THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Inbetriebnahme und Überwachung von grafikkartenintensiven Anwendungen in verteilten Systemen unter Berücksichtigung der Steuerung von handlungsbedürftigen Zuständen

Bachlorthesis/Seminararbeit
Studiengang Informatik

vorgelegt von
Simon Stockhause

Januar 2020

Referent der Arbeit:	Prof. Dr. Harald Ritz
Koreferent der Arbeit:	Prof. Dr. Harald Ritz
Betreuer:	M.Sc. Pascal Bormann

Zusammenfassung

Die heutigen Bedürfnisse der Anwender ein stets erreichbaren Service zur Verfügung zu haben, stellt hohe Erwartung an Unternehmen und damit hohe Erwartungen an die Infrastruktur ihrer Mircoserviceanwendungen. Die enorme Skalierbarkeit einzelner Komponenten und die aussgezeichnete Resourcennutzung der Hardware löst viele Probleme der Vergangenheit. Allerdings schafft diese Umstellung neue Herausfoderungenen die es zu bewältigen gilt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Zielsetzung

Das Ziel dieser Arbeit soll sein, eine Möglichkeiten aufzuzeigen, die enorme Komplexität der Infrastruktur von Mircoservicearchitekturen unter Kontrolle zu bringen. Ausserdem

2 Die untersuchten Sprachen

2.1 Die Sprache C

C ist eine universell einsetzbare Programmiersprache. Zudem ist C eine relativ hardwarenahe Sprache. Diese Eigenschaft ist nicht abwertend zu verstehen. Es bedeutet nur, dass C mit der gleichen Art von Objekten umgeht, wie Computer. Genauer gesagt Zahlen, Zeichen und Adressen. Es gibt auch keine Heap- oder Garbage collection.[1]

2.2 Die Sprache R

Die Sprache R wird hauptsächlich für *statistical Computing* eingesetzt. R besteht nicht nur aus der Sprache selbst, sonder auch einer Laufzeitumgebung mit Grafiken, einem Debugger, Zugang zu verschiedenen Systemfunktionen und der Möglichkeit Scripte auszuführen.
Viele in R genutzte mathematische Funktionen sind selbst in R geschrieben, dennoch lassen sich auch in C/C++ oder FORTRAN geschriebene Funktionen in R nutzen[2]

2.3 Relevante unterschiede der Sprachen

Der relevenateste Unterschied zwischen den Sprachen C und R ist, dass C eine kompilierte und R eine interpretierte Sprache ist.
Ein großer Vorteil von R ist die Entwicklungszeit, die gebraucht wird, um Programme zu schreiben. Für die naive Implementierung zum invertieren von Matrizen wurde in C insgesamt 235 Zeilen Code geschrieben. Im Gegensatz dazu der R Code mit nur 21 Zeilen, wovon 15 zeilen dafür zuständig sind, sich um Ausgabe und Zeitmessung zu kümmern. Auch die Lesbarkeit ist in R oftmals besser, was zum einem an dem geringeren Umfang des Programmcodes liegt und zum anderen an den komplizierteren Sprachkonstrukten von C.

3 Messmethode zur Ermittlung der Rechenzeit

3.1 Messen der Gesamtlaufzeit mithilfe eines system call

Zur Berechnung der Gesamtlaufzeit wurde ein Tool entwickelt, das eine Shell imitiert. Innerhalb dieser Shell lassen sich ausführbare Programme oder Systemkommandos aufrufen. Diese werden ausgeführt und anschließend deren Gesamtlaufzeit im Benutzermodus in *clock-ticks* ausgegeben. Damit lassen sich exakte Laufzeiten eines Programms ermitteln, ohne von äußeren Einflüsssen, wie dem Prozessor Scheduler oder im Kernel Mode ausgeführte Funktionen wie z.B. print Funktionen, beinflusst zu werden. Allerdings fließt die Initialisierungszeit, der in dieser Arbeit benutzten Programme, d.h. die Belegung einer Matrix mit Zufallswerten usw. mit in die Gesamtlaufzeit ein. Auch die Anzahl der Iterationen, also die Anzahl an zufälligen Matrizen, die invertiert werden, spielt eine ausschlaggebende Rolle. Deshalb wird diese gemessene Zeit nur als Prüfwert benutzt, um sicherzustellen, dass die Summe der gemessenen Zeiten pro Iteration, mit der im nächsten Abschnitt vorgestellten Methode, ungefähr mit der Gesamtlaufzeit des Programmes übereinstimmmt.

3.2 Messen der Laufzeit einer Iteration durch Zeitdifferenz zwischen Codeabschnitten

Um die durschnittliche Laufzeit einer Iteration zu ermitteln, wird eine Zeitmessung durchgeführt, die nur den Teil des Programms umfasst, der sich mit dem Finden der Inversen einer Matrix beschäftigt. Anschließend wird die Summe aller Iterationszeiten durch die Anzahl der Iterationen geteilt, um den Durschnitt zu erhalten.
Problematiken bei dieser Messmethode sind leider unausweichlich. So entstehen z.B. Schwankungen der Rechenzeiten, aufgrund von Kontextwechsel bzw. Taskswitching der CPU.

4 Verwendete mathematische Funktion

4.1 Gauss-Jordan Algorithmus

Der Gauss-Jordan Algorithmus kann in 3 Schritten durchgeführt werden. Dabei sei vorausgesetzt, dass die Matrix quadratisch und ihre Determinate ungleich 0 ist. Die naive Implementierung in C und das Matlib Packet in R setzen genau diesen Algorithmus um.

Der Gauss-Jordan Algorithmus wird anhand eines Beispiels genauer erläutert.

Gegeben seien folgende Matrizen A und B:

$$A: \left\{ \begin{array}{ccc} 0 & 2 & 0 \\ 4 & 5 & 6 \\ 0 & 0 & 9 \end{array} \right\} \quad B: \left\{ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right\}$$

1. Transformation der ggb. Matrix in die Einheitsmatrix und anwenden aller Operationen auf die mitgeführte Matrix.

$$\left\{ \begin{array}{ccc|ccc} 0 & 2 & 0 & 1 & 0 & 0 \\ 4 & 5 & 6 & 0 & 1 & 0 \\ 0 & 0 & 9 & 0 & 0 & 1 \end{array} \right\}$$

   1.1 Reduziere Matrix zu Stufenform

$$\left\{ \begin{array}{ccc|ccc} 4 & 5 & 6 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 1 \end{array} \right\} Tausche\, Reihe\, 1\, mit\, Reihe 2$$

   1.2 Reduziere Matrix zu reduzierter Stufenform

$$\left\{ \begin{array}{ccc|ccc} 4 & 5 & 0 & 0 & 1 & -\frac{2}{3} \\ 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 1 \end{array} \right\} Reihe\, 1 - Reihe\, 3 * (6/9)$$

$$\left\{ \begin{array}{ccc|ccc} 4 & 0 & 0 & \text{-2,5} & 1 & -\frac{2}{3} \\ 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 1 \end{array} \right\} Reihe\, 1 - Reihe\, 2 * (5/2)$$

1.3 Teile Reihe der rechte Teilmatrix durch Diagonalelement der Reihe der linken Teilmatrix

$$\left\{\begin{array}{ccc|ccc} 1 & 0 & 0 & -\frac{5}{8} & \frac{1}{4} & -\frac{2}{3} \\ 0 & 1 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{9} \end{array}\right\}$$

Durch diese 3 Schritte erhalten wir nun das Inverse der untersuchten Matrix. Es ist das Nebenprodukt der Operationen, die gebraucht werden, um die ggb. Matrix in eine Einheitsmatrix zu transformieren.

$$Inverse(A) : \left\{\begin{array}{ccc} -\frac{5}{8} & \frac{1}{4} & -\frac{2}{3} \\ \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{9} \end{array}\right\}$$

4.2 Optimierte Gaußsche Eliminationsverfahren

Dieses Verfahren wird in GSL (GNU Scientific Library) und in LAPACK (Linear Algebra PACKage) verwendet. Die Programme mit den besten gemessenen Laufzeiten, benutzten jeweils eines der Pakete.

Entsprechen der Dokumentation von LAPACK[3] kann der DGESV Algorithmus wie folgt abgeleitet werden:

Der DGSV Algorithmus berechnet die Lösung eines linearen Gleichungssystems

$$A * X = B$$

Dabei sei A eine NxN Matrix. X und B seien NxM Matrizen, wobei M für die Anzahl der Spalten von B steht.

Die LR-Zerlegung arbeitet mit Pivotisierung. Das Pivotelement beschreibt das erste Element, welches benutzt wird um eine Rechnung durchzuführen. Dieses Pivotelement wird ggf. durch Reihentausch angepasst. Die LR-Zerlegung wird benutzt um A in 3 Teile zu unterteilen.

$$A = P * L * U$$

P steht hierbei für die Permutationsmatrix, einer Matrix die Vertauschungen darstellt.
L steht für die untere Dreiecksmatrix.
U steht für die obere Dreiecksmatrix.
Diese unterteilte Form wird benutzt, um lineare Gleichungssystem A * X = B zu lösen.

5 Betrachtung der Rechenzeit verschiedener Implementierungen

Zu jeder Implementierung werden folgende Messdaten erhoben:

- Berechnung der Inverse einer Matrix mit einer Größe von 100x100 über 10000 Iterationen
- Berechnung der Inverse einer Matrix mit einer Größe von 1000x1000 über 50 Iterationen

Dabei ist zu erwähnen, dass sich die Rechenzeiten auf unterschiedlichen Systemen stark unterscheiden können. Das hier genutzte System besteht aus:

- 64-Bit Ubuntu Betriebssystem
- i5-2520M 2,5 GHz Prozessor mit 4 Kernen
- 8 Gigabyte Arbeitsspeicher

5.1 C naive Implementierung

Die naive C Implementierung kann sowohl die 100x100 als auch die 1000x1000 Matrix berechnen. Der Speicherverbrauch steigt mit der Anzahl der Iterationen.

5.1.1 100x100 Matrix

Die Rechenzeit beinhaltet in den ersten 4000 Iterationen nur kleine Schwankungen und diverse Ausreißer. Ab der ca. 4500 Iteration stellt sich eine Art Kurvenbewegung mit gleichgroßen Abschnitten in den Rechenzeiten ein, die zwischen 0.014 und 0.017 Sekunden Schwanken. Zum Ende, also ab ca. 9000 Iterationen nimmt diese Schwankung stark zu.

Die durschnittliche Rechenzeit pro Iteration über 10000 Iterationen beträgt 0,014822 Sekunden.

Der Speicherverbrauch nimmt stetig zu. Dieser ist zum Ende hin bei ca. 1,5 Gigabyte.

Siehe Abbildung: **??**

5.1.2 1000x1000 Matrix

Auch bei der Berechnung von 1000x1000 ist ein stetiger Anstieg der Rechenzeiten zu erkennen. Die durchschnittliche Rechenzeit beträgt circa 17,0109 Sekunden. Der Speicherverbrauch nimmt stetig zu. Nach 50 Iterationen verbraucht das Programm ca. 750 Megabyte. Siehe Abbildung: **??**

5.2 C unter Vewendung von GSL

Die GSL Implementierung kann sowohl die 100x100 als auch die 1000x1000 Matrix berechnen. Der Speicherverbrauch bleibt konstant.

5.2.1 100x100 Matrix

Es sind Schwankungung in der Rechenzeit zu erkennen. Diese erstrecken sich im Bereich von 0.00145 bis 0.002. Es gibt diverse Ausreißer. Dennoch bleibt die Rechenzeit grundsätzlich konstant. Es ist ein Block bei ca. 9500 Iterationen zu erkennen, der mit seiner erhöhten Rechenzeit aufällt. Die durchschnittliche Laufzeit pro Iteration beträgt 0,001517 Sekunden

Der Speicherverbrauch bleibt mit 344 Kilobyte extrem gering und konstant.

Siehe Abbildung: **??**

5.2.2 1000x1000 Matrix

Es fallen sehr konstante Rechenzeiten auf. Dennoch sind die beiden ersten Rechenzeiten um etwa 0,1 bis 0,2 Sekunden höher. Auch ein extremer Aussreißer mit ca. 6,37 Sekunden ist zu erkennen. Die durchschnittliche Laufzeit beträgt 6,03 Sekunden.

Der Speicherverbrauch bleibt wieder mit 15,8 MiB über die gesamte Laufzeit aller Iterationen gleich.

Siehe Abbildung: **??**

5.3 R standard solve Funktion

5.3.1 100x100 Matrix

Es fallen absolut konstante Rechenzeiten in insgesamt 4 Kategorien auf. Die durchschnittliche Rechenzeit liegt bei 0,00266 Sekunden Der Speicherverbrauch liegt bei 40.5 Megabyte

Siehe Abbildung: **??**

5.3.2 1000x1000 Matrix

Bei sehr großen Matrizen scheint die Schwankung der Rechenzeiten etwas deutlicher auszufallen. Dennoch ist die Laufzeit pro Iteration extrem gering. Der Durschschnitt liegt bei 1,327 Sekunden. Der Speicherverbrauch verdoppelt sich circa auf 71 bis 86 Megabyte.

Siehe Abbildung: **??**

5.4 R Matlib-Package

Das Matlib Paket ist für Bildungszwecken entwickelt worden. Es kann verschiedene Schritten innerhalb der Algorithmen veranschaulichen. Dementsprechen ist die Implementierung nicht auf Laufzeit bzw. Speichereffizients ausgerichtet. Intern wird auch das Gaußsches Eliminationsverfahren implementiert.

5.4.1 100x100 Matrix

Bereits der Berechnung des Inversen von 100x100 Matrizen geht diese Implementierung an die Grenzen des genutzten Systems. Bei 100 Iterationen werden 1,74 Gigabyte Speicher beansprucht. Aus diesem Grund konnten keine 10000 Iterationen durchgeführt werden, da der Arbeitspeicher nicht ausreicht. Die durschnittliche Rechenzeit beträgt 2,45 Sekunden. Auch sind starke Schwankungen der Laufzeit zu erkennen.

5.4.2 1000x1000 Matrix

Der Versuch mit dem Matlib Paket, das Inverse einer 1000x1000 Matrix zu berechnen, scheitert. Intern scheinen extrem ineffiziente Datenstrukturen verwendet zu werden, die die Systemgrenze von 8 Gigabyte Arbeitsspeicher überbeanspruchen. Aus diesem Grund können keine Messsdaten erhoben werden.

5.5 Ergebnisdiskussion

Jede Implementierung weist spezielle Eigenschaften auf. Die naive Implementierung in C scheint stark von den genannten Problematiken der Messmethode betroffen zu sein. Der Kontextwechsel des Prozessors ist einer dieser Problematiken. Dies könnte für die *Blockformung* der Rechenzeiten verantworlich sein. Auch ist durch den stetig ansteigenden Speicherverbrauch davon auszugehen, das die Implementierung ein *Speicherleck* (engl. Memory Leak) aufweist. Dies könnte der Grund dafür sein, das die Rechenzeiten stufenweise schlechter werden.

Die naive C Implementierung kann als ineffizient bezeichnet werden. Sie ist zwar in der Lage das Problem zu lösen, dennoch schneidet sie im Gegensatz zu GSL und der standard R Implementierung, bei der Berechnung des Inversen von 100x100 Matrizen um ca. den Faktor 10 schlechter ab. Diese Erkenntnis trifft auch auf die Berechnung der 1000x1000 zu. Dennoch sind die Ergebnisse deutlich besser, als die Messdaten, die mithilfe des Matlib Paket erhobenen Daten. Die durchschnittlichen Rechenzeit bei Matlib ist um den Faktor 165 langsamer.

Die Matlib Implementierung lässt sich somit definitiv auf den letzten Platz einordnen. Nicht nur die Rechenzeit, sondern auch der Speicherverbrauch ist ungemein höher, als bei den anderen Programmen. Dieses ineffiziente Verhalten des Programms sorgt sogar dafür, dass 1000x1000 Matrizen nicht bearbeitet werden können.

Die hochoptimierten Funktionen der GNU Scientific Library und der Standard R Library liefern interessante Ergebnisse, die es in zwei Kategorien zu unterteilen gilt:

1. 100x100 Matizen
   Die *solve* Funktion in R scheint bei der Berechnung der Matrizen mit verschiedenen Schwierigkeitsstufen konfrontiert zu sein, was die 4 Kategorien der Rechenzeit erklären würde. Das könnte mit unterschiedlichen Eigenschaften der gegebenen Matrix zusammenhängen. Nichtsdestotrotz lässt sich nicht ausschließen, dass die Messdaten vom Kontextwechsel der CPU betroffen waren. Anhand der gegebenen Daten lässt sich dennoch darauf schließen, dass die GSL Implementierung um ca. den Faktor 1,5 schneller ist, als die R Funktion *solve*, wenn es darum geht 100x100 Matizen zu behandeln. Der Speicherverbrauch unterscheidet sich hingegen extrem. Bei GSL werden nur 336 Kilobyte Speicher beansprucht. Die R Variante ist durch ihren gesamten Ballast, der durch die Sprache mitgeliefert wird, bei 40,5 Megabyte. Das bedeutet, dass GSL 121 mal weniger Speicherplatz benötigt, als R.
2. 1000x1000 Matizen
   In Anbetracht der bisherigen Daten, ist es interessant, dass die *solve* Funktion bei großen Matizen signifikant besser abschneidet, als GSL. Dabei ist solve ca. 4,5 mal schneller. Dies könnte an der Natur von R liegen, mit großen Datenmengen effizient zu arbeiten. Der Speicherverbrauch nähert sich bei großen Matrizen etwas an. Dabei liegt GSL mit 15,3 Megabyte aber immernoch deutlich vor den 71 bis 89 Megabyte des R-Scripts.

Fazit

Die Hypothese, die in der Zielsetzung definiert worden ist, besagt, dass die Implementierungen der Inversenberechnung in C bessere Laufzeiten aufweisen, als in R.

Dies kann in Anbetracht der Messergebnis nur Teilweise bestätigt werden.

- Bei kleinen Matrizen trifft dies definitiv zu, wie in der Ergebnissdiskussion aufgezeigt wurde.
- Bei immer größer werdenden Matrizen steigt die Performance von R im Verhältnis zu den C-Varianten.

Diese Arbeit hat jedoch auch deutlich gemacht, wie stark die Performance von einer Implementierung abhängt. Es wurden gleiche Algorithmen mit unterschiedlichen Implementierungen verglichen und es hat sich herausgestellt, dass deutliche Unterschiede festzustellen sind.

Die beiden naiven Ansätze leiden an gefährlichen Fehlern, wie z.B. Speicherlecks, die zum Absturz des Programms führen können. Das bedeutet, dass man bei der Auswahl seiner Werkzeuge, unbedingt die Rahmenbedingungen miteinbeziehen sollte. Damit ist gemeint, dass falls man die Möglichkeit hat, bewährte Packete einzusetzten, dann ist es empfohlen dies zu tun, anstatt das Rad neu zu erfinden.

Die C-Varianten können vorallem bei dem Speicherbedarf glänzen. Dafür sind jedoch hoher Entwicklungsaufwand und extreme Vorsicht bei der Implementierung aufzubringen.

Die R-Variante bietet eine enorme Auswahl an mathematischen Funktionen, die meistens sehr perfomant sind. Der Entwicklungsaufwand von Code ist ungemein geringer.

In Anbetracht der heutigen Ressourcen kann man in vielen Anwendungsfällen beruhigt auf die einfacheren Scriptsprachen zurückgreifen. Es lässt sich aber in speziellen Fällen nicht umgehen, die volle Kontrolle, bei z.B. sicherheitskritischen Bereichen, zu übernehmen. In diesem Fall bietet sich C an.

Literatur

[1] B. W. Kernighan und D. M. Ritchie, *The C programming language*, 2. ed., Ser. Safari books online. Upper Saddle River, NJ: Prentice-Hall PTR, 1988, ISBN: 9780133086249. Adresse: http://proquest.tech.safaribooksonline.de/book/programming/c/9780133086249.

[2] *R FAQ*, 5.10.2017. Adresse: https://cran.r-project.org/doc/FAQ/R-FAQ.html#What-is-R_003f.

[3] *LAPACK: dgesv*, 13.11.2017. Adresse: http://www.netlib.org/lapack/explore-html/d7/d3b/group__double_g_esolve_ga5ee879032a8365897c3ba91e3dc8d512.html#ga5ee879032a8365897c3ba91e3dc8d512.