

5. Parallelprogrammierung II und q-gram Indizes

AIDaBi Praktikum

David Weese
© 2010/11

Enrico Siragusa
WS 2011/12

Inhalt

- Parallelprogrammierung II
- q-gram Indizes
- Bemerkungen zur P-Aufgabe

PARALLELPROGRAMMIERUNG II

OpenMP - Eine Einführung in die parallele Programmierung mit C/C++

Simon Hoffmann und Rainer Lienhart, <http://www.springer.com/computer/book/978-3-540-73122-1>

Einführung

- Es sollen Fibonacci-Zahlen berechnet werden:

```
vec[0] = 1;  
vec[1] = 1;  
for (int i = 2; i < vec.size(); ++i)  
    vec[i] = vec[i-1] + vec[i-2];
```

- Der Wert an Stelle i hängt offensichtlich von Stelle $i-1$ und $i-2$ ab
 - Datenabhängigkeit zwischen den Iterationen (*read-after-write*)
 - Die Iterationen $i-1$ und $i-2$ müssen vor i ausgeführt werden
 - Kein Problem wenn Schleife seriell ausgeführt wird
- Lässt sich die Schleife parallelisieren?

Einführung (II)

- So geht es nicht:

```
vec[0] = 0;  
vec[1] = 1;  
#pragma omp parallel for  
for (int i = 2; i < n; ++i)  
    vec[i] = vec[i-1] + vec[i-2];
```

- Warum nicht?
 - Angenommen t Threads teilen sich den Indexraum gleichmäßig auf
 - Thread k bearbeitet zusammenhängendes Teilstücke $i=a_k, \dots, a_{k+1}-1$
 - $a_0 = 2$ und $a_t = n$
 - Starten alle gleichzeitig, hat nur Thread 0 die benötigten 2 vorherigen Einträge
 - Threads $1, \dots, t-1$ verletzen *read-after-write* Abhängigkeit (race condition)

Einführung (III)

- Lösung:
 - Man müsste die Einträge a_{k-1} und a_{k-2} vorab initialisieren
 - Im Allgemeinen schwierig
 - Hier ginge es expliziter Formel für Fibonacci-Zahlen von Moivre-Binet

$$vec[i] = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right]$$

- Aufteilung muss vorab bekannt sein (manuelles Aufteilen erforderlich)

```
int thd_num = omp_get_num_threads();
int from = 2;
int to = n;
for (int k = 0; k <= thd_num; ++k)
    a[k] = from + (to - from) * thd_num / t;
```

Arten der Datenabhängigkeit

- Direkte (Fluss-)Abhängigkeit:
 - read-after-write (2 nach 1, 3 nach 2)

```
a = 7;           // 1
b = a + 1;       // 2
c = b;           // 3
```

- Indirekte oder Gegenabhängigkeit:
 - write-after-read (5 nach 4)

```
b = a + 1;       // 4
a = 3;           // 5
```

- Ausgabeabhängigkeit:
 - write-after-write (7 nach 6)

```
a = 10;          // 6
a = 20;          // 7
```

Parallelisierung

- Ausführungsreihenfolge von datenabhängigen Instruktionen darf nicht verändert werden
 - In seriellen Programmen immer gegeben
 - In parallelen Programmen Aufgabe des Programmierers
- Nur unabhängige Instruktionen können vertauscht werden
 - Parallel ausgeführte Instruktionen müssen also unabhängig sein
 - Suche nach unabhängigen Variablen, Berechnungen, Teilproblemen
 - Parallelisiere diese
- Datenabhängigkeit wird erkannt und benutzt von
 - Prozessoren mit Pipelining
 - Compilern beim Optimieren von Code
 - Parallelisierenden Compilern

Auflösen von Datenabhängigkeiten

- Direkte (Fluss-)Abhängigkeit:

- Lässt sich nicht auflösen

```
a = 7;           // 1
b = a + 1;      // 2
c = b;          // 3
```

- Indirekte oder Gegenabhängigkeit:

- Umbenennen

```
b = a + 1;      // 4
a2 = 3;        // 5
```

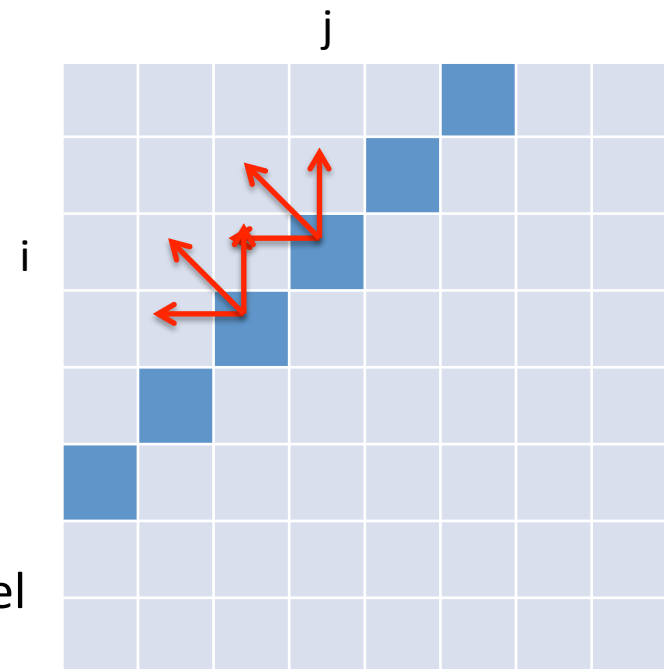
- Ausgabeabhängigkeit:

- Umbenennen

```
a = 10;         // 6
a2 = 20;       // 7
```

Abhängigkeiten in DP-Alignments

- Beispiel: Needleman-Wunsch Alignment
 - $M_{i,j}$ ist **flussabhängig** von $M_{i-1,j-1}$, $M_{i,j-1}$ und $M_{i-1,j}$
 - Keine Abhängigkeit zwischen $M_{i,j}$ und $M_{i+1,j-1}$
 - Alle Elemente auf einer Antidiagonalen sind voneinander **unabhängig**
- Mögliche Parallelisierung Needleman-Wunsch:
 - Berechne Matrix antidiagonalweise
 - Antidiagonale selbst kann mit **parallel for** berechnet werden
- Alternativ:
 - Jeder Thread berechnet ein Tile (siehe cache-aware DP-Alignment *P-VL3*)
 - Berechne diagonal benachbarte Tiles parallel



Reduktion

- Szenario:
 - Parallel arbeitende Threads teilen sich manchmal eine Resource
 - Um Race Conditions zu vermeiden muss synchronisiert werden
 - `#pragma omp critical`
 - `#pragma omp atomic`
 - critical Bereiche können zum Flaschenhals werden, weil sich davor Threads stauen
- Beispiel (P-A4):
 - Ein Vector zum Speichern aller Vorkommen eines Pattern im Text
 - Einfügen (`push_back`) wird synchronisiert durch `omp critical`

Reduktion (II)

- Alternative (ohne Synchronisation):
 - Resource wird dupliziert, jeder Thread erhält eigene Resource
 - Am Ende des parallelen Bereichs werden lokale Ressourcen zu einer globalen vereint (Reduktion)
 - Meist effizienter, weil Threads nicht warten müssen

- Beispiel (P-A4):
 - Jeder Thread erhält eigenen Vector zum Speichern der Vorkommen
 - Master-Thread fügt Vorkommen jedes Threads zum globalen Vector hinzu

Reduktion (III)

- Reduktion muss manuell implementiert werden
- Bei einfachen Reduktionen hilft OpenMP:
 - Direktive **reduction (op: var)** in einem parallelen Bereich

```
int summe = 0;

#pragma omp parallel for reduction (+:summe)
for (int i = 0; i < 10000; ++i)
    summe += A[i];
```

- Funktionsweise
 - Jeder Thread erhält lokale Kopien der Variable **var**
 - **op** ist Operation, die am Ende mit allen lokalen Variablen ausgeführt wird
 - Ergebnis wird mit der ursprüngliche, globale Variablen **var** verknüpft

Reduktionsoperationen

- In OpenMP (C++) gibt es folgende Reduktionsoperationen:

| Operator | Initialwert der lok. Variablen |
|----------|--------------------------------|
| + | 0 |
| - | 0 |
| * | 1 |
| & | ~0 |
| | 0 |
| ^ | 0 |
| && | true |
| | false |

- Jede lokale Variable erhält zu Beginn der Reduktion den entsprechenden Initialwert

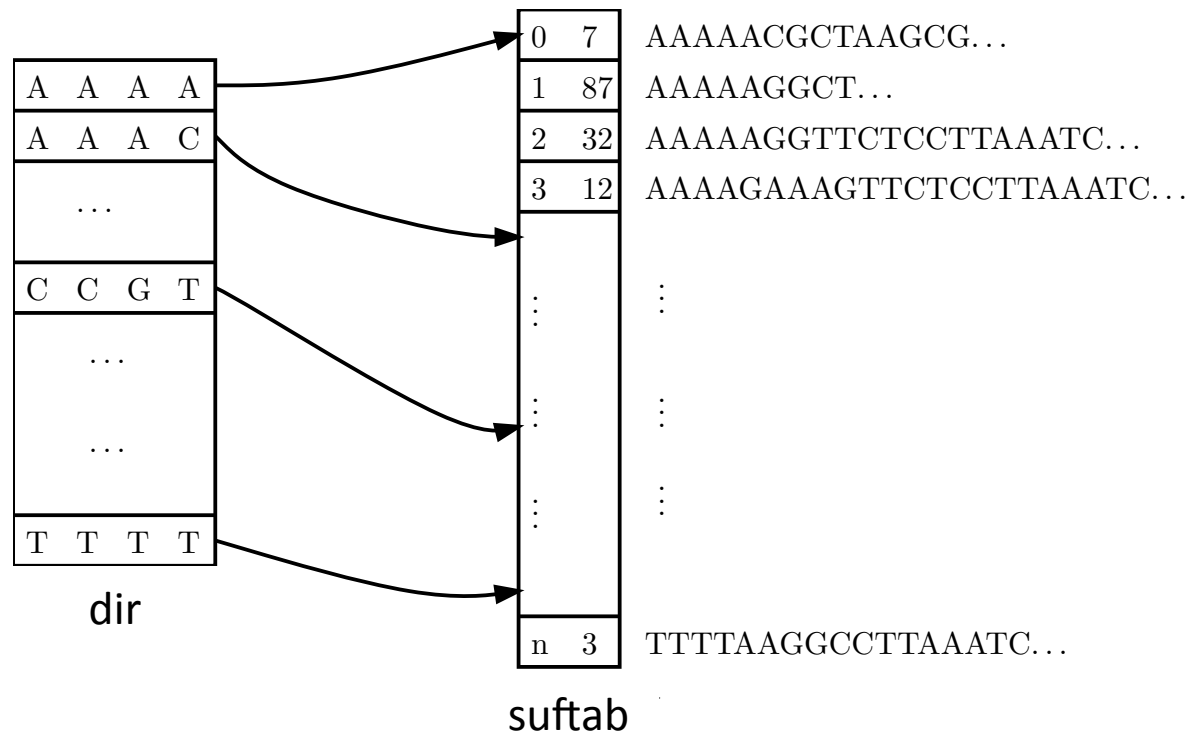
Q-GRAM INDEX

Begriffe

- q-gram
 - Kurzer String der Länge q
- q-gram Index
 - Speichert und liefert effizient alle Vorkommen eines gegebenen q-grams in einem Text

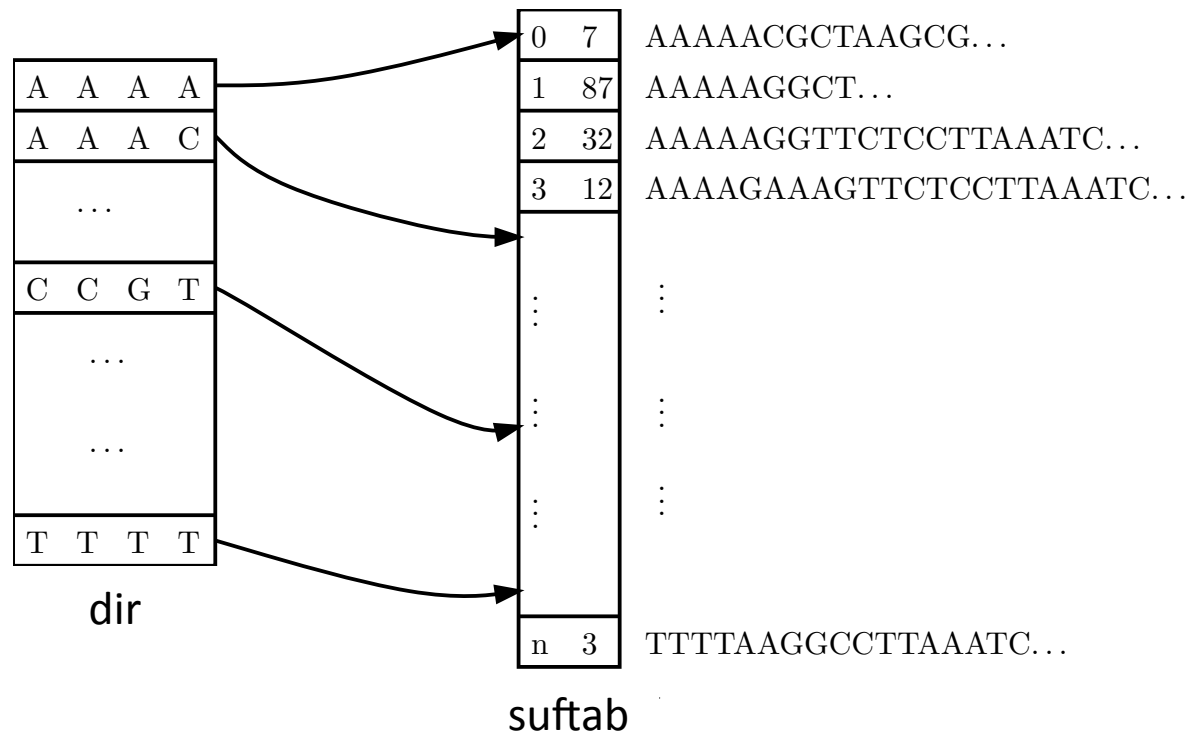
q-gram Index

- Kann über ein Suffix-Array implementiert werden
 - **suftab** speichert die Anfänge aller Suffixe in lexikographischer Ordnung
 - **dir** speichert zu jedem q-gram Q die Position des ersten Suffix in **suftab**, das mit Q beginnt



q-gram Index (II)

- Abfrage aller Vorkommen eines q-grams Q im Text
 - Ermittle Position $h(Q)$ des zu Q gehörenden Eintrags in **dir** (mittels q-gram hash)
 - $dir[h(Q)]$ und $dir[h(Q)+1]$ sind Anfang und Ende (ausschl.) des Intervalls (bucket) von Textpositionen in **suftab**



q-gram Hashing

- Wie findet man effizient den zu Q gehörenden Eintrag in **dir**
 - **dir** enthält alle möglichen q-gramme in lex. Ordnung
 - man braucht also eine bijektive Funktion h , die die Menge aller q-gramme auf Positionen in **dir** abbildet:

$$h : \Sigma^q \rightarrow [0, |\Sigma|^q - 1]_{\mathbb{N}}$$

- h muss die Ordnung der q-gramme erhalten:

$$Q_1 <_{lex} Q_2 \Rightarrow h(Q_1) < h(Q_2)$$

- $h(Q)$ heißt Hashwert oder Rang von Q

q-gram Hashing (II)

- h ist durch Bedingungen eindeutig bestimmt und lässt sich wie folgt berechnen:
 - Weise jedem Buchstaben x aus Σ seinen Rang $\text{ord}(x)$ zu
 - Beispiel $\Sigma_{\text{DNA}}=\{A,C,G,T\}$:
 - $\text{ord}(A)=0, \text{ord}(C)=1, \text{ord}(G)=2, \text{ord}(T)=3$
 - Für q-gram Q gilt dann:

$$h(Q) = \sum_{i=0 \dots q-1} \text{ord}(Q[i]) \cdot |\Sigma|^{q-1-i}$$

- Beispiel Σ_{DNA} :

$$\begin{aligned} h(GATTACA) &= 2033010_4 \\ &= 2 \cdot 4^6 + 3 \cdot 4^4 + 3 \cdot 4^3 + 1 \cdot 4^1 \\ &= 9156 \end{aligned}$$

q-gram Hashing (III)

- Hashwerte aller 2-gramme über Σ_{DNA} :

| Q | | H(Q) |
|---|---|------|
| A | A | 0 |
| A | C | 1 |
| A | T | 2 |
| A | G | 3 |
| C | A | 4 |
| C | C | 5 |
| C | G | 6 |
| C | T | 7 |
| G | A | 8 |
| G | C | 9 |
| G | G | 10 |
| G | T | 11 |
| T | A | 12 |
| T | C | 13 |
| T | G | 14 |
| T | T | 15 |

Überlappendes q-gram Hashing

- Rang von sich überlappenden q-grammen wird benötigt
 - Aufbau des q-gram Index
 - Während des Scans einer Sequenz

- Gegeben: Q_0 , Q_1 und $h(Q_0)$
 - Es gilt $Q_0[1..q-1] = Q_1[0..q-2]$
 - Beispiel: $Q_0 = \mathbf{GATT}$, $Q_1 = \mathbf{ATTA}$

- Lässt sich $h(Q_1)$ in konstanter Zeit berechnen?
 - Ja, durch anpassen des Rangs:

$$h(Q_1) = h(Q_0) \cdot |\Sigma| - \text{ord}(Q_0[0]) \cdot |\Sigma|^q + \text{ord}(Q_1[q-1])$$

q-gram Index Erzeugen

- Anmerkungen zum q-gram Index mit Suffix Array
 - Im Suffix Array sind alle Suffixe vollständig lex. Sortiert
 - Eigentlich genügt die Sortierung nach den ersten q Zeichen in **suftab**
 - Sortierung der ersten q Zeichen ist um Größenordnungen schneller als Konstruktion des Suffix Array
- Wie sortiert man effizient alle Suffixe nach den ersten q Zeichen?
 - Quick Sort [P-A2]
 - **suftab** mit Werten $0, \dots, n-q$ initialisieren
 - **suftab** mit `std::sort` und eigenem Vergleichsfunktor sortieren, der höchstens die ersten q Zeichen der entsprechenden Suffixe vergleicht
 - Schreibe den Bucket-Anfang jedes q-grams Q an die Stelle $h(Q)$ in **dir** (linearer Scan über **suftab** und **dir**)
 - Counting Sort

Counting Sort

- Sortiert Elemente nicht über Vergleiche sondern über ihren Rang $h(Q)$
 - Laufzeit ist $O(n + |\Sigma|^q)$ statt $O(n \cdot \log n)$
 - Funktioniert nur für kleine Alphabete (DNA) und kleine q ($q < 14$)
- Funktionsweise:
 1. Fülle **dir** mit Nullen
 2. Scanne Text und zähle q -gramme mittels **dir**
 3. Berechne kumulative Summe der Zähler in **dir**, so dass der Eintrag an Stelle $h(Q)$ das Bucket-Ende zum q -gram Q ist
 4. Scanne Text und verringere Eintrag an Stelle $h(Q)$ in **dir**, um Bucket in **suftab** von hinten nach vorne mit Textposition zu füllen

Counting Sort (II)

- Pseudo Code:

```
for j=0 to  $|\Sigma|^q$  // 1. Zähler auf Null setzen
    dir[j] = 0

for i=0 to n-q // 2. q-gramme zählen
    j = h(T[i..i+q-1])
    dir[j] = dir[j] + 1

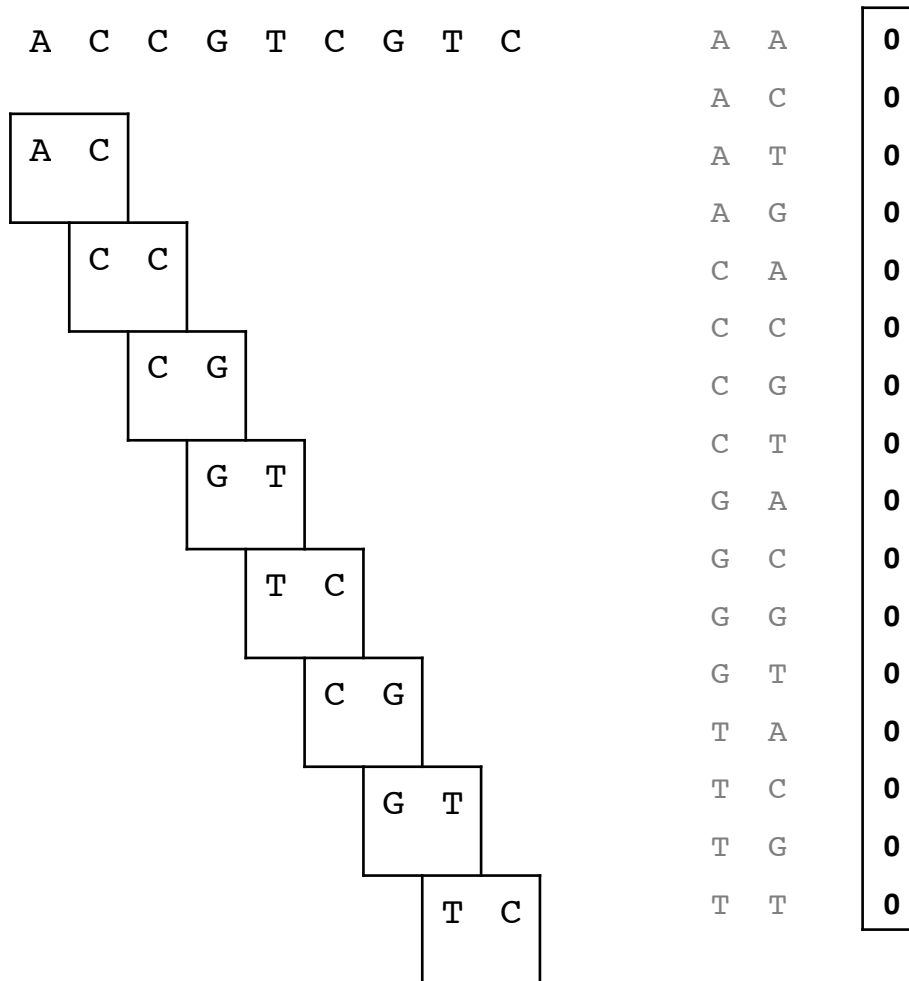
for j=1 to  $|\Sigma|^q$  // 3. Kumulative Summe bilden
    dir[j] = dir[j] + dir[j-1]

for i=0 to n-q // 4. Vorkommen einsortieren
    j = h(T[i..i+q-1])
    dir[j] = dir[j] - 1
    suftab[dir[j]] = i
```

- Siehe auch: <http://de.wikipedia.org/wiki/Countingsort>

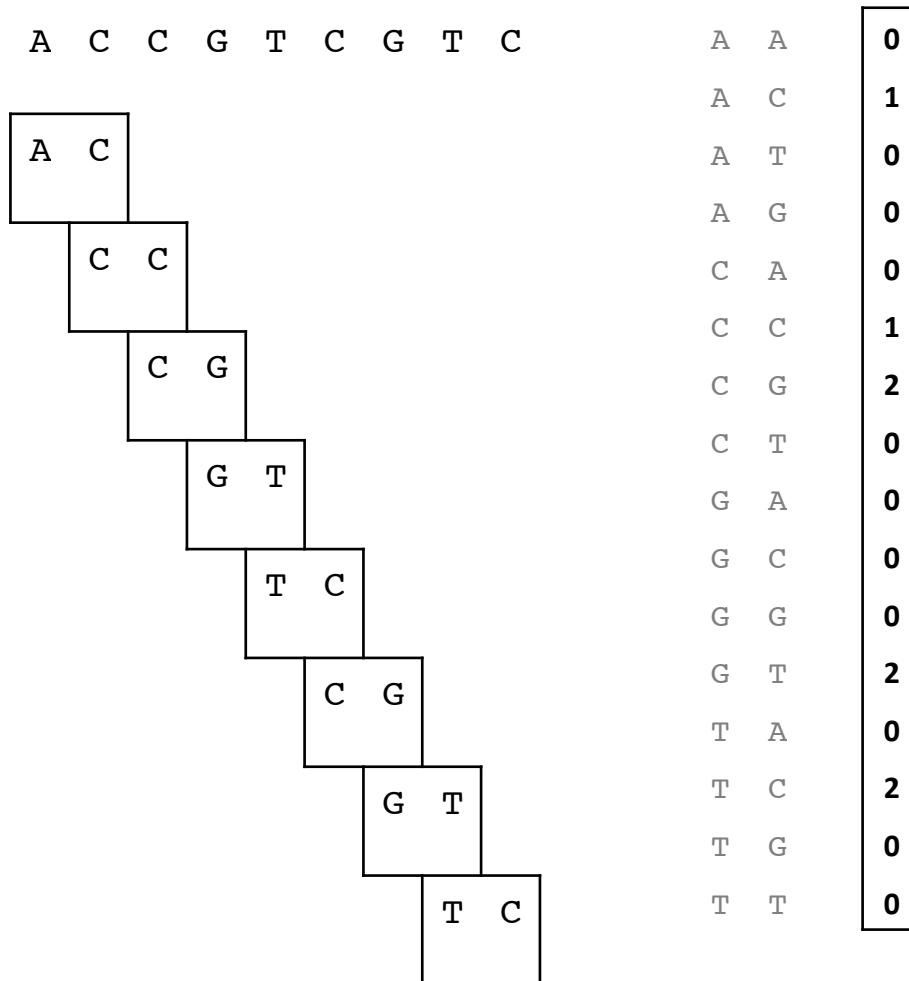
Beispiel (Schritt 1)

dir nach Initialisierung



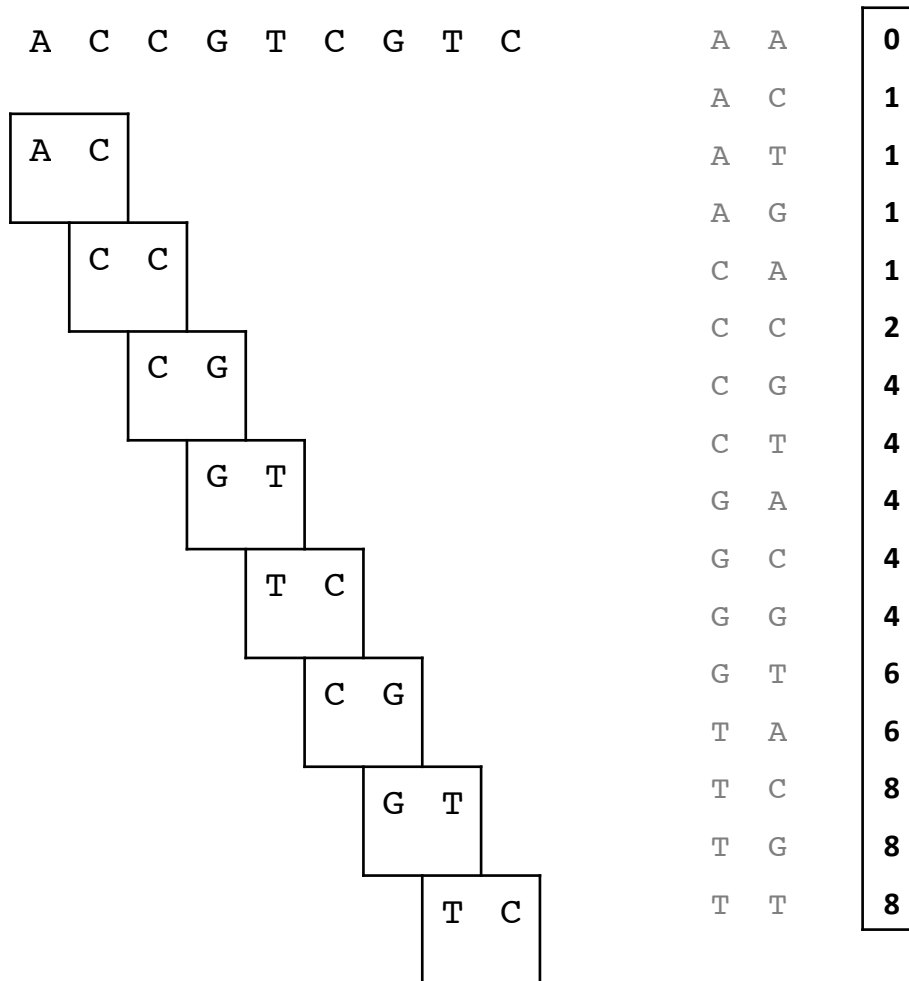
Beispiel (Schritt 2)

dir nach Zählung

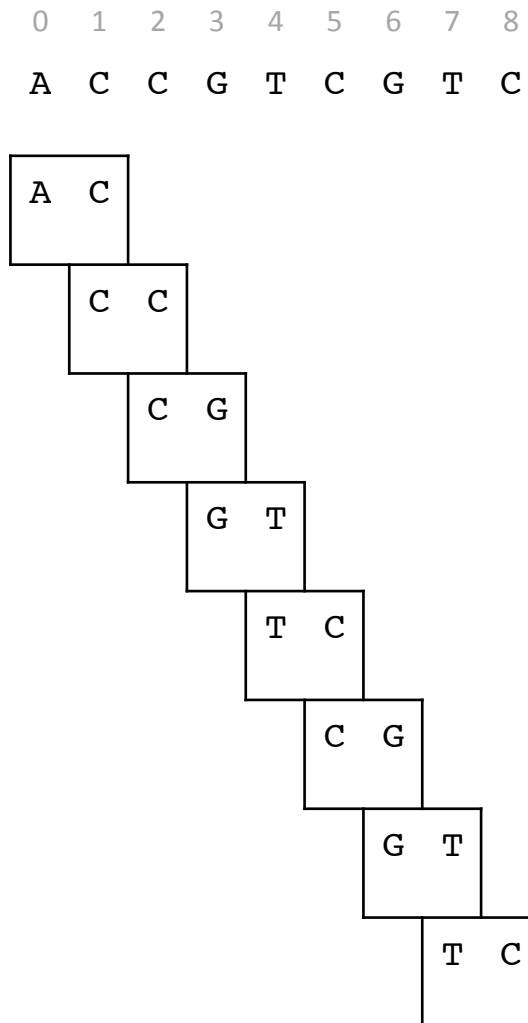


Beispiel (Schritt 3)

kumulative Summe



Beispiel (Schritt 4)



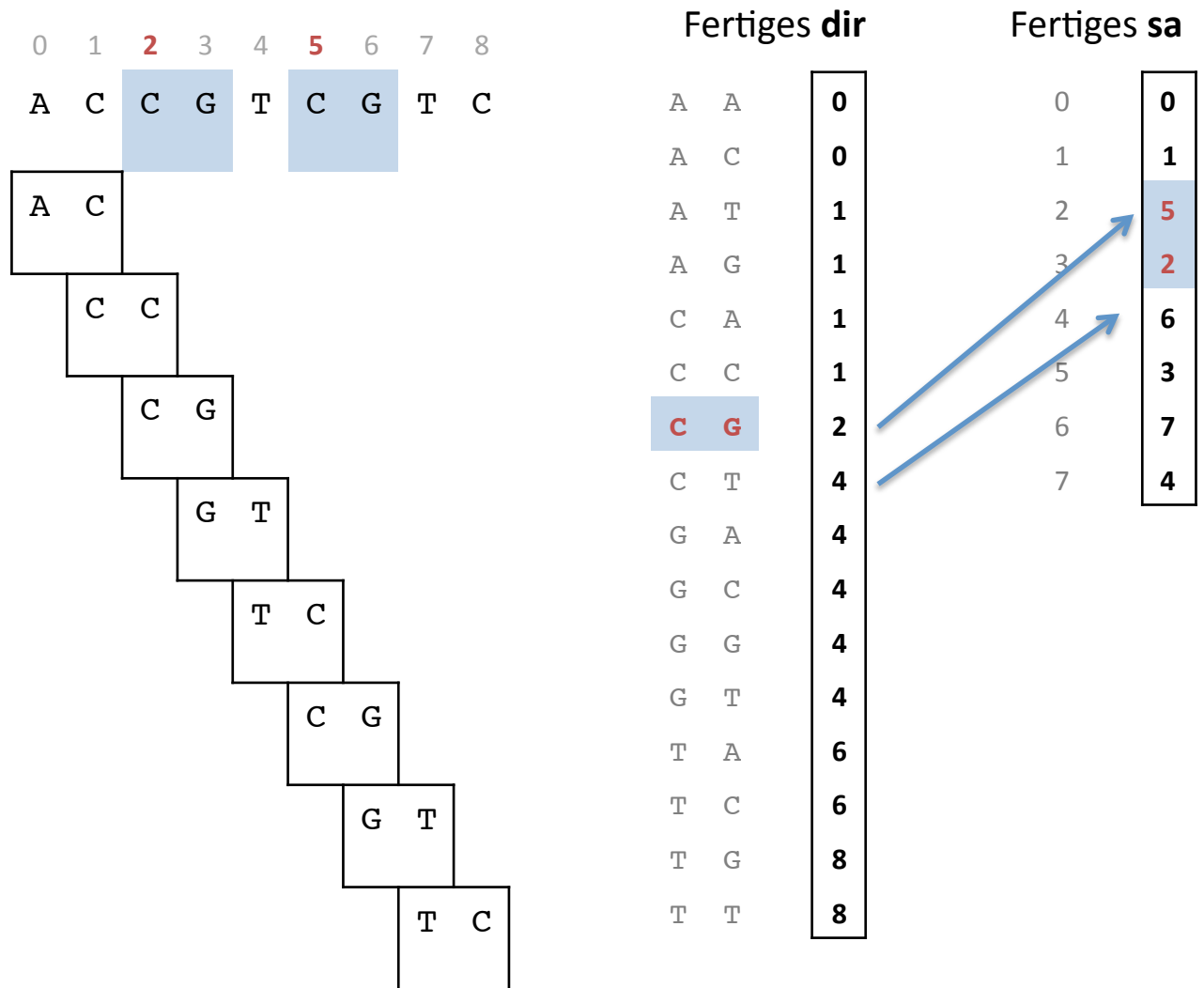
Fertiges **dir**

| | | |
|---|---|---|
| A | A | 0 |
| A | C | 0 |
| A | T | 1 |
| A | G | 1 |
| C | A | 1 |
| C | C | 1 |
| C | G | 2 |
| C | T | 4 |
| G | A | 4 |
| G | C | 4 |
| G | G | 4 |
| G | T | 4 |
| T | A | 6 |
| T | C | 6 |
| T | G | 8 |
| T | T | 8 |

Fertiges **sa**

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 5 |
| 3 | 2 |
| 4 | 6 |
| 5 | 3 |
| 6 | 7 |
| 7 | 4 |

q-gramme Finden



Paralleles Counting Sort

- Paralleles Counting Sort mit k Threads:
 1. Zerlege Text in k Teile (Überlappung in $q-1$ Zeichen) und erzeuge k Felder dir_1, \dots, dir_k
 2. Ordne Teile den k Threads zu. Für jeden Thread t :
 1. Fülle dir_t mit Nullen auf
 2. Zähle q -gramme in eigenem Teil
 3. Berechne aus lokalen Zählern gemeinsame kumulative Summe in den dir_t
 4. Für jeden Thread t :
 1. Scanne Teiltext und verringere Eintrag an Stelle $h(Q)$ in dir_t , um Teilbucket in **suftab** von hinten nach vorne mit Textposition zu füllen
 5. dir_1 wird endgültiges **dir**. Gib dir_2, \dots, dir_k frei.
- Siehe auch: <http://snippets.dzone.com/tag/countingsort>

BEMERKUNGEN ZUR P-AUFGABE

Bemerkungen zu Aufgabe 4

- Typische Fehler:
 - Der Text wurde in nichtüberlappende Teile zerlegt
 - `vector.push_back()` ohne Kritischer Abschnitt
 - Ausgabeformat nicht beachtet

- Unnötig:

```
#pragma omp parallel for num_threads(t)
for (int k = 0; k < t; k++)
{ int begin = ...; int end = ...; ...; }
```

- Besser so:

```
#pragma omp parallel num_threads(t)
{ int begin = ...; int end = ...; ...; }
```

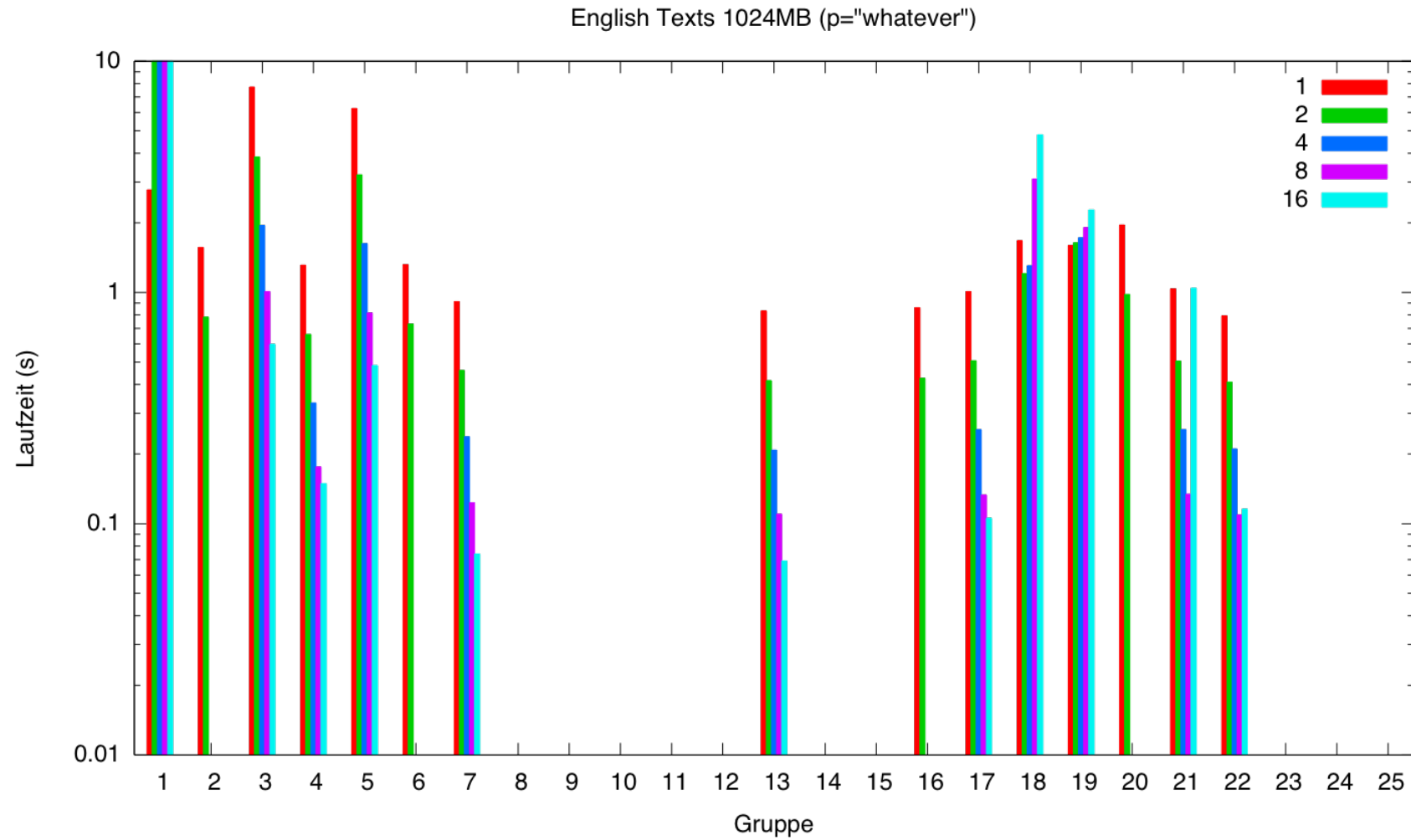
Bemerkungen zu Aufgabe 4

- Verbesserungen:
 - Als String kodierte Zahl in int konvertieren:

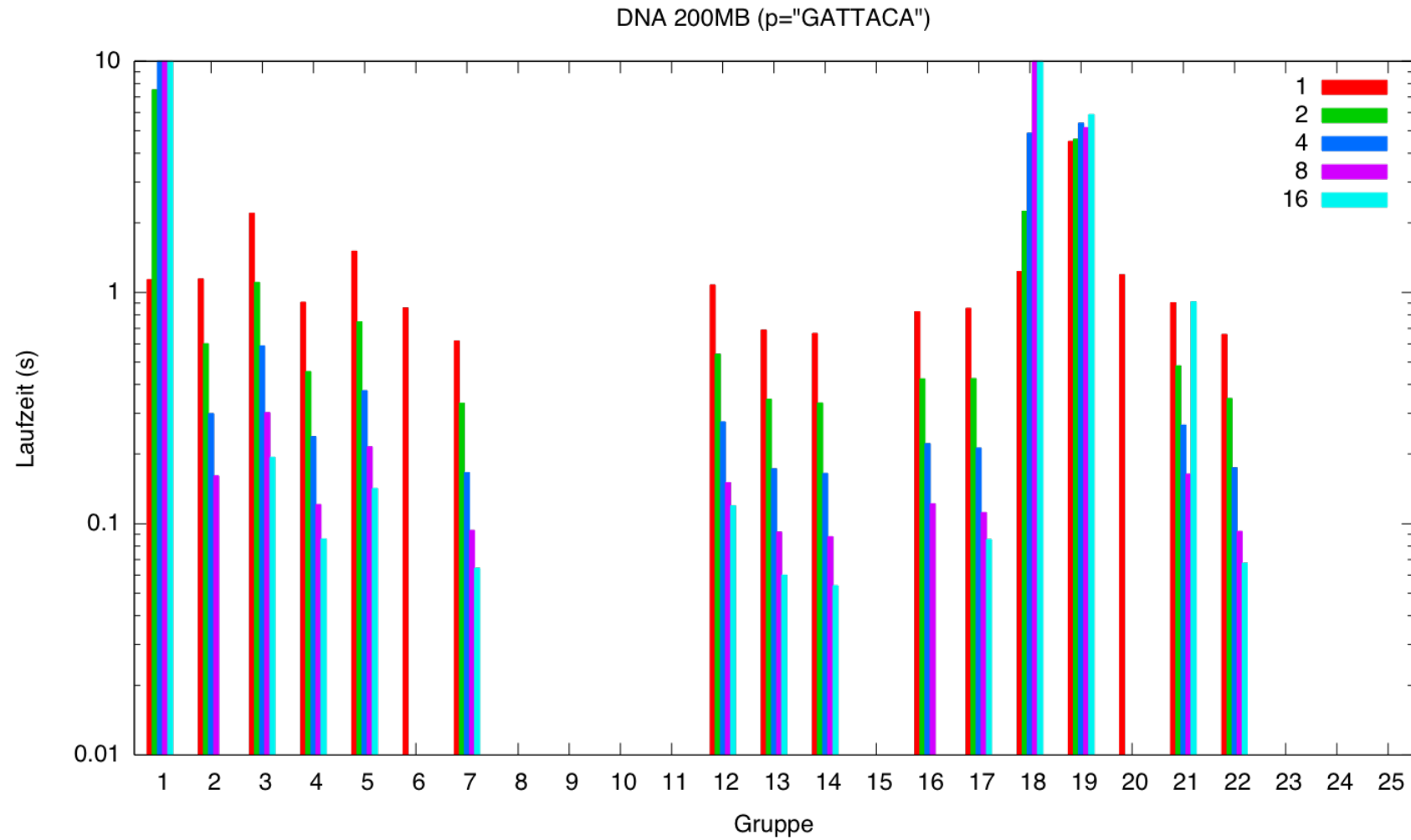
```
const char *s = "73";  
int i;  
  
std::istringstream strm(s);    // #include <sstream>  
strm >> i;                    // in C++  
  
i = atoi(str);                // in C
```

- <http://www.lmgf.com/?q=convert+string+to+int+c%2B%2B>

Laufzeiten | english.1024MB



Laufzeiten | dna.200MB



Hinweise zu Aufgabe 5

- Schubfachprinzip anwenden
 - Jede Read in $k+1$ Teilstücke teilen
 - Jeden Teilstücken mit einem q -gram Index suchen
 - Jeden Hit durch den naive Hamming Distance Algorithmus prüfen
- Duplikate durch Filterung vermeiden
- Parallelisieren nach Reads ist am einfachsten und wahrscheinlich effizientesten
- Beispiel
 - $R = \text{CATTTC}$, $k = 1$ Mismatches

Hinweise zu Aufgabe 5

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | | A | A | G | A | T | A | C | A | T | T | T | T | A | A | A | A | A | A | C | A | A | T | T | T |
| 0 | C | | | | | | | C | | | | | | | | | | | | | | | | | |
| 1 | A | | | | | | | | A | | | | | | | | | | | | | | | | |
| 2 | T | | | | | | | | | T | | | | | | | | | | | | | | | |
| 3 | T | | | | | | | | | T | T | | | | | | | | | | | | T | | |
| 4 | T | | | | | | | | | | T | T | | | | | | | | | | | | T | |
| 5 | T | | | | | | | | | | | T | T | | | | | | | | | | | | T |

| 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|----|----|
| C | A | T | T | T | T |
| C | A | T | T | T | T |

| 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|----|
| A | C | A | T | T | T |
| C | A | T | T | T | T |

| 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|----|----|
| C | A | T | T | T | T |
| C | A | T | T | T | T |

| 18 | 19 | 20 | 21 | 22 | 23 |
|----|----|----|----|----|----|
| C | A | A | T | T | T |
| C | A | T | T | T | T |