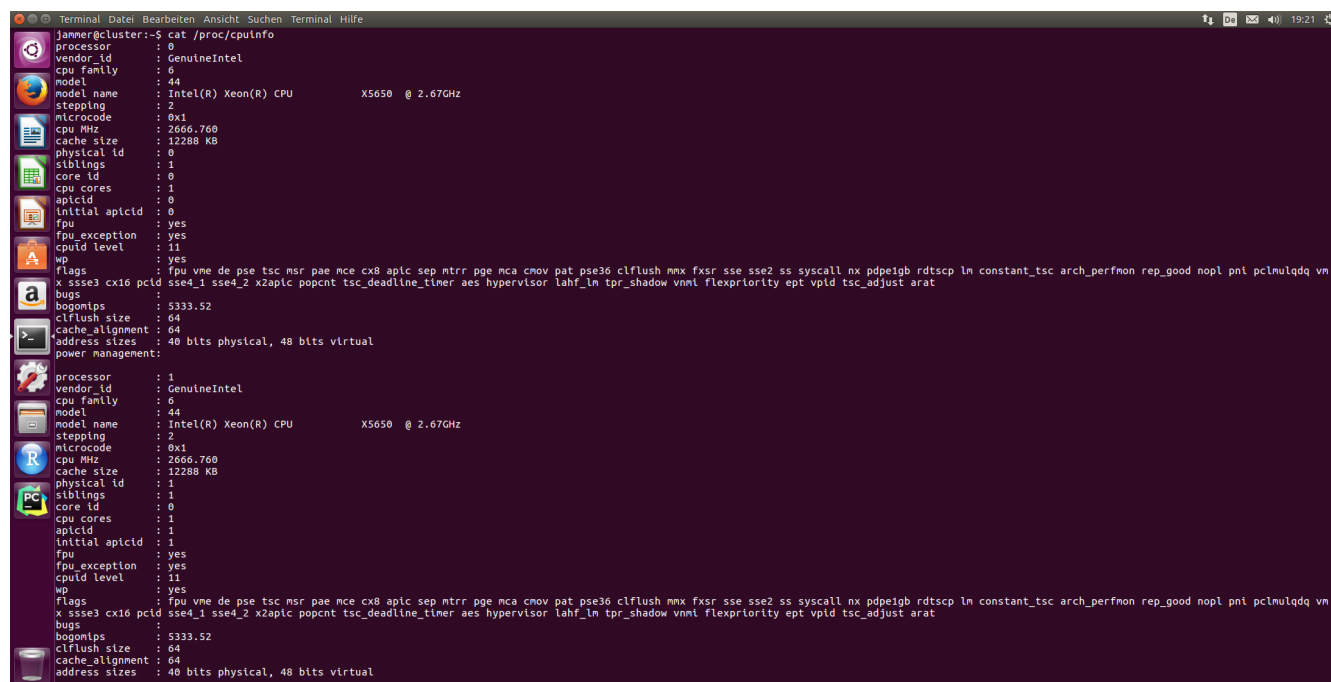
2.3. Einige ausgewählte Tools

In diesem Abschnitt werden die Tools aus Tabelle 2.1 kurz vorgestellt.

2.3.1. /proc/cpuinfo und andere statische Informationen in /proc

Unter Linux kann man unter dem Pfad /proc viele Informationen zum Kernel und den Hardwareressourcen erhalten. Die Abbildung 2.1 zeigt beispielsweise Informationen über die CPU auf dem Login-Knoten des Clusters. Man kann unter anderem noch folgende Informationen zum System erhalten:

- meminfo (Informationen über den verfügbaren Speicher)
- interrupts (Informationen über verschiedene aufgetretene Interrupts und deren Häufigkeit)
- net/dev (Informationen über die Netzwerk-Komponenten und deren Status)
- stat (eine Zusammenfassung verschiedener Statistiken)



```
jammer@cluster:~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 44
model name     : Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
stepping       : 2
microcode      : 0x1
cpu MHz        : 2666.760
cache size     : 12288 KB
physical id    : 0
siblings       : 1
core id        : 0
cpu cores      : 1
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 11
wp             : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep_good nopl pni pclmulqdq vm
x sse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes hypervisor lahf_lm tpr_shadow vnmi flexpriority ept vpid tsc_adjust arat
bugs           :
bogomips       : 5333.52
cache alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id      : GenuineIntel
cpu family     : 6
model          : 44
model name     : Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
stepping       : 2
microcode      : 0x1
cpu MHz        : 2666.760
cache size     : 12288 KB
physical id    : 1
siblings       : 1
core id        : 0
cpu cores      : 1
apicid         : 1
initial apicid : 1
fpu            : yes
fpu_exception  : yes
cpuid level    : 11
wp             : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep_good nopl pni pclmulqdq vm
x sse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes hypervisor lahf_lm tpr_shadow vnmi flexpriority ept vpid tsc_adjust arat
bugs           :
bogomips       : 5333.52
cache alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
```

Abbildung 2.1.: Ausgabe des Befehls `cat /proc/cpuinfo`

Man findet in `/proc` auch verschiedene Informationen zu einzelnen Prozessen und deren Status. Diese Informationen findet man in `/proc` im Unterordner der jeweiligen Prozess-ID.

2.3.2. top

`top` oder `htop` oder andere ähnliche Tools lassen sich der Kategorie der Monitoring-Tools zuordnen. Die Abbildung 2.2 zeigt die Ausgabe des `htop`-Befehls auf dem Login-Knoten des Clusters. Im oberen Bereich sieht man die aktuelle Auslastung der CPU-Kerne und des Arbeitsspeichers. Unten sind die einzelnen Prozesse und deren Anteil an der Nutzung der Systemressourcen. Die Daten werden laufend aktualisiert. `htop` erlaubt es einem zusätzlich noch die Prozesse anhand verschiedener Kriterien (wie z.B. die aktuelle CPU-Nutzung) zu sortieren sowie die Prozesse durch verschiedene Signale zu beeinflussen (z.B. zu beenden).

2.3.3. df

`df` zeigt Informationen über die verfügbaren Speichersysteme und den darin verfügbaren Speicherplatz. Das Tool wird in Abbildung 2.3 gezeigt. Man kann dort z.B. sehen, dass in `/def/vda1` noch 12 GB Speicherplatz frei sind.

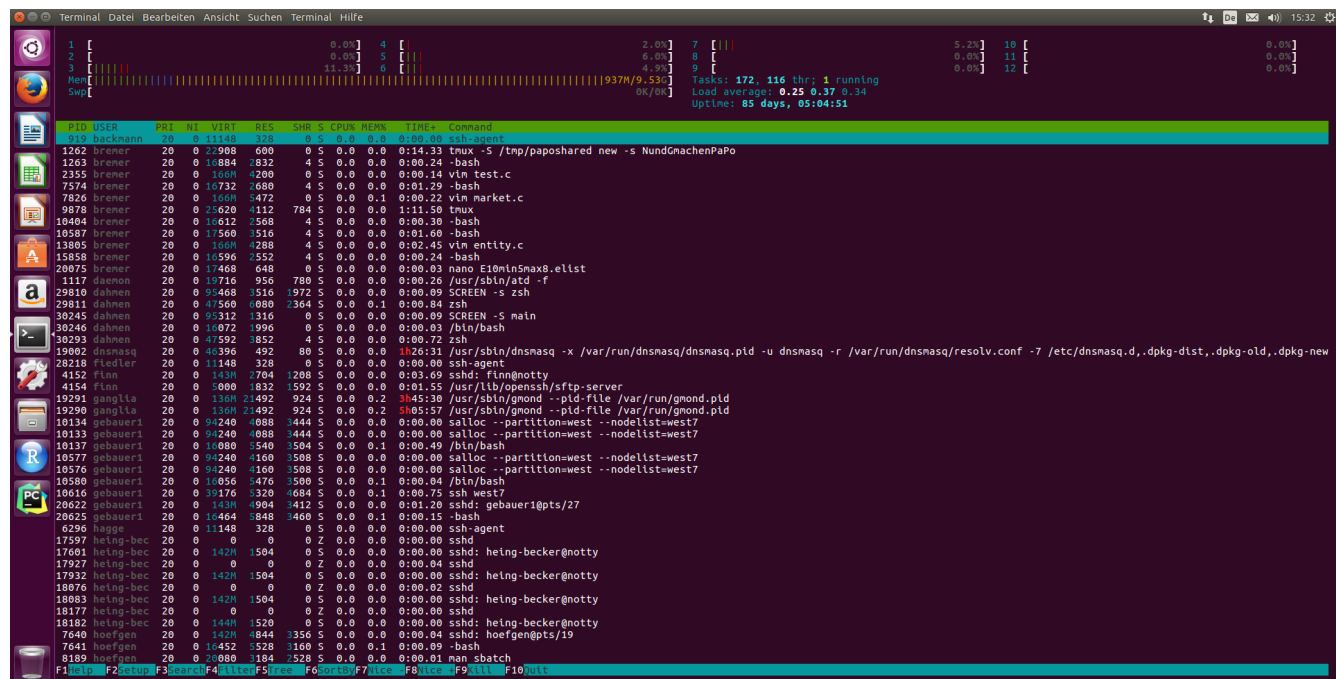


Abbildung 2.2.: Ausgabe des Befehls htop

```
jammer@cluster:~$ df -h
Filesystem                Size      Used Avail Use% Mounted on
udev                     4.8G         0   4.8G   0% /dev
tmpfs                    977M      110M   867M  12% /run
/dev/vda1                 20G       7.3G    12G  40% /
tmpfs                    4.8G       92M   4.7G   2% /dev/shm
tmpfs                    5.0M         0   5.0M   0% /run/lock
tmpfs                    4.8G         0   4.8G   0% /sys/fs/cgroup
10.0.0.200:/home/cluster-opt 1.7T      44G    1.7T   3% /opt
10.0.0.200:/home/cluster   5.5T      4.7T   843G  85% /home
10.0.0.200:/home/cluster-nodes 1.7T      7.0G    1.7T   1% /srv/nodes
10.0.0.200:/home/cluster/pool 1.9T      1.1T   843G  56% /home/pool
```

Abbildung 2.3.: Ausgabe des Befehls df -h

2.3.4. iostat

iostat zeigt, wie in Abbildung 2.4 illustriert, Statistiken darüber, wie stark die IO-Geräte aktuell beansprucht werden. Man sieht z.B., dass zuletzt mit 0.43 KB pro Sekunde gelesen wurde.

```
jammer@west1:~$ iostat
Linux 4.4.0-112-generic (west1) 02/14/2018 _x86_64_ (24 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.03    0.00    0.04    0.00    0.00   99.93

Device:            tps    kB_read/s    kB_wrtn/s    kB_read  kB_wrtn
sda                 0.12         0.43         6.59     277935   4279788
```

Abbildung 2.4.: Ausgabe des Befehls iostat

2.3.5. nstat

netstat zeigt Statistiken über die Netzwerknutzung. Die Abbildung 2.5 zeigt beispielsweise Statistiken zur Netzwerknutzung des Login-Knotens seit der letzten Ausführung von nstat (hier etwa eine Sekunde vorher). Z.B. wurden 55 TCP-Segmente über aktive Verbindungen empfangen. Je nach den Einstellungen lassen sich noch viele weitere Werte anzeigen und auslesen.

2.3.6. perf

perf ist ein Tool, um verschiedene Statistiken von Anwendungen zu profilieren. Die Abbildung 2.6 zeigt unter anderem Statistiken zur CPU-Nutzung, wie beispielsweise, dass es 28,991,127,772 Sprünge im Programmablauf gab.

2.3.7. scoreP

Score-P (Scalable Performance Measurement Infrastructure for Parallel Codes) ist ein Softwaresystem, das eine Infrastruktur bietet, eine Vielzahl von Systemparametern zu messen. Das System sammelt sehr umfangreiche Daten und kann über Plugins erweitert werden.

Um die umfangreichen Daten zu visualisieren, kann man z.B. Vampir verwenden. Ein wichtiger Einsatzzweck ist beispielsweise die Analyse des Kommunikationsschemas, da man diese in Vampir gut visualisieren kann. Da sehr umfangreiche Trace-Dateien erzeugt werden, wird der Programmablauf vom Einsatz von scoreP allerdings erheblich beeinflusst.

```
jammer@cluster:~$ nstat
#kernel
IpInReceives          55          0.0
IpForwDatagrams        12          0.0
IpInDelivers           43          0.0
IpOutRequests          28          0.0
TcpInSegs              10          0.0
TcpOutSegs             12          0.0
UdpInDatagrams         33          0.0
UdpOutDatagrams         4          0.0
TcpExtTCPHPHits        1          0.0
TcpExtTCPPureAcks       4          0.0
TcpExtTCPHPAcks         3          0.0
TcpExtTCPOrigDataSent  11          0.0
IpExtInOctets          12210       0.0
IpExtOutOctets          16746       0.0
IpExtInNoECTPkts       55          0.0
```

Abbildung 2.5.: Ausgabe des Befehls nstat

```
jammer@cluster:~/HLR/meineLoesung/Blatt3/pde$ perf stat ./partdiff-seq 1 1 512 2 2 100
=====
Program for calculation of partial differential equations.
=====
(c) Dr. Thomas Luedig, TU München.
    Thomas A. Zochler, TU München.
    Andreas C. Schmidt, TU München.
=====
Berechnungszzeit: 55.501720 s
Berechnungsmethode: Gauss-Seidel
Interlines: 512
Stoerfunktion: f(x,y)=2p1*2*sin(p1*x)*sin(p1*y)
Terminierung: Anzahl der Iterationen
Anzahl Iterationen: 100
Norm des Fehlers: 5.859487e-07
Matrix:
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0001 0.0001 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0001 0.0001 0.0001 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
Performance counter stats for './partdiff-seq 1 1 512 2 2 100':
55920.749297 task-clock (msec)    # 0.999 CPUs utilized
146          context-switches    # 0.003 K/sec
0           cpu-migrations       # 0.000 K/sec
32,984      page-faults          # 0.590 K/sec
0           cycles                # 0.000 GHz
74,402,097,266 stalled-cycles-frontend # 0.00% frontend cycles idle
15,307,119,320 stalled-cycles-backend  # 0.00% backend cycles idle
220,843,849,967 instructions      # 0.33 stalled cycles per insn
28,991,127,772 branches           # 518.432 M/sec
341,600,472 branch-misses        # 1.18% of all branches

55.951491620 seconds time elapsed
jammer@cluster:~/HLR/meineLoesung/Blatt3/pde$
```

Abbildung 2.6.: Ausgabe des Befehls perf stat

Der obere Teil mit der Matrix gehört zur Ausgabe des ausgeführten Programmes und ist nicht Teil von perf.

3. Herausforderungen bei der Performance-Evaluation

In diesem Kapitel sollen mehrere Herausforderungen beim Evaluieren der Performance eines Systems betrachtet werden. Nach [Lil05] kann man die auftretenden Fehler in systematische Fehler und zufällige Fehler einteilen.

3.1. Systematische Fehler

Systematische Fehler entstehen durch häufig kontrollierbare Einflüsse auf die Messung. Allen voran ist hier die Messung selbst zu betrachten, da die gemessenen Informationen abgespeichert werden müssen. Das Auslesen und Abspeichern dieser Informationen kostet Zeit, die der eigentlichen Anwendung nicht mehr zur Verfügung steht. Der Programmablauf wird oft durch das Erheben von Informationen gestört, wodurch es nicht möglich ist unverfälschte Daten zu bekommen. Verschiedene Arten des Messens von Ereignissen beeinflussen das System dabei auf unterschiedliche Weise (vgl. Abschnitte 2.1 und 4.1). Allerdings ist nicht nur die Messung selbst für systematische Fehler verantwortlich. Auch andere externe Einflüsse, wie z.B. die Temperatur, können ein System beeinflussen indem sie beispielsweise für eine Anpassung der Taktrate sorgen.¹

Bei systematischen Fehlern ist es von hoher Bedeutung, dass man sich dieser Einflüsse bewusst ist. Sofern möglich kann man sie dann gezielt steuern und eventuell ihren Einfluss benennen.

¹Im HPC-Kontext ist der Faktor der Temperatur meist durch entsprechende Kühlanlagen möglichst konstant gehalten und daher nicht als systematischer Messfehler relevant. Insbesondere bei mobilen Systemen, die häufig keine aktive Kühlung haben, ist die Temperatur aber ein höchst relevanter Umwelteinfluss. [Tit17] [GSvB⁺16]

3.2. Zufällige Fehler

Zufällige Fehler sind im Gegensatz zu systematischen (vgl. Abschnitt 3.1) weder vorhersehbar noch kontrollierbar. Beispiele hierfür sind (physikalische) Messungenauigkeiten wie z.B. die Auflösung der eingesetzten Timer. Bei einer Samplingrate von einem Hertz ist es nicht unbedingt gegeben, dass zwischen zwei Samples exakt "das 9.192.631.770-fache der Periodendauer der Strahlung, die dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustandes von Atomen des Nuklids ^{133}Cs entspricht"² vergeht. Hierdurch können insbesondere bei der Kombination verschiedener Messdaten und deren Zuordnung zu bestimmten Samples Fehler entstehen (vgl. [Jam16]).

Man beachte, dass im HPC-Bereich auch kleine Ungenauigkeiten schnell zu fehlerhaften Interpretationen führen können, da im HPC-Bereich auch kleinste Verbesserungen der Performance erwünscht sind. Es ist bei einer Anpassung, die minimale Verbesserung bringt, daher schwierig zu zeigen, dass es wirklich eine statistisch signifikante Verbesserung gibt. Möglicherweise tritt die Verbesserung nicht mehr auf, wenn die Anwendung nicht mehr durch eine Messung beeinflusst wird.

Weitere Herausforderungen ergeben sich im komplexen Zusammenspiel aller Systemkomponenten, welches letztendlich für die resultierende Performance verantwortlich ist. Einzelne Aspekte können daher nur schwer für sich betrachtet werden und müssen immer in Bezug zu den anderen Systemkomponenten gesetzt werden. Dies steigert die Komplexität der Performanceanalyse erheblich. [Lil05] Das gilt insbesondere im HPC-Bereich, da dort nicht nur das Zusammenspiel der zahlreichen Komponenten eines Rechenknotens betrachtet werden müssen, sondern zusätzlich auch noch das Zusammenwirken mit allen anderen Rechenknoten.

²SI Definition einer Sekunde nach [dPeM06]

4. Überblick über das Framework-Konzept

Um Werte über verschiedene Komponenten zusammenzuführen benötigt man mehrere Tools. Daher soll ein Framework gestaltet werden, dass Daten von verschiedenen Tools sammeln und zusammenführen kann.

Das Framework soll aus drei wesentlichen Teilen bestehen:

Erreichte Performance bestimmen

Für eine Anwendung soll es möglich sein, die erreichte Performance der Anwendung zu bestimmen.

Theoretisch maximale Performance ermitteln

Um die ermittelte Performance der Anwendung bewerten zu können, ist es vonnöten, die theoretisch maximal mögliche Performance der einzelnen Werte zu bestimmen. Hierfür sollen Hardwareinformationen ausgelesen werden. Als zukünftige verbesserung kann man die gewonnenen Informationen zusätzlich durch das Ausführen von spezialisierten Benchmarks validieren.

Auslastung bestimmen

Wenn man die erreichte Performance einer Anwendung bestimmt hat, soll es möglich sein, die Auslastung (utilisation) der einzelnen Systemkomponenten zu berechnen. Auf dieser Basis soll das Framework automatisch einen möglichen Flasschenhals für die Leistung des Programmes finden. Ziel ist es, dem Nutzer nach Möglichkeit vorzuschlagen wie man die Performance genauer analysieren kann, um die Leistung an dem identifizierten Flaschenhals zu verbessern.

4.1. Auswahl von Performance-Metriken

Zunächst sollen folgende Systemcharakteristika ermittelt werden:

- CPU-Operationen
- Cache miss rate
- Brance miss rate
- IO-Durchsatz

- Netzwerkdurchsatz

Nach [Lil05] müssen gute Performance-Metriken folgende 6 Eigenschaften haben:

1. **Linearität:** Beim Umgang mit Computersystemen ist ein linearer Zusammenhang zwischen der tatsächlichen Performance und der betrachteten Metrik gewünscht, da dieser einfacher verständlich ist.
2. **Verlässlichkeit:** Es ist auch zu beachten, dass die Ergebnisse einer Metrik verlässlich sind. Beispielsweise wären FLOPS¹ keine geeignete Metrik um verlässlich die Performanz eines Programmes zu bewerten, da ein Programm mit eineren niedrigeren Anzahl FLOPS beispielsweise durch geschickte Wahl der Algorithmen auch mit weniger Operationen Ergebnis kommen kann und daher insgesamt als Performanter zu bewerten ist.
3. **Wiederholbarkeit:** Es ist außerdem wichtig, dass die gleiche Messung der Metrik wiederholt zu einem gleichen Ergebnis führt.
4. **Einfache Messung:** Eine Metrik sollte einfach zu messen sein, denn dies ermöglicht es den Herausforderungen aus Kapitel 3 durch das Verstehen der Messung besser zu begegnen. Außerdem ist dies nötig um bewerten zu können, ob eine Messung fehlerhaft durchgeführt wurde. Dabei ist es auch denkbar, dass eine Metrik aus anderen einfach zu messenden Metriken abgeleitet wird.
5. **Konsistenz:** Bei einer konsistenten Metrik bedeutet eine Verbesserung um einen bestimmten Wert auch in verschiedenen Kontexten eine reale Verbesserung um einen korrespondierenden Wert. FLOPS ist beispielsweise bei einem Programm, das gar keine Fließkommaarithmetik benutzt, keine sinnvolle Metrik.
6. **Unabhängigkeit:** Eine Metrik sollte unabhängig von ihrem Einsatzzweck definiert sein, sodass sie nicht das misst, was gezeigt werden soll, sondern eine unabhängige Einschätzung des Systems liefert.

Die Auswahl der Metriken, die vom Framework erhoben werden, basiert zum einen auf den oben dargestellten Kriterien für gute Performance-Metriken. Zum anderen werden diese Metriken gewählt, da die Messung dieser Metriken das System möglichst wenig beeinflusst (vgl. Abschnitt 3.1).

Außerdem werden nur solche Metriken erhoben, bei denen es vergleichsweise einfache Techniken gibt, die Performanz zu verbessern. Z.B. kann die Reihenfolge von Array-Indizes die Cache-Performance stark verbessern [opt14]. Bei anderen Metriken ist es nicht einfach Techniken abzuleiten, wie die Performanz des Systems verbessert werden kann. Beispielsweise gibt es keine einfachen Techniken, eine Anwendung zu optimieren, um den RAM-Durchsatz eines Systems zu verbessern. [Tho17] Die Cache-Performance ist hier eine viel aussagekräftigere Metrik, anhand derer man konkrete Verbesserungsschritte ableiten kann.

¹Floting Point Operations Per Second

Die Auswahl dieser Systemcharakteristika ermöglicht es außerdem, dass alle diese Werte in Anzahl / Sekunde bzw. in Byte /Sekunde oder direkt in Prozent gemessen werden können. Zusätzlich ist es für diese Werte einfach, die theoretisch maximal erreichbare Performance anhand von Hardwareinformationen zu bestimmen.

Es ist nicht vorgesehen andere Einheiten zu unterstützen.

Ziel ist es, das Framework so zu gestalten, dass es für die Analyse weitere Systemcharakteristika erweitert werden kann. Siehe dafür Abschnitt 4.1.3

4.1.1. Datenformat

Die Metriken werden für eine Messung in einer Datei gespeichert. Die Datei enthält als erste Zeile eine Beschreibung der Messung (gemessenes Programm+Timestamp+gemessenerKnoten). Darunter folgt eine Metrik je Zeile. Es wird ein Double je Metrik gespeichert in der Einheit Anzahl bzw. Bytes je Sekunde oder Prozent, je nach Metrik. Die Zahl wird zur besseren Übersicht in menschenlesbarer Textform gespeichert. Um Vergleichbarkeit zu vereinfachen wird es vom Tool nicht unterstützt, in der Datei andere Basiseinheiten zu speichern. Die Zahl wird gefolgt, vom Namen der Metrik. Jede Zeile enthält eine Metrik. Jede Metrik hat eine eindeutige ID, sodass die Zeile der Metrik angibt, welche Metrik enthalten ist. Es können dabei möglicherweise Leerzeilen entstehen, sodass am Ende der Datei ein Hinweis eingefügt wird, mit welcher Version der Implementation des Frameworks die Messung gemacht wurde, und dass eventuelle Leerzeilen nicht entfernt werden sollen.

Außerdem wird es möglich sein, mehrere Messungen durchzuführen und alle gemessenen Werte sowie die durchschnittlichen Werte in einer Tabulator-separierten² Datei zu speichern.

4.1.2. Beschreibung der Metriken

Die zur Konfiguration einer Metrik benötigten Daten werden in einem Struct gespeichert.

Listing 4.1: Definition des metric_struct.

```
1 struct metric_struct {
2     int id;
3     int standard;
4     int call_type_flag;
5     start_monitor monitor_function;
6     profile profiling_function;
7     double maximum;
8     int margin;
9     int positive;
```

²man kann in der constants.h Datei auch jedes Belibige andere Symbol als Seperator Definieren.


```

10     char* keyword;
11     char* name;
12     char* help;
13     void* arguments;
14     clear clear_function; };
15 typedef struct metric_struct metric;

```

Listing 4.1 Zeigt die Definition des `metric_struct`. Die einzelnen Datenfelder werden im Folgenden erläutert:

`id` Eindeutige ID der Metrik.

`standard` Flag, das anzeigt, ob diese Metrik standardmäßig gemessen werden soll.

`call_type_flag` Flag, das verschiedene Arten von Aufrufen anzeigt (siehe Abschnitt 4.1.5).

`monitor_function` Aufruf, der die Messung der Metrik veranlasst (siehe Abschnitt 4.1.5).

`profiling_function` Aufruf, der die Messung der Metrik veranlasst (siehe Abschnitt 4.1.5).

`maximum` Die theoretisch mögliche Leistung dieser Metrik.

`margin` Toleranz (in Prozent) gibt an, welche Abweichung der Messergebnisse man erwartet. In der Analyse der Daten wird dieser Wert verwendet, um festzustellen, ob es Abweichungen in den Messergebnissen verschiedener Samples gibt. (vgl. Abschnitt 4.2.1).

`positive` Flag, das anzeigt, ob sich ein hoher Wert positiv auf die Performance auswirkt. Bei nicht gesetztem Positive-Flag wird ein Wert von 0 als gut angenommen.

`keyword` Schlüsselwort, mit dem man die Messung der Metrik an- oder abschalten kann. (siehe Abschnitt 5.2)

`name` Name, der in die Mess-Datei eingefügt wird.

`help` Beschreibung für die Hilfe.

`arguments` Argumente, die beim Aufruf zur Messung der Metrik übergeben werden (siehe auch Abschnitt 4.1.5).

`clear_function` Funktion, die aufgerufen wird, bevor der Speicher des `metric_struct` freigegeben wird, falls `arguments` nicht NULL ist. Die Funktion soll den durch `arguments` belegten Speicherbereich freigeben.

4.1.3. Einfügen neuer Metriken

Das Framework soll es ermöglichen das Messen weiterer Metriken einfach einzufügen. Die dafür nötigen Schritte sind in diesem Abschnitt dargestellt.

Zum Initialisieren des in Abschnitt 4.1.2 dargestellten `metric_struct` wird eine neue Funktion benutzt:

```
1 metric** initNEW_metrics(unsigned int*  
    ↪ result_count, GKeyFile* config_file, int start_ID);
```

Es ist vorgesehen, dass diese Funktion in der Datei `metric_definition.h` bekanntgemacht wird. Um die neu definierte Funktion zu benutzen, ist es noch nötig einen neuen Aufruf der `fetch_metric_definition` Funktion mit der neuen `initNEW_metrics` Funktion als ersten Parameter einzufügen. Dieser Aufruf soll an der mit Kommentar entsprechend markierten Stelle innerhalb der in `metric_definition.c` definierten Funktion `init_metrics()` erfolgen.

Wenn die neu definierte Funktion `initNEW_metrics` so implementiert ist, dass sie alle nötigen `metric_struct` allokiert und mit entsprechenden Werten initialisiert, sowie `result_count` entsprechend setzt, sind die neuen Metriken damit nach erneutem Kompilieren bereit zur Verwendung.

Die Funktionsweise der neu zu implementierenden Funktion wird hier dargestellt:

```
1 metric** initNEW_metrics(unsigned int*  
    ↪ result_count, GKeyFile* config_file, int start_ID);
```

Als Parameter werden vom Framework übergeben:

`result_count` Es wird ein Pointer auf die Speicherstelle übergeben, die gleich der Anzahl der initialisierten Metriken gesetzt werden soll (Ausgabe Parameter).

`config_file` Es wird die Konfigurationsdatei als !Key-value REFEREN GLIB C! übergeben. Man kann aus dieser Datei die Konfiguration der Metrik auslesen (z.B. um das Standardfeld im `metric_struct` entsprechend zu initialisieren)

`start_ID` Es wird die ID übergeben, mit der die ID der ersten Metrik initialisiert werden soll. Für jede weitere Metrik soll die ID dann um 1 inkrementiert werden. Die ID einer Metrik ist nicht statisch festgelegt, da sich die Gesamtzahl der Metriken beispielsweise mit der Zahl der CPU-Kerne ändern kann.

Der Rückgabewert ist ein Pointer, der auf einen Speicherbereich zeigt, indem `result_count` viele Pointer auf entsprechend Abschnitt 4.1.2 initialisierte `metric_struct` zu finden sind. Der Speicherbereich mit den Pointern, sowie die Speicherbereiche der einzelnen `metric_struct` werden, sobald diese nicht mehr benötigt werden mit `free()` wieder freigegeben. Falls das Feld `arguments` gesetzt wurde (d.h. nicht NULL ist) wird vor der Freigabe des `metric_struct` die gesetzte `clear_arguments` Funktion aufgerufen. Bei

der Freigabe des `metric_struct` werden auch die drei Felder vom Typ `char*` freigegeben, sodass es wichtig ist, diese Strings mit `malloc` zu allokieren und nicht nur ein Stringliteral anzugeben.

Die Implementation der monitor- bzw. profiling- und maximum-Funktion wird im folgenden Abschnitt 4.1.5 dargestellt. Ein Template, wie eine solche Implementation aussehen kann, liegt in der Datei `metric1.c`.

4.1.4. Verschiedene Arten Metriken zu messen

Nach [Lil05] werden Metriken immer Event-basiert erhoben. Dabei kann man zwischen vier verschiedenen Messstrategien unterscheiden.

1. **Ereignisgesteuert** Bei einer ereignisgesteuerten Messung werden die benötigten Daten (im einfachsten Fall das Zählen des eingetretenen Events) bei jedem Eintreten des interessanten Ereignisses bzw. der interessanten Ereignisse erfasst. Dies hat zur Folge, dass das eigentliche Ereignis verändert wird zu "Erfasse Informationen und bearbeite dann das Ereignis". Die Beeinflussung des Systems durch eine ereignisgesteuerte Messung ist daher abhängig von der Häufigkeit der beobachteten Ereignisse.
2. **Tracing** Tracing kann man gut mit ereignisgesteuerter Messung vergleichen. Es werden bei Eintritt der interessanten Ereignisse allerdings noch zusätzliche Informationen zum Systemzustand, beispielsweise die Funktion in der sich das Programm gerade befindet, erfasst. Tracing wird dazu verwendet das Eintreten bestimmter Ereignisse z.B. cache-misses im Kontext mit anderen Systeminformationen zu betrachten, beispielsweise um die cache-miss-rate verschiedener Funktionen des Programms zu vergleichen. Da bei jedem Eintreten des Ereignisses zusätzliche Kontext-Informationen erfasst werden, wird der Systemablauf durch eine solche Form der Messung stärker gestört.
3. **Sampling** Beim Sampling wird der Systemzustand nicht bei einem bestimmten Ereignis betrachtet, sondern wiederholt nach einer festgelegten Zeitspanne (Samplingrate). Vorteil von einer solchen Strategie ist, dass man die Störung im Systemablauf durch die Festlegung einer geeigneten Samplingrate besser steuern kann. Dies geht bei einer zu geringen Samplingrate aber möglicherweise auf Kosten der Genauigkeit, da man kurzzeitige Änderungen im Systemzustand nicht erfassen kann.
4. **Indirekte Messung** Bei einer indirekten Messung leitet man eine Metrik von einer oder mehrerer anderer, direkt messbaren Metriken ab. Beispielsweise wird die cache-miss-rate aus den ereignisgesteuert messbaren Metrikenanzahl der cache-hits sowie Anzahl der cache-misses abgeleitet.

4.1.5. Wrapper, um die Metriken zu messen

Die Metriken werden durch verschiedene Wrapper-Programme gemessen. Dafür müssen im `metric_struct` angegeben werden:

- ein Aufruf, der die Messung der Metrik veranlasst
- ein Flag, das verschiedene Arten von Aufrufen anzeigt

Es werden zwei Arten von Messungen unterstützt:

Monitoring

Beim Monitoring wird zunächst der Aufruf ausgeführt, der das Starten der Messung veranlasst. Anschließend wird der zu messende Befehl ausgeführt und nach dessen Abschluss erfolgt der Aufruf, der die Messung beendet. Entsprechend der Einstellung des Benutzers (siehe Abschnitt 5.2) werden mehrere Monitoring-Tools gleichzeitig benutzt.

Der Monitoring-Modus wird benutzt, wenn im `metric_struct` das Feld `call_type_flag` auf den Wert `CALL_TYPE_MONITOR` gesetzt ist.

Listing 4.2: Definition des Funktionsaufrufes zum monitoring

```
1 #define CALL_TYPE_MONITOR 1
2
3 typedef rstruct* (*stop_monitor)(void*,void*);
4 struct stop_monitor_struct {
5     stop_monitor stop_call;
6     void* args;
7 };
8 typedef struct stop_monitor_struct*
   ↪ (*start_monitor)(char*,void*);
```

In Listing 4.2 sind die für den Monitoring-Aufruf benötigten Definitionen gegeben. Im Monitoring-Modus wird der Funktionspointer, der im Feld `monitor_function` gegebene Funktionspointer aufgerufen. Wie Zeile 8 in Listing 4.2 zeigt, muss diese Funktion als Rückgabewert einen Pointer auf ein `stop_monitor_struct` liefern und nimmt als ersten Parameter den Namen des Programmes, das später ausgeführt werden soll³,

Als zweiter Parameter wird der im Feld `arguments` stehende Pointer übergeben (vgl. Abschnitt 4.1.2).

Das zurückgegebene `stop_monitor_struct` enthält dann den Aufruf, der zum Beenden des Monitorings benutzt wird (vgl. Zeile 5 im Listing 4.2), sowie einen Zeiger auf Argumente, die dieser Funktion übergeben werden sollen (vgl. Zeile 6 im Listing 4.2).

Der `stop_call` nimmt als ersten Parameter den Pointer aus dem `stop_monitor_struct`

³Falls man nur diesen Prozess überwachen möchte. Alle implementierten Metriken mit Monitor-Aufruf sammeln aber systemweite Daten.

und als zweiten Parameter den `arguments` Pointer aus der Metrik-Definition. Als Rückgabe wird ein Pointer auf ein `rstruct` verlangt (vgl. Zeile 3 im Listing 4.2).

Dieses `rstruct` wird unten im Abschnitt Ergebnisübergabe beschrieben.

Der Speicherbereich des `stop_monitor_struct` wird anschließend an den `stop_call` mit `free()` freigegeben. Der Speicherbereich, auf den mit `args` verwiesen wird, wird nicht gesondert freigegeben, Dies ist Aufgabe der aufgerufenen Funktion zum Stoppen der Messung.

Profiling

Im Gegensatz zum Monitoring ist hier der Wrapper auch für das Ausführen des eigentlichen Befehls verantwortlich. Der Profiling-Modus wird verwendet, falls im `metric_struct` das Feld `call_type_flag` auf den Wert `CALL_TYPE_PROFILE` gesetzt ist.

Listing 4.3: Definition des Funktionsaufrufes zum Profiling

```
1 #define CALL_TYPE_PROFILE 2
2
3 typedef rstruct* (*profile)(char*,void*);
```

Der im Feld `profiling_function` gegebene Funktionsaufruf ist, wie in Listing 4.3 dargestellt, analog dem Aufruf zum Starten des Monitorings (vgl. Zeile 8 im Listing 4.2), mit dem Unterschied, das als Rückgabewert direkt ein Pointer auf ein `rstruct` gefordert wird. Dieses `rstruct` wird unten im Abschnitt Ergebnisübergabe beschrieben.

Als ersten Parameter nimmt die Funktion den auszuführenden Befehl entgegen.

Als zweiter Parameter wird der im Feld `arguments` stehende Pointer übergeben (vgl. Abschnitt 4.1.2).

Ergebnisübergabe

Sowohl beim Monitoring als auch beim Profiling werden die Ergebnisse am Ende der Messung in einem `rstruct` übergeben.

Listing 4.4: Definition des `rstruct`

```
1 struct result_struct {
2     int resultcount;
3     int* id_list;
4     double* result_list;
5 };
6 typedef struct result_struct rstruct;
```

Die Definition ist in Listing 4.4 gezeigt. Das Struct enthält als `result_count` die Anzahl der zurückgegebenen Messwerte. Diese können auch von unterschiedlichen Metriken sein. Daher enthält das Struct zwei Arrays⁴ mit jeweils der Länge `result_count`. Die Werte in der `id_list` zeigen durch Angabe der Metrik-ID an, zu welcher Metrik der Messwert

⁴Pointer auf die Speicherbereiche dieser Arrays

gehört, der an der gleichen Position in `result_list` steht. Da die IDs der Metriken variabel sein können (vgl. Abschnitt 4.1.3) ist es zum Teil vonnöten, die IDs der Metriken, zu denen die Messwerte gehören, in den im Feld `arguments` (vgl. Abschnitt 4.1.2) gespeicherten Argumenten zu übergeben. Sowohl die beiden Arrays als auch das `rstruct` wird mit `free()` freigegeben, sobald es nicht mehr benötigt wird.

Die Unterscheidung zwischen Monitoring und Profiling ist dabei durch die unterschiedlichen Arten des Aufrufs von Tools bedingt. Beim Monitoring je einen Aufruf für das Starten und Beenden einer Messung, gegenüber einem einzigen Aufruf beim Profiling, der dann auch den zu messenden Befehl ausführen soll. Die Unterscheidung zwischen Monitoring und Profiling muss nicht unbedingt die in Abschnitt 4.1.4 vorgestellte Unterscheidung der verschiedenen Arten eine Metrik zu messen widerspiegeln.

4.2. Analyse der gesammelten Daten

Teil des Projektes ist es auch die gesammelten Daten zu analysieren, um dem Benutzer Hinweise geben zu können, welche Bereiche besonders interessant sind bzw. wo sich eine weiterführende Analyse mit anschließender Optimierung wahrscheinlich lohnt.

Die Analyse der gesammelten Daten erfolgt mittels eines eigenen Programms nach dem Sammeln der Daten. Daher ist es auch denkbar Daten zu analysieren, die mit anderen Tools z.b. Diamond [dia] gesammelt wurden.

4.2.1. Analysemethodik

Eine Methode für diesen ersten Analyseschritt ist es, die gemessene Systemauslastung mit der erwarteten Auslastung zu vergleichen, um festzustellen wo sich eventuelle Abweichungen ergeben. Diese Methodik soll durch eine automatische Aufbereitung der gemessenen Daten unterstützt werden.

Die Daten werden dabei in 4 Schritten analysiert:

1. Überprüfen, ob ein bestimmtes nicht optimales Programmverhalten vorliegt

Im ersten Schritt wird überprüft, ob ein bekanntes, häufig vorkommendes, nicht optimales Verhalten des Programms vorliegen könnte, damit der User dies optimieren kann. Konkret wird auf folgende Eigenschaften überprüft:

- Ungenutzte CPU-Kerne
Ungenutzte CPU-Kerne deuten darauf hin, dass die Anwendung möglicherweise besser parallelisiert werden kann.
- Schlechte Cache-Performance

Eine schlechte Cache-Performance deutet darauf hin, dass das Speicherzugriffsmuster optimiert werden sollte.

- Viele Branch-misses

Viele branch-misses deuten daraufhin, dass man die Schleifen/Verzweigungen im Programmablauf optimieren kann.

- Häufiges Warten auf Disk-IO

Wenn eine Anwendung lange auf viele Disk-IO Operationen warten muss, deutet dies darauf hin, dass der Benutzer möglicherweise das Zugriffsmuster der Anwendung optimieren kann.

Das häufige Warten auf eingehende Nachrichten wird hier nicht extra festgestellt, obwohl es daraufhindeutet, dass das Kommunikationsschema optimiert werden kann. Dieser Aspekt wird dann allerdings in Schritt 4 der Analyse betrachtet.

2. Ausgelastete Systemkomponenten ermitteln

Um ein mögliches Bottleneck zu zeigen, werden hier für den Nutzer die am stärksten ausgelasteten Systemkomponenten ermittelt. Damit kann er feststellen auf welche Bereiche er seine Optimierungen am besten priorisieren sollte.

3. Nicht ausgelastete Komponenten ermitteln

Um dem Benutzer die Optimierung zu erleichtern, werden in diesem Schritt die am wenigsten belasteten Systemkomponenten ermittelt. So kann der User überprüfen ob es in seinem Programm möglich ist, mehr Last an diese Komponenten abzugeben.

Die Schritte 2 und 3 erlauben es dann, dem Benutzer einen Vergleich der erwarteten Last mit der tatsächlichen vorzunehmen und möglicherweise auftretende Diskrepanzen genauer zu untersuchen.

4. Vergleich verschiedener Samples

Als letzten Schritt präsentiert das Programm dem Benutzer ein Vergleich auffallend unterschiedlicher Samples (bzw. die Abwesenheit solcher) eines Programmlaufs. Dies dient dazu, dem Nutzer die Möglichkeit zu geben verschiedene Arten der Lastverteilung zu verbessern.

Zunächst werden verschiedene Samples des gleichen Hosts verglichen, um festzustellen ob das Programm die Last zeitlich besser verteilen kann. Damit lassen sich z.B. Schwächen bei der Lastaufteilung auf die einzelnen Threads erkennen, aber auch ob Komponenten, wie z.B. die Festplatte, nur sporadisch, dann aber sehr stark, anstatt konstant unter moderater Last benutzt werden.

Auch ein Vergleich zwischen den Prozessen auf verschiedenen Rechenknoten wird durchgeführt, um mögliche Schwächen bei der Lastaufteilung auf die einzelnen Rechenknoten zu entdecken.

5. Benutzung des Frameworks

5.1. Aufruf und Funktionen des Programms zum Erstellen einer Konfigurationsdatei

Das Programm `./configure` erstellt interaktiv eine Konfigurationsdatei `configuration.conf`. Die eventuell vorhandene Konfigurationsdatei wird dabei überschrieben. Das Programm fragt den Benutzer dabei für jedes erkannte Netzwerkinterface und jedes Block-device, ob der Benutzer Metriken für dieses Gerät erheben möchte.

Die Konfiguration wird dann bei der Messung und Analyse benutzt um festzustellen, welche Metriken gemessen werden sollen.

Für das Erstellen und Auslesen der Konfigurationsdatei wird der Key-value file parser aus der `glib` verwendet¹.

Die Datei enthält für Netzwerk-Metriken eine Gruppe `net`, in der zu jedem Device, das der Benutzer messen möchte, die Standardeinstellung der Metriken für dieses Gerät gespeichert ist. Dabei ist der Key der Devicename und Value der Wert der Standardeinstellung. Geräte die gar nicht gemessen werden sollen, haben keinen Eintrag in der Konfigurationsdatei. Für die Blockdevices gibt es eine analog aufgebaute Gruppe `disk`. Weitere Gruppen in der Datei werden ignoriert, sodass es für die Implementation eigener Metriken möglich ist, weitere Gruppen zu definieren, in denen benötigte Informationen gespeichert werden.

¹<https://developer.gnome.org/glib/stable/glib-Key-value-file-parser.html>

5.2. Aufruf und Funktionen des Programms zur Messung

Ein Aufruf des Programmes `./measure` muss mit dem Parameter des zu messenden Befehls ausgeführt werden.

Zusätzlich sind folgende optionale Parameter möglich:

- `-o outputDatei` oder `-output outputDatei`: Legt den Namen der OutputDatei fest.
- `-n Anzahl` oder `-number Anzahl`: Legt die Anzahl der Messungen fest, von denen der Durchschnitt gebildet werden soll.
- `-m Anzahl` oder `-max Anzahl`: Bestimmt die maximale Anzahl an Tools, die gleichzeitig genutzt werden können.
- `-d` oder `-dir`: Erzeugt ein Verzeichnis, in dem die Messdaten gespeichert werden. Dieser Modus ist für das Verwenden beim Messen mehrerer Rechenknoten gedacht.
- `-a` oder `-additional`: Erzeugt eine weitere Datei mit allen gemessenen Werten (der Metriken, die ausgewählt wurden) und gibt bei der Zusammenfassung neben den Durchschnittswerten auch die maximale Abweichung der einzelnen Messwerte an.
- `-s samplingrate` oder `-sampling samplingrate`: Die Messwerte werden in der angegebenen Samplingrate in Hz gesampelt. Die Samplingrate wird dabei als Fließkommazahl entgegengenommen. Um die gesampelten Werte auszugeben sollte `-a` gesetzt werden, da ansonsten nur der Mittelwert aller Sampels gespeichert wird.
- `-metric1=true`: Schaltet die Messung der metrik1 an.
- `-metric2=false`: Schaltet die Messung der metrik2 aus.

Das Programm bestimmt dann zunächst, welche Metriken gemessen werden sollen. Dazu wird die Konfiguration der Metriken mit der Eingabe des Benutzers kombiniert, um zu prüfen, welche Metriken gefragt sind und welche nicht.

Der übergebene Befehl wird dann so oft durchgeführt wie nötig, um alle Metriken zu messen, ohne mehr als `m` (standardmäßig so viele, wie es Metriken gibt, wenn der Benutzer keinen anderen Wert übergibt) Tools gleichzeitig zu benutzen. Der User kann hier also selbst festlegen, wie stark der Programmablauf durch die Mess-Tools gestört werden darf.

Dieser Vorgang wird `n`-mal (standardmäßig 1, wenn der Benutzer keinen anderen Wert übergibt) wiederholt, um einen aussagekräftigeren Durchschnitt von verschiedenen Messungen zu bilden.

Der Durchschnitt der Messergebnisse wird dann in die Ausgabedatei geschrieben. Wenn `-a` gesetzt ist, wird die Ausgabe um eine weitere Datei mit allen gemessenen Werten erweitert. Wenn `-d` gesetzt ist, wird der mit `-o` übergebene Parameter als Ordnername aufgefasst, in den die Messergebnisse geschrieben werden. Mit `-s` kann man eine Samplingrate in Hz

angeben, in der die Werte für die Metriken erfasst werden sollen. Dabei werden Metriken, die durch Profiling erfasst werden (vgl. Abschnitt 4.1.5) nicht mit gesampelt.

Das Programm erwartet eine Konfigurations-Datei (configurations.conf) im gleichen Verzeichnis wie das measure Programm. Der Inhalt und Aufbau der Konfigurationsdatei wird in Abschnitt 5.1 beschrieben. Das Programm verwendet Funktionen der glib, daher muss diese verfügbar sein.

Wrapper für MPI-Programme

Das Programm ./mpi-wrapper dient dazu, dass Mess-Programm mit einer MPI-Applikation über mehrere Knoten verteilt auszuführen. Der Aufruf des Programms ist genauso wie der oben (Abschnitt 5.2) beschriebene Aufruf des Programms zur Messung. Dabei wird immer -d gesetzt. Außerdem ist der mpi-wrapper nur zum Sampling gedacht, weshalb standardmäßig -s 1 gesetzt wird. -a wird aber nicht standardmäßig gesetzt, sodass standardmäßig nur die Mittelwerte der gemessenen Metriken gespeichert werden.

Der zu messende Befehl wird mit mpiexec erwartet, sodass ein Aufruf beispielsweise so aussehen könnte:

Listing 5.1: Beispielaufruf des MPI-wrappers

```
1 mpiexec ./mpi-wrapper -a -s 1 -d -o testmessung 'mpiexec  
  ↪ ./partdiff-par 1 2 512 2 2 500'
```

Das Programm verwendet Funktionen der glib, daher muss diese verfügbar sein, außerdem werden Funktionen einer MPI-Bibliothek verwendet.

5.3. Aufruf und Funktionen des Programms zur Analyse

Zusätzlich gibt es dann das Programm ./analyze. Als Parameter muss die Eingabedatei, aus der die Messdaten gelesen werden sollen, übergeben werden.

Zusätzlich sind folgende optionale Parameter möglich:

- -o outputDatei oder -output outputDatei: Legt den Namen der Ausgabedatei fest. standardmäßig wird stdout für die Ausgabe der Analyseergebnisse verwendet.
- -v oder -verbose: Es werden zusätzliche Analyseergebnisse für alle Samples ausgegeben.
- -d oder -dir: Liest alle Eingabedateien im angegebenen Ordner ein. Diese Einstellung ist analog zur -d Einstellung beim Programm zur Messung (vgl. Abschnitt 5.2).
- -t threshold oder -threshold threshold: Setzt den threshold-Wert (als Fließkommazahl im bereich 0.0 bis 100.0). Eine Auslastung der Systemkomponenten größer als der gesetzte Wert wird dabei als signifikante Last angenommen, bei einer kleineren Auslastung wird dem Benutzer vorgeschlagen die Auslastung dieser Ressourcen zu erhöhen. Der Wert ist standardmäßig auf 50 gesetzt.

- `-c cluster` oder `-cluster cluster`: Setzt die Anzahl der verschiedenen Gruppen von ähnlichen Samples, die miteinander verglichen werden sollen, als Anteil der Gesamtanzahl von Samples. Es werden $1/\text{cluster}$ viele Gruppen gebildet, `cluster` wird dabei als ganze Zahl entgegengenommen. Der Wert ist standardmäßig 10, wobei nie weniger als 3 Gruppen verwendet werden.
- `-t top x` oder `-top x`: Beim Vergleich verschiedener Samples werden nur die top x größten Unterschiede verglichen. Standardmäßig ist dieser Wert 3.
- `-h host` oder `-host host`: Setzt den Hostnamen, der als Ausgangspunkt für den Vergleich mit anderen Hosts dient (nur im Zusammenhang mit `-d` relevant). Standardmäßig wird der erste gefundene Host genutzt.

Das Programm führt die Analyse der durch die Inputdatei, bzw. der Inputdateien mehrerer Hosts wenn `-d` gesetzt ist, gegebenen Messwerte nach dem in Abschnitt 4.2.1 dargestellten Konzept durch.

Dabei ist wichtig, dass das Programm mit der gleichen Konfigurations-Datei und bei gleicher Systemkonfiguration wie die Messung ausgeführt wird. Da es nur dann möglich ist, die gemessene Belastung mit der durch Hardwareinformation gegebenen maximalen Belastung korrekt in Beziehung zu setzen. Wichtig ist, dass in den Messergebnissen alle Metriken einer Gruppe (z.b. alle Metriken zur CPU-Auslastung) gespeichert sind, da die Analyse mit unvollständigen Daten zu unvollständigen Ergebnissen führen kann.

Das Programm verwendet Funktionen der `glib`, daher muss diese verfügbar sein, außerdem können die möglicherweise schneller berechnet werden, wenn `openmp` verwendet wird. Aufgrund der Struktur von `openmp` ist aber auch eine Verwendung ohne `openmp` möglich.

6. Zusammenfassung und Future Work

Das entwickelte Framework bietet einen leichtgewichtigen Einstieg in die Performanceanalyse und unterstützt den Benutzer in seiner Entscheidung, welche weiteren Schritte sinnvoll sind. Der Benutzer kann mit den gewonnenen Informationen besser entscheiden, wie er gezielt weitere Tools zur Analyse der vom Framework vorgeschlagenen möglichen Leistungsengpässe verwenden möchte. Das Framework unterstützt die Entscheidung, bei welchen Maßnahmen nach dem Pareto Prinzip¹ die größten Verbesserungen der Performance zu erwarten sind.

Damit fällt es dem Benutzer leichter, den in Kapitel 3 dargestellten Herausforderungen effizient zu begegnen.

Das Framework wurde möglichst leichtgewichtig gehalten. In einem kurzen Versuch konnte keine signifikante Beeinflussung des Systems durch das Framework festgestellt werden (vgl. Anhang A).

Das Framework wurde erweiterbar gestaltet, sodass neue Metriken sowohl für die Messung als auch für die Analyse einfach hinzugefügt werden können. Als weiterführende Arbeiten bietet es sich daher an, weitere Metriken zu implementieren (siehe dafür Abschnitt 4.1.3).

Zusätzlich kann man auch das Messen der Metriken durch andere Programme evaluieren, da die Analyse der gesammelten Daten einen unabhängigen eigenständigen Schritt darstellt. Für die Analyse werden die gemessenen Werte aus einer oder mehrerer Eingabedateien eingelesen (vgl. Abschnitt 4.1.1).

Als wichtige Erweiterung der Analyse kann man die theoretisch ermittelte maximale Leistungsfähigkeit des Systems durch das Ausführen spezialisierter Benchmarks validieren. Die durch Benchmarks ermittelten realistischen Leistungsgrenzwerte können beispielsweise in der Konfigurationsdatei (siehe Abschnitt 5.1) gespeichert werden.

¹auch 80/20 Regel

A. Messung des Overheads des Frameworks

Befehl:			
time mpiexec -n 24 ./partdiff-par 4 2 1024 2 2 2048	5m1.589s	5m1.923s	5m0.180s
./measure -a -s 0.1 -d -o testmessung 'time mpiexec -n 24 ./partdiff-par 4 2 1024 2 2 2048'	4:59.40	5:02.49	5:03.14
time ./measure -a -s 0.1 -d -o testmessung 'mpiexec -n 24 ./partdiff-par 4 2 1024 2 2 2048'	5m10.077s	5m10.080s	5m10.080s
./measure -a -s 1 -d -o testmessung 'time mpiexec -n 24 ./partdiff-par 4 2 1024 2 2 2048'	5:00.00	4:59.80	5:01.09

Tabelle A.1.: Messung der zusätzlichen Laufzeit mit dem partdiff-par Programm auf west1

Ich kann keine signifikante Erhöhung der Laufzeit des Programmes feststellen. Wie erwartet ist der gesamte Prozess der Messung sowie Aufbereitung (u.a. den Mittelwert bestimmen) und Ausgabe der Ergebnisse länger als die eigentliche Messung.

Abbildungsverzeichnis

2.1	Ausgabe des Befehls <code>cat /proc/cpunifo</code>	8
2.2	Ausgabe des Befehls <code>htop</code>	9
2.3	Ausgabe des Befehls <code>df -h</code>	9
2.4	Ausgabe des Befehls <code>iostat</code>	10
2.5	Ausgabe des Befehls <code>nstat</code>	11
2.6	Ausgabe des Befehls <code>perf stat</code>	11

Listingverzeichnis

4.1	Definition des <code>metric_struct</code>	16
4.2	Definition des Funktionsaufrufes zum monitoring	20
4.3	Definition des Funktionsaufrufes zum Profiling	21
4.4	Definition des <code>rstruct</code>	21
5.1	Beispielaufruf des MPI-wrappers	27

Literaturverzeichnis

- [dia] Diamond. <https://github.com/python-diamond/Diamond> zugriff 17.2.18.
- [dPeM06] Bureau International des Poids et Mesures. The international system of units (si). https://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf Zugriff 14.2.18, 2006.
- [Gre15] Brendan Gregg. Linux performance tools. Netflix Technology Blog <https://medium.com/netflix-techblog/netflix-at-velocity-2015-linux-performance-tools-51964ddb81cf> [zugriff 30.10.2017], 2015.
- [GSvB⁺16] Jorge Goncalves, Zhanna Sarsenbayeva, Niels van Berkel, Chu Luo, Simo Hosio, Sirkka Risanen, Hannu Rintamäki, and Vassilis Kostakos. Tapping task performance on smartphones in cold temperature. *Interacting with Computers*, 29(3):355–367, 2016.
- [Int17] Intel. Analyzing python performance with intel vtune amplifier xe. https://www.alcf.anl.gov/files/Tullos-Analyzing_Python_Performance.pdf zugriff 16.3.18, 2017.
- [Jai91] Raj Jain. *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*. John Wiley & Sons, Inc, 1991.
- [Jam16] Tim Jammer. Energy usage analysis of hpc applications, 12 2016.
- [Lil05] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge university press, 2005.
- [opt14] Tips for optimizing c/c++ code. <https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf> zugriff 6.3.18, 2014.
- [Tho17] Frank Thomas. Measure ram speed for application/ current speed utilization. <https://superuser.com/questions/1222074/measure-ram-speed-for-application-current-speed-utilization> zugriff 6.3.18, 2017.
- [Tit17] James Titcomb. Why your phone slows down and loses battery in hot weather. <http://www.telegraph.co.uk/technology/2017/06/20/phone-slows-loses-battery-hot/> Zugriff 14.2.18, 2017.

[Wik] Computer performance - aspects of performance. https://en.wikipedia.org/wiki/Computer_performance#Aspects_of_Performance Zugriff 14.2.18.