Kurs 01727 Parallele Programmierung und Grid Computing Einsendearbeit 2

Aufgabe 2-1 (10 Punkte)

Wir wollen die Arbeitsschritte in einer Wäscherei parallelisieren. Die Reinigung eines Kleidungsstücks besteht dabei aus folgenden Schritten [1]

1. Warenschau: Das Kleidungsstück wird eingehend inspiziert und für die anschließende Reinigungsbehandlung vorbereitet
2. Vordetachur: Dieser Arbeitsschritt der professionellen Fleckenentfernung vor der eigentlichen Reinigungsbehandlung im Lösemittelbad ist oftmals nötig, um Verschmutzungen, die Wasser enthielten als sie sich auf dem Kleidungsstück festsetzten, vorzuweichen und anschließend zu entfernen.
3. Reinigungsprozess: Die Textilien werden in der Trommel der Reinigungsmaschine in einem Reinigungsbad bewegt, das je nach Pflegekennzeichnung und Verschmutzung aus einem Lösemittel oder Wasser besteht.
4. Nachdetachur: Manche Flecken zeigen sich überhaupt erst nach einer Lösemittelbehandlung. Diese werden dann mittels der sogenannten Nachdetachur endgültig beseitigt.
5. Finishen/Bügeln: Das gereinigte Kleidungsstück wird so in Form gebracht, dass es dem Kunden beim Abholen schrank- bzw. anziehfertig ausgehändigt werden kann.

Aufgaben:

a) Nehmen wir im ersten Schritt an, dass alle Schritte etwa gleich lang (7 Minuten) benötigen. Es stehen fünf Personen zur Verfügung, welche die Reinigung durchführen können. Berechnen Sie, wie lange diese 5 Personen benötigen, um 100 Kleidungsstücke zu reinigen, wenn Sie Pipelining/Temporäre Parallelität verwenden. Danach berechnen Sie das Ergebnis wenn Datenparallelität mit statischer Verteilung verwendet wird. Um die Kleidungsstücke bei der Datenparallelität auf die Personen zu verteilen, werden 10 Minuten benötigt.

b) Wir gehen im nächsten Schritt davon aus, dass die einzelnen oben beschrieben Schritte unterschiedlich lange benötigen, die Kleidungsstücke aber jeweils gleichlange behandelt werden, siehe Tabelle 1. Wir unterscheiden hier zwei Fälle, A und B. Wie lange dauert nun die Ausführungszeit mit den oben verwendeten Techniken zur Datenaufteilung, d.h. Pipelining/Temporäre Parallelität und Datenparallelität für Fall A und Fall B?

c) Nicht alle Kleidungsstücke brauchen die gleiche Behandlung. Gehen wir jetzt davon aus, dass bei etwa einem drittel der Kleidung die Vordetachur nicht notwendig ist. Bei der Hälfte der Kleidungsstücke ist dagegen die Nachdetachur nicht notwendig. Die Waschzeiten schwanken auch zwischen 8 und 12 Minuten. Beschreiben Sie, welche Probleme sich dadurch für die bisher verwendeten Methoden zur Parallelisierung ergeben und wie man diese lösen kann.

d) Wir wollen uns im Folgenden noch einmal die statische und die dynamische Verteilung der Kleidungsstücke bei einer „Datenparallelität“ betrachten. Wir betrachten dazu die Situation aus der Teilaufgabe b), in der zwischen Fall A und B unterschieden wird. Dabei werden die Kleidungsstücke in zwei möglichen Reihenfolgen verteilt, die in Abbildung 1 dargestellt sind. Dabei ist die gesamte

[1] Quelle: `https://www.textilpflegebern.ch/ueber-uns/unsere-arbeit/einzelne-arbeitsschritte`

Schritt	Fall A	Fall B
Warenschau:	5,0	5,0
Vordetachur:	7,0	4,0
Reinigung:	10,0	7,0
Nachdetachur:	7,0	4,0
Finish:	5,0	5,0

Tabelle 1
Zeitbedarf der verschiedenen Schritte

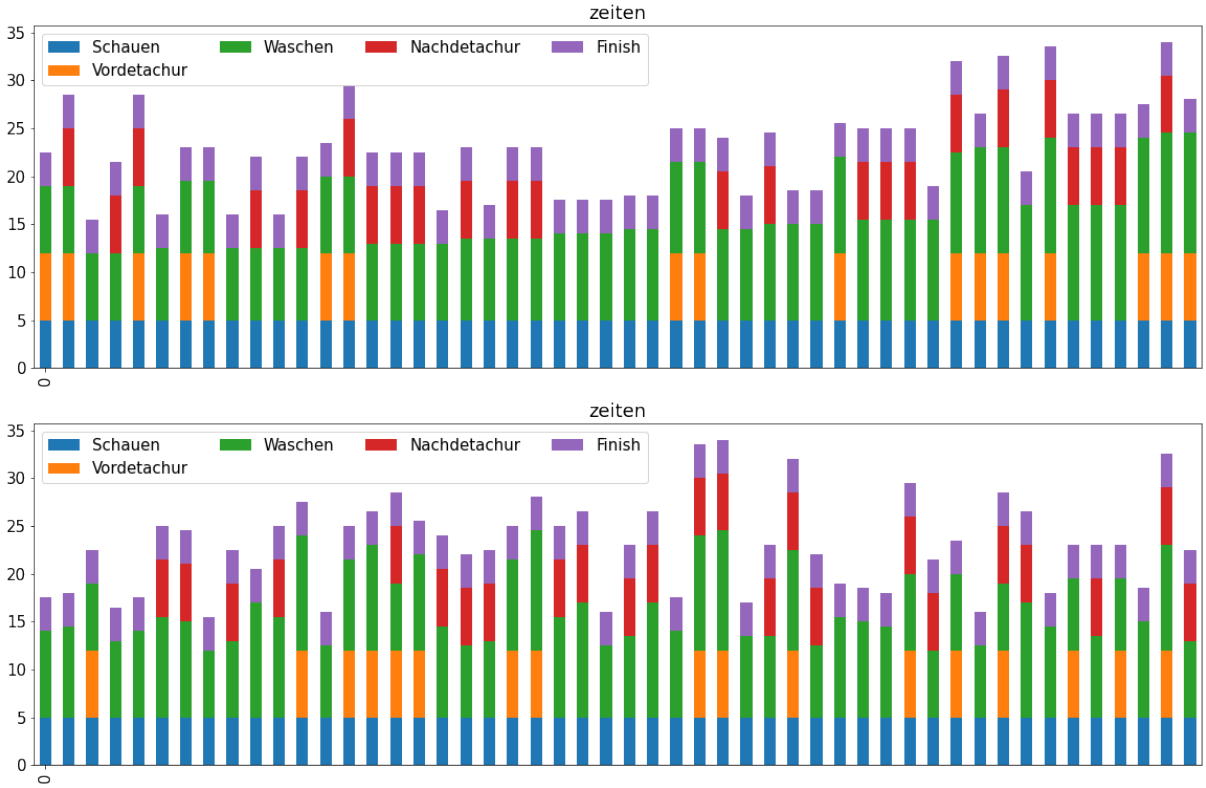

Abbildung 1
Zwei mögliche Verteilungen der Kleidungsstücke

Menge der Aufträge gleich, nur die Reihenfolge, in der die Aufträge „Auf einen Stapel“ kommen, ist unterschiedlich. Im ersten Fall wurden die Kleidungsstücke nach ihrer Waschdauer sortiert, im zweiten Fall ist die Reihenfolge zufällig. Für welchen Fall wird eine statische Verteilung der Daten wahrscheinlich besser funktionieren? Warum? Wie sieht es bei der dynamischen Verteilung aus?

Für die letzten Aufgaben c) und d) finden Sie unter den Materialien auf Hopper und in Moodle das Jupyter Notebook `Scheduling.ipynb`, in denen Sie sich die statischen und die dynamischen Schedules ansehen können. Sie können dort auch einige Parameter verändern.

Musterlösung

a) Für das Pipelining/ temporäre Parallelität gilt: $t(k) = p\frac{k+n-1}{k}$ also $t(100) = (5 \cdot 7min) \cdot \frac{5+100-1}{5} = 728min$
Für die Datenparallelität gilt: $t(k) = k \cdot q + \frac{n \cdot p}{k} = 10min + \frac{100 \cdot 35min}{5} = 710min$ (da die Aufgabenstellung nicht klar war, gilt natürlich auch: $10min \cdot 5 + \frac{100 \cdot 35min}{5} = 750min$

b) Für das Pipelining gilt: Ein Wäschestück braucht im Fall A $(5 + 7 + 10 + 7 + 5)min = 34min$ Das erste Kleidungsstück ist also nach 34 Minuten fertig, danach wird alle 10 Minuten ein neues Kleidungsstück fertig (hier bestimmt die längste Stufe den Takt).

Also gilt für A: $34min + 99 \cdot 10min = 1024min$.
Für B gilt analog: $(5 + 4 + 7 + 4 + 5)min + 7min \cdot 99 = 718min$.
Für die Datenparallelität gilt: Fall A : $s(5) = 10min + \frac{(5+7+10+7+5)min \cdot 100}{5} = 690min$
und für Fall B: $s(5) = 10min + \frac{(5+4+7+4+5)min \cdot 100}{5} = 510min$

c) Einen Nachteil für die temporäre Parallelität haben wir bereits in der vorherigen Teilaufgabe gesehen: Dadurch, dass die Aufträge unterschiedlich lange dauern, kommt es zu Idle-Zeiten bei einigen Arbeitern. Das wird hier verstärkt, man spricht hier auch von Blasen. Für die Datenparallelität kommt nun das Problem dazu, dass die einzelnen Wäschestücke unterschiedlich lange bearbeitet werden müssen. Daher kann es passieren, dass auch hier die Arbeitskräfte unterschiedlich stark ausgelastet sind, da einer z.B. nur Wäschestücke bekommt, die keine Vordachtur benötigen.

   Lösen kann man dies durch ein dynamisches Scheduling, welches die Aufträge (also Kleidung) dynamisch verteilt, hier ergibt sich jedoch der Overhead des Schedulings

d) Für das statische Scheduling wird sehr wahrscheinlich die nicht-sortierte Verteilung am besten sein. Durch eine Zufallsverteilung ist es wahrscheinlicher, dass sich die Arbeitszeiten der einzelnen Aufträge insgesamt ausgleichen. Sortiert man die Aufträge zunächst und macht eine naive Aufteilung, in welcher der erste Arbeiter z.B. die ersten n/p Aufträge bekommt, ist es wahrscheinlich, dass dieser Arbeiter dann auch die bekommt, die schnell beendet sind – und der letzte nur die, die am längsten dauern. Natürlich könnte man hier eine bessere Verteilung nehmen, in dem man z.B. in einem Round-Robin-Verfahren die Aufträge nacheinander an die Arbeiter verteilt. Dann würde man eine gleichmäßigere Verteilung erhalten. Allerdings wäre dann der Verteilungsoverhead größer. Das dynamische Verfahren sollte von der Reihenfolge der Aufträge unabhängig sein.

Aufgabe 2-2 (10 Punkte)

Abbildung 2
Anwendungsbeispiel

In dieser Aufgabe wollen wir ein Beispiel für Task-Parallelismus betrachten. Dafür wollen wir, passend zur Jahreszeit, eine Laterne basteln, wie in Abbildung 2 dargestellt, und diesen Vorgang parallelisieren.[2]
Die einzelnen Arbeitsschritte sind:

1. Schablone vorne ausschneiden ✂
2. Schablone hinten ausschneiden ✂
3. Zeichnen des vorderen Igelteils auf Tonpapier
4. Zeichnen des hinteren Igelteils auf Tonpapier
5. Ausschneiden des vorderen Igelteils ✂
6. Ausschneiden des hinteren Igelteils ✂

[2]Für die richtige Bauanleitung können Sie z.B. hier schauen: `https://www.kinderspiele-welt.de/sankt-martin/laternen-basteln.html`

7. Ausschneiden von gelbem Transparentpapier für den Laternenausschnitt vorne und hinten ✂
8. Kleben des Transparentpapiers vorne
9. Kleben des Transparentpapiers hinten
10. Ausreißen von orangen und braunem Transparentpapier für die Stacheln
11. Aufkleben der Stacheln vorne
12. Aufkleben der Stacheln hinten
13. Augen zeichnen und ausschneiden auf weißem Papier ✂
14. Zeichnen und ausschneiden der Pupillen und der Nase auf schwarzen Karton ✂
15. Kleben von Augen (mit Pupillen) und Nase auf vorderem Igel
16. Kleben von Augen (mit Pupillen) und Nase auf hinterem Igel
17. Befestigen der Igelteile an Käseschachtel

.

Hinweis: Es gibt nur einen Bogen schwarzes und weißes Tonpapier, so wie nur einen Bogen pro Farbe Transparentpapier. Es sind jedoch ausreichend Scheren, Stifte und Kleber vorhanden. Wir gehen außerdem davon aus, dass die Augen erst aufgeklebt werden können, wenn die Stacheln aufgeklebt sind.

a) Zeichnen Sie einen Task-Graphen für diese Bauanleitung des Igels

b) Wie viele Leute können maximal parallel an der Fertigstellung der Laterne gleichzeitig arbeiten?

c) Wie verändert sich der Graph, wenn es nur eine Schere gibt? Dabei können Sie davon ausgehen, dass immer der Task mit der niedrigeren Nummer die Ressource erhält. Alle Tasks, die eine Schere benötigen, haben das ✂-Symbol.

Musterlösung

a) Der Graph ist in Abbildung 3 dargestellt.

b) Maximal 6 können parallel arbeiten. Allerdings sind wahrscheinlich nicht mehr als drei sinnvoll.

c) Die Lösung ist in Abbildung 4 dargestellt. Dabei sind die neuen Kanten in Rot dargestellt. Durch diese neuen Kanten werden jedoch einige andere Kanten überflüssig, diese sind gestrichelt dargestellt.

Aufgabe 2-3 (10 Punkte)

Betrachten wir eine CRCW PRAM mit einem Kombinationsmodell für parallele Schreibzugriffe

a) Erläutern Sie, wie dieses Modell definiert ist und wie es sich von anderen PRAM-Modellen unterscheidet

b) Entwerfen Sie einen Algorithmus, der mit n^2 Prozessoren eine Liste mit n Elementen auf dieser Maschine sortiert. Nutzen Sie dabei die besondere Eigenschaft des kombinierten CRCW PRAMs aus. Die Idee des Algorithmus ist es, dass n^2 Vergleiche gleichzeitig ausgeführt werden und Sie den Index eines jeden Elements so leicht bestimmen können.

c) Was sind die Vor- und Nachteile dieses Algorithmus?

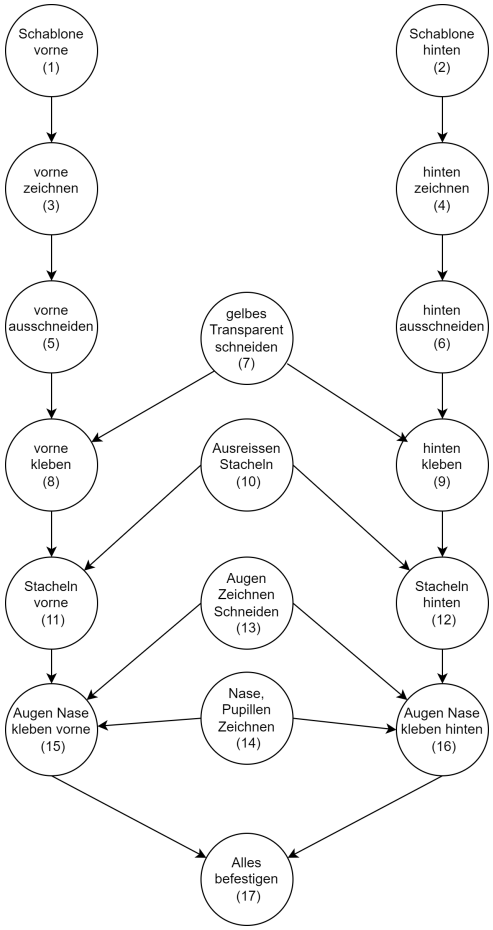

Abbildung 3
Lösung für 2-2 a)

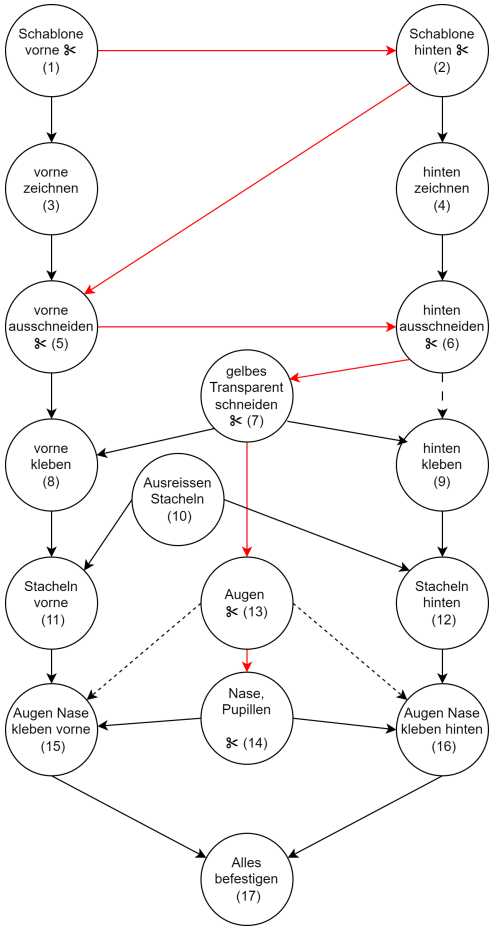

Abbildung 4
Lösung 2-2 c)

Musterlösung

- Der entscheidende Vorteil bei dem CRCW Modell mit Kombinationsmodell ist, dass es sowohl gleichzeitige Lese- als auch Schreibzugriffe erlaubt. Bei gleichzeitigen Schreibzugriffen werden die Werte kombiniert, wir gehen hier von einer Addition aus.
- Die Lösung ist in Algorithmus 1 dargestellt
- Der Vorteil ist, dass n^2 Prozessoren den Algorithmus in zwei Schritten parallel berechnen können. Der Nachteil ist, dass n^2 Vergleiche notwendig sind. Hier gibt es effizientere Algorithmen.

Algorithmus 1 Sortierung CRCW mit Kombinationsmodell

Input: Unsortiertes Array mit n Elementen $a[1..n]$
Output: Sortiertes Array $b[1..n]$

```
1: d[1...n]
2: for all 1 ≤ i, j ≤ n do
3:     if a[i] ≤ a[j] then
4:         d[j] ← 1
5:     end if
6: end for
7: for all 1 ≤ i ≤ n do
8:     b[d[i]] ← a[i]
9: end for
```

Aufgabe 2-4 (7 Punkte)

Sie wollen ein Cluster mit 256 Knoten entwerfen.

a) Vergleichen Sie die Gesamt-Bandbreite, die Bisektions-Bandbereite und die maximale Anzahl an benötigten Hops für einen Ring, ein 2-D Grid, einen 2-D Torus und einen n-Cube. Die direkte Bandbreite zwischen zwei Knoten sei dabei $B = 5\,GB/s$.

b) Wenn eine Nachricht zwischen zwei Hops $5\,ns$ braucht, wie lange braucht sie dann maximal in den unterschiedlichen Topologien?

c) Wie viele Verbindungen werden jeweils gebraucht?

Musterlösung

a)
- Für die Bandbreite:
  - Ring: $256 \cdot B = 1280\,GB/s$
  - 2D Grid: $2 \cdot \sqrt{256} \cdot (\sqrt{256} - 1) \cdot 5 = 2400\,GB/s$
  - 2-D Torus: $2 \cdot 256 \cdot B = 2560\,GB/s$
  - n-Cube: $\frac{B \cdot 256 \log_2 256}{2} = 5120\,GB/s$
- Bisektions-Bandbreite:
  - Ring: $2 \cdot B = 10\,GB/s$
  - 2-D Grid: $\sqrt{256} \cdot B = 80\,GB/s$
  - 2-D Torus: $2\sqrt{256} \cdot B = 160\,GB/s$
  - n-Cube: $B \cdot 256/2 = 640\,GB/s$
- Hops:
  - Ring: $N/2 = 128$
  - 2-D Grid: $2 \cdot (\sqrt{256} - 1) = 30$
  - 2-D Torus: $\sqrt{256} = 16$

– N-Cube: $\log_2 256 = 8$

b) Hier nehmen wir einfach die maximale Anzahl an Hops aus der letzten Aufgabe:

- Ring: $256/2 \cdot 5\,ns = 128 \cdot 5\,ns = 640\,ns$
- 2-D Grid: $2 \cdot (\sqrt{256} - 1) \cdot 5\,ns = 150\,ns$
- 2-D Torus: $(\sqrt{256}) \cdot 5ns = 70\,ns$
- N-Cube : $\log_2 256 \cdot 5\,ns = 40\,ns$

c)

- Ring: $N = 256$
- 2-D Grid: $2 \cdot (\sqrt{256} - 1) \cdot \sqrt{256} = 480$
- 2-D Torus: $n \cdot 2 = 512$
- N-Cube : $n/2 \cdot \log_2 n = 1024$

Alternative für Grid und Torus (wenn man ein Rechteck bildet):
Für das Grid:

- $z = 8$ Anzahl Zeilen
- $s = 16$ Anzahl Spalten
- Bandbreite: $(z(s-1) + s(z-1))B = 1160GB/s$
- Bisektions-Bandbreite: sB oder zB, also $80GB/s$ oder $40GB/s$
- max hops: $s + z - 2$, also 22
- für b):$(s + z - 2) \cdot 5ns = 110ns$
- für c): $z(2s - 1) - s = 232$

Für den Torus:

- $z = 8$ Anzahl Zeilen
- $s = 16$ Anzahl Spalten
- Bandbreite: $2zsB = 1280GB/s$
- Bisektions-Bandbreite: $2sB$ oder $2zB$, also $160GB/s$ oder $80GB/s$
- max hops: $max(s, z) - 1$, also 15
- für b):$(max(s, z - 1)) \cdot 5ns = 75ns$
- für c): $2sz = 256$

Aufgabe 2-5 (13 Punkte)

Die Multiplikation einer $n \times n$ Matrix A mit einem n-elementigen Vektor b liefert einen n-elementigen Vektor c und ist definiert durch:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} a_{1i} b_i \\ \sum_{i=0}^{n} a_{2i} b_i \\ \vdots \\ \sum_{i=0}^{n} a_{mi} b_i \end{bmatrix}$$

Schreiben Sie ein C-Programm, das eine Matrix und einen Vektor mit zufälligen `float`-Werten initialisiert und sie miteinander multipliziert. Wählen Sie n zwischen 1024 und 8192.

Implementieren Sie das Programm einmal so, dass auf die Matrix zeilenweise zugegriffen wird und einmal so, dass auf die Matrix spaltenweise zugegriffen wird. Vergleichen Sie die Zeiten, welche die Berechnung jeweils braucht! Begründen Sie die Unterschiede in der Laufzeit!

Hinweis: Sie können `rand()` aus `stdlib.h` zur Generierung von Zufallszahlen und `clock()` aus `time.h` zur Zeitmessung verwenden.

Musterlösung

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void MatrixVectorMultRowsFirst(double *A, double *b, double *c, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      double prod = A[i * n + j] * b[j];
      c[i] += prod;
    }
  }
}
void MatrixVectorMultColsFirst(double *A, double *b, double *c, int n) {
  for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++) {
      double prod = A[i * n + j] * b[j];
      c[i] += prod;
    }
  }
}

double *A = NULL;
double *b = NULL;
double *c = NULL;

void init(int n) {
  if (A)
    free(A);
  if (b)
    free(b);
  if (c)
    free(c);
  A = (double *)malloc(n * n * sizeof(double));
  b = (double *)malloc(n * sizeof(double));
  c = (double *)malloc(n * sizeof(double));
  for (int i = 0; i < n; i++) {
    b[i] = rand();
    for (int j = 0; j < n; j++) {
      A[i * n + j] = rand();
    }
  }
}

int main(int argc, char *argv[]) {
  long t1, t2;
  int n = 1024;
  double elapsed_time;
  init(n);
  t1 = clock();
  for (int k = 0; k < 10; k++) {
    MatrixVectorMultRowsFirst(A, b, c, n);
  }
  t2 = clock();
  elapsed_time = (double)(t2 - t1) / CLOCKS_PER_SEC;
  printf("Rows first, n: %d, time: %lf seconds \n", n, elapsed_time);

  init(n);
```

```
  t1 = clock();
  for (int k = 0; k < 10; k++) {
    MatrixVectorMultColsFirst(A, b, c, n);
  }
  t2 = clock();
  elapsed_time = (double)(t2 - t1) / CLOCKS_PER_SEC;
  printf("Columns first, n: %d, time: %lf seconds \n", n, elapsed_time);

  n = 8192;
  init(n);
  t1 = clock();
  for (int k = 0; k < 10; k++) {
    MatrixVectorMultRowsFirst(A, b, c, n);
  }
  t2 = clock();
  elapsed_time = (double)(t2 - t1) / CLOCKS_PER_SEC;
  printf("Rows first, n: %d, time: %lf seconds \n", n, elapsed_time);

  init(n);
  t1 = clock();
  for (int k = 0; k < 10; k++) {
    MatrixVectorMultColsFirst(A, b, c, n);
  }
  t2 = clock();
  elapsed_time = (double)(t2 - t1) / CLOCKS_PER_SEC;
  printf("Columns first, n: %d, time: %lf seconds \n", n, elapsed_time);

  free(A);
  free(b);
  free(c);
}
```

Die Berechnung mit spaltenweisen Zugriffen ist langsamer als die Berechnung mit zeilenweisen Zugriffen. Der Grund dafür ist die Anordnung der Daten im Speicher. Die Daten werden dort zeilenweise abgelegt. Wenn auf ein neues Element zugegriffen wird, welches noch nicht im Cache liegt, werden die Daten einer gesamten Cache-Line in den Cache geladen. Je nach Cache-Line Größe (zwischen 32 und 128 Byte) sind zwischen 8 und 32 float, Bzw. zwischen 4 und 16 double Werte. Auf die nächsten Elemente dieser Cache-Line kann dann ohne einen weiteren Cache-Miss zugegriffen werden. Werden die Daten spaltenweise gelesen, wird auf die Daten im Speicher nicht sequentiell zugegriffen. Das bedeutet, dass jeder Zugriff auf ein Element einen neuen Cache-Miss erzeugt. Dadurch wird das Programm deutlich langsamer.