




# Optimierung sequentieller Programme

1. Motivation und Ansätze
2. Ebenen und Potentiale
3. Anwendungsklassen
4. Leistungsabschätzungen
5. Optimierung der Mathematik
6. Optimierung der Programmierung
7. Optimierung mit dem Compiler
8. Fazit



Programmierung ist eine Krücke, die wir nur benutzen,  
weil wir noch nicht weit genug sind, mathematische  
Darstellungen direkt in Ergebnisse zu transformieren

Mathematik

Numerik

Programm

Ergebnisdaten

# 1. Motivation und Ansätze

## „Gefühlte Programmgeschwindigkeit“

- Wichtig für den eigenen Arbeitsablauf
- Stark situationsabhängig
- Psychologische Effekte beachten

## Beispiele

- Reaktionszeit beim Anklicken eines Knopfes
  - Nach max. 2 Sekunden sollte eine Rückmeldung kommen
- Ausführungszeit der angeklickten Operation
  - Beliebig, aber durch Benutzererwartungen beschränkt
  - Vorhersagbarkeit wäre gut
  - Falls nicht das, dann wenigstens Fortschrittsanzeige



# Motivation (2)

---

## Programme mit GUI (Geschäftsprogramme)

- Schnelle Reaktion der GUI
- Beliebige Reaktion des Programms

## Programme auf Kommandozeile (technisch/wissensch.)

- Meist wünscht man sich kürzere Programmlaufzeiten

## Programme in Maschinensteuerungen

- Vorgegebene maximale Laufzeit (Echtzeitprogramme)  
Z.B. Auslösung eines Airbags

## Wir warten auf schnellere Hardware

- Ging von 1941 bis ca. 2005

Permanente Leistungssteigerung der Prozessoren unter Beibehaltung des Nutzungskonzepts

- Compiler erzeugt Programm für den Prozessor (1bit – 64bit)
- Seit ca. 2005 Mehrkernprozessoren auch im PC

D.h. weitere Leistungssteigerungen nur noch durch manuelles Parallelisieren des Codes

- Leider kann der Compiler das nicht (bzw. nur unzureichend)

## Wir parallelisieren den Code

- Konzept seit den 1970ern genutzt
- Seit den 1990ern Rechnercluster, dann auch mit Linux
- Wird später besprochen

## 2. Ebenen und Potentiale

### Mathematisch/algorithmische Ebene

- Welche alternativen mathematischen Verfahren sind in der Ausführung schneller?

### Programmiersprachliche Ebene

- Geeignete Verfahren
  - Welcher Algorithmus läuft am besten?
  - Welche Datenstrukturen sind am geeignetsten?
- Optimale Anpassung an die Architektur
  - Wieviel Hauptspeicher hat mein Rechner?
- Optimale Anpassung an den Compiler
  - Wie legt der Compiler die Daten im Speicher ab?



# Ebenen der Optimierung (2)

## Compilerebene

- Welche Optimierungen führt der Compiler durch?
- Wie kann ich Optimierungen gezielt auswählen?

## Hardware-Ebene

- Kann ich meinen Rechner an das Problem anpassen?
  - Z.B. Einbau von mehr Hauptspeicher
  - Z.B. Einbau von speziellen Beschleunigerkarten (GPGPU, FPGA)



# Benötigte Methodenkenntnisse

- Wissen über die Anwendung
  - Wissen zu mathematischen Verfahren
  - Wissen zu Programmiertechniken
  - Wissen über Compilerkonzepte
  - Wissen über Rechnersysteme
- + Wissen über das Zusammenwirken aller fünf Ebenen**

Wunschdenken des Naturwissenschaftlers

Mich kümmert nur meine Naturwissenschaft!

- Geht klar, aber dann wird das Werkzeug Computer nicht optimal eingebunden werden können

Disziplinabhängige Unterschiede: Physiker vs. alle anderen



# Optimierungspotentiale

## Mathematik

- Sehr hoch
  - Komplexitätsverringern der Verfahren bringt Größenordnungen

## Programmiertechnik

- Sehr hoch
  - Komplexitätsverringern der Algorithmen bringt Größenordnungen
  - Effiziente Datenstrukturen bringen vielleicht noch eine Größenordnung
  - Optimale Anpassung an eingesetzte Hardware bringt vielleicht auch noch eine Größenordnung

## Compiler

- Mittel
  - Optimierungen des Maschinencodes werden durchgeführt

# Optimierungspotentiale (2)

## Hardware-Umbauten im normalen Rechner

- Mittel bis hoch, aber schwierig in der Umsetzung

## Hardware-Umbau: Erwerb eines Hochleistungsrechners

- Sehr hoch: Faktor 10 bis 1.000.000
  - Schwierig in der Realisierung

## Wahl einer optimalen Programmiersprache

- Gering
- Komfort vs. Geschwindigkeit
- Mathematische und programmiertechnische Optimierungen mit jeder Sprache möglich
  - Pfusch ebenso!

# 3. Anwendungsklassen

## Geschäftssoftware (als Gegenbeispiel)

- Oft nicht zeitkritisch in der Ausführung, weil eher kurz
- Ggf. Einmaloptimierung und dann langer Produktionsbetrieb

## Wissenschaftliche Software

- Typischerweise oft zeitkritisch, weil komplexe Berechnungen
- Probleme
  - Ständiger Wandel des Codes, der z.B. ein mathematisches Modell realisiert (computergestütztes Experimentieren)
  - Wenig Produktionsbetrieb mit unverändertem Code
  - Wissenschaftler hat keine Zeit zur Codeoptimierung
  - Wissenschaftler hat keine Kenntnis über Möglichkeiten

# (Traurige) Tatsachen

## Kein systematisches Leistungs-Engineering

- Kenntnisse über Optimierungen auf allen Ebenen sind nur bruchstückhaft bei den Anwendern vorhanden
- Keine Lehre zu diesem Thema
- Nahezu keine schriftlichen Unterlagen
- Niemand weiß, wie schnell das Programm sein müsste

## Ausnutzung der nominellen Prozessorleistung gering

- In vielen Fällen werden nur 5-10% der **Rechenleistung** genutzt
- Gründe beispielhaft:
  - Sprünge im Code, nicht genügend Mathematik im Code
  - Indirektionen beim Speicherzugriff und schlechte Cache-Nutzung



## (Traurige) Tatsachen (2)

Wissenschaftliche Software meist schlecht optimiert

- Ergebnisse kommen zu langsam
- Ressourcen werden nicht optimal genutzt
  - Klimacodes für IPCC AR5 brauchen am DKRZ 30 MCPUh und das kostet 1 MEuro für Strom

Energiekosten sind jetzt ein wichtiger Faktor

- Nicht mehr alleine interessant: time-to-solution  
Sondern auch: kWh-to-solution



# Kosten/Nutzen-Analyse

---

## Kosten

- Einführung neuer Mathematik
- Verbesserung der Programmstruktur
- Installation optimierter Bibliotheken

## Nutzen

- Verkürzte Programmlaufzeit

## Sinnvolles Vorgehen

- Aufgewandte Zeit und gesparte Zeit in Relation setzen
- Unterm Strich sollte eine Ersparnis herauskommen
  - Aufwand für Optimierung an das Einsparpotential anpassen





# Die Wahl der Programmiersprache

C

- Gute Anpassung an Hardware möglich
- Programmierung vergleichsweise maschinennah
- Effizienter Maschinencode

C++

- Vorteile/Nachteile von C
- Zusätzliche objektorientierte Programmierung

Fortran

- Gute Anpassung an Mathematik
- Programmierung nicht so maschinennah wie C
- Trotzdem effizienter Maschinencode



# Die Wahl der Programmiersprache (2)

## Java

- Gute Programmierkonzepte und hoher Programmierkomfort
- Keine optimale Anpassung an die Hardware
- Keine optimale Anpassung an die Mathematik
- Einigermaßen effizienter Maschinencode

## Skriptsprachen / Matlab etc.

- Schnelle Programmerstellung möglich
- Komfort geht auf Kosten der Laufzeitoptimierung

## Zusammenfassung

- Leistungsausbeute bei Sprachen hängt eher vom Vermögen des Programmierers ab
- Objektorientierung kostet Leistung und bringt Komfort



# Ideale Vorgehensweise

- Sauberer Entwurf der Mathematik und der Implementierung
  - Nicht nur wegen Laufzeit sondern auch wegen
    - Fehlerfreiheit, Wartbarkeit, Erweiterbarkeit
- Messen der Programmlaufzeiten
- Bewerten
  - Kann ich mit dieser Laufzeit meine wissenschaftliche Arbeit durchführen?
- Aufdecken von Leistungsengpässen
  - Welche Werkzeuge gibt es?
- Beseitigung von Leistungsengpässen
  - Die wichtigsten zuerst

## 4. Leistungsabschätzungen

### Theoretisch

- Komplexitätsmaße für Zeit- und Speicherbedarf ermitteln  
Sehr schwierig – am besten ggf. Literatur heranziehen

### Praktisch

- Laufzeiten und Speichernutzungen messen  
Das kann jeder

# Theoretische Leistungsabschätzungen

Lernt der Informatiker in der Komplexitätstheorie

In aller kürze:





- Zeitbedarf und Speicherbedarf haben eine funktionale Abhängigkeit von der Anzahl und Größe der Eingabedaten
- Bezeichnet durch  $O(X)$ , wobei  $X$  eine Funktion von  $n$  ist

Beispiele (für Laufzeitkomplexität)

- $O(n)$ : Das Programm ist linear von  $n$  abhängig
  - Beispiel: Durchlaufen aller Eingabewerte und Maximum finden
- $O(n^2)$ : Das Programm ist quadratisch von  $n$  abhängig
  - Beispiel: Schlechte Sortierverfahren, die alle Werte mit allen vergleichen

# Theoretische Leistungsabschätzung (2)

## Auszug aus Sortierverfahren bei Wikipedia

Sortierverfahren 	Best-Case 	Average-Case 	Worst-Case 
<a href="#">AVL Tree Sort</a> (höhen-balanciert)	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$
<a href="#">Binary Tree Sort</a>	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$
<a href="#">Bogosort</a>	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot n!)$	$\infty$
<a href="#">Bubblesort</a> (Vergleiche) (Kopieraktionen)	$\mathcal{O}(n)$ $\binom{n-1}{0}$	$\mathcal{O}(n^2)$ $\approx \left(\frac{n^2}{4}\right)$ $\approx \left(\frac{n^2}{4}\right)$	$\mathcal{O}(n^2)$ $\approx \left(\frac{n^2}{2}\right)$ $\approx \left(\frac{n^2}{2}\right)$
<a href="#">Combsort</a>	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n^2)$
<a href="#">Gnomesort</a>	$\mathcal{O}(n)$		$\mathcal{O}(n^2)$
<a href="#">Heapsort</a>	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$	$\mathcal{O}(n \cdot \log(n))$

Wichtig: welches ist die minimale Komplexität?

# Theoretische Leistungsabschätzung (3)

## Weitere Beispiele

- $O(1)$ : Die Laufzeit ist fest und hängt nicht von  $n$  ab
  - Beispiel: Ein Programm, das immer gleich abstürzt 😊
- $O(\log n)$ : Die Laufzeit hängt logarithmisch von  $n$  ab
  - Beispiel: Suche in einem Binärbaum
- $O(n^k)$ : Die Laufzeit hängt polynomial von  $n$  ab
  - Beispiele: kaum welche mit Praxisrelevanz für  $k > 2$
- $O(2^n)$ : Die Laufzeit hängt exponentiell von  $n$  ab
  - Beispiele: viele theoretische, die praktisch nicht berechnet werden können

## Beachte

- Bei sehr kleinen  $n$  ist manchmal auch eine höhere Komplexität noch akzeptabel (z.B.  $O(n^2)$  beim Sortieren statt  $O(n \log n)$ )
- Die Bestimmung der Komplexität ist sehr komplex 😊



# Praktische Leistungsabschätzung

Es gibt verschiedene Messwerkzeuge

- Wenige für sequentielle Programme
- Einige komplexe für parallele Programme

Unter Linux

- Kommandos **time** (der Shell) und **/usr/bin/time**
  - Letzteres zeigt auch den Speicherverbrauch und andere Daten
  - Einfach auf der Kommandozeile dem Programmaufruf voranstellen
  - Ermittelt die Gesamtlaufzeit des Programms
- Kommando **gprof**
  - Programm mit Compileroption für Profiling übersetzen (gcc: -pg)
  - Laufenlassen des Programms erzeugt Datei gmon.out
  - Kann mit gprof angesehen werden
- Kommando **perf**



# Praktische Leistungsabschätzung (2)

## Beispiel (Hager/Wellein):

%	cummulative	self		self	total	
Time	seconds	seconds	calls	ms/call	ms/call	name
70.45	5.14	5.14	26074562	0.00	0.00	intersect
26.01	7.03	1.90	4000000	0.00	0.00	shade
3.72	7.30	0.27	100	2.71	73.03	calc_tile

## Erläuterung

- self seconds ist die Laufzeit in der Funktion
- cummulative seconds ist die aufsummierte Laufzeit, wenn nach self seconds sortiert wird

# Praktische Leistungsabschätzung (3)

## Vorgehensweise

- Wir optimieren die Funktionen mit dem höchsten Zeitanteil
  - Hier zunächst intersect
- Eine Abschätzung des Optimierungspotentials der einzelnen Funktionen gibt Aufschluss über das Gesamtpotential

## Aspekte von gprof

- Funktionsbasiert – d.h., wer sein Programm nicht in Funktionen unterteilt, kann nichts messen 😊
- Inlining von Funktionen durch den Compiler muss korrekt behandelt werden, sonst sind die Messwerte falsch
  - Inlining: der Compiler ersetzt im Maschinencode einen Funktionsaufruf durch die Funktion selber



## 5. Optimierung der Mathematik

- Am besten zusammen mit den Mathematikern
  - Kooperationen mit Numerikern/Optimierern
- Bessere mathematische Verfahren brauchen Zeit für die Entwicklung und Evaluation
- Kann oft nicht vom Naturwissenschaftler geleistet werden
  - Muß aber in Zusammenarbeit mit ihm erfolgen, da meist die Kenntnis der Anwendung von Nöten ist

## Optimierung der Mathematik (2)

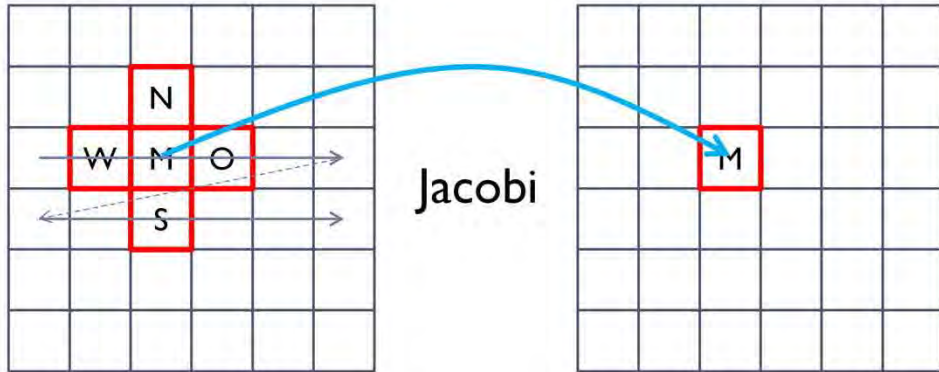
Beispiel: partielle Differentialgleichungen mittels Jacobi- oder Gauß/Seidel-Verfahren

- Löst ein lineares Gleichungssystem
- Z.B. für folgende Anwendung: wir erwärmen eine Platte an den Ecken auf eine bestimmte Temperatur – wie ist dann die Verteilung der Temperatur über die Platte hinweg?

Bewertung:

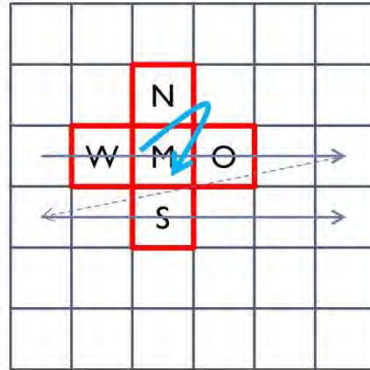
- Gauß-Seidel-Verfahren konvergiert schneller
- Seit neuestem aber: Jacobi lässt sich für hohe Anzahl von Prozessoren besser parallelisieren

# Optimierung der Mathematik (3)



- Es werden zwei Matrizen verwandt: eine mit den aktuellen Werten und eine für die Werte der nächsten Iteration
- Der neue Wert M wird aus den alten Nachbarwerten von M (N, W, S und O) ermittelt
- Wenn alle neuen Werte bestimmt sind, werden die beiden Matrizen getauscht und die nächste Iteration beginnt
- Das Verfahren endet, wenn für alle neuen M die Änderung kleiner einer unteren Schranke ist

# Optimierung der Mathematik (4)



Gauß-Seidel

- Nur eine Matrix verwandt, also weniger Speicher *und* auch schneller
- Der neue Wert M wird aus den Nachbarwerten von M (N, W, S und O) ermittelt
- Hier jetzt: N und W wurden bereits aktualisiert, S und O noch nicht. Das mathematische Verfahren läuft somit anders ab
- Das Verfahren endet, wenn für alle neuen M die Änderung kleiner einer unteren Schranke ist



# 6. Optimierung der Programmierung

Am besten zusammen mit den Informatikern

## Drei Ebenen

- Pure Programmierung ohne Berücksichtigung von Compiler und Hardware
- Programmoptimierung im Zusammenspiel mit dem Compiler
- Programmoptimierung im Zusammenspiel mit der Hardware

## Allgemeine Probleme

- Bei einem Wechsel der Zielarchitektur müssen die Optimierungen erneut evaluiert und dann angepasst oder ausgetauscht werden
- Dasselbe gilt bei einem Wechsel auf parallele Architekturen



# Optimierung der Programmierung (2)

## Unabhängig von Compiler und Hardware

- Effiziente Algorithmen
  - Effizientes Sortieren, effizientes Suchen
- Effiziente Datenstrukturen
  - Listen, Bäume, Hashtabellen, dünn besetzte Matrizen

## Findet man in Büchern und Vorlesungen

- Informatikergrundvorlesung „Algorithmen & Datenstrukturen“
  - Das Minimum dessen, was der Naturwissenschaftler wissen sollte!
- Amazon: „Algorithmen und Datenstrukturen“

# Optimierung der Programmierung (3)

## Beispiel: dünnbesetzte Matrizen

- $N \times N$  Einträge, aber nur 0,1% sind ungleich von null

## Speicherung

- Ablage in einer verketteten Liste (einfach oder doppelt) mit Angabe der  $x, y$ -Koordinate
- Zusätzlich noch ein Feld mit Zeigern auf z.B. jedes 1000ste Element

## Zugriff

- Einstieg an einem der Zeiger, Ablaufen der Liste bis zur gewünschten Koordinate

## Matrizenmultiplikation

- Wird jetzt ganz neu implementiert

# Optimierung der Programmierung (4)

Abhängig vom Compiler

Beispiel:

- Abbildung von logischen Datenstrukturen in den Hauptspeicher
- Hier: zweidimensionale Felder
- Z.B. wird zeilenweise in den Hauptspeicher abgebildet
- Programm lese z.B. die Werte zeilenweise oder spaltenweise
  - Was passiert mit der Zugriffszeit?
- Man würde meinen: gar nichts – wäre da nicht der Cache
  - Der holt sich nicht nur den fehlenden Wert sondern noch mehrere andere

# Optimierung der Programmierung (5)

1	2	3
4	5	6
7	8	9

logische  
Datenstruktur

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Speicherabbild

	a	b	c
Cache	1	2	3
	x	y	z

Cacheinhalt  
nach Zugriff auf  
Element '1'

- Nach Zugriff auf '1' ist eine Cacheline geladen, gerade eben die Elemente '1', '2' und '3'
- Greift das Programm auf '2' und '3' zu, so geht dies schnell (Cache-Hits)
- Greift das Programm nach '1' auf '4' zu, so muß eine zweite Cacheline '4', '5' und '6' geladen werden
  - Das kostet Zeit! (Cache-Miss)
  - Dasselbe Problem wiederholt sich, wenn dann auf '7' zugegriffen wird

# Optimierung der Programmierung (6)

Abhängig von der Hardware

Beispiel 1:

- Wir haben 2 GB Hauptspeicher
- Das Programm hat aber viele GB virtuellen Adreßraum
- Daten, die nicht im Hauptspeicher gehalten werden können, werden auf die Platte ausgelagert (Swapping)
- Das kostet Zeit!
- Also: Datenstrukturen des Programms an freien Platz anpassen

Beispiel 2:

- Im Rechner ist eine zusätzliche Grafikkarte verbaut
- Wir könnten diese zur Beschleunigung von Berechnungen verwenden

Beispiel 3:

- Das Programm wurde für einen 32bit-Prozessor entwickelt
- Es soll jetzt auf einem 64-bit Prozessor laufen
- Im einfachsten Fall Neuübersetzung für den neuen Zielprozessor (der kommt nie vor ☺)

## 7. Optimierung mit dem Compiler

Alle Compiler haben aufwendig Codeoptimierungen eingebaut

- Manche sind unabhängig vom Zielprozessor
- Manche sind genau auf Befehlssätze, Register usw. zugeschnitten

Der Programmierer kann per Optionen verschiedene Optimierungsstufen auswählen

- Weiß dann mehr oder weniger, was passiert. Eher weniger



# Optimierung mit dem Compiler (2)

## Optimierungsoptionen für den GNU-C-Compiler `gcc`

-O0

führe keine Optimierungen durch

-O1

der Compiler versucht, Codegröße und Programmlaufzeit zu verringern, ohne die Übersetzungszeit wesentlich zu erhöhen

-O2

mehr Optimierungen aber kein Inlining, kein Loop Unrolling  
Code wird schneller, Übersetzungszeit steigt

-O3

Inlining wird auch aktiviert

-Os

Wie -O2, aber ohne Optimierungen, die den Code vergrößern



# Optimierung mit dem Compiler (3)

## Zwei Beispiele für Optimierungsverfahren

- Inlining von Funktionen
  - Der Sourcecode einer Funktion wird übersetzt und überall da direkt eingebaut, wo die Funktion aufgerufen wird
  - Zeitaufwendige Sprünge entfallen
  - Maschinencode wird länger
- Loop Unrolling
  - Wenn eine Schleife z.B. 10x durchlaufen wird, dann wird der Code 10x hintereinander abgelegt
  - Zeitaufwendige Sprünge entfallen
  - Maschinencode wird länger

# Optimierung mit dem Compiler (4)

## Wann wähle ich welche Stufe?

- Phase der Fehlersuche: immer mit -O0
  - Ansonsten sind die Umbauten im Code für die Fehlersuche hinderlich, weil Code umgestellt, zum Teil eliminiert wird usw.
  - Eine eindeutige Zuordnung zu den Zeilen des Quellcodes ist dann nicht mehr möglich
- Phase des Profiling: ohne Funktionen-Inlining
  - Nicht alle Level sind mit der Funktionsweise des gewählten Profiler kompatibel
  - Im Einzelfall das Handbuch lesen
- Phase des Produktionsbetriebs z.B. mit -O3
  - Manchmal treten aber Fehler auf, die aus einem komplexen Zusammenspiel zwischen maschinennaher Programmierung und Compileroptimierung entstehen
  - Dann den Optimierungslevel heruntersetzen

## 8. Fazit

Es kann nur in Zusammenarbeit der Disziplinen eine Verbesserung erzielt werden

Anwendungswissenschaftler – Informatiker – Mathematiker

Viel Wissen für eine optimale Optimierung nötig

- Der Einzelne weiß dazu meist zu wenig
- Ist aber keine Entschuldigung, jegliches Wissen abzuweisen

Aufwandsabschätzung

- Was nützt es mir bei meiner Abschlussarbeit?
- Was kostet mich das?
- Was kostet es in der Folge Dritte?



# To-Do-Liste

---

## Was muss ich wissen?

- Grundlagen der Komplexität bzgl. Zeit- und Speicherbedarf
- Wichtige Datenstrukturen und wichtige Algorithmen
- Kenntnis über das Ausmessen von Programmen
- Kenntnis über wichtige Aspekte der Compileroptimierung

## Was muss ich tun?

- Ein Buch zu „Algorithmen und Datenstrukturen“ besorgen und nach nützlichen Konzepten durchsehen
- **time** und **gprof** ausprobieren und einüben
- Programme regelmäßig bzgl. Ihrer Leistung analysieren und wichtige Einsichten aufschreiben
- Diskussion mit anderen Entwicklern über dieses Thema

# Optimierung sequentieller Programme

## Zusammenfassung

- Es gibt keine systematischen Ansätze zur Optimierung von sequentiellm Code
- Die Optimierungen finden auf der Ebene der Mathematik, der Programmierung und des Compilers statt
- Die Optimierungspotentiale sind da durchwegs sehr hoch
- Die konkrete Wahl der Programmiersprache ist weniger wichtig (solange es eine Übersetzer-Sprache ist)
- Mit der Komplexitätstheorie schätzt man Laufzeiten und Speicherbedürfnisse von Algorithmen ab
- Zur konkreten Programmanalyse gibt es bei Linux verschiedene Werkzeuge
- Optimierungen der Mathematik können die Laufzeit deutlich verbessern
- Optimierungen bei der Programmierung können die Laufzeit deutlich verbessern
- Optimierungen mit dem Compiler können die Laufzeit deutlich verbessern



# Optimierung sequentieller Programme

## Die wichtigsten Fragen

- In welchen Fällen versuche ich, die Leistung zu steigern?
- Auf welchen Ebenen setzt eine Optimierung an?
- Wie sind hier die Optimierungspotentiale?
- Welche Kenntnisse muss ich haben?
- Wie schätze ich die theoretische Leistungsfähigkeit ab?
- Wie schätze ich die praktische Leistungsfähigkeit ab?
- Wie optimiere ich die Mathematik?
- Wie optimiere ich auf der Programmebene?
- Wie nutze ich die Compileroptimierungen?
- Was muss ich alles lernen?