Programmierkonzepte in der Physikalischen Chemie

21. Codeoptimierung

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2018/19

Zentrale Aspekte

- Laufzeit und Ressourcenverbrauch sind in den seltensten Fällen ein ernsthaftes Problem.
- Code sollte nur dann optimiert werden, wenn es unumgänglich ist.
- Voraussetzungen für erfolgreiche Optimierung sind Messbarkeit, Tests und Versionsverwaltung.
- Optimierungsstrategien hängen oft von Architektur, Programmiersprache und Compiler ab.
- Optimierungen haben häufig (ungewollt) Einfluss auf die Funktionalität.

Übersicht

UNI FREIBURG

Motivation

Voraussetzungen

Strategien

Zusammenhang mit „Sauberem Code“

Warum Code optimieren?

Es gibt wenige wirklich gute Gründe.

UNI FREIBURG

> *...the first principle of optimization is* don't.
>
> – Kernighan und Pike

Zwei Motivationen für Codeoptimierung

- Steigerung der Ausführungsgeschwindigkeit
- Verringerung des Ressourcenverbrauchs (meist Speicher)

Meist sind beide Aspekte nicht von Belang.

- Hardware wird schneller und billiger.
- Compiler sind mittlerweile sehr gut.
- Meist wartet der Computer eh auf den Nutzer.

Kernighan und Pike: The Practice of Programming, Addison-Wesley, Boston 1999, S. 165

Wenn Optimierung einen Unterschied macht

Beispiele aus den Naturwissenschaften

UNI FREIBURG

- Anpassung komplexer Simulationen an Daten
  - iterativer Vorgang: viele Aufrufe der Simulationsfunktion
  - Laufzeit der Simulationsroutine ist entscheidend.
  - Faktor 5–10 kann über Durchführbarkeit entscheiden
- Pulvermittelung in Simulationsprogrammen
  - Mittelung über Simulationen für viele Orientierungen
  - Punkte gleichmäßig auf einer Kugel zu verteilen ist schwer.
- quantenmechanische Behandlung vieler Teilchen
  - häufig mit innerem Produkt zweier Matrizen ($\otimes$) verbunden
  - Matrixdimensionen steigen quadratisch an: Speicherplatz!

☛ Die Lösung ist in allen Fällen meist keine Codeoptimierung, sondern eine intelligente Wahl von Algorithmen.

Wenn Optimierung einen Unterschied macht

Wenn Matrizen den vorhandenen Speicher „sprengen“

UNI FREIBURG

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} a\alpha & a\beta & b\alpha & b\beta \\ a\gamma & a\delta & b\gamma & b\delta \\ c\alpha & c\beta & d\alpha & d\beta \\ c\gamma & c\delta & d\gamma & d\delta \end{pmatrix}$$

Beispiel

- Für N Teilchen ist die Matrix-Dimension 2^N.
- Jedes Feld in der Matrix ist eine 64-Bit-Gleitkommazahl.
- Für $N = 14$ sind das 2^{14^2} Felder, also ca. 2×10^9 Byte.

☛ Für $N > 15$ wird es auch mit moderner Hardware eng…

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c > 0\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0 \,:\, f(n) \leq c \cdot g(n)\}$$

- $g(n)$ ist die obere asymptotische Schranke von $f(n)$.
- $\mathcal{O}(g(n))$ ist die Menge aller $f(n)$,
  für die $g(n)$ eine asymptotische obere Schranke ist.

Anwendung in der Informatik

- gilt sowohl für Ausführungsgeschwindigkeit als auch für Ressourcenverbrauch (meist Speicher)
- gibt jeweils den „*worst case*“ (obere Grenze) an
- sagt nichts über die reale Geschwindigkeit eines Algorithmus für fixes n aus

Komplexität von Algorithmen

Typische Beispiele für die Ausführungsgeschwindigkeit

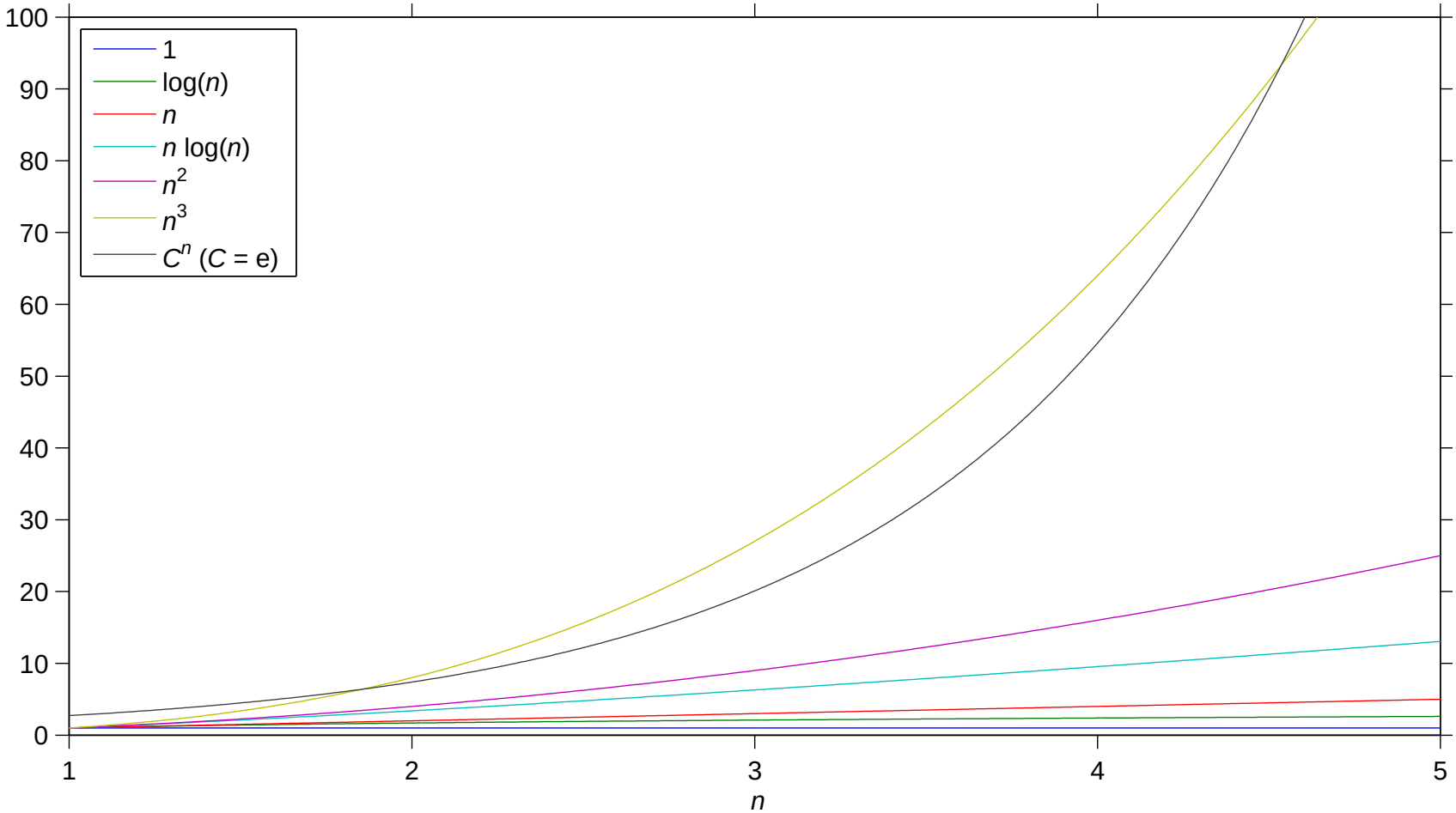
UNI FREIBURG

Ordnung	typisches Beispiel
$\mathcal{O}(1)$	Zugriff auf Vektorelement
$\mathcal{O}(\log(n))$	Binärsuche
$\mathcal{O}(n)$	sequentielle Suche
$\mathcal{O}(n\log(n))$	Quicksort, Heapsort
$\mathcal{O}(n^2)$	Selectionsort
$\mathcal{O}(n^3)$	Multiplikation zweier $n \times n$-Matrizen
$\mathcal{O}(C^n)$	Problem des Handlungsreisenden

- Häufig lässt sich $\mathcal{O}(n^2)$ zu $\mathcal{O}(n\log(n))$ reduzieren.
- Algorithmen mit $\mathcal{O}(n^2)$ Speicherbedarf sind problematisch.

Komplexität von Algorithmen

Grafische Repräsentation einzelner Ordnungen



☛ Die Größe von n kann entscheidend sein.

Bevor man anfängt zu optimieren

Vier essentielle Voraussetzungen für Codeoptimierungen

UNI FREIBURG

> *A fast program that gets the wrong answer doesn't save any time.*
>
> – Kernighan und Pike

- funktionierender Code
- Tests mit ausreichender Testabdeckung
- Versionsverwaltung
- Messungen der Codegeschwindigkeit

- (fast) gleiche Voraussetzungen wie für Refactoring
- Erst Geschwindigkeit messen, (*nur*) dann optimieren.

Kernighan und Pike: The Practice of Programming, Addison-Wesley, Boston 1999, S. 165

Grundvoraussetzungen

Letztlich die gleichen Voraussetzungen wie die von Refactoring

UNI FREIBURG

- funktionierender Code
  - möglichst in einer Hochsprache entwickeln
  - Code sollte das gesamte System abbilden.
  - Optimierung ist oft auf Systemebene effektiver.
- Tests mit ausreichender Testabdeckung
  - Codeoptimierung führt oft zu subtilen Fehlern.
  - Nur automatisierte Tests können ausreichend sicherstellen, dass Code (immer noch) funktioniert.
  - Tests sollten nach jedem Optimierungsschritt durchlaufen.
- Versionsverwaltung
  - Codeoptimierung ist nie ein linearer Prozess.
  - Funktionierende Versionen des Codes dienen als Referenz.

☛ Es gibt vor dem Optimieren erstmal genug zu tun. . .

Messungen der Codegeschwindigkeit

Intuition führt aus Erfahrung meist in die Irre.

UNI FREIBURG

> *...intuition is a poor guide to where the bottlenecks are, even for one who knows the code in question intimately.*
>
> – Eric S. Raymond

- Regeln zur Messung der Codegeschwindigkeit
  - Werkzeuge verwenden (Profiler etc.)
  - mehrfach messen (Streuung ist teilweise erheblich)
  - Messergebnisse sauber mitdokumentieren (Logbuch!)

- Meist sind weniger als fünf Prozent des Codes für 90 Prozent der Ausführungszeit verantwortlich.
- Intuition weist meist vollständig in die Irre.

Raymond: The Art of UNIX Programming, Addison-Wesley, Boston 2004, S. 288

Strategien zur Codeoptimierung

Konkrete Strategien hängen von sehr vielen Parametern ab.

UNI FREIBURG

> *The most effective way to optimize your code is to keep it small and simple.*
>
> – Eric S. Raymond

- Problem mit konkreten Optimierungsstrategien:
  - von vielen Parametern abhängig
  - gelten jeweils nur für wenige Spezialfälle
- mögliche Abhängigkeiten – eine Auswahl:
  - Programmiersprache und Version
  - Compiler, dessen Flags und Version
  - verwendete Bibliotheken und deren Versionen
  - Architektur

Raymond: The Art of UNIX Programming, Addison-Wesley, Boston 2004, S. 290

Allgemeines Vorgehen

Strukturiert, überprüfbar, iterativ

UNI FREIBURG

1. bestehende Version sichern (VCS)
2. Geschwindigkeit messen (Profiler o.ä.)
3. Grund für schlechte Leistung herausfinden
4. entdeckten Flaschenhals beheben (Refactoring)
5. nach jedem Einzelschritt Geschwindigkeit messen
6. wenn erfolglos: zurück zur gesicherten Vorversion

- ☛ Meist sind über die Hälfte der Versuche erfolglos.
- ☛ (Erfolgreiche) Optimierung ist ein iterativer Prozess.
- ☛ nur optimieren, wenn es wirklich angebracht ist
- ☛ saubere Buchführung über erreichte Geschwindigkeiten

Konkrete Strategien

Wie man Programme eventuell beschleunigen kann

UNI FREIBURG

- Compileroptimierungen verwenden
  - Moderne Compiler bieten oft Optimierungsmöglichkeiten.
  - Optimierung durch Compiler ist oft besser als per Hand.
- teure durch billige Berechnungen ersetzen
  - abhängig von der jeweiligen Programmiersprache
  - Division ist meist „teurer“ als Multiplikation.
- häufige Berechnungen nur einmal durchführen
  - Bsp.: Berechnung eines festen Wertes in einer Schleife
  - Speicherplatz ist oft ausreichend vorhanden.
- Werte vorberechnen
  - Bsp.: langwierige Berechnungen fixer, oft genutzter Werte
  - Werte können ggf. aus einer Datei eingelesen werden.

Konkrete Strategien

Wie man Programme eventuell beschleunigen kann

UNI FREIBURG

- Reihenfolge verschachtelter Schleifen optimieren
  - Schleifeninitialisierung kostet Rechenzeit.
  - Innere Schleifen sollten häufiger laufen als äußere.
- Näherungen statt exakter Werte
  - Nicht immer ist die maximale Genauigkeit notwendig.
  - Genauigkeit korreliert direkt mit Speichergröße.
- Auslagern in maschinennähere Sprache
  - setzt Kenntnisse in mehr als einer Sprache voraus
  - kann entscheidende Geschwindigkeitsvorteile bringen
  - wichtig: besonderer Fokus auf größtmögliche Lesbarkeit

☛ Kenntnis der jeweiligen Sprache und ihrer Eigenheiten und genaue Messung der Laufzeiten ist essentiell.

Konkrete Strategien

Intime Kenntnis der Problemstellung und Vereinfachung

UNI FREIBURG

These

Die intelligente Wahl von Algorithmen ist meist wesentlich effizienter als jede Strategie zur Codeoptimierung.

- Näherungen sind oft die einzige gangbare Möglichkeit.
  - Die Grenzen der Näherungen müssen bekannt sein.
  - Ein Einsatz jenseits der Grenzen verfälscht die Ergebnisse.
  - Der Anwender ist für den korrekten Einsatz verantwortlich.
- Näherungen setzen ein tiefes Verständnis voraus.
  - Verantwortungsbereich der Naturwissenschaftler
  - Realität: Programme häufig als „*black box*“ eingesetzt

☛ Verantwortung des Programmierers:
Grenzen der Näherungen sauber dokumentieren

Keine Optimierung ohne sauberen Code

Sauberer Code ist eine zwingende Voraussetzung für Optimierung.

UNI FREIBURG

> *...the only result of optimization you can usually be sure of without measuring performance is that you've made your code harder to read.*
>
> – Steve McConnell

- „Sauberer Code“ ist (zwingende) Voraussetzung für erfolgreiche Optimierung.
- Optimierende Compiler tun sich oft wesentlich leichter mit sauber geschriebenem Code als mit „handoptimiertem“.
- Ausführungsgeschwindigkeit ist kein Grund für schlechte Lesbarkeit und schlecht gepflegten Code.

McConnell: Code Complete, Microsoft Press, Redmond 2004

Optimierung vs. sauberer Code

Sorgfältige Kosten-Nutzen-Abwägung von Optimierung

UNI FREIBURG

- Optimierung und Lesbarkeit stehen oft im Widerspruch.
  - Die schnellste Implementierung ist selten die lesbarste.
  - Lesbarkeit hat immensen Einfluss auf die Qualität.
- Lesbarkeit ist meist wichtiger als Geschwindigkeit.
  - Lesbarkeit sorgt für bessere Erweiterbar- und Wartbarkeit.
- Optimierung erfordert besonderen Fokus auf Lesbarkeit.
  - Es gibt auch bei optimiertem Code noch genug Potential.
  - Bsp.: gute Namen und übersichtliche Codeformatierung
- Optimierung schränkt oft die Plattformunabhängigkeit ein.
  - immer den nichtoptimierten, portablen Code behalten

☛ Optimiert werden sollte nur, wenn es unumgänglich ist, und immer nur soviel wie unbedingt nötig.

Optimierung vs. sauberer Code

Ansprüche an die Wissenschaftlichkeit der Datenauswertung

UNI FREIBURG

- Wissenschaftlichkeit setzt Nachvollziehbarkeit voraus.
  - Lesbarkeit von Quellcode ist essentiell.
  - Codeoptimierung geht meist zulasten der Lesbarkeit.
  - Lesbarkeit sollte *vorher* trainiert werden.
- Codeoptimierung ist fast nie relevant.
  - (Fast) nur für aufwendige Simulationen notwendig.
  - (Fast) niemand schreibt selbst Simulationsprogramme.
- Die Wahl des Algorithmus ist oft entscheidend.
  - Setzt tiefe Kenntnis der grundlegenden Mathematik voraus.
  - Viele Algorithmen sind in Bibliotheken implementiert.

☞ Codeoptimierung sollte nicht mit den Ansprüchen an die Wissenschaftlichkeit des Quellcodes kollidieren.

Zentrale Aspekte

- Laufzeit und Ressourcenverbrauch sind in den seltensten Fällen ein ernsthaftes Problem.
- Code sollte nur dann optimiert werden, wenn es unumgänglich ist.
- Voraussetzungen für erfolgreiche Optimierung sind Messbarkeit, Tests und Versionsverwaltung.
- Optimierungsstrategien hängen oft von Architektur, Programmiersprache und Compiler ab.
- Optimierungen haben häufig (ungewollt) Einfluss auf die Funktionalität.