universität freiburg

Algorithmen und Datenstrukturen SS 2025

Vorlesung 11: Algorithm Engineering

Dienstag, 15. Juli 2025

Dr. Patrick Brosi und Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen mit dem Ü10 — DA und Routenplaner
  - Vorgucker auf das Ü11 — Schnelleres MergeSort
- Inhalt
  - Algorithm Engineering — kleine Änderungen, große Wirkung
  - Programmiersprachen — Vergleich: Python, Java, C++
  - Compileroptimierung — Source code ⇆ Maschinencode
  - Maschinencode — Geschichte + Crashkurs

Erfahrungen mit dem Ü10 1/2

- Auszüge aus Ihrem Feedback [halbwegs repräsentativ]

  "Mit echten Datensätzen macht das immer besonders viel Spaß"

  "Hat sehr viel Spaß gemacht, aber deutlich länger gedauert"

  "Very interesting topic, happy for the application of all the data structures"

  "Spaß gemacht da es auch mal eine greifbare Anwendung hatte"

  "Cooles Übungsblatt, aber schwer gewesen zu überprüfen wo etwas schief geht"

  "Etwas nervig, dass die Musterlösung Pylance-Fehler macht"

Erfahrungen mit dem Ü10 2/2

- Edsger Dijkstra
  - "Brilliant, colourful, and opinionated"
  - Dijkstras Algorithmus: 20 Minuten Nachdenken während Café-Besuch

"Bevorzuge Persönlichkeiten Richtung Linus Torvalds"

"Spannend, dass Dijkstra lange keinen Computer benutzt hat"

"Interessant, dass er seine Arbeiten handschriftlich verfasst hat"

"Speziell, dass er Mozart hört. Von allen Komponisten Mozart?"

"Habe von einer Freundin Infos zu Dijkstra bekommen, da er mir anscheinend ähnlich ist"

https://www.cwi.nl/en/about/history/e-w-dijkstra-brilliant-colourful-and-opinionated/

Vorgucker auf das Ü11

- Das letzte Übungsblatt, natürlich auch Klausur 

  A1 MergeSort schneller machen

  Sie können dafür Ihre Abgabe für das Ü1 als Ausgangspunkt nehmen, oder die Musterlösung für das Ü1

  Sie können sich eine der folgenden beiden Varianten aussuchen:

  Variante A: **3 Änderungen**, die Ihren Code schneller machen (aber nicht die von Folie 7)

  Variante B: Implementierung in einer Sprache **Ihrer Wahl** (Java, C++, Go, Rust, Lisp, Plankalkül...)

Algorithm Engineering 1/3

- Einführung
  - Beim **Algorithm Engineering** geht es darum, die Lücke zwischen Algorithmentheorie und der Praxis zu schließen, dazu gehört zum Beispiel (heute nur ein kleiner Einblick):

    1. Kleine Veränderungen am Programmcode, ohne den grundlegenden Algorithmus zu ändern → nächste Folie
    2. Auf die Lokalität der Speicherzugriffe achten (sowohl RAM als auch HDD oder SSD) → V7 (Blockoperationen)
    3. Wahl der Programmiersprache → Folien 9 – 13
    4. Verständnis von Compiler, generiertem Maschinencode und der Maschine → Folien 14 – 17 und 18 – 26

    Um guten und schnellen Code zu schreiben, braucht man **beides**: Algorithmentheorie und Algorithm Engineering

Algorithm Engineering 2/3

- Schnelleres Mischen von zwei sortierten Listen ~ 200 ms
  - Wir ändern drei Sachen an unserem Code vom Ü1:

    1. len(A) und len(B) nicht immer wieder auswerten ~ 90 ms

    In manchen Sprachen findet der Compiler so etwas selber heraus, aber nicht in Python

    2. Ersetzen der inneren if durch while ~ 60 ms

    Das spart Operationen, wenn wir größere Stücke in einer Liste überspringen können

    3. Wir setzen Wächter ("sentinels") ans Ende der Listen ~ 50 ms

    Dadurch müssen wir nicht bei jedem Vergleich nachprüfen, ob wir schon am Ende der jeweiligen Liste sind

Algorithm Engineering 3/3

- Profiling
  - Ein "Profiler" ist ein Programm, das einem sagt, welcher Anteil einer Ressource in welchem Teil des Code verbraucht wird

    In dieser Veranstaltung interessiert uns vor allem **Laufzeit**, aber es kann z.B. auch um Speicherverbrauch gehen
  - In Python geht das zum Beispiel sehr einfach mit

    python3 –m cProfile –s time merge.py
  - Bei unserem Ausgangscode sehen wir damit zum Beispiel sofort, dass ein nicht unwesentlicher Teil der Laufzeit mit der Auswertung der "len" Funktion verbracht wird

    Das war ja auch unsere erste Verbesserung auf Folie 7

Programmiersprachen 1/5

- Python vs. Java vs. C++
  - Die Laufzeit hängt stark von der Programmiersprache ab
  - Wir implementieren jetzt zusammen folgenden sehr einfachen "Algorithmus" in verschiedenen Sprachen und verschiedenen Varianten:

    Fülle ein Feld mit 10 Millionen Zahlen vom Typ "int"

    Das ist durchaus realistisch, weil viele Programme einen Großteil ihrer Zeit mit der Iteration über solche Felder verbringen

    Für das Ü11 können Sie, anstatt die Python-Implementierung von MergeSort zu verbessern, den Code alternativ in einer Sprache Ihrer Wahl (die schneller als Python ist) schreiben

Programmiersprachen 2/5

- Python
  - Wir messen die Laufzeit des folgenden Codes, mit n = 10M

```
for i in range(0, n):
    A.append(i)
```

  - Wir vergleichen drei Datentypen für das Feld A:

Die eingebaute List: A = [] ~ 650 ms

Typ array.array [1]: A = array.array["i"] ~ 7200 ms

Typ numpy.array [2]: A =numpy.zeros(n, dtype=int) ~ 7200 ms

[1] Effizientere Implementierung für einfache Typen wie int

[2] numpy ist eine Python-Bibliothek für effiziente lineare Algebra

Programmiersprachen 3/5

- PyPy
  - Normaler Python Code wird **interpretiert**

    das heißt: während der Ausführung kompiliert, siehe Folie 15ff
  - PyPy möchte ein "drop-in replacement" für Python sein, das durch "Just-in-Time Compilation" (JIT) schneller ist

    python3 array_fill.py

    pypy3 array_fill.py
  - PyPy kann vieles, aber (noch) nicht alles von Python

    Zum Beispiel noch nicht: numpy, Python 3.10 pattern matching
  - PyPy ist in RPython geschrieben ("restricted" Python) und der RPython Interpreter ist in Python geschrieben

Programmiersprachen 4/5

- Java
  - Java hat Felder fester Größe und dynamische Felder (V6)

    Die Felder fester Größe funktionieren mit beliebigen Typen, die dynamischen Feldern unterstützen in Java keine Basistypen

    Statt **int** muss man dann zum Beispiel **Integer** verwenden, s.u.
  - Wir vergleichen diese beiden Implementierungen:

```
var A = new ArrayList<Integer>(n);
for (int i = 0; i < n; i++) { A.add(i); }
```

~ 750 ms

```
var A = new int[n];
for (int i = 0; i < n; i++) { A[i] = i; }
```

~ 25 ms

Programmiersprachen 5/5

- C++
  - Auch C++ hat native Felder (deren Größe zu Beginn feststeht) und dynamische Felder (deren Größe sich verändern kann)

    Anders als in Java funktionieren aber beide mit allen Typen
  - Wir vergleichen die beiden Implementierungen:

    ```
    std::vector<int> A(n);
    for (int i = 0; i < n; i++) { A[i] = i; }
    ```

    ~ 20 ms

    ```
    int* A = new int[n];
    for (int i = 0; i < n; i++) { A[i] = i; }
    ```

    ~ 17 ms
  - Der Code für std::vector<int> ist langsamer!

    Aber: Wir können std::vector<int> genauso schnell machen, wenn wir aufpassen und viel verstehen (typisch C++) → Folie 27

Compileroptimierung 1/4

- Grundprinzip eines Compilers
  - Der Code wird in eine entsprechende Folge von Anweisungen in Maschinencode übersetzt

    Kurze Einführung dazu auf den **Folien 18 – 26**
  - Im einfachsten Fall wird jede Zeile Code in eine Folge von Anweisungen in Maschinencode übersetzt

    Das ist korrekt, ergibt aber selten den schnellsten Code

    Das schauen wir uns jetzt mal anhand eines sehr einfachen Programms für alle drei Programmiersprachen an

Compileroptimierung 2/4

- Python
  - Python übersetzt ebenfalls in einen Bytecode

    Aber ein etwas anderer als der von Java
  - Den kann man sich in Python Interpreter anschauen mit z.B.

```
>>> import dis                  Modul zum "disassemblen"
>>> import simple               Unser Code
>>> print(dis.dis(simple))      Eine Funktion daraus
```

    Geht auch in einer Zeile mit:

```
python3 –c "import dis; import simple; print(dis.dis(simple))"
```

Compileroptimierung 3/4

- Java
  - Der Java-Compiler übersetzt erst in sog. Bytecode

    Ein abstrakter Maschinencode … siehe Folie 26
  - Den Bytecode kann man sich einfach anschauen mit

    javac Simple.java kompiliert zu Simple.class

    javap –c Simple Bytecode aus Simple.class
  - Dieser Bytecode wird dann zur Laufzeit in "richtigen" (auf der CPU direkt ausführbaren) Maschinencode übersetzt
  - Bei häufig benutzten Funktionen wird der Maschinencode wiederverwendet (wenn die Korrektheit sichergestellt ist)

Compileroptimierung 4/4

- C++
  - In C++ lässt sich der **Assembler**code leicht erzeugen mit

    g++ -S Simple.cpp

    Das gibt dann eine Datei Simple.s, die man sich einfach in einem Texteditor anschauen kann

    Wir schauen uns das auf https://godbolt.org an (schöner visualisiert, verschiedene Compiler, und vieles andere mehr)
  - Ohne Optimierung: der Code wird in der Tat Zeile für Zeile in Maschinencode übersetzt
  - Mit Optimierung: der Compiler tut erstaunliche Dinge

    Das meiste passiert schon bei –O1 (Stufe 1), mit –O3 (Stufe 3) werden die meisten Tricks aktiviert, die es gibt

Maschinencode 1/9

- Motivation
  - Das ist der Code, für den die CPU eigentlich gemacht ist
  - Code in einer höheren Sprache muss erstmal in Maschinencode übersetzt werden, damit man ihn ausführen kann

    Insbesondere Code in Python, Java oder C++
  - Anweisungen in Maschinencode sind durch Zahlen codiert
  - Die menschenlesbare Form davon nennt man **Assembler**

    Dafür sehen wir gleich zahlreiche Beispiele

Maschinencode 2/9

- Kurz zur Geschichte
  - 1972: Intel 8008 (der erste 8-Bit Mikroprozessor)
  - 1974: Intel 8080 (die ersten 16-Bit Operationen)
  - 1978: Intel 8086 (16 Bit, erstes Mitglied der x86 Familie)
  - 1985: Intel 80386 aka i386 (32 Bit)
  - 1993: Intel Pentium (32 Bit)
  - 2003: AMD 64, Intel 64 (64 Bit, manchmal x64 genannt)
  - Die sind alle rückwärts kompatibel bis zum Intel 8086

    Maschinencode, der 1978 für einen Intel 8086 kompiliert wurde läuft auch noch auf einem modernen Prozessor

    Grundprinzip über die Jahre unverändert ... nächste Folien

Maschinencode 3/9

- Register
  - Das sind Variablen, die es "in Hardware" in der CPU gibt
  - Die ursprünglichen Intel 8086 Register (**16 Bit**) heißen:

    AX, **BX**, **CX**, **DX** : "accumulator", "base", "counter", "data"

    SI, **DI**: "source index", "destination index"

    SP, **BP**: "stack pointer", "base pointer"
  - Die können im Prinzip alle für alles verwendet werden, haben aber für bestimmte Befehle / in bestimmten Kontexten eine besondere Bedeutung

    Zum Beispiel arbeiten viele Rechenoperationen auf **AX**

Maschinencode 4/9

- Register
  - Die Intel 80836 Register (32 Bit) heißen:

    EAX, **EBX**, **ECX**, **EDX**, etc. [E = extended]

    außerdem zusätzliche 64-Bit Register **MMX0**, **MMX1**, ...
  - Die AMD Opteron Register (64 Bit) heißen:

    RAX, **RBX**, **RCX**, **RDX**, etc. [R = register]

    außerdem zusätzliche 64-Bit Register **R8**, **R9**, ..., **R15**

    und sechzehn 128-Bit Register **XMM0**, **XMM1**, ...

    Die XMM Register (eins davon fasst 32 ints) sind wichtig für optimal schnellen std::vector<int> Code in C/C++

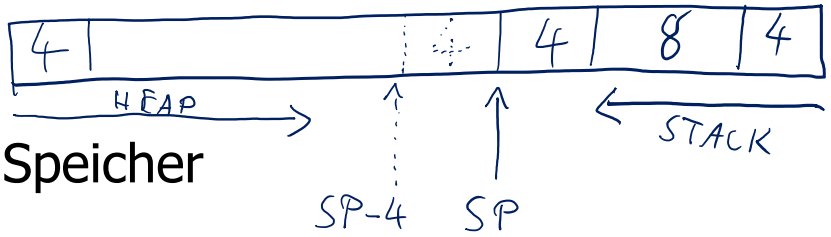
Maschinencode 5/9

- Heap und Stack
  - Es gibt zwei Arten von Speicher
  - **Heap:** wächst von "unten nach oben"

    Hier liegt alles, was während der Ausführung des Programms dynamisch alloziert wird (mit new)
  - **Stack:** wächst von "oben nach unten"

    Jeder Funktionsaufruf hat ein zusammenhängendes Stück auf dem Stack, da liegen:

    die Argumente, die lokalen Variablen, die Rücksprungadresse, die Adresse des Stücks Stack von der aufrufenden Funktion



Maschinencode 6/9

- Basisinstruktionen
  - mov X, Y : weise den Wert von Y an X zu

    Hier, wie auch bei vielen anderen Instruktionen, können X und Y Register sein oder auch Inhalt einer Stelle im Speicher, auf die ein Register zeigt

    Zum Beispiel ist **−4(%rsp)** oder **[rsp−4]** der Inhalt an der Adresse, die im Register RSP steht, minus 4

    Aber: X und Y können nie **beide** eine Speicheradresse sein

Maschinencode 7/9

- Arithmetische Operationen
  - Zum Beispiel:

    add, sub, mul, div, inc (increment), dec (decrement), ...

    and, or, xor, sal (shift left), sar (shift right), ...
  - Suffixe bei den Anweisungen:

    Kein Suffix = 16 bits, l = 32 bits ("long"), q = 64 bits ("quad")

    Beispiele: mov, movl, movq, add, addl, addq, ...

Maschinencode 8/9

- Operationen auf dem Stack
  - push X : X auf Stack legen … vermindert SP = stack pointer
  - pop X : X vom Stack holen … erhöht SP = stack pointer
- Vergleiche und Sprünge
  - cmp X, Y : vergleiche X und Y ob < oder > oder =
  - je X, jne X, jl X : springe nach X je nach < oder > oder =
  - jmp X : springe nach X ohne Bedingung

Maschinencode 9/9

- Java Bytecode
  - Ein abstrakter Maschinencode
  - Sehr ähnlich zu x86, aber bewusst einfach gehalten
  - Register heißen einfach 1, 2, 3, ...
  - Beispiel x86 Assembler (links) vs. Java Bytecode (rechts)

```
mov   eax, [rbp−4]      iload_1
mov   edx, [rbp−8]      iload_2
add   eax, edx          iadd
mov   ecx, eax          istore_3
```

Compileroptimierung (ctd.)

- C++, std::vector<int> ohne Laufzeitverlust
  - Durch einen Blick in den Assemblercode können wir jetzt verstehen, warum std::vector<int> langsamer war (Folie 17)
  - Der std::vector initialisiert seine Objekte standardmäßig, das heißt alle ints werden (unnötigerweise) auf 0 gesetzt

    Das können wir verhindern, indem wir unseren eigenen allocator verwenden, der die ints nicht initialisiert
  - Alternativ können wir auch nur Platz reservieren (wie in Java) und dann die Elemente anhängen

    Das ist aber immer noch 50% langsamer als int[n] und der Assemblercode sagt uns warum: durch das push_back kann der Compiler die XMM Register (Folie 21) nicht verwenden

    Hocheffizienten Code zu schreiben ist spannend, aber tricky

Literatur / Links

- Profiling mit cProfile
  - https://docs.python.org/3/library/profile.html
- Heap und Stack
  - http://en.wikipedia.org/wiki/Memory_management
  - http://en.wikipedia.org/wiki/Call_stack
- x86 Befehlssatz / Java Bytecode
  - http://en.wikipedia.org/wiki/X86_instruction_listings
  - http://en.wikipedia.org/wiki/Java_bytecode
- Godbolt compiler explorer
  - https://godbolt.org