18.11 Beispiel: Matrixmultiplikation

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$$

```
const int N = 16;
void matMult(double a[N][N], double b[N][N],
             double c[N][N])
{ int i, j, k;
  double col[N];
  for (i = 0;  i < N;  i++) {
    // aktuelle Spalte kompaktieren
    for (j = 0;  j < N; j++) {
      col[j] = b[i][j];
    }
    for (j = 0;  j < N; j++) {
      c[i][j] = scalarProduct(a[i], col);
    }
  }
}
```

Gut aber nicht gut genug.

18.11.1 Blocking

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ und } \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

Allgemein:

$$C_{ij} = \sum_{k} A_{ik} \cdot B_{kj}$$

(nicht notwendig quadratisch)

18.12 Beispiel: Hashing

(siehe auch Blatt 3, Aufgabe 2, sowie Vorlesung 3, Abschnitt 4.2) quasizufällige Funktion $h : \texttt{Key} \to 1..n$. Speichere unbekannte Teilmenge $M \subseteq \texttt{Key}$, $|M| = m$.

Idee: Speichere Element mit Schlüssel x in einem Array an der Stelle $h(x)$.

18.12.1 Geschlossenes Hashing

Jeder Array-Eintrag ist eine lineare Liste der dort kollidierenden Elements. Durchschnittliche Anzahl Zeigerdereferenzierungen:

$$T \approx 1 + m/n \ .$$

18.12.2 Offenen Hashing

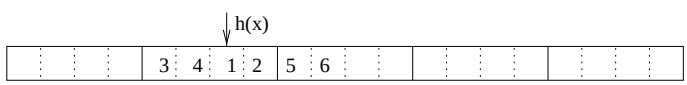
Speichere Einträge direkt. Kollisionsauflösung:

- Linear: Erfolgreiche Suche dauert etwa $\frac{1}{2}(1 + \frac{1}{1-m/n})$ Schritte
- Doppeltes Hashen: Weniger Suchschritte. Komplizierte Analyse. Besser?

18.12.3 Cache-Effekte

Geschlossenes Hashing und offenes doppeltes Hashing haben etwa genauso viele Cache-Faults wie Suchschritte. Lineares Hashing i. allg. nur einen!

Verfeinerung:



18.13 Beispiel: Zufallspermutationen

Eingabe: Array

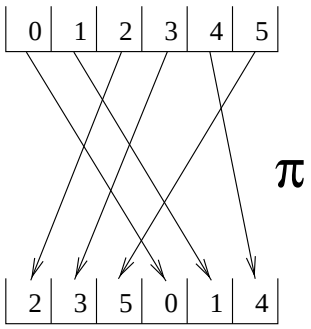
$$[e_0, \ldots, e_{n-1}]$$

Ausgabe:

$$\downarrow \pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$$

$$[e_{\pi^{-1}(0)}, \ldots, e_{\pi^{-1}(n-1)}]$$

π is gleichverteilte zufällige Permutation



18.13.1 Der „klassische“ Algorithmus

for $i := 0$ **to** $n-2$ **do**
swap e_i with a random e_j such that $j \geq i$

- Korrekt
- Zeit $\mathcal{O}(n)$
  - $\approx n$ Zufallszahlen
  - $\approx 2n$ Lesezugriffe
  - $\approx 2n$ Schreibzugriffe
- $\approx n$ Cache faults

18.13.2 Ein Cache-freundlicher Algorithmus

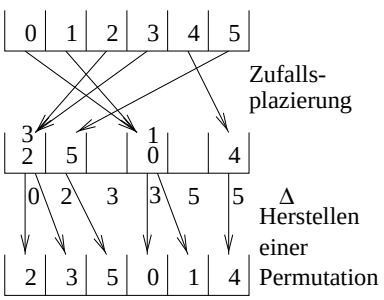
- M schneller interner Speicher
- B Cache-Line-Größe

werfe Elemente in P Buckets

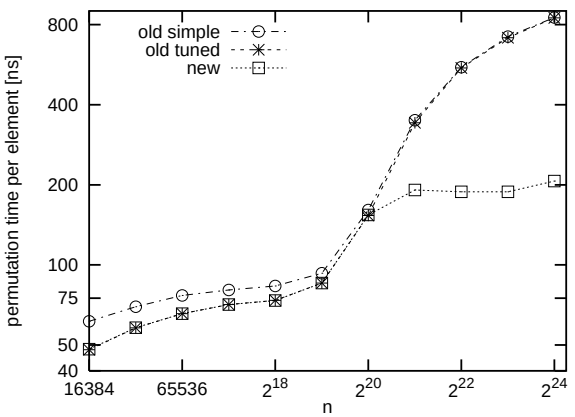
Buckets permutieren

buckets aneinander hängen

$\mathcal{O}(n/B)$ Cache faults — optimal



18.13.3 Implementierung



18.14 Tuning Werkzeuge

18.14.1 Profiler

- Übersetzen mit Option `-pg`. (richtige Optimierer-option und Debugoption kann problematisch sein (inlining))
- Programm ausführen. Realistische Eingaben verwenden! Als Seiteneffekt wir die Datei `gmon.out` erzeugt
- `gprof` aufrufen. Liefert folgende Informationen:
  - Welche Funktion verbraucht wieviel Zeit
  - Wie oft aufgerufen
  - Differenziert nach Aufrufern

Spürt zeitkritische Teile auf.

Schließt wenig lohnende Optimierungskandidaten aus.

Wie funktionieren Profiler

Aufrufzähler: Der Compiler setzt entsprechenden Code ein. (Instrumentierung)

Zeitmessung: Programmausführung wird periodisch unterbrochen (sampling) und nachgesehen welcher Programmteil gerade ausgeführt wird. (Warum nicht direkt messen?)

18.14.2 Assemblerausgaben

```
CC -fast -O4 -S sprod.C
```

- Mit Option `-S` übersetzen
- `datei.s` enthält den Assemblercode
- Nach relevanter Funktion suchen
- Inner loop aufspüren
- Überflüssige Speicherzugriffe?
- Ungeschicktes Instruction Scheduling?
- Ausreichende Inlining?
- Findet Ausrollen von Schleifen statt?
- Findet Pipelining statt?

Folgerung: Dem Compiler das Leben erleichtern oder seine Arbeit übernehmen. (Nicht immer portabel.)

18.15 Zusammenfassung

18.15.1 Allgemeines

- Kryptisch $\neq$ effizient
- Compileroptimierungen sind wichtig. (Aber nicht immer ist die höchste Optimierungsstufe die beste)
- Für jede Maschine neu übersetzen
- Bibliotheken verwenden (BLAS, `valarrays`, ...)
- Für *Gesamtanwendung* kritische inner loops mit Profiler bestimmen. Nur diese tunen
- An ausgewählten Stellen dem Compiler auf die Finger schauen

18.15.2 Programmstruktur

- `inline` (nicht nur wegen Aufrufoverhead
- `template` statt `virtual`
- Felder statt Listen (dynamic arrays, ...)
- Referenz versus Kopie (Parameter, Datenstrukturen)
- Lokale Variablen landen oft in Registern, members von `this` nicht!

18.15.3 Befehlsparallelismus ermöglichen

- Unvorhersagbare `if`s vermeiden
- Werte so früh wie möglich berechnen (vor allem für Bedingungen) kann z.T. der Compiler
- Datenabhängigkeiten vermeiden
- Software Pipelining
- Schleifen ausrollen

Nochmal Skalarprodukt

```
double scalarProduct2(const double *a,
                      const double *b, int n)
{ double acc0, acc1, acc2, acc3;
  double a0, a1, a2, a3,
         b0, b1, b2, b3,
         p0, p1, p2, p3;
  Assert3((n / 4) * 4 == n && m >= 8);
  acc0 = acc1 = acc2 = acc3 = 0.0;
  a1 = a[1]; b1 = b[1];
  a2 = a[2]; b2 = b[2];
  a3 = a[3]; b3 = b[3]; p1 = a1 * b1;
  for (const double *beyond = a + n - 4;
       a < beyond;  a+=4, b+=4) {
    p2=a2*b2; acc1+=p1; a0=a[0]; b0=b[0];
    p3=a3*b3; acc2+=p2; a1=a[1]; b1=b[1];
    p0=a0*b0; acc3+=p3; a2=a[6]; b2=b[6];
    p1=a1*b1; acc0+=p0; a3=a[7]; b3=b[7];
  }
  return acc0+acc1+p1+ acc2+a2*b2 + acc3+a3*b3;
}
```


18.15.4 Cache-Optimierungen

- Lokalität erhöhen
  - Loop fusing
  - Blocking
  - Sequentieller Zugriff statt zufällig (Beispiel: Hashing)
- Ganze Cache-Lines verwenden
  - Zeilen- versus Spaltenweise Speicherung
  - Structs versus mehrere Arrays
  - Ballast vermeiden (Gegenbeispiel: LEDA Graphen)
- Interference Misses vermeiden
  - konsekutive Bereiche allozieren (z.B. auf Stapel)
  - Padding
- Algorithmus ändern

19 Java

≈ Teilmenge von C++. Ziel: Einfacher, sicherer portabler. Ursprünglich für Haushaltsgeräte?

1995 als Web-Programmiersprache.

Überblick

- Ziele
- Unterschiede zu C
- Unterschiede zu C++
- New features
- Entwicklung und Ausführung von Java Programmen
- Die Java Libraries

Literatur: Java in a Nutshell von David Flannagan (2. Auflage, O'Reilly, 1997)

19.1 Ziele

- Portabilität (hat nicht wirklich was mit Interpretern zu tun)
- Einfachheit
- Sicherheit

Was man dafür opfert

(vor allem für die letzten beiden Punkte)

- Kompatibilität zu C/C++
- Einsatzspektrum/Flexibilität (z.B. hardwarenahe Programmierung)
- Performance
- Ausdrucksfähigkeit

19.2 Hello World

Datei `hello.java`

```
public class hello {
  public static void main(String argv[]) {
    System.out.println("Hallo");
  }
}
```

Und jetzt

```
>javac hello.java
>java hello
```

```
Hallo
```


19.3 Unterschiede zu C

Fundamentale Datentypen

- `char` ist 16-bit Unicode (ca. 38835 verschiedene Buchstaben)
- `double` und `float` auf IEEE festgelegt
- Wohldefinierte ganzzahlige Typen `byte`, `short`, `int`, `long` mit 8, 16, 32, respektive 64 bit.
  - Keine `signed`-Typen
- `boolean` ist kein Zahltyp! (Im Gegensatz zum `bool` von ISO C++

19.3.1 Hype: Java hat keine Pointer

Java hat *nur* Pointer:

Objektvariablen: Immer Referenzen (=verkappte Zeiger im Sinne von Pascal) Java: `SomeClass a` ↔ C++: `SomeClass& a`

Fundamentale Typen: Immer Werte. Auch als Parameter ⇒ Wertrückgabe nur über Funktionswert und Objektreferenzen

- Java kennt keine **Pointer-Arithmetik**! (sowie diverse unsichere casts etc.)
- Alle `*,&,->` aus C's Deklarationen und Ausdrücken sind überflüssig. (Syntaktisch gesehen ein Erlösung.)

19.3.2 Keine globalen Variablen

Statt dessen: static members

Zusätzlich: Packages (analog den `namespaces` von C++)

19.3.3 Kein Präprozessor

include: Aufgabe der Kopplung von Dateien und Modulen. `import` macht Namen eines Package sichtbar.

Konstanten: `static final members` (ähnlich `static const` in C++

Makros, inline Angeblich nicht nötig

Ein Gegenbeispiel

(NICHT unbedingt zur Nachahmung empfohlen)

```
void merge3(double **f0, double **f1,
            double **f2, double  *to, int sz)
{ double *from0=*f0, *from1=*f1, *from2=*f2;
  double *done    = to + sz;
  double key0=*from0, key1=*from1, key2=*from2;
  // startup code
  [...]
  Merge3Case(0, 1, 2); // key0<=key1<=key2
  Merge3Case(1, 2, 0); // etc.
  Merge3Case(2, 0, 1);
  Merge3Case(1, 0, 2);
  Merge3Case(0, 2, 1);
  Merge3Case(2, 1, 0);

 finish:
  *f0=from0; *f1=from1; *f2=from2;
}
```

Goto, Token Pasting und Lokalität

```
// handle case key##a <= key##b <= key##c
#define Merge3Case(a,b,c)\
  s ## a ## b ## c :\
  if (to == done)      goto finish;\
  to->key = key ## a; to++;\
  from##a ++; key##a = from##a;\
  if (key##a < key##b) goto s##a##b##c;\
  if (key##a < key##c) goto s##b##a##c;\
  goto                      s##b##c##a;
```

Solcher Code gehört in den Giftschrank! Aber in der rechten Dosierung ist fast jedes Gift auch Medizin [Paracelsus]

19.3.4 GOTO Statement Considered Harmful

[Dijkstra 68]

- Labelled `break` und `continue` ermöglichen das Verlassen geschachtelter Schleifen
- – Effiziente Implementierung endlicher Automaten schwierig.
- Ausnahmefallbehandlung: `setjmp`, `longjmp` von C durch Exception-Mechanismus ersetzt (ähnlich C++)